Digital Logic Design

Lecture 11

Don’t Care Conditions

- The minterms of a Boolean function specify all combinations of variable values for which the function is equal to 1. The function is assumed to be equal to 0 for the rest of the minterms.
- This assumption, however, is not always valid, since there are applications in which the function is not specified for certain variable value combinations.
- There are two cases in which this occurs.
  - In the first case, the input combinations never occur. As an example, the four-bit binary code for the decimal digits has six combinations that are not used and not expected to occur.
  - In the second case, the input combinations are expected to occur, but we do not care what the outputs are in response to these combinations.

Contd.

- In both cases, the outputs are said to be unspecified for the input combinations.

Incompletely specified functions

- Functions that have unspecified outputs for some input combinations are called incompletely specified functions.
- In most applications, we simply do not care what value is assumed by the function for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function don't-care conditions.
- These conditions can be used on a map to provide further simplification of the function.

Contd.

- It should be realized that a don’t-care minterm cannot be marked with a 1 on the map, because that would require that the function always be a 1 for such a minterm.
- Likewise, putting a 0 in the square requires the function to be 0.
- To distinguish the don’t-care condition from 1s and 0s, an **X** is used.

Simplification of an incompletely specified function

- When choosing adjacent squares to simplify the function in the map, the ×’s may be assumed to be either 0 or 1, whichever gives the simplest expression.
- An × need not be used at all if it does not contribute to covering a larger area.

Example 2.14

- Simplify the Boolean function F (A, B, C, D) = Σm(1,3,7,11,15)
- which has the don't-care conditions d (A, B, C, D) = Σm(0,2,5).

SOP: $F = C\,D + \overline{A}\,\overline{B}$, $F = C\,D + \overline{A}\,D$

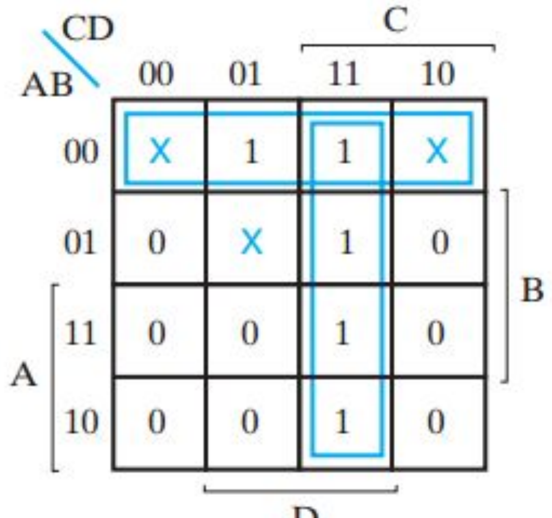
- It is also possible to obtain an optimized **product-of-sums** expression for the function.

POS: In this case, the way to combine the 0s is to include don't-care minterms 0 and 2 with the 0s, giving the optimized complemented function.

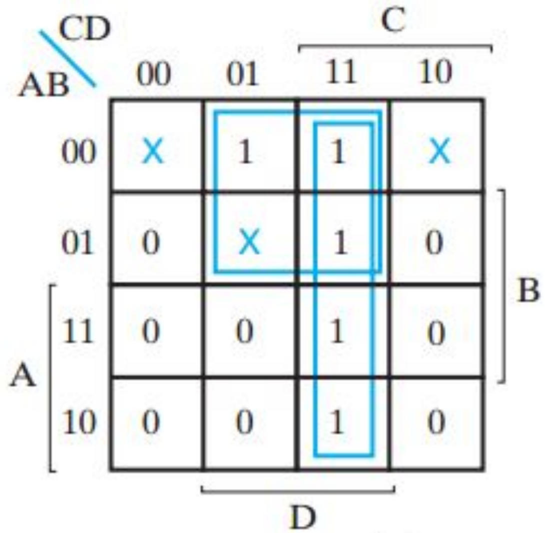
$\overline{F} = \overline{D} + A\,\overline{C}$

Taking the complement of F

$F = D(\overline{A} + C)$



(a) $F = CD + \overline{A}\,\overline{B}$



(b) $F = CD + \overline{A}D$

Exclusive OR/ Exclusive NOR

- **The *eXclusive OR* (*XOR*) function is an important Boolean function used extensively in logic circuits.**
- **The XOR function may be;**
  - **implemented directly as an electronic circuit (truly a gate) or**
  - **implemented by interconnecting other gate types (used as a convenient representation)**
- **The *eXclusive NOR* function is the complement of the XOR function**
- **By definition, XOR and XNOR gates are complex gates.**

Exclusive OR/ Exclusive NOR

- Uses for the XOR and XNORs gate include:
  - Adders/subtractors/multipliers
  - Counters/incrementers/decrementers
  - Parity generators/checkers
- Definitions
  - The XOR function is: $X \oplus Y = X\overline{Y} + \overline{X}Y$
  - The eXclusive NOR (XNOR) function, otherwise known as *equivalence* is: $\overline{X \oplus Y} = XY + \overline{X}\,\overline{Y}$
- Strictly speaking, XOR and XNOR gates do no exist for more than two inputs. Instead, they are replaced by odd and even functions.

Truth Tables for XOR/XNOR

- Operator Rules: XOR XNOR

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

X	Y	$\overline{(X \oplus Y)}$ or $X \equiv Y$
0	0	1
0	1	0
1	0	0
1	1	1

- The XOR function means:

  X OR Y, but NOT BOTH

- Why is the XNOR function also known as the *equivalence* function, denoted by the operator ≡?

XOR/XNOR (Continued)

- **The XOR function can be extended to 3 or more variables. For more than 2 variables, it is called an *odd function* or *modulo 2 sum* (*Mod 2 sum*), not an XOR:**

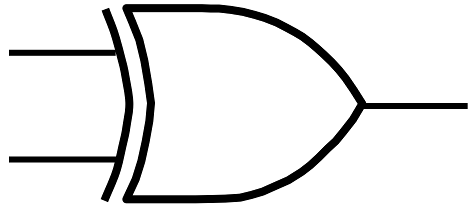
$$X \oplus Y \oplus Z = \overline{X}\,\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\,\overline{Z} + XYZ$$

- **The complement of the odd function is the even function.**
- **The XOR identities:**

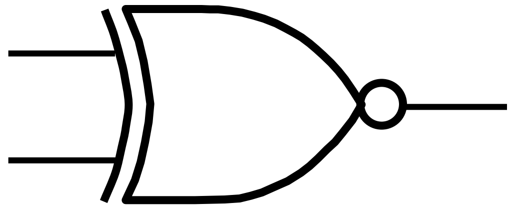
$$X \oplus 0 = X \qquad\qquad X \oplus 1 = \overline{X}$$
$$X \oplus X = 0 \qquad\qquad X \oplus \overline{X} = 1$$
$$X \oplus Y = Y \oplus X$$
$$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z$$

Symbols For XOR and XNOR

- XOR symbol:



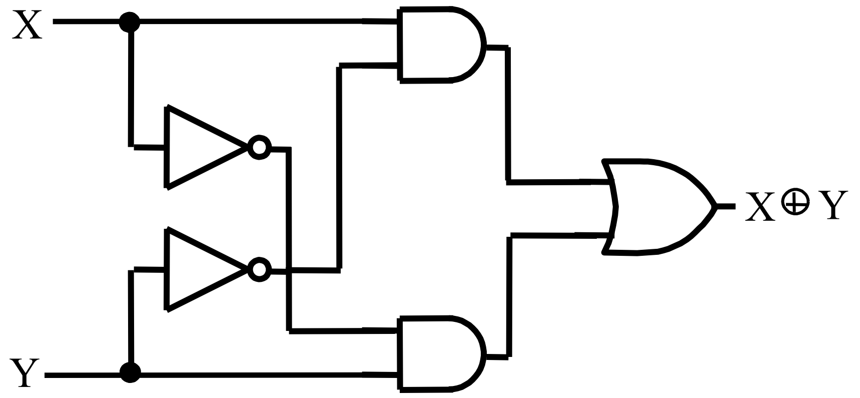
- XNOR symbol:



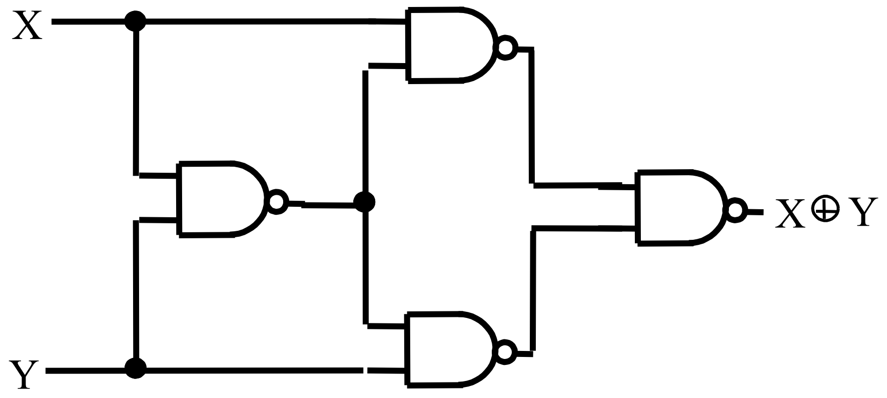
- Shaped symbols exist only for two inputs

XOR Implementations

- The simple SOP implementation uses the following structure:



- A NAND only implementation is:

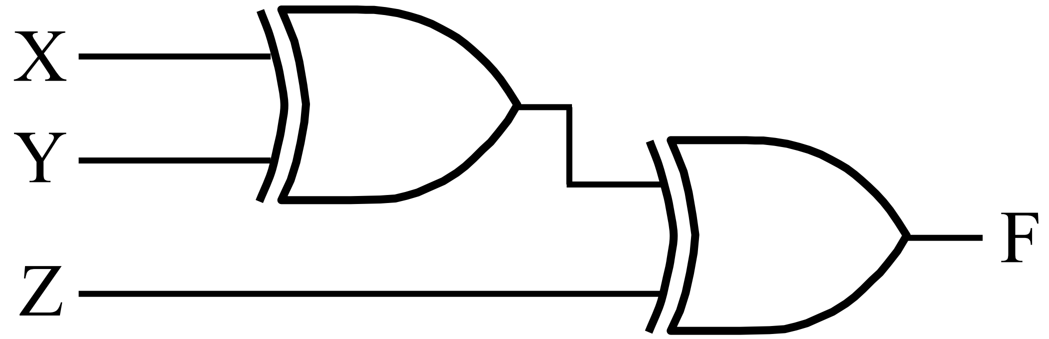


Odd and Even Functions

- The odd and even functions on a K-map form “checkerboard” patterns.
- The 1s of an odd function correspond to minterms having an index with an odd number of 1s.
- The 1s of an even function correspond to minterms having an index with an even number of 1s.
- Implementation of odd and even functions for greater than four variables as a two-level circuit is difficult, so we use “trees” made up of :
  - 2-input XOR or XNORs
  - 3- or 4-input odd or even functions

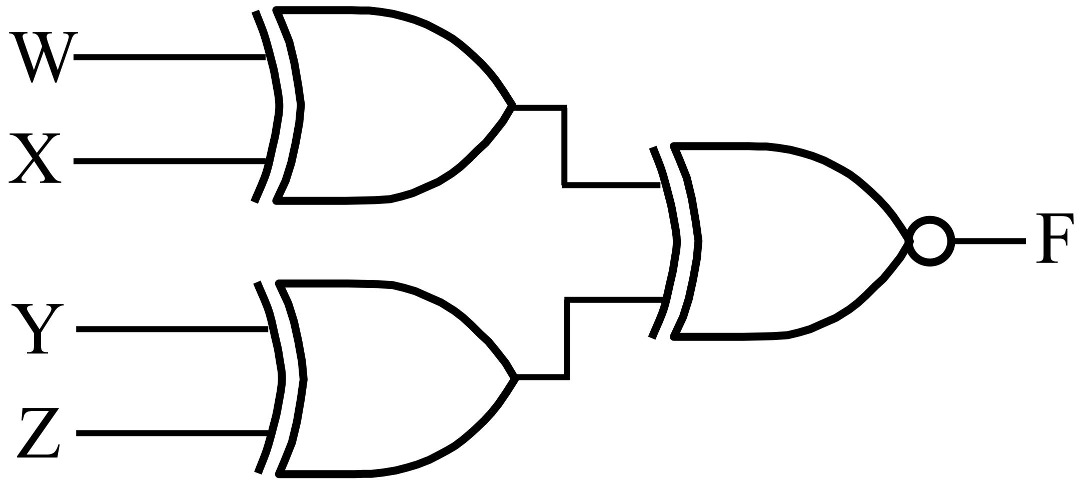
Example: Odd Function Implementation

- Design a 3-input odd function $F = X \oplus Y \oplus Z$ with 2-input XOR gates
- Factoring, $F = (X \oplus Y) \oplus Z$
- The circuit:



Example: Even Function Implementation

- Design a 4-input odd function $F = W \oplus X \oplus Y \oplus Z$ with 2-input XOR and XNOR gates
- Factoring, $F = (W \oplus X) \oplus (Y \oplus Z)$
- Implementation of even function using odd inputs
- The circuit:



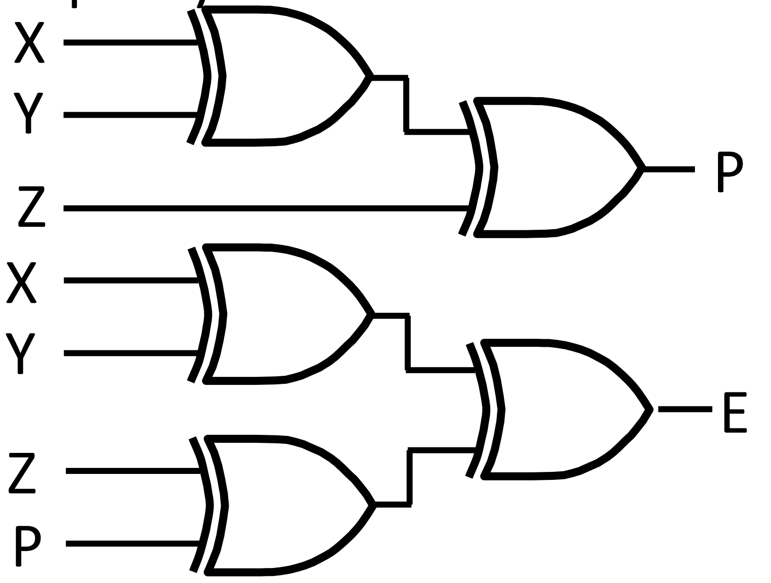
Parity Generators/ Checkers

- In Chapter 1, a parity bit added to n-bit code to produce an n + 1 bit code:
    - Add odd parity bit to generate code words with even parity
    - Add even parity bit to generate code words with odd parity
    - Use odd parity circuit to check code words with even parity
    - Use even parity circuit to check code words with odd parity
- Example: n = 3. Generate even parity code words of length four with odd parity generator:
- Check even parity code words of length four with

odd parity checker:

- Operation: (X,Y,Z) = (0,0,1) gives (X,Y,Z,P) = (0,0,1,1)

and E = 0. If Y changes from 0 to 1 between generator and checker, then E = 1 indicates an error.



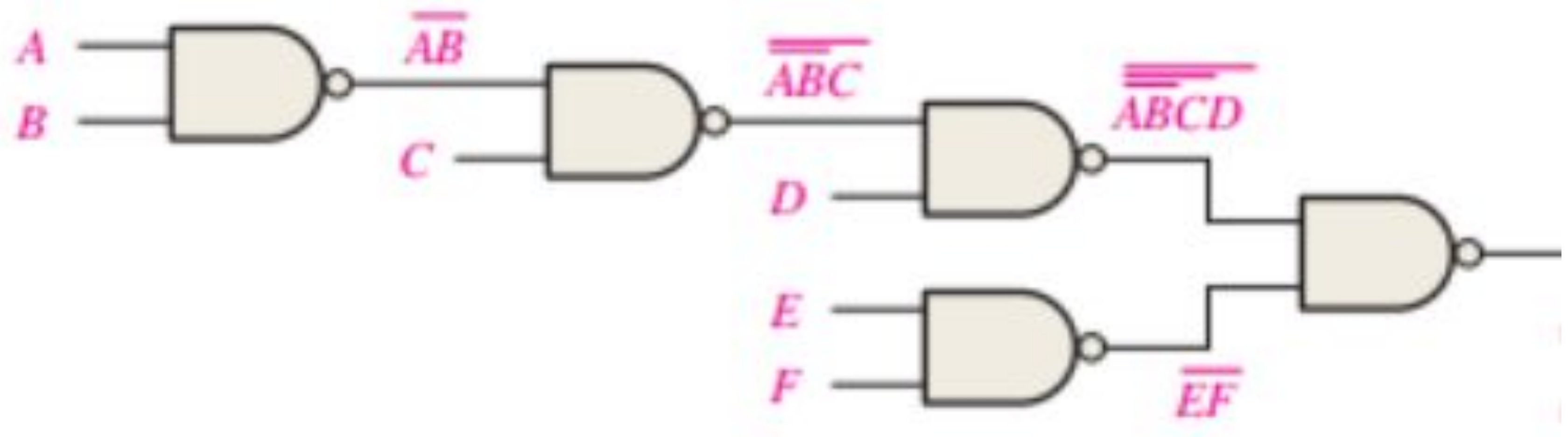
Implementing functions with only OR and NOT gates

- $F = AB'C + A'C' + AB$
- $F = \overline{\overline{AB'C + A'C' + AB}}$
- $F = \overline{\overline{AB'C} \,.\, \overline{A'C'} \,.\, \overline{AB}}$
- By De Morgan's Laws
- $F = \overline{(A' + B + C') \,.\, (A + C) \,.\, (A' + B')}$
- $F = \overline{(A' + B + C')} + \overline{(A + C)} + \overline{(A' + B')}$

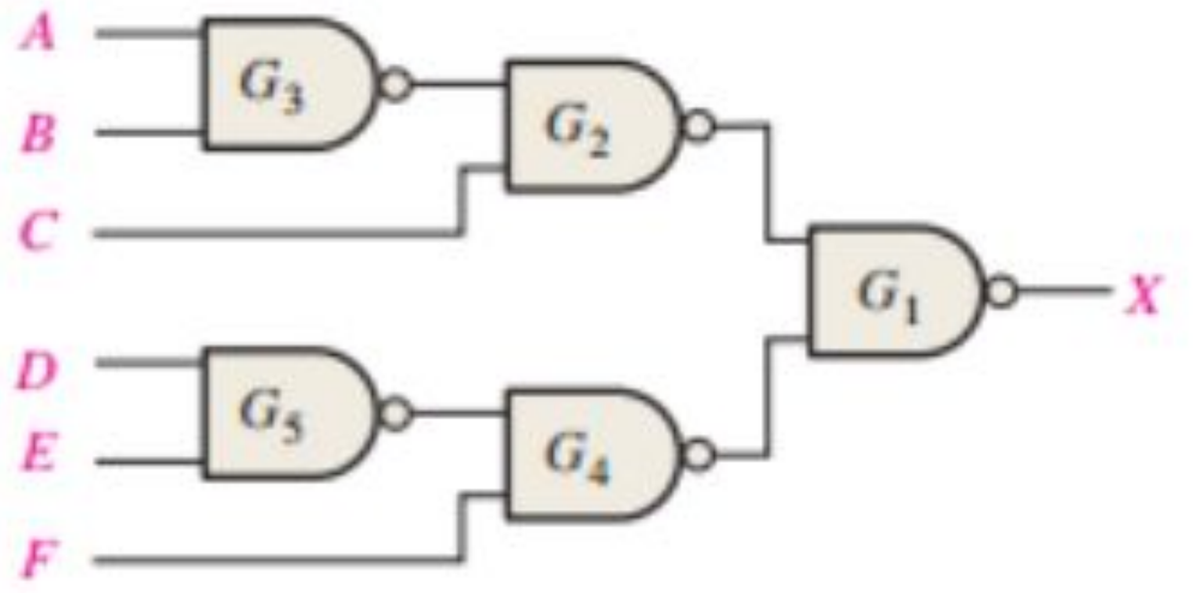
Implementing functions with only AND and NOT gates

- $F = (A+B'+C) . (A'+B+C) . (A'+C) . (B+C')$
- $F = \overline{\overline{(A+B'+C) . (A'+B+C) . (A'+C) . (B+C')}}$
- $F = \overline{\overline{(A+B'+C)} + \overline{(A'+B+C)} + \overline{(A'+C)} + \overline{(B+C')}}$
- $F = \overline{(A'.B.C') + (A.B'.C') + (A.C') + (B'.C)}$
- $F = \overline{(A'.B.C')} . \overline{(A.B'.C')} . \overline{(A.C')} . \overline{(B'.C)}$

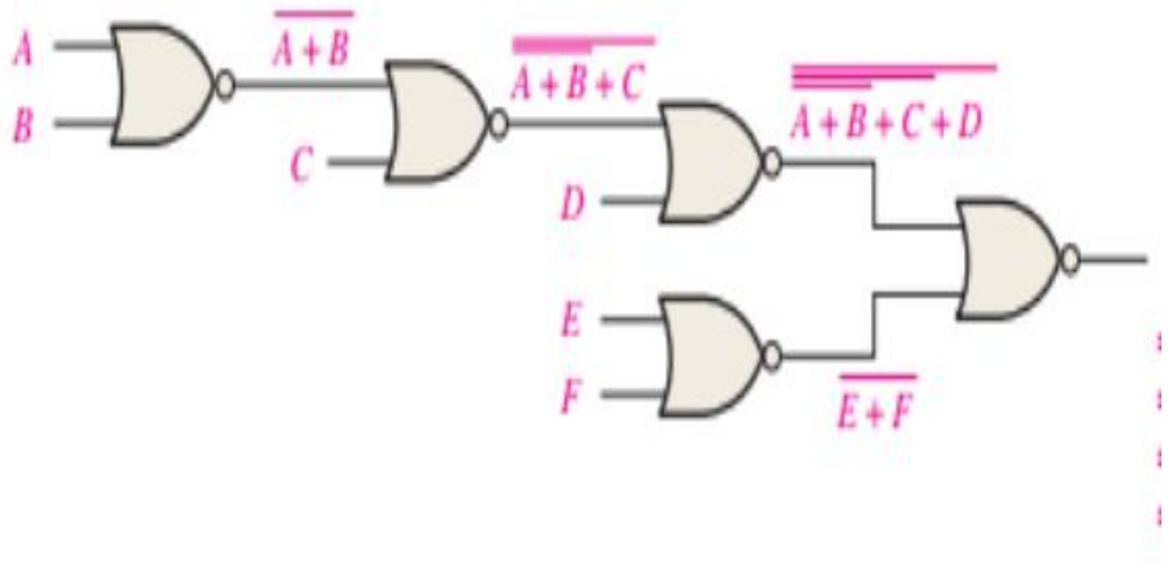
Implementation using only NAND Gate:
F=(AB + C’)D + EF


A
B
$\overline{AB}$
C
$\overline{\overline{AB}C}$
D
$\overline{\overline{\overline{AB}C}D}$
E
F
$\overline{EF}$

Example 2: F = (A'+B')C + (D'+E')F

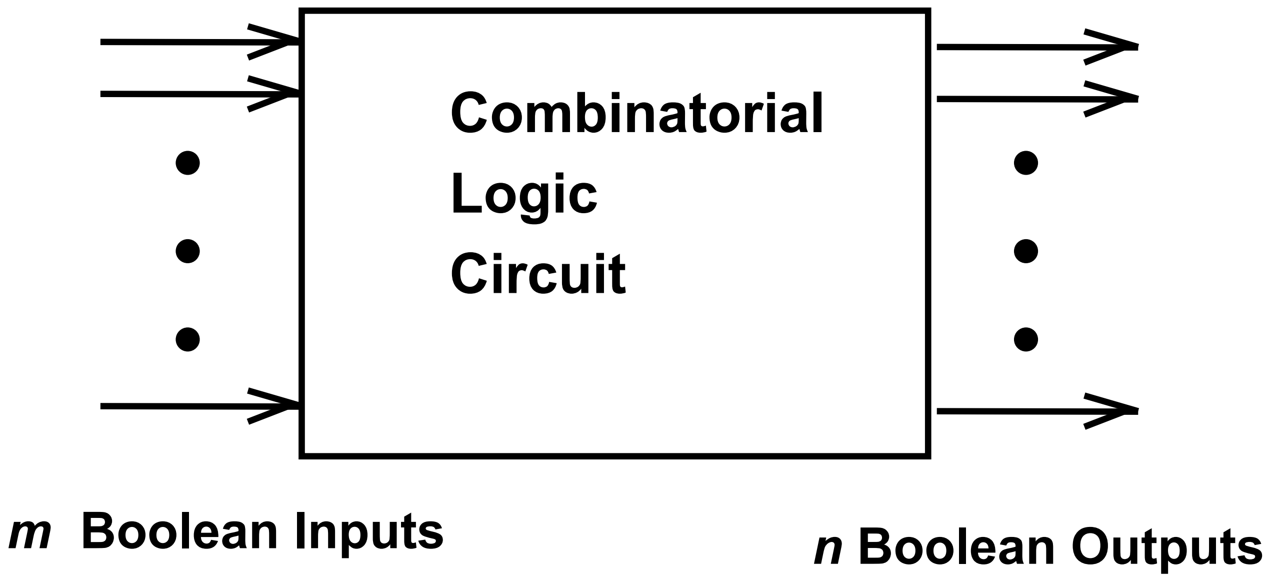


Implementation using only NOR Gates:
F = ((A+B)C’+ D) (E+F)


A
B
$\overline{A+B}$
C
$\overline{\overline{A+B}+C}$
D
$\overline{\overline{\overline{A+B}+C}+D}$
E
F
$\overline{E+F}$

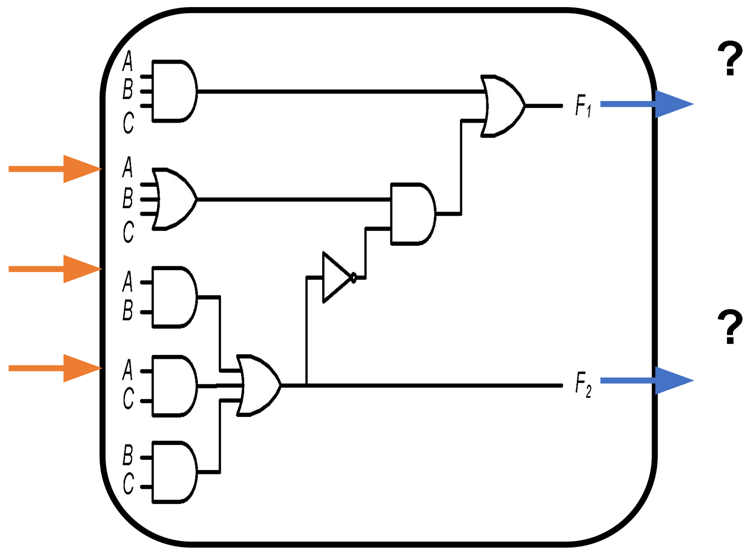
Combinational Circuits

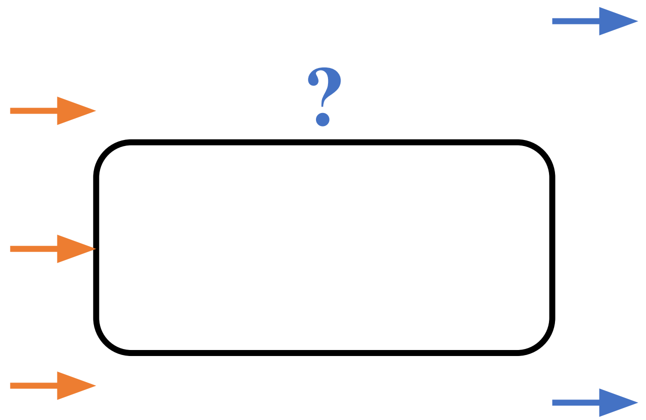
- A combinational logic circuit has:
  - A set of *m* Boolean inputs,
  - A set of *n* Boolean outputs, and
  - *n* switching functions, each mapping the 2^m input combinations to an output such that the current output depends only on the current input values
- A block diagram:



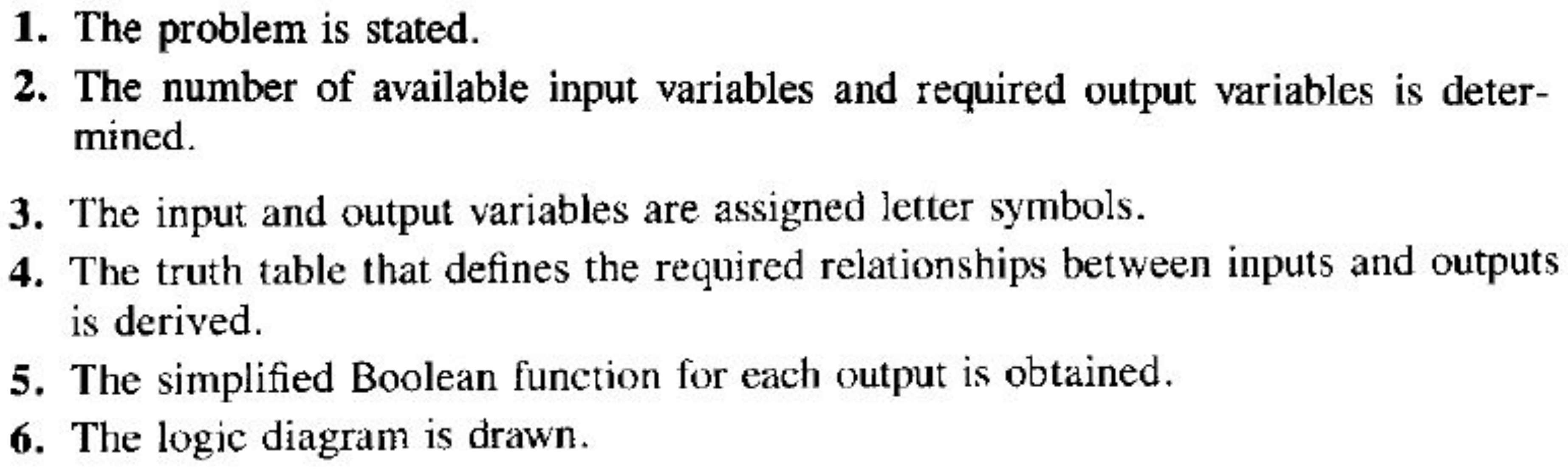
Combinational Circuits

- Analysis
  - Given a circuit, find out its *function*
  - Function may be expressed as:
    - Boolean function
    - Truth table
- Design
  - Given a desired function, determine its *circuit*
  - Function may be expressed as:
    - Boolean function
    - Truth table





Design Procedure



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.

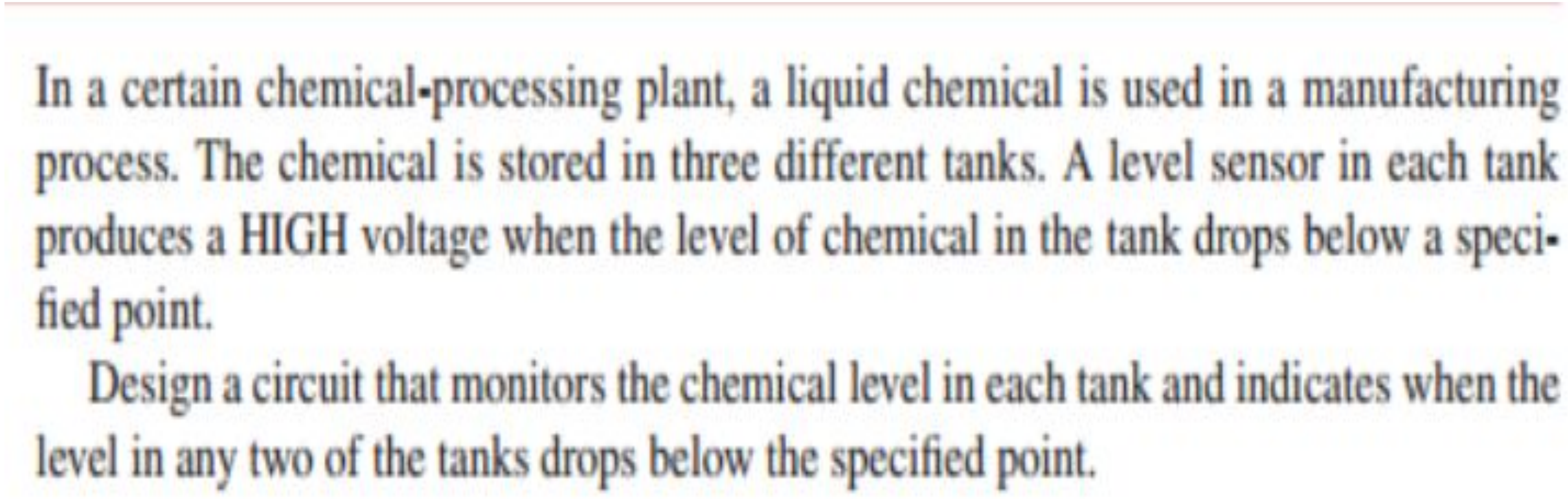
Design Procedure

1. Specification
   - Write a specification for the circuit if one is not already available
2. Formulation
   - Derive a truth table or initial Boolean equations that define the required relationships between the inputs and outputs, if not in the specification
   - Apply hierarchical design if appropriate
3. Optimization
   - Apply 2-level and multiple-level optimization
   - Draw a logic diagram or provide a netlist for the resulting circuit using ANDs, ORs, and inverters

Design Procedure

4. Technology Mapping
   - Map the logic diagram or netlist to the implementation technology selected
5. Verification
   - Verify the correctness of the final design manually or using simulation

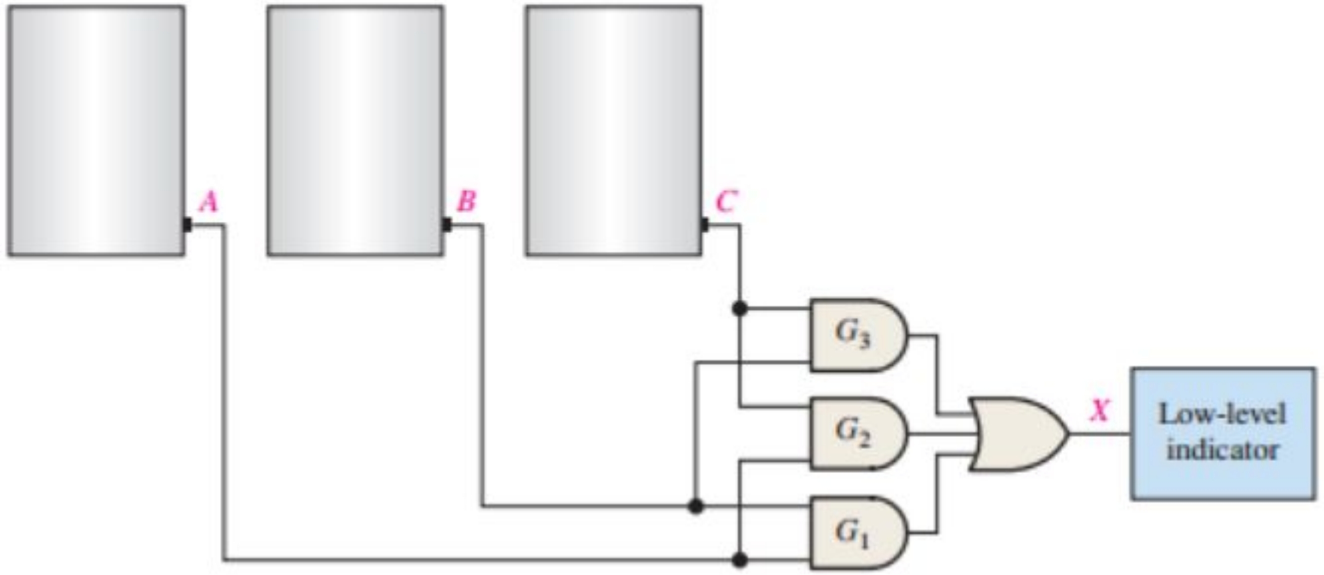
Example: Design a circuit diagram


In a certain chemical-processing plant, a liquid chemical is used in a manufacturing process. The chemical is stored in three different tanks. A level sensor in each tank produces a HIGH voltage when the level of chemical in the tank drops below a specified point.

Design a circuit that monitors the chemical level in each tank and indicates when the level in any two of the tanks drops below the specified point.

Solution

The AND-OR circuit in Figure 5–2 has inputs from the sensors on tanks *A*, *B*, and *C* as shown. The AND gate G_1 checks the levels in tanks *A* and *B*, gate G_2 checks tanks *A* and *C*, and gate G_3 checks tanks *B* and *C*. When the chemical level in any two of the tanks gets too low, one of the AND gates will have HIGHs on both of its inputs, causing its output to be HIGH; and so the final output *X* from the OR gate is HIGH. This HIGH input is then used to activate an indicator such as a lamp or audible alarm, as shown in the figure.


A
B
C
G_3
G_2
G_1
X
Low-level indicator