MODULE-2

COMBINATIONAL LOGIC

4.1 INTRODUCTION

- Logic circuits for digital systems may be **combinational or sequential**.
- A **combinational circuit** consists of **logic gates** whose outputs at any time are determined from only the present combination of inputs.
- A combinational circuit performs an **operation** that can be specified logically by a set of **Boolean functions**.
- In contrast, **sequential circuits** employ **storage elements** in addition to logic gates. Their outputs are a function of the inputs and the state of the **storage elements**. Because the state of the storage elements is a function of previous inputs, the outputs of a sequential circuit depend not only on present values of inputs, but also on past inputs, and the circuit behaviour must be specified by a time sequence of inputs and internal states.

4.2 COMBINATIONAL CIRCUITS

A combinational circuit consists of an **interconnection of logic gates**. Combinational logic gates react to the values of the signals at their inputs and produce the value of the output signal, transforming binary information from the given input data to a required output data.

A block diagram of a combinational circuit is shown in **Fig. 4.1** . The n input binary variables come from an external source; the m output variables are produced by the internal combinational logic circuit and go to an external destination. Each input and output variable exists physically as an analog signal whose values are interpreted to be a binary signal that represents logic 1 and logic 0. (Note : Logic simulators show only 0's and 1's, not the actual analog signals.)

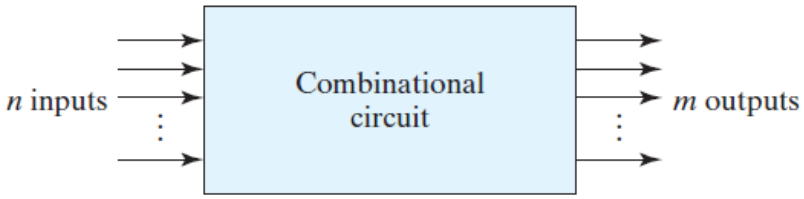


FIGURE 4.1
Block diagram of combinational circuit

In many applications, the **source and destination are storage registers**. If the registers are included with the combinational gates, then the total circuit must be a **sequential circuit.**

For n input variables, there are 2^n possible combinations of the binary inputs. For each possible input combination, there is one possible value for each output variable. Thus, a **combinational circuit can be specified with a truth table that lists the output values for each combination of input variables**.

A combinational circuit also can be described by **m Boolean functions**, one for each output variable. Each output function is expressed in terms of the n input variables.

The purpose of the current chapter is to use the knowledge acquired in previous chapters to formulate systematic analysis and design procedures for combinational circuits.

We will address three tasks:

(1) Analyze the behavior of a given logic circuit,

(2) synthesize a circuit that will have a given behaviour, and

(3) write hardware description language (HDL) models for some common circuits.

There are **several combinational circuits** that are employed extensively in the design of digital systems. These circuits are available in **integrated circuits** and are classified as standard components. They perform specific **digital functions** commonly needed in the design of digital systems.

In this chapter, we introduce the most important standard combinational circuits, such as adders, subtractors, comparators, decoders, encoders, and multiplexers. These components are available in integrated circuits as medium-scale integration (MSI) circuits. They are also used as standard cells in complex very largescale integrated (VLSI) circuits such as application-specific integrated circuits (ASICs). The standard cell functions are interconnected within the VLSI circuit in the same way that they are used in multiple-IC MSI design.

4.4 DESIGN PROCEDURE

The design of combinational circuits starts from the specification of the design objective and culminates in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be obtained. The procedure involves the following steps:

1. From the **specifications** of the circuit, determine the required number of inputs and outputs and assign a symbol to each.

2. Derive the **truth table** that defines the required relationship between inputs and outputs

3. Obtain the **simplified Boolean functions** for each output as a function of the input variables.

4. **Draw the logic diagram** and verify the correctness of the design (manually or by simulation).

A **truth table** for a combinational circuit consists of **input columns and output columns**. The input columns are obtained from the 2^n binary numbers for the n input variables. The binary values for the outputs are determined from the stated specifications. The output functions specified in the truth table give the exact definition of the combinational circuit. It is important that the verbal specifications be interpreted correctly in the truth table, as they are often incomplete, and any wrong interpretation may result in an incorrect truth table.

The **output binary functions** listed in the truth table are **simplified** by any available method, such as **algebraic manipulation, the map method, or a computer-based simplification program.**

A practical design must consider such constraints as the number of gates, number of inputs to a gate, propagation time of the signal through the gates, number of interconnections, limitations of the driving capability of each gate (i.e., the number of gates to which the output of the circuit may be connected), and various other criteria that must be taken into consideration when designing integrated circuits.

In most cases, the simplification begins by satisfying an elementary objective, such as producing the simplified **Boolean functions in a standard form**. Then the simplification proceeds with further steps to meet other performance criteria.

Code Conversion Example

The availability of a large variety of codes for the same discrete elements of information results in the use of different codes by different digital systems.

It is sometimes necessary to use the output of one system as the input to another. A conversion circuit must be inserted between the two systems if each uses different codes for the same information. Thus, a code converter is a circuit that makes the two systems compatible even though each uses a different binary code.

To convert from binary code A to binary code B, the input lines must supply the bit combination of elements as specified by code A and the output lines must generate the corresponding bit combination of code B. A combinational circuit performs this transformation by means of logic gates. The design procedure will be illustrated by an **example** that converts **binary coded decimal (BCD) to the excess-3 code for the decimal digits.**

The bit combinations assigned to the BCD and excess-3 codes are listed in **Table 1.5** (Section 1.7). Since each code uses **four bits to represent a decimal digit**, there must be four input variables and four output variables. We designate the four input binary variables by the

symbols A, B, C, and D, and the four output variables by w, x, y , and z . The truth table relating the input and output variables is shown in **Table 4.2** .

Note that four binary variables may have **16 bit combinations**, but only 10 are listed in the truth table. The **six bit combinations** not listed for the input variables are **don't-care combinations**. These values have no meaning in BCD and we assume that they will never occur in actual operation of the circuit. Therefore, we are at liberty to assign to the output variables either a 1 or a 0, whichever gives a simpler circuit.

Table 1.5
Four Different Binary Codes for the Decimal Digits

Decimal Digit	BCD 8421	2421	Excess-3	8, 4, −2, −1
0	0000	0000	0011	0000
1	0001	0001	0100	0111
2	0010	0010	0101	0110
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1011
6	0110	1100	1001	1010
7	0111	1101	1010	1001
8	1000	1110	1011	1000
9	1001	1111	1100	1111
	1010	0101	0000	0001
Unused	1011	0110	0001	0010
bit	1100	0111	0010	0011
combi-	1101	1000	1101	1100
nations	1110	1001	1110	1101
	1111	1010	1111	1110

Table 4.2
Truth Table for Code Conversion Example

Input BCD				Output Excess-3 Code			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
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

The maps in **Fig. 4.3** are plotted to obtain **simplified Boolean functions for the outputs**. Each one of the **four maps represents one of the four outputs** of the circuit as a function of the four input variables. The 1's marked inside the squares are obtained from the minterms that make the output equal to 1. The 1's are obtained from the truth table by going over the output columns one at a time. For example, the column under output z has five 1's; therefore, the map for z has five 1's, each being in a square corresponding to the minterm that makes z equal to 1. The six don't-care minterms 10 through 15 are marked with an X . One possible way to simplify the functions into sum-of-products form is listed under the map of each variable.(See Chapter 3 .)

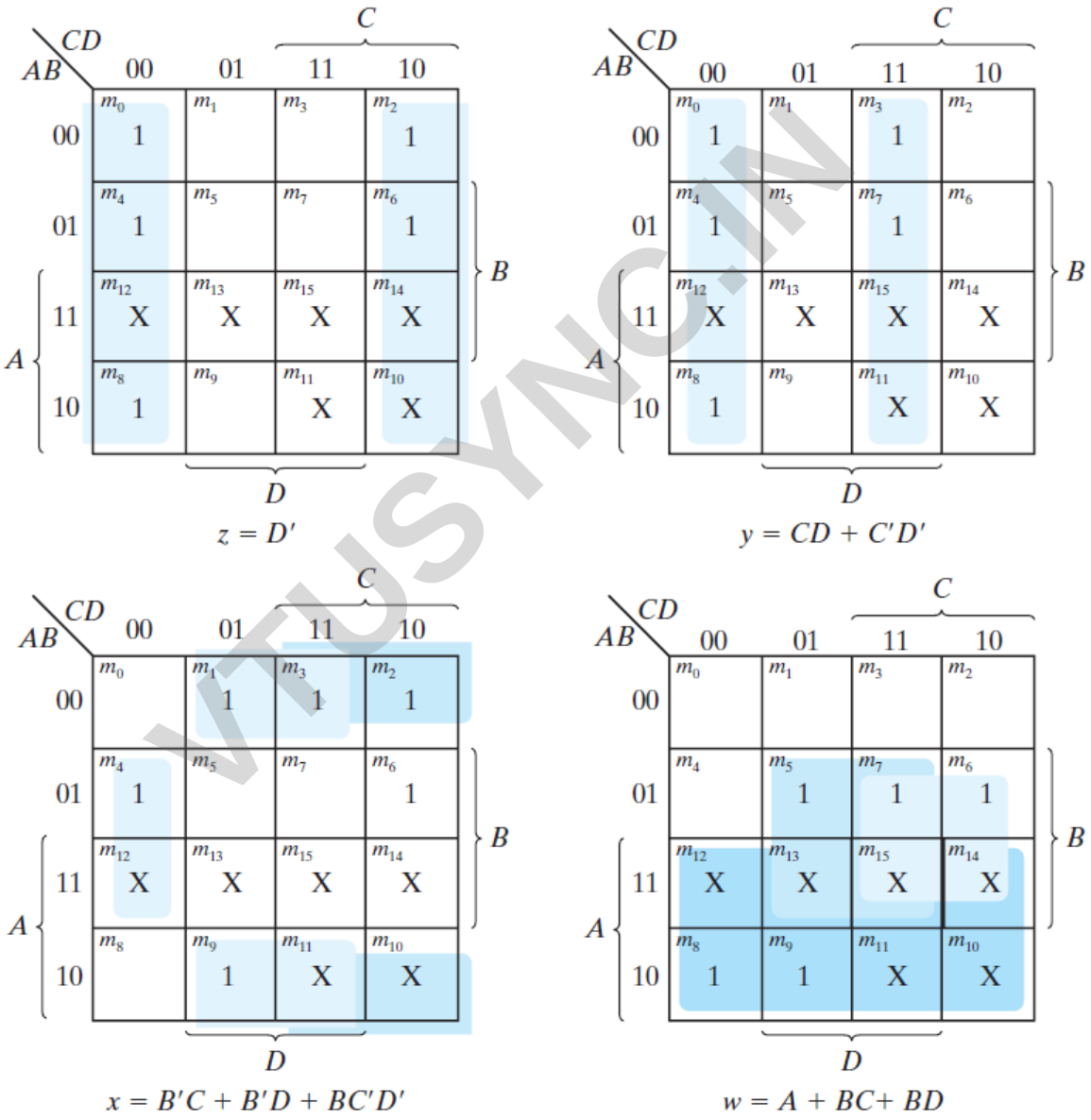


FIGURE 4.3
Maps for BCD-to-excess-3 code converter

A two-level logic diagram for each output may be obtained directly from the Boolean expressions derived from the maps. The expressions obtained in Fig. 4.3 may be manipulated algebraically for the purpose of using common gates for two or more outputs. This manipulation, shown next, illustrates the flexibility obtained with multiple-output systems when implemented with three or more levels of gates:

$$z = D'$$
$$y = CD + C'D' = CD + (C + D)'$$
$$x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$$
$$= B'(C + D) + B(C + D)'$$
$$w = A + BC + BD = A + B(C + D)$$

The **logic diagram** that implements these expressions is shown in **Fig. 4.4** . Note that the OR gate whose output is C + D has been used to implement partially each of three outputs. Not counting input inverters, the implementation in sum-of-products form requires seven AND gates and three OR gates. The implementation of Fig. 4.4 requires four AND gates, four OR gates, and one inverter. If only the normal inputs are available, the first implementation will require inverters for variables *B, C* , and *D* , and the second implementation will require inverters for variables *B* and *D* . Thus, the three-level logic circuit requires fewer gates, all of which in turn require no more than two inputs.

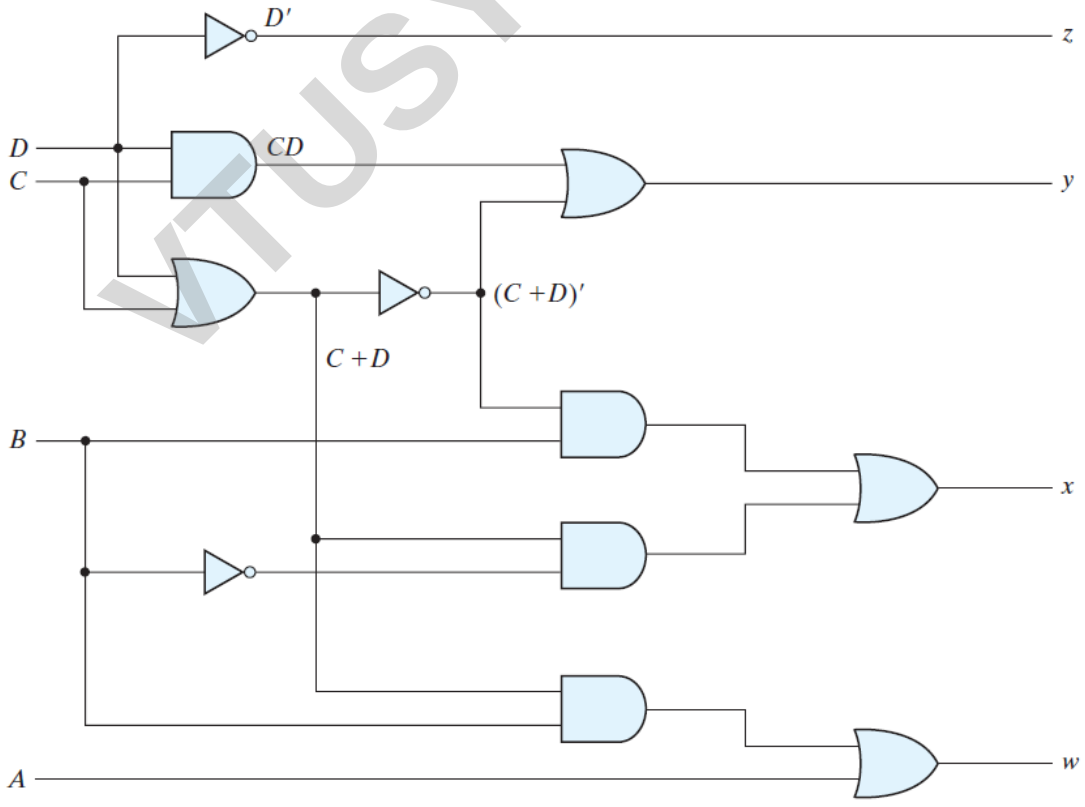


FIGURE 4.4
Logic diagram for BCD-to-excess-3 code converter

4.5 BINARY ADDER–SUBTRACTOR

The most basic arithmetic operation is the addition of two binary digits. This simple addition consists of four possible elementary operations: 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, and 1 + 1 = 10. The first three operations produce a sum of one digit, but when both augend and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a carry ., the carry obtained from the addition of two bits is added to the next higher order pair of significant bits.

A combinational circuit that performs the addition of two bits is called a half adder. One that performs the addition of three bits (two significant bits and a previous carry) is a full adder . The names of the circuits stem from the fact that two half adders can be employed to implement a full adder.

A **binary adder–subtractor** is a combinational circuit that performs the arithmetic operations of addition and subtraction with binary numbers. We will develop this circuit by means of a hierarchical design. The half adder design is carried out first, from which we develop the full adder. Connecting n full adders in cascade produces a binary adder for two n -bit numbers. The subtraction circuit is included in a complementing circuit.

HALF ADDER

From the verbal explanation of a half adder, we find that this circuit needs two binary inputs and two binary outputs. The input variables designate the augend and addend bits; the output variables produce the sum and carry. We assign symbols x and y to the two inputs and S (for sum) and C (for carry) to the outputs. The truth table for the half adder is listed in **Table 4.3 .** The C output is 1 only when both inputs are 1. The S output represents the least significant bit of the sum.

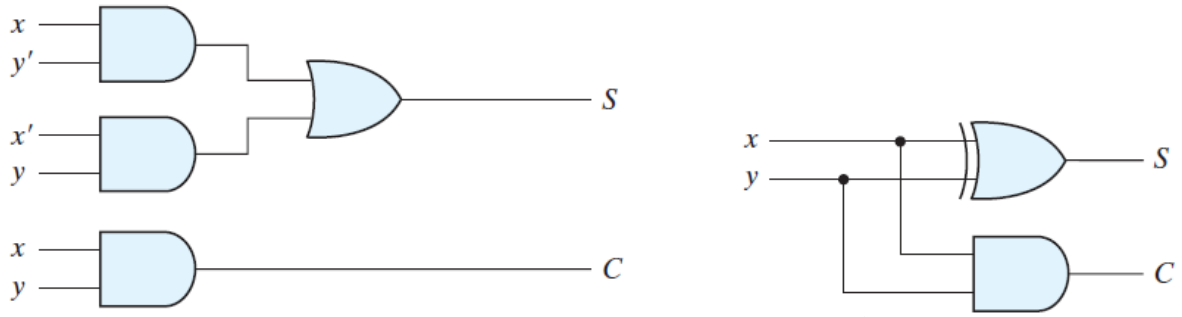
Table 4.3
Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum-of-products expressions are

$$S = x'y + xy'$$
$$C = xy$$

The logic diagram of the half adder implemented in sum of products is shown in **Fig. 4.5(a).** It can be also implemented with an exclusive-OR and an AND gate as shown in **Fig. 4.5(b)**. This form is used to show that two half adders can be used to construct a full adder.



(a) $S = xy' + x'y$
$C = xy$

(b) $S = x \oplus y$
$C = xy$

FIGURE 4.5
Implementation of half adder

FULL ADDER

Addition of n-bit binary numbers requires the use of a full adder, and the process of addition proceeds on a bit-by-bit basis, right to left, beginning with the least significant bit. After the least significant bit, addition at each position adds not only the respective bits of the words, but must also consider a possible carry bit from addition at the previous position.

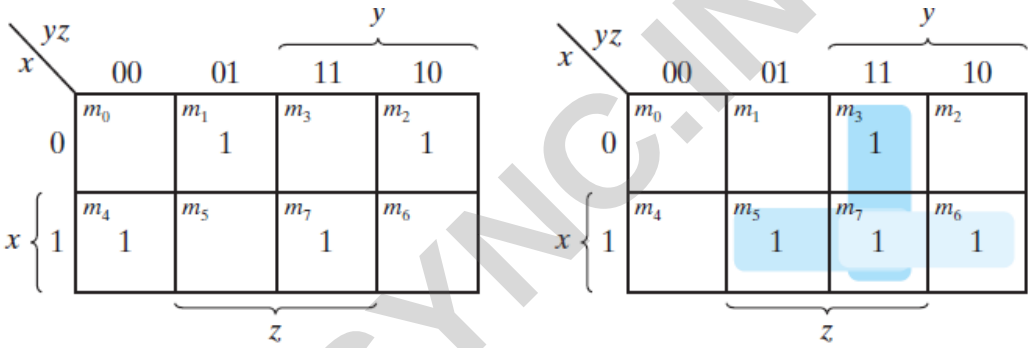
A full adder is a **combinational circuit** that forms the arithmetic sum of three bits. It consists of **three inputs and two outputs**. Two of the input variables, denoted by x and y ,represent the two significant bits to be added. The third input, z , represents the carry from the previous lower significant position. The two outputs are designated by the symbols S for sum and C for carry. The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry formed by adding the input carry and the bits of the words. The truth table of the full adder is listed in **Table 4.4** . When all input bits are 0, the output is 0. The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output has a carry of 1 if two or three inputs are equal to 1.

The input and output bits of the combinational circuit have different interpretations at various stages of the problem. The maps for the outputs of the full adder are shown in **Fig. 4.6** . The simplified expressions are

$$S = x'y'z + x'yz' + xy'z' + xyz$$
$$C = xy + xz + yz$$

Table 4.4
Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



(a) $S = x'y'z + x'yz' + xy'z' + xyz$ (b) $C = xy + xz + yz$

FIGURE 4.6
K-Maps for full adder

The logic diagram for the full adder implemented in sum-of-products form is shown in **Fig. 4.7** . It can also be implemented with two half adders and one OR gate, as shown

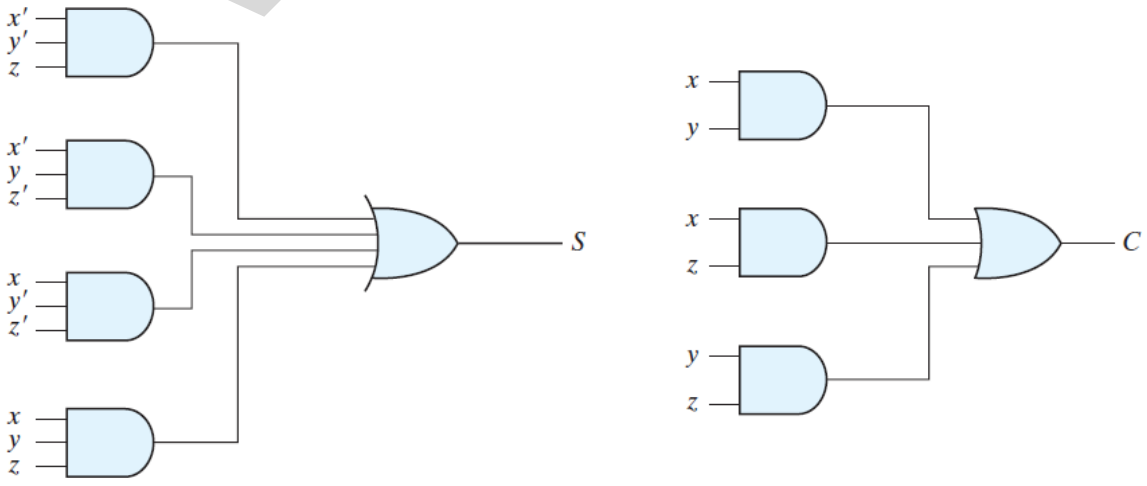


FIGURE 4.7
Implementation of full adder in sum-of-products form

In **Fig. 4.8** . The S output from the second half adder is the exclusive-OR of z and the output of the first half adder, giving

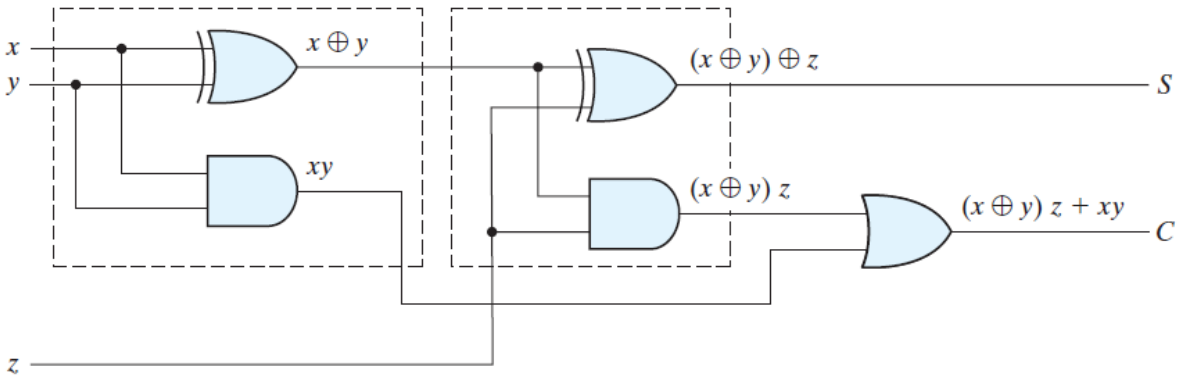


FIGURE 4.8
Implementation of full adder with two half adders and an OR gate

$$\begin{aligned} S &= z \oplus (x \oplus y) \\ &= z'(xy' + x'y) + z(xy' + x'y)' \\ &= z'(xy' + x'y) + z(xy + x'y') \\ &= xy'z' + x'yz' + xyz + x'y'z \end{aligned}$$

The carry output is

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

Binary Adder

A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be **constructed with full adders connected in cascade**, with the output carry from each full adder connected to the input carry of the next full adder in the chain.

Addition of n-bit numbers requires a chain of n full adders or a chain of one-half adder and n -1 full adders.

Figure 4.9 shows the interconnection of four full-adder (FA) circuits to provide a **four-bit binary ripple carry adder**. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the least significant bit. The carries are connected in a chain through the full adders. The input carry to the adder is C0, and it ripples through the full adders to the output carry C4. The S outputs generate the required sum bits. An n -bit adder requires n full adders, with each output carry connected to the input carry of the next higher order full adder.

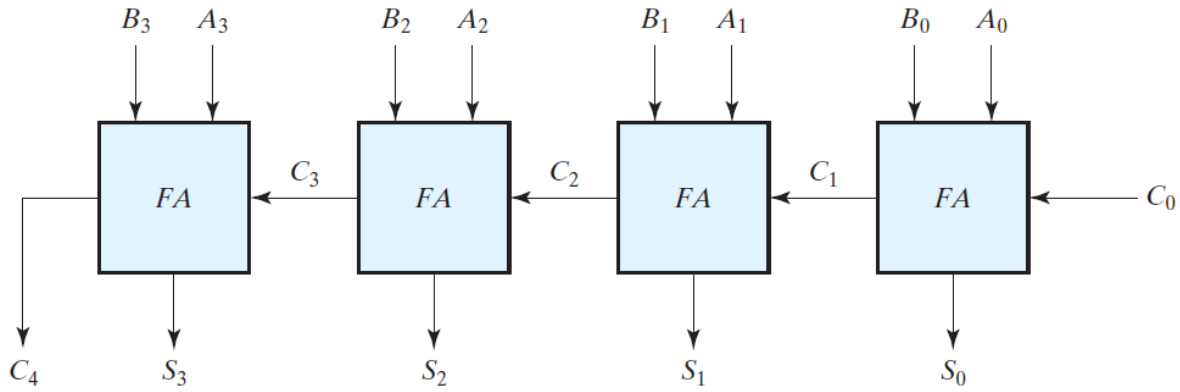


FIGURE 4.9
Four-bit adder

To demonstrate with a specific **example**, consider the two binary numbers A = 1011 and B = 0011. Their sum S = 1110 is formed with the four-bit adder as follows:

Subscript *i*:	**3**	**2**	**1**	**0**	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

Carry Propagation

As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals.

The **total propagation time** is equal to the propagation **delay of a typical gate**, times the **number of gate levels** in the circuit. The longest **propagation delay time in an adder** is the time it takes the carry to propagate through the full adders. Since each bit of the sum output depends on the value of the input carry, the value of Si at any given stage in the adder will be in its steady-state final value only after the input carry to that stage has been propagated. In this regard, consider output S3 in **Fig. 4.9** . Inputs A3 and B3 are available as soon as input signals are applied to the adder. However, input carry C3 does not settle to its final value until C2 is available from the previous stage. Similarly, C2 must wait for C1 and so on down to C0. Thus, only after the carry propagates and ripples through all stages will the last output S3 and carry C4 settle to their final correct value.

The **number of gate levels** for the carry propagation can be found from the circuit of the full adder. The circuit is redrawn with different labels in **Fig. 4.10** for convenience. The input and output variables use the **subscript i to denote a typical stage of the adder**. The signals at Pi and Gi settle to their steady-state values after they propagate through their

respective gates. These two signals are common to all half adders and depend on only the input augend and addend bits. The signal from the input carry Ci to the output carry Ci+1 propagates through an AND gate and an OR gate, which constitute two gate levels. If there are four full adders in the adder, the output carry C4 would have 2 * 4 = 8 gate levels from C0 to C4. **For an n -bit adder, there are 2 n gate levels for the carry to propagate from input to output**.

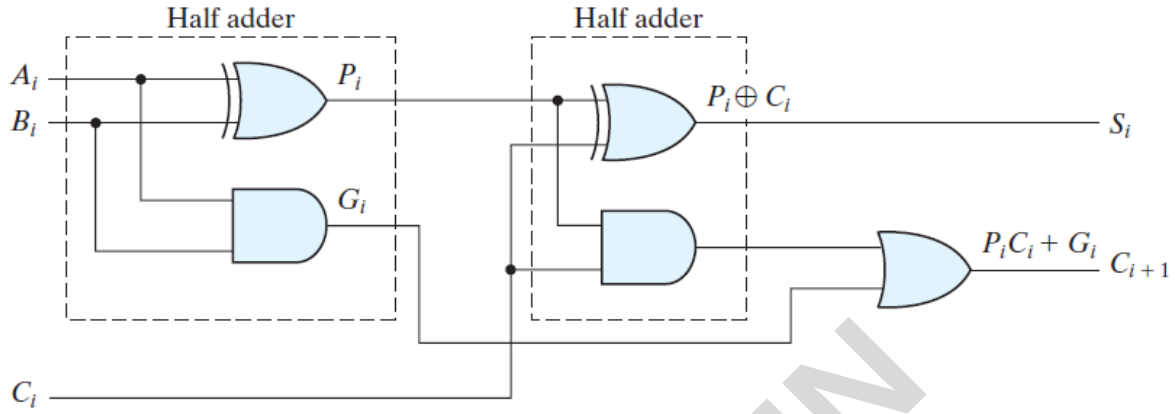


FIGURE 4.10
Full adder with *P* and *G* shown

The **carry propagation time** is an important attribute of the adder because it limits the speed with which two numbers are added.

An obvious solution for reducing the carry propagation delay time is to employ **faster gates with reduced delays**. However, physical circuits have a limit to their capability. Another solution is to increase the complexity of the equipment in such a way that the carry delay time is reduced. There are several techniques for reducing the carry propagation time in a parallel adder. The most widely used technique employs the principle of **carry lookahead logic** .

Consider the circuit of the full adder shown in **Fig. 4.10** . If we define two new binary variables

$$P_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$

the output sum and carry can respectively be expressed as

$$S_i = P_i \oplus C_i$$
$$C_{i+1} = G_i + P_i C_i$$

Gi is called a ***carry generate*** , and it produces a carry of 1 when both *Ai* and *Bi* are 1, regardless of the input carry Ci. **Pi is called a carry propagate** , because it determines whether a carry into stage i will propagate into stage i + 1 (i.e., whether an assertion of Ci will propagate to an assertion of Ci+1).

We now write the Boolean functions for the carry outputs of each stage and substitute the value of each *Ci* from the previous equations:

$$C_0 = \text{input carry}$$
$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 = P_2P_1P_0C_0$$

Since the Boolean function for each output carry is expressed in **sum-of-products form**, each function can be implemented with **one level of AND gates followed by an OR gate**.

The three Boolean functions for C1, C2, and C3 are implemented in the carry lookahead generator shown in **Fig. 4.11** . Note that this circuit can add in less time because C3 does not have to wait for C2 and C1 to propagate; in fact, C3 is propagated at the same time as C1 and C2. This **gain in speed of operation** is achieved at the expense of additional complexity (hardware).

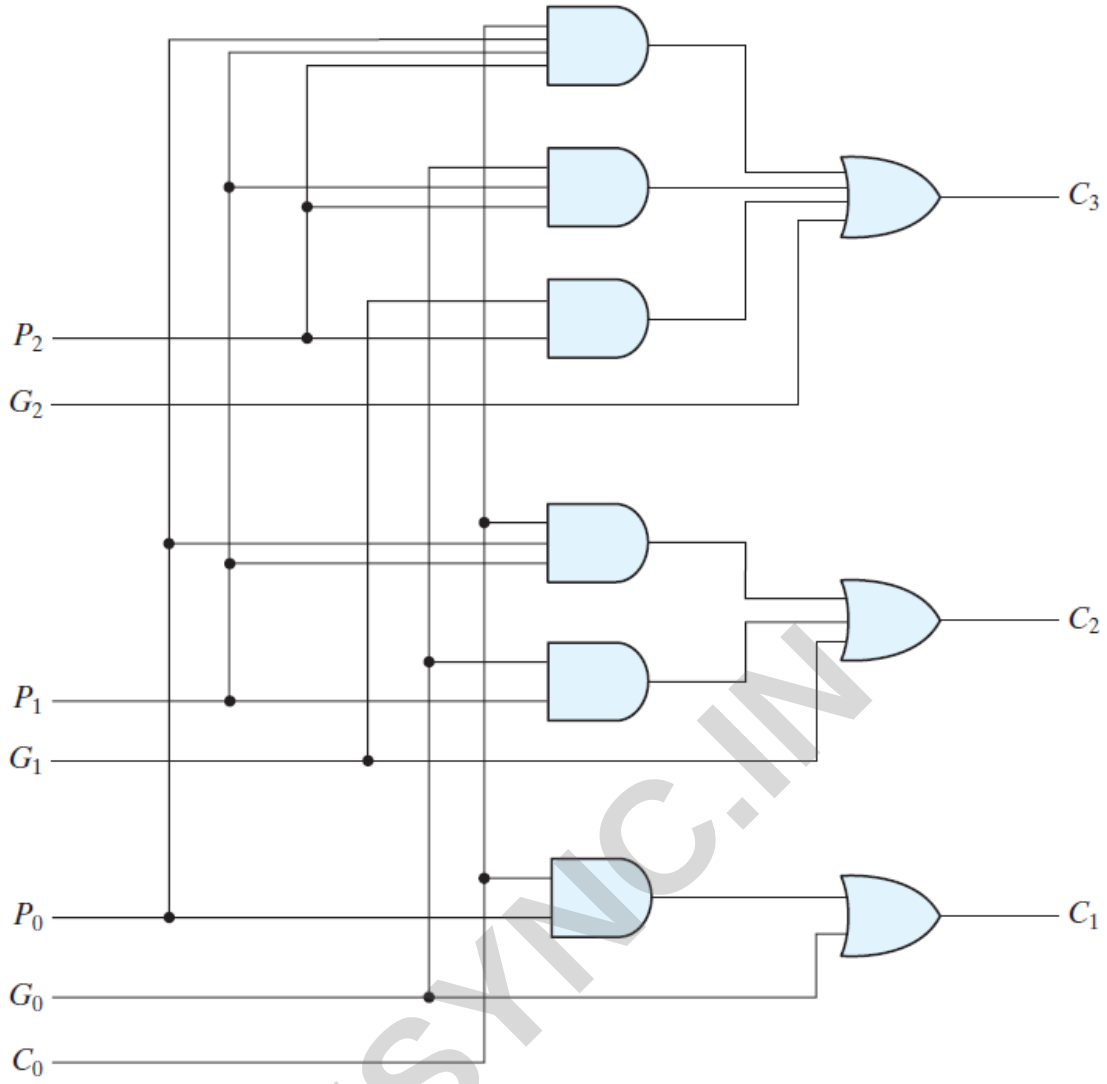


FIGURE 4.11
Logic diagram of carry lookahead generator

The construction of **a four-bit adder with a carry lookahead scheme** is shown in **Fig. 4.12.** Each sum output requires two exclusive-OR gates.

The output of the first exclusive-OR gate generates the Pi variable, and the AND gate generates the Gi variable. The carries are propagated through the carry lookahead generator (similar to that in **Fig. 4.11**) and applied as inputs to the second exclusive-OR gate. All output carries are generated after a delay through two levels of gates. Thus, outputs S1 through S3 have equal propagation delay times. The two-level circuit for the output carry C4 is not shown. This circuit can easily be derived by the equation-substitution method.

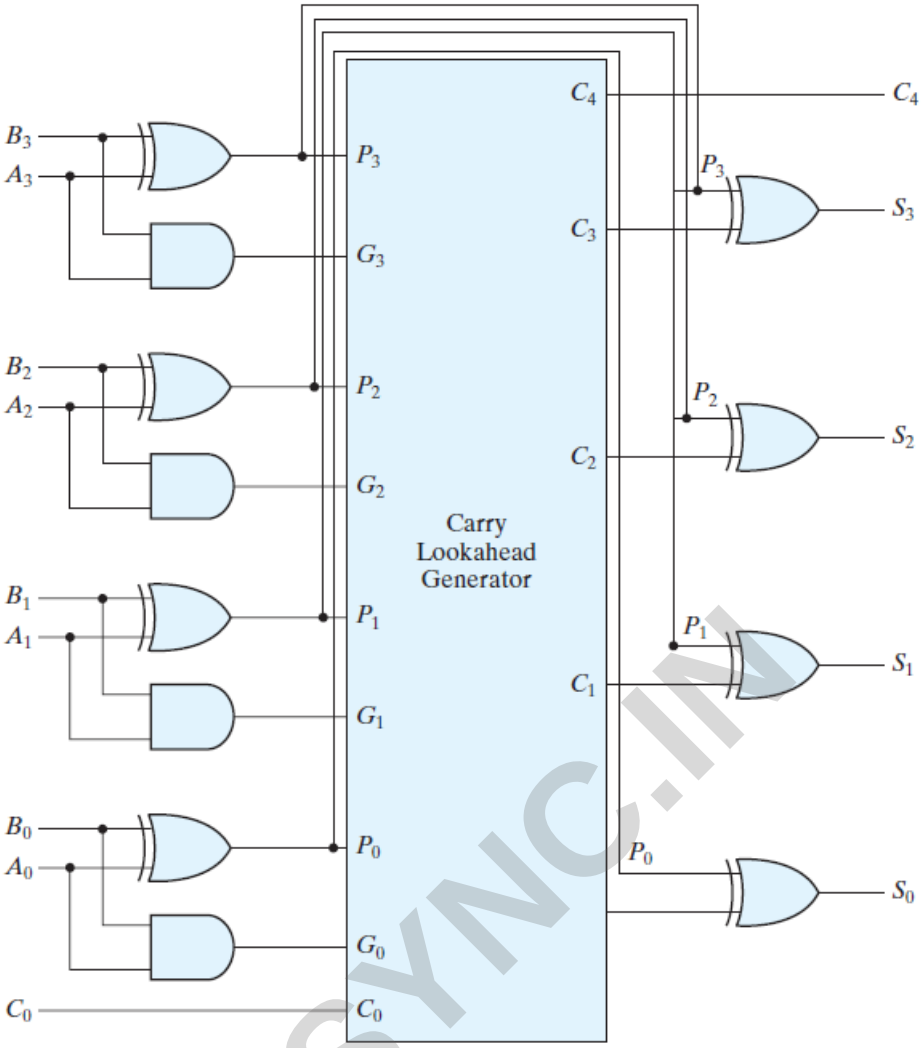


FIGURE 4.12
Four-bit adder with carry lookahead

Binary Subtractor

- The subtraction of unsigned binary numbers can be done most conveniently by means of complements, Remember that the subtraction A - B can be done by taking the 2's complement of B and adding it to A .
- The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits. The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry.
- The **circuit for subtracting A - B** consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C0 must be equal to 1 when subtraction is performed.
- The operation thus performed becomes A ,plus the 1's complement of B , plus 1. This is equal to A plus the 2's complement of B .

- For unsigned numbers, that gives A - B if A ≥ B or the 2's complement of (B – A) if A < B. For signed numbers, the result is A - B, provided that there is no overflow.

The **addition and subtraction operations can be combined into one circuit** with one common binary adder by including an **exclusive-OR gate with each full adder**.

- A four-bit adder–subtractor circuit is shown in **Fig. 4.13** . The mode input M controls the operation. When **M = 0, the circuit is an adder, and when M = 1, the circuit becomes a subtractor**
- .Each exclusive-OR gate receives input M and one of the inputs of B . When M = 0, we have B XOR 0 = B. The full adders receive the value of B , the input carry is 0, and the circuit performs A plus B .
- When M = 1, we have B XOR 1 = B' and C0 = 1. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B . (The exclusive-OR with output V is for detecting an overflow.)

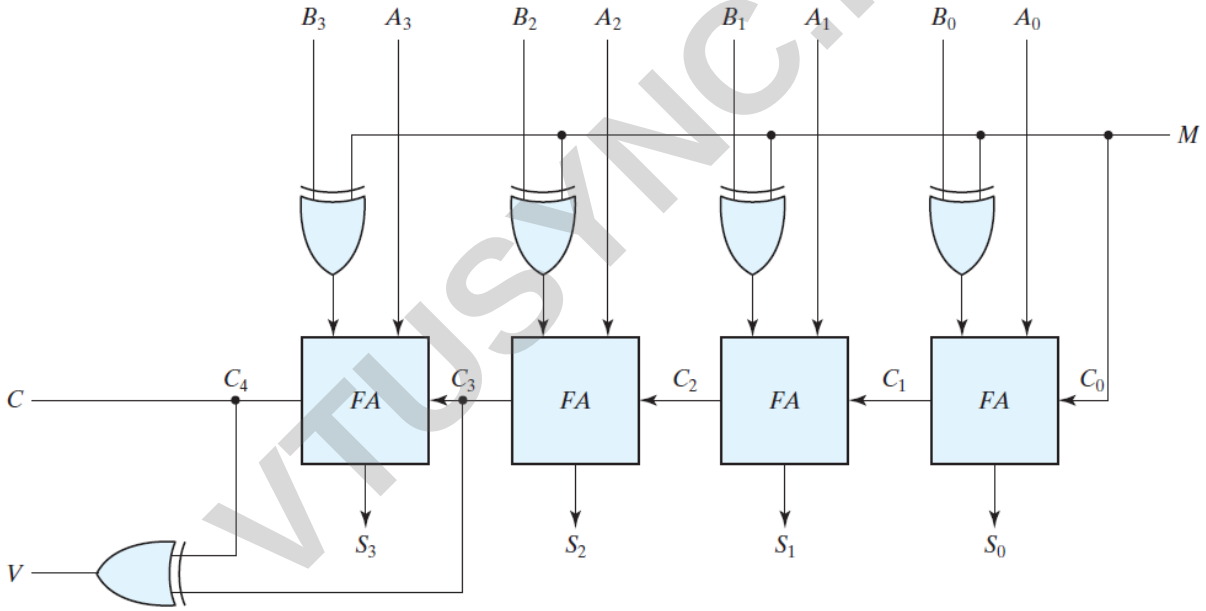


FIGURE 4.13
Four-bit adder–subtractor (with overflow detection)

Overflow

When two numbers with **n digits** each are added and the **sum is** a number occupying **n + 1 digits**, we say that an **overflow** occurred. This is true for binary or decimal numbers, signed or unsigned. Many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set that can then be checked by the user.

The **detection of an overflow** after the addition of two binary numbers depends on whether the numbers are signed or unsigned. When two **unsigned numbers are added**, an overflow is detected from the **end carry out** of the most significant position.

In the case of **signed numbers**, two details are important: the leftmost bit always represents the sign, and negative numbers are in 2's-complement form. When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result whose magnitude is smaller than the larger of the two original numbers.

An overflow may occur if the two numbers added are both positive or both negative. To see how this can happen, consider the following **example:** Two signed binary numbers, +70 and +80, are stored in two eight-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary -128. Since the sum of the two numbers is +150, it exceeds the capacity of an eight-bit register.

carries:	0 1	carries:	1 0
+70	0 1000110	−70	1 0111010
+80	0 1010000	−80	1 0110000
+150	1 0010110	−150	0 1101010

Note that the eight-bit result that should have been positive has a negative sign bit (i.e., the eighth bit) and the eight-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, then the nine-bit answer so obtained will be correct. But since the answer cannot be accommodated within eight bits, we say that an overflow has occurred.

An **overflow condition can be detected** by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow has occurred. This is indicated in the examples in which the two carriers are explicitly shown. If the two carriers are applied to an exclusive-OR gate, an overflow is detected when the output of the gate is equal to 1.

The binary adder–subtractor circuit with outputs C and V is shown in **Fig. 4.13** . If the two binary numbers are considered to be unsigned, then the C bit detects a carry after addition or a borrow after subtraction. If the numbers are considered to be signed, then the V bit detects an overflow. If **V = 0 after an addition or subtraction, then no overflow** occurred and the n -bit result is correct**. If V = 1, then the result of the operation contains n + 1 bits**, but only

the rightmost n bits of the number fit in the space available, so an **overflow has occurred**. The (n+1) th bit is the actual sign and has been shifted out of position.

4.9 DECODERS

Discrete quantities of information are represented in digital systems by **binary codes**. A binary code of n bits is capable of representing up to 2^n distinct elements of coded information. A decoder is a **combinational circuit** that converts binary information **from n input lines to a maximum of 2^n unique output lines**. If the n -bit coded information has unused combinations, the decoder may have fewer than 2^n outputs.

The decoders presented here are called n -to- m -line decoders, where $m \leq 2^n$.Their purpose is to **generate the 2^n** (or fewer) minterms of **n input** variables. Each combination of inputs will assert a unique output..

As an **example,** consider the **three-to-eight-line decoder** circuit of **Fig. 4.18** . The three inputs are decoded into eight outputs, each representing one of the minterms of the three input variables. The three inverters provide the complement of the inputs, and each one of the eight AND gates generates one of the minterms. A particular application of this decoder is **binary-to-octal** conversion. The input variables represent a binary number, and the outputs represent the eight digits of a number in the octal number system. However, a three-to-eight-line decoder can be used for **decoding any three-bit code to provide eight outputs**, one for each element of the code.

The **operation of the decoder** may be clarified by the truth table listed in **Table 4.6** .For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1. The output whose value is equal to 1 represents the minterm equivalent of the binary number currently available in the input lines.

Table 4.6
Truth Table of a Three-to-Eight-Line Decoder

Inputs			Outputs							
x	*y*	*z*	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

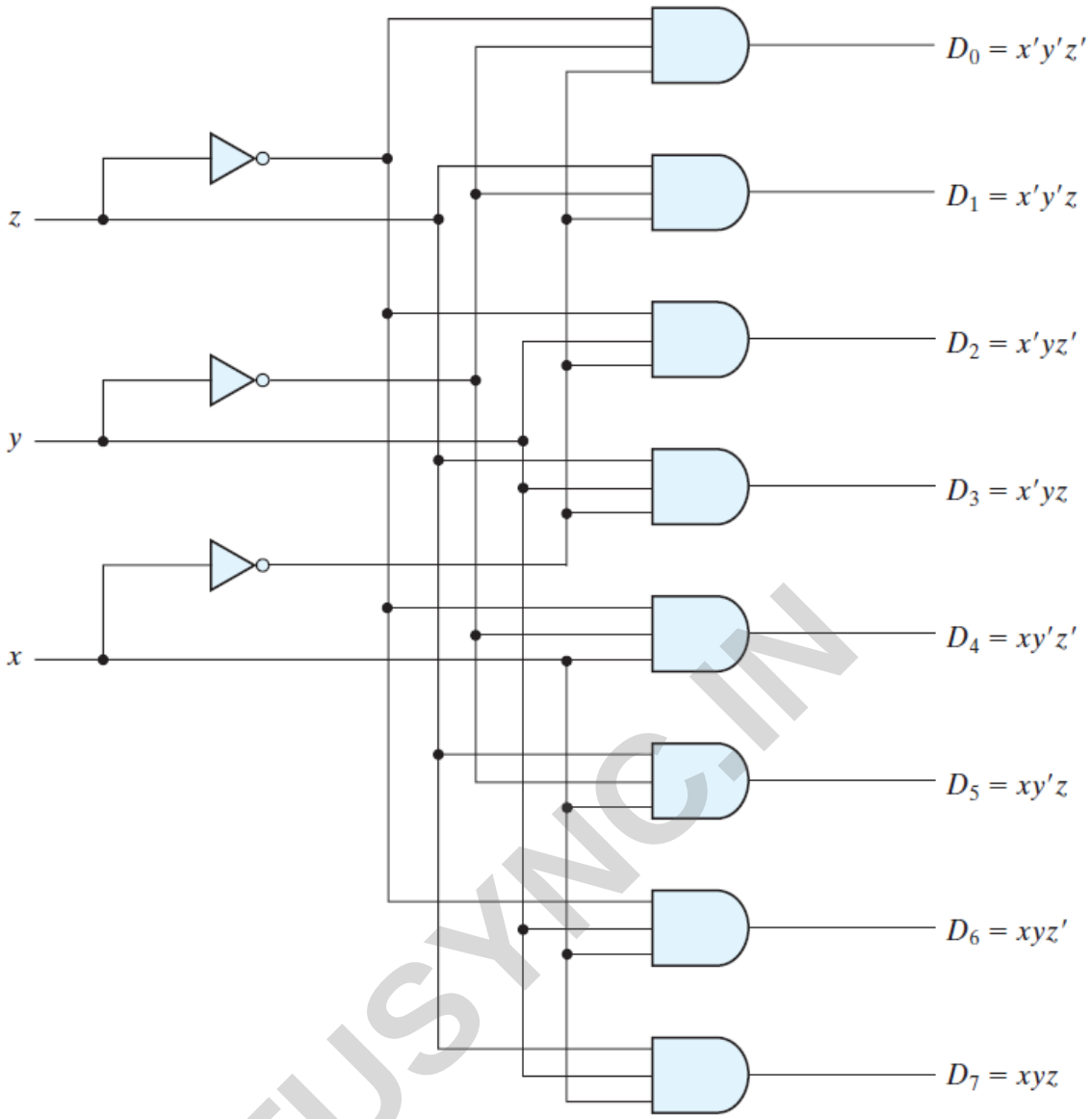
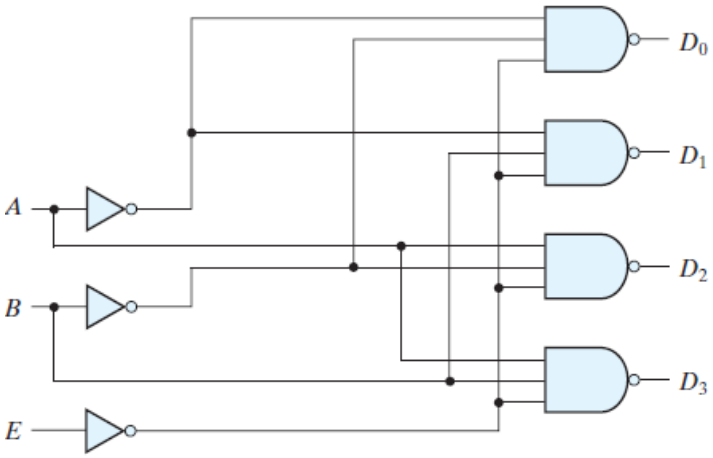


FIGURE 4.18
Three-to-eight-line decoder

Some decoders are constructed with NAND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder minterms in their complemented form. Furthermore, decoders include one or more **enable inputs to control the circuit operation**. A **two-to-four-line decoder** with an enable input constructed with NAND gates is shown in **Fig. 4.19 .**

The circuit operates with complemented outputs and a complement enable input. The decoder is enabled when E is equal to 0 (i.e., active-low enable). As indicated by the truth table, only one output can be equal to 0 at any given time; all other outputs are equal to 1. The output whose value is equal to **0 represents the minterm selected by inputs A and B**.

The circuit is disabled when E is equal to 1, regardless of the values of the other two inputs. When the circuit is disabled, none of the outputs are equal to 0 and none of the minterms are selected.



(a) Logic diagram

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	1

(b) Truth table

FIGURE 4.19
Two-to-four-line decoder with enable input

A decoder with enable input can function as a **demultiplexer**— a circuit that receives information from a single line and directs it to one of 2^n possible output lines. The selection of a specific output is controlled by the bit combination of n selection lines.

The decoder of **Fig. 4.19** can function as a **one-to-four-line demultiplexer** when E is taken as a data input line and A and B are taken as the selection inputs. The single input variable E has a path to all four outputs, but the input information is directed to only one of the output lines, as specified by the binary combination of the two selection lines A and B.

This feature can be verified from the truth table of the circuit. For example, if the selection lines AB = 10, output D2 will be the same as the input value E, while all other outputs are maintained at 1. Because decoder and demultiplexer, operations are obtained from the same circuit, a decoder with an enable input is referred to as a decoder – demultiplexer.

Decoders with enable inputs can be connected together to form a larger decoder circuit. **Figure 4.20** shows two **3-to-8-line decoders** with enable inputs connected to form a 4-to-16-line decoder. When **w = 0, the top decoder is enabled** and the other is disabled. The bottom decoder outputs are all 0's, and the top eight outputs generate minterms 0000 to 0111. When **w = 1**, the enable conditions are reversed: The bottom decoder outputs generate minterms 1000 to 1111, while the outputs of the top decoder are all 0's. This example demonstrates the usefulness of enable inputs in decoders and other combinational logic components.

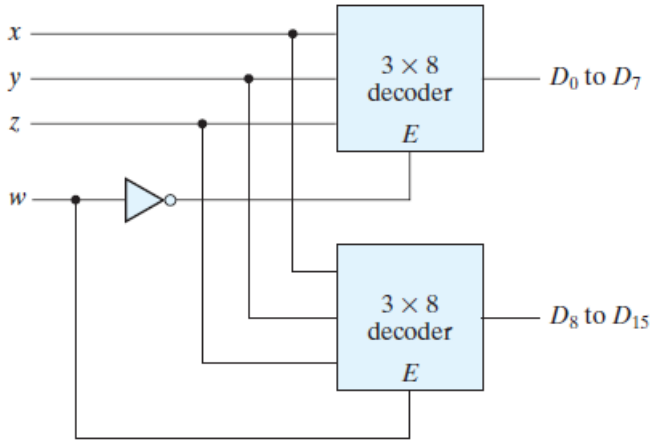


FIGURE 4.20
4×16 decoder constructed with two 3×8 decoders

Combinational Logic Implementation

A decoder provides the 2^n minterms of n input variables. Each asserted output of the decoder is associated with a unique pattern of input bits. Since any Boolean function can be expressed in sum-of-minterms form, a decoder that generates the **minterms** of the function, together with an **external OR gate** that forms their logical sum, provides a hardware implementation of the function. In this way, any combinational circuit with n inputs and m outputs can be implemented with an n -to-2^n -line decoder and m OR gates.

The procedure for implementing a combinational circuit by means of a **decoder and OR gates** requires that the Boolean function for the circuit be expressed as a sum of minterms. The inputs to each OR gate are selected from the decoder outputs according to the list of minterms of each function. This procedure will be illustrated by an **example that implements a full-adder circuit**.

From the **truth table of the full adder** (see **Table 4.4**), we obtain the functions for the combinational circuit in sum-of-minterms form:

$$S(x, y, z) = \Sigma(1, 2, 4, 7)$$

$$C(x, y, z) = \Sigma(3, 5, 6, 7)$$

Since there are three inputs and a total of eight minterms, we need a three-to-eight-line decoder. The implementation is shown **in Fig. 4.21** . The decoder generates the eight minterms for x , y , and z . The OR gate for output S forms the logical sum of minterms 1,2, 4, and 7. The OR gate for output C forms the logical sum of minterms 3, 5, 6, and 7.

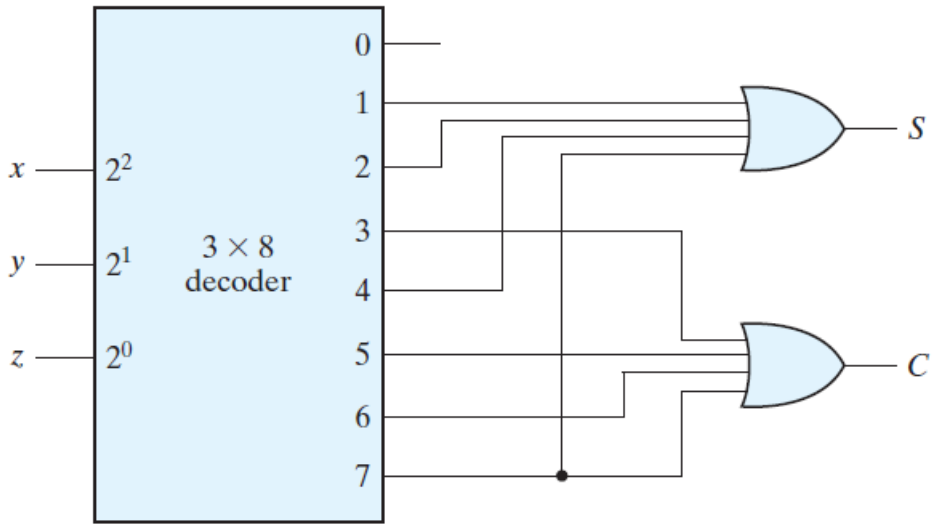


FIGURE 4.21
Implementation of a full adder with a decoder

A **function with a long list of minterms** requires an OR gate with a large number of inputs. A function having a list of k minterms can be expressed in its complemented form F' with 2^n - k minterms. If the number of minterms in the function is greater than $2^n/2$, then F' can be expressed with fewer minterms. In such a case, it is advantageous to **use a NOR gate to sum the minterms of F'**. The output of the NOR gate complements this sum and generates the normal output F. **If NAND gates are used for the decoder**, as in **Fig. 4.19**, then the external gates must be NAND gates instead of OR gates. This is because **a two-level NAND gate** circuit implements a sum-of-minterms function and is **equivalent to a two-level AND–OR** circuit.

4.10 ENCODERS

An encoder is a digital circuit that performs the **inverse operation of a decoder**. An encoder has **2^n (or fewer) input lines and n output lines**. The output lines, as an aggregate, generate the binary code corresponding to the input value. An example of an encoder is the octal-to-binary encoder whose truth table is given in **Table 4.7 .** It has eight inputs (one for each of the octal digits) and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time.

The **encoder can be implemented with OR gates** whose inputs are determined directly from the truth table. Output z is equal to 1 when the input octal digit is 1, 3, 5,or 7. Output y is 1 for octal digits 2, 3, 6, or 7, and output x is 1 for digits 4, 5, 6, or 7. These conditions can be expressed by the following Boolean output functions:

$$z = D_1 + D_3 + D_5 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$x = D_4 + D_5 + D_6 + D_7$$

Table 4.7
Truth Table of an Octal-to-Binary Encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

The encoder can be implemented with three OR gates.

Priority Encoder

A priority encoder is an encoder circuit that **includes the priority function**. The operation of the priority encoder is such that if two or **more inputs are equal to 1** at the same time, the **input having the highest priority will take precedence**.

The truth table of a four-input priority encoder is given in **Table 4.8** . In addition to the two outputs x and y , the circuit has a **third output designated by V** ; this is a valid bit indicator that is set to 1 when one or more inputs are equal to 1. If **all inputs are 0**, there is no valid input and **V is equal to 0**. The other two outputs are not inspected when V equals 0 and are **specified as don't-care conditions**. Note that whereas X 's in output columns represent don't-care conditions, the X 's in the input columns are useful for representing either 1 or 0. For **example**, X 100 represents the two minterms 0100 and 1100.

Table 4.8
Truth Table of a Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

According to **Table 4.8** , the higher the subscript number, the higher the priority of the input. Input D3 has the highest priority, so, regardless of the values of the other inputs, when this input is 1, the output for xy is 11 (binary 3). D2 has the next priority level. The output is 10 if D2 = 1, provided that D3 = 0, regardless of the values of the other two lower priority inputs. The output for D1 is generated only if higher priority inputs are 0, and so on down the priority levels.

The maps for simplifying outputs x and y are shown in **Fig. 4.22** . The minterms for the two functions are derived from **Table 4.8**. Although the table has only five rows, when each X in a row is replaced first by 0 and then by 1, we obtain all 16 possible input combinations.

For example, the fourth row in the table, with inputs XX10, represents the four minterms 0010, 0110, 1010, and 1110. The simplified Boolean expressions for the priority encoder are obtained from the maps. The condition for output V is an OR function of all the input variables.

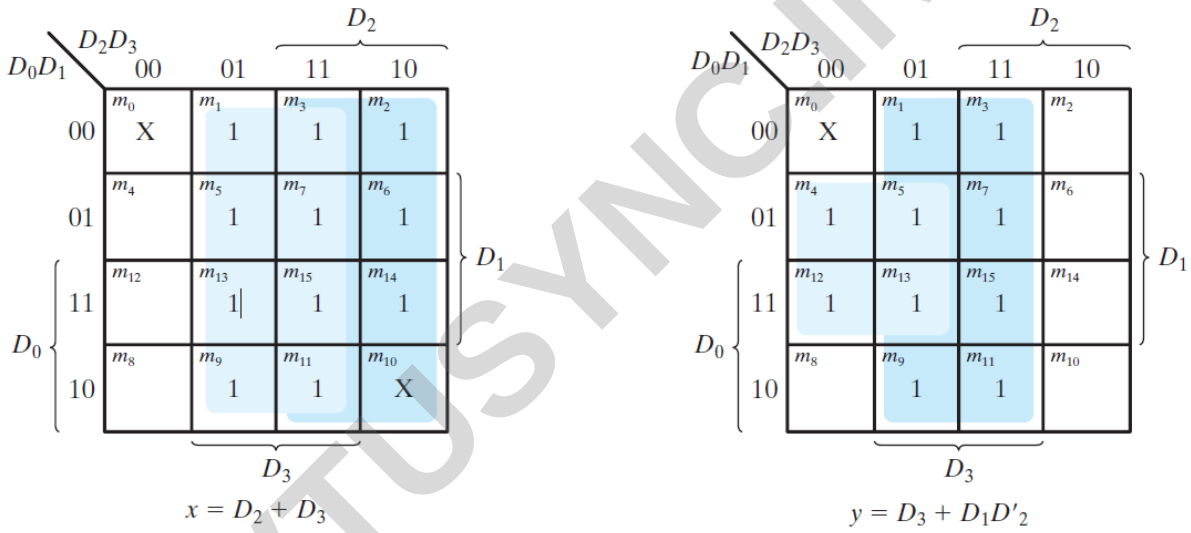


FIGURE 4.22
Maps for a priority encoder

$$x = D_2 + D_3$$
$$y = D_3 + D_1 D_2'$$
$$V = D_0 + D_1 + D_2 + D_3$$

The priority encoder is implemented in **Fig. 4.23** according to the following Boolean functions:

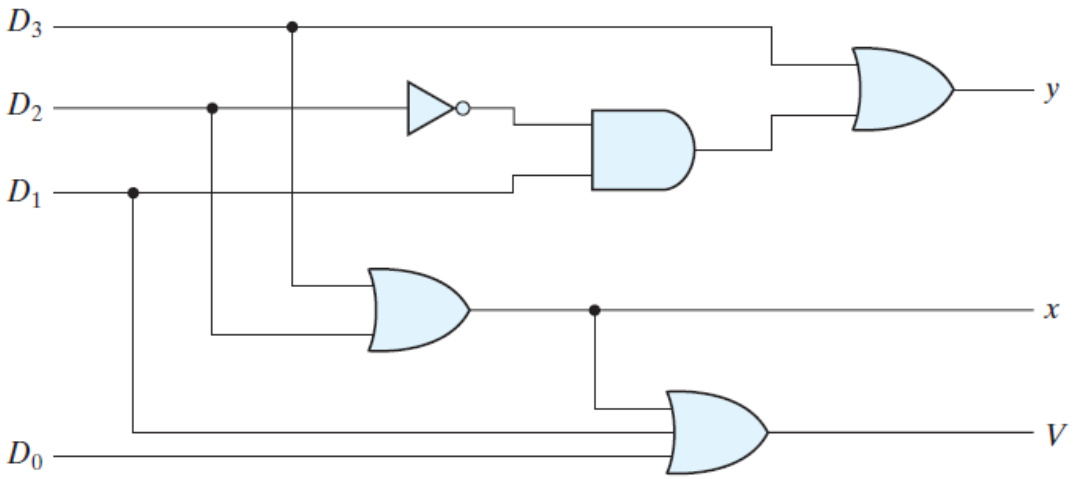


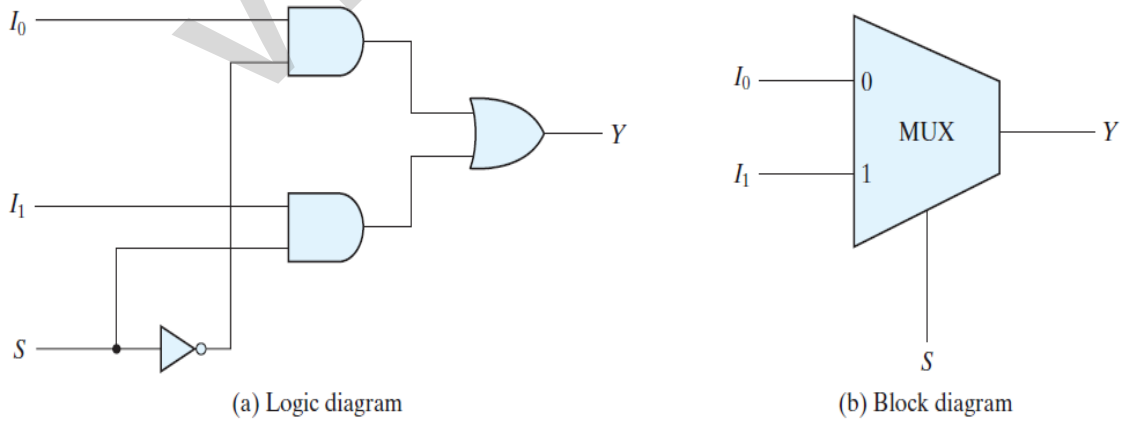
FIGURE 4.23
Four-input priority encoder

4.11 MULTIPLEXERS

A multiplexer is a combinational circuit that selects binary information from one of **many input lines and directs it to a single output line**. The selection of a particular input line is controlled by a set of **selection lines**.

A two-to-one-line multiplexer connects one of two 1-bit sources to a common destination, as shown in **Fig. 4.24** . The circuit has two data input lines, one output line, and one selection line S. When S = 0, the upper AND gate is enabled and I_0 has a path to the output. When S = 1, the lower AND gate is enabled and I_1 has a path to the output.

The multiplexer acts like an electronic switch that selects one of two sources. The block diagram of a multiplexer is sometimes depicted by a wedge-shaped symbol, as shown in **Fig. 4.24(b)**.



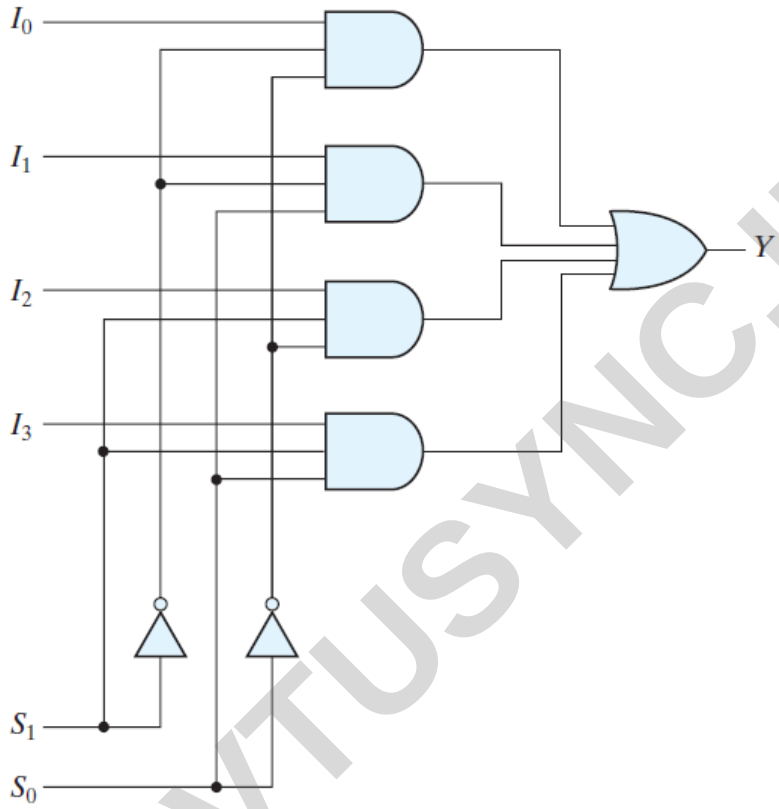
(a) Logic diagram (b) Block diagram

FIGURE 4.24
Two-to-one-line multiplexer

A **four-to-one-line multiplexer** is shown in **Fig. 4.25**. Each of the four inputs, I_0 through I_3, is applied to one input of an AND gate. Selection lines S1 and S0 are decoded to

select a particular AND gate. The outputs of the AND gates are applied to a single OR gate that provides the one-line output.

The function table lists the input that is passed to the output for each combination of the binary selection values. To demonstrate the operation of the circuit, consider the case when $S_1S_0 = 10$. The AND gate associated with input I_2 has two of its inputs equal to 1 and the third input connected to I_2. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The output of the OR gate is now equal to the value of I_2, providing a path from the selected input to the output. A multiplexer is also called a **data selector**, since it selects one of many inputs and steers the binary information to the output line.



(a) Logic diagram

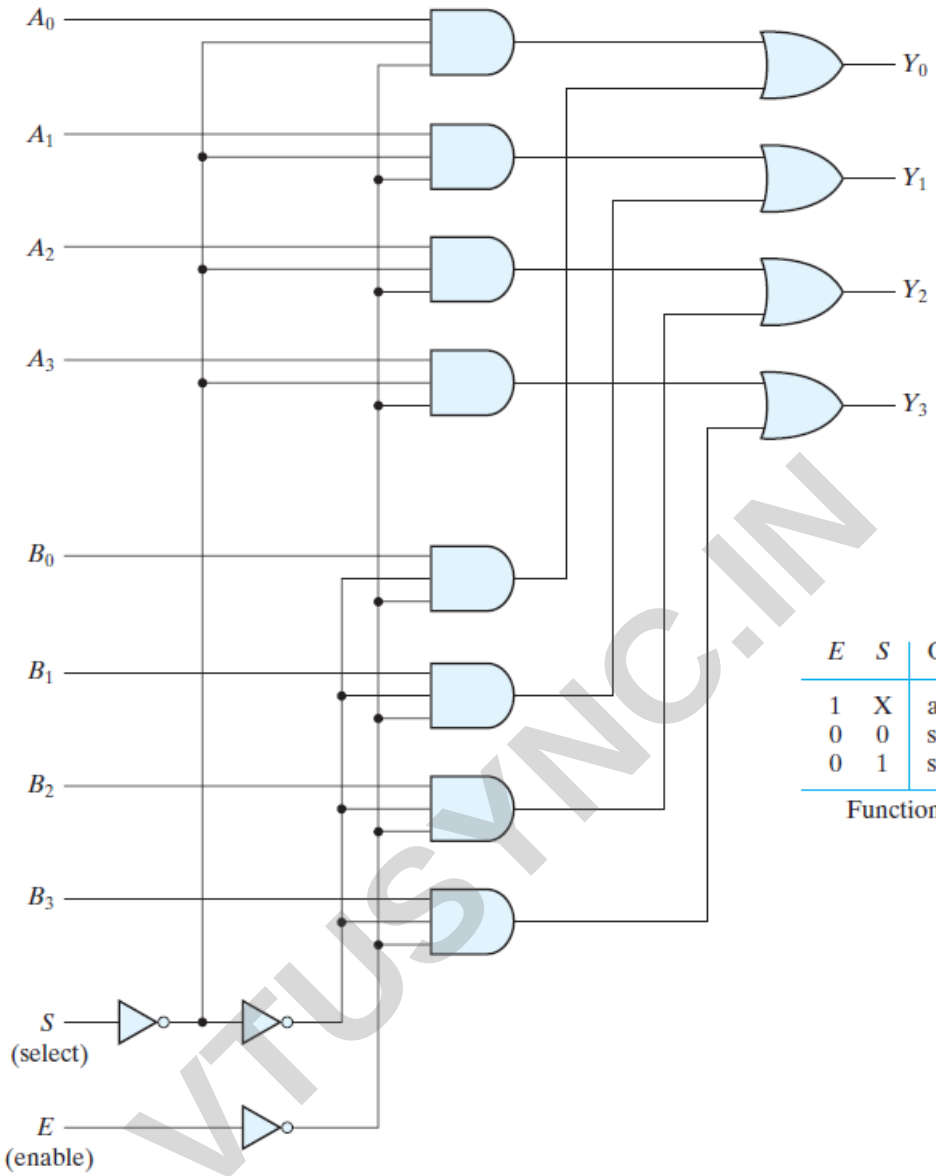
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

FIGURE 4.25
Four-to-one-line multiplexer

Multiplexer circuits can be combined with common selection inputs to provide multiple-bit selection logic. As an illustration, **a quadruple 2-to-1-line multiplexer** is shown in **Fig. 4.26**. The circuit has four multiplexers, each capable of selecting one of two input lines. Output Y_0 can be selected to come from either input A_0 or input B_0. Similarly, output Y_1 may have the value of A_1 or B_1, and so on. Input selection line S selects one of the lines in each of the four multiplexers. The enable input E must be active (i.e., asserted) for normal operation. Although the circuit contains four 2-to-1-line multiplexers, we are more likely to view it as a circuit that selects one of two 4-bit sets of data lines. As shown in the function table, the unit is

enabled when E = 0. Then, if S = 0, the four A inputs have a path to the four outputs. If, by contrast, S = 1, the four B inputs are applied to the outputs. The outputs have all 0's when E = 1, regardless of the value of S.



E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

Function table

FIGURE 4.26
Quadruple two-to-one-line multiplexer

Boolean Function Implementation

The minterms of a function are generated in a multiplexer by the circuit associated with the selection inputs. The individual minterms can be selected by the data inputs, thereby providing a method of implementing a Boolean function of n variables with a multiplexer that has n selection inputs and 2^n data inputs, one for each minterm.

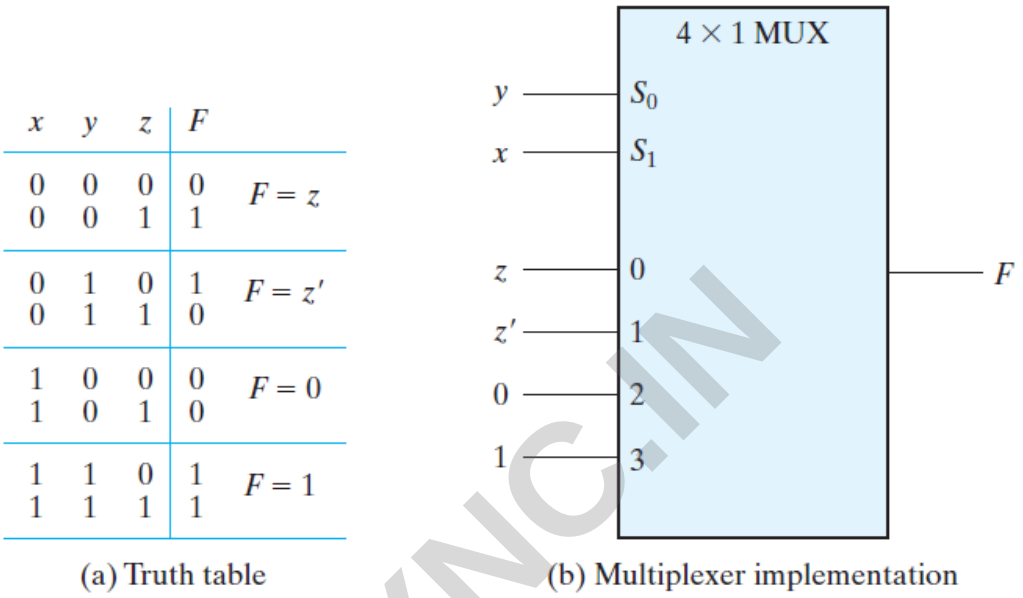
We will now show a more efficient method for implementing a Boolean function of n variables with a multiplexer that has n - 1 selection inputs. The first n - 1 variables of the function are connected to the selection inputs of the multiplexer. The remaining single variable

of the function is used for the data inputs. If the single variable is denoted by z, each data input of the multiplexer will be z, z', 1, or 0.

To demonstrate this procedure, consider the Boolean function

$$F(x, y, z) = \Sigma(1, 2, 6, 7)$$

This function of three variables can be implemented with a four-to-one-line multiplexer as shown in **Fig. 4.27**



x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table

(b) Multiplexer implementation

FIGURE 4.27
Implementing a Boolean function with a multiplexer

The two variables x and y are applied to the selection lines in that order; x is connected to the S1 input and y to the S_0 input. The values for the data input lines are determined from the truth table of the function. When xy = 00, output F is equal to z because F = 0 when z = 0 and F = 1 when z = 1.

In a similar fashion, we can determine the required input to data lines 1, 2, and 3 from the value of F when xy = 01, 10, and 11, respectively. This particular example shows all four possibilities that can be obtained for the data inputs.

As a **second example**, consider the implementation of the Boolean function

$$F(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$$

This function is implemented with a multiplexer with three selection inputs as shown in **Fig. 4.28** . Note that the first variable A must be connected to selection input S2 so that A , B, and C correspond to selection inputs S2, S1, and S0, respectively. The values for the data inputs are determined from the truth table listed in the figure. The corresponding data line number is determined from the binary combination of ABC . For example, the table shows that

when ABC = 101, F = D, so the input variable D is applied to data input 5. The binary constants 0 and 1 correspond to two fixed signal values.

A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

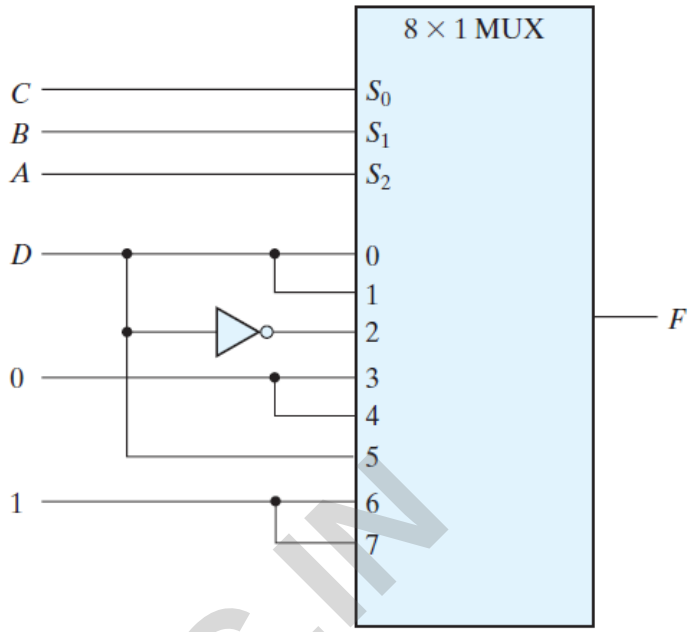


FIGURE 4.28
Implementing a four-input function with a multiplexer

Three-State Gates

A multiplexer can be constructed with three-state gates—digital circuits that exhibit three states. **Two of the states** are signals equivalent to **logic 1 and logic 0** as in a conventional gate.

The **third state is a high-impedance state** in which (1) the logic behaves like an open circuit, which means that the output appears to be disconnected, (2) the circuit has no logic significance, and (3) the circuit connected to the output of the three-state gate is not affected by the inputs to the gate.

Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used is the **buffer gate**.

The graphic symbol for a three-state buffer gate is shown in **Fig. 4.29**

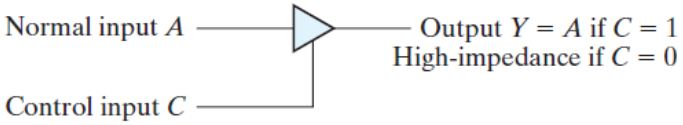


FIGURE 4.29
Graphic symbol for a three-state buffer

The buffer has a normal input, an output, and a **control input that determines the state of the output**. When the control input is equal to 1, output equal to the normal input.

When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input.

The construction of multiplexers with three-state buffers is demonstrated in **Fig. 4.30**.**Figure 4.30(a)** shows the construction of a **two-to-one-line multiplexer** with 2 three-state buffers and an inverter. The two outputs are connected together to form a single output line. When the select input is 0, the upper buffer is enabled by its control input and the lower buffer is disabled. Output Y is then equal to input A . When the select input is 1, the lower buffer is enabled and Y is equal to B .

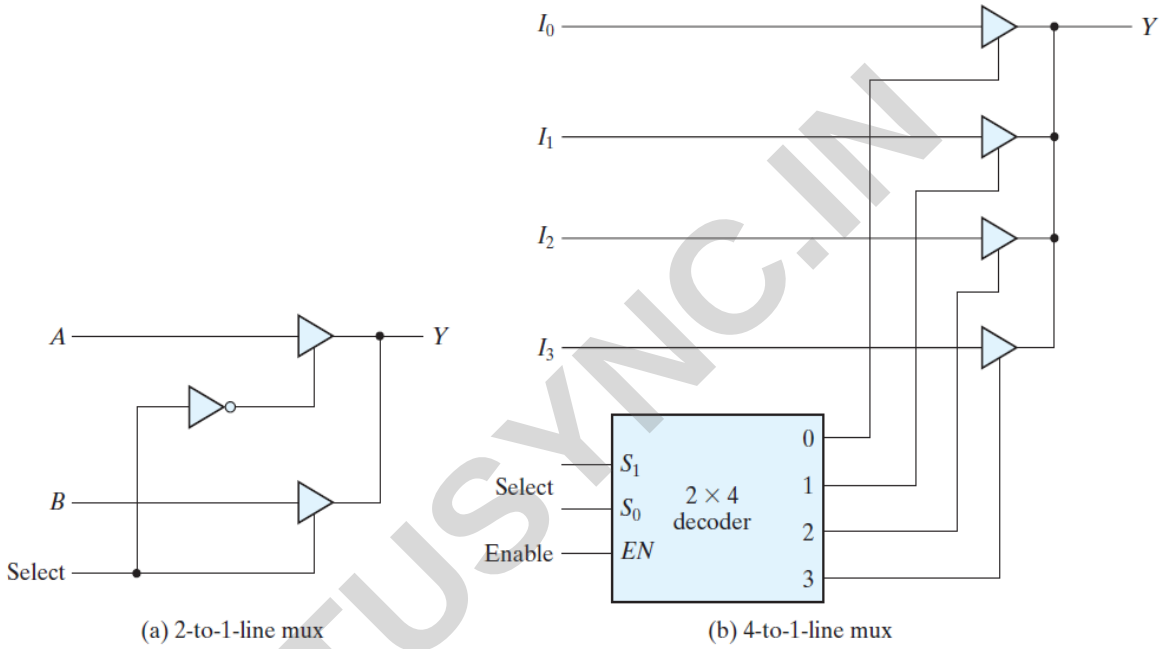


FIGURE 4.30
Multiplexers with three-state gates

The construction of a four-to-one-line multiplexer is shown in **Fig. 4.30(b).** The outputs of 4 three-state buffers are connected together to form a single output line. The control inputs to the buffers determine which one of the four normal inputs I0 through I3 will be connected to the output line, while all other buffers are maintained in a high impedance state.

One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs are 0 and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three state buffers will be active.

4.12 HDL MODELS OF COMBINATIONAL CIRCUITS

The module is the basic building block for modelling hardware with the Verilog HDL. The logic of a module can be described in any one (or a combination) of the following modeling styles:

- **Gate-level modeling** using instantiations of predefined and user-defined primitive gates.
- **Dataflow modeling** using continuous assignment statements with the keyword **assign**.
- **Behavioral modeling** using procedural assignment statements with the keyword **always**.

- Gate-level (structural) modeling describes a circuit by specifying its gates and how they are connected with each other.
- Dataflow modeling is used mostly for describing the Boolean equations of combinational logic.
- Behavioural modelling that is used to describe combinational and sequential circuits at a higher level of abstraction. Combinational logic can be designed with truth tables, Boolean equations, and schematics;
- Verilog has a construct corresponding to each of these "classical" approaches to design: user-defined primitives, continuous assignments, and primitives, as shown in **Fig. 4.31.**

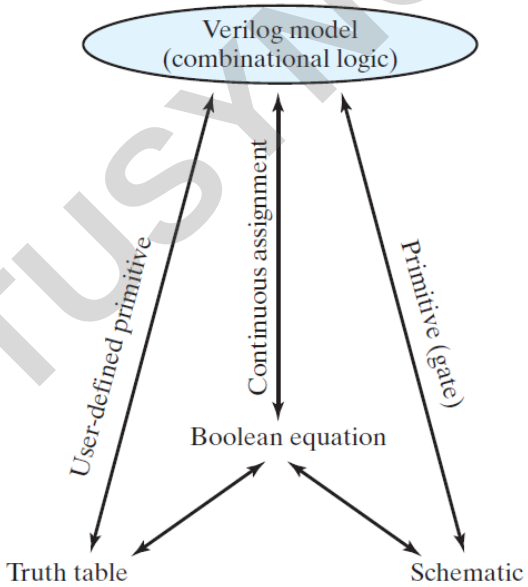


FIGURE 4.31
Relationship of Verilog constructs to truth tables, Boolean equations, and schematics

Gate-Level Modeling

- A circuit is specified by its **logic gates** and their interconnections. Gate level modeling provides a textual description of a schematic diagram. The Verilog HDL includes **12 basic gates** as predefined primitives. Four of these primitive gates are of the **three-state** type.

They are all declared with the **lowercase** keywords **and, nand, or, nor, xor, xnor, not,** and **buf**. Primitives such as **and** are n -input primitives. They can have any number of

scalar inputs (e.g., a three-input **and** primitive). The **buf** and **not** primitives are n -output primitives. A single input can drive multiple output lines distinguished by their identifiers.

The Verilog language includes a functional description of each type of gate, too. The logic of each gate is based on a four-valued system. When the gates are simulated, the simulator assigns one value to the output of each gate at any instant. In addition to the two logic values of **0 and 1**, there are two other values: **unknown and high impedance**. An unknown value is **denoted by x and a high impedance by z** . An unknown value is assigned during simulation when the logic value of a signal is ambiguous, A high-impedance condition occurs at the output of three-state gates that are not enabled. The four-valued logic truth tables for the **and, or, xor,** and **not** primitives are shown in **Table 4.9**.

Note that for the **and gate**, the output is 1 only when both inputs are 1 and the output is 0 if any input is 0. Otherwise, if one input is x or z , the output is x . The output of the **or gate** is 0 if both inputs are 0, is 1 if any input is 1, and is x otherwise.

Table 4.9
Truth Table for Predefined Primitive Gates

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

not	input	output
	0	1
	1	0
	x	x
	z	x

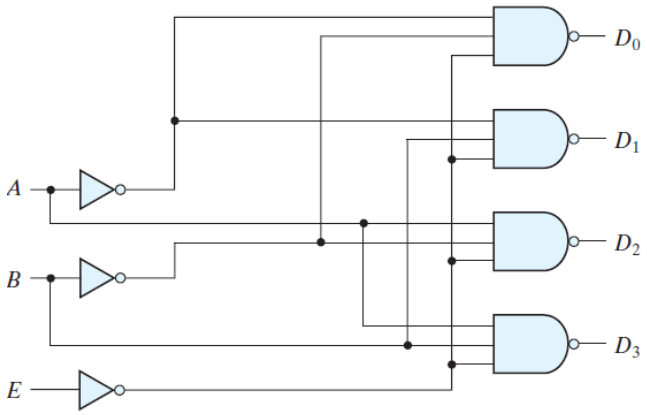
We now present two examples of gate-level modeling. Both examples use identifiers having multiple bit widths, called **vectors** .

The following Verilog statements specify two vectors:

```
output [0: 3] D;
wire   [7: 0] SUM;
```

The first statement declares an output vector D with four bits, 0 through 3. The second declares a wire vector SUM with eight bits numbered 7 through 0.

HDL **Example 4.1** shows the gate-level description of a **two-to-four-line decoder. (See Fig. 4.19 .)**



(a) Logic diagram

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) Truth table

FIGURE 4.19
Two-to-four-line decoder with enable input

This decoder has two data inputs A and B and an enable input E. The four outputs are specified with the vector D.

The **wire** declaration is for internal connections. Three **not gates** produce the complement of the inputs, and four **nand gates** provide the outputs for D .

HDL Example 4.1 (Two-to-Four-Line Decoder)

```
// Gate-level description of two-to-four-line decoder
// Refer to Fig. 4.19 with symbol E replaced by enable, for clarity.

module decoder_2x4_gates (D, A, B, enable);
 output      [0: 3]      D;
 input                   A, B;
 input                   enable;
 wire                    A_not,B_not, enable_not;

 not
   G1  (A_not, A),
   G2  (B_not, B),
   G3  (enable_not, enable);
 nand
   G4  (D[0], A_not, B_not, enable_not),
   G5  (D[1], A_not, B, enable_not),
   G6  (D[2], A, B_not, enable_not),
   G7  (D[3], A, B, enable_not);
endmodule
```

Two or more **modules can be combined** to build a hierarchical description of a design. There are two basic types of design methodologies: **top down and bottom up**.

In a **top-down design**, the top-level block is defined and then the subblocks necessary to build the top-level block are identified.

In a **bottom-up design**, the building blocks are first identified and then combined to build the top-level block, for example, the **binary adder**. It can be considered as a top-block component built with four full-adder blocks, while each full adder is built with two half-adder blocks. In a top-down design, the four-bit adder is defined first, and then the two adders are described. In a bottom-up design, the half adder is defined, then each full adder is constructed, and then the four-bit adder is built from the full adders.

HDL Example 4.2 (Ripple-Carry Adder)

```
// Gate-level description of four-bit ripple carry adder
// Description of half adder (Fig. 4.5b)

// module half_adder (S, C, x, y);                    // Verilog 1995 syntax
// output  S, C;
// input   x, y;

module half_adder (output S, C, input x, y);          // Verilog 2001, 2005 syntax
// Instantiate primitive gates
  xor (S, x, y);
  and (C, x, y);
endmodule

// Description of full adder (Fig. 4.8)               // Verilog 1995 syntax
// module full_adder (S, C, x, y, z);
// output          S, C;
// input           x, y, z;

module full_adder (output S, C, input x, y, z);       // Verilog 2001, 2005 syntax
  wire S1, C1, C2;

// Instantiate half adders
  half_adder HA1 (S1, C1, x, y);
  half_adder HA2 (S, C2, S1, z);
  or G1 (C, C2, C1);
endmodule

// Description of four-bit adder (Fig. 4.9)       // Verilog 1995 syntax
// module ripple_carry_4_bit_adder (Sum, C4, A, B, C0);
// output [3: 0]   Sum;
// output          C4;
// input  [3: 0]   A, B;
// input           C0;
// Alternative Verilog 2001, 2005 syntax:

module ripple_carry_4_bit_adder ( output [3: 0] Sum, output C4,
  input [3: 0] A, B, input C0);
  wire          C1, C2, C3;          // Intermediate carries
// Instantiate chain of full adders
full_adder      FA0 (Sum[0], C1, A[0], B[0], C0),
                FA1 (Sum[1], C2, A[1], B[1], C1),
                FA2 (Sum[2], C3, A[2], B[2], C2),
                FA3 (Sum[3], C4, A[3], B[3], C3);
endmodule
```


Three-State Gates

o A three-state gate has a control input that can place the gate into a high-impedance state. The high-impedance state is symbolized by **z** in Verilog.

- There are **four types** of three-state gates, as shown in **Fig. 4.32**. The **bufif1** gate behaves like a normal buffer if *control* = 1. The output goes to a high-impedance state **z** when *control* = 0.
- The **bufif0** gate behaves in a similar fashion, except that the high-impedance state occurs when *control* = 1.
- The two **notif** gates operate in a similar manner, except that the output is the complement of the input when the gate is not in a high-impedance state. The gates are instantiated with the statement.

***gatename* (*output*, *input*, *control*);**

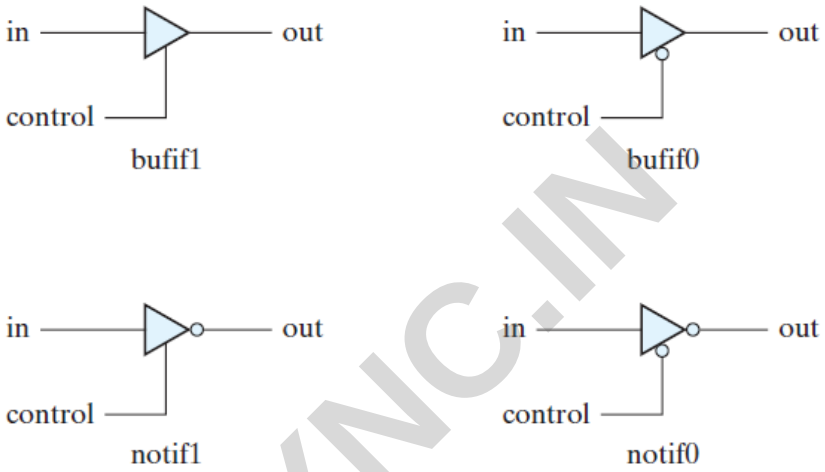


FIGURE 4.32
Three-state gates

The gate name can be that of any 1 of the 4 three-state gates. In simulation, the output can result in 0, 1, x , or z . Two examples of gate instantiation are

```
bufif1   (OUT, A, control);
notif0   (Y, B, enable);
```

In the first example, input A is transferred to OUT when control = 1. OUT goes to z when control = 0. In the second example, output Y = z when enable = 1 and output Y = B' when enable = 0.

The outputs of three-state gates can be connected together to form a common output line. To identify such a connection, Verilog HDL uses the keyword **tri** (for tristate) to indicate that the output has multiple drivers. As an example, consider the **two-to-one line multiplexer** with three-state gates shown in **Fig. 4.33** .

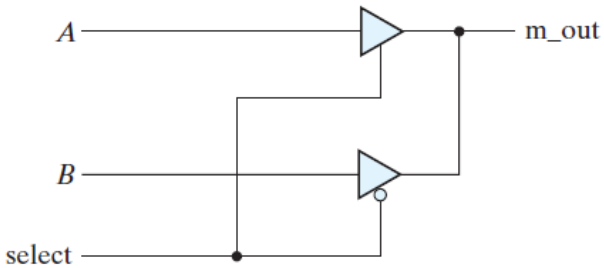


FIGURE 4.33
Two-to-one-line multiplexer with three-state buffers

```
// Mux with three-state output

module mux_tri (m_out, A, B, select);
  output  m_out;
  input   A, B, select;
  tri     m_out;

  bufif1 (m_out, A, select);
  bufif0 (m_out, B, select);
endmodule
```

The 2 three-state buffers have the same output. In order to show that they have a common connection, it is necessary to declare m_out with the keyword tri.

Keywords **wire** and **tri** are examples of a set of data types called **nets** , which represent connections between hardware elements.

The word **net** is not a keyword, but represents a class of data types, such as **wire , wor, wand, tri,supply1, and supply0.**

The wire declaration is used most frequently. In fact, if an identifier is used, but not declared, the language specifies that it will be interpreted (by default) as a wire . The net **wor** models the hardware implementation of the **wired-OR** configuration (emitter-coupled logic). The **wand** models the **wired-AND** configuration (open-collector technology; see Fig. 3.26). The nets **supply1** and **supply0** represent power supply and ground, respectively. They are used to hardwire an input of a device to either 1 or 0.

Dataflow modeling of combinational logic uses a number of operators that act on binary operands to produce a binary result. Verilog HDL provides about 30 different operators.**Table 4.10** lists some of these operators, their symbols, and the operation that they perform.

It should be noted that a bitwise operator (e.g., &) and its corresponding logical operator (e.g., !) may produce different results, depending on their operand. If the operands are scalar the results will be identical; if the operands are vectors the result will not necessarily match. For example, ~ **(1010)** is (0101), and **!(1010) is 0**. A binary value is considered to be logically true if it is not 0. In general, use the **bitwise operators to describe arithmetic operations** and the **logical operators to describe logical operations.**

Table 4.10
Some Verilog HDL Operators

Symbol	Operation	Symbol	Operation
+	binary addition		
–	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
= =	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

Dataflow modeling uses continuous assignments and the keyword **assign**. A continuous assignment is a statement that assigns a value to a net. The **data type family net is used in Verilog HDL to represent a physical connection between circuit elements**.

EXAMPLE : `assign Y = (A && S) || (B && S)`

The variables were declared, a two-to-one-line multiplexer with scalar data inputs A and B, select input S, and output Y is described with the continuous assignment.

The relationship between Y, A, B, and S is declared by the keyword assign, followed by the target output Y and an equals sign. Following the equals sign is a Boolean expression. In hardware terms, this assignment would be equivalent to connecting the output of the OR gate to wire Y .

EXAMPLE :

HDL Example 4.3 (Dataflow: Two-to-Four Line Decoder)

```
// Dataflow description of two-to-four-line decoder

// See Fig. 4.19. Note: The figure uses symbol E, but the
// Verilog model uses enable to clearly indicate functionality.

module decoder_2x4_df (                    // Verilog 2001, 2005 syntax
  output    [0: 3]        D,
  input                   A, B,
                          enable
);
  assign    D[0] = !((!A) && (!B) && (!enable)),
            D[1] = !(*!A) && B && (!enable)),
            D[2] = !(A && B && (!enable)
            D[3] = !(A && B && (!enable))
endmodule
```

The dataflow description of a **two-to-four-line** decoder with active-low output enable and inverted output is shown in HDL **Example 4.3**. The circuit is defined with four continuous assignment statements using Boolean expressions, one for each output.

EXAMPLE:

HDL Example 4.4 (Dataflow: Four-Bit Adder)

```
// Dataflow description of four-bit adder
// Verilog 2001, 2005 module port syntax
module binary_adder (
  output [3: 0]        Sum,
  output               C_out,
  input [3: 0]         A, B,
  input                C_in
);
  assign {C_out, Sum} = A + B + C_in;
endmodule
```

The dataflow description of the four-bit adder is shown in HDL **Example 4.4.** The addition logic is described by a single statement using the operators of addition and concatenation.

The plus symbol (+) specifies the binary addition of the four bits of A with the four bits of B and the one bit of Cin . The target output is the concatenation of the output carry Cout and the four bits of Sum . Concatenation of operands is expressed within braces and a comma separating the operands. Thus, {C_out, Sum} represents the five-bit result of the addition operation.

EXAMPLE:

HDL Example 4.5 (Dataflow: Four-Bit Comparator)

```
// Dataflow description of a four-bit comparator //V2001, 2005 syntax
module mag_compare
( output               A_lt_B, A_eq_B, A_gt_B,
  input [3: 0]         A, B
);
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (A = = B);
endmodule
```

To show how dataflow descriptions facilitate digital design, consider the 4-bit magnitude comparator described in HDL **Example 4.5**.

The module specifies two 4-bit inputs *A* and *B* and three outputs. One output (*A_lt_B*) is logic 1 if *A* is less than *B* , a second output (*A_gt_B*) is logic 1 if *A* is greater than *B* , and a third output (*A_eq_B*) is logic 1 if *A* is equal to *B* .

EXAMPLE: conditional operator (? :). This operator takes three operands:

condition ? true-expression : false-expression;

assign OUT = select ? A : B;

The condition is evaluated. If the result is logic 1, the true expression is evaluated and used to assign a value to the left-hand side of an assignment statement. If the result is logic 0, the false expression is evaluated. The two conditions together are equivalent to an if–else condition.

HDL **Example 4.6** describes a **two-to-one-line multiplexer** using the conditional operator.

HDL Example 4.6 (Dataflow: Two-to-One Multiplexer)

```
// Dataflow description of two-to-one-line multiplexer
module mux_2x1_df(m_out, A, B, select);
  output    m_out;
  input     A, B;
  input     select;
  assign m_out = (select)? A : B;
endmodule
```

Behavioral Modeling

Behavioral modeling represents digital circuits at a **functional and algorithmic level**. It is used mostly to describe sequential circuits, but can also be used to describe combinational circuits.

Behavioral descriptions use the keyword **always** , followed by an optional **event control expression** and a list of procedural **assignment statements**. The event control expression specifies when the statements will execute. The **target output** of a procedural assignment statement must be of the **reg data type**. Contrary to the wire data type, whereby the target output of an assignment may be continuously updated, a reg data type retains its value until a new value is assigned.

HDL Example 4.7 shows the behavioral description of a two-to-one-line multiplexer.

(Compare it with HDL Example 4.6.) Since variable m_out is a target output, it must be declared as reg data .The procedural assignment statements inside the always block are executed every time there is a change in any of the variables listed after the @ symbol. (Note that there is no semicolon (;) at the end of the always statement.)

The input variables A , B, and select. The statements execute if A, B , or select changes value. Note that the keyword **or** , instead of the bitwise logical OR operator "|", is used between variables.

HDL Example 4.7 (Behavioral: Two-to-One Line Multiplexer)

```
// Behavioral description of two-to-one-line multiplexer

module mux_2x1_beh (m_out, A, B, select);
  output      m_out;
  input       A, B, select;
  reg         m_out;

  always      @(A or B or select)
   if (select == 1) m_out = A;
   else m_out 5 B;
endmodule
```

HDL **Example 4.8** describes the function of a **four-to-one-line multiplexer**. The **select** input is defined as a **two-bit vector**, and output **m_out** is declared to have type **reg** . The always statement, in this example, has a sequential block enclosed between the keywords **case** and **endcase** . The block is executed whenever any of the inputs listed after the @ symbol changes in value. The **case** statement is a **multiway conditional** branch construct. Since **select** is a two-bit number, it can be equal to **00, 01, 10, or 11**.Unlisted **case** items, treated by using the **default** keyword as the last item in the list of **case** items.

HDL Example 4.8 (Behavioral: Four-to-One Line Multiplexer)

```
// Behavioral description of four-to-one line multiplexer

// Verilog 2001, 2005 port syntax

module mux_4x1_beh
( output reg m_out,
  input      in_0, in_1, in_2, in_3,
  input [1: 0] select
);
always @ (in_0, in_1, in_2, in_3, select)      // Verilog 2001, 2005 syntax
  case (select)
    2'b00:          m_out = in_0;
    2'b01:          m_out = in_1;
    2'b10:          m_out = in_2;
    2'b11:          m_out = in_3;
  endcase
endmodule
```

Binary numbers in Verilog are specified and interpreted with the **letter b** preceded by a prime. The size of the number is written first and then its value. Thus, **2_b01** specifies a two-bit binary number whose value is 01.

If the **base** of the number is not specified,its interpretation **defaults to decimal**. If the **size** of the number is not specified, the system assumes that the size of the number is at least **32 bits**;

Writing a Simple Test Bench

- A test bench is an HDL program used for describing and applying a stimulus to an HDL model of a circuit in order to test it and observe its response during simulation.
- Test benches can be quite complex and lengthy and may take longer to develop than the design that is tested.
- The results of a test are only as good as the test bench that is used to test a circuit. Care must be taken to write stimuli that will test a circuit thoroughly, exercising all of the operating features that are specified.
- Test benches use the **initial** statement **to provide a stimulus to the circuit** being tested. The initial statement **executes only once**, starting from simulation **time 0**, and may continue with any operations that are delayed by a given number of time units, as specified by the symbol **#.** For **example**, consider the initial block.

```
initial
  begin
        A = 0; B = 0;
    #10 A = 1;
    #20 A = 0; B = 1;
  end
```

The block is enclosed between the keywords **begin and end** . At time 0, A and B are set to 0. Ten time units later, A is changed to 1. Twenty time units after that (at t =30), A is changed to 0 and B to 1.

Inputs specified by a **three-bit truth table** can be generated with the **initial** block:

```
initial
begin
  D = 3'b000;
  repeat (7)
  #10 D = D + 3'b001;
end
```

When the simulator runs, the three-bit vector *D* is initialized to 000 at time=0. The keyword **repeat** specifies a looping statement: *D* is incremented by 1 seven times, once every 10 time units. The result is a sequence of binary numbers from 000 to 111.

A stimulus module has the following form:

module test_module_name;

// Declare local reg and wire identifiers.

// Instantiate the design module under test.

// Specify a stopwatch, using $finish to terminate the simulation.

// Generate stimulus, using initial and always statements.

// Display the output response (text or graphics (or both)).

endmodule

A test module is written like any other module, but it typically has no inputs or outputs. The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local **reg** data type. The outputs of the design module that are displayed for testing are declared in the stimulus module as local **wire** data type**.**

The module under test is then instantiated, using the **local identifiers** in its port list. **Figure 4.34** clarifies this relationship. The stimulus module generates inputs for the design module by declaring local identifiers **t_A and t_B as reg type** and checks the output of the design unit with the **wire identifier t_C** . The local identifiers are then used to instantiate the design module being tested. The simulator associates the (actual) local identifiers within the test bench, t_A, t_B , and t_C, with the formal identifiers of the module (A, B, C).

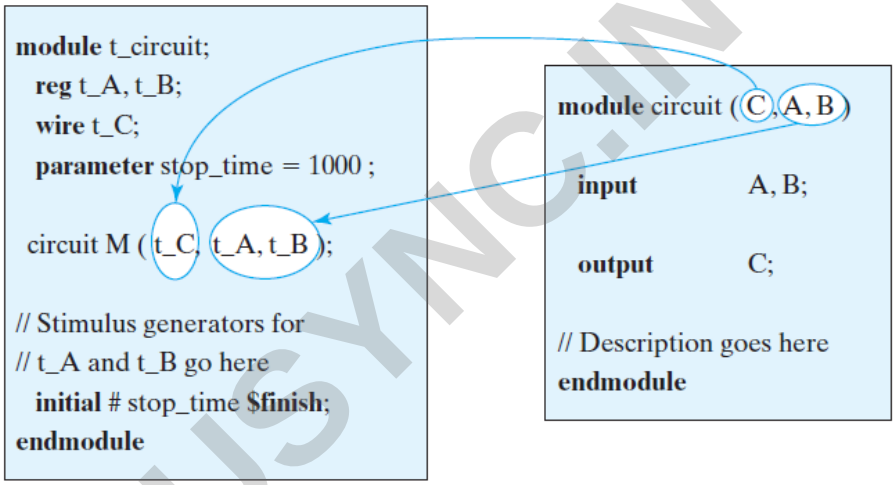


FIGURE 4.34
Interaction between stimulus and design modules

Some of the system tasks that are useful for display are:

$display —display a one-time value of variables or strings with an end-of-line return,

$write —same as **$display ,** but without going to next line,

$monitor —display variables whenever a value changes during a simulation run,

$time —display the simulation time,

$finish —terminate the simulation.

HDL Example 4.9 (Test Bench)

```
// Test bench with stimulus for mux_2x1_df

module t_mux_2x1_df;
  wire          t_mux_out;
  reg           t_A, t_B;
  reg           t_select;
  parameter stop_time = 50;

mux_2x1_df M1 (t_mux_out, t_A, t_B, t_select);        // Instantiation of circuit to be tested

initial # stop_time $finish;

  initial begin                                       // Stimulus generator
          t_select = 1; t_A = 0; t_B = 1;
    #10 t_A = 1; t_B = 0;
    #10 t_select = 0;
    #10 t_A = 0; t_B = 1;
  end

  initial begin                                       // Response monitor
   // $display (" time  Select  A  B  m_out");
   // $monitor ($time,, " %b   %b   %b   %b ", t_select, t_A, t_B, t_m_out);

     $monitor ("time =", $time,, "select = %b A = %b B = %b OUT = %b",
     t_select, t_A, t_B, t_mux_out);
    end
  endmodule

  // Dataflow description of two-to-one-line multiplexer

  // from Example 4.6
  module mux_2x1_df (m_out, A, B, select);
    output        m_out;
    input         A, B;
    input         select;

    assign m_out = (select)? A : B;
  endmodule

  Simulation log:
  select = 1 A = 0 B = 1 OUT = 0 time = 0
  select = 1 A = 1 B = 0 OUT = 1 time = 10
  select = 0 A = 1 B = 0 OUT = 0 time = 20
  select = 0 A = 0 B = 1 OUT = 1 time = 30
```