20
24

# Digital Systems
## Section 2

## Chapter (3)

Mohammed Khalil

# *Gate-Level Minimization*

- A Boolean **Function** is **uniquely** represented by a **truth table**
- Boolean **Function** can be implemented (**NOT Uniquely**) by a Boolean **Equation** and the corresponding **logic diagram**
- Simplest Functions use the **smallest** number of the **smallest** gates and therefore give the *most economical* and *efficient* circuit implementations

Requires: Minimization

- Boolean **Function** can be **simplified** by algebraic methods learned earlier
  - This process is **not** always **straight-forward** and may **not** result in the **simplest** form of an expression
- A **formal** approach for simplification is needed (**systematic** procedure)

**The Map Method**

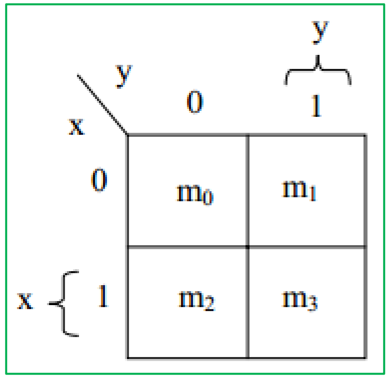
# *The Map Method – Karnaugh Map*

- A **Straight-forward/Simpler** method to achieve minimization **systemically** **K-Map**
  - **Graphical** representation of a Truth Table

- A K-map is a **diagram** made up of **squares** representing **minterms**
  - K-map for **n** variables is a collection of **$2^n$** squares/cells [**n variables → $2^n$ minterms**]
  - Each **Square/Cell → Minterm**
  - **Squares** arranged such that physically **adjacent cells** differ in the value of **only one literal**

- Different **patterns** in this diagram can be detected to simplify expressions
  - **Adjacent** minterms can be **combined** to form simpler terms

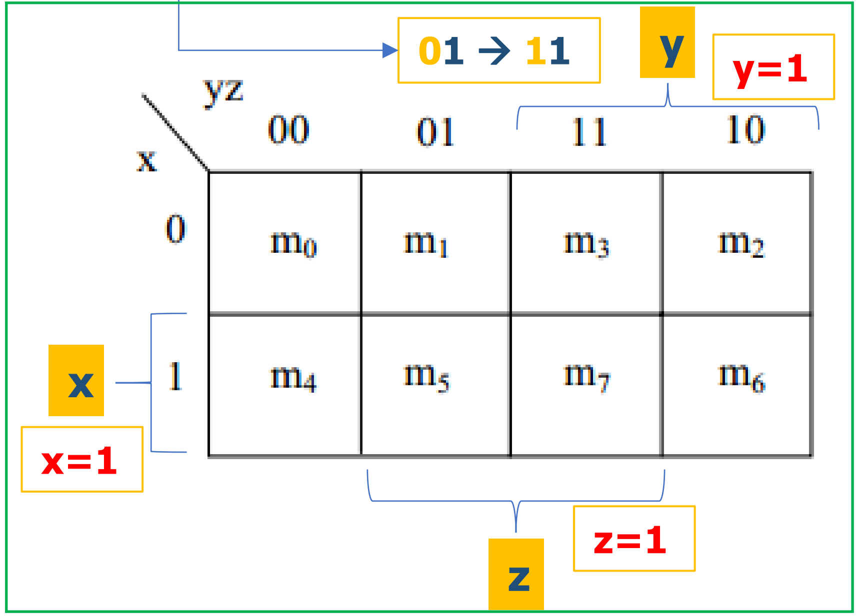
- The **simplified** expression will always be in sum-of-products or product-of-sums form



- K-Map produces a circuit diagram with **minimum** number of **gates**

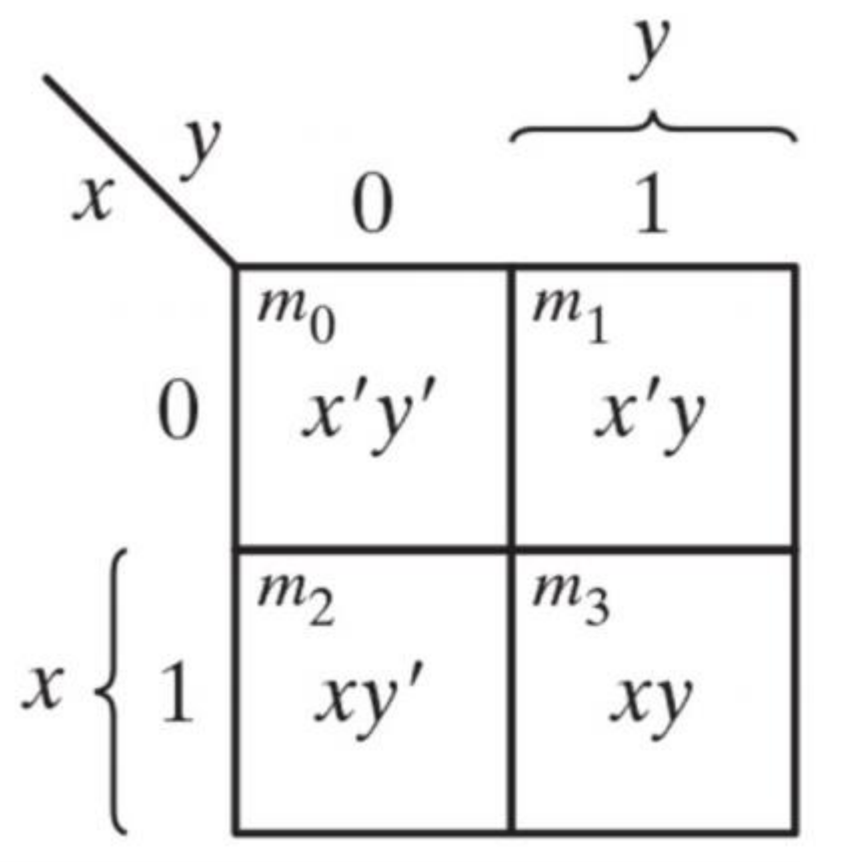
- K-Map produces circuits with gates having **minimum** number of **inputs**

- The simplest expression is **not unique** – two or more optimal expressions may exist



# Two-Variable K-Maps

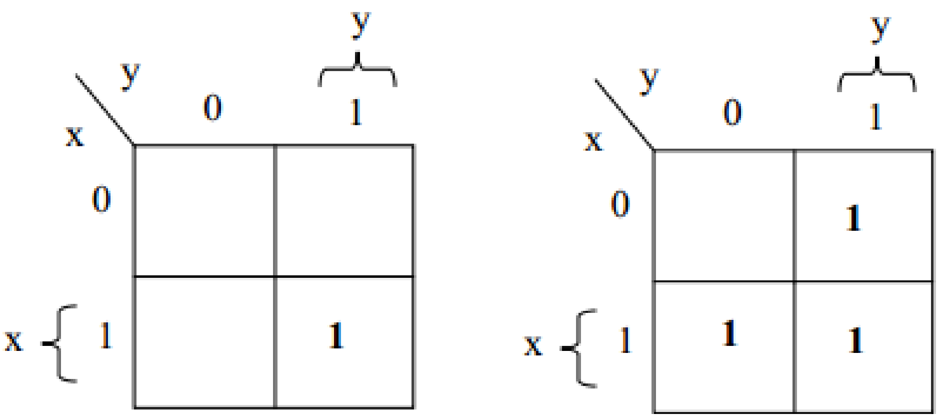
- Boolean functions having **two** variables x and y
- There are **$2^2$ = 4** minterms for two variables $xy, x'y, xy', x'y'$
- A K-map for **two** variables will have **four squares** → Each cell will represent a minterm



Each **cell** represents the **minterm**
of the corresponding **row** in the truth table

# Two-Variable K-Maps

## Example:



$F = x.y$

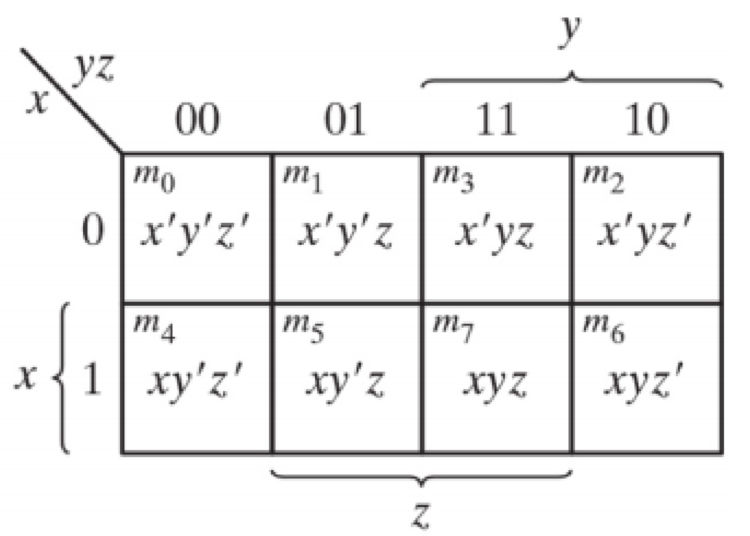
$F = x + y$

# Three-Variable K-Maps

- Boolean functions having **three** variables x,y and z
- There are **$2^3$ = 8** minterms for three variables

$$x'y'z',\ x'y'z,\ x'yz,\ x'yz',\ xy'z',\ xy'z,\ xyz,\ xyz'$$

- A K-map for **three** variables will have **eight squares** → Each cell will represent a minterm



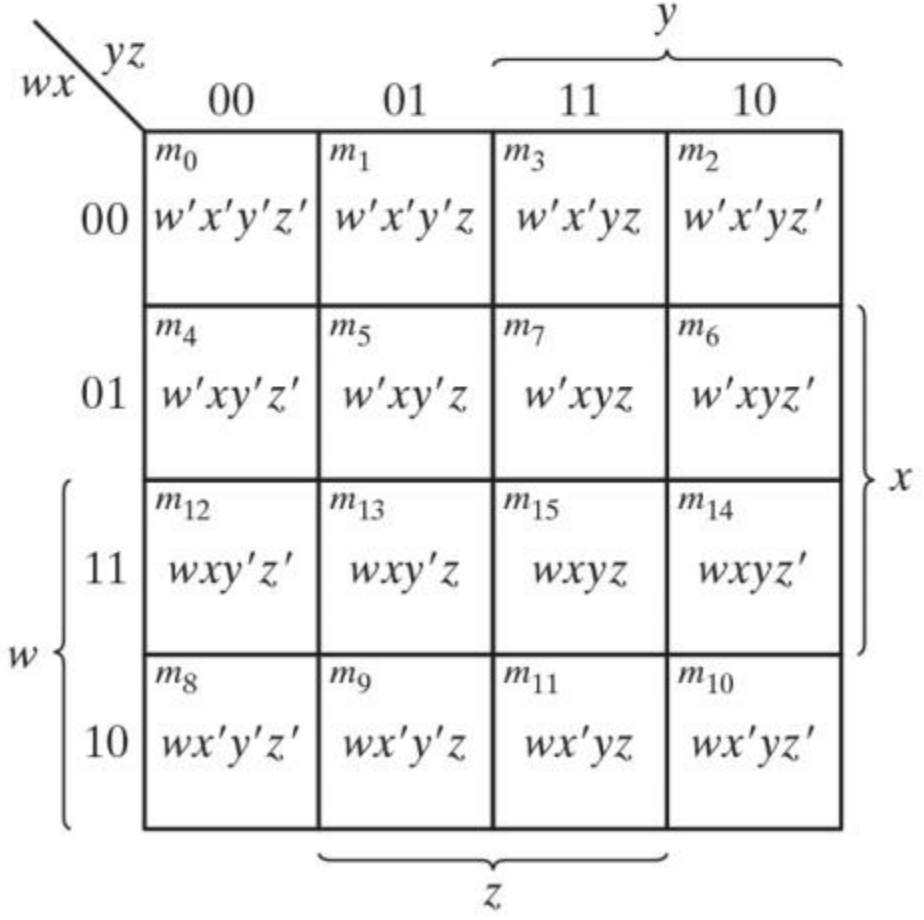
| x \ yz | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | $m_0$<br>$x'y'z'$ | $m_1$<br>$x'y'z$ | $m_3$<br>$x'yz$ | $m_2$<br>$x'yz'$ |
| 1 | $m_4$<br>$xy'z'$ | $m_5$<br>$xy'z$ | $m_7$<br>$xyz$ | $m_6$<br>$xyz'$ |

**Be Careful:**
The order is **not** sequential
$m_3$ before $m_2$
$m_7$ before $m_6$

Each **cell** represents the **minterm** of the corresponding **row** in the truth table

# Four-Variable K-Maps

- Boolean functions having **Four** variables w,x,y and z
- There are **$2^4$ = 16** minterms for Four variables
- A K-map for **Four** variables will have **sixteen squares** → Each cell will represent a minterm



| wx \ yz | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | $m_0$<br>$w'x'y'z'$ | $m_1$<br>$w'x'y'z$ | $m_3$<br>$w'x'yz$ | $m_2$<br>$w'x'yz'$ |
| 01 | $m_4$<br>$w'xy'z'$ | $m_5$<br>$w'xy'z$ | $m_7$<br>$w'xyz$ | $m_6$<br>$w'xyz'$ |
| 11 | $m_{12}$<br>$wxy'z'$ | $m_{13}$<br>$wxy'z$ | $m_{15}$<br>$wxyz$ | $m_{14}$<br>$wxyz'$ |
| 10 | $m_8$<br>$wx'y'z'$ | $m_9$<br>$wx'y'z$ | $m_{11}$<br>$wx'yz$ | $m_{10}$<br>$wx'yz'$ |

**Be Careful:**
The order is **not** sequential
$m_3$ before $m_2$
$m_7$ before $m_6$
$m_{12} – m_{15}$ before $m_8 – m_{11}$

Each **cell** represents the **minterm** of the corresponding **row** in the truth table

# K-Mapping Procedure

- **Construct** the corresponding map (based on number of variables)
- **Enter** function **output (1's)** values on the map (from Truth Table or Canonical Form) to the corresponding cell/square

**Map** the following Function on a K-map

$$F_1(A, B, C) = A'B'C + A'BC' + ABC' + ABC$$

Canonical Form

1) Three variables → $2^3$ = **8**-cell K-map
2) Place a **1** on the K-map in the cell having the **same** minterm index/value

$$A'B'C = 001, \; A'BC' = 010, \; ABC' = 110, \; ABC = 111$$

Primed → 0
Unprimed → 1

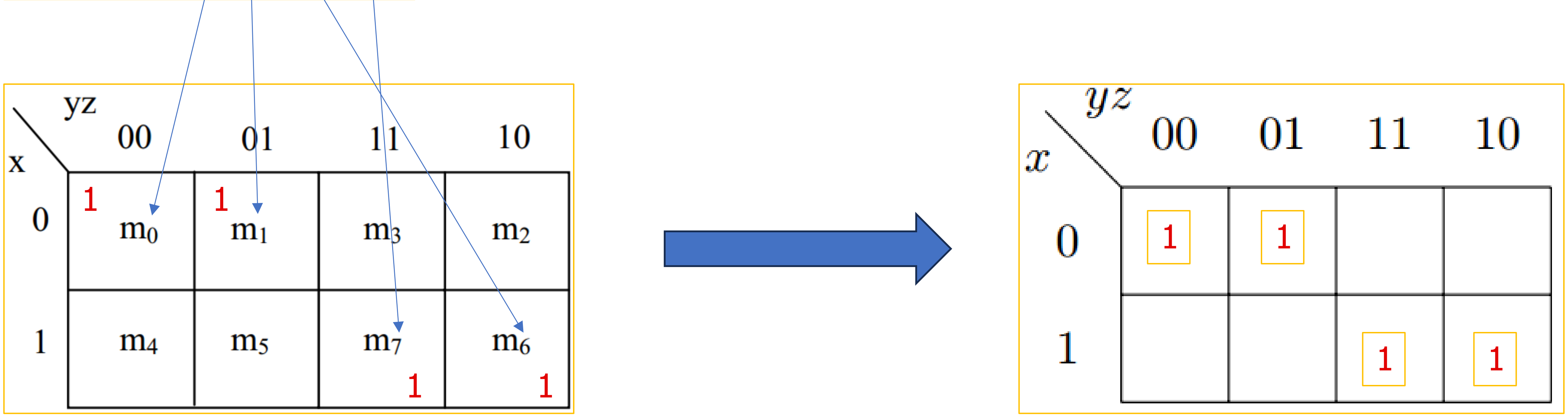
| $A \backslash BC$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | 1 | | 1 |
| 1 | | | 1 | 1 |

**Be Careful:**
The order is **not** sequential
$m_3$ before $m_2$
$m_7$ before $m_6$

# K-Mapping Procedure

**Example:**

$$F = \sum(0,1,6,7)$$



| x \ yz | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | | |
| 1 | | | 1 | 1 |

# K-Mapping Procedure

**Map** the following Function on a K-map

$$F_3(A,B,C) = A'C + A'BC' + ABC' + A$$

**Convert** to Canonical Form

**Expand** each term to represent *F₃* as a sum-of-minterms (SOM) form

$A'C$ will expand to $A'BC + A'B'C$

$A$ will expand to $ABC + ABC' + AB'C + AB'C'$

Therefore, $F_3(A,B,C) =$
$A'BC + A'B'C + A'BC' + ABC' + ABC + \cancel{ABC'} + AB'C + AB'C'$

$$F_3(A,B,C) = A'BC + A'B'C + A'BC' + ABC' + ABC + AB'C + AB'C'$$

011 001 010 110 111 101 100

Canonical Form

Primed → 0
Unprimed → 1

| $A \backslash BC$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**Be Careful:**
The order is **not** sequential
11 before 10

# K-Mapping Procedure

**Map** the following Function on a K-map

$$F_2(A,B,C,D) = A'BCD' + ABCD' + ABC'D' + ABCD$$

Canonical Form

1) Four variables → $2^4$ = **16**-cell K-map
2) Place a **1** on the K-map in the cell having the **same** minterm index/value

$$F_2(A,B,C,D) = \underset{0110}{A'BCD'} + \underset{1110}{ABCD'} + \underset{1100}{ABC'D'} + \underset{1111}{ABCD}$$

Primed → 0
Unprimed → 1

| $AB \backslash CD$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | | | | 1 |
| 11 | 1 | | 1 | 1 |
| 10 | | | | |

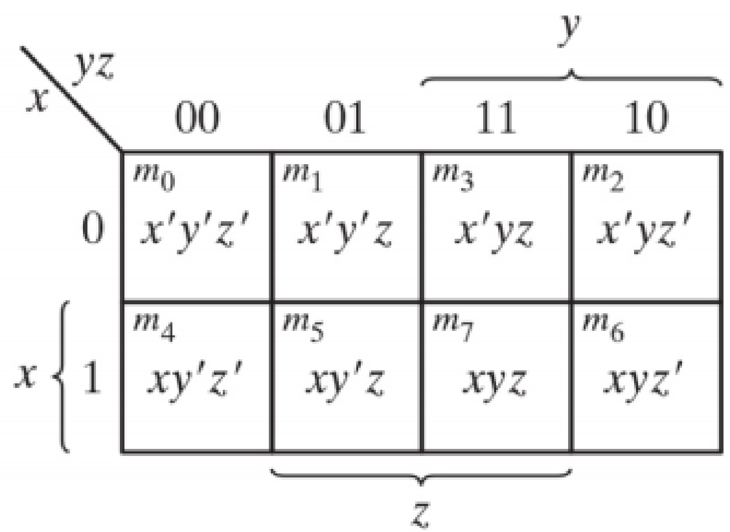
**Be Careful:**
The order is **not** sequential
11 before 10 (in Rows & Columns)

# Cell Adjacency

- **Single** variable changes in **adjacent** cells
- Cells that differ by only **one** variable are called **adjacent** cells
- Example:
  - 011 is adjacent to 010
  - 011 is **not** adjacent to 101
- **Wrap-around** adjacency:
  - Cells in the left-most column are **adjacent** to the cells in the right-most column (100 & 110)

What is the **sum** of minterms in two adjacent squares?

$$m_0 + m_4 = x'y'z' + xy'z'$$
$$= (x' + x)y'z' = y'z'$$
$$m_7 + m_6 = xyz + xyz'$$
$$= xy(z + z') = xy$$
$$m_0 + m_2 = x'y'z' + x'yz'$$
$$= x'z'(y' + y) = x'z'$$



| x \ yz | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | $m_0$ $x'y'z'$ | $m_1$ $x'y'z$ | $m_3$ $x'yz$ | $m_2$ $x'yz'$ |
| 1 | $m_4$ $xy'z'$ | $m_5$ $xy'z$ | $m_7$ $xyz$ | $m_6$ $xyz'$ |

**Sum** of two minterms in **adjacent** squares can be simplified to a **single product** term consisting of only **two literals**. The **dissimilar** variable will go away.

# *K-Map Simplification*

- Once a SOM expression has been **mapped** on the K-map, there are three steps in obtaining a simplified form
  1) **Group** 1's
  2) **Determine** the **product** term for each group
  3) **Sum** the resulting terms

- Group 1's with the following goal in mind: **Maximize** the **size** of the groups and **minimize** the **number** of groups

- Group 1's according to the following rules:
  - ✓ Group **size** must be powers of **2** (1, 2, 4, 8, or 16,.. Cells)
  - ✓ Each **cell** in a group must be **adjacent** to one or more cells in that same group.

    (***Not** all cells in a group have to be adjacent to each other*)
  - ✓ Always include the **largest** possible number of **1's** in a group
  - ✓ Each **1** on the map must be included in at **least one** group
  - ✓ The **1's** already in a group can be included in another group as long as the **overlapping** groups include **non-common 1's**

# K-Map Simplification - Groups of 1's

**Example:**

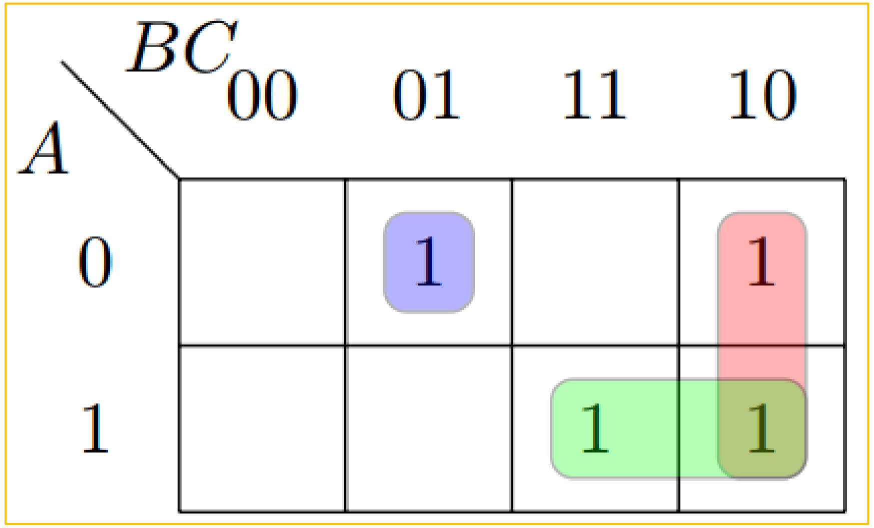
3-Variables

$$F = A'B'C + A'BC' + ABC + ABC'$$

Map

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 |  | 1 |  | 1 |
| 1 |  |  | 1 | 1 |

Group



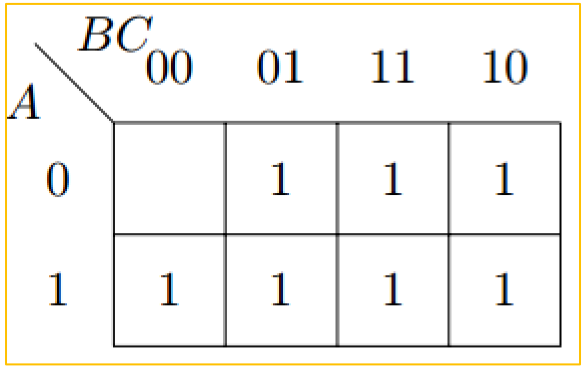
| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 |  | 1 |  | 1 |
| 1 |  |  | 1 | 1 |

# K-Map Simplification - Groups of 1's

**Example:**

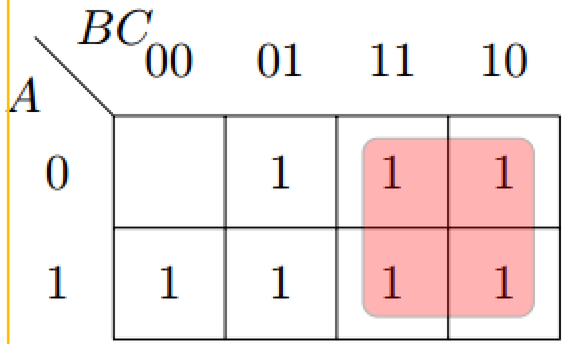
$$F(A,B,C)=\Sigma(1,2,3,4,5,6,7)$$

**Map**

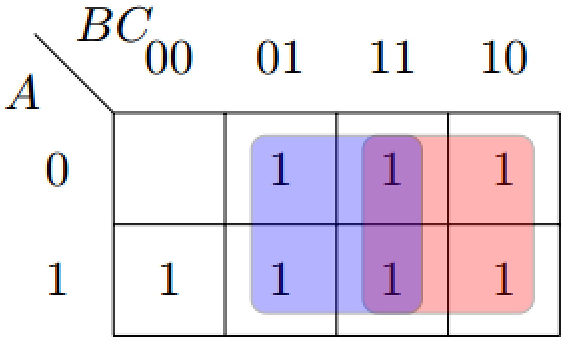


| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

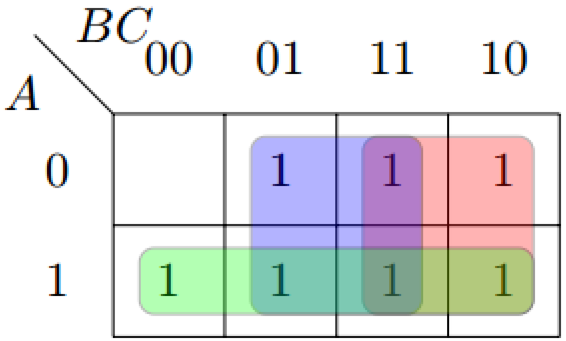
**Group**



| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |



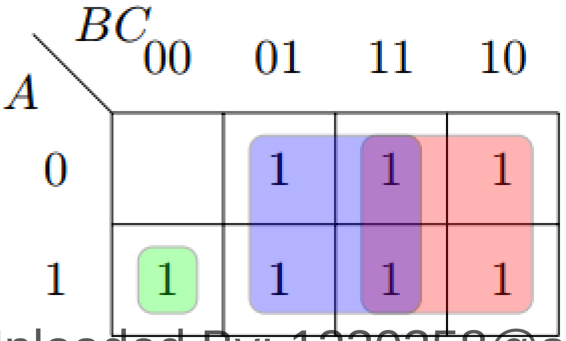
| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |



| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

✓ **Maximize** Group Size

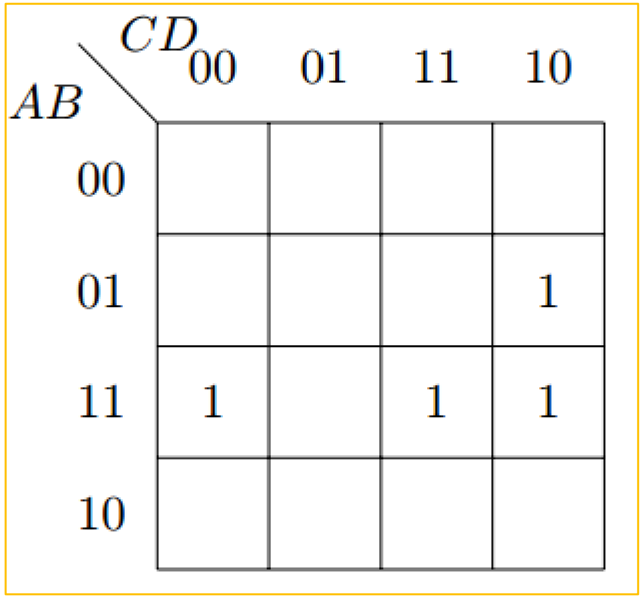
2 - Choices



| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

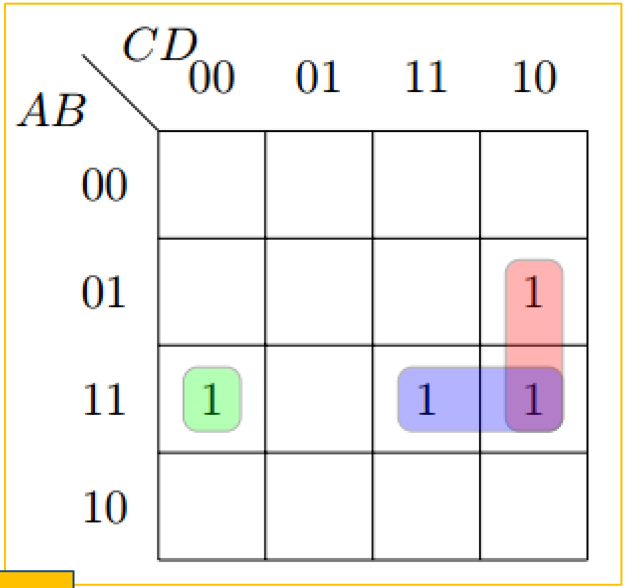
# K-Map Simplification - Groups of 1's

**Example:**



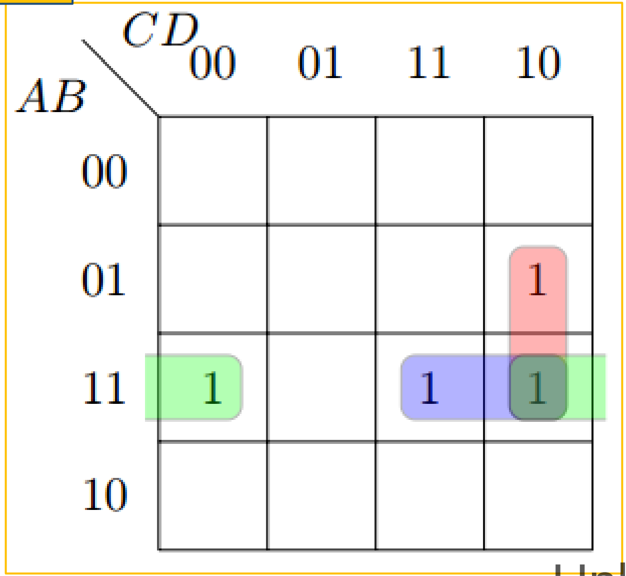
| AB \ CD | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | | | | 1 |
| 11 | 1 | | 1 | 1 |
| 10 | | | | |

2 - Choices



| AB \ CD | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | | | | 1 |
| 11 | 1 | | 1 | 1 |
| 10 | | | | |

☒ **NOT**
Optimal

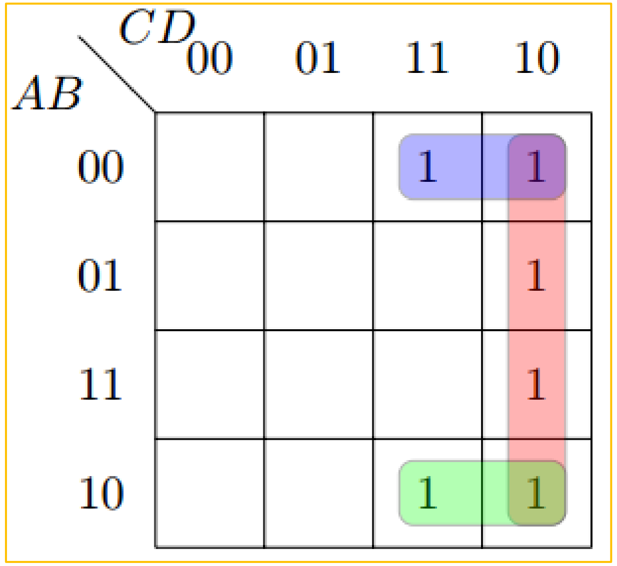


| AB \ CD | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | | | | 1 |
| 11 | 1 | | 1 | 1 |
| 10 | | | | |

✓ **Maximize**
Group Size

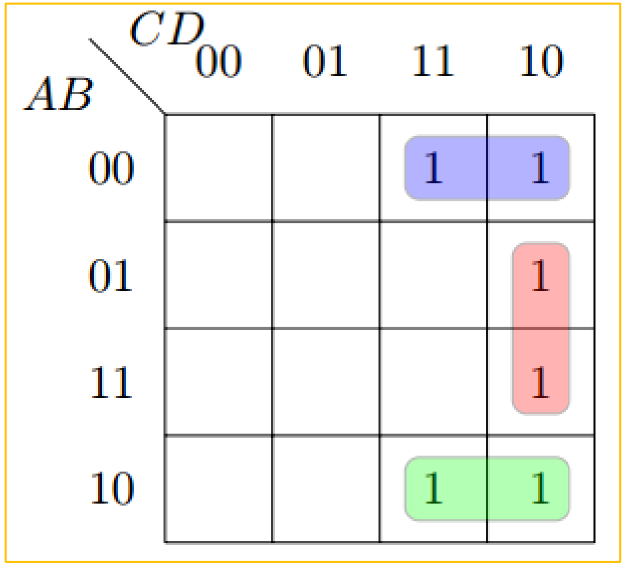
# *K-Map Simplification - Groups of 1's*

**Example:**



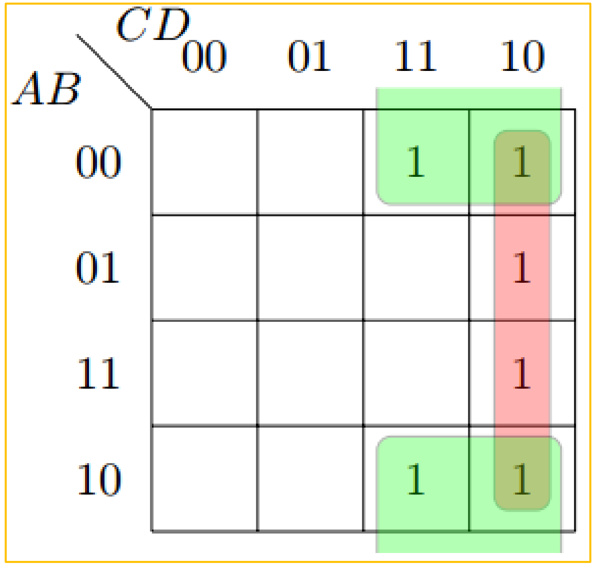
| $AB$ \ $CD$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | 1 | 1 |
| 01 | | | | 1 |
| 11 | | | | 1 |
| 10 | | | 1 | 1 |

☒ **NOT**
Optimal



| $AB$ \ $CD$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | 1 | 1 |
| 01 | | | | 1 |
| 11 | | | | 1 |
| 10 | | | 1 | 1 |

✓ **Maximize**
Group Size

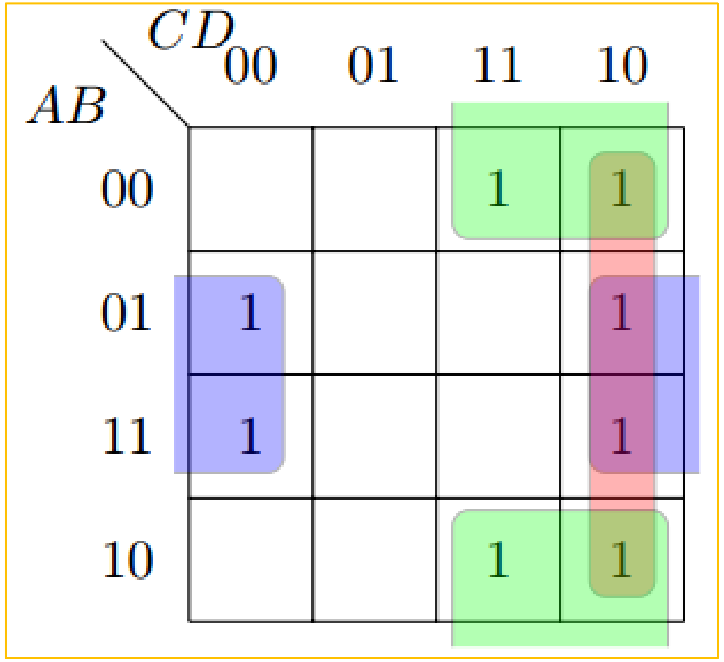


| $AB$ \ $CD$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | 1 | 1 |
| 01 | | | | 1 |
| 11 | | | | 1 |
| 10 | | | 1 | 1 |

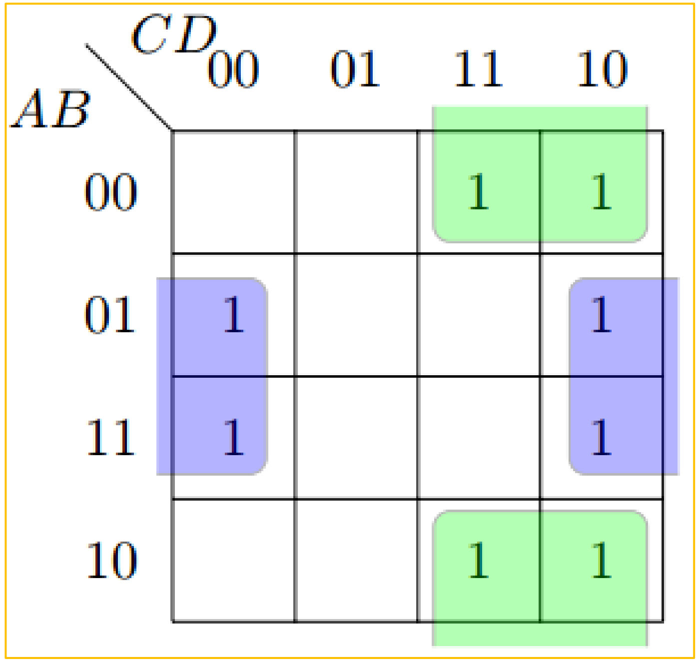
# K-Map Simplification - Groups of 1's

**Example:**

**☒ Unnecessary/Duplications**



**✓ Optimal/No Duplication**

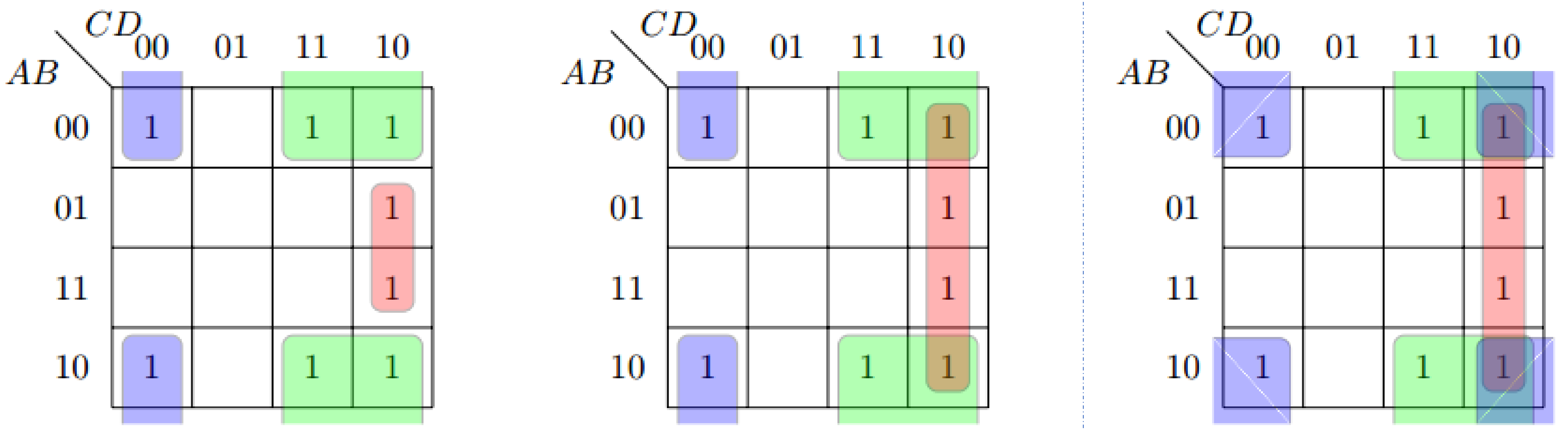


# K-Map Simplification - Groups of 1's

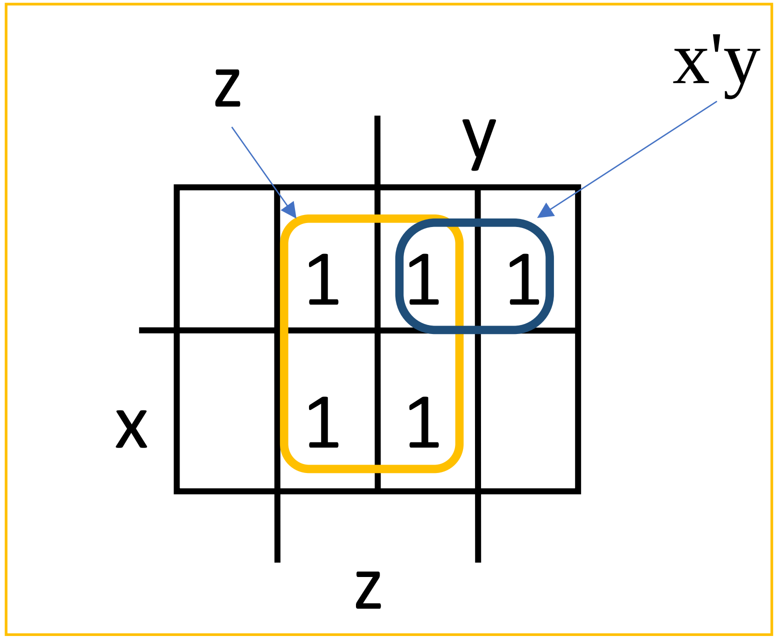
**Example:**

☒ Groups of (2 Minterms) → Not **Maximized Size**

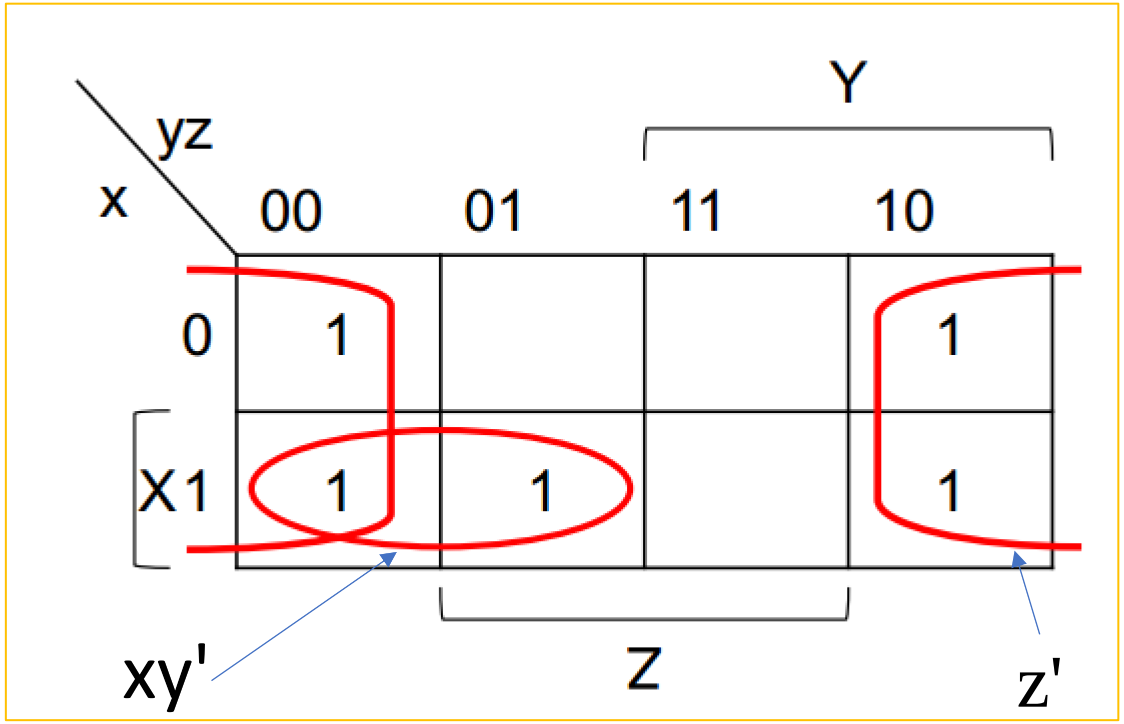
✓ **Maximize** Group Size



**Examples:**



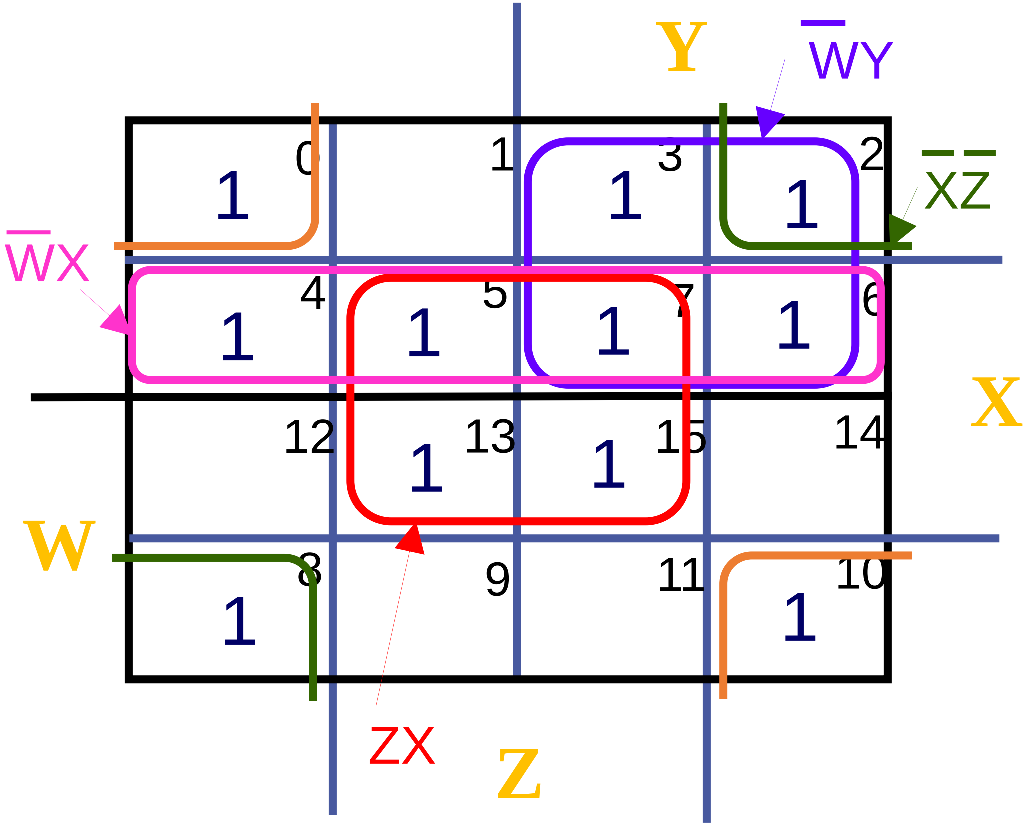
$$F(x, y, z) = z + x'y$$



$$F = z' + xy'$$

**Examples:**

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



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

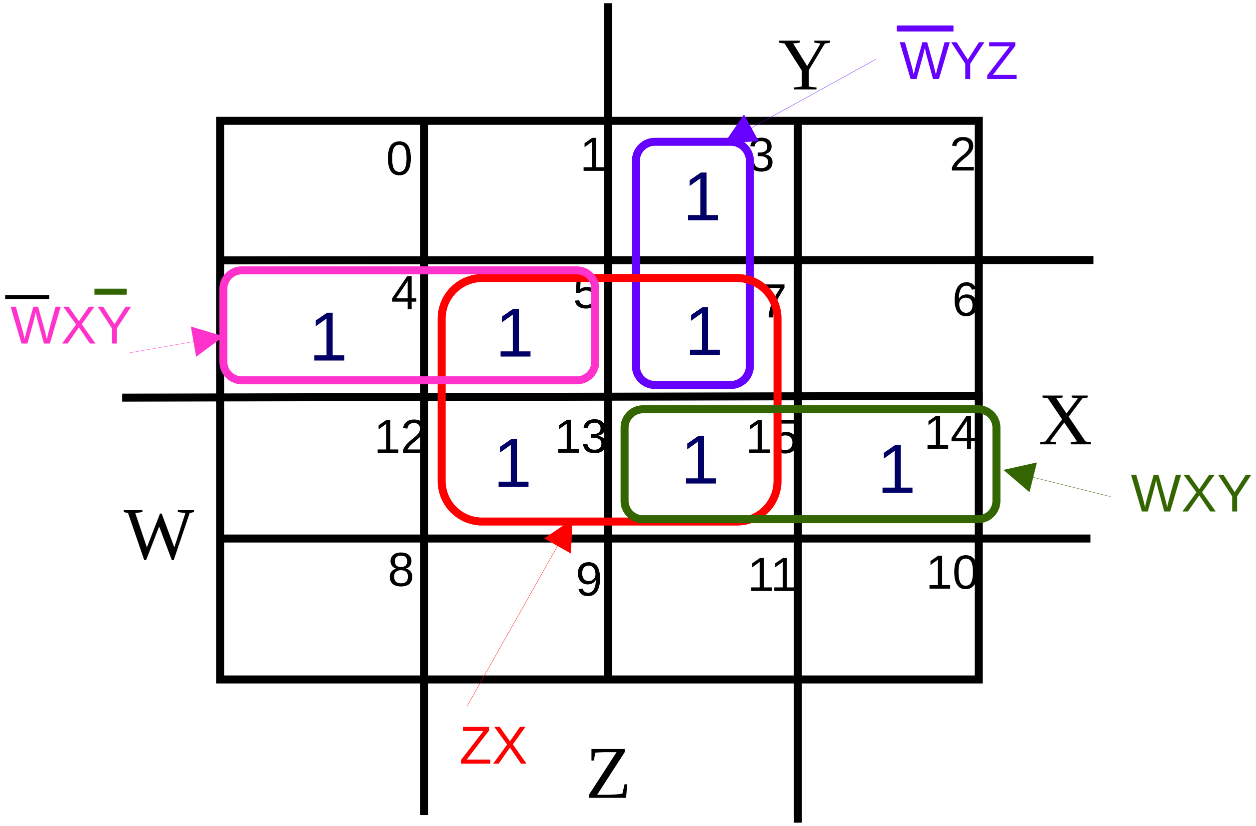
# K-Map Simplification

No. of **literals** in an expression = Total No. of variables - $\log_2$ (No. of cells in group)

| 3-Variables | | |
|---|---|---|
| No. Cells | Literals | e.g. |
| 1 | 3 | xyz |
| 2 | 2 | xy |
| 4 | 1 | x |
| 8 | Zero | F=1 |

| 4-Variables | | |
|---|---|---|
| No. Cells | Literals | e.g. |
| 1 | 4 | wxyz |
| 2 | 3 | xyz |
| 4 | 2 | xy |
| 8 | 1 | x |
| 16 | Zero | F=1 |

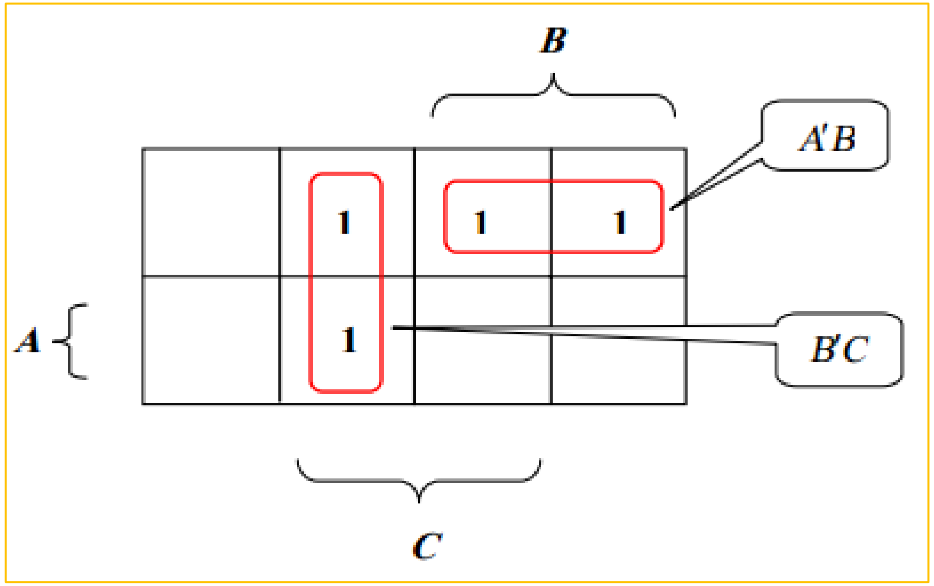
**More Examples:** $F(W, X, Y, Z) = \Sigma m(3,4,5,7,13,14,15)$



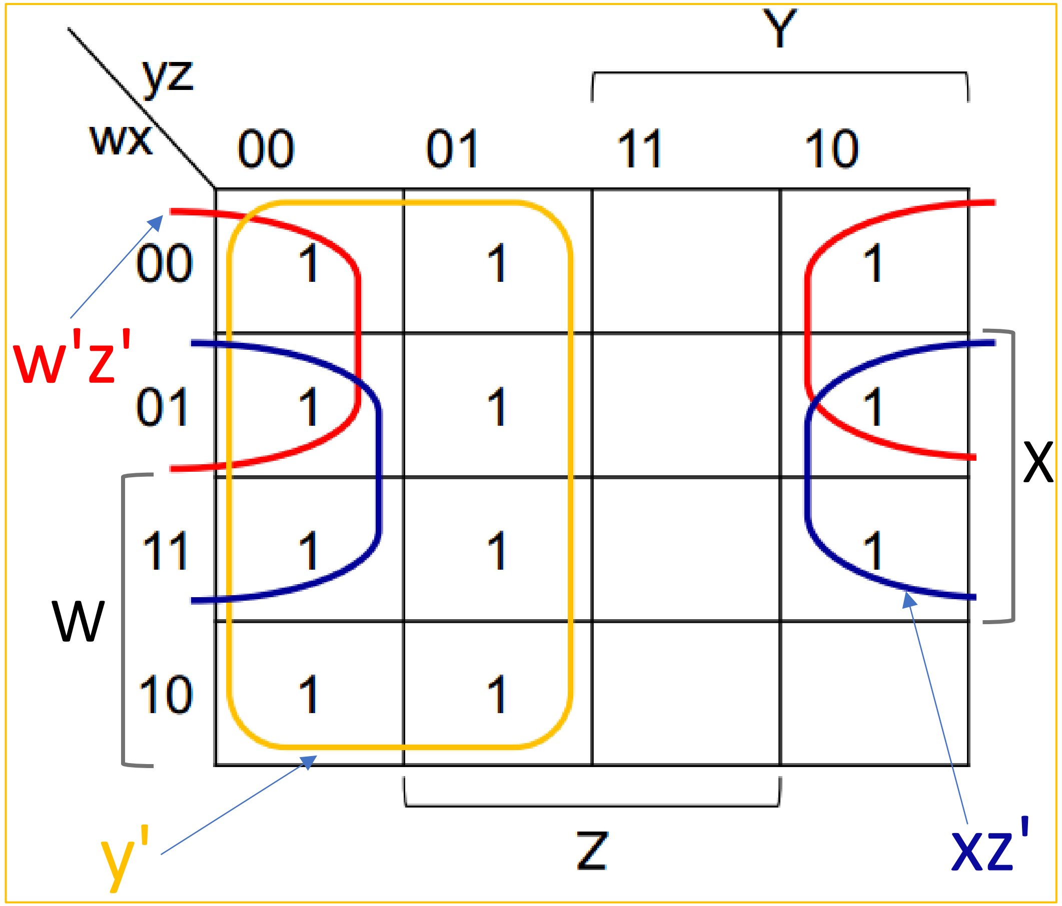
$$F(W, X, Y, Z) = ZX + \overline{W}YZ + WXY + \overline{W}X\overline{Y}$$

# K-Map Simplification

## More Examples:



$$F = A'B + B'C$$



$$F = y' + w'z' + xz'$$

# *K-Map Minimization - Recap*

- In choosing **adjacent** squares in a map, we must **ensure** that:
  - **All** minterms of the function are **covered** when we combine squares
  - The number of **terms** in the expression is **minimized**
  - **No** **redundant** terms

- Sometimes there might be **two or more expressions** that **satisfy** the simplification criteria

# *K-Map Minimization - Implicants*

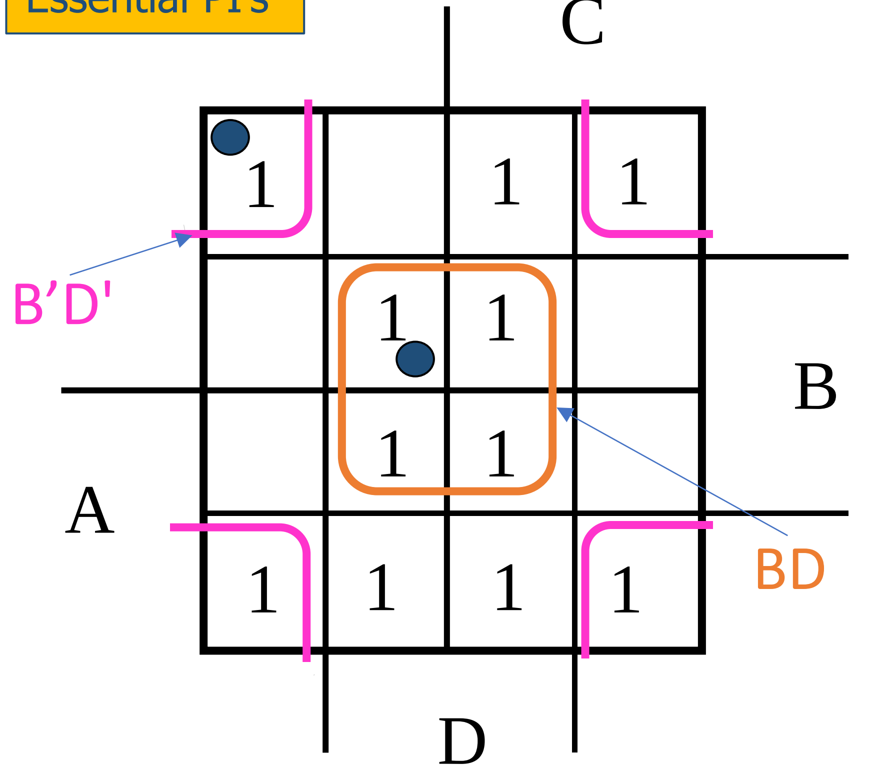
- The procedure for **combining** squares in the map may be made more **systematic** if we understand the meaning of the following terms.
  - **Implicant**: is a product term of a function obtained by **valid grouping** of adjacent squares (minterms or 1’s)
  - **Prime Implicant (PI):** is a product term obtained by combining the **maximum possible** number of adjacent squares
    - **Examples:**
      - ✓ **1** that is not adjacent to any other 1’s.
      - ✓ **Two** adjacent 1’s that are not in a group of four adjacent 1’s.
      - ✓ **Four** adjacent 1’s that are not in a group of eight adjacent 1’s
  - **Essential Prime Implicant (EPI):** If a **minterm** is **covered** by **only one** prime implicant, that prime implicant is said to be **essential prime implicant**

The **simplified** expression is obtained from the **logical sum** of **all the essential prime implicants**, plus **other prime implicants that may be needed** to **cover** any remaining minterms **not covered** by the essential prime implicants.

# K-Map Minimization - Implicants

**Example:**

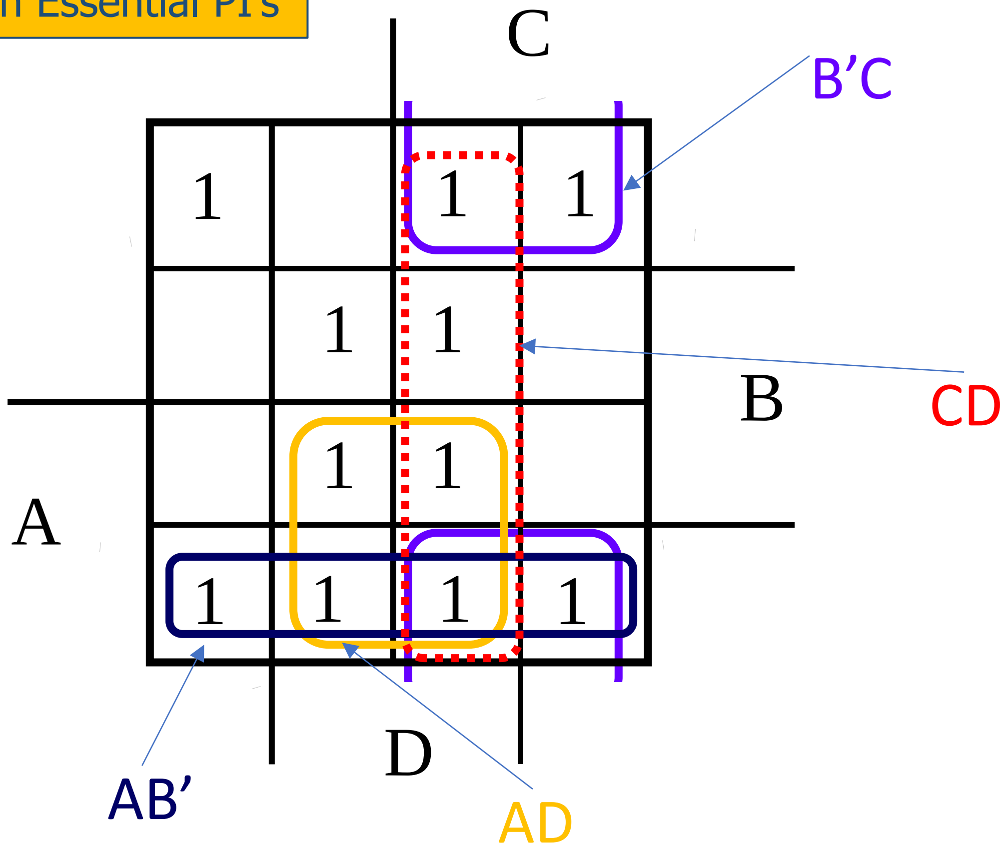
Essential PI’s



● The Minterm is **only** covered by this **PI.**

Essential prime implicants $BD$ and $B'D'$

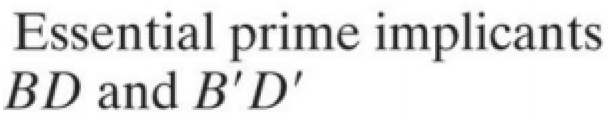
Non Essential PI’s



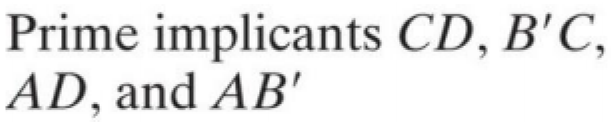
Prime implicants $CD$, $B'C$, $AD$, and $AB'$

# K-Map Minimization - Implicants

**Example Continue:**

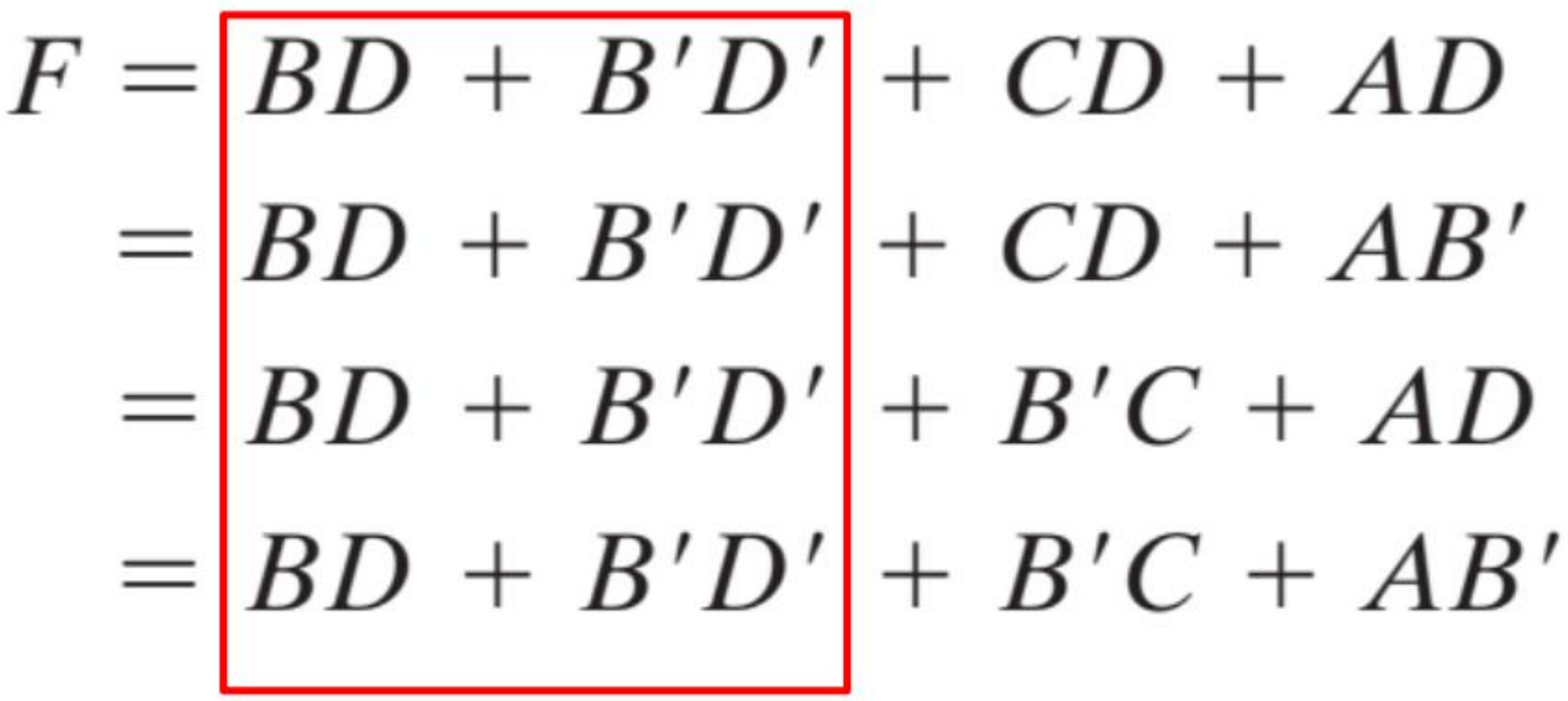


Essential prime implicants $BD$ and $B'D'$



Prime implicants $CD$, $B'C$, $AD$, and $AB'$

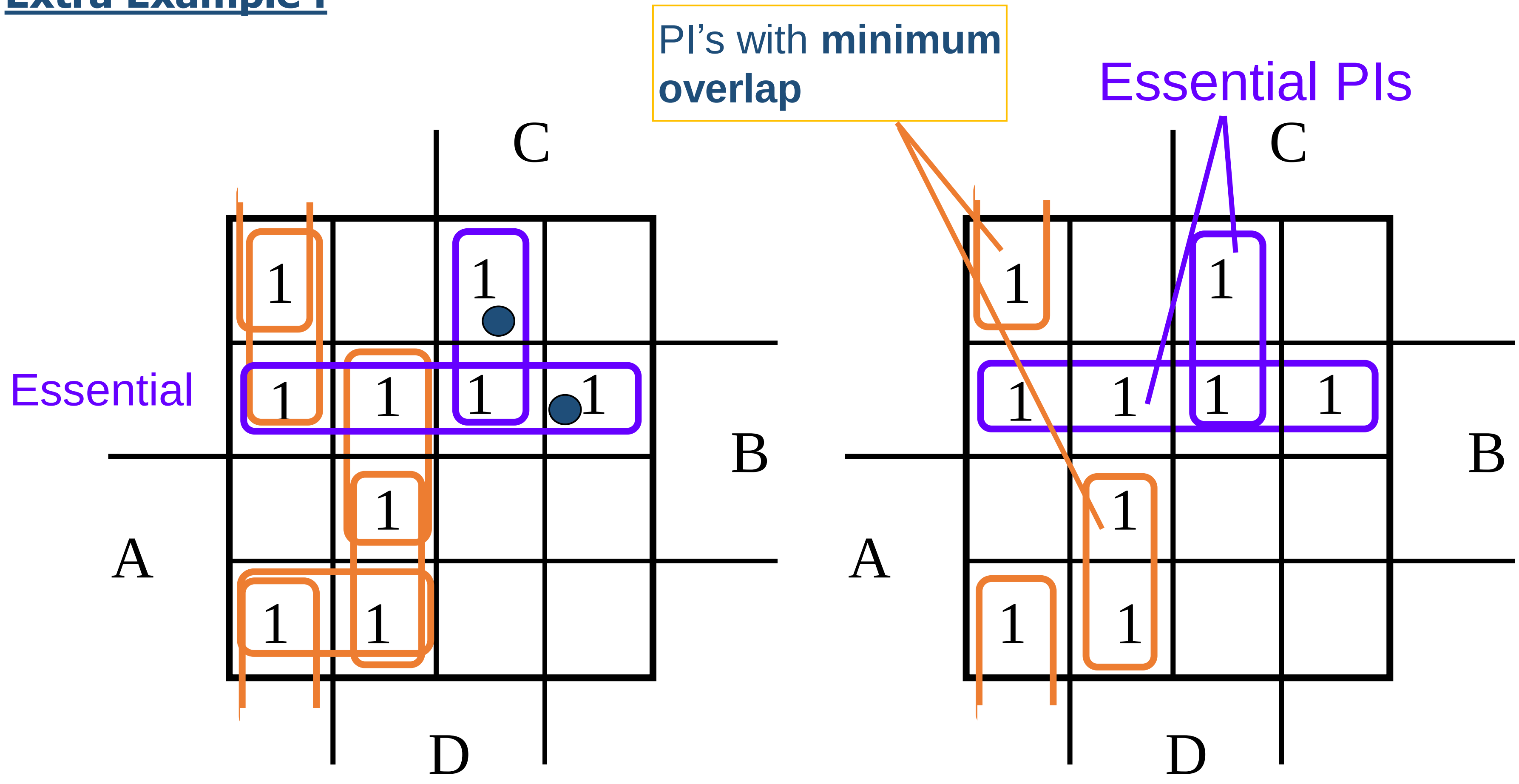
The **simplified** expression is obtained from the **logical sum** of the **two essential** prime implicants and **any** two prime implicants that **cover** the remaining minterms (m3, m9, m11)



$$\begin{aligned} F &= BD + B'D' + CD + AD \\ &= BD + B'D' + CD + AB' \\ &= BD + B'D' + B'C + AD \\ &= BD + B'D' + B'C + AB' \end{aligned}$$

ENCS
2340

**Extra Example :**


PI’s with **minimum overlap**

Essential PIs

Essential

Avoid **Unnecessary overlap** amongst additional **selected PI’s**

# *K-Map Minimization - Don't-Care Conditions*

- **Functions** that have **unspecified** outputs for some input combinations are called incompletely specified functions
  - Such situation arises in which some input variable combinations are not allowed OR might **never occur**
  - For example, in **BCD** there are six **invalid combinations**.
- Since these **unspecified**/**unallowed** terms will **never** occur, they can be treated as **don't-care** terms with respect to their effect on the outputs
- For these "**don't-care**" terms either a **1 or a 0 may** be assigned to the output
  - it really **does not matter** since they will **never occur**
- For the "**don't-care**" input combinations, an **X** is placed in the corresponding square (minterm)
- When **grouping 1's**, the **X's** can be **treated as 1's** to form larger groups → **simpler expression**
  - To get the simplified expression, we **must include all 1's** in the map, but we **may or may not include** any of the **X's**

# K-Map Minimization - Don't-Care Conditions

**Example:**

| $A$ | $B$ | $C$ | $D$ | $F$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | X |
| 0 | 0 | 1 | 0 | X |
| 0 | 0 | 1 | 1 | X |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | X |
| 0 | 1 | 1 | 1 | X |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | X |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | X |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

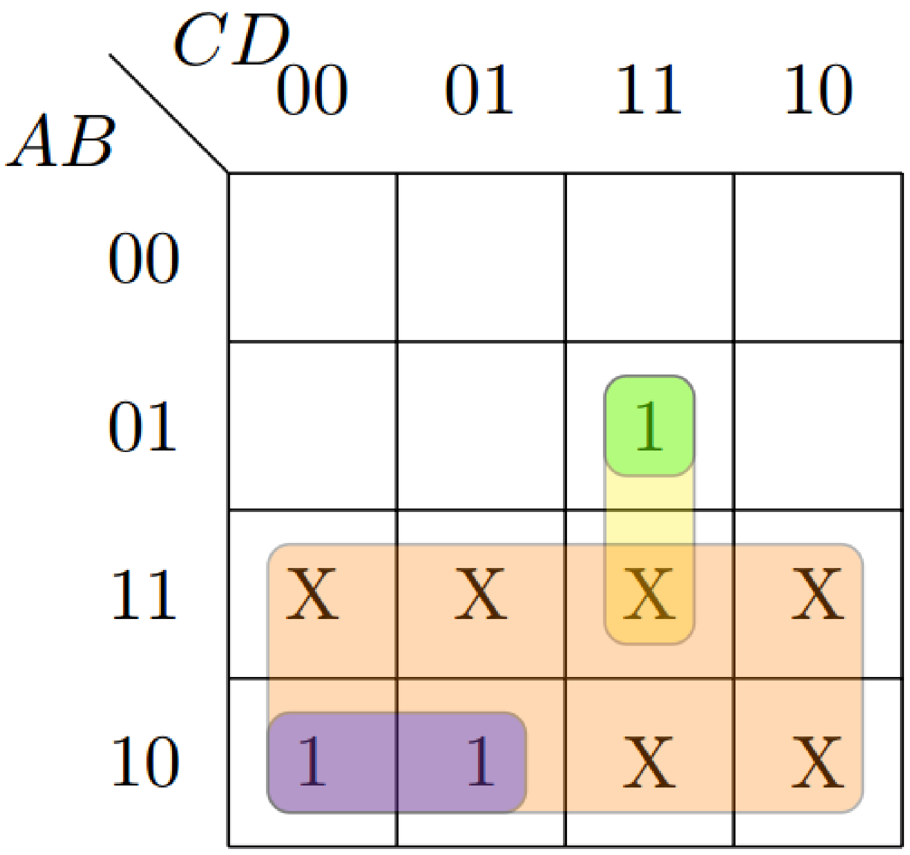
| $AB \backslash CD$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | X | X | X |
| 01 | | 1 | X | X |
| 11 | | 1 | 1 | 1 |
| 10 | | X | X | 1 |

# K-Map Minimization - Don't-Care Conditions

**Example:**

**BCD Code : > 6 → F=1**

| $A$ | $B$ | $C$ | $D$ | $F$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | **X** |
| 1 | 0 | 1 | 1 | **X** |
| 1 | 1 | 0 | 0 | **X** |
| 1 | 1 | 0 | 1 | **X** |
| 1 | 1 | 1 | 0 | **X** |
| 1 | 1 | 1 | 1 | **X** |

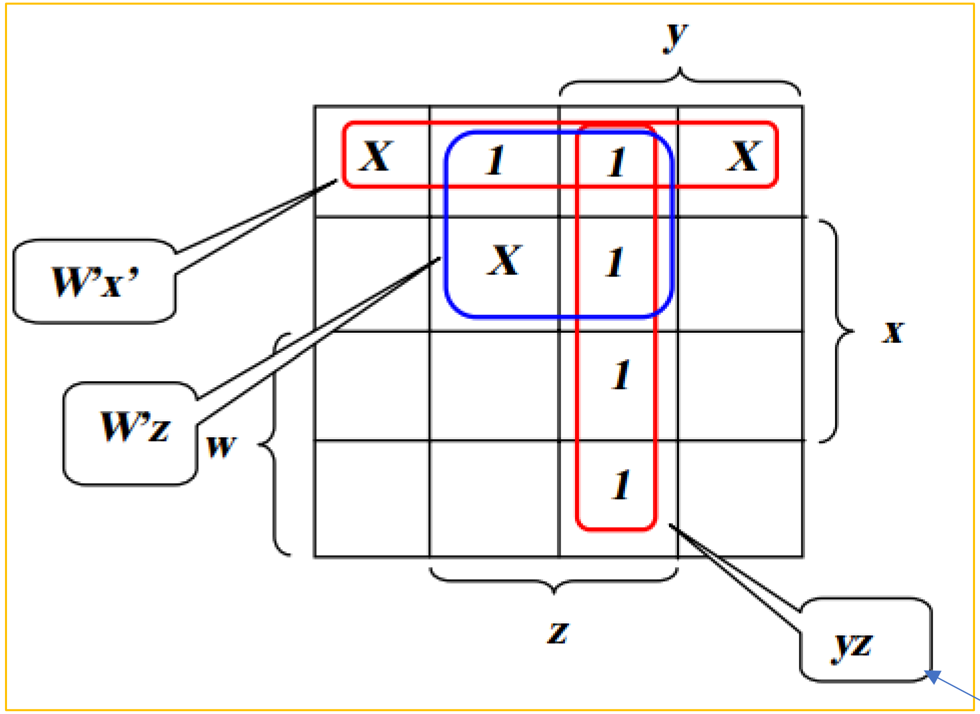


| $AB$ \ $CD$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | | | 1 | |
| 11 | X | X | X | X |
| 10 | 1 | 1 | X | X |

**F = A + BCD**

**Extra Example:**

Simplify the function $F(w,x,y,z)=\Sigma(1,3,7,11,15)$, which has the don't care conditions $d(w,x,y,z)=\Sigma(0,2,5)$



EPI

OR

F = **yz** + w′x′
F = **yz** + w′z

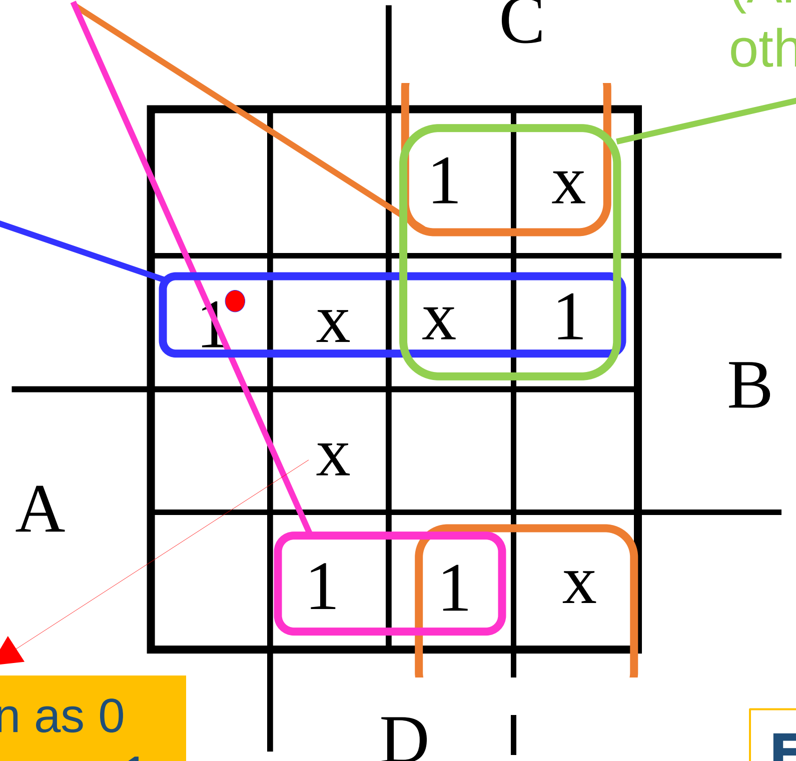
# K-Map Minimization - Don't-Care Conditions

## Extra Example:

Selected PIs with minimum overlap

Non-Selected PI (Already Covers By others)

Essential PI



|   |   |   |   |
|---|---|---|---|
|   |   | 1 | x |
| 1 | x | x | 1 |
|   | x |   |   |
|   | 1 | 1 | x |

This x is taken as 0
All others taken as 1

$F = A'B + B'C + AB'D$

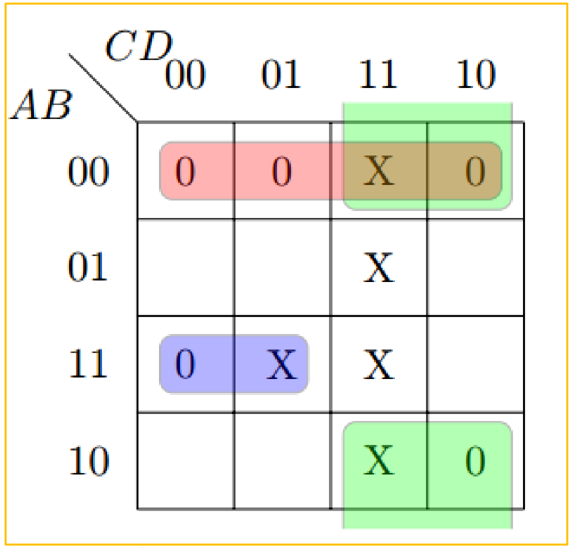
# K-Map Minimization – Product of Sums (POS)

- So far, We have been **combining minterms** (**‘1’ squares**), to get **F** as a **Sum of Products** (SOP) out of the K-Map simplification process

- If we combine the **remaining** minterms (**‘0’ squares**), we get the **compliment** of **F** => (**F’**)
  - Using **DeMorgan**’ s Theorem we can get **F** as a **Product-of-Sums** (POS) by complementing **F’** => **(F’)’ = F**

***Product of Sums (POS) Procedure***

1) Combine the 0’s into groups
2) Form a sum-of-products (SOP) expression from these groups of 0’s → **F’**
3) Take its **complement** using DeMorgan’s theorem to get **F** as **product-of-sum**



| $AB \backslash CD$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | X | 0 |
| 01 | | | X | |
| 11 | 0 | X | X | |
| 10 | | | X | 0 |

The following groups of 0’s are formed:

- Red group: $A'B'$
- Blue group: $ABC'$
- Green group: $B'C$

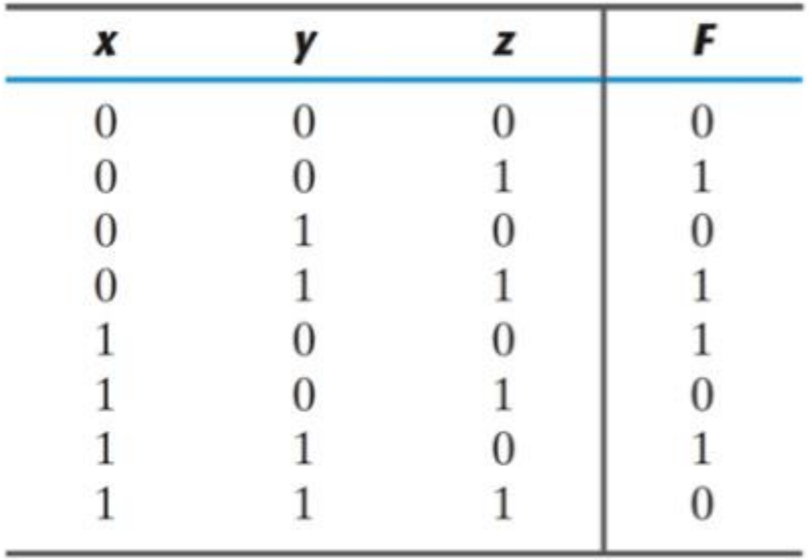
The expression of $F' = A'B' + ABC' + B'C$

Use DeMorgan’s theorem to find its complement in order to write an expression of $F$

$$F = (A + B)(A' + B' + C)(B + C')$$

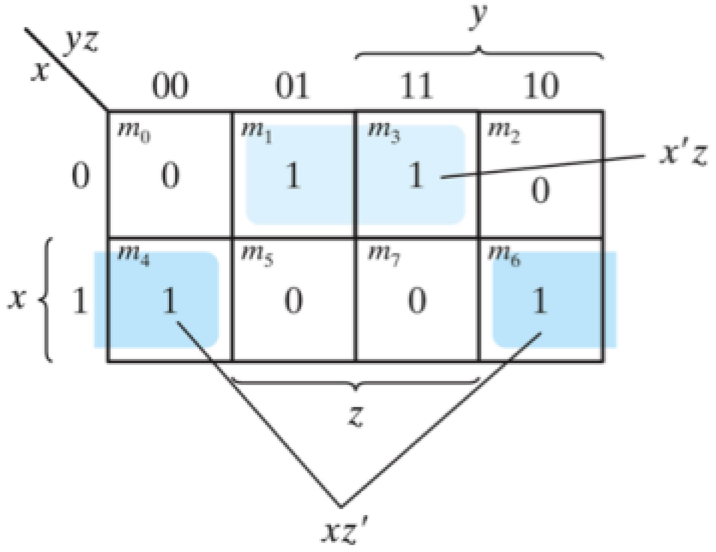
# K-Map Minimization – Product of Sums (POS)

**Example:**



| x | y | z | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

$$F = m_1 + m_3 + m_4 + m_6$$



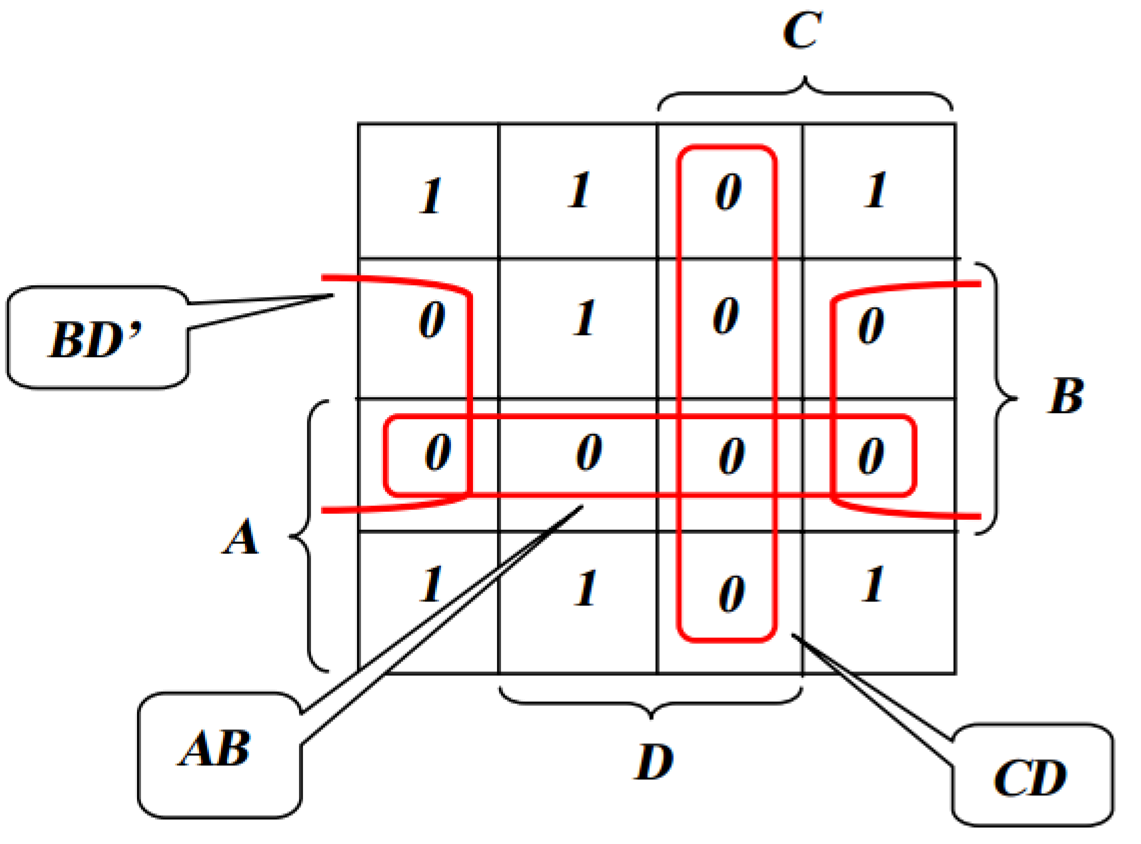
$$F = x'z + xz'$$

$$F' = xz + x'z'$$

$$(F')' = F = (x' + z')(x + z)$$

# K-Map Minimization – Product of Sums (POS)

**Extra Example:** $F(A,B,C,D)=\Sigma(0,1,2,5,8,9,10)$



$$F' = AB + CD + BD'$$

DeMorgan's

$$F = \left(AB + CD + BD'\right)' = (A' + B')(C' + D')(B' + D)$$

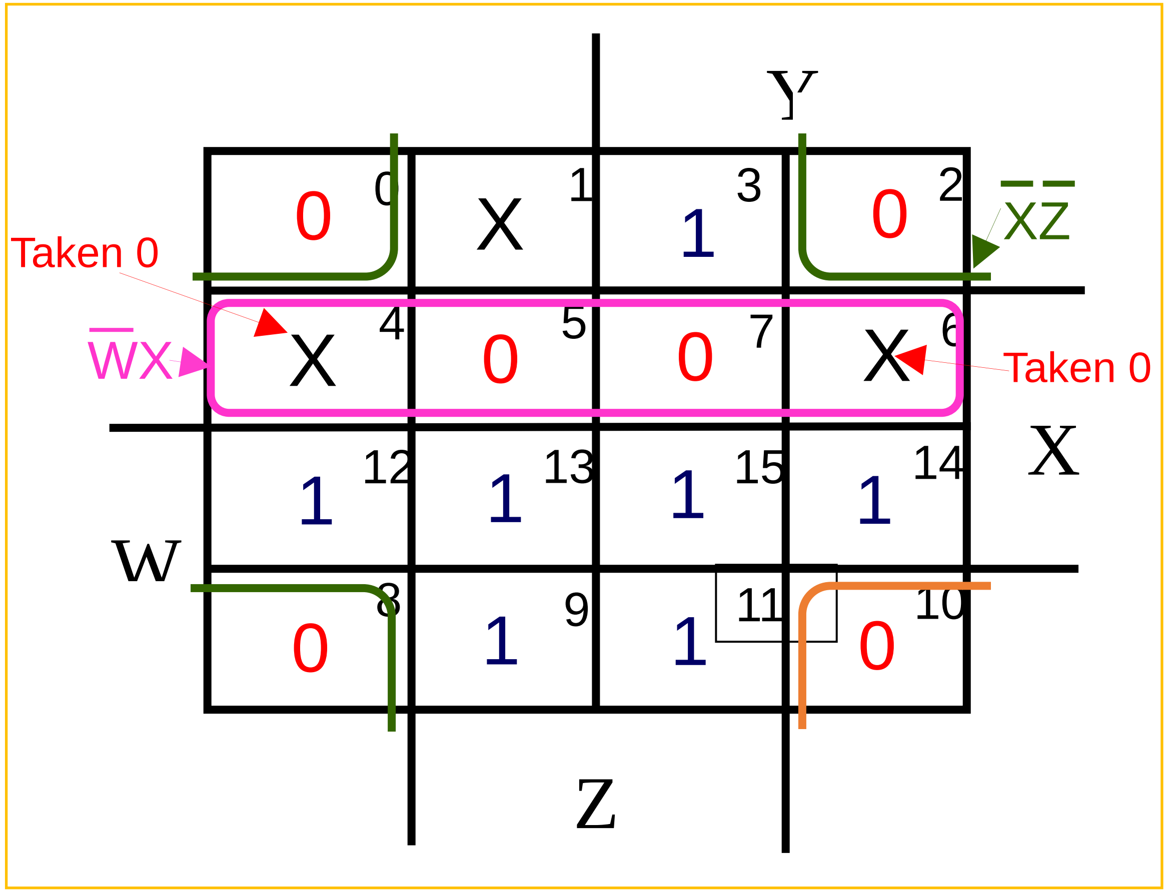
## K-Map Minimization – Product of Sums (POS)

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

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

DeMorgan's

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



| | | | |
|---|---|---|---|
| 0 (0) | X (1) | 1 (3) | 0 (2) |
| X (4) | 0 (5) | 0 (7) | X (6) |
| 1 (12) | 1 (13) | 1 (15) | 1 (14) |
| 0 (8) | 1 (9) | 1 (11) | 0 (10) |

# What if the function was **given** as **product of maxterms**?

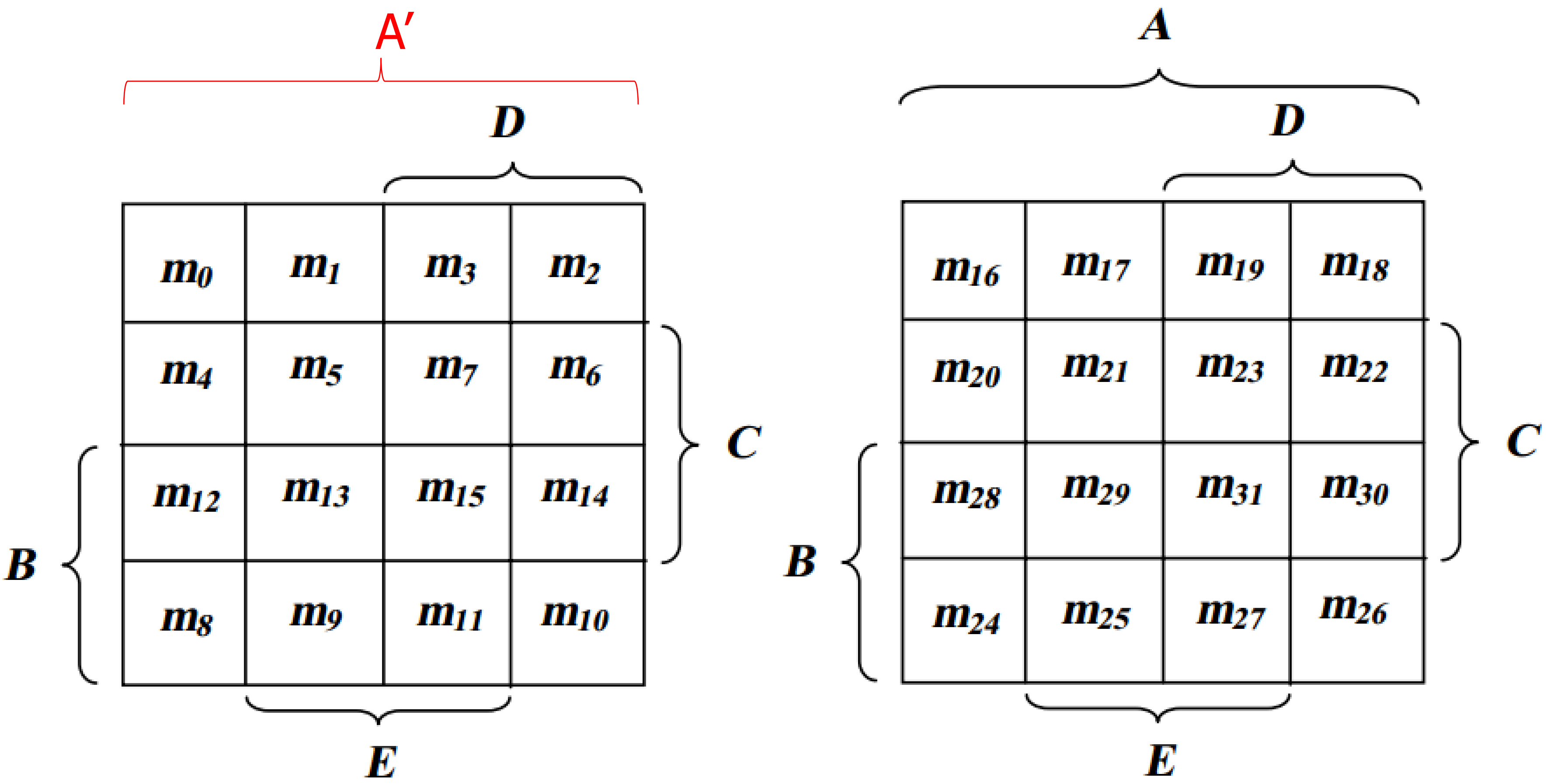
1) **Complement** F → **F’** in **Sum-of Product** From (SOP)
2) Mark **F’ minterms**’ squares with **0’s** and the **remaining** squares with **1’s**.

# K-Map Minimization – 5 Variables

ENCS
2340

The map of **five** variables → **two** **four** variable maps.

**F(A,B,C,D,E)**

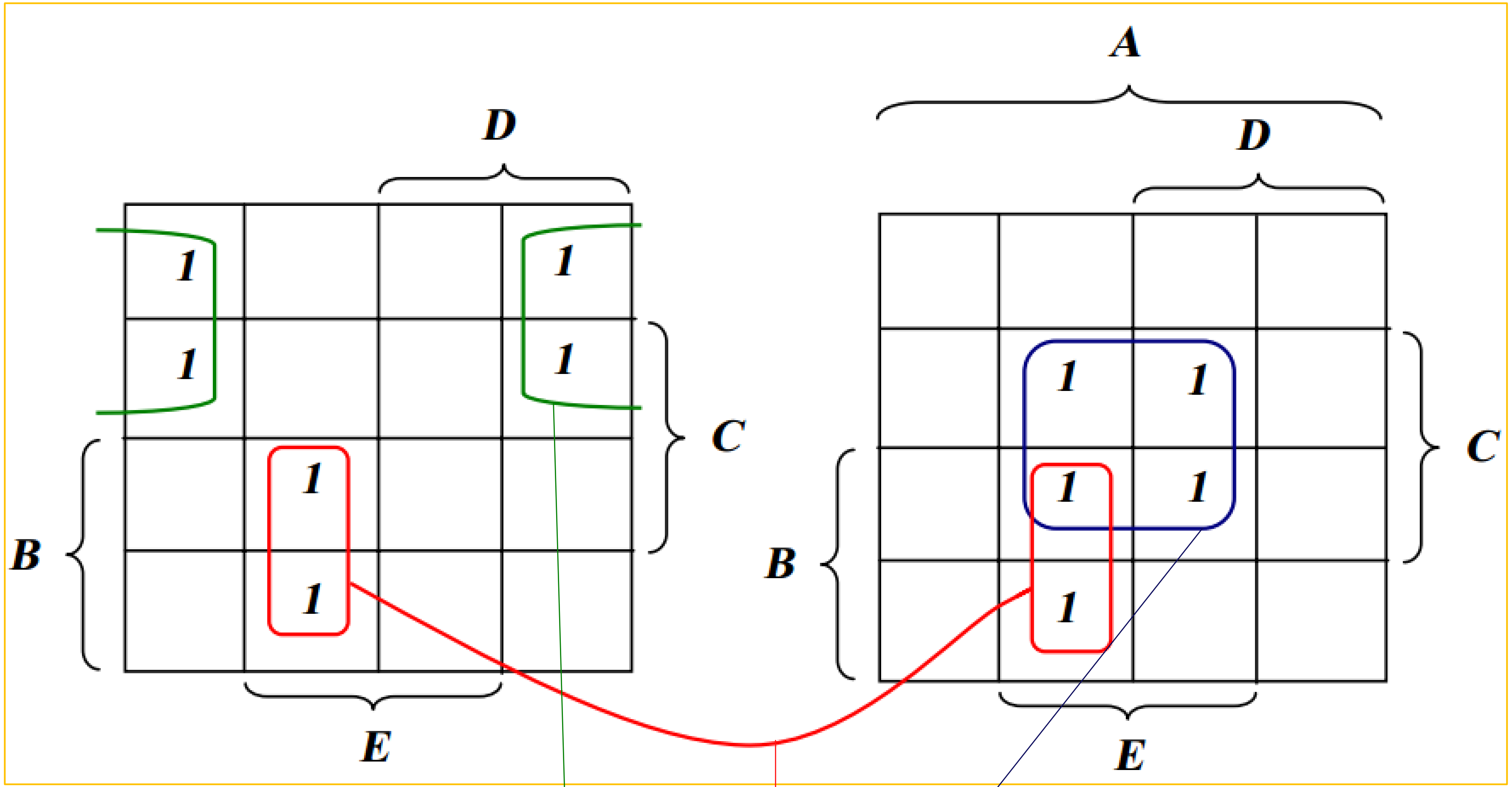


| | | | |
|---|---|---|---|
| $m_0$ | $m_1$ | $m_3$ | $m_2$ |
| $m_4$ | $m_5$ | $m_7$ | $m_6$ |
| $m_{12}$ | $m_{13}$ | $m_{15}$ | $m_{14}$ |
| $m_8$ | $m_9$ | $m_{11}$ | $m_{10}$ |

| | | | |
|---|---|---|---|
| $m_{16}$ | $m_{17}$ | $m_{19}$ | $m_{18}$ |
| $m_{20}$ | $m_{21}$ | $m_{23}$ | $m_{22}$ |
| $m_{28}$ | $m_{29}$ | $m_{31}$ | $m_{30}$ |
| $m_{24}$ | $m_{25}$ | $m_{27}$ | $m_{26}$ |

# K-Map Minimization – 5 Variables

**Example:** $F(A,B,C,D,E)=\sum(0,2,4,6,9,13,21,23,25,29,31)$

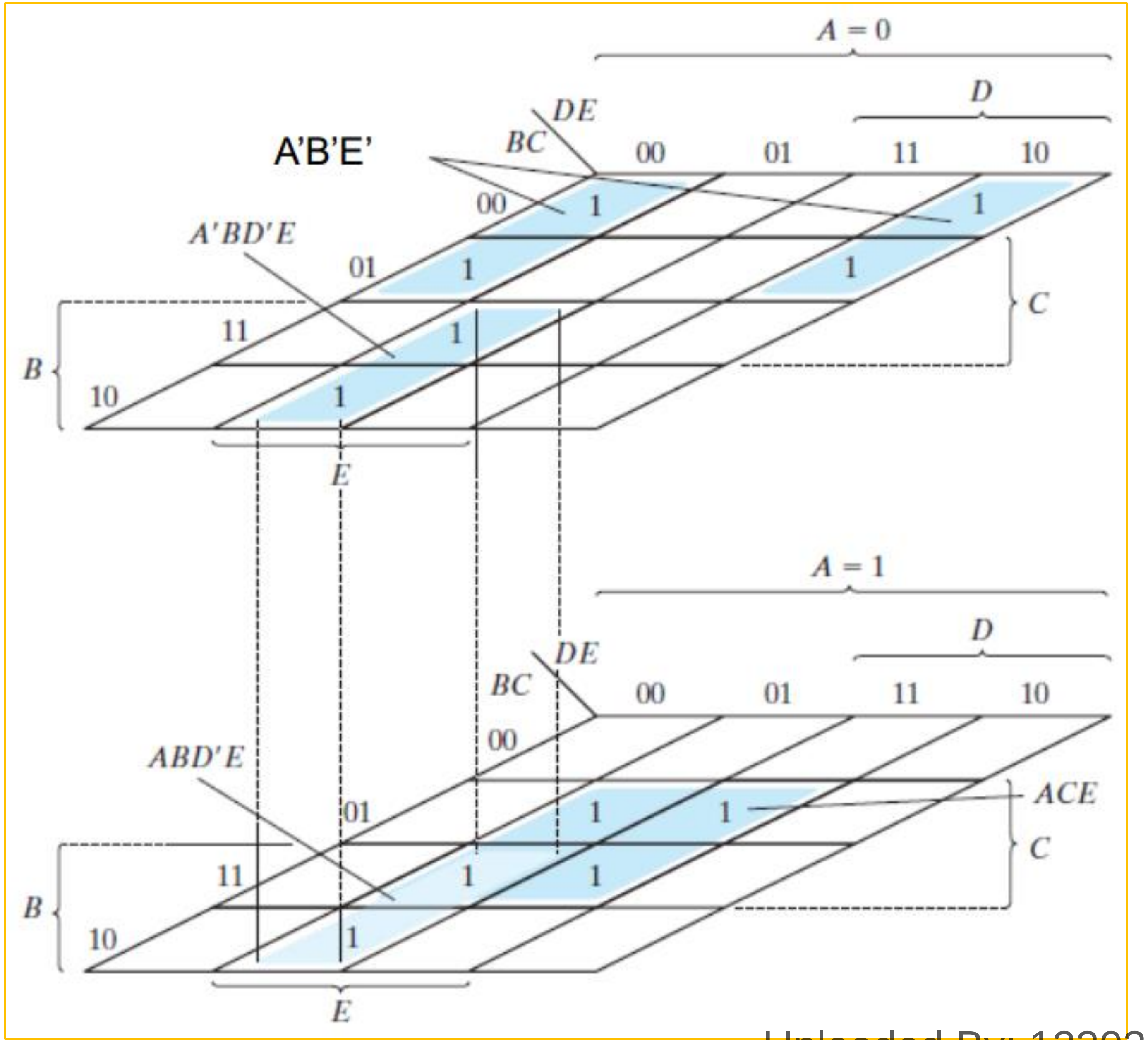


| 5-Variables | | |
|---|---|---|
| No. Cells | Literals | e.g. |
| 1 | 5 | ABCDE |
| 2 | 4 | BCDE |
| 4 | 3 | CDE |
| 8 | 2 | DE |
| 16 | 1 | E |
| 32 | Zero | F=1 |

$F(A,B,C,D,E)=A'B'E'+BD'E+ACE$

# K-Map Minimization – 5 Variables

**Example:**



# NAND & NOR IMPLEMENTATION

- Digital circuits are frequently constructed with **NAND** or **NOR** gates rather than with AND or OR gates.

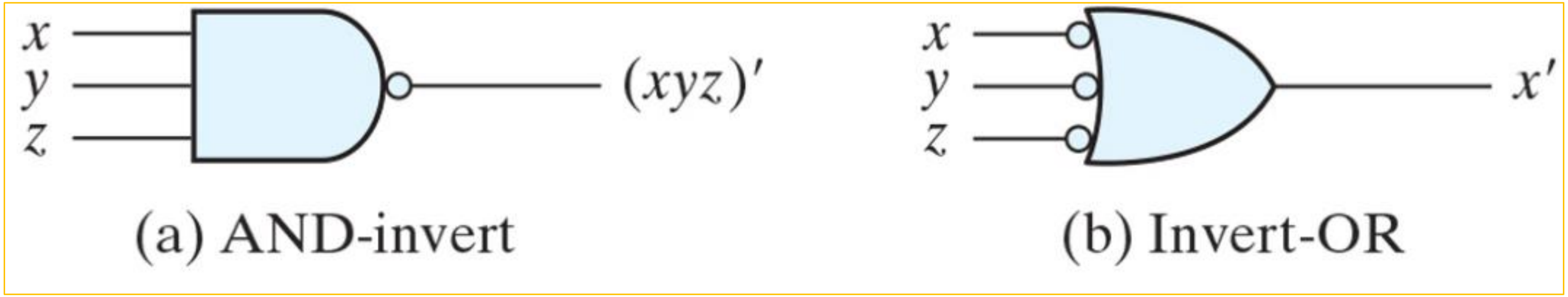
- **NAND** and **NOR** gates are **easier to fabricate** with electronic components and are **the basic gates** used in all Integrated Circuit (IC) digital logic families.

- NAND and NOR gates are **universal gates**
  - **Any** digital system can be **implemented** with **them**

- To implement a function with **NAND** it need to be in **Sum-of Products** (SOP) form
- To implement a function with **NOR** it need to be in **Product-of Sums** (POS) form

Rules have been developed to **convert** any logic circuit to its **equivalent** form in just **NAND gates or NOR gates**

## NAND Graphic Symbols



(a) AND-invert

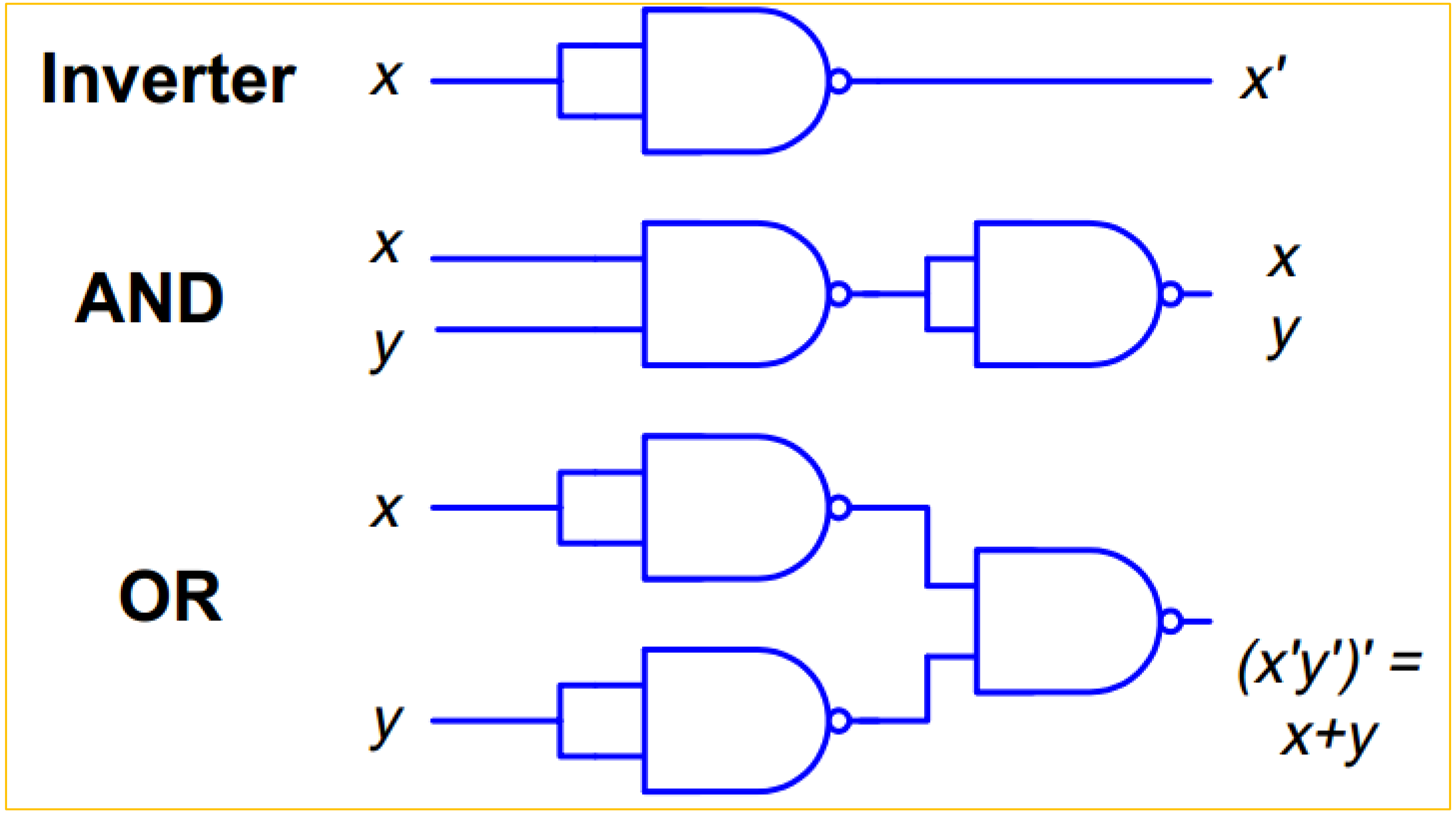
(b) Invert-OR

- **Both** Symbols are **same** because of DeMorgan’s theorem
- Circuits could be **drawn** using **any** of these symbols
- When a circuit uses **both** symbols it is said to follow **mixed notation**

# NAND IMPLEMENTATION

- A **set** of gates is **functionally complete** if **any** Boolean expression can be **realized** with this set of gates
- **AND, OR, and inverter** is functionally complete
- **Any** set of gates which can **implement** AND, OR and inverter is also functionally complete

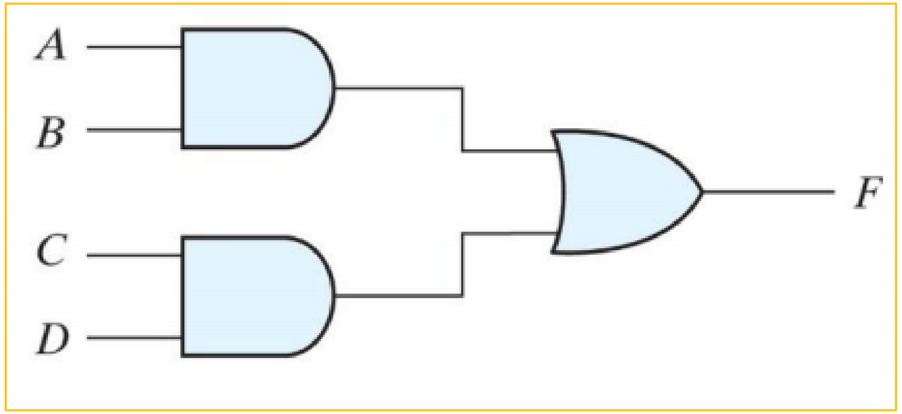
**All** gates can be **represented** using Only **NAND** gates → NAND is **functionally complete**



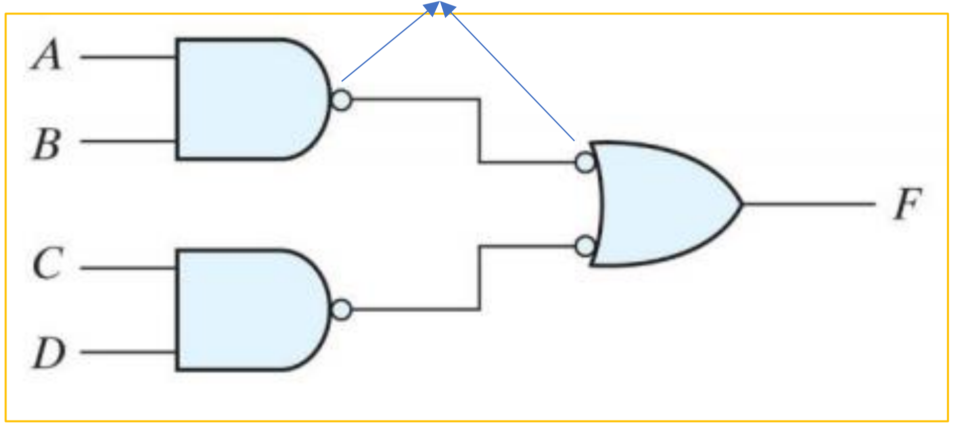
Logic Operations with NAND gates

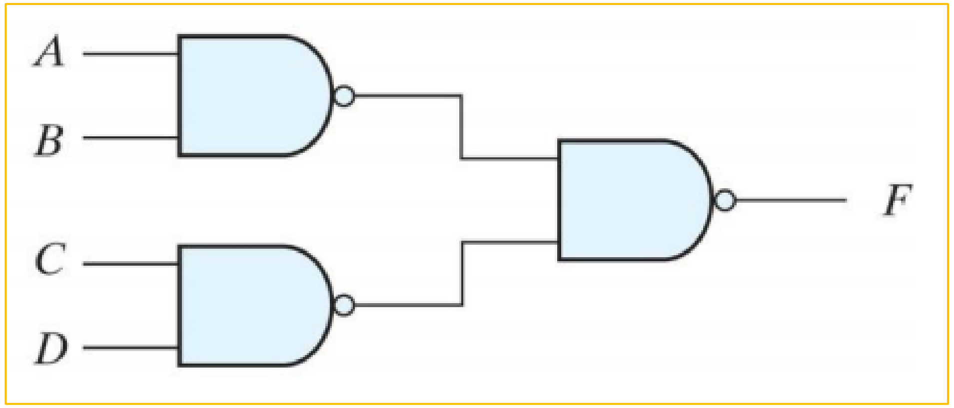
# *NAND IMPLEMENTATION – 2-Level*

**Example:** Implement $F = AB + CD$ , using NAND gates



**Inverts** on **same** line **cancel** each others





**Use the AND-Invert Form**

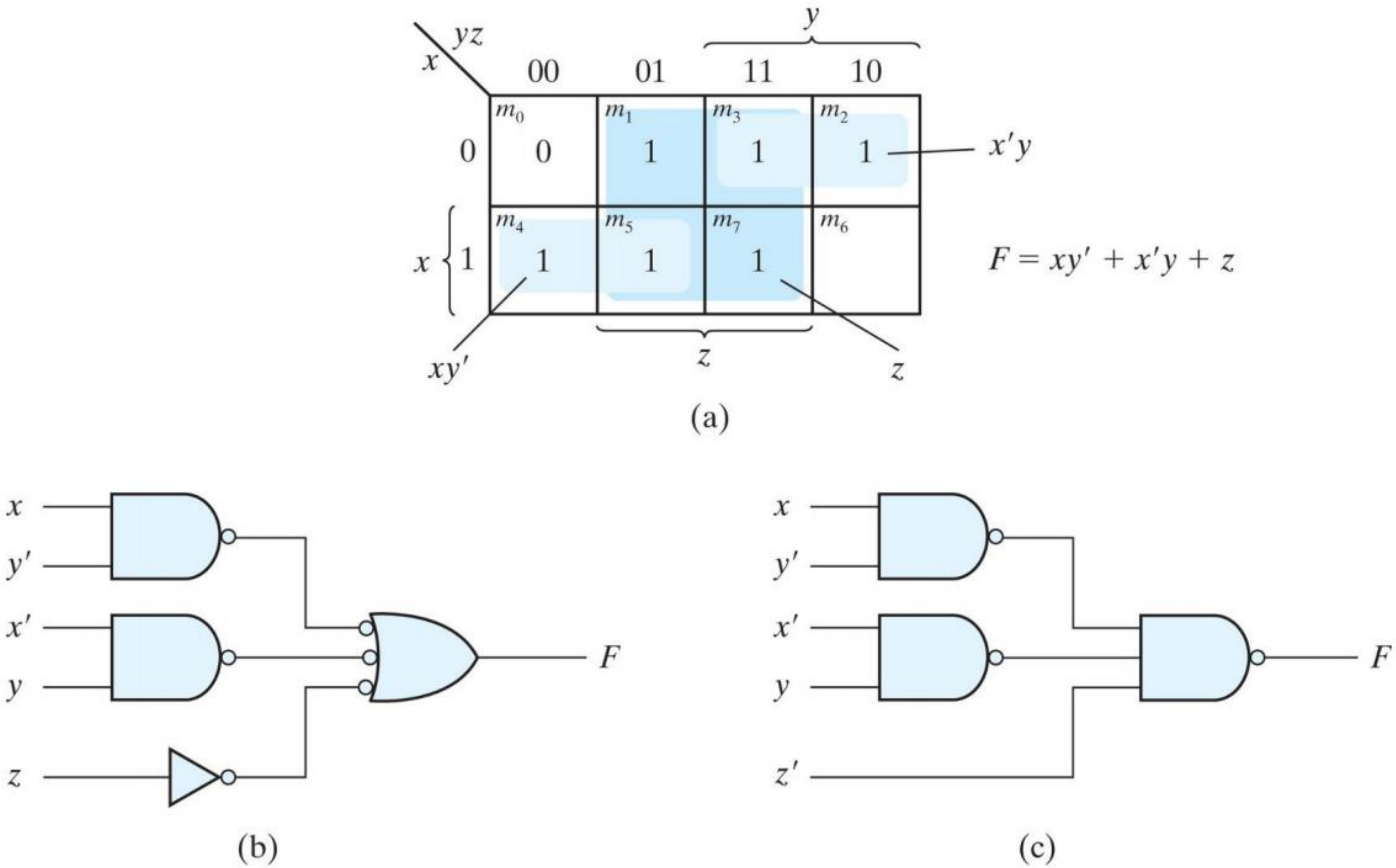
# *NAND IMPLEMENTATION – 2-Level*

- **Two-level Implementation using NAND Gates**
  1) **Simplify** the function and express it in **sum-of-products** form.
  2) Draw a **NAND gate** for each product term of the expression that has **at least two literals**. The inputs to each NAND gate are the literals of the term. This procedure produces a group of **first-level gates**.
  3) Draw a **single gate** using the AND-invert or the invert-OR graphic symbol in the **second** level, with **inputs** coming from outputs of **first-level gates**.
  4) A term with a **single literal** requires an inverter in the first level. However, if the single literal is **complemented**, it can be connected **directly** to an input of the **second level NAND gate.**

# *NAND IMPLEMENTATION – 2-Level*

**Example:**

Implement *F* (*x*, *y*, *z*) = (1, 2, 3, 4, 5, 7) with NAND gates

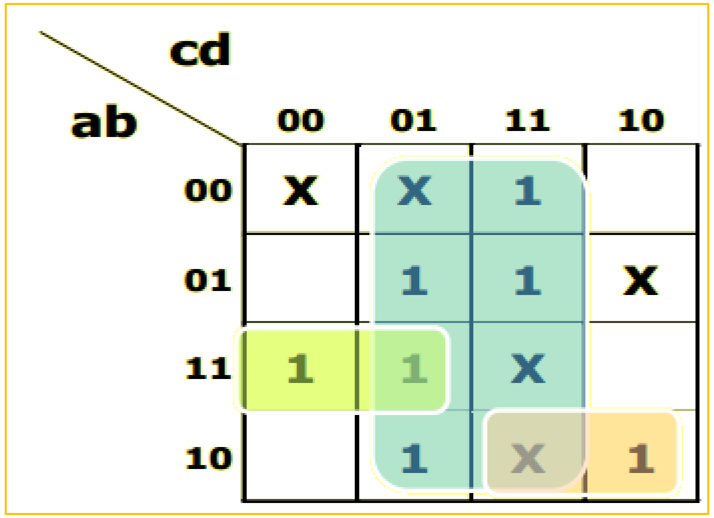


# NAND IMPLEMENTATION – 2-Level

**Example:**

$$F(a, b, c, d) = \sum m(3, 5, 7, 9, 10, 12, 13) + \sum d(0, 1, 6, 11, 15)$$

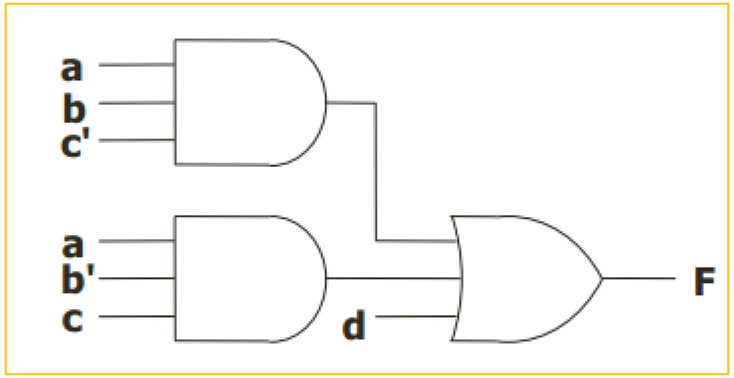
**Find SOP**



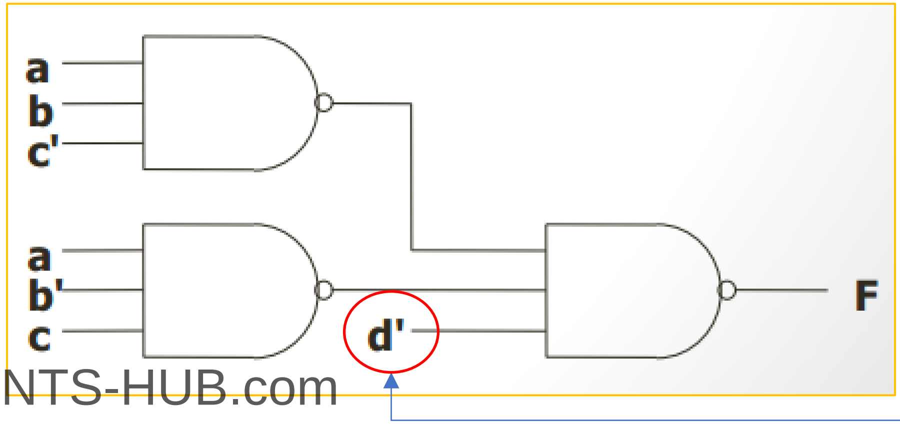
| ab \ cd | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | X | X | 1 | |
| 01 | | 1 | 1 | X |
| 11 | 1 | 1 | X | |
| 10 | | 1 | X | 1 |

$$F = d + abc' + ab'c$$

**Draw Logic Gates**



**Convert to NANDs**



When Converting to NAND, Don't forget to **complement** the **direct** inputs (Single Literals)

# *NAND IMPLEMENTATION – Multilevel*

- The general procedure for converting a **multilevel** AND–OR diagram into an **all-NAND** diagram using mixed notation is as follows:

  1) **Convert** all AND gates to NAND gates with **AND-invert** graphic symbols.

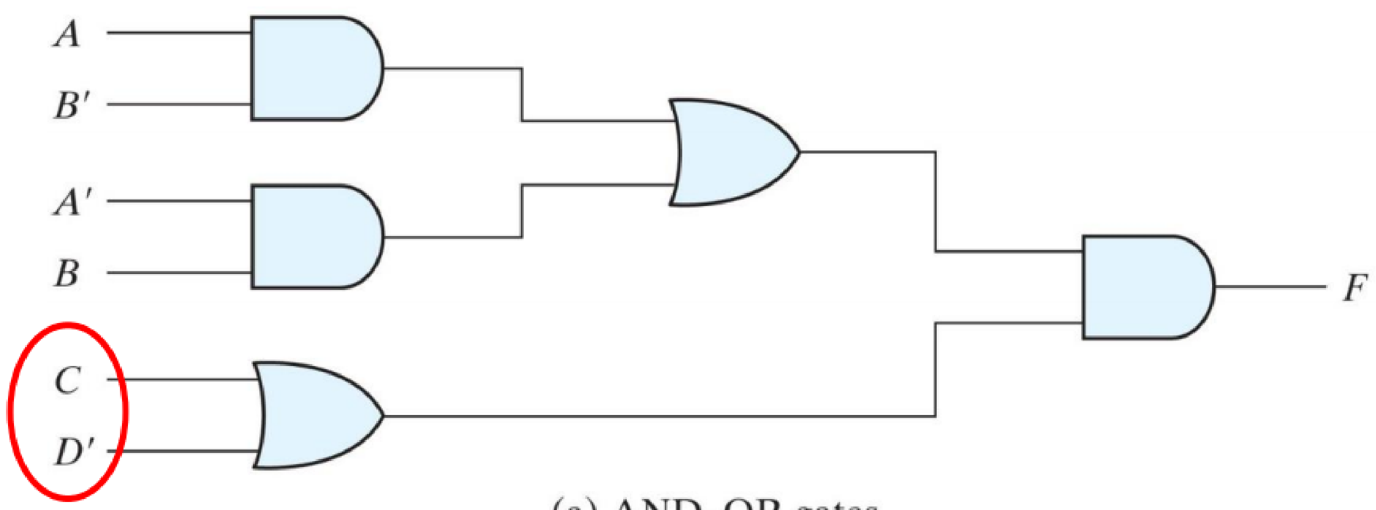
  2) **Convert** all OR gates to NAND gates with **invert-OR** graphic symbols.

  3) **Check** all the bubbles in the diagram. For every **bubble** that is not compensated by another **small circle (bubble)** along the same line, **insert an inverter** (a one-input NAND gate) **or complement the input literal.**

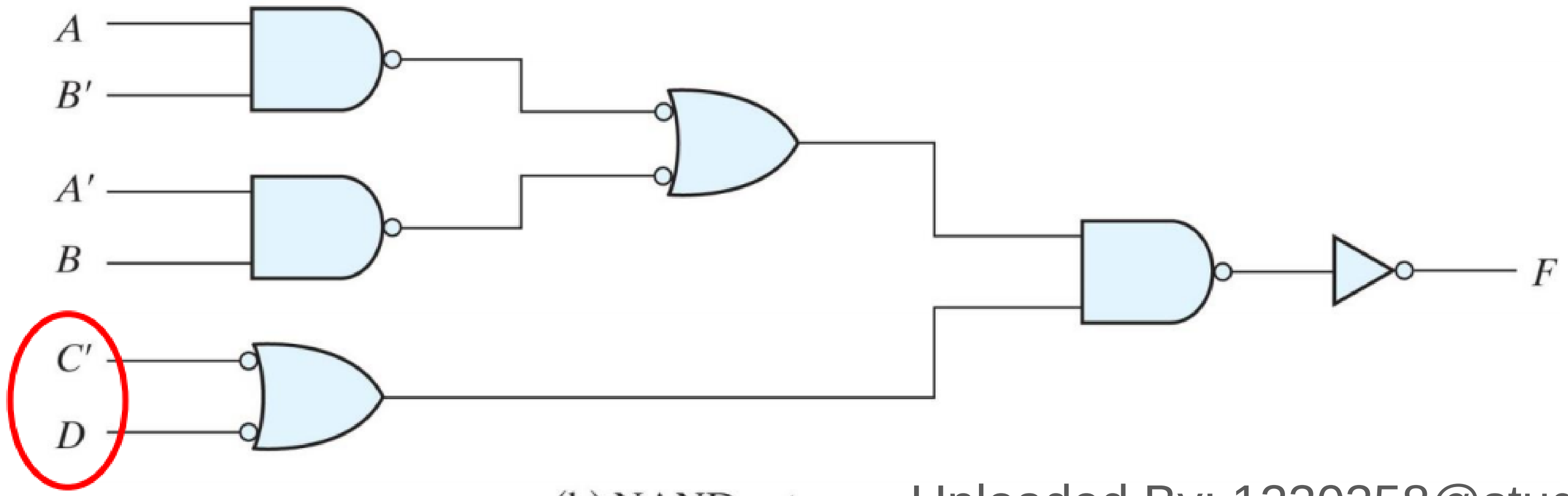
# NAND IMPLEMENTATION – Multilevel

**Example:**

$$F = (AB' + A'B)(C + D')$$



(a) AND–OR gates

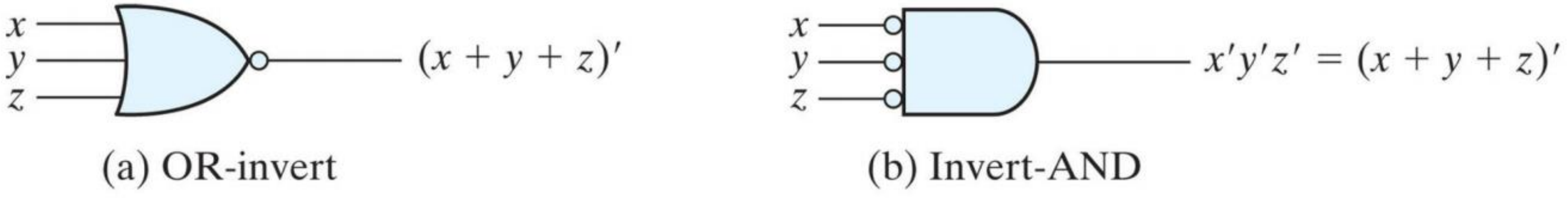


(b) NAND gates

# NOR IMPLEMENTATION

The **NOR** operation is the **dual** of the **NAND** operation. Therefore, all procedures and rules for NOR logic are the **duals** of the corresponding procedures and rules developed for NAND logic

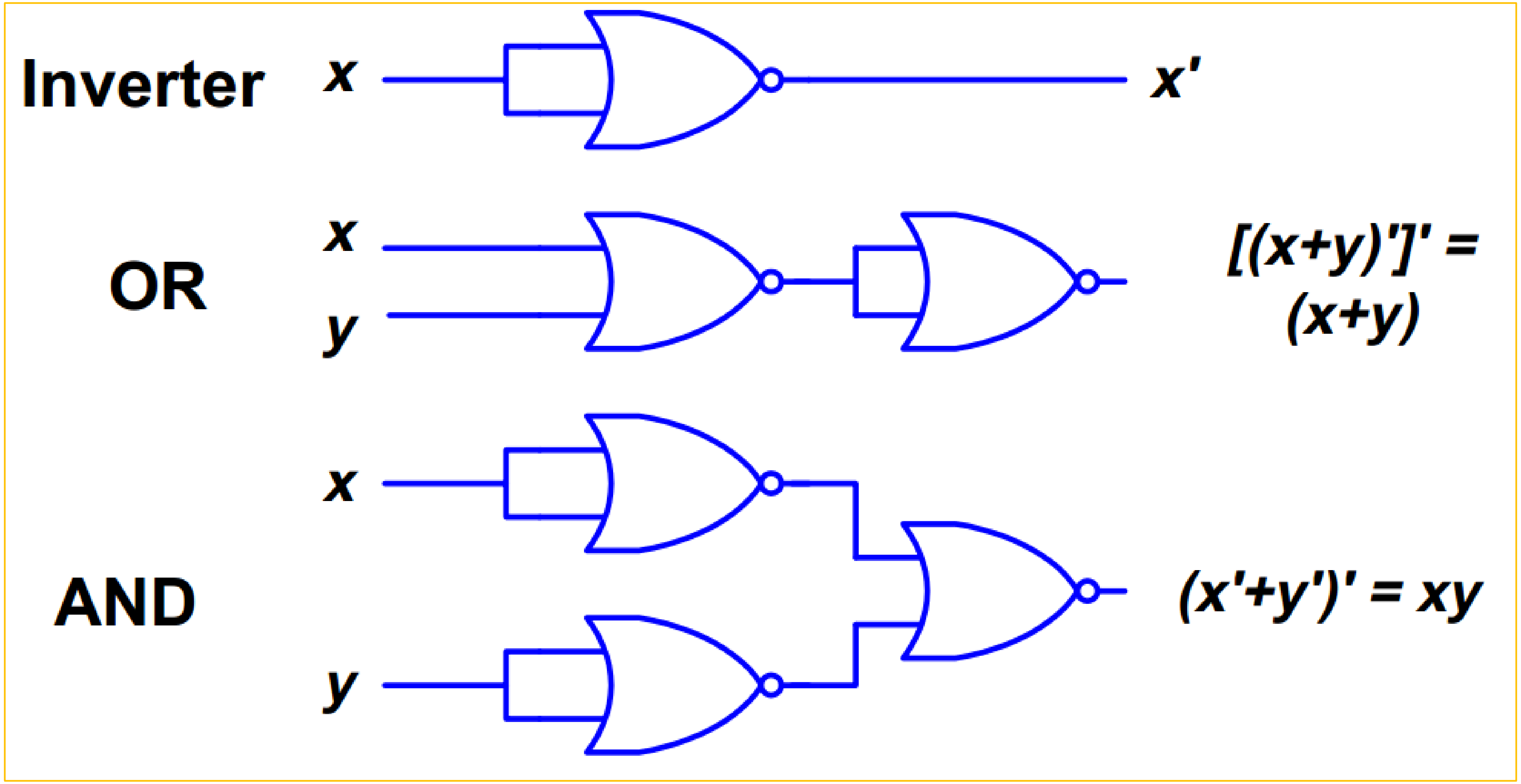
## NOR Graphic Symbols



- **Both** Symbols are **same** because of DeMorgan's theorem
- Circuits could be **drawn** using **any** of these symbols
- When a circuit uses **both** symbols it is said to follow **mixed notation**

# NOR IMPLEMENTATION

**All** gates can be **represented** using Only **NOR** gates → NOR is **functionally complete**



Logic Operations with NOR gates

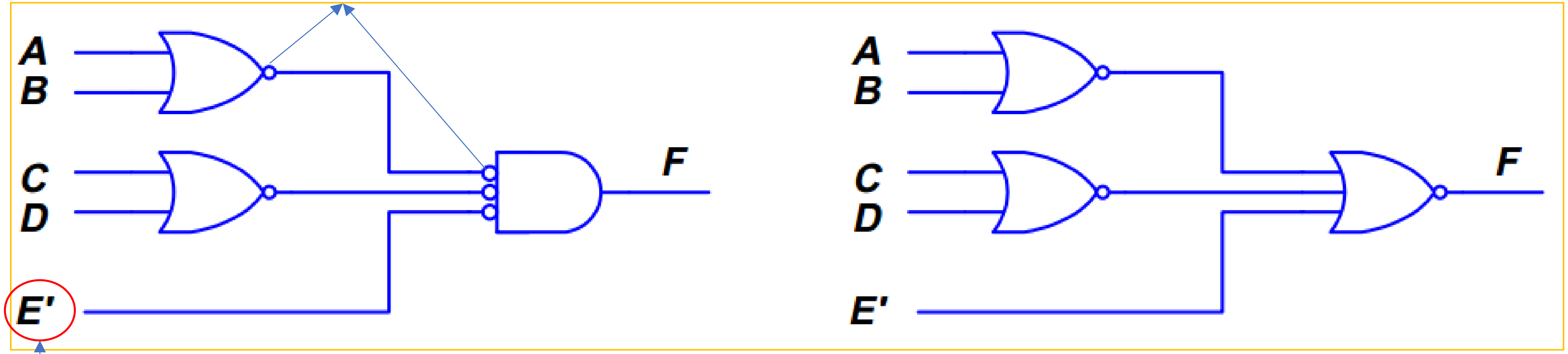
# *NOR IMPLEMENTATION – 2-Level*

**Example:** $F = (A + B)(C + D)E$

**Simplify** the function and express it in **product-of-soms** form

**Inverts** on **same** line **cancel** each others

**Use the OR-Invert Form**

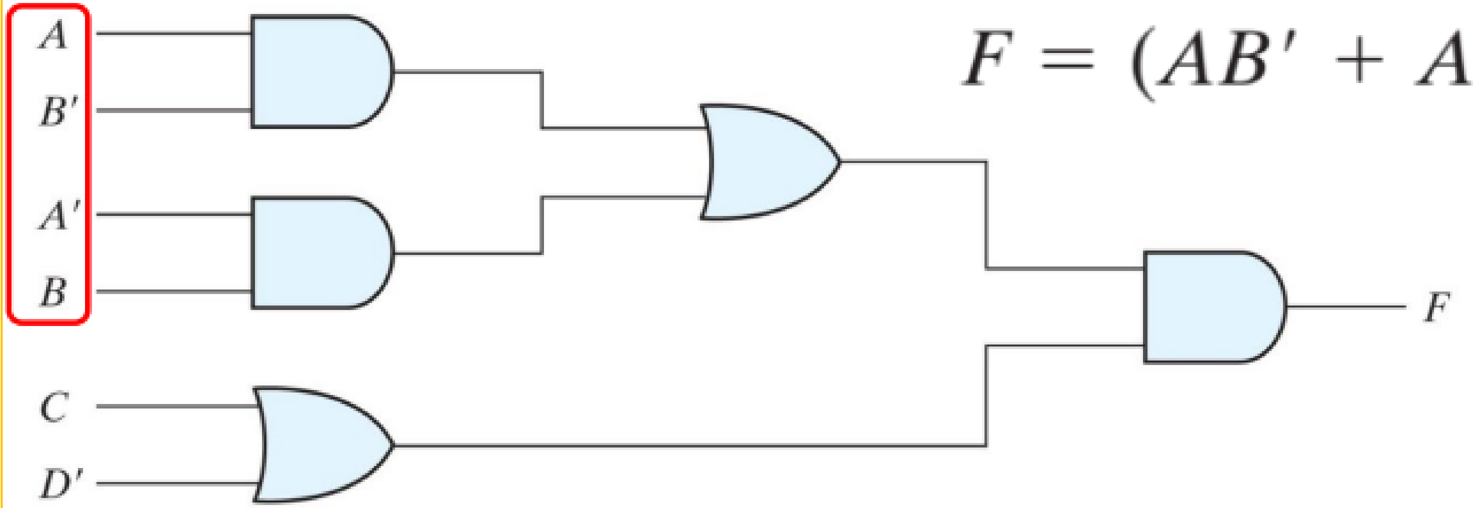


When Converting to NOR, Don't forget to **complement** the **direct** inputs (Single Literals)

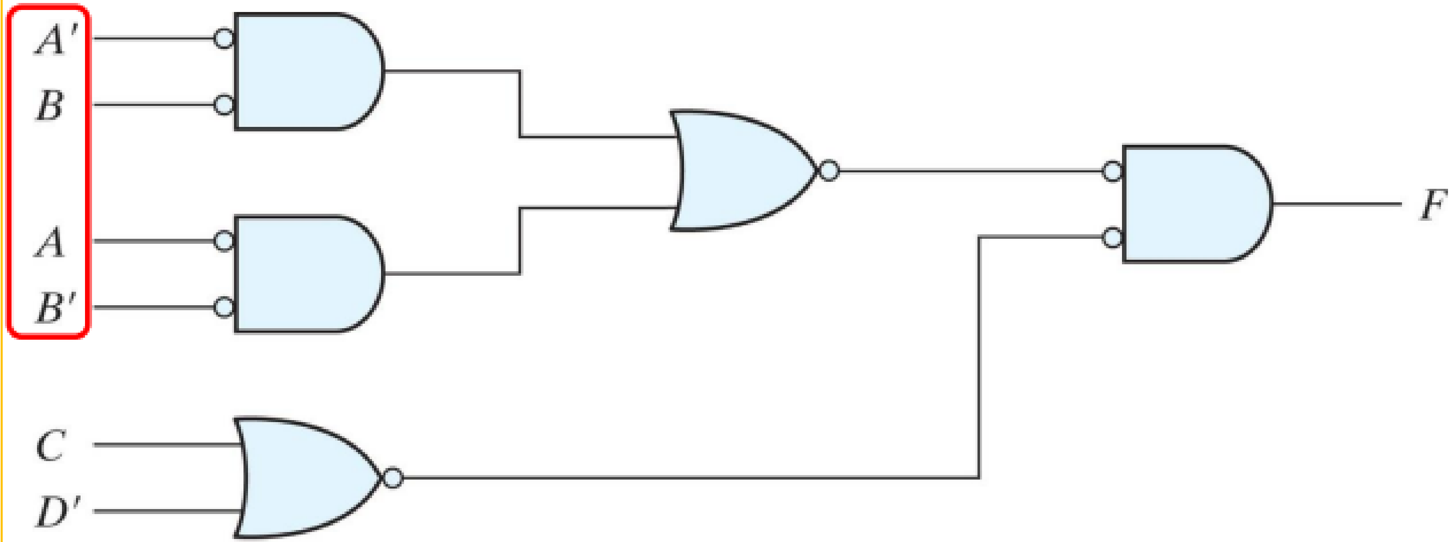
# NOR IMPLEMENTATION – 2-Level

**Extra Example:**

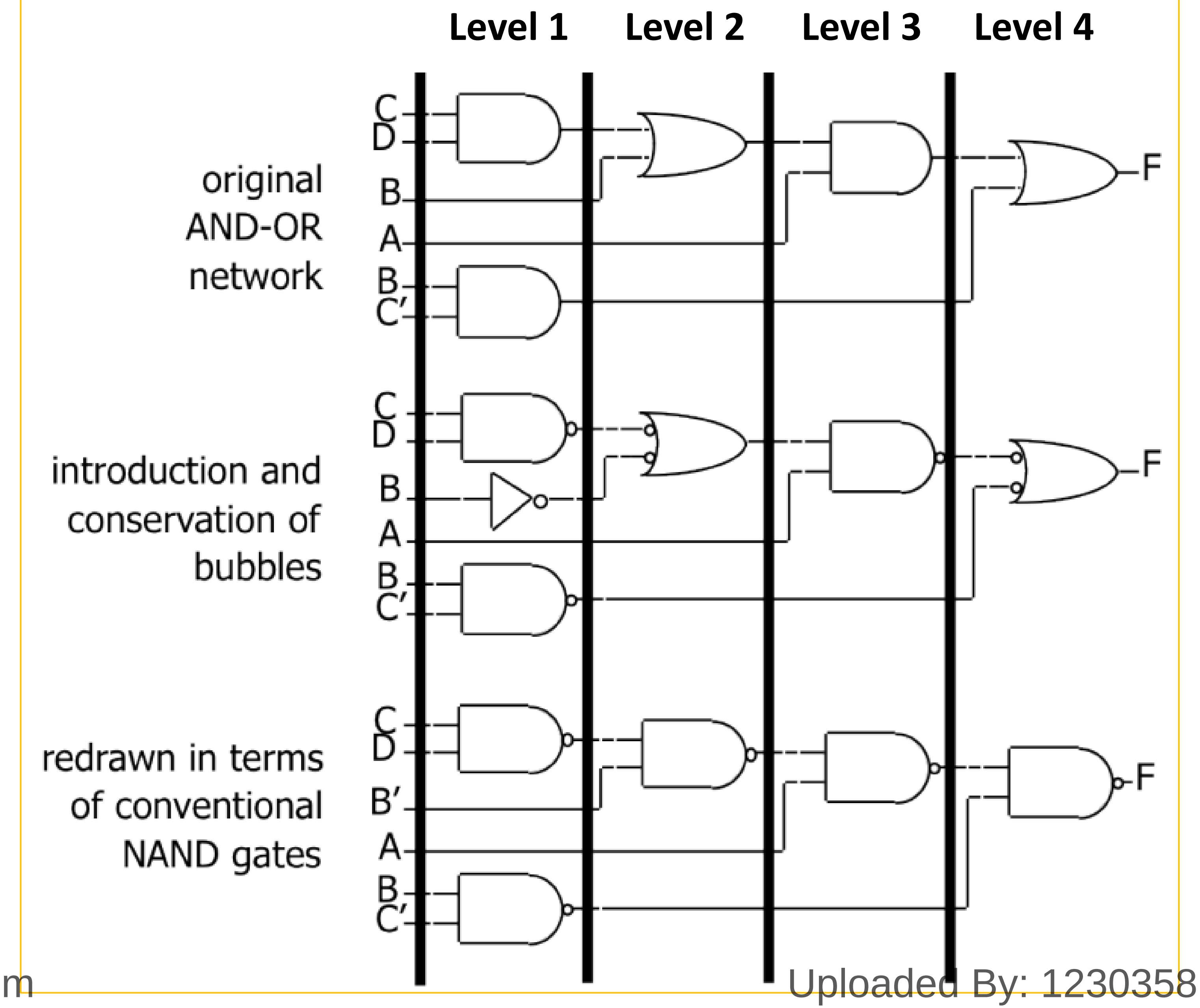
$$F = (AB' + A'B)(C + D')$$



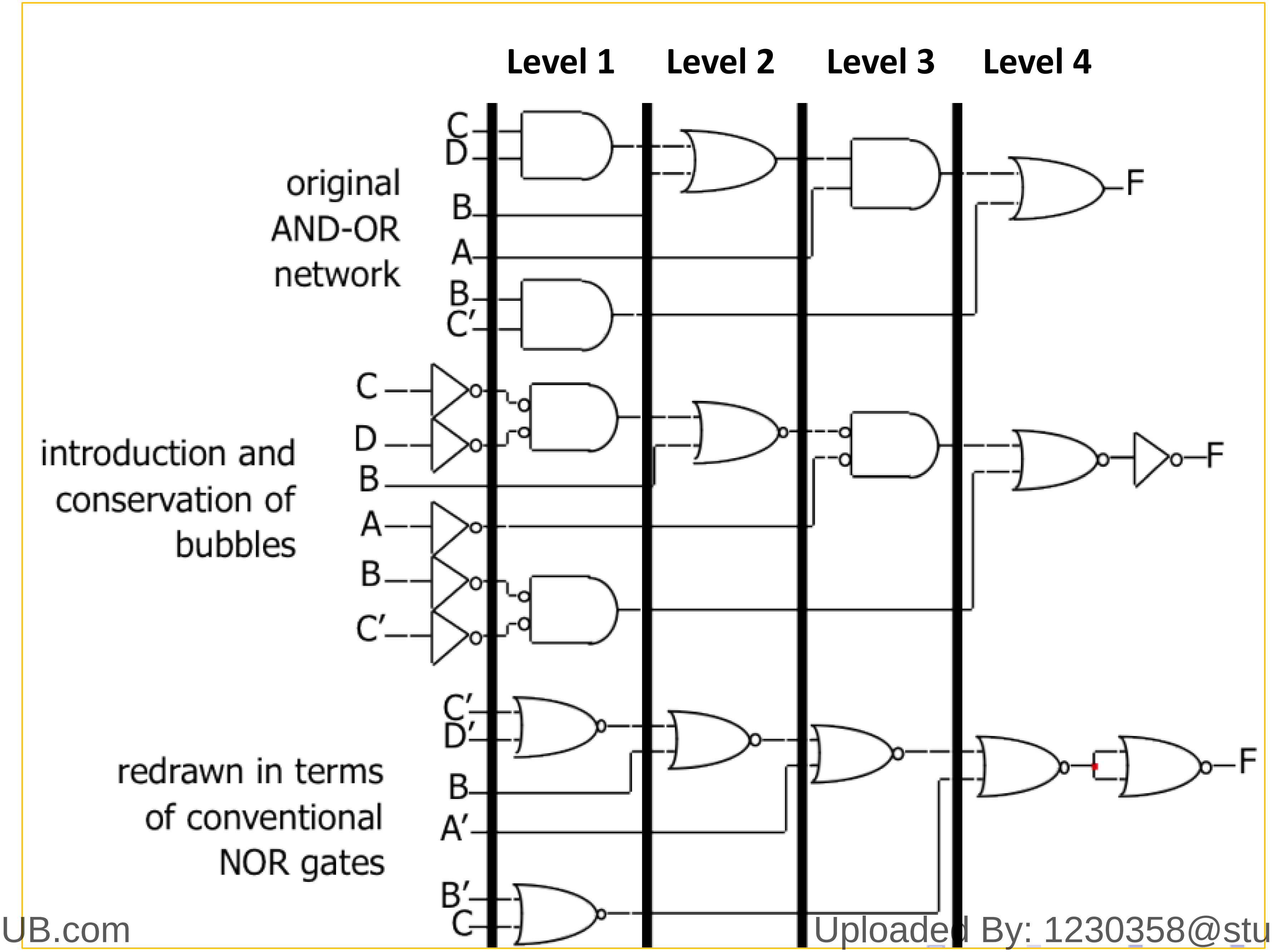
(a) AND–OR gates



# AND-OR to NAND Conversion



# AND-OR to NOR Conversion



# *Other Two-Level Implementations*

- SOP and POS are **basic** forms of expressing functions

- These functions can be implemented in **different** ways

- So far, we have seen **multiple** ways of implementing the SOP and POS expressions
  - ✓ SOP can be represented as two-level **AND-OR** logic
  - ✓ SOP can be represented as two-level **NAND-NAND** logic
  - ✓ POS can be represented as two-level **OR-AND** logic
  - ✓ POS can be represented as two-level **NOR-NOR** logic

**How many two-level logics may be formed using the four types of gates (AND, OR, NAND, NOR)?**

**$2^4$ → 16 Ways**

# *Other Two-Level Implementations*

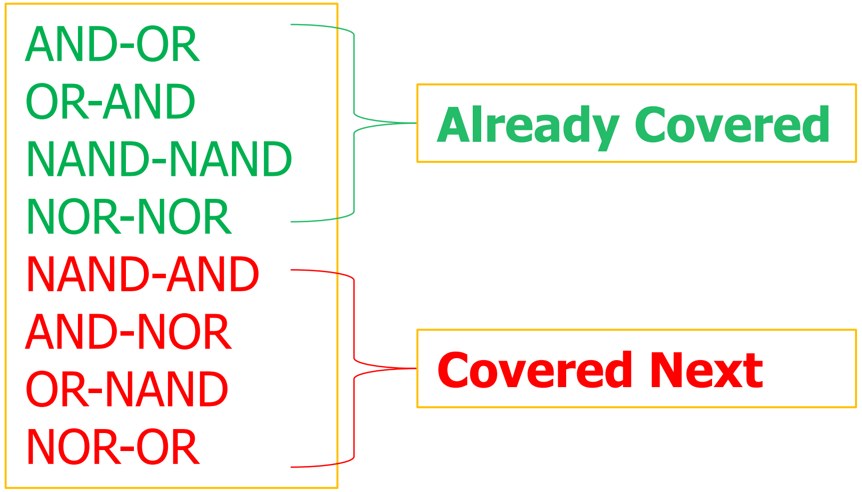
**How many two-level logics may be formed using the four types of gates (AND, OR, NAND, NOR)?**

**$2^4$ → 16 Ways**

AND-AND
AND-NAND
OR-OR
OR-NOR
NAND-OR
NAND-NOR
NOR-AND
NOR-NAND

**8**
**Degenerate**

**degenerate to a single operation → Straightforward**



AND-OR
OR-AND
NAND-NAND
NOR-NOR

**Already Covered**

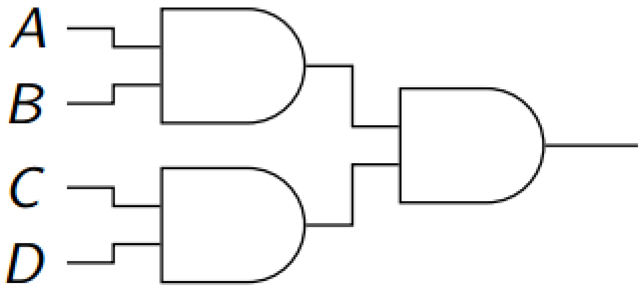
NAND-AND
AND-NOR
OR-NAND
NOR-OR

**Covered Next**

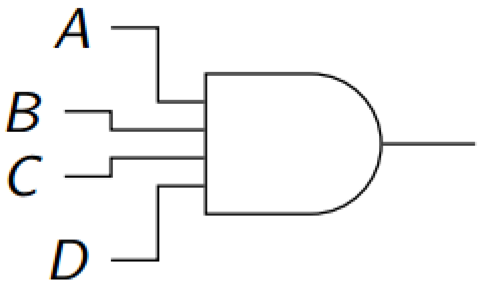
**8**
**NonDegenerate**

# Other Two-Level Implementations - Degenerate

## AND-AND

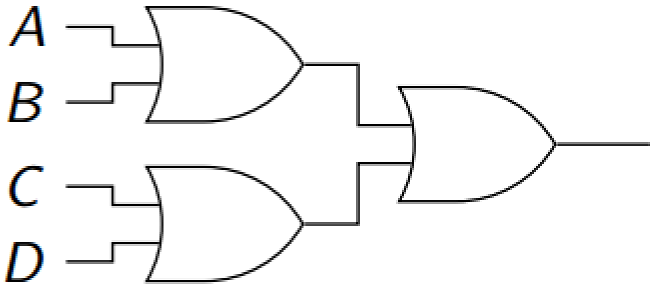


$$(AB)(CD) = ABCD$$

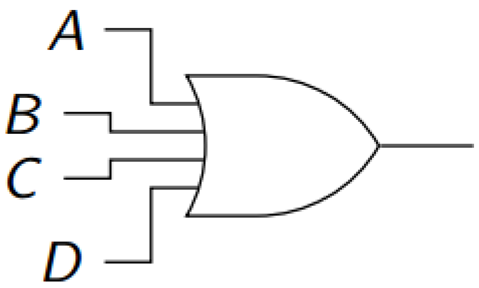


$\Rightarrow$ AND-AND $\equiv$ AND

## OR-OR



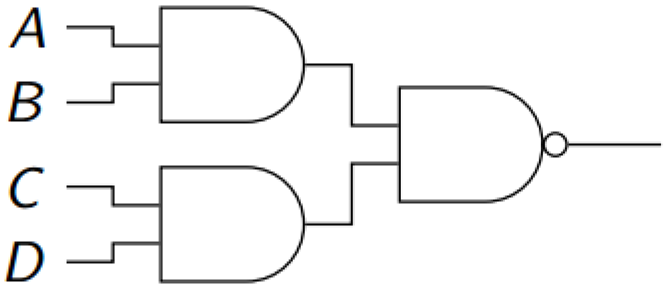
$$(A + B) + (C + D) = A + B + C + D$$



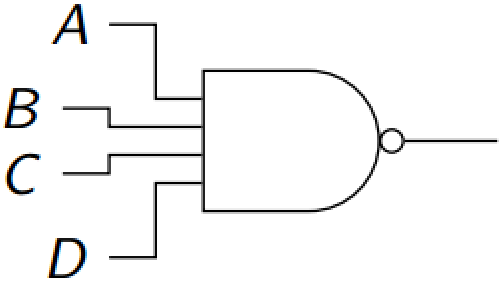
$\Rightarrow$ OR-OR $\equiv$ OR

# Other Two-Level Implementations - Degenerate

## AND-NAND

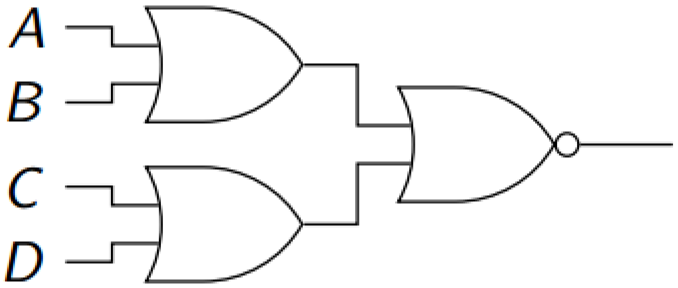


$$\overline{(AB)(CD)} = \overline{(ABCD)}$$

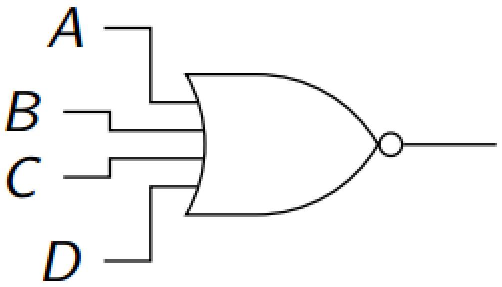


$\Rightarrow$ AND-NAND $\equiv$ NAND

## OR-NOR



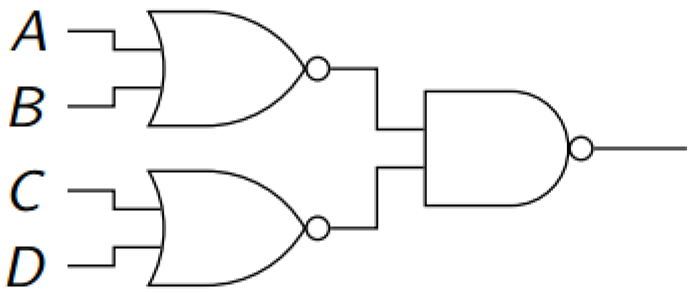
$$\overline{(A+B)+(C+D)} = \overline{(A+B+C+D)}$$



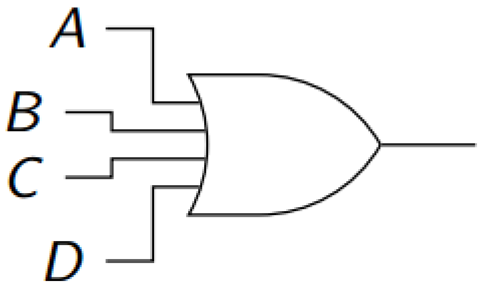
$\Rightarrow$ OR-NOR $\equiv$ NOR

# Other Two-Level Implementations - Degenerate

## NOR-NAND

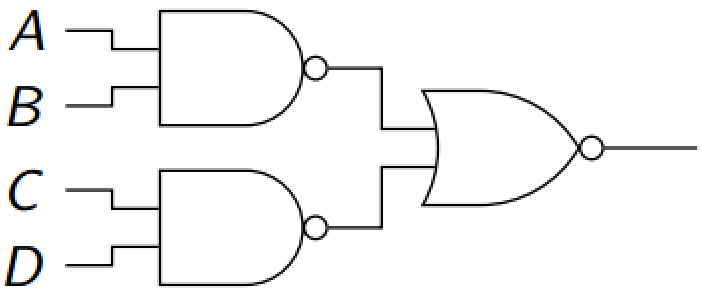


$$((\overline{A+B})(\overline{C+D}))' =$$
$$(\overline{A}\,\overline{B}\,\overline{C}\,\overline{D})' = A+B+C+D$$

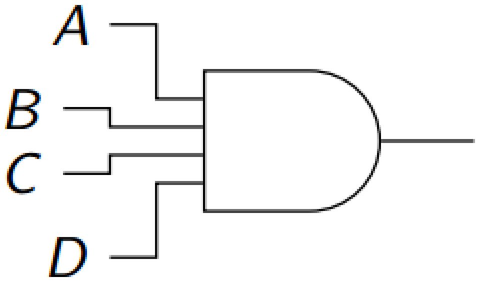


$\Rightarrow$ NOR-NAND $\equiv$ OR

## NAND-NOR



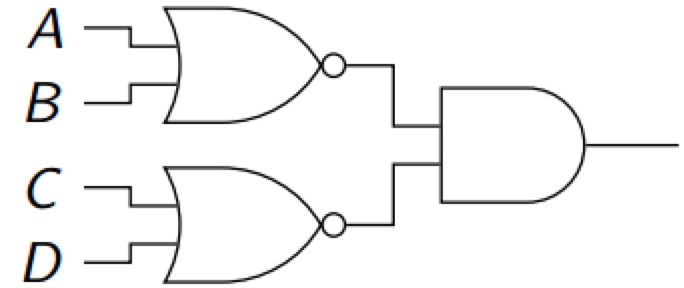
$$(\overline{AB} + \overline{CD})' = ABCD$$



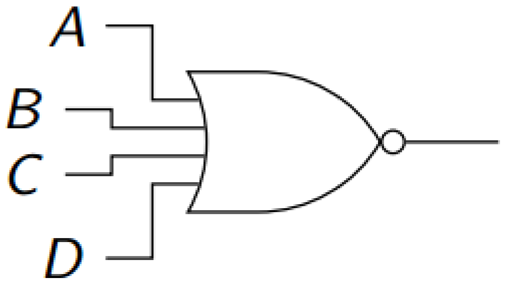
$\Rightarrow$ NAND-NOR $\equiv$ AND

# Other Two-Level Implementations - Degenerate

NOR-AND

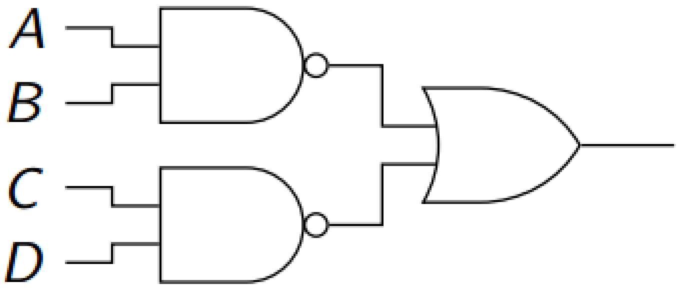


$$(\overline{A+B})(\overline{C+D}) =$$
$$\overline{(A+B)+(C+D)} =$$
$$\overline{A+B+C+D}$$

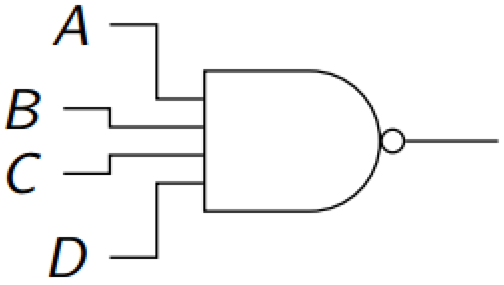


$\Rightarrow$ NOR-AND $\equiv$ NOR

NAND-OR



$$\overline{(AB)} + \overline{(CD)} =$$
$$\overline{(AB)(CD)} = \overline{ABCD}$$



$\Rightarrow$ NAND-OR $\equiv$ NAND

## *Other Two-Level Implementations - NonDegenerate*

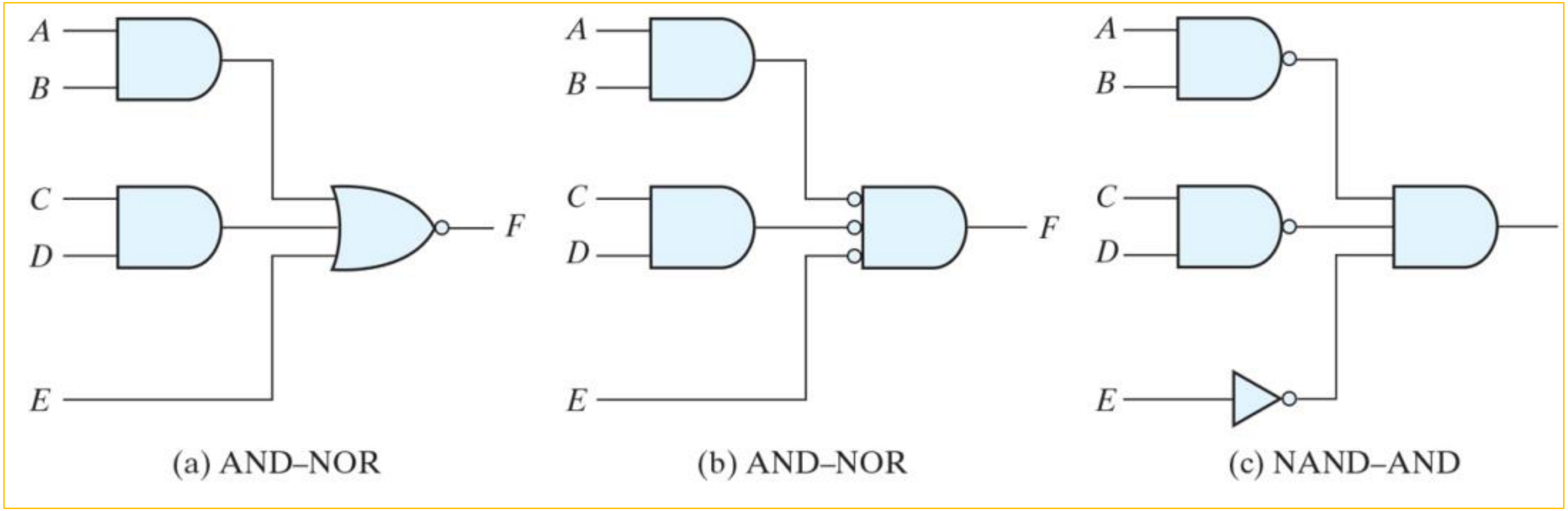
| | |
|---|---|
| OR-AND | AND-OR |
| NOR-NOR | NAND-NAND |
| NAND-AND | NOR-OR |
| AND-NOR | OR-NAND |

- The **first** gate listed in each of the forms constitutes a **first level** implementation.
- The **second** gate listed is a single gate placed in the **second level**.
- Note that any **two forms** listed in the same line are **duals** of each other.
- The green four forms have been investigated **previously**.
- The red four forms are investigated **next**.

# **AND-OR-INVERT** Implementation

- The two forms **NAND-AND** and **AND-NOR** are equivalent forms and can be treated together
- Both perform the **AND-OR-INVERT** function
- The **AND–OR–INVERT** implementation is similar to **AND-OR** (SOP), except for the **inversion**

$$F = (AB + CD + E)'$$

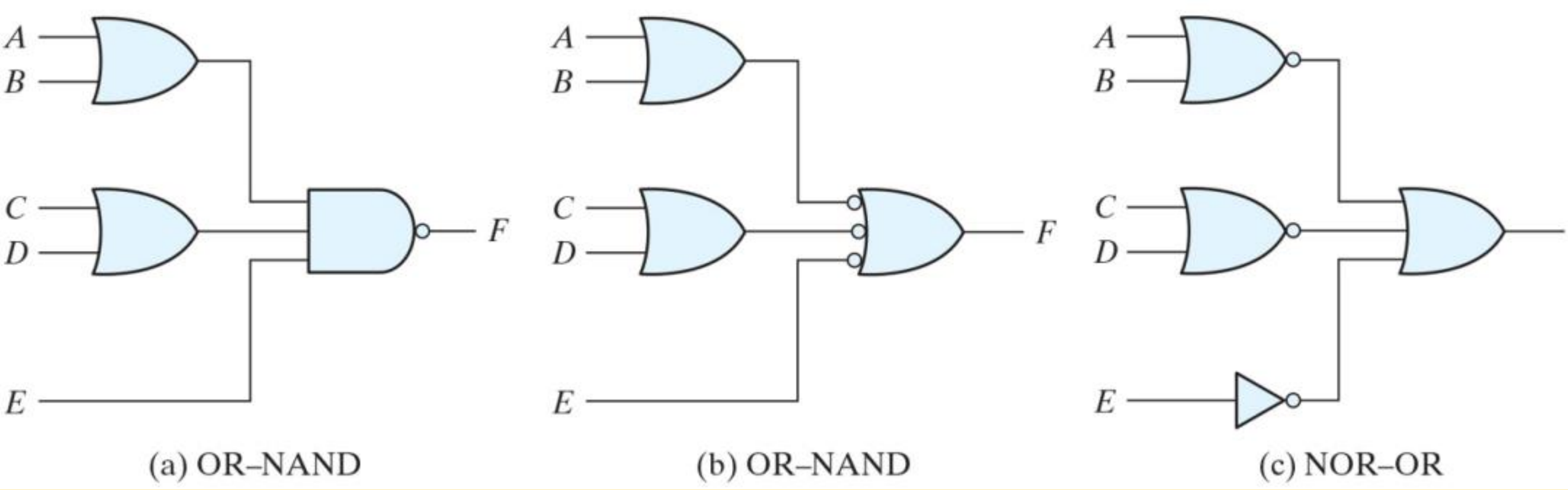


(a) AND–NOR (b) AND–NOR (c) NAND–AND

## **OR–AND–INVERT** Implementation

- The two forms **OR–NAND** and **NOR–OR** are equivalent forms and can be treated together
- Both perform the **OR–AND–INVERT** function
- The **OR–AND–INVERT** implementation is similar to **OR-AND** (POS), except for the **inversion**

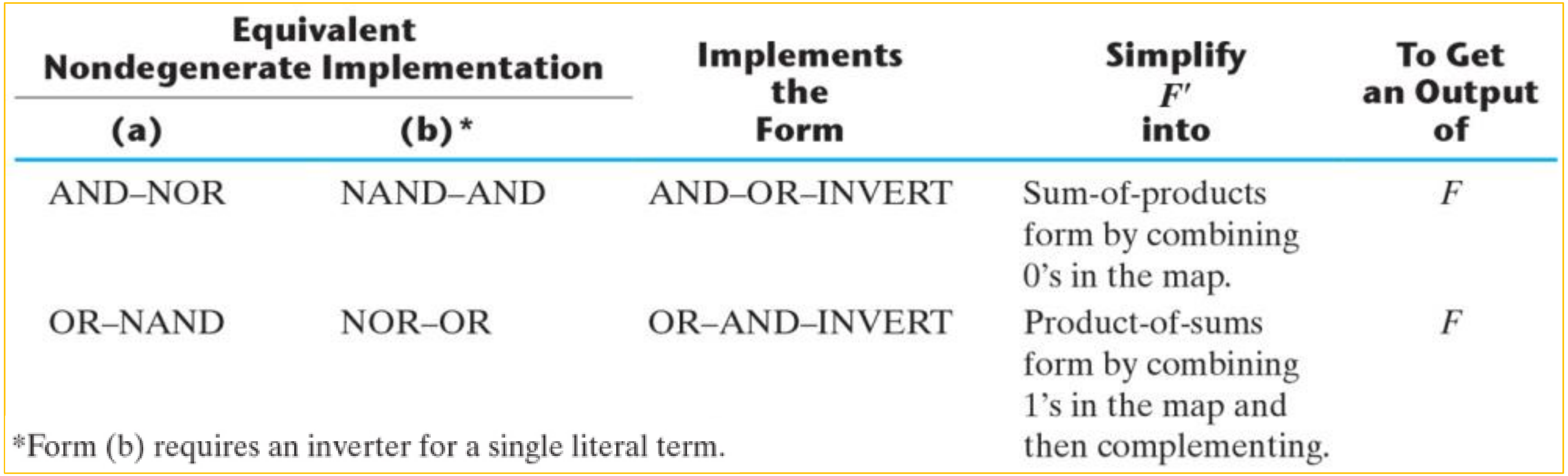
$$F = [\,(A+B)\,(C+D)\,E]'$$



(a) OR–NAND (b) OR–NAND (c) NOR–OR

# Other Two-Level Implementations – NonDegenerate Procedure

- The following table summarizes the procedures for implementing a Boolean function in any one of the **four** **2-level** forms: **NAND-AND, AND-NOR, OR-NAND and NOR-OR**



| **Equivalent Nondegenerate Implementation** | | **Implements the Form** | **Simplify $F'$ into** | **To Get an Output of** |
|---|---|---|---|---|
| **(a)** | **(b)*** | | | |
| 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.

- Because of the **INVERT** part in each case, it is convenient if we find the **simplification of $F'$**

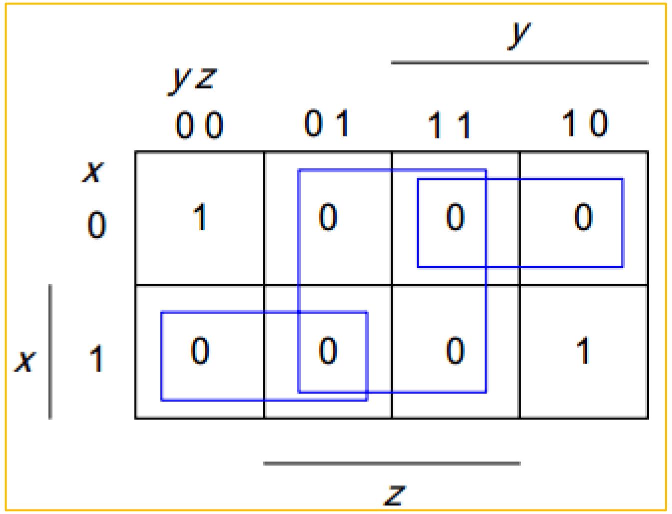
- When **$F'$** is implemented as an AND-OR or an OR-AND form, we can easily **get F** by simply **adding an INVERT at the end**. This will give us the circuit in one of the above-mentioned forms

# *Other Two-Level Implementations – NonDegenerate Procedure*

## Implementation With **AND-NOR** and **NAND-AND**

$$F(x, y, z) = \Sigma(0, 6)$$

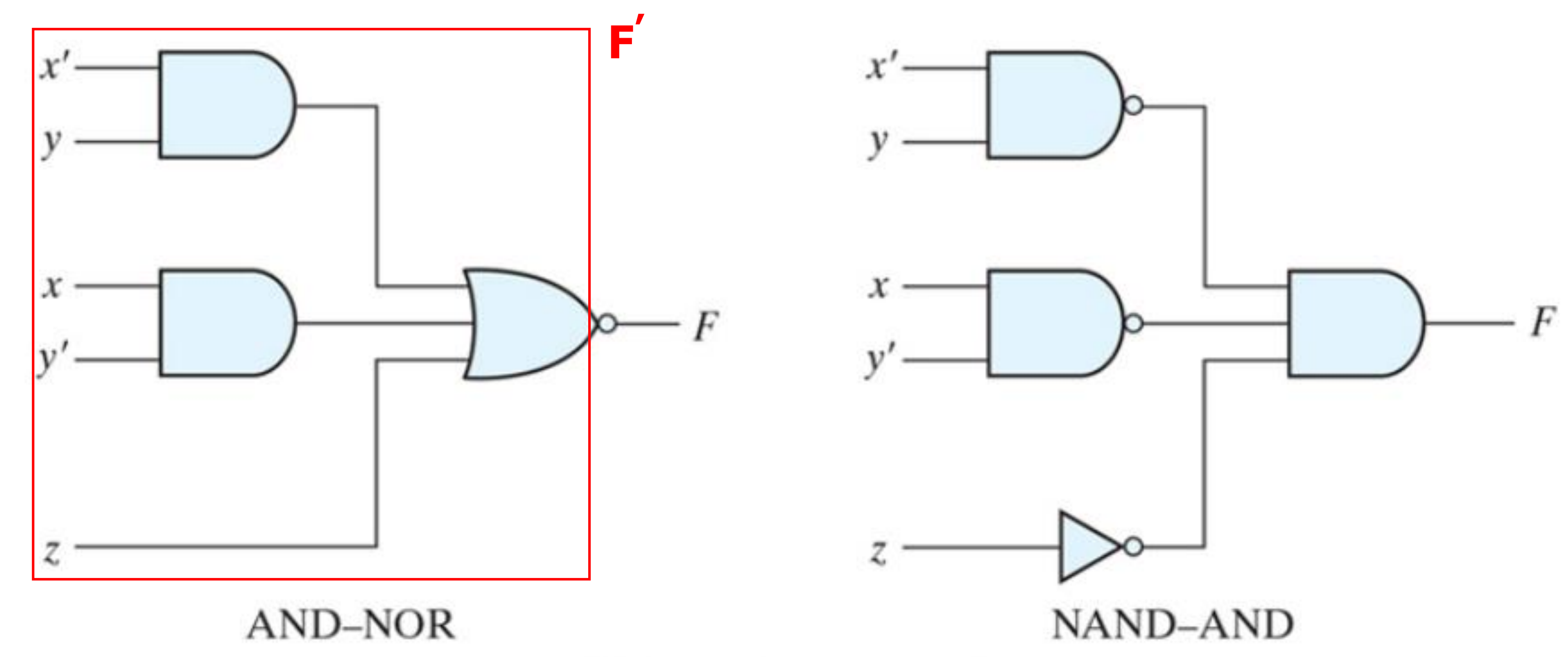
1) Find the **simplified** form of **F′**
   - ✓ Form a K-map and **group 0's** → SOP form of **F′**
2) Implement **F′** in a two-level **AND-OR** form
3) By adding a **NOT** gate at the output, we get **F**
   - ✓ The resulting form is **AND-OR-INVERT** form which can be easily converted to get AND-NOR and NAND-AND forms



| x \ yz | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |

$$F' = x'y + xy' + z$$

Remember: We always get a SOP from the K-map



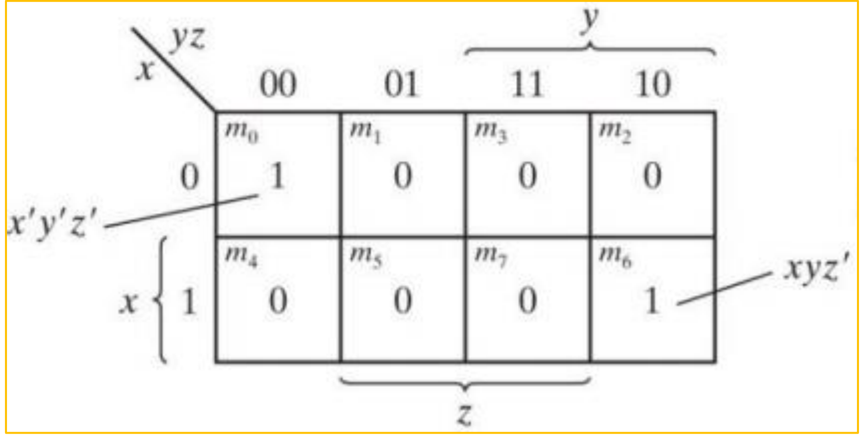
$$F = (x'y + xy' + z)'$$

# Other Two-Level Implementations – NonDegenerate Procedure

Implementation With **OR-NAND** and **NOR-OR**

$$F(x, y, z) = \Sigma(0, 6)$$

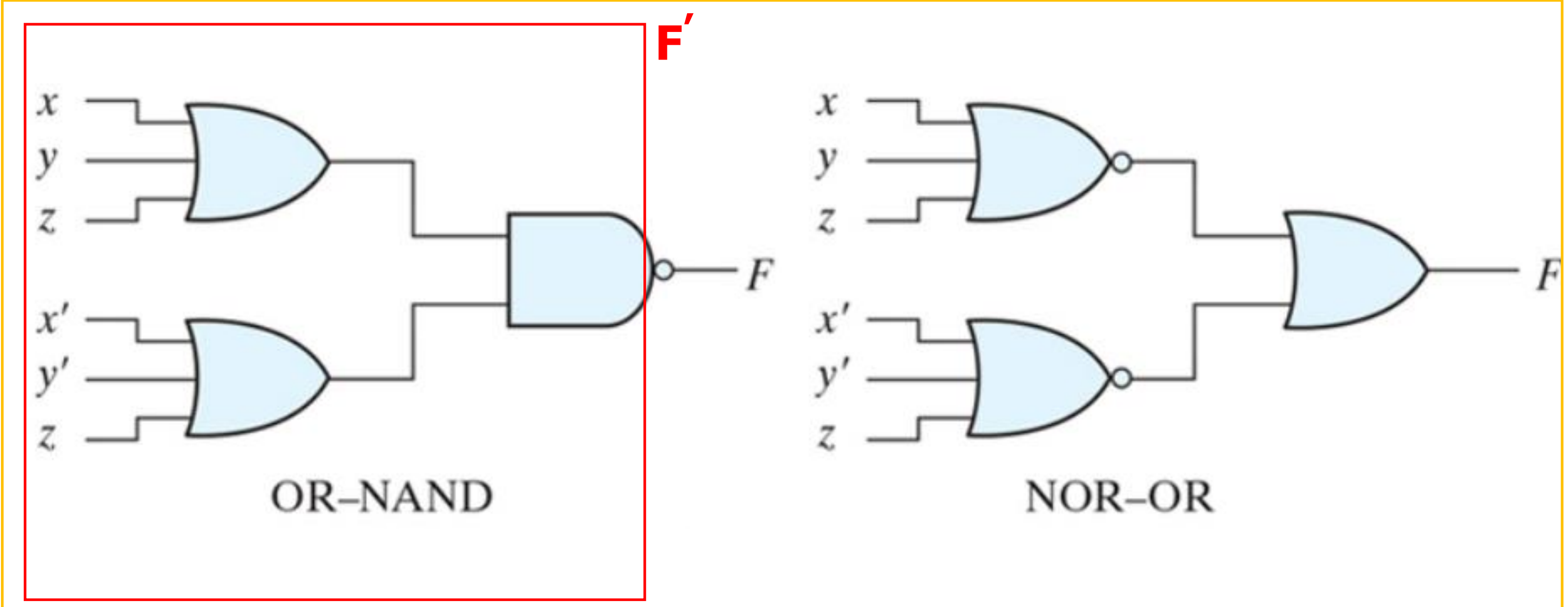
1) Find the **simplified** form of **F**
   - ✓ Form a K-map and **group 1's** → SOP form of **F** → find **F′** by taking the **complement** of **F**
2) Implement **F′** in a two-level **OR-AND** form
3) By adding a **NOT** gate at the output, we get **F**
   - ✓ The resulting form is **OR-AND-INVERT** form which can be easily converted to get OR-NAND and NOR-OR forms



$$F = x'y'z' + xyz'$$

⬇

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



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

# *Other Two-Level Implementations – NonDegenerate Summary*

| **Form** | **SOP from K-map?**<br>**DeMorgan Invert?** | **Gate Type** | **Procedure** |
|---|---|---|---|
| AND-OR (AO) | F<br>No | AND-OR = NAND-NAND (SOP) | Circle **1's** in the K-Map and minimize |
| AND-OR-INVERT (AOI) | **F’**<br>No | AND-NOR =NAND-AND (SOP **Invert**) | Circle **0's** in the K-Map and minimize |
| OR-AND (OA) | **F’**<br>Yes | OR-AND =NOR-NOR (POS) | Circle **0's** in the K-Map and minimize SOP. Use **DeMorgan's** to transform to POS |
| OR-AND-INVERT (OAI) | F<br>Yes | OR-NAND = NOR-OR (POS **Invert**) | Circle **1's** in the K-Map and minimize SOP. Use **DeMorgan's** to transform to POS. |

Remember: We always get a **SOP** from the K-map

# Other Two-Level Implementations –Summary

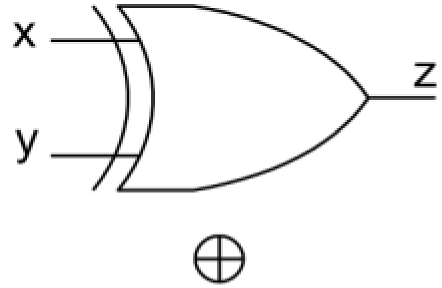
| Level 1 | Level 2 | Equivalent | Final Form | Remarks |
|---|---|---|---|---|
| AND | AND | AND-AND | AND | Degenerate |
| AND | OR | AND-OR | SOP | |
| AND | NAND | AND-AND-NOT | AND-NOT | Degenerate |
| AND | NOR | AND-OR-NOT | SOP-INVERT | |
| OR | AND | OR-AND | POS | |
| OR | OR | OR-OR | OR | Degenerate |
| OR | NAND | OR-AND-NOT | POS-INVERT | |
| OR | NOR | OR-OR-NOT | OR-NOT | Degenerate |
| NAND | AND | AND-NOT-NOT-NOR | SOP-INVERT | |
| NAND | OR | NOT-OR-OR | NOT-OR | Degenerate |
| NAND | NAND | AND-NOT-NOT-OR | SOP | |
| NAND | NOR | AND-NOT-NOT-AND | AND | Degenerate |
| NOR | AND | NOT-AND-AND | NOT-AND | Degenerate |
| NOR | OR | OR-NOT-NOT-AND-NOT | POS-INVERT | |
| NOR | NAND | OR-NOT-NOT-OR | OR | Degenerate |
| NOR | NOR | OR-NOT-NOT-AND | POS | |

# *Exclusive Functions*

- The **Exclusive-OR (XOR)** function is an important Boolean function used **extensively** in logic circuits.
- Implemented **directly** as an electronic circuit (true gate) or implemented by **interconnecting other** gate types.
- The **Exclusive-NOR (XNOR)** function, also known as **equivalence** function, is the **complement** of the **XOR** function.

# Exclusive Functions

## XOR

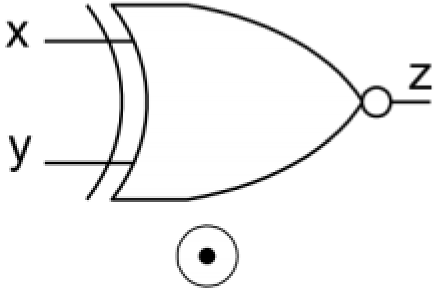


$$z = x \oplus y$$
$$= x\bar{y} + \bar{x}y$$

| x | y | $x \oplus y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

The result is 1 only when X and Y are NOT equal (Not **equivalent**).

## XNOR



$$z = x \odot y$$
$$= \bar{x}\bar{y} + xy$$

| x | y | $x \odot y$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

The result is 1 only when both X and Y are equal (**equivalent**).

# XOR / XNOR Identities

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

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

$x \oplus 0 = x, \; x \oplus 1 = x'$

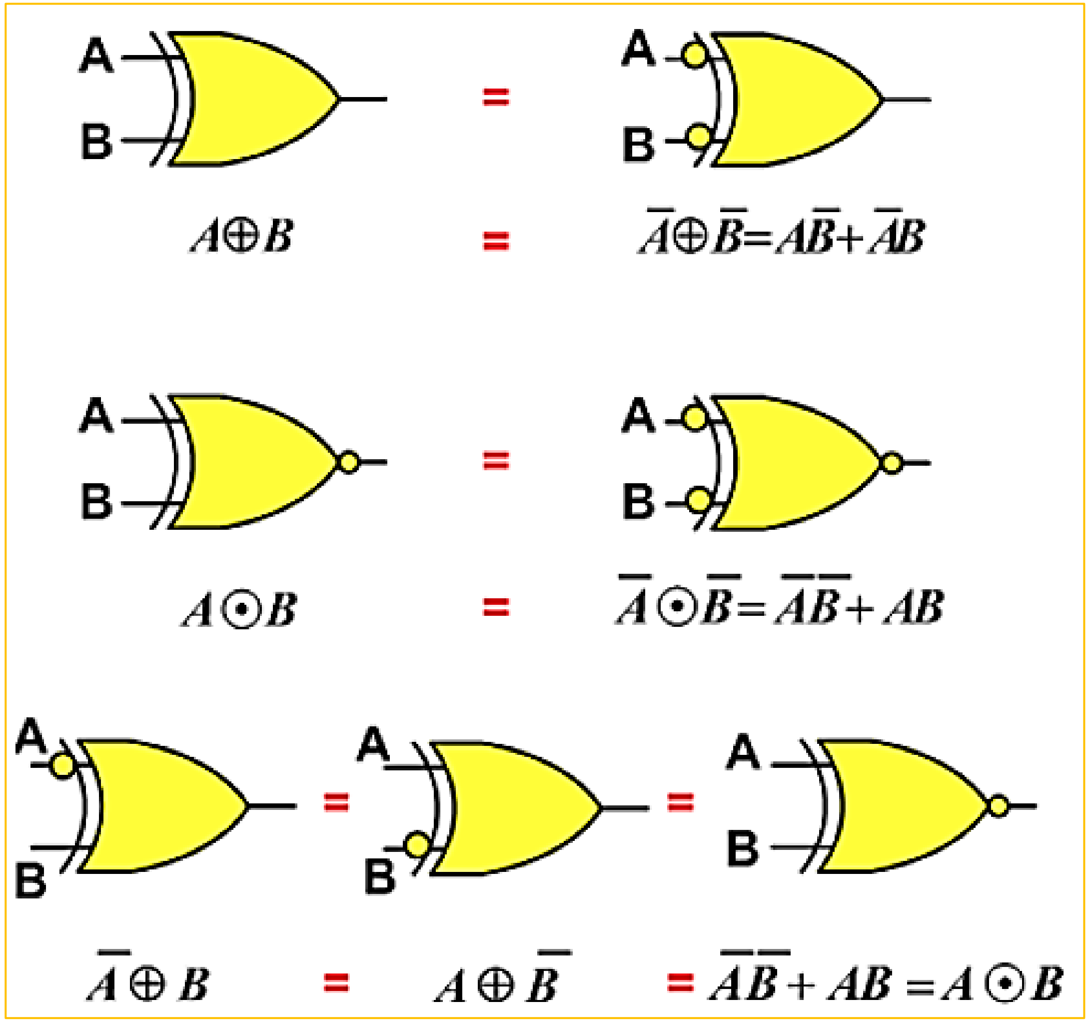
$x \oplus x = 0, \; x \oplus x' = 1$

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

$x \oplus y = y \oplus x$ **Commutative**

$(x \oplus y) \oplus z = x \oplus (y \oplus z) = x \oplus y \oplus z$ **Associative**

# XOR / XNOR Properties

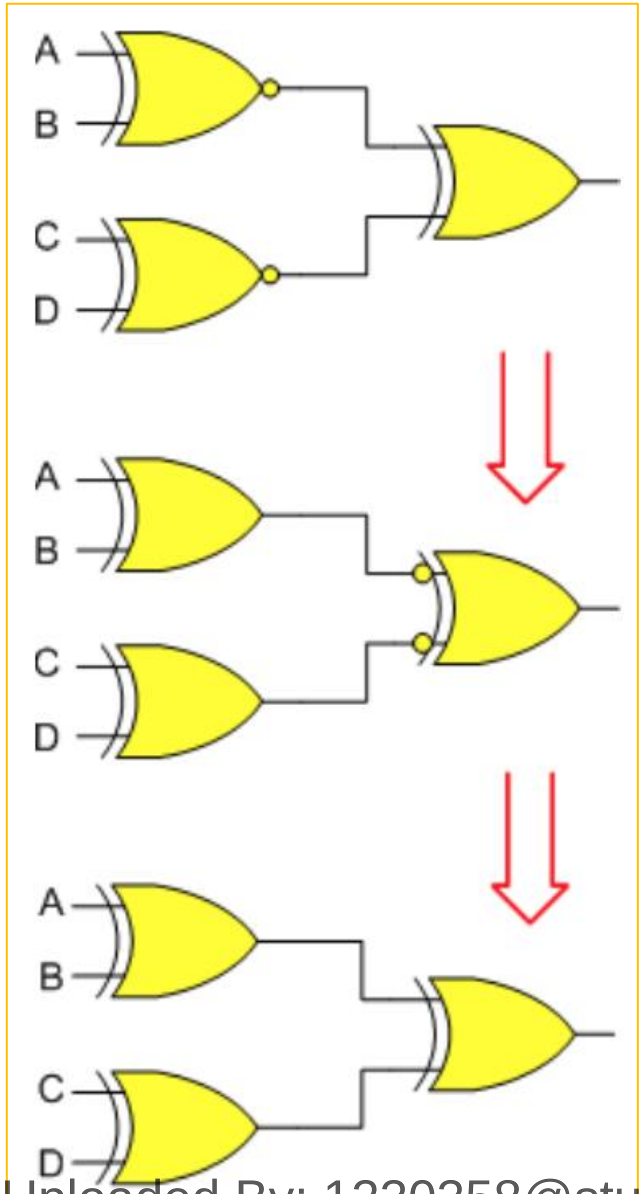


$A \oplus B$ = $\bar{A} \oplus \bar{B} = A\bar{B} + \bar{A}B$

$A \odot B$ = $\bar{A} \odot \bar{B} = \bar{A}\bar{B} + AB$

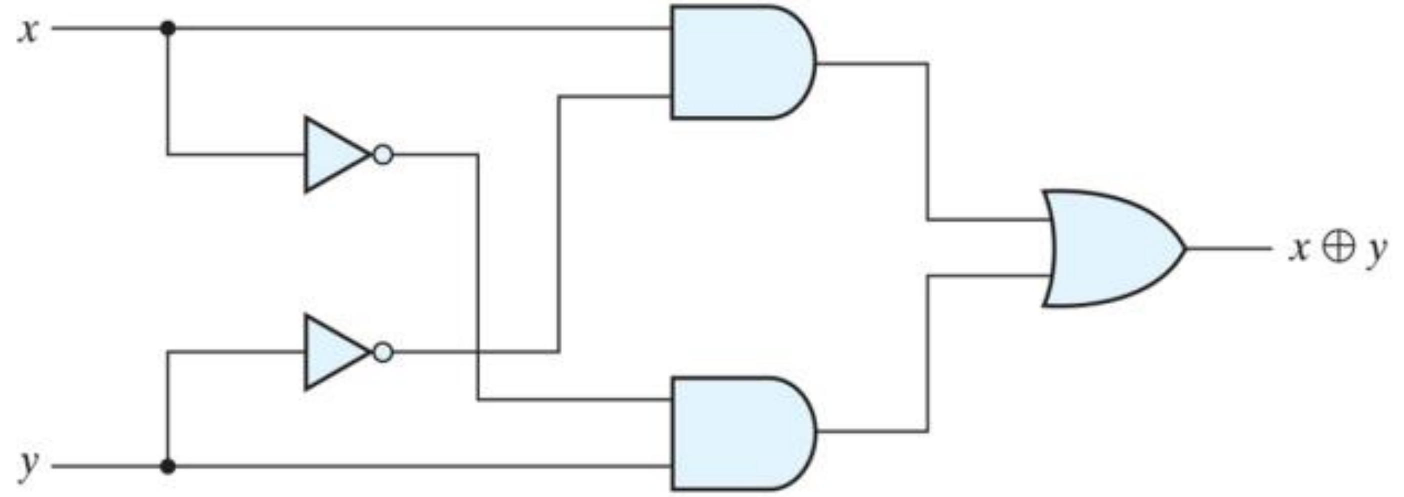
$\bar{A} \oplus B$ = $A \oplus \bar{B}$ = $\bar{A}\bar{B} + AB = A \odot B$

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

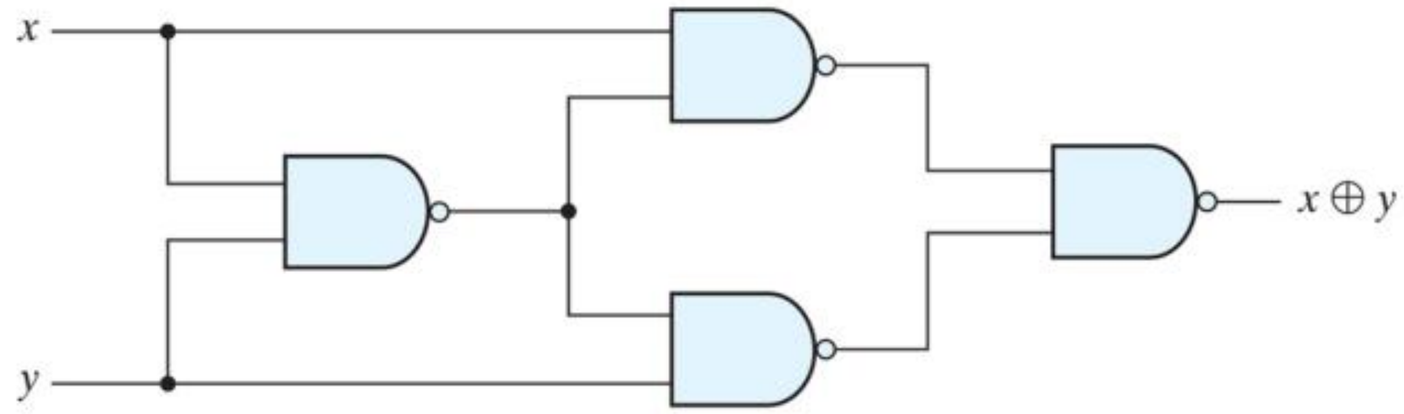


# Implementation of XOR Gate

- Multi-input XOR gates are **difficult** to fabricate with hardware
- Even a two-input gate is usually **constructed** with **other** types of gates



(a) Exclusive-OR with AND–OR–NOT gates



(b) Exclusive-OR with NAND gates

# XOR is an Odd Function

- The exclusive-OR operation with three or more variables can be expressed as:

$$\begin{aligned} & A \oplus B \oplus C \\ &= (AB' + A'B)C' + (AB + A'B')C \\ &= AB'C' + A'BC' + ABC + A'B'C \\ &= \sum(1,2,4,7) \longrightarrow \begin{cases} 001 \\ 010 \\ 100 \\ 111 \end{cases} \end{aligned}$$

- The multiple-variable exclusive-OR operation is defined as an **odd function** ('1' when **odd** number of **variables** are equal to **1**)

| $A$ | $B$ | $C$ | $A \oplus B \oplus C$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | |
| 0 | 0 | 1 | 1 | Odd number of 1's |
| 0 | 1 | 0 | 1 | Odd number of 1's |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 1 | Odd number of 1's |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 0 | |
| 1 | 1 | 1 | 1 | Odd number of 1's |

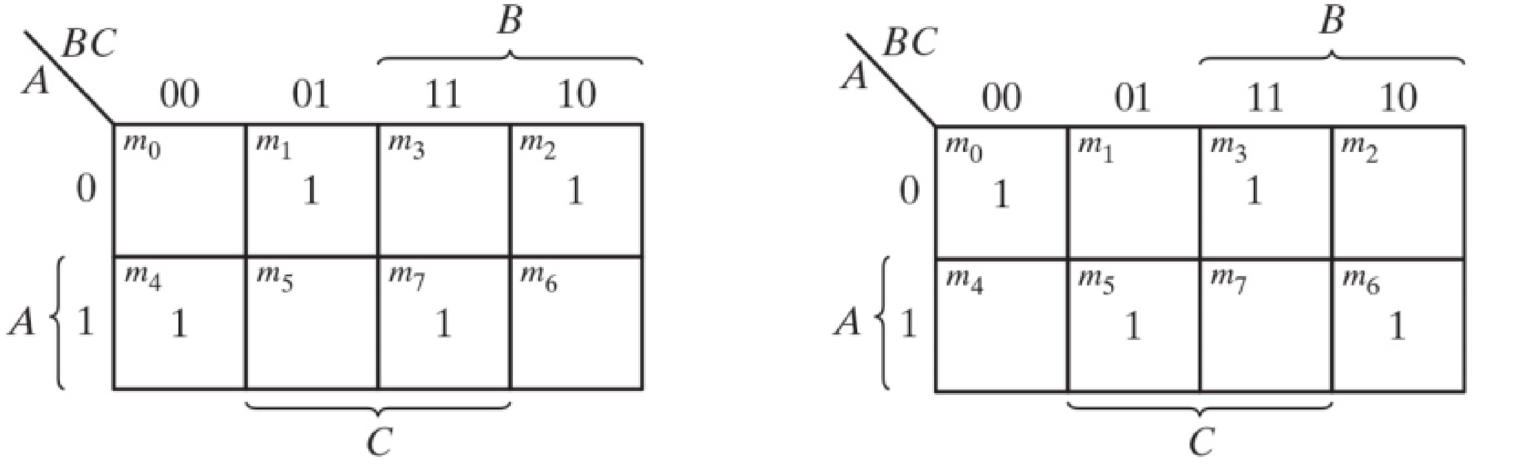
# XNOR is an Even Function

- The complement of an **odd** function is an **even** function

$$(A \oplus B \oplus C)' = \Sigma(0,3,5,6)$$

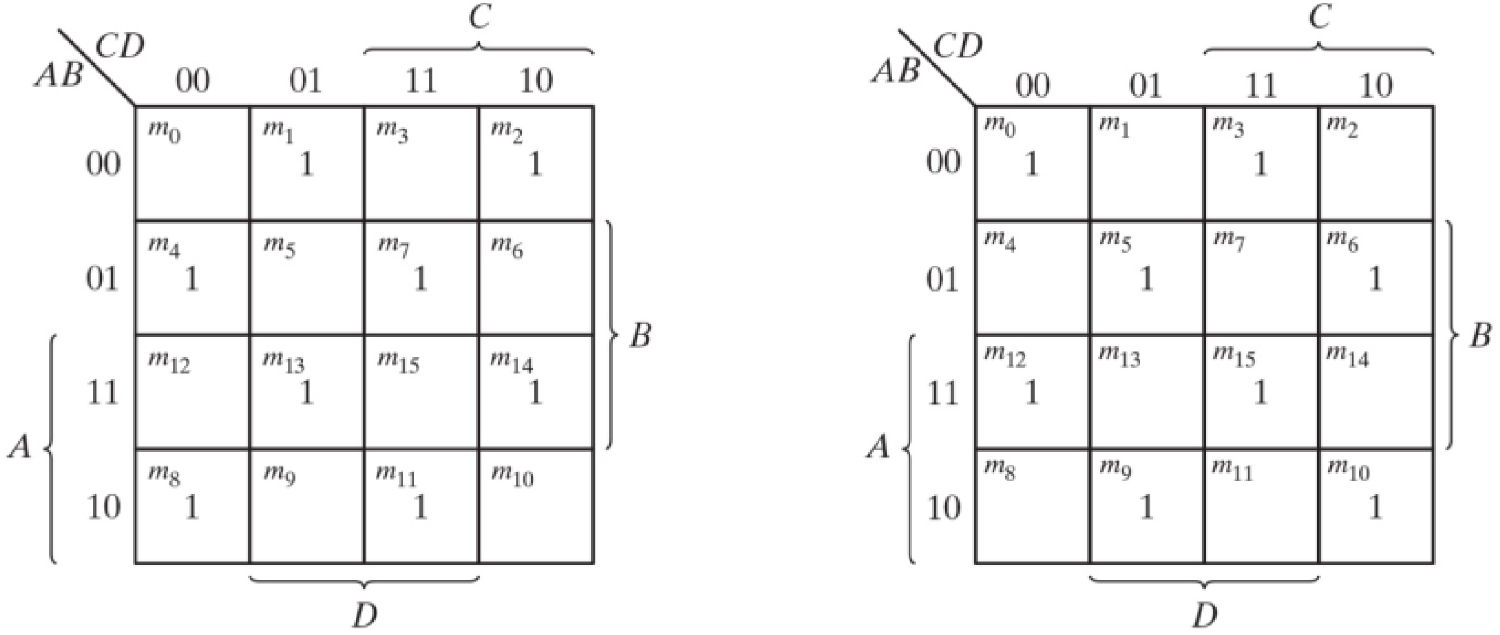
| $A$ | $B$ | $C$ | $(A \oplus B \oplus C)'$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | Even number of 1's |
| 0 | 0 | 1 | 0 | |
| 0 | 1 | 0 | 0 | |
| 0 | 1 | 1 | 1 | Even number of 1's |
| 1 | 0 | 0 | 0 | |
| 1 | 0 | 1 | 1 | Even number of 1's |
| 1 | 1 | 0 | 1 | Even number of 1's |
| 1 | 1 | 1 | 0 | |

# K-Maps for XOR and XNOR Operations



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

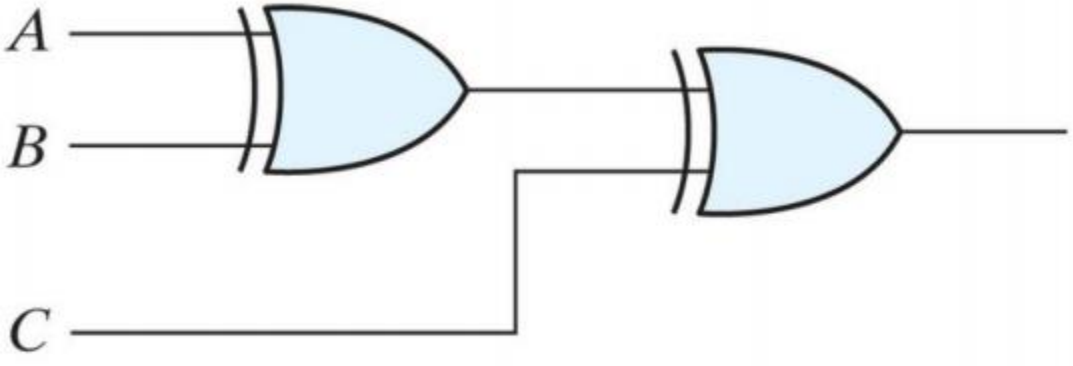
(b) Even function $F = (A \oplus B \oplus C)'$



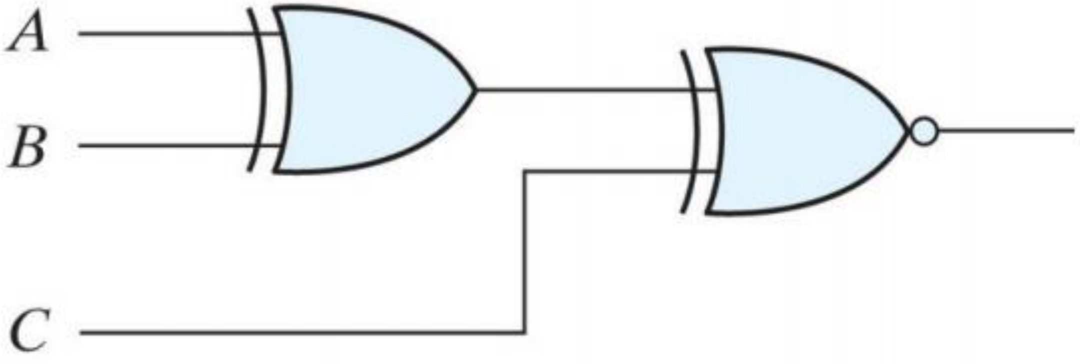
(a) Odd function $F = A \oplus B \oplus C \oplus D$

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

# Gate Implementation for 3-Inputs Odd/Even Functions



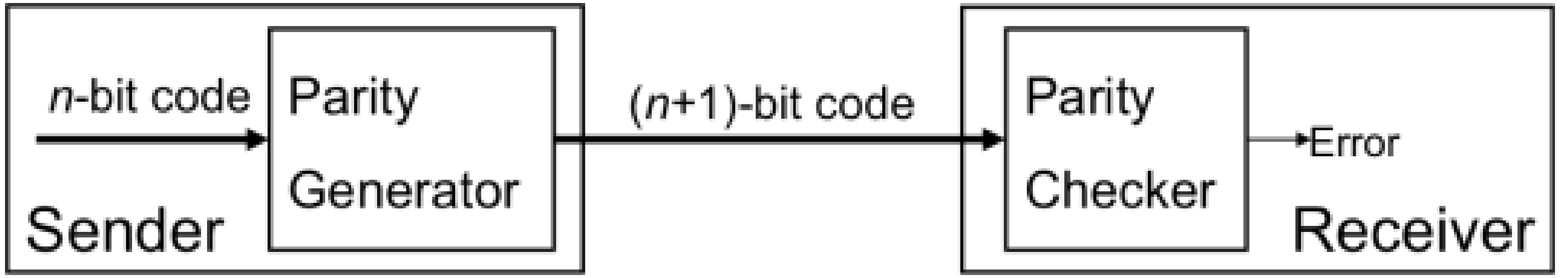
(a) 3-input odd function



(b) 3-input even function

# Parity Generation/Checking (Application for Odd/Even Functions)

- A **parity bit** is an extra bit included with a binary message to make the **number of 1's** either **odd or even**.
- Parity bit is used for the purpose of **detecting errors** during the transmission of binary information.
- The circuit that **generates** the parity bit in the **transmitter** is called a **parity generator**.
- The circuit that **checks** the parity in the **receiver** is called a **parity checker**.
- **Exclusive-OR/NOR** gates (Odd/Even Functions) are useful for generating and checking a parity bit



# Parity Generation/Checking (Application for Odd/Even Functions)

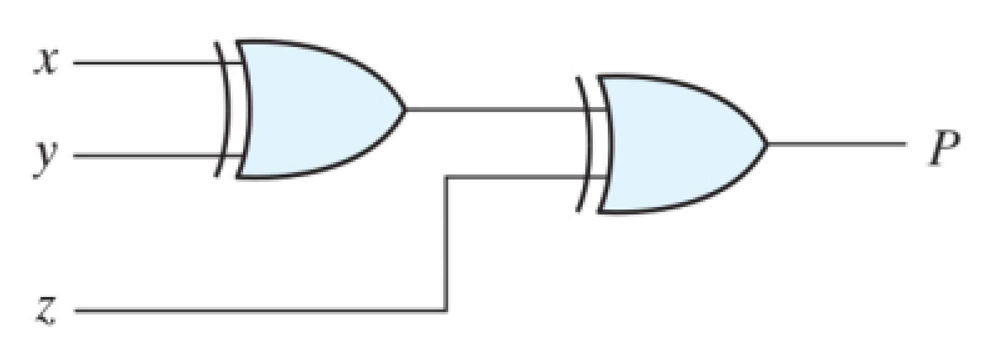
**Example:**

Transmitting a **3-bit** message with **even parity bit**. The three bits – x, y, and z constitute the message and are the inputs to the circuit.
The parity bit **P** is the **output.**

## Parity Generator

**P** is **Even parity** bit → P = 1 if the number of 1's in the 3-bit message is **odd** → **P** is an **odd function** and can be implemented using **3-Inputs XOR**

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



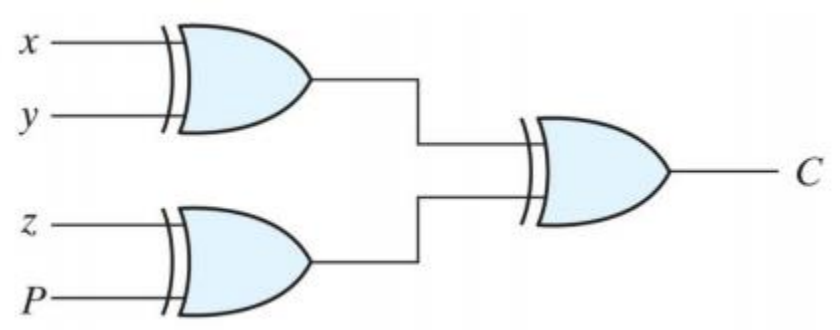
**Parity Generator**

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

# Parity Checker

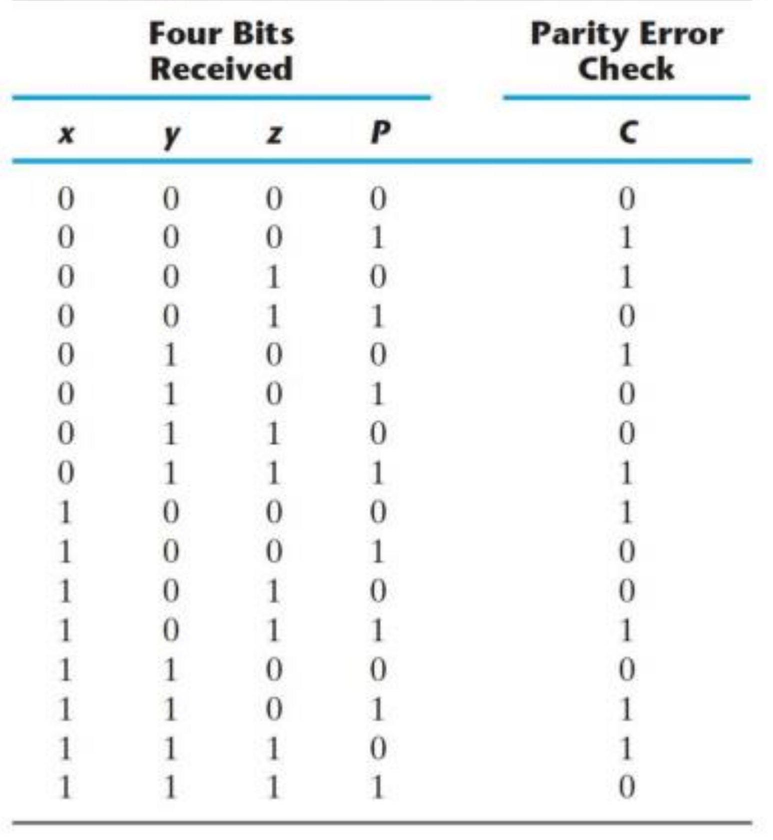
1) The **three bits** in the message together with the **even** parity bit **P** are transmitted (**4 bits**)
2) The **receiver** at the destination checks for an **even** number of 1's in the **4-bit** message and generates an **error C** equal to 1 if the **number of 1's** in the message is **odd**
3) **C** (error) can be implemented using **XOR**

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



**Parity Checker**

**C = 1 → Error**



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