Logic and Computer Design Fundamentals

Chapter 2 – Combinational Logic Circuits

Part 2 – Circuit Optimization

Ming Cai

cm@zju.edu.cn

College of Computer Science and Technology
Zhejiang University

Overview

- **Part 1 – Gate Circuits and Boolean Equations**
  - Binary Logic and Gates
  - Boolean Algebra
  - Standard Forms
- **Part 2 – Circuit Optimization**
  - Two-Level Optimization
  - Map Manipulation
  - Multi-Level Circuit Optimization
- **Part 3 – Additional Gates and Circuits**
  - Other Gate Types
  - Exclusive-OR Operator and Gates
- **Part 4 – HDLs overview**
  - Logic Synthesis
  - HDL Representations—Verilog

Circuit Optimization

- **Goal: To obtain the simplest implementation for a given function**
- **Optimization is a more formal approach to simplification that is performed using a specific procedure or algorithm**
- **Optimization requires a cost criterion to measure the simplicity of a circuit**
- **Distinct cost criteria we will use:**
  - **Literal cost (L)**
  - **Gate input cost (G)**
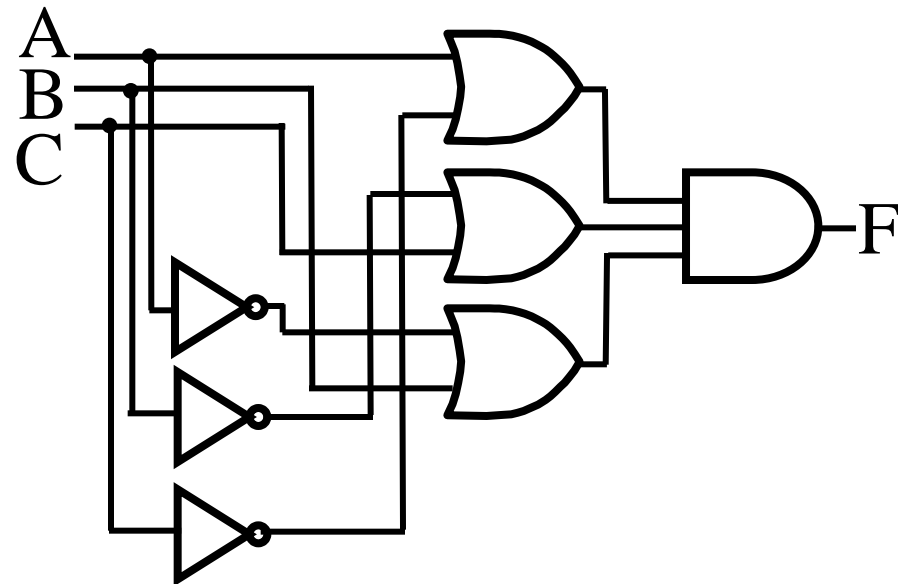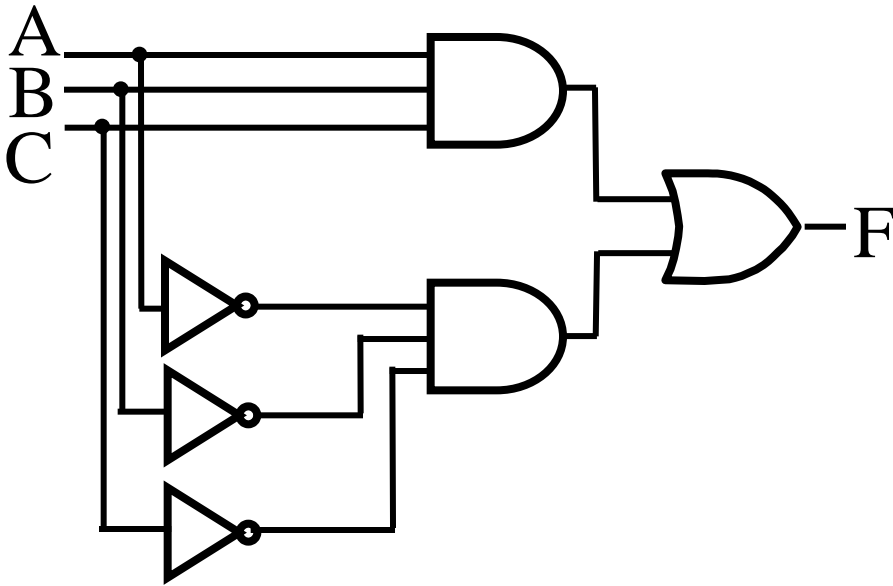 - **Gate input cost with NOTs (GN)**

Literal Cost

- **Literal – a variable or its complement**

- **Literal cost – the number of literal appearances in a Boolean expression corresponding to the logic circuit diagram**

- **Examples:**
  - $F = BD + A\overline{B}C + A\overline{C}\,\overline{D}$ $L = 8$
  - $F = BD + A\overline{B}C + A\overline{B}\,\overline{D} + AB\overline{C}$ $L =$
  - $F = (A + B)(A + D)(B + C + \overline{D})(\overline{B} + \overline{C} + D)$ $L =$
  - **Which solution is best?**

Literal Cost

- **Example (equivalent boolean expressions):**
  - $F = ABC + \overline{A}\,\overline{B}\,\overline{C}$ $L = 6$
  - $F = (A + \overline{C})(\overline{B} + C)(\overline{A} + B)$ $L = 6$
  - **Which solution is best?**

Gate Input Cost

- **Example 1:**
- $F = A + B\,C + \bar{B}\,\bar{C}$

$GN = G + 2 = 9$
$L = 5$
$G = L + 2 = 7$
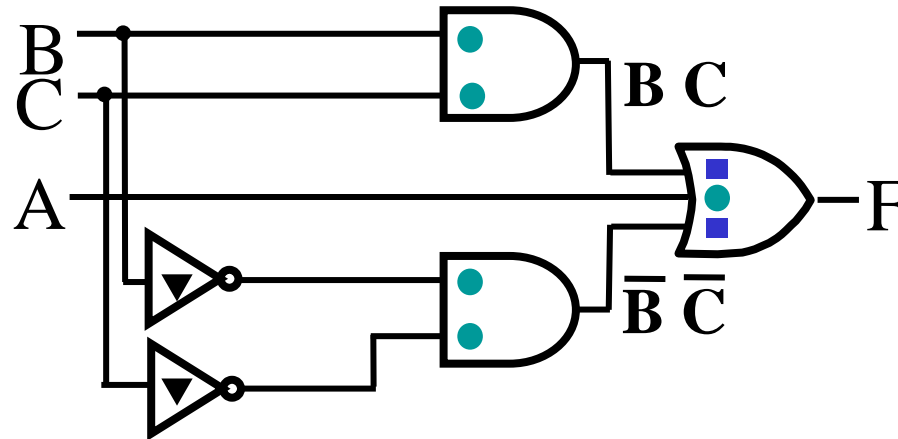
- **L (literal count) counts the AND inputs and the single literal OR input.**
- **G (gate input count) adds the remaining OR gate inputs**
- **GN(gate input count with NOTs) adds the inverter inputs**

Gate Input Cost (continued)

Example 2:

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

  L = 6   G = 8   GN = 11

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

  L = 6   G = 9   GN = 12

- **<u>Same</u> function and <u>same</u> literal cost**

- **But first circuit has <u>better</u> gate input count and <u>better</u> gate input count with NOTs**

- **Select it!**

L = 6

G = 8

GN = 11

L = 6

G = 9

GN = 12
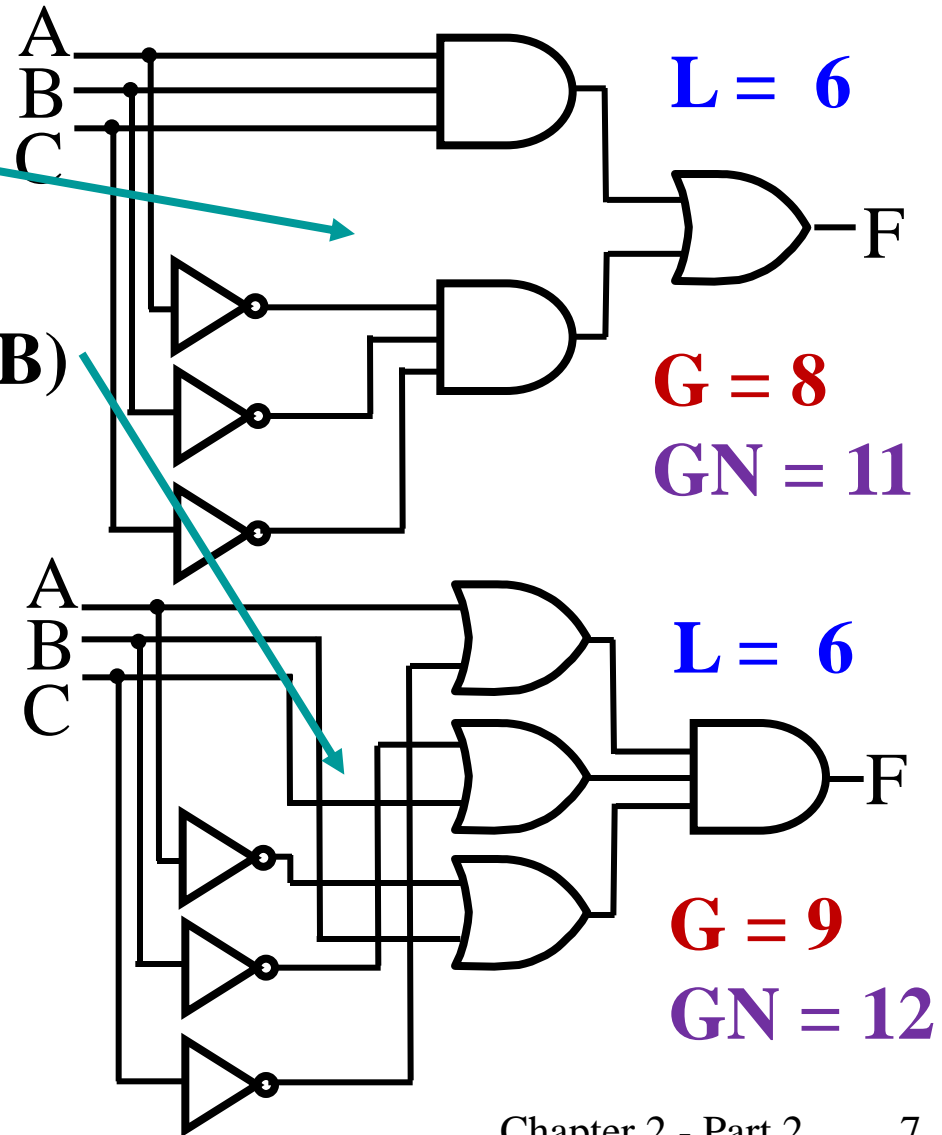
Gate Input Cost (continued)

- **Gate input costs** - the number of inputs to the gates in the implementation corresponding exactly to the given equation or equations. (**G** - inverters not counted, **GN** - inverters counted)

- **For SOP and POS equations, it can be found from the equation(s) by finding the sum of:**
  - **all literal appearances**
  - **the number of terms excluding single literal terms (G) and**
  - **optionally, the number of distinct complemented single literals (GN).**

- **Example:**
  - $F = BD + A\overline{B}C + A\overline{C}\,\overline{D}$ **G = 11, GN = 14**
  - $F = BD + A\overline{B}C + A\overline{B}\,\overline{D} + AB\overline{C}$ **G = 15, GN = 18**
  - $F = (A + \overline{B})(A + D)(B + C + \overline{D})(\overline{B} + \overline{C} + D)$ **G = 14 , GN = 17**
  - **Which solution is best?**

Gate Input Count

- **Example 1**

  $$F = \overline{A}\,\overline{C}\,\overline{D} + \overline{A}\,B\,\overline{C} + ABC + AC\overline{D} \qquad G = 16$$

- **Example 2**

  $$F = \overline{A}\,(\overline{C}\,\overline{D} + B\overline{C}) + A\,(BC + C\overline{D}) \qquad G = 18$$

- **Example 3**

  $$F = \overline{A}\,\overline{C}(B + \overline{D}) + AC(B + \overline{D}) \qquad G = 12$$

- **Example 4**

  $$F = (\overline{A}\,\overline{C} + AC)(B + \overline{D}) \qquad G = \,?$$

Boolean Function Optimization

- **Minimizing the gate input (or literal) cost** of a (a set of) Boolean equation(s) reduces circuit cost.
- Boolean Algebra and graphical techniques are tools to minimize cost criteria values.
- Some important questions:
  - When do we stop trying to reduce the cost?
  - Do we know when we have a minimum cost?
- Treat optimum or near-optimum cost functions for two-level (SOP and POS) circuits first.
- Introduce a graphical technique using **Karnaugh maps** (**K-maps**, for short)

Big Picture of Boolean Function Optimization

Boolean function

$$\overline{B}\,\overline{D} + CD + AD + A\overline{B} + \overline{B}C$$

algebraic transformation →

Canonical form (**SOM/POM**)

$$\sum m(0,2,3,7,8,9, 10,11,13,15)$$

fill the K-map →

G = 15

Gate input cost

K-map

ab\cd	00	01	11	10
00	1	0	1	1
01	0	0	1	0
11	0	1	1	0
10	1	1	1	1

Boolean function

$$\overline{B}\,\overline{D} + (A + C)\,D$$

G = 8

← **literal cost**

multiple-level optimization

Standard form (**SOP/POS**)

$$\overline{B}\,\overline{D} + CD + AD$$

G = 9

← **literal cost**

minimization theorem
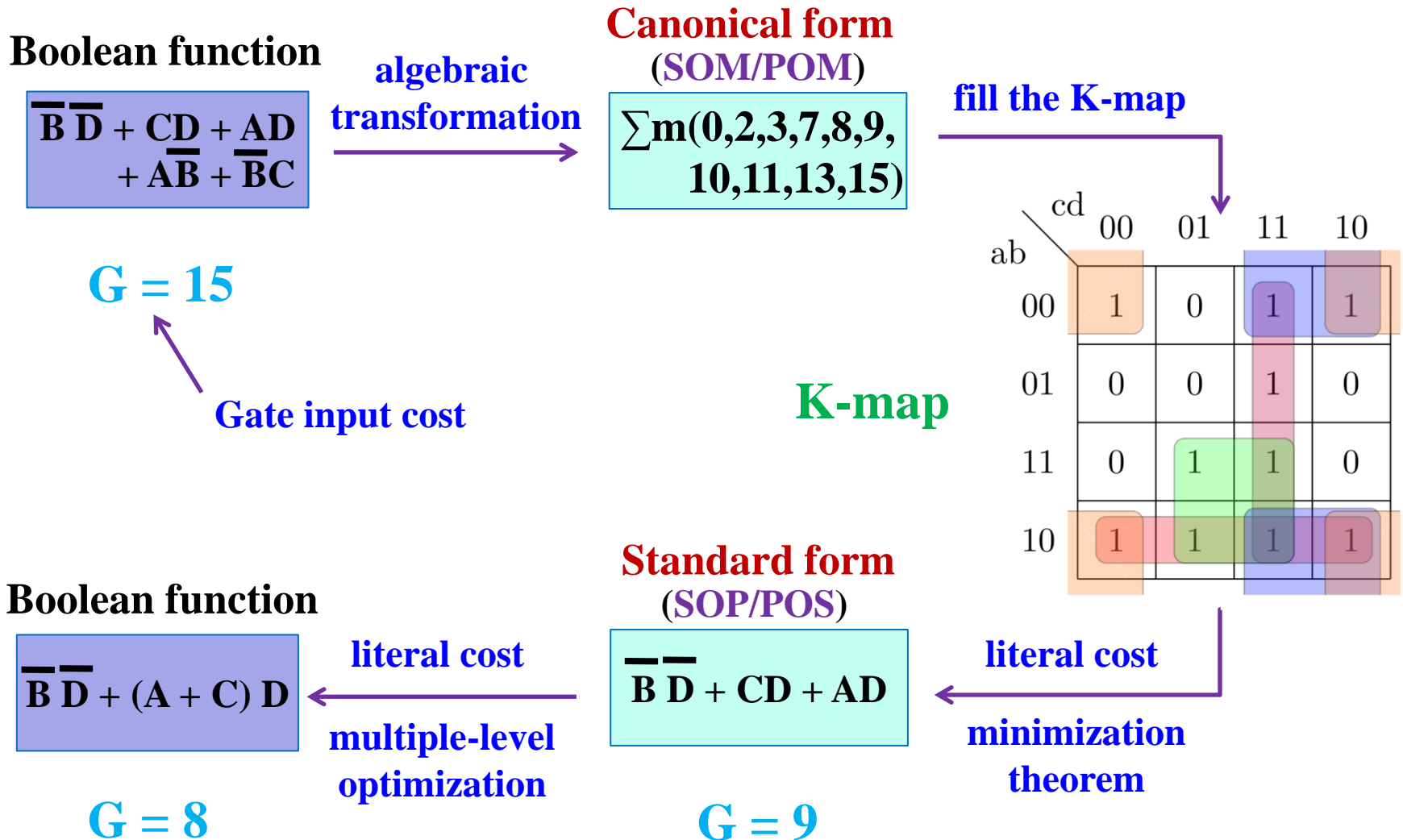
Observations

- **When a boolean function is expressed as a sum of minterms (or a product of maxterms), optimization can be done using minimization theorem:**

$$X \cdot Y + \overline{X} \cdot Y = Y \qquad \text{or} \qquad (X+Y)(\overline{X}+Y) = Y$$

X	Y	F	m_i
0	0	1	$\overline{X}\,\overline{Y}$
0	1	0	$\overline{X}Y$
1	0	1	$X\overline{Y}$
1	1	0	XY

$$F(X,Y) = m_0 + m_2$$
$$= \overline{X}\,\overline{Y} + X\overline{Y} = \overline{Y}$$

or

$$F(X,Y) = M_1 \cdot M_3$$
$$= (X+\overline{Y}) \cdot (\overline{X}+\overline{Y})$$
$$= \overline{Y}$$

K-Map and Truth Tables

- **The K-Map is just a different form of the truth table.**
- **Example – Two variable function:**
  - **We choose m_0, m_1, m_2 and m_3 from the set {0,1} to implement a particular function, $G(x,y)$.**

Truth Table

Input Values (x,y)	Function Value $G(x,y)$
0 0	m_0 (0)
0 1	m_1 (1)
1 0	m_2 (1)
1 1	m_3 (1)

K-Map

$G(x,y)$	$y = 0$	$y = 1$
$x = 0$	m_0	m_1
$x = 1$	m_2	m_3

Karnaugh Maps (K-map)

- **A K-map is a collection of squares**
  - **Each square represents a minterm or maxterm**
  - **The Boolean results are transferred from a truth table onto a two-dimensional grid**
  - **Adjacent squares differ in the value of one variable (each square is ordered in Gray Code)**
  - **Alternative algebraic expressions for the same function are derived by recognizing patterns of squares**
- **The K-map can be viewed as**
  - **A reorganized version of the truth table**
  - **A clever way to rewrite truth tables to make it easier to figure out the logic.**
  - **A topologically-warped Venn diagram as used to visualize sets in algebra of sets**

Some Uses of K-Maps

- **Provide a means for:**
  - **Finding optimum or near optimum**
    - SOP and POS standard forms, and
    - two-level AND/OR and OR/AND circuit implementations

    for functions with 2 to 5 variables
  - **Visualizing concepts related to manipulating Boolean expressions, and**
  - **Demonstrating concepts used by computer-aided design programs to simplify large circuits**

Steps to Minimize Expressions using K-map

- **Step 1**: **Identify minterms or maxterms**

- **Step 2**: **Fill the K-map with minterms or maxterms**
  - For SOM, put 1's in squares respective to the minterms
  - For POM, put 0's in squares respective to the maxterms

- **Step 3**: **Form the rectangular groups** containing maximum number of terms in power of two

- **Step 4**: **Obtain the simplified expression** for groups
  - For SOM, find the product terms and sum them up
  - For POM, find the sum terms and take product of them

Two Variable Maps

- **Example:**

$$F(x,y) = m_2 + m_3$$

$$= x\overline{y} + xy$$

F(x,y)	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:**

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

Two Variable Maps

- **Example:**

$$G(x,y) = m_1 + m_2 + m_3$$
$$= \overline{x}y + x\overline{y} + xy$$

G(x,y)	y = 0	y = 1
x = 0	0	1
x = 1	1	1

repeated term

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

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

Duplicate x y

Three Variable Maps

- **A three-variable K-map:**

	yz=00	yz=01	yz=11	yz=10
x=0	m_0	m_1	m_3	m_2
x=1	m_4	m_5	m_7	m_6

- **Where each minterm corresponds to the product terms:**

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

- **Note that if the binary value for an <u>index</u> differs in one bit position, the minterms are adjacent on the K-Map**

Alternative Map Labeling

- **Each K-map square can be addressed using a unique Gray code-labeling.**
- **Alternate labelings are useful:**

Example Functions

- **By convention, we represent the minterms of F by a "1" in the map and leave the minterms of $\overline{F}$ blank**

- **Example:**
  $$F(x, y, z) = \Sigma_m(2,3,4,5)$$

	y		
0	1	3 **1**	2 **1**
4 **1**	5 **1**	7	6

x (left), z (bottom)

- **Example:**
  $$G(a, b, c) = \Sigma_m(3,4,6,7)$$

- **Learn the locations of the 8 indices based on the variable order shown (x, most significant and z, least significant) on the map boundaries**

	y		
0	1	3 **1**	2
4 **1**	5	7 **1**	6 **1**

x (left), z (bottom)

Combining Squares

- **By combining squares, we reduce number of literals in a product term, reducing the literal cost, thereby reducing the other two cost criteria.**

- **On a 3-variable K-Map:**

  - **One square** represents a minterm with three variables

  - **Two adjacent squares** represent a product term with two variables

  - **Four "adjacent" terms** represent a product term with one variable

  - **Eight "adjacent" terms** is the function of all ones (no variables) = 1.

Example: Combining Squares

- **Example: Let F = Σm(2,3,6,7)**

	$\bar{y}$		y	
$\bar{x}$	0	1	³**1**	²**1**
x	4	5	⁷**1**	⁶**1**

$\bar{z}$ z $\bar{z}$

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

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

- **Thus the four terms that form a 2 $\times$ 2 square correspond to the term "y".**

Three Variable Maps

- **Reduced literal product terms for SOP standard forms correspond to <u>rectangles</u> on K-maps containing cell counts that are powers of 2.**

- **Rectangles of 2 cells represent 2 adjacent minterms; Rectangles of 4 cells represent 4 minterms that form a "pairwise adjacent" ring.**

- **Rectangles can contain non-adjacent cells as illustrated by the "pairwise adjacent" ring above.**

Three Variable Maps

- **Topological warps of 3-variable K-maps that show *all* adjacencies:**
  - **Cylinder**
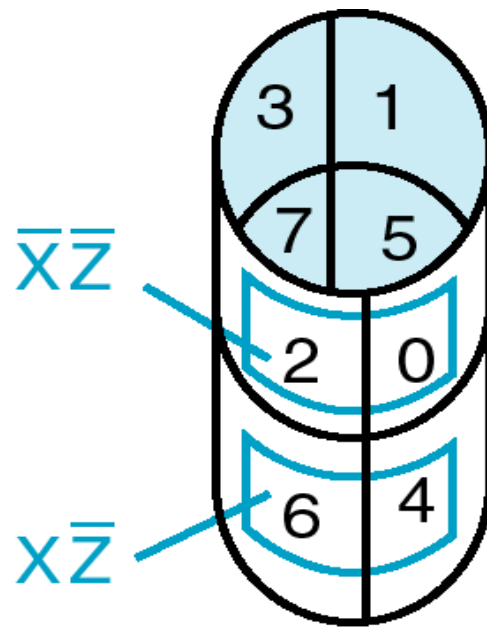
Three Variable Maps

- **Example Shapes of 2-cell Rectangles:**
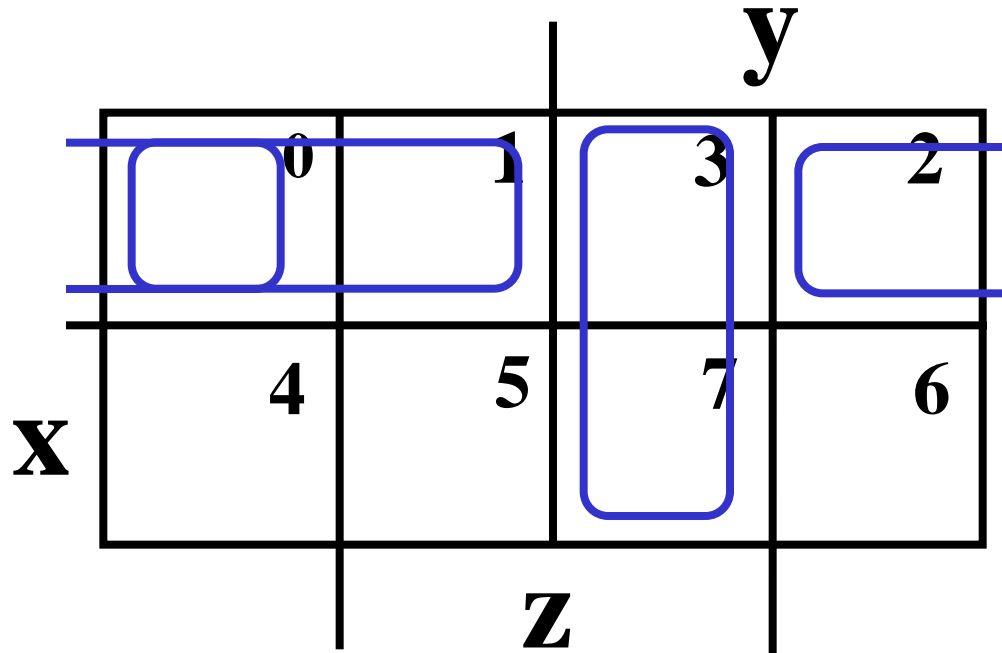
- **Read off the product terms for the rectangles shown**

Three Variable Maps

- **Example Shapes of 4-cell Rectangles:**
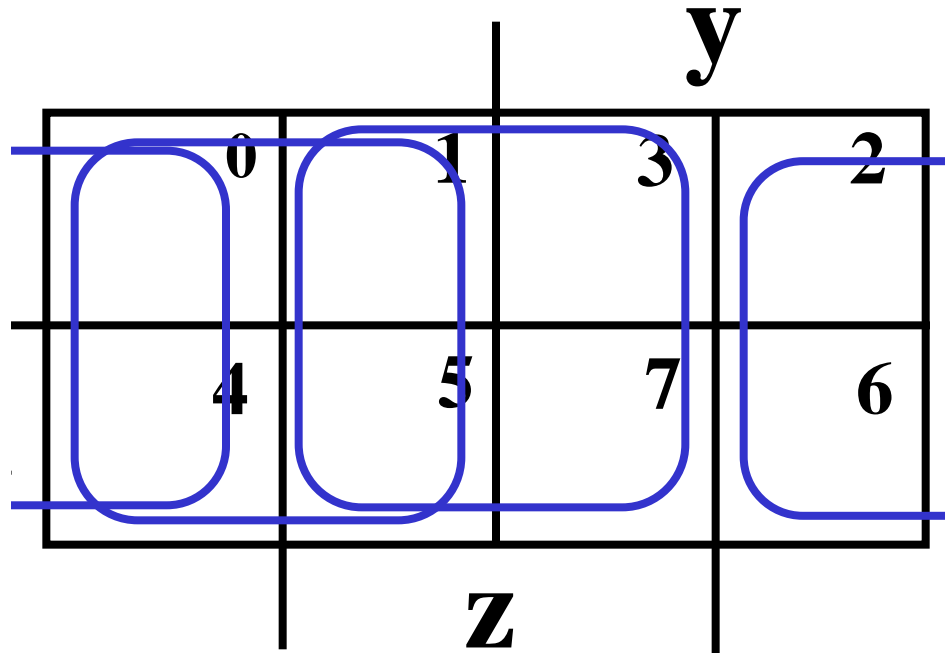
- **Read off the product terms for the rectangles shown**

Three Variable Maps

- **K-Maps can be used to simplify Boolean functions by systematic methods. Terms are selected to cover the "1s" in the map.**

- **Example: Simplify $F(x, y, z) = \Sigma_m(1,2,3,5,7)$**
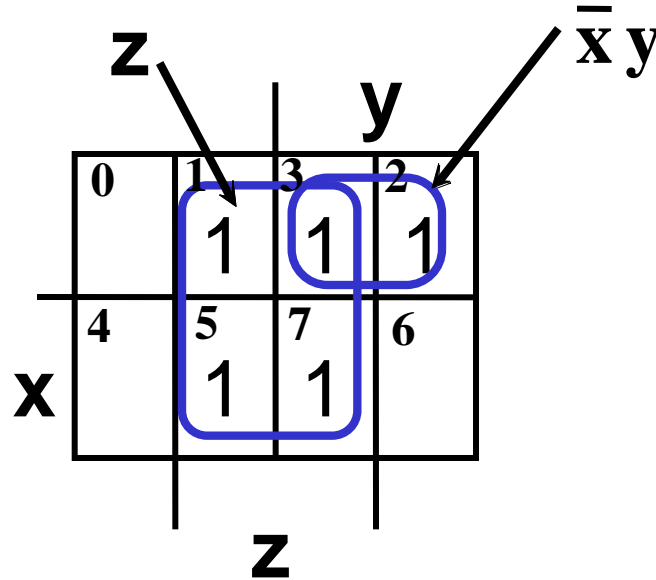
$$F(x, y, z) = z + \bar{x}\,y$$

Three Variable Map Simplification

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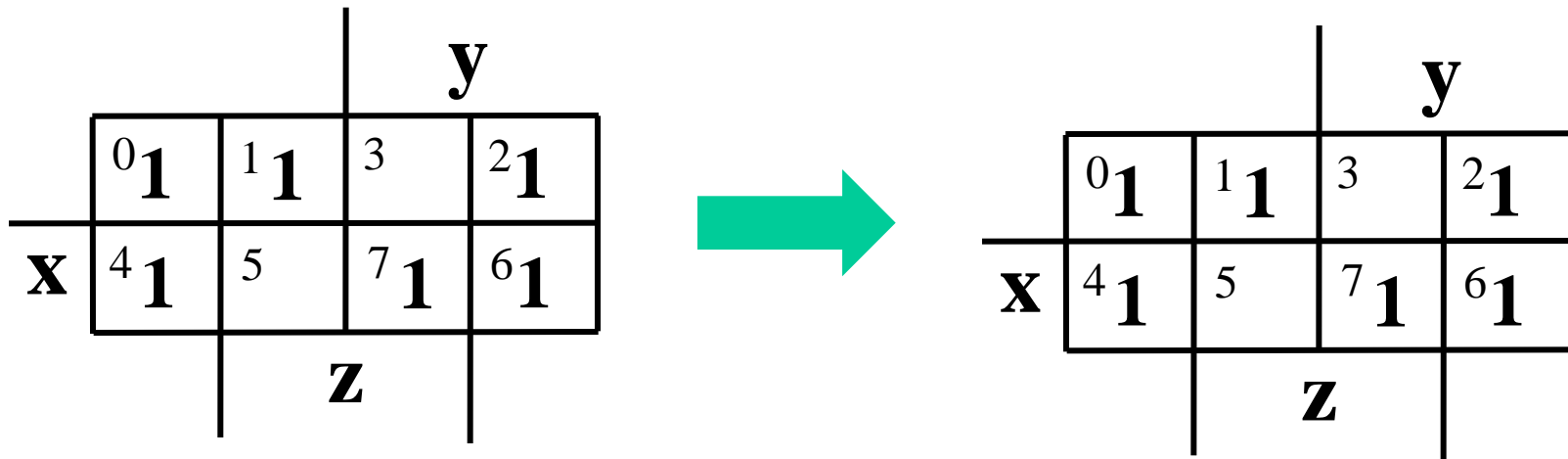
$$F(X, Y, Z) = \overline{Z} + \overline{X}\,\overline{Y} + XY$$

Pitfalls of Combining Squares (1/2)

- **No zeros allowed.**
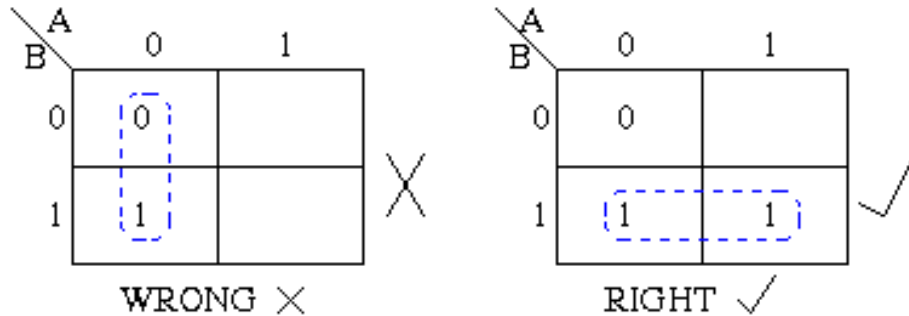
- **Only 2^n number of squares in each group.**
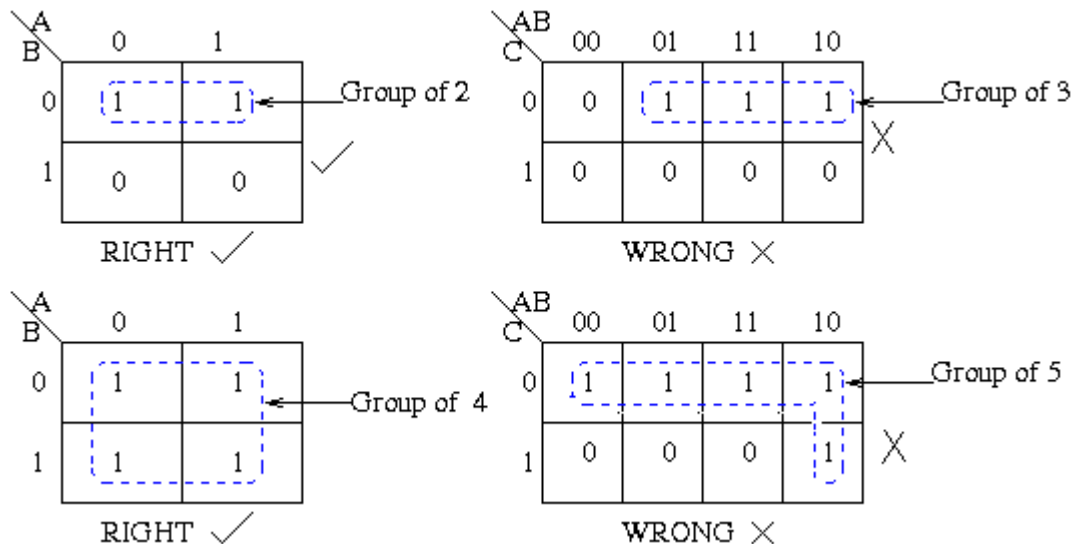
Pitfalls of Combining Squares (2/2)

- **Groups should be as large as possible.**
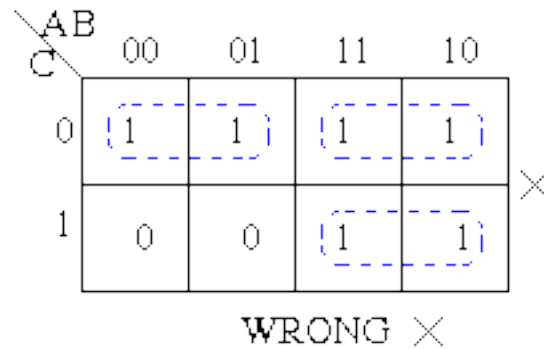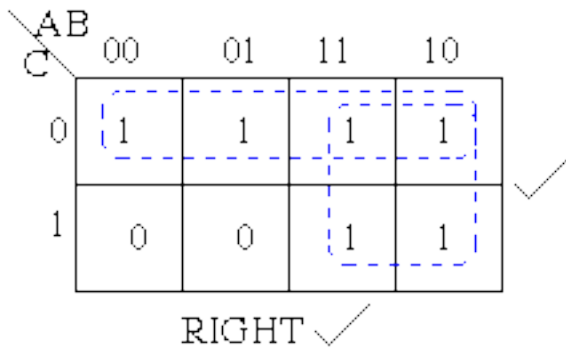
- **Groups may overlap.** - **Wrap around allowed.**

refer to: http://www.ee.surrey.ac.uk/Projects/Labview/minimisation/karrules.html
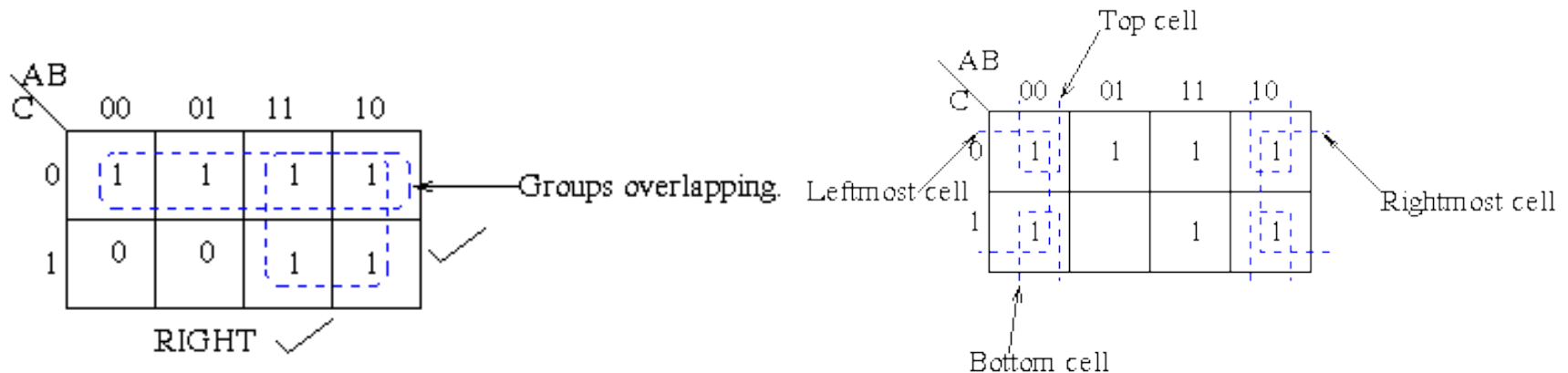
Questions about Combining Squares

- **Why are only 2^n number of squares considered in each group?**
  - reducing 1 literal, needs 2 cells with X and $\overline{X}$,
  - reducing 2 literals, needs 4 cells with $\overline{X}\,\overline{Y}$, $\overline{X}Y$, $X\overline{Y}$ and XY,
  - therefore, reducing n literals, needs 2^n cells.
- **Why should each group be as large as possible?**
  - reducing the number of variables to a minimum

not minimal
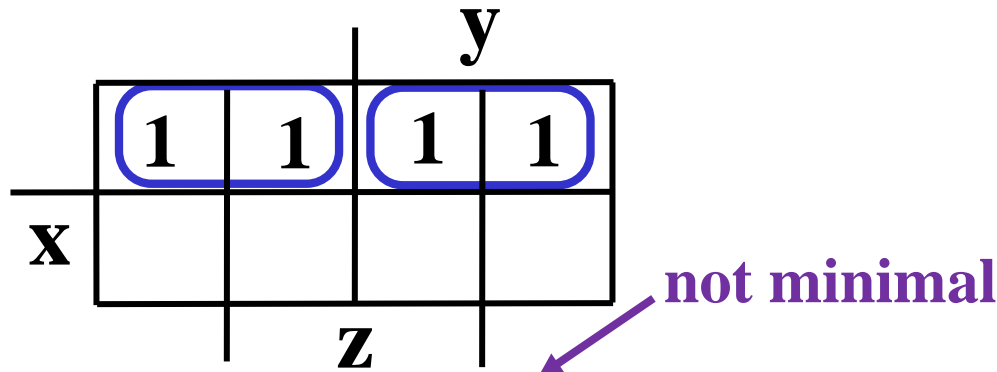
$$F(X, Y, Z) = \overline{X}\,\overline{Y} + \overline{X}Y = \overline{X}$$
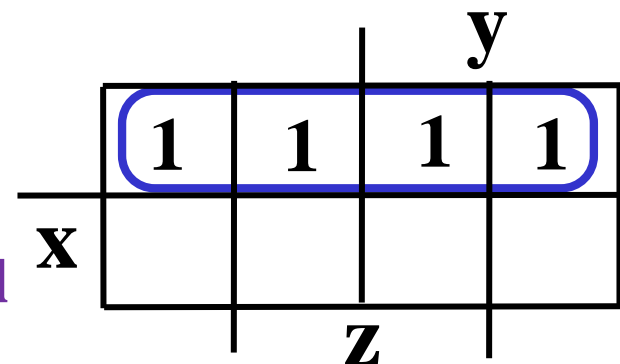
$$F(X, Y, Z) = \overline{X}$$

Four Variable Maps

- **Map and location of minterms:**

WXYZ

	00	01	11 Y 10	
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

Variable Order

X

W

Z

Four Variable Terms

- **Four variable maps can have rectangles corresponding to:**
    - **A single 1 = 4 variables, (Minterm)**
    - **Two 1s = 3 variables,**
    - **Four 1s = 2 variables,**
    - **Eight 1s = 1 variable,**
    - **Sixteen 1s = zero variables (Constant "1")**

Four Variable Maps
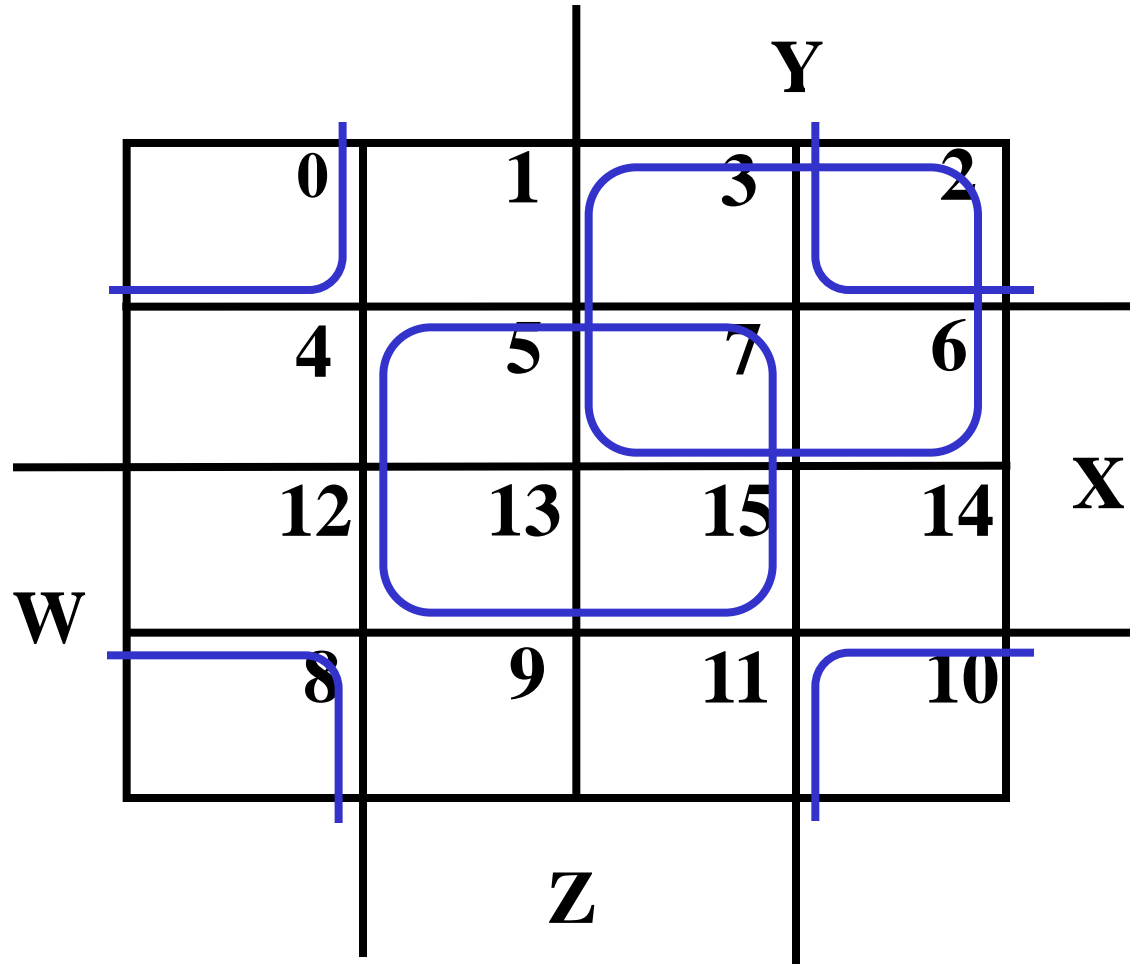
- **Example Shapes of Rectangles:**

Four Variable Maps

- **Example Shapes of Rectangles:**
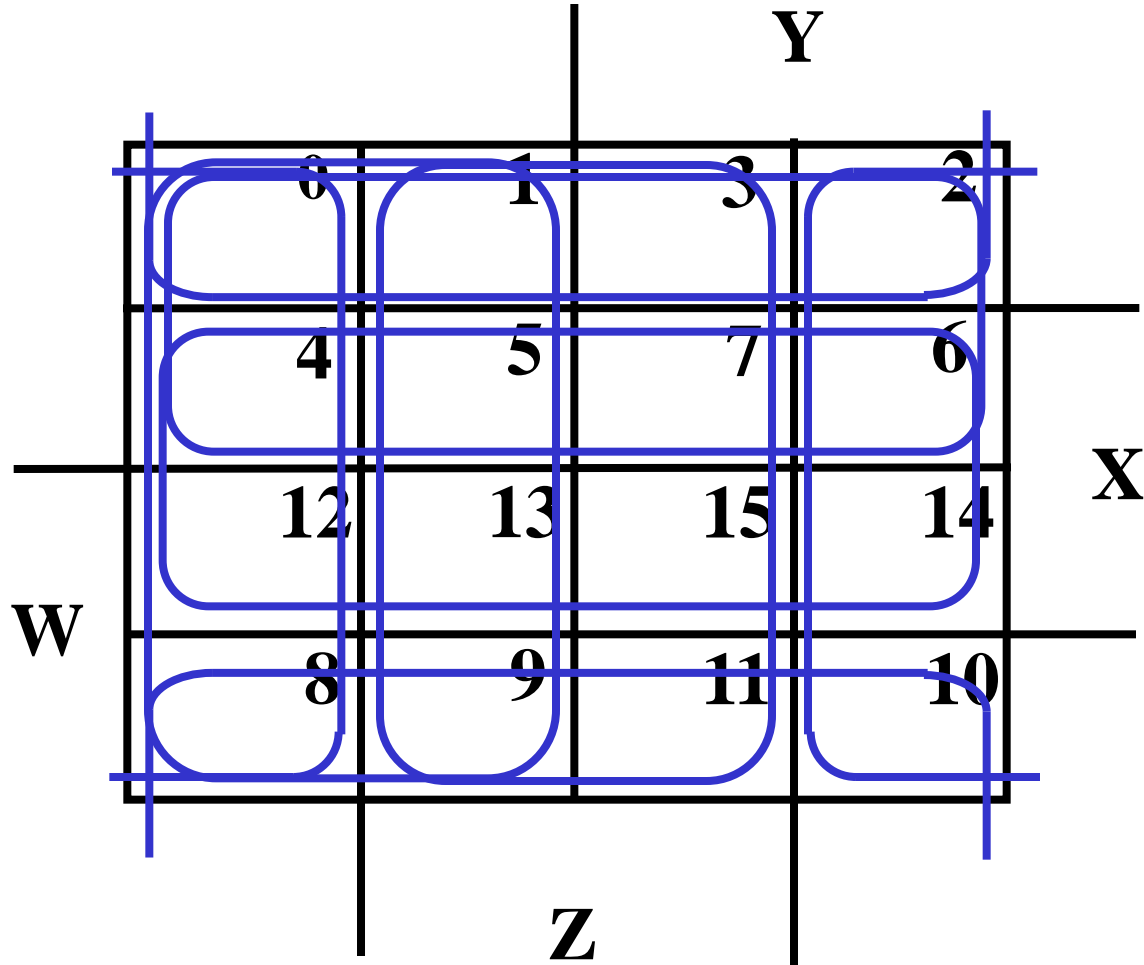
Four Variable Map Simplification

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

Four Variable Map Simplification

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

Five Variable Maps

- **2^5 Minterms, consists of 32 squares**

Gray Code order

XYZ VW	000	001	011	010	110	111	101	100
00	0	1	3	2	6	7	5	4
01	8	9	11	10	14	15	13	12
11	24	25	27	26	30	31	29	28
10	16	17	19	18	22	23	21	20

5 variable K' Map: https://www.bilibili.com/video/BV1Nq4y1k71R?
spm_id_from= 333.337.search-card.all.click

Don't Cares in K-Maps

- **Sometimes a function table or map contains entries for which it is known:**
  - The input values for the **minterm will never occur**, or
  - The output value for the **minterm is not used**
- **In these cases, the output value need not be defined.**
- **The output value is defined as a "don't care" that may take on either a 0 or 1 value in resulting solutions.**
- **By placing "don't cares" (an "x" entry) in the function table or map, the cost of the logic circuit may be lowered.**
- **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 will never occur, so the output values for these codes are "don't cares."**

Some input values will never occur!

Example: BCD "5 or More"

- **Function F=(w,x,y,z) is defined as "5 or more" over BCD inputs (e.g., F(0011)=0, F(1001)=1). With the don't cares used for the 6 non-BCD combinations:**
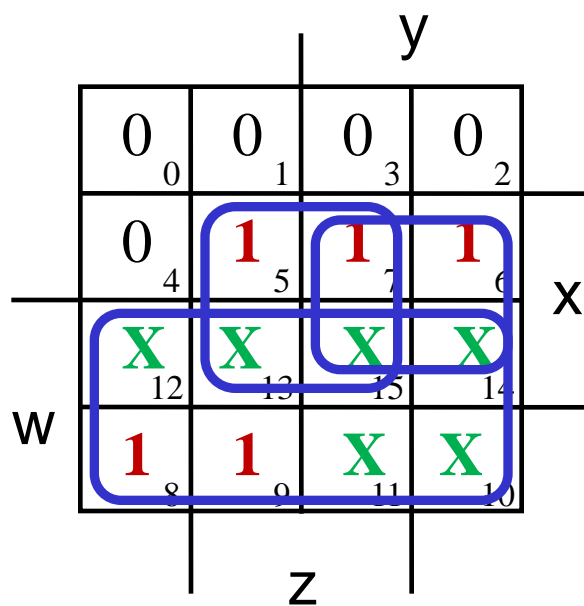
$$F_1(w, x, y, z) = \overline{w}\,x\,z + \overline{w}\,x\,y + w\,\overline{x}\,\overline{y}$$

(without don't cares)   **G = 12**

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

(with don't cares)   **G = 7**

- **F_2 is much lower in cost than F_1 where the "don't cares" were treated as "0s".**

- **For this particular function, cost G for the POS solution for F_2(w,x,y,z) is not changed by using the don't cares.**

Product of Sums Example

- **Find the optimum <u>POS</u> solution:**

$$F(A, B, C, D) = \Sigma_m(3,9,11,12,13,14,15) + \Sigma d\,(1,4,6) \quad \Longleftarrow \quad \text{don't cares}$$

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

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

$$\overline{F}(A,B,C,D) = \overline{A}B + \overline{B}\,\overline{D}$$
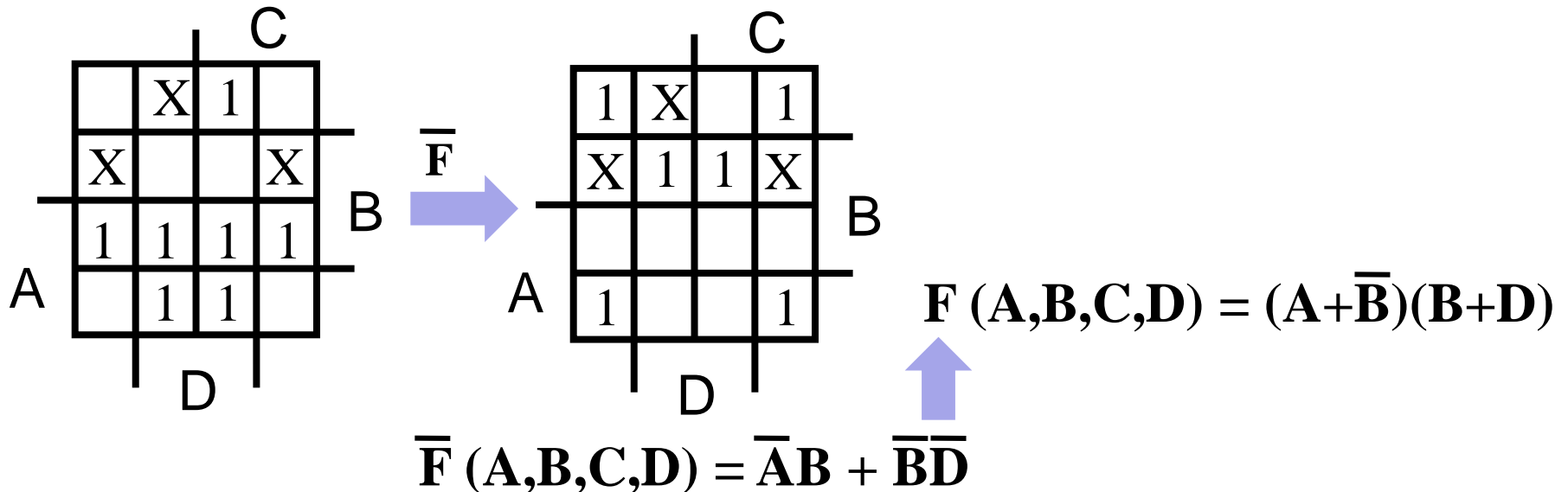
Encoding Schema and Optimization

- **Using binary and one-hot encodings to represent three elements (e.g., three primary colors):**

Binary encoding

2-bit binary encoding

X	Y	F1	F2
0	0	1	0
0	1	0	1
1	0	0	1
1	1	X	X

$F1(X,Y) = \overline{X}\,\overline{Y}$

$F2(X,Y) = X + Y$

3-bit one-hot

One-hot encoding

X	Y	Z	F1	F2
0	0	1	1	0
0	1	0	0	1
1	0	0	0	1
X	X	X	X	X

$F1(X,Y,Z) = Z$

$F2(X,Y,Z) = \overline{Z}$

F1

	y		
^{0}X	^{1}**1**	^{3}X	^{2}
^{4}	^{5}X	^{7}X	^{6}X

z

F2

	y		
^{0}X	^{1}	^{3}X	^{2}**1**
^{4}**1**	^{5}X	^{7}X	^{6}X

z
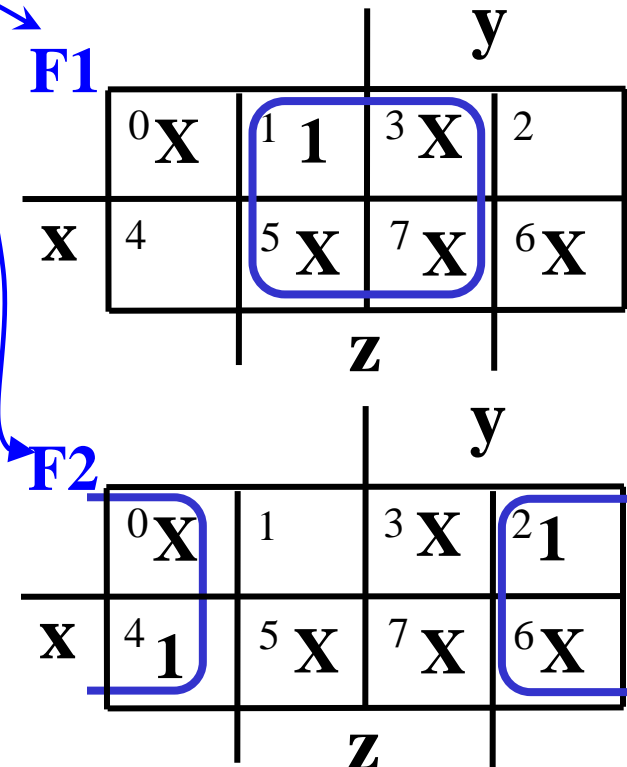
- **We can do one-hot design by inspection.**

Advantages of K-Maps

- **The K-map simplification technique is simpler and less error-prone.**

- **It prevents the need to remember each and every Boolean algebraic theorem.**

- **It involves fewer steps than the algebraic minimization technique to arrive at a simplified expression.**

- **K-map simplification technique always results in minimum expression if carried out properly.**

Disadvantages of K-Maps
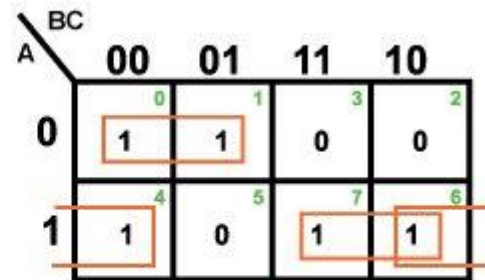
- **As the number of variables in the logical expression increases, the K-map simplification process becomes complicated.**
- **The minimum logical expression may not be unique depending on the choices made while forming the groups.**
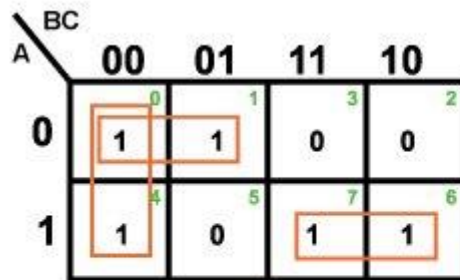
$$Y = \overline{A}\overline{B} + AB + \overline{B}\overline{C}$$

$$Y = \overline{A}\overline{B} + AB + A\overline{C}$$

**K-map with a non-unique solution**

Optimization Algorithm

- **Minimizing boolean functions by hand using the classical K-map is a laborious, tedious and error prone process.**

- **It isn't suited for more than six input variables and practical only for up to 5 variables, while product term sharing for multiple output functions is even harder to carry out.**

- **Moreover, this method doesn't lead itself to be automated in the form of a computer program.**

- **Demonstration: Truth Table Solver**

  from: https://truthtablesolve.sourceforge.net

Example of Boolean Minimization

- **Truth Table Solver is a program that solves the truth table and output all the possible minimized boolean expressions.**

- **It uses Quine-McCluskey algorithm (the method of prime implicants) for boolean minimization.**

- **Quine-McCluskey Solver: http://www.quinemccluskey.com**

Systematic Simplification

- **Implicant** is a product in SOP or a sum in POS. For example:
  F = AB + ABC + BC. Implicants are AB, ABC and BC.
- **In K-map, the implicant is a group containing 2^n squares.**
- **There are two types of implicants: Prime Implicant and Essential Prime Implicant.**
- **A *Prime Implicant* is a product term obtained by combining the maximum possible number of adjacent squares in the map with 2^n number of squares.**
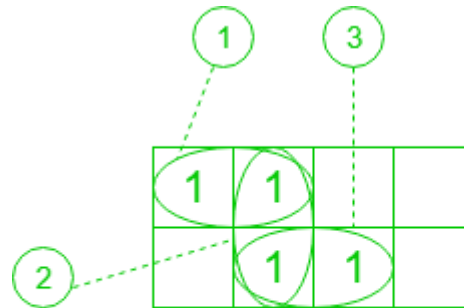
No. of Prime Implicants = 3

Systematic Simplification (1/2)

- **An *Essential Prime Implicant* is a prime implicant that covers one or more minterms that no combination of other prime implicants are able to include.**
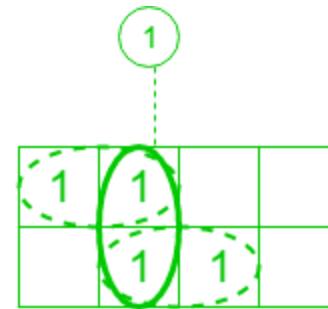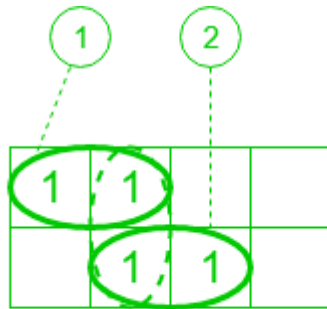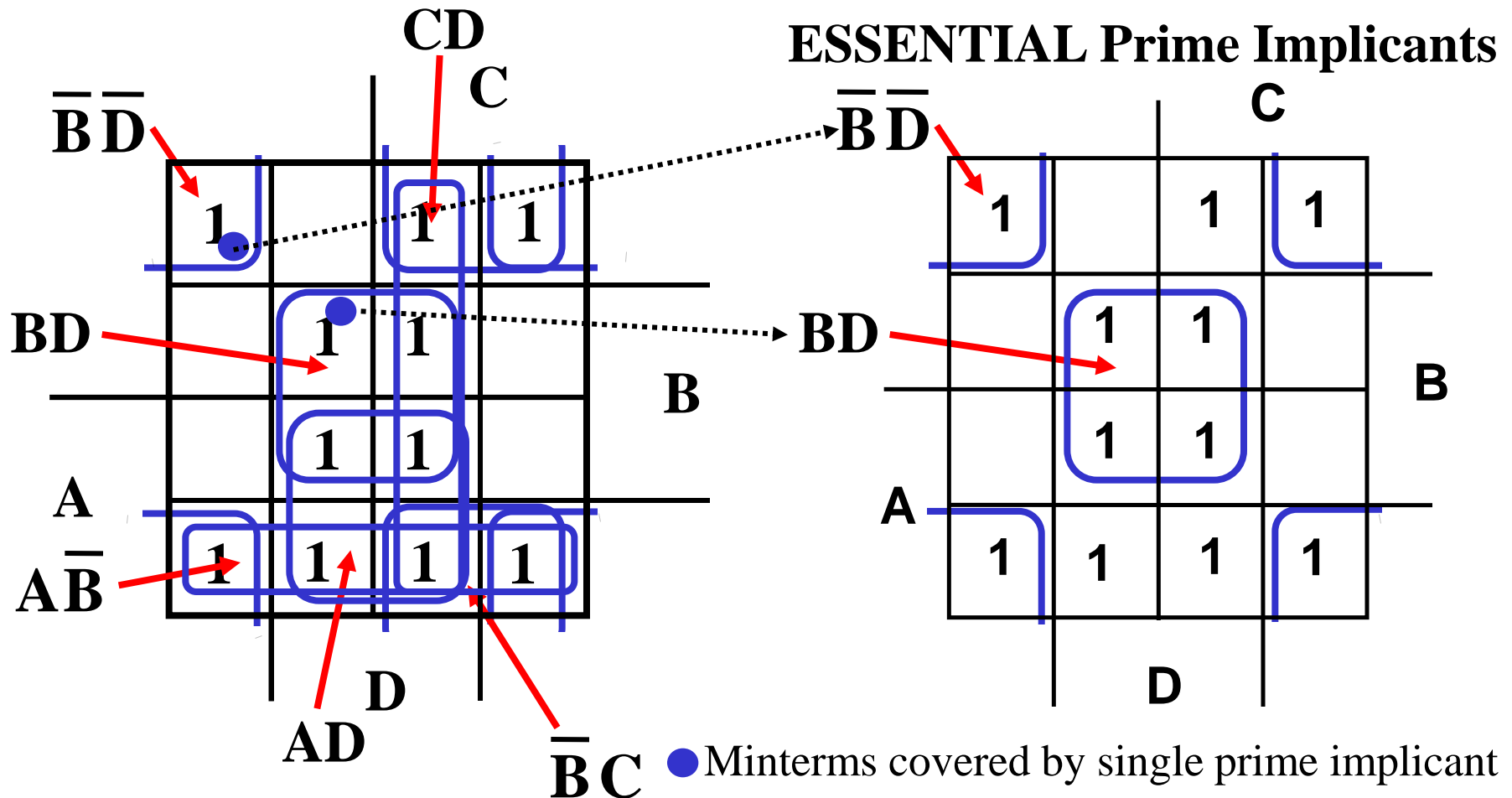

No. of Essential Prime Implicants = 2   No. of Redundant Prime Implicants = 1

- **A set of prime implicants *"covers all minterms"* if, for each minterm of the function, at least one prime implicant in the set of prime implicants includes the minterm.**

Systematic Simplification (2/2)

■ **However a cyclic Boolean function doesn't have Essential Prime Implicant. Every minterm is covered by at least two prime implicants and every prime implicant is of same size.**

3 variables cyclic function

4 variables cyclic function

Example of Prime Implicants

- **Find ALL Prime Implicants**

ESSENTIAL Prime Implicants

Minterms covered by single prime implicant

Prime Implicant Practice

- **Find all prime implicants for:**
  $$F(A, B, C, D) = \Sigma_m(0,2,3,8,9,10,11,12,13,14,15)$$
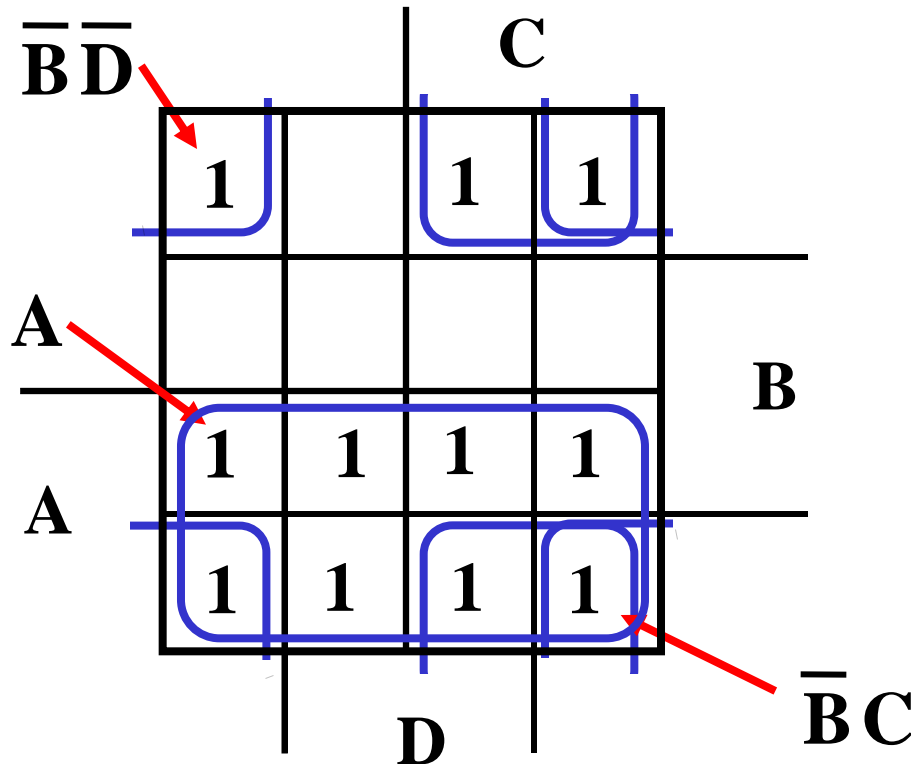
Another Example

- **Find all 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!**

Quine–McCluskey Algorithm

- **Find all prime implicants.**

- **Include all essential prime implicants in the solution**

- **Select a minimum cost set of non-essential prime implicants to cover all minterms not yet covered:**

  - **Obtaining an optimum solution: See Reading Supplement - More on Optimization**

  - **Obtaining a good simplified solution: Use the Selection Rule**

- **Appendix C: Quine–McCluskey Solver**

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.
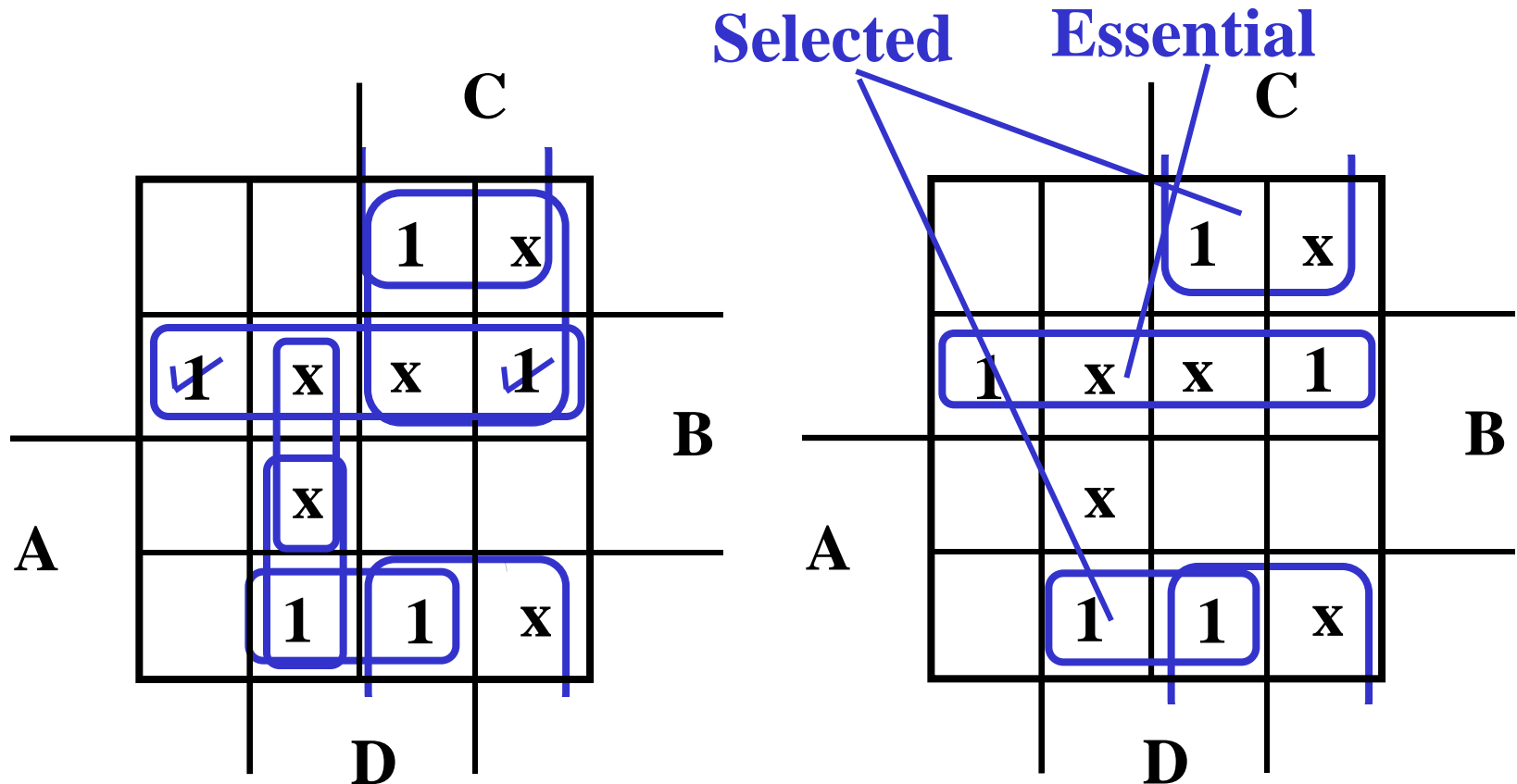
Selection Rule Example

- **Simplify F(A, B, C, D) given on the K-map.**
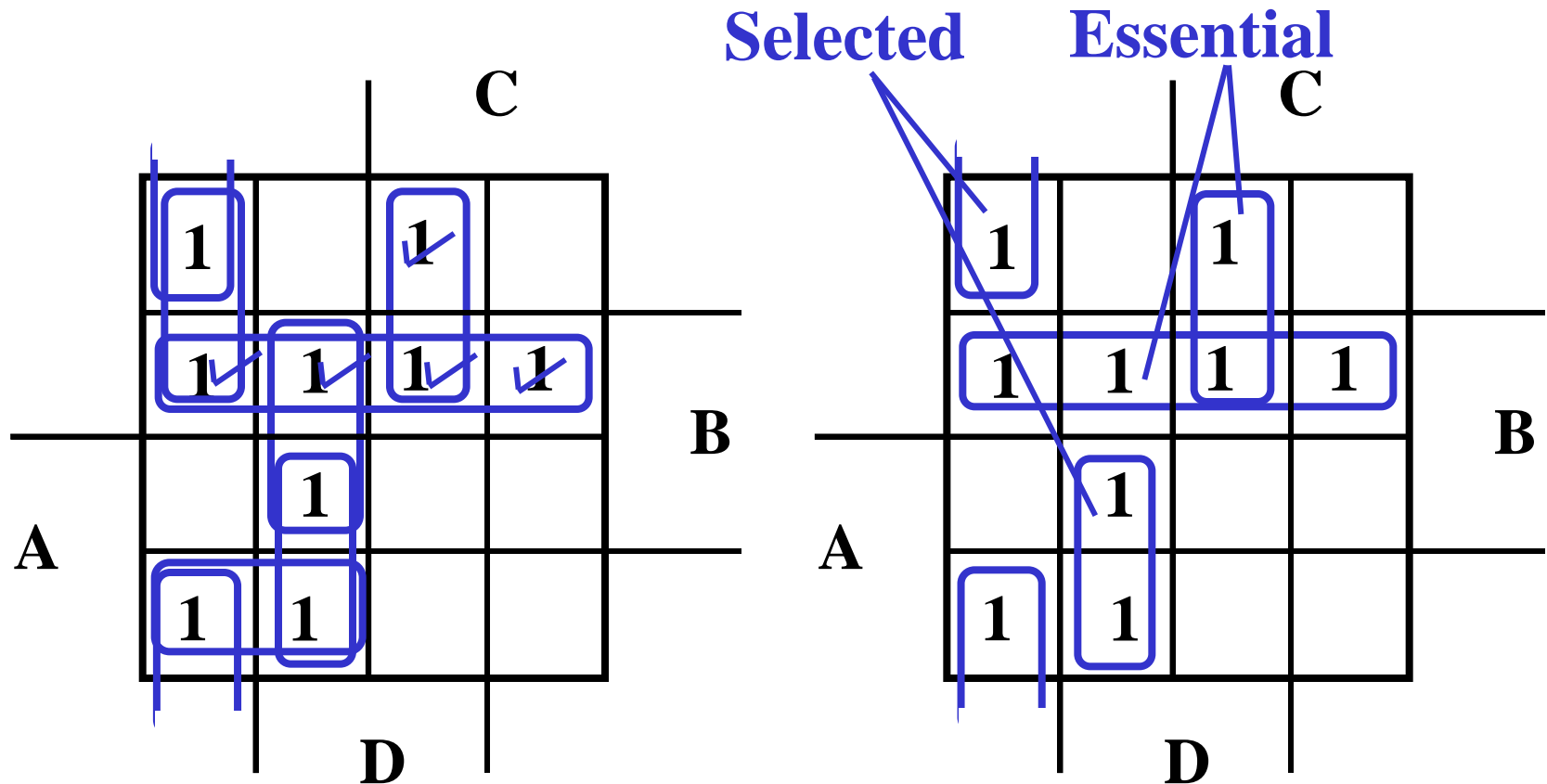
✔ Minterms covered by essential prime implicants

Selection Rule Example with Don't Cares

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

Selected   **Essential**

✔ Minterms covered by essential prime implicants

Multiple-Level Optimization

- **Multiple-level circuits - circuits that are not two-level (with or without input and/or output inverters)**

- **Multiple-level circuits can have reduced gate input cost compared to two-level (SOP and POS) circuits**

- **Multiple-level optimization is performed by applying transformations to circuits represented by equations while evaluating cost**

Transformation Examples

- **F(A, B, C, D) = $\sum_m$ (0, 4, 5, 10, 14, 15)**

- **Algebraic Factoring**
  $$F = \overline{A}\,\overline{C}\,\overline{D} + \overline{A}\,B\,\overline{C} + ABC + AC\overline{D} \qquad G = 16$$

  - **Factoring:**
  $$F = \overline{A}\,(\overline{C}\,\overline{D} + B\overline{C}) + A\,(BC + C\overline{D}) \qquad G = 18$$

  - **Factoring again:**
  $$F = \overline{A}\,\overline{C}(B + \overline{D}) + AC\,(B + \overline{D}) \qquad G = 12$$

  - **Factoring again:**
  $$F = (\overline{A}\,\overline{C} + AC)(B + \overline{D}) \qquad G = 10$$

Transformation Examples

- **Algebraic Factoring**

$$F = \overline{A}\,\overline{C}\,\overline{D} + \overline{A}\,B\,\overline{C} + ABC + AC\overline{D} \qquad G = 16$$

$$F = \overline{A}\,(\overline{C}\,\overline{D} + B\overline{C}) + A\,(BC + C\overline{D}) \qquad G = 18$$
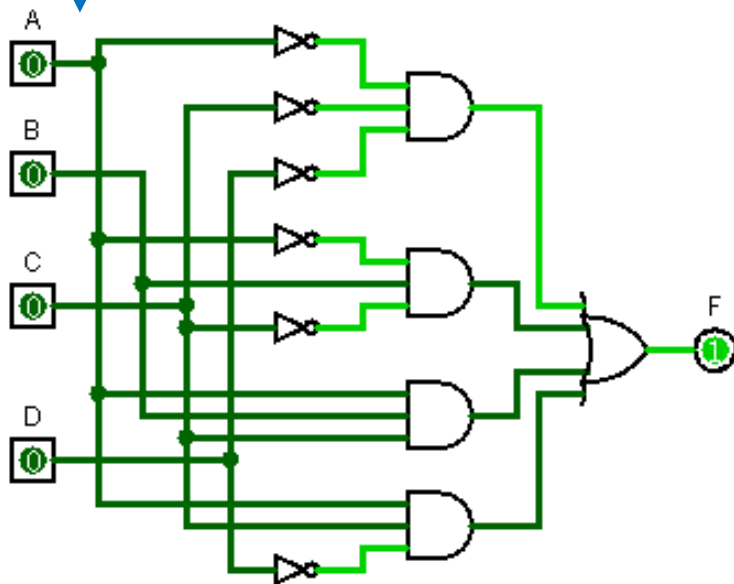
two-level circuit
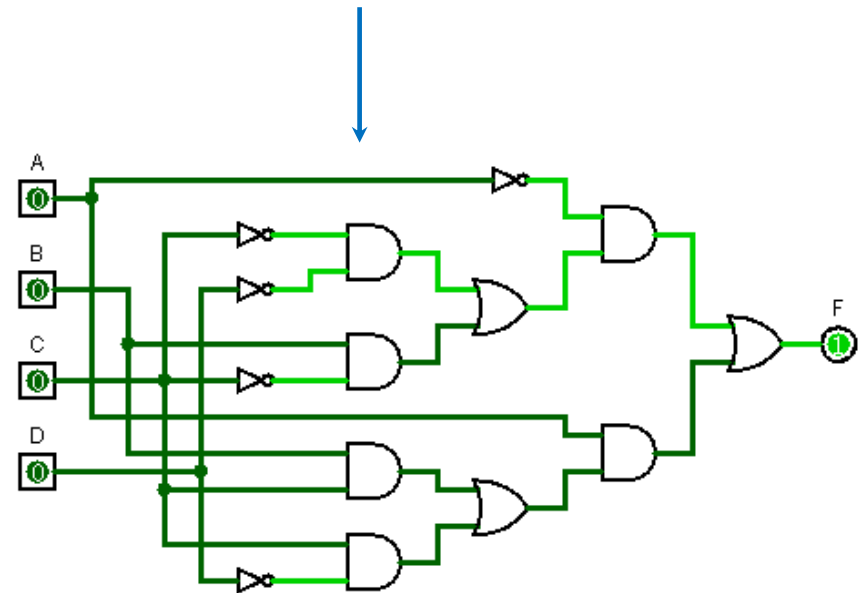
four-level circuit

Transformation Examples

- **Algebraic Factoring**

$$F = \overline{A}\ \overline{C}(\ B + \overline{D}\ ) + AC\ (B + \overline{D})$$ $G = 12$

$$F = (\overline{A}\overline{C} +\ AC)\ (B + \overline{D})$$ $G = 10$
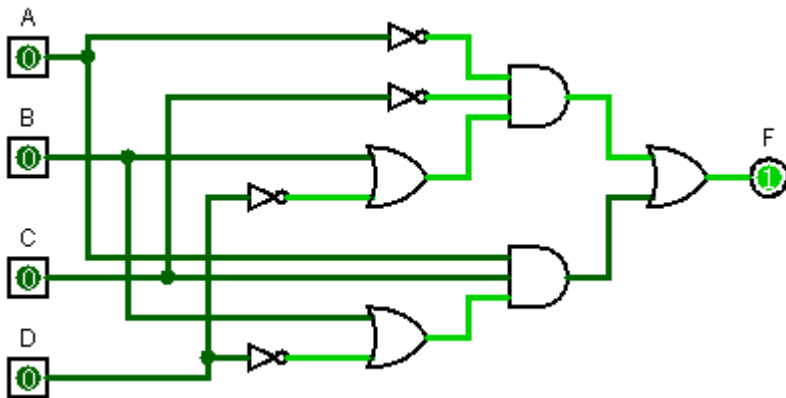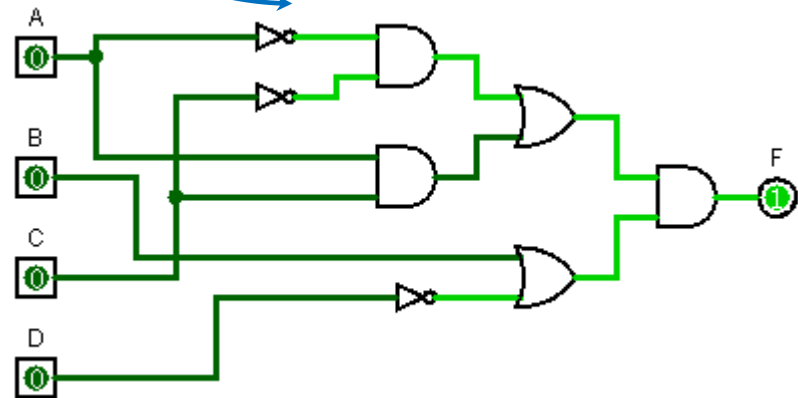
three-level circuit   **three-level circuit**

Overview

- **Part 1 – Gate Circuits and Boolean Equations**
  - **Binary Logic and Gates**
  - **Boolean Algebra**
  - **Standard Forms**
- **Part 2 – Circuit Optimization**
  - **Two-Level Optimization**
  - **Map Manipulation**
  - **Practical Optimization**
  - **Multi-Level Circuit Optimization**
- **Part 3 – Additional Gates and Circuits**
  - **Other Gate Types**
  - **Exclusive-OR Operator and Gates**
  - **High-Impedance Outputs**

Other Gate Types

- **Why?**
  - **Implementation feasibility and low cost**
  - **Power in implementing Boolean functions**
  - **Convenient conceptual representation**

- **Gate classifications**
  - **Primitive gate - a gate that can be described using a single primitive operation type (AND or OR) plus an optional inversion(s).**
  - **Complex gate - a gate that requires more than one primitive operation type for its description**

- **Primitive gates will be covered first**

NAND Gate

- **The basic NAND gate has the following symbol, illustrated for three inputs:**

  - **AND-Invert (NAND)**

$$\text{X} \quad \text{Y} \quad \text{Z} \qquad F(X, Y, Z) = \overline{X \ Y \ Z}$$
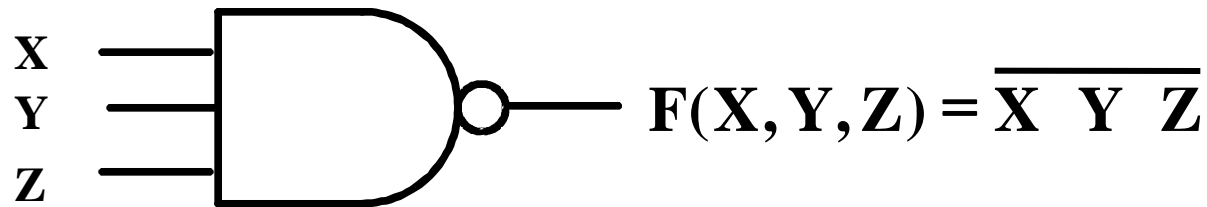
- **NAND represents <u>NOT</u> <u>AND</u>, i. e., the AND function with a NOT applied. The symbol shown is an AND-Invert. The small circle ("bubble") represents the invert function.**

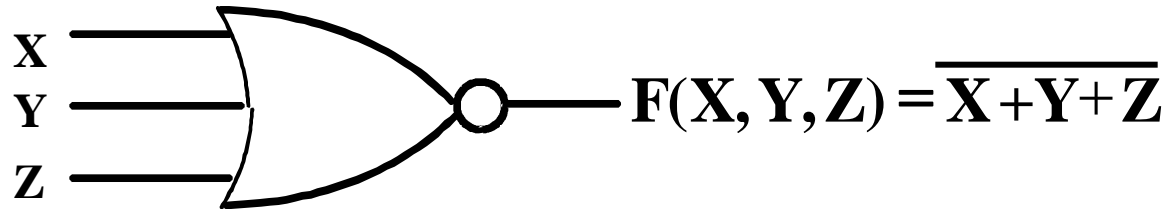# NAND Gates (continued)

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

$$X, Y, Z \qquad F(X, Y, Z) = \overline{X} + \overline{Y} + \overline{Z}$$
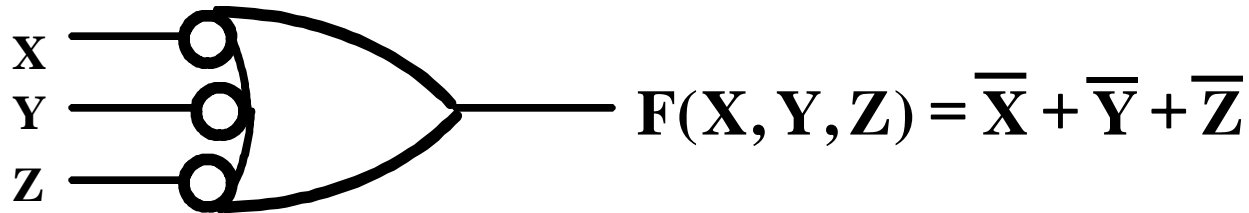
- **This NAND symbol is called Invert-OR, since inputs are inverted and then ORed together.**

- **AND-Invert and Invert-OR both represent the NAND gate. Having both makes visualization of circuit function easier.**

- **A NAND gate with one input degenerates to an inverter.**

NAND Gates (continued)

- **The NAND gate is the natural implementation for CMOS technology in terms of chip area and speed.**
- *Universal gate* **- a gate type that can implement any Boolean function.**
- **The NAND gate is a universal gate as shown in Figure 2-4 of the text.**
- **NAND usually does not have an operation symbol defined since**
  - **the NAND operation is not associative, and**
 - **we have difficulty dealing with non-associative mathematics!**

NOR Gate

- **The basic NOR gate has the following symbol, illustrated for three inputs:**

  - **OR-Invert (NOR)**

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

with inputs X, Y, Z.

- **NOR represents <u>NOT - OR</u>, i. e., the OR function with a NOT applied. The symbol shown is an OR-Invert. The small circle ("bubble") represents the invert function.**

NOR Gate (continued)

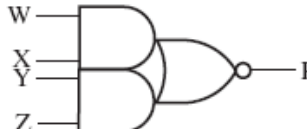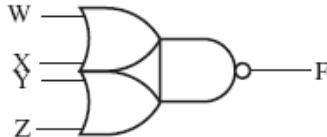- **Applying DeMorgan's Law gives Invert-AND (NOR)**
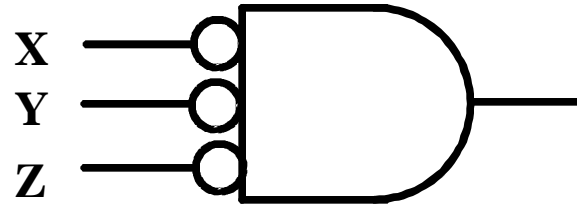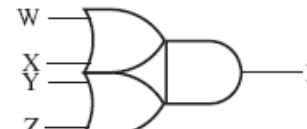

- **This NOR symbol is called Invert-AND, since inputs are inverted and then ANDed together.**

- **OR-Invert and Invert-AND both represent the NOR gate. Having both makes visualization of circuit function easier.**

- **A NOR gate with one input degenerates to an inverter.**

NOR Gate (continued)

- **The NOR gate is a natural implementation for some technologies other than CMOS in terms of chip area and speed.**

- **The NOR gate is a universal gate**

- **NOR usually does not have a defined operation symbol since**
  - the NOR operation is not associative, and
  - we have difficulty dealing with non-associative mathematics!

Other Gate Types

- **Other gate types aim to lower the cost and transmit time in circuit.**

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

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**
- **XOR and XNOR gates are both complex gates.**

Truth Tables for XOR/XNOR

- **Operator Rules: XOR   XNOR**

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

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 ≡?**

Exclusive OR/ Exclusive NOR

- **Definitions**
  - **The XOR function is:** $X \oplus Y = X \overline{Y} + \overline{X} Y$
  - **The eXclusive NOR (XNOR) function, otherwise known as *equivalence* is:** $\overline{X \oplus Y} = X Y + \overline{X} \, \overline{Y}$
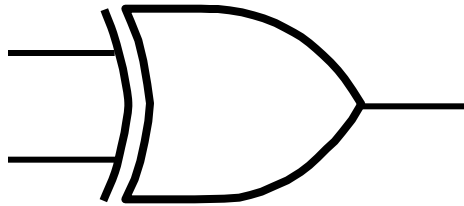- **Uses for the XOR and XNORs gate include:**
  - **Adders/subtractors/multipliers**
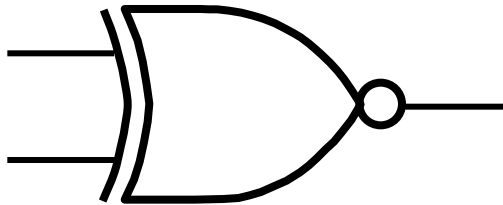 - **Counters/incrementers/decrementers**
  - **Parity generators/checkers**
- **Strictly speaking, XOR and XNOR gates do not exist for more than two inputs. Instead, they are replaced by odd and even functions.**
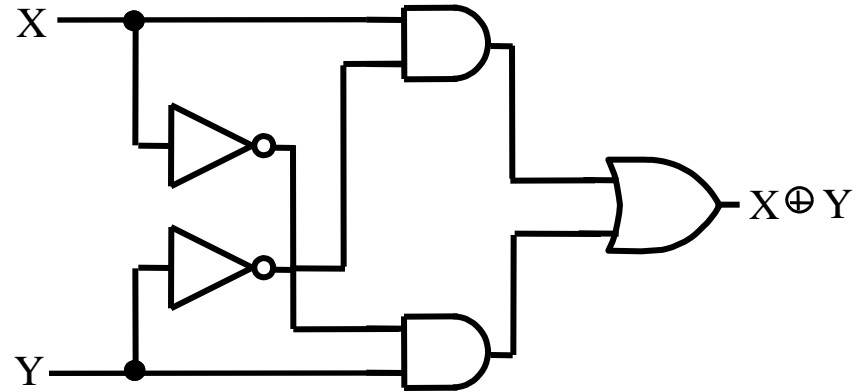
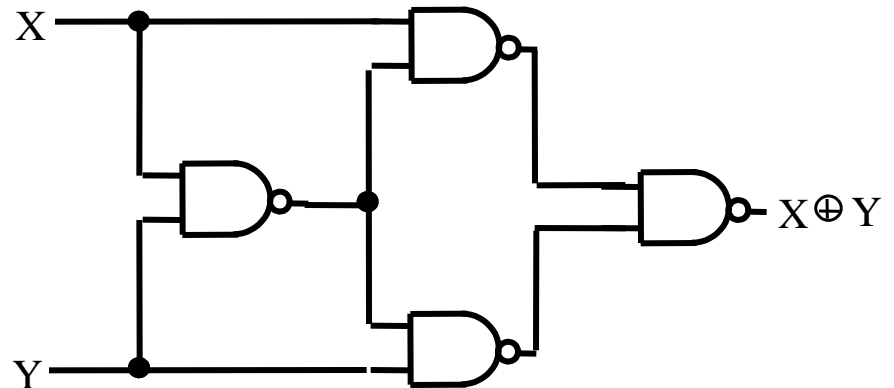# Symbols For XOR and XNOR

- **XOR symbol:**

- **XNOR symbol:**

- **Shaped symbols exist only for two inputs**

XOR Implementations

- **The simple SOP implementation uses the following structure:**

X $\oplus$ Y

- **A NAND only implementation is:**

X $\oplus$ Y

XOR/XNOR (Continued)

- **The XOR identities:**

$$X \oplus 0 = X \qquad\qquad X \oplus 1 = \overline{X}$$

$$X \oplus X = 0 \qquad\qquad X \oplus \overline{X} = 1$$

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

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

- **XOR and XNOR are associative operations.**

Find the Element that Appears Once

- **An array is consisting of integer numbers. Each number except one appears twice. The remaining number appears only once. How to find a number which occurs only once?**

- **Example: Suppose that an array contains values 1, 4, 2, 1, 3, 4, 2. Find 3 that appears only once.**

Function Singular(int a[], int Count)
    value = 0
    for i = 0, Count-1
        value = value $\oplus$ a[i]
    return value

- **using 4 XOR identities**
  - ✓ **commutative law**
  - ✓ **associative law**
  - ✓ $X \oplus X = 0$
  - ✓ $X \oplus 0 = X$

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:**

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

- **The complement of the odd function is the even function.**

Odd and Even Functions

- **The odd and even functions on a K-map form "checkerboard" patterns.**

YZ

X

	00	01	11	10
0		1		1
1	1		1	

$$X \oplus Y \oplus Z$$

YZ

WX

	00	01	11	10
00		1		1
01	1		1	
11		1		1
10	1		1	

$$W \oplus X \oplus Y \oplus Z$$

Odd and Even Functions (continued)

- **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:**
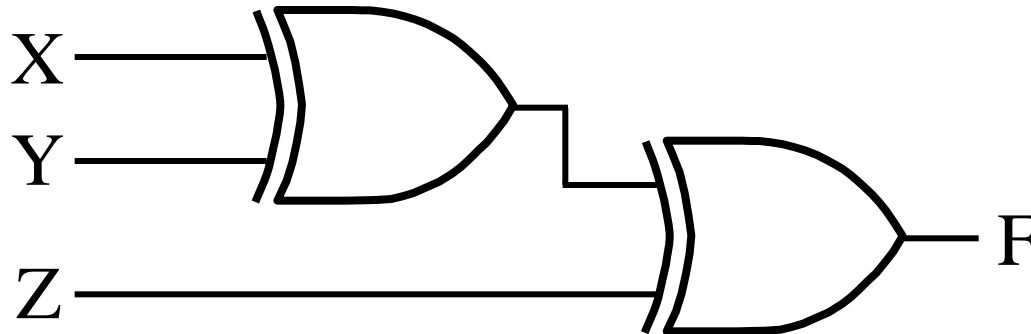
Example: Odd Function Implementation

- **Design a 4-input odd function $F = W \oplus X \oplus Y \oplus Z$ with 2-input XOR gates**

- **Factoring, $F = (W \oplus X) \oplus (Y \oplus Z)$**
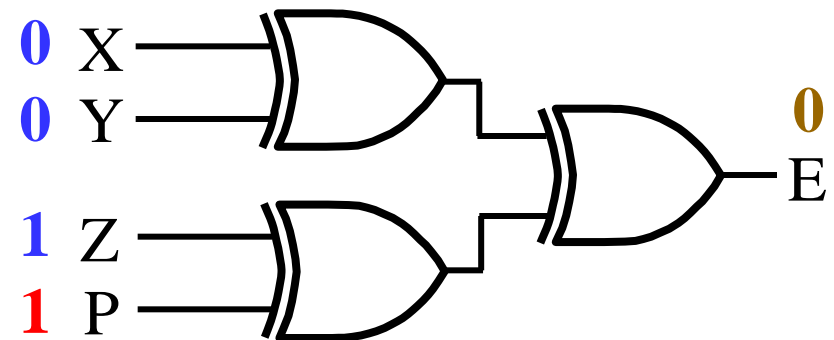
- **The circuit:**
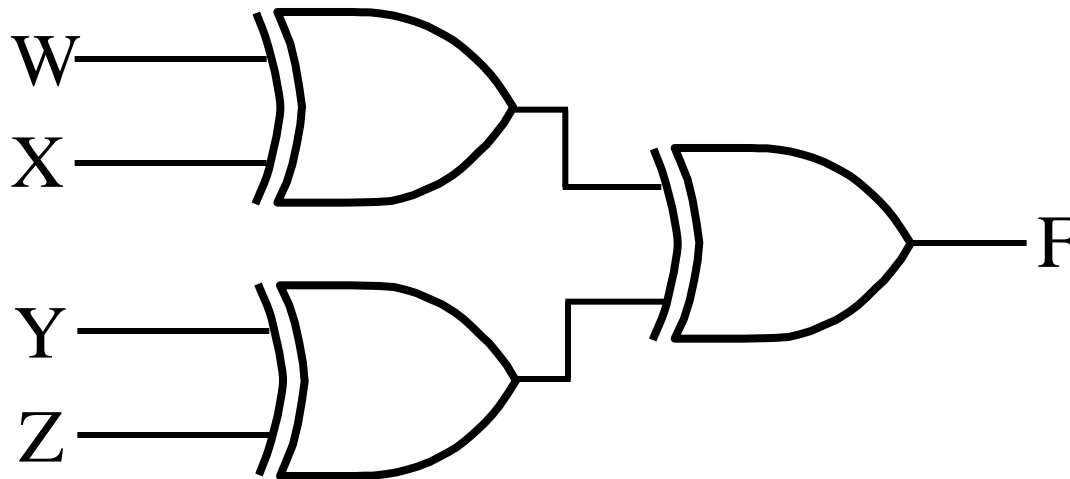
Parity Generator & Checkers
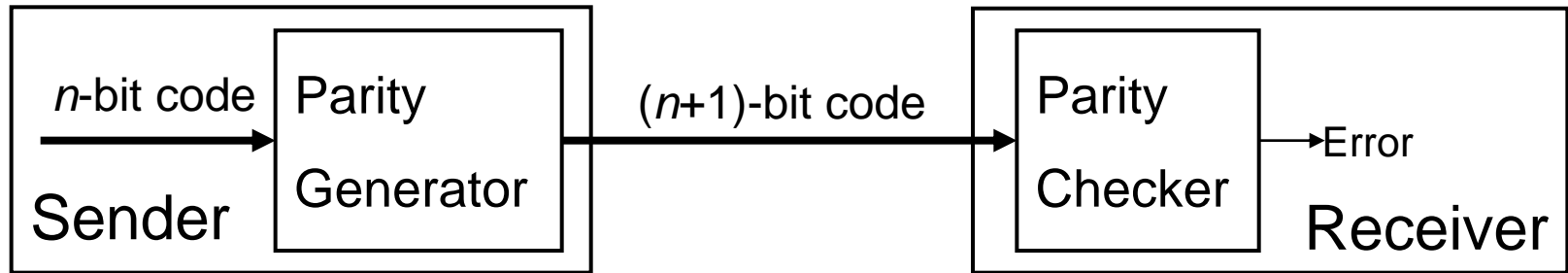
- **Even Parity bit: count of 1s in (n+1)-bit code is even**
  - **So use an odd function to generate the even parity bit**
- **To check for even parity**
  - **Use an odd function to check the (n+1)-bit code**
- **Odd Parity bit: count of 1s in (n+1)-bit code is odd**
  - **So use an even function to generate the odd parity bit**
- **To check for odd parity**
  - **Use an even function to check the (n+1)-bit code**

Parity Generator & Checkers

- **Design an even parity generator and checker for 3-bit codes**
- **Solution: Use 3-bit odd function to generate even parity bit**
- **Use 4-bit odd function to check for errors in even parity codes**
- **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**
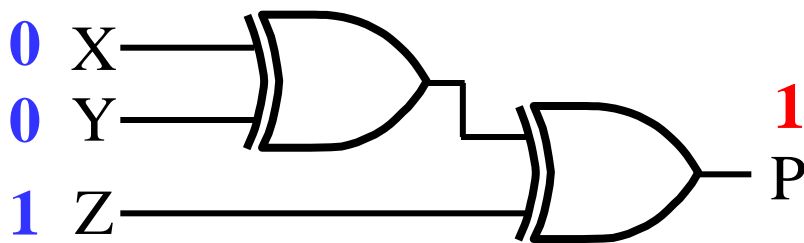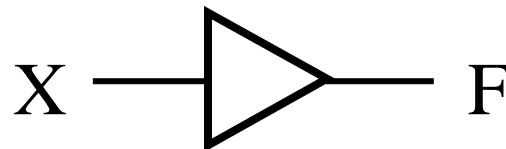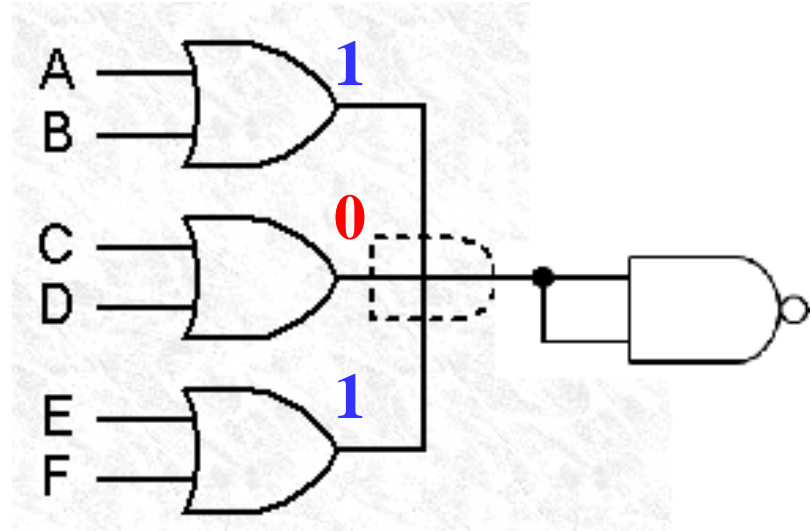

even parity generator

even parity checker

Buffer
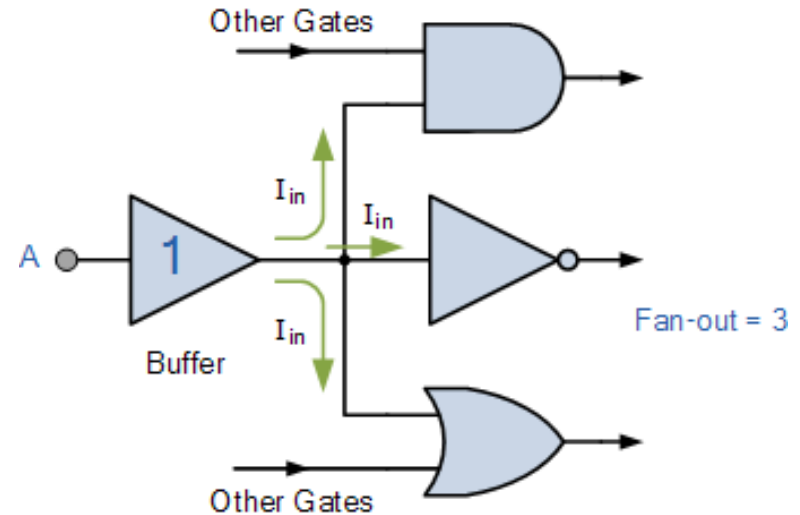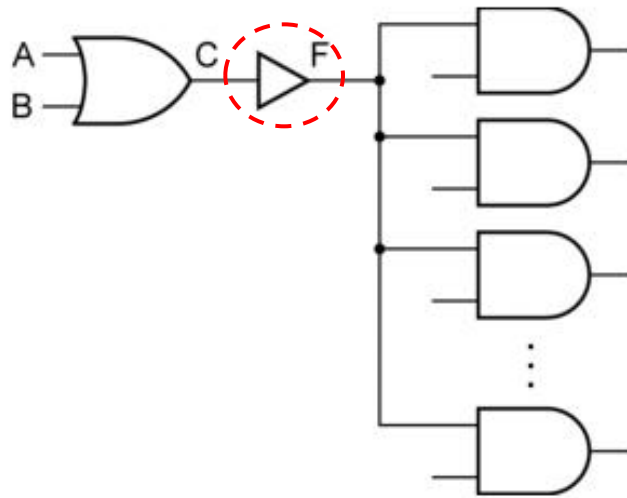
- **A buffer is an electronic amplifier used to improve circuit voltage levels and increase the speed of circuit operation.**

- **The buffer is a gate with the function F = X**

$$X \longrightarrow\!\!\!\!\!\triangleright\!\!\!\!- F$$

Wired Output

- **Can we connect the output of multiple gates together?**

- **Logic gates introduced thus far …**
  - **Have 1 and 0 output values**
  - **Cannot have their outputs connected together**

Wired Output (continued)

■ **In the case of two inverters wired together, if one is logic high and the other is logic low, the current will flow freely from VDD to ground!**
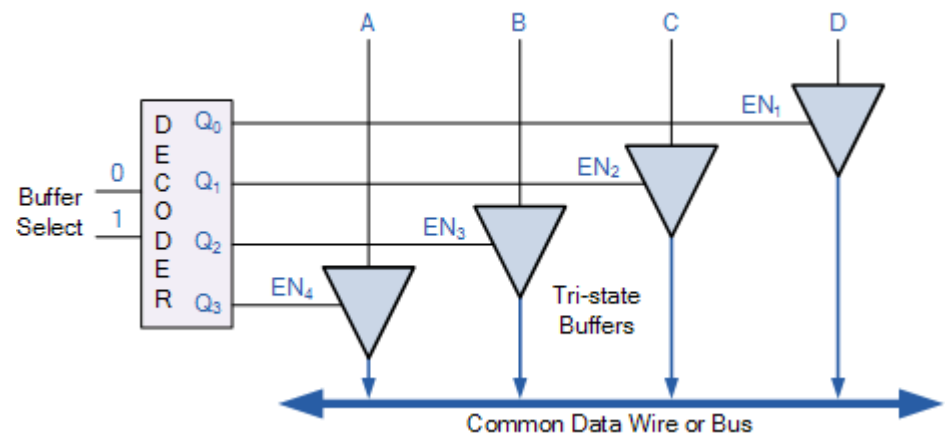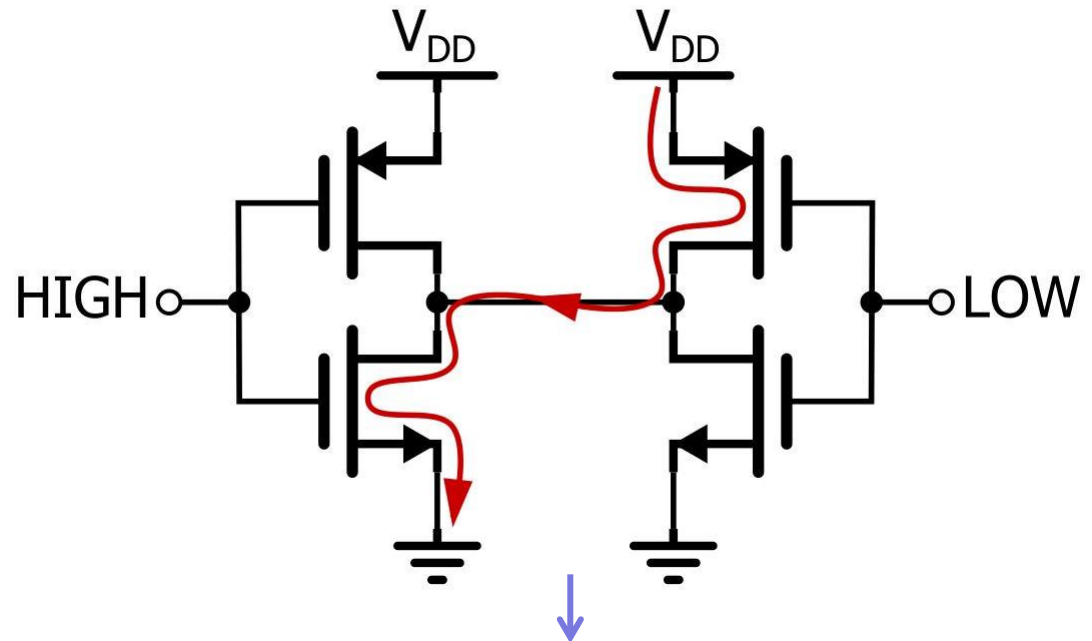
The 3-State Buffer
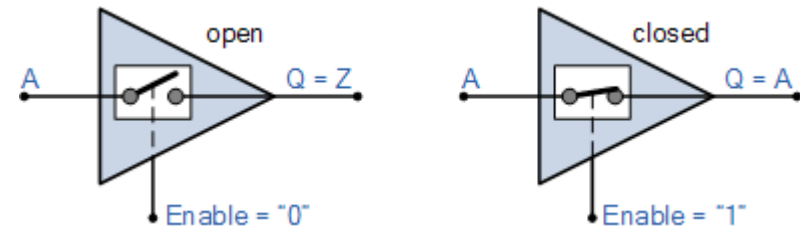
- **Three-state logic** adds a third logic value:

  - Hi-Impedance output: **Hi-Z**

- **What is Hi-Impedance output?**

  - The output appears to be **disconnected** from the input

 - Behaves as an **open circuit** between gate input & output

- **Hi-Z state makes a gate output behave differently:**

  - Three output values: 1, 0, and Hi-Z

  - Hi-impedance gates **can connect** their outputs together

The 3-State Buffer

- **IN = data input**

- **EN = Enable control input**

- **OUT = data output**

- **If EN = 0, then OUT = Hi-Z**
  - **Regardless of the value on IN**
  - **Output disconnected from input**

- **If EN = 1, then OUT =IN**
  - **Output follows the input value**

- **Variations:**
  - **EN can be inverted**
 - **OUT can be inverted**
  - **By addition of bubbles to signals**

Symbol

IN ———▷——— OUT

EN
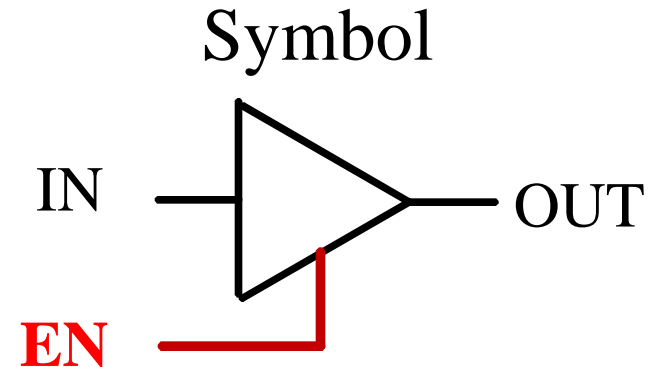
Truth Table

EN	IN	OUT
0	X	Hi-Z
1	0	0
1	1	1

The 3-State Buffer: Resolving Output Value

- **The output of 3-state buffers can be wired together**

- **At most one 3-state buffer can be enabled. Resolved output is equal to the output of the enabled 3-state buffer**

- **If multiple 3-state buffers are enabled at the same time then conflicting outputs will burn the circuit**
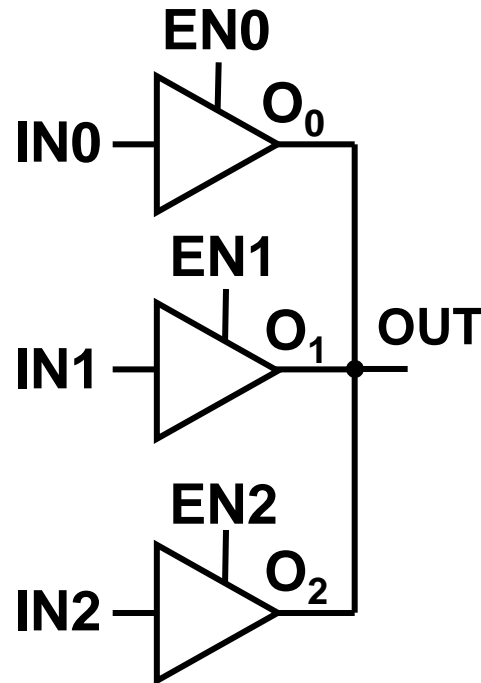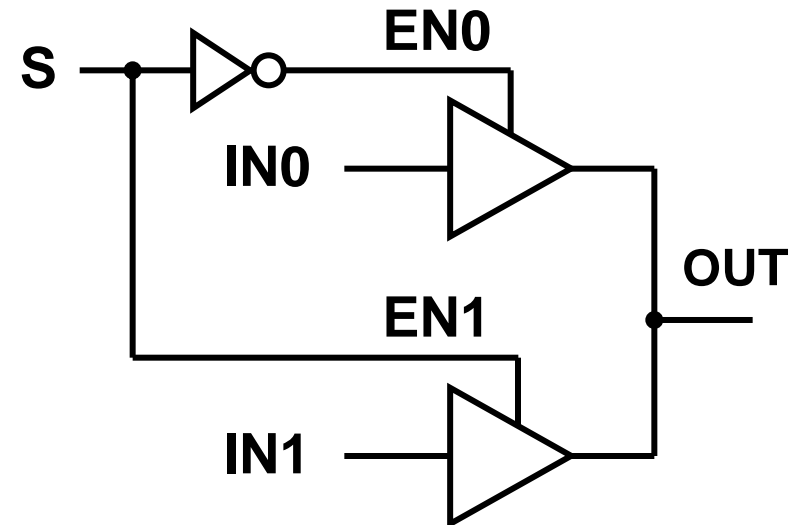

Resolution Table			
O_0	O_1	O_2	OUT
0 or 1	Hi-Z	Hi-Z	O0
Hi-Z	0 or 1	Hi-Z	O1
Hi-Z	Hi-Z	0 or 1	O2
Hi-Z	Hi-Z	Hi-Z	Hi-Z
0 or 1	0 or 1	0 or 1	Burn

Data Selection Circuit

- **Performing data selection with 3-state buffers:**
 - **If S = 0 then OUT = IN0 else OUT = IN1**

| S	EN0	EN1	IN0	IN1	OUT
0	1	0	0	X	0
0	1	0	1	X	1
1	0	1	X	0	0
1	0	1	X	1	1

- **The outputs of the 3-state buffers are wired together**
- **Since EN0 = $\overline{S}$ and EN1 = S, one of the two buffer outputs is always Hi-Z**

More Complex Gates

- **The remaining complex gates are SOP or POS structures with and without an output inverter.**

- **The names are derived using:**
  - **A - AND**
  - **O - OR**
  - **I - Inverter**

2-2 AOI   **2-1 AOI**

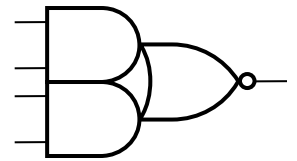 - **Numbers of inputs** **on first-level "gates" or directly to second-level "gates"**
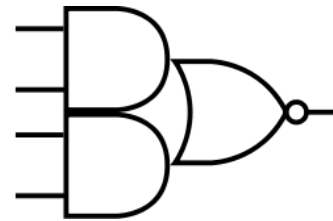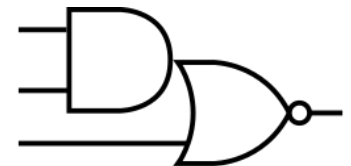
More Complex Gates (continued)

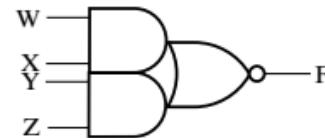- **Example: 2-2 AOI consists of two 2-input ANDS driving an OR function which is inverted.**

$$F = \overline{WX + YZ}$$

- **Example: 2-2-1 AO has two 2-input ANDS driving an OR with one additional OR input.**
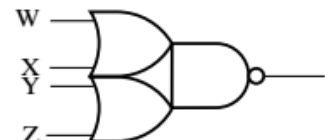
$$F = WX + YZ + V$$
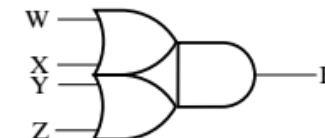
AND-OR-INVERT (AOI)	W, X, Y, Z	$F = \overline{WX + YZ}$
OR-AND -INVERT (OAI)	W, X, Y, Z	$F = \overline{(W + X)(Y + Z)}$
AND-OR (AO)	W, X, Y, Z	$F = WX + YZ$
OR-AND (OA)	W, X, Y, Z	$F = (W + X)(Y + Z)$

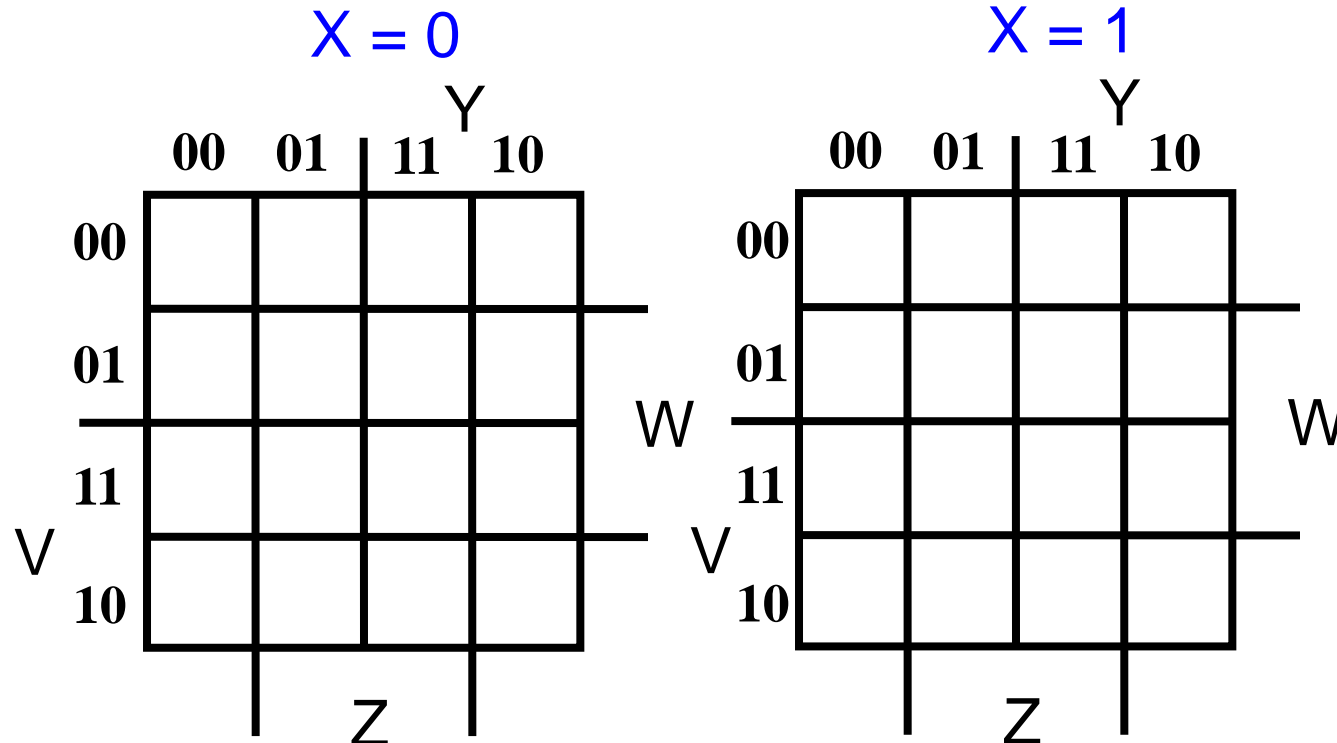# Assignments

Reading:

- **2.4-2.6**

Problem assignment:
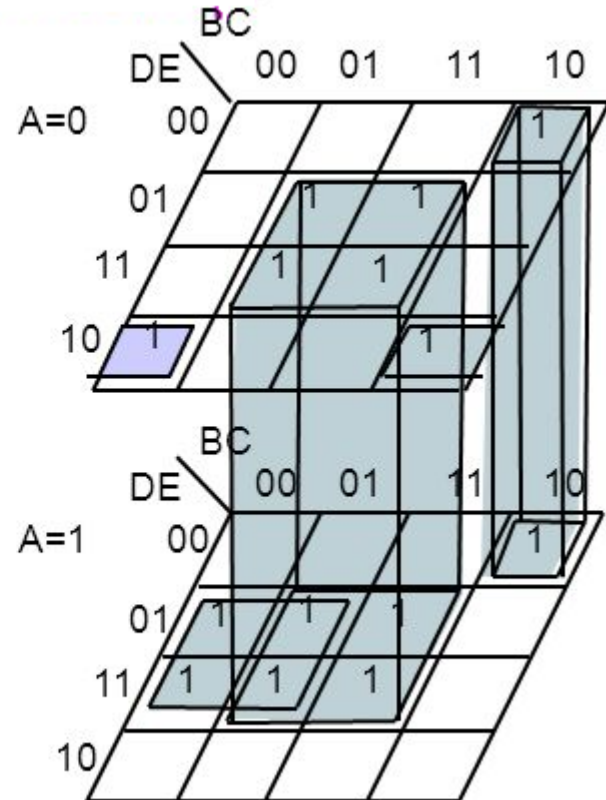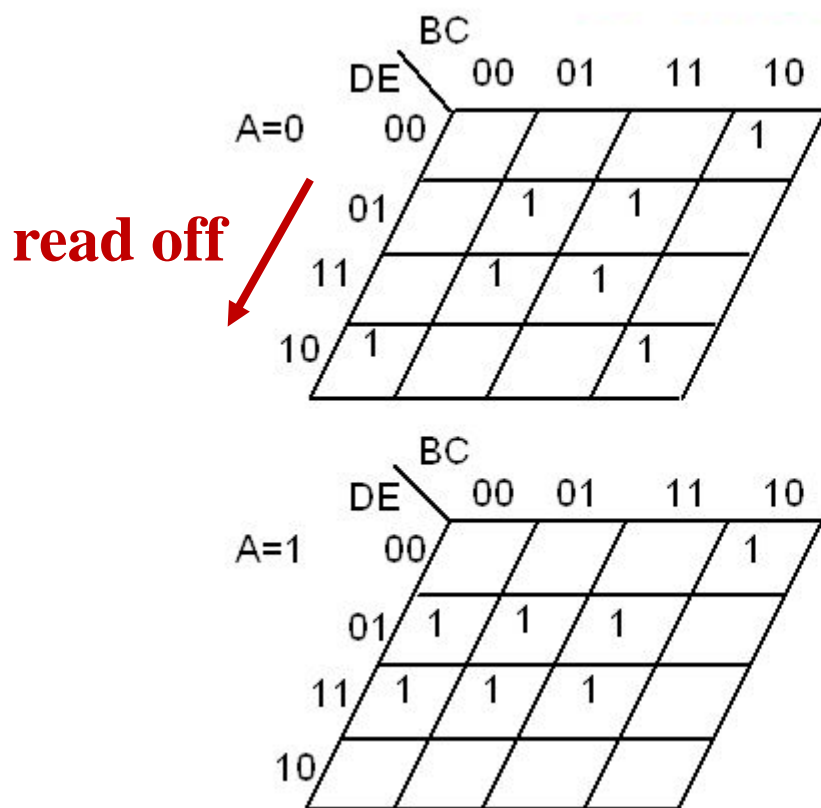
- **2-15c; 2-17b; 2-19a; 2-22a; 2-25b**

Appendix A: Five Variable Maps

- **For five variable problems, we use *two adjacent K-maps*. It becomes harder to visualize adjacent minterms for selecting PIs. You can extend the problem to six variables by using four K-Maps.**

Five Variable Maps

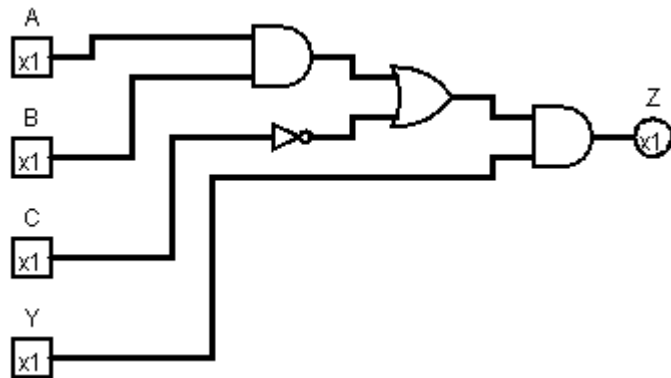- **The overlay mode of the Karnaugh map**

read off

$$F(ABCDE) = \sum_m (2, 5, 7, 8, 10, 13, 15, 17, 19, 21, 23, 24, 29, 31)$$

$$F(ABCDE) = CE + A\overline{B}E + B\overline{C}\,\overline{D}\,\overline{E} + \overline{A}\,\overline{C}D\overline{E}$$

Appendix B: Don't Cares in K-Maps

- **Example 2: A circuit that represents a very common situation that has two distinct sets of input variables and a single output Z:**

if $\quad AB + \overline{C} = 1$ **then** $\quad Z = Y$

else Z is a don't care

Some output values are not used!

- • **A, B, and C which take on all possible combinations, and**
- • **Y which takes on values 0 or 1.**

The circuit that receives the input Y observes it only for combinations of A, B, and C such A = 1 and B = 1 or C = 0, otherwise ignoring it.

- **Thus, Z is specified only for those combinations, and for all other combinations of A, B, and C, Z is a don't care.**

- **Any minterm with value "x" *need not* be covered by a prime implicant.**

Appendix C: Six Variable Maps
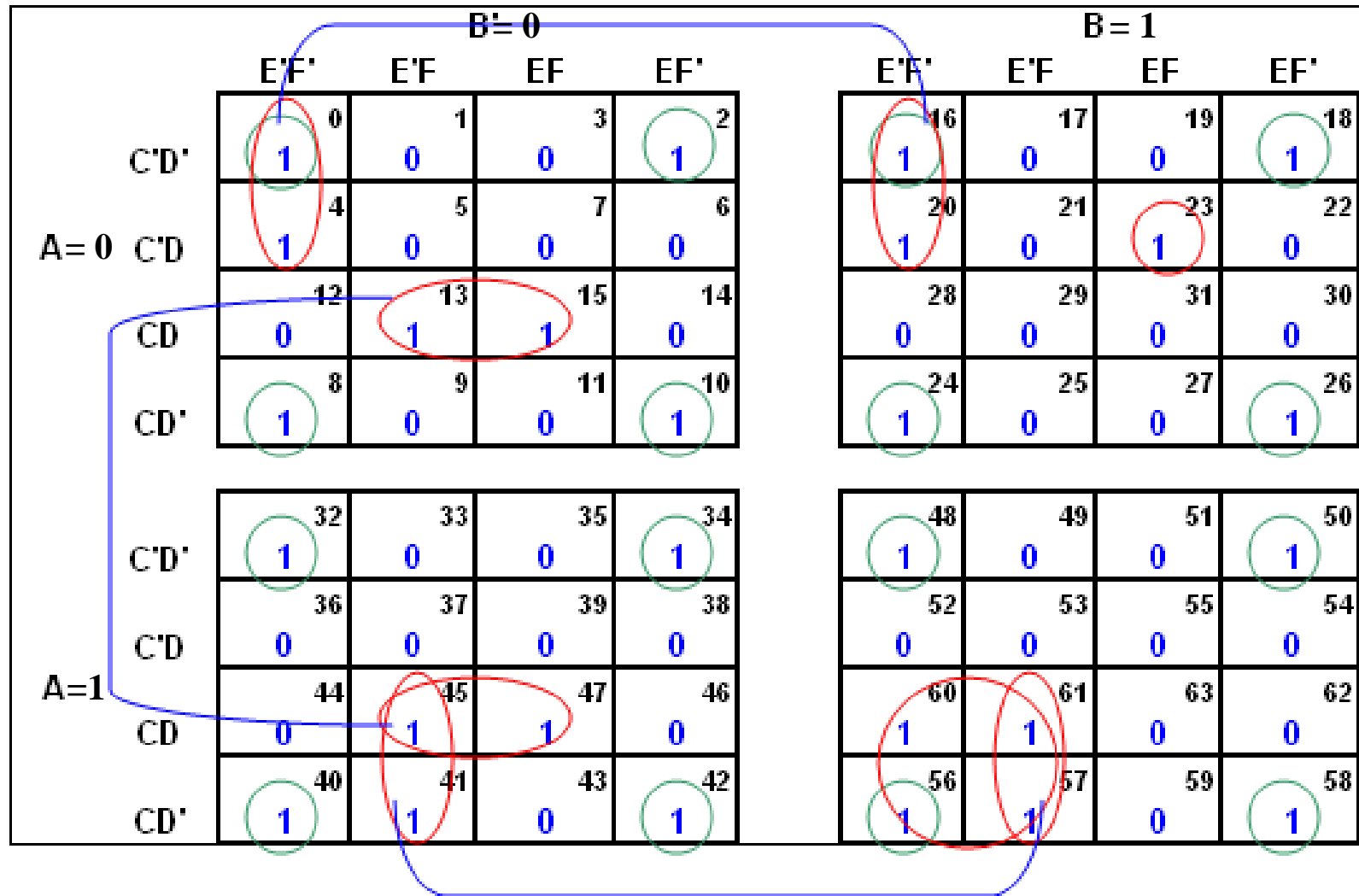
Appendix D: Quine–McCluskey Solver

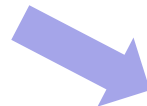- **Quine-McCluskey Solver http://www.quinemccluskey.com/**
- **Input**

- **Comparasion**

- **Prime Implicants**

- **Coverage Table**

- **Essential Prime Implicants**

Resolving 3-State Values on a Connection

- **Connection** of two 3-state buffer outputs, B1 and B0, to a wire, OUT

- **Assumption: Buffer data inputs can take any combination of values (0/1)**

- **Resulting Rule: At least one buffer output value must be Hi-Z.**

- **How many valid buffer output combinations exist?** **5**

- **What is the rule for n 3-state buffers connected to wire, OUT?**

  n-1 buffer outputs must be Hi-Z

- **How many valid buffer output combinations exist?** **2n+1**

Resolution Table		
B1	**B0**	**OUT**
0	Hi-Z	0
1	Hi-Z	1
Hi-Z	0	0
Hi-Z	1	1
Hi-Z	Hi-Z	Hi-Z

Appendix E: Bidirectional

- **Bidirectional means the data incoming and outgoing data flows over the same channel.**

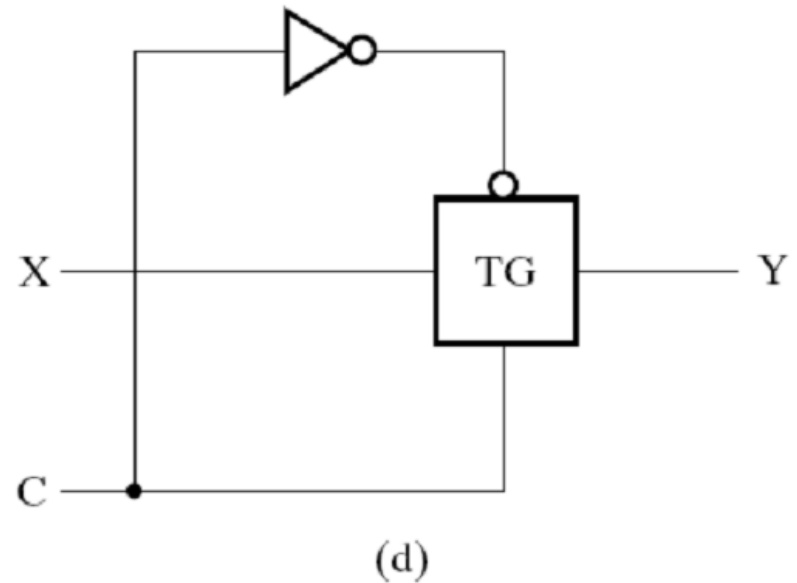- **3-state buffers are always used to create bidirectional.**
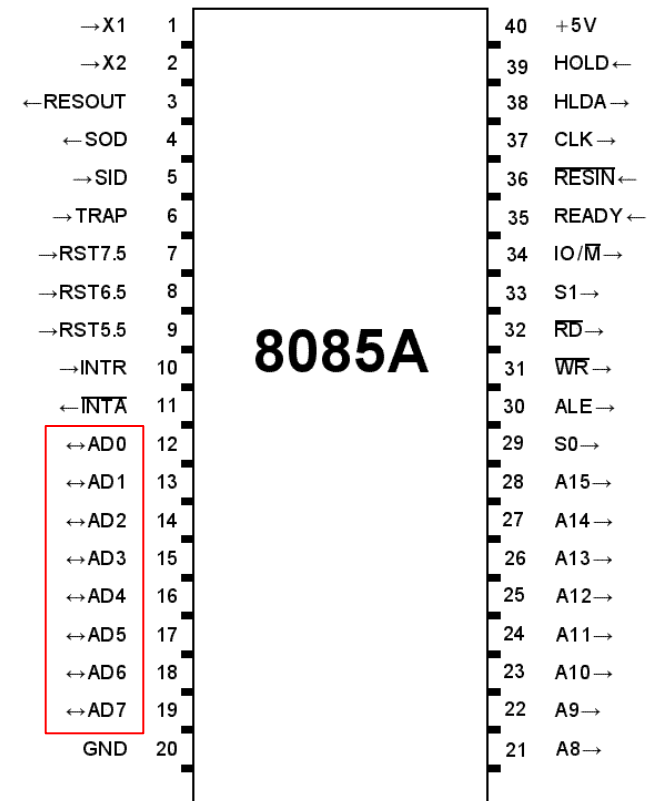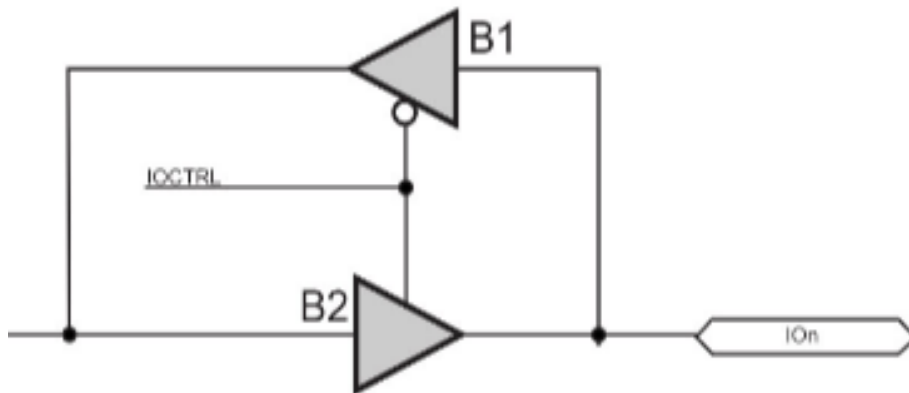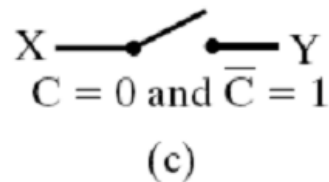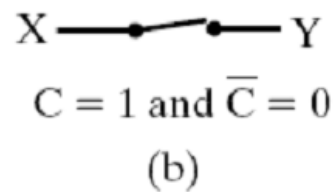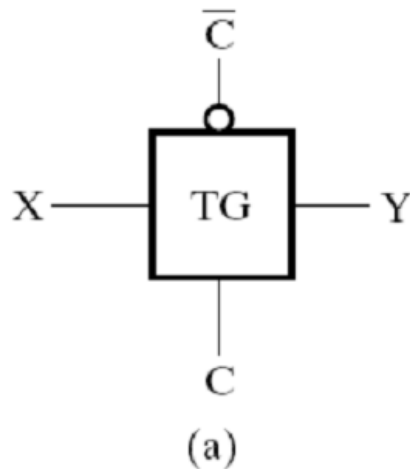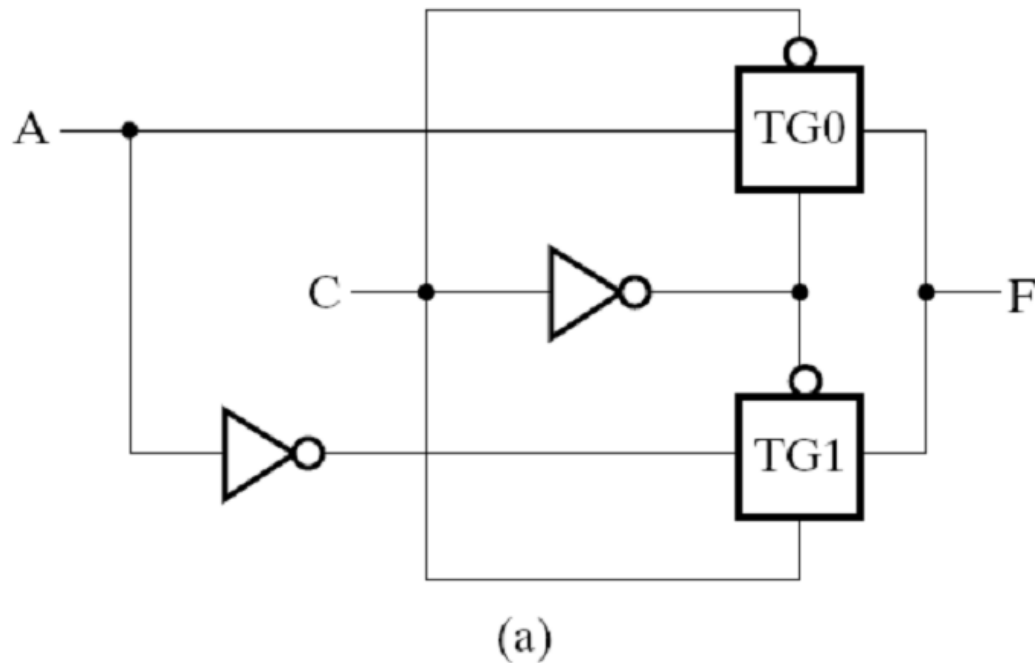

Pin 12-19

- multiplexed addr/data bus
- bidirectional

Appendix F: Transmission gate

- **A transmission gate is similar to a relay that can conduct in both directions or block by a control signal with almost any voltage potential.**

XOR Function with transmission gate

A	C	TG1	TG0	F
0	0	No path	Path	0
0	1	Path	No path	1
1	0	No path	Path	1
1	1	Path	No path	0

(a)   (b)