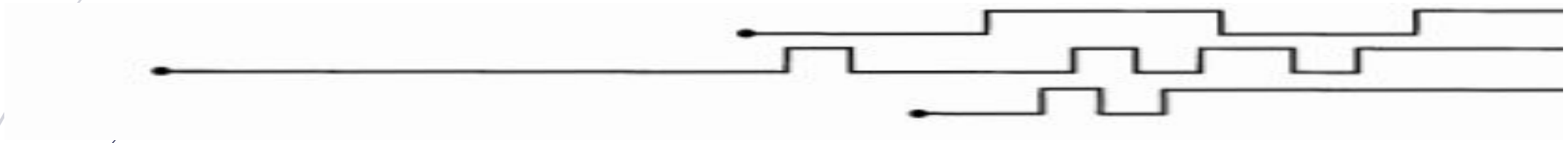
ensia

Digital systems



Chapter 2

Boolean algebra

1

Dr. Iness NEDJI MILAT (Lecturer)
iness.nedji@ensia.edu.dz

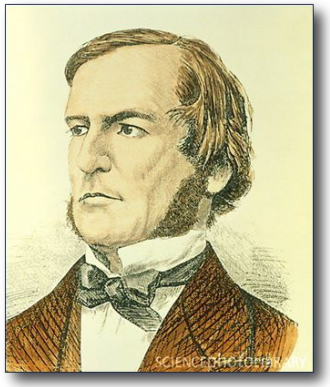
Pr. Nasreddine LAGRAA
Nasredine,lagraa@ensia.edu.dz

First year – 2023/2024

combinational logic

Introduction

- Digital system is **a combination of several circuits** (electronic devices),
- These circuits are designed and their behaviors are analyzed with the use of a mathematical discipline known as **Boolean algebra.**
- Named after George Boole (1815-1864)
  - An English mathematician, who was first to develop and describe a formal system to work with truth values.

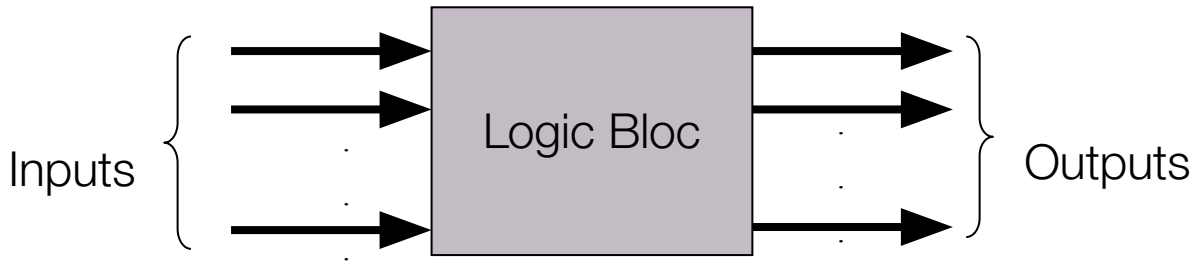


- Boolean logic is a branch of mathematics that deals with rules for manipulating the two logical truth values true and false.

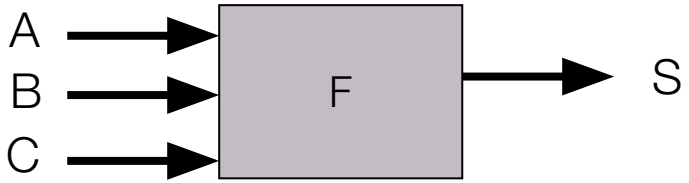
Boolean Algebra

- A digital circuit can be represented by

Set of electronic elements (logic gates)



- Boolean Algebra is the logic mathematics used for understanding of digital system (circuits)
- It is used to describe the **relationship** between inputs and outputs using basically three operators :
  - OR
  - AND
  - NOT (unary)



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

Algebraic expression

Boolean Algebra : Basic operators

- NOT operator
  - Result is 1 only if operand is 0
  - Inversion or negation of the value
  - Notation: $\overline{A}$

0	1
1	0

- AND operator
  - result is 1 only if both operands (variables) is 1
  - also known as logical product
  - notation: $A \cdot B$

0	0	0
0	1	0
1	0	0
1	1	1

- OR operator
  - result is 1 if either of the operands (variables) is 1
  - also known as logical sum
  - notation: $A + B$

0	0	0
0	1	1
1	0	1
1	1	1

Boolean Algebra: Useful Laws

1. *Operations with 0 and 1:*

$X + 0 = X$ $\qquad$ $X \bullet 1 = X$

$X + 1 = 1$ $\qquad$ $X \bullet 0 = 0$

2. *Idempotent Law:*

$X + X = X$ $\qquad$ $X \bullet X = X$

3. *Involution Law:*

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

4. *Complementarity Law :*

$X + \overline{X} = 1$ $\qquad$ $X \bullet \overline{X} = 0$

5. *Commutative Law:*

$X + Y = Y + X$ $\qquad$ $X \bullet Y = Y \bullet X$

Boolean Algebra: Useful Laws

6. *Associative Law :*

$(X + Y) + Z = X + (Y + Z) = X + Y + Z$

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

7. *Distributive Law :*

$X \bullet (Y + Z) = (X \bullet Y) + (X \bullet Z)$

$X + (Y \bullet Z) = (X + Y) \bullet (X + Z)$

8. *Simplification Theorems :*

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

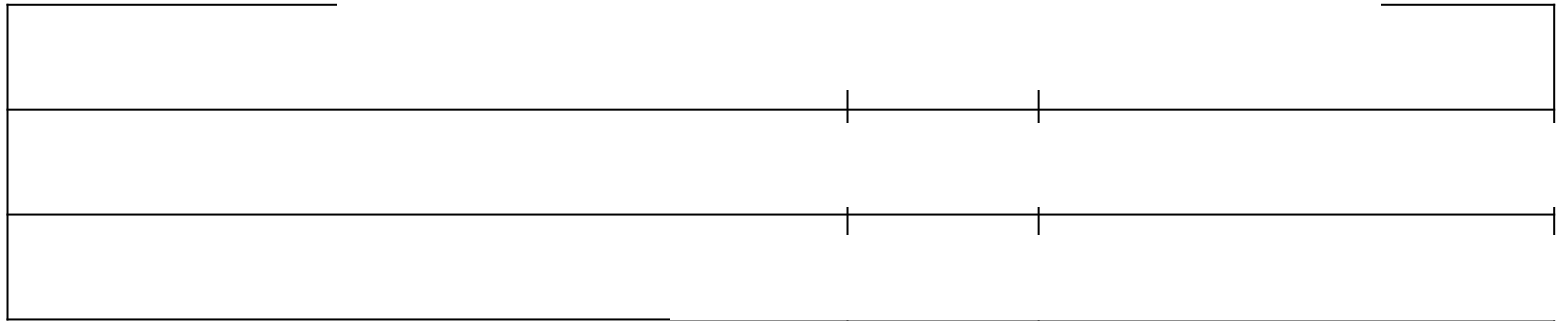
- $(X + Y) \bullet (X + \overline{Y}) = X$ **Expansion**
- $X \bullet (X + Y) = X$ **Absorption**
- $(X \bullet \overline{Y}) + Y = X + Y$

Actually worth remembering — they show up a lot in real designs…

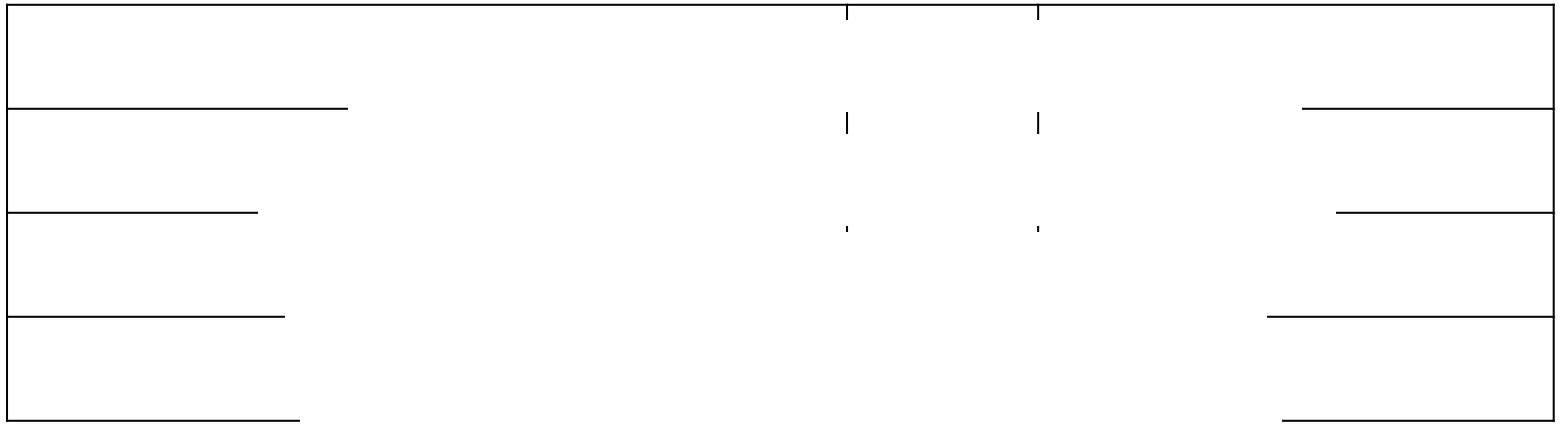
Useful for simplifying expressions

Boolean Algebra : Proving Theorems

EX: Prove the theorem: $X \bullet Y + X \bullet \overline{Y} = X$



EX2: Prove the theorem: $X + X \bullet Y = X$



DeMorgan's Law: Enabling Transformations

DeMorgan's Law :

$$\overline{(X + Y + Z + \cdots)} = \overline{X}.\overline{Y}.\overline{Z}.\ldots$$

$$\overline{(X.Y.Z.\ldots)} = \overline{X} + \overline{Y} + \overline{Z} + \quad \ldots$$

- Think of this as a transformation
  - Let's say we have:

    F = A + B + C

  - Applying DeMorgan's Law, gives us

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

DeMorgan's Law

- These are conversions between different types of logic functions
- They can prove useful if you do not have every type of gate

$$A = \overline{(X + Y)} = \overline{X} \bullet \overline{Y}$$

NOR is equivalent to AND with inputs complemented

0	0	1	1	1	1
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	0

$$B = \overline{(X \bullet Y)} = \overline{X} + \overline{Y}$$

NAND is equivalent to OR with inputs complemented

0	0	1	1	1	1
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	0	0	0

Abstract vs. Implementation

- Computers are built around the idea of two states :
  - true and false
  - on and off
  - 1 and 0
- Computers, as we know them today, are implementations of Boole's *Laws of Thought*
- Boolean Algebra is used for **understanding** or **designing** electronic circuits by describing the interconnections between their inputs and outputs.
- These electronic circuits (called logic circuits)
  - use voltages to represent logical values (1 or 0)
  - perform Boolean operations on Boolean variables .

Function/Behaviour

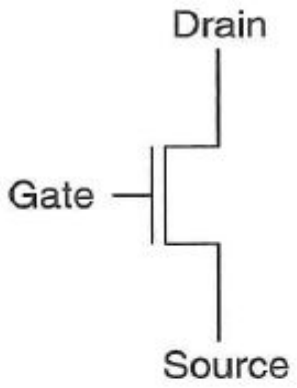
Transistors

- Problem: How to achieve a hardware implementation of binary logic?
- Solution: Use Transistors

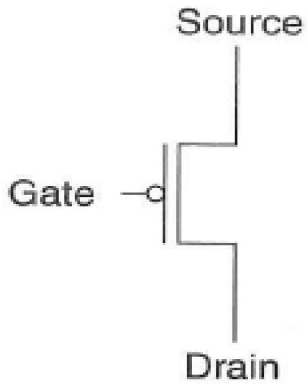
- Computers are built from very large numbers of very small (and relatively simple) structures: transistors
  - Intel's Pentium IV microprocessor, 2000, was made up of more than 42 million MOS transistors
  - Apple's M1 Max, offered for sale in 2021, is made up of more than 56 billion MOS transistors

Different Types of MOS Transistors

- MOS Transistor : Metal Oxide Semi-conductor transistor
- There are two types of MOS transistors: n-type and p-type



n-type



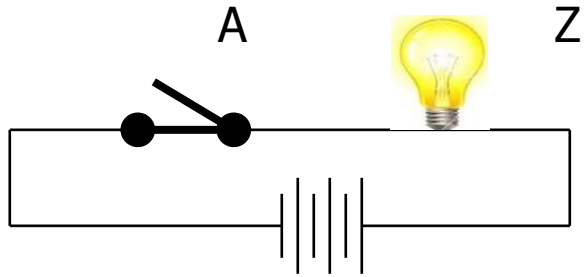
p-type

- They both operate "logically," very similar to the way wall switches work

How Does a Transistor Work?

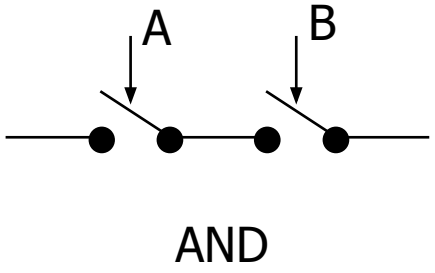
Implementation using switches

- The lamp can be turned on and off by simply manipulating the switch to make or break the closed circuit



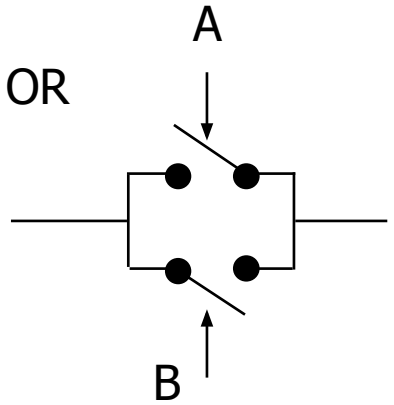
Z ≡ A

- Compose switches into more complex ones (Boolean functions):



Z ≡ A and B =

A.B



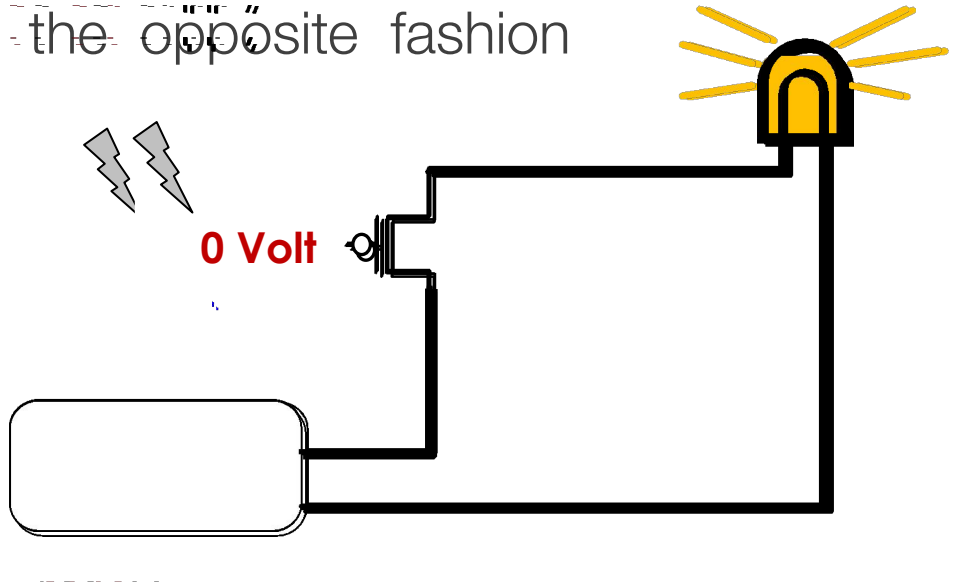
Z ≡ A or B =

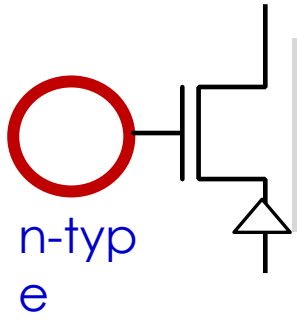
A+B

- Instead of switch, we could use MOS transistor to make or break the closed circuit.

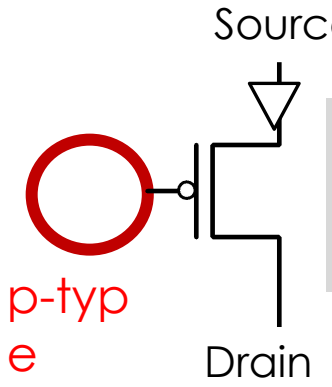
How Does a Transistor Work?

- The n-type transistor in a circuit with a battery and a bulb
- The p-type transistor works in exactly the opposite fashion from the n-type transistor





The circuit is closed when the gate is supplied with 3V



The circuit is closed when the gate is supplied with 0V

Transistors vs Logic Gates vs Circuit

- Now, we know how a transistor works
- How do we build logic structures out of transistors?

1. We construct **basic logical units** out of MOS transistors.

↓

Logic gates

They implement Boolean oprations

(NOT, AND, OR, NAND, NOR,..)

2. We construct **logical Circuit** out of logic gates to perform more complicated tasks.

↳ **Boolean Function** ⇔ **Algebraic expression**

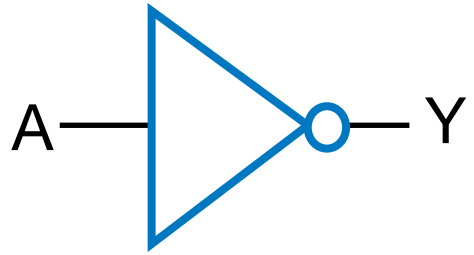
Truth table

Logic Gates

CMOS NOT Gate (Inverter)

A	P	N	Y
0	ON	OFF	1
1	OFF	ON	0

$$Y = \bar{A}$$

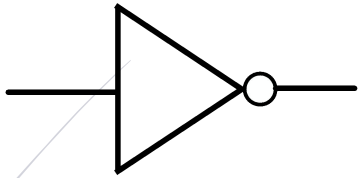


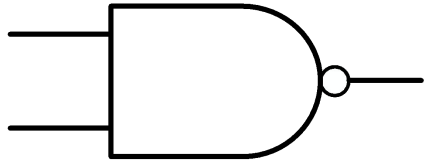
- This is actually the CMOS NOT Gate
  (Complementary MOS = n-type + p-type)
- Why do we call it NOT?
  - If A = 0V then Y = 3V
  - If A = 3V then Y = 0V
- **Digital circuit:** one possible interpretation
  - Interpret 0V as logical (binary) 0 value
  - Interpret 3V as logical (binary) 1 value

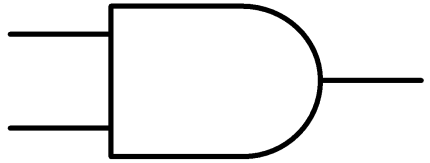
Truth table: shows what is the logical output of the circuit for each possible input

A	Y
0	1
1	0

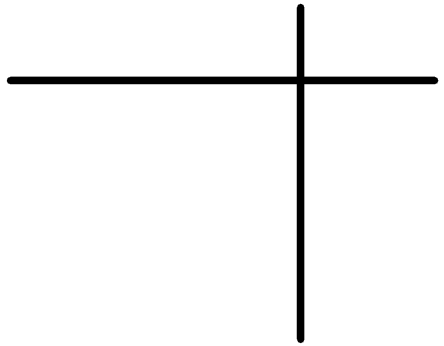
CMOS NOT, NAND, AND Gates

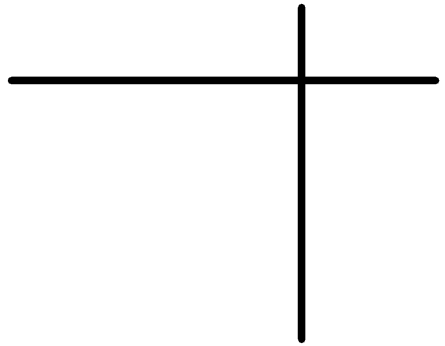


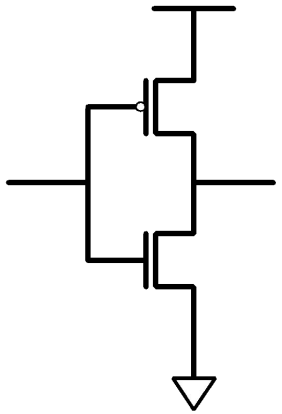


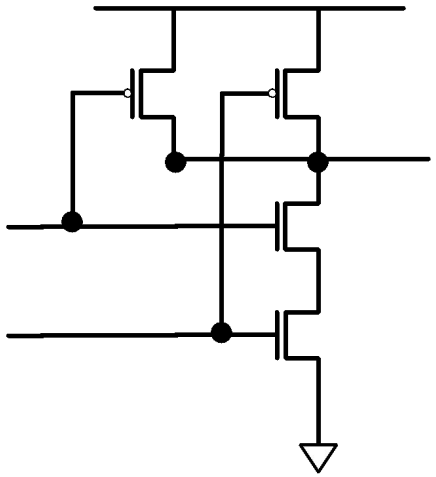


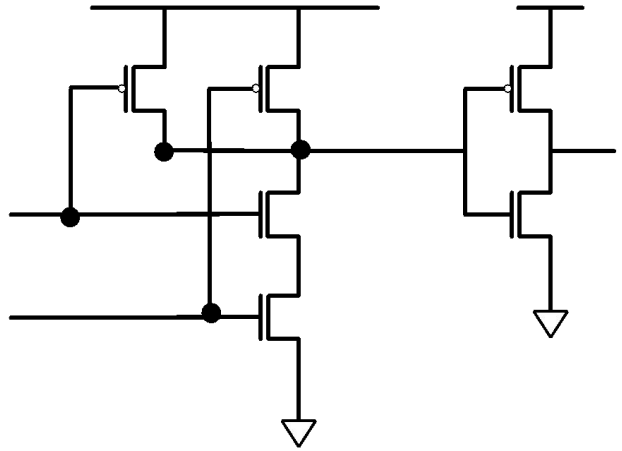
A	Y
0	1
1	0











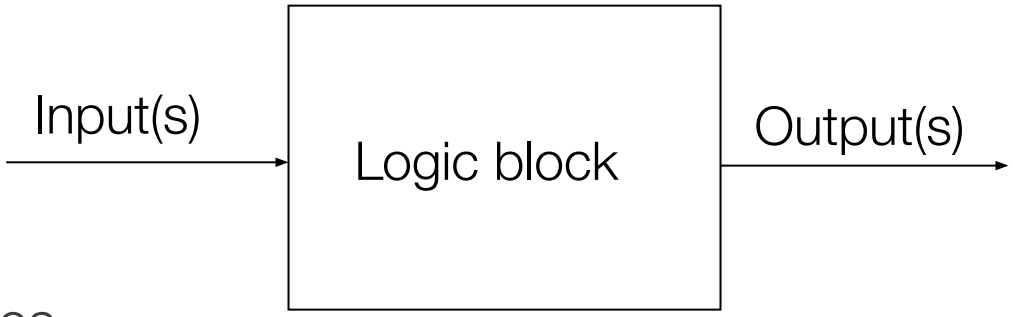
COMBINATIONAL GATES

Name	Symbol	Function	Truth
Table AND		X = A • B or X = AB	A B X: 0 0 0; 0 1 0; 1 0 0; 1 1 1
OR		X = A + B	A B X: 0 0 0; 0 1 1; 1 0 1; 1 1 1
I		X = A’	A X: 0 1; 1 0
Buffer		X = A	A X: 0 0; 1 1
NAND		X = (AB)’	A B X: 0 0 1; 0 1 1; 1 0 1; 1 1 0
NOR		X = (A + B)’	A B X: 0 0 1; 0 1 0; 1 0 0; 1 1 0
XOR Exclusive OR		X = A ⊕ B or X = A’B + AB’	A B X: 0 0 0; 0 1 1; 1 0 1; 1 1 0
XNOR Exclusive NOR or Equivalence		X = (A ⊕ B)’ or X = A’B’+ AB	A B X: 0 0 1; 0 1 0; 1 0 0; 1 1 1

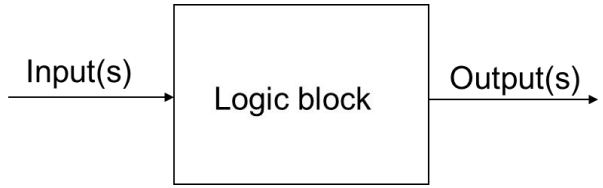
Logic Circuits

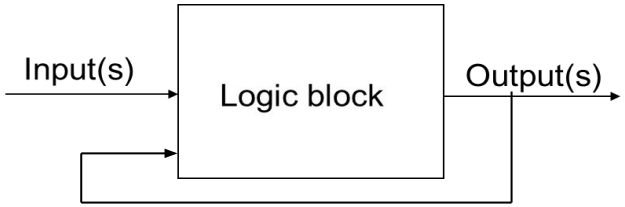
Logic Block

- A logic circuit is composed of :
  - Inputs : each input maybe 0 or 1
  - Outputs : each input maybe 0 or 1
  - Logic block : contains the logic gates.

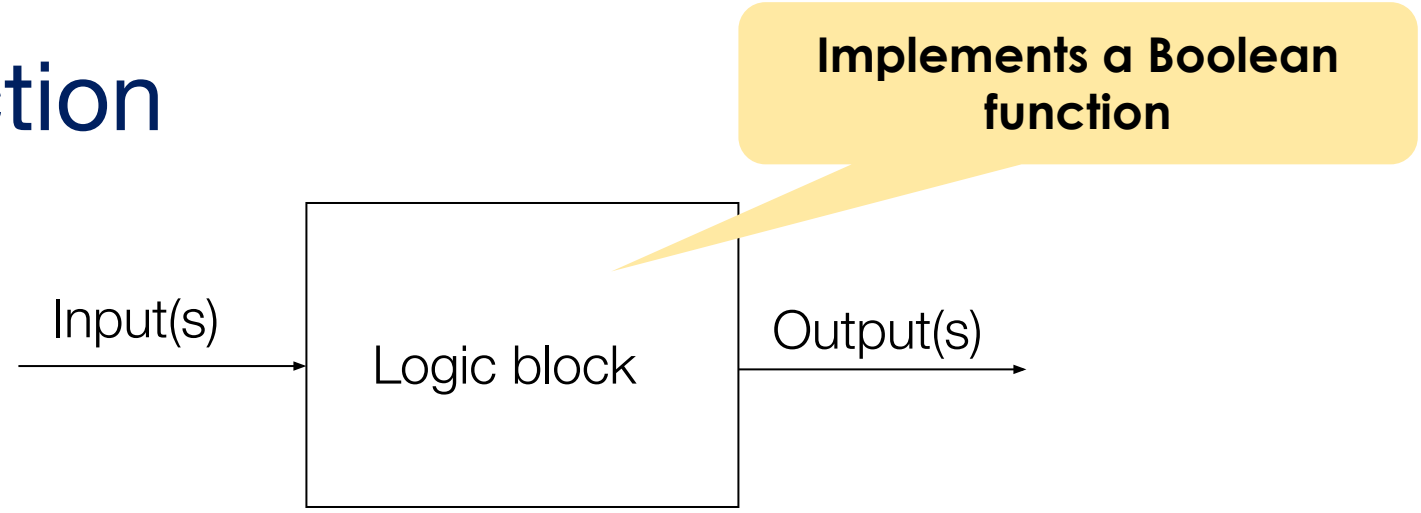


- Two types of circuit :
  - Combinational (arithmetic and logical)
    - The output depends only on the present values of the inputs
    - Logic gates are used
  - Sequential (storage (bit of SRAM))
    - The output depends on present input values and past output value





Boolean function



- Boolean function describes relationship between inputs and outputs
- What do we mean by “function”?
  - Unique mapping from input values to output values
  - The same input values produce the same output value every time
- A Boolean function may be represented as
  - An algebraic expression
  - A truth table

Boolean function : as an Algebraic Expression

$$W = X + \overline{Y} \cdot Z$$

- Variable W (Output) is a function of X, Y, and Z, can also be written as :

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

- The RHS of the equation is called an expression
- The symbols X, Y, Z are the literals of the function
- For a given Boolean function, there may be more than one algebraic expressions

$$W = X + \overline{Y} \cdot Z$$

$$W = X + \overline{Y} \cdot Z + Z \,.\overline{Z}$$

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

Boolean function : as a Truth Table

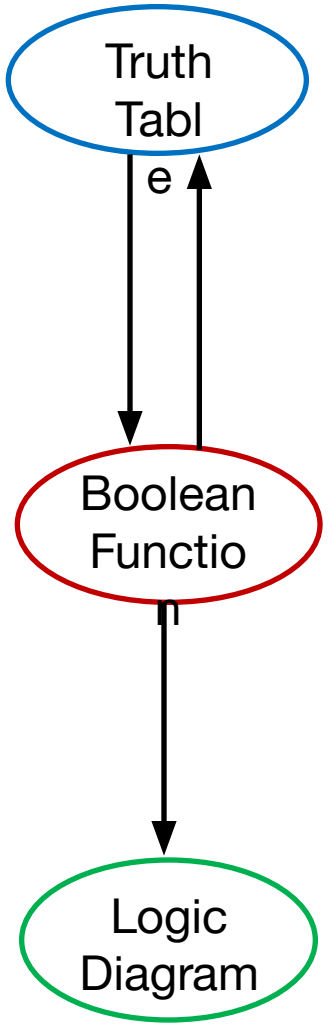
- The number of rows in the truth table is equal to 2^n, where *n* is the number of literals in the function
- The combinations of 0s and 1s for rows of this table are obtained from the binary numbers by counting from 0 to 2^n-1

$$W = X + \overline{Y} \cdot Z$$

X	Y	Z	W
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Logic circuit Designing : Steps to follow

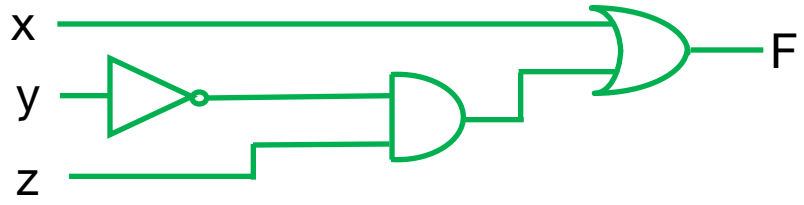
1. Determine the truth table for problem statement
2. Consider each output independently
   a) Consider all nonzero entries for the output
   b) Write the logic equation
   c) Simplify the logic equation using the laws of Boolean algebra (if possible) or Karnaugh Map method.
3. Draw the digital logic gate implementation of the simplified logic equation



x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

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

$F = x + y'z$



Boolean Equations to Represent a Logic Circuit (step 2)

Forms of Boolean Equation

- Assume we have the truth table of a Boolean Function
- How do we express the function in terms of the inputs in a standard manner?
- Idea: Express the truth table as a two-level canonical Boolean expression

SOP Form

Sum Of Products

Ex : F = ABC + A'BC + A'BC' + ..

POS Form

Product Of Sums

Ex : G = (A+B+C) . (A+B'+C') ..

Some Definitions

- **Variable :**

  A, B, C

- **Complement:** variable with a bar over it

  $\overline{A}, \overline{B}, \overline{C}$

- **Literal:** variable or its complement

  $A, \overline{A}, B, \overline{B}, C, \overline{C}$

- **Implicant:** product (AND) of literals

  $(A \cdot B \cdot \overline{C})$, $(\overline{A} \cdot C)$, $(B \cdot \overline{C})$

- **Minterm:** product (AND) that includes **all** input variables

  $(A \cdot B \cdot \overline{C})$, $(\overline{A} \cdot \overline{B} \cdot C)$, $(\overline{A} \cdot B \cdot \overline{C})$

- **Maxterm:** sum (OR) that includes **all** input variables

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

Two-Level Canonical (Standard) Forms

- Truth table is the unique signature of a Boolean *function* …
  - But, it is an expensive representation **Problem**
- A Boolean function can have many alternative Boolean expressions
  - i.e., many alternative Boolean expressions may have the same truth table (function)
  - Different Boolean expressions lead to different gate realizations **Problem**
- Sol : use Canonical form: standard form for a Boolean expression
  - Provides a unique algebraic signature
  - Two canonical forms : SOP and POS

Sum of Products Form (SOP)

Also known as disjunctive normal form or minterm expansion

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$F = \underbrace{\bar{A}BC}_{0\ 1\ 1} + \underbrace{A\bar{B}\bar{C}}_{1\ 0\ 0} + \underbrace{A\bar{B}C}_{1\ 0\ 1} + \underbrace{AB\bar{C}}_{1\ 1\ 0} + \underbrace{ABC}_{1\ 1\ 1}$$

- Each row in a truth table has a minterm
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)

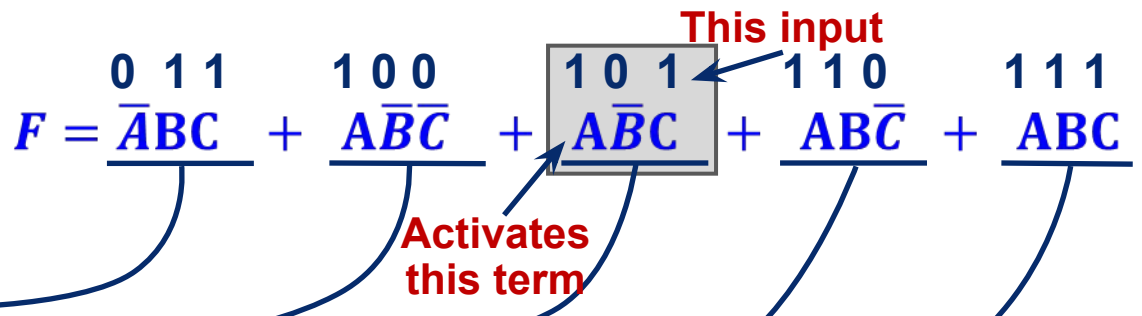
All Boolean equations can be written in SOP form

More used

Find all the input combinations (minterms) for which the output of the function is TRUE.

SOP Form — Why Does It Work?

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



- Only the shaded product term — $A\bar{B}C = 1 \cdot \bar{0} \cdot 1$ — will be 1 (activate)
- No other product terms will "turn on" — they will all be 0
- So if inputs A B C correspond to a product term in expression,
  - We get 0 + 0 + 1 + 0 + 0 = 1 for output
- If inputs A B C do not correspond to any product term in expression
  - We get 0 + 0 + 0 + 0 + 0 = 0 for output

Notation for SOP

- Standard “shorthand” notation
  - If we agree on the order of the variables in the rows of truth table…
    - then we can enumerate each row with the decimal number that corresponds to the binary number created by the input pattern

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

100 = decimal 4 so this is minterm #4, or m4

111 = decimal 7 so this is minterm #7, or m7

F =

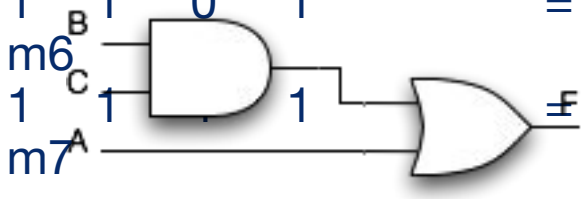
We can write this as a sum of products

Or, we can use a summation notation

Canonical SOP Forms

A	B	C	F	minterms
0	0	0	0	$\bar{A}\bar{B}\bar{C}$ = m0
0	0	1	0	$\bar{A}\bar{B}C$ = m1
0	1	0	0	$\bar{A}B\bar{C}$ = m2
0	1	1	1	$\bar{A}BC$ = m3
1	0	0	1	$A\bar{B}\bar{C}$ = m4
1	0	1	1	$A\bar{B}C$ = m5
1	1	0	1	$AB\bar{C}$ = m6
1	1	1	1	ABC = m7

Shorthand Notation for Minterms of 3 Variables



2-Level AND/OR Realization

F in canonical SOP form:

F(A,B,C) = ∑m(3,4,5,6,7)
= m3 + m4 + m5 + m6 + m7

$F =$

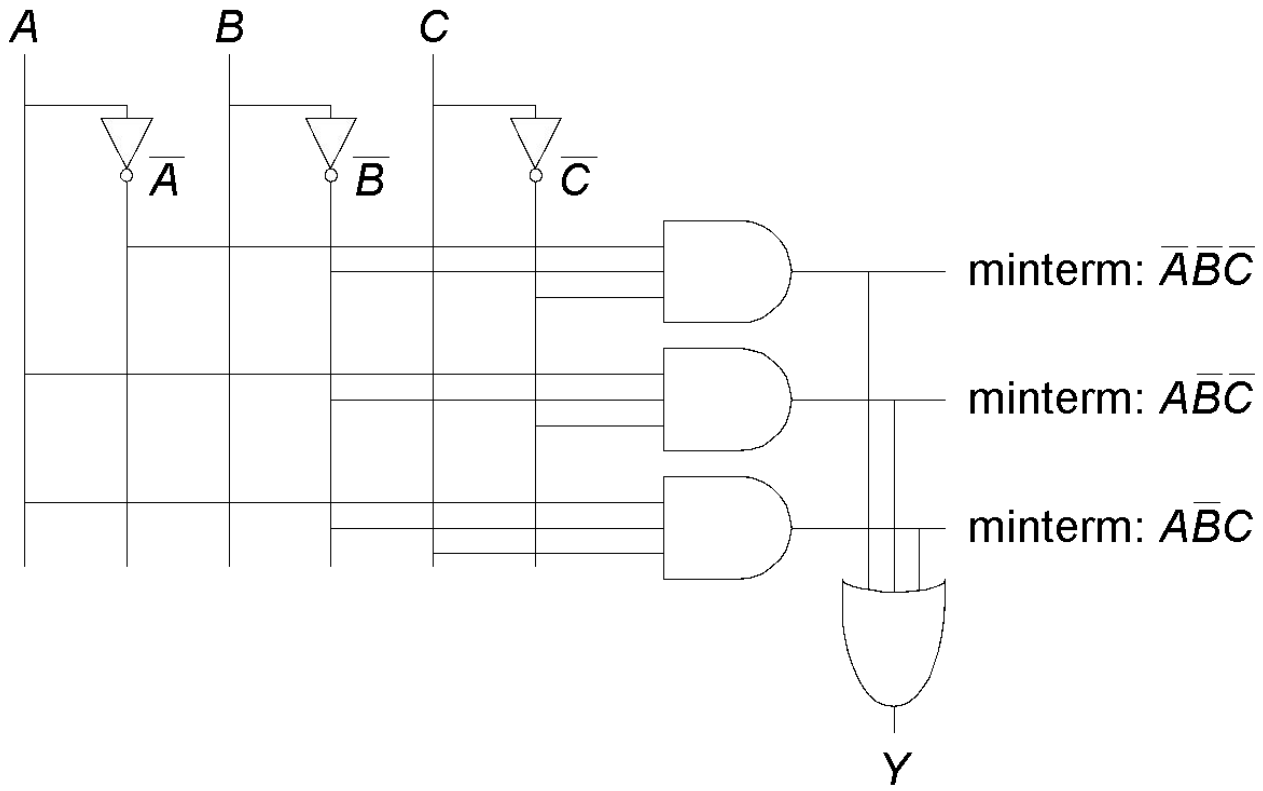
canonical form ≠ minimal form

F

minimal form

Canonical SOP Forms

- SOP (sum-of-products) leads to two-level logic
- Example: $Y = (\overline{A} \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot C)$

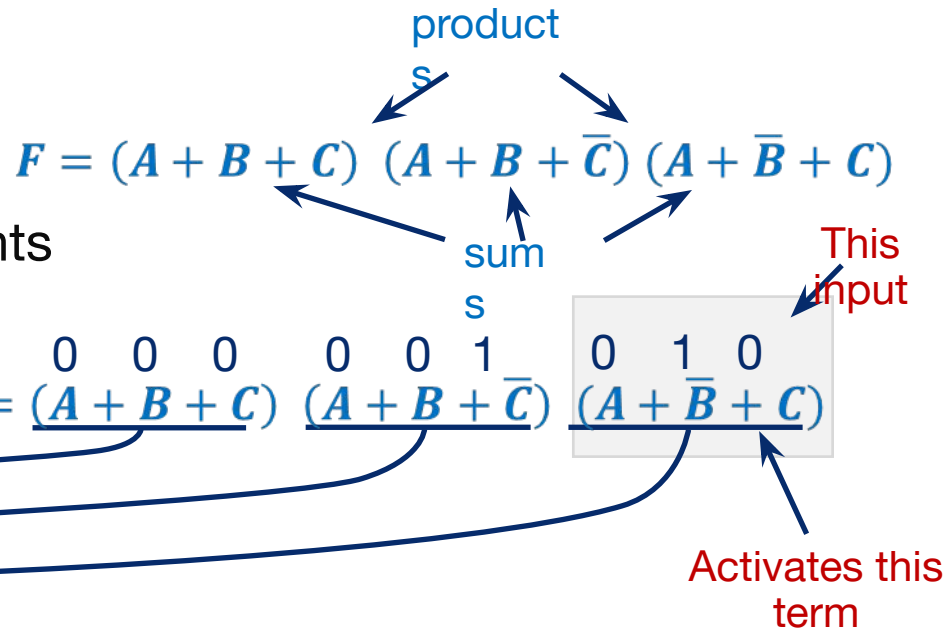


Alternative Canonical Form: POS

We can have another from of representation

A product of sums (POS)

Each sum term (Maxterm) represents one of the "zeros" of the function



A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

For the given input, only the shaded sum term will equal 0

$$A + \overline{B} + C = 0 + \overline{1} + 0$$

Anything ANDed with 0 is 0; Output F will be 0

Consider A=0, B=1, C=0

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Inpu

0t 1 0 →

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

$$0 \quad 1 \quad 0 \quad 0 \quad 1 \quad \overline{0} \quad 0 \quad \overline{1} \quad 0$$

$$1 \qquad 1 \qquad 0$$

$$F = 0$$

Only one of the products will be 0, anything ANDed with 0 is 0

Therefore, the output is F = 0

POS: How to Write It

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

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

$$A \quad \overline{B} \quad C$$

$$A + \overline{B} + C$$

Maxterm form:

1. Find truth table rows where F is 0

2. 0 in input col → true literal
3. 1 in input col → complemented literal

4. OR the literals to get a Maxterm
5. AND together all the Maxterms

Or just remember, POS of F is the same as the DeMorgan of SOP of $\overline{F}$!!

Canonical POS Forms

Product of Sums / Conjunctive Normal Form / Maxterm Expansion

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

$$= \prod M(0, 1, 2)$$

	A	B	C	Maxterms
0	0	0	0	$A + B + C = M0$
1	0	0	1	$A + B + \bar{C} = M1$
2	0	1	0	$A + \bar{B} + C = M2$
3	0	1	1	$A + \bar{B} + \bar{C} = M3$
4	1	0	0	$\bar{A} + B + C = M4$
5	1	0	1	$\bar{A} + B + \bar{C} = M5$
6	1	1	0	$\bar{A} + \bar{B} + C = M6$
7	1	1	1	$\bar{A} + \bar{B} + \bar{C} = M7$

Maxterm shorthand notation for a function of three variables

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Note that you form the maxterms around the “zeros” of the function

This is not the complement of the function!

Useful Conversions

1. Minterm to Maxterm conversion:

   rewrite minterm shorthand using maxterm shorthand

   replace minterm indices with the indices not already used

   $$\text{E.g., } F(A,B,C) = \sum m(3,4,5,6,7) = \prod M(0,1,2)$$

2. Maxterm to Minterm conversion:

   rewrite maxterm shorthand using minterm shorthand
   replace maxterm indices with the indices not already used

   $$\text{E.g., } F(A,B,C) = \prod M(0,1,2) = \sum m(3,4,5,6,7)$$

Logic Simplification: Karnaugh Maps (K-Maps)

Quick Recap on Logic Simplification

- The original Boolean expression may not be optimal

$$F = \sim A(A + B) + (B + AA)(A + \sim B)$$

- Can we reduce a given Boolean expression to an equivalent expression with fewer terms?

$$F = A + B$$

- The goal of logic simplification:
  - Reduce the number of gates/inputs
  - Reduce implementation cost

Logic Simplification

- Systematic techniques for simplifications

Key Tool: *The Uniting Theorem* —

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

A	B	F
0	0	0
0	1	0
1	0	1
1	1	1

$$F = A\bar{B} + AB = A(\bar{B} + B) = A(1) = A$$

B's value changes within the rows where F=1 ("ON set")

A's value does NOT change within the ON-set rows

If an input (B) can change without changing the output, that input value is not needed

→ *B is eliminated, A remains*

A	B	G
0	0	1
0	1	0
1	0	1
1	1	0

$$G = \bar{A}\bar{B} + A\bar{B} = (\bar{A} + A)\bar{B} = \bar{B}$$

B's value stays the same within the ON-set rows

A's value changes within the ON-set rows

→ *A is eliminated, B remains*

Complex Cases

- One example

$$Cout = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

- Problem
  - Easy to see how to apply Uniting Theorem…
  - Hard to know if you applied it in all the right places…
  - …especially in a function of many more variables

- Solution :
  - Need an intrinsically *geometric* representation for Boolean f( )

↳ Karnaugh map

Karnaugh Map (K-map) method

- In 1953, Maurice Karnaugh was a telecommunications engineer at Bell Labs.
  - While exploring the application of digital logic to the design of telephone circuits, he invented a graphical way of visualizing and then simplifying Boolean expressions.

2-variable K-map

A \ B	0	1
0	0	1
1	2	3

Output

3-variable K-map

A \ BC	00	01	11	10
0	0	1	3	2
1	4	5	7	6

4-variable K-map

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

The Gray Code is used to represent Numbering Scheme: 00, 01, 11, 10

K-map Rules

- What can be legally combined (circled) in the K-map?
  - Groupings can contain only 1s; no 0s.
  - Groups can be formed only at right angles; diagonal groups are not allowed.
  - The number of 1s in a group must be a power of 2 – even if it contains a single 1.
  - The groups must be made as large as possible.
  - Groups can overlap and wrap around the sides of the Kmap.
- How does a group become a term in an expression?
  - Determine which literals are constant, and which vary across group
  - Eliminate varying literals, then AND the constant literals
    - constant 1 → use X, constant 0 → use $\overline{X}$
- What is a good solution?
  - Biggest groupings → eliminate more variables (literals) in each term
  - Fewest groupings → fewer terms (gates) all together
  - OR together all AND terms you create from individual groups

Kmap Simplification for Three Variables

- In this Kmap, we see an example of a group that wraps around the sides of a Kmap.

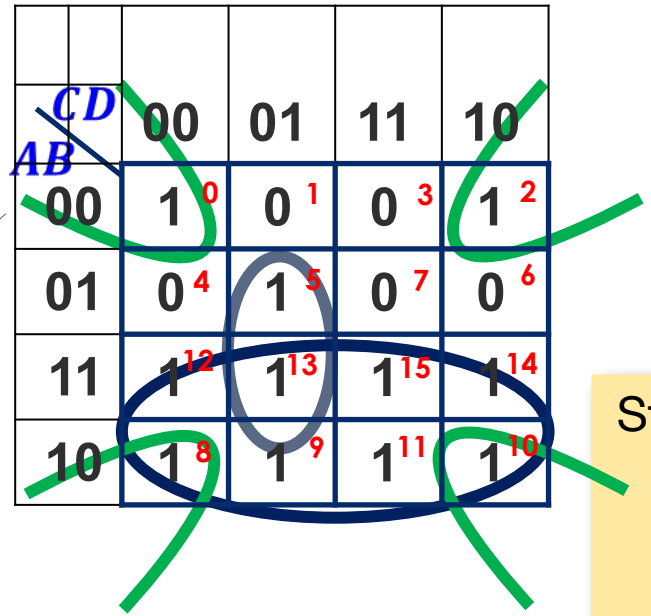
	X	Y	Z	F
0	0	0	0	1
1	0	0	1	1
2				
3	0	1	0	1
4				
5	0	1	1	1
6				
7	1	0	0	1
	1	0	1	0
	1	1	0	1
	1	1	1	0

X \ YZ	00	01	11	10
0	1 (0)	1 (1)	1 (3)	1 (2)
1	1 (4)	0 (5)	0 (7)	1 (6)

$$F(X,Y,Z) = X'Y'Z' + X'Y'Z + X'YZ' + X'YZ + XY'Z' + XYZ' = \sum m(0,1,2,3,4,6)$$

- The green group in the top row tells us that the values of y and z are not relevant to the term of the function that is encompassed by the group.
  - So, the term is reduced to $\overline{X}$
- The second group tells us that only the value of z is significant in that group.
- We see that it is complemented in that row, so the other term of the reduced function is $\overline{Z}$.
- Our reduced function is: $F(X,Y,Z) = \overline{X} + \overline{Z}$

K-map Cover - 4 Input Variables



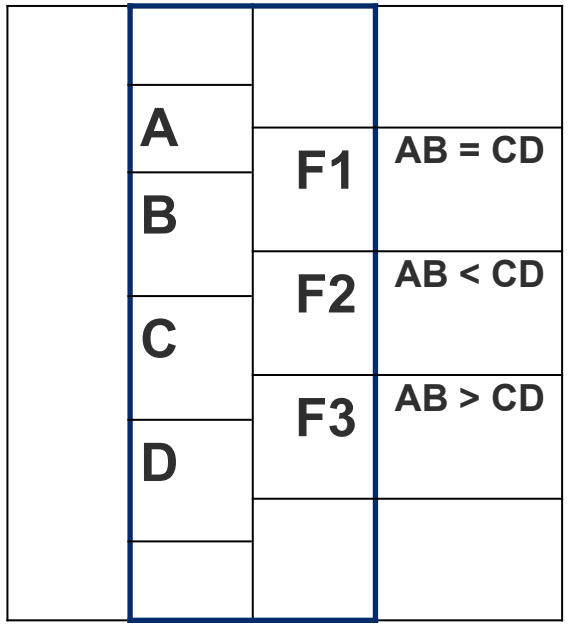
$$F(A,B,C,D) = \sum m(0,2,5,8,9,10,11,12,13,14,15)$$

$$F = A + \bar{B}\bar{D} + B\bar{C}D$$

Strategy for “circling” rectangles on Kmap:

Biggest “oops!” that people forget:

K-map Example: Two-bit Comparator

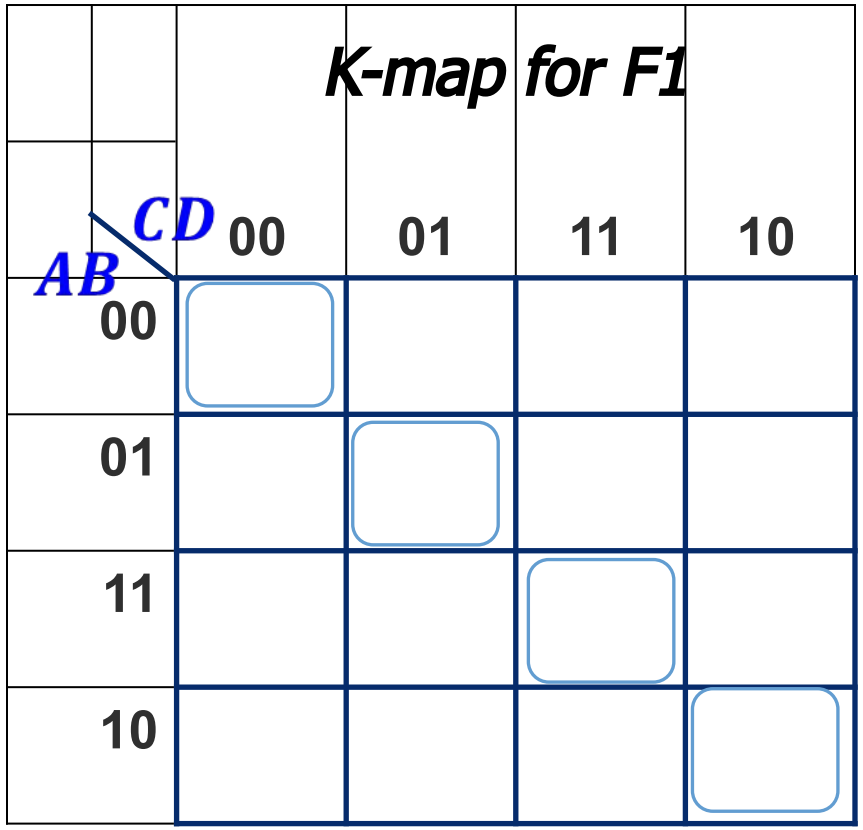


Design Approach:

Write a 4-Variable K-map for each of the 3 output functions

A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

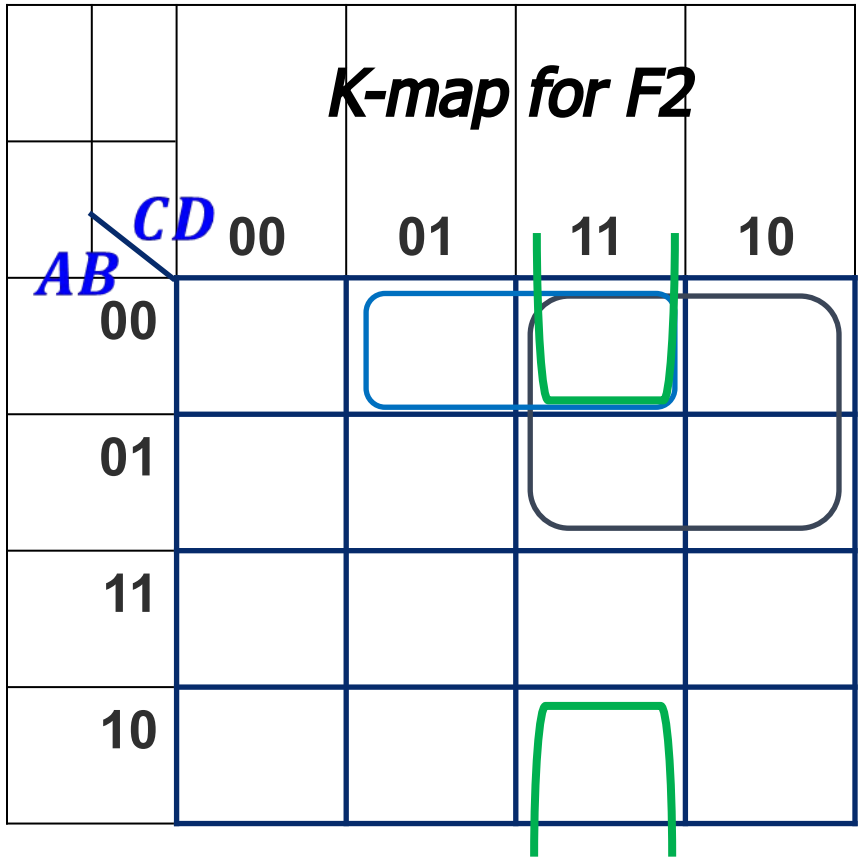
K-map Example: Two-bit Comparator (2)



F1 =

A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

K-map Example: Two-bit Comparator (3)



A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

F2 =

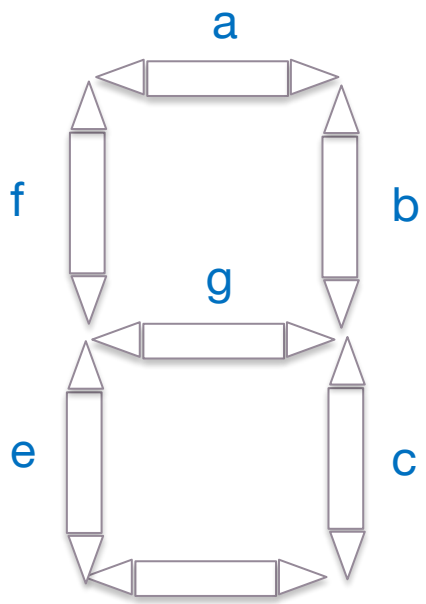
F3 = ? (Exercise for you)

K-maps with “Don’t Care”

- Real circuits don’t always need to have an output defined for every possible input.
  - For example, some calculator displays consist of 7-segment LEDs. These LEDs can display 2^7 -1 patterns, but only ten of them are useful.
- If a circuit is designed so that a particular set of inputs can never happen, we call this set of inputs a don’t care condition.
- They are very helpful to us in Kmap circuit simplification.

Example 1 : Seven-segment Decoder (Don't Care)

- Don't Care really means *I don't care what my circuit outputs if this appears as input*
  - You have an engineering choice to use DON'T CARE patterns intelligently as 1 or 0 to better simplify the circuit



I can pick 0 or 1 for any x

	A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	0	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
	1	0	1	0	x	x	x	x	x	x	x
	1	0	1	1	x	x	x	x	x	x	x
	1	1	0	0	x	x	x	x	x	x	x
	1	1	0	1	x	x	x	x	x	x	x
	1	1	1	0	x	x	x	x	x	x	x
	1	1	1	1	x	x	x	x	x	x	x

Example 2 : BCD Increment Function (Don't Care)

- BCD (Binary Coded Decimal) digits
  - Encode decimal digits 0 - 9 with bit patterns 0000_2 — 1001_2
  - When incremented, the decimal sequence is 0, 1, …, 8, 9, 0, 1

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

These input patterns should never be encountered in practice
(hey -- it's a BCD number!)
So, associated output values are

K-map for BCD Increment Function

A B C D
\+ 1
W X Y Z

X (without don't cares) =

X (with don't cares) = E E

Z (without don't cares) =

Z (with don't cares) =

0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1							
1							
1							
1							

W

Y

Y	CD 00	01	11	10
AB 00				
01				
11				
10				

X

X	CD 00	01	11	10
AB 00			1	
01	1	1		1
11	X	X	X	X
10			X	X

Z

Z	CD 00	01	11	10
AB 00	1			1
01	1			1
11	X	X	X	X
10	1		X	X