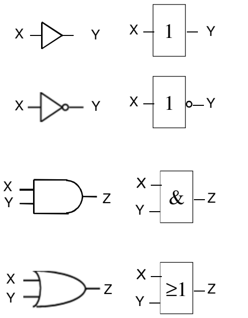
Logic Gates

Logic gates are physical devices, which implement simple Boolean functions.
Some of the simple gates:

ANSI/IEEE-1973 ANSI/IEEE-1984



BUFFER Y=X

X	Y
0	0
1	1

INVERTER (*NOT*) $\mathbf{Y} = \overline{\mathbf{X}}$

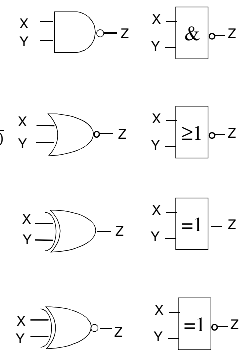
X	Y
0	1
1	0

AND $Z = X \bullet Y$

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

OR $Z = X + Y$

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



NAND (NOT AND) $\mathbf{Z} = \overline{(\mathbf{XY})}$

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

NOR (NOT OR) $\mathbf{Z} = \overline{(\mathbf{X}+\mathbf{Y})}$

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

XOR (Difference) $\mathbf{Z} = \mathbf{X} \oplus \mathbf{Y}$
$\mathbf{Z} = \mathbf{X}\overline{\mathbf{Y}} + \overline{\mathbf{X}}\mathbf{Y}$

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

XNOR (Equality) $\mathbf{Z} = \mathbf{X} \odot \mathbf{Y}$
$\mathbf{Z} = \mathbf{XY} + \overline{\mathbf{X}}\,\overline{\mathbf{Y}}$

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

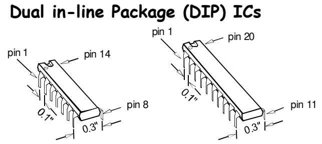
Integrated Circuits – IC

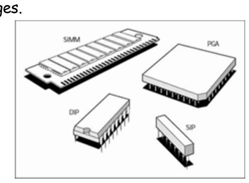
Logic gates are manufactured in integrated circuit (IC) (chip) form.

Recently, a large number of mixed logic gates are packed in a single integrated circuit. For example, an ULSI (Ultra large-scale integration) chip can include more than 100,000 gates.

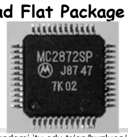
ICs, themselves, come in different types of packages.

Dual in-line Package (DIP) ICs

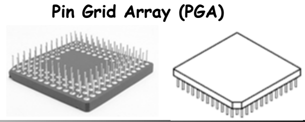




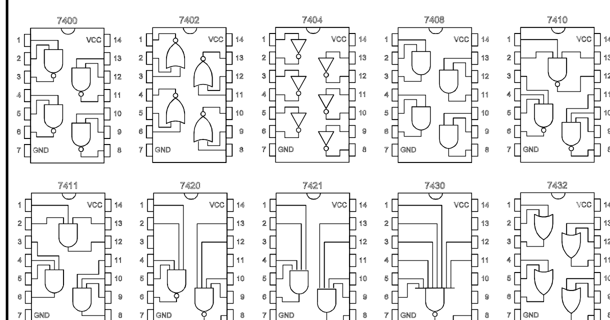
Quad Flat Package (QFP)



Pin Grid Array (PGA)



Examples of 74xx Series



You may find necessary information about ICs in their datasheet catalogs.

Positive and Negative Logic

Boolean values (zero and one) represent physical quantities such as voltage or state of an entity (door is open, light is off).

Assigning "1" to high value, and "0" to low value is called positive logic, and assigning "0" to high value, and "1" to low value is called negative logic.

Example:

Function table of a physical device with 2 inputs and one output is shown below.

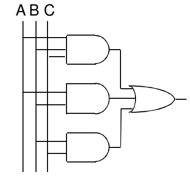
If we use the positive logic, the device can be implemented with an AND gate.

In negative logic system, the device is implemented with an OR gate.

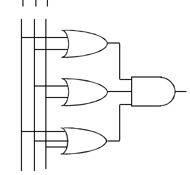
Physical Device			Positive Logic			Negative Logic		
Inputs:		Output:	Inputs:		Output:	Inputs:		Output:
x1	x2	z	x1	x2	z	x1	x2	z
L	L	L	0	0	0	1	1	1
L	H	L	0	1	0	1	0	1
H	L	L	1	0	0	0	1	1
H	H	H	1	1	1	0	0	0

Implementation of Boolean Functions Using Logic Gates

- Sum of Products (SoP)
  - AND gates implement the products.
  - OR gate implements the sum.



- Product of Sums (PoS)
  - OR gates implement the sums.
  - AND gate implements the product.

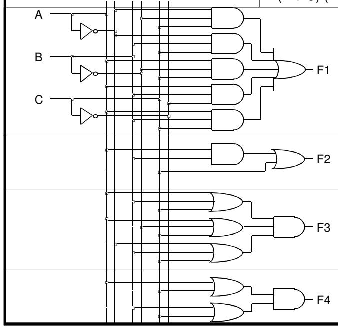


NOT gates can be also used, where they are necessary.

A	B	C	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	1
1	1	1	1

Example: Implementation of a Boolean Function represented (given) by the truth table

F(A, B, C)= Σm(1,3,5,6,7) 1. canonical form
= A'B'C + A'BC + AB'C + ABC' + ABC = F1
= AB+C = F2 (minimized)
F(A, B, C)= ΠM(0,2,4) 2. canonical form
= (A + B + C) (A + B' + C) (A' + B + C) = F3
= (A + C) (B + C) = F4 (minimized)



F1 = F2 = F3 = F4

1. canonical form (SoP)

minimized (SoP)

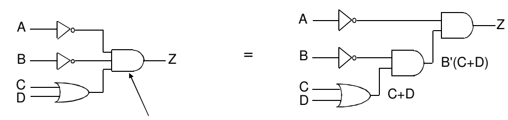
2. canonical form (PoS)

minimized (PoS)

Implementation of Boolean Functions Using Logic Gates (cont'd)

As a Boolean function has many different expressions, it can be implemented in different ways using different logic gates.

Example: Z = A' • B' • (C + D) = (A' • (B' • (C + D))) (Associative Law)



3-input gate

Only 2-input gates

Sometimes it would be necessary to manipulate logical expressions of functions according to currently available gates.

Functional Completeness and Universal Gates

A set of logic operations is said to be **functionally complete**, if any Boolean function can be expressed using only this set of operations.

From the definition of the Boolean algebra {AND, OR, NOT} is obviously a functionally complete set.

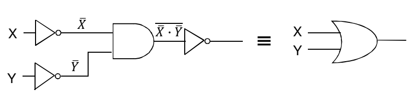
Any function can be expressed in sum-of-products form, and a product-of-sums expression using only the AND, OR, and NOT operations.

Since the set of operations {AND, OR, NOT} is functionally complete, any set of logic gates which can realize {AND, OR, NOT} is also functionally complete.

For example, {AND, NOT} is also a functionally complete set of gates, because OR can be realized using only AND and NOT.

De Morgan's Law:

$\overline{\bar{X} \cdot \bar{Y}} = X + Y$



Universal Logic Gates

If a single gate forms a functionally complete set by itself, then any Boolean function can be realized using only gates of that type.

This type of a gate is called **universal logic gate**.

- The NAND gate is an example of such a gate.
- NOT, AND, and OR can be realized using only NAND gates.
- Thus, any Boolean function can be realized using only NAND gates.
- Similarly, the set consisting only of the binary operator NOR is also functionally complete.
- All other logic functions can be realized using only NOR gates.

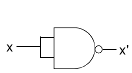
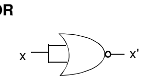
NAND (and also NOR) gates are called **universal logic gates**.

Proof of completeness

To prove that NAND and NOR operators are functionally complete , we have to show that AND, OR, NOT operations can be implemented by using only NAND (or alternatively, NOR) gates.

NAND is denoted by symbol |

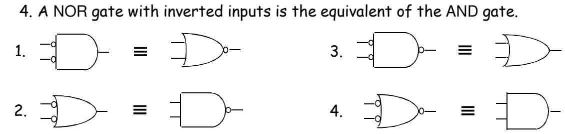
NOR is denoted by symbol ↓

	NAND	NOR
NOT:	x'=x | x = (x·x)' = x' 	x' = x ↓ x = (x+x)' = x' 
AND:	x·y = (x | y)'	x·y = (x' ↓ y') de Morgan x·y = (x' + y')'
OR:	x+y = (x' | y') de Morgan x+y = (x'·y')'	x+y=(x ↓ y)'

Relation between NAND and NOR

- NAND - NOR Conversions
  - de Morgan:
    1. A' • B' = (A + B)'
    2. A' + B' = (A • B)'
    3. (A' • B')' = A + B
    4. (A' + B')' = (A • B)
- These expressions show that,

1. An AND gate with inverted inputs is the equivalent of the NOR gate.
2. An OR gate with inverted inputs is the equivalent of the NAND gate.
3. A NAND gate with inverted inputs is the equivalent of the OR gate.
4. A NOR gate with inverted inputs is the equivalent of the AND gate.



Implementation of Boolean functions using only NAND (NOR) gates

As the binary operator NAND is functionally complete, all Boolean functions can be implemented using only NAND gates.

NOR gates have the same property.

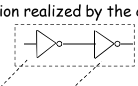
Implementation of Boolean functions in the SOP form using only NAND gates

Shortcut: In circuits in the SOP form, we can replace all AND gates and OR gates with NAND gates. These changes do not affect the output.

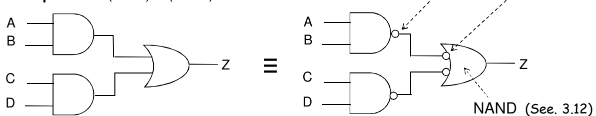
Details:

If we add NOT gates to the outputs of AND gates and to the inputs of the OR gates, we obtain NAND gates. (See 3.12)

If we always add inverters in pairs (NOT-NOT), the function realized by the circuit will not change. (a')' = a (Involution)



Example: Z = (A • B) + (C • D)



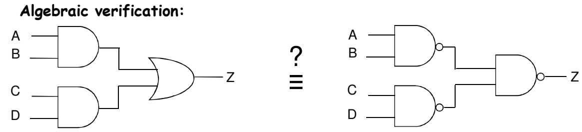
Example (cont'd):

Algebraic conversion:

Expression is inverted twice. (Z')' = Z (Involution)

Z = (A • B) + (C • D) (SoP form)
= [((A • B) + (C • D))']'
= [(A • B)' • (C • D)']' (De Morgan)
= (A | B) | (C | D) (only NAND gates)

Algebraic verification:



Z = [(A • B)' • (C • D)']' Expression with NANDs (right part)
= [(A' + B') • (C' + D')]'
= [(A' + B')' + (C' + D')']
= (A • B) + (C • D) ✓ Expression of the circuit on left

Implementation with gates having limited number of inputs

Sometimes, it is necessary to implement products (or sums) with many literals using gates having only 2 inputs (remember the integrated circuits in 3.4).

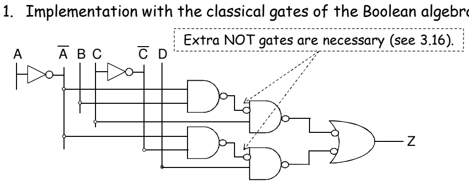
Example:

$Z = \overline{A}BC + \overline{A}\overline{C}D$

Implement this expression using **only 2-input** NAND Gates.

Solution 1:

1. Implementation with the classical gates of the Boolean algebra

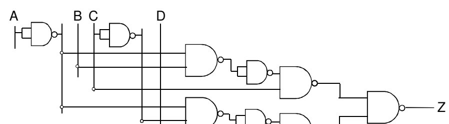


2. Inserting NOT gates to obtain NAND gates

Example (cont'd):

Solution 1:

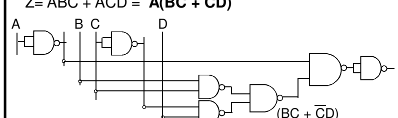
3. Implementation with 2-input NAND gates



Solution 2:

Manipulating the original expression to obtain a <u>simpler</u> circuit

$Z = \overline{A}BC + \overline{A}\overline{C}D = \mathbf{\overline{A}(BC + \overline{C}D)}$



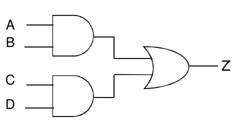
The circuit in solution 2 is cheaper to implement than the circuit in solution 1.

Therefore solution 2 is preferable to solution 1.

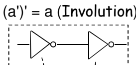
Implementation of Boolean functions in the SOP form using only NOR gates

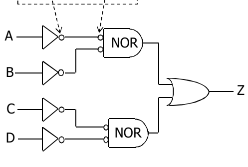
In this case, we must connect extra NOT gates to the inputs and outputs of the circuit.

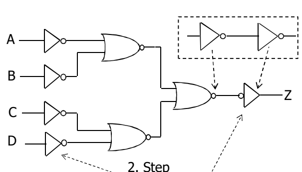
Example: Z = (A • B) + (C • D)



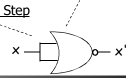
(a')' = a (Involution)







Remember: We can implement NOT gates using NOR gates.



Implementation of Boolean functions in the POS form using only NOR gates

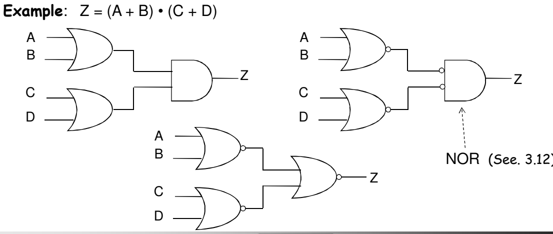
Shortcut: In circuits in the POS form, we can replace all AND gates and OR gates with NOR gates.

Details:

If we add NOT gates to the outputs of OR gates, and to the inputs of the AND gates, we obtain NOR gates. (See 3.12)

Remember: If we always add inverters in pairs, the function realized by the circuit will not change. (a')' = a (Involution)

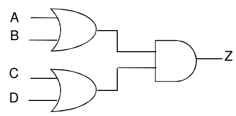
Example: Z = (A + B) • (C + D)

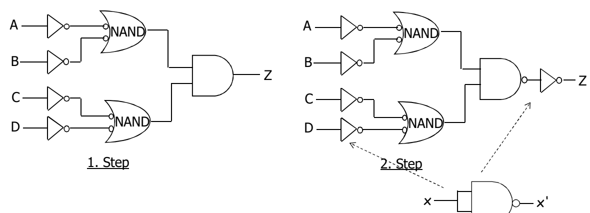


Implementation of Boolean functions in the POS form using only NAND gates

In this case, we must connect extra NOT gates to the inputs and outputs of the circuit.

Example: Z = (A + B) • (C + D)



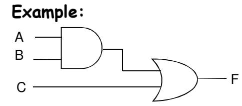


Remember: We can implement NOT gates using NAND gates.

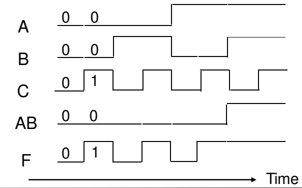
Timing Diagrams

- Truth tables are not enough to show some physical phenomena in digital circuits (for example delays). (Slide 3.21)
  In these cases, we need to use timing diagrams to describe the behavior of the circuit.
- Timing diagrams show various signals in the circuit (indicated as 0/1 or L/H on the vertical axis) as a function of time (on the horizontal axis).
- Several variables are usually plotted with the same time scale so that the times at which these variables change with respect to each other and the relationship between these variables can easily be observed.
- In more detailed timing diagrams, values of the outputs are written in terms of electrical voltage or current.

Example:



In this diagram, we show only logic behavior of the circuit and ignore time delays (described in the coming slides).



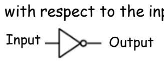
Propagation Delay

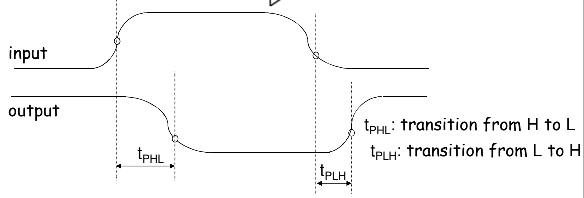
When the input to a logic gate is changed, the output will not change instantaneously.

The transistors and other switching elements within the gate take a finite time to react to a change in input.

The delay in the change in output with respect to the input is called **propagation delay**.

Example:





t_{PHL}: transition from H to L
t_{PLH}: transition from L to H

The shorter the propagation delay, the higher is **the speed** of the logic circuit.

The inputs of a circuit must be kept stable (constant) until it finishes its previous job.

A new input value can be applied only after the previous input value has been processed.

Hazards caused by propagation delays

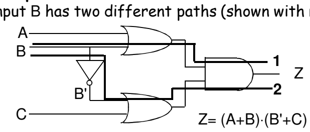
A digital circuit may malfunction due to timing problems.

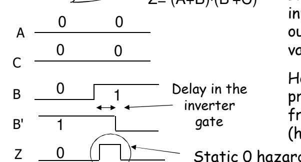
Such timing problems, which arise due to delays are referred to as hazards.

If an input propagates through multiple paths to the output, unexpected output values (hazards) may appear at the output.

Example:

Input B has two different paths (shown with red and yellow lines) to the output Z.





When the inputs A, B, and C are all zero (A=0, B=0, C=0), the output will also be zero (Z=0).

In this state, if the value of input B is changed to 1, the output should not change its value (A=0, B=1, C=0 → **Z=0**).

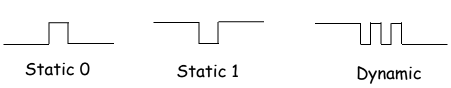
However due to the different propagation delays in the paths from B to Z, a "short" change (hazard) appears at the output.

Types of hazards:

a) Static 0: The output momentarily goes to 1 when it should stay a constant 0.
   The output becomes 1 for a short time and gets back to 0.
   A static 0-hazard may occur in a product-of-sums implementation.

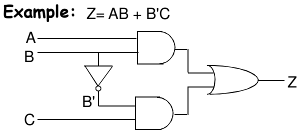
b) Static 1: The output momentarily goes to 0 when it should stay a constant 1.
   The output becomes 0 (from 1) and gets back to 1 after a short time.
   A static 1-hazard may occur in a sum-of-products implementation.

c) Dynamic: When the output is supposed to change from 0 to 1 (or 1 to 0), the output changes three or more times (i.e., oscillates).



Avoiding hazards:

Example: Z= AB + B'C



The circuit on the left (SoP implementation) has output Z=1 when A=1, B=1 and C=1.
In this state if B becomes 0 (1→0), the output should stay as Z=1.
However, **static 1 hazard** occurs at the output.

Possible hazards can be foreseen from the Karnaugh map.

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

Z= AB + B'C

A change in B (1→0) causes a transition from a prime implicant to another. These type of transitions cause hazards because of delays.
To avoid the hazards, **consensus** of the transitions are added to the design which increases the cost.

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

Z= AB + B'C + **AC**