Digital Circuit
&
Logic Design
(CSDC-0101)

Universal Gates

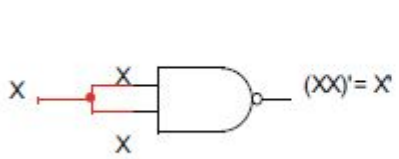
The NAND and NOR gates are called universal gates.

With NANDs or NORs, designers are able to construct other logic gates such as OR, AND, and NOT gates.

SOURCE: HTTPS://DOI.ORG/10.1007/978-3-319-66775-1_1

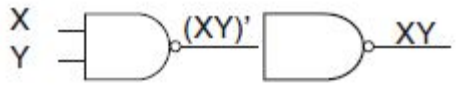
Using NAND Gate

NOT from NAND

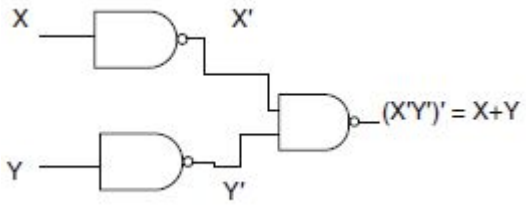


Or, it can be represented by:

AND from NAND

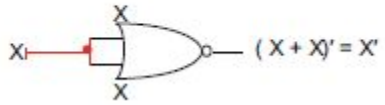


OR from NAND

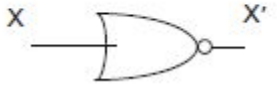


Using NOR Gate

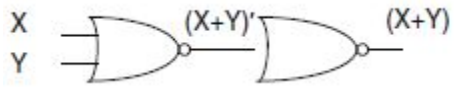
NOT from NOR



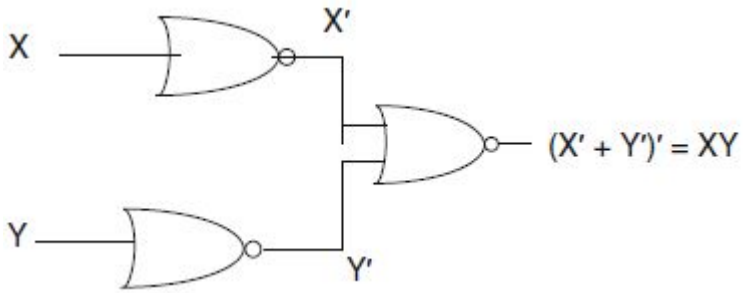
Or, it can be represented by:



AND from NOR



OR from NOR



Implementation of Logic Functions Using NAND Gates or NOR Gates Only

Many logic functions are implemented using only NAND or NOR gates, rather than a combination of various gates. Most logic gate ICs contain multiple gates of a single type. Using a single type of gate can reduce the number of ICs needed.

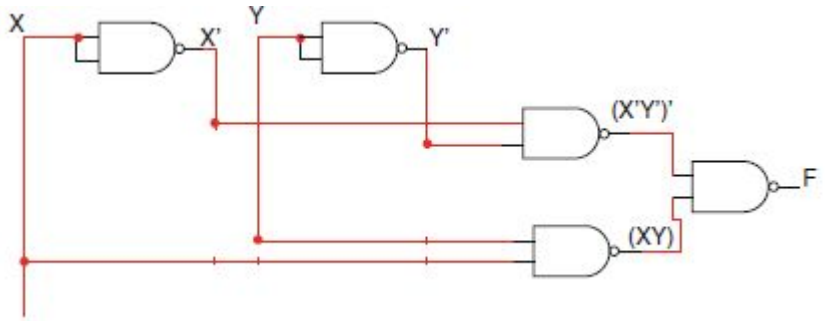
Using NAND Gates

A convenient way to implement a Boolean function with NAND gates is to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic.

- Complement the right side of the equation twice.

  F(X,Y) = X'Y' + XY -> F(X, Y) = [(X'Y'+ XY)']'.
- Use Boolean theorems to make it NAND friendly:

  F(X,Y) = [(X'Y')'(XY)']'



The implementation of Boolean functions with NAND gates requires that the functions be in sum-of-products form.

SOURCE: HTTPS://DOI.ORG/10.1007/978-3-319-66775-1_1

Using NOR Gates

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

The NOR gate is another universal gate that can be used to implement any Boolean function.

- Complement the right side of the equation twice.

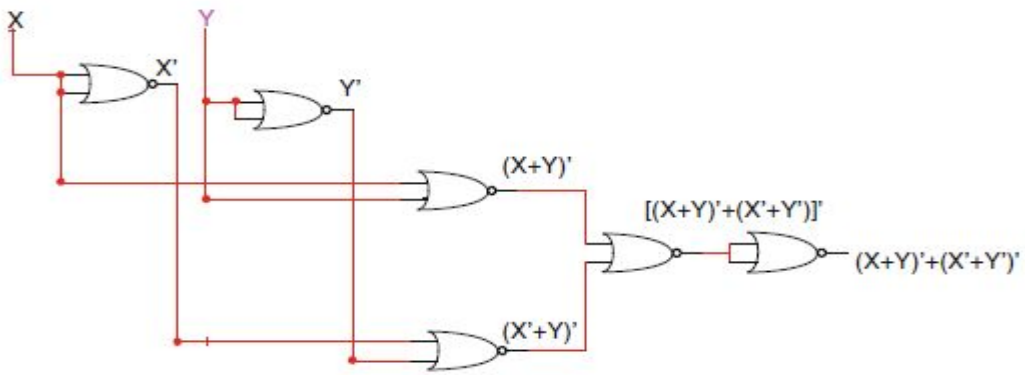
  F(X,Y) = X'Y' + XY -> F(X, Y) = [(X'Y'+ XY)']'.

- Use Boolean theorems to make it NOR friendly:

  F(X,Y) = [(X'Y')'(XY)']'

  F(X,Y) = [(X+Y)(X'+Y')]'

  F(X,Y) = [(X+Y)' + (X' +Y')']



SOURCE: HTTPS://DOI.ORG/10.1007/978-3-319-66775-1_1

Other Two-Level Implementations

The standard form of expressing Boolean functions results in a **two-level implementation**.

NAND and NOR logic implementations are the most important and are most often found in integrated circuits.

Some (but not all) 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**.

The AND gate is drawn with the lines going through the center of the gate to distinguish it from a conventional gate. The wired-AND gate is not a physical gate, but only a symbol to designate the function obtained from the indicated wired connection.

$$\text{Example, } F = (AB)' \cdot (CD)' = (AB + CD)' = (A' + B')(C' + D')$$

and is called an **AND–OR–INVERT function**.

Similarly, the NOR outputs of Emitter-coupled Logic (ECL) gates can be tied together to perform a wired-OR function.

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

and is called an **OR–AND–INVERT function**.

SOURCE: HTTPS://DOI.ORG/10.1007/978-3-319-66775-1_1

Nondegenerate Forms

To find out how many two-level combinations of gates are possible:

Considering four types of gates: AND, OR, NAND, and NOR.

- If we assign one type of gate for the first level and one type for the second level, we find that there are 16 possible combinations of two-level forms. (The same type of gate can be in the first and second levels) Eight of these combinations are said to be **degenerate forms** because they degenerate to a single operation.
- The remaining eight **nondegenerate forms** produce an implementation in sum-of-products form or product-of-sums form. The eight nondegenerate forms are as follows:

AND–OR	OR–AND
NAND–NAND	NOR–NOR
NOR–OR	NAND–AND
OR–NAND	AND–NOR

- The first gate listed in each of the forms constitutes a first level in the implementation.
- The second gate listed is a single gate placed in the second level.

Note that any two forms listed on the same line are duals of each other.

SOURCE: HTTPS://DOI.ORG/10.1007/978-3-319-66775-1_1

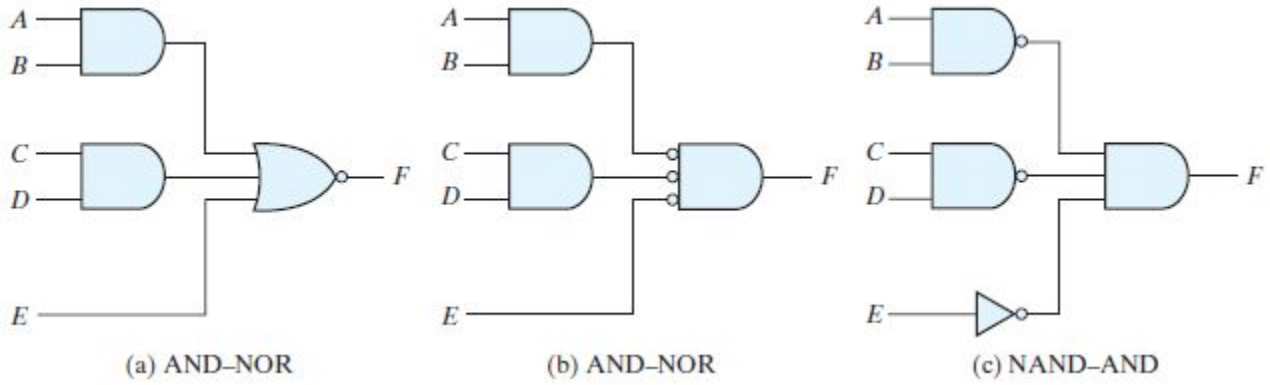
AND–OR–INVERT Implementation

The two forms, NAND–AND and AND–NOR, are equivalent and can be treated together.

Both perform the AND–OR–INVERT function

The AND–NOR form resembles the AND–OR form, but with an inversion done by the bubble in the output of the NOR gate. It implements the function

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



Note that the single variable E is not complemented, because the only change made is in the graphic symbol of the NOR gate.

Now we move the bubble from the input terminal of the second-level gate to the output terminals of the first-level gates. An inverter is needed for the single variable in order to compensate for the bubble. Alternatively, the inverter can be removed, provided that input E is complemented.

An AND–OR implementation requires an expression in sum-of-products form. The AND–OR–INVERT implementation is similar, except for the inversion. Therefore, if the complement of the function is simplified into sum-of-products form (by combining the 0's in the map), it will be possible to implement F' with the AND–OR part of the function. When F' passes through the always present output inversion (the INVERT part), it will generate the output F of the function.

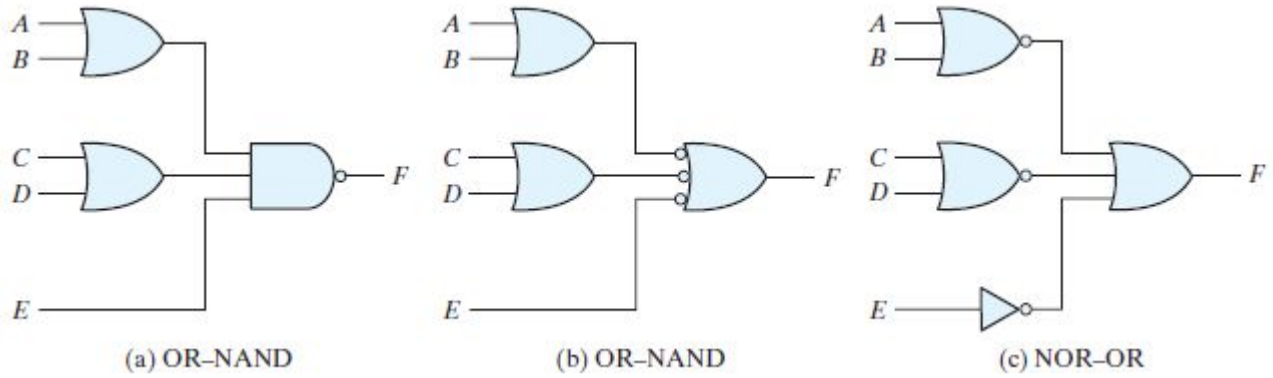
SOURCE: HTTPS://DOI.ORG/10.1007/978-3-319-66775-1_1

OR–AND–INVERT Implementation

The OR–NAND and NOR–OR forms perform the OR–AND–INVERT function.

The OR–NAND form resembles the OR–AND form, except for the inversion done by the bubble in the NAND gate. It implements the function

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



SOURCE: HTTPS://DOI.ORG/10.1007/978-3-319-66775-1_1

By using the alternative graphic symbol for the NAND gate and by moving the small circles from the inputs of the second-level gate to the outputs of the first-level gates, a NOR–OR form to implement the OR–AND–INVERT function.

The OR–AND–INVERT implementation requires an expression in product-of-sums form. If the complement of the function is simplified into that form, we can implement F’ with the OR–AND part of the function. When F’ passes through the INVERT part, we obtain the complement of F’, or F , in the output.

SOURCE: HTTPS://DOI.ORG/10.1007/978-3-319-66775-1_1

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

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

Implementation with Other Two-Level Forms

SOURCE: HTTPS://DOI.ORG/10.1007/978-3-319-66775-1_1

Exclusive-OR Function

The exclusive-OR (XOR), denoted by the symbol $\oplus$, is a logical operation that performs the following Boolean operation:

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

Only a limited number of Boolean functions can be expressed in terms of exclusive-OR operations.

It is particularly useful in arithmetic operations and error detection and correction circuits.

The exclusive-NOR, also known as equivalence, performs the following Boolean operation:

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

The exclusive-NOR can be shown to be the complement of the exclusive-OR by means of a truth table or by algebraic manipulation:

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

SOURCE: HTTPS://DOI.ORG/10.1007/978-3-319-66775-1_1

The following identities apply to the exclusive-OR operation:

$$x \oplus 0 = x$$

$$x \oplus 1 = x'$$

$$x \oplus x = 0$$

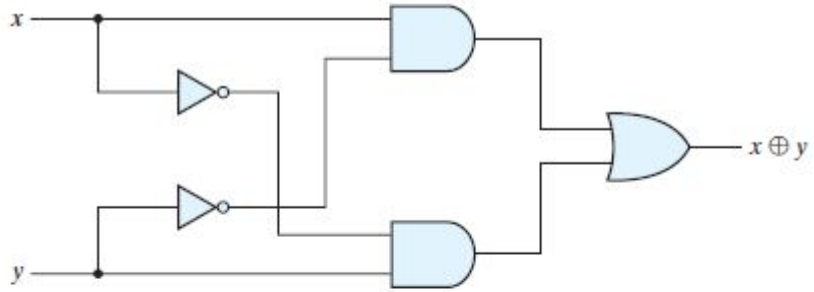
$$x \oplus x' = 1$$

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

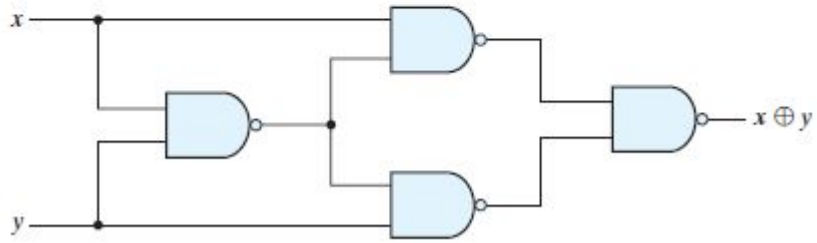
Also, it can be shown that the exclusive-OR operation is both commutative and associative; that is,

$$A \oplus B = B \oplus A \text{ and}$$

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



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



(b) Exclusive-OR with NAND gates

Odd Function:

The exclusive-OR operation with three or more variables can be converted into an ordinary Boolean function by replacing the $\oplus$ symbol with its equivalent Boolean expression. Example,

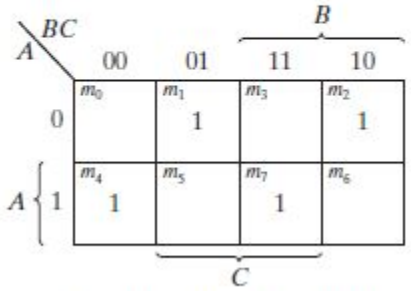
$$A \oplus B \oplus C = (AB'+A'B)C' + (AB+A'B')C$$

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

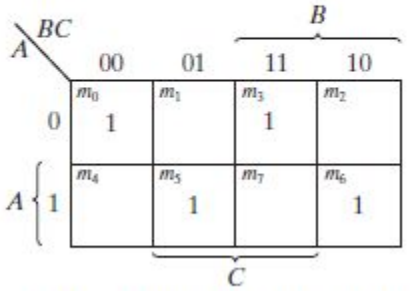
$$= \sum(1, 2, 4, 7)$$

Contrary to the two-variable case, in which only one variable must be equal to 1, in the case of three or more variables the requirement is that an odd number of variables be equal to 1. As a consequence, the multiple-variable exclusive-OR operation is defined as an *odd function*. The complement of an odd function is an *even function*.

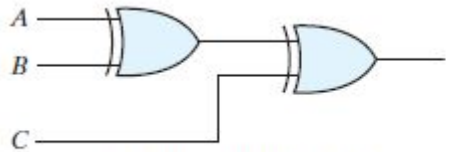
In general, an n -variable exclusive-OR function is an odd function defined as the logical sum of the $2n>2$ minterms whose binary numerical values have an odd number of 1's. Example, The Boolean function derived from the three-variable exclusive-OR operation is expressed as the logical sum of four minterms whose binary numerical values are 001, 010, 100, and 111. Each of these binary numbers has an odd number of 1's.



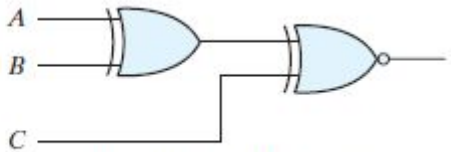
(a) Odd function $F = A \oplus B \oplus C$



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

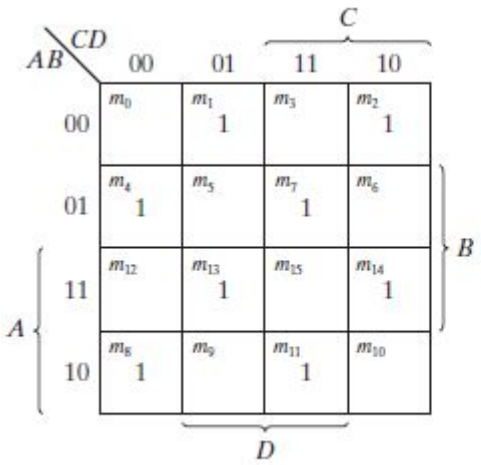


(a) 3-input odd function

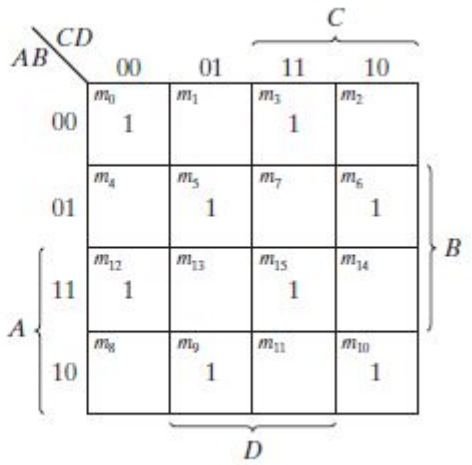


(b) 3-input even function

$$
\begin{aligned}
A \oplus B \oplus C \oplus D &= (AB' + A'B) \oplus (CD' + C'D) \\
&= (AB' + A'B)(CD + C'D') + (AB + A'B')(CD' + C'D) \\
&= \Sigma(1, 2, 4, 7, 8, 11, 13, 14)
\end{aligned}
$$



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



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

Parity Generation and Checking:

Exclusive-OR functions are very useful in systems requiring error detection and correction codes.

A parity bit is used for the purpose of detecting errors during the transmission of binary information.

A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even. The message, including the parity bit, is transmitted and then checked at the receiving end or errors. An error is detected if the checked parity does not correspond with the one transmitted.

The circuit that generates the parity bit in the transmitter is called a *parity generator*. The circuit that checks the parity in the receiver is called a *parity checker*.

For *even parity*, the bit P must be generated to make the total number of 1's (including P) even.

Even-Parity-Generator Truth Table

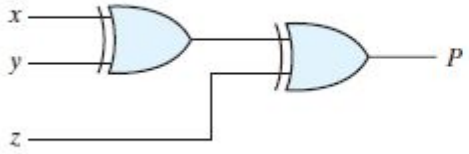
Three-Bit Message			Parity Bit
x	*y*	*z*	*P*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$P = x \oplus y \oplus z$

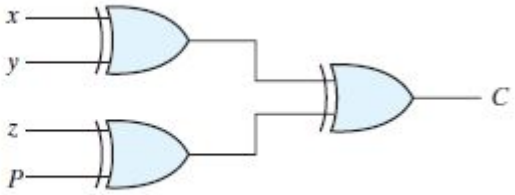
Even-Parity-Checker Truth Table

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

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



(a) 3-bit even parity generator



(b) 4-bit even parity checker