EE 200: Digital Logic Circuit Design

Unit 2
Binary Logic and Gates

Charles Kime & Thomas Kaminski

Unit 2: Binary Logic and Gates
Contents

1. Binary logic and gates, Boolean Algebra, Basic identities of Boolean algebra 2.1, 2.2
2. Boolean functions, Algebraic manipulation, Complement of a function 2.2
3. Canonical & Standard forms, Minterms & Maxterms, Sum of products, Product of Sums. Algebraic simplification of logic functions 2.3
4. Physical properties of gates: Fan-in, Fan-out, Propagation Delay, HiZ (Tristate) outputs 6.1, 6.2.
5. Map method of logic circuit optimization:
   - Two-, Three-, and Four-literal K-Map 2.4
   - Optimization procedure: Essential prime implicants, Selected Additional prime implicants 2.5
   - Simplification with Don't care conditions 2.5
6. Other Gate Types: Universal gates (NAND and NOR), 2-level Complex gates (AO, AOI, OA. OAI) 2.7
7. Exclusive-OR (XOR) and Equivalence (XNOR) gates, Parity generation and checking 2.8

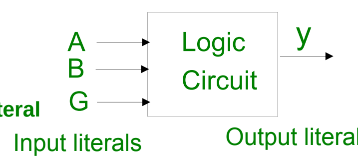
1. Binary Logic and Gates: **Definitions**

- Binary literals take on one of two values: e.g. (1,0) (T,F)
- Logical operators operate on binary values and binary literals
- Basic logical operators perform the logic functions AND, OR and NOT
- Logic gates: Circuits that implement logic functions
- Boolean Algebra: a useful mathematical system for specifying and transforming logic functions
- We will study Boolean algebra as a foundation for designing and analyzing digital systems

Binary **literals**

- A literal is a binary variable or its complement and therefore takes only one of two possible values
- Recall from Unit 1 that these two binary values can have different names:
  - True/False
  - On/Off
  - Yes/No
  - 1/0
- We use 1 (=true) and 0 (false) here to denote these two values
- literal identifier examples:
  - A, B, y, z, or X_1 for now
  - RESET, START_IT, or ADD1 later

More meaningful names that describe function of literal



Logical **Operations** on Binary **literals**

- The three basic logical operations are:
  - AND
  - OR
  - NOT
- AND is denoted by a dot (·)
- OR is denoted by a plus (+)
- NOT is denoted by an overbar (¯), a single quote mark (') after, or (~ or #) before the literal, e.g. $\overline{A}$, ‘A, ~A, or #A

Notation Examples- Logical **Operators**

- Examples:

If no ambiguity is caused, we may omit the dot: **Y = AB**

Product, Intersection
- **Y = A · B** is read “Y is equal to A AND B”
  (Y is True when Both A & B are True)

Sum, Union
- **z = x + y** is read “z is equal to x OR y”
  (Z is True when either X or Y are True)

Negation, Complementing
- **X =** $\overline{A}$ is read “X is equal to NOT A”
  (X is True when Y is Not True)

Note that both the “.” (dot) and the “+” operators also have mathematical functions of multiplication and addition, respectively

Definitions of the 3 Basic Logic Operations

Operations are defined on the values "0" and "1" for each operator:

AND (.)

$0 \cdot 0 = 0$
$0 \cdot 1 = 0$
$1 \cdot 0 = 0$
$1 \cdot 1 = 1$

Anything 'ANDed' with zero gives a zero result

The only way to get a 1 is to **AND** ALL 1s

OR (+)

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 1$

The only way to get a 0 is to **OR** ALL 0s

Anything 'ORed' with one gives a one result

NOT ($\overline{\ }$)

$\overline{0} = 1$
$\overline{1} = 0$

No corresponding Math operator for **NOT**

Multiplication and **AND** give identical results

Addition and **OR** give different Results for 1+1

Truth Tables

- ***Truth table*** **- A tabular listing of the values of a logic function for all possible combinations of the values of its argument (input) variables**
- **Truth tables for the three basic logic operations:**

AND

X	Y	$Z = X \cdot Y$
0	0	0
0	1	0
1	0	0
1	1	1

OR

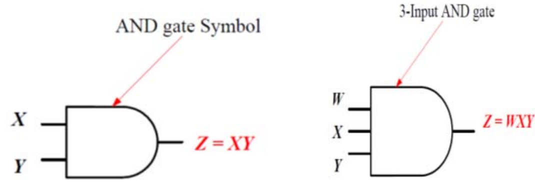
X	Y	$Z = X + Y$
0	0	0
0	1	1
1	0	1
1	1	1

NOT

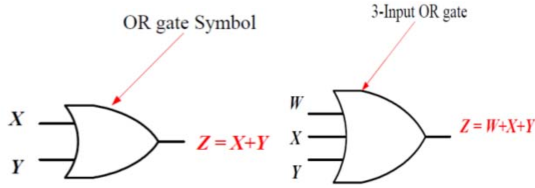
X	$Z = \overline{X}$
0	1
1	0

Logic Gates

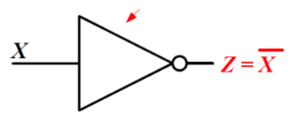
- Electronic devices that implement logic operators are called Gates:
  - **AND gate** implements AND operation



  - **OR gate** implements OR operation



  - **NOT gate** (or simply an **INVERTER**) implements NOT operation

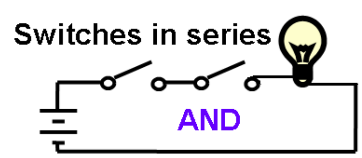


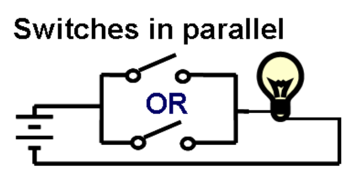
Practical Implementation of the Basic Logic Gates

Basically…Using Switches

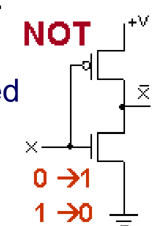
Input/Output Definitions

- Input:
  - logic 1 is switch closed
  - logic 0 is switch open
- Output:
  - logic 1 is lamp on
  - logic 0 is lamp off.



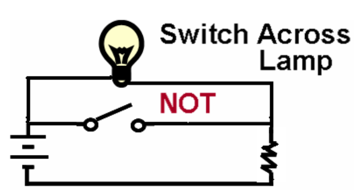


In today's computers, Switches are implemented using transistors, e.g. →



0 = Low Voltage, e.g. 0 V

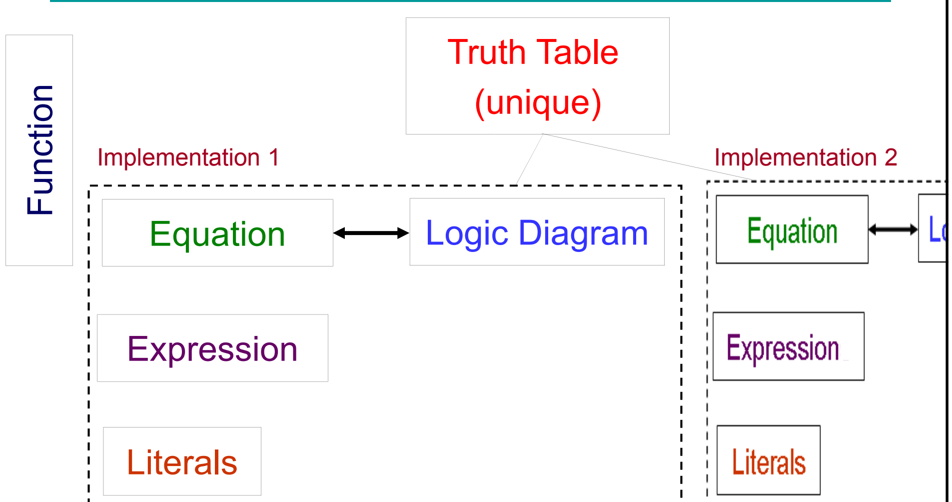
1 = High Voltage, e.g. 3 V



2. Boolean Algebra- Formal Definitions

- The algebra that deals with binary literals and logic functions
- literals: Denote by letters of the alphabet, e.g. A, B, X, Y, Z
- Basic Logic operations (operators) on those literals: AND, OR, NOT
- A Boolean Expression (e.g. X+YZ) is Formed by:
  - Binary literals
  - Logic operations (operators) on the literals and constants
  - Parenthesis
  - Constants 0,1
- A Boolean Function can be described by a Boolean Equation of the form: Output = Boolean Expression (not unique)
- Each Function can be represented as a logic diagram (not unique)
- A Boolean Function can be uniquely expressed as a truth table that maps each possible combination of the input literals to the corresponding output literal (n input literals → 2^n combinations)
- Later in this unit, we will consider optimization methods to derive the simplest Boolean functions that implement a given truth table
- Simplest functions require the smallest number of the smallest gates and therefore are most economical to implement

Boolean Algebra



Several equivalent implementations
- Use the optimum one to implement the function

Optimization will be basically simplification

Boolean **Function**: Represented by many Equations, Logic Diagrams, but a single Truth Table

Truth Table (Unique)

Inputs

X Y Z	Output F
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	1

Unique truth table listing function output for all possible input combinations ($2^3 = 8$)

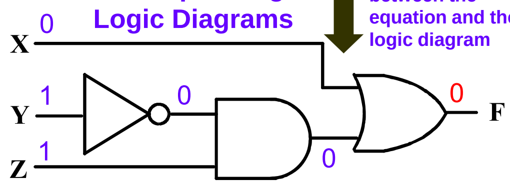
Design → Many Corresponding Logic Diagrams

Analyze ← Unique

Equation/Diagram Pair

$$F = X + \overline{Y}Z$$

One-to-one correspondence between the equation and the logic diagram



Other possible Equation/Diagram Pairs

- Boolean equations, logic diagrams, and truth tables describe the **same logic function**!
- Truth tables are unique; but equations and logic diagrams are not. They can be manipulated to produce simpler expressions requiring fewer gates → Optimization

Boolean Algebra Identities

		Dual		Comments
Single literal	1. $X + 0 = X$	OR ↔ AND	2. $X \cdot 1 = X$	0 opens OR, 1 opens AND
	3. $X + 1 = 1$	AND ↔ OR	4. $X \cdot 0 = 0$	1 blocks OR, 0 blocks AND
	5. $X + X = X$	1 ↔ 0	6. $X \cdot X = X$	Duplicating a literal has no effect
	7. $X + \overline{X} = 1$	0 ↔ 1	8. $X \cdot \overline{X} = 0$	
	9. $\overline{\overline{X}} = X$	Complementing is not changed		Order of inputs is irrelevant
Two or more literals	10. $X + Y = Y + X$		11. $XY = YX$	Commutative
	12. $X+Y+Z = (X + Y) + Z = X + (Y + Z)$		13. $XYZ = (XY)Z = X(YZ)$	Associative
	14. $X(Y + Z) = XY + XZ$		15. $X + YZ = (X + Y)(X + Z)$	Distributive
	16. $\overline{X + Y} = \overline{X} \cdot \overline{Y}$		17. $\overline{X \cdot Y} = \overline{X} + \overline{Y}$	DeMorgan's

This Does **not** hold in ordinary Algebra: e.g. 5+(3*4) ≠ (5+3)*(5+4)

Associativity: An n-input operation can be performed as a sequence of 2-input operations in any order, e.g. a 3-input OR

Some Properties of Identities & the Algebra

- If the meaning is unambiguous, we leave out the symbol "·"
- The identities above are organized into pairs. These pairs have names as follows:

1-4 Existence of 0 and 1	5-6 Idempotence
7-8 Existence of complement	9 Involution
10-11 Commutative Laws	12-13 Associative Laws
14-15 Distributive Laws	16-17 DeMorgan's Laws

- The dual of an algebraic expression is obtained by interchanging + and · and interchanging 0's and 1's.
- The identities appear in dual pairs. When there is only one identity on one line the identity is self-dual, i. e., the dual expression = the original expression, e.g. No. 9.

Some Properties of Identities & the Algebra **(Continued)**

- **Unless it happens to be self-dual, the dual of an expression does not equal the expression itself.**
- **Example:** $F = (A + \overline{C}) \cdot B + 0$

  dual $F = ((A \cdot \overline{C}) + B) \cdot 1 = A \cdot \overline{C} + B$
- **Example:** $G = X \cdot Y + (\overline{W + Z})$

  dual $G = (X+Y) . (\overline{WX})$
- **Example:** $H = A \cdot B + A \cdot C + B \cdot C$

  dual $H = (A+B) . (A+C) . (B+C)$

When taking the dual, Complementing is not changed

- **Are any of these functions self-dual?**

  Check if truth tables for (F) and (dual F) are identical

Boolean Operator Precedence

- The order of evaluation in a Boolean expression is:
  1. Parentheses
  2. NOT
  3. AND
  4. OR
- Consequence: Put parentheses around OR expressions when they have to be evaluated first
- Example: $F = E + A(B + C)(\overline{C} + D)$

Boolean Algebraic Proofs: Example

Show algebraically that the LHS is logically equivalent to the RHS
i.e. will have same truth table

- $(\overline{X + Y})Z + X\overline{Y} = \overline{Y}(X + Z)$

Proof Steps	**Justification (identity # or theorem)**
$(\overline{X + Y})Z + X\overline{Y}$	
$= (\overline{X}\,\overline{Y})\,Z + X\,\overline{Y}$	16 (DeMorgan's)
$= \overline{Y}\,(X + \overline{X}Z)$	10, 14
$= \overline{Y}\,[(X + \overline{X})\,(X + Z)]$	7, 10
$= \overline{Y}\,(X + Z)$	

Verify equivalence of 1 and 2
By comparing the truth tables

Compare circuit costs of both sides
to show Benefit of simplification

Useful **Theorems** (in Dual forms)

Expression **Dual**

- $x \cdot y + \overline{x} \cdot y = y$ $\quad (x + y)(\overline{x} + y) = y$ **Minimization**
- $x + x \cdot y = x$ $\quad x \cdot (x + y) = x$ **Absorption**
- $x + \overline{x} \cdot y = x + y$ $\quad x \cdot (\overline{x} + y) = x \cdot y$ **Simplification**
- $x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z$ **Consensus**
  $(x + y) \cdot (\overline{x} + z) \cdot (y + z) = (x + y) \cdot (\overline{x} + z)$
- $\overline{x + y} = \overline{x} \cdot \overline{y}$ $\quad \overline{x \cdot y} = \overline{x} + \overline{y}$ **DeMorgan's Laws**

Proof of **Minimization**

$$x \cdot y + \overline{x} \cdot y = y \quad (x + y)(\overline{x} + y) = y$$

- **Consider the LHS form**

$x\, y + \overline{x}\, y = y\, (x + \underbrace{\overline{x}) }_{1}= y$

Proof of **Absorption**

- A + A·B = A (Absorption Theorem)
  i.e. B is irrelevant (redundant, absorbed) in this expression!

Proof Steps	Justification (identity or theorem)
A + A·B	
= A · 1 + A · B	X = X · 1
= A · (1 + B)	X · Y + X · Z = X ·(Y + Z)(Distributive Law)
= A · 1	1 + X = 1
= A	X · 1 = X

- Our primary reason for doing proofs is to learn:
  - Careful and efficient use of the identities and theorems of Boolean algebra, and
  - How to choose the appropriate identity or theorem to apply to make forward progress, irrespective of the application.

Proof of **Simplification**

$$x + \overline{x} \cdot y = x + y \qquad x \cdot (\overline{x} + y) = x \cdot y$$ **Simplification**

- **Consider the LHS form**

$$x + \overline{x}\, y = (x + \overline{x})\,(x + y)$$
$$= 1.\,(x + y)$$
$$= (x + y)$$

Proof of Consensus

- $AB + \overline{A}C + BC = AB + \overline{A}C$ (Consensus Theorem)

Proof Steps X Justification (identity # or theorem)

Proof Steps	Justification (identity # or theorem)
$AB + \overline{A}C + BC$	
$= AB + \overline{A}C + 1 \cdot BC$	2
$= AB + \overline{A}C + (A + \overline{A}) \cdot BC$	7
$= AB + \overline{A}C + ABC + \overline{A}BC$	11, 14
$= AB + ABC + \overline{A}C + \overline{A}BC$	12
$= AB(1+C) + \overline{A}C(1+B)$	14
$= AB + \overline{A}C$	3, 2

Proof of DeMorgan's Laws $\overline{x \cdot y} = \overline{x} + \overline{y}$

Given the Basic Identities X X' = 0 and X + X' = 1,
we can prove any theorem Y = X, if we can show that X Y' = 0 and X + Y' = 1,

DeMorgan's Theorem states that: (A B)' = A' + B'
i.e. here Y = (A B)' and X = A' + B'

So we need to show that:
1. (A' + B') (A B)'' = (A' + B') (A B) = 0:
(A' + B') (A B) = A' A B + B' A B = A' A B + B' B A = 0 + 0 = 0 (Q.E.D.)

2. (A' + B') + (A B)'' = (A' + B') + (A B) = 1:
(A' + B') + (A B) = (A' + B') + (A B) + (A B) (since X + X = X)
= A' + AB + B' + AB
= (A'+A) (A'+B) + (B' + A) (B'+B)
= 1 (A'+B) + 1 (B'+ A)
= (A + A') + (B + B') = 1 + 1 = 1 (Q.E.D.)

AND-Invert = OR of inverts

DeMorgan's Laws $\overline{x \cdot y} = \overline{x} + \overline{y}$

Verification by Truth Tables:

x	y	x.y	(x.y)'	x'	y'	x' + y'
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Note: DeMorgan's is also valid for any number of variables

$\overline{A\,B\,C\,.....H} = \overline{A} + \overline{B} + \overline{C}\ .....+ \overline{H}$

Deriving the Truth Table of a Boolean Function

$F1 = xy\overline{z}$

$F2 = x + \overline{y}z$

$F3 = \overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,y\,z + x\overline{y}$

$F4 = x\overline{y} + \overline{x}\,z$

x	y	z	F1	F2	F3	F4
0	0	0	0	0		
0	0	1	0	1		
0	1	0	0	0		
0	1	1	0	0		
1	0	0	0	1		
1	0	1	0	1		
1	1	0	1	1		
1	1	1	0	1		

Function of 3 input variables

→ $2^3 = 8$ input combinations

→ Truth table has 8 rows

→ Table lists all possible combinations of the inputs and the corresponding output

Expression Simplification

Exercise: Verify Equivalence with Truth Table

- **An application of Boolean algebra**
- **Simplify to contain the smallest number of <u>literals</u> (complemented and uncomplemented variables):**

15 literals, 6 gates

$$AB + \bar{A}CD + \bar{A}BD + \bar{A}C\bar{D} + ABCD$$

$$= AB + ABCD + \bar{A}CD + \bar{A}C\bar{D} + \bar{A}BD$$

$$= AB + AB(CD) + \bar{A}C(D + \bar{D}) + \bar{A}BD$$

$$= AB + \bar{A}C + \bar{A}BD = B(A + \bar{A}D) + \bar{A}C$$

$$= B(A + D) + \bar{A}C$$

Simplification

5 literals, 4 (smaller gates)

Simpler Expressions → Fewer gates, Fewer gate inputs, and simpler circuits….This improves reliability and reduces power consumption

Complementing Functions

- **Use DeMorgan's Theorem to complement a function:**
  1. **Interchange AND and OR operators**
  2. **Complement each constant value and literal**
- **Example: Complement** $F = \bar{x}y\bar{z} + x\bar{y}\bar{z}$

$$\bar{F} = (x + \bar{y} + z)(\bar{x} + y + z)$$

Note: Here we used DeMorgan’s 3 times at two levels!

Verify Result Using Truth Tables

Complementing Functions, Contd.

- **Example: Complement $G = (\bar{a} + bc)\bar{d} + e$**

$$\bar{G} = [(\bar{a} + bc)\bar{d} + e]' = [(\bar{a} + bc)\bar{d}]'. e'$$
$$= [(a' + bc)' + d'']. e'$$
$$= [a''. (bc)' + d]. e'$$
$$= [a. (b'+c') + d]. e'$$
$$= ab'e' + ac'e' + de'$$

Verify Result Using Truth Tables

Towards a more systematic treatment….

3. Canonical Forms- Overview

- **What are Canonical Forms?**
- **Minterms and Maxterms**
- **Index Representation of Minterms and Maxterms**
- **Sum-of-Minterm (SOm) Representations**
- **Product-of-Maxterm (POM) Representations**
- **Representation of Complements of Functions**
- **Conversion between various Representations**

Canonical Forms

- **It is useful to specify a Boolean function in a form that:**
  - **Has a direct correspondence to the truth table**
  - **Allows comparison for equality**
- **Two main Canonical Forms in common use:**
  - **Sum of Minterms (SOm)**
  - **Product of Maxterms (POM)**

Minterms of n Variables

- **Minterms are AND (product) terms that contains ALL the inputs (each in either true or complemented form) which is equal to 1 for only one input combination and equal 0 otherwise**
- **Given that each binary literal may appear as normal (e.g., x) or complemented (e.g., $\overline{x}$), there are 2^n minterms for *n* variables.**
- **Example: Two variables (X and Y) produce $2^2 = 4$ combinations (i.e. 4 minterms):**

XY **(both normal, m = 1 only for 11)**
$X\overline{Y}$ **(X normal, Y complemented, m = 1 only for 10)**
$\overline{X}Y$ **(X complemented, Y normal, m = 1 only for 01)**
$\overline{X}\,\overline{Y}$ **(both complemented, m = 1 only for 00)**

Maxterms of n Variables

- **<u>Maxterms</u> are OR (sum) terms that contain all the input variables (each in either true or complemented form) which is equal to 0 for one input combination and equal 1 otherwise**
- **Given that each binary variable may appear as normal (e.g., x) or complemented (e.g., $\bar{x}$), there are 2^n maxterms for *n* variables.**
- **<u>Example:</u> Two literals (X and Y) produce $2^2 = 4$ combinations (i.e. 4 maxterms):**

$X+Y$ **(both normal, M = 0 only for 00)**

$X+\bar{Y}$ **(X normal, Y complemented, M = 0 only for 01)**

$\bar{X}+Y$ **(X complemented, Y normal, M = 0 only for 10)**

$\bar{X}+\bar{Y}$ **(both complemented, M = 0 only for 11)**

Maxterms and Minterms from the Truth Table

- **Example: minterms and Maxterms for Two Variables**

Input Combination | A product that gives 1 | A sum that gives 0

Index	xy	minterm	Maxterm
0	00	$\bar{x}\,\bar{y}$	$x + y$
1	01	$\bar{x}\,y$	$x + \bar{y}$
2	10	$x\,\bar{y}$	$\bar{x} + y$
3	11	$x\,y$	$\bar{x} + \bar{y}$

Complement

Index represents the Input combination in decimal

m_2 AND that gives 1

M_2 OR that gives 0

Reason for min and Max names? See slide 40

Note: m_i is the complement of M_i and vice versa, e.g. for m_2:

$\overline{x\,\bar{y}} = \bar{x} + y$ (Use Demorgan's Theorem)

Minterm and Maxterm Relationship

- **Review: DeMorgan's Theorem**
  $\overline{x \cdot y} = \overline{x} + \overline{y}$ **and** $\overline{x + y} = \overline{x} \cdot \overline{y}$
- **Two-literal example:**
  $M_2 = \overline{x} + y$ **and** $m_2 = x \cdot \overline{y}$
  Thus M_2 is the complement of m_2 and vice-versa.
- **Since DeMorgan's Theorem holds for *n* literals, the above holds for terms of *n* literals**
- **giving:**
  $$M_i = \overline{m_i} \text{ and } m_i = \overline{M_i}$$
  Thus M_i is the complement of m_i.

Truth Tables for minterms and Maxterms for two literals x, y

minterms

Index	x y	m_0 ($\overline{x}\,\overline{y}$)	m_1 ($\overline{x}\,y$)	m_2 ($x\,\overline{y}$)	m_3 ($x\,y$)
0	0 0	1	0	0	0
1	0 1	0	1	0	0
2	1 0	0	0	1	0
3	1 1	0	0	0	1

Maxterms

x y	M_0 ($x + y$)	M_1 ($x + \overline{y}$)	M_2 ($\overline{x} + y$)	M_3 ($\overline{x} + \overline{y}$)
0 0	0	1	1	1
0 1	1	0	1	1
1 0	1	1	0	1
1 1	1	1	1	0

- **Verify that m_i and M_i are complements of one another**
- **Observe how to derive the logic function for m_i and M_i from its index i expressed in binary, e.g. $m_2 = m_{10} = x\overline{y}$, $M_2 = M_{10} = \overline{x}+y$**
- **Reason for the names min and Max:**
  - **a minterm has a minimum of 1's in its truth table: Only one 1 while a Maxterm has a maximum of 1's in its truth table: 2^n-1 1's**

Standard order of variables

- Minterms and maxterms are designated with a subscript
- The subscript is a decimal number that represents the binary pattern of input literals in the straight binary (e.g. 8421) code
- The bits in the pattern represent the complemented or normal state of each literal listed in a standard fixed order (MSB…LSB)
- All input variables will be present in a minterm or maxterm and will be listed in the same order (usually alphabetically)

Standard Order → LSB

- **Examples of Standard forms: For 3 variables: a, b, c**
  - **Maxterms:** $(a + \bar{b} + c) = M_{010} = M_2$, $(\bar{a} + b + \bar{c}) = M_{101} = M_5$
  - **Minterms:** $a\ b\ \bar{c} = m_{110} = m_6$, $\bar{a}\ \bar{b}\ c = m_{001} = m_1$

Examples of non-standard forms for 3 variables:

- **Terms:** $(a + c)$, $b\ c$, and $(\bar{a} + b)$ **do not contain all literals**
- **Terms:** $(b + a + \bar{c})$, $a\ c\ b$, and $b\ c\ \bar{a}$ **not in standard order**

Index Example in Three literals: X, Y, and Z

Standard Order → LSB

- **The standard order is: X, then Y, then Z**
- **With Index 5 = $101)_2$** (XYZ)
  - **As a minterm (AND): Complement literals corresponding to 0 → $m_5 = X\bar{Y}Z$**
  - **As a Maxterm (OR): Complement literals corresponding to 1 → $M_5 = \bar{X}+Y+\bar{Z}$**

  - **$m_2 = m_{010} = ?$**
  - **$M_3 = M_{011} = ?$**
  - **$XY\bar{Z} = m_?$**
  - **$X+Y+\bar{Z} = M_?$**

Index Examples – Four literals

Index i	Binary Pattern	Minterm m_i	Maxterm M_i
0	0000	$\bar{a}\bar{b}\bar{c}\bar{d}$	$a+b+c+d$
1	0001	$\bar{a}\bar{b}\bar{c}d$	?
3	0011	?	$a+b+\bar{c}+\bar{d}$
5	0101	$\bar{a}b\bar{c}d$	$a+\bar{b}+c+\bar{d}$
7	0111	?	$a+\bar{b}+\bar{c}+\bar{d}$
10	1010	$a\bar{b}c\bar{d}$	$\bar{a}+b+\bar{c}+d$
13	1101	$ab\bar{c}d$	?
15	1111	$abcd$	$\bar{a}+\bar{b}+\bar{c}+\bar{d}$
	a b c d		

Verify using DeMorgan's

Minterm Function Example: 3 Variables XYZ

- **Truth Table for the Function** $F_1 = m_1 + m_4 + m_7$ (or)

$F1 = \bar{x}\ \bar{y}\ z + x\ \bar{y}\ \bar{z} + x\ y\ z$

And the truth table is:

Function is **1** at each of its specified minterms

So, given a truth table, How to determine the function?

→ As the sum of all minterms for which the function is 1 !....

x y z	index	m_1	+	m_4	+	m_7	$= F_1$
0 0 0	0	0	+	0	+	0	= 0
0 0 1	1	1	+	0	+	0	= 1
0 1 0	2	0	+	0	+	0	= 0
0 1 1	3	0	+	0	+	0	= 0
1 0 0	4	0	+	1	+	0	= 1
1 0 1	5	0	+	0	+	0	= 0
1 1 0	6	0	+	0	+	0	= 0
1 1 1	7	0	+	0	+	1	= 1

Maxterm Function Example

- **Example: Implement F1 in maxterms:**

$F_1 = M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6$ (and)

$$F_1 = (x + y + z)\cdot(x + \overline{y} + z)\cdot(x + \overline{y} + \overline{z})\cdot(\overline{x} + y + \overline{z})\cdot(\overline{x} + \overline{y} + z)$$

And the truth table is:

Function is **0** at each of its specified maxterms

x y z	i	M_0	· M_2	· M_3	· M_5	· M_6	= F1
0 0 0	0	0	· 1	· 1	· 1	· 1	= 0
0 0 1	1	1	· 1	· 1	· 1	· 1	= 1
0 1 0	2	1	· 0	· 1	· 1	· 1	= 0
0 1 1	3	1	· 1	· 0	· 1	· 1	= 0
1 0 0	4	1	· 1	· 1	· 1	· 1	= 1
1 0 1	5	1	· 1	· 1	· 0	· 1	= 0
1 1 0	6	1	· 1	· 1	· 1	· 0	= 0
1 1 1	7	1	· 1	· 1	· 1	· 1	= 1

So, given a truth table, How to determine the function?

→ **As the product of all maxterms for which the function is 0** !….

Observations from the Truth Tables

- **In the function tables:**
  - **Each minterm has one and only one 1 present in the 2^n rows (a minimum of 1s). All other entries are 0**
  - **Each maxterm has one and only one 0 present in the 2^n rows All other entries are 1 (a maximum of 1s)**
- We can implement any function by "ORing" the minterms corresponding to "1" entries in the function table. These are called the minterms of the function→ Sum of Minterms (SOM)
- We can implement any function by "ANDing" the maxterms corresponding to "0" entries in the function table. These are called the maxterms of the function→ Product of Maxterms (POM)
- **This gives us two canonical forms for a Boolean function:**
  - **Sum of Minterms (SOM)**
  - **Product of Maxterms (POM)**

Minterm Function Example: 5 literals

- F(A, B, C, D, E) = $m_2 + m_9 + m_{17} + m_{23}$
- 5 literals, so express each index as 5 bits
- F(A, B, C, D, E) =

  $m_{00010} + m_{01001} + m_{10001} + m_{10111}$
- F(A, B, C, D, E) in the SOM canonical form =

$$\bar{A}\,\bar{B}\,\bar{C}\,D\,\bar{E} + \bar{A}\,B\,\bar{C}\,\bar{D}\,E + A\,\bar{B}\,\bar{C}\,\bar{D}\,E + A\,\bar{B}\,C\,D\,E$$

- **Short-hand Form**

$$F(A,B,C,D,E) = \sum m(2,9,17,23)$$

Standard order of input literals → Sum minterms

Maxterm Function Example: 4 literals

- $F(A,B,C,D) = M_3 \cdot M_8 \cdot M_{11} \cdot M_{14}$
- F(A, B,C,D) = $M_{0011} \cdot M_{1000} \cdot M_{1011} \cdot M_{1110}$

$$= (A+B+\bar{C}+\bar{D}).\ (\bar{A}+B+C+D).\ (\bar{A}+B+\bar{C}+\bar{D}).\ (\bar{A}+\bar{B}+\bar{C}+D)$$

- **Short-hand Form**

$$F(A,B,C,D) = \Pi M(3,8,11,14)$$

Standard order of input literals → Product Maxterms

Observations on complementing and form Conversion

1. Complementing a function

$$\overline{F}(x,y,z) = \Pi_M(1,3,5,7)$$

Complement Function ⟵ Other Form, Same Indices

$$F(x,y,z) = \Sigma_m(1,3,5,7)$$ ⟵ **Start**

Complement Function ⟵ Same Form, Complement Indices

$$\overline{F}(x,y,z) = \Sigma_m(0,2,4,6)$$

2. Form Conversion for the same function

$$F(x,y,z) = \Pi_M(0,2,4,6)$$

Same Function ⟵ Other Form, Complement Indices

$$F(x,y,z) = \Sigma_m(1,3,5,7)$$

Standard (as opposed to canonical) Forms

- **Standard Sum-of-Products (SOP) form: equations are written as ORing of Products (not minterms)**
- **Standard Product-of-Sums (POS) form: equations are written as ANDing of Sums (not maxterms)**
- **Examples: For 3 variables A,B,C**
  - **SOP:** $BC + \overline{A}\,\overline{B}C + B$
  - **POS:** $(A+B)\cdot(A+\overline{B}+\overline{C})\cdot C$

  Standard, Still 2-level Form
- **The following "mixed" forms are neither SOP nor POS**
  - $(AB + C)(A + C)$
  - $AB\overline{C} + AC(A + B)$

  Non-Standard, > 2-level Form

i.e. these are not in the standard 2-level from

Transforming Standard to Canonical SOm
1. Algebraically

- Any Boolean function can be expressed as a <u>Sum of Minterms</u>
  - From the function's truth table, the <u>minterms</u> used are the terms corresponding to the 1's of the function From expression, <u>expand</u> all terms first to explicitly include all minterms

    → Do this by "ANDing" any term missing variable v with a term ($v + \overline{v}$) (=1) (Easier way with K-maps later)
- Example: Express $f = x + \overline{x}\ \overline{y}$ as sum of minterms

  First expand terms: $f = x(y + \overline{y}) + \overline{x}\ \overline{y}$

  Then distribute terms: $f = xy + x\overline{y} + \overline{x}\ \overline{y}$

  Note: Complement in Minterm → Var is 0

  Express as sum of minterms: $f = m_{11} + m_{10} + m_{00}$

  $= m_3 + m_2 + m_0$

Transforming Standard to Canonical SOm
2. Using the Truth Table

- Example: $F = A + \overline{B}\,C$
- There are three variables, A, B, and C which we take to be the standard order
- Construct the truth table for the function
- Minterms are the standard terms where the function is 1
- For minterms we complement a literal when it is 0
- F (A,B,C) = $m_1 + m_4 + m_5 + m_6 + m_7$

  $= \overline{A}\,\overline{B}C + A\overline{B}\,\overline{C} + A\overline{B}C + AB\overline{C} + ABC$
- In the **standard short hand form**:

$$F(A,B,C) = \Sigma_m(1,4,5,6,7)$$

Standard order of input literals — Sum — minterms

Truth Table for F

A	B	C	Index	F
0	0	0	0	0
0	0	1	1	1
0	1	0	2	0
0	1	1	3	0
1	0	0	4	1
1	0	1	5	1
1	1	0	6	1
1	1	1	7	1

Exercise: Do by including missing variables as on previous slide – should get same result

Transforming Standard to Canonical SOm
3. Algebraically, again!

- Example: $F = A + \overline{B}\,C$ F has three input variables; A, B and C → any term in F missing one variable, corresponds to four minterms, and terms that are missing one corresponds to two minterms! So looking at $F = A + \overline{B}\,C$

Missing two variables; **B** and **C**

$A\_\,\_$ → put all combinations of **B** and **C**
→ $A\overline{B}\overline{C}, A\overline{B}C, AB\overline{C}, ABC$
m_4 m_5 m_6 $m7$

Missing **A**

$\_\overline{B}C$ → put all combinations of **A**
→ $\overline{A}\overline{B}C, ABC$
m_1 m_7

Hence F (A,B,C) = $m_1 + m_4 + m_5 + m_6 + m_7$ (do not repeat redundant minterms)

$$F(A,B,C) = \Sigma_m(1,4,5,6,7)$$

Standard order of input literals — Sum — minterms

Transforming Standard to Canonical POM
1. Algebraically

- **Any Boolean Function can be expressed as a <u>Product of Maxterms (POM)</u>**
  - From function table, the maxterms used are the terms corresponding to the **0's** of the function
  - From function expression, Expand all terms to explicitly include all maxterms by: 1. Applying the second distributive law
    2. "ORing" terms missing literal v with a term equal to $v \cdot \overline{v}$ (=0) and then applying the distributive law again
- **Example: Convert to product of maxterms:**

$$f(x,y,z) = x + \overline{x}\,\overline{y}$$

Variable z is missing in expression

Apply the distributive law:

$$x + \overline{x}\,\overline{y} = (x + \overline{x})(x + \overline{y}) = 1 \cdot (x + \overline{y}) = x + \overline{y}$$

Not a maxterm

Introduce missing literal z by ORing with $z \cdot \overline{z}$:

Add a 0 to an OR

$$(x + \overline{y}) + z \cdot \overline{z} = (x + \overline{y} + z)(x + \overline{y} + \overline{z})$$

Express as POM: f = $M_{010} \cdot M_{011}$

= $M_2 \cdot M_3$

Note: Complement in Maxterm → Var is 1

Transforming Standard to Canonical POM
2. Using the Truth Table

- Example: $F = A + \overline{B}C$
- There are three variables, A, B, and C which we take to be the standard order
- Construct the truth table for the function
- Maxterms are the standard terms where the function is 0
- For Maxterms we complement a literal when it is 1
- F (A,B,C) = $M_0 \cdot M_2 \cdot M_3$
  = (A+B+C) . (A+$\overline{B}$+C) . (A+$\overline{B}$+$\overline{C}$)
- In the **standard short hand form**:

$$F(A,B,C) = \Pi_M(0,2,3)$$

Standard order of input literals — Product — Maxterms

Truth Table for F

A	B	C	Index	F
0	0	0	0	0
0	0	1	1	1
0	1	0	2	0
0	1	1	3	0
1	0	0	4	1
1	0	1	5	1
1	1	0	6	1
1	1	1	7	1

Exercise: Do by including missing variables as on previous slide – should get same result

Implementing the Complement of a Function

- For a function (F) expressed as a canonical sum of minterms, the complement of the function ($\overline{F}$) can be constructed as either:
  - A sum of **the minterms missing** in the given sum-of-minterms canonical form for F
  - A Product of the **Maxterms having the same indices**
- Example: Given $F(x, y, z) = \Sigma_m(1,3,5,7)$
- Then we have:

F is 1 for these indices

∴ $\overline{F}$ is 1 for the remaining indices

∴ $\overline{F}$ is 0 for the these indices

$$\overline{F}(x,y,z) = \Sigma_m(0,2,4,6)$$

$$\overline{F}(x,y,z) = \Pi_M(1,3,5,7)$$

Standard order of input literals — Product — Maxterms

Conversion Between the Two Canonical Forms

- To convert between sum-of-minterms and product-of-maxterms form (or vice-versa) we follow these steps:
  - Find the function complement by swapping terms in the list with terms not in the list
  - Change from products to sums, or vice versa

- Example: Given F as: $F(x,y,z) = \Sigma_m(1,3,5,7)$
- $\overline{F}$ in the same form is: $\overline{F}(x,y,z) = \Sigma_m(0,2,4,6)$
- $\overline{\overline{F}} = F$ in the other form is: this is the original function in the other form of $F(x,y,z) = \Pi_M(0,2,4,6)$

Logic Implementation of SOM form

- A sum of minterms (SOm) expression for a function of *n* variables can be written down directly from its truth table
  - Implementation of this form is a network of gates in two levels:
  - Level 1 consists of a maximum of (2^n – 1) identical AND gates, each with *n*-input and
  - Level 2 is a single OR gate (with a maximum of $2^n - 1$ inputs).
- This form can often be simplified to a smaller standard SOP expression (Fewer and smaller level 1 gates, smaller level 2 gate → smaller circuits)
- Two approaches to do this simplification:
  - Manipulations using Boolean Algebra
  - Graphical approach using Karnaugh maps (K-maps)

SOm → SOP Simplification Example
1. Using Boolean Algebra manipulations

- **Obtain sum of minterms from Truth Table:**
  $F(A,B,C) = \Sigma m(1,4,5,6,7)$
- **Write the minterms as algebraic expressions:**
  $F = \overline{A}\,\overline{B}\,C + A\,\overline{B}\,\overline{C} + A\,\overline{B}\,C + AB\overline{C} + ABC$ — 15 literals, 6 gates
- **Simplifying:**
  $F = \overline{A}\,\overline{B}\,C + A\overline{B} + AB$ ($A\overline{B} + AB$ = A)

  Simplification

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

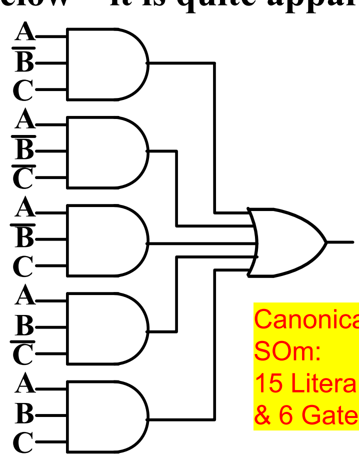
  $F = (A + \overline{A})\,(A + \overline{B}\,C) = A + \overline{B}\,C$ — 3 literals, 2 (smaller) gates

  Standard or not?
- **Simplification reduced circuit cost from (15,6) to only (3,2)**

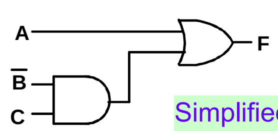
Canonical (SOm, POM) forms can be costly to implement. Luckily, they can be greatly simplified into standard (SOP, POS) forms

AND/OR Two-level Implementation of SOP Expression

- **The two implementations for F are shown below – it is quite apparent which is simpler!**



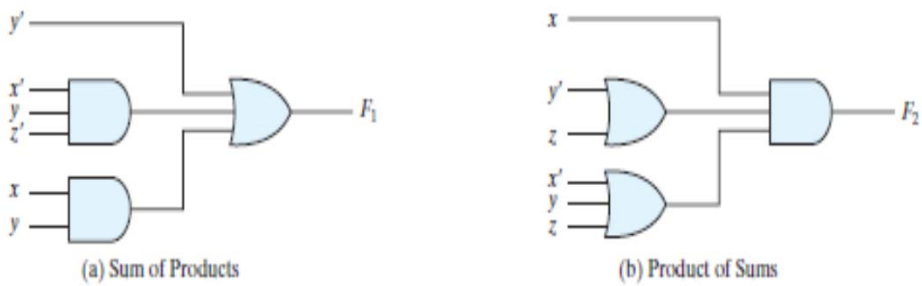
Canonical SOm: 15 Literal & 6 Gates



Simplified Standard SOP: 3 Literals & 2 (much simpler) Gates

Two Logic-Level Implementation of Standard Forms (SOP & POS)

- SOP → AND-OR Implementation
- POS → OR-AND Implementation



5. 2-Level Logic Circuit Optimization and K-maps

- Goal: To obtain the simplest implementation for a given function
- Optimization is a more formal approach to simplification.
- It is performed using a specific systematic procedure or algorithm as opposed to the **ad hoc approach** of algebraic manipulation
- Optimization requires a distinct cost criterion to measure the simplicity of a logic circuit
- Two useful cost criteria we will use:
  - Literal cost (L)
  - Gate input cost: (G)

Boolean Function Optimization

- Minimizing the gate input (or literal) cost of Boolean equations reduces circuit cost
- We will use the gate input G as the cost criterion
- Boolean Algebra and graphical techniques are tools to minimize cost criteria values
- Will cover optimum or near-optimum cost functions for two-level (SOP and POS) circuits
- Will Introduce a graphical optimization technique using Karnaugh maps (K-maps, for short)

Karnaugh Maps (K-map)

- A K-map is a collection of cells
  - Each cell represents a minterm
  - The collection of cells is a graphical representation of a Boolean function
  - **Adjacent** cells **differ in the value of one literal** only
  - Alternative algebraic expressions for the same function are derived by recognizing patterns of cells
- The K-map can be viewed as
  - A reorganized version of the truth table
  - A topologically-warped Venn diagram

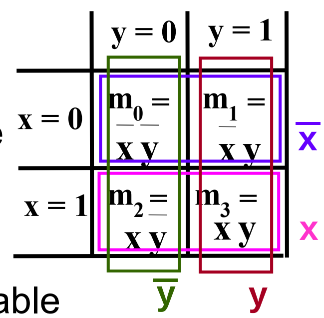
Some Uses of K-Maps

For functions with small numbers of literals, e.g. up to 5 literals:

- Finding optimum or near optimum implementations
  - SOP and POS standard forms
  - → Two-level AND/OR and OR/AND logic circuits
- Visualizing concepts related to manipulating Boolean expressions
- Demonstrating concepts used by computer-aided design programs to simplify larger circuits

K-Map for two variables (x,y)

- Minterm m_0 and minterm m_1 are “adjacent”
  - They differ in the value of the variable y
- Similarly, minterm m_0 and minterm m_2 differ in the x variable
- Also, m_1 and m_3 differ in the x variable
- Finally, m_2 and m_3 differ in the variable y
- Are m_0 and m_3 adjacent?



Each square represents an input Combination (index value) which Can possibly be a minterm for a function

K-Map and Truth Tables

- **The K-Map is just a different form of the truth table.**
- **Example – Two literal function:**
  - **For a given function F(x,y), output assumes values a,b,c and d from the set {0,1}**

Function Table

Input Combination, Minterms ↓

Input Values (x,y)	Function Value F(x,y)
0 0	a
0 1	b
1 0	c
1 1	d

Corresponding Function Output ↓

K-Map

	y = 0	y = 1
x = 0	[0] a	[1] b
x = 1	[2] c	[3] d

Enter function output at the box for the corresponding minterm

K-Map Function Representation

- **Example: F(x,y) = x**

F = x	y = 0	y = 1
x = 0	0	0
x = 1	1	1

- For function F(x,y), the two adjacent cells containing 1's can be **combined** using the Minimization Theorem:

$$F(x,y) = x\bar{y} + xy = x$$

- i.e. algebraic simplification is achieved **graphically** by simply combining **"adjacent"** cells as this allows omitting literals with different values

K-Map Function Representation

- **Example: G(x,y) = x + y**

G = x+y	y = 0	y = 1
x = 0	0	1
x = 1	1	1

- **For G(x,y), two pairs of adjacent cells containing 1's can be combined using the Minimization Theorem:**

$$G(x,y) = \left(x\,\overline{y} + x\,y\right) + \left(xy + \overline{x}\,y\right) = x + y$$

Duplicate xy

Three-literal Maps

- **A three-literal K-map:**

MSB

xyz is the Standard **order** of the literals

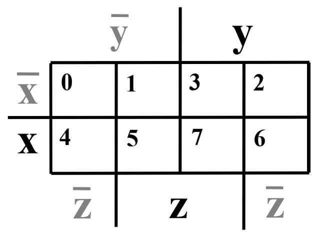
MSB	yz=00	yz=01	yz=11	yz=10
x=0	000 m_0	001 m_1	011 m_3	010 m_2
x=1	100 m_4	101 m_5	111 m_7	110 m_6

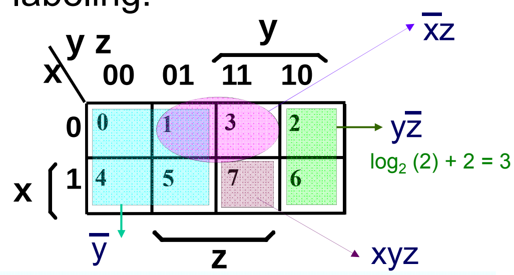
- The distribution of minterms on the K-map satisfies **logical adjacency** (note positions of m_3 and m_7).
- Note that m_2 is adjacent to m_0 and that m_6 is adjacent to m_4: Wrap-around effect
- Each minterm represents the corresponding product term:

	yz=00	yz=01	yz=11	yz=10
x=0	$\overline{x}\,\overline{y}\,\overline{z}$	$\overline{x}\,\overline{y}\,z$	$\overline{x}\,y\,z$	$\overline{x}\,y\,\overline{z}$
x=1	$x\,\overline{y}\,\overline{z}$	$x\,\overline{y}\,z$	$x\,y\,z$	$x\,y\,\overline{z}$

Alternative Map Labeling

- Will use maps for:
  - Entering function output values on the map
  - Reading off simplified product terms from the map
- Alternative useful map labeling:





# of literals in expression = Total # of variables - $\log_2$ (# of cells in rectangle)

- Which is the most complex expression? Is it a minterm? How many literals?, cells?
- Which is the simplest expression? How many literals?, cells?

Representing a Logic Function on the K-map

- By convention, we represent the minterms of F by a "1" in the map and leave the remaining cells blank
- Example:

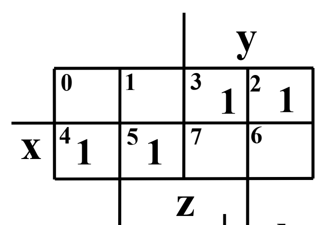
MSB LSB

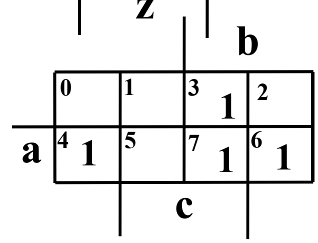
$$F(x,y,z) = \Sigma m(2,3,4,5)$$

- Example:

$$G(a,b,c) = \Sigma m(3,4,6,7)$$

- Learn the locations of the 8 indices based on the literal order shown (e.g. x, most significant and z, least significant) on the map boundaries





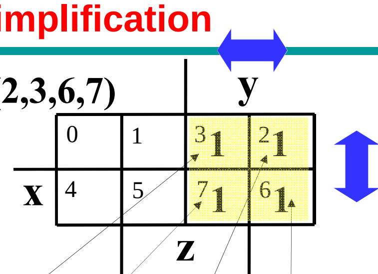
Combining cells

- By combining cells, we reduce number of literals in a product term, reducing the literal cost and the gate input cost
- On a 3-literal K-Map:
  - One cell represents a minterm with three literals
  - Two "adjacent" cells represent a product term with two literals
  - Four "adjacent" terms represent a product term with one literal
  - Eight "adjacent" terms is the function of all ones (zero literals – but here output is not a function of the inputs)

# of literals in expression = Total # of variables - $\log_2$ (# of cells in rectangle)

Example: Simplifying by **Combining** cells
Graphical Vs Boolean Simplification

- **Example: Let** $F = \Sigma m(2,3,6,7)$



- **Applying the Minimization Theorem three times:**

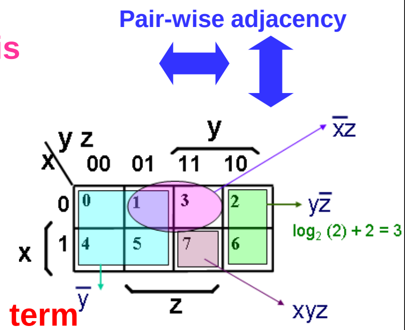
$$F(x,y,z) = \bar{x}\,y\,z + x\,y\,z + \bar{x}\,y\,\bar{z} + x\,y\,\bar{z}$$
$$= yz + y\bar{z}$$
$$= y$$

- **Thus the four terms that form a 2 × 2 cell correspond to the term "y".**

Rules for combining cells to larger rectangles

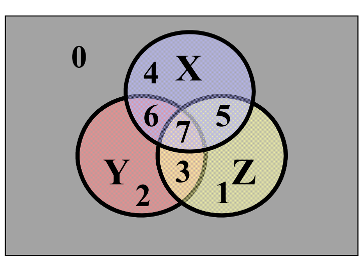
- → Combine only "pair-wise adjacent" cells
- → Combine cells only up to a rectangle/square with a size that is a power of 2 cells.

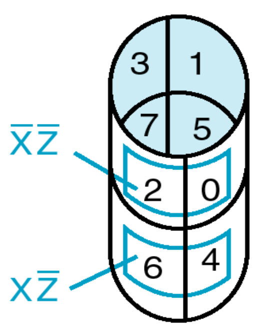
  For 3 variables, this means:
  - 2^0 = 1 cell (3 literals)
  - 2^1 = 2 cells (2 literals)
  - 2^2 = 4 cells (1 literal)
- **Check:** Result of combination should give only a single product term
- A grouping can include cells that are not directly adjacent, but are related together through pair-wise adjacency, e.g. cells 1 (001) and 4 (100)

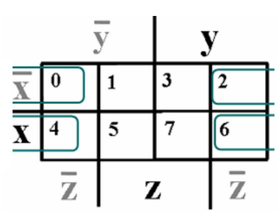


Three-literal Maps

- **Topological warps of 3-literal K-maps that show *all* adjacencies:**
  - **Venn Diagram**
  - **Cylinder**
  - **2-D K-Map**





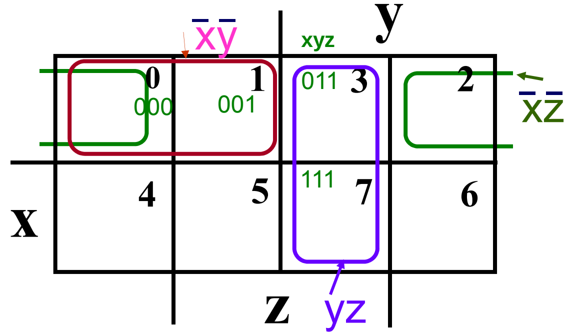


Standard order: XYZ
MSB

Adjacency needs Common line Boundary,
e.g. (7 and 3,5,6). 3 is not adjacent to 5 or 6

Three-literal Maps

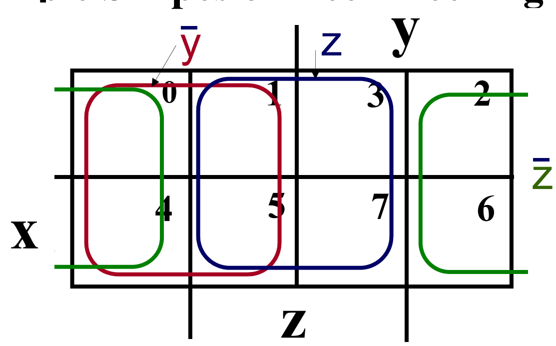
- **Example Shapes of valid 2-cell groupings:**



- Two Ways to read off the product term for a rectangle shown:
  1. Express the joint area on the map (Venn diagram mentality)
  2. The product includes each variable that has the same value in all cells of the rectangle. A variable that is equally divided between 1 and 0 in the cells of the rectangle is **excluded**

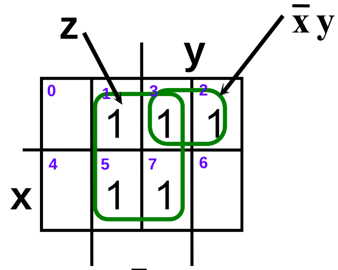
Three-literal Maps: 4-Cell Groupings

- **Example Shapes of 4-cell Rectangles:**



Function Simplification with a 3-literal Maps

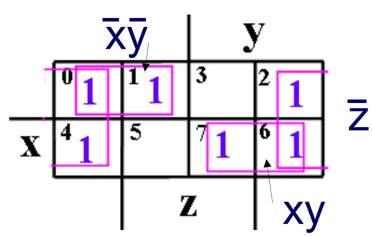
- K-Maps can be used as a systematic method to **simplify** Boolean functions. Cells are combined to form a set of the largest possible pair-wise adjacent rectangles/squares that **cover all the "1s"** of the function
- Example: Simplify $F(x, y, z) = \Sigma_m(1,2,3,5,7)$



$$F(x, y, z) = z + \bar{x}\,y$$

Function Simplification with a 3-literal Maps

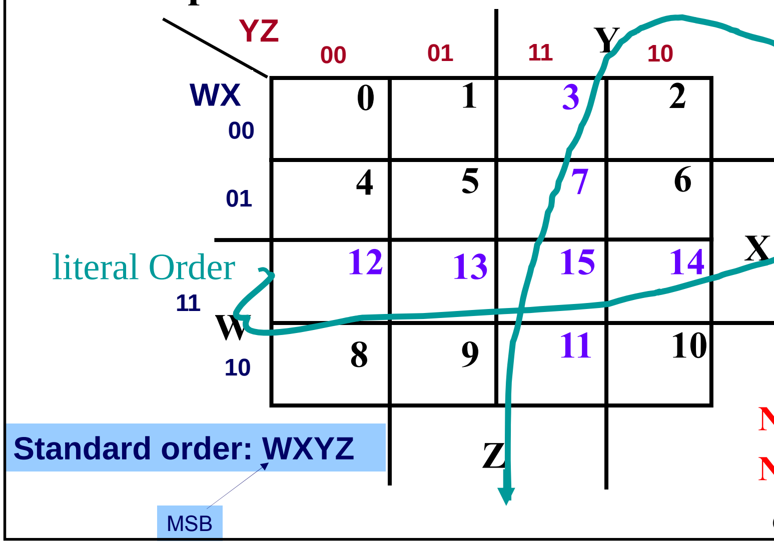
- **Use a K-map to find an optimum SOP equation for $F(X, Y, Z) = \Sigma_m(0,1,2,4,6,7)$**



$$F(x, y, z) = \bar{z} + x\,y + \bar{x}\,\bar{y}$$

Four-literal Maps

- **Map and location of minterms:**



Note Cell Numbering

Four literal Terms

- On four literal maps we can have rectangles corresponding to:
  - A single cell → 4 literals, (i.e. Minterm)
  - Two cells → 3 literals,
  - Four cells → 2 literals
  - Eight cells → 1 literal,
  - Sixteen cells → zero literals (i.e. Constant "1")

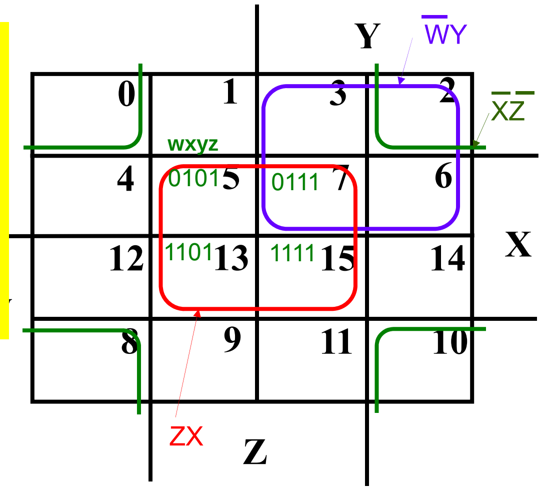
# of literals in expression = Total # of variables (4) - $\log_2$ (# of cells)

Four-literal Maps

- **Examples of valid 4-cell groupings:**

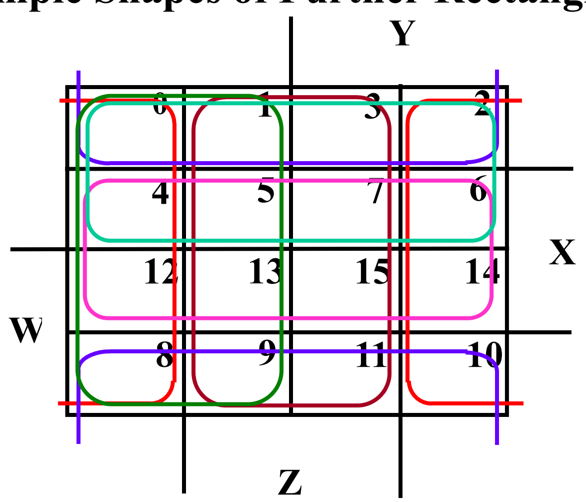
→ Rectangle should contain a power of 2 (< 2^4) group of pair-wise adjacent cells : 1, 2, 4, 8

Each rectangle should be expressible as a single product term



Four-literal Maps

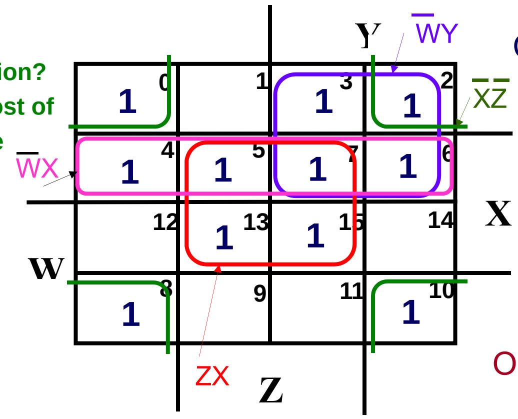
- **Example Shapes of Further Rectangles:**



Simplification with a Four-literal Map : Example 1

- $F(W, X, Y, Z) = \Sigma_m(0, 2,3,4,5,6,7,8,10,13,15)$

How effective Is our simplification?
L (# of literals) cost of Implementing the given Logic expression Is Reduced from ? To ?



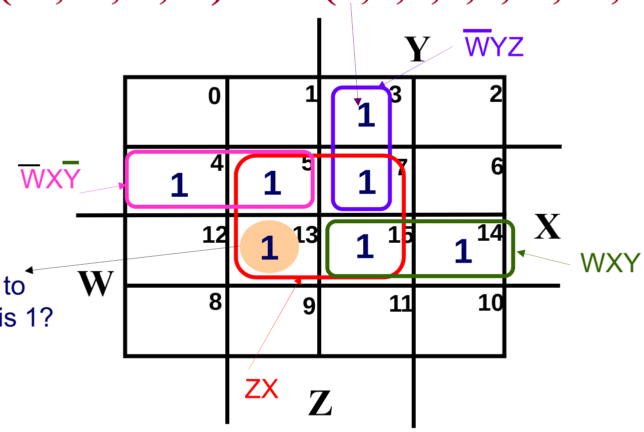
Canonical, SOm
L = ?

Optimized, SOP
L = ?

$$F(W, X, Y, Z) = ZX + \overline{W}Y + \overline{X}\overline{Z} + \overline{W}X$$

Simplification with a Four-literal Map : Example 2

- $F(W, X, Y, Z) = \Sigma_m(3,4,5,7,9,13,14,15)$



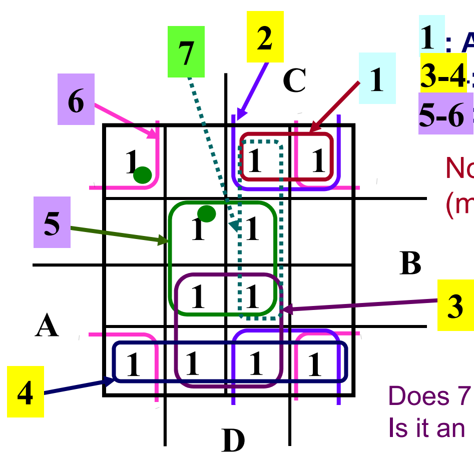
Best way to handle this 1?

$$F(W, X, Y, Z) = ZX + \overline{W}YZ + WXY + \overline{W}X\overline{Y}$$

Systematic Simplification (minimization) of a logic function: Implicants, Prime Implicants, and Essential Prime Implicants

- An ***Implicant*** is any single product term of a function obtained by combining a number of pair-wise adjacent "1" cells in the map into a rectangle with the number of cells a power of 2 (a minterm is the smallest implicant)
- A ***Prime Implicant* (PI)** is a single product term obtained by combining the maximum possible number of pair-wise adjacent cells in the map into a rectangle with the # of cells a power of 2 (can 1 cell be a PI?)
- A prime implicant is called an ***Essential Prime Implicant*** if it is the **only** prime implicant that covers (includes) one or more minterms (cells)
- Prime Implicants and Essential Prime Implicants can be determined by inspecting the K-Map.
- A set of prime implicants *"covers all minterms"* if, for each minterm of the function (i.e. 1 of the function), at least one prime implicant in the set includes that minterm….i.e. simply if No 1's are left out!

Examples of the three types of Implicants



1 : An implicant
3-4: Prime Implicants
5-6: Essential Prime Implicants

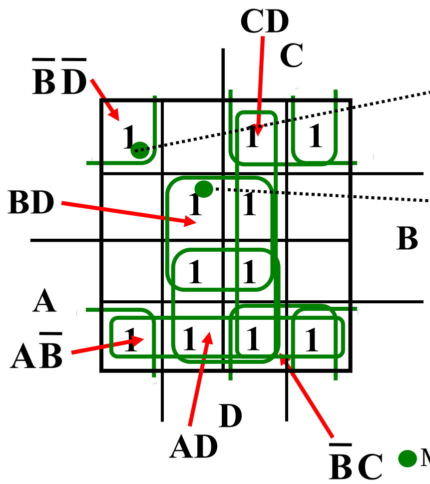
Note: Any single cell (minterm) is an implicant

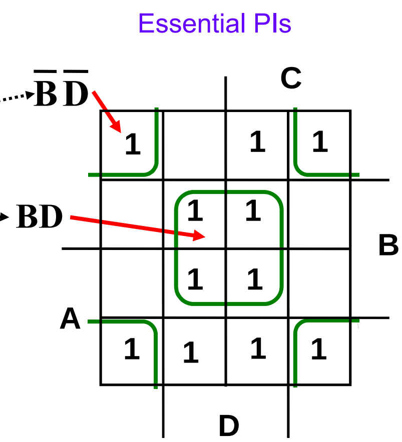
Give a situation that makes implicant 1 a prime implicant

Does 7 represent **a single** product term? Is it an implicant? Why?

● Minterm covered by only one prime implicant
So minterm 5 is ………?

Example: Find all Prime Implicants



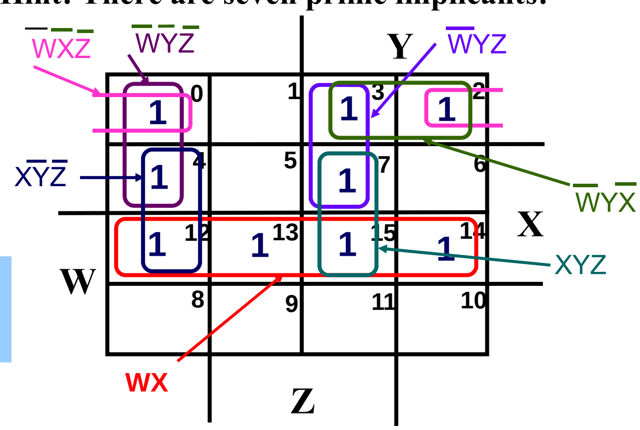


● Minterms covered by single prime implicant

Another Example

- **Find all possible prime implicants for:**
  $G(A, B, C, D) = \Sigma_m(0,2,3,4,7,12,13,14,15)$
  - **Hint: There are seven prime implicants!**

Not only those needed to cover all 1's

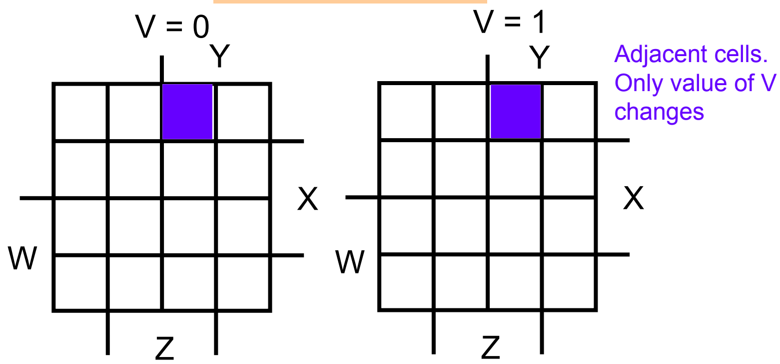


Any essential PIs?

K-Maps for five or more Variables

- For five literal problems (32 cells), we use *two **adjacent** K-maps*. It becomes harder to visualize adjacent minterms for selecting PIs.

F (W, X, Y, Z, V)



Don't Cares in K-Maps

- Sometimes a function table or map contains entries for which it is known that:
  - The input values for the minterm will never occur, e.g. with 4-bit (0-9) BCD codes (10-15 input values not used)
  - The output value of the function for that minterm will not be used
- In such cases, the output value of the function need not be defined as 1 or 0
- Instead, the output value is specified as a **"don't care"**
- By placing "don't cares" (labeled as an **"x"** entry) in the function table or map, the cost of the logic circuit may be reduced

Don't Cares in K-Maps

- **Example:** A logic function having the binary codes for the BCD digits as its inputs. Only the codes for 0 through 9 are used. The six codes, 1010 through 1111 never occur, so the output values for these codes are "x" to represent "don't cares"

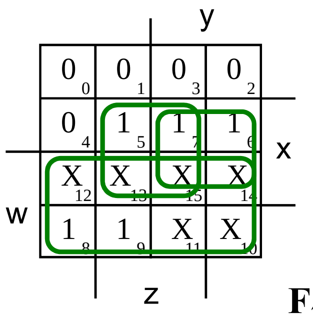
How can this help us minimize our circuits?

→ Each "x" entry may be given either a 0 or 1 value in resulting solution to an advantage

- For example, an "x" may be taken as "0" in an SOP solution or as "1" in a POS solution
- An "x" can be taken as 1 to maximize the size of a PI
- A cell with "x" needs not be covered by any prime implicant

Example: BCD "5 or More" (BCD codes 6,7,8,9)

- **The map below gives a function F1(w,x,y,z) which is defined as "5 or more" over BCD inputs. With the don't cares used for the 6 non-BCD input combinations:**



Function Output:
0 for input= 0 to 4
1 for input = 5 to 9
X (don't care) for input = 10-15

$$F1(w,x,y,z) = w + x\,z + x\,y$$

All X's = 1

- **This is much lower in cost than F2 where the "don't cares" were treated as "0"**

$$F_2(w, x, y, z) = \overline{w}\,x\,z + \overline{w}\,x\,y + w\,\overline{x}\,\overline{y}$$

All X's = 0

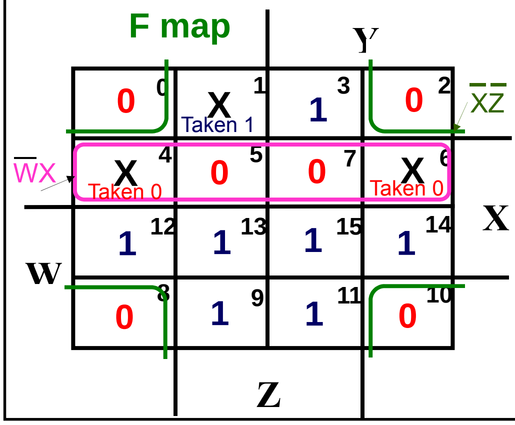
Product of Sums Example

- **Find the optimum POS solution for F, given:**

$$F(A,B,C,D) = \Sigma m(3,9,11,12,13,14,15) + \Sigma d(1,4,6)$$

Don't care

- **Hint: Use $\overline{F}$ and complement it to get the result**



- We still get the **SOP, but** for $\overline{F}$ by Constructing PIs containing its 1s (these are the 0s of F)

$\overline{F} = \overline{X}\,\overline{Z} + \overline{W}X$

- POS of F is obtained by Complementing $\overline{F}$ using DeMorgan's

$F = \overline{\overline{F}} = \overline{(\overline{X}\,\overline{Z} + \overline{W}X)}$

$= \overline{(\overline{X}\,\overline{Z})} \cdot \overline{(\overline{W}X)}$

$= (X+Z) \cdot (W+\overline{X})$

Algorithm for Systematic Optimization

1. Find all possible **prime** implicants (PIs)
2. From these PIs, select:

   → **All** essential PIs and mark all 1's covered by them

   → A minimum cost set of non-essential PIs that cover all minterms not yet covered by the essential PIs above

   - To obtain a **good** simplified solution: (not necessarily optimum), use the Selection Rule on next slide

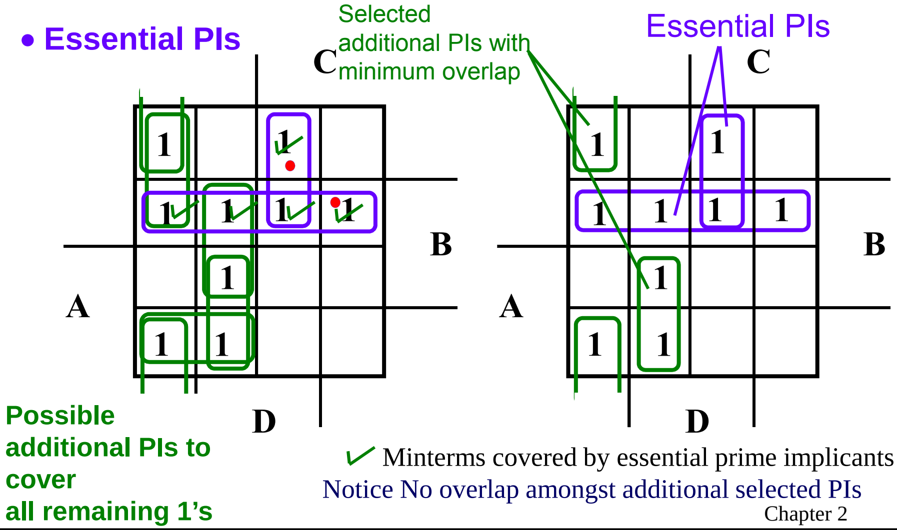
Prime Implicant Selection Rule

- Minimize the overlap among prime implicants as much as possible. In particular, in the final solution, make sure that each prime implicant selected includes at least one minterm not included in any other prime implicant selected

Note: Good solutions are not necessarily unique

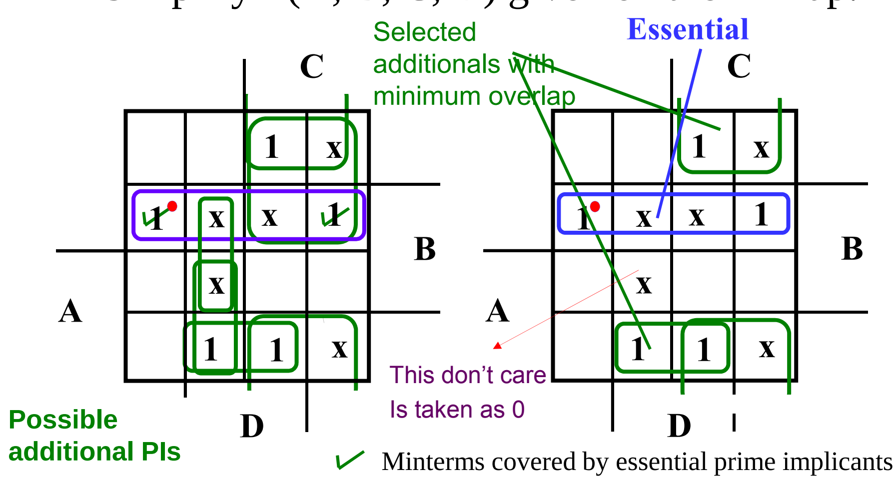
Example

- **Simplify F(A, B, C, D) given on the K-map**



Selection Rule Example with Don't Cares

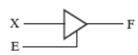
- Simplify F(A, B, C, D) given on the K-map.



6. Other Gate Types

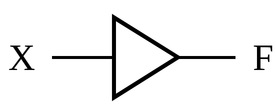
- Why?
  - Feasibility and cost of implementing the gate circuit in transistors
  - Potential for implementing any Boolean function using only a single gate type
  - Convenient conceptual representation
- Gate classifications
  - **Primitive gate** - a gate that can be described using a single primitive operation type (AND or OR) plus optional inversion(s), e.g. NAND
  - **Complex gate** - a gate that requires more than one primitive operation to describe it, e.g. XOR

Primitive gates

Name	Graphics Symbols: Distinctive shape	Algebraic equation	Truth table
AND		$F = XY$	X Y | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = X + Y$	X Y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT (inverter)		$F = \overline{X}$	X | F 0 | 1 1 | 0
Buffer		$F = X$	X | F 0 | 0 1 | 1
3-State Buffer			E X | F 0 0 | Hi-Z 0 1 | Hi-Z 1 0 | 0 1 1 | 1
NAND		$F = \overline{X \cdot Y}$	X Y | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = \overline{X + Y}$	X Y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0

Buffer

- A buffer is a gate with the function F = X:



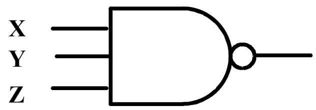
- In terms of Boolean logic, a buffer is the same as a direct connection!
- So why use it?

A buffer is an electronic amplifier that can be used to:

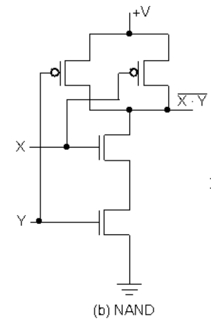
  - Improve circuit voltage levels e.g. of a received signal
  - Increase current drive capability (i.e. get a larger fan out)
  - Introduce desirable circuit delay

NAND Gate [NOT (AND)]

- The basic NAND gate has the following symbol, illustrated for three inputs:
  - AND-Invert (NAND)



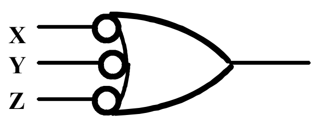
$F(X,Y,Z) = \overline{X \cdot Y \cdot Z}$



- NAND represents AND NOT, i. e., an AND function followed by an inverter (NOT). The symbol shown is an AND-Invert. The small circle ("bubble") represents the invert function.

NAND Gates (continued)

- Applying DeMorgan's Law gives Invert-OR (NAND)



$F(X,Y,Z) = \overline{X} + \overline{Y} + \overline{Z}$

- This NAND symbol is called Invert-OR, since inputs are inverted and then ORed together
- Note the above symbol is still for a NAND
- So a NAND gate can be represented in two different but equivalent forms: → AND-then-Invert form
  → Invert-then-OR form

$$F(X,Y,Z) = \overline{X \cdot Y \cdot Z} = \overline{X} + \overline{Y} + \overline{Z}$$

AND-Invert Invert-OR

Observations on the NAND Gate:
1. The NAND is not Associative

- NAND usually does not have an operation symbol defined like the "." for the AND and the "+" for the OR
- This is because NAND is not associative and we have difficulty dealing with non-associative arithmetic!:

$$Z = \overline{A \bullet B \bullet C} \neq Z = \overline{\overline{(A \bullet B)} \bullet C}$$

C	B	A	Z
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

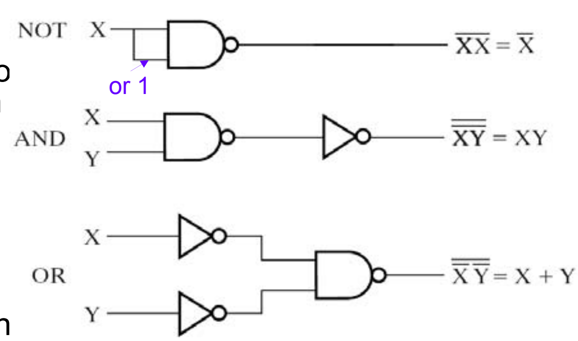
≠

C	B	A	Z
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

- i.e. the n-input NAND function can not be derived from a sequence of 2-input NAND operations
- But it can be derived as a sequence 2-input AND operation (which is associative) followed by a single final inversion

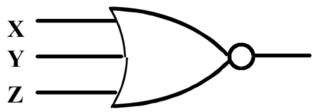
Observations on the NAND Gate:
2. The NAND is a Universal Gate

- Universal gate – is a gate that can be used to implement any Boolean function through implementing the 3 basic logic operations: (AND, OR, and NOT) (advantage)
- The NAND gate is a universal gate as shown opposite
- The NAND gate is the natural implementation for the simplest and fastest electronic circuits

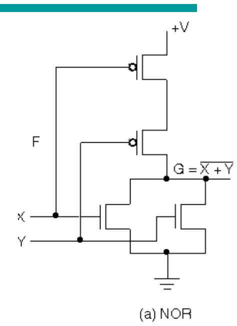


NOR Gate [NOT (OR)]

- The basic NOR gate has the following symbol, illustrated for three inputs:
  - OR-Invert (NOR)



$F(X,Y,Z) = \overline{X+Y+Z}$



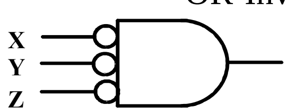
- NOR represents OR NOT, i. e., the OR function followed by a NOT. The symbol shown is an OR-Invert. The small circle ("bubble") represents the invert function.

NOR Gate (continued)

- Applying DeMorgan's Law gives Invert-AND (NOR)

$$F(X,Y,Z) = \overline{X+Y+Z} = \overline{X} \cdot \overline{Y} \cdot \overline{Z}$$

OR-Invert Invert-AND



$F(X,Y,Z) = \overline{X} \cdot \overline{Y} \cdot \overline{Z}$

- This NOR symbol is called Invert-AND, since inputs are inverted and then ANDed together.
- Note the above symbol is still for a NOR
- So a NOR gate can be represented in two different but equivalent forms: OR-then-Invert & Invert-then-AND

Observations on the NOR Gate: 1. The NOR gate is not Associative

- NOR usually does not have an operation symbol defined like the "." for the AND and the "+" for the OR
- This is because NOR is not associative and we have difficulty dealing with non-associative arithmetic!:

$Z = \overline{A+B+C}$ $\neq$ $Z = \overline{\overline{A+B}+C}$

C	B	A	Z
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

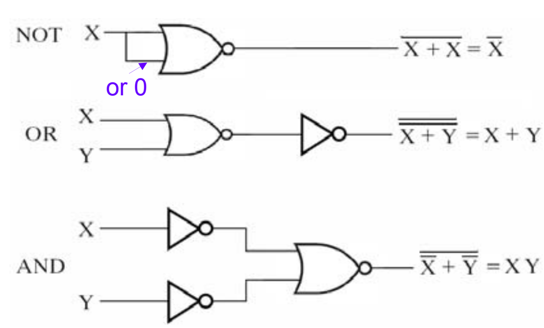
$\neq$

C	B	A	Z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

- i.e. the n-input NOR function can not be derived from a sequence of 2-input NOR operations
- But it can be derived as a sequence 2-input OR operation (which is associative) followed by a single final inversion

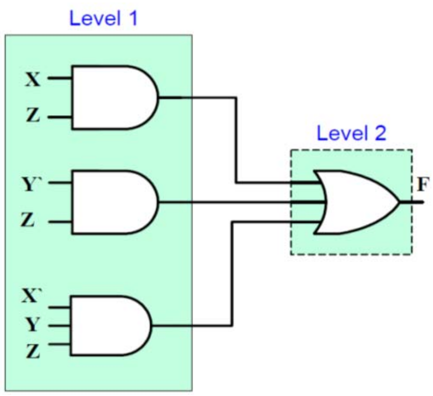
Observations on the NOR Gate: 2. The NOR is a Universal Gate

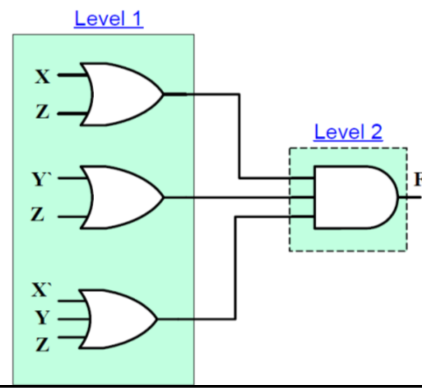
- The NOR gate is a universal gate as shown opposite
- The NOR gate is another natural implementation for the simplest and fastest electronic circuits



Two-Level Logic Implementation: Using AND & OR gates

- **For SOP forms: AND gates in the first level and a single OR gate in the second level.**
- **For POS forms OR gates will be in the first level and a single AND gate will be in the second level.**



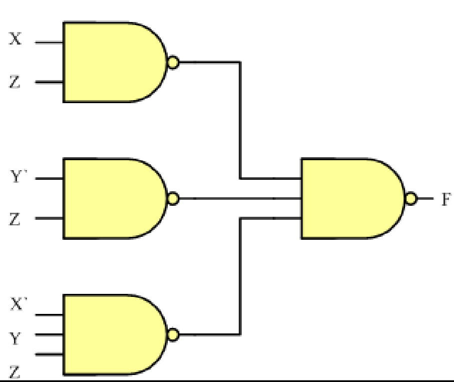


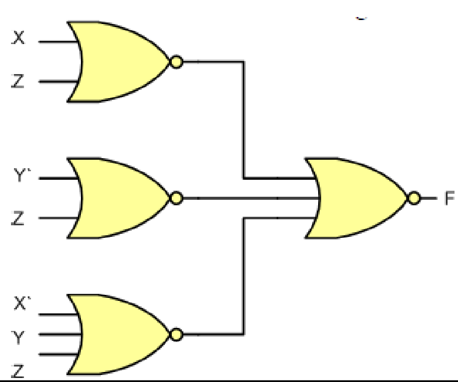
Two-Level Logic Implementation: Using Universal Gates NANDs & NORs

- **SOP forms can be implemented using two-logic levels of only NAND gates , while POS forms can be implemented using two-logic levels of only NOR gates**

F = XZ + Y'Z + X'YZ

F = (X+Z) (Y'+Z) (X'+Y+Z)



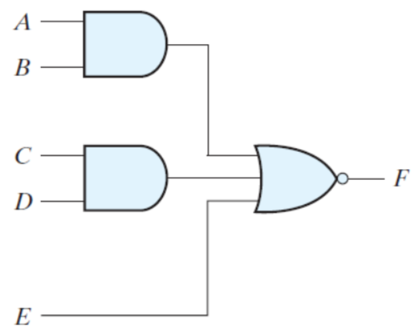


Other Two-Level Logic Implementation: AND-NOR

- If we express the function in AND-OR-Invert form, then it can be implemented directly as AND-NOR (AND gates for product terms and a NOR gate for Oring them and then inverting)
- To Obtain F in AND-OR-Invert Format: 1st Obtain F' in SOP by combining the 1's of F' in the K-map → then F is simply obtained by complementing the SOP expression of F' and we get the AND-OR-INVERT representation of F.

EX: F'= AB+CD+E →
F=(AB+CD+E)'

Then the AND-NOR is readily available
(OR-INVERT is simply NOR)

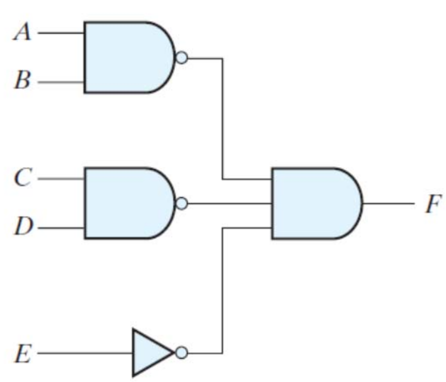


Other Two-Level Logic Implementation: NAND-AND

- The NAND-AND implementation is very similar to the AND-NOR -- We need to express the function in AND-OR-Invert form, then expand the complement one level to get the NAND-AND form directly:

EX: F'= AB+CD+E → F=(AB+CD+E)' = (AB)' (CD)' E'

Notice the single literals in F have inverters instead of NANDs

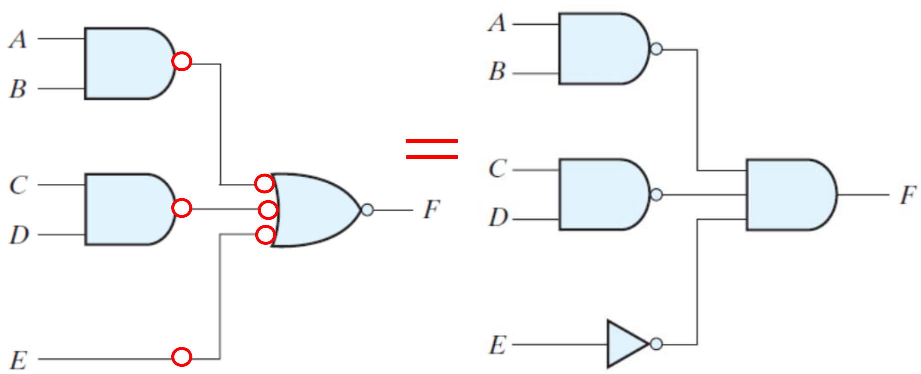


Other Two-Level Logic Implementation: NAND-AND

We could also have obtained the NAND-AND implementation from the AND-NOR through logic transformations: Inserting Bubbles in pairs!

AND-INVERT == NAND ,

INVERT-OR-INVERT = NAND-INVERT=AND



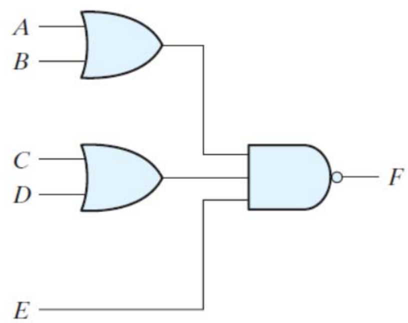
Other Two-Level Logic Implementation: OR-NAND

- If we express the function in OR-AND-INVERT form, then it can be implemented directly as OR-NAND (OR gates for SUM terms and a NAND gate for Anding them and then inverting)
- To Obtain F in OR-AND-Invert Format: 1st Obtain F' in POS by combining the 0's of F' in the K-map → then F is simply obtained by complementing the POS expression of F' and we get the OR-AND-INVERT representation of F.

EX: F' = (A+B) (C+D) E →

F = [(A+B) (C+D) E]'

Then the OR-NAND is readily available (AND-INVERT is simply NAND)

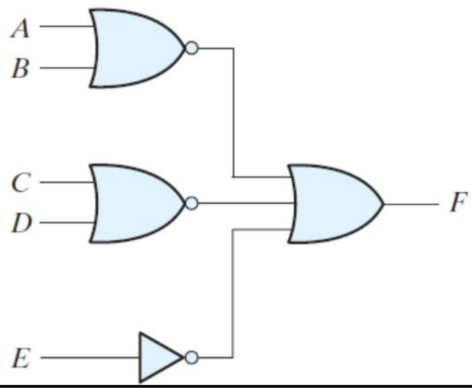


Other Two-Level Logic Implementation: NOR-OR

- The NOR-OR implementation is very similar to the OR-NAND; we need to express the function in OR-AND-INVERT form, then expand the complement one level to get the NOR-OR form directly.

EX: F’ = (A+B) (C+D) E → F = [(A+B) (C+D) E]’ = (A+B)’ + (C+D)’ + E’

Notice the single literals in F have inverters instead of NORs

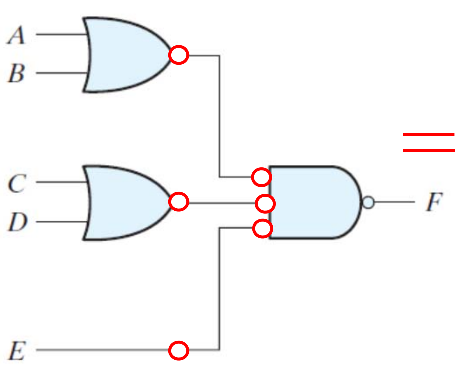


Other Two-Level Logic Implementation: NOR-OR

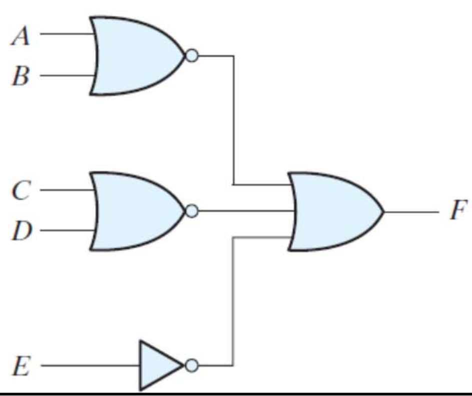
We could also have obtained the NOR-OR implementation from the OR-NAND through logic transformations: Inserting Bubbles in pairs!

OR-INVERT == NOR ,

INVERT-AND-INVERT = NOR-INVERT= OR



=



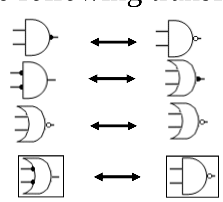
Other Two-Level Logic Implementation: Summary

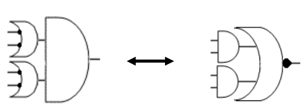
Equivalent Nondegenerate Form (a)	Equivalent Nondegenerate Form (b)*	Implements the Function	Simplify F' into	To Get an Output of
AND–NOR	NAND–AND	AND–OR–INVERT	Sum-of-products form by combining 0's in the map.	F
OR–NAND	NOR–OR	OR–AND–INVERT	Product-of-sums form by combining 1's in the map and then complementing.	F

*Form (b) requires an inverter for a single literal term.

Multi Logic-Implementation using NANDs & NORs

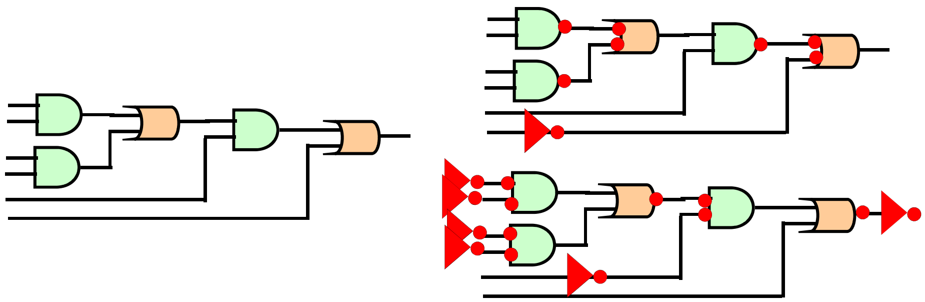
- ANY logic implementation could be converted to NAND-only or NOR-only implementation using the following transformations:
  - **AND-Invert ↔ NAND**
  - **Invert-AND ↔ NOR**
  - **OR-Invert ↔ NOR**
  - **Invert-OR ↔ NAND**
  - **Invert-AND-OR ↔ OR-AND-Invert**
  - **Invert-OR-AND ↔ AND-OR-Invert**



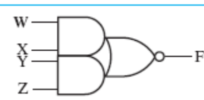
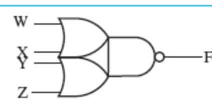
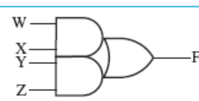
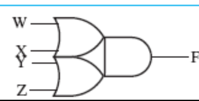


Multi Logic-Implementation using NANDs & NORs, *Contd.*

- **So when converting to NANDs:**
  - **Start from inputs, insert bubbles in pairs: at inputs of OR gates or at outputs of AND gates to convert them to NANDs**
- **And when converting to NORs:**
  - **Start from inputs, insert bubbles in pairs: at inputs of AND gates or at outputs of OR gates to convert them to NORs**



Complex gates

Name	Distinctive shape symbol	Algebraic equation	Truth table
Exclusive–OR (XOR)		$F = X\overline{Y} + \overline{X}Y = X \oplus Y$ XOR	X Y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
Exclusive–NOR (XNOR)		$F = XY + \overline{X}\overline{Y} = \overline{X \oplus Y}$ Inverted XOR = XNOR	X Y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1
AND-OR-INVERT (AOI)		$F = \overline{WX + YZ}$ Inverted SOP	
OR-AND -INVERT (OAI)		$F = \overline{(W + X)(Y + Z)}$ Inverted POS	
AND-OR (AO)		$F = WX + YZ$ SOP	
OR-AND (OA)		$F = (W + X)(Y + Z)$ POS	

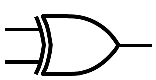
6. Complex Gates: Exclusive OR/ Exclusive NOR

- The *eXclusive OR* (*XOR*) function is an important Boolean function used extensively in arithmetic & communication circuits
- XOR is associative and is represented as the XOR **operator** (⊕)
- The *eXclusive NOR* (XNOR) function is the <u>complement</u> of the XOR function.
- XNOR is not associative
- By our definition, XOR and XNOR gates are complex gates
- The XOR/XNOR functions may be implemented:
  - Directly as an electronic circuit (a true gate) or
  - Indirectly by interconnecting other gate types (used as a convenient representation)

Definitions of XOR/XNOR as functions of 2 inputs: Truth Tables

XOR

- Sum
- Parity

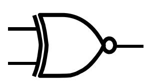


	$\overline{y}$	y
$\overline{x}$	0	1 **1**
x	2 **1**	3

	X	Y	X⊕Y
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	0

1 for **different** inputs

XNOR



	$\overline{y}$	y
$\overline{x}$	0 **1**	1
x	2	3 **1**

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

1 for **equal** inputs

- The XOR function means:

  X OR Y, but **NOT** BOTH
- XNOR is called the ***equivalence*** function, operator (≡): Why?
- From the K-maps:

  $X \oplus Y = X\overline{Y} + \overline{X}Y \Rightarrow \overline{X \oplus Y} = XY + \overline{X}\,\overline{Y}$ Prove Algebraically
- From eqns above, note that $\overline{X} \oplus Y = X \oplus \overline{Y} = \overline{X \oplus Y}$

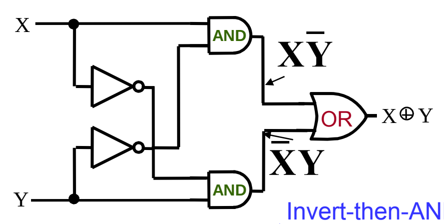
7. Exclusive OR/ Exclusive NOR

- Uses for the XOR and XNORs gate include:
  - Parity generators/checkers
  - Adders /subtractors
  - Counters/incrementers/decrementers
- Functions (see previous slide)
  - The XOR function is: $\mathbf{X \oplus Y = X\overline{Y} + \overline{X}Y}$
  - The eXclusive NOR (XNOR) function, otherwise known as *equivalence* is: $\mathbf{\overline{X \oplus Y} = X\,Y + \overline{X}\,\overline{Y}}$
- Strictly speaking, XOR and XNOR gates are defined only for **two inputs**. For more than two inputs, we use the terminology odd and even functions (considered later), respectively

XOR Implementations

- The simple SOP implementation uses the following structure:

$$\mathbf{X \oplus Y = X\,\overline{Y} + \overline{X}\,Y}$$



- **A NAND only implementation:**

Output of top AND :

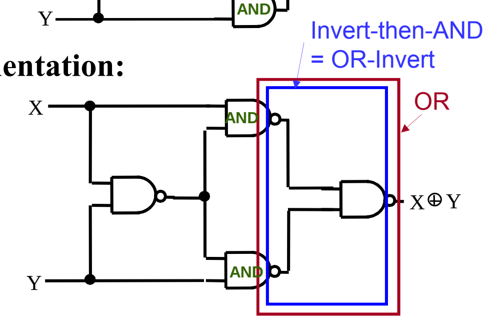
$= X.(\overline{XY})$

$= X\,(\overline{X} + \overline{Y})$

$= X\overline{X} + X\overline{Y}$

$= 0 + X\overline{Y}$

$= X\overline{Y}$ as above



XOR Identities

Derive from the truth table

XOR can be used as Controlled Inverter (1's Complementer)

$X \oplus 0 = X$; similar to OR

$X \oplus 1 = \overline{X}$; similar to NAND

$X \oplus X = 0$; Inputs are always identical

$X \oplus \overline{X} = 1$; Inputs are always different

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	0

Commutativity:

$$X \oplus Y = Y \oplus X$$

Associativity: Sequence of 2-input operations: **Yes!**

$$X \oplus Y \oplus Z = (X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$$

But for 3 or more inputs the function is called the odd function
(it is not called XOR)

XNOR Identities

Derive from the truth table

$\overline{X \oplus 0} = \overline{X}$; similar to NOR

$\overline{X \oplus 1} = X$; similar to AND

$\overline{X \oplus X} = 1$; Inputs are always identical

$\overline{X \oplus \overline{X}} = 0$; Inputs are always different

X	Y	F
0	0	1
0	1	0
1	0	0
1	1	1

Commutativity:

$$\overline{X \oplus Y} = \overline{Y \oplus X}$$

Demonstrate that XNOR is NOT associative

Associativity: Sequence of 2-input operations: **No!**

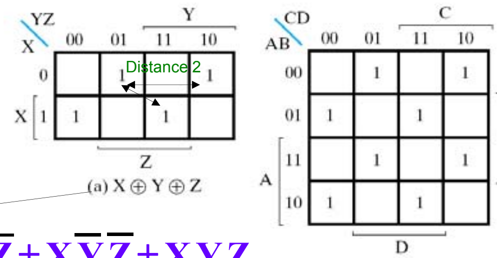
$$\overline{X \oplus Y \oplus Z} \neq \overline{\overline{(X \oplus Y)} \oplus Z} \neq \overline{X \oplus \overline{(Y \oplus Z)}}$$

For 3 or more inputs the function is called the even function
(it is not called XNOR)

XOR for >2 Variables: The Odd Function (for even parity generation and checking)

- The XOR function can be extended to 3 or more literals. For more than 2 literals, it is called:
  → An *odd function*, or
  → *modulo 2 sum*

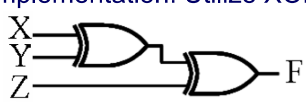
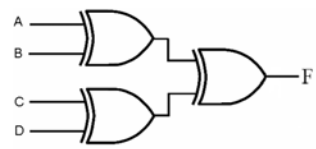
  The odd function for 3 inputs and 4 inputs



(a) $X \oplus Y \oplus Z$ (b) $A \oplus B \oplus C \oplus D$

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

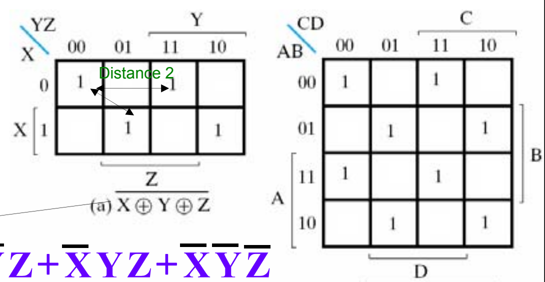
- 1s in the K-map correspond to minterms with indices having an odd number of 1s in binary, hence the name. Use to **generate** **even** parity bit and to **check** **even** parity (output = 1 for parity error)
- Implementation: Utilize XOR associatively

XNOR for >2 Variables: The Even Function (for odd parity generation and checking)

- The XONR function can be extended to 3 or more literals. For more than 2 literals, it is called:
  → An *Even function*

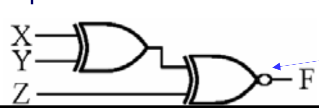
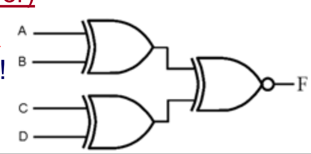
  The odd function for 3 inputs and 4 inputs



(a) $X \oplus Y \oplus Z$ (b) $\overline{A \oplus B \oplus C \oplus D}$

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

- 1s in the K-map correspond to minterms with indices having an even number of 1s in binary, hence the name. Use to **generate** **odd** parity bit and to **check** odd parity (output = 1 for parity error)
- Implementation: Utilize associatively of the XOR then invert!

Unit 2: Binary Logic and Gates Overview

1. Binary logic and gates, Boolean Algebra, Basic identities of Boolean algebra
2. Boolean functions, Algebraic manipulation, Complement of a function
3. Canonical & Standard forms, Minterms & Maxterms, Sum of products, Product of Sums. Algebraic simplification of logic functions
4. Physical properties of gates: Fan-in, Fan-out, Propagation Delay, HiZ (Tristate) outputs
5. Map method of logic circuit optimization:
   - Two-, Three-, and Four-literal K-Map
   - Optimization procedure: Essential prime implicants, Selected Additional prime implicants
   - Simplification with Don't care conditions
6. Other Gate Types: Universal gates (NAND and NOR), 2-level Complex gates (AO, AOI, OA. OAI)
7. Exclusive-OR (XOR) and Equivalence (XNOR) gates, Parity generation and checking