Digital Logic Circuits

- ***Combinatorial***:
  - Output depends on **combination of inputs**
- ***Sequential***:
  - Output depends on **inputs** as well as a **clock signal** (***sequenced operation***)
- ***Digital Signals***:
  - ***Discrete***, having values of only 0 and 1 (**binary**)
  - **Hi** (***Logic 1***) and **Lo** (***Logic 0***)
  - There is a region in between which is ***indeterminate*** (***non-discernible***)

Binary Number System:

* ***Based on powers of 2***: contrast with ***decimal number system***, which is based on ***powers of 10***
* **Uses only two digits**: ***0 and 1***
* ***Representation and Place Value***:
  - ***Integer Part***:

    $\cdots \quad 2^4\ (16) \quad 2^3\ (8) \quad 2^2\ (4) \quad 2^1\ (2) \quad 2^0\ (1)$ **.**

  - ***Fractional Part***:

    . $2^{-1}\ (0.5) \quad 2^{-2}\ (0.25) \quad 2^{-3}\ (0.125) \quad 2^{-4}\ (0.0625) \quad \cdots$

* ***Example***: 101**.**101 = 5.625

* ***Example*: Binary to Decimal Conversion:**

$$1\quad 1\quad 0\quad 0\quad 1 = (1\times 2^4 + 1\times 2^3 + 1\times 2^0) = 25$$

* ***Example*: Decimal to Binary Conversion:**

$$15 = (1\times 2^3 + 1\times 2^2 + 1\times 2^1 + 1\times 2^0) = 1111$$

* ***Fractional Numbers*: Binary to Decimal:**

$$11.101 = (1\times 2^1 + 1\times 2^0 + 1\times 2^{-1} + 1\times 2^{-3}) = 3.625$$

* ***Fractional Numbers*: Decimal to Binary:**

$$0.4475 = (1\times 2^{-2} + 1\times 2^{-3} + 1\times 2^{-4}) = 0.0111$$

* ***Note***: The decimal equivalent of 0.0111 is 0.4375, but we started with 0.4475 $\Rightarrow$ Known as **Conversion** (or **Quantization**) **Error** (***QE***): ***The number of bits is not sufficient, we need to increase it to reduce the error***

* ***Note***:
  - **1 bit numbers**: 0 and 1 (***Decimal equivalent***: 0 and 1)
  - **2 bit numbers**: 00 01 10 and 11 (***Decimal equivalent***: 0 to 3)
  - **3 bit numbers**: 000 ... 111 (***Decimal equivalent***: 0 to 7)
  - **4 bit numbers**: 0000 ... 1111 (***Decimal equivalent***: 0 to 15)

* ***Inference***: With ***N number of bits***, we can represent **decimal numbers ranging from 0 to** $\left(2^{N}-1\right)$
* The ***leftmost bit*** is known as the **Most Significant Bit** (***MSB***), while the ***rightmost bit*** is known as the **Least Significant Bit** (***LSB***)
* ***Definitions***:
  - **4 Bits** $\rightarrow$ ***1 Nibble***
  - **8 Bits** or **2 Nibbles** $\rightarrow$ ***1 Byte***
  - **16 Bits** or **2 Bytes** or **4 Nibbles** $\rightarrow$ ***1 Word***

* ***Addition of Binary Numbers***:

$0+0=0,\ 0+1=1,\ 1+0=1,\ 1+1=0$ (**with a carry of 1**)

* ***Subtraction of Binary Numbers***:

$0-0=0,\ 1-0=1,\ 1-1=0,\ 0-1=1$ (**with a borrow of 1**)

* ***Example***: $5 + 6 \Rightarrow 101 + 110 = 1011 \Rightarrow 11$

$9-5 \Rightarrow 1001-101=0100 \Rightarrow 4$

* ***Multiplication of Binary Numbers***:

$0\times0 = 0,\ 0\times1= 0,\ 1\times0 = 0,\ 1\times1 = 1$ (**no carry**)

* ***Division of Binary Numbers***:

$0\div1 = 0,\ 1\div1 = 1$, **Division by zero not allowed**

(***overflow***)

* ***Some more examples***:

- **Convert decimal 35 to binary (*by division*)**:

  $35 \div 2 = 17 +$ **1** (LSB), $17 \div 2 = 8 +$ **1** (next bit), $8 \div 2 = 4 +$ **0**, $4 \div 2 = 2 +$ **0**, $2 \div 2 = 1 +$ **0**, $1 \div 2 = 0 +$ **1** (MSB)

  $\Rightarrow$ **Final binary representation**: 100011

- **Express 0.9×2 in binary**:

  $1.8 =$ **1** (the lone digit in the integer part) + 0.8

  $0.8 \times 2 =$ **1** (the first digit in the fractional part) + 0.6, $0.6 \times 2 =$ **1** + 0.2, $0.2 \times 2 =$ **0** + 0.4, ...

  $\Rightarrow$ **Final binary representation**: 1.1100...

* Note that with 4 digits after the decimal point, the converted number is 1.75, which gives ***QE*** of 0.05, which is the same even if we had only 2 digits after the decimal point

  $\Rightarrow$ Concept of ***Bit Optimization*** depending on the **maximum allowed QE**

  $\Rightarrow$ ***Generally, increase in number of bits leads to lower QE, but not always, as this example clearly illustrates***

Complements and Negative Numbers:

* ***Note***: $X - Y$ is same as $X + (-Y)$
* Thus, before subtraction, if we ***negate*** the number that is to be subtracted, then we just need to do an ***addition*** operation $\Rightarrow$ **simplifies circuitry**
* ***Two methods of negation***:
  - **One's Complement**
  - **Two's Complement**
* ***One's Complement***:
  - Just ***invert*** all the bits of the binary number
  - ***Example***: +5 (0101) becomes -5 (1010)

- To get the original number back, apply ***reverse process***, i.e., ***invert*** all the bits
- Not much used, since it returns both ***positive and negative zero*** (+0 and -0) (to be discussed)

* ***Two's Complement***:

- **One's complement plus 1**, i.e., ***invert all the bits***, ***and add 1 to the LSB***
- ***Example***: +5 (0101) becomes -5 (1011)
- **Avoids the problem of positive and negative zeros**
- Apply ***reverse process*** to get the original number back
- ***Widely used by modern computer systems***

Sign-Magnitude Convention:

* In this convention, the ***MSB is reserved for the sign bit*** (**0 for positive**, and **1 for negative**)
* Thus, for an ***N bit number***, with 1 bit reserved for sign, we can express **0 to** $\pm\left(2^{N-1}-1\right)$
* ***Example***: An ***8 bit number*** can represent ***0 to*** $\pm$ ***127***
* ***One's Complement Notation***:
  - ***Example***: $+15 \rightarrow 01111$, $-15 \rightarrow 10000$
  - ***Note***: $0000 \rightarrow +0$, $1111 \rightarrow -0$
  - This is why one's complement notation is ***avoided***

* ***Two's Complement Notation***:
  - ***Example***: $3 - 7 = 3 + (-7) = -4$
  - ***Note***: To express -7 in two's complement notation, we need ***4 bits***
  - ***Add sign bit***: $3 \rightarrow 0\ 0\ 1\ 1$

    $-7 \rightarrow \underline{1\ 0\ 0\ 1}$

    Add $\rightarrow 1\ 1\ 0\ 0 \rightarrow -4$
  - ***Note***: In two's complement notation, $+4 \rightarrow 0100$, while $-4 \rightarrow 1100 \Rightarrow$ ***Unique situation***
  - ***Example***: $2 - 7 = 2 + (-7) = -5$

    $\Rightarrow 0010 + 1001 = 1011$, which is -5

The Hexadecimal Number System:

* Abbreviated as ***HEX*** (H): ***base 16***, and ***grouped in 4 bits each***
* Extremely convenient and all computer systems use HEX number system
* 0000 to 1001: 0_D to 9_D: 0_H to 9_H, 1010: 10_D: A_H, 1011: 11_D: B_H, 1100: 12_D: C_H, 1101: 13_D: D_H, 1110: 14_D: E_H, and 1111: 15_D: F_H (***D : Decimal***)
* Thus, $(1010\ 1111)_B \leftrightarrow (AF)_H$ (***B : Binary***)

The Octal Number System:

* ***Base 8***, and ***grouped in 3 bits each***
* **Can count only from 0 to 7 (*000 to 111*)**
* $542_8 = 5\times8^2 + 4\times8^1 + 2\times8^0 = 320+32+2 = 354_{10}$
* 542_8 ***grouped in 3 bits each***: $101\ 100\ 010_2$
* $101100010_2 = 2^8+2^6+2^5+2^1 = 256+64+32+2$

  $= 354_{10}$
* 101100010_2 ***grouped in 4 bits*** would result in HEX
* $0001\ 0110\ 0010_2 = 162_{16} = 1\times16^2+6\times16^1+2\times16^0$

  $= 256+96+2 = 354_{10}$ (***Note***: **Added 3 zeros to left**)
* Neat, isn't it?

Boolean Algebra:

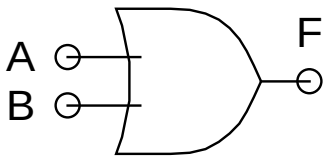
* Developed by ***George Boole***
* **Uses only two variables**: ***0 and 1***: **Binary**
* Mathematics associated with binary number system is ***Boolean Algebra***
* **Positive Logic**: 1 (***True - Hi***) and 0 (***False - Lo***)
* **Negative Logic**: 1 (***False - Hi***) and 0 (***True - Lo***)
* **Logical Addition**: ***OR (+) Operation***:
  - $0 + 0 = 0,\ 0 + 1 = 1,\ 1 + 0 = 1,\ 1 + 1 = 1$
* **Logical Multiplication**: ***AND ($\cdot$) Operation***:
  - $0 \cdot 0 = 0,\ 0 \cdot 1 = 0,\ 1 \cdot 0 = 0,\ 1 \cdot 1 = 1$

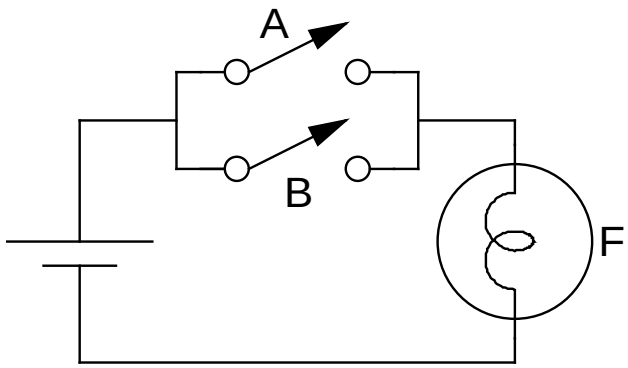
* **Truth Table**: ***Listing of all possible input combinations and correrspondingouputs***

OR Gate:

* ***Switches in parallel***:

  $\Rightarrow$ If ***either*** one of them is ***on***, the light will ***glow***





* ***Truth Table***:

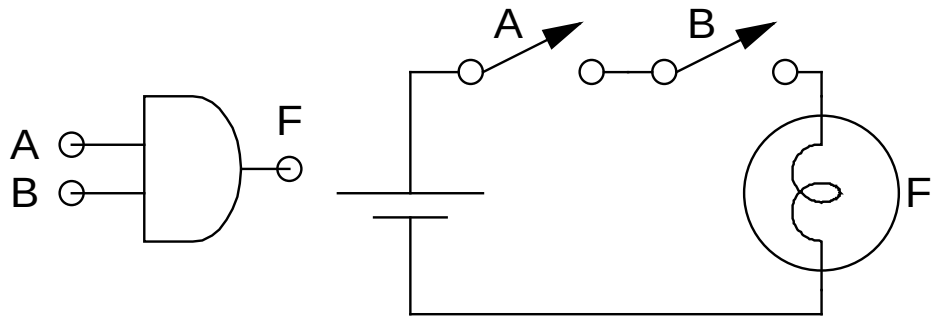
A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

$F = A + B$

AND Gate:

* ***Switches in series***:

$\Rightarrow$ Light will ***glow***
only if ***both***
switches are ***on***



* ***Truth Table***:

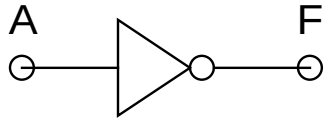
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

$F = A \cdot B$

* **Note that the number of entries in the truth table is 2^N, where N = number of input variables**

NOT Gate:

* ***Simply an inverter***



* ***Truth Table***:

A	F
0	1
1	0

$F = A' \left(\text{or } \overline{A}\right)$

Some Useful Rules:

$0 + X = X, \quad 1 + X = 1, \quad X + X = X, \quad X + \overline{X} = 1,$

$0 \cdot X = 0, \quad 1 \cdot X = X, \quad X \cdot X = X, \quad X \cdot \overline{X} = 0, \quad \overline{\overline{X}} = X,$

$X + Y = Y + X, \quad X \cdot Y = Y \cdot X, \quad X + (Y + Z)$

$= (X + Y) + Z, \quad X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z$

Some Useful Rules (Contd.):

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

De Morgan's Theorem:

* **Two extremely important theorems**:
  - $\overline{X + Y} = \overline{X} \cdot \overline{Y}$ and $\overline{X \cdot Y} = \overline{X} + \overline{Y}$
* ***Thus, any logic function can be implemented by using either only OR and NOT gates, or only AND and NOT gates***

* ***AND and OR operations are DUAL of each other***
* **Any function can be realized by just one of these basic two operations, along with the complement operation**
* ***Two families of logic functions*:**
  - **Sum of Products (*SOP*)** $\rightarrow \left(X \cdot Y\right) + \left(W \cdot Z\right)$
  - **Product of Sums (*POS)*** $\rightarrow \left(A + B\right) \cdot \left(C + D\right)$
* **Any logical expression can be reduced to one of these two forms, which are actually equivalent**

Logic Minimization by Simplification:

* Consider f(A,B,C,D) = A′B′D + A′BD + BCD + ACD

  = A′D(B′ + B) + BCD + ACD = A′D + BCD + ACD

  = (A′ + AC)D + BCD = (A′ + C)D + BCD

  = A′D + CD + BCD = A′D + CD(1 + B)

  = A′D + CD (SOP) = (A′ + C)D (POS)

* Note that in the final expression, B is not even there!

* ***Applied prudently, logic minimization is immensely powerful, and saves a lot of hardware in implementing logic functions***

Realization of Logic Function from Truth Table:

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

* Need to form ***SOP***
* **Pick rows for which Y = 1, and write A, B, and C in AND form, to produce Y = 1**
* For example, 2nd row would be:

  $Y = A'B'C$
* Thus, $Y = A'B'C + A'BC + AB'C' + AB'C + ABC' + ABC$

* Now, start ***minimizing***:

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

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

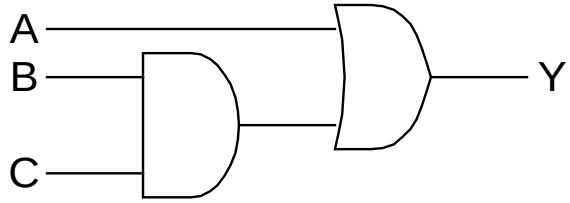
$= A'C + AB' + AB = A'C + A(B' + B)$

$= A'C + A = A + C$

* Implementation is a simple ***OR*** gate with two inputs A and C (note that B doesn't even appear in the final expression)
* ***Though both SOP and POS are valid representations, generally, SOP representation yields a lower gate count as compared to POS representation***

Example:

* $Y = A + BC$ (***SOP***) $\Rightarrow$



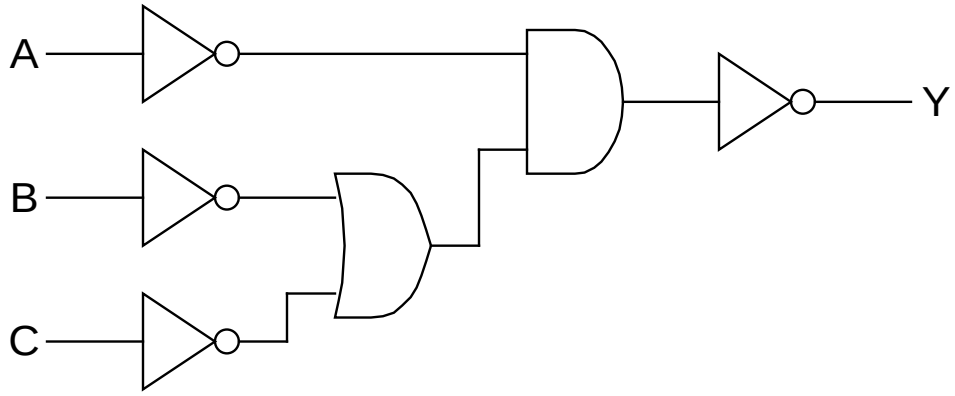
* Let's convert it to ***POS*** representation:

$$\overline{Y} = \overline{A + BC} = \overline{A} \cdot \overline{BC}$$

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

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

$\Rightarrow$ ***POS Representation***



* ***Note***: While SOP representation needed only ***2 gates***, POS representation needed ***6 gates***

$\Rightarrow$ ***SOP representation preferred***

NAND and NOR Gates:

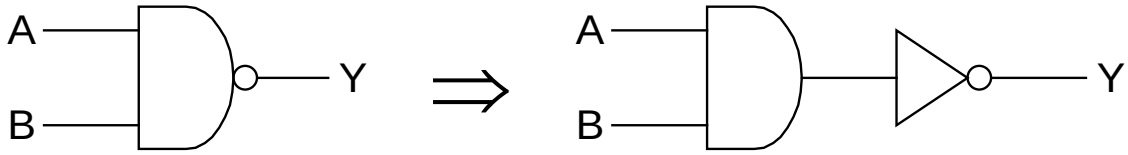
* **Complements of AND and OR Gates**, respectively
* Called ***Universal Gates***, since any logic function can be implemented using these two gates
* ***Gate Array*** based designs have only ***NAND*** and ***NOR*** Gate Arrays – only ***interconnections*** need to be changed to implement any logic function

  $\Rightarrow$ Extremely powerful and versatile
* Special class of VLSI circuits known as **Field Programmable Gate Array** (***FPGA***)

  $\Rightarrow$ The interconnections can be done ***in-situ***

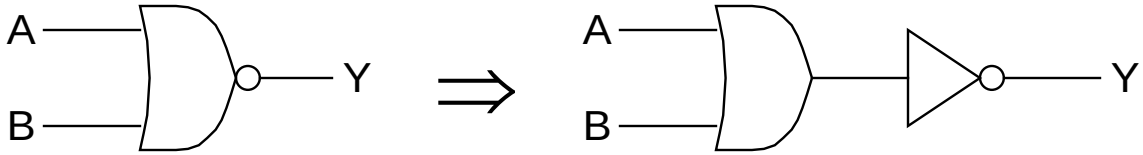
* **NAND**: ***NOT AND***: Performs ***logical addition*** of the complements of the inputs

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



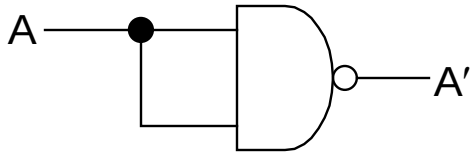
* **NOR**: ***NOT OR***: Performs ***logical multiplication*** of the complements of the inputs

$$Y = \overline{A + B} = \overline{A} \cdot \overline{B}$$



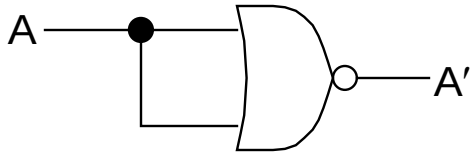
Truth Tables of NAND and NOR Gates:

NAND:	A	B	AND	NAND
	0	0	0	1
	0	1	0	1
	1	0	0	1
Odd one:	1	1	1	0



NOT Gate

NOR:	A	B	OR	NOR
Odd one:	0	0	0	1
	0	1	1	0
	1	0	1	0
	1	1	1	0



NOT Gate

Exclusive-OR (XOR) and Exclusive-NOR (XNOR):

XOR: **Basically OR gate**, ***with one exclusion***, i.e., when both inputs are 1, output is 0

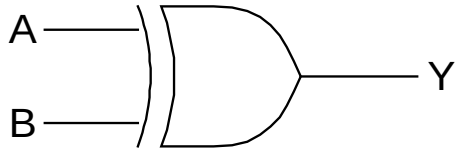
Truth Table:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

$Y = A \oplus B = AB' + A'B$

Either A high or B ***high***, ***but not both***, would produce a ***high*** output

Known as ***odd parity detector***



***XNOR*: Basically NOR gate, *with one exclusion*, i.e.,**

when both inputs are 1, output is 1

***Truth Table*:**

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

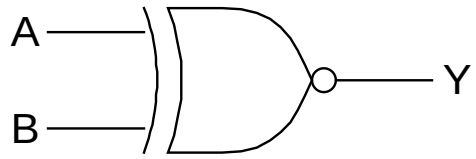
$Y = \overline{A \oplus B} = A'B' + AB$

For output to be ***high***,

both A and B should

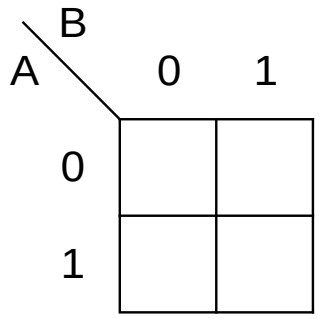
either be high or low

Known as ***even parity detector***

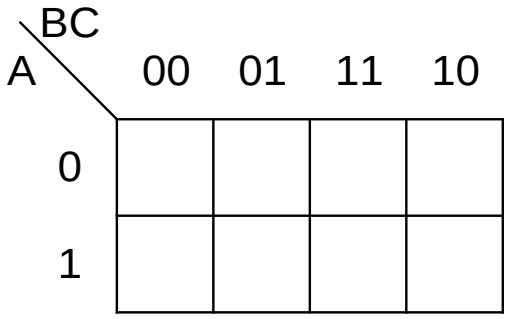


Karnaugh Map (K-Map) and Logic Design:

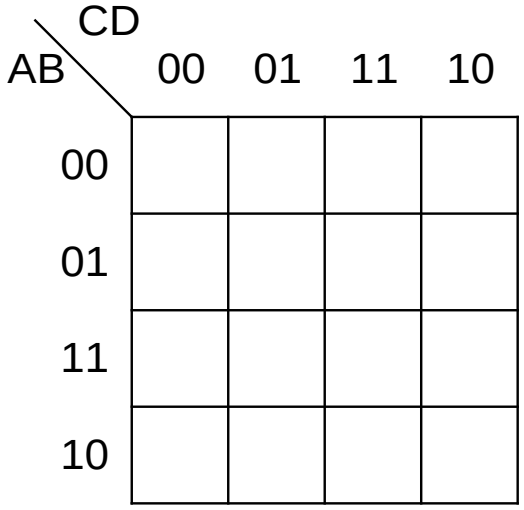
* Named after its inventor ***Maurice Karnaugh*** (1953)
* Extremely simple yet highly efficient
* Used extensively for ***logic minimization*** to arrive at the **most optimal logic design**



2-Variable Karnaugh Grid



3-Variable Karnaugh Grid

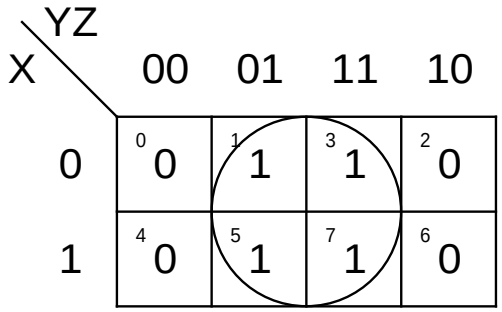


4-Variable Karnaugh Grid

* For ***N variables***, total number of grid boxes = 2^N
* Rows and columns are assigned such that the adjacent terms ***change by only 1 bit***, e.g., 00, 01, 11 (**note that it is not 10**), and 10
* Each cell content is known as a **minterm**
  - ***Example***: 3-variable minterms: A′B′C′, A′B′C, A′BC′, A′BC, AB′C′, AB′C, ABC′, and ABC
* Grid boxes are filled by putting ***either 0 or 1***, corresponding to the ***input-output combination***

Example:

X	Y	Z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



X: MSB
Z: LSB

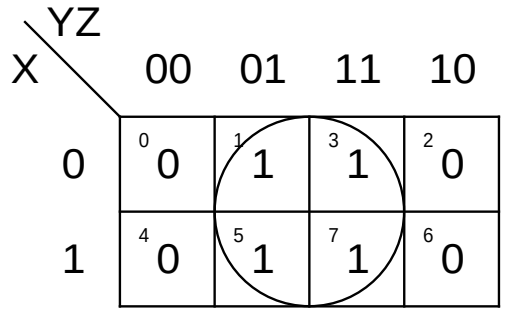
* K-map is **foldable (*left*/*right* and *up*/*down*) $\Rightarrow$ what is put in the first row/column is immaterial**
* **Each cell is numbered (left top corner), corresponding to the decimal equivalent of the binary bits**

Steps of Minimization:

* Define a **supercube** (some books say ***subcube***) as a set of 2^m adjacent cells, all of which have ***logical value 1***, for m = 1, 2, 3, ..., N
* Thus, a subcube can consist of **1, 2, 4, 8, 16, 32, ...,** number of individual cubes
* Identify the ***largest possible*** such supercubes (**there may be more than one such supercube in a K-map**)
* ***How does it do the minimization***?
  - **Since $\left(XY + X'Y\right)$ is always equal to Y**

    $\Rightarrow$ X becomes immaterial (*redundant*)

* For the example considered, first note that we have formed a ***supercube*** by combining adjacent 1s in the cubes



* Thus, f = 1 when either X′Y′Z or X′YZ or XY′Z or XYZ is 1

$$\Rightarrow \; f = X'Y'Z + X'YZ + XY'Z + XYZ$$

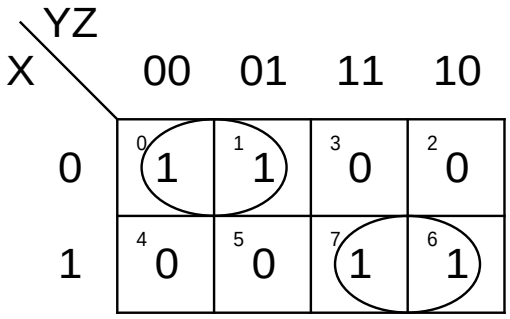
$$= \left(Y' + Y\right)X'Z + \left(Y' + Y\right)XZ$$

$$= X'Z + XZ = \left(X' + X\right)Z = Z$$

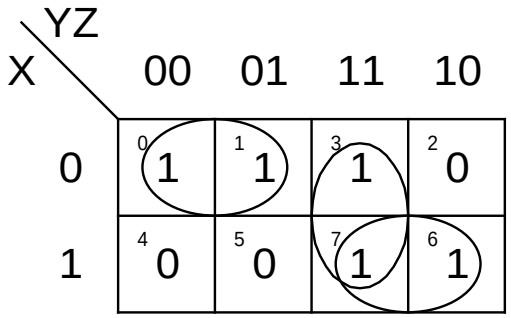
* From the K-map, we see that f = 1 only when Z = 1, and X and Y both are either 0 or 1

$\Rightarrow$ **X and Y are redundant and f = Z**

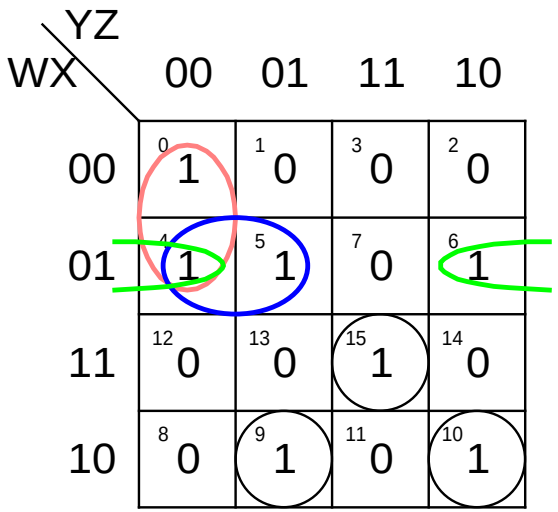
***Some More Examples*:**



$f = X'Y' + XY$



$f = X'Y' + XY + YZ$



W: MSB, Z: LSB

$f = W'Y'Z' + W'XY' + W'XZ' + WXYZ + WX'Y'Z + WX'YZ'$

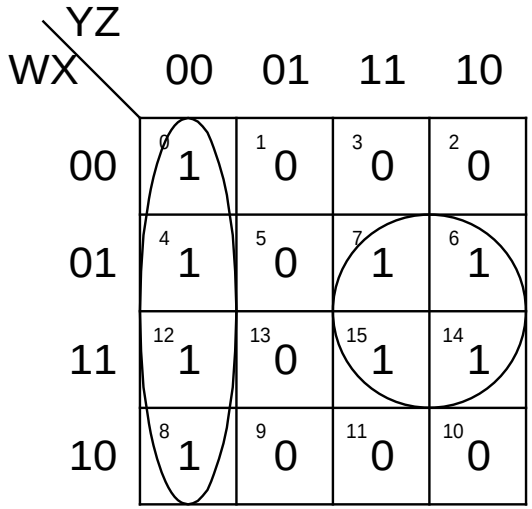
Tips:

*** Make sure that all 1s are accounted for**

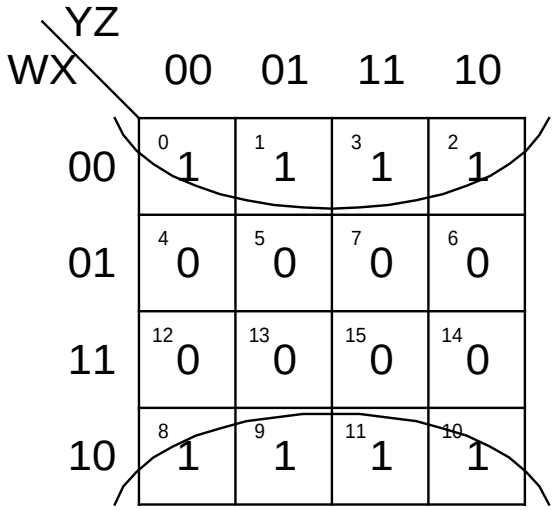
*** Pay special attention to the corner 1s and foldability**

*** No supercube should have odd number of cubes (*except 1 – monocube*)**

*** Supercubes can be intersecting**

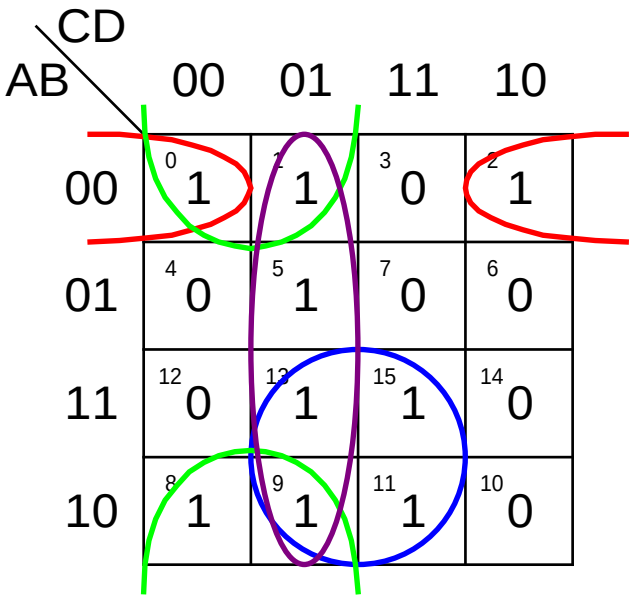

YZ
WX
00 01 11 10
00
01
11
10
0 1
1 0
3 0
2 0
4 1
5 0
7 1
6 1
12 1
13 0
15 1
14 1
8 1
9 0
11 0
10 0

$f = Y'Z' + XY$


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

$f = X'$

More Complicated Example:



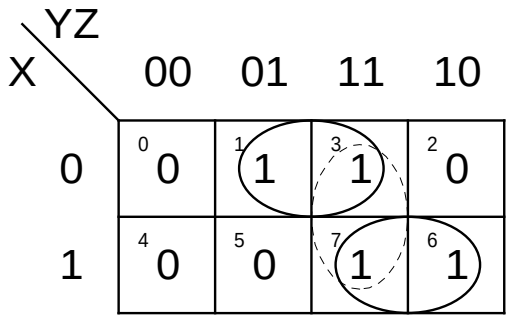
A: MSB, D: LSB

$f = A'B'D' + C'D + AD + B'C'$

* ***K-map*** is an excellent tool for ***logic minimization***

* Consider f = XY + X′Z + YZ

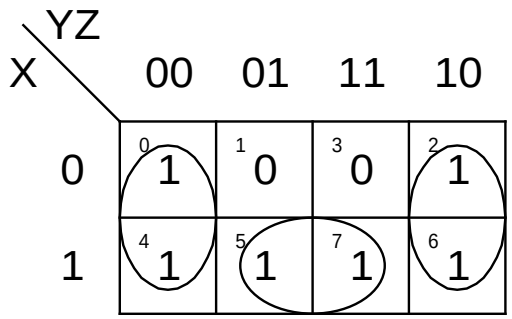
  • Can't be minimized any more using ***Boolean algebra***



f = X′Z + XY
(A simpler form)

* Note that the YZ term is ***redundant*** (the dotted supercube), since all 1s have already been accounted for using the two other supercubes

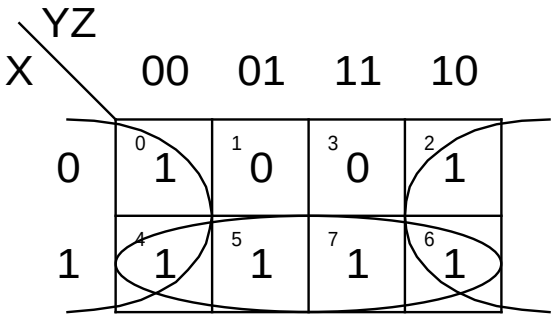
* Consider another case:



f = Y′Z + XZ + YZ′

* This minimization is ok, however, ***inefficient***, since it can be further reduced

  • Look for the ***largest supercube***:



f = X + Z′
Much simpler & compact expression results

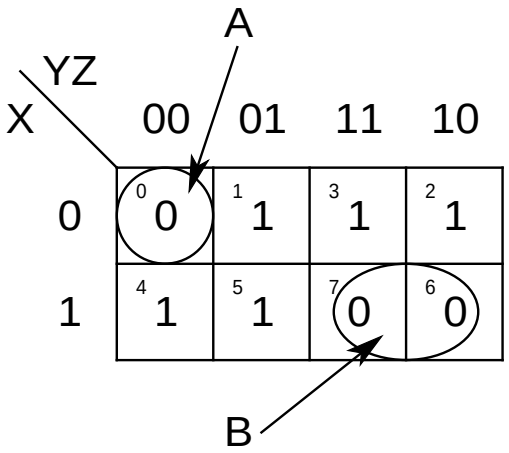
Product of Sums (POS) Realization & Minimization:

* Resulting expression in the form $(A+B)\cdot(C+D)$
* SOP and POS representations are completely ***equivalent***, however, in general, ***SOP representation is more efficient***

Algorithm for POS Representation:

* Instead of 1, bunch ***0s*** in supercubes
* Variables causing this ***0*** is written in ***OR*** form
* After identifying all such terms, ***AND*** them
* Thus, the output will be ***0*** only if all the results of the functions are ***0***

Example:



* ***Term A will be zero if X or Y or Z or all are zero***

$\Rightarrow$ It is equivalent to $\left(X + Y + Z\right)$

* ***Term B will be zero if $X = 1\,(X' = 0)$ and/or $Y = 1\,(Y' = 0)$, with Z immaterial***

$\Rightarrow$ It is equivalent to $\left(X' + Y'\right)$

* Thus, POS Representation:

$$f = \left(X + Y + Z\right)\cdot\left(X' + Y'\right)$$

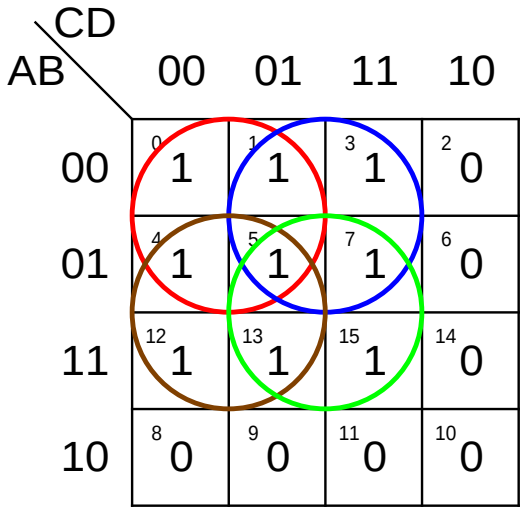
Formal Definitions:

* **Minterm**: Boolean expression resulting in output ***1*** for a single cell or a group of cells (arranged in 2^N supercube)
* **Maxterm**: Boolean expression resulting in output ***0*** for a single cell or a group of cells (arranged in 2^N supercube)

Σ and π Notations:

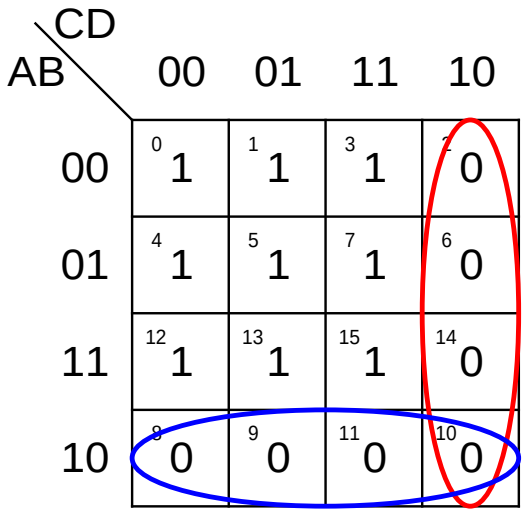
* Σ_m: **SOP** (m - ***minterms***)
* π_M: **POS** (M - ***maxterms***)
* Very compact means of describing ***Truth Tables***

Example: $f(A,B,C,D) = \Sigma_m(0,1,3,4,5,7,12,13,15)$



f = A′C′ + A′D + BC′ + BD

Example: $f(A,B,C,D) = \pi_M\left(2,6,8,9,10,11,14\right)$



$f = (A' + B)\bullet(C' + D)$

* ***Note***: Both representations yielded the ***same final result***, which obviously, is expected!
* ***Care***: In ***POS*** minimization, make sure that **all 0s are accounted for**

DON'T CARE (X) Condition:

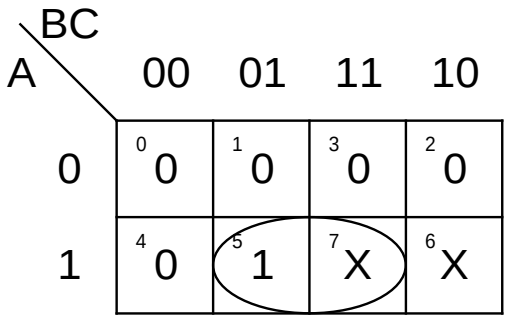
* In digital logic design, there may be ***some states*** which may ***never occur*** in real situations or may be ***completely redundant***

  $\Rightarrow$ These states are referred to as **DON'T CARE** (***X***) states, and can be assigned ***either 0 or 1***, as per our convenience $\Rightarrow$ ***tremendous flexibility***

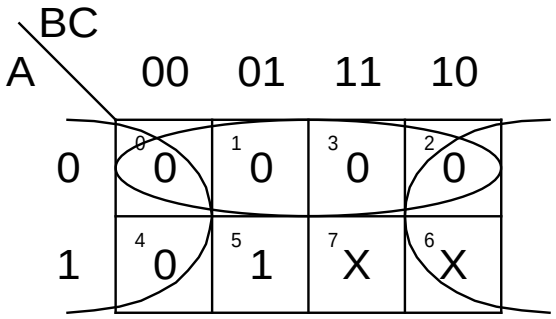
* In ***SOP*** representation, it would be prudent to assign ***1*** to these states

* In ***POS*** representation, it would be prudent to assign ***0*** to these states

Example:



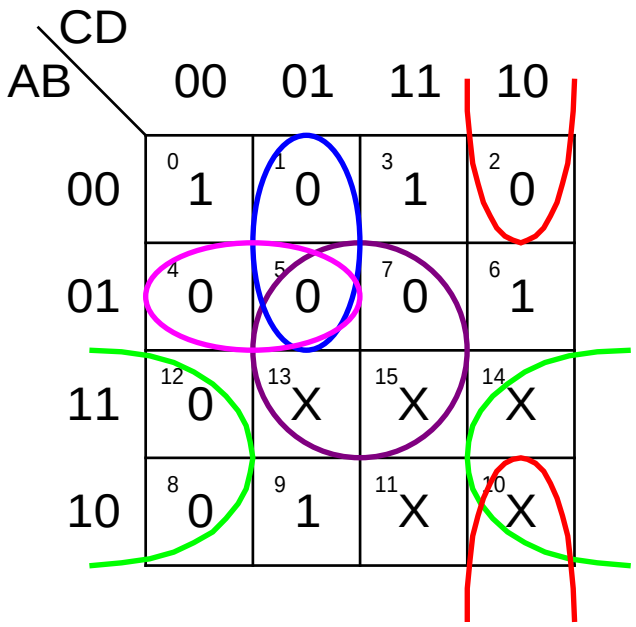
Assigning X = 1 (SOP)
f = AC



Assigning X = 0 (POS)
f = A•C

Note: Both representations yielded the ***same final result***, which obviously, is expected!

More Complicated Example (POS):



f = (A + C + D′)•(A + B′ + C)•
(B + C′ + D)•(A′ + D)•(B′ + D′)

Exercise: Using this expression for f, show that for A = B = C = D = 0, f = 1.