We now recognize that the expression for S describes the exclusive-OR operation, whereas C indicates the AND operation. Therefore, these functions can be implemented simply with an exclusive-OR gate and an AND gate, respectively, as shown in Fig. 12.3*a*. This logic circuit with two inputs A and B, and two outputs S and C, is known as a **half-adder.** A block diagram for a half-adder is shown in Fig. 12.3*b*.

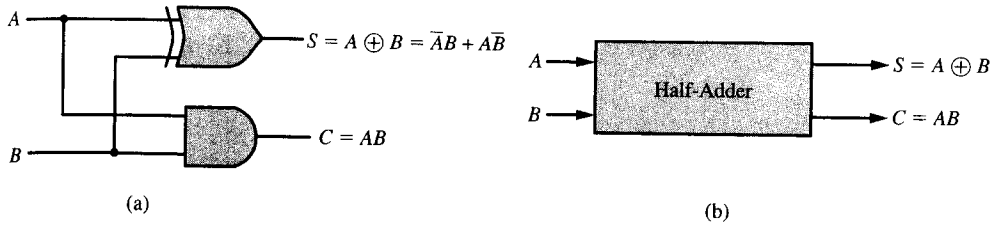

Fig. 12.3 (*a*) Half-adder, and (*b*) block diagram of half-adder.

Now let us see how we can extend the concept of a half-adder. First consider the example of adding $(93)_{10} = (1011101)_2$ and $(121)_{10} = (1111001)_2$ as shown below.

		1	1	1			1	←carries
		1	0	1	1	1	0	1
+		1	1	1	1	0	0	1
	1	1	0	1	0	1	1	0
								↑
								LSB

In determining the **least significant bit** (LSB)—the bit on the extreme right—2 bits are added together. This can be accomplished with a half-adder. To obtain any other bit, however, there must be the capability to deal with carries. In general, given two n-bit binary numbers $A_{n-1}A_{n-2} \ldots A_3A_2A_1A_0$ and $B_{n-1}B_{n-2} \ldots B_3B_2B_1B_0$, the process of addition is described as follows:

$$\begin{array}{r} C_nC_{n-1}C_{n-2} \ldots C_3C_2C_1 \\ A_{n-1}A_{n-2} \ldots A_3A_2A_1A_0 \\ + \quad B_{n-1}B_{n-2} \ldots B_3B_2B_1B_0 \\ \hline S_nS_{n-1}S_{n-2} \ldots S_3S_2S_1S_0 \end{array}$$

ere S_i is the sum of A_i, B_i, and C_i for $i = 1, 2, 3, \ldots, n-1$, and C_{i+1} is the ulting carry. This means that the sums $S_1, S_2, S_3, \ldots, S_{n-2}, S_{n-1}$ are formed as carries $C_2, C_3, \ldots, C_{n-2}, C_{n-1}, C_n$, according to Table 12.3*a*. Furthermore, ble 12.3*b* indicates how to obtain S_0 and C_1. (This is a half-adder truth table.) In dition, $S_n = C_n$. The result of adding the two given binary numbers is the binary mber $S_nS_{n-1}S_{n-2} \ldots S_3S_2S_1S_0$.

Table 12.3(a) Truth Table for S_i and C_{i+1} for $i = 1, 2, \ldots, n-1$.

A_i	B_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 12.3(b) Truth Table for S_0 and C_1.

A_0	B_0	S_0	C_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

From Table 12.3*a*, we have that

$$S_i = m_1 + m_2 + m_4 + m_7 = \overline{A}_i\overline{B}_iC_i + \overline{A}_iB_i\overline{C}_i + A_i\overline{B}_i\overline{C}_i + A_iB_iC_i \quad (12.5)$$

and

$$C_{i+1} = m_3 + m_5 + m_6 + m_7 = \overline{A}_iB_iC_i + A_i\overline{B}_iC_i + A_iB_i\overline{C}_i + A_iB_iC_i \quad (12.6)$$

The Karnaugh map for S_i is given in Fig. 12.4. From this map, we see that the expression for S_i cannot be simplified as a sum of products. On the other hand, the expression for C_{i+1} has the form of Eq. 12.2 and the Karnaugh map in Fig. 12.2*a* and, therefore, can be written as is Eq. 12.3. In other words, we can put C_{i+1} in the form

$$C_{i+1} = A_iB_i + B_iC_i + A_iC_i \quad (12.7)$$

The expressions for S_i and C_i can be implemented by using the two-level AND-OR logic circuits shown in Fig. 12.5. A logic circuit, such as the one given in Fig. 12.5, whose inputs are A_i, B_i, C_i and whose outputs are S_i and C_{i+1} as described by Eq. 12.5 and Eq. 12.7, respectively, is called a **full-adder.** A block diagram for a full-adder is shown in Fig. 12.6.

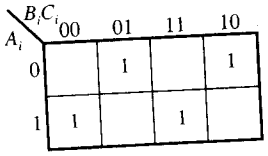

Fig. 12.4 Karnaugh map for S_i.

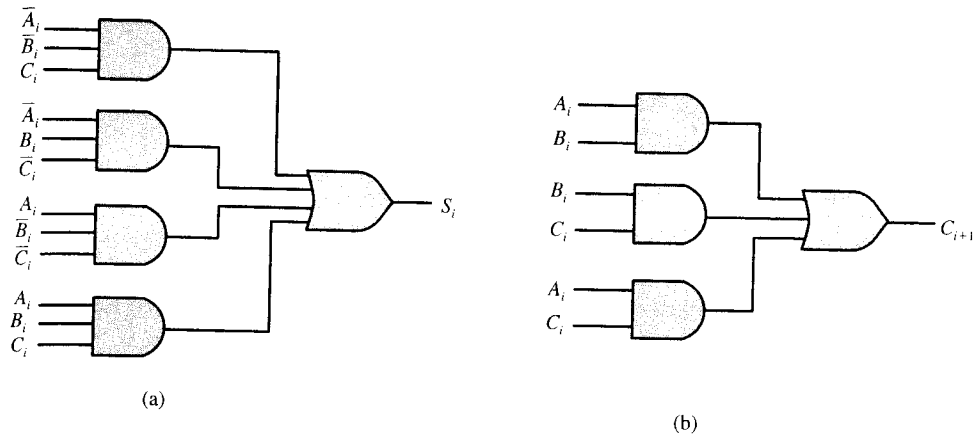

Fig. 12.5 AND-OR implementation of (*a*) S_i and (*b*) C_{i+1}.

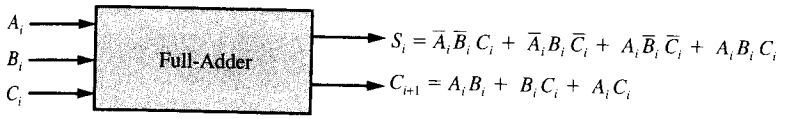

Fig. 12.6 Block diagram for a full-adder.

By connecting a half-adder and $n - 1$ full-adders as shown in Fig. 12.7, we get a combinational logic circuit that will enable us to add any two n-bit binary numbers $A_{n-1}A_{n-2} \ldots A_3A_2A_1A_0$ and $B_{n-1}B_{n-2} \ldots B_3B_2B_1B_0$.

An important alternative to the full-adder implementation given in Fig. 12.5 may also be derived. Although Eq. 12.5 is in simplified sum-of-products form, we can manipulate S_i algebraically as follows:

$$S_i = \overline{A}_i\overline{B}_iC_i + \overline{A}_iB_i\overline{C}_i + A_i\overline{B}_i\overline{C}_i + A_iB_iC_i = \overline{A}_i(\overline{B}_iC_i + B_i\overline{C}_i) + A_i(\overline{B}_i\overline{C}_i + B_iC_i)$$

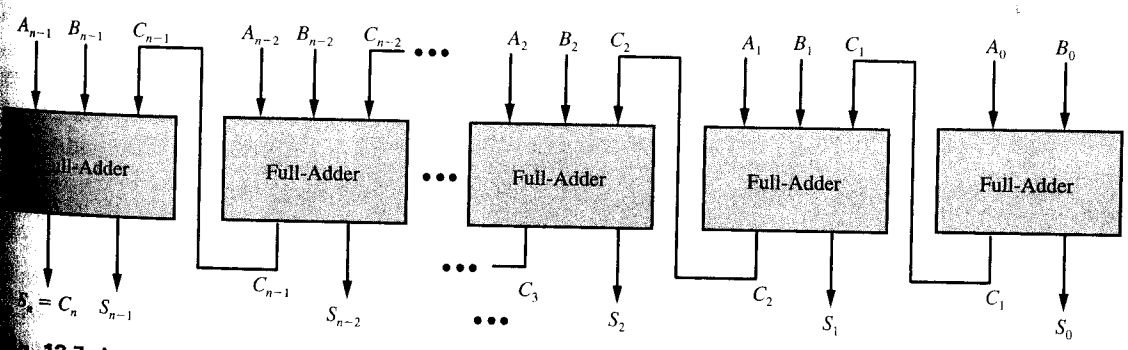

Fig. 12.7 An n-bit binary adder.

But note that

$$\begin{aligned}\overline{B}_iC_i + B_i\overline{C}_i &= \overline{\overline{\overline{B}_iC_i + B_i\overline{C}_i}} = \overline{(\overline{B}_i\overline{C}_i + B_iC_i)} \\ &= \overline{(B_i + C_i)(\overline{B}_i + \overline{C}_i)} = \overline{B_i\overline{B}_i + B_i\overline{C}_i + \overline{B}_iC_i + C_i\overline{C}_i} \\ &= \overline{\overline{B}_iC_i + B_i\overline{C}_i}\end{aligned}$$

If we define $D_i = \overline{B}_iC_i + B_i\overline{C}_i$ then $\overline{D}_i = \overline{\overline{B}_iC_i + B_i\overline{C}_i} = \overline{B}_i\overline{C}_i + B_iC_i$ and we can write

$$S_i = \overline{A}_iD_i + A_i\overline{D}_i = A_i \oplus D_i$$

where

$$D_i = \overline{B}_iC_i + B_i\overline{C}_i = B_i \oplus C_i$$

In other words, we can write S_i as

$$S_i = A_i \oplus (B_i \oplus C_i)$$

However, since the exclusive-OR operation is associative (see Problem 11.39 on p. 805), it is unambiguous to write

$$S_i = A_i \oplus B_i \oplus C_i$$

In addition to this, from Eq. 12.6, we have

$$\begin{aligned}C_{i+1} &= \overline{A}_iB_iC_i + A_i\overline{B}_iC_i + A_iB_i\overline{C}_i + A_iB_iC_i \\ &= (\overline{A}_iB_i + A_i\overline{B}_i)C_i + A_iB_i(\overline{C}_i + C_i) = (A_i \oplus B_i)C_i + A_iB_i\end{aligned}$$

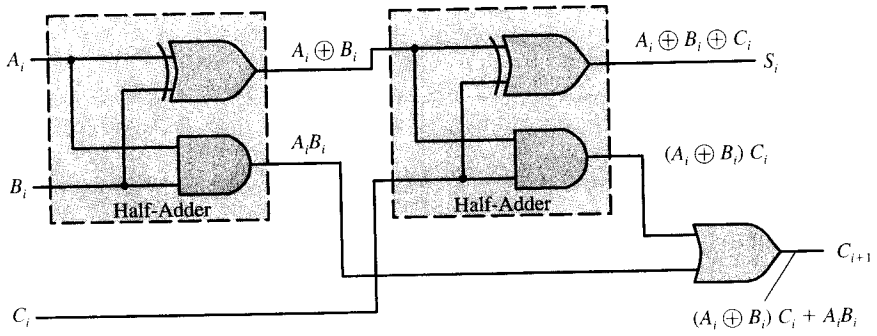

Fig. 12.8 Full-adder consisting of two half-adders and an OR gate.

be that such packages produce the exact implementations that are desired, or that such realizations can be obtained by appropriately connecting ICs together. This approach to design is our next topic of study.

12.2 MSI and LSI Design

In the preceding section, we saw how to design an n-bit binary adder (see Fig. 12.7) using a half-adder and $n - 1$ full-adders. To be able to include the case of a possible previous carry, the half-adder can be replaced by a full-adder. Figure 12.17 shows this more general **binary parallel adder** for the case that $n = 4$. Such a device is available as an MSI package (e.g., the TTL 74283 IC), as are other n-bit ($n \neq 4$) binary parallel adders. By connecting such ICs together appropriately, larger adders can be formed. For example, by connecting two 4-bit adders as shown in Fig. 12.18, we get an 8-bit binary parallel adder.

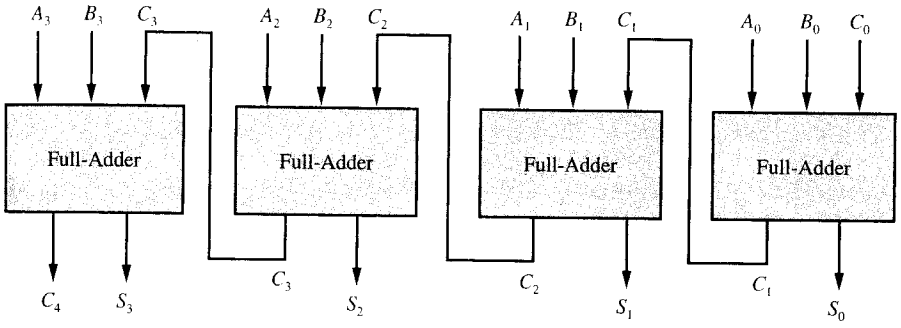

Fig. 12.17 A 4-bit binary parallel adder.

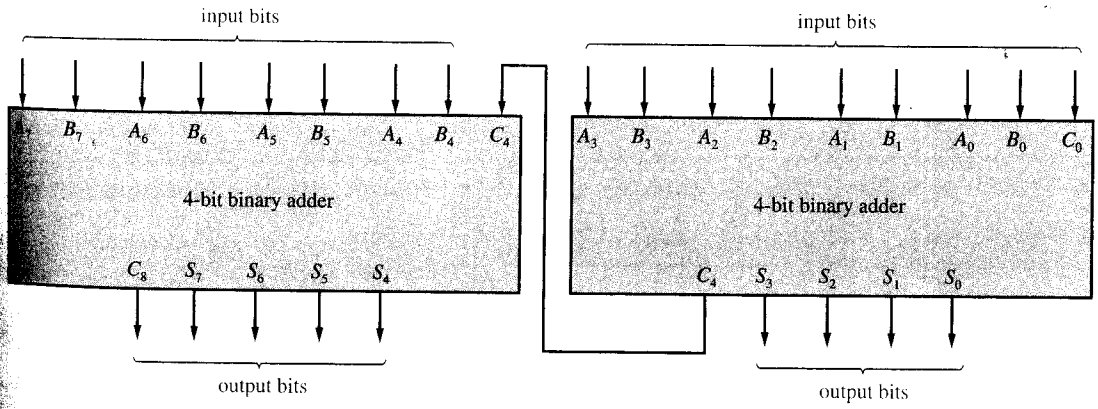

Fig. 12.18 An 8-bit binary parallel adder.

Propagation Delay, ns
10
6
33
3
10
2
25

Not only can adders be used to construct larger adders, but they also can be used for other types of applications as well. An example of such a situation is the conversion from BCD to the **excess-3 code.** Table 12.7 shows the BCD representations of the decimal digits and the corresponding words of the excess-3 code. Note that the words of the excess-3 code can be obtained by adding the binary number 3 (0011) to the corresponding BCD representations of the decimal digits. Because of this property, we can use a 4-bit binary parallel adder if we make $B_0 = B_1 = 1$ and $B_2 = B_3 = C_0 = 0$. Doing this will result in the desired addition, and thus produce outputs that are the appropriate excess-3 code words. An MSI implementation of such a code converter is shown in Fig. 12.19.

Table 12.7 Excess-3 Code-Word Correspondence with BCD.

BCD				Excess-3 Code			
A_3	A_2	A_1	A_0	S_3	S_2	S_1	S_0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

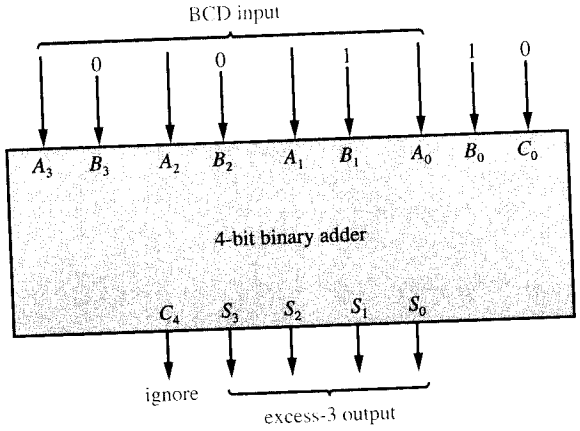

Fig. 12.19 A BCD-to-excess-3-code converter.

BCD Adders

There is an alternative to the binary addition of numbers—specifically, an alternative is decimal addition. By this we mean the addition of decimal digits in BCD form. Suppose we perform binary addition on the BCD representations of two decimal digits. If the sum of the two digits is 9 or less, then performing binary addition results in the appropriate number. If the sum of the digits is greater than 9, however, then what results from the addition is the binary number representation of the sum, not the BCD representation. Table 12.8 lists the sums that result from the binary addition of decimal digits in BCD form, and it also shows the BCD versions of these sums. Even though $9 + 9 = 18$, the table extends to include sums equaling 19 so that carries from a preceding addition can be included.

By applying two decimal digits in BCD form, say $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$, to a 4-bit binary adder (see Fig. 12.17), we obtain a sum $S_4S_3S_2S_1S_0$ in binary form, where $S_4 = C_4$ is the carry bit. If this binary number is 9 or less, then this sum is also in BCD form. If this binary number is greater than 9, however, we must modify the

Table 12.8 Sums of BCD Digits in Binary Form and BCD Form.

Sums in Binary Form					Sums in BCD Form				
S_4	S_3	S_2	S_1	S_0	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	1	0
0	0	0	1	1	0	0	0	1	1
0	0	1	0	0	0	0	1	0	0
0	0	1	0	1	0	0	1	0	1
0	0	1	1	0	0	0	1	1	0
0	0	1	1	1	0	0	1	1	1
0	1	0	0	0	0	1	0	0	0
0	1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	0	0
0	1	0	1	1	1	0	0	0	1
0	1	1	0	0	1	0	0	1	0
0	1	1	0	1	1	0	0	1	1
0	1	1	1	0	1	0	1	0	0
0	1	1	1	1	1	0	1	0	1
1	0	0	0	0	1	0	1	1	0
1	0	0	0	1	1	0	1	1	1
1	0	0	1	0	1	1	0	0	0
1	0	0	1	1	1	1	0	0	1

sum to get its BCD form. From Table 12.8, we see that if we add the binary number 6 (0110) to the sum in binary form, we get the sum in BCD form. But how can we state this in logical terms?

From Table 12.8 we have that the binary number 6 should be added to a sum in binary form when $S_4 = 1$, OR $S_3 = 1$ AND $S_2 = 1$, OR $S_3 = 1$ AND $S_1 = 1$. This description corresponds to the Boolean function

$$F = S_4 + S_3S_2 + S_3S_1$$

As a consequence of this discussion we can realize a **BCD adder,** that is, a logic circuit that adds decimal digits in BCD form, by using two 4-bit binary adders as shown in Fig. 12.20. Two decimal digits in BCD form are added together in binary

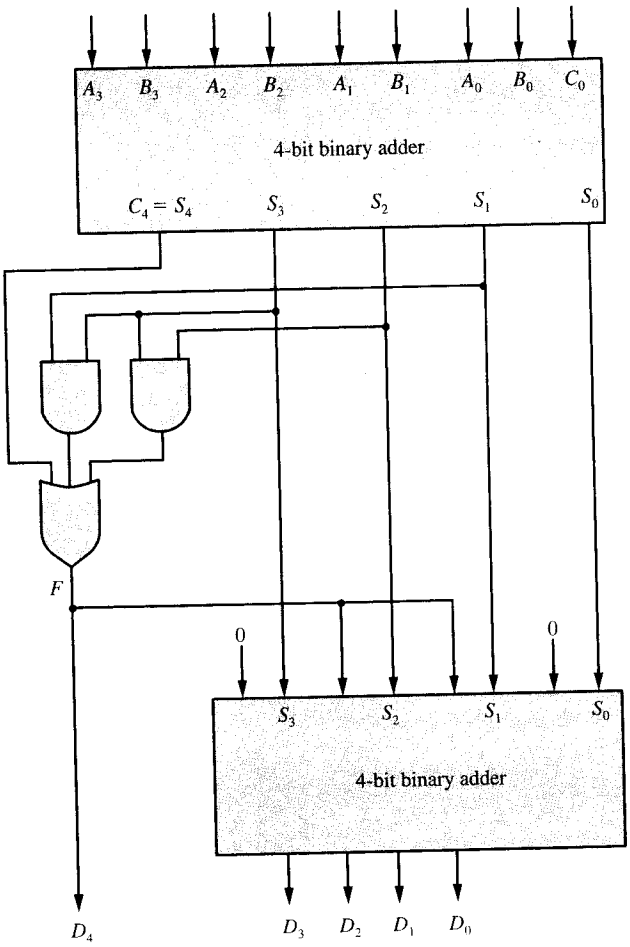

Fig. 12.20 A BCD adder.

by the top 4-bit adder. If the resulting binary number $S_4S_3S_2S_1S_0$ is 9 or less, then $S_4S_3S_2S_1S_0 = S_3S_2S_1S_0$, $F = 0$, and 0 is added to $S_3S_2S_1S_0$ by the bottom 4-bit adder. If the resulting binary number $S_4S_3S_2S_1S_0$ is greater than 9, however, then $F = 1$ and 6 is added to $S_3S_2S_1S_0$ by the bottom 4-bit adder. Because it is a 4-bit adder and S_4 is not applied to its input, the output carry D_4 cannot be obtained for the sums 16, 17, 18, and 19. This is no problem because a carry results when $F = 1$. Therefore, we can use the output of the OR gate to produce D_4. Note, too, that the bottom 4-bit adder employs no carry from a preceding stage and, therefore, no input for such a carry is indicated.

Even though we have spent time discussing how to construct them from two 4-bit adders, BCD adders are available as MSI packages (e.g., the TTL 82S83 IC). Despite this, it is worthwhile studying how to combine MSI circuits that implement one kind of function in order to produce logic circuits that realize another type of function.

Magnitude Comparators

Another very important digital logic function is performed by a **magnitude comparator,** and such a device is also available in MSI form (e.g., the TTL 7485 IC is a 4-bit magnitude comparator). As its name suggests, the function that is performed is the comparison of two numbers A and B, and the determination of whether $A < B$, $A > B$, or $A = B$.

Given two n-bit binary numbers $A = A_{n-1} \ldots A_2A_1A_0$ and $B = B_{n-1} \ldots B_2B_1B_0$, an n-bit magnitude comparator therefore will have $2n$ inputs and 3 outputs C, D, and E, which signify $A < B$, $A > B$, and $A = B$, respectively. (For example, $C = 0, D = 1, E = 0 \Rightarrow A > B$.) A truth table characterizing magnitude comparison has 2^{2n} rows—therefore, the construction of such a table will be avoided.

Specifically, let us consider the case that $n = 3$. In other words, suppose that we wish to compare the 3-bit binary numbers $A = A_2A_1A_0$ and $B = B_2B_1B_0$. The equality of a pair of bits A_i and B_i can be determined from Table 12.9. From this truth table, we see that $A_i = B_i$ is described by the Boolean expression $F_i = \overline{A}_i\overline{B}_i + A_iB_i = A_i \odot B_i$. We wish to have the output $C = 1$ when $A < B$. This inequality occurs

Table 12.9 Truth Table for the Equality of a Pair of Bits.

B_i	F_i
0	1
1	0
0	0
1	1

when $A_2 = 0$ ($\overline{A}_2 = 1$) AND $B_2 = 1$, OR when $A_1 = 0$ ($\overline{A}_1 = 1$) AND $B_1 = 1$ given that (AND) $A_2 = B_2$ ($F_2 = 1$), OR when $A_0 = 0$ ($\overline{A}_0 = 1$) AND $B_0 = 1$ given that (AND) $A_2 = B_2$ AND $A_1 = B_1$ ($F_2 = 1$ AND $F_1 = 1$). This description can be characterized by the Boolean function

$$C = \overline{A}_2 B_2 + \overline{A}_1 B_1 F_2 + \overline{A}_0 B_0 F_1 F_2$$

where $F_1 = \overline{A}_1 \overline{B}_1 + A_1 B_1 = A_1 \odot B_1$ and $F_2 = \overline{A}_2 \overline{B}_2 + A_2 B_2 = A_2 \odot B_2$ are exclusive-NOR functions.

Proceeding in a similar manner for the case that $A > B$, we get the Boolean function

$$D = A_2 \overline{B}_2 + A_1 \overline{B}_1 F_2 + A_0 \overline{B}_0 F_1 F_2$$

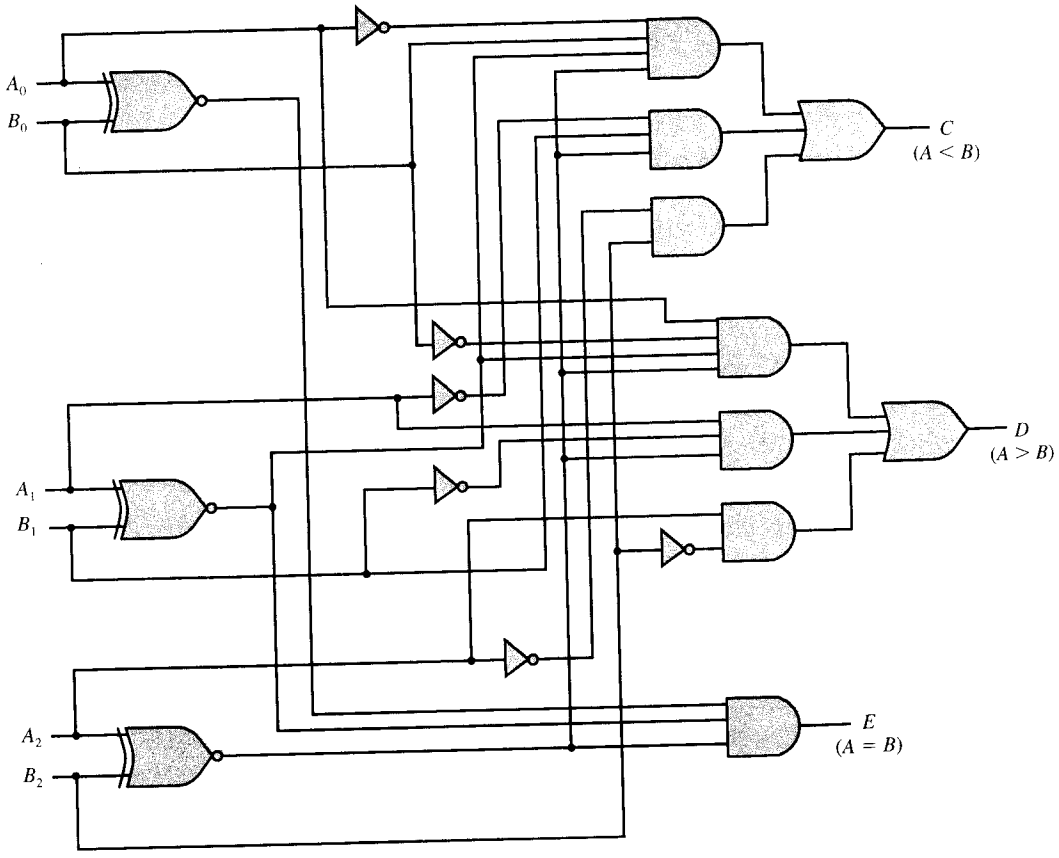

Fig. 12.21 A 3-bit magnitude comparator.

Finally, for the case that $A = B$ (i.e., $A_2 = B_2$ AND $A_1 = B_1$ AND $A_0 = B_0$), we have that

$$E = F_2F_1F_0 = (A_2 \odot B_2)(A_1 \odot B_1)(A_0 \odot B_0)$$

A logic circuit for a 3-bit magnitude comparator is shown in Fig. 12.21.

Encoders

In Section 11.2, we mentioned that various pieces of information could be represented by a code, that is, a set of binary sequences. As a simple example, consider the four pieces of information A_1, A_2, A_3, and A_4. To have four distinct binary sequences, each code word must consist of a minimum of 2 bits. Even though we can use sequences of more than 2 bits, for the sake of brevity, let us pick code words with 2 bits as given in Table 12.10. This table indicates that the code words for A_1, A_2, A_3, and A_4 are 00, 01, 10, and 11, respectively.

Table 12.10 A Binary Encoding.

Pieces of Information				Code Words	
A_1	A_2	A_3	A_4	B_1	B_2
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

Suppose we wish to construct a logic circuit to encode each piece of information. In other words, when $A_1 = 1$, we want 00 to be produced; when $A_2 = 1$, we want 01 to result, and so on. We will also assume that only one piece of information is encoded at a time (e.g., if $A_3 = 1$, then $A_1 = A_2 = A_4 = 0$). Under these conditions, from Table 12.10, we can write the Boolean expressions

$$B_1 = A_3 + A_4 \quad \text{and} \quad B_2 = A_2 + A_4$$

combinational logic circuit that implements these functions is shown in Fig. 12.22. ote that input A_1 is not connected to anything.) Such a device is called an **encoder.** eneral, an encoder for m pieces of information has m inputs (one for each piece information) and n outputs, where $2^n \geq m$. This can be called an ***m*-to-*n*-line** signated $m \times n$) **encoder.**

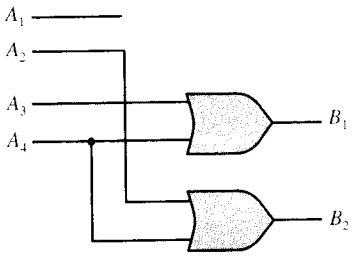

Fig. 12.22 Encoder corresponding to Table 12.10.

Table 12.11 A Priority Encoding.

Pieces of Information				Code Words	
A_1	A_2	A_3	A_4	B_1	B_2
1	×	×	×	0	0
0	1	×	×	0	1
0	0	1	×	1	0
0	0	0	1	1	1

Now let us consider the situation that two or more inputs are equal to 1 at the same time. For such a case, an encoder's output has no meaning—two pieces of information cannot be encoded at the same time. Therefore, let us assign a priority to the information to be encoded. For the example above, let the subscript of the information indicate its priority, that is, A_1 has the highest priority, A_2 is second, A_3 is third, and A_4 is last. Thus if more than one input is 1, the piece of information with the highest priority is encoded. These conditions are described by Table 12.11. Here the symbol × indicates a don't-care condition. For example, if $A_3 = 1$, it does not matter whether $A_4 = 0$ or $A_4 = 1$; since A_3 has priority over A_4, the resulting encoding is 10. From Table 12.11, we can write the Boolean expressions

$$B_1 = \overline{A}_1\overline{A}_2A_3 + \overline{A}_1\overline{A}_2\overline{A}_3A_4 = \overline{A}_1\overline{A}_2(A_3 + \overline{A}_3A_4) = \overline{A}_1\overline{A}_2(A_3 + A_4)$$

and

$$B_2 = \overline{A}_1A_2 + \overline{A}_1\overline{A}_2\overline{A}_3A_4 = \overline{A}_1(A_2 + \overline{A}_2\overline{A}_3A_4) = \overline{A}_1(A_2 + \overline{A}_3A_4)$$

A logic circuit for such an encoder, known as a **priority encoder,** is shown in Fig. 12.23.

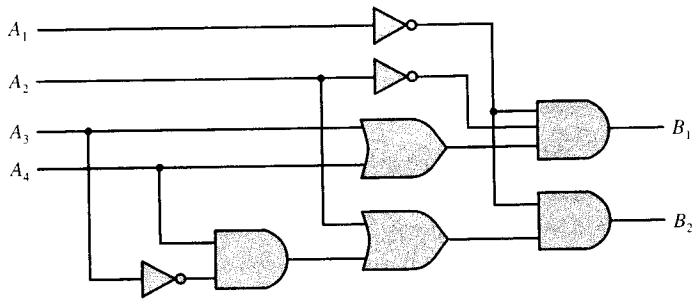

Fig. 12.23 A 4-to-2-line priority encoder.

Code Words	
B_1	B_2
0	0
0	1
1	0
1	1

Because of its simplicity of construction, ordinary (nonpriority) encoders do not come in the form of MSI packages. Priority encoders, however, are available as MSI units. For example, the 74147 IC is a 10-to-4-line decimal-to-BCD priority encoder, and the IC 74148 is an 8-to-3-line octal-to-binary priority encoder.

Decoders

The converse of the process of encoding is called **decoding.** In general, an ***n*-to-*m*-line decoder** has n inputs and m outputs, where $2^n \geq m$. For each n-bit input, there is a unique output that is equal to 1—the remaining outputs being equal to 0.

An example of a 3-to-8-line decoder is a device whose input is a 3-bit binary number between 0 and 7, and whose output is the corresponding octal digit. The truth table describing this decoder is given by Table 12.12, where D_0 through D_7 designate the octal digits 0 through 7 respectively. From this table we see that

$$D_i = m_i \qquad \text{for } i = 0, 1, 2, 3, 4, 5, 6, 7$$

Therefore, each output function is one of the minterms of the input variables. The implementations of these Boolean expressions with logic circuits should be a routine exercise by now, so they are omitted. Do note, however, that each output function can be implemented with one (3-input) AND gate.

Unlike (nonpriority) encoders, decoders are available in MSI form (e.g., the TTL 74138 IC is a 3-to-8-line decoder).

The 3×8 decoder characterized by Table 12.12 is an example of a decoder with n inputs and $m = 2^n$ outputs. The output functions for such a device are all the minterms of the input variables. Since any Boolean function can be expressed as a

Table 12.12 Truth Table for a (3-to-8-Line) Binary-to-Octal Decoder.

Inputs		Outputs							
B	C	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	1	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	1	0
1	1	0	0	0	0	0	0	0	1

sum of minterms, this type of decoder can be connected to an OR gate to implement any Boolean function. For example, from Eq. 12.5 and Eq. 12.6, we know that a full-adder is characterized by the Boolean functions

$$S_i = m_1 + m_2 + m_4 + m_7 \qquad \text{and} \qquad C_{i+1} = m_3 + m_5 + m_6 + m_7$$

Therefore, we can use OR gates and a 3-to-8-line decoder such as the one described by Table 12.12 to implement a full-adder. Figure 12.24 gives the realization.

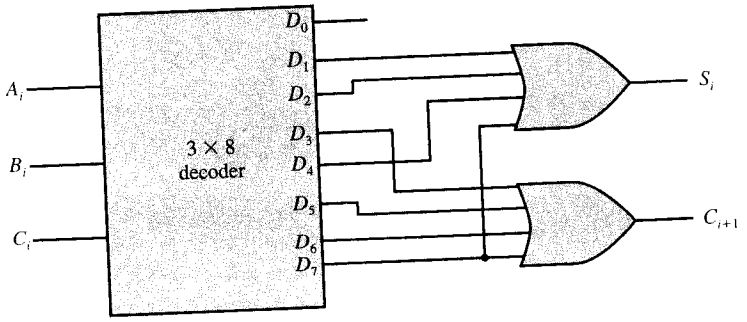

Fig. 12.24 Realization of a full-adder with a 3-to-8-line decoder.

Multiplexers

A **digital multiplexer** is a logic circuit typically with 2^n inputs, one output, and n selection variables or lines. For such a device, the n selection lines are used to make the output equal to one of the inputs. Because of this, a digital multiplexer (designated MUX) is also referred to as a **data selector.**

Figure 12.25 shows a realization of a 4-to-1-line multiplexer (designated 4 × 1 MUX). For this case ($n = 2$) there are four input lines labeled I_0, I_1, I_2, I_3; two selection lines labeled A, B; and one output line F. Note that when $A = B = 0$, then the output of the top AND gate is $\overline{A}\,\overline{B}I_0 = I_0$ and the output of each remaining AND gate is 0. Thus $F = I_0$. Similarly, when $A = 0$ and $B = 1$, then the output of the second (from the top) AND gate is $\overline{A}BI_1 = I_1$ and each of the other AND gates has an output of 0. Thus $F = I_1$. Continuing in this manner, we see that the output is equal to one of the inputs—which one depends on the selection-line values. These results are summarized by Table 12.13. The MSI representation of a 4-to-1-line multiplexer is shown in Fig. 12.26. Note that a Boolean expression for the output F is given by

$$F = \overline{A}\,\overline{B}I_0 + \overline{A}BI_1 + A\overline{B}I_2 + ABI_3 \tag{12.8}$$

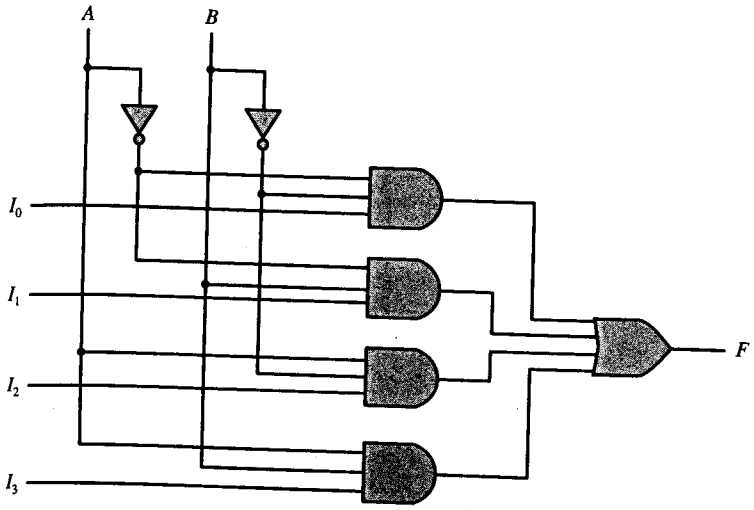

Fig. 12.25 A 4-to-1-line multiplexer.

Table 12.13 Table Characterizing a 4-to-1-Line Multiplexer.

Selection Lines		Output
A	B	F
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

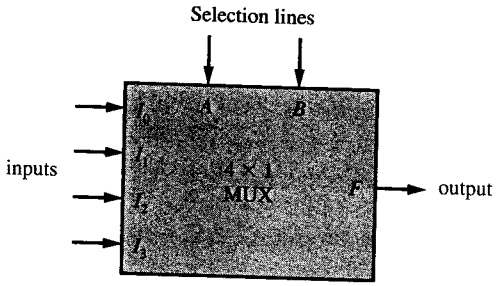

Fig. 12.26 Circuit symbol of a 4-to-1-line multiplexer.

Let us now see how we can use a multiplexer to implement Boolean functions. Suppose that we would like to realize the function F characterized by Table 12.14. From this truth table, we can write

$$F = \overline{A}BC + A\overline{B}\,\overline{C} + AB\overline{C} + ABC = \overline{A}BC + A\overline{B}\,\overline{C} + AB(\overline{C} + C)$$

$$F = \overline{A}\,\overline{B}(0) + \overline{A}BC + A\overline{B}\,\overline{C} + AB(1) \qquad (12.9)$$

Therefore, by putting the expression for F (Eq. 12.9) into the form of Eq. 12.8, we can make the associations

$$I_0 = 0 \qquad I_1 = C \qquad I_2 = \overline{C} \qquad I_3 = 1$$

Table 12.14 Truth Table for a Boolean Function *F*.

A	*B*	*C*	*F*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

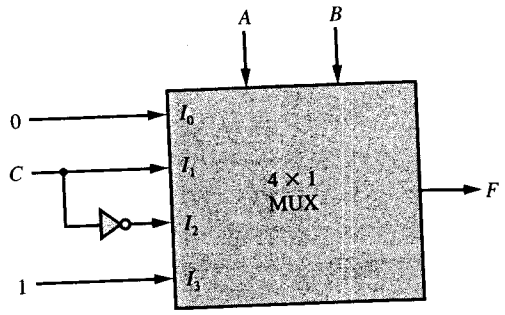

Fig. 12.27 Realization of *F* with a 4-to-1-line multiplexer.

This means that the 4-to-1-line multiplexer shown in Fig. 12.27 implements the Boolean function *F* described by Table 12.14.

Demultiplexers

The converse process of multiplexing is **demultiplexing.** Specifically, a **demultiplexer** is a device with one input, n selection lines, and 2^n outputs. For such a device, the n selection lines are used to make one of the outputs equal to the input. Figure 12.28 shows a realization of a 1-to-4-line demultiplexer. For this case ($n = 2$) there is one input line labeled E; two selection lines labeled A, B; and four output lines

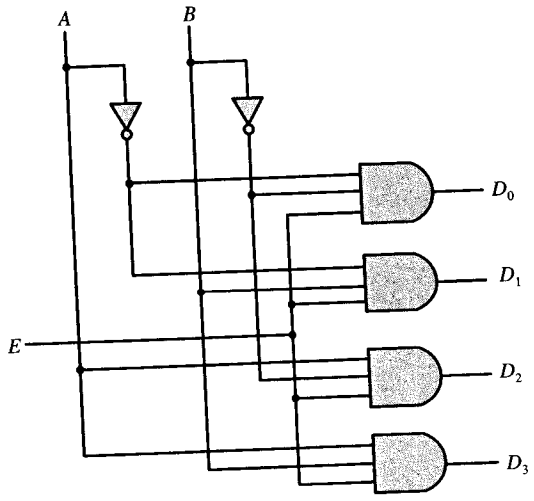

Fig. 12.28 A 1-to-4-line demultiplexer.

labeled D_0, D_1, D_2, D_3. Note that when $A = B = 0$, then the output of the top AND gate is $D_0 = \overline{A}\,\overline{B}E = E$ and the remaining outputs are $D_1 = D_2 = D_3 = 0$. Similarly, when $A = 0$ and $B = 1$, then $D_1 = \overline{A}BE = E$ and $D_0 = D_2 = D_3 = 0$. Continuing in this manner, we see that the input is transferred to one of the outputs—which one depends on the selection-line values. These results are summarized by Table 12.15. The MSI representation of a 1-to-4-line demultiplexer is shown in Fig. 12.29.

Table 12.15 Table Characterizing a 1-to-4-line demultiplexer.

Selection Lines		Outputs			
A	B	D_0	D_1	D_2	D_3
0	0	E	0	0	0
0	1	0	E	0	0
1	0	0	0	E	0
1	1	0	0	0	E

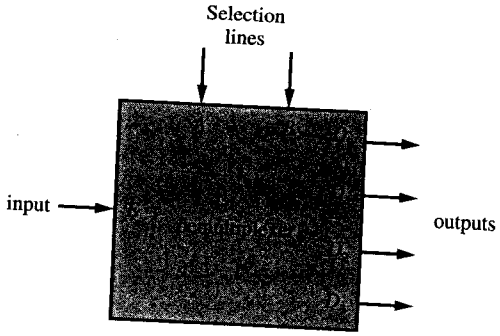

Fig. 12.29 Circuit symbol of a 1-to-4-line demultiplexer.

From Table 12.15, note that if we always have the condition that $E = 1$, then the 1-to-4-line demultiplexer becomes a 2-to-4-line decoder. On the other hand, if we always have $E = 0$, then the device does nothing—all the outputs are 0 regardless of values of the selection lines. Thus we can use a demultiplexer as a decoder; to do this, we just set $E = 1$ and we use the selection lines as the decoder inputs. In this case, E is referred to as the **enable** input—making $E = 1$ enables decoding, whereas $E = 0$ does not. Using the 1-to-4-line demultiplexer given in Fig. 12.29 as a 2-to-4-line decoder yields the MSI representation shown in Fig. 12.30.

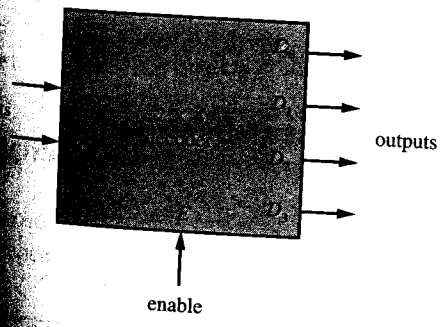

Fig. 12.30 A 2-to-4-line decoder with an enable