Logic Circuit Design

CO 2206 Computer Organization

Topics

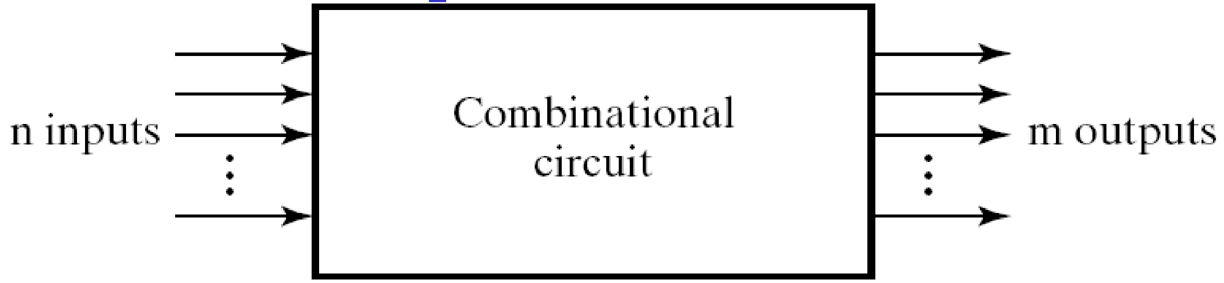
- The Process
- Standard Forms
- Simplification techniques
  - Algebraic manipulation
  - Karnaugh Map (K Map)
  - Quine-McCluskey method
- Implementation matters
  - Circuit Implementation
    - 2-level implementations
  - XOR Implementation
  - Hi-Z and Enable

Introduction

- Logic circuits for digital systems may be
  - *Combinational logic circuit/network (CLN)*
  - *Sequential logic circuit/network (SLN)*

Combinational Logic

- ***Combinational circuits***
  - Consist of logic gates whose outputs at any time are determined directly (and solely) from the present combination of inputs



  - For n input variables, there are 2^n possible binary input combinations
  - Combinational circuit can be described by m Boolean functions, one for each output variable

Sequential Logic

- Outputs of ***sequential circuits*** depend not only on present inputs, but also on stored values (states), which are a function of previously applied inputs
- Output determined by
  - inputs
  - present state of the storage elements
    - ‘previous’ outputs

Logic Design Process

- A simple *logic design process* involves
  1. ***Problem specification*** - discover the *input and output requirement*
  2. ***Problem formulation*** – e.g. derive a *truth table* from the input and output requirement
  3. Derivation of ***logical expression(s)*** – e.g. from the *truth table*, derive the *Boolean expression*
  4. ***Optimization*** – in simplest is to *minimize* the *Boolean expression(s)* if necessary, however more to it (cost factors)
  5. ***Implementation*** - build the *circuit(s)* from the simplest *Boolean expression(s)*
- If there are more than one output, we treat each output as a separate design or circuit

Design Example: the Problem

- Access to a compound that contains dangerous high voltage equipment can be gained by a maintenance electrician under the following conditions:
  - The *high voltage* is *off* (*Logic 0*).
  - A keyswitch on the *control panel* 100 yards away is *off* (*Logic 0*).
  - A keyswitch on the *gate* is turned *on* (*Logic 1*).
- Under all the other conditions the gate cannot physically be opened.

Design Example: Solution - 1

- *Step 1: Discover the input and output requirements*
  - The input and output requirements are given in the question
- *Step 2: Derive a truth table from the input and output requirement*
  - Assignment input variables and output function.

A	=	High voltage
B	=	Control panel switch
C	=	Gate switch

Design Example: Solution - 2

– Conditions for entry : $Q=1$
  - Requirements are $A=0$, $B=0$, $C=1$
– truth table

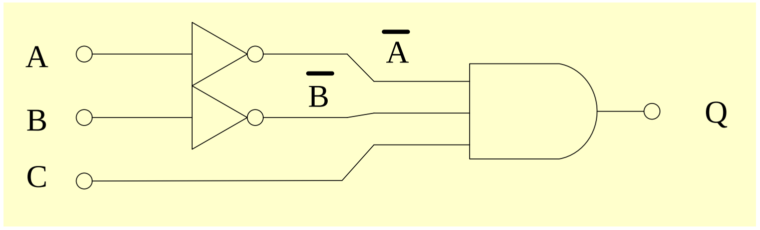
A	B	C	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Design Example: Solution - 3

- *Step 3: From the truth table, derive the Boolean expression*

$$Q = \overline{A}\,\overline{B}C$$

- *Step 4: Minimize the Boolean expression if necessary*
  - There is only one possible output condition and the expression is in its simplest form
- *Step 5: Build the circuit from the simplest Boolean expression*



Standard Forms

- *Standard forms* facilitate the simplification
- *Standard forms* contain
  - *Product terms*
    - e.g. xy'z
  - *Sum terms*
    - e.g. x + y + z'
- *Minterms*
  - Product terms in which all the variables appear exactly once, either primed or unprimed
- *Maxterms*
  - Sum terms in which all the variables appear exactly once

Deriving Logical Expression

- *Logical expression* can be expressed as:
  - *Sum of Minterms*
  - *Product of Maxterms*
- In *sum of minterms*
  - we specify combination inputs for which the output is 1
- In *product of maxterms*
  - we specify combination inputs for which the output is 0

Example - 1

- 3-input majority function
  - *Sum of minterms*

    $F_1 =$ A'BC + AB'C + ABC' + ABC

    $F_1 = \Sigma\, m(3,5,6,7)$
  - *Product of maxterms*

    $F'_1 =$ (A+B+C) (A+B+C') (A+B'+C) (A'+B+C)

    $F'_1 = \Pi\, M(0,1,2,4)$

A	B	C	F_1
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The selected *minterm*, A'BC, will give an output of 1 when A=0, B=1, C=1.

The selected *maxterm*, A'+B+C, will give an output of 0 (hence F') when A=1, B=0, C=0.

Example - 2

- 3-input even parity function
  - *Sum of minterms*

    $F_2 =$ A'B'C + A'BC' + AB'C' + ABC

    $F_2 = \Sigma\, m(1,2,4,7)$
  - *Product of maxterms*

    $F'_2 =$ (A+B+C) (A+B'+C') (A'+B+C') (A'+B'+C)

    $F'_2 = \Pi\, M(0,3,5,6)$

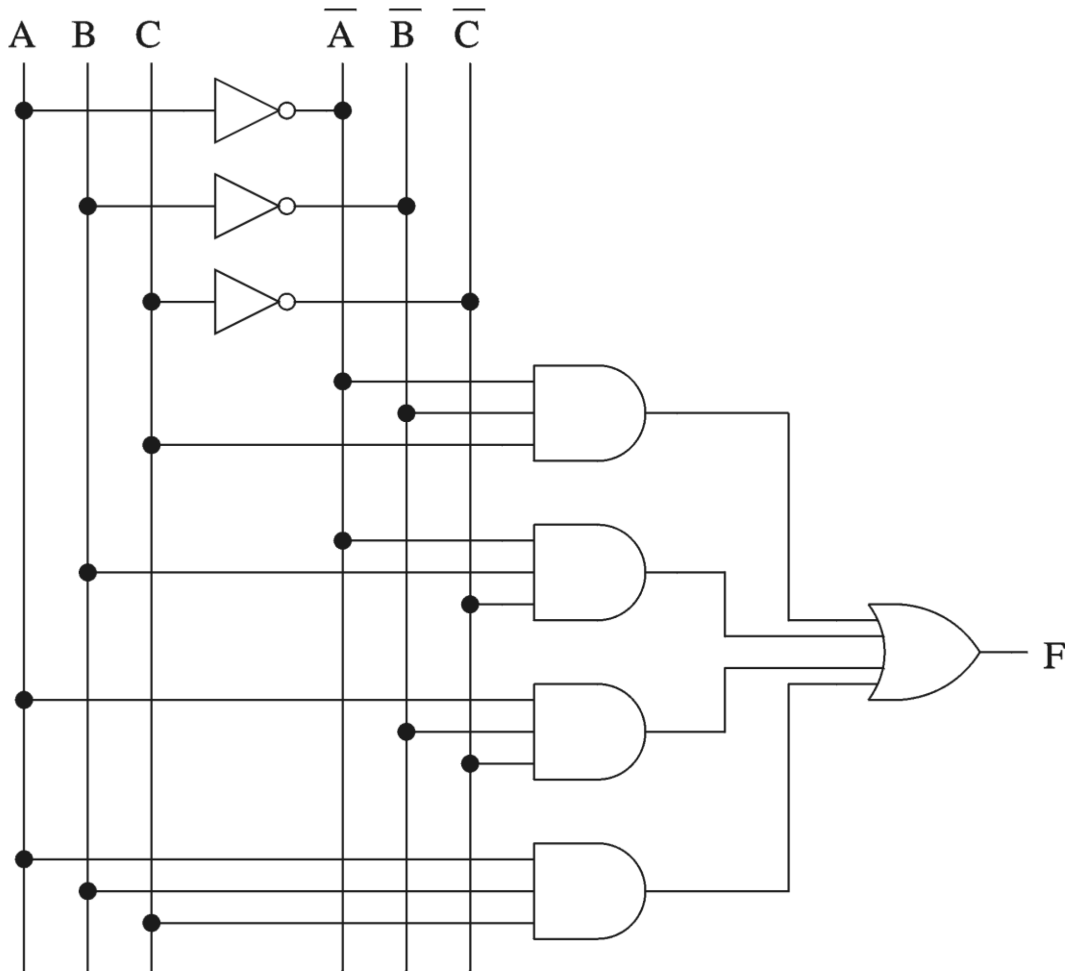
Note that the *maxterms* are expressed in inverted form, i.e. 000 = ABC and 111 = A'B'C'.

Whereas *minterms* are expressed non-inverted, i.e. 000 = A'B'C' and 111 = ABC

A	B	C	F_2
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

Straight Implementation

- Implementing 3-input even parity function
  - from sum of *minterms*



Simplifying Logical Expression

- *Sum of minterms* and *product of maxterms* can be obtained directly from the truth table,
  - but the expression contains maximum number of literals (or variables) in each term and usually has more terms than necessary
- It may require simplification

Simplifying Techniques

- 3 techniques
  - *Algebraic manipulation*
    - Do not know if expression is in final simplified form
  - *Karnaugh map* (*K map*)
    - Graphical method suitable for expression with 4 or less number of variables
  - *Quine-McCluskey methods*
    - Tabular method for simplifying expression with large no. of variables
    - Can be automated (programmed)

Algebraic Manipulation

- Using *Theorems of Boolean Algebra*
  - no fixed steps to follow
  - requires good intuition and experience
  - inherent problem of which rule to apply
- Example:

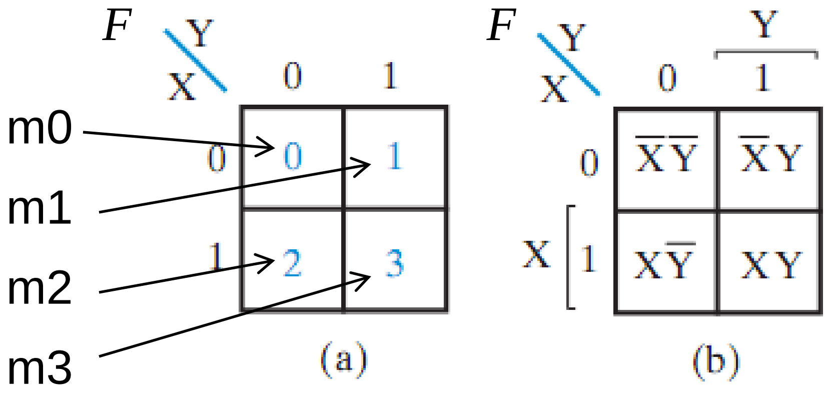
$$
\begin{aligned}
F_1 &= A'BC + AB'C + ABC' + ABC \\
&= A'BC + AB'C + ABC' + ABC + ABC + ABC \\
&= A'BC + ABC + AB'C + ABC + ABC' + ABC \\
&= BC(A'+A) + AC(B'+B) + AB(C'+C) \\
&= BC + AC + AB
\end{aligned}
$$

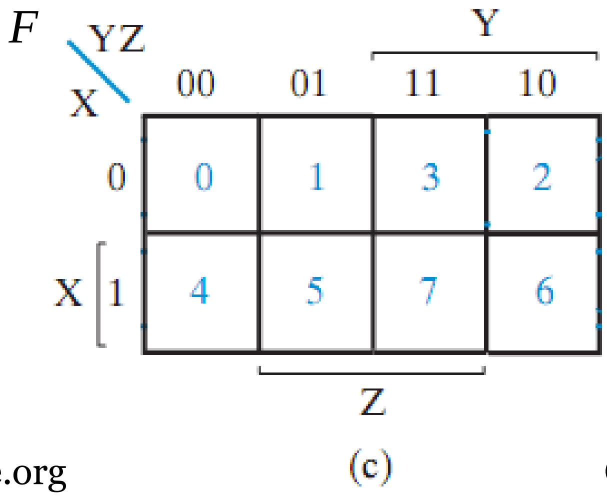
Karnaugh Map - 1

- ***Karnaugh Map*** (pronounced *car-no*), like a *truth table*, is a mean for showing the relationship between logic inputs and the desired output
- *Karnaugh map* is usually abbreviated ***K-map***. *K-map* can be used for problems involving *two*-, *three*-, *four*-, *five*- or *six*- different input variables
  - *K-map* for more than six-variable is practically impossible
  - Solving *five*- and *six*- variable *K-map* is extremely cumbersome; they can be more practically solved using advanced computer techniques
- In this course, we will only deal with *two*-, *three- and four*-variable *K-map*

K-Map Format - 1

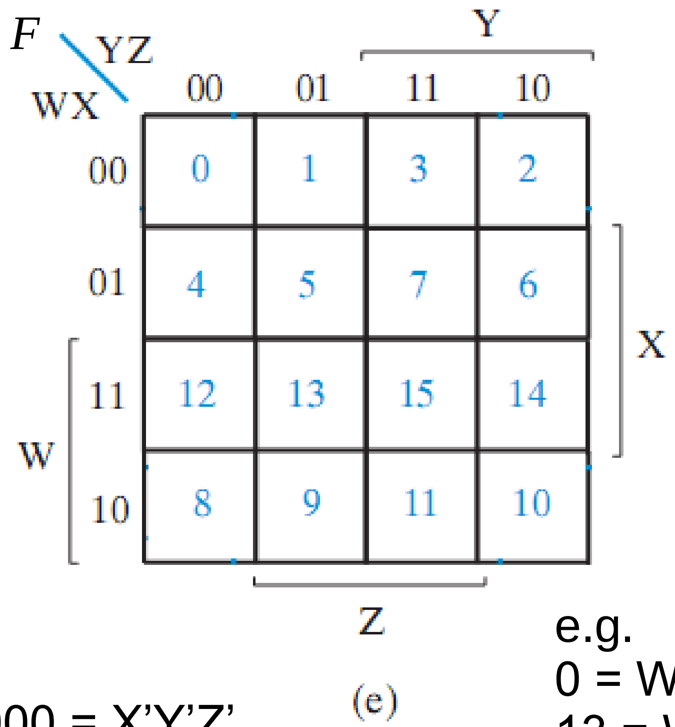
- Size depends on number of input variables, 2, 3, 4





	00	01	11	10
0	0	1	3	2
1	4	5	7	6

e.g.
0 = 000 = X'Y'Z'
5 = 101 = XY'Z



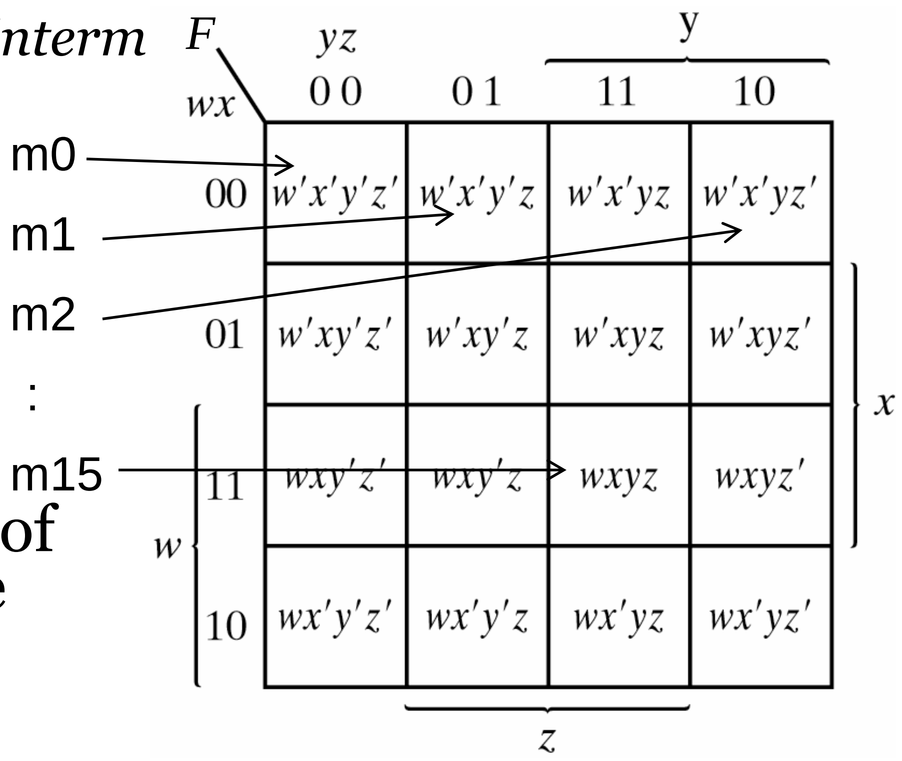
	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

e.g.
0 = W'X'Y'Z'
13 = WXY'Z

F is output function

K-Map Format - 2

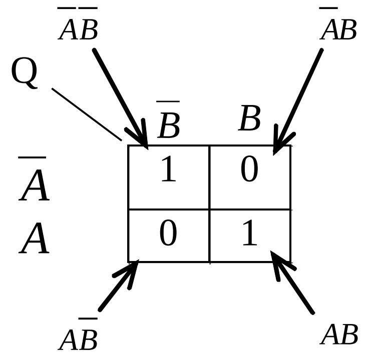
- *K-map* is a map describing all possible combinations of variables present in the logic function of interest
  - A *K-map* consists of 2^N cells, where *N* is the number of logic variables
  - each square represents one *minterm*
- *Minterms* are arranged in sequence similar to *Gray code*
- Any 2 adjacent cells differ by only one variable, which is primed in one cell and unprimed in another
- Possible to derive a number of algebraic expressions for the same function



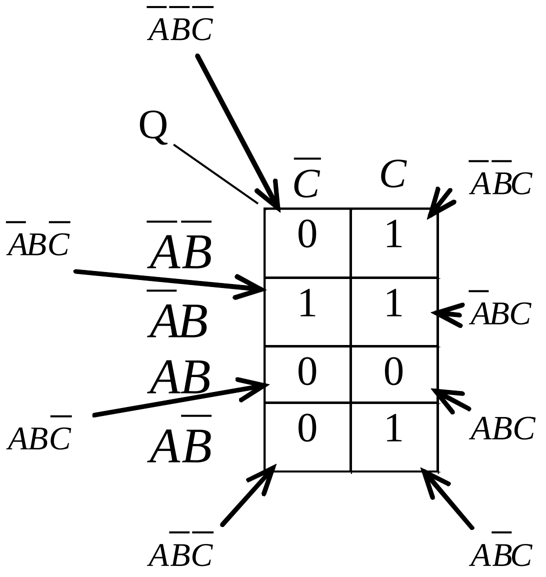
wx \ yz	00	01	11	10
00	$w'x'y'z'$	$w'x'y'z$	$w'x'yz$	$w'x'yz'$
01	$w'xy'z'$	$w'xy'z$	$w'xyz$	$w'xyz'$
11	$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$
10	$wx'y'z'$	$wx'y'z$	$wx'yz$	$wx'yz'$

K-Map Examples

Inputs		*Output*	
A	B	Q	
0	0	1	$\leftarrow \bar{A}\bar{B}$
0	1	0	$\leftarrow \bar{A}B$
1	0	0	$\leftarrow A\bar{B}$
1	1	1	$\leftarrow AB$



Inputs			*Output*	
A	B	C	Q	
0	0	0	0	$\leftarrow \bar{A}\bar{B}\bar{C}$
0	0	1	1	$\leftarrow \bar{A}\bar{B}C$
0	1	0	1	$\leftarrow \bar{A}B\bar{C}$
0	1	1	1	$\leftarrow \bar{A}BC$
1	0	0	0	$\leftarrow A\bar{B}\bar{C}$
1	0	1	1	$\leftarrow A\bar{B}C$
1	1	0	0	$\leftarrow AB\bar{C}$
1	1	1	0	$\leftarrow ABC$

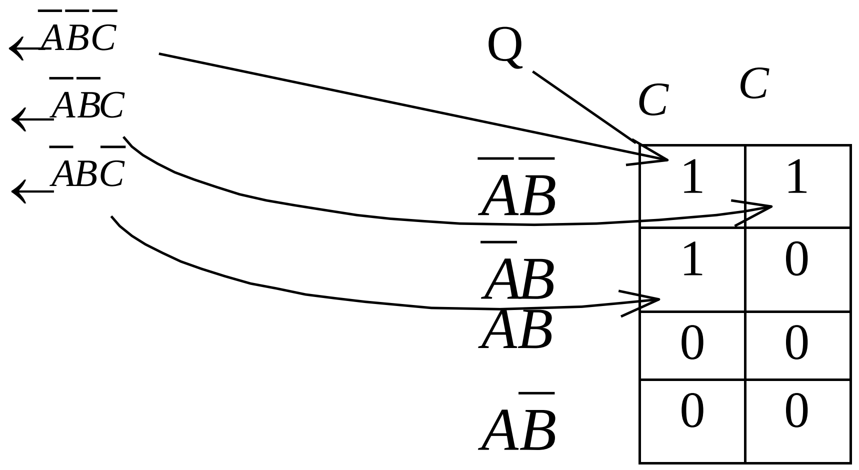


2-variable *K-Map* can be 2x4 or 4x2

From TT to K-Map

- To transfer a *truth table* into a *K-map*, we simply transfer the output level for each case of the *truth table* into the corresponding cell in *K-map*

Inputs			*Output*
A	B	C	Q
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



$\leftarrow \bar{A}\bar{B}\bar{C}$
$\leftarrow \bar{A}\bar{B}C$
$\leftarrow \bar{A}B\bar{C}$
Q
$\bar{C}$
C
$\bar{A}\bar{B}$
$\bar{A}B$
AB
$A\bar{B}$
1 1
1 0
0 0
0 0

From Expression to K-Map - 1

- The following steps can be followed to transform *Boolean expression* into *K-map*
  - Express the *Boolean expression* into *Sum-Of-Product (SOP) expression*. For example

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

$$= \underbrace{\bar{A}\bar{B}C + \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C}}_{\text{product terms}} \longleftarrow SOP$$

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

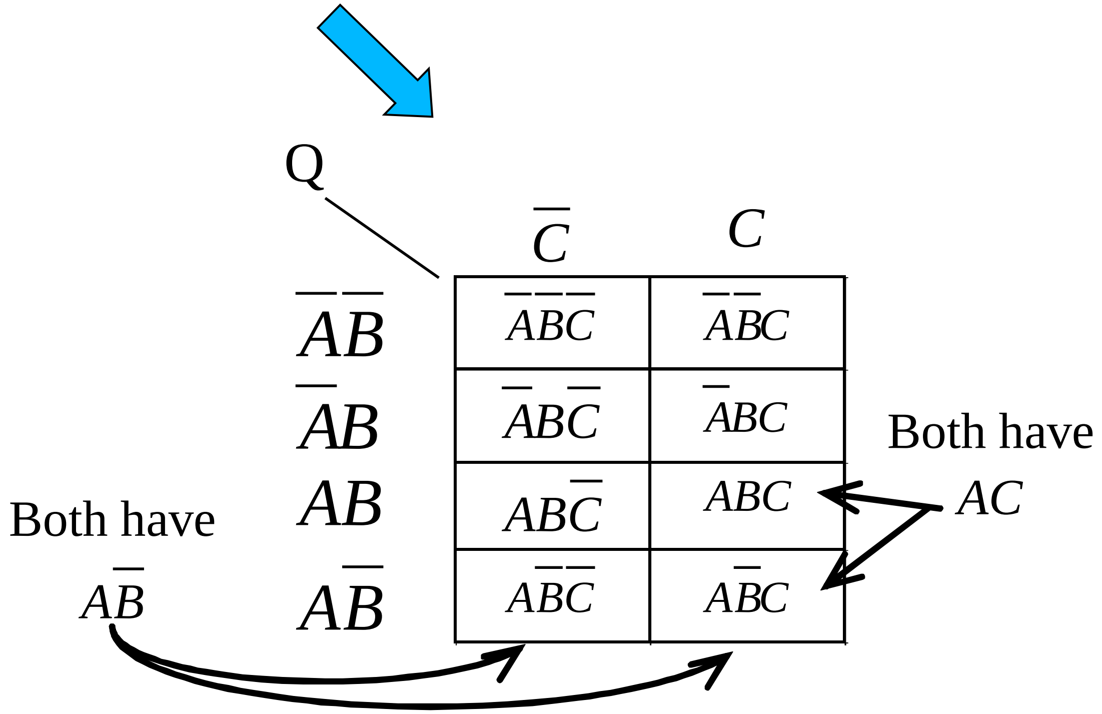
$$= \underbrace{A\bar{B} + AC}_{\text{product terms}} \longleftarrow SOP$$

Note the *product terms* need not be *minterms*

  - Fill in each of the *cells* which has the *product terms* with a "1“. See next slide for example.

From Expression to K-Map - 2

$$Q = A\overline{B} + AC$$



Q
$\overline{C}$
C
$\overline{A}\,\overline{B}$
$\overline{A}B$
AB
$A\overline{B}$
$\overline{A}\,\overline{B}\,\overline{C}$
$\overline{A}\,\overline{B}C$
$\overline{A}B\overline{C}$
$\overline{A}BC$
$AB\overline{C}$
ABC
$A\overline{B}\,\overline{C}$
$A\overline{B}C$
Both have $A\overline{B}$
Both have AC

$$\begin{array}{c|cc} Q & \overline{C} & C \\ \hline \overline{A}\,\overline{B} & 0 & 0 \\ \overline{A}B & 0 & 0 \\ AB & 0 & 1 \\ A\overline{B} & 1 & 1 \end{array}$$

From Expression to K-Map - 3

- Alternatively, derive the *truth table* from the *expression* and then transfer the *truth table* to *K-map*

Simplest Expression from K-Map

- The following steps can be followed to obtain a simplest *Boolean expression* from the *K-map*:
  - Encircle *adjacent cells* filled with "1" in *groups* of *2*, *4*, *8*, etc. (i.e. *power of 2*)
  - For each *group* or *circle*, find the *product term* which is common in all *cells* within the *group*.
  - The simplest expression is given by the *sum* of the *product terms* for all *groups*

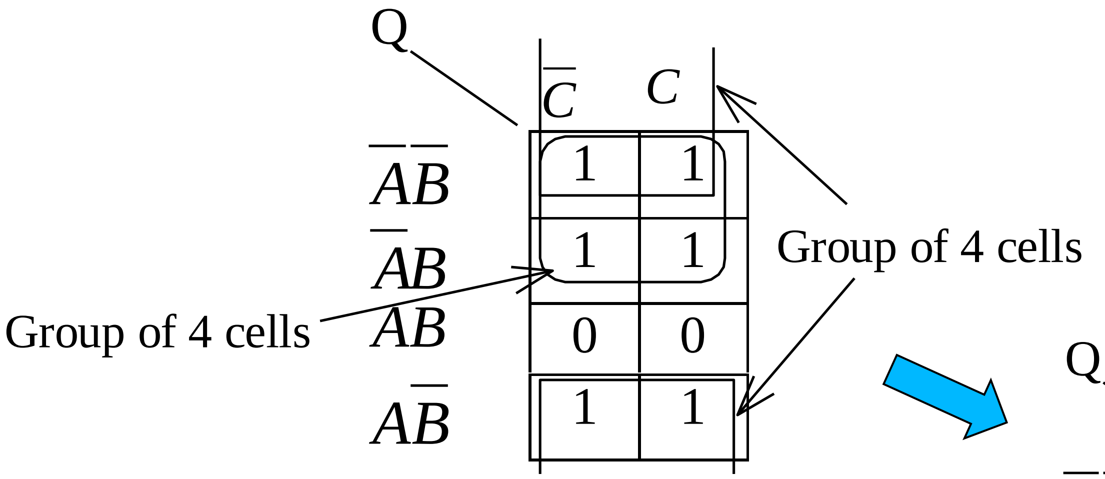
Grouping Rules

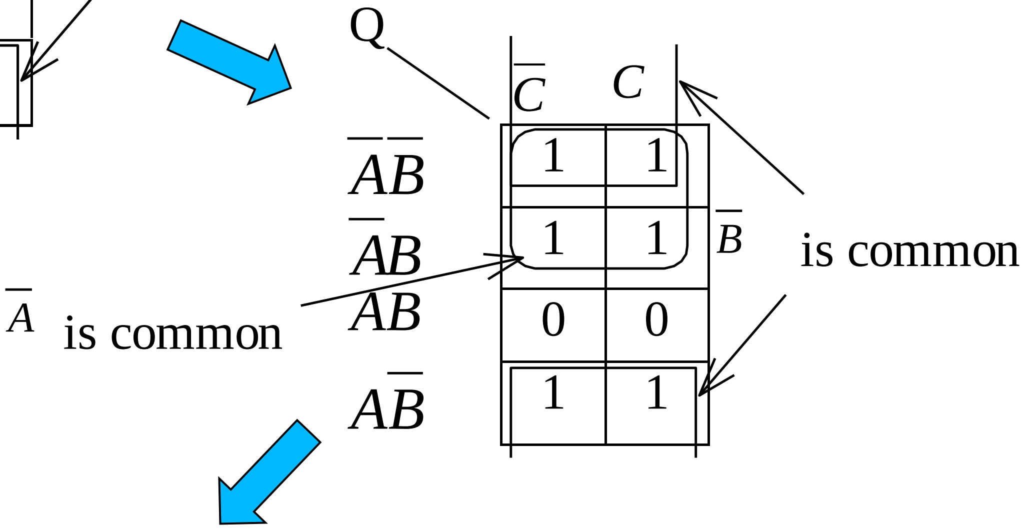
- The following ***grouping rules*** should be followed:
  - the number of cells must be a power of 2 using the rule 2^N
  - the more *adjacent cells* encircled, the simpler the final expression will be; for the simplest expression the maximum number of cells must be grouped
  - a cell can appear in more than one group
  - cells must have a common edge, i.e. the map can be imagined as a sphere opened out (just like the map of the World) so that the *top edge is adjacent to the bottom edge* and the *right edge is adjacent to the left edge*
  - all the "1" should be encircled

Some Grouping Terms

- The following terms are associated with K-maps:
  - ***Pair*** – group of 2 cells (with 1 less variable in the product term, i.e. for 3-variable function, a *pair* will be a product term with 2 variables)
  - ***Quad*** – group of 4 cells (with 2 less variables)
  - ***Octet*** – group of 8 cells (with 3 less variables)
  - ***Redundant group*** – a group with all its 1's already in other groups

Example - Grouping



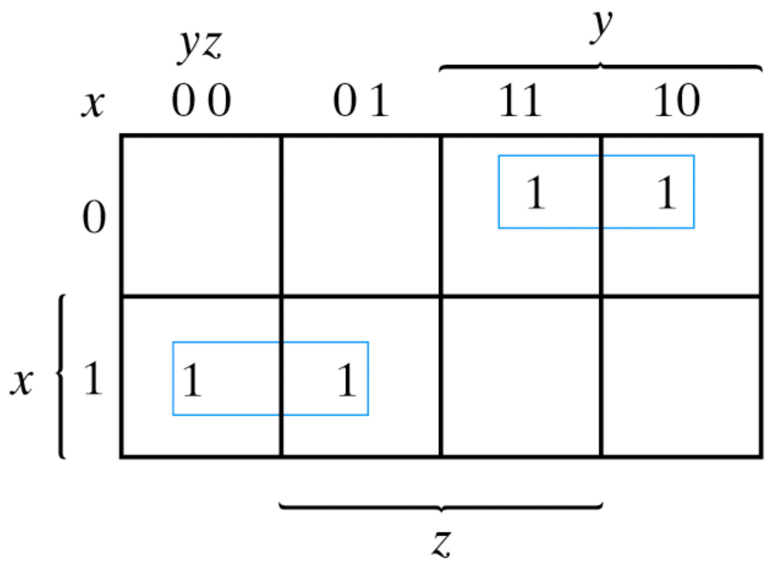


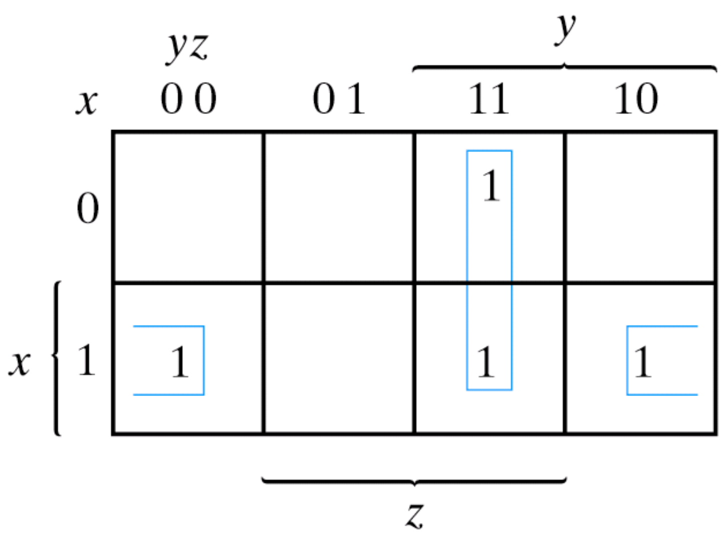
$$Q = \overline{A} + \overline{B}$$

SOP Method

- In summary, the following steps, called ***SOP Method***, are used to simplify *Boolean Equations*:
  - Enter a *1* on the *K-map* for each fundamental product that produces a *1* output in *truth table*. Enter *0* elsewhere.
  - Encircle the *octets*, *quads* and *pairs*. Remember to roll or overlap to get the largest groups possible
  - If any isolated *1* remains, encircle each
  - Eliminate any *redundant group*
  - Write the *Boolean Equation* by *ORing* the products corresponding to the encircled groups

3-variable Examples - 1

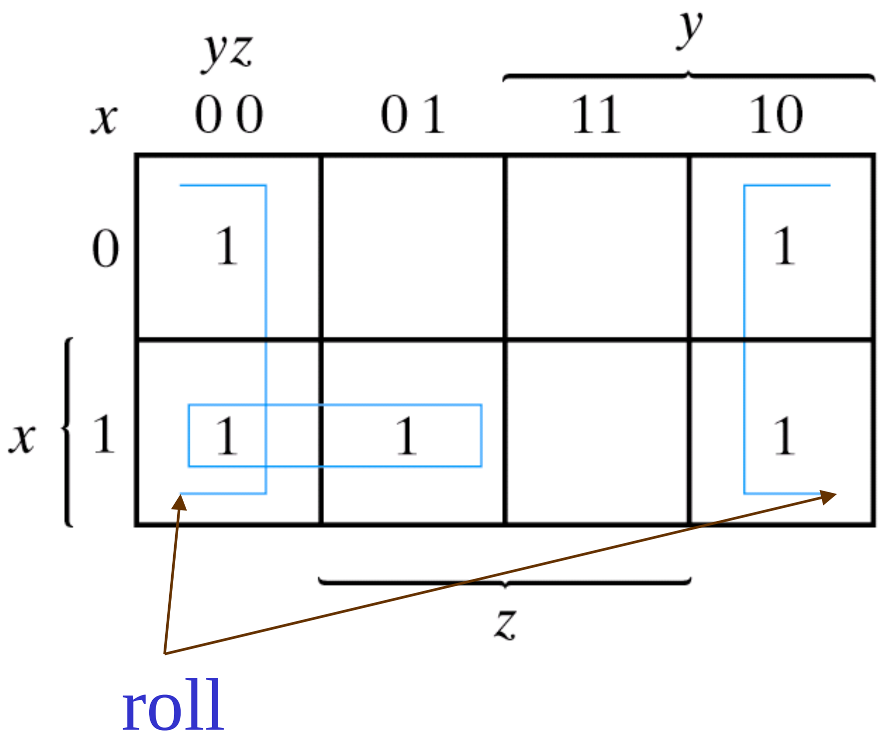


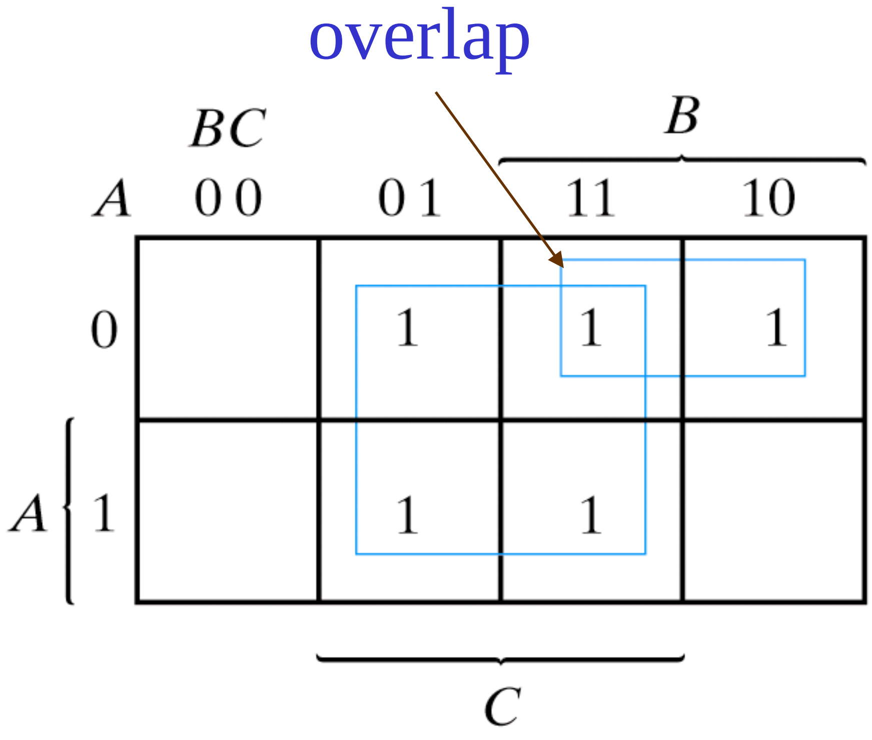


$F_1 = xy' + x'y$

$F_2 = yz + xz'$

3-variable Example - 2





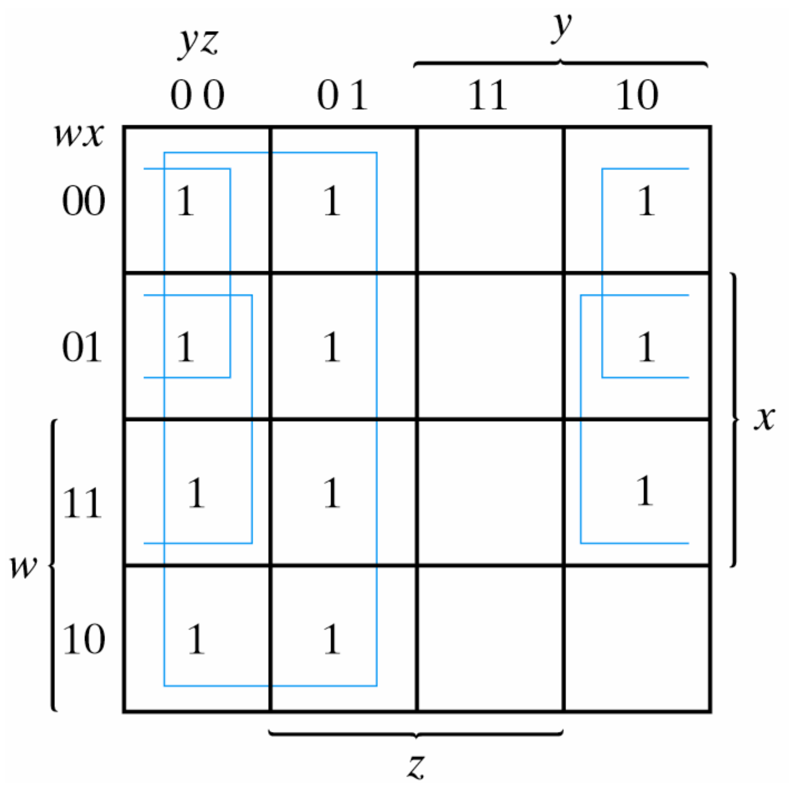
$F_3 = z' + xy'$

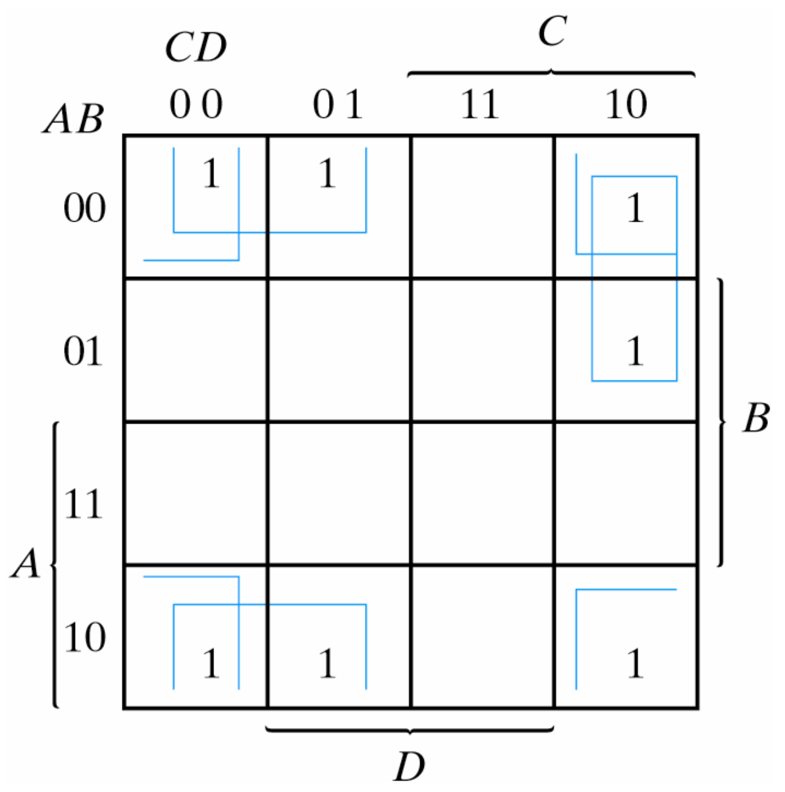
$F_4 = C + A'B$

3-variable Example - Observations

- More cells in the group, fewer literals in the product term
  - 1 cell represent 1 minterm, giving a term of 3 literals
  - 2 adjacent cells represent a term of 2 literals
  - 4 adjacent cells represent a term of 1 literal
  - adjacent cells encompass the entire map, function always equal to 1

4-variable Example - 1



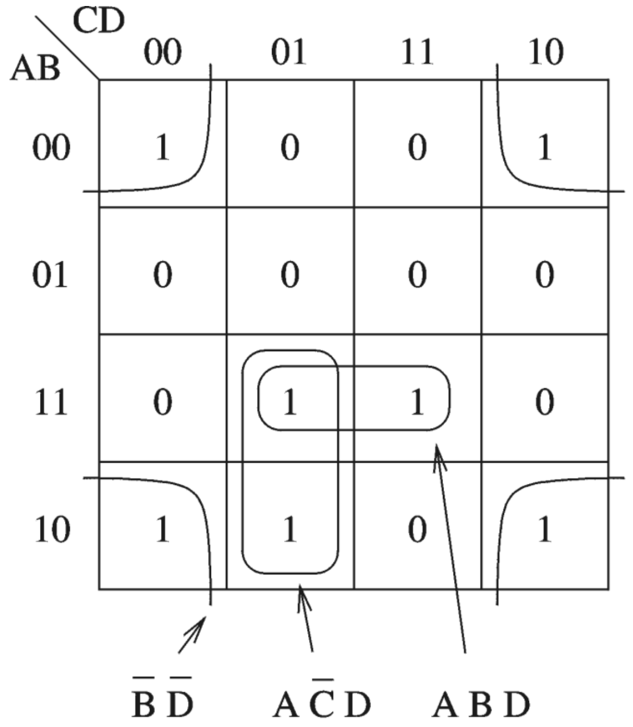


$F_5 = y' + w'z' + xz'$

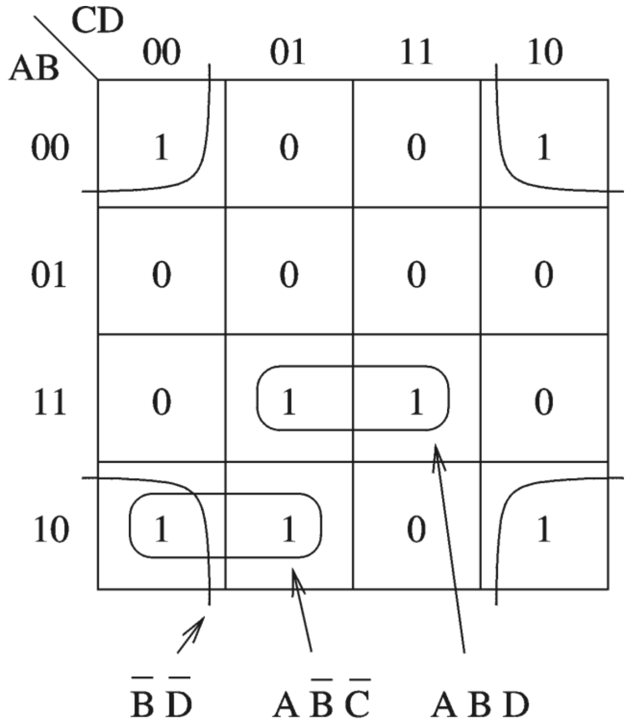
$F_6 = B'D' + B'C' + A'CD'$

4-variable Example - 2

- Minimal expression will depend on groupings



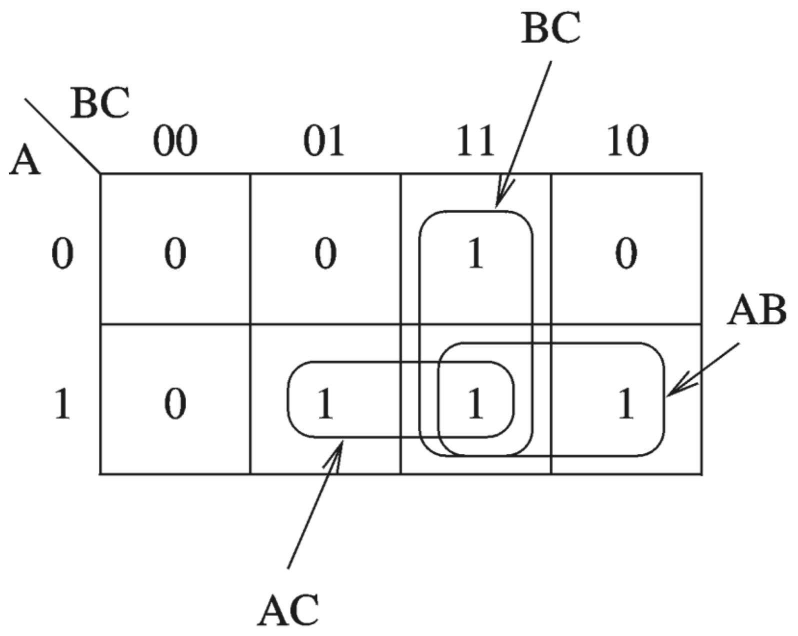
(a)



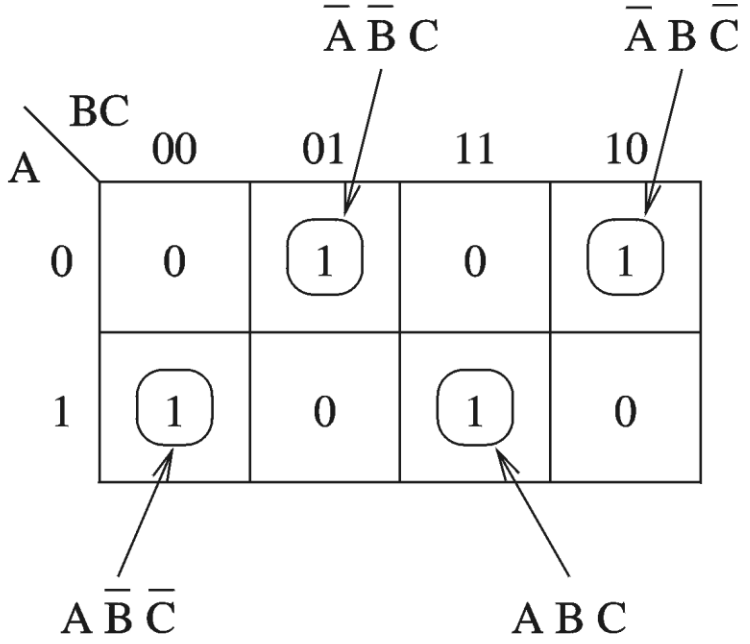
(b)

More Examples

F_2 = A'B'C+A'BC'+AB'C'+ABC
(not all functions can be simplified)



(a) Majority function



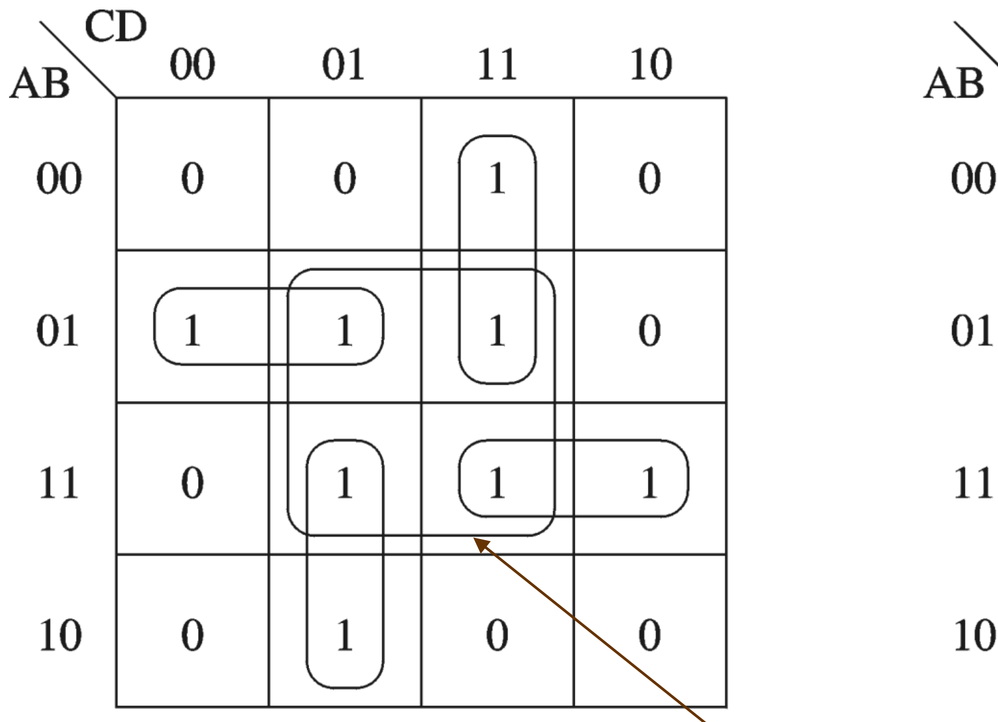
(b) Even-parity function

F_1 = AB+AC+BC
(before simplification: F = A'BC+AB'C+ABC'+ABC)

Essential Prime Implicants - 1

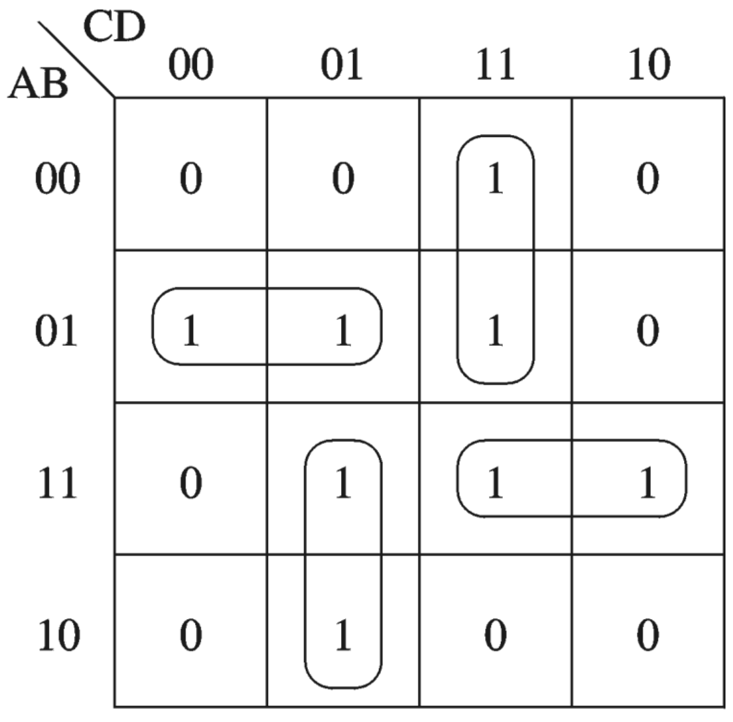
- On a *K map*, ***prime implicants*** correspond to all rectangles (groups) containing 1's
- If a *minterm* of a function is included in only one *prime implicant*, that *prime implicant* is said to be ***essential***
- Optimised expression obtained from
  - Sum of all *essential prime implicants*, plus
  - Other *prime implicants* needed to include remaining minterms not included in the essential prime implicants

Essential Prime Implicants - 2



AB \ CD	00	01	11	10
00	0	0	1	0
01	1	1	1	0
11	0	1	1	1
10	0	1	0	0

(a) Nonminimal simplification

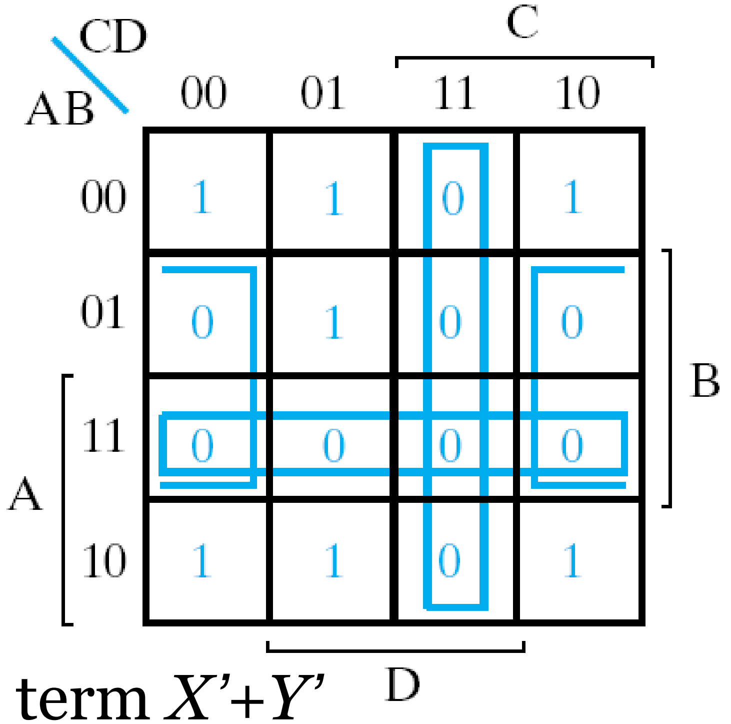


AB \ CD	00	01	11	10
00	0	0	1	0
01	1	1	1	0
11	0	1	1	1
10	0	1	0	0

(b) Minimal simplification

Product of Sums – the Alternative

- Cells with 1's give F, cells with 0's give F'
- Combining squares marked with 0's
  - F' = AB + CD + BD'
- Taking the dual
  - F = (AB + CD + BD')'
  - = (AB)'(CD)'(BD')'
- Dual each product term
  - F = (A'+B')(C'+D')(B'+D)
- Note 1st and 2nd step can be skipped by observing that product term *XY* becomes sum term *X'*+*Y'*

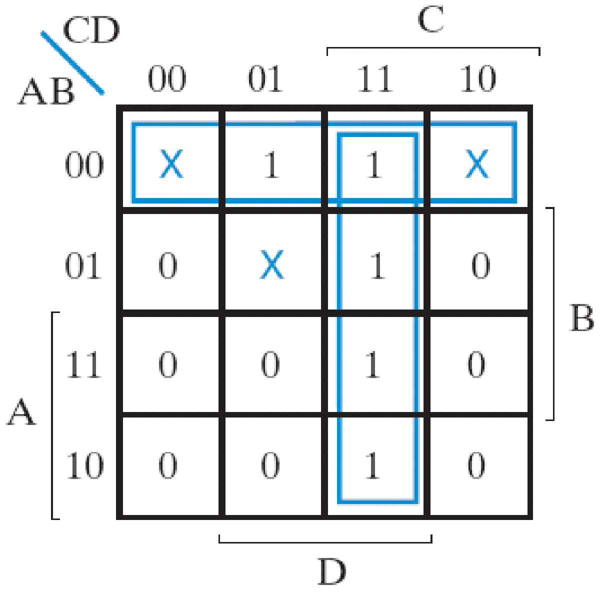


Don’t Care Conditions - 1

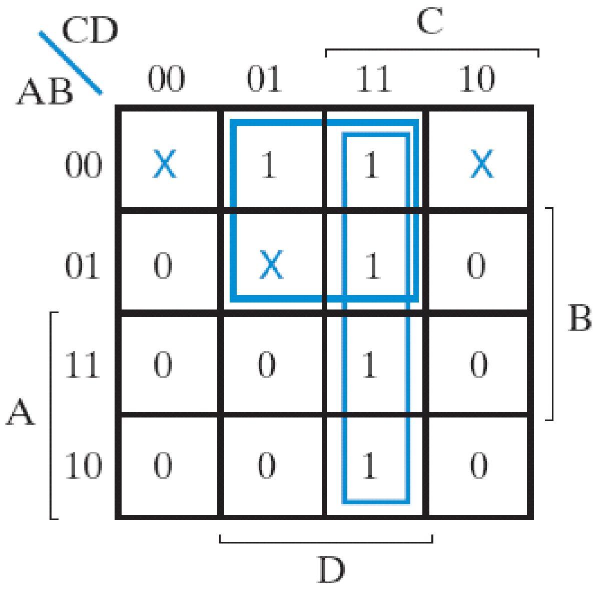
- In some applications, some outputs are not specified for certain combinations of variables because
  - The input combinations never occur
  - We do not care what the outputs are in response to the input combinations
- These outputs are unspecified and are called ***don’t care conditions***

Don’t Care Conditions - 2

- “*Don’t cares*” are marked ‘x’ in *K map* and may be assumed to be 0 or 1
  - assumptions are made in the way to produce simplest expression



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



(b) $F = CD + \overline{A}D$

Quine-McCluskey Method

- The ***Quine-McCluskey*** algorithm (or the method of prime implicants) is a method used for minimization of boolean functions which was developed by *W.V. Quine* and *Edward J. McCluskey*
- It is functionally identical to *K-map*, but the *tabular form* makes it more efficient for use in computer algorithms, and it also gives a deterministic way to check that the minimal form of a *Boolean* function has been reached
  - it is sometimes referred to as the ***tabulation method***
- The method involves two major steps:
  - ***Table*** - finding all *prime implicants* of the function
  - ***Chart*** - use those *prime implicants* in a *prime implicant chart* to find the *essential prime implicants* of the function, as well as other *prime implicants* that are necessary to cover the function

Quine-McCluskey: the Table

(a)

Group 0	A'B'C'D'	✓
Group 1	A'B'C'D	✓
	A'B'CD'	✓
	AB'C'D'	✓
Group 2	AB'CD'	✓
Group 3	AB'CD	✓
	ABCD'	✓
Group 4	ABCD	✓

(b)

A'B'C'	
A'B'D'	✓
B'C'D'	✓
B'CD'	✓
AB'D'	✓
AB'C	✓
ACD'	✓
ACD	✓
ABC	✓

(c)

B'D'
~~B'D'~~
AC
~~AC~~

Quine-McCluskey: Step 1.1

- Specify the function in *Sum of Minterms*
- Group the *minterms* in accordance to "number of true conditions (1's)" and arranged in a column (a). Example:
  - a'b'c'd' ≡ 0000 (no 1's) is in group 0
  - a'b'c'd ≡ 0001 (one 1's) is in group 1
  - ab'cd' ≡ 1010 (two 1's) is in group 2, etc
- *First iteration*: remove one variable from the *minterms* by looking at a pair of terms in *adjacent groups* that contain a variable and its complement e.g. a'b'c'd' + a'b'c'd = a'b'c'
  - equivalent to forming group of size 2 in *K-map*

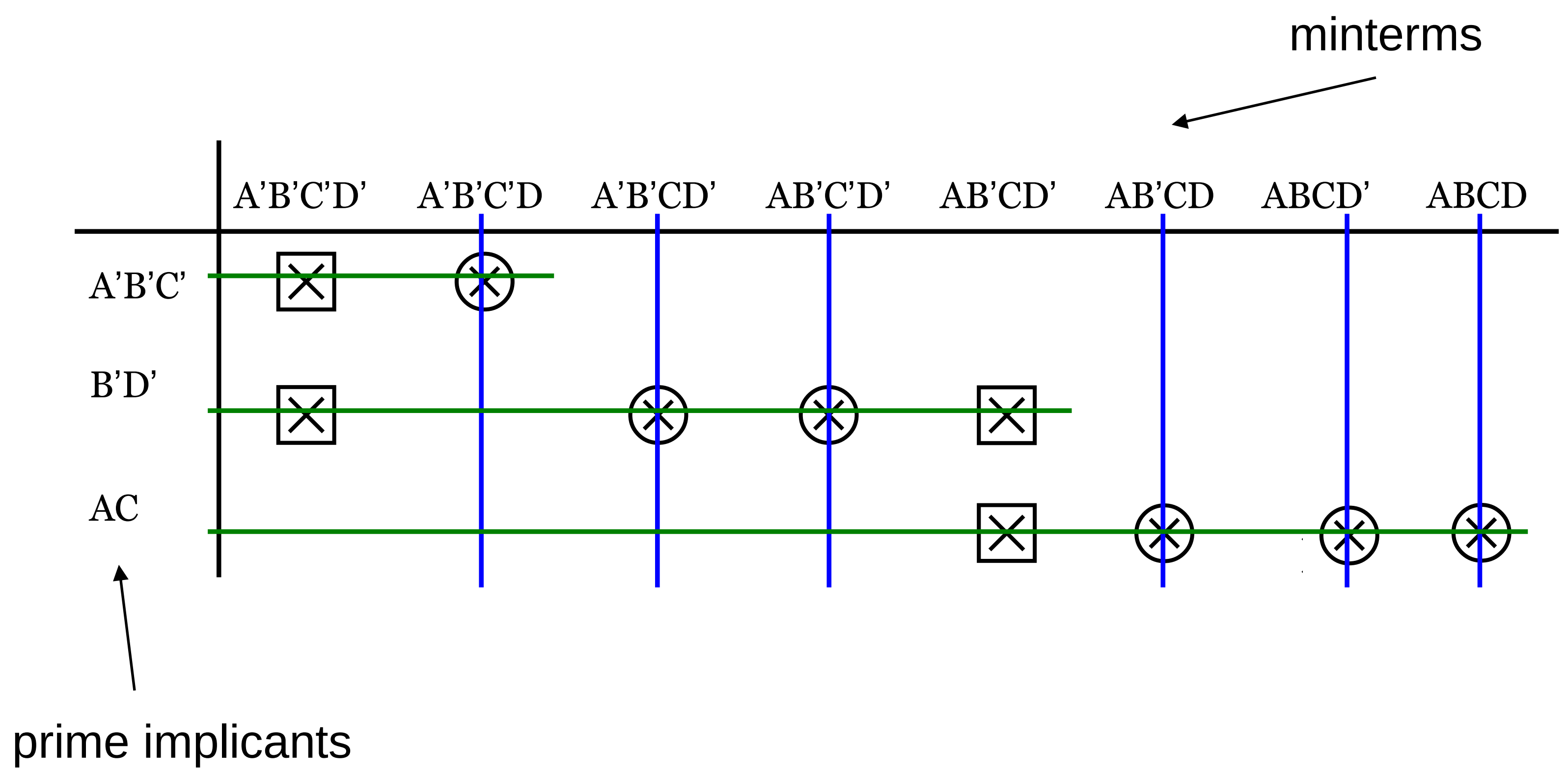
Quine-McCluskey: Step 1.2

- From the first iteration:
  - a set of product terms (with one less variable) is generated
  - these product terms are grouped in number of true conditions (1's) and arranged in next column (b)
  - terms not simplified in the first iteration will be left in column (a).
- *Second iteration*: remove one variable from the terms in (b) by looking at a pair of terms in *adjacent groups* that contain a variable and its complement e.g. a'b'c' + a'bc' = a'c'
  - equivalent to forming group of size 4 (2x2) in *K-map*
  - a set of product terms (with one less variable) is generated
  - these product terms are grouped in number of true conditions (1's) and arranged in next column (c)
  - terms not simplified in this iteration will be left in column (b)

Quine-McCluskey: Step 1.3

- Continue the iteration until no further reduction (in variable) can be done. In each iteration:
  - tick those terms that have been reduced
  - cross (remove) duplicating terms generated in each iteration
  - leave those terms that cannot be reduced unticked
- The outcome of the iterations is a table, where each column to the right has one less variable (reduction)
  - those terms not ticked are the *prime implicants*
- The next step is to draw the *Prime Implicant Chart*

Quine-McCluskey: the Chart

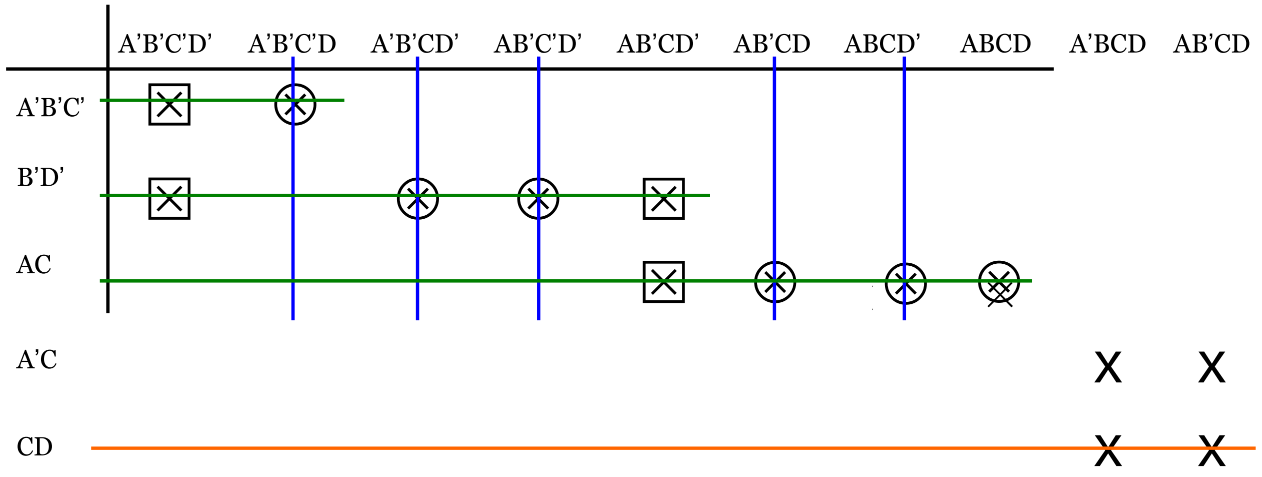


Quine-McCluskey: Step 2.1

- One row for each *prime implicant*
- One column for each *minterm* in original expression
- Mark $\times$ where prime *implicant* for row is in column terms
- Circle each $\times$ that is alone in a column. These are *essential prime implicants* that must appear in any final simplified expression.
- Place a square around all $\times$ in a row that has a $\otimes$. This indicates those *minterms* with $\boxtimes$ under it (as in column) has been included together with the *essential prime implicants* (those marked $\otimes$).

Quine-McCluskey: Step 2.2

- If there are columns without a ⊗ or ⊠
  - select a minimum number of *prime implicants* to cover these columns
- To incorporate *don't care* conditions
  - include the *don't care* terms in the 1st step (table) and then ignore them as we apply the 2nd step (chart)



Quine-McCluskey Example

- Simplify $F = \Sigma m(0,1,2,8,10,11,14,15)$

no.		ABCD		min	group
0	=	0000	≡	A'B'C'D'	0
1	=	0001	≡	A'B'C'D	1
2	=	0010	≡	A'B'CD'	1
8	=	1000	≡	AB'C'D'	1
10	=	1010	≡	AB'CD'	2
11	=	1011	≡	AB'CD	3
14	=	1110	≡	ABCD'	3
15	=	1111	≡	ABCD	4

Quine-McCluskey Example: Table

(a)

Group 0	0	A'B'C'D'	✓
Group 1	1	A'B'C'D	✓
	2	A'B'CD'	✓
	8	AB'C'D'	✓
Group 2	10	AB'CD'	✓
Group 3	11	AB'CD	✓
	14	ABCD'	✓
Group 4	15	ABCD	✓

(b)

A'B'C' (0,1)	
A'B'D' (0,2)	✓
B'C'D' (0,8)	✓
B'CD' (2,10)	✓
AB'D' (8,10)	✓
AB'C (10,11)	✓
ACD' (10,14)	✓
ACD (11,15)	✓
ABC (14,15)	✓

(c)

B'D'
~~B'D'~~
AC
~~AC~~

Quine-McCluskey Example: Chart

	A'B'C'D'	A'B'C'D	A'B'CD'	AB'C'D'	AB'CD'	AB'CD	ABCD'	ABCD
A'B'C'	⊠	⊗						
B'D'	⊠		⊗	⊗	⊠			
AC					⊠	⊗	⊗	⊗

F = A'B'C' + B'D' + AC

Quine-McCluskey Exercise

- Simplify

  $F(A,B,C,D,E) = \Sigma m(0,1,4,5,16,17,21,25,29)$

Quine-McCluskey Exercise: Table

0	A’B’C’D’E’	*
1	A’B’C’D’E	*
	A’B’CD’E’	*
	AB’C’D’E’	*
2	A’B’CD’E	*
	AB’C’D’E	*
3	AB’CD’E	*
	ABC’D’E	*
4	ABCD’E	*

0	A’B’C’D’	*
	A’B’D’E’	*
	B’C’D’E’	*
1	A’B’D’E	*
	B’C’D’E	*
	A’B’CD’	*
	AB’C’D’	*
2	B’CD’E	*
	AB’D’E	*
	AC’D’E	*
3	ACD’E	*
	ABD’E	*

0	B’C’D’	
	A’B’D’	
	B’C’D’	-
	A’B’D’	-
	B’C’D’	-
1	B’D’E	
	B’D’E	-
2	AD’E	
	AD’E	-

Quine-McCluskey Exercise: Chart

	A’B’C’D’E’	A’B’C’D’E	A’B’CD’E’	AB’C’D’E’	A’B’CD’E	AB’C’D’E	AB’CD’E	ABC’D’E	ABCD’E
B’C’D’	x	x		x		x			
A’B’D’	x	x	x		x				
B’D’E		x			x	x	x		
AD’E						x	x	x	x

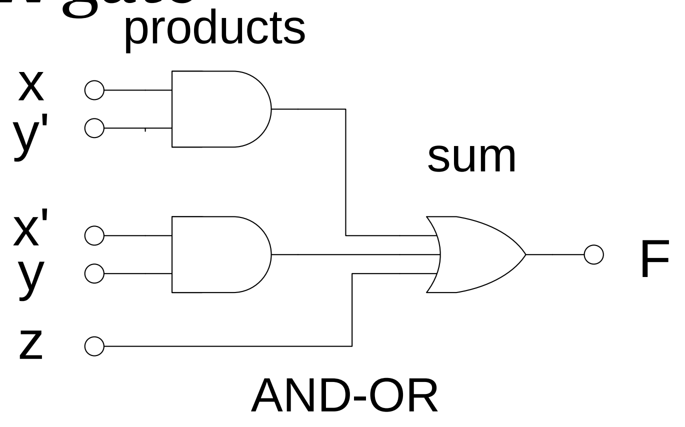
F = B’C’D’+A’B’D’+AD’E

Implementation

- Simple circuit can be directly implemented (from the expression)
- Technology mapping
  - Transform the logic diagram or netlist to a new diagram or netlist that implementation technology supports, e.g. NAND
  - Optimization and mapping may repeat multiple times to meet technology specifications, e.g. e.g. gate cost, gate delay, fan-out limits, etc
- Verification
  - Verify the correctness of final design

2-Level Implementations - 1

- Implementation can be directly using the Sum-*of-Product* (SOP) function, which is a *2-level AND-OR implementation*: the *AND* gates generates the *product terms*, which the outputs are summed by an *OR* gate
- e.g. F = xy'+x'y+z

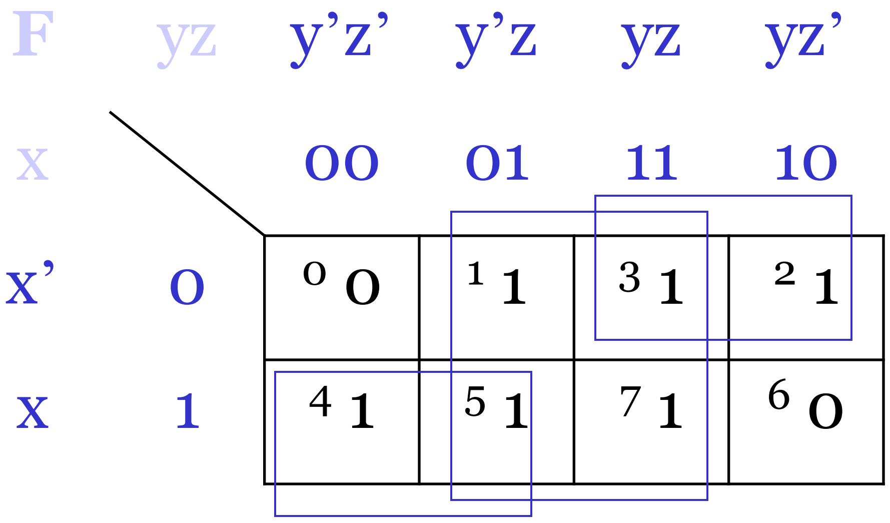


NOT gates are required to implement the inverted literals, however they are not usually considered a level

2-Level Implementations - 2

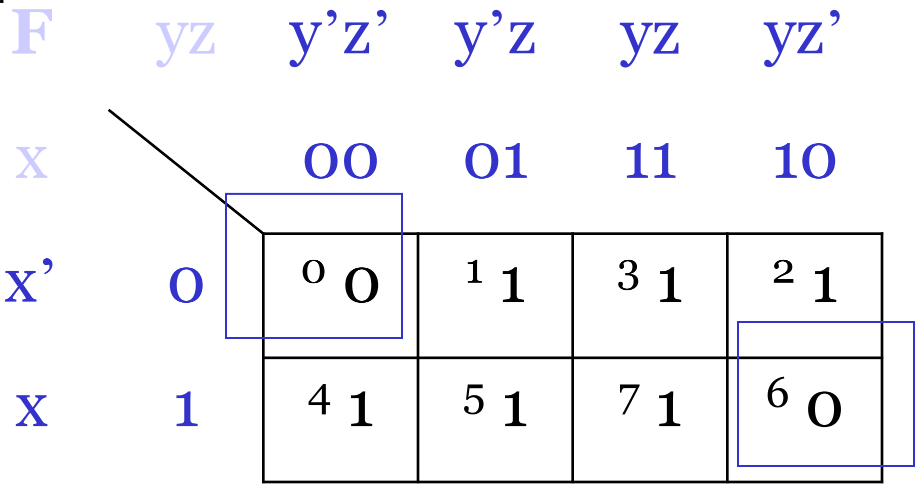
- By converting SOP into other forms, including POS, *2-level implementations* can be either:
  - AND-OR from SOP
  - NOR-OR
  - NAND-NAND
  - OR-NAND
  - AND-NOR
  - NOR-NOR
  - NAND-AND
  - OR-AND from POS

An Example Function



F	yz	y'z'	y'z	yz	yz'
x		00	01	11	10
x'	0	^{0} 0	^{1} 1	^{3} 1	^{2} 1
x	1	^{4} 1	^{5} 1	^{7} 1	^{6} 0

$F = xy' + x'y + z$



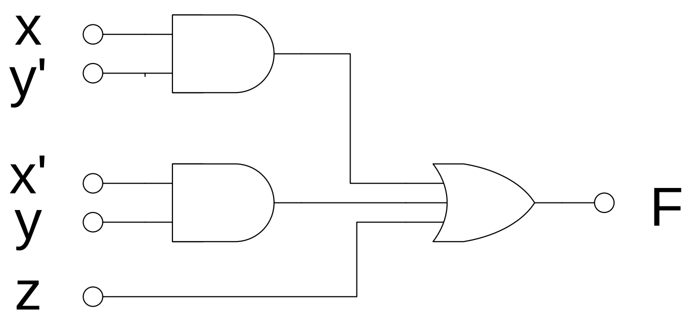
F	yz	y'z'	y'z	yz	yz'
x		00	01	11	10
x'	0	^{0} 0	^{1} 1	^{3} 1	^{2} 1
x	1	^{4} 1	^{5} 1	^{7} 1	^{6} 0

$F' = x'y'z' + xyz'$

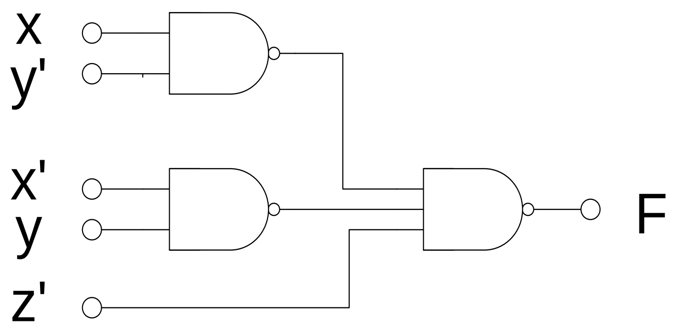
From F to other forms - 1

- By applying *De Morgan's Duality*, *F* can be converted into three other forms for *2-level implements*:
  - original F
    - F = xy'+x'y+z (AND-OR)
  - applying duality to each product term in original F
    - F = (x'+y)'+(x+y')'+z (NOR-OR)
  - applying duality to both side of original F
    - F' = (xy'+x'y+z)' = (xy')'(x'y)'z'
      F = ((xy')'(x'y)'z')' (NAND-NAND)
  - applying duality on each product term of NAND-NAND function
    - F = ((x'+y)(x+y')z')' (OR-NAND)

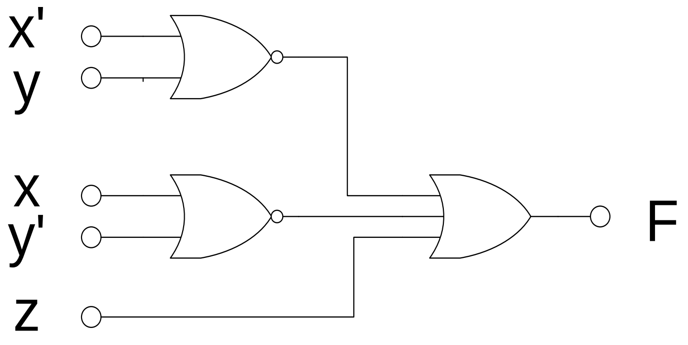
From F to other forms - 2



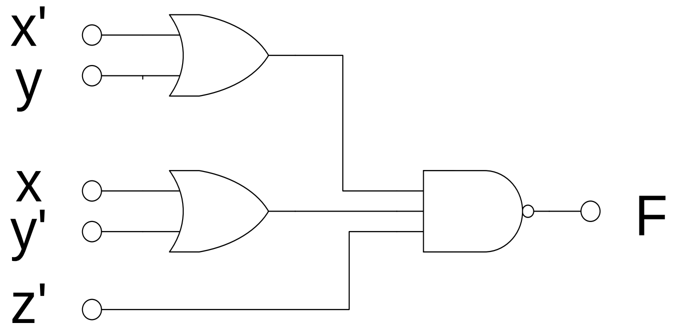
AND-OR



NAND-NAND



NOR-OR

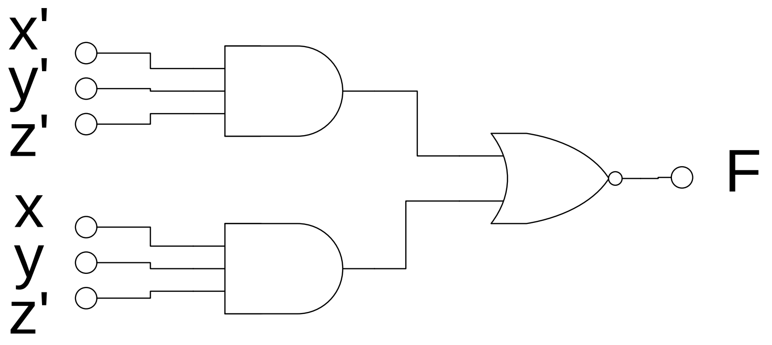


OR-NAND

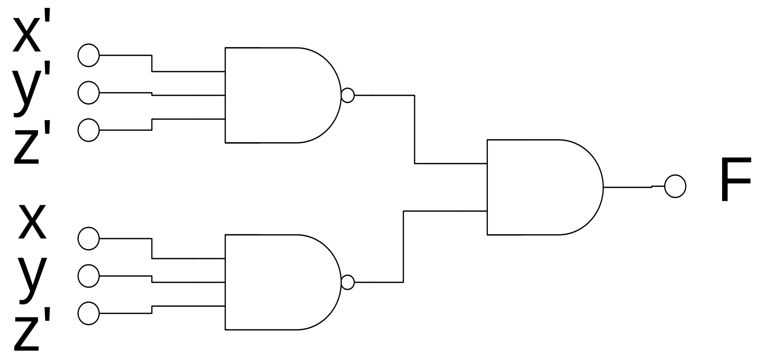
From F’ to other forms - 1

- By applying *De Morgan’s Duality*, *F’* can be converted into four different forms for *2-level implements*:
  - original F’
    - $F' = x'y'z' + xyz'$
    - $F = (x'y'z' + xyz')'$ (AND-NOR)
  - applying duality to each product term in AND-NOR function
    - $F = ((x+y+z)' + (x'+y'+z))'$ (NOR-NOR)
  - applying duality to AND-NOR function
    - $F = (x'y'z')'(xyz')'$ (NAND-AND)
  - applying duality to each term in NAND-AND function
    - $F = (x+y+z)(x'+y'+z)$ (OR-AND)

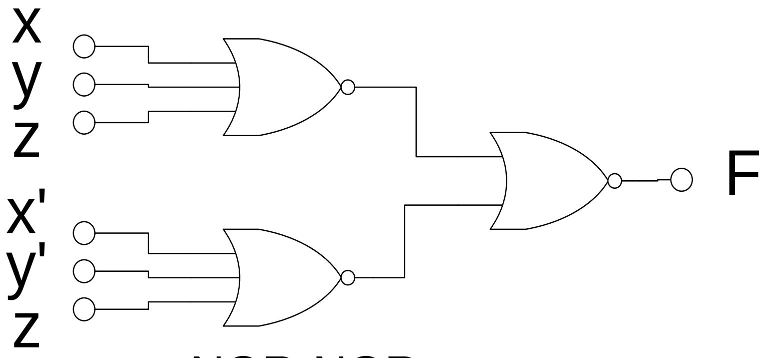
From F’ to other forms - 2



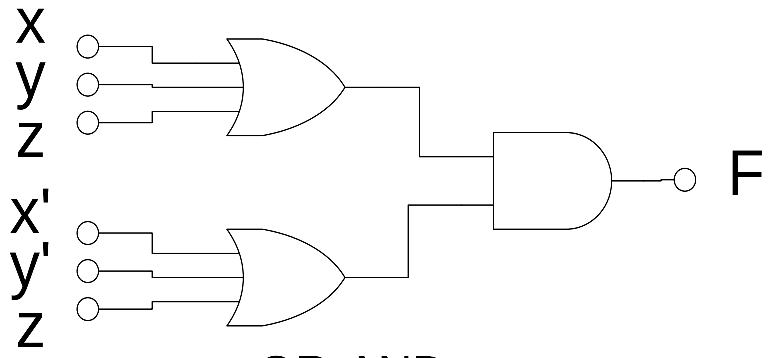
AND-NOR



NAND-AND

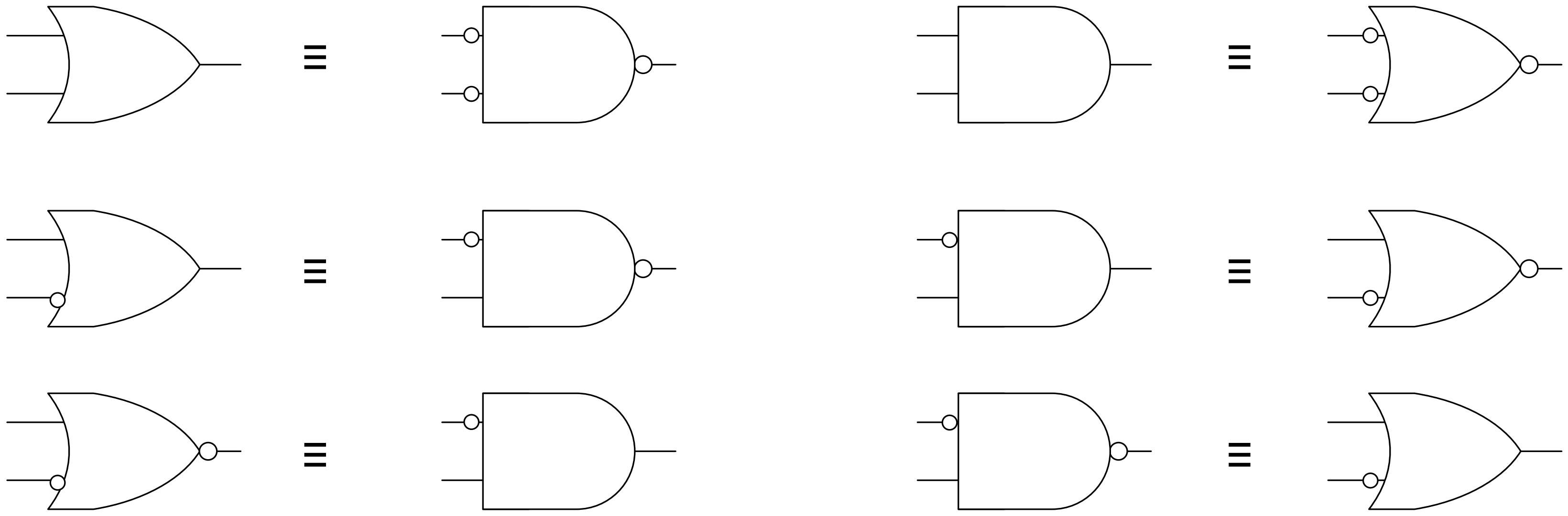


NOR-NOR



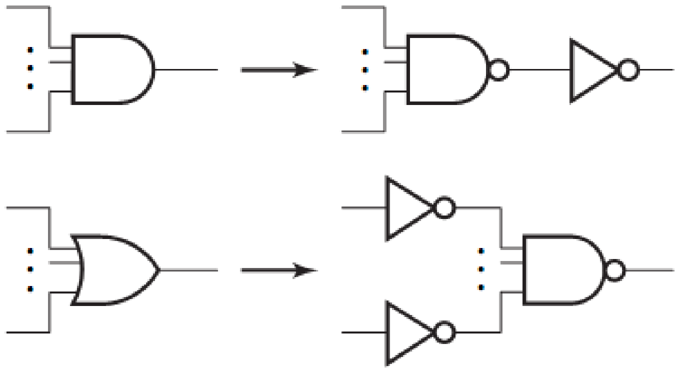
OR-AND

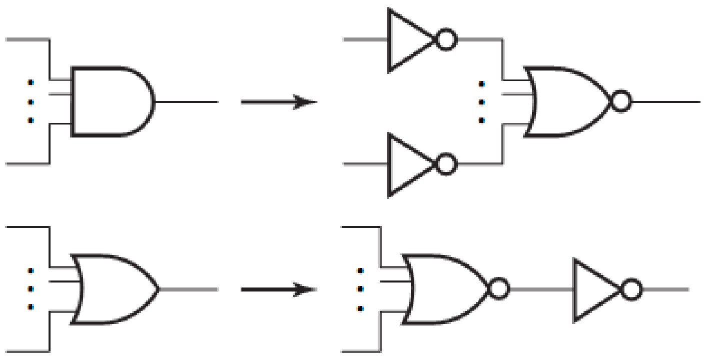
Duality on Logic Gate



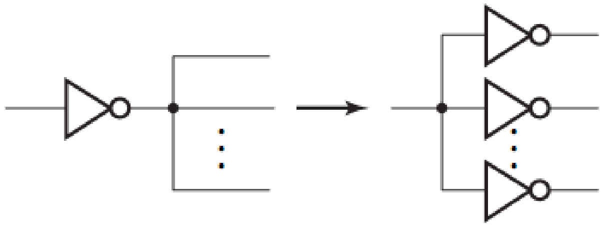
Technology Mapping

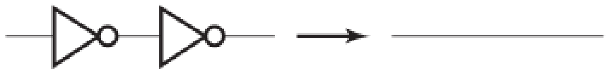
- Mapping to NAND or NOR


(a) Mapping to NAND gates


(b) Mapping to NOR gates

- Fine tuning the circuit for optimization with the selected technology

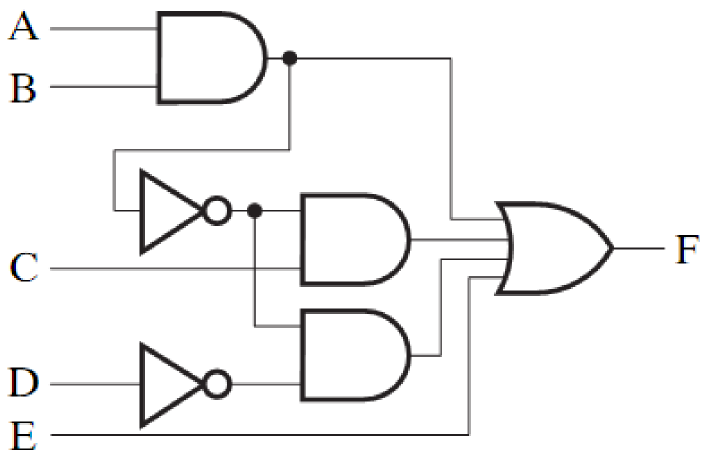

(c) Pushing an inverter through a "dot"


(d) Cancelling inverter pairs

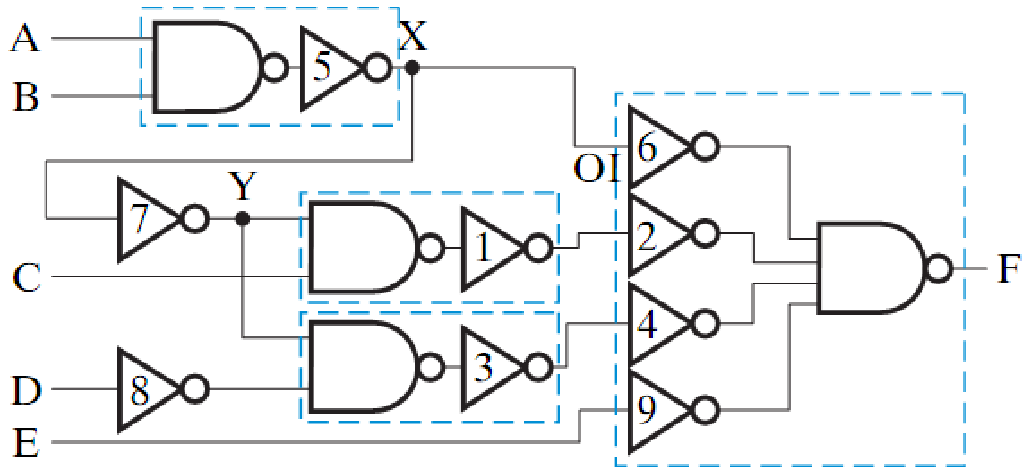
Technology Mapping: NAND

- NAND technology
  - Consists of a collection of cell types
    - each of which includes a NAND gate with fixed number of inputs
    - the cells have numerous properties
      - e.g. propagation delay, fan-in, fan-out, etc
- E.g. Implement F = AB + (AB)'C + (AB)'D' + E with NAND gates
  - next slide

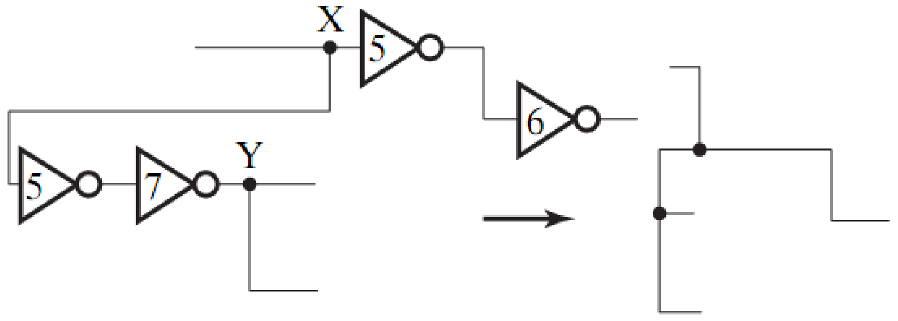
NAND Example



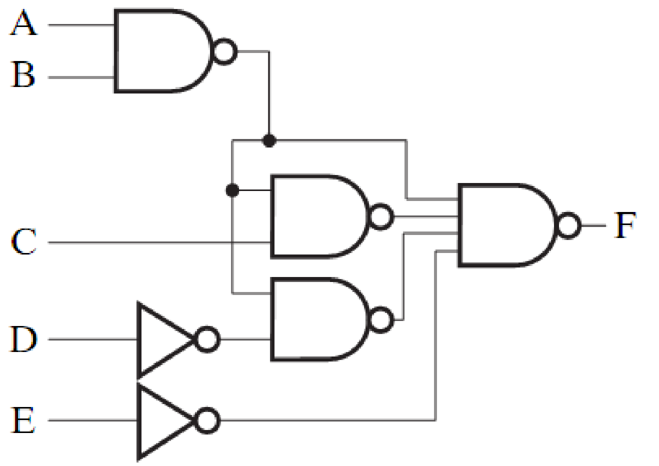
(a)



(b)



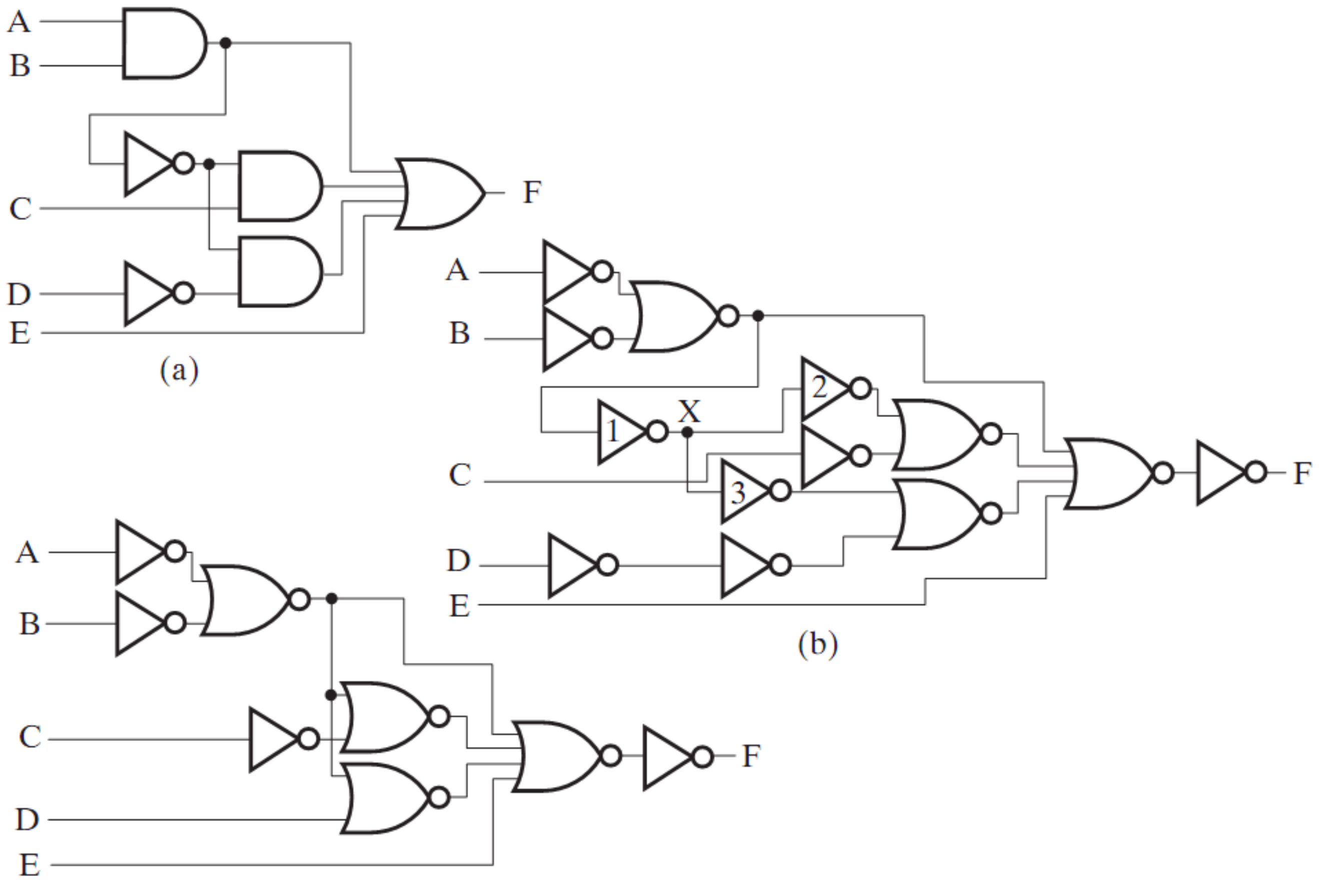
(c)



(d)

NOR Example

- Implement F = AB + (AB)'C + (AB)'D' + E with NOR gates



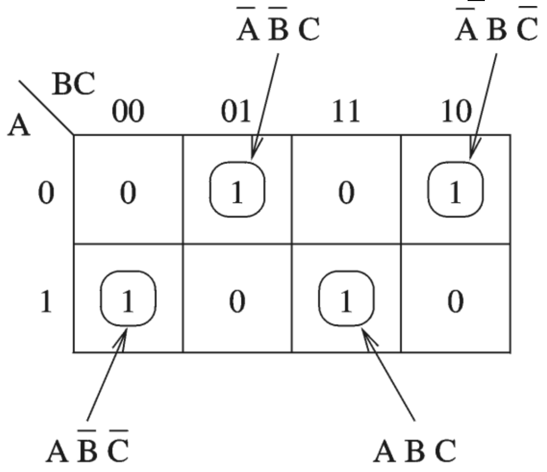
(c)

NAND vs NOR

- Comparisons
  - Gate-input cost
    - NAND implementation is 12
    - NOR implementation is 14
  - Gate delay
    - NAND – max. 3 gates in series
    - NOR – max. 5 gates in series
  - So for the e.g. NAND circuit is superior to NOR circuit in both cost and delay

Implementation using XOR

- If the true conditions (1's) in a K-map is scattered, it may be difficult to obtain a simple expression. An example is the even parity function earlier.



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

(b) Even-parity function

- However, it may be possible to obtain simplified expression using XOR operators

XOR Identities

XOR: $x \oplus y = xy' + x'y$ **XNOR**: $(x \oplus y)' = xy + x'y$

Basic theorems

T1. $x \oplus x = 0$ T2. $x \oplus x' = 1$ T3. $x \oplus 0 = x$ T4. $x \oplus 1 = x'$

Inversion theorems

T5. $(x \oplus y)' = x' \oplus y = x \oplus y'$

T6. $x' \oplus y' = x \oplus y$

T7. $x \oplus y = y \oplus x$	**Commutative law**
T8. $(x \oplus y) \oplus z = x \oplus (y \oplus z)$	**Associative law**
T9. $x(y \oplus z) = xy \oplus xz$	**Distributive law**
T9'. $x(y \oplus z) = (x'+y) \oplus (x'+z)$	**Distributive law with OR function**
T10. If: $f = g \oplus h$ and $gh = 0$, then $f = g + h$	**Disjunction theorem**
T11. If: $f = g \oplus h$, then $g = f \oplus h$ and $h = g \oplus f$	**Transposition theorem**

XOR general properties

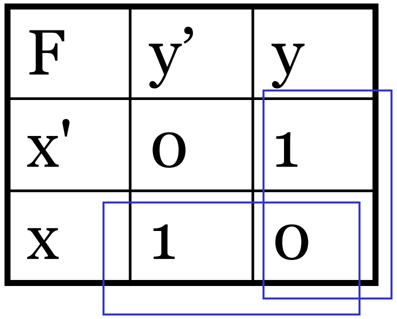
- Multiple-variable XOR operation is defined as an *odd function*
  - Function equal 1 if odd number of variables equal 1
- *Even function*
  - Even number of variables is equal to 1
  - Complement of odd function

A	B	C	F
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

3 variable XOR

XOR function from K-map

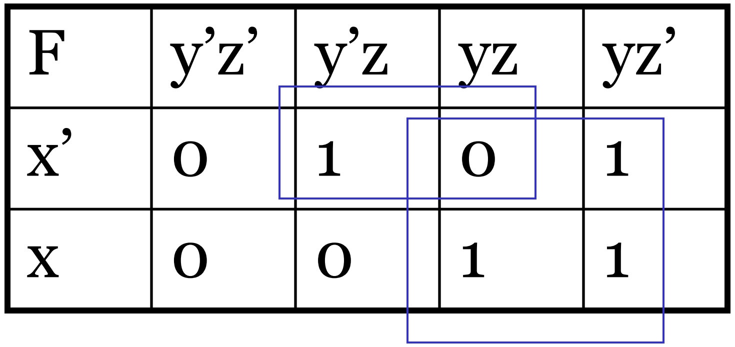
- In view of XOR properties, the following rules enable determining of XOR function from K-map:
  - there must be overlap of the groups
  - cells with true condition (1) must be encircled odd number of times (includes one time)
  - cells with false condition (0) must be encircled even number of times (includes zero time)



F	y’	y
x'	0	1
x	1	0

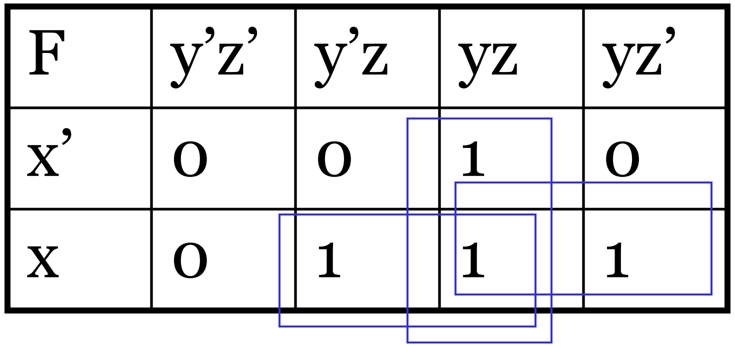
F = x ⊕ y

XOR function examples - 1



F	y'z'	y'z	yz	yz'
x'	0	1	0	1
x	0	0	1	1

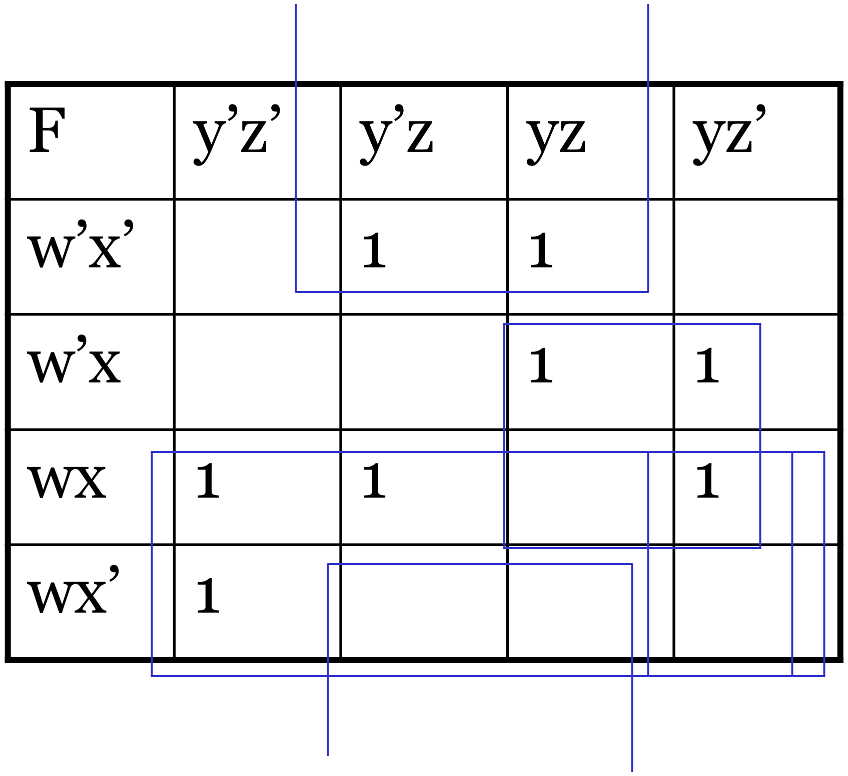
F = y ⊕ x'z



F	y'z'	y'z	yz	yz'
x'	0	0	1	0
x	0	1	1	1

F = xz ⊕ xy ⊕ yz

XOR function examples - 2

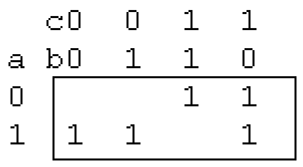


F	y'z'	y'z	yz	yz'
w'x'		1	1	
w'x			1	1
wx	1	1		1
wx'	1			

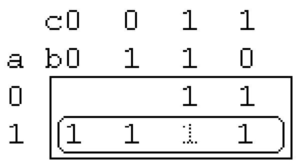
F = w ⊕ x'z ⊕ xy ⊕ wyz'

XOR function illustrations

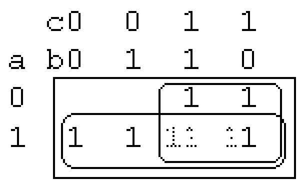
- Start by plotting (encircle) variable **a** as shown in (b)
  - this covers three of the 1s in the map but places an additional 1 (in grey) at position **abc**
- Next plot variable **c** as shown in (c)
  - this cancels the extra 1 at **abc**, covers the 1s at position **a'bc** and **a'b'c** but cancels the 1 at position **ab'c**
- To regain a 1 at this position we place an additional 1 there and map that position as shown in (d)



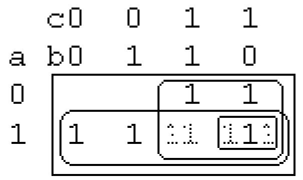
(a)



(b)



(c)



(d)

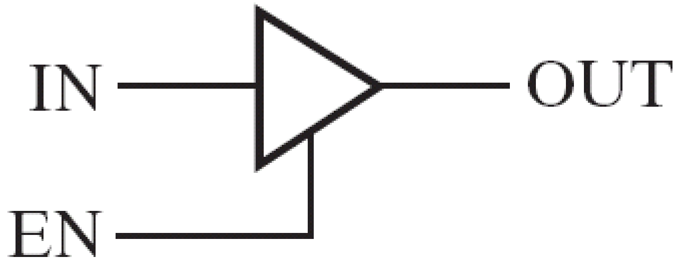
$y = a \oplus c \oplus ab'c$

High-Impedance Output

- Gates may produce a third output value known as high-impedance state, Hi-Z, Z or z
- Hi-Z behaves as an open circuit, thus output appears to be disconnected
  - allows the output of a logic circuit to be disconnected from the main circuit
- Gates with Hi-Z output can have their outputs connected together
  - Provided that no 2 gates drive the line at the same time to opposite 0 and 1 values

Three-State Buffers - 1

- Tri-state buffer is distinguished from normal buffer by the *enable* input

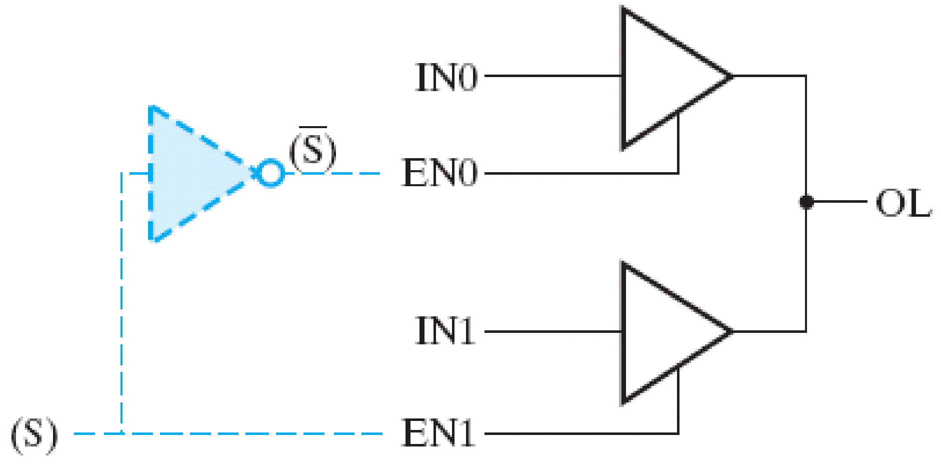


(a) Logic symbol

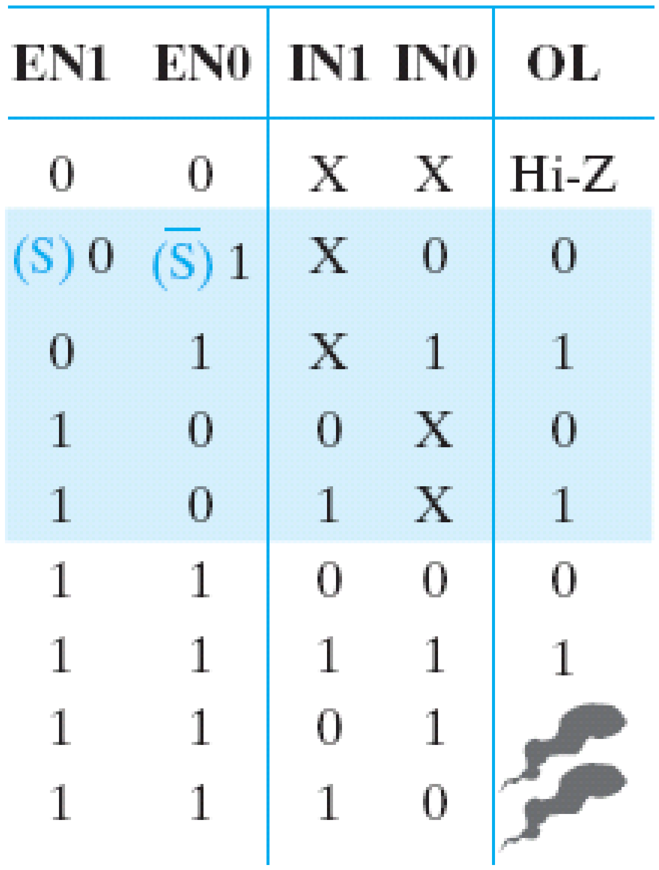
EN	IN	OUT
0	X	Hi-Z
1	0	0
1	1	1

(b) Truth table

Three-State Buffers - 2



(a) Logic Diagram



EN1	EN0	IN1	IN0	OL
0	0	X	X	Hi-Z
(S) 0	$(\overline{S})$ 1	X	0	0
0	1	X	1	1
1	0	0	X	0
1	0	1	X	1
1	1	0	0	0
1	1	1	1	1
1	1	0	1	
1	1	1	0	

(b) Truth table

Three-State Buffers - 3

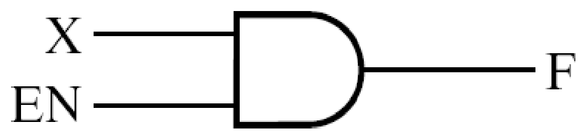
- If conflicting output appears at the connecting line
  - then the current is often large enough to cause heating and may even destroy the circuit
  - designer must ensure that EN0 and EN1 never equal 1 at the same time
    - e.g. by using a decoder to generate the EN signals

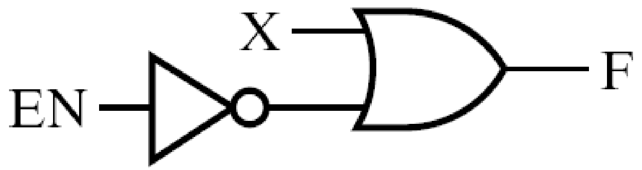
Enabling - 1

- In general, enabling permits an input signal to pass through to (affect) an output
  - Enable (EN) input signal is required to determine whether the output is enabled
    - Enable the operation of the logic circuit
- In addition to replacing the input signal with the Hi-Z (high impedance) state
  - Disabling also can replace the input signal with a fixed output value of 0 or 1

Enabling - 2

- Disabled value 0
  - If EN = 1
    - Input X reaches output
  - If EN = 0
    - Output always 0
- Disabled value 1
  - If EN = 1
    - Input X reaches output
  - If EN = 0
    - Output always 1





Summary

- Five steps in logic circuit design
- Three techniques for minimization (simplification) with K-map being simplest while Quine-McCluskey method being systematic
- Variations in implementation incluing AND-OR, NAND-NAND and NOR-NOR being most common
- XOR offers alternative to SOP and POS in simplifying functions with scattered ones
- Enable and Hi-Z facilitate circuit interconnections