

ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028



A T M E

College of Engineering

DEPARTMENT OF CSE- CYBER SECURITY



LECTURE NOTES

**COURSE: DIGITAL DESIGN AND COMPUTER
ORGANISATION**

COURSE CODE: 1BCS303

SEMESTER: III

COURSE COORDINATOR

Mrs. Razikha Amreen M I

(ACADEMIC YEAR 2026-27)

INSTITUTIONAL VISION AND MISSION

Vision

Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

DEPARTMENT VISION AND MISSION

Vision

To nurture competent and ethical cyber security professionals capable of contributing to secure and resilient digital ecosystems.

Mission

M1: To impart strong foundational knowledge in cyber security through effective teaching–learning practices and a continuously evolving curriculum that addresses emerging cyber threats and technologies.

M2: To promote research, innovation, and industry collaboration in emerging areas of cyber security to address real-world digital security challenges.

M3: To instill professional ethics, teamwork, leadership qualities, and a sense of social responsibility in addressing real-world cyber challenges.

M4: To foster a culture of lifelong learning among students by encouraging higher studies, professional development, and active engagement with the global cyber security community.

Program Educational Objectives

PEO1 –Excel as cyber security professionals or entrepreneurs by applying strong fundamentals of cyber security engineering.

PEO2 – Pursue higher studies, research, and professional certifications while engaging in lifelong learning to keep pace with evolving cyber technologies and threats.

PEO3 – Exhibit leadership, teamwork, professional ethics, and social responsibility while addressing cyber security challenges in multidisciplinary environments.

Program Specific Outcomes

PSO1 – Apply principles of computer science, programming, algorithms, operating systems, and computer networks to design and develop efficient and reliable computing solutions.

PSO2 – Identify, analyze, and mitigate cyber threats and vulnerabilities using techniques related to network security, ethical hacking, penetration testing, and digital forensics.

PSO3 – Apply cyber security policies, cyber laws, risk management practices, and incident response strategies to ensure the protection and resilience of digital systems.

Module 2

Combinational Logic

Introduction

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.

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. 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.



Block Diagram of Combinational Circuit

For n input variables, there are 2^n possible combinations of the binary inputs. For each possible input combination, there is one possible value for each output variable. Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables. A combinational circuit also can be described by m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

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.

Examples of Combinational Logic Circuits: Adders, Subtractors, Decoders, Encoders, Multipliers, Dividers, Display Drivers, Keyboard Encoders, Etc...

Design Procedure

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

- 1 From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each.
- 2 Derive the truth table that defines the required relationship between inputs and outputs.
- 3 Obtain the simplified Boolean functions for each output as a function of the input variables.
- 4 Draw the logic diagram and verify the correctness of the design (manually or by simulation).

A truth table for a combinational circuit consists of input columns and output columns. The input columns are obtained from the 2^n binary numbers for the n input variables. The binary values for the outputs are determined from the stated specifications. The output functions specified in the truth table

give the exact definition of the combinational circuit. It is important that the verbal specifications be interpreted correctly in the truth table, as they are often incomplete, and any wrong interpretation may result in an incorrect truth table.

The output binary functions listed in the truth table are simplified by any available method, such as algebraic manipulation, the map method, or a computer-based simplification program. Frequently, there is a variety of simplified expressions from which to choose. In a particular application, certain criteria will serve as a guide in the process of choosing an implementation. A practical design must consider such constraints as the number of gates, number of inputs to a gate, propagation time of the signal through the gates, number of interconnections, limitations of the driving capability of each gate (i.e., the number of gates to which the output of the circuit may be connected), and various other criteria that must be taken into consideration when designing integrated circuits.

Example: Consider the BCD to Excess-3 code conversion.

The bit combinations assigned to the BCD and excess-3 codes are listed in Table below

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 variables is shown in Table above. Note that four binary variables may have 16 bit combinations, but only 10 are listed in the truth table. The six bit combinations not listed for the input variables are don't-care combinations. These values have no meaning in BCD and we assume that they will never occur in actual operation of the circuit.

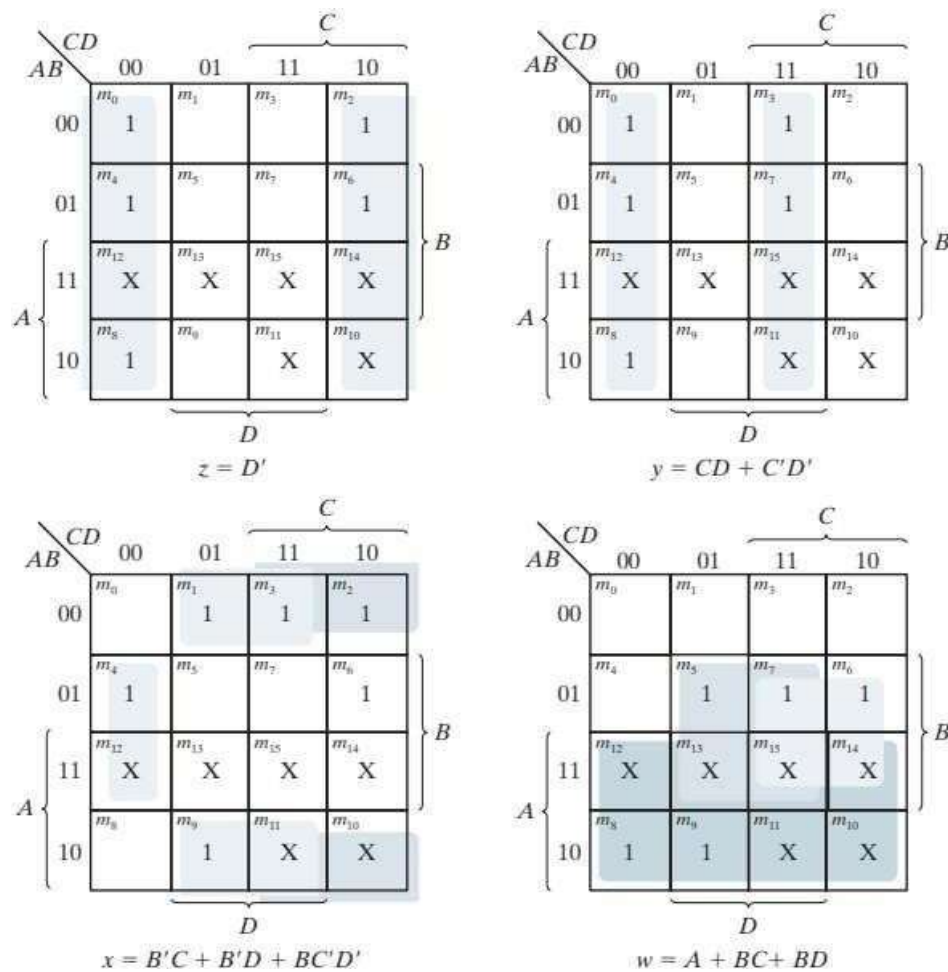
The maps in Fig. 4.3 are plotted to obtain simplified Boolean functions for the outputs. Each one of the four maps represents one of the four outputs of the circuit as a function of the four input variables. The 1's marked inside the squares are obtained from the minterms that make the output equal to 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. Following are the expressions obtained using K map.

$$z = D'$$

$$y = CD + C'D' = CD + (C + D)'$$

$$\begin{aligned} x &= B'C + B'D + BC'D' = B'(C + D) + BC'D' \\ &= B'(C + D) + B(C + D)' \end{aligned}$$

$$w = A + BC + BD = A + B(C + D)$$



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

Binary 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:

$$0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, \text{ and } 1 + 1 = 10$$

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.

Half Adder

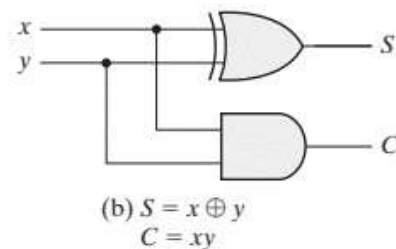
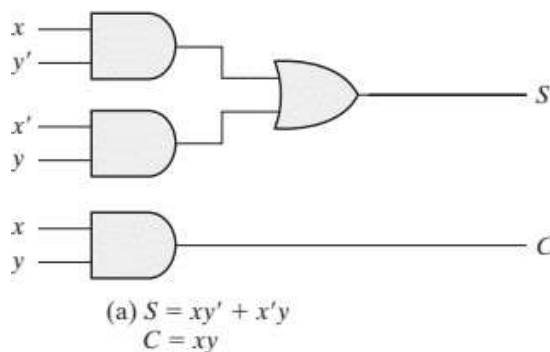
From the verbal explanation of a half adder, we find that this circuit needs two binary inputs and two

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 below. The C output is 1 only when both inputs are 1. The S output represents the least significant bit of the sum.

Half Adder

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



The logic diagram of the half adder implemented in sum of products is shown in Fig.(a). It can be also implemented with an exclusive-OR and an AND gate as shown in Fig.(b)

Full Adder

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. s . 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 below. 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.

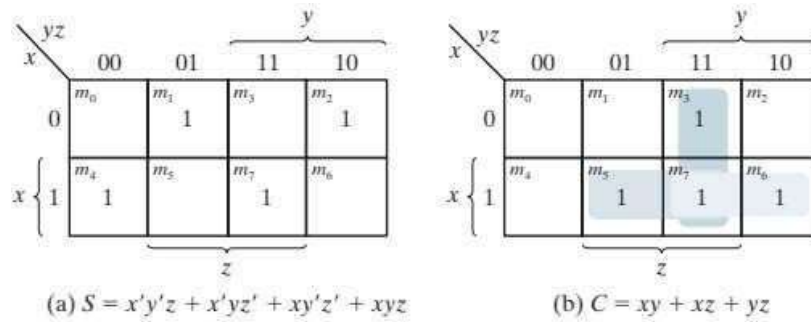
Full Adder

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

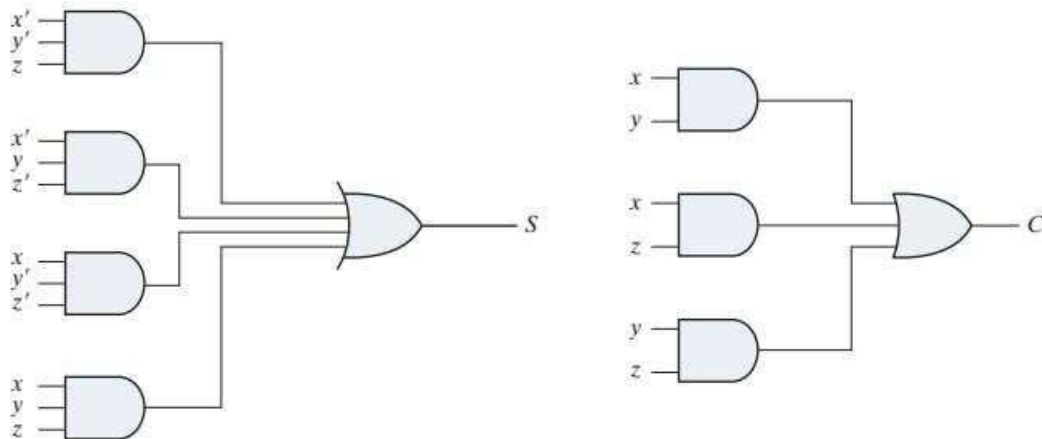
The maps for the outputs of the full adder are shown in Fig below. The simplified expressions are

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

$$C = xy + xz + yz$$



The logic diagram for the full adder implemented in sum-of-products form is shown in Fig below.



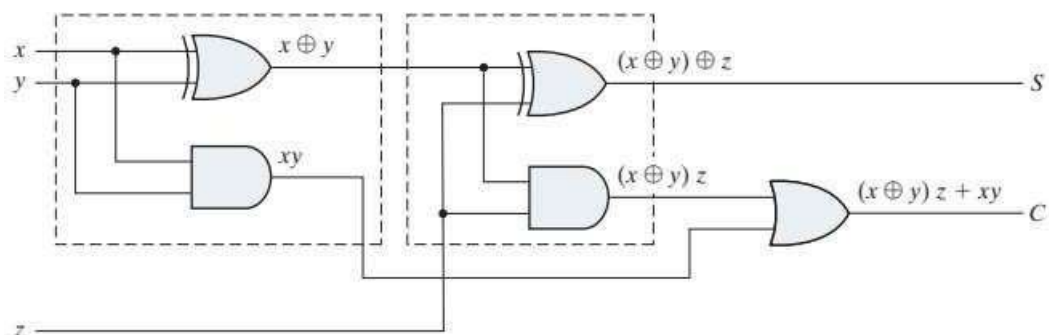
Implementation of full adder in sum-of-products form

It can also be implemented with two half adders and one OR gate, as shown in Fig below. 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

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



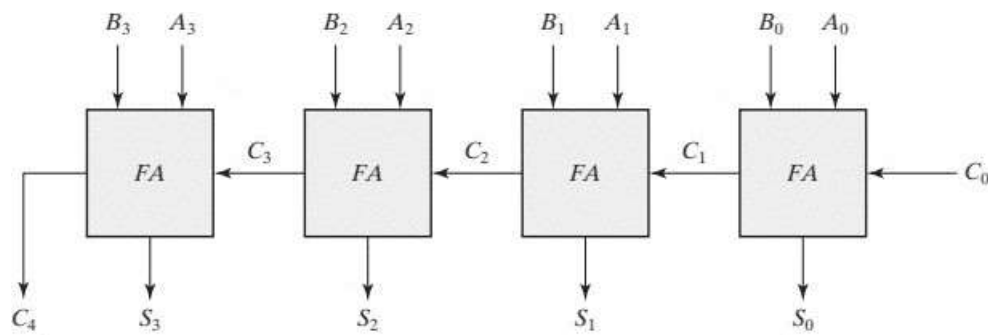
Implementation of full adder with two half adders and an OR gate

Binary Adder

A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be

constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain. Addition of n-bit numbers requires a chain of n full

adders or a chain of one-half adder and $n-1$ full adders. Figure below shows the interconnection of four full-adder (FA) circuits to provide a four-bit binary ripple carry adder.



Four-bit 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. All the carries must be generated for the correct sum bits to appear at the outputs.

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 by the classical method would require a truth table with $2^9 = 512$ entries, since there are nine inputs to the circuit. By using an iterative method of cascading a standard function, it is possible to obtain a simple and straightforward implementation.

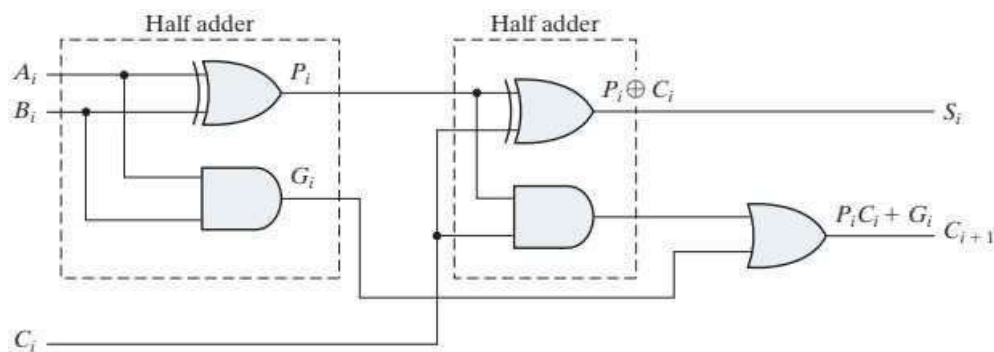
Carry Propagation

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 S_3 in Four-Bit Adder (above Fig). Inputs A_3 and B_3 are available as soon as input signals are applied to the adder. However, input carry C_3 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 below for convenience. The input and output variables use the subscript i to denote a typical stage of the adder. The signals at P_i and G_i settle to their steady-state values after they propagate through their respective gates. These two signals are common to all half adders and depend on only the input augend and addend bits. The signal from the input carry C_i to the output carry C_{i+1} propagates through an AND gate and an OR gate, which constitute two gate levels. If there are four full adders in the adder, the output carry C_4 would have $2 * 4 = 8$ gate levels from C_0 to C_4 . For an n -bit adder, there are $2n$ gate levels for the carry to propagate from input to output.



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. 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. Solution is to reduce the propagation time. 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 above Fig. If we define two new binary variables

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

the output sum and carry can respectively be expressed as

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

G_i 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 $i + 1$ (assertion of C_i will propagate to an assertion of C_{i+1}).

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

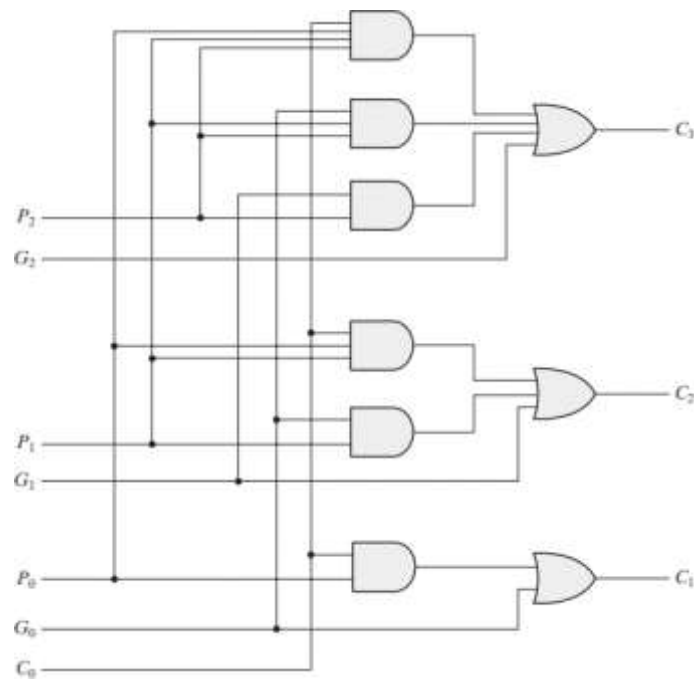
$$C_0 = \text{input carry}$$

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$

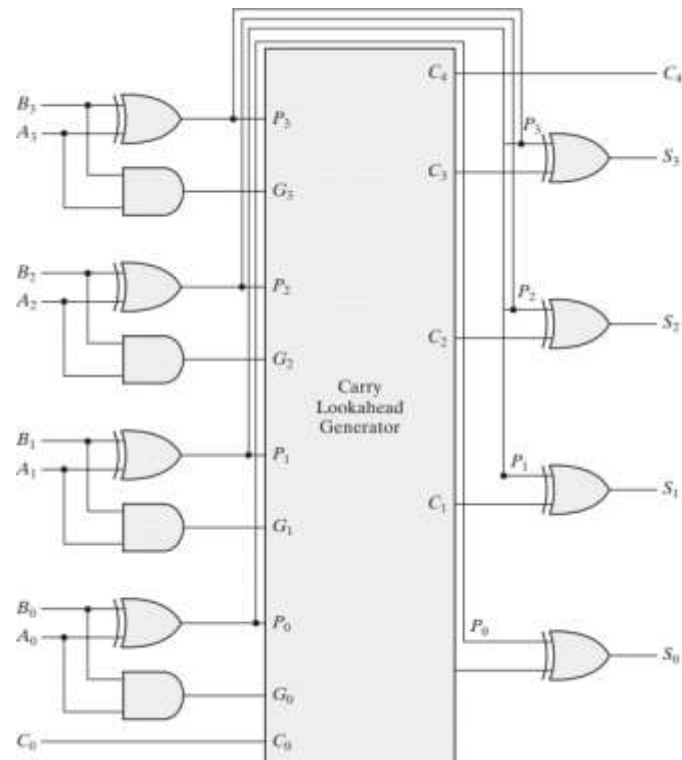
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 = P_2 P_1 P_0 C_0$$

The three Boolean functions for C_1 , C_2 , and C_3 are implemented in the carry lookahead generator shown in Fig below.



Logic diagram of carry lookahead generator

The construction of a four-bit adder with a carry lookahead scheme is shown in Fig below. Each sum output requires two exclusive-OR gates. The output of the first exclusive-OR gate generates the P_i variable, and the AND gate generates the G_i variable. The carries are propagated through the carry lookahead generator. Thus, outputs S_1 through S_3 have equal propagation delay times

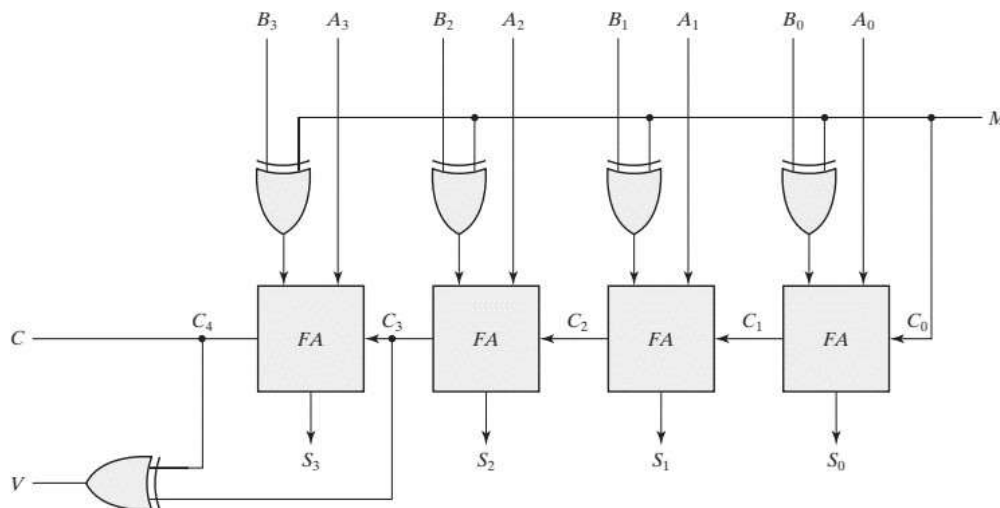


Four-bit adder with carry lookahead

Binary Subtractor

The subtraction of unsigned binary numbers can be done most conveniently by means of complements. Remember that the subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A . The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits. The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry.

The circuit for subtracting $A - B$ consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C_0 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 .



Four-bit adder-subtractor (with overflow detection)

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 above Fig. 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 $B \oplus 0 = B$. 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 $B \oplus 1 = \bar{B}$ 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 . (The exclusive-OR with output V is for detecting an overflow.)

Overflow

When two numbers with n digits each are added and the sum is a number occupying $n + 1$ digits, we say that an overflow occurred. This is true for binary or decimal numbers, signed or unsigned. When the addition is performed with paper and pencil, an overflow is not a problem, since there is no limit by the width of the page to write down the sum. Overflow is a problem in digital computers because the number of bits that hold the number is finite and a result that contains $n + 1$ bits cannot be accommodated by an n -bit word. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set that can then be checked by the user.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In the case of signed numbers, two details are important: the leftmost bit always represents the sign, and negative numbers are in 2's-complement form. When two

signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

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

Consider the following example: Two signed binary numbers, +70 and +80, are stored in two eight-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary -128. Since the sum of the two numbers is +150, it exceeds the capacity of an eight-bit register. This is also true for -70 and -80.

carries:	0 1	carries:	1 0
+70	0 1000110	-70	1 0111010
+80	0 1010000	-80	1 0110000
<u>+150</u>	<u>1 0010110</u>	<u>-150</u>	<u>0 1101010</u>

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.

Decoders

A decoder is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. If the n -bit coded information has unused combinations, the decoder may have fewer than 2^n outputs. The decoders presented here are called n -to- m -line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) minterms of n input variables. Each combination of inputs will assert a unique output. The name decoder is also used in conjunction with other code converters, such as a BCD-to-seven-segment decoder.

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

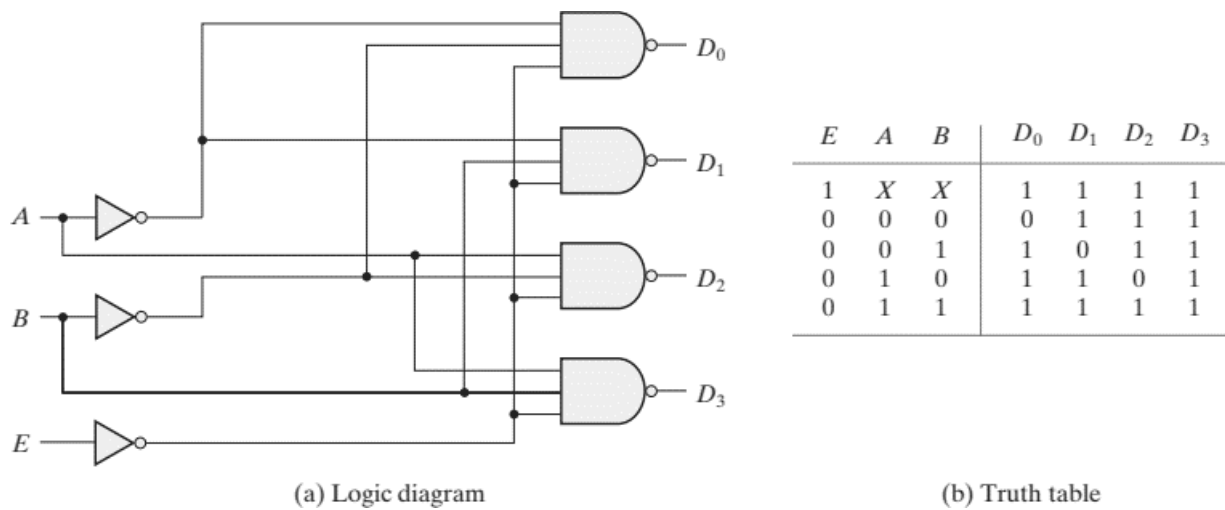
Truth Table of a Three-to-Eight-Line Decoder

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 above. For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1. The output whose value is equal to 1 represents the minterm equivalent of the binary number currently available in the

input lines.

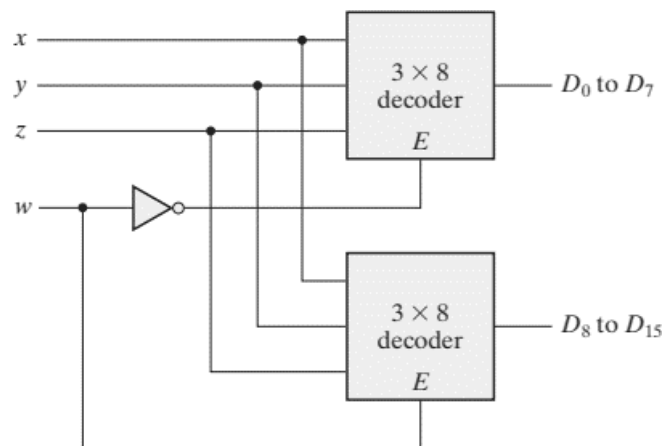
Some decoders are constructed with NAND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder minterms in their complemented form. Furthermore, decoders include one or more enable inputs to control the circuit operation. A two-to-four-line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



Two-to-four-line decoder with enable input

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.

Decoders with enable inputs can be connected together to form a larger decoder circuit. Figure below 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.



4 X 16decoder constructed with two 3 X 8decoders

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.

Combinational Logic Implementation using Decoders

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. This procedure will be illustrated by an example that implements a full-adder circuit.

Consider the truth table of Full adder-

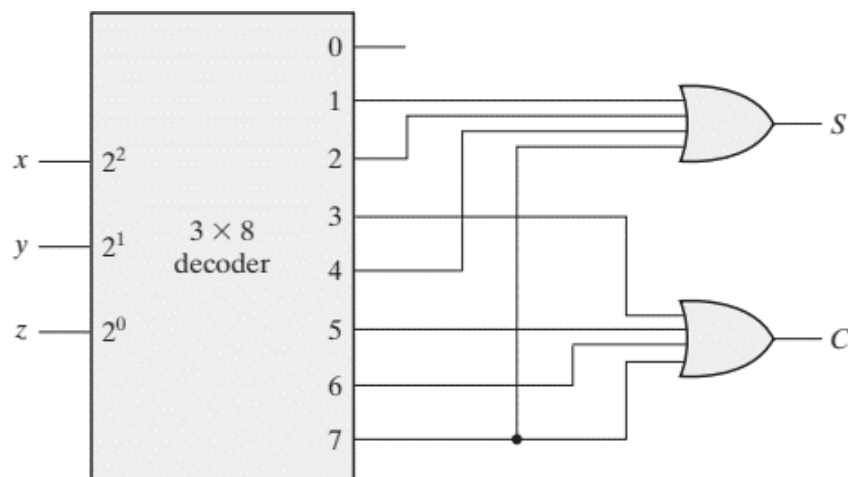
Full Adder				
<i>x</i>	<i>y</i>	<i>z</i>	<i>C</i>	<i>S</i>
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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 below. 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.



Implementation of a full adder with a decoder

Exercise:

Using a decoder and external gates, design the combinational circuit defined by the following three Boolean functions:

(a) $F_1 = x'yz' + xz$

$$F_2 = xy'z' + x'y$$

$$F_3 = x'y'z' + xy$$

(b) $F_1 = (y' + x)z$

$$F_2 = y'z' + x'y + yz'$$

$$F_3 = (x + y)z$$

Encoders

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has $2n$ (or fewer) input lines and n output lines. The output lines, as an aggregate, generate the binary code corresponding to the input value. An example of an encoder is the octal-to-binary encoder whose truth table is given in Table

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

$$z = D_1 + D_3 + D_5 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$x = D_4 + D_5 + D_6 + D_7$$

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 defined in Table above 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 D_6 has higher priority than D_3 .

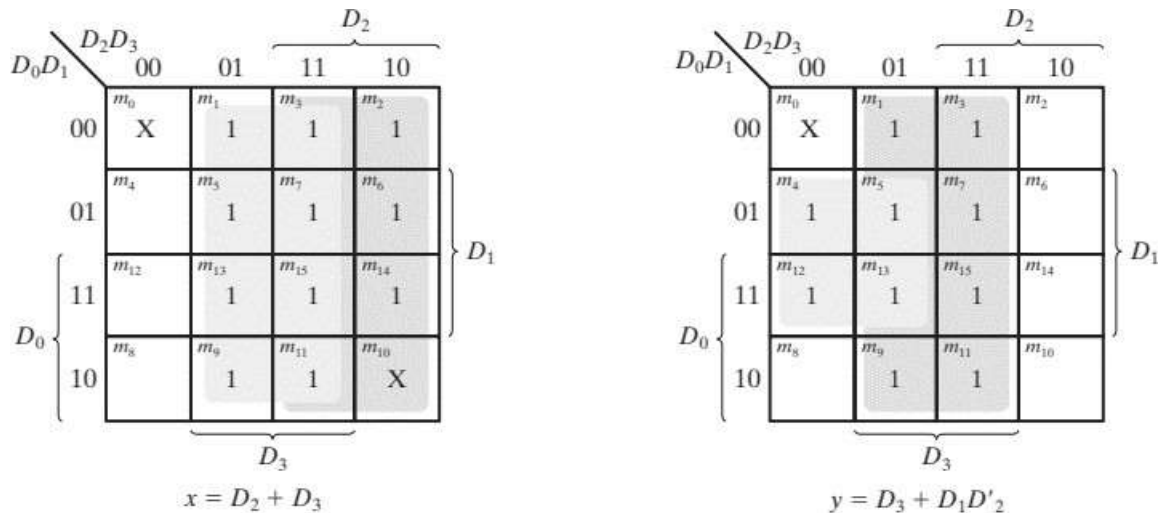
Priority Encoder

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. 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, $X100$ represents the two minterms 0100 and 1100.

Truth Table of a Priority Encoder

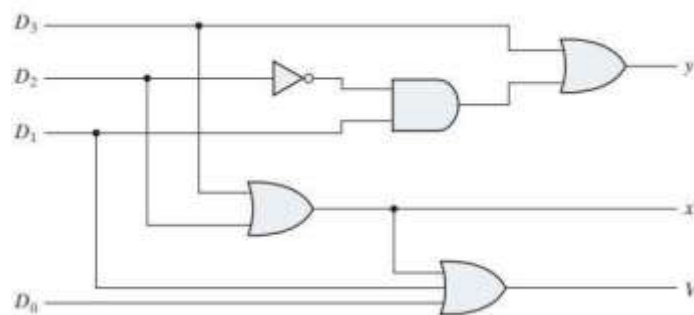
Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

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



The maps for simplifying outputs x and y are shown above. 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

$$\begin{aligned} x &= D_2 + D_3 \\ y &= D_3 + D_1 D'_2 \\ V &= D_0 + D_1 + D_2 + D_3 \end{aligned}$$



Four-input priority encoder

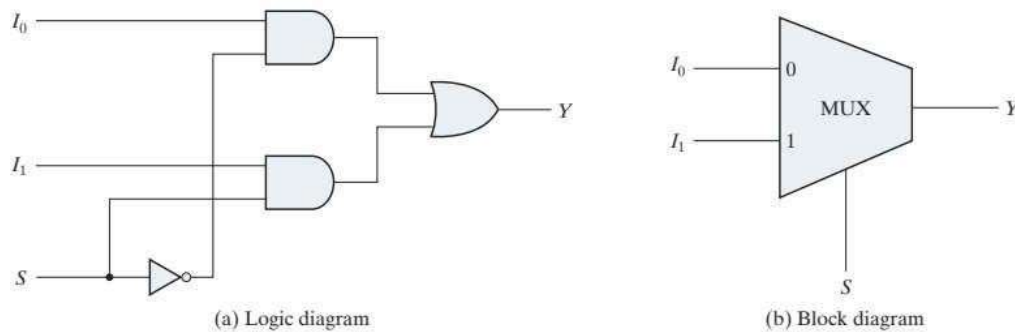
Multiplexers

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.

Two-to-One-line multiplexer

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

enabled and I_1 has a path to the output. The multiplexer acts like an electronic switch that selects one of two sources. The block diagram of a multiplexer is sometimes depicted by a wedge-shaped symbol, as shown in Fig. The multiplexer is often labeled "MUX" in block diagrams.

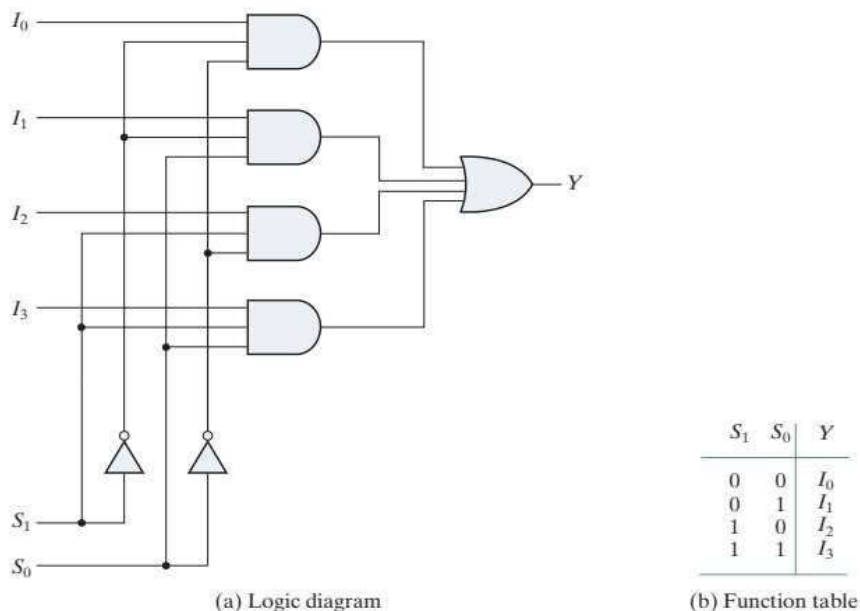


Two-to-One-line multiplexer

Four-to-One-line multiplexer

A four-to-one-line multiplexer is shown in Fig below. Each of the four inputs, I_0 through I_3 , 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.

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.



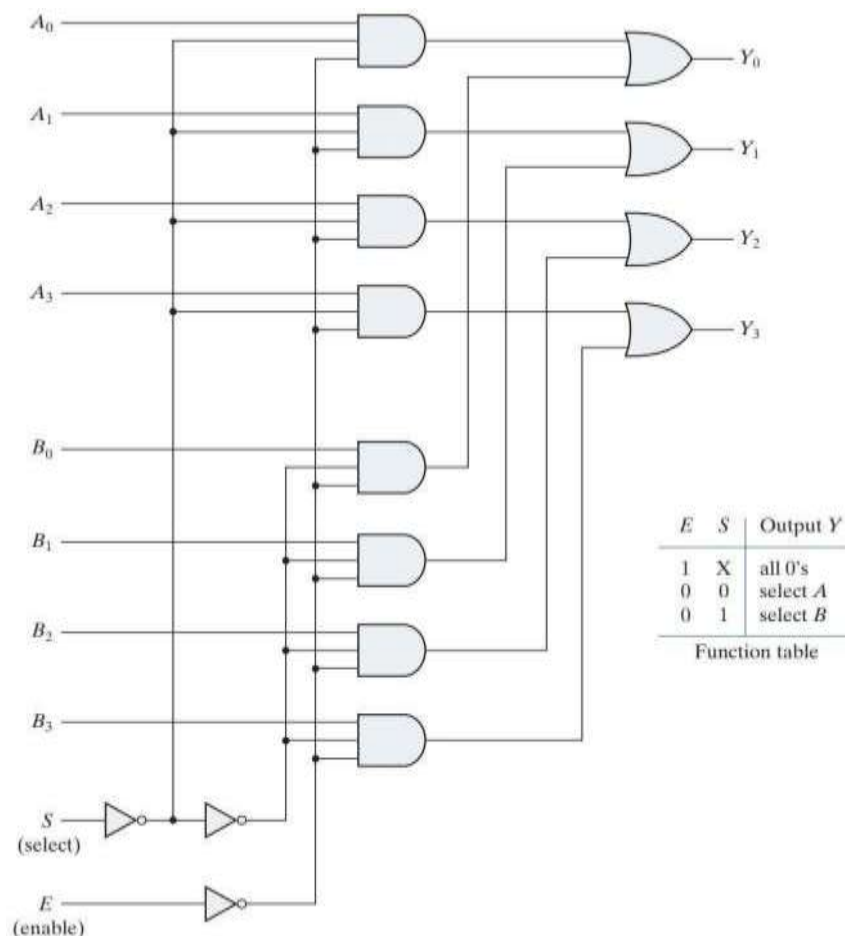
Four-to-One-line Multiplexer

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 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.

Quadruple two-to-one-line multiplexer

Multiplexer circuits can be combined with common selection inputs to provide multiple-bit selection logic. As an illustration, a quadruple 2-to-1-line multiplexer is shown in Fig below. 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 .



Quadruple two-to-one-line multiplexer

Boolean Function Implementation using MUX

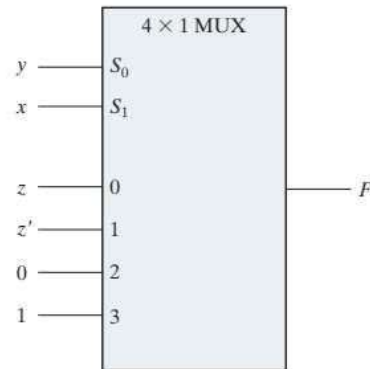
2 Consider the Boolean function $F(x, y, z) = \Sigma(1, 2, 6, 7)$.

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

particular example shows all four possibilities that can be obtained for the data inputs.

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(a) Truth table



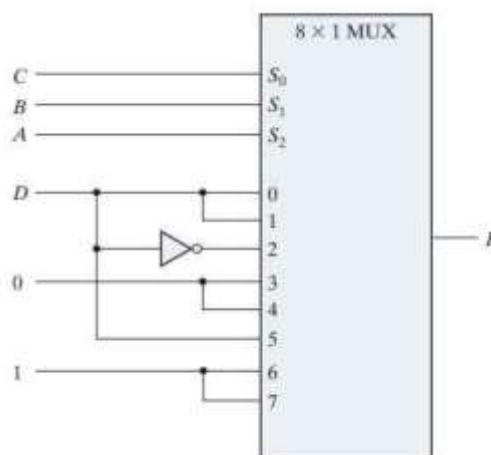
(b) Multiplexer implementation

Implementing a Boolean function with a multiplexer

- 3 Consider the implementation of the Boolean function $F(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$

This function is implemented with a multiplexer with three selection inputs as shown in Fig below. Note that the first variable A must be connected to selection input S_2 so that A, B, and C correspond to selection inputs S_2 , S_1 , and S_0 respectively. The values for the data inputs are determined from the truth table listed in the figure. The corresponding data line number is determined from the binary combination of ABC. For example, the table shows that when $ABC = 101$, $F = D$, so the input variable D is applied to data input 5.

A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



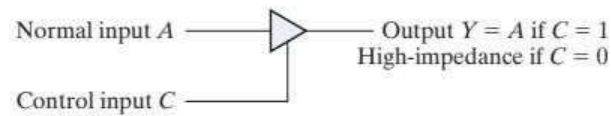
Implementing a four-input function with a multiplexer

Three-State Gates

A multiplexer can be constructed with three-state gates—digital circuits that exhibit three states. Two of the states are signals equivalent to logic 1 and logic 0 as in a conventional gate. The third state is a high-impedance state in which (1) the logic behaves like an open circuit, which means that the output appears to be disconnected, (2) the circuit has no logic significance, and (3) the circuit connected to the output of the three-state gate is not affected by the inputs to the gate. Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used is the buffer gate. The graphic symbol for a three-state buffer gate is shown in Fig below. 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

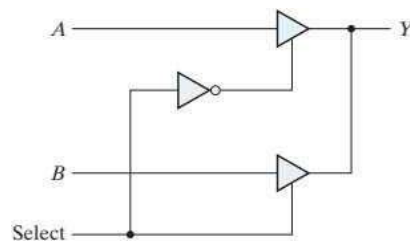
gate behaves like a conventional buffer, with the output equal to the normal

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.



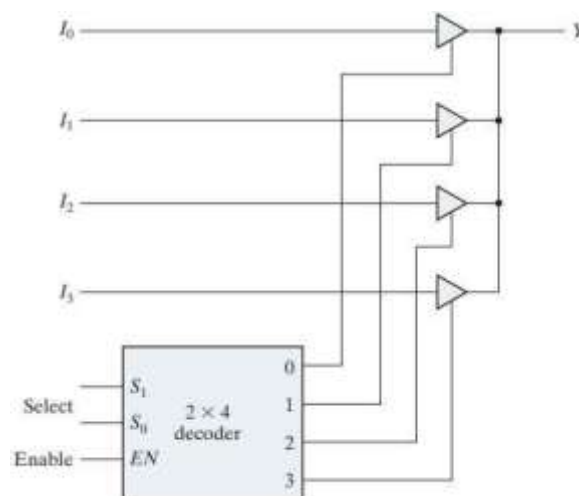
Graphic symbol for a three-state buffer

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



2-to-1-line mux using three state gates

The construction of a four-to-one-line multiplexer is shown in Fig below. 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 I_0 through 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 highimpedance 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 threestate buffers will be active, depending on the binary value in the select inputs of the decoder.



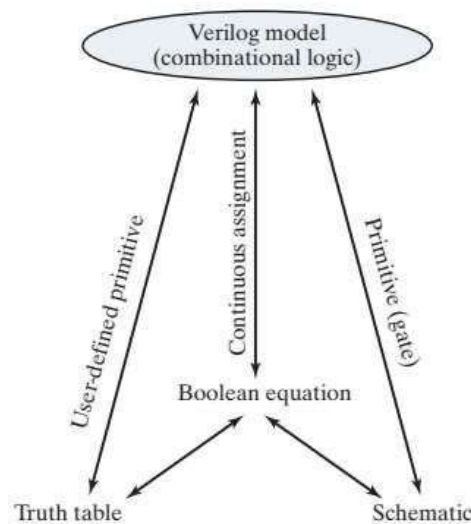
4-to-1-line MUX using three state gates and decoder

HDL MODELS OF COMBINATIONAL CIRCUITS

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

- 3 Gate-level modeling using instantiations of predefined and user-defined primitive gates.
- 4 Dataflow modeling using continuous assignment statements with the keyword assign.
- 5 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. 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 below. 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. We will not consider switch-level modeling.



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. Gate-level modeling provides a textual description of a schematic diagram. The Verilog HDL includes 12 basic gates as predefined primitives. Four of these primitive gates are of the three-state type. The other eight are the basic gates listed. They are all declared with the lowercase keywords **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, and **buf**. Primitives such as **and** are n-input primitives. They can have any number of scalar inputs (e.g., a three-input **and** primitive). The **buf** and **not** primitives are n-output primitives. A single input can drive multiple output lines distinguished by their identifiers. The Verilog language includes a functional description of each type of gate, too. The logic of each gate is based on a four-valued system. When the gates are simulated, the simulator assigns one value to the output of each gate at any instant. In addition to the two logic values of 0 and 1, there are two other values: unknown and high impedance. An unknown value is denoted by x and a high impedance by z. An unknown value is assigned during simulation when the logic value of a signal is ambiguous—for instance, if it cannot be determined whether its value is 0 or 1 (e.g., a flip-flop without a reset condition). A high-impedance condition occurs at the output of three-state gates that are not enabled or if a wire is inadvertently left

unconnected. The four-valued logic truth tables for the **and**, **or**, **xor**, and **not** primitives are shown in Table.

The truth table for the other four gates is the same, except that the outputs are complemented. Note that for the **and** gate, the output is 1 only when both inputs are 1 and the output is 0 if any input is 0. Otherwise, if one input is x or z, the output is x. The output of the **or** gate is 0 if both inputs are 0, is 1 if any input is 1, and is x otherwise.

Truth Table for Predefined Primitive Gates

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

xor	0	1	x	z	not	input	output
0	0	1	x	x		0	1
1	1	0	x	x		1	0
x	x	x	x	x		x	x
z	x	x	x	x		z	x

We now present two examples of gate-level modeling. Both examples use identifiers having multiple bit widths, called vectors. The syntax specifying a vector includes within square brackets two numbers separated with a colon. 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. (Note: The first (left-most) number (array index) listed is always the most significant bit of the vector). The individual bits are specified within square brackets, so D[2] specifies bit 2 of D. It is also possible to address parts (contiguous bits) of vectors. For example, SUM[2: 0] specifies the three least significant bits of vector SUM.

HDL Example below shows the gate-level description of a two-to-four-line decoder.

```

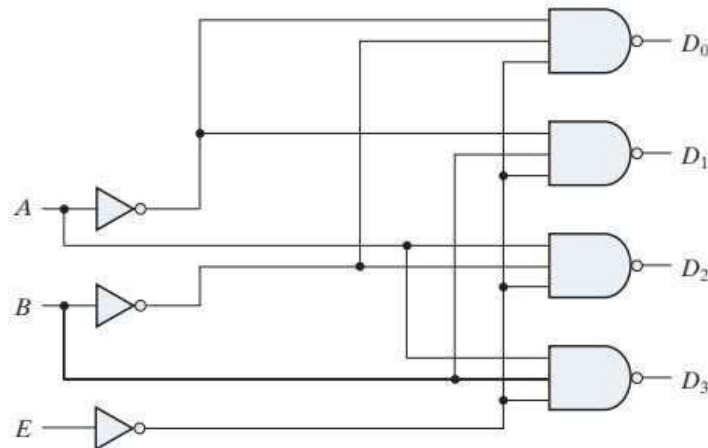
module decoder_2x4_gates (D, A, B, enable);
  output      [0: 3]      D;
  input       A, B;
  input       enable;
  wire        A_not,B_not, enable_not;

  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

```

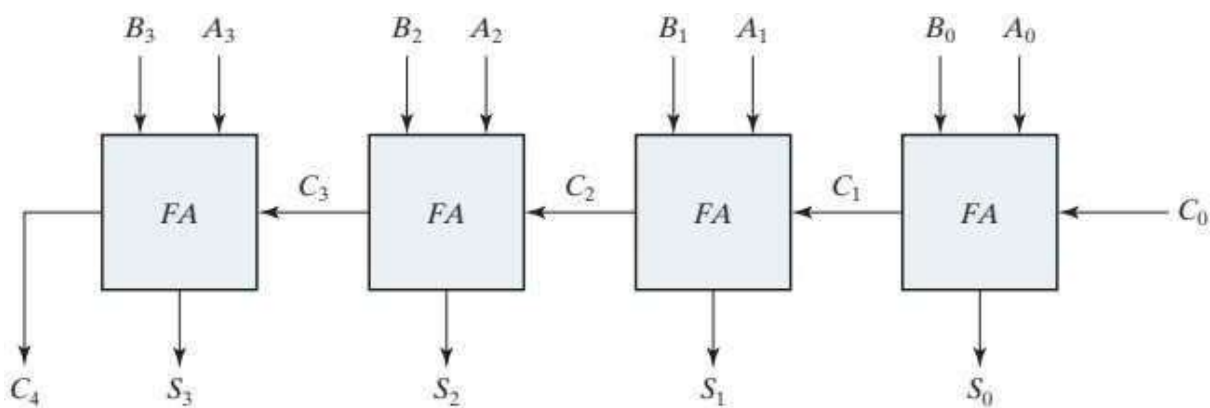
This decoder has two data inputs A and B and an enable input E. The four outputs are specified with the vector D. The **wire** declaration is for internal connections. Three not gates produce the complement of the inputs, and four **nand** gates provide the outputs for D. Remember that the output is always listed

first in the port list of a primitive, followed by the inputs. This example describes the decoder implementation shown in below Fig.



HDL Example below shows the gate-level description of a Four-bit ripple carry adder

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. Take, for example, the binary adder shown in below fig. It can be considered as a top-block component built with four full-adder blocks, while each full adder is built with two half-adder blocks. In a top-down design, the four-bit adder is defined first, and then the two adders are described. In a bottom-up design, the half adder is defined, then each full adder is constructed, and then the four-bit adder is built from the full adders.



Four-bit adder

A bottom-up hierarchical description of a four-bit adder is shown in HDL Example below. 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.

// Gate-level description of four-bit ripple carry adder

```

// 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

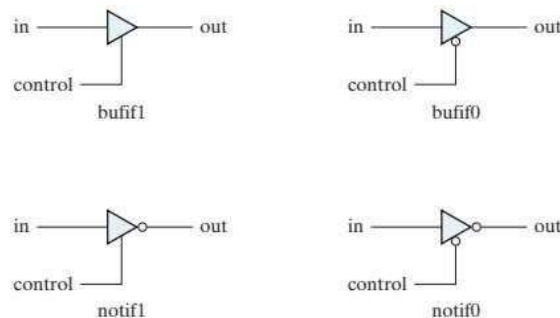
```

HDL Example above illustrates Verilog 2001, 2005 syntax, which eliminates extra typing of identifiers declaring the mode (e.g., output), type (reg), and declaration of a vector range (e.g., [3: 0]) of a port. The first version of the standard (1995) uses separate statements for these declarations.

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 below. 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 (output, input, control);



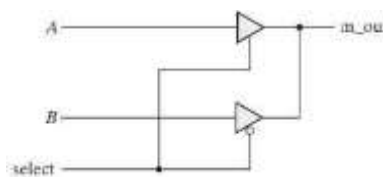
Three-state gates

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 buffers shown in Fig below.



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

The HDL description must use a **tri** data type for the 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
```

Dataflow Modeling

Dataflow modeling of combinational logic uses a number of operators that act on binary operands to produce a binary result. Dataflow modeling uses continuous assignments and the keyword **assign**. 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.

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

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

```
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
```

// 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
```

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 below 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 (in addition to the output declaration). The

procedural assignment statements inside the **always** block are executed every time there is a change in any of the variables listed after the **@** symbol. (Note that there is no semicolon (;) at the end of the

always statement.) 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.

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 = B;
endmodule
```

Behavioral description of four-to-one-line multiplexer

```
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
```

Synchronous Sequential Logic

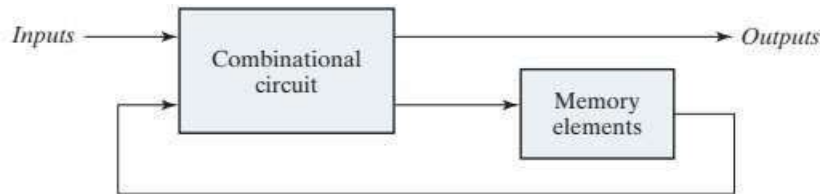
Hand-held devices, cell phones, navigation receivers, personal computers, digital cameras, personal media players, and virtually all electronic consumer products have the ability to send, receive, store, retrieve, and process information represented in a binary format. The technology enabling and supporting these devices is critically dependent on electronic components that can store information, i.e., have memory. The digital circuits considered thus far have been combinational—their output depends only and immediately on their inputs—they have no memory, i.e., dependence on past values of their inputs. Sequential circuits, however, act as storage elements and have memory. They can store, retain, and then retrieve information when needed later.

Sequential Circuits

A block diagram of a sequential circuit is shown in Fig below. 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.**

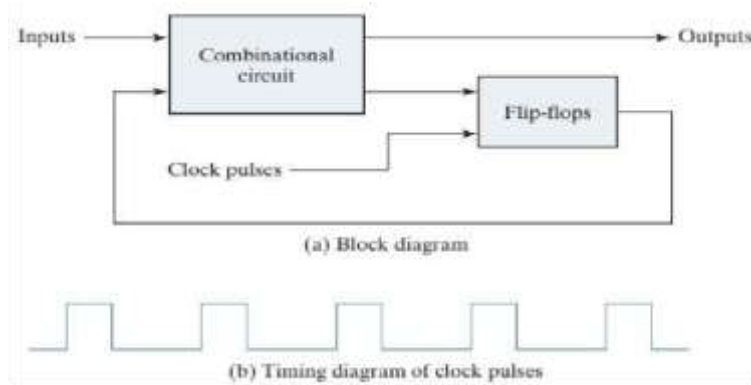


Block diagram of sequential circuit

There are two main types of sequential circuits, and their classification is a function of the timing of their signals. A **synchronous** sequential circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time. The behavior of an **asynchronous** sequential circuit depends upon the input signals at any instant of time and the order in which the inputs change. The storage elements commonly used in asynchronous sequential circuits are time-delay devices. The storage capability of a time-delay device varies with the time it takes for the signal to propagate through the device.

A synchronous sequential circuit employs signals that affect the storage elements at only discrete instants of time. Synchronization is achieved by a timing device called a **clock generator**, which provides a clock signal having the form of a periodic train of clock pulses. The clock signal is commonly denoted by the identifiers **clock** and **clk**. The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of each pulse. Synchronous sequential circuits that use *clock pulses* to control storage elements are called *clocked sequential circuits* and are the type most frequently encountered in practice. They are called *synchronous circuits* because the activity within the circuit and the resulting updating of stored values is synchronized to the occurrence of clock pulses.

The storage elements (memory) used in clocked sequential circuits are called *flip-flops*. A *flip-flop* is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either 0 or 1. A sequential circuit may use many flip-flops to store as many bits as necessary. The block diagram of a synchronous clocked sequential circuit is shown in Fig(a) below. The outputs are formed by a combinational logic function of the inputs to the circuit or the values stored in the flip-flops (or both). The value that is stored in a flip-flop when the clock pulse occurs is also determined by the inputs to the circuit or the values presently stored in the flip-flop (or both). If the clock (synchronizing) pulses arrive at a regular interval, as shown in the timing diagram in Fig(b), the combinational logic must respond to a change in the state of the flip-flop in time to be updated before the next pulse arrives.



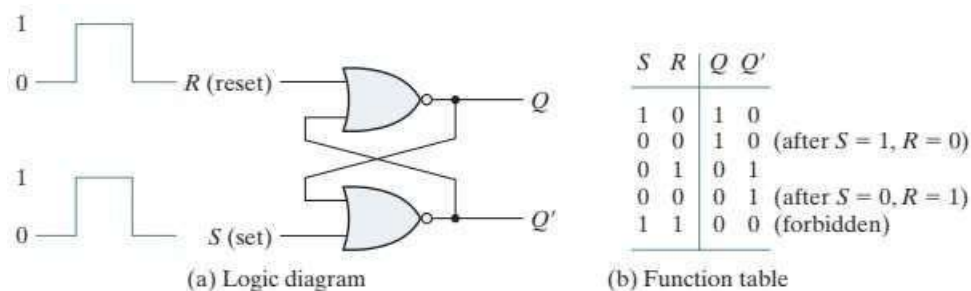
A change in state of the flip-flops is initiated only by a clock pulse transition—for example, when the value of the clock signals changes from 0 to 1. Thus, the transition from one state to the next occurs only at predetermined intervals dictated by the clock pulses.

Storage Elements: Latches

A storage element in a digital circuit can maintain a binary state indefinitely (as long as power is delivered to the circuit), 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 (rather than signal transitions) 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.

SR Latch

The SR latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled S for set and R for reset. The SR latch constructed with two cross-coupled NOR gates is shown in Fig(a). 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.



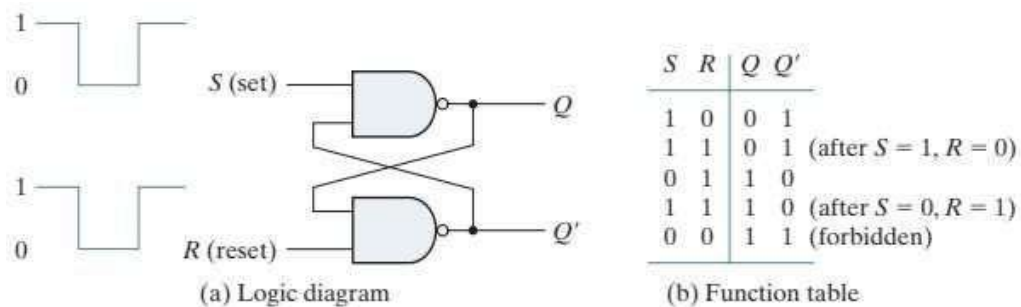
SR latch with NOR gates

As shown in the function table of above Fig(b), two input conditions cause the circuit to be in the set state. 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. Removing 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 momentarily applying a 1 to the R input. The 1 can then be removed from R, whereupon 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 are not applied to both inputs simultaneously.

The SR latch with two cross-coupled NAND gates is shown in below Fig. 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

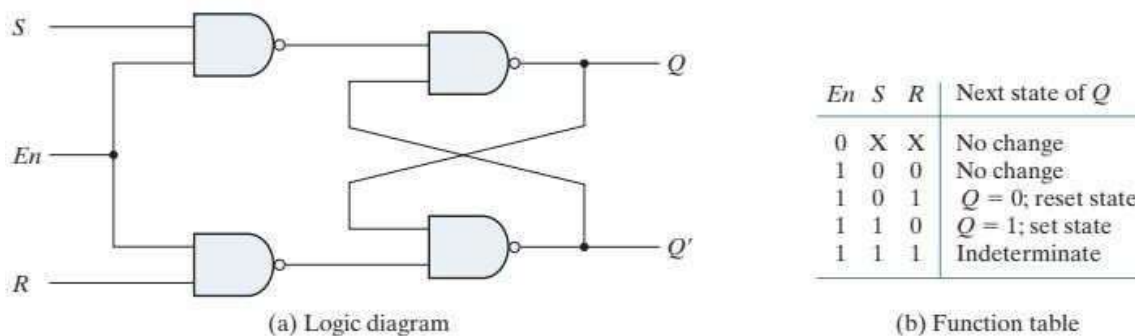
in the set state. After both inputs go back to 1, we are allowed to change the state of the latch by placing a 0 in the R input. This action causes the circuit to go to the reset state and stay there even after both inputs return to 1. The condition that is forbidden for the NAND latch is both inputs being equal to 0 at the same time, an input combination that should be avoided.



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 below Fig. 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 (active-high enabled).

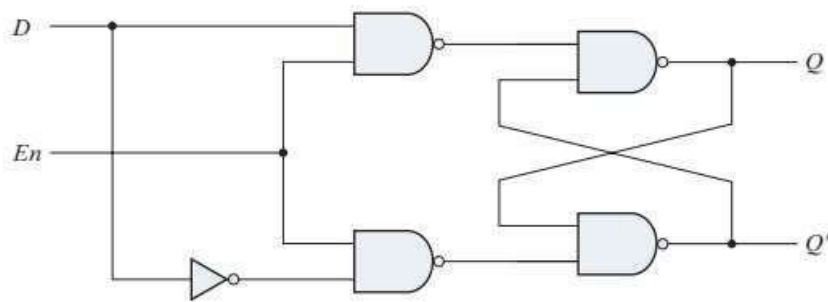


SR latch with control input

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.

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 below Fig. This latch has only two inputs: D (data) and En (enable).



(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

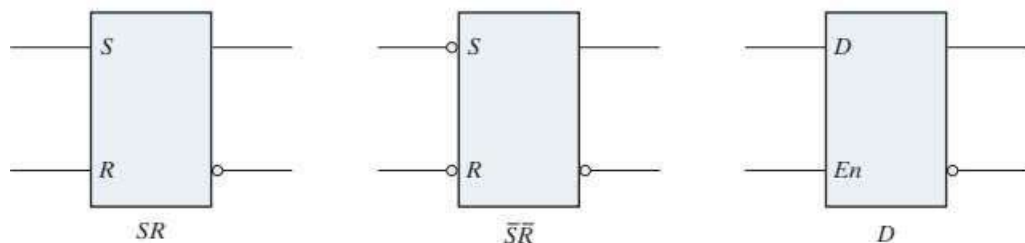
(b) Function table

D latch

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.

The binary information present at the data input of the D latch is transferred to the Q output when the enable input is asserted. The output follows changes in the data input as long as the enable input is asserted. This situation provides a path from input D to the output, and for this reason, the circuit is often called a transparent latch.

The graphic symbols for the various latches are shown in Fig. 5.7. 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. In the case of a NAND gate latch, bubbles are added to the inputs to indicate that setting and resetting occur with a logic-0 signal. The graphic symbol for the D latch has inputs D and En indicated inside the block.

Graphic symbols for latches

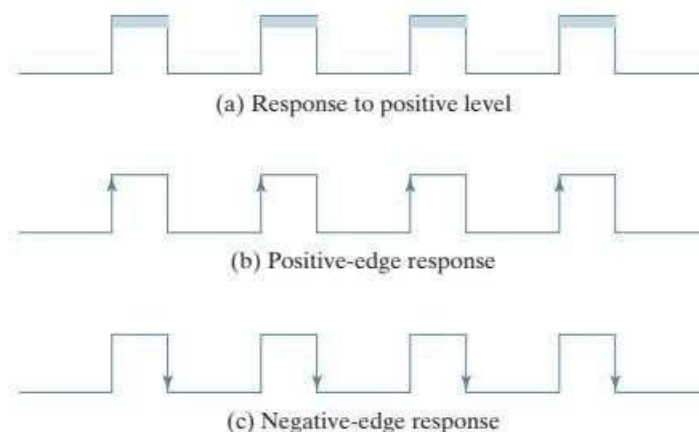
Storage Elements: Flip-Flops

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 we know, a sequential circuit has a feedback path from the outputs of the flip-flops to the input of the combinational circuit. 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. Because of this unreliable operation, the output of a latch cannot be applied directly or through combinational logic to the input of the same or another latch when all the latches are triggered by a common clock source.

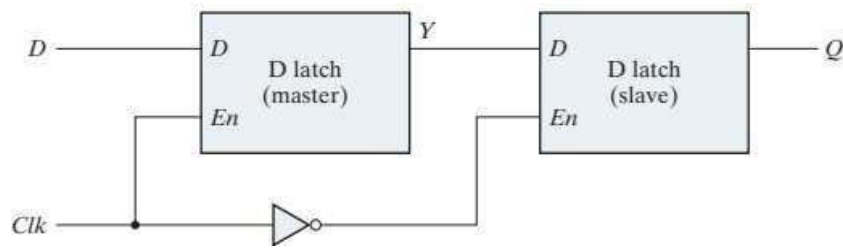
Flip-flop circuits are constructed in such a way as to make them operate properly when they are part of a sequential circuit that employs a common clock. As shown in below Fig(a), a positive level response in the enable input allows changes in the output when the D input changes while the clock pulse stays at logic 1. The key to the proper operation of a flip-flop is to trigger it only during a signal transition. This can be accomplished by eliminating the feedback path that is inherent in the operation of the sequential circuit using latches. A clock pulse goes through two transitions: from 0 to 1 and the return from 1 to 0. As shown in below Fig, the positive transition is defined as the positive edge and the negative transition as the negative edge. There are two ways that a latch can be modified to form a flip-flop. One way is to employ two latches in a special configuration that isolates the output of the flip-flop and prevents it from being affected while the input to the flip-flop is changing. Another way is to produce a flip-flop that triggers only during a signal transition (from 0 to 1 or from 1 to 0) of the synchronizing signal (clock) and is disabled during the rest of the clock pulse.



Clock response in latch and flip-flop

Edge-Triggered D Flip-Flop

The construction of a D flip-flop with two D latches and an inverter is shown in Fig below. The first latch is called the master and the second the slave. The circuit samples the D input and changes its output Q only at the negative edge of the synchronizing or controlling clock (designated as Clk). When the clock is 0, the output of the inverter is 1. The slave latch is enabled, and its output Q is equal to the master output Y. The master latch is disabled because Clk = 0. When the input pulse changes to the logic-1 level, the data from the external D input are transferred to the master. The slave, however, is disabled as long as the clock remains at the 1 level, because its enable input is equal to 0. Any change in the input changes the master output at Y, but cannot affect the slave output. When the clock pulse returns to 0, the master is disabled and is isolated from the D input. At the same time, the slave is enabled and the value of Y is transferred to the output of the flip-flop at Q. Thus, a change in the output of the flip-flop can be triggered only by and during the transition of the clock from 1 to 0.

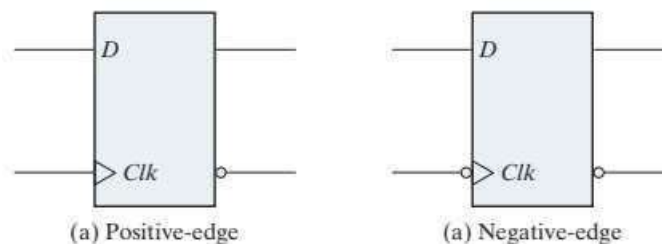


Master-slave D flip-flop

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.

Another construction of an edge-triggered D flip-flop uses three SR latches as shown in below Fig. 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.

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 graphic symbol for the edge-triggered D flip-flop is shown in above Fig. 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. The absence of a bubble designates a positive-edge response.

Other Flip-Flops

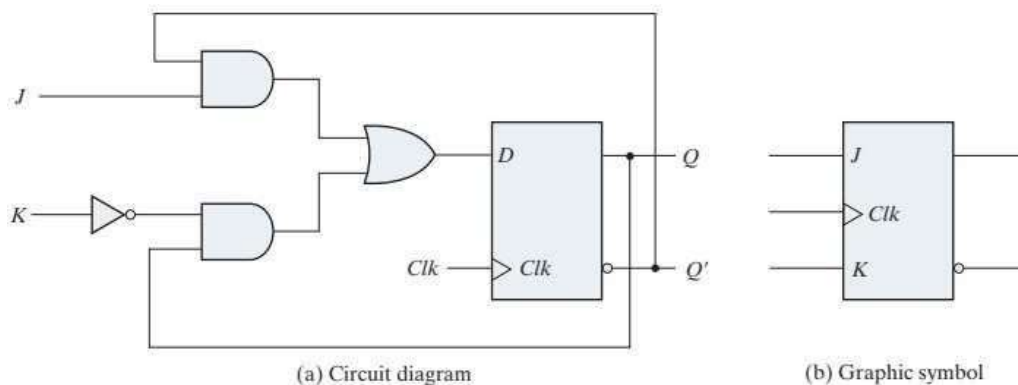
The most economical and efficient flip-flop constructed in this manner is the edge-triggered D flip-flop, because it requires the smallest number of gates. Other types of flip-flops can be constructed by using the D flip-flop and external logic. Two flip-flops less widely used in the design of digital 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(a) below. 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:

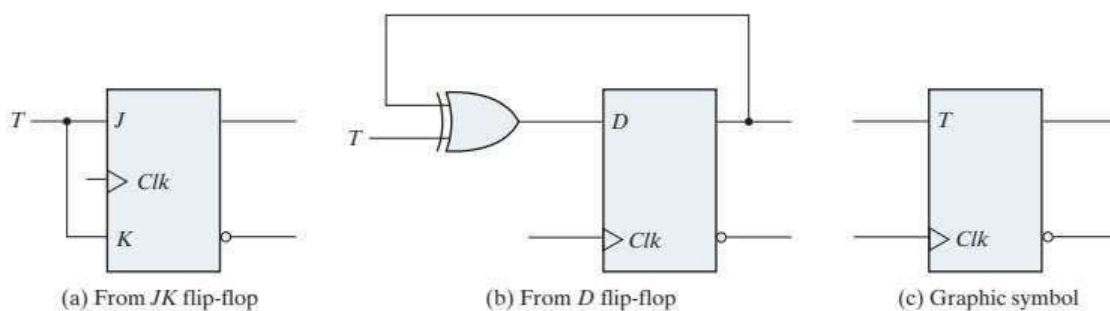
$$D = JQ' + K'Q$$

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(b).



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 below in Fig(a). 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.



T flip-flop

The T flip-flop can be constructed with a D flip-flop and an exclusive-OR gate as shown in above Fig(b). The expression for the D input is

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

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.

1. CHARACTERISTIC EQUATIONS: [TB2, Pg. NO - 329]

The algebraic description of the next-state table of a flip-flop is called the characteristic equation of the flip flop.

Characteristic equation of SR flip flop.

Simplified Function Table.

S	R	Q^+
0	0	Q
0	1	0
1	0	1
1	1	-

S	RQ			
	00	01	11	10
0	0	1	0	0
1	1	1	-	-

$$Q^+ = S + \bar{R}Q$$

Next State Function Table.

S	R	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	-
1	1	1	-

Characteristic Equation of D flip-flop.

Simplified Function Table Next State Function Table.

D	Q
0	0
1	1

D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

D	Q
0	1
0	0
1	0

characterSimplified

T
0
1

T \ Q	0	1
0	0	0
1	1	1

characterSimplified

J
0
0
1
1

J \ K	0	1
0	0	0
1	1	1

$$Q^+ = J$$

Note: In

explicitly

Module 2

Combinational Logic

Introduction

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.

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. 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.



Block Diagram of Combinational Circuit

For n input variables, there are 2^n possible combinations of the binary inputs. For each possible input combination, there is one possible value for each output variable. Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables. A combinational circuit also can be described by m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

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.

Examples of Combinational Logic Circuits: Adders, Subtractors, Decoders, Encoders, Multipliers, Dividers, Display Drivers, Keyboard Encoders, Etc...

Design Procedure

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

1. From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each.
2. Derive the truth table that defines the required relationship between inputs and outputs.
3. Obtain the simplified Boolean functions for each output as a function of the input variables.

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

A truth table for a combinational circuit consists of input columns and output columns. The input columns are obtained from the 2^n binary numbers for the n input variables. The binary values for the outputs are determined from the stated specifications. The output functions specified in the truth table

give the exact definition of the combinational circuit. It is important that the verbal specifications be interpreted correctly in the truth table, as they are often incomplete, and any wrong interpretation may result in an incorrect truth table.

The output binary functions listed in the truth table are simplified by any available method, such as algebraic manipulation, the map method, or a computer-based simplification program. Frequently, there is a variety of simplified expressions from which to choose. In a particular application, certain criteria will serve as a guide in the process of choosing an implementation. A practical design must consider such constraints as the number of gates, number of inputs to a gate, propagation time of the signal through the gates, number of interconnections, limitations of the driving capability of each gate (i.e., the number of gates to which the output of the circuit may be connected), and various other criteria that must be taken into consideration when designing integrated circuits.

Example: Consider the BCD to Excess-3 code conversion.

The bit combinations assigned to the BCD and excess-3 codes are listed in Table below

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 variables is shown in Table above. Note that four binary variables may have 16 bit combinations, but only 10 are listed in the truth table. The six bit combinations not listed for the input variables are don't-care combinations. These values have no meaning in BCD and we assume that they will never occur in actual operation of the circuit.

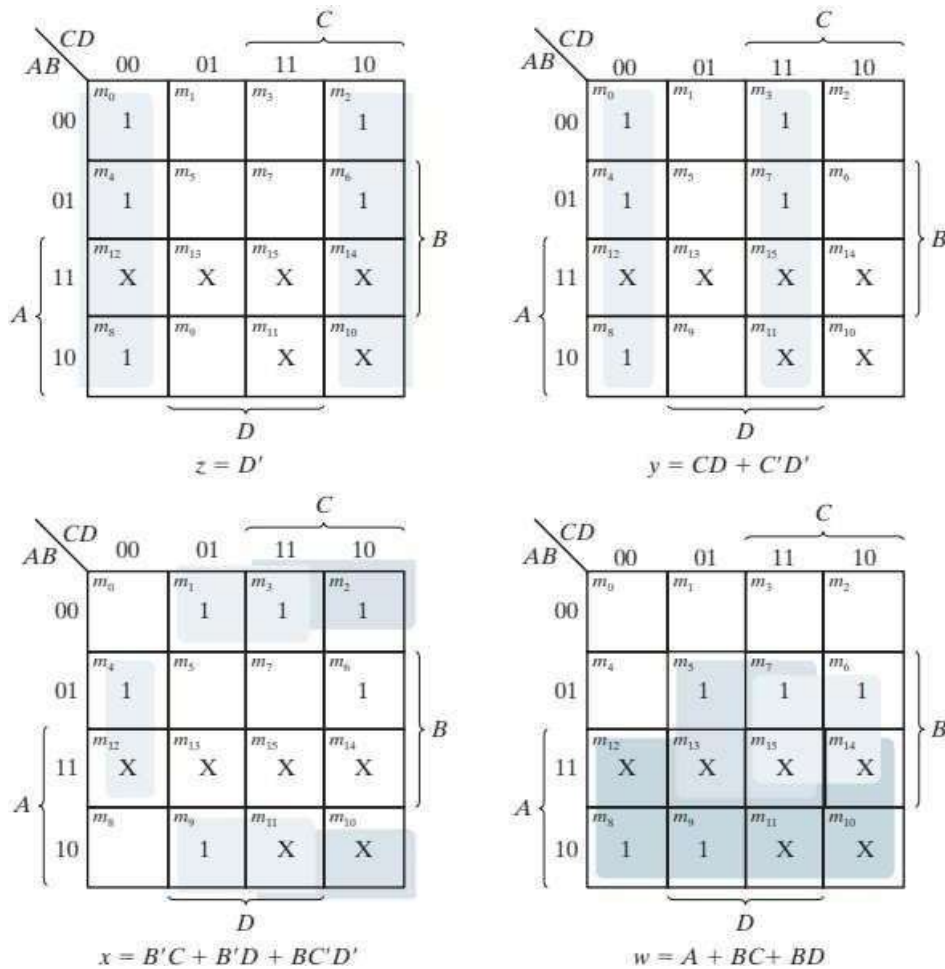
The maps in Fig. 4.3 are plotted to obtain simplified Boolean functions for the outputs. Each one of the four maps represents one of the four outputs of the circuit as a function of the four input variables. The 1's marked inside the squares are obtained from the minterms that make the output equal to 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. Following are the expressions obtained using K map.

$$z = D'$$

$$y = CD + C'D' = CD + (C + D)'$$

$$\begin{aligned} x &= B'C + B'D + BC'D' = B'(C + D) + BC'D' \\ &= B'(C + D) + B(C + D)' \end{aligned}$$

$$w = A + BC + BD = A + B(C + D)$$



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

Binary 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:

$$0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, \text{ and } 1 + 1 = 10$$

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.

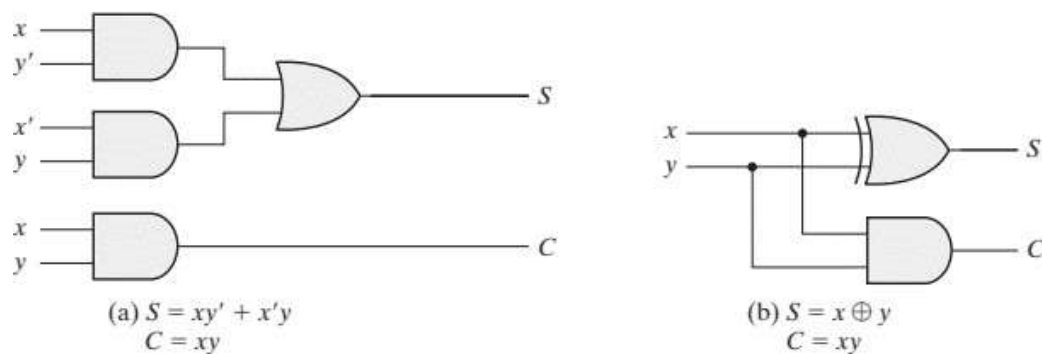
Half Adder

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 below. The C output is 1 only when both inputs are 1. The S output represents the least significant bit of the sum.

Half Adder

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



The logic diagram of the half adder implemented in sum of products is shown in Fig.(a). It can be also implemented with an exclusive-OR and an AND gate as shown in Fig.(b)

Full Adder

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. s . 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 below. 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.

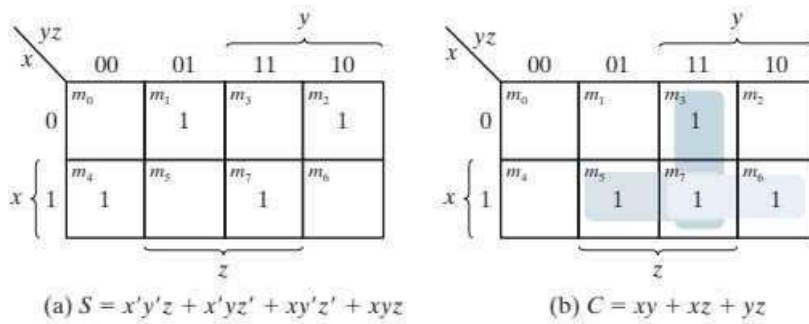
Full Adder

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

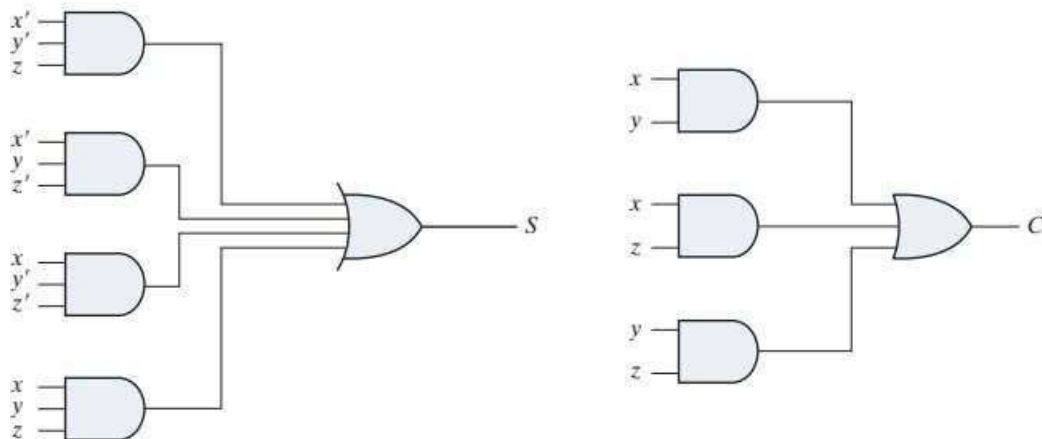
The maps for the outputs of the full adder are shown in Fig below. The simplified expressions are

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

$$C = xy + xz + yz$$



The logic diagram for the full adder implemented in sum-of-products form is shown in Fig below.



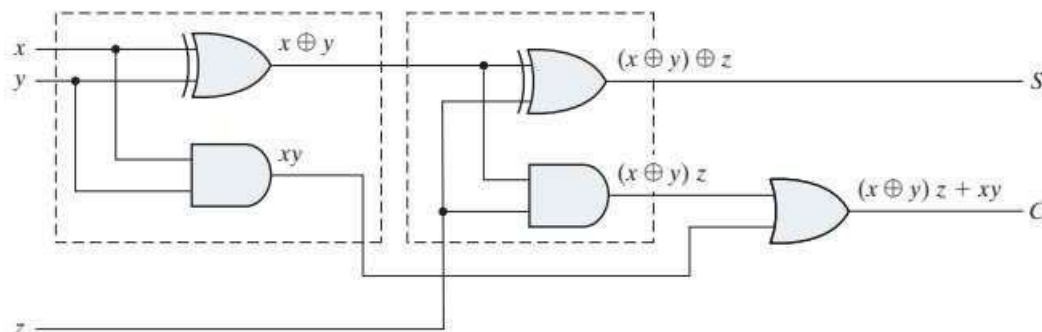
Implementation of full adder in sum-of-products form

It can also be implemented with two half adders and one OR gate, as shown in Fig below. 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

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



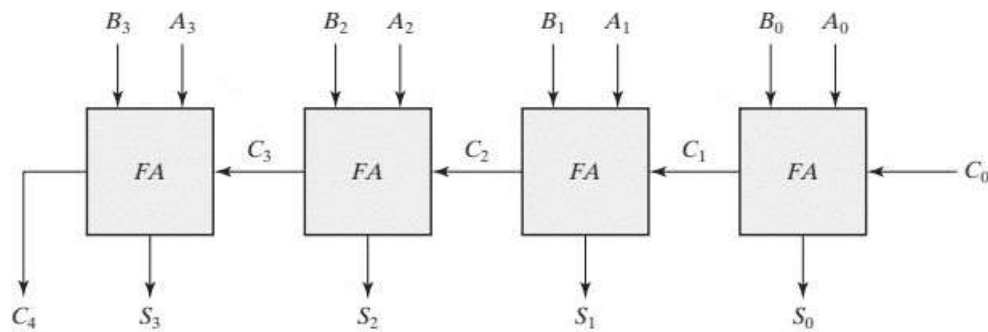
Implementation of full adder with two half adders and an OR gate

Binary Adder

A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be

constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain. Addition of n-bit numbers requires a chain of n full

adders or a chain of one-half adder and $n-1$ full adders. Figure below shows the interconnection of four full-adder (FA) circuits to provide a four-bit binary ripple carry adder.



Four-bit 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. All the carries must be generated for the correct sum bits to appear at the outputs.

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 by the classical method would require a truth table with $2^9 = 512$ entries, since there are nine inputs to the circuit. By using an iterative method of cascading a standard function, it is possible to obtain a simple and straightforward implementation.

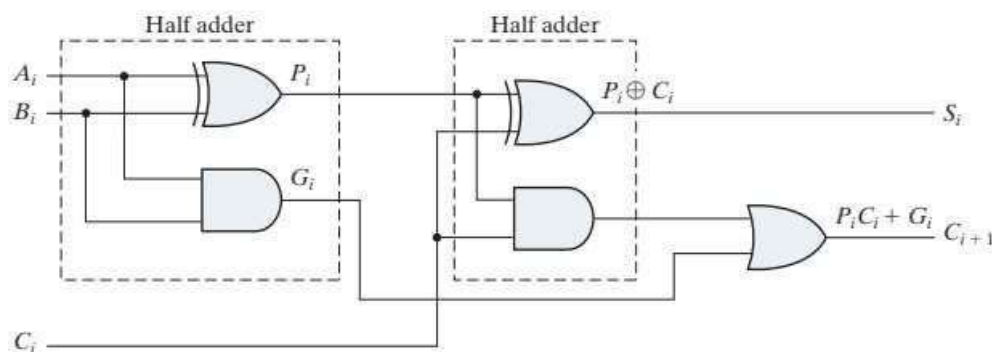
Carry Propagation

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 S_3 in Four-Bit Adder (above Fig). Inputs A_3 and B_3 are available as soon as input signals are applied to the adder. However, input carry C_3 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 below for convenience. The input and output variables use the subscript i to denote a typical stage of the adder. The signals at P_i and G_i settle to their steady-state values after they propagate through their respective gates. These two signals are common to all half adders and depend on only the input augend and addend bits. The signal from the input carry C_i to the output carry C_{i+1} propagates through an AND gate and an OR gate, which constitute two gate levels. If there are four full adders in the adder, the output carry C_4 would have $2 * 4 = 8$ gate levels from C_0 to C_4 . For an n -bit adder, there are $2n$ gate levels for the carry to propagate from input to output.



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. 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. Solution is to reduce the propagation time. 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 above Fig. If we define two new binary variables

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

the output sum and carry can respectively be expressed as

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

G_i 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 $i + 1$ (assertion of C_i will propagate to an assertion of C_{i+1}).

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

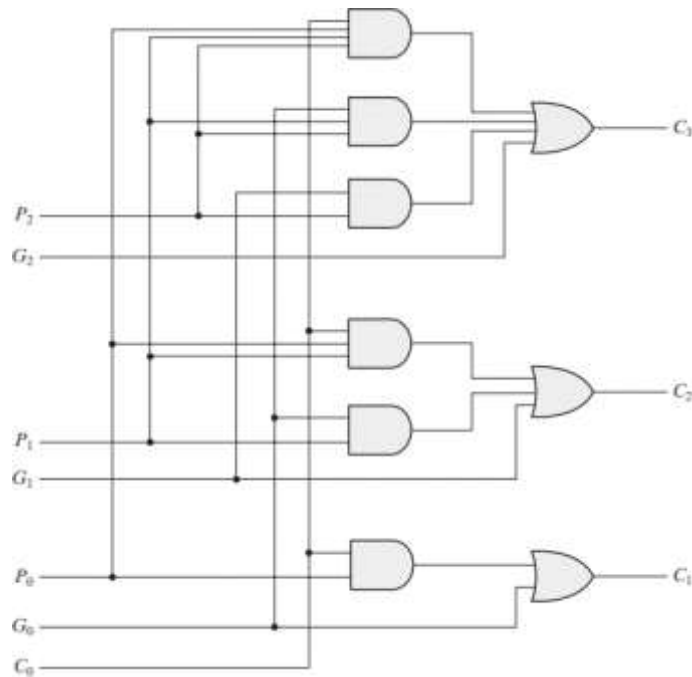
$$C_0 = \text{input carry}$$

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$

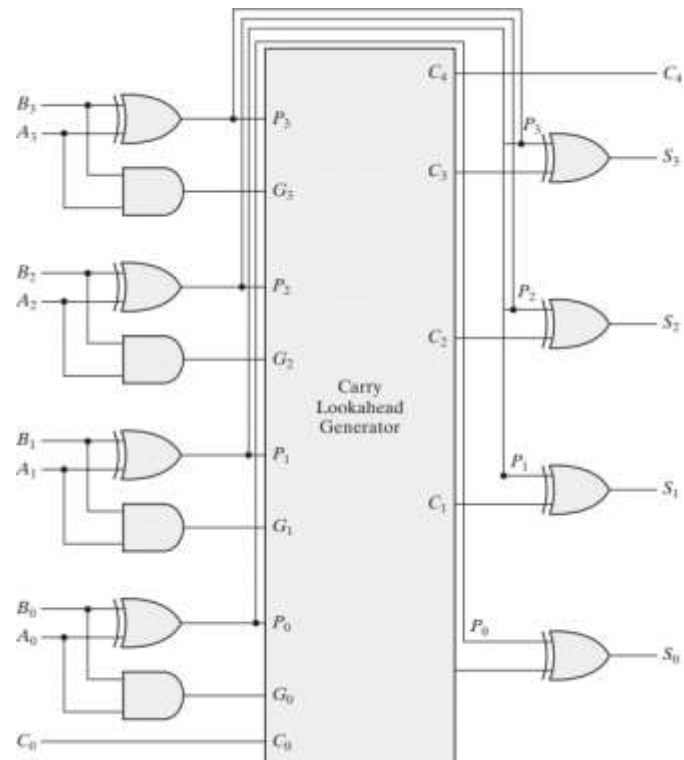
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 = P_2 P_1 P_0 C_0$$

The three Boolean functions for C_1 , C_2 , and C_3 are implemented in the carry lookahead generator shown in Fig below.



Logic diagram of carry lookahead generator

The construction of a four-bit adder with a carry lookahead scheme is shown in Fig below. Each sum output requires two exclusive-OR gates. The output of the first exclusive-OR gate generates the P_i variable, and the AND gate generates the G_i variable. The carries are propagated through the carry lookahead generator. Thus, outputs S_1 through S_3 have equal propagation delay times

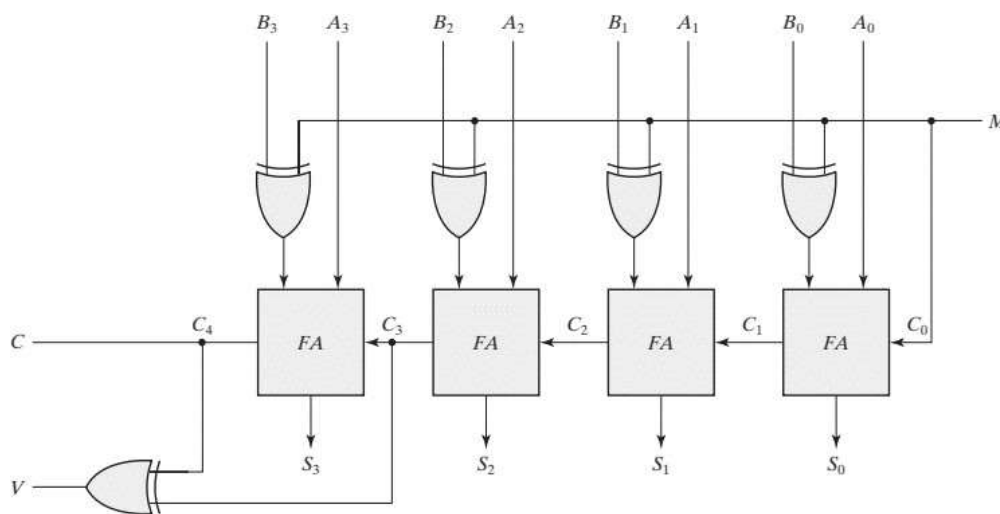


Four-bit adder with carry lookahead

Binary Subtractor

The subtraction of unsigned binary numbers can be done most conveniently by means of complements. Remember that the subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A . The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits. The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry.

The circuit for subtracting $A - B$ consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C_0 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 .



Four-bit adder-subtractor (with overflow detection)

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 above Fig. 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 $B \oplus 0 = B$. 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 $B \oplus 1 = \bar{B}$ 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 . (The exclusive-OR with output V is for detecting an overflow.)

Overflow

When two numbers with n digits each are added and the sum is a number occupying $n + 1$ digits, we say that an overflow occurred. This is true for binary or decimal numbers, signed or unsigned. When the addition is performed with paper and pencil, an overflow is not a problem, since there is no limit by the width of the page to write down the sum. Overflow is a problem in digital computers because the number of bits that hold the number is finite and a result that contains $n + 1$ bits cannot be accommodated by an n -bit word. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set that can then be checked by the user.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected

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

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

Consider the following example: Two signed binary numbers, +70 and +80, are stored in two eight-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary -128. Since the sum of the two numbers is +150, it exceeds the capacity of an eight-bit register. This is also true for -70 and -80.

carries:	0	1	carries:	1	0
+70	0	1000110	-70	1	0111010
+80	0	1010000	-80	1	0110000
<u>+150</u>	1	0010110	<u>-150</u>	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.

Decoders

A decoder is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. If the n -bit coded information has unused combinations, the decoder may have fewer than 2^n outputs. The decoders presented here are called n -to- m -line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) minterms of n input variables. Each combination of inputs will assert a unique output. The name decoder is also used in conjunction with other code converters, such as a BCD-to-seven-segment decoder.

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

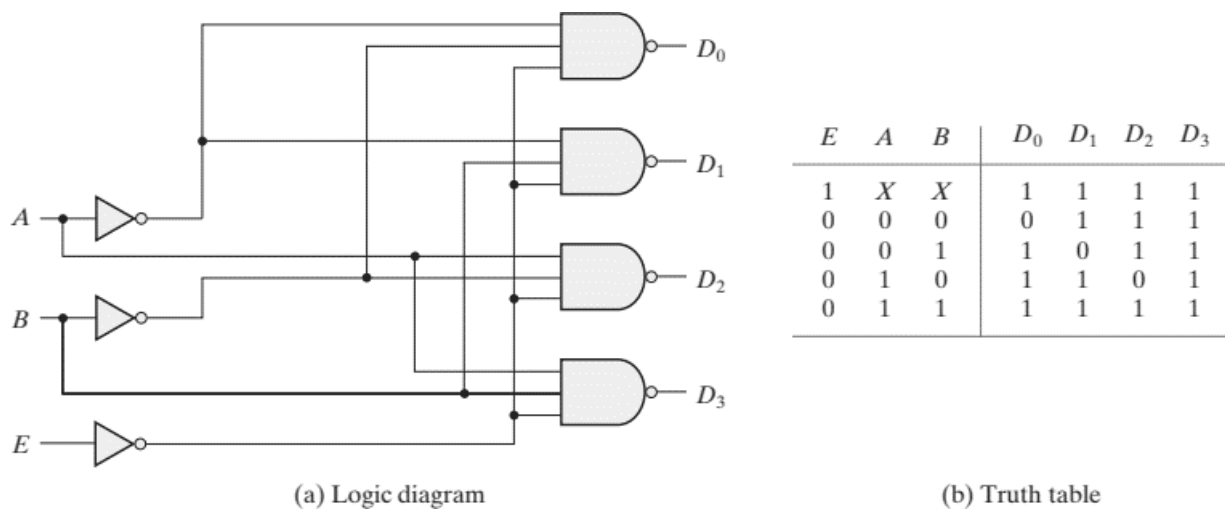
Truth Table of a Three-to-Eight-Line Decoder

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 above. For each possible

input combination, there are seven outputs that are equal to 0 and only one that is equal to 1. The output whose value is equal to 1 represents the minterm equivalent of the binary number currently available in the input lines.

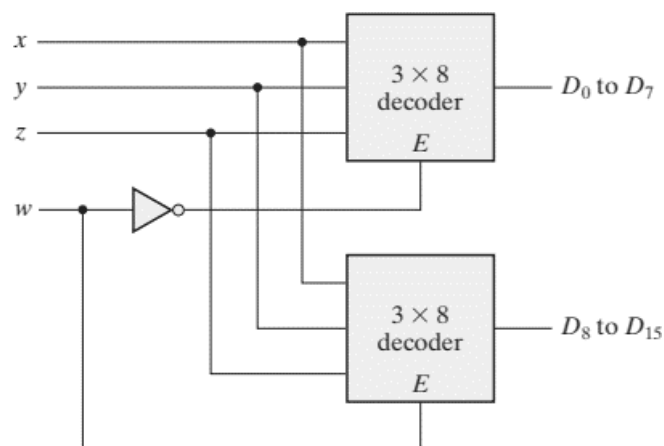
Some decoders are constructed with NAND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder minterms in their complemented form. Furthermore, decoders include one or more enable inputs to control the circuit operation. A two-to-four-line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



Two-to-four-line decoder with enable input

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.

Decoders with enable inputs can be connected together to form a larger decoder circuit. Figure below 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.



4 X 16decoder constructed with two 3 X 8decoders

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.

Combinational Logic Implementation using Decoders

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. This procedure will be illustrated by an example that implements a full-adder circuit.

Consider the truth table of Full adder-

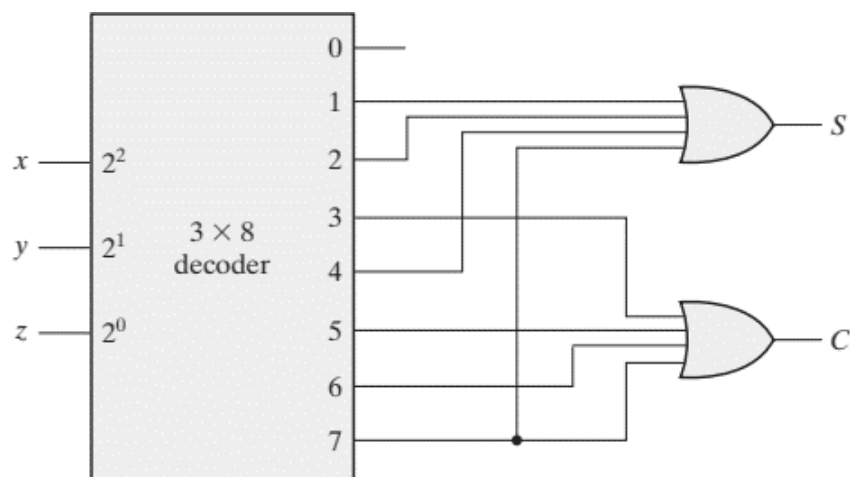
Full Adder				
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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 below. 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.



Implementation of a full adder with a decoder

Exercise:

Using a decoder and external gates, design the combinational circuit defined by the following three Boolean functions:

(a) $F_1 = x'yz' + xz$

$$F_2 = xy'z' + x'y$$

$$F_3 = x'y'z' + xy$$

(b) $F_1 = (y' + x)z$

$$F_2 = y'z' + x'y + yz'$$

$$F_3 = (x + y)z$$

Encoders

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has $2n$ (or fewer) input lines and n output lines. The output lines, as an aggregate, generate the binary code corresponding to the input value. An example of an encoder is the octal-to-binary encoder whose truth table is given in Table

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

$$z = D_1 + D_3 + D_5 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$x = D_4 + D_5 + D_6 + D_7$$

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 defined in Table above 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 D_6 has higher priority than D_3 .

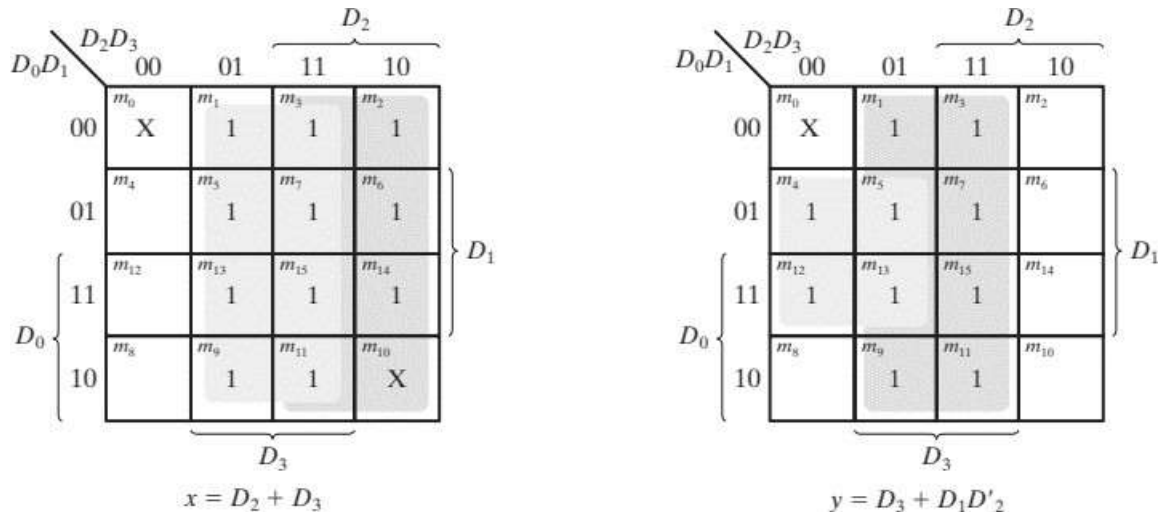
Priority Encoder

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. 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, $X100$ represents the two minterms 0100 and 1100.

Truth Table of a Priority Encoder

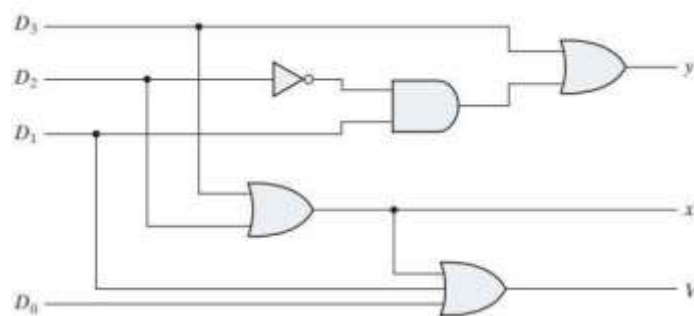
Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

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



The maps for simplifying outputs x and y are shown above. 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

$$\begin{aligned} x &= D_2 + D_3 \\ y &= D_3 + D_1 D'_2 \\ V &= D_0 + D_1 + D_2 + D_3 \end{aligned}$$



Four-input priority encoder

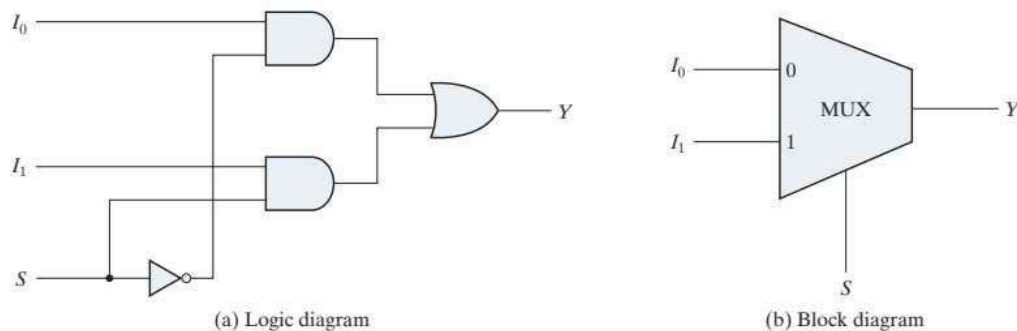
Multiplexers

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.

Two-to-One-line multiplexer

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

enabled and I_1 has a path to the output. The multiplexer acts like an electronic switch that selects one of two sources. The block diagram of a multiplexer is sometimes depicted by a wedge-shaped symbol, as shown in Fig. The multiplexer is often labeled “MUX” in block diagrams.

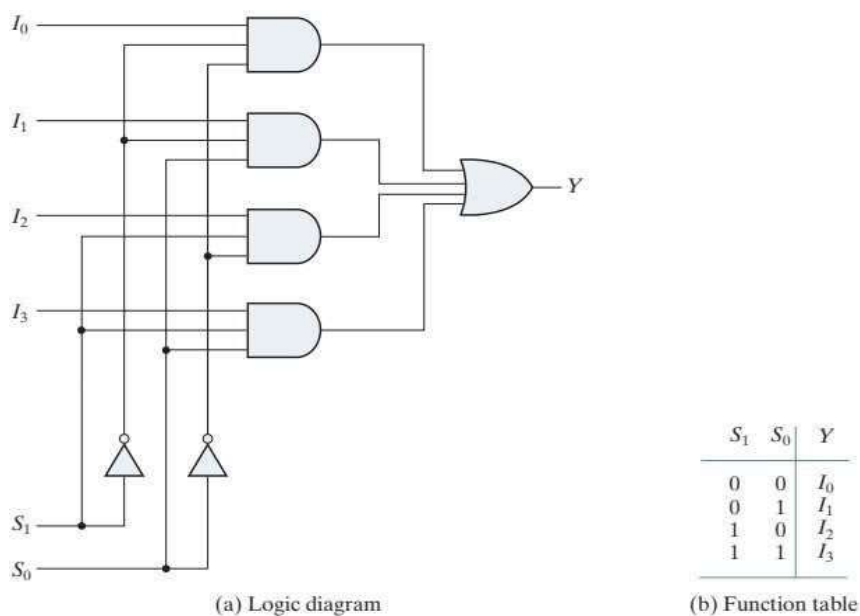


Two-to-One-line multiplexer

Four-to-One-line multiplexer

A four-to-one-line multiplexer is shown in Fig below. Each of the four inputs, I_0 through I_3 , 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.

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.



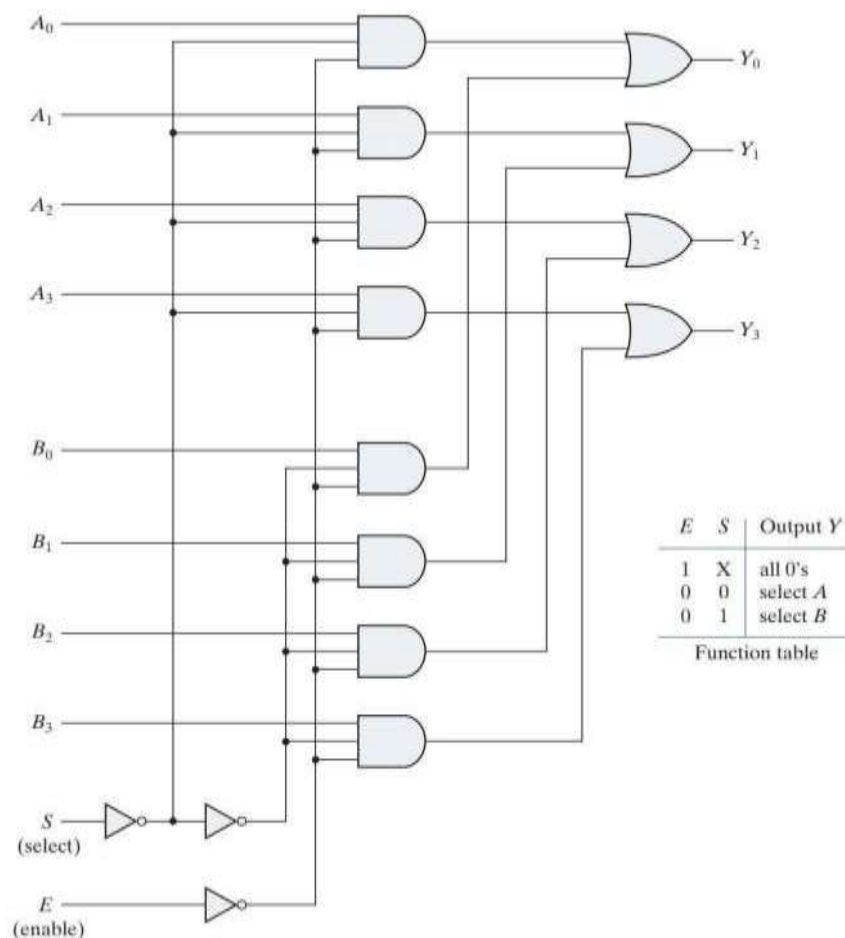
Four-to-One-line Multiplexer

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 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.

Quadruple two-to-one-line multiplexer

Multiplexer circuits can be combined with common selection inputs to provide multiple-bit selection logic. As an illustration, a quadruple 2-to-1-line multiplexer is shown in Fig below. 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 .



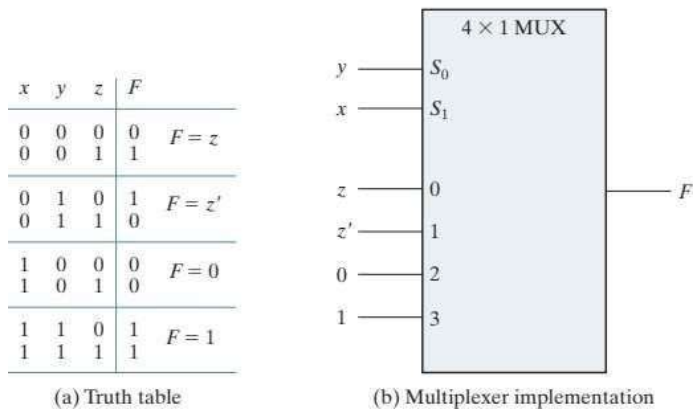
Quadruple two-to-one-line multiplexer

Boolean Function Implementation using MUX

1. Consider the Boolean function $F(x, y, z) = \Sigma(1, 2, 6, 7)$.

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

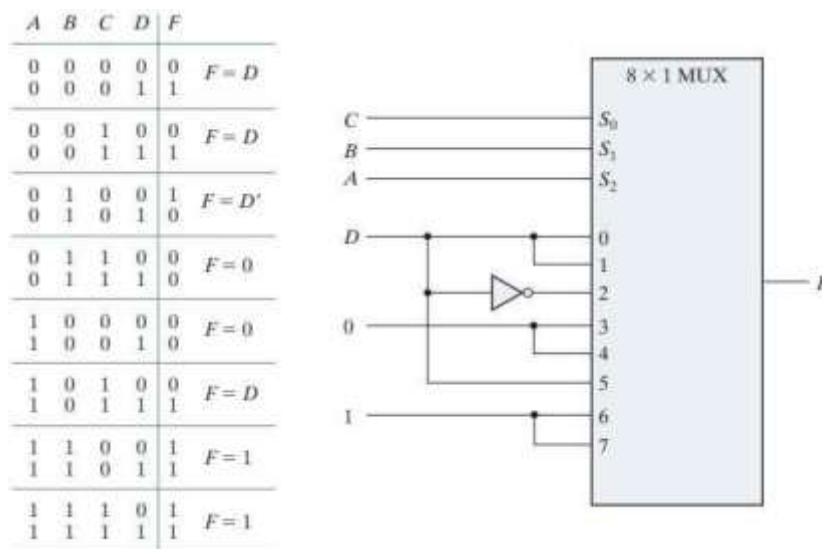
= 00, data input 0 has a path to the output, and that makes F equal to z . In a similar fashion, we can determine the required input to data lines 1, 2, and 3 from the value of F when $xy = 01$, 10, and 11, respectively. This particular example shows all four possibilities that can be obtained for the data inputs.



Implementing a Boolean function with a multiplexer

2. Consider the implementation of the Boolean function $F(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$

This function is implemented with a multiplexer with three selection inputs as shown in Fig below. Note that the first variable A must be connected to selection input S_2 so that A, B, and C correspond to selection inputs S_2 , S_1 , and S_0 respectively. The values for the data inputs are determined from the truth table listed in the figure. The corresponding data line number is determined from the binary combination of ABC. For example, the table shows that when $ABC = 101$, $F = D$, so the input variable D is applied to data input 5.



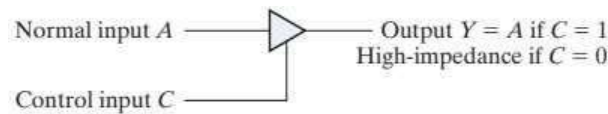
Implementing a four-input function with a multiplexer

Three-State Gates

A multiplexer can be constructed with three-state gates—digital circuits that exhibit three states. Two of the states are signals equivalent to logic 1 and logic 0 as in a conventional gate. The third state is a high-impedance state in which (1) the logic behaves like an open circuit, which means that the output appears to be disconnected, (2) the circuit has no logic significance, and (3) the circuit connected to the output of the three-state gate is not affected by the inputs to the gate. Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used is the buffer gate. The graphic symbol for a three-state buffer gate is shown in Fig below. 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

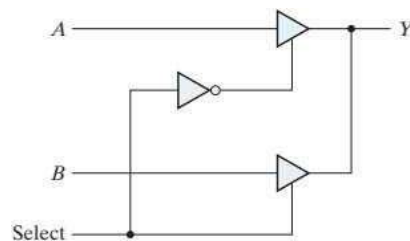
input that determines the state of the output. When the control input is equal to 1, the output is enabled and the gate behaves like a conventional buffer, with the output equal to the normal

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.



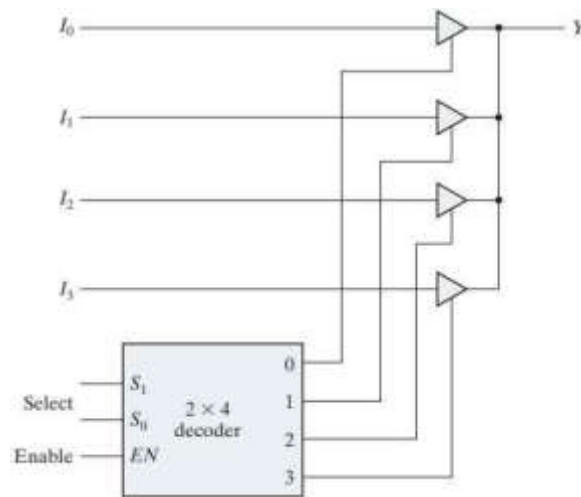
Graphic symbol for a three-state buffer

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



2-to-1-line mux using three state gates

The construction of a four-to-one-line multiplexer is shown in Fig below. 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 I_0 through 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 threestate buffers will be active, depending on the binary value in the select inputs of the decoder.



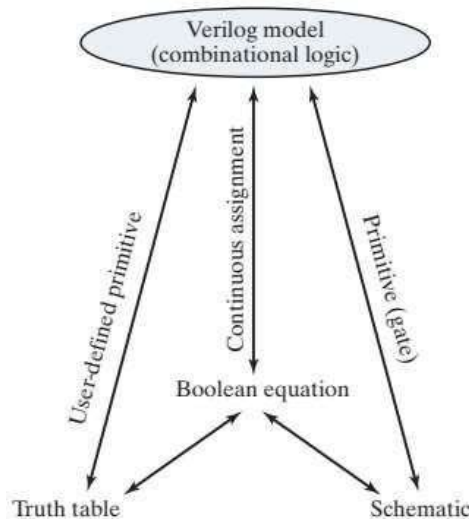
4-to-1-line MUX using three state gates and decoder

HDL MODELS OF COMBINATIONAL CIRCUITS

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

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

Gate-level (structural) modeling describes a circuit by specifying its gates and how they are connected with each other. Dataflow modeling is used mostly for describing the Boolean equations of combinational logic. 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 below. 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. We will not consider switch-level modeling.



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. Gate-level modeling provides a textual description of a schematic diagram. The Verilog HDL includes 12 basic gates as predefined primitives. Four of these primitive gates are of the three-state type. The other eight are the basic gates listed. They are all declared with the lowercase keywords **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, and **buf**. Primitives such as **and** are n-input primitives. They can have any number of scalar inputs (e.g., a three-input **and** primitive). The **buf** and **not** primitives are n-output primitives. A single input can drive multiple output lines distinguished by their identifiers. The Verilog language includes a functional description of each type of gate, too. The logic of each gate is based on a four-valued system. When the gates are simulated, the simulator assigns one value to the output of each gate at any instant. In addition to the two logic values of 0 and 1, there are two other values: unknown and high impedance. An unknown value is denoted by x and a high impedance by z. An unknown value is assigned during simulation when the logic value of a signal is ambiguous—for instance, if it cannot be determined whether its value is 0 or

1 (e.g., a flip-flop without a reset condition). A high-impedance condition occurs at the output of three-state gates that are not enabled or if a wire is inadvertently left

unconnected. The four-valued logic truth tables for the **and**, **or**, **xor**, and **not** primitives are shown in Table.

The truth table for the other four gates is the same, except that the outputs are complemented. Note that for the **and** gate, the output is 1 only when both inputs are 1 and the output is 0 if any input is 0. Otherwise, if one input is x or z, the output is x. The output of the **or** gate is 0 if both inputs are 0, is 1 if any input is 1, and is x otherwise.

Truth Table for Predefined Primitive Gates

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

xor	0	1	x	z	not	input	output
0	0	1	x	x		0	1
1	1	0	x	x		1	0
x	x	x	x	x		x	x
z	x	x	x	x		z	x

We now present two examples of gate-level modeling. Both examples use identifiers having multiple bit widths, called vectors. The syntax specifying a vector includes within square brackets two numbers separated with a colon. 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. (Note: The first (left-most) number (array index) listed is always the most significant bit of the vector). The individual bits are specified within square brackets, so D[2] specifies bit 2 of D. It is also possible to address parts (contiguous bits) of vectors. For example, SUM[2: 0] specifies the three least significant bits of vector SUM.

HDL Example below shows the gate-level description of a two-to-four-line decoder.

```

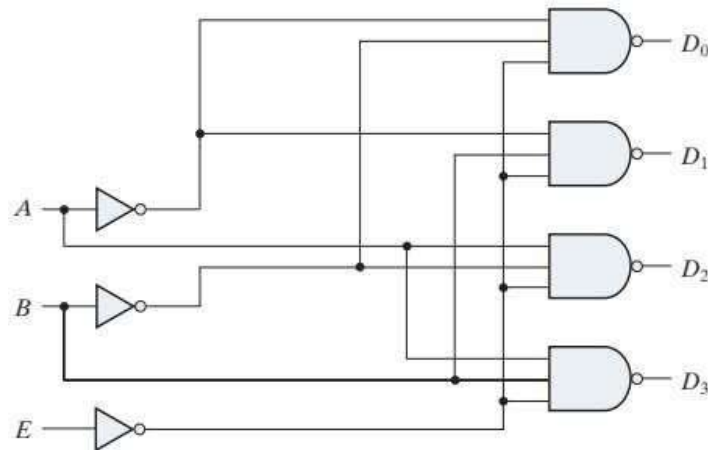
module decoder_2x4_gates (D, A, B, enable);
output      [0: 3]      D;
input      A, B;
input      enable;
wire      A_not, B_not, enable_not;

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

```

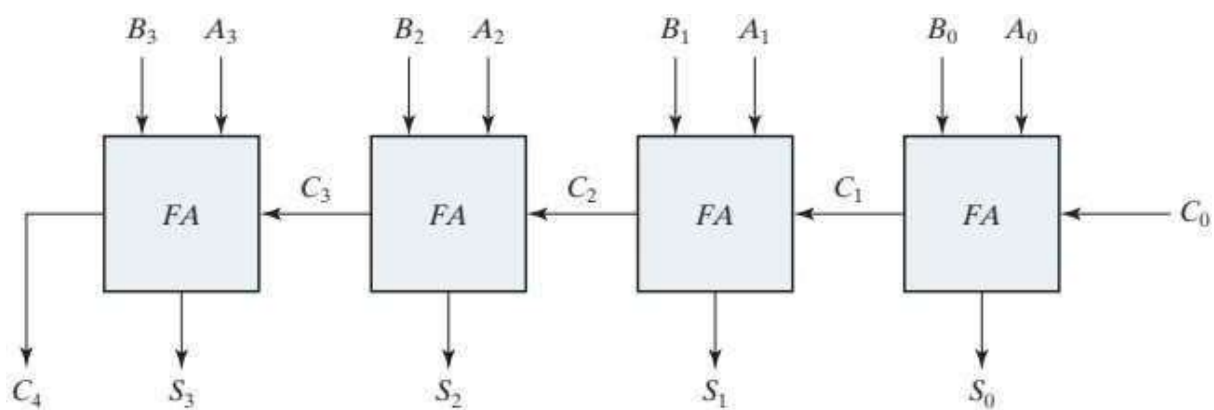
This decoder has two data inputs A and B and an enable input E. The four outputs are specified with the vector D. The **wire** declaration is for internal connections. Three not gates produce the complement of the inputs, and four **nand** gates provide the outputs for D. Remember that the output is always listed

first in the port list of a primitive, followed by the inputs. This example describes the decoder implementation shown in below Fig.



HDL Example below shows the gate-level description of a Four-bit ripple carry adder

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. Take, for example, the binary adder shown in below fig. It can be considered as a top-block component built with four full-adder blocks, while each full adder is built with two half-adder blocks. In a top-down design, the four-bit adder is defined first, and then the two adders are described. In a bottom-up design, the half adder is defined, then each full adder is constructed, and then the four-bit adder is built from the full adders.



Four-bit adder

A bottom-up hierarchical description of a four-bit adder is shown in HDL Example below. 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.

// Gate-level description of four-bit ripple carry adder

```

// 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

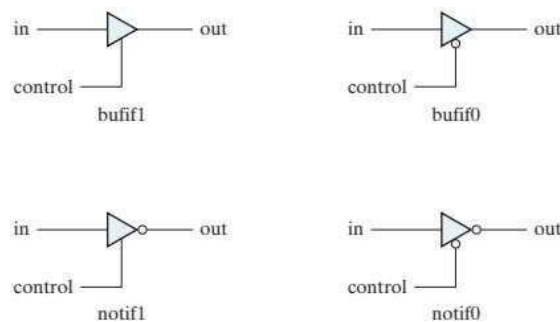
```

HDL Example above illustrates Verilog 2001, 2005 syntax, which eliminates extra typing of identifiers declaring the mode (e.g., output), type (reg), and declaration of a vector range (e.g., [3: 0]) of a port. The first version of the standard (1995) uses separate statements for these declarations.

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 below. 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 (output, input, control);



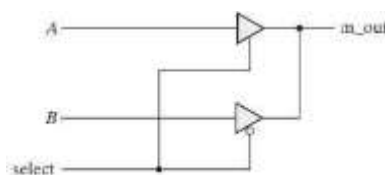
Three-state gates

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 below.



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

The HDL description must use a **tri** data type for the 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
```

Dataflow Modeling

Dataflow modeling of combinational logic uses a number of operators that act on binary operands to produce a binary result. Dataflow modeling uses continuous assignments and the keyword **assign**. 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

$$\text{assign } Y = (A \&\& S) \parallel (B \&\& \neg S)$$

The relationship between Y, A, B, and S is declared by the keyword **assign**, followed by the target output Y and an equals sign.

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

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

```
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
```

// 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
```

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 below 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 (in addition to the output declaration). The procedural assignment statements inside the **always** block are executed every time there is a change in any of the variables listed after the **@** symbol. (Note that there is no semicolon (;) at the end of the

always statement.) 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.

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 = B;
endmodule
```

Behavioral description of four-to-one-line multiplexer

```
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
```

Synchronous Sequential Logic

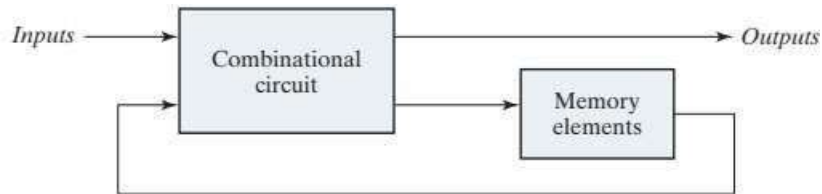
Hand-held devices, cell phones, navigation receivers, personal computers, digital cameras, personal media players, and virtually all electronic consumer products have the ability to send, receive, store, retrieve, and process information represented in a binary format. The technology enabling and supporting these devices is critically dependent on electronic components that can store information, i.e., have memory. The digital circuits considered thus far have been combinational—their output depends only and immediately on their inputs—they have no memory, i.e., dependence on past values of their inputs. Sequential circuits, however, act as storage elements and have memory. They can store, retain, and then retrieve information when needed later.

Sequential Circuits

A block diagram of a sequential circuit is shown in Fig below. 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.**

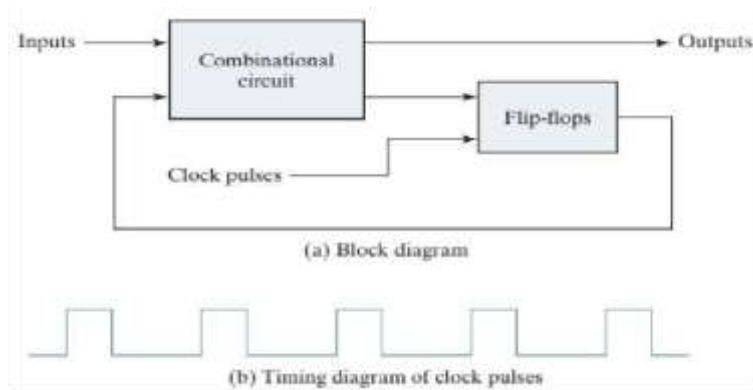


Block diagram of sequential circuit

There are two main types of sequential circuits, and their classification is a function of the timing of their signals. A **synchronous** sequential circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time. The behavior of an **asynchronous** sequential circuit depends upon the input signals at any instant of time and the order in which the inputs change. The storage elements commonly used in asynchronous sequential circuits are time-delay devices. The storage capability of a time-delay device varies with the time it takes for the signal to propagate through the device.

A synchronous sequential circuit employs signals that affect the storage elements at only discrete instants of time. Synchronization is achieved by a timing device called a **clock generator**, which provides a clock signal having the form of a periodic train of clock pulses. The clock signal is commonly denoted by the identifiers **clock** and **clk**. The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of each pulse. Synchronous sequential circuits that use *clock pulses* to control storage elements are called *clocked sequential circuits* and are the type most frequently encountered in practice. They are called *synchronous circuits* because the activity within the circuit and the resulting updating of stored values is synchronized to the occurrence of clock pulses.

The storage elements (memory) used in clocked sequential circuits are called *flip-flops*. A *flip-flop* is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either 0 or 1. A sequential circuit may use many flip-flops to store as many bits as necessary. The block diagram of a synchronous clocked sequential circuit is shown in Fig(a) below. The outputs are formed by a combinational logic function of the inputs to the circuit or the values stored in the flip-flops (or both). The value that is stored in a flip-flop when the clock pulse occurs is also determined by the inputs to the circuit or the values presently stored in the flip-flop (or both). If the clock (synchronizing) pulses arrive at a regular interval, as shown in the timing diagram in Fig(b), the combinational logic must respond to a change in the state of the flip-flop in time to be updated before the next pulse arrives.



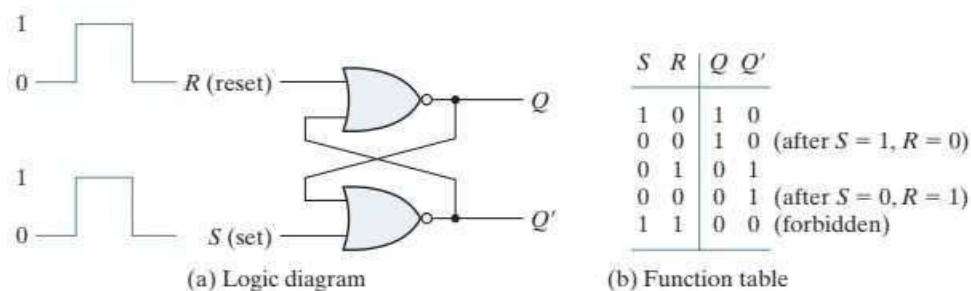
A change in state of the flip-flops is initiated only by a clock pulse transition—for example, when the value of the clock signals changes from 0 to 1. Thus, the transition from one state to the next occurs only at predetermined intervals dictated by the clock pulses.

Storage Elements: Latches

A storage element in a digital circuit can maintain a binary state indefinitely (as long as power is delivered to the circuit), 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 (rather than signal transitions) 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.

SR Latch

The SR latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled S for set and R for reset. The SR latch constructed with two cross-coupled NOR gates is shown in Fig(a). 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.



SR latch with NOR gates

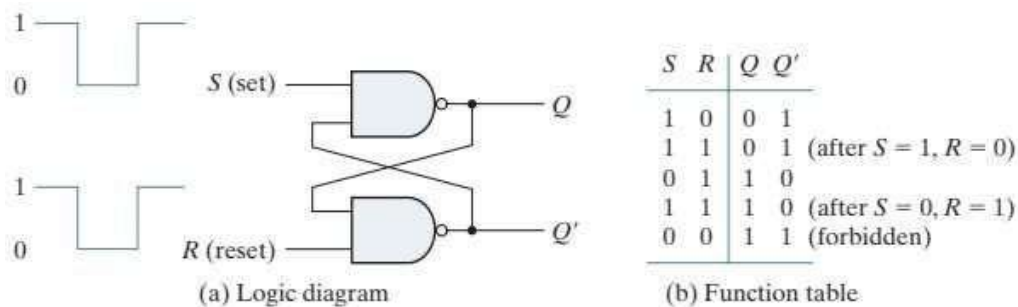
As shown in the function table of above Fig(b), two input conditions cause the circuit to be in the set state. The first condition ($S = 1, R = 0$) is the action that must be taken by input S to bring the circuit to the set state. Removing the active input from S leaves the circuit in the same state. After both inputs return to 0, it is then possible to shift to the reset state by momentarily applying a 1 to the R input. The 1 can then be removed from R, whereupon 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 are not applied to both inputs simultaneously.

The SR latch with two cross-coupled NAND gates is shown in below Fig. 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

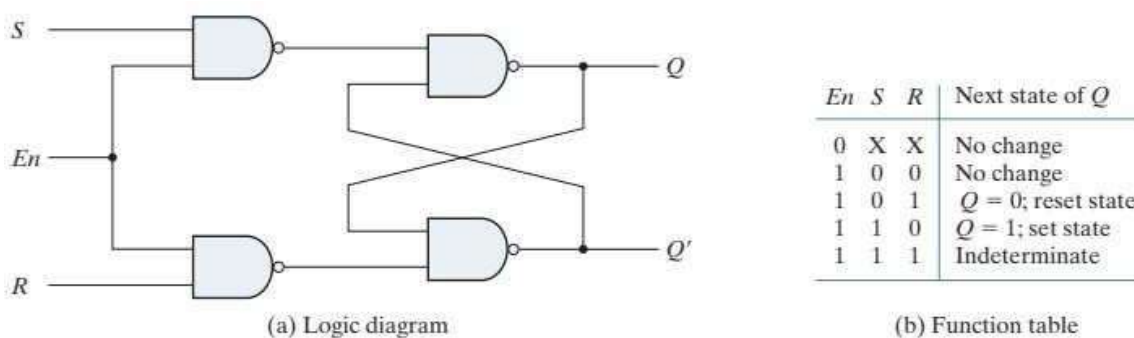
in the set state. After both inputs go back to 1, we are allowed to change the state of the latch by placing a 0 in the R input. This action causes the circuit to go to the reset state and stay there even after both inputs return to 1. The condition that is forbidden for the NAND latch is both inputs being equal to 0 at the same time, an input combination that should be avoided.



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 below Fig. 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 (active-high enabled).



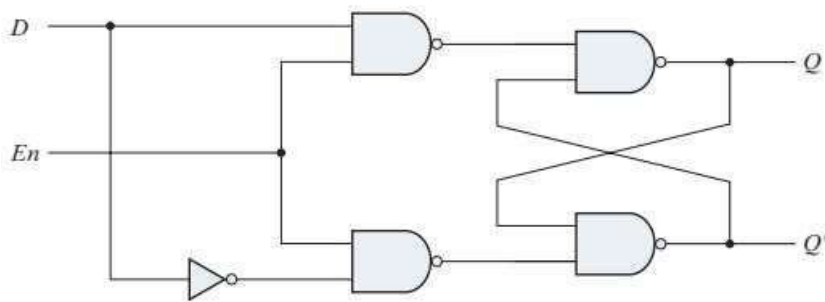
SR latch with control input

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.

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 below Fig. This latch has only two inputs: D (data) and En (enable).



(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

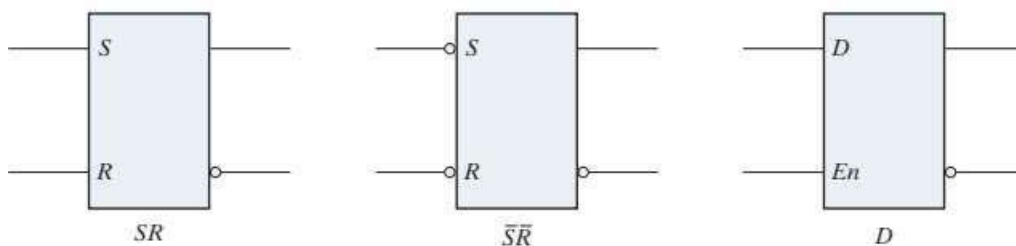
(b) Function table

D latch

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.

The binary information present at the data input of the D latch is transferred to the Q output when the enable input is asserted. The output follows changes in the data input as long as the enable input is asserted. This situation provides a path from input D to the output, and for this reason, the circuit is often called a transparent latch.

The graphic symbols for the various latches are shown in Fig. 5.7. 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. In the case of a NAND gate latch, bubbles are added to the inputs to indicate that setting and resetting occur with a logic-0 signal. The graphic symbol for the D latch has inputs D and En indicated inside the block.

Graphic symbols for latches

Storage Elements: Flip-Flops

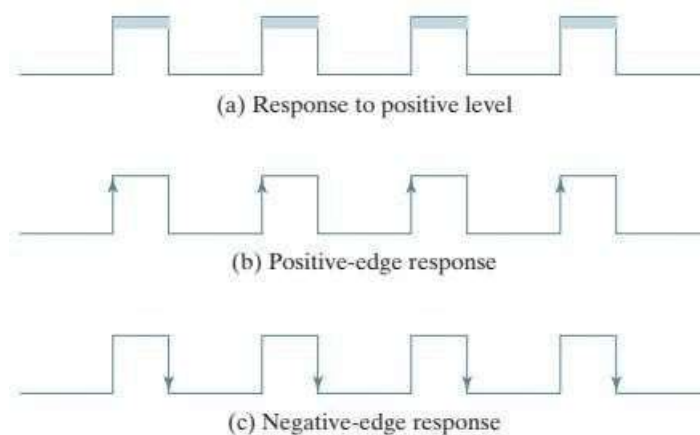
The state of a latch or flip-flop is switched by a change in the control input. This momentary change is called a trigger, and the transition it causes is said to trigger the flip-flop. The D latch with pulses in its control input is essentially a flip-flop that is triggered every time the pulse goes to the logic-1 level.

As we know, a sequential circuit has a feedback path from the outputs of the flip-flops to the input of the combinational circuit. 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. Because of this unreliable operation, the output of a latch cannot be applied directly or through combinational logic to the input of the same or another latch when all the latches are triggered by a common clock source.

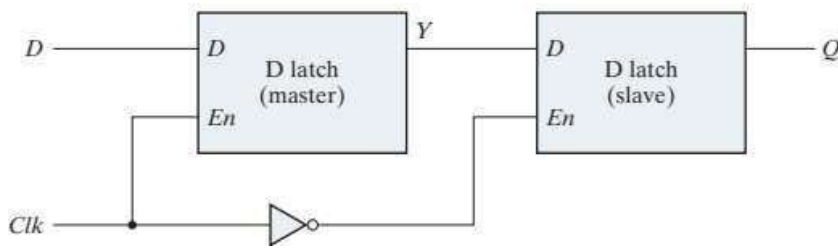
Flip-flop circuits are constructed in such a way as to make them operate properly when they are part of a sequential circuit that employs a common clock. As shown in below Fig(a), a positive level response in the enable input allows changes in the output when the D input changes while the clock pulse stays at logic 1. The key to the proper operation of a flip-flop is to trigger it only during a signal transition. This can be accomplished by eliminating the feedback path that is inherent in the operation of the sequential circuit using latches. A clock pulse goes through two transitions: from 0 to 1 and the return from 1 to 0. As shown in below Fig, the positive transition is defined as the positive edge and the negative transition as the negative edge. There are two ways that a latch can be modified to form a flip-flop. One way is to employ two latches in a special configuration that isolates the output of the flip-flop and prevents it from being affected while the input to the flip-flop is changing. Another way is to produce a flip-flop that triggers only during a signal transition (from 0 to 1 or from 1 to 0) of the synchronizing signal (clock) and is disabled during the rest of the clock pulse.



Clock response in latch and flip-flop

Edge-Triggered D Flip-Flop

The construction of a D flip-flop with two D latches and an inverter is shown in Fig below. The first latch is called the master and the second the slave. The circuit samples the D input and changes its output Q only at the negative edge of the synchronizing or controlling clock (designated as Clk). When the clock is 0, the output of the inverter is 1. The slave latch is enabled, and its output Q is equal to the master output Y. The master latch is disabled because Clk = 0. When the input pulse changes to the logic-1 level, the data from the external D input are transferred to the master. The slave, however, is disabled as long as the clock remains at the 1 level, because its enable input is equal to 0. Any change in the input changes the master output at Y, but cannot affect the slave output. When the clock pulse returns to 0, the master is disabled and is isolated from the D input. At the same time, the slave is enabled and the value of Y is transferred to the output of the flip-flop at Q. Thus, a change in the output of the flip-flop can be triggered only by and during the transition of the clock from 1 to 0.

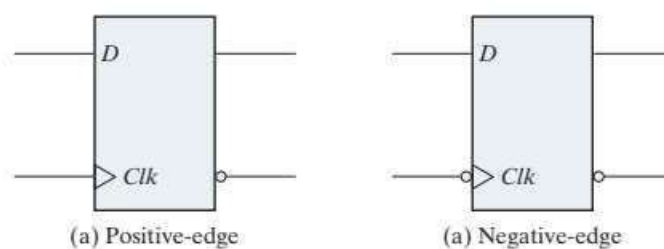


Master–slave D flip-flop

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.

Another construction of an edge-triggered D flip-flop uses three SR latches as shown in below Fig. 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.

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 graphic symbol for the edge-triggered D flip-flop is shown in above Fig. 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. The absence of a bubble designates a positive-edge response.

Other Flip-Flops

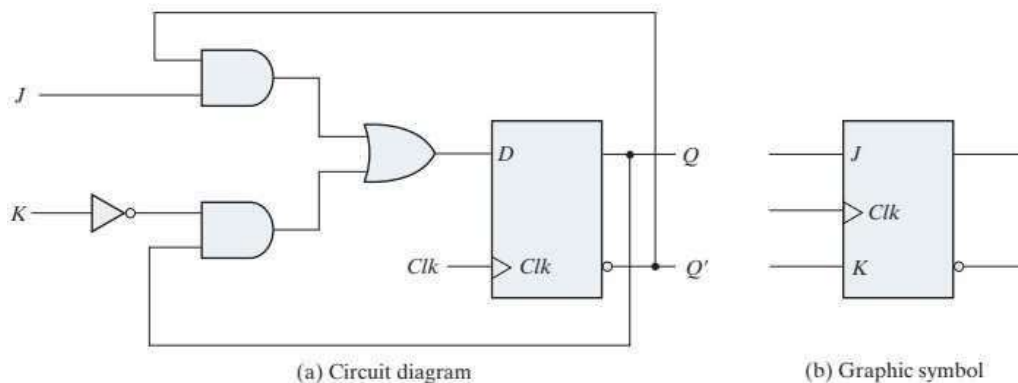
The most economical and efficient flip-flop constructed in this manner is the edge-triggered D flip-flop, because it requires the smallest number of gates. Other types of flip-flops can be constructed by using the

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(a) below. 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:

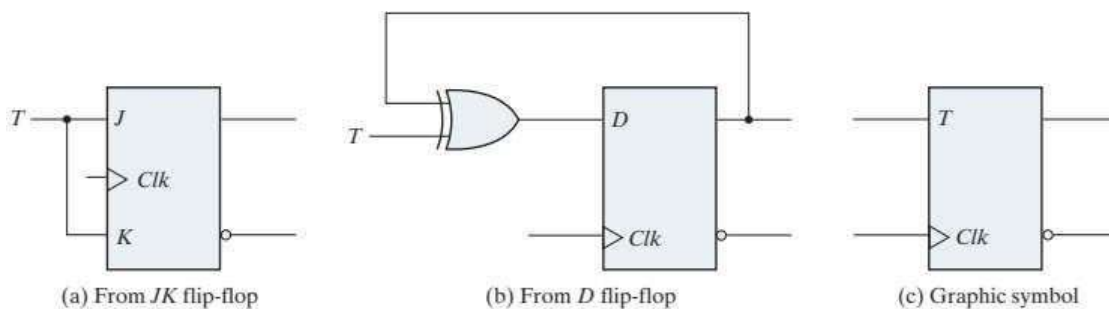
$$D = JQ' + K'Q$$

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(b).



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 below in Fig(a). 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.



T flip-flop

The T flip-flop can be constructed with a D flip-flop and an exclusive-OR gate as shown in above Fig(b). The expression for the D input is

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

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.

1. CHARACTERISTIC EQUATIONS: [TB2, Pg. NO - 329]

The algebraic description of the next-state table of a flip-flop is called the characteristic equation of the flip flop.

Characteristic equation of SR flip flop.

Simplified Function Table.

S	R	Q^+
0	0	Q
0	1	0
1	0	1
1	1	-

S	00	01	11	10
0	0	1	0	0
1	1	1	-	-

$$Q^+ = S + \bar{R}Q$$

Next State Function Table.

S	R	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	-
1	1	1	-

Characteristic Equation of D flip-flop.

Simplified Function Table

D	Q
0	0
1	1

D	0	1
0	0	0
1	0	1

Next State Function Table.

D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

Characteristic equation of T-Flip-Flop.Simplified Function Table

T	Q
0	Q
1	$\bar{Q}$

T \ Q	0	1
0	0	①
1	①	0

$$Q^+ = \bar{T}Q + T\bar{Q}$$

$$= T \oplus Q$$

Next-State Function Table

T	Q	Q ⁺
0	0	0
0	1	1
1	0	1
1	1	0

Characteristic equation of JK-Flip-Flop.Simplified Function Table

J	K	Q
0	0	Q
0	1	0
1	0	1
1	1	$\bar{Q}$

J \ KQ	00	01	11	10
0	0	①	0	0
1	①	①	0	①

$$Q^+ = J\bar{Q} + \bar{K}Q$$

Next-State Function Table

J	K	Q	Q ⁺
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Note: In the function tables the control signal is not

