

# Quine-McCluskey Method

---

## ❖ Simplification of logic expressions using Quine-McCluskey Method (Tabulation Method)

- Deriving “Prime Implicants”
- Deriving “Essential Prime Implicants”

## ❖ Using NAND and NOR gates as universal gates

- Implementation of logical gates using universal gates
- Converting logical expressions to NAND and NOR forms using Boolean algebra

# Simplification of logic expressions using Quine-McCluskey Method (Tabulation Method)

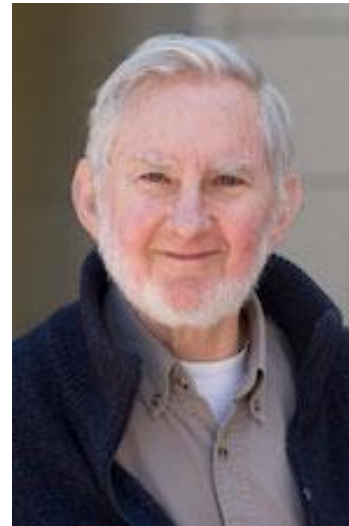
---

Quine-McCluskey method (A.K.A. Tabulation Method) is an algorithmic approach to simplification of logic circuits which has many (especially more than 5) variables.

First, we group minterms according to the number of 1's they have. We include the don't care situations when grouping.



Willard Van Orman Quine



Edward J. McCluskey

## Simplification of logic expressions using Quine-McCluskey Method (Tabulation Method)

---

Then, we combine neighbor minterms by putting a dash (–) in place of the different bit and keep the others as they are. And we mark the grouped minterms with a tick (✓).

The terms which don't have a neighbor are the prime implicants. We also combine the previously combined terms if possible. We continue combining until there's nothing more to combine.

Then, we continue to step 2 to find out the essential prime implicants.

## Example

**Example:** Let's simplify the expression below using tabulation method.

$$f(A,B,C,D)=\sum(4,8,10,11,12,15)$$

- First, we group minterms according to the number of 1's they contain.

Number of 1's in the terms	Minterms	Binary Representation of Minterms
1	$m_4$	0100
	$m_8$	1000
2	$m_{10}$	1010
	$m_{12}$	1100
3	$m_{11}$	1011
4	$m_{15}$	1111

## Example

Number of 1's in the terms	Minterms	Binary Values of Minterms	Grouped Minterms	Binary Values of Grouped Minterms
1	m <sub>4</sub>	0100 ✓	m(4,12)	-100
	m <sub>8</sub>	1000 ✓	m(8,10)	10-0
2	m <sub>10</sub>	1010 ✓	m(8,12)	1-00
	m <sub>12</sub>	1100 ✓	m(10,11)	101-
3	m <sub>11</sub>	1011 ✓	m(11,15)	1-11
4	m <sub>15</sub>	1111 ✓		

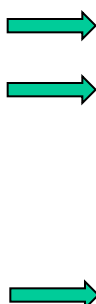


**Unchecked binary values are the prime implicants**

## Example

Now we need to find out the “essential prime implicants” out of “prime implicants”. In the essential prime implicants table, the rows consist of the prime implicants, and the columns consist of the minterms in the expression. We put an “X” mark to the cells where prime implicants contain the corresponding minterm. Then, we scan all the columns and mark the cells where there’s only one “X” mark.

	Prime Implicants	4	8	10	11	12	15
m(4,12)	-100 (BC'D')	X				X	
m(8,10)	10-0 (AB'D')		X	X			
m(8,12)	1-00 (AC'D')		X			X	
m(10,11)	101- (AB'C)			X	X		
m(11,15)	1-11 (ACD)				X		X



}

**Essential Prime  
Implicants**

$$F(A,B,C,D) = BC'D' + ACD + AB'D'$$

## Example

Let's create the K-Map of the same expression and find out all the possible groups that can be created. We can see that, these groups are the prime implicants we have found in the tabulation method. And, the prime implicants are the groups we select to form the simplified expression.

AB \ CD	00	01	11	10
00		1	1	1
01				
11			1	1
10				1

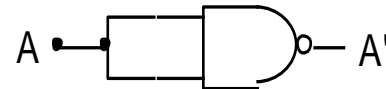
The gray and brown groups are essential prime implicants because they both have a “1” that is covered only in that group. These minterms are  $m_4$  and  $m_{15}$ . The other essential prime implicant is the blue group.

# Using NAND and NOR Gates as Universal Gates

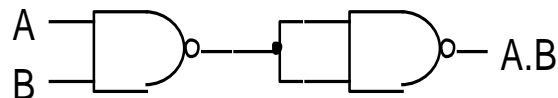
We have built combinational circuits using NOT, AND and OR gates so far. In fact, it is possible to implement any gate using only NAND gates or only NOR gates. Thus, we can implement logic circuits using a single type of gate and in a more economic way.

## Implementing other gates using NAND gates

- Considering that  $(A.A)' = A'$ , we can implement a NOT gate as depicted below.



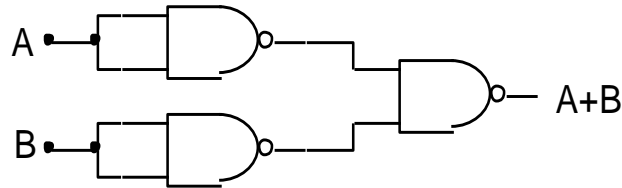
- Considering that  $((A.B)')' = A.B$ , we can implement an AND gate as depicted below.



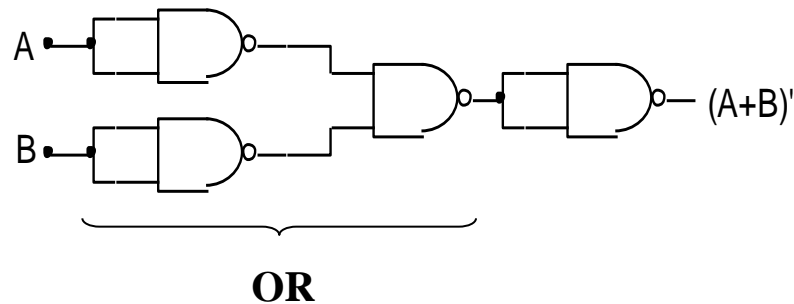


## Using NAND and NOR Gates as Universal Gates

- Considering that  $(A'.B')' = A+B$ , we can implement an OR gate as depicted below.



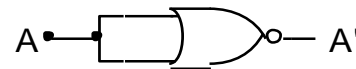
- Considering that  $[(A'.B')']' = (A+B)'$ , we can implement a NOR gate as depicted below.



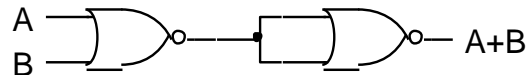
# Using NAND and NOR Gates as Universal Gates

## Implementing other gates using NOR gates

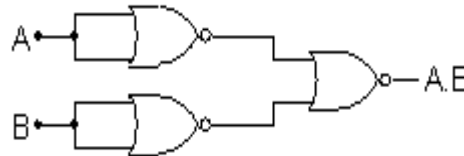
- Considering that  $(A+A)' = A'$ , we can implement a NOT gate as depicted below.



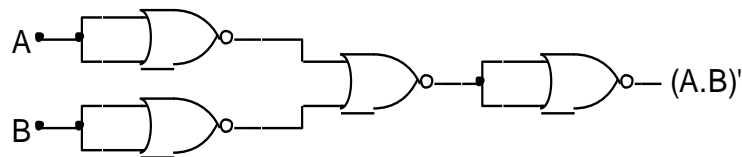
- Considering that  $((A+B)')' = A+B$ , we can implement an OR gate as depicted below.



- Considering that  $(A'+B')' = A.B$ , we can implement an AND gate as depicted below.

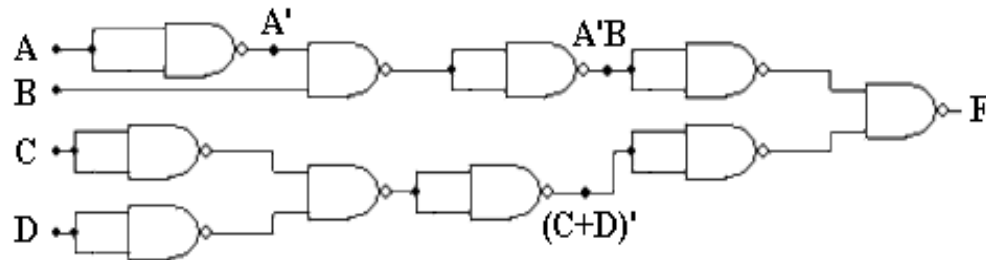


- Considering that  $[ (A'+B')' ]' = (A.B)'$ , we can implement a NAND gate as depicted below.

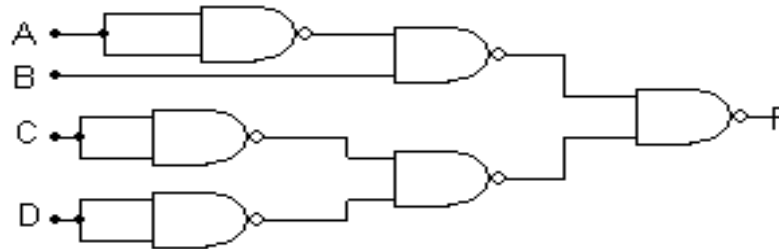


# Implementing Logic Expressions with NAND Gates

**Example:** Let's implement  $F(A,B,C,D) = A'B + (C+D)'$  using only NAND gates. What we need to do is to replace NOT, AND, OR and NOR gates with their NAND gate equivalents.



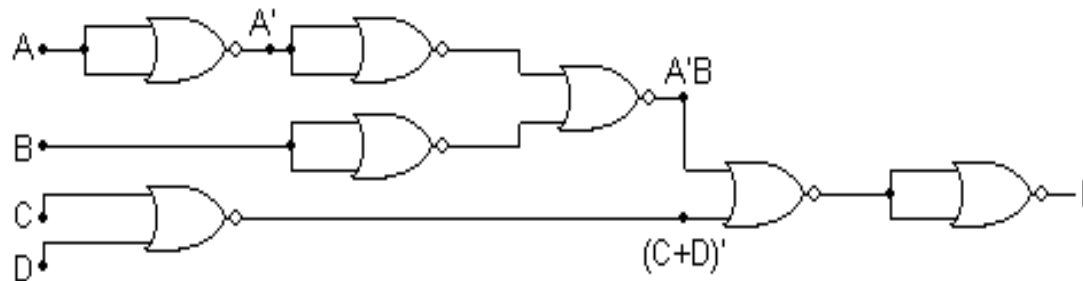
Then, we eliminate sequential NOT gates.



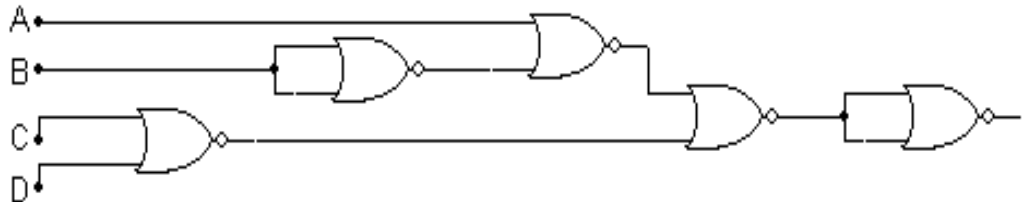
# Implementing Logic Expressions with NOR Gates

**Example:** Let's implement the same expression using NOR gates.

$$F(A,B,C,D) = A'B + (C+D)'$$



Again, we eliminate sequential NOT gates.



# Converting Logic Expressions to NAND and NOR Forms using Boolean Algebra

---

We can convert logic expressions to NAND or NOR form using DeMorgan rules.

- By double negating a logic expression in SoP form, we can obtain the NAND form of that expression.
- By double negating a logic expression in PoS form, we can obtain the NOR form of that expression.

**Example:** Let's convert  $f(a,b,c,d) = ab' + c'd$  to NAND form.

$$(f')' = f. \text{ So, } f = [(ab' + c'd)']' = [(ab')' \cdot (c'd)']'$$

**Example:** Let's convert  $f(a,b,c,d) = (a+b')(c'+d)$  to NOR form.

$$(f')' = f. \text{ So, } f = [(a+b')(c'+d)]' = [(a+b')' + (c'+d)']'$$

# Converting Logic Expressions to NAND and NOR Forms using Boolean Algebra

---

**Example:** Let's convert  $f(a,b,c,d) = ab' + c'd$  to NOR form.

What we need to do first, is to double negate each term in the expression.

$$f(a,b,c,d) = [(ab')']' + [(c'd)']' = (a' + b)' + (c + d')'$$

Then, we need to double negate the whole expression.

$$f(a,b,c,d) = ([ (a' + b)' + (c + d')' ]')'$$

**Example:** Let's convert  $f(a,b,c,d) = (a+b')(c'+d)$  to NAND form.

Again, what we need to do first, is to double negate each term in the expression.

$$f(a,b,c,d) = (a+b')(c'+d) = ([ (a+b') ]')' . ([ (c'+d) ]')' = (a'b)' . (cd')'$$

Then, we need to double negate the whole expression.

$$f(a,b,c,d) = ([ (a'b)' . (cd')' ]')'$$

# Converting Logic Expressions Given as Minterms to NAND and NOR Forms Using Karnaugh Maps

**Example:** Let's implement  $f(a,b,c) = \sum(0,1,3,5,6,7)$  using NAND and NOR gates.

First, we create the K-Map.

$a \backslash bc$	00	01	11	10
0	1	1	1	
1		1		1

The simplified logic expression will be  $f(a,b,c) = c + a'b' + ab$ . We need to double negate the whole expression to convert it to NAND form.

$$f(a,b,c) = [(c + a'b' + ab)']' = [c' \cdot (a'b')' \cdot (ab)']'$$

In real life, we use 2-input NAND gates to implement the circuits. So, we need to double negate 2<sup>nd</sup> and 3<sup>rd</sup> terms inside the brackets.

$$f(a,b,c) = [c' \cdot (a'b')' \cdot (ab)']' = [c' \cdot ((a'b')' \cdot (ab)')]'$$

# Converting Logic Expressions Given as Minterms to NAND and NOR Forms Using Karnaugh Maps

To implement  $f$  using NOR gates in a practical way, we first create the K-Map of  $f'$ , then negate it to find out  $f$  in PoS form.

$f$

$a \backslash bc$	00	01	11	10
0	1	1	1	
1		1	1	1

$f'$

$a \backslash bc$	00	01	11	10
0				1
1	1			

$$f'(a,b,c) = ab'c' + a'bc'$$

$$f = (f')' = (a' + b + c).(a + b' + c)$$

Then we double negate the expression in PoS form to convert it to NOR form.

$$f = [(a' + b + c)' + (a + b' + c)']'$$

Then, we convert it to 2-input NOR gates form.

$$f = ([((a' + b)')' + c]' + [((a + b')')' + c]')'$$