PPTS ON
DIGITAL SYSTEM DESIGN

III Semester

Course code AECB07
Academic year 2019-20

by

Dr. V Vijay

Associate Professor

UNIT - I

LOGIC SIMPLIFICATION AND COMBINATIONAL LOGIC DESIGN

Boolean Algebra

- Boolean algebra is the basic mathematics needed for the study of the logic design of digital systems
- Its application to switching circuits is of interest
- switching devices are essentially two-state devices (such as a transistor with high or low output voltage)
- we study the special case of Boolean algebra in which all of the variables assume only one of two values.
- This two-valued Boolean algebra is often referred to as switching algebra.
- if X is a Boolean (switching) variable, then either X = 0 or X = 1
- The values ‘0’ & ‘1’ represent two states of a switching circuit
- In a logic gate circuit, ‘0’ (usually) represents a range of low voltages, and ‘1’ represents a range of high voltages.

Boolean Algebra : Basic Operations

- In a switch circuit, ‘0’ (usually) represents an open switch, and ‘1’ represents a closed switch.
- In general, ‘0’ and ‘1’ can be used to represent the two states in any binary-valued system.

Basic Operations:

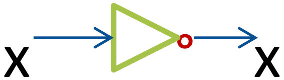
- The basic operations of Boolean algebra are AND, OR, and complement (or inverse).
- The complement of ‘0’ is ‘1’, and the complement of ‘1’ is ‘0’. Symbolically, we write
- 0′ = 1 and 1′ = 0
- where the prime (′) denotes complementation. If X is a switching variable, X′ = 1 if X = 0 & X′ = 0 if X = 1

Boolean Algebra : Basic Operations

Inverter:

- The electronic circuit which forms the inverse of X is referred to as an inverter.

Symbol: Inverter

x  x’ (complement of x)

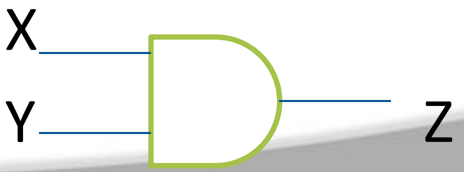
where the circle at the output indicates inversion. Inversion is also called NOT operation . x: High implies x’: Low

High & Low have to do with voltage levels

AND Operation: ‘.’ (Boolean Multiplication)

Z = X.Y : When both inputs are ‘1’ (high) the output is ‘1’ (high), else the output is ‘0’ (Low)

X	Y	Z	X	Y	Z
0	0	0	1	0	0
0	1	0	1	1	1

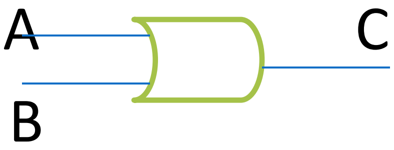


Boolean Algebra : Basic Operations

The OR Operation: ‘+’ (Boolean Addition)

Defined as C = A + B

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



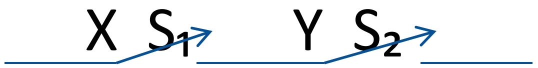
If one of the two inputs A or B is ‘1’ (High)

the output is ‘1’ (High).

If both the inputs are ‘1’ (High) the output is ‘1’

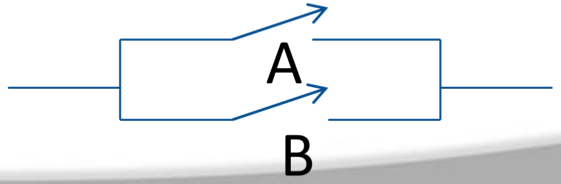
If both inputs are ‘0’ (Low) the output is ‘0’ (Low)

AND Operation: Two Switches in Cascade



= X.Y

OR Operation: Two Switches in Parallel



C = A + B

Boolean Expressions & Truth Tables

Boolean Expression:

One or more variables can be combined in a certain way to yield a Boolean expression that serves the desired purpose.

Examples:

$$A.B' + C; \quad (A+D)'.C + B.D.E$$

Generally ‘.’ is not explicit in a term like ABC; it is understood to be AND operation; ‘+’ implies OR operation and ‘ implies NOT (Inversion/Complementation) operation

The Boolean expressions are synthesized using basic gates like AND OR & NOT.

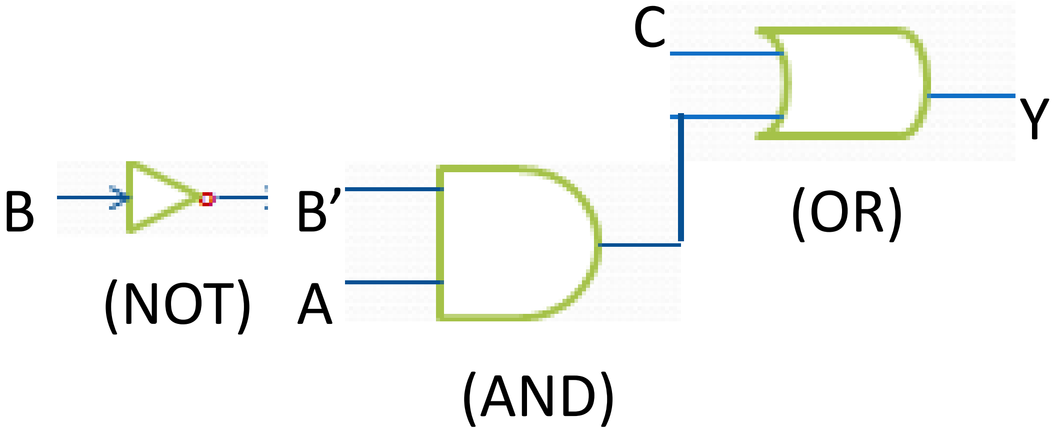
NAND Gate is obtained as NOT[AND] → NOT(A.B) → $\overline{(A.B)}$

NOR Gate is obtained as NOT[OR] → NOT(A+B) → $\overline{(A+B)}$

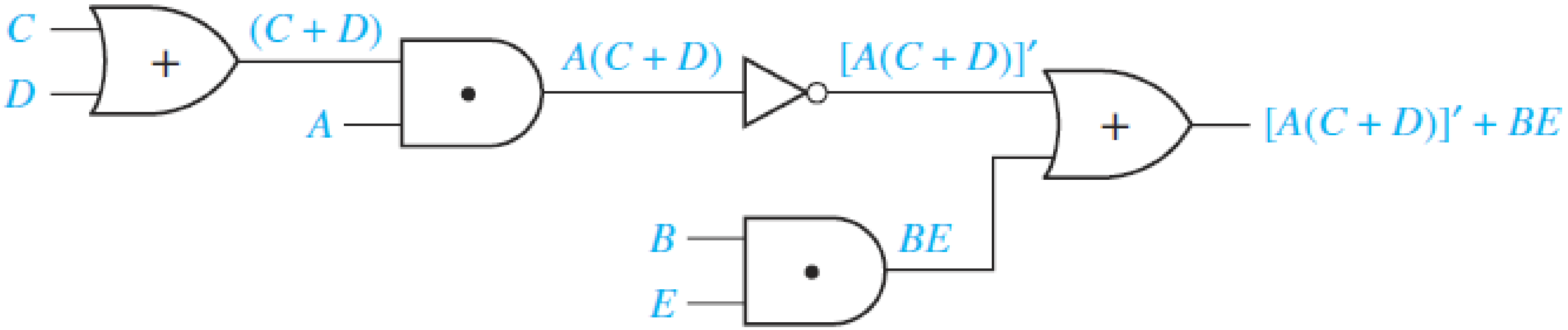
Boolean Expressions & Truth Tables

Boolean Expression Realization with Gates:

Examples: Y = AB’ + C in Figure as shown below



Y = [A(C + D)]′ + BE in Figure as shown below



Boolean Expressions & Truth Tables

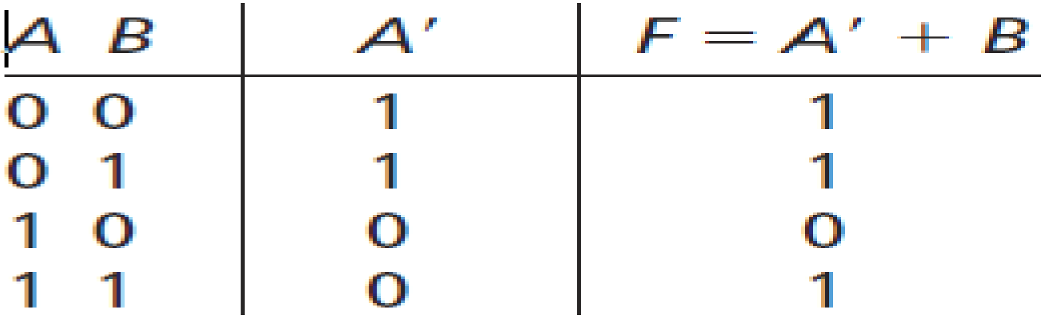
Truth Table:

It specifies the values of a Boolean expression for every possible combination of values of its variables.

Example:

Consider a Boolean expression: F = A’ + B

Its Truth Table is shown below:



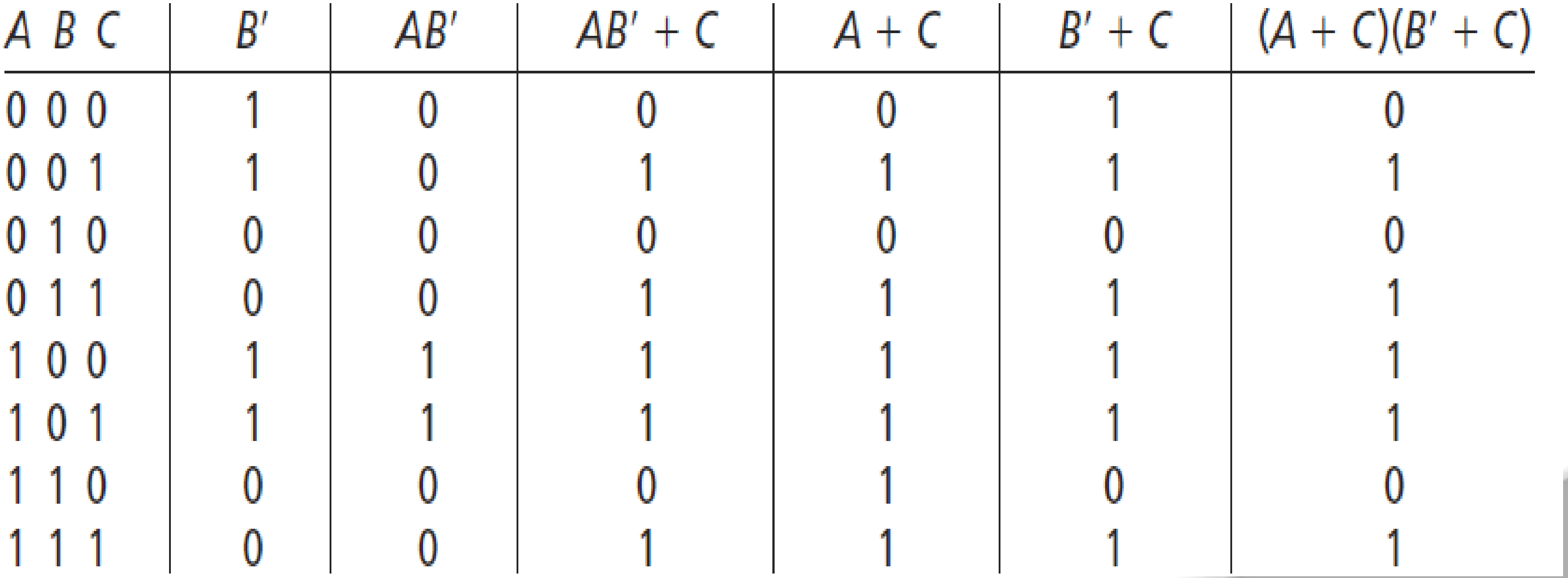
A	B	A'	$F = A' + B$
0	0	1	1
0	1	1	1
1	0	0	0
1	1	0	1

Boolean Expressions & Truth Tables

We will now tabulate Truth table for some functions. We do this for n=3; that is we have 3 bits corresponding to 3 variables.

The number of combinations = N = 2^3 = 8

The table is shown below:



$A\ B\ C$	B'	AB'	$AB' + C$	$A + C$	$B' + C$	$(A + C)(B' + C)$
0 0 0	1	0	0	0	1	0
0 0 1	1	0	1	1	1	1
0 1 0	0	0	0	0	0	0
0 1 1	0	0	1	1	1	1
1 0 0	1	1	1	1	1	1
1 0 1	1	1	1	1	1	1
1 1 0	0	0	0	1	0	0
1 1 1	0	0	1	1	1	1

The table has functions like NOT OR AND & their combinations

Boolean Algebra Basic Theorems

The following basic laws and theorems of Boolean algebra involve only a single variable:

1. Operations with ‘0’ and ‘1’:

$X + 0 = X \qquad X . 1 = X$

$X + 1 = 1 \qquad X . 0 = 0$

(OR Function) (AND Function)

2. Idempotent Laws:

$X + X = X \qquad X . X = X$

3. Involution Law:

$(X')' = X$

4. Laws of Complementarities:

$X + X' = 1 \qquad X . X' = 0$

Boolean Algebra Laws

Commutative Associate and Distributive Laws:

The commutative laws for AND & OR Gates:

$XY = YX$ $\quad$ $X+Y = Y+X$

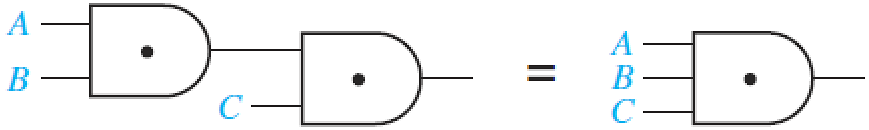
(AND Gate) $\quad$ (OR Gate)

The associative laws for AND & OR Gates:

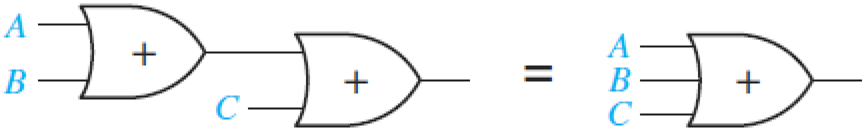
$(XY)Z = X(YZ) = XYZ$ (AND GATE)

$(X + Y) + Z = X + (Y + Z) = X + Y + Z$ (OR Gate)

Thus we conclude that while forming the AND (OR) of three variables, the result is independent of which pair of variables we consider first



$(AB)\,C = ABC$



$(A + B) + C = A + B + C$

Boolean Algebra Laws

Distributive Laws:

$$X(Y + Z) = XY + XZ \quad \dots\dots\dots \; 1$$

$$X + YZ = (X + Y)(X + Z) \quad \dots\dots\dots \; 2$$

Proof for (2):

$(X + Y)(X + Z) = XX + XZ + YX + YZ$

$= X + XZ + XY + YZ$ (ANDing of X & X yields X)

$= X.1 + XZ + XY +YZ$ (ANDing of X with 1 = X)

$= X(1 + Z + Y) + YZ$ (ORing 1 with (Z OR Y) = 1)

$= X.1 + YZ$ (ANDing X with 1 = X)

$= X + YZ$

This second law is very useful in manipulating Boolean expressions. It cannot be factored in ordinary algebra.

Boolean Algebra Simplification Theorems

1. $XY + XY' = X$

Proof:

$XY + XY' = X(Y + Y') = X$; as Y OR Y’ = 1

2. $X + XY = X$

Proof:

$X + XY = X(1 + Y) = X$; as ‘1’ OR ‘Y’ = 1

3. $(X + Y')Y = XY$

Proof:

$(X + Y')\,Y = XY + Y'Y = XY$; as Y’ AND Y = 0

4. $(X + Y)(X + Y') = X$

Proof:

$(X + Y)(X + Y') = XX + XY' + YX + YY'= X + X(Y' + Y) = X + X = X$

Boolean Algebra Simplification Theorems

5. $X(X + Y) = X$

Proof:

$X(X + Y) = XX + XY = X + XY = X(1+Y) = X$

6. $XY' + Y = X + Y$

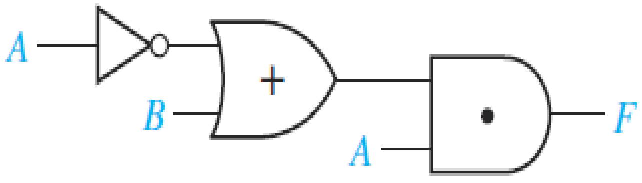
Proof:

$$XY' + Y = (Y + X)(Y + Y') = YY + YY' + XY + XY'$$
$$= Y + 0 + X(Y + Y') = Y + X$$

Usefulness of Simplification:

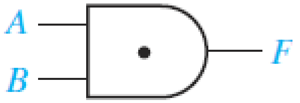
Consider a function $F = A(A' + B)$

Realization without Simplification:



With Simplification

(Reduction in number of Gates)



$F = AB$

Boolean Expression Simplification Examples

1. Simplify

$$Z = [A + B'C + D + EF][A + B'C +(D + EF)']$$

First step: Look for similarity of expressions in two factors.

We observe that we can let

$$Y = A + B'C \qquad X = D + EF$$

$$Z = [Y + X][Y + X'] = YY + YX' + XY + XX'$$

$$= Y + YX' +XY + 0$$

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

Therefore $Z = A + BC'$

Boolean Expression Simplification Examples

2. Simplify:

$$Z = (AB + C)(B'D + C'E') + (AB + C)'$$

We let $Y = (AB + C)'$ & $X = B'D + C'E'$ to get the familiar form

$$Z = Y'X + Y = X + Y$$ (by theorem)

Therefore,

$$Z = B'D + C'E' + (AB + C)'$$

is the simplified version.

Application of Second Distributive Law for simplifying Product terms:

The law is : $(X+Y)(X+Z) = X + YZ$

Example:

Multiply out $(A + BC)(A+D+E)$; Let $BC = Y$ & $D+E = Z$; $A = X$

to get $(X + Y)(X + Z) = X + YZ = A + BC(D+E)$

Boolean Expression Simplification Examples

Factorization using Second Distributive Law:

1. Factorize A + B’CD (Sum of Products)

Let X = A; B’ = Y & CD = Z

A + B’CD = X + Y Z = (X+Y)(X+Z)

= (A + B’)(A + CD)

2. Factorize AB’ + C’D using 2nd distributive law

Let AB’ = X & C’ = Y , D = Z

AB’ + C’D = X + YZ = (X+Y)(X+Z)

= (AB’+ Y)(AB’ + Z)

= (Y + A)(Y + B’)(A + Z)(Z + B’)

= (C’ + A)(C’+B’)(A + D)(D+B’)

Factorized function is a Product of Sums function

Boolean Expression Simplification Examples

3. Factorize:

$C'D + C'E' + G'H$ (Sum of Products)

$= C'(D + E') + G'H$; Let $G'H = X$; $C' = Y$ & $(D+E') = Z$

$= YZ + X$

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

$= (G'H + C')(G'H + (D+E'))$

$= (C' + G')(C' + H)[(D+E') + G'][(D+E') + H]$

$= (C' + G')(C' + H)(D + E' + G')(D + E' + H)$

is the factorize form (Product of Sums expression) .

Boolean Expression Simplification Examples

4. Factorize

C’D + C’E’ + G’H (Sum of products expression)

= C’(D+E) +G’H

Let X = G’H Y = C’ Z = D+E

We get the expression in X + YZ form which can be factorized as

(X+Y)(X+Z)

Therefore,

C’(D+E) +G’H = (G’H + C’) (G’H + D + E)

Now, (G’H + C’) = (C’ + G’)(C’ + H) , &

(G’H + D + E) = (D + E + G’)(D+E+H)

Therefore, we get Product of Sums expression

C’(D+E) + G’H = (C’ + G’)(C’ + H) (D + E + G’)(D+E+H)

Realization of Functions with Gates

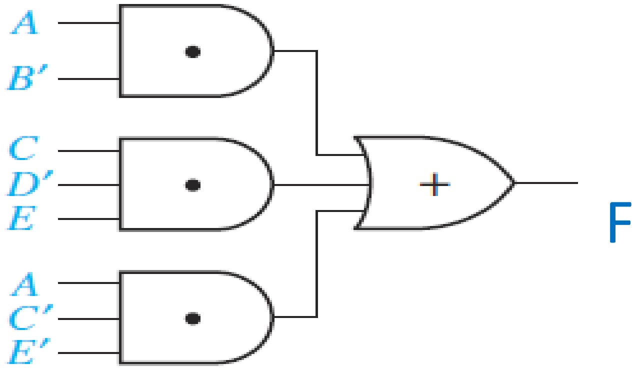
Sum of Products Expression:

It is realizable directly by one or more AND gates feeding a single OR gate at the circuit output.

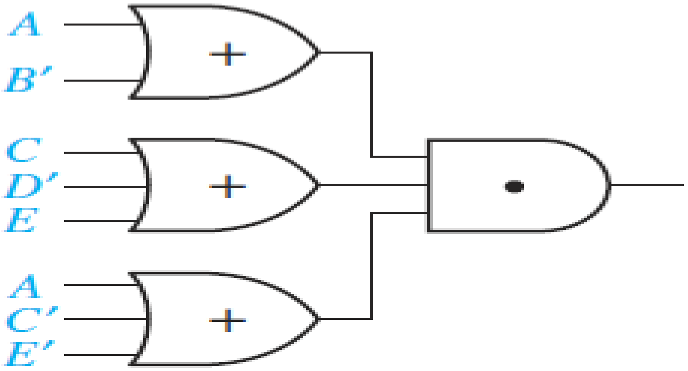
Product of Sums Expression:

It is realizable directly by one or more OR gates feeding a single AND gate at the circuit output.

Let F = AB′ + CD′E + AC′E′



Let F = (A +B′)(C+ D′ + E)(A + C′ + E′)



De Morgan’s Laws

De Morgan’s laws are applied to determine complement (inverse) of a Boolean function.

The Law’s state that (for a 2 variable function)

$(X + Y)' = X' Y'$ (complement of sum = product of complements)

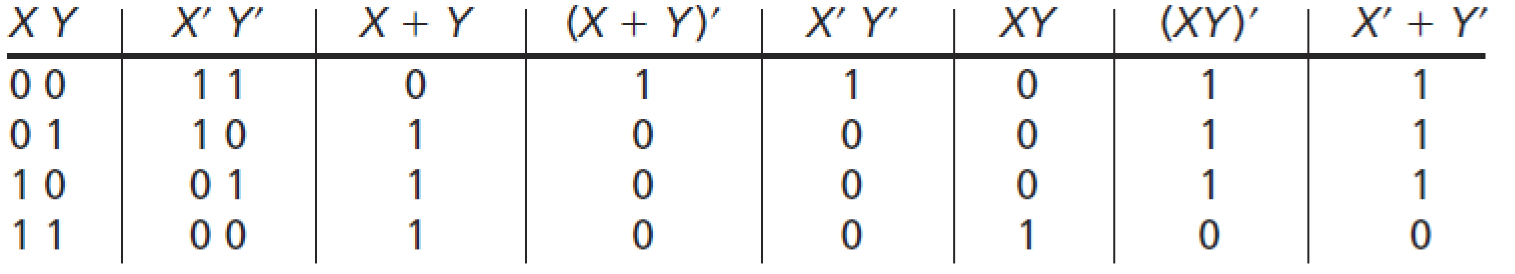
$(XY)' = X' + Y'$ (complement of product = sum of complements)

Extending to ‘N’ number of variables we have,

$(X_1 + X_2 + X_3 \ . \ . \ . + Xn)' = X_1' X_2' X_3' \ . \ . \ . \ Xn'$

$(X_1 X_2 X_3 \ . \ . \ . \ Xn)' = X_1' + X_2' + X_3' \ . \ . \ . + Xn'$

Verification using Truth table



XY	$X'Y'$	$X + Y$	$(X + Y)'$	$X'Y'$	XY	$(XY)'$	$X' + Y'$
0 0	1 1	0	1	1	0	1	1
0 1	1 0	1	0	0	0	1	1
1 0	0 1	1	0	0	0	1	1
1 1	0 0	1	0	0	1	0	0

De Morgan's Laws: Applications

Example 1:

Find the complement of $F = (A' + B)C'$

$F' = [(A'+B)C']'$ (this is complement of products)

Now according to the Law:

complement of products = sum of complements , &

complement of sums = product of complements

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

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

$F' = A B' + C$

Example 2: Find the complement of $(AB' + C)D' + E$

$F' = [(AB' + C)D' + E]'$

$F' = [(AB' + C)D']' E' = [(AB' + C)' + D]E'$ (Contd...)

De Morgan’s Laws: Applications

$$F' = [(AB' + C)' + D]E'$$

$$= [(AB')'C' + D]E'$$

$$F' = [(A' + B)C' + D]E'$$

Example 3: Determine complement of $F = A'B + AB'$

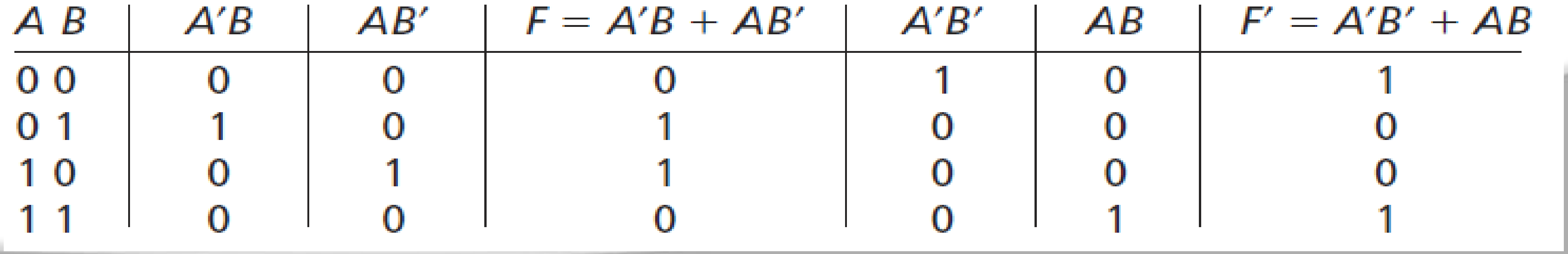
$$F' = (A'B + AB')'$$

$$= (A'B)'(AB')'$$

$$= (A + B')(A' + B)$$

$$= AA' + AB + B'A' + BB' = A'B' + AB$$

Verification using Truth table:



$A\ B$	$A'B$	AB'	$F = A'B + AB'$	$A'B'$	AB	$F' = A'B' + AB$
0 0	0	0	0	1	0	1
0 1	1	0	1	0	0	0
1 0	0	1	1	0	0	0
1 1	0	0	0	0	1	1

Dual of a Boolean Expression

Given a Boolean expression,

- the dual is formed by replacing
- AND with OR,OR with AND
- ‘0’ with ‘1’, and ‘1’ with ‘0’
- Variables and complements are left unchanged
- dual of AND is OR and the dual of OR is AND
- $(XYZ\ldots)^{d} = X + Y + Z \ \ldots$; superscript ‘d’ implies dual
- $(X + Y + Z \ \ldots)^{d} = XYZ\ldots$

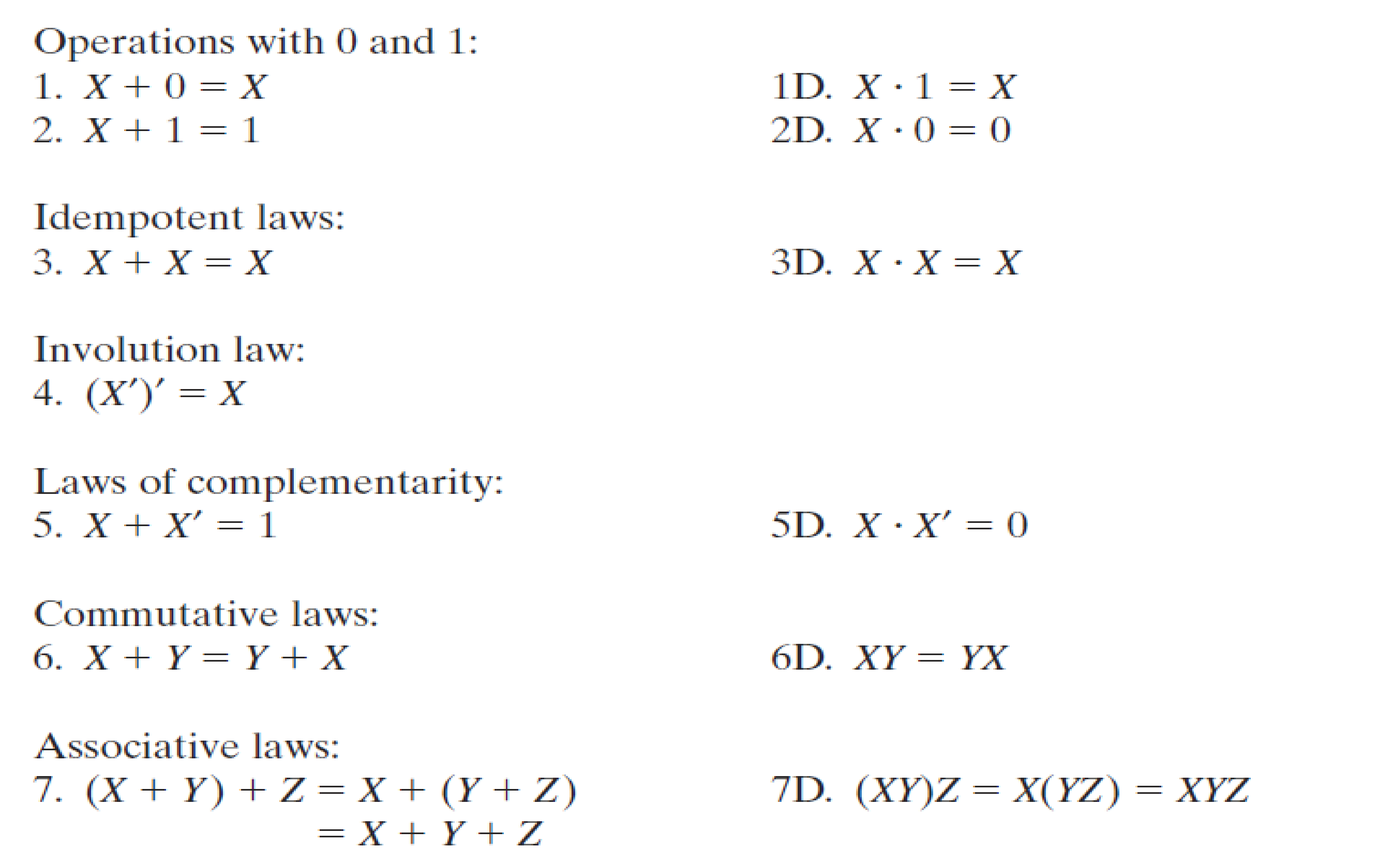
The dual of an expression may be found by complementing the entire expression and then complementing each individual variable.

Dual of $AB' + C = (AB' + C)' = (AB')'C' = (A' + B)C'$,

Therefore, $(AB' + C)^{d} = (A + B')C$

Laws and theorems of Boolean algebra in dual pairs

Laws and theorems of Boolean algebra listed in dual pairs



Operations with 0 and 1:

1. $X + 0 = X$ 1D. $X \cdot 1 = X$

2. $X + 1 = 1$ 2D. $X \cdot 0 = 0$

Idempotent laws:

3. $X + X = X$ 3D. $X \cdot X = X$

Involution law:

4. $(X')' = X$

Laws of complementarity:

5. $X + X' = 1$ 5D. $X \cdot X' = 0$

Commutative laws:

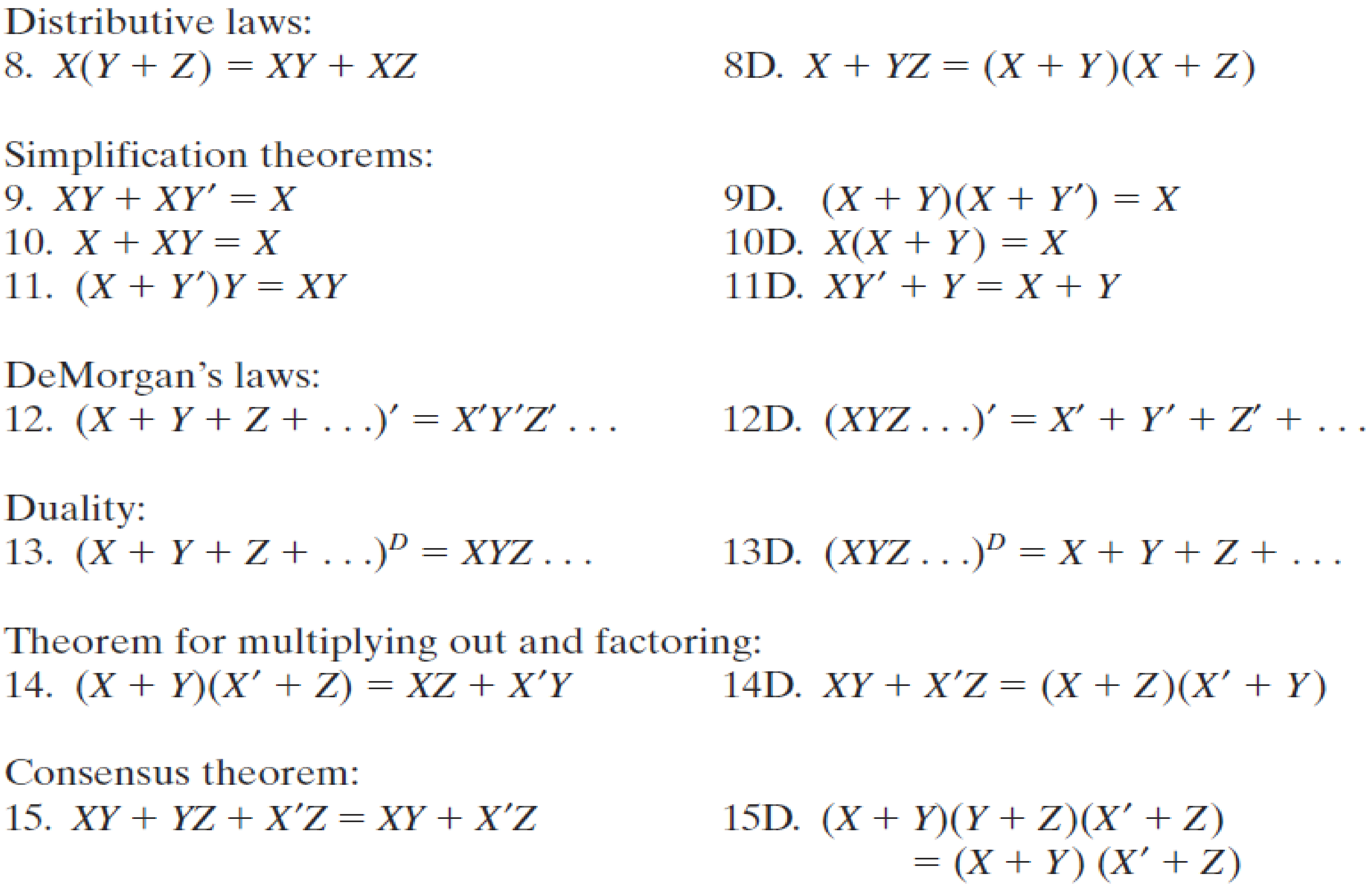
6. $X + Y = Y + X$ 6D. $XY = YX$

Associative laws:

7. $(X + Y) + Z = X + (Y + Z) = X + Y + Z$ 7D. $(XY)Z = X(YZ) = XYZ$

Laws and theorems of Boolean algebra in dual pairs

(Continued …..)



Distributive laws:

8. $X(Y + Z) = XY + XZ$

8D. $X + YZ = (X + Y)(X + Z)$

Simplification theorems:

9. $XY + XY' = X$

9D. $(X + Y)(X + Y') = X$

10. $X + XY = X$

10D. $X(X + Y) = X$

11. $(X + Y')Y = XY$

11D. $XY' + Y = X + Y$

DeMorgan's laws:

12. $(X + Y + Z + \ldots)' = X'Y'Z' \ldots$

12D. $(XYZ \ldots)' = X' + Y' + Z' + \ldots$

Duality:

13. $(X + Y + Z + \ldots)^D = XYZ \ldots$

13D. $(XYZ \ldots)^D = X + Y + Z + \ldots$

Theorem for multiplying out and factoring:

14. $(X + Y)(X' + Z) = XZ + X'Y$

14D. $XY + X'Z = (X + Z)(X' + Y)$

Consensus theorem:

15. $XY + YZ + X'Z = XY + X'Z$

15D. $(X + Y)(Y + Z)(X' + Z) = (X + Y)(X' + Z)$

Useful Distributive Law Theorem

We know two distributive laws as

$$X(Y + Z) = XY + XZ \quad ; \quad (X + Y)(X + Z) = X + YZ$$

There is another distributive law (theorem) quite useful for simplifying expressions. It can be applied when there are two terms, one which contains a variable and another which contains its complement.

$$(X + Y)(X' + Z) = XZ + X'Y \quad \ldots\ldots (1)$$

We observe that the variable that is paired with X on one side of the equation is paired with X' on the other side, and vice versa.

Example: Factorize AB + A'C

Since in this expression one term has 'A' & other has A′ we can use (1) to factorize it. We get

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

Useful Distributive Law Theorem

Example 1:

(Q + AB′)(C′D + Q′) = QC′D + Q′AB′

In the LHS, we have Q & in the RHS we have Q′. Therefore Q will combine with C′D & Q′ will combine with AB′.

Using the theorem It is easier to simplify the expression on the LHS than to expand it in Sum of Products form for further simplification. It is not easy to simplify a term like AB′C′D.

Example 2:

F = AC + A′BD′ + A′BE + A′C′DE

= AC + A′(BD′ + BE + C′DE) ; Apply theorem to get

= (A + BD′ + BE + C′DE)(C + A′) = [(A+C′DE) + B(D′+E)](C + A′);

Let X = A+C′DE; Y = B ; Z = D′+ E & apply distributive law to get

F = (A+B+C′DE)(A+C′DE+D′+E)(C+A′)

(continued …)

Useful Distributive Law Theorem

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

Let us simplify it term wise:

$(A+B+C'DE) = (A+B+C')(A+B+DE) = (A+B+C')(A+B+D)(A+B+E)$ ... 1

$(A+C'DE+D'+E) = (A+E+D'+C')(A+E+D'+DE)=(A+E+D'+C')(A+D'+E)$..2

Let $A+D'+E = X$

$(A+E+D'+C')(A+D'+E) = (X + C')(X) = X.X + C'.X = X(1+C') = X$

$= A+D'+E$ 3

Therefore from 1 & 3, we have

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

Minterm (Maxterm)/Standard Product(Sum)

- A binary variable may appear either in its normal form (X) or in its complement form (X′)
- Consider two binary variables X and Y combined with an AND operation
- Since each variable may appear in either form, there are four possible combinations:
- X′ Y′ , X′ Y, X Y′ , and X Y ..... (1)
- Each of the terms in (1) is called a minterm or standard product.
- n variables can be combined to form 2^n minterms

In a similar fashion,

- n variables forming an OR term, with each variable being primed or un-primed
- provide 2^n possible combinations, called maxterms, or standard sums
- The binary numbers from 0 to $2^n - 1$ are listed under the n(=3) variables.

Minterm (Maxterm)/Standard Product(Sum)

- Each minterm is obtained from an AND term of the n variables
- each variable being primed if the corresponding bit of the binary number is a ‘0’ and un-primed if a ‘1’
- each maxterm is obtained from an OR term of the *n variables,*
- each variable being un-primed if the corresponding bit is a ‘0’ and primed if a ‘1’
- each maxterm is the complement of its corresponding minterm and vice versa

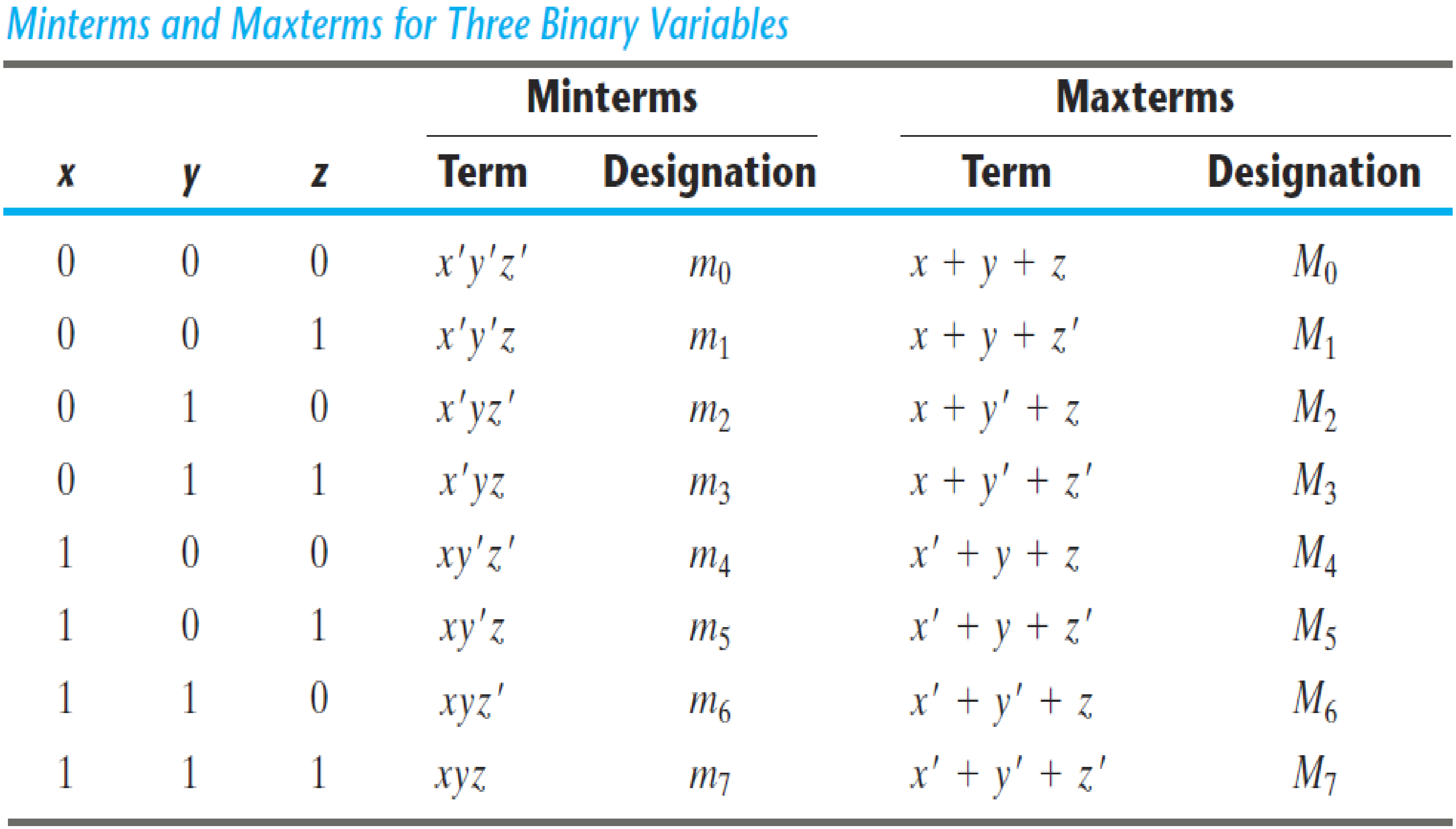
SEE TABLE 1 in the next slide

Determination of Boolean function from a Truth Table:

A Boolean function can be expressed algebraically from a given truth table by forming a minterm for each combination of the variables that produces a ‘1’ in the function and then taking the OR of all those terms (See Table 2)

Minterm (Maxterm)/Standard Product(Sum)

The eight minterms/maxterms for three variables, together with their symbolic designations, are listed in the Table 1 below

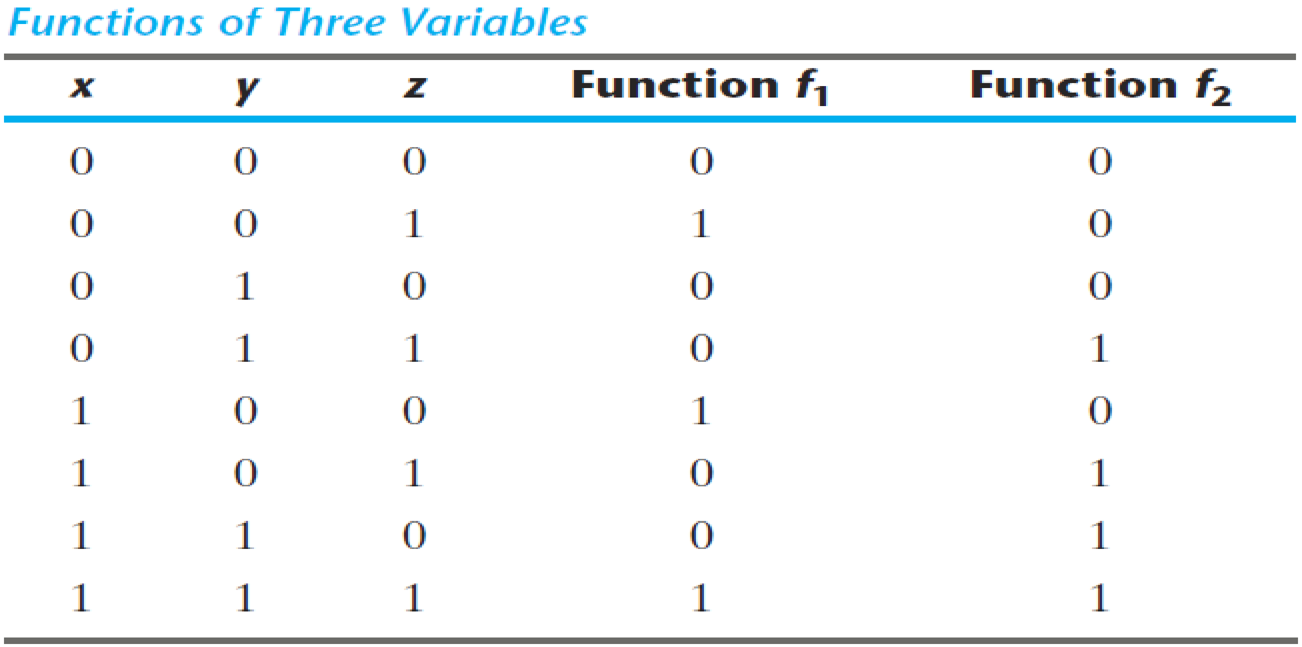


Minterms and Maxterms for Three Binary Variables

			Minterms		Maxterms	
x	**y**	**z**	**Term**	**Designation**	**Term**	**Designation**
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

Minterm (Maxterm)/Standard Product(Sum)

Let us consider the table as shown. **Table 2**



Functions of Three Variables

x	y	z	Function f_1	Function f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$F_1 = m_1 + m_4 + m_7$

$F_2 = m_3 + m_5 + m_6 + m_7$

$F_1 = 1$ for { 001 100 111} ; $F_2 = 1$ for { 011 101 110 111}

Therefore, $F_1 = X'YZ + XY'Z' + XYZ$ & $F_2 = X'YZ + XY'Z + XYZ' + XYZ$

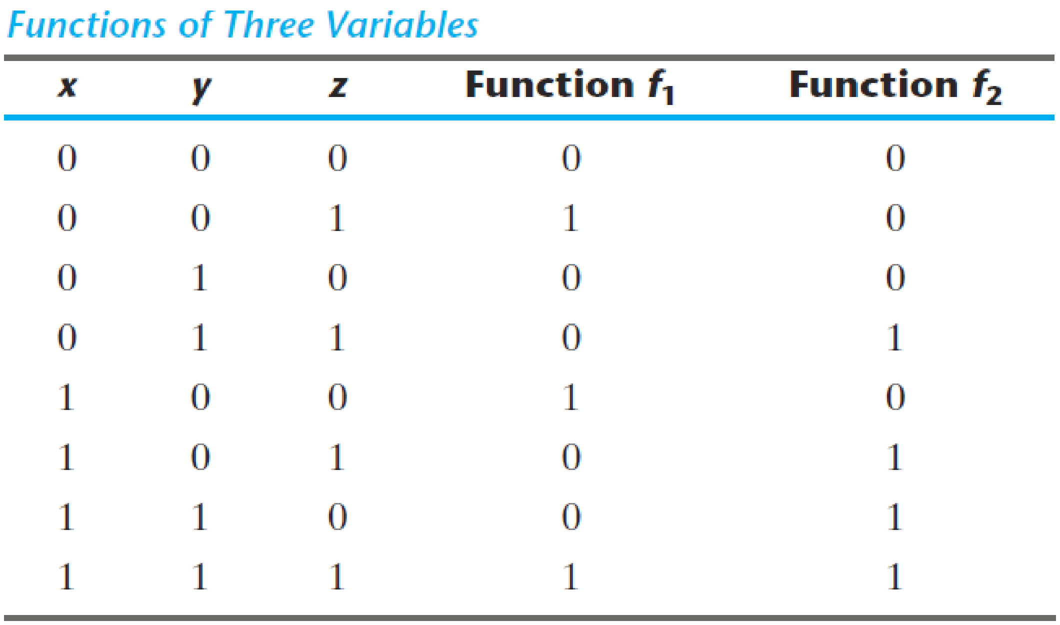
F_1 & F_2 are expressed in Sum of Products Form (each product term is a minterm)

Thus any Boolean function can be expressed as a sum of minterms

Minterm (Maxterm)/Standard Product(Sum)

Now consider the complement of a Boolean function.

Complement may be read from the truth table by forming a minterm for each combination that produces a ‘0’ in the function and then ORing those terms.



Functions of Three Variables

x	*y*	*z*	**Function f_1**	**Function f_2**
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$F_1' = X'Y'Z' + X'YZ' + X'YZ$
$+ XY'Z + XYZ'$

$F_2' = X'Y'Z' + X'Y'Z + X'YZ'$
$+ XY'Z'$

Take complement of F_1 & F_2 (Continued…)

Minterm (Maxterm)/Standard Product(Sum)

$F_1' = X'Y'Z' + X'YZ' + X'YZ + XY'Z + XYZ'$

$F_2' = X'Y'Z' + X'Y'Z + X'YZ' + XY'Z'$

Take complement of F_1 & F_2

$F_1 = (X+Y+Z)(X+Y'+Z)(X+Y'+Z')(X'+Y+Z')(X'+Y'+Z) = M_oM_2M_3M_5M_6$

$F_2 = (X+Y+Z)(X+Y+Z')(X+Y'+Z)(X'+Y+Z) = M_oM_1M_2M_4$

Thus Any Boolean function can be expressed as a product of maxterms (with “product” meaning the ANDing of terms)

Boolean functions expressed as a sum of minterms or product of maxterms are said to be in ***canonical*** *form* .

Minterm (Maxterm)/Standard Product(Sum)

Sum of Minterms:

- It is sometimes convenient to express a Boolean function in its sum-of-minterms form
- If the function is not in this form, it can be made so by first expanding the expression into a sum of AND terms.
- Each term is then inspected to see if it contains all the variables.
- If it misses one or more variables, it is ANDed with an expression such as x + x, where x is one of the missing variables.

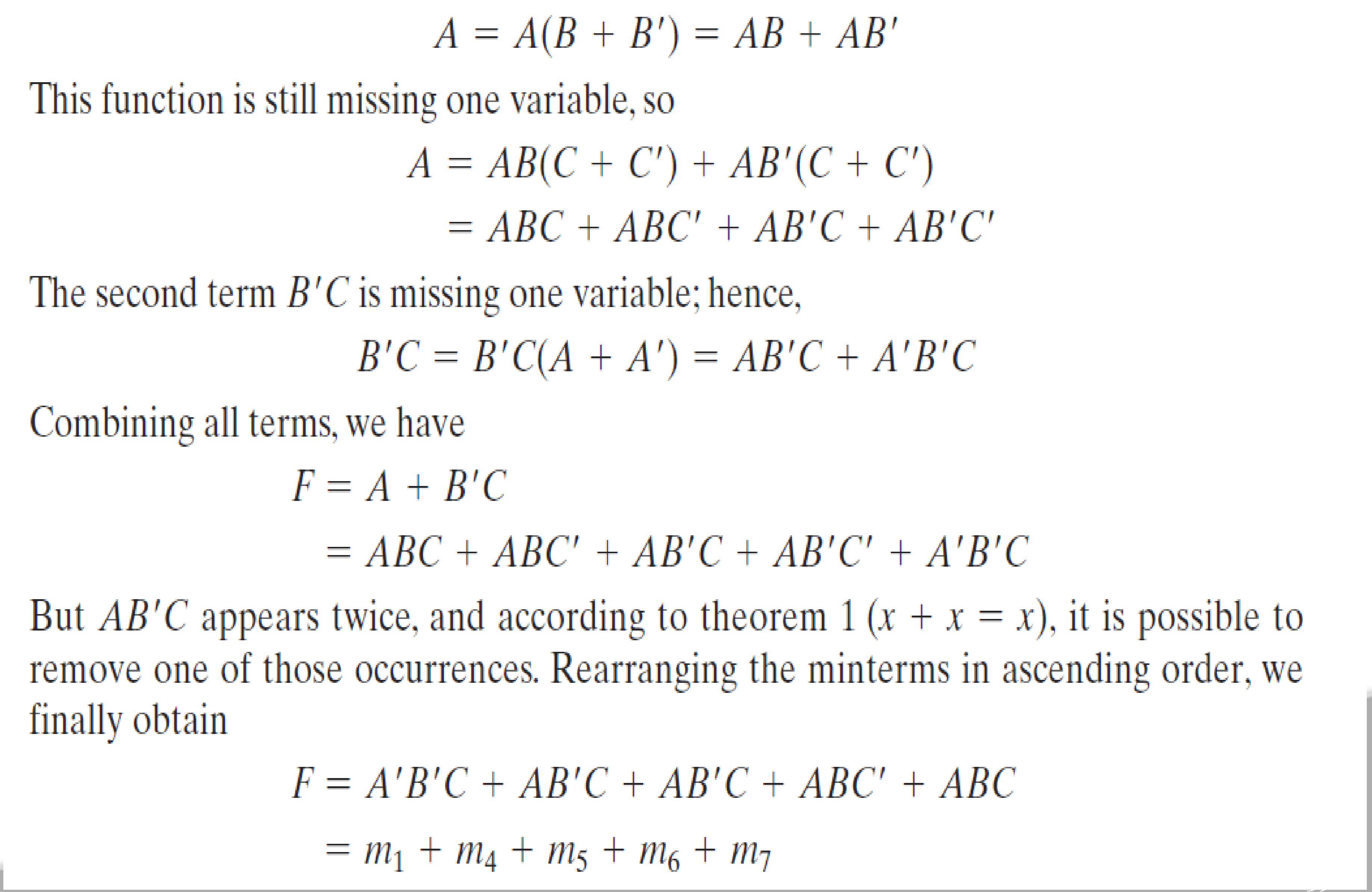
Example:

Express the Boolean function $F = A + B'C$ *as a sum of minterms*

- The function has three variables: A, B, and C.
- The first term A is missing two variables; therefore,

(Continued ..)

Minterm (Maxterm)/Standard Product(Sum)



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

This function is still missing one variable, so

$$A = AB(C + C') + AB'(C + C')$$
$$= ABC + ABC' + AB'C + AB'C'$$

The second term $B'C$ is missing one variable; hence,

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

Combining all terms, we have

$$F = A + B'C$$
$$= ABC + ABC' + AB'C + AB'C' + A'B'C$$

But $AB'C$ appears twice, and according to theorem 1 ($x + x = x$), it is possible to remove one of those occurrences. Rearranging the minterms in ascending order, we finally obtain

$$F = A'B'C + AB'C + AB'C + ABC' + ABC$$
$$= m_1 + m_4 + m_5 + m_6 + m_7$$

Minterm/Maxterm Form & Conversion

Notation for Sum of Minterm Form:

it is sometimes convenient to express the function in the following brief notation: $F(A, B, C) = \Sigma(1, 4, 5, 6, 7)$

- The summation symbol Σ stands for the ORing of terms;
- the numbers following it are the indices of the minterms of the function.
- The letters in parentheses following F form a list of the variables in the order taken when the minterm is converted to an AND term

For example,

$$\begin{aligned} F &= A'B'C + AB'C + AB'C + ABC' + ABC \\ &= m_1 + m_4 + m_5 + m_6 + m_7 \end{aligned}$$

Minterm/Maxterm Form & Conversion

We studied the procedure for determining the minterms of such a Boolean function in which all the terms do not contain all the literals. We will now explore an alternative method for doing the same.

Let us consider the same function ; F = A + B′C

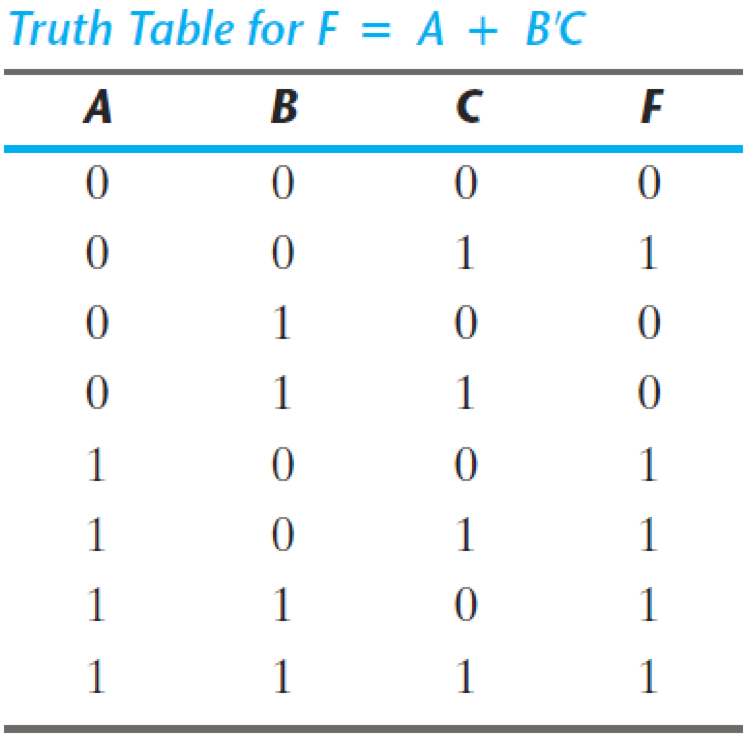
Alternative Procedure:

- Obtain the truth table of the function directly from the algebraic expression and
- then read the minterms from the truth table.

We will first form the Truth table for the given function

(continued …)

Minterm/Maxterm Form & Conversion



Truth Table for $F = A + B'C$

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

From the truth table, we can then read the five minterms of the function to be 1, 4, 5, 6, and 7.

Minterm/Maxterm Form & Conversion

Product of Maxterms:

To express a Boolean function as a product of maxterms,

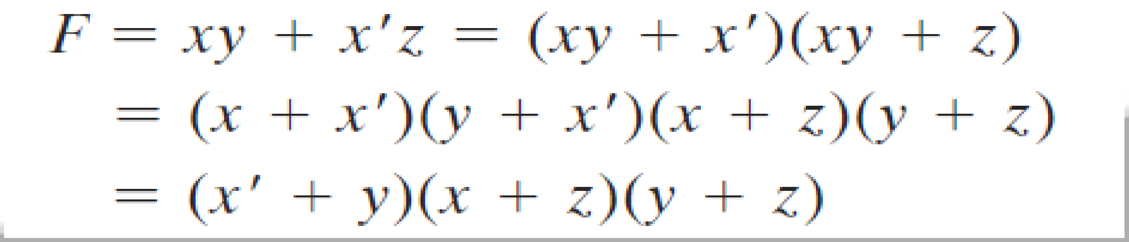
- it must first be brought into a form of OR terms.
- This may be done by using the distributive law,
- x + y z = (x + y)(x + z). Then any missing variable x in each OR term is ORed with xx′

Example:

Express the Boolean function F = x y + x′z as a product of maxterms.

First, convert the function into OR terms by using the distributive law: Law used is a + b c = (a + b)(a + c)

1st step: a=x y b = x′ c = z; 2nd step: a = x′& z b = x c = y (contd ..)



$$\begin{aligned} F = xy + x'z &= (xy + x')(xy + z) \\ &= (x + x')(y + x')(x + z)(y + z) \\ &= (x' + y)(x + z)(y + z) \end{aligned}$$

Minterm/Maxterm Form & Conversion

The function, F, has three variables: x, y, and z. Each OR term is missing one variable;

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

Therefore we add

z z′ to first factor ; y y′ to second factor ; x x′ to third factor & use the law a +b c = (a + b)(a + c)

$(x' + y + z z') = (x' + y + z)(x' + y + z'); a = x' + y , b = z, c = z'$

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

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

Combining all the terms and removing those which appear more than once, we finally obtain

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

Next we find input combinations for which F = 0 (contd…)

Minterm/Maxterm Form & Conversion

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

For F to attain a ‘0’ value,

first term : $x = 1$ $y = 0$ $z = 0$; input sequence:100 (4)

second term : $x = 1$ $y = 0$ $z = 1$; input sequence: 101 (5)

third term: $x = 0$ $y = 0$ $z = 0$; input sequence: 000 (0)

fourth term : $x = 0$ $y = 1$ $z = 0$; input sequence: 010 (2)

Therefore Maxterms are : M_o M_2 M_4 M_5

$F = M_o\ M_2\ M_4\ M_5 = (x + z + y)(x + z + y')(x' + y + z)(x' + y + z')$

A convenient way to express this function is as follows:

$$F(x, y, z) = \Pi(0, 2, 4, 5)$$

The product symbol, ∏, denotes the ANDing of maxterms; the numbers are the indices of the maxterms of the function.

Conversion between Canonical Forms

Conversion between Canonical forms:

The original function is expressed by those minterms which make the function equal to 1, whereas its complement is a 1 for those minterms for which the function is a 0.

Therefore,

The complement of a function expressed as the sum of minterms equals the sum of minterms missing from the original function.

As an example, consider the function

$$F(A, B, C) = \Sigma(1, 4, 5, 6, 7)$$

This function has a complement that can be expressed as

$$F'(A, B, C) = \Sigma(0, 2, 3) = m_0 + m_2 + m_3$$

Now, if we take the complement of F′ by De Morgan’s theorem, we obtain F in a different form:

Conversion between Canonical Forms

We obtain function F as:

$$F = (m_0 + m_2 + m_3)' = m_0' \cdot m_2' \cdot m_3' = M_0 M_2 M_3 = \Pi(0, 2, 3)$$

where m_o m_2 m_3 are the product terms (minterms) & therefore M_o M_2 M_3 (maxterms) are sum terms.

Therefore we can say that

$$m_j' = M_j$$

That is, the maxterm with subscript ‘j ‘is a complement of the minterm with the same subscript ‘j ‘and vice versa.

General conversion procedure:

To convert from one canonical form to another, interchange the symbols and list those numbers missing from the original form.

In order to find the missing terms, one must realize that the total number of minterms or maxterms is 2n, where n is the number of binary variables in the function.

Conversion between Canonical Forms

Conversion of a Boolean algebraic expression to a product of maxterms:

This is done by means of a truth table and the canonical conversion procedure.

Example:

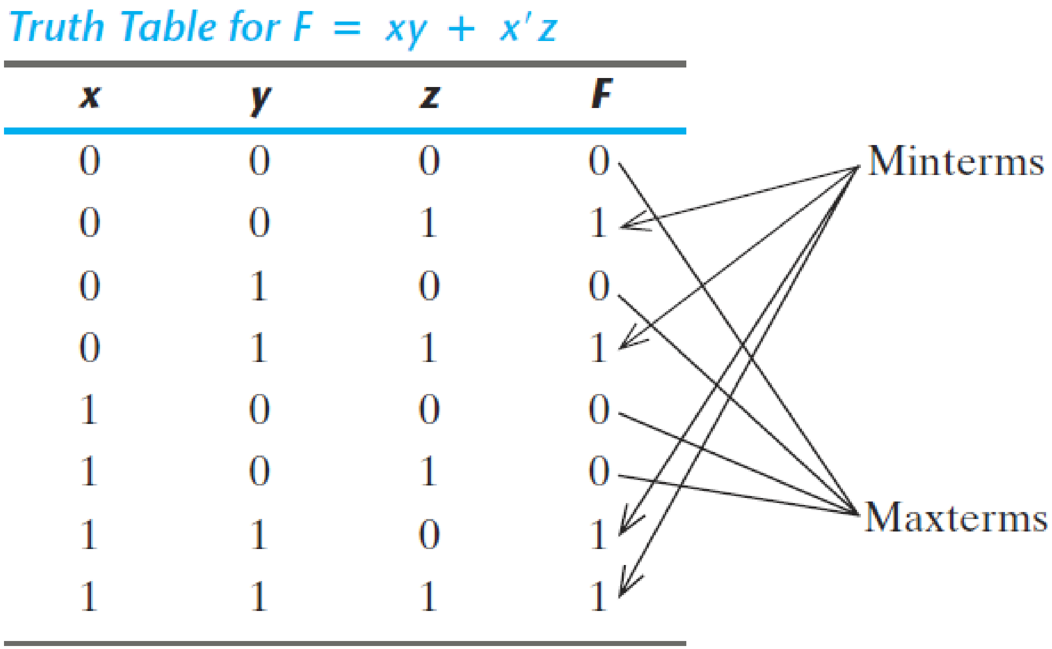
Consider the Boolean expression $F = x\,y + x'z$

Procedure:

- Derive the truth table of the function
- The 1’s under F in the table are determined from the combination of the variables for which x y = 11 or x z = 01.
- The minterms of the function are read from the truth table
- Express function as sum of minterms

Conversion between Canonical Forms

1. Truth table for the given function:



Truth Table for $F = xy + x'z$

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

Minterms
Maxterms

2. Minterms (F = 1) are read from the table to be 1, 3, 6, & 7
3. Express F as a sum of minterms, as
4. $$F(x, y, z) = \Sigma(1, 3, 6, 7)$$

(contd..)

Conversion between Canonical Forms

- determine the missing terms in the expression for F
- the missing terms are 0, 2, 4 & 5
- express the function as product of maxterms
- the function is

$$F(x, y, z) = \Pi(0, 2, 4, 5)$$

Realization of Standard Forms

We have seen that the canonical forms of Boolean algebra are obtained from the truth table.

These forms generally do not contain the least number of literals, because each minterm or maxterm contains, by definition, all the variables, either complemented or un-complemented.

Standard Form:

In this configuration the terms that form the function may contain one, two, or any number of literals.

There are two types of standard forms:

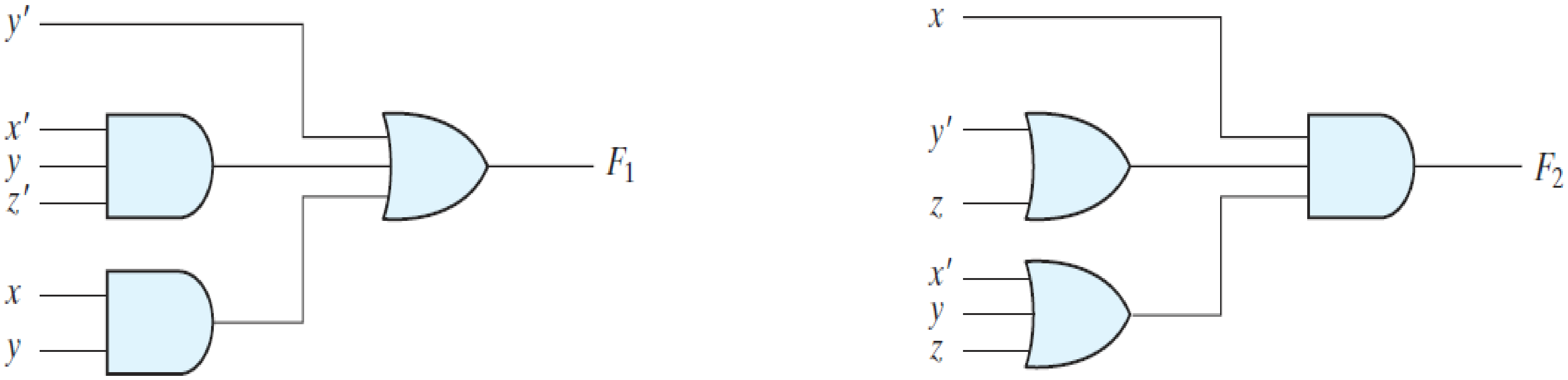
- the sum of products $F_1 = y' + xy + x'yz'$
- products of sums. $F_2 = x(y' + z)(x' + y + z')$

Realization of Standard Forms

Two Level Implementation:

In this implementation,

It is assumed that the input variables are directly available in their complements, so inverters are not included in the diagram.



(a) Sum of Products

(b) Product of Sums

Realization of Standard Forms

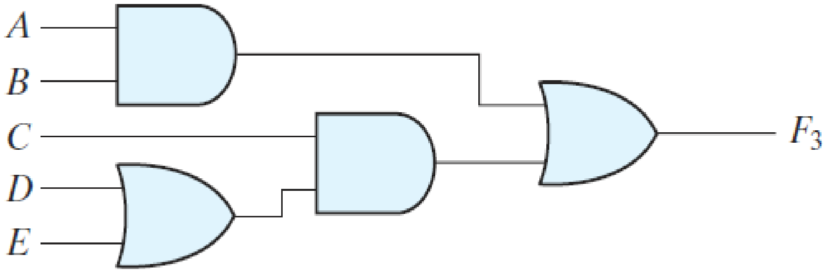
A Boolean function may be expressed in a nonstandard form.

For example, the function $F_3 = AB + C(D + E)$ is neither in sum-of-products nor in product-of-sums form.

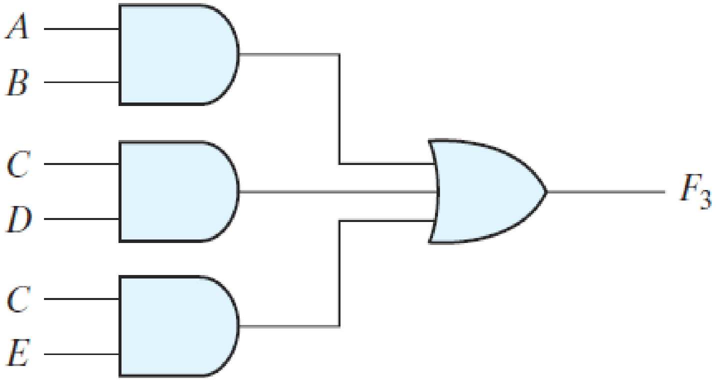
It can be changed to a standard form by using the distributive law to remove the parentheses:

$$F3 = AB + C(D + E) = AB + CD + CE$$

Implementation: Two & Three Level



(a) $AB + C(D + E)$



(b) $AB + CD + CE$

Realization of Standard Forms

In general, a two-level implementation is preferred because it produces the least amount of delay through the gates when the signal propagates from the inputs to the output. However, the number of inputs to a given gate might not be practical.

Positive & Negative Logic

- The binary signal at the inputs and outputs of any gate has one of two values, except during transition.
- One signal value represents logic ‘1’ and the other logic ‘0’.

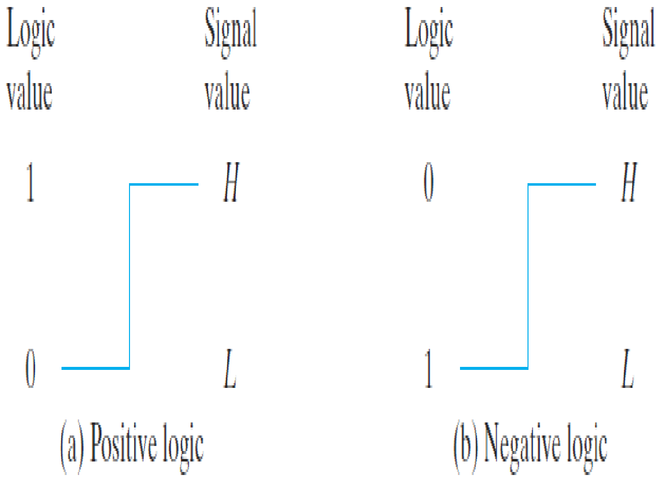
Since two signal values are assigned to two logic values, there exist two different assignments of signal level to logic value as shown.

The higher signal level is designated by H and the lower signal level by L.

Choosing the high-level H to represent logic 1 defines a positive logic system.

Choosing the low-level L to represent logic 1 defines a negative logic system.

The terms positive and negative are somewhat misleading, since both signals may be positive or both may be negative.



(a) Positive logic

(b) Negative logic

It is not the actual values of the signals that determine the type of logic, but rather the assignment of logic values to the relative amplitudes of the two signal levels.

Positive & Negative Logic

Hardware digital gates are defined in terms of signal values such as H and L. It is up to the user to decide on a positive or negative logic polarity.

Example:

Suppose we are given a digital gate with associated truth table as shown.

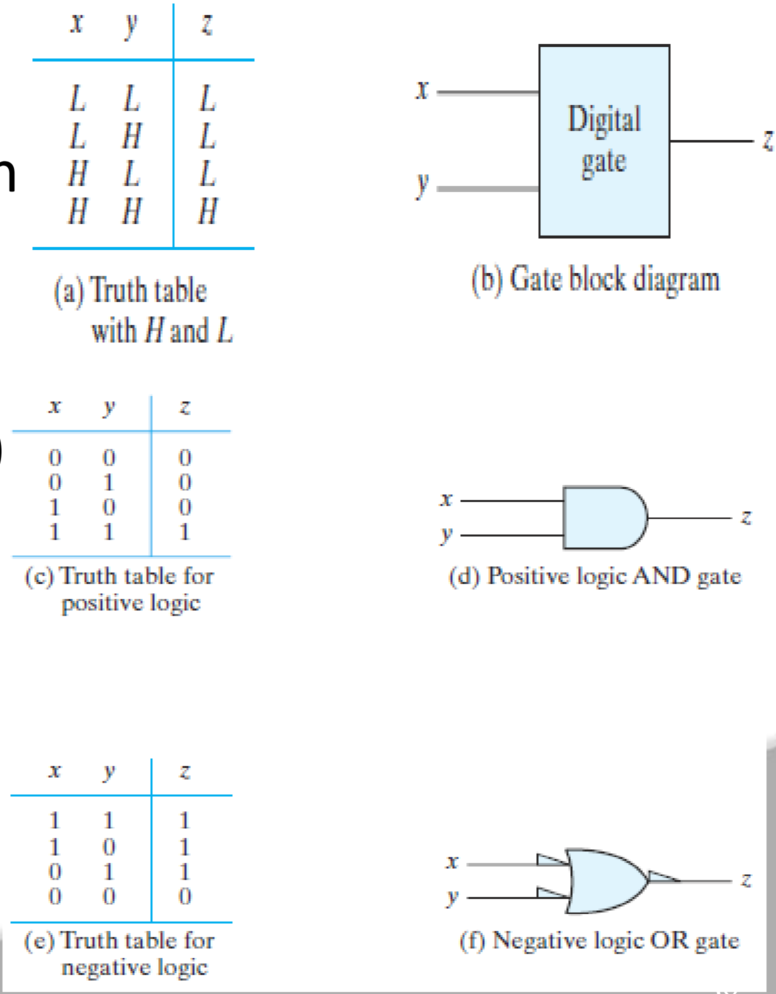
If we associate ‘0’ with ‘L’ & ‘1’ with ‘H’

It becomes an AND Gate (positive logic)

If we associate ‘1’ with ‘L’ & ‘0’ with ‘H’

It becomes OR Gate (negative logic).

Thus, the same physical gate can operate either as a positive-logic AND gate or as a negative-logic OR gate.



x	*y*	*z*
L	*L*	*L*
L	*H*	*L*
H	*L*	*L*
H	*H*	*H*

(a) Truth table with *H* and *L*

(b) Gate block diagram

x	*y*	*z*
0	0	0
0	1	0
1	0	0
1	1	1

(c) Truth table for positive logic

(d) Positive logic AND gate

x	*y*	*z*
1	1	1
1	0	1
0	1	1
0	0	0

(e) Truth table for negative logic

(f) Negative logic OR gate

Positive & Negative Logic

- The conversion from positive logic to negative logic and vice versa is essentially an operation that changes 1’s to 0’s and 0’s to 1’s in both the inputs and the output of a gate.
- Since this operation produces the dual of a function, the change of all terminals from one polarity to the other results in taking the dual of the function.
- The upshot is that all AND operations are converted to OR operations (or graphic symbols) and vice versa.
- In addition, one must not forget to include the polarity-indicator triangle in the graphic symbols when negative logic is assumed.

UNIT - II
MSI DEVICES

Introduction

Gate-level minimization:

It is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit.

Logic gates (of different types) combine in a certain way to synthesize a Boolean function.

The complexity of the digital circuit is directly related to the complexity of the algebraic expression that defines a given function.

The truth table representation of a function is unique, but the function itself is representable in many different, but equivalent, forms.

For minimizing the number of gates employed to synthesize a function, the function representation is simplified to the extent possible.

Simplification Methods for Boolean Functions

We will discuss two methods which are used for simplification of Boolean functions. Before synthesizing a given function with gates, it is simplified to minimize requirement of number of gates.

The two methods are:

Karnaugh Map Method , &

Quine- Mc Clusky Method

Karnaugh Method (K Method):

- A K-map is a diagram made up of squares,
- with each square representing one minterm of the function that is to be minimized
- any Boolean function can be expressed as a sum of minterms
- K-map is filled with the minterms

Simplification Methods for Boolean Functions

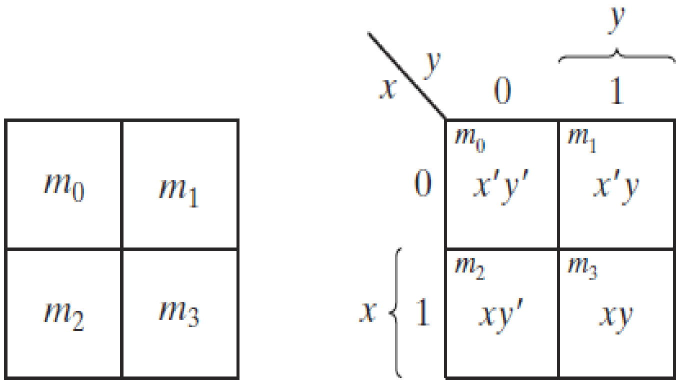
- The simplified expressions produced by the map are always in one of the two standard forms:
- sum of products or product of sums.
- simplest algebraic expression is defined as the one with a minimum number of terms, and
- with the smallest possible number of literals in each term.
- simplest expression produces a circuit diagram
- with a minimum number of gates and
- the minimum number of inputs to each gate.
- the simplest expression is not unique
- It is sometimes possible to find two or more expressions that satisfy the minimization criteria.
- In that case, either solution is satisfactory

Simplification Methods for Boolean Functions

Two-Variable K-Map:

1. only two variables x & y form Boolean function: n = 2
2. number of states: 00 01 10 11 = 2^n = 2^2
3. number of squares in K-map = number of states = 4
4. minterms: $m_0 = 00 = x' y'$ $m_1 = 01 = x' y$ $m_2 = 10 = x y'$ $m_3 = 11 = x y$

The Map:



m_0	m_1
m_2	m_3

(a)

$x \backslash y$	0	1
0	m_0 $x'y'$	m_1 $x'y$
1	m_2 xy'	m_3 xy

(b)

Whichever minterms are present in a function their corresponding squares are filled with ‘1’

Simplification Methods for Boolean Functions

K-map for AND & OR Function:

The truth tables for AND & OR functions are :

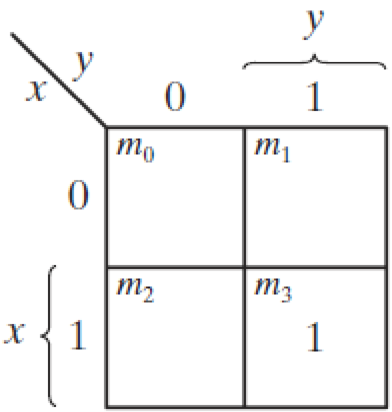
x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

We pick up such minterms for which F = 1

F = x y

AND Gate

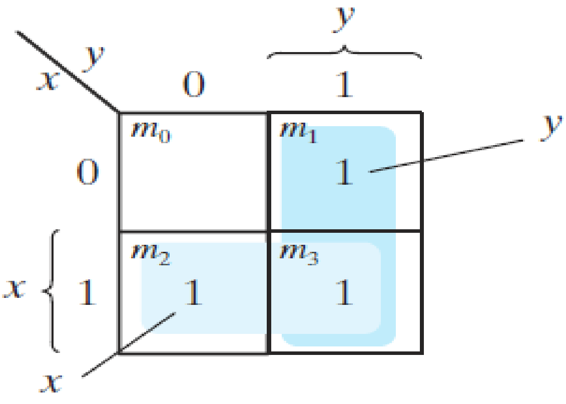


(a) xy

F = x′ y + x y′ + x y

OR Gate

F = x + y



(b) $x + y$

Simplification Methods for Boolean Functions

Importance of Simplification:

For OR gate the Boolean function is given by

$$F = x' y + x y' + x y$$

If we synthesize this function we would need

1. 3 AND Gates
2. Either 3 input OR Gate – 1 number, or

   2 input OR Gate (with 2 inputs)

We try to simplify F for achieving reduction in number of gates.

First we will try to simplify F directly as illustrated:

$$F = x' y + x y' + x y = x(y + y') + x'y = x + x'y = (x + x')(x + y)$$

$$F = x + y$$

Synthesizing this function will need 1 OR gate only. Number of gates have been reduced.

Simplification Methods for Boolean Functions

Now let us see how using K-map yields reduction in usage of gates for synthesizing function F.

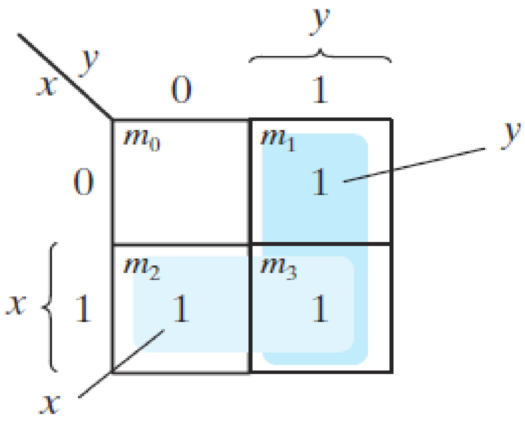
The K-map Is reproduced here:

m_2 (1 0) can be combined with m_3 (1 1)

$$x\,y' + x\,y = x$$

m_3 (1 1) can be combined with m_1 (0 1)

$$x\,y + x'y = y$$



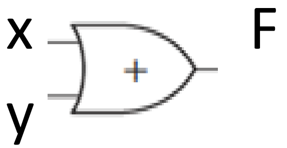
(b) $x + y$

Therefore,

$$F = x + y = x \text{ OR } y$$

Thus we need only one OR gate to synthesize F

We see that K-maps help us in achieving maximum simplification



Gate Reduction: Karnaugh Map Method

Three Variable K-Map:

1. 3 variables : x y z (n=3)
2. number of minterms = $2^3 = 8$
3. minterms: 000 001 010 011 100 101 110 111
4. minterms (in literals) : $x'y'z'(m_o)$ $x'y'z\ (m_1)$ $x'y\ z'\ (m_2)$ $x'y\ z\ (m_3)$ $x\ y'z'(m_4)$ $x\ y'z(m_5)$ $x\ y\ z'(m_6)'$ $x\ y\ z(m_7)$

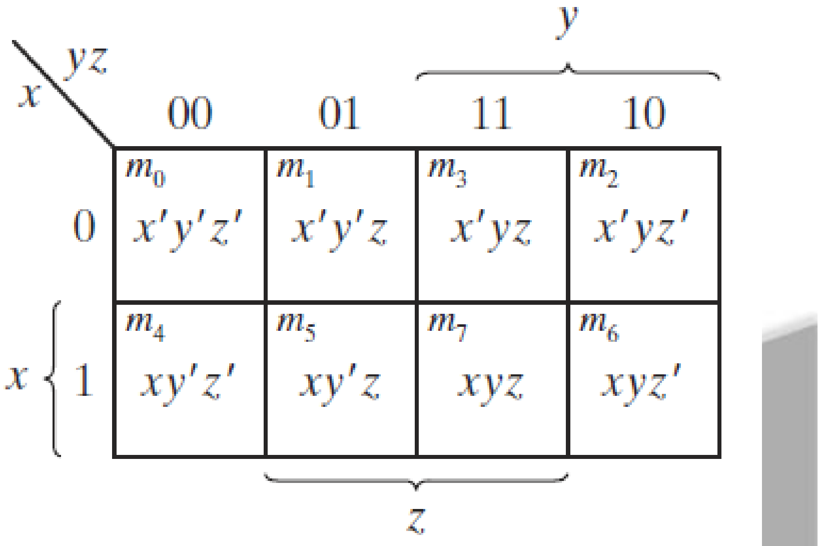
The 3 variable K-map:

It has 2 rows & 4 columns .

2 rows correspond to 2 states of x:(0 1)

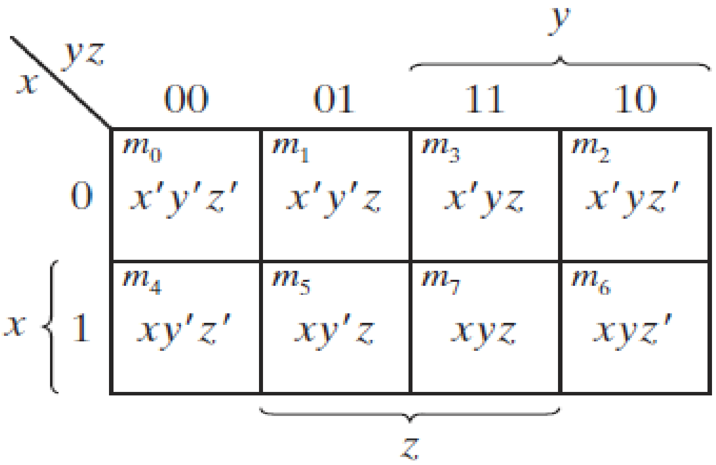
4 columns correspond to 4 states that

'y z' can assume ($2^2 = 4$)



x \ yz	00	01	11	10
0	m_0 $x'y'z'$	m_1 $x'y'z$	m_3 $x'yz$	m_2 $x'yz'$
1	m_4 $xy'z'$	m_5 $xy'z$	m_7 xyz	m_6 xyz'

Gate Reduction: Karnaugh Map Method



x \ yz	00	01	11	10
0	m_0 $x'y'z'$	m_1 $x'y'z$	m_3 $x'yz$	m_2 $x'yz'$
1	m_4 $xy'z'$	m_5 $xy'z$	m_7 xyz	m_6 xyz'

3 Variable K-Map:

If we combine 2nd row minterms we get binary variable ‘x’

If we combine minterms of 2nd & 3rd columns we get binary variable ‘z’

If we combine minterms of 3rd & 4th columns we get binary variable ‘y’

Usefulness of Maps in simplifying Boolean Functions:

The basic property possessed by adjacent squares:

- Any two adjacent squares in the map differ by only one variable, which is primed in one square and un-primed in the other.
- it follows that the sum of two minterms in adjacent squares can be simplified to a single product term consisting of only two literals.

e.g. $m_5 + m_4 = x\,y'z + x\,y'z' = x\,y'$

Gate Reduction: Karnaugh Map Method

Thus, any two minterms in adjacent squares (vertically or horizontally, but not diagonally, adjacent) that are ORed together will cause a removal of the dissimilar variable.

Example1:

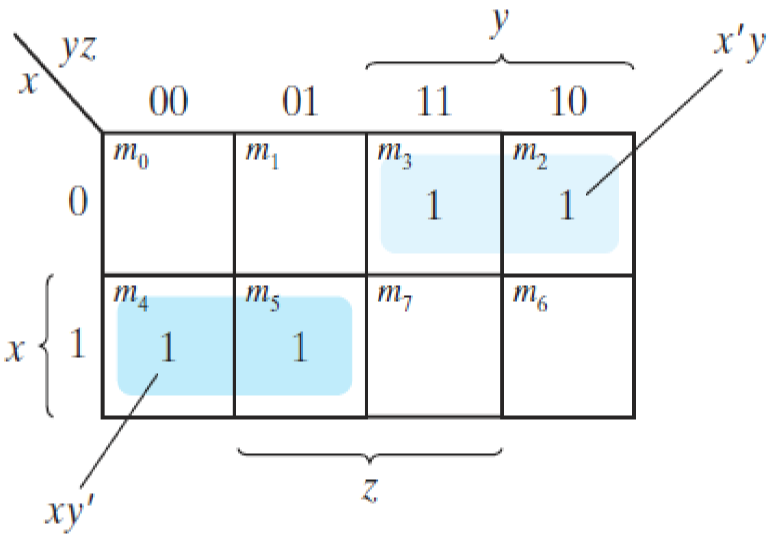
Simplify the Boolean Function $F(x, y, z) = \Sigma(2, 3, 4, 5)$

Procedure:

Mark each minterm square that represents the function with '1'

This is shown in the K-map in which the squares for minterms 010, 011, 100, and 101 are marked with 1's.

The next step is to find the possible adjacent squares. Adjacent squares are shown by shaded areas.



Gate Reduction: Karnaugh Map Method

Adding m_4 to m_5 yields $x\,y'$

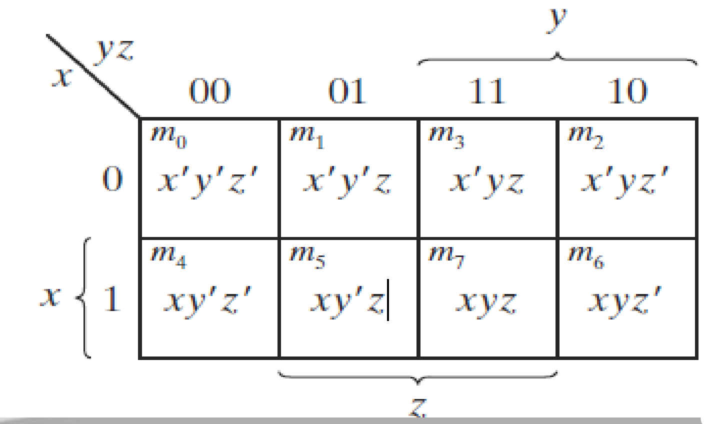
Adding m_3 to m_2 yields $x'y$

The sum of four minterms can be replaced by a sum of only two product terms.

The logical sum of these two product terms gives the simplified expression $F = x\,y' + x'y$

In certain cases, two squares in the map are considered to be adjacent even though they do not touch each other.

In the figure m_o , m_2 & m_4, m_6 are adjacent because their minterms differ by only one variable & can combined for simplification purpose



$x \backslash yz$	00	01	11	10
0	m_0 $x'y'z'$	m_1 $x'y'z$	m_3 $x'yz$	m_2 $x'yz'$
1	m_4 $xy'z'$	m_5 $xy'z$	m_7 xyz	m_6 xyz'

Gate Reduction: Karnaugh Map Method

For the Boolean function

$F = A'C + A'B + AB'C + BC$

(a) Express this function as a sum of minterms.

(b) Find the minimal sum-of-products expression.

Solution:

Sum of Minterms:

It is a canonical form in which the product terms have all the literals corresponding to the number of binary variables.

F is in standard form. Conversion to canonical form is as follows.

$A'C = A'BC + A'B'C \quad ; \; B + B' = 1$

$A'B = A'BC + A'BC' \quad ; \; C + C' = 1$

$BC = ABC + A'BC \quad ; \; A + A' = 1$ (contd ...)

Gate Reduction: Karnaugh Map Method

We write F as

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

We see $A'BC$ repeated in F. Reducible to one term as shown

$$A'BC + A'BC + A'BC = A'BC \quad \text{(OR operation)}$$

Therefore,

$F = A'BC + A'B'C + A'BC' + ABC + AB'C$ is the sum of minterms

$F = 011 \quad + \quad 001 \quad + \quad 010 \quad + 111 \quad + 101$

$F = m_3 \quad + \quad m_1 \quad + \quad m_2 \quad + \quad m_7 \quad + \quad m_5$

The function, F, can be expressed in sum-of-minterms form as

$$F(A, B, C) = \Sigma(1, 2, 3, 5, 7)$$

(continued ...)

Gate Reduction: Karnaugh Map Method

Minimal Sum of Products Expression:

We make use of K-map to obtain minimal sum of products form.

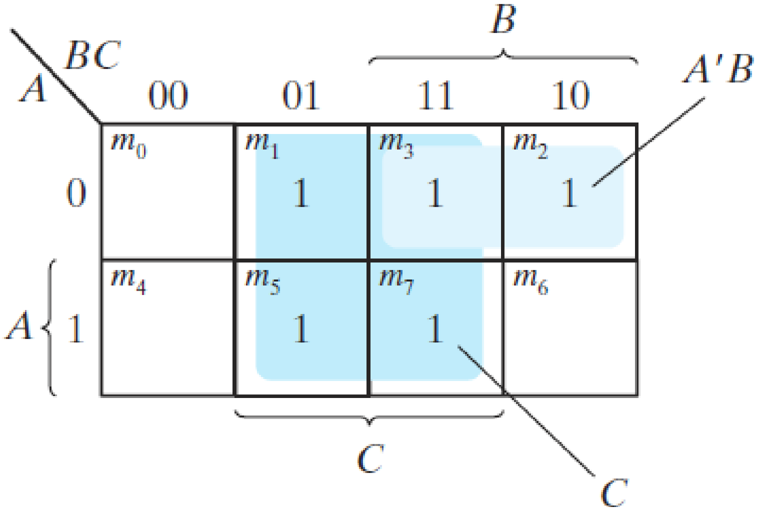
Corresponding to minterms of F fill up squares with ‘1’ as shown:

We can combine:

- m_1 with m_5 to get B′C
- m_3 with m_7 to get BC
- m_3 with m_2 to get A′B

Therefore,

F = B′C + BC + A′B = C + A′B



Gate Reduction: Karnaugh Map Method

Simplify the given function: $F(x, y, z) = (1, 2, 3, 4, 5, 7)$

Solution:

The minterms are : m_1 m_2 m_3 m_4 m_5 m_7

Fill the K-map as shown:

Combine :

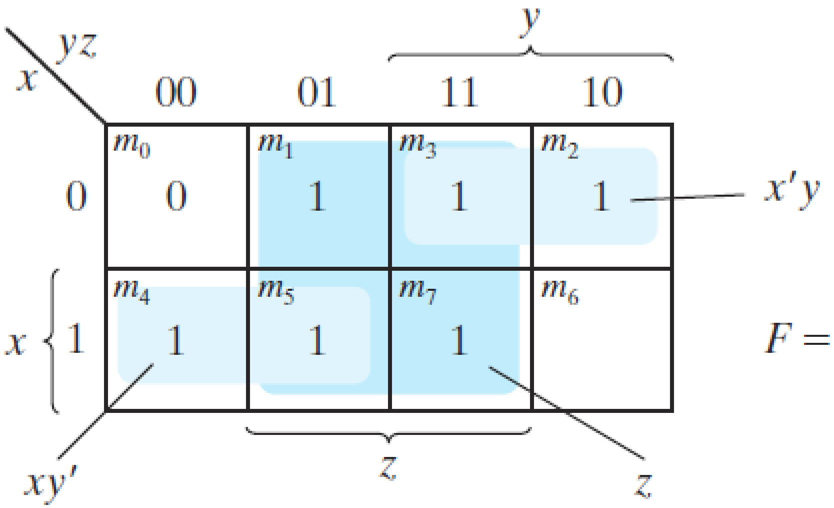
m_4 with m_5 yields x y′

m_2 with m_3 yields x′y

m_1 with m_5 & m_3 with m_7 yields z

The reduced function is:

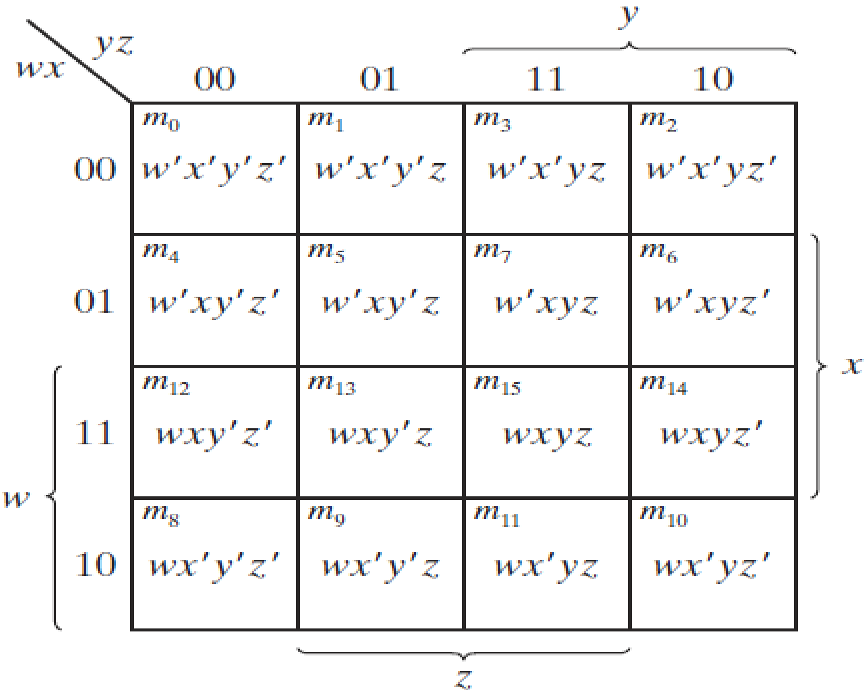
F = x y′+ x′y + z



Gate Reduction: Karnaugh Map Method

4 Variable K-Map:

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}



$wx \backslash yz$	00	01	11	10
00	m_0 $w'x'y'z'$	m_1 $w'x'y'z$	m_3 $w'x'yz$	m_2 $w'x'yz'$
01	m_4 $w'xy'z'$	m_5 $w'xy'z$	m_7 $w'xyz$	m_6 $w'xyz'$
11	m_{12} $wxy'z'$	m_{13} $wxy'z$	m_{15} $wxyz$	m_{14} $wxyz'$
10	m_8 $wx'y'z'$	m_9 $wx'y'z$	m_{11} $wx'yz$	m_{10} $wx'yz'$

Gate Reduction: Karnaugh Map Method

The combination of adjacent squares that is useful during the simplification process is easily determined from inspection of the four-variable map:

- One square represents one minterm, giving a term with four literals.
- Two adjacent squares represent a term with three literals.
- Four adjacent squares represent a term with two literals.
- Eight adjacent squares represent a term with one literal.
- Sixteen adjacent squares produce a function that is always = to‘1’.
- No other combination of squares can simplify the function.

Gate Reduction: Karnaugh Map Method

Example:

Simplify the function: $F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$

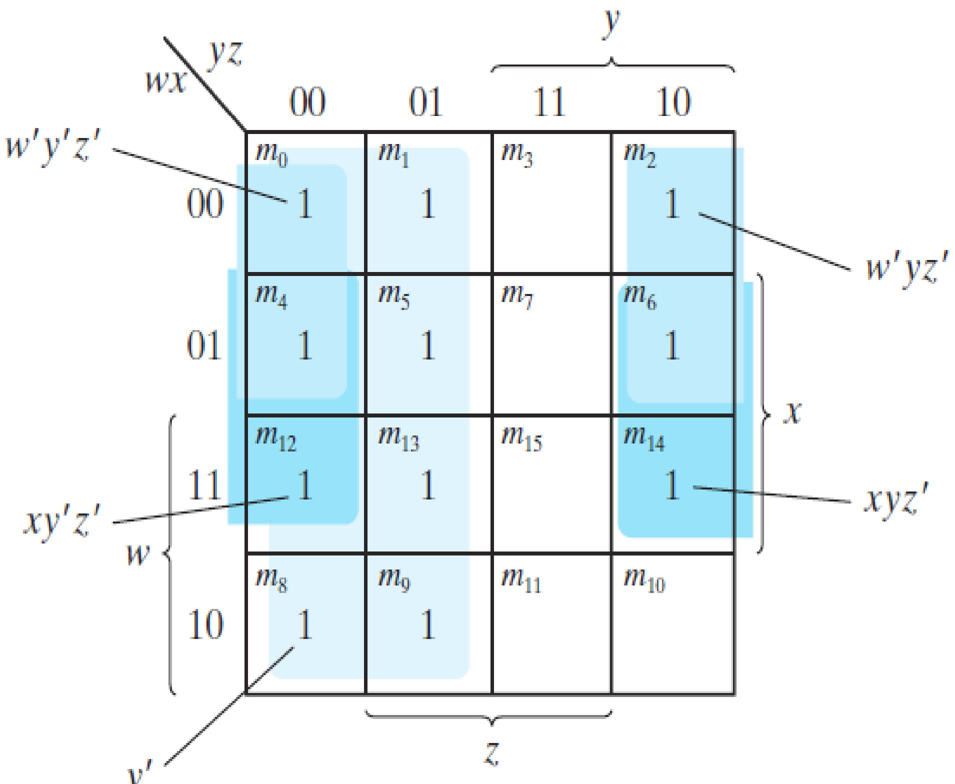
1. Combine 1st column with 2nd column to yield y′
2. Combine m_o with m_2 to yield w′x′z′
3. Combine m_4 with m_6 to yield w′x z′

   Combine 2 & 3 to get w′z′

4. Combine m_{12} with m_{14} to yield w x z′

Combine 4&3 to get x z′

The simplified F = y′+ w′z′+ x z′



wx \ yz	00	01	11	10
00	m_0 1	m_1 1	m_3	m_2 1
01	m_4 1	m_5 1	m_7	m_6 1
11	m_{12} 1	m_{13} 1	m_{15}	m_{14} 1
10	m_8 1	m_9 1	m_{11}	m_{10}

Note: $w'y'z' + w'yz' = w'z'$

$xy'z' + xyz' = xz'$

Gate Reduction: Karnaugh Map Method

Simplify the Boolean function $F = A'B'C' + B'CD' + A'BCD' + AB'C'$

F is in standard form

Convert in to canonical form .

Therefore,

1st term: 0000+0001

i.e. m_o & m_1

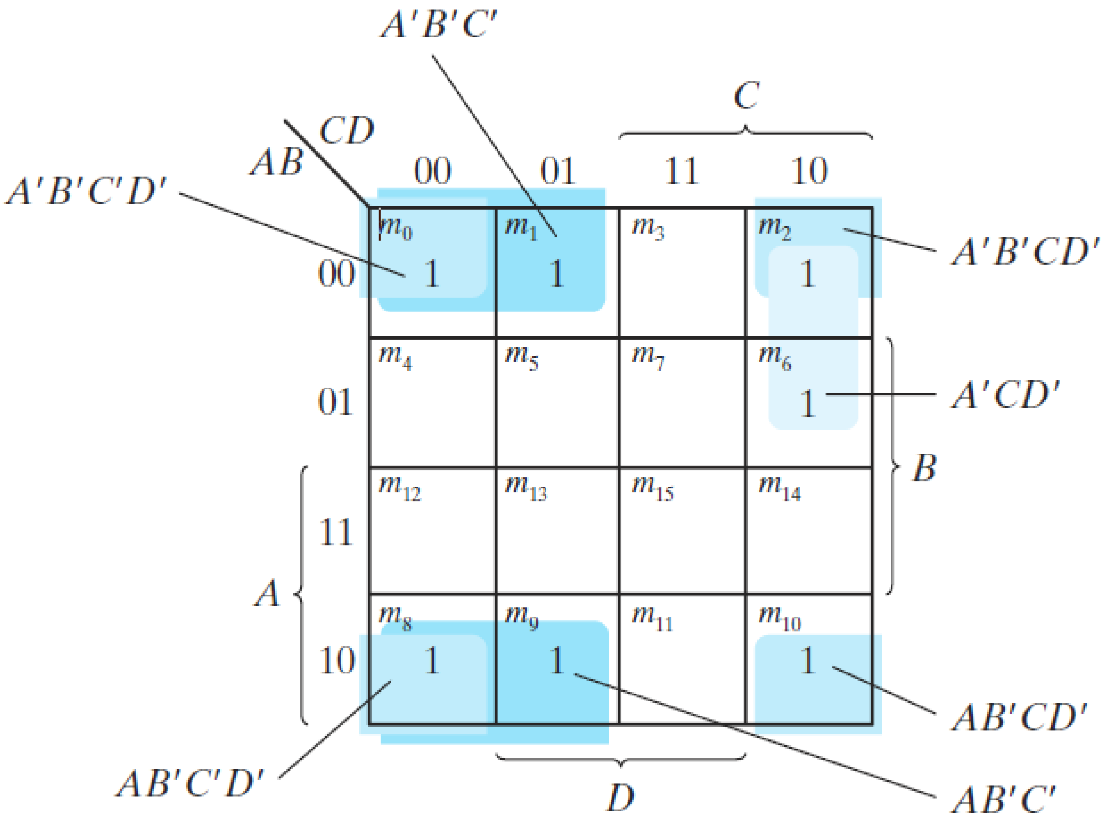
2nd term: 0010+1010

i.e. m_2 & m_{16}

4th term:1000 + 1001

i.e. m_8 & m_9

3rd term: 0110: m_6



Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$
$A'B'C' + AB'C' = B'C'$

Gate Reduction: Karnaugh Map Method

Step 1: combine m_0 with m_1 to yield $A'B'C'$

Step 2 : combine m_8 with m_9 to yield $AB'C'$

Step 3: combine 2 & 3 to yield $B'C'$

Step 4: combine m_0 with m_2 to yield $A'B'D'$

Step 5: combine m_8 with m_{10} to yield $AB'D'$

Step 6: combine 4 & 5 to yield $B'D'$

Step 7 : combine m_6 with m_2 to yield $A'C D'$

Therefore the simplified F is given by

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

Gate Reduction: Karnaugh Map Method

Prime Implicants:

- In choosing adjacent squares in a map, we must ensure that
- all the minterms of the function are covered when we combine the squares,
- the number of terms in the expression is minimized, and
- there are no redundant terms (i.e., minterms already covered by other terms)

A prime implicant is a product term obtained by combining the maximum possible number of adjacent squares in the map.

If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be essential.

The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares.

Gate Reduction: Karnaugh Map Method

- The essential prime implicants are found by looking at each square marked with a ‘1’ and checking the number of prime implicants that cover it.
- The prime implicant is essential if it is the only prime implicant that covers the minterm.

Example:

Consider the following four-variable Boolean function:

$$F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$$

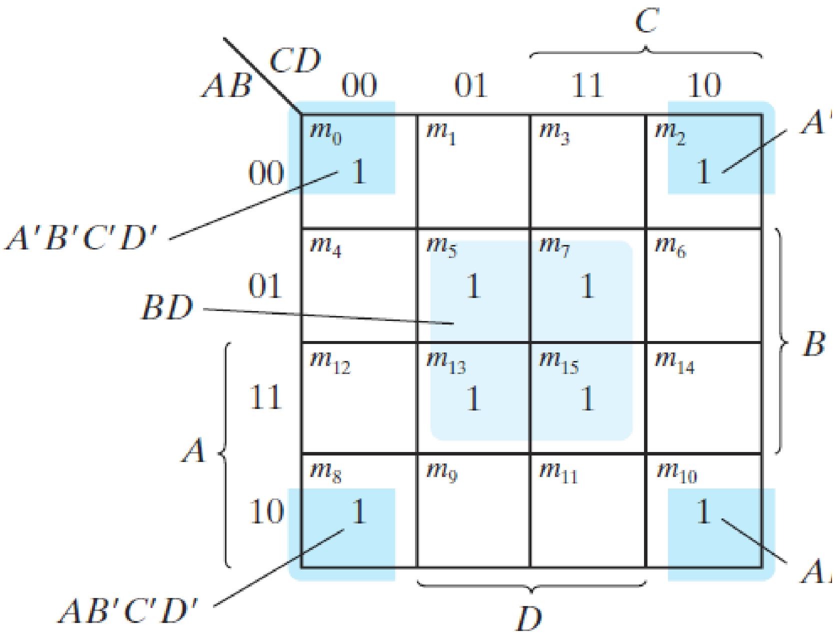
For the purpose of explaining the determination of essential prime implicants and the other prime implicants, we make 2 partial K-maps for the given F.

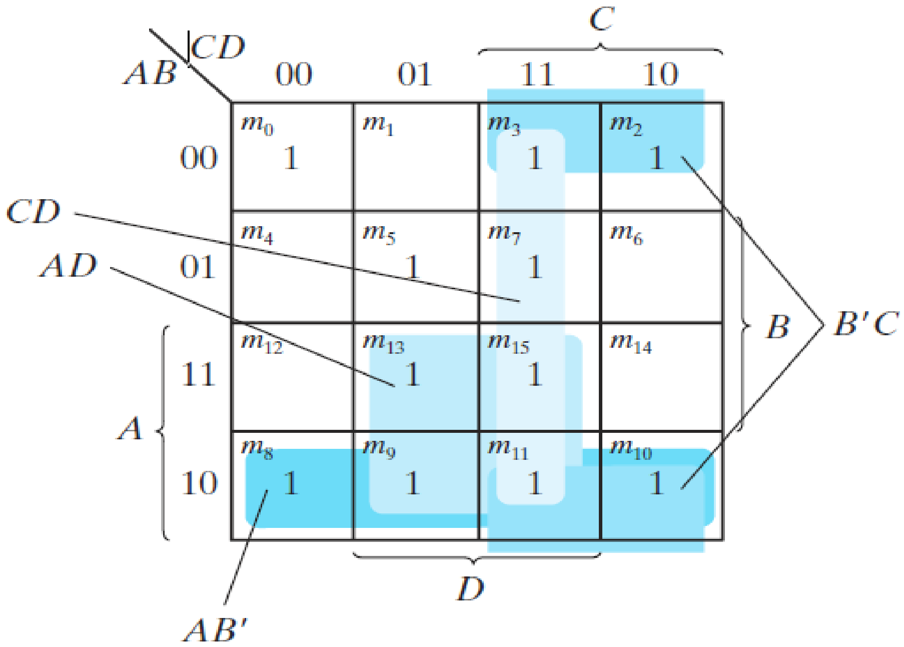
The maps are shown in the next slide.

(continued ..)

Gate Reduction: Karnaugh Map Method

$$F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$$





LHS Map:

It does not include minterms m_3 , m_9 & m_{11}

1. We observe that m_5 , m_7 , m_{13} and m_{15} can be combined to get BD

   m_5 , m_7 , m_{13} and m_{15} cannot be combined with any other square hence BD is essential prime implicant.

Gate Reduction: Karnaugh Map Method

2. m_o , m_2 , m_8 and m_{1o} can be combined to get B′D′

m_o , m_2 , m_8 and m_{1o} cannot be combined with any other square & hence B′D′ is an essential prime implicant

RHS Map:

We will now investigate in how many ways m_3 , m_9 & m_{11} can be combined with other squares.

1. We can combine m_{13} , m_{15}, m_9 & m_{11} to give AD
2. We can combine m_3 , m_2, m_{1o} & m_{11} to give B′C
3. We can combine m_3 , m_7, m_{15} & m_{11} to give CD
4. We can combine m_8 , m_9, m_{1o} & m_{11} to give AB′

Now BD & B′D′ are essential prime implicants & from the 4 listed above we have the choice (ensuring that m_3 , m_9 & m_{11} are included)

Gate Reduction: Karnaugh Map Method

Therefore, we have the following combinations:

$F = BD + B'D' + AD + B'C$

$F = BD + B'D' + AD + CD$

$F = BD + B'D' + AB' + CD$

$F = BD + B'D' + AB' + B'C$

Thus we see there may not be a unique minimum function and therefore we are afforded a choice of F for implementation with gates.

Gate Reduction: Karnaugh Map Method

Product of Sums Simplification:

Thus far we obtained minimized Boolean functions in sum-of-products form.

We will see how to obtain product-of-sums form for the minimized function.

Procedure:

The 1’s placed in the squares of the map represent the minterms of the function.

The minterms not included in the standard sum-of-products form of a function denote the complement of the function.

Therefore, the complement of a function is represented in the map by the squares not marked by 1’s.

(continued …)

Gate Reduction: Karnaugh Map Method

To obtain complement of the function , F, we mark empty squares in the K-map by 0’s.

Complement of the function, F′, obtained is in sum of products form.

To obtain the product of sums form, we take complement of F′ to get back F.

Example:

Simplify the following Boolean function into

(a) sum-of-products form, and

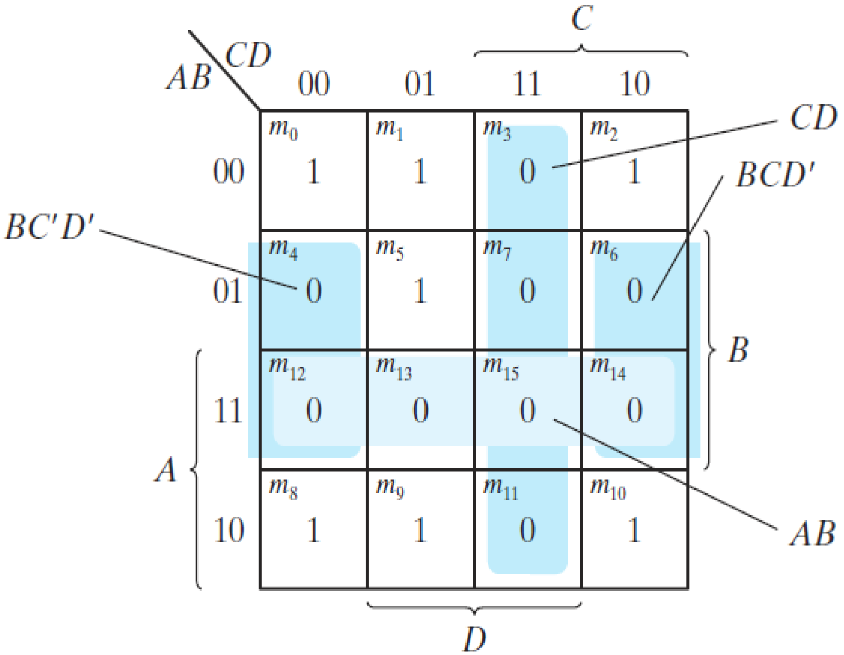
(b) product-of-sums form

$$F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10)$$

Gate Reduction: Karnaugh Map Method

The K-map filled with 1's & 0's is shown:

1. The 1's marked in the map represent all the minterms of the function.
2. The squares marked with 0's represent the minterms not included in F and therefore denote the complement of F .



AB \ CD	00	01	11	10
00	m_0 1	m_1 1	m_3 0	m_2 1
01	m_4 0	m_5 1	m_7 0	m_6 0
11	m_{12} 0	m_{13} 0	m_{15} 0	m_{14} 0
10	m_8 1	m_9 1	m_{11} 0	m_{10} 1

3. Combining the squares with 1's gives the simplified function in sum-of-products form: $F = B'D' + B'C' + A'C'D$
4. If the squares marked with 0's are combined, we get simplified complemented function as : $F' = AB + CD + BD'$

Gate Reduction: Karnaugh Map Method

In 3 we have $F = B'D' + B'C' + A'C'D$

Realization:

m_o m_2 m_8 m_{1o} are adjacent to each other & hence can be combined to yield: A′B′D′ + A B′D′ = B′D′

m_1 m_5 can be combined to yield: A′C′D

m_o m_1 m_8 m_9 are adjacent to each other & hence can be combined to yield: A′B′C′ + A B′C′ = B′C′

Thus all the minterms have been considered.

In 4 we have $F' = AB + CD + BD'$

Realization:

m_3 m_7 m_{11} m_{15} are adjacent to each other & hence can be combined to yield: A′C D + A C D = CD

(continued …)

Gate Reduction: Karnaugh Map Method

m_{12} m_{13} m_{14} m_{15} are adjacent to each other & hence can be combined to yield: ABC′ + ABC = AB

m_4 m_{12} m_6 m_{14} are adjacent to each other & hence can be combined to yield: BC′D′ + BCD′ = BD′

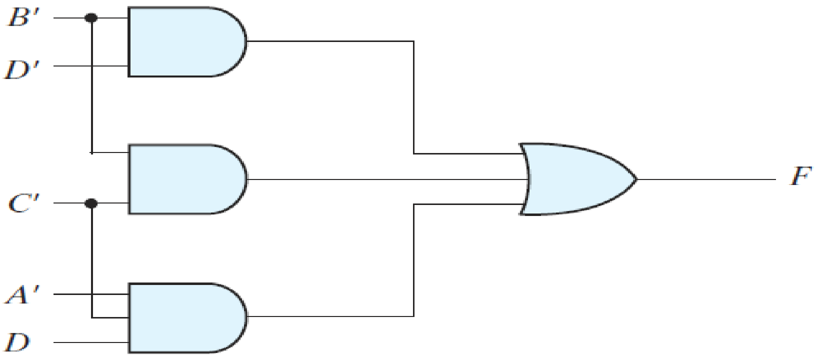
Thus all the minterms have been considered.

Now, we have

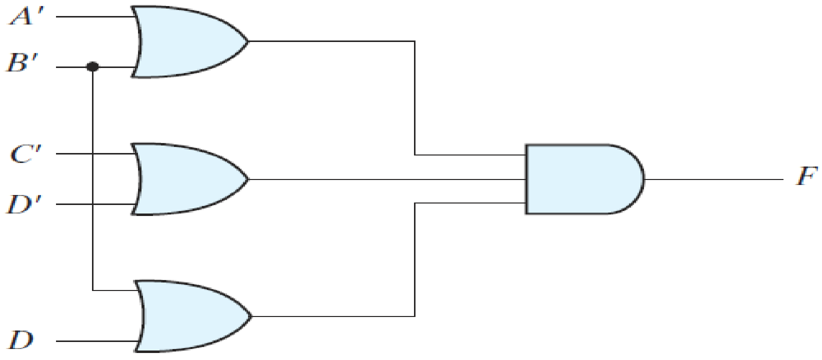
$$F' = AB + CD + BD'$$

To get the product of sum form we take complement of F′

(F′)′ = F = (A′+ B′)(C′+ D′)(B′+ D)



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



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

Gate Reduction: Karnaugh Map Method

We have considered the procedure for obtaining the product of sums simplification when the function is originally expressed in the sum-of-minterms canonical form.

The procedure is also valid when the function is originally expressed in the product of maxterms canonical form.

Suppose the function is expressed as:

$$F(x, y, z) = \Pi(0, 2, 5, 7)$$

This is product of sums form $F(x, y, z) = \Sigma(1, 3, 4, 6)$

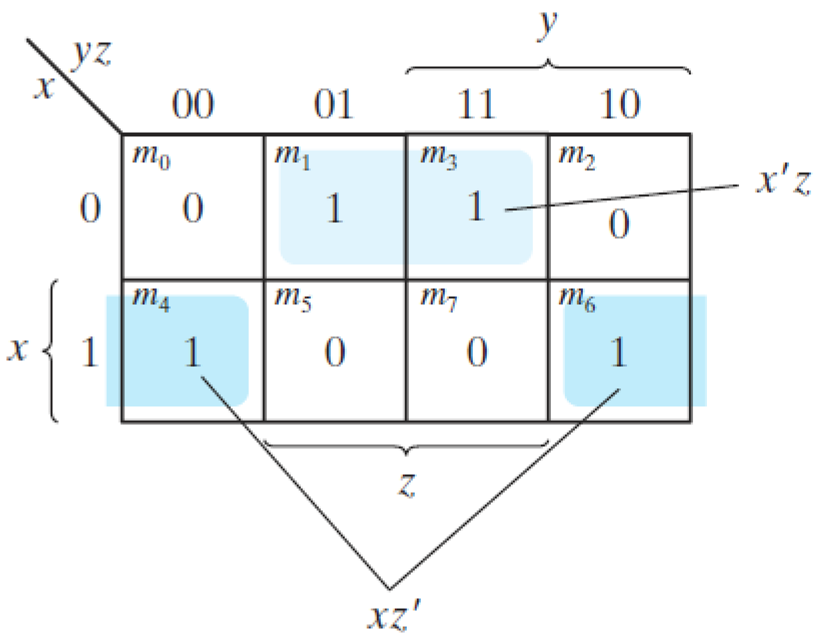
The corresponding sum of products form is

which is obtained by using missing minterms in the given product of sums form.

So we will have corresponding 1's & 0's in the K-map.

Gate Reduction: Karnaugh Map Method

The corresponding K-map is drawn as



$x \backslash yz$	00	01	11	10
0	m_0 0	m_1 1	m_3 1	m_2 0
1	m_4 1	m_5 0	m_7 0	m_6 1

For the sum of products, we combine the 1's to obtain: $F = x'z + xz'$

For the product of sums, we combine the 0's to obtain: $F' = xz + x'z'$

We take complement of F′ to get the desired product of sum form as

(F′)′ = $F = (x' + z')(x + z)$

Which when expanded yields the original sum of products form.

Don’t Care Conditions in Function Reduction

In practice, in some applications the function is not specified for certain combinations of the variables.

For example in BCD code where each decimal digit (0 to 9) is represented by 4 binary bits; out of 2^4 = 16 combinations 6 combinations (for decimal numbers between 10 to 15) are not used & hence considered to be unspecified.

Functions that have unspecified outputs for some input combinations are called incompletely specified functions .

In most applications, we simply don’t care what value is assumed by the function for the unspecified minterms.

The unspecified minterms of a function are don’t-care conditions .

These don’t-care conditions can be used on a map to provide further simplification of the Boolean expression.

Don’t Care Conditions in Function Reduction

- A don’t-care minterm is a combination of variables whose logical value is not specified.
- Such a minterm cannot be marked with a ‘1’ in the map, because it would require that the function always be a 1 for such a combination.
- Likewise, putting a ‘0’ on the square requires the function to be 0.
- To distinguish the don’t-care condition from 1’s and 0’s, an ‘X’ is used
- Thus, an ‘X’ inside a square in the map indicates that we don’t care whether the value of ‘0’ or ‘1’ is assigned to F for the particular minterm.
- In choosing adjacent squares to simplify the function in a map, the don’t-care minterms may be assumed to be either 0 or 1.
- When simplifying the function, we can choose to include each don’t-care minterm with either the 1’s or the 0’s, depending on which combination gives the simplest expression.

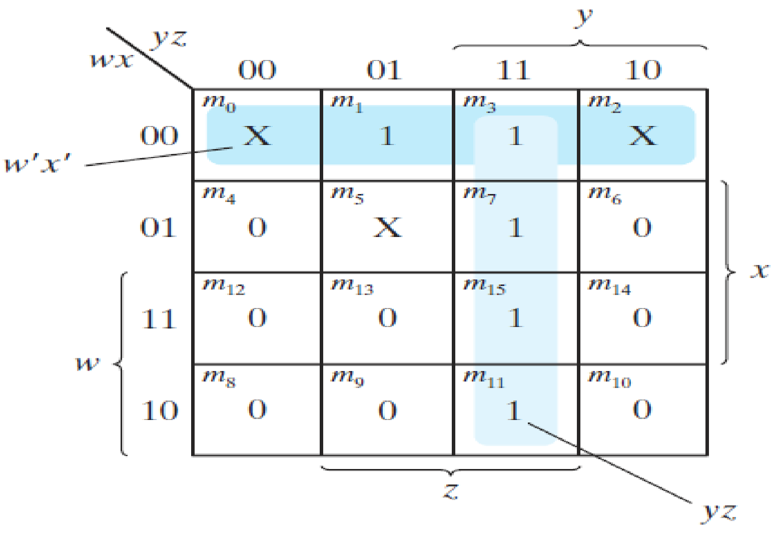
Don’t Care Conditions in Function Reduction

Example:

Simplify the Boolean function $F(w, x, y, z) = \Sigma(1, 3, 7, 11, 15)$

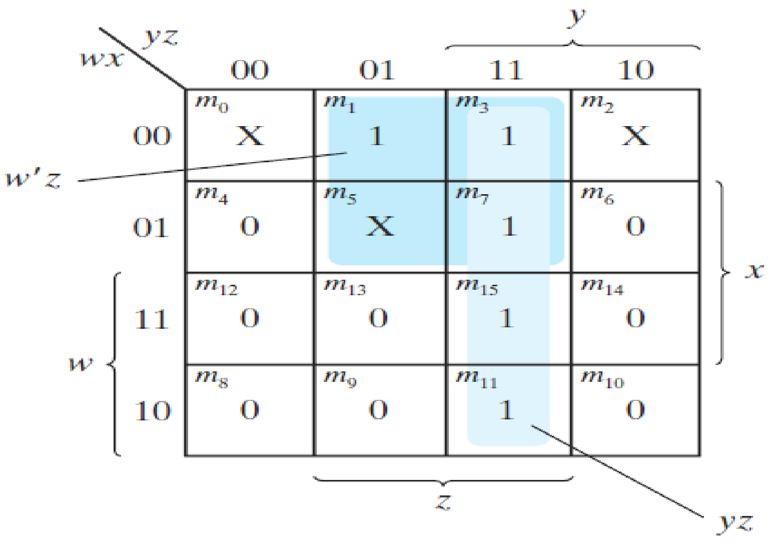
which has the don’t-care conditions $d(w, x, y, z) = \Sigma(0, 2, 5)$

The K-maps are shown for different ways of combining minterms with don’t care conditions (realizing two different minimum functions for the given Boolean function).



wx \ yz	00	01	11	10
00	m_0 X	m_1 1	m_3 1	m_2 X
01	m_4 0	m_5 X	m_7 1	m_6 0
11	m_{12} 0	m_{13} 0	m_{15} 1	m_{14} 0
10	m_8 0	m_9 0	m_{11} 1	m_{10} 0

(a) $F = yz + w'x'$



wx \ yz	00	01	11	10
00	m_0 X	m_1 1	m_3 1	m_2 X
01	m_4 0	m_5 X	m_7 1	m_6 0
11	m_{12} 0	m_{13} 0	m_{15} 1	m_{14} 0
10	m_8 0	m_9 0	m_{11} 1	m_{10} 0

(b) $F = yz + w'z$

Don’t Care Conditions in Function Reduction

Note:

The first expression includes minterms 0 and 2 with the 1’s and leaves minterm 5 with the 0’s.

The second expression includes minterm 5 with the 1’s and leaves minterms 0 and 2 with the 0’s.

The two expressions represent two functions that are not algebraically equal.

Both cover the specified minterms of the function, but each covers different don’t-care minterms.

Simplified product-of-sums expression:

Combine squares with 0’s in the map with don’t care conditions (0 & 2 can be combined). The reduced function obtained is

$F' = z' + wy'$ (F′)′ = z(w ′+y) ; F = z(w ′+y)

Quine-Mc Cluskey method

- In order to find all of the prime implicants, all possible pairs of minterms should be compared and combined whenever possible.
- To reduce the required number of comparisons, the binary minterms are sorted into groups according to the number of 1's in each term.

Given the function: $f(a, b, c, d) = \Sigma m(0, 1, 2, 5, 6, 7, 8, 9, 10, 14)$

First we make a table wherein we arrange the minterms row wise . Starting with row 1 minterms are placed in successive rows according to the number of 1's in each term.

1st row : minterm with no 1's

2nd row: minterms with 1 number of 1's

3rd row: minterms with 2 number of 1's

4th row : minterms with 3 number of 1's , & so on …

Quine-Mc Cluskey method

For the given function we have the following table:

After formulation of the table, we compare

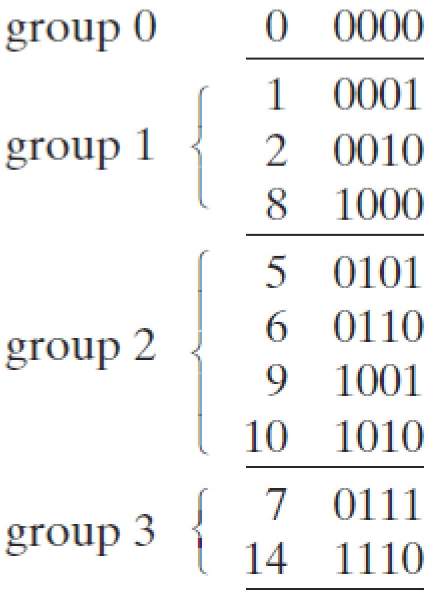
The minterm in each group with the minterms

In its adjacent group , because two terms can be combined only if they differ in 1 bit.

Therefore there is no need to compare group 0 with group 2 or 3.

Group 0 is compared only with group 1;

group 1 is compared only with group 2 & so on.



group 0	0	0000
group 1	1	0001
	2	0010
	8	1000
group 2	5	0101
	6	0110
	9	1001
	10	1010
group 3	7	0111
	14	1110

Thus,

0000 in group 0 combines with 0001 (1), 0010 (2), & 1000 (8) in group 1: to yield 000- , 00-0, -000, respectively. ‘1’ + ‘0’ = ‘-’ because $x + x' = 1$; $a'b'c'd' + a'b'c'd = a'b'c'- = 000-$

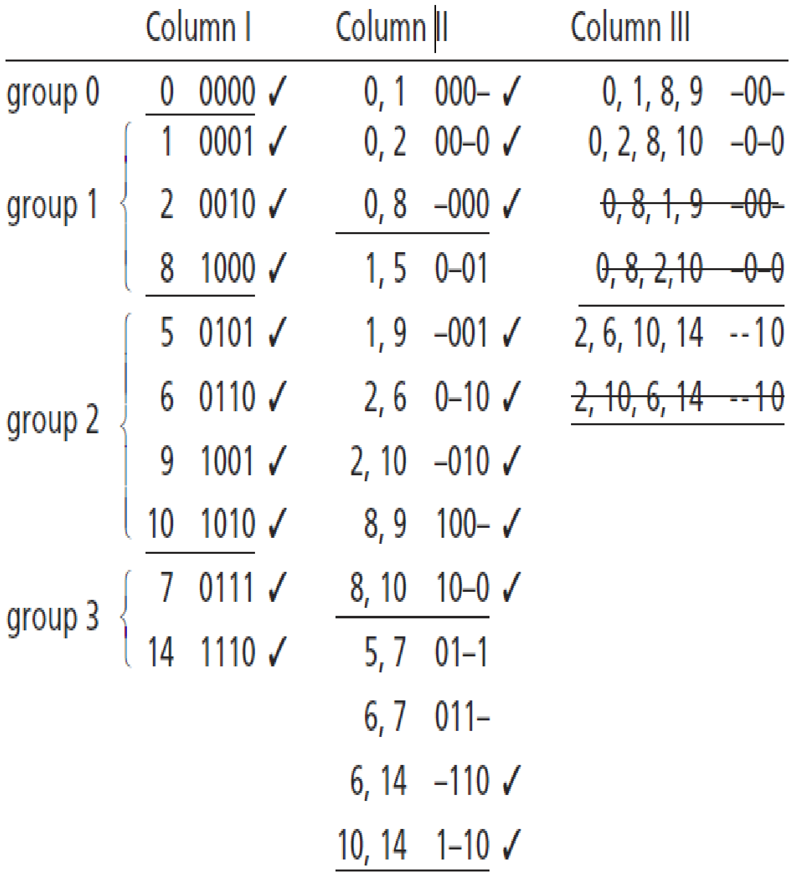
Quine-Mc Cluskey method

Following the procedure of comparing 2 adjacent groups we get the table for determining prime implicants. The table is shown below: We ensure that all the minterms are covered in process.

Column II lists the combinations as obtained by comparing terms of the adjacent groups in column I.

Column III lists the combinations as obtained by comparing terms of the adjacent groups in column II.

Duplicate combinations are not considered & hence crossed.



	Column I		Column II		Column III	
group 0	0	0000 ✓	0, 1	000– ✓	0, 1, 8, 9	-00–
group 1	1	0001 ✓	0, 2	00–0 ✓	0, 2, 8, 10	-0–0
	2	0010 ✓	0, 8	-000 ✓	~~0, 8, 1, 9~~	~~-00-~~
	8	1000 ✓	1, 5	0–01	~~0, 8, 2,10~~	~~-0-0~~
group 2	5	0101 ✓	1, 9	-001 ✓	2, 6, 10, 14	--10
	6	0110 ✓	2, 6	0–10 ✓	~~2, 10, 6, 14~~	~~--10~~
	9	1001 ✓	2, 10	-010 ✓		
	10	1010 ✓	8, 9	100– ✓		
group 3	7	0111 ✓	8, 10	10–0 ✓		
	14	1110 ✓	5, 7	01–1		
			6, 7	011–		
			6, 14	–110 ✓		
			10, 14	1–10 ✓		

Quine-Mc Cluskey method

While comparing, we tick mark the minterms that have been combined (as shown in the table).

We have the following observations:

1. that in column II the terms (1,5) (5,7) (6,7) could not be combined while forming column III, and
2. in column III no further combining of terms in its 2 groups is possible

The conclusions are:

1. The un-combined terms are prime implicants, and
2. Because further combining is not possible, the minimization process is completed.

Therefore the minimized function consists of the following terms:

(1,5), (5,7), (6,7), (0,1,8,9), (0,2,8,10) & (2,6,10,14)

Quine-Mc Cluskey method

Therefore, we write reduced Boolean function as:

f = (1,5) + (5,7) + (6,7) + (0,1,8,9) + (0,2,8,10) + (2,6,10,14)

f = 0–01 + 01–1 + 011– + –00– + –0–0 + --10

The function f was defined as f(a, b, c, d); therefore we follow the same order , i.e., a b c d to assign literals to the reduced function, f. ‘-’ implies that the literal is eliminated.

Therefore,

$$f = a'c'd + a'bd + a'bc + b'c' + b'd' + cd'$$

In this expression, each term has a minimum number of literals, but the number of terms is not minimum.

So, we make a chart called The Prime Implicant Chart.

Making this chart is 2nd step in the minimization method

Quine-Mc Cluskey method

The Prime Implicant Chart:

A prime implicant chart is made to select a minimum set of prime implicants.

The minterms of the function are listed across the top of the chart and

the prime implicants are listed down the side.

A prime implicant = sum of minterms & covers all of them

If a prime implicant covers a given minterm, an ‘X’ is placed at the intersection of the corresponding row and column.

In the Chart all of the prime implicants (terms which have not been ticked off in the table) are listed on the left.

This is shown in the next slide

(continued ....)

Quine-Mc Cluskey method

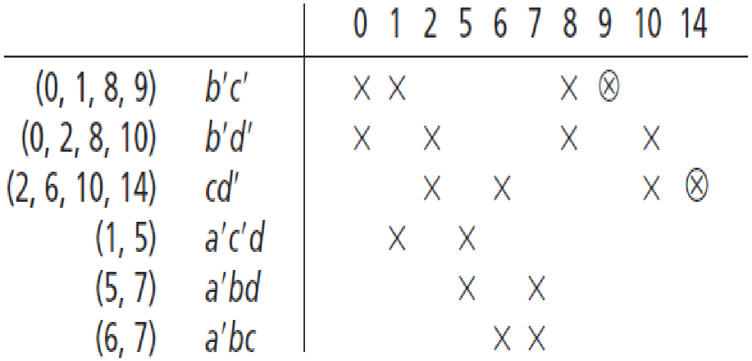
The chart is shown below:

We put a ‘X’ under the minterm contained in a prime implicant.

We have six rows corresponding to six prime implicants.

Minterms corresponding to each prime Implicant are marked ‘X’.

prime implicants minterms



		0	1	2	5	6	7	8	9	10	14
(0, 1, 8, 9)	$b'c'$	X	X					X	⊗		
(0, 2, 8, 10)	$b'd'$	X		X				X		X	
(2, 6, 10, 14)	cd'			X		X				X	⊗
(1, 5)	$a'c'd$		X		X						
(5, 7)	$a'bd$				X		X				
(6, 7)	$a'bc$					X	X				

If a minterm is covered by only one prime implicant, then that prime implicant is called an essential prime implicant and must be included in the minimum sum of products.

Minterms 9 & 14 are covered by only one prime implicant.

b′c′ for ‘9’ & cd′ for ‘14’ Therefore b′c′ & cd′are essential.

Quine-Mc Cluskey method

Procedure for picking prime implicants for minimum sum from the chart:

1. Pick up a prime implicant for inclusion in the minimum sum
2. Cross out its corresponding row (of minterms)
3. Cross out the columns corresponding to minterms of selected prime implicant
4. While picking up a prime implicant ensure that it combines maximum number of minterms.

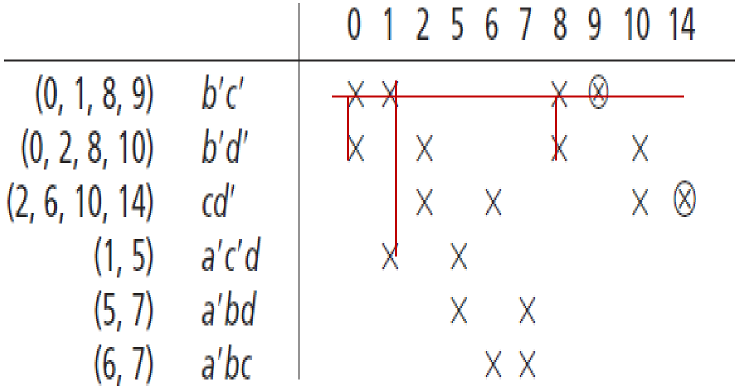
Starting with b′c′:

Cross out the row (as shown)

Minterm ‘0’ & ‘8’ also appear in row 2

Cross out columns corresponding to

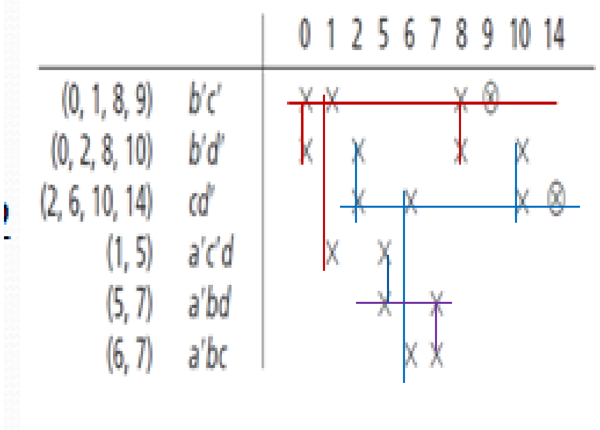
minterms ‘0’ & ‘8’, as shown.



		0	1	2	5	6	7	8	9	10	14
(0, 1, 8, 9)	b′c′	X	X					X	⊗		
(0, 2, 8, 10)	b′d′	X		X				X		X	
(2, 6, 10, 14)	cd′			X		X				X	⊗
(1, 5)	a′c′d		X		X						
(5, 7)	a′bd				X		X				
(6, 7)	a′bc					X	X				

Quine-Mc Cluskey method

Now we have the chart as:



		0	1	2	5	6	7	8	9	10	14
(0, 1, 8, 9)	b′c′	X	X					X	⊗		
(0, 2, 8, 10)	b′d′	X		X				X		X	
(2, 6, 10, 14)	cd′			X		X				X	⊗
(1, 5)	a′c′d		X		X						
(5, 7)	a′bd				X		X				
(6, 7)	a′bc					X	X				

Now we need to pick up a prime implicant between b′d′ & cd′.

We see that picking up cd′ covers more number of minterms than b′d′would do.

Therefore pick up cd′

Cross out row corresponding to it, and columns corresponding to minterms 2 & 10

Between a′c′d & a′bd

we choose a′bd

it covers more number of minterms. Cross out the row & columns.

We see that all the minterms are covered. Therefore the function is

f = a′bd + cd′+ b′c′

Quine-Mc Cluskey Method

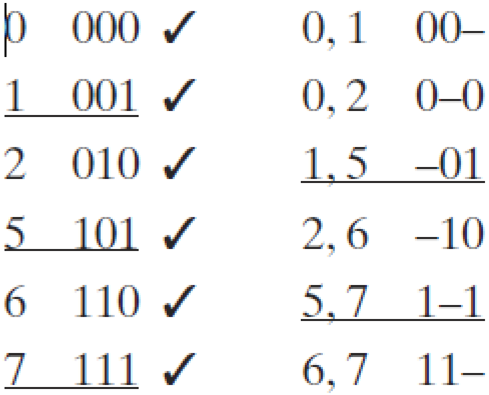
Cyclic Prime Implicant Chart:

A prime implicant chart which has two or more X’s in every column is called a cyclic prime implicant chart.

Example:

Consider the function: $F = \Sigma\, m(0, 1, 2, 5, 6, 7)$

Derivation of Prime Implicants:



0	000	✓	0, 1	00–
1	001	✓	0, 2	0–0
2	010	✓	1, 5	–01
5	101	✓	2, 6	–10
6	110	✓	5, 7	1–1
7	111	✓	6, 7	11–

Now we will draw the Prime Implicant Chart.

(continued ……)

Quine-Mc Cluskey Method

Prime Implicant Chart:

		Prime Implicants		Minterms					
				0	1	2	5	6	7
①	→	(0, 1)	$a'b'$	×	×				
		(0, 2)	$a'c'$	×		×			
		(1, 5)	$b'c$		×		×		
②	→	(2, 6)	bc'			×		×	
③	→	(5, 7)	ac				×		×
		(6, 7)	ab					×	×

All columns have two X’s, so we will proceed by trial and error.

Both (0, 1) and (0, 2) cover column 0, so we will try (0, 1)

After crossing out row (0, 1) and columns 0 and 1,

we examine column 2, which is covered by (0, 2) and (2, 6).

The best choice is (2, 6) because it covers two of the remaining columns while (0, 2) covers only one of the remaining columns.

After crossing out row (2, 6) and columns 2 and 6, we see that (5, 7) covers the remaining columns and completes the solution.

Therefore, one solution is: $F = a'b' + bc' + ac.$

Quine-Mc Cluskey Method

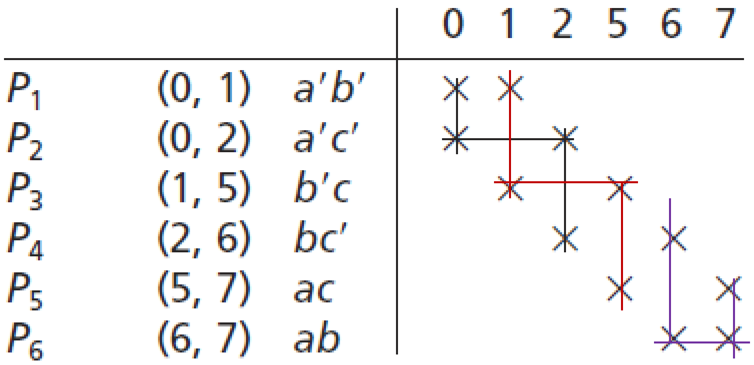
However, we are not guaranteed that this solution is minimum.

Therefore, we must go back & solve the problem over again starting with the other prime implicant that covers column 0.

Prime Implicant Chart:

Instead of (0,1) we now try (0,2)

Cross out corresponding row & columns, as shown.



			0	1	2	5	6	7
P_1	(0, 1)	$a'b'$	×	×				
P_2	(0, 2)	$a'c'$	×		×			
P_3	(1, 5)	$b'c$		×		×		
P_4	(2, 6)	bc'			×		×	
P_5	(5, 7)	ac				×		×
P_6	(6, 7)	ab					×	×

Next we pick up (1,5) & cross out corresponding row & columns, as shown.

We pick up (6,7) which covers all the remaining minterms.

Therefore the reduced function is : $F = a'c' + b'c + ab.$

Since both ways we get same number of terms & literals, we take it as the reduced function.

Quine- Mc Cluskey Method (Don’t Care Conditions)

Simplification of incompletely specified functions:

Given an incompletely specified function, the proper assignment of values to the don’t-care terms is necessary in order to obtain a minimum form for the function.

In the process of finding the prime implicants, we will treat the don’t-care terms as if they were required minterms.

When forming the prime implicant chart, the don’t-cares are not listed at the top. This way, when the prime implicant chart is solved, all of the required minterms will be covered by one of the selected prime implicants.

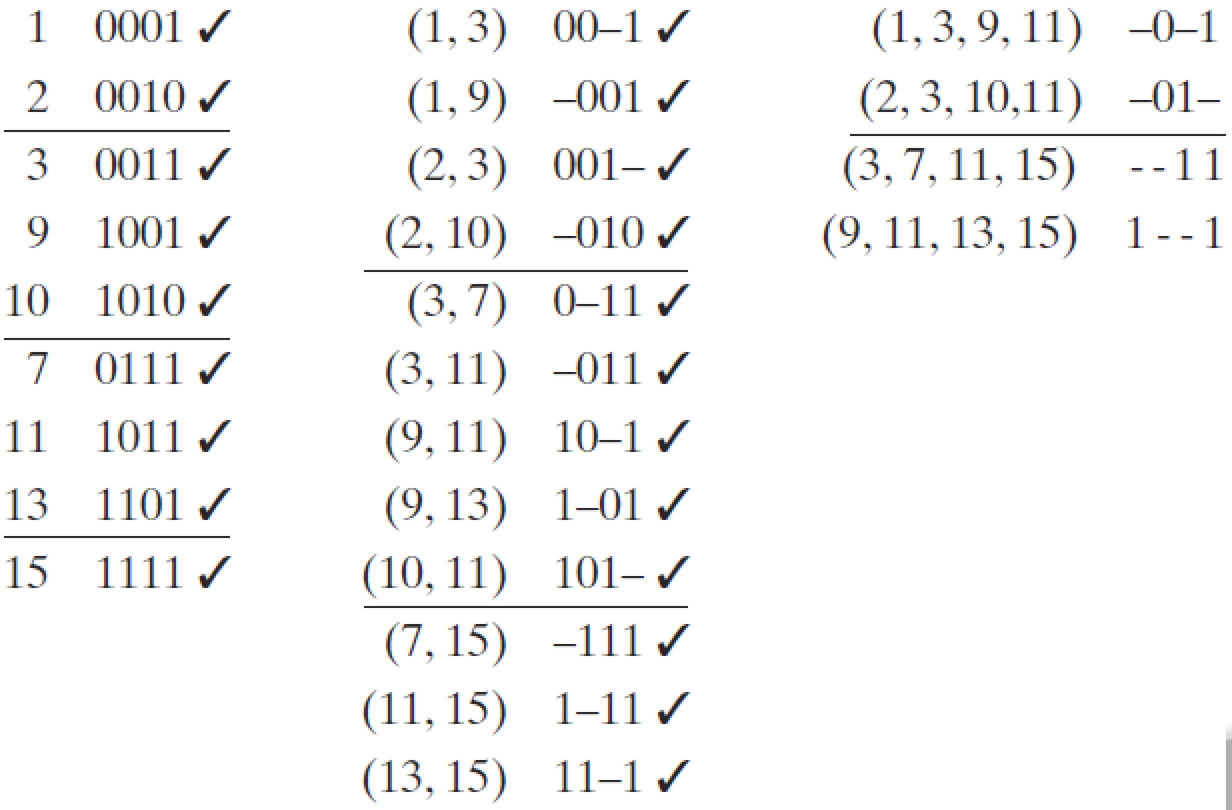
However, the don’t-care terms are not included in the final solution unless they have been used in the process of forming one of the selected prime implicants.

Quine- Mc Cluskey Method (Don’t Care Conditions)

Example: $F(A, B, C, D) = \Sigma\, m(2, 3, 7, 9, 11, 13) + \Sigma\, d(1, 10, 15)$

The second summation term corresponds to don’t care conditions.

The Table:



1	0001	✓
2	0010	✓
3	0011	✓
9	1001	✓
10	1010	✓
7	0111	✓
11	1011	✓
13	1101	✓
15	1111	✓

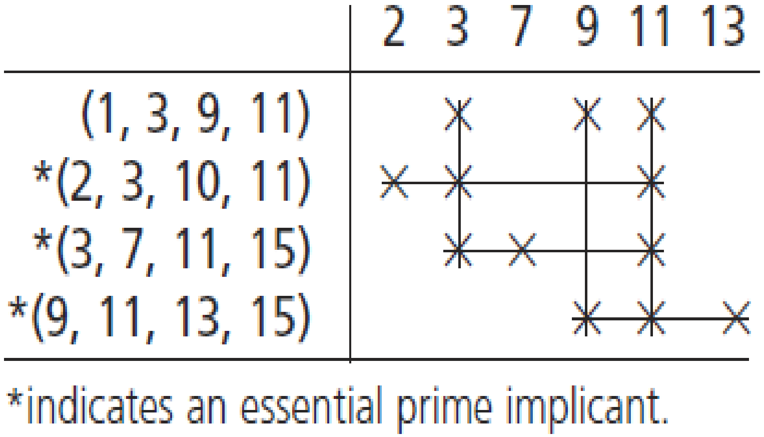
(1, 3)	00–1	✓
(1, 9)	–001	✓
(2, 3)	001–	✓
(2, 10)	–010	✓
(3, 7)	0–11	✓
(3, 11)	–011	✓
(9, 11)	10–1	✓
(9, 13)	1–01	✓
(10, 11)	101–	✓
(7, 15)	–111	✓
(11, 15)	1–11	✓
(13, 15)	11–1	✓

(1, 3, 9, 11)	–0–1
(2, 3, 10,11)	–01–
(3, 7, 11, 15)	- - 1 1
(9, 11, 13, 15)	1 - - 1

Quine- Mc Cluskey Method (Don’t Care Conditions)

Prime Implicant Chart:

Prime Implicants Minterms



	2	3	7	9	11	13
(1, 3, 9, 11)		X		X	X	
*(2, 3, 10, 11)	X	X			X	
*(3, 7, 11, 15)		X	X		X	
*(9, 11, 13, 15)				X	X	X

*indicates an essential prime implicant.

Start with (2,3,10,11): It covers minterms 2,3 & 11

Next (9,11,13,15): It covers minterms 9 & 13

Minterm 7 is left out therefore we have to pick up (3,7,11,15).

The reduced function is $F = B'C + CD + AD$

Realization with NAND & NOR Gates

Digital circuits are frequently constructed with NAND or NOR gates rather 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.

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.

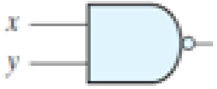
NAND Circuits:

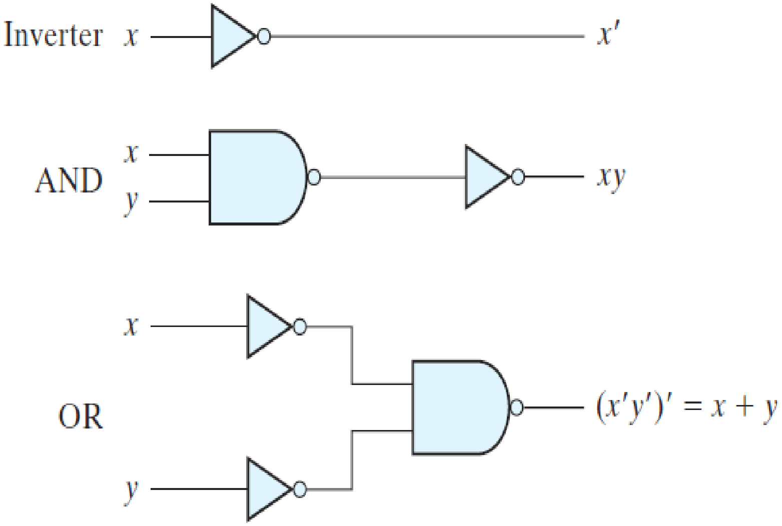
The NAND gate is called universal gate because any logic circuit can be implemented with it.

Realization with NAND & NOR Gates

Logic operations with NAND gates :

Implementation of NOT AND & OR gates is shown using NAND gate.





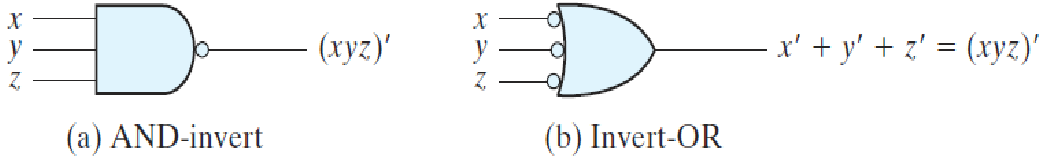
NOT / INVERTER /COMPLEMENT function is synthesized using single input NAND gate.

Conversion to NAND logic:

The conversion of an algebraic expression from AND, OR, and complement to NAND can be done by simple circuit manipulation techniques that change AND–OR diagrams to NAND diagrams.

Realization with NAND & NOR Gates

Two graphic symbols for a three-input NAND gate:



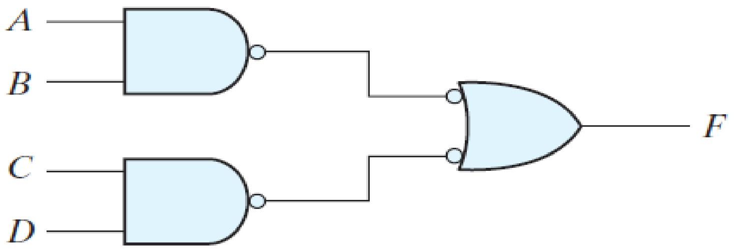
(a) AND-invert (b) Invert-OR

Two Level Implementation:

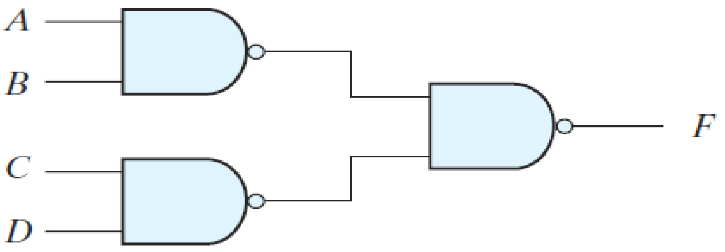
The implementation of Boolean functions with NAND gates requires that the functions be in sum-of-products form.

Example 1: Implement the function: F = AB + CD

We make use of AND-invert & Invert-OR gates to synthesize given F.



(a)



(b)

Realization with NAND & NOR Gates

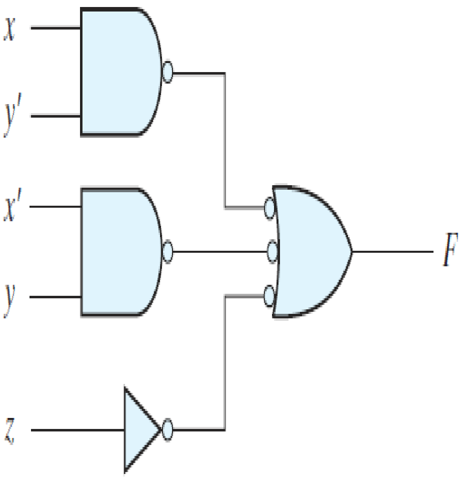
Example 2:

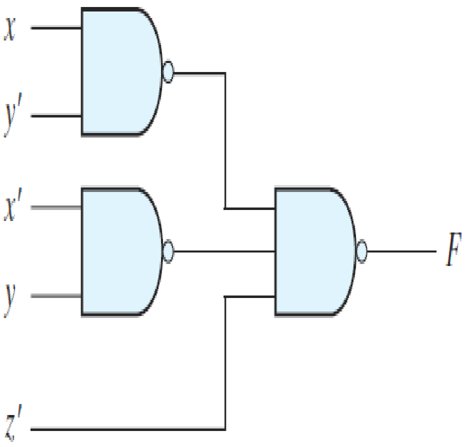
Implement the following Boolean function with NAND gates:

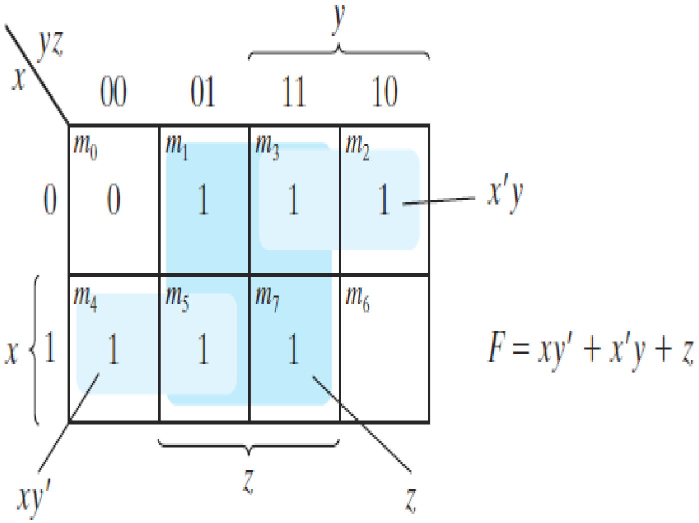
$$F(x, y, z) = (1, 2, 3, 4, 5, 7)$$

The first step is to simplify the function into sum-of-products form. This is done by means of the K-map from which the simplified function is obtained: $F = xy' + x'y + z$

Logic Diagram:







x \ yz	00	01	11	10
0	m_0 0	m_1 1	m_3 1	m_2 1
1	m_4 1	m_5 1	m_7 1	m_6

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

Realization with NAND & NOR Gates

Now we list the steps for obtaining the logic diagram from a Boolean function.

- Simplify the function and express it in sum-of-products form.
- Draw a NAND gate for each product term of the expression that has at least two literals
- The inputs to each NAND gate are the literals of the term.
- This procedure produces a group of first-level gates.
- Draw a single gate using the AND-invert or the invert-OR graphic symbol in the second level, with inputs coming from outputs of first-level gates.
- A term with a single literal requires an inverter in the first level. However, if the single literal is complemented, it can be connected directly to an input of the second level NAND gate.

Realization with NAND & NOR Gates

Multilevel NAND circuits:

The standard form (sum of products) of expressing Boolean functions results in a two-level implementation.

Procedure for design of multilevel circuits:

In the design of multilevel circuits a given Boolean function is expressed in terms of AND, OR, and complement operations. The function is then implemented with AND & OR gates.

If necessary, it is then converted into an all-NAND circuit.

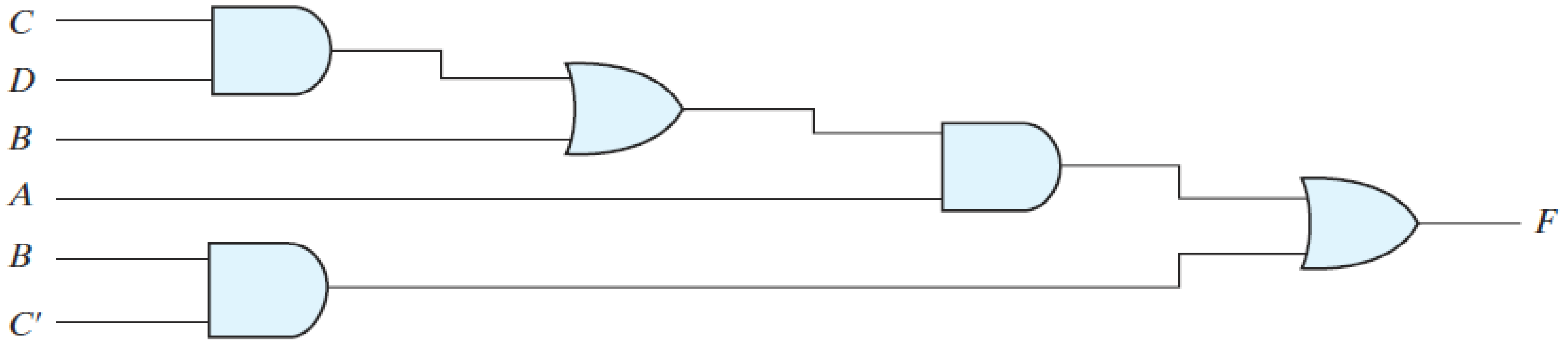
Example:

Consider the Boolean function: $F = A(CD + B) + BC'$

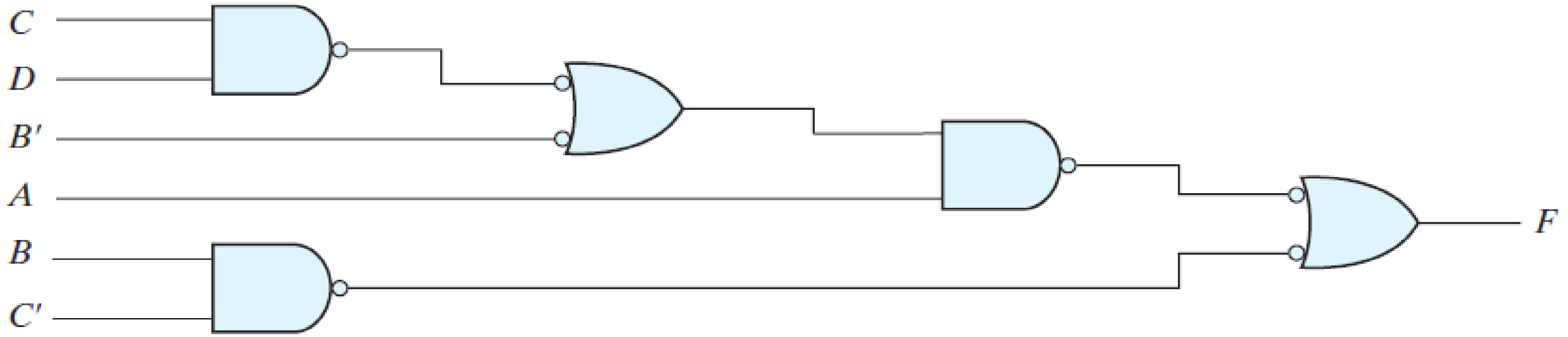
Although it is possible to remove the parentheses and reduce the expression into a standard sum-of-products form, for illustration purpose we choose to implement it as a multilevel circuit .

Realization with NAND & NOR Gates

Logic Diagram with AND-OR gates & with NAND gates



(a) AND–OR gates



(b) NAND gates

Realization with NAND & NOR Gates

Procedure for converting a multilevel AND–OR diagram into an all-NAND diagram using mixed notation:

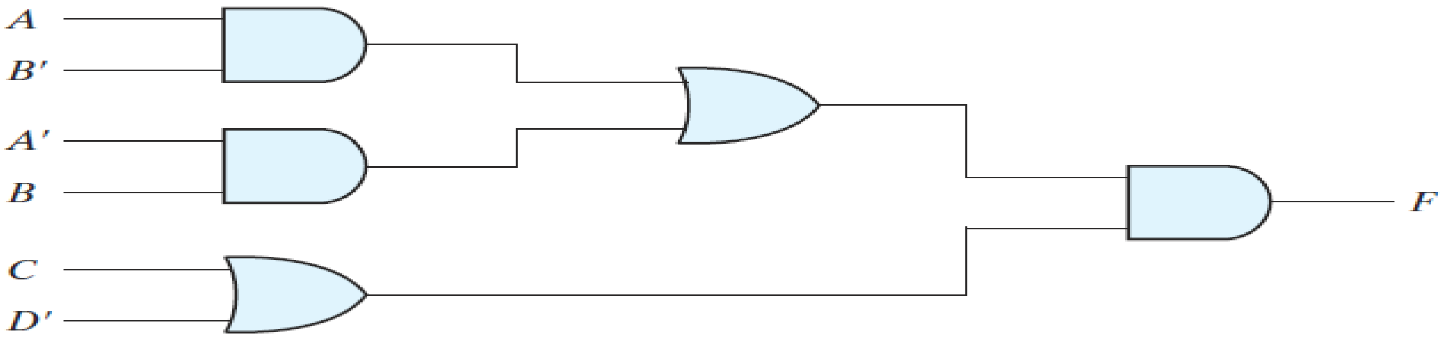
- Convert all AND gates to NAND gates with AND-invert graphic symbols.
- Convert all OR gates to NAND gates with invert-OR graphic symbols.
- Check all the bubbles in the diagram. For every bubble that is not compensated by another small circle along the same line, insert an inverter (a one-input NAND gate) or complement the input literal.

Realization with NAND & NOR Gates

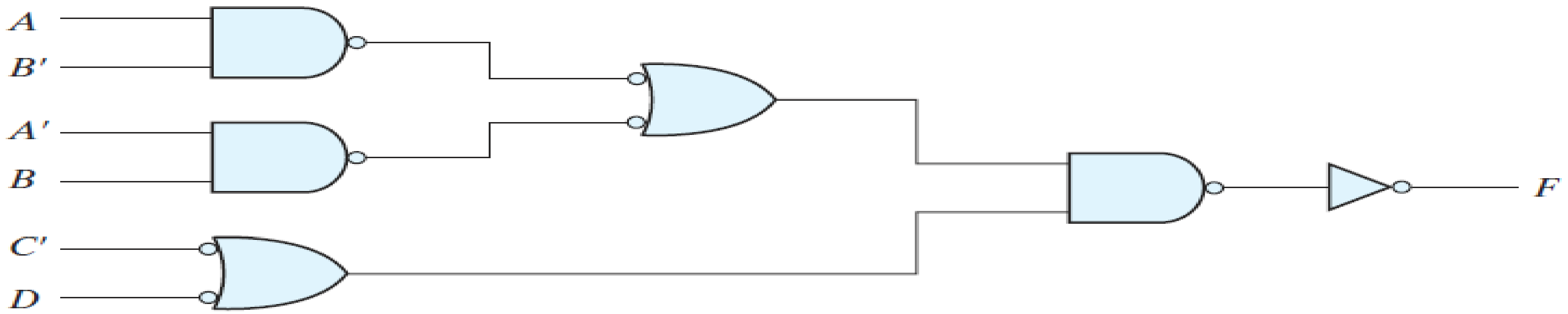
Example:

Consider the multilevel Boolean function $F = (AB' + A'B)(C + D')$

Logic Diagram: Conversion from AND-OR gates to NAND gates



(a) AND–OR gates



(b) NAND gates

UNIT - III

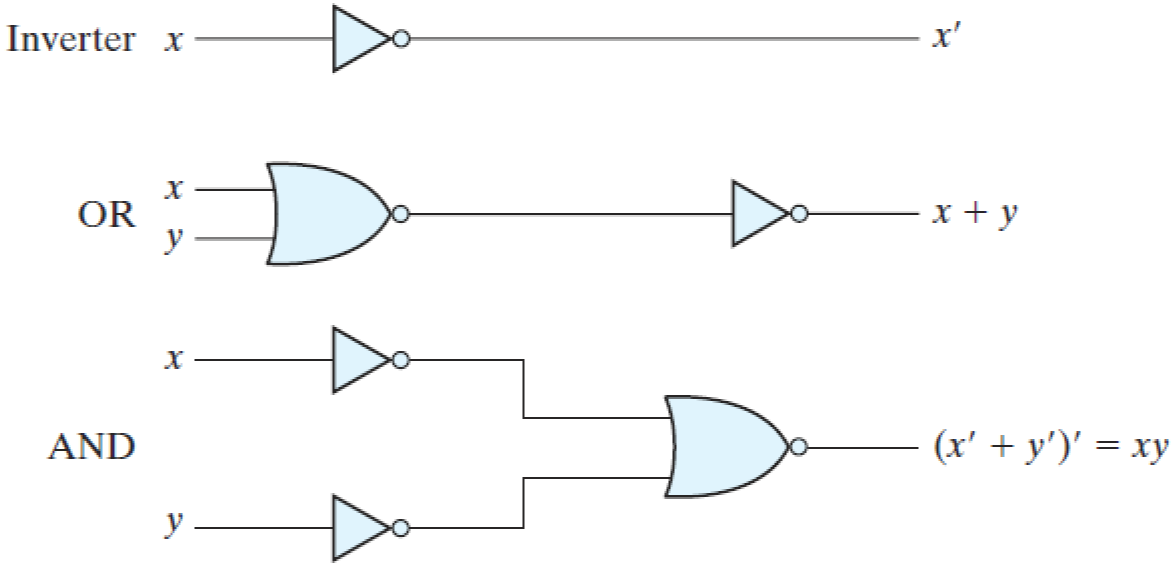
SEQUENTIAL LOGIC DESIGN

Design with NOR Gates

- The NOR operation is the dual of the NAND operation. Therefore,
- all procedures and rules for NOR logic are the duals of the corresponding procedures and rules developed for NAND logic
- The NOR gate is another universal gate that can be used to implement any Boolean function.

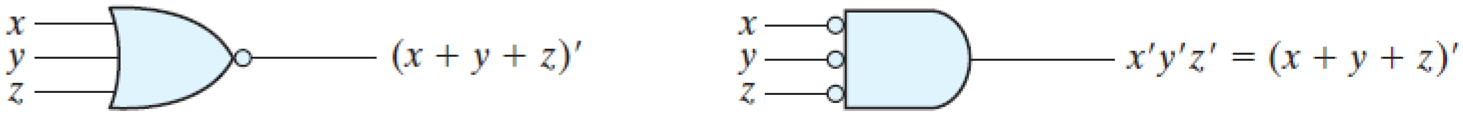
Implementation of the complement, OR, & AND operations with NOR gates :

The complement operation is obtained from a one input NOR gate that behaves exactly like an inverter.



Design with NOR Gates

Two graphic symbols for the NOR gate:



(a) OR-invert (b) Invert-AND

- The OR-invert symbol defines the NOR operation as an OR followed by a complement.
- The invert-AND symbol complements each input and then performs an AND operation.
- The two symbols designate the same NOR operation and are logically identical because of De Morgan's theorem.

Design with NOR Gates

Procedure for implementation with NOR gates:

- A two-level implementation with NOR gates requires that the function be simplified into product-of-sums form.
- We know the simplified product-of-sums expression is obtained from the map by combining the 0's and complementing.
- A product-of-sums expression is implemented with a first level of OR gates that produce the sum terms followed by a second-level AND gate to produce the product.

Conversion from OR – AND gates to NOR gates implementation:

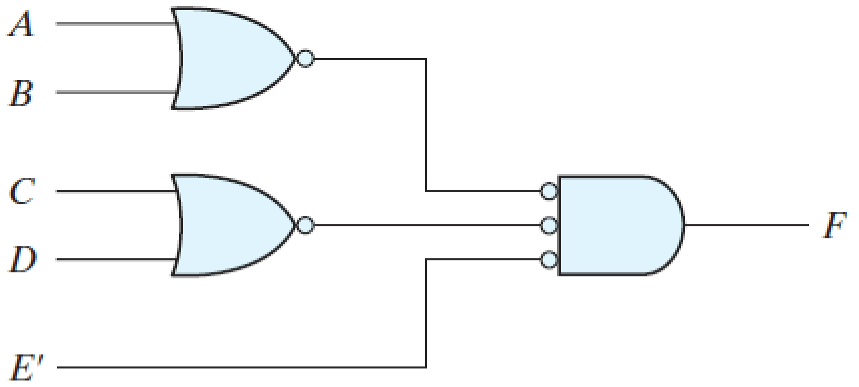
- It is achieved by changing the OR gates to NOR gates with OR-invert graphic symbols, and
- the AND gate to a NOR gate with an invert-AND graphic symbol.
- A single literal term going into the second-level gate must be complemented.

Design with NOR Gates

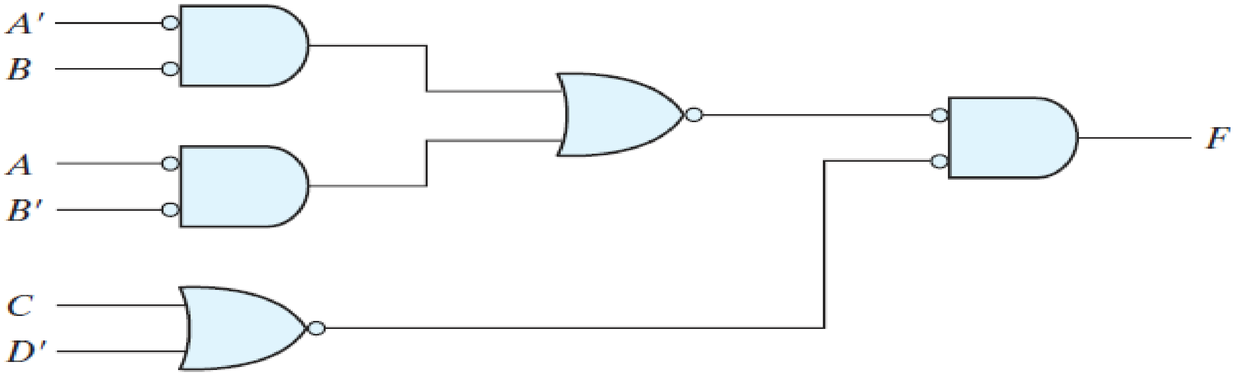
Example:

Given a function in product of sums form: $F = (A + B)(C + D)E$

NOR Gate based logic diagram:



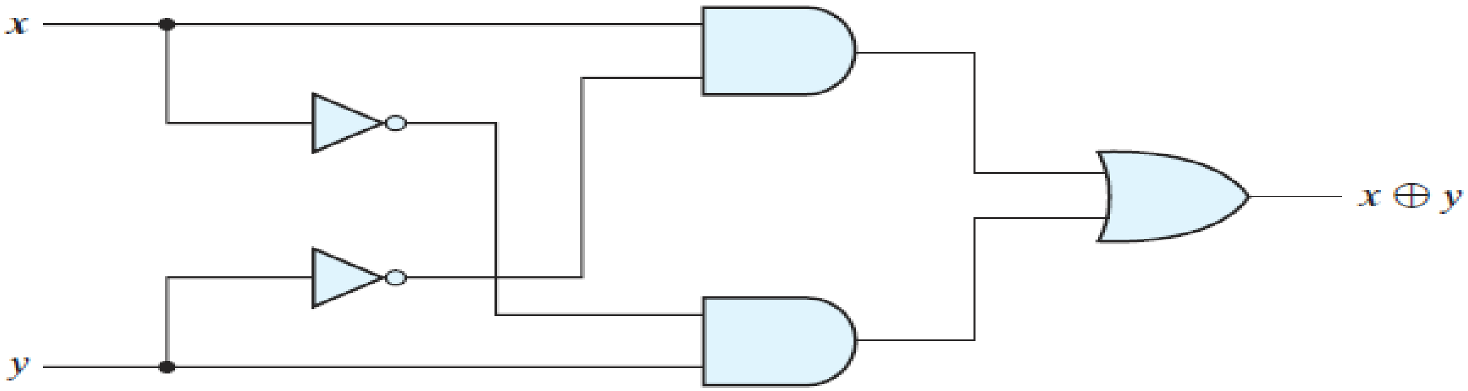
Example: $F = (AB' + A'B)(C + D')$



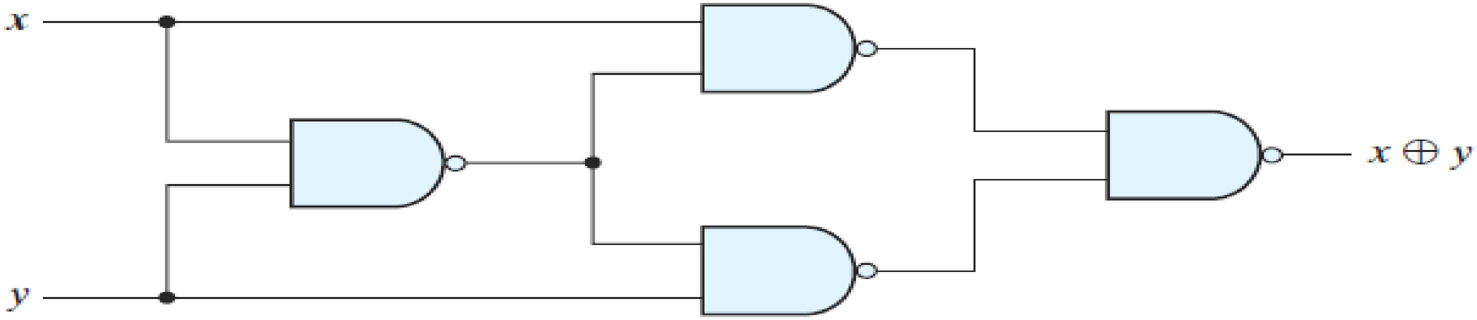
Exclusive OR (XOR) with NAND Gates

Implement Exclusive OR with NAND gates:

The XOR function is $x \oplus y = xy' + x'y$



(a) Exclusive-OR with AND–OR–NOT gates

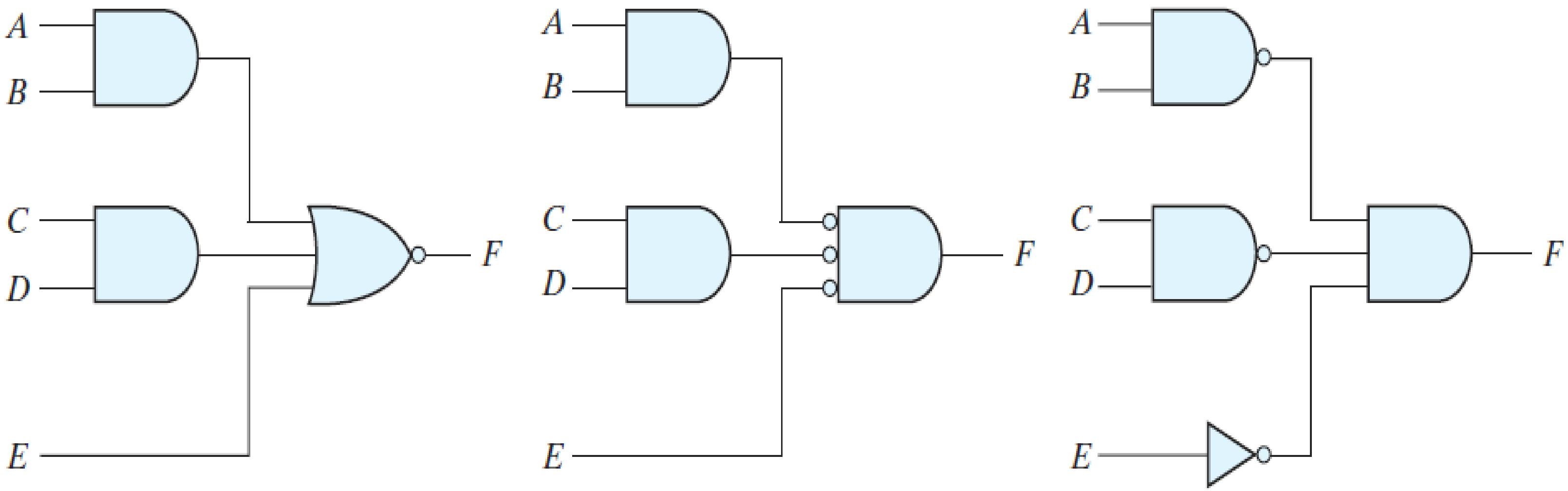


(b) Exclusive-OR with NAND gates

AND–OR–INVERT Implementation

- The two non-degenerate forms, NAND–AND & AND–NOR, are equivalent and can be treated together
- Both perform the AND–OR–INVERT function
- The AND–NOR form resembles the AND–OR form, but with an inversion done by the bubble in the output of the NOR gate.

Example: Implement a function $F = (AB + CD + E)'$



(a) AND–NOR (b) AND–NOR (c) NAND–AND

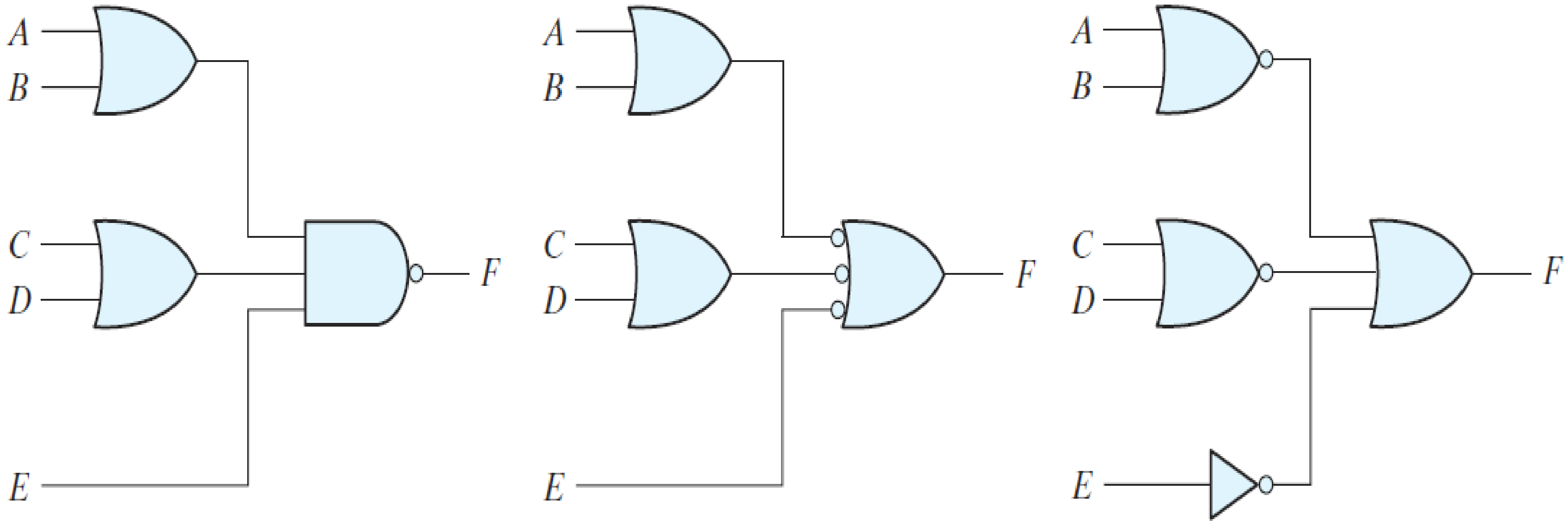
OR–AND–INVERT Implementation

- The OR–NAND and NOR–OR forms perform the OR–AND–INVERT function
- The OR–NAND form resembles the OR–AND form, except for the inversion done by the bubble in the NAND gate.
- The OR–AND–INVERT implementation requires an expression in product-of-sums form.
- If the complement of the function is simplified into product-of-sums form, we can implement
- F with the OR–AND part of the function.
- When F passes through the INVERT part, we obtain the complement of F′, or F , in the output.

It implements the function like: $F = [(A + B)(C + D)E]'$

OR–AND–INVERT Implementation

Logic Diagram for $F = [(A + B)(C + D)E]'$



(a) OR–NAND (b) OR–NAND (c) NOR–OR

AND-OR-INVERT & OR–AND–INVERT Example

Example: Given the function $F = x'y'z' + xyz'$

Synthesize AND-OR-INVERT & OR-AND-INVERT implementations

Solution:

AND-OR-INVERT implementation yields AND-OR & NAND-AND configurations. Synthesizes ‘complement of the sum of product form’

OR-AND-INVERT implementation yields OR-NAND & NOR-OR configurations. Synthesizes ‘complement of product of sum form’

AND-OR-INVERT Implementation:

Since it synthesizes ‘complement of the sum of product form’ , in the K-map for the given function we select squares filled with 0s & obtain simplified function which is F′ & is in sum of products form . Its complement yields F.

AND-OR-INVERT & OR–AND–INVERT Example

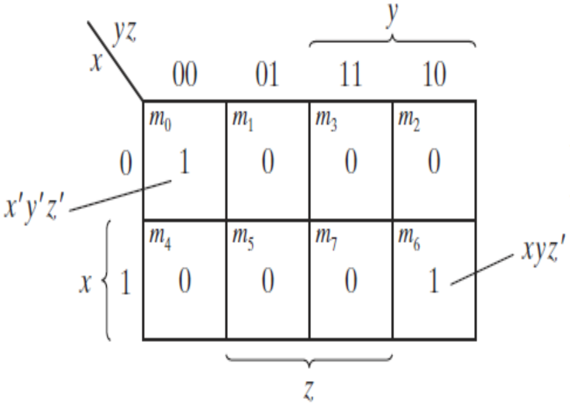
The K-map:

Simplifying the function for squares filled with 0s, we get $F' = x'y + xy' + z$

F′ is in sum of product form.

AND-OR-INVERT yields its complement

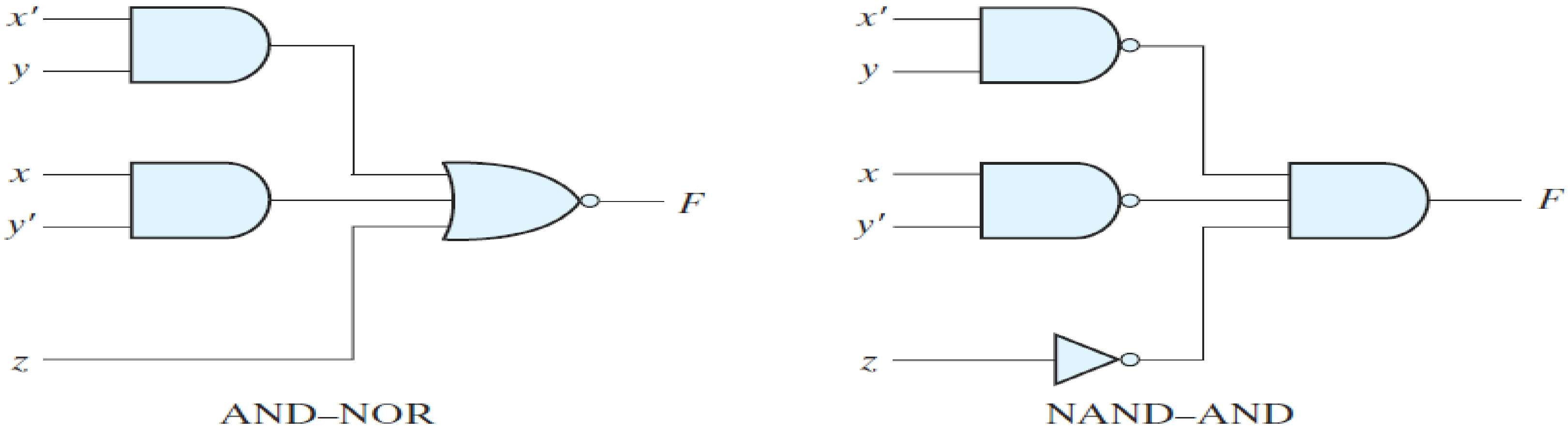
& hence we get F. The logic diagram is:



$x \backslash yz$	00	01	11	10
0	m_0 1	m_1 0	m_3 0	m_2 0
1	m_4 0	m_5 0	m_7 0	m_6 1

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

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



AND–NOR

NAND–AND

(b) $F = (x'y + xy' + z)'$

AND-OR-INVERT & OR–AND–INVERT Example

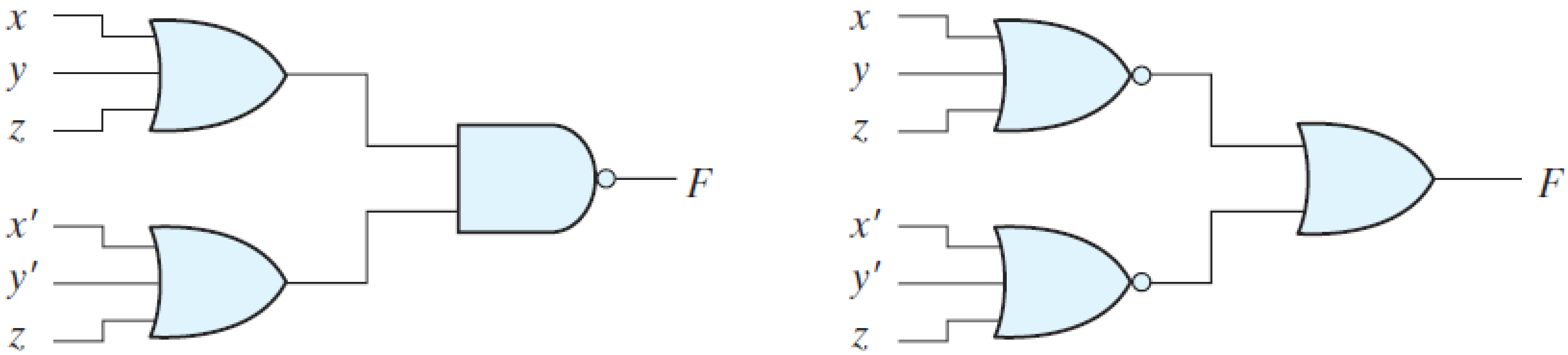
OR-AND-INVERT implementation:

- The OR–AND–INVERT forms require a simplified expression of the complement of the function in product-of-sums form.
- To obtain this expression, we first combine the 1’s in the map:

We get the function $F = x'y'z' + xyz'$

Then we take the complement of the function, to get:

The function F = (F’)’ $F' = (x + y + z)(x' + y' + z)$



OR–NAND NOR–OR

(c) $F = [(x + y + z)\ (x' + y' + z)]'$

Exclusive OR : An Odd Function

Odd Function:

The exclusive-OR operation with three or more variables can be converted into an ordinary Boolean function by expanding it, as shown:

$$\begin{aligned} A \oplus B \oplus C &= (AB' + A'B)C' + (AB + A'B')C \\ &= AB'C' + A'BC' + ABC + A'B'C \\ &= \Sigma(1, 2, 4, 7) \end{aligned}$$

We observe from the Boolean expression that :

the three -variable exclusive-OR function is equal to 1 if only one variable is equal to 1 or if all three variables are equal to 1.

Contrary to the two-variable case, in which only one variable must be equal to 1, in the case of three or more variables the requirement is that an odd number of variables be equal to 1.

Therefore the multiple-variable exclusive-OR operation is defined as an odd function.

Exclusive OR : An Odd Function

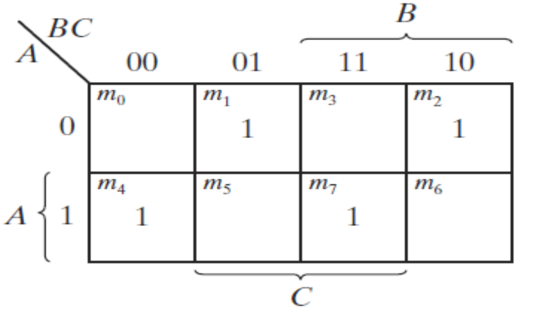
- The Boolean function derived from the three-variable exclusive-OR operation is expressed as the logical sum of four minterms whose binary numerical values are 001, 010,100, and 111.
- Each of these binary numbers has an odd number of 1's.
- The remaining four minterms not included in the function are 000, 011, 101, and 110, and they have an even number of 1's in their binary numerical values.
- In general, an n -variable exclusive-OR function is an odd function defined as the logical sum of the 2^n /2 minterms whose binary numerical values have an odd number of 1's.

In K-map we fill only those squares with '1' which have odd number of 1's in them. Remaining squares are filled with 0s.

If we want to determine complement then we fill '1' in the squares that have '0' (i.e. terms not included in the logic function)

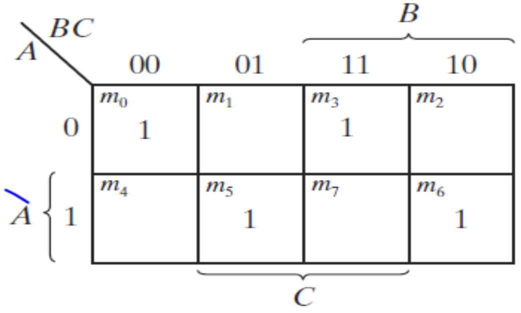
Exclusive OR : An Odd Function

The K-map for 3 input Exclusive –OR function:



A \ BC	00	01	11	10
0	m_0	m_1 1	m_3	m_2 1
1	m_4 1	m_5	m_7 1	m_6

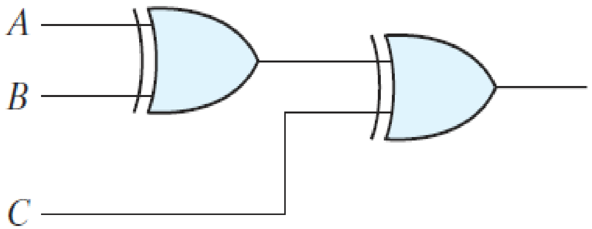
(a) Odd function $F = A \oplus B \oplus C$



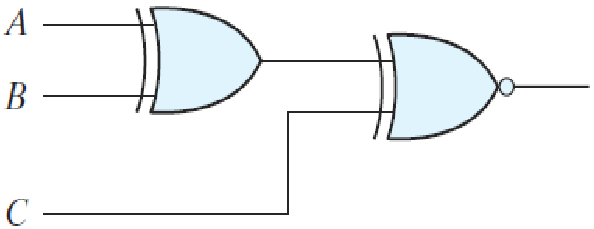
A \ BC	00	01	11	10
0	m_0 1	m_1	m_3 1	m_2
1	m_4	m_5 1	m_7	m_6 1

(b) Even function $F = (A \oplus B \oplus C)'$

- The three-input odd function is implemented by means of two-input exclusive-OR gates,
- The complement of an odd function is obtained by replacing the output gate with an exclusive-NOR gate.



(a) 3-input odd function

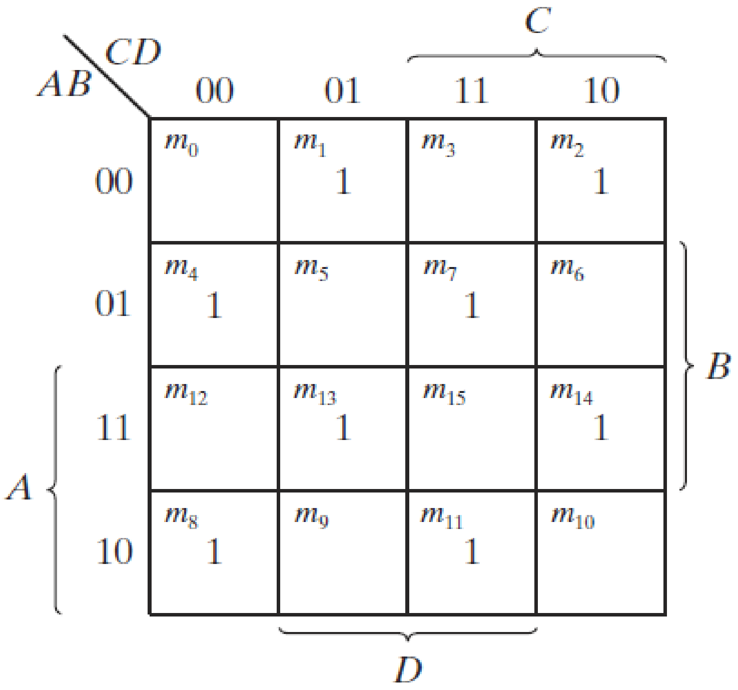


(b) 3-input even function

Exclusive OR : An Odd Function

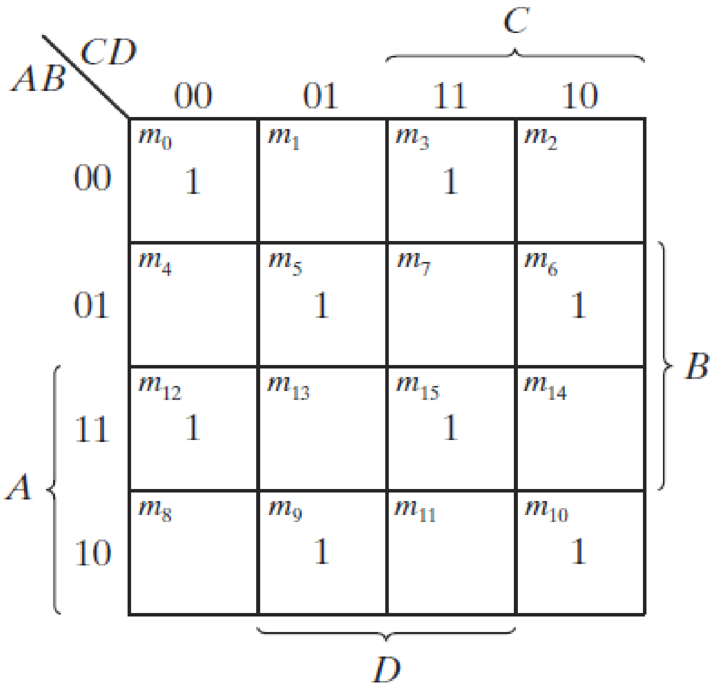
4 variable Exclusive – OR operation:

The K-maps are shown:



AB \ CD	00	01	11	10
00	m_0	m_1 1	m_3	m_2 1
01	m_4 1	m_5	m_7 1	m_6
11	m_{12}	m_{13} 1	m_{15}	m_{14} 1
10	m_8 1	m_9	m_{11} 1	m_{10}

(a) Odd function $F = A \oplus B \oplus C \oplus D$



AB \ CD	00	01	11	10
00	m_0 1	m_1	m_3 1	m_2
01	m_4	m_5 1	m_7	m_6 1
11	m_{12} 1	m_{13}	m_{15} 1	m_{14}
10	m_8	m_9 1	m_{11}	m_{10} 1

(b) Even function $F = (A \oplus B \oplus C \oplus D)'$

Analysis of Clocked Sequential Circuits

Analysis of a system/circuit:

When we say ‘we want to analyze a system/circuit’ we mean to determine what a given system/circuit will do under certain operating conditions.

The behaviour of a clocked sequential circuit is determined from:

- the inputs fed to it,
- the outputs obtained from it, and
- the state of its flip-flops.

The outputs and the next state are both a function of the inputs and the present state.

The analysis of a sequential circuit consists of obtaining a table or a diagram for the time sequence of

- inputs,
- outputs, and
- internal states.

It is possible to write Boolean expressions that describe the behaviour of the sequential circuit. These expressions must include the necessary time sequence, either directly or indirectly.

Analysis of Clocked Sequential Circuits

Clocked sequential circuit:

A logic diagram is recognized as a clocked sequential circuit if it includes flip-flops with clock inputs.

The flip-flops may be of any type, and the logic diagram may or may not include combinational logic gates.

We will study about

- ○ how to specify the next-state condition in terms of the present state and inputs,
- State tables,
- State diagram to describe the behaviour of the sequential circuit.

State Equations

State Equations:

These are algebraic equations which describe behaviour of a clocked sequential circuit.

A state equation (also called a transition equation) specifies the next state as a function of the present state and inputs.

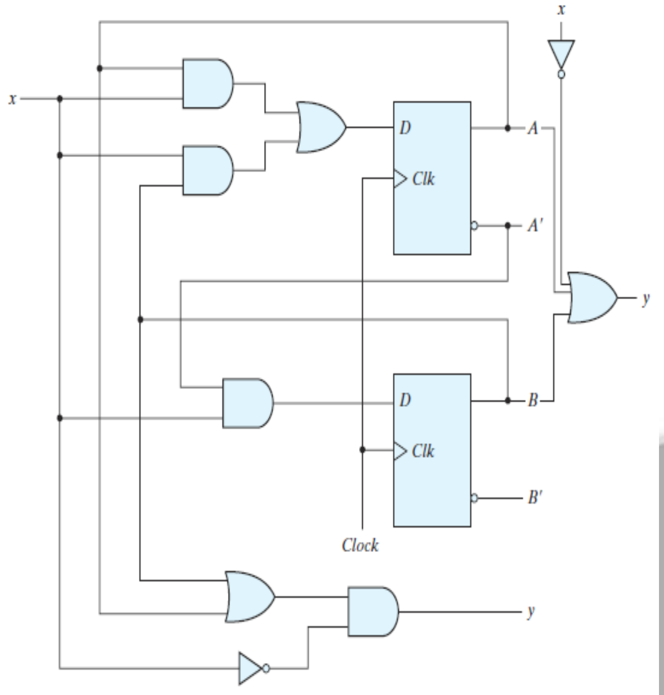
Example:

Consider a sequential circuit shown in the diagram:

It consists of two D flip-flops A and B, an input x and an output y . Since the D input of a flip-flop determines the value of the next state (i.e., the state reached after the clock transition), it is possible to write a set of state equations for the circuit, as:

$$A(t + 1) = A(t)\,x(t) + B(t)\,x(t) \qquad \text{.... (1)}$$

$$B(t + 1) = A'(t)\,x(t) \qquad \text{................ (2)}$$



A(t) & B(t): are ‘present’ states (outputs) of flip flops;

A(t+1) & B(t+1): are ‘next’ states (outputs) of flip flop

State equations are algebraic equations (1) & (2) .

State Equations

A state equation is an algebraic expression that specifies the condition for a flip-flop state transition.

Consider Equations (1) & (2):

$A(t + 1) = A(t)\,x(t) + B(t)\,x(t)$ .... (1)

$B(t + 1) = A'(t)\,x(t)$ .... (2)

The left side of the equation, with (t + 1), denotes the next state of the flip-flop one clock edge later.

The right side of the equation is a Boolean expression that specifies the present state and input conditions that make the next state equal to ‘1’.

In more compact representation of the State equations, in the RHS of the state equation we omit ‘t’ after each variable for convenience and express the state equations in the form where ‘t’ is implicitly present.

$A(t + 1) = A\,x + B\,x$ .... (3)

$B(t + 1) = A'\,x$ .... (4)

Next state occurs only at the appearance of a clock pulse.

State Equations

The Boolean expressions for the state equations can be derived directly from the gates that form the combinational circuit part of the sequential circuit, since the D values of the combinational circuit determine the next state.

Similarly, the present-state value of the output can be expressed algebraically as:

$$y(t) = [A(t) + B(t)]x'(t)$$

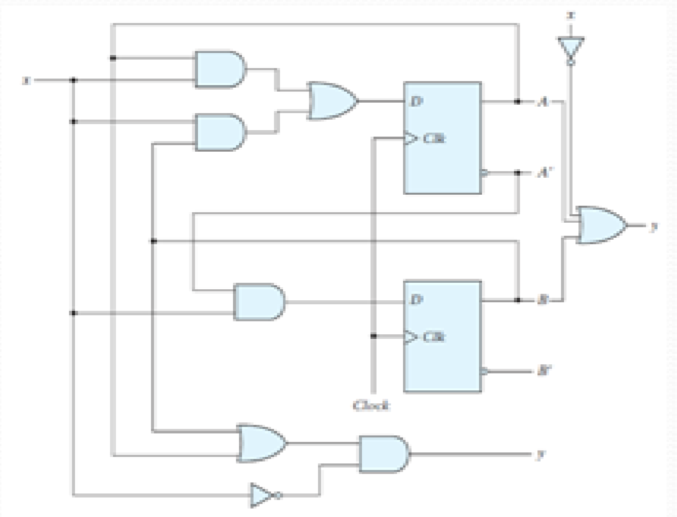
By removing the symbol (t) for the present state, we obtain the output Boolean equation:

$$y = (A + B)x'$$

State Table

State table:

In a state table (transition table)we enumerate the time sequence of inputs, outputs, and flip-flop states. The state table for the given circuit is shown:



Present State		Input	Next State		Output
A	*B*	*x*	*A*	*B*	*y*
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

The table consists of four variables labelled present state, input, next state, and output . The present-state (2 columns) shows the states of flip-flops A and B at any given time t.

The input column gives a value of x for each possible present state.

The next-state column shows the states of the flip-flops one clock cycle later, at time t + 1. The output column gives the value of y at time t for each present state and input condition.

State Table

Derivation of a state table:

The derivation of a state table requires listing all possible binary combinations of present states and inputs.

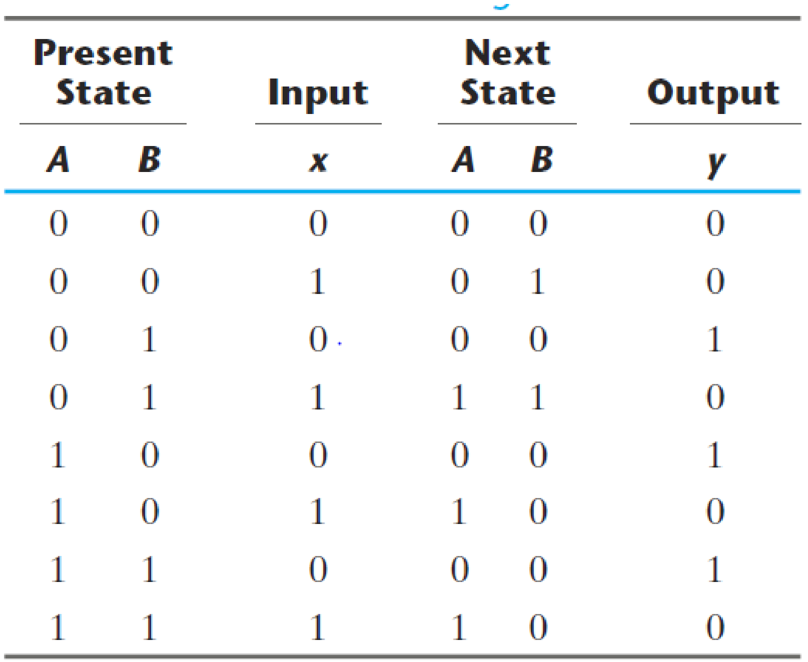
In the diagram we have 2 flip flops & hence we have 2 states (A & B). Each one can assume a value ‘0’ or ‘1’. (AB) : (00 01 10 11)

We have an input ‘x’ which can be ‘0’ or ‘1’.

Therefore, from the table we see that for each possible combination of ‘A’ &’B’ we have assigned ‘x’ a value ‘0’ & ‘1’.

So we have eight binary combinations from 000 to 111 for the sequence (A B x).

The next-state values are then determined from the logic diagram or from the state equations [A(t+1) = A x + B x & B(t+1) = A′ x].



Present State		Input	Next State		Output
A	*B*	*x*	*A*	*B*	*y*
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

The output column is derived from the output equation y = (A + B) x′

State Table

General Remark:

In general, a sequential circuit with m flip flops and n inputs needs 2^{m+n} rows in the state table. present example: m = 2 & n=1

The binary numbers from ‘0’ through 2^{m+n} - 1 (= 7) are listed under the present-state and input columns.

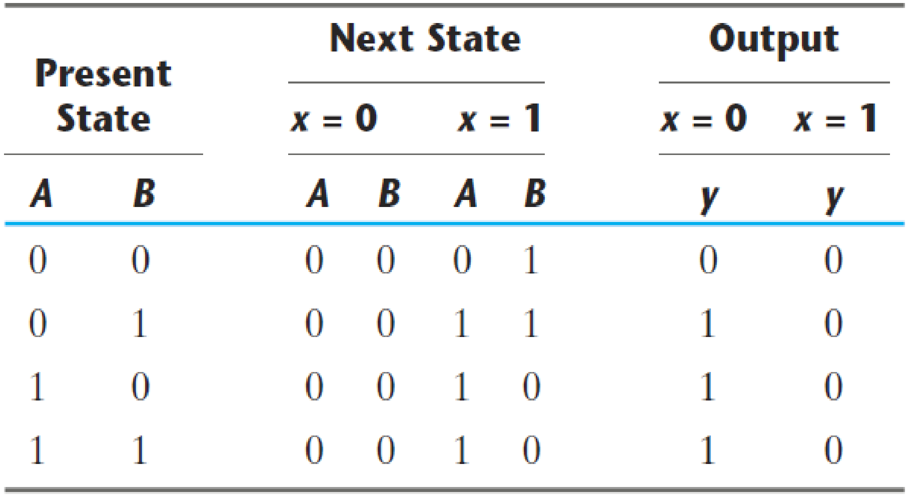
The next-state section has m (= 2) columns, one for each flip-flop.

The binary values for the next state are derived directly from the state equations. The output section has as many columns as there are output variables. Its binary value is derived from the circuit or from the Boolean function in the same manner as in a truth table.

State table in another form:

In this form, the input conditions are listed under the next-state and output columns.

Second Form of the State Table



Present State		Next State				Output	
		$x = 0$		$x = 1$		$x = 0$	$x = 1$
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

State Diagram

State diagram:

It is a graphical representation for a given state table. The information available in a state table can be represented graphically in the form of a state diagram.

In this type of diagram, a state is represented by a circle, and the (clock-triggered) transitions between states are indicated by directed lines connecting the circles.

Example:

The State Table & its

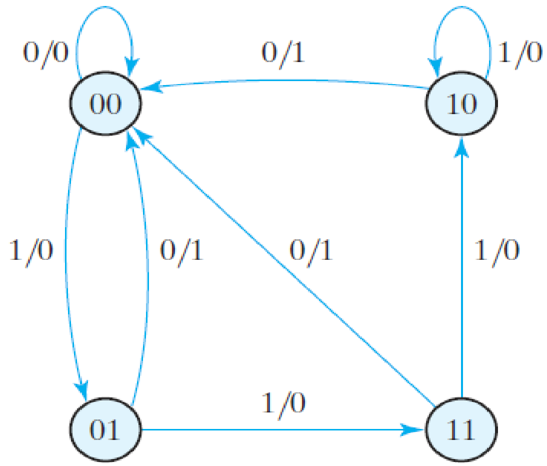
State Diagram is shown.

States: (A B): (00 01 10 11)

Each state is depicted by a

circle.

Present State		Next State				Output	
		x = 0		x = 1		x = 0	x = 1
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

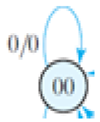


We have 4 combinations and hence we have 4 circles.

Each link (branch) connecting the circles shows the transition from one state to another (in the direction of arrow) at the occurrence of clock pulse. Each link is labeled with input/output tag like 1/0; implying that input = 1 & output = 0

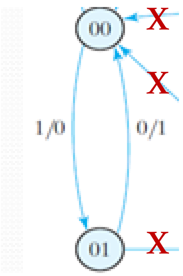
State Diagram

How to read a State diagram:



: A self loop implies no transition. The system continues to remain in its

: previous state (00). ‘0/0’ (in/out) implies that input (x) & output (y) = 0.



System input (X) is different from input to a flip flop.

x: implies that the links do not exist

The output, y, is given as: $y = A x' + B x'$

It is important to remember that the bit value listed for the output along the directed line occurs during the present state and with the indicated input, and has nothing to do with the transition to the next state.

For example, the directed line from state ’00’ to ’01’ is labelled 1/0, meaning that when the sequential circuit is in the present state ’00’ and the input is ‘1’, the output is ‘0’. After the next clock cycle, the circuit goes to the next state, ’01’.

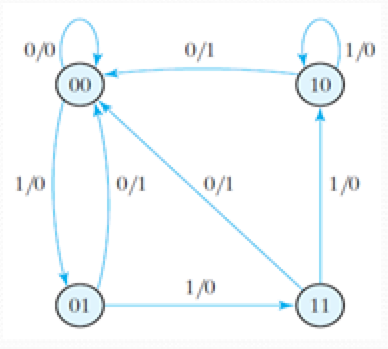
If the input changes to ‘0’, then the output becomes ‘1’, but if the input remains at ‘1’, the output stays at ‘0’.

Thus, while system output may change (while system is in particular state) depending upon input, the state transition occurs only if clock pulse occurs.

State Diagram

We summarize as under:

- Start from Circuit / system diagram
- Determine system equations
- Formulate State table, & then
- Draw the State diagram



There is no difference between a state table and a state diagram, except in the manner of representation.

The state table is easier to derive from a given logic diagram and the state equation.

The state diagram follows directly from the state table.

The state diagram gives a pictorial view of state transitions and is the form more suitable for human interpretation of the circuit’s operation .

For example, the state diagram clearly shows that, starting from state ’00’, the output is ‘0’ as long as the input stays at ‘1’. The first ‘0’ input after a string of 1’s gives an output of 1 and transfers the circuit back to the initial state, ’00’.

The machine represented by this state diagram acts to detect a zero in the bit stream of data because output becomes ‘1’ only for input bit = 0.

Flip – Flop Input Equations

Excitation Equations:

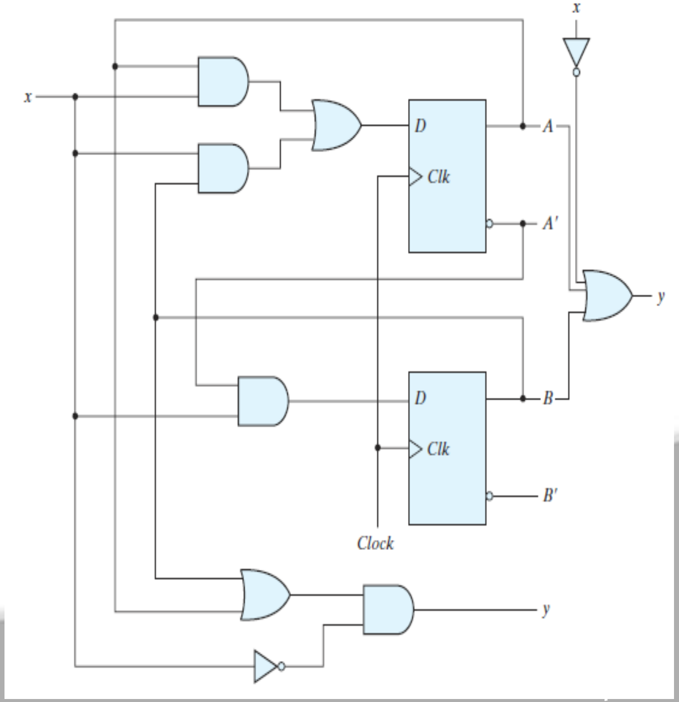
Flip flop input equations are also known as excitation equations.

The part of the circuit that generates the inputs to flip-flops is described algebraically by a set of Boolean functions called flip-flop input equations.

Convention used for input (excitation) equations:

We will adopt the convention of using the flip-flop input symbol to denote the input equation variable and a subscript to designate the name of the flip-flop output.

Example:



Consider the input equation: $D_Q = x + y$.

It specifies an OR gate with inputs x and y connected to the D input of a flip-flop whose output is labelled with the symbol Q.

For the shown diagram we write input equations as:

$D_A = A\,x + B\,x$ & $D_B = A'\,x$

The output equation as : $y = (A + B)x'$

Flip – Flop Input Equations

D_A = A x + B x & D_B = A′ x; y = (A + B)x′

The three equations, as reproduced above, provide the necessary information for drawing the logic diagram of the sequential circuit.

The symbol D_A specifies a D flip-flop labelled A . D_B specifies a second D flip-flop labelled B.

The Boolean expressions associated with D_A & D_B and the expression for output y specify the combinational circuit part of the sequential circuit because these equations can be simulated using logic gates.

Note:

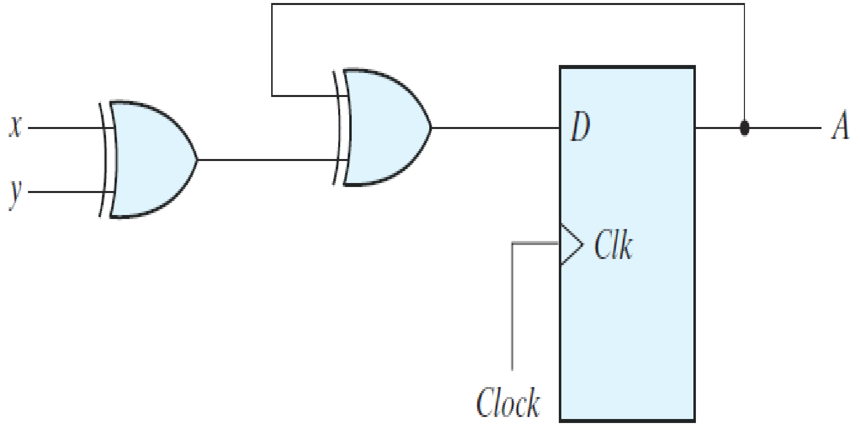
Note that the expression for the input equation for a D flip-flop is identical to the expression for the corresponding state equation. This is because of the characteristic equation that equates the next state to the value of the D input: Q(t + 1) = D_Q.

Analysis using Flip - Flops

Analysis with D Flip-Flops:

The circuit diagram to be analyzed is shown:

XOR gates are at the input.



The input (excitation)equation:

For the D Flip flop is given by:

$$D_A = A \oplus x \oplus y$$

State Equation:

Since the next state of D flip flop is

equal to the D input (1 or 0), the state equation for the flip flop is :

$$A(t + 1) = A \oplus x \oplus y$$

State table:

It will have 4 columns: 2 corresponding to inputs x & y and the other 2 to present & next state, A. Next state is a function of x, y & ‘A’ value when clock pulse occurs.

The pair (x y) combinations:(00 01 10 11); A : (0 1)

For each combination of (x y), A can take a value ‘0’ or ‘1’ (continued….)

Analysis using Flip - Flops

State table is determinable from the state equation, given as:

$$A(t+1) = A \oplus x \oplus y$$

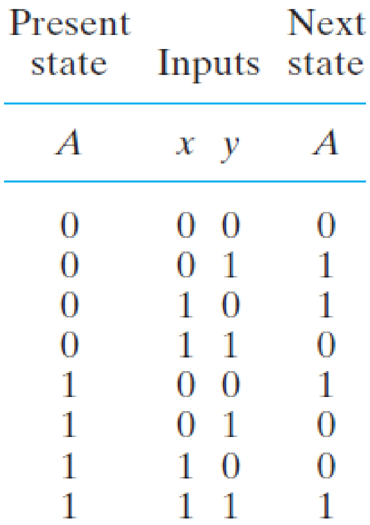
(next state) (present state)

Let Z = $x \oplus y$; $A(t+1) = A \oplus Z$

$$Z = x\,y' + x'y \quad \text{................ } 1$$

$$A(t+1) = A\,Z' + A'Z \quad \text{.............. } 2$$

In the table we have for all the combinations of input pair (x y) & present state 'A' value = '0' & '1' tabulated.



Present state	Inputs		Next state
A	x	y	A
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

Determination of Next state:

Making use of Eqs. 1 & 2, we determine the next state, for each & every row in the table. For example:

1st Row: A = 0, x=0, y=0; therefore, x′ = y′= A′=1

Therefore, Z = (0) (1) + (1) (0) = 0; Z′= 1.

Hence, next state = A(t + 1) = A (in the table) = (0) (1) + (1) (0) = 0

Similarly we can determine next state for the rows in the table, using Eqs. 1 & 2.

Analysis using Flip - Flops

Analysis with J-K Flip Flops:

We know that,

- A state table consists of four sections: present state, inputs, next state, and outputs.
- The first two (present state & inputs) are obtained by listing all binary combinations.
- The output column is determined from the output equations.
- The next-state values are evaluated from the state equations.

We also know that for a D -type flip-flop, the state equation is the same as the input equation.

When a flip-flop other than the D type is used, such as JK or T, it is necessary to refer to the corresponding characteristic table or characteristic equation to obtain the next state values.

Procedure for determining next state: (2 methods)

- by using the characteristic (state) table and
- by using the characteristic equation.

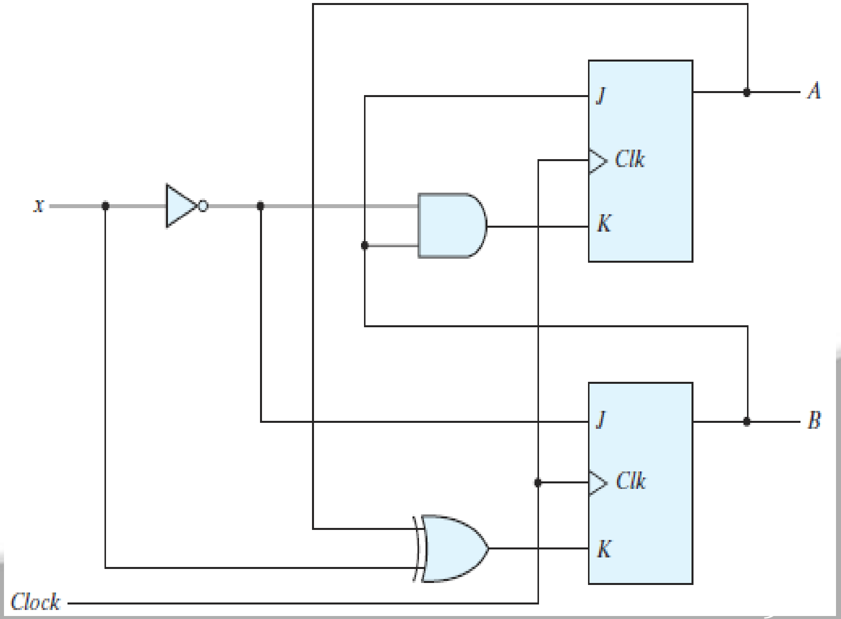
Analysis using Flip - Flops

Steps for determining next state using JK or T flip flop:

- Determine the flip-flop input equations in terms of the present state and input variables.
- List the binary values of each input equation.
- Use the corresponding flip-flop characteristic table to determine the next-state values in the state table.

Example using JK flip flop:

Consider the sequential circuit with two JK flip-flops A and B and one input x, as shown in the Fig.



The circuit has no outputs; therefore, the state table does not need an output column. (The outputs of the flip-flops may be considered as the outputs in this case.)

The flip flop input equations are written as:

- $J_A = B$ & $K_A = B x'$
- $J_B = x'$ & $K_B = A'x + A x' = A \oplus x$

Analysis using Flip - Flops

State table:

The state table of the sequential circuit is shown in the Table.

The present-state and input columns list the eight binary combinations.

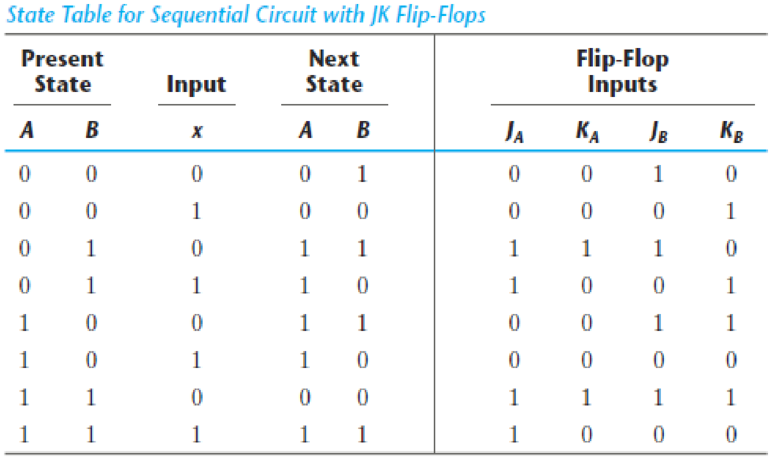
The binary values listed under the columns labelled flip-flop inputs are not part of the state table, but they are needed for the purpose of evaluating the next state as specified in step 2 of the procedure.

These binary values are obtained directly from the four input equations in a manner similar to that for obtaining a truth table from a Boolean expression.

From the table we see that the present states of flip flops A & B are defined for every possible value (0 or 1) of input x. So we get 8 possible Combinations of A, B & x.

The columns under 'Flip-Flop Inputs' are filled using flip flop input equations,

- $J_A = B$ & $K_A = B x'$
- $J_B = x'$ & $K_B = A'x + A x' = A \oplus x$



State Table for Sequential Circuit with JK Flip-Flops

Present State		Input	Next State		Flip-Flop Inputs			
A	B	x	A	B	J_A	K_A	J_B	K_B
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0

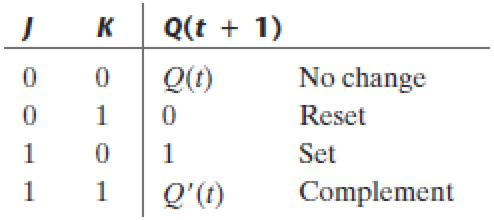
Where, present states of A & B are substituted. We have completed 2 steps.

Analysis using Flip - Flops

Determination of next state of flip flops:

The next state of each flip-flop is evaluated from the corresponding J and K inputs and the characteristic table of the JK flip-flop listed in the Table.

There are four cases to consider (as seen from the table).



J	K	$Q(t+1)$	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

- When J = 1 and K = 0, the next state is 1.
- When J = 0 and K = 1, the next state is 0.
- When J = K = 0, there is no change of state and the next-state value is the same as that of the present state.
- When J = K = 1, the next-state bit is the complement of the present-state bit.

So we need to fill up columns related to ‘Next State’

A flip-flop: Present State : PS NO Change : NC

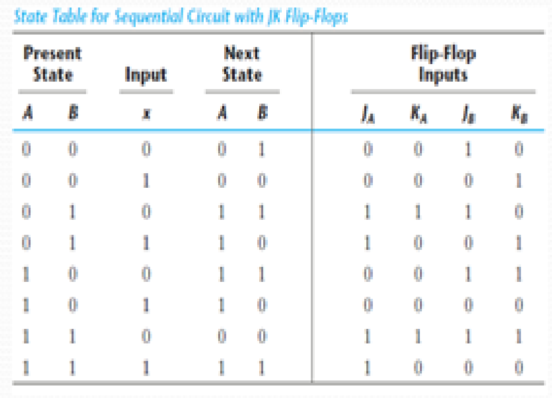
1st Row: PS = 0; J_A & K_A = 0 therefore next state A = 0: NC

2nd Row: PS = 0; J_A & K_A = 0 therefore next state A = 0: NC

3rd Row: PS = 0; J_A & K_A = 1 therefore next state A = A′= 1

4th Row: PS = 0; J_A =1 & K_A = 0 therefore next state A = 1

5th Row: PS = 1; J_A & K_A =0 therefore next state A = 1: NC (contd. ….)



State Table for Sequential Circuit with JK Flip-Flops

Present State		Input	Next State		Flip-Flop Inputs			
A	B	x	A	B	J_A	K_A	J_B	K_B
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0

Analysis using Flip - Flops

6th Row: PS = 1; J_A & K_A = 0 therefore next state A = 1: NC

7th Row: PS = 1; J_A & K_A = 1 therefore next state A = A'= 0

8th Row: PS = 1; J_A = 1 & K_A = 0 therefore next state A = 1

B flip flop: Present State : PS NO Change : NC

1st Row: PS = 0; J_B = 1 & K_B = 0 therefore next state B = 1

2nd Row: PS = 0; J_B = 0 & K_B = 1 therefore next state B = 0

3rd Row: PS = 1; J_B =1 & K_B = 0 therefore next state B =1

4th Row: PS = 1; J_B = 0 & K_B = 1 therefore next state B = 0

5th Row: PS = 0; J_B & K_B = 1 therefore next state B = B'=1

6th Row: PS = 0; J_B & K_B = 0 therefore next state B = 0: NC

7th Row: PS = 1; J_B & K_B = 1 therefore next state B = B'= 0

8th Row: PS = 1; J_B & K_B = 0 therefore next state B = 1: NC

Thus we have completed the State table by filling up next state values based on the present state & flip-flop inputs.

Analysis using Flip - Flops

Determination of next state values using Characteristic Equation:

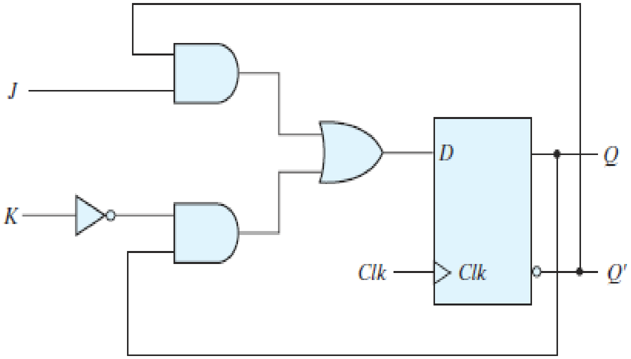
The characteristic equation for JK flip flop is given by:

$$Q(t+1) = J\,Q' + K'\,Q$$

For each flip flop ‘A’ & ‘B’, we write the respective Characteristic equations as:

$$A(t+1) = J\,A' + K'\,A$$

$$B(t+1) = J\,B' + K'\,B$$



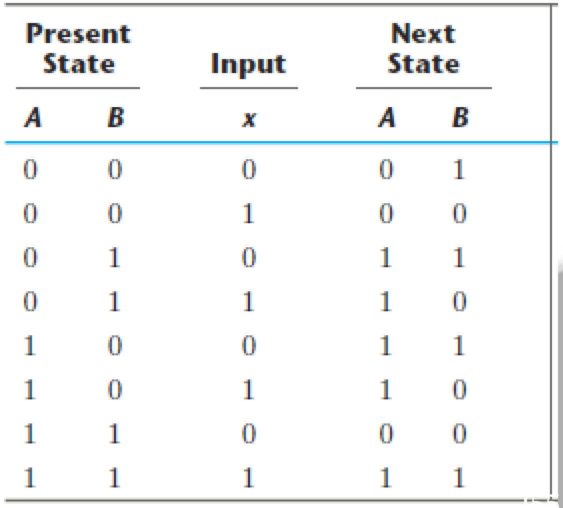
(J-K Flip Flop)

Since the output states in our circuit are designated as ‘A’ & ‘B’ instead of ‘Q’.

Substituting the values of $J = J_A$ and $K = K_A$ from the input equations, $J_A = B$ & $K_A = B\,x'$, we obtain the state equation for A :

$$A(t+1) = BA' + (Bx')'A$$

or, $A = A'B + AB' + Ax$ 1



Present State		Input	Next State	
A	B	x	A	B
0	0	0	0	1
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

The state equation provides the bit values for the column headed “Next State” for A in the state table. Therefore by using Eq. 1 we can fill up the ‘Next State’ column corresponding to ‘A’

Analysis using Flip - Flops

Similarly, the state equation for flip-flop B can be derived from the characteristic equation by substituting the values of J_B and K_B:

$$B(t + 1) = J\,B' + K'\,B\,;$$

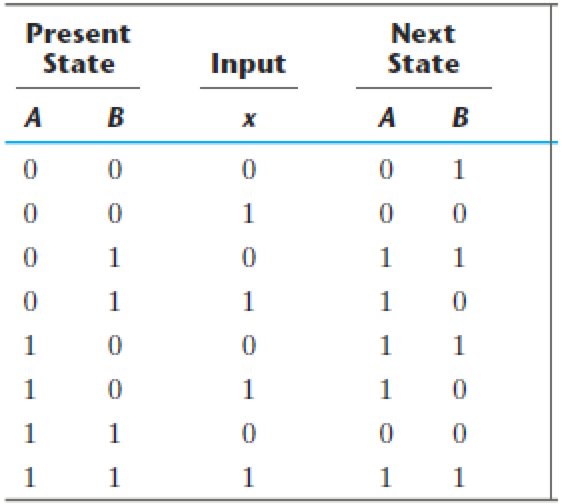
where, $J = J_B = x'$ & $K = K_B = A'x + A\,x' = A \oplus x$

We get the state equation for B as:

$$B(t+1) = x'B' + (A \oplus x)'B = B'x' + AB\,x + A'B\,x';$$

$$(A \oplus x)' = A\,x + A'x'$$

or, $B = B'x' + AB\,x + A'B\,x'$ 2



Present State		Input	Next State	
A	*B*	*x*	*A*	*B*
0	0	0	0	1
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

The state equation provides the bit values for the column headed ‘Next State’ for B in the state table. Therefore making use of Eq. 2 we fill up the ‘Next State’ column corresponding to ‘B’.

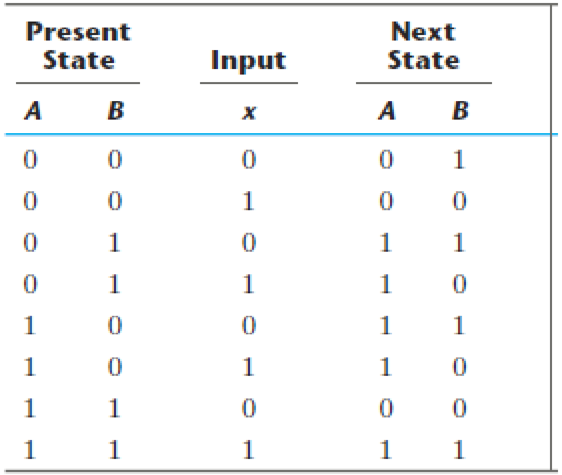
Note: The State table obtained using State equations does not have columns corresponding to “Flip-Flop Inputs” because in the obtained state equations the ‘J’ & ‘K’ inputs are implicit.

(contd. ….)

Analysis using Flip - Flops

State Diagram:

Next we draw the state diagram for the given circuit. There is no specified output for the given circuit & therefore in the state diagram the 'links'/'branches' do not indicate output of the system for a given (present) state.



Present State		Input	Next State	
A	*B*	*x*	*A*	*B*
0	0	0	0	1
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

In absence of any specified output we may choose the flip flop state itself as the system output.

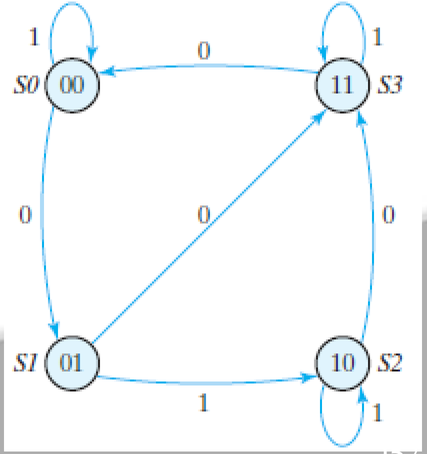
From the state table we see that the system has 4 states namely : 00 01 10 11. & input can be '0' or '1'.

We know that transition takes place at the occurrence of clock pulse.

So, transition from '00' to '01' takes place while x =0 and stays at '00' while x = 1.

Similarly we see for transitions from '01' '10' & '11' and also note down the input value to draw the state diagram as shown.

The directive links do not show any output value because it is not specified.



Analysis using Flip - Flops

Analysis using T Flip Flops:

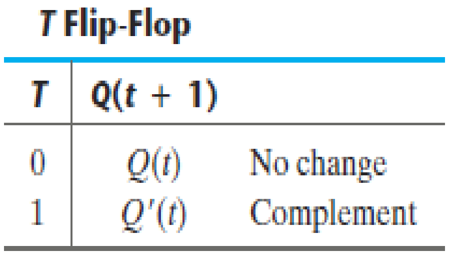
The analysis of a sequential circuit with T flip-flops follows the same procedure outlined for JK flip-flops.

The next-state values in the state table can be obtained by using

- either the characteristic table as listed , or
- the characteristic equation given by

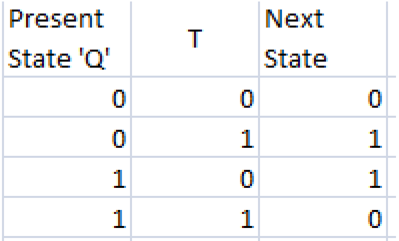
$$Q(t + 1) = T \oplus Q = T'Q + TQ' \quad \text{..... 1}$$

This equation is easily derivable from the following state table of T flip flop. We observe that 'next state' is XOR of present state , Q and the T value. Therefore the characteristic equation is given by Eq. 1



***T* Flip-Flop**

T	*Q*(*t* + 1)	
0	$Q(t)$	No change
1	$Q'(t)$	Complement



Present State 'Q'	T	Next State
0	0	0
0	1	1
1	0	1
1	1	0

State Table

Example:

We will consider a circuit having 2 T flip flops.

Analysis using Flip - Flops

Example:

Consider the sequential circuit as shown. It has

- two T flip flops A and B,
- one input x, and
- one output y

It can be described algebraically by two input equations: $T_A = B\,x$; $T_B = x$, and

an output equation: $y = AB$

The characteristic equations are:

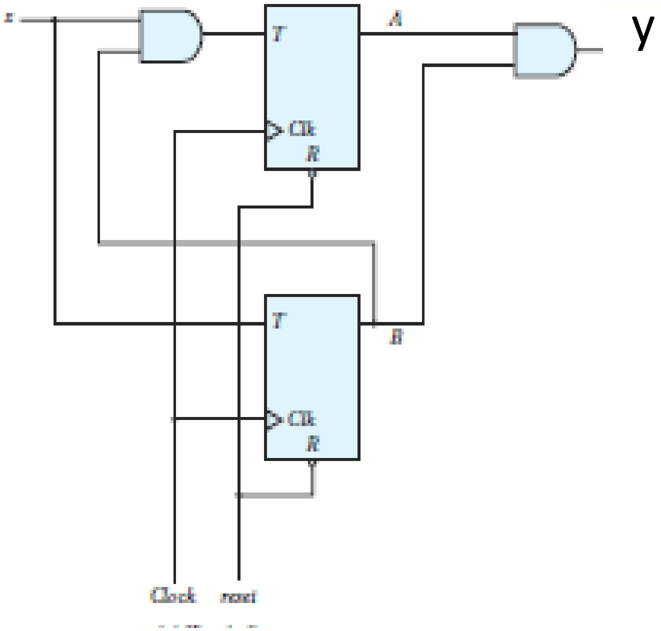
$$A(t+1) = T'_A\,A + A'\,T_A\ ;$$

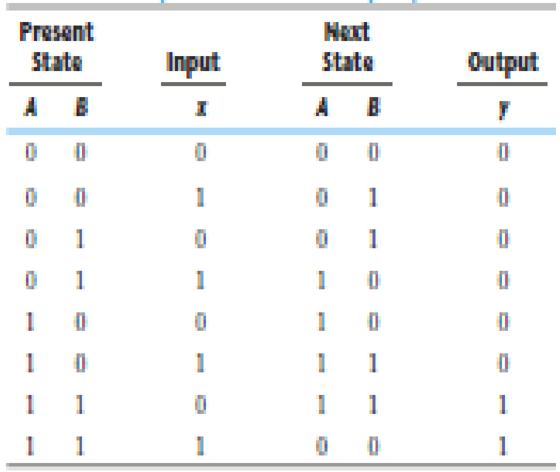
$$A = (B\,x)'\,A + A'\,(B\,x) = AB' + Ax' + A'\,B\,x$$

$$B(t+1) = T'_B\,B + B'\,T_B$$

$$B = x'\,B + B'\,x = x \oplus B$$

The next-state values for A and B in the state table are obtained from the expressions of the two state equations. This is characteristic equation method.





State Table for Sequential Circuit with T Flip-Flops

Present State		Input	Next State		Output
A	B	x	A	B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	1

Analysis using Flip - Flops

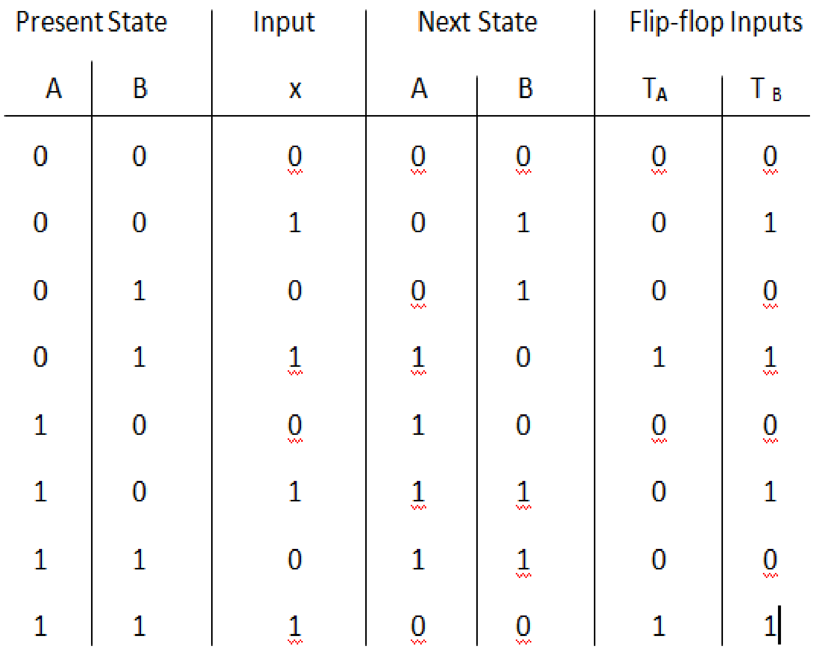
Characteristic (State) Table Method:

In this method we draw the state table which includes flip flop input values (based on present state & input value), in addition to columns for ‘present state’, ‘input’ & ‘next state’.

The state table is as shown.

Flip flop inputs are determined from their Input equations.

Based on TA & TB values, the ‘next state’ is determined from the ‘present state’ , as explained in the earlier example.



Present State		Input	Next State		Flip-flop Inputs	
A	B	x	A	B	T_A	T_B
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	0	1	0	0
0	1	1	1	0	1	1
1	0	0	1	0	0	0
1	0	1	1	1	0	1
1	1	0	1	1	0	0
1	1	1	0	0	1	1

State Diagram:

From the state table, states are 00 01 10 11

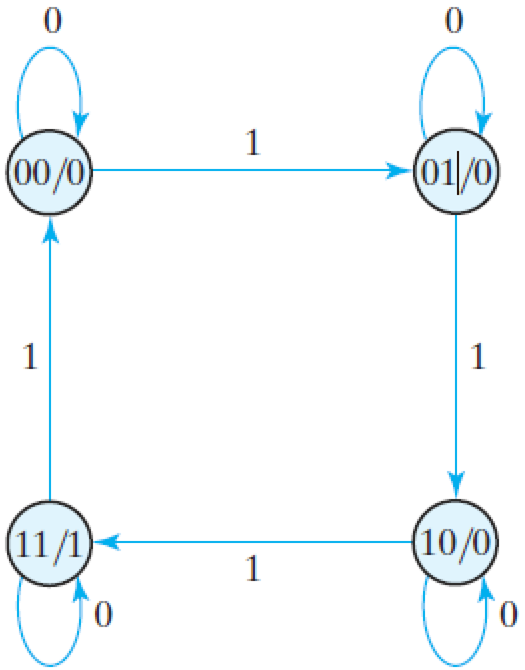
Input & output values are given.

Transition takes place at the occurrence of clock pulse.

The state diagram is shown in the next slide

Analysis using Flip - Flops

State Diagram:



State Reduction

Analysis & Design of Sequential Circuits:

Analysis:

The analysis of sequential circuits starts from a circuit (logic) diagram and culminates in a state table or diagram.

Design:

The design (synthesis) of a sequential circuit starts from a set of specifications and culminates in a logic (circuit) diagram.

State Reduction: Its requirement

Two sequential circuits may exhibit the same input–output behaviour, but have a different number of internal states in their state diagram.

We will discusses certain properties of sequential circuits that may simplify a design by reducing the number of gates and flip-flops it uses.

In general, reducing the number of flip-flops reduces the cost of a circuit.

So the purpose behind achieving reduction in number of states defining a sequential circuit is to reduce the complexity of hardware design.

State Reduction

State Reduction:

The state reduction problem is about reducing the number of flip-flops in a sequential circuit. Lesser the number of states lesser the number of flip-flops.

While trying to achieve reduction, it is ensured that the input-output requirements remain unaltered.

Since m flip-flops produce 2^m states, a reduction in the number of states may (or may not) result in a reduction in the number of flip-flops.

For example, if m = 3; number of states = 8. We may be able to reduce number of states to 6, but that does not change number of flip-flops because for ‘6’ states ‘m’ has to be ‘3’.

An unpredictable effect in reducing the number of flip-flops is that sometimes the equivalent circuit (with fewer flip-flops) may require more combinational gates to realize its next state and output logic.

The procedure for achieving state reduction is illustrated with the help of examples.

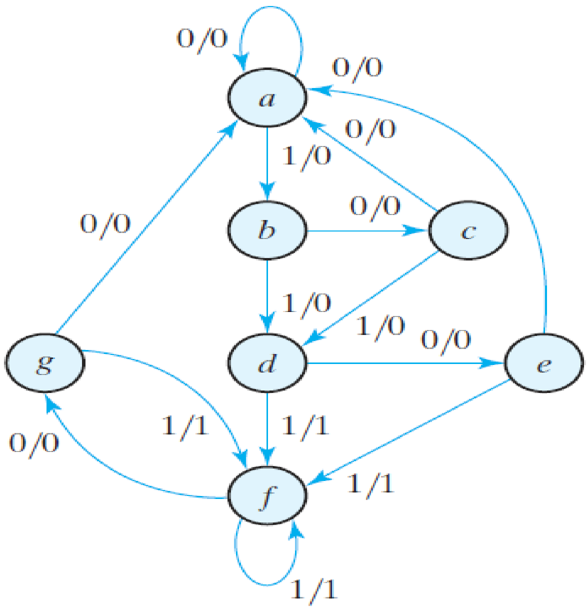
State Reduction

Example:

For achieving reduction in the number of states, the starting point is 'state diagram'. A typical state diagram is shown in the figure.

Since only input-output sequences are important, for that reason, the states marked inside the circles are denoted by letter symbols instead of their binary values. This is in contrast to a binary counter, wherein the binary value sequence of the states themselves is taken as the outputs.

There are an infinite number of input sequences that may be applied to the circuit; each results in a unique output sequence.



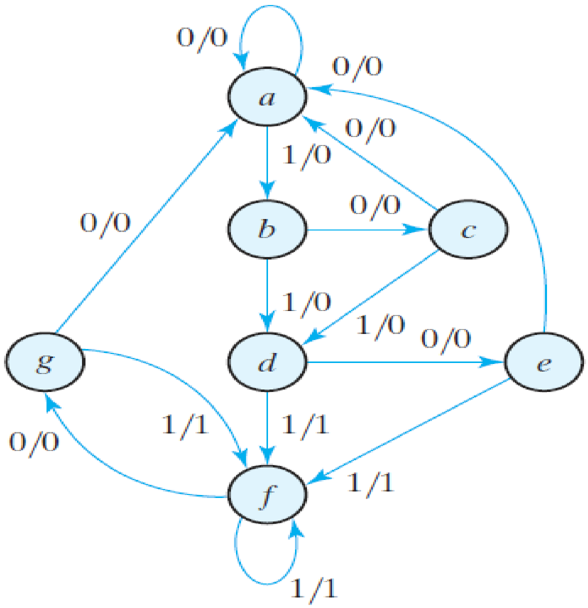
We see from the diagram that each input of '0' or '1' produces an output of '0' or '1' and causes the circuit to go to next state.

Consider an input sequence 01010110100 starting from the initial state 'a' and extending up to state 'g'. We could have considered just any other sequence.

State Reduction

Determination of Output & State sequence:

From the state diagram, we obtain the output and state sequence for the given input sequence as follows: With the circuit in initial state ‘a’, an input of ‘0’ produces an output of ‘0’ and the circuit remains in state ‘a’ .



With present state ‘a’ and an input of ‘1’, the output is ‘0’ and the next state is ‘b’ .

With present state ‘b’ and an input of ‘0’, the output is ‘0’ and the next state is ‘c' .

Continuing this process, we find the complete sequence to be as follows:

state	*a*	*a*	*b*	*c*	*d*	*e*	*f*	*f*	*g*	*f*	*g*	*a*
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

The ↗ indicates the transition from one state to another. Column at beginning end of the ↗ denote input/output values for which a transition occurs/does not occur. While tracing the diagram we have followed input sequence bit pattern.

State Reduction

We reassert that in this circuit the states themselves are of secondary importance, because we are interested only in output sequences caused by input sequences.

Equivalent Sequential Circuits:

Two sequential circuits are equivalent if their respective output sequences match (are identical) for a defined set of (identical) input sequences; while the number of states in each one of them may be different (less or more w.r.t each other)

Therefore the problem of state reduction is to find ways of reducing the number of states in a sequential circuit without altering the input–output relationships.

Procedure for State Reduction:

It is more convenient to apply procedures for state reduction with the use of a table rather than a diagram. The state table of the circuit is listed in next slide and is obtained directly from the state diagram.

State Reduction

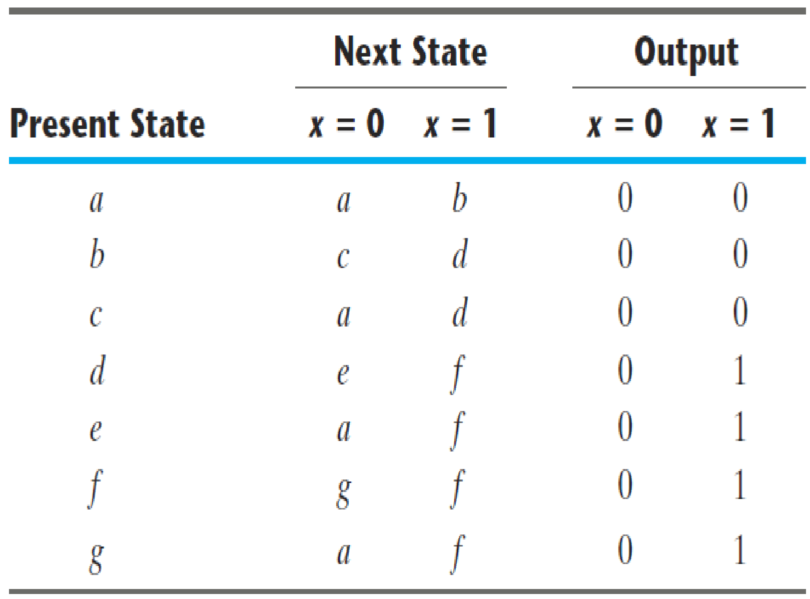
Equivalence of States:

The equivalence of states is given by the following algorithm:

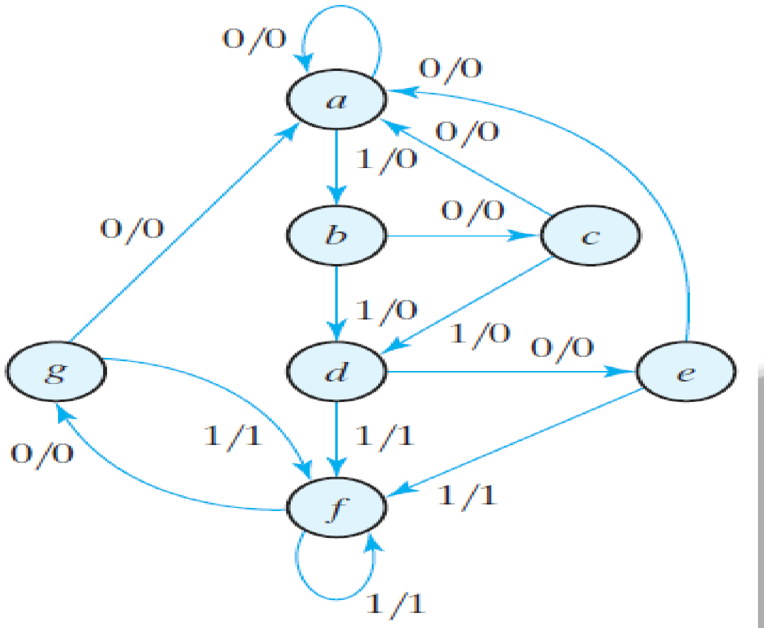
“Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an equivalent state.”

When two states are equivalent, one of them can be removed without altering the input–output relationships.

We will apply the above algorithm to our state table for the purpose of achieving reduction.



	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

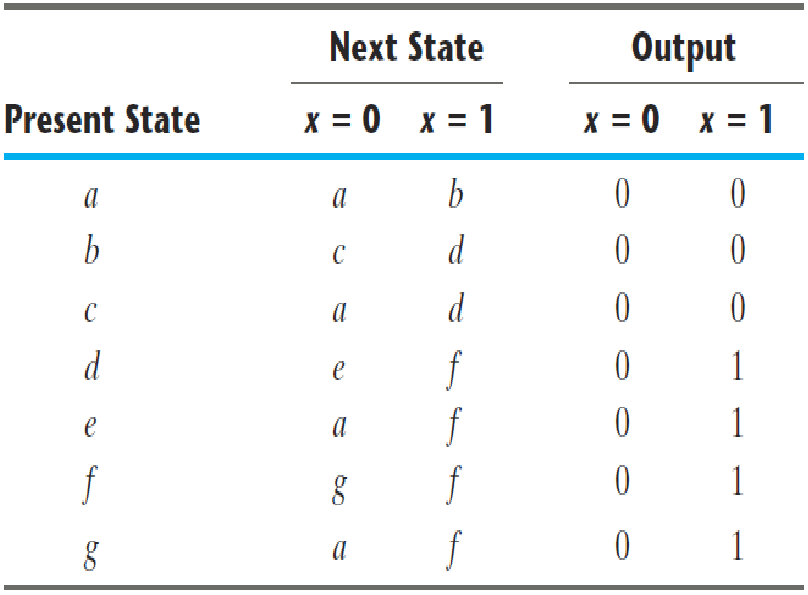


State Reduction

The state table is reproduced.

The procedure is that in the state table we look for two present states that go to the same next state and have the same output for both input combinations.

From the table we see that two such states are ‘g’ & ‘e’. They both go to states ‘a’ and ‘f’ and have outputs of ‘0’ and ‘1’ for x = ‘0’ and x = ‘1’, respectively.



	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*f*	0	1
e	*a*	*f*	0	1
f	*g*	*f*	0	1
g	*a*	*f*	0	1

Therefore, states ‘g’ and ‘e’ are equivalent and one of these states can be removed.

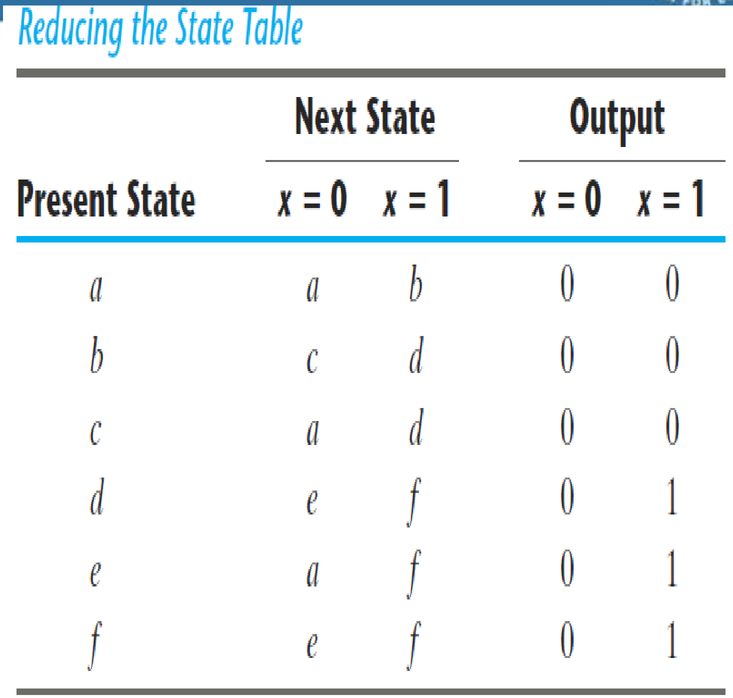
The procedure for removal is that we remove ‘g’ under column ‘present state’ from the table & replace ‘g’ by ‘e’ in the columns under ‘next state’ wherever it appears.

So we have a reduced table in which the row corresponding to ‘g’ does not exist. The reduced table is shown in the next slide.

State Reduction

The reduced state table is shown.

Again in the reduced state table we look for two present states that go to the same next state and have same outputs for both input combinations. From the table we see that two such states are ‘d’ & ‘f’. They both go to states ‘e’ and ‘f’ and have outputs of ‘0’ and ‘1’ for x = ‘0’ and x = ‘1’, respectively.



Reducing the State Table

	Next State		Output	
Present State	x = 0	x = 1	x = 0	x = 1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

Therefore, states ‘d’ and ‘f’ are equivalent and one of these states can be removed.

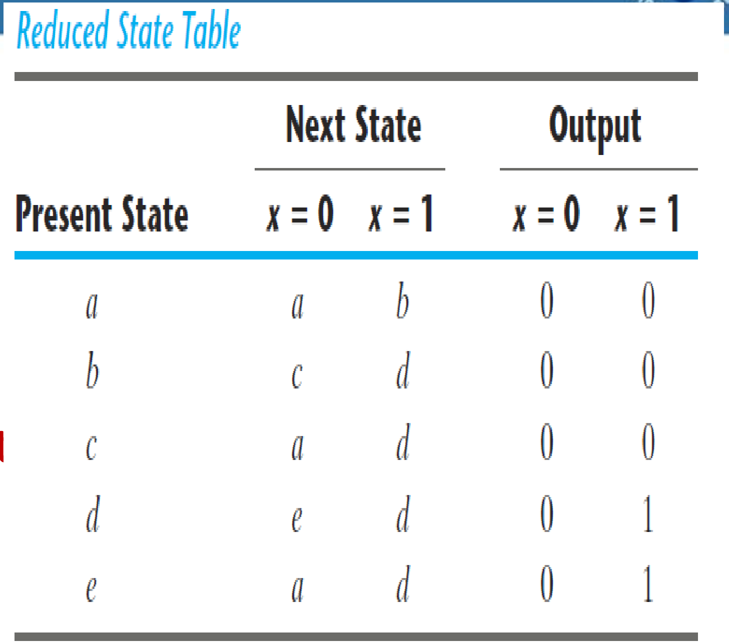
The procedure for removal is that we remove ‘f’ under column ‘present state’ from the table & replace ‘f’ by ‘d’ in the columns under ‘next state’ wherever it appears.

So we have a further reduced table in which the row corresponding to ‘f’ does not exist. The reduced table is shown in the next slide.

State Reduction

The further reduced state table is shown.

Again in the reduced state table we look for two present states that go to the same next state and have same outputs for both input combinations. From the table we see that there are no two such states. Therefore further reduction of states is not possible.

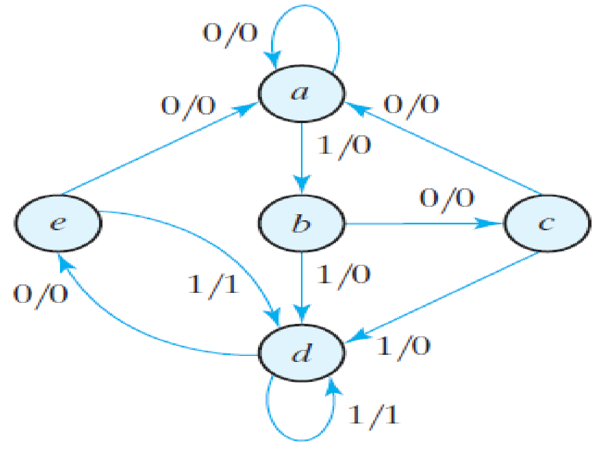


Reduced State Table

	Next State		Output	
Present State	x = 0	x = 1	x = 0	x = 1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

The reduced state diagram is as shown.

The sequential circuit of this example was reduced from seven to five states. In general, reducing the number of states in a state table may result in a circuit with less equipment. However, the fact that a state table has been reduced to fewer states does not guarantee a saving in the number of flip-flops or the number of gates. In actual practice designers may skip this step because target devices are rich in resources.



State Reduction

State Reduction using Implication Table:

As we know reduction is the problem of determining state equivalence so that redundant states can be deleted to achieve reduction in number of states.

The implication table method of determining state equivalence is as follows:

1. Construct a chart which contains a square for each pair of states.
2. Compare each pair of rows in the state table. If the outputs associated with states ‘I’ and ‘j’ are different, place an X in square i-j to indicate that $i \neq j$. If the outputs are the same, place the implied pairs in square i-j. (If the next states of i and j are m and n for some input x, then m-n is an implied pair.) If the outputs and next states are the same (or if i-j only implies itself), place a check (√) in square i-j to indicate that $i \equiv j$.
3. Go through the table square-by-square. If square i-j contains the implied pair m-n, and square m-n contains an X, then $i \neq j$, and an X should be placed in square i-j.
4. If any X’s were added in step 3, repeat step 3 until no more X’s are added.
5. For each square i-j which does not contain an X, $i \equiv j$.

If desired, row matching can be used to partially reduce the state table before constructing the implication table.

State Reduction

Example:

The state table is given. First step is to construct a chart containing squares.

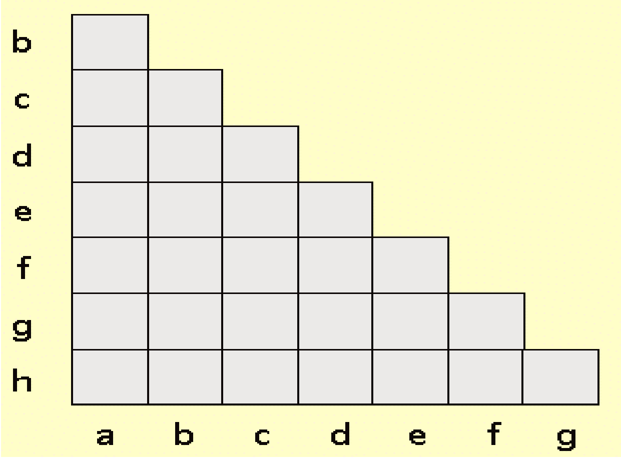
Rows of the chart:

Exclude row corresponding to state ‘a’. That means we exclude 1st state from rows.

Columns of the chart:

Exclude row corresponding to state ‘h’; exclude the ‘last’ state from the columns.

Present State	Next State $X = 0$	1	Present Output
a	*d*	*c*	0
b	*f*	*h*	0
c	*e*	*d*	1
d	*a*	*e*	0
e	*c*	*a*	1
f	*f*	*b*	1
g	*b*	*h*	0
h	*c*	*g*	1



In this method each row of the table is compared with remaining rows to determine such rows which have the same output; e.g. row ‘a’ is compared with rows ‘b’ to ‘h’ to find out that outputs of ‘a’, ‘b’, ‘d’, & ‘g’ are same.

State Reduction

Next ‘b’ is compared with ‘c’ to ‘h’; ‘c’ is compared with ‘d’ to ‘h’; ‘d’ is compared with ‘e’ to ‘h’ & so on.

So we get pairs like (a b) (ac) (ad) ….. (ah) ; (b c) (b d) (b e) …. (b h); & so on for all the rows in the given table.

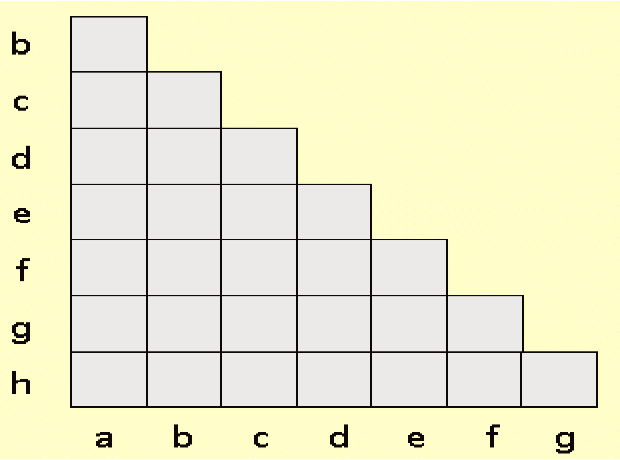
The chart represents these pairs (pair of states). We are interested in pair of states because we want to know whether they are equivalent or not.

1st row : (a b) ;

2nd row : (a c :1st square) & (b c: 2nd square)

3rd row: (a d) (b d) (c d); & so on. Thus we say,

A square in column i and row j corresponds to state pair i-j. Thus, the squares in the first column correspond to state pairs a-b, a-c, etc.



Note that the squares above the diagonal are not included in the chart because if $i \equiv j.$ & j Ξ i , and only one of the state pairs i-j and j-i is needed.

Also, squares corresponding to pairs a-a, b-b, etc., are omitted.

State Reduction

To fill in the first column of the chart,

We compare row ‘a’ of the Table with each of the other rows. Because the output for row ‘a’ is different than the output for row ‘c’, we place an X in the a-c square of the chart to indicate that . $a \not\equiv c.$

Similarly, we place X’s in squares a-e, a-f, and a-h to indicate that $a \not\equiv e, a \not\equiv f,$ and $a \not\equiv h$ of output differences.

States a and b have the same outputs,

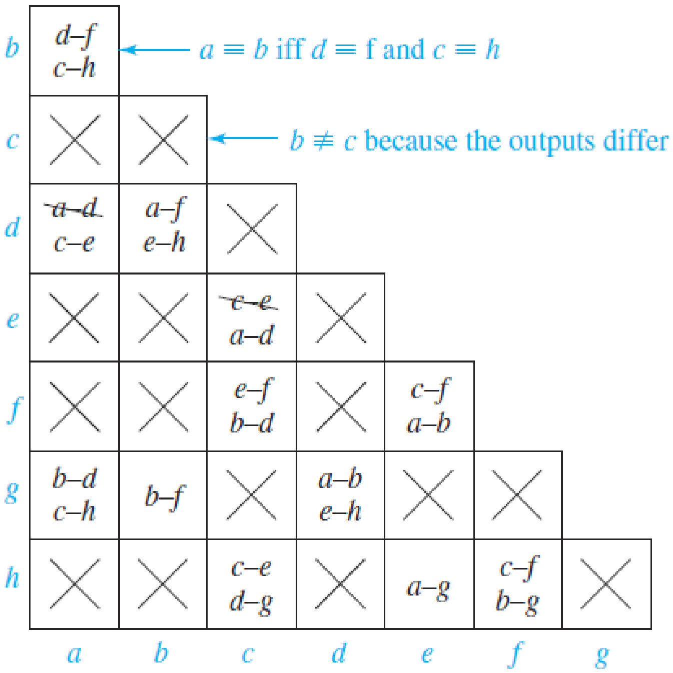
and thus, $a \equiv b$ iff $d \equiv f$ and $c \equiv h$

To indicate this, we place the implied pairs, d-f and c-h, in the a-b square.

Similarly, because a and d have the same outputs, we place a-d and c-e in the a-d square to indicate that $a \equiv d$ iff $a \equiv d$ and $c \equiv e$

The entries b-d and c-h in the a-g square indicate that:

$a \equiv g$ iff $b \equiv d$ and $c \equiv h$



b	d–f c–h						
c	X	X					
d	~~a–d~~ c–e	a–f e–h	X				
e	X	X	~~c–e~~ a–d	X			
f	X	X	e–f b–d	X	c–f a–b		
g	b–d c–h	b–f	X	a–b e–h	X	X	
h	X	X	c–e d–g	X	a–g	c–f b–g	X
	a	b	c	d	e	f	g

$a \equiv b$ iff $d \equiv$ f and $c \equiv h$

$b \not\equiv c$ because the outputs differ

(contd. ....)

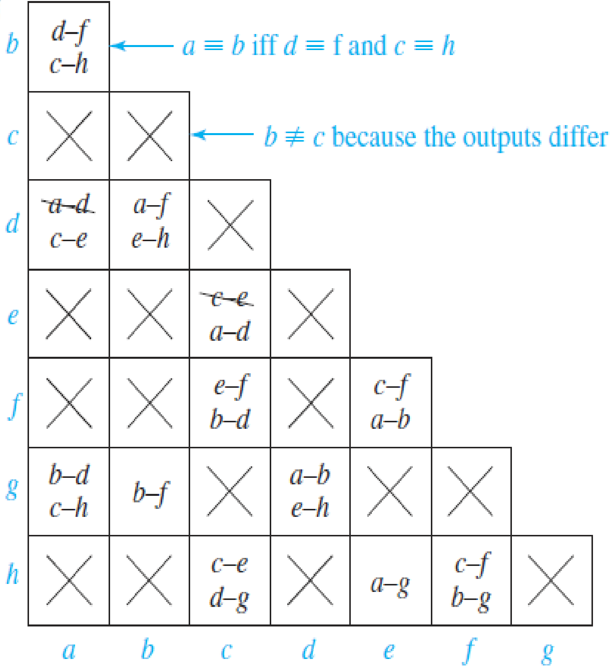
State Reduction

Next, row ‘b’ of the state table is compared with each of the remaining rows of the table, and column ‘b’ of the implication chart is filled in.

Similarly, the remaining columns in the chart are filled in to complete the chart.

Self-implied pairs are redundant, so a-d can be eliminated from square a-d, and c-e from square c-e.

Now, each square in the implication table has either been filled in with an X to indicate that the corresponding state pair is not equivalent (because the outputs are different) or filled in with implied pairs.



b	d–f c–h						
c	X	X					
d	~~a–d~~ c–e	a–f e–h	X				
e	X	X	~~c–e~~ a–d	X			
f	X	X	e–f b–d	X	c–f a–b		
g	b–d c–h	b–f	X	a–b e–h	X	X	
h	X	X	c–e d–g	X	a–g	c–f b–g	X
	a	b	c	d	e	f	g

← $a \equiv b$ iff $d \equiv$ f and $c \equiv h$

← $b \not\equiv c$ because the outputs differ

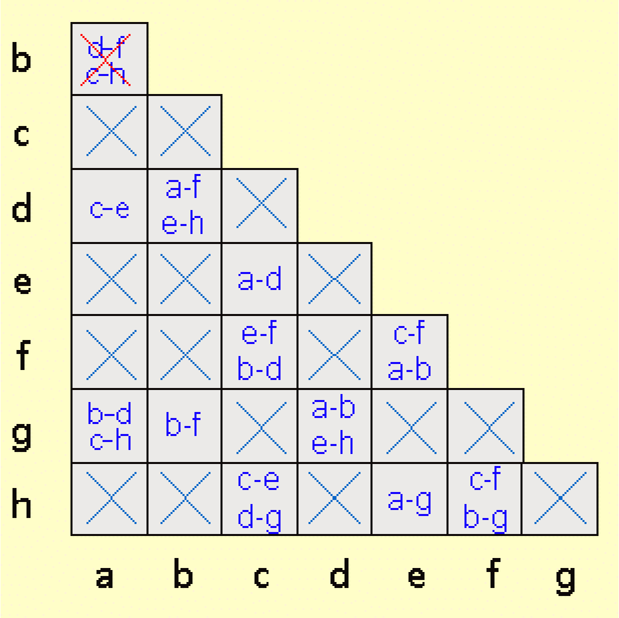
Check each implied pair:

We now check each implied pair. If one of the implied pairs in square i-j is not equivalent, then $i \neq j$.

The a-b square has 2 implied pairs: d-f & c-h; but d-f square in chart is marked X therefore $a \neq b$. Hence we put a X on a-b square.

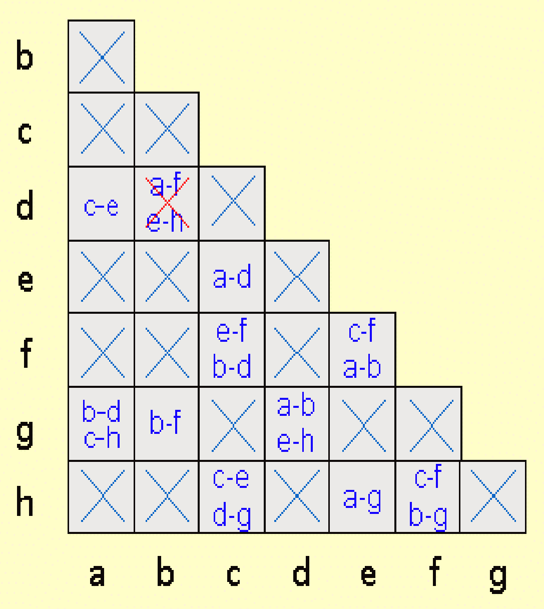
State Reduction

It is shown in the chart.



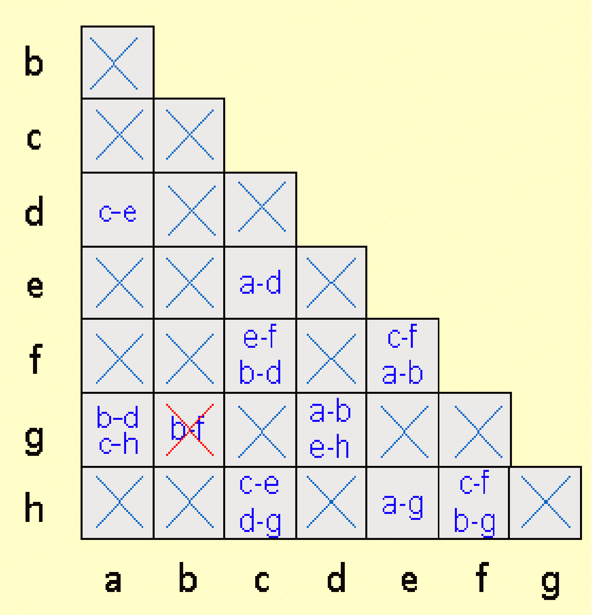
b	~~d-f c-h~~						
c	X	X					
d	c-e	a-f e-h	X				
e	X	X	a-d	X			
f	X	X	e-f b-d	X	c-f a-b		
g	b-d c-h	b-f	X	a-b e-h	X	X	
h	X	X	c-e d-g	X	a-g	c-f b-g	X
	a	b	c	d	e	f	g

b-d, c-h & c-e squares do not have a X therefore move to next column ‘b’.



b	X						
c	X	X					
d	c-e	~~a-f e-h~~	X				
e	X	X	a-d	X			
f	X	X	e-f b-d	X	c-f a-b		
g	b-d c-h	b-f	X	a-b e-h	X	X	
h	X	X	c-e d-g	X	a-g	c-f b-g	X
	a	b	c	d	e	f	g

We see square a-f has X & hence we put a X on b-d square.

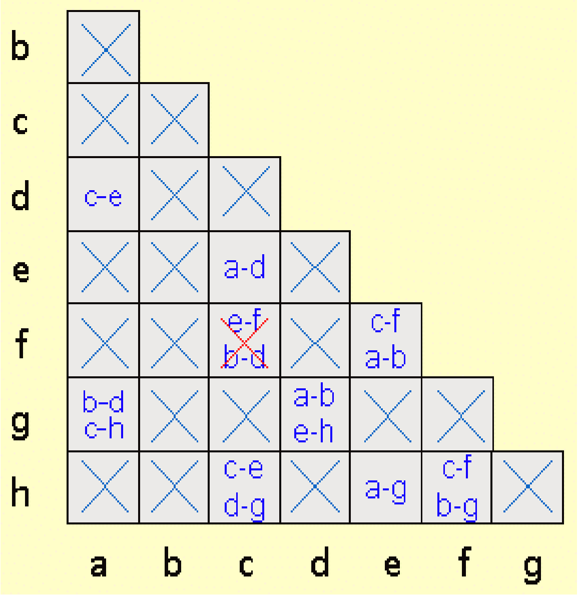


b	X						
c	X	X					
d	c-e	X	X				
e	X	X	a-d	X			
f	X	X	e-f b-d	X	c-f a-b		
g	b-d c-h	~~b-f~~	X	a-b e-h	X	X	
h	X	X	c-e d-g	X	a-g	c-f b-g	X
	a	b	c	d	e	f	g

square b-g has implied pair b-f, but square b-f has X in it therefore b-g square is crossed.

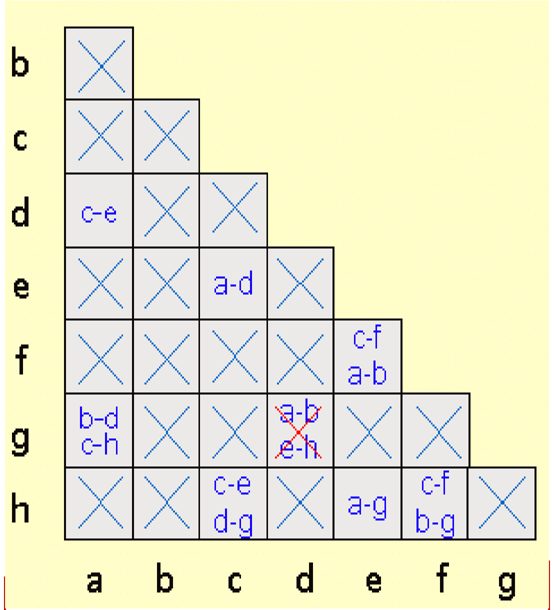
State Reduction

Move on to column c: explanation for putting X is similar as discussed for ‘a’ & ‘b’



	a	b	c	d	e	f	g
b	X						
c	X	X					
d	c-e	X	X				
e	X	X	a-d	X			
f	X	X	~~e-f~~ ~~b-d~~	X	c-f a-b		
g	b-d c-h	X	X	a-b e-h	X	X	
h	X	X	c-e d-g	X	a-g	c-f b-g	X

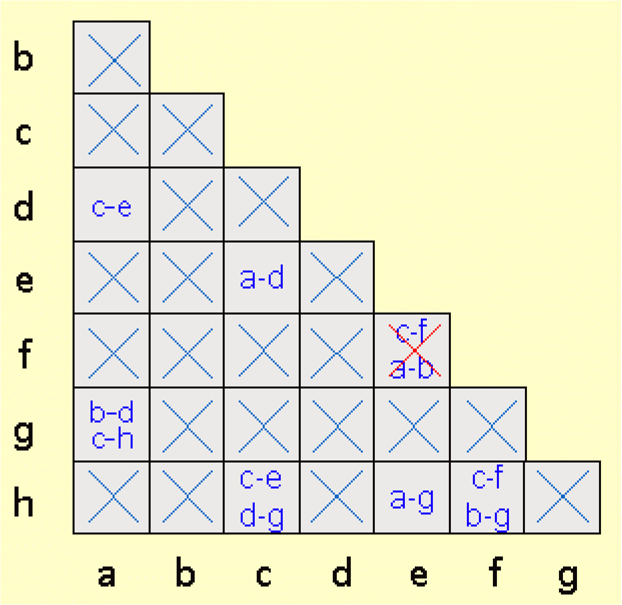
b-d square has X hence c-f square is crossed. Nothing can be said about squares c-h & c-e, so move on to column d.



	a	b	c	d	e	f	g
b	X						
c	X	X					
d	c-e	X	X				
e	X	X	a-d	X			
f	X	X	X	X	c-f a-b		
g	b-d c-h	X	X	~~a-b~~ ~~e-h~~	X	X	
h	X	X	c-e d-g	X	a-g	c-f b-g	X

Square a-b is crossed & hence square d-g is crossed.

Now, move on to column e.



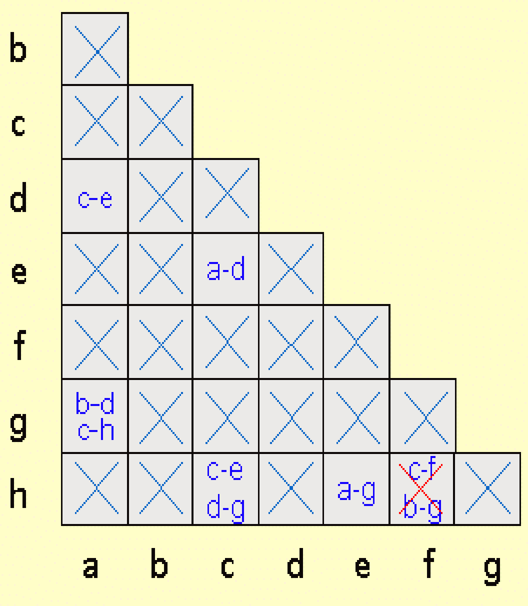
	a	b	c	d	e	f	g
b	X						
c	X	X					
d	c-e	X	X				
e	X	X	a-d	X			
f	X	X	X	X	~~c-f~~ ~~a-b~~		
g	b-d c-h	X	X	X	X	X	
h	X	X	c-e d-g	X	a-g	c-f b-g	X

Square a-b is crossed and square e-f is crossed.

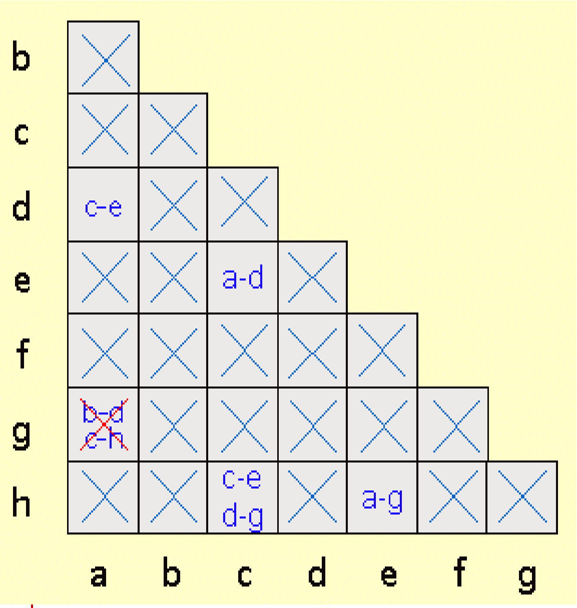
Move on to column f.

State Reduction

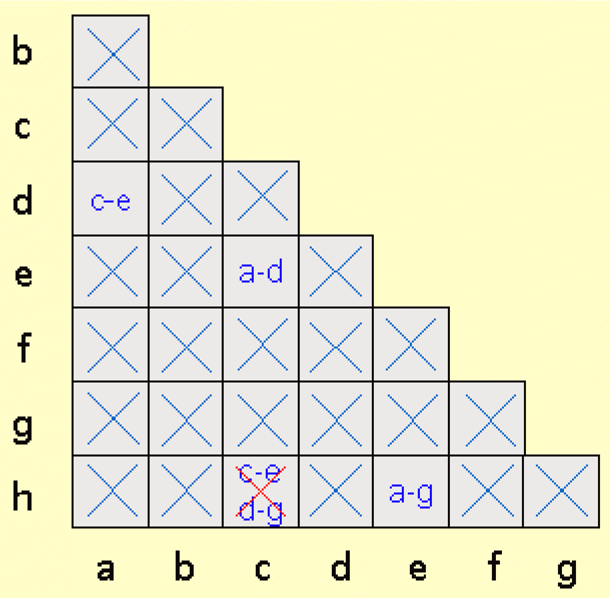
Move on to column f:



b	☒						
c	☒	☒					
d	c-e	☒	☒				
e	☒	☒	a-d	☒			
f	☒	☒	☒	☒	☒		
g	b-d c-h	☒	☒	☒	☒	☒	
h	☒	☒	c-e d-g	☒	a-g	~~c-f~~ ~~b-g~~	☒
	a	b	c	d	e	f	g



b	☒						
c	☒	☒					
d	c-e	☒	☒				
e	☒	☒	a-d	☒			
f	☒	☒	☒	☒	☒		
g	~~b-d~~ ~~c-h~~	☒	☒	☒	☒	☒	
h	☒	☒	c-e d-g	☒	a-g	☒	☒
	a	b	c	d	e	f	g



b	☒						
c	☒	☒					
d	c-e	☒	☒				
e	☒	☒	a-d	☒			
f	☒	☒	☒	☒	☒		
g	☒	☒	☒	☒	☒	☒	
h	☒	☒	~~c-e~~ ~~d-g~~	☒	a-g	☒	☒
	a	b	c	d	e	f	g

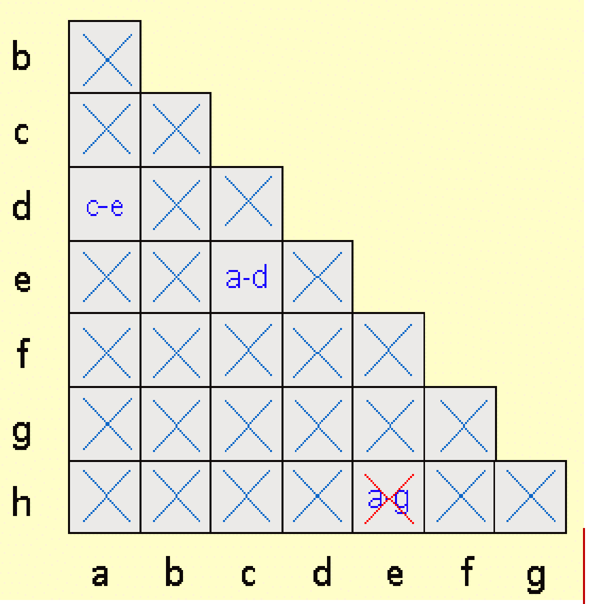
Square b-g has X in it, hence square f-h is crossed. Square h-g has X in it. We have completed one round, we again start from column ‘a’.

Since square b-d is crossed we cross square a-g. Move on to column c.

Square d-g is crossed We cross square c-h. Move on to column e.

State Reduction

Move on to column e:



b	✗						
c	✗	✗					
d	c-e	✗	✗				
e	✗	✗	a-d	✗			
f	✗	✗	✗	✗	✗		
g	✗	✗	✗	✗	✗	✗	
h	✗	✗	✗	✗	~~a-g~~	✗	✗
	a	b	c	d	e	f	g

The equivalent states are $a \equiv d$ and $c \equiv e$

Therefore we can eliminate 2 states if we replace d with a & e with c.

The reduced state table is reproduced below:

Present State	Next State X = 0	1	Output
a	*a*	*c*	0
b	*f*	*h*	0
c	*c*	*a*	1
f	*f*	*b*	1
g	*b*	*h*	0
h	*c*	*g*	1

Square a-g is crossed and hence we cross square e-h. No more squares to be considered for elimination. The process of finding Equivalent states has ended.

State Reduction & Assignment

Partition Method: (Moore Reduction Procedure)

States Si and Sj of machine M are said to be equivalent If and only if, for every possible input sequence, the same output sequence will be produced regardless of whether Si or Sj is the initial state.

Two states, Si and Sj, of machine M are distinguishable if and only if there exists at least one finite input sequence which, when applied to M, causes different output sequences depending on whether Si or Sj is the initial state.

The sequence which distinguishes these states is called a distinguishing sequence of the pair (Si , Sj)

If there exists for pair (Si , Sj) a distinguishing sequence of length k, the states in (Si , Sj) are said to be k-distinguishable .

States that are not k-distinguishable are said to be k-equivalent

State Reduction & Assignment

We seek to partition the states of machine M such that two equivalent states are in the same block.

P_0 corresponds to 0-distinguishablity (includes all states of machine M)

P_1 is obtained simply by inspecting the table and placing those states having the same outputs , under all inputs, in the same block.

P_1 establishes the sets of states which are 1-equivalent

P_2 partition is carried out by splitting blocks of P_1,whenever their successors are not contained in a common block of P_1

Iterate process of splitting blocks

If for some k, $P_{k+1} = P_k$, the process terminates and P_k defines the sets of equivalent states of the machine.

P_k is thus called the equivalence partition The equivalence partition is unique.

We will consider the same example that was solved using Implication method.

State Reduction & Assignment

Example:

Consider the state table:

PS: present state;   NS: next state

P_0 partition: = (ABCDEFGH)

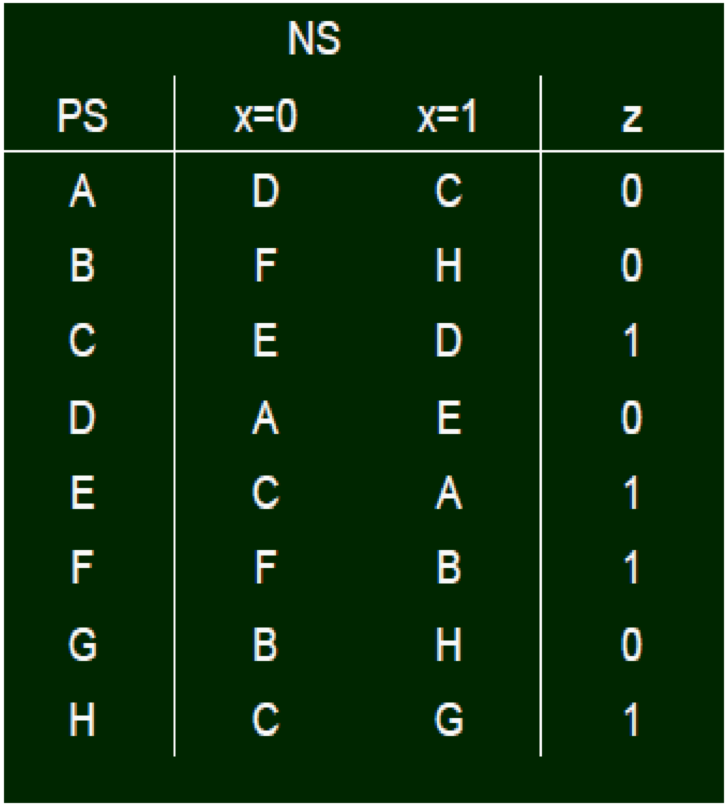
P_1 partition is obtained by splitting states having different outputs. Therefore, we have

P_1 =(ABDG)(CEFH)

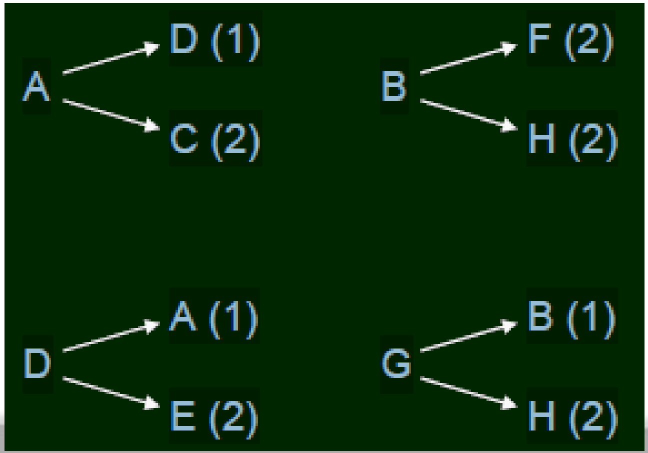
We define

Block 1 = ABDG, Block 2 = CEFH

Obtain P_2: (Consider Block 1 states)



	NS		
PS	x=0	x=1	z
A	D	C	0
B	F	H	0
C	E	D	1
D	A	E	0
E	C	A	1
F	F	B	1
G	B	H	0
H	C	G	1



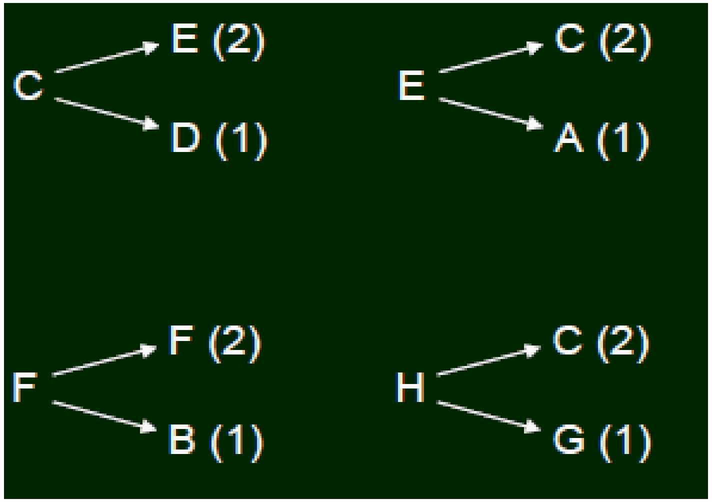
The ↗ indicate transition from one state to another. The numeral in () denotes the Block number. Like, D belongs to Block 1 & C to Block 2.

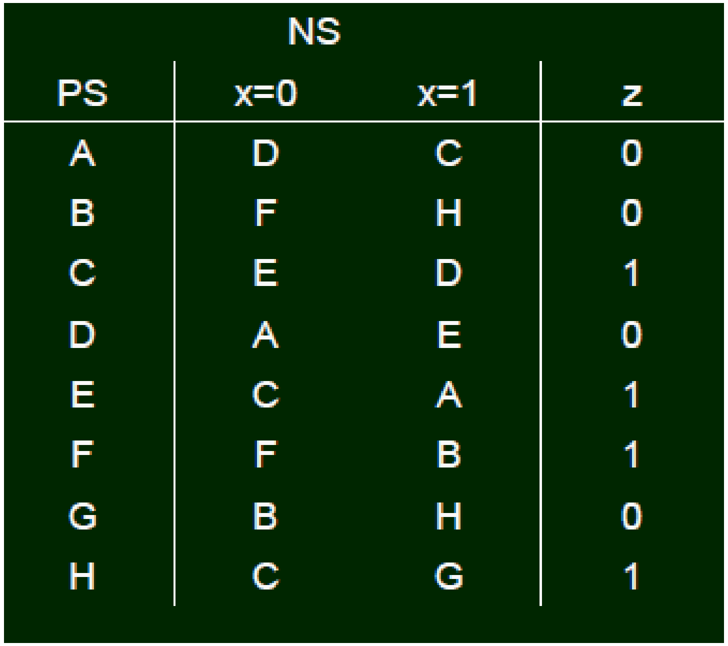
(contd. ....)

State Reduction & Assignment

Obtain P_2:

Consider Block 2 states (CEFH).





	NS		
PS	x=0	x=1	z
A	D	C	0
B	F	H	0
C	E	D	1
D	A	E	0
E	C	A	1
F	F	B	1
G	B	H	0
H	C	G	1

For the states A B D & G in block 1, we observe that state B does not follow transition pattern of states A D & G; because B makes transition to states of block 2 only whereas A D & G make transition to their respective states in both the blocks. So, we split B out of block 1. B is “2 distinguishable” from A, D and G, because it belongs neither to Block 1 nor Block 2. That is, it is distinguishable from these two blocks.

State Reduction & Assignment

No states of block 2 are “2 distinguishable”

Therefore, we factorize and obtain P_2 as:

P_2 = (ADG)(B)(CEFH)

We have removed B from Block 1. We redefine blocks as:

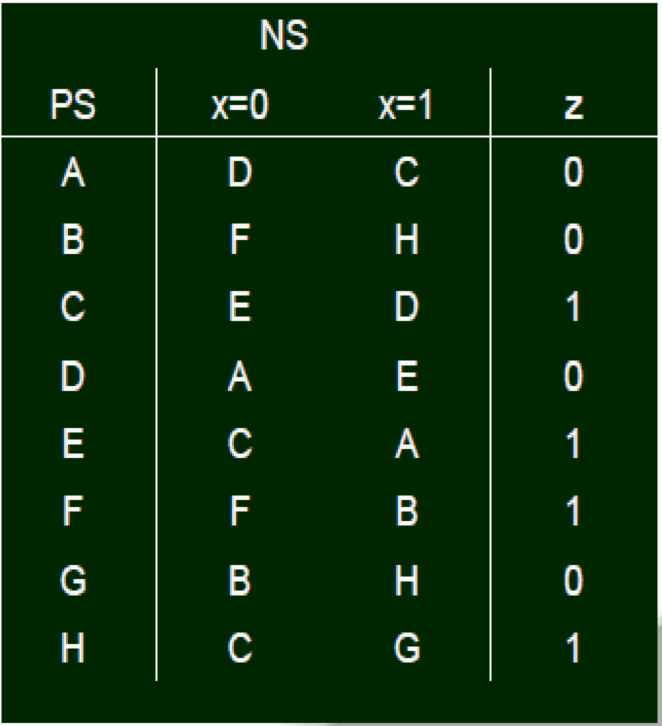
Block 1 = ADG

Block 2 = B

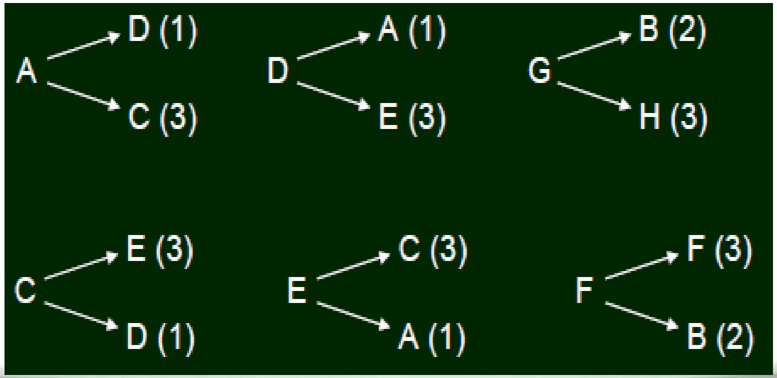
Block 3 = CEFH

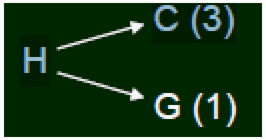
Obtain P_3:

Once again we transitions of states, now that we have 3 blocks.



	NS		
PS	x=0	x=1	z
A	D	C	0
B	F	H	0
C	E	D	1
D	A	E	0
E	C	A	1
F	F	B	1
G	B	H	0
H	C	G	1





We observe that G & F are 3 distinguishable.

State Reduction & Assignment

Obtain P_3 (contd.):

Split G from block 1; G is 3-distinguishable from A and D

Split F from block 3; F is 3-distinguishable from C, E and H

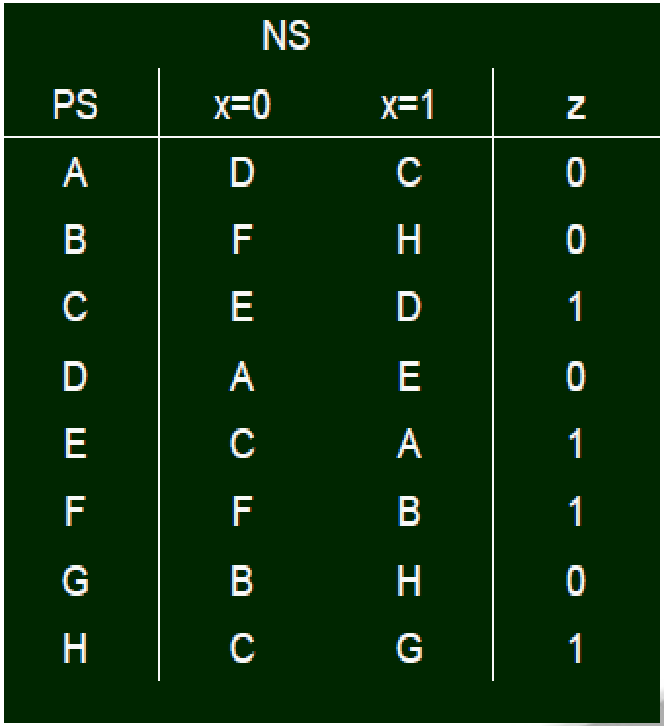
Therefore, factorize P_3 as:

$$P_3 = (AD)(G)(B)(CEH)(F)$$

Redefine the blocks as:

block 1 = AD, block 2 = G, block 3 = B,

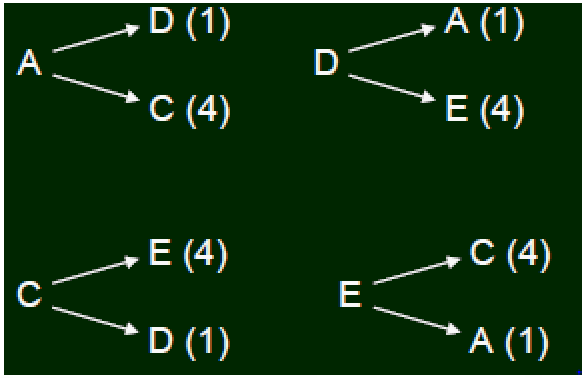
block 4 = CEH and block 5 = F

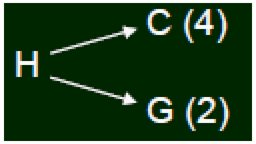


	NS		
PS	x=0	x=1	z
A	D	C	0
B	F	H	0
C	E	D	1
D	A	E	0
E	C	A	1
F	F	B	1
G	B	H	0
H	C	G	1

Obtain P_4:

Redefine state transitions for blocks 1 & 4 as shown.





From block 4, split H because it is 4 distinguishable.

$$P_4 = (AD)(G)(B)(CE)(H)(F)$$

State Reduction & Assignment

P_4 = (AD)(G)(B)(CE)(H)(F)

block 1 = AD, block 2 = G, block 3 = B,

block 4 = CE, block 5 = H and block 6 = F

Obtain P_5:

In view of the changed number of blocks we redefine the transitions in block 1 & 4 as shown in the figure.

No state is split-table from block 1 & 4 because in Each block the states make transitions to such other states which lie in the same block.

Since there is no splitting, therefore:

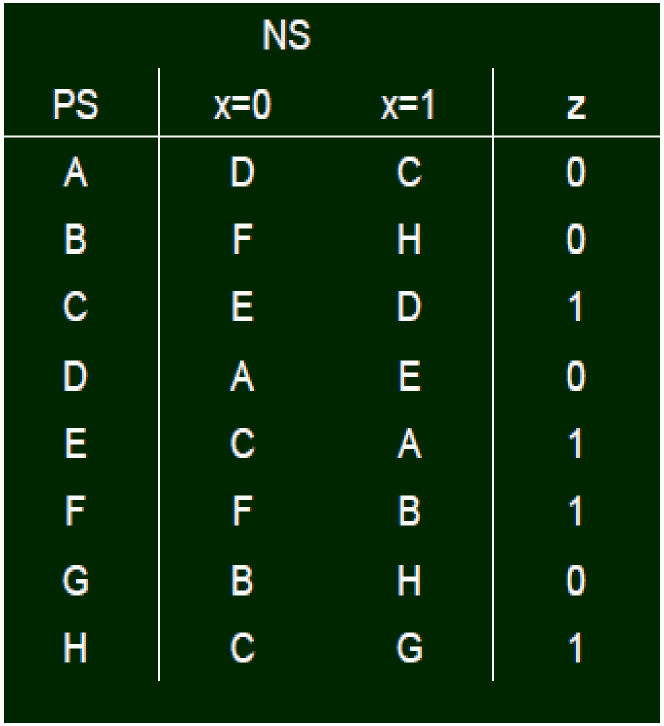
$P_5 = P_4$

Hence the factorization process stops at this stage.

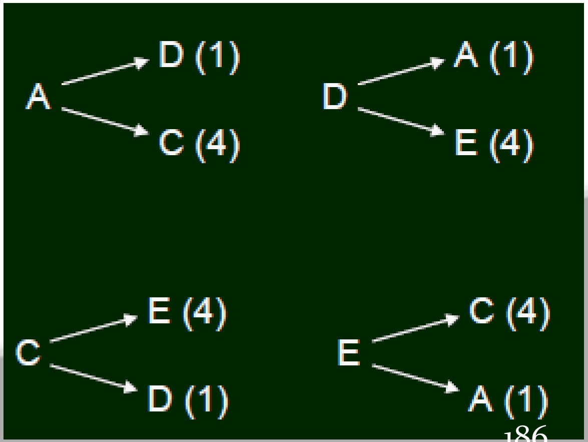
Therefore A Ξ D & C Ξ E

The minimized state function has the states

A B C F G & H ; reduction from 8 to 6 states



	NS		
PS	x=0	x=1	z
A	D	C	0
B	F	H	0
C	E	D	1
D	A	E	0
E	C	A	1
F	F	B	1
G	B	H	0
H	C	G	1



UNIT – IV
LOGIC FAMILIES AND SEMICONDUCTOR MEMORIES

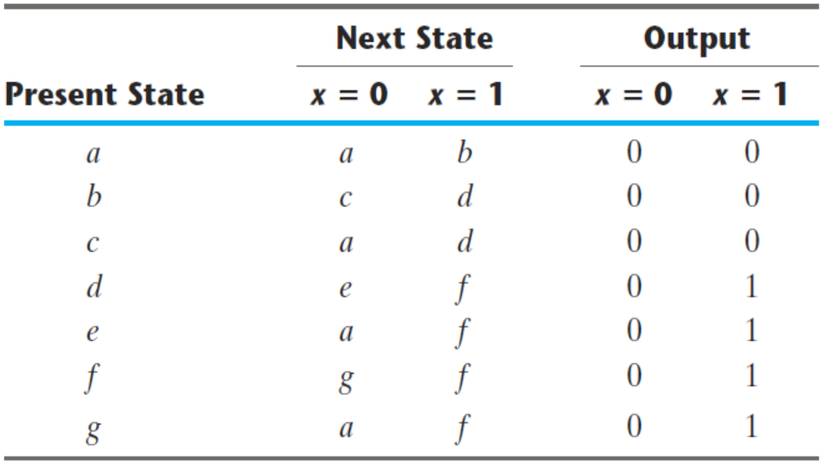
State Assignment

State Assignment:

Before designing a sequential circuit with physical components, we assign unique coded binary values to the states.

For a circuit with m states, the codes must contain n bits, where $2^n \geq m$. The equality (=) sign will hold if reduction in number of states is not possible else, greater than (>) sign holds.

For example, with three bits, it is possible to assign codes to eight states, denoted by binary numbers 000 through 111.



	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*f*	0	1
e	*a*	*f*	0	1
f	*g*	*f*	0	1
g	*a*	*f*	0	1

Table 1

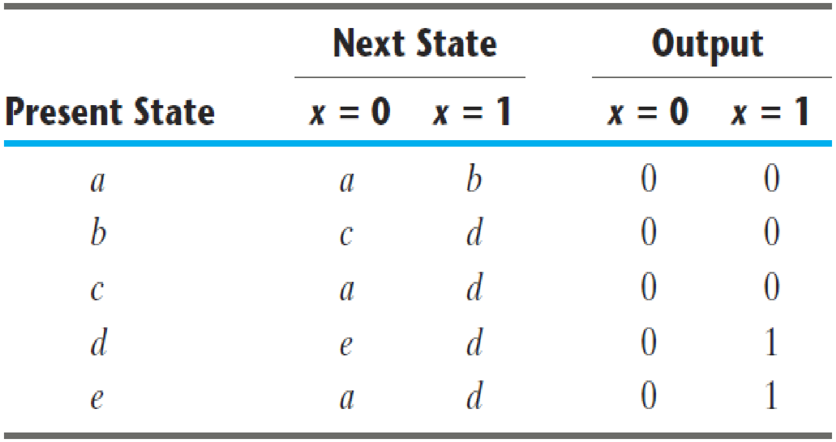
If we have this state table & we have 3 bits available for coding of states; the state ‘g’ will be left unused.

State Assignment

On the other hand if we use reduced version of the Table 1, we have the state table as Table 2.

In this case we have only 5 states and since 3 bits are required for coding, we are left with 3 unused states.

Unused states are treated as don’t-care conditions during the design.



	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*d*	0	1
e	*a*	*d*	0	1

Table 2

Since don’t-care conditions usually help in obtaining a simpler circuit, it is more likely but not certain that the circuit with five states will require fewer combinational gates than the one with seven states.

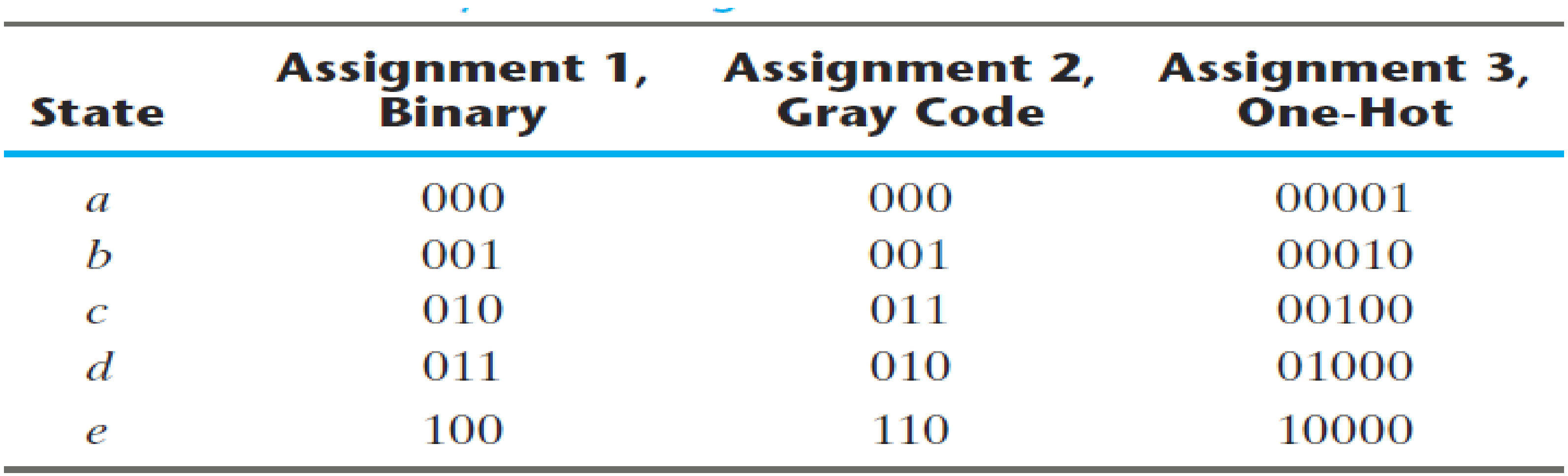
We can assign different types of codes to the states of a sequential circuit. Like, we assign binary code or Gray code or 1 Hot code to the states. The simplest way to code five states is to use the first five integers in binary counting order

Another similar assignment is the Gray code. Here, only one bit in the code group changes when going from one number to the next. This code makes it easier for the Boolean functions to be placed in the map for simplification.

State Assignment

Another possible assignment is 'one-hot' assignment. This configuration uses as many bits as there are states in the circuit. At any given time, only one bit is equal to '1' while all others are kept at '0'. This type of assignment uses one flip-flop per state, which is not an issue for register-rich field-programmable gate arrays.

Table 3 shows all the three types of assignments.



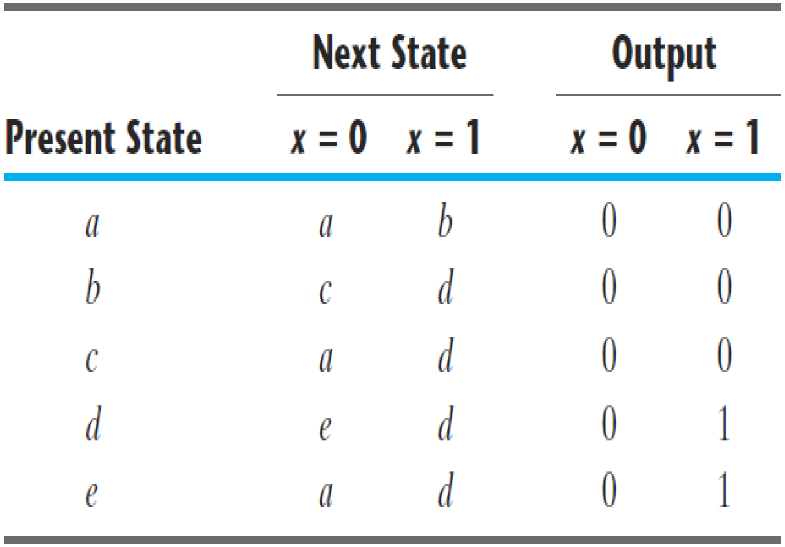
State	Assignment 1, Binary	Assignment 2, Gray Code	Assignment 3, One-Hot
a	000	000	00001
b	001	001	00010
c	010	011	00100
d	011	010	01000
e	100	110	10000

Table 3

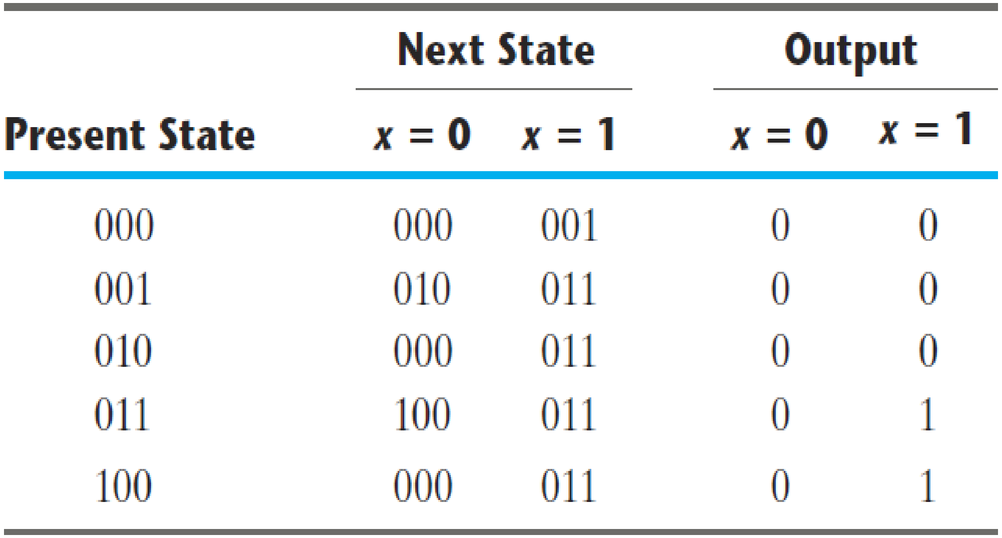
Having decided upon the type of assignment, we fill up the state table with the chosen code.

State Assignment

The Table 4 shows the state table filled up with chosen set of codes. We know that during state minimization we designate states of a circuit using alphabets. The process of assignment assigns a code to each state (named as an alphabet).



	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1



	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
000	000	001	0	0
001	010	011	0	0
010	000	011	0	0
011	100	011	0	1
100	000	011	0	1

Reduced State Table (Table 4) Reduced State (Transition) Table with Binary Assignment

A different assignment will result in a state table with different binary values for the states. The binary form of the state table is used to derive the next state and output-forming combinational logic part of the sequential circuit. The complexity of the combinational circuit depends on the binary state assignment chosen.

Mealy & Moore Models

Introduction:

A sequential circuit has inputs, outputs, and internal states. The sequential circuits are classified as:

- Moore model.
- Mealy model

They differ only in the way the output is generated.

Mealy Model:

In this model, the output is a function of both the present state and the input.

Moore Model:

In this model, the output is a function of only the present state.

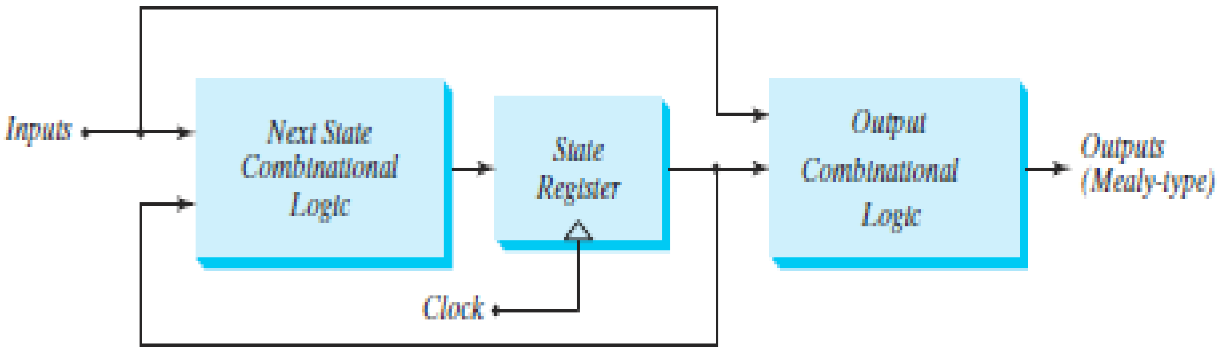
A circuit may have both types of outputs.

The two models of a sequential circuit are commonly referred to as a finite state machine (FSM).

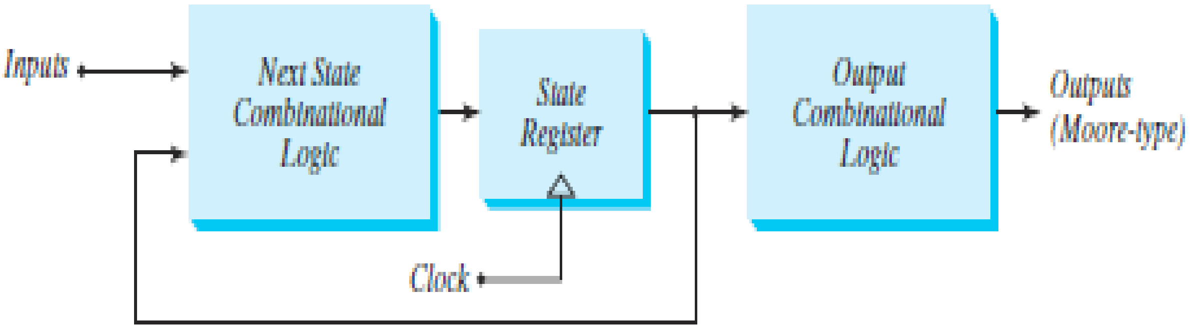
The Mealy model of a sequential circuit is referred to as a Mealy FSM or Mealy machine. The Moore model is referred to as a Moore FSM or Moore machine.

Mealy & Moore Models

Mealy and Moore machines are shown in the figures.



Mealy Machine



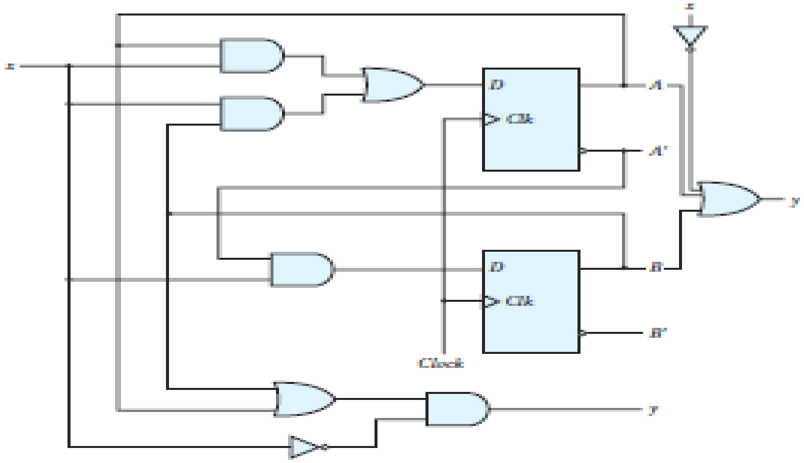
Moore Machine

We observe the difference in the output dependence of the two machines.

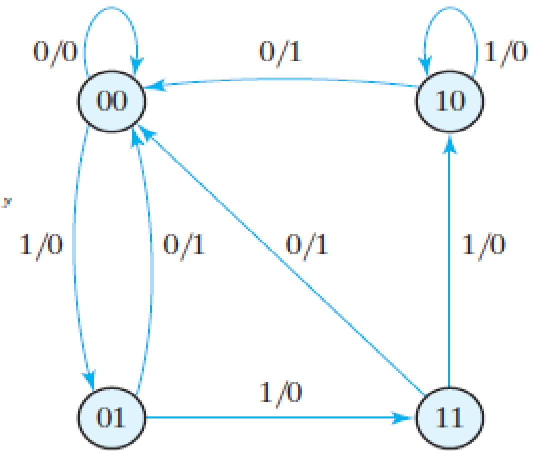
Mealy & Moore Models

Example of Mealy Machine:

The output functions are defined; so the directing links carry information about input/output.



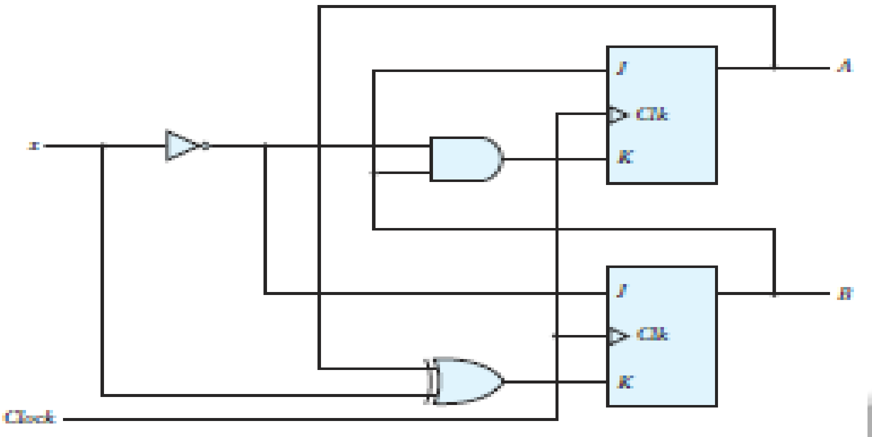
Machine



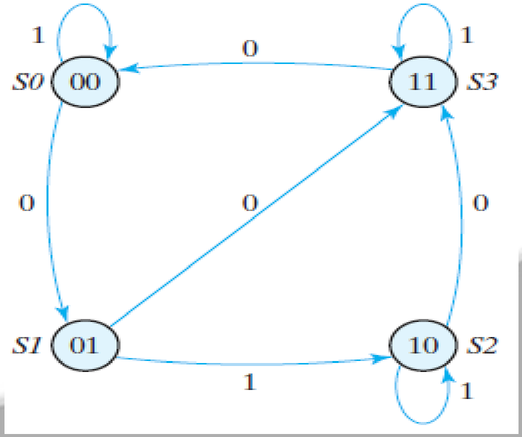
State Diagram

Example of Moore Machine:

The output function is not Defined; so the States of flip-flops are equivalent outputs.



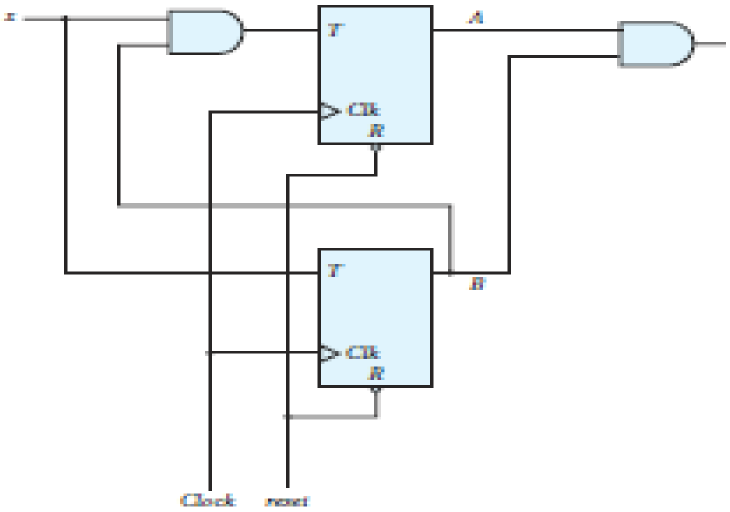
Machine

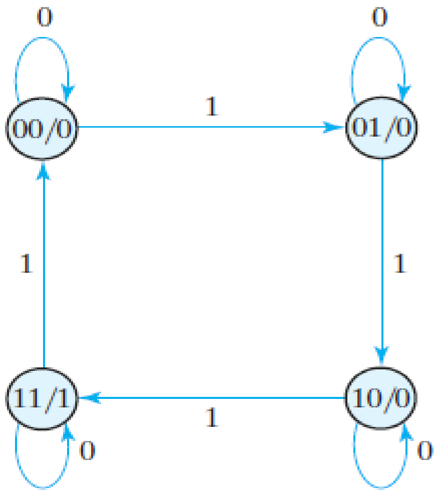


State Diagram

Mealy & Moore Models

Another Example of Moore Machine:





In comparison to previous example, in this machine the output is obtained by ANDing the outputs of 2 flip-flops. The output is a function of present state only. The directive links in the state diagram show only input values, whereas the outputs for different combinations of flip-flop states are depicted inside the circles (depicting states).

We see that the output shall be ‘1’ iff states of both the flip-flops are =1.

Mealy & Moore Models

Mealy Vs. Moore Machine:

Mealy	Moore
Output is a function of both the present state and the input.	Output is a function of present state Only.
Output may change if the input changes during clock cycle.	The outputs of the sequential circuit are synchronized with the clock, because they depend only on flip-flop outputs that are synchronized with the clock.

Mealy & Moore Models

Concerns related to Mealy machine:

The outputs may have momentary false values because of the delay encountered from the time that the inputs change and the time that the flip-flop outputs change.

In order to synchronize a Mealy-type circuit, the inputs of the sequential circuit must be synchronized with the clock and the outputs must be sampled immediately before the clock edge.

The inputs are changed at the inactive edge of the clock to ensure that the inputs to the flip-flops stabilize before the active edge of the clock occurs. Thus, the output of the Mealy machine is the value that is present immediately before the active edge of the clock.

Equivalent Sequential Circuits

Definition:

Sequential circuit N_1 is equivalent to sequential circuit N_2 if for each state ‘p’ in N_1, there is a state ‘q’ in N_2 such that p Ξ q, and conversely, for each state ‘s’ in N_2, there is a state ‘t’ in N_1 such that s Ξ t.

Simply said: two sequential circuits are equivalent if they are capable of doing the same work.

Explanation:

Thus if $N_1 \equiv N_2$, for every starting state ‘p’ in N_1, we can find a corresponding starting state ‘q’ such that $\lambda_1(p, \underline{X}) \equiv \lambda_2(q, \underline{X})$ for all input sequences $\underline{X}$ (i.e., the output sequences are the same for the same input sequence).

Then, we can replace N_1 with its equivalent circuit N_2.

If both N_1 and N_2 have a minimum number of states and N_1 Ξ N_2, then N_1 and N_2 must have the same number of states. Otherwise, one circuit would have a state left over which was not equivalent to any state in the other circuit

Equivalent Sequential Circuits

Example: (Inspection Method)

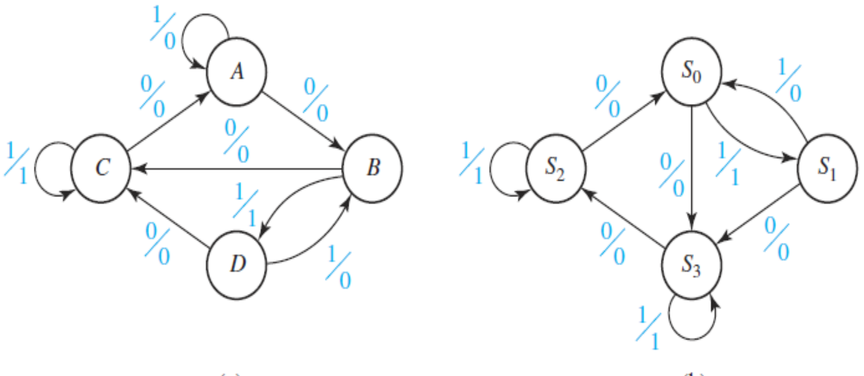
Figure shows two reduced state tables and their corresponding state graphs.

By inspecting the state graphs, it appears that if the circuits are equivalent, we must have A equivalent to either S_2 or S_3 because these are the only states in N_2 with self-loops; but the outputs of A match only with S_2 & hence A Ξ S_2.

If we assume that A Ξ S_2, this implies that we must have B Ξ S_0 which in turn implies that we must have D Ξ S1 and C Ξ S3.

	N_1			
	$X = 0$	1	$X = 0$	1
A	B	A	0	0
B	C	D	0	1
C	A	C	0	1
D	C	B	0	0

	N_2			
	$X = 0$	1	$X = 0$	1
S_0	S_3	S_1	0	1
S_1	S_3	S_0	0	0
S_2	S_0	S_2	0	0
S_3	S_2	S_3	0	1



(a) (b)

Using the state tables, we can verify that these assumptions are correct because for every pair of assumed equivalent states, the next states are equivalent and the outputs are equal when X = 0 and also when X = 1. This verifies that N_1 Ξ N_2.

Equivalent Sequential Circuits

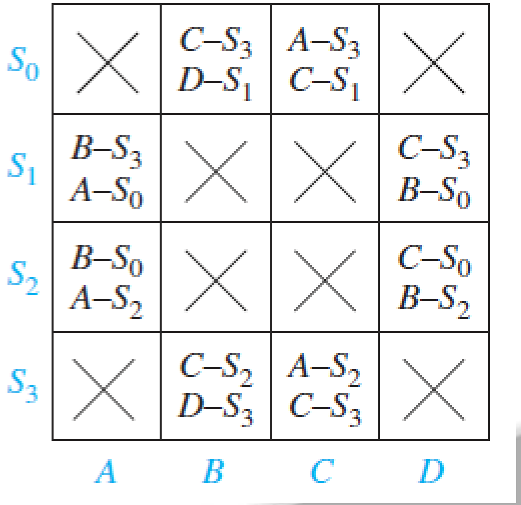
Implication table Method:

When machines have large number of states their equivalence can be determined using Implication table.

Procedure:

Because the states of one circuit must be checked for equivalence against states of the other circuit, an implication chart is constructed with rows corresponding to states of one circuit and columns corresponding to states of the other.

The implication table is shown in the figure:



S_0	X	$C–S_3$ $D–S_1$	$A–S_3$ $C–S_1$	X
S_1	$B–S_3$ $A–S_0$	X	X	$C–S_3$ $B–S_0$
S_2	$B–S_0$ $A–S_2$	X	X	$C–S_0$ $B–S_2$
S_3	X	$C–S_2$ $D–S_3$	$A–S_2$ $C–S_3$	X
	A	B	C	D

The first column of Figure is filled in by comparing row A of the state table in the Figure (a) with each of the rows in Figure (b). Because states A and S_0 have different outputs, an X is placed in the A-S_0 square. Because states A and S_1 have the same outputs, the implied next-state pairs (B-S_3 and A-S_0) are placed in the A-S_1 square, etc. Similarly we can fill up the remainder of the table.

Equivalent Sequential Circuits

The next step is:

Squares corresponding to additional non-equivalent state pairs are crossed out.

Same procedure is adopted as in the state reduction where we were interested in determining equivalent states.

This is shown in the figure (b). Fig(a) is reproduced from previous slide.

In Fig (b);

square A-S_1 is crossed because A-S_0 square in (a) has X in it; square B-S_3 is crossed becauseD-S_3 square in Fig (a) has X in it.

We continue with this procedure until no square is left to be crossed.

	A	B	C	D
S_0	×	$C–S_3$ $D–S_1$	$A–S_3$ $C–S_1$	×
S_1	$B–S_3$ $A–S_0$	×	×	$C–S_3$ $B–S_0$
S_2	$B–S_0$ $A–S_2$	×	×	$C–S_0$ $B–S_2$
S_3	×	$C–S_2$ $D–S_3$	$A–S_2$ $C–S_3$	×

(a)

	A	B	C	D
S_0	×	$C–S_3$ $D–S_1$	~~$A–S_3$~~ ~~$C–S_1$~~	×
S_1	~~$B–S_3$~~ ~~$A–S_0$~~	×	×	$C–S_3$ $B–S_0$
S_2	$B–S_0$ $A–S_2$	×	×	~~$C–S_0$~~ ~~$B–S_2$~~
S_3	×	~~$C–S_2$~~ ~~$D–S_3$~~	$A–S_2$ $C–S_3$	×

(b)

Therefore, the state equivalence is found out to be as:

C - S_3; D - S_1; A - S_2;

State Reduction

Merger Graph Method:

In this as a first step, we define the Merger Graph & explain the procedure for drawing it.

Merger graph of an n-state machine M is an undirected graph defined as follows:

1. It consists of n vertices, each of which corresponds to a state of M
2. For each pair of states (Si,Sj) in M, whose next-state and output entries are not conflicting , an undirected arc is drawn between vertices Si and Sj

   ‘not conflicting’: it means that both the states make transition to a same state & the outputs too are same. If ‘p’ & ‘q’ are 2 states then both make transition to a state ‘r’ & their outputs also same , i.e. either ‘0’ or ‘1’. We have compatible pair
3. If, for a pair of states (Si,Sj), the corresponding output symbols under all input symbols are not conflicting, but the successors (respective states they make transition to) are not the same (conflicting), an interrupted arc is drawn between Si and Sj, and the implied pairs are entered in the space.

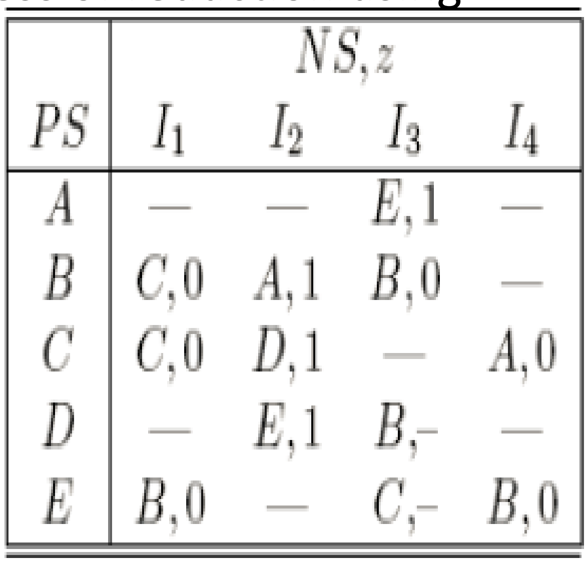
   Two states are ‘implied pairs’ if their outputs are same but they make transition to different states.

State Reduction

Example:

With the help of an example we will explain the process of reduction using Merger Graph.

The state table for a machine is shown:



	NS, z			
PS	I_1	I_2	I_3	I_4
A	—	—	E,1	—
B	C,0	A,1	B,0	—
C	C,0	D,1	—	A,0
D	—	E,1	B,–	—
E	B,0	—	C,–	B,0

PS: Present State ; NS: Next State

I_1 I_2 I_3 I_4 : Inputs ; z : output

In the table,

the entry ‘ –’ denote unspecified state & output.

C, - : indicate unspecified output; state is specified as C

Compatible pair:

When for an input, the state and/or output are not specified then both the state & output can assume any value. In view of this, A & C is defined as compatible pair. In the graph we draw a line connecting node A with node C. No other pair is compatible.

Implied Compatible pair:

For an input the outputs are same, but successor states are different. BC, BD, BE & so on. We draw a broken line between two state nodes & marked

State Reduction

Conflicting pair:

For an input, the outputs are different. Pair: AB.

Conflicting states are not connected through a link.

The Merger Graph:

The graph along with the state table is shown.

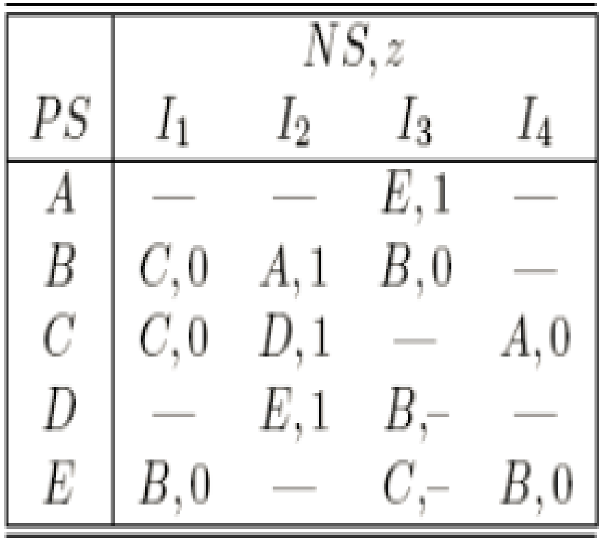
1. Nodes represent states of a machine, as shown in the graph.

2. Starting with state A, its outputs, for different inputs, are compared with outputs of states from B to E to determine 1) compatible 2) implied compatible & 3) conflicting, pairs.

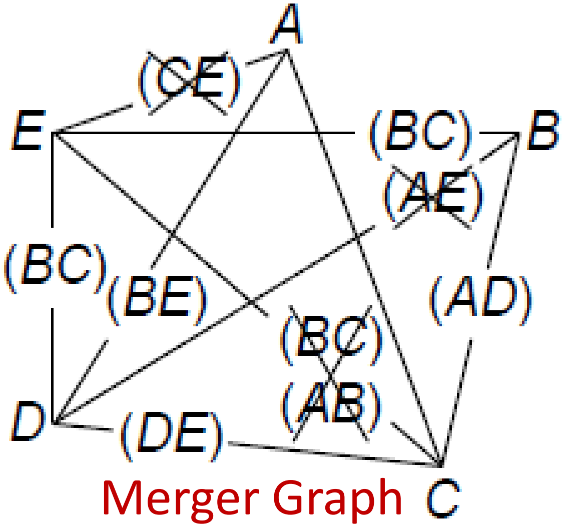
We find that:

A & C are compatible because undefined state & output can assume any value. A direct link between A & C.

A & B are conflicting because outputs do not match. Hence no connectivity between A & B



	NS, z			
PS	I_1	I_2	I_3	I_4
A	—	—	$E, 1$	—
B	$C, 0$	$A, 1$	$B, 0$	—
C	$C, 0$	$D, 1$	—	$A, 0$
D	—	$E, 1$	$B, -$	—
E	$B, 0$	—	$C, -$	$B, 0$

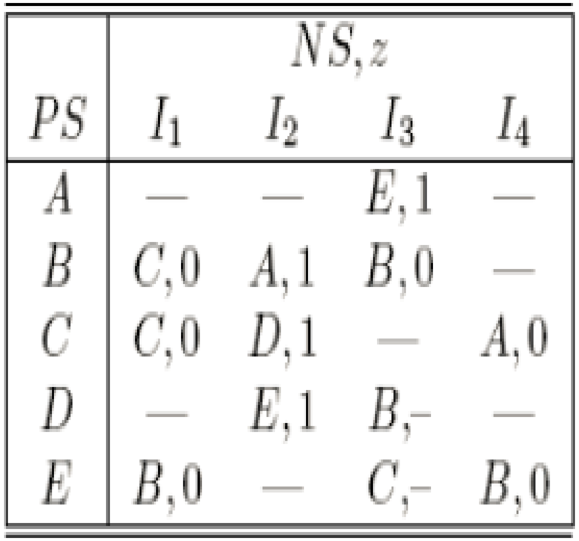


Merger Graph

State Reduction

Between A & D:

For input I_3, the states do not match. A makes transition to E & D makes transition to B; (BE) is implied compatible pair. Hence an interrupted link is drawn between A & D is drawn with (BE) indicated.



PS	NS, z			
	I_1	I_2	I_3	I_4
A	—	—	E, 1	—
B	C, 0	A, 1	B, 0	—
C	C, 0	D, 1	—	A, 0
D	—	E, 1	B, –	—
E	B, 0	—	C, –	B, 0

Between A & E:

Same as between A & D. The pair is (EC). So an interrupted Link between A & E with (CE) shown.

Similarly we draw for B & C; B& D; B&E:

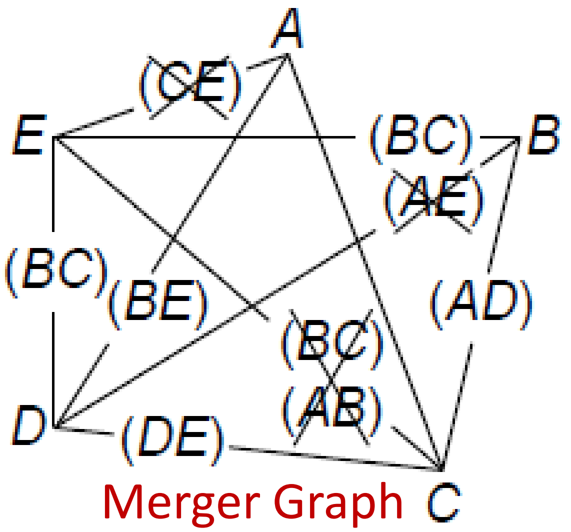
All are interrupted links with implied compatible pairs shown as (AD) (AE) & (BC).

For C&D ; C&E:

Interrupted links with implied compatible pairs shown as (DE) for C&D; { (BC) (AB)} for C&E.

For D&E:

Interrupted link with implied compatible pair (BC)



Merger Graph

State Reduction

Now we examine the graph to find out which nodes are not connected (conflicting states).

A & B: not connected : conflicting states.

Find out in which interrupted link (AB) appears, strike out that link. So, we see ‘X’ on E-C link.

Thus E-C no longer exits.

Therefore A-E ceases to exist (because of EC).

So cross out A-E link.

Thus A-E no longer exits. Hence (DB) ceases to exit.

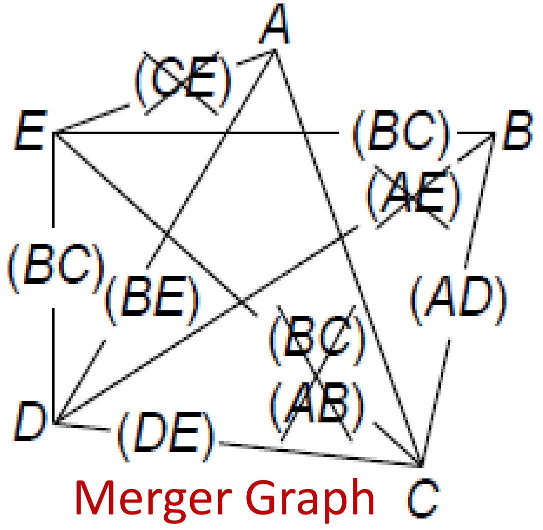
(DB) is not appearing anywhere as implied compatible Pair. So crossing out links stops at this point.

Therefore , Compatible pairs are:

(AC) (CD) (DE) (BC) (AD) (BE)

Having found compatible pairs we develop ‘Compatibility Graph’ to achieve reduction in states.

	NS,z			
PS	I_1	I_2	I_3	I_4
A	—	—	E,1	—
B	C,0	A,1	B,0	—
C	C,0	D,1	—	A,0
D	—	E,1	B,-	—
E	B,0	—	C,-	B,0



Merger Graph

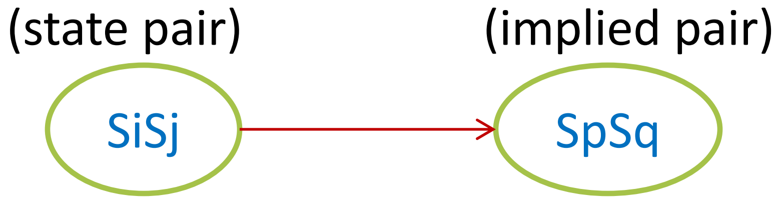
State Reduction

Compatibility graph:

A directed graph whose vertices correspond to all compatible pairs, and an arc leads from (Si,Sj) to (Sp,Sq) if and only if (Si,Sj) implies (Sp,Sq).

That is, if we have two compatible states Si &Sj and correspondingly we have Sp & Sq as implied compatibles, then in compatible graph:

(Si,Sj) becomes a node & (Sp,Sq) becomes another node which are connected through a directed link.



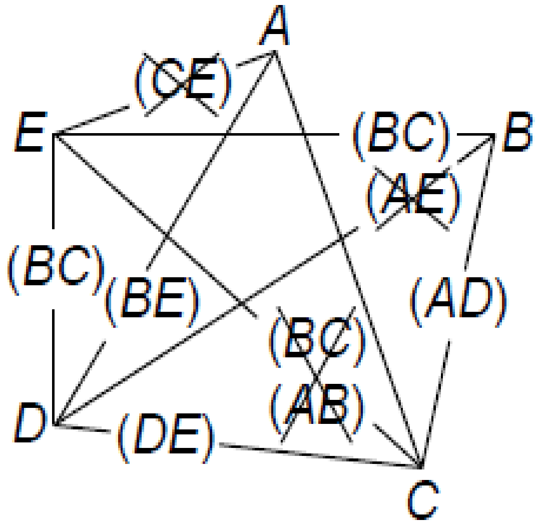
(Si,Sj) are compatible if (Sp,Sq) are compatible.

State Reduction

Procedure to develop Compatibility Graph from Merger Graph:

The compatible pairs are:

(AC) (CD) (DE) (BC) (AD) (BE)



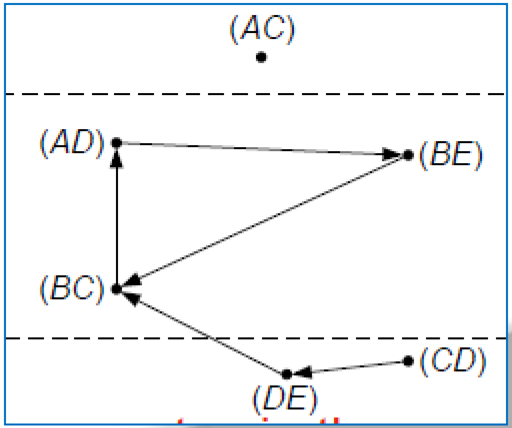
We have the Merger Graph:

Each compatible pair is a node. Hence we have 6 nodes.

Node A:

pair (AC):

No outgoing arm from (AC) because between A & C we have uninterrupted link. No dependence on other pair for compatibility.



pair (AD):

There is an interrupting pair (BE); hence a branch will be shown from AD to BE in the compatibility graph.

Node B:

We have 2 interrupted links, BE (BC) & BC (AD), from node B. Hence a branch is shown from BE to BC & from BC to AD in the compatibility graph.

State Reduction

Procedure to develop Compatibility Graph from Merger Graph:

Node C:

CA & CB are already covered. CE is crossed. So, the remaining interrupted link is CD (DE).

We draw a directed link from CD to DE in the compatibility graph.

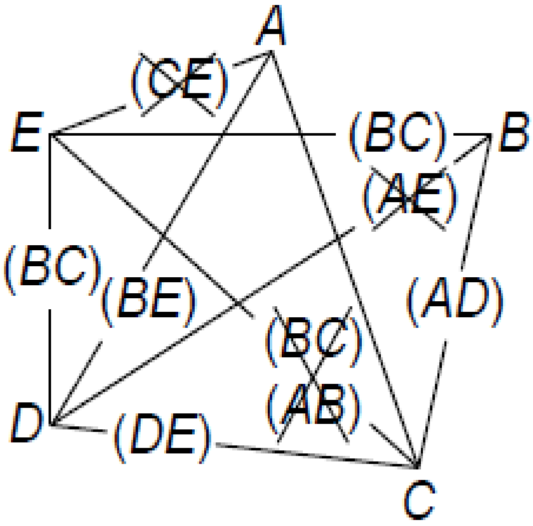
Node D:

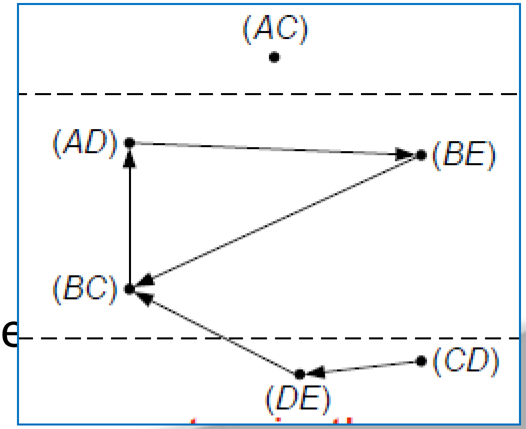
All the interrupted links from this node have been considered except CD(DE).

A directed link from CD to DE is drawn in the compatibility graph.

Node E:

All interrupted links have been considered & hence no more additions to the compatibility graph.





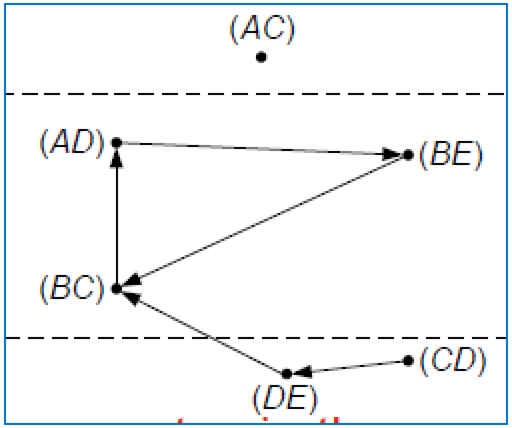
State Reduction

We need to understand what is Closed Sub Graph of a compatibility graph.

Closed Sub Graph:

A set of compatibles for machine M is said to be closed if: for every compatible contained in the set, all its implied compatibles are also contained in the set

A closed set of compatibles, which contains all states of M, is called a closed covering.



For example, if we choose (BE) then (BC) too should be chosen because (BC) is implied compatible of (BE). So, having chosen (BC) we need to choose (AD) because (AD) is implied compatible of (BC). Since we started with (BE) so implied compatible of (AD) is already chosen.

Therefore, closed set of compatibles is given as:

{ BE AD BC}

Second choice:

If we take (AC) it has no implied compatible. Then we may choose (AD) (BE) & (BC)

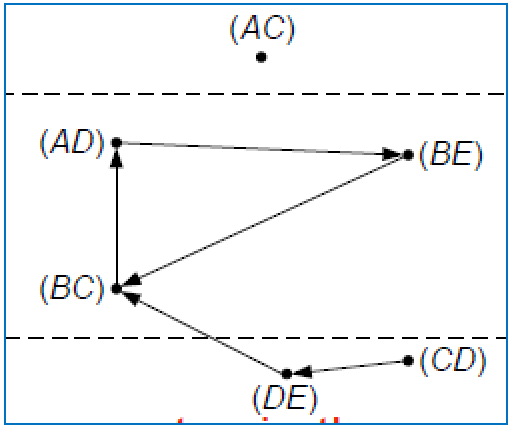
So , closed set of compatibles is: {AC AD BE BC}

State Reduction

Third Choice:

We can start with (CD) & go on to form a closed set of compatibles, as:

{CD DE BC AD BE}



There can be many closed set of compatibles.

Now, we define closed covering.

Closed Covering:

A closed covering set is one amongst the closed set of compatibles. If a given closed set of compatibles contains all the vertices (nodes) of a Merger Graph then it constitutes a closed covering.

All the closed set of compatibles as obtained for this machine contain all the nodes of the Merger Graph & hence all are closed covering.

From the available ones we choose a set with minimum number of compatible states. Closed covering with minimum number of compatibles is called Minimum Closed Covering . So we get a closed covering with minimum number of compatibles as : { BE AD BC}

Thus, a closed covering with minimum compatibles defines a minimal machine

State Reduction

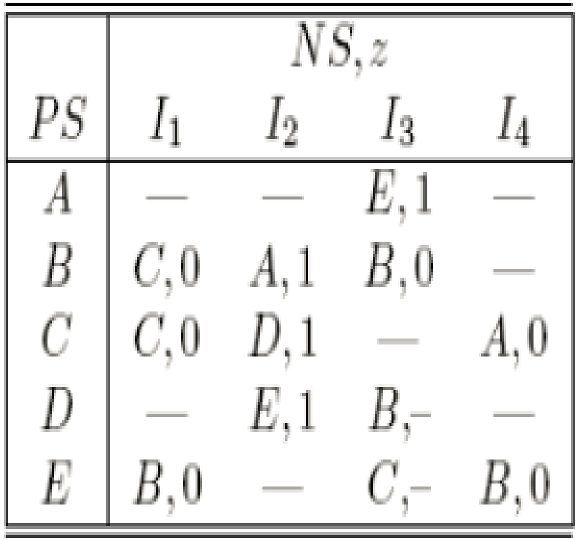
Minimal State Machine:

Each compatible in minimal closed covering is defined as a state of the minimal machine. We have: { BE AD BC}

We define AD as α ; BC as β ; & BE as ϒ

A state table is drawn with newly defined states as shown:

The original state table is reproduced.



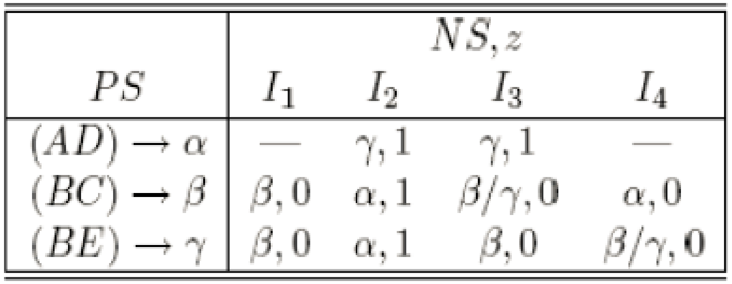
PS	NS, z			
	I_1	I_2	I_3	I_4
A	—	—	$E, 1$	—
B	$C, 0$	$A, 1$	$B, 0$	—
C	$C, 0$	$D, 1$	—	$A, 0$
D	—	$E, 1$	$B, -$	—
E	$B, 0$	—	$C, -$	$B, 0$

From the original state table we see that under the column PS: transition from A to D implies don't care for I_1; D to E for I_2; A to E & D to B for I_3; A & D to don't care for I_4

(BE) forms a compatible which Is defined as state ϒ for minimal machine.

So, in the state table for minimal machine

we replace E & B by ϒ.



PS	NS, z			
	I_1	I_2	I_3	I_4
$(AD) \rightarrow \alpha$	—	$\gamma, 1$	$\gamma, 1$	—
$(BC) \rightarrow \beta$	$\beta, 0$	$\alpha, 1$	$\beta/\gamma, 0$	$\alpha, 0$
$(BE) \rightarrow \gamma$	$\beta, 0$	$\alpha, 1$	$\beta, 0$	$\beta/\gamma, 0$

(MINIMAL MACHINE)

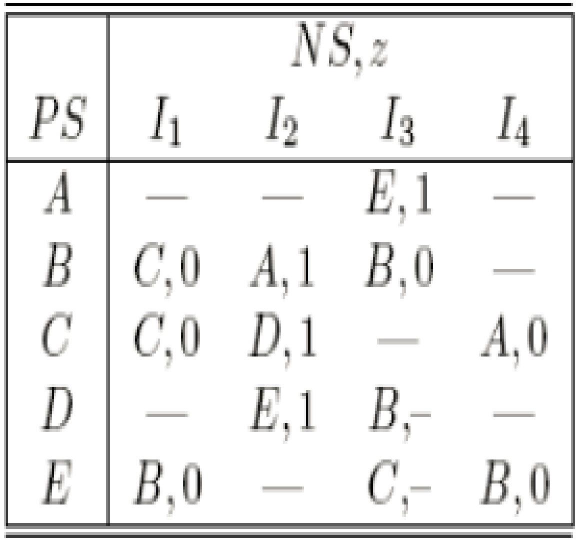
State Reduction

Minimal State Machine:

(BC) : β

From the old table we see that in transition from B to C: For I_1, B to C & C to C (β); For I_2, B to A & C to D (α); For I_3, B to B & C to don’t care (β/ϒ), because B is in both β & ϒ; Any one from β & ϒ can be chosen as the state for minimal machine.

For I_4, B to don’t care & C to A (α).

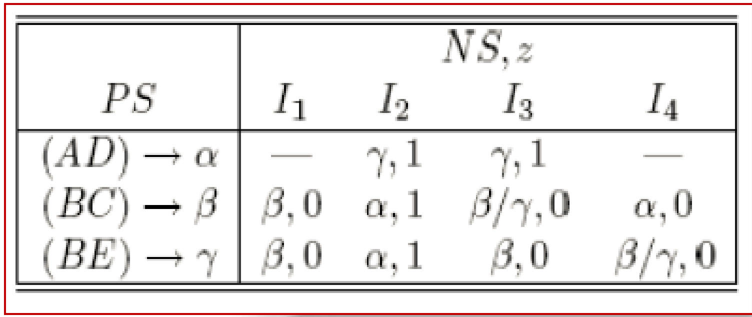


PS	NS, z			
	I_1	I_2	I_3	I_4
A	—	—	$E,1$	—
B	$C,0$	$A,1$	$B,0$	—
C	$C,0$	$D,1$	—	$A,0$
D	—	$E,1$	$B,-$	—
E	$B,0$	—	$C,-$	$B,0$

(BE) : ϒ

From the old table we see that in transition from B to E: For I_1, B to C & E to B (β); For I_2, B to A & E to don’t care (α); For I_3, B to B (β/ϒ), because B is in both β & ϒ; For I4, B to don’t care & E to B: β/ϒ

Any one from β & ϒ can be chosen as the state for minimal machine.



PS	NS, z			
	I_1	I_2	I_3	I_4
$(AD) \rightarrow \alpha$	—	$\gamma, 1$	$\gamma, 1$	—
$(BC) \rightarrow \beta$	$\beta, 0$	$\alpha, 1$	$\beta/\gamma, 0$	$\alpha, 0$
$(BE) \rightarrow \gamma$	$\beta, 0$	$\alpha, 1$	$\beta, 0$	$\beta/\gamma, 0$

(MINIMAL MACHINE)

Design of Clocked Sequential Circuits

Design Procedure:

From a given set of specifications a logic diagram is generated; or, a list of Boolean functions are generated from which the logic diagram can be obtained.

Whereas a combinational circuit is completely defined by its Truth Table, the sequential circuit requires a state table for specifications.

The procedure is listed:

1. From the word description and specifications of the desired operation, derive a state diagram/ state table for the circuit.
2. Reduce the number of states if necessary.
3. Assign binary values to the states.
4. Obtain the binary-coded state table.
5. Choose the type of flip-flops to be used.
6. Derive the simplified flip-flop input equations and output equations.
7. Draw the logic diagram.

All the above steps have been studied in somewhat detail.

Design of Clocked Sequential Circuits

For designing a sequential circuit, we need to derive a state diagram/ state table from the word description and specifications of the desired operation. This is the most critical step in the design process because if a state diagram/table is wrongly drawn, the designed sequential circuit will serve no purpose.

We will explain design procedure with the help of an example.

Example:

Let us design a circuit that detects a sequence of three or more consecutive 1’s in a string of bits coming through an input line (i.e., the input is a serial bit stream). The output = 1: iff 3 or more number of consecutive 1s are detected; else it is = 0.

Solution:

We will design a Moore model sequential circuit. In Moore sequential circuit, the output is a function of ‘states only’ & not a function of ‘states and input’

1st step is to obtain state diagram or state table.

Each time ‘1’ appears in the input sequence, state transition takes place during clock time. Whenever ‘0’ appears in the input stream, the system goes to reset (initial) state.

Design of Clocked Sequential Circuits

State Diagram:

The state diagram for this type of circuit is shown in the Fig.

S_0 (reset / initial state) ;S_1 S_2 S_3 are states of the system.

It is derived by starting with state S_0, the reset state.

If the input is ‘0’, the circuit stays in S_0, but

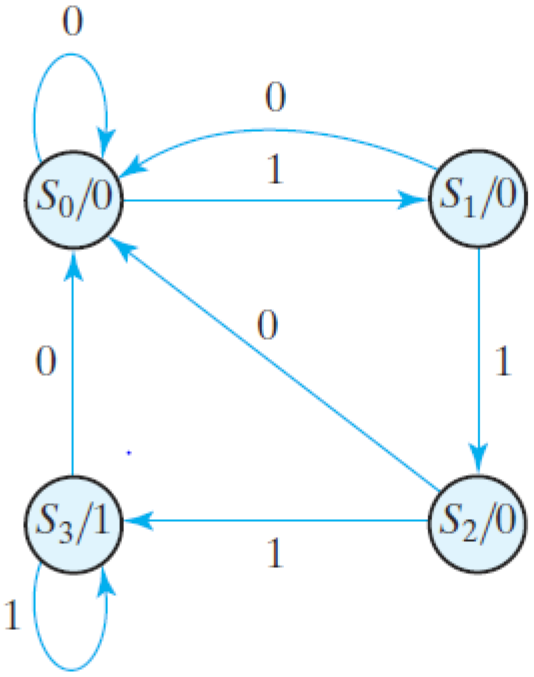
if the input is ‘1’, it goes to state S_1 to indicate that a ‘1’ was detected.

If the next input is ‘1’, the change is to state S_2 to indicate the arrival of two consecutive 1’s, but

if the input is ‘0’, the state goes back to S_0.

The third consecutive 1 sends the circuit to state S_3. If more 1’s are detected, the circuit stays in S_3.

Any ‘0’ input sends the circuit back to S_0.



In this way, the circuit stays in S_3 as long as there are three or more consecutive 1’s received.

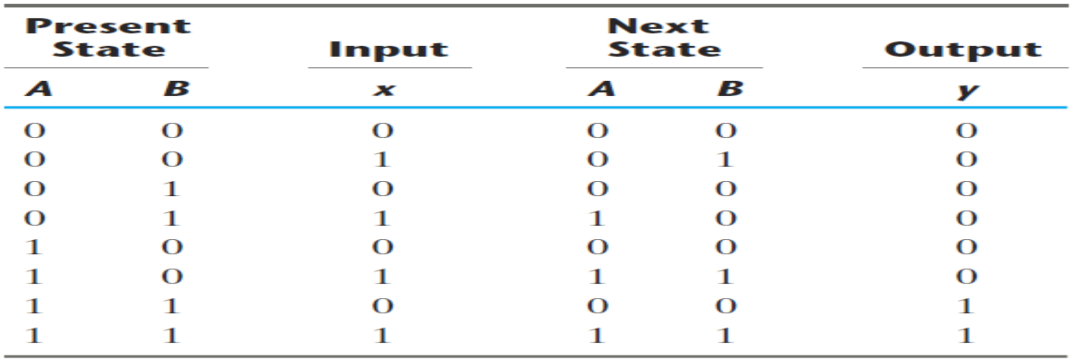
This is a Moore model sequential circuit, since the output is ‘1’ when the circuit is in state S_3 and is ‘0’ otherwise.

Design of Clocked Sequential Circuits

Next we need to assign binary codes to the states and list the state table. This is shown in the Table.

The table is derived from the state diagram (drawn earlier) with a sequential binary assignment.

S_0: (00); S_1: (01); S_2: (10): S_3: (11)



Present State		Input	Next State		Output
A	*B*	*x*	*A*	*B*	*y*
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

We can now synthesize a circuit using any flip-flop.

Design using D flip-flop:

Required number of flip-flops:

We will need two D flip-flops to represent the four states (0 to 3). Label their outputs as 'A' and 'B'.

Number of inputs = 1 ; Number of outputs = 1

Characteristic equation:

$$Q(t + 1) = D_Q$$

It means that the next-state values in the state table specify the D input condition for the flip-flop.

Design of Clocked Sequential Circuits

The flip-flop input equations are obtained directly from the next-state columns of A and B and expressed in sum-of-minterms form.

Present State		Input	Next State		Output
A	*B*	*x*	*A*	*B*	*y*
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

We have two flip-flops.

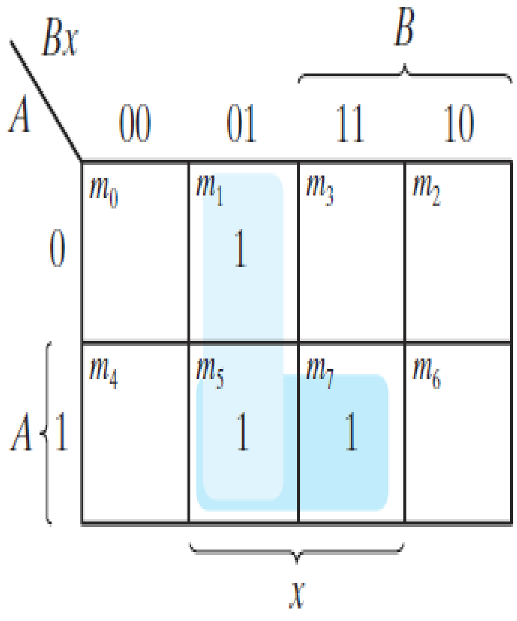
To write an equation for each flip-flop, we look for Next State (NS)= 1 in the table.

A flip-flop:

NS = 1 for minterms: m_3, m_5 & m_7.

Therefore, $D_A = m_3 + m_5 + m_7$.

We simplify this input using K-map, as shown : $D_A = A\,x + B\,x$

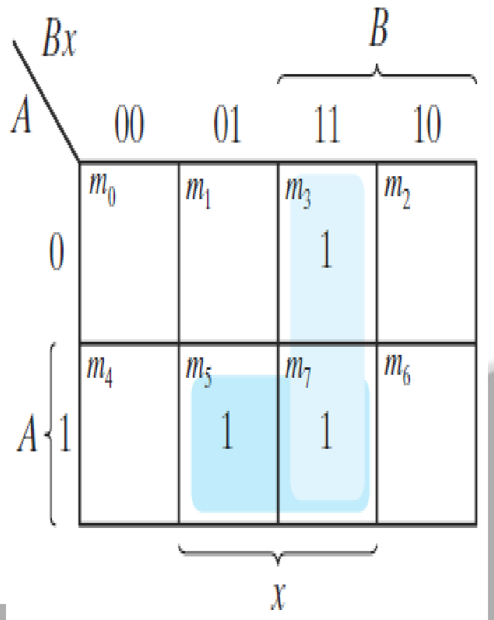


B flip-flop:

NS = 1 for minterms: m_1, m_5 & m_7.

Therefore, $D_B = m_1 + m_5 + m_7$.

We simplify this input using K-map diagram. $D_B = A\,x + B'x$

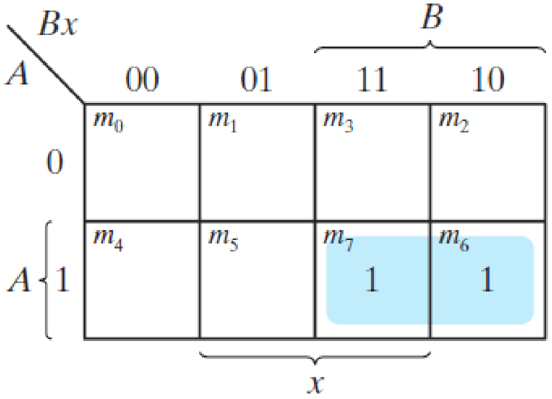


Design of Clocked Sequential Circuits

The output, y:

From the state table we see that it is governed by minterms m6 & m7. Simplified expression for y is obtained using K-map, as shown.

The simplified form for output , y is : y = AB



A \ Bx	00	01	11	10
0	m_0	m_1	m_3	m_2
1	m_4	m_5	m_7 1	m_6 1

Mathematically in closed form, we write the flip-flop input Equations & the output equation as:

$$A(t+1) = D_A(A, B, x) = \Sigma(3, 5, 7)$$

$$B(t+1) = D_B(A, B, x) = \Sigma(1, 5, 7)$$

$$y(A, B, x) = \Sigma(6, 7)$$

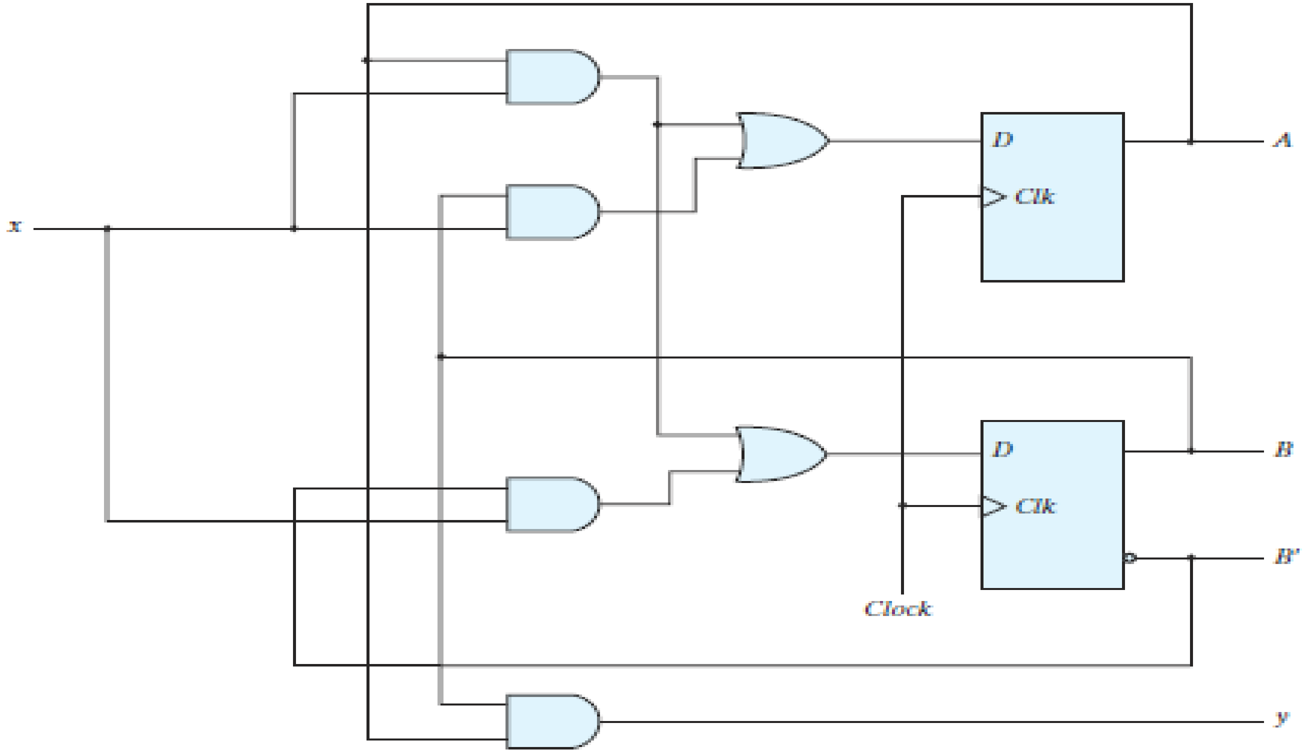
Where A and B are the present-state values of flip-flops A and B, x is the input, and DA and DB are the input equations.

The minterms for output y are obtained from the output column in the state table.

The advantage of designing with D flip-flops is that the Boolean equations describing the inputs to the flip-flops can be obtained directly from the state table.

Design of Clocked Sequential Circuits

The schematic for the design is given as:



Design of Clocked Sequential Circuits

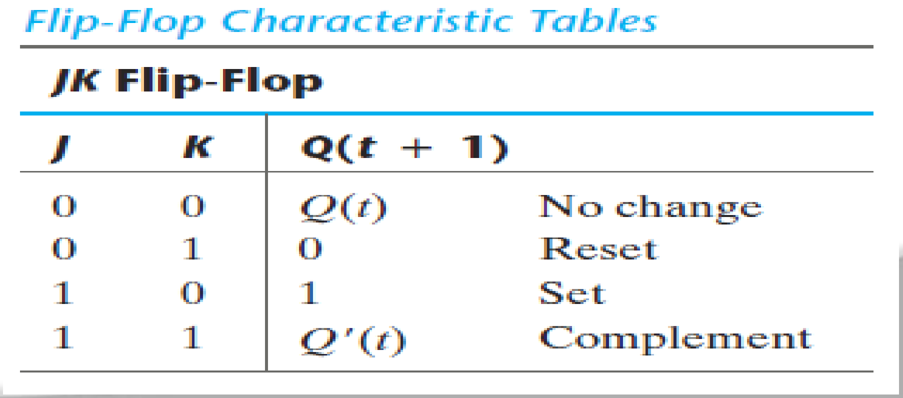
Design using other flip-flops:

When D -type flip-flops are used, the input equations are obtained directly from the next state.

This is not the case for the JK and T types of flip-flops. In order to determine the input equations for these flip-flops, it is necessary to derive a functional relationship between the state table and the input equations; thereby making the design process complicated.

Excitation Table:

The flip-flop characteristic tables presented in the Tables below provide the value of the next state when the inputs and the present state are known. These tables are useful for analyzing sequential circuits and for defining the operation of the flip-flops.



Flip-Flop Characteristic Tables

JK Flip-Flop

J	*K*	$Q(t + 1)$	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

Design of Clocked Sequential Circuits

Excitation Table:

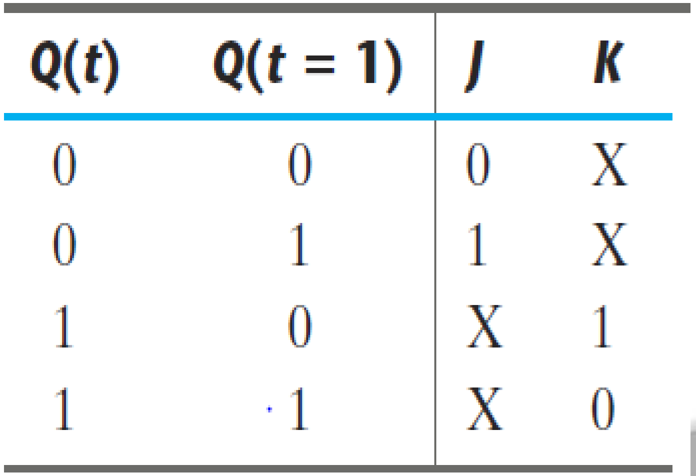
During the design process, we usually know the transition from the present state to the next state and wish to find the flip-flop input conditions that will cause the required transition.

For this reason, we need a table that lists the required inputs for a given change of state. Such a table is called an excitation table.

The excitation tables for JK flip-flop is shown below:

When both present state and next state are ‘0’, the ‘J’ input must remain at ‘0’ and the ‘K’ input can be either ‘0’ or ‘1’.

Similarly, when both present state and next state are ‘1’, the ‘K’ input must remain at ‘0’, while the ‘J’ input can be ‘0’ or ‘1’.



$Q(t)$	$Q(t = 1)$	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(a) *JK* Flip-Flop

Design of Clocked Sequential Circuits

If the flip-flop is to have a transition from the 0-state to the 1-state,

J must be =1, since the J input sets the flip-flop.

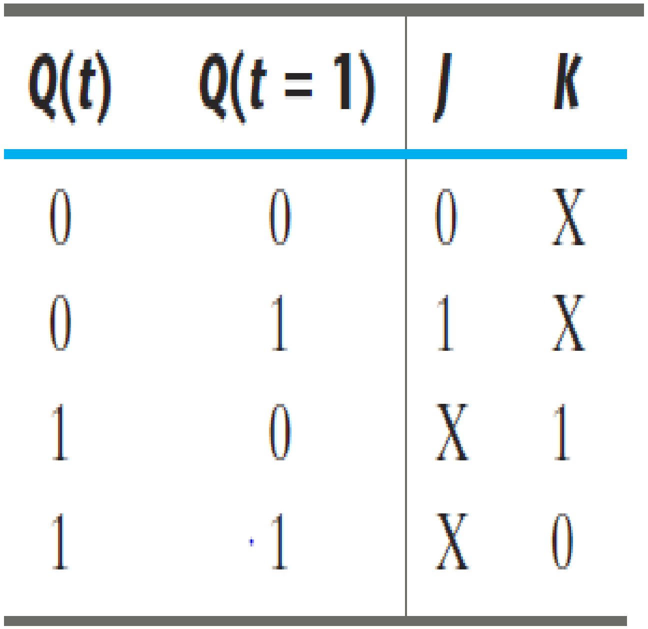
However, input K may be either 0 or 1. If K = 0, the J = 1 condition sets the flip-flop as required; if K = 1 and J = 1, the flip-flop is complemented and goes from the 0-state to the 1-state as required.

Therefore, the K input is marked with a don't-care condition for the 0-to-1 transition.

For a transition from the 1-state to the 0-state, we must have K = 1, since the K input clears the flip-flop.

However, the J input may be either 0 or 1, since J = 0 has no effect and J = 1 together with K = 1 complements the flip-flop with a resultant transition from the 1-state to the 0-state.

Therefore, the J input is marked with a don't-care condition for the 1-to-0 transition.



$Q(t)$	$Q(t = 1)$	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

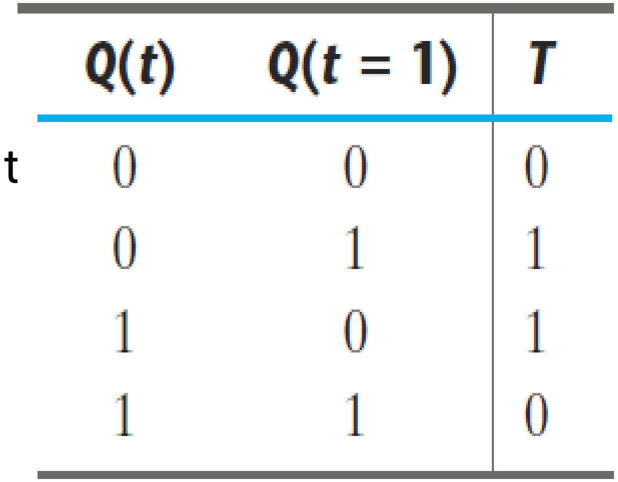
(a) *JK* Flip-Flop

Design of Clocked Sequential Circuits

Excitation Table for T Flip-Flop:

The excitation table for the T flip-flop is shown in (b). From the characteristic table, we find that when input T = 1, the state of the flip-flop is complemented, and when T = 0, the state of the flip-flop remains unchanged.

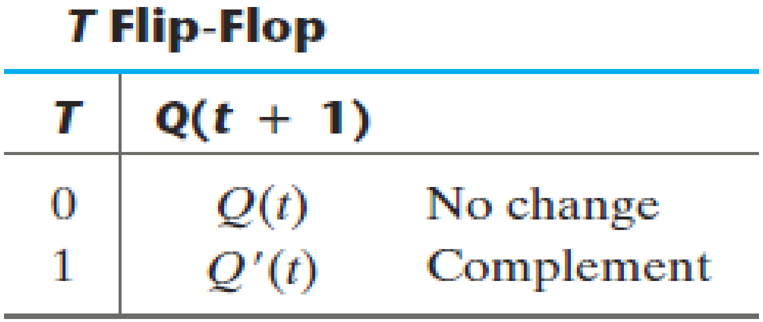
Therefore, when the state of the flip-flop must remain the same, the requirement is that T = 0. When the state of the flip-flop has to be complemented, T must equal 1.



$Q(t)$	$Q(t = 1)$	T
0	0	0
0	1	1
1	0	1
1	1	0

(b) *T* Flip-Flop

Excitation Table



***T* Flip-Flop**

T	$Q(t + 1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

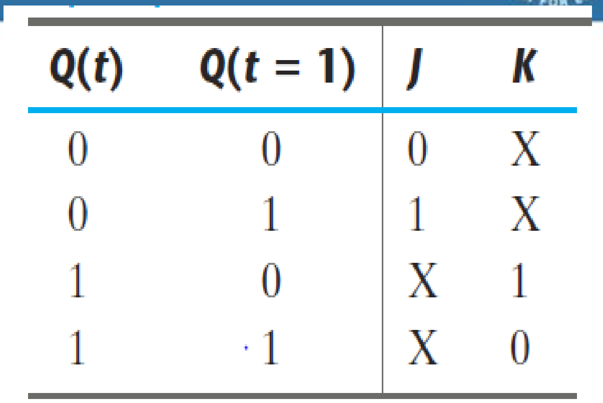
Characteristic Table

Design of Clocked Sequential Circuits

Synthesis using JK Flip Flops:

The synthesis procedure for sequential circuits with JK flip-flops is the same as with D flip-flops.

The only difference being that the input equations for the flip-flop are evaluated from the present state to the next-state transition. Input equations are derived from the excitation table of the flip-flop.



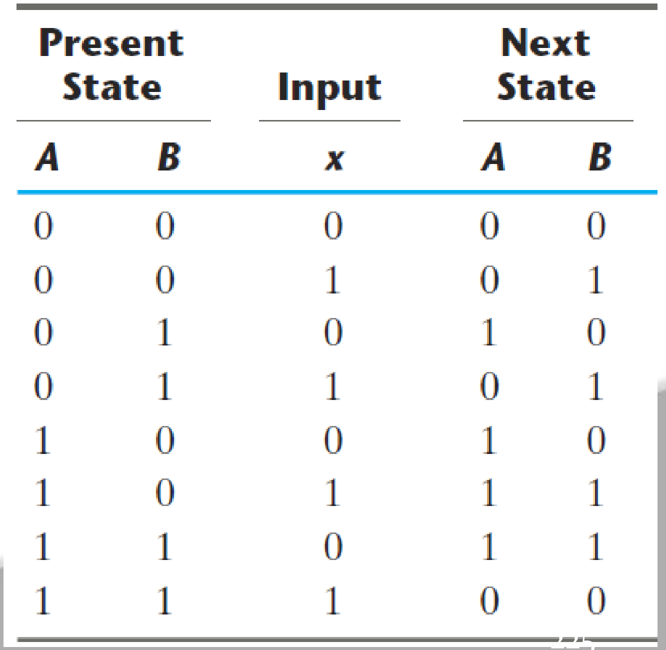
$Q(t)$	$Q(t = 1)$	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(a) *JK* Flip-Flop

Excitation Table

Example:

We will synthesize a sequential circuit for which the state table is as shown:



Present State		Input	Next State	
A	B	x	A	B
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

We have 2 flip flops A & B. The inputs are designated as : J_A & K_A ; J_B & K_B.

1st row:

Transition for both the flip flops is from ‘0’ to ‘0’. From The excitation table we see that

$J_A = 0$ & $K_A = X$; $J_B = 0$ & $K_B = X$

Design of Clocked Sequential Circuits

Synthesis using JK Flip Flops:

2nd row:

‘A’ makes a transition from ‘0’ to ‘0’: J_A = 0 & K_A = X

‘B’ makes a transition from ‘0’ to ‘1’: J_B = 1 & K_B = X

3rd row:

‘A’ makes a transition from ‘0’ to ‘1’: J_A = 1 & K_A = X

‘B’ makes a transition from ‘1’ to ‘0’: J_B = X & K_B = 1

4th row:

‘A’ makes a transition from ‘0’ to ‘0’: J_A = 0 & K_A = X

‘B’ makes a transition from ‘1’ to ‘1’: J_B = X & K_B = 0

5th row:

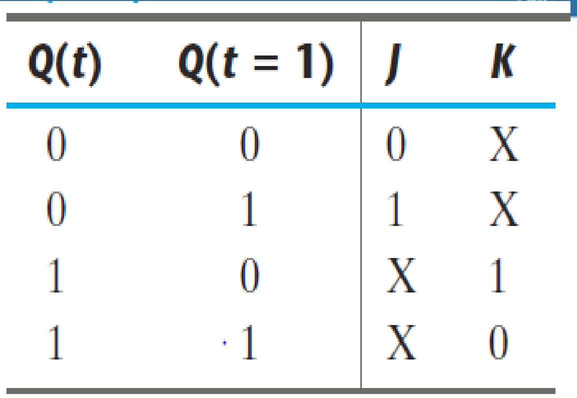
‘A’ makes a transition from ‘1’ to ‘1’: J_A = X & K_A = 0

‘B’ makes a transition from ‘0’ to ‘0’: J_B = 0 & K_B = X

6th row:

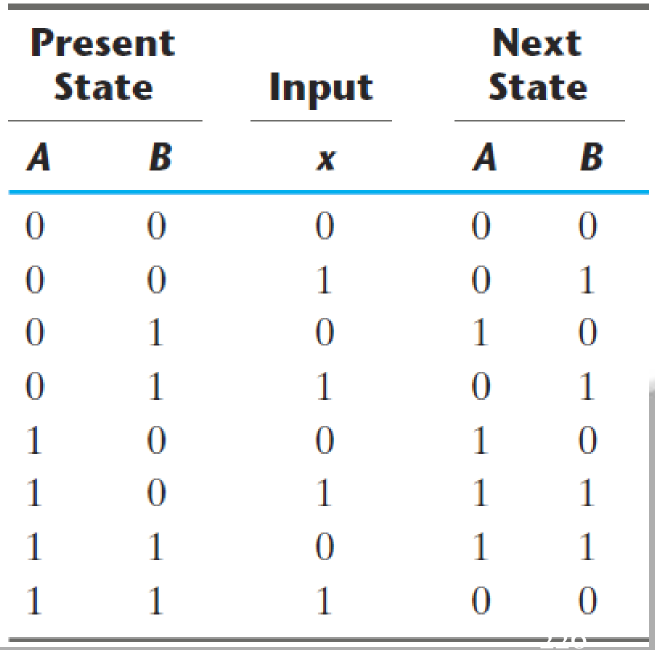
‘A’ makes a transition from ‘1’ to ‘1’: J_A = X & K_A = 0

‘B’ makes a transition from ‘0’ to ‘1’: J_B = 1 & K_B = X



$Q(t)$	$Q(t = 1)$	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(a) *JK* Flip-Flop



Present State		Input	Next State	
A	*B*	*x*	*A*	*B*
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

Design of Clocked Sequential Circuits

Synthesis using JK Flip Flops:

7th row:

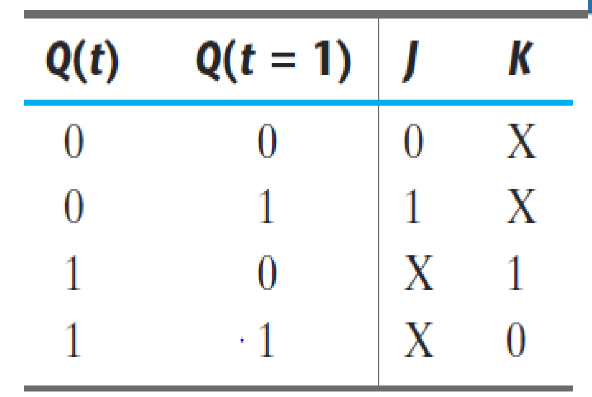
‘A’ makes a transition from ‘1’ to ‘1’: $J_A = X$ & $K_A = 0$

‘B’ makes a transition from ‘1’ to ‘1’: $J_B = X$ & $K_B = 0$

8th row:

‘A’ makes a transition from ‘1’ to ‘0’: $J_A = X$ & $K_A =1$

‘B’ makes a transition from ‘1’ to ‘0’: $J_B = X$ & $K_B = 1$

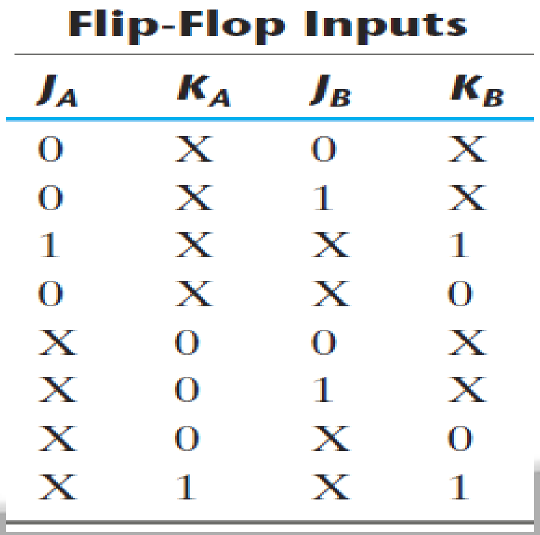


$Q(t)$	$Q(t = 1)$	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

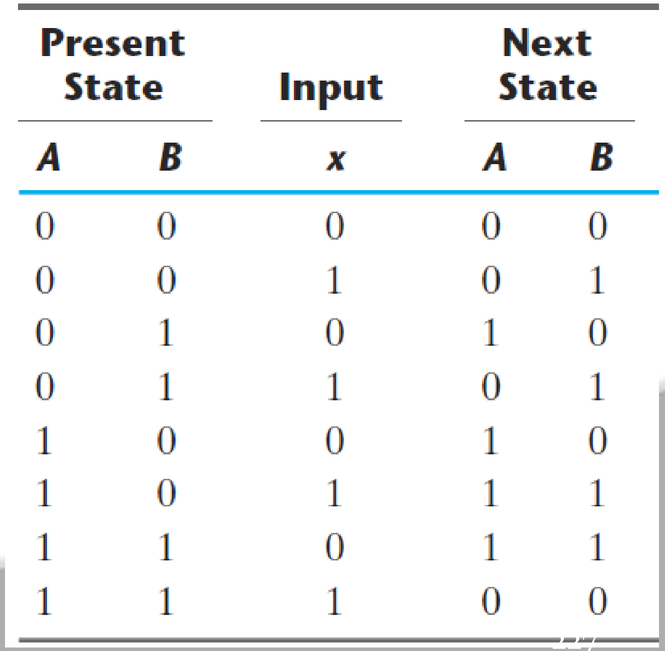
(a) *JK* Flip-Flop

Flip-Flop Input Table:

Having determined the flip-flop inputs we make a table.



Flip-Flop Inputs			
J_A	K_A	J_B	K_B
0	X	0	X
0	X	1	X
1	X	X	1
0	X	X	0
X	0	0	X
X	0	1	X
X	0	X	0
X	1	X	1



Present State		Input	Next State	
A	B	x	A	B
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

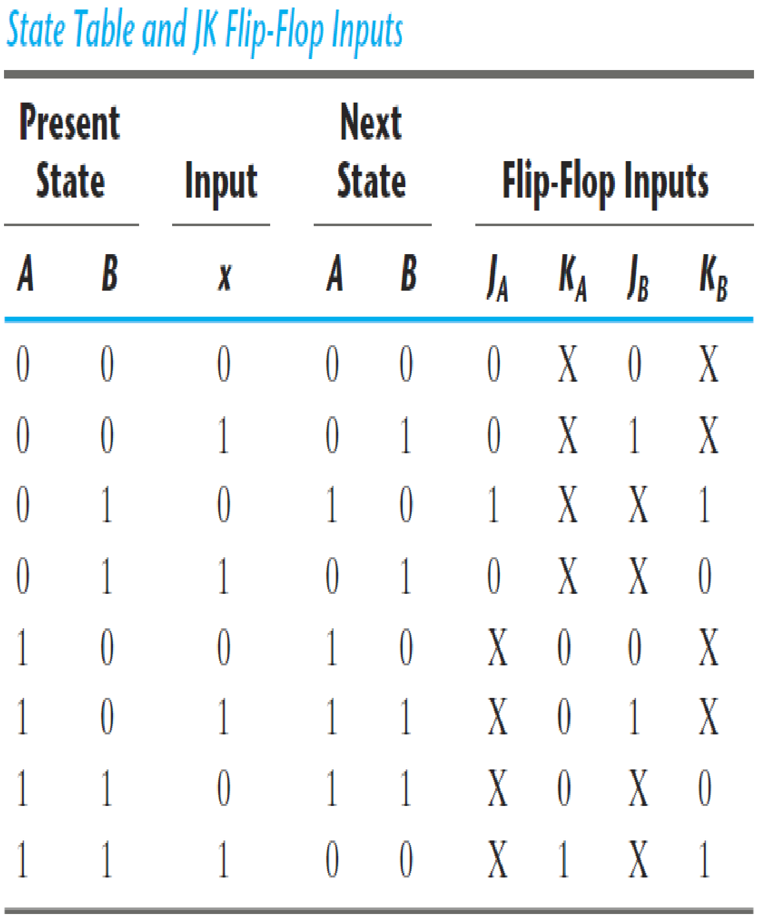
Design of Clocked Sequential Circuits

Now we make a complete table that includes state table & flip-flop input table.

The flip-flop inputs in the Table specify the truth table for the input equations as a function of present state A, present state B, and input x . The next-state values are not used during the simplification.

Representation of Flip-Flop Inputs in Sum of product form:

In each column in the table corresponding to inputs J_A & K_A ; J_B & K_B; respectively; we look for the presence of '1' & their corresponding minterms. We make use of don't care condition 'X' to simplify the input equation.



State Table and JK Flip-Flop Inputs

Present State		Input	Next State		Flip-Flop Inputs			
A	B	x	A	B	J_A	K_A	J_B	K_B
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1

(contd. ...)

Design of Clocked Sequential Circuits

J_A = A′B x′ = m_2;

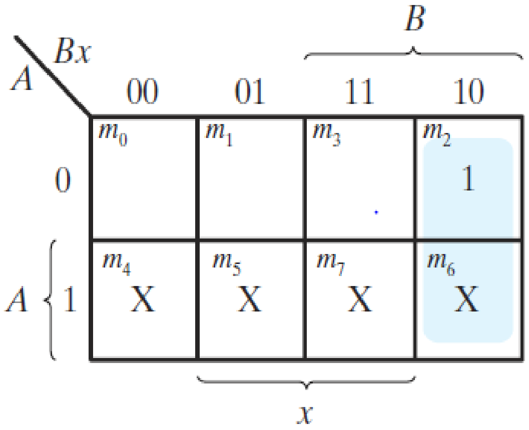
Simplify using K-map & ‘X’ conditions: J_A = BX′

K_A = A B x = m_7;

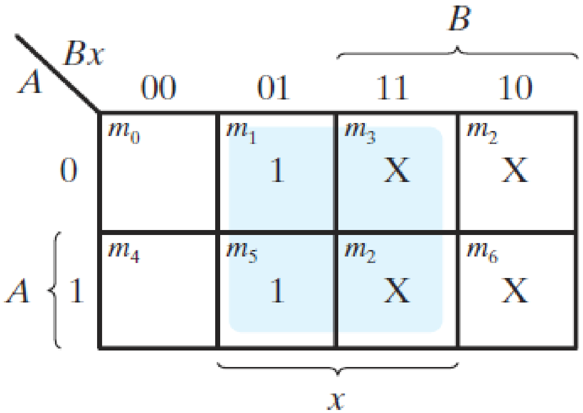
Simplify using K-map & ‘X’ conditions: K_A = B x

J_B = A′B′ x + A B′ x = m_1 + m_5;

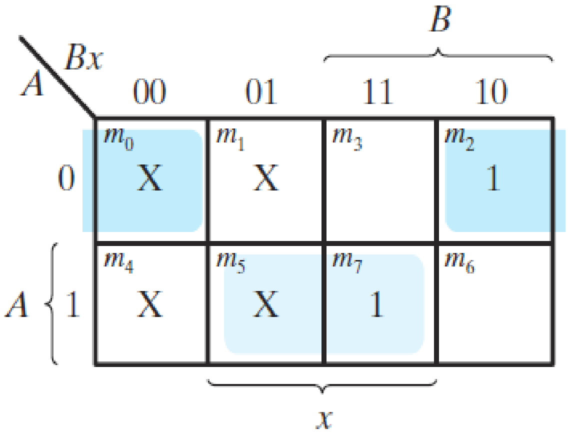
Simplify using K-map & ‘X’ conditions: J_B = x



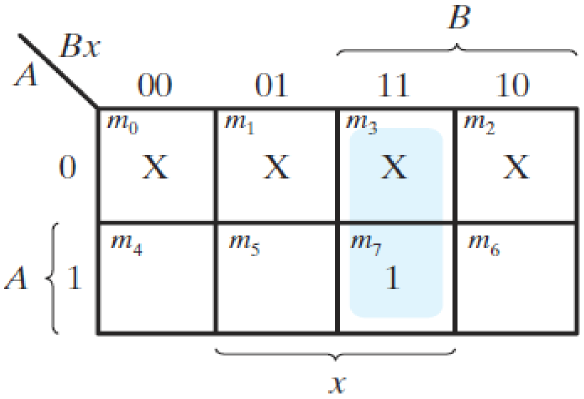
$J_A = Bx'$



$J_B = x$



$K_B = (A \oplus x)'$



$K_A = Bx$

K_B = A′B x′ + A B x = m_2 + m_7;

Simplify using K-map & ‘X’ conditions: $K_B = (A \oplus x)'$ = A x + A′ x′

Design of Clocked Sequential Circuits

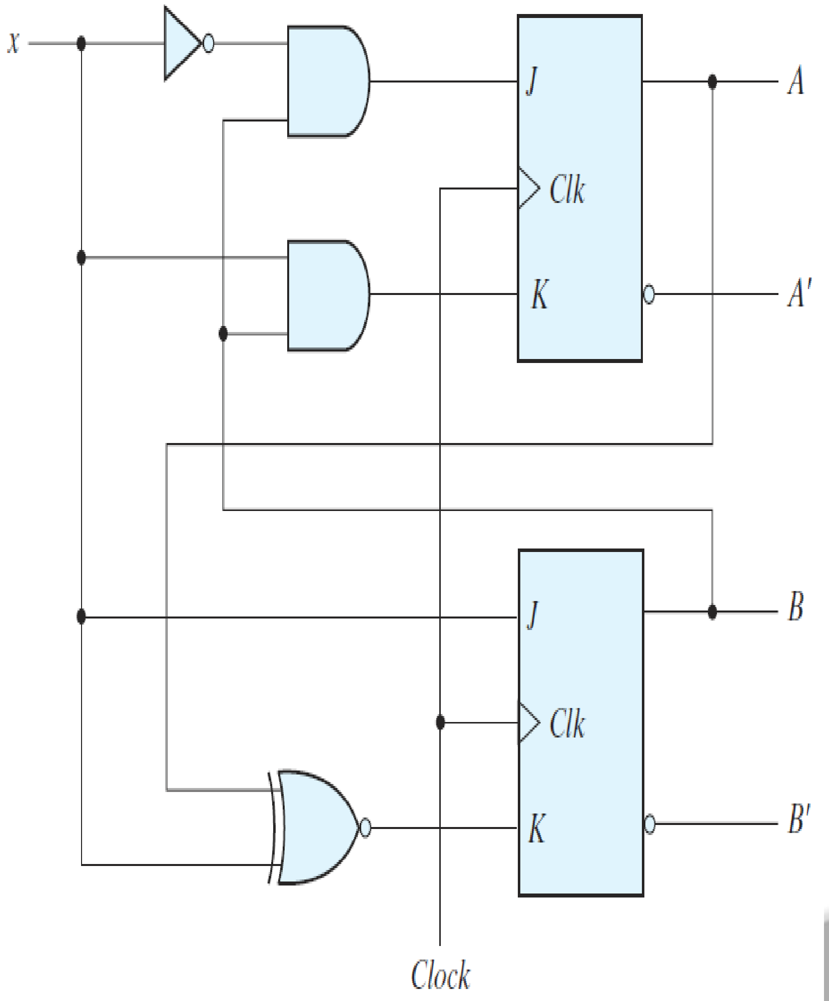
Logic diagram of the sequential circuit:

Advantage of using JK -type flip-flops:

When sequential circuits are designed manually, using JK flip flops is an advantage. The fact that there are so many don't-care entries indicates that the combinational circuit for the input equations is likely to be simpler, because don't-care minterms usually help in obtaining simpler expressions.

If there are unused states in the state table, there will be additional don't-care conditions in the map.

However, D-type flip-flops are more amenable to an automated design flow.



Design of Clocked Sequential Circuits

Synthesis using T Flip-Flops:

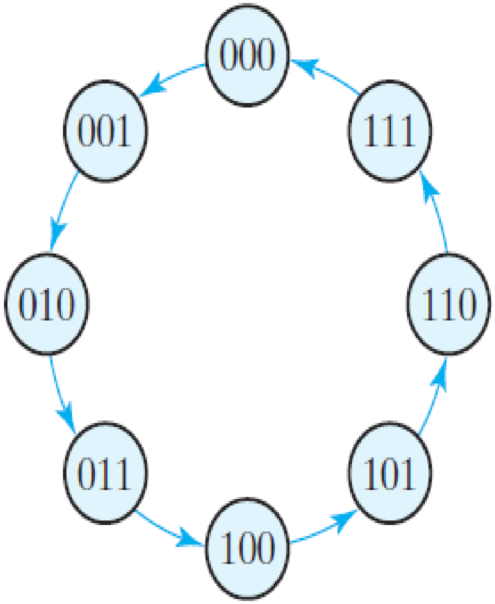
We will explain the procedure with the help of an example.

Example:

Using T flip-flops design a sequential circuit for a 3-bit binary counter.

An n -bit binary counter consists of n flip-flops that can count in binary from ‘0’ to 2^n - 1. A three-bit counter will have, accordingly, 3 flip-flops.

State diagram of a three-bit counter:



We see from the binary states indicated inside the circles that, the flip-flop outputs repeat the binary count sequence with a return to ‘000’ after ‘111’.

We know that state transitions in clocked sequential circuits are initiated by a clock edge; therefore, the flip-flops remain in their present states if no clock is applied.

The only input to the circuit is the clock, and the outputs are specified by the present state of the flip-flops.

The next state of a counter depends entirely on its present state, and the state transition occurs every time the clock goes through a transition.

Design of Clocked Sequential Circuits

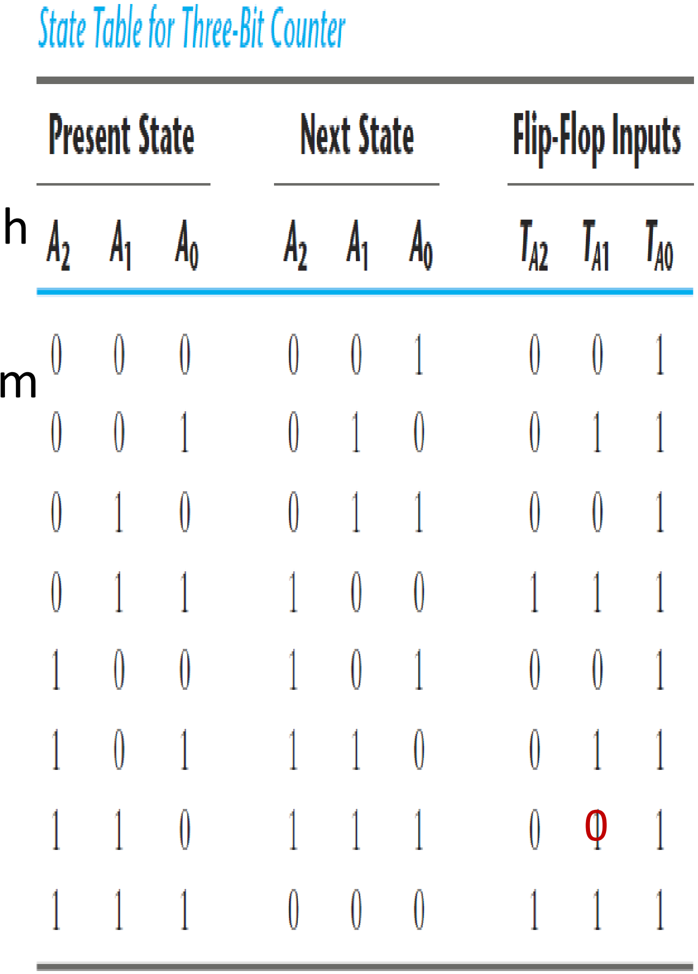
State Table:

The state table for a 3-bit binary counter is shown.

The three flip-flops are symbolized by A_2, A_1, and A_0.

Binary counters are constructed most efficiently with T flip-flops because of their complement property.

The flip-flop excitation for the T inputs is derived from the excitation table of the T flip-flop and by inspection of the state transition of the present state to the next state.



State Table for Three-Bit Counter

Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A2}	T_{A1}	T_{A0}
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

Input equations for flip-flops:

$T_{A2} = A'_2 A_1 A_0 + A_2 A_1 A_0 = m_3 + m_7;$

$T_{A1} = A'_2 A'_1 A_0 + A'_2 A_1 A_0 + A_2 A'_1 A_0 + A_2 A_1 A_0$

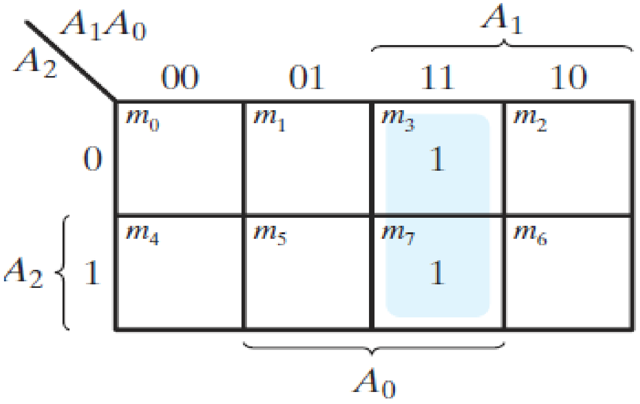
$= m_1 + m_3 + m_5 + m_7;$

T_{A0} = sum of all minterms = 1

Minimization of input equations is done using K-map.

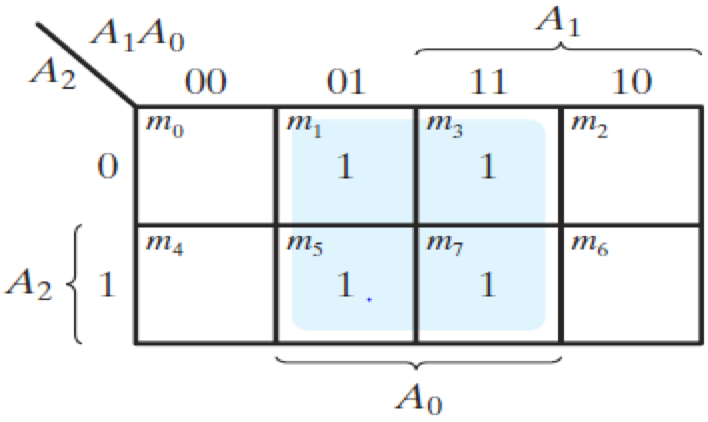
Design of Clocked Sequential Circuits

Minimization:



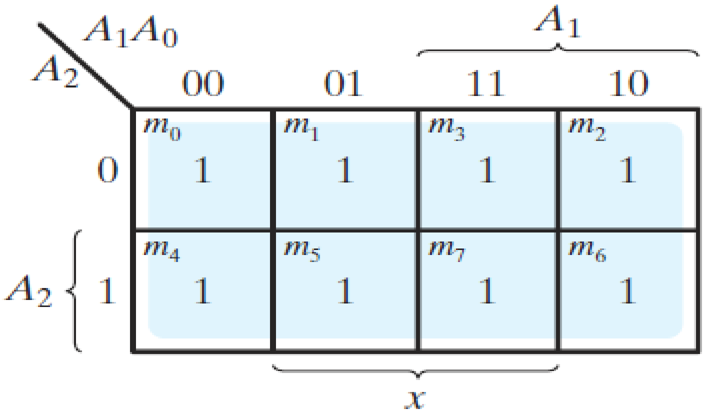
A_2 \ A_1A_0	00	01	11	10
0	m_0	m_1	m_3 1	m_2
1	m_4	m_5	m_7 1	m_6

$T_{A2} = A_1A_0$



A_2 \ A_1A_0	00	01	11	10
0	m_0	m_1 1	m_3 1	m_2
1	m_4	m_5 1	m_7 1	m_6

$T_{A1} = A_0$

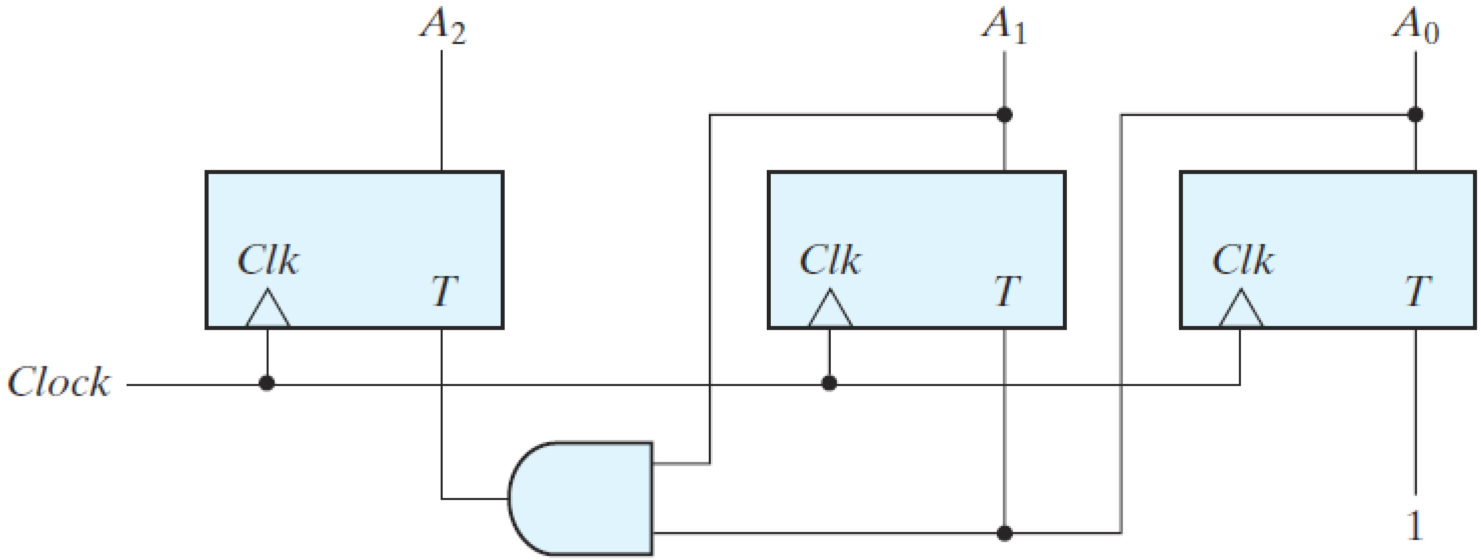


A_2 \ A_1A_0	00	01	11	10
0	m_0 1	m_1 1	m_3 1	m_2 1
1	m_4 1	m_5 1	m_7 1	m_6 1

$T_{A0} = 1$

Design of Clocked Sequential Circuits

Logic diagram:



For simplicity, the reset signal is not shown, but be aware that every design should include a reset signal.

UNIT – V
SUB SYSTEM DESIGN

Shift Registers:

- A register consists of a group of flip-flops with a common clock input.
- Registers are commonly used to store and shift binary data.
  - Example: Counters

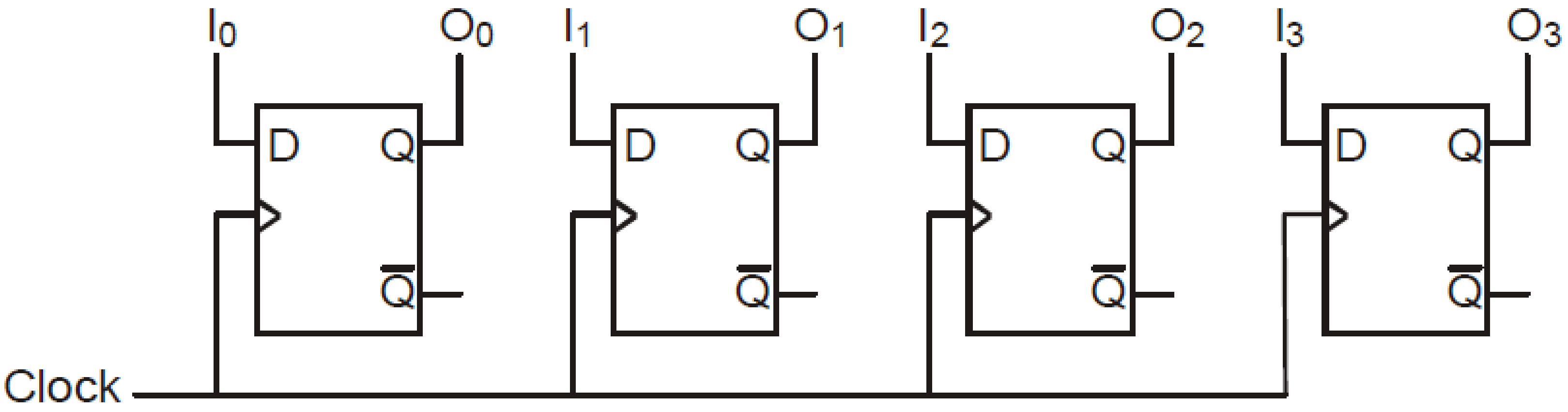


Fig. 4-bit D register

Registers: Types

- **Buffer Register.**

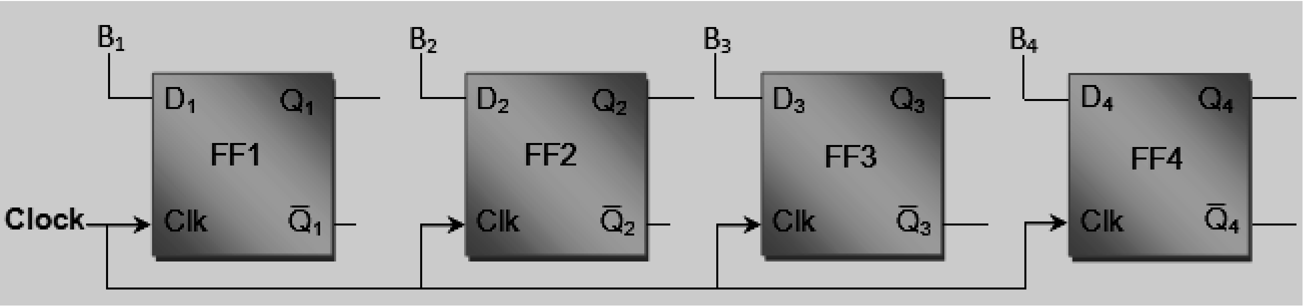


Fig. 1. 4-bit Buffer register

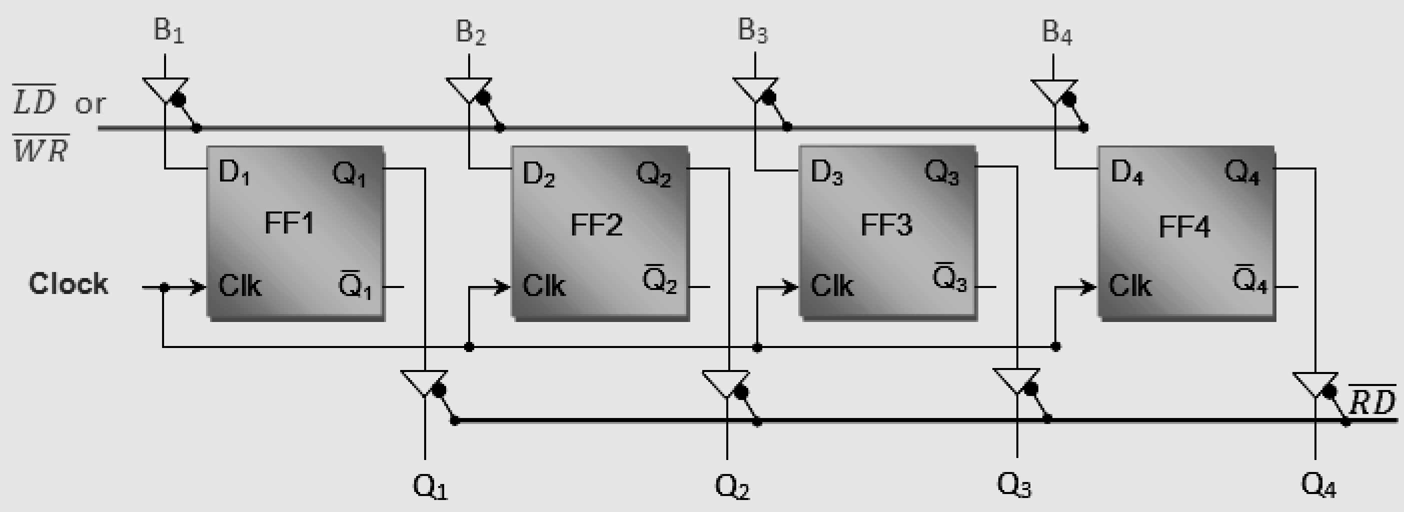


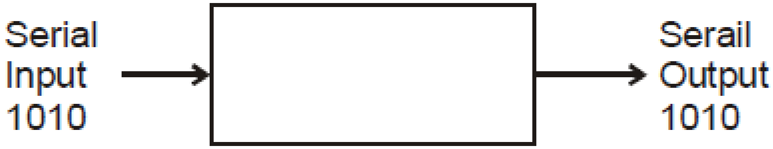
Fig. 2. 4-bit Controlled buffer register

Shift Registers:

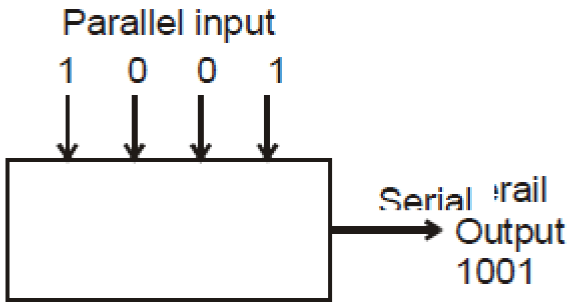
- **A shift register is a register in which binary data can be stored, and this data can be shifted to the left or right when a shift signal is applied.**

Shift Register: Types

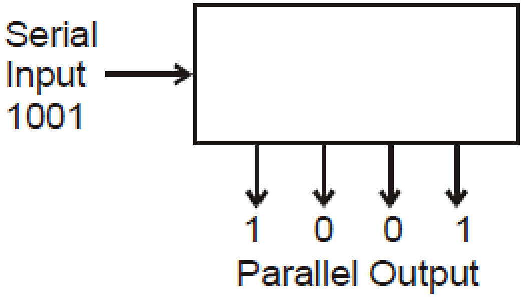
Serial In Serial Out Shift Register



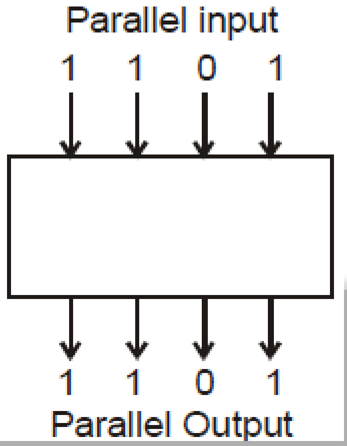
Parallel In Serial Out Shift Register



Serial In Parallel Out Shift Register



Parallel In Parallel Out Shift Register



1. Serial In Serial Out Shift Register

Serial
Input
1010
Serail
Output
1010

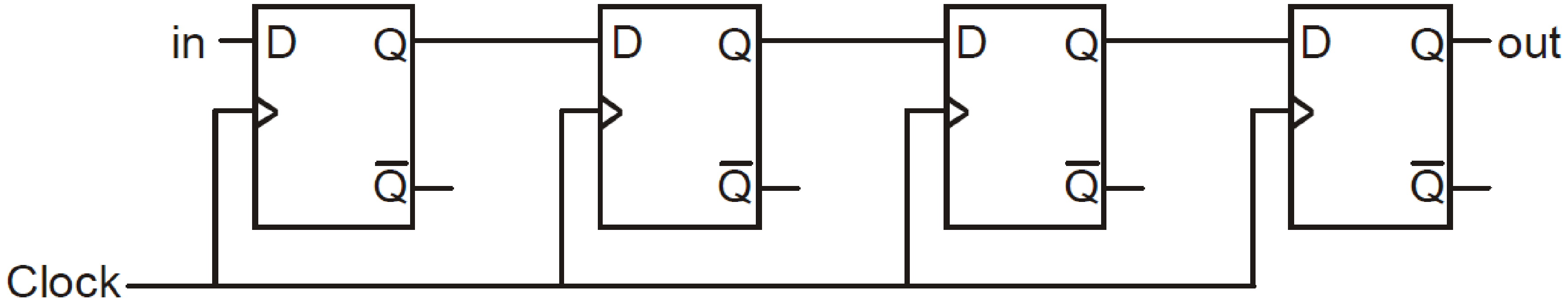


Fig. 4-bit serial-in serial-out shift register

1. Serial In Serial Out Shift Register

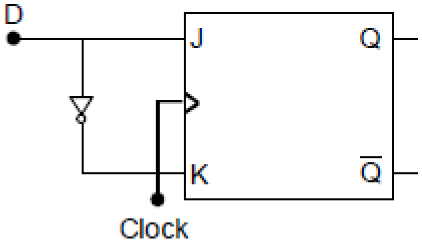


Fig. 1. JK Flip-flop converted into D-Flip-Flop

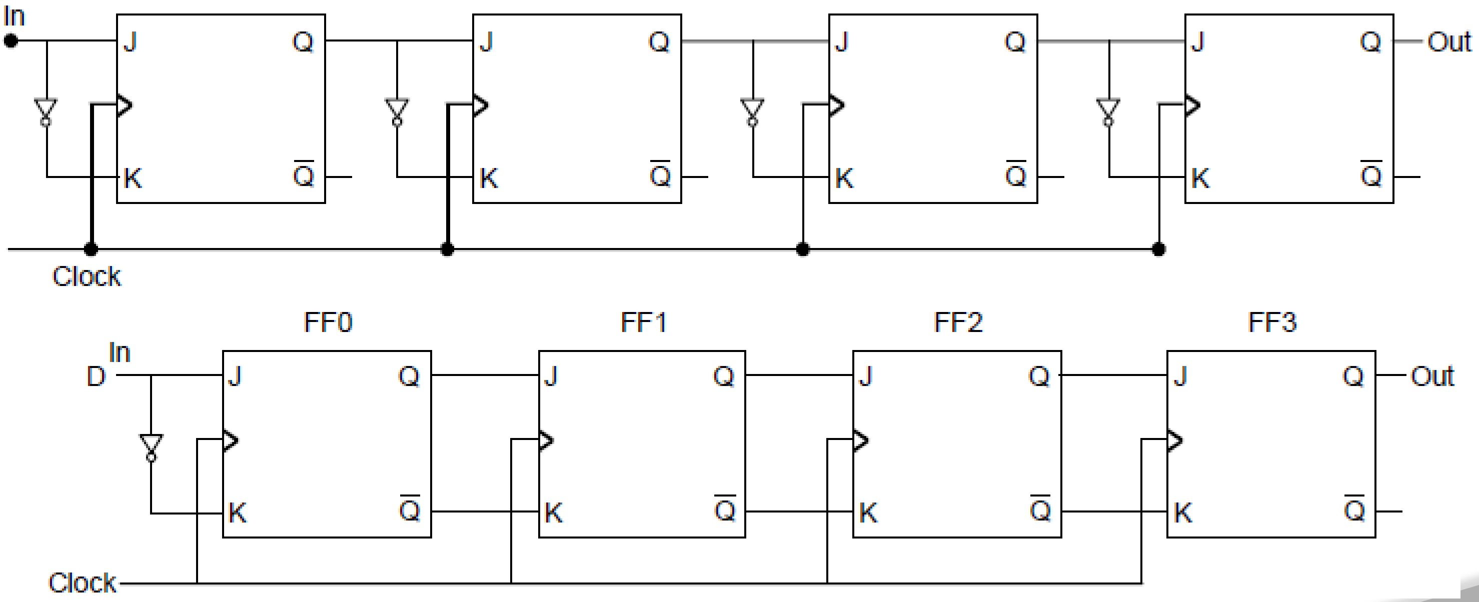
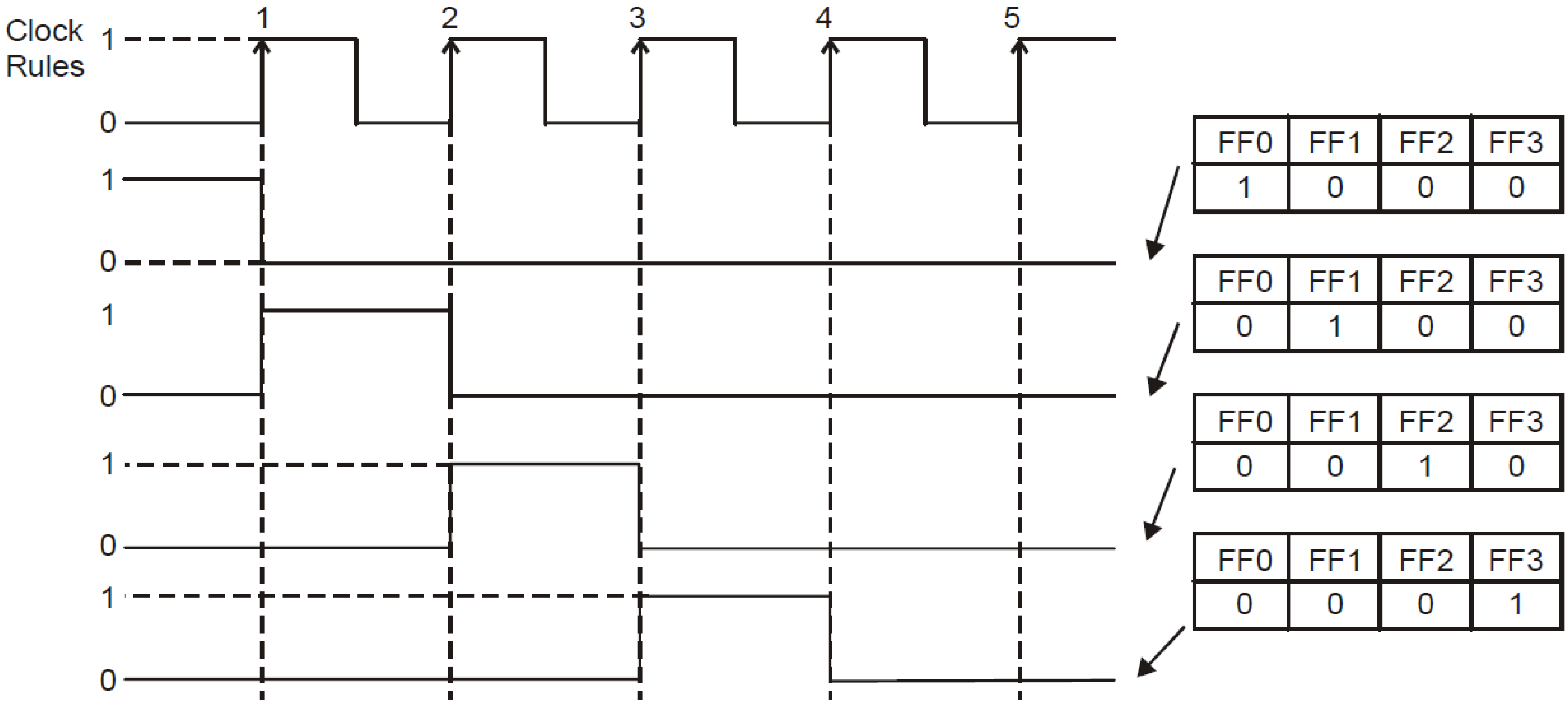


Fig. 2. 4-bit serial in serial out shift register using JK Flip-Flop

1. Serial In Serial Out Shift Register



FF0	FF1	FF2	FF3
1	0	0	0

FF0	FF1	FF2	FF3
0	1	0	0

FF0	FF1	FF2	FF3
0	0	1	0

FF0	FF1	FF2	FF3
0	0	0	1

Fig. 1. A 4-bit shift register operation

Right-Shift Register

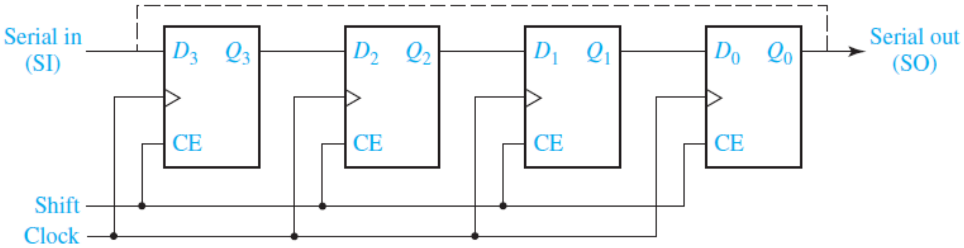


Fig. 1. Flip-flop connections

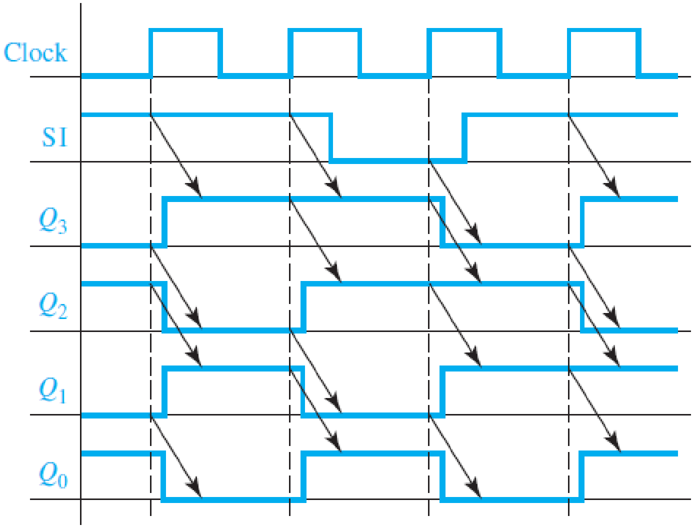
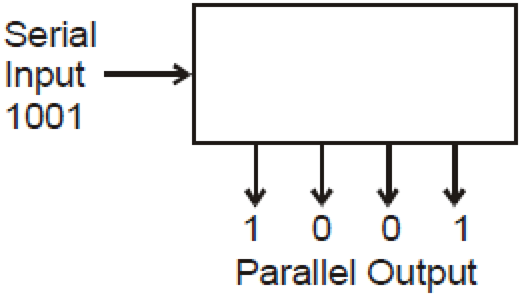


Fig. 2. Timing diagram

2. Serial In Parallel Out Shift Register



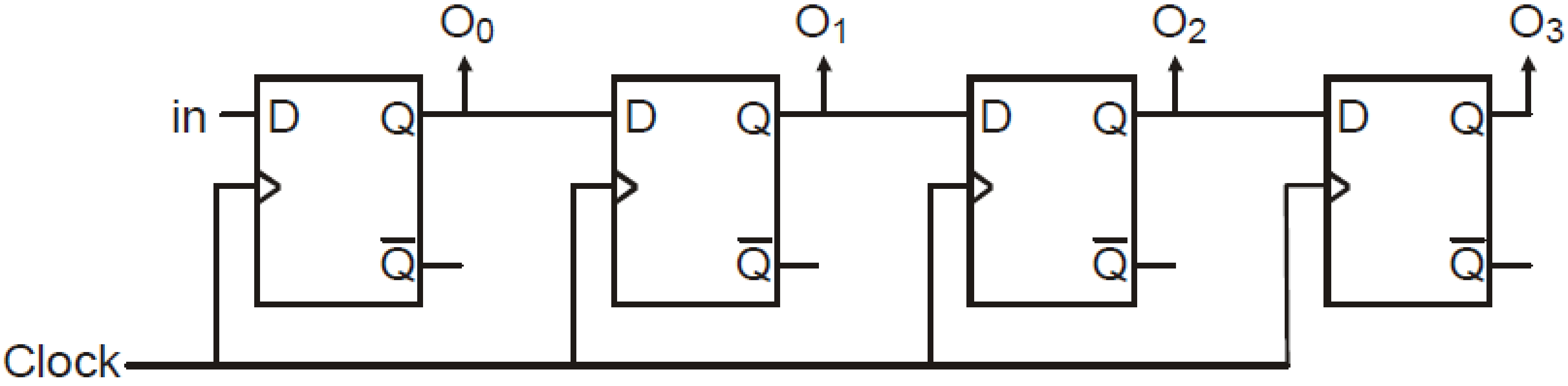


Fig. 4-bit serial-in parallel-out shift register

2. Serial In Parallel Out Shift Register

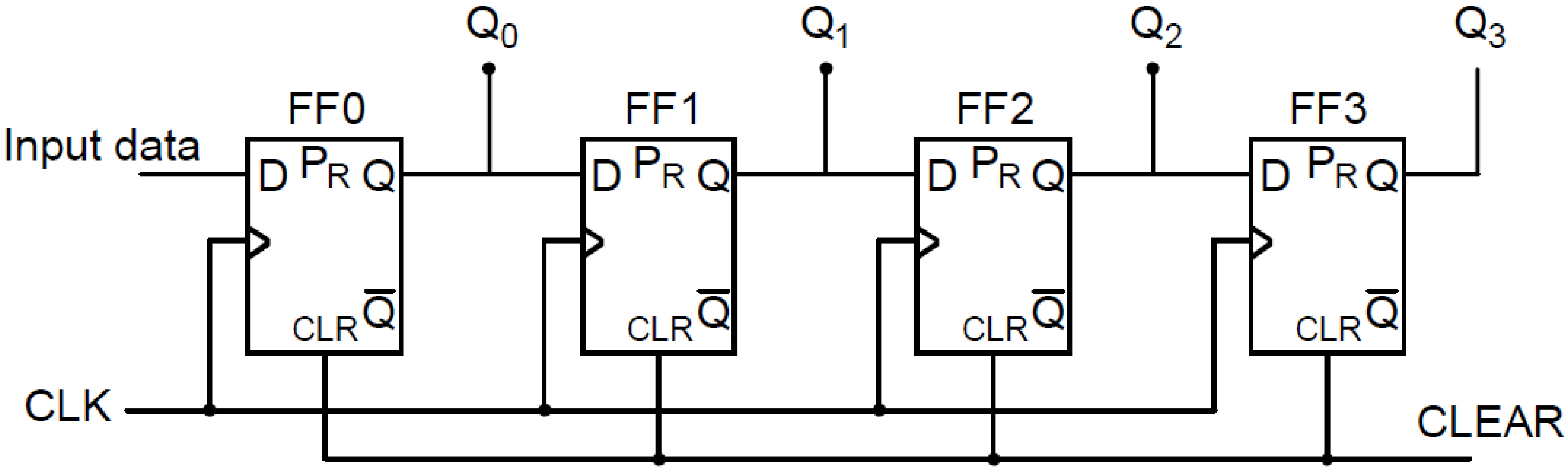
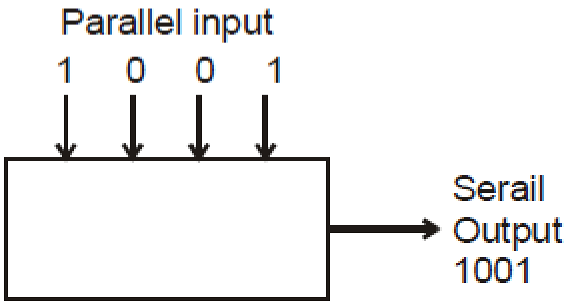


Fig. 4-bit serial-in parallel-out shift register

3. Parallel In Serial Out Shift Register



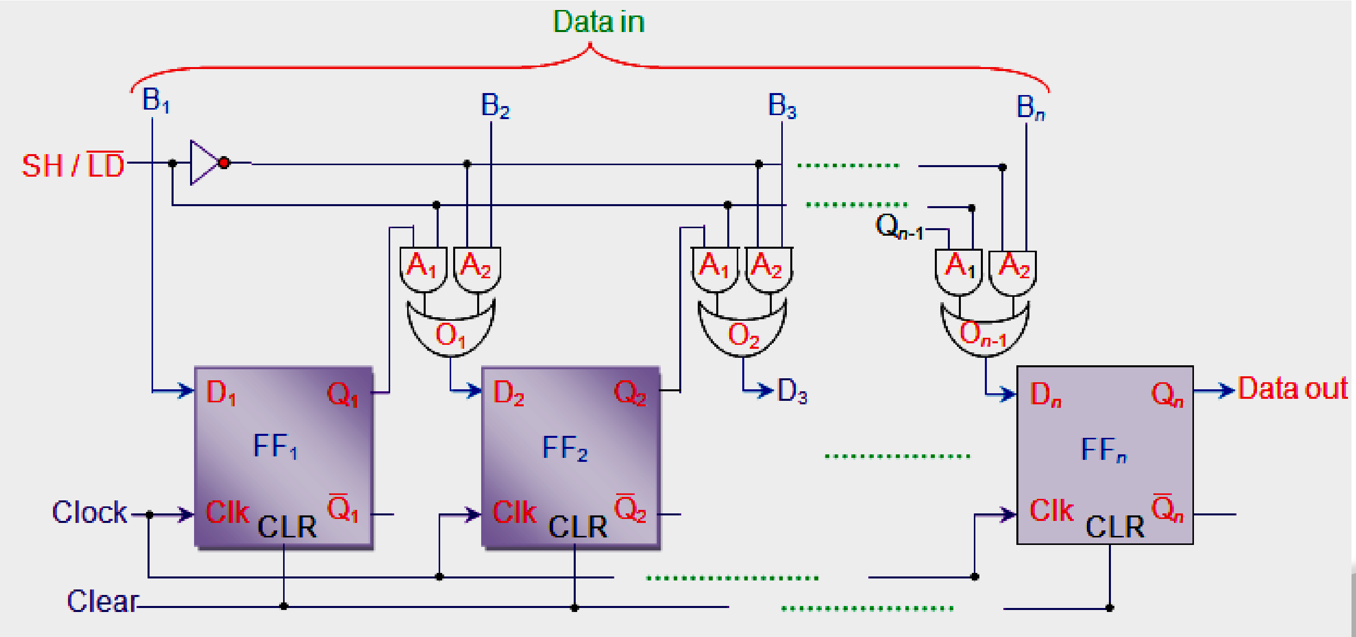


Figure 1 *n*-bit Parallel-In Serial-Out Right-Shift Shift Register

4. Parallel In Parallel Out Shift Register

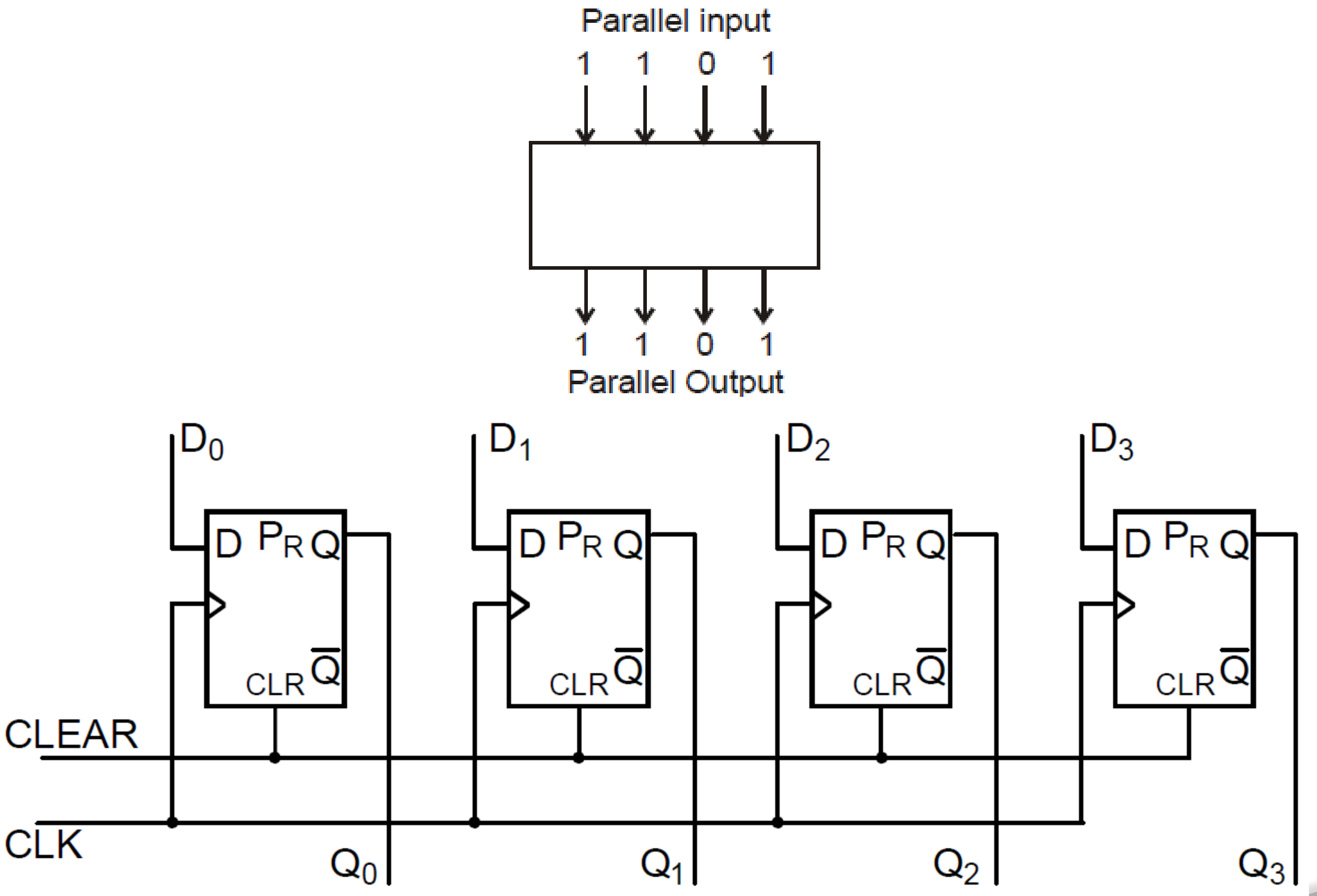
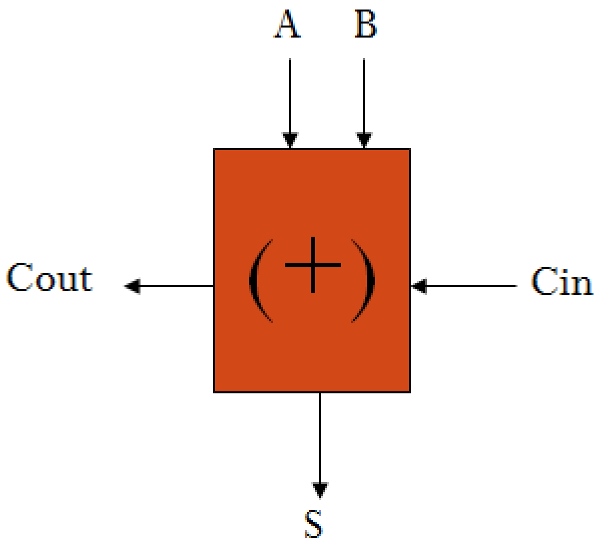


Fig. 4-bit parallel-in parallel-out shift register

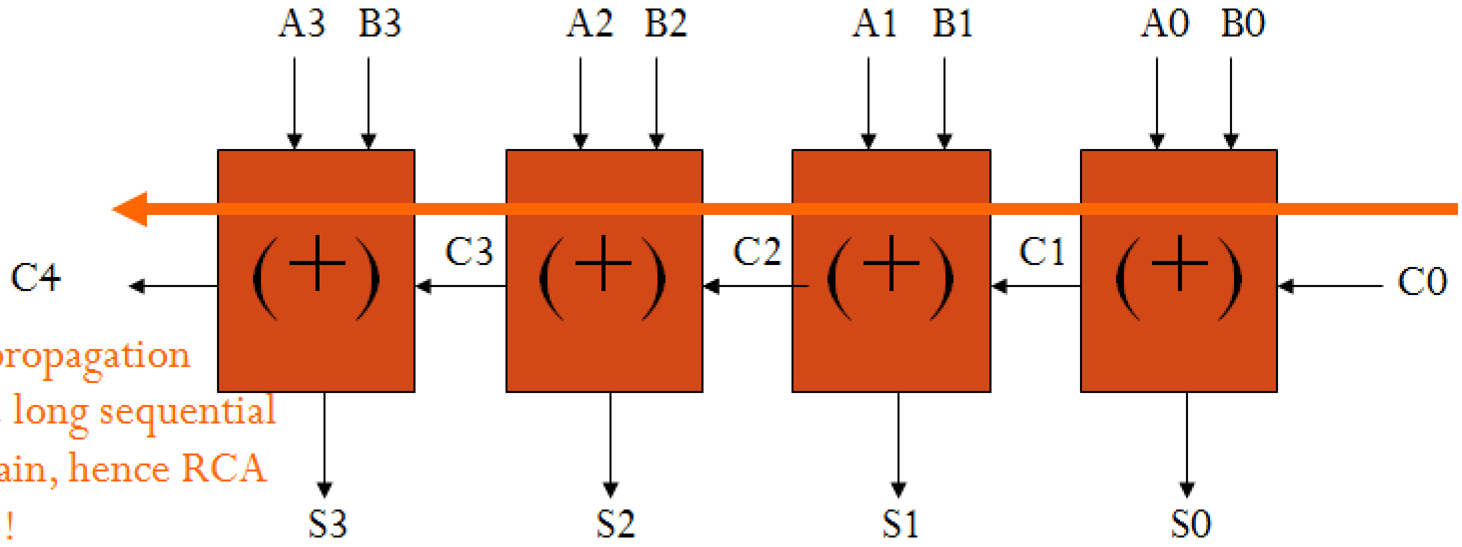
CARRY LOOK AHEAD ADDER

A simple one-bit full adder



- It takes A, B, and Cin as input and generates S and Cout in 2 gate delays (SOP)

4-bit RCA



Carry propagation forms a long sequential wait chain, hence RCA Is slow!!

- Work from lowest bit to highest bit sequentially.
- With A0, B0, and C0, the lowest bit adder generates S0 and C1 in 2 gate delay.
- With A1, B1, and C1 ready, the second bit adder generates S1 and C2 in 2 gate delay.
- Each bit adder has to wait for the lower bit adder to propagate the carry.

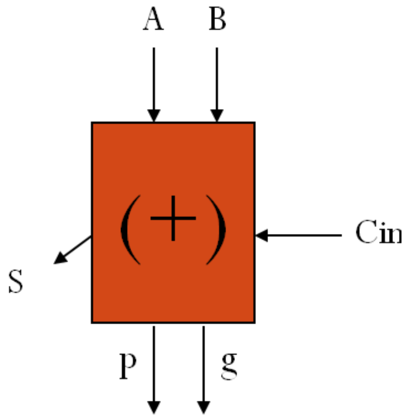
Observations

- **The critical component each bit adder waits for is the carry input.**
- **Instead of generating and propagating carry bit-by-bit, can we generate all of them in parallel and break the sequential chain?**
- **This is exactly the idea of CLA (carry look-ahead adder).**

Carry Look Ahead Logic

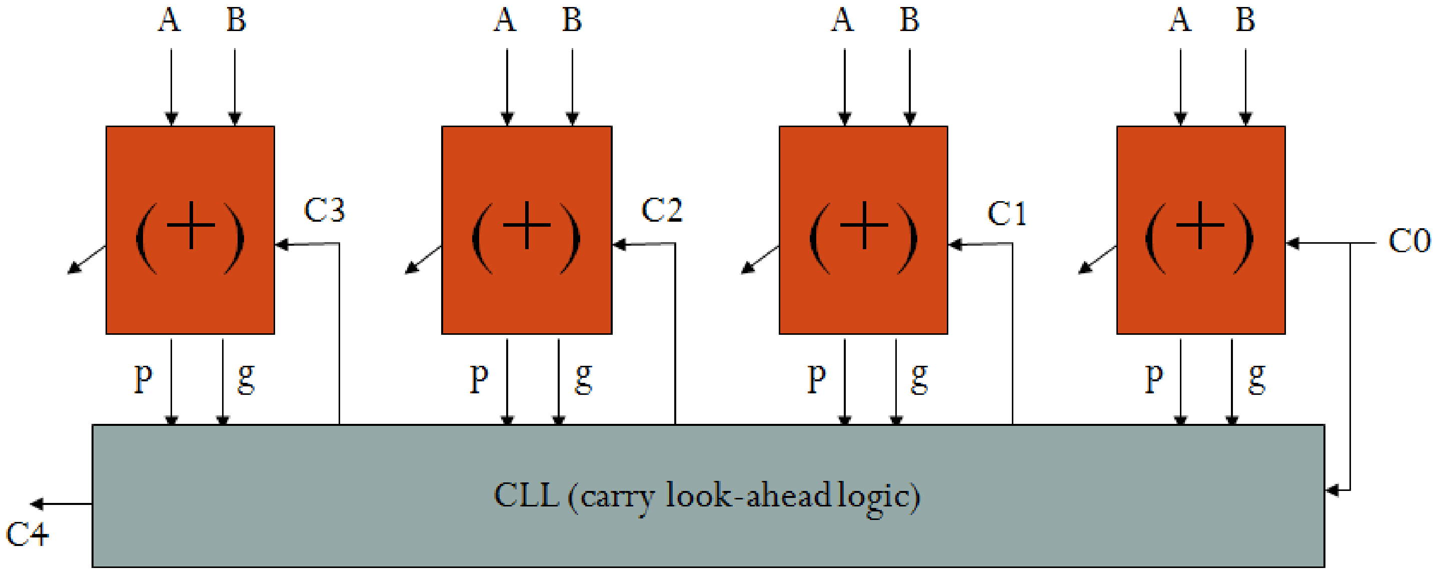
- Now even before the carry in (Cin) is available, based on the inputs (A,B) only, can we say anything about the carry out?
- Under what condition will the bit propagate an outgoing carry (Cout), if there is an incoming carry (Cin)?
- Under what condition will the bit generate an outgoing carry (Cout), regardless of whether there is an incoming carry (Cin)?

1-bit CLA adder



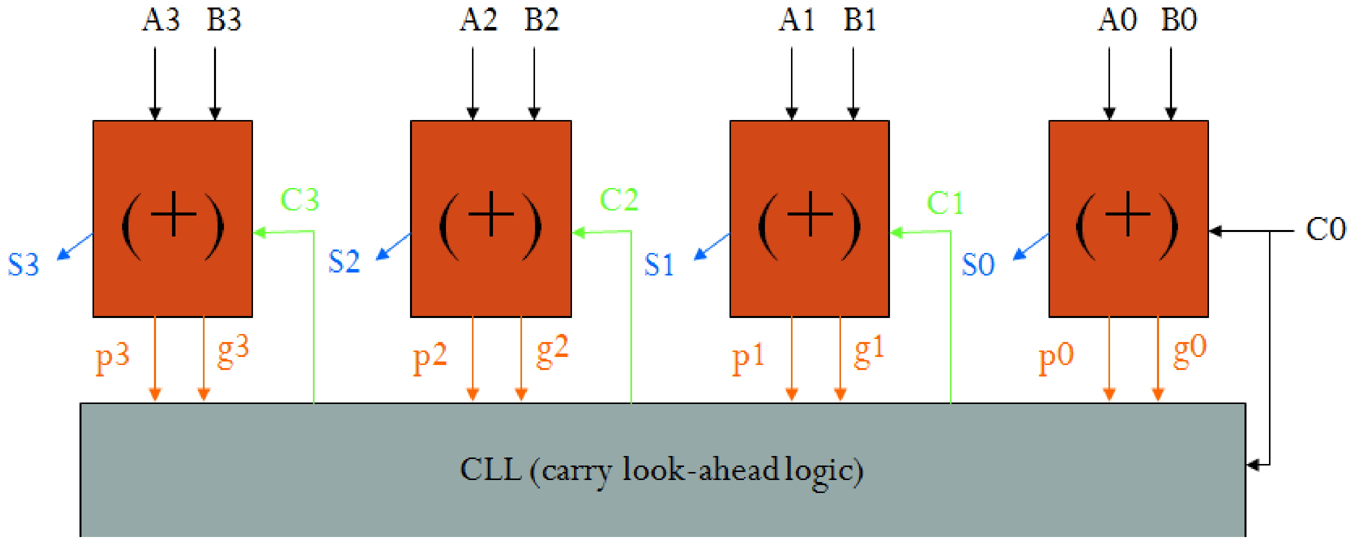
- Instead of Cout, an 1-bit CLA adder block takes A, B inputs and generates p,g
- p=propagator =>I will propagate the Cin to the next bit. p = A+B (If either A or B is 1, Cin=1 causes Cout=1)
- g=generator =>I will generate a Cout independent of what Cin is. g = AB (If both A and B are 1, Cout=1 for sure)
- p,g are generated in 1 gate delay after we have A,B. Note that Cin is not needed to generate p,g.
- S is generated in 2 gate delay after we get Cin (SOP).

4-bit CLA



- The CLL takes p,g from all 4 bits and C0 as input to generate all Cs in 2 gate delay.
- C1=g0+p0C0,
- C2=g1+p1g0+p1p0C0,
- C3=g2+p2g1+p2p1g0+p2p1p0c0,
- C4=g3+p3g2+p3p2g1+p3p2p1g0+p3p2p1p0c0 (Note: this C4 is too complicated to generate in 2-level SOP representation)

4-bit CLA



- Given A,B’s, all p,g’s are generated in 1 gate delay in parallel.
- Given all p,g’s, all C’s are generated in 2 gate delay in parallel.
- Given all C’s, all S’s are generated in 2 gate delay in parallel.
- Key virtue of CLA: sequential operation in RCA is broken into parallel operation

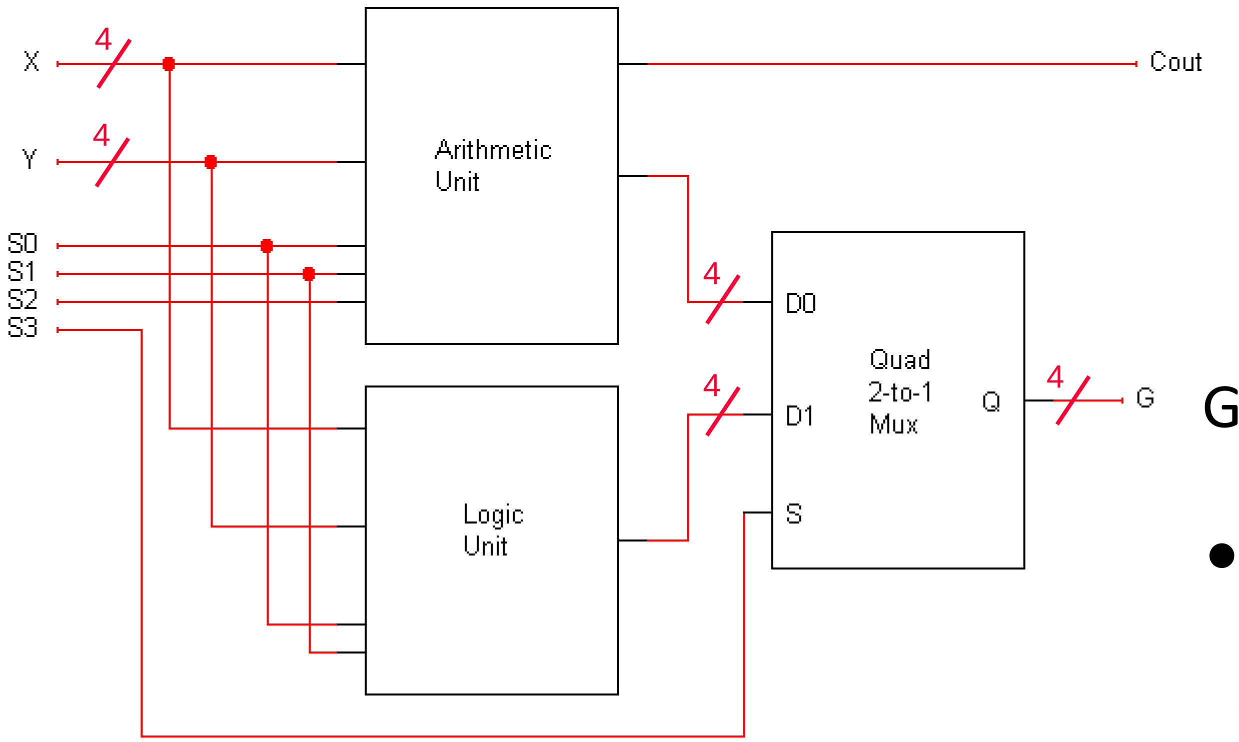
ALU function table

- This table shows a sample function table for an ALU.
- All of the arithmetic operations have S_3=0, and all of the logical operations have S_3=1.
- These are the same functions we saw when we built our arithmetic and logic units a few minutes ago.
- Since our ALU only has 4 logical operations, we don't need S_2. The operation done by the logic unit depends only on S_1 and S_0.

S_3	S_2	S_1	S_0	Operation
0	0	0	0	$G = X$
0	0	0	1	$G = X + 1$
0	0	1	0	$G = X + Y$
0	0	1	1	$G = X + Y + 1$
0	1	0	0	$G = X + Y'$
0	1	0	1	$G = X + Y' + 1$
0	1	1	0	$G = X - 1$
0	1	1	1	$G = X$
1	x	0	0	$G = X$ and Y
1	x	0	1	$G = X$ or Y
1	x	1	0	$G = X \oplus Y$
1	x	1	1	$G = X'$

A complete ALU circuit

The / and 4 on a line indicate that it’s actually *four* lines.



C_{out} should be ignored when logic operations are performed (when S3=1).

G is the final ALU output.

- When S3 = 0, the final output comes from the arithmetic unit.
- When S3 = 1, the output comes from the logic unit.

The arithmetic and logic units share the select inputs S1 and S0, but only the arithmetic unit uses S2.

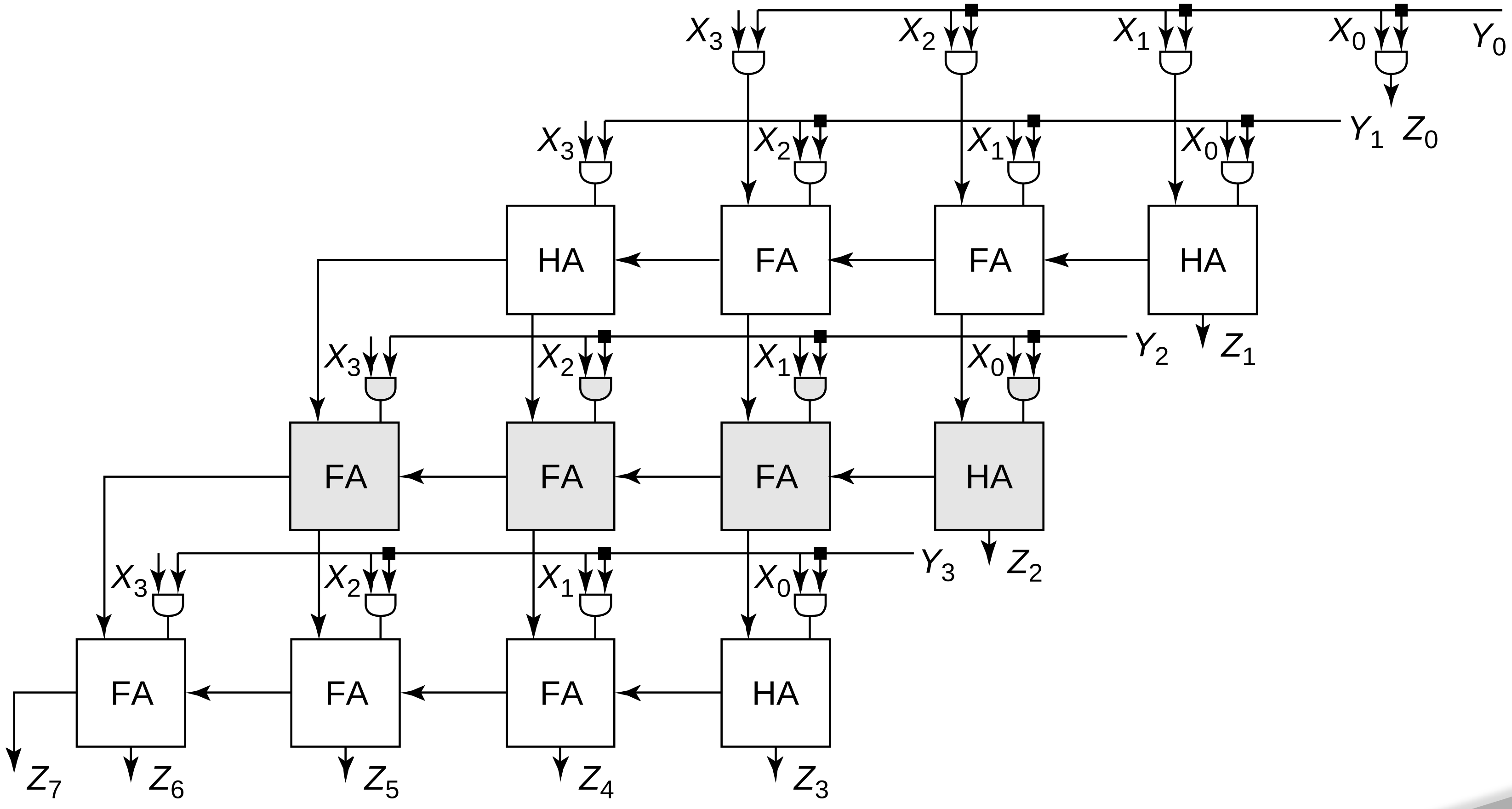
Multiplier Design: Basic Building Blocks

- Datapath
- Execution units
- Adder, multiplier, divider, shifter, etc.
- Register file and pipeline registers
- Multiplexers, decoders
- Control
- Finite state machines (PLA, ROM, random logic)
- Interconnect
- Switches, arbiters, buses
- Memory
- Caches (SRAMs), TLBs, DRAMs, buffers

The Binary Multiplication

$$\begin{array}{cccccccccc l}
 & & & & 1 & 0 & 1 & 0 & 1 & 0 & \text{Multiplicand} \\
x & & & & & & 1 & 0 & 1 & 1 & \text{Multiplier} \\
\hline
 & & & & 1 & 0 & 1 & 0 & 1 & 0 & \\
 & & & 1 & 0 & 1 & 0 & 1 & 0 & & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & & & \text{Partial products} \\
+ & 1 & 0 & 1 & 0 & 1 & 0 & & & & \\
\hline
 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & \text{Result}
\end{array}$$

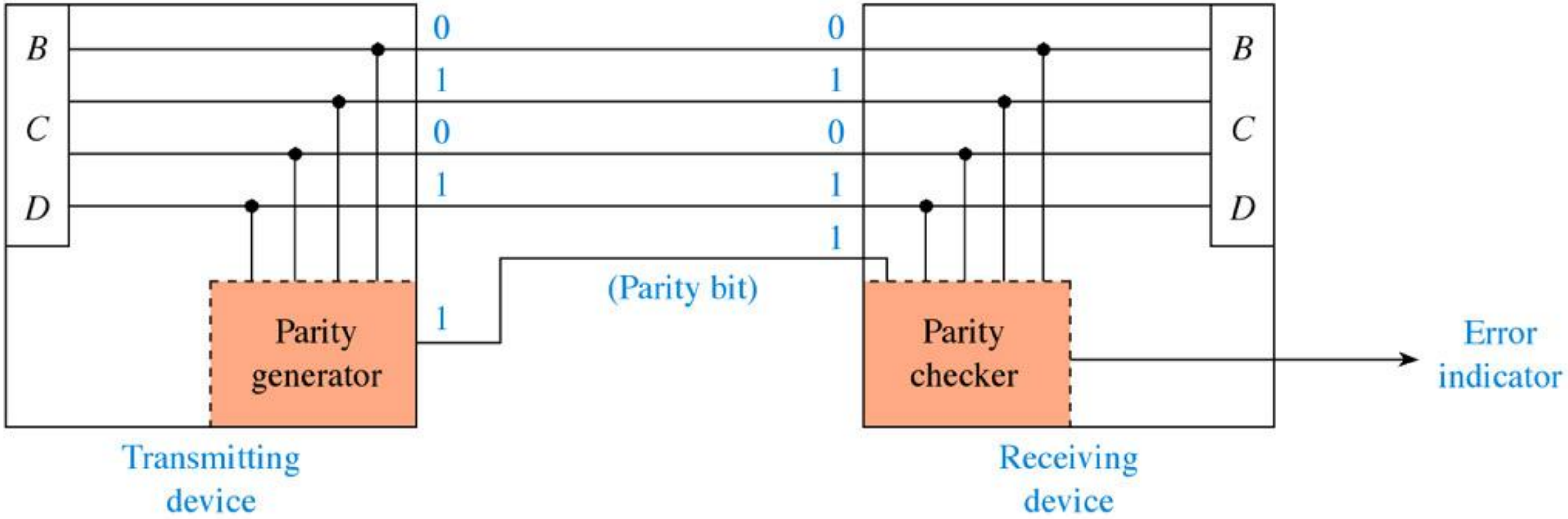
The Array Multiplier:



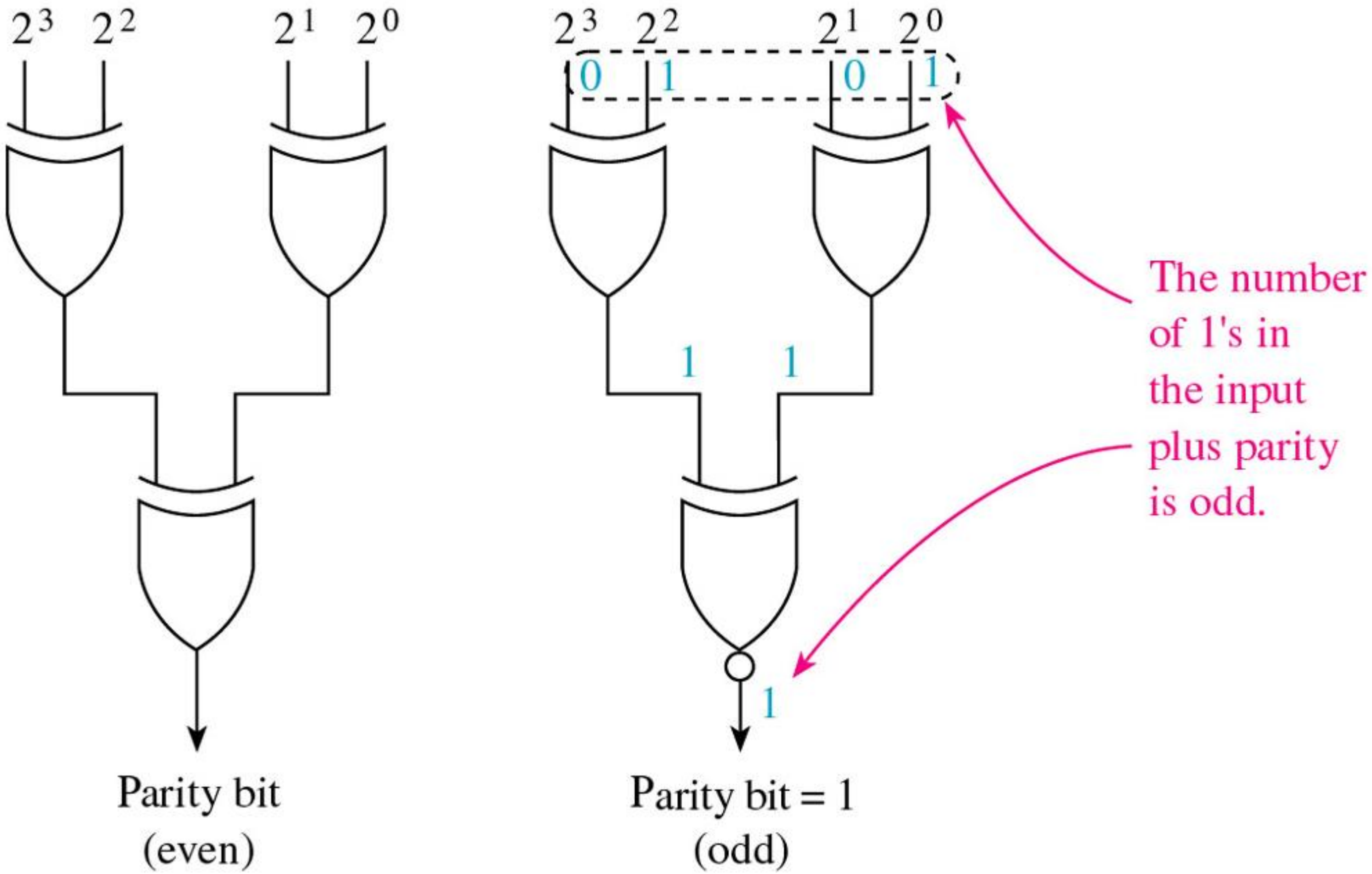
Parity Generator / Checker

- Electrical noise in the transmission of binary information can cause errors. Parity can detect these types of errors.
- Parity systems
- Odd parity
- Even parity
- Adds a bit to the binary information

Parity Generator / Checker



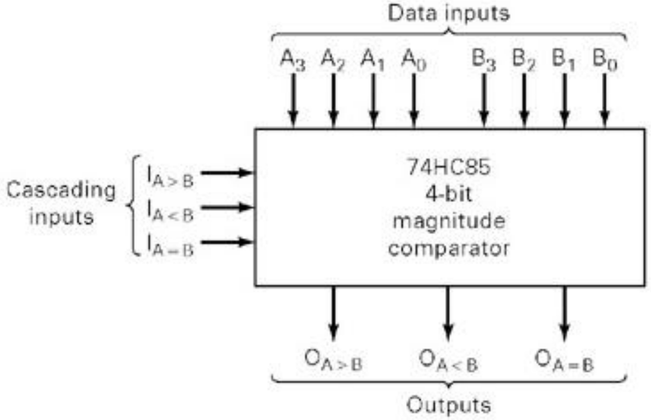
Four-bit even- and odd-parity generators



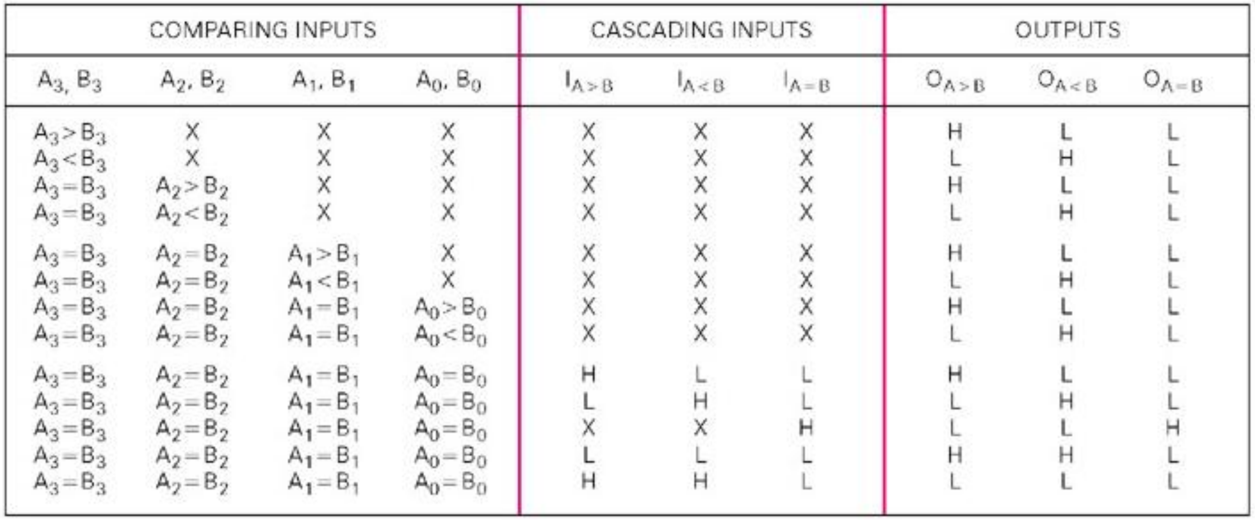
Comparator:

- A comparator is a precision instrument employed to compare the dimension of a given component with a working standard (usually slip gauges).
- It thus does not measure the actual dimension but indicate how much it differs from the basic dimension.

Magnitude Comparator



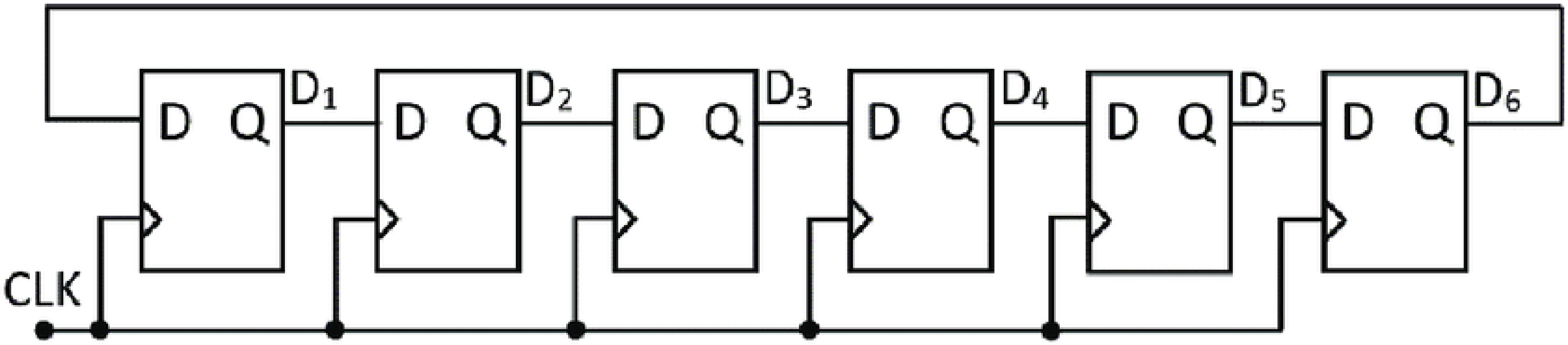
TRUTH TABLE

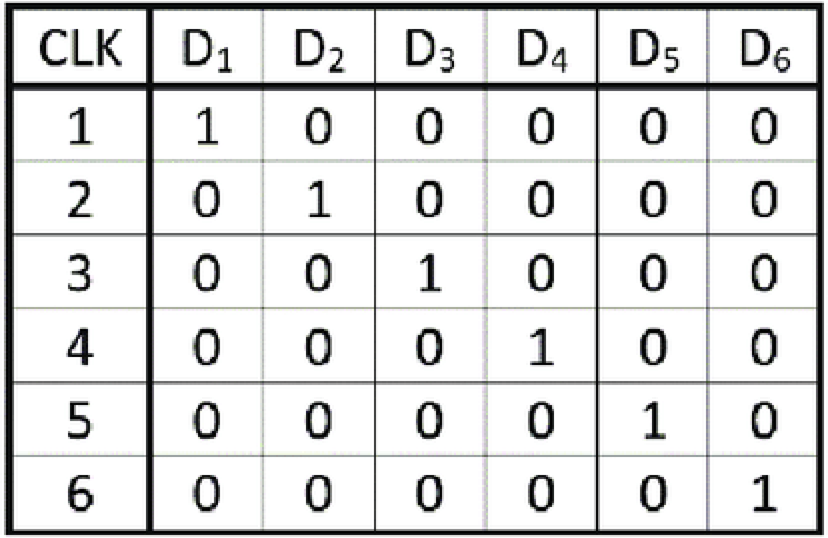


COMPARING INPUTS				CASCADING INPUTS			OUTPUTS		
A_3, B_3	A_2, B_2	A_1, B_1	A_0, B_0	$I_{A>B}$	$I_{A<B}$	$I_{A=B}$	$O_{A>B}$	$O_{A<B}$	$O_{A=B}$
$A_3>B_3$	X	X	X	X	X	X	H	L	L
$A_3<B_3$	X	X	X	X	X	X	L	H	L
$A_3=B_3$	$A_2>B_2$	X	X	X	X	X	H	L	L
$A_3=B_3$	$A_2<B_2$	X	X	X	X	X	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1>B_1$	X	X	X	X	H	L	L
$A_3=B_3$	$A_2=B_2$	$A_1<B_1$	X	X	X	X	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0>B_0$	X	X	X	H	L	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0<B_0$	X	X	X	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	H	L	L	H	L	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	L	H	L	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	X	X	H	L	L	H
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	L	L	L	H	H	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	H	H	L	L	L	L

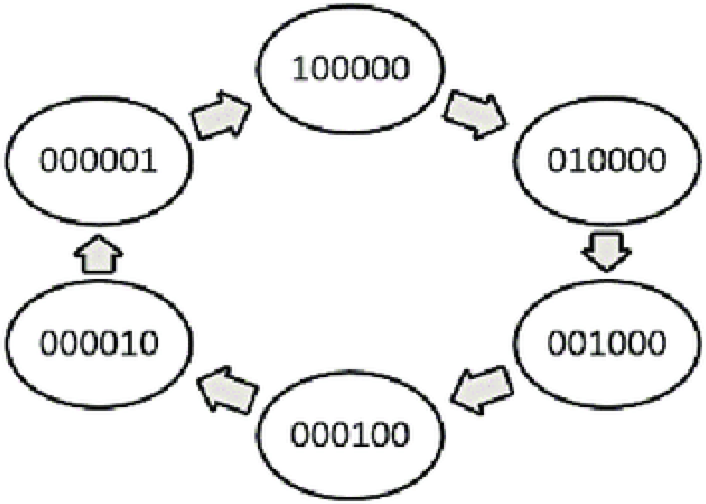
H = HIGH Voltage Level
L = LOW Voltage Level
X = Immaterial

Ring Counter

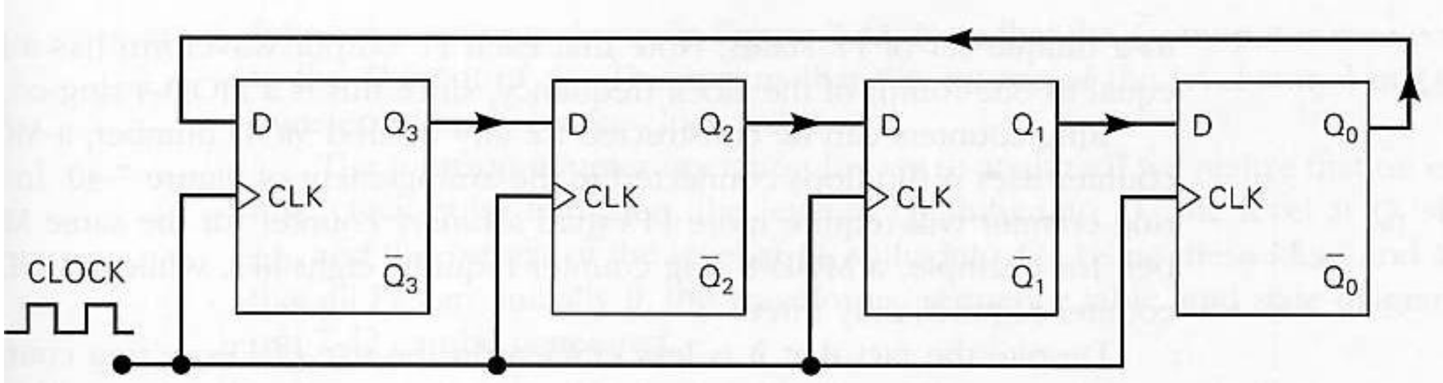


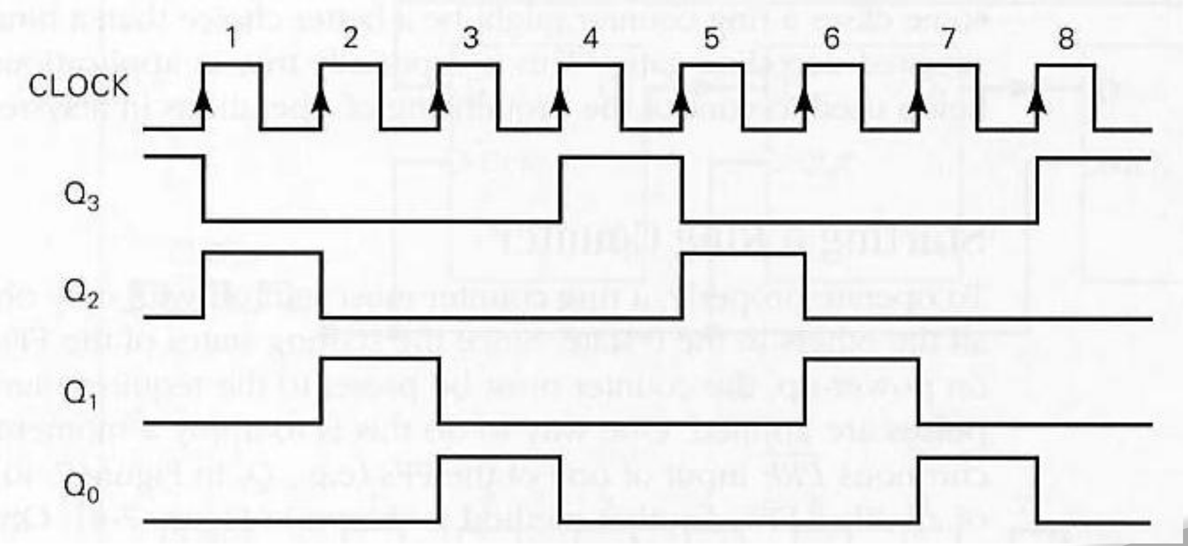


CLK	D_1	D_2	D_3	D_4	D_5	D_6
1	1	0	0	0	0	0
2	0	1	0	0	0	0
3	0	0	1	0	0	0
4	0	0	0	1	0	0
5	0	0	0	0	1	0
6	0	0	0	0	0	1

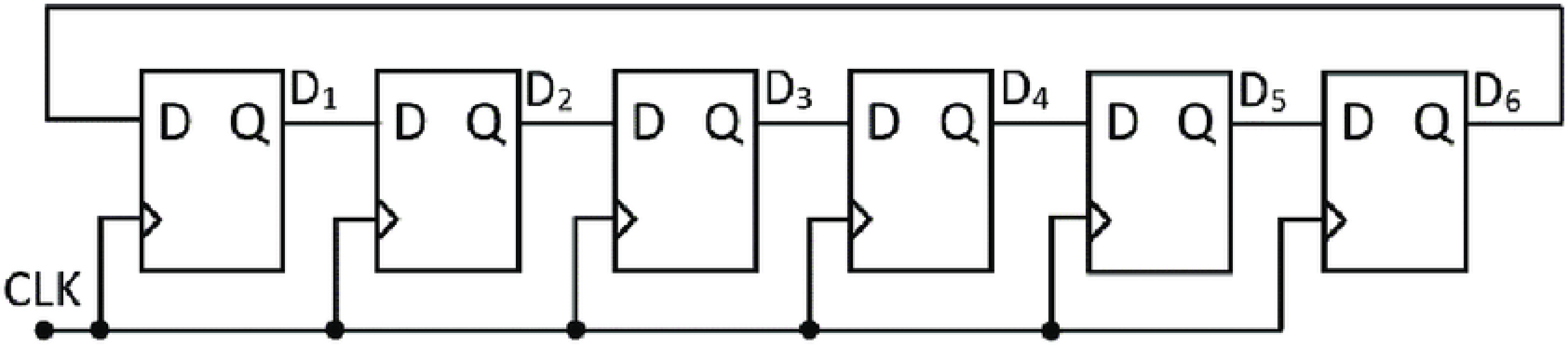


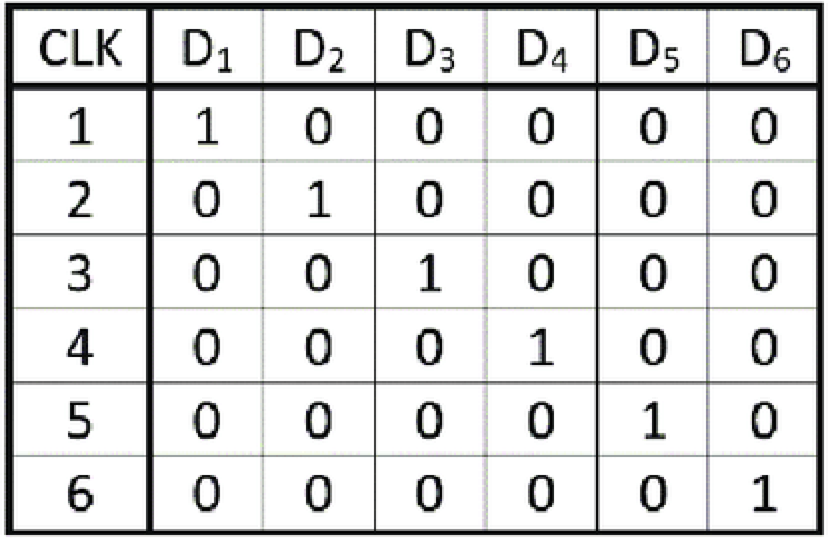
Ring Counter



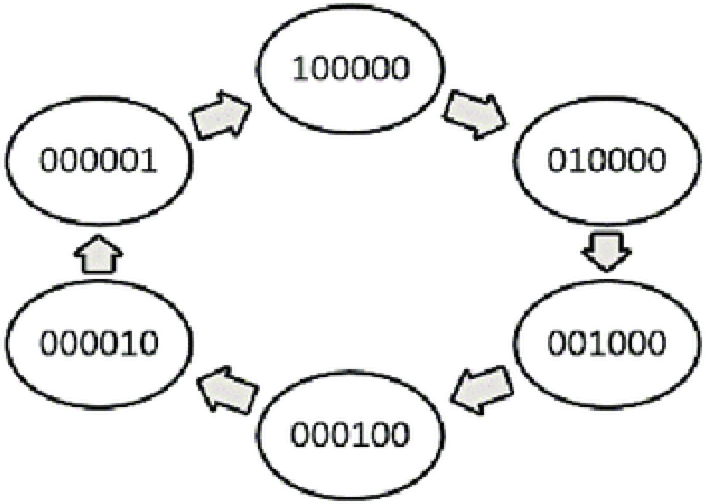


Ring Counter

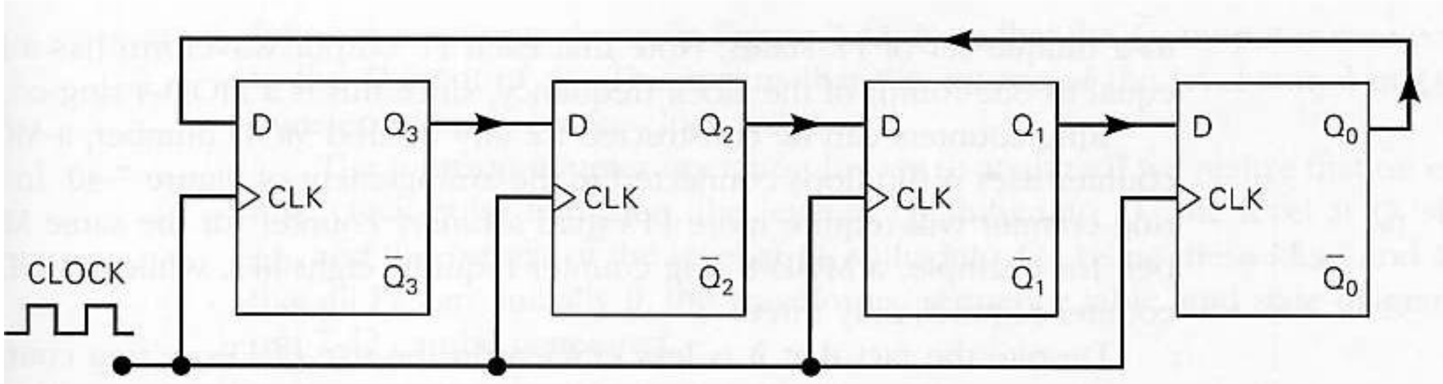


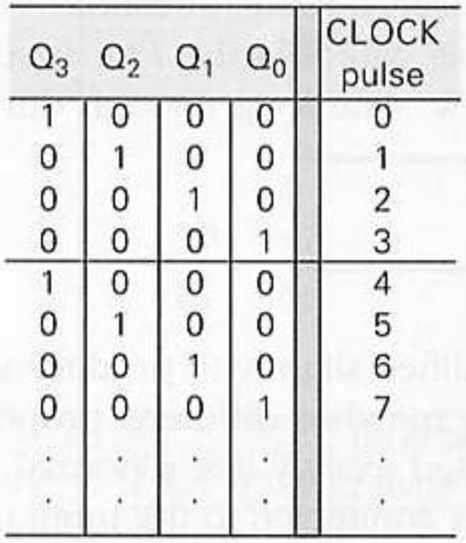


CLK	D_1	D_2	D_3	D_4	D_5	D_6
1	1	0	0	0	0	0
2	0	1	0	0	0	0
3	0	0	1	0	0	0
4	0	0	0	1	0	0
5	0	0	0	0	1	0
6	0	0	0	0	0	1



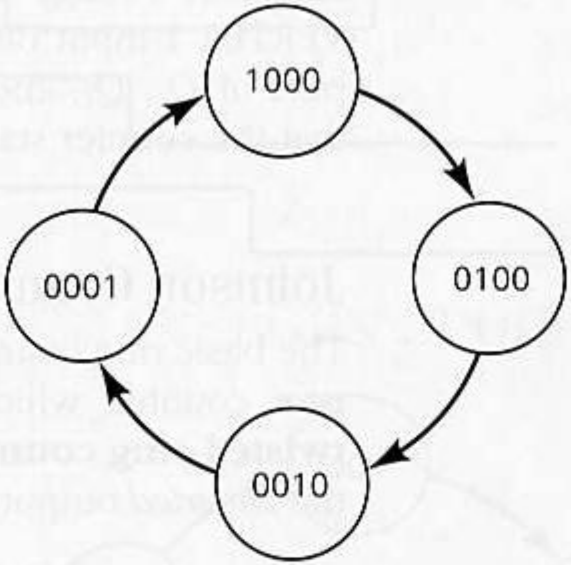
Ring Counter





Q_3	Q_2	Q_1	Q_0	CLOCK pulse
1	0	0	0	0
0	1	0	0	1
0	0	1	0	2
0	0	0	1	3
1	0	0	0	4
0	1	0	0	5
0	0	1	0	6
0	0	0	1	7
.	.	.	.	.
.	.	.	.	.

(c)



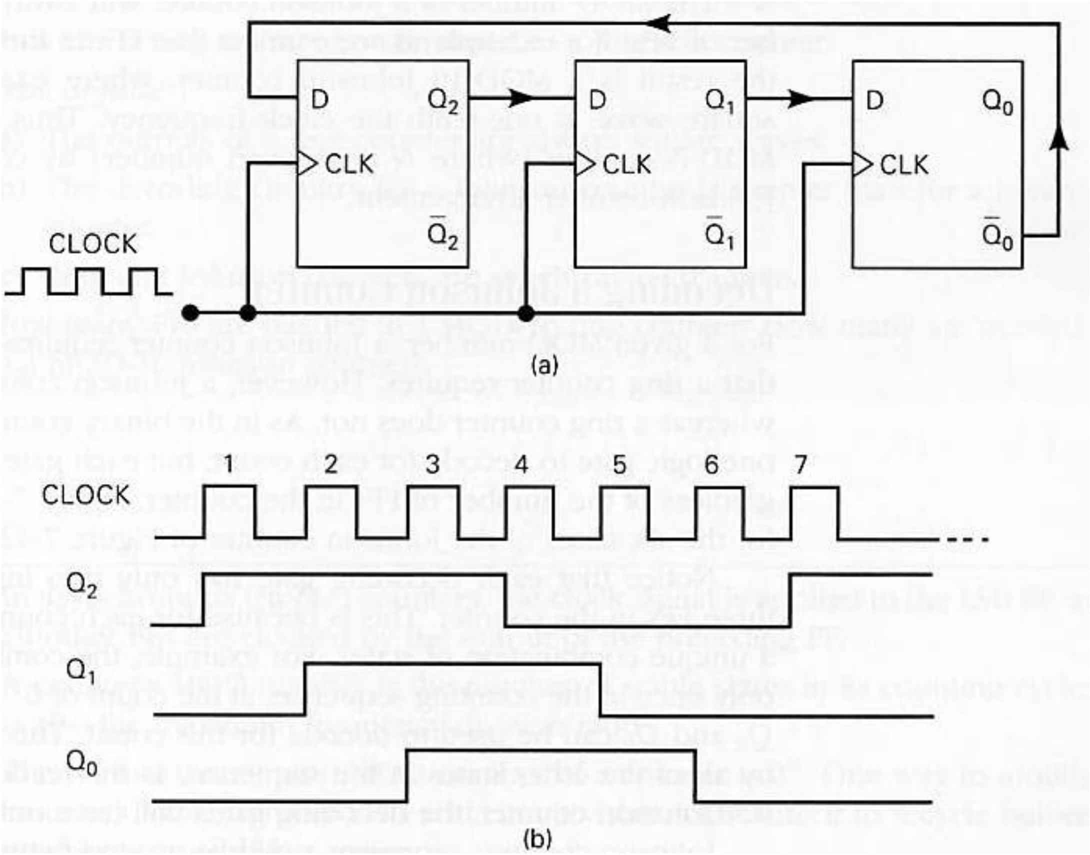
(d)

Johnson Counter

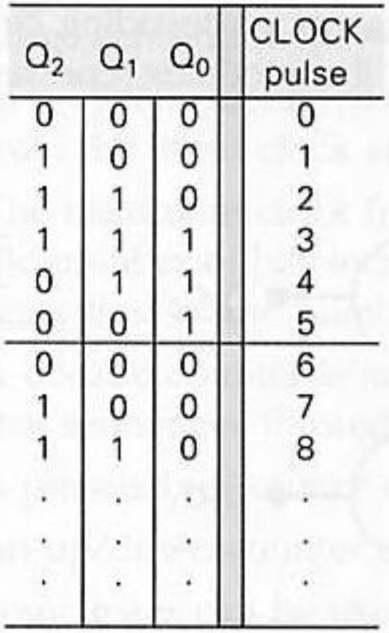
- **Also known as the twisted-ring counter.**
- **Same as the ring counter except that the inverted output of the last FF is connected to the input of the first FF.**
- **Counting sequence:**

  000→100→110→111→011→001→000

Johnson Counter

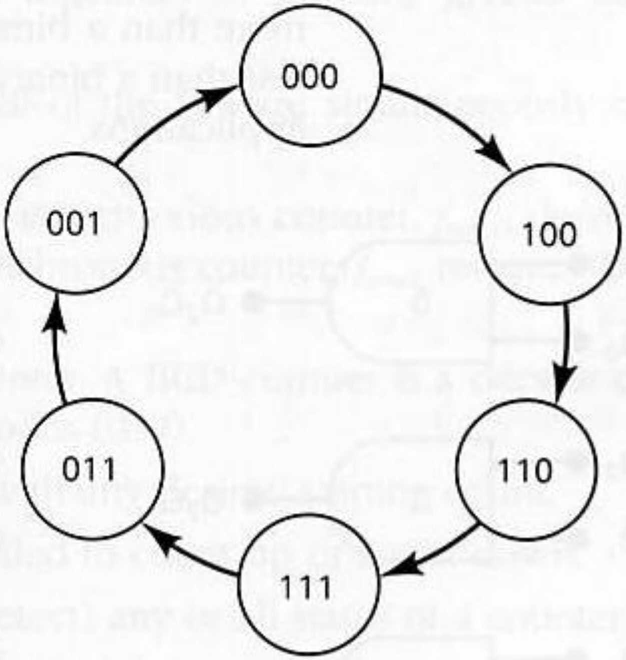


Johnson Counter



Q_2	Q_1	Q_0	CLOCK pulse
0	0	0	0
1	0	0	1
1	1	0	2
1	1	1	3
0	1	1	4
0	0	1	5
0	0	0	6
1	0	0	7
1	1	0	8
.	.	.	.
.	.	.	.
.	.	.	.

(c)

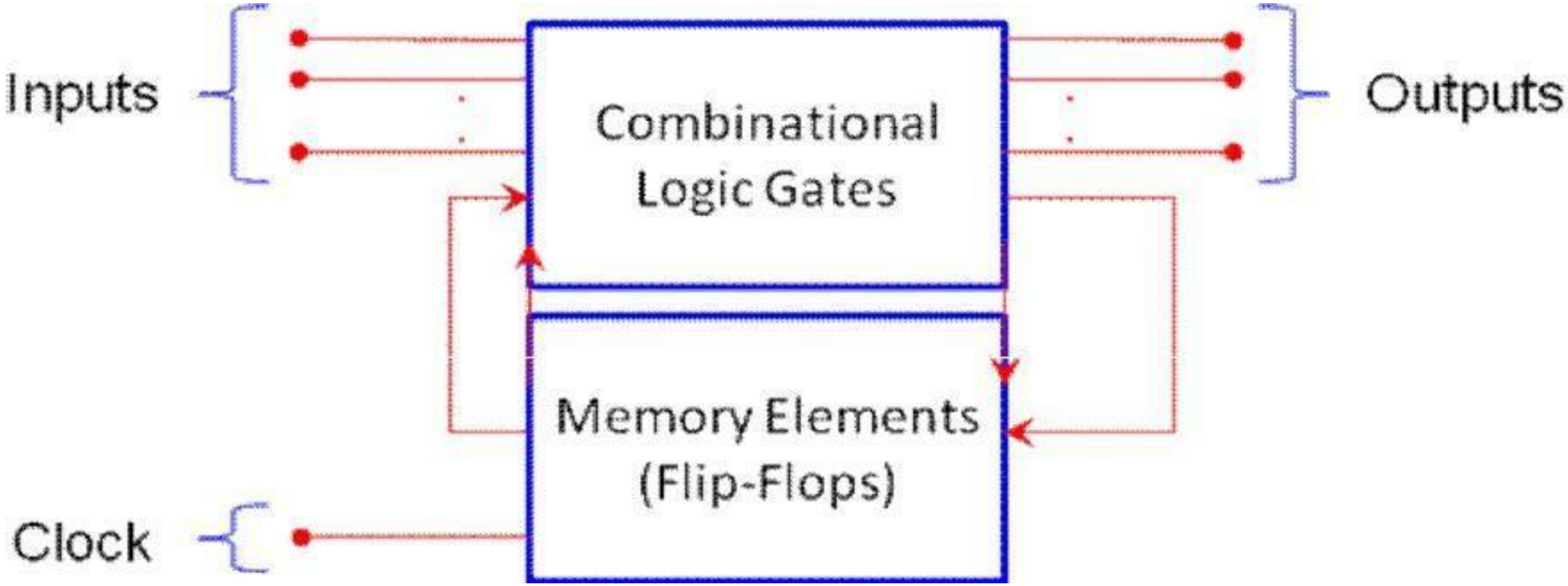


(d)

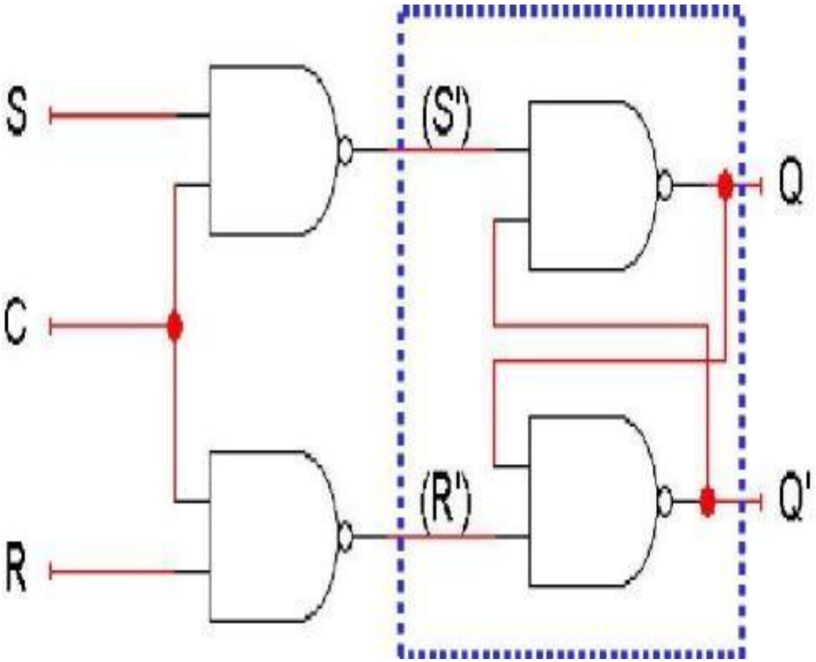
Sequential Circuits

- Latches and flip-flops (FFs) are the basic building blocks of sequential circuits.
- latch: bistable memory device with level sensitive triggering (no clock), watches all of its inputs continuously and changes its outputs at any time, independent of a clocking signal.
- flip-flop: bistable memory device with edge-triggering (with clock), samples its inputs, and changes its output only at times determined by a clocking signal.

Sequential Circuits



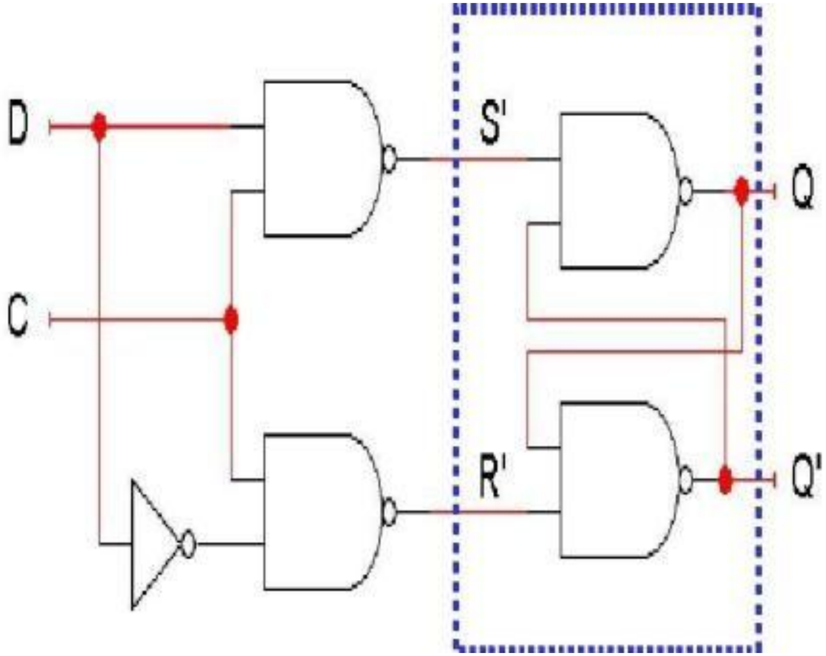
SR Latch



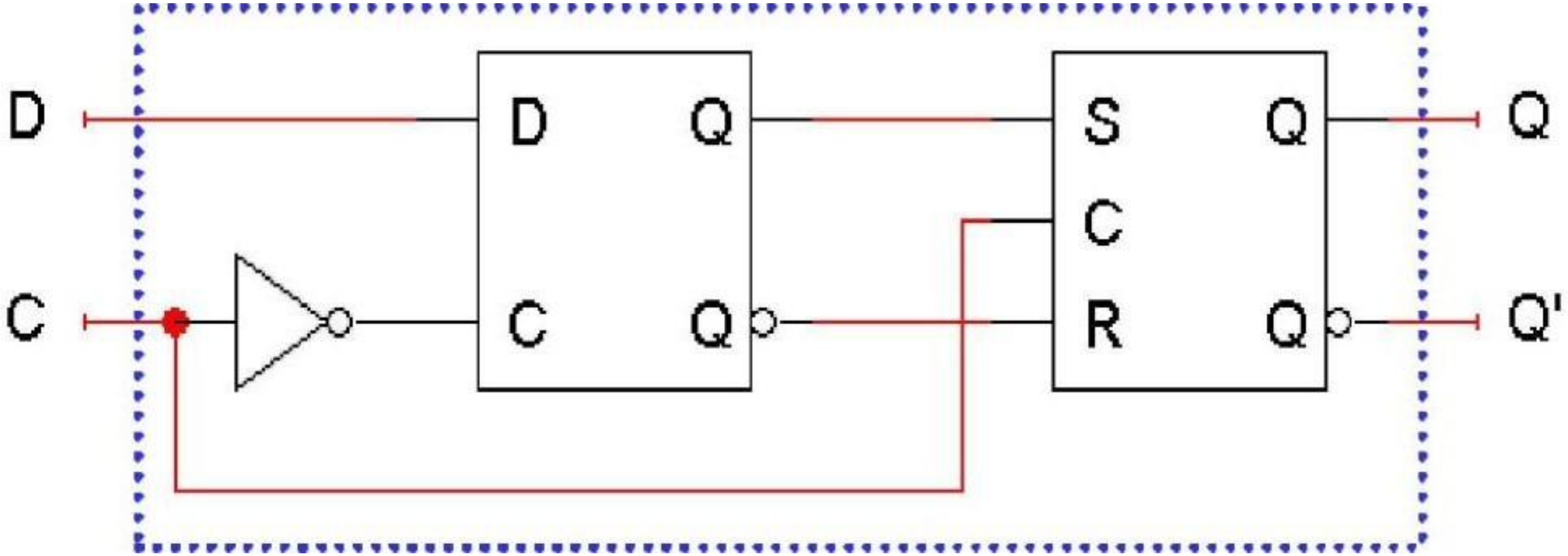
-

-

D latch



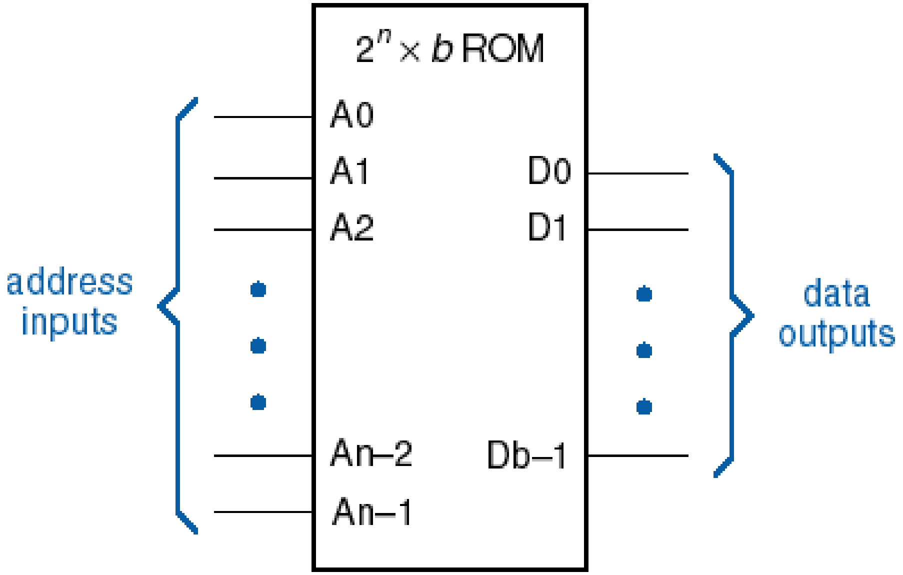
D flip-flop



Flip-Flop Vs. Latch

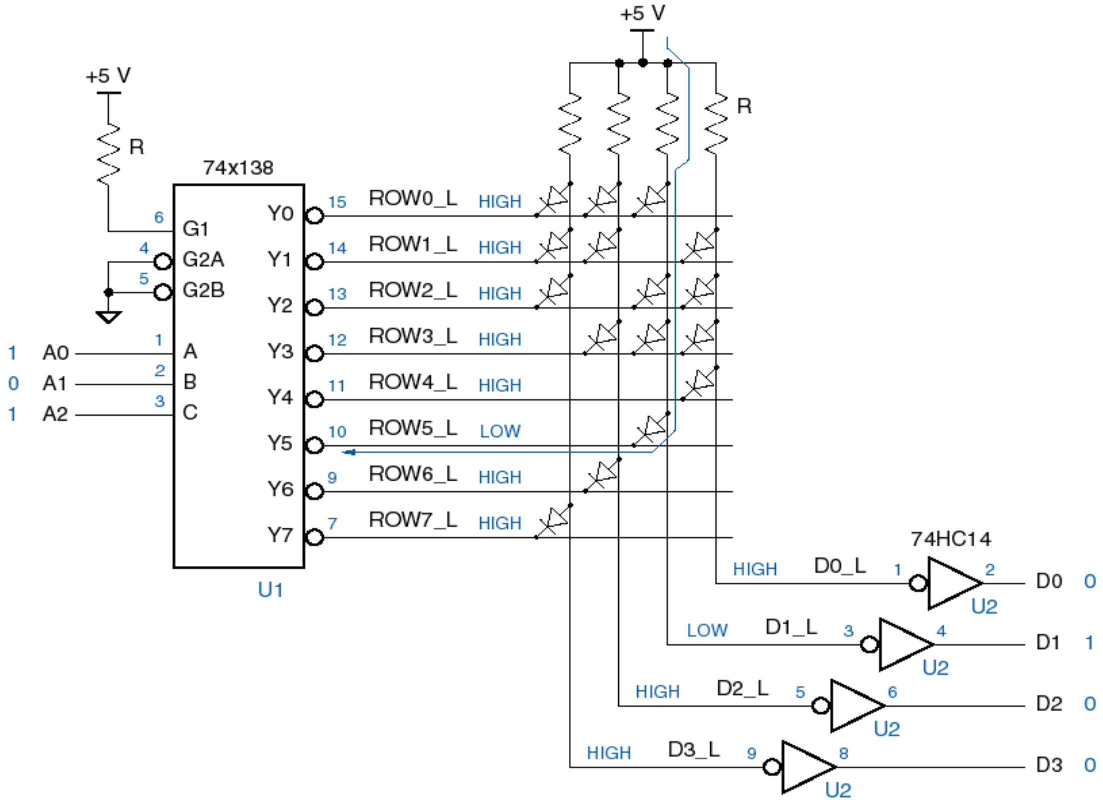
- The primary difference between a D flip-flop and D latch is the EN/CLOCK input.
- The flip-flop’s CLOCK input is edge sensitive, meaning the flip-flop’s output changes on the edge (rising or falling) of the CLOCK input.
- The latch’s EN input is level sensitive, meaning the latch’s output changes on the level (high or low) of the EN input

Read-Only Memory (ROM)

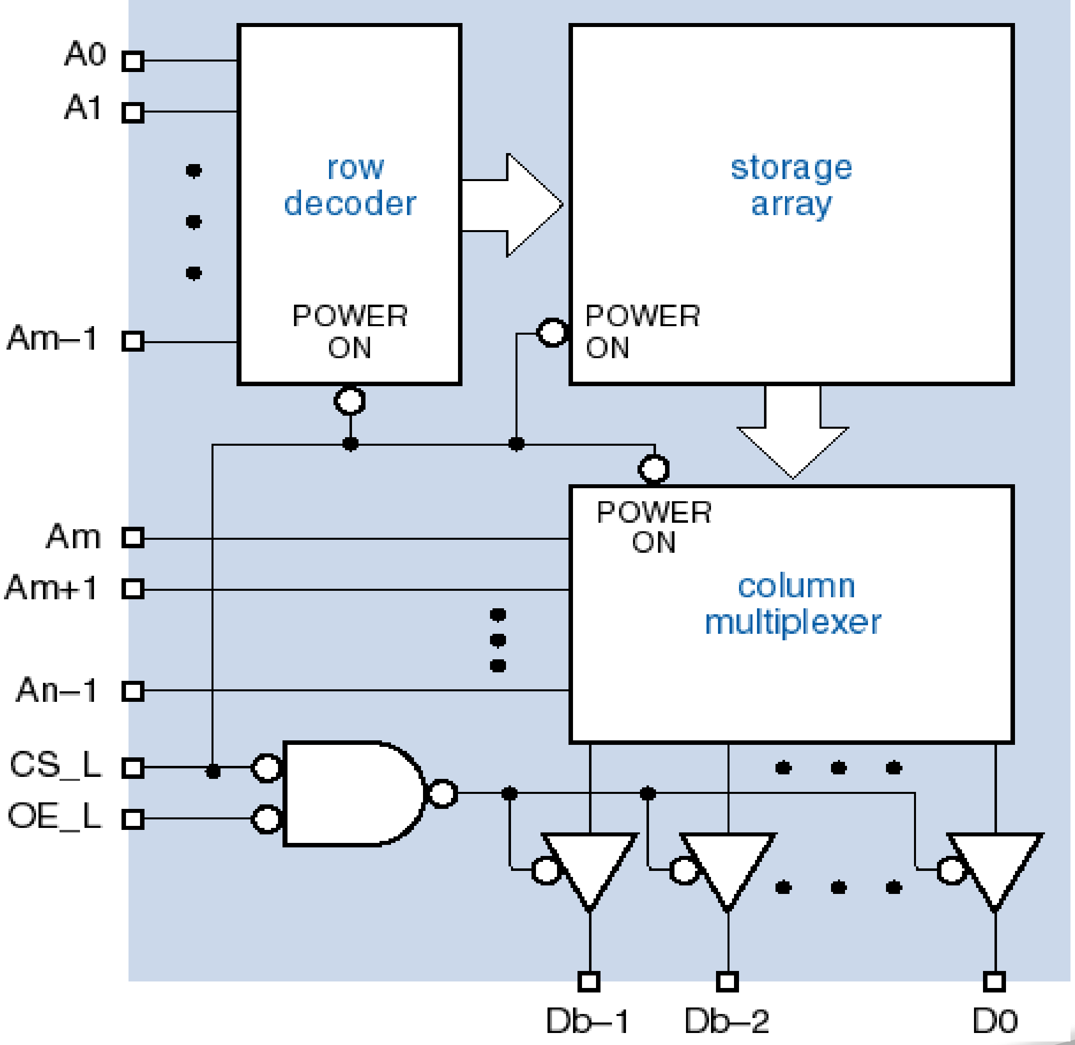


- ➢The inputs are called Address inputs and are traditionally named A0, A1, …An-1
- ➢The outputs are called Data outputs and are typically named D0, D1, …Db-1

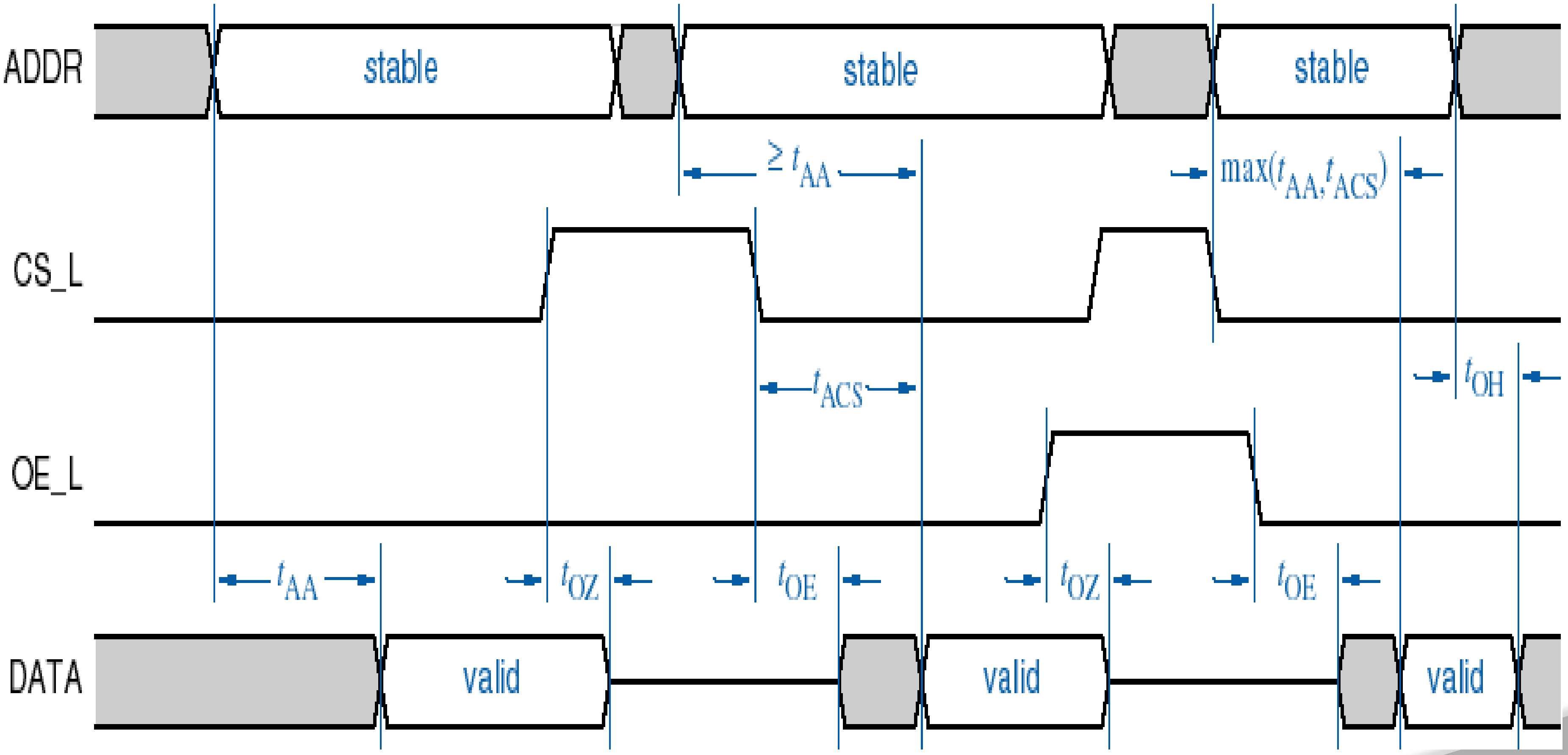
Logic Diagram of simple 8x4 diode ROM



Internal ROM structure, showing use of control inputs

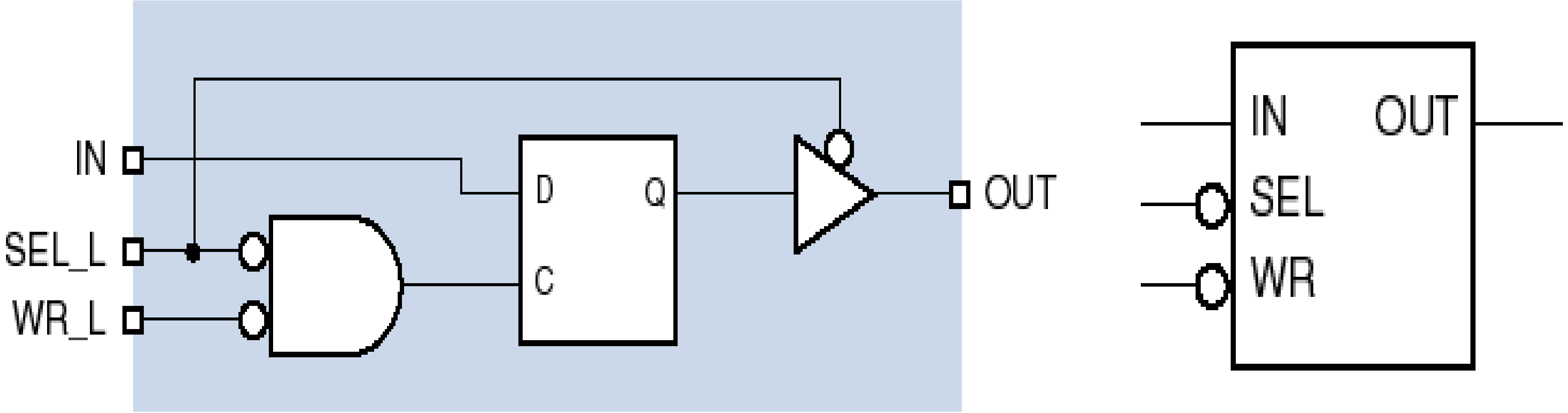


ROM timing


ADDR
stable
stable
stable
≥ t_{AA}
max(t_{AA}, t_{ACS})
CS_L
t_{ACS}
t_{OH}
OE_L
t_{AA}
t_{OZ}
t_{OE}
t_{OZ}
t_{OE}
DATA
valid
valid
valid

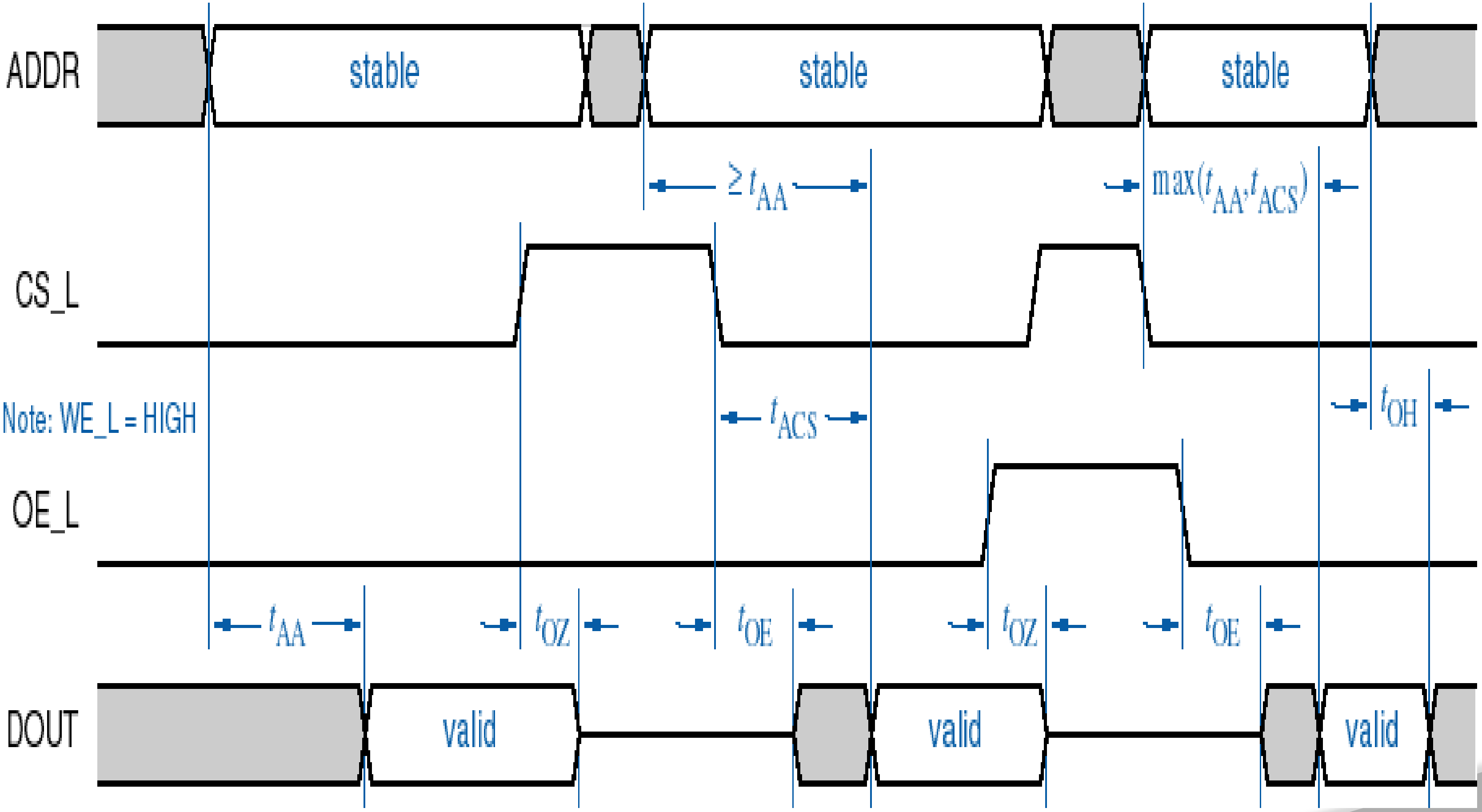
Static-RAM Internal Structure

- Each SRAM cell has the same functional behavior in the SRAM circuit.
- The storage device in each cell is a D-latch.

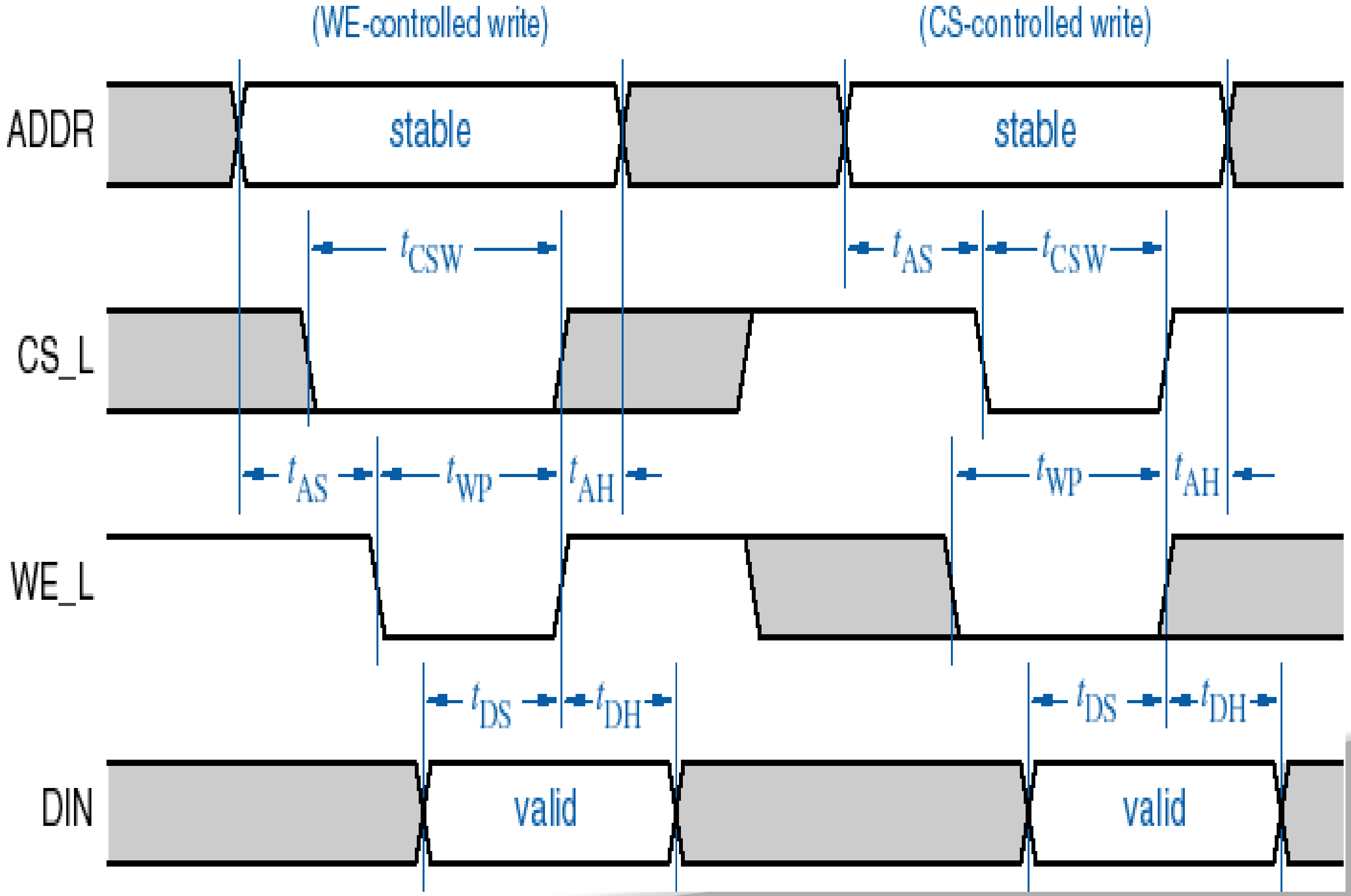


Static RAM cell

SRAM read timing



SRAM write timing



Thank you