Heaven's Light is Our Guide

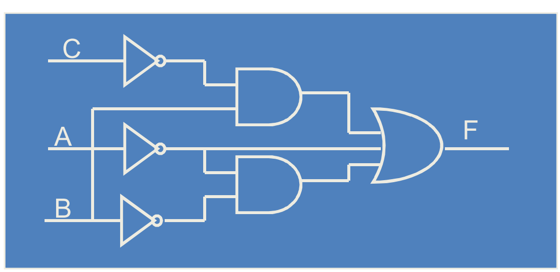
Department of Computer Science & Engineering
Rajshahi University of Engineering & Technology, Bangladesh

Course Code: CSE 2203
Course Title: Digital Techniques

Date : 20.09.2017
Session : V
Topic : Gate Level Minimization
Faculty : Dr. Boshir Ahmed
Professor
Department of CSE, RUET
E mail: boshir78@gmail.com

Combinational Logic Circuit from Logic Function

- Consider function F = A' + B•C' + A'•B'
- A combinational logic circuit can be constructed to implement F, by appropriately connecting input signals and logic gates:
  - Circuit input signals → from function variables (A, B, C)
  - Circuit output signal → function output (F)
  - Logic gates → from logic operations

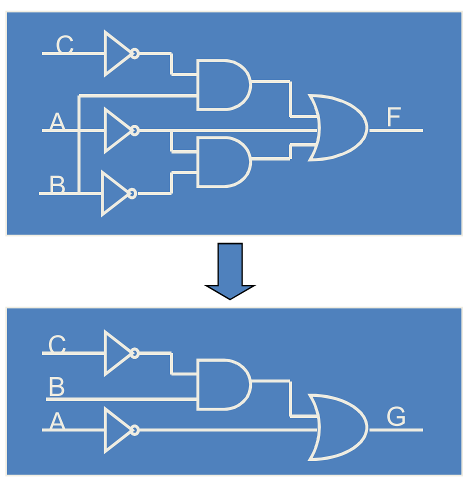


Combinational Logic Circuit from Logic Function (cont.)

- In order to design a cost-effective and efficient circuit, we must minimize the circuit's size (area) and propagation delay (time required for an input signal change to be observed at the output line)
- Observe the truth table of F=A' + B•C' + A'•B' and G=A' + B•C'
- Truth tables for F and G are identical → same function
- Use G to implement the logic circuit (less components)

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

Combinational Logic Circuit from Logic Function (cont.)



Boolean expressions-NOT unique

- Unlike truth tables, expressions representing a Boolean function are NOT unique.
- Example:
  - F(x,y,z) = x'•y'•z' + x'•y•z' + x•y•z'
  - G(x,y,z) = x'•y'•z' + y•z'
- The corresponding truth tables for F() and G() are to the right. They are identical.
- Thus, F() = G()

x	y	z		F	G
0	0	0		1	1
0	0	1		0	0
0	1	0		1	1
0	1	1		0	0
1	0	0		0	0
1	0	1		0	0
1	1	0		1	1
1	1	1		0	0

Algebraic Manipulation

- Boolean algebra is a useful tool for simplifying digital circuits.
- Why do it? Simpler can mean cheaper, smaller, faster.
- Example: Simplify F = x'yz + x'yz' + xz.

  F = x'yz + x'yz' + xz
  = x'y(z+z') + xz
  = x'y•1 + xz
  = x'y + xz

Algebraic Manipulation (cont.)

- Example: Prove
  $x'y'z' + x'yz' + xyz' = x'z' + yz'$
- **Proof:**
  $x'y'z' + x'yz' + xyz'$
  $= x'y'z' + x'yz' + x'yz' + xyz'$
  $= x'z'(y'+y) + yz'(x'+x)$
  $= x'z' \cdot 1 + yz' \cdot 1$
  $= x'z' + yz'$

Complement of a Function

- The complement of a function is derived by interchanging (• and +), and (1 and 0), and complementing each variable.
- Otherwise, interchange 1s to 0s in the truth table column showing F.
- The *complement* of a function IS NOT THE SAME as the *dual* of a function.

Complementation: Example

- Find G(x,y,z), the complement of F(x,y,z) = xy'z' + x'yz
- G = F' = (xy'z' + x'yz)'
  = (xy'z')' • (x'yz)' *DeMorgan*
  = (x'+y+z) • (x+y'+z') *DeMorgan* again
- Note: The complement of a function can also be derived by finding the function's *dual,* and then complementing all of the literals

Canonical and Standard Forms

- We need to consider formal techniques for the simplification of Boolean functions.
  - Identical functions will have exactly the same canonical form.
  - Minterms and Maxterms
  - Sum-of-Minterms and Product-of- Maxterms
  - Product and Sum terms
  - Sum-of-Products (SOP) and Product-of-Sums (POS)

Definitions

- *Literal:* A variable or its complement
- *Product term:* literals connected by •
- *Sum term:* literals connected by +
- *Minterm:* a product term in which all the variables appear exactly once, either complemented or uncomplemented
- *Maxterm:* a sum term in which all the variables appear exactly once, either complemented or uncomplemented

Minterm

- Represents exactly one combination in the truth table.
- Denoted by m_j, where j is the decimal equivalent of the minterm's corresponding binary combination (b_j).
- A variable in m_j is complemented if its value in b_j is 0, otherwise is uncomplemented.
- Example: Assume 3 variables (A,B,C), and j=3. Then, b_j = 011 and its corresponding minterm is denoted by m_j = A'BC

Maxterm

- Represents exactly one combination in the truth table.
- Denoted by M_j, where j is the decimal equivalent of the maxterm's corresponding binary combination (b_j).
- A variable in M_j is complemented if its value in b_j is 1, otherwise is uncomplemented.
- Example: Assume 3 variables (A,B,C), and j=3. Then, b_j = 011 and its corresponding maxterm is denoted by M_j = A+B'+C'

Truth Table notation for Minterms and Maxterms

- Minterms and Maxterms are easy to denote using a truth table.
- Example: Assume 3 variables x,y,z (order is fixed)

x	y	z		Minterm	Maxterm
0	0	0		$x'y'z' = m_0$	$x+y+z = M_0$
0	0	1		$x'y'z = m_1$	$x+y+z' = M_1$
0	1	0		$x'yz' = m_2$	$x+y'+z = M_2$
0	1	1		$x'yz = m_3$	$x+y'+z'= M_3$
1	0	0		$xy'z' = m_4$	$x'+y+z = M_4$
1	0	1		$xy'z = m_5$	$x'+y+z' = M_5$
1	1	0		$xyz' = m_6$	$x'+y'+z = M_6$
1	1	1		$xyz = m_7$	$x'+y'+z' = M_7$

Canonical Forms (Unique)

- Any Boolean function F( ) can be expressed as a *unique* **sum** of **min**terms and a unique **product** of **max**terms (under a fixed variable ordering).
- In other words, every function F() has two canonical forms:
  - Canonical Sum-Of-Products (sum of minterms)
  - Canonical Product-Of-Sums (product of maxterms)

Canonical Forms (cont.)

- Canonical Sum-Of-Products:
  The minterms included are those m_j such that F( ) = 1 in row *j* of the truth table for F( ).
- Canonical Product-Of-Sums:
  The maxterms included are those M_j such that F( ) = 0 in row *j* of the truth table for F( ).

Example

- Truth table for $f_1(a,b,c)$ at right
- The canonical sum-of-products form for f_1 is
  $f_1(a,b,c) = m_1 + m_2 + m_4 + m_6$
  $= a'b'c + a'bc' + ab'c' + abc'$
- The canonical product-of-sums form for f_1 is
  $f_1(a,b,c) = M_0 \bullet M_3 \bullet M_5 \bullet M_7$
  $= (a+b+c)\bullet(a+b'+c')\bullet(a'+b+c')\bullet(a'+b'+c')$.
- Observe that: $m_j = M_j'$

a	b	c		f_1
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		1
1	1	1		0

Shorthand: ∑ and ∏

- $f_1(a,b,c) = \sum m(1,2,4,6)$, where $\sum$ indicates that this is a sum-of-products form, and $m(1,2,4,6)$ indicates that the minterms to be included are m_1, m_2, m_4, and m_6.
- $f_1(a,b,c) = \prod M(0,3,5,7)$, where $\prod$ indicates that this is a product-of-sums form, and $M(0,3,5,7)$ indicates that the maxterms to be included are M_0, M_3, M_5, and M_7.
- Since $m_j = M_j'$ for any *j*,
  $\sum m(1,2,4,6) = \prod M(0,3,5,7) = f_1(a,b,c)$

Conversion Between Canonical Forms

- Replace $\sum$ with $\prod$ (or *vice versa*) and replace those *j*'s that appeared in the original form with those that do not.
- Example:
  $f_1(a,b,c) = a'b'c + a'bc' + ab'c' + abc'$
  $= m_1 + m_2 + m_4 + m_6$
  $= \sum(1,2,4,6)$
  $= \prod(0,3,5,7)$
  $= (a+b+c) \bullet (a+b'+c') \bullet (a'+b+c') \bullet (a'+b'+c')$

Standard Forms (NOT Unique)

- Standard forms are "*like*" canonical forms, except that not all variables need appear in the individual product (SOP) or sum (POS) terms.
- Example:
  $f_1(a,b,c) = a'b'c + bc' + ac'$
  is a *standard* sum-of-products form
- $f_1(a,b,c) = (a+b+c) \bullet (b'+c') \bullet (a'+c')$
  is a *standard* product-of-sums form.

Conversion of SOP from standard to canonical form

- Expand *non-canonical* terms by inserting equivalent of 1 in each missing variable x: (x + x') = 1
- Remove duplicate minterms
- $f_1(a,b,c) = a'b'c + bc' + ac'$
  $= a'b'c + (a+a')bc' + a(b+b')c'$
  $= a'b'c + abc' + a'bc' + abc' + ab'c'$
  $= a'b'c + abc' + a'bc + ab'c'$

Conversion of POS from standard to canonical form

- Expand noncanonical terms by adding 0 in terms of missing variables (*e.g.*, xx' = 0) and using the distributive law
- Remove duplicate maxterms
- $f_1(a,b,c) = (a+b+c)\bullet(b'+c')\bullet(a'+c')$
  $= (a+b+c)\bullet(aa'+b'+c')\bullet(a'+bb'+c')$
  $= (a+b+c)\bullet(a+b'+c')\bullet(a'+b'+c')\bullet$
  $(a'+b+c')\bullet(a'+b'+c')$
  $= (a+b+c)\bullet(a+b'+c')\bullet(a'+b'+c')\bullet(a'+b+c')$

Karnaugh Maps

- Karnaugh maps (K-maps) are *graphical* representations of boolean functions.
- One ***map cell*** corresponds to a row in the truth table.
- Also, one map cell corresponds to a minterm or a maxterm in the boolean expression
- Multiple-cell areas of the map correspond to standard terms.

Two-Variable Map

$x_1 \backslash x_2$	0	1
0	0 m_0	1 m_1
1	2 m_2	3 m_3

OR

$x_2 \backslash x_1$	0	1
0	0 m_0	2 m_2
1	1 m_1	3 m_3

NOTE: ordering of variables is IMPORTANT for $f(x_1,x_2)$, x_1 is the row, x_2 is the column.

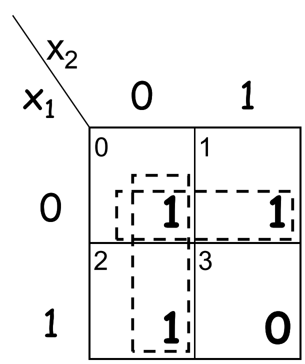
Cell 0 represents $x_1'x_2'$; Cell 1 represents $x_1'x_2$; etc. If a minterm is present in the function, then a 1 is placed in the corresponding cell.

Two-Variable Map (cont.)

- Any two adjacent cells in the map differ by ONLY one variable, which appears complemented in one cell and uncomplemented in the other.
- Example:
  m_0 ($=x_1'x_2'$) is adjacent to m_1 ($=x_1'x_2$) and m_2 ($=x_1x_2'$) but NOT m_3 ($=x_1x_2$)

2-Variable Map -- Example

- $f(x_1,x_2) = x_1'x_2' + x_1'x_2 + x_1x_2'$
  $= m_0 + m_1 + m_2$
  $= x_1' + x_2'$
- 1s placed in K-map for specified minterms m_0, m_1, m_2
- Grouping (ORing) of 1s allows simplification
- What (simpler) function is represented by each dashed rectangle?
  - $x_1' = m_0 + m_1$
  - $x_2' = m_0 + m_2$
- Note m_0 covered twice

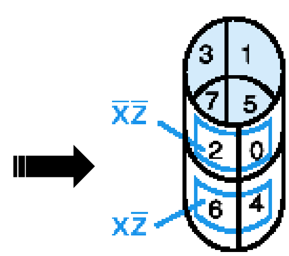


Minimization as SOP using K-map

- Enter 1s in the K-map for each product term in the function
- Group *adjacent* K-map cells containing 1s to obtain a product with fewer variables. Group size must be in power of 2 (2, 4, 8, …)
- Handle “boundary wrap” for K-maps of 3 or more variables.
- Realize that answer may not be unique

Three-Variable Map

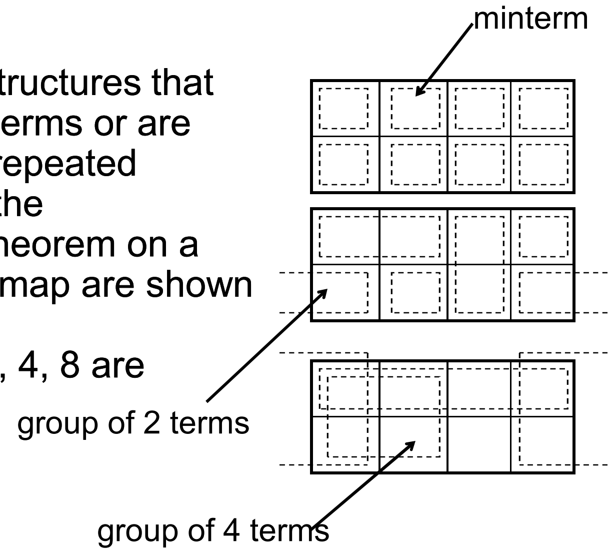
x \ yz	00	01	11	10
0	0 m_0	1 m_1	3 m_3	2 m_2
1	4 m_4	5 m_5	7 m_7	6 m_6



-Note: variable ordering is (x,y,z); yz specifies column, x specifies row.
-Each cell is adjacent to ***three*** other cells (left or right or top or bottom or edge wrap)

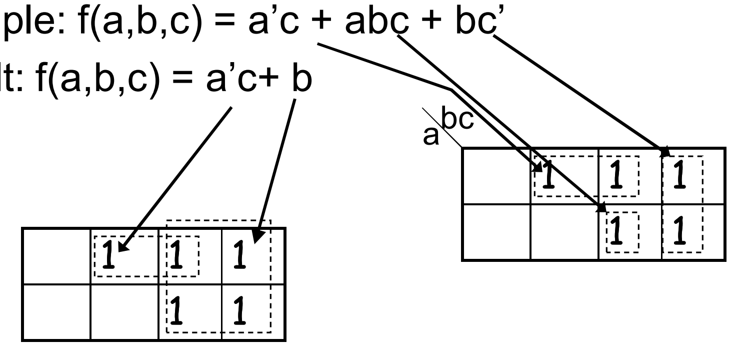
Three-Variable Map (cont.)

The types of structures that are either minterms or are generated by repeated application of the minimization theorem on a three variable map are shown at right.
Groups of 1, 2, 4, 8 are possible.



Simplification

- Enter minterms of the Boolean function into the map, then group terms
- Example: f(a,b,c) = a'c + abc + bc'
- Result: f(a,b,c) = a'c+ b



More Examples

- $f_1(x, y, z) = \sum m(2,3,5,7)$
  - $f_1(x, y, z) = x'y + xz$

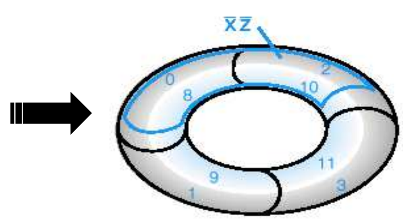
x \ yz	00	01	11	10
0			1	1
1		1	1	

- $f_2(x, y, z) = \sum m(0,1,2,3,6)$
  - $f_2(x, y, z) = x'+yz'$

1	1	1	1
			1

Four-Variable Maps

WX \ YZ	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}



- Top cells are adjacent to bottom cells. Left-edge cells are adjacent to right-edge cells.
- Note variable ordering (WXYZ).

Four-variable Map Simplification

- One square represents a minterm of 4 literals.
- A rectangle of 2 adjacent squares represents a product term of 3 literals.
- A rectangle of 4 squares represents a product term of 2 literals.
- A rectangle of 8 squares represents a product term of 1 literal.
- A rectangle of 16 squares produces a function that is equal to logic 1.

Example

- Simplify the following Boolean function (A,B,C,D) = ∑m(0,1,2,4,5,7,8,9,10,12,13).
- First put the function g( ) into the map, and then group as many 1s as possible.

ab \ cd			
1	1		1
1	1	1	
1	1		
1	1		1

1	1		1
1	1	1	
1	1		
1	1		1

g(A,B,C,D) = c'+b'd'+a'bd

Don't Care Conditions

- There may be a combination of input values which
  - will **never** occur
  - if they do occur, the output is of no concern.
- The function value for such combinations is called a *don't care*.
- They are denoted with **x** or **–**. Each **x** may be arbitrarily assigned the value 0 or 1 in an implementation.
- Don't cares can be used to ***further*** simplify a function

Example

- Simplify the function f(a,b,c,d) whose K-map is shown at the right.
- f = a'c'd+ab'+cd'+a'bc'

  or
- f = a'c'd+ab'+cd'+a'bd'

ab \ cd	00	01	11	10
00	0	1	0	1
01	1	1	0	1
11	0	0	x	x
10	1	1	x	x

0	1	0	1
1	1	0	1
0	0	x	x
1	1	x	x

0	1	0	1
1	1	0	1
0	0	x	x
1	1	x	x

Another Example

- Simplify the function g(a,b,c,d) whose K-map is shown at right.
- g = a’c’+ ab
  or
- g = a’c’+b’d

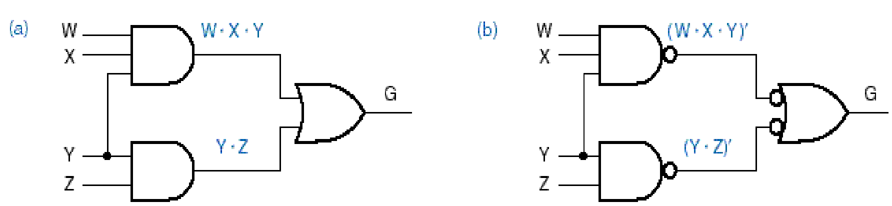
ab\cd

x	1	0	0
1	x	0	x
1	x	x	1
0	x	x	0

x	1	0	0
1	x	0	x
1	x	x	1
0	x	x	0

x	1	0	0
1	x	0	x
1	x	x	1
0	x	x	0

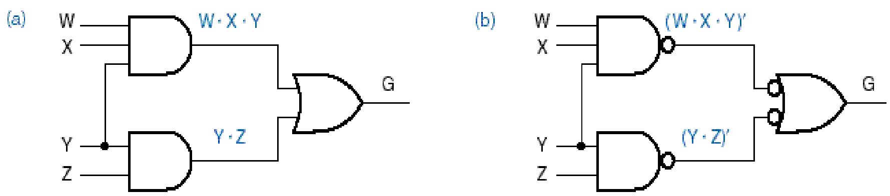
AND-OR (SOP) Emulation Using NANDs



Two-level implementations

a) Original SOP
b) Implementation with NANDs

AND-OR (SOP) Emulation Using NANDs (cont.)

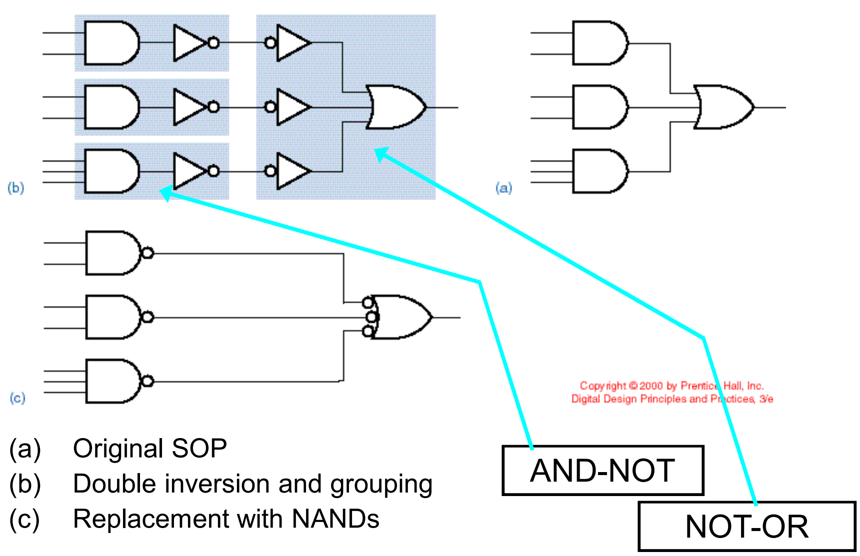


Verify:

(a) G = WXY + YZ

(b) G = ((WXY)' • (YZ)')'
= (WXY)'' + (YZ)'' = WXY + YZ

SOP with NAND



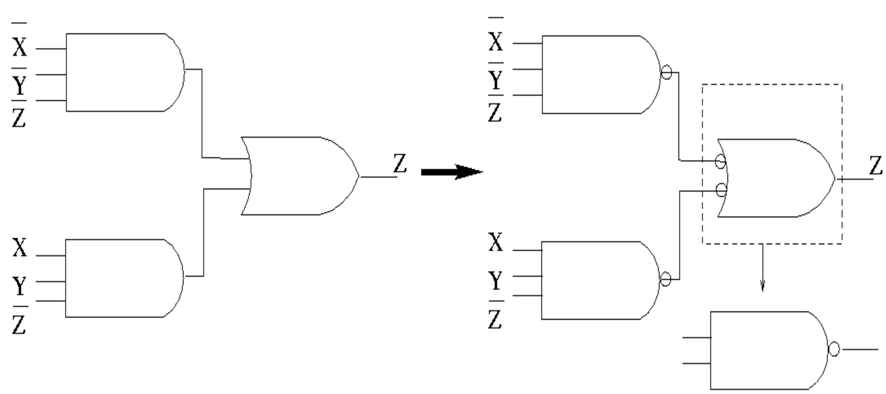
(a) Original SOP
(b) Double inversion and grouping
(c) Replacement with NANDs

Two-Level NAND Gate Implementation - Example

F (X,Y,Z) = Σm(0,6)

1. Express F in SOP form:
   F = X'Y'Z' + XYZ'
2. Obtain the AND-OR implementation for F.
3. Add bubbles and inverters to transform AND-OR to NAND-NAND gates.

Example (cont.)



Two-level implementation with NANDs
F = X'Y'Z' + XYZ'

Multilevel NAND Circuits

Starting from a multilevel circuit:

1. Convert all AND gates to NAND gates with AND-NOT graphic symbols.
2. Convert all OR gates to NAND gates with NOT-OR graphic symbols.
3. Check all the bubbles in the diagram. For every bubble that is not counteracted by another bubble along the same line, insert a NOT gate or complement the input literal from its original appearance.

Example

Use NAND gates and NOT gates to implement

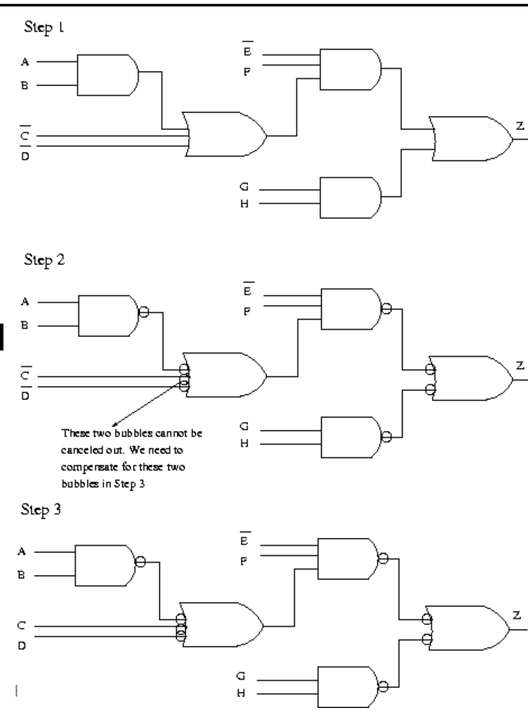
Z=E’F(AB+C’+D’)+GH

AB

AB+C’+D’

E’F(AB+C’+D’)

E’F(AB+C’+D’)+GH



Yet Another Example!

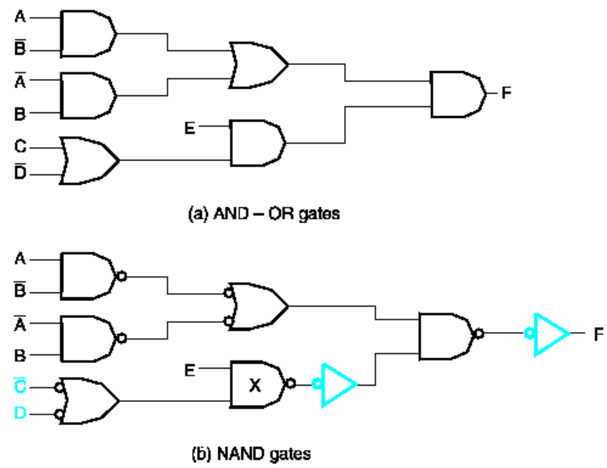


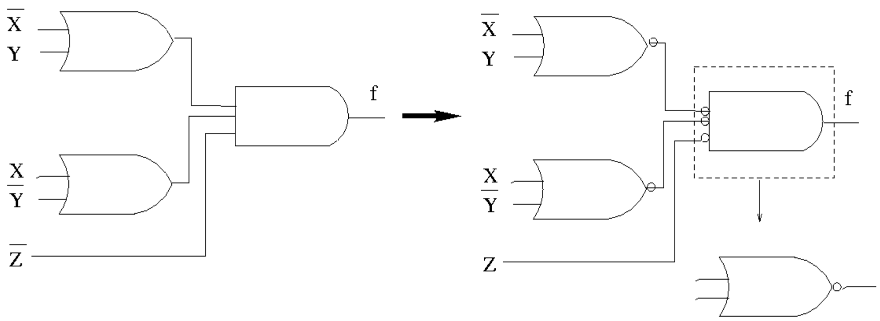
Fig. 2-32 Implementing $F = (A\overline{B} + \overline{A}B)E(C + \overline{D})$

Two-Level NOR Gate Implementation - Example

F(X,Y,Z) = Σm(0,6)

1. Express F’ in SOP form:
   1. F’ = Σm(1,2,3,4,5,7)
      = X’Y’Z + X’YZ’ + X’YZ + XY’Z’ + XY’Z + XYZ
   2. F’ = XY’ + X’Y + Z
2. Take the complement of F’ to get F in the POS form: F = (F’)' = (X'+Y)(X+Y')Z'
3. Obtain the OR-AND implementation for F.
4. Add bubbles and inverters to transform OR-AND implementation to NOR-NOR implementation.

Example (cont.)

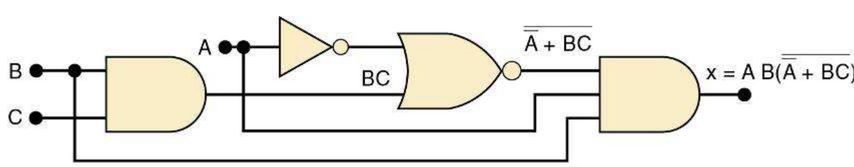


Two-level implementation with NORs

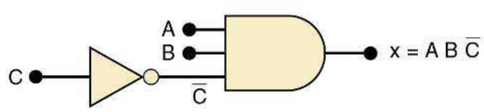
F = (F’)' = (X'+Y)(X+Y')Z'

Simplifying Logic Circuits (Practice)

- The circuits shown provide the same output
  - Circuit (b) is clearly less complex.



(a)

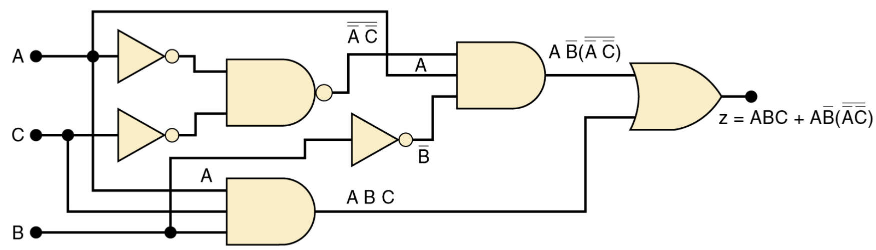


(b)

Logic circuits can be simplified using Boolean algebra and Karnaugh mapping.

Algebraic Simplification

Simplify the logic circuit shown.



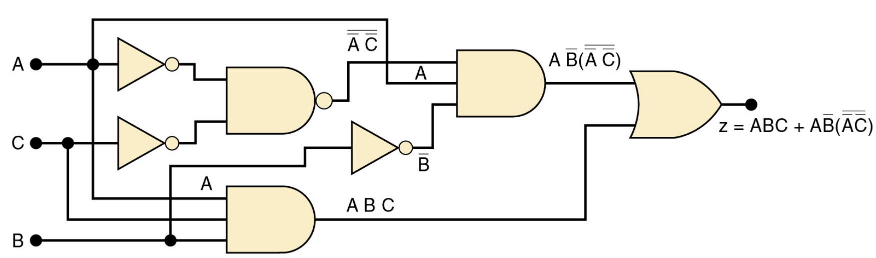
The first step is to determine the expression for the output: $z = ABC + A\overline{B} \cdot \overline{(\overline{A}\,\overline{C})}$

Once the expression is determined, break down large inverter signs by DeMorgan's theorems & multiply out all terms.

$$
\begin{aligned}
z &= ABC + A\overline{B}(\overline{\overline{A}} + \overline{\overline{C}}) && \text{[theorem (17)]}\\
&= ABC + A\overline{B}(A + C) && \text{[cancel double inversions]}\\
&= ABC + A\overline{B}A + A\overline{B}C && \text{[multiply out]}\\
&= ABC + A\overline{B} + A\overline{B}C && [A \cdot A = A]
\end{aligned}
$$

Algebraic Simplification

Simplify the logic circuit shown.



Factoring—the first & third terms above have ***AC*** in common, which can be factored out:

$$z = AC(B + \overline{B}) + A\overline{B}$$

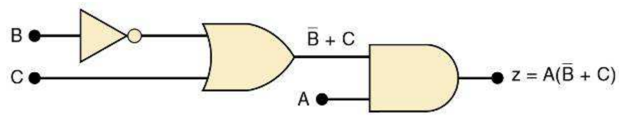
Since ***B*** + ***B*** = 1, then…

$$
\begin{aligned}
z &= AC(1) + A\overline{B}\\
&= AC + A\overline{B}
\end{aligned}
$$

Factor out ***A***, which results in…

Algebraic Simplification

Simplifed logic circuit.



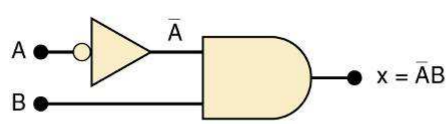
$$z = A(C + \overline{B})$$

Designing Combinational Logic Circuits

- To solve any logic design problem:
  - Interpret the problem and set up its truth table.
  - Write the **AND** (product) term for each case where output = 1.
  - Combine the terms in SOP form.
  - Simplify the output expression if possible.
  - Implement the circuit for the final, simplified expression.

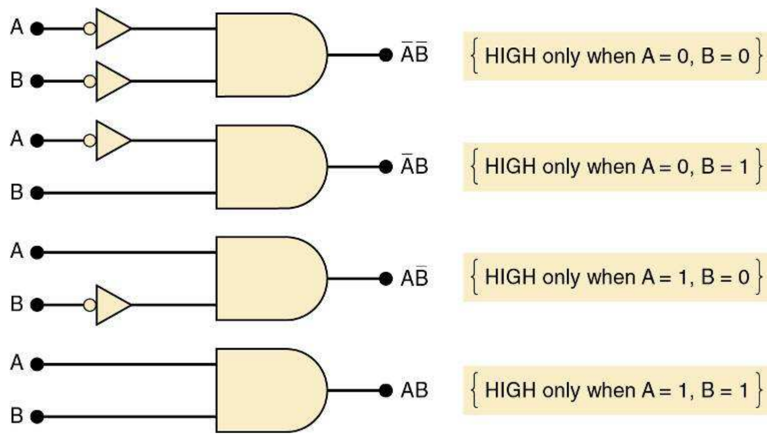
Circuit that produces a 1 output only for the A = 0, B = 1 condition.

A	B	x
0	0	0
0	1	1
1	0	0
1	1	0



Designing Combinational Logic Circuits

An **AND** gate with appropriate inputs can be used to produce a HIGH output for a specific set of input levels.

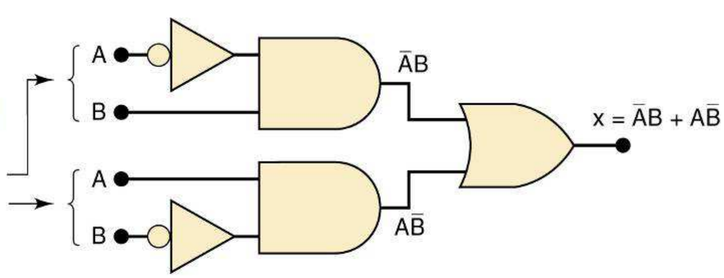


Designing Combinational Logic Circuits

Each set of input conditions that is to produce a 1 output is implemented by a separate **AND** gate.

The **AND** outputs are **OR**ed to produce the final output.

A	B	x
0	0	0
0	1	1
1	0	1
1	1	0



Designing Combinational Logic Circuits

Truth table for a 3-input circuit.

AND terms for each case where output is 1.

A	*B*	*C*	*x*	
0	0	0	0	
0	0	1	0	
0	1	0	1	$\rightarrow \overline{A}B\overline{C}$
0	1	1	1	$\rightarrow \overline{A}BC$
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	$\rightarrow ABC$

Designing Combinational Logic Circuits

Design a logic circuit with three inputs, A, B, and C. Output to be HIGH only when a majority inputs are HIGH.

Truth table.

AND terms for each case where output is 1.

A	*B*	*C*	*x*	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\rightarrow \overline{A}BC$
1	0	0	0	
1	0	1	1	$\rightarrow A\overline{B}C$
1	1	0	1	$\rightarrow AB\overline{C}$
1	1	1	1	$\rightarrow ABC$

SOP expression for the output:

$$x = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

Designing Combinational Logic Circuits

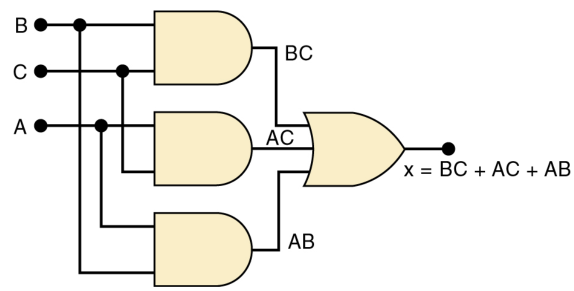
Design a logic circuit with three inputs, A, B, and C. Output to be HIGH only when a majority inputs are HIGH.

Simplified output expression:

$$x = ABC + ABC + ABC + ABC + ABC + ABC$$

Implementing the circuit after factoring:

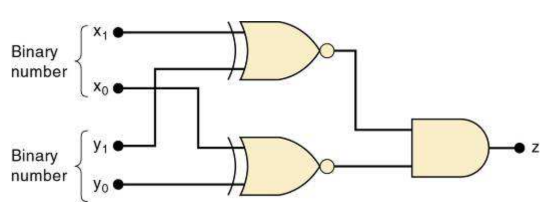
$$x = BC + AC + AB$$



Since the expression is in SOP form, the circuit is a group of **AND** gates, working into a single **OR** gate,

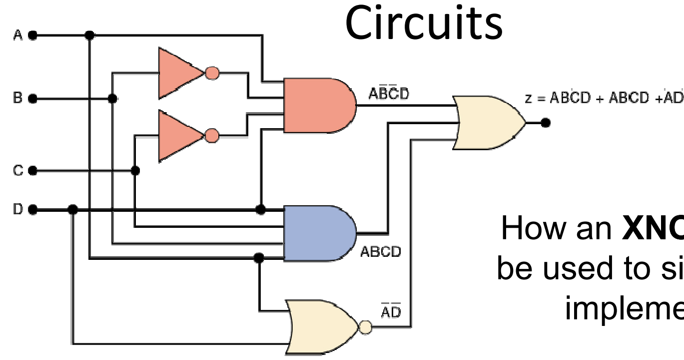
Exclusive OR and Exclusive NOR Circuits

Truth table and circuit for detecting equality of two-bit binary numbers.

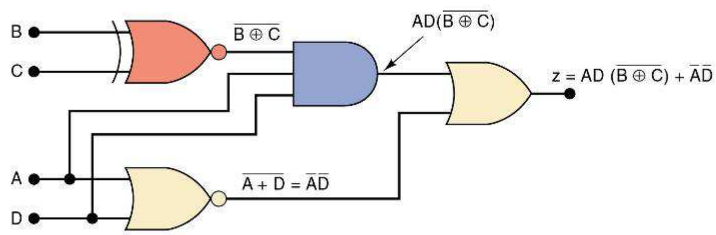


x_1	x_0	y_1	y_0	z (Output)
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Exclusive OR and Exclusive NOR Circuits



How an **XNOR** gate may be used to simplify circuit implementation.



Thank You

Next topic