- **Module-2**

Combinational Logic: Introduction, Combinational Circuits, Design Procedure, Binary Adder-Subtractor, Decoders, Encoders, Multiplexers. HDL Models of Combinational Circuits – Adder, Multiplexer, Encoder. **Sequential Logic:** Introduction, Sequential Circuits, Storage Elements: Latches, Flip-Flops.

Text book 1: 4.1, 4.2, 4.4, 4.5, 4.9, 4.10, 4.11, 4.12, 5.1, 5.2, 5.3, 5.4.

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.Yykyy

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

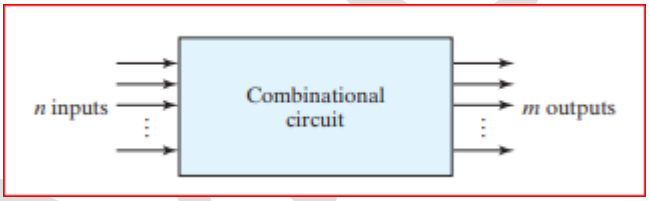


Figure 2.1: Block diagram of combinational circuit

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. 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 considered to be a sequential circuit. For n input variables, there are 2n 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.

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.

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

ANALYSIS PROCEDURE

The analysis of a combinational circuit requires that we determine the function that the circuit implements. This task starts with a given logic diagram and culminates with a set of Boolean functions, a truth table, or, possibly, an explanation of the circuit operation.

If the logic diagram to be analyzed is accompanied by a function name or an explanation of what it is assumed to accomplish, then the analysis problem reduces to a verification of the stated function. The analysis can be performed manually by finding the Boolean functions or truth table or by using a computer simulation program. The first step in the analysis is to make sure that the given circuit is combinational and not sequential. The diagram of a combinational circuit has logic gates with no feedback paths or memory elements. A feedback path is a connection from the output of one gate to the input of a second gate whose output forms part of the input to the first gate. Feedback paths in a digital circuit define a sequential circuit and must be analyzed by special methods and will not be considered here.

Once the logic diagram is verified to be that of a combinational circuit, one can proceed to obtain the output Boolean functions or the truth table. If the function of the circuit is under investigation, then it is necessary to interpret the operation of the circuit from the derived Boolean functions or truth table. The success of such an investigation is enhanced if one has previous experience and familiarity with a wide variety of digital circuits.

To obtain the output Boolean functions from a logic diagram, we proceed as follows:

1. Label all gate outputs that are a function of input variables with arbitrary symbols— but with meaningful names. Determine the Boolean functions for each gate output.
2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the Boolean functions for these gates.
3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

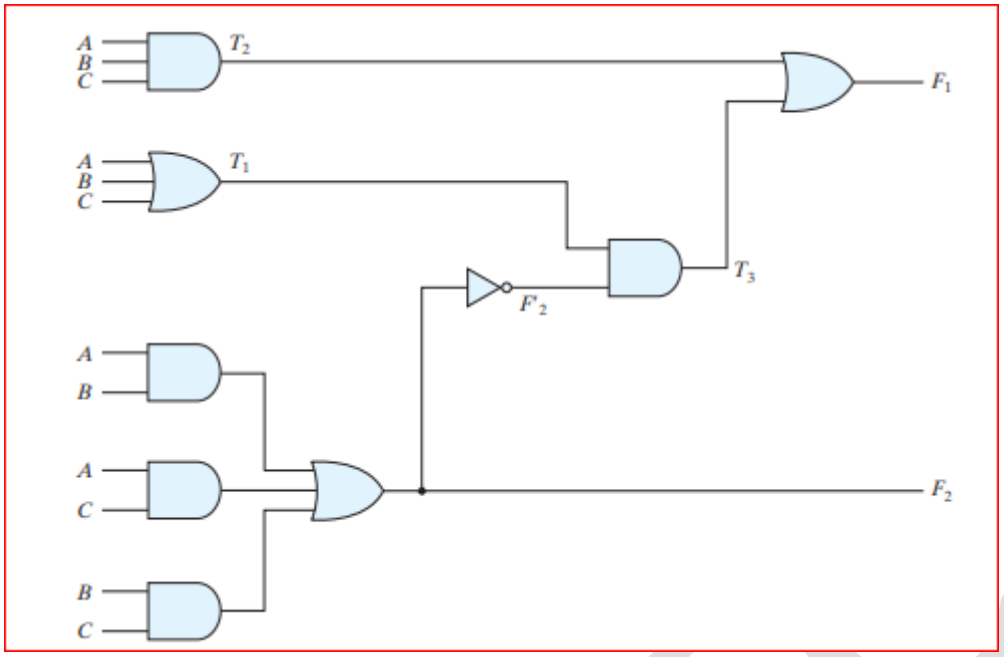


Figure 2.2: Logic diagram for analysis example

The analysis of the combinational circuit of Fig. 2.2

We note that the circuit has three binary inputs— A, B, and C —and two binary outputs— F_1 and F_2. The outputs of various gates are labelled with intermediate symbols. The outputs of gates that are a function only of input variables are T_1 and T_2. Output F_2 can easily be derived from the input variables. The Boolean functions for these three outputs are

$$F_2 = AB + AC + BC$$
$$T_1 = A + B + C$$
$$T_2 = ABC$$

Next, we consider outputs of gates that are a function of already defined symbols:

$$T_3 = F'_2 T_1$$
$$F_1 = T_3 + T_2$$

To obtain F1 as a function of A , B , and C , we form a series of substitutions as follows:

$$\begin{aligned} F_1 &= T_3 + T_2 = F'_2 T_1 + ABC = (AB + AC + BC)'(A + B + C) + ABC \\ &= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\ &= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\ &= A'BC' + A'B'C + AB'C' + ABC \end{aligned}$$

If we want to pursue the investigation and determine the information transformation task achieved by this circuit, we can draw the circuit from the derived Boolean expressions and try to recognize a familiar operation.

The derivation of the truth table for a circuit is a straightforward process once the output Boolean functions are known

Consistency

Read consistency

Update consistency

Relax consistency

R

. To obtain the truth table directly from the logic diagram without going through the derivations of the Boolean functions, we proceed as follows:

1. Determine the number of input variables in the circuit. For n inputs, form the 2^n possible input combinations and list the binary numbers from 0 to $(2^n - 1)$ in a table.
2. Label the outputs of selected gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates which are a function of the input variables only.
4. Proceed to obtain the truth table for the outputs of those gates which are a function of previously defined values until the columns for all outputs are determined.

Table 2.1:Truth Table for the Logic Diagram of Fig. 2.2

A	B	C	F_2	F_2'	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

The truth table for F_2 is determined directly from the values of A , B , and C , with F_2 equal to 1 for any combination that has two or three inputs equal to 1. The truth table for F'_2 is the complement of F_2. The truth tables for T_1 and T_2 are the OR and AND functions of the input variables, respectively. The values for T3 are derived from T1 and F'_2:T_3 is equal to 1 when both T_1 and F'_2 are equal to 1, and T_3 is equal to 0 otherwise. Finally, F_1 is equal to 1 for those combinations in which either T_2 or T_3 or both are equal to 1

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 truth tables and the design process of combinational circuits:

Truth Table Basics:

- A truth table for a combinational circuit contains input columns and output columns.
- Inputs are derived from the 2^n binary numbers for n input variables.
- Output values are determined based on specified conditions.

Interpreting Verbal Specifications:

- Verbal specifications are often incomplete and must be carefully interpreted to create an accurate truth table.
- Incorrect interpretations can lead to errors in the truth table.

Vector spaces are considered to be one of the best handheld devices .

Output Function Simplification:

- Output functions listed in the truth table are typically simplified.
- Simplification methods include algebraic manipulation, the map method, or computer-based simplification programs.
- Multiple simplified expressions may be available.

Choosing an Implementation:

- Practical design considerations play a crucial role in choosing an implementation.
- Constraints to consider include:
    - Number of gates
    - Number of inputs to a gate
    - Propagation time of signals through gates
    - Number of interconnections
    - Limitations of the driving capability of each gate
    - Other specific performance criteria dictated by the application.

Design Process:

- The design process often starts by satisfying elementary objectives, such as achieving simplified Boolean functions in a standard form.
- Subsequent steps focus on meeting additional performance criteria.
- The importance of each constraint varies based on the specific application.

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 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 2.2

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

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 0variables is shown in Table 2.2

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.

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.

A two-level logic diagram for each output may be obtained directly from the Boolean expressions derived from the maps. There are various other possibilities for a logic diagram that implements this circuit. The expressions obtained in Fig. 2.3 may be manipulated algebraically for the purpose of using common gates for two or more outputs

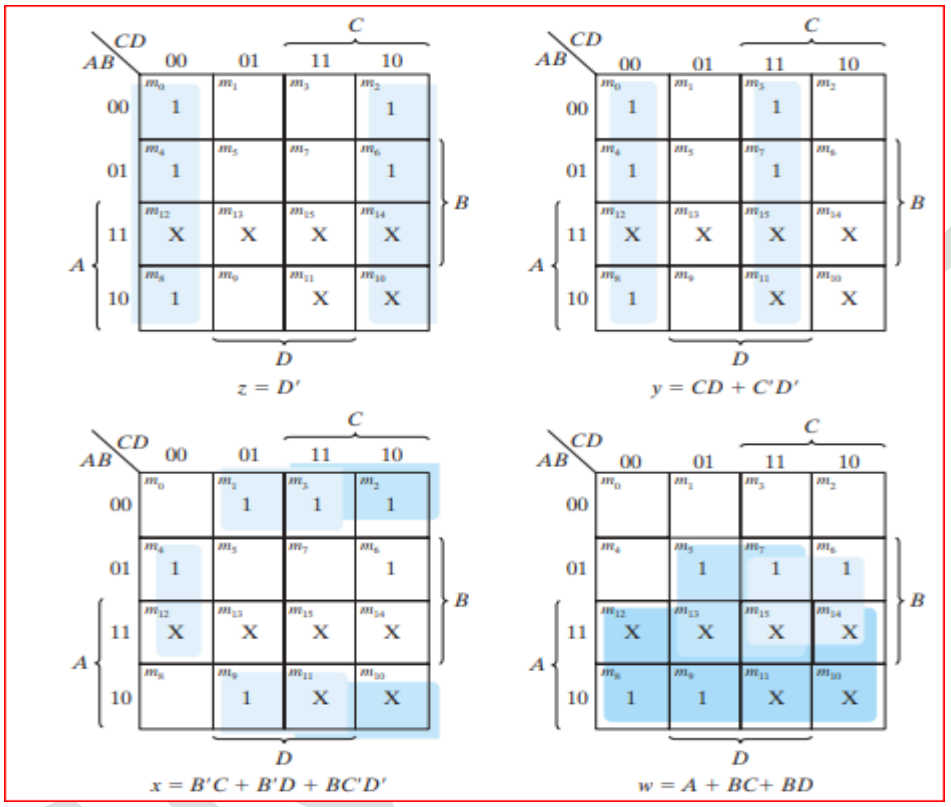


Figure 2.3: Maps for BCD-to-excess-3 code converter

implemented with three or more levels of gates:

$$
\begin{aligned}
z &= D' \\
y &= CD + C'D' = CD + (C + D)' \\
x &= B'C + B'D + BC'D' = B'(C + D) + BC'D' \\
&= B'(C + D) + B(C + D)' \\
w &= A + BC + BD = A + B(C + D)
\end{aligned}
$$

The logic diagram that implements these expressions is shown in Fig. 2.3 . Note that the OR gate whose output is C + D has been used to implement partially each of three outputs.

The implementation of Fig. 2.4 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.

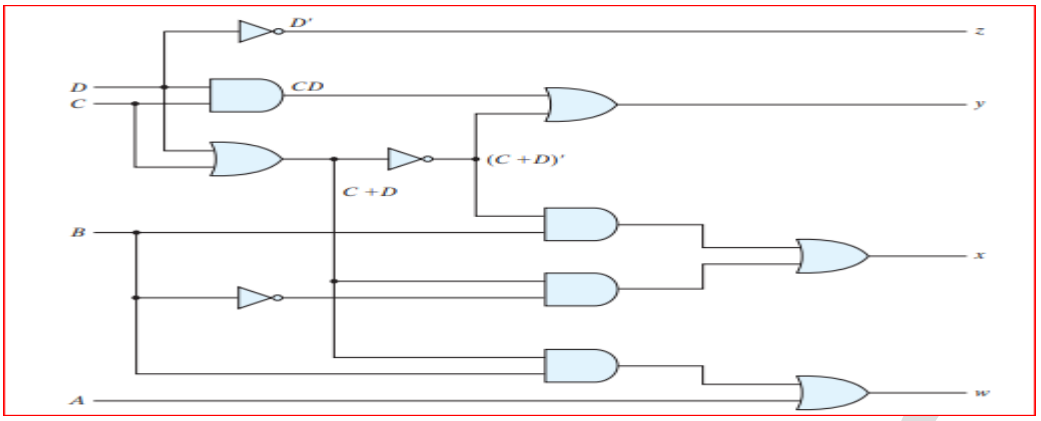


Figure 2.4: Logic diagram for BCD-to-excess-3 code converter

BINAY ADDER – SUBTRACTOR

Digital computers perform a variety of information-processing tasks. Among the functions encountered are the various arithmetic operations. 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.

When the augend and addend numbers contain more significant digits, 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.

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 2.3.

The C output is 1 only when both inputs are 1. The S output represents the least significant bit of the sum. 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. 2.5(a) . It can be also implemented with an exclusive-OR and an AND gate as shown in Fig. 2.5(b) . This form is used to show that two half adders can be used to construct a full adder.

Table 2.3

Half Adder

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

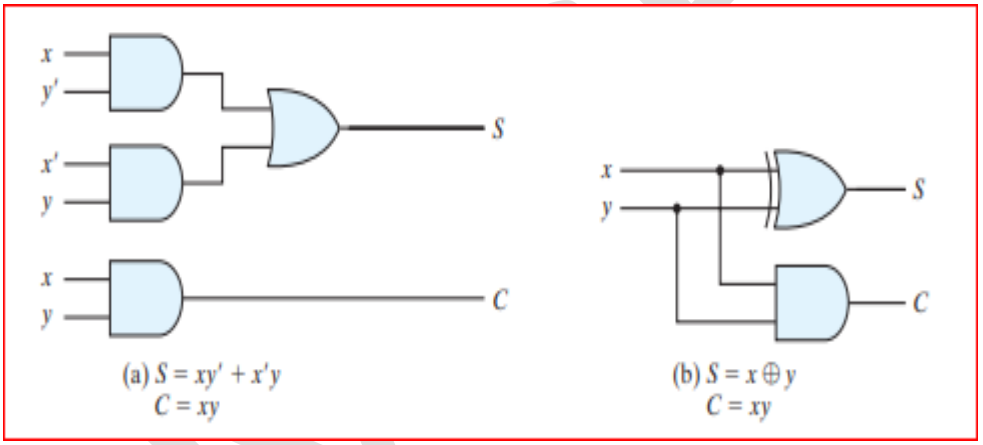


FIGURE 2.5: Implementation of half adder

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. Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3, and binary representation of 2 or 3 needs two bits. 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 2.4. The eight rows under the input variables designate all possible combinations of the three variables. The output variables are determined from the arithmetic sum of the input bits. 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 maps for the outputs of the full adder are shown in Fig. 2.6 . The simplified expressions are

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

The logic diagram for the full adder implemented in sum-of-products form is shown in Fig. 2.7 . It can also be implemented with two half adders and one OR gate, as shown in Fig. 2.8

Table 2.4 Full Adder

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

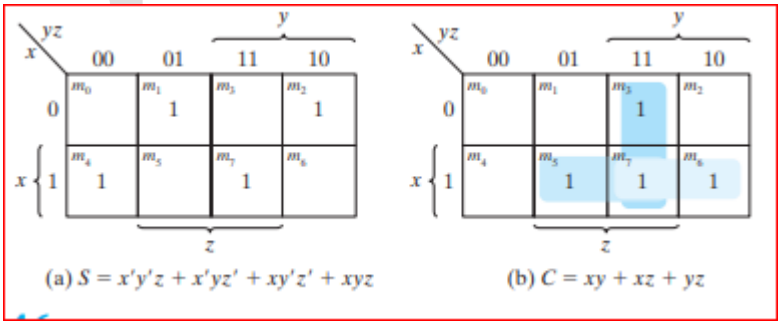


FIGURE 2.6 K-Maps for full adder

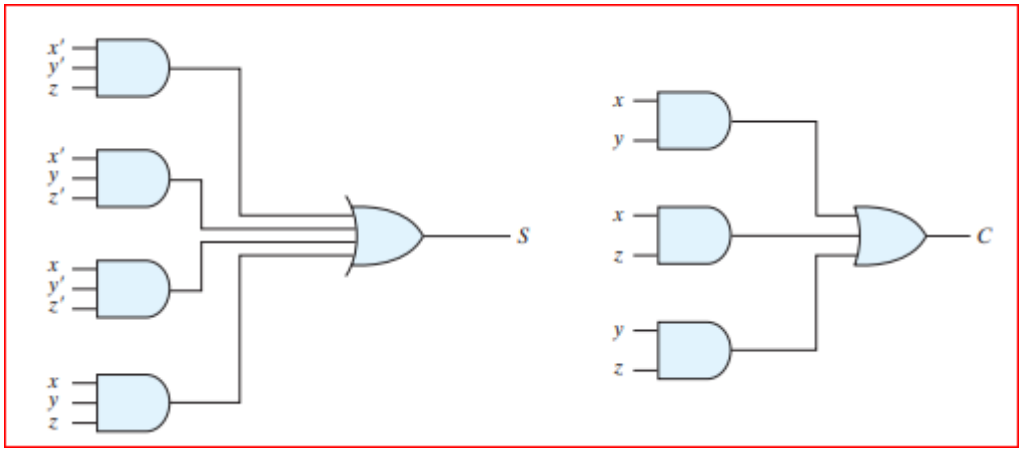


FIGURE 2.7 Implementation of full adder in sum-of-products form

The S output from the second half adder is the exclusive-OR of z and the output of the first half adder, giving

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

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.

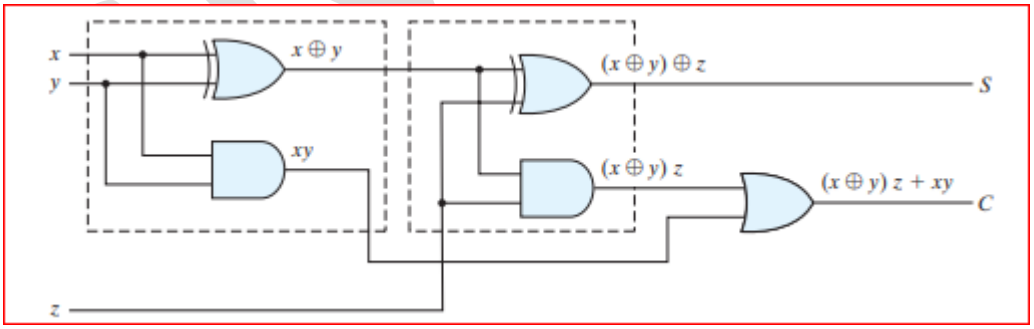


FIGURE 2.8 Implementation of full adder with two half adders and an OR gate

Addition of n-bit numbers requires a chain of n full adders or a chain of one-half adder and n-1 full adders. In the former case, the input carry to the least significant position is fixed at 0. Figure 2.9 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 C_0, and it ripples

through the full adders to the output carry C_4. 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. 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}

The bits are added with full adders, starting from the least significant position (subscript 0), to form the sum bit and carry bit. The input carry C_0 in the least significant position must be 0. The value of C_{i+1} in a given significant position is the output carry of the full adder. This value is transferred into the input carry of the full adder that adds the bits one higher significant position to the left. The sum bits are thus generated starting from the rightmost position and are available as soon as the corresponding previous carry bit is generated.

The four-bit adder is a typical example of a standard component. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit

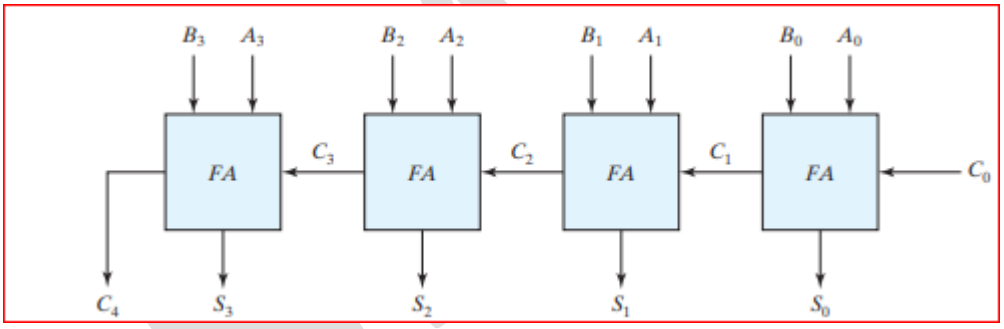


FIGURE 2.9 Four-bit adder

by the classical method would require a truth table with $2^9 = 512$ entries, since there are nine inputs to the circuit.

Carry Propagation

The addition of two binary numbers in parallel implies that all the bits of the augend and addend are available for computation at the same time. 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 S_i 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. 2.9. A_3 and B_3 are available as soon as input signals are applied to the adder. However, input carry C3 does not settle to its final value until C_2 is available from the previous stage. Similarly, C_2 has to wait for C_1 and so on down to C_0. Thus, only after the carry propagates and ripples through all stages will the last output S_3 and carry C_4 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. 2.10

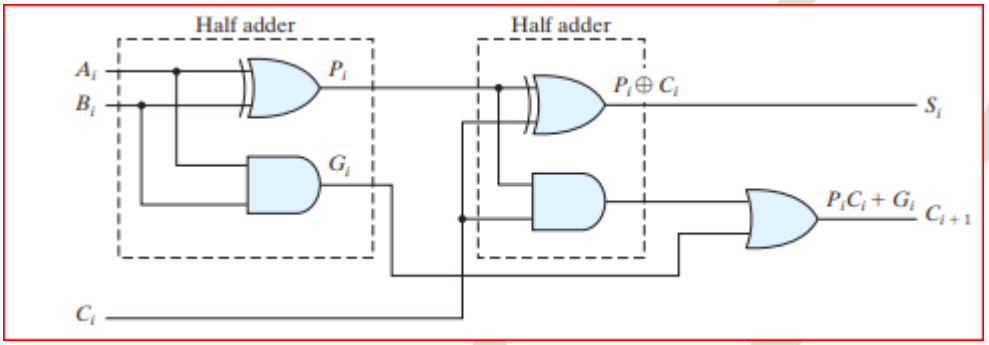


FIGURE 2.10 Full adder with P and G shown

The carry propagation time is an important attribute of the adder because it limits the speed with which two numbers are added. Although the adder—or, for that matter, any combinational circuit—will always have some value at its output terminals, the outputs will not be correct unless the signals are given enough time to propagate through the gates connected from the inputs to the outputs. Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is critical.

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. 2.10 . If we define two new binary variables

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

the output sum and carry can respectively be expressed as

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

Gi is called a carry generate , and it produces a carry of 1 when both A_i and B_i are 1, regardless of the input carry C_i . P_i is called a carry propagate , because it determines whether a carry into stage i will propagate into stage i + 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 C_1, C_2, and C_3 are implemented in the carry lookahead generator shown in Fig. 2.11

Note that this circuit can add in less time because C_3 does not have to wait for C_2 and C_1 to propagate; in fact, C_3 is propagated at the same time as C_1 and C_2.

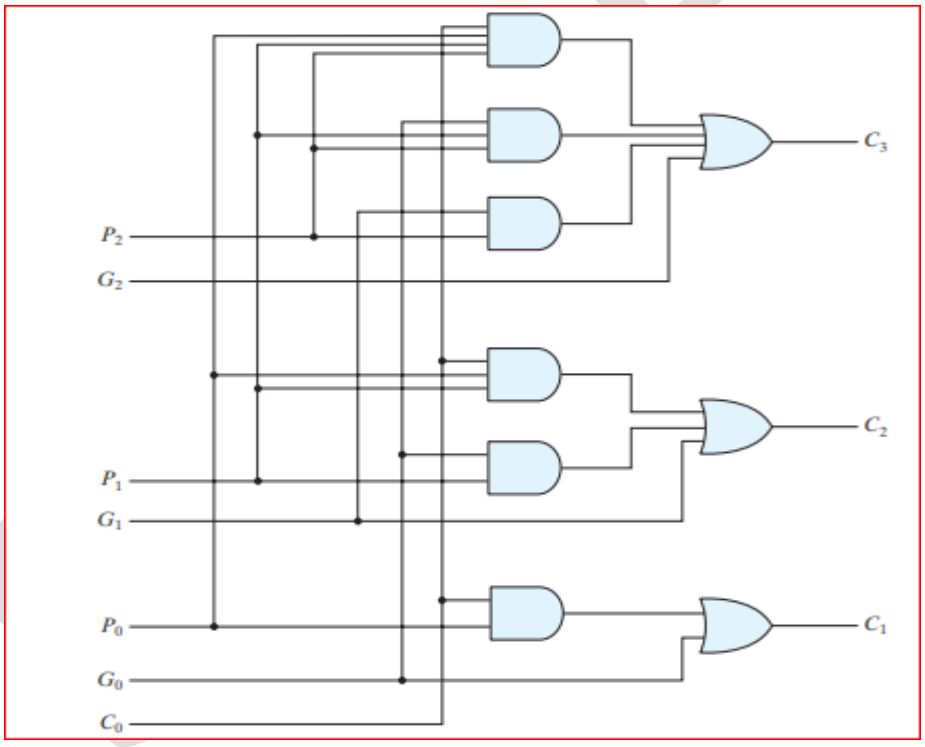


FIGURE 2.11 Logic diagram of carry lookahead generator

The construction of a four-bit adder with a carry lookahead scheme is shown in Fig. 2.12 . 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

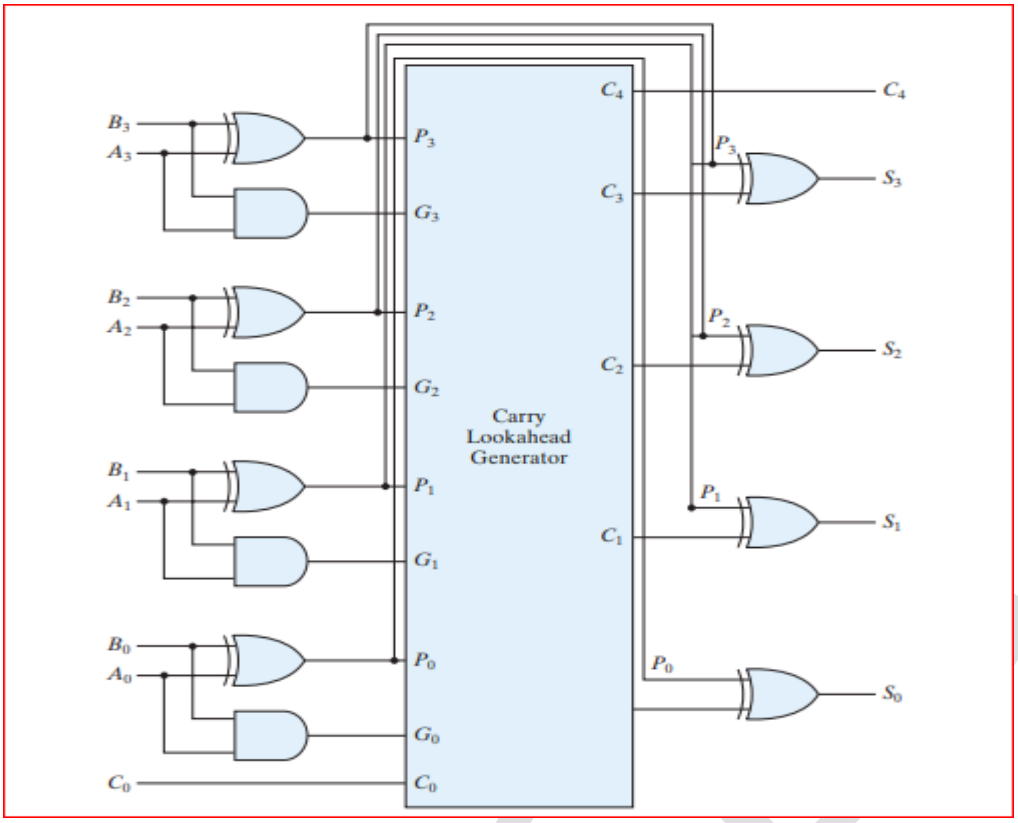


FIGURE 2.12 Four-bit adder with carry lookahead

a delay through two levels of gates. Thus, outputs S_1 through S_3 have equal propagation delay times. The two-level circuit for the output carry C_4 is not shown. This circuit can easily be derived by the equation-substitution method.

Binary Subtractor

The subtraction of unsigned binary numbers can be done most conveniently by means of complements 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.

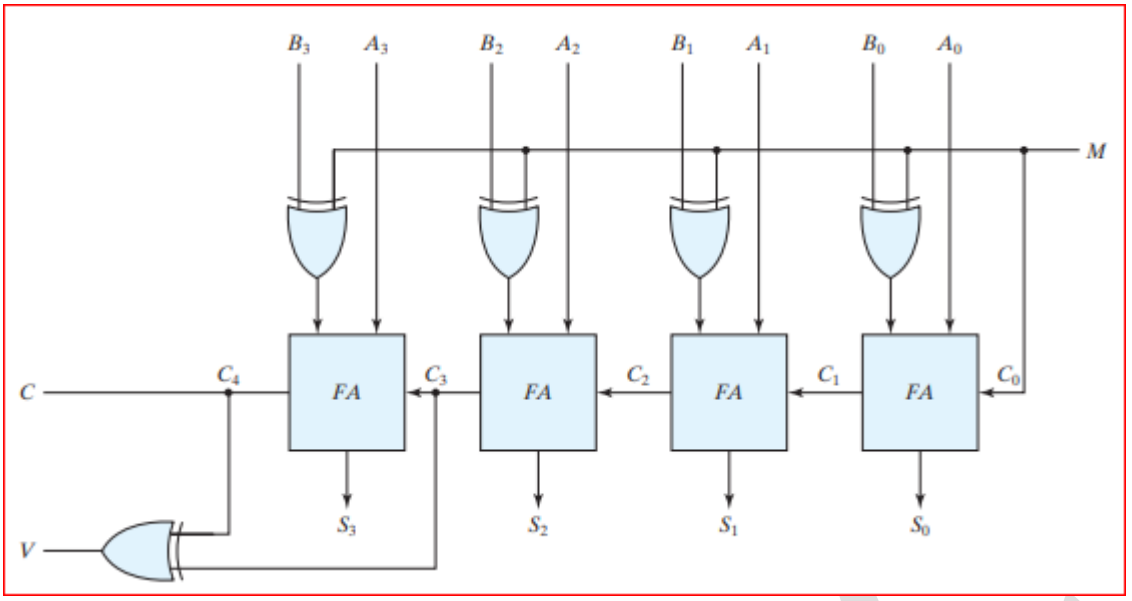


FIGURE 2.13 Four-bit adder–subtractor (with overflow detection)

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 1(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. 2.13. 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⊕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⊕ 1= B' and C_0 = 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.

Overflow

- Overflow occurs when two numbers with 'n' digits each are added, and the sum results in a number with 'n + 1' digits.
- This concept applies to both binary and decimal numbers, whether they are signed or unsigned.
- In manual calculations (paper and pencil), overflow is not a problem because there are no space limitations to write down the result.

- In digital computers, overflow is a concern because the number of bits used to represent a number is finite, and a result with 'n + 1' bits cannot fit into an 'n'-bit word.
- To handle overflow, many computers detect its occurrence and set a corresponding flip-flop that can be checked by the user.

Overflow Detection for Unsigned Binary Numbers

- When adding two unsigned binary numbers, overflow is detected by examining the end carry-out from the most significant position (leftmost bit).
- If there is a carry-out from the leftmost bit, it indicates an overflow.

Overflow Detection for Signed Binary Numbers

- In the case of signed binary numbers:
  - The leftmost bit (most significant bit) represents the sign, where '0' typically denotes positive and '1' denotes negative.
  - Negative numbers are typically represented in 2's complement form.
- When adding two signed binary numbers:
  - The sign bit is treated as part of the number and not as a sign indicator during the addition.
  - The end carry does not indicate an overflow because it is part of the signed number's representation.

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.

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. This is also true for -70 and -80. The two additions in binary are shown next, together with the last two carries:

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 carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow is detected when the output of the gate is equal to 1.

For this method to work correctly, the 2's complement of a negative number must be computed by taking the 1's complement and adding 1. This takes care of the condition when the maximum negative number is complemented.

The binary adder–subtractor circuit with outputs C and V is shown in Fig. 2.13. 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.

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 2n 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 … 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. The name decoder is also used in conjunction with other code converters, such as a BCD-to-seven-segment decoder.

Example, consider the three-to-eight-line decoder circuit of Fig. 2.14 . 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. Howeve r, 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.

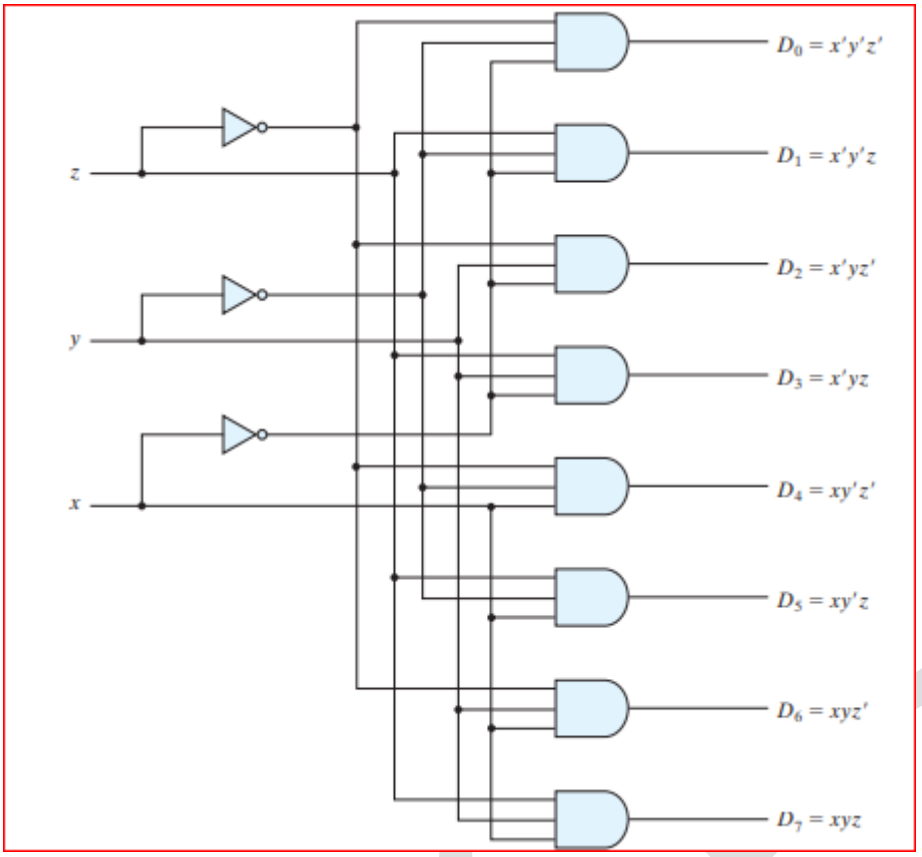


FIGURE 2.14 Three-to-eight-line decoder

Table 2.5 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

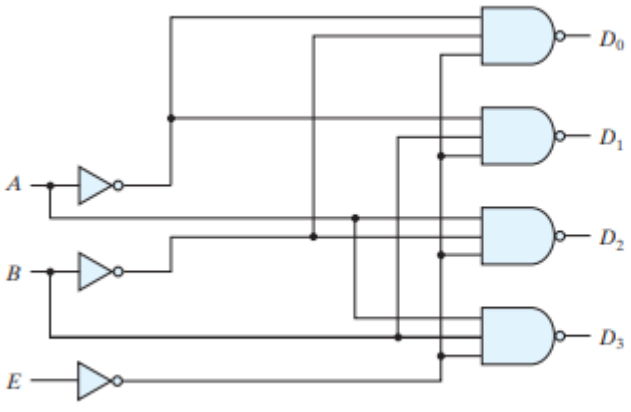
The operation of the decoder may be clarified by the truth table listed in Table 2.5 . For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1.

The binary number now existing in the input lines is represented by the output whose value is 1, which is the minterm equivalent. NAND gates are used in the construction of some decoders. It is more cost-effective to manufacture the decoder minimum terms in their complemented form since a NAND gate generates the AND operation with an inverted output.

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. 2.15 .

- 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. In general, a decoder may operate with complemented or uncomplemented outputs. The enable input may be activated with a 0 or with a 1 signal.
- Some decoders have two or more enable inputs that must satisfy a given logic condition in order to enable the circuit



(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 2.15 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. 2.15 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 D_2 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 2.16 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. In general, enable inputs are a convenient feature for interconnecting two or more standard components for the purpose of combining them into a similar function with more inputs and outputs.

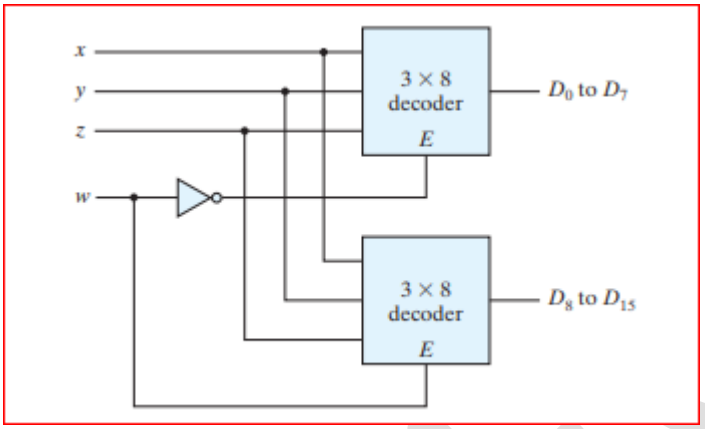


FIGURE 2.16 4x16 decoder constructed with two 3x8 decoders

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. A decoder is then chosen that generates all the minterms of the input variables. 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 2.4), 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. 2.17 . 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.

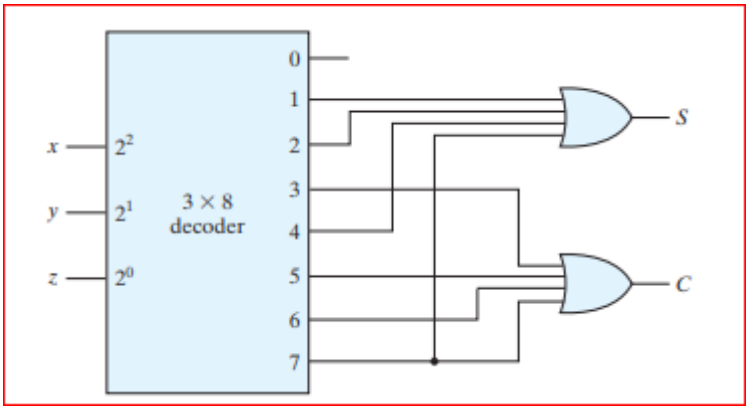


FIGURE 2.17 Implementation of a full adder with a decoder

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.

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. 2.17, 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.

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.

Example of an encoder is the octal-to-binary encoder whose truth table is given in Table 2.6 . 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:

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

The encoder can be implemented with three OR gates.

Table 2.6 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 defined in Table 2.6 has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination. For example, if D_3 and D_6 are 1 simultaneously, the output of the encoder will be 111 because all three outputs are equal to 1. The output 111 does not represent either binary 3 or binary 6. To resolve this ambiguity, encoder circuits must establish an input priority to ensure that only one input is encoded. If we establish a higher priority for inputs with higher subscript numbers, and if both D_3 and D_6 are 1 at the same time, the output will be 110 because D6 has higher priority than D3. Another ambiguity in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0; but this output is the same as when D_0 is equal to 1. The discrepancy can be resolved by providing one more output to indicate whether at least one input is equal to 1.

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 2.7 . 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 a truth table in condensed form. Instead of listing all 16 minterms of four variables, the truth table uses an X to represent either 1 or 0. For example, X 100 represents the two minterms 0100 and 1100.

According to Table 2.7 , 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). D_2 has the next priority level. The output is 10 if $D_2 = 1$, provided that $D_3 = 0$, regardless of the values of the other two lower priority inputs. The output for D_1 is generated only if higher priority inputs are 0, and so on down the priority levels.

Table 2.7 Truth Table of a Priority Encoder

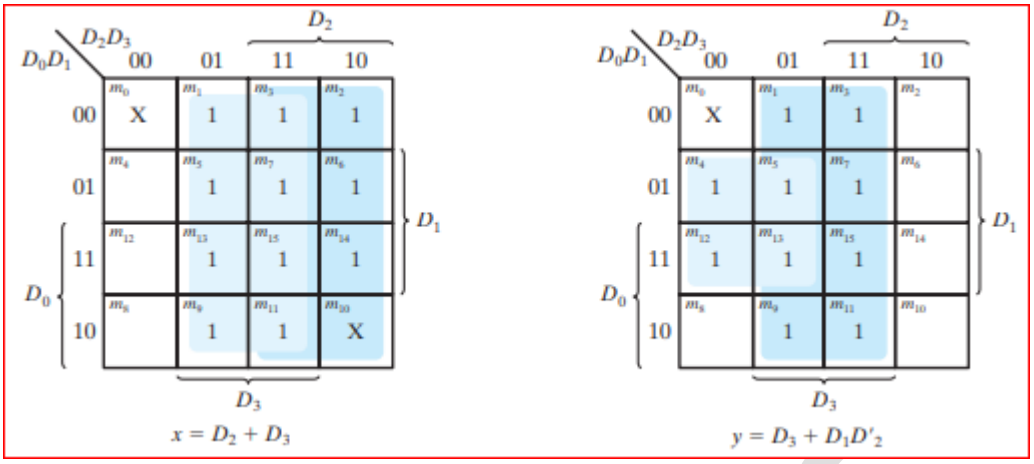


FIGURE 2.18 Maps for a priority encoder

The maps for simplifying outputs x and y are shown in Fig. 2.18 . 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. The priority encoder is implemented in Fig. 2.19 according to the following Boolean functions:

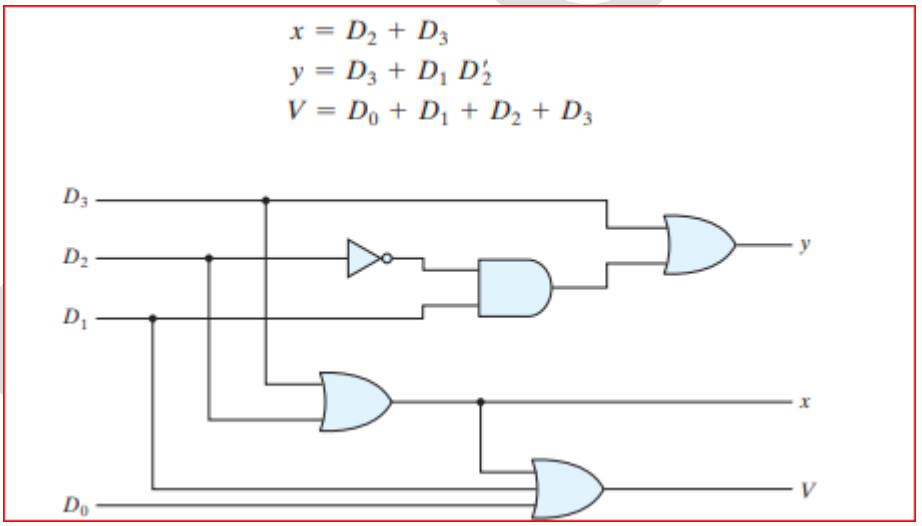


FIGURE 2.19 Four-input priority encoder

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. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.

A two-to-one-line multiplexer connects one of two 1-bit sources to a common destination, as shown in Fig. 2.20. 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 I1 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. 2.20(b). It suggests visually how a selected one of multiple data sources is directed into a single destination. The multiplexer is often labeled "MUX" in block diagrams.

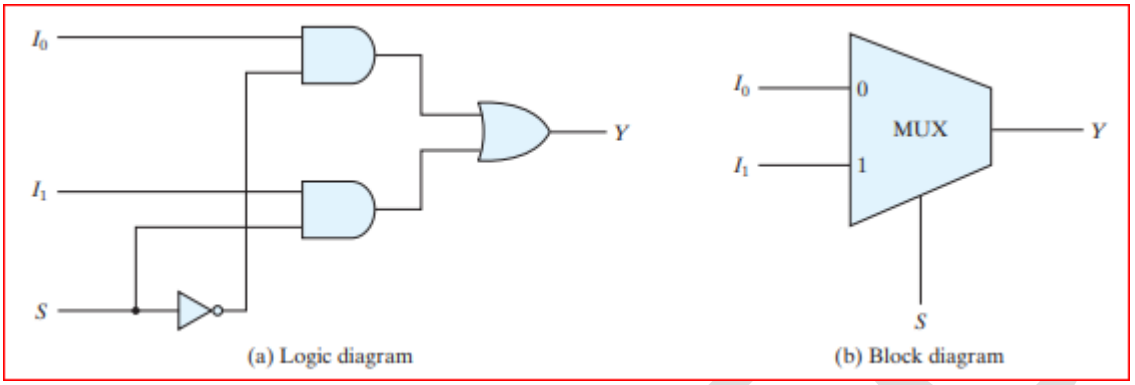


FIGURE 2.20 Two-to-one-line multiplexer

A four-to-one-line multiplexer is shown in Fig. 2.21. Each of the four inputs, I_0 through I3, is applied to one input of an AND gate. Selection lines S_1 and S_0 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.

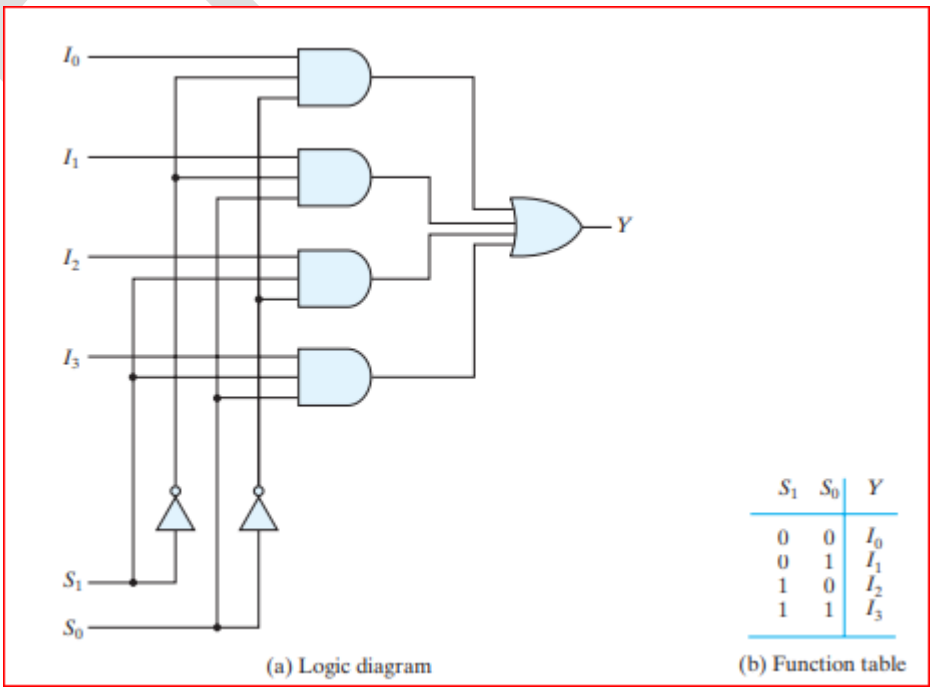


FIGURE 2.21 Four-to-one-line multiplexer

The AND gates and inverters in the multiplexer resemble a decoder circuit, and indeed, they decode the selection input lines. In general, a 2^n -to-1-line multiplexer is constructed from an n -to-2^n decoder by adding 2^n input lines to it, one to each AND gate. The outputs of the AND gates are applied to a single OR gate. The size of a multiplexer is specified by the number 2^n of its data input lines and the single output line. The n selection lines are implied from the 2^n data lines. As in decoders, multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as a normal 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. 2.22. 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.

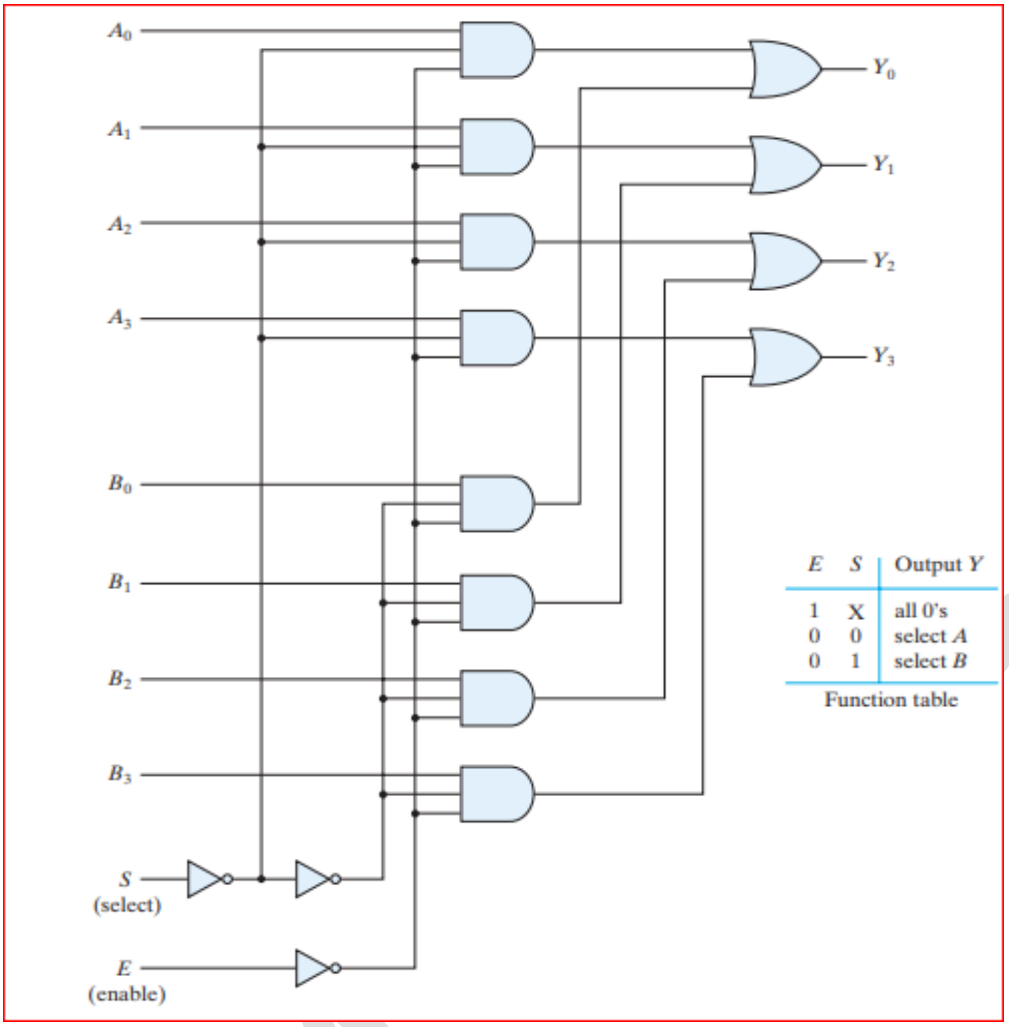


FIGURE 2.22 Quadruple two-to-one-line multiplexer

Boolean Function Implementation

An examination of the logic diagram of a multiplexer reveals that it is essentially a decoder that includes the OR gate within the unit. 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.

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) = ∑(1, 2, 6, 7)

This function of three variables can be implemented with a four-to-one-line multiplexer as shown in Fig. 2.23

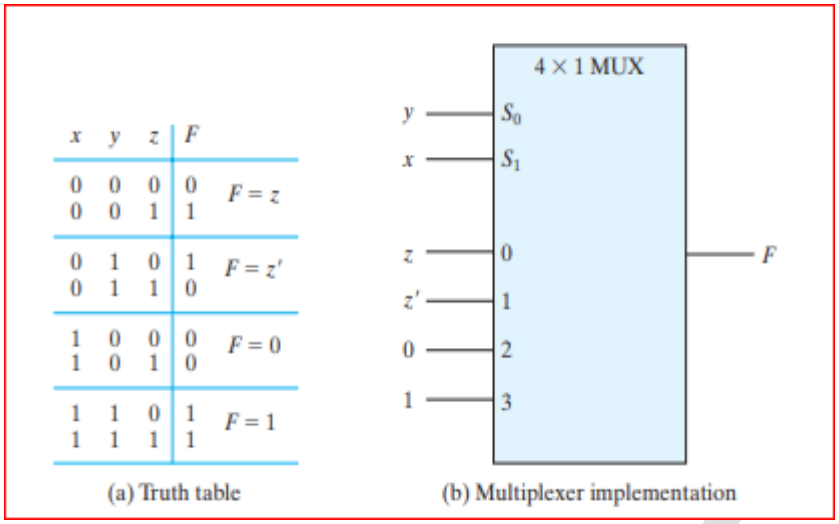


FIGURE 2.23 Implementing a Boolean function with a multiplexer

The two variables x and y are applied to the selection lines in that order; x is connected to the S_1 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. This requires that variable z be applied to data input 0. The operation of the multiplexer is such that when xy = 00, data input 0 has a path to the output, and that makes F equal to z . 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.

The general procedure for implementing any Boolean function of n variables with a multiplexer with n - 1 selection inputs and 2^n-1 data inputs follows from the previous example. To begin with, Boolean function is listed in a truth table. Then first n - 1 variables in the table are applied to the selection inputs of the multiplexer. For each combination of the selection variables, we evaluate the output as a function of the last variable. This function can be 0, 1, the variable, or the complement of the variable. These values are then applied to the data inputs in the proper order. As a second example, consider the implementation of the Boolean function

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

This function is implemented with a multiplexer with three selection inputs as shown in Fig. 2.24 . Note that the first variable A must be connected to selection input S2 so that A , B, and C correspond to selection inputs S_2, S_1, and S_0, respectively.

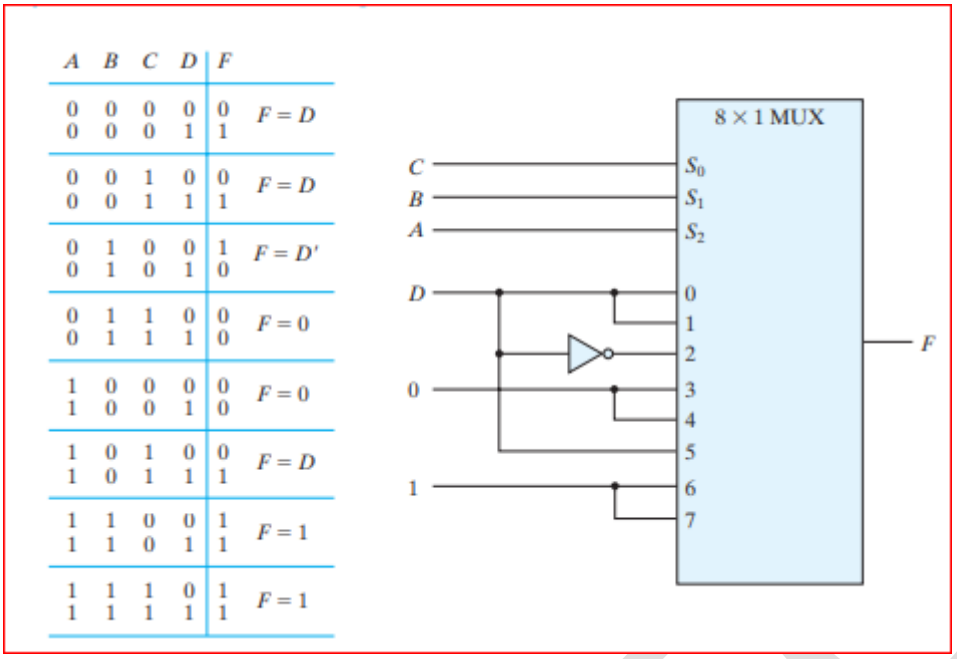


FIGURE 2.24 Implementing a four-input function with a multiplexer

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. When integrated circuits are used, logic 0 corresponds to signal ground and logic 1 is equivalent to the power signal, depending on the technology.

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. The graphic symbol for a three-state buffer gate is shown in Fig. 2.25 .

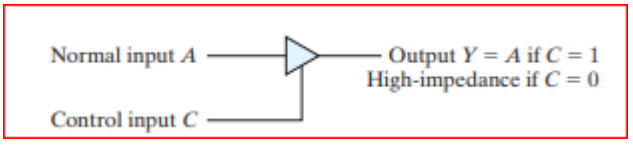


FIGURE 2.25 Graphic symbol for a three-state buffer

It is distinguished from a normal buffer by an input control line entering the bottom of the symbol. 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, the output is enabled and the gate behaves like a conventional buffer, with the 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 high-impedance state of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common line without endangering loading effects.

The construction of multiplexers with three-state buffers is demonstrated in Fig. 2.26 . Figure 2.26(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. (Note that this type of connection cannot be made with gates that do not have three-state outputs.) 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 . The construction of a four-to-one-line multiplexer is shown in Fig. 2.26 (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

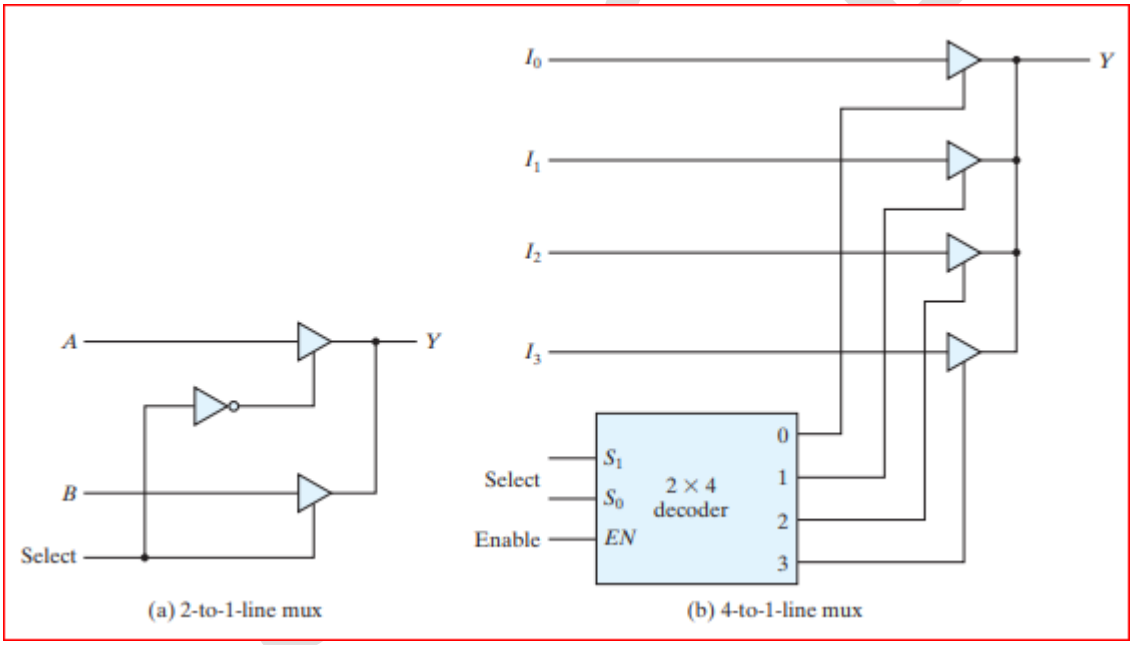


FIGURE 2.26 Multiplexers with three-state gates

I_3 will be connected to the output line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only 1 three state buffer has access to the output 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, depending on the binary value in the select

inputs of the decoder. Careful investigation reveals that this circuit is another way of constructing a four-to-one-line multiplexer.

HDL MODELS OF COMBINATIONAL CIRCUITS

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.

Behavioral modeling that is used to describe combinational and sequential circuits at a higher level of abstraction 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. 2.27. There is one other modeling style, called switch-level modeling. It is sometimes used in the simulation of MOS transistor circuit models, but not in logic synthesis.

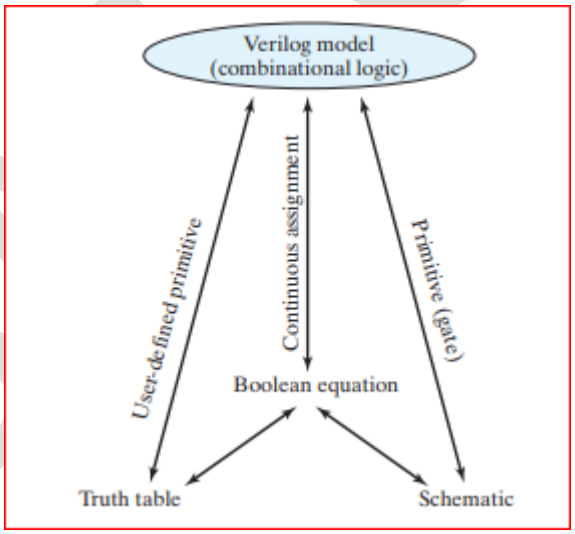


FIGURE 2.27 Relationship of Verilog constructs to truth tables, Boolean equations, and schematics

Gate-Level Modeling

In this type of representation, a circuit is specified by its logic gates and their interconnections. Gatelevel 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.

- Verilog, a hardware description language, includes a functional description for each type of gate.
- Verilog uses a four-valued logic system, including 0, 1, unknown (x), and high impedance (z).
- An unknown value (x) is assigned when the logic value of a signal is ambiguous, such as in the absence of a reset condition in a flip-flop.
- High-impedance (z) occurs at the output of three-state gates that are not enabled or when a wire is left unconnected.
- The four-valued logic truth tables for and, or, xor, and not primitives are provided, where x represents ambiguity and z represents high impedance.
- For the and gate, the output is 1 only when both inputs are 1, and it's 0 if any input is 0. If either input is x or z, the output is x.
- The or gate's output is 0 if both inputs are 0, 1 if any input is 1, and x otherwise.
- The truth tables for the other four gates are similar but with complemented outputs.
- Verilog, a hardware description language, includes a functional description for each type of gate.
- Verilog uses a four-valued logic system, including 0, 1, unknown (x), and high impedance (z).
- An unknown value (x) is assigned when the logic value of a signal is ambiguous, such as in the absence of a reset condition in a flip-flop.
- High-impedance (z) occurs at the output of three-state gates that are not enabled or when a wire is left unconnected.
- The four-valued logic truth tables for and, or, xor, and not primitives are provided, where x represents ambiguity and z represents high impedance. Shown in table 2.8.

Table 2.8 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

- For the and gate, the output is 1 only when both inputs are 1, and it's 0 if any input is 0. If either input is x or z, the output is x.
- The or gate's output is 0 if both inputs are 0, 1 if any input is 1, and x otherwise.
- The truth tables for the other four gates are similar but with complemented outputs.

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. The individual bits are specified within square brackets, so D[2] specifies bit 2 of D . It is also possible to address parts of vectors. For example, SUM[2: 0] specifies the three least significant bits of vector SUM .

HDL Example 2.1 shows the gate-level description of a two-to-four-line decoder. (See Fig. 2.15 .) 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 . Remember that the output is always listed first in the port list of a primitive , followed by the inputs.

Note that the keywords **not** and **nand** are written only once and do not have to be repeated for each gate, but commas must be inserted at the end of each of the gates in the series, except for the last statement, which must be terminated with a semicolon.

HDL Example 2.1 (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.

A bottom-up hierarchical description of a four-bit adder is shown in HDL Example 2.2. The half adder is defined by instantiating primitive gates. The next module describes the full adder by instantiating and connecting two half adders. The third module describes the four-bit adder by instantiating and connecting four full adders. Note that the first character of an identifier cannot be a number, but can be an underscore, so the module name _4bitadder is valid. An alternative name that is meaningful, but does not require a leading underscore, is adder_4_bit . The instantiation is done by using the name of the module that is instantiated together with a new (or the same) set of port names. For example, the half adder HA1 inside the full adder module is instantiated with ports S1 , C1 , x, and y . This produces a half adder with outputs S1 and C1 and inputs x and y.

HDL Example 2.2 (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
```

Note that modules can be instantiated (nested) within other modules, but module declarations cannot be nested; that is, a module definition (declaration) cannot be placed within another module declaration.

Three-State Gates

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. 2.28 . 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

gate name 1output,input, control2;

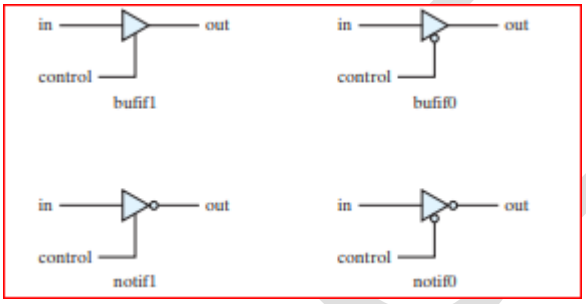


FIGURE 2.28 Three-state gates

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. 2.29 .

The HDL description must use a tri data type for the output:

```
// 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. In simulation, their value is determined by a continuous assignment statement or by the device whose output they represent. 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 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.

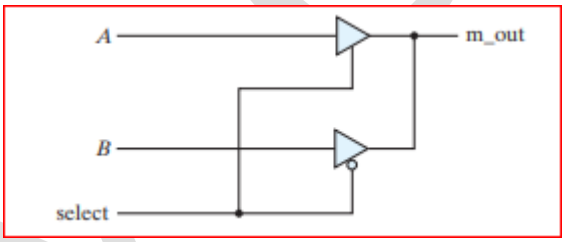


FIGURE 2.29 Two-to-one-line multiplexer with three-state buffers

Dataflow Modeling

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 2.9 lists some of these operators, their symbols, and the operation that they perform.

It is necessary to distinguish between arithmetic and logic operations, so different symbols are used for each. The plus symbol (+) indicates the arithmetic operation of addition; the bitwise logic AND operation (conjunction) uses the symbol &. There are special symbols for bitwise logical OR (disjunction), NOT, and XOR. The equality symbol uses two equals signs (without spaces between them) to distinguish it from the equals sign used with the assign statement. The bitwise operators operate bit by bit on a pair of vector operands to produce a vector result. The concatenation operator provides a mechanism for appending multiple operands. For example, two operands with two bits each can be concatenated to form an operand with four bits.

Table 2.9 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. A net is declared explicitly by a net keyword (e.g., wire) or by declaring an identifier to be an input port. The logic value associated with a net is determined by what the net is connected to. If the net is connected to an output of a gate, the net is said to be driven by the gate, and the logic value of the net is determined by the logic values of the inputs to the gate and the truth table of the gate. If the identifier of a net is the left-hand side of a continuous assignment statement or a procedural assignment statement, the value assigned to the net is specified by a Boolean expression that uses operands and operators. As an example, assuming that 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

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

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.

The next two examples show the dataflow models of the two previous gate-level examples. The dataflow description of a two-to-four-line decoder with active-low output enable and inverted output is shown in HDL Example 2.3. The circuit is defined with four continuous assignment statements using Boolean expressions, one for each output.

HDL Example 2.3 (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 the four-bit adder is shown in HDL Example 2.4. 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 C _ in . The target output is the concatenation of the output carry C _ out 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.

HDL Example 2.4 (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
```

Dataflow HDL models describe combinational circuits by their function rather than by their gate structure.

To show how dataflow descriptions facilitate digital design, consider the 4-bit magnitude comparator described in HDL Example 2.5. 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. Note that equality (identity) is symbolized with two equals signs (= =) to distinguish the operation from that of the assignment operator (=).

HDL Example 2.5 (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
```

The next example uses the conditional operator (?:). This operator takes three operands:

condition? true-expression : false-expression;

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 2.6 describes a two-to-one-line multiplexer using the conditional operator. The continuous assignment.

```
assign OUT = select ? A : B;
```

specifies the condition that OUT = A if select = 1, else OUT =B if select =0.

HDL Example 2.6 (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 2.7 shows the behavioral description of a two-to-one-line multiplexer. 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. In this case, these variables are 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. The conditional statement

if–else provides a decision based upon the value of the select input. The if statement can be written without the equality symbol:

```
if (select) OUT = A;
```

The statement implies that select is checked for logic 1.

HDL Example 2.7 (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 2.8 describes the function of a four-to-one-line multiplexer. The select input is defined as a two-bit vector, and output y 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. Whenever in_0, in_1, in_2, in_3 or select change, the case expression (select) is evaluated and its value compared, from top to bottom, with the values in the list of statements that follow, the so-called case items. The statement associated with the first case item that matches the case expression is executed. In the absence of a match, no statement is executed. Since select is a two-bit number, it can be equal to 00, 01, 10, or 11. The case items have an implied priority because the list is evaluated from top to bottom.

HDL Example 2.8 (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.

The case construct has two important variations**: casex** and **casez .** The first will treat as don't-cares any bits of the case expression or the **case** item that have logic value x or z . The **casez** construct treats as

don't-cares only the logic value z , for the purpose of detecting a match between the **case** expression and a **case** item.

The list of case items need not be complete. If the list of case items does not include all possible bit patterns of the case expression, no match can be detected. Unlisted case items, i.e., bit patterns that are not explicitly decoded can be treated by using the default keyword as the last item in the list of case items.

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.

In addition to employing the always statement, test benches use the initial statement to provide a stimulus to the circuit being tested. We use the term "always statement" loosely. Actually, always is a Verilog language construct specifying how the associated statement is to execute #.

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
```

- Test Module Structure: A test module in Verilog is written like any other module but usually doesn't have any inputs or outputs of its own.

- Stimulus Module: In Verilog, signals that are applied as inputs to the design module for simulation are declared in the stimulus module using local reg data types.
- Displaying Outputs: The outputs of the design module that need to be observed and tested are declared in the stimulus module as local wire data types.
- Instantiating the Design Module: To perform testing, the module under test (the design module) is instantiated within the stimulus module using the local identifiers defined for inputs and outputs.
- Identifier Association: The Verilog simulator associates the actual local identifiers in the test bench (e.g., t_A, t_B, t_C) with the formal identifiers of the design module's ports (e.g., A, B, C) based on their positions in the port list.
- Flexible Port Connection: Verilog provides a more flexible name association mechanism for connecting ports in larger circuits, allowing you to explicitly connect ports by name rather than relying on position.
- Simulation Results: The response to the stimulus generated by initial and always blocks in the test module appears as text output in standard output. Simulators with graphical capabilities can also display the results as waveforms (timing diagrams).
- Numerical Outputs: Verilog provides system tasks for displaying numerical outputs. These tasks are recognized by keywords that start with the symbol $.

$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.

The syntax for **$display, $write, and $monitor** is of the form

Task-name (format specification, argumentlist);

The format specification uses the symbol % to specify the radix of the numbers that are displayed and may have a string enclosed in quotes (&). The base may be binary, decimal, hexadecimal, or octal, identified with the symbols %b, %d, %h, and %o, respectively (%B, %D, %H, and %O are valid too).
For example, the statement

$display ("%d %b %b", C, A, B);

specifies the display of C in decimal and of A and B in binary. Note that there are no commas in the format specification, that the format specification and argument list are separated by a comma, and that the argument list has commas between the variables. An example that specifies a string enclosed in quotes may look like the statement

$display ("time= %0d A=%b", $time, A, B);

and will produce the display

time=3 A =10 B=1

where (time=), (A=), and (B =) are part of the string to be displayed. The format specifiers %0d, %b, and %b specify the base for **$time** , A , and B , respectively. In displaying time values, it is better to use the format %0d instead of %d. This provides a display of the significant digits without the leading spaces that %d will include. (%d will display about 10 leading spaces because time is calculated as a 32-bit number.)

An example of a stimulus module is shown in HDL Example 2.9. The circuit to be tested is the two-to-one-line multiplexer described in Example 2.6. The module t_mux_2x1_df has no ports. The inputs for the mux are declared with a reg keyword and the outputs with a wire keyword. The mux is instantiated with the local variables. The initial block specifies a sequence of binary values to be applied during the simulation. The output response is checked with the $monitor system task. Every time a variable in its argument changes value, the simulator displays the inputs, output, and time. The result of the simulation is listed under the simulation log in the example. It shows that m_out =A when select = 1 and m_out = B when select ! 0 verifying the operation of the multiplexer.

HDL Example 2.9 (Test Bench)

```
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
```

Logic simulation is a fast and accurate method of verifying that a model of a combinational circuit is correct. There are two types of verification: functional and timing. This can be done by observing the waveforms at the outputs of the gates when they respond to a given input. An example HDL Example 2.3 A $monitor system task displays the output caused by the given stimulus. A commented alternative statement having a $display task would create a header that could be used with a $monitor statement to eliminate the repetition of names on each line of output.

The gate-level description of this circuit is shown in HDL Example 4.10. The circuit has three inputs, two outputs, and nine gates. The description of the circuit follows the interconnections between the gates according to the schematic diagram of Fig. 2.2 . The stimulus for the circuit is listed in the second module. The inputs for simulating the circuit are specified with a three-bit reg vector D . D[2] is equivalent to input A , D[1] to input B , and D[0] to input C . The outputs of the circuit F1 and F2 are declared as wire. The complement of F2 is named F2_b to illustrate a common industry practice for designating the complement of a signal (instead of appending _not).

The repeat loop provides the seven binary numbers after 000 for the truth table. The result of the simulation generates the output truth table displayed with the example. The truth table listed shows that the circuit is a full adder.

HDL Example 2.10 (Gate-Level Circuit)

```
// Gate-level description of circuit of Fig. 4.2
module Circuit_of_Fig_4_2 (A, B, C, F1, F2);
  input A, B, C;
  output F1, F2;
  wire  T1, T2, T3, F2_b, E1, E2, E3;
  or  g1 (T1, A, B, C);
  and g2 (T2, A, B, C);
  and g3 (E1, A, B);
  and g4 (E2, A, C);
  and g5 (E3, B, C);
  or  g6 (F2, E1, E2, E3);
  not g7 (F2_b, F2);
  and g8 (T3, T1, F2_b);
  or  g9 (F1, T2, T3);
endmodule

// Stimulus to analyze the circuit
module test_circuit;
  reg [2: 0] D;
  wire F1, F2;
  Circuit_of_Fig_4_2 (D[2], D[1], D[0], F1, F2);
  initial
   begin
    D = 3'b000;
    repeat (7) #10 D = D 1 1'b1;
   end
  initial
  $monitor ("ABC = %b F1 = %b F2 =%b", D, F1, F2);
endmodule
```

```
Simulation log:  ABC = 000 F1 = 0 F2 =0
ABC = 001 F1 = 1 F2 =0 ABC = 010 F1 = 1 F2 =0
ABC = 011 F1 = 0 F2 =1 ABC = 100 F1 = 1 F2 =0
ABC = 101 F1 = 0 F2 =1 ABC = 110 F1 = 0 F2 =1
ABC = 111 F1 = 1 F2 =1
```

Synchronous Sequential Logic

A block diagram of a sequential circuit is shown in Fig 2.30. It consists of a combinational circuit to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information. The binary information stored in these elements at any given time defines the state of the sequential circuit at that time. The sequential circuit receives binary information from external inputs that, together with the present state of the storage elements, determine the binary value of the outputs. These external inputs also determine the condition for changing the state in the storage elements. The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the inputs, but also of the present state of the storage elements. The next state of the storage elements is also a function of external inputs and the present state. Thus, **a sequential circuit is specified by a time sequence of inputs, outputs, and internal states** . In contrast, the outputs of combinational logic depend only on the present values of the inputs.

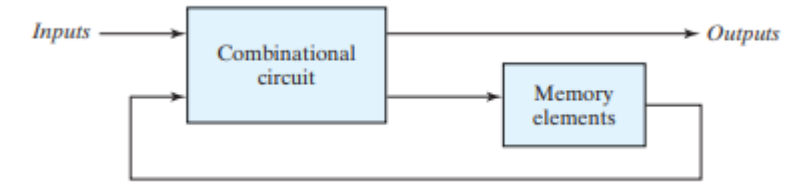


FIGURE 2.30 Block diagram of sequential circuit

Sequential Circuits Classification:

- There are two main types of sequential circuits.
- Classification is based on the timing of their signals.

Synchronous Sequential Circuits:

- Behavior is defined at discrete instants of time.
- Behavior determined by knowledge of signals at specific time points.

Asynchronous Sequential Circuits:

- Behavior depends on input signals at any instant of time.
- Behavior influenced by the order in which inputs change.

Storage Elements in Asynchronous Sequential Circuits:

- Commonly use time-delay devices.
- Storage capability varies with signal propagation time through the device.

Internal Propagation Delay:

- In practice, logic gate propagation delays are often sufficient for required delays.
- Eliminates the need for separate delay units.

Gate-Type Asynchronous Systems:

- Storage elements are logic gates.
- Propagation delay of these gates provides the necessary storage.

Asynchronous Sequential Circuit Nature:

- Can be regarded as a combinational circuit with feedback.
- Feedback among logic gates is a characteristic feature.

Potential for Instability:

- Due to feedback, asynchronous sequential circuits may become unstable at times.
- Stability must be carefully managed in the design.

- Synchronous sequential circuits rely on discrete clock pulses for synchronization, ensuring that storage elements are affected only at specific time instants.
- A clock generator generates periodic clock pulses denoted as "clock" or "clk," regulating when changes occur within the circuit.
- The clock pulses dictate when computational operations happen, while external inputs determine what changes affect storage elements and outputs.
- For instance, in binary addition, the sum is computed from input values and stored when a clock pulse arrives.
- These circuits are known as clocked sequential circuits and are prevalent in practical applications due to their stability and well-defined timing.
- Synchronous circuits break down their timing into separate, independent steps, simplifying their design and analysis.
- Flip-flops are storage elements in clocked sequential circuits.
- They store one binary bit of information each.
- In a stable state, flip-flop output is 0 or 1.
- Sequential circuits use multiple flip-flops.
- Flip-flops collectively store multi-bit data.
- Clock signals control flip-flop transitions.
- Each flip-flop has a unique identifier in a circuit.
- Flip-flops enable memory and state retention.
- They facilitate data processing and logic operations.
- Key components in digital circuit design and automation.

The block diagram of a synchronous clocked sequential circuit is shown in Fig. 2.31 .

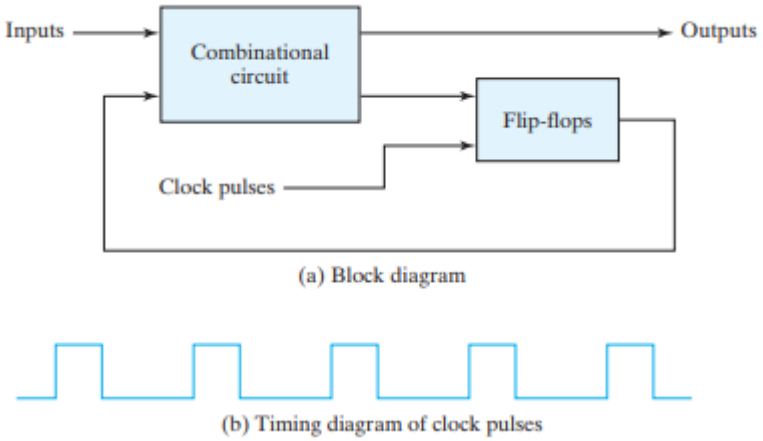


FIGURE 2.31 Synchronous clocked sequential circuit

- Combinational Logic Outputs: Outputs in digital circuits are determined by combinational logic functions of inputs and flip-flop values.
- Flip-Flop State Determination: The value stored in a flip-flop during a clock pulse is influenced by inputs and current flip-flop values.
- Clock Pulse Update: Flip-flop values are updated during a clock pulse.
- Timing Criticality: Combinational logic speed is crucial to meet clock pulse intervals, ensuring correct operation.
- Propagation Delays: Delays in signal propagation impact the minimum allowable clock pulse interval.
- Clock Pulse Trigger: Flip-flop state changes occur only during clock pulse transitions (e.g., from 0 to 1).
- Feedback Loop Disconnection: During inactive clock pulses, the feedback loop between flip-flop values and inputs is disrupted.
- Deterministic State Transition: Transitions between states occur at predetermined intervals defined by clock pulses.

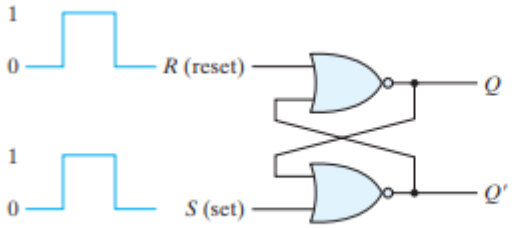
STORAGE ELEMENTS : LATCHES

- A storage element in a digital circuit can maintain a binary state indefinitely, until directed by an input signal to switch states.
- The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state.
- Storage elements that operate with signal levels are referred to as latches; those controlled by a clock transition are flip-flops.
- Latches are said to be level sensitive devices; flip-flops are edge-sensitive devices.
- The two types of storage elements are related because latches are the basic circuits from which all flip-flops are constructed. Although latches are useful for storing binary

information and for the design of asynchronous sequential circuits, they are not practical for use as storage elements in synchronous sequential circuits.

SR Latch

- The SR latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labelled S for set and R for reset.
- The SR latch constructed with two cross-coupled NOR gates is shown in Fig. 2.32. The latch has two useful states. When output Q = 1 and Q' = 0, the latch is said to be in the set state. When Q = 0 and Q' = 1, it is in the reset state.
- Outputs Q and Q' are normally the complement of each other. However, when both inputs are equal to 1 at the same time, a condition in which both outputs are equal to 0 (rather than be mutually complementary occurs. If both inputs are then switched to 0 simultaneously, the device will enter an unpredictable or undefined state or a metastable state.
- Under normal conditions, both inputs of the latch remain at 0 unless the state has to be changed. The application of a momentary 1 to the S input causes the latch to go to the set state.
- The S input must go back to 0 before any other changes take place, in order to avoid the occurrence of an undefined next state that results from the forbidden input condition. As shown in the function table of Fig. 2.32 (b) , two input conditions cause the circuit to be in the set state.



(a) Logic diagram

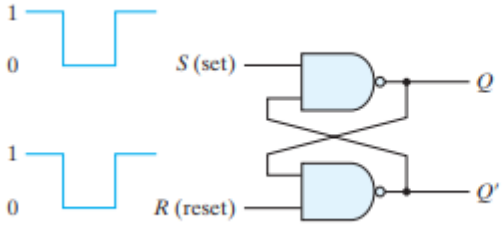
S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	(forbidden)

(b) Function table

FIGURE 2.32 SR latch with NOR gates

- The first condition (S = 1,R = 0) is the action that must be taken by input S to bring the circuit to the set state. R
- emoving the active input from S leaves the circuit in the same state.
- After both inputs return to 0, it is then possible to shift to the reset state by momentary applying a 1 to the R input.
- The 1 can then be removed from R, where upon the circuit remains in the reset state. Thus, when both inputs S and R are equal to 0, the latch can be in either the set or the reset state, depending on which input was most recently a 1.
- If a 1 is applied to both the S and R inputs of the latch, both outputs go to 0. This action produces an undefined next state, because the state that results from the input transitions depends on the order in which they return to 0.
- It also violates the requirement that outputs be the complement of each other. In normal operation, this condition is avoided by making sure that 1's is not applied to both inputs simultaneously.

The SR latch with two cross-coupled NAND gates is shown in Fig. 2.32 . It operates with both inputs normally at 1, unless the state of the latch has to be changed. The application of 0 to the S input causes output Q to go to 1, putting the latch in the set state. When the S input goes back to 1, the circuit remains in the set state. After both inputs go back to 1, we are allowed to change the state of the latch by placing a 0 in the R input. This action causes the circuit to go to the reset state and stay there even after both inputs return to 1. The condition that is forbidden for the NAND latch is both inputs being equal to 0 at the same time, an input combination that should be avoided.



(a) Logic diagram

S	R	Q	Q′	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	(forbidden)

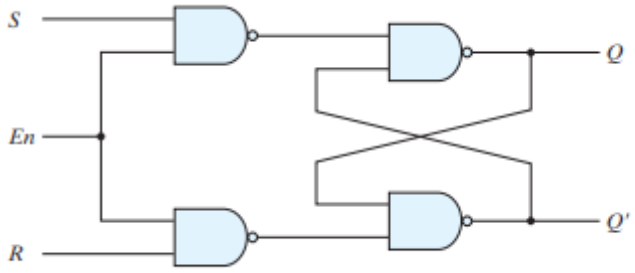
(b) Function table

FIGURE 2.32 SR latch with NAND gates

In comparing the NAND with the NOR latch, note that the input signals for the NAND require the complement of those values used for the NOR latch. Because the NAND latch requires a 0 signal to change its state, it is sometimes referred to as an S'R' latch.

The operation of the basic SR latch can be modified by providing an additional input signal that determines (controls) when the state of the latch can be changed by determining whether S and R (or S" and R") can affect the circuit. An SR latch with a control input is shown in Fig. 2.33 . It consists of the basic SR latch and two additional NAND gates. The control input En acts as an enable signal for the other two inputs. **The outputs of the NAND gates stay at the logic-1 level as long as the enable signal remains at 0.** This is the quiescent condition for the SR latch. When the enable input goes to 1, information from the S or R input is allowed to affect the latch. The set state is reached with S = 1, R = 0, and En = 1.

To change to the reset state, the inputs must be S = 0, R = 1, and En = 1. In either case, when En returns to 0, the circuit remains in its current state. The control input disables the circuit by applying 0 to En, so that the state of the output does not change regardless of the values of S and R . Moreover, when En = 1 and both the S and R inputs are equal to 0, the state of the circuit does not change. These conditions are listed in the function table accompanying the diagram.



(a) Logic diagram

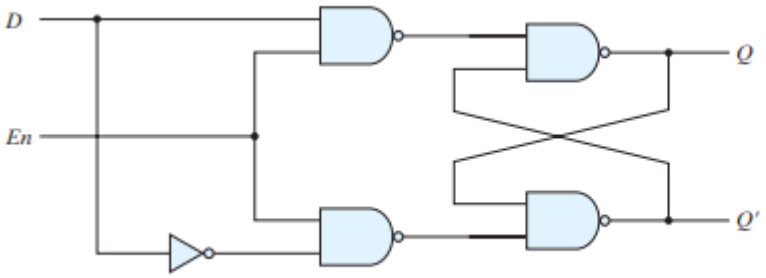
En	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

(b) Function table

FIGURE 2.33 SR latch with control inpu

D Latch (Transparent Latch)

One way to eliminate the undesirable condition of the indeterminate state in the SR latch is to ensure that inputs S and R are never equal to 1 at the same time. This is done in the D latch, shown in Fig. 2.34 . This latch has only two inputs: D (data) and En (enable). The D input goes directly to the S input, and its complement is applied to the R input. As long as the enable input is at 0, the cross-coupled SR latch has both inputs at the 1 level and the circuit cannot change state regardless of the value of D . The D input is sampled when En = 1. If D = 1, the Q output goes to 1, placing the circuit in the set state. If D = 0, output Q goes to 0, placing the circuit in the reset state.



(a) Logic diagram

En	D	Next state of Q
0	X	No change
1	0	Q = 0; reset state
1	1	Q = 1; set state

(b) Function table

FIGURE 2.34 D latch

- D Latch Purpose: The D latch is named for its ability to store data internally, making it useful as a temporary binary information storage device between a unit and its external environment.

- Data Transfer: Binary data present at the D (data) input of the D latch is copied to the Q (output) when the enable input is activated (asserted).
- Transparency: The D latch operates as a transparent latch while the enable input is active, meaning it mirrors changes in the data input, allowing a direct path from input D to the output Q.
- Data Retention: When the enable input is de-asserted, the D latch holds and stores the binary information that was present at the data input when the transition occurred.
- Enable Input Polarity: Depending on the circuit design, an inverter can be added to the enable input, making it either active low (0) or active high (1) for external control.

The graphic symbols for the various latches are shown in Fig. 2.35 . A latch is designated by a rectangular block with inputs on the left and outputs on the right. One output designates the normal output, and the other (with the bubble designation) designates the complement output. The graphic symbol for the SR latch has inputs S and R indicated inside the block.

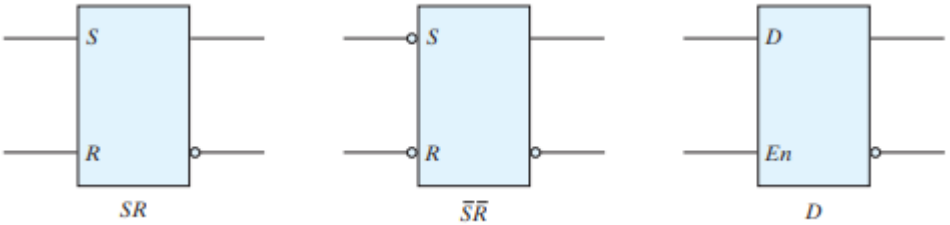


FIGURE 2.35 Graphic symbols for latches

STORAGE ELEMENTS : FLIP - FLOPS

The state of a latch or flip-flop is switched by a change in the control input. This momentary change is called a trigger, and the transition it causes is said to trigger the flip-flop. The D latch with pulses in its control input is essentially a flip-flop that is triggered every time the pulse goes to the logic-1 level. As long as the pulse input remains at this level, any changes in the data input will change the output and the state of the latch. Consequently, the inputs of the flip-flops are derived in part from the outputs of the same and other flip-flops. When latches are used for the storage elements, a serious difficulty arises. The state transitions of the latches start as soon as the clock pulse changes to the logic-1 level. The new state of a latch appears at the output while the pulse is still active. This output is connected to the inputs of the latches through the combinational circuit. If the inputs applied to the latches change while the clock pulse is still at the logic-1 level, the latches will respond to new values and a new output state may occur. The result is an unpredictable situation, since the state of the latches may keep changing for as long as the clock pulse stays at the active level.

Flip-flop Functionality: Flip-flops are essential components in sequential circuits, designed to work in synchronization with a common clock signal. Unlike latches, flip-flops are triggered only during specific signal transitions, ensuring reliable operation.

Latch Behavior: Latches respond to changes in their enable input, allowing output changes when the D input changes while the clock pulse is at logic 1, resulting in sensitivity to level changes.

Positive and Negative Transitions: Flip-flops distinguish between two clock signal transitions: the positive edge (0 to 1) and the negative edge (1 to 0). These transitions are crucial for proper operation. As shown in figure 2.36

Feedback Elimination: To transform a latch into a flip-flop, the feedback path inherent in latches must be eliminated to prevent output interference while input changes occur.

Dual-Latch Configuration: One approach to creating a flip-flop is using two latches in a special configuration. This setup isolates the flip-flop's output and safeguards it from changes during input transitions.

Transition-Triggered Flip-Flop: Another method is to design a flip-flop that triggers exclusively during signal transitions (0 to 1 or 1 to 0) of the clock signal. It remains disabled during the remainder of the clock pulse for reliable operation in sequential circuits.

Synchronization with Clock: Flip-flops are synchronized with a clock signal to ensure changes occur precisely during clock transitions, enhancing the predictability and reliability of sequential circuit operation.

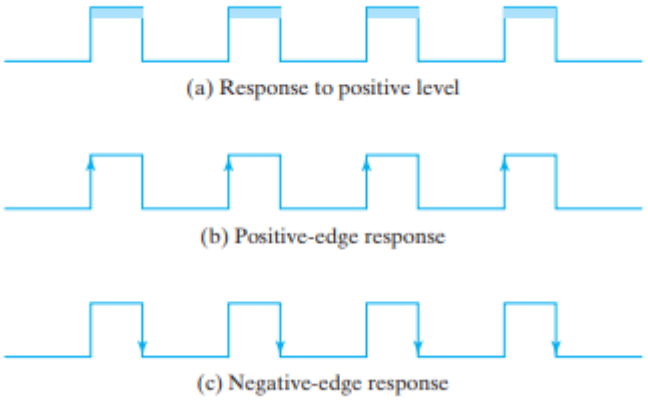


FIGURE 2.36 Clock response in latch and flip-flop

Edge-Triggered D Flip-Flop

Master and Slave Latches: The D flip-flop consists of two D latches: a master and a slave latch. These latches work together to store and transfer data.

Clock-Triggered Operation: The D flip-flop operates based on a clock signal (Clk). Changes in the output Q occur only at the negative edge (falling edge) of the clock signal. This means that data is sampled and transferred precisely at the moment when the clock transitions from 1 to 0.

Inverter and Clock Control: An inverter is used to control the enable inputs of the master and slave latches. When the clock signal is 0, the inverter outputs 1, enabling the slave latch and disabling the master latch. This means that the flip-flop is in a "latch" state when the clock is low, and no changes can occur in Q. As shown in the figure 2.37.

Data Transfer on Clock Transition: When the clock signal transitions from 0 to 1, the D input data is transferred to the master latch. However, the slave latch remains disabled because its enable input is 0 when the clock is high. Therefore, changes in the D input do not affect the flip-flop's output during this time.

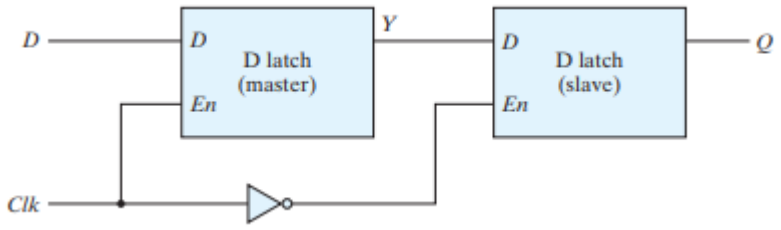


FIGURE 2.37 Master–slave D flip-flop

Output Update on Falling Clock Edge: The key operation occurs when the clock signal transitions from 1 to 0 (falling edge). At this moment, the master latch is disabled and isolated from the D input, while the slave latch is enabled. The value stored in the master latch (Y) is then transferred to the flip-flop's output Q. Therefore, any change in the output of the flip-flop can only occur during this specific clock transition.

The behavior of the master–slave flip-flop just described dictates that

(1) the output may change only once,

(2) a change in the output is triggered by the negative edge of the clock, and

(3) the change may occur only during the clock's negative level. The value that is produced at the output of the flip-flop is the value that was stored in the master stage immediately before the negative edge occurred .

It is also possible to design the circuit so that the flip-flop output changes on the positive edge of the clock. This happens in a flip-flop that has an additional inverter between the Clk terminal and the junction between the other inverter and input En of the master latch. Such a flip-flop is triggered with a negative pulse, so that the negative edge of the clock affects the master and the positive edge affects the slave and the output terminal.

Another construction of an edge-triggered D flip-flop uses three SR latches as shown in Fig. 2.38 . Two latches respond to the external D (data) and Clk (clock) inputs. The third latch provides the outputs for the flip-flop. The S and R inputs of the output latch are maintained at the logic-1 level when Clk = 0. This causes the output to remain in its present state. Input D may be equal to 0 or 1. If D = 0 when Clk becomes 1, R changes to 0. This causes the flip-flop to go to the reset state, making Q = 0. If there is a change in the D input while Clk = 1, terminal R remains at 0 because Q is 0. Thus, the flip-flop is locked out and is unresponsive to further changes in the input. When the clock returns to 0, R goes to 1, placing the output latch in the quiescent condition without changing the output. Similarly, if D = 1 when Clk goes from 0 to 1, S changes to 0. This causes the circuit to go to the set state, making Q = 1. Any change in D while Clk = 1 does not affect the output.

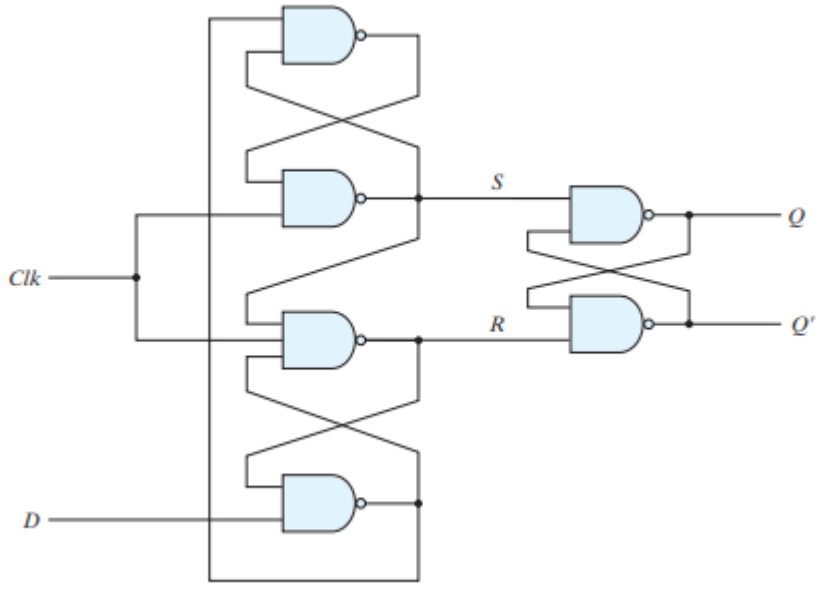


FIGURE 2.38 D-type positive-edge-triggered flip-flop

In sum, when the input clock in the positive-edge-triggered flip-flop makes a positive transition, the value of D is transferred to Q . A negative transition of the clock (i.e., from 1 to 0) does not affect the output, nor is the output affected by changes in D when Clk is in the steady logic-1 level or the logic-0 level. Hence, this type of flip-flop responds to the transition from 0 to 1 and nothing else.

The timing of the response of a flip-flop to input data and to the clock must be taken into consideration when one is using edge-triggered flip-flops. There is a minimum time called the setup time during which

the D input must be maintained at a constant value prior to the occurrence of the clock transition. Similarly, there is a minimum time called the hold time during which the D input must not change after the application of the positive transition of the clock. The propagation delay time of the flip-flop is defined as the interval between the trigger edge and the stabilization of the output to a new state.

The graphic symbol for the edge-triggered D flip-flop is shown in Fig. 2.39 . It is similar to the symbol used for the D latch, except for the arrowhead-like symbol in front of the letter Clk, designating a dynamic input. The dynamic indicator (>) denotes the fact that the flip-flop responds to the edge transition of the clock. A bubble outside the block adjacent to the dynamic indicator designates a negative edge for triggering the circuit.

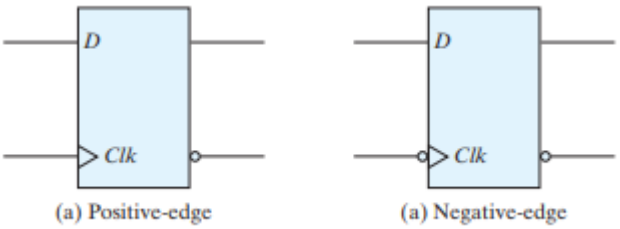


FIGURE 2.39 Graphic symbol for edge-triggered D flip-flop

Other Flip-Flops

Other types of flip-flops can be constructed by using the D flip-flop and external logic. Two flip-flops less widely used in the design of digital systems are the JK and T flip-flops.

There are three operations that can be performed with a flip-flop: Set it to 1, reset it to 0, or complement its output. With only a single input, the D flip-flop can set or reset the output, depending on the value of the D input immediately before the clock transition. Synchronized by a clock signal, the JK flip-flop has two inputs and performs all three operations. The circuit diagram of a JK flip-flop constructed with a D flip-flop and gates is shown in Fig. 2.40 (a). The J input sets the flip-flop to 1, the K input resets it to 0, and when both inputs are enabled, the output is complemented. This can be verified by investigating the circuit applied to the D input:

$$D = JQ' + K'Q$$

When J = 1 and K = 0, D = Q' + Q = 1, so the next clock edge sets the output to 1. When J = 0 and K = 1, D = 0, so the next clock edge resets the output to 0. When both J = K = 1 and D = Q', the next clock edge complements the output. When both J = K = 0 and D = Q, the clock edge leaves the output unchanged. The graphic symbol for the JK flip-flop is shown in Fig. 2.40 (b). It is similar to the graphic symbol of the D flip-flop, except that now the inputs are marked J and K .

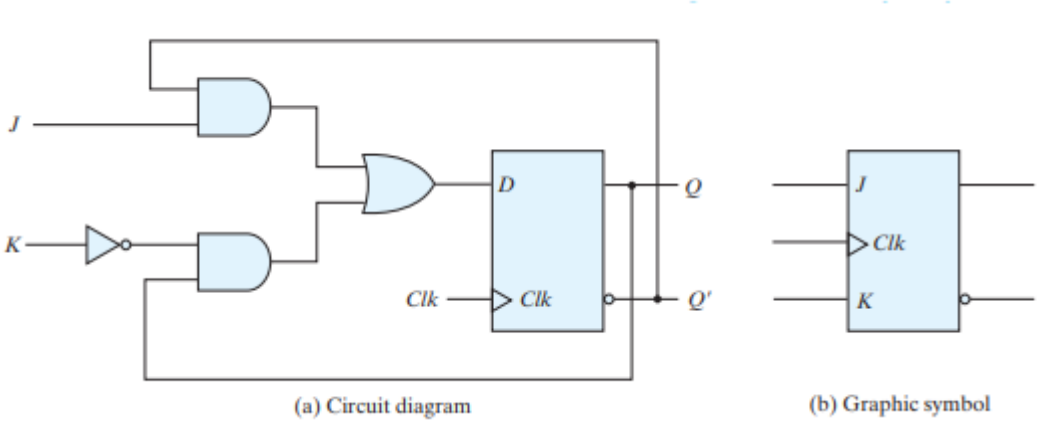


FIGURE 2.40 JK flip-flop

The T (toggle) flip-flop is a complementing flip-flop and can be obtained from a JK flip-flop when inputs J and K are tied together. This is shown in Fig. 2.41 (a).

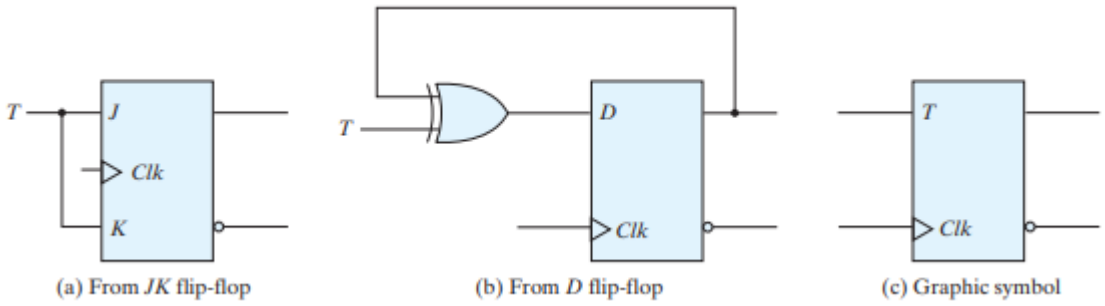


FIGURE 2.41 T flip-flop

When T = 0 (J = K = 0), a clock edge does not change the output. When T = 1 (J = K = 1), a clock edge complements the output. The complementing flip-flop is useful for designing binary counters. The T flip-flop can be constructed with a D flip-flop and an exclusive-OR gate as shown in Fig. 2.41 (b). The expression for the D input is

$$D = T \oplus Q = TQ' + T'Q$$

When T = 0, D = Q and there is no change in the output. When T = 1, D = Q" and the output complements. The graphic symbol for this flip-flop has a T symbol in the input.

Characteristic Tables

A characteristic table defines the logical properties of a flip-flop by describing its operation in tabular form. The characteristic tables of three types of flip-flops are presented in Table 2.10 . They define the next state .

Table 2.10 Flip-Flop Characteristic Tables

JK Flip-Flop

J	K	Q(t + 1)	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

D Flip-Flop

D	Q(t + 1)	
0	0	Reset
1	1	Set

T Flip-Flop

T	Q(t + 1)	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

as a function of the inputs and the present state. Q (t) refers to the present state (i.e., the state present prior to the application of a clock edge). Q(t + 1) is the next state one clock period later. Note that the clock edge input is not included in the characteristic table, but is implied to occur between times t and t + 1. Thus, Q(t) denotes the state of the flip-flop immediately before the clock edge, and Q(t + 1) denotes the state that results from the clock transition.

The characteristic table for the JK flip-flop shows that the next state is equal to the present state when inputs J and K are both equal to 0. This condition can be expressed as Q(t + 1) = Q(t), indicating that the clock produces no change of state. When K = 1 and J = 0, the clock resets the flip-flop and Q(t + 1) = 0. With J = 1 and K = 0, the flip-flop sets and Q(t + 1) = 1. When both J and K are equal to 1, the next state changes to the complement of the present state, a transition that can be expressed as Q(t + 1) = Q'(t).

The next state of a D flip-flop is dependent only on the D input and is independent of the present state. This can be expressed as Q(t + 1) = D. It means that the next-state value is equal to the value of D . Note that the D flip-flop does not have a "no-change" condition.

The characteristic table of the T flip-flop has only two conditions: When T = 0, the clock edge does not change the state; when T = 1, the clock edge complements the state of the flip-flop.

The logical properties of a flip-flop, as described in the characteristic table, can be expressed algebraically with a characteristic equation. For the D flip-flop, we have the characteristic equation

$$\mathbf{Q(t + 1) = D}$$

which states that the next state of the output will be equal to the value of input D in the present state. The characteristic equation for the JK flip-flop can be derived from the characteristic table or from the circuit of Fig. 2.40 . We obtain

$$Q(t + 1) = JQ' + K'Q$$

where Q is the value of the flip-flop output prior to the application of a clock edge. The characteristic equation for the T flip-flop is obtained from the circuit of Fig. 2.41 :

$$D = T \oplus Q = TQ' + T'Q$$

When T = 0, D = Q and there is no change in the output. When T = 1, D = Q' and the output complements. The graphic symbol for this flip-flop has a T symbol in the input.

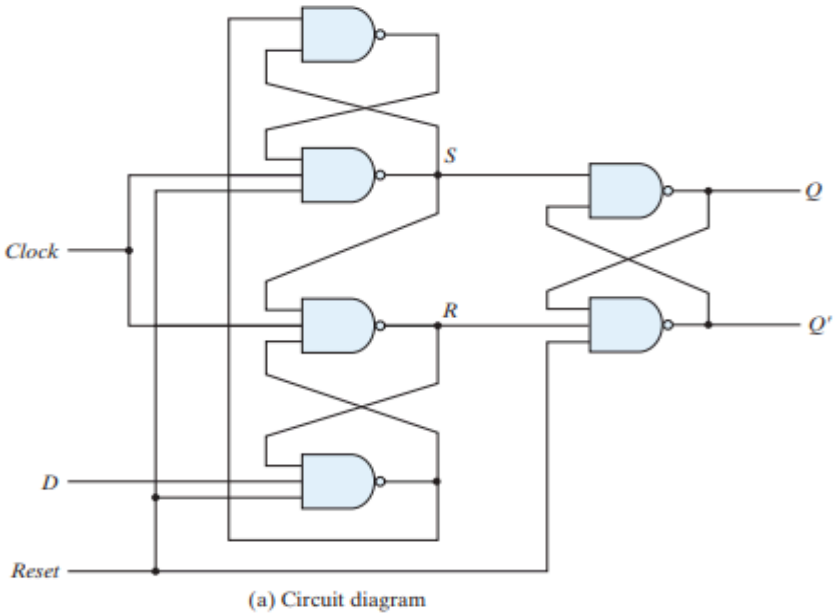
Direct Inputs

Some flip-flops have asynchronous inputs that are used to force the flip-flop to a particular state independently of the clock. The input that sets the flip-flop to 1 is called preset or direct set . The input that clears the flip-flop to 0 is called clear or direct reset . When power is turned on in a digital system, the state of the flip-flops is unknown. The direct inputs are useful for bringing all flip-flops in the system to a known starting state prior to the clocked operation.

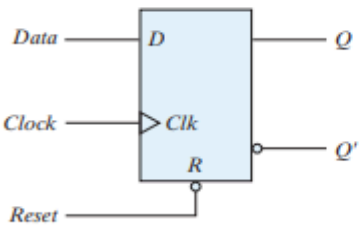
A positive-edge-triggered D flip-flop with active-low asynchronous reset is shown in Fig. 2.42. When the reset input is 0, it forces output Q' to stay at 1, which, in turn, clears output Q to 0, thus resetting the flip-flop. Two other connections from the reset input ensure that the S input of the third SR latch stays at logic 1 while the reset input is at 0, regardless of the values of D and Clk .

The graphic symbol for the D flip-flop with a direct reset has an additional input marked with R . The bubble along the input indicates that the reset is active at the logic-0 level. Flip-flops with a direct set use the symbol S for the asynchronous set input.

The function table specifies the operation of the circuit. When R = 0, the output is reset to 0. This state is independent of the values of D or Clk . Normal clock operation can proceed only after the reset input goes to logic 1. The clock at Clk is shown with an upward arrow to indicate that the flip-flop triggers on the positive edge of the clock. The value in D is transferred to Q with every positive-edge clock signal, provided that R = 1.



(a) Circuit diagram



(b) Graphic symbol

R	Clk	D	Q	Q'
0	X	X	0	1
0	↑	0	0	1
0	↑	1	1	0

(b) Function table

FIGURE 2.42 D flip-flop with asynchronous reset