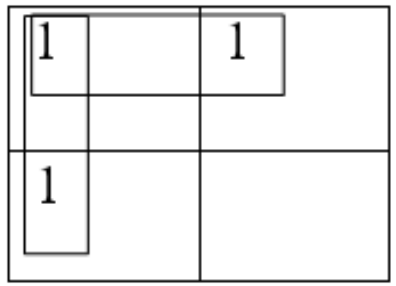
Unit-3
Simplification of Boolean Functions

Karnaugh Map (K-Map) method for simplification of logic expression: K- map is a simple and straight forward graphical technique for simplifying Boolean functions. It is a diagram consisting of squares. It is pictorial form of truth table. Each square of map represents a minterm and logic expression can be written as SOP, i.e. sum of minterms.

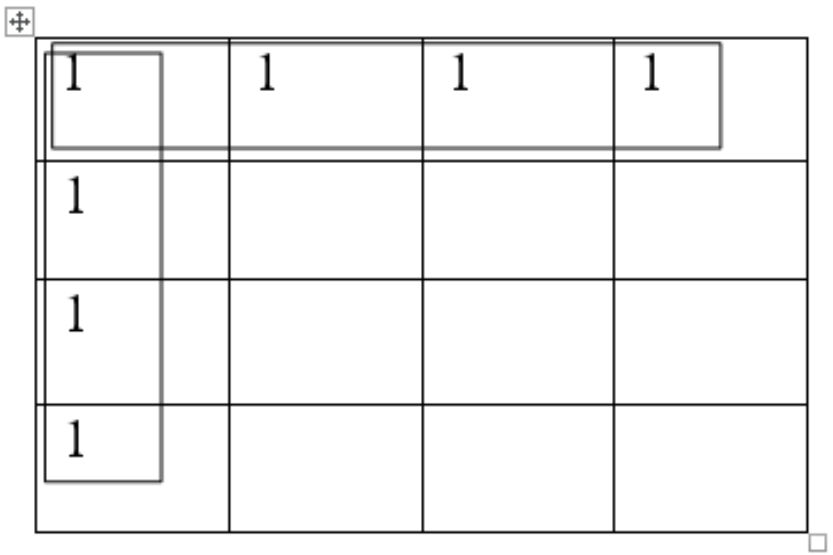
K-Map for n variables is made up of 2^n squares. We can write either 1s or 0s in the K-Map square considering grouping of 1s gives the minimal SOP and grouping of 0s gives minimal POS.

Depending on the number of 1s and 0s available for grouping, the types of groups are:

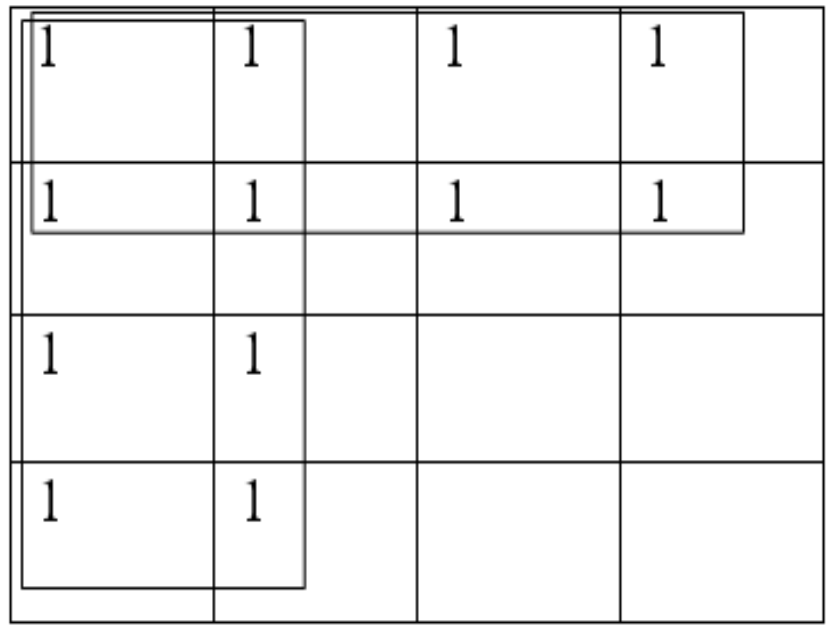
1. **Pair:** Grouping of two 1s or two 0s is called pair. It consists of 1 Row x 2 Columns and 2 Rows x 1Column



2. **Quad:** Grouping of four 1s or four 0s. It consists of 1 Row x 4 Columns and 4 Rows x 1 Column.



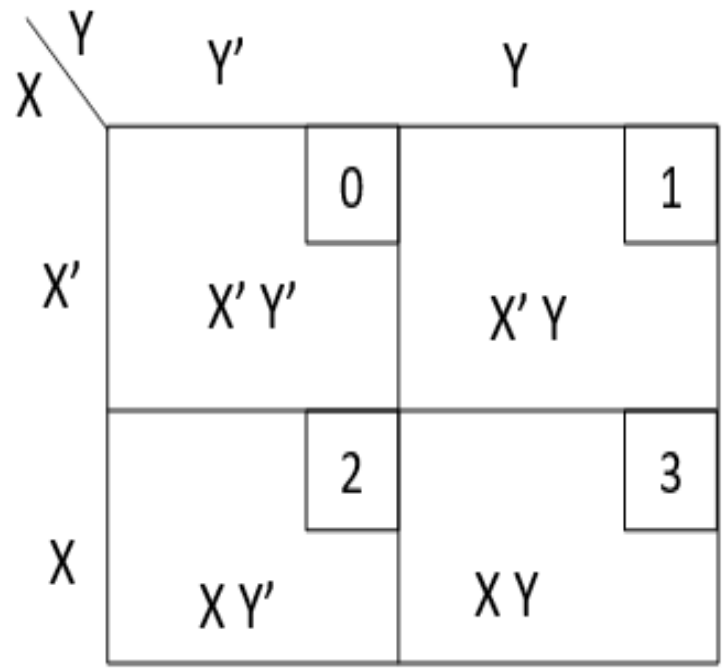
3. **Octet**: Grouping of eight 1s or eight 0s. It consists of 2 Rows x 4 Columns and 2 columns x 4 rows.



Rules for making groups in SOP:

1) No zeros are allowed in grouping.
2) No diagonals are allowed in grouping.
3) Groups should be as larger as possible.
4) Overlapping is allowed.

Two Variable K-Map: There are four minterms in two variable K-map. ($2^n = 2^2 = 4$). Hence map consists of 4 squares one for each minterm.

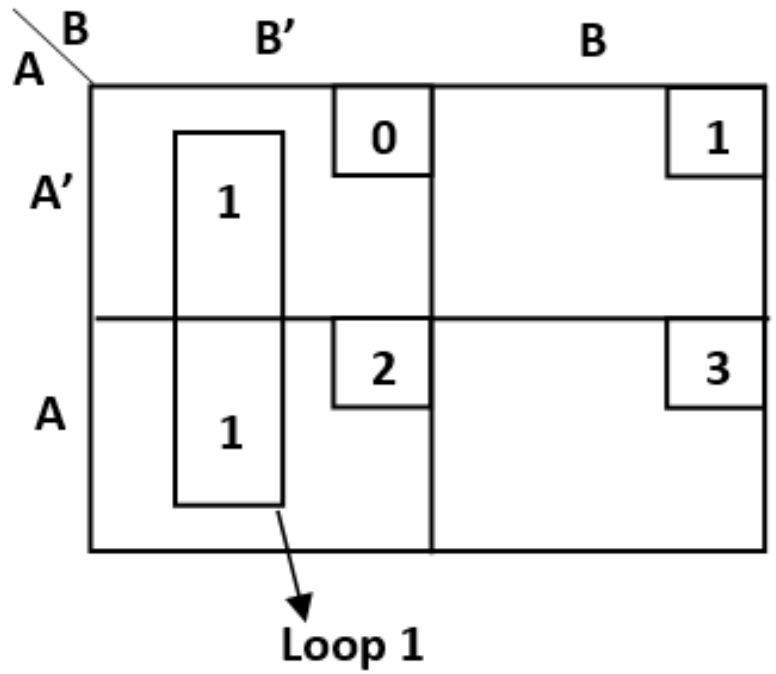


Example: Simplify: Y = A' B' + A B' using K-map.

Solution;

Here,

A'B' + AB' =B'(A' +A)



Therefore; F = B'

Example: Simplify: $F(A,B) = \sum(0,1,3)$ using K-map.

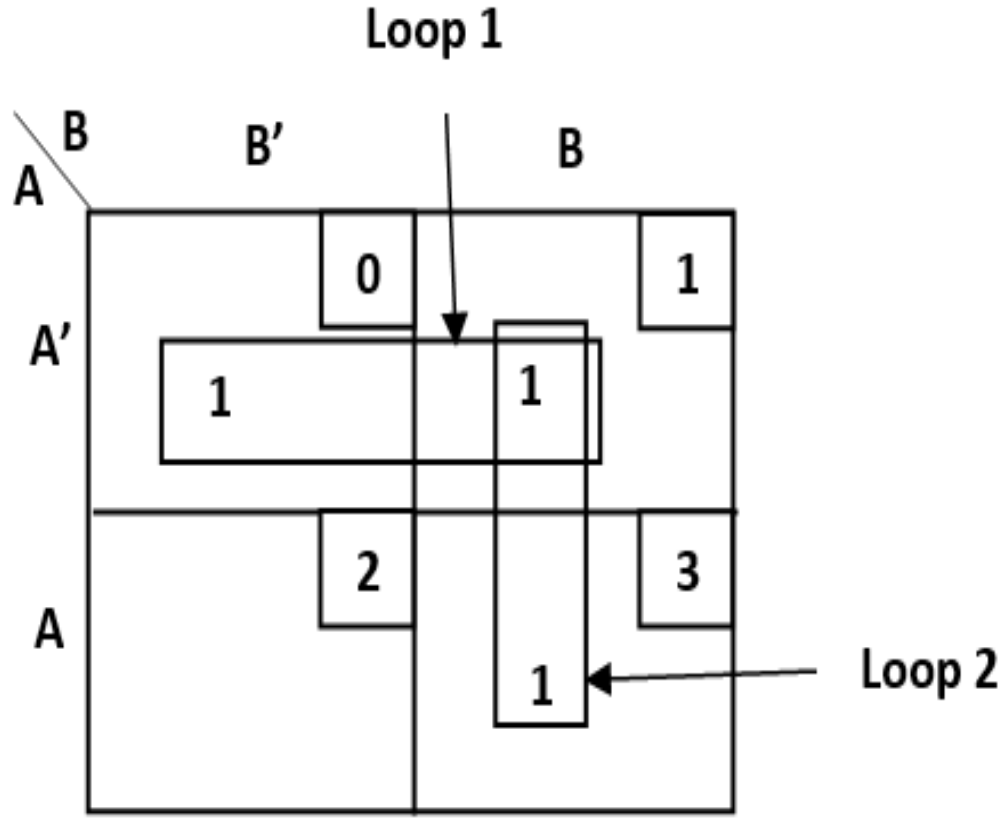
Solution;

Here,

0 = 0 0 = A' B'

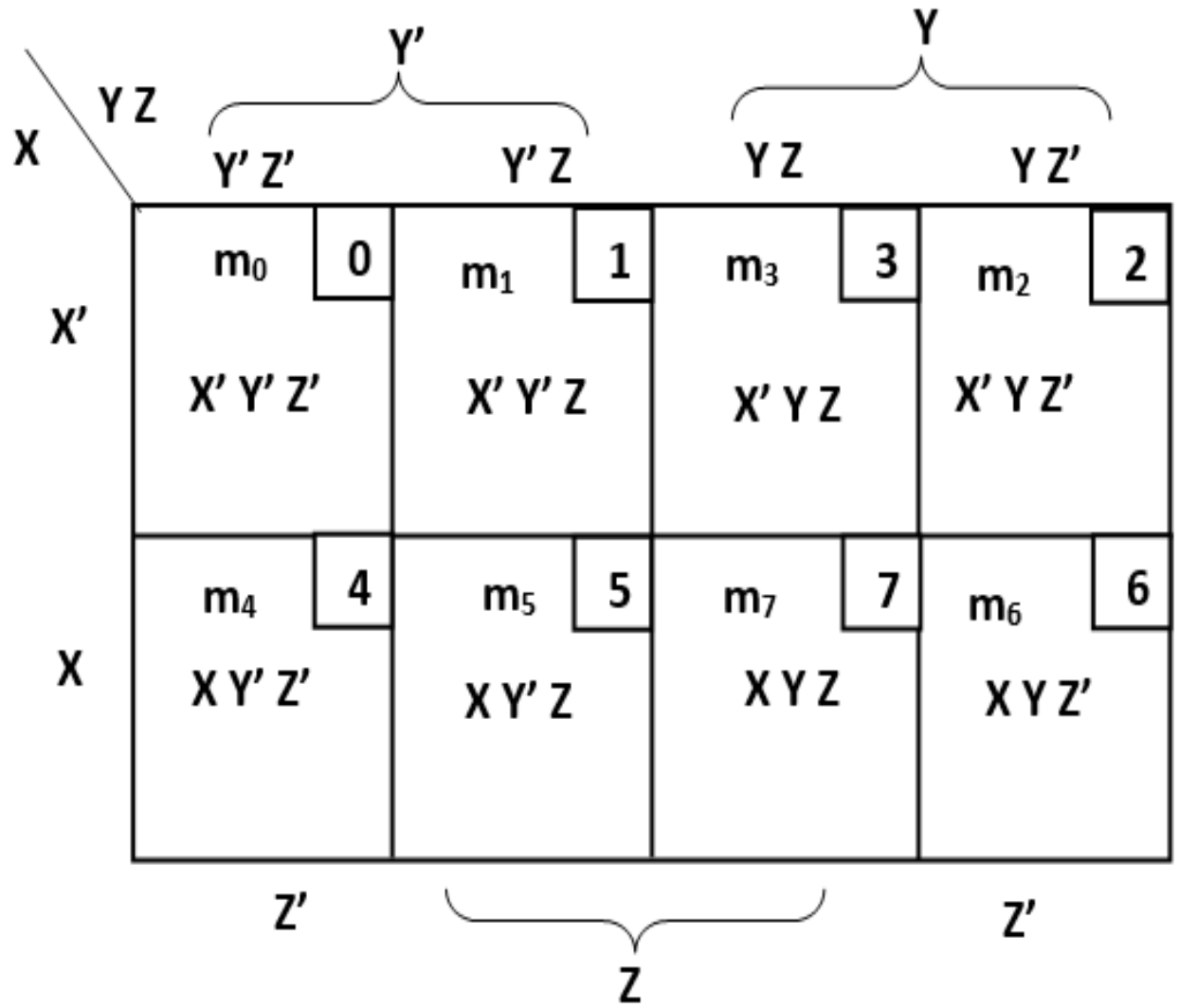
1 = 0 1 = A' B

3 = 1 1 = A B



Therefore; F = Loop1 + Loop2 = A' + B

Three Variable K-Map: there are eight minterms for three variable K-Map (i.e. $2^n = 2^3 = 8$). Hence map consists of eight squares for each minterm.



Simplify: F= X'Y'Z +X'YZ' + XY'Z'+XY'Z using k-map.

Solution;

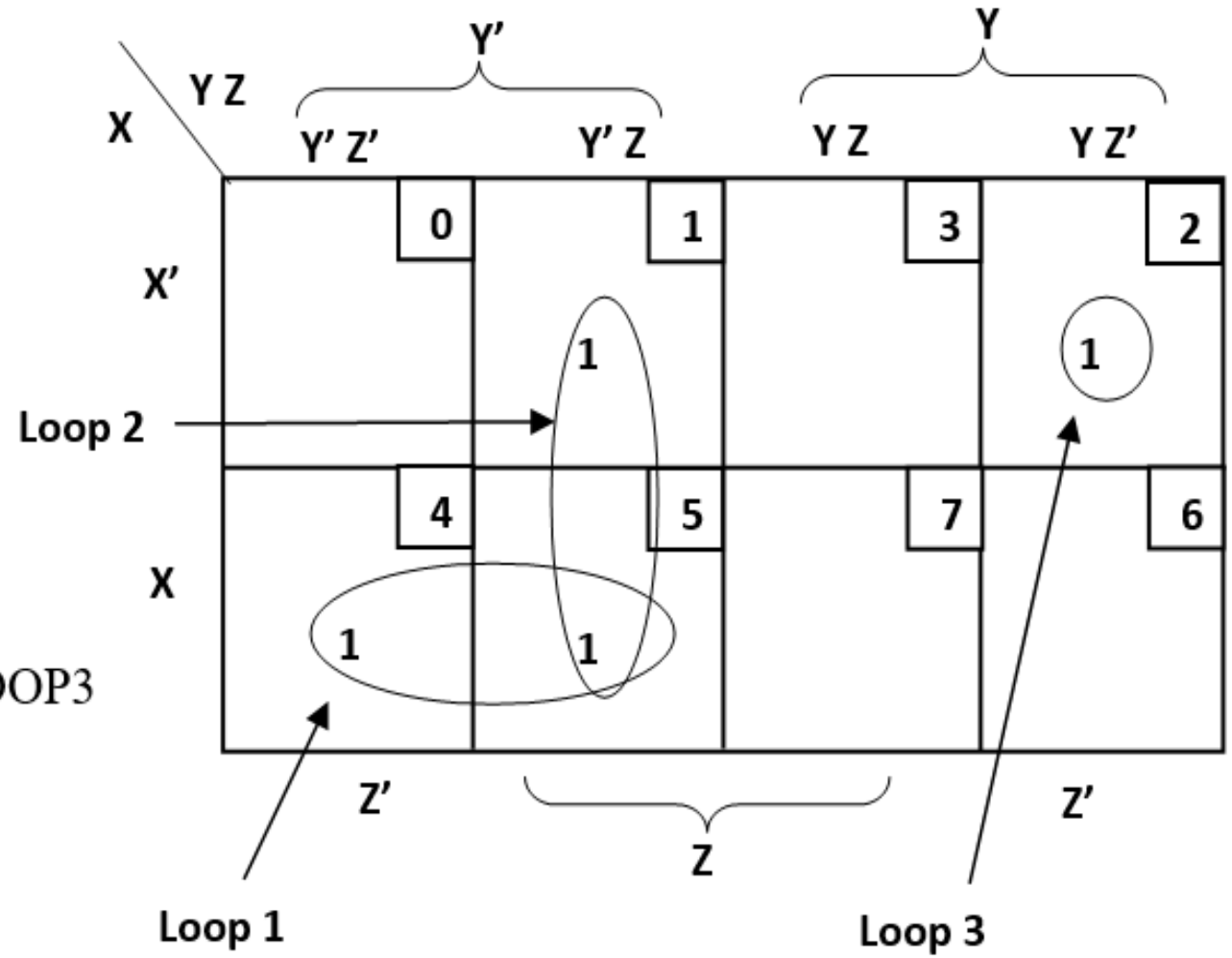
Here,

X' Y' Z = 0 0 1 = 1

X' Y Z' = 0 1 0 = 2

X Y' Z' = 1 0 0 = 4

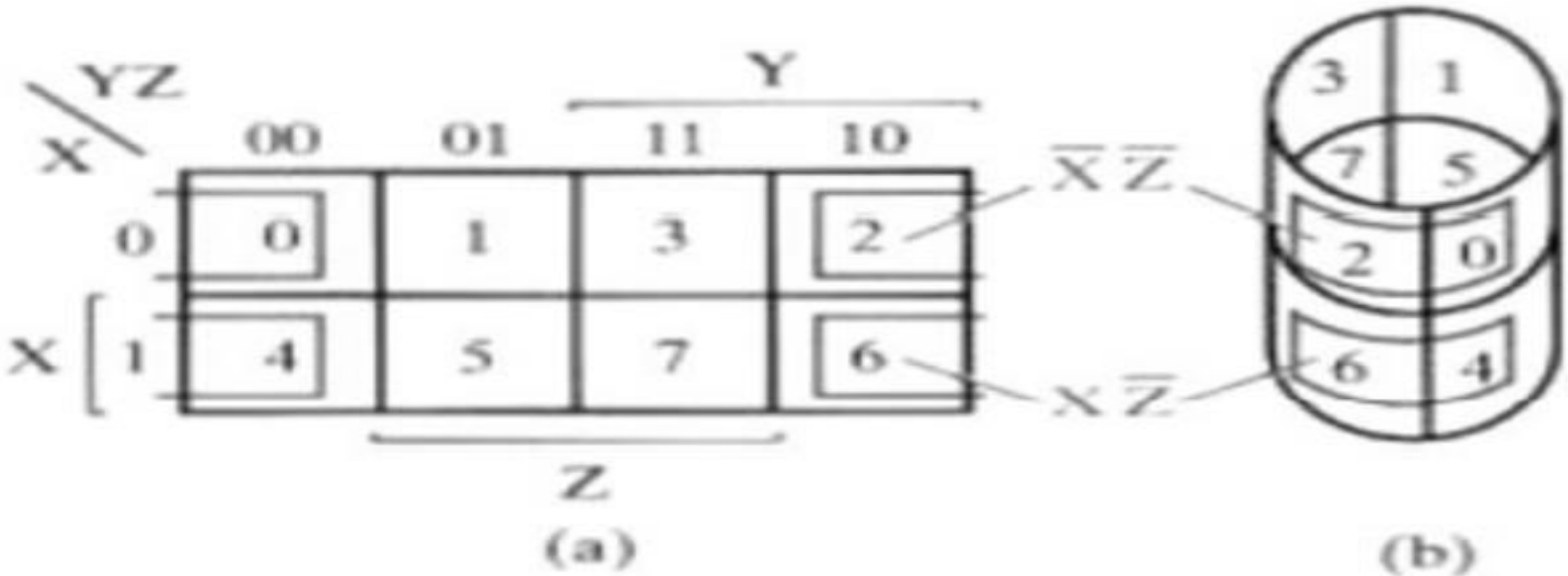
X Y' Z = 1 0 1 = 5



Therefore, F = LOOP1 + LOOP2 + LOOP3

= X Y' + Y' Z + X' Y Z'

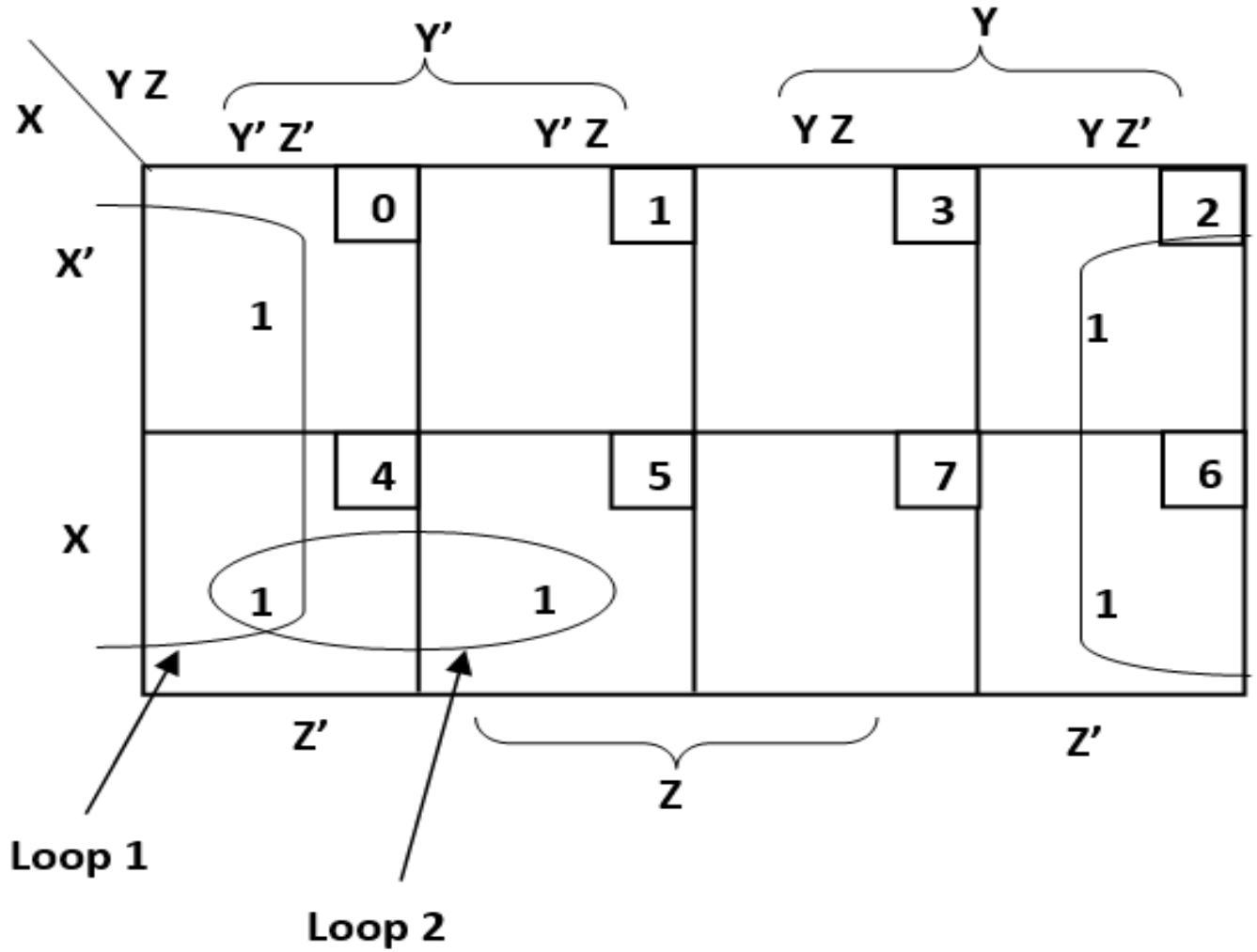
Adjacent loop:



Minterm adjacencies are circular in nature. This figure shows Three-Variable Map in Flat and on a Cylinder to show adjacent squares.

Example: Simplify: $F(X, Y, Z) = \sum(0, 2, 4, 5, 6)$

Taking adjacent loop of minterms 0, 2, 4 and 6



$F = Z' + X\,Y'$

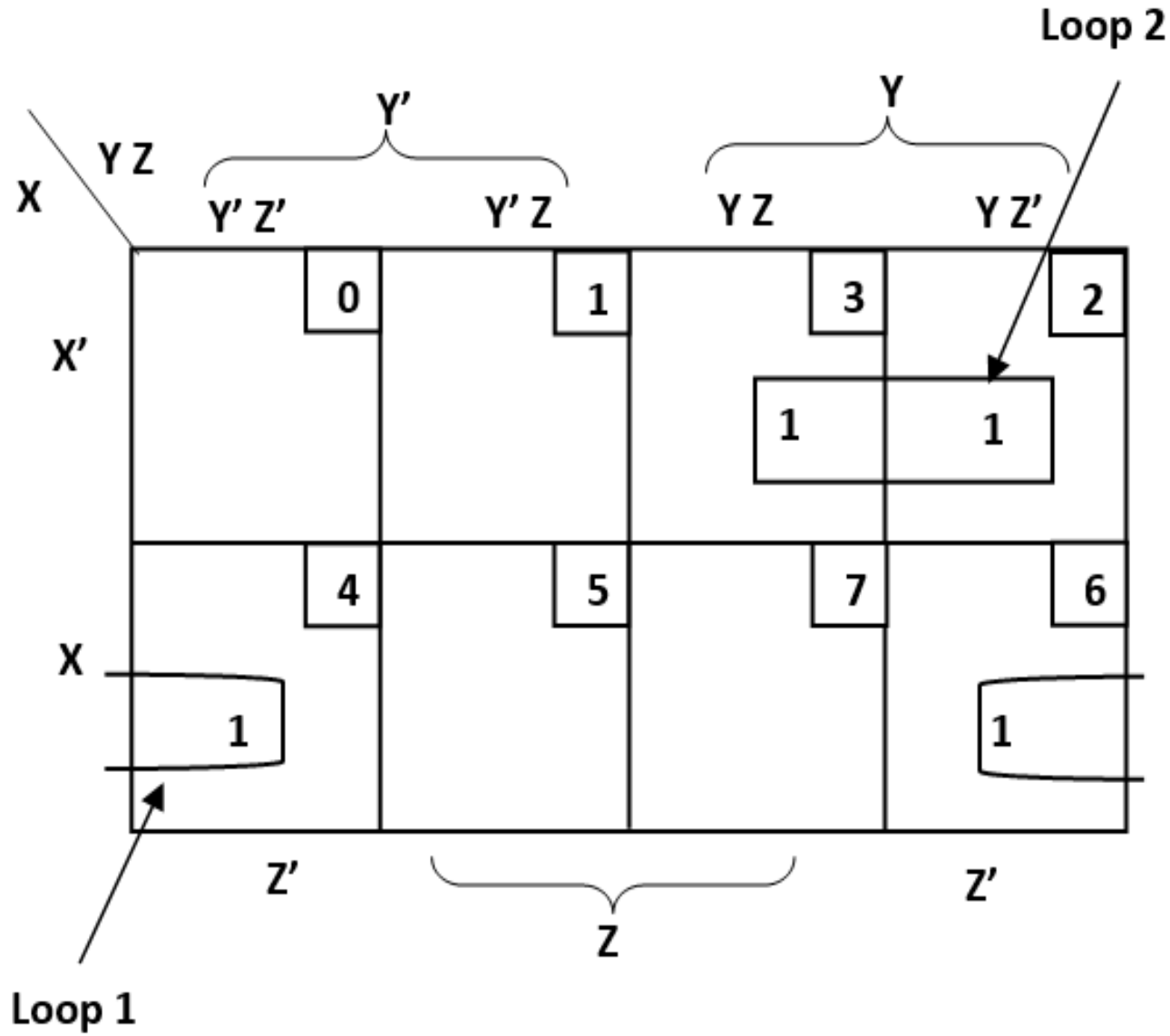
Example: Simplify: F = X Y Z' + X Z' + X' Y + X' Y Z'

$XZ'(Y+Y') = XYZ' + XY'Z'$

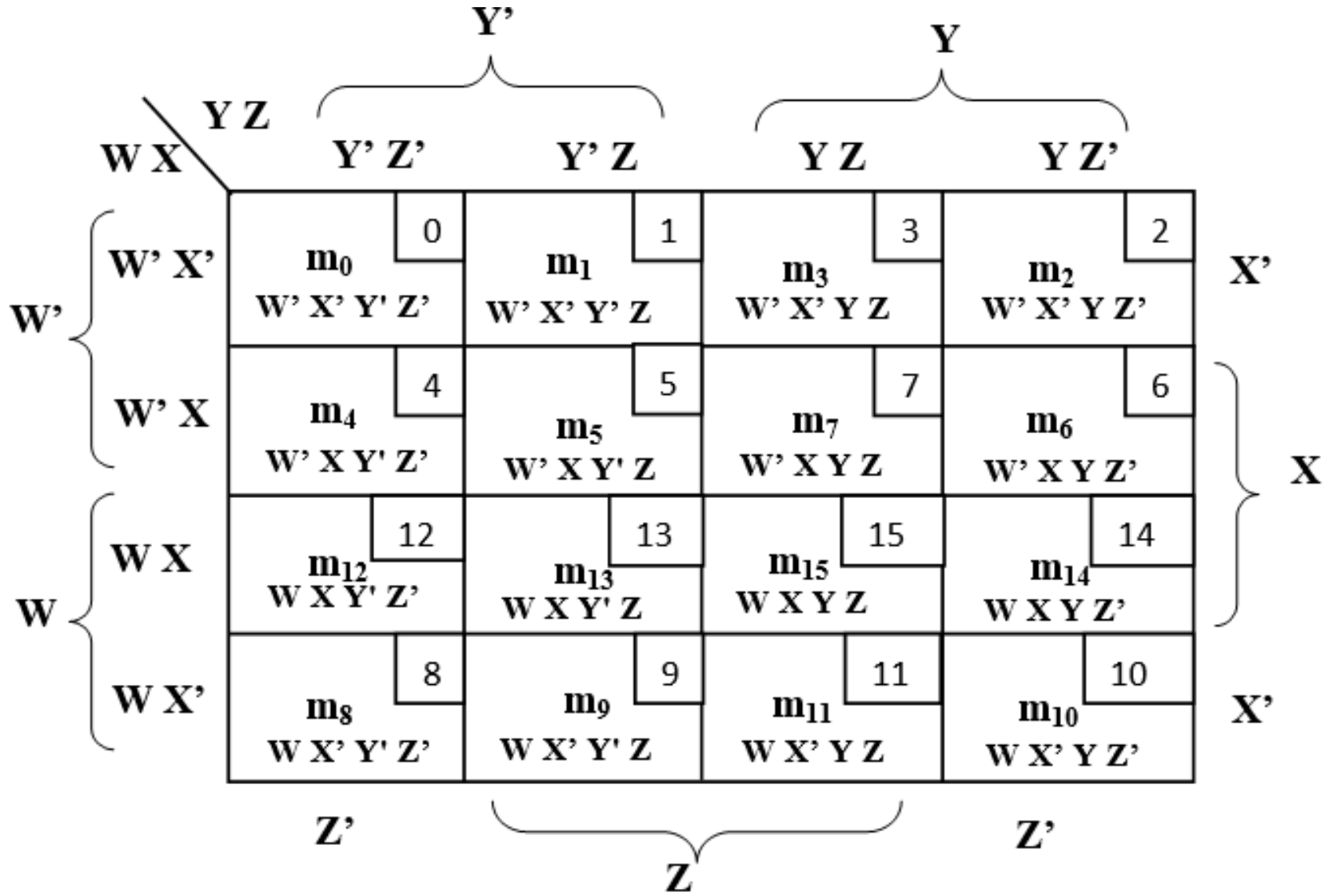
$X'Y(Z+Z') = X'YZ + X'YZ'$

$F = Loop1 + Loop2$

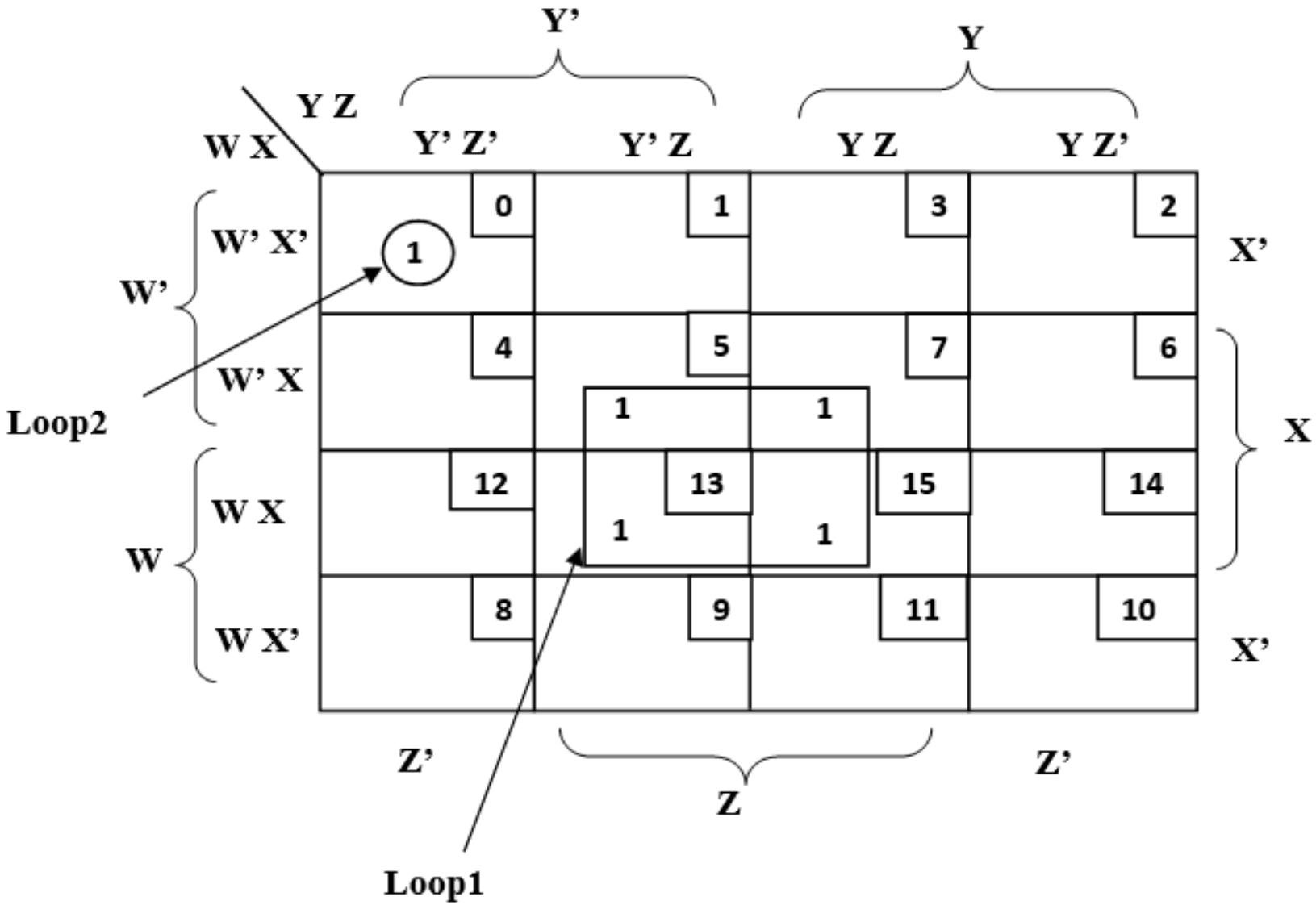
$= X Z' + X' Y$



Four Variable K-Map: There are 16 minterms for four variable k-Map (i.e. $2^n = 2^4 = 16$). Hence map consists of 16 squares for each minterm.



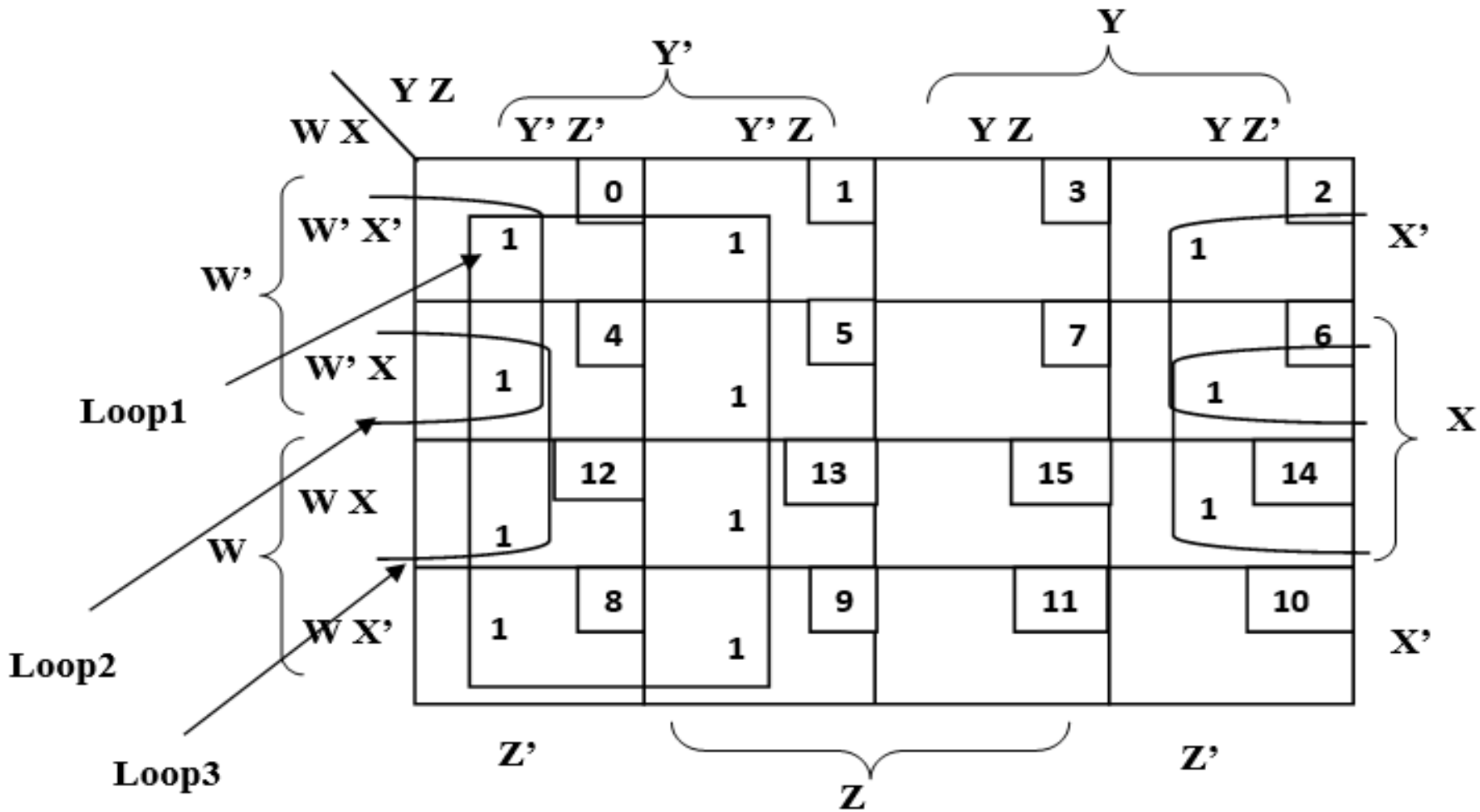
Example: Simplify: F= W'XY'Z + WXY'Z + W'XYZ + WXYZ + W'X'Y'Z'



F = Loop1 + Loop2

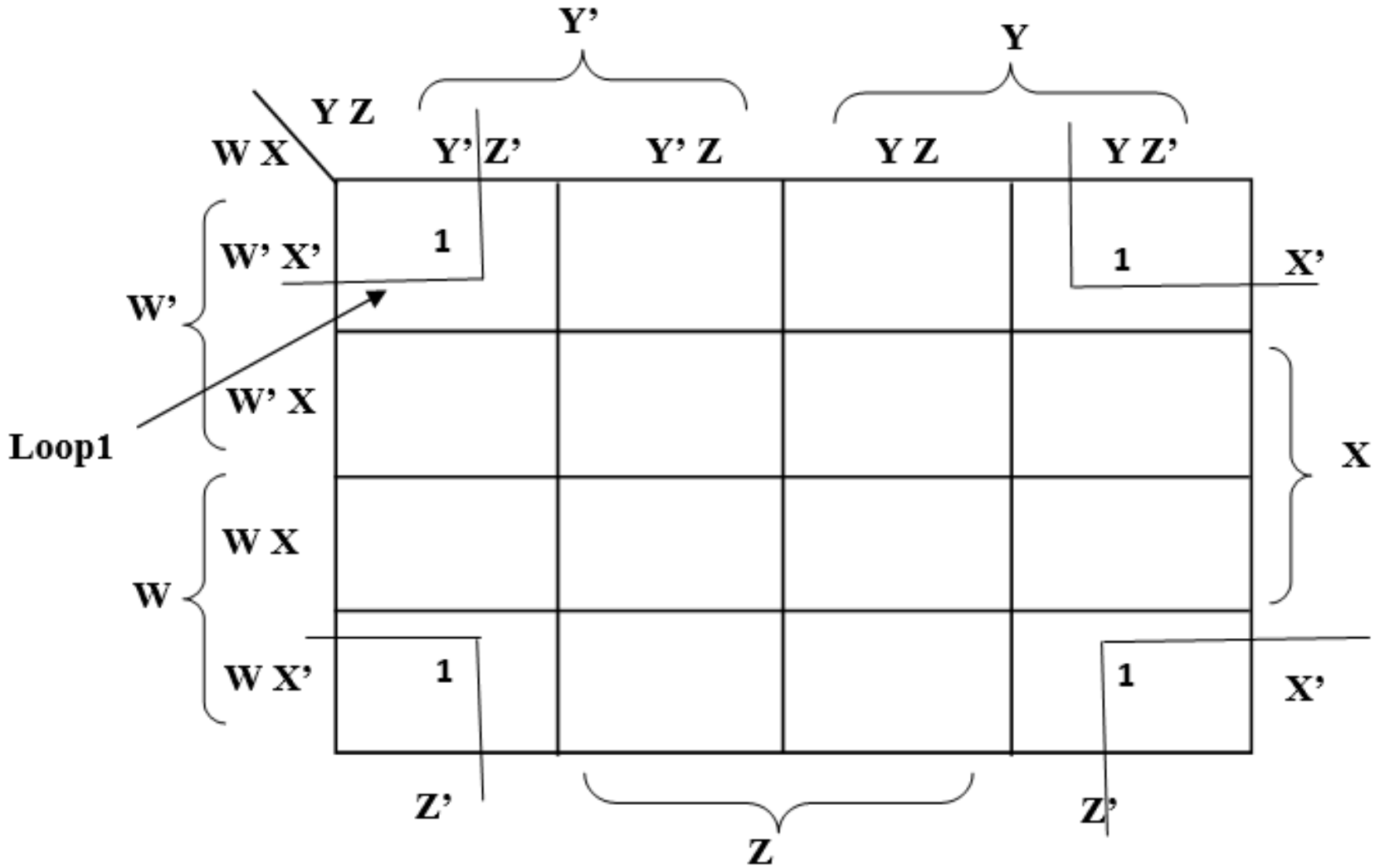
= X Z + W' X' Y' Z'

Example: Simplify: $F(W, X, Y, Z) = \sum(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$



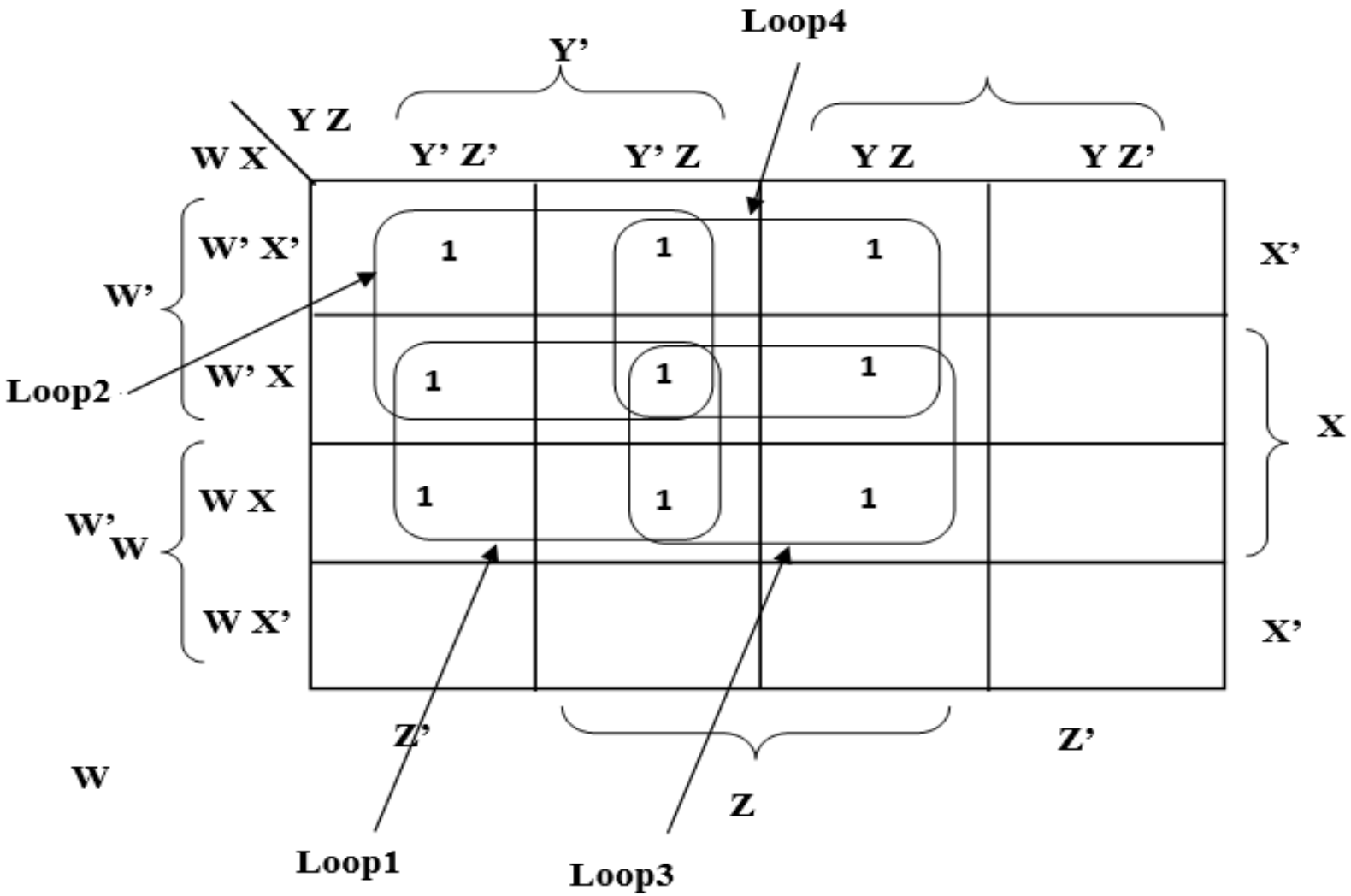
$F = Y' + W' Z' + X Z'$

Example: F = W'X'Y'Z' + W'X'YZ' + WX'Y'Z' + WX'YZ'



F= X' Z'

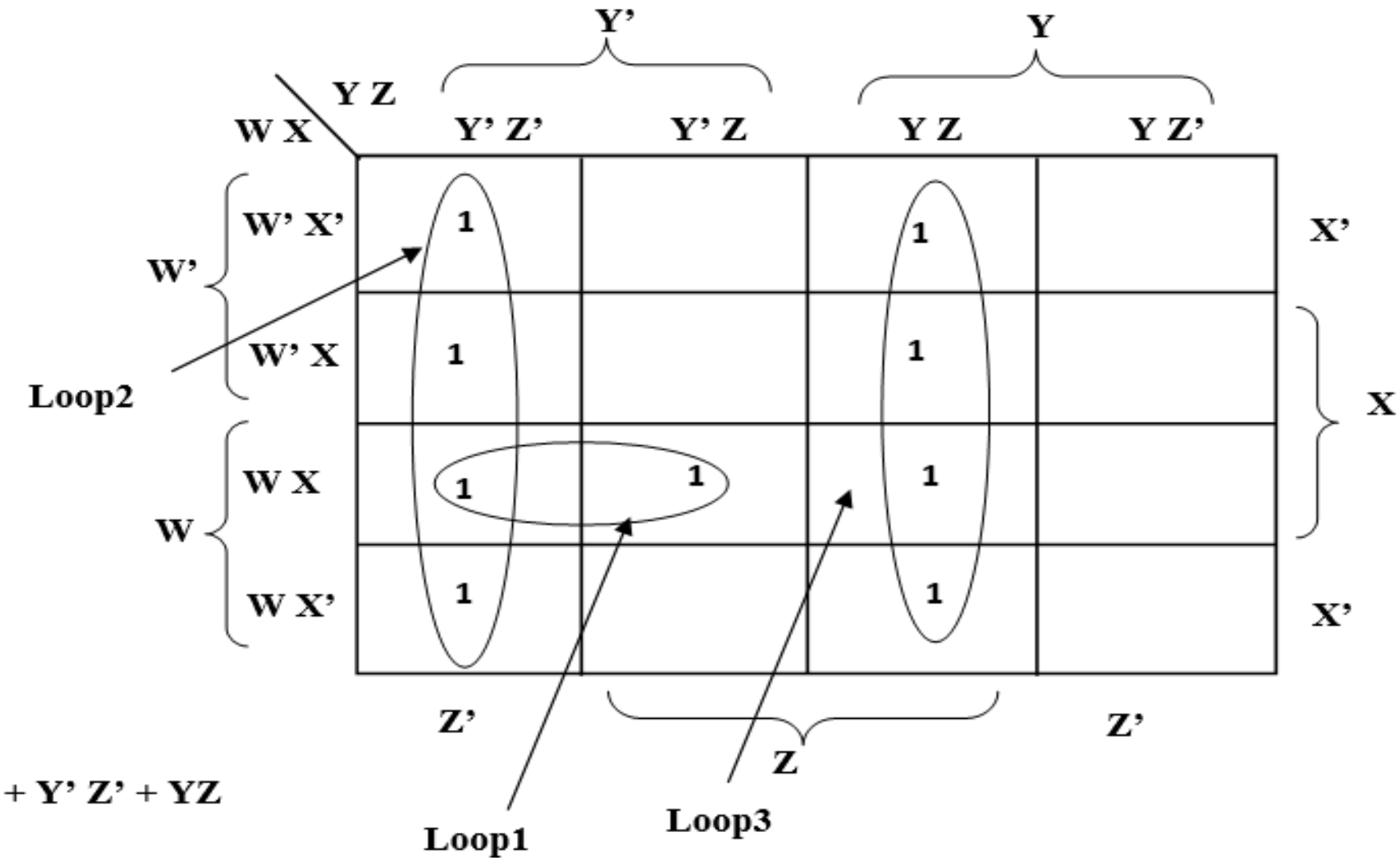
Example: F = W'X'Y'Z' + W'X'Y'Z + W'X'YZ + W'XY'Z' + W'XY'Z + W'XYZ + WXY'Z' + WXY'Z + WXYZ



F= XY' + W' Y' + XZ + W' Z

Example: W'X'Y'Z' + W'X'YZ + W'XY'Z' + W'XYZ + WXY'Z' + WXY'Z + WXYZ + WX'Y'Z'

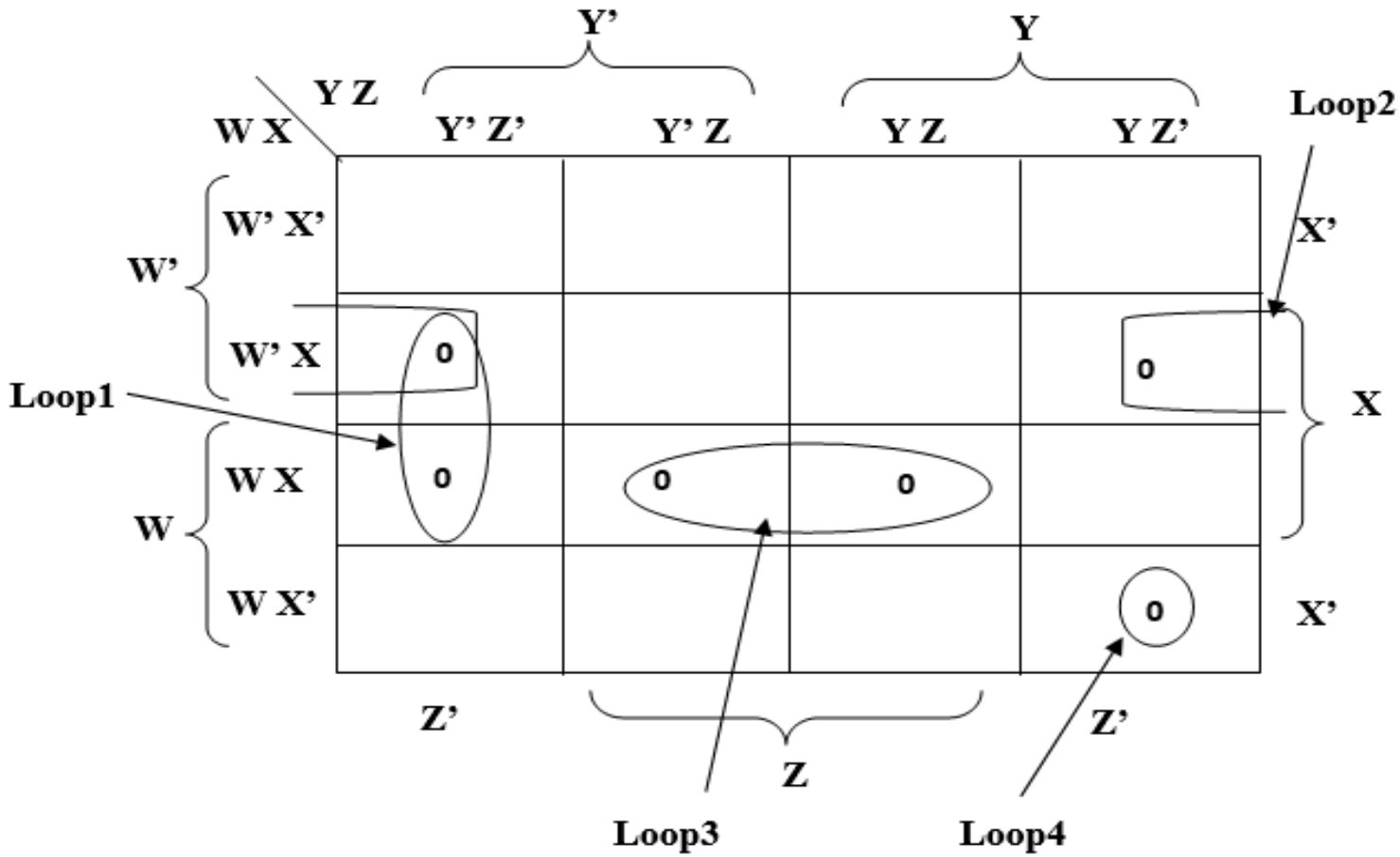
+ WX'YZ + WXY'Z



F = W X Y' + Y' Z' + YZ

Minimization of POS form using K-Map: In POS, 0 is included in grouping.

Example: Simplify: F(W, X, Y, Z) = $\pi(4, 6, 10, 12, 13, 15)$

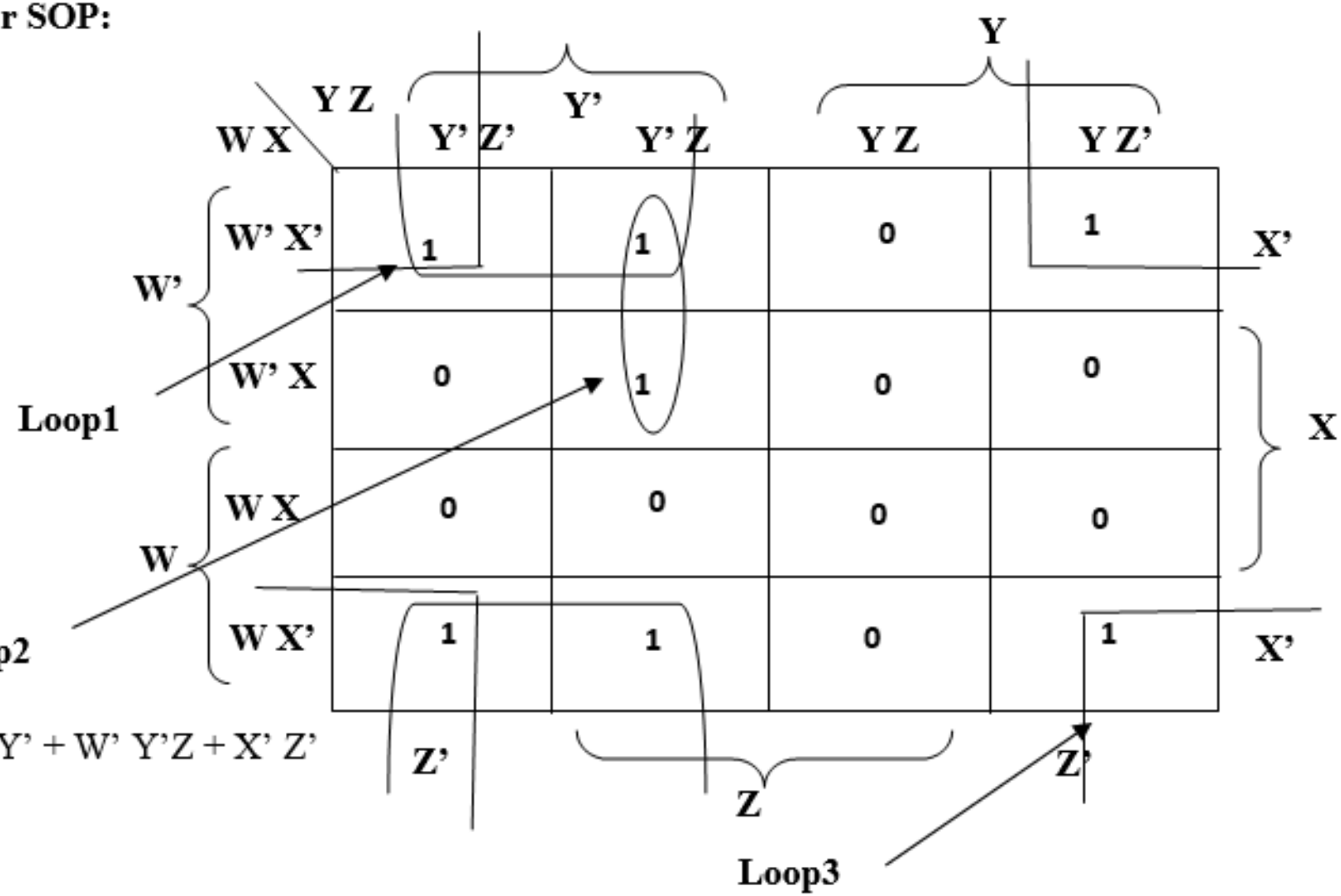


$F = (XY'Z' + W'XZ' + W X Z + WX'YZ')'$

$= (X' + Y + Z)\ (W + X' + Z)\ (W' + X' + Z')\ (W' + X + Y' + Z)$

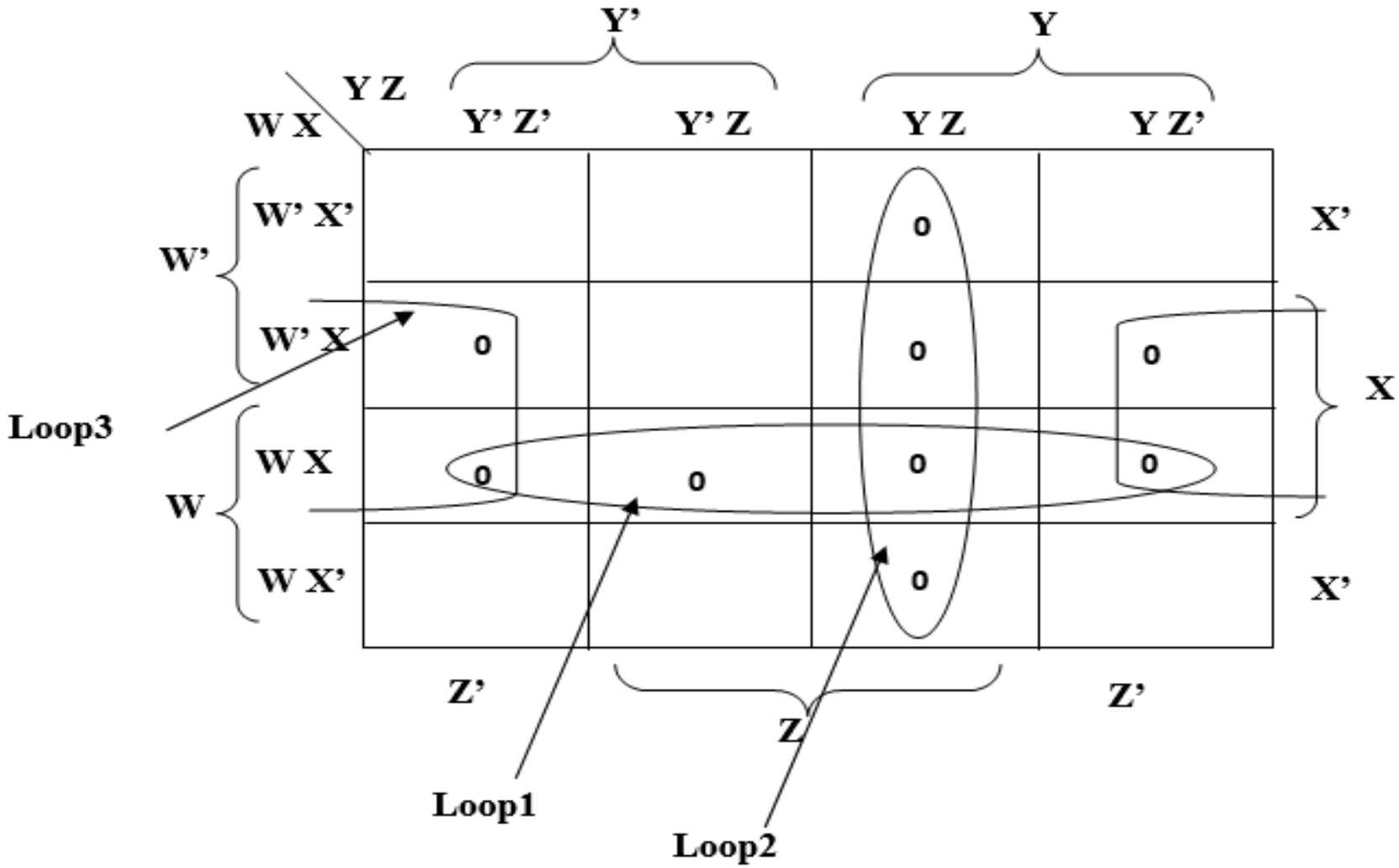
Example: Simplify: F(W, X, Y, Z) = $\sum m(0, 1, 2, 5, 8, 9, 10)$ *in both SOP and POS*

Map for SOP:



F = X' Y' + W' Y'Z + X' Z'

Map for POS:

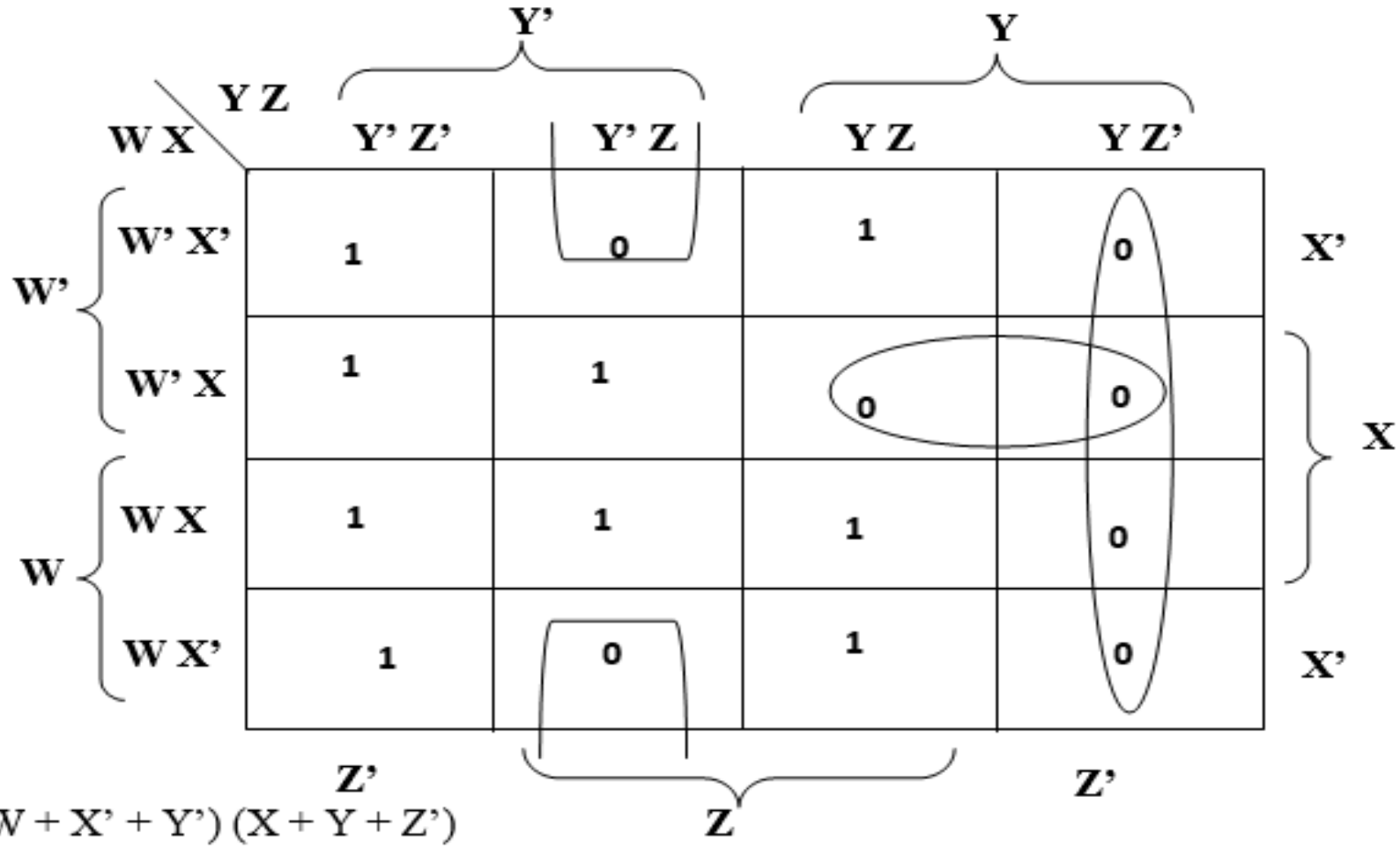


F = (W' + X') (Y' + Z') (X' + Z)

Simplify: F = (A + B + C + D') (A + B + C' + D) (A + B' + C' + D') (A + B' + C' + D)

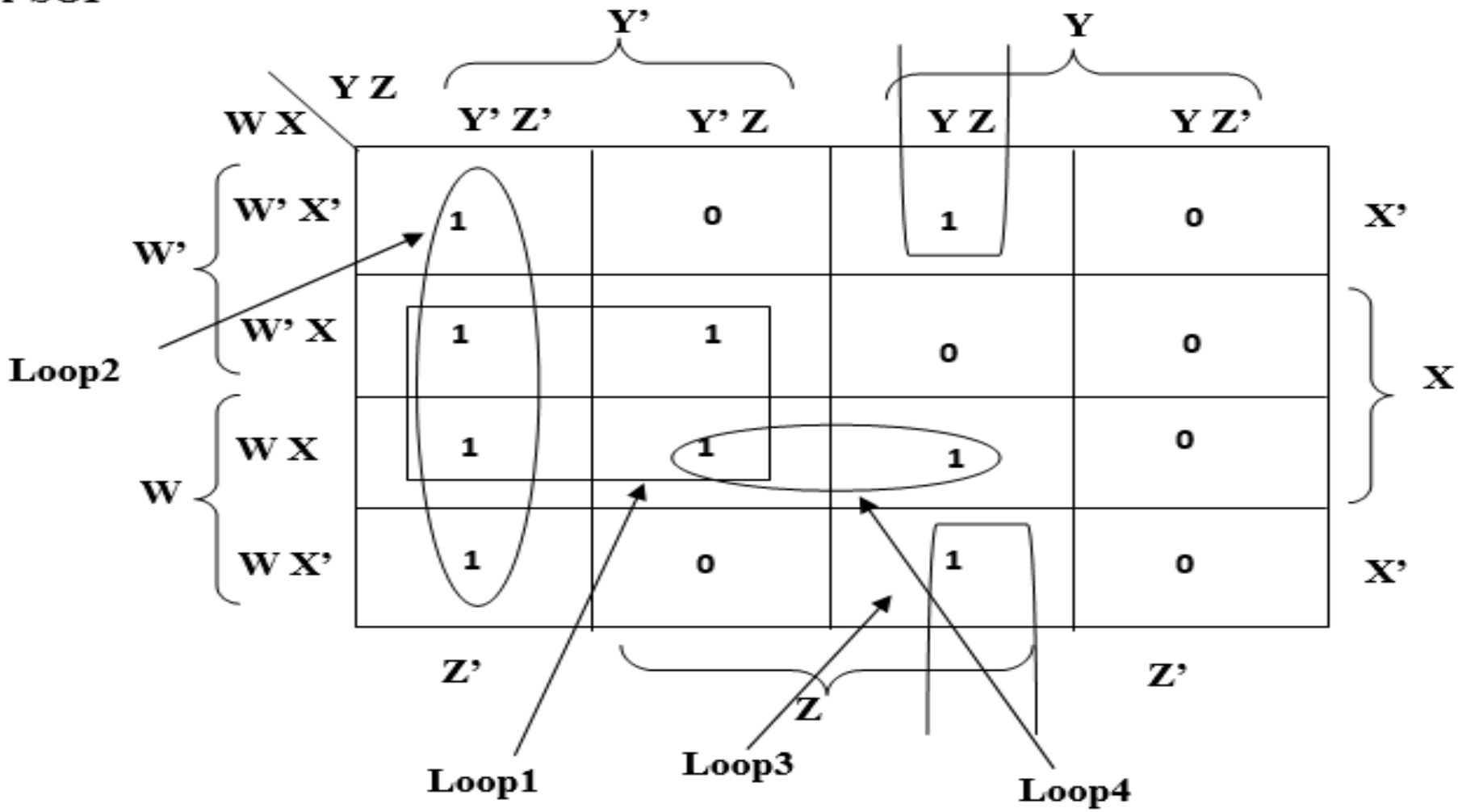
(A' + B' + C' + D) (A' + B + C + D') (A' + B + C' + D) in both SOP and POS.

Map for POS:



F = (Y' + Z) (W + X' + Y') (X + Y + Z')

Map for SOP



F = X Y' + Y' Z' + X' Y Z + W X Z

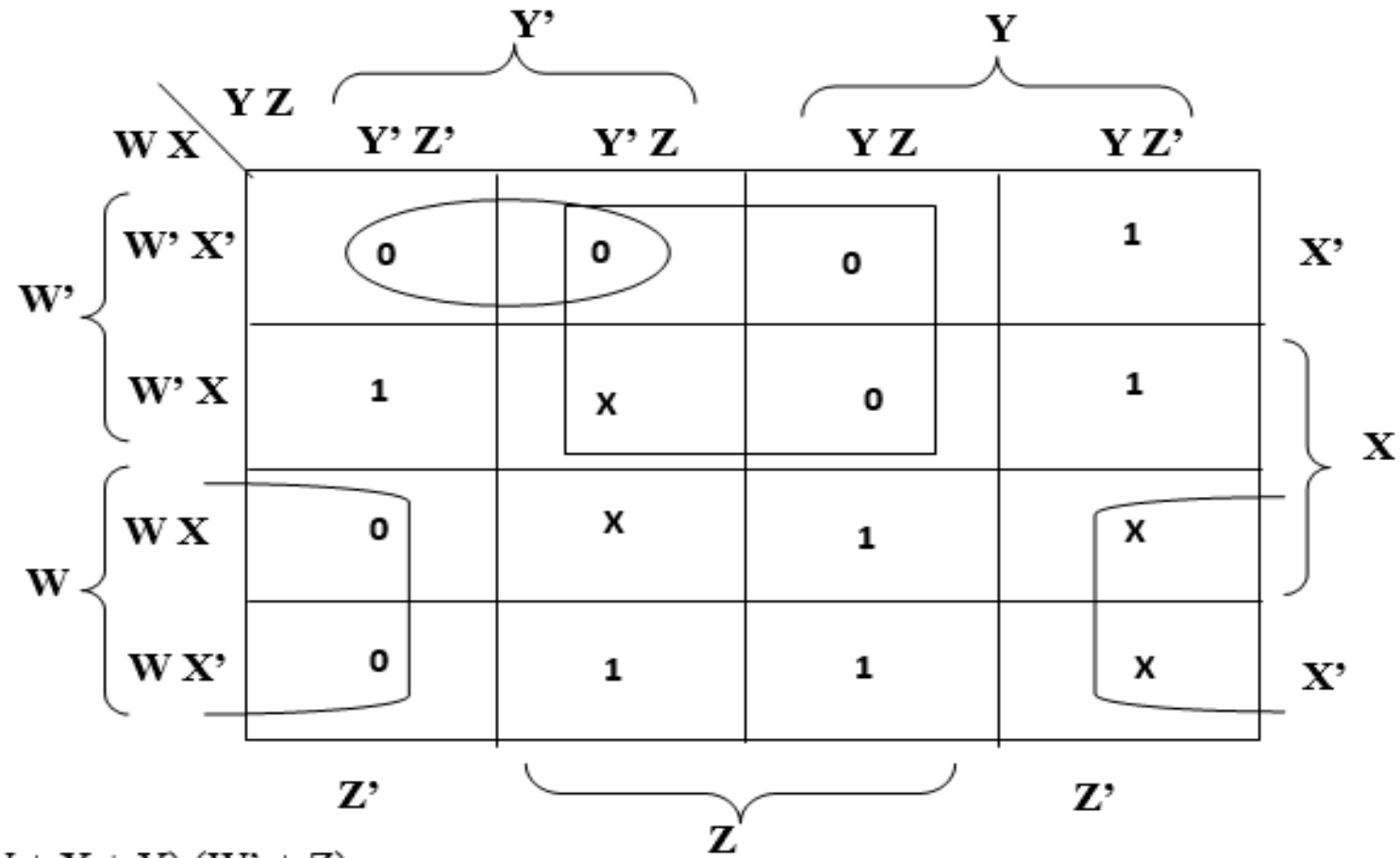
Don't Care Combination:

1. This is the combination which can be used as either with once or zeros.
2. Don't care combination is denoted by cross(x) sign. While making groups of either ones or zeros, any number of don't care can be taken.
3. There should not be any group exclusively with don't care combination.

Don't care symbol used with SOP is $\sum d$ and with POS is πd.

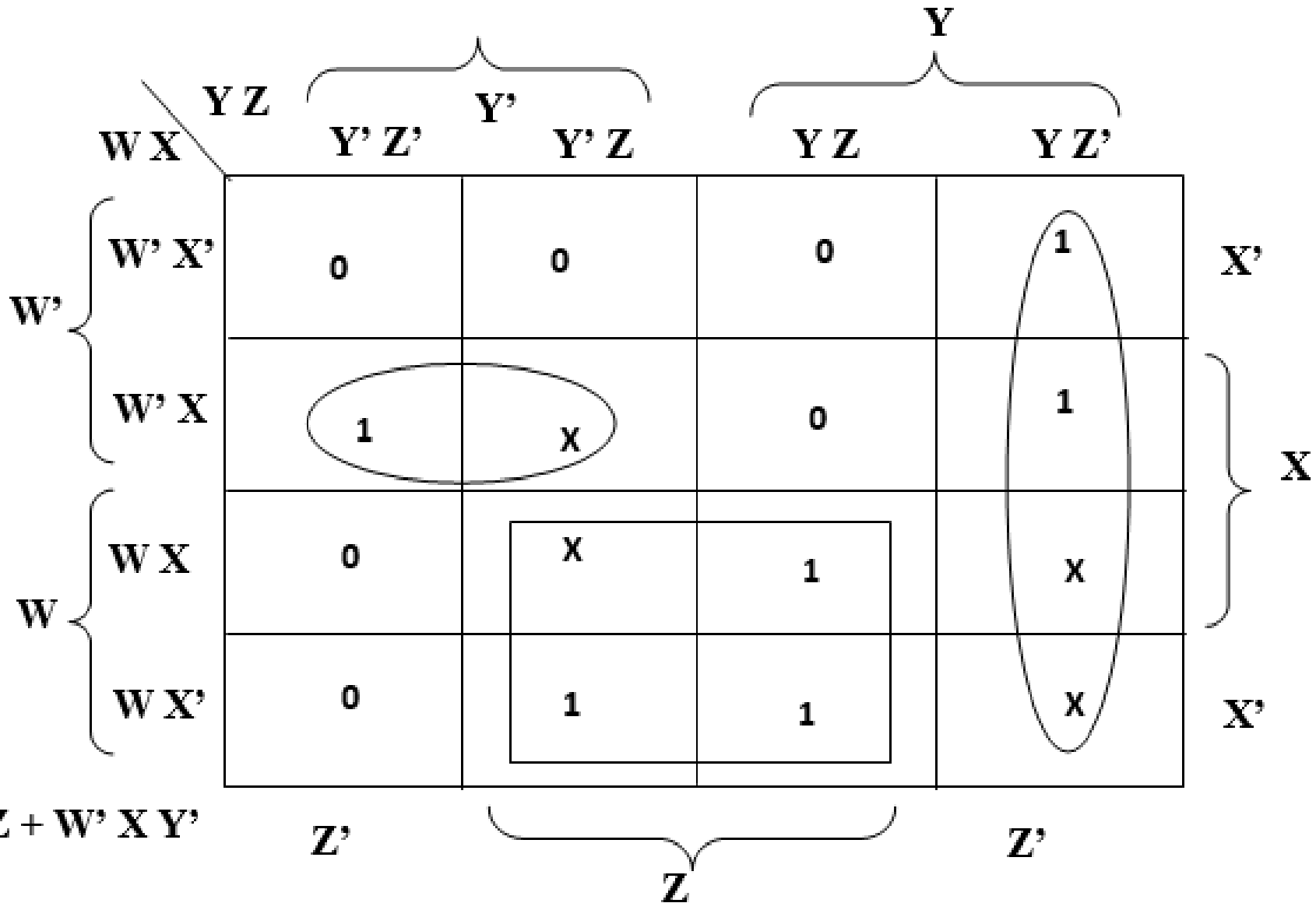
Example: Given F (W, X, Y, Z) = $\pi(0, 1, 3, 7, 8, 12) and\ \pi d(5, 10, 13, 14)$. Obtain minimal SOP and POS.

Map for POS:



F= (W + Z') (W + X + Y) (W' + Z)

Map for SOP:

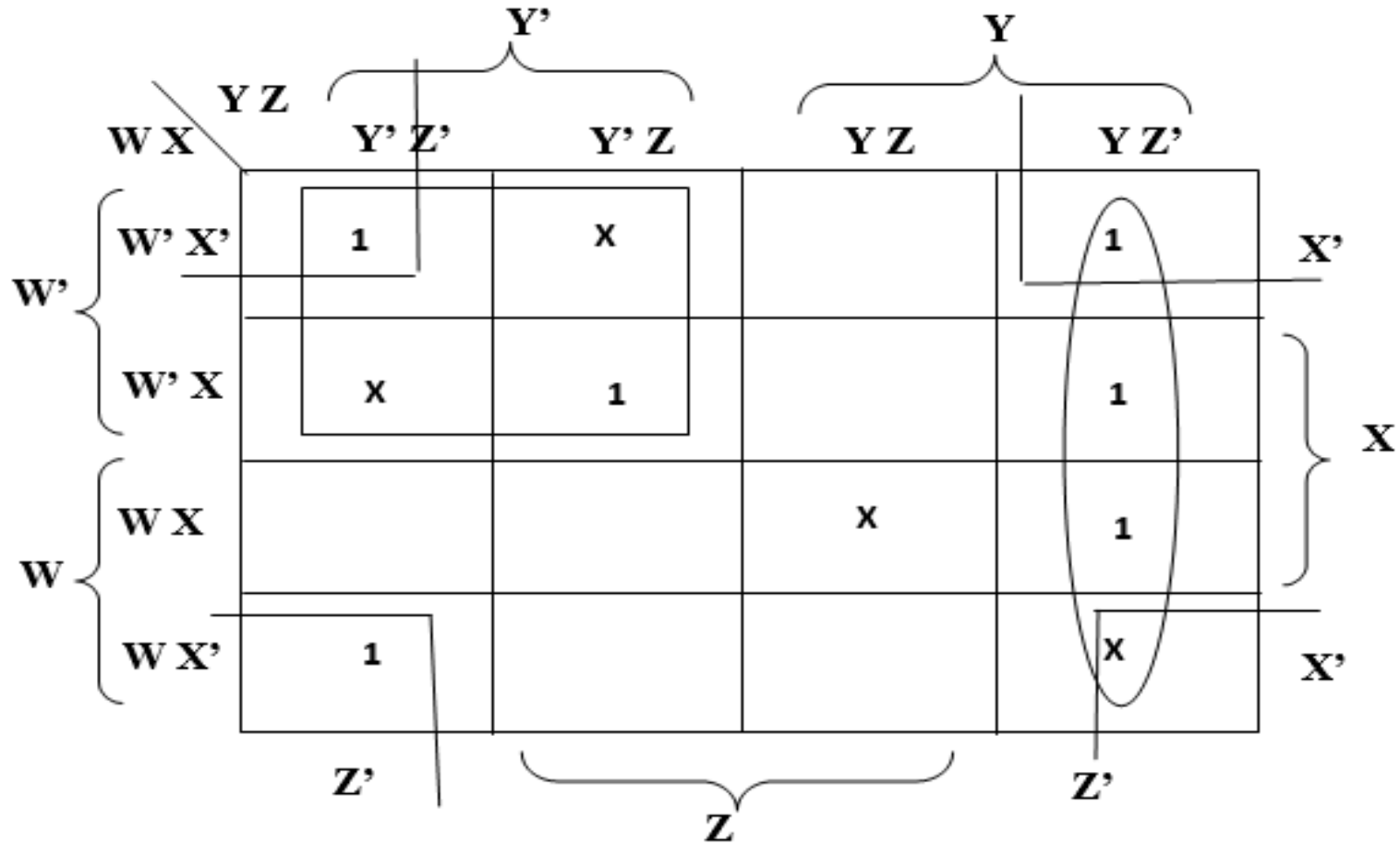


$F = Y Z' + W Z + W' X Y'$

Example: Draw a K-Map and simplify with the help of given truth table both for SOP and POS.

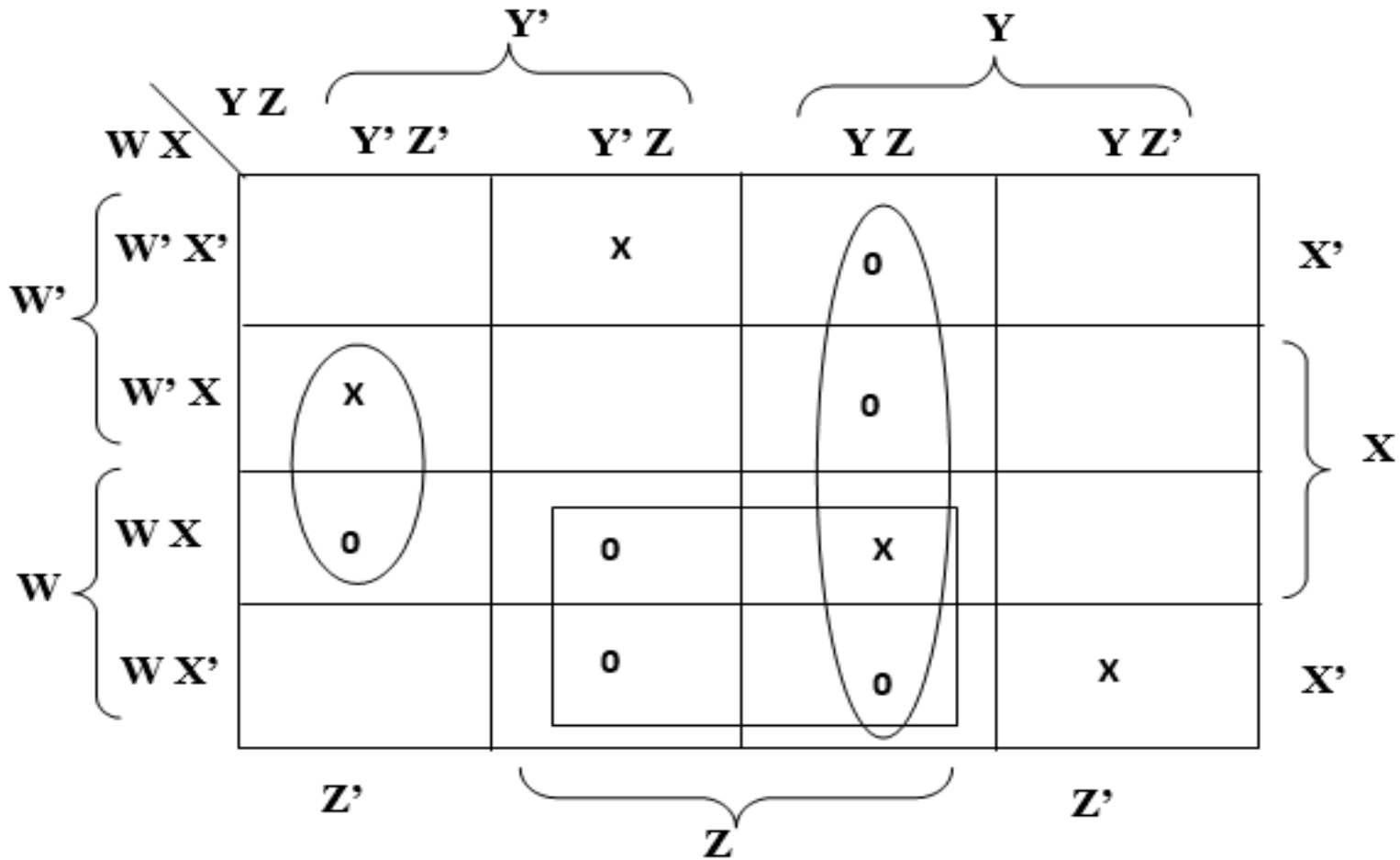
Decimal	W X Y Z	F
0	0 0 0 0	1
1	0 0 0 1	X
2	0 0 1 0	1
3	0 0 1 1	0
4	0 1 0 0	X
5	0 1 0 1	1
6	0 1 1 0	1
7	0 1 1 1	0
8	1 0 0 0	1
9	1 0 0 1	0
10	1 0 1 0	X
11	1 0 1 1	0
12	1 1 0 0	0
13	1 1 0 1	0
14	1 1 1 0	1
15	1 1 1 1	X

Map for SOP:



F = Y Z' + W' Y' + X' Z'

Map for POS:

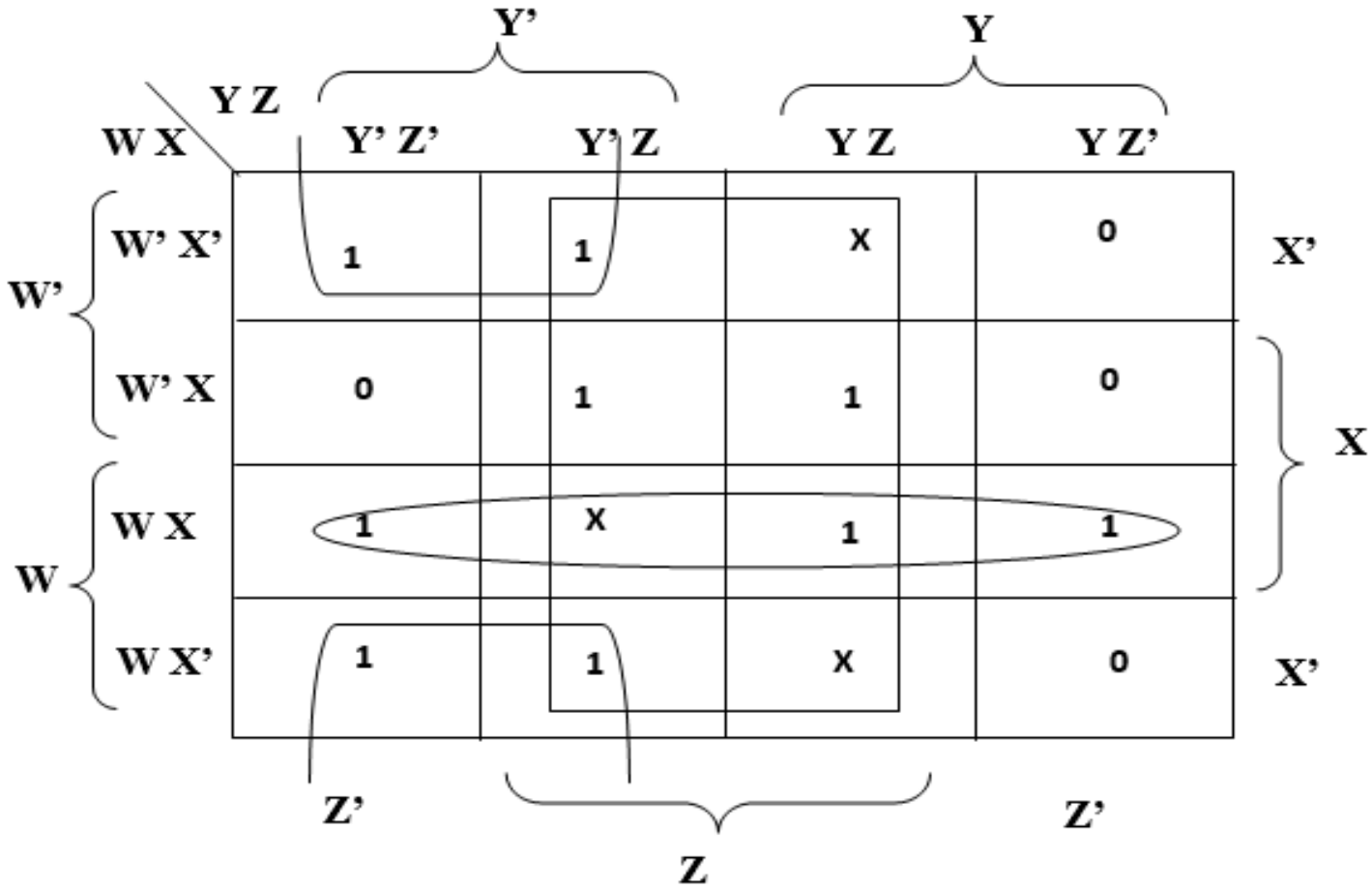


F = (Y' + Z') (W' + Z') (X' + Y + Z)

Example: Minimize and realize using basic gates, NAND gates and NOR gates for both SOP and POS.

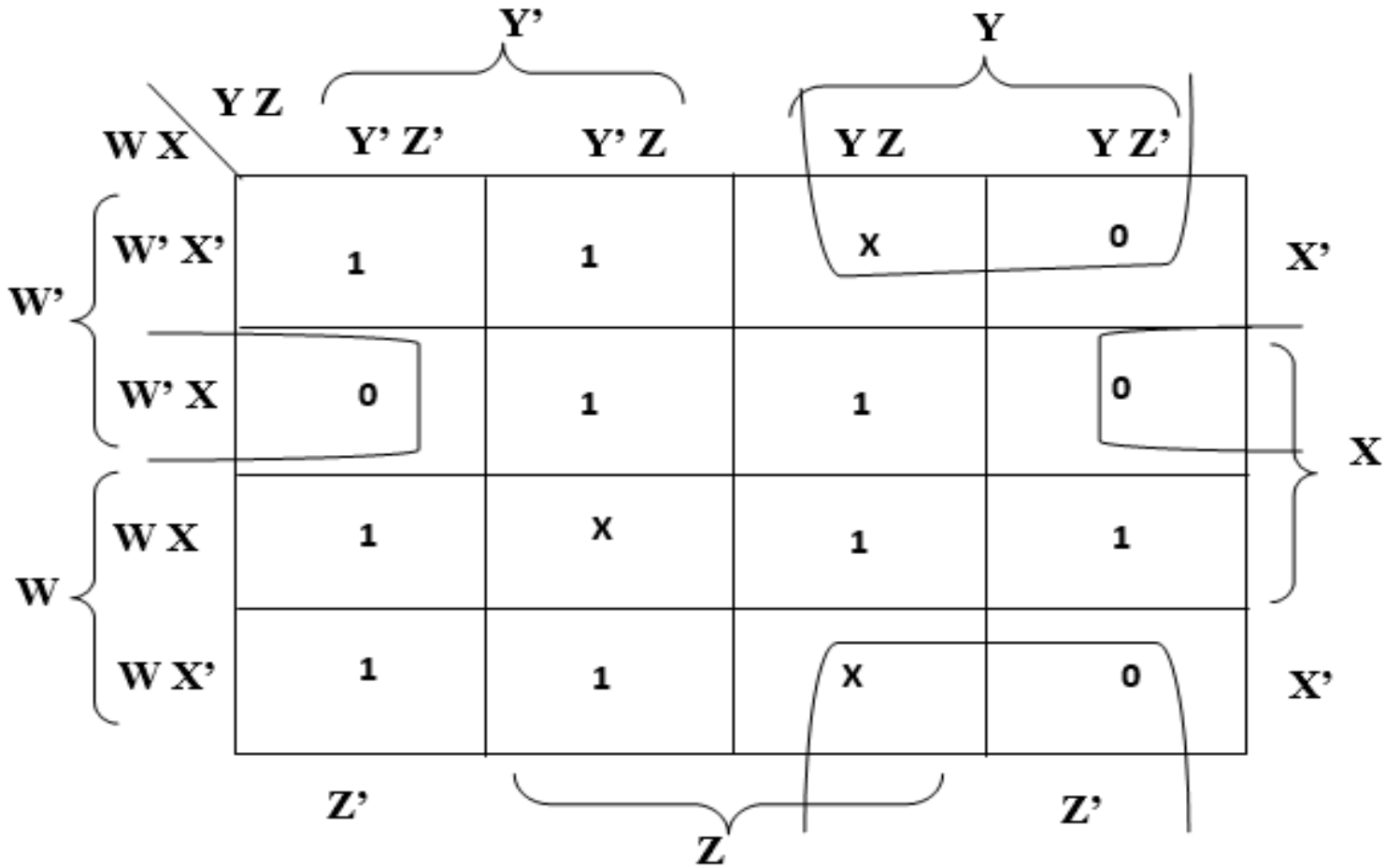
F (W, X, Y, Z) = (P0 + P1 + P5 + P7 + P8 + P9 + P12 + P14 + P15) and don't cares (P3, P11, P13).

Map for SOP:



F= Z + W X + X' Y'

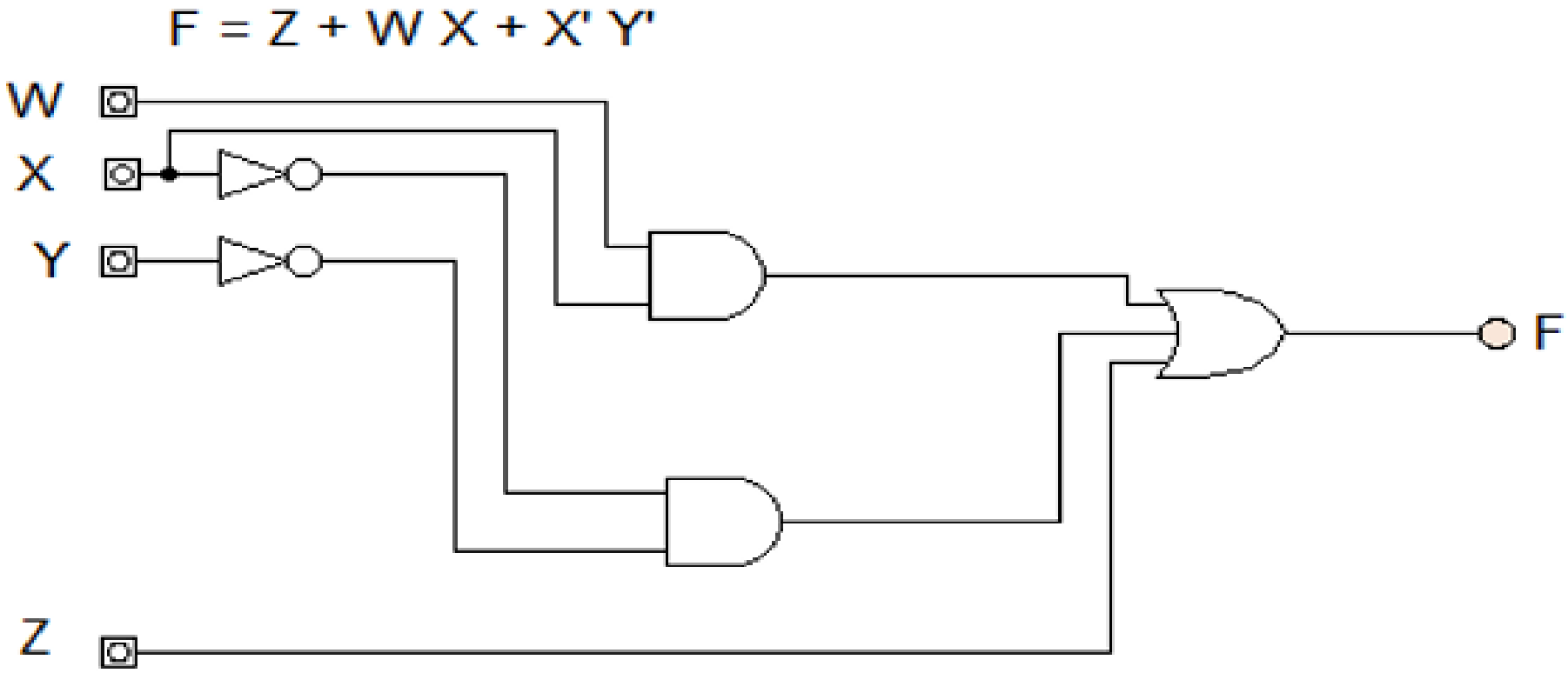
Map for POS:



F = (X + Y') (W + X' + Z)

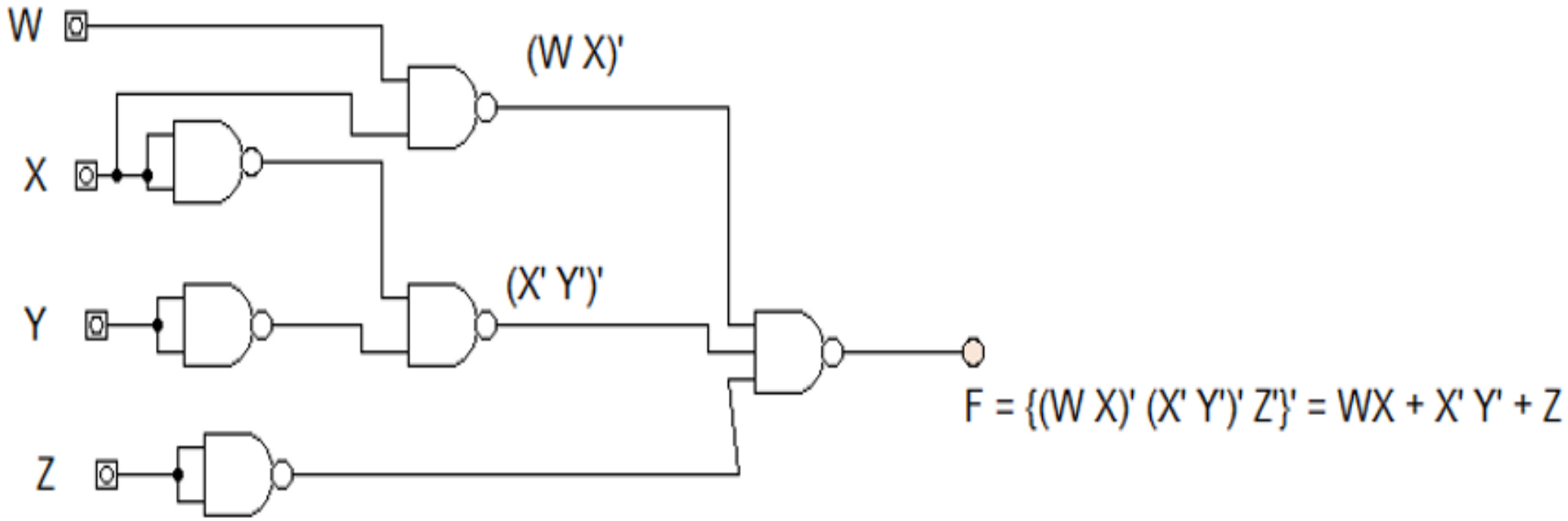
Logic diagram implementation of the function.

Basic gates:



Logic diagram implementation of the function.

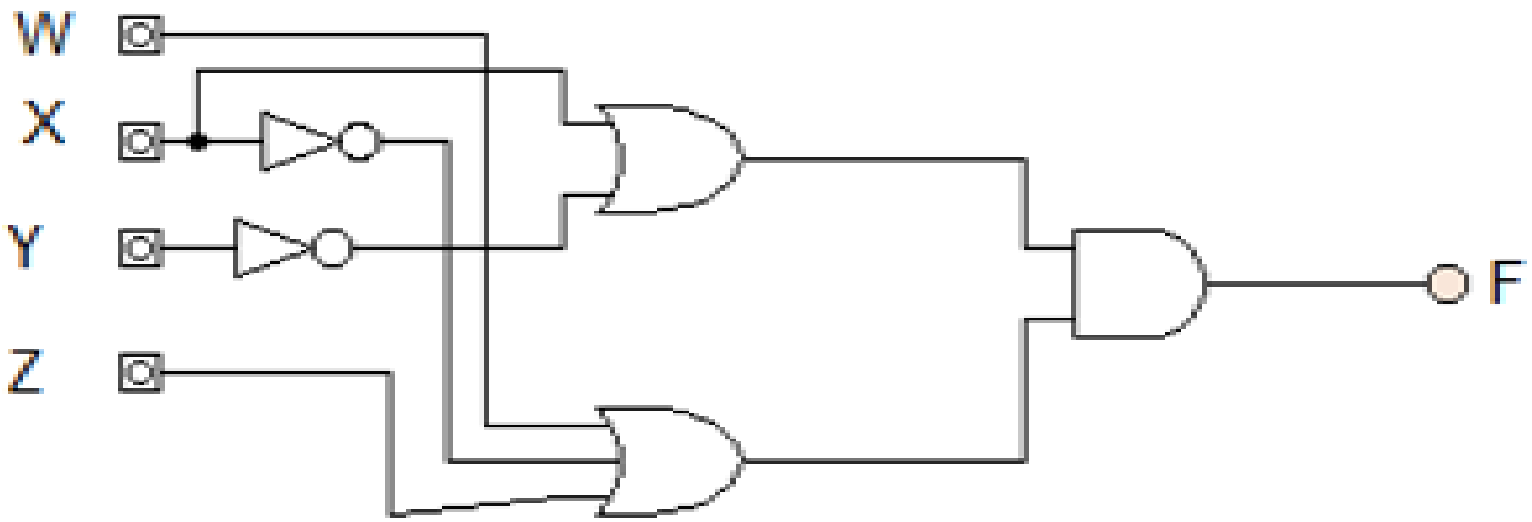
NAND gates:



Logic diagram implementation of the function.

Basic logic gates implementation of the POS expression

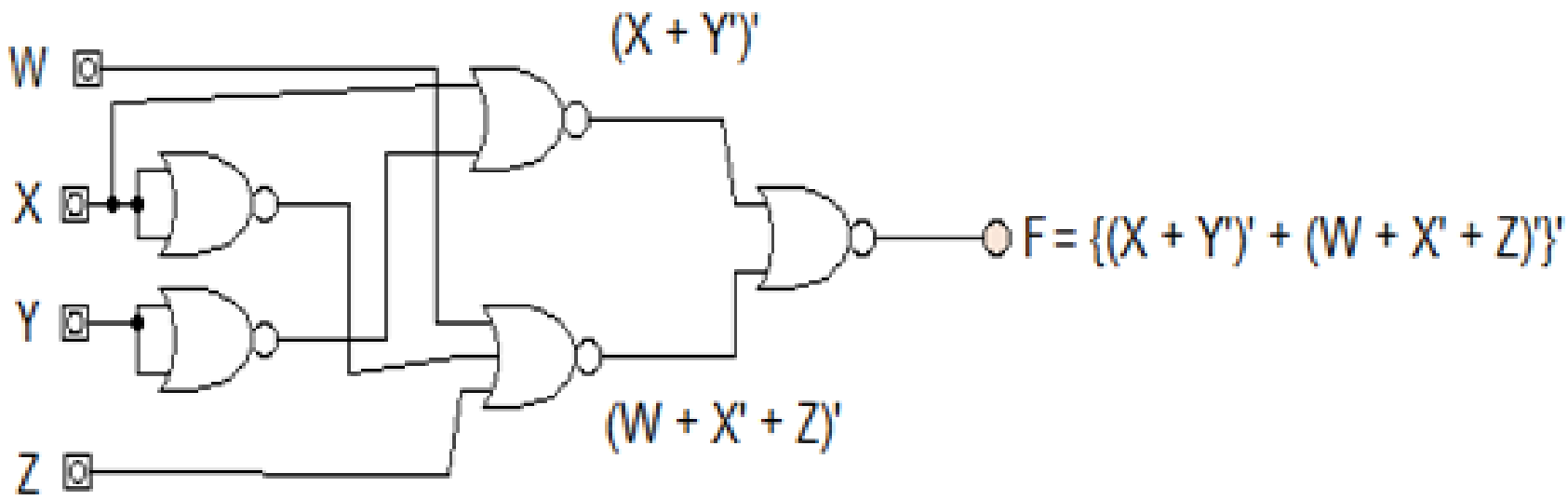
$F = (x + y')\,(W + X' + Z)$



Logic diagram implementation of the function.

NOR gates implementation of the POS expression

$F = (x + y')\,(W + X' + Z)$



NAND and NOR implementation

NAND and NOR implementation

Digital circuits are more frequently constructed with NAND or NOR gates than with AND and OR gates. NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families. The procedure for **two-level implementation** is presented in this section.

NAND and NOR conversions (from AND, OR and NOT implemented Boolean functions)

Because of the prominence of NAND and NOR gates in the design of digital circuits, rules and procedures have been developed for the conversion from Boolean functions given in terms of AND, OR, and NOT into equivalent NAND and NOR logic diagrams.

NAND and NOR implementation

To facilitate the conversion to NAND and NOR logic, there are two other graphic symbols for these gates.

(a) NAND gate

Two equivalent symbols for the NAND gate are shown in diagram below:

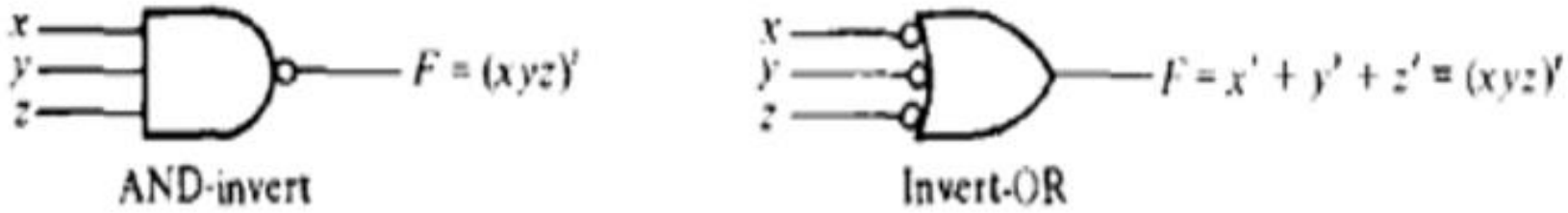


Fig: Two graphic symbols for NAND gate

(b) NOR gate

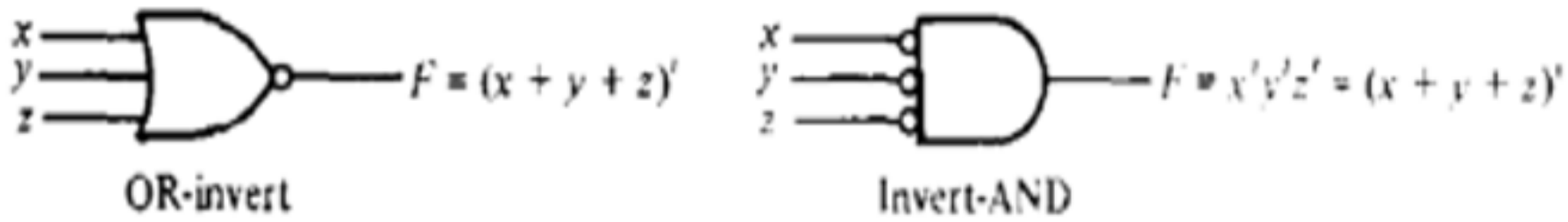
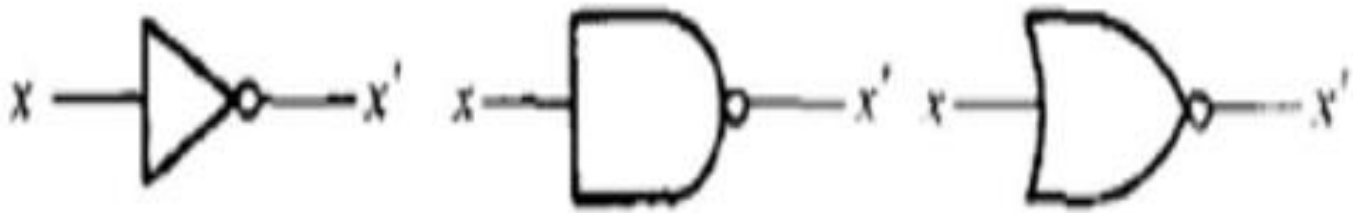


Fig: Two graphic symbols for NOR gate

NAND and NOR implementation

(c) Inverter



Buffer-invert AND-invert OR-invert

Fig: Three graphic symbols for NOT gate

NAND and NOR implementation

The rule for obtaining the NAND logic diagram from a Boolean function is as follows:

First method:

(a) Simplify the function and express it in **sum of products**.

(b) Draw a NAND gate for each product term of the function that has at least two literals. The inputs to each NAND gate are the literals of the term. This constitutes a group of **first-level gates.**

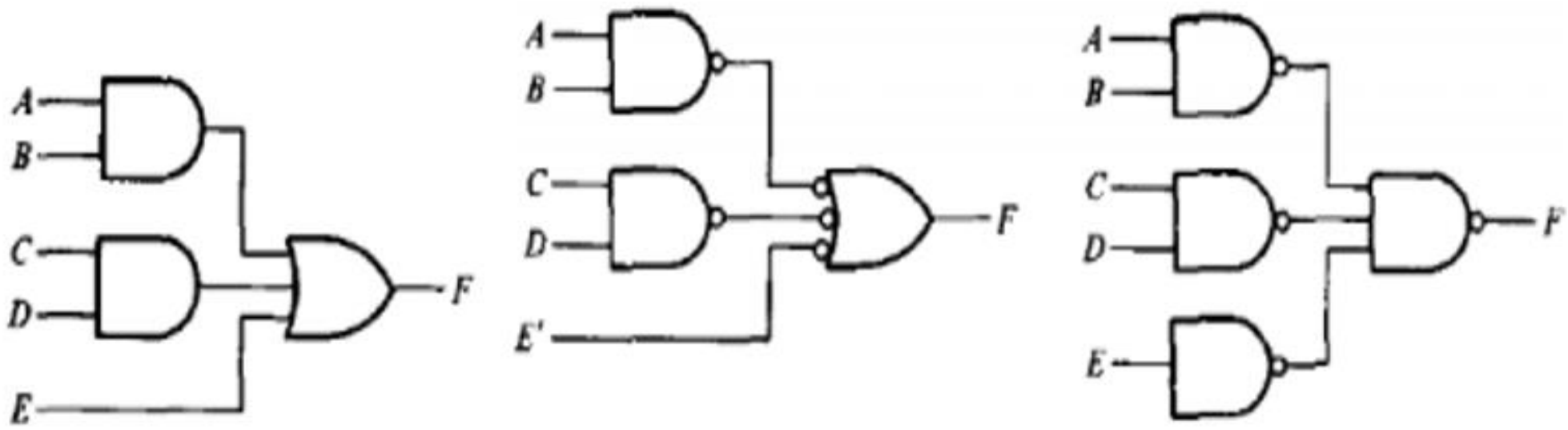
(c) Draw a single NAND gate (using the AND-invert or invert-OR graphic symbol) in the second level, with inputs coming from outputs of first-level gates.

(d) A term with a single literal requires an inverter in the first level or may be complemented and applied as an input to the **second-level NAND gate**.

NAND and NOR implementation

NAND implementation

The implementation of a Boolean function with NAND gates requires that the function be simplified in the sum of products form. To see the relationship between a sum of products expression and its equivalent NAND implementation, consider the logic diagrams of Fig below. All three diagrams are equivalent and implement the function: $F=AB + CD + E$



(a) AND-OR (b) NAND-NAND (c) NAND-NAND

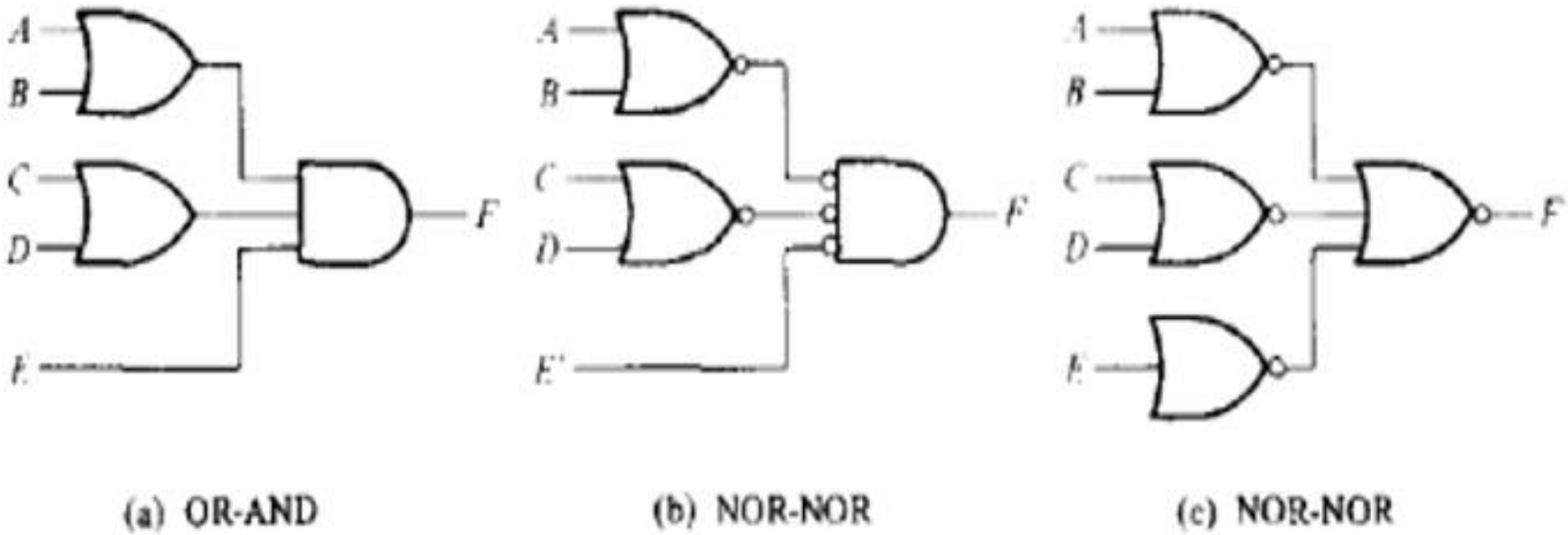
NAND and NOR implementation

NOR Implementation

The NOR function is the dual of the NAND function. For this reason, all procedures and rules for NOR logic are the duals of the corresponding procedures and rules developed for NAND logic. The implementation of a Boolean function with NOR gates requires that the function be simplified in product of sums form. A product of sums expression specifies a group of OR gates for the sum terms, followed by an AND gate to produce the product. The transformation from the OR-AND to the NOR-NOR diagram is depicted in Fig below. It is **similar to the NAND transformation discussed previously, except that now we use** the product of sums expression.

NAND and NOR implementation

$F = (A + B)(C + D)E$



(a) OR-AND (b) NOR-NOR (c) NOR-NOR