

LOGIC DESIGN

Chapter 3

Gate-Level Minimazation

OUTLINE OF CHAPTER 3

- The Map Method
- Four – Variable K-Map
- Product of Sums Simplification
- Don’t Care Conditions
- NAND and NOR Implementations
- Other Two Level Implementations
- Exclusive-OR Function
- Hardware Description Language



3.1 THE MAP METHOD

THE MAP METHOD

- **Gate-level minimization** refers to the design task of finding an optimal gate-level implementation of Boolean functions describing a digital circuit.

THE MAP METHOD

- The complexity of the digital logic gates
  - The complexity of the algebraic expression
- Logic minimization
  - Algebraic approaches: lack of specific rules
  - **The Karnaugh map**
    - A simple straight forward procedure
    - A pictorial form of a truth table
    - Applicable if the # of variables < 7

THE MAP METHOD

- A diagram made up of squares
  - Each square represents one minterm
- The simplified expression produced by the map are always in one of the two standard forms:
  - Sum of products
  - Product of sums

THE MAP METHOD

- Boolean function
  - Sum of minterms
  - Sum of products (or product of sum) in the simplest form
  - A minimum number of terms
  - A minimum number of literals
  - The simplified expression may not be unique

THE MAP METHOD

- A two-variable map
  - **Four** minterms
  - **x' =** row 0; **x** = **row 1**
  - **y'** = **column 0**; **y** = **column 1**
  - A truth table in square diagram
  - xy

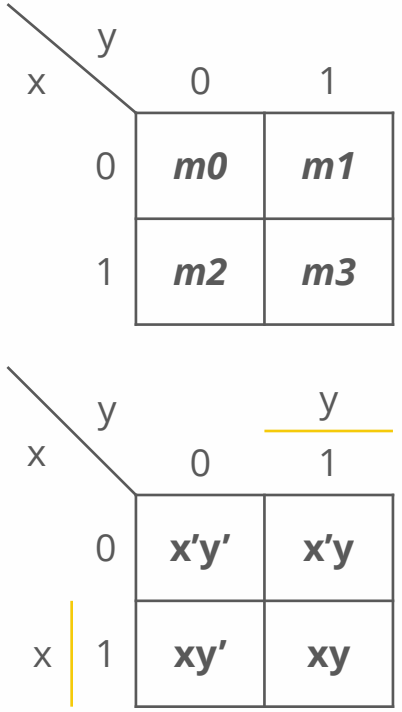
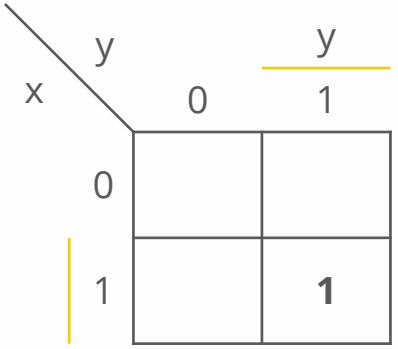


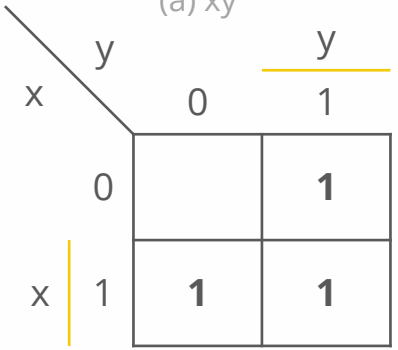
Figure 3.1 Two-variable Map

THE MAP METHOD

- Fig. 3.2(a):
  - $xy = m_3$
- Fig. 3.2(b):
  - $x+y = x'y+xy' +xy = m_1+m_2+m_3$



(a) xy



(b) x+y

Figure 3.2 Two-variable Map

THE MAP METHOD

- Three-variable map:
  - For **3** binary variables.
  - **2^n** = **8** minterms.
  - Map consists of **8** squares.
  - Minterms are not arranged in a binary sequence.
  - Only one bit changes in value from one adjacent column to the next.

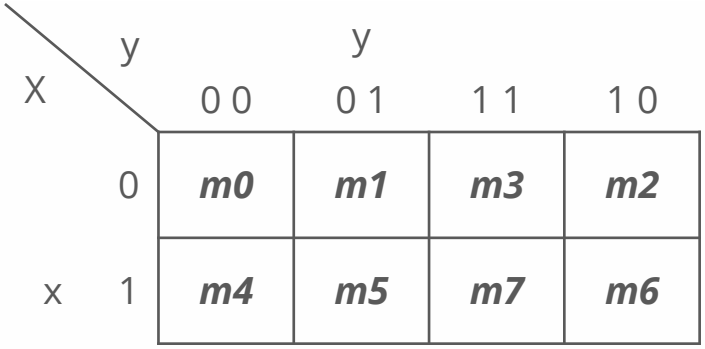
x \ y	0 0	0 1	1 1	1 0
0	***m0***	***m1***	***m3***	***m2***
1	***m4***	***m5***	***m7***	***m6***

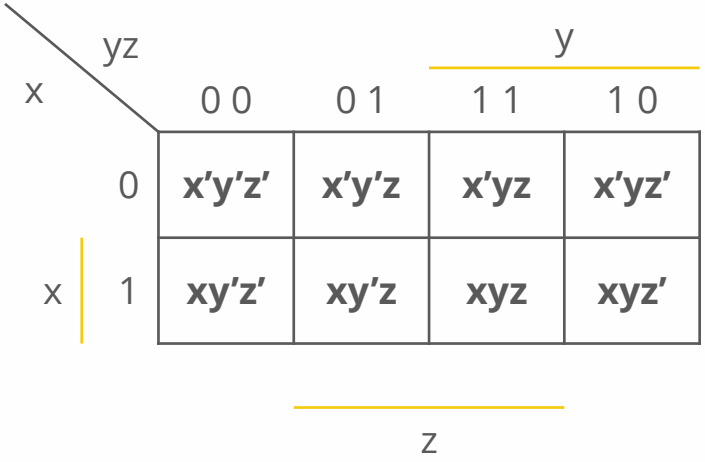
x \ yz	0 0	0 1	1 1	1 0
0	**x'y'z'**	**x'y'z**	**x'yz**	**x'yz'**
1	**xy'z'**	**xy'z**	**xyz**	**xyz'**

y y z

THE MAP METHOD

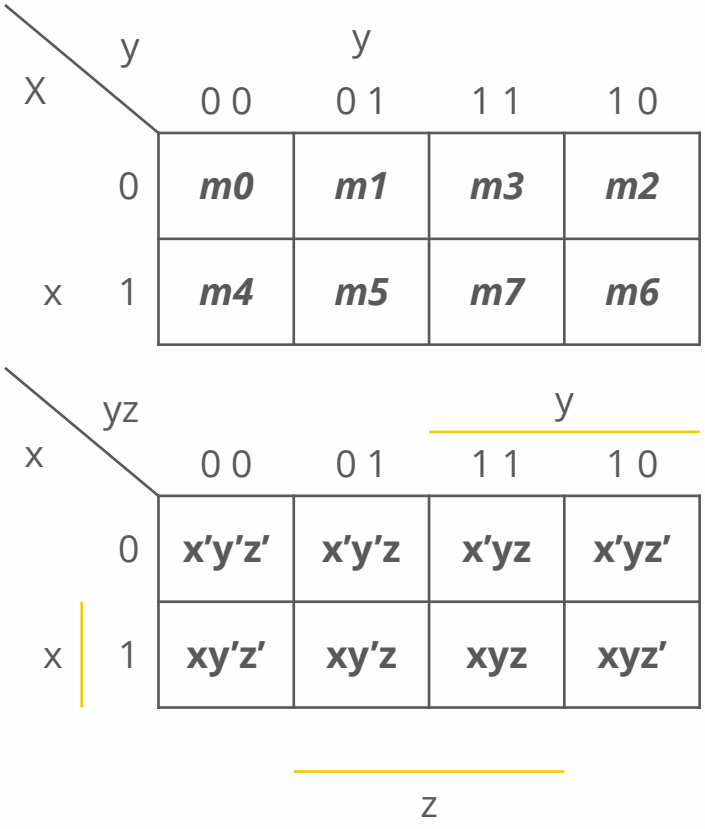
- Any two adjacent squares in the map differ by only one variable
  - Primed in one square and unprimed in the other
  - e.g., m_5 and m_7 can be simplified
  - $m_5 + m_7 = xy'z + xyz = xz\,(y'+y) = xz$





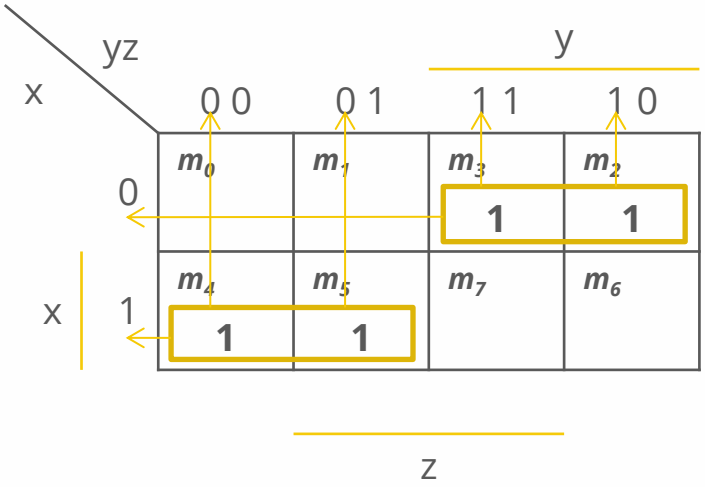
THE MAP METHOD

- m_0 and m_2 (m_4 and m_6) are adjacent
- $m_0 + m_2 = x'y'z' + x'yz' = x'z'(y'+y) = x'z'$
- $m_4 + m_6 = xy'z' + xyz' = xz'(y'+y) = xz'$



THE MAP METHOD

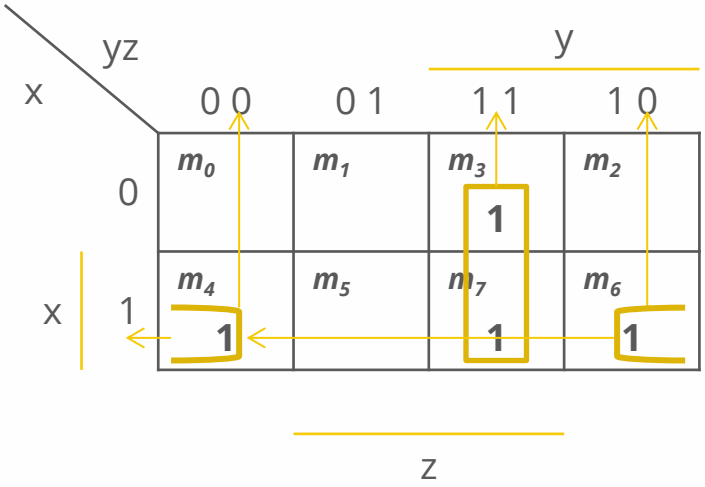
- Example 3.1: Simplify the Boolean function F(*x*, *y*, *z*) = Σ(2, 3, 4, 5)



F = **x'y + xy'**

THE MAP METHOD

- Example 3.2: Simplify the Boolean function F(*x*, *y*, *z*) = Σ(3, 4, 6, 7)



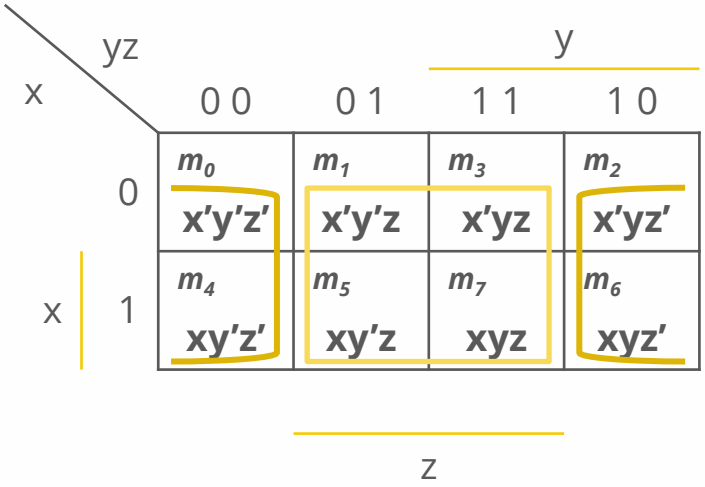
F = **yz + xz'**

THE MAP METHOD

- Consider four adjacent squares in the three-variable map.
- Any such combination represents the logical sum of four minterms and results in an expression of only one literal.
- The number of adjacent squares that may be combined
  - power of two
  - 1,2,4 and 8.
- Larger number of adjacent squares
  - Product term with fewer literal

THE MAP METHOD

- $m_0+m_2+m_4+m_6$ = **x‘ y‘ z‘ + x‘ y z‘ + x y‘ z‘ + x y z'**
  = x'z'(y'+y) +xz'(y'+y) = x‘ z' + x z‘ = **z'**
- $m_1+m_3+m_5+m_7$ = **x‘ y‘ z + x‘ y z + x y‘ z + x y z**
  =x‘ z (y‘ + y) + x z (y‘ + y) = x‘ z + x z = **z**

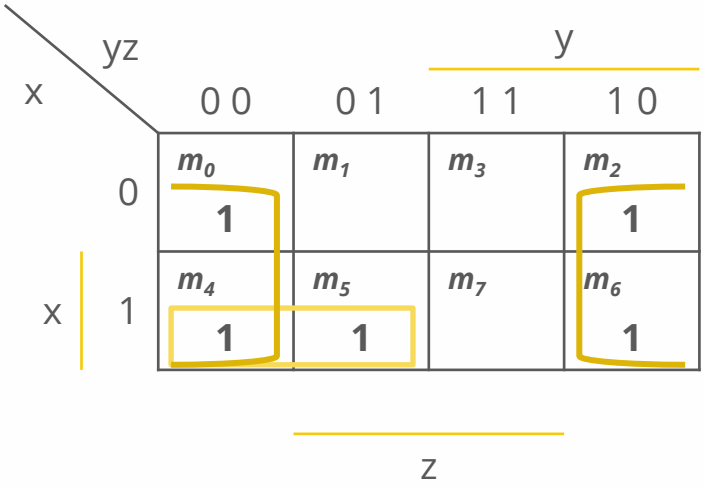


THE MAP METHOD

- **1** square represents **1** minterm
  - A Term of **3** literals.
- **2** adjacent squares
  - A term of **2** literals.
- **4** adjacent squares
  - A term of **1** literal.
- **8** adjacent squares
  - Entire map
  - Function **F = 1**

THE MAP METHOD

- Example 3.2: Simplify the Boolean function F(x, y, z) = Σ(0, 2, 4, 5, 6)



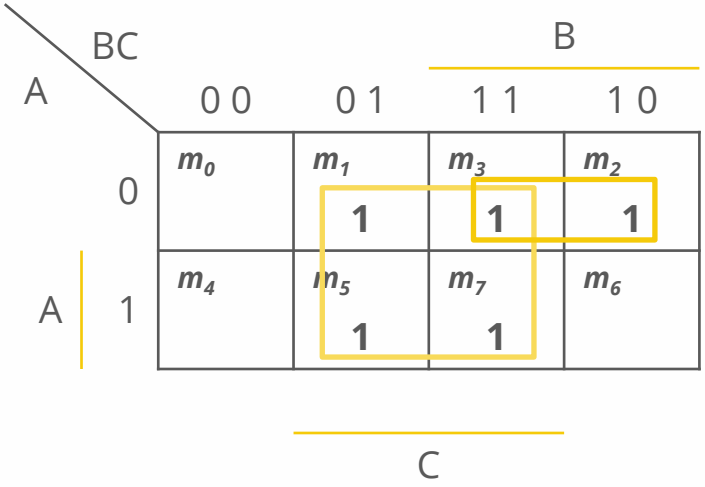
F = **z’ + xy’**

THE MAP METHOD

- If a funtion is not expressed in sum of minterms
  - Use the map to obtain the minterms
  - Simplify the function and find the minimum number of terms.
  - Make sure that the algebraic expression is in sum of products form.

THE MAP METHOD

- Example 3.2: Simplify the Boolean function F = A' C + A' B + A B' C + B C.
  a. Express it in sum of minterms
  b. And find the minimal sum of products expression.



F = **C + A'B**



3.2 FOUR – VARIABLE
K-MAP

FOUR-VARIABLE MAP

- 16 minterms
- Combinations of 2, 4, 8, and 16 adjacent squares

wx \ yz	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

y, x, w, z

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

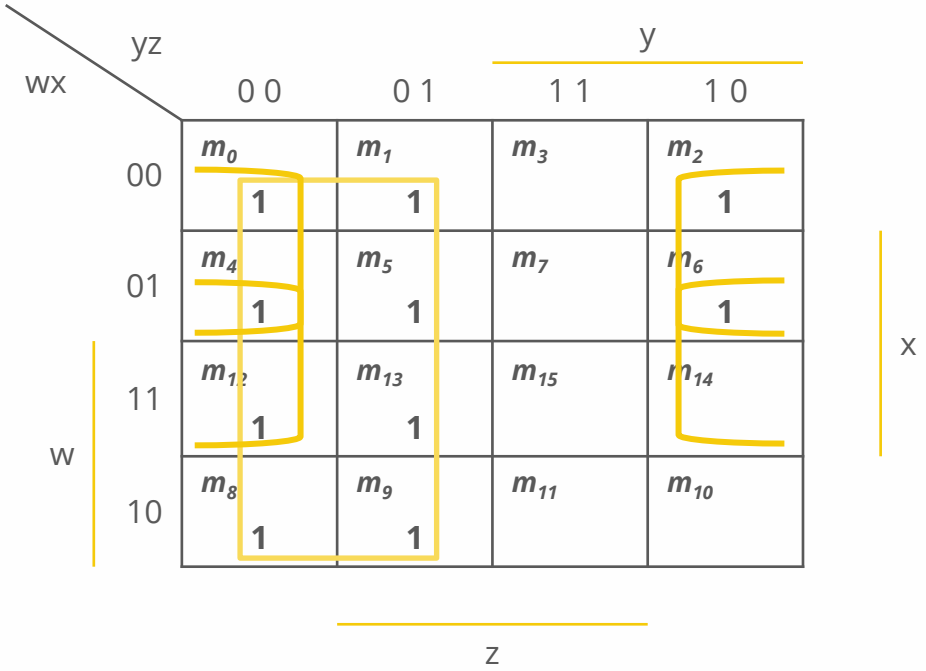
y, x, w, z

FOUR-VARIABLE MAP

- Minimization of four-variable Boolean function is similar to three-variable functions.
- Adjacent squares are defined to be squares next to each other.
  - Ex: m_0 and m_2, m_3 and m_{11}.
- **1** square represents **1** minterm
  - A Term of **4** literals.
- **2** adjacent squares
  - A term of **3** literals.
- **4** adjacent squares
  - A term of **2** literal.
- **8** adjacent squares
  - A term of **1** literal.
- **16** adjacent squares
  - Entire map
  - Function **F = 1**

FOUR-VARIABLE MAP

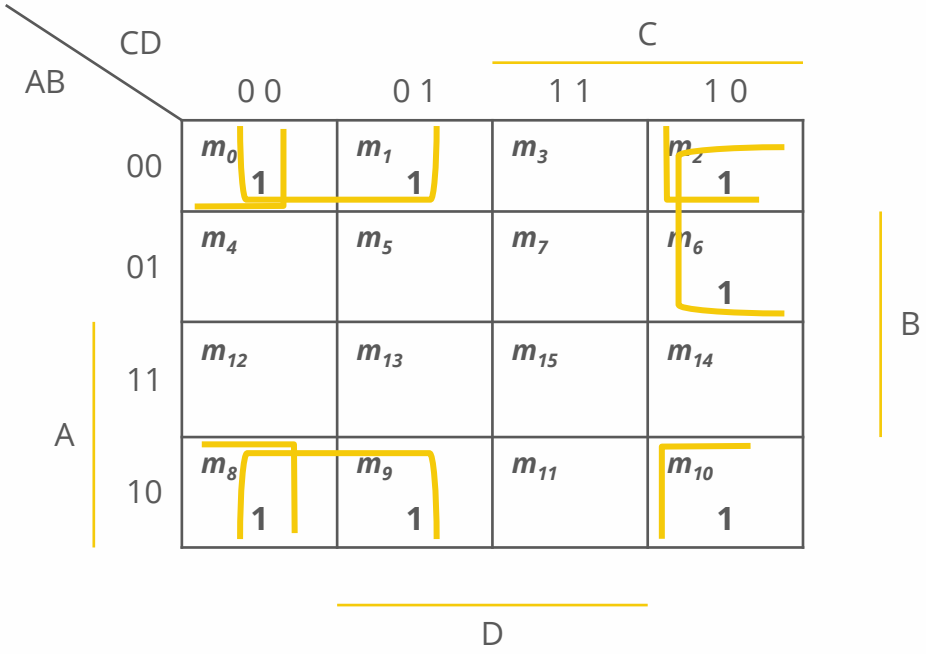
- Example 3.5: simplify $F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$



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

FOUR-VARIABLE MAP

- Example 3-6: simplify F = A′ B′ C′ + B′ C D′ + A′ B C D′ + A B′ C′



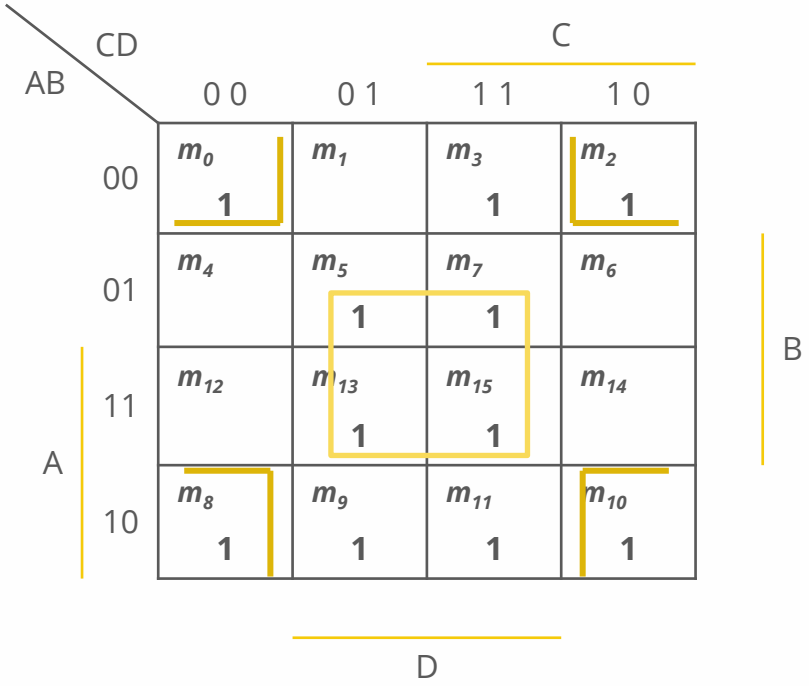
F = **B′ C′ + A′CD′ + B′D′**

FOUR-VARIABLE MAP

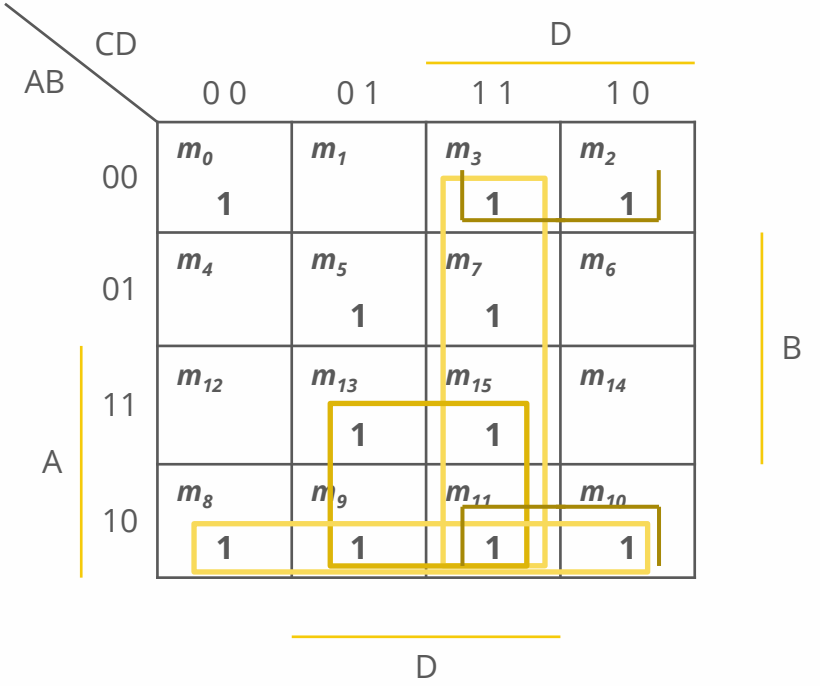
- **Prime implicant** is a product term obtained by combining the maximum possible number of adjacent squares in the map.
- If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be **essential**.

FOUR-VARIABLE MAP

- Simplify F (A, B, C, D) = ∑(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)



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



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

FOUR-VARIABLE MAP

- Essential prime implicants **BD** and **B'D'**.
- Prime implicants **CD**, **B'C**, **AD** and **AB'**.
- Logical sum of two essential prime implicants and any two prime implicants. There are four possible ways that the function can be expressed with four product terms of two literals each:
  - F = BD + B'D' + CD + AD
    = BD + B'D' + CD + AB'
    = BD + B'D' + B'C + AD
    = BD + B'D' + B'C + AB'

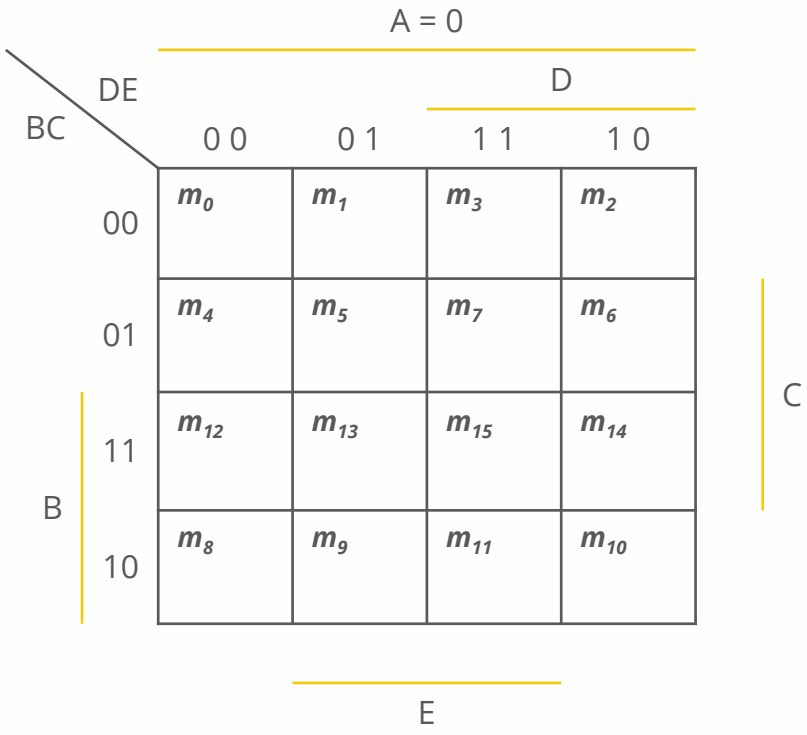


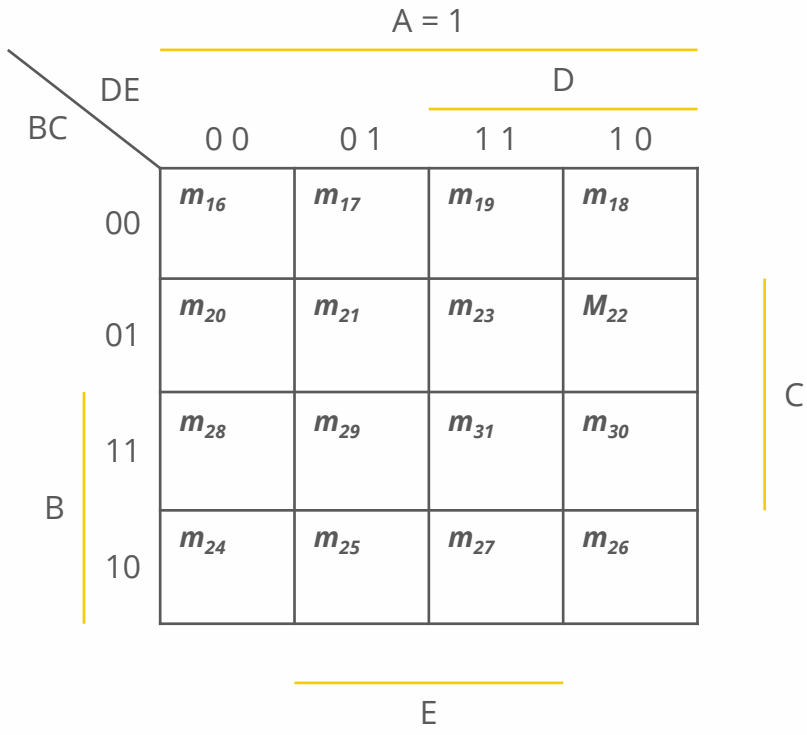
3.3 FIVE – VARIABLE K-MAP

FIVE-VARIABLE MAP

- Maps for more than four variables becomes complicated.
- A five-variable map needs 32 squares and a six-variable map needs 64 squares.
- When the number of variables becomes large,
- The number of squares becomes excessively large,
- The geometry for combining adjacent squares becomes more involved.
- Five-variable map: two four-variable map (one on the top of the other).

FIVE-VARIABLE MAP





FIVE-VARIABLE MAP

- Each four-variable map retains the previously defined adjacency when taken separately.
- Each square in the A = 0 map is a adjacent to the corresponding square in the A = 1.
  - Ex: $\mathbf{m_4}$ is adjacent to $\mathbf{m_{20}}$ and $\mathbf{m_{15}}$ to $\mathbf{m_{31}}$.
- For six-variable map 4 x four-variable maps needed to obtain the required 64 squares.
- Variables > 6 need to many squares and are impractical to use.

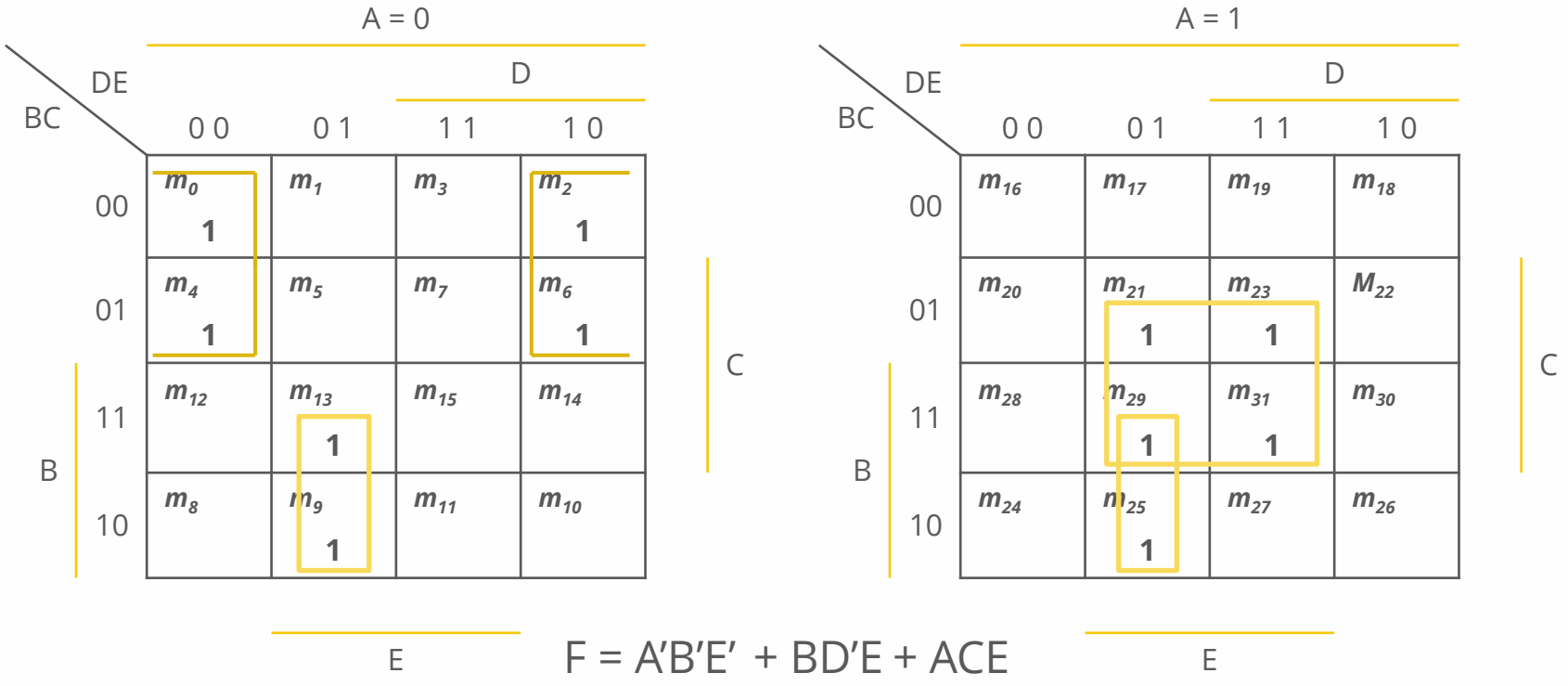
FIVE-VARIABLE MAP

K	Number of Adjacent Squares 2^k	Number of Literals in a Term in an n-variable Map			
		n = 2	n = 3	n = 4	n = 5
0	1	2	3	4	5
1	2	1	2	3	4
2	4	0	1	2	3
3	8		0	1	2
4	16			0	1
5	32				0

Table 3.1 shows the relationship between the number of adjacent squares and the number of literals in the term.

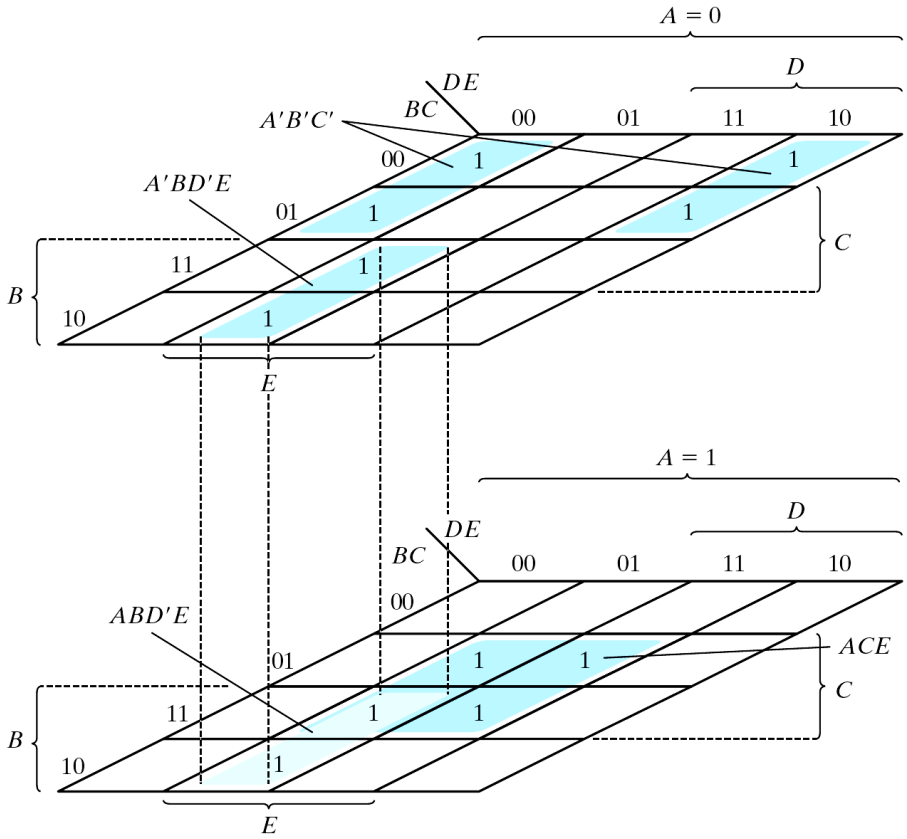
FIVE-VARIABLE MAP

- Example 3.7: simplify $F = \Sigma(0, 2, 4, 6, 9, 13, 21, 23, 25, 29, 31)$



$$F = A'B'E' + BD'E + ACE$$

FIVE-VARIABLE MAP





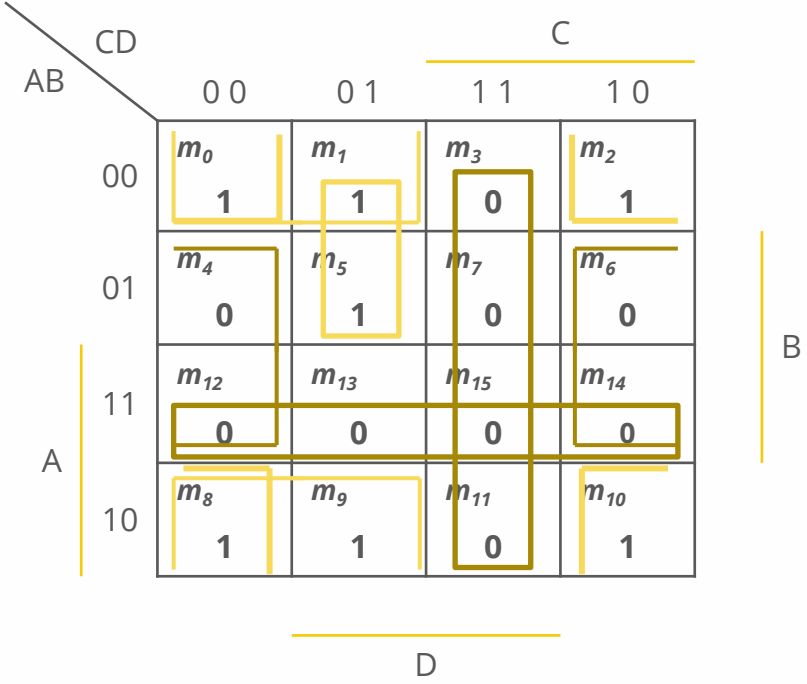
3.4 PRODUCT OF SUMS SIMPLIFICATION

PRODUCT OF SUMS SIMPLIFICATION

- The **1's** placed in the squares of the map represent the **minterms** of the function.
- The **minterms** not included in the function denote the complement of the function.
- Mark the empty squares by **0's**
- Combine them into valid adjacent squares
- Obtain simplified expression of the complement function **F'**.

PRODUCT OF SUMS SIMPLIFICATION

- Simplify the following function in (a) sum of product and (b) product of sums:F (A, B, C, D) = ∑(0,1, 2, 5, 8, 9, 10)



F = B′D′ + B′C′ + A′C′D

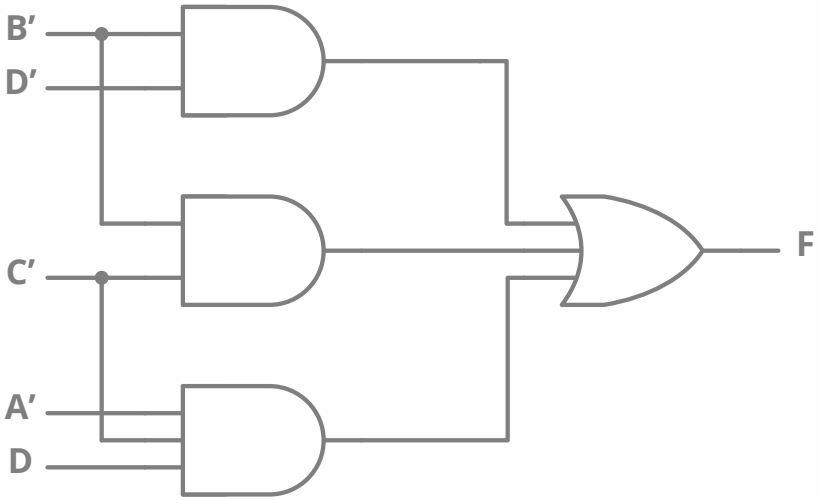
F′ = AB + CD + BD′

Apply DeMorgan's theorem

(F′ = (AB + CD + BD′))′

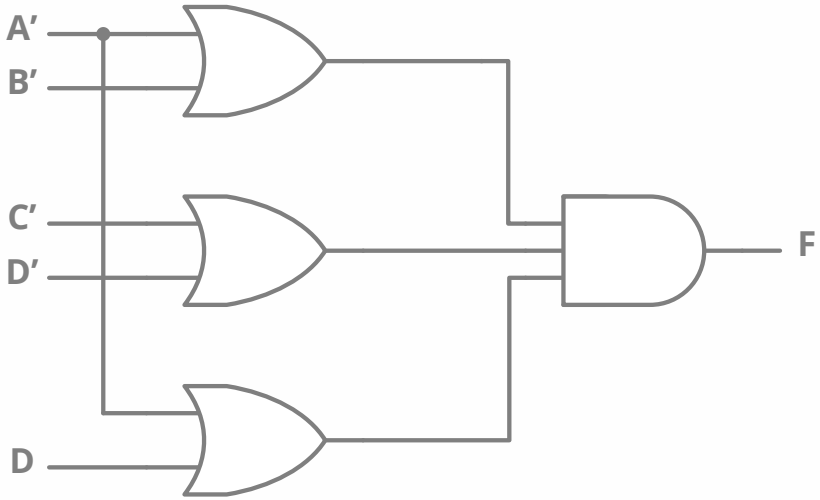
F = (A′ + B′) (C′ + D′) (B′ + D)

PRODUCT OF SUMS SIMPLIFICATION



F = B′D′ + B′C′ + A′C′D

Sum of Products

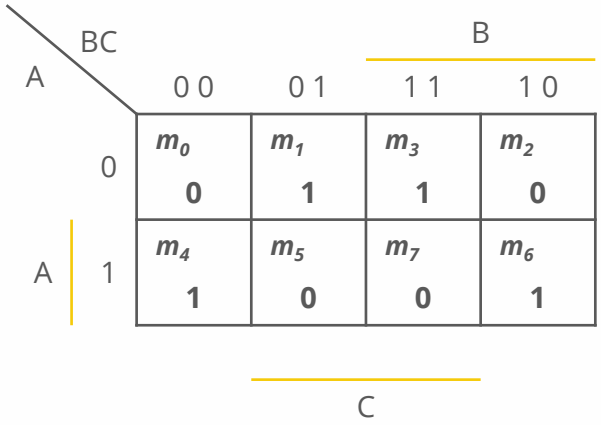


F = (A′ + B′) (C′ + D′) (B′ + D)

Products of Sum

PRODUCT OF SUMS SIMPLIFICATION

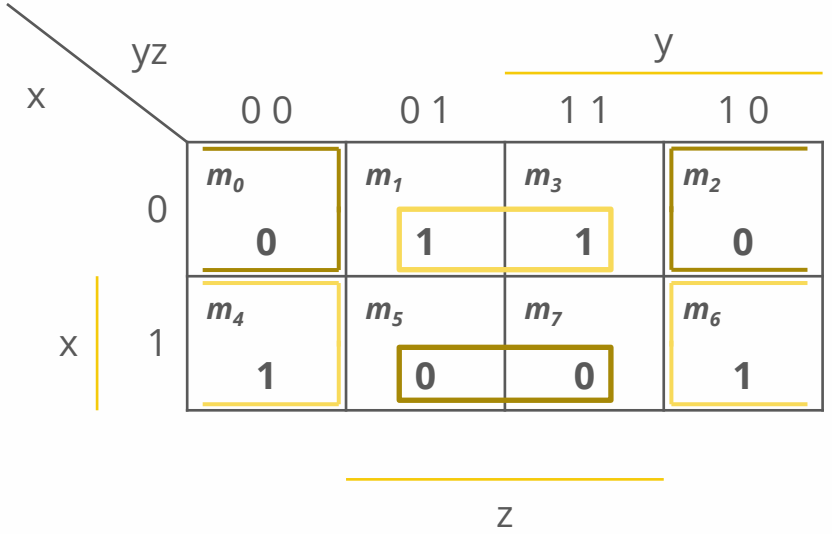
- Consider the function defined in Table 3.2.
- Sum of minterms:
  - F (x, y, z) = ∑(1, 3, 4, 6)
- Product of maxterms:
  - F (x, y, z) = ∏(0,2,5,7)



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

PRODUCT OF SUMS SIMPLIFICATION

- Sum of minterms:
  - F (x, y, z) = $\sum$(1, 3, 4, 6)
  - Sum of products: x'z + xz'
- Product of maxterms:
  - F (x, y, z) = $\prod$(0, 2, 5, 7)
  - Product of sums: (x'+z') (x+z)





3.5 DON’T CARE CONDITIONS

DON'T CARE CONDITIONS

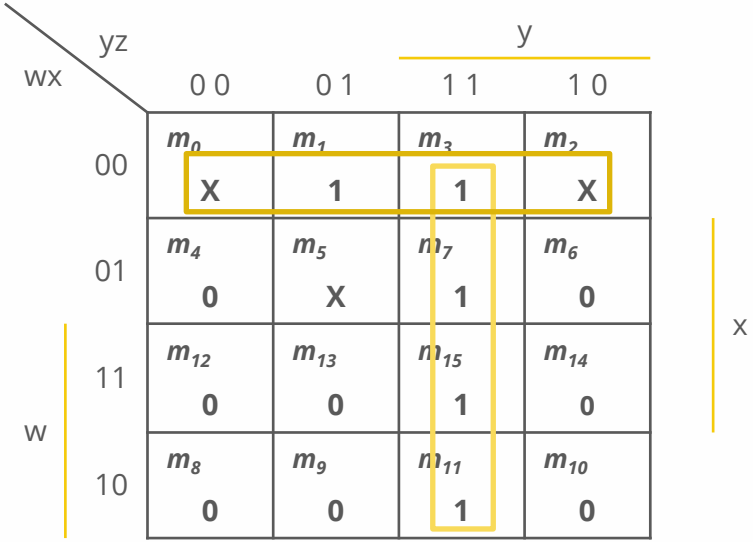
- The value of a function is not specified for certain combinations of variables
  - BCD; 1010-1111: don't care
- These don't care conditions can be used on a map to provide further simplification of the Boolean expression.
- Don't care minterm is a combination of variables whole logical value is not specified.

DON'T CARE CONDITIONS

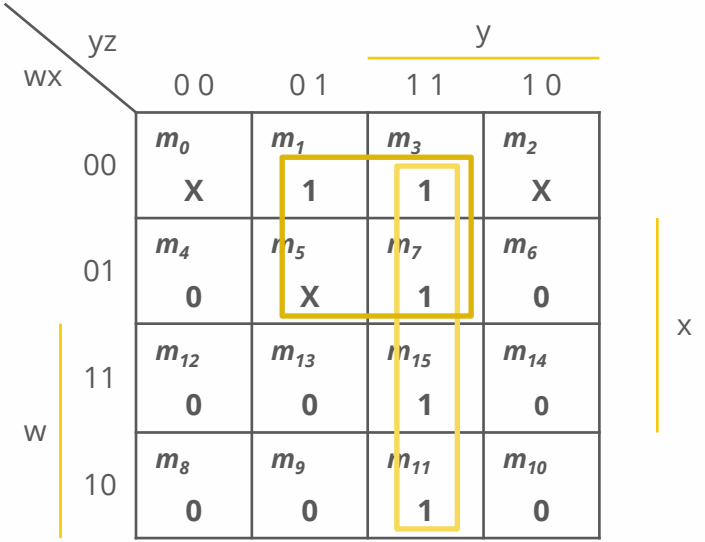
- It cannot be marked with a 1 in the map
  - It would require that the function always be a 1 for such combination.
- It cannot be marked with a 0 in the map
  - It would require that the function always be a 0 for such combination.
- For don't care conditions an **X** is used.
- For adjacent squares in the map to simplify the function
  - The don't care minterms may be assumed to be either 0 or 1.

DON'T CARE CONDITIONS

- Simplify the Boolean function F (w, x, y, z) = ∑(1, 3, 7, 11, 15)
  - Don't care conditions d(w, x, y, z) = ∑(0, 2, 5).



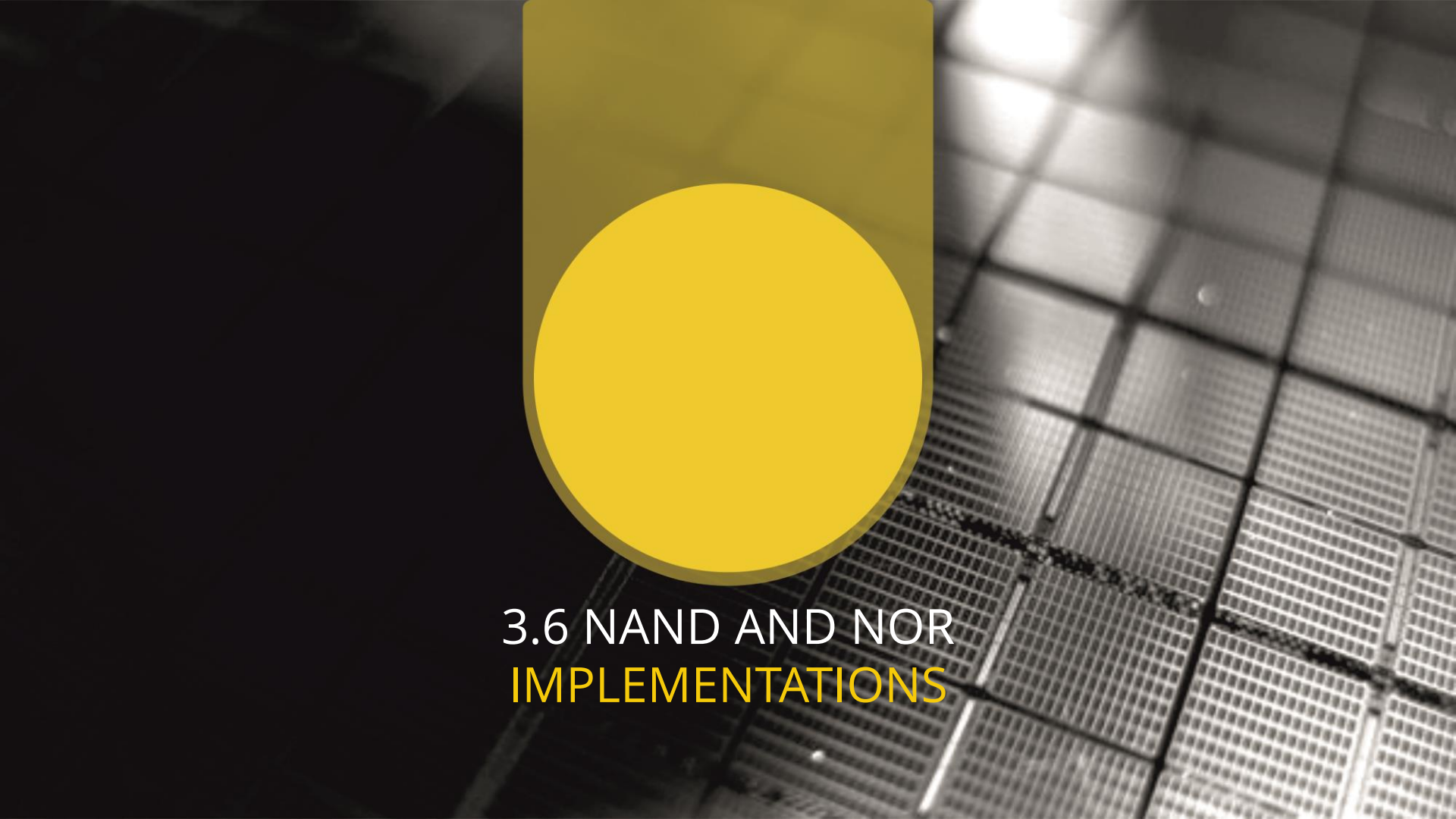
F = yz + w'x'



F = yz + w'z

DON'T CARE CONDITIONS

- Don't care minterms in the map are initially marked with **X's**.
- The choice between 0 and 1 is made depending on the way the incompletely specified function is simplified.
- Once the choice is made,
  - the simplified function obtained will consist of a sum of minterms
  - including those minterms that were initially marked with X's and
  - have been chosen to be included with **1's**.
- $F(w, x, y, z) = yz + w'x' = \sum(0, 1, 2, 3, 7, 11, 15)$
- $F(w, x, y, z) = yz + w'z = \sum(0, 1, 3, 5, 7, 11, 15)$

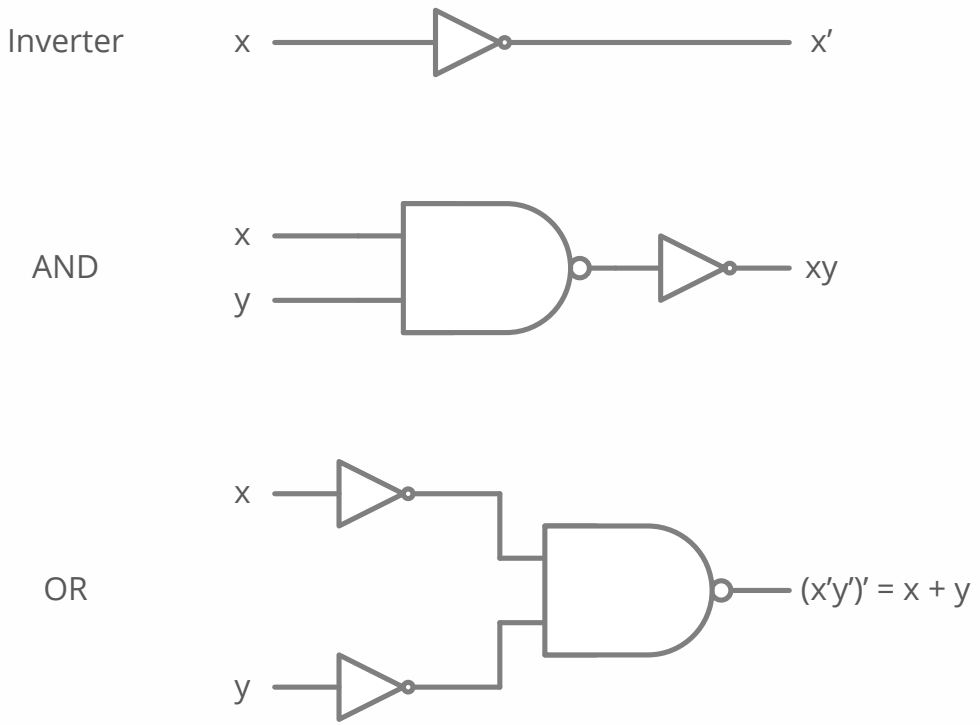


3.6 NAND AND NOR IMPLEMENTATIONS

NAND AND NOR IMPLEMENTATIONS

NAND Circuits:

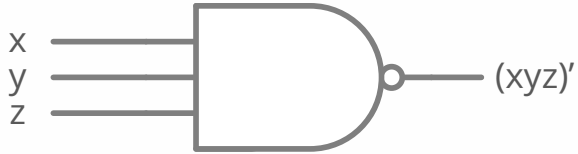
- The NAND gate is a universal gate.
  - Can implement any digital system.
  - Complement operation is obtained from one-input NAND gate.
  - AND operation requires two NAND gates
  - OR operation is achieved through NAND gate with additional inverters in each input



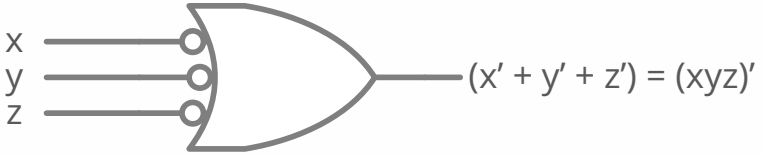
NAND AND NOR IMPLEMENTATIONS

NAND Circuits:

- Two graphic symbols for NAND gate



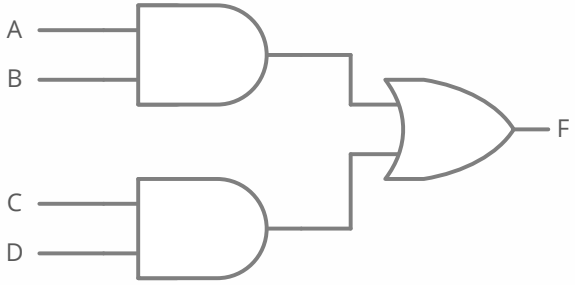
(a) AND-invert



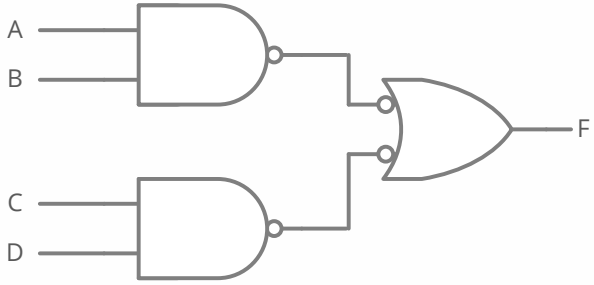
(b) Invert-OR

NAND AND NOR IMPLEMENTATIONS

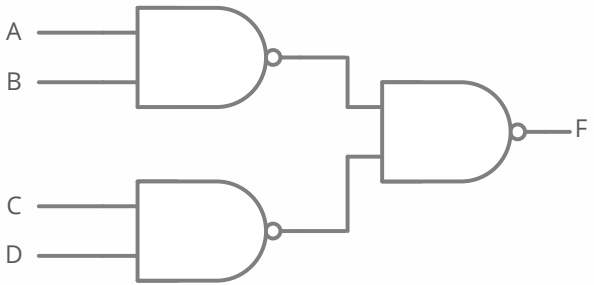
- The implementation of Boolean functions with NAND gates
  - Require that the function be in sum of products form.
  - F = A.B + C.D



(a)



(b)



(c)

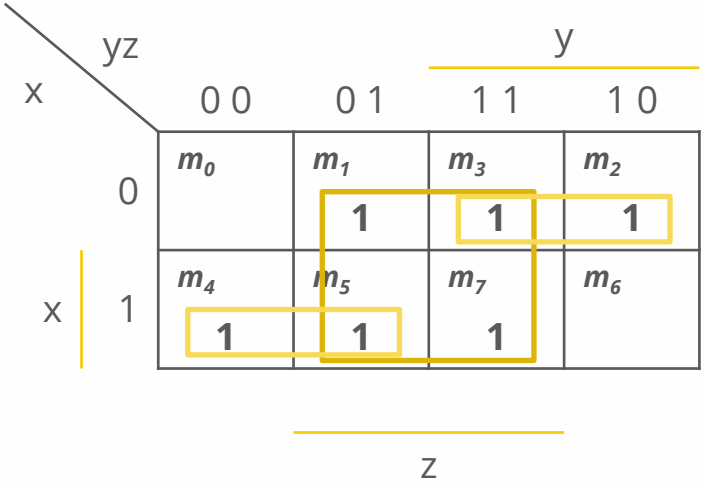
(a) F = A.B +C.D

(b) F = ((A.B)')' + ((C.D)')'
= (A+B)' + (C+D)'
= A.B + C.D

(c) F = ((A.B)' (C.D)')'
= ((A+B) (C + D))'
= A.B + C.D

NAND AND NOR IMPLEMENTATIONS

- Implement the following Boolean function F = (x ,y, z) = (1, 2, 3, 4, 5, 7).
  - Simplify the function in sum of products using K-map.



F = z + xy′+ x′y

NAND AND NOR IMPLEMENTATIONS

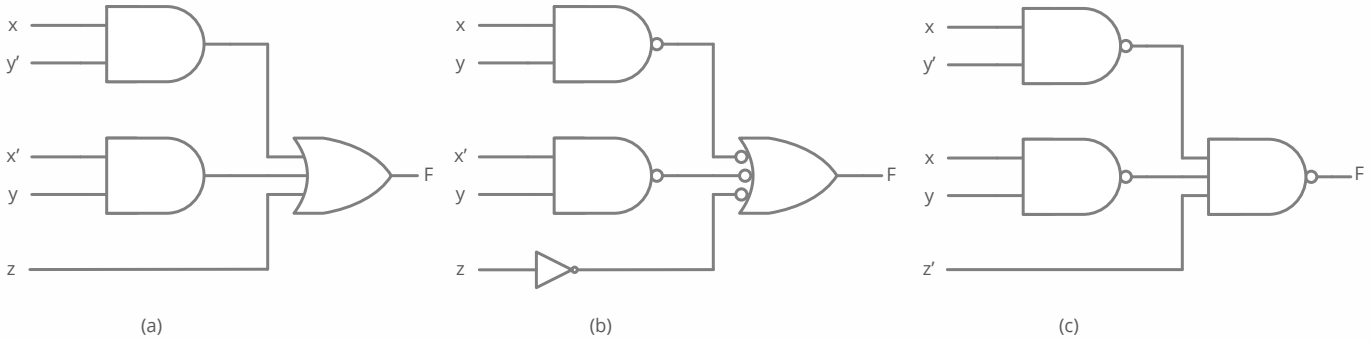
x \ yz	00	01	11	10
0	m_0	m_1 1	m_3 1	m_2 1
1	m_4 1	m_5 1	m_7 1	m_6

y

x

z

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



NAND AND NOR IMPLEMENTATIONS

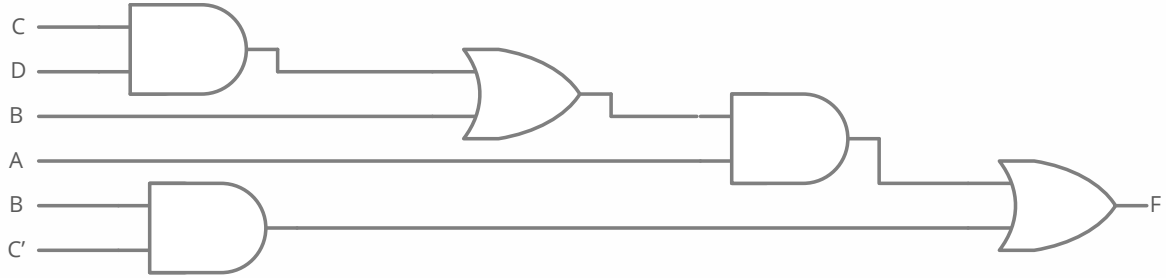
- The procedure
  - Simplify in the form of sum of products;
  - Draw a NAND gate for each product term;
    - The inputs to each NAND gate are the literals of the term (the first level);
  - A single NAND gate for the second sum term (the second level);
  - A term with a single literal requires an inverter in the first level if the single literal is not complemented. Otherwise it can be connected directly.

NAND AND NOR IMPLEMENTATIONS

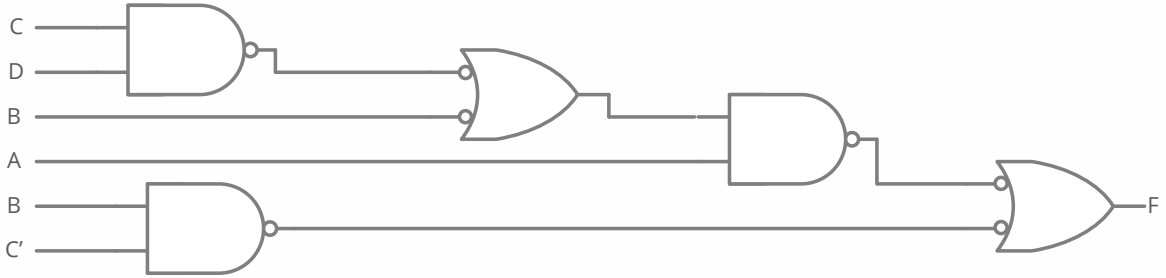
- General procedure for converting multilevel AND-OR diagram into all NAND diagram:
  - Convert all AND gates to NAND gates with AND-invert graphic symbol.
  - Convert all OR gates to NAND gates with invert-OR graphic symbol.
  - Check all the bubbles in the diagram.
    - For every bubble that is not compensated by another small circle along the same line,
    - insert an inverter (one-input NAND gate) or
    - complement the input literal

NAND AND NOR IMPLEMENTATIONS

- F = A (CD + B) + BC’



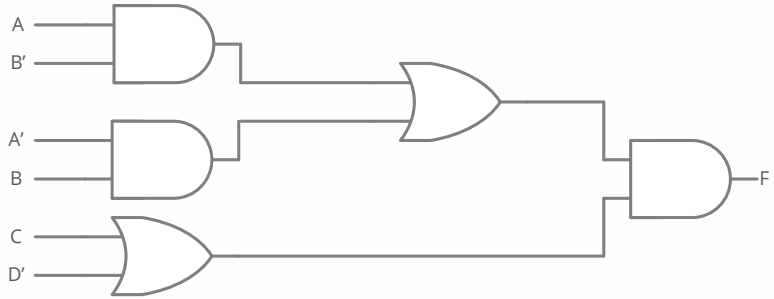
(a) AND-OR gates



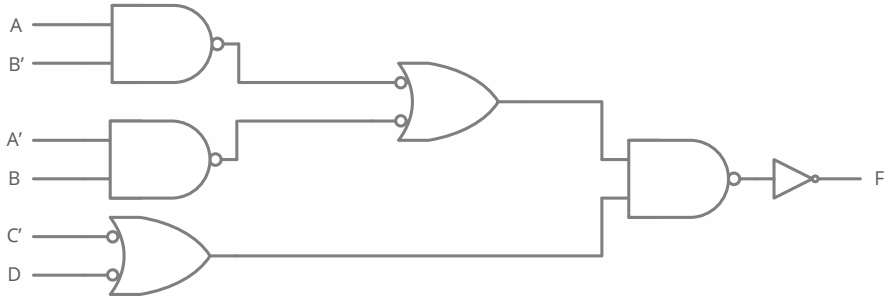
(b) NAND gates

NAND AND NOR IMPLEMENTATIONS

- F = (AB′ + A′B) (C + D′)



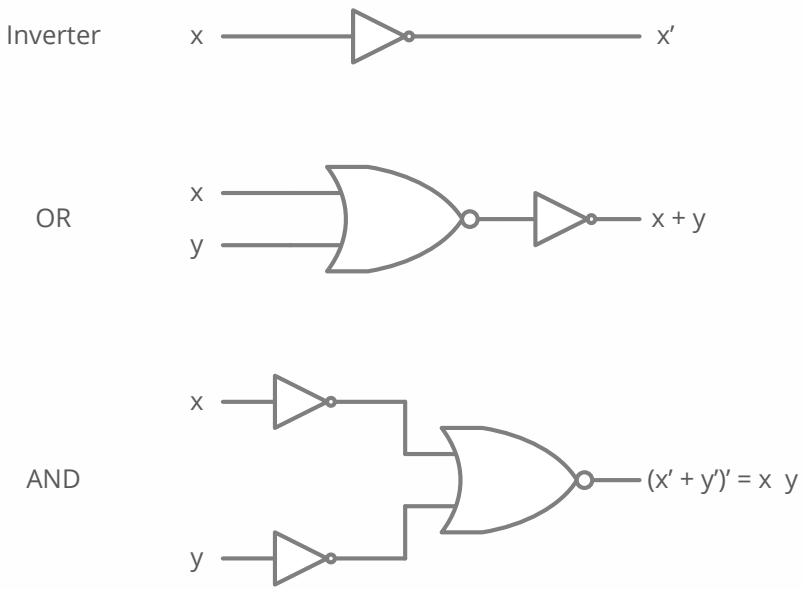
(a) AND-OR gates



(b) NAND gates

NAND AND NOR IMPLEMENTATIONS

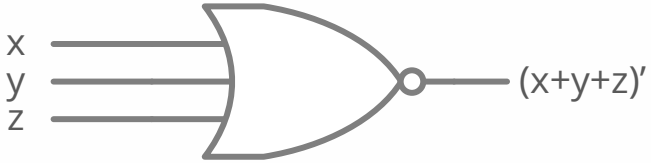
- NOR function is the dual of NAND function.
- The NOR gate is also universal.



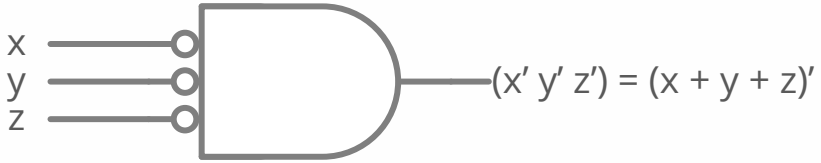
NAND AND NOR IMPLEMENTATIONS

NAND Circuits:

- Two graphic symbols for NAND gate



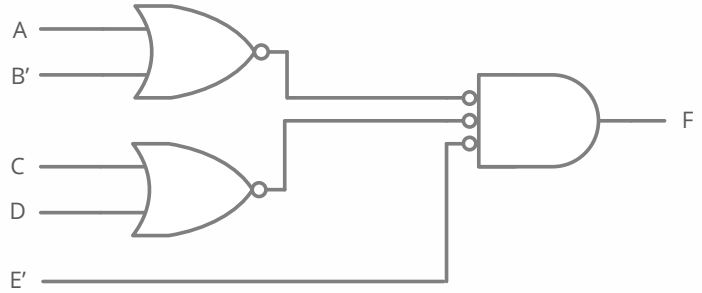
(a) OR-invert



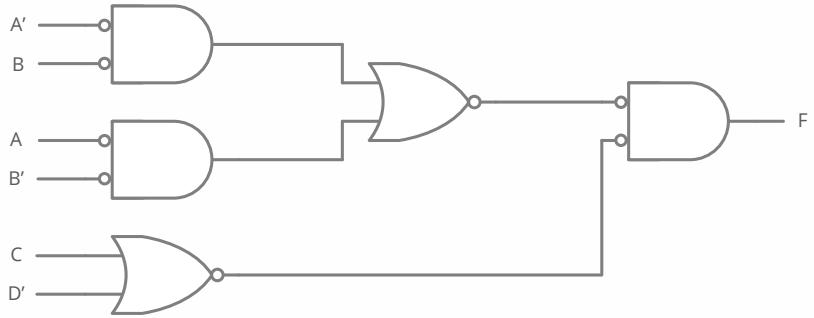
(b) Invert-AND

NAND AND NOR IMPLEMENTATIONS

- Example: Implementing F = (A + B) (C + D) E



- Example: Implementing F = (A B' + A' B) (C + D')





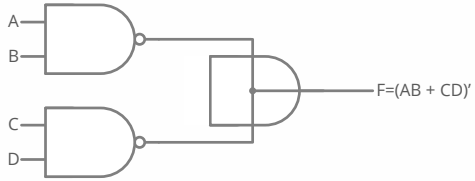
3.7 OTHER TWO LEVEL IMPLEMENTATIONS

OTHER TWO LEVEL IMPLEMENTATIONS

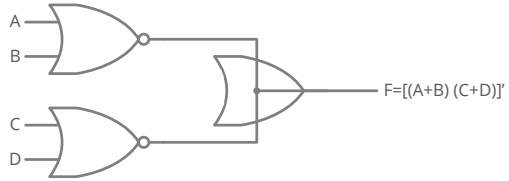
- NAND and NOR gates most often found in integrated circuits.
- NAND and NOR are the most important gates from a practical point of view.
- Some NAND or NOR gates allow the possibility of a wire connection between the outputs of two gates to provide a specific logic function.
- This type of logic is called **wired logic**.
- Example:
  - Open-collector TTL NAND gates when tied together performs wired-AND logic.

OTHER TWO LEVEL IMPLEMENTATIONS

- Example:
  - Open-collector TTL NAND gates when tied together performs wired-AND logic.
  - The wired – AND gate is not a physical gate, but only a symbol to designated the function obtained from the indicated wired connection.
  - F = (AB)' · (CD)' = (AB + CD)'



(a) Wired-AND in open collector TTL
NAND gates
AND-OR-INVERT



(b) Wired-OR in ECL gates
OR-AND-INVERT

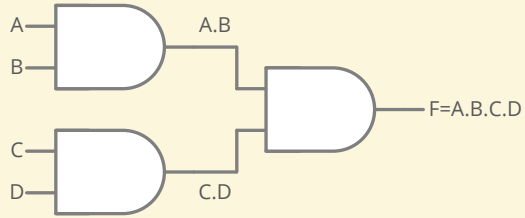
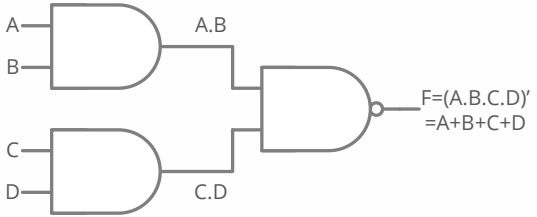
OTHER TWO LEVEL IMPLEMENTATIONS

Nondegenerate Forms

- Consider four types of gate: **AND**, **OR**, **NAND** and **NOR**.
- Assign one type of gate for the first level and one type of gate for the second level.
- There are 16 possible combination of two-level forms.
  - Eight of them: **degenerate forms** = **a single operation**
    - AND-AND, AND-NAND, OR-OR, OR-NOR, NAND-OR, NAND-NOR, NOR-AND, NOR-NAND.

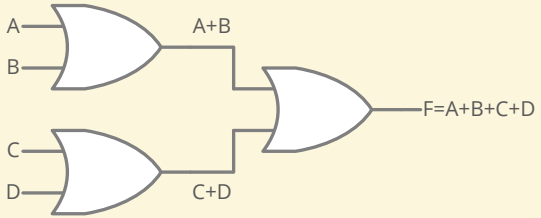
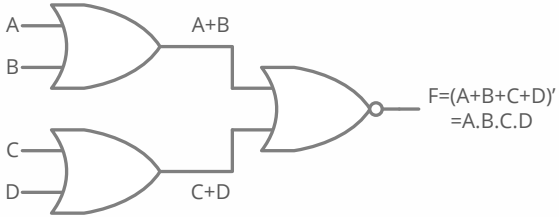
OTHER TWO LEVEL IMPLEMENTATIONS

degenerate forms = **a single operation**

Logic Operation	Circuit Diagram
AND-AND	
AND-NAND	

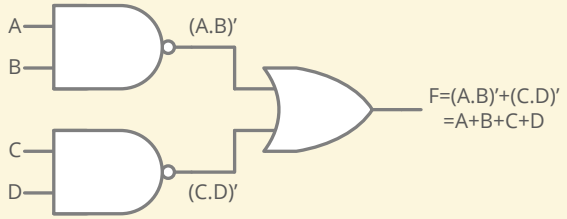
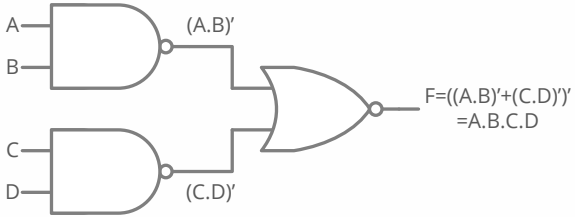
OTHER TWO LEVEL IMPLEMENTATIONS

degenerate forms = **a single operation**

Logic Operation	Circuit Diagram
OR-OR	
OR-NOR	

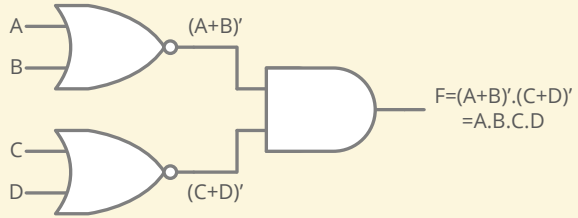
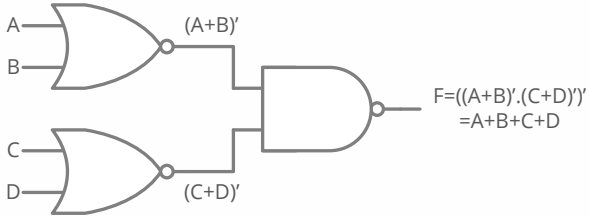
OTHER TWO LEVEL IMPLEMENTATIONS

degenerate forms = **a single operation**

Logic Operation	Circuit Diagram
NAND-OR	
NAND-NOR	

OTHER TWO LEVEL IMPLEMENTATIONS

degenerate forms = **a single operation**

Logic Operation	Circuit Diagram
NOR-AND	
NOR-NAND	

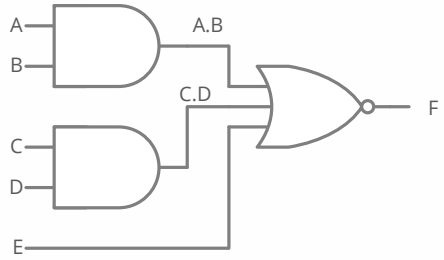
OTHER TWO LEVEL IMPLEMENTATIONS

- The eight **non-degenerate forms**
  - AND-OR, OR-AND, NAND-NAND, NOR-NOR, NOR-OR, NAND-AND, OR-NAND, AND-NOR.
  - **AND-OR** and **NAND-NAND** = **sum of products**.
  - **OR-AND** and **NOR-NOR** = **product of sums.**
  - NOR-OR, NAND-AND, OR-NAND, AND-NOR = ?

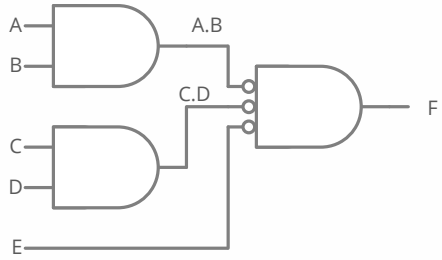
OTHER TWO LEVEL IMPLEMENTATIONS

AND – OR – INVERT Implementation

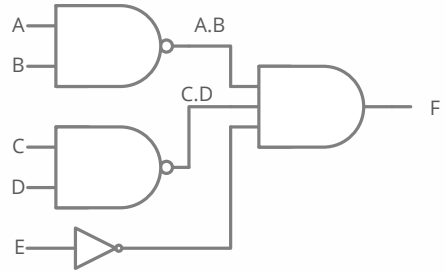
- The two forms **NAND** – **AND** = **AND** – **NOR**
- Both perform the **AND – OR – INVERT** function.
- F = (AB + CD + E)’
- F’ = AB + CD + E (sum of product)



(a) AND-NOR



(b) AND-NOR

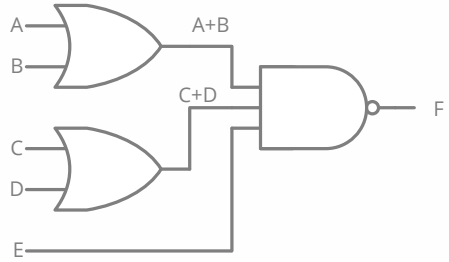


(c) NAND-AND

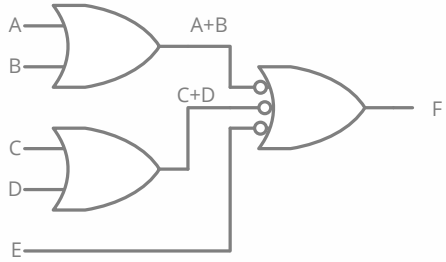
OTHER TWO LEVEL IMPLEMENTATIONS

OR – AND – INVERT Implementation

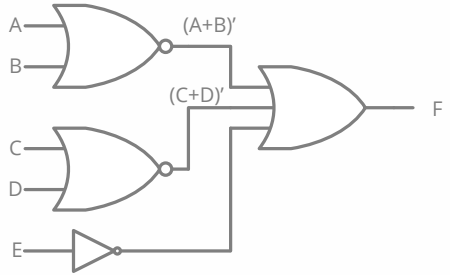
- The two forms **OR** – **NAND** = **NOR** – **OR**
- Both perform the **OR – AND – INVERT** function.
- F = [(A+B) (C+D) E]’
- F’ = (A+B) (C+D) E (product of sum)



(a) OR-NAND



(b) OR-NAND



(c) NOR-OR

OTHER TWO LEVEL IMPLEMENTATIONS

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

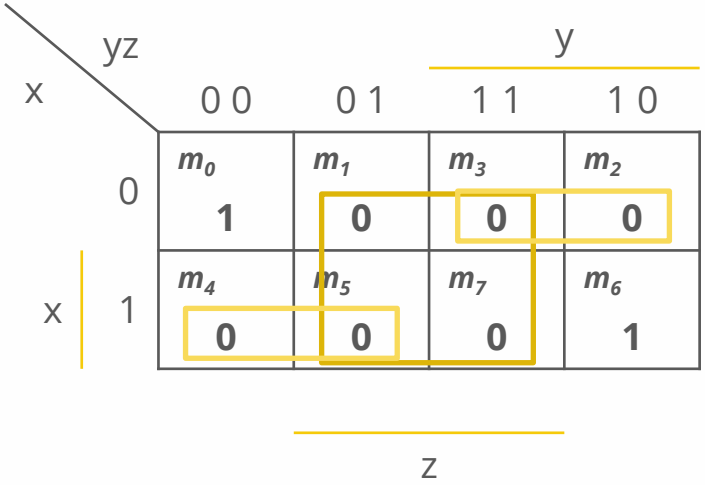
*Form (b) requires and inverter for a single literal term.

OTHER TWO LEVEL IMPLEMENTATIONS

- Example 3-11: F' = x'y+xy'+z

  (F': sum of products)

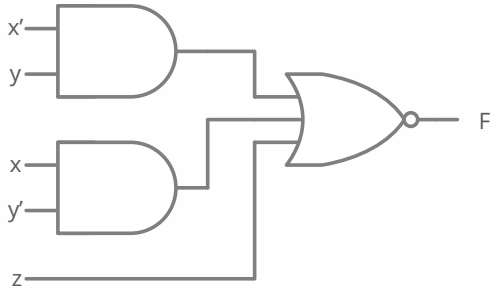
  - Step1: x′y (z + z′) = x′yz + x′yz′
  - Step2: xy′(z + z′) = xy′z + xy′z′
  - Step3: z (x + x′) = xz + x′z
  - Step4: xz+x′z (y+y′) = xyz + x′yz + xy′z + x′y′z
  - Step5: x′yz + x′yz′ + xy′z + xy′z′+ xyz + x′y′z



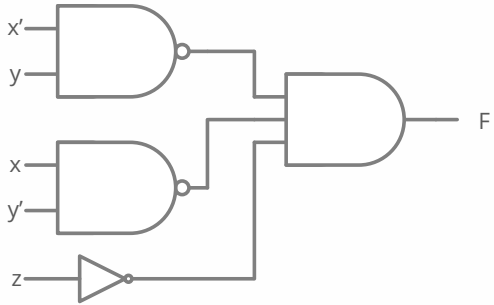
F′ = z + xy′ + x′y

OTHER TWO LEVEL IMPLEMENTATIONS

- Example 3-11: F' = x'y+xy'+z (F': sum of products)



AND-NOR



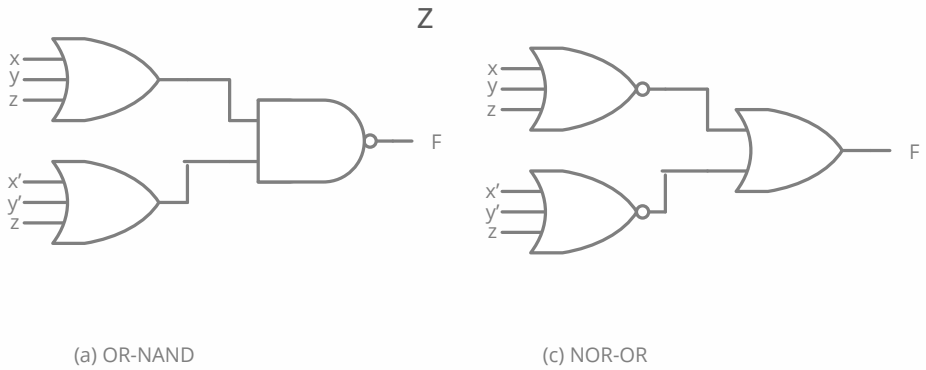
NAND-NAND

- AND-OR-INVERT

OTHER TWO LEVEL IMPLEMENTATIONS

- AND-OR-INVERT form require a simplified expression of the complement of the function in product of sums.
  - Combine the 1's in the map
    - F = x'y'z' + xyz'
  - Take the complement
    - F' = (x+y+z) (x'+y'+z)
    - F = [(x+y+z) (x'+y'+z)]'

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



(a) OR-NAND (c) NOR-OR

OTHER TWO LEVEL IMPLEMENTATIONS

- Summary
  - F' = x'y+xy'+z (F': sum of products)
  - F = (x'y+xy'+z)' (F: AOI implementation)
  - F = x'y'z' + xyz' (F: sum of products)
  - F' = (x+y+z)(x'+y'+z) (F': product of sums)
  - F = ((x+y+z)(x'+y'+z))' (F: OAI)



3.8 EXCLUSIVE-OR FUNCTION

EXCLUSIVE-OR FUNCTION

- Exclusive-OR (XOR)
  - $x \oplus y = x\,y' + x'\,y$
  - F = 1, if only x = 1 or y = 1, but not when x and y = 1.
- Exclusive-NOR (XNOR) (equivalence)
  - $(x \oplus y)' = x\,y + x'\,y'$
  - F = 1, if both x and y = 1 or both = 0.
  - $(x \oplus y)' = (x\,y' + x'\,y)' = (x' + y)\,(x + y') = x'x + x'y' + xy = xy + x'y'$

EXCLUSIVE-OR FUNCTION

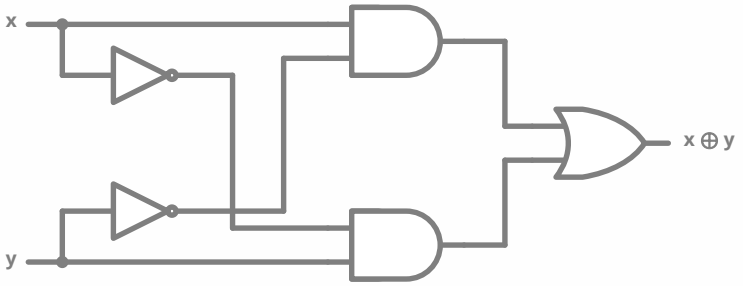
- Some identities
  - $x \oplus 0 = x$
  - $x \oplus 1 = x'$
  - $x \oplus x = 0$
  - $x \oplus x' = 1$
  - $x \oplus y' = (x \oplus y)'$
  - $x' \oplus y = x' \oplus y = (x \oplus y)'$
- Commutative and associative
  - $A \oplus B = B \oplus A$
  - $(A \oplus B) \oplus C = A \oplus (B \oplus C)$

  $= A \oplus B \oplus C$

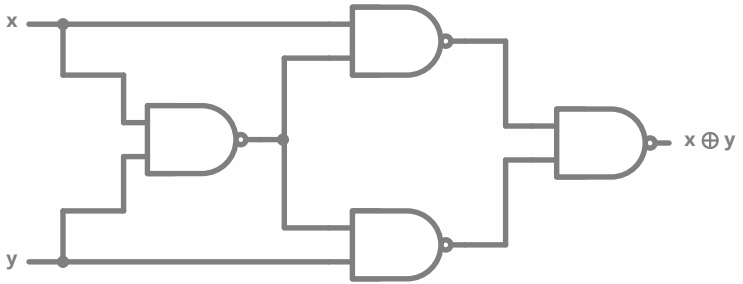
EXCLUSIVE-OR FUNCTION

- Implementations
  - $(x' + y')\, x + (x' + y')\, y$

    $= x\, y' + x'\, y = x \oplus y$



(a) with AND – OR – NOT gates



(b) with NAND gates

EXCLUSIVE-OR FUNCTION

- The XOR operation with three or more variables can be converted into an ordinary Boolean function by replacing ⊕ symbol with its equivalent Boolean expression.
  - $A \oplus B \oplus C = (AB'+A'B)C' + (AB+A'B')C$

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

    $= \Sigma(1, 2, 4, 7)$
  - XOR is a odd function → an odd number of 1's, then $F = 1$.
  - XNOR is a even function → an even number of 1's, then $F = 1$.

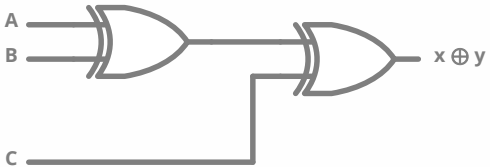
EXCLUSIVE-OR FUNCTION

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

z

(a) Odd Function

$F = A \oplus B \oplus C$



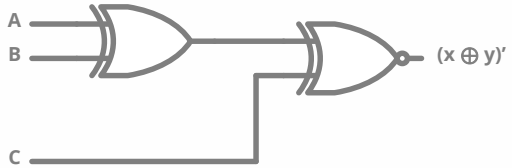
(a) 3-input odd function

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

z

(b) Even Function

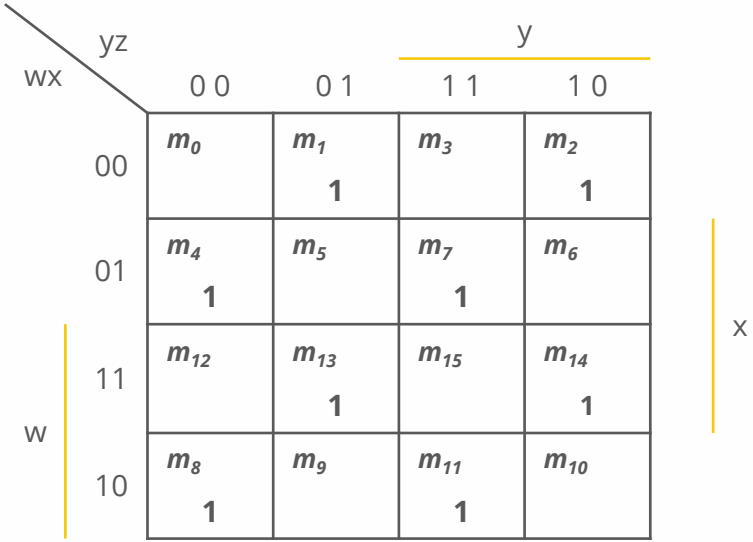
$F = (A \oplus B \oplus C)'$



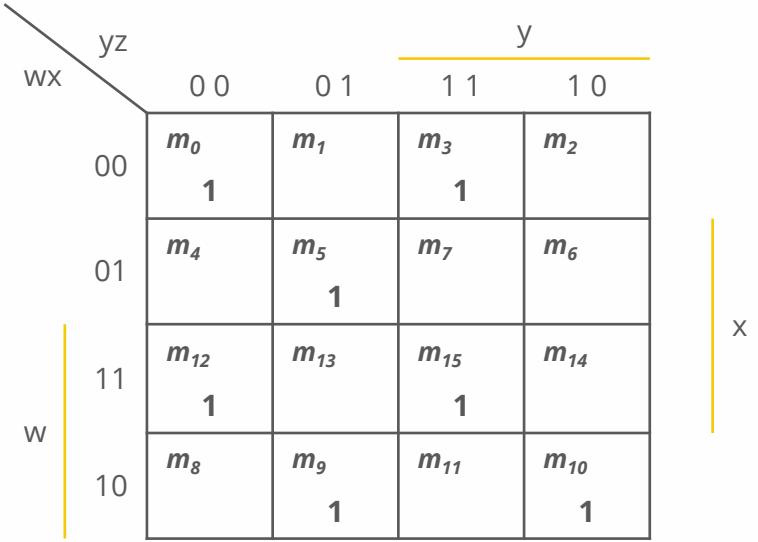
(b) 3-input even function

EXCLUSIVE-OR FUNCTION

- Four-variable Exclusive-OR function
  - A⊕B⊕C⊕D = (AB'+A'B)⊕(CD'+C'D) = (AB'+A'B)(CD+C'D')+(AB+A'B')(CD'+C'D)



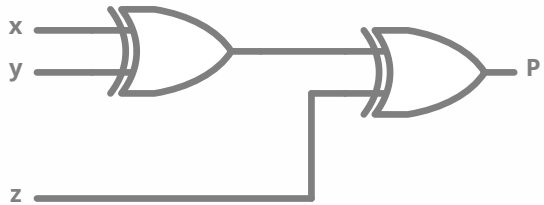
F = A⊕B⊕C⊕D



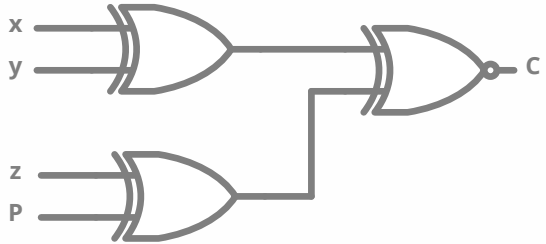
F = (A⊕B⊕C⊕D)'

EXCLUSIVE-OR FUNCTION

- Parity Generation and Checking
  - A parity bit: $P = x \oplus y \oplus z$
  - Parity check: $C = x \oplus y \oplus z \oplus P$
    - C=1: one bit error or an odd number of data bit error
    - C=0: correct or an even # of data bit error



(a) 3-input odd function



(b) 3-input even function

EXCLUSIVE-OR FUNCTION

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

EXCLUSIVE-OR FUNCTION

Three-Bit Message				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

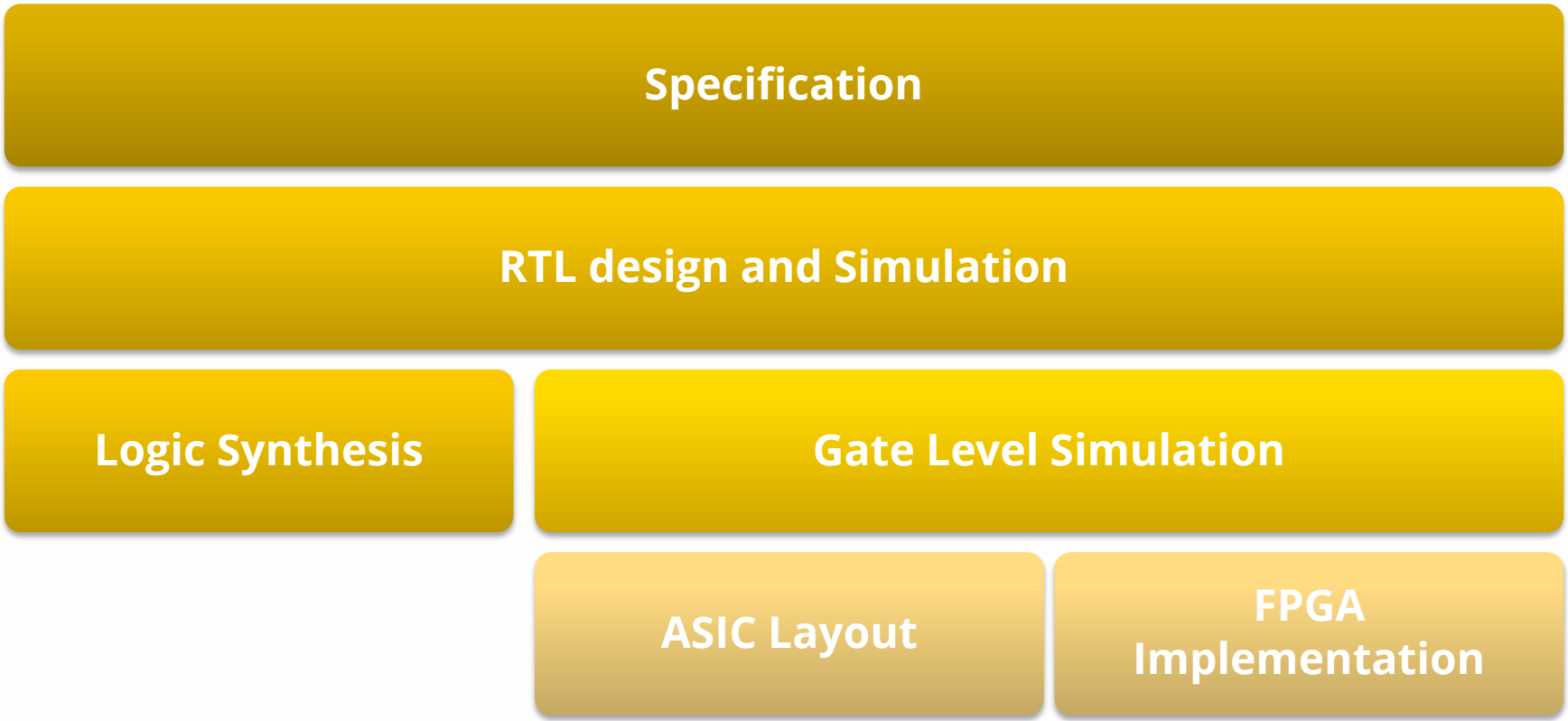


3.9 HARDWARE DESCRIPTION LANGUAGE

HARDWARE DESCRIPTION LANGUAGE

- Describe the design of digital systems in a textual form
  - Hardware structure
  - Function/behavior
  - Timing
- VHDL and Verilog HDL

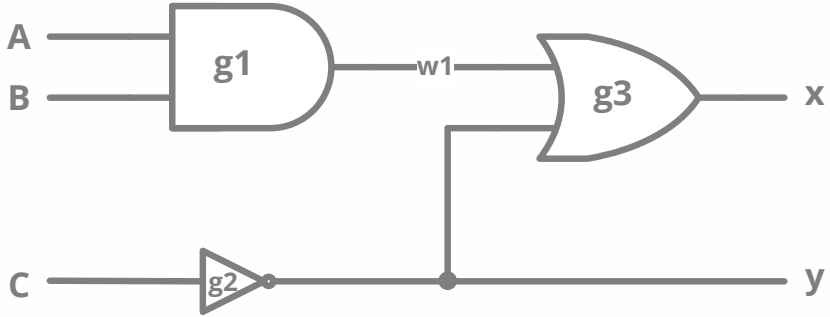
HARDWARE DESCRIPTION LANGUAGE



HARDWARE DESCRIPTION LANGUAGE

- Documentation language
- HDL is used to represent and document digital systems in a form that can be read by both humans and computers
- Examples of keywords:
  - **module, end-module, input, output, wire, and, or,** and **not**

HARDWARE DESCRIPTION LANGUAGE

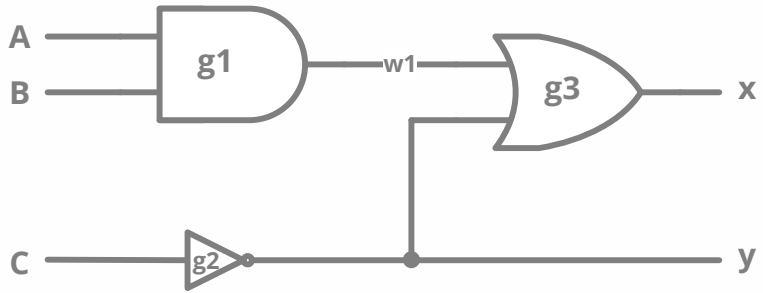



```
//Description of Simple circuit
module smpl_circuit (A, B, C, x, y);
    input A, B, C;
    output x, y;
    wire w1;
    and g1 (e, A, B);
    not g2 (y, C);
    or g3 (x, e, y);
endmodule
```

HARDWARE DESCRIPTION LANGUAGE

- Boolean expressions are specified in Verilog HDL with a
  - continuous assignment statement consisting of the keyword **assign** followed by a Boolean expression.
- To distinguish the arithmetic plus from logical OR, Verilog HDL uses the symbols
  - (&) for AND,
  - (|) for OR,
  - (~) for NOT

HARDWARE DESCRIPTION LANGUAGE




```
//Description of Simple circuit
module smpl_circuit (A, B, C, x, y);
    input A, B, C;
    output x, y;
    assign x = (A & B) | ~C
    assing y = ~C
endmodule
```

- Boolean expression:
  - x = A.B + C′
  - y = C′