Unit 4

Combinational Logic

Content:

- Design Procedure
- Adders
- Subtractors
- Code Conversion
- Analysis Procedure
- NAND Circuits
- NOR Circuits
- Exclusive -OR Circuit

Introduction

- Logic circuits for digital systems may be combinational or sequential

- **Combinational logic** is a *type of digital logic which is implemented by Boolean circuits, where the output is a pure function of the present input only*.
- This is in contrast to sequential logic, in which the output depends not only on the present input but also on the past input. In other words, sequential logic has *memory* while combinational logic does not.

Combinational Circuit

These are the circuit gates employing combinational logic.

- A combinational circuit consists of *n* input variables, logic gates, and *m* output variables. The logic gates accept signals from the inputs and generate signals to the outputs.
- For *n* input variables, there are 2^n possible combinations of binary input values. For each possible input combination, there is one and only one possible output combination. A combinational circuit can be described by *m* Boolean functions, one for each output variable. Each output function is expressed in terms of the *n* input variables.

Obviously, both input and output data are represented by binary signals, i.e., logic-1 and the other logic-0. The n input binary variables come from an external source; the m output variables go to an external destination.

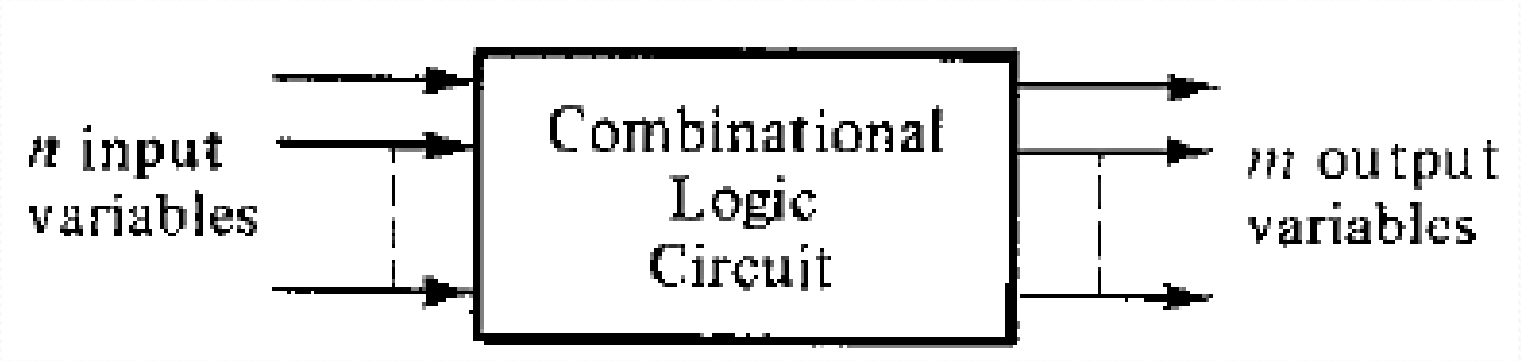


Fig: Block diagram of combinational circuit

Design procedure

The design of combinational circuits starts from the verbal outline of the problem and ends in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be easily obtained.

The procedure involves the following steps:

1. Specification

- Write a specification for the circuit if one is not already available.
- Specification such as names for inputs and outputs.

2. Formulation

- Derive a truth table or initial Boolean equations that define the required relationships between the inputs and outputs

3. Optimization

- Apply different optimization methods, such as algebraic manipulation, K-map method.
- Draw a logic diagram for the resulting circuit using ANDs, ORs, and inverters

4. Technology Mapping

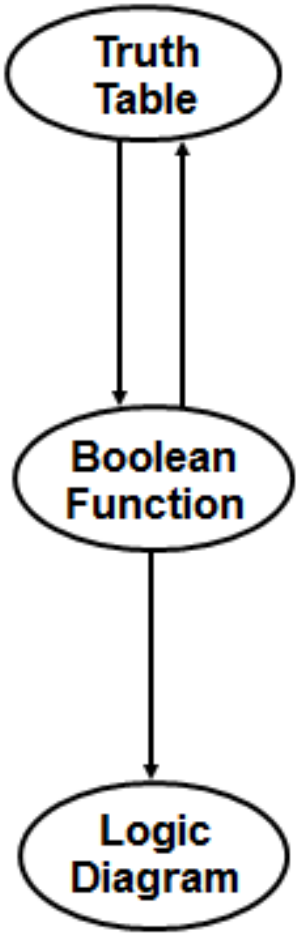
- Map the logic diagram to the implementation technology selected.

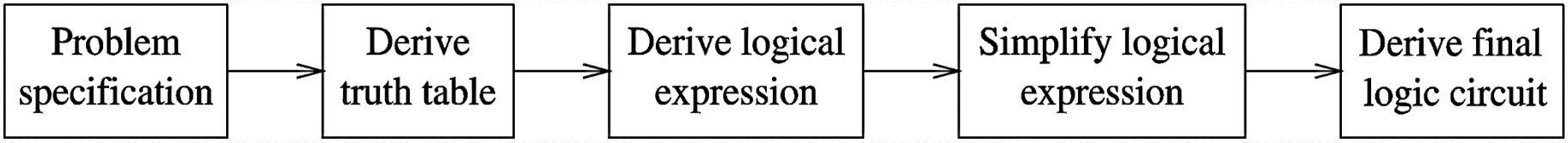
5. Verification

- Verify the correctness of the final design manually or using simulation.

In simple words, we can list out the design procedure of combinational circuits as:

1. The problem is stated.
2. The number of available input variables and required output variables is determined.
3. The input and output variables are assigned letter symbols.
4. The truth table that defines the required relationships between inputs and outputs is derived.
5. The simplified Boolean function for each output is obtained.
6. The logic diagram is drawn.
7. Verify the correctness of the design.





Example: Design of Combinational Circuit

Q. Design a combinational circuit with three inputs and one output. The output must be logic 1 when the binary value of the inputs is less than 110 and logic 0 otherwise. Use only NAND gates.

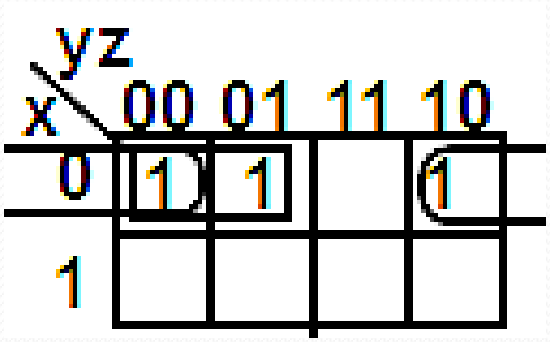
Input: 3 (let x, y and z)

Output: 1 (let F)

Truth Table:

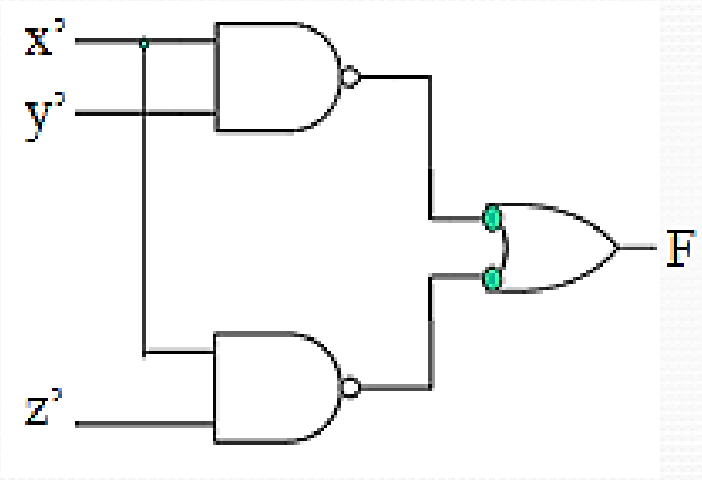
x	y	z	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Simplification:



Map: F = x'y' + x'z'

Logic Diagram:



- **Adders**

Digital computers perform a variety of information-processing tasks. Among the basic functions encountered are the various arithmetic operations. The most basic arithmetic operation, no doubt, is the addition of two binary digits.

1. Half-Adder

- A combinational circuit that performs the addition of two bits is called a *half-adder.*
- Circuit needs **two inputs** and **two outputs**. The input variables designate the augend (x) and addend (y) bits; the output variables produce the sum (S) and carry (C).
- Now we **formulate a Truth table** to exactly identify the function of half-adder.

- Design of Half Adder:

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

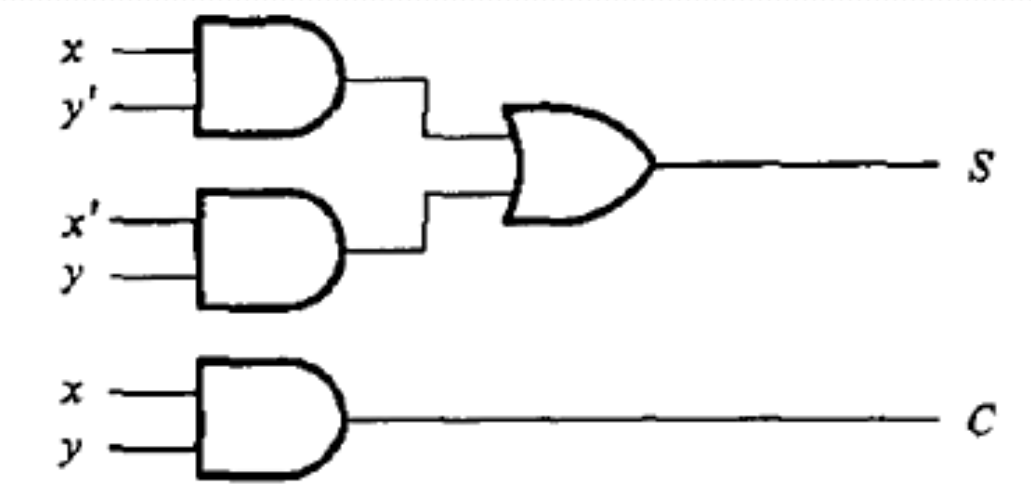
Input: 2 (let x and y)
Output: 2 (let C for Carry and S for Sum)
Truth table:

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

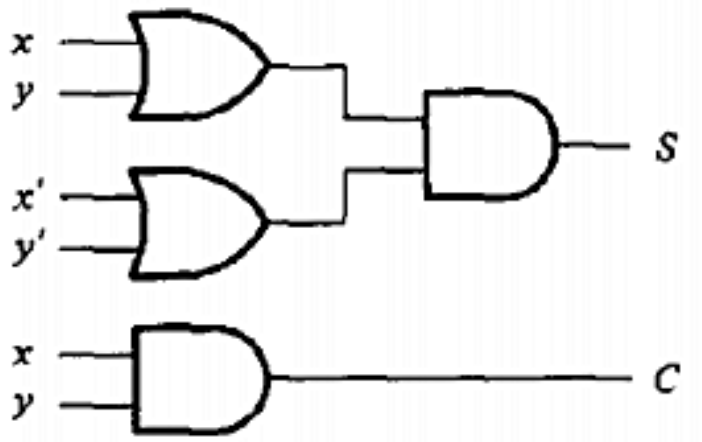
S = x'y + xy'

C = xy

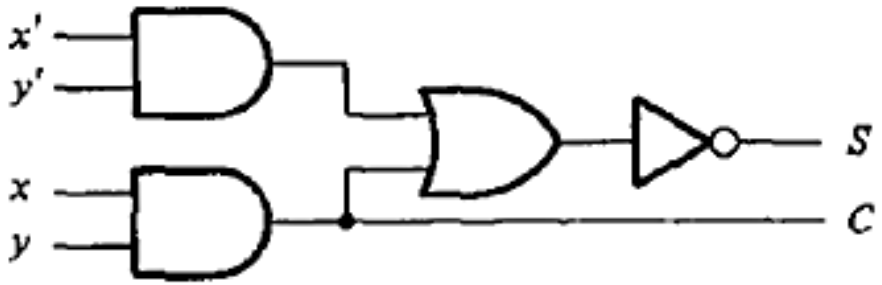
Implementation:



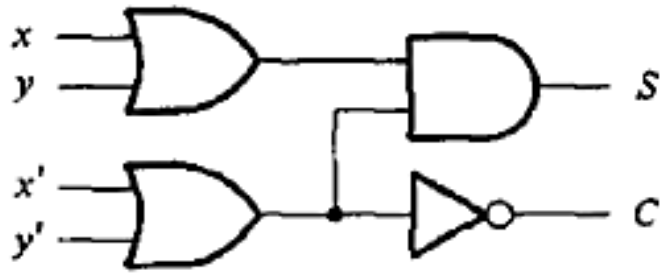
Other realizations and implementations of Half-adders are:



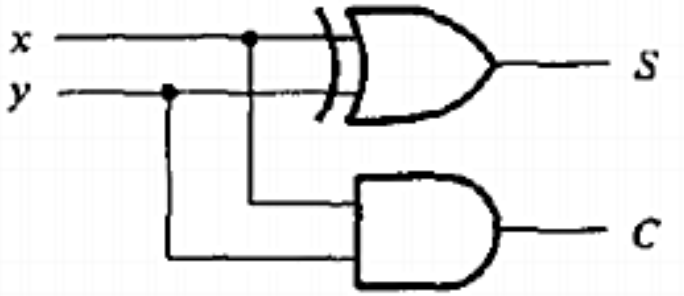
(b) $S = (x + y)(x' + y')$
$C = xy$



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

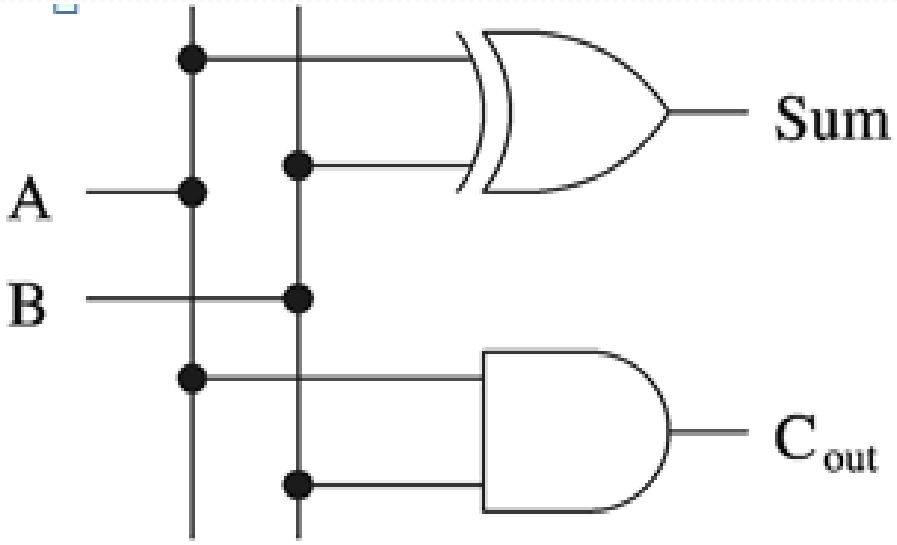


(d) $S = (x + y)(x' + y')$
$C = (x' + y')'$



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

A	B	Sum	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



(a) Half-adder truth table and implementation

2. Full-Adder

- A *full-adder* is a combinational circuit that forms the arithmetic sum of three input 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.

- Design:

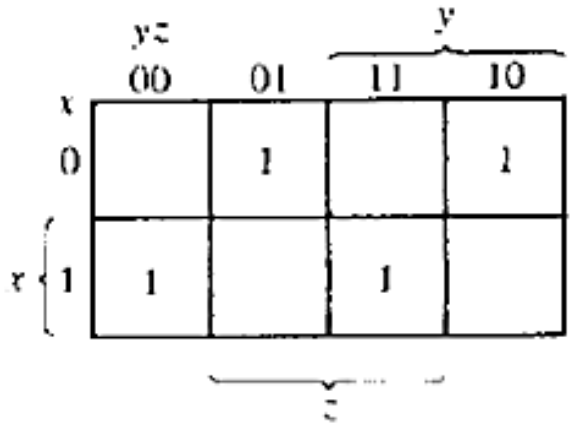
Input: 3(Let *x*, *y*, *z*)

Output: 2(Let C for carry and S for sum)

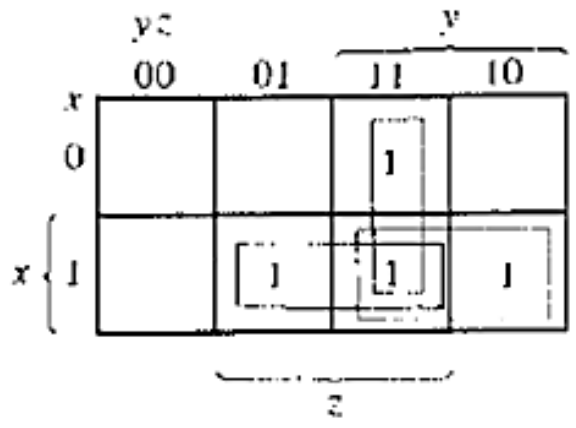
Truth Table:

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

Boolean functions with simplification by K-map:



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



$C = xy + xz + yz$

Implementation:

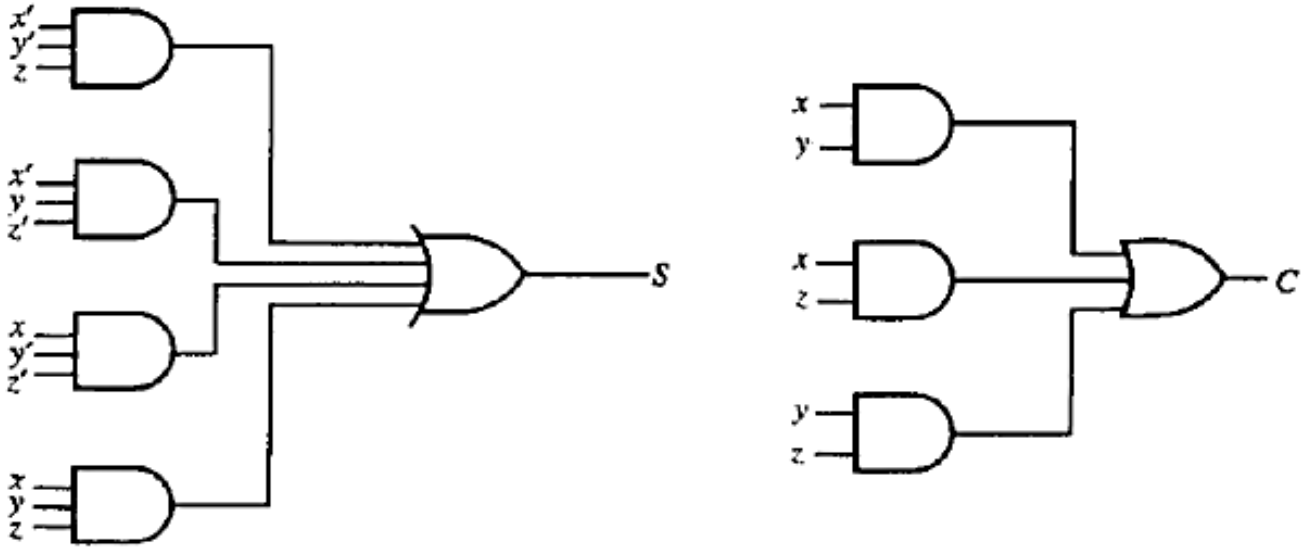


Fig: Implementation of a full-adder in sum of products.

A full-adder can be implemented with **two half-adders** and one OR gate.

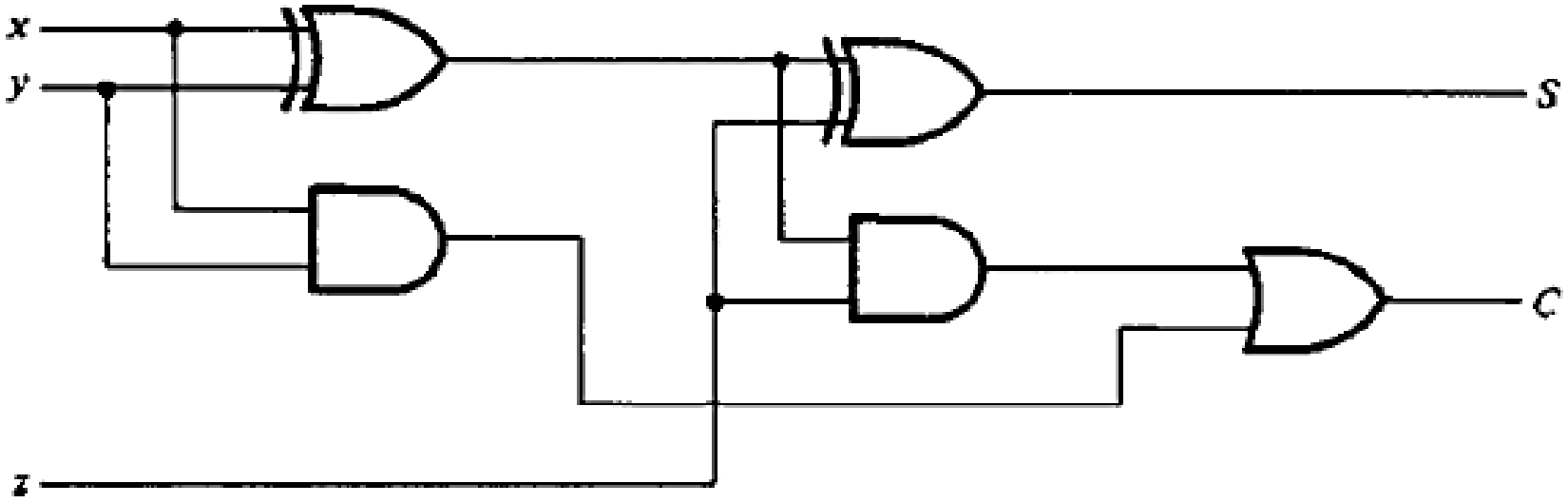


Fig: Implementation of a full-adder with two half-adders and an OR gate

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

$S = z \oplus (x \oplus y)$

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

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

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

$C = z(x \oplus y) + xy$

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

$= xy'z + x'yz + xy$

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

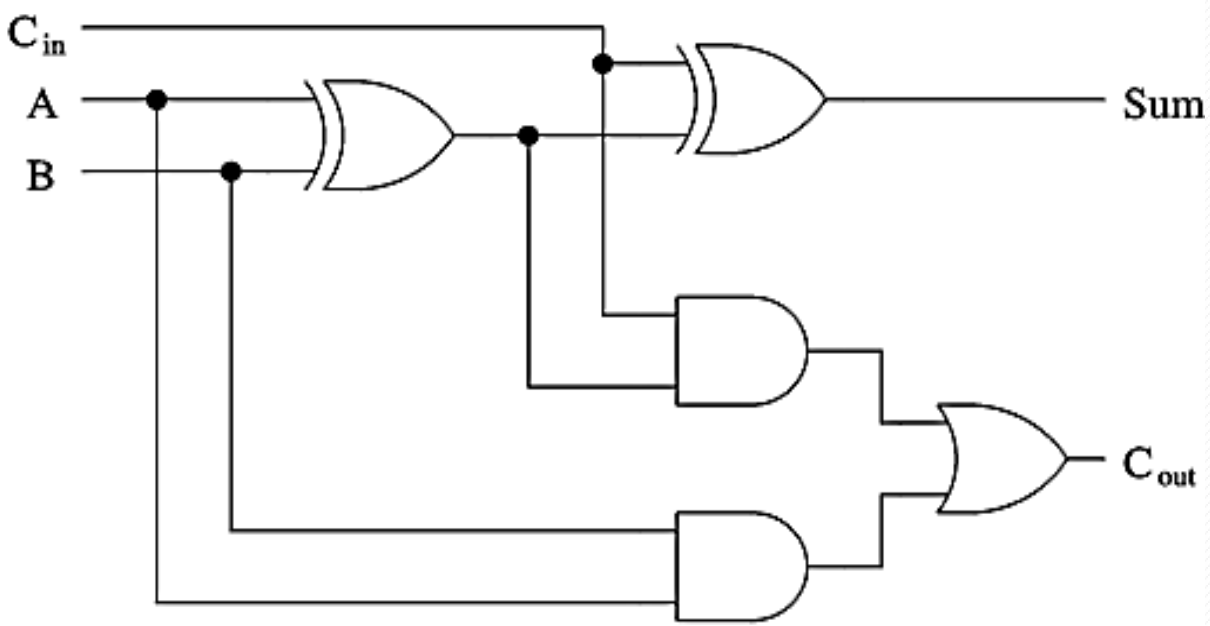


Fig: Full-adder truth table and implementation

- **Subtractors:**

The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding it to the minuend. By this method, the subtraction operation becomes an addition operation requiring full-adders for its machine implementation. It is possible to implement subtraction with logic circuits in a direct manner, as done with paper and pencil. By this method, each subtrahend bit of the number is subtracted from its corresponding **significant minuend bit** to form a **difference bit**. If the minuend bit is smaller than the subtrahend bit, a 1 is borrowed from the next significant position. Just as there are half- and full-adders, there are half- and full-subtractors.

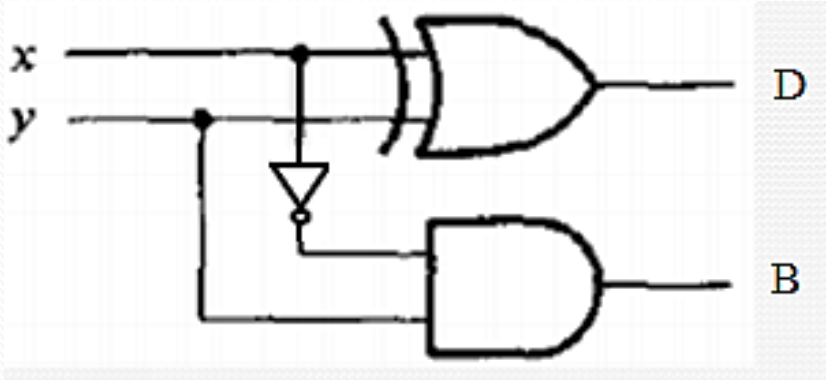
1. Half-Subtractor

- A half-subtractor is a combinational circuit that subtracts two bits and produces their difference bit.
- Denoting minuend bit by x and the subtrahend bit by *y*. To perform *x* - *y*, we have to check the relative magnitudes of *x* and *y*:
  - If $x \geq y$, we have three possibilities: 0 - 0 = 0, 1 - 0 = 1, and 1 - 1 = 0.
  - If $x < y$, we have 0 - 1, and it is necessary to borrow a 1 from the next higher stage which adds 2 to minuend bit.
- The half-subtractor needs **two outputs**, difference(D) and borrow(B).
- The **truth table** **Boolean Expression** **Implementation**

x	*y*	*B*	*D*
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

D = x'y + xy'

B = x'y



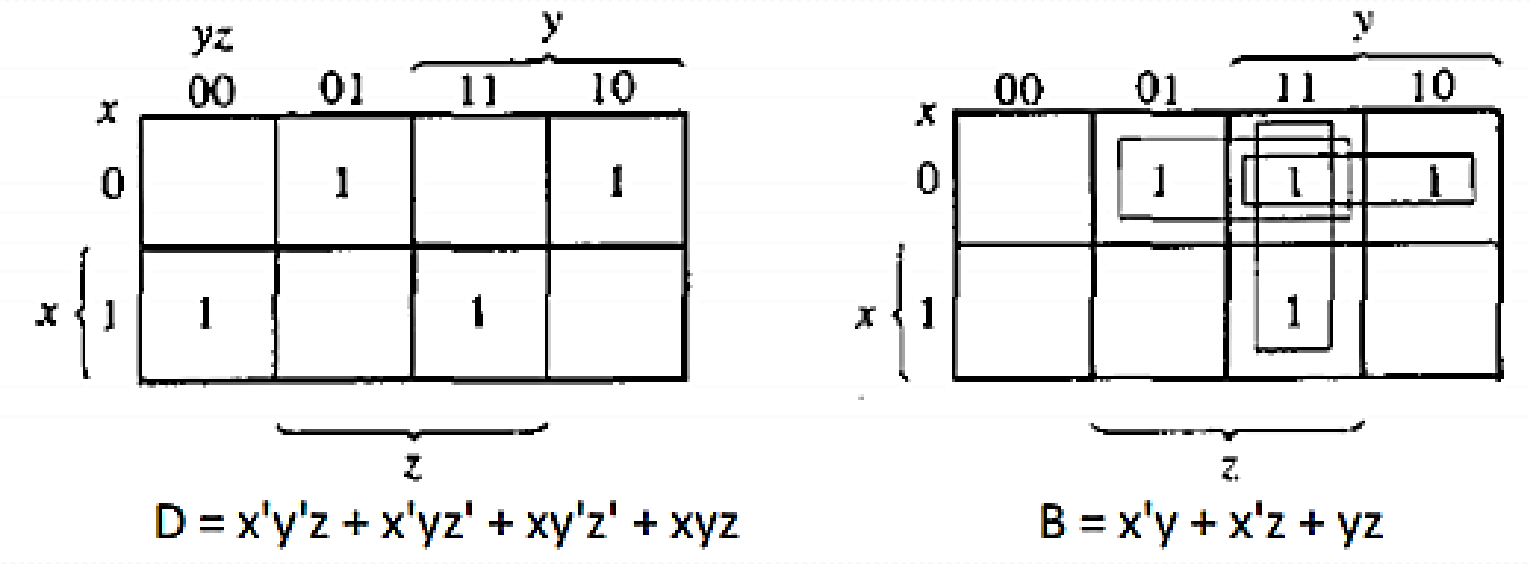
2. Full-Subtractor

- A full-subtractor is a combinational circuit that performs a subtraction between two bits, taking into account that a 1 **may have been borrowed** by a lower significant stage.
- This circuit has **three inputs** and **two outputs**. The three inputs, *x*, *y*, and z, denote the minuend, subtrahend, and previous borrow, respectively. The two outputs, *D* and *B*, represent the difference and output-borrow, respectively.
- **Truth-table**

x	*y*	*z*	*B*	*D*
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

- The 1's and 0's for the output variables are determined from the subtraction of $x - y - z$.
- The combinations having input borrow $z = 0$ reduce to the same four conditions of the half-adder.
- For $x = 0$, $y = 0$, and $z = 1$, we have to borrow a 1 from the next stage, which makes $B = 1$ and adds 2 to x. Since $2 - 0 - 1 = 1$, $D = 1$.
- For $x = 0$ and $y\ z = 1\ 1$, we need to borrow again, making $B = 1$ and $x = 2$. Since $2 - 1 - 1 = 0$, $D = 0$.
- For $x = 1$ and $y\ z = 0\ 1$, we have $x - y - z = 0$, which makes $B = 0$ and $D = 0$.
- Finally, for $x = 1$, $y = 1$, $z = 1$, we have to borrow 1, making $B = 1$ and $x = 3$, and $3 - 1 - 1 = 1$, making $D=1$.

The simplified Boolean functions for the two outputs of the full-subtractor are derived in the maps:



Code Conversion

The availability of a large variety of codes for the same discrete elements of information results in the use of different codes by different digital systems. It is sometimes necessary to use the output of one system as the input to another. A conversion circuit must be inserted between the two systems if each uses different codes for the same information. Thus, a **code converter** is a circuit that makes the two systems compatible even though each uses a different binary code.

To convert from binary code A to binary code B, code converter has **input lines** supplying the bit combination of elements as specified by code A and the output lines of the converter generating the corresponding bit combination of code B.

A Code converter (combinational circuit) performs this transformation by means of logic gates.

The design procedure of code converters will be illustrated by means of a *specific example* of conversion from the BCD to the excess-3 code.

Note:

Excess-3 Code:

The excess-3 code for a decimal digit is the binary combination corresponding to the decimal digit plus 3.

For example, the excess-3 code for decimal digit 5 is the binary combination for 5 + 3 = 8, which is 1000.

Design example: BCD to Excess-3 code converter

5-step design procedure of this code converter

1. Specification

- Transforms BCD code for the decimal digits to Excess-3 code for the decimal digits
- BCD code words for digits 0 through 9: 4-bit patterns 0000 to 1001, respectively.
- Excess-3 code words for digits 0 through 9: 4-bit patterns consisting of 3 (binary 0011) added to each BCD code word.

2. Formulation

- Conversion of 4-bit codes can be most easily formulated by a truth table.
- Variables- BCD: A, B, C, D
- Variables- Excess-3: W, X, Y, Z
- Don’t Cares: BCD 1010 to 1111

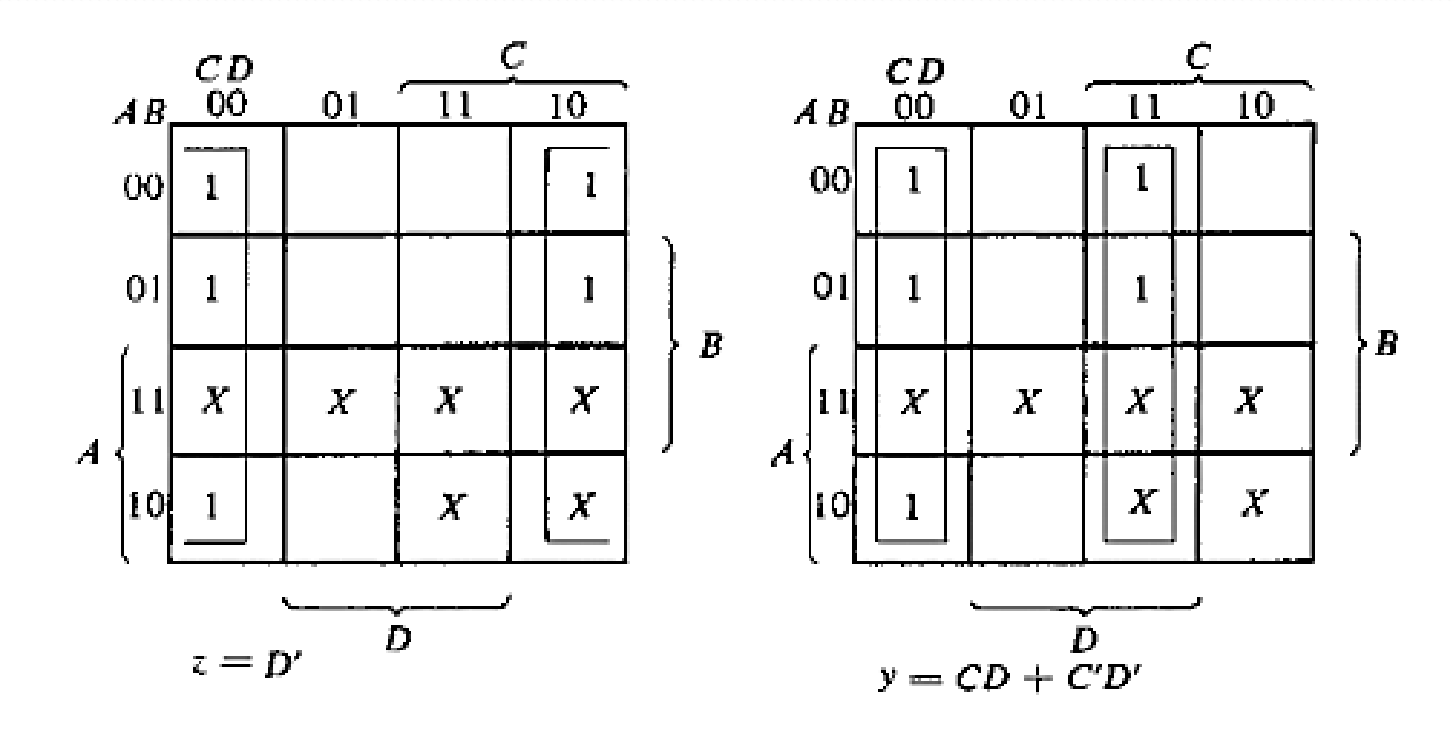
Input BCD				Output Excess-3 Code			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

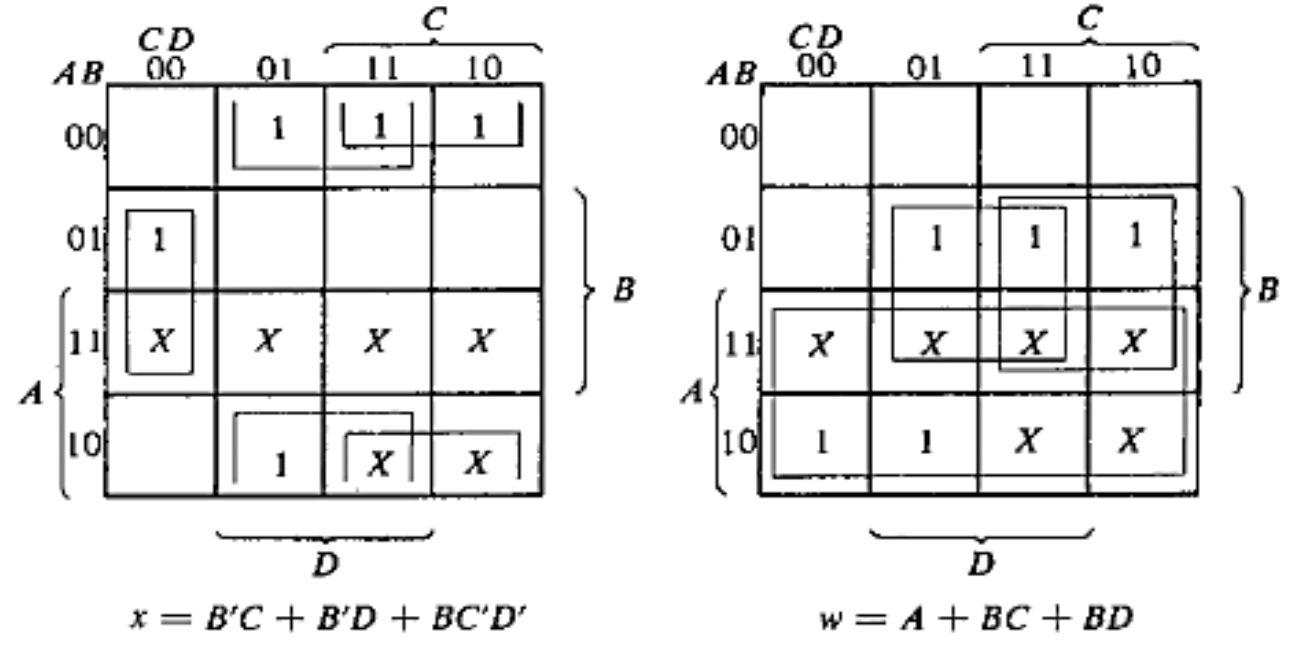
Note that the four BCD input variables may have 16 bit combinations, but only 10 are listed in the truth table. Others designate “don’t care conditions”.

3. Optimization

a. 2-level optimization

The k-maps are plotted to obtain simplified sum-of-products Boolean expressions for the outputs. Each of the four maps represents one of the outputs of the circuit as a function of the four inputs.





b. Multiple-level optimization

This second optimization step reduces the number of gate inputs. The following manipulation illustrates optimization with multiple-output circuits implemented with three levels of gates:

$$z = D'$$

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

$$x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$$

$$= B'(C + D) + B(C + D)'$$

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

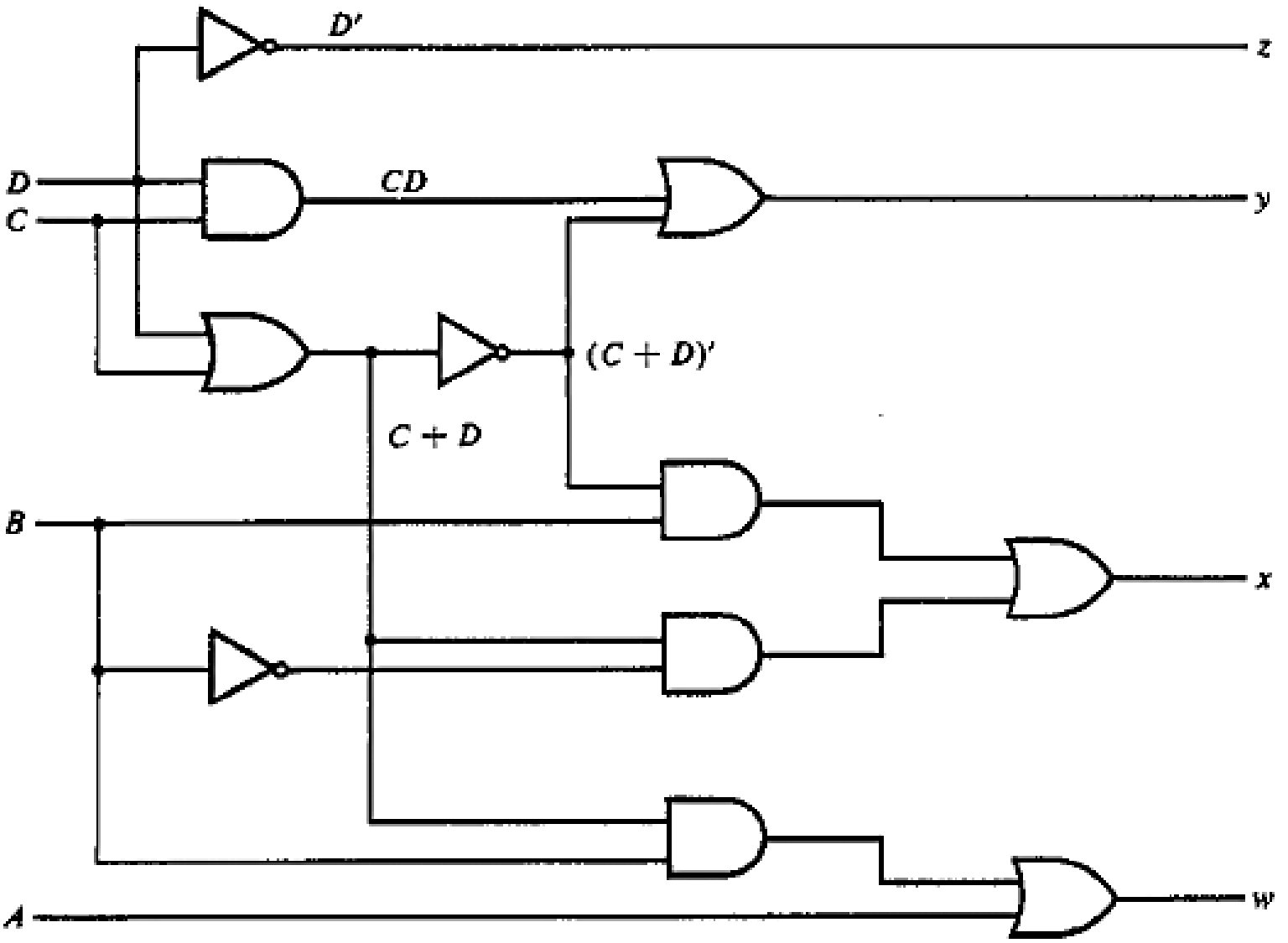


Fig: Logic Diagram of BCD- to-Excess-3 Code Converter

4. Technology mapping

This is concerned with the act of mapping of basic circuit (using AND, OR and NOT gates) to a specific circuit technology (such as NAND, NOR gate tech.).

Note: This is advanced topic, and I won't discuss here. For exam point of view, if you are asked for BCD-to-Excess-3 code converter, you will finish up your answer by drawing basic circuit shown above.

5. Verification

Assignment:

Q. Design BCD to Seven-Segment Decoder.

- **Analysis Procedure**

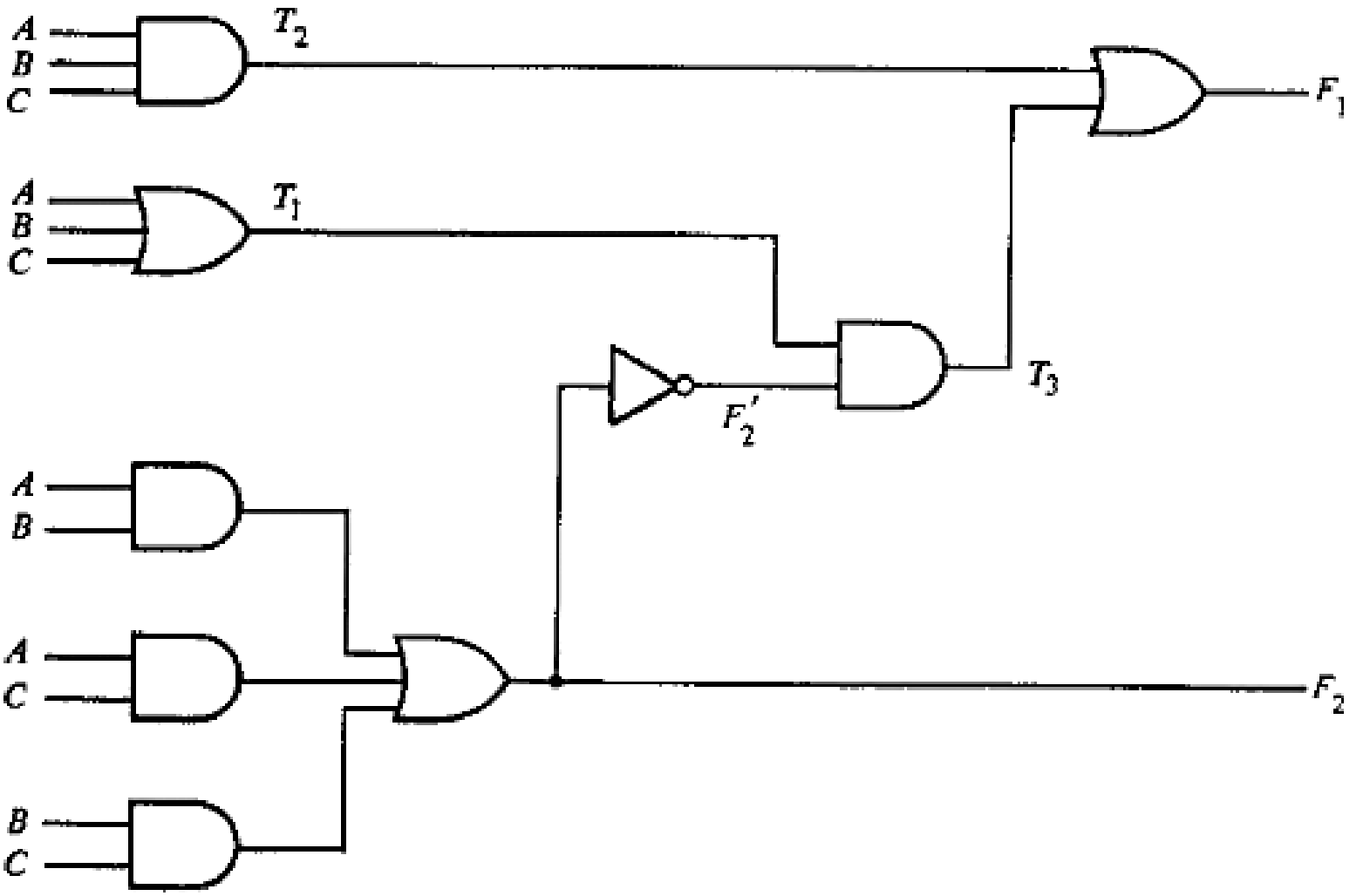
The design of a combinational circuit starts from the verbal specifications of a required function and ends with a set of output Boolean functions or a logic diagram. The **analysis of a combinational circuit** is somewhat the reverse process. It starts with a given logic diagram and culminates with a set of Boolean functions, a truth table, or a verbal explanation of the circuit operation.

- **Obtaining Boolean functions from logic diagram**

Steps in analysis:

1. The first step in the analysis is to make sure that the given circuit is combinational and not sequential.
2. Assign symbols to all gate outputs that are a function of the input variables. Obtain the Boolean functions for each gate.
3. Label with other arbitrary symbols those gates that are a function of input variables and/or previously labeled gates. Find the Boolean functions for these gates.
4. Repeat step 3 until the outputs of the circuit are obtained.
5. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables only.

- Analysis of the combinational circuit below illustrates the proposed procedure:



We note that the circuit has three binary inputs, *A*, *B*, and C, and two binary outputs, *F1* and *F2*. The outputs of various gates are labeled with intermediate symbols. The outputs of gates that are a function of input variables only are *F2*, *T1 and T2*. The Boolean functions for these three outputs are

$$F2 = AB + AC + BC$$
$$T1 = A + B + C$$
$$T2 = ABC$$

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

$$T3 = F2'T1$$
$$F1 = T3 + T2$$

The output Boolean function *F2* just expressed is already given as a function of the in puts only. To obtain *F1* as a function of *A*, *B*, and C, form a series of substitutions as follows:

$$
\begin{aligned}
F1 &= T3 + T2 \\
&= F2'T1 + ABC \\
&= (AB + AC + BC)'(A+B+C) + ABC \\
&= (AB)'(AC)'(BC)'(A+B+C) + ABC \\
&= (A'+B')(A'+C')(B'+C')(A+B+C) + ABC \\
&= (A'+B'C')(B'+C')(A+B+C) + ABC \\
&= (A'+B'C')(AB'+B'C+AC'+BC') + ABC \\
&= A'B'C + A'BC' + AB'C' + ABC
\end{aligned}
$$

Obtaining truth-table from logic diagram

The derivation of the truth table for the circuit is a straightforward process once the output Boolean functions are known. To obtain the truth table directly from the logic diagram without going through the derivations of the Boolean functions, proceed as follows:

Steps in analysis:

1. Determine the number of input variables to the circuit. For *n* inputs, form the 2n possible input combinations of 1's and 0's by listing the binary numbers from 0 to 2n – 1.
2. Label the outputs of selected gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates that are a function of the input variables only.
4. Proceed to obtain the truth table for the outputs of those gates that are a function of previously defined values until the columns for all outputs are determined.

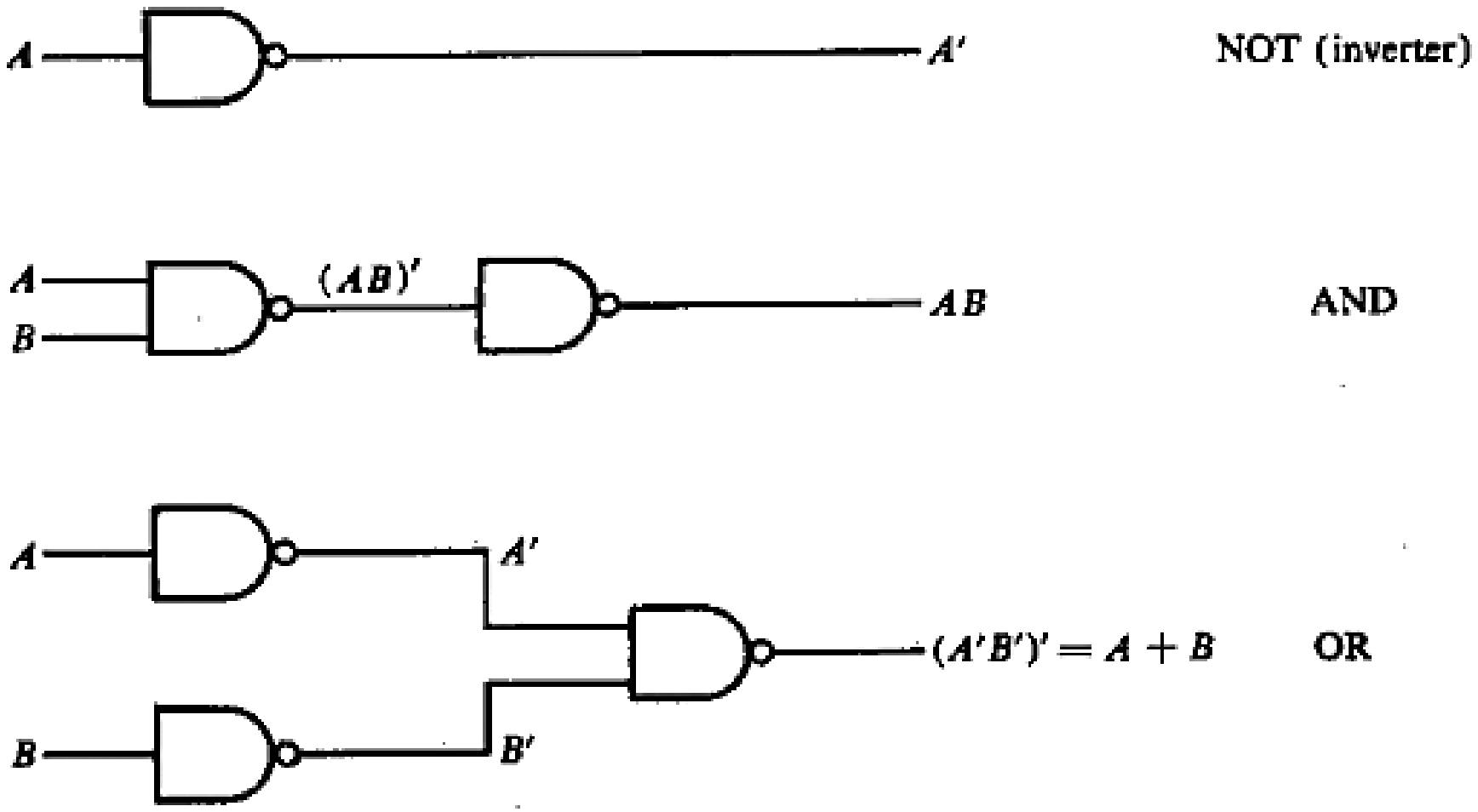
This process can be illustrated using the circuit above:

We form the eight possible combinations for the three input variables. The truth table for F2 is determined directly from the values of A, B, and C, with F2 equal to 1 for any combination that has two or three inputs equal to 1. The truth table for F2’ is the complement of F2.

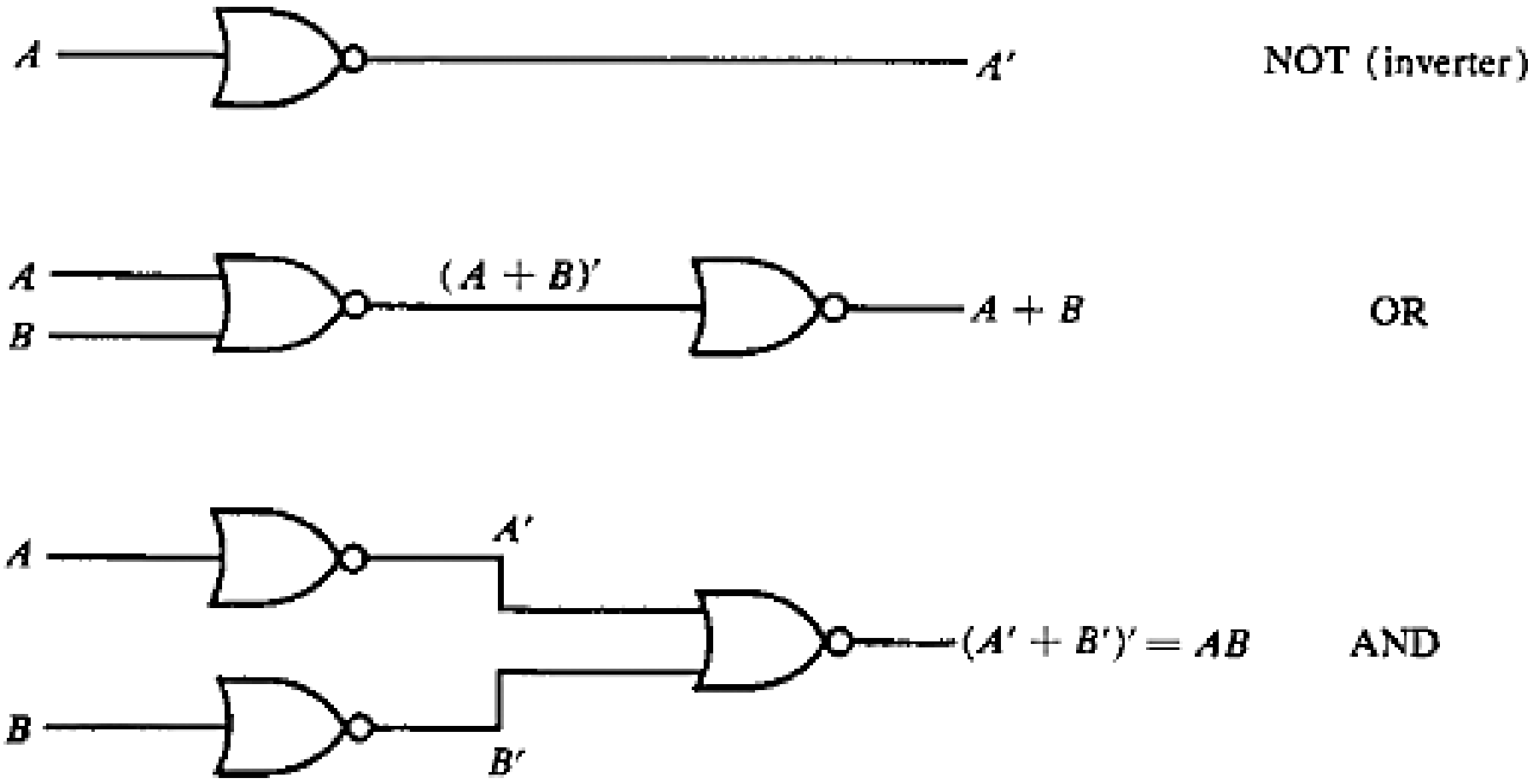
The truth tables for T1 and T2 are the OR and AND functions of the input variables, respectively. The values for T3 are derived from T1 and F2'. T3 is equal to 1 when both T1 and F2' are equal to 1, and to 0 otherwise. Finally, F1 is equal to 1 for those combinations in which either T2 or T3 or both are equal to 1.

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

- **Implementation of NOT, AND and OR gates with NAND gates**



- **Implementation of NOT, AND and OR gates with NOR gates**



- **NAND, NOR and Ex-OR circuits**

In unit 3, SOP and POS form of Boolean functions are studied. Also we got to know, Such Boolean functions can be implemented with 2-level circuits using universal gates (look at NAND and NOR implementation of Boolean function, unit 3). Here we will look at the multiple level circuits employing universal gates i.e we will treat the functions which are in standard form.

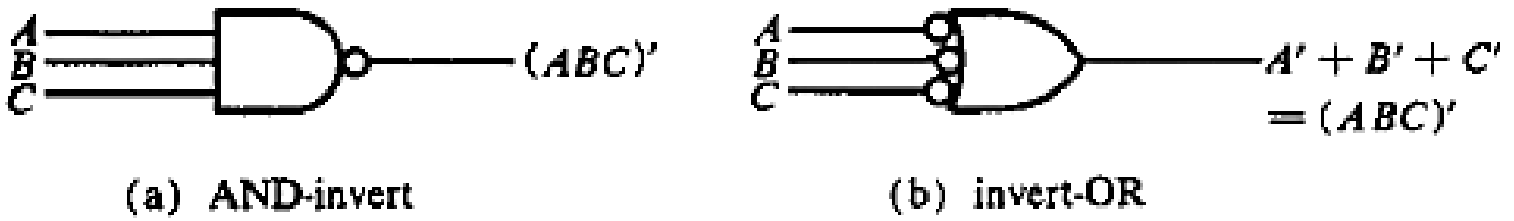


Fig: Graphical Symbol for NAND Gate

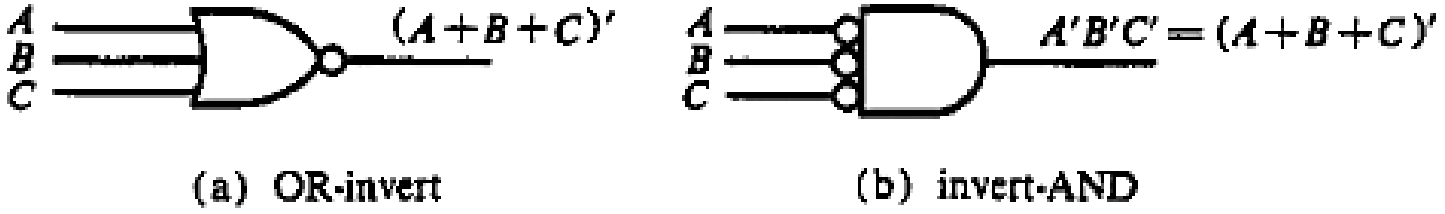


Fig: Graphical Symbol for NOR Gate

A. Multi-level NAND circuits

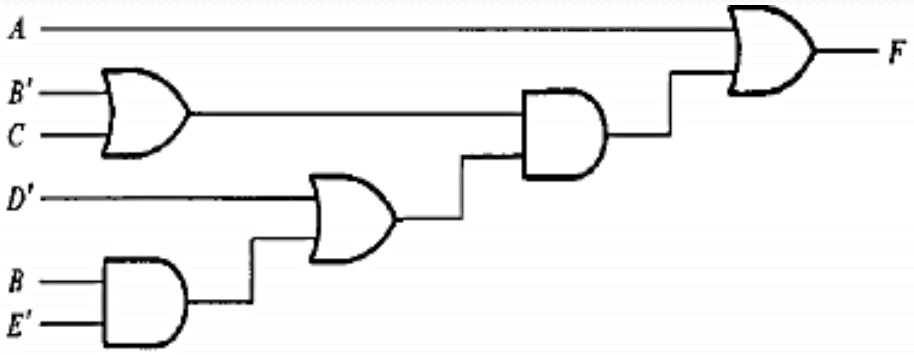
To implement a Boolean function with NAND gates we need to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic. The conversion of an algebraic expression from AND, OR, and complement to NAND can be done by simple circuit-manipulation techniques that change AND-OR diagrams to NAND diagrams.

To obtain a multilevel NAND diagram from a Boolean expression, proceed as follows:

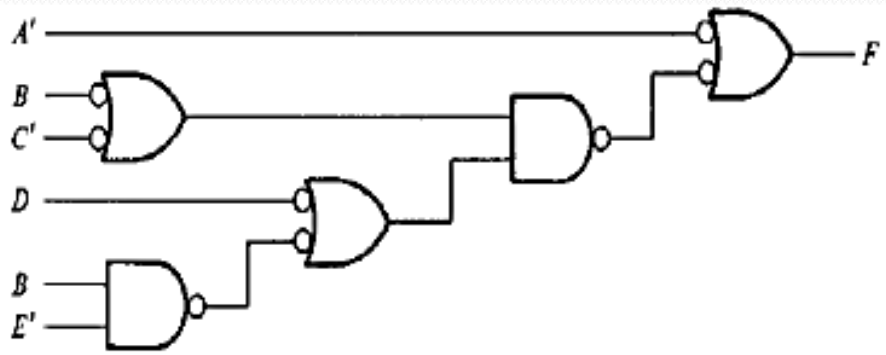
1. From the given Boolean expression, draw the logic diagram with AND, OR, and inverter gates. Assume that both the normal and complement inputs are available.
2. Convert all AND gates to NAND gates with AND-invert graphic symbols.
3. Convert all OR gates to NAND gates with invert-OR graphic symbols.

4. Check all small circles in the diagram. For every small circle that is not compensated by another small circle along the same line, insert an inverter (one-input NAND gate) or complement the input variable.

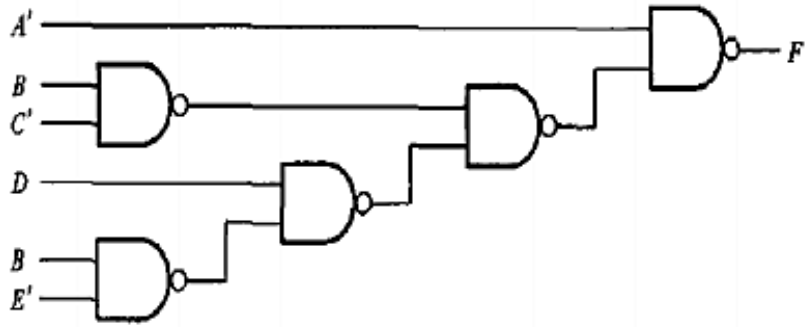
Example: F = A + (B' + C) (D' + BE ')



AND-OR diagram



NAND diagram using two graphic symbols



NAND diagram using one graphic symbol

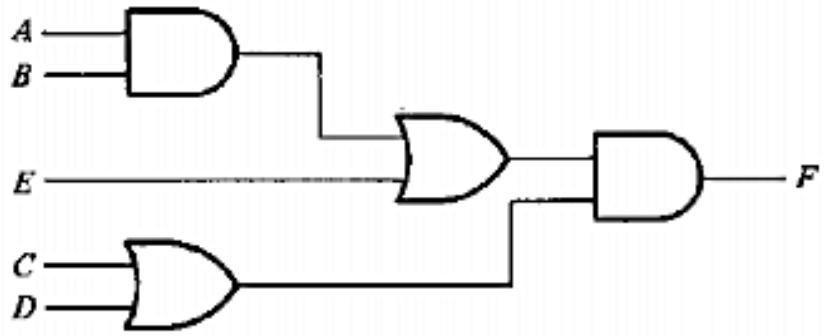
B. Multi-level NOR circuits

The NOR function is the dual of the NAND function. For this reason, all procedures and rules for NOR logic form a dual of the corresponding procedures and rules developed for NAND logic. Similar to NAND, NOR has also two graphic symbols: OR-invert and invert-AND symbol.

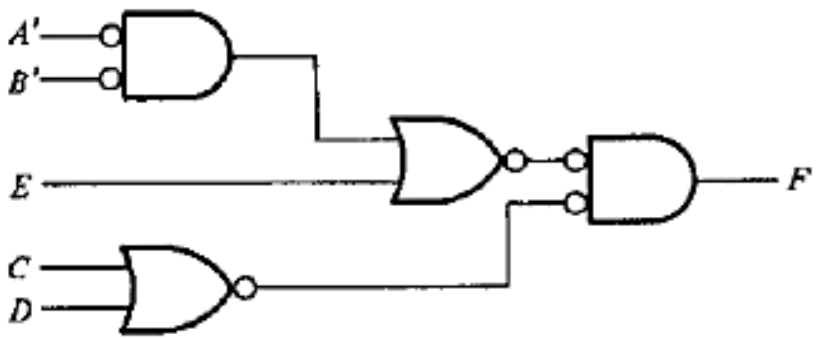
The procedure for implementing a Boolean function with NOR gates is similar to the procedure outlined in the previous section for NAND gates:

1. Draw the AND-OR logic diagram from the given algebraic expression. Assume that both the normal and complement inputs are available.
2. Convert all OR gates to NOR gates with OR-invert graphic symbols.
3. Convert all AND gates to NOR gates with invert-AND graphic symbols.
4. Any small circle that is not compensated by another small circle along the same line needs an inverter or the complementation of the input variable.

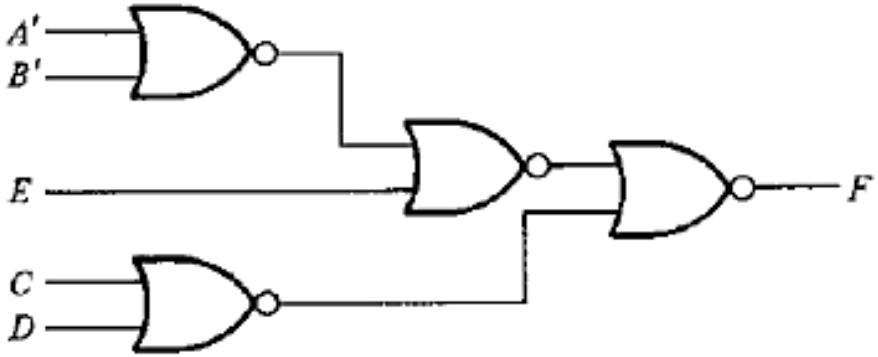
Example: F = (AB + E) (C + D)



AND-OR diagram



NOR diagram



Alternate NOR diagram

- **Ex-OR function**

The **exclusive-OR (XOR)** denoted by the symbol $\oplus$, is a logical operation that performs the following Boolean operation:

$$x \oplus y = xy' + x'y = (x+y)(x'+y')$$

i.e.

It is equal to 1 if only *x* is equal to 1 or if only y is equal to 1 but not when both are equal.

the exclusive-OR operation is both commutative and associative.

$$A \oplus B = B \oplus A$$

$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$$

Exclusive-NOR, also known as equivalence, performs the following Boolean operation:

$$(x \oplus y)' = ((x+y)(x'+y'))' = (x+y)' + (x'+y')' = x'y' + xy$$

i.e.

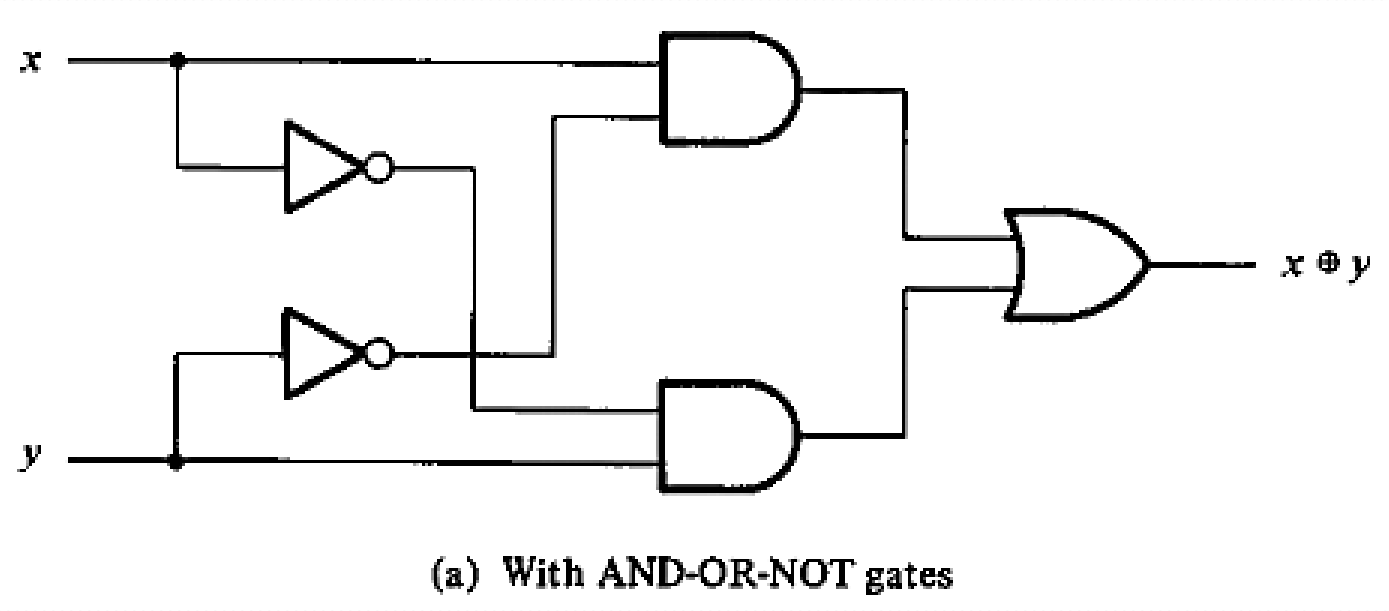
It is equal to 1 if both x and y are equal to 1 or if both are equal to 0. The exclusive-NOR can be shown to be the complement of the exclusive-OR by means of a truth table or by algebraic manipulation.

$$(x \oplus y)' = (xy' + x'y)' = (x' + y)(x + y') = xy + x'y \text{ ‘}$$

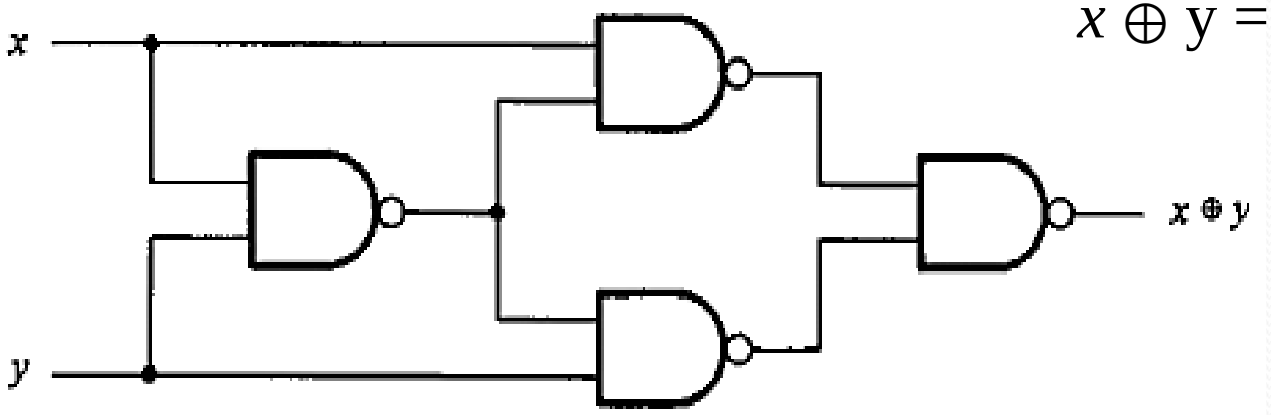
Note: $(x \oplus y)' = ((x+y)(x'+y'))' = (x+y)' + (x'+y')' =$

Realization of XOR using Basic gates and universal gates

A two-input exclusive-OR function is constructed with conventional gates using two inverters, two AND gates, and an OR gate and next figure shows the implementation of the exclusive-OR with four NAND gates.



(a) With AND-OR-NOT gates



(b) With NAND gates

$x \oplus y = xy' + x'y$

$= (x+y)(x'+y')$

$=(x+y)(xy)'$

$=x(xy)'+y(xy)'$

$=[\{x(xy)'+y(xy)'\}']'$

$=[\{x(xy)'\}'\{y(xy)'\}']'$

- *Similarly realize X-OR gate with NOR gate also.*
- *Also realize X-NOR gate with NAND and NOR gate.*

X-NOR using NOR:

$(x \oplus y)' = xy + x'y' = xy + (x+y)'$

$=(x+(x+y)')(y+(x+y)')$

$=[(x+(x+y)')(y+(x+y)')]''$

$=[\{(x+(x+y)')\}'+\{(y+(x+y)')\}']'$

Parity generator and Checker

Exclusive-OR functions are very useful in systems requiring error-detection and correction codes. As discussed before, a **parity bit** is used for the purpose of detecting errors during transmission of binary information. A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even. The message, including the parity bit, is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted.

- The circuit that generates the parity bit in the transmitter is called a ***parity generator*.**
- The circuit that checks the parity in the receiver is called a ***parity checker.***

Example: Consider a 3-bit message to be transmitted together with an even parity bit.

The three bits, *x*, *y*, and z, constitute the message and are the inputs to the circuit. The parity bit *P* is the output. For even parity, the bit *P* must be generated to make the total number of 1's even (including *P).*

Three-Bit Message			Parity Bit
x	*y*	*z*	*P*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Table: Even parity generator truth table

Note: From the truth table, we see that *P* constitutes an odd function because it is equal to 1 for those minterms whose numerical values have an odd number of 1's.

Therefore, *P* can be expressed as a three variable exclusive-OR function: $P = x \oplus y \oplus z$

Or,

From K-Map,

$$P = x'y'z + x'yz' + xyz + xy'z'$$
$$= x'y'z + xyz + x'yz' + xy'z'$$
$$= (x'y' + xy)z + (x'y + xy')z'$$
$$= (x \odot y)z + (x \oplus y)z'$$
$$= (x \oplus y)'z + (x \oplus y)z'$$
$$= (x \oplus y) \oplus z$$
$$= (x \oplus y \oplus z)$$

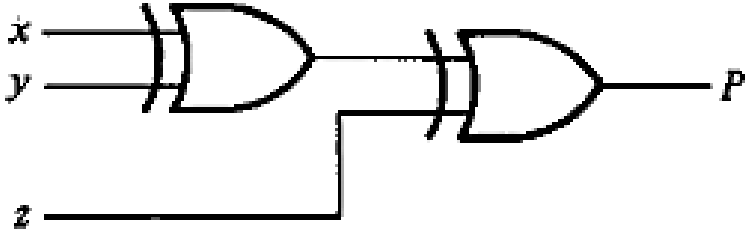
The three bits in the message together with the parity bit are transmitted to their destination, where they are applied to a parity-checker circuit to check for possible errors in the transmission.

Four Bits Received				Parity Error Check
x	*y*	*z*	*P*	*C*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

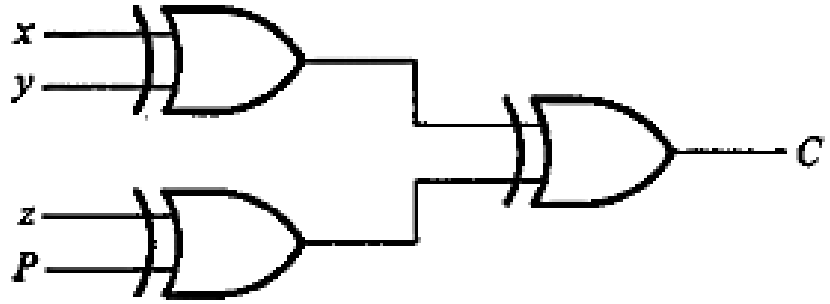
Table: Even parity checker truth table

- Since the information was transmitted with even parity, the four bits received must have an even number of 1's. An error occurs during the transmission if the four bits received have an odd number of 1's, indicating that one bit has changed in value during transmission.
- The output of the parity checker, denoted by C, will be equal to 1 if an error occurs, that is, if the four bits received have an odd number of 1's.
- The parity checker can be implemented with exclusive-OR gates: $C = x \oplus y \oplus z \oplus P$.

Logic diagrams for parity generator and Parity checker are shown below:

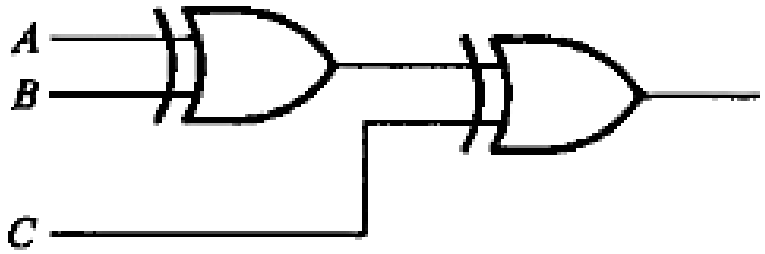


(a) 3-bit even parity generator

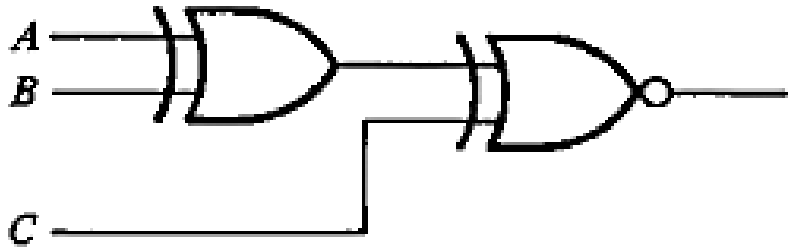


(b) 4-bit even parity checker

Note:



(a) 3-input odd function



(b) 3-input even function


Thank You!