CSE 4205
Digital Logic Design

Combinational Logic

Course Teacher: Md. Hamjajul Ashmafee

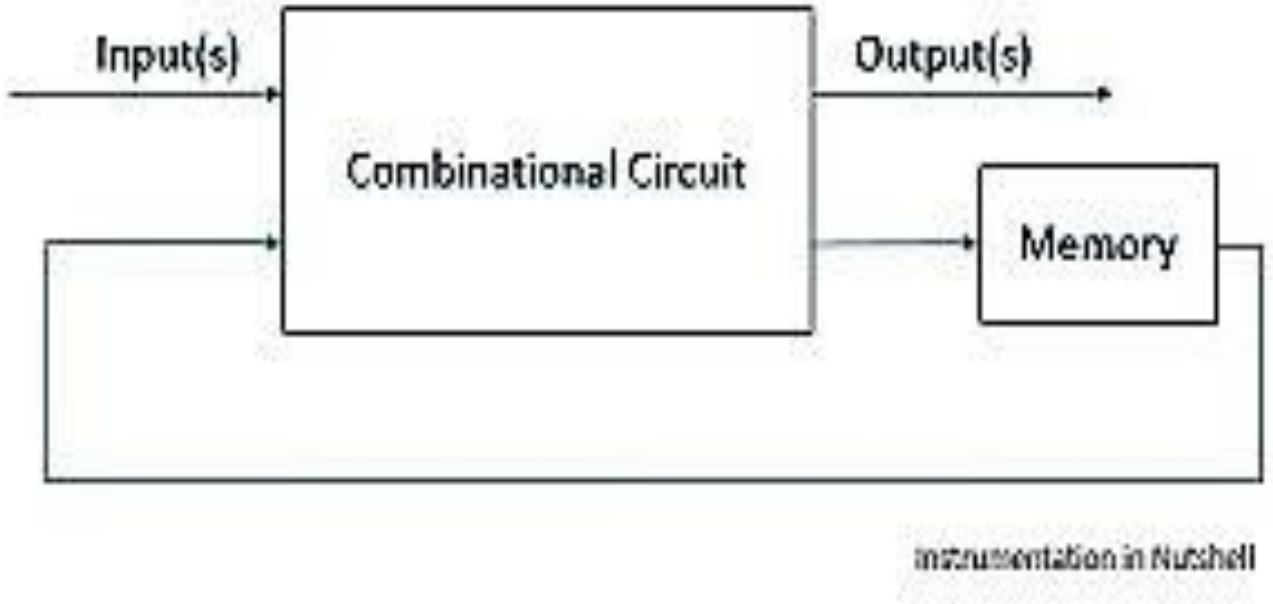
Lecturer, CSE, IUT

Email: ashmafee@iut-dhaka.edu

Introduction

- Logic circuits of any digital system
  - ❖Combination Logic Circuit
  - ❖Sequential Logic Circuit
- A **combinational circuit** is the circuit whose outputs at any time are determined directly from the present combination of inputs **without** regard to previous inputs or outputs.
- On the other hand, **Sequential circuits** employ memory elements (binary cells) in addition to the logic gates.
  - Their **outputs** are the function of the present inputs and the state of the memory elements. Its behavior must be **time sequence specified**.

Introduction



Introduction…

We have already learned that

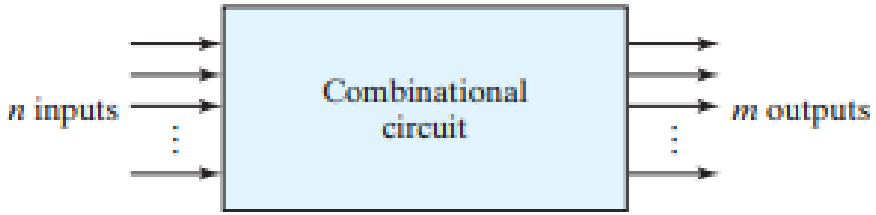
- Binary numbers and Binary codes are representation of discrete quantities of information.
- These binary variables are represented by electric voltages or by some other signal.
- This signal can be manipulated in digital logic gates to perform required functions.
- Boolean algebra is used to express Binary logic functions
- Several methods (K-map, tabular method,…) can be used to simplify the Boolean functions

*In this chapter, we will learn about how to formulate various **systematic design** and analysis the **procedures** of combinational circuits.*

Introduction…

Combinational Circuits consists of –

- **Input variables (1/0)** – n input variables from external sources – primed and unprimed
- **Logic gates**
- **Output variables (1/0)** – m output variables to external destinations



- For n input variables, there are 2^n possible combinations of binary input values – for each of them there is **only one** possible output combination.
- For m output variables, a combinational circuit can be described by m Boolean functions – **one for each output variab**les.

Design Procedure

Design of a combinational ***circuit starts from a problem statement*** *and* ***ends in a logic circuit diagram****.*

The procedure involves:

- The **problem** is stated
- The **number** of available input variables and required output variables is determined
- The input and output variables are assigned **letter symbols**
- The **truth table** that defines the required relationships between inputs and outputs are derived
- The **simplified Boolean function** for each output is obtained
- The **logic diagram** is drawn

Design Procedure…

- The 1s and 0s in the input columns are obtained from the 2^n binary **combinations** available for n input variables
- The binary values for the outputs are determined from **examination of the stated problem**
- The specifications may indicate that some input combinations will not occur – **don't care conditions**
- Truth table – exact definition of the combinational circuit
- Output Boolean Function from the truth table – **simplified by any simplification method** like k-map, tabular method, algebraic manipulation

Design Procedure…

- A practical design method considers such constraints:
  1. Minimum number of **gates**
  2. Minimum number of **inputs to a gate**
  3. Minimum **propagation time of the signal** through the circuit
  4. Minimum number of **interconnections**
  5. Limitations of the **driving capabilities of each gate**
- Based on the application, those constraints will be prioritized.
- A **logic diagram** is useful to visualize the gate implementation of the expression

Adder

- A digital computer performs a variety of information processing tasks amongst them most common are arithmetic operations
- The most basic arithmetic operation – addition of two binary digits – 4 possible elementary operations

a	b	sum
0	0	0
1	0	1
0	1	1
1	1	10

Adder…

- 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*****.**
- **Two half adders** can be employed to implement **a full adder**.

Half Adder

- Two binary inputs (augend and addend) and two binary outputs (sum and carry)
- Determination the number of variables – two inputs and two outputs
- Assignment of symbols – two inputs [x & y] and two outputs [S & C]
- S represents the LSB and C represents the MSB of the sum
- Truth table to identify the function of the half adder

Half Adder

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

Half Adder…

- Simplified Boolean functions for two outputs:

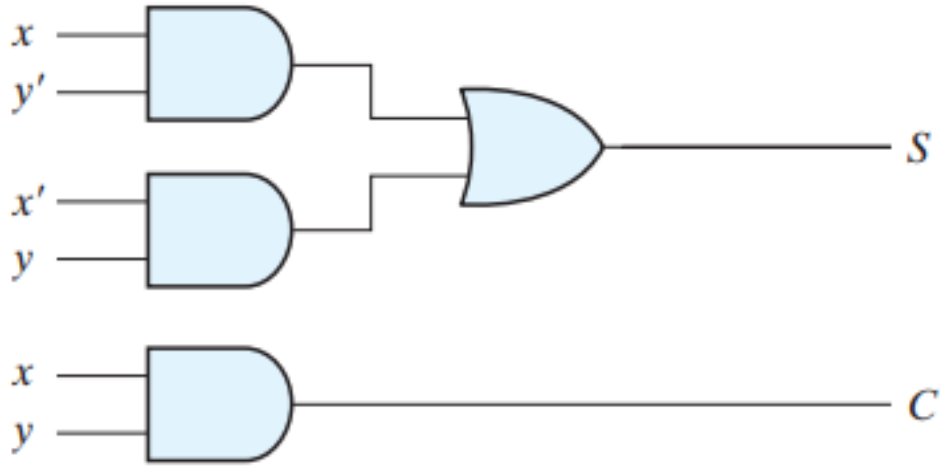
$$S = x'y + xy'$$
$$C = xy$$

It also can be written as:
S = (x+y)(x'+y') = (x'y'+xy)
C = (x'+y')'

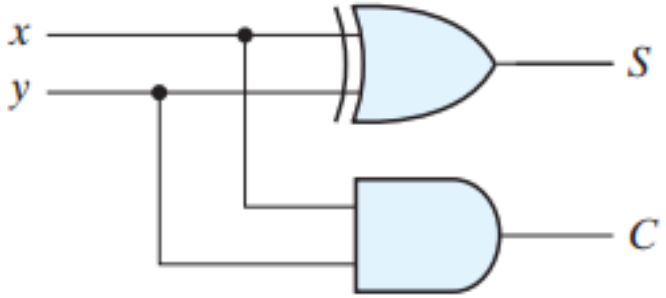
- To be flexible, there are more available implementations.
- To be noted :
  - XOR of x and y = (x+y)(x'+y')
  - Equivalence of x and y = (xy+x'y')
  - De Morgan's Law: xy = (x'+y')'

Half Adder…

Half Adder Implementation



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



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

Full Adder

- A **combinational circuit** that forms the **arithmetic sum** of **three input bits**
- Three inputs (x, y and z) and two outputs (S and C)
- x and y are **two significant bits to be added**
- z represents the **carry** from the **previous lower significant position**
- Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3
- The two outputs are designated by the symbols *S* for **sum** and *C* for **carry**.
- *S* is the least significant bit of the sum and *C* is the most significant bit

Full Adder…

- The truth table of the full adder:

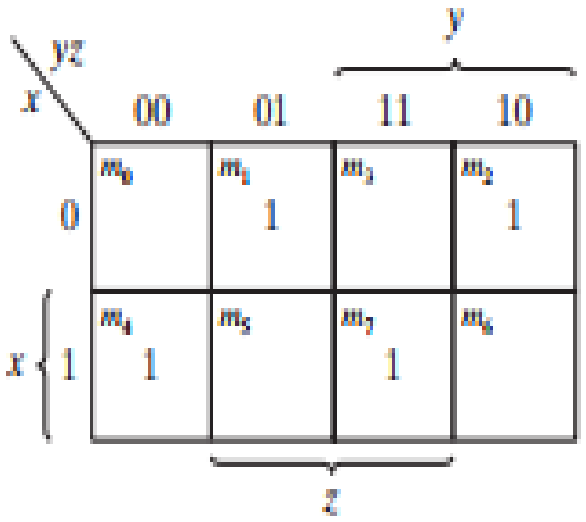
Full Adder

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

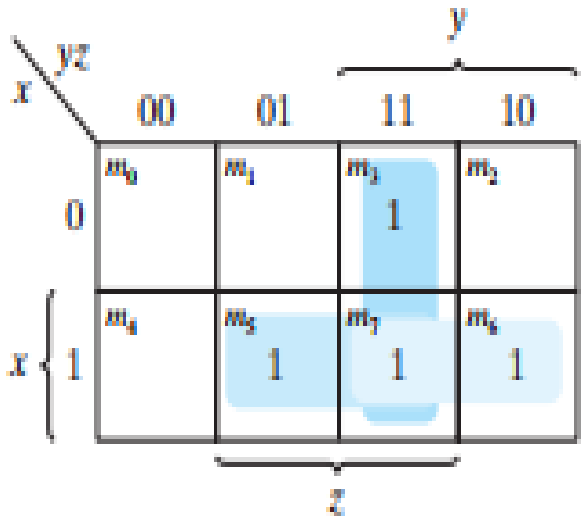
- The input-output logical relationship can be expressed in two Boolean functions – two unique maps are required for each to simplify

Full Adder…

Maps for full adder to simplify



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

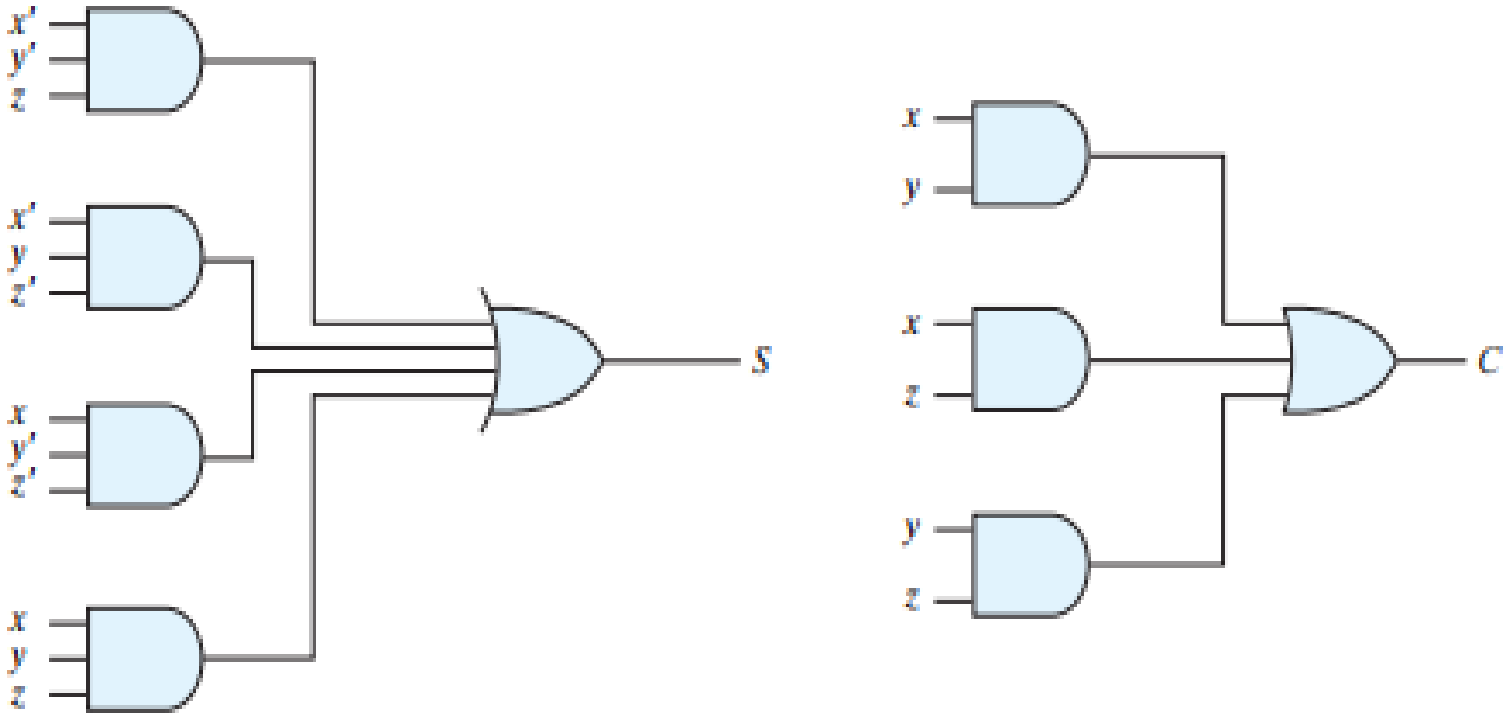


(b) $C = xy + xz + yz$

In SOP form

Full Adder…

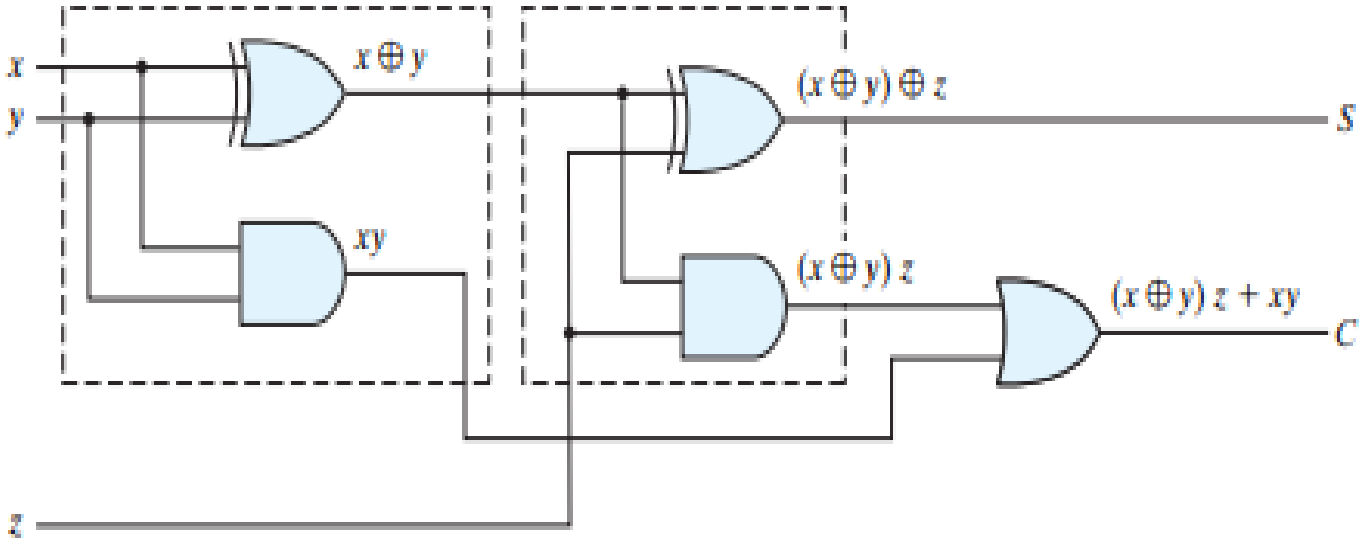
Implementation of full adder



In SOP form

Full Adder…

Full Adder with two Half Adders and OR gate



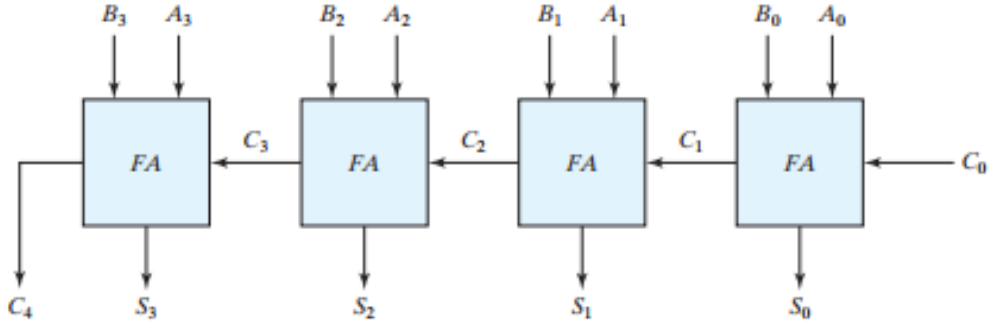
Full Adder…

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

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

Binary Adder

- **Binary adder** – produces the arithmetic sum of **two binary numbers**
- Addition of *n*-bit numbers requires a chain of *n* **full adders** or a chain of one **half adder** and (*n* – 1) **full adders** in cascade.
- The input carry C_0 in the least significant position must be 0.
- If we follow the classical method, it would require a truth table with 2^9=512 entries. By using an iterative method of cascading a standard function, it is possible to obtain a simple and straightforward implementation.



Subtractors

- The subtraction of two binary numbers – *taking the complement of the subtrahend* and adding it to the minuend
- So subtraction is a kind of addition operation – needs **full adder** for machine implementation
- If the minuend bit is smaller than the subtrahend bit, a 1 is borrowed from the next significant position – *also must be conveyed to the next higher pair of bits by means of a binary signal from current stage to next higher stage*
- Two kind of subtractor based on the number of inputs P:
  - Half-subtractor
  - Full subtractor

Half-Subtractor

- A combinational circuit – subtracts two bits and produces their differences and the output that specifies if a 1 is borrowed or not to next stage.
- Designation of the input variables – minuend (x) and subtrahend (y)
- *D = 2B + x – y, [if x<y, B=1]*

Inputs		Outputs	
X	Y	D	B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

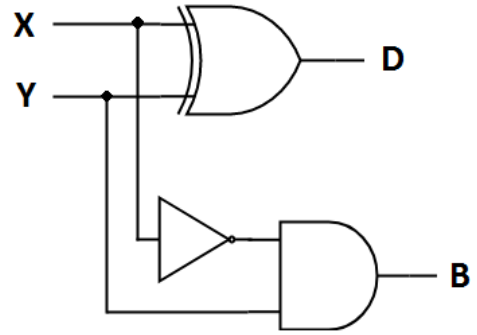
Half-Subtractor…

- Boolean functions of two outputs of the half-subtractor:

$$D = x'y+xy'$$

$$B = x'y$$

- Logic for D is same as the logic for S in half-adder
- Circuit Implementation of half-subtractor:



Full-Subtractor

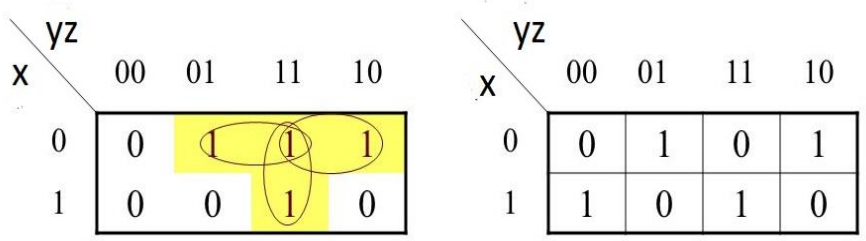
- A combinational circuit – performs a subtraction between two bits considering the previous borrow – three inputs and two outputs
- Designation of variables – inputs [minuend(x), subtrahend(y), previous borrow(z)] and outputs [difference(D) and current borrow (B)]
- Considering *2B+x-y-z,* if z=0, it will be same as half-subtractor
- If (x<y+z),B=1

Full-Subtractor…

- Truth table of the full-subtractor:

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

- K-map for full-subtractor to simplify:



For Borrow For Difference

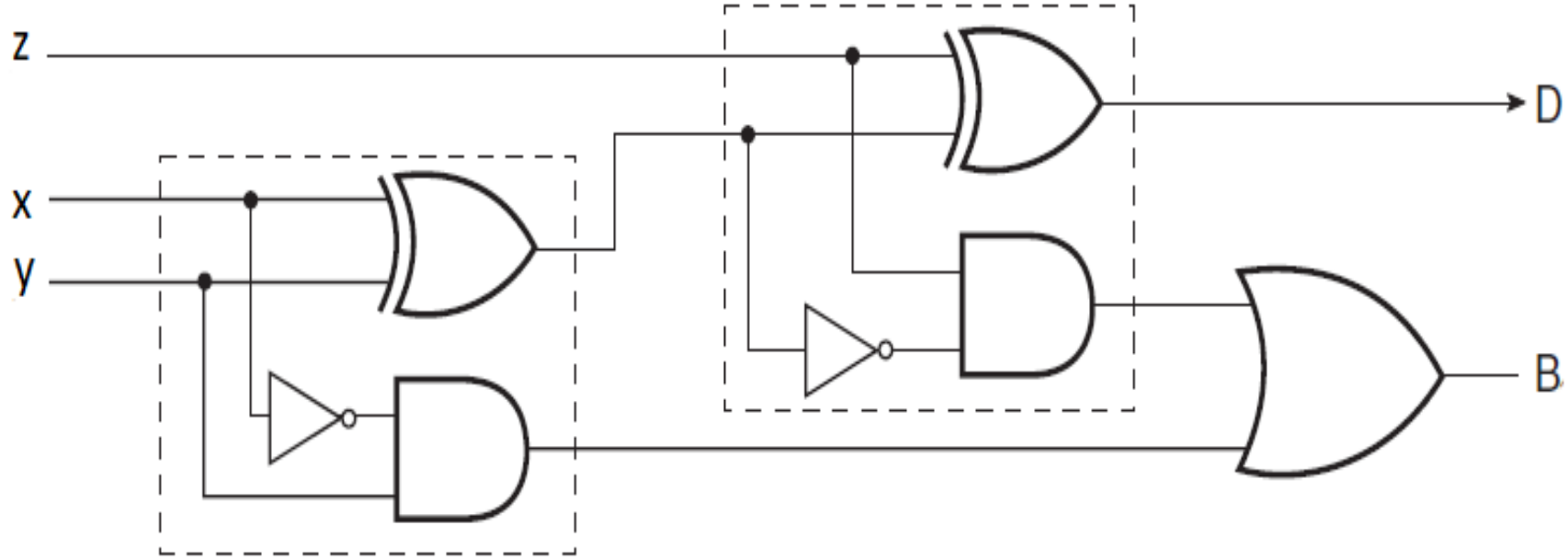
Full-Subtractor…

- The simplified Boolean function of full-subtractor:

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

- The output of **B** resembles the function for **C** in the full adder – **except x is complemented**
- The output of **D** is exactly same as the output **S** in the full adder
- Because of these similarities, it is possible to convert a full adder into a full subtractor - ???

Full-Subtractor with Half-Subtractors



Code Conversion

- Sometimes **same informati**on used in **different codes** by different digital systems – output of one system used as input to another
- **A conversion circuit** – between these two systems for this information flow.
- Code converter – makes these two systems compatible even though their code is different
- **To convert from code A to code B** – input lines supply the bit combination specified by A and output lines generates the bit combination specified by B

Example – BCD to Excess-3

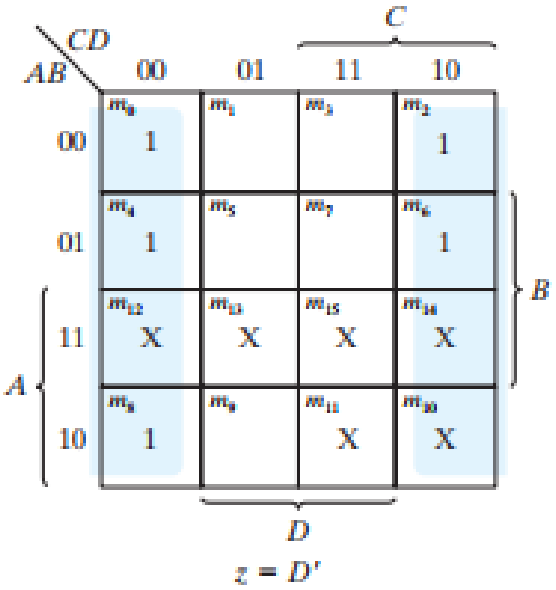
- Since each code uses four bits to represent a decimal digit – there are **four input variables (A,B,C,D)** and **four output variables (w,x,y,z)**
- The truth table relating the input and output variables:

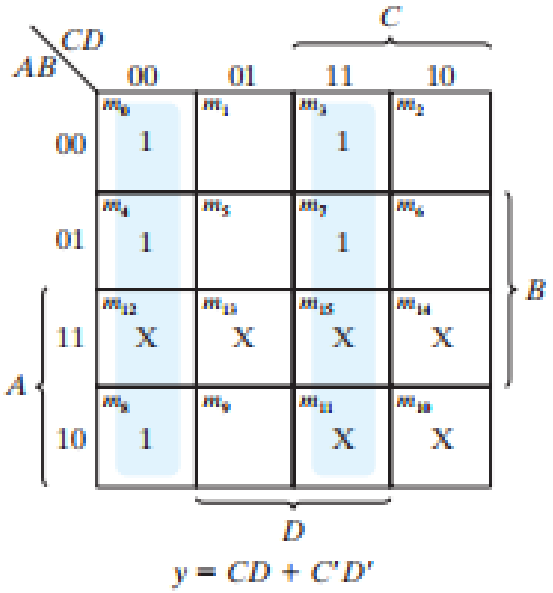
Truth Table for Code Conversion Example

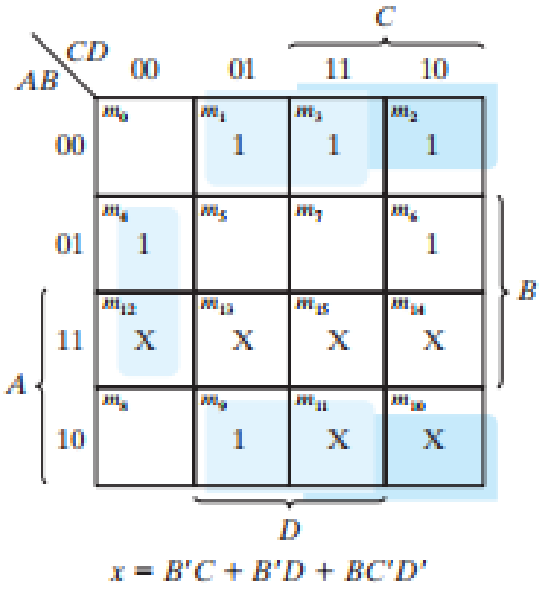
Input BCD				Output Excess-3 Code			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

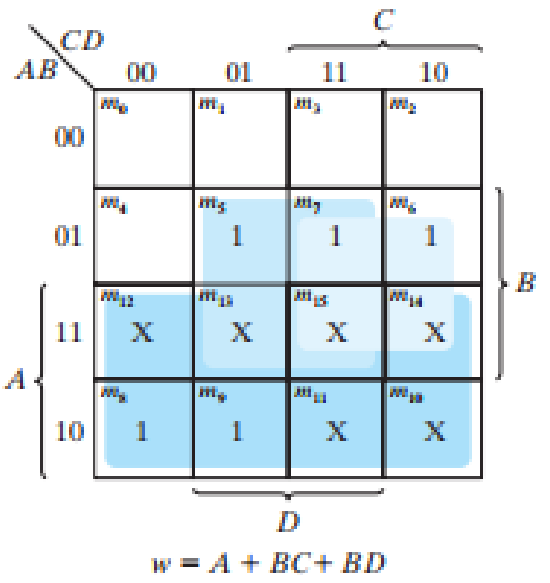
Example – BCD to Excess-3…

- The six bit combinations **not listed** for the inputs – **don't care combinations**
- **The maps** for simplification of the output functions – each of them are functions of four input variables









Example – BCD to Excess-3…

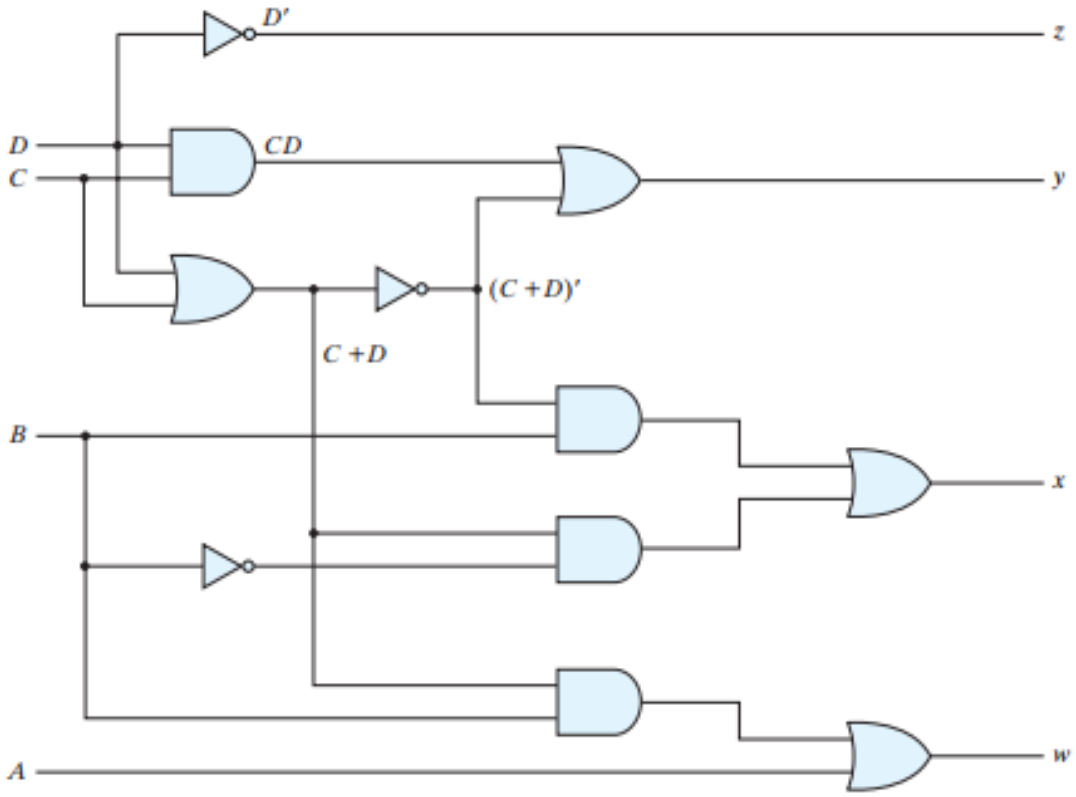
- There are various possibilities for the logic diagram for this circuit – because of having don't care conditions
- One of the possible solutions with multiple representations:

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

- Here we can see that the term *(C+D)* has been obtained in multiple outputs

Example – BCD to Excess-3…

- Logic diagram for BCD to Excess-3 Code Converter:

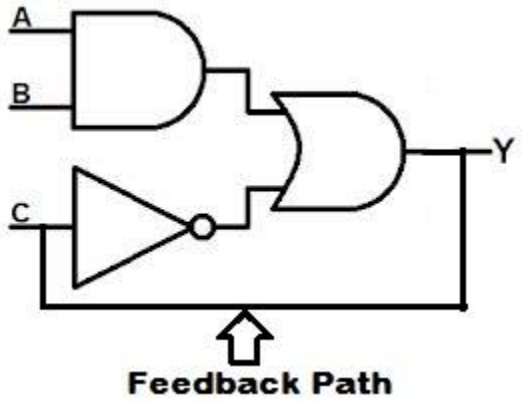


Analysis Procedure

- The **design** of a combinational circuit starts from the problem statement and its required specifications (or verbal explanation) and ends with a set of output Boolean functions or a logic diagram
- But the **analysis** of a combinational circuit is the reverse process – it starts with a logic diagram and ends with a set of Boolean functions, a truth table, or a verbal explanation of the circuit operation
- First step in this analysis – check the circuit – combinational or sequential

Analysis Procedure…

- Combinational logic – no feed back path/ no memory element
- Feedback path – a connection from the output from one gate to input of second gate **that forms part** of the input to the first gate

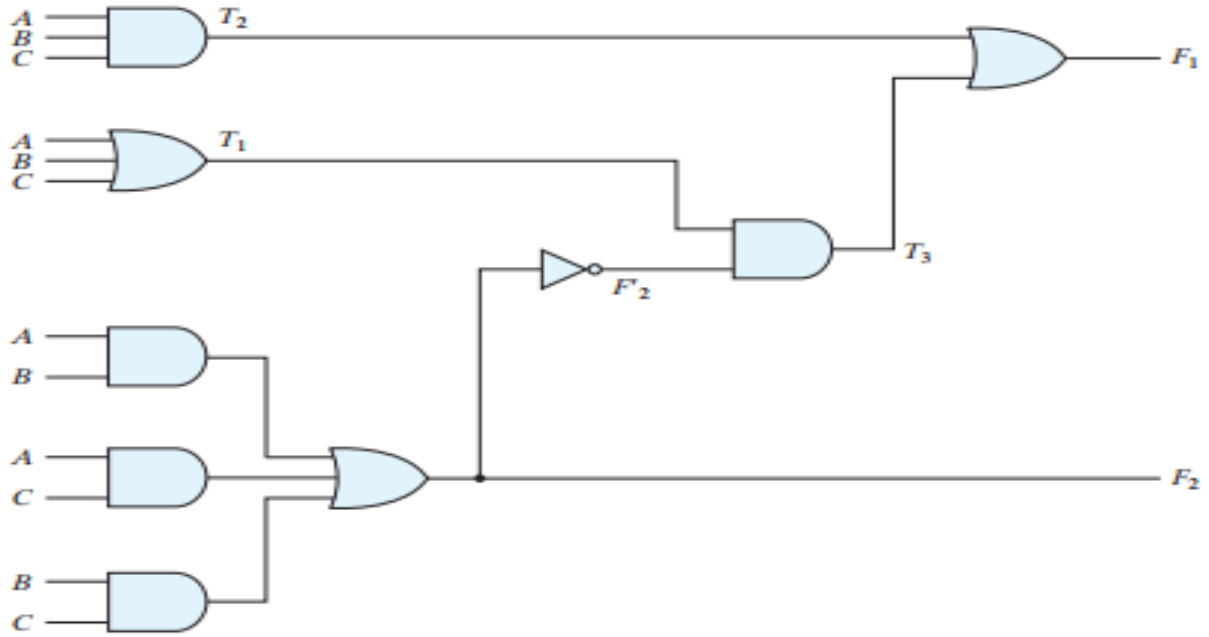


Analysis Procedure…

- After verification of the circuit as combinational logic, we proceed to obtain the output Boolean functions or truth table (based on specification)
- *Steps we follow to proceed:*
  - **Label** with arbitrary symbols **all gate outputs** that are a function of the **input variables**. Obtain the Boolean functions for each gate.
  - Label with other arbitrary symbols those gates which are a function of input variables and/or previously labeled gates. Find the Boolean functions for those gates.
  - **Repeat** the process outlined in **step 2** until the outputs of the circuit are obtained
  - By repeated substitution of previously defined functions, **obtain the output Boolean functions in terms of input variables only.**

Analysis Procedure…

Analysis the following combinational circuit:



Analysis Procedure…

- The circuit has **3 binary inputs** – *A,B,* and *C* and **2 binary outputs** – F_1 and F_2
- The outputs of various gates are labeled with **intermediate symbols**
- The outputs of gates that are a function only of input variables are T_1 and T_2.
- Output F_2 can easily be derived from the input variables.

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

Analysis Procedure…

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

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

- To obtain F_1 as a function of A, B, and C, we form a series of substitutions as follows:

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

Analysis Procedure…

- To pursue the investigation and determine the information transformation task achieved by the circuit – truth table from the Boolean function.
- For example: this circuit is a full adder with F1 being the ***sum*** and F2 being the ***carry*** outputs. A,B and C are three inputs to be added.
- The derivation of the truth table for the circuit is a straightforward process when output Boolean functions are known

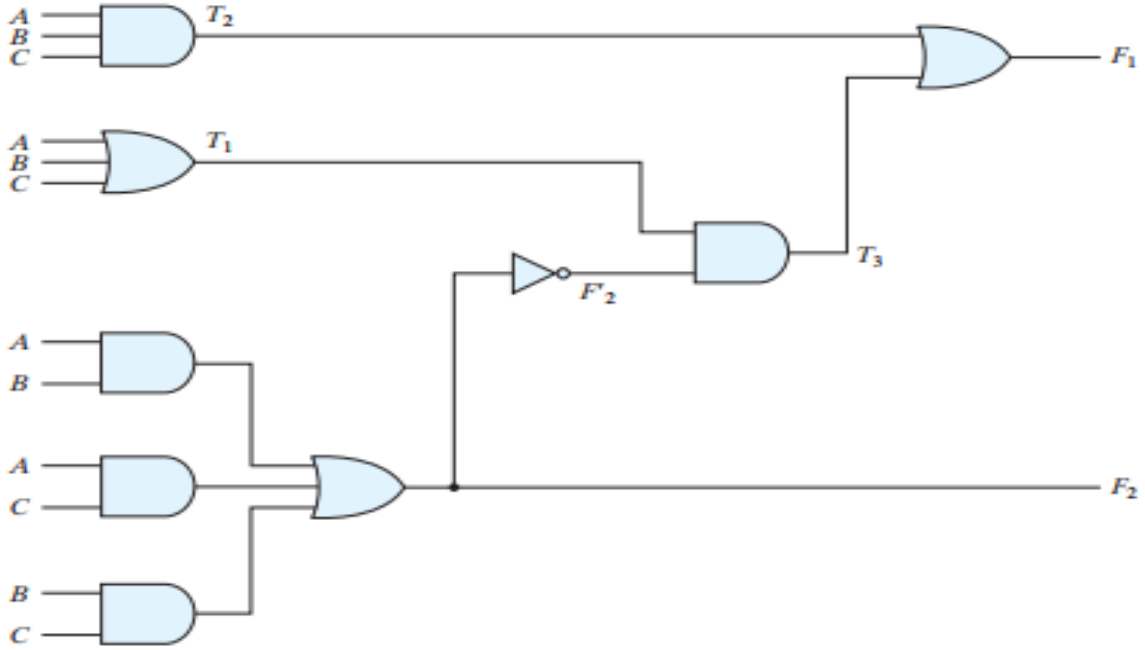
Analysis Procedure…

To obtain the **truth table** directly from **the logic diagram** without going through the derivations:

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

Analysis Procedure…

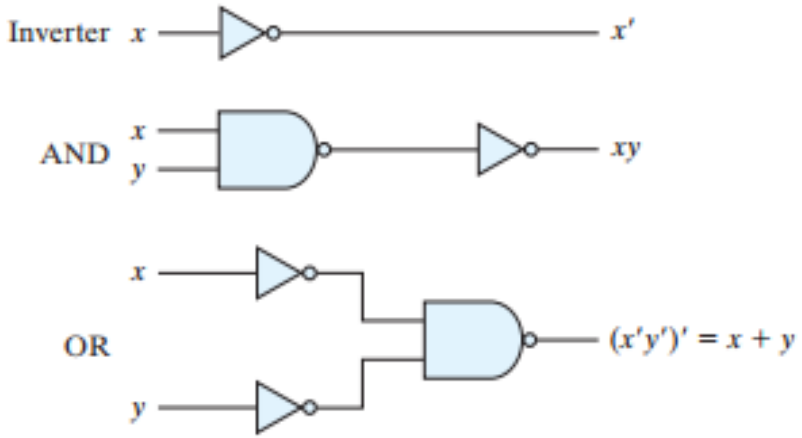
Truth table derivation directly from the logic diagram.

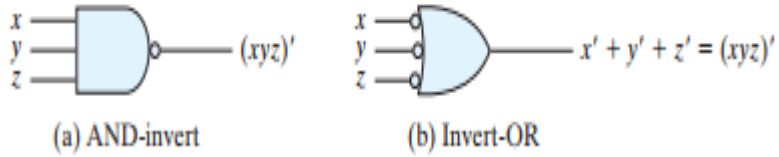


A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Combinational Circuit – NAND Implementation

- NAND – universal gate
- Both combination and sequential circuit can be constructed with NAND gates



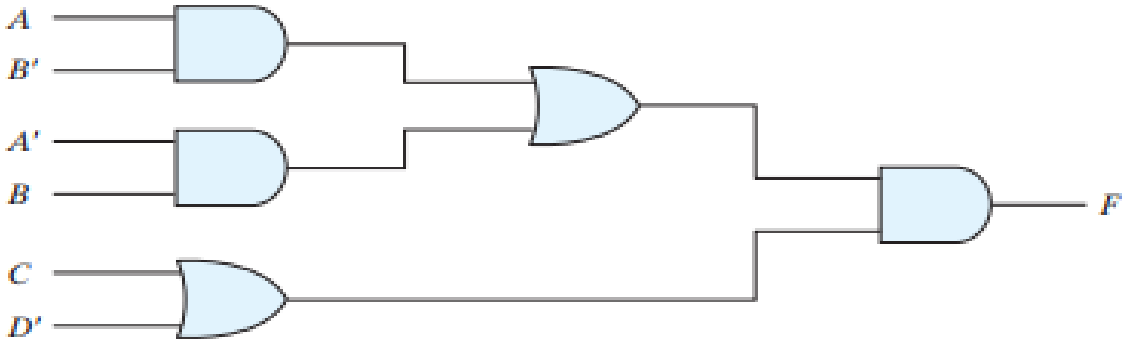


NAND gate

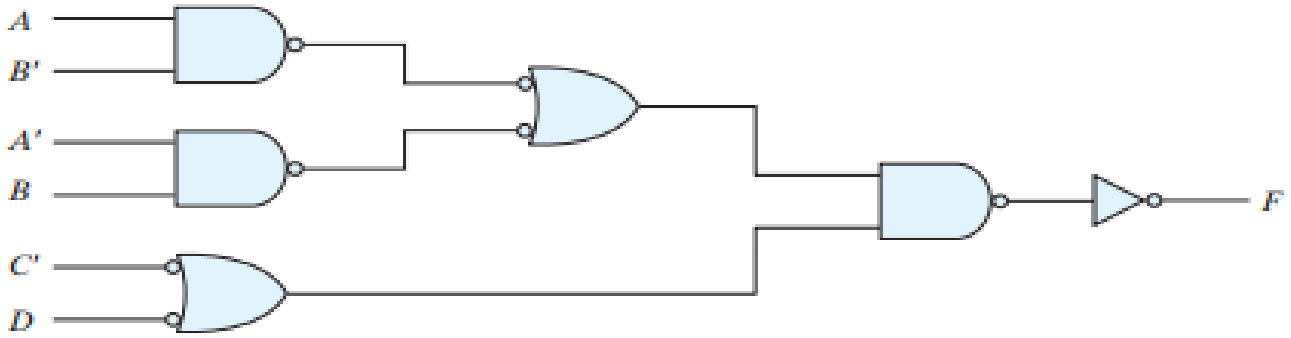
NAND Implementation…

- It can be done by *obtaining* the simplified Boolean functions in terms of AND, OR, NOT gates and *converting* the functions to NAND logic
- Procedure of implementation of Boolean functions with NAND gates: **Block Diagram Method**
  1. From the given algebraic expression, **draw the logic diagram with AND, OR and NOT gates**. Assume that both the normal and complement inputs are available.
  2. Draw the **second logic diagram** with the **equivalent NAND logic** substituted for each AND, OR and NOT gate
  3. Remove any two cascaded inverts from the diagram, since double inversion doesn't perform a logic function. Remove inverters connected to single external inputs and complement the corresponding input variable.

NAND Implementation…



(a) AND–OR gates



(b) NAND gates

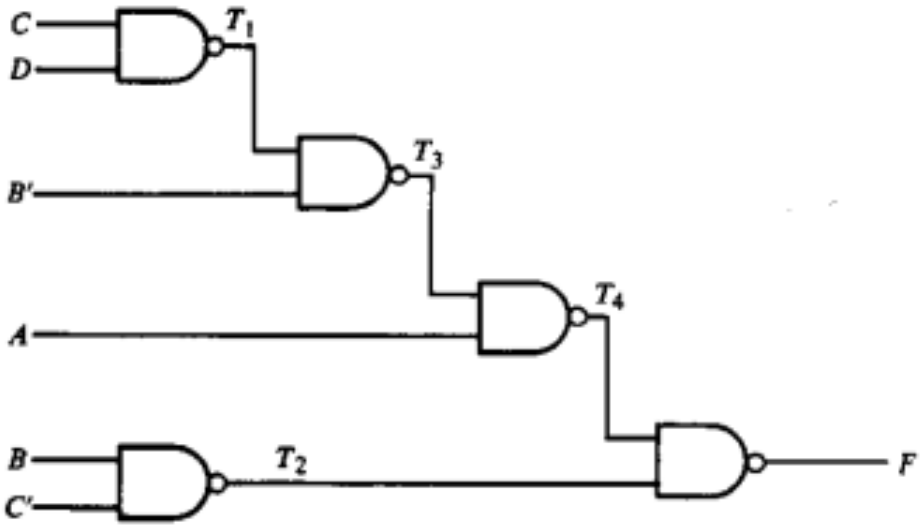
In general, the number of NAND gates required to implement a function is equal to the number of AND-OR gates, except for an occasional inverter. It will be applicable when both normal and complement inputs are available.

NAND Implementation: Analysis Procedure

- Reverse process – **starts** from given NAND logic diagram and **ends** with a Boolean expression or truth table
- Specialty – Application of De Morgan's Law
- **Derivation of Boolean Function from NAND Logic Diagram**
  - All gates' outputs are **labeled with arbitrary** symbols
  - The **Boolean functions for the outputs** of gates that receive only external inputs are derived. It may follow **De Morgan's law** to make it convenient to use
  - **Boolean output functions of gates** which have inputs from previously derived functions are determined in consecutive order **until the output is expressed in terms of input variables**

NAND Imp.: Analysis Procedure …

Example:



$$T_1 = (CD)' = C' + D'$$

$$T_2 = (BC')' = B' + C$$

$$T_3 = (B'T_1)' = (B'C' + B'D')'$$

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

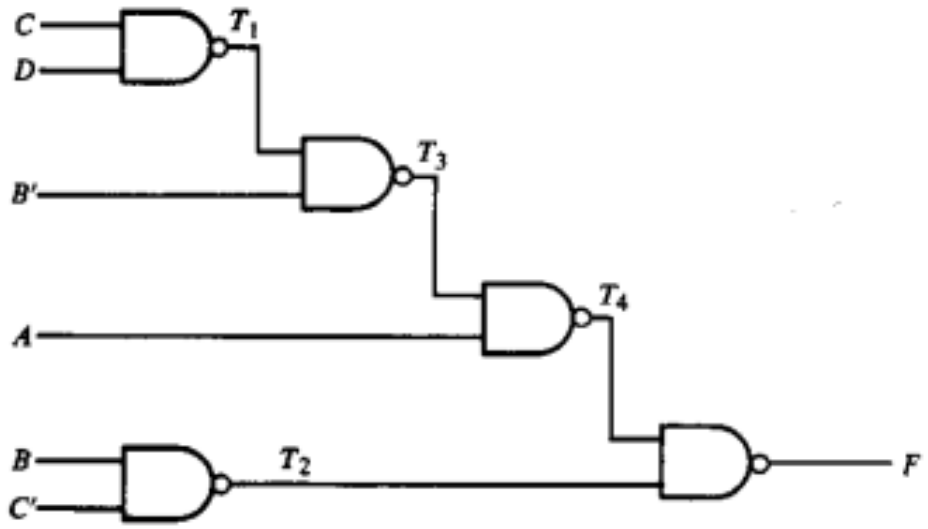
$$T_4 = (AT_3)' = [A(B + CD)]'$$

$$F = (T_2T_4)' = \{(BC')'[A(B + CD)]'\}'$$

$$= BC' + A(B + CD)$$

NAND Imp.: Derivation of Truth Table

Same as before.

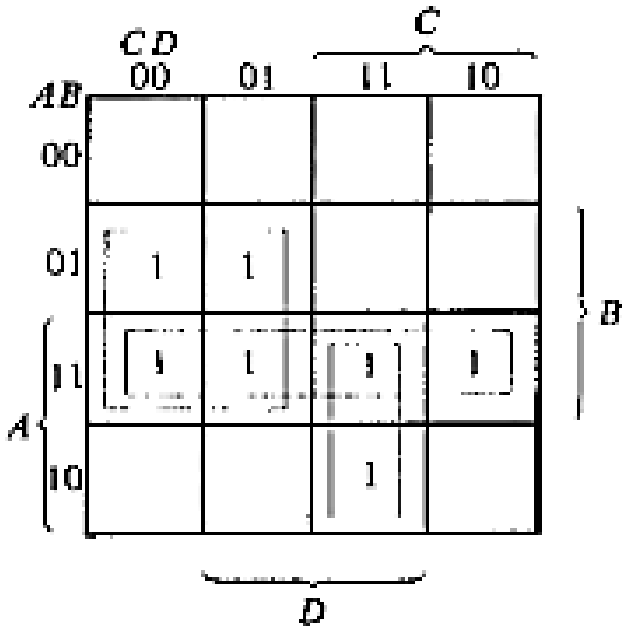


Truth Table for the Circuit of Figure 4-14

A	B	C	D	T_1	T_2	T_3	T_4	F
0	0	0	0	1	1	0	1	0
0	0	0	1	1	1	0	1	0
0	0	1	0	1	1	0	1	0
0	0	1	1	0	1	1	1	0
0	1	0	0	1	0	1	1	1
0	1	0	1	1	0	1	1	1
0	1	1	0	1	1	1	1	0
0	1	1	1	0	1	1	1	0
1	0	0	0	1	1	0	1	0
1	0	0	1	1	1	0	1	0
1	0	1	0	1	1	0	1	0
1	0	1	1	0	1	1	0	1
1	1	0	0	1	0	1	0	1
1	1	0	1	1	0	1	0	1
1	1	1	0	1	1	1	0	1
1	1	1	1	0	1	1	0	1

NAND Imp.: Derivation of Truth Table …

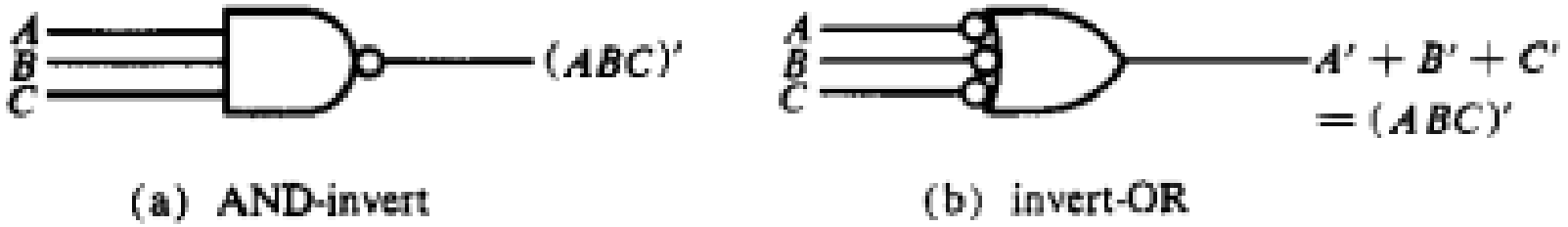
Now we can use K-map to get the simplified expression of the given NAND logic circuit.



$$F = AB + ACD + BC' = A(B + CD) + BC'$$

NAND Imp.: Block Diagram Transformation

- Without employing **De Morgan's Law.**
- We use two **alternate graphic symbols** of NAND gates

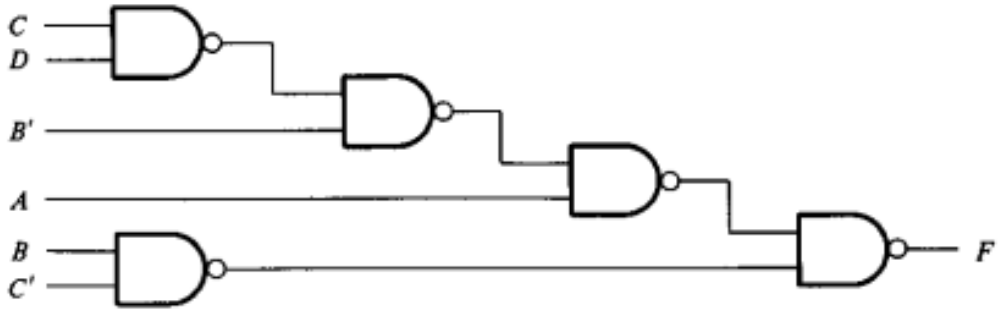


(a) AND-invert (b) invert-OR

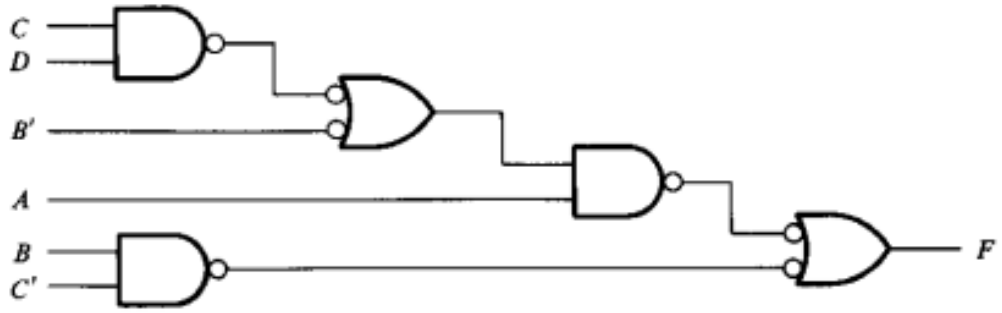
NAND Imp.: Block Diagram Transformation…

- **NAND logic diagram – AND-OR diagram conversion**
  - A change in symbols from AND-invert to an invert-OR **in alternate levels** of gates.
  - **At first** we start from the **last level**, to be changed **to an invert-OR symbol**.
  - These changes produce **pairs of circles along the same line** – can be removed as they can be cancelled.
  - **One input AND or OR gate** can be changed to an inverter.
  - **Circle with external input** can be changed to corresponding complemented variable.

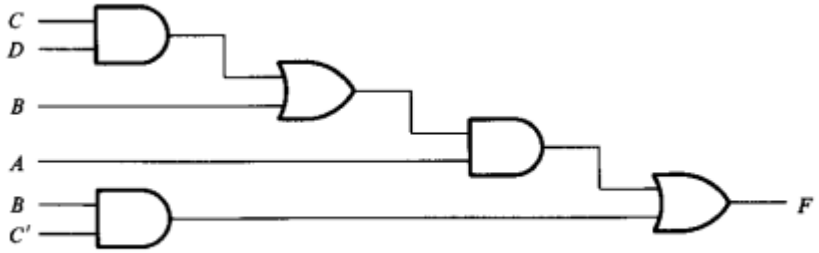
NAND Imp.: Block Diagram Transformation…



(a) NAND logic diagram



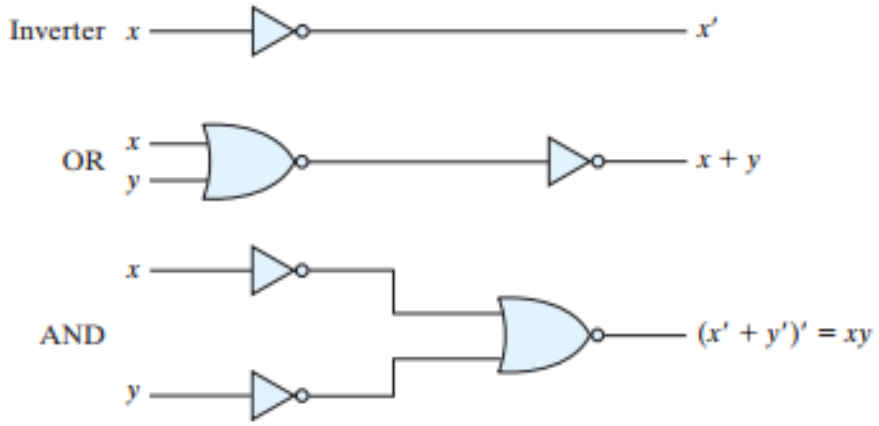
(b) Substitution of invert-OR symbols in alternate levels

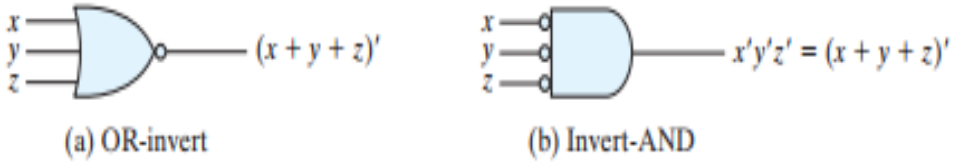


(c) AND-OR logic diagram

NOR Implementation

- NOR function – Dual of NAND function – also dual of the corresponding procedures and rules for NAND logic
- NOR gate – universal gate – used to construct combinational and sequential circuit





(a) OR-invert (b) Invert-AND

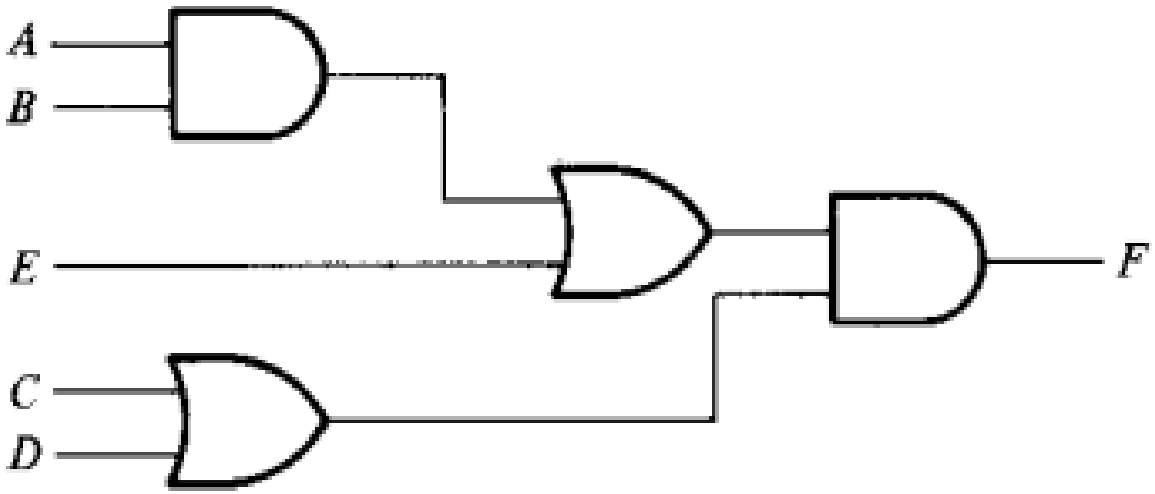
NOR gate

Boolean Function Imp. With NOR Gates

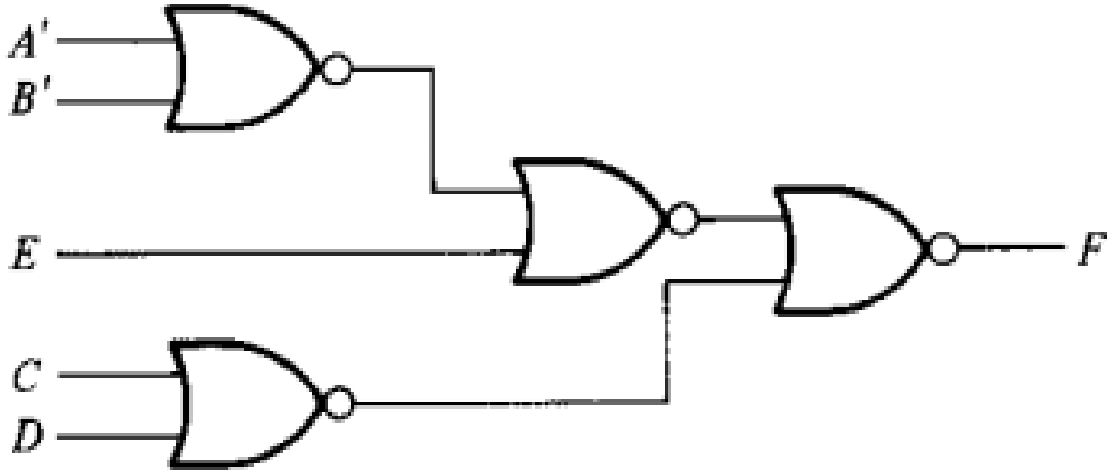
Block Diagram Method:

1. Draw the AND-OR logic diagram from the given algebraic expression considering normal and complement form of inputs.
2. Draw the **second logic diagram** with the **equivalent NOR logic** substituted for each AND, OR and NOT gate
3. Remove any two cascaded inverts from the diagram, since double inversion doesn't perform a logic function. Remove inverters connected to single external inputs and complement the corresponding input variable.

Block Diagram Method: NOR Imp.



(a) AND-OR diagram



(c) Alternate NOR diagram

In general, the number of NOR gates required to implement a function is equal to the number of AND-OR gates, except for an occasional inverter. It will be applicable when both normal and complement inputs are available.

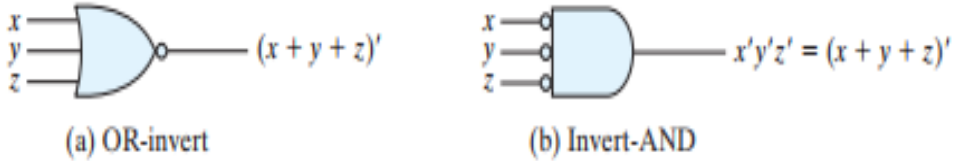
NOR Implementation: Analysis Procedure

- To derive the Boolean function from a logic diagram, we mark the outputs of various gates with arbitrary symbols.
- By repetitive substitutions, we obtain the output variables as a function of the input variables.

To obtain the truth table from a logic diagram without deriving the Boolean function first, we form a table listing the n input variables with 2^n rows of 1s and 0s. The truth table of various NOR gate outputs is derived in succession until the output truth table is obtained.

Block Diagram Transformation

To convert a NOR logic diagram to its equivalent AND-OR logic diagram, we use the two symbols for NOR gates.



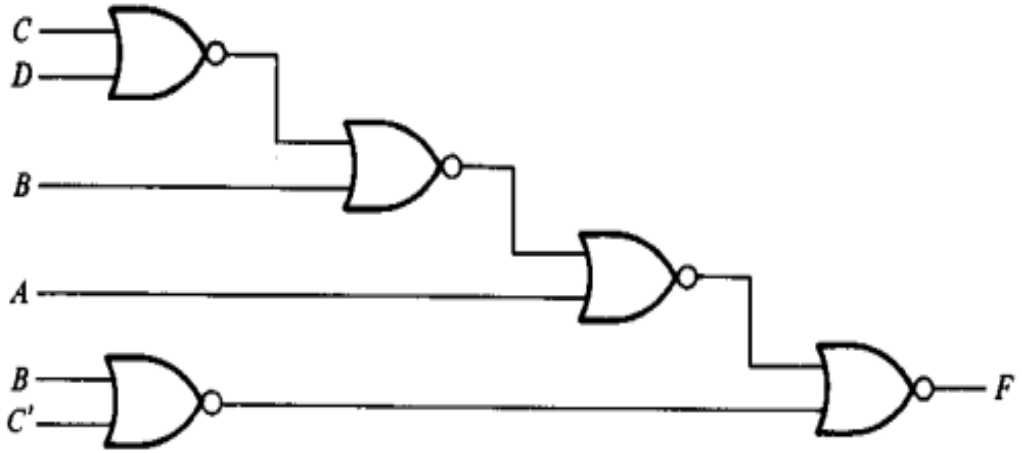
(a) OR-invert (b) Invert-AND

The conversion of a NOR logic diagram to an AND-OR diagram can be achieved through a change in symbols from OR-invert to invert-AND starting from the last level and in alternative levels.

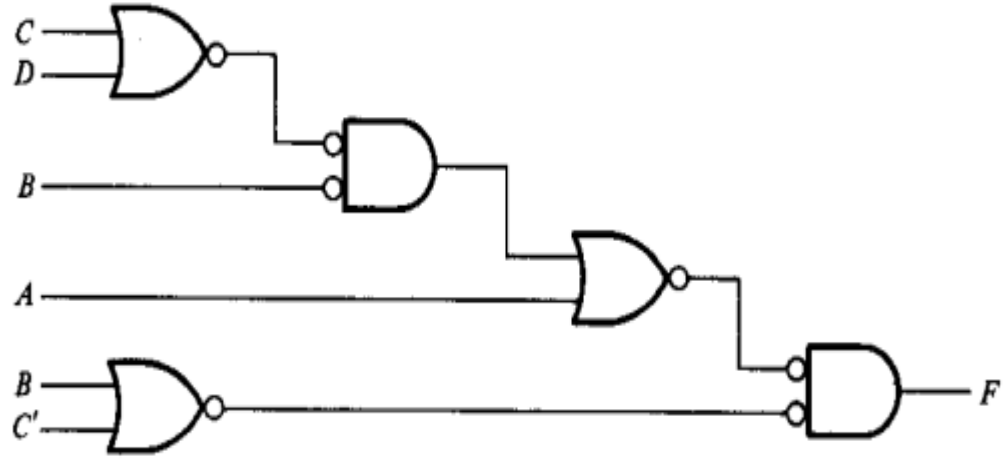
Pairs of small circles along the same line must be removed.

A one-input AND or OR gate should be removed but if it has a small circle at the input or output, it should be converted to an inverter.

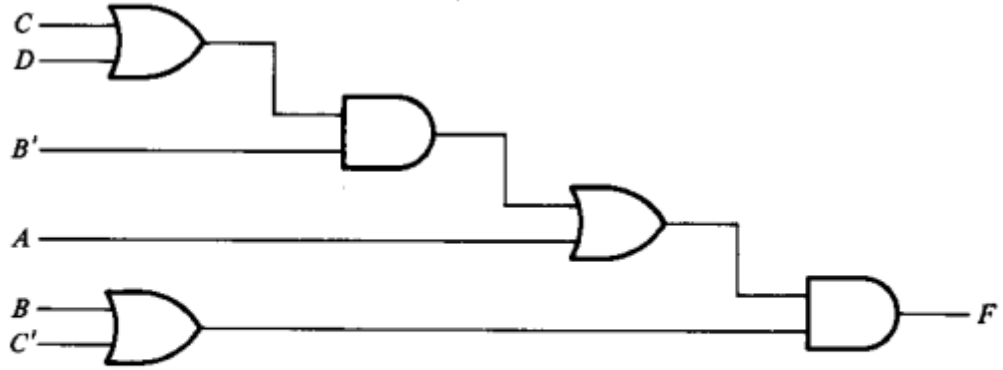
Block Diagram Transformation



(a) NOR logic diagram



(b) Sustituting invert-AND in alternate levels

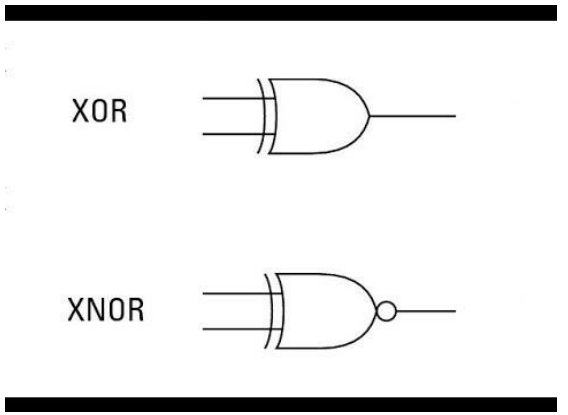


(c) AND-OR logic diagram

FIGURE 4-20
Conversion of NOR diagram to AND-OR

Exclusive-OR and Equivalence Functions

- **Exclusive OR** denoted as $\oplus$ (also named as XOR)
- **Equivalence** denoted as $\odot$ (also named as XNOR)
- They perform following Boolean functions:
  - $x \oplus y = xy' + x'y$
  - $x \odot y = xy + x'y'$
- They are complements each other.

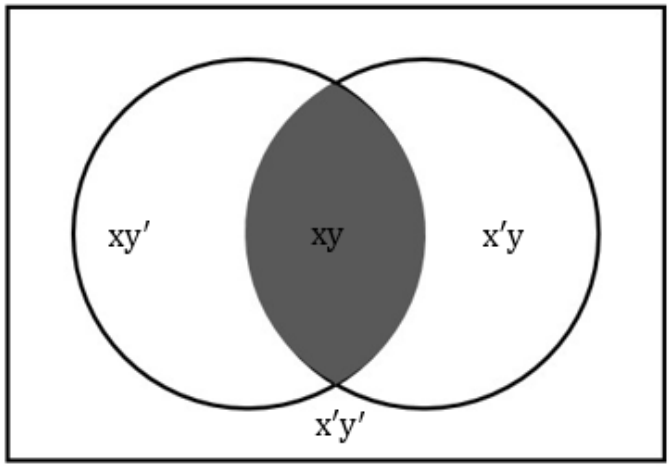


A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

XOR

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

XNOR



XOR and XNOR …

- Each of them are commutative and associative. For these two properties, a function of three or many variables can be expressed without parenthesis as follows:

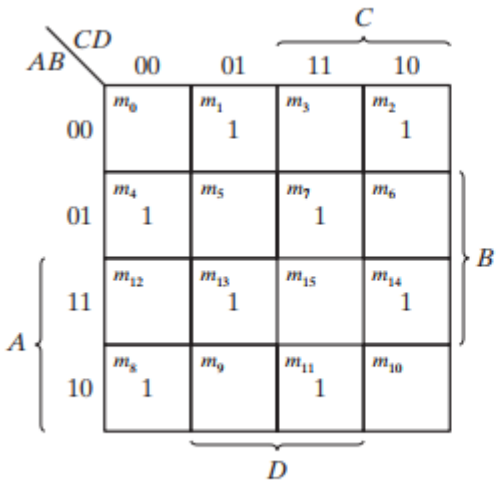
$$A \oplus B = B \oplus A$$

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

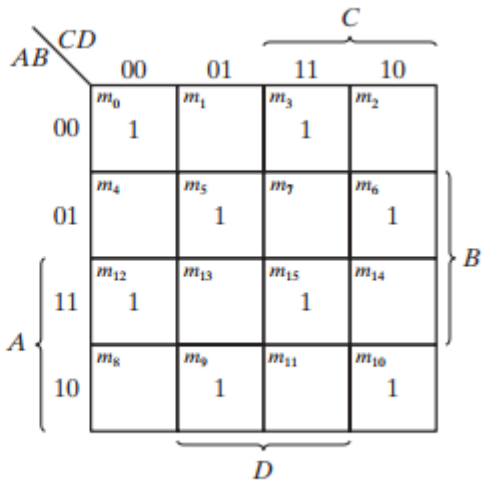
- These two gates allow three or more inputs but uneconomical from hardware standpoint
- XOR and XNOR – with AND-OR-NOT, NAND, NOR implementation. (HW)
- Exclusively expressed in limited number of Boolean functions – emerge quite often – particularly in **arithmetic operation** and **error detection and correction**

XOR and XNOR …

- An n variable **XOR** expression has $2^n/2$ minterms whose equivalent binary numbers have **an odd number of 1s.**
- An n variable **XNOR** expression has $2^n/2$ minterms whose equivalent binary numbers have **an even number of 0s.**



(a) Odd function $F = A \oplus B \oplus C \oplus D$



(b) Even function $F = (A \oplus B \oplus C \oplus D)'$

XOR and XNOR …

- When the number of variables (n) in a function is odd, the minterms with **an even number of 0s (XNOR)** and **an odd number of 1s (XOR)** are same. (XOR = XNOR, when they have same odd number of variables)
- But when the number of variables (n) in a function is even, they form the complements of each other.
- **Odd function:** In multiple variable XOR operation, if the number of 1 is odd.
- **Even function:** In multiple variable XOR operation, if the number of 1 is even.

Implementation of XOR and XNOR

- In S output of a full adder and D output of a full subtractor
- Also very useful in systems requiring error detection and error correction code (in parity bit)
- Parity generator – generates the parity bit in the transmitter
- Parity checker – checks the parity bit in the receiver

Example:

As an example, consider a three-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 (including P) even.

Table 3.3
Even-Parity-Generator Truth Table

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

Example…

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. 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.

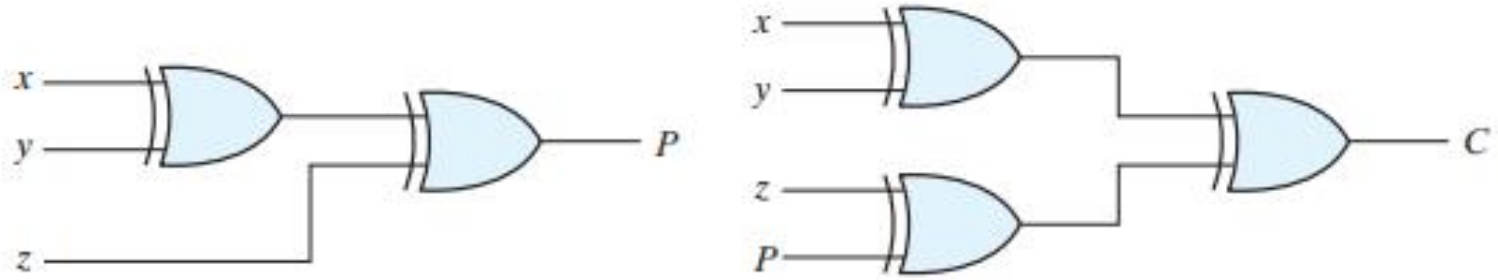
Example…

Table 3.4
Even-Parity-Checker Truth Table

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

Example…

$$P = x \oplus y \oplus z \qquad C = x \oplus y \oplus z \oplus P$$



(a) 3-bit even parity generator (b) 4-bit even parity checker

FIGURE 3.34
Logic diagram of a parity generator and checker

Administration