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.

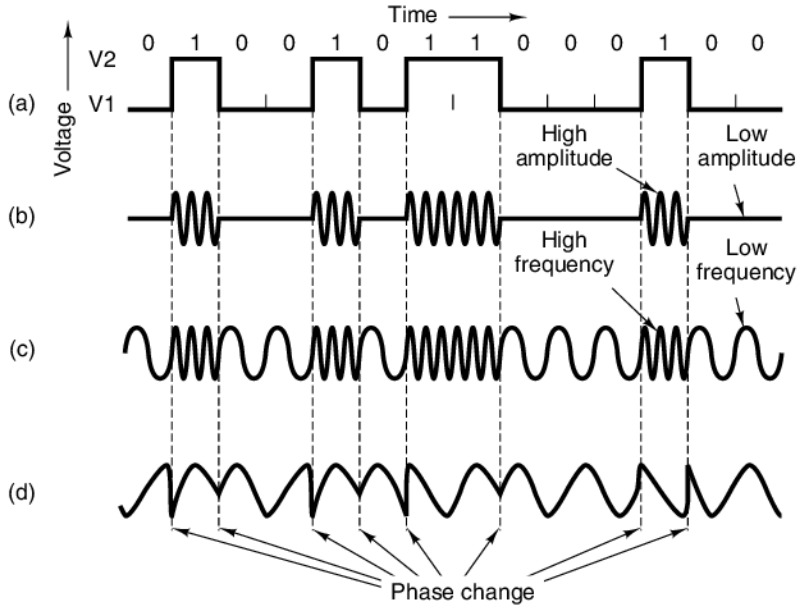


Figure 2-38. Transmission of the binary number 01001011000100 over a telephone line bit by bit. (a) Two-level signal. (b) Amplitude modulation. (c) Frequency modulation. (d) Phase modulation.

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.

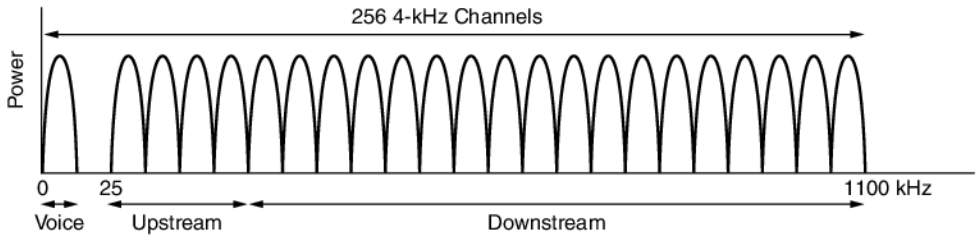


Figure 2-39. Operation of ADSL.

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.

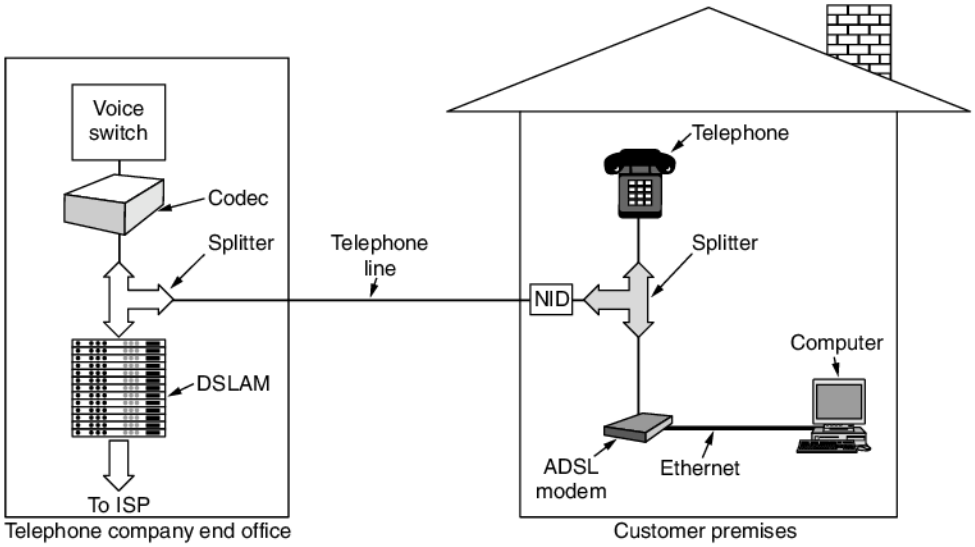


Figure 2-40. A typical ADSL equipment configuration.

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.

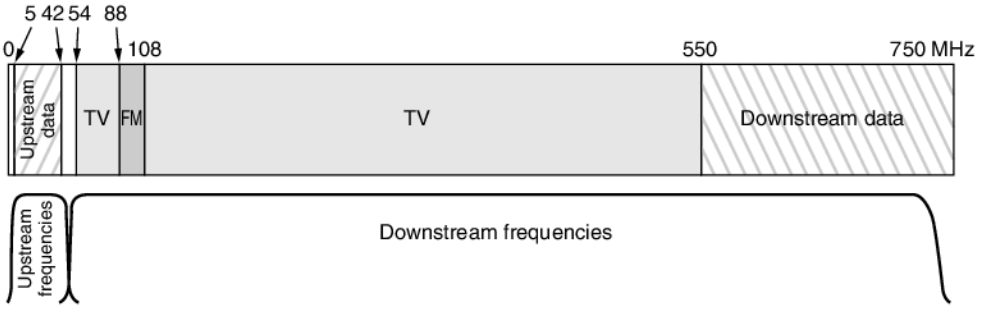


Figure 2-41. Frequency allocation in a typical cable TV system used for Internet access.

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

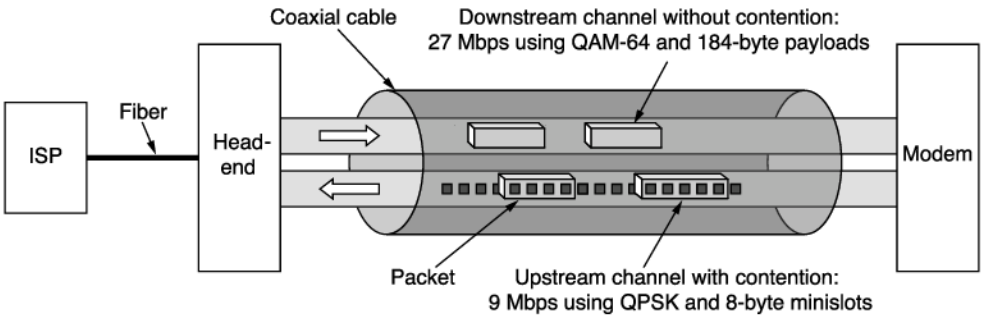


Figure 2-42. Typical details of the upstream and downstream channels in North America. QAM-64 (Quadrature Amplitude Modulation) allows 6 bits/Hz but works only at high frequencies. QPSK (Quadrature Phase Shift Keying) works at low frequencies but allows only 2 bits/Hz.

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

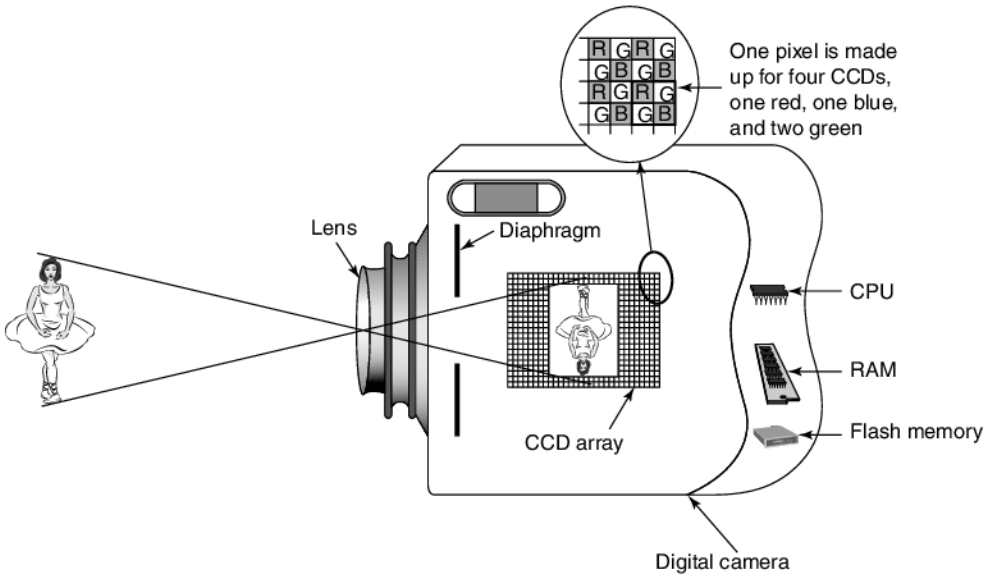


Figure 2-43. A digital camera.

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).

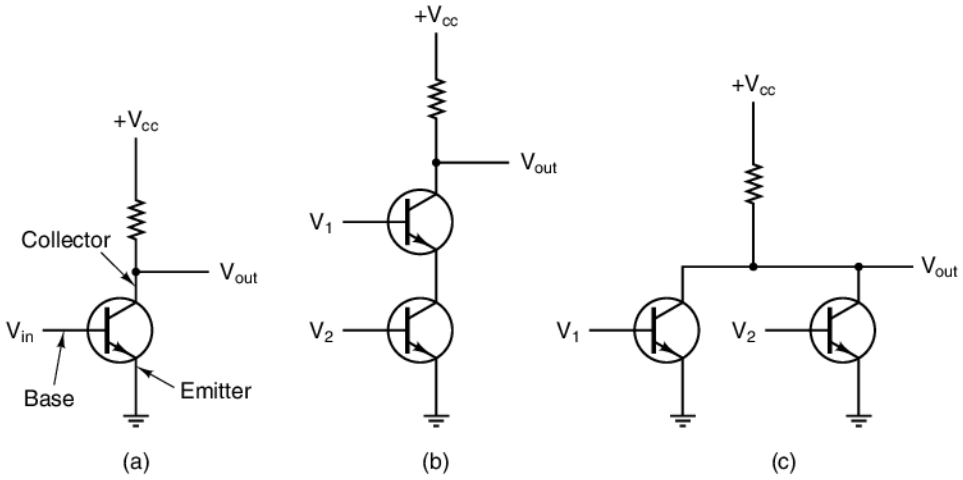


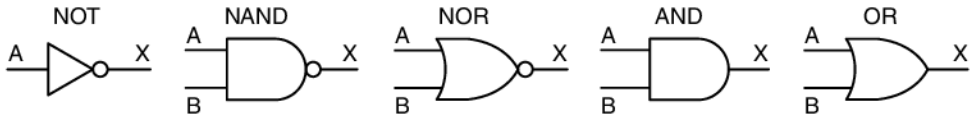
Figure 3-1. (a) A transistor inverter. (b) A NAND gate. (c) A NOR gate.

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).

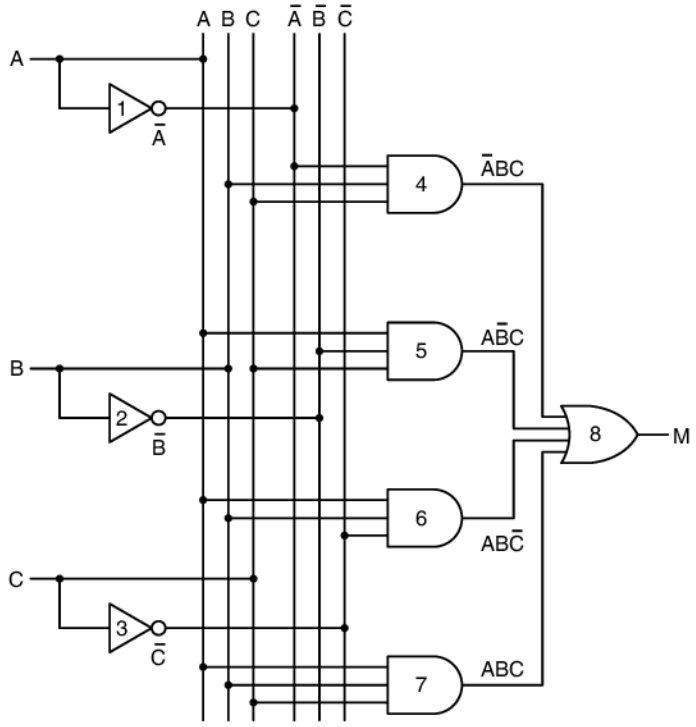
Because a Boolean function of n variables has only 2^n possible combinations of input values, the function can be completely described by giving a table with 2^n rows, each row telling the value of the function for a different combination of input values. Such a table is called a **truth table**. The 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

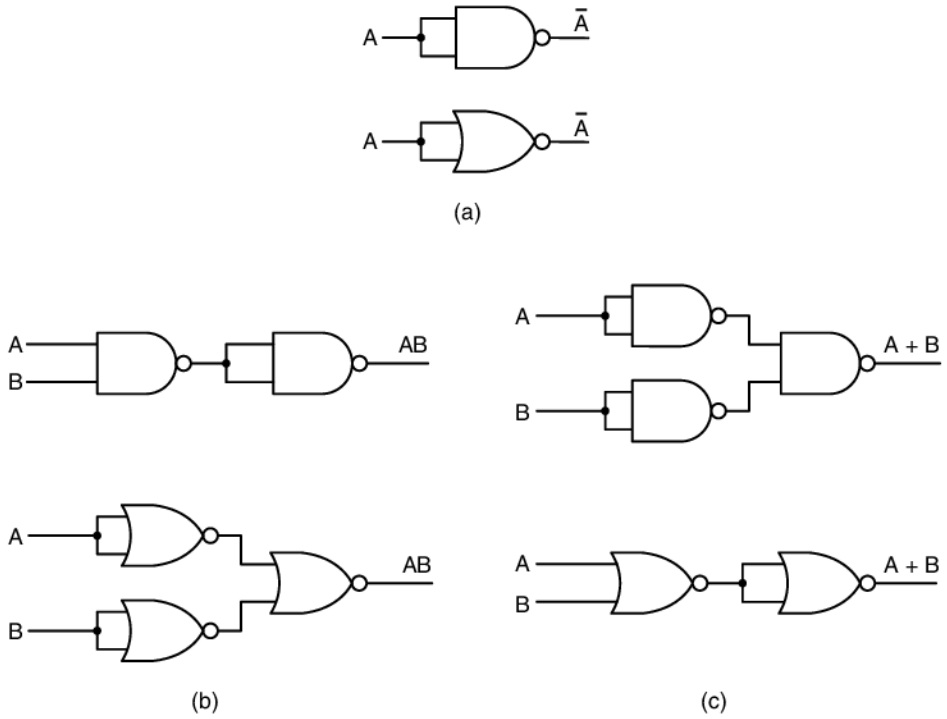


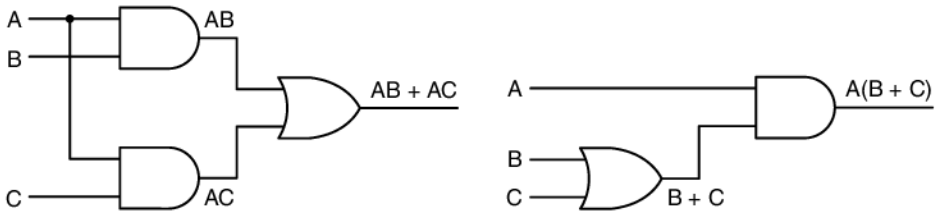
Figure 3-4. Construction of (a) NOT, (b) AND, and (c) OR gates using only NAND gates or only NOR gates.

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.

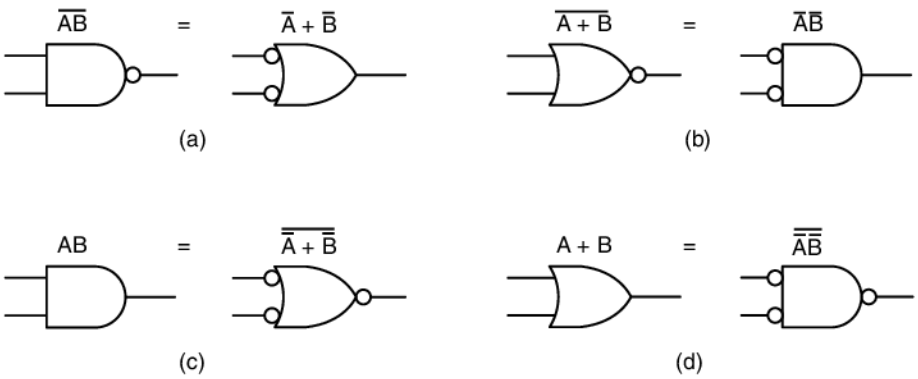
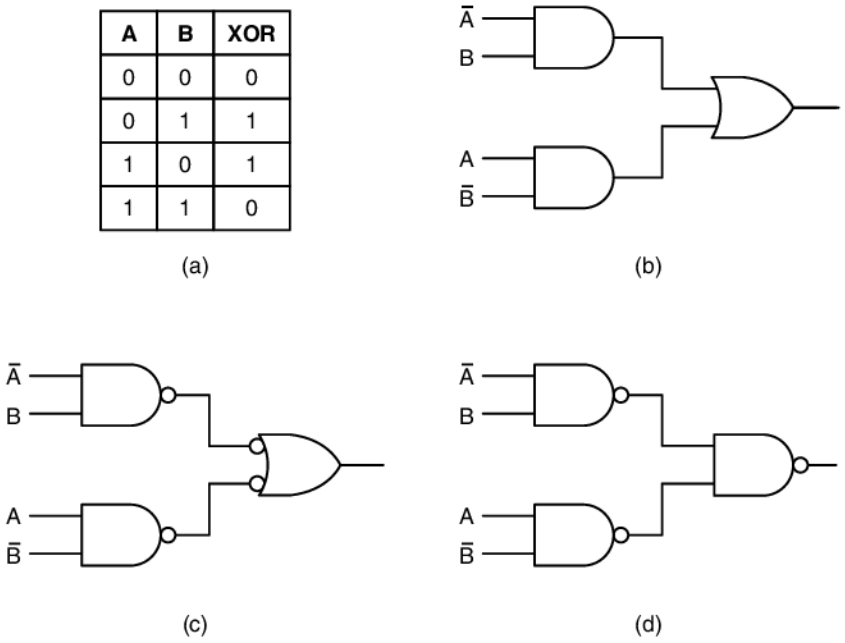


Figure 3-7. Alternative symbols for some gates: (a) NAND (b) NOR (c) AND (d) OR.

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.

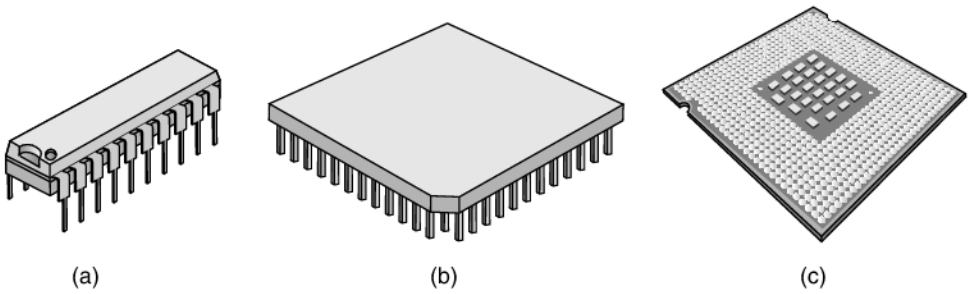


Figure 3-10. Common types of integrated-circuit packages, including a dual-in-line package (a), pin grid array (b), and land grid array (c).

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.

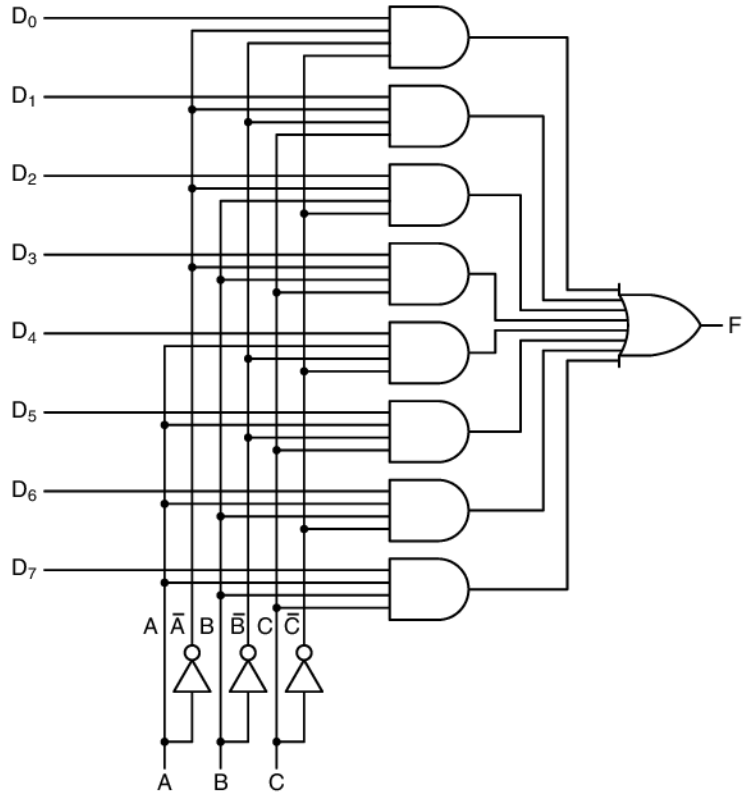


Figure 3-11. An eight-input multiplexer circuit.

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

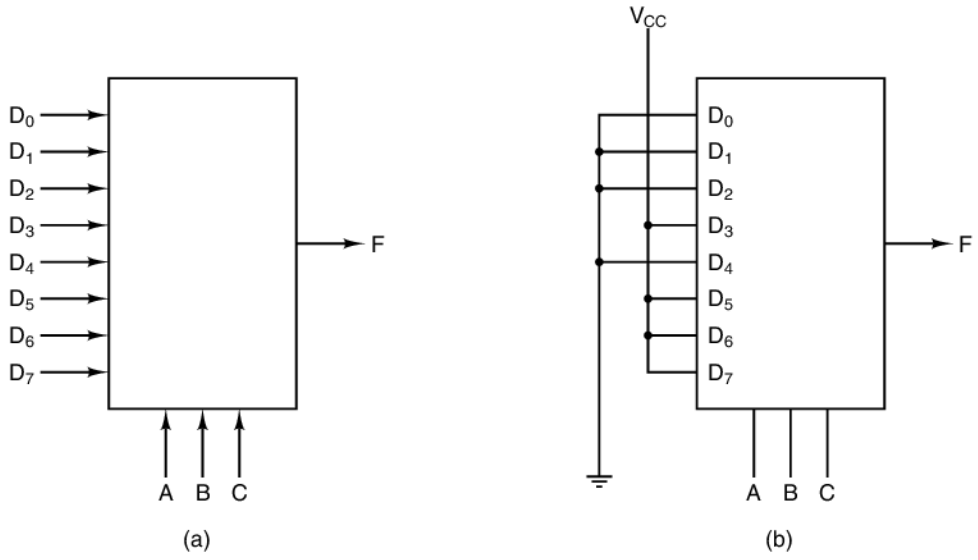


Figure 3-12. (a) An eight-input multiplexer. (b) The same multiplexer wired to compute the majority function.

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.

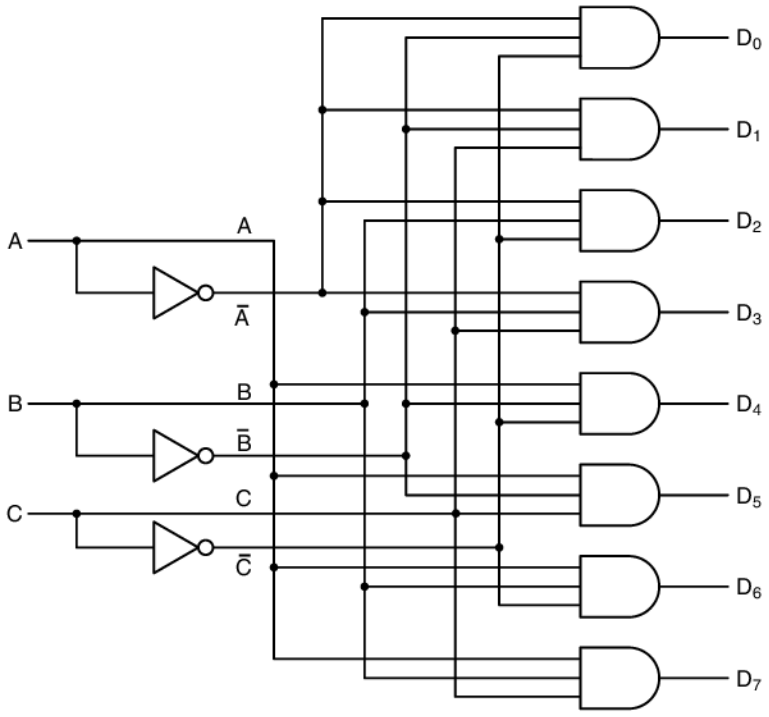


Figure 3-13. A three-to-eight decoder circuit.

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.

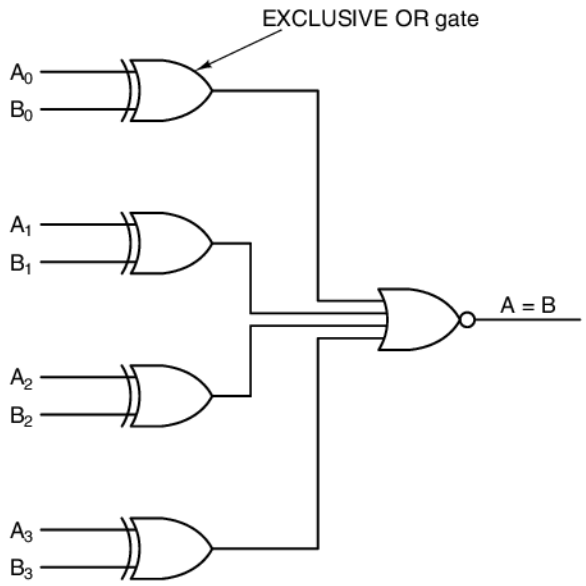


Figure 3-14. A simple 4-bit comparator.

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.

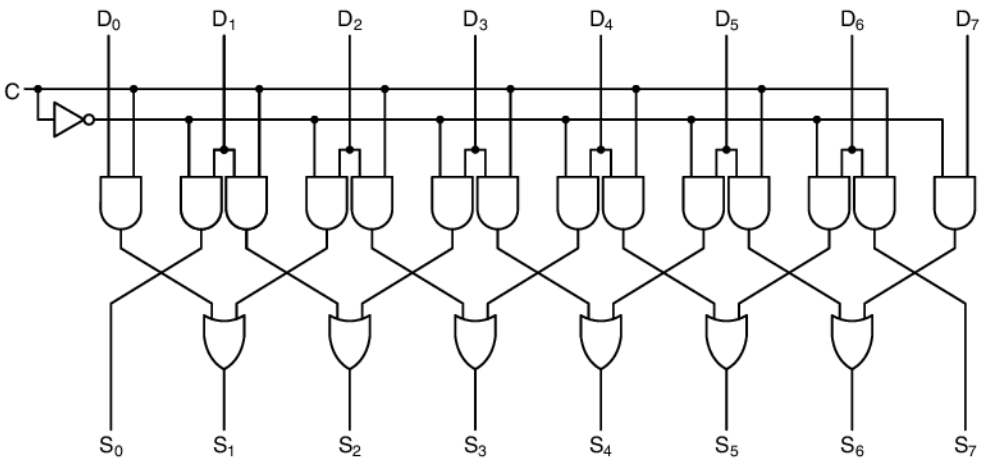


Figure 3-15. A 1-bit left/right shifter.

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

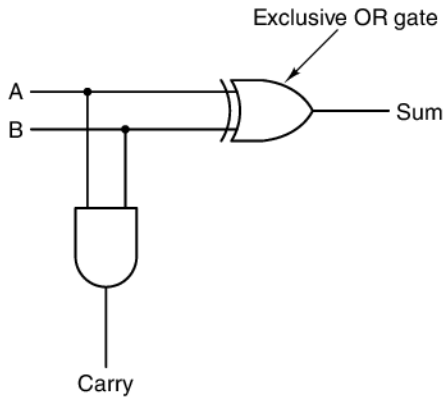
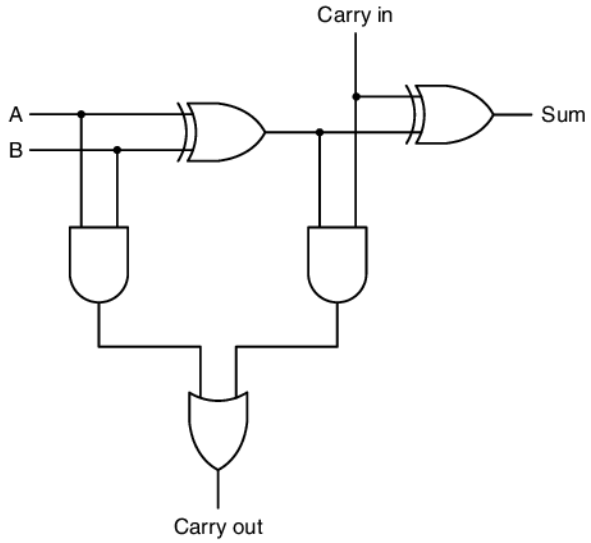


Figure 3-16. (a) Truth table for 1-bit addition. (b) A circuit for a half adder.

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 $A + B + 1$.

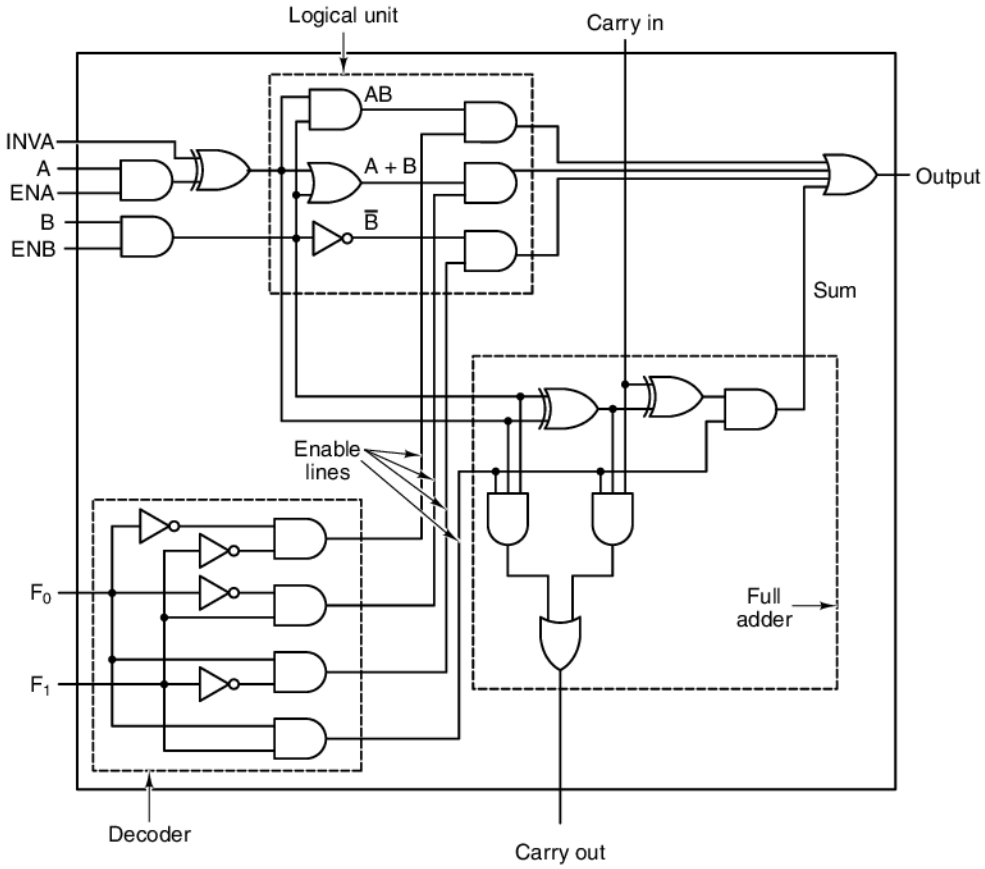


Figure 3-18. A 1-bit ALU.

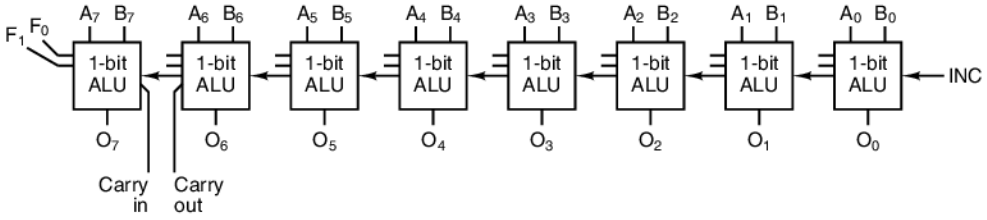


Figure 3-19. Eight 1-bit ALU slices connected to make an 8-bit ALU. The enables and invert signals are not shown for simplicity.

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.

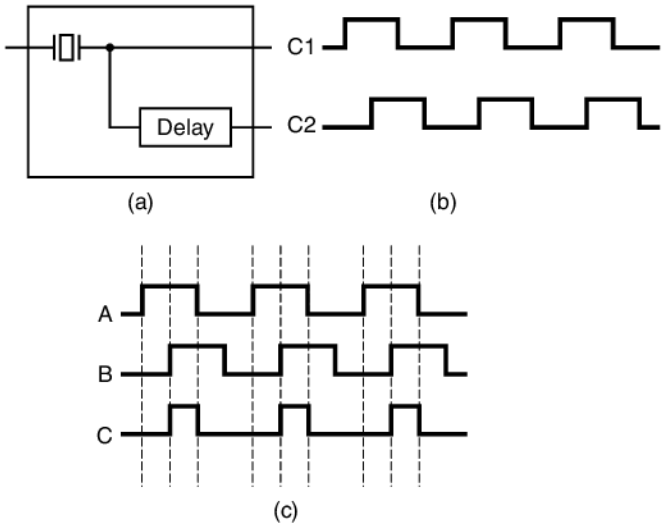


Figure 3-20. (a) A clock. (b) The timing diagram for the clock. (c) Generation of an asymmetric clock.

3.3 MEMORY

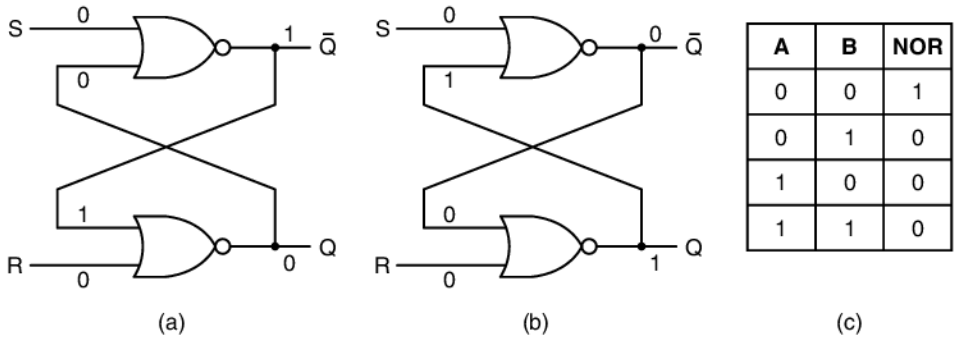
An essential component of every computer is its memory. Without memory there could be no computers as we now know them. Memory is used for storing both instructions to be executed and data. In the following sections we will examine the basic components of a memory system starting at the gate level to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

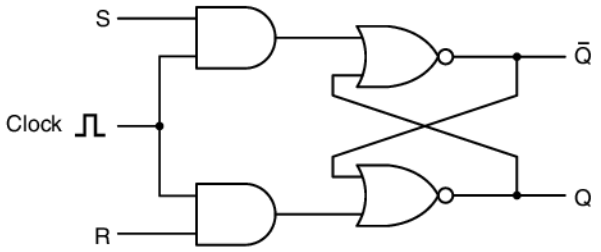


Figure 3-22. A clocked SR latch.

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.

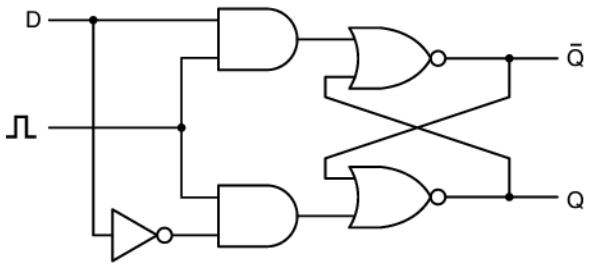


Figure 3-23. A clocked D latch.

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

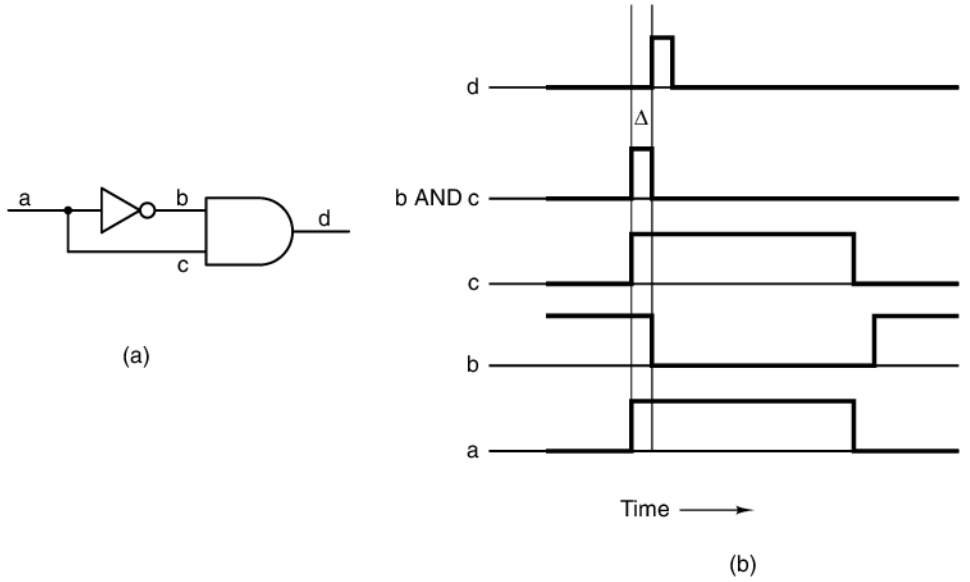


Figure 3-24. (a) A pulse generator. (b) Timing at four points in the circuit.

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.

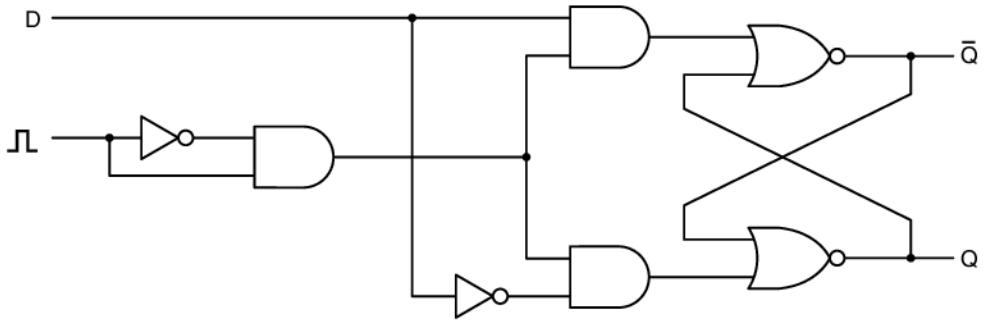


Figure 3-25. A D flip-flop.

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 $Q = 0$).

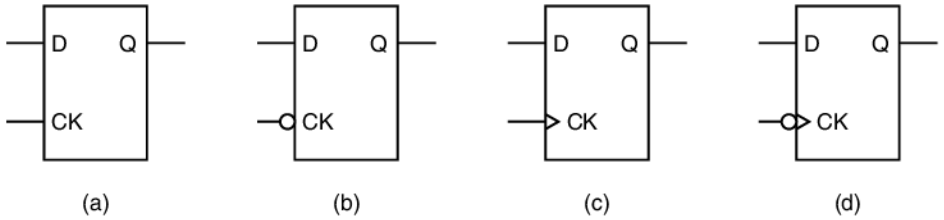


Figure 3-26. D latches and flip-flops.

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

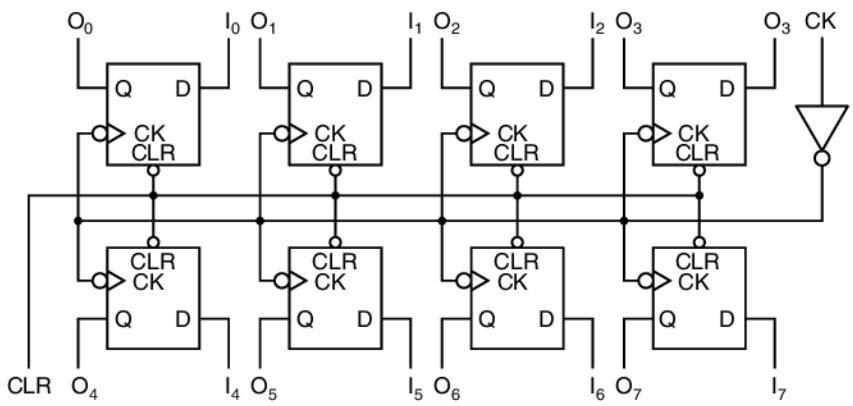


Figure 3-27. An 8-bit register constructed from single-bit flip-flops.

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,

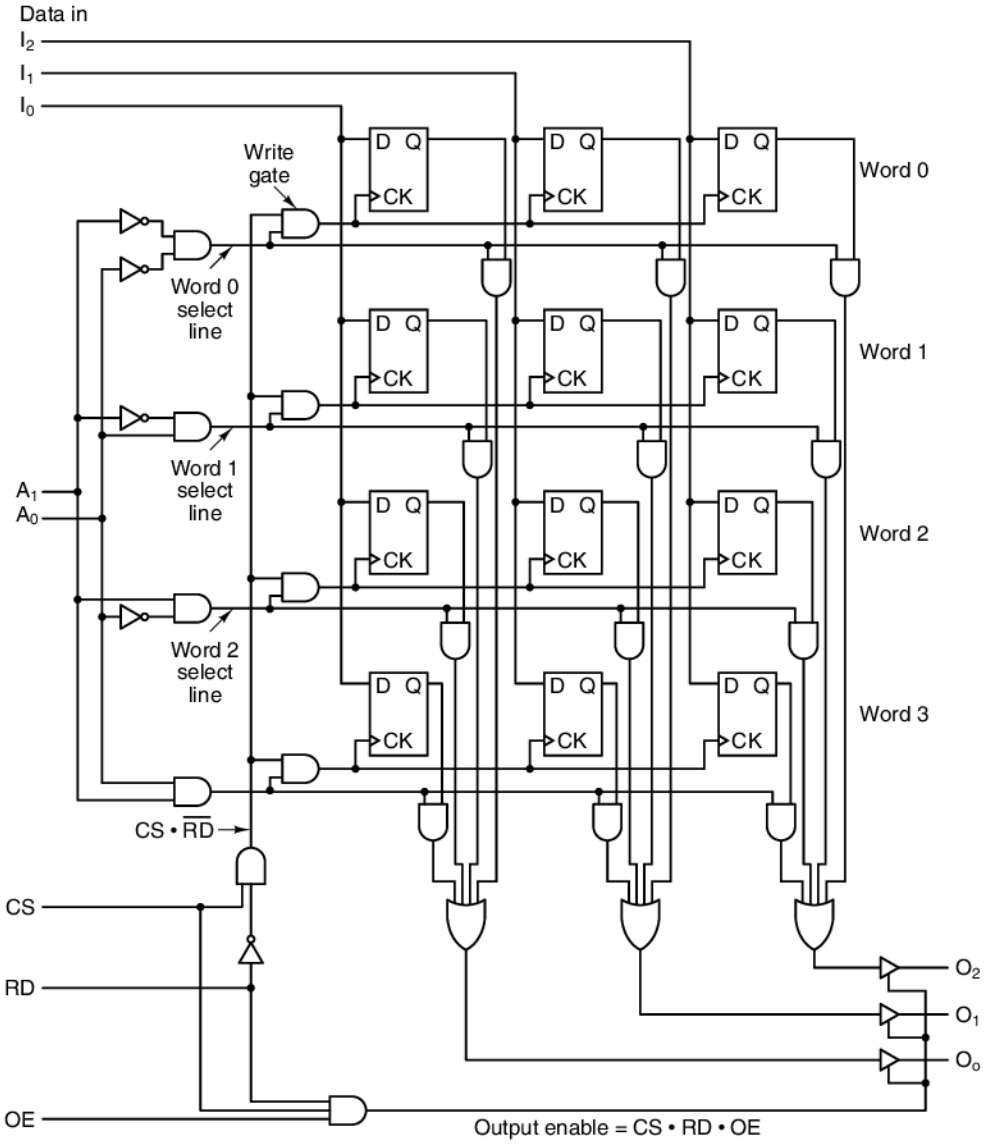


Figure 3-28. Logic diagram for a 4 × 3 memory. Each row is one of the four 3-bit words. A read or write operation always reads or writes a complete word.

enabling one of the four write gates, depending on which word select line is high. The output of the write gate drives all the CK signals for the selected word, loading the input data into the flip-flops for that word. A write is done only if CS is high and RD is low, and even then only the word selected by A_0 and A_1 is written; the other words are not changed at all.

Read is similar to write. The address decoding is exactly the same as for write. But now the CS · $\overline{\text{RD}}$ line is low, so all the write gates are disabled and none of the flip-flops is modified. Instead, the word select line that is chosen enables the AND gates tied to the Q bits of the selected word. Thus, the selected word outputs its data into the four-input OR gates at the bottom of the figure, while the other three words output 0s. Consequently, the output of the OR gates is identical to the value stored in the word selected. The three words not selected make no contribution to the output.

Although we could have designed a circuit in which the three OR gates were just fed into the three output data lines, doing so sometimes causes problems. In particular, we have shown the data input lines and the data output lines as being different, but in actual memories the same lines are used. If we had tied the OR gates to the data output lines, the chip would try to output data, that is, force each line to a specific value, even on writes, thus interfering with the input data. For this reason, it is desirable to have a way to connect the OR gates to the data output lines on reads but disconnect them completely on writes. What we need is an electronic switch that can make or break a connection in a fraction of a nanosecond.

Fortunately, such switches exist. Figure 3-29(a) shows the symbol for what is called a **noninverting buffer**. It has a data input, a data output, and a control input. When the control input is high, the buffer acts like a wire, as shown in Fig. 3-29(b). When the control input is low, the buffer acts like an open circuit, as shown in Fig. 3-29(c); it is as though someone detached the data output from the rest of the circuit with a wirecutter. However, in contrast to the wirecutter analogy the connection can be subsequently restored in a fraction of a nanosecond by just making the control signal high again.

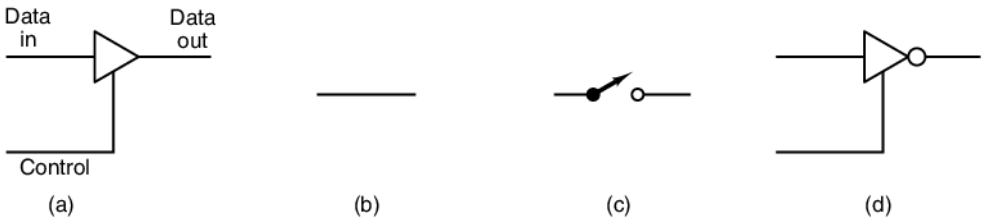


Figure 3-29. (a) A noninverting buffer. (b) Effect of (a) when control is high. (c) Effect of (a) when control is low. (d) An inverting buffer.

Figure 3-29(d) shows an **inverting buffer**, which acts like a normal inverter when control is high and disconnects the output from the circuit when control is low. Both kinds of buffers are **tri-state devices**, because they can output 0, 1, or none of the above (open circuit). Buffers also amplify signals, so they can drive many inputs simultaneously. They are sometimes used in circuits for this reason, even when their switching properties are not needed.

Getting back to the memory circuit, it should now be clear what the three noninverting buffers on the data output lines are for. When CS, RD, and OE are all high, the output enable signal is also high, enabling the buffers and putting a word onto the output lines. When any one of CS, RD, or OE is low, the data outputs are disconnected from the rest of the circuit.

3.3.5 Memory Chips

The nice thing about the memory of Fig. 3-28 is that it extends easily to larger sizes. As we drew it, the memory is 4×3, that is, four words of 3 bits each. To extend it to 4×8 we need only add five more columns of four flip-flops each, as well as five more input lines and five more output lines. To go from 4×3 to 8×3 we must add four more rows of three flip-flops each, as well as an address line A_2. With this kind of structure, the number of words in the memory should be a power of 2 for maximum efficiency, but the number of bits in a word can be anything.

Because integrated-circuit technology is well suited to making chips whose internal structure is a repetitive two-dimensional pattern, memory chips are an ideal application for it. As the technology improves, the number of bits that can be put on a chip keeps increasing, typically by a factor of two every 18 months (Moore's law). The larger chips do not always render the smaller ones obsolete due to different trade-offs in capacity, speed, power, price, and interfacing convenience. Commonly, the largest chips currently available sell at a premium and thus are more expensive per bit than older, smaller ones.

For any given memory size, there are various ways of organizing the chip. Figure 3-30 shows two possible organizations for an older memory chip of size 4 Mbit: $512\text{K} \times 8$ and $4096\text{K} \times 1$. (As an aside, memory-chip sizes are usually quoted in bits, rather than in bytes, so we will stick to that convention here.) In Fig. 3-30(a), 19 address lines are needed to address one of the 2^{19} bytes, and eight data lines are needed for loading or storing the byte selected.

A note on terminology is in order here. On some pins, the high voltage causes an action to happen. On others, the low voltage causes the action. To avoid confusion, we will consistently say that a signal is **asserted** (rather than saying it goes high or goes low) to mean that it is set to cause some action. Thus, for some pins, asserting it means setting it high. For others, it means setting the pin low. Pins that are asserted low are given signal names containing an overbar. Thus, a signal named CS is asserted high, but one named $\overline{\text{CS}}$ is asserted low. The opposite of asserted is **negated**. When nothing special is happening, pins are negated.

Now let us get back to our memory chip. Since a computer normally has many memory chips, a signal is needed to select the chip that is currently needed so that it responds and all the others do not. The $\overline{\text{CS}}$ (Chip Select) signal is provided for this purpose. It is asserted to enable the chip. Also, a way is needed to distinguish reads from writes. The $\overline{\text{WE}}$ signal (Write Enable) is used to indicate that data are

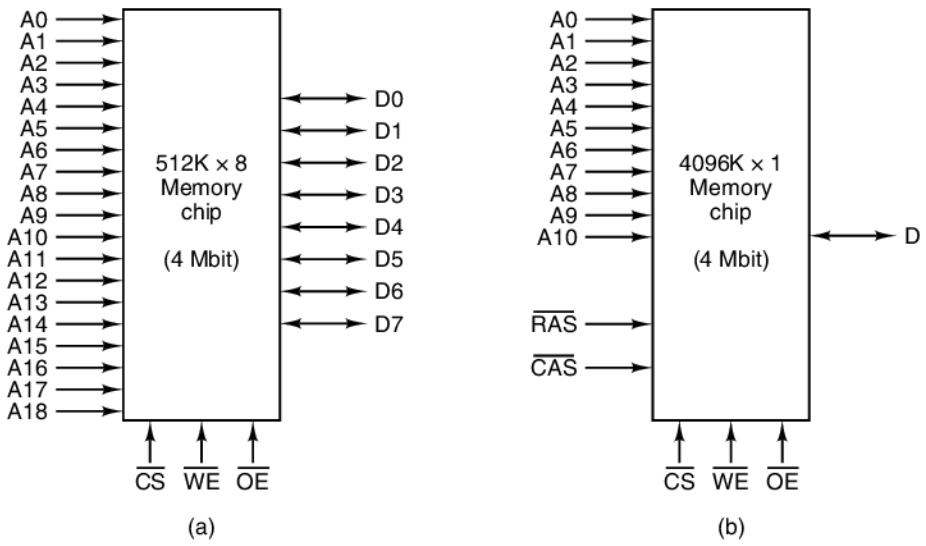


Figure 3-30. Two ways of organizing a 4-Mbit memory chip.

being written rather than being read. Finally, the $\overline{\text{OE}}$ (Output Enable) signal is asserted to drive the output signals. When it is not asserted, the chip output is disconnected from the circuit.

In Fig. 3-30(b), a different addressing scheme is used. Internally, this chip is organized as a 2048 × 2048 matrix of 1-bit cells, which gives 4 Mbits. To address the chip, first a row is selected by putting its 11-bit number on the address pins. Then the $\overline{\text{RAS}}$ (Row Address Strobe) is asserted. After that, a column number is put on the address pins and $\overline{\text{CAS}}$ (Column Address Strobe) is asserted. The chip responds by accepting or outputting one data bit.

Large memory chips are often constructed as $n \times n$ matrices that are addressed by row and column. This organization reduces the number of pins required but also makes addressing the chip slower, since two addressing cycles are needed, one for the row and one for the column. To win back some of the speed lost by this design, some memory chips can be given a row address followed by a sequence of column addresses to access consecutive bits in a row.

Years ago, the largest memory chips were often organized like Fig. 3-30(b). As memory words have grown from 8 bits to 32 bits and beyond, 1-bit-wide chips began to be inconvenient. To build a memory with a 32-bit word from 4096K × 1 chips requires 32 chips in parallel. These 32 chips have a total capacity of at least 16 MB, whereas using 512K × 8 chips requires only four chips in parallel and allows memories as small as 2 MB. To avoid having 32 chips for memory, most chip manufacturers now have chip families with 4-, 8-, and 16-bit widths. And the situation with 64-bit words is even worse, of course.

Two examples of 512-Mbit chips are given in Fig. 3-31. These chips have four internal memory banks of 128 Mbit each, requiring two bank select lines to choose a bank. The design of Fig. 3-31(a) is a 32M × 16 design, with 13 lines for the $\overline{\text{RAS}}$ signal, 10 lines for the $\overline{\text{CAS}}$ signal, and 2 lines for the bank select. Together, these 25 signals allow each of the 2^{25} internal 16-bit cells to be addressed. In contrast, Fig. 3-31(b) is a 128M × 4 design, with 13 lines for the $\overline{\text{RAS}}$ signal, 12 lines for the $\overline{\text{CAS}}$ signal, and 2 lines for the bank select. Here, 27 signals can select any of the 2^{27} internal 4-bit cells to be addressed. The decision about how many rows and how many columns a chip has is made for engineering reasons. The matrix need not be square.

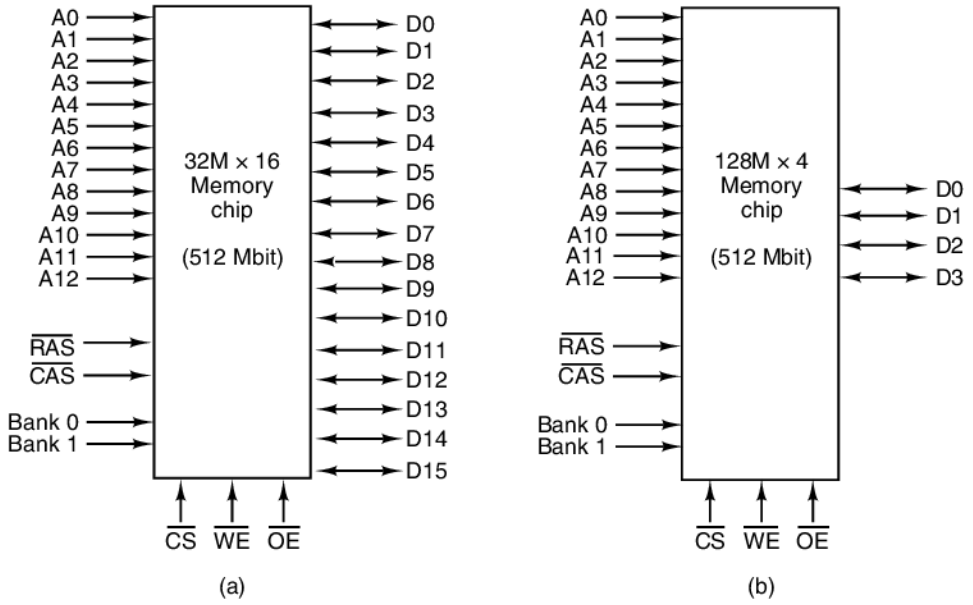


Figure 3-31. Two ways of organizing a 512-Mbit memory chip.

These examples demonstrate two separate and independent issues for memory-chip design. First is the output width (in bits): does the chip deliver 1, 4, 8, 16, or some other number of bits at once? Second, are all the address bits presented on separate pins at once or are the rows and columns presented sequentially as in the examples of Fig. 3-31? A memory-chip designer has to answer both questions before starting the chip design.

3.3.6 RAMs and ROMs

The memories we have studied so far can all be read and written. Such memories are called **RAM**s (Random Access Memories), which is a misnomer because all memory chips are randomly accessible, but the term is too well established to

get rid of now. RAMs come in two varieties, static and dynamic. **Static RAM**s (**SRAM**s) are constructed internally using circuits similar to our basic D flip-flop. These memories have the property that their contents are retained as long as the power is kept on: seconds, minutes, hours, even days. Static RAMs are very fast. A typical access time is on the order of a nanosecond or less. For this reason, static RAMS are popular as cache memory.

Dynamic RAMs (**DRAM**s), in contrast, do not use flip-flops. Instead, a dynamic RAM is an array of cells, each cell containing one transistor and a tiny capacitor. The capacitors can be charged or discharged, allowing 0s and 1s to be stored. Because the electric charge tends to leak out, each bit in a dynamic RAM must be **refreshed** (reloaded) every few milliseconds to prevent the data from leaking away. Because external logic must take care of the refreshing, dynamic RAMs require more complex interfacing than static ones, although in many applications this disadvantage is compensated for by their larger capacities.

Since dynamic RAMs need only one transistor and one capacitor per bit (vs. six transistors per bit for the best static RAM), dynamic RAMs have a very high density (many bits per chip). For this reason, main memories are nearly always built out of dynamic RAMs. However, this large capacity has a price: dynamic RAMs are slow (tens of nanoseconds). Thus, the combination of a static RAM cache and a dynamic RAM main memory attempts to combine the good properties of each.

Several types of dynamic RAM chips exist. The oldest type still around (in elderly computers) is **FPM** (**Fast Page Mode**) DRAM. Internally it is organized as a matrix of bits and it works by having the hardware present a row address and then step through the column addresses, as we described with $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ in the context of Fig. 3-30. Explicit signals tell the memory when it is time to respond, so the memory runs asynchronously from the main system clock.

FPM DRAM was replaced with **EDO** (**Extended Data Output**) DRAM, which allows a second memory reference to begin before the previous memory reference has been completed. This simple pipelining did not make a single memory reference go faster but did improve the memory bandwidth, giving more words per second.

FPM and EDO worked reasonably well when memory chips had cycle times of 12 nsec and slower. When processors got so fast that faster memories were really needed, FPM and EDO were replaced by **SDRAM** (**Synchronous DRAM**), which is a hybrid of static and dynamic RAM and is driven by the main system clock. The big advantage of SDRAM is that the clock eliminates the need for control signals to tell the memory chip when to respond. Instead, the CPU tells the memory how many cycles it should run, then starts it. On each subsequent cycle, the memory outputs 4, 8, or 16 bits, depending on how many output lines it has. Eliminating the need for control signals increases the data rate between CPU and memory.

The next improvement over SDRAM was **DDR** (**Double Data Rate**) SDRAM. With this kind of memory, the memory chip produces output on both the rising

edge of the clock and the falling edge, doubling the data rate. Thus, an 8-bit-wide DDR chip running at 200 MHz outputs two 8-bit values 200 million times a second (for a short interval, of course), giving a theoretical burst rate of 3.2 Gbps. The DDR2 and DDR3 memory interfaces provide additional performance over DDR by increasing the memory-bus speeds to 533 MHz and 1067 MHz, respectively. At the time this book went to press, the fastest DDR3 chips could output data at 17.067 GB/sec.

Nonvolatile Memory Chips

RAMs are not the only kind of memory chips. In many applications, such as toys, appliances, and cars, the program and some of the data must remain stored even when the power is turned off. Furthermore, once installed, neither the program nor the data are ever changed. These requirements have led to the development of **ROM**s (Read-Only Memories), which cannot be changed or erased, intentionally or otherwise. The data in a ROM are inserted during its manufacture, essentially by exposing a photosensitive material through a mask containing the desired bit pattern and then etching away the exposed (or unexposed) surface. The only way to change the program in a ROM is to replace the entire chip.

ROMs are much cheaper than RAMs when ordered in large enough volumes to defray the cost of making the mask. However, they are inflexible, because they cannot be changed after manufacture, and the turnaround time between placing an order and receiving the ROMs may be weeks. To make it easier for companies to develop new ROM-based products, the **PROM** (Programmable ROM) was invented. A PROM is like a ROM, except that it can be programmed (once) in the field, eliminating the turnaround time. Many PROMs contain an array of tiny fuses inside. A specific fuse can be blown out by selecting its row and column and then applying a high voltage to a special pin on the chip.

The next development in this line was the **EPROM** (Erasable PROM), which can be not only field-programmed but also field-erased. When the quartz window in an EPROM is exposed to a strong ultraviolet light for 15 minutes, all the bits are set to 1. If many changes are expected during the design cycle, EPROMs are far more economical than PROMs because they can be reused. EPROMs usually have the same organization as static RAMs. The 4-Mbit 27C040 EPROM, for example, uses the organization of Fig. 3-31(a), which is typical of a static RAM. What is interesting is that ancient chips like this one do not die off. They just become cheaper and find their way into lower-end products that are highly cost sensitive. A 27C040 can now be bought retail for under $3 and much less in large volumes.

Even better than the EPROM is the **EEPROM** which can be erased by applying pulses to it instead of putting it in a special chamber for exposure to ultraviolet light. In addition, an EEPROM can be reprogrammed in place, whereas an EPROM has to be inserted in a special EPROM programming device to be programmed. On the minus side, the biggest EEPROMs are typically only 1/64 as

large as common EPROMs and they are only half as fast. EEPROMs cannot compete with DRAMs or SRAMs because they are 10 times slower, 100 times smaller in capacity, and much more expensive. They are used only in situations where their nonvolatility is crucial.

A more recent kind of EEPROM is **flash memory**. Unlike EPROM, which is erased by exposure to ultraviolet light, and EEPROM, which is byte erasable, flash memory is block erasable and rewritable. Like EEPROM, flash memory can be erased without removing it from the circuit. Various manufacturers produce small printed-circuit cards with up to 64 GB of flash memory on them for use as "film" for storing pictures in digital cameras and many other purposes. As we discussed in Chap. 2, flash memory is now starting to replace mechanical disks. As a disk, flash memory provides faster access times at lower power, but with a much higher cost per bit. A summary of the various kinds of memory is given in Fig. 3-32.

Type	Category	Erasure	Byte alterable	Volatile	Typical use
SRAM	Read/write	Electrical	Yes	Yes	Level 2 cache
DRAM	Read/write	Electrical	Yes	Yes	Main memory (old)
SDRAM	Read/write	Electrical	Yes	Yes	Main memory (new)
ROM	Read-only	Not possible	No	No	Large-volume appliances
PROM	Read-only	Not possible	No	No	Small-volume equipment
EPROM	Read-mostly	UV light	No	No	Device prototyping
EEPROM	Read-mostly	Electrical	Yes	No	Device prototyping
Flash	Read/write	Electrical	No	No	Film for digital camera

Figure 3-32. A comparison of various memory types.

Field-Programmable Gate Arrays

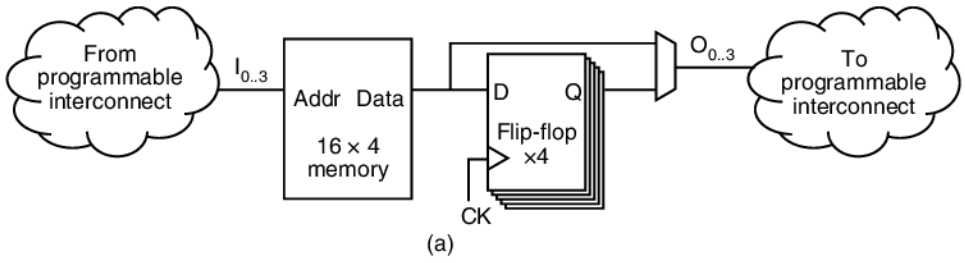
As we saw in Chap. 1, **field-programmable gate arrays** (**FPGA**s) are chips which contain programmable logic such that we can form arbitrary logic circuit by simply loading the FPGA with appropriate configuration data. The main advantage of FPGAs is that new hardware circuits can be built in hours, rather than the months it takes to fabricate ICs. Integrated circuits are not going the way of the dodo, however, as they still hold a significant cost advantage over FPGAs for high-volume applications, and they run faster and use much less power. Because of their design-time advantages, however, FPGAs are often used for design prototyping and low-volume applications.

Let's now look inside an FPGA and understand how it can be used to implement a wide range of logic circuits. The FPGA chip contains two primary components that are replicated many times: **LUT**s (**LookUp Tables**) and **programmable interconnects**. Let us now examine how they are used.

A LUT, shown in Fig. 3-33(a), is a small programmable memory that produces a signal output optionally to a register, which is then output to the programmable interconnect. The programmable memory is used to create an arbitrary logic function. The LUT in the figure has a 16×4 memory, which can emulate any logic circuit with 4 bits of input and 4 bits of output. Programming the LUT requires loading the memory with the appropriate responses of the combinational logic being emulated. In other words, if the combinational logic produces the value Y when given the input X, the value Y would be written into the LUT at index X.

The example design in Fig. 3-32(b) shows how a single 4-input LUT could implement a 3-bit counter with reset. The example counter continually counts up by adding one (modulo 4) to the current value of the counter, unless the reset signal *CLR* is asserted, in which case the counter resets its value to zero.

To implement the example counter, the upper four entries of the LUT are all zero. These entries output the value zero when the counter is reset. Thus, the most significant bit of the LUT input (I_3) represents the reset input (*CLR*) which is asserted with a logic 1. For the remaining LUT entries, the value at index $I_{0..3}$ of the LUT contains the value $(I + 1)$ modulo 4. To complete the design, the output signal $O_{0..3}$ must be connected, using the programmable interconnect to the internal input signal $I_{0..3}$.



(a)

Signal Assignment

FPGA	Counter
I_3	CLR
$O_{2..0}$	$O_{2..0}$
CK	CK

Addr	Data
0	1
1	2
2	3
3	0

Addr	Data
4	0
5	0
6	0
7	0

(b)

Figure 3-33. (a) A field-programmable logic array lookup table (LUT). (b) The LUT configuration to create a 3-bit clearable counter.

To better understand the FPGA-based counter with reset, let's consider its operation. If, for example, the current state of the counter is 2 and the reset (*CLR*) signal is not asserted, the input address to the LUT will be 2, which will produce an output to the flip-flops of 3. If the reset signal (*CLR*) were asserted for the same state, the input to the LUT would be 6, which would produce the next state of 0.

All in all, this may seem like an arcane way to build a counter with reset, and in fact a fully custom design with an incrementer circuit and reset signals to the flip-flops would be smaller, faster, and use less power. The main advantage of the FPGA-based design is that you can craft it in an hour at home, whereas the more efficient fully custom design must be fabricated from silicon, which could take a month or more.

To use an FPGA, the design must be described using a circuit description or a hardware description language (i.e., a programming language used to describe hardware structures). The design is then processed by a synthesizer, which maps the circuit to a specific FPGA architecture. One challenge of using FPGAs is that the design you want to map never seems to fit. FPGAs are manufactured with varying number of LUTs, with larger quantities costing more. In general, if your design does not fit, you need to simplify or throw away some functionality, or purchase a larger (and more expensive) FPGA. Very large designs may not fit into the largest FPGAs, which will require the designer to map the design into multiple FPGAs; this task is definitely more difficult, but still a walk in the park compared to designing a complete custom integrated circuit.

3.4 CPU CHIPS AND BUSES

Armed with information about integrated circuits, clocks, and memory chips, we can now start to put all the pieces together to look at complete systems. In this section, we will first look at some general aspects of CPUs as viewed from the digital logic level, including **pinout** (what the signals on the various pins mean). Since CPUs are so closely intertwined with the design of the buses they use, we will also provide an introduction to bus design in this section. In subsequent sections we will give detailed examples of both CPUs and their buses and how they are interfaced.

3.4.1 CPU Chips

All modern CPUs are contained on a single chip. This makes their interaction with the rest of the system well defined. Each CPU chip has a set of pins, through which all its communication with the outside world must take place. Some pins output signals from the CPU to the outside world; others accept signals from the outside world; some can do both. By understanding the function of all the pins, we can learn how the CPU interacts with the memory and I/O devices at the digital logic level.

The pins on a CPU chip can be divided into three types: address, data, and control. These pins are connected to similar pins on the memory and I/O chips via a collection of parallel wires called a bus. To fetch an instruction, the CPU first puts the memory address of that instruction on its address pins. Then it asserts one or

more control lines to inform the memory that it wants to read (for example) a word. The memory replies by putting the requested word on the CPU's data pins and asserting a signal saying that it is done. When the CPU sees this signal, it accepts the word and carries out the instruction.

The instruction may require reading or writing data words, in which case the whole process is repeated for each additional word. We will go into the detail of how reading and writing works below. For the time being, the important thing to understand is that the CPU communicates with the memory and I/O devices by presenting signals on its pins and accepting signals on its pins. No other communication is possible.

Two of the key parameters that determine the performance of a CPU are the number of address pins and the number of data pins. A chip with m address pins can address up to 2^m memory locations. Common values of m are 16, 32, and 64. Similarly, a chip with n data pins can read or write an n-bit word in a single operation. Common values of n are 8, 32, and 64. A CPU with 8 data pins will take four operations to read a 32-bit word, whereas one with 32 data pins can do the same job in one operation. Thus, the chip with 32 data pins is much faster but is invariably more expensive as well.

In addition to address and data pins, each CPU has some control pins. They regulate the flow and timing of data to and from the CPU and have other miscellaneous uses. All CPUs have pins for power (usually +1.2 to +1.5 volts), ground, and a clock signal (a square wave at some well-defined frequency), but the other pins vary greatly from chip to chip. Nevertheless, the control pins can be roughly grouped into the following major categories:

1. Bus control.
2. Interrupts.
3. Bus arbitration.
4. Coprocessor signaling.
5. Status.
6. Miscellaneous.

We will briefly describe each of these categories below. When we look at the Intel Core i7, TI OMAP4430, and Atmel ATmega168 chips later, we will provide more detail. A generic CPU chip using these signal groups is shown in Fig. 3-34.

The bus control pins are mostly outputs from the CPU to the bus (thus inputs to the memory and I/O chips) telling whether the CPU wants to read or write memory or do something else. The CPU uses these pins to control the rest of the system and tell it what it wants to do.

The interrupt pins are inputs from I/O devices to the CPU. In most systems, the CPU can tell an I/O device to start an operation and then go off and do some

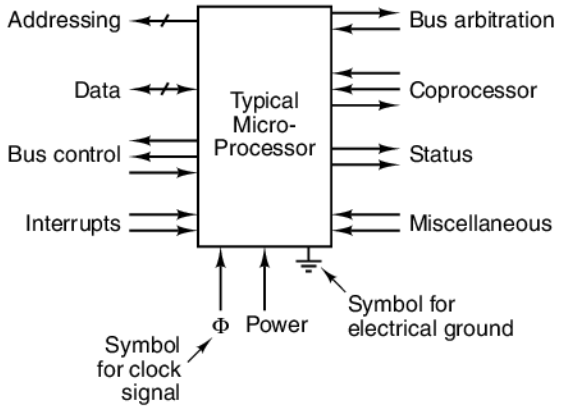


Figure 3-34. The logical pinout of a generic CPU. The arrows indicate input signals and output signals. The short diagonal lines indicate that multiple pins are used. For a specific CPU, a number will be given to tell how many.

other activity, while the I/O device is doing its work. When the I/O has been completed, the I/O controller chip asserts a signal on one of these pins to interrupt the CPU and have it service the I/O device, for example to check whether if I/O errors occurred. Some CPUs have an output pin to acknowledge the interrupt signal.

The bus arbitration pins are needed to regulate traffic on the bus, in order to prevent two devices from trying to use it at the same time. For arbitration purposes, the CPU counts as a device and has to request the bus like any other device.

Some CPU chips are designed to operate with coprocessors such as floating-point chips, but sometimes graphics or other chips as well. To facilitate communication between CPU and coprocessor, special pins are provided for making and granting various requests.

In addition to these signals, there are various miscellaneous pins that some CPUs have. Some of these provide or accept status information, others are useful for debugging or resetting the computer, and still others are present to assure compatibility with older I/O chips.

3.4.2 Computer Buses

A **bus** is a common electrical pathway between multiple devices. Buses can be categorized by their function. They can be used internal to the CPU to transport data to and from the ALU, or external to the CPU to connect it to memory or to I/O devices. Each type of bus has its own requirements and properties. In this section and the following ones, we will focus on buses that connect the CPU to the memory and I/O devices. In the next chapter we will examine more closely the buses inside the CPU.

Early personal computers had a single external bus or **system bus**. It consisted of 50 to 100 parallel copper wires etched onto the motherboard, with connectors spaced at regular intervals for plugging in memory and I/O boards. Modern personal computers generally have a special-purpose bus between the CPU and memory and (at least) one other bus for the I/O devices. A minimal system, with one memory bus and one I/O bus, is illustrated in Fig. 3-35.

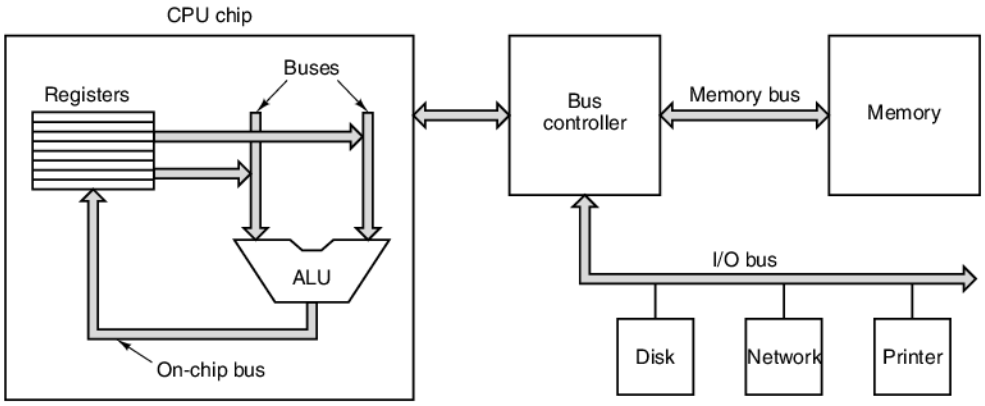


Figure 3-35. A computer system with multiple buses.

In the literature, buses are sometimes drawn as "fat" arrows, as in this figure. The distinction between a fat arrow and a single line with a diagonal line through it and a bit count next to it is subtle. When all the bits are of the same type, say, all address bits or all data bits, then the short-diagonal-line approach is commonly used. When address, data, and control lines are involved, a fat arrow is more common.

While the designers of the CPU are free to use any kind of bus they want inside the chip, in order to make it possible for boards designed by third parties to attach to the system bus, there must be well-defined rules about how the external bus works, which all devices attached to it must obey. These rules are called the **bus protocol**. In addition, there must be mechanical and electrical specifications, so that third-party boards will fit in the card cage and have connectors that match those on the motherboard mechanically and in terms of voltages, timing, etc. Still other buses do not have mechanical specifications because they are designed to be used only within an integrated circuit, for example, to connect components together within a system-on-a-chip (SoC).

A number of buses are in widespread use in the computer world. A few of the better-known ones, historical and current (with examples), are the Omnibus (PDP-8), Unibus (PDP-11), Multibus (8086), VME bus (physics lab equipment), IBM PC bus (PC/XT), ISA bus (PC/AT), EISA bus (80386), Microchannel (PS/2), Nubus (Macintosh), PCI bus (many PCs), SCSI bus (many PCs and workstations),

Universal Serial Bus (modern PCs), and FireWire (consumer electronics). The world would probably be a better place if all but one would suddenly vanish from the face of the earth (well, all right, how about all but two?). Unfortunately, standardization in this area seems very unlikely, as there is already too much invested in all these incompatible systems.

As an aside, there is another interconnect, PCI Express, that is widely referred to as a bus but is not a bus at all. We will study it later in this chapter.

Let us now begin our study of how buses work. Some devices that attach to a bus are active and can initiate bus transfers, whereas others are passive and wait for requests. The active ones are called **masters**; the passive ones are called **slaves**. When the CPU orders a disk controller to read or write a block, the CPU is acting as a master and the disk controller is acting as a slave. However, later on, the disk controller may act as a master when it commands the memory to accept the words it is reading from the disk drive. Several typical combinations of master and slave are listed in Fig. 3-36. Under no circumstances can memory ever be a master.

Master	Slave	Example
CPU	Memory	Fetching instructions and data
CPU	I/O device	Initiating data transfer
CPU	Coprocessor	CPU handing instruction off to coprocessor
I/O device	Memory	DMA (Direct Memory Access)
Coprocessor	CPU	Coprocessor fetching operands from CPU

Figure 3-36. Examples of bus masters and slaves.

The binary signals that computer devices output are frequently too weak to power a bus, especially if it is relatively long or has many devices on it. For this reason, most bus masters are connected to the bus by circuitry called a **bus driver**, which is essentially a digital amplifier. Similarly, most slaves are connected to the bus by a **bus receiver**. For devices that can act as both master and slave, a combined circuit called a **bus transceiver** is used. These bus interfaces are often tri-state devices, to allow them to float (disconnect) when they are not needed, or are hooked up in a somewhat different way, called **open collector**, that achieves a similar effect. When two or more devices on an open-collector line assert the line at the same time, the result is the Boolean OR of all the signals. This arrangement is often called **wired-OR**. On most buses, some of the lines are tri-state and others, which need the wired-OR property, are open collector.

Like a CPU, a bus also has address, data, and control lines. However, there is not necessarily a one-to-one mapping between the CPU pins and the bus signals. For example, some CPUs have three pins that encode whether the CPU is doing a memory read, memory write, I/O read, I/O write, or some other operation. A typical bus might have one line for memory read, a second for memory write, a third for I/O read, a fourth for I/O write, and so on. A decoder circuit would then be

needed between the CPU and such a bus to match the two sides up, that is, to convert the 3-bit encoded signal into separate signals that can drive the bus lines.

Bus design and operation are sufficiently complex subjects that a number of entire books have been written about them (Anderson et al., 2004, Solari and Willse, 2004). The principal bus design issues are bus width, bus clocking, bus arbitration, and bus operations. Each of these issues has a substantial impact on the speed and bandwidth of the bus. We will now examine each of these in the next four sections.

3.4.3 Bus Width

Bus width is the most obvious design parameter. The more address lines a bus has, the more memory the CPU can address directly. If a bus has n address lines, then a CPU can use it to address 2^n different memory locations. To allow large memories, buses need many address lines. That sounds simple enough.

The problem is that wide buses need more wires than narrow ones. They also take up more physical space (e.g., on the motherboard) and need bigger connectors. All of these factors make the bus more expensive. Thus, there is a trade-off between maximum memory size and system cost. A system with a 64-line address bus and 2^{32} bytes of memory will cost more than one with 32 address lines and the same 2^{32} bytes of memory. The possibility of expansion later is not free.

The result of this observation is that many system designers tend to be short-sighted, with unfortunate consequences later. The original IBM PC contained an 8088 CPU and a 20-bit address bus, as shown in Fig. 3-37(a). Having 20 bits allowed the PC to address 1 MB of memory.

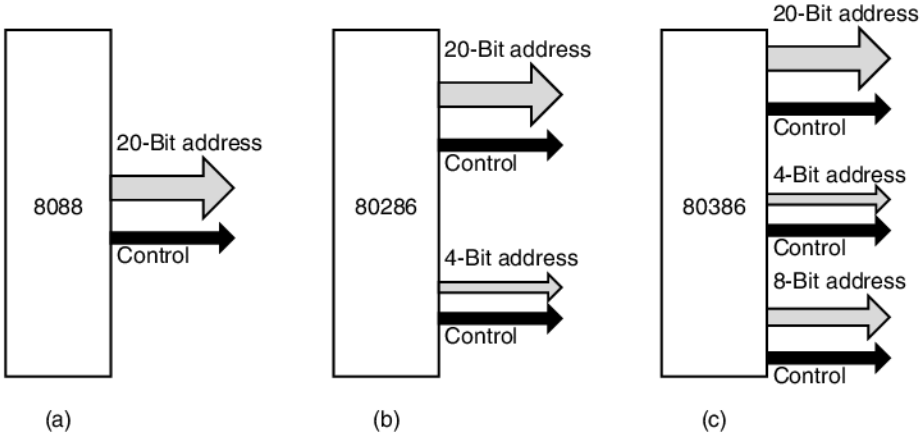


Figure 3-37. Growth of an address bus over time.

When the next CPU chip (the 80286) came out, Intel flet it had to increase the address space to 16 MB, so four more bus lines were added (without disturbing the

original 20, for reasons of backward compatibility), as illustrated in Fig. 3-37(b). Unfortunately, more control lines had to be added to deal with the new address lines. When the 80386 came out, another eight address lines were added, along with still more control lines, as shown in Fig. 3-37(c). The resulting design (the EISA bus) is much messier than it would have been had the bus been given 32 lines at the start.

Not only does the number of address lines tend to grow over time, but so does the number of data lines, albeit for a different reason. There are two ways to increase the bandwidth of a bus: decrease the bus cycle time (more transfers/sec) or increase the data bus width (more bits/transfer). Speeding the bus up is possible (but difficult) because the signals on different lines travel at slightly different speeds, a problem known as **bus skew**. The faster the bus, the more the skew.

Another problem with speeding up the bus is it will not be backward compatible. Old boards designed for the slower bus will not work with the new one. Invalidating old boards makes both the owners and manufacturers of the old boards unhappy. Therefore the usual approach to improving performance is to add more data lines, analogous to Fig. 3-37. As you might expect, however, this incremental growth does not lead to a clean design in the end. The IBM PC and its successors, for example, went from 8 data lines to 16 and then 32 on essentially the same bus.

To get around the problem of very wide buses, sometimes designers opt for a **multiplexed bus**. In this design, instead of the address and data lines being separate, there are, say, 32 lines for address and data together. At the start of a bus operation, the lines are used for the address. Later on, they are used for data. For a write to memory, for example, this means that the address lines must be set up and propagated to the memory before the data can be put on the bus. With separate lines, the address and data can be put on together. Multiplexing the lines reduces bus width (and cost) but results in a slower system. Bus designers have to carefully weigh all these options when making choices.

3.4.4 Bus Clocking

Buses can be divided into two distinct categories depending on their clocking. A **synchronous bus** has a line driven by a crystal oscillator. The signal on this line consists of a square wave with a frequency generally between 5 and 133 MHz. All bus activities take an integral number of these cycles, called **bus cycles**. The other kind of bus, the **asynchronous bus**, does not have a master clock. Bus cycles can be of any length required and need not be the same between all pairs of devices. Below we will examine each bus type.

Synchronous Buses

As an example of how a synchronous bus works, consider the timing of Fig. 3-38(a). In this example, we will use a 100-MHz clock, which gives a bus cycle of 10 nsec. While this may seem a bit slow compared to CPU speeds of 3

GHz and more, few existing PC buses are much faster. For example, the popular PCI bus usually runs at either 33 or 66 MHz, and the upgraded (but now defunct) PCI-X bus ran at a speed of up to 133 MHz. The reasons current buses are slow were given above: technical design problems such as bus skew and the need for backward compatibility.

In our example, we will further assume that reading from memory takes 15 nsec from the time the address is stable. As we will see shortly, with these parameters, it will take three bus cycles to read a word. The first cycle starts at the rising edge of T_1 and the third one ends at the rising edge of T_4, as shown in the figure. Note that none of the rising or falling edges has been drawn vertically, because no electrical signal can change its value in zero time. In this example we will assume that it takes 1 nsec for a signal to change. The clock, ADDRESS, DATA, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$, and $\overline{\text{WAIT}}$ lines are all shown on the same time scale.

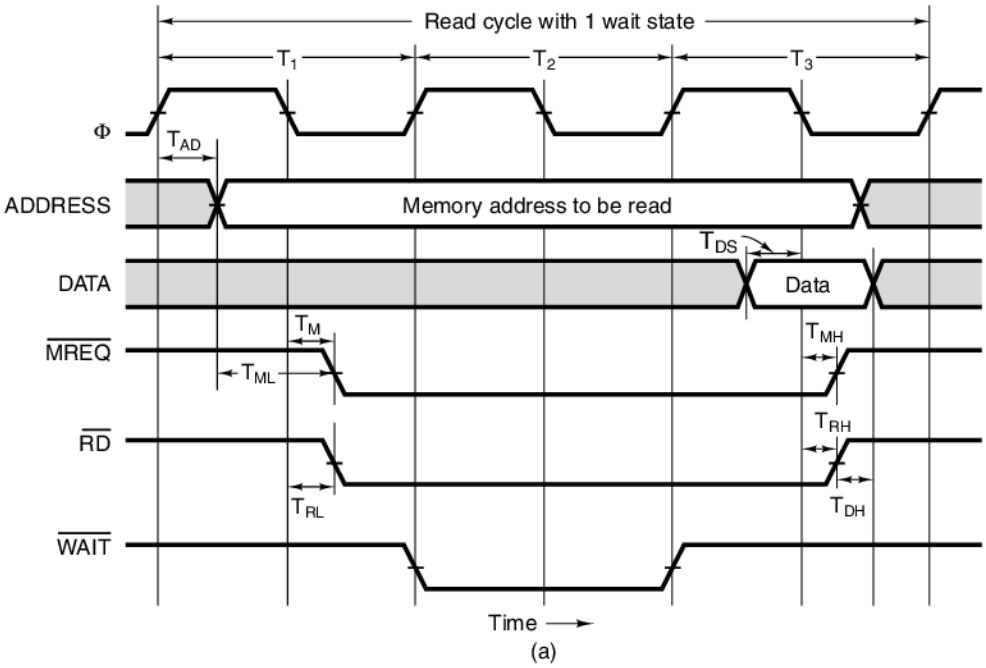
The start of T_1 is defined by the rising edge of the clock. Partway through T_1 the CPU puts the address of the word it wants on the address lines. Because the address is not a single value, like the clock, we cannot show it as a single line in the figure; instead, it is shown as two lines, with a crossing at the time that the address changes. Furthermore, the shading prior to the crossing indicates that the shaded value is not important. Using the same shading convention, we see that the contents of the data lines are not significant until well into T_3.

After the address lines have had a chance to settle down to their new values, $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$ are asserted. The former indicates that memory (as opposed to an I/O device) is being accessed, and the latter is asserted for reads and negated for writes. Since the memory takes 15 nsec after the address is stable (partway into the first clock cycle), it cannot provide the requested data during T_2. To tell the CPU not to expect it, the memory asserts the $\overline{\text{WAIT}}$ line at the start of T_2. This action will insert **wait states** (extra bus cycles) until the memory is finished and negates $\overline{\text{WAIT}}$. In our example, one wait state (T_2) has been inserted because the memory is too slow. At the start of T_3, when it is sure it will have the data during the current cycle, the memory negates $\overline{\text{WAIT}}$.

During the first half of T_3, the memory puts the data onto the data lines. At the falling edge of T_3 the CPU strobes (i.e., reads) the data lines, latching (i.e., storing) the value in an internal register. Having read the data, the CPU negates $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$. If need be, another memory cycle can begin at the next rising edge of the clock. This sequence can be repeatedly indefinitely.

In the timing specification of Fig. 3-38(b), eight symbols that occur in the timing diagram are further clarified. T_{AD}, for example, is the time interval between the rising edge of the T_1 clock and the address lines being set. According to the timing specification, $T_{\text{AD}} \leq 4$ nsec. This means that the CPU manufacturer guarantees that during any read cycle, the CPU will output the address to be read within 4 nsec of the midpoint of the rising edge of T_1.

The timing specifications also require that the data be available on the data lines at least T_{DS} (2 nsec) before the falling edge of T_3, to give it time to settle



Symbol	Parameter	Min	Max	Unit
T_{AD}	Address output delay		4	nsec
T_{ML}	Address stable prior to $\overline{\text{MREQ}}$	2		nsec
T_M	$\overline{\text{MREQ}}$ delay from falling edge of Φ in T_1		3	nsec
T_{RL}	RD delay from falling edge of Φ in T_1		3	nsec
T_{DS}	Data setup time prior to falling edge of Φ	2		nsec
T_{MH}	$\overline{\text{MREQ}}$ delay from falling edge of Φ in T_3		3	nsec
T_{RH}	$\overline{\text{RD}}$ delay from falling edge of Φ in T_3		3	nsec
T_{DH}	Data hold time from negation of $\overline{\text{RD}}$	0		nsec

(b)

Figure 3-38. (a) Read timing on a synchronous bus. (b) Specification of some critical times.

down before the CPU strobes it in. The combination of constraints on T_{AD} and T_{DS} means that, in the worst case, the memory will have only $25 - 4 - 2 = 19$ nsec from the time the address appears until it must produce the data. Because 10 nsec is enough, even in the worst case, a 10-nsec memory can always respond during T_3. A 20-nsec memory, however, would just miss and have to insert a second wait state and respond during T_4.

The timing specification further guarantees that the address will be set up at least 2 nsec prior to $\overline{\text{MREQ}}$ being asserted. This time can be important if $\overline{\text{MREQ}}$ drives chip select on the memory chip because some memories require an address setup time prior to chip select. Clearly, the system designer should not choose a memory chip that needs a 3-nsec setup time.

The constraints on T_M and T_{RL} mean that $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$ will both be asserted within 3 nsec from the T_1 falling clock. In the worst case, the memory chip will have only 10 + 10 − 3 − 2 = 15 nsec after the assertion of $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$ to get its data onto the bus. This constraint is in addition to (and independent of) the 15-nsec interval needed after the address is stable.

T_{MH} and T_{RH} tell how long it takes $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$ to be negated after the data have been strobed in. Finally, T_{DH} tells how long the memory must hold the data on the bus after $\overline{\text{RD}}$ has been negated. As far as our example CPU is concerned, the memory can remove the data from the bus as soon as $\overline{\text{RD}}$ has been negated. On some actual CPUs, however, the data must be kept stable a little longer.

We would like to point out that Fig. 3-38 is a highly simplified version of real timing constraints. In reality, many more critical times are always specified. Nevertheless, it gives a good flavor for how a synchronous bus works.

A last point worth making is that control signals can be asserted high or low. It is up to the bus designers to determine which is more convenient, but the choice is essentially arbitrary. One can regard it as the hardware equivalent of a programmer's choice to represent free disk blocks in a bit map as 0s vs. 1s.

Asynchronous Buses

Although synchronous buses are easy to work with due to their discrete time intervals, they also have some problems. For example, everything works in multiples of the bus clock. If a CPU and memory are able to complete a transfer in 3.1 cycles, they have to stretch it to 4.0 because fractional cycles are forbidden.

Worse yet, once a bus cycle has been chosen, and memory and I/O cards have been built for it, it is difficult to take advantage of future improvements in technology. For example, suppose a few years after the system of Fig. 3-38 was built, new memories became available with access times of 8 nsec instead of 15 nsec. These would get rid of the wait state, speeding up the machine. Then suppose 4-nsec memories became available. There would be no further gain in performance because the minimum time for a read is two cycles with this design.

Putting this in slightly different terms, if a synchronous bus has a heterogeneous collection of devices, some fast and some slow, the bus has to be geared to the slowest one and the fast ones cannot use their full potential.

Mixed technology can be handled by going to an asynchronous bus, that is, one with no master clock, as shown in Fig. 3-39. Instead of tying everything to the clock, when the bus master has asserted the address, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$, and anything else

it needs to, it then asserts a special signal that we will call $\overline{\text{MSYN}}$ (Master SYNchronization). When the slave sees this, it performs the work as fast as it can. When it is done, it asserts $\overline{\text{SSYN}}$ (Slave SYNchronization).

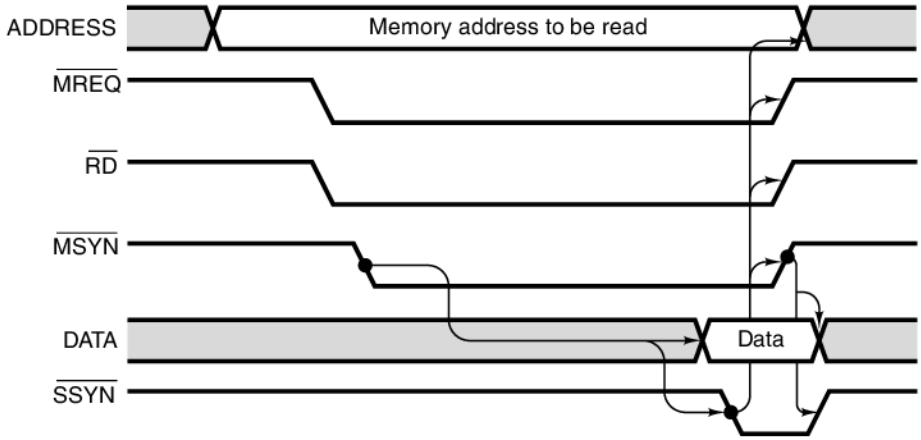


Figure 3-39. Operation of an asynchronous bus.

As soon as the master sees $\overline{\text{SSYN}}$ asserted, it knows that the data are available, so it latches them and then negates the address lines, along with $\overline{\text{MREQ}}$, $\overline{\text{RD}}$, and $\overline{\text{MSYN}}$. When the slave sees the negation of $\overline{\text{MSYN}}$, it knows that the cycle has been completed, so it negates $\overline{\text{SSYN}}$, and we are back in the original situation, with all signals negated, waiting for the next master.

Timing diagrams of asynchronous buses (and sometimes synchronous buses as well) use arrows to show cause and effect, as in Fig. 3-39. The assertion of $\overline{\text{MSYN}}$ causes the data lines to be asserted and also causes the slave to assert $\overline{\text{SSYN}}$. The assertion of $\overline{\text{SSYN}}$, in turn, causes the negation of the address lines, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$, and $\overline{\text{MSYN}}$. Finally, the negation of $\overline{\text{MSYN}}$ causes the negation of $\overline{\text{SSYN}}$, which ends the read and returns the system to its original state.

A set of signals that interlocks this way is called a **full handshake**. The essential part consists of four events:

1. $\overline{\text{MSYN}}$ is asserted.
2. $\overline{\text{SSYN}}$ is asserted in response to $\overline{\text{MSYN}}$.
3. $\overline{\text{MSYN}}$ is negated in response to $\overline{\text{SSYN}}$.
4. $\overline{\text{SSYN}}$ is negated in response to the negation of $\overline{\text{MSYN}}$.

It should be clear that full handshakes are timing independent. Each event is caused by a prior event, not by a clock pulse. If a particular master/slave pair is slow, that in no way affects a subsequent master/slave pair that is much faster.

The advantage of an asynchronous bus should now be clear, but the fact is that most buses are synchronous. The reason is that it is easier to build a synchronous system. The CPU just asserts its signals, and the memory just reacts. There is no feedback (cause and effect), but if the components have been chosen properly, everything will work without handshaking. Also, there is a lot of investment in synchronous bus technology.

3.4.5 Bus Arbitration

Up until now, we have tacitly assumed that there is only one bus master, the CPU. In reality, I/O chips have to become bus master to read and write memory, and also to cause interrupts. Coprocessors may also need to become bus master. The question then arises: "What happens if two or more devices all want to become bus master at the same time?" The answer is that some **bus arbitration** mechanism is needed to prevent chaos.

Arbitration mechanisms can be centralized or decentralized. Let us first consider centralized arbitration. One particularly simple form of this is shown in Fig. 3-40(a). In this scheme, a single bus arbiter determines who goes next. Many CPUs have the arbiter built into the CPU chip, but sometimes a separate chip is needed. The bus contains a single wired-OR request line that can be asserted by one or more devices at any time. There is no way for the arbiter to tell how many devices have requested the bus. The only categories it can distinguish are some requests and no requests.

When the arbiter sees a bus request, it issues a grant by asserting the bus grant line. This line is wired through all the I/O devices in series, like a cheap string of Christmas tree lamps. When the device physically closest to the arbiter sees the grant, it checks to see if it has made a request. If so, it takes over the bus but does not propagate the grant further down the line. If it has not made a request, it propagates the grant to the next device in line, which behaves the same way, and so on until some device accepts the grant and takes the bus. This scheme is called **daisy chaining**. It has the property that devices are effectively assigned priorities depending on how close to the arbiter they are. The closest device wins.

To get around the implicit priorities based on distance from the arbiter, many buses have multiple priority levels. For each priority level there is a bus request line and a bus grant line. The one of Fig. 3-40(b) has two levels, 1 and 2 (real buses often have 4, 8, or 16 levels). Each device attaches to one of the bus request levels, with more time-critical devices attaching to the higher-priority ones. In Fig. 3-40(b) devices, 1, 2, and 4 use priority 1 while devices 3 and 5 use priority 2.

If multiple priority levels are requested at the same time, the arbiter issues a grant only on the highest-priority one. Among devices of the same priority, daisy chaining is used. In Fig. 3-40(b), in the event of conflicts, device 2 beats device 4, which beats 3. Device 5 has the lowest-priority because it is at the end of the lowest priority daisy chain.

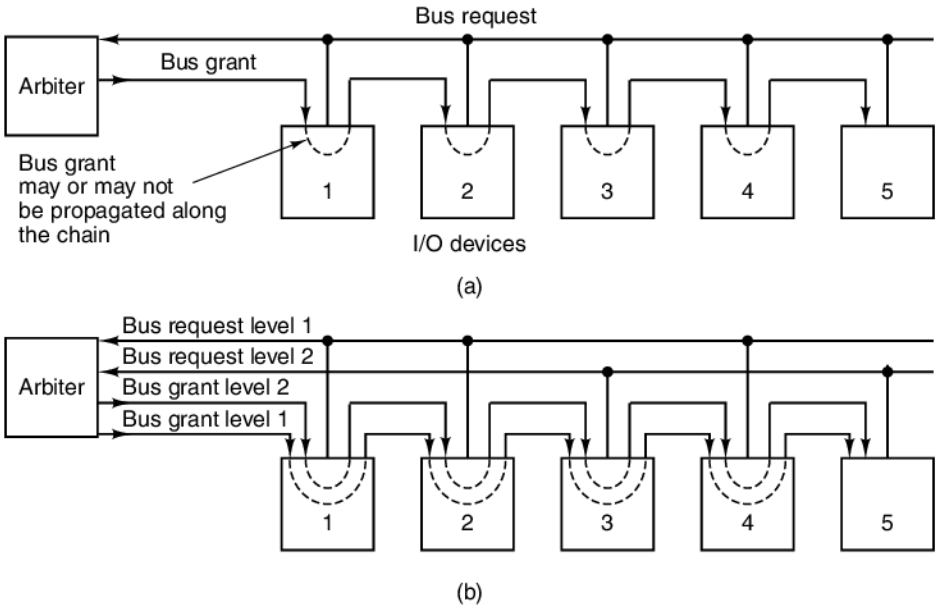


Figure 3-40. (a) A centralized one-level bus arbiter using daisy chaining. (b) The same arbiter, but with two levels.

As an aside, it is not technically necessary to wire the level 2 bus grant line serially through devices 1 and 2, since they cannot make requests on it. However, as an implementation convenience, it is easier to wire all the grant lines through all the devices, rather than making special wiring that depends on which device has which priority.

Some arbiters have a third line that a device asserts when it has accepted a grant and seized the bus. As soon as it has asserted this acknowledgement line, the request and grant lines can be negated. As a result, other devices can request the bus while the first device is using the bus. By the time the current transfer is finished, the next bus master will have already been selected. It can start as soon as the acknowledgement line has been negated, at which time the following round of arbitration can begin. This scheme requires an extra bus line and more logic in each device, but it makes better use of bus cycles.

In systems in which memory is on the main bus, the CPU must compete with all the I/O devices for the bus on nearly every cycle. One common solution for this situation is to give the CPU the lowest priority, so it gets the bus only when nobody else wants it. The idea here is that the CPU can always wait, but I/O devices frequently must acquire the bus quickly or lose incoming data. Disks rotating at high speed cannot wait. This problem is avoided in many modern computer systems by

putting the memory on a separate bus from the I/O devices so they do not have to compete for access to the bus.

Decentralized bus arbitration is also possible. For example, a computer could have 16 prioritized bus request lines. When a device wants to use the bus, it asserts its request line. All devices monitor all the request lines, so at the end of each bus cycle, each device knows whether it was the highest-priority requester, and thus whether it is permitted to use the bus during the next cycle. Compared to centralized arbitration, this arbitration method requires more bus lines but avoids the potential cost of the arbiter. It also limits the number of devices to the number of request lines.

Another kind of decentralized bus arbitration, shown in Fig. 3-41, uses only three lines, no matter how many devices are present. The first bus line is a wired-OR line for requesting the bus. The second bus line is called BUSY and is asserted by the current bus master. The third line is used to arbitrate the bus. It is daisy chained through all the devices. The head of this chain is held asserted by tying it to the power supply.

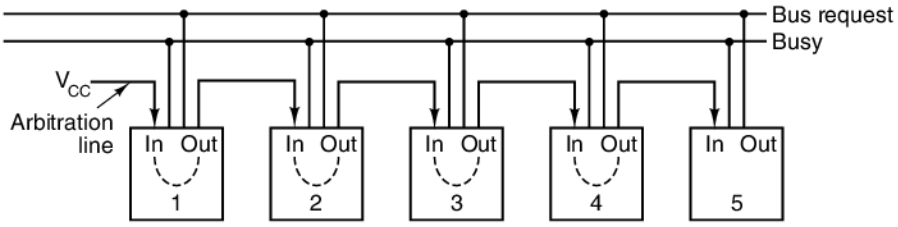


Figure 3-41. Decentralized bus arbitration.

When no device wants the bus, the asserted arbitration line is propagated through to all devices. To acquire the bus, a device first checks to see if the bus is idle and the arbitration signal it is receiving, IN, is asserted. If IN is negated, it may not become bus master, and it negates OUT. If IN is asserted, however, and the device wants the bus, the device negates OUT, which causes its downstream neighbor to see IN negated and to negate its OUT. Then all downstream devices all see IN negated and correspondingly negate OUT. When the dust settles, only one device will have IN asserted and OUT negated. This device becomes bus master, asserts BUSY and OUT, and begins its transfer.

Some thought will reveal that the leftmost device that wants the bus gets it. Thus, this scheme is similar to the original daisy chain arbitration, except without having the arbiter, so it is cheaper, faster, and not subject to arbiter failure.

3.4.6 Bus Operations

Up until now, we have discussed only ordinary bus cycles, with a master (typically the CPU) reading from a slave (typically the memory) or writing to one. In fact, several other kinds of bus cycles exist. We will now look at some of these.

Normally, one word at a time is transferred. However, when caching is used, it is desirable to fetch an entire cache line (e.g., 8 consecutive 64-bit words) at once. Often block transfers can be made more efficient than successive individual transfers. When a block read is started, the bus master tells the slave how many words are to be transferred, for example, by putting the word count on the data lines during T_1. Instead of just returning one word, the slave outputs one word during each cycle until the count has been exhausted. Figure 3-42 shows a modified version of Fig. 3-38(a), but now with an extra signal $\overline{\text{BLOCK}}$ that is asserted to indicate that a block transfer is requested. In this example, a block read of 4 words takes 6 cycles instead of 12.

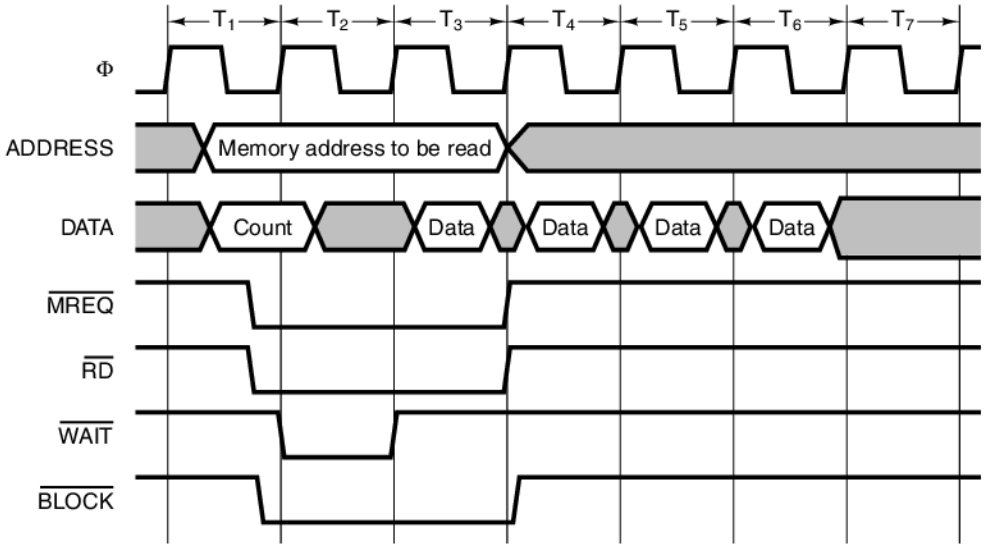


Figure 3-42. A block transfer.

Other kinds of bus cycles also exist. For example, on a multiprocessor system with two or more CPUs on the same bus, it is often necessary to make sure that only one CPU at a time uses some critical data structure in memory. A typical way to arrange this is to have a variable in memory that is 0 when no CPU is using the data structure and 1 when it is in use. If a CPU wants to gain access to the data structure, it must read the variable, and if it is 0, set it to 1. The trouble is, with some bad luck, two CPUs might read it on consecutive bus cycles. If each one sees that the variable is 0, then each one sets it to 1 and thinks that it is the only CPU using the data structure. This sequence of events leads to chaos.

To prevent this situation, multiprocessor systems often have a special read-modify-write bus cycle that allows any CPU to read a word from memory, inspect and modify it, and write it back to memory, all without releasing the bus.

This type of cycle prevents competing CPUs from being able to use the bus and thus interfere with the first CPU's operation.

Another important kind of bus cycle is for handling interrupts. When the CPU commands an I/O device to do something, it usually expects an interrupt when the work is done. The interrupt signaling requires the bus.

Since multiple devices may want to cause an interrupt simultaneously, the same kind of arbitration problems are present here that we had with ordinary bus cycles. The usual solution is to assign priorities to devices and use a centralized arbiter to give priority to the most time-critical devices. Standard interrupt interfaces exist and are widely used. In Intel-processor-based PCs, the chipset incorporates an 8259A interrupt controller, illustrated in Fig. 3-43.

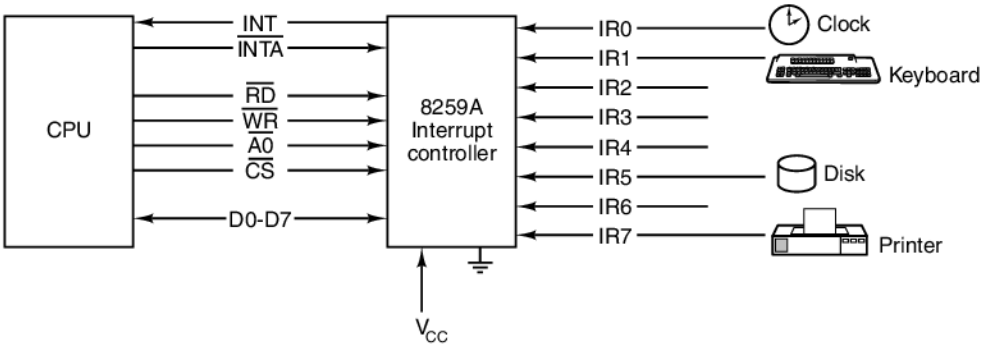


Figure 3-43. Use of the 8259A interrupt controller.

Up to eight 8259A I/O controllers can be directly connected to the eight IRx (Interrupt Request) inputs to the 8259A. When any of these devices wants to cause an interrupt, it asserts its input line. When one or more inputs are asserted, the 8259A asserts INT (INTerrupt), which directly drives the interrupt pin on the CPU. When the CPU is able to handle the interrupt, it sends a pulse back to the 8259A on INTA (INTerrupt Acknowledge). At that point the 8259A must specify which input caused the interrupt by outputting that input's number on the data bus. This operation requires a special bus cycle. The CPU hardware then uses that number to index into a table of pointers, called **interrupt vectors**, to find the address of the procedure to run to service the interrupt.

The 8259A has several registers inside that the CPU can read and write using ordinary bus cycles and the $\overline{\text{RD}}$ (ReaD), $\overline{\text{WR}}$ (WRite), $\overline{\text{CS}}$ (Chip Select), and $\overline{\text{A0}}$ pins. When the software has handled the interrupt and is ready to take the next one, it writes a special code into one of the registers, which causes the 8259A to negate INT, unless it has another interrupt pending. These registers can also be written to put the 8259A in one of several modes, mask out a set of interrupts, and enable other features.

When more than eight I/O devices are present, the 8259As can be cascaded. In the most extreme case, all eight inputs can be connected to the outputs of eight

more 8259As, allowing for up to 64 I/O devices in a two-stage interrupt network. The Intel ICH10 I/O controller hub, one of the of the chips in the Core i7 chipset, incorporates two 8259A interrupt controllers. This gives the ICH10 15 external interrupts, one less than 16 interrupts on the two 8259A controllers because one of the interrupts is used to cascade the second 8259A onto the first one. The 8259A has a few pins devoted to handling this cascading, which we have omitted for the sake of simplicity. Nowadays, the "8259A" is really part of another chip.

While we have by no means exhausted the subject of bus design, the material above should give enough background to understand the essentials of how a bus works, and how CPUs and buses interact. Let us now move from the general to the specific and look at some examples of actual CPUs and their buses.

3.5 EXAMPLE CPU CHIPS

In this section we will examine the Intel Core i7, TI OMAP4430, and Atmel ATmega168 chips in some detail at the hardware level.

3.5.1 The Intel Core i7

The Core i7 is a direct descendant of the 8088 CPU used in the original IBM PC. The first Core i7 was introduced in November 2008 as a four-processor 731-million transistor CPU running up to 3.2 GHz with a line width of 45 nanometers. The line width is how wide the wires between transistors are (as well as being a measure of the size of the transistors themselves). The narrower the line width, the more transistors can fit on the chip. Moore's law is fundamentally about the ability of process engineers to keep reducing the line widths. For comparison purposes, human hairs range from 20,000 to 100,000 nanometers in diameter, with blonde hair being finer than black hair.

The initial release of the Core i7 architecture was based on the "Nahalem" architecture; however, the newest versions of the Core i7 are built on the newer "Sandy Bridge" architecture. The architecture in this context represents the internal organization of the CPU, which is often given a code name. Despite being generally serious people, computer architects will sometimes come up with very clever code names for their projects. One of particular note was the AMD K-series architectures, which were designed to break Intel's seeming invulnerable hold on the desktop CPU market. The K-series processors' code name was "Kryptonite," a reference to the only substance that could hurt Superman, and a clever jab at the dominant Intel.

The new Sandy-Bridge-based Core i7 has evolved to having 1.16 billion transistors and running at speeds up to 3.5 GHz with line widths of 32 nanometers. Although the Core i7 is a far cry from the 29,000-transistor 8088, it is fully backward

compatible with the 8088 and can run unmodified 8088 binary programs (not to mention programs for all the intermediate processors as well).

From a software point of view, the Core i7 is a full 64-bit machine. It has all the same user-level ISA features as the 80386, 80486, Pentium, Pentium II, Pentium Pro, Pentium III, and Pentium 4 including the same registers, same instructions, and a full on-chip implementation of the IEEE 754 floating-point standard. In addition, it has some new instructions intended primarily for cryptographic operations.

The Core i7 processor is a multicore CPU, thus the silicon die contains multiple processors. The CPU is sold with a varying number of processors, ranging from 2 to 6 with more planned for the near future. If programmers write a parallel program, using threads and locks, it is possible to gain significant program speedups by exploiting parallelism on multiple processors. In addition, the individual CPUs are "hyperthreaded" such that multiple hardware threads can be active simultaneously. Hyperthreading (more typically called "simultaneous multithreading" by computer architects) allows very short latencies, such as cache misses, to be tolerated with hardware thread switches. Software-based threading can tolerate only very long latencies, such as page faults, due to the hundreds of cycles needed to implement software-based thread switches.

Internally, at the microarchitecture level, the Core i7 is a very capable design. It is based on the architecture of its predecessors, the Core 2 and Core 2 Duo. The Core i7 processor can carry out up to four instructions at once, making it a 4-wide superscalar machine. We will examine the microarchitecture in Chap. 4.

The Core i7 processors all have three levels of cache. Each processor in a Core i7 processor has a 32-KB level 1 (L1) data cache and a 32-KB level 1 instruction cache. Each core also has its own 256-KB level 2 (L2) cache. The second-level cache is unified, which means that it can hold a mixture of instructions and data. All cores share a single level 3 (L3) unified cache, the size of which varies from 4 to 15 MB depending on the processor model. Having three levels of cache significantly improves processor performance but at a great cost in silicon area, as Core i7 CPUs can have as much as 17 MB total cache on a single silicon die.

Since all Core i7 chips have multiple processors with private data caches, a problem arises when a processor modifies a word in this private cache that is contained in another processor's cache. If the other processor tries to read that word from memory, it will get a stale value, since modified cache words are not written back to memory immediately. To maintain memory consistency, each CPU in a multiprocessor system **snoops** on the memory bus looking for references to words it has cached. When it sees such a reference, it jumps in and supplies the required data before the memory gets a chance to do so. We will study snooping in Chap. 8.

Two primary external buses are used in Core i7 systems, both of them synchronous. A DDR3 memory bus is used to access the main memory DRAM, and a PCI Express bus connects the processor to I/O devices. High-end versions of the Core i7 include multiple memory and PCI Express buses, and they also include a

Quick Path Interconnect (QPI) port. The QPI port connects the processor to an external multiprocessor interconnect, allowing systems with more than six processors to be built. The QPI port sends and receives cache coherency requests, plus a variety of other multiprocessor management messages such as interprocessor interrupts.

A problem with the Core i7 as well as with most other modern desktop-class CPUs, is the power it consumes and the heat it generates. To prevent damaging the silicon, the heat must be moved away from the processor die soon after it is produced. The Core i7 consumes between 17 and 150 watts, depending on the frequency and model. Consequently, Intel is constantly searching for ways to manage the heat produced by its CPU chips. Cooling technologies and heat-conductive packaging are vital to protecting the silicon from burning up.

The Core i7 comes in a square LGA package 37.5 mm on edge. It contains 1155 pads on the bottom, of which 286 are for power and 360 are grounded to reduce noise. The pads are arranged roughly as a 40×40 square, with the middle 17×25 missing. In addition, 20 more pads are missing at the perimeter in an asymmetric pattern, to prevent the chip from being inserted incorrectly in its socket. The physical pinout is shown in Fig. 3-44.

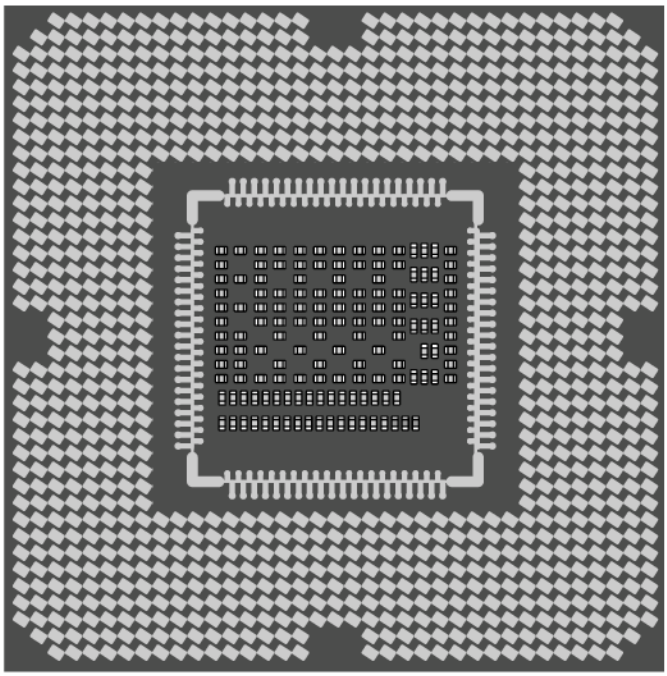


Figure 3-44. The Core i7 physical pinout.

The chip is outfitted with a mounting plate for a heat sink to distribute the heat and a fan to cool it. To get some idea of how large the power problem is, turn on a

150-watt incandescent light bulb, let it warm up, and then put your hands around it (but do not touch it). This amount of heat must be dissipated continuously by a high-end Core i7 processor. Consequently, when a Core i7 has outlived its usefulness as a CPU, it can always be used as a camp stove.

According to the laws of physics, anything that puts out a lot of heat must suck in a lot of energy. In a portable computer with a limited battery charge, using a lot of energy is not desirable because it drains the battery quickly. To address this issue, Intel has provided a way to put the CPU to sleep when it is idle and to put it into a deep sleep when it is likely to be that way for a while. Five states are provided, ranging from fully active to deep sleep. In the intermediate states, some functionality (such as cache snooping and interrupt handling) is enabled, but other functions are disabled. When in deep sleep state, the register values are preserved, but the caches are flushed and turned off. When in deep sleep, a hardware signal is required to wake it up. It is not known whether a Core i7 can dream when it is in deep sleep.

The Core i7's Logical Pinout

The 1155 pins on the Core i7 are used for 447 signals, 286 power connections (at several different voltages), 360 grounds, and 62 spares for future use. Some of the logical signals use two or more pins (such as the memory-address requested), so there are only 131 different signals. A somewhat simplified logical pinout is given in Fig. 3-45. On the left side of the figure are five major groups of bus signals; on the right side are various miscellaneous signals.

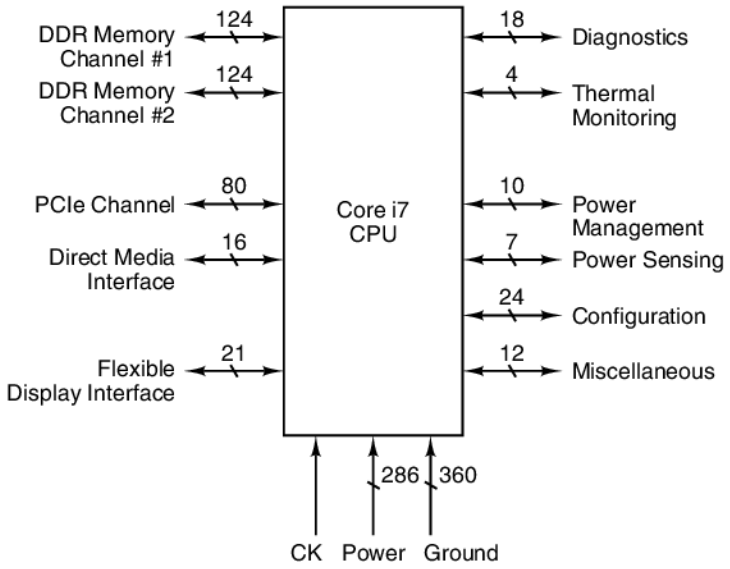


Figure 3-45. Logical pinout of the Core i7.

Let us examine the signals, starting with the bus signals. The first two bus signals are used to interface to DDR3 compatible DRAM. This group of signals provides address, data, control and clock to the DRAM bank. The Core i7 supports two independent DDR3 DRAM channels, running with a 666-MHz bus clock that transfers on both edges to allow 1333 million transactions per second. The DDR3 interface is 64 bits wide, thus, the two DDR3 interfaces work in tandem to provide memory-hungry programs up to 20 gigabytes of data each second.

The third bus group is the PCI Express interface, which is used to connect peripherals directly to the Core i7 CPU. The PCI Express interface is a high-speed serial interface, with each single serial link forming a "lane" of communication with peripherals. The Core i7 link is an x16 interface, which means that it can utilize 16 lanes simultaneously for an aggregate bandwidth of 16 GB/sec. Despite its being a serial channel, a rich set of commands travel over PCI Express links, including device reads, writes, interrupts, and configuration setup commands.

The next bus group is the Direct Media Interface (DMI), which is used to connect the Core i7 CPU to its companion **chipset**. The DMI interface is similar to the PCI Express interface, although it runs at about half the speed since four lanes can provide only up to 2.5-GB-per-second data transfer rates. A CPU's chipset contains a rich set of additional peripheral interface support, typically required for higher-end system with many I/O devices. The Core i7 chipset is composed of the P67 and ICH10 chips. The P67 chip is the Swiss Army knife of chips, providing SATA, USB, Audio, PCIe, and Flash memory interfaces. The ICH10 chip provides legacy interface support, including a PCI interface and the 8259A interrupt control functionality. Additionally, the ICH10 contains a handful of other circuits, such as real-time clocks, event timers, and direct memory access (DMA) controllers. Having chips like these greatly simplifies construction of a full PC.

The Core i7 can be configured to use interrupts the same way as on the 8088 (for purposes of backward compatibility), or it can also use a new interrupt system using a device called an **APIC** (**Advanced Programmable Interrupt Controller**).

The Core i7 can run at any one of several predefined voltages, but it has to know which one. The power-management signals are used for automatic power-supply voltage selection, telling the CPU that power is stable, and other power-related matters. Managing the various sleep states is also done here since sleeping is done for reasons of power management.

Despite sophisticated power management, the Core i7 can get very hot. To protect the silicon, each Core i7 processor contains multiple internal heat sensors that detect when the chip is about to overheat. The thermal monitoring group deals with thermal management, allowing the CPU to indicate to its environment that it is in danger of overheating. One of the pins is asserted by the CPU if its internal temperature reaches 130°C (266°F). If a CPU ever hits this temperature, it is probably dreaming about retirement and becoming a camp stove.

Even at camp-stove temperatures, you need not worry about the safety of the Core i7. If the internal sensors detect that the processor is about to overheat, it will

initiate **thermal throttling**, which is a technique that quickly reduces heat generation by running the processor only every *N*th clock cycle. The larger the value of *N*, the more the processor is throttled down, and the more quickly it will cool. Of course, the cost of this throttling is a decrease in system performance. Prior to the invention of thermal throttling, CPUs would burn up if their cooling system failed. Evidence of these dark times of CPU thermal management can be found by searching for "exploding CPU" on YouTube. The video is fake but the problem is not.

The Clock signal provides the system clock to the processor, which internally is used to generate variety of clocks based on a multiple or fraction of the system clock. Yes, it is possible to generate a multiple of the system clock frequency, using a very clever device called a delay-locked loop, or DLL.

The Diagnostics group contains signals for testing and debugging systems in conformance with the IEEE 1149.1 JTAG (Joint Test Action Group) test standard. Finally, the miscellaneous group is a hodge-podge of other signals that have various special purposes.

Pipelining on the Core i7's DDR3 Memory Bus

Modern CPUs like the Core i7 place heavy demands on DRAM memories. Individual processors can produce access requests much faster than a slow DRAM can produce values, and this problem is compounded when multiple processors are making simultaneous requests. To keep the CPUs from starving for lack of data, it is essential to get the maximum possible throughput from the memory. For this reason, the Core i7 DDR3 memory bus can be operated in a pipelined manner, with as many as four simultaneous memory transactions going on at the same time. We saw the concept of pipelining in Chap. 2 in the context of a pipelined CPU (see Fig. 2-4), but memories can also be pipelined.

To allow pipelining, Core i7 memory requests have three steps:

1. The memory ACTIVATE phase, which "opens" a DRAM memory row, making it ready for subsequent memory accesses.
2. The memory READ or WRITE phase, where multiple accesses can be made to individual words within the currently open DRAM row or to multiple sequential words within the current DRAM row using a burst mode.
3. The PRECHARGE phase, which "closes" the current DRAM memory row, and prepares the DRAM memory for the next ACTIVATE command.

The secret to the Core i7's pipelined memory accesses is that DDR3 DRAMs are organized with multiple **banks** within the DRAM chip. A bank is a block of DRAM memory, which may be accessed in parallel with other DRAM memory

banks, even if they are contained in the same chip. A typical DDR3 DRAM chip will have as many as 8 banks of DRAM. However, the DDR3 interface specification allows only up to four concurrent accesses on a single DDR3 channel. The timing diagram in Fig. 3-46 illustrates the Core i7 making 4 memory accesses to three distinct DRAM banks. The accesses are fully overlapped, such that the DRAM reads occur in parallel within the DRAM chip. The figure shows which commands lead to later operations through the use of arrows in the timing diagram.

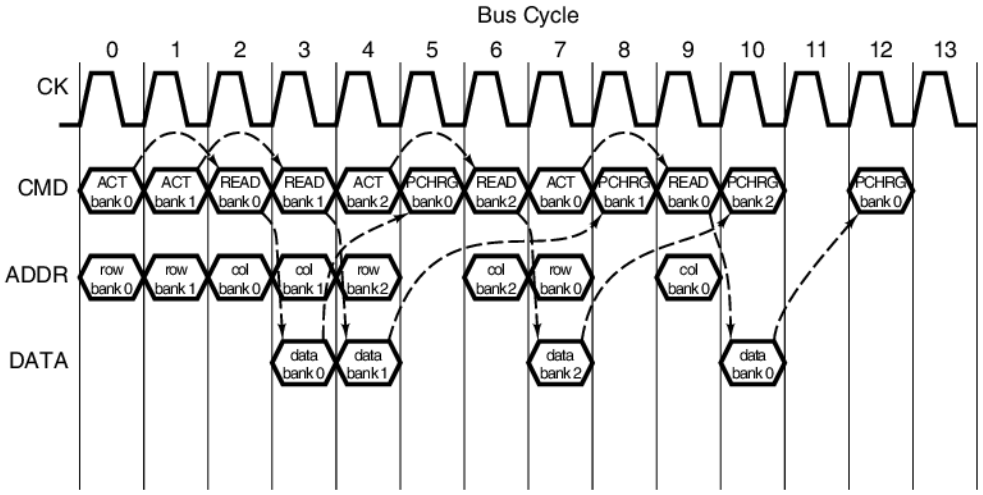


Figure 3-46. Pipelining memory requests on the Core i7's DDR3 interface.

As shown in Fig. 3-46, the DDR3 memory interface has four primary signal paths: bus clock (CK), bus command (CMD), address (ADDR), and data (DATA). The bus clock signal CK orchestrates all bus activity. The bus command CMD indicates what activity is requested of the connect DRAM. The ACTIVATE command specifies the address of the DRAM row to open via the ADDR signal. When a READ is executed, the DRAM column address is given via the ADDR signals, and the DRAM produces the read value a fixed time later on the DATA signals. Finally, the PRECHARGE command indicates the bank to precharge via the ADDR signals. For the purpose of the example, the ACTIVATE command must precede the first READ to the same bank by two DDR3 bus cycles, and data are produced one bus cycle after the READ command. Additionally, the PRECHARGE operation must occur at least two bus cycles after the last READ operation to the same DRAM bank.

The parallelism in the memory requests can be seen in the overlapping of the READ requests to the different DRAM banks. The first two READ accesses to banks 0 and 1 are completely overlapped, producing results in bus cycles 3 and 4, respectively. The access to bank 2 partially overlaps with the first access of bank 1, and finally the second read of bank 0 partially overlaps with the access to bank 2.

You might be wondering how the Core i7 knows when to expect its READ command data to return, and when it can make a new memory request. The answer is that it knows when to receive and initiate requests because it fully models the internal activities of each attached DDR3 DRAM. Thus, it will anticipate the return of data in the correct cycle, and it will know to avoid initiating a precharge operation until two cycles after its last read operation. The Core i7 can anticipate all of these activities because the DDR3 memory interface is a **synchronous memory interface**. Thus, all activities take a well-known number of DDR3 bus cycles. Even with all of this knowledge, building a high-performance fully pipelined DDR3 memory interface is a nontrivial task, requiring many internal timers and conflict detectors to implement efficient DRAM request handling.

3.5.2 The Texas Instruments OMAP4430 System-on-a-Chip

As our second example of a CPU chip, we will now examine the Texas Instruments (TI) OMAP4430 **system-on-a-chip** (**SoC**). The OMAP4430 implements the ARM instruction set, and it is targeted at mobile and embedded applications such as smartphones, tablets, and Internet appliances. Aptly named, a system-on-a-chip incorporates a wide range of devices such that the SoC combined with physical peripherals (touchscreen, flash memory, etc.) implements a complete computing device.

The OMAP4430 SoC includes two ARM A9 cores, additional accelerators, and a wide range of peripheral interfaces. The internal organization of the OMAP4430 is shown in Fig. 3-47. The ARM A9 cores are 2-wide superscalar microarchitectures. In addition, there are three more accelerator processors on the OMAP4430 die: a POWERVR SGX540 graphics processor, an image signal processor (ISP), and an IVA3 video processor. The SGX540 provides efficient programmable 3D rendering, similar to the GPUs found in desktop PCs, albeit smaller and slower. The ISP is a programmable processor designed for efficient image manipulation, for the type of operations that would be required in a high-end digital camera. The IVA3 implements efficient video encoding and decoding, with enough performance to support 3D applications like those found in handheld game consoles. Also included in the OMAP4430 SoC is a wide range of peripheral interfaces, including a touchscreen and keypad controllers, DRAM and flash interfaces, USB, and HDMI. Texas Instruments has detailed the roadmap for the OMAP series of CPUs. Future designs will have more of everything—more ARM cores, more GPUs, and more diverse peripherals.

The OMAP4430 SoC was introduced in early 2011 with two ARM A9 cores running at 1 GHz using a 45-nanometer silicon implementation. A key aspect of the OMAP4430 design is that it performs significant amounts of computation with very little power, since it is targeted to mobile applications that are powered by a battery. In battery-powered mobile applications, the more efficiently the design operates, the longer the user can go between battery charges.

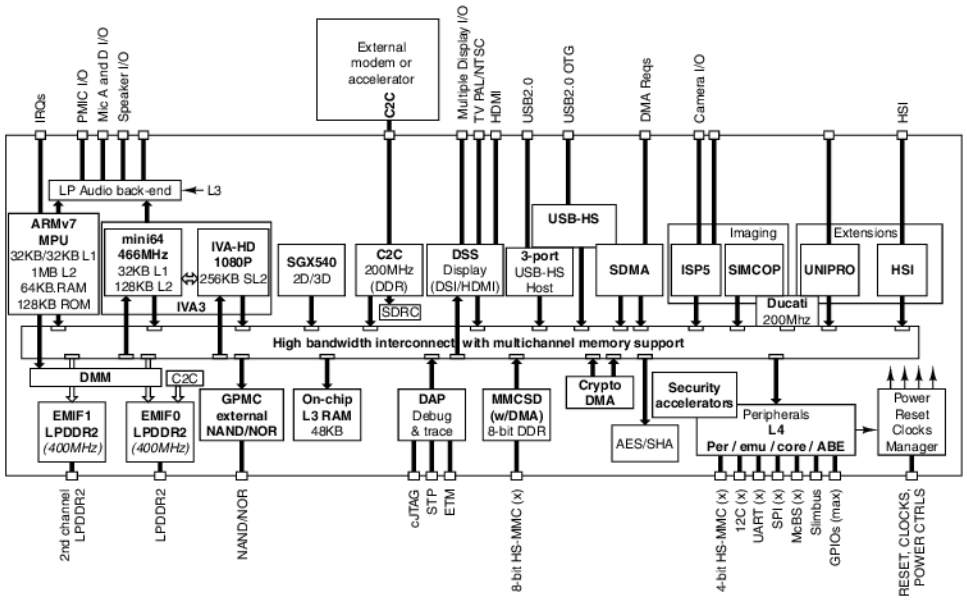


Figure 3-47. The internal organization of the OMAP4430 system-on-a-chip.

The many processors of the OMAP4430 are incorporated specifically to support its mission of low-power operation. The graphics processor, ISP, and IVA3 are all programmable accelerators that provide efficient computation capabilities at significantly less energy compared to the same tasks running on the ARM A9 CPUs alone. Fully powered, the OMAP4430 SoC draws only 600 mW of power. Compared to a high-end Core i7, the OMAP4430 uses about 1/250 as much power. The OMAP4430 also implements a very efficient sleep mode; when all components are asleep, the design draws only 100 μW. Efficient sleep modes are crucial to mobile applications with long periods of standby time, such as a cell phone. The less energy used in sleep mode, the longer the cell phone can stay in standby mode.

To further reduce power demands of the OMAP4430, the design incorporates a variety of power-management facilities, including **dynamic voltage scaling** and **power gating**. Dynamic voltage scaling allows components to run slower at a lower voltage, which significantly reduces power requirements. If you do not need the CPU's most blazing speed for computation, the voltage of the design can be lowered to run the CPU at a slower speed and much energy will be saved. Power gating is an even more aggressive power-management technique where a component is powered down completely when it is not in use, thereby eliminating its power draw. For example in a tablet application, if the user is not watching a movie, the IVA3 video processor is completely powered down and draws no power. Conversely, when the user is watching a movie, the IVA3 video processor is speeding through its video decoding tasks, while the two ARM A9 CPUs are asleep.

Despite its bent for joule-frugal operation, the ARM A9 cores utilize a very capable microarchitecture. They can decode and execute up to two instructions each cycle. As we will learn in Chap. 4, this execution rate represents the maximum throughput of the microarchitecture. But do not expect it to execute this many instructions each cycle. Rather, think of this rate as the manufacturer's guaranteed maximum performance, a level that the processor will never exceed, no matter what. In many cycles, fewer than two instructions will execute due to the myriad of "hazards" that can stall instructions, leading to lower execution throughput. To address many of these throughput limiters, the ARM A9 incorporates a powerful branch predictor, out-of-order instruction scheduling, and a highly optimized memory system.

The OMAP4430's memory system has two main internal L1 caches for each ARM A9 processor: a 32-KB cache for instructions and a 32-KB cache for data. Like the Core i7, it also uses an on-chip level 2 (L2) cache, but unlike the Core i7, it is a relatively tiny 1 MB in size, and it is shared by both ARM A9 cores. The caches are fed with dual LPDDR2 low-power DRAM channels. LPDDR2 is derived from the DDR2 memory interface standard, but changed to require fewer wires and to operate at more power-efficient voltages. Additionally, the memory controller incorporates a number of memory-access optimizations, such as tiled memory prefetching and in-memory rotation support.

While we will discuss caching in detail in Chap. 4, a few words about it here will be useful. All of main memory is divided up into cache lines (blocks) of 32 bytes. The 1024 most heavily used instruction lines and the 1024 most heavily used data lines are in the level 1 cache. Cache lines that are heavily used but which do not fit in the level 1 cache are kept in the level 2 cache. This cache contains both data lines and instruction lines from both ARM A9 CPUs mixed at random. The level 2 cache contains the most recently touched 32,768 lines in main memory.

On a level 1 cache miss, the CPU sends the identifier of the line it is looking for (Tag address) to the level 2 cache. The reply (Tag data) provides the information for the CPU to tell whether the line is in the level 2 cache, and if so, what state it is in. If the line is cached there, the CPU goes and gets it. Getting a value out of the level 2 cache takes 19 cycles. This is a long time to wait for data, so clever programmers will optimize their programs to use less data, making it more likely to find data in the fast level 1 cache.

If the cache line is not in the level 2 cache, it must be fetched from main memory via the LPDDR2 memory interface. The OMAP4430 LPDDR2 interface is implemented on-chip such that LPDDR2 DRAM can be connected directly to the OMAP4430. To access memory, the CPU must first send the upper portion of the DRAM address to the DRAM chip, using the 13 address lines. This operation, called an *ACTIVATE*, loads an entire row of memory within the DRAM into a row buffer. Subsequently, the CPU can issue multiple READ or $\overline{\text{WRITE}}$ commands, sending the remainder of the address on the same 13 address lines, and sending (or receiving) the data for the operation on the 32 data lines.

While waiting for the results, the CPU may well be able to continue with other work. For example, a cache miss while prefetching an instruction does not inhibit the execution of one or more instructions already fetched, each of which may refer to data not in any cache. Thus, multiple transactions on the two LPDDR2 interfaces may be outstanding at once, even for the same processor. It is up to the memory controller to keep track of all this and to make actual memory requests in the most efficient order.

When data finally arrives from the memory, it can come in 4 bytes at a time. A memory operation may utilize a burst mode read or write, which will allow multiple contiguous addresses within the same DRAM row to be read or written. This mode is particularly efficient for reading or writing cache blocks. Just for the record, the description of the OMAP4430 given above, like that of the Core i7 before it, has been highly simplified, but the essence of its operation has been described.

The OMAP4430 comes in a 547-pin **ball grid array** (PBGA), as shown in Fig. 3-48. A ball grid array is similar to a land grid array except that the connections on the chip are small metal balls, rather than the square pads used in the LGA. The two packages are not compatible, providing further evidence that you cannot stick a square peg into a round hole. The OMAP4430's package consists of a rectangular array of 28×26 balls, with two inner rings of balls missing, plus two asymmetric half rows and columns of balls missing to prevent the chip from being inserted incorrectly in the BGA socket.

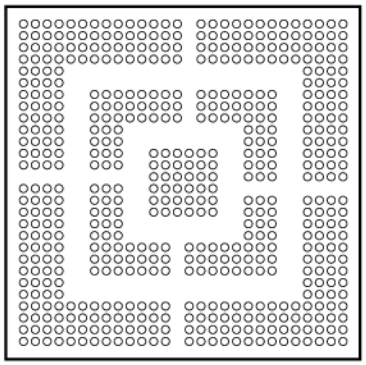


Figure 3-48. The OMAP4430 system-on-a-chip pinout.

It is difficult to compare a CISC chip (like the Core i7) and a RISC chip (like the OMAP4430) based on clock speed alone. For example, the two ARM A9 cores in the OMAP4430 have a peak execution speed of four instructions per clock cycle, giving it almost the same execution rate as that of the Core i7's 4-wide superscalar processors. The Core i7 achieves faster program execution, however, since it has up to six processors running with a clock speed 3.5 times faster (3.5 GHz) than the OMAP4430. The OMAP4430 may seem like a turtle running next

to the Core i7 rabbit, but the turtle uses much less power, and the turtle may finish first, especially if the rabbit's battery is not very big.

3.5.3 The Atmel ATmega168 Microcontroller

Both the Core i7 and the OMAP4430 are examples of high-performance computing platforms designed for building highly capable computing devices, with the Core i7 focusing on desktop applications while the OMAP4430 focuses on mobile applications. When many people think about computers, systems like these come to mind. However, another whole world of computers exists that is actually far more pervasive: embedded systems. In this section we will take a brief look at that world.

It is probably only a slight exaggeration to say that every electrical device costing more than $100 has a computer in it. Certainly televisions, cell phones, electronic personal organizers, microwave ovens, camcorders, VCRs, laser printers, burglar alarms, hearing aids, electronic games, and other devices too numerous to mention are all computer controlled these days. The computers inside these things tend to be optimized for low price rather than for high performance, which leads to different trade-offs than the high-end CPUs we have been studying so far.

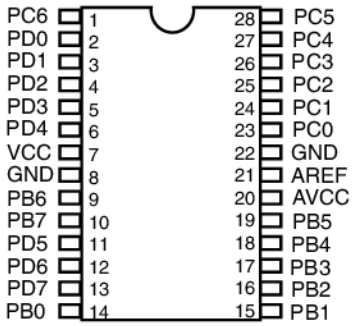


Figure 3-49. Physical pinout of the ATmega168.

As we mentioned in Chap. 1, the Atmel ATmega168 microcontroller is widely used, mostly due to its very low cost (about $1). As we will see shortly, it is also a versatile chip, which makes interfacing to it simple and inexpensive. So let us now examine this chip, whose physical pinout is shown in Fig. 3-49.

As can be seen from the figure, the ATmega168 normally comes in a standard 28-pin package (although other packages are available). At first glance, you probably noticed that the pinout on this chip is a bit strange compared to the previous two designs we examined. In particular, this chip has no address and data lines. This is because the chip is not designed to be connected to memory, only to devices. All of the memory, SRAM and flash, is contained within the processor, obviating the need for any address and data pins as shown in Fig. 3-50.

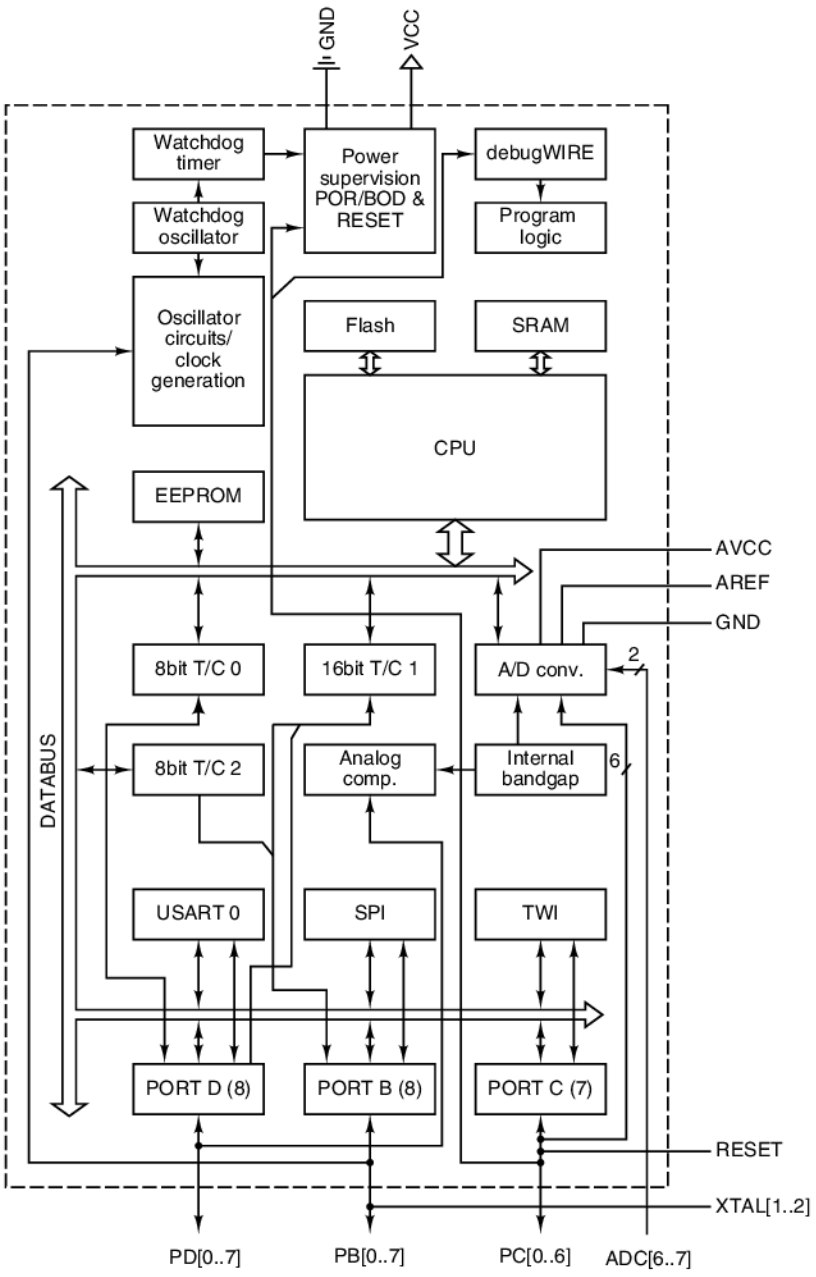


Figure 3-50. The internal architecture and logical pinout of the ATmega168.

Instead of address and data pins, the ATmega168 has 27 digital I/O ports, 8 lines in port B and D, and 7 lines in port C. These digital I/O lines are designed to be connected to I/O peripherals, and each line can be internally configured by startup software to be an input or an output. For example, when used in a microwave oven, one digital I/O line would be an input from the "door open" sensor. Another digital I/O line would be an output used to turn the microwave generator on and off. Software in the ATmega168 would check that the door was closed before turning on the microwave generator. If the door is suddenly opened, the software must kill the power. In practice, hardware interlocks are always present, too.

Optionally, six of the inputs from port C can be configured to be analog I/O. Analog I/O pins can read the voltage level of an input or set the voltage level of an output. Extending our microwave oven example, some ovens have a sensor that allows the user to heat food to a given temperature. The temperature sensor would be connected to a C port input, and software could read the voltage of the sensor and then convert it to a temperature using a sensor-specific translation function. The remaining pins on the ATmega168 are the power input (VCC), two ground pins (GND), and two pins to configure the analog I/O circuitry (AREF, AVCC).

The internal architecture of the ATmega168, like that of the OMAP4430, is a system-on-a-chip with a rich array of internal devices and memory. The ATmega168 comes with up to 16 KB of internal flash memory, for storage of infrequently changing nonvolatile information such as program instructions. It also includes up to 1 KB of EEPROM, which is nonvolatile memory that can be written by software. The EEPROM stores system-configuration data. Again, using our microwave example, the EEPROM would store a bit indicating whether the microwave displayed time in 12- or 24-hour format. The ATmega168 also incorporates up to 1 KB of internal SRAM, where software can store temporary variables.

The internal processor runs the AVR instruction set, which is composed of 131 instructions, each 16 bits in length. The processor is an 8-bit processor, which means that it operates on 8-bit data values, and internally its registers are 8 bits in size. The instruction set incorporates special instructions that allow the 8-bit processor to efficiently operate on larger data types. For example, to perform 16-bit or larger additions, the processor provides the "add-with-carry" instruction, which adds two values plus the carry out of the previous addition. The remaining internal components include the real-time clock and a variety of interface logic, including support for serial links, pulse-width-modulated (PWM) links, I2C (Inter-IC bus) link, and analog and digital I/O controllers.

3.6 EXAMPLE BUSES

Buses are the glue that hold computer systems together. In this section we will take a close look at some popular buses: the PCI bus and the Universal Serial Bus. The PCI bus is the primary I/O peripheral bus used today in PCs. It comes in two

forms, the older PCI bus, and the new and much faster PCI Express (PCIe) bus. The Universal Serial Bus is an increasingly popular I/O bus for low-speed peripherals such as mice and keyboards. A second and third version of the USB bus run at much higher speeds. In the following sections, we will look at each of these buses in turn to see how they work.

3.6.1 The PCI Bus

On the original IBM PC, most applications were text based. Gradually, with the introduction of Windows, graphical user interfaces came into use. None of these applications put much strain on early system buses such as the ISA bus. However, as time went on and many applications, especially multimedia games, began to use computers to display full-screen, full-motion video, the situation changed radically.

Let us make a simple calculation. Consider a 1024×768 color video with 3 bytes/pixel. One frame contains 2.25 MB of data. For smooth motion, at least 30 screens/sec are needed, for a data rate of 67.5 MB/sec. In fact, it is worse than this, since to display a video from a hard disk, CD-ROM, or DVD, the data must pass from the disk drive over the bus to the memory. Then for the display, the data must travel over the bus again to the graphics adapter. Thus, we need a bus bandwidth of 135 MB/sec for the video alone, not counting the bandwidth that the CPU and other devices need.

The PCI bus' predecessor, the ISA bus, ran at a maximum rate of 8.33 MHz and could transfer 2 bytes per cycle, for a maximum bandwidth of 16.7 MB/sec. The enhanced ISA bus, called the EISA bus, could move 4 bytes per cycle, to achieve 33.3 MB/sec. Clearly, neither of these approached what is needed for full-screen video.

With modern full HD video the situation is even worse. It requires 1920×1080 frames at 30 frames/sec for a data rate of 155 MB/sec (or 310 MB/sec if the data have to traverse the bus twice). Clearly, the EISA bus does not even come close to handling this.

In 1990, Intel saw this coming and designed a new bus with a far higher bandwidth than the EISA bus. It was called the **PCI bus** (**Peripheral Component Interconnect bus**). To encourage its use, Intel patented the PCI bus and then put all the patents into the public domain, so any company could build peripherals for it without having to pay royalties. Intel also formed an industry consortium, the PCI Special Interest Group, to manage the future of the PCI bus. As a result, the PCI bus became extremely popular. Virtually every Intel-based computer since the Pentium has a PCI bus, and many other computers do, too. The PCI bus is covered in gory detail in Shanley and Anderson (1999) and Solari and Willse (2004).

The original PCI bus transferred 32 bits per cycle and ran at 33 MHz (30-nsec cycle time) for a total bandwidth of 133 MB/sec. In 1993, PCI 2.0 was introduced,

and in 1995, PCI 2.1 came out. PCI 2.2 has features for mobile computers (mostly for saving battery power). The PCI bus runs at up to 66 MHz and can handle 64-bit transfers, for a total bandwidth of 528 MB/sec. With this kind of capacity, full-screen, full-motion video is doable (assuming the disk and the rest of the system are up to the job). In any event, the PCI bus will not be the bottleneck.

Even though 528 MB/sec sounds pretty fast, the PCI bus still had two problems. First, it was not good enough for a memory bus. Second, it was not compatible with all those old ISA cards out there. The solution Intel thought of was to design computers with three or more buses, as shown in Fig. 3-51. Here we see that the CPU can talk to the main memory on a special memory bus, and that an ISA bus can be connected to the PCI bus. This arrangement met all requirements, and as a consequence it was widely used in the 1990s.

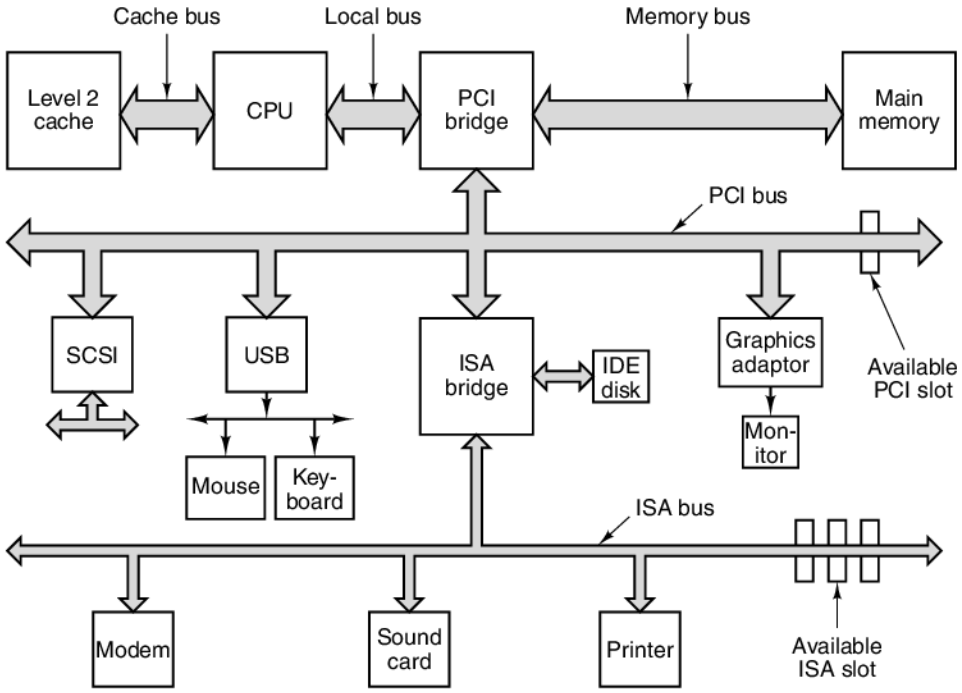


Figure 3-51. Architecture of an early Pentium system. The thicker buses have more bandwidth than the thinner ones but the figure is not to scale.

Two key components in this architecture are the two bridge chips (which Intel manufactures—hence its interest in this whole project). The PCI bridge connects the CPU, memory, and PCI bus. The ISA bridge connects the PCI bus to the ISA bus and also supports one or two IDE disks. Nearly all PC systems using this architecture would have one or more free PCI slots for adding new high-speed peripherals, and one or more ISA slots for adding low-speed peripherals.

The big advantage of the design of Fig. 3-51 is that the CPU has an extremely high bandwidth to memory using a proprietary memory bus; the PCI bus offers high bandwidth for fast peripherals such as SCSI disks, graphics adaptors, etc.; and old ISA cards can still be used. The USB box in the figure refers to the Universal Serial Bus, which will be discussed later in this chapter.

It would have been nice had there been only one kind of PCI card. Unfortunately, such is not the case. Options are provided for voltage, width, and timing. Older computers often use 5 volts and newer ones tend to use 3.3 volts, so the PCI bus supports both. The connectors are the same except for two bits of plastic that are there to prevent people from inserting a 5-volt card in a 3.3-volt PCI bus or vice versa. Fortunately, universal cards exist that support both voltages and can plug into either kind of slot. In addition to the voltage option, cards come in 32-bit and 64-bit versions. The 32-bit cards have 120 pins; the 64-bit cards have the same 120 pins plus an additional 64. A PCI bus system that supports 64-bit cards can also take 32-bit cards, but the reverse is not true. Finally, PCI buses and cards can run at either 33 or 66 MHz. The choice is made by having one pin wired either to the power supply or to ground. The connectors are identical for both speeds.

By the late 1990s, pretty much everyone agreed that the ISA bus was dead, so new designs excluded it. By then, however, monitor resolution had increased in some cases to 1600×1200 and the demand for full-screen full motion video had also increased, especially in the context of highly interactive games, so Intel added yet another bus just to drive the graphics card. This bus was called the **AGP bus** (**Accelerated Graphics Port bus**). The initial version, AGP 1.0, ran at 264 MB/sec, which was defined as 1x. While slower than the PCI bus, it was dedicated to driving the graphics card. Over the years, newer versions came out, with AGP 3.0 running at 2.1 GB/sec (8x). Today, even the high-performance AGP 3.0 bus has been usurped by even faster upstarts, in particular, the PCI Express bus, which can pump an amazing 16 GB/sec of data over high-speed serial bus links. A modern Core i7 system is illustrated in Fig. 3-52.

In a modern Core i7 based system, a number of interfaces have been integrated directly into the CPU chip. The two DDR3 memory channels, running at 1333 transactions/sec, connect to main memory and provide an aggregate bandwidth of 10 GB/sec per channel. Also integrated into the CPU is a 16-lane PCI Express channel, which optimally can be configured into a single 16-bit PCI Express bus or dual independent 8-bit PCI Express buses. The 16 lanes together provide a bandwidth of 16 GB/sec to I/O devices.

The CPU connects to the primary bridge chip, the P67, via the 20-Gb/sec (2.5 GB/sec) serial direct media interface (DMI). The P67 provides interfaces to a number of modern high-performance I/O interfaces. Eight additional PCI Express lanes are provided, plus SATA disk interfaces. The P67 also implements 14 USB 2.0 interfaces, 10G Ethernet and an audio interface.

The ICH10 chip provides legacy interface support for old devices. It is connected to the P67 via a slower DMI interface. The ICH10 implements the PCI bus,

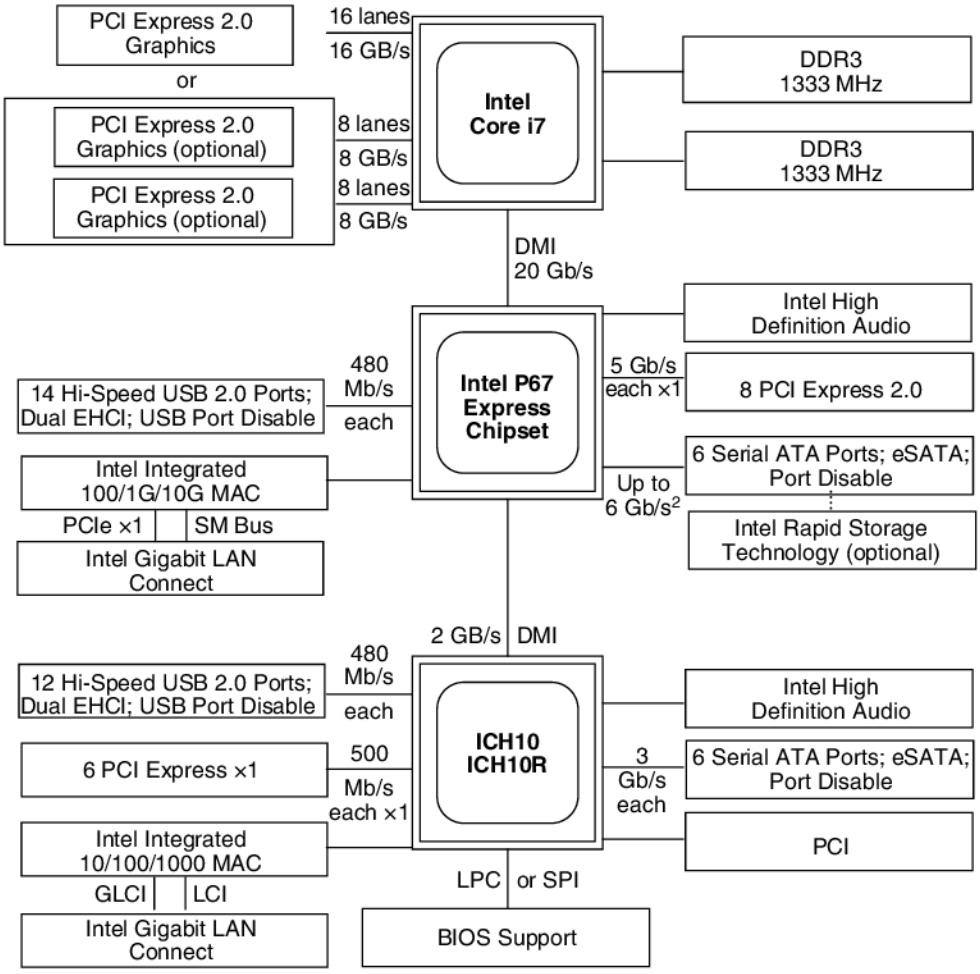


Figure 3-52. The bus structure of a modern Core i7 system.

1G Ethernet, USB ports, and old-style PCI Express and SATA interfaces. Newer systems may not incorporate the ICH10; it is required only if the system needs to support legacy interfaces.

PCI Bus Operation

Like all PC buses going back to the original IBM PC, the PCI bus is synchronous. All transactions on the PCI bus are between a master, officially called the **initiator**, and a slave, officially called the **target**. To keep the PCI pin count

down, the address and data lines are multiplexed. In this way, only 64 pins are needed on PCI cards for address plus data signals, even though PCI supports 64-bit addresses and 64-bit data.

The multiplexed address and data pins work as follows. On a read operation, during cycle 1, the master puts the address onto the bus. On cycle 2, the master removes the address and the bus is turned around so the slave can use it. On cycle 3, the slave outputs the data requested. On write operations, the bus does not have to be turned around because the master puts on both the address and the data. Nevertheless, the minimum transaction is still three cycles. If the slave is not able to respond in three cycles, it can insert wait states. Block transfers of unlimited size are also allowed, as well as several other kinds of bus cycles.

PCI Bus Arbitration

To use the PCI bus, a device must first acquire it. PCI bus arbitration uses a centralized bus arbiter, as shown in Fig. 3-53. In most designs, the bus arbiter is built into one of the bridge chips. Every PCI device has two dedicated lines running from it to the arbiter. One line, REQ#, is used to request the bus. The other line, GNT#, is used to receive bus grants. Note: REQ# is PCI-speak for $\overline{\text{REQ}}$.

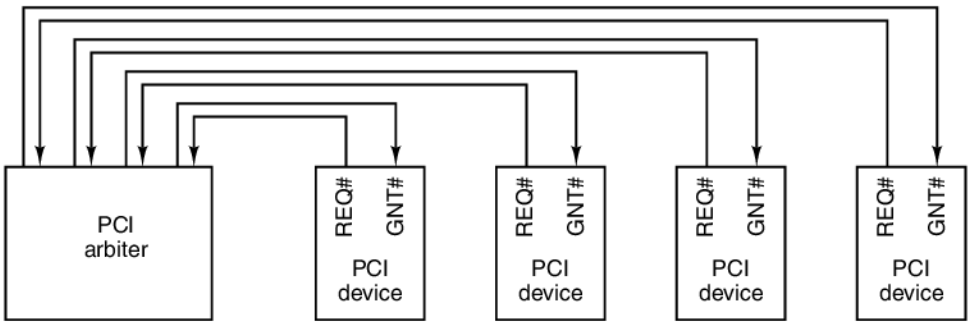


Figure 3-53. The PCI bus uses a centralized bus arbiter.

To request the bus, a PCI device (including the CPU) asserts REQ# and waits until it sees its GNT# line asserted by the arbiter. When that event happens, the device can use the bus on the next cycle. The algorithm used by the arbiter is not defined by the PCI specification. Round-robin arbitration, priority arbitration, and other schemes are all allowed. Clearly, a good arbiter will be fair, so as not to let some devices wait forever.

A bus grant is for only one transaction, although the length of this transaction is theoretically unbounded. If a device wants to run a second transaction and no other device is requesting the bus, it can go again, although often one idle cycle between transactions has to be inserted. However, under special circumstances, in

the absence of competition for the bus, a device can make back-to-back transactions without having to insert an idle cycle. If a bus master is making a very long transfer and some other device has requested the bus, the arbiter can negate the GNT# line. The current bus master is expected to monitor the GNT# line, so when it sees the negation, it must release the bus on the next cycle. This scheme allows very long transfers (which are efficient) when there is only one candidate bus master but still gives fast response to competing devices.

PCI Bus Signals

The PCI bus has a number of mandatory signals, shown in Fig. 3-54(a), and a number of optional signals, shown in Fig. 3-54(b). The remainder of the 120 or 184 pins are used for power, ground, and related miscellaneous functions and are not listed here. The *Master* (initiator) and *Slave* (target) columns tell who asserts the signal on a normal transaction. If the signal is asserted by a different device (e.g., CLK), both columns are left blank.

Let us now look briefly at each of the PCI bus signals. We will start with the mandatory (32-bit) signals, then move on to the optional (64-bit) signals. The CLK signal drives the bus. Most of the other signals are synchronous with it. A PCI bus transaction begins at the falling edge of CLK, which is in the middle of the cycle.

The 32 AD signals are for the address and data (for 32-bit transactions). Generally, during cycle 1 the address is asserted and during cycle 3 the data are asserted. The PAR signal is a parity bit for AD. The C/BE# signal is used for two different things. On cycle 1, it contains the bus command (read 1 word, block read, etc.). On cycle 2 it contains a bit map of 4 bits, telling which bytes of the 32-bit word are valid. Using C/BE# it is possible to read or write any 1, 2, or 3 bytes, as well as an entire word.

The FRAME# signal is asserted by the master to start a bus transaction. It tells the slave that the address and bus commands are now valid. On a read, usually IRDY# is asserted at the same time as FRAME#. It says the master is ready to accept incoming data. On a write, IRDY# is asserted later, when the data are on the bus.

The IDSEL signal relates to the fact that every PCI device must have a 256-byte configuration space that other devices can read (by asserting IDSEL). This configuration space contains properties of the device. The plug-and-play feature of some operating systems uses the configuration space to find out what devices are on the bus.

Now we come to signals asserted by the slave. The first of these, DEVSEL#, announces that the slave has detected its address on the AD lines and is prepared to engage in the transaction. If DEVSEL# is not asserted within a certain time limit, the master times out and assumes the device addressed is either absent or broken.

The second slave signal is TRDY#, which the slave asserts on reads to announce that the data are on the AD lines and on writes to announce that it is prepared to accept data.

Signal	Lines	Master	Slave	Description
CLK	1			Clock (33 or 66 MHz)
AD	32	×	×	Multiplexed address and data lines
PAR	1	×		Address or data parity bit
C/BE	4	×		Bus command/bit map for bytes enabled
FRAME#	1	×		Indicates that AD and C/BE are asserted
IRDY#	1	×		Read: master will accept; write: data present
IDSEL	1	×		Select configuration space instead of memory
DEVSEL#	1		×	Slave has decoded its address and is listening
TRDY#	1		×	Read: data present; write: slave will accept
STOP#	1		×	Slave wants to stop transaction immediately
PERR#	1			Data parity error detected by receiver
SERR#	1			Address parity error or system error detected
REQ#	1			Bus arbitration: request for bus ownership
GNT#	1			Bus arbitration: grant of bus ownership
RST#	1			Reset the system and all devices

(a)

Signal	Lines	Master	Slave	Description
REQ64#	1	×		Request to run a 64-bit transaction
ACK64#	1		×	Permission is granted for a 64-bit transaction
AD	32	×		Additional 32 bits of address or data
PAR64	1	×		Parity for the extra 32 address/data bits
C/BE#	4	×		Additional 4 bits for byte enables
LOCK	1	×		Lock the bus to allow multiple transactions
SBO#	1			Hit on a remote cache (for a multiprocessor)
SDONE	1			Snooping done (for a multiprocessor)
INTx	4			Request an interrupt
JTAG	5			IEEE 1149.1 JTAG test signals
M66EN	1			Wired to power or ground (66 MHz or 33 MHz)

(b)

Figure 3-54. (a) Mandatory PCI bus signals. (b) Optional PCI bus signals.

The next three signals are for error reporting. The first of these is STOP#, which the slave asserts if something disastrous happens and it wants to abort the current transaction. The next one, PERR#, is used to report a data parity error on the previous cycle. For a read, it is asserted by the master; for a write it is asserted by the slave. It is up to the receiver to take the appropriate action. Finally, SERR# is for reporting address errors and system errors.

The REQ# and GNT# signals are for doing bus arbitration. These are not asserted by the current bus master, but rather by a device that wants to become bus master. The last mandatory signal is RST#, used for resetting the system, either due to the user pushing the RESET button or some system device noticing a fatal error. Asserting this signal resets all devices and reboots the computer.

Now we come to the optional signals, most of which relate to the expansion from 32 bits to 64 bits. The REQ64# and ACK64# signals allow the master to ask permission to conduct a 64-bit transaction and allow the slave to accept, respectively. The AD, PAR64, and C/BE# signals are just extensions of the corresponding 32-bit signals.

The next three signals are not related to 32 bits vs. 64 bits, but to multiprocessor systems, something that PCI boards are not required to support. The LOCK signal allows the bus to be locked for multiple transactions. The next two relate to bus snooping to maintain cache coherence.

The INTx signals are for requesting interrupts. A PCI card can have up to four separate logical devices on it, and each one can have its own interrupt request line. The JTAG signals are for the IEEE 1149.1 JTAG testing procedure. Finally, the M66EN signal is either wired high or wired low, to set the clock speed. It must not change during system operation.

PCI Bus Transactions

The PCI bus is really very simple (as buses go). To get a better feel for it, consider the timing diagram of Fig. 3-55. Here we see a read transaction, followed by an idle cycle, followed by a write transaction by the same bus master.

When the falling edge of the clock happens during T_1, the master puts the memory address on AD and the bus command on C/BE#. It then asserts FRAME# to start the bus transaction.

During T_2, the master floats the address bus to let it turn around in preparation for the slave to drive it during T_3. The master also changes C/BE# to indicate which bytes in the word addressed it wants to enable (i.e., read in).

In T_3, the slave asserts DEVSEL# so the master knows it got the address and is planning to respond. It also puts the data on the AD lines and asserts TRDY# to tell the master that it has done so. If the slave were not able to respond so quickly, it would still assert DEVSEL# to announce its presence but keep TRDY# negated until it could get the data out there. This procedure would introduce one or more wait states.

In this example (and often in reality), the next cycle is idle. Starting in T_5 we see the same master initiating a write. It starts out by putting the address and command onto the bus, as usual. Only now, in the second cycle it asserts the data. Since the same device is driving the AD lines, there is no need for a turnaround cycle. In T_7, the memory accepts the data.

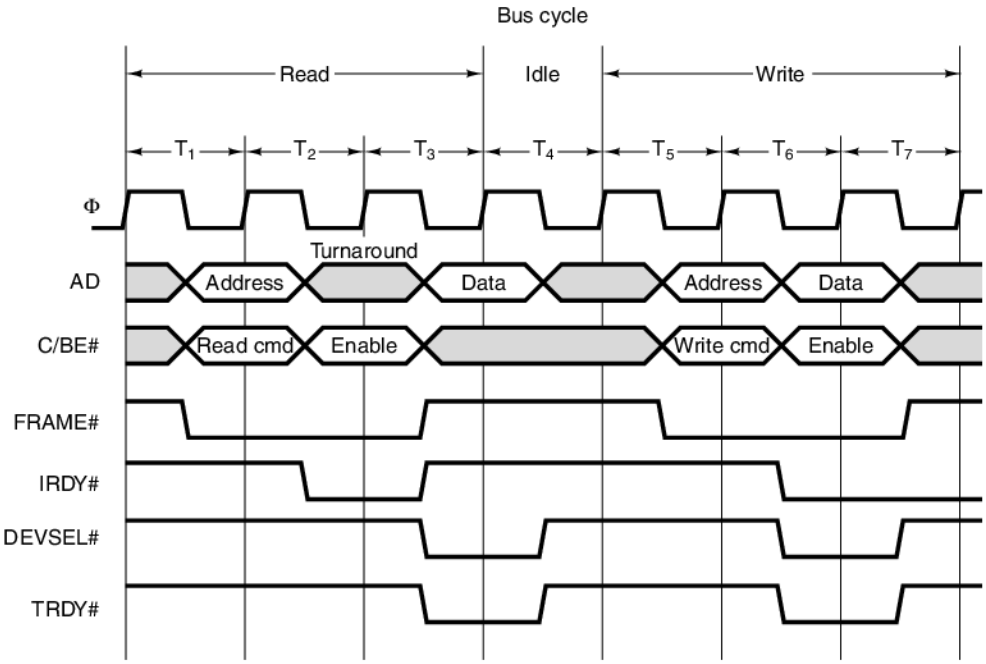


Figure 3-55. Examples of 32-bit PCI bus transactions. The first three cycles are used for a read operation, then an idle cycle, and then three cycles for a write operation.

3.6.2 PCI Express

Although the PCI bus works adequately for most current applications, the need for greater I/O bandwidth is making a mess of the once-clean internal PC architecture. In Fig. 3-52, it is clear that the PCI bus is no longer the central element that holds the parts of the PC together. The bridge chip has taken over part of that role.

The essence of the problem is that increasingly many I/O devices are too fast for the PCI bus. Cranking up the clock frequency on the bus is not a good solution because then problems with bus skew, crosstalk between the wires, and capacitance effects just get worse. Every time an I/O device gets too fast for the PCI bus (like the graphics card, hard disk, network, etc.), Intel adds a new special port to the bridge chip to allow that device to bypass the PCI bus. Clearly, this is not a long-term solution either.

Another problem with the PCI bus is that the cards are quite large. Standard PCI cards are generally 17.5 cm by 10.7 cm and low-profile cards are 12.0 cm by 3.6 cm. Neither of these fit well in laptop computers and and certainly not in mobile devices. Manufacturers would like to produce even smaller devices. Also,

some companies like to repartition the PC, with the CPU and memory in a tiny sealed box and the hard disk inside the monitor. With PCI cards, doing this is impossible.

Several solutions have been proposed, but the one that won a place in all modern PCs today (in no small part because Intel was behind it) is called **PCI Express**. It has little to do with the PCI bus and in fact is not a bus at all, but the marketing folks did not like letting go of the well-known PCI name. PCs containing it are now the standard. Let us now see how it works.

The PCI Express Architecture

The heart of the PCI Express solution (often abbreviated PCIe) is to get rid of the parallel bus with its many masters and slaves and go to a design based on high-speed point-to-point serial connections. This solution represents a radical break with the ISA/EISA/PCI bus tradition, borrowing many ideas from the world of local area networking, especially switched Ethernet. The basic idea comes down to this: deep inside, a PC is a collection of CPU, memory, and I/O controller chips that need to be interconnected. What PCI Express does is provide a general-purpose switch for connecting chips using serial links. A typical configuration is illustrated in Fig. 3-56.

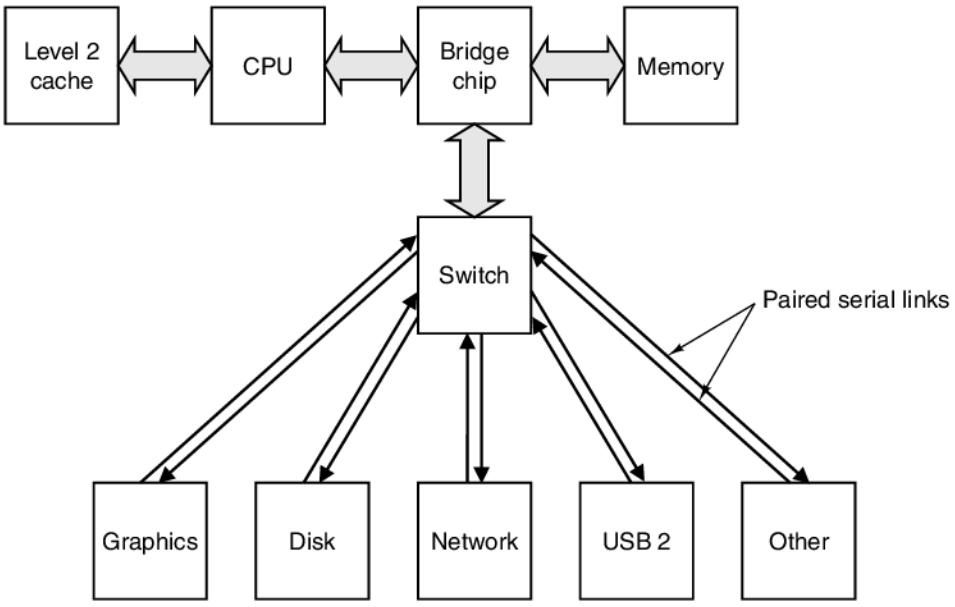


Figure 3-56. A typical PCI Express system.

As illustrated in Fig. 3-56, the CPU, memory, and cache are connected to the bridge chip in the traditional way. What is new here is a switch connected to the

bridge (possibly part of the bridge chip itself or integrated directly into the processor). Each I/O chip has a dedicated point-to-point connection to the switch. Each connection consists of a pair of unidirectional channels, one to the switch and one from it. Each channel is made up of two wires, one for the signal and one for ground, to provide high noise immunity during high-speed transmission. This architecture has replaced the old one with a much more uniform model, in which all devices are treated equally.

The PCI Express architecture differs from the old PCI bus architecture in three key ways. We have already seen two of them: a centralized switch vs. a multidrop bus and a the use of narrow serial point-to-point connections vs. a wide parallel bus. The third difference is more subtle. The conceptual model behind the PCI bus is that of a bus master issuing a command to a slave to read a word or a block of words. The PCI Express model is that of a device sending a data packet to another device. The concept of a **packet**, which consists of a header and a payload, is taken from the networking world. The **header** contains control information, thus eliminating the need for the many control signals present on the PCI bus. The **payload** contains the data to be transferred. In effect, a PC with PCI Express is a miniature packet-switching network.

In addition to these three major breaks with the past, there are also several minor differences as well. Fourth, an error-detecting code is used on the packets, providing a higher degree of reliability than on the PCI bus. Fifth, the connection between a chip and the switch is longer than it was, up to 50 cm, to allow system partitioning. Sixth, the system is expandable because a device may actually be another switch, allowing a tree of switches. Seventh, devices are hot pluggable, meaning that they can be added or removed from the system while it is running. Finally, since the serial connectors are much smaller than the old PCI connectors, devices and computers can be made much smaller. All in all, PCI Express is a major departure from the PCI bus.

The PCI Express Protocol Stack

In keeping with the model of a packet-switching network, the PCI Express system has a layered protocol stack. A **protocol** is a set of rules governing the conversation between two parties. A protocol stack is a hierarchy of protocols that deal with different issues at different layers. For example, consider a business letter. It has certain conventions about the placement and content of the letterhead, the recipient's address, the date, the salutation, the body, the signature, and so on. This might be thought of as the letter protocol. In addition, there is another set of conventions about the envelope, such as its size, where the sender's address goes and its format, where the receiver's address goes and its format, where the stamp goes, and so on. These two layers and their protocols are independent. For example, it is possible to reformat the letter but use the same envelope or vice versa. Layered

protocols make for a modular and flexible design, and have been widely used in the world of network software for decades. What is new here is building them into the "bus" hardware.

The PCI Express protocol stack is shown in Fig. 3-57(a). It is discussed below.

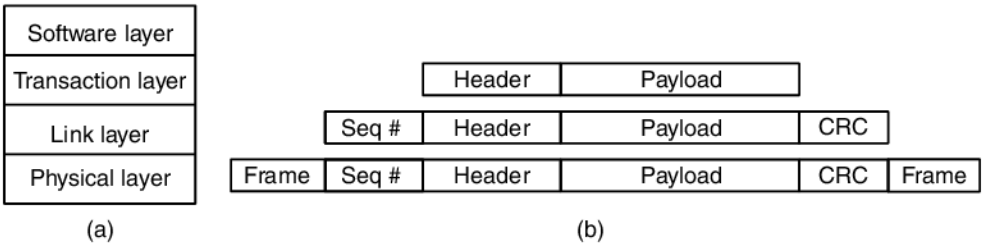


Figure 3-57. (a) The PCI Express protocol stack. (b) The format of a packet.

Let us examine the layers from the bottom up. The lowest is the **physical layer**. It deals with moving bits from a sender to a receiver over a point-to-point connection. Each point-to-point connection consists of one or more pairs of simplex (i.e., unidirectional) links. In the simplest case, there is one pair in each direction, but having 2, 4, 8, 16, or 32 pairs is also allowed. Each link is called a **lane**. The number of lanes in each direction must be the same. First-generation products must support a data rate each way of at least 2.5 Gbps, but the speed is expected to migrate to 10 Gbps each way fairly soon.

Unlike the ISA/EISA/PCI buses, PCI Express does not have a master clock. Devices are free to start transmitting as soon as they have data to send. This freedom makes the system faster but also leads to a problem. Suppose that a 1 bit is encoded as +3 volts and a 0 bit as 0 volts. If the first few bytes are all 0s, how does the receiver know data are being transmitted? After all, a run of 0 bits looks the same as an idle link. The problem is solved using what is called **8b/10b encoding**. In this scheme, 10 bits are used to encode 1 byte of actual data in a 10-bit symbol. Of the 1024 possible 10-bit symbols, the legal ones have been chosen to have enough clock transitions to keep the sender and receiver synchronized on the bit boundaries even without a master clock. A consequence of 8b/10b encoding is that a link with a gross capacity of 2.5 Gbps can carry only 2 Gbps of (net) user data.

Whereas the physical layer deals with bit transmission, the **link layer** deals with packet transmission. It takes the header and payload given to it by the transaction layer and adds to them a sequence number and an error-correcting code called a **CRC** (**Cyclic Redundancy Check**). The CRC is generated by running a certain algorithm on the header and payload data. When a packet is received, the receiver performs the same computation on the header and data and compares the result with the CRC attached to the packet. If they agree, it sends back a short **acknowledgment packet** affirming its correct arrival. If they disagree, the receiver asks for a retransmission. In this manner, data integrity is greatly improved

over the PCI bus system, which does not have any provision for verification and retransmission of data sent over the bus.

To prevent having a fast sender bury a slow receiver in packets it cannot handle, a **flow control** mechanism is used. The mechanism is that the receiver gives the transmitter a certain number of credits, basically corresponding to the amount of buffer space it has available to store incoming packets. When the credits are used up, the transmitter has to stop sending until it is given more credits. This scheme, which is widely used in all networks, prevents losing data due to a mismatch of transmitter and receiver speeds.

The **transaction layer** handles bus actions. Reading a word from memory requires two transactions: one initiated by the CPU or DMA channel requesting some data and one initiated by the target supplying the data. But the transaction layer does more than handle pure reads and writes. It adds value to the raw packet transmission offered by the link layer. To start with, it can divide each lane into up to eight **virtual circuits**, each handling a different class of traffic. The transaction layer can tag packets according to their traffic class, which may include attributes such as high priority, low priority, do not snoop, may be delivered out of order, and more. The switch may use these tags when deciding which packet to handle next.

Each transaction uses one of four address spaces:

1. Memory space (for ordinary reads and writes).
2. I/O space (for addressing device registers).
3. Configuration space (for system initialization, etc.).
4. Message space (for signaling, interrupts, etc.).

The memory and I/O spaces are similar to what current systems have. The configuration space can be used to implement features such as plug-and-play. The message space takes over the role of many of the existing control signals. Something like this space is needed because none of the PCI bus' control lines exist in PCI Express.

The **software layer** interfaces the PCI Express system to the operating system. It can emulate the PCI bus, making it possible to run existing operating systems unmodified on PCI Express systems. Of course, operating like this will not exploit the full power of PCI Express, but backward compatibility is a necessary evil that is needed until operating systems have been modified to fully utilize PCI Express. Experience shows this can take a while.

The flow of information is illustrated in Fig. 3-57(b). When a command is given to the software layer, it hands it to the transaction layer, which formulates it in terms of a header and a payload. These two parts are then passed to the link layer, which attaches a sequence number to the front and a CRC to the back. This enlarged packet is then passed on to the physical layer, which adds framing information on each end to form the physical packet that is actually transmitted. At the

receiving end, the reverse process takes place, with the link header and trailer being stripped and the result being given to the transaction layer.

The concept of each layer adding additional information to the data as it works its way down the protocol has been used for decades in the networking world with great success. The big difference between a network and PCI Express is that in the networking world the code in the various layers is nearly always software that is part of the operating system. With PCI Express it is all part of the device hardware.

PCI Express is a complicated subject. For more information see Mayhew and Krishnan (2003) and Solari and Congdon (2005). It is also still evolving. In 2007, PCIe 2.0 was released. It supports 500 MB/s per lane up to 32 lines, for a total bandwidth of 16 GB/sec. Then came PCIe 3.0 in 2011, which changed the encoding from 8b/10b to 128b/130b and can run at 8 billion transactions/sec, double PCIe 2.0.

3.6.3 The Universal Serial Bus

The PCI bus and PCI Express are fine for attaching high-speed peripherals to a computer, but they are too expensive for low-speed I/O devices such as keyboards and mice. Historically, each standard I/O device was connected to the computer in a special way, with some free ISA and PCI slots for adding new devices. Unfortunately, this scheme has been fraught with problems from the beginning.

For example, each new I/O device often comes with its own ISA or PCI card. The user is often responsible for setting switches and jumpers on the card and making sure the settings do not conflict with other cards. Then the user must open up the case, carefully insert the card, close the case, and reboot the computer. For many users, this process is difficult and error prone. In addition, the number of ISA and PCI slots is very limited (two or three typically). Plug-and-play cards eliminate the jumper settings, but the user still has to open the computer to insert the card and bus slots are still limited.

To deal with this problem, in 1993, representatives from seven companies (Compaq, DEC, IBM, Intel, Microsoft, NEC, and Northern Telecom) got together to design a better way to attach low-speed I/O devices to a computer. Since then, hundreds of other companies have joined them. The resulting standard, officially released in 1998, is called **USB** (**Universal Serial Bus**) and it is now widely implemented in personal computers. It is described further in Anderson (1997) and Tan (1997).

Some of the goals of the companies that originally conceived of the USB and started the project were as follows:

1. Users must not have to set switches or jumpers on boards or devices.
2. Users must not have to open the case to install new I/O devices.
3. There should be only one kind of cable to connect all devices.

4. I/O devices should get their power from the cable.
5. Up to 127 devices should be attachable to a single computer.
6. The system should support real-time devices (e.g., sound, telephone).
7. Devices should be installable while the computer is running.
8. No reboot should be needed after installing a new device.
9. The new bus and its I/O devices should be inexpensive to manufacture.

USB meets all these goals. It is designed for low-speed devices such as keyboards, mice, still cameras, snapshot scanners, digital telephones, and so on. Version 1.0 has a bandwidth of 1.5 Mbps, which is enough for keyboards and mice. Version 1.1 runs at 12 Mbps, which is enough for printers, digital cameras, and many other devices. Version 2.0 supports devices with up to 480 Mbps, which is sufficient to support external disk drives, high-definition webcams, and network interfaces. The recently defined USB version 3.0 pushes speeds up to 5 Gbps; only time will tell what new and bandwidth-hungry applications will spring forth from this ultra-high-bandwidth interface.

A USB system consists of a **root hub** that plugs into the main bus (see Fig. 3-51). This hub has sockets for cables that can connect to I/O devices or to expansion hubs, to provide more sockets, so the topology of a USB system is a tree with its root at the root hub, inside the computer. The cables have different connectors on the hub end and on the device end, to prevent people from accidentally connecting two hub sockets together.

The cable consists of four wires: two for data, one for power (+5 volts), and one for ground. The signaling system transmits a 0 as a voltage transition and a 1 as the absence of a voltage transition, so long runs of 0s generate a regular pulse stream.

When a new I/O device is plugged in, the root hub detects this event and interrupts the operating system. The operating system then queries the device to find out what it is and how much USB bandwidth it needs. If the operating system decides that there is enough bandwidth for the device, it assigns the new device a unique address (1–127) and downloads this address and other information to configuration registers inside the device. In this way, new devices can be added on-the-fly, without requiring any user configuration or the installation of any new ISA or PCI cards. Uninitialized cards start out with address 0, so they can be addressed. To make the cabling simpler, many USB devices contain built-in hubs to accept additional USB devices. For example, a monitor might have two hub sockets to accommodate the left and right speakers.

Logically, the USB system can be viewed as a set of bit pipes from the root hub to the I/O devices. Each device can split its bit pipe up into a maximum of 16 subpipes for different types of data (e.g., audio and video). Within each pipe or

subpipe, data flows from the root hub to the device or the other way. There is no traffic between two I/O devices.

Precisely every 1.00 ± 0.05 msec, the root hub broadcasts a new frame to keep all the devices synchronized in time. A frame is associated with a bit pipe and consists of packets, the first of which is from the root hub to the device. Subsequent packets in the frame may also be in this direction, or they may be back from the device to the root hub. A sequence of four frames is shown in Fig. 3-58.

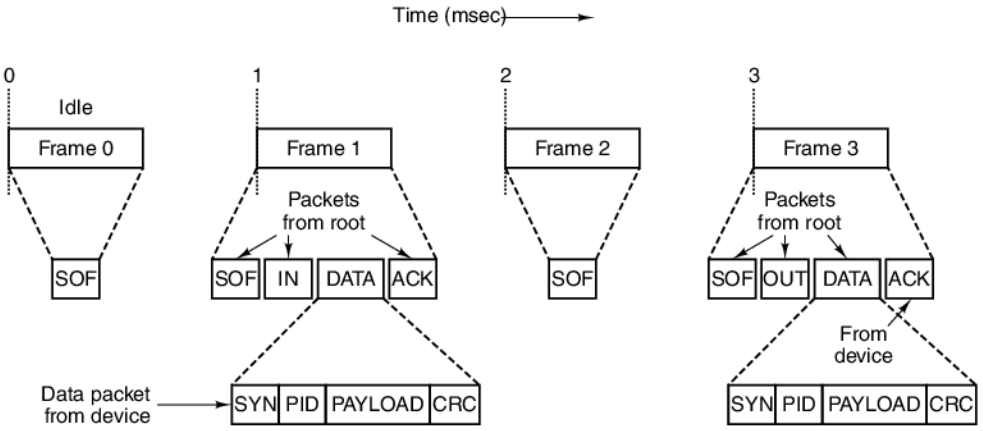


Figure 3-58. The USB root hub sends out frames every 1.00 msec.

In Fig. 3-58 there is no work to be done in frames 0 and 2, so all that is needed is one SOF (Start of Frame) packet. This packet is always broadcast to all devices. Frame 1 is a poll, for example a request to a scanner to return the bits it has found on the image it is scanning. Frame 3 consists of delivering data to some device, for example to a printer.

USB supports four kinds of frames: control, isochronous, bulk, and interrupt. Control frames are used to configure devices, give them commands, and inquire about their status. Isochronous frames are for real-time devices such as microphones, loudspeakers, and telephones that need to send or accept data at precise time intervals. They have a highly predictable delay but provide no retransmissions in the event of errors. Bulk frames are for large transfers to or from devices with no real-time requirements, such as printers. Finally, interrupt frames are needed because USB does not support interrupts. For example, instead of having the keyboard cause an interrupt whenever a key is struck, the operating system can poll it every 50 msec to collect any pending keystrokes.

A frame consists of one or more packets, possibly some in each direction. Four kinds of packets exist: token, data, handshake, and special. Token packets are from the root to a device and are for system control. The SOF, IN, and OUT packets in Fig. 3-58 are token packets. The SOF packet is the first in each frame and marks the beginning of the frame. If there is no work to do, the SOF packet is the

only one in the frame. The IN token packet is a poll, asking the device to return certain data. Fields in the IN packet tell which bit pipe is being polled so the device knows which data to return (if it has multiple streams). The OUT token packet announces that data for the device will follow. A fourth type of token packet, SETUP (not shown in the figure), is used for configuration.

Besides the token packet, three other kinds exist. These are DATA (used to transmit up to 64 bytes of information either way), handshake, and special packets. The format of a DATA packet is shown in Fig. 3-58. It consists of an 8-bit synchronization field, an 8-bit packet type (PID), the payload, and a 16-bit **CRC** (**Cyclic Redundancy Check**) to detect errors. Three kinds of handshake packets are defined: ACK (the previous data packet was correctly received), NAK (a CRC error was detected), and STALL (please wait—I am busy right now).

Now let us look at Fig. 3-58 again. Every 1.00 msec a frame must be sent from the root hub, even if there is no work. Frames 0 and 2 consist of just an SOF packet, indicating that there was no work. Frame 1 is a poll, so it starts out with SOF and IN packets from the computer to the I/O device, followed by a DATA packet from the device to the computer. The ACK packet tells the device that the data were received correctly. In case of an error, a NAK would be sent back to the device and the packet would be retransmitted for bulk data (but not for isochronous data). Frame 3 is similar in structure to frame 1, except that now the data flow is from the computer to the device.

After the USB standard was finalized in 1998, the USB designers had nothing to do so they began working on a new high-speed version of USB, called **USB 2.0**. This standard is similar to the older USB 1.1 and backward compatible with it, except that it adds a third speed, 480 Mbps, to the two existing speeds. There are also some minor differences, such as the interface between the root hub and the controller. With USB 1.1 two new interfaces were available. The first one, **UHCI** (**Universal Host Controller Interface**), was designed by Intel and put most of the burden on the software designers (read: Microsoft). The second one, **OHCI** (**Open Host Controller Interface**), was designed by Microsoft and put most of the burden on the hardware designers (read: Intel). In USB 2.0 everyone agreed to a single new interface called **EHCI** (**Enhanced Host Controller Interface**).

With USB now operating at 480 Mbps, it clearly competes with the IEEE 1394 serial bus popularly called FireWire, which runs at 400 Mbps or 800 Mbps. Because virtually every new Intel-based PC now comes with USB 2.0 or USB 3.0 (see below) 1394 is likely to vanish in due course. This disappearance is not so much due to obsolescence as to turf wars. USB is a product of the computer industry whereas 1394 comes from the consumer electronics industry. When it came to connecting cameras to computers, each industry wanted everyone to use its cable. It looks like the computer folks won this one.

Eight years after the introduction of USB 2.0, the **USB 3.0** interface standard was announced. USB 3.0 supports a whopping 5-Gbps bandwidth over the cable, although the link modulation is adaptive, and one is likely to achieve this top speed

only with professional-grade cabling. USB 3.0 devices are structurally identical to earlier USB devices, and they fully implement the USB 2.0 standard. If plugged into a USB 2.0 socket, they will operate correctly.

3.7 INTERFACING

A typical small- to medium-sized computer system consists of a CPU chip, chipset, memory chips, and some I/O devices, all connected by a bus. Sometimes, all of these devices are integrated into a system-on-a-chip, like the TI OMAP4430 SoC. We have already studied memories, CPUs, and buses in some detail. Now it is time to look at the last part of the puzzle, the I/O interfaces. It is through these I/O ports that the computer communicates with the external world.

3.7.1 I/O Interfaces

Numerous I/O interfaces are already available and new ones are being introduced all the time. Common interfaces include UARTs, USARTs, CRT controllers, disk controllers, and PIOs. A **UART** (**Universal Asynchronous Receiver Transmitter**) is an I/O interface that can read a byte from the data bus and output it a bit at a time on a serial line for a terminal, or input data from a terminal. UARTs usually allow various speeds from 50 to 19,200 bps; character widths from 5 to 8 bits; 1, 1.5, or 2 stop bits; and provide even, odd, or no parity, all under program control. **USART**s (**Universal Synchronous Asynchronous Receiver Transmitters**) can handle synchronous transmission using a variety of protocols as well as performing all the UART functions. Since UARTs have become less important as telephone modems are vanishing, let us now study the parallel interface as an example of an I/O chip.

PIO Interfaces

A typical **PIO** (**Parallel Input/Output**) interface (based on the classic Intel 8255A PIO design) is illustrated in Fig. 3-59. It has a collection of I/O lines (e.g., 24 I/O lines in the example in the figure) that can interface to any digital logic device interface, for example, keyboards, switches, lights, or printers. In a nutshell, the CPU program can write a 0 or 1 to any line, or read the input status of any line, providing great flexibility. A small CPU-based system using a PIO interface can control a variety of physical devices, such as a robot, toaster, or electron microscope. Typically, PIO interfaces are found in embedded systems.

The PIO interface is configured with a 3-bit configuration register, which specifies if the three independent 8-bit ports are to be used for digital signal input (0) or output (1). Setting the appropriate value in the configuration register will allow any combination of input and output for the three ports. Associated with

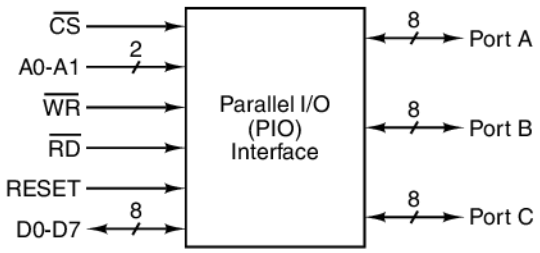


Figure 3-59. A 24-bit PIO Interface.

each port is an 8-bit latch register. To set the lines on an output port, the CPU just writes an 8-bit number into the corresponding register, and the 8-bit number appears on the output lines and stays there until the register is rewritten. To use a port configured for input, the CPU just reads the corresponding 8-bit latch register.

It is possible to build more sophisticated PIO interfaces. For example, one popular operating mode provides for handshaking with external devices. For example, to output to a device that is not always ready to accept data, the PIO can present data on an output port and wait for the device to send a pulse back saying that it has accepted the data and wants more. The necessary logic for latching such pulses and making them available to the CPU includes a ready signal plus an 8-bit register queue for each output port.

From the functional diagram of the PIO we can see that in addition to 24 pins for the three ports, it has eight lines that connect directly to the data bus, a chip select line, read and write lines, two address lines, and a line for resetting the chip. The two address lines select one of the four internal registers, corresponding to ports A, B, C, and the port configuration register. Normally, the two address lines are connected to the low-order bits of the address bus. The chip select line allows the 24-bit PIO to be combined to form larger PIO interfaces, by adding additional address lines and using them to select the proper PIO interface by asserting its chip select line.

3.7.2 Address Decoding

Up until now we have been deliberately vague about how chip select is asserted on the memory and I/O chips we have looked at. It is now time to look more carefully at how this is done. Let us consider a simple 16-bit embedded computer consisting of a CPU, a 2KB $\times$ 8 byte EPROM for the program, a $2-$KB $\times$ 8 byte RAM for the data, and a PIO interface. This small system might be used as a prototype for the brain of a cheap toy or simple appliance. Once in production, the EPROM might be replaced by a ROM.

The PIO interface can be selected in one of two ways: as a true I/O device or as part of memory. If we choose to use it as an I/O device, then we must select it

using an explicit bus line that indicates that an I/O device is being referenced, and not memory. If we use the other approach, **memory-mapped I/O**, then we must assign it 4 bytes of the memory space for the three ports and the control register. The choice is somewhat arbitrary. We will choose memory-mapped I/O because it illustrates some interesting issues in I/O interfacing.

The EPROM needs 2 KB of address space, the RAM also needs 2K of address space, and the PIO needs 4 bytes. Because our example address space is 64K, we must make a choice about where to put the three devices. One possible choice is shown in Fig. 3-60. The EPROM occupies addresses to 2K, the RAM occupies addresses 32 KB to 34 KB, and the PIO occupies the highest 4 bytes of the address space, 65532 to 65535. From the programmer's point of view, it makes no difference which addresses are used; however, for interfacing it does matter. If we had chosen to address the PIO via the I/O space, it would not need any memory addresses (but it would need four I/O space addresses).

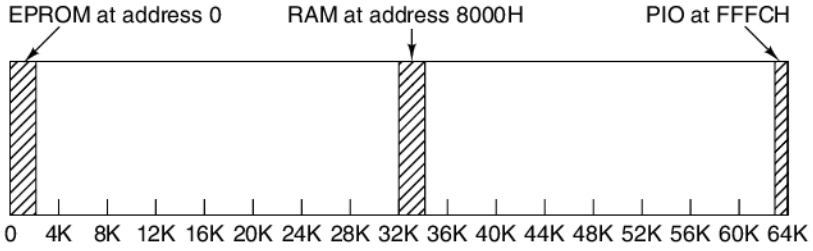


Figure 3-60. Location of the EPROM, RAM, and PIO in our 64-KB address space.

With the address assignments of Fig. 3-60, the EPROM should be selected by any 16-bit memory address of the form 00000xxxxxxxxxxx (binary). In other words, any address whose 5 high-order bits are all 0s falls in the bottom 2 KB of memory, hence in the EPROM. Thus, the EPROM's chip select could be wired to a 5-bit comparator, one of whose inputs was permanently wired to 00000.

A better way to achieve the same effect is to use a five-input OR gate, with the five inputs attached to address lines A11 to A15. If and only if all five lines are 0 will the output be 0, thus asserting $\overline{\text{CS}}$ (which is asserted low). This addressing approach is illustrated in Fig. 3-60(a) and is called full-address decoding.

The same principle can be used for the RAM. However, the RAM should respond to binary addresses of the form 10000xxxxxxxxxxx, so an additional inverter is needed as shown in the figure. The PIO address decoding is somewhat more complicated, because it is selected by the four addresses of the form 11111111111111xx. A possible circuit that asserts $\overline{\text{CS}}$ only when the correct address appears on the address bus is shown in the figure. It uses two eight-input NAND gates to feed an OR gate.

However, if the computer really consists of only the CPU, two memory chips, and the PIO, we can use a trick to greatly simplify the address decoding. The trick

is based on the fact that all EPROM addresses, and only EPROM addresses, have a 0 in the high-order bit, A15. Therefore, we can just wire $\overline{\text{CS}}$ to A15 directly, as shown in Fig. 3-61(b).

At this point the decision to put the RAM at 8000H may seem much less arbitrary. The RAM decoding can be done by noting that the only valid addresses of the form 10xxxxxxxxxxxxx are in the RAM, so 2 bits of decoding are sufficient. Similarly, any address starting with 11 must be a PIO address. The complete decoding logic is now two NAND gates and an inverter.

The address decoding logic of Fig. 3-61(b) is called **partial address decoding**, because the full addresses are not used. It has the property that a read from addresses 0001000000000000, 0001100000000000, or 0010000000000000 will give the same result. In fact, every address in the bottom half of the address space will select the EPROM. Because the extra addresses are not used, no harm is done, but if one is designing a computer that may be expanded in the future (an unlikely occurrence in a toy), partial decoding should be avoided because it ties up too much address space.

Another common address-decoding technique is to use a decoder, such as that shown in Fig. 3-13. By connecting the three inputs to the three high-order address lines, we get eight outputs, corresponding to addresses in the first 8K, second 8K, and so on. For a computer with eight RAMs, each 8K $\times$ 8, one such chip provides the complete decoding. For a computer with eight 2K $\times$ 8 memory chips, a single decoder is also sufficient, provided that the memory chips are each located in distinct 8-KB chunks of address space. (Remember our earlier remark that the position of the memory and I/O chips within the address space matters.)

3.8 SUMMARY

Computers are constructed from integrated circuit chips containing tiny switching elements called gates. The most common gates are AND, OR, NAND, NOR, and NOT. Simple circuits can be built up by directly combining individual gates.

More complex circuits are multiplexers, demultiplexers, encoders, decoders, shifters, and ALUs. Arbitrary Boolean functions can be programmed using a FPGA. If many Boolean functions are needed, FPGAs are often more efficient. The laws of Boolean algebra can be used to transform circuits from one form to another. In many cases more economical circuits can be produced this way.

Computer arithmetic is done by adders. A single-bit full adder can be constructed from two half adders. An adder for a multibit word can be built by connecting multiple full adders in such a way as to allow the carry out of each one feed into its left-hand neighbor.

The components of (static) memories are latches and flip-flops, each of which can store one bit of information. These can be combined linearly into latches and

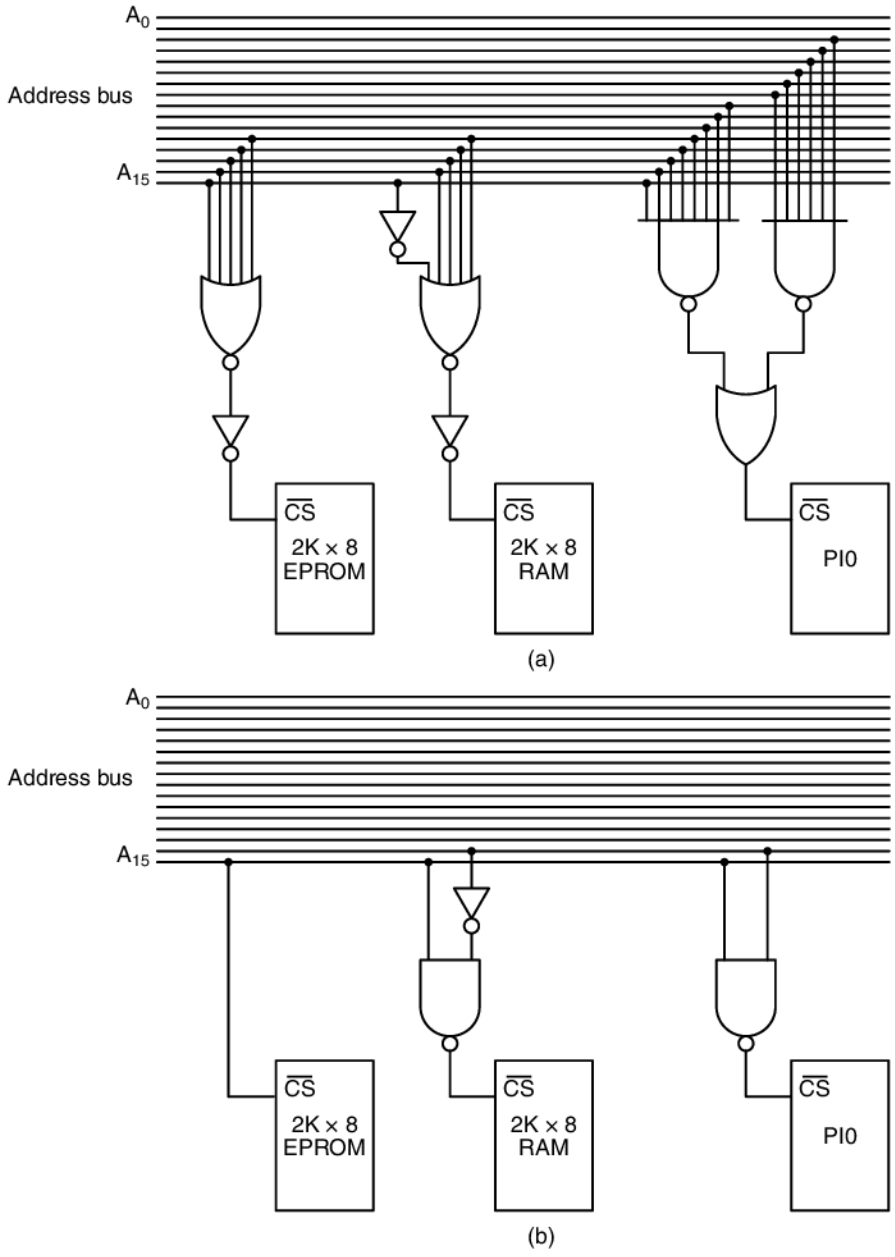


Figure 3-61. (a) Full address decoding. (b) Partial address decoding.

flip-flops for memories with any word size desired. Memories are available as RAM, ROM, PROM, EPROM, EEPROM, and flash. Static RAMs need not be refreshed; they keep their stored values as long as the power remains on. Dynamic RAMs, on the other hand, must be refreshed periodically to compensate for leakage from the little capacitors on the chip.

The components of a computer system are connected by buses. Many, but not all, of the pins on a typical CPU chip directly drive one bus line. The bus lines can be divided into address, data, and control lines. Synchronous buses are driven by a master clock. Asynchronous buses use full handshaking to synchronize the slave to the master.

The Core i7 is an example of a modern CPU. Modern systems using it have a memory bus, a PCIe bus, and a USB bus. The PCIe interconnect is the dominant way to connect the internal parts of a computer at high speeds. The ARM is also a modern high-end CPU but is intended for embedded systems and mobile devices where low power consumption is important. The Atmel ATmega168 is an example of a low-priced chip good for small, inexpensive appliances and many other price-sensitive applications.

Switches, lights, printers, and many other I/O devices can be interfaced to computers using parallel I/O interfaces. These chips can be configured to be part of the I/O space or the memory space, as needed. They can be fully decoded or partially decoded, depending on the application.

PROBLEMS

1. Analog circuits are subject to noise that can distort their output. Are digital circuits immune to noise? Discuss your answer.

2. A logician drives into a drive-in restaurant and says, "I want a hamburger or a hot dog and french fries." Unfortunately, the cook flunked out of sixth grade and does not know (or care) whether "and" has precedence over "or." As far as he is concerned, one interpretation is as good as the other. Which of the following cases are valid interpretations of the order? (Note that in English "or" means "exclusive or.")

a. Just a hamburger.
b. Just a hot dog.
c. Just french fries.
d. A hot dog and french fries.
e. A hamburger and french fries.
f. A hot dog and a hamburger.
g. All three.
h. Nothing—the logician goes hungry for being a wiseguy.

3. A missionary lost in Southern California stops at a fork in the road. He knows that two motorcycle gangs inhabit the area, one of which always tells the truth and one of which always lies. He wants to know which road leads to Disneyland. What question should he ask?

4. Use a truth table to show that $X = (X \text{ AND } Y) \text{ OR } (X \text{ AND NOT } Y)$.

5. There exist four Boolean functions of a single variable and 16 functions of two variables. How many functions of three variables are there? Of n variables?

6. There exist four Boolean functions of a single variable and 16 functions of two variables. How many functions of four variables are there?

7. Show how the AND function can be constructed from two NAND gates.

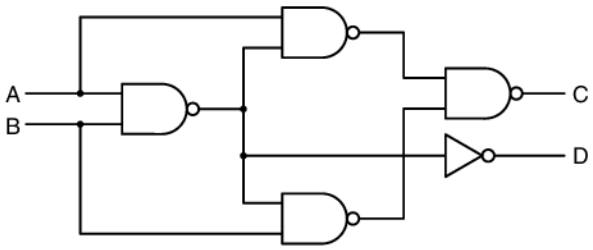
8. Using the three-variable multiplexer chip of Fig. 3-12, implement a function whose output is the parity of the inputs, that is, the output is 1 if and only if an even number of inputs are 1.

9. Put on your thinking cap. The three-variable multiplexer chip of Fig. 3-12 is actually capable of computing an arbitrary function of *four* Boolean variables. Describe how, and as an example, draw the logic diagram for the function that is 0 if the English word for the truth-table row has an even number of letters, 1 if it has an odd number of letters (e.g., 0000 = zero = four letters $\rightarrow$ 0; 0111 = seven = five letters $\rightarrow$ 1; 1101 = thirteen = eight letters $\rightarrow$ 0). *Hint*: If we call the fourth input variable D, the eight input lines may be wired to V_{cc}, ground, D, or $\bar{D}$.

10. Draw the logic diagram of a 2-bit encoder, a circuit with four input lines, exactly one of which is high at any instant, and two output lines whose 2-bit binary value tells which input is high.

11. Draw the logic diagram of a 2-bit demultiplexer, a circuit whose single input line is steered to one of the four output lines depending on the state of the two control lines.

12. What does this circuit do?



13. A common chip is a 4-bit adder. Four of these chips can be hooked up to form a 16-bit adder. How many pins would you expect the 4-bit adder chip to have? Why?

14. An n-bit adder can be constructed by cascading n full adders in series, with the carry into stage i, C_i, coming from the output of stage $i - 1$. The carry into stage 0, C_0, is 0. If each stage takes T nsec to produce its sum and carry, the carry into stage i will not be valid until iT nsec after the start of the addition. For large n the time required for the

carry to ripple through to the high-order stage may be unacceptably long. Design an adder that works faster. *Hint*: Each C_i can be expressed in terms of the operand bits A_{i-1} and B_{i-1} as well as the carry C_{i-1}. Using this relation it is possible to express C_i as a function of the inputs to stages 0 to $i-1$, so all the carries can be generated simultaneously.

15. If all the gates in Fig. 3-18 have a propagation delay of 1 nsec, and all other delays can be ignored, what is the earliest time a circuit using this design can be sure of having a valid output bit?

16. The ALU of Fig. 3-19 is capable of doing 8-bit 2's complement additions. Is it also capable of doing 2's complement subtractions? If so, explain how. If not, modify it to be able to do subtractions.

17. A 16-bit ALU is built up of 16 1-bit ALUs, each one having an add time of 10 nsec. If there is an additional 1-nsec delay for propagation from one ALU to the next, how long does it take for the result of a 16-bit add to appear?

18. Sometimes it is useful for an 8-bit ALU such as Fig. 3-19 to generate the constant -1 as output. Give two different ways this can be done. For each way, specify the values of the six control signals.

19. What is the quiescent state of the S and R inputs to an SR latch built of two NAND gates?

20. The circuit of Fig. 3-25 is a flip-flop that is triggered on the rising edge of the clock. Modify this circuit to produce a flip-flop that is triggered on the falling edge of the clock.

21. The 4×3 memory of Fig. 3-28 uses 22 AND gates and three OR gates. If the circuit were to be expanded to 256×8, how many of each would be needed?

22. To help meet the payments on your new personal computer, you have taken up consulting for fledgling SSI chip manufacturers. One of your clients is thinking about putting out a chip containing four D flip-flops, each containing both Q and $\bar{Q}$, on request of a potentially important customer. The proposed design has all four clock signals ganged together, also on request. Neither preset nor clear is present. Your assignment is to give a professional evaluation of the design.

23. As more and more memory is squeezed onto a single chip, the number of pins needed to address it also increases. It is often inconvenient to have large numbers of address pins on a chip. Devise a way to address 2^n words of memory using fewer than n pins.

24. A computer with a 32-bit wide data bus uses $1\text{M} \times 1$ dynamic RAM memory chips. What is the smallest memory (in bytes) that this computer can have?

25. Referring to the timing diagram of Fig. 3-38, suppose that you slowed the clock down to a period of 20 nsec instead of 10 nsec as shown but the timing constraints remained unchanged. How much time would the memory have to get the data onto the bus during T_3 after $\overline{\text{MREQ}}$ was asserted, in the worst case?

26. Again referring to Fig. 3-38, suppose that the clock remained at 100 MHz, but T_{DS} was increased to 4 nsec. Could 10-nsec memory chips be used?

27. In Fig. 3-38(b), T_{ML} is specified to be at least 2 nsec. Can you envision a chip in which it is negative? Specifically, could the CPU assert $\overline{\text{MREQ}}$ before the address was stable? Why or why not?

28. Assume that the block transfer of Fig. 3-42 were done on the bus of Fig. 3-38. How much more bandwidth is obtained by using a block transfer over individual transfers for long blocks? Now assume that the bus is 32 bits wide instead of 8 bits wide. Answer the question again.

29. Denote the transition times of the address lines of Fig. 3-39 as T_{A1} and T_{A2}, and the transition times of $\overline{\text{MREQ}}$ as T_{MREQ1} and T_{MREQ2}, and so on. Write down all the inequalities implied by the full handshake.

30. Multicore chips, with multiple CPUs on the same die, are becoming popular. What advantages do they have over a system consisting of multiple PCs connected by Ethernet?

31. Why have multicore chips suddenly appeared? Are there technological factors that have paved the way? Does Moore's law play a role here?

32. What is the difference between the memory bus and the PCI bus?

33. Most 32-bit buses permit 16-bit reads and writes. Is there any ambiguity about where to place the data? Discuss.

34. Many CPUs have a special bus cycle type for interrupt acknowledge. Why?

35. A 64-bit computer with a 400-MHz bus requires four cycles to read a 64-bit word. How much bus bandwidth does the CPU consume in the worst case, that is, assuming back-to-back reads or writes all the time?

36. A 64-bit computer with a 400-MHz bus requires four cycles to read a 64-bit word. How much bus bandwidth does the CPU consume in the worst case, that is, assuming back-to-back reads or writes all the time?

37. A 32-bit CPU with address lines A2–A31 requires all memory references to be aligned. That is, words have to be addressed at multiples of 4 bytes, and half-words have to be addressed at even bytes. Bytes can be anywhere. How many legal combinations are there for memory reads, and how many pins are needed to express them? Give two answers and make a case for each one.

38. Modern CPU chips have one, two, or even three levels of cache on chip. Why are multiple levels of cache needed?

39. Suppose that a CPU has a level 1 cache and a level 2 cache, with access times of 1 nsec and 2 nsec, respectively. The main memory access time is 10 nsec. If 20% of the accesses are level 1 cache hits and 60% are level 2 cache hits, what is the average access time?

40. Calculate the bus bandwidth needed to display 1280×960) color video at 30 frames/sec. Assume that the data must pass over the bus twice, once from the CD-ROM to the memory and once from the memory to the screen.

41. Which of the signals of Fig. 3-55 is not strictly necessary for the bus protocol to work?

42. A PCI Express system has 10 Mbps links (gross capacity). How many signal wires are needed in each direction for 16x operation? What is the gross capacity each way? What is the net capacity each way?

43. A computer has instructions that each require two bus cycles, one to fetch the instruction and one to fetch the data. Each bus cycle takes 10 nsec and each instruction takes 20 nsec (i.e., the internal processing time is negligible). The computer also has a disk with 2048 512-byte sectors per track. Disk rotation time is 5 msec. To what percent of its normal speed is the computer reduced during a DMA transfer if each 32-bit DMA transfer takes one bus cycle?

44. The maximum payload of an isochronous data packet on the USB bus is 1023 bytes. Assuming that a device may send only one data packet per frame, what is the maximum bandwidth for a single isochronous device?

45. What would the effect be of adding a third input line to the NAND gate selecting the PIO of Fig. 3-61(b) if this new line were connected to A13?

46. Write a program to simulate the behavior of an $m \times n$ array of two-input NAND gates. This circuit, contained on a chip, has j input pins and k output pins. The values of j, k, m, and n are compile-time parameters of the simulation. The program should start off by reading in a "wiring list," each wire of which specifies an input and an output. An input is either one of the j input pins or the output of some NAND gate. An output is either one of the k output pins or an input to some NAND gate. Unused inputs are logical 1. After reading in the wiring list, the program should print the output for each of the 2^j possible inputs. Gate array chips like this one are widely used for putting custom circuits on a chip because most of the work (depositing the gate array on the chip) is independent of the circuit to be implemented. Only the wiring is specific to each design.

47. Write a program in your favorite programming language to read in two arbitrary Boolean expressions and see if they represent the same function. The input language should include single letters, as Boolean variables, the operands AND, OR, and NOT, and parentheses. Each expression should fit on one input line. The program should compute the truth tables for both functions and compare them.

This page intentionally left blank

4

THE MICROARCHITECTURE LEVEL

Above the digital logic level is the microarchitecture level. Its job is to implement the ISA (Instruction Set Architecture) level above it, as illustrated in Fig. 1-2. Its design depends on the ISA being implemented, as well as the cost and performance goals of the computer. Many modern ISAs, particularly RISC designs, have simple instructions that can usually be executed in a single clock cycle. More complex ISAs, such as the Core i7 instruction set, may require many cycles to execute a single instruction. Executing an instruction may require locating the operands in memory, reading them, and storing results back into memory. The sequencing of operations within a single instruction often leads to a different approach to control than that for simple ISAs.

4.1 AN EXAMPLE MICROARCHITECTURE

Ideally, we would like to introduce this subject by explaining the general principles of microarchitecture design. Unfortunately, there are no general principles; every ISA is a special case. Consequently, we will discuss a detailed example instead. For our example ISA, we have chosen a subset of the Java Virtual Machine. This subset contains only integer instructions, so we have named it **IJVM** to emphasize it deals only with integers.

We will start out by describing the microarchitecture on top of which we will implement IJVM. IJVM has some complex instructions. Many such architectures

have been implemented through microprogramming, as discussed in Chap. 1. Although IJVM is small, it is a good starting point for describing the control and sequencing of instructions.

Our microarchitecture will contain a microprogram (in ROM), whose job is to fetch, decode, and execute IJVM instructions. We cannot use the Oracle JVM interpreter because we need a tiny microprogram that drives the individual gates in the actual hardware efficiently. In contrast, the Oracle JVM interpreter was written in C for portability and cannot control the hardware at all.

Since the actual hardware used consists only of the basic components described in Chap. 3, in theory, after fully understanding this chapter, the reader should be able to go out and buy a large bag full of transistors and build this subset of the JVM machine. Students who successfully accomplish this task will be given extra credit (and a complete psychiatric examination).

As a convenient model for the design of the microarchitecture we can think of it as a programming problem, where each instruction at the ISA level is a function to be called by a master program. In this model, the master program is a simple, endless loop that determines a function to be invoked, calls the function, then starts over, very much like Fig. 2-3.

The microprogram has a set of variables, called the **state** of the computer, which can be accessed by all the functions. Each function changes at least some of the variables making up the state. For example, the Program Counter (PC) is part of the state. It indicates the memory location containing the next function (i.e., ISA instruction) to be executed. During the execution of each instruction, the PC is advanced to point to the next instruction to be executed.

IJVM instructions are short and sweet. Each instruction has a few fields, usually one or two, each of which has some specific purpose. The first field of every instruction is the **opcode** (short for **operation code**), which identifies the instruction, telling whether it is an ADD or a BRANCH, or something else. Many instructions have an additional field, which specifies the operand. For example, instructions that access a local variable need a field to tell *which* variable.

This model of execution, sometimes called the **fetch-decode-execute cycle**, is useful in the abstract and may also be the basis for implementation for ISAs like IJVM that have complex instructions. Below we will describe how it works, what the microarchitecture looks like, and how it is controlled by the microinstructions, each of which controls the data path for one cycle. Together, the list of microinstructions forms the microprogram, which we will present and discuss in detail.

4.1.1 The Data Path

The **data path** is that part of the CPU containing the ALU, its inputs, and its outputs. The data path of our example microarchitecture is shown in Fig. 4-1. While it has been carefully optimized for interpreting IJVM programs, it is fairly similar to the data path used in most machines. It contains a number of 32-bit

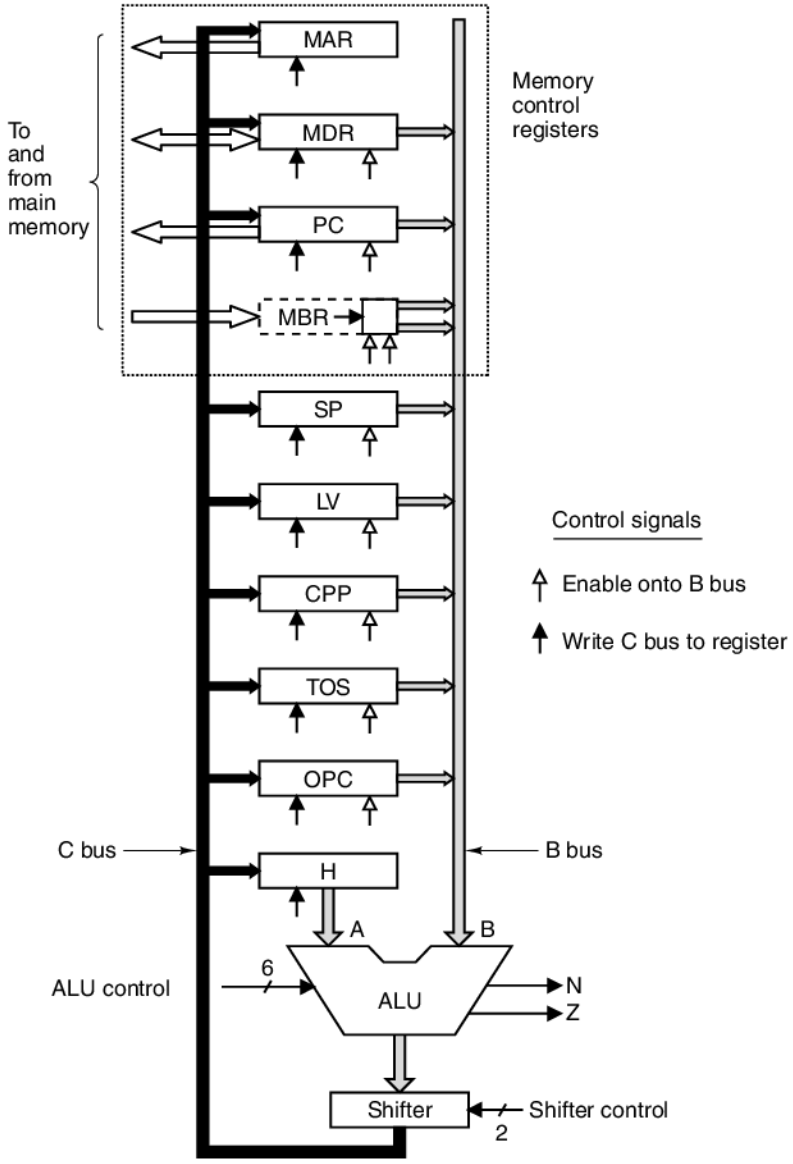


Figure 4-1. The data path of the example microarchitecture used in this chapter.

registers, to which we have assigned symbolic names such as PC, SP, and MDR. Though some of these names are familiar, it is important to understand that these registers are accessible only at the microarchitecture level (by the microprogram). They are given these names because they usually hold a value corresponding to the variable of the same name in the ISA level architecture. Most registers can drive

their contents onto the B bus. The output of the ALU drives the shifter and then the C bus, whose value can be written into one or more registers at the same time. There is no A bus for the moment; we will add one later.

The ALU is identical to the one shown in Figs. 3-18 and 3-19. Its function is determined by six control lines. The short diagonal line labeled "6" in Fig. 4-1 indicates that there are six ALU control lines. These are F_0 and F_1 for determining the ALU operation, ENA and ENB for individually enabling the inputs, INVA for inverting the left input, and INC for forcing a carry into the low-order bit, effectively adding 1 to the result. However, not all 64 combinations of ALU control lines do something useful.

Some of the more interesting combinations are shown in Fig. 4-2. Not all of these functions are needed for IJVM, but for the full JVM many of them would come in handy. In many cases, there are multiple possibilities for achieving the same result. In this table, + means arithmetic plus and − means arithmetic minus, so, for example, $-A$ means the two's complement of A.

F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\bar{A}$
1	0	1	1	0	0	$\bar{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	0	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	–1

Figure 4-2. Useful combinations of ALU signals and the function performed.

The ALU of Fig. 4-1 needs two data inputs: a left input (A) and a right input (B). Attached to the left input is a holding register, H. Attached to the right input is the B bus, which can be loaded from any one of nine sources, indicated by the nine gray arrows touching it. An alternative design, with two full buses, has a different set of trade-offs and will be discussed later in this chapter.

H can be loaded by choosing an ALU function that just passes the right input (from the B bus) through to the ALU output. One such function is adding the ALU inputs, only with ENA negated so the left input is forced to zero. Adding zero to the value on the B bus just yields the value on the B bus. This result can then be passed through the shifter unmodified and stored in H.

In addition to the above functions, two other control lines can be used independently to control the output from the ALU. SLL8 (Shift Left Logical) shifts the contents left by 1 byte, filling the 8 least significant bits with zeros. SRA1 (Shift Right Arithmetic) shifts the contents right by 1 bit, leaving the most significant bit unchanged.

It is explicitly possible to read and write the same register on one cycle. For example, it is allowed to put SP onto the B bus, disable the ALU's left input, enable the INC signal, and store the result in SP, thus incrementing SP by 1 (see the eighth line in Fig. 4-2). How can a register be read and written on the same cycle without producing garbage? The solution is that reading and writing are actually performed at different times within the cycle. When a register is selected as the ALU's right input, its value is put onto the B bus early in the cycle and kept there continuously throughout the entire cycle. The ALU then does its work, producing a result that passes through the shifter onto the C bus. Near the end of the cycle, when the ALU and shifter outputs are known to be stable, a clock signal triggers the store of the contents of the C bus into one or more of the registers. One of these registers may well be the one that supplied the B bus with its input. The precise timing of the data path makes it possible to read and write the same register on one cycle, as described below.

Data Path Timing

The timing of these events is shown in Fig. 4-3. Here a short pulse is produced at the start of each clock cycle. It can be derived from the main clock, as shown in Fig. 3-20(c). On the falling edge of the pulse, the bits that will drive all the gates are set up. This takes a finite and known time, Δw. Then the register needed on the B bus is selected and driven onto the B bus. It takes Δx before the value is stable. Then the ALU and shifter, which as combinational circuits have been running continuously, finally have valid data to operate on. After another Δy, the ALU and shifter outputs are stable. After an additional Δz, the results have propagated along the C bus to the registers, where they can be loaded on the rising edge of the next pulse. The load should be edge triggered and fast, so that even if some of the input registers are changed, the effects will not be felt on the C bus until long after the registers have been loaded. Also on the rising edge of the pulse, the register driving the B bus stops doing so, in preparation for the next cycle. MPC, MIR, and the memory are mentioned in the figure; their roles will be discussed shortly.

It is important to realize that even though there are no storage elements in the data path, there is a finite propagation time through it. Changing the value on the

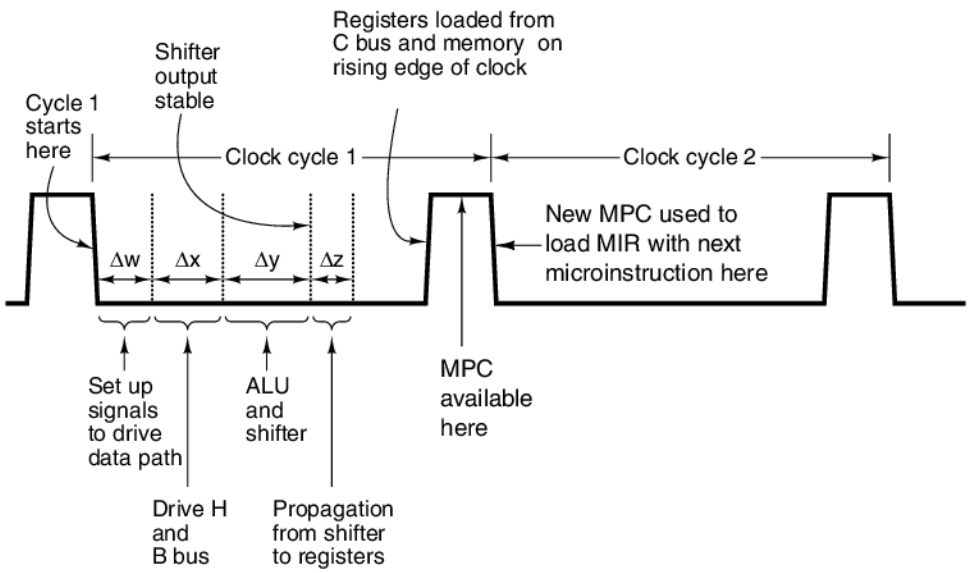


Figure 4-3. Timing diagram of one data path cycle.

B bus does not cause the C bus to change until a finite time later (due to the finite delays of each step). Consequently, even if a store changes one of the input registers, the value will be safely tucked away in the register long before the (now-incorrect) value being put on the B bus (or H) can reach the ALU.

Making this design work requires rigid timing, a long clock cycle, a known minimum propagation time through the ALU, and a fast load of the registers from the C bus. However, with careful engineering, the data path can be designed so that it functions correctly all the time. Actual machines work this way.

A somewhat different way to look at the data path cycle is to think of it as broken up into implicit subcycles. The start of subcycle 1 is triggered by the falling edge of the clock. The activities that go on during the subcycles are shown below along with the subcycle lengths (in parentheses).

1. The control signals are set up (Δw).
2. The registers are loaded onto the B bus (Δx).
3. The ALU and shifter operate (Δy).
4. The results propagate along the C bus back to the registers (Δz).

The time interval after Δz provides some tolerance since the times are not exact. At the rising edge of the next clock cycle, the results are stored in the registers.

We said that the subcycles can be best thought of as being *implicit*. By this we mean there are no clock pulses or other explicit signals to tell the ALU when to operate or tell the results to enter the C bus. In reality, the ALU and shifter run all the time. However, their inputs are garbage until a time $\Delta w + \Delta x$ after the falling edge of the clock. Likewise, their outputs are garbage until $\Delta w + \Delta x + \Delta y$ has elapsed after the falling edge of the clock. The only explicit signals that drive the data path are the falling edge of the clock, which starts the data path cycle, and the rising edge of the clock, which loads the registers from the C bus. The other subcycle boundaries are implicitly determined by the inherent propagation times of the circuits involved. It is the design engineers' responsibility to make sure that the time $\Delta w + \Delta x + \Delta y + \Delta z$ comes sufficiently in advance of the rising edge of the clock to have the register loads work reliably all the time.

Memory Operation

Our machine has two different ways to communicate with memory: a 32-bit, word-addressable memory port and an 8-bit, byte-addressable memory port. The 32-bit port is controlled by two registers, MAR (**Memory Address Register**) and MDR (**Memory Data Register**), as shown in Fig. 4-1. The 8-bit port is controlled by one register, PC, which reads 1 byte into the low-order 8 bits of MBR. This port can only read data from memory; it cannot write data to memory.

Each of these registers (and every other register in Fig. 4-1) is driven by one or two **control signals**. An open arrow under a register indicates a control signal that enables the register's output onto the B bus. Since MAR does not have a connection to the B bus, it does not have an enable signal. H does not have one either because, being the only possible left ALU input, it is always enabled.

A solid black arrow under a register indicates a control signal that writes (i.e., loads) the register from the C bus. Since MBR cannot be loaded from the C bus, it does not have a write signal (although it does have two other enable signals, described below). To initiate a memory read or write, the appropriate memory registers must be loaded, then a read or write signal issued to the memory (not shown in Fig. 4-1).

MAR contains *word* addresses, so that the values 0, 1, 2, etc. refer to consecutive words. PC contains *byte* addresses, so that the values 0, 1, 2, etc. refer to consecutive bytes. Thus putting a 2 in PC and starting a memory read will read out byte 2 from memory and put it in the low-order 8 bits of MBR. Putting a 2 in MAR and starting a memory read will read out bytes 8–11 (i.e., word 2) from memory and put them in MDR.

This difference in functionality is needed because MAR and PC will be used to reference two different parts of memory. The need for this distinction will become clearer later. For the moment, suffice it to say that the MAR/MDR combination is used to read and write ISA-level data words and the PC/MBR combination is used

to read the executable ISA-level program, which consists of a byte stream. All other registers that contain addresses use word addresses, like MAR.

In the actual physical implementation, there is only one real memory and it is byte oriented. Allowing MAR to count in words (needed due to the way JVM is defined) while the physical memory counts in bytes is handled by a simple trick. When MAR is placed on the address bus, its 32 bits do not map onto the 32 address lines, 0–31, directly. Instead MAR bit 0 is wired to address bus line 2, MAR bit 1 to address bus line 3, and so on. The upper 2 bits of MAR are discarded since they are needed only for word addresses above 2^{32}, none of which are legal for our 4-GB machine. Using this mapping, when MAR is 1, address 4 is put onto the bus; when MAR is 2, address 8 is put onto the bus, and so forth. This trick is illustrated in Fig. 4-4.

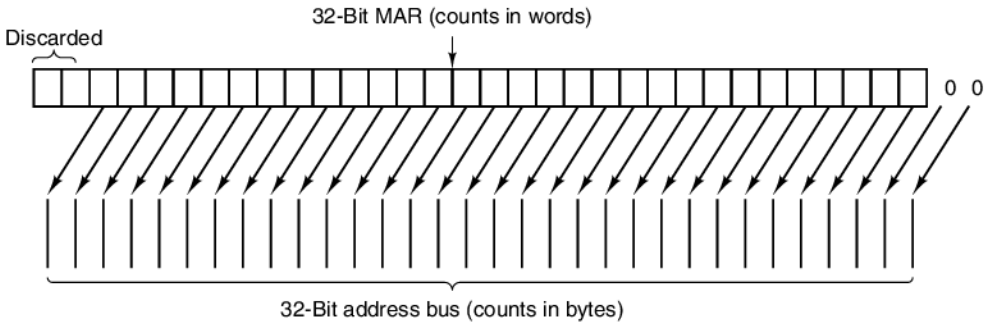


Figure 4-4. Mapping of the bits in MAR to the address bus.

As mentioned above, data read from memory through the 8-bit memory port are returned in MBR, an 8-bit register. MBR can be gated (i.e., copied) onto the B bus in one of two ways: unsigned or signed. When the unsigned value is needed, the 32-bit word put onto the B bus contains the MBR value in the low-order 8 bits and zeros in the upper 24 bits. Unsigned values are useful for indexing into a table, or when a 16-bit integer has to be assembled from 2 consecutive (unsigned) bytes in the instruction stream.

The other option for converting the 8-bit MBR to a 32-bit word is to treat it as a signed value between −128 and +127 and use this value to generate a 32-bit word with the same numerical value. This conversion is done by duplicating the MBR sign bit (leftmost bit) into the upper 24 bit positions of the B bus, a process known as **sign extension**. When this option is chosen, the upper 24 bits will either be all 0s or all 1s, depending on whether the leftmost bit of the 8-bit MBR is a 0 or a 1.

The choice of whether the 8-bit MBR is converted to an unsigned or a signed 32-bit value on the B bus is determined by which of the two control signals (open arrows below MBR in Fig. 4-1) is asserted. The need for these two options is why two arrows are present. The ability to have the 8-bit MBR act like a 32-bit source to the B bus is indicated by the dashed box to the left of MBR in the figure.

4.1.2 Microinstructions

To control the data path of Fig. 4-1, we need 29 signals. These can be divided into five functional groups, as described below.

9 Signals to control writing data from the C bus into registers.

9 Signals to control enabling registers onto the B bus for ALU input.

8 Signals to control the ALU and shifter functions.

2 Signals (not shown) to indicate memory read/write via MAR/MDR.

1 Signal (not shown) to indicate memory fetch via PC/MBR.

The values of these 29 control signals specify the operations for one cycle of the data path. A cycle consists of gating values out of registers and onto the B bus, propagating the signals through the ALU and shifter, driving them onto the C bus, and finally writing the results in the appropriate register or registers. In addition, if a memory read data signal is asserted, the memory operation is started at the end of the data path cycle, after MAR has been loaded. The memory data are available at the very end of the *following* cycle in MBR or MDR and can be used in the cycle *after that*. In other words, a memory read on either port initiated at the end of cycle k delivers data that cannot be used in cycle $k+1$, but only in cycle $k+2$ or later.

This seemingly counterintuitive behavior is explained by Fig. 4-3. The memory control signals are not generated in clock cycle 1 until just after MAR and PC are loaded at the rising edge of the clock, toward the end of clock cycle 1. We will assume the memory puts its results on the memory buses within one cycle so that MBR and/or MDR can be loaded on the next rising clock edge, along with the other registers.

Put in other words, we load MAR at the end of a data path cycle and start the memory shortly thereafter. Consequently, we cannot really expect the results of a read operation to be in MDR at the start of the next cycle, especially if the clock pulse is narrow. There is just not enough time if the memory takes one clock cycle. One data path cycle must intervene between starting a memory read and using the result. Of course, other operations can be performed during that cycle, just not those that need the memory word.

The assumption that the memory takes one cycle to operate is equivalent to assuming that the level 1 cache hit rate is 100%. This assumption is never true, but the complexity introduced by a variable-length memory cycle time is more than we want to deal with here.

Since MBR and MDR are loaded on the rising edge of the clock, along with all the other registers, they may be read during cycles when a new memory read is being performed. They return the old values, since the read has not yet had time to overwrite them. There is no ambiguity here; until new values are loaded into MBR

and MDR at the rising edge of the clock, the previous values are still there and usable. Note that it is possible to perform back-to-back reads on two consecutive cycles, since a read takes only 1 cycle. Also, both memories may operate at the same time. However, trying to read and write the same byte simultaneously gives undefined results.

While it may be desirable to write the output on the C bus into more than one register, it is never desirable to enable more than one register onto the B bus at a time. (In fact, some real implementations will suffer physical damage if this is done.) With a small increase in circuitry, we can reduce the number of bits needed to select among the possible sources for driving the B bus. There are only nine possible input registers that can drive the B bus (where the signed and unsigned versions of MBR each count separately). Therefore, we can encode the B bus information in 4 bits and use a decoder to generate the 16 control signals, 7 of which are not needed. In a commercial design, the architects would experience an overwhelming urge to get rid of one of the registers so that 3 bits would do the job. As academics, we have the enormous luxury of being able to waste 1 bit to give a cleaner and simpler design.

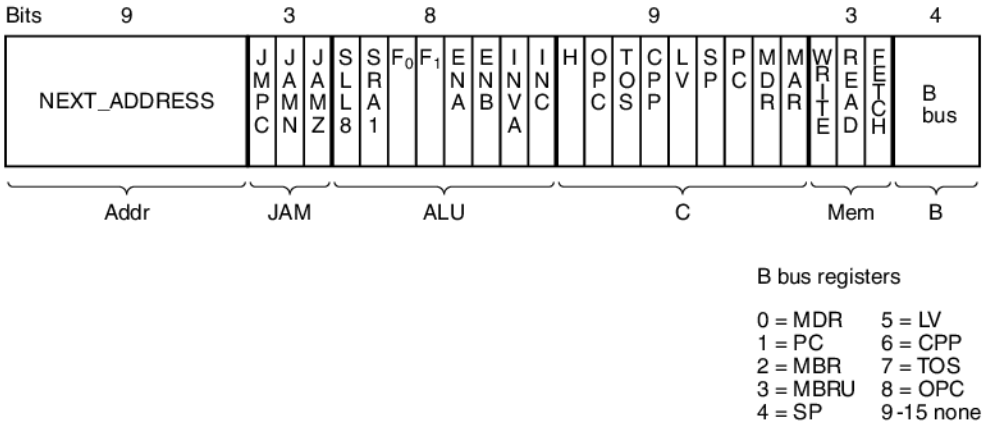


Figure 4-5. The microinstruction format for the Mic-1 (to be described shortly).

At this point we can control the data path with 9 + 4 + 8 + 2 + 1 = 24 signals, hence 24 bits. However, these 24 bits control the data path for only one cycle. The second part of the control is to determine what is done on the following cycle. To include this in the design of the controller, we will create a format for describing the operations to be performed using the 24 control bits plus two additional fields: NEXT_ADDRESS and JAM. Their contents will be discussed shortly. Figure 4-5 shows a possible format, divided into the six groups (listed below the instruction) and containing the following 36 signals:

Addr – Contains the address of a potential next microinstruction.

JAM – Determines how the next microinstruction is selected.

ALU – ALU and shifter functions.

C – Selects which registers are written from the C bus.

Mem – Memory functions.

B – Selects the B bus source; it is encoded as shown.

The ordering of the groups is, in principle, arbitrary, although we have actually chosen it very carefully to minimize line crossings in Fig. 4-6. Line crossings in schematic diagrams like Fig. 4-6 often correspond to wire crossings on chips, which cause trouble in two-dimensional designs and are best minimized.

4.1.3 Microinstruction Control: The Mic-1

So far we have described how the data path is controlled, but we have not yet described how it is decided which control signals should be enabled on each cycle. This is determined by a **sequencer** that is responsible for stepping through the sequence of operations for the execution of a single ISA instruction.

The sequencer must produce two kinds of information each cycle:

1. The state of every control signal in the system.
2. The address of the microinstruction that is to be executed next.

Figure 4-6 is a detailed block diagram of the complete microarchitecture of our example machine, which we will call the **Mic-1**. It may look intimidating initially but is worth studying carefully. When you fully understand every box and every line in this figure, you will be well on your way to understanding the microarchitecture level. The block diagram has two parts: the data path, on the left, which we have already discussed in detail, and the control section, on the right, which we will now look at.

The largest and most important item in the control portion of the machine is a memory called the **control store**. It is convenient to think of it as a memory that holds the complete microprogram, although it is sometimes implemented as a set of logic gates. In general, we will refer to it as the control store, to avoid confusion with the main memory, accessed through MBR and MDR. Functionally, however, the control store is a memory that simply holds microinstructions instead of ISA instructions. For our example machine, it contains 512 words, each consisting of one 36-bit microinstruction of the kind illustrated in Fig. 4-5. Actually, not all of these words are needed, but (for reasons to be explained shortly) we need addresses for 512 distinct words.

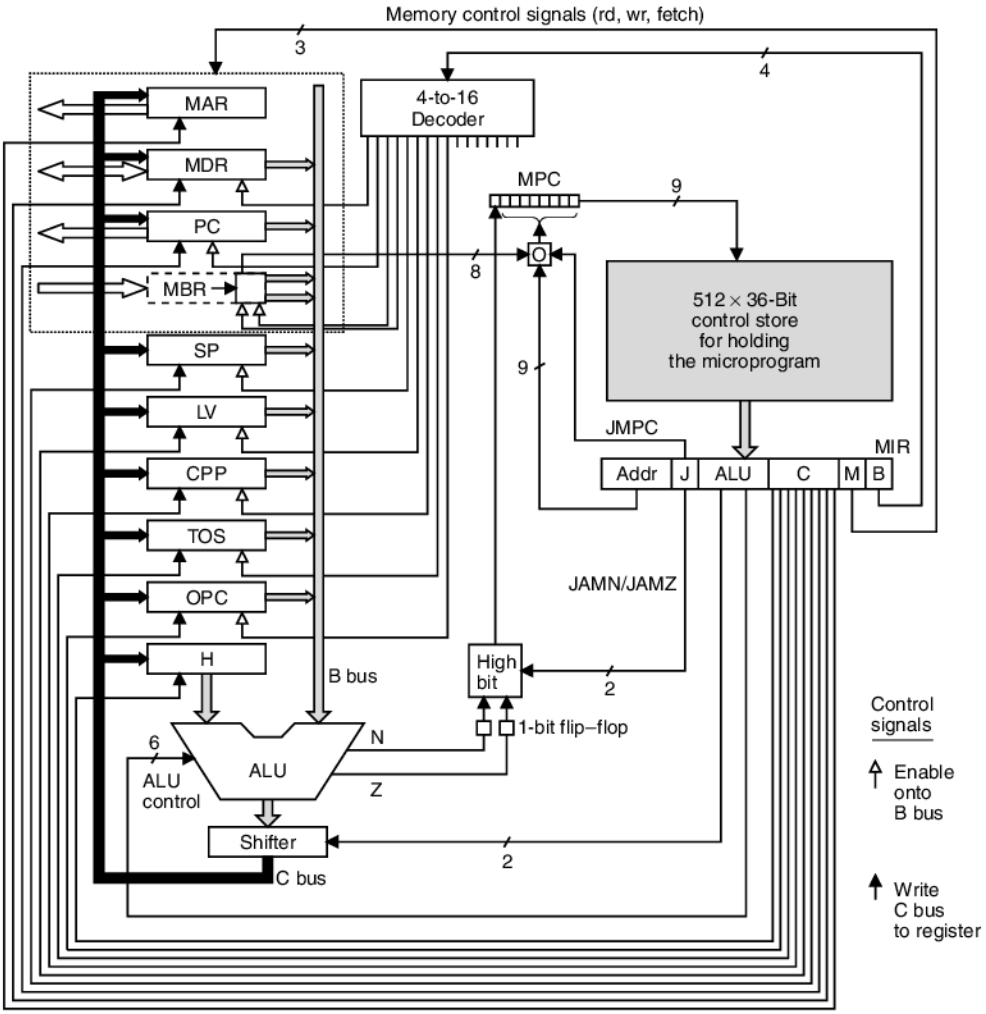


Figure 4-6. The complete block diagram of our example microarchitecture, the Mic-1.

In one important way, the control store is quite different from the main memory: instructions in main memory are always executed in address order (except for branches); microinstructions are not. The act of incrementing the program counter in Fig. 2-3 reflects the fact that the default instruction to execute after the current one is the instruction following the current one in memory. Microprograms need more flexibility (because microinstruction sequences tend to be short), so they usually do not have this property. Instead, each microinstruction explicitly specifies its successor.

Since the control store is functionally a (read-only) memory, it needs its own memory address register and memory data register. It does not need read and write

signals, because it is continuously being read. We will call the control store's memory address register **MPC** (**MicroProgram Counter**). This name is ironic since the locations in it are explicitly not ordered, so the concept of counting is not useful (but who are we to argue with tradition?). The memory data register is called **MIR** (**MicroInstruction Register**). Its function is to hold the current microinstruction, whose bits drive the control signals that operate the data path.

The MIR register in Fig. 4-6 holds the same six groups as in Fig. 4-5. The Addr and J (for JAM) groups control the selection of the next microinstruction and will be discussed shortly. The ALU group contains the 8 bits that select the ALU function and drive the shifter. The C bits cause individual registers to load the ALU output from the C bus. The M bits control memory operations.

Finally, the last 4 bits drive the decoder that determines what goes onto the B bus. In this case we have chosen to use a standard 4-to-16 decoder, even though only nine possibilities are required. In a more finely tuned design, a 4-to-9 decoder could be used. The trade-off here is using a standard circuit taken from a library of circuits versus designing a custom one. Using the standard circuit is simpler and is unlikely to introduce any bugs. Rolling your own uses less chip area but takes longer to design and you might get it wrong.

The operation of Fig. 4-6 is as follows. At the start of each clock cycle (the falling edge of the clock in Fig. 4-3), MIR is loaded from the word in the control store pointed to by MPC. The MIR load time is indicated in the figure by Δw. If one thinks in terms of subcycles, MIR is loaded during the first one.

Once the microinstruction is set up in MIR, the various signals propagate out into the data path. A register is put out onto the B bus, the ALU knows which operation to perform, and there is lots of activity out there. This is the second subcycle. After an interval $\Delta w + \Delta x$ from the start of the cycle, the ALU inputs are stable.

Another Δy later, everything has settled down and the ALU, N, Z, and shifter outputs are stable. The N and Z values are then saved in a pair of 1-bit flip-flops. These bits, like all the registers that are loaded from the C bus and from memory, are saved on the rising edge of the clock, near the end of the data path cycle. The ALU output is not latched but just fed into the shifter. The ALU and shifter activity occurs during subcycle 3.

After an additional interval, Δz, the shifter output has reached the registers via the C bus. Then the registers can be loaded near the end of the cycle (at the rising edge of the clock pulse in Fig. 4-3). Subcycle 4 consists of loading the registers and N and Z flip-flops. It terminates a little after the rising edge of the clock, when all the results have been saved and the results of the previous memory operations are available and MPC has been loaded. This process goes on and on until somebody gets bored with it and turns the machine off.

In parallel with driving the data path, the microprogram also has to determine which microinstruction to execute next, as they need not be executed in the order they happen to appear in the control store. The calculation of the address of the

next microinstruction begins after MIR has been loaded and is stable. First, the NEXT_ADDRESS field is copied to MPC. While this copy is taking place, the JAM field is inspected. If it has the value 000, nothing else is done; when the copy of NEXT_ADDRESS completes, MPC will point to the next microinstruction.

If one or more of the JAM bits are 1, more work is needed. If JAMN is set, the 1-bit N flip-flop is ORed into the high-order bit of MPC. Similarly, if JAMZ is set, the 1-bit Z flip-flop is ORed there. If both are set, both are ORed there. The reason that the N and Z flip-flops are needed is that after the rising edge of the clock (while the clock is high), the B bus is no longer being driven, so the ALU outputs can no longer be assumed to be correct. Saving the ALU status flags in N and Z makes the correct values available and stable for the MPC computation, no matter what is going on around the ALU.

In Fig. 4-6, the logic that does this computation is labeled "High bit." The Boolean function it computes is

$$F = (\text{JAMZ AND Z}) \text{ OR } (\text{JAMN AND N}) \text{ OR } \text{NEXT\_ADDRESS}[8]$$

Note that in all cases, MPC can take on only one of two possible values:

1. The value of NEXT_ADDRESS.
2. The value of NEXT_ADDRESS with the high-order bit ORed with 1.

No other possibilities make sense. If the high-order bit of NEXT_ADDRESS was already 1, using JAMN or JAMZ makes no sense.

Note that when the JAM bits are all zeros, the address of the next microinstruction to be executed is simply the 9-bit number in the NEXT_ADDRESS field. When either JAMN or JAMZ is 1, there are two potential successors: NEXT_ADDRESS and NEXT_ADDRESS ORed with 0x100 (assuming that NEXT_ADDRESS $\leq$ 0xFF). (Note that 0x indicates that the number following it is in hexadecimal.) This point is illustrated in Fig. 4-7. The current microinstruction, at location 0x75, has NEXT_ADDRESS = 0x92 and JAMZ set to 1. Consequently, the next address of the microinstruction depends on the Z bit stored on the previous ALU operation. If the Z bit is 0, the next microinstruction comes from 0x92. If the Z bit is 1, the next microinstruction comes from 0x192.

The third bit in the JAM field is JMPC. If it is set, the 8 MBR bits are bitwise ORed with the 8 low-order bits of the NEXT_ADDRESS field coming from the current microinstruction. The result is sent to MPC. The box with the label "O" in Fig. 4-6 does an OR of MBR with NEXT_ADDRESS if JMPC is 1 but just passes NEXT_ADDRESS through to MPC if JMPC is 0. When JMPC is 1, the low-order 8 bits of NEXT_ADDRESS are normally zero. The high-order bit can be 0 or 1, so the NEXT_ADDRESS value used with JMPC is normally 0x000 or 0x100. The reason for sometimes using 0x000 and sometimes using 0x100 will be discussed later.

The ability to OR MBR together with NEXT_ADDRESS and store the result in MPC allows an efficient implementation of a multiway branch (jump). Notice that

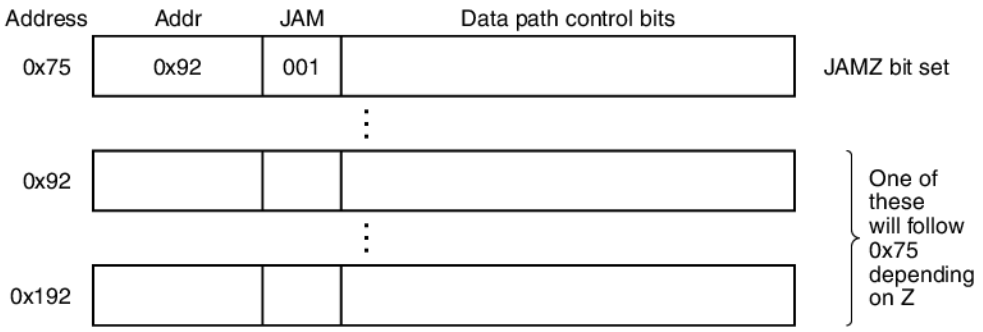


Figure 4-7. A microinstruction with JAMZ set to 1 has two potential successors.

any of 256 addresses can be specified, determined solely by the bits present in MBR. In a typical use, MBR contains an opcode, so the use of JMPC will result in a unique selection for the next microinstruction to be executed for every possible opcode. This method is useful for quickly branching directly to the function corresponding to the just-fetched opcode.

Understanding the timing of the machine is critical to what will follow, so it is perhaps worth repeating. We will do it in terms of subcycles, since this is easy to visualize, but the only real clock events are the falling edge, which starts the cycle, and the rising edge, which loads the registers and the N and Z flip-flops. Please refer to Fig. 4-3 once more.

During subcycle 1, initiated by the falling edge of the clock, MIR is loaded from the address currently held in MPC. During subcycle 2, the signals from MIR propagate out and the B bus is loaded from the selected register. During subcycle 3, the ALU and shifter operate and produce a stable result. During subcycle 4, the C bus, memory buses, and ALU values become stable. At the rising edge of the clock, the registers are loaded from the C bus, N and Z flip-flops are loaded, and MBR and MDR get their results from the memory operation started at the end of the previous data path cycle (if any). As soon as MBR is available, MPC is loaded in preparation for the next microinstruction. Thus MPC gets its value sometime during the middle of the interval when the clock is high but after MBR/MDR are ready. It could be either level triggered (rather than edge triggered), or edge trigger a fixed delay after the rising edge of the clock. All that matters is that MPC is not loaded until the registers it depends on (MBR, N, and Z) are ready. As soon as the clock falls, MPC can address the control store and a new cycle can begin.

Note that each cycle is self contained. It specifies what goes onto the B bus, what the ALU and shifter are to do, where the C bus is to be stored, and finally, what the next MPC value should be.

One final note about Fig. 4-6 is worth making. We have been treating MPC as a proper register, with 9 bits of storage capacity that is loaded while the clock is high. In reality, there is no need to have a register there at all. All of its inputs can be fed directly through, right to the control store. As long as they are present at the

control store at the falling edge of the clock when MIR is selected and read out, that is sufficient. There is no need to actually store them in MPC. For this reason, MPC might well be implemented as a **virtual register**, which is just a gathering place for signals, more like an electronic patch panel, than a real register. Making MPC a virtual register simplifies the timing: now events happen only on the falling and rising edges of the clock and nowhere else. But if it is easier for you to think of MPC as a real register, that is also a valid viewpoint.

4.2 AN EXAMPLE ISA: IJVM

Let us continue our example by introducing the ISA level of the machine to be interpreted by the microprogram running on the microarchitecture of Fig. 4-6 (IJVM). For convenience, we will sometimes refer to the Instruction Set Architecture as the **macroarchitecture**, to contrast it with the microarchitecture. Before we describe IJVM, however, we will digress slightly to motivate it.

4.2.1 Stacks

Virtually all programming languages support the concept of procedures (methods), which have local variables. These variables can be accessed from inside the procedure but cease to be accessible once the procedure has returned. The question thus arises: "Where should these variables be kept in memory?"

The simplest solution, to give each variable an absolute memory address, does not work. The problem is that a procedure may call itself. We will study these recursive procedures in Chap. 5. For the moment, suffice it to say that if a procedure is active (i.e., called) twice, its variables cannot be stored in absolute memory locations because the second invocation will interfere with the first.

Instead, a different strategy is used. An area of memory, called the **stack**, is reserved for variables, but individual variables do not get absolute addresses in it. Instead, a register, say, LV, is set to point to the base of the local variables for the current procedure. In Fig. 4-8(a), a procedure *A*, which has local variables *a1*, *a2*, and *a3*, has been called, so storage for its local variables has been reserved starting at the memory location pointed to by LV. Another register, SP, points to the highest word of *A*'s local variables. If LV is 100 and words are 4 bytes, then SP will be 108. Variables are referred to by giving their offset (distance) from LV. The data structure between LV and SP (and including both words pointed to) is called *A*'s **local variable frame**.

Now let us consider what happens if *A* calls another procedure, *B*. Where should *B*'s four local variables (*b1, b2, b3, b4*) be stored? *Answer*: On the stack, on top of *A*'s, as shown in Fig. 4-8(b). Notice that LV has been adjusted by the procedure call to point to *B*'s local variables instead of *A*'s. We can refer to *B*'s local variables by giving their offset from LV. Similarly, if *B* calls *C*, LV and SP are adjusted again to allocate space for *C*'s two variables, as shown in Fig. 4-8(c).

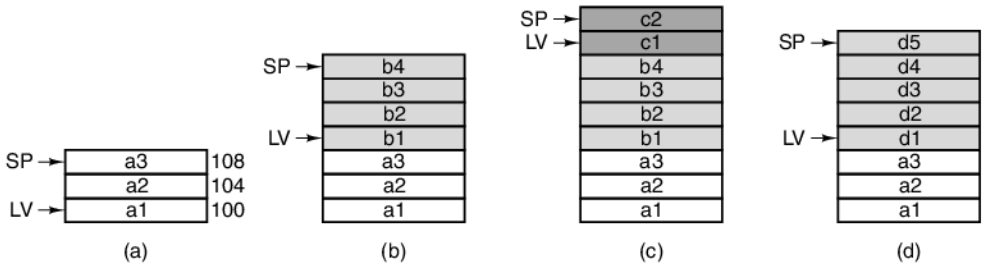


Figure 4-8. Use of a stack for storing local variables. (a) While *A* is active. (b) After *A* calls *B*. (c) After *B* calls *C*. (d) After *C* and *B* return and *A* calls *D*.

When *C* returns, *B* becomes active again, and the stack is adjusted back to Fig. 4-8(b) so that LV now points to *B*'s local variables again. Likewise, when *B* returns, we get back to the situation of Fig. 4-8(a). Under all conditions, LV points to the base of the stack frame for the currently active procedure, and SP points to the top of the stack frame.

Now suppose that *A* calls *D*, which has five local variables. We get the situation of Fig. 4-8(d), in which *D*'s local variables use the same memory that *B*'s did, as well as part of *C*'s. With this memory organization, memory is allocated only for procedures that are currently active. When a procedure returns, the memory used by its local variables is released.

Besides holding local variables, stacks have another use. They can hold operands during the computation of an arithmetic expression. When used this way, the stack is referred to as the **operand stack**. Suppose, for example, that before calling *B*, *A* has to do the computation

```
a1 = a2 + a3;
```

One way of doing this sum is to push *a2* onto the stack, as shown in Fig. 4-9(a). Here SP has been incremented by the number of bytes in a word, say, 4, and the first operand stored at the address now pointed to by SP. Next, *a3* is pushed onto the stack, as shown in Fig. 4-9(b). (As an aside on notation, we will typeset all program fragments in Helvetica, as above. We will also use this font for assembly-language opcodes and machine registers, but in running text, program variables and procedures will be given in *italics*. The difference is that variables and procedure names are chosen by the user; opcodes and register names are built in.)

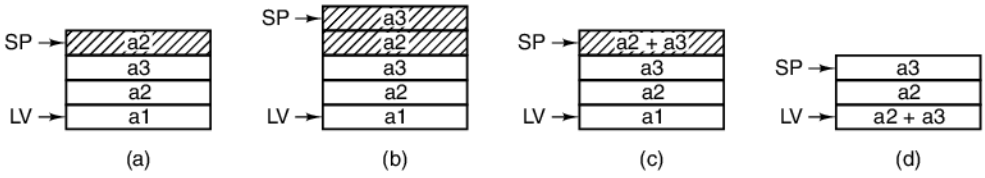


Figure 4-9. Use of an operand stack for doing an arithmetic computation.

The actual computation can now be done by executing an instruction that pops two words off the stack, adds them together, and pushes the result back onto the stack, as shown in Fig. 4-9(c). Finally, the top word can be popped off the stack and stored back in local variable *a1*, as illustrated in Fig. 4-9(d).

The local variable frames and the operand stacks can be intermixed. For example, when computing an expression like $x^2 + f(x)$, part of the expression (e.g., x^2) may be on the operand stack when a function f is called. The result of the function is left on the stack, on top of x^2, so the next instruction can add them.

It is worth noting that while all machines use a stack for storing local variables, not all use an operand stack like this for doing arithmetic. In fact, most of them do not, but JVM and IJVM work like this, which is why we have introduced stack operations here. We will study them in more detail in Chap. 5.

4.2.2 The IJVM Memory Model

We are now ready to look at the IJVM's architecture. Basically, it consists of a memory that can be viewed in either of two ways: an array of 4,294,967,296 bytes (4 GB) or an array of 1,073,741,824 words, each consisting of 4 bytes. Unlike most ISAs, the Java Virtual Machine makes no absolute memory addresses directly visible at the ISA level, but there are several implicit addresses that provide the base for a pointer. IJVM instructions can access memory only by indexing from these pointers. At any time, the following areas of memory are defined:

1. *The constant pool.* This area cannot be written by an IJVM program and consists of constants, strings, and pointers to other areas of memory that can be referenced. It is loaded when the program is brought into memory and not changed afterward. There is an implicit register, CPP, that contains the address of the first word of the constant pool.

2. *The Local variable frame.* For each invocation of a method, an area is allocated for storing variables during the lifetime of the invocation. It is called the **local variable frame**. At the beginning of this frame reside the parameters (also called arguments) with which the method was invoked. The local variable frame does not include the operand stack, which is separate. However, for efficiency reasons, our implementation chooses to implement the operand stack immediately above the local variable frame. An implicit register contains the address of the first location in the local variable frame. We will call this register LV. The parameters passed at the invocation of the method are stored at the beginning of the local variable frame.

3. *The operand stack.* The stack frame is guaranteed not to exceed a certain size, computed in advance by the Java compiler. The operand stack space is allocated directly above the local variable frame, as

illustrated in Fig. 4-10. In our implementation, it is convenient to think of the operand stack as part of the local variable frame. In any case, an implicit register contains the address of the top word of the stack. Notice that, unlike CPP and LV, this pointer, SP, changes during the execution of the method as operands are pushed onto the stack or popped from it.

4. *The method area.* Finally, there is a region of memory containing the program, referred to as the "text" area in a UNIX process. An implicit register contains the address of the instruction to be fetched next. This pointer is referred to as the Program Counter, or PC. Unlike the other regions of memory, the method area is treated as a byte array.

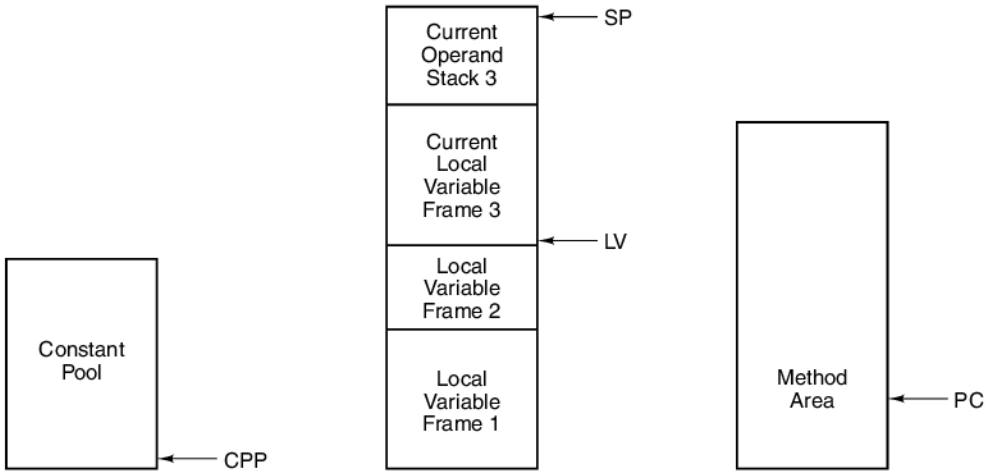


Figure 4-10. The various parts of the IJVM memory.

One point needs to be made regarding the pointers. The CPP, LV, and SP registers are all pointers to *words*, not *bytes*, and are offset by the number of words. For the integer subset we have chosen, all references to items in the constant pool, the local variables frame, and the stack are words, and all offsets used to index into these frames are word offsets. For example, LV, LV + 1, and LV + 2 refer to the first three words of the local variables frame. In contrast, LV, LV + 4, and LV + 8 refer to words at intervals of four words (16 bytes).

In contrast, PC contains a byte address, and an addition or subtraction to PC changes the address by a number of bytes, not a number of words. Addressing for PC is different from the others, and this fact is apparent in the special memory port provided for PC on the Mic-1. Remember that it is only 1 byte wide. Incrementing PC by one and initiating a read results in a fetch of the next *byte*. Incrementing SP by one and initiating a read results in a fetch of the next *word*.

4.2.3 The IJVM Instruction Set

The IJVM instruction set is shown in Fig. 4-11. Each instruction consists of an opcode and sometimes an operand, such as a memory offset or a constant. The first column gives the hexadecimal encoding of the instruction. The second gives its assembly-language mnemonic. The third gives a brief description of its effect.

Hex	Mnemonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to a local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Figure 4-11. The IJVM instruction set. The operands *byte*, *const*, and *varnum* are 1 byte. The operands *disp*, *index*, and *offset* are 2 bytes.

Instructions are provided to push a word from various sources onto the stack. These sources include the constant pool (LDC_W), the local variable frame (ILOAD), and the instruction itself (BIPUSH). A variable can also be popped from the stack and stored into the local variable frame (ISTORE). Two arithmetic operations (IADD and ISUB) as well as two logical (Boolean) operations (IAND and IOR) can be performed using the two top words on the stack as operands. In all the arithmetic and logical operations, two words are popped from the stack and the result pushed back onto it. Four branch instructions are provided, one unconditional (GOTO) and three conditional ones (IFEQ, IFLT, and IF_ICMPEQ). All the branch instructions, if taken, adjust the value of PC by the size of their (16-bit signed) offset, which follows the opcode in the instruction. This offset is added to the address of the opcode. There

are also IJVM instructions for swapping the top two words on the stack (SWAP), duplicating the top word (DUP), and removing it (POP).

Some instructions have multiple formats, allowing a short form for commonly used versions. In IJVM we have included two of the various mechanisms JVM uses to accomplish this. In one case we have skipped the short form in favor of the more general one. In another case we show how the prefix instruction WIDE can be used to modify the ensuing instruction.

Finally, there is an instruction (INVOKEVIRTUAL) for invoking another method, and another instruction (IRETURN) for exiting the method and returning control to the method that invoked it. Due to the complexity of the mechanism we have slightly simplified the definition, making it possible to produce a straightforward mechanism for invoking a call and return. The restriction is that, unlike Java, we allow a method to invoke only a method existing within its own object. This restriction severely cripples the object orientation but allows us to present a much simpler mechanism, by avoiding the requirement to locate the method dynamically. (If you are not familiar with object-oriented programming, you can safely ignore this remark. What we have done is turn Java back into a nonobject-oriented language, such as C or Pascal.) On all computers except JVM, the address of the procedure to call is determined directly by the CALL instruction, so our approach is actually the normal case, not the exception.

The mechanism for invoking a method is as follows. First, the callr pushes onto the stack a reference (pointer) to the object to be called. (This reference is not needed in IJVM since no other object may be specified, but it is retained for consistency with JVM.) In Fig. 4-12(a) this reference is indicated by OBJREF. Then the caller pushes the method's parameters onto the stack, in this example, *Parameter 1*, *Parameter 2*, and *Parameter 3*. Finally, INVOKEVIRTUAL is executed.

The INVOKEVIRTUAL instruction includes a displacement which indicates the position in the constant pool that contains the start address within the nethod area for the method being invoked. However, while the method code resides at the location pointed to by this pointer, the first 4 bytes in the method area contain special data. The first 2 bytes are interpreted as a 16-bit integer indicating the number of parameters for the method (the parameters themselves have previously been pushed onto the stack). For this count, OBJREF is counted as a parameter: parameter 0. This 16-bit integer, together with the value of SP, provides the location of OBJREF. Note that LV points to OBJREF rather than the first real parameter. The choice of where LV points is somewhat arbitrary.

The second 2 bytes in the method area are interpreted as another 16-bit integer indicating the size of the local variable area for the method being invoked. This is necessary because a new stack will be established for the method, beginning immediately above the local variable frame. Finally, the fifth byte in the method area contains the first opcode to be executed.

The actual sequence that occurs for INVOKEVIRTUAL is as follows and is depicted in Fig. 4-12. The two unsigned index bytes that follow the opcode are

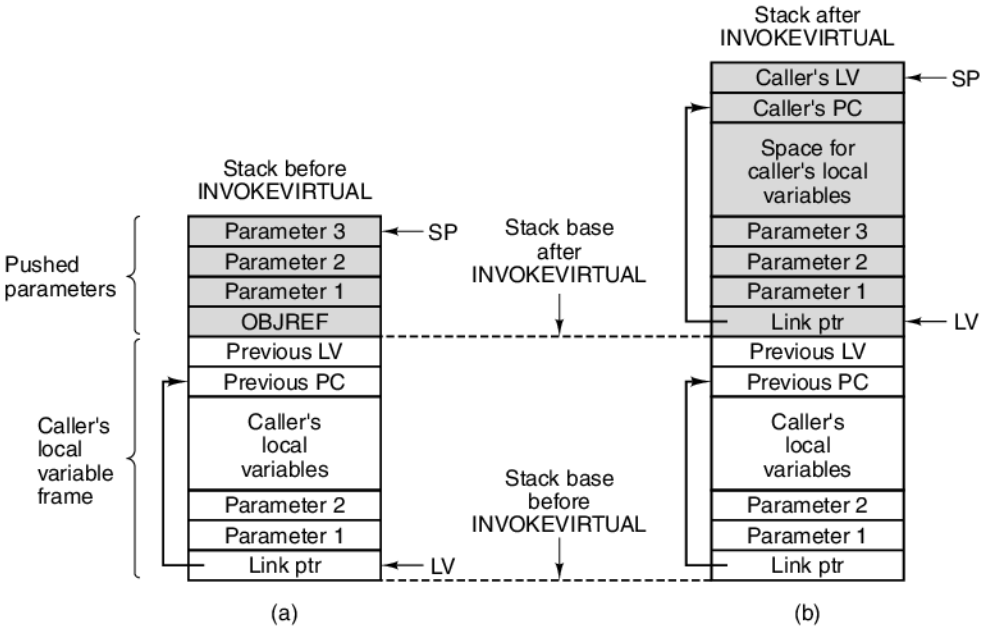


Figure 4-12. (a) Memory before executing INVOKEVIRTUAL. (b) After executing it.

used to construct an index into the constant pool table (the first byte is the high-order byte). The instruction computes the base address of the new local variable frame by subtracting off the number of parameters from the stack pointer and setting LV to point to OBJREF. At this location, overwriting OBJREF, the implementation stores the address of the location where the old PC is to be stored. This address is computed by adding the size of the local variable frame (parameters + local variables) to the address contained in LV. Immediately above the address where the old PC is to be stored is the address where the old LV is to be stored. Immediately above that address is the beginning of the stack for the newly called procedure. SP is set to point to the old LV, which is the address immediately below the first empty location on the stack. Remember that SP always points to the top word on the stack. If the stack is empty, it points to the first location below the end of the stack because our stacks grow upward, toward higher addresses. In our figures, stacks always grow upward, toward the higher address at the top of the page.

The last operation needed to carry out INVOKEVIRTUAL is to set PC to point to the fifth byte in the method code space.

The IRETURN instruction reverses the operations of the INVOKEVIRTUAL instruction, as shown in Fig. 4-13. It deallocates the space that was used by the returning method. It also restores the stack to its former state, except that (1) the (now overwritten) OBJREF word and all the parameters have been popped from the stack, and (2) the returned value has been placed at the top of the stack, at the

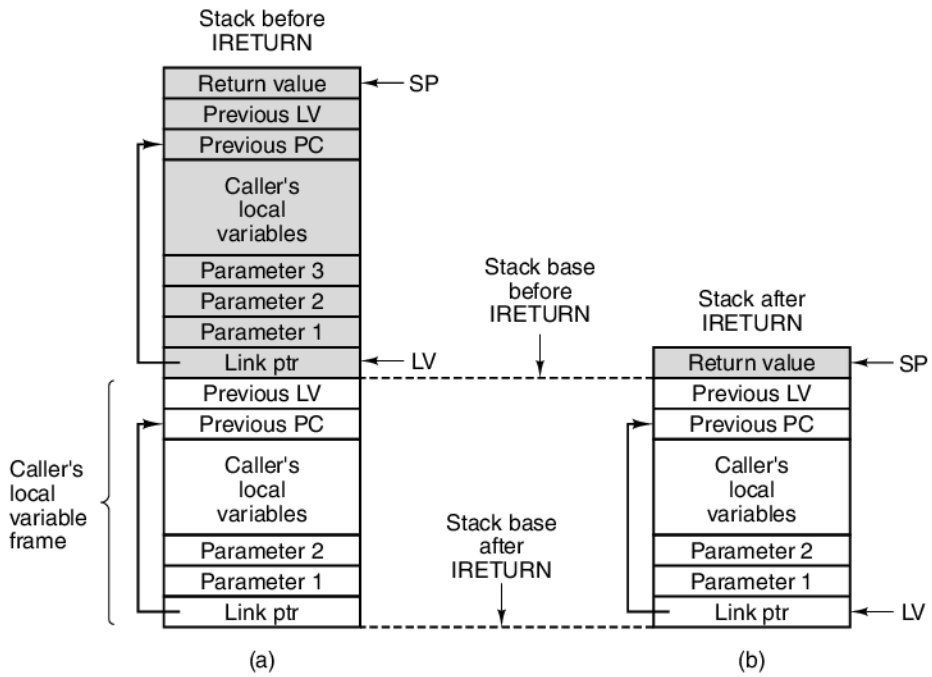


Figure 4-13. (a) Memory before executing IRETURN. (b) After executing it.

location formerly occupied by OBJREF. To restore the old state, the IRETURN instruction must be able to restore the PC and LV pointers to their old values. It does this by accessing the link pointer (which is the word identified by the current LV pointer). In this location, remember, where the OBJREF was originally stored, the INVOKEVIRTUAL instruction stored the address containing the old PC. This word and the word above it are retrieved to restore PC and LV, respectively, to their old values. The return value, which is stored at the top of the stack of the terminating method, is copied to the location where the OBJREF was originally stored, and SP is restored to point to this location. Control is therefore returned to the instruction immediately following the INVOKEVIRTUAL instruction.

So far, our machine does not have any input/output instructions. Nor are we going to add any. It does not need them any more than the Java Virtual Machine needs them, and the official specification for JVM never even mentions I/O. The theory is that a machine that does no input or output is "safe." (Reading and writing are performed in JVM by means of calls to special I/O methods.)

4.2.4 Compiling Java to IJVM

Let us now see how Java and IJVM relate to one another. In Fig. 4-14(a) we show a simple fragment of Java code. When fed to a Java compiler, the compiler would probably produce the IJVM assembly-language shown in Fig. 4-14(b). The

line numbers from 1 to 15 at the left of the assembly language program are not part of the compiler output. Nor are the comments (starting with //). They are there to help explain a subsequent figure. The Java assembler would then translate the assembly program into the binary program shown in Fig. 4-14(c). (Actually, the Java compiler does its own assembly and produces the binary program directly.) For this example, we have assumed that *i* is local variable 1, *j* is local variable 2, and *k* is local variable 3.

```
i = j + k;                1     ILOAD j          // i = j + k      0x15 0x02
if (i == 3)               2     ILOAD k                         0x15 0x03
    k = 0;                3     IADD                            0x60
else                      4     ISTORE i                        0x36 0x01
   j = j – 1;             5     ILOAD i          // if (i == 3)    0x15 0x01
                          6     BIPUSH 3                        0x10 0x03
                          7     IF_ICMPEQ L1                    0x9F 0x00 0x0D
                          8     ILOAD j          // j = j – 1      0x15 0x02
                          9     BIPUSH 1                        0x10 0x01
                         10     ISUB                            0x64
                         11     ISTORE j                        0x36 0x02
                         12     GOTO L2                         0xA7 0x00 0x07
                         13  L1: BIPUSH 0         // k = 0        0x10 0x00
                         14     ISTORE k                        0x36 0x03
                         15  L2:
   (a)                             (b)                              (c)
```

Figure 4-14. (a) A Java fragment. (b) The corresponding Java assembly language. (c) The IJVM program in hexadecimal.

The compiled code is straightforward. First *j* and *k* are pushed onto the stack, added, and the result stored in *i*. Then *i* and the constant 3 are pushed onto the stack and compared. If they are equal, a branch is taken to *L1*, where *k* is set to 0. If they are unequal, the compare fails and code following IF_ICMPEQ is executed. When it is done, it branches to *L2*, where the then and else parts merge.

The operand stack for the IJVM program of Fig. 4-14(b) is shown in Fig. 4-15. Before the code starts executing, the stack is empty, indicated by the horizontal line above the 0. After the first ILOAD, *j* is on the stack, as indicated by the boxed *j* above the 1 (meaning instruction 1 has executed). After the second ILOAD, two words are on the stack, as shown above the 2. After the IADD, only one word is on the stack, and it contains the sum $j + k$. When the top word is popped from the stack and stored in *i*, the stack is empty, as shown above the 4.

Instruction 5 (ILOAD) starts the if statement by pushing *i* onto the stack (in 5) Next comes the constant 3 (in 6). After the comparison, the stack is empty again (7). Instruction 8 is the start of the else part of the Java program fragment. The else part continues until instruction 12, at which time it branches over the then part and goes to label *L2*.

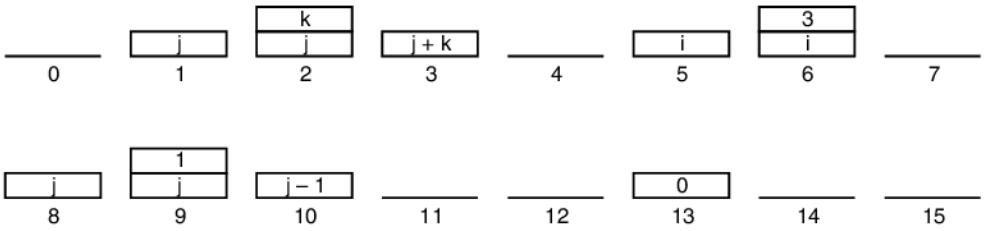


Figure 4-15. The stack after each instruction of Fig. 4-14(b).

4.3 AN EXAMPLE IMPLEMENTATION

Having specified both the microarchitecture and the macroarchitecture in detail, the remaining issue is the implementation. In other words, what does a program running on the former and interpreting the latter look like, and how does it work? Before we can answer these questions, we must carefully consider the notation we will use to describe the implementation.

4.3.1 Microinstructions and Notation

In principle, we could describe the control store in binary, 36 bits per word. But as in conventional programming languages, there is great benefit in introducing notation that conveys the essence of the issues we need to deal with while obscuring the details that can be ignored or better handled automatically. It is important to realize here that the language we have chosen is intended to illustrate the concepts rather than to facilitate efficient designs. If the latter were our goal, we would use a different notation to maximize the flexibility available to the designer. One aspect where this issue is important is the choice of addresses. Since the memory is not logically ordered, there is no natural "next instruction" to be implied as we specify a sequence of operations. Much of the power of this control organization derives from the ability of the designer (or the assembler) to select addresses efficiently. We therefore begin by introducing a simple symbolic language that fully describes each operation without explaining fully how all addresses may have been determined.

Our notation specifies all the activities that occur in a single clock cycle in a single line. We could, in theory, use a high-level language to describe the operations. However, cycle-by-cycle control is very important because it gives the opportunity to perform multiple operations concurrently, and we need to be able to analyze each cycle to understand and verify the operations. If the goal is a fast, efficient implementation (other things being equal, fast and efficient is always better than slow and inefficient), then every cycle counts. In a real implementation, many subtle tricks are hidden in the program, using obscure sequences or operations in order to save a single cycle. There is a high payoff for saving cycles: a four-cycle

instruction that can be reduced by two cycles now runs twice as fast. And this speedup is obtained every time we execute the instruction.

One possible approach is simply to list the signals that should be activated each clock cycle. Suppose that in one cycle we want to increment the value of SP. We also want to initiate a read operation, and we want the next instruction to be the one residing at location 122 in the control store. We might write

```
ReadRegister = SP, ALU = INC, WSP, Read, NEXT_ADDRESS = 122
```

where WSP means "write the SP register." This notation is complete but hard to understand. Instead we will combine the operations in a natural and intuitive way to capture the effect of what is happening:

```
SP = SP + 1; rd
```

Let us call our high-level Micro Assembly Language "MAL" (French for "sick," something you become if you have to write too much code in it). MAL is tailored to reflect the characteristics of the microarchitecture. During each cycle, any of the registers can be written, but typically only one is. Only one register can be gated to the B side of the ALU. On the A side, the choices are +1, 0, –1, and the register H. Thus we can use a simple assignment statement, as in Java, to indicate the operation to be performed. For example, to copy something from SP to MDR, we can say

```
MDR = SP
```

To indicate the use of the ALU functions other than passing through the B bus, we can write, for example,

```
MDR = H + SP
```

which adds the contents of the H register to SP and writes the result into MDR. The + operator is commutative (which means that the order of the operands does not matter), so the above statement can also be written as

```
MDR = SP + H
```

and generate the same 36-bit microinstruction, even though strictly speaking H must be the left ALU operand.

We have to be careful to use only permitted operations. The most important of them are shown in Fig. 4-16, where SOURCE can be any of MDR, PC, MBR, MBRU, SP, LV, CPP, TOS, or OPC (MBRU implies the unsigned version of MBR). These registers can all act as sources to the ALU on the B bus. Similarly, DEST can be any of MAR, MDR, PC, SP, LV, CPP, TOS, OPC, or H, all of which are possible destinations for the ALU output on the C bus. This format is deceptive because many seemingly reasonable statements are illegal. For example,

```
MDR = SP + MDR
```

looks perfectly reasonable, but there is no way to execute it on the data path of Fig. 4-6 in one cycle. This restriction exists because for an addition (other than increment or decrement) one of the operands must be the H register. Likewise,

H = H – MDR

might be useful, but it, too, is impossible, because the only possible source of a subtrahend (the value being subtracted) is the H register. It is up to the assembler to reject statements that look valid but are, in fact, illegal.

DEST = H
DEST = SOURCE
DEST = $\overline{\text{H}}$
DEST = $\overline{\text{SOURCE}}$
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE – H
DEST = SOURCE – 1
DEST = –H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = –1

Figure 4-16. All permitted operations. Any of the above operations may be extended by adding "<< 8" to them to shift the result left by 1 byte. For example, a common operation is H = MBR << 8.

We extend the notation to permit multiple assignments by the use of multiple equal signs. For example, adding 1 to SP and storing it back into SP as well as writing it into MDR can be accomplished by

SP = MDR = SP + 1

To indicate memory reads and writes of 4-byte data words, we will just put rd and wr in the microinstruction. Fetching a byte through the 1-byte port is indicated by fetch. Assignments and memory operations can occur in the same cycle. This is indicated by writing them on the same line.

To avoid any confusion, let us repeat that the Mic-1 has two ways of accessing memory. Reads and writes of 4-byte data words use MAR/MDR and are indicated in the microinstructions by rd and wr, respectively. Reads of 1-byte opcodes from the

instruction stream use PC/MBR and are indicated by fetch in the microinstructions. Both kinds of memory operations can proceed simultaneously.

However, the same register may not receive a value from memory and the data path in the same cycle. Consider the code

```
MAR = SP; rd
MDR = H
```

The effect of the first microinstruction is to assign a value from memory to MDR at the end of the second microinstruction. However, the second microinstruction also assigns a value to MDR at the same time. These two assignments are in conflict and are not permitted, as the results are undefined.

Remember that each microinstruction must explicitly supply the address of the next microinstruction to be executed. However, it commonly occurs that a microinstruction is invoked only by one other microinstruction, namely, by the one on the line immediately above it. To ease the microprogrammer's job, the microassembler normally assigns an address to each microinstruction (not necessarily consecutive in the control store) and fills in the NEXT_ADDRESS field so that microinstructions written on consecutive lines are executed consecutively.

However, sometimes the microprogrammer wants to branch away, either unconditionally or conditionally. The notation for unconditional branches is easy:

```
goto label
```

can be included in any microinstruction to explicitly name its successor. For example, most microinstruction sequences end with a return to the first instruction of the main loop, so the last instruction in each such sequence typically includes

```
goto Main1
```

Note that the data path is available for normal operations even during a microinstruction that contains a goto. After all, every single microinstruction contains a NEXT_ADDRESS field. All goto does is instruct the microassembler to put a specific value there instead of the address where it has decided to place the microinstruction on the next line. In principle, every line should have a goto statement. As a convenience to the microprogrammer, when the target address is the next line, it may be omitted.

For conditional branches, we need a different notation. Remember that JAMN and JAMZ use the N and Z bits, which are set based on the ALU output. Sometimes we need to test a register to see if it is zero, for example. One way to do this would be to run it through the ALU and store it back in itself. Writing

```
TOS = TOS
```

looks peculiar, although it does the job (setting the Z flip-flop based on TOS). However, to make microprograms look nicer, we now extend MAL, adding two new imaginary registers, N and Z, which can be assigned to. For example,

```
Z = TOS
```

runs TOS through the ALU, thus setting the Z (and N) flip-flops, but it does not do a store into any register. What using Z or N as a destination really does is tell the microassembler to set all the bits in the C field of Fig. 4-5 to 0. The data path executes a normal cycle, with all normal operations allowed, but no registers are written to. Note that it does not matter whether the destination is N or Z; the microinstruction generated by the microassembler is identical. Programmers who intentionally choose the "wrong" one should be forced to work on a 4.77-MHz original IBM PC for a week as punishment.

The syntax for telling the microassembler to set the JAMZ bit is

```
if (Z) goto L1; else goto L2
```

Since the hardware requires these two addresses to be identical in their low-order 8 bits, it is up to the microassembler to assign them such addresses. On the other hand, since *L2* can be anywhere in the bottom 256 words of the control store, the microassembler has a lot of freedom in finding an available pair.

Normally, these two statements will be combined, for example,

```
Z = TOS; if (Z) goto L1; else goto L2
```

The effect of this statement is that MAL generates a microinstruction in which TOS is run through the ALU (but not stored anywhere) so that its value sets the Z bit. Shortly after Z has been loaded from the ALU condition bit, it is ORed into the high-order bit of MPC, forcing the address of the next microinstruction to be fetched from either *L2* or *L1* (which must be exactly 256 more than *L2*). MPC will be stable and ready to use for fetching the next microinstruction.

Finally, we need a notation for using the JMPC bit. The one we will use is

```
goto (MBR OR value)
```

This syntax tells the microassembler to use *value* for NEXT_ADDRESS and set the JMPC bit so that MBR is ORed into MPC along with NEXT_ADDRESS. If *value* is 0, which is the normal case, it is sufficient to just write

```
goto (MBR)
```

Note that only the low-order 8 bits of MBR are wired to MPC (see Fig. 4-6), so the issue of sign extension (i.e., MBR versus MBRU) does not arise here. Also note that the MBR available at the end of the current cycle is the one used. A fetch started in *this* microinstruction is too late to affect the choice of the next microinstruction.

4.3.2 Implementation of IJVM Using the Mic-1

We have finally reached the point where we can put all the pieces together. Figure 4-17 is the microprogram that runs on Mic-1 and interprets IJVM. It is a surprisingly short program—only 112 microinstructions total. Three columns are

Label	Operations	Comments
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch
nop1	goto Main1	Do nothing
iadd1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
iadd2	H = TOS	H = top of stack
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Add top two words; write to top of stack
isub1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
isub2	H = TOS	H = top of stack
isub3	MDR = TOS = MDR − H; wr; goto Main1	Do subtraction; write to top of stack
iand1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
iand2	H = TOS	H = top of stack
iand3	MDR = TOS = MDR AND H; wr; goto Main1	Do AND; write to new top of stack
ior1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
ior2	H = TOS	H = top of stack
ior3	MDR = TOS = MDR OR H; wr; goto Main1	Do OR; write to new top of stack
dup1	MAR = SP = SP + 1	Increment SP and copy to MAR
dup2	MDR = TOS; wr; goto Main1	Write new stack word
pop1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
pop2		Wait for new TOS to be read from memory
pop3	TOS = MDR; goto Main1	Copy new word to TOS
swap1	MAR = SP − 1; rd	Set MAR to SP − 1; read 2nd word from stack
swap2	MAR = SP	Set MAR to top word
swap3	H = MDR; wr	Save TOS in H; write 2nd word to top of stack
swap4	MDR = TOS	Copy old TOS to MDR
swap5	MAR = SP − 1; wr	Set MAR to SP − 1; write as 2nd word on stack
swap6	TOS = H; goto Main1	Update TOS
bipush1	SP = MAR = SP + 1	MBR = the byte to push onto stack
bipush2	PC = PC + 1; fetch	Increment PC, fetch next opcode
bipush3	MDR = TOS = MBR; wr; goto Main1	Sign-extend constant and push on stack
iload1	H = LV	MBR contains index; copy LV to H
iload2	MAR = MBRU + H; rd	MAR = address of local variable to push
iload3	MAR = SP = SP + 1	SP points to new top of stack; prepare write
iload4	PC = PC + 1; fetch; wr	Inc PC; get next opcode; write top of stack
iload5	TOS = MDR; goto Main1	Update TOS
istore1	H = LV	MBR contains index; copy LV to H
istore2	MAR = MBRU + H	MAR = address of local variable to store into
istore3	MDR = TOS; wr	Copy TOS to MDR; write word
istore4	SP = MAR = SP − 1; rd	Read in next-to-top word on stack
istore5	PC = PC + 1; fetch	Increment PC; fetch next opcode
istore6	TOS = MDR; goto Main1	Update TOS
wide1	PC = PC + 1; fetch;	Fetch operand byte or next opcode
wide2	goto (MBR OR 0x100)	Multiway branch with high bit set
wide_iload1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_iload2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_iload3	H = MBRU OR H	H = 16-bit index of local variable
wide_iload4	MAR = LV + H; rd; goto iload3	MAR = address of local variable to push
wide_istore1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_istore2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_istore3	H = MBRU OR H	H = 16-bit index of local variable
wide_istore4	MAR = LV + H; goto istore3	MAR = address of local variable to store into
ldc_w1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
ldc_w2	H = MBRU << 8	H = 1st index byte << 8
ldc_w3	H = MBRU OR H	H = 16-bit index into constant pool
ldc_w4	MAR = H + CPP; rd; goto iload3	MAR = address of constant in pool

Label	Operations	Comments
iinc1	H = LV	MBR contains index; copy LV to H
iinc2	MAR = MBRU + H; rd	Copy LV + index to MAR; read variable
iinc3	PC = PC + 1; fetch	Fetch constant
iinc4	H = MDR	Copy variable to H
iinc5	PC = PC + 1; fetch	Fetch next opcode
iinc6	MDR = MBR + H; wr; goto Main1	Put sum in MDR; update variable
goto1	OPC = PC − 1	Save address of opcode.
goto2	PC = PC + 1; fetch	MBR = 1st byte of offset; fetch 2nd byte
goto3	H = MBR << 8	Shift and save signed first byte in H
goto4	H = MBRU OR H	H = 16-bit branch offset
goto5	PC = OPC + H; fetch	Add offset to OPC
goto6	goto Main1	Wait for fetch of next opcode
iflt1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
iflt2	OPC = TOS	Save TOS in OPC temporarily
iflt3	TOS = MDR	Put new top of stack in TOS
iflt4	N = OPC; if (N) goto T; else goto F	Branch on N bit
ifeq1	MAR = SP = SP − 1; rd	Read in next-to-top word of stack
ifeq2	OPC = TOS	Save TOS in OPC temporarily
ifeq3	TOS = MDR	Put new top of stack in TOS
ifeq4	Z = OPC; if (Z) goto T; else goto F	Branch on Z bit
if_icmpeq1	MAR = SP = SP − 1; rd	Read in next-to-top word of stack
if_icmpeq2	MAR = SP = SP − 1	Set MAR to read in new top-of-stack
if_icmpeq3	H = MDR; rd	Copy second stack word to H
if_icmpeq4	OPC = TOS	Save TOS in OPC temporarily
if_icmpeq5	TOS = MDR	Put new top of stack in TOS
if_icmpeq6	Z = OPC − H; if (Z) goto T; else goto F	If top 2 words are equal, goto T, else goto F
T	OPC = PC − 1; goto goto2	Same as goto1; needed for target address
F	PC = PC + 1	Skip first offset byte
F2	PC = PC + 1; fetch	PC now points to next opcode
F3	goto Main1	Wait for fetch of opcode
invokevirtual1	PC = PC + 1; fetch	MBR = index byte 1; inc. PC, get 2nd byte
invokevirtual2	H = MBRU << 8	Shift and save first byte in H
invokevirtual3	H = MBRU OR H	H = offset of method pointer from CPP
invokevirtual4	MAR = CPP + H; rd	Get pointer to method from CPP area
invokevirtual5	OPC = PC + 1	Save return PC in OPC temporarily
invokevirtual6	PC = MDR; fetch	PC points to new method; get param count
invokevirtual7	PC = PC + 1; fetch	Fetch 2nd byte of parameter count
invokevirtual8	H = MBRU << 8	Shift and save first byte in H
invokevirtual9	H = MBRU OR H	H = number of parameters
invokevirtual10	PC = PC + 1; fetch	Fetch first byte of # locals
invokevirtual11	TOS = SP − H	TOS = address of OBJREF − 1
invokevirtual12	TOS = MAR = TOS + 1	TOS = address of OBJREF (new LV)
invokevirtual13	PC = PC + 1; fetch	Fetch second byte of # locals
invokevirtual14	H = MBRU << 8	Shift and save first byte in H
invokevirtual15	H = MBRU OR H	H = # locals
invokevirtual16	MDR = SP + H + 1; wr	Overwrite OBJREF with link pointer
invokevirtual17	MAR = SP = MDR;	Set SP, MAR to location to hold old PC
invokevirtual18	MDR = OPC; wr	Save old PC above the local variables
invokevirtual19	MAR = SP = SP + 1	SP points to location to hold old LV
invokevirtual20	MDR = LV; wr	Save old LV above saved PC
invokevirtual21	PC = PC + 1; fetch	Fetch first opcode of new method.
invokevirtual22	LV = TOS; goto Main1	Set LV to point to LV Frame

Figure 4-17. The microprogram for the Mic-1 (part 1 on facing page, part 2 above, part 3 on next page).

Label	Operations	Comments
ireturn1	MAR = SP = LV; rd	Reset SP, MAR to get link pointer
ireturn2		Wait for read
ireturn3	LV = MAR = MDR; rd	Set LV to link ptr; get old PC
ireturn4	MAR = LV + 1	Set MAR to read old LV
ireturn5	PC = MDR; rd; fetch	Restore PC; fetch next opcode
ireturn6	MAR = SP	Set MAR to write TOS
ireturn7	LV = MDR	Restore LV
ireturn8	MDR = TOS; wr; goto Main1	Save return value on original top of stack

Fig. 4-17. The microprogram for the Mic-1 (part 3 of 3).

given for each microinstruction: the symbolic label, the actual microcode, and a comment. Note that consecutive microinstructions are not necessarily located in consecutive addresses in the control store, as we have already pointed out.

By now the choice of names for most of the registers in Fig. 4-1 should be obvious: CPP, LV, and SP are used to hold the pointers to the constant pool, local variables, and the top of the stack, respectively, while PC holds the address of the next byte to be fetched from the instruction stream. MBR is a 1-byte register that sequentially holds the bytes of the instruction stream as they come in from memory to be interpreted. TOS and OPC are extra registers. Their use is described below.

At certain times, each of these registers is guaranteed to hold a certain value, but each can be used as a temporary register if needed. At the beginning and end of each instruction, TOS contains the value of the memory location pointed to by SP, the top word on the stack. This value is redundant since it can always be read from memory, but having it in a register often saves a memory reference. For a few instructions maintaining TOS means *more* memory operations. For example, the POP instruction throws away the top word and therefore must fetch the new top-of-stack word from the memory into TOS.

The OPC register is a temporary (i.e., scratch) register. It has no preassigned use. It is used, for example, to save the address of the opcode for a branch instruction while PC is incremented to access parameters. It is also used as a temporary register in the IJVM conditional branch instructions.

Like all interpreters, the microprogram of Fig. 4-17 has a main loop that fetches, decodes, and executes instructions from the program being interpreted, in this case, IJVM instructions. Its main loop begins on the line labeled Main1. It starts with the invariant that PC has previously been loaded with an address of a memory location containing an opcode. Furthermore, that opcode has already been fetched into MBR. Note this implies, however, that when we get back to this location, we must ensure that PC has been updated to point to the next opcode to be interpreted and the opcode byte itself has already been fetched into MBR.

This initial instruction sequence is executed at the beginning of every instruction, so it is important that it be as short as possible. Through careful design of the Mic-1 hardware and software, we have reduced the main loop to only a single microinstruction. Once the machine has started, every time this microinstruction is

executed, the IJVM opcode to execute is already present in MBR. What the microinstruction does is branch to the microcode for executing this IJVM instruction and also begin fetching the byte following the opcode, which may be either an operand byte or the next opcode.

Now we can reveal the real reason each microinstruction explicitly names its successor, instead of having them executed sequentially. All the control store addresses corresponding to opcodes must be reserved for the first word of the corresponding instruction interpreter. Thus from Fig. 4-11 we see that the code that interprets POP starts at 0x57 and the code that interprets DUP starts at 0x59. (How does MAL know to put POP at 0x57? Possibly there is a file that tells it.)

Unfortunately, the code for POP is three microinstructions long, so if placed in consecutive words, it would interfere with the start of DUP. Since all the control store addresses corresponding to opcodes are effectively reserved, the microinstructions other than the initial one in each sequence must be stuffed away in the holes between reserved addresses. For this reason, there is a great deal of jumping around, so having an explicit microbranch (a microinstruction that branches) every few microinstructions to hop from hole to hole would be very wasteful.

To see how the interpreter works, let us assume, for example, that MBR contains the value 0x60, that is, the opcode for IADD (see Fig. 4-11). In the one-microinstruction main loop we accomplish three things:

1. Increment the PC, leaving it containing the address of the first byte after the opcode.
2. Initiate a fetch of the next byte into MBR. This byte will always be needed sooner or later, either as an operand for the current IJVM instruction or as the next opcode (as in the case of the IADD instruction, which has no operand bytes).
3. Perform a multiway branch to the address contained in MBR at the start of Main1. This address is equal to the numerical value of the opcode currently being executed. It was placed there by the previous microinstruction. Note carefully that the value being fetched in this microinstruction does not play any role in the multiway branch.

The fetch of the next byte is started here so it will be available by the start of the third microinstruction. It may or may not be needed then, but it will be needed eventually, so starting the fetch now cannot do any harm in any case.

If the byte in MBR happens to be all zeros, the opcode for a NOP instruction, the next microinstruction is the one labeled nop1, fetched from location 0. Since this instruction does nothing, it simply branches back to the beginning of the main loop, where the sequence is repeated, but with a new opcode having been fetched into MBR.

Once again we emphasize that the microinstructions in Fig. 4-17 are not consecutive in memory and that Main1 is not at control store address 0 (because nop1