

Boolean Logic

Outline

- Digital Logic and Logic Gates
- Boolean Algebra
- Truth Tables
- Minterm and Maxterm expansions
- Gate level minimisation
- Karnaugh maps
- NAND and NOR implementations

Digital Logic

- The binary numbering system maps well to logical expressions.
- Conventionally we refer to the pair of binary variables as '0' and '1'. We can also label the pair of variables as 'true' and 'false', 'yes' and 'no', etc.
- By combining binary variables with **logical operators** such as AND, OR and NOT, a logical algebra can be examined.

Digital Logic

- Logical operators can be understood in terms of true and false statements.

e.g.	X.	'The sky is blue'	TRUE	1
	Y.	'4 is an odd number'	FALSE	0

$X \text{ AND } Y \Rightarrow \text{FALSE}$

$$X \bullet Y = 0$$

Digital Logic

X.	'The sky is blue'	TRUE	1
Y.	'4 is an odd number'	FALSE	0

$$X \text{ OR } Y \Rightarrow \text{TRUE}$$

$$X + Y = 1$$

Digital Logic

X.	'The sky is blue'	TRUE	1
Y.	'4 is an odd number'	FALSE	0

NOT $X \Rightarrow$ FALSE

$$\overline{X} = 0$$

$$X' = 0$$

Logic Functions

- Computers take inputs and produce outputs, just like functions in math!
- Mathematical logic functions can be expressed in two ways:
Boolean Expression/Function and **Truth Table**

$$\begin{aligned}f(x,y) &= 2x + y \\ &= x + x + y \\ &= 2(x + y/2) \\ &= \dots\end{aligned}$$

x	y	f(x,y)
0	0	0
...
2	2	6
...
23	41	87
...

- **Boolean expression** can be easily converted into **Truth Table**, or vice versa.
- A truth table shows **all possible** inputs and outputs of a function.
- Remember that each input variable represents either 1 or 0.
 - Because there are only a finite number of values (1 and 0), truth tables themselves are finite.
 - A function with n variables has 2^n possible combinations of inputs.
- Inputs are listed in binary order—in this example, from 000 to 111.

$$f(x,y,z) = (x + y')z + x'$$



$$\begin{array}{llll} f(0,0,0) & = & (0 + 1)0 + 1 & = 1 \\ f(0,0,1) & = & (0 + 1)1 + 1 & = 1 \\ f(0,1,0) & = & (0 + 0)0 + 1 & = 1 \\ f(0,1,1) & = & (0 + 0)1 + 1 & = 1 \\ f(1,0,0) & = & (1 + 1)0 + 0 & = 0 \\ f(1,0,1) & = & (1 + 1)1 + 0 & = 1 \\ f(1,1,0) & = & (1 + 0)0 + 0 & = 0 \\ f(1,1,1) & = & (1 + 0)1 + 0 & = 1 \end{array}$$



x	y	z	f(x,y,z)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Basic Boolean Operations

Operation:

AND (product)
of two inputs

OR (sum) of
two inputs

NOT
(complement)
on one input

Expression:

xy , or $x \cdot y$

$x + y$

x' or \bar{x}

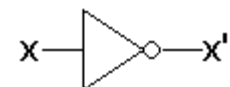
Truth table:

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

x	x'
0	1
1	0

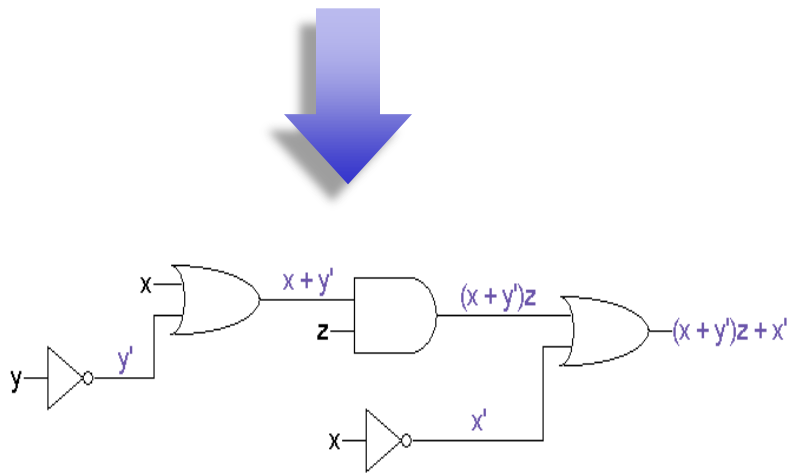
Logic gate:



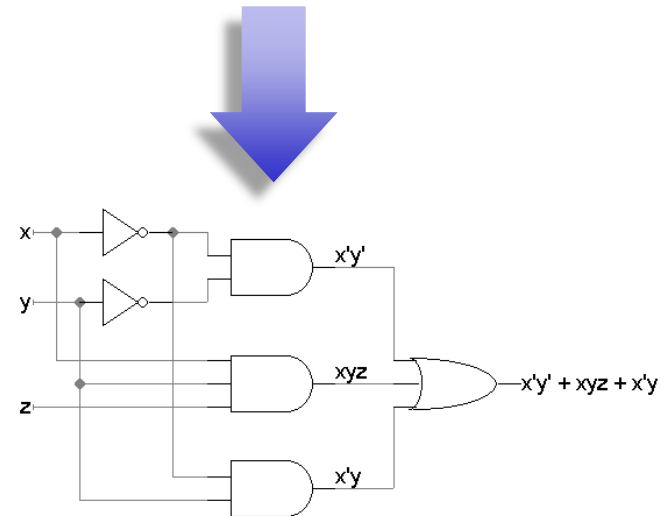
Each of the basic operations can be implemented in hardware using a **logic gate**. Symbols for each of the logic gates are shown above.

- Any Boolean expression can be converted into a **circuit** by combining basic gates in a relatively straightforward way.
- The diagram below shows the inputs and outputs of each gate.

$$(x + y')z + x'$$



$$x'y' + xyz + x'y$$



Can we make these circuits “better”?

- Cheaper: fewer gates
- Faster: fewer delays from inputs to outputs

Expression Simplification

- Normal mathematical expressions can be simplified using the laws of algebra
- For binary systems, we can use **Boolean algebra**, which is superficially similar to regular algebra
- But there are many differences, due to
 - having only two values (0 and 1) to work with
 - having a complement operation
 - the OR operation is not the same as addition

Formal definition of Boolean algebra

- A Boolean algebra requires
 - A set of elements **B**, which needs *at least* two elements (0 and 1)
 - Two binary (two-argument) operations OR and AND
 - A unary (one-argument) operation NOT
 - The **axioms** below must always be true
 - The **magenta axioms** deal with the complement operation
 - **Blue axioms** (especially 15) are different from regular algebra

$$1. x + 0 = x$$

$$2. x \bullet 1 = x$$

$$3. x + 1 = 1$$

$$4. x \bullet 0 = 0$$

$$5. x + x = x$$

$$6. x \bullet x = x$$

$$7. x + x' = 1$$

$$8. x \bullet x' = 0$$

$$9. (x')' = x$$

$$10. x + y = y + x$$

$$11. xy = yx$$

Commutative

$$12. x + (y + z) = (x + y) + z$$

$$13. x(yz) = (xy)z$$

Associative

$$14. x(y + z) = xy + xz$$

$$15. x + yz = (x + y)(x + z)$$

Distributive

$$16. (x + y)' = x'y'$$

$$17. (xy)' = x' + y'$$

DeMorgan's

- We can do function simplifications using Boolean algebra

$$\begin{aligned}
 & x' y' + x y z + x' y \\
 &= x' (y' + y) + x y z \quad [\text{Distributive; } x' y' + x' y = x' (y' + y)] \\
 &= x' \cdot 1 + x y z \quad [\text{Axiom 7; } y' + y = 1] \\
 &= x' + x y z \quad [\text{Axiom 2; } x' \cdot 1 = x'] \\
 &= (x' + x)(x' + y z) \quad [\text{Distributive}] \\
 &= 1 \cdot (x' + y z) \quad [\text{Axiom 7; } x' + x = 1] \\
 &= x' + y z \quad [\text{Axiom 2}]
 \end{aligned}$$

1. $x + 0 = x$

2. $x \cdot 1 = x$

3. $x + 1 = 1$

4. $x \cdot 0 = 0$

5. $x + x = x$

6. $x \cdot x = x$

7. $x + x' = 1$

8. $x \cdot x' = 0$

9. $(x')' = x$

10. $x + y = y + x$

11. $xy = yx$

Commutative

12. $x + (y + z) = (x + y) + z$

13. $x(yz) = (xy)z$

Associative

14. $x(y + z) = xy + xz$

15. $x + yz = (x + y)(x + z)$

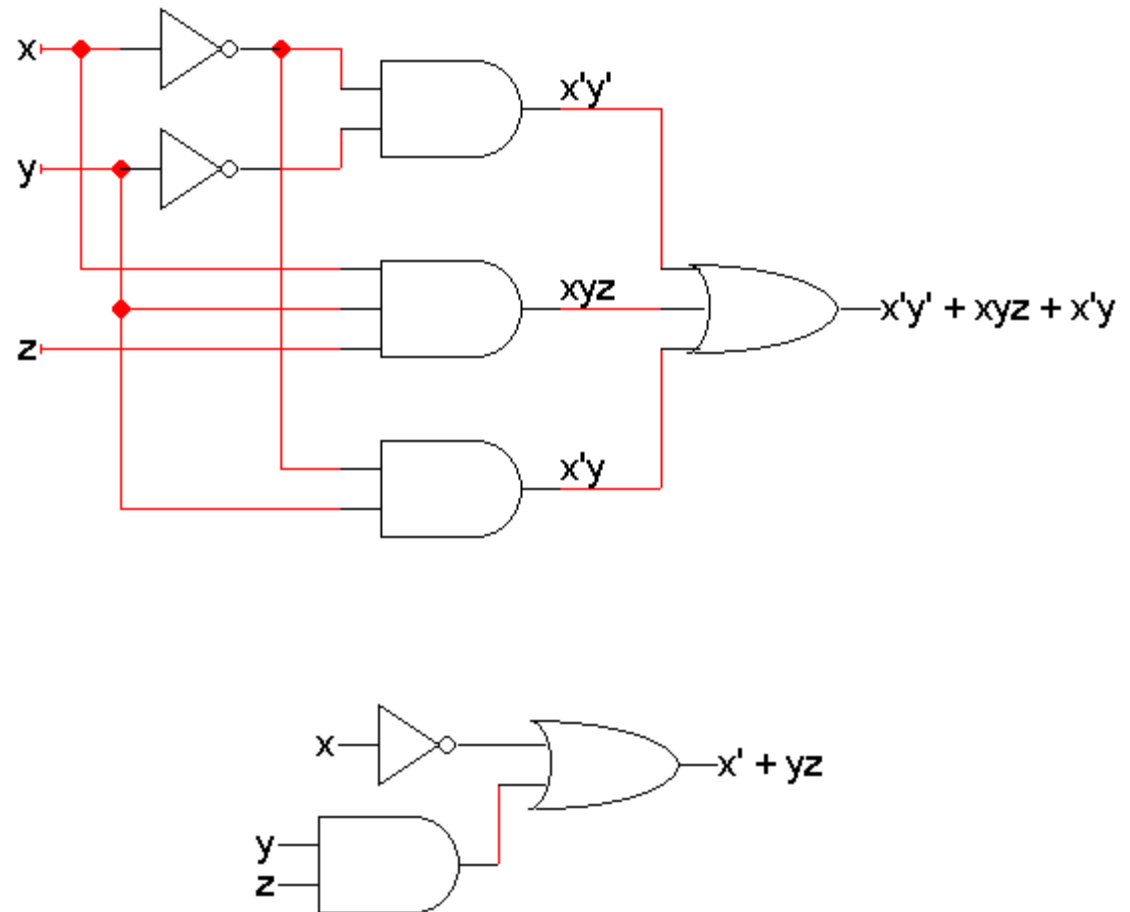
Distributive

16. $(x + y)' = x' y'$

17. $(xy)' = x' + y'$

DeMorgan's

- Here are two different but *equivalent* circuits.
- In general the one with fewer gates is “better”:
 - It costs less to build
 - It requires less power
 - But we have to do some work to find the second form

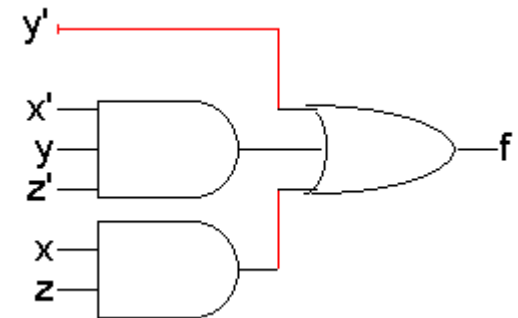


Sum of Products

- We can write expressions in many ways, but some ways are more useful than others
- A **sum of products (SOP)** expression contains:
 - Only OR (sum) operations at the “outermost” level
 - Each term that is summed must be a product of literals

$$f(x,y,z) = y' + x'yz' + xz$$

- The advantage is that any sum of products expression can be implemented using a **two-level circuit**
 - literals and their complements at the “0th” level
 - AND gates at the first level
 - a single OR gate at the second level



Minterms

- A **minterm** is a special product of literals, in which each input variable appears exactly once.
- A function with n variables has 2^n minterms (since each variable can appear complemented or not)
- A three-variable function, such as $f(x,y,z)$, has $2^3 = 8$ minterms:

$$\begin{array}{cccc} x'y'z' & x'y'z & x'yz' & x'yz \\ xy'z' & xy'z & xyz' & xyz \end{array}$$

- Each minterm is *true* for exactly one combination of inputs:

Minterm	Is true when...	Shorthand
$x'y'z'$	$x=0, y=0, z=0$	m_0
$x'y'z$	$x=0, y=0, z=1$	m_1
$x'yz'$	$x=0, y=1, z=0$	m_2
$x'yz$	$x=0, y=1, z=1$	m_3
$xy'z'$	$x=1, y=0, z=0$	m_4
$xy'z$	$x=1, y=0, z=1$	m_5
xyz'	$x=1, y=1, z=0$	m_6
xyz	$x=1, y=1, z=1$	m_7

“0” corresponds to “complement”
“1” corresponds to “original”

Sum of Minterms form

- Every function can be written as a **sum of minterms**, which is a special kind of sum of products form
- The sum of minterms form for any function is *unique*
- If you have a truth table for a function, you can write a sum of **minterms** expression just by picking out the rows of the table where the function **output is 1**.

x	y	z	f(x,y,z)	f'(x,y,z)
0	0	0	1	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	0	1

$$\begin{aligned}
 f &= x'y'z' + x'y'z + x'yz' + x'yz + xyz' \\
 &= m_0 + m_1 + m_2 + m_3 + m_6 \\
 &= \Sigma m(0,1,2,3,6)
 \end{aligned}$$

$$\begin{aligned}
 f' &= xy'z' + xy'z + xyz \\
 &= m_4 + m_5 + m_7 \\
 &= \Sigma m(4,5,7)
 \end{aligned}$$

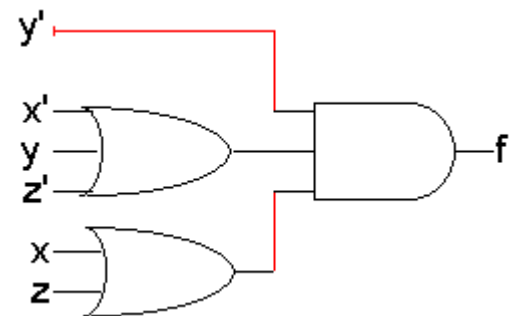
f' contains all the minterms not in f

Product of Sums

- A **product of sums (POS)** expression contains:
 - Only AND (product) operations at the “outermost” level
 - Each term must be a sum of literals

$$f(x,y,z) = y' (x' + y + z') (x + z)$$

- Product of sums expressions can also be implemented with two-level circuits
 - literals and their complements at the “0th” level
 - *OR gates* at the first level
 - a single *AND gate* at the second level



Maxterms

- A **maxterm** is a *sum* of literals, in which each input variable appears exactly once.
- A function with n variables has 2^n maxterms
- The maxterms for a three-variable function $f(x,y,z)$:

$$\begin{array}{cccc} x + y + z & x + y + z' & x + y' + z & x + y' + z' \\ x' + y + z & x' + y + z' & x' + y' + z & x' + y' + z' \end{array}$$

- Each maxterm is *false* for exactly one combination of inputs:

Maxterm	Is <i>false</i> when...	Shorthand
$x + y + z$	$x=0, y=0, z=0$	M_0
$x + y + z'$	$x=0, y=0, z=1$	M_1
$x + y' + z$	$x=0, y=1, z=0$	M_2
$x + y' + z'$	$x=0, y=1, z=1$	M_3
$x' + y + z$	$x=1, y=0, z=0$	M_4
$x' + y + z'$	$x=1, y=0, z=1$	M_5
$x' + y' + z$	$x=1, y=1, z=0$	M_6
$x' + y' + z'$	$x=1, y=1, z=1$	M_7

“1” corresponds to “complement”
“0” corresponds to “original”

Product of Maxterms form

- Every function can also be written as a *unique product of maxterms*
- If you have a truth table for a function, you can write a product of **maxterms** expression by picking out the rows of the table where the function **output is 0**.

x	y	z	f(x,y,z)	f'(x,y,z)
0	0	0	1	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	0	1

$$\begin{aligned}
 f &= (x' + y + z)(x' + y + z')(x' + y' + z') \\
 &= M_4 M_5 M_7 \\
 &= \prod M(4,5,7)
 \end{aligned}$$

$$\begin{aligned}
 f' &= (x + y + z)(x + y + z')(x + y' + z) \\
 &\quad (x + y' + z')(x' + y' + z) \\
 &= M_0 M_1 M_2 M_3 M_6 \\
 &= \prod M(0,1,2,3,6)
 \end{aligned}$$

f' contains all the maxterms not in f

Minterms and Maxterms are related

- Any minterm m_i is the *complement* of the corresponding maxterm M_i

Minterm	Shorthand	Maxterm	Shorthand
$x'y'z'$	m_0	$x + y + z$	M_0
$x'y'z$	m_1	$x + y + z'$	M_1
$x'yz'$	m_2	$x + y' + z$	M_2
$x'yz$	m_3	$x + y' + z'$	M_3
$xy'z'$	m_4	$x' + y + z$	M_4
$xy'z$	m_5	$x' + y + z'$	M_5
xyz'	m_6	$x' + y' + z$	M_6
xyz	m_7	$x' + y' + z'$	M_7

- For example, $m_4' = M_4$ because $(xy'z')' = x' + y + z$

Converting between two forms

- We can convert a sum of minterms to a product of maxterms

From $f = \Sigma m(0,1,2,3,6)$

to complement: $f' = \Sigma m(4,5,7)$

$$= m_4 + m_5 + m_7$$

Complement again: $(f')' = (m_4 + m_5 + m_7)'$

So final: $f = m_4' m_5' m_7'$ [DeMorgan's law]

$$= M_4 M_5 M_7$$
 [By the previous page]

$$= \Pi M(4,5,7)$$

- In general, just replace the minterms with maxterms using maxterm numbers that don't appear in the sum of minterms:

$$f = \Sigma m(0,1,2,3,6)$$

$$= \Pi M(4,5,7)$$

- The same thing works for converting from a product of maxterms to a sum of minterms.

Gate Level Minimization

- Circuits of AND and OR: the cost is related to the number of gates and the number of gate inputs
- Looking for a minimum cost implementation of 2-level circuits of AND and OR gates
 - Minimum number of terms
 - Of those implementations with the same minimum number of terms, choose the one with the minimum number of literals
 - Not necessarily a unique solution
- **Sum of Products**: group of AND gates, one OR gate
- **Product of Sums**: group of OR gates, one AND gate

Gate Level Minimization

In the earlier lecture we had an example of simplification using Boolean Algebra. Now we use Boolean Algebra again to simplify the sum of products expression:

$$\begin{aligned}
 F &= A'BC + AB'C' + AB'C + ABC' + ABC \\
 &= A'BC + AB'C' + AB'C + ABC' + ABC + ABC \\
 &= BC(A' + A) + AC'(B' + B) + AC(B' + B) \\
 &= BC + AC' + AC \\
 &= BC + A
 \end{aligned}$$

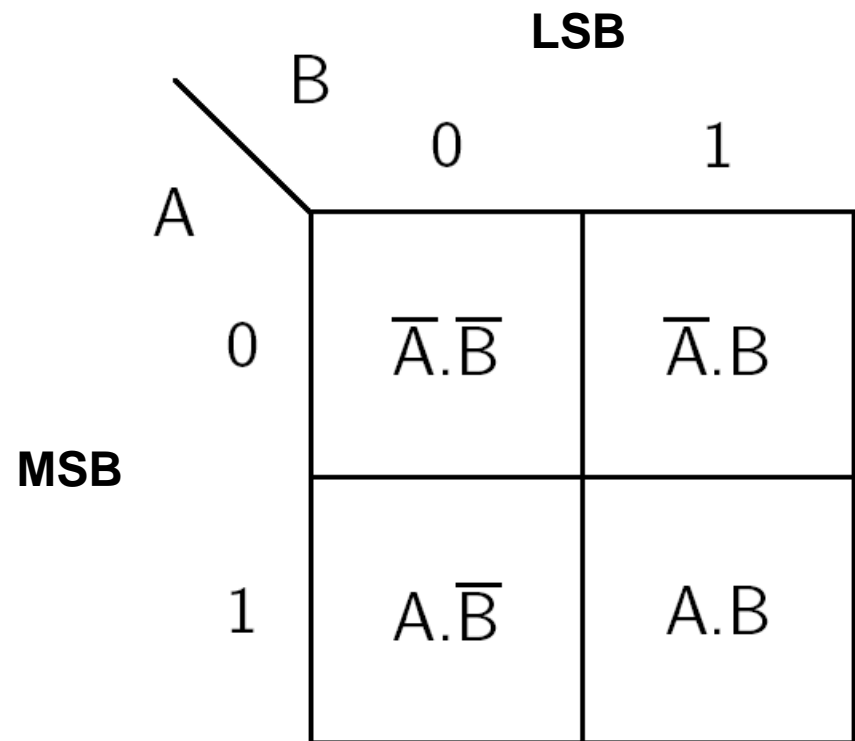
A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Gate Level Minimization

- Simplifying expressions using Boolean Algebra requires intuition and experience:
 - Difficult to apply systematically
 - Even when those suggested rules are followed will not necessarily find minimum solution
 - Difficult to recognise minimum solutions
- For complicated circuits this method becomes increasingly difficult.
- In this lecture we shall approach the problem of simplifying logical expressions in a systematic way using the **Karnaugh map** method.

Two Variable Karnaugh Map

- Each possible minterm of a sum of products expression is represented in a two-dimensional table or 'map'.



Two Variable Karnaugh Map

- For the expression

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

the Karnaugh map is given on the right.
For each minterm present in the sum of products the corresponding square contains a '1'.

- By circling the '1' squares, an equivalent expression results

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

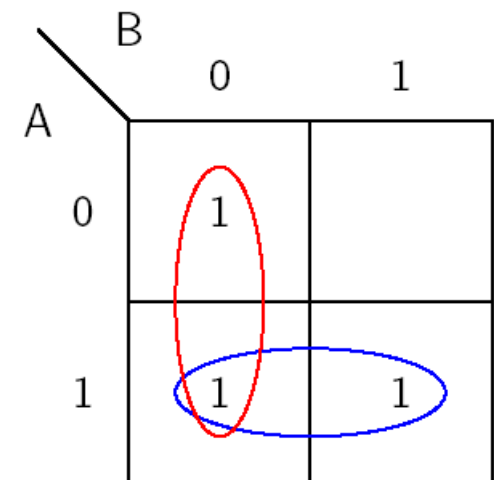
$$\Rightarrow C = \overline{B} + A$$

		B	
		0	1
A	0	1	
	1	1	1

		B	
		0	1
A	0	1	
	1	1	1

Two Variable Karnaugh Map

- By entering the minterms of the logical expression into the Karnaugh map a simplified but equivalent expression is obtained by summing the variables corresponding to the circled squares.
- The variables corresponding to circled squares are those which remain constant, i.e.



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

Three Variable Karnaugh Map

- The idea can be extending to logical expressions of three variables.
- The Karnaugh map is constructed such that adjacent squares differ only by one variable. This variable is complemented in one square and un-complemented in the other square.
- This arrangement is known as Gray coding.
- For the expression

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

the Karnaugh map is on the right

		C	
		0	1
AB	00	1	
	01	1	1
	11		1
	10	1	

Three Variable Karnaugh Map

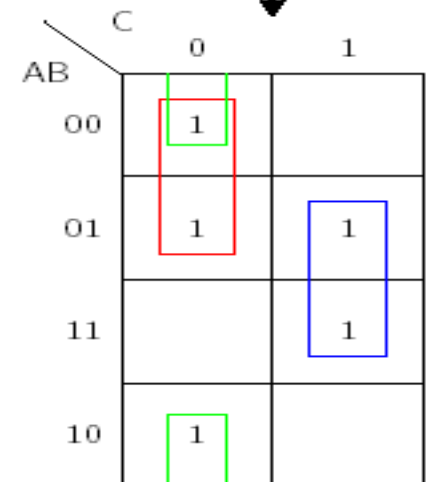
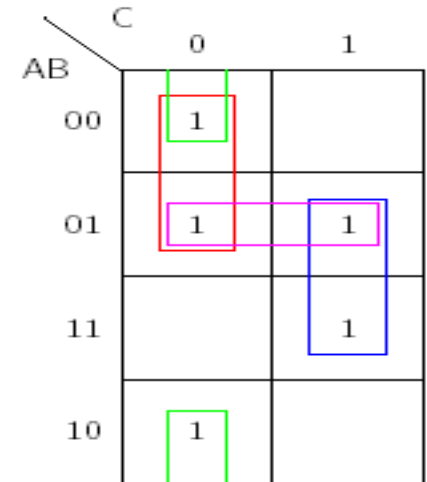
- Grouping adjacent '1' squares together into four groups allows the expression to be simplified.

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

- It is not necessary to use every possible grouping of '1' squares - it is sufficient that each '1' be a member of one of the groupings.
- This expression can be further simplified by discarding the first term or the last term – these would be two different applications of the **consensus theorem**.

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

- It should also be noted that groups can wrap around the edge of the map, e.g. the group corresponding to the term $\overline{B}.\overline{C}$.



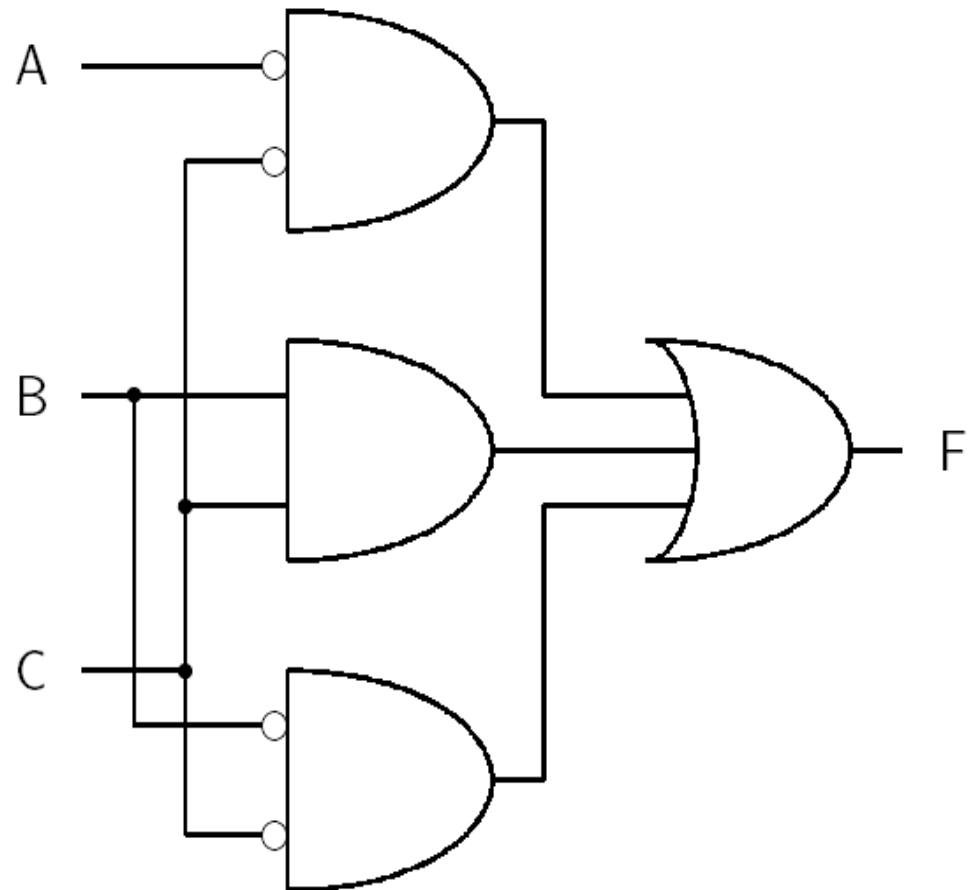
Logic Gate Implementation

- The simplified logical expression

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

has an equivalent logic gate implementation:

- This implementation involves one OR gate and three AND gates (two with inverting inputs); 9 gate inputs. The original expression involved one OR gate and five AND gates; 20 gate inputs.



Three Variable Karnaugh Map

- Another way of using the Karnaugh map is to seek the **complementary** expression, i.e. construct a map with '0' squares corresponding to min-terms not present in the expression
- The complement in this case is:

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

- Taking the complement of both sides and using DeMorgan's Law provides the minimum product of sums expression

$$\begin{aligned} F &= \overline{A B \bar{C} + \bar{B} C} \\ &= \overline{A B \bar{C}} . \overline{\bar{B} C} \\ &= (\bar{A} + \bar{B} + C). (B + \bar{C}) \end{aligned}$$

		C	
		0	1
AB	00		0
	01		
	11	0	
	10		0

		C	
		0	1
AB	00		0
	01		
	11	0	
	10		0

Three Variable Karnaugh Map

- It is also possible to **directly** find the minimum **Product of Sums** expression without using DeMorgan's Law:
 - Group adjacent "0"s
 - Construct **Sums** term: "0" corresponds to original; "1" corresponds to complement. (the same as in the form of Maxterms).
 - Then form **Products**.

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

		C	
		0	1
AB	00		0
	01		
	11	0	
	10		0

		C	
		0	1
AB	00		0
	01		
	11	0	
	10		0

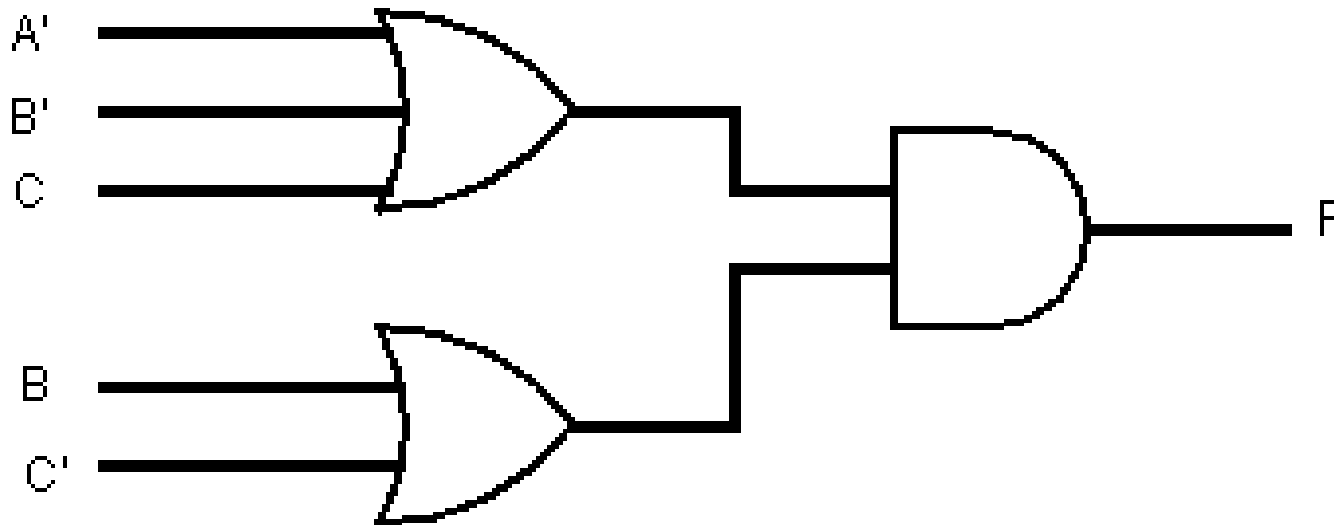
Logic Gate Implementation

- The simplified logical expression

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

has the logic gate implementation:

- The implementation involves one AND gate and two OR gates; 7 gate inputs.



Four Variable Karnaugh Map

- The Karnaugh map becomes useful when the numbers of inputs increase.
- For the four variable logical expression

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

the map is given on the right.

- Larger groups of four and eight minterms now exist.
- Summing the variables which remain constant within the groups gives the minimum sum of products:

$$F = \overline{C} + \overline{A}.D$$

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

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

Four Variable Karnaugh Map

- It is also possible to group the zero terms in the Karnaugh map.
- The minimum Product of Sums expression:

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

AB \ CD				
	00	01	11	10
00			0	
01			0	
11			0	0
10			0	0

AB \ CD				
	00	01	11	10
00			0	
01			0	
11			0	0
10			0	0

Karnaugh Map Procedure

1. The logical function is expressed as a sum of products.
2. The Karnaugh Map is constructed with a '1' in each square corresponding to a **minterm** present in the expression.
3. Squares containing '1's are grouped together, such that each square is contained within at least one group. In the case of the four variable map, groups may contain **one, two, four, eight or sixteen** members.
4. The largest of these groupings are sought while groupings which contain members which occur in other groups are discarded since they are redundant.
5. The variables which remain constant within each group represent the new min-terms of a simplified **sum of products** expression.
6. The same procedures for constructing simplified **product of sums**, except grouping "0" instead of "1".
7. Minimum **sum of products** and **product of sums** implementations should be checked for minimized 2-level gate circuits.

Don't Care Terms

- In a logic design it may occur that it does not matter whether or not a particular minterm is '1' or '0'.
- A so called *don't care* term is used instead and is usually represented by an 'x'.
- These don't care terms act as wild cards in the Karnaugh map. By pretending a don't care term is a '1' larger groupings result.

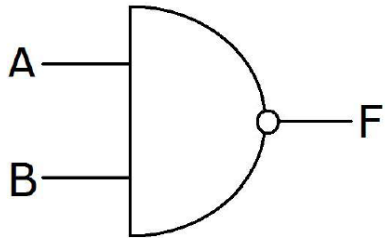
AB \ C	C	
	0	1
00	1	0
01	1	1
11	x	1
10	1	0

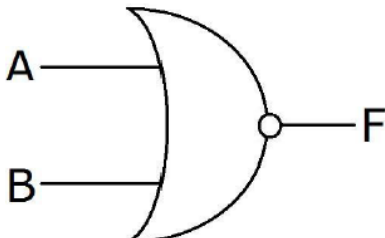
AB \ C	C	
	0	1
00	1	0
01	1	1
11	x	1
10	1	0

Methods of Simplification Summary

- Algebraic Simplification
 - Good for simple expressions of a few variables
 - Cases of large numbers of variables but a small number of terms
- Karnaugh Maps
 - Best method for expressions of 3-4 variables
- Quine-McCuskey Method
 - Underneath the method is identical to the Karnaugh Map, but it is termed in a tabular way making it more efficient for use in computer algorithms. Can be extended to many variables.
- Other methods
 - mostly based on Karnaugh map or Quine-McCluskey
 - Heuristic Procedures if the absolute minimum solution is not required, e.g. Espresso which gives near-min solutions

NAND and NOR Gates

Logic Gate	Graphical Symbol	Boolean Function	Truth Table															
NAND		$F = \overline{A \cdot B}$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	F	0	0	1	0	1	1	1	0	1	1	1	0
A	B	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																

Logic Gate	Graphical Symbol	Boolean Function	Truth Table															
NOR		$F = \overline{A + B}$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	F	0	0	1	0	1	0	1	0	0	1	1	0
A	B	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																

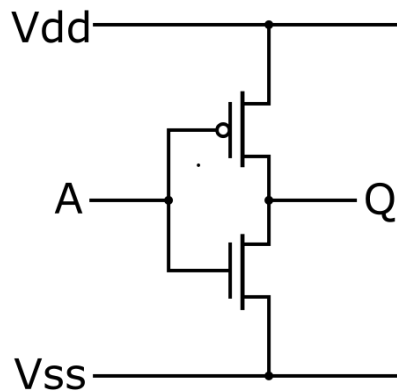
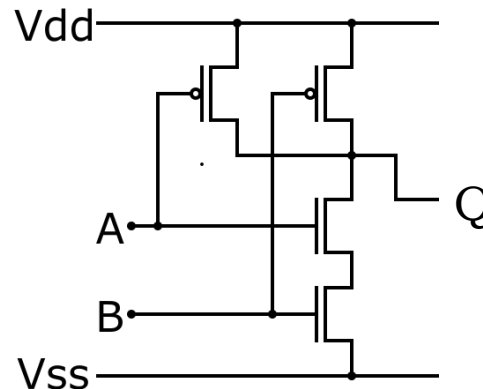
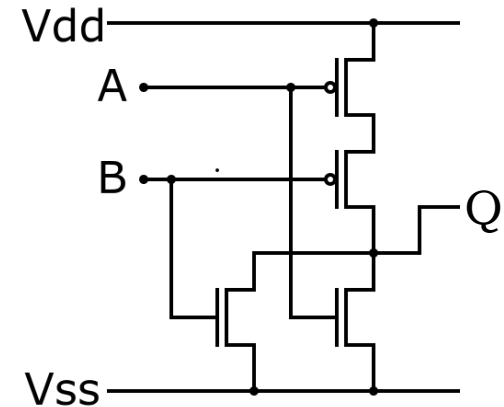
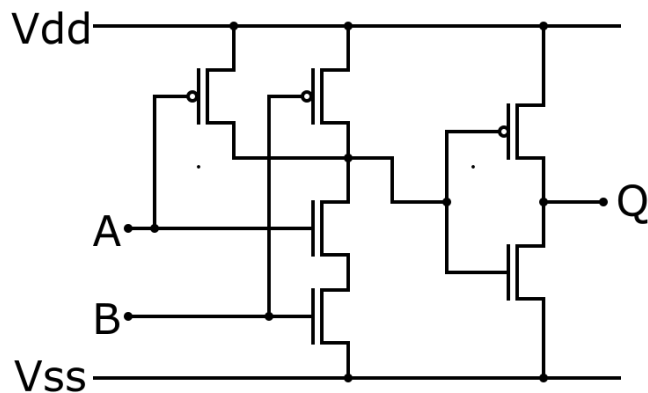
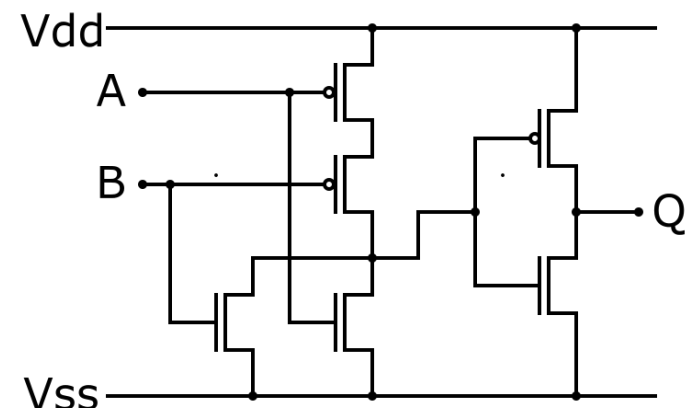
NAND and NOR Gates

Frequently used because they are faster and use fewer components

e.g. The number of transistors used in CMOS gates:

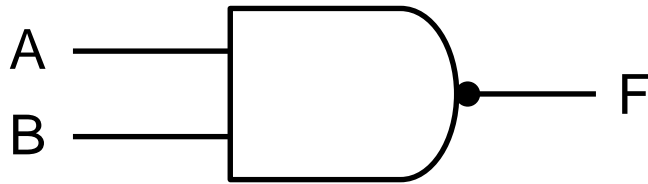
- n -input CMOS OR gate $2n+2$
- n -input CMOS AND gate $2n+2$
- n -input CMOS NOR gate $2n$
- n -input CMOS NAND gate $2n$
- CMOS inverter 2

CMOS Gate Implementation

**NOT****NAND****NOR****AND****OR**

NAND-NAND Logic

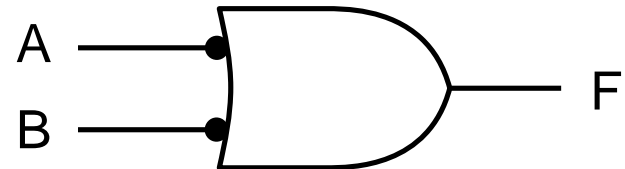
- We can perform all logic operations using just combinations of NAND gates. Consider the following two-input NAND gate and its corresponding Boolean algebra description



$$F(A, B) = \overline{A \bullet B}$$

- Use De Morgan's Law, we can re-write $F(A, B)$ as

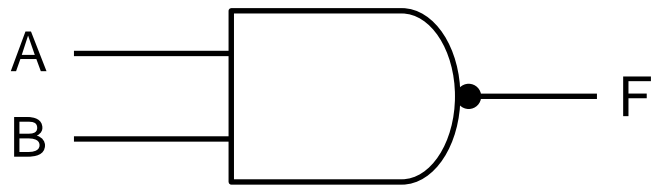
$$F(A, B) = \overline{A} + \overline{B}$$



- This is just an OR gate fed with the original inputs (A and B) inverted

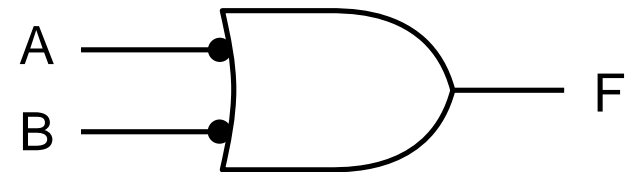
NAND-NAND Logic

- Hence the two circuits are equivalent



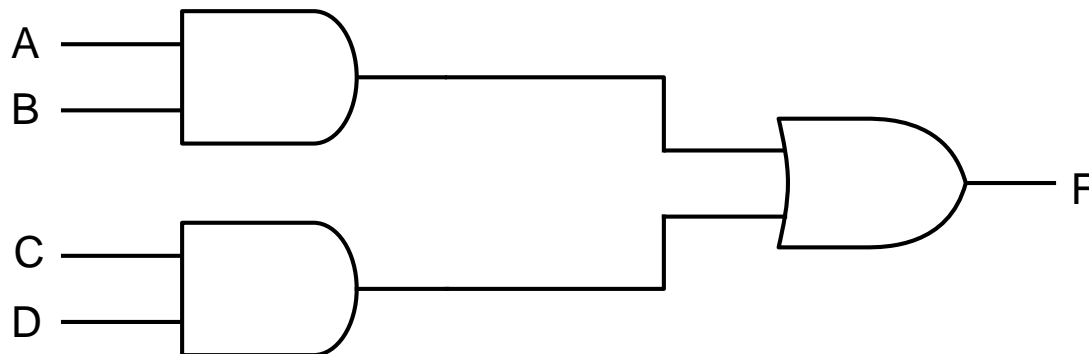
$$F(A, B) = \overline{A \bullet B}$$

≡



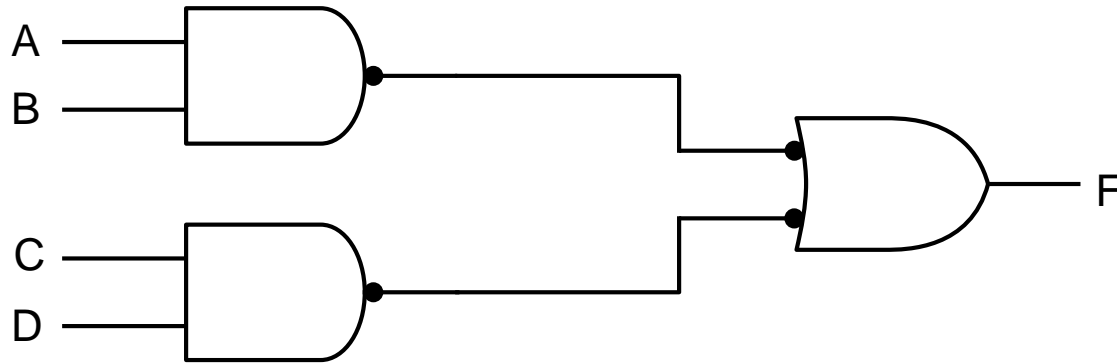
$$F(A, B) = \overline{A} + \overline{B}$$

- This allows us to start with an AND-OR logic description

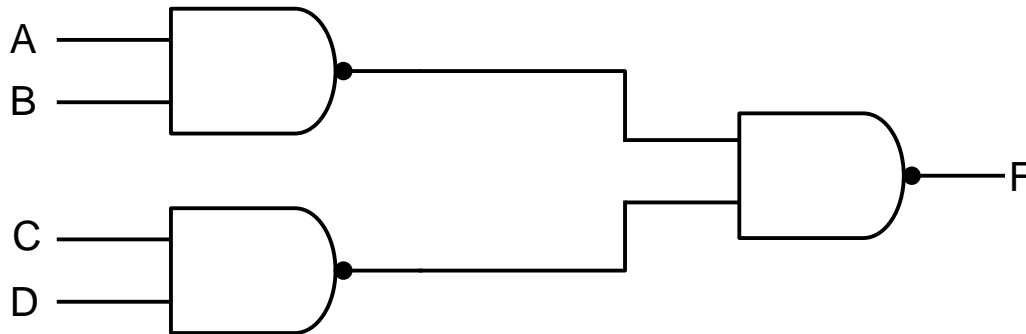


NAND-NAND Logic

- Adding two inverters in series does not change the output, F:



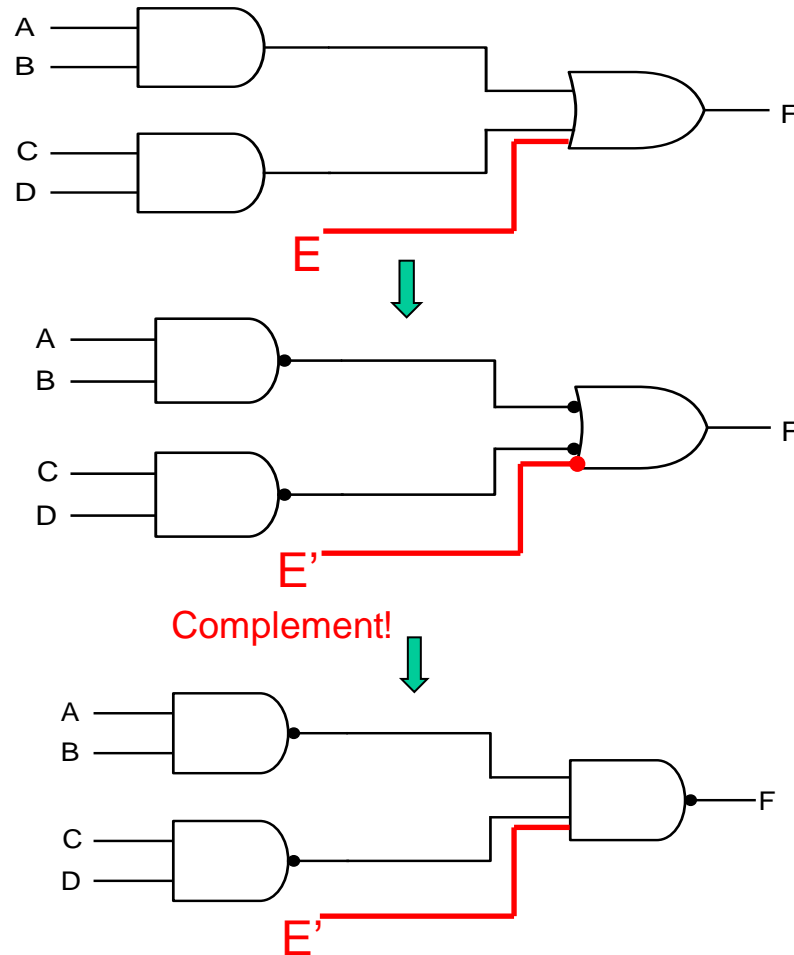
- As we have just seen, the last gate above is equivalent to a NAND gate



- Only NAND gates are used – this is known as *NAND-NAND logic*.

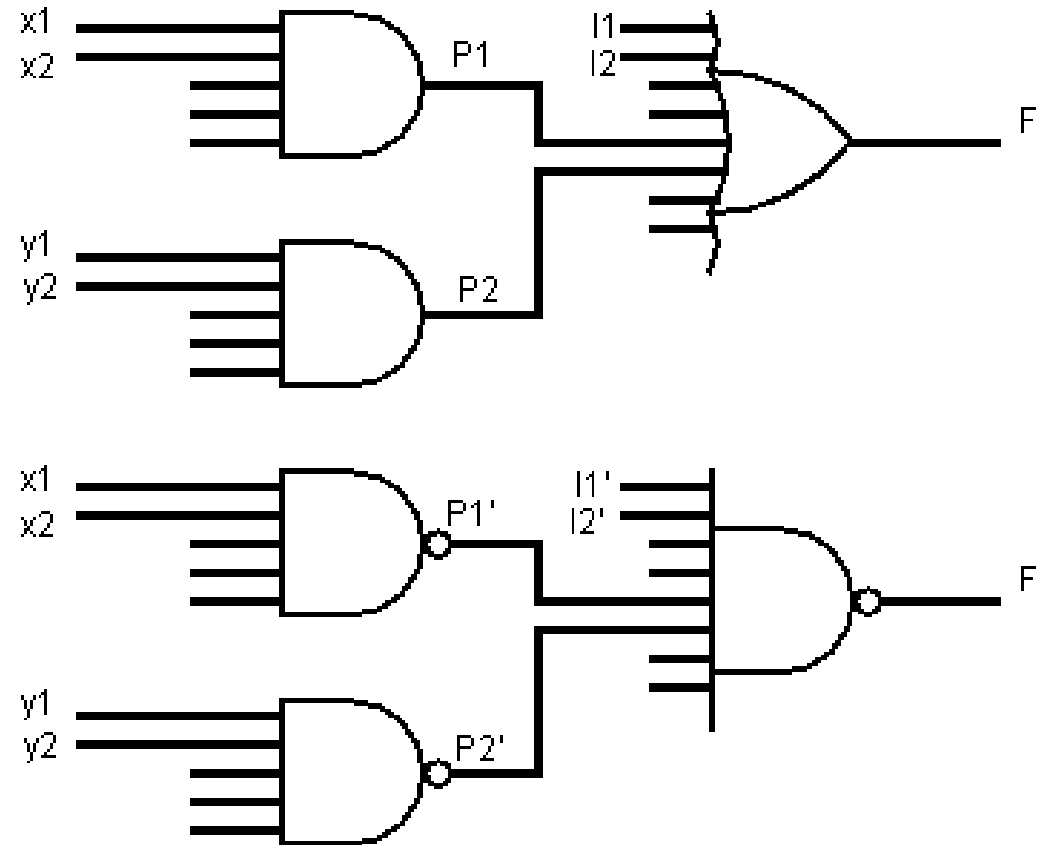
NAND-NAND Logic

- What about a single literal appearing at the 2nd stage (OR gate input)?



NAND-NAND

1. Find minimum **sum of products**
2. Draw corresponding AND-OR circuit
3. Replace all gates with NAND gates leaving interconnections as they were
4. If the output gate has any single literal inputs, complement these



$$F = l_1 + l_2 + \dots + P_1 + P_2 + \dots$$

$$F = \overline{\overline{l_1} \cdot \overline{l_2} \cdot \dots \cdot \overline{P_1} \cdot \overline{P_2} \cdot \dots}$$

NOR-NOR

1. Find minimum **product of sums**
2. Draw corresponding OR-AND circuit
3. Replace all gates with NOR gates leaving interconnections as they were
4. If the output gate has any single literal inputs, complement these

$$F = l_1 \cdot l_2 \cdots S_1 \cdot S_2 \cdots$$

$$F = \overline{\overline{l_1} + \overline{l_2} + \cdots + \overline{S_1} + \overline{S_2} + \cdots}$$

