CHAPTER 9

MSI LOGIC CIRCUITS

OUTLINE

9-1	Decoders
9-2	BCD-to-7-Segment Decoder/Drivers

- 9-3 Liquid-Crystal Displays
- 9-4 Encoders
- 9-5 Troubleshooting
- 9-6 Multiplexers (Data Selectors)
- 9-7 Multiplexer Applications
- 9-8 Demultiplexers (Data Distributors)
- 9-9 More Troubleshooting
- 9-10 Magnitude Comparator
- 9-11 Code Converters

- 9-12 Data Busing
- 9-13 The 74ALS173/HC173 Tristate Register
- 9-14 Data Bus Operation
- 9-15 Decoders Using HDL
- 9-16 The HDL 7-Segment Decoder/Driver
- 9-17 Encoders Using HDL
- 9-18 HDL Multiplexers and Demultiplexers
- 9-19 HDL Magnitude Comparators
- 9-20 HDL Code Converters

OBJECTIVES

Upon completion of this chapter, you will be able to:

- Analyze and use decoders and encoders in various types of circuit applications.
- Compare the advantages and disadvantages of LEDs and LCDs.
- Utilize the observation/analysis technique for troubleshooting digital circuits.
- Understand the operation of multiplexers and demultiplexers by analyzing several circuit applications.
- Compare two binary numbers by using the magnitude comparator circuit.
- Understand the function and operation of code converters.
- Cite the precautions that must be considered when connecting digital circuits using the data bus concept.
- Use HDL to implement the equivalent of MSI logic circuits.

INTRODUCTION

Digital systems obtain binary-coded data and information that are continuously being operated on in some manner. Some of the operations include: (1) *decoding and encoding*, (2) *multiplexing*, (3) *demultiplexing*, (4) *comparison*, (5) *code conversion*, and (6) *data busing*. All of these operations and others have been facilitated by the availability of numerous ICs in the MSI (medium-scale-integration) category.

In this chapter, we will study many of the common types of MSI devices. For each type, we will start with a brief discussion of its basic operating principle and then introduce specific ICs. We then show how they can be used alone or in combination with other ICs in various applications.

9-1 DECODERS

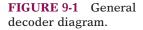
A **decoder** is a logic circuit that accepts a set of inputs that represents a binary number and activates only the output that corresponds to that input number. In other words, a decoder circuit looks at its inputs, determines which binary number is present there, and activates the one output that corresponds to that number; all other outputs remain inactive. The

diagram for a general decoder is shown in Figure 9-1 with N inputs and M outputs. Because each of the N inputs can be 0 or 1, there are 2^N possible input combinations or codes. For each of these input combinations, only one of the M outputs will be active (HIGH); all the other outputs are LOW. Many decoders are designed to produce active-LOW outputs, where only the selected output is LOW while all others are HIGH. This situation is indicated by the presence of small circles on the output lines in the decoder diagram.

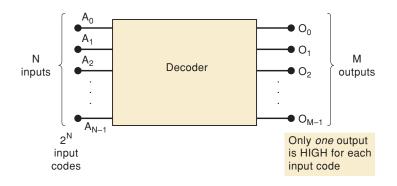
Some decoders do not utilize all of the 2^N possible input codes but only certain ones. For example, a BCD-to-decimal decoder has a four-bit input code and *ten* output lines that correspond to the *ten* BCD code groups 0000 through 1001. Decoders of this type are often designed so that if any of the unused codes are applied to the input, *none* of the outputs will be activated.

In Chapter 7, we saw how decoders are used in conjunction with counters to detect the various states of the counter. In that application, the FFs in the counter provided the binary code inputs for the decoder. The same basic decoder circuitry is used no matter where the inputs come from. Figure 9-2 shows the circuitry for a decoder with three inputs and $2^3 = 8$ outputs. It uses all AND gates, and so the outputs are active-HIGH. Note that for a given input code, the only output that is active (HIGH) is the one corresponding to the decimal equivalent of the binary input code (e.g., output O_6 goes HIGH only when $CBA = 110_2 = 6_{10}$).

This decoder can be referred to in several ways. It can be called a *3-line-to-8-line decoder* because it has three input lines and eight output lines. It can also be called a *binary-to-octal decoder* or *converter* because it takes a three-bit binary input code and activates one of the eight (octal) outputs corresponding to that code. It is also referred to as a *1-of-8 decoder* because only 1 of the 8 outputs is activated at one time.

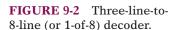






ENABLE Inputs

Some decoders have one or more ENABLE inputs that are used to control the operation of the decoder. For example, refer to the decoder in Figure 9-2 and visualize having a common ENABLE line connected to a fourth input of each gate. With this ENABLE line held HIGH, the decoder will function normally, and the *A*, *B*, *C* input code will determine which output is HIGH. With ENABLE held LOW, however, *all* of the outputs will be forced to the LOW state regardless of the levels at the *A*, *B*, *C* inputs. Thus, the decoder is enabled only if ENABLE is HIGH.



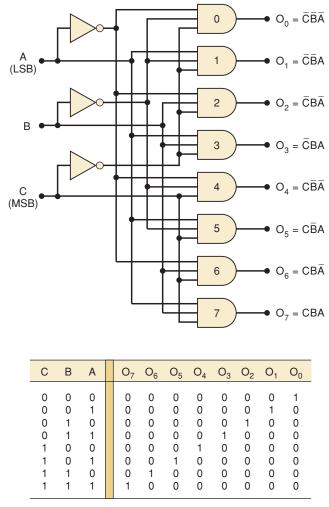
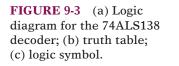


Figure 9-3(a) shows the logic diagram for the 74ALS138 decoder. By examining this diagram carefully, we can determine exactly how this decoder functions. First, notice that it has NAND gate outputs, so its outputs are active-LOW. Another indication is the labeling of the outputs as \overline{O}_7 , \overline{O}_6 , \overline{O}_5 , and so on; the overbar indicates active-LOW outputs.

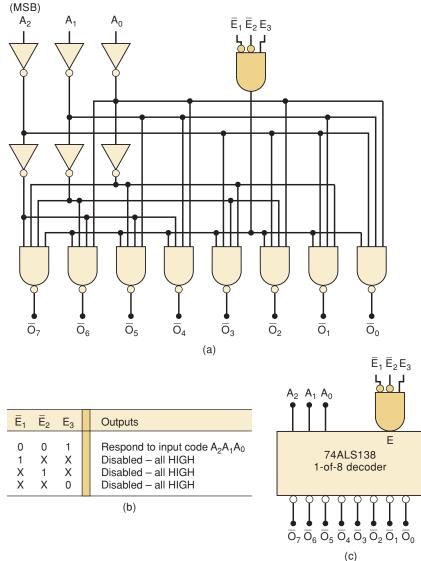
The input code is applied at A_2 , A_1 , and A_0 , where A_2 is the MSB. With three inputs and eight outputs, this is a 3-to-8 decoder or, equivalently, a 1-of-8 decoder.

Inputs E_1 , E_2 , and E_3 are separate enable inputs that are combined in the AND gate. In order to enable the output NAND gates to respond to the input code at $A_2A_1A_0$, this AND gate output must be HIGH. This will occur only when $\overline{E_1} = \overline{E_2} = 0$ and $E_3 = 1$. In other words, $\overline{E_1}$ and $\overline{E_2}$ are active-LOW, E_3 is active-HIGH, and all three must be in their active states to activate the decoder outputs. If one or more of the enable inputs is in its inactive state, the AND output will be LOW, which will force all NAND outputs to their inactive HIGH state regardless of the input code. This operation is summarized in the truth table in Figure 9-3(b). Recall that *x* represents the don't-care condition.

The logic symbol for the 74ALS138 is shown in Figure 9-3(c). Note how the active-LOW outputs are represented and how the enable inputs are represented. Even though the enable AND gate is shown as external to the decoder block, it is part of the IC's internal circuitry. The 74HC138 is the high-speed CMOS version of this decoder.







EXAMPLE 9-1

Indicate the states of the 74ALS138 outputs for each of the following sets of inputs.

(a) $E_3 = \overline{E}_2 = 1, \overline{E}_1 = 0, A_2 = A_1 = 1, A_0 = 0$ (b) $E_3 = 1, \overline{E}_2 = \overline{E}_1 = 0, A_2 = 0, A_1 = A_0 = 1$

Solution

- (a) With $\overline{E}_2 = 1$, the decoder is disabled and all of its outputs will be in their inactive HIGH state. This can be determined from the truth table or by following the input levels through the circuit logic.
- (b) All of the enable inputs are activated, so the decoding portion is enabled. It will decode the input code $011_2 = 3_{10}$ to activate output \overline{O}_3 . Thus, \overline{O}_3 will be LOW and all other outputs will be HIGH.

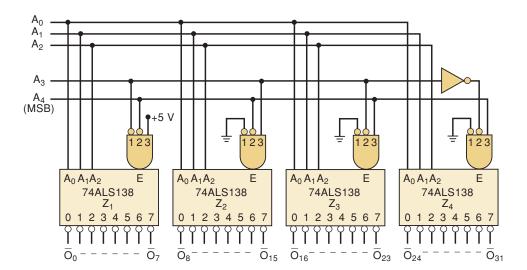
EXAMPLE 9-2

Figure 9-4 shows how four 74ALS138s and an INVERTER can be arranged to function as a 1-of-32 decoder. The decoders are labeled Z_1 to Z_4 for easy reference, and the eight outputs from each one are combined into 32 outputs. Z_1 's outputs are \overline{O}_0 to \overline{O}_7 ; Z_2 's outputs \overline{O}_0 to \overline{O}_7 are renamed \overline{O}_8 to \overline{O}_{15} , respectively; Z_3 's outputs are renamed \overline{O}_{16} to \overline{O}_{23} ; and Z_4 's are renamed \overline{O}_{24} to \overline{O}_{31} . A five-bit input code $A_4A_3A_2A_1A_0$ will activate only one of these 32 outputs for each of the 32 possible input codes.

- (a) Which output will be activated for $A_4A_3A_2A_1A_0 = 01101$?
- (b) What range of input codes will activate the Z_4 chip?

FIGURE 9-4 Four 74ALS138s forming a 1-of-32 decoder.



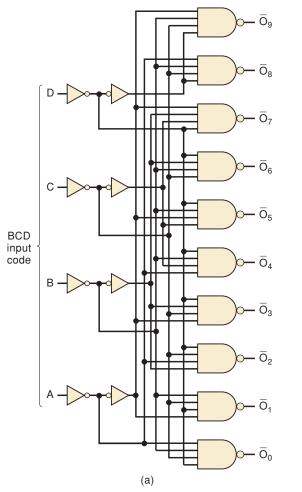


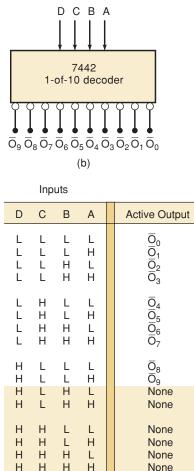
Solution

- (a) The five-bit code has two distinct portions. The A_4 and A_3 bits determine which one of the decoder chips Z_1 to Z_4 will be enabled, while $A_2A_1A_0$ determine which output of the enabled chip will be activated. With $A_4A_3 = 01$, only Z_2 has all of its enable inputs activated. Thus, Z_2 responds to the $A_2A_1A_0 = 101$ code and activates its \overline{O}_5 output, which has been renamed \overline{O}_{13} . Thus, the input code 01101, which is the binary equivalent of decimal 13, will cause output \overline{O}_{13} to go LOW, while all others stay HIGH.
- (b) To enable Z_4 , both A_4 and A_3 must be HIGH. Thus, all input codes ranging from 11000 (24₁₀) to 11111 (31₁₀) will activate Z_4 . This corresponds to outputs \overline{O}_{24} to \overline{O}_{31} .

BCD-to-Decimal Decoders

Figure 9-5(a) shows the logic diagram for a 7442 **BCD-to-decimal decoder**. It is also available as a 74LS42 and a 74HC42. Each output goes LOW only when its corresponding BCD input is applied. For example, \overline{O}_5 will go LOW only when inputs DCBA = 0101; \overline{O}_8 will go LOW only when DCBA = 1000. For input combinations that are invalid for BCD, none of the outputs will be activated. This decoder can also be referred to as a 4-to-10 decoder or a 1-of-10 decoder. The logic symbol and the truth table for the 7442 are also shown in





H = HIGH Voltage Level L = LOW Voltage Level

(c)

FIGURE 9-5 (a) Logic diagram for the 7442 BCD-to-decimal decoder; (b) logic symbol; (c) truth table.

the figure. Note that this decoder does not have an enable input. In Problem 9-7, we will see how the 7442 can be used as a 3-to-8 decoder, with the D input used as an enable input.

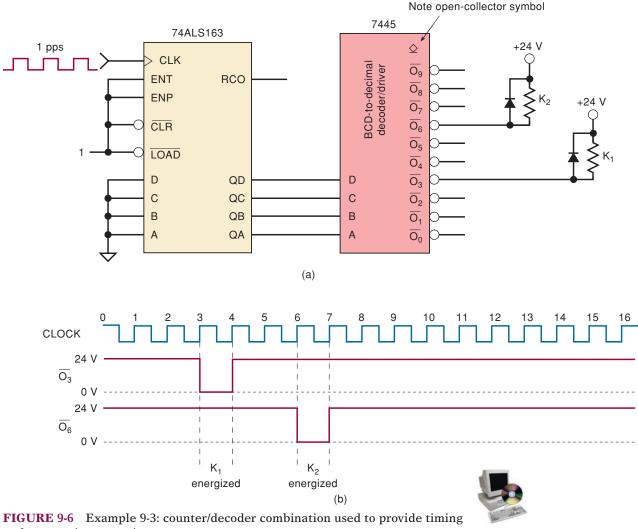
BCD-to-Decimal Decoder/Driver

The TTL 7445 is a BCD-to-decimal decoder/**driver**. The term *driver* is added to its description because this IC has open-collector outputs that can operate at higher current and voltage limits than a normal TTL output. The 7445's outputs can sink up to 80 mA in the LOW state, and they can be pulled up to 30 V in the HIGH state. This makes them suitable for directly driving loads such as indicator LEDs or lamps, relays, or dc motors.

Decoder Applications

Decoders are used whenever an output or a group of outputs is to be activated only on the occurrence of a specific combination of input levels. These input levels are often provided by the outputs of a counter or a register.

When the decoder inputs come from a counter that is being continually pulsed, the decoder outputs will be activated sequentially, and they can be used as timing or sequencing signals to turn devices on or off at specific times. An example of this operation is shown in Figure 9-6 using the 74ALS163 counter and the 7445 decoder/driver described above.



and sequencing operations.

EXAMPLE 9-3

Describe the operation of the circuit in Figure 9-6(a).

Solution

The counter is being pulsed by a 1-pps signal so that it will sequence through the binary counts at the rate of 1 count/s. The counter FF outputs are connected as the inputs to the decoder. The 7445 open-collector outputs \overline{O}_3 and \overline{O}_6 are used to switch relays K_1 and K_2 on and off. For instance, when \overline{O}_3 is in its inactive HIGH state, its output transistor will be off (nonconducting) so that no current can flow through relay K_1 and it will be deenergized. When \overline{O}_3 is in its active-LOW state, its output transistor is on and acts as a current sink for current through K_1 so that K_1 is energized. Note that the relays operate from +24 V. Also note the presence of the diodes across the relay coils; these protect the decoder's output transistors from the large "inductive kick" voltage that would be produced when coil current is stopped abruptly.

The timing diagram in Figure 9-6(b) shows the sequence of events. If we assume that the counter is in the 0000 state at time 0, then both outputs \overline{O}_3 and \overline{O}_6 are initially in the inactive HIGH state, where their output transistors are off and both relays are deenergized. As clock pulses are applied, the counter will be incremented once per second. On the NGT of the third pulse (time 3), the counter will go to the 0011 (3) state. This will activate decoder output \overline{O}_3 and thereby energize K_1 . On the NGT of the fourth pulse, the counter goes to the 0100 (4) state. This will deactivate \overline{O}_3 and deenergize relay K_1 .

Similarly, at time 6, the counter will go to the 0110 (6) state; this will make $\overline{O}_6 = 0$ and energize K_2 . At time 7, the counter goes to 0111 (7) and deactivates O_6 to deenergize K_2 .

The counter will continue counting as pulses are applied. After 16 pulses, the sequence just described will start over.

Decoders are widely used in the memory system of a computer where they respond to the address code generated by the central processor to activate a particular memory location. Each memory IC contains many registers that can store binary numbers (data). Each register needs to have its own unique address to distinguish it from all the other registers. A decoder is built into the memory IC's circuitry and allows a particular storage register to be activated when a unique combination of inputs (i.e., its address) is applied. In a system, there are usually several memory ICs combined to make up the entire storage capacity. A decoder is used to select a memory chip in response to a range of addresses by decoding the most significant bits of the system address and enabling (selecting) a particular chip. We will examine this application in Problem 9-63, and we will study it in much more depth when we read about memories in Chapter 12.

In more complicated memory systems, the memory chips are arranged in multiple banks that must be selected individually or simultaneously, depending on whether the microprocessor wants one or more bytes at a time. This means that under certain circumstances, more than one output of the decoder must be activated. For systems such as this, a programmable logic device is often used to implement the decoder because a simple 1-of-8 decoder alone is not sufficient. Programmable logic devices can be used easily for custom decoding applications.

REVIEW QUESTIONS	1. Can more than one decoder output be activated at one time?
	2. What is the function of a decoder's enable input(s)?
	3. How does the 7445 differ from the 7442?
	4. The 74154 is a 4-to-16 decoder with two active-LOW enable inputs. How many pins (including power and ground) does this IC have?

9-2 BCD-TO-7-SEGMENT DECODER/DRIVERS

Most digital equipment has some means for displaying information in a form that can be understood readily by the user or operator. This information is often numerical data but can also be alphanumeric (numbers and letters). One

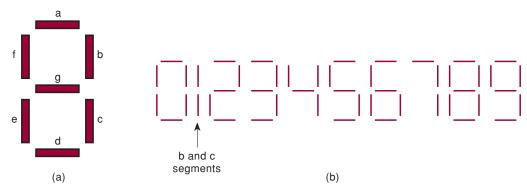


FIGURE 9-7 (a) 7-segment arrangement; (b) active segments for each digit.

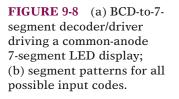
of the simplest and most popular methods for displaying numerical digits uses a 7-segment configuration [Figure 9-7(a)] to form the decimal characters 0 through 9 and sometimes the hex characters A through F. One common arrangement uses light-emitting diodes (LEDs) for each segment. By controlling the current through each LED, some segments will be light and others will be dark so that the desired character pattern will be generated. Figure 9-7(b) shows the segment patterns that are used to display the various digits. For example, to display a "6," the segments a, c, d, e, f, and g are made bright while segment b is dark.

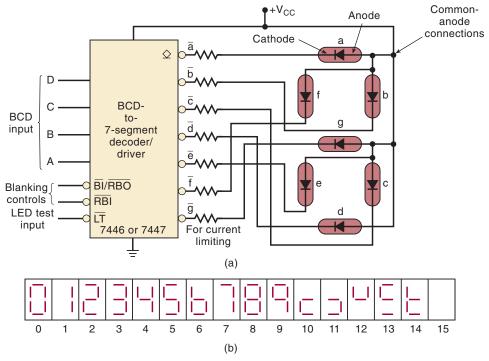
A **BCD-to-7-segment decoder/driver** is used to take a four-bit BCD input and provide the outputs that will pass current through the appropriate segments to display the decimal digit. The logic for this decoder is more complicated than the logic of decoders that we have looked at previously because each output is activated for more than one combination of inputs. For example, the *e* segment must be activated for any of the digits 0, 2, 6, and 8, which means whenever any of the codes 0000, 0010, 0110, or 1000 occurs.

Figure 9-8(a) shows a BCD-to-7-segment decoder/driver (TTL 7446 or 7447) being used to drive a 7-segment LED readout. Each segment consists of an LED (light-emitting diode). Diodes are solid-state devices that allow current to flow through them in one direction, but block the flow in the other direction. Whenever the anode of an LED is more positive than the cathode by approximately 2 V, the LED will light up. The anodes of the LEDs are all tied to V_{CC} (+5 V). The cathodes of the LEDs are connected through current-limiting resistors to the appropriate outputs of the decoder/driver. The decoder/driver has active-LOW outputs that are open-collector driver transistors and can sink a fairly large current because LED readouts may require 10 to 40 mA per segment, depending on their type and size.

To illustrate the operation of this circuit, let us suppose that the BCD input is D = 0, C = 1, B = 0, A = 1, which is BCD for 5. With these inputs, the decoder/driver outputs $\overline{a}, \overline{f}, \overline{g}, \overline{c}$, and \overline{d} will be driven LOW (connected to ground), allowing current to flow through the a, f, g, c, and d LED segments and thereby displaying the numeral 5. The \overline{b} and \overline{e} outputs will be HIGH (open), so that LED segments b and e cannot conduct.

The 7446/47 decoder/drivers are designed to activate specific segments even for non-BCD input codes (greater than 1001). Figure 9-8(b) shows the activated segment patterns for all possible input codes from 0000 to 1111. Note that an input code of 1111 (15) will blank out all the segments.





Seven-segment decoder/drivers such as the 7446/47 are exceptions to the rule that decoder circuits activate only one output for each combination of inputs. Rather, they activate a unique pattern of outputs for each combination of inputs.

Common-Anode Versus Common-Cathode LED Displays

The LED display used in Figure 9-8 is a **common-anode** type because the anodes of all of the segments are tied together to V_{CC} . Another type of 7-segment LED display uses a **common-cathode** arrangement where the cathodes of all of the segments are tied together and connected to ground. This type of display must be driven by a BCD-to-7-segment decoder/driver with active-HIGH outputs that apply a HIGH voltage to the anodes of those segments that are to be activated. Because each segment requires 10 to 20 mA of current to light it, TTL and CMOS devices are normally not used to drive the common-cathode display directly. Recall from Chapter 8 that TTL and CMOS outputs are not able to source large amounts of current. A transistor interface circuit is often used between decoder chips and the common-cathode display.

EXAMPLE 9-4

Each segment of a typical 7-segment LED display is rated to operate at 10 mA at 2.7 V for normal brightness. Calculate the value of the current-limiting resistor needed to produce approximately 10 mA per segment.

Solution

Referring to Figure 9-8(a), we can see that the series resistor must have a voltage drop equal to the difference between $V_{CC} = 5$ V and the segment voltage of 2.7 V. This 2.3 V across the resistor must produce a current of about 10 mA. Thus, we have

$$R_S = \frac{2.3 \text{ V}}{10 \text{ mA}} = 230 \Omega$$

A standard resistor value close to this can be used. A 220- Ω resistor would be a good choice.

REVIEW QUESTIONS

- 1. Which LED segments will be on for a decoder/driver input of 1001?
- 2. *True or false:* More than one output of a BCD-to-7-segment decoder/driver can be active at one time.

9-3 LIQUID-CRYSTAL DISPLAYS

An LED display generates or emits light energy as current is passed through the individual segments. A liquid-crystal display (LCD) controls the reflection of available light. The available light may simply be ambient (surrounding) light such as sunlight or normal room lighting; *reflective* LCDs use ambient light. Or the available light might be provided by a small light source that is part of the display unit; *backlit* LCDs use this method. In any case, LCDs have gained wide acceptance because of their very low power consumption compared to LEDs, especially in battery-operated equipment such as calculators, digital watches, and portable electronic measuring instruments. LEDs have the advantage of a much brighter display that, unlike reflective LCDs, is easily visible in dark or poorly lit areas.

Basically, LCDs operate from a low-voltage (typically 3 to 15 V rms), lowfrequency (25 to 60 Hz) ac signal and draw very little current. They are often arranged as 7-segment displays for numerical readouts as shown in Figure 9-9(a). The ac voltage needed to turn on a segment is applied between the segment and the **backplane**, which is common to all segments. The segment and the backplane form a capacitor that draws very little current as long as the ac frequency is kept low. It is generally not lower than 25 Hz because this would produce visible flicker.

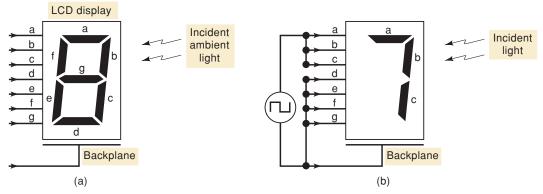


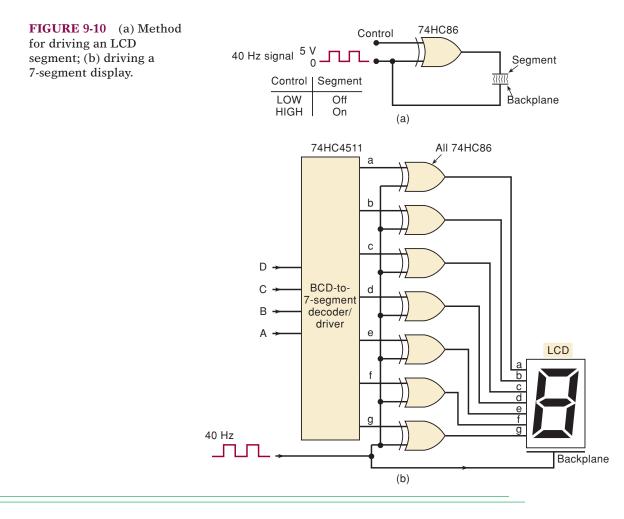
FIGURE 9-9 Liquid-crystal display: (a) basic arrangement; (b) applying a voltage between the segment and the backplane turns ON the segment. Zero voltage turns the segment OFF.

An admittedly simplified explanation of how an LCD operates goes something like this. When there is no difference in voltage between a segment and the backplane, the segment is said to be *nonactivated* (OFF). Segments d, e, f, and g in Figure 9-9(b) are OFF and will reflect incident light so that they appear invisible against their background. When an appropriate ac voltage is applied between a segment and the backplane, the segment is activated (ON). Segments a, b, and c in Figure 9-9(b) are ON and will not reflect the incident light, and thus they appear dark against their background.

Driving an LCD

An LCD segment will turn ON when an ac voltage is applied between the segment and the backplane, and will turn OFF when there is no voltage between the two. Rather than generating an ac signal, it is common practice to produce the required ac voltage by applying out-of-phase square waves to the segment and the backplane. This is illustrated in Figure 9-10(a) for one segment. A 40-Hz square wave is applied to the backplane and also to the input of a CMOS 74HC86 XOR. The other input to the XOR is a CONTROL input that will control whether the segment is ON or OFF.

When the CONTROL input is LOW, the XOR output will be exactly the same as the 40-Hz square wave, so that the signals applied to the segment and



the backplane are equal. Because there is no difference in voltage, the segment will be OFF. When the CONTROL input is HIGH, the XOR output will be the INVERSE of the 40-Hz square wave, so that the signal applied to the segment is out of phase with the signal applied to the backplane. As a result, the segment voltage will alternately be at +5 V and at -5 V relative to the backplane. This ac voltage will turn ON the segment.

This same idea can be extended to a complete 7-segment LCD display, as shown in Figure 9-10(b). Here, the CMOS 74HC4511 BCD-to-7-segment decoder/driver supplies the CONTROL signals to each of seven XOR for the seven segments. The 74HC4511 has active-HIGH outputs because a HIGH is required to turn on a segment. The decoder/driver and XOR gates of Figure 9-10(b) are available on a single chip. The CMOS 74HC4543 is one such device. It takes the BCD input code and provides the outputs to drive the LCD segments directly.

In general, CMOS devices are used to drive LCDs for two reasons: (1) they require much less power than TTL and are more suited to the batteryoperated applications where LCDs are used; (2) the TTL LOW-state voltage is not exactly 0 V and can be as much as 0.4 V. This will produce a dc component of voltage between the segment and the backplane that considerably shortens the life of an LCD.

Types of LCDs

Liquid crystals are available as multidigit 7-segment decimal numeric displays. They come in many sizes and with many special characters such as colons (:) for clock displays, + and - indicators for digital voltmeters, decimal points for calculators, and battery-low indicators because many LCD devices are battery-powered. These displays must be driven by a decoder/driver chip such as the 74HC4543.

A more complicated but readily available LCD display is the alphanumeric LCD module. These modules are available from many companies in numerous formats such as 1-line-by-16-characters up to 4-lines-by-40-characters. The interface to these modules has been standardized so that an LCD module from any manufacturer will use the same signals and data format. The module includes some VLSI chips that make this device simple to use. Eight data lines are used to send the ASCII code for whatever you wish to display. These data lines also carry special control codes to the LCD command register. Three other inputs (Register Select, Read/Write, and Enable) are used to control the location, direction, and timing of the data transfer. As characters are sent to the module, it stores them in its own memory and types them across the display screen.

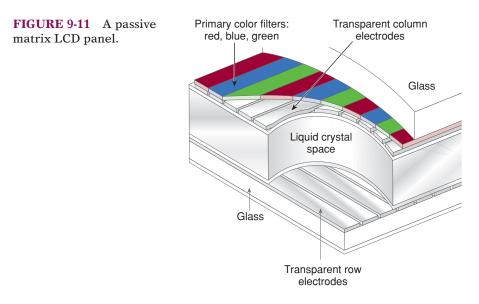
Other LCD modules allow the user to create a graphical display by controlling individual dots on the screen called **pixels**. Larger LCD panels can be scanned at a high rate, producing high-quality video motion pictures. In these displays, the control lines are arranged in a grid of rows and columns. At the intersection of each row and column is a pixel that acts like a "window" or "shutter" that can be electronically opened and closed to control the amount of light that is transmitted through the cell. The voltage from a row to a column determines the brightness of each pixel. In a laptop computer, a binary number for each pixel is stored in the "video" memory. These numbers are converted to voltages that are applied to the display.

Each pixel on a color display is actually made up of three subpixels. These subpixels control the light that passes through a red, green, or blue filter to produce the color of each pixel. On a 640-by-480 LCD screen there would be 640×3 connections for columns and 480 connections for rows, for a total of 2400 connections to the LCD. Obviously, the driver circuitry for such a device is a very complicated VLSI circuit.

The advances in technology for LCD displays have increased the speed at which the pixels can be turned on and off. The older screens are called Twisted Nematic (TN) or Super Twisted Nematic (STN). These devices are referred to as passive LCDs. Instead of using a uniform backplane like the 7-segment LCD displays, they have conducting parallel lines manufactured onto two pieces of glass. The two glass sheets are used to sandwich the liquid crystal material with the conducting lines at 90°, forming a grid of rows and columns, as shown in Figure 9-11. The intersection of each row and column forms a pixel. The actual switching of the current on and off is done in the driver IC that is connected to the rows and columns of the display. Passive matrix displays are rather slow at turning off. This limits the rate at which objects can move on the screen without leaving a shadow trail behind them.

The newer displays are called active matrix TFT LCDs. The active matrix means that an active element on the display is used to switch the pixels on and off. The active component is a thin film transistor (TFT) that is manufactured directly onto one piece of glass. The other piece of glass has a uniform coating to form a backplane. The control lines for these transistors run in rows and columns between the pixels. The technology that allows these transistors to be manufactured in a matrix on a thin film the size of a laptop computer screen has made these displays possible. They provide a much faster-response, higher-resolution display. The use of polysilicon technology allows the driver circuits to be integrated into the display unit, reducing connection problems and requiring very little perimeter space around the LCD.

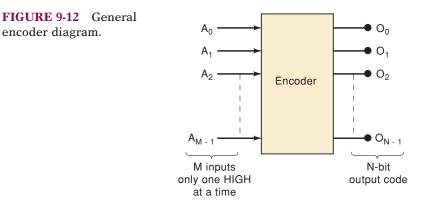
Other display technologies are being refined, including vacuum fluorescent, gas discharge plasma, and electroluminescence. The optical physics for each of these displays varies, but the means of controlling all of them is the same. A digital system must activate a row and a column of a matrix in order to control the amount of light at the pixel located at the row/column intersection.



REVIEW QUESTIONS	1. Indicate which of the following statements refer to LCD displays and which refer to LED displays.
	(a) Emit light
	(b) Reflect ambient light
	(c) Are best for low-power applications
	(d) Require an ac voltage
	(e) Use a 7-segment arrangement to produce digits
	(f) Require current-limiting resistors
	2. What form of data is sent to each of the following?
	(a) A 7-segment LCD display with a decoder/driver
	(b) An alphanumeric LCD module
	(c) An LCD computer display

9-4 ENCODERS

Most decoders accept an input code and produce a HIGH (or a LOW) at *one* and only one output line. In other words, we can say that a decoder identifies, recognizes, or detects a particular code. The opposite of this decoding process is called **encoding** and is performed by a logic circuit called an **encoder**. An encoder has a number of input lines, only one of which is activated at a given time, and produces an *N*-bit output code, depending on which input is activated. Figure 9-12 is the general diagram for an encoder with *M* inputs and *N* outputs. Here, the inputs are active-HIGH, which means that they are normally LOW.



We saw that a *binary-to-octal decoder* (3-line-to-8-line decoder) accepts a three-bit input code and activates one of eight output lines corresponding to that code. An *octal-to-binary encoder* (8-line-to-3-line encoder) performs the opposite function: it accepts eight input lines and produces a three-bit output code corresponding to the activated input. Figure 9-13 shows the logic circuit and the truth table for an octal-to-binary encoder with active-LOW inputs.

By following through the logic, you can verify that a LOW at any single input will produce the output binary code corresponding to that input. For instance, a LOW at \overline{A}_3 (while all other inputs are HIGH) will produce

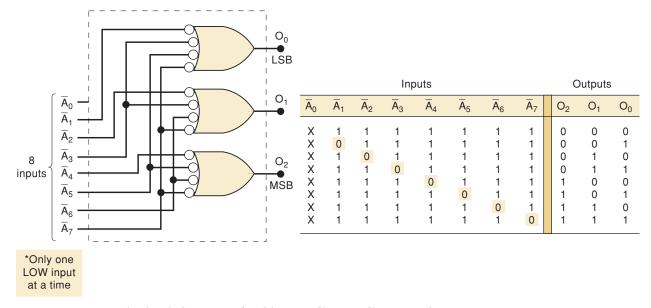


FIGURE 9-13 Logic circuit for an octal-to-binary (8-line-to-3-line) encoder. For proper operation, only one input should be active at one time.

 $O_2 = 0, O_1 = 1$, and $O_0 = 1$, which is the binary code for 3. Notice that \overline{A}_0 is not connected to the logic gates because the encoder outputs will normally be at 000 when none of the inputs \overline{A}_1 to \overline{A}_9 is LOW.

EXAMPLE 9-5

Determine the outputs of the encoder in Figure 9-13 when \overline{A}_3 and \overline{A}_5 are simultaneously LOW.

Solution

Following through the logic gates, we see that the LOWs at these two inputs will produce HIGHs at each output, in other words, the binary code 111. Clearly, this is not the code for either activated input.

Priority Encoders

This last example identifies a drawback of the simple encoder circuit of Figure 9-13 when more than one input is activated at one time. A modified version of this circuit, called a **priority encoder**, includes the necessary logic to ensure that when two or more inputs are activated, the output code will correspond to the highest-numbered input. For example, when both \overline{A}_3 and \overline{A}_5 are LOW, the output code will be 101 (5). Similarly, when $\overline{A}_6, \overline{A}_2$, and \overline{A}_0 are all LOW, the output code is 110 (6). The 74148, 74LS148, and 74HC148 are all octal-to-binary priority encoders.

74147 Decimal-to-BCD Priority Encoder

Figure 9-14 shows the logic symbol and the truth table for the 74147 (74LS147, 74HC147), which functions as a decimal-to-BCD priority encoder. It has nine active-LOW inputs representing the decimal digits 1 through 9, and it produces the *inverted* BCD code corresponding to the highest-numbered activated input.

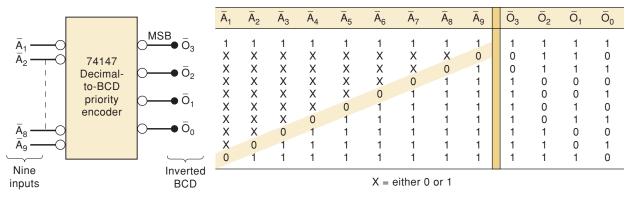


FIGURE 9-14 74147 decimal-to-BCD priority encoder.



Let's examine the truth table to see how this IC works. The first line in the table shows all inputs in their inactive HIGH state. For this condition, the outputs are 1111, which is the inverse of 0000, the BCD code for 0. The second line in the table indicates that a LOW at \overline{A}_9 , regardless of the states of the other inputs, will produce an output code of 0110, which is the inverse of 1001, the BCD code for 9. The third line shows that a LOW at \overline{A}_8 , provided that \overline{A}_9 is HIGH, will produce an output code of 0111, the inverse of 1000, the BCD code for 8. In a similar manner, the remaining lines in the table show that a LOW at any input, provided that all higher-numbered inputs are HIGH, will produce the inverse of the BCD code for that input.

The 74147 outputs will normally be HIGH when none of the inputs are activated. This corresponds to the decimal 0 input condition. There is no \overline{A}_0 input because the encoder assumes the decimal 0 input state when all other inputs are HIGH. The 74147 inverted BCD outputs can be converted to normal BCD by putting each one through an INVERTER.

EXAMPLE 9-6

Determine the states of the outputs in Figure 9-14 when \overline{A}_5 , \overline{A}_7 , and \overline{A}_3 are LOW and all other inputs are HIGH.

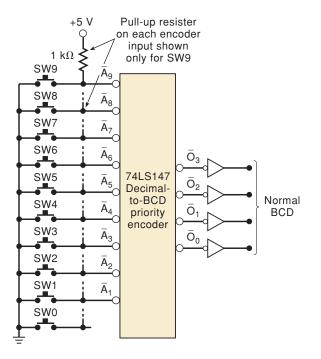
Solution

The truth table shows that when \overline{A}_7 is LOW, the levels at \overline{A}_5 and \overline{A}_3 do not matter. Thus, the outputs will each be 1000, the inverse of 0111 (7).

Switch Encoder

Figure 9-15 shows how a 74147 can be used as a *switch encoder*. The 10 switches might be the keyboard switches on a calculator representing digits 0 through 9. The switches are of the normally open type, so that the encoder inputs are all normally HIGH and the BCD output is 0000 (note the INVERTERs). When a digit key is depressed, the circuit will produce the BCD code for that digit. Because the 74LS147 is a priority encoder, simultaneous key depressions will produce the BCD code for the higher-numbered key.

The switch encoder of Figure 9-15 can be used whenever BCD data must be entered manually into a digital system. A prime example would be in an electronic calculator, where the operator depresses several keyboard switches in succession to enter a decimal number. In a simple, basic calculator, the BCD code for each decimal digit is entered into a four-bit storage register. In other **FIGURE 9-15** Decimal-to-BCD switch encoder.



words, when the first key is depressed, the BCD code for that digit is sent to a four-bit FF register; when the second switch is depressed, the BCD code for that digit is sent to *another* four-bit FF register, and so on. Thus, a calculator that can handle eight digits will have eight four-bit registers to store the BCD codes for these digits. Each four-bit register drives a decoder/driver and a numerical display so that the eight-digit number can be displayed.

The operation described above can be accomplished with the circuit in Figure 9-16. This circuit will take three decimal digits entered from the keyboard in sequence, encode them in BCD, and store the BCD in three FF output registers. The 12 D-type flip-flops Q_0 to Q_{11} are used to receive and store the BCD codes for the digits. Q_8 to Q_{11} store the BCD code for the most significant digit (MSD), which is the first one entered on the keyboard. Q_4 to Q_7 store the second entered digit, and Q_0 to Q_3 store the third entered digit. Flip-flops X, Y, and Z form a ring counter (Chapter 7) that controls the transfer of data from the encoder outputs to the appropriate output register. The OR gate produces a HIGH output any time one of the keys is depressed. This output may be affected by switch contact bounce, which would produce several pulses before settling down to the HIGH state. The OS is used to neutralize the switch bounce by triggering on the first positive transition from the OR gate and remaining HIGH for 20 ms, well past the time duration of the switch bounce. The OS output clocks the ring counter.

The circuit operation is described as follows for the case where the decimal number 309 is being entered:

- 1. The CLEAR key is depressed. This clears all storage flip-flops Q_0 to Q_{11} to 0. It also clears flip-flops X and Y and presets flip-flop Z to 1, so that the ring counter begins in the 001 state.
- 2. The CLEAR key is released and the "3" key is depressed. The encoder outputs 1100 are inverted to produce 0011, the BCD code for 3. These binary values are sent to the *D* inputs of the three four-bit output registers.
- 3. The OR output goes HIGH (because two of its inputs are HIGH) and triggers the OS output Q = 1 for 20 ms. After 20 ms, Q returns LOW and clocks

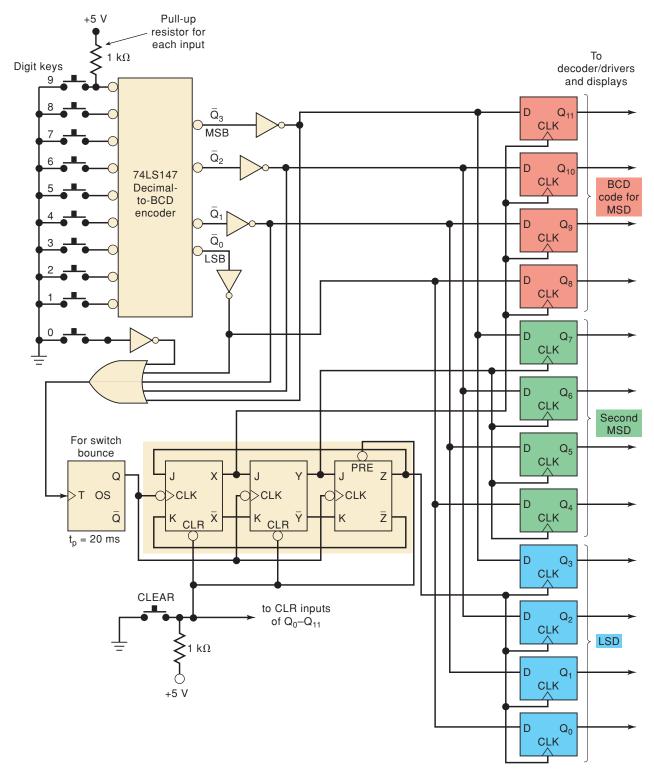


FIGURE 9-16 Circuit for keyboard entry of three-digit number into storage registers.

the ring counter to the 100 state (X goes HIGH). The positive transition at X is fed to the *CLK* inputs of flip-flops Q_8 to Q_{11} , so that the encoder outputs are transferred to these FFs. That is, $Q_{11} = 0$, $Q_{10} = 0$, $Q_9 = 1$, and $Q_8 = 1$. Note that flip-flops Q_0 to Q_7 are not affected because their *CLK* inputs have not received a positive transition.

- 4. The "3" key is released and the OR gate output returns LOW. The "0" key is then depressed. This produces the BCD code of 0000, which is fed to the inputs of the three registers.
- 5. The OR output goes HIGH in response to the "0" key (note the IN-VERTER) and triggers the OS for 20 ms. After 20 ms, the ring counter shifts to the 010 state (Y goes HIGH). The positive transition at Y is fed to the *CLK* inputs of Q_4 to Q_7 and transfers the 0000 to these FFs. Note that flip-flops Q_0 to Q_3 and Q_8 to Q_{11} are not affected by the Y transition.
- 6. The "0" key is released and the OR output returns LOW. The "9" key is depressed, producing BCD outputs 1001, which are fed to the storage registers.
- 7. The OR output goes HIGH again, triggering the OS, which in turn clocks the ring counter to the 001 state (Z goes HIGH). The positive transition at Z is fed to the *CLK* inputs of Q_0 to Q_3 and transfers the 1001 into these FFs. The other storage FFs are unaffected.
- 8. At this point, the storage register contains 001100001001, beginning with Q_{11} . This is the BCD code of 309. These register outputs feed decoder/drivers that drive appropriate displays for indicating the decimal digits 309.
- 9. The storage FF outputs are also fed to other circuits in the system. In a calculator, for example, these outputs would be sent to the arithmetic section to be processed.

Several problems at the end of the chapter will deal with some other aspects of this circuit, including troubleshooting exercises.

The 74ALS148 is slightly more sophisticated than the '147. It has eight inputs that are encoded into a three-bit binary number. This IC also provides three control pins as indicated in Table 9-1. The Enable Input ($\overline{\text{EI}}$) and Enable Output ($\overline{\text{EO}}$) can be used to cascade two IC's producing a hexadecimal-tobinary encoder. The $\overline{\text{EI}}$ pin must be LOW in order for any output pin to go LOW, and the $\overline{\text{EO}}$ pin will only go LOW when none of the eight inputs is active and the $\overline{\text{EI}}$ is active. The $\overline{\text{GS}}$ output is used to indicate when at least one of the eight inputs is activated. It should be noted that the outputs A_2 through A_0 are inverted, just as in the 74147.

TABLE 9-174ALS148 function table.

	INPUTS									OUTPUTS			
EI	ō	1	2	3	4	5	6	8	\overline{A}_2	Ā ₁	\overline{A}_0	GS	ĒŌ
Н	х	х	х	х	х	х	х	х	н	Н	Н	н	Н
L	Н	Н	Н	Н	Н	Н	Н	Н	Н	Н	Н	Н	L
L	х	х	x	х	x	x	x	L	L	L	L	L	Н
L	х	х	х	х	х	х	L	Н	L	L	Н	L	Н
L	х	х	x	х	x	L	Н	Н	L	Н	L	L	Н
L	х	х	х	х	L	Н	Н	Н	L	Н	Н	L	Н
L	х	х	х	L	Н	Н	Н	Н	Н	L	L	L	Н
L	х	х	L	Н	Н	Н	Н	Н	Н	L	Н	L	Н
L	х	L	Н	Н	Н	Н	Н	Н	Н	Н	L	L	Н
L	L	Н	Н	Н	Н	Н	Н	Н	Н	Н	Н	L	Н

REVIEW QUESTIONS	1. How does an encoder differ from a decoder?
	2. How does a priority encoder differ from an ordinary encoder?
	3. What will the outputs be in Figure 9-15 when SW6, SW5, and SW2 are all closed?
	4. Describe the functions of each of the following parts of the keyboard en- try circuit of Figure 9-16.
	(a) OR gate (d) Flip-flops X, Y, Z
	(b) 74147 encoder (e) Flip-flops Q_0 to Q_{11}
	(c) One-shot
	5. What is the purpose of each control input and output on a 74148 encoder?

9-5 TROUBLESHOOTING

As circuits and systems become more complex, the number of possible causes of failure obviously increases. Whereas the procedure for fault isolation and correction remains essentially the same, the application of the **observation/analysis** process is more important for complex circuits because it helps the troubleshooter narrow the location of the fault to a small area of the circuit. This reduces to a reasonable amount the testing steps and resulting data that must be analyzed. By understanding the circuit operation, observing the symptoms of the failure, and reasoning through the operation, the troubleshooter can often predict the possible faults before ever picking up a logic probe or an oscilloscope. This observation/analysis process is one that inexperienced troubleshooters are hesitant to apply, probably because of the great variety and capabilities of modern test equipment available to them. It is easy to become overly reliant on these tools while not adequately utilizing the human brain's reasoning and analytical skills.

The following examples illustrate how the observation/analysis process can be applied. Many of the end-of-chapter troubleshooting problems will provide you with the opportunity to develop your skill at applying this process.

Another vital strategy in troubleshooting is known as **divide-and-conquer**. It is used to identify the location of the problem after observation/analysis has generated several possibilities. A less efficient method would be to investigate each possible cause, one by one. The divide-and-conquer method finds a point in the circuit that can be tested, thereby dividing the total possible number of causes in half. In simple systems, this may seem unnecessary, but as complexity increases, the total number of possible causes also increases. If there are eight possible causes, then a test should be performed that eliminates four of them. The next test should eliminate two more, and the third test should identify the problem.

EXAMPLE 9-7

A technician tests the circuit of Figure 9-4 by using a set of switches to apply the input code at A_4 through A_0 . She runs through each possible input code and checks the corresponding decoder output to see if it is activated. She observes that all of the odd-numbered outputs respond correctly, but all of the even-numbered outputs fail to respond when their code is applied. What are the most probable faults?

Solution

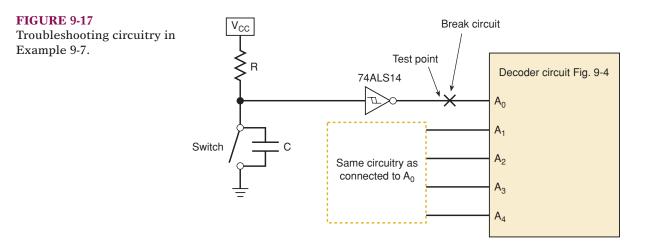
In a situation where so many outputs are failing, it is unreasonable to expect that each of these outputs has a fault. It is much more likely that some faulty input condition is causing the output failures. What do all of the evennumbered outputs have in common? The input codes for several of them are listed in Table 9-2.

Clearly, each even-numbered output requires an input code with an $A_0 = 0$ in order to be activated. Thus, the most probable faults would be those that prevent A_0 from going LOW. These include:

- 1. A faulty switch connected to the A_0 input
- 2. A break in the path between the switch and the A_0 line
- 3. An external short from the A_0 line to V_{CC}
- 4. An internal short to V_{CC} at the A_0 inputs of any one of the decoder chips

Through observation and analysis, the technician has identified several possible causes. Potential causes 1 and 2 are in the switches generating the address. Causes 3 and 4 are in the decoder circuit itself. The circuit can be divided by opening the connection between the least significant switch and the A_0 input, as shown in Figure 9-17. A logic probe can be used to see if the switch can generate a LOW as well as a HIGH. Regardless of the outcome, two of the four possible causes have been eliminated.

Thus, the fault is narrowed to a specific area of the circuit. The exact fault can be traced with the testing and measurement techniques that we are already familiar with.



EXAMPLE 9-8

A technician wires the outputs from a BCD counter to the inputs of the decoder/driver of Figure 9-8. He applies pulses to the counter at a very slow rate and observes the LED display, which is shown below, as the counter counts up from 0000 to 1001. Examine this observed sequence carefully and try to predict the most probable fault.

TABLE 9-2

Output	Input Code
\overline{O}_0	00000
\overline{O}_4	00100
\overline{O}_{14}	01110
\overline{O}_{18}	10010

COUNT	0	1	2	3	4	5	6	7	8	9
Observed display		- {	Ū į		יר			7		
Expected display		- {			Ц		5	7		4

Solution

Comparing the observed display with the expected display for each count, we see several important points:

- For those counts where the observed display is incorrect, the observed display is not one of the segment patterns that correspond to counts greater than 1001.
- This rules out a faulty counter or faulty wiring from the counter to the decoder/driver.
- The correct segment patterns (0, 1, 3, 6, 7, and 8) have the common property that segments *e* and *f* are either both on or both off.
- The incorrect segment patterns have the common property that segments *e* and *f* are in opposite states, and if we interchange the states of these two segments, the correct pattern is obtained.

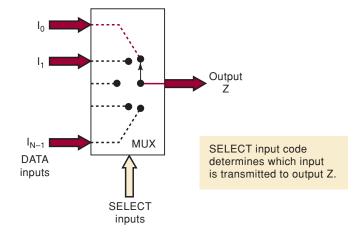
Giving some thought to these points should lead us to conclude that the technician has probably "crossed" the connections to the e and f segments.

9-6 MULTIPLEXERS (DATA SELECTORS)

A modern home stereo system may have a switch that selects music from one of four sources: a cassette tape, a compact disc (CD), a radio tuner, or an auxilliary input such as audio from a VCR or DVD. The switch selects one of the electronic signals from one of these four sources and sends it to the power amplifier and speakers. In simple terms, this is what a **multiplexer** (**MUX**) does: it selects one of several input signals and passes it on to the output.

A *digital multiplexer* or *data selector* is a logic circuit that accepts several digital data inputs and selects one of them at any given time to pass on to the output. The routing of the desired data input to the output is controlled by SELECT inputs (often referred to as ADDRESS inputs). Figure 9-18 shows the functional diagram of a general digital multiplexer. The inputs and outputs are drawn as wide arrows rather than lines; this indicates that they may actually be more than one signal line.

The multiplexer acts like a digitally controlled multiposition switch where the digital code applied to the SELECT inputs controls which data inputs will be switched to the output. For example, output Z will equal data input I_0 for some particular SELECT input code, Z will equal I_1 for another particular SELECT input code, and so on. Stated another way, a multiplexer selects 1 out of N input data sources and transmits the selected data to a single output channel. This is called **multiplexing**. FIGURE 9-18 Functional diagram of a digital multiplexer (MUX).



Basic Two-Input Multiplexer

Figure 9-19 shows the logic circuitry for a two-input multiplexer with data inputs I_0 and I_1 and SELECT input S. The logic level applied to the S input determines which AND gate is enabled so that its data input passes through the OR gate to output Z. Looking at it another way, the Boolean expression for the output is

$$Z = I_0 \overline{S} + I_1 S$$

With S = 0, this expression becomes

$$Z = I_0 \cdot 1 + I_1 \cdot 0$$

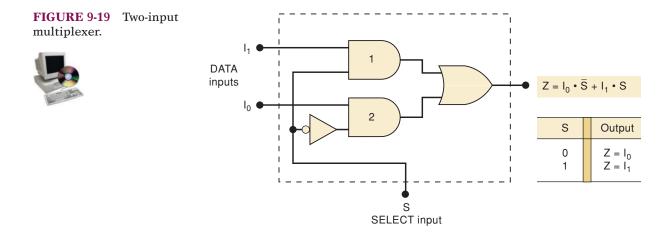
$$= I_0$$
[gate 2 enabled]

which indicates that Z will be identical to input signal I_0 , which in turn can be a fixed logic level or a time-varying logic signal. With S = 1, the expression becomes

$$Z = I_0 \cdot 0 + I_1 \cdot 1 = I_1 \qquad [gate 1 enabled]$$

showing that output *Z* will be identical to input signal I_1 .

An example of where a two-input MUX could be used is in a digital system that uses two different MASTER CLOCK signals: a high-speed clock (say, 10 MHz) in one mode and a slow-speed clock (say, 4.77 MHz) for the



other. Using the circuit of Figure 9-19, the 10-MHz clock would be tied to I_0 , and the 4.77-MHz clock would be tied to I_1 . A signal from the system's control logic section would drive the SELECT input to control which clock signal appears at output Z for routing to the other parts of the circuit.

Four-Input Multiplexer

The same basic idea can be used to form the four-input multiplexer shown in Figure 9-20(a). Here, four inputs are selectively transmitted to the output according to the four possible combinations of the S_1S_0 select inputs. Each data input is gated with a different combination of select input levels. I_0 is gated with $\overline{S}_1\overline{S}_0$ so that I_0 will pass through its AND gate to output Z only when $S_1 = 0$ and $S_0 = 0$. The table in the figure gives the outputs for the other three input-select codes.

Another circuit that performs exactly the same function is shown in Figure 9-20(b). This approach uses tristate buffers to select one of the signals. The decoder ensures that only one buffer can be enabled at any time. S_1 and S_0 are used to specify which of the input signals is allowed to pass through its buffer and arrive at the output.

Two-, four-, eight-, and 16-input multiplexers are readily available in the TTL and CMOS logic families. These basic ICs can be combined for multiplexing a larger number of inputs.

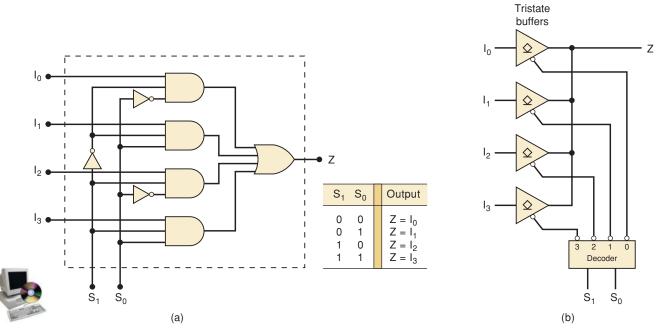
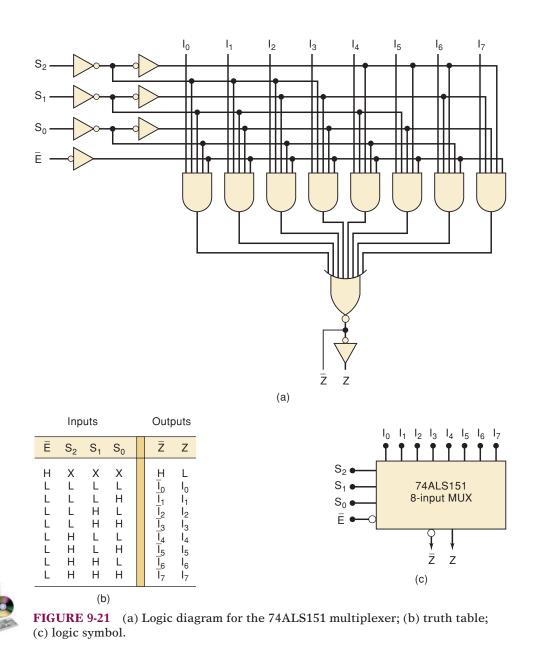


FIGURE 9-20 Four-input multiplexer: (a) using sum of products logic; (b) using tristate buffers.

Eight-Input Multiplexer

Figure 9-21(a) shows the logic diagram for the 74ALS151 (74HC151) eightinput multiplexer. This multiplexer has an enable input \overline{E} and provides both the normal and the inverted outputs. When $\overline{E} = 0$, the select inputs $S_2S_1S_0$ will select one data input (from I_0 through I_7) for passage to output Z. When $\overline{E} = 1$, the multiplexer is disabled so that Z = 0 regardless of the select input code. This operation is summarized in Figure 9-21(b), and the 74151 logic symbol is shown in Figure 9-21(c).



EXAMPLE 9-9

The circuit in Figure 9-22 uses two 74HC151s, an INVERTER, and an OR gate. Describe this circuit's operation.

Solution

This circuit has a total of 16 data inputs, eight applied to each multiplexer. The two multiplexer outputs are combined in the OR gate to produce a single output *X*. The circuit functions as a 16-input multiplexer. The four select inputs $S_3S_2S_1S_0$ will select one of the 16 inputs to pass through to *X*.

The S_3 input determines which multiplexer is enabled. When $S_3 = 0$, the top multiplexer is enabled, and the $S_2S_1S_0$ inputs determine which of its data inputs will appear at its output and pass through the OR gate to X. When $S_3 = 1$, the bottom multiplexer is enabled, and the $S_2S_1S_0$ inputs select one of its data inputs for passage to output X.

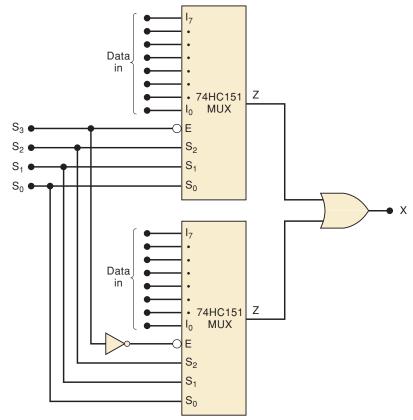




FIGURE 9-22 Example 9-9: two 74HC151s combined to form a 16-input multiplexer.

Quad Two-Input MUX (74ALS157/HC157)

The 74ALS157 is a very useful multiplexer IC that contains four two-input multiplexers like the one in Figure 9-19. The logic diagram for the 74ALS157 is shown in Figure 9-23(a). Note the manner in which the data inputs and outputs are labeled.

EXAMPLE 9-10

Determine the input conditions required for each Z output to take on the logic level of its corresponding I_0 input. Repeat for I_1 .

Solution

First of all, the enable input must be active; that is, $\overline{E} = 0$. In order for Z_a to equal I_{0a} , the select input must be LOW. These same conditions will produce $Z_b = I_{0b}, Z_c = I_{0c}$, and $Z_d = I_{0d}$.

With $\overline{E} = 0$ and S = 1, the Z outputs will follow the set of I_1 inputs; that is, $Z_a = I_{1a}$, $Z_b = I_{1b}$, $Z_c = I_{1c}$, and $Z_d = I_{1d}$.

All of the outputs will be disabled (LOW) when $\overline{E} = 1$.

It is helpful to think of this multiplexer as being a simple two-input multiplexer, but one in which each input is four lines and the output is four lines. The four output lines switch back and forth between the two sets of four input lines under the control of the select input. This operation is represented by the 74ALS157's logic symbol in Figure 9-23(b).

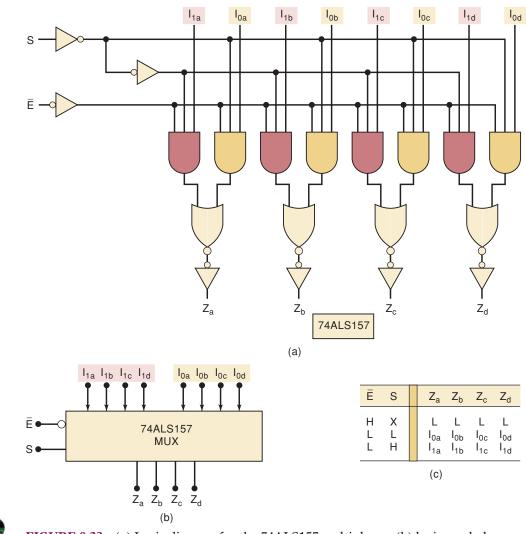


FIGURE 9-23 (a) Logic diagram for the 74ALS157 multiplexer; (b) logic symbol; (c) truth table.

REVIEW QUESTIONS

- 1. What is the function of a multiplexer's select inputs?
- 2. A certain multiplexer can switch one of 32 data inputs to its output. How many different inputs does this MUX have?

9-7 MULTIPLEXER APPLICATIONS

Multiplexer circuits find numerous and varied applications in digital systems of all types. These applications include data selection, data routing, operation sequencing, parallel-to-serial conversion, waveform generation, and logic-function generation. We shall look at some of these applications here and several more in the problems at the end of the chapter.

Data Routing

Multiplexers can route data from one of several sources to one destination. One typical application uses 74ALS157 multiplexers to select and display the

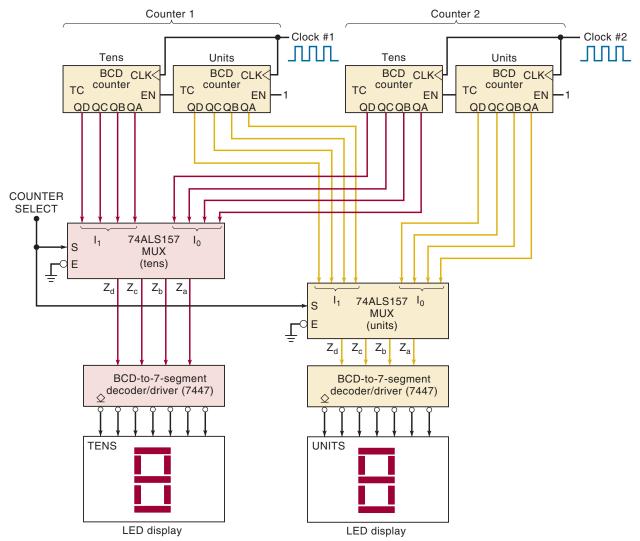


FIGURE 9-24 System for displaying two multidigit BCD counters one at a time.

contents of either of two BCD counters using a *single* set of decoder/drivers and LED displays. The circuit arrangement is shown in Figure 9-24.

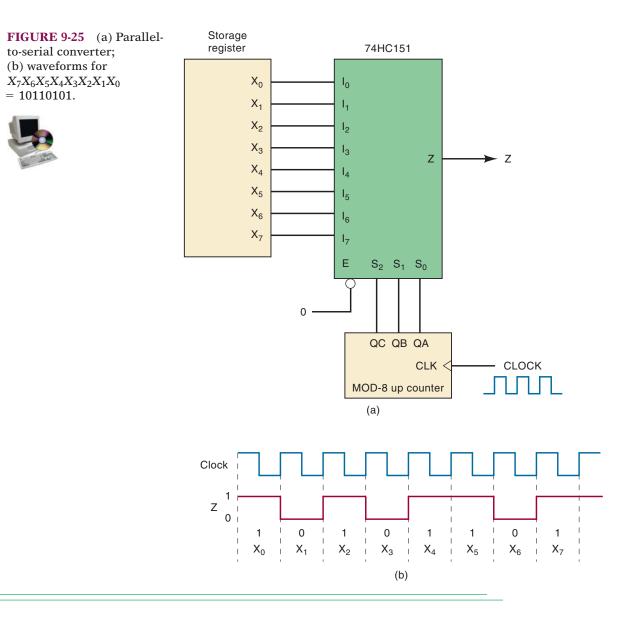
Each counter consists of two cascaded BCD stages, and each one is driven by its own clock signal. When the COUNTER SELECT line is HIGH, the outputs of counter 1 will be allowed to pass through the multiplexers to the decoder/drivers to be displayed on the LED readouts. When COUNTER SELECT = 0, the outputs of counter 2 will pass through the multiplexers to the displayes. In this way, the decimal contents of one counter or the other will be displayed under the control of the COUNTER SELECT input. A common situation where this might be used is in a digital watch. The digital watch circuitry contains many counters and registers that keep track of seconds, minutes, hours, days, months, alarm settings, and so on. A multiplexing scheme such as this one allows different data to be displayed on the limited number of decimal readouts.

The purpose of the multiplexing technique, as it is used here, is to *time-share* the decoder/drivers and display circuits between the two counters rather than have a separate set of decoder/drivers and displays for each counter. This results in a significant saving in the number of wiring connections, especially when more BCD stages are added to each counter. Even more important, it represents a significant decrease in power consumption because

decoder/drivers and LED readouts typically draw relatively large amounts of current from the V_{CC} supply. Of course, this technique has the limitation that only one counter's contents can be displayed at a time. However, in many applications, this limitation is not a drawback. A mechanical switching arrangement could have been used to perform the function of switching first one counter and then the other to the decoder/drivers and displays, but the number of required switch contacts, the complexity of wiring, and the physical size could all be disadvantages over the completely logic method of Figure 9-24.

Parallel-to-Serial Conversion

Many digital systems process binary data in parallel form (all bits simultaneously) because it is faster. When data are to be transmitted over relatively long distances, however, the parallel arrangement is undesirable because it requires a large number of transmission lines. For this reason, binary data or information in parallel form is often converted to serial form before being transmitted to a remote destination. One method for performing this **parallel-to-serial conversion** uses a multiplexer, as illustrated in Figure 9-25.



The data are present in parallel form at the outputs of the X register and are fed to the eight-input multiplexer. A three-bit (MOD-8) counter is used to provide the select code bits $S_2S_1S_0$ so that they cycle through from 000 to 111 as clock pulses are applied. In this way, the output of the multiplexer will be X_0 during the first clock period, X_1 during the second clock period, and so on. The output Z is a waveform that is a serial representation of the parallel input data. The waveforms in the figure are for the case where $X_7X_6X_5X_4X_3X_2X_1X_0 = 10110101$. This conversion process takes a total of eight clock cycles. Note that X_0 (the LSB) is transmitted first and the X_7 (MSB) is transmitted last.

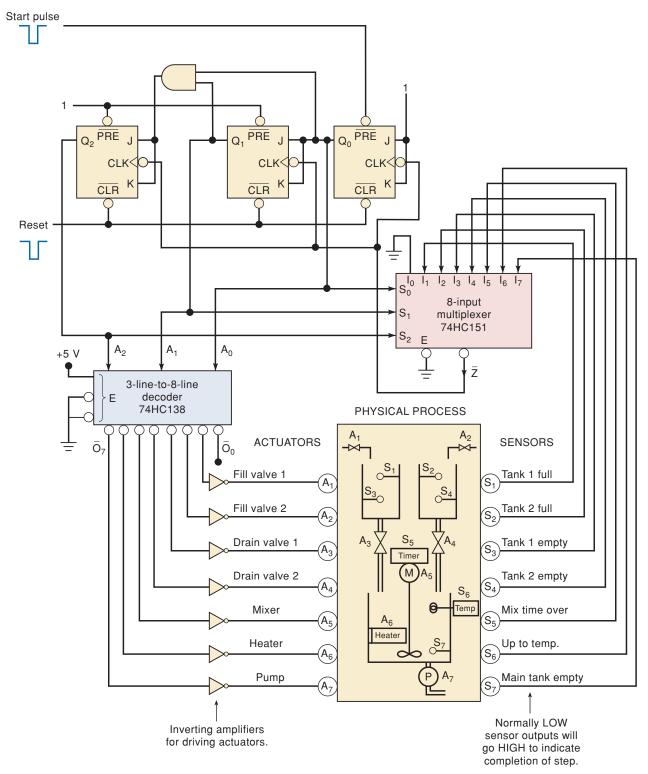
Operation Sequencing

The circuit of Figure 9-26 uses an eight-input multiplexer as part of a control sequencer that steps through seven steps, each of which actuates some portion of the physical process being controlled. This could be, for example, a process that mixes two liquid ingredients and then cooks the mixture. The circuit also uses a 3-line-to-8-line decoder and a MOD-8 binary counter. The operation is described as follows.

- 1. Initially the counter is reset to the 000 state. The counter outputs are fed to the select inputs of the multiplexer and to the inputs of the decoder. Thus, the decoder output $\overline{O}_0 = 0$ and the others are all 1, so that all the ACTUATOR inputs of the process are LOW. The SENSOR outputs of the process all start out LOW. The multiplexer output $\overline{Z} = \overline{I}_0 = 1$ because the *S* inputs are 000.
- 2. The START pulse initiates the sequencing operation by setting flip-flop Q_0 HIGH, bringing the counter to the 001 state. This causes decoder output \overline{O}_1 to go LOW, thereby activating actuator 1, which is the first step in the process (opening fill valve 1).
- 3. Some time later, SENSOR output 1 goes HIGH, indicating the completion of the first step (the float switch indicates that the tank is full). This HIGH is now present at the I_1 input of the multiplexer. It is inverted and reaches the \overline{Z} output because the select code from the counter is 001.
- 4. The LOW transition at \overline{Z} is fed to the *CLK* of flip-flop Q_0 . This negative transition advances the counter to the 010 state.
- 5. Decoder output \overline{O}_2 now goes LOW, activating actuator 2, which is the second step in the process (opening fill valve 2). \overline{Z} now equals \overline{I}_2 (the select code is 010). Because SENSOR output 2 is still LOW, \overline{Z} will go HIGH.
- 6. When the second process step is complete, SENSOR output 2 goes HIGH, producing a LOW at \overline{Z} and advancing the counter to 011.
- 7. This same action is repeated for each of the other steps. When the seventh step is completed, SENSOR output 7 goes HIGH, causing the counter to go from 111 to 000, where it will remain until another START pulse reinitiates the sequence.

Logic Function Generation

Multiplexers can be used to implement logic functions directly from a truth table without the need for simplification. When a multiplexer is used for this purpose, the select inputs are used as the logic variables, and each data input is connected permanently HIGH or LOW as necessary to satisfy the truth table.



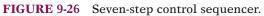
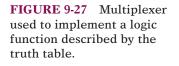
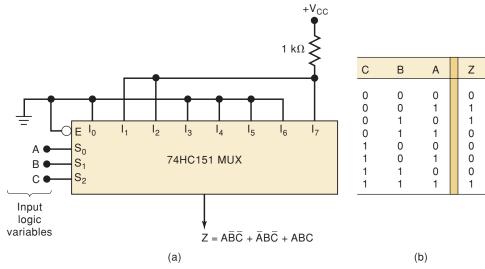


Figure 9-27 illustrates how an eight-input multiplexer can be used to implement the logic circuit that satisfies the given truth table. The input variables A, B, C are connected to S_0, S_1, S_2 , respectively, so that the levels on these inputs determine which data input appears at output Z. According to the truth table, Z is supposed to be LOW when CBA = 000. Thus, multiplexer input I_0 should be







connected LOW. Likewise, *Z* is supposed to be LOW for CBA = 011, 100, 101, and 110, so that inputs I_3 , I_4 , I_5 , and I_6 should also be connected LOW. The other sets of *CBA* conditions must produce Z = 1, and so multiplexer inputs I_1 , I_2 , and I_7 are connected permanently HIGH.

It is easy to see that any three-variable truth table can be implemented with this eight-input multiplexer. This method of implementation is often more efficient than using separate logic gates. For example, if we can write the sum-of-products expression for the truth table in Figure 9-27, we have

$$Z = A\overline{B}\overline{C} + \overline{A}B\overline{C} + ABC$$

This *cannot* be simplified either algebraically or by K mapping, and so its gate implementation would require three INVERTERs and four NAND gates, for a total of three ICs.

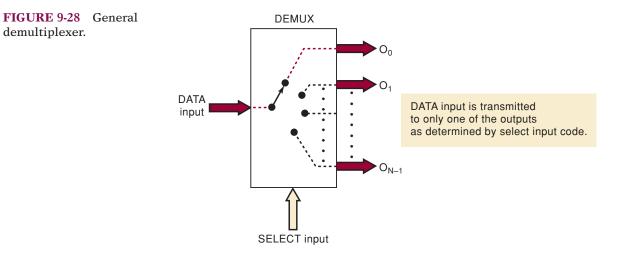
There is an even more efficient method for using multiplexers to implement logic functions. This method will allow the logic designer to use a multiplexer with three select inputs (e.g., a 74HC151) to implement a *four-variable* logic function. We will present this method in Problem 9-37.

The most important concept to be gained from using a MUX to implement a sum-of-products expression is the fact that the logic function can be very easily changed by simply changing the 1s and 0s on the MUX inputs. In other words, a MUX can very easily be used as a programmable logic device (PLD). Many PLDs use this strategy in hardware blocks that are generally referred to as look-up tables (LUTs). We will discuss look-up tables in more detail in Chapters 12 and 13.

REVIEW QUESTIONS	 What are some of the major applications of multiplexers? <i>True or false:</i> When a multiplexer is used to implement a logic function, the logic variables are applied to the multiplexer's data inputs.
	3. What type of circuit provides the select inputs when a MUX is used as a parallel-to-serial converter?

9-8 DEMULTIPLEXERS (DATA DISTRIBUTORS)

A multiplexer takes several inputs and transmits *one* of them to the output. A **demultiplexer (DEMUX)** performs the reverse operation: it takes a single input and distributes it over several outputs. Figure 9-28 shows the functional diagram for a digital demultiplexer. The large arrows for inputs and outputs can represent one or more lines. The select input code determines to which output the DATA input will be transmitted. In other words, the demultiplexer takes one input data source and selectively distributes it to 1 of N output channels just like a multiposition switch.

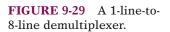


1-Line-to-8-Line Demultiplexer

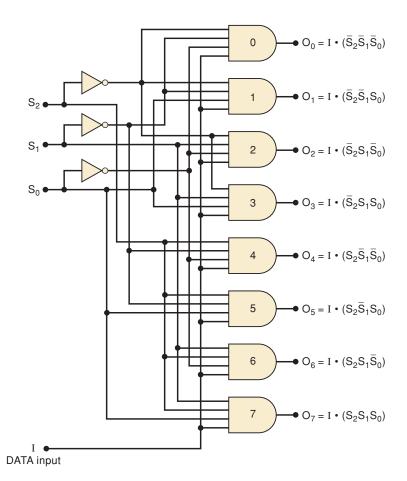
Figure 9-29 shows the logic diagram for a demultiplexer that distributes one input line to eight output lines. The single data input line *I* is connected to all eight AND gates, but only one of these gates will be enabled by the SELECT input lines. For example, with $S_2S_1S_0 = 000$, only AND gate 0 will be enabled, and data input *I* will appear at output O_0 . Other SELECT codes cause input *I* to reach the other outputs. The truth table summarizes the operation.

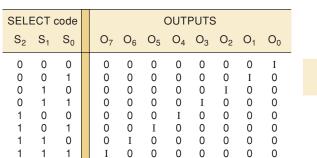
The demultiplexer circuit of Figure 9-29 is very similar to the 3-line-to-8line decoder circuit in Figure 9-2 except that a fourth input (I) has been added to each gate. It was pointed out earlier that many IC decoders have an ENABLE input, which is an extra input added to the decoder gates. This type of decoder chip can therefore be used as a demultiplexer, with the binary code inputs (e.g., A, B, C in Figure 9-2) serving as the SELECT inputs and the ENABLE input serving as the data input I. For this reason, IC manufacturers often call this type of device a *decoder/demultiplexer*, and it can be used for either function.

We saw earlier how the 74ALS138 is used as a 1-of-8 decoder. Figure 9-30 shows how it can be used as a demultiplexer. The enable input \overline{E}_1 is used as the data input I, while the other two enable inputs are held in their active states. The $A_2A_1A_0$ inputs are used as the select code. To illustrate the operation, let's assume that the select inputs are 000. With this input code, the only output that can be activated is \overline{O}_0 , while all other outputs are HIGH. \overline{O}_0 will go LOW only if \overline{E}_1 goes LOW and will be HIGH if \overline{E}_1 goes HIGH. In other words, \overline{O}_0 will follow the signal on \overline{E}_1 (i.e., the data input, I) while all other outputs stay HIGH. In a similar manner, a different select code applied to $A_2A_1A_0$ will cause the corresponding output to follow the data input, I.

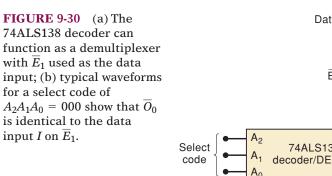








Note: I is the data input



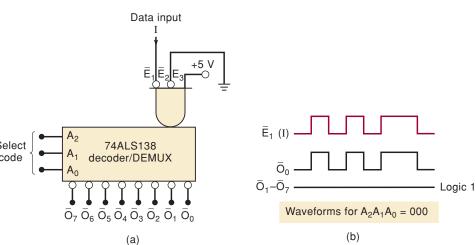


Figure 9-30(b) shows typical waveforms for the case where $A_2A_1A_0 = 000$ selects output \overline{O}_0 . For this case, the data signal applied to \overline{E}_1 will be transmitted to \overline{O}_0 , and all other outputs will remain in their inactive HIGH state.

Security Monitoring System

Consider the case of a security monitoring system in an industrial plant where the open/closed status of many access doors is to be monitored. Each door controls the state of a switch, and it is necessary to display the state of each switch on LEDs that are mounted on a remote monitoring panel at the security guard's station. One way to do this would be to run a separate signal from each door switch to an LED on the monitoring panel. This setup would require running many wires over a long distance. A better approach that would reduce the amount of wiring to the monitoring panel uses a multiplexer/demultiplexer combination. Figure 9-31 shows a system that can handle eight doors, but the basic idea can be expanded to any number.

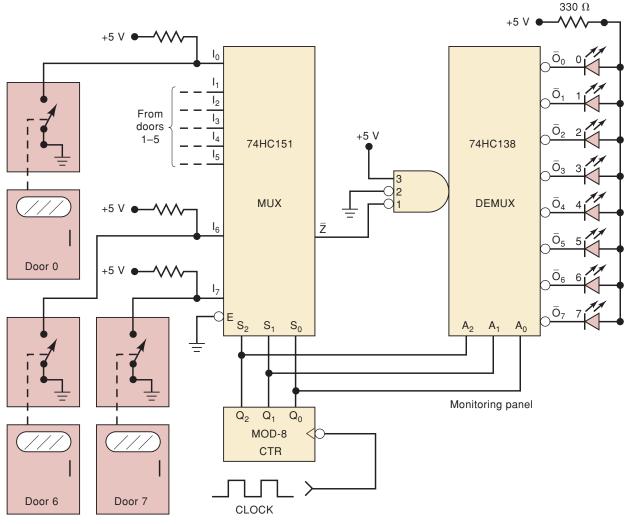


FIGURE 9-31 Security monitoring system.

Examine Figure 9-31 carefully and describe the complete operation.

Solution

The eight door switches are the data inputs to the MUX; they produce a HIGH when a door is open and a LOW when it is closed. The MOD-8 counter provides the select inputs to the MUX and also to the DEMUX on the remote monitoring panel. Each DEMUX output is connected to an indicator LED that will be on when the output is LOW. Clock pulses applied to the counter will cause the select inputs to sequence through all of the possible states 000 through 111. At each number of the counter, the switch status for the door of the same number will be inverted by the MUX and passed to output \overline{Z} . From there, it is transmitted to the DEMUX input, which passes it through to the output corresponding to the same number.

For example, let's say that the counter is at the count of 110 (6). While the counter is in this state, let's say that door 6 is closed. The LOW at I_6 will pass through the MUX and be inverted to produce a HIGH at \overline{Z} . This HIGH will be passed through the DEMUX to output \overline{O}_6 so that LED 6 will be off, indicating that door 6 is closed. Now let's say that door 6 is open. A LOW will appear at \overline{Z} and \overline{O}_6 so that LED 6 will be on to signal that door 6 is open. Of course, all other LEDs will be off during this time because \overline{O}_6 is the only active output.

As the counter is clocked through its eight states 000 through 111, the LEDs will sequentially indicate the status of the eight doors. If all the doors are closed, none of the LEDs will be on even when the corresponding DEMUX output is selected. If a door is open, its LED will turn on only during the time interval that the counter is at the appropriate count; it will be off at all other counts. Thus, the LED will be flashing on and off if its door is open. The flashing rate can be adjusted by changing the frequency of the clock.

Note that there are only four signal lines going from the "door-sensing" circuitry to the remote monitoring panel: the \overline{Z} output and the three select lines. This is a saving of four lines when compared with the alternative of having one line per door. The MUX/DEMUX combination is used to transmit the status of each door to its LED one at a time (serially) instead of all at once (parallel).

Synchronous Data Transmission System

Figures 9-32 and 9-33 show the logic diagrams for a synchronous data transmission system that is used to transmit four, four-bit words serially from a transmitter to a remote receiver. To operate this system, four data words are parallel-loaded into the input registers of the transmitter block and the transmit signal is activated. The 16 data bits are then sent over a single data line, one bit at a time, reassembled by the receiver, and stored in output registers. Let's look at the transmitter details in Figure 9-32 first. The *clock* input is a high-frequency, constantly running, periodic clock signal that synchronizes all activities in the system. The four-bit data words are stored individually (synchronously) in the PISO registers when enabled by the appropriate ld_x input. For simplicity, the parallel data inputs to the PISO registers are not shown in the diagram. These input registers are designed to shift the data to the right and also recirculate the LSB (rightmost bit) to the MSB

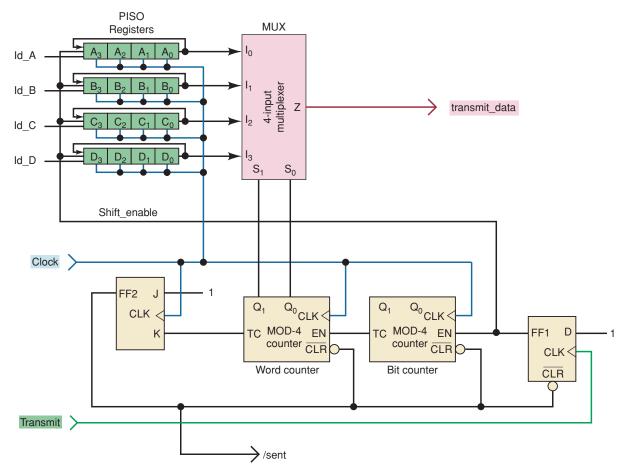


FIGURE 9-32 Transmitter block in synchronous data transmission system.

(leftmost bit). With this arrangement the bits are all shifted to the serial output and also end up back in their proper locations after four clock pulses.

TRANSMITTER OPERATION Initially, let's assume that all the flip-flops and the two MOD-4 counters in Figure 9-32 are all cleared. On the next PGT of clock, FF2 is SET, removing the asynchronous clear command from the counters and FF1. When the transmit signal goes HIGH, FF1 is SET, putting all the shift registers in the shift mode. The MUX selects input 0 (register A) because the MOD-4 Word counter is at 0. At this point the LSB of register A is on the *transmit data* line. The next three clock pulses (counted by the Bit counter) shift the other bits of register A to the serial output. As a result, the *transmit_data* line outputs each of the register A bits, one at a time from the least to the most significant. On the fourth PGT, the Bit counter rolls over to zero, the Word counter increments to 1, all of the shift registers have recirculated their data back to the original position, and the MUX now selects the LSB data from register B to output on the *transmit_data* line. The next three clocks shift out the contents of register B, followed by registers C and D. On the 16th PGT, FF2 toggles to a zero state, resetting all the counters and disabling any further counting by also clearing FF1. The next PGT sets FF2 again, and the system is waiting for new data to be loaded and the next *transmit* signal.

RECEIVER OPERATION The receiver circuit shown in Figure 9-33 is very similar in operation to the transmitter. Notice that all flip-flops, counters,

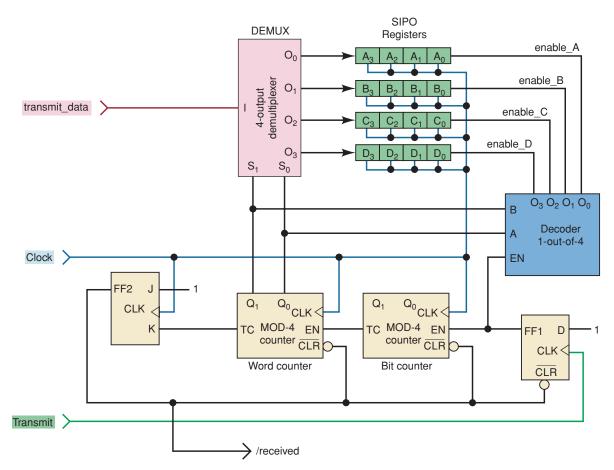
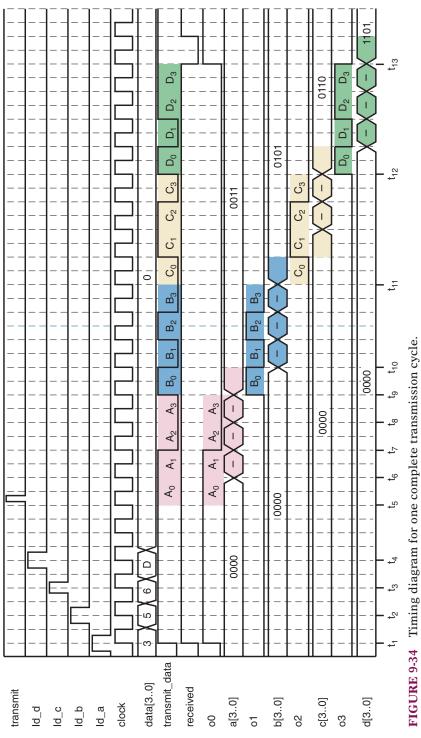


FIGURE 9-33 Receiver block in synchronous data transmission system.

and registers use the same clock as the transmitter. The receiver uses a DE-MUX to distribute the serial data to the appropriate SIPO register and a decoder to enable one register at a time. Let's begin analyzing this circuit with all counters and flip-flops at zero. The next *clock* sets *FF2*, removing the asynchronous clear command from the counters and FF1. When the transmit line goes HIGH, FF1 is SET, enabling the Bit counter, Word counter, and also the decoder. With the Word counter at zero, the decoder enables register A and the DEMUX connects the serial data line (which currently contains the LSB of transmit register A) to the serial data input of receive register A. The next PGT shifts the least significant data bit into register A and advances the Bit counter. The next three PGTs shift the next three data bits into register A, the Bit counter rolls over to zero, the Word counter increments to 1, and the decoder and DEMUX switch to register B. After the 16th PGT, all four registers contain the proper data, FF2 has toggled to a zero state, FF1 is cleared and disables the decoder, which disables all the SIPO registers. On the next PGT, *FF2* is set and the system is waiting for the next transmission of data.

SYSTEM TIMING The timing diagram in Figure 9-34 shows the parallel data that is loaded into the transmitter, the serial data stream, and the distribution and storage of the four data values in the receiver registers. At times t_{1-4} , the binary data values (shown as hex 3, 5, 6, and D) are loaded into transmit registers *A*, *B*, *C*, and *D*, respectively. The system is idle until the *transmit* line goes HIGH at t_5 . At this point the LSB from register *A* (A_0) is already on the *transmit_data* line. Also notice that at t_5-t_8 , the data on output



 O_0 of the DEMUX is identical to the *transmit_data* line. This shows that the DEMUX has distributed the *transmit_data* to shift register A. At t_6 , the PGT of the clock shifts A_0 into the MSB of receive register A, all transmit data registers (not shown in the timing) are shifted, and data bit A_1 appears on the *transmit_data* line. At times t_7 , t_8 , and t_9 , the other three bits are shifted into register A such that after t_9 , receive register A contains the data bits that were stored in transmit register A. The diagram shows that the DEMUX has switched to distribute data to register B because the DEMUX output O_1 is now identical to *transmit_data* from t_9 through t_{11} . Starting at t_{10} , the data are shifted into receive register B. Register C and Register D are sent and stored from t_{11} to t_{12} and from t_{12} to t_{13} , respectively.

REVIEW QUESTIONS

- 1. Explain the difference between a DEMUX and a MUX.
- 2. *True or false:* The circuit for a DEMUX is basically the same as for a decoder.
- 3. For the system of Figure 9-31, what will the security guard see on the monitoring panel when all of the doors are open?

9-9 MORE TROUBLESHOOTING

Here are three more examples to illustrate the observation/reasoning process that is such an important initial step when troubleshooting. For each case, try to determine the circuit fault before looking at the solution.

EXAMPLE 9-12

Consider the circuit of Figure 9-24. A test performed on this circuit yields the result shown in Table 9-3. What is the probable circuit fault?

TABLE 9-3

		Actual Count	Displayed Count
Case 1	Counter 1	25	25
	Counter 2	37	35
Case 2	Counter 1	49	49
	Counter 2	72	79
Case 3	Counter 1	96	96
	Counter 2	14	16

Solution

In each of the test cases, the display of counter 1 matches the counter's actual count. This indicates that the I_1 inputs, all MUX outputs, and both displays are probably working correctly. On the other hand, each test case shows that counter 2's *tens* digit is displayed correctly but its *units* digit is displayed incorrectly. This could mean that there is a fault somewhere between the

output of the units section of counter 2 and the I_0 inputs of the units MUX. We should compare the bit patterns of the actual and displayed values of the units for counter 2 (Table 9-4). The idea is to look for things such as a bit that does not change (stuck LOW or HIGH) or two bits that are reversed (crossed connections). The data in Table 9-4 reveal no obvious pattern.

TABLE 9-4		Actual Units	Displayed Units
	Case 1	0111 (7)	0101 (5)
	Case 2	0010 (2)	1001 (9)
	Case 3	0100 (4)	0110 (6)

If we take another look at the recorded test results, we see that the displayed units digit of counter 2 is always the same as the units digit of counter 1. This symptom is probably the result of a constant logic HIGH at the select input of the units MUX because that would continually pass the units digit of counter 1 to the units MUX output. This constant HIGH at the select input is most likely caused by an open path somewhere between the select input of the tens MUX and the select input of the units MUX. It could not be caused by a short to V_{CC} because that would also keep the select input of the tens MUX at a constant HIGH, and we know that the tens MUX is working.

EXAMPLE 9-13

The security monitoring system of Figure 9-31 is tested and the results are recorded in Table 9-5. What are the possible faults that could produce these results?

TABLE 9-5

Condition	LEDs
All doors closed	All LEDs off
Door 0 open	LED 4 flashing
Door 1 open	LED 5 flashing
Door 2 open	LED 6 flashing
Door 3 open	LED 7 flashing
Door 4 open	LED 4 flashing
Door 5 open	LED 5 flashing
Door 6 open	LED 6 flashing
Door 7 open	LED 7 flashing

Solution

Again, the data should be reviewed to see if there is some pattern that could help to narrow down the search for the fault to a small area of the circuit. The data in Table 9-5 reveal that the correct LEDs flash for open doors 4 through 7. They also show that for open doors 0 through 3, the number of the flashing LED is *four* more than the number of the door, and LEDs 0 through 3 are always off. This is most probably caused by a constant logic HIGH at A_2 , the MSB of the select input of the DEMUX, because this would always

make the select code 4 or greater, and it would add 4 to the select codes 0 through 3.

Thus, we have two possibilities: A_2 is somehow shorted to V_{CC} , or there is an open connection at A_2 . A little thought will eliminate the first choice as a possibility because this would also mean that S_2 of the MUX would also be stuck HIGH. If that were so, then the status of doors 0 through 3 would not get through the MUX and into the DEMUX. We know that this is not true because the data show that when any of these doors is open, it affects one of the DEMUX outputs.

EXAMPLE 9-14

An extremely important principle of troubleshooting, called *divide-and-conquer*, was introduced in Section 9-5. It is really not about military strategy, but rather describes the most efficient way to eliminate from consideration all the parts of the circuit that are working correctly. Assume that data have been loaded into the four transmit registers of Figure 9-32 and the transmit pulse has occurred, but after the next 16 clock pulses, no new data have appeared in the receive registers shown in Figure 9-33. How can we most efficiently find the problem?

Solution

In a synchronous digital system that is simply not functioning, it is reasonable first to check to see if the power supply and clock are working, just as you might check for a pulse if you found a person lying on the ground. However, assuming the clock is oscillating, there is a much more efficient way to isolate the problem than randomly picking points in the circuit and determining if the correct signal is present. We want to perform a test on this circuit such that, if we obtain the desired results, we know that half of the circuit is working correctly and we can eliminate that half from consideration. In this circuit the best place to look is at the *transmit_data* line. A logic probe should be placed on the *transmit_data* line and the *transmit* signal should be activated. If a burst of pulses is observed on the logic probe, it means that the transmit section is functioning. We may not know if the data are correct, but remember, the receiver is not getting incorrect data but rather no data at all. However, if no burst of pulses is observed, there is certainly a problem in the transmit section.

A troubleshooting tree diagram as shown in Figure 9-35 is helpful in isolating problems in a system. Let's assume there were no pulses on *transmit_data*. Now we need to perform a test on the transmitter to prove that half of the transmitter is working properly. In this case the circuit does not divide exactly in half easily. A good choice might be to examine the output of the word counter. A logic probe should be placed on the select inputs of the MUX and the *transmit* signal activated. If brief pulses occur immediately after transmit, then the entire control section (made up of two counters and two flip-flops) is probably functioning properly and we can look elsewhere. The next place to look is at the outputs of the PISO registers (or data inputs of the MUX). If data pulses are present on each line after *transmit* is activated, the problem must be in the MUX. If not, we can further break down the PISO section. Each test that is performed should eliminate the largest possible amount of the remaining circuitry until all that is left is a small block containing the fault.

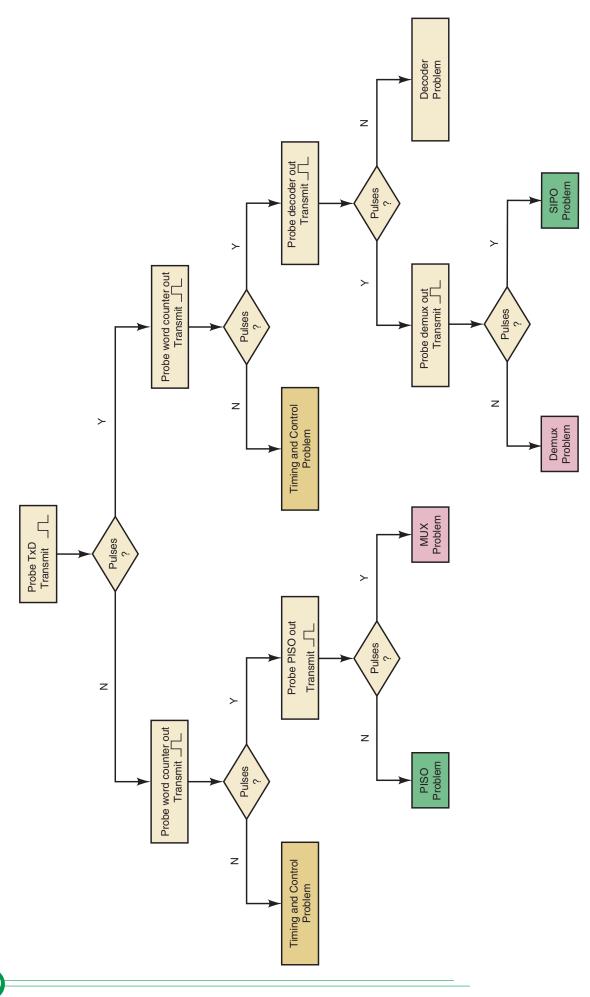
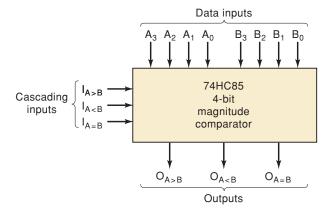


FIGURE 9-35 Example 9-14: A troubleshooting tree diagram.

9-10 MAGNITUDE COMPARATOR

Another useful member of the MSI category of ICs is the **magnitude comparator**. It is a combinational logic circuit that compares two input binary quantities and generates outputs to indicate which one has the greater magnitude. Figure 9-36 shows the logic symbol and the truth table for the 74HC85 four-bit magnitude comparator, which is also available as the 74LS85.





COMPARING INPUTS	CASCADING INPUTS	OUTPUTS
A ₃ , B ₃ A ₂ , B ₂ A ₁ , B ₁ A ₀ , B ₀	I _{A>B} I _{A<b< sub=""> I_{A=B}</b<>}	O _{A>B} O _{A<b< sub=""> O_{A=B}</b<>}
$ \begin{array}{c ccccccccccccccccccccccccccccccccccc$	X X X X X X X X H L L L H L L H L L H L L H L H H L H H L	H L L L H L L H L L H L H L L H L L H L L L H L L H L L H L L L H H L L L L H

TRUTH TABLE

H = HIGH Voltage Level L = LOW Voltage Level

X = Immaterial

FIGURE 9-36 Logic symbol and truth table for a 74HC85 (7485, 74LS85) four-bit magnitude comparator.

Data Inputs

The 74HC85 compares two *unsigned* four-bit binary numbers. One of them is $A_3A_2A_1A_0$, which is called word *A*; the other is $B_3B_2B_1B_0$, which is called word *B*. The term *word* is used in the digital computer field to designate a group of bits that represents some specific type of information. Here, word *A* and word *B* represent numerical quantities.

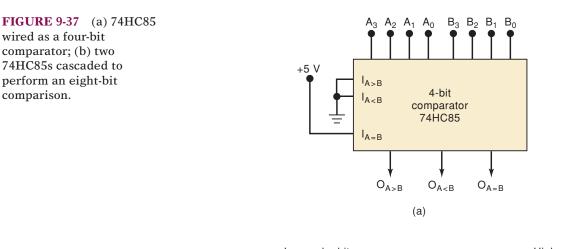
Outputs

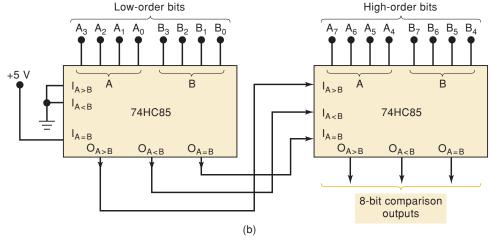
The 74HC85 has three active-HIGH outputs. Output $O_{A>B}$ will be HIGH when the magnitude of word A is greater than the magnitude of word B. Output $O_{A<B}$ will be HIGH when the magnitude of word A is less than the magnitude of word B. Output $O_{A=B}$ will be HIGH when word A and word B are identical.

Cascading Inputs

Cascading inputs provide a means for expanding the comparison operation to more than four bits by cascading two or more four-bit comparators. Note that the cascading inputs are labeled the same as the outputs. When a four-bit comparison is being made, as in Figure 9-37(a), the cascading inputs should be connected as shown in order for the comparator to produce the correct outputs.

When two comparators are to be cascaded, the outputs of the lower-order comparator are connected to the corresponding inputs of the higher-order comparator. This is shown in Figure 9-37(b), where the comparator on the left is comparing the lower-order four bits of the two eight-bit words: $A_7A_6A_5A_4A_3A_2A_1A_0$ and $B_7B_6B_5B_4B_3B_2B_1B_0$. Its outputs are fed to the cascade inputs of the comparator on the right, which is comparing the high-order bits. The outputs of the high-order comparator are the final outputs that indicate the result of the eight-bit comparison.





Describe the operation of the eight-bit comparison arrangement in Figure 9-37(b) for the following cases:

(a) $A_7A_6A_5A_4A_3A_2A_1A_0 = 10101111; B_7B_6B_5B_4B_3B_2B_1B_0 = 10110001$ (b) $A_7A_6A_5A_4A_3A_2A_1A_0 = 10101111; B_7B_6B_5B_4B_3B_2B_1B_0 = 10101001$

Solution

- (a) The high-order comparator compares its inputs $A_7A_6A_5A_4 = 1010$ and $B_7B_6B_5B_4 = 1011$ and produces $O_{A < B} = 1$ regardless of what levels are applied to its cascade inputs from the low-order comparator. In other words, once the high-order comparator senses a difference in the high-order bits of the two eight-bit words, it knows which eight-bit word is greater without having to look at the results of the low-order comparison.
- (b) The high-order comparator sees $A_7A_6A_5A_4 = B_7B_6B_5B_4 = 1010$, so it must look at its cascade inputs to see the result of the low-order comparison. The low-order comparator has $A_3A_2A_1A_0 = 1111$ and $B_3B_2B_1B_0 = 1001$, which produces a 1 at its $O_{A>B}$ output and the $I_{A>B}$ input of the high-order comparator. The high-order comparator senses this 1, and because its data inputs are equal, it produces a HIGH at its $O_{A>B}$ to indicate the result of the eight-bit comparison.

Applications

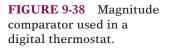
Magnitude comparators are also useful in control applications where a binary number representing the physical variable being controlled (e.g., position, speed, or temperature) is compared with a reference value. The comparator outputs are used to actuate circuitry to drive the physical variable toward the reference value. The following example will illustrate one application. We will examine another comparator application in Problem 9-52.

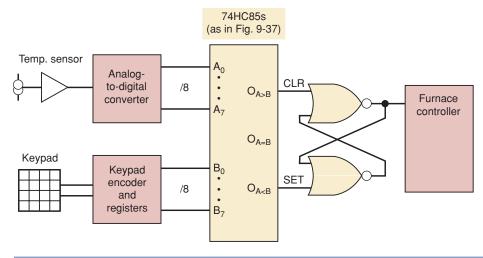
EXAMPLE 9-16

Consider a digital thermostat in which the measured room temperature is converted to a digital number and applied to the *A* inputs of a comparator. The desired room temperature, entered from a keypad, is stored in a register that is connected to the *B* inputs. If A < B, the furnace should be activated to heat the room. The furnace should continue to heat while A = B and shut off when A > B. As the room cools off, the furnace should stay off while A = B and turn on again when A < B. What digital circuit can be used to interface a magnitude comparator to a furnace to perform the thermostat control application described above?

Solution

Using the $O_{A < B}$ output to drive the furnace directly would cause it to turn off as soon as the values became equal. This can cause severe on/off cycling of the furnace when the actual temperature is very close to the boundary between A < B and A = B. By using a NOR gate SET-CLEAR latch circuit (refer to Chapter 5) as shown in Figure 9-38, the system will operate as described. Notice that $O_{A < B}$ is connected to the SET input and $O_{A > B}$ is connected to the CLEAR input of the latch. When the temperature is hotter than desired, it clears the latch, shutting off the furnace. When the temperature is cooler than desired, it sets the latch, turning the furnace on.





REVIEW QUESTIONS

- 1. What is the purpose of the cascading inputs of the 74HC85?
- 2. What are the outputs of a 74HC85 with the following inputs: $A_3A_2A_1A_0 = B_3B_2B_1B_0 = 1001$, $I_{A>B} = I_{A<B} = 0$, and $I_{A=B} = 1$?

9-11 CODE CONVERTERS

A code converter is a logic circuit that changes data presented in one type of binary code to another type of binary code. The BCD-to-7-segment decoderdriver that we presented earlier is a code converter because it changes a BCD input code to the 7-segment code needed by the LED display. A partial list of some of the more common code conversions is given in Table 9-6.

As an example of a code converter circuit, let's consider a BCD-to-binary converter. Before we get started on the circuit implementation, we should review the BCD representation.

Two-digit decimal values ranging from 00 to 99 can be represented in BCD by two four-bit code groups. For example, 57_{10} is represented as

5	7	
$\overrightarrow{0101}$	$\overrightarrow{0111}$	(BCD)

The straight binary representation for decimal 57 is

$$57_{10} = 111001_2$$

The largest two-digit decimal value of 99 has the following representations:

 $99_{10} = 10011001 (BCD) = 1100011_2$

Note that the binary representation requires only seven bits.

Basic Idea

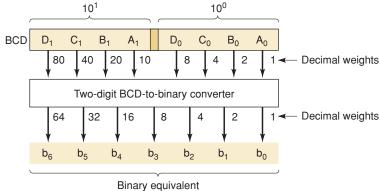
The diagram of Figure 9-39 shows the basic idea for a two-digit BCD-to-binary converter. The inputs to the converter are the two four-bit code groups

TABLE 9-6

BCD to 7-segment
BCD to binary
Binary to BCD
Binary to Gray code
Gray code to binary
ASCII to EBCDIC*
EBCDIC to ASCII

*EBCDIC is an alphanumeric code developed by IBM and is similar to ASCII.





 $D_0C_0B_0A_0$, representing the 10^0 or units digit, and $D_1C_1B_1A_1$, representing the 10^1 or tens digit of the decimal value. The outputs from the converter are $b_6b_5b_4b_3b_2b_1b_0$, the seven bits of the binary equivalent of the same decimal value. Note the difference in the weights of the BCD bits and those of the binary bits.

A typical use of a BCD-to-binary converter would be where BCD data from an instrument such as a DMM (digital multimeter) are being transferred to a computer for storage or processing. The data must be converted to binary so that they can be operated on in binary by the computer ALU, which may not have the capability of performing arithmetic operations on BCD data. The BCD-to-binary conversion can be accomplished with either hardware or software. The hardware method (which we will look at momentarily) is generally faster but requires extra circuitry. The software method uses no extra circuitry, but it takes more time because the software does the conversion step by step. The method chosen in a particular application depends on whether or not conversion time is an important consideration.

Conversion Process

The bits in a BCD representation have decimal weights that are 8, 4, 2, 1 within each code group but that differ by a factor of 10 from one code group (decimal digit) to the next. Figure 9-39 shows the bit weights for the two-digit BCD representation.

The decimal weight of each bit in the BCD representation can be converted to its binary equivalent. The results are given in Table 9-7. Using these weights, we can perform the BCD-to-binary conversion by simply doing the following:

	Decimal		Binary Equivalent					
BCD Bit	Weight	b ₆	b 5	b 4	b 3	b ₂	b 1	b 0
A ₀	1	0	0	0	0	0	0	1
B_0	2	0	0	0	0	0	1	0
C_0	4	0	0	0	0	1	0	0
D_0	8	0	0	0	1	0	0	0
<i>A</i> ₁	10	0	0	0	1	0	1	0
<i>B</i> ₁	20	0	0	1	0	1	0	0
C_1	40	0	1	0	1	0	0	0
D_1	80	1	0	1	0	0	0	0

TABLE 9-7Binaryequivalents of decimalweights of each BCD bit.

Compute the binary sum of the binary equivalents of all bits in the BCD representation that are 1s.

The following example will illustrate.

Circuit Implementation

Clearly, one way to implement the logic circuit that performs this conversion process is to use binary adder circuits. Figure 9-40 shows how two 74HC83

→0000100 (binary for 4) →0001010 (binary for 10) →+ 1010000 (binary for 80) 1011111 (binary for 95)

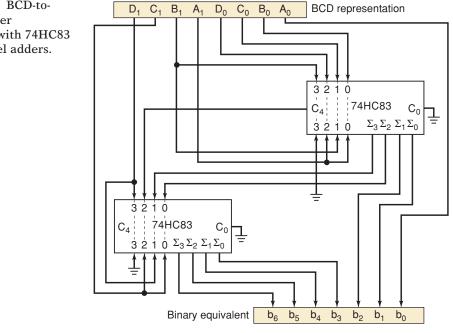


FIGURE 9-40 BCD-tobinary converter implemented with 74HC83 four-bit parallel adders. four-bit parallel adders can be wired to perform the conversion. This is one of several possible adder arrangements that will work. You may want to review the operation of this IC in Section 6-14.

The two adder ICs perform the addition of the BCD bits in the proper combinations according to Table 9-7. For instance, Table 9-7 shows that A_0 is the only BCD bit that contributes to the LSB, b_0 , of the binary equivalent. Because there is no carry into this bit position, A_0 is connected directly as output b_0 . The table also shows that only BCD bits B_0 and A_1 contribute to bit b_1 of the binary output. These two bits are combined in the upper adder to produce output b_1 . Likewise, only BCD bits D_0 , A_1 , and C_1 contribute to bit b_3 . The upper adder combines D_0 and A_1 to generate Σ_2 , which is connected to the lower adder, where C_1 is added to it to produce b_3 .

EXAMPLE 9-18

The BCD representation for decimal 56 is applied to the converter of Figure 9-40. Determine the Σ outputs from each adder and the final binary output.

Solution

Write down the bits of the BCD representation 01010110 on the circuit diagram. Because $A_0 = 0$, the b_0 bit of the output is 0.

The top inputs to the upper adder are 0011. The bottom inputs are 0101. This adder adds these to produce

 $0011 + \frac{0101}{1000} = \Sigma_3 \Sigma_2 \Sigma_1 \Sigma_0 \text{ outputs of the upper adder}$

The Σ_1 and Σ_0 bits become binary outputs b_2 and b_1 , respectively. The Σ_3 and Σ_2 bits are fed to the lower adder. The top inputs to the lower adder are therefore 0010. The bottom inputs are 0101. This adder adds these to produce

 $0010 + 0101 = \Sigma_3 \Sigma_2 \Sigma_1 \Sigma_0 \text{ outputs of the lower adder}$

These bits become $b_6b_5b_4b_3$, respectively.

Thus, we have $b_6b_5b_4b_3b_2b_1b_0 = 0111000$ as the correct binary equivalent for decimal 56.

Other Code Converter Implementations

Whereas all types of code converters can be made by combining logic gates, adder circuits, or other combinational logic, the circuitry can become quite complex, requiring many ICs. It is often more efficient to use a read-only memory (ROM) or programmable logic device (PLD) to function as a code converter. As we will see in Chapters 12 and 13, these devices contain the equivalent of hundreds of logic gates, and they can be programmed to provide a wide range of logic functions.

REVIEW QUESTIONS

- 1. What is a code converter?
- 2. How many binary outputs would a three-digit BCD-to-binary converter have?

9-12 DATA BUSING

In most modern computers, the transfer of data takes place over a common set of connecting lines called a **data bus**. In these bus-organized computers, many different devices can have their outputs and inputs tied to the common data bus lines. Because of this, the devices that are tied to the data bus will often have tristate outputs, or they will be tied to the data bus through tristate buffers.

Some of the devices that are commonly connected to a data bus are (1) microprocessors; (2) semiconductor memory chips, covered in Chapter 12; and (3) digital-to-analog converters (DACs) and analog-to-digital converters (ADCs), described in Chapter 11.

Figure 9-41 illustrates a typical situation in which a microprocessor (the CPU chip in a microcomputer) is connected to several devices over an eight-line

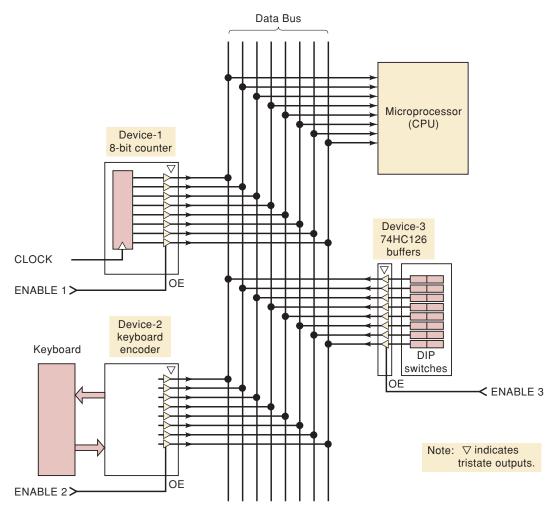


FIGURE 9-41 Three different devices can transmit eight-bit data over an eightline data bus to a microprocessor; only one device at a time is enabled so that bus contention is avoided. data bus. The data bus is simply a collection of conducting paths over which digital data are transmitted from one device to another. Each device provides an eight-bit output that is sent to the inputs of the microprocessor over the eightline data bus. Clearly, because the outputs of each of the three devices are connected to the same microprocessor inputs over the data bus conducting paths, we must be aware of bus contention problems (Section 8-12), where two or more signals tied to the same bus line are active and are essentially fighting each other. Bus contention is avoided if the devices have tristate outputs or are connected to the bus through tristate buffers (Section 8-12). The output enable inputs (*OE*) to each device (or its buffer) are used to ensure that no more than one device's outputs are active at a given time.

(a) For Figure 9-41, describe the conditions necessary to transmit data from device 3 to the microprocessor. (b) What will the status of the data bus be when none of the devices is enabled? Solution (a) ENABLE 3 must be activated; ENABLE 1 and ENABLE 2 must be in their inactive state. This will put the outputs of device 1 and device 2 in the Hi-Z state and essentially disconnect them from the bus. The outputs of device 3 will be activated so that their logic levels will appear on the data bus lines and be transmitted to the inputs of the microprocessor. We can visualize this by covering up device 1 and device 2 as if they are not even part of the circuit; then we are left with device 3 alone connected to the microprocessor over the data bus.

(b) If none of the device enable inputs are activated, all of the device outputs are in the Hi-Z state. This disconnects all device outputs from the bus. Thus, there is no definite logic level on any of the data bus lines; they are in the indeterminate state. This condition is known as a **floating bus**, and each data bus line is said to be in a *floating* (indeterminate) state. An oscilloscope display of a floating bus line would be unpredictable. A logic probe would indicate an indeterminate logic level.

REVIEW QUESTIONS

- 1. What is meant by the term *data bus*?
- 2. What is bus contention, and what must be done to prevent it?
- 3. What is a *floating bus*?

9-13 THE 74ALS173/HC173 TRISTATE REGISTER

The devices connected to a data bus will contain registers (usually flip-flops) that hold the device data. The outputs of these registers are usually connected to tristate buffers that allow them to be tied to a data bus. We will demonstrate the details of data bus operation by using an IC register that includes the tristate buffers on the same chip: the TTL 74ALS173 (also available in CMOS 74HC173 versions). Its logic diagram and truth table are shown in Figure 9-42.

		Inputs			FF Outputs
MR	CP	\overline{IE}_1	$\overline{IE_2}$	D _n	Q
H L L L L	X L I I J	X X H X L L	X X H L L	X X X L H	L Q Q Q L H

When either \overline{OE}_1 or \overline{OE}_2 is HIGH, the output is in the OFF state (high impedance); however, this does not affect the contents or sequential operating of the register.

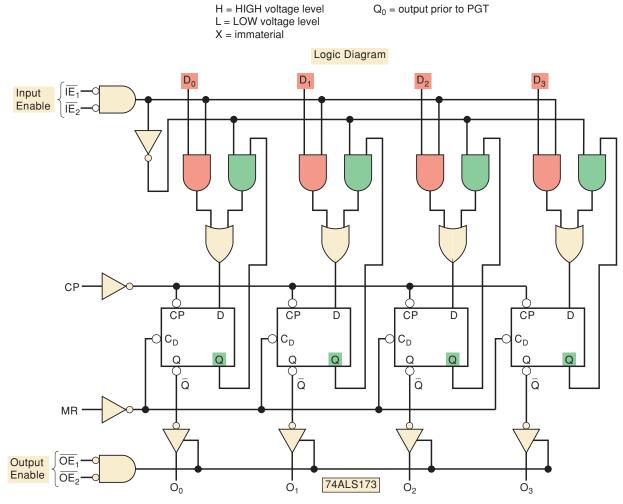


FIGURE 9-42 Truth table and logic diagram for the 74ALS173 tristate register.

The 74ALS173 is a four-bit register with parallel in/parallel out capability. Note that the FF outputs are connected to tristate buffers that provide outputs O_0 through O_3 . Also note that the data inputs D_0 through D_3 are connected to the *D* inputs of the register FFs through logic circuitry. This logic allows two modes of operation: (1) *load*, where the data at inputs D_0 to D_3 are transferred into the FFs on the PGT of the clock pulse at *CP*; and (2) *hold*, where the data in the register do not change when the PGT of *CP* occurs.



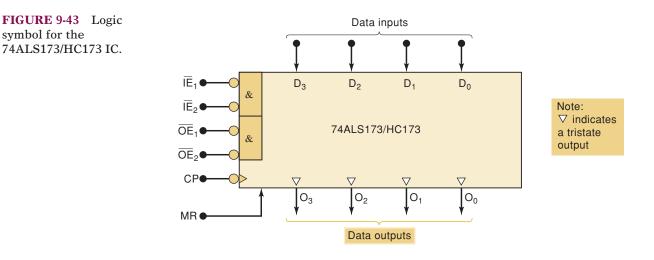
- (a) What input conditions will produce the load operation?
- (b) What input conditions will produce the hold operation?
- (c) What input conditions will allow the internal register outputs to appear at O_0 to O_3 ?

Solution

- (a) The last two entries in the truth table show that each Q output takes on the value present at its D input when a PGT occurs at CP provided that *MR* is LOW and *both* input-enable inputs, *IE*₁ and *IE*₂, are LOW.
- (b) The third and fourth lines of the truth table state that when either \overline{IE} input is HIGH, the D inputs have no effect, and the Q outputs will retain their current values when the PGT occurs.
- (c) The output buffers are enabled when both output-enable inputs, OE_1 and \overline{OE}_2 , are LOW. This will pass the register outputs through to the external outputs O_0 to O_3 . If either output-enable input is HIGH, the buffers will be disabled, and the outputs will be in the Hi-Z state.

Note that the OE inputs have no effect on the data load operation. They are used only to control whether or not the register outputs are passed to the external outputs.

The logic symbol for the 74ALS173/HC173 is given in Figure 9-43. We have included the IEEE/ANSI "&" notation to indicate the AND relationship of the two pairs of enable inputs.



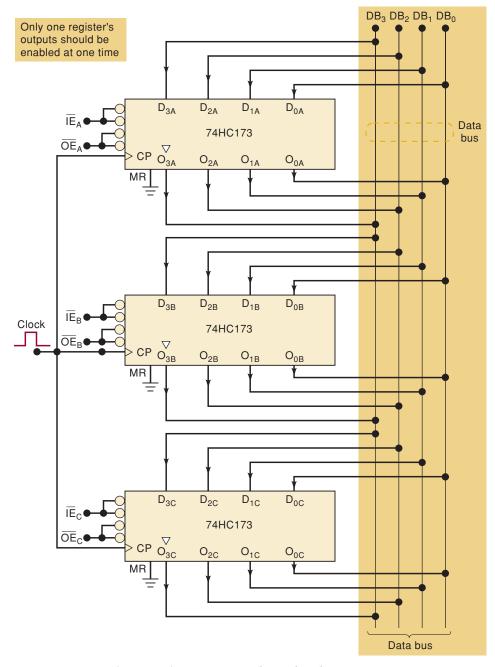
REVIEW QUESTIONS

symbol for the

- 1. Assume that both *IE* inputs are LOW and that $D_0D_1D_2D_3 = 1011$. What logic levels are present at the FF D inputs?
- 2. True or false: The register cannot be loaded when the master reset input (MR) is held HIGH.
- 3. What will the output levels be when *MR* = HIGH and both *OE* inputs are held low?

9-14 DATA BUS OPERATION

The data bus is very important in computer systems, and its significance will not be appreciated until our later studies of memories and microprocessors. For now, we will illustrate the data bus operation for register-to-register data transfer. Figure 9-44 shows a bus-organized system for three 74HC173 tristate registers. Note that each register has its pair of \overline{OE} inputs tied together as one \overline{OE} input, and likewise for the \overline{IE} inputs. Also note that the registers will be referred to as registers *A*, *B*, and *C* from top to bottom. This is indicated by the subscripts on each input and output.





In this arrangement, the data bus consists of four lines labeled DB_0 to DB_3 . Corresponding outputs of each register are connected to the same data bus line (e.g., O_{3A} , O_{3B} , and O_{3C} are connected to DB_3). Because the three registers have their outputs connected together, it is imperative that only one register have its outputs enabled and that the other two register outputs remain in the Hi-Z state. Otherwise, there will be bus contention (two or more sets of outputs fighting each other), producing uncertain levels on the bus and possible damage to the register output buffers.

Corresponding register inputs are also tied to the same bus line (e.g., D_{3A} , D_{3B} , and D_{3C} are tied to DB_3). Thus, the levels on the bus will always be ready to be transferred to one or more of the registers depending on the \overline{IE} inputs.

Data Transfer Operation

The contents of any one of the three registers can be parallel-transferred over the data bus to one of the other registers through the proper application of logic levels to the register enable inputs. In a typical system, the control unit of a computer (i.e., the CPU) will generate the signals that select which register will put its data on the data bus and which one will take the data from the data bus. The following example will illustrate this.

EXAMPLE 9-21

Describe the input signal requirements for transferring $[A] \rightarrow [C]$.

Solution

First of all, only register A should have its outputs enabled. That is, we need

$$\overline{OE}_A = 0$$
 $\overline{OE}_B = \overline{OE}_C = 1$

This will place the contents of register A onto the data bus lines. Next, only register C should have its inputs enabled. For this, we want

$$\overline{IE}_C = 0$$
 $\overline{IE}_A = \overline{IE}_B = 1$

This will allow only register *C* to accept data from the data bus when the PGT of the clock signal occurs.

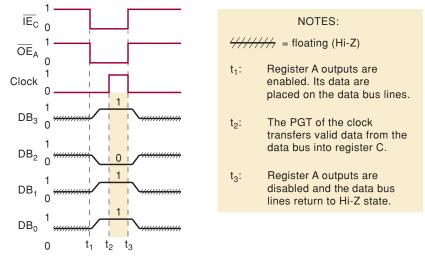
Finally, a clock pulse is required to transfer the data from the bus into the register *C* flip-flops.

Bus Signals

The timing diagram in Figure 9-45 shows the various signals involved in the transfer of the data 1011 from register A to register C. The \overline{IE} and \overline{OE} lines that are not shown are assumed to be in their inactive HIGH state. Prior to time t_1 , the \overline{IE}_C and \overline{OE}_A lines are also HIGH, so that all of the register outputs are disabled, and none of the registers will be placing their data on the bus lines. In other words, the data bus lines are in the Hi-Z or "floating" state as represented by the hatched lines on the timing diagram. The Hi-Z state does not correspond to any particular voltage level.

At t_1 the IE_C and OE_A inputs are activated. The outputs of register A are enabled, and they start changing the data bus lines DB_3 through DB_0 from the Hi-Z state to the logic levels 1011. After allowing time for the logic levels

FIGURE 9-45 Signal activity during the transfer of the data 1011 from register *A* to register *C*.



to stabilize on the bus, the PGT of the clock is applied at t_2 . This PGT will transfer these logic levels into register *C* because \overline{IE}_C is active. If the PGT occurs before the data bus has valid logic levels, unpredictable data will be transferred into *C*.

At t_3 , the IE_C and OE_A lines return to the inactive state. As a result, register *A*'s outputs go to the Hi-Z state. This removes the register *A* output data from the bus lines, and the bus lines return to the Hi-Z state.

Note that the data bus lines show valid logic levels only during the time interval when register *A*'s outputs are enabled. At all other times, the data bus lines are floating, and there is no way to predict easily what they would look like if displayed on an oscilloscope. A logic probe would give an "indeterminate" indication if it were monitoring a floating bus line. Also note the relatively slow rate at which the signals on the data bus lines are changing. Although this effect has been somewhat exaggerated in the diagram, it is a characteristic common to bus systems and is caused by the capacitive load on each line. This load consists of a combination of parasitic capacitance and the capacitances contributed by each input and output connected to the line.

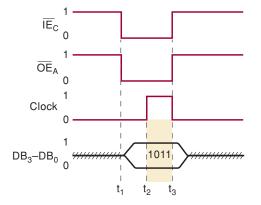
Simplified Bus Timing Diagram

The timing diagram in Figure 9-45 shows the signals on each of the four data bus lines. This same kind of signal activity occurs in digital systems that use the more common data buses of 8, 16, or 32 lines. For these larger buses, the timing diagrams like Figure 9-45 would get excessively large and cumbersome. There is a simplified method for showing the signal activity that occurs on a set of bus lines that uses only a single timing waveform to represent the complete set of bus lines. This is illustrated in Figure 9-46 for the same data transfer situation depicted in Figure 9-45. Notice how the data bus activity is represented. Especially note how the valid data 1011 are indicated on the diagram during the t_2 - t_3 interval. We will generally use this simplified bus timing diagram from now on.

Expanding the Bus

The data transfer operation of the four-line data bus of Figure 9-44 is typical of the operation of larger data buses found in most computers and other dig-

FIGURE 9-46 Simplified way to show signal activity on data bus lines.



ital systems, usually the 8-, 16-, or 32-line data buses. These larger buses generally have many more than three devices tied to the bus, but the basic data transfer operation is the same: one device has its outputs enabled so that its data are placed on the data bus; another device has its inputs enabled so that it can take these data off the bus and latch them into its internal circuitry on the appropriate clock edge.

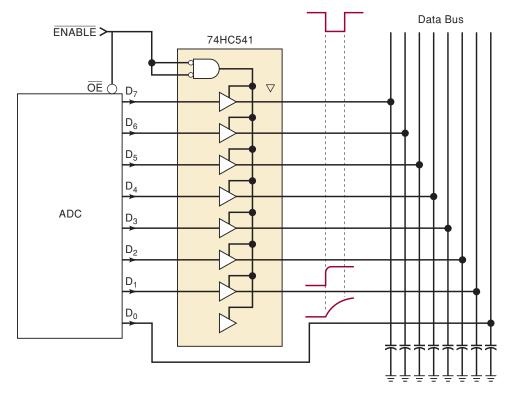
The number of lines on the data bus will depend on the size of the data word (unit of data) that is to be transferred over the bus. A computer that has an 8-bit word size will have an eight-line data bus, a computer that has a 16-bit word size will have a 16-line data bus, and so on. The number of devices connected to a data bus varies from one computer to another and depends on factors such as how much memory the computer has and the number of input and output devices that must communicate with the CPU over the data bus.

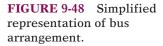
All device outputs must be tied to the bus through tristate buffers. Some devices, such as the 74173 register, have these buffers on the same chip. Other devices will need to be connected to the bus through an IC called a **bus driver**. A bus driver IC has tristate outputs with a very low output impedance that can rapidly charge and discharge the bus capacitance. This bus capacitance represents the cumulative effect of all of the parasitic capacitances of the different inputs and outputs tied to the bus, and it can cause deterioration of the bus signal transition times if they are not driven from a low-impedance signal source. Figure 9-47 shows a 74HC541 octal bus driver IC connecting the outputs of an eight-bit analog-to-digital converter (ADC) to a data bus. The ADC has tristate outputs but lacks the drive capability to charge the bus capacitance (shown as capacitors to ground in the drawing). Notice that data bit 0 is driving the bus directly, without the assistance of the bus driver. If the transition time is slow enough, the voltage may never reach a HIGH logic level in the allotted enable time. The bus driver's two enable inputs are tied together so that a LOW on the common enable line will allow the ADC's outputs through the buffers and onto the data bus, from which they can be transferred to another device.

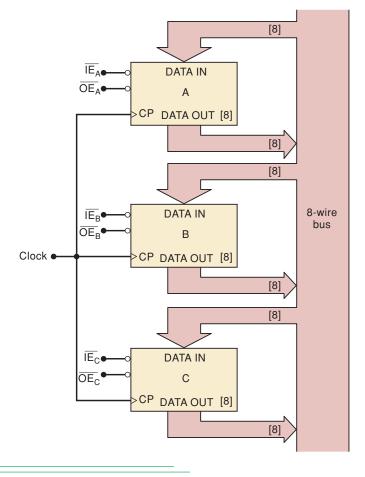
Simplified Bus Representation

Usually, many devices are connected to the same data bus. On a circuit schematic, this can produce a confusing array of lines and connections. For this reason, a more simplified representation of data bus connections is often used on block diagrams and in some circuit schematics. One type of simplified representation is shown in Figure 9-48 for an eight-line data bus.

FIGURE 9-47 A 74HC541 octal bus driver connects the outputs of an analog-to-digital converter (ADC) to an eight-line data bus. The D_0 output connects directly to the bus showing the capacitive effects.







The connections to and from the data bus are represented by wide arrows. The numbers in brackets indicate the number of bits that each register contains, as well as the number of lines connecting the register inputs and outputs to the bus.

Another common method for representing buses on a schematic is presented in Figure 9-49 for an eight-line data bus. It shows the eight individual output lines from a 74HC541 bus driver labeled D_7-D_0 bundled (not connected) together and shown as a single line. These bundled data output lines are then connected to the data bus, which is also shown as one line (i.e., the eight data bus lines are bundled together). The "/8" notation indicates the number of lines represented by each bundle. This bundle method is used to represent the connections from the data bus to the eight microprocessor data inputs. When the bundle method is used, it is very important to label both ends of every wire that is in the bundle because the connection cannot be traced visually on the diagram.

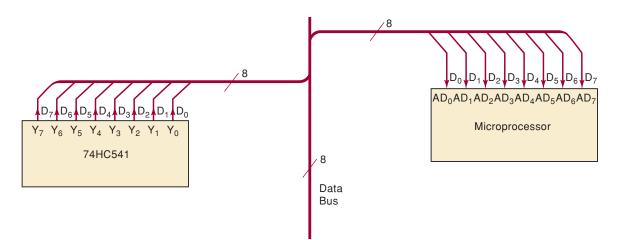


FIGURE 9-49 Bundle method for simplified representation of data bus connections. The "/8" denotes an eight-line data bus.

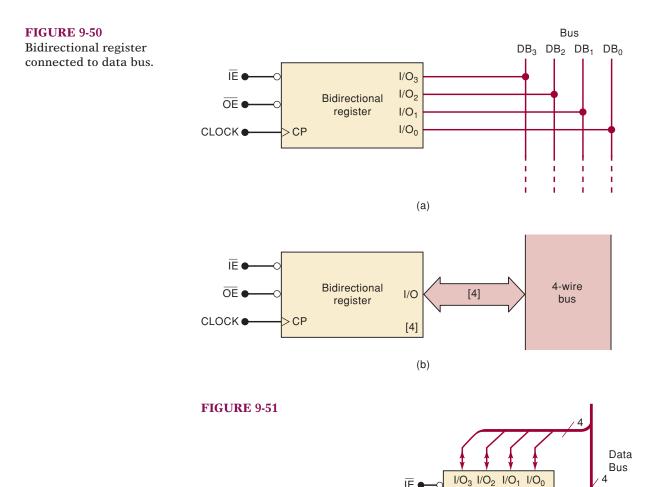
Bidirectional Busing

Each register in Figure 9-44 has both its inputs and its outputs connected to the data bus, so that corresponding inputs and outputs are shorted together. For example, each register has output O_2 connected to input D_2 because of their common connection to DB_2 . This, of course, would not be true if external bus drivers were connected between the register outputs and the data bus.

Because inputs and outputs are often connected together in bus systems, IC manufacturers have developed ICs that connect inputs and outputs together *internal* to the chip in order to reduce the number of IC pins and the number of connections to the bus. Figure 9-50 illustrates this for a four-bit register. The separate data input lines (D_0 to D_3) and output lines (O_0 to O_3) have been replaced by input/output lines (I/O₀ to I/O₃).

Each I/O line will function as either an input or an output depending on the states of the enable inputs. Thus, they are called **bidirectional data lines**. The 74ALS299 is an eight-bit register with common I/O lines. Many memory ICs and microprocessors have bidirectional transfer of data.

We will return to the important topic of data busing in our comprehensive coverage of memory systems in Chapter 12.





1. What will happen if $\overline{OE}_A = \overline{OE}_B = \text{LOW}$ in Figure 9-44?

ŌĒ

CLOCK (

2. What logic level is on a data bus line when all devices tied to the bus are disabled?

Bidirectional

register

- 3. What is the function of a bus driver?
- 4. What are the reasons for having registers with common I/O lines?
- 5. Redraw Figure 9-50(a) using the bundled line representation. (The answer is shown in Figure 9-51.)

9-15 DECODERS USING HDL

Section 9-1 introduced the decoder as a device that can recognize a binary number on its input and activate a corresponding output. Specifically, the 74138 1-of-8 decoder was presented. It uses three binary inputs to activate one of the eight outputs when the chip is enabled. In order to study HDL methods for implementing the types of digital devices that are covered in this chapter, we will focus primarily on conventional MSI parts, which have been discussed earlier. Not only is the operation of these devices already described in this book, but further reference material is readily available in logic data books. In all of these cases, it is vital that you understand what the device is supposed to do before trying to dissect the HDL code that describes it.

In actual practice, we are not recommending, for example, that new code be written to perform the task of a 74138. After all, there is a macrofunction already available that works exactly like this standard part. Using these devices as examples and showing the HDL techniques used to create them opens the door for embellishment of these devices so that a circuit that will uniquely fit the application at hand can be described. In some instances, we will add our own embellishments to a circuit that has been described; in other instances, we will describe a simpler version of a part in order to focus on the core principle in HDL and avoid other confusing features.

The methods used to define the inputs and outputs should take into consideration the purpose of these signals. In the case of a 1-of-8 decoder such as the 74138 described in Figure 9-3, there are three enable inputs (\overline{E}_1 , \overline{E}_2 , and E_3) that should be described as individual inputs to the device. On the other hand, the binary inputs that are to be decoded (A_2 , A_1 , A_0) should be described as three-bit numbers. The outputs can be described as eight individual bits. They can also be described as an array of eight bits, with output 0 represented by element 0 in the array, and so on, to output 7 represented by element 7. Depending on the way the code is written, one strategy may be easier to write than the other. Generally, using individual names can make the purpose of each I/O bit clearer, and using bit arrays makes it easier to write the code.

When an application such as a decoder calls for a unique response from the circuit corresponding to each combination of its input variables, the two methods that best serve this purpose are the CASE construct and the truth TABLE. The interesting aspect of this decoder is that the output response should happen only when *all* the enables are activated. If any of the enables are not in their active state, it should cause all the outputs to go HIGH. Each of the examples that follow will demonstrate ways to decode the input number only when *all* of the enables are activated.

AHDL DECODERS

The first illustration of an AHDL decoder, shown in Figure 9-52, is intended to demonstrate the use of a CASE construct that is evaluated only under the condition that all enables are active. The outputs must all revert back to HIGH as soon as any enable is deactivated. This example also illustrates a way to accomplish this without explicitly assigning a value to each output for each case, and it uses individually named output bits.

Line 3 defines the three-bit binary number that will be decoded. Line 4 defines the three enable inputs, and line 5 specifically names each output. The unique property of this solution is the use of the **DEFAULTS** keyword in AHDL (lines 10 to 13) to establish a value for variables that are not specified elsewhere in the code. This maneuver allows each case to force one bit LOW without specifically stating that the others must go HIGH.

The next illustration, in Figure 9-53, is intended to demonstrate the same decoder using the truth table approach. Notice that the outputs are defined as bit arrays but are still numbered y[7] down to y[0]. The unique aspect of this code is the use of the don't-care values in the truth table. Line 11 is used to concatenate the six input bits into a single variable (bit array) named *inputs[]*. Notice that in lines 14, 15, and 16 of the table, only one bit value is specified as 1 or 0. The others are all in the don't-care state (X). Line 14 says, "As long as e3

FIGURE 9-52 AHI		
equivalent to the	1	SUBDESIGN fig9_52
74138 decoder.	2	
	3	a[20] :INPUT; binary inputs
	4	e3, e2bar, e1bar :INPUT; enable inputs
	5	y7,y6,y5,y4,y3,y2,y1,y0 :OUTPUT; decoded outputs
	6)
	7	VARIABLE
	8	enable :NODE;
	9	BEGIN
	10	DEFAULTS
	11	y7=VCC;y6=VCC;y5=VCC;y4=VCC;
	12	y3=VCC;y2=VCC;y1=VCC;y0=VCC; defaults all HIGH out
	13	END DEFAULTS;
	14	enable = e3 & !e2bar & !e1bar; all enables activated
	15	IF enable THEN
	16	CASE a[] IS
	17	WHEN $0 \Rightarrow y0 = GND;$
	18	WHEN 1 => $y1 = GND;$
	19	WHEN 2 => y^2 = GND;
	20	WHEN 3 => $y3$ = GND;
	21	WHEN 4 => $y4$ = GND;
	22	WHEN 5 => $y5$ = GND;
	23	WHEN 6 => $Y6 = GND;$
	24	WHEN 7 => $y7$ = GND;
	25	END CASE;
	26	END IF;
	27	END;
FIGURE 9-53 AHI	DL	
decoder using a	1	SUBDESIGN fig9_53
TABLE.	2	(
		a[20] :INPUT; decoder inputs
	3	
	4	e3, e2bar, e1bar :INPUT; enable inputs
	4 5	
	4 5 6	e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs)
	4 5 6 7	e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE
	4 5 6 7 8	e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs)
	4 5 6 7 8 9	e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined
	4 5 7 8 9 10	e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN
	4 5 7 8 9 10 11	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs</pre>
	4 5 7 8 9 10 11 12	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE</pre>
	4 5 7 8 9 10 11 12 13	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE inputs[] => y[];</pre>
	4 5 7 8 9 10 11 12 13 14	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE inputs[] => y[]; B"0XXXXX" => B"11111111"; el not enabled</pre>
	4 5 7 8 9 10 11 12 13 14 15	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE inputs[] => y[]; B"0XXXXX" => B"11111111"; el not enabled B"X1XXXX" => B"11111111"; e2bar disabled</pre>
	4 5 7 8 9 10 11 12 13 14 15 16	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE inputs[] => y[]; B"OXXXXX" => B"1111111"; e1 not enabled B"X1XXXX" => B"1111111"; e2bar disabled B"X1XXXX" => B"1111111"; e3bar disabled</pre>
	4 5 6 7 8 9 10 11 12 13 14 15 16 17	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE inputs[] => y[]; B"OXXXXX" => B"1111111"; e1 not enabled B"X1XXXX" => B"1111111"; e2bar disabled B"X1XXXX" => B"1111111"; e3bar disabled B"100000" => B"1111110"; Y0 active</pre>
	4 5 6 7 8 9 10 11 12 13 14 15 16 17 18	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE inputs[] => y[]; B"OXXXXX" => B"1111111"; e1 not enabled B"X1XXXX" => B"1111111"; e2bar disabled B"X1XXXX" => B"1111111"; e3bar disabled B"100000" => B"1111110"; Y0 active B"100001" => B"1111110"; Y1 active</pre>
	4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE inputs[] => y[]; B"OXXXXX" => B"1111111"; e1 not enabled B"X1XXXX" => B"1111111"; e2bar disabled B"X1XXXX" => B"1111111"; e3bar disabled B"100000" => B"1111110"; Y0 active B"100001" => B"1111101"; Y1 active B"100010" => B"1111101"; Y2 active</pre>
	4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE inputs[] => y[]; B"0XXXXX" => B"1111111"; e1 not enabled B"X1XXXX" => B"1111111"; e2bar disabled B"X1XXXX" => B"1111111"; e3bar disabled B"10000" => B"1111110"; Y0 active B"10001" => B"1111101"; Y1 active B"100010" => B"1111011"; Y2 active B"100011" => B"1111011"; Y3 active</pre>
	4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE inputs[] => y[]; B"0XXXXX" => B"1111111"; e1 not enabled B"X1XXXX" => B"1111111"; e2bar disabled B"X1XXXX" => B"1111111"; e3bar disabled B"10000" => B"1111110"; Y0 active B"10001" => B"1111101"; Y1 active B"10001" => B"1111011"; Y2 active B"10001" => B"1111011"; Y3 active B"10010" => B"1110111"; Y4 active</pre>
	4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE inputs[] => y[]; B"OXXXXX" => B"1111111"; e1 not enabled B"X1XXX" => B"1111111"; e2bar disabled B"X1XXX" => B"1111111"; e3bar disabled B"10000" => B"1111110"; Y0 active B"10001" => B"1111101"; Y1 active B"10001" => B"1111011"; Y2 active B"10011" => B"1111011"; Y3 active B"10010" => B"1110111"; Y4 active B"10010" => B"1101111"; Y5 active</pre>
	4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE</pre>
	4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24	<pre>e3, e2bar, elbar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, elbar, a[]); concatenate the inputs TABLE inputs[] => y[]; B"OXXXXX" => B"1111111"; el not enabled B"X1XXX" => B"1111111"; e2bar disabled B"X1XXX" => B"1111111"; e3bar disabled B"10000" => B"1111110"; Y0 active B"10001" => B"1111101"; Y1 active B"10001" => B"1111011"; Y2 active B"10011" => B"1110111"; Y3 active B"10010" => B"1110111"; Y4 active B"10010" => B"1101111"; Y5 active B"10010" => B"1011111"; Y5 active B"10011" => B"1011111"; Y6 active B"100111" => B"01111111"; Y7 active</pre>
	4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE inputs[] => y[]; B"OXXXXX" => B"11111111"; e1 not enabled B"X1XXX" => B"1111111"; e2bar disabled B"X1XXX" => B"1111111"; e3bar disabled B"10000" => B"1111110"; Y0 active B"10001" => B"1111101"; Y1 active B"10001" => B"1111011"; Y2 active B"10011" => B"1110111"; Y3 active B"10010" => B"1110111"; Y4 active B"10010" => B"1101111"; Y5 active B"10011" => B"1011111"; Y6 active B"100111" => B"0111111"; Y7 active END TABLE;</pre>
	4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24	<pre>e3, e2bar, e1bar :INPUT; enable inputs y[70] :OUTPUT; decoded outputs) VARIABLE inputs[50] :NODE; all 6 inputs combined BEGIN inputs[] = (e3, e2bar, e1bar, a[]); concatenate the inputs TABLE inputs[] => y[]; B"OXXXXX" => B"1111111"; e1 not enabled B"X1XXXX" => B"1111111"; e2bar disabled B"XX1XXX" => B"1111111"; e3bar disabled B"10000" => B"1111110"; Y0 active B"10001" => B"1111101"; Y1 active B"10001" => B"1111101"; Y2 active B"10011" => B"1111011"; Y3 active B"10010" => B"1110111"; Y4 active B"10010" => B"1101111"; Y5 active B"100110" => B"1011111"; Y6 active B"100111" => B"1011111"; Y7 active</pre>

is *not* enabled, it does not matter what the other inputs are doing; the outputs will be HIGH." Lines 15 and 16 do the same thing, making sure that if *e2bar* or *e1bar* is HIGH (disabled), the outputs will be HIGH. Lines 17 through 24 state that as long as the first three bits (enables) are "100," the proper decoder output will be activated to correspond with the lower three bits of *inputs[]*.

VHDL DECODERS

The VHDL solution presented in Figure 9-54 essentially uses a truth table approach. The key strategy in this solution involves the concatenation of the three enable bits (*e3, e2bar, e1bar*) with the binary input *a* on line 11. The VHDL selected signal assignment is used to assign a value to a signal when a specific combination of inputs is present. Line 12 (WITH inputs SELECT) indicates that we are using the value of the intermediate signal *inputs* to determine which value is assigned to *y*. Each of the *y* outputs is listed on lines 13–20. Notice that only combinations that begin with 100 follow the WHEN clause on lines 13–20. This combination of *e3, e2bar,* and *e1bar* is necessary to make each of the enables active. Line 21 assigns a disabled state to each output when any combination other than 100 is present on the enable inputs.

```
1
      ENTITY fig9_54 IS
 2
      PORT (
 3
                             :IN BIT_VECTOR (2 DOWNTO 0);
           а
 4
           e3, e2bar, e1bar : IN BIT;
 5
                             :OUT BIT_VECTOR (7 DOWNTO 0)
           У
 6
          );
 7
      END fig9_54 ;
 8
      ARCHITECTURE truth OF fig9_54 IS
 9
      SIGNAL inputs: BIT_VECTOR (5 DOWNTO 0); --combine enables w/ binary in
         BEGIN
10
            inputs <= e3 & e2bar & e1bar & a;
11
12
            WITH inputs SELECT
13
               y <= "11111110" WHEN "100000",
                                                 --Y0 active
                     "11111101" WHEN "100001",
                                                 --Y1 active
14
15
                     "11111011" WHEN "100010",
                                                 --Y2 active
                     "11110111" WHEN "100011",
                                                 --Y3 active
16
17
                     "11101111" WHEN "100100", --Y4 active
                     "11011111" WHEN "100101",
18
                                                  --Y5 active
                     "10111111" WHEN "100110",
19
                                                 --Y6 active
20
                     "011111111" WHEN "100111",
                                                 --Y7 active
                     "111111111" WHEN OTHERS;
                                                  --disabled
21
22
          END truth;
```

FIGURE 9-54 VHDL equivalent to the 74138 decoder.

REVIEW QUESTIONS

- 1. What is the purpose of the three inputs e3, e2bar, and e1bar?
- 2. Name two AHDL methods to describe a decoder's operation.
 - 3. Name two VHDL methods to describe a decoder's operation.

9-16 THE HDL 7-SEGMENT DECODER/DRIVER

Section 9-2 described a BCD-to-7-segment decoder/driver. The standard part number for the circuit described is a 7447. In this section, we look into the HDL code necessary to produce a device that meets the same criteria as the 7447. Recall that the BI (blanking input) is the overriding control that turns all segments off regardless of other input levels. The LT (lamp test) input is used to test all the segments on the display by lighting them up. The RBO (ripple blanking output) is designed to go LOW when *RBI* (ripple blanking input) is LOW and the BCD input value is 0. Typically, in multiple-digit display applications, each RBO pin is connected to the RBI pin of the next digit to the right. This setup creates the feature of blanking all leading zeros in a display value without blanking zeros in the middle of a number. For example, the number 2002 would display as 2002, but the number 0002 would not display as 0002, but rather _ _ _ 2. One feature of the 7447 that would be difficult to replicate in HDL is the combination input/output pin named *BI/RBO*. Rather than complicate the code, we have decided to create a separate input (BI) and an output (RBO) on two different pins. This discussion also makes no attempt to replicate the non-BCD display characters of a 7447 but simply blanks all segments for values greater than 9.

Several decisions must be made when designing a circuit such as this one. The first involves the type of display we intend to use. If it is a common cathode, then a logic 1 lights the LED segment. If it is a common anode, then a logic 0 is required to turn on a segment. Next, we must decide on the type of inputs, outputs, and intermediate variables. We have decided that the outputs for each individual segment should be assigned a bit name (a-g) rather than using a bit array. This arrangement will make it clearer when connecting the display to the IC. These individual bits can be grouped as a set of bits and assigned binary values, as we have done in AHDL, or an intermediate variable bit array can be used to make it convenient when assigning all seven bit levels in a single statement, as we have done in VHDL. The BCD inputs are treated as a four-bit number, and the blanking controls are individual bits. The other issue that greatly affects the bit patterns in the HDL code is the arbitrary decision of the order of the segment names a-g. In this discussion, we have assigned segment *a* to the leftmost bit in the binary bit pattern, with the bits moving alphabetically left to right.

Some of the controls must have precedence over other controls. For example, the \overline{LT} (lamp test) should override any regular digit display, and the \overline{BI} (blanking input) should override even the lamp test input. In these illustrations, the IF/ELSE control structure is used to establish precedence. The first condition that is evaluated as true will determine the resulting output, regardless of the other input levels. Subsequent ELSE statements will have no effect, which is why the code tests first for \overline{BI} , then \overline{LT} , then \overline{RBI} , and finally determines the correct segment pattern.

AHDL DECODER/DRIVER

The AHDL code for this circuit is shown in Figure 9-55. AHDL allows output bits to be grouped in a set by separating the bits with commas and enclosing them in parentheses. A group of binary states can be assigned directly to these bit sets, as shown on lines 9, 11, 13, and 15. This convention avoids the need for an intermediate variable and is much shorter than eight separate assignment statements. The TABLE feature of AHDL is useful in this application to correlate an input BCD value to a 7-segment bit pattern.

```
1
      SUBDESIGN fig9_55
 2
       (
 3
          bcd[3..0]
                               : INPUT;
                                                -- 4-bit number
 4
          lt, bi, rbi
                               : INPUT;
                                                -- 3 independent controls
 5
                               :OUTPUT;
                                                -- individual outputs
          a,b,c,d,e,f,g,rbo
 6
      )
 7
      BEGIN
 8
           IF !bi THEN
 9
                                                                 % blank all %
               (a,b,c,d,e,f,g,rbo) = (1,1,1,1,1,1,1,0);
10
           ELSIF
                      !lt THEN
11
                (a,b,c,d,e,f,g,rbo) = (0,0,0,0,0,0,0,1);
                                                                 % test segments %
12
           ELSIF !rbi & bcd[] == 0 THEN
13
                (a,b,c,d,e,f,g,rbo) = (1,1,1,1,1,1,1,0);
                                                                 % blank leading 0's %
           ELSIF bcd[] > 9 THEN
14
15
                (a,b,c,d,e,f,g,rbo) = (1,1,1,1,1,1,1,1);
                                                                 % blank non BCD input %
16
           ELSE
17
               TABLE
                                            % display 7 segment Common Anode pattern %
18
               bcd[]
                         =>
                                a,b,c,d,e,f,g,rbo;
19
               0
                                0, 0, 0, 0, 0, 0, 0, 1, 1;
                         =>
20
               1
                                1,0,0,1,1,1,1,1;
                         =>
21
               2
                                0, 0, 1, 0, 0, 1, 0, 1;
                         =>
               3
22
                         =>
                                0, 0, 0, 0, 1, 1, 0, 1;
               4
23
                         =>
                                1,0,0,1,1,0,0,1;
               5
                                0, 1, 0, 0, 1, 0, 0, 1;
24
                         =>
25
               6
                                1,1,0,0,0,0,0,1;
                          =>
               7
26
                                0,0,0,1,1,1,1,1;
                         =>
27
               8
                         =>
                                0, 0, 0, 0, 0, 0, 0, 0, 1;
28
               9
                                0,0,0,1,1,0,0,1;
                          =>
29
               END TABLE;
30
           END IF;
31
      END;
```

FIGURE 9-55 AHDL 7-segment BCD display decoder.

VHDL DECODER/DRIVER

The VHDL code for this circuit is shown in Figure 9-56. This illustration demonstrates the use of a VARIABLE as opposed to a SIGNAL. A VARIABLE can be thought of as a piece of scrap paper used to write down some numbers that will be needed later. A SIGNAL, on the other hand, is usually thought of as a wire connecting two points in the circuit. In line 12, the keyword VARIABLE is used to declare *segments* as a bit vector with seven bits. Take special note of the order of the indices for this variable. They are declared as 0 TO 6. In VHDL, this means that element 0 appears on the left end of the binary bit pattern and element 6 appears on the right end. This is exactly opposite of the way most examples in this text have presented variables, but it is important to realize the significance of the declaration statement in VHDL. For this illustration, segment *a* is bit 0 (on the left), segment *b* is bit 1 (moving to the right), and so on.

Notice that on line 3, the BCD input is declared as an INTEGER. This allows us to refer to it by its numeric value in decimal rather than being limited to bit pattern references. A PROCESS is employed here in order to allow us to use the IF/ELSE constructs to establish the precedence of one input over the other. Notice that the sensitivity list contains all the inputs. The code within the

```
1
      ENTITY fig9_56 IS
 2
      PORT (
 3
               bcd
                                  :IN INTEGER RANGE 0 TO 15;
               lt, bi, rbi :IN BIT;
 4
 5
               a,b,c,d,e,f,g,rbo :OUT BIT
 6
            );
 7
      END fig9_56 ;
 8
 9
      ARCHITECTURE vhdl OF fig9_56 IS
10
      BEGIN
      PROCESS (bcd, lt, bi, rbi)
11
12
      VARIABLE segments :BIT_VECTOR (0 TO 6);
13
          BEGIN
             IF bi = '0' THEN
14
15
                segments := "11111111"; rbo <= '0'; -- blank all</pre>
             ELSIF lt = '0' THEN
16
17
                segments := "0000000"; rbo <= '1'; -- test segments</pre>
             ELSIF (rbi = '0' AND bcd = 0) THEN
18
                segments := "1111111"; rbo <= '0'; -- blank leading 0's</pre>
19
20
             ELSE
21
                rbo <= '1';
22
                CASE bcd IS
                                -- display 7 segment Common Anode pattern
                   WHEN 0
23
                                => segments := "0000001";
                   WHEN 1
                                => segments := "1001111";
24
25
                   WHEN 2
                                => segments := "0010010";
26
                   WHEN 3
                                => segments := "0000110";
27
                   WHEN 4
                                => segments := "1001100";
28
                   WHEN 5
                                => segments := "0100100";
                   WHEN 6
                                => segments := "1100000";
29
30
                   WHEN 7
                               => segments := "0001111";
31
                   WHEN 8
                                => segments := "0000000";
32
                   WHEN 9
                                => segments := "0001100";
33
                   WHEN OTHERS => segments := "1111111";
34
                END CASE;
35
             END IF;
36
          a <= segments(0); --assign bits of array to output pins</pre>
37
          b <= segments(1);</pre>
38
          c <= segments(2);</pre>
39
          d <= segments(3);</pre>
40
          e <= segments(4);</pre>
41
          f <= segments(5);</pre>
42
          g <= segments(6);
43
              END PROCESS;
44
      END vhdl;
```

```
FIGURE 9-56 VHDL 7-segment BCD display decoder.
```

PROCESS describes the behavioral operation of the circuit that is necessary whenever any of the inputs in the sensitivity list changes state. Another very important point in this illustration is the assignment operator for variables. Notice in line 15, for example, the statement *segments* := "1111111". The variable assignment operator := is used for variables in place of the <= operator that was used for signal assignments. In lines 36–42, the individual bits that were established in the IF/ELSE decisions are assigned to the proper output bits.

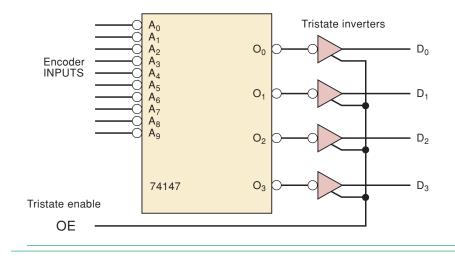
REVIEW QUESTIONS

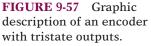
- 1. What feature of a 7447 is very difficult to duplicate in PLD hardware and HDL code?
- 2. Are these illustrations intended to drive common-anode or commoncathode 7-segment displays?
- 3. How are certain inputs (e.g., lamp test) given precedence over other inputs (e.g., RBI) in the HDL code in this section?

9-17 ENCODERS USING HDL

In Section 9-4, we discussed encoders and priority encoders. Similarities exist, of course, between decoders and encoders. Decoders take a binary number and activate one output that corresponds to that number. An encoder works in the other direction by monitoring one of its several inputs; when one of the inputs is activated, it produces a binary number corresponding to that input. If more than one of its inputs is activated at the same time, a priority encoder ignores the input of lower significance and produces the binary value that corresponds to the most significant input. In other words, it gives more significant inputs priority over less significant inputs. This section focuses on the methods that can be used in HDL to describe circuits that have this characteristic of priority for some inputs over others.

Another very important concept, which was presented in Chapter 8, was the tristate output circuit. Devices with tristate outputs can produce a logic HIGH or a logic LOW, just like a normal circuit, when their output is enabled. However, these devices can have their outputs disabled, which puts them in a "disconnected" or a high-impedance state. This is very important for devices connected to common buses, as described in Section 9-12. The next logical question is, "How do we describe tristate outputs using HDL?" This section incorporates tristate outputs in the encoder design to address this issue. In order to keep the discussion focused on the essentials, we create a circuit that emulates the 74147 priority encoder, with one added feature of having active-HIGH tristate outputs. Other features like cascading inputs and outputs (such as those found on a 74148) are left for you to try later. A symbol for the circuit we are describing is shown in Figure 9-57. Because the inputs are all labeled in a manner very similar to bit array notation, it makes sense to use a bit array to describe the encoder inputs. The tristate enable must be a single bit, and the encoded outputs can be described as an integer numeric value.





AHDL ENCODER

The most important point to be made from Figure 9-58 is the method of establishing priority, but also note the I/O assignments. The AHDL input/output descriptions do not provide a separate type for integers but allow a bit array to be referred to as an integer. Consequently, line 4 describes the outputs as a bit array. In this illustration, a TABLE is used that is very similar to the tables often found in data books describing this circuit's operation. The key to this table is the use of the don't-care state (X) on inputs. The priority is described by the way we position these don't-care states in the truth table. Reading line 15, for instance, we see that as soon as we encounter an active input (LOW on input a/4), the lower order input bits do not matter. The output has been determined to be 4. The tristate outputs are made possible by using the built-in primitive function :TRI on line 6. This line assigns the attributes of a tristate buffer to the variable that has been named buffer. Recall that this is the same way a flip-flop is described in AHDL. The ports of a tristate buffer are quite straightforward. They represent the input (*in*), the output (out), and the tristate output enable (oe).

FIGURE 9-58 AHDL priority encoder with tristate outputs.

```
1
      SUBDESIGN fig9_58
 2
      (
 3
         a[9..0], oe
                               : INPUT;
 4
         d[3..0]
                               :OUTPUT;
 5
 6
      VARIABLE buffer[3..0]
                               :TRI;
 7
      BEGIN
 8
         TABLE
 9
            a[]
                            => buffer[].in;
10
            B"1111111111" => B"1111";
                                            -- no input active
            B"1111111110" => B"0000";
                                            -- 0
11
12
            B″111111110X″
                            => B"0001";
                                               1
            B"11111110XX" => B"0010";
                                               2
13
            B"1111110XXX" => B"0011";
                                            -- 3
14
15
            B″111110XXXX″
                           => B"0100";
                                               4
16
            B"11110XXXXX" => B"0101";
                                               5
17
            B"1110XXXXXX" => B"0110";
                                              6
            B"110XXXXXXX" => B"0111";
                                               7
18
19
            B″10XXXXXXXX
                           => B"1000";
                                              8
                                            _ _
20
            B"0XXXXXXXXX => B"1001";
                                            -- 9
21
         END TABLE;
22
         buffer[].oe = oe;
                               -- hook up enable line
23
         d[] = buffer[].out; -- hook up outputs
24
      END;
```

The next illustration (Figure 9-59) uses the IF/ELSE construct to establish priority, very much like the method demonstrated in the 7-segment decoder example. The first IF condition that evaluates TRUE will THEN cause the corresponding value to be applied to the tristate buffer inputs. The priority is established by the order in which we list the IF conditions. Notice that they start with input 9, the highest-order input. This illustration adds another feature of putting the outputs into the high-impedance state when no input is being activated. Line 20 shows that the output enables will be activated only

```
1
      SUBDESIGN fig9_59
 2
      (
 3
         sw[9..0], oe
                         : INPUT;
         d[3..0]
 4
                         :OUTPUT;
 5
      )
 6
      VARIABLE
 7
         buffers[3..0]
                         :TRI;
 8
      BEGIN
 9
         TF
                !sw[9]
                         THEN
                               buffers[].in = 9;
                               buffers[].in = 8;
10
         ELSIF !sw[8]
                         THEN
         ELSIF !sw[7]
                               buffers[].in = 7;
11
                         THEN
12
         ELSIF !sw[6]
                               buffers[].in = 6;
                         THEN
13
         ELSIF !sw[5]
                         THEN
                               buffers[].in = 5;
14
         ELSIF !sw[4]
                         THEN
                               buffers[].in = 4;
15
                               buffers[].in = 3;
         ELSIF !sw[3]
                         THEN
                               buffers[].in = 2;
16
         ELSIF !sw[2]
                         THEN
17
         ELSIF !sw[1]
                         THEN
                               buffers[].in = 1;
18
         ELSE
                                buffers[].in = 0;
         END IF;
19
20
         buffers[].oe = oe & sw[]!=b"1111111111"; -- enable on any input
21
         d[] = buffers[].out;
                                                      -- connect to outputs
22
      END:
```

FIGURE 9-59 AHDL priority encoder using IF/ELSE.

when the *oe* pin is activated and one of the inputs is activated. Another item of interest in this illustration is the use of bit array notation to describe individual inputs. For example, line 9 states that IF switch input 9 is activated (LOW), THEN the inputs to the tristate buffer will be assigned the value 9 (in binary, of course).

VHDL ENCODER

Two very important VHDL techniques are demonstrated in this description of a priority encoder. The first is the use of tristate outputs in VHDL, and the second is a new method of describing priority. Figure 9-60 shows the input/output definitions for this encoder circuit. Notice on line 6 that the input switches are defined as bit vectors with indices from 9 to 0. Also note that the d output is defined as an IEEE standard bit array (std_logic_vector type). This definition is necessary to allow the use of high-impedance states (tristate) on the outputs and also explains the need for the LIBRARY and USE statements on lines 1 and 2. As we mentioned, a very important point of this illustration is the method of describing precedence for the inputs. This code uses the conditional signal assignment statement starting on line 14 and continuing through line 24. On line 14, it assigns the value listed to the right of \leq to the variable *d* on the left, assuming the condition following WHEN is true. If this clause is not true, the clauses following ELSE are evaluated one at a time until one that is true is found. The value preceding WHEN will then be assigned to d. A very important attribute of the conditional signal assignment statement is the sequential evaluation. The precedence of these statements is established by the order in which they are listed. Notice that in this illustration, the first condition being tested (line 14) is the enabling of the tristate outputs. Recall from Chapter 8 that the three states of a tristate out-

CHAPTER 9/MSI LOGIC CIRCUITS

FIGURE 9-60		
VHDL priority	1	LIBRARY ieee;
encoder using	2	USE ieee.std_logic_1164.ALL;
conditional	3	
signal	4	ENTITY fig9_60 IS
assignment.	5	PORT (
	6	sw :IN BIT_VECTOR (9 DOWNTO 0); standard logic not needed
	7	oe :IN BIT; standard logic not needed
	8	d :OUT STD_LOGIC_VECTOR (3 DOWNTO 0) std logic for hi-Z
	9);
	10	END fig9_60;
	11	
	12	ARCHITECTURE a OF fig9_60 IS
	13	BEGIN
	14	d <= "ZZZZ" WHEN ((oe = '0') OR (sw = "1111111111")) ELSE
	15	"1001" WHEN sw(9) = '0' ELSE
	16	"1000" WHEN sw(8) = '0' ELSE
	17	"0111" WHEN sw(7) = '0' ELSE
	18	"0110" WHEN sw(6) = '0' ELSE
	19	"0101" WHEN sw(5) = '0' ELSE
	20	"0100" WHEN $sw(4) = '0'$ ELSE
	21	" $0011''$ WHEN sw(3) = '0' ELSE
	22	0010'' WHEN sw(2) = '0' ELSE
	23	0010 WHEN $sw(2) = 0$ HESE 0001'' WHEN $sw(1) = 0'$ ELSE
	23 24	0001 WHEN $sw(1) = 0$ ELSE 0000'' WHEN $sw(0) = '0';$
	25	END a;

put are HIGH, LOW, and high impedance, which is referred to as high Z. When the value "ZZZZ" is assigned to the output, each output is in the highimpedance state. If the outputs are to be disabled (high Z), then none of the other encoding matters. Line 15 tests the highest priority input, which is bit 9 of the *sw* input array. If it is active (LOW), then a value of 9 is output regardless of whether other inputs are being activated at the same time.

REVIEW QUESTIONS

- 1. Name two methods in AHDL for giving priority to some inputs over others.
- 2. Name two methods in VHDL for giving priority to some inputs over others.
- 3. In AHDL, how are tristate outputs implemented?
- 4. In VHDL, how are tristate outputs implemented?

9-18 HDL MULTIPLEXERS AND DEMULTIPLEXERS

A multiplexer is a device that acts like a selector switch for digital signals. The select inputs are used to specify the input channel that is to be "connected" to the output pins. A demultiplexer works in the opposite direction by taking a digital signal as an input and distributing it to one of its outputs. Figure 9-61 shows a multiplexer/demultiplexer system with four data input channels. Each input is a four-bit number. These devices are not exactly like any of the multiplexers or demultiplexers described earlier in this chapter, but they operate in the same way. The system in this illustration allows the four digital signals to share a common "pipeline" in order to get data from

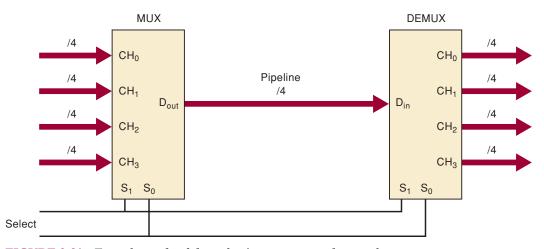


FIGURE 9-61 Four channels of data sharing a common data path.

one point to the other. The select inputs are used to decide which signal is going through the pipeline at any time.

In this section, we examine some code that implements both the multiplexer and the demultiplexer. The key HDL issue in both the MUX and DE-MUX is assigning signals under certain conditions. For the demux, another issue is assigning a state to whichever outputs are not currently selected to distribute data. In other words, when an output is not being used for data (not selected), do we want it to have all bits HIGH, all bits LOW, or the tristate disabled? In the following descriptions, we have chosen to make them all HIGH when not selected, but with the structure shown, it would be a simple matter to change to one of the other possibilities.

AHDL MUX AND DEMUX

We will implement the multiplexer first. Figure 9-62 describes a multiplexer with four inputs of four bits each. Each input channel is named in a way that identifies its channel number. In this figure, each input is described as a four-bit array. The select input (s[]) requires two bits to specify the four channel numbers (0–3). A CASE construct is used here to assign an input channel conditionally to the output pins. Line 9, for example, states that in the case when the select inputs (s[]) are set to 0 (that is, binary 00), the

FIGURE 9-62 Four-bit × fourchannel MUX in AHDL.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

```
SUBDESIGN fig9_62
  ch0[3..0], ch1[3..0], ch2[3..0], ch3[3..0]:INPUT;
  s[1..0]
                                               :INPUT; -- select inputs
  dout[3..0]
                                               : OUTPUT;
)
BEGIN
   CASE s[] IS
         WHEN 0 =>
                        dout[] = ch0[];
                        dout[] = ch1[];
         WHEN 1 =>
         WHEN 2 =>
                        dout[] = ch2[];
         WHEN 3 =>
                        dout[] = ch3[];
    END CASE;
END;
```

1

2

3

4

5

6

7

8

9

10

11 12

13

14 15

16 17

18

19

20

21

FIGURE 9-63 Four-bit × four-channel DEMUX in AHDL.

```
SUBDESIGN fig9_63
(
   ch0[3..0], ch1[3..0], ch2[3..0], ch3[3..0]
                                                   : OUTPUT;
   s[1..0]
                                                   : INPUT;
   din[3..0]
                                                   : INPUT;
)
BEGIN
   DEFAULTS
      ch0[] = B''1111'';
      ch1[] = B"1111";
      ch2[] = B"1111";
      ch3[] = B"1111";
   END DEFAULTS;
   CASE S[] IS
         WHEN 0 =>
                         ch0[] = din[];
         WHEN 1 =>
                        ch1[] = din[];
                         ch2[] = din[];
         WHEN 2 =>
                        ch3[] = din[];
         WHEN 3 =>
    END CASE;
END;
```

circuit should connect the channel 0 input to the data output. Notice that when assigning connections, the destination (output) of the signal is on the left of the = sign and the source (input) is on the right.

The demultiplexer code works in a similar way but has only one input channel and four output channels. It must also ensure that the outputs are all HIGH when they are not selected. In Figure 9-63, the inputs and outputs are declared as usual on lines 3–5. The default condition for each channel is specified after the keyword DEFAULTS, which tells the compiler to generate a circuit that will have HIGHs on the outputs unless specifically assigned a value elsewhere in the code. If this default section is not specified, the output values would default automatically to all LOW. Notice on lines 16–19 that the input signal is assigned conditionally to one of the output channels. Consequently, the output channel is on the left of the = sign and the input signal is on the right.

VHDL MUX AND DEMUX

Figure 9-64 shows the code that creates a four-channel MUX with four bits per channel. The inputs are declared as bit arrays on line 3. They could have been declared just as easily as integers ranging from 0 to 15. Whichever way the inputs are declared, the outputs must be of the same type. Notice on line 4 that the select input (*s*) is declared as a decimal integer from 0 to 3 (equivalent to binary 00 to 11). This allows us to refer to it by its decimal channel number in the code, making it easier to understand. Lines 11–15 use the selected signal assignment statement to "connect" the appropriate input to the output, depending on the value on the select inputs. For example, line 15 states that channel 3 should be selected to connect to the data outputs when the select inputs are set to 3.

The demultiplexer code works in a similar way but has only one input channel and four output channels. In Figure 9-65, the inputs and outputs are declared as usual on lines 3–5. Notice that in line 3, the select input(s) is typed as an integer, just like the MUX code in Figure 9-64. The operation of a DEMUX is described most easily using several conditional signal assignment

FIGURE 9-64 Four-bit \times		
four-channel MUX in	1	ENTITY fig9_64 IS
VHDL.	2	PORT (
	3	ch0, ch1, ch2, ch3 :IN BIT_VECTOR (3 DOWNTO 0);
	4	s :IN INTEGER RANGE 0 TO 3;
	5	dout :OUT BIT_VECTOR (3 DOWNTO 0)
	6);
	7	END fig9_64;
	8	<u> </u>
	9	ARCHITECTURE selecter OF fig9_64 IS
	10	-
		BEGIN
	11	WITH S SELECT
	12	dout <= ch0 WHEN 0, switch channel 0 to output
	13	ch1 WHEN 1, switch channel 1 to output
	14	ch2 WHEN 2, switch channel 2 to output
	15	ch3 WHEN 3; switch channel 3 to output
	16	END selecter;
FIGURE 9-65 Four-bit ×		
four-channel DEMUX in	1	ENTITY fig9_65 IS
VHDL.	2	PORT (
	3	s :IN INTEGER RANGE 0 TO 3;
	4	din :IN BIT_VECTOR (3 DOWNTO 0);
	5	ch0, ch1, ch2, ch3 :OUT BIT_VECTOR(3 DOWNTO 0)
	6);
	7	
		END fig9_65;
	8	
	9	ARCHITECTURE selecter OF fig9_65 IS
	10	BEGIN
	11	ch0 <= din WHEN s = 0 ELSE "1111";
	12	ch1 <= din WHEN s = 1 ELSE "1111";
	13	$ch2 \ll din WHEN s = 2 ELSE "1111";$
	14	$ch3 \leq din WHEN s = 3 ELSE "1111";$
	15	END selecter;
	DE sel nal fau be	tements, as shown in lines 11–14. We decided earlier that the code for this MUX must ensure that the outputs are all HIGH when they are not ected. This is accomplished with the ELSE clause of each conditional signassignment. If the ELSE clause is not used, the output values would delt automatically to all LOW. For example, line 13 states that channel 2 will connected to the data inputs whenever the select inputs are set to 2. If <i>s</i> is to any other value, then channel 2 will be forced to have all bits HIGH.
REVIEW QUESTIONS		For the four-bit by four-channel MUX, name the data inputs, the data outputs, and the control inputs that choose one channel of the four. For the four-bit by four-channel DEMUX, name the data inputs, the data

- 2. For the four-bit by four-channel DEMUX, name the data inputs, the data outputs, and the control inputs that choose one channel of the four.
- 3. In the AHDL example, how are the logic states determined for the channels that are not selected?
- 4. In the VHDL example, how are the logic states determined for the channels that are not selected?

9-19 HDL MAGNITUDE COMPARATORS

In Section 9-10, we studied a 7485 magnitude comparator chip. As the name implies, this device compares the magnitude of two binary numbers and indicates the relationship between the two (greater than, less than, equal to). Control inputs are provided for the purpose of cascading these chips. These chips are interconnected so that the chip comparing the lower-order bits has its outputs connected to the control inputs of the next higher-order chip, as shown in Figure 9-37. When the highest-order stage detects that its data inputs have equal magnitude, it will look to the next lower stage and use these control inputs to make the final decision. This gives us a chance to look at one of the defining differences between using traditional logic ICs and using HDL to design a circuit. If we need to compare bigger values using HDL we can simply adjust the size of the comparator input ports to be whatever we need, rather than trying to cascade several four-bit comparators. Consequently, there is no need for cascading input controls in the HDL version.

There are many possible ways to describe the operation of a comparator. However, it is best to use an IF/ELSE construct because each IF clause evaluates a relationship between two values, as opposed to looking for the single value of a variable, like a CASE. The two inputs being compared should definitely be declared as numerical values. The three comparator outputs should be declared as individual bits in order to label each bit's purpose clearly.

AHDL COMPARATOR

The AHDL code in Figure 9-66 follows the algorithm we have described using IF/ELSE constructs. Notice in line 3 that the data values are declared as four-bit numbers. Also note in lines 8, 10, and 11 that several statements can be used to specify the circuit's operation when the IF clause is true. Each statement is used to set the level on one of the outputs. These three statements are considered concurrent, and the order in which they are listed makes no difference. For example, in line 8, when *A* is greater than *B*, the *agtb* output will go HIGH at the same time the other two outputs (*altb, aeqb*) go LOW.

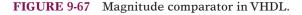
```
1
      SUBDESIGN fiq9 66
 2
 3
         a[3..0], b[3..0]
                               : INPUT;
         aqtb, altb, aeqb
 4
                               : OUTPUT;
 5
      BEGIN
 6
 7
         IF
                a[] > b[] THEN
 8
                                  altb = GND;
                   agtb = VCC;
                                                aeqb = GND;
         ELSIF a[] < b[] THEN
 9
                                  altb = VCC;
10
                   agtb = GND;
                                                aeqb = GND;
11
         ELSE
                   agtb = GND ; altb = GND ; aeqb = VCC;
12
         END IF;
      END;
13
```

FIGURE 9-66 Magnitude comparator in AHDL.

VHDL COMPARATOR

The VHDL code in Figure 9-67 follows the algorithm we have described using IF/ELSE constructs. Notice in line 2 that the data values are declared as four-bit integers. Remember, in VHDL, the IF/ELSE constructs can be used only inside a PROCESS. In this case, we want to evaluate the PROCESS whenever any of the inputs change state. Consequently, each input is listed in the sensitivity list within the parentheses. Also note in lines 10, 11, and 12 that several statements can be used to specify the circuit's operation when the IF clause is true. Each statement is used to set the level on one of the outputs. These three statements are considered concurrent, and the order in which they are listed makes no difference. For example, on line 11, when *A* is greater than *B*, the *agtb* output will go HIGH at the same time the other two outputs (*altb, aeqb*) go LOW.

```
1
      ENTITY fig9_67 IS
2
                                     : IN INTEGER RANGE 0 TO 15;
      PORT ( a, b
                agtb, altb, aeqb : OUT BIT);
3
      END fig9_67;
4
5
 6
      ARCHITECTURE vhdl OF fig9 67 IS
 7
      BEGIN
8
         PROCESS (a, b)
 9
         BEGIN
10
                    a < b THEN
                                     altb <= '1'; agtb <= '0'; aeqb <= '0';
             IF
                                     altb <= '0'; agtb <= '1'; aeqb <= '0';
altb <= '0'; agtb <= '0'; aeqb <= '1';
             ELSIF a > b THEN
11
12
             ELSE
13
             END IF;
14
          END PROCESS;
15
      END vhdl;
```



REVIEW QUESTIONS

- 1. What type of data objects must be declared for data inputs to a comparator?
- 2. What is the key control structure used to describe a comparator?
- 3. What are the key operators used?

9-20 HDL CODE CONVERTERS

Section 9-11 demonstrated some methods using adder circuits in an interesting but not at all intuitive way to create a BCD-to-binary conversion. In Chapter 6, we discussed adder circuits, and the circuit of Figure 9-40 can certainly be implemented using HDL and 7483 macrofunctions or adder descriptions that we know how to write. However, this is an excellent opportunity to point out the tremendous advantage that HDL can offer because it allows a circuit to be described in a way that makes the most sense. In the case of BCD-to-binary conversion, the sensible method of conversion is to use the concepts that we all learned in the third grade about the decimal number system. You were once taught that the number 275 was actually:

	2	\times	100	=	200
+	7	\times	10	=	70
+	5	\times	1	=	5
					275

Now we have studied the BCD number system and realize that 275 is represented in BCD as 0010 0111 0101. Each digit is simply represented in binary. If we could multiply these binary digits by the decimal weight (represented in binary) and add them, we would have a binary answer that is equivalent to the BCD quantity. For example, let's try using the BCD representation for 275:

BCD	Decimal	Partial	
	Weight	Product	
	(in binary)	(in binary)	
0010 $ imes$	1100100 =	11001000	
+ 0111 $ imes$	1010 =	01000110	
+ 0101 \times	1 =	0101	
		100010011	$= 275_{10}$

The solution presented here for our eight-bit (two BCD digits) HDL code converter will use the following strategy:

Take the most significant BCD digit (the tens place) and multiply it by 10. Add this product to the least significant BCD digit (the ones place).

The answer will be a binary number representing the BCD quantity. It is important to realize that the HDL compiler does not necessarily try to implement an actual multiplier circuit in its solution. It will create the most efficient circuit that will do the job, which allows the designer to describe its behavior in the most sensible way.

AHDL BCD-TO-BINARY CODE CONVERTER

The key to this strategy is in being able to multiply by 10. AHDL does not offer a multiplication operator, so in order to use this overall strategy, we need some math tricks. We will use the shifting of bits to perform multiplication and then employ the distributive property from algebra to multiply by 10. In the same way that we can shift a decimal number left by one digit, thus multiplying it by 10, we can likewise shift a binary number one place to the left and multiply it by 2. Shifting two places multiplies a binary number by 4, and shifting three places multiplies by 8. The distributive property tells us that:

 $num \times 10 = num \times (8 + 2) = (num \times 8) + (num \times 2)$

If we can take the BCD tens digit and shift it left three bit positions (i.e., multiply it by 8), then take the same number and shift it left one place (i.e., multiply it by 2), and then add them together, the result will be the same as multiplying the BCD digit by 10. This value is then added to the BCD ones digit to produce the binary equivalent of the two-digit BCD input.

655

The next challenge is to shift the BCD digit left using AHDL. Because AHDL allows us to make up sets of variables, we can shift the bits by appending zeros to the right end of the array. For example, if we have the number 5 in BCD (0101) and we want to shift it three places, we can concatenate the number 0101 with the number 000 in a set, as follows:

(B"0101", B"000") = B"0101000"

The AHDL code in Figure 9-68 begins by declaring inputs for the BCD ones and tens digits. The binary output must be able to represent 99₁₀, which requires seven bits. We also need a variable to hold the product of the BCD digit multiplied by 10. Line 5 declares this variable as a seven-bit number. Line 8 performs the shifting of the *tens[]* array three times and adds it to the *tens[]* array shifted one place to the left. Notice that this latter set must have seven bits in order to be added to the first set, thus the need to concatenate B"00" on the left end. Finally, in line 10, the result from line 8 is added to the BCD ones digit with leading zero extensions (to make seven bits) to form the binary output.

```
SUBDESIGN fig9 68
1
2
      ( ones[3..0], tens[3..0]
                                    : INPUT;
3
         binary[6..0]
                                    :OUTPUT;
                                              )
 4
 5
      VARIABLE times10[6..0]
                                    :NODE;
                                               % variable for tens digit times 10%
 6
7
      BEGIN
8
         times10[] = (tens[],B"000") + (B"00",tens[],B"0");
 9
               % shift left 3X (times 8) + shift left 1X (times 2) %
10
         binary[] = times10[] + (B"000", ones[]);
               % tens digit times 10 + ones digit %
11
12
      END;
```

FIGURE 9-68 BCD-to-binary code converter in AHDL.

VHDL BCD-TO-BINARY CODE CONVERTER

The VHDL solution in Figure 9-69 is very simple due to the powerful math operations available in the language. The inputs and outputs must be declared as integers because we intend to perform arithmetic operations on them. Notice that the range is specified based on the largest valid BCD number using two digits. In line 9, the tens digit is multiplied by ten, and in line 10, the ones digit is added to form the binary equivalent of the BCD input.

```
FIGURE 9-69 BCD-to-
binary code converter in
                              ENTITY fig9_69 IS
                         1
VHDL.
                         2
                              PORT (
                                        ones, tens : IN INTEGER RANGE 0 TO 9;
                         3
                                        binary
                                                     :OUT INTEGER RANGE 0 TO 99);
                         4
                              END fig9_69;
                         5
                              ARCHITECTURE vhdl OF fig9_69 IS
                         6
                         7
                              SIGNAL times10 :INTEGER RANGE 0 TO 90;
                         8
                              BEGIN
                         9
                                  times10 <= tens * 10;
                        10
                                 binary <= times10 + ones;</pre>
                        11
                              END vhdl;
```

REVIEW QUESTIONS

- 1. For a two-digit BCD (eight-bit) number, what is the decimal weight of the most significant digit?
- 2. In AHDL, how is multiplication by 10 accomplished?
- 3. In VHDL, how is multiplication by 10 accomplished?

SUMMARY

- 1. A decoder is a device whose output is activated only when a unique binary combination (code) is presented on its inputs. Many MSI decoders have several outputs, each one corresponding to only one of the many possible input combinations.
- 2. Digital systems often need to display decimal numbers. This is done using 7-segment displays that are driven by special chips that decode the binary number and translate it into segment patterns that represent decimal numbers to people. The segment elements can be light-emitting diodes, liquid crystals, or glowing electrodes surrounded by neon gas.
- 3. Graphical LCDs use a matrix of picture elements called pixels to create an image on a large screen. Each pixel is controlled by activating the row and column that have that pixel in common. The brightness level of each pixel is stored as a binary number in the video memory. A fairly complex digital circuit must scan through the video memory and all the row/column combinations, controlling the amount of light that can pass through each pixel.
- 4. An encoder is a device that generates a unique binary code in response to the activation of each individual input.
- 5. Troubleshooting a digital system involves *observation/analysis* to identify the possible causes, and a process of elimination called *divide-andconquer* to isolate and identify the cause.
- 6. Multiplexers act like digitally controlled switches that select and connect one logic input at a time to the output pin. By taking turns, many different data signals can share the same data path using multiplexers. Demultiplexers are used at the other end of the data path to separate the signals that are sharing a data path and distribute them to their respective destinations.
- 7. Magnitude comparators serve as an indicator of the relationship between two binary numbers, with outputs that show >, <, and =.
- 8. It is often necessary to translate between and among various methods of representing quantities with binary numbers. Code converters are devices that take in one form of binary representation and convert it to another form.
- 9. In digital systems, many devices must often share the same data path. This data path is often called a *data bus*. Even though many devices can be "riding" on the bus, there can be only one bus "driver" at any one time. Thus, devices must take turns applying logic signals to the data bus.
- 10. In order to take turns, the devices must have *tristate outputs* that can be disabled when another device is driving the bus. In the disabled state, the device's output is essentially electrically disconnected from the bus by going into a state that offers a high-impedance path to both ground and the positive power supply. Devices designed to interface to a bus have outputs that can be HIGH, LOW, or disabled (high impedance).

- 11. PLDs offer an alternative to the use of MSI circuits to implement digital systems. Boolean equations can be used to describe the operation of these circuits, but HDLs also offer high-level language constructs.
- 12. HDL macrofunctions are available for many MSI standard parts described in this chapter.
- 13. Custom code can be written in HDL to describe each of the common logic functions presented in this chapter.
- 14. Priority and precedence can be established in AHDL using don't-care entries in truth tables and using IF/ELSE decisions. Priority and precedence can be established in VHDL using conditional signal assignments or using a PROCESS containing IF/ELSE or CASE decisions.
- 15. Tristate outputs can be created in HDL. AHDL uses :TRI primitives that drive the outputs. VHDL assigns Z (high impedance) as a valid state for STD_LOGIC outputs.
- 16. The DEFAULTS statement in AHDL can be used to define the proper level for outputs that are not explicitly defined in the code.
- 17. The ELSE clause in the conditional signal assignment statement of VHDL can be used to define the default state of an output.

IMPORTANT TERMS

encoding magnitude comparator decoder **BCD-to-decimal** encoder data bus decoder priority encoder floating bus driver observation/analysis word BCD-to-7-segment divide-and-conquer bus driver multiplexer (MUX) bidirectional data decoder/driver common anode multiplexing lines common cathode parallel-to-serial DEFAULTS LCD conversion conditional signal backplane demultiplexer assignment pixel (DEMUX) statement

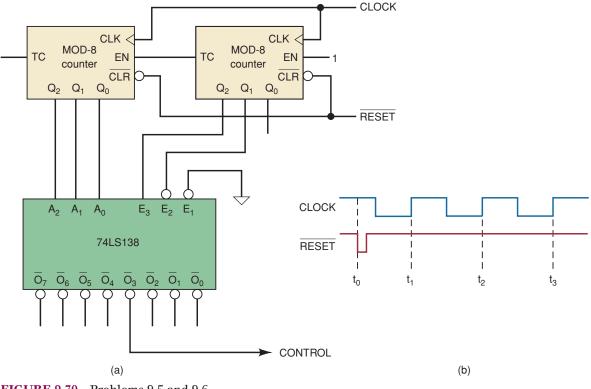
PROBLEMS

SECTION 9-1

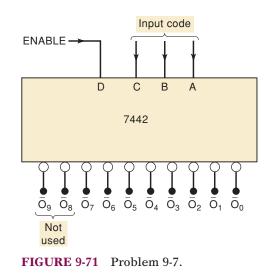
- B 9-1. Refer to Figure 9-3. Determine the levels at each decoder output for the following sets of input conditions.
 - (a)*All inputs LOW
 - (b)*All inputs LOW except E_3 = HIGH
 - (c) All inputs HIGH except $\overline{E}_1 = \overline{E}_2 = \text{LOW}$
 - (d) All inputs HIGH
- **B** 9-2.*What is the number of inputs and outputs of a decoder that accepts 64 different input combinations?

^{*}Answers to problems marked with an asterisk can be found in the back of the text.

- **B** 9-3. For a 74ALS138, what input conditions will produce the following outputs:
 - (a)*LOW at \overline{O}_6
 - (b)*LOW at \overline{O}_3
 - (c) LOW at \overline{O}_5
 - (d) LOW at \overline{O}_0 and \overline{O}_7 , simultaneously
- **D** 9-4. Show how to use 74LS138s to form a 1-of-16 decoder.
 - 9-5.*Figure 9-70 shows how a decoder can be used in the generation of control signals. Assume that a RESET pulse has occurred at time t_0 , and determine the CONTROL waveform for 32 clock pulses.



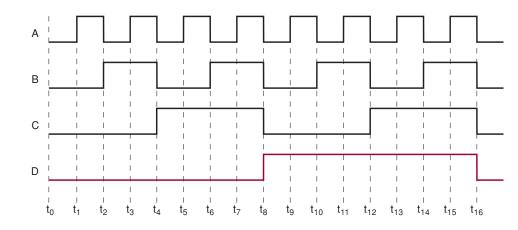
- FIGURE 9-70 Problems 9-5 and 9-6.
 - **D** 9-6. Modify the circuit of Figure 9-70 to generate a CONTROL waveform that goes LOW from t_{20} to t_{24} . (*Hint:* The modification does not require additional logic.)
 - 9-7.*The 7442 decoder of Figure 9-5 does not have an ENABLE input. However, we can operate it as a 1-of-8 decoder by not using outputs \overline{O}_8 and \overline{O}_9 and by using the *D* input as an ENABLE. This is illustrated in Figure 9-71. Describe how this arrangement works as an enabled 1-of-8 decoder, and state how the level on *D* either enables or disables the outputs.



9-8. Consider the waveforms in Figure 9-72. Apply these signals to the 74LS138 as follows:

 $A \rightarrow A_0$ $B \rightarrow A_1$ $C \rightarrow A_2$ $D \rightarrow E_3$

Assume that \overline{E}_1 and \overline{E}_2 are tied LOW, and draw the waveforms for outputs \overline{O}_0 , \overline{O}_3 , \overline{O}_6 , and \overline{O}_7 .



D 9-9. Modify the circuit of Figure 9-6 so that relay K_1 stays energized from PGT 3 to 5 and K_2 stays energized from PGT 6 to 9. (*Hint:* This modification requires no additional circuitry.)

SECTIONS 9-2 AND 9-3

B, **D** 9-10.*Show how to connect BCD-to-7-segment decoder/drivers and LED 7-segment displays to the counter circuit of Figure 7-22. Assume that each segment is to operate at approximately 10 mA at 2.5 V.



- **B** 9-11. (a) Refer to Figure 9-10 and draw the segment and backplane waveforms relative to ground for CONTROL = 0. Then draw the waveform of segment voltage relative to backplane voltage.
 - (b) Repeat part (a) for CONTROL = 1.
- **C, D** 9-12.* The BCD-to-7-segment decoder/driver of Figure 9-8 contains the logic for activating each segment for the appropriate BCD inputs. Design the logic for activating the *g* segment.

SECTION 9-4

B 9-13.*DRILL QUESTION

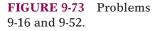
For each item, indicate whether it is referring to a decoder or an encoder.

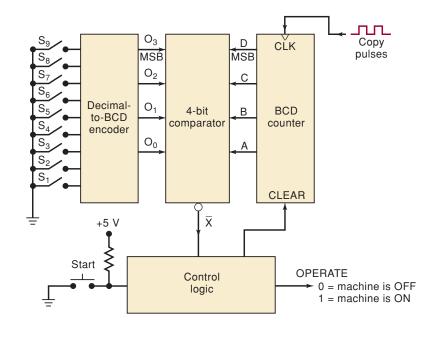
- (a) Has more inputs than outputs.
- (b) Is used to convert key actuations to a binary code.
- (c) Only one output can be activated at one time.
- (d) Can be used to interface a BCD input to an LED display.
- (e) Often has driver-type outputs to handle large *I* and *V*.
- 9-14. Determine the output levels for the 74147 encoder when $\overline{A}_8 = \overline{A}_4 = 0$ and all other inputs are HIGH.
- 9-15. Apply the signals of Figure 9-72 to the inputs of a 74147 as follows:

$$A \to \overline{A}_7 \qquad B \to \overline{A}_4 \qquad C \to \overline{A}_2 \qquad D \to \overline{A}_1$$

Draw the waveforms for the encoder's outputs.

C, **D** 9-16. Figure 9-73 shows the block diagram of a logic circuit used to control the number of copies made by a copy machine. The machine operator selects the number of desired copies by closing one of the selector switches S_1 to S_9 . This number is encoded in BCD by the encoder and is sent to a comparator circuit. The operator then hits a momentary-contact START switch, which clears the counter and initiates a HIGH





OPERATE output that is sent to the machine to signal it to make copies. As the machine makes each copy, a copy pulse is generated and fed to the BCD counter. The counter outputs are continually compared with the switch encoder outputs by the comparator. When the two BCD numbers match, indicating that the desired number of copies has been made, the comparator output \overline{X} goes LOW; this causes the OPERATE level to return LOW and stop the machine so that no more copies are made. Activating the START switch will cause this process to be repeated. Design the complete logic circuitry for the comparator and control sections of this system.

C, D 9-17.*The keyboard circuit of Figure 9-16 is designed to accept a three-digit decimal number. What would happen if *four* digit keys were activated (e.g., 3095)? Design the necessary logic to be added to this circuit so that after three digits have been entered, any additional digits will be ignored until the CLEAR key is depressed. In other words, if 3095 is entered on the keyboard, the output registers will display 309 and will ignore the 5 and any subsequent digits until the circuit is cleared.

SECTION 9-5

Т

- T 9-18.*A technician breadboards the keyboard entry circuit of Figure 9-16 and tests its operation by trying to enter a series of three-digit numbers. He finds that sometimes the digit 0 is entered instead of the digit he pressed. He also observes that it happens with all of the keys more or less randomly, although it is worse for some keys than others. He replaces all of the ICs, and the malfunction persists. Which of the following circuit faults would explain his observations? Explain each choice.
 - (a) The technician forgot to ground the unused inputs of the OR gate.
 - (b) He has mistakenly used \overline{Q} instead of Q from the one-shot.
 - (c) The switch bounce from the digit keys lasts longer than 20 ms.
 - (d) The Y and Z outputs are shorted together.
- T 9-19. Repeat Problem 9-18 with the following symptom: the registers and displays stay at 0 no matter how many times a key is pressed.
 - 9-20.* While testing the circuit of Figure 9-16, a technician finds that every odd-numbered key results in the correct digit being entered, but every even-numbered key results in the wrong digit being entered as follows: the 0 key causes a 1 to be entered, the 2 key causes a 3 to be entered, the 4 key causes a 5 to be entered, and so on. Consider each of the following faults as possible causes of the malfunction. For each one, explain why it can or cannot be the actual cause.
 - (a) There is an open connection from the output of the LSB inverter to the *D* inputs of the FFs.
 - (b) The *D* input of flip-flop Q_8 is internally shorted to V_{CC} .
 - (c) A solder bridge is shorting \overline{O}_0 to ground.
- **T** 9-21.* A technician tests the circuit of Figure 9-4 as described in Example 9-7, and she obtains the following results: all of the outputs work except \overline{O}_{16} to \overline{O}_{19} and \overline{O}_{24} to \overline{O}_{27} , which are permanently HIGH. What is the most probable circuit fault?
- T 9-22. A technician tests the circuit of Figure 9-4 as described in Example 9-7 and finds that the correct output is activated for each possible input

TABLE	9-8
-------	-----

Input Code						
A 4	A 3	A ₂	A ₁	A ₀	Activated Outputs	
1	0	0	0	0	\overline{O}_{16} and \overline{O}_{24}	
1	0	0	0	1	\overline{O}_{17} and \overline{O}_{25}	
1	0	0	1	0	\overline{O}_{18} and \overline{O}_{26}	
1	0	0	1	1	\overline{O}_{19} and \overline{O}_{27}	
1	0	1	0	0	$\overline{\mathcal{O}}_{20}$ and $\overline{\mathcal{O}}_{28}$	
1	0	1	0	1	\overline{O}_{21} and \overline{O}_{29}	
1	0	1	1	0	\overline{O}_{22} and \overline{O}_{30}	
1	0	1	1	1	\overline{O}_{23} and \overline{O}_{31}	

code except those listed in Table 9-8. Examine this table and determine the probable cause of the malfunction.

- **T** 9-23.*Suppose that a 22- Ω resistor was mistakenly used for the *g* segment in Figure 9-8. How would this affect the display? What possible problems could occur?
- T 9-24. Repeat Example 9-8 with the observed sequence shown below:

COUNT	0	1	2	3	4	5	6	7	8	9
Observed display		1		ורו		7	Ĺ		Ц	

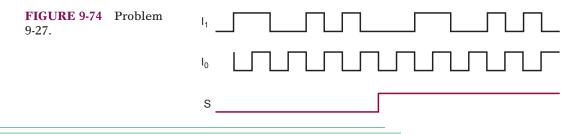
T 9-25.*Repeat Example 9-8 with the observed sequence shown below:

COUNT	0	1	2	3	4	5	6	7	8	9
Observed display		7								Ģ

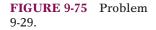
T 9-26.*To test the circuit of Figure 9-11, a technician connects a BCD counter to the 74HC4511 inputs and pulses the counter at a very slow rate. She notices that the *f* segment works erratically, and no particular pattern is evident. What are some of the possible causes of the malfunction? (*Hint:* Remember, the ICs are CMOS.)

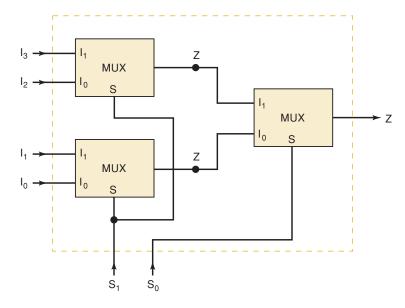
SECTIONS 9-6 AND 9-7

B 9-27. The timing diagram in Figure 9-74 is applied to Figure 9-19. Draw the output waveform Z.

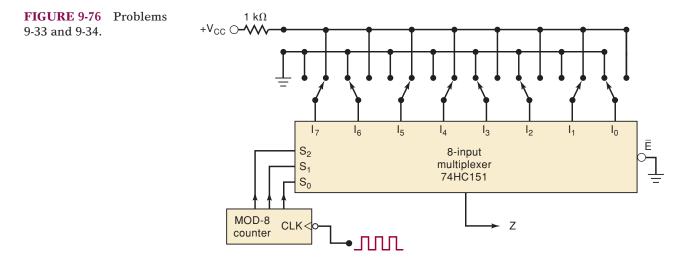


- 9-28. Figure 7-68 shows an eight-bit shift register that could be used to delay a signal by 1 to 8 clock periods. Show how to wire a 74151 to this shift register in order to select the desired Q output and indicate the logic level necessary on the select inputs to provide a delay of $6 \times T_{clk}$.
- 9-29.*The circuit in Figure 9-75 uses three two-input multiplexers (Figure 9-19). Determine the function performed by this circuit.





- **D** 9-30. Use the idea from Problem 9-29 to arrange several 74151 1-of-8 multiplexers to form a 1-of-64 multiplexer.
- **C**, **D** 9-31.*Show how two 74157s and a 74151 can be arranged to form a 1-of-16 multiplexer with no other required logic. Label the inputs I_0 to I_{15} to show how they correspond to the select code.
 - D 9-32. (a) Expand the circuit of Figure 9-24 to display the contents of two three-stage BCD counters.
 - (b)*Count the number of connections in this circuit, and compare it with the number required if a separate decoder/driver and display were used for each stage of each counter.
 - 9-33.*Figure 9-76 shows how a multiplexer can be used to generate logic waveforms with any desirable pattern. The pattern is programmed



using eight SPDT switches, and the waveform is repetitively produced by pulsing the MOD-8 counter. Draw the waveform at Z for the given switch positions.

- 9-34. Change the MOD-8 counter in Figure 9-76 to a MOD-16 counter, and connect the MSB to the multiplexer \overline{E} input. Draw the Z waveform.
- **D** 9-35.*Show how a 74151 can be used to generate the logic function Z = AB + BC + AC.
- **D** 9-36. Show how a 16-input multiplexer such as the 74150 is used to generate the function $Z = \overline{AB}\overline{CD} + BCD + A\overline{B}\overline{D} + AB\overline{CD}$.
- **N** 9-37.* The circuit of Figure 9-77 shows how an eight-input MUX can be used to generate a four-variable logic function, even though the MUX has only three SELECT inputs. Three of the logic variables *A*, *B*, and *C* are connected to the SELECT inputs. The fourth variable *D* and its inverse \overline{D} are connected to selected data inputs of the MUX as required by the desired logic function. The other MUX data inputs are tied to a LOW or a HIGH as required by the function.

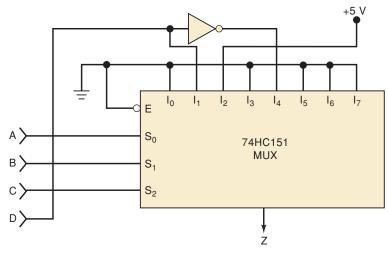


FIGURE 9-77 Problems 9-37 and 9-38.

- (a) Set up a truth table showing the output *Z* for the 16 possible combinations of input variables.
- (b) Write the sum-of-products expression for Z and simplify it to verify that

$$Z = \overline{C}B\overline{A} + D\overline{C}\overline{B}A + \overline{D}C\overline{B}\overline{A}$$

- 9-38. The hardware method used in Figure 9-77 can be used to generate any four-variable logic function. For example, $Z = \overline{DBCA} + \overline{CBA} + D\overline{CBA} + CB\overline{A}$ is implemented by following these steps:
 - 1. Set up a truth table in two halves, side by side as shown in Table 9-9. Notice that the left half shows all combinations of *CBA* when D = 0, and the right half shows all combinations of *CBA* when D = 1.
 - 2. Write the value of Z for each four-bit combination when D = 0 and also when D = 1.
 - 3. Make a column on the right side as shown, which describes what must be connected to each of the eight MUX inputs I_n .

TABLE 9-9		D = 0		<i>D</i> = 1	
	DCBA	Ζ	DCBA	Ζ	I _n
	0000	0	1000	0	$I_0 = 0$
	0001	1	1001	0	$I_1 = \overline{D}$
	0010	0	<mark>1</mark> 010	0	$I_2 = 0$
	0011	1	<mark>1</mark> 011	1	$I_3 = 1$
	0100	0	1100	0	$I_4 = 0$
	0101	0	<mark>1</mark> 101	1	$I_5 = D$
	0110	1	<mark>1</mark> 110	1	$I_6 = 1$
	0111	0	<mark>1</mark> 111	0	$I_7 = 0$

4. For each line of this table, compare the value for Z when D = 0 with the value for Z when D = 1. Enter the appropriate information for I_n as follows:

When Z = 0 regardless of whether D = 0 or 1, THEN $I_n = 0$ (GND).

When Z = 1 regardless of whether D = 0 or 1, THEN $I_n = 1$ (VCC).

When Z = 0 when D = 0 AND Z = 1 when D = 1, THEN $I_n = D$.

When Z = 1 when D = 0 AND Z = 0 when D = 1, THEN $I_n = \overline{D}$.

- (a) Verify the design of Figure 9-77 using this method.
- (b) Use this method to implement a function that will produce a HIGH only when the four input variables are at the same level or when the *B* and *C* variables are at different levels.

SECTION 9-8

B 9-39.***DRILL QUESTION**

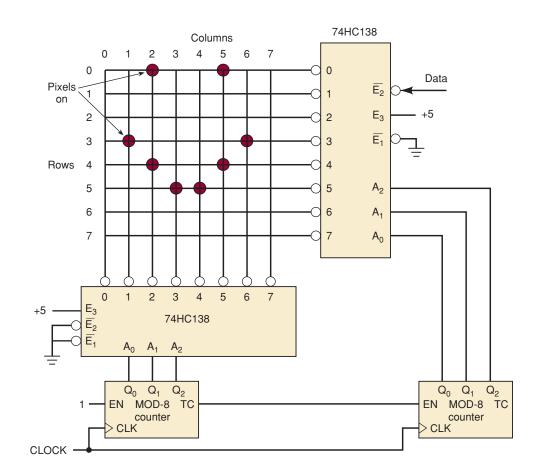
For each item, indicate whether it is referring to a decoder, an encoder, a MUX, or a DEMUX.

- (a) Has more inputs than outputs.
- (b) Uses SELECT inputs.
- (c) Can be used in parallel-to-serial conversion.
- (d) Produces a binary code at its output.
- (e) Only one of its outputs can be active at one time.
- (f) Can be used to route an input signal to one of several possible outputs.
- (g) Can be used to generate arbitrary logic functions.
- 9-40. Show how the 7442 decoder can be used as 1-to-8 demultiplexer. (*Hint:* See Problem 9-7.)
- 9-41.*Apply the waveforms of Figure 9-72 to the inputs of the 74LS138 DEMUX of Figure 9-30(a) as follows:

$$D \to A_2$$
 $C \to A_1$ $B \to A_0$ $A \to \overline{E}_1$

Draw the waveforms at the DEMUX outputs.

- 9-42. Consider the system of Figure 9-31. Assume that the clock frequency is 10 pps. Describe what the monitoring panel indications will be for each of the following cases.
 - (a) All doors closed
 - (b) All doors open
 - (c) Doors 2 and 6 open
- C, D 9-43.* Modify the system of Figure 9-31 to handle 16 doors. Use a 74150 16input MUX and two 74LS138 DEMUXes. How many lines are going to the remote monitoring panel?
 - 9-44. Draw the waveforms at transmit_data, and DEMUX outputs O_0, O_1, O_2 , and O_3 in Figure 9-33 for the following register data loaded into the transmit registers in Figure 9-32: [A] = 0011, [B] = 0110, [C] = 1001, [D] = 0111.
 - 9-45. Figure 9-78 shows an 8×8 graphic LCD display grid controlled by a 74HC138 configured as a decoder, and a 74HC138 configured as a demultiplexer. Draw 48 cycles of the clock and the data input necessary to activate the pixels shown on the display.



SECTION 9-9

- T 9-46. Consider the control sequencer of Figure 9-26. Describe how each of the following faults will affect the operation.
 - (a)*The I_3 input of the MUX is shorted to ground.
 - (b) The connections from sensors 3 and 4 to the MUX are reversed.



TABLE 9-10			Actual Count	Displayed Count
	Case 1	Counter 1	33	33
		Counter 2	47	47
	Case 2	Counter 1	82	02
		Counter 2	64	64
	Case 3	Counter 1	63	63
		Counter 2	95	15

- Т 9-47.* Consider the circuit of Figure 9-24. A test of the circuit yields the results shown in Table 9-10. What are the possible causes of the malfunction?
- Т 9-48.* A test of the security monitoring system of Figure 9-31 produces the results recorded in Table 9-11. What are the possible faults that could cause this operation?

TADLE 0 11		
TABLE 9-11	Condition	LEDs
	All doors closed	All LEDs off
	Door 0 open	LED 0 flashing
	Door 1 open	LED 2 flashing
	Door 2 open	LED 1 flashing
	Door 3 open	LED 3 flashing
	Door 4 open	LED 4 flashing
	Door 5 open	LED 6 flashing
	Door 6 open	LED 5 flashing
	Door 7 open	LED 7 flashing

Т 9-49.* A test of the security monitoring system of Figure 9-31 produces the results recorded in Table 9-12. What are the possible faults that could cause this operation? How can this be verified or eliminated as a fault?

E E O 13		
LE 9-12	Condition	LEDs
	All doors closed	All LEDs off
	Door 0 open	LED 0 flashing
	Door 1 open	LED 1 flashing
	Door 2 open	LED 2 flashing
	Door 3 open	LED 3 flashing
	Door 4 open	LED 4 flashing
	Door 5 open	LED 5 flashing
	Door 6 open	No LED flashing
	Door 7 open	No LED flashing
	Doors 6 and 7 open	LEDs 6 and 7 flashing

9-50.* The synchronous data transmission system of Figure 9-32 and Figure 9-33 Т is malfunctioning. An oscilloscope is used to monitor the MUX and

TAB

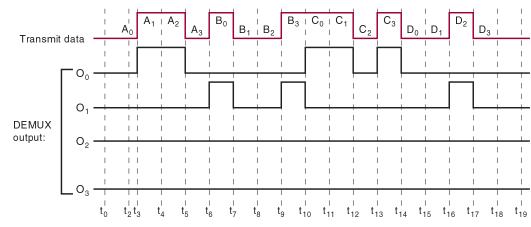


FIGURE 9-79 Problem 9-50.

DEMUX outputs during the transmission cycle, with the results shown in Figure 9-79. What are the possible causes of the malfunction?

T 9-51. The synchronous data transmission system of Figures 9-32 and 9-33 is not working properly and the troubleshooting tree diagram of Figure 9-35 has been used to isolate the problem to the timing and control section of the receiver. Draw a troubleshooting tree diagram that will isolate the problem further to one of the four blocks in that section (*FF*1, Bit counter, Word counter, or *FF*2). Assume that all wires are connected as shown, with no wiring errors.

SECTION 9-10

- C, D 9-52. Redesign the circuit of Problem 9-16 using a 74HC85 magnitude comparator. Add a "copy overflow" feature that will activate an ALARM output if the OPERATE output fails to stop the machine when the requested number of copies is done.
 - **D** 9-53.*Show how to connect 74HC85s to compare two 10-bit numbers.

SECTION 9-11

- 9-54. Assume a BCD input of 69 to the code converter of Figure 9-40. Determine the levels at each Σ output and at the final binary output.
- T 9-55.* A technician tests the code converter of Figure 9-40 and observes the following results:

BCD Input	Binary Output
52	0110011
95	1100000
27	0011011

What is the probable circuit fault?

SECTIONS 9-12 TO 9-14

B 9-56. **DRILL QUESTION**

True or false:

- (a) A device connected to a data bus should have tristate outputs.
- (b) Bus contention occurs when more than one device takes data from the bus.

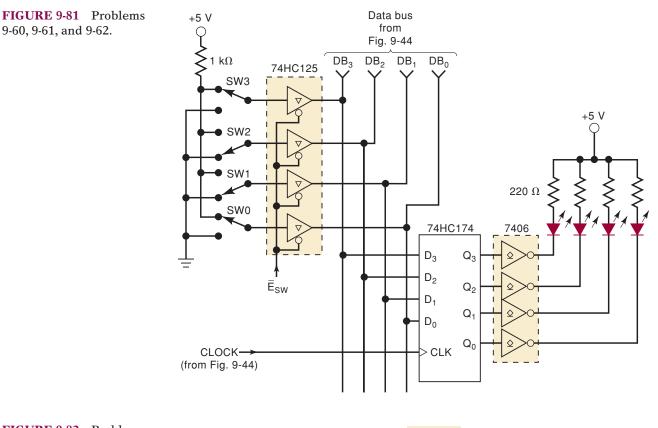
OE₄

- (c) Larger units of data can be transferred over an eight-line data bus than over a four-line data bus.
- (d) A bus driver IC generally has a high output impedance.
- (e) Bidirectional registers and buffers have common I/O lines.
- 9-57.*For the bus arrangement of Figure 9-44, describe the input signal requirements for simultaneously transferring the contents of register *C* to both of the other registers.
- 9-58. Assume that the registers in Figure 9-44 are initially [A] = 1011, [B] = 1000, and [C] = 0111. The signals in Figure 9-80 are applied to the register inputs.
 - (a) Determine the contents of each register at times t_1 , t_2 , t_3 , and t_4 .
 - (b) Describe what would happen if \overline{IE}_A were LOW when the third clock pulse occurred.

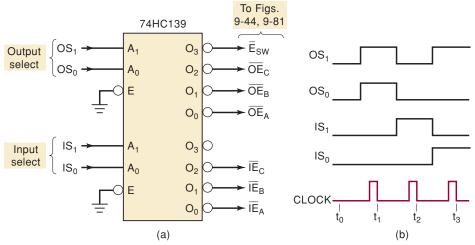
 $\begin{array}{c} \overline{OE}_{B} \\ \overline{OE}_{C} & 1 \\ \hline \overline{IE}_{A} & 1 \\ \hline \overline{IE}_{B} & 1 \\ \hline \overline{IE}_{C} & 1 \\ \hline \overline{IE}_{L} & 1 \\ \hline \overline{IE}_{C} & 1 \\ \hline \overline{IE}_{L} &$



- 9-59. Assume the same initial conditions of Problem 9-58, and sketch the signal on DB_3 for the waveforms of Figure 9-80.
- 9-60. Figure 9-81 shows two more devices that are to be added to the data bus of Figure 9-44. One is a set of buffered switches that can be used to enter data manually into any of the bus registers. The other device is an output register that is used to latch any data that are on the bus during a data transfer operation and display them on a set of LEDs.
 - (a) Assume that all registers contain 0000. Outline the sequence of operations needed to load the registers with the following data from the switches: [A] = 1011, [B] = 0001, [C] = 1110.
 - (b) What will the state of the LEDs be at the end of this sequence?
- C 9-61. Now that the circuitry of Figure 9-81 has been added to Figure 9-44, a total of five devices are connected to the data bus. The circuit in Figure 9-82(a) will now be used to generate the enable signals needed to perform the different data transfers over the data bus. It uses a 74HC139 chip that contains two identical independent 1-of-4 decoders with an active-LOW enable. The top decoder is used to select the device that will put data on the data bus (output select), and the bottom decoder is used to select the device that is to take the data from the data bus (input select). Assume that the decoder outputs are connected to the corresponding enable inputs of the devices tied to the data bus. Also assume that all registers initially







contain 0000 at time t_0 , and the switches are in the positions shown in Figure 9-81.

- (a)*Determine the contents of each register at times t_1 , t_2 , and t_3 in response to the waveforms in Figure 9-82(b).
- (b) Can bus contention ever occur with this circuit? Explain.
- 9-62. Show how a 74HC541 (Figure 9-47) can be used in the circuit of Figure 9-81.

MICROCOMPUTER APPLICATIONS

C, N 9-63.*Figure 9-83 shows the basic circuitry to interface a microprocessor (MPU) to a memory module. The memory module will contain one or

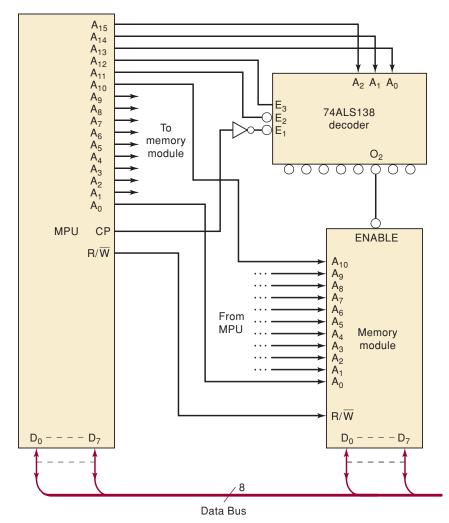


FIGURE 9-83 Basic microprocessor-to-memory interface circuit for Problem 9-63.

more memory ICs (Chapter 12) that can either receive data from the MPU (a WRITE operation) or send data to the MPU (a READ operation). The data are transferred over the eight-line data bus. The MPU's data lines and the memory's I/O data lines are connected to this common bus. For now we will be concerned with how the MPU controls the selection of the memory module for a READ or WRITE operation. The steps involved are as follows:

- 1. The MPU places the memory address on its address output lines A_{15} to A_0 .
- 2. The MPU generates the R/\overline{W} signal to inform the memory module which operation is to be performed: $R/\overline{W} = 1$ for READ, $R/\overline{W} = 0$ for WRITE.
- 3. The upper five bits of the MPU address lines are decoded by the 74ALS138, which controls the ENABLE input of the memory module. This ENABLE input must be active in order for the memory module to do a READ or WRITE operation.
- 4. The other 11 address bits are connected to the memory module, which uses them to select the specific *internal* memory location being accessed by the MPU, provided that ENABLE is active.

In order to read from or write into the memory module, the MPU must put the correct address on the address lines to enable the memory, and then pulse *CP* to the HIGH state.

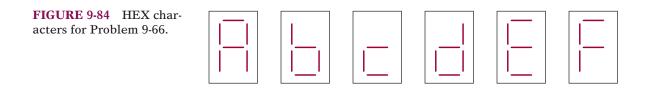
- (a) Determine which, if any, of these hexadecimal addresses will activate the memory module: 607F, 57FA, 5F00.
- (b) Determine what range of hex addresses will activate the memory. (*Hint:* Inputs A_0 to A_{10} to memory can be any combination.)
- (c) Assume that a second identical memory module is added to the circuit with its address, R/\overline{W} , and data I/O lines connected exactly the same as the first module *except* that its ENABLE input is tied to decoder output \overline{O}_4 . What range of hex addresses will activate this second module?
- (d) Is it possible for the MPU to read from or write to both modules at the same time? Explain.

DESIGN PROBLEM

- C, D 9-64. The keyboard entry circuit of Figure 9-16 is to be used as part of an electronic digital lock that operates as follows: when activated, an UNLOCK output goes HIGH. This HIGH is used to energize a solenoid that retracts a bolt and allows a door to be opened. To activate UNLOCK, the operator must press the CLEAR key and then enter the correct three-key sequence.
 - (a) Show how 74HC85 comparators and any other needed logic can be added to the keyboard entry circuit to produce the digital lock operation described above for a key sequence of CLEAR-3-5-8.
 - (b) Modify the circuit to activate an ALARM output if the operator enters something other than the correct three-key sequence.

SECTIONS 9-15 TO 9-20

- H, D, N 9-65.*Write the HDL code for a BCD-to-decimal decoder (the equivalent of a 7442).
- H, D, N 9-66. Write the HDL code for a HEX decoder/driver for a 7-segment display. The first 10 characters should appear as shown in Figure 9-7. The last six characters should appear as shown in Figure 9-84.



- **B**, **H**, **N** 9-67. Write a low-priority ENCODER description that will always encode the lowest number if two inputs are activated simultaneously.
 - H, N 9-68. Rewrite the code of the four-bit comparator of Figures 9-66 or 9-67 to make an eight-bit comparator without using macrofunctions.
 - H, N 9-69. Use HDL to describe a four-bit binary number to a two-digit BCD code converter.
 - H, N 9-70. Use HDL to describe a three-digit BCD code to eight-bit binary number converter. (Maximum BCD input is 255.)

ANSWERS TO SECTION REVIEW QUESTIONS

SECTION 9-1

1. No2. The enable input controls whether or not the decoder logic responds to
the input binary code.3. The 7445 has open-collector outputs that can handle up
to 30 V and 80 mA.4. 24 pins: 2 enables, 4 inputs, 16 outputs, V_{CC} , and ground

SECTION 9-2

1. *a*, *b*, *c*, *f*, *g* 2. True

SECTION 9-3

1. LEDs: (a), (e), (f). LCDs: (b), (c), (d), (e) 2. (a) four-bit BCD, (b) seven- or eightbit ASCII, (c) binary value for pixel intensity

SECTION 9-4

1. An encoder produces an output code corresponding to the activated input. A decoder activates one output corresponding to an applied input code. 2. In a priority encoder, the output code corresponds to the *highest*-numbered input that is activated. 3. Normal BCD = 0110 4. (a) produces a PGT when a key is pressed, (b) converts key actuation to its BCD code, (c) generates bounce-free pulse to trigger the ring counter, (d) form a ring counter that sequentially clocks output registers, (e) store BCD codes generated by key actuations 5. E_1 and E_0 are used for cascading and GS indicates an active input.

SECTION 9-6

 The binary number at the select inputs determines which data input will pass through to the output.
 Thirty-two data inputs and five select inputs

SECTION 9-7

1. Parallel-to-serial conversion, data routing, logic-function generation, operations sequencing 2. False; they are applied to the select inputs. 3. Counter

SECTION 9-8

A MUX selects one of many input signals to be passed to its output; a DEMUX selects one of many outputs to receive the input signal.
 True, provided that the decoder has an ENABLE input
 The LEDs will go on and off in sequence.

SECTION 9-10

1. To provide a means for expanding the compare operations to numbers with more than four bits. 2. $O_{A=B} = 1$; other outputs are 0.

SECTION 9-11

 A code converter takes input data represented in one type of binary code and converts it to another type of binary code.
 Three digits can represent decimal values up to 999. To represent 999 in straight binary requires 10 bits.

SECTION 9-12

1. A set of connecting lines to which the inputs and outputs of many different devices can be connected 2. Bus contention occurs when the outputs of more than one device connected to a bus are enabled at the same time. It is prevented by controlling the device enable inputs so that this cannot happen. 3. A condition in which all devices connected to a bus are in the Hi-Z state

SECTION 9-13

1. 1011 2. True 3. 0000

SECTION 9-14

1. Bus contention 2. Floating, Hi-Z 3. Provides tristate low-impedance outputs 4. Reduces the number of IC pins and the number of connections to the data bus 5. See Figure 9-51.

SECTION 9-15

1. They are enable inputs. All must be active for the decoder to work.2. The CASEand the TABLE3. The selected signal assignment statement and the CASE

SECTION 9-16

1. The combination input/output pin BI/RBO 2. Common anode. The outputs connect to the cathodes and go LOW to light the segments. 3. The IF/ELSE control structure is evaluated sequentially and gives precedence in the order in which decisions are listed.

SECTION 9-17

1. The don't-care entry in a truth table and the IF/ELSE control structure2. TheIF/ELSE control structure and the conditional signal assignment statement3. Byuse of the :TRI primitive and assigning a value to OE4. By use of the IEEESTD_LOGIC data type that has a possible value of Z

SECTION 9-18

1. Inputs: *ch0*, *ch1*, *ch2*, *ch3*; output: *dout*; control inputs (Select): s 2. Input *din*; outputs: *ch0*, *ch1*, *ch2*, *ch3*; control inputs (Select): *s* 3. DEFAULTS 4. ELSE

SECTION 9-19

1. numerical data objects (e.g., INTEGER in VHDL) 2. IF/ ELSE 3. Relational operators (<, >)

SECTION 9-20

1. 10 2. By multiplying by 8 + 2. Shifting the BCD digit three places left multiplies by 8, and shifting the same BCD digit one place left multiplies by 2. Adding these results produced the BCD digit multiplied by 10. 3. VHDL simply uses the * operator to multiply.