- NAME: S VIDYA
- SUBJECT: DLD (20A04303T)
- REGULATION: R20

UNIT-1
LIST OF CONTENTS

- **BINARY NUMBER SYSTEM**
  - BINARY NUMBERS
  - OCTAL NUMBERS
  - HEXA DECIMAL NUMBERS
  - NUMBER BASE CONVERSIONS
  - COMPLIMENTS
  - SIGNED BINARY NUMBERS
  - BINARY CODES

LOGIC SIMPLIFICATION

- Binary logic and Gates
- Boolean Algebra
  - Basic Properties
  - Algebraic Manipulation
- Standard and Canonical Forms
  - Minterms and Maxterms (Canonical forms)
  - SOP and POS (Standard forms)

THE DECIMAL NUMBER SYSTEM

- The decimal system is the base-10 system that we use every day.
- A number, say 6357, represented in the base-10 system consists of multiple ordered digits. (In other words, digits are normally combined together in groups to create larger numbers.)
- A digit is a single place that can hold numerical values between 0 and 9 (10 different values).

Let us start from an arbitrary decimal number

- For example, 6,357 has four digits.

- It is understood that in the number 6,357,
  - the 7 is filling the "1s place,"
  - while the 5 is filling the 10s place,
  - the 3 is filling the 100s place
  - and the 6 is filling the 1,000s place.

- So you could express 6,357 this way if you want to be explicit:

$$(6 * 1000) + (3 * 100) + (5 * 10) + (7 * 1)$$
$$= 6000 + 300 + 50 + 7$$
$$= 6357$$

- What you can see from this expression is that each digit is a **placeholder** for the power of the index of that placeholder of base 10, starting from the least significant digit with 10 raised to the power of zero (i.e. counting from the rightmost digit).
- But why do we human beings use 10 based number system?
- The most commonly accepted explanation is that our **base-10** number system was adopted by our ancestors most likely because we have 10 fingers.
- Interestingly enough, maybe that is why digit in English also means a finger or toe.

BINARY NUMBER

- Computers happen to operate using the base-2 number system, also known as the **binary number system,** just like the base-10 number system is known as the decimal number system to human beings
- Modern computers use binary number system, in which there are only zeros and ones. (Only two symbols)
- A "bit" to binary is similar a "digit" to a decimal information. (Again, the easiest way to understand bits is to compare them to something you know: **digits**.)
- A **bit** has a single binary value, either 0 or 1.

Binary vs. Decimal

- Binary is a base two **system** which works just like our **decimal system**.
- Considering the **decimal** number **system**, it has a set of values which range from 0 to 9.
- The binary number system is base 2 and therefore requires only two digits, 0 and 1.

Bits

- The binary number system uses **binary digits (bits)** in place of decimal digits.
- A binary number is composed of only 0s and 1s, like this: 1011.

- How do you figure out what the value of the binary number 1011 is in decimal world?
- 0 = 0
- 1 = 1
- 2 = 10
- 3 = 11
- 4 = 100
- 5 = 101
- 6 = 110
- 7 = 111
- 8 = 1000
- 9 = 1001
- 10 = 1010
- 11 = 1011
- 12 = 1100

DECIMAL TO BINARY

- Keep dividing by 2
- Ex 2 : 237_{10}

237 / 2 = 118 Remainder 1

118 / 2 = 59 Remainder 0

59 / 2 = 29 Remainder 1

29 / 2 = 14 Remainder 1

14 / 2 = 7 Remainder 0

7 / 2 = 3 Remainder 1

3 / 2 = 1 Remainder 1

1 / 2 = 0 Remainder 1

1 1 1 0 1 1 0 1

Binary arithmetic operation

- Look at adder in binary and decimal

  3
+ 3
= 6

$\boxed{6}$

  11
+ 11
= (carry) which is 6 in decimal.

$\boxed{110}$

The Hexadecimal System

- Although not a problem internally, long binary number seems a problem to display in some situations. A common practice to solve this problem is to use hexadecimal to represent Binary numbers more compactly externally.
- The hexadecimal system is base 16. Therefore, it requires 16 different symbols. The values 0 through 9 are used, along with the letters A through F, which represent the decimal values 10 through 15.

    0..9, A,   B,   C,   D, E, F
    0..9, 10, 11, 12,13, 14, 15

Hexadecimal <->binary

binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

THE OCTAL SYSTEM

- The Octal system is **base 8**. Therefore it requires 8 digits. The values 0 through 7 are used.

- Octal to hexadecimal conversion, or visa versa, is most easily performed by first converting to binary.

BINARY TO OCTAL

- A binary number is converted to octal by grouping of 3 bits

1 1 0 1 0 1 1 0 1-------------→655

- The binary, hexadecimal (hex) and octal system share one common feature – they are all based on powers of 2.
- Each digit in the hex system is equivalent to a four-digit binary number and each digit in the octal system is equivalent to a 3-digit binary number.

MinTerms

- Consider a system of 3 input signals (variables) x, y, & z.
- A term which ANDs all input variables, either in the true or complement form, is called a minterm.
- Thus, the considered 3-input system has 8 minterms, namely:
- Each minterm equals 1 at exactly one particular input combination and is equal to 0 at all other combinations

$$\bar{x}\bar{y}\bar{z}, \bar{x}\bar{y}z, \bar{x}y\bar{z}, \bar{x}yz, x\bar{y}\bar{z}, x\bar{y}z, xy\bar{z} \;\&\; xyz$$

- Thus, for example, is a ... bination xyz = 000, where it is equal to 1.
- Accordingly, the minterm is referred to as m_0.
- In general, minterms are designat $\bar{x}\bar{y}\bar{z}_i$, where i corresponds the input combination at which this minterm is equal to 1.

$$\bar{x}\bar{y}\bar{z}$$

MinTerms

- For the 3-input system under consideration, the number of possible input combinations is 2^3, or 8. This means that the system has a total of 8

- $m_0 = \bar{x}\,\bar{y}\,\bar{z} = 1$ IFF $xyz = \mathbf{000}$, *otherwise it equals 0*
- $m_1 = \bar{x}\,\bar{y}z = 1$ IFF $xyz = \mathbf{001}$, *otherwise it equals 0*
- $m_2 = \bar{x}y\bar{z} = 1$ IFF $xyz = \mathbf{010}$, *otherwise it equals 0*
- $m_3 = \bar{x}yz = 1$ IFF $xyz = \mathbf{011}$, *otherwise it equals 0*
- $m_4 = x\bar{y}\,\bar{z} = 1$ IFF $xyz = \mathbf{100}$, *otherwise it equals 0*
- $m_5 = x\bar{y}z = 1$ IFF $xyz = \mathbf{101}$, *otherwise it equals 0*
- $m_6 = xy\bar{z} = 1$ IFF $xyz = \mathbf{110}$, *otherwise it equals 0*
- $m_7 = xyz = 1$ IFF $xyz = \mathbf{111}$, *otherwise it equals 0*

MinTerms

- In general, for n-input variables, the **number of minterms** = the total number of possible input combinations = 2^n.
- A minterm = 0 at all input combinations except one where the minterm = 1.
- Example: What is the number of minterms for a function with 5 input variables?
  - Number of minterms = $2^5 = 32$ minterms.

MaxTerms

- Consider a circuit of 3 input signals (variables) x, y, & z.
- A term which ORs all input variables, either in the true or complement form, is called a Maxterm.
- With 3-input variables, the system under consideration has a total of 8 Maxterms, namely:

$$(x+y+z),(x+y+\bar{z}),(x+\bar{y}+z),(x+\bar{y}+\bar{z}),(\bar{x}+y+z),(\bar{x}+y+\bar{z}),(\bar{x}+\bar{y}+z)\&(\bar{x}+\bar{y}+\bar{z})$$

- Ea … t combinations and is equal to 1 at all other combinations.
  - For example, (x + y + z) equals 1 at all input combinations except for the combination xyz = 000, where it is equal to 0.
  - Accordingly, the Maxterm (x + y + z) is referred to as M_0.

MaxTerms

- In general, Maxterms are designated M_i, where i corresponds to the input combination at which this Maxterm is equal to 0.
- For the 3 -input system, the number of possible ... ns that the ... ws:

- $M_0 = (x + y + z) = 0$ IFF $xyz = 000$, *otherwise it equals 1*
- $M_1 = (x + y + \overline{z}) = 0$ IFF $xyz = 001$, *otherwise it equals 1*
- $M_2 = (x + \overline{y} + z) = 0$ IFF $xyz = 010$, *otherwise it equals 1*
- $M_3 = (x + \overline{y} + \overline{z}) = 0$ IFF $xyz = 011$, *otherwise it equals 1*
- $M_4 = (\overline{x} + y + z) = 0$ IFF $xyz = 100$, *otherwise it equals 1*
- $M_5 = (\overline{x} + y + \overline{z}) = 0$ IFF $xyz = 101$, *otherwise it equals 1*
- $M_6 = (\overline{x} + \overline{y} + z) = 0$ IFF $xyz = 110$, *otherwise it equals 1*
- $M_7 = (\overline{x} + \overline{y} + \overline{z}) = 0$ IFF $xyz = 111$, *otherwise it equals 1*

MaxTerms

- For n-input variables, the number of Maxterms = the total number of possible input combinations = 2^n.
- A Maxterm = 1 at all input combinations except one where the Maxterm = 0.
- Using De-Morgan's theorem, or truth tables, it can be easily shown that:

$$M_i = \overline{m_i} \qquad \forall i = 0, 1, 2, \ldots\ldots, (2^n - 1)$$

Expressing Functions as a Sum of Minterms

- Consider the function F defined by the shown truth table:
- Now let's rewrite the table, with few added columns.
  - A column i indicating the input combination
  - Four columns of minterms m_2, m_4, m_5 and m_7
  - One last column OR-ing the above minterms ($m_2+m_4+m_5+m_7$)
- From this table, we can clearly see that $F = m_2+m_4+m_5+m_7$

i	$x\ y\ z$	F	m_2	m_4	m_5	m_7	$m_2+m_4+m_5+m_7$
0	0 0 0	0	0	0	0	0	0
1	0 0 1	0	0	0	0	0	0
2	0 1 0	1	1	0	0	0	1
3	0 1 1	0	0	0	0	0	0
4	1 0 0	1	0	1	0	0	1
5	1 0 1	1	0	0	1	0	1
6	1 1 0	0	0	0	0	0	0
7	1 1 1	1	0	0	0	1	1

Expressing Functions as a Sum of Minterms

- In general, Any function can be expressed by **OR-ing** all minterms (m_i) corresponding to input combinations (i) at which the function has a value of 1.
- The resulting expression is commonly referred to as the **SUM of minterms** and is typically expressed as $F = \Sigma(2, 4, 5, 7)$, where **Σ** indicates **OR-ing** of the indicated minterms. Thus, $F = \Sigma(2, 4, 5, 7) = (m2 + m4 + m5 + m7)$

Expressing Functions as a Sum of Minterms

- Consider the example with F and F\`.
- The truth table of F\` shows that F\` equals 1 at i = 0, 1, 3 and 6, then,
  - $F` = m_0 + m_1 + m_3 + m_6$,
  - $F` = \Sigma(0, 1, 3, 6)$,
  - $F = \Sigma(2, 4, 5, 7)$
- The sum of minterms expression of F\` contains all minterms that do not appear in the sum of minterms expression of F.

i	*x*	*y*	*z*	*F*	F\`
0	0	0	0	0	1
1	0	0	1	0	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	1	0
6	1	1	0	0	1
7	1	1	1	1	0

Expressing Functions as a Product of Sums

- Using De-Morgan theorem on equation:

$$\overline{F} = \overline{(m_2 + m_4 + m_5 + m_7)} = \overline{m_2}.\overline{m_4}.\overline{m_5}.\overline{m_7} = M_2.M_4.M_5.M_7$$

- This form is designated as the Product of Maxterms and is expressed using the **Π** symbol, which is used to designate product in regular algebra, but is used to designate AND-ing in Boolean algebra.
- F` = Π (2, 4, 5, 7) = M_2. M_4. M_5. M_7
- F` = Σ(0, 1, 3, 6) = Π (2, 4, 5, 7)

$$F = \overline{\overline{F}} = \overline{m_0 + m_1 + m_3 + m_6} = \overline{m_0} \cdot \overline{m_1} \cdot \overline{m_3} \cdot \overline{m_6} = M_0 \cdot M_1 \cdot M_3 \cdot M_6$$

F = Σ(2, 4, 5, 7) = ∏ (0, 1, 3, 6)

F` = ∏ (2, 4, 5, 7) = Σ (0, 1, 3, 6)

Expressing Functions as Sum of Minterms or Product of Maxterms

- Any function can be expressed both as a sum of minterms (Σ mi) and as a product of maxterms (Π Mj).
- The product of maxterms expression (Π Mj) of F contains all maxterms Mj ($\forall$ j $\neq$ i) that do not appear in the sum of minterms expression of F.
- The sum of minterms expression of F` contains all minterms that do not appear in the sum of minterms expression of F.
- This is true for all complementary functions. Thus, each of the 2^n minterms will appear either in the sum of minterms expression of F or the sum of minterms expression of F` but not both.

Expressing Functions as Sum of Minterms or Product of Maxterms

- The product of maxterms expression of F` contains all maxterms that do not appear in the product of maxterms expression of F.
- This is true for all complementary functions. Thus, each of the 2^n maxterms will appear either in the product of maxterms expression of F or the product of maxterms expression of F` but not both.

Expressing Functions as Sum of Minterms or Product of Maxterms

- Example: Given that F (a, b, c, d) = $\Sigma(0, 1, 2, 4, 5, 7)$, derive the product of maxterms expression of F and the two standard form expressions of F`.
- Since the system has 4 input variables (a, b, c & d), the number of minterms and maxterms = $2^4 = 16$
- F (a, b, c, d) = $\Sigma(0, 1, 2, 4, 5, 7)$
- F = Π (3, 6, 8, 9, 10, 11, 12, 13, 14, 15)
- F` = Σ (3, 6, 8, 9, 10, 11, 12, 13, 14, 15).
- F` = Π (0, 1, 2, 4, 5, 7)

Finding the Sum of Minterms from a Given Expression

- Let F(A,B,C)= A B + A’ C, express F as a sum of minterms
- F(A,B,C)= A B (C+C’) + A’ C (B+B’)
- = ABC + ABC’ + A’BC + A’B’C
- = Σ(1, 3, 6, 7)
- Short Cut Method:
  - A B = 1 1 - This gives us the input combinations 110 and 111 which correspond to m6 and m7
  - A’ C = 0 – 1 This gives us the input combinations 001and 011 which correspond to m1 and m3

Operations on Functions

- The AND operation on two functions corresponds to the intersection of the two sets of minterms of the functions
- The OR operation on two functions corresponds to the union of the two sets of minterms of the functions
- Example
  - Let $F(A,B,C)=\Sigma m(1, 3, 6, 7)$ and $G(A,B,C)=\Sigma m(0,1, 2, 4,6, 7)$
  - $F . G = \Sigma m(1, 6, 7)$
  - $F + G = \Sigma m(0,1, 2, 3, 4,6, 7)$
  - $F' . G = ?$
  - $F' = \Sigma m(0, 2, 4, 5)$
  - $F . G = \Sigma m(0, 2, 4)$

Canonical Forms

- The sum of minterms and the product of maxterms forms of Boolean expressions are known as **canonical** forms.
- Canonical form means that all equivalent functions will have a unique and equal representation.
- Two functions are equal if and only if they have the same sum of minterms and the same product of maxterms.
- Example:
  - Are the functions $F1 = a' b' + a c + b c'$ and $F2 = a' c' + a b + b' c$ Equal?
  - $F1 = a' b' + a c + b c' = \Sigma m(0, 1, 2, 5, 6, 7)$
  - $F2 = a' c' + a b + b' c = \Sigma m(0, 1, 2, 5, 6, 7)$
  - They are equal as they have the same set of **minterms.**

Standard Forms

- A **product term** is a term with ANDed literals. Thus, AB, A'B, A'CD are all product terms.
- A minterm is a special case of a product term where all input variables appear in the product term either in the true or complement form.
- A **sum term** is a term with ORed literals. Thus, (A+B), (A'+B), (A'+C+D) are all sum terms.
- A maxterm is a special case of a sum term where all input variables, either in the true or complement form, are ORed together.

Standard Forms

- Boolean functions can generally be expressed in the form of a **Sum of Products (SOP)** or in the form of a **Product of Sums (POS).**
- The sum of minterms form is a special case of the SOP form where all product terms are minterms.
- The product of maxterms form is a special case of the POS form where all sum terms are maxterms.
- The **SOP** and **POS** forms are **Standard forms** for representing Boolean functions.

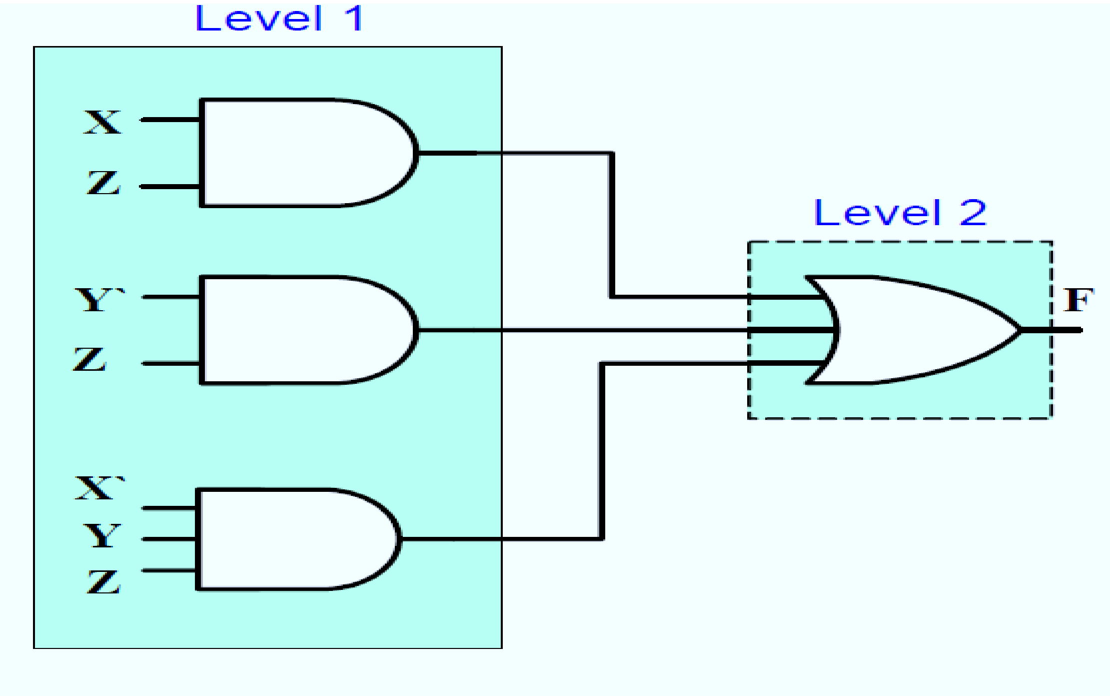
Two-Level Implementations of Standard Forms

Sum of Products Expressions (SOP):

- Any SOP expression can be implemented in 2-levels of gates.
- The first level consists of a number of **AND gates** which equals the number of product terms in the expression.
- Each AND gate implements one of the product terms in the expression.
- The second level consists of a **SINGLE OR gate** whose number of inputs equals the number of product terms in the expression.

Two-Level Implementations of Standard Forms

- Example: Implement the following SOP function
  F = XZ + Y`Z + X`YZ



Two-Level Implementation (F = XZ + Y`Z + X`YZ)
Level-1: AND-Gates : Level-2: One OR-Gate

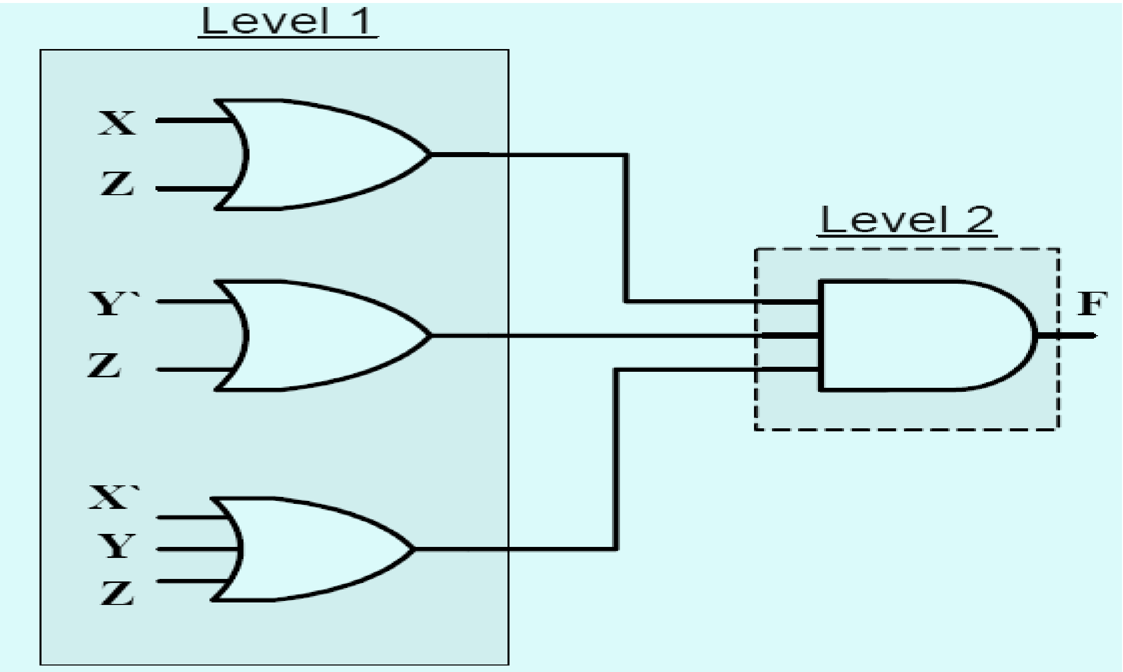
Two-Level Implementations of Standard Forms

Product of Sums Expression (POS):

- Any POS expression can be implemented in 2-levels of gates.
- The first level consists of a number of **OR gates** which equals the number of sum terms in the expression.
- Each gate implements one of the sum terms in the expression.
- The second level consists of a **SINGLE AND gate** whose number of inputs equals the number of sum terms.

Two-Level Implementations of Standard Forms

- Example: Implement the following POS function
  $F = (X+Z)(Y`+Z)(X`+Y+Z)$



Two-Level Implementation {F = (X+Z)(Y`+Z)(X`+Y+Z)}

Level-1: OR-Gates ; Level-2: One AND-Gate

Boolean Algebra

Binary Logic

- Deals with binary variables that take 2 discrete values (0 and 1), and with logic operations
- Three basic logic operations:
  - AND, OR, NOT
- Binary/logic variables are typically represented as letters: A,B,C,…,X,Y,Z

Binary Logic Function

F(vars) = *expression*

set of binary variables

- Operators (+, •, ‘)
- Variables
- Constants (0, 1)
- Groupings (parenthesis)

Example: F(a,b) = a’•b + b’

G(x,y,z) = x•(y+z’)

Boolean Algebra

- George Boole (1815-1864): “An investigation of the laws of thought”
- Terminology:
  - *Literal:* A variable or its complement
  - *Product term:* literals connected by •
  - *Sum term:* literals connected by +

Introduction

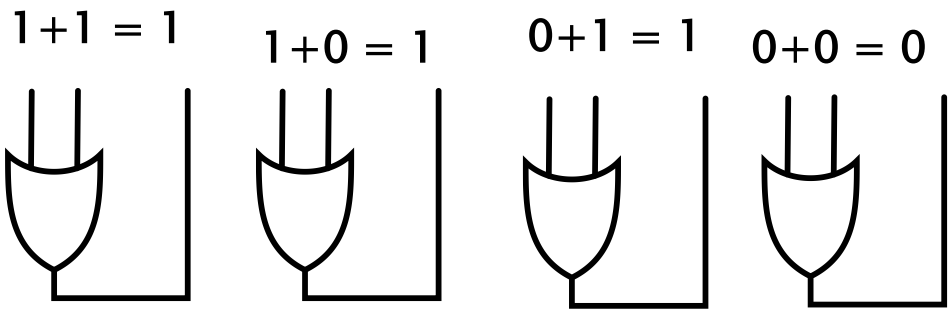
- 1854: Logical algebra was published by **George Boole** → known today as "Boolean Algebra"
- It's a convenient way and systematic way of expressing and analyzing the operation of logic circuits.
- 1938: **Claude Shannon** was the first to apply Boole's work to the analysis and design of logic circuits.

Boolean Operations & Expressions

- **Variable** – a symbol used to represent a logical quantity.
- **Complement** – the inverse of a variable and is indicated by a bar over the variable.
- **Literal** – a variable or the complement of a variable.

Boolean Addition

- Boolean addition is equivalent to the OR operation



A **sum term** is produced by an OR operation with no AND ops involved.

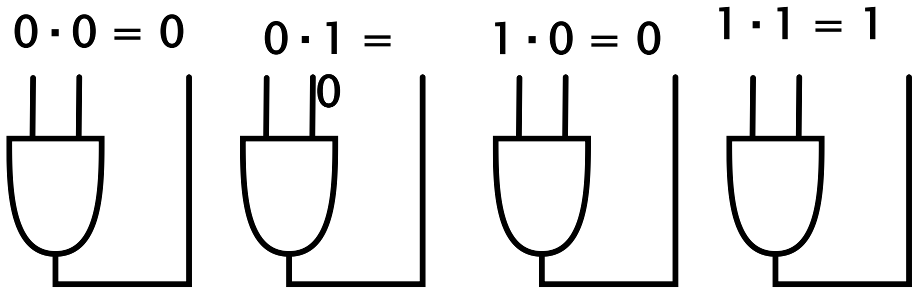
i.e. $A+B, A+\overline{B}, A+B+\overline{C}, \overline{A}+B+C+\overline{D}$

A **sum term** is equal to 1 when one or more of the literals in the term are 1.

A **sum term** is equal to 0 only if each of the literals is 0.

Boolean Multiplication

- Boolean multiplication is equivalent to the AND operation



A **product term** is produced by an AND operation with no OR ops involved.

i.e.

A **product term** is equal to 1 only if each of the literals in the term is 1.

A **product term** is equal to 0 when one or more of the literals are 0.

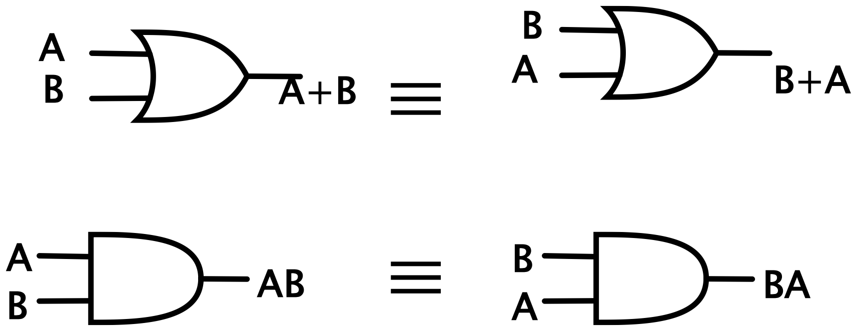
Laws & Rules of Boolean Algebra

- The basic laws of Boolean algebra:
  - The **commutative** laws
  - The **associative** laws
  - The **distributive** laws

Commutative Laws

- The **commutative law of addition** for two variables is written as: $A+B = B+A$

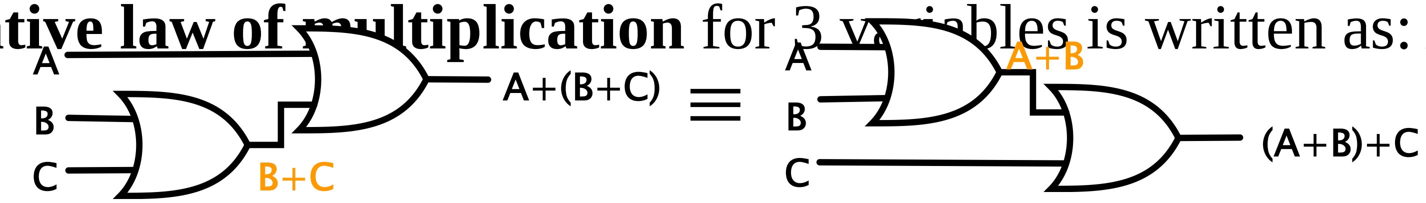
- The **commutative law of multiplication** for two variables is written as: $AB = BA$

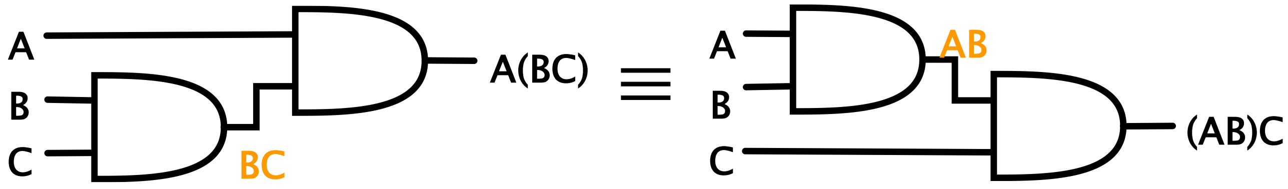


Associative Laws

- The **associative law of addition** for 3 variables is written as: $A+(B+C) = (A+B)+C$

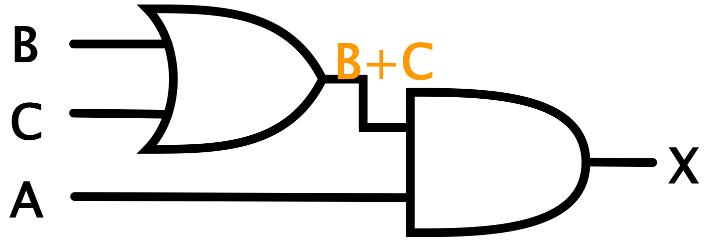
- The **associative law of multiplication** for 3 variables is written as: $A(BC) = (AB)C$



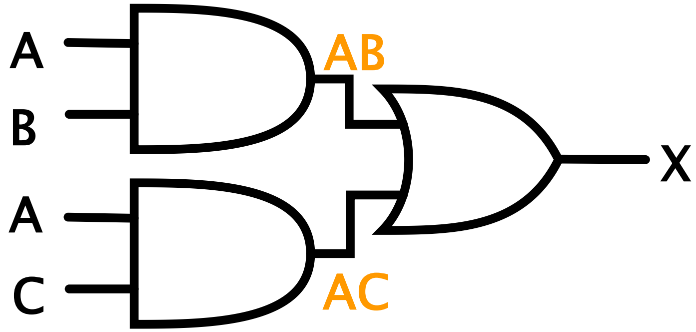


Distributive Laws

- The **distributive law** is written for 3 variables as follows:
  $A(B+C) = AB + AC$



≡



X=A(B+C)

X=AB+AC

Rules of Boolean Algebra

$1. A + 0 = A$

$2. A + 1 = 1$

$3. A \bullet 0 = 0$

$4. A \bullet 1 = A$

$5. A + A = A$

$6. A + \overline{A} = 1$

$7. A \bullet A = A$

$8. A \bullet \overline{A} = 0$

$9. \overline{\overline{A}} = A$

$10. A + AB = A$

$11. A + \overline{A}B = A + B$

$12. (A + B)(A + C) = A + BC$

A, B, and C can represent a single variable or a combination of variables.

DeMorgan's Theorems

- DeMorgan's theorems provide mathematical verification of:
  - the equivalency of the NAND and negative-OR gates
  - the equivalency of the NOR and negative-AND gates.

DeMorgan's Theorems

- The complement of two or more **ANDed** variables is equivalent to the **OR** of the complements of the individual variables.

NAND

Negative-OR

$$\overline{X \bullet Y} = \overline{X} + \overline{Y}$$

- The complement of two or more **ORed** variables is equivalent to the **AND** of the complements of the individual variables.

NOR

Negative-AND

$$\overline{X + Y} = \overline{X} \bullet \overline{Y}$$

DeMorgan’s Theorems

- Apply DeMorgan’s theorems to the expressions:

$$\overline{X \bullet Y \bullet Z}$$

$$\overline{X + Y + Z}$$

$$\overline{\overline{X} + \overline{Y} + \overline{Z}}$$

$$\overline{\overline{W} \bullet \overline{X} \bullet \overline{Y} \bullet \overline{Z}}$$

DeMorgan's Theorems

- Apply DeMorgan's theorems to the expressions:

$$\overline{(A+B+C)D}$$

$$\overline{ABC+DEF}$$

$$\overline{A\overline{B}+\overline{C}D+EF}$$

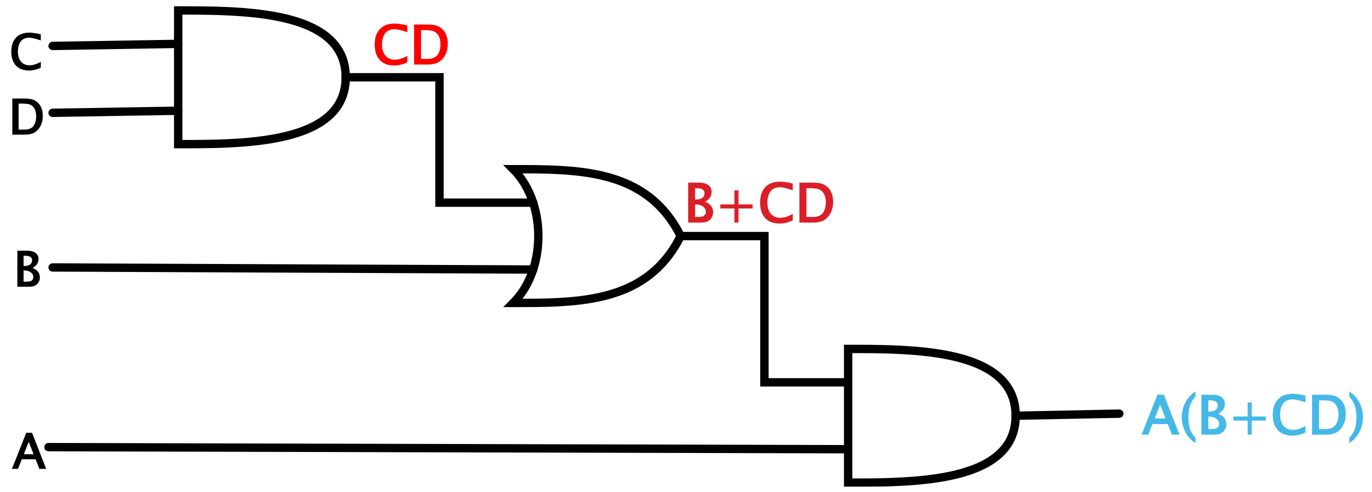
$$\overline{\overline{A+B\overline{C}}+D(\overline{E+\overline{F}})}$$

Boolean Analysis of Logic Circuits

- Boolean algebra provides a concise way to express the operation of a logic circuit formed by a combination of logic gatese
- So that the output can be determined for various combinations of input values.

Boolean Expression for a Logic Circuit

- To derive the Boolean expression for a given logic circuit, begin at the left-most inputs and work toward the final output, writing the expression for each gate.



Constructing a Truth Table for a Logic Circuit

- Once the Boolean expression for a given logic circuit has been determined, a truth table that shows the output for all possible values of the input variables can be developed.
  - Let's take the previous circuit as the example:

$$A(B+CD)$$

  - There are four variables, hence 16 (2^4) combinations of values are possible.

Constructing a Truth Table for a Logic Circuit

- Evaluating the expression
  - To evaluate the expression $A(B+CD)$, first find the values of the variables that make the expression equal to 1 (using the rules for Boolean add & mult).
  - In this case, the expression equals 1 only if A=1 and B+CD=1 because

$$A(B+CD) = 1 \cdot 1 = 1$$

Constructing a Truth Table for a Logic Circuit

- Evaluating the expression (cont')
  - Now, determine when $B+CD$ term equals 1.
  - The term $B+CD=1$ if either $B=1$ or $CD=1$ or if both B and CD equal 1 because

$$B+CD = 1+0 = 1$$
$$B+CD = 0+1 = 1$$
$$B+CD = 1+1 = 1$$

- The term $CD=1$ only if $C=1$ and $D=1$

Constructing a Truth Table for a Logic Circuit

- Evaluating the expression (cont')
  - Summary:
  - $A(B+CD)=1$
    - When $A=1$ and $B=1$ regardless of the values of C and D
    - When $A=1$ and $C=1$ and $D=1$ regardless of the value of B
  - The expression $A(B+CD)=0$ for all other value combinations of the variables.

Constructing a Truth Table for a Logic Circuit

- Putting the results in truth table format

A(B+CD)=1

When A=1 and B=1 regardless of the values of C and D

When A=1 and C=1 and D=1 regardless of the value of B

INPUTS				OUTPUT
A	B	C	D	A(B+CD)
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Boolean Algebra Properties

Let X: boolean variable, 0,1: constants

1. $X + 0 = X$ -- Zero Axiom
2. $X \bullet 1 = X$ -- Unit Axiom
3. $X + 1 = 1$ -- Unit Property
4. $X \bullet 0 = 0$ -- Zero Property

Boolean Algebra Properties

Let X: boolean variable, 0,1: constants

5. $X + X = X$ -- Idepotence
6. $X \bullet X = X$ -- Idepotence
7. $X + X' = 1$ -- Complement
8. $X \bullet X' = 0$ -- Complement
9. $(X')' = X$ -- Involution

Duality

- The dual of an expression is obtained by exchanging (• and +), and (1 and 0) in it, provided that the precedence of operations is not changed.
- Cannot exchange x with x'
- Example:
  - Find $H(x,y,z)$, the dual of $F(x,y,z) = x'yz' + x'y'z$
  - $H = (x'+y+z')(x'+y'+z)$

Duality

With respect to duality, Identities 1 – 8 have the following relationship:

1. $X + 0 = X$	2. $X \bullet 1 = X$	(dual of 1)
3. $X + 1 = 1$	4. $X \bullet 0 = 0$	(dual of 3)
5. $X + X = X$	6. $X \bullet X = X$	(dual of 5)
7. $X + X' = 1$	8. $X \bullet X' = 0$	(dual of 8)

More Boolean Algebra Properties

Let X,Y, and Z: boolean variables

10. $X + Y = Y + X$ 11. $X \bullet Y = Y \bullet X$ -- Commutative
12. $X + (Y+Z) = (X+Y) + Z$ 13. $X\bullet(Y\bullet Z) = (X\bullet Y)\bullet Z$ -- Associative
14. $X\bullet(Y+Z) = X\bullet Y + X\bullet Z$ 15. $X+(Y\bullet Z) = (X+Y) \bullet (X+Z)$ -- Distributive
16. $(X + Y)' = X' \bullet Y'$ 17. $(X \bullet Y)' = X' + Y'$

-- DeMorgan's In general,

$(X_1 + X_2 + \ldots + X_n)' = X_1'\bullet X_2'\bullet \ldots \bullet X_n'$, and

$(X_1\bullet X_2\bullet \ldots \bullet X_n)' = X_1' + X_2' + \ldots + X_n'$

Absorption Property

1. $x + x \bullet y = x$
2. $x \bullet (x+y) = x$ (dual)

- **Proof:**

$$
\begin{aligned}
x + x \bullet y &= x \bullet 1 + x \bullet y \\
&= x \bullet (1+y) \\
&= x \bullet 1 \\
&= x
\end{aligned}
$$

QED (2 true by duality, why?)

Power of Duality

1. $x + x \bullet y = x$ is true, so $(x + x \bullet y)' = x'$
2. $(x + x \bullet y)' = x' \bullet (x' + y')$
3. $x' \bullet (x' + y') = x'$
4. Let $X = x'$, $Y = y'$
5. $X \bullet (X + Y) = X$, which is the dual of $x + x \bullet y = x$.
6. The above process can be applied to any formula. So if a formula is valid, then its dual must also be valid.
7. Proving one formula also proves its dual.

Consensus Theorem

1. $xy + x'z + yz = xy + x'z$
2. $(x+y)\bullet(x'+z)\bullet(y+z) = (x+y)\bullet(x'+z)$ -- (dual)

- **Proof:**

$$
\begin{aligned}
xy + x'z + yz &= xy + x'z + (x+x')yz \\
&= xy + x'z + xyz + x'yz \\
&= (xy + xyz) + (x'z + x'zy) \\
&= xy + x'z
\end{aligned}
$$

QED (2 true by duality).

Truth Tables (revisited)

- Enumerates all possible combinations of variable values and the corresponding function value
- Truth tables for some arbitrary functions $F_1(x,y,z)$, $F_2(x,y,z)$, and $F_3(x,y,z)$ are shown to the right.

x	y	z		F_1	F_2	F_3
0	0	0		0	1	1
0	0	1		0	0	1
0	1	0		0	0	1
0	1	1		0	1	1
1	0	0		0	1	0
1	0	1		0	1	0
1	1	0		0	0	0
1	1	1		1	0	1

Truth Tables

- Truth table: a <u>unique</u> representation of a Boolean function
- If two functions have identical truth tables, the functions are equivalent (and vice-versa).
- Truth tables can be used to prove equality theorems.
- However, the size of a truth table grows <u>exponentially</u> with the number of variables involved, hence unwieldy. This motivates the use of Boolean Algebra.

Algebraic Manipulation

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

$$
\begin{aligned}
F &= x'yz + x'yz' + xz \\
&= x'y(z+z') + xz \\
&= x'y \bullet 1 + xz \\
&= x'y + xz
\end{aligned}
$$

Algebraic Manipulation

- Example: Prove

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

- **Proof:**

  $x'y'z' + x'yz' + xyz'$

  $$\begin{aligned} &= x'y'z' + x'yz' + x'yz' + xyz' \\ &= x'z'(y'+y) + yz'(x'+x) \\ &= x'z' \bullet 1 + yz' \bullet 1 \\ &= x'z' + yz' \end{aligned}$$

  QED.

Complement of a Function

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

Complementation: Example

- Find G(x,y,z), the complement of
  F(x,y,z) = xy'z' + x'yz

- G = F' = (xy'z' + x'yz)'
  = (xy'z')' • (x'yz)' **DeMorgan**
  = (x'+y+z) • (x+y'+z') **DeMorgan** again

- Note: The complement of a function can also be derived by finding the function's *dual*, and then complementing all of the literals

Truth Table notation for Minterms and Maxterms

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

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

UNIT II

- GATE LEVEL MINIMIZATION
  - KARNAUGH MAP
  - TABULAR MINIMIZATION METHOD
- COMBINATIONAL LOGIC
  - BINARY ADDER –SUBTRACTER
  - DECIMAL ADDER
  - BINARY MMULTIPLIER
  - MAGNITUDE COMPARATOR
  - DECODER, ENCODER
  - MULTIPLEXERS AND DEMULTIPLEXER

UNIT II

- GATE LEVEL MINIMIZATION
  - KARNAUGH MAP
  - TABULAR MINIMIZATION METHOD
  - POS AND SOP IMPLIMENTATION
  - NAND & NOR IMPLIMENTATION NOR
  - OTHER TWO LEVEL IMPLIMENTATION
  - XOR IMPLIMENTATION

Karnaugh Maps

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

Two-Variable Map

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

OR

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

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

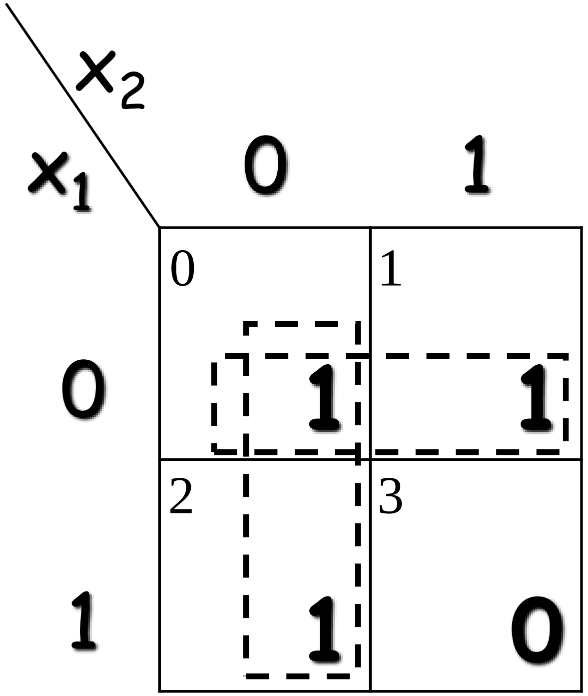
Cell 0 represents $x_1'x_2'$; Cell 1 represents $x_1'x_2$; etc. If a minterm is present in the function, then a 1 is placed in the corresponding cell.

Two-Variable Map (cont.)

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

2-Variable Map -- Example

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

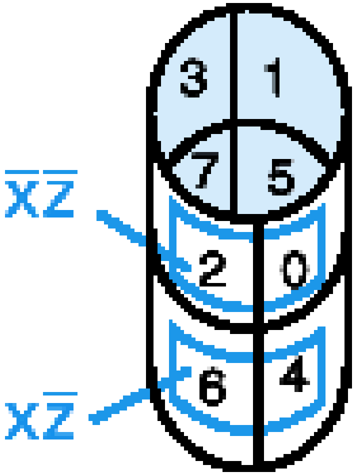


Minimization as SOP using K-map

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

Three-Variable Map

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

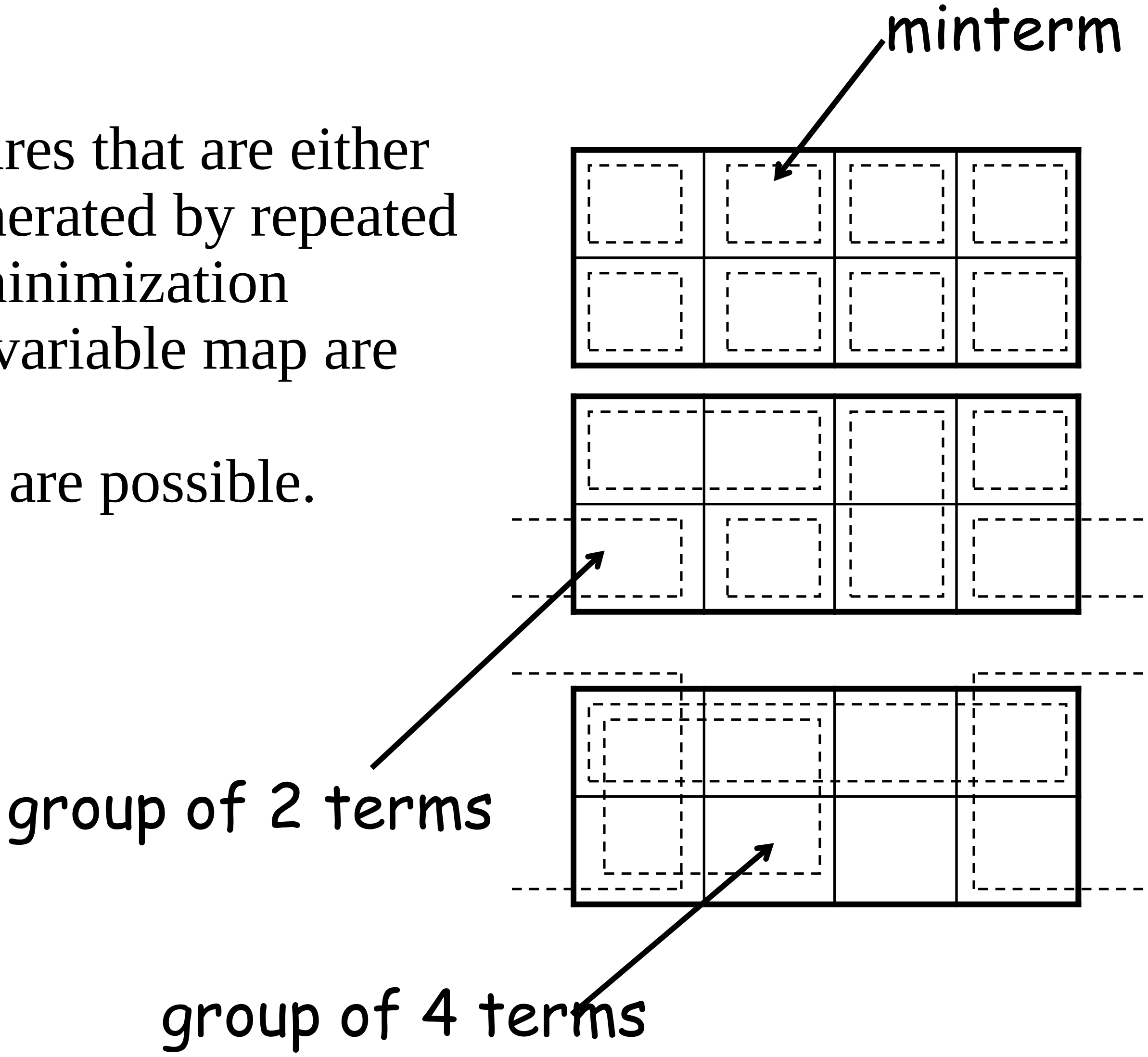


-Note: variable ordering is (x,y,z); yz specifies column, x specifies row.

-Each cell is adjacent to ***<u>three</u>*** other cells (left or right or top or bottom or edge wrap)

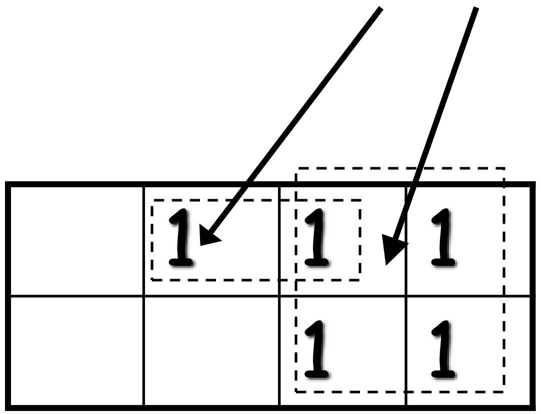
Three-Variable Map (cont.)

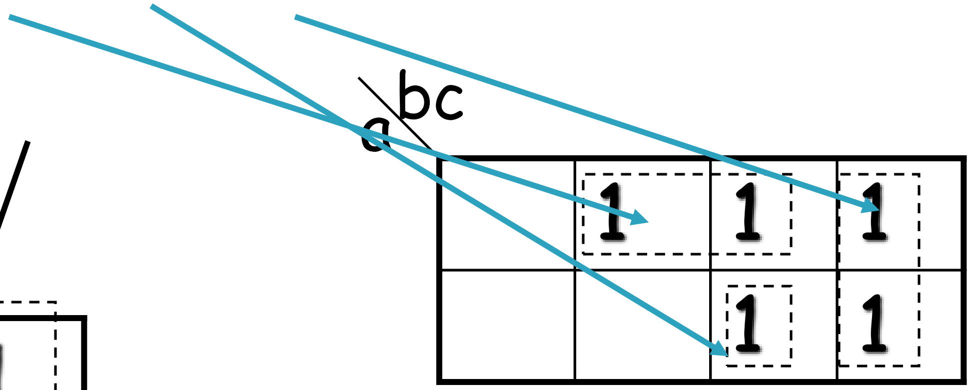
The types of structures that are either minterms or are generated by repeated application of the minimization theorem on a three variable map are shown at right.
Groups of 1, 2, 4, 8 are possible.



Simplification

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





More Examples

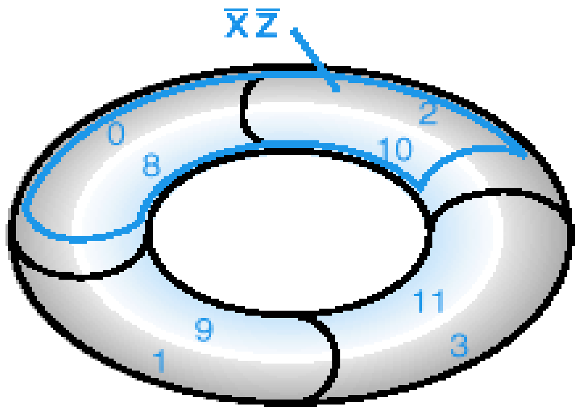
- $f_1(x, y, z) = \sum m(2,3,5,7)$
  - $f_1(x, y, z) = x'y + xz$
- $f_2(x, y, z) = \sum m(0,1,2,3,6)$
  - $f_2(x, y, z) = x'+yz'$

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

1	1	1	1
			1

Four-Variable Maps

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



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

Four-variable Map Simplification

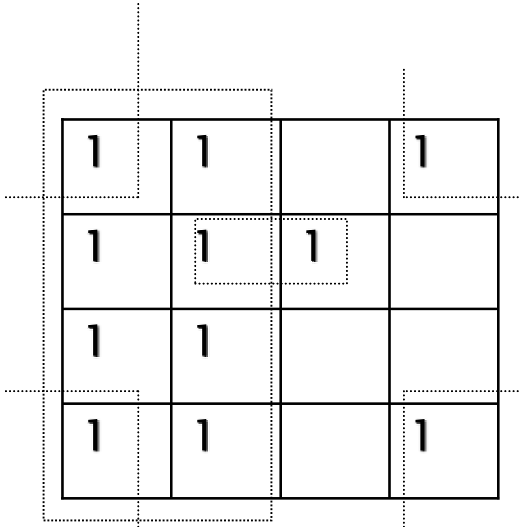
- One square represents a minterm of 4 literals.
- A rectangle of 2 adjacent squares represents a product term of 3 literals.
- A rectangle of 4 squares represents a product term of 2 literals.
- A rectangle of 8 squares represents a product term of 1 literal.
- A rectangle of 16 squares produces a function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

Example

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

ab \ cd

1	1		1
1	1	1	
1	1		
1	1		1



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

Don't Care Conditions

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

Minimization using Don’t Cares

- Treat don't cares as if they are 1s to generate PIs.
- Delete PI's that cover only don't care minterms.
- Treat the covering of remaining don't care minterms as optional in the selection process (*i.e.* they may be, but need not be, covered).

Example

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

  or
- f = a’c’d+ab’+cd’+a’bd’

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

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

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

Another Example

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

ab\cd

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

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

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

Algorithmic minimization

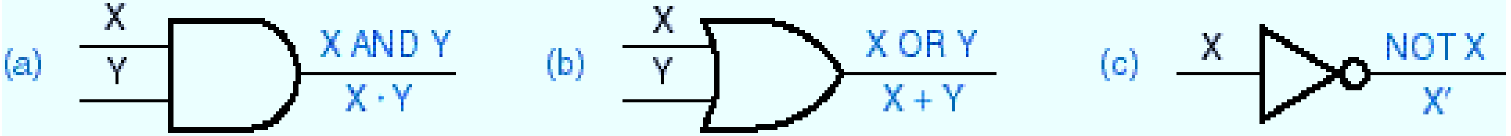
- What do we do for functions with more variables?
- You can "code up" a minimizer (Computer-Aided Design, CAD)
  - Quine-McCluskey algorithm
  - Iterated consensus
- We won't discuss these techniques here

More Logic Gates

- NAND and NOR Gates
  - NAND and NOR circuits
  - Two-level Implementations
  - Multilevel Implementations
- Exclusive-OR (XOR) Gates
  - Odd Function
  - Parity Generation and Checking

More Logic Gates

- We can construct any combinational circuit with AND, OR, and NOT gates

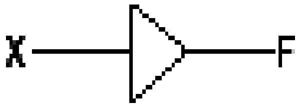


X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

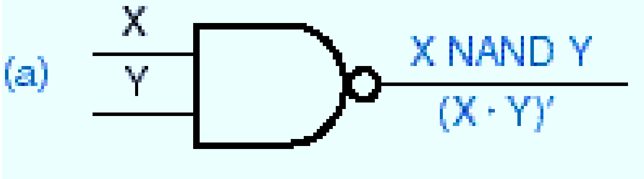
X	NOT X
0	1
1	0

BUFFER, NAND and NOR

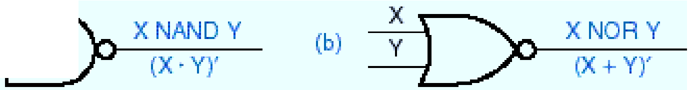


F = X

X	F
0	0
1	1



X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0



X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0

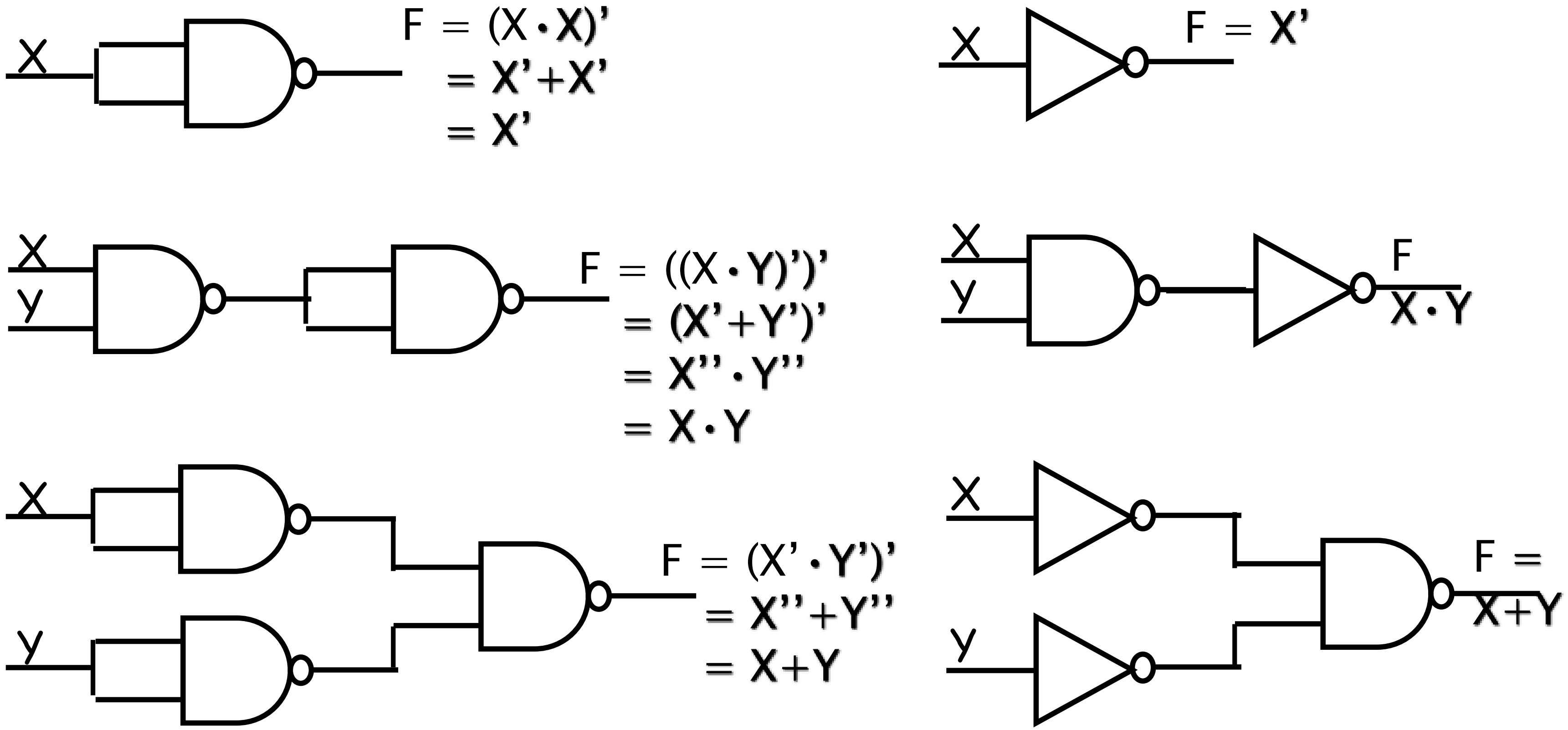
X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0

X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0

NAND Gate

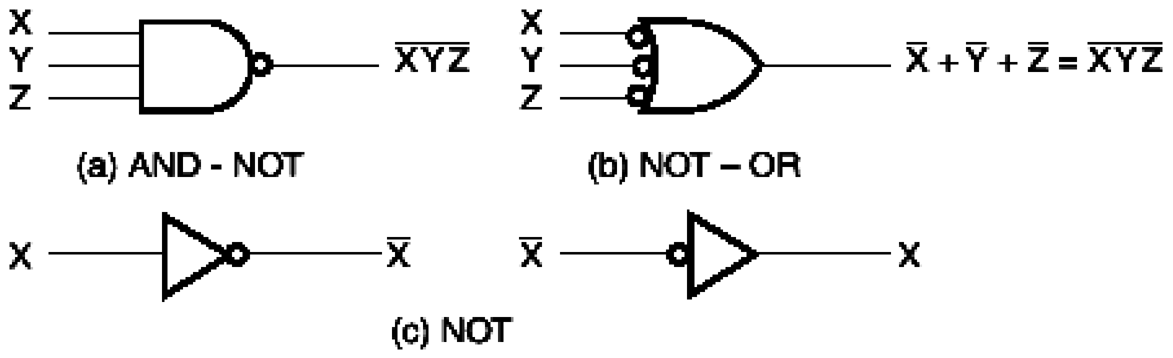
- Known as a “**Universal**” gate because ANY digital circuit can be implemented with NAND gates alone.
- To prove the above, it suffices to show that AND, OR, and NOT can be implemented using NAND gates only**.**

NAND Gate Emulation

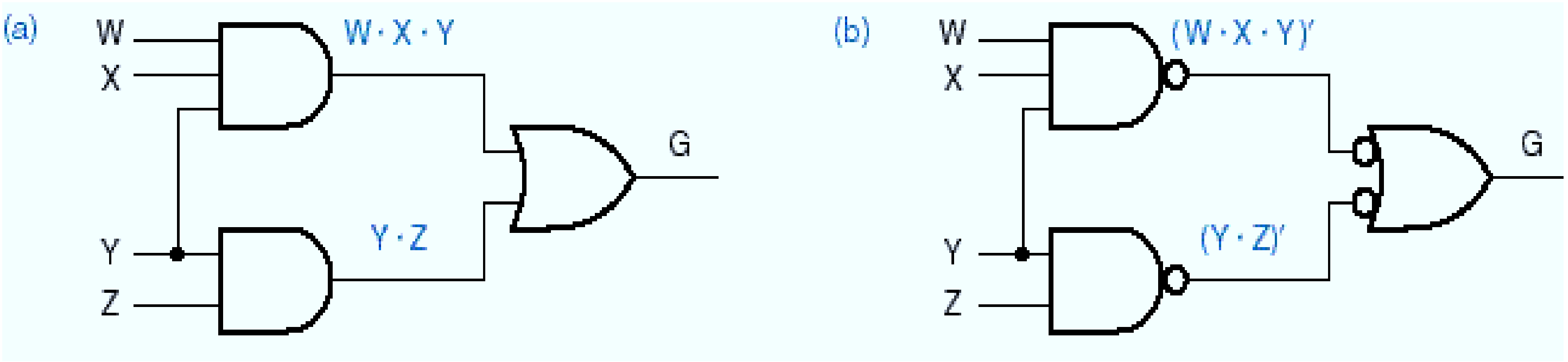


NAND Circuits

- To easily derive a NAND implementation of a boolean function:
  - Find a simplified SOP
  - SOP is an AND-OR circuit
  - Change AND-OR circuit to a NAND circuit
  - Use the alternative symbols below



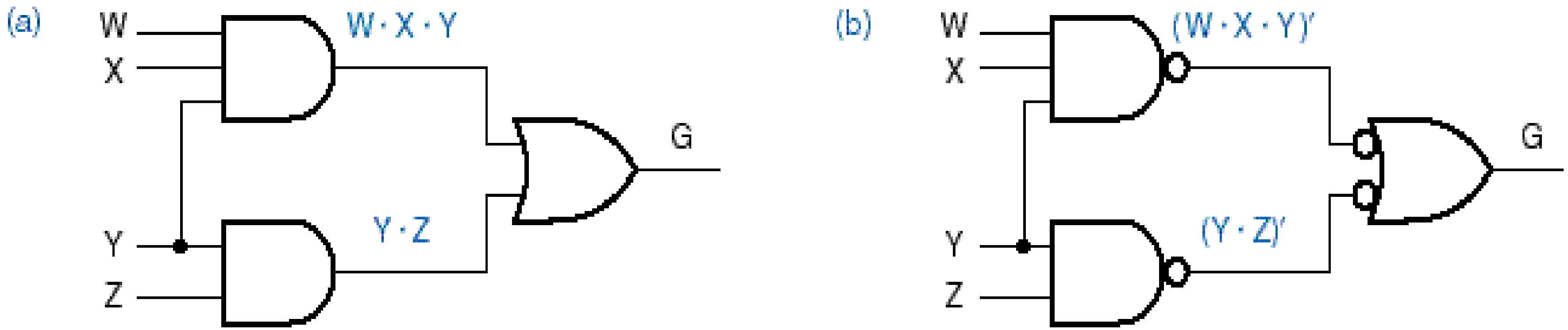
AND–OR (SOP) Emulation Using NANDs



Two–level implementations

a) Original SOP
b) Implementation with NANDs

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

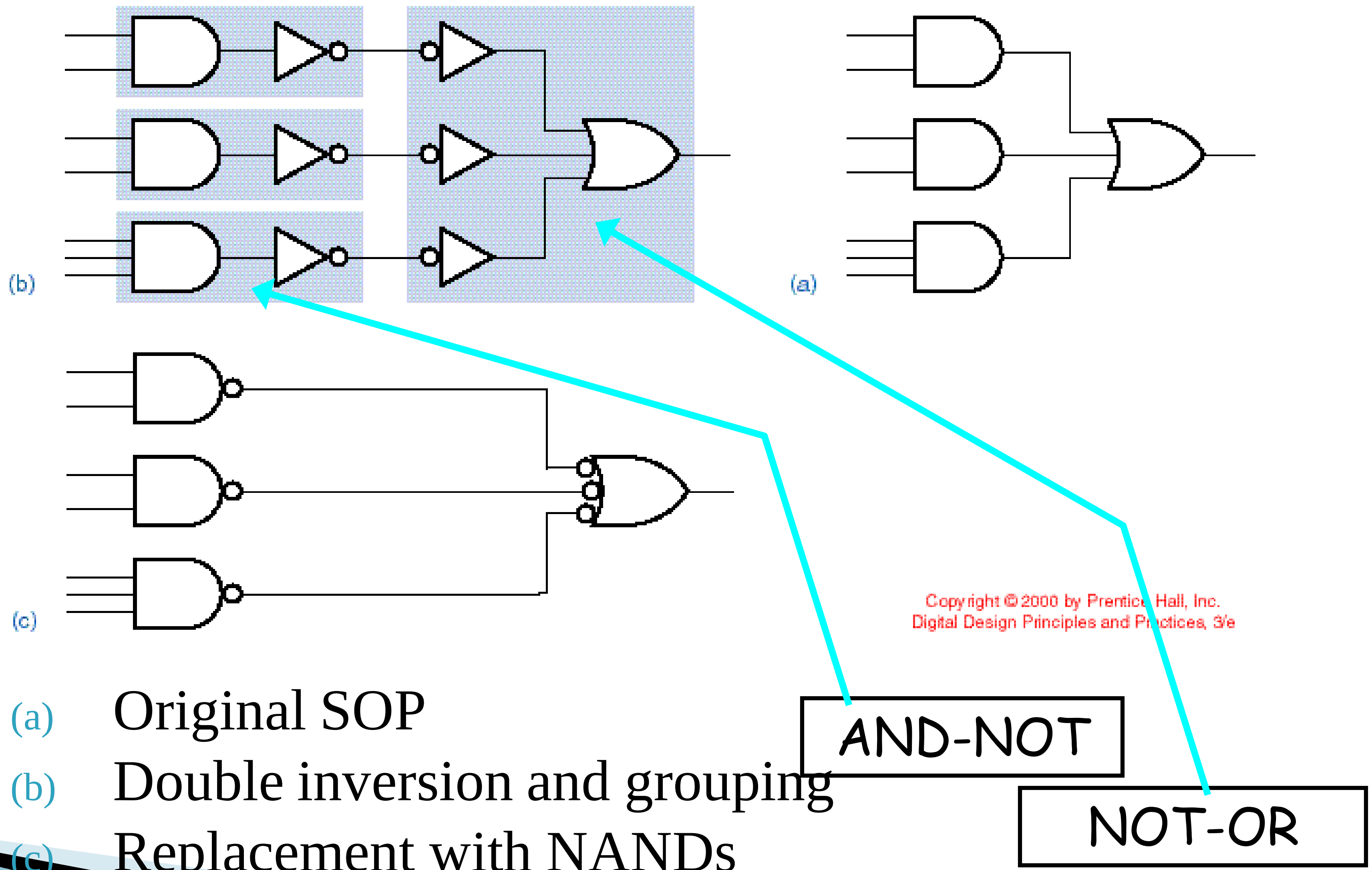


Verify:

(a) $G = WXY + YZ$

(b) $G = ((WXY)' \bullet (YZ)')'$
$= (WXY)'' + (YZ)'' = WXY + YZ$

SOP with NAND



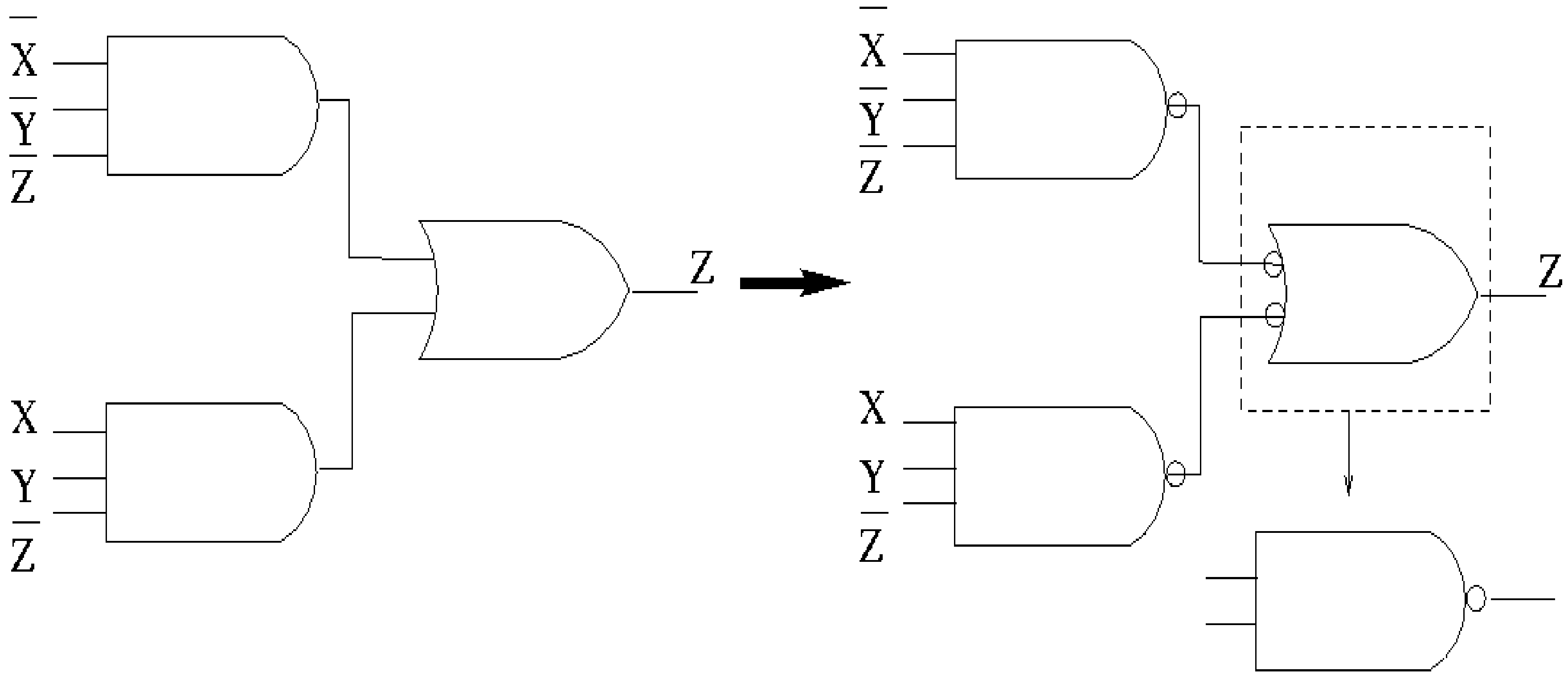
(a) Original SOP
(b) Double inversion and grouping
(c) Replacement with NANDs

Two-Level NAND Gate Implementation – Example

F (X,Y,Z) = Σm(0,6)

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

Example (cont.)



Two-level implementation with NANDs

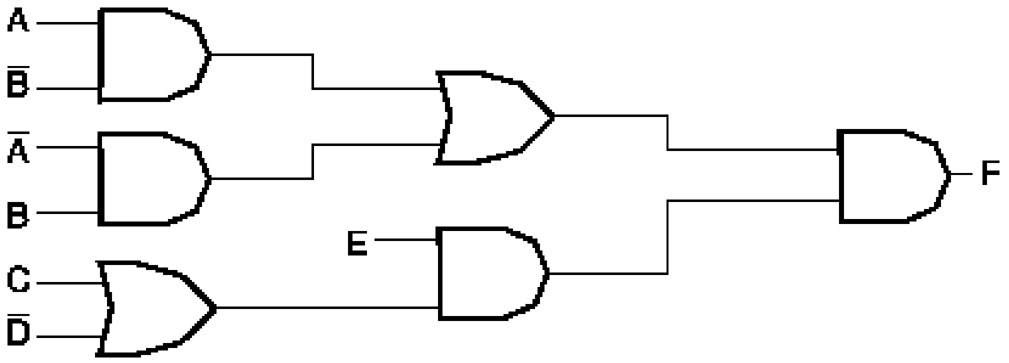
F = X'Y'Z' + XYZ'

Multilevel NAND Circuits

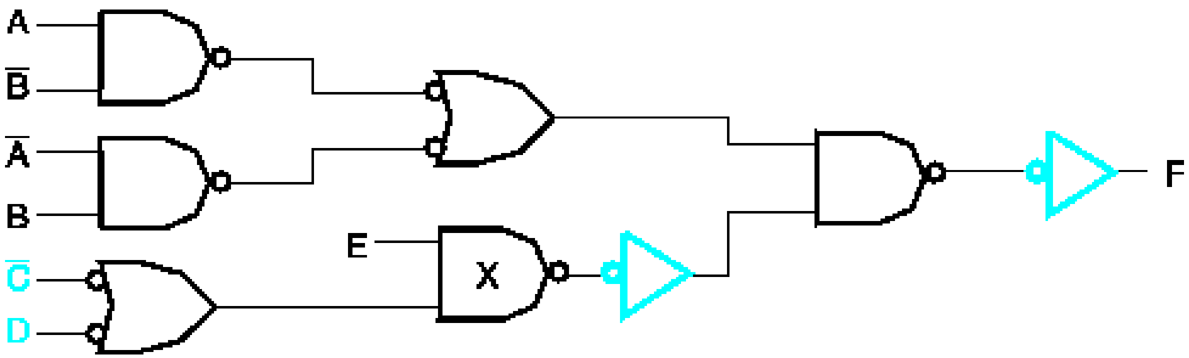
Starting from a multilevel circuit:

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

Yet Another Example!



(a) AND – OR gates



(b) NAND gates

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

NOR Gate

- Also a “**Universal**” gate because ANY digital circuit can be implemented with NOR gates alone.
- This can be similarly proven as with the NAND gate.

NOR Circuits

- To easily derive a NOR implementation of a boolean function:
  - Find a simplified POS
  - POS is an OR-AND circuit
  - Change OR-AND circuit to a NOR circuit
  - Use the alternative symbols below

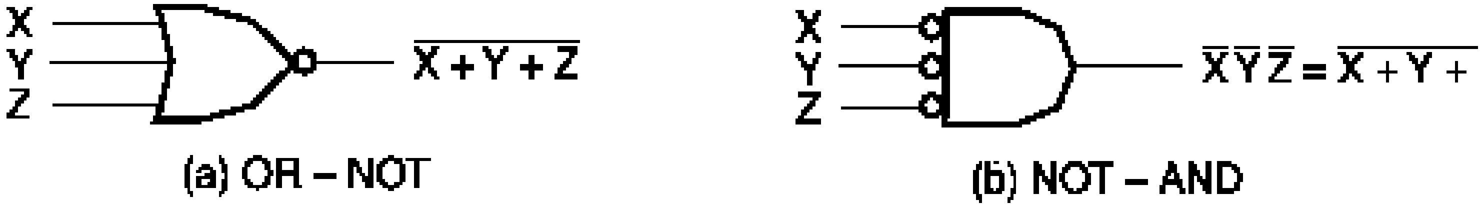


Fig. 2-34 Two Graphic Symbols for NOR Gate

Two-Level NOR Gate Implementation – Example

F(X,Y,Z) = Σm(0,6)

1. Express F' in SOP form:

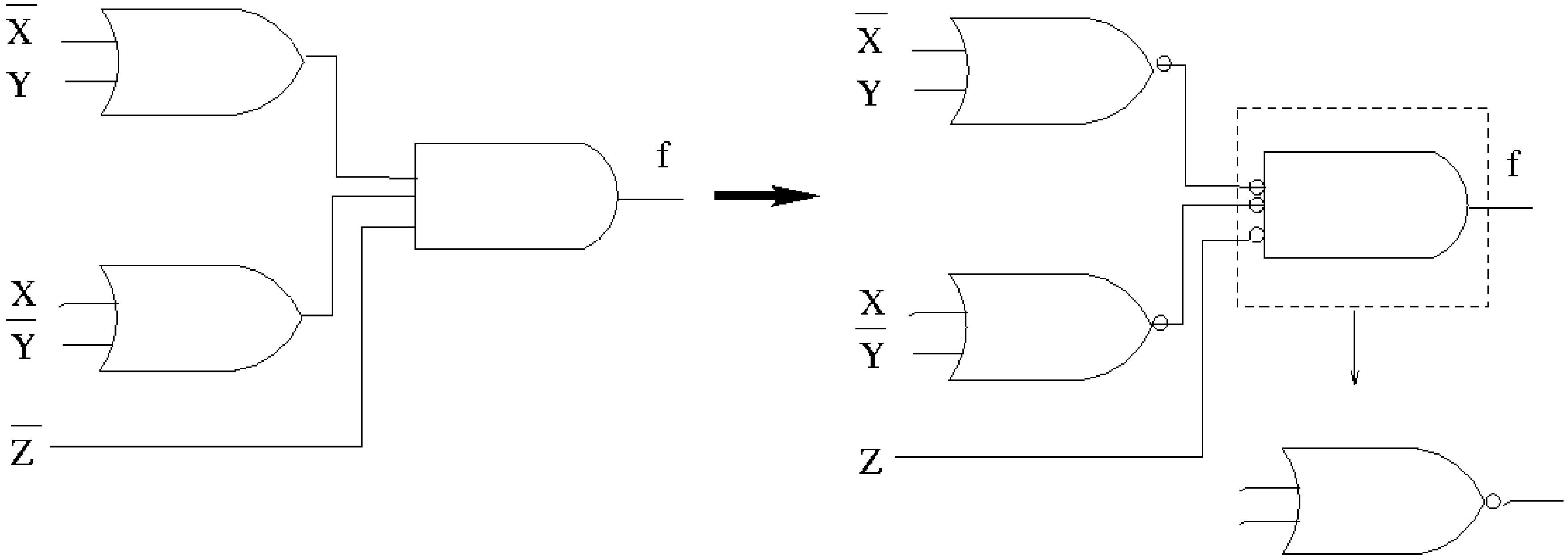
   a) F' = Σm(1,2,3,4,5,7)

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

   b) F' = XY' + X'Y + Z

2. Take the complement of F' to get F in the POS form: F = (F')' = (X'+Y)(X+Y')Z'
3. Obtain the OR-AND implementation for F.
4. Add bubbles and inverters to transform OR-AND implementation to NOR-NOR implementation.

Example (cont.)

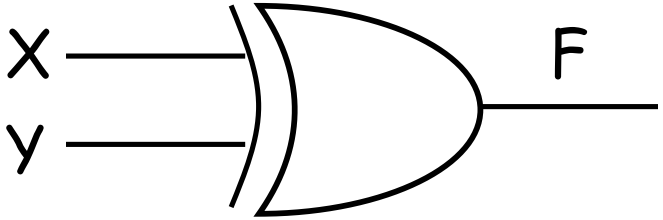


Two-level implementation with NORs

F = (F’)' = (X'+Y)(X+Y')Z'

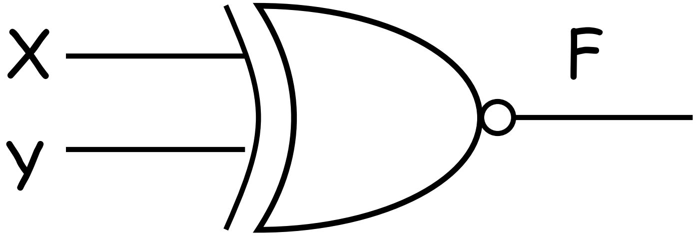
XOR and XNOR

XOR: "not-equal" gate



X	Y	$F = X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

XNOR: "equal" gate



X	Y	$F = \overline{X \oplus Y}$
0	0	1
0	1	0
1	0	0
1	1	1

Exclusive-OR (XOR) Function

- XOR (also $\oplus$) : the "not-equal" function
- $XOR(X,Y) = X \oplus Y = X'Y + XY'$
- Identities:
  - $X \oplus 0 = X$
  - $X \oplus 1 = X'$
  - $X \oplus X = 0$
  - $X \oplus X' = 1$
- Properties:
  - $X \oplus Y = Y \oplus X$
  - $(X \oplus Y) \oplus W = X \oplus (Y \oplus W)$

XOR function implementation

- XOR(a,b) = ab’ + a’b
- Straightforward: 5 gates
  - 2 inverters, two 2-input ANDs, one 2-input OR
  - 2 inverters & 3 2-input NANDs
- Nonstraightforward:
  - 4 NAND gates

XOR circuit with 4 NANDs

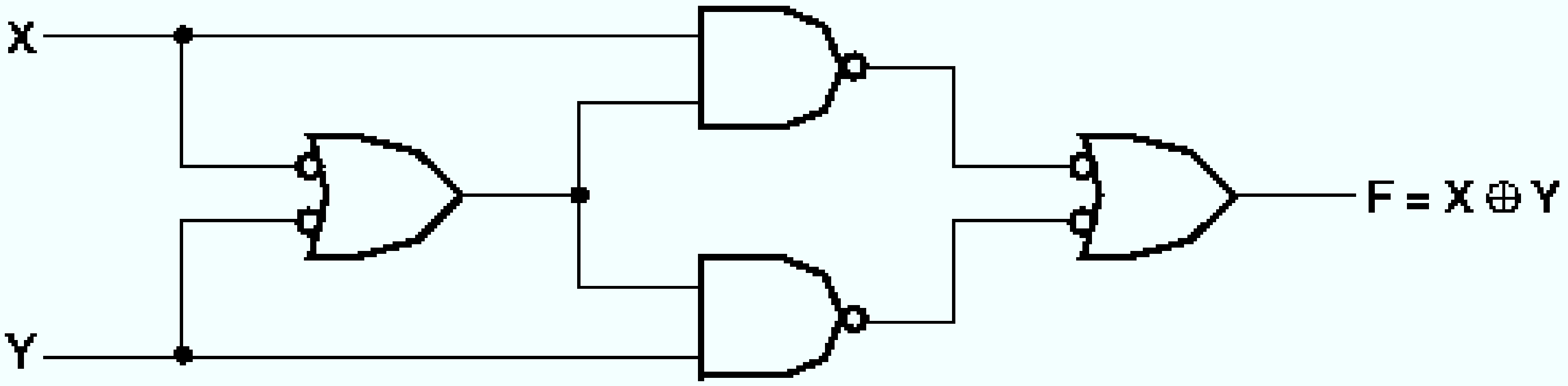


Fig. 2-37 Exclusive-OR Constructed with NAND Gates

UNIT III

- COMBINATIONAL LOGIC
  - BINARY ADDER –SUBTRACTER
  - DECIMAL ADDER
  - BINARY MMULTIPLIER
  - MAGNITUDE COMPARATOR
  - DECODER, ENCODER
  - MULTIPLEXERS AND DEMULTIPLEXER

III UNIT

Combinational Logic

- Logic circuits for digital systems may be combinational or sequential.
- A combinational circuit consists of input variables, logic gates, and output variables.

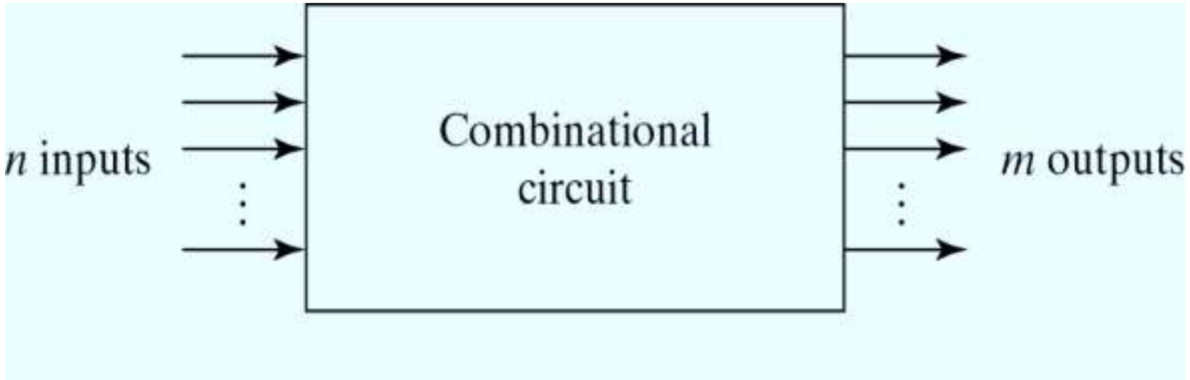


Fig. 4-1 Block Diagram of Combinational Circuit

Analysis procedure

- To obtain the output Boolean functions from a logic diagram, proceed as follows:

1. Label all gate outputs that are a function of input variables with arbitrary symbols. Determine the Boolean functions for each gate output.
2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the Boolean functions for these gates.
3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

Example

$F_2 = AB + AC + BC$; $T_1 = A + B + C$; $T_2 = ABC$; $T_3 = F_2'T_1$;

$F_1 = T_3 + T_2$

$F_1 = T_3 + T_2 = F_2'T_1 + ABC = A'BC' + A'B'C + AB'C' + ABC$

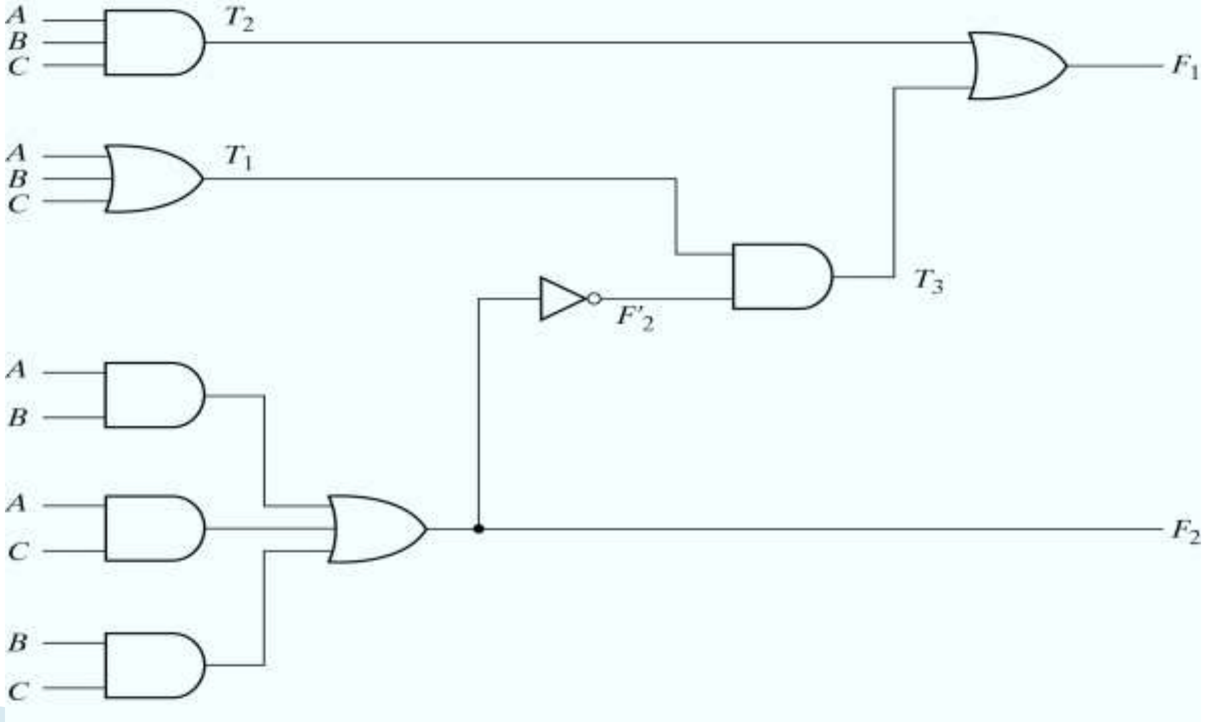


Fig. 4-2 Logic Diagram for Analysis Example

Derive truth table from logic diagram

- We can derive the truth table in Table 4-1 by using the circuit of Fig.4-2.

Table 4-1
Truth Table for the Logic Diagram of Fig. 4-2

A	B	C	F_2	F_2'	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Design procedure

1. Table4-2 is a Code-Conversion example, first, we can list the relation of the BCD and Excess-3 codes in the truth table.

Table 4-2
Truth Table for Code-Conversion Example

Input BCD				Output Excess-3 Code			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Karnaugh map

2. For each symbol of the Excess-3 code, we use 1’s to draw the map for simplifying Boolean function.

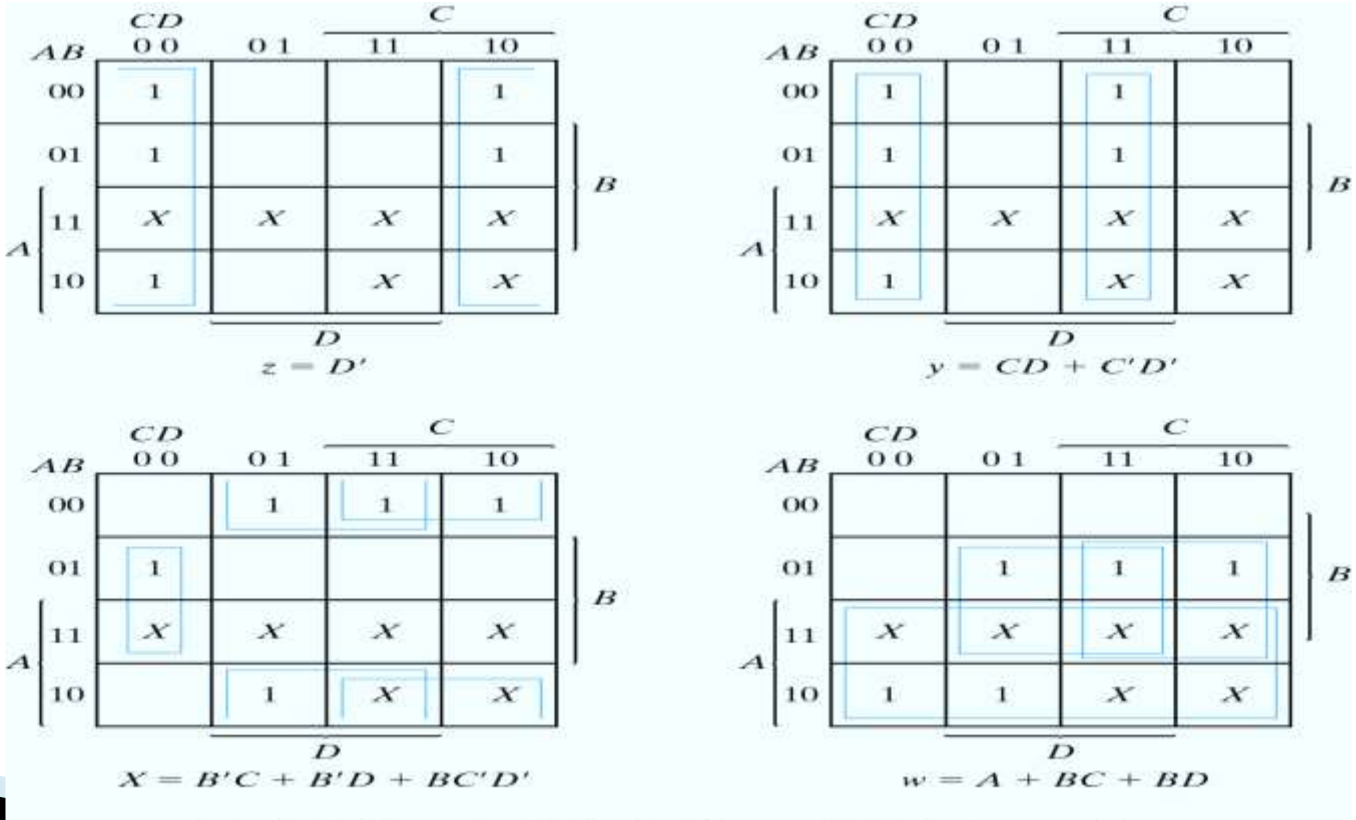


Fig. 4-3 Maps for BCD to Excess-3 Code Converter

Circuit implementation

$z = D'$; $y = CD + C'D' = CD + (C + D)'$

$x = B'C + B'D + BC'D' = B'(C + D) + B(C + D)'$

$w = A + BC + BD = A + B(C + D)$

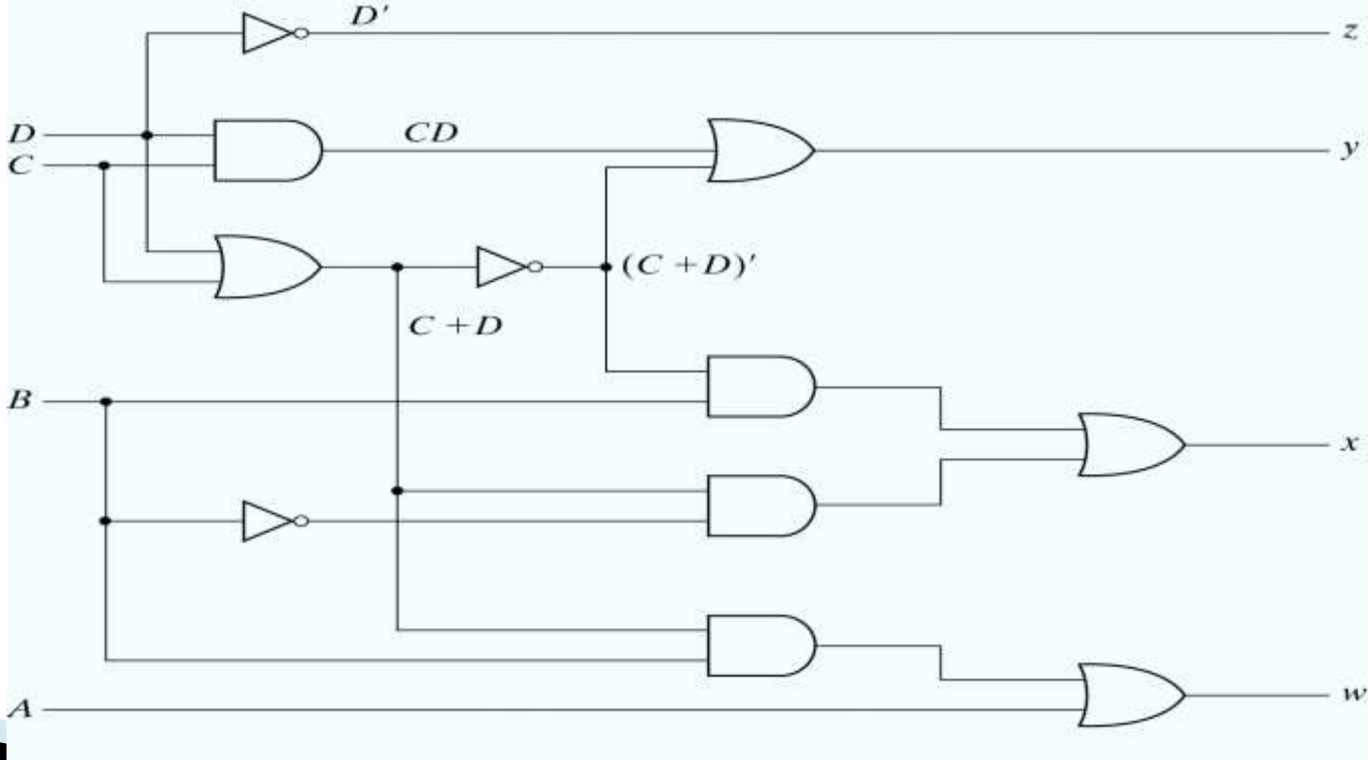


Fig. 4-4 Logic Diagram for BCD to Excess-3 Code Converter

Binary Adder–Subtractor

- A combinational circuit that performs the addition of two bits is called a half adder.
- The truth table for the half adder is listed below:

Table 4-3
Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S: Sum
C: Carry

$S = x'y + xy'$

$C = xy$

Implementation of Half-Adder

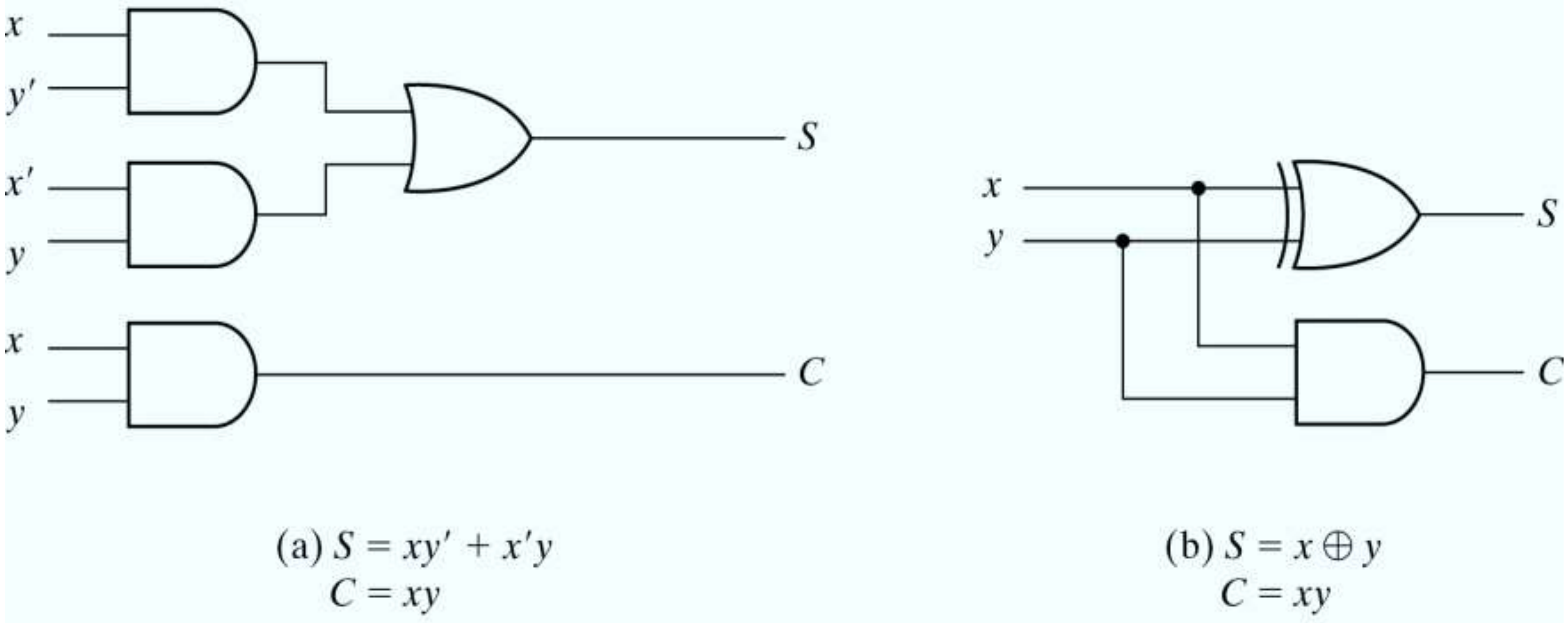


Fig. 4-5 Implementation of Half-Adder

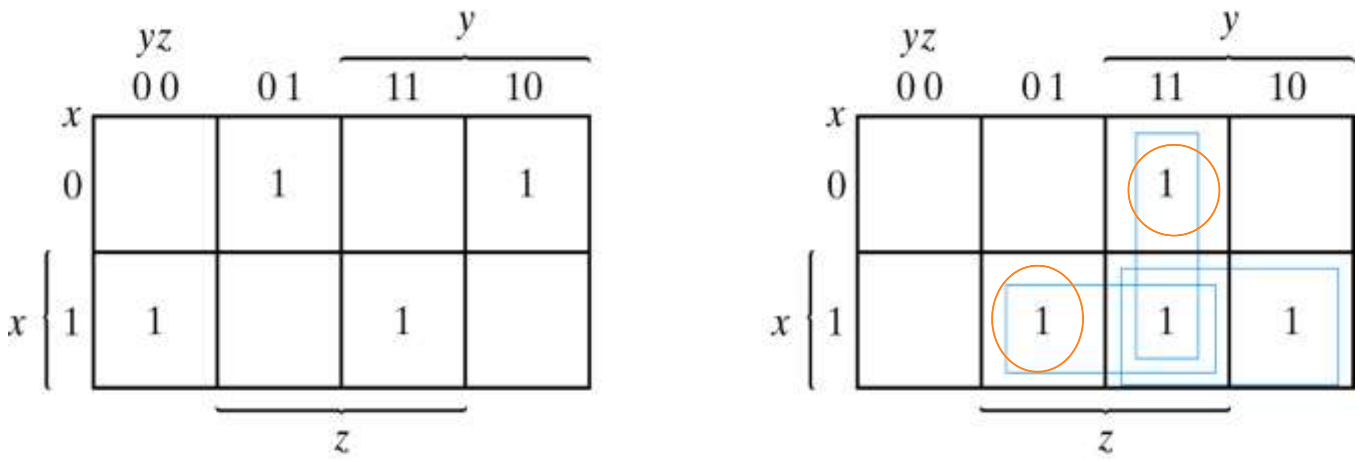
Full-Adder

- One that performs the addition of three bits(two significant bits and a previous carry) is a full adder.

Table 4-4
Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Simplified Expressions



$S = x'y'z + x'yz' + xy'z' + xyz$

$$C = xy + xz + yz = xy + \boxed{xy'z + x'yz}$$

Fig. 4-6 Maps for Full Adder

$$S = x'y'z + x'yz' + xy'z' + xyz$$
$$C = xy + xz + yz$$

Full adder implemented in SOP

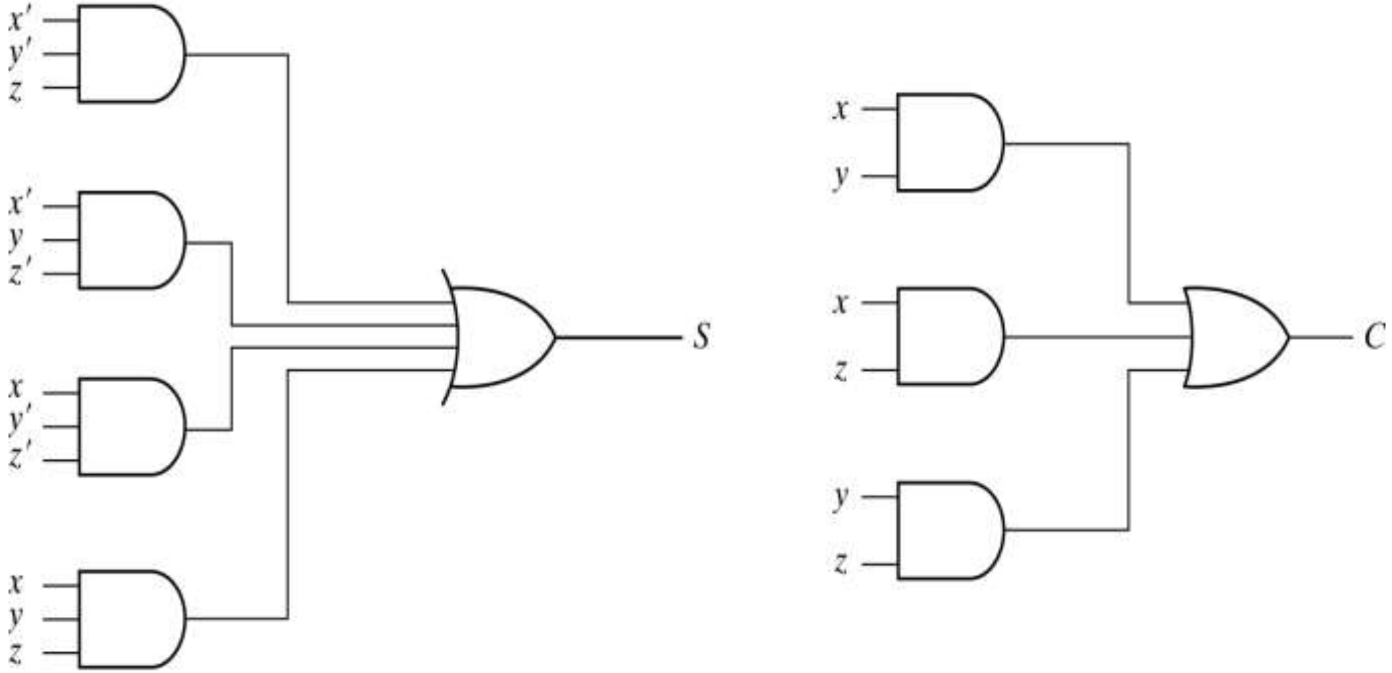


Fig. 4-7 Implementation of Full Adder in Sum of Products

Another implementation

- Full-adder can also implemented with two half adders and one OR gate (Carry Look-Ahead adder).

$$S = z \oplus (x \oplus y)$$
$$= z'(xy' + x'y) + z(xy' + x'y)'$$
$$= xy'z' + x'yz' + xyz + x'y'z$$
$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

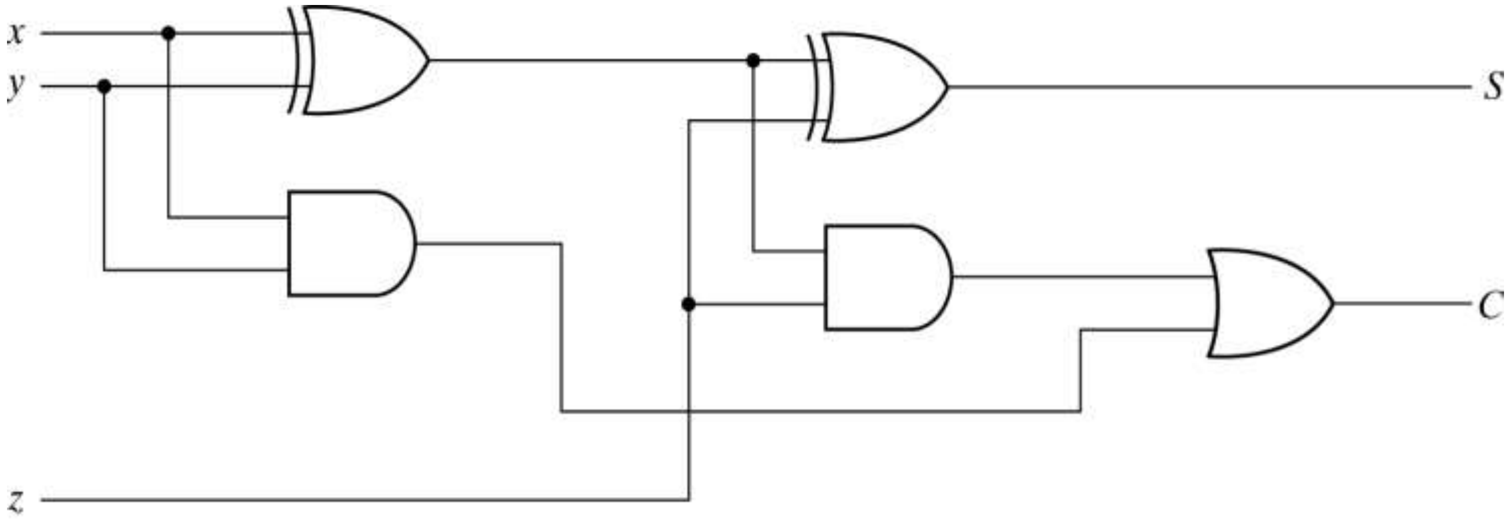


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

Binary adder

- This is also called Ripple Carry Adder ,because of the construction with full adders are connected in cascade.

Subscript *i*:	**3**	**2**	**1**	**0**	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

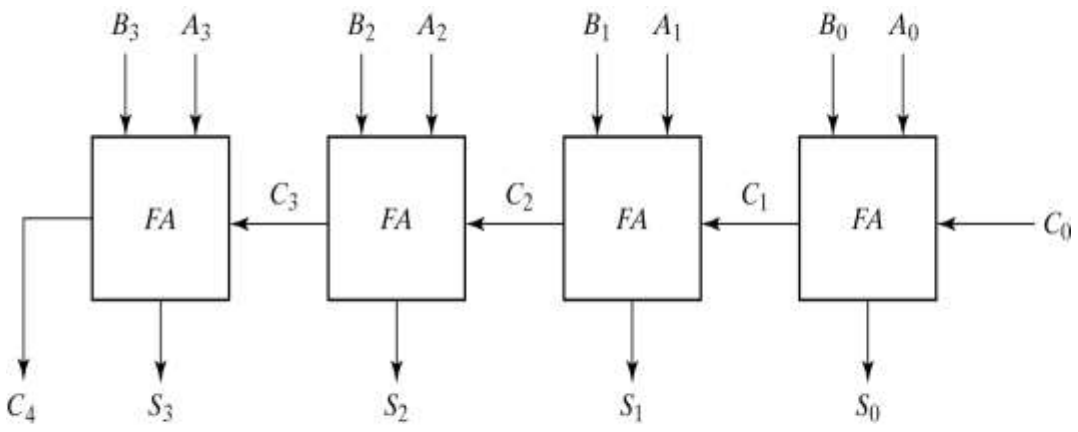


Fig. 4-9 4-Bit Adder

Carry Propagation

- Fig.4-9 causes a unstable factor on carry bit, and produces a longest propagation delay.
- The signal from C_i to the output carry C_{i+1}, propagates through an AND and OR gates, so, for an n-bit RCA, there are 2n gate levels for the carry to propagate from input to output.

Carry Propagation

- Because the propagation delay will affect the output signals on different time, so the signals are given enough time to get the precise and stable outputs.
- The most widely used technique employs the principle of carry look-ahead to improve the speed of the algorithm.

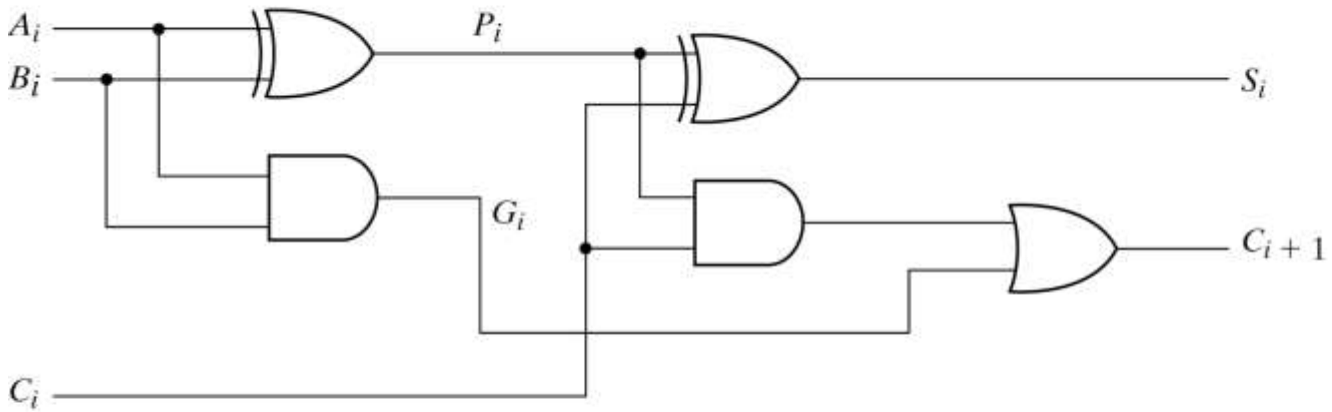


Fig. 4-10 Full Adder with P and G Shown

Boolean functions

$P_i = A_i \oplus B_i$ steady state value

$G_i = A_iB_i$ steady state value

Output sum and carry

$S_i = P_i \oplus C_i$

$C_{i+1} = G_i + P_iC_i$

G_i : carry generate P_i : carry propagate

C_0 = input carry

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

- C_3 does not have to wait for C_2 and C_1 to propagate.

Logic diagram of carry look-ahead generator

- C_3 is propagated at the same time as C_2 and C_1.

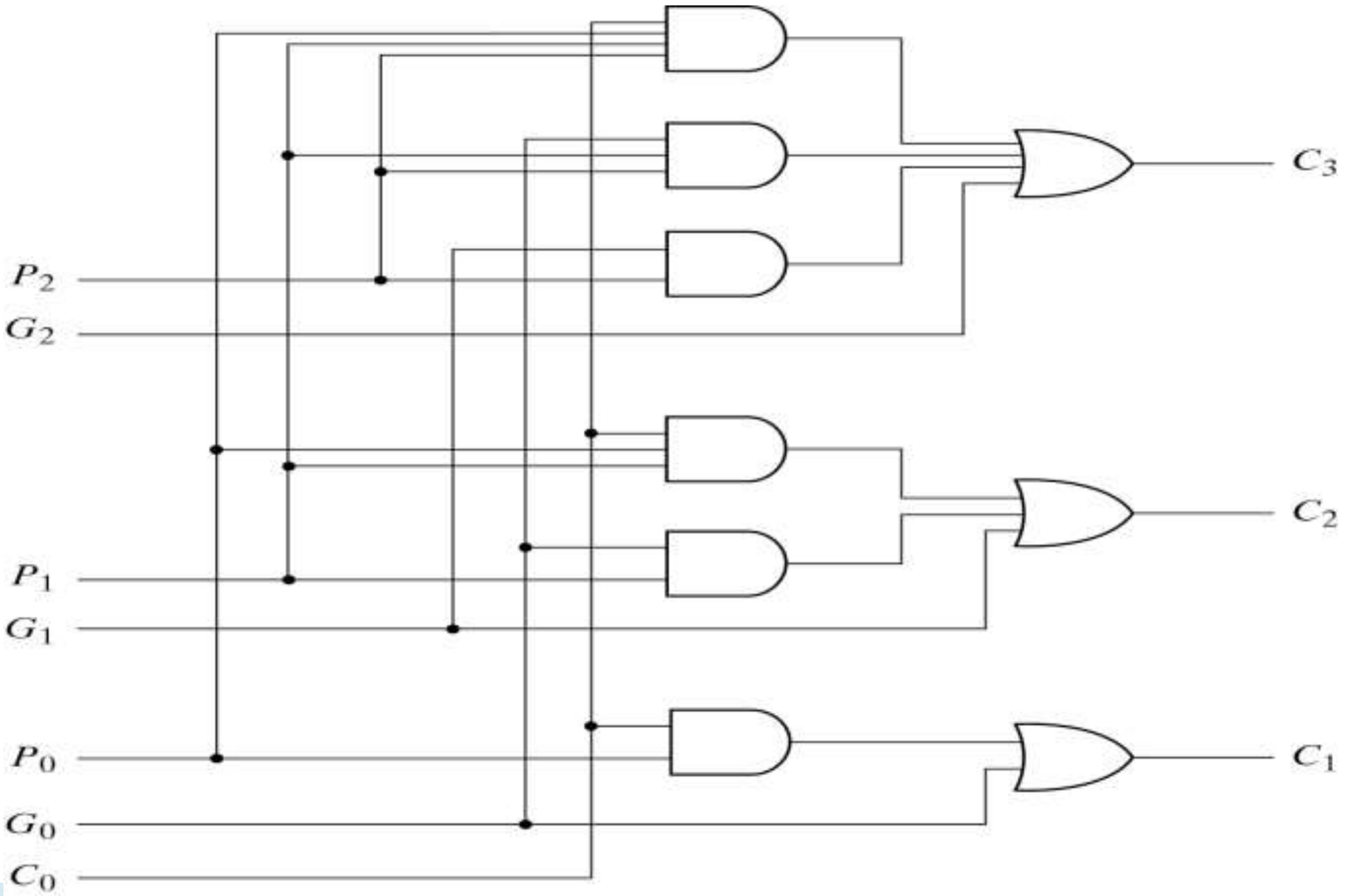


Fig. 4-11 Logic Diagram of Carry Lookahead Generator

4-bit adder with carry lookahead

- Delay time of n-bit CLAA = XOR + (AND + OR) + XOR

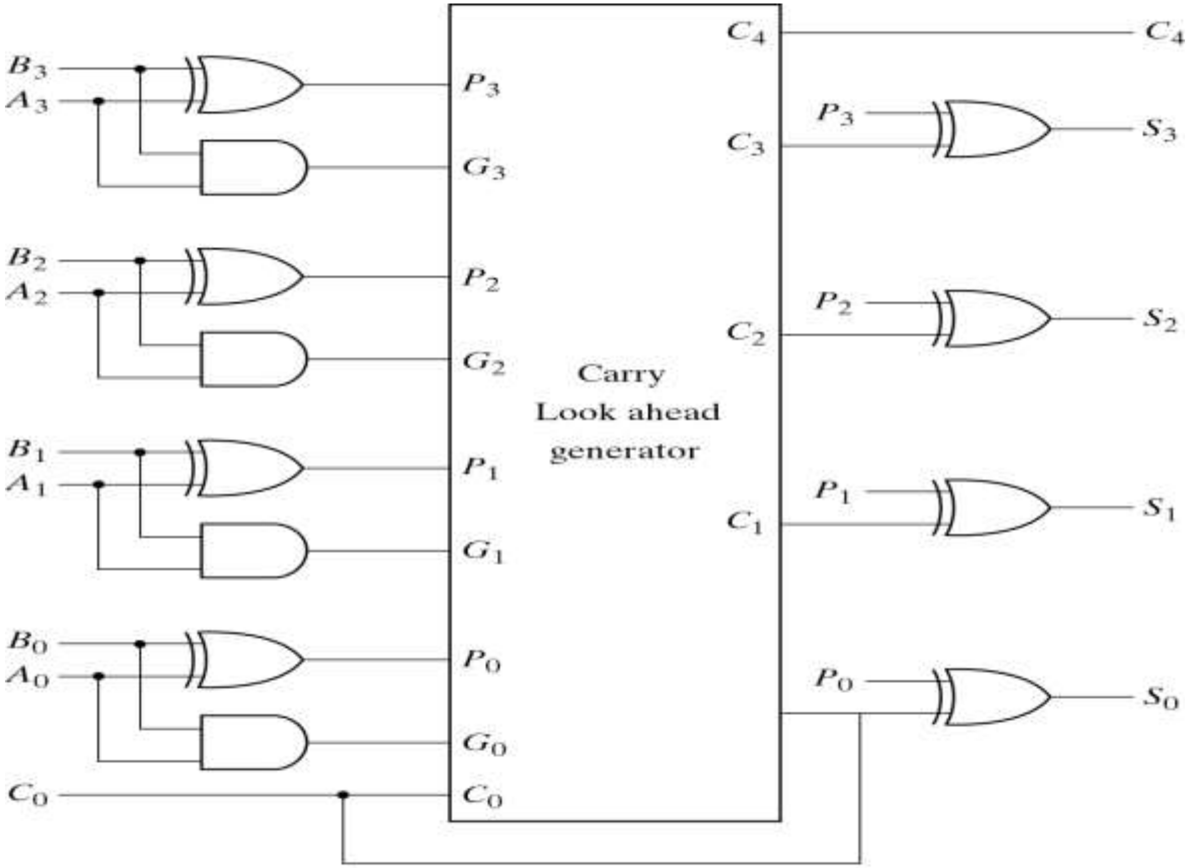


Fig. 4-12 4-Bit Adder with Carry Lookahead

Binary subtractor

M = 1→subtractor ; M = 0→adder

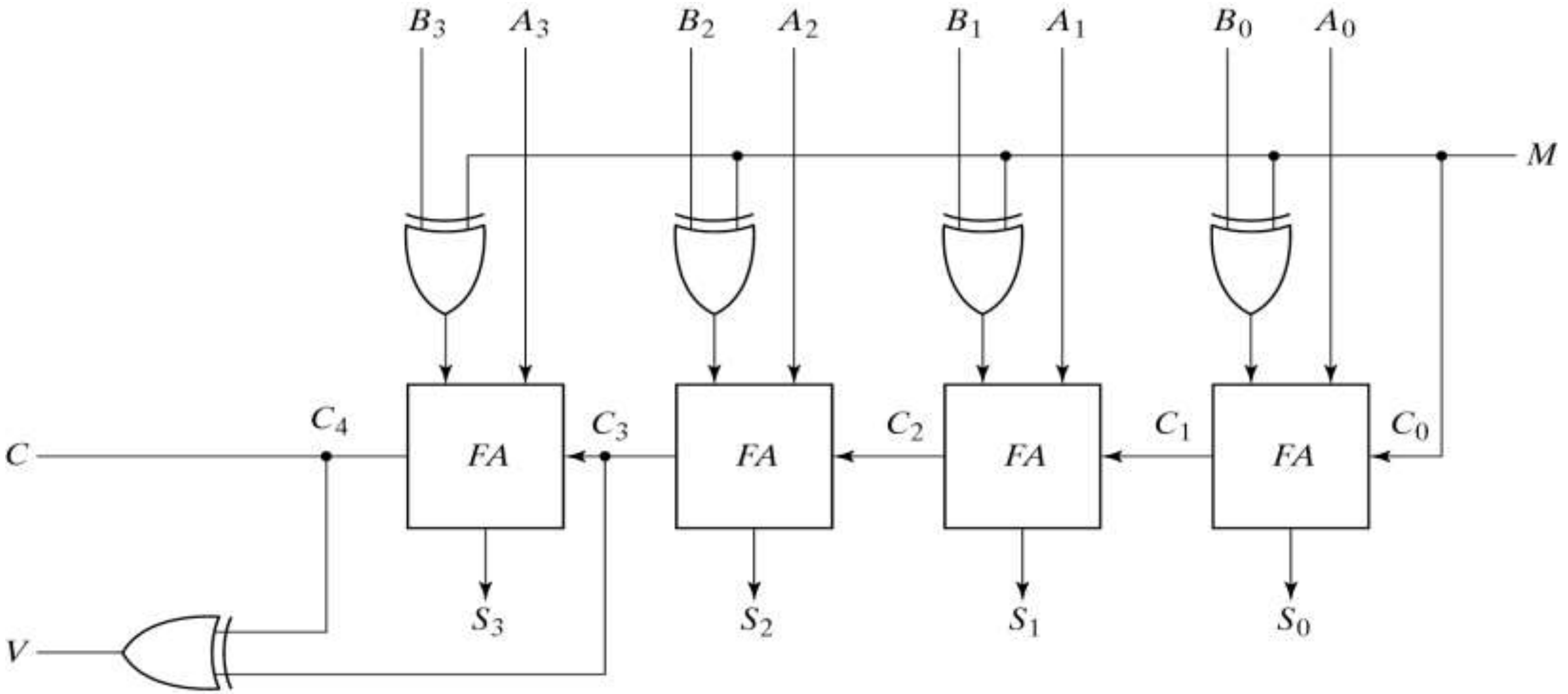


Fig. 4-13 4-Bit Adder Subtractor

Decimal adder

BCD adder can't exceed 9 on each input digit. K is the carry.

Table 4-5
Derivation of BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

Rules of BCD adder

- When the binary sum is greater than 1001, we obtain a non-valid BCD representation.
- The addition of binary 6(0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.
- To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8, we specify further that either Z_4 or Z_2 must have a 1.

$$C = K + Z_8Z_4 + Z_8Z_2$$

Implementation of BCD adder

- A decimal parallel adder that adds n decimal digits needs n BCD adder stages.
- The output carry from one stage must be connected to the input carry of the next higher-order stage.

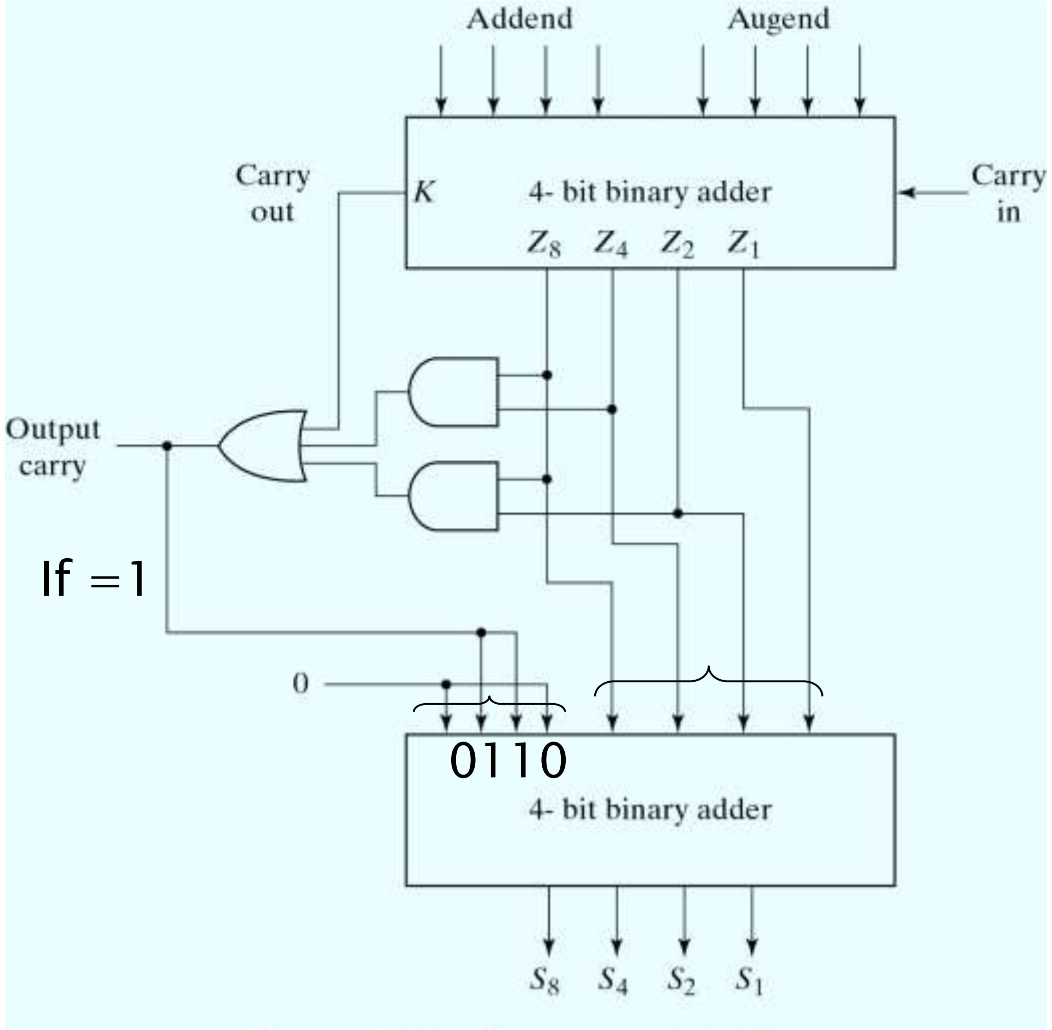


Fig. 4-14 Block Diagram of a BCD Adder

Binary multiplier

- Usually there are more bits in the partial products and it is necessary to use full adders to produce the sum of the partial products.

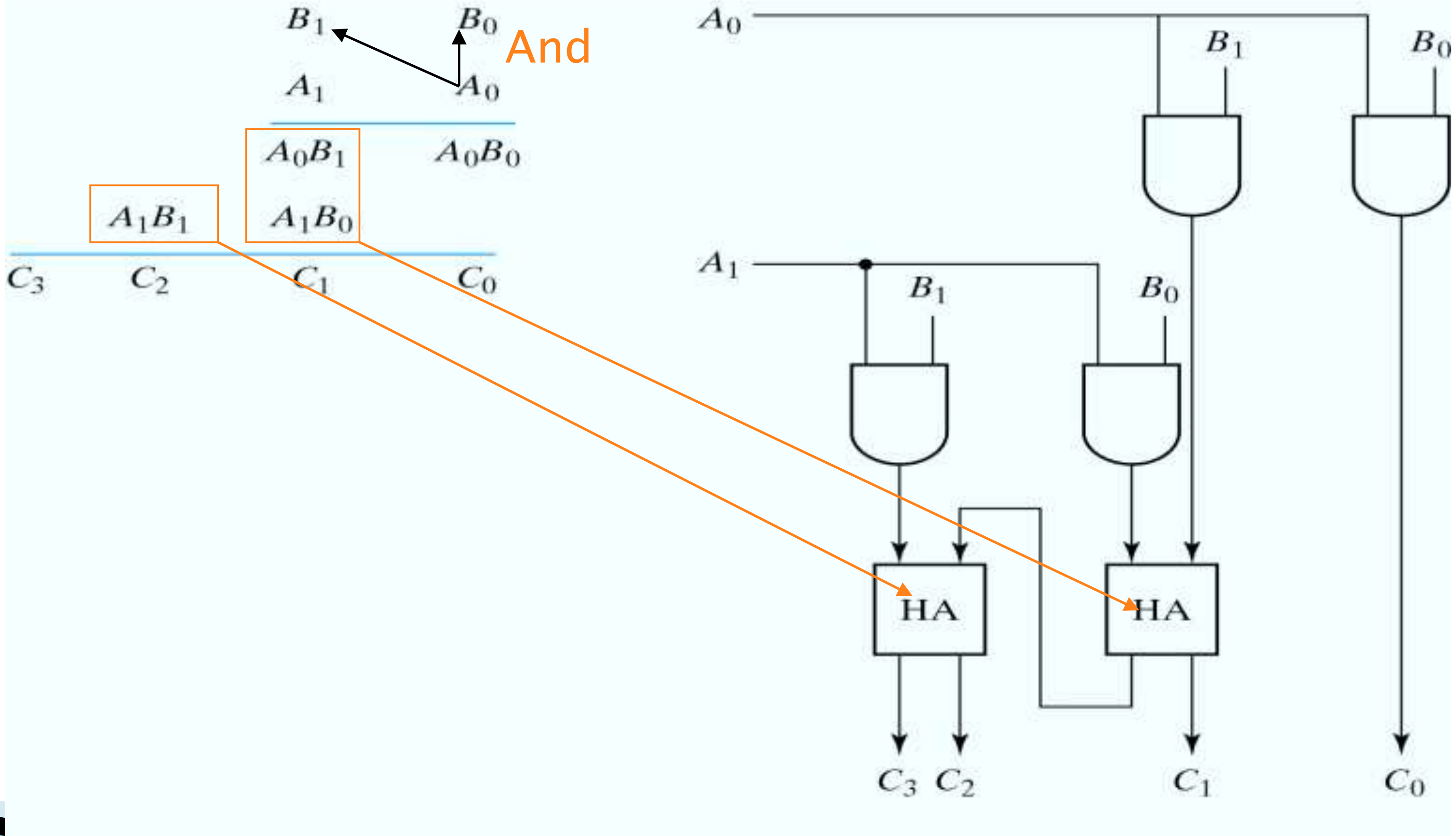


Fig. 4-15 2-Bit by 2-Bit Binary Multiplier

4-bit by 3-bit binary multiplier

- For J multiplier bits and K multiplicand bits we need (J X K) AND gates and (J − 1) K-bit adders to produce a product of J+K bits.
- K=4 and J=3, we need 12 AND gates and two 4-bit adders.

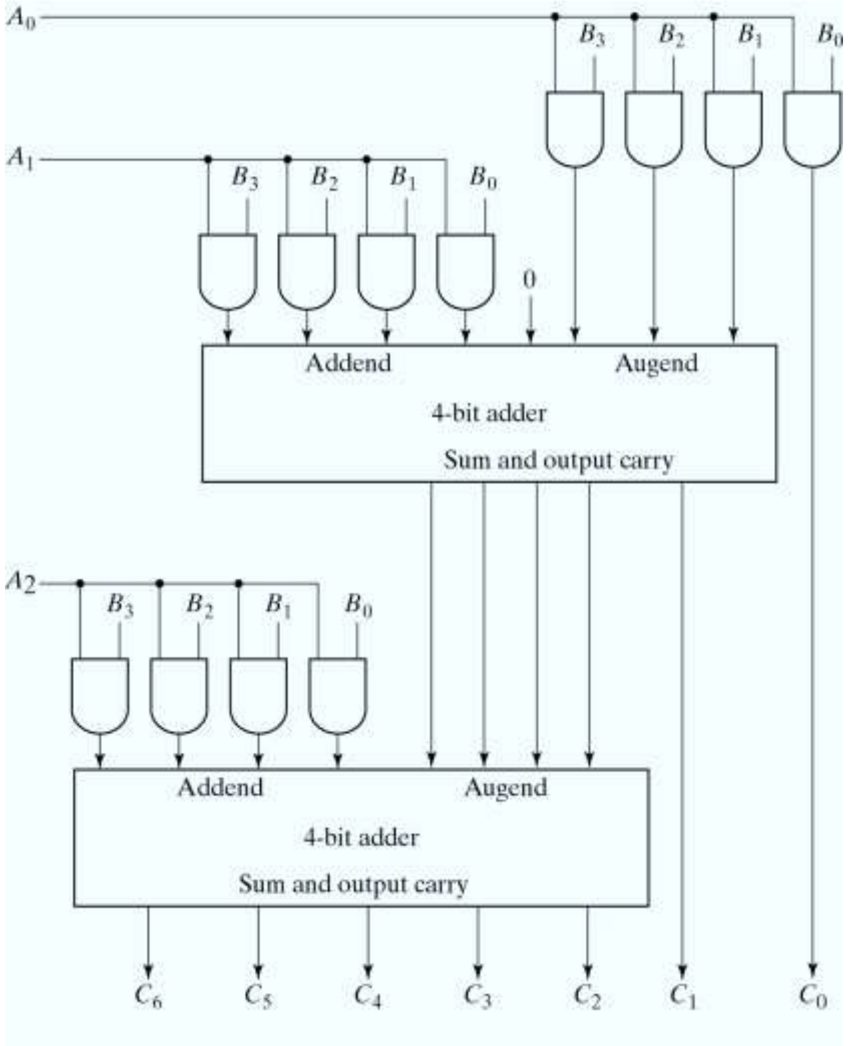


Fig. 4-16 4-Bit by 3-Bit Binary Multiplier

Magnitude comparator

- The equality relation of each pair of bits can be expressed logically with an exclusive-NOR function as:

  $A = A_3A_2A_1A_0$; $B = B_3B_2B_1B_0$

  $x_i = A_iB_i + A_i'B_i'$ for $i = 0, 1, 2, 3$

  $$(A = B) = x_3x_2x_1x_0$$

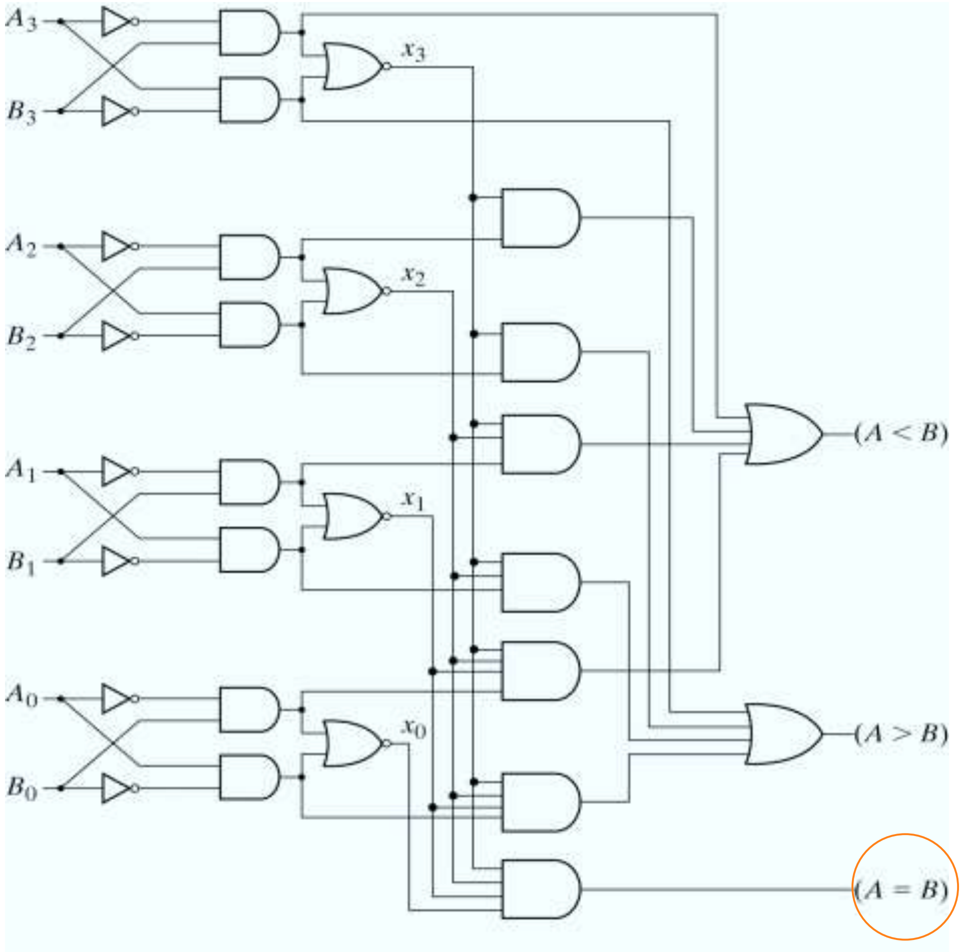


Fig. 4-17 4-Bit Magnitude Comparator

Magnitude comparator

- We inspect the relative magnitudes of pairs of MSB. If equal, we compare the next lower significant pair of digits until a pair of unequal digits is reached.
- If the corresponding digit of A is 1 and that of B is 0, we conclude that A>B.

$(A>B)=A_3B'_3+x_3A_2B'_2+x_3x_2A_1B'_1+x_3x_2x_1A_0B'_0$

$(A<B)=A'_3B_3+x_3A'_2B_2+x_3x_2A'_1B_1+x_3x_2x_1A'_0B_0$

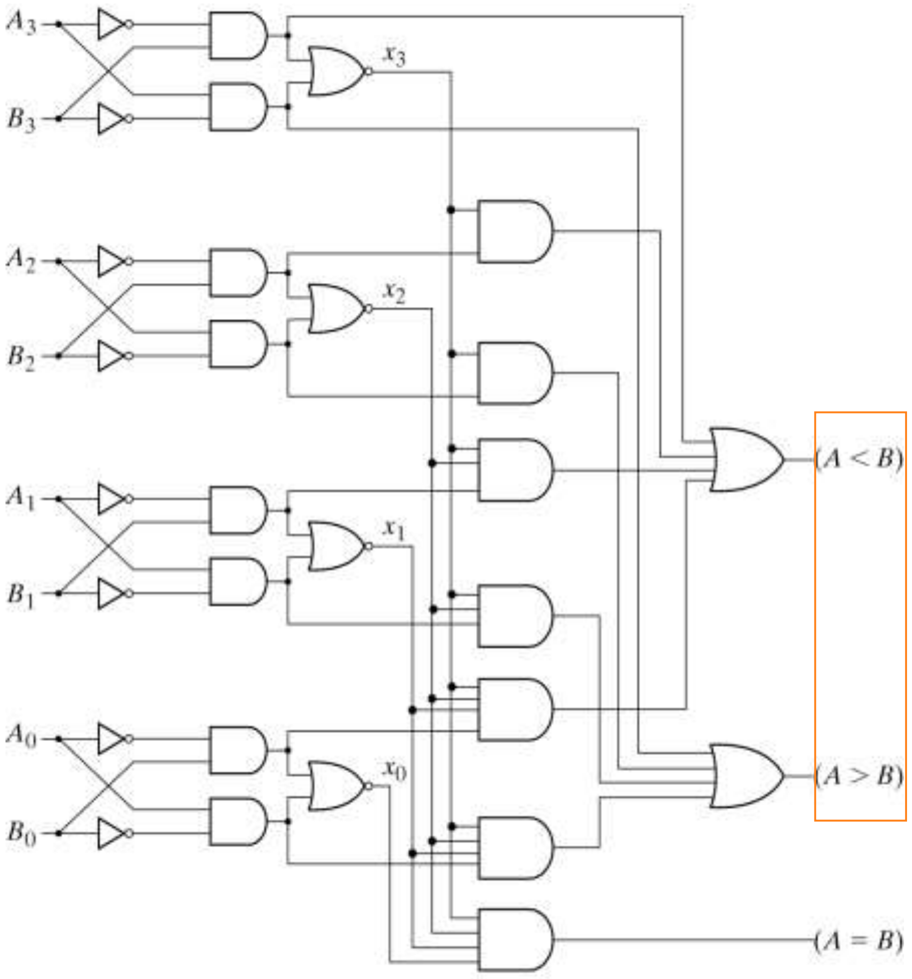


Fig. 4-17 4-Bit Magnitude Comparator

Decoders

- The decoder is called n-to-m-line decoder, where $m \leq 2^n$.
- the decoder is also used in conjunction with other code converters such as a BCD-to-seven_segment decoder.
- 3-to-8 line decoder: For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1.

Implementation and truth table

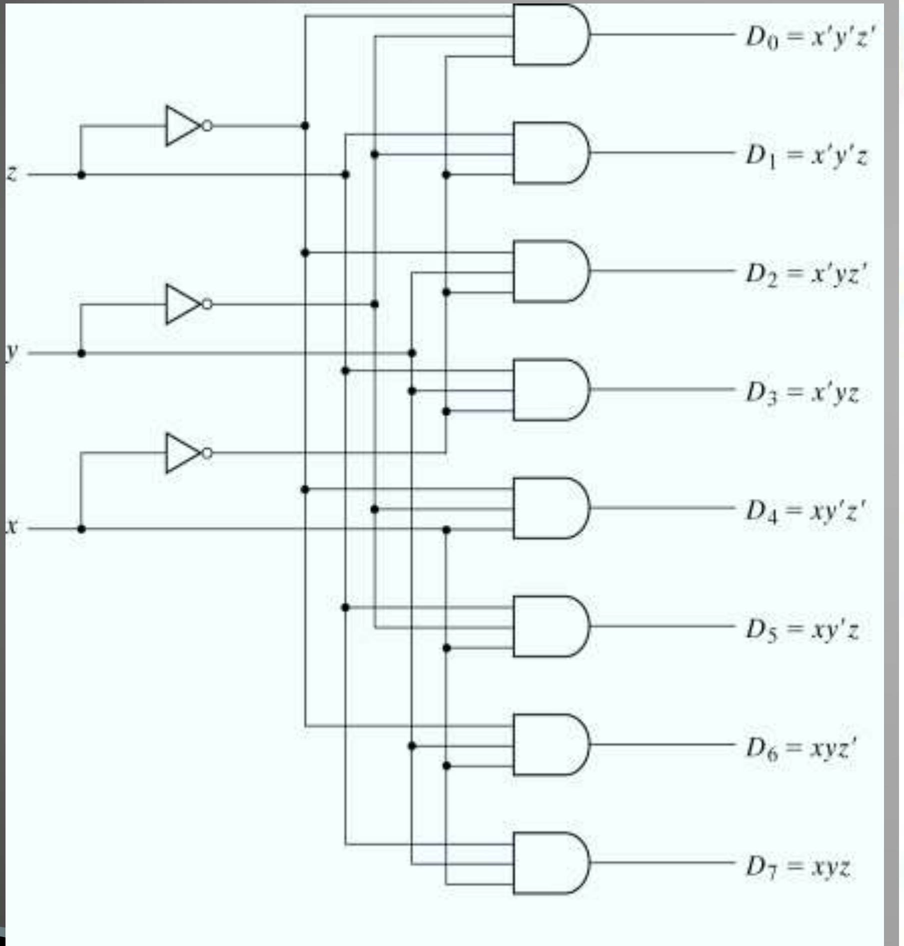


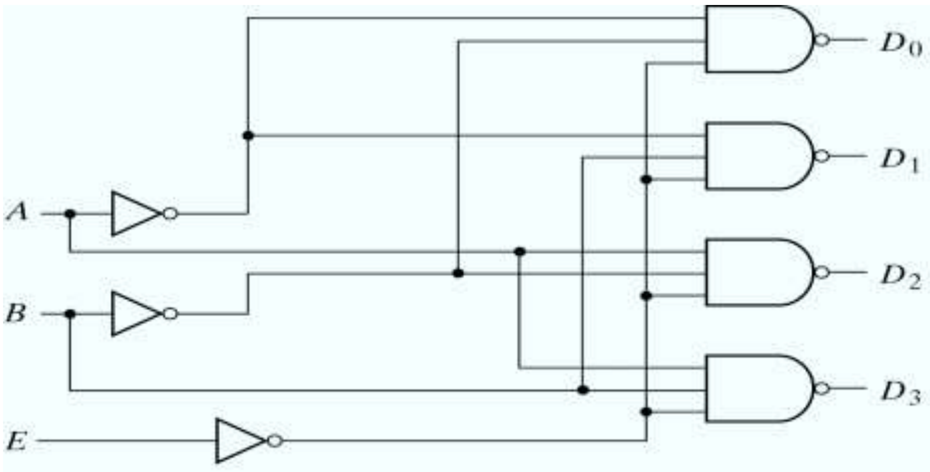
Fig. 4-18 3-to-8-Line Decoder

Table 4-6
Truth Table of a 3-to-8-Line Decoder

Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Decoder with enable input

- Some decoders are constructed with NAND gates, it becomes more economical to generate the decoder minterms in their complemented form.
- As indicated by the truth table , only one output can be equal to 0 at any given time, all other outputs are equal to 1.



(a) Logic diagram

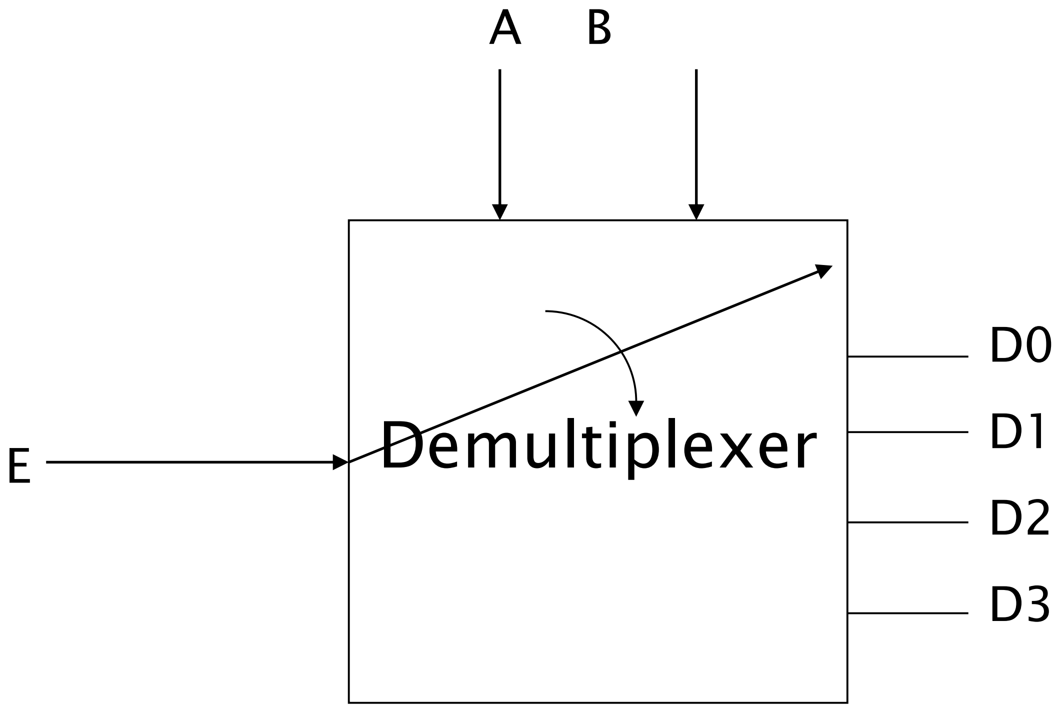
E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) Truth table

Fig. 4-19 2-to-4-Line Decoder with Enable Input

Demultiplexer

- A decoder with an enable input is referred to as a decoder/demultiplexer.
- The truth table of demultiplexer is the same with decoder.



3-to-8 decoder with enable implement the 4-to-16 decoder

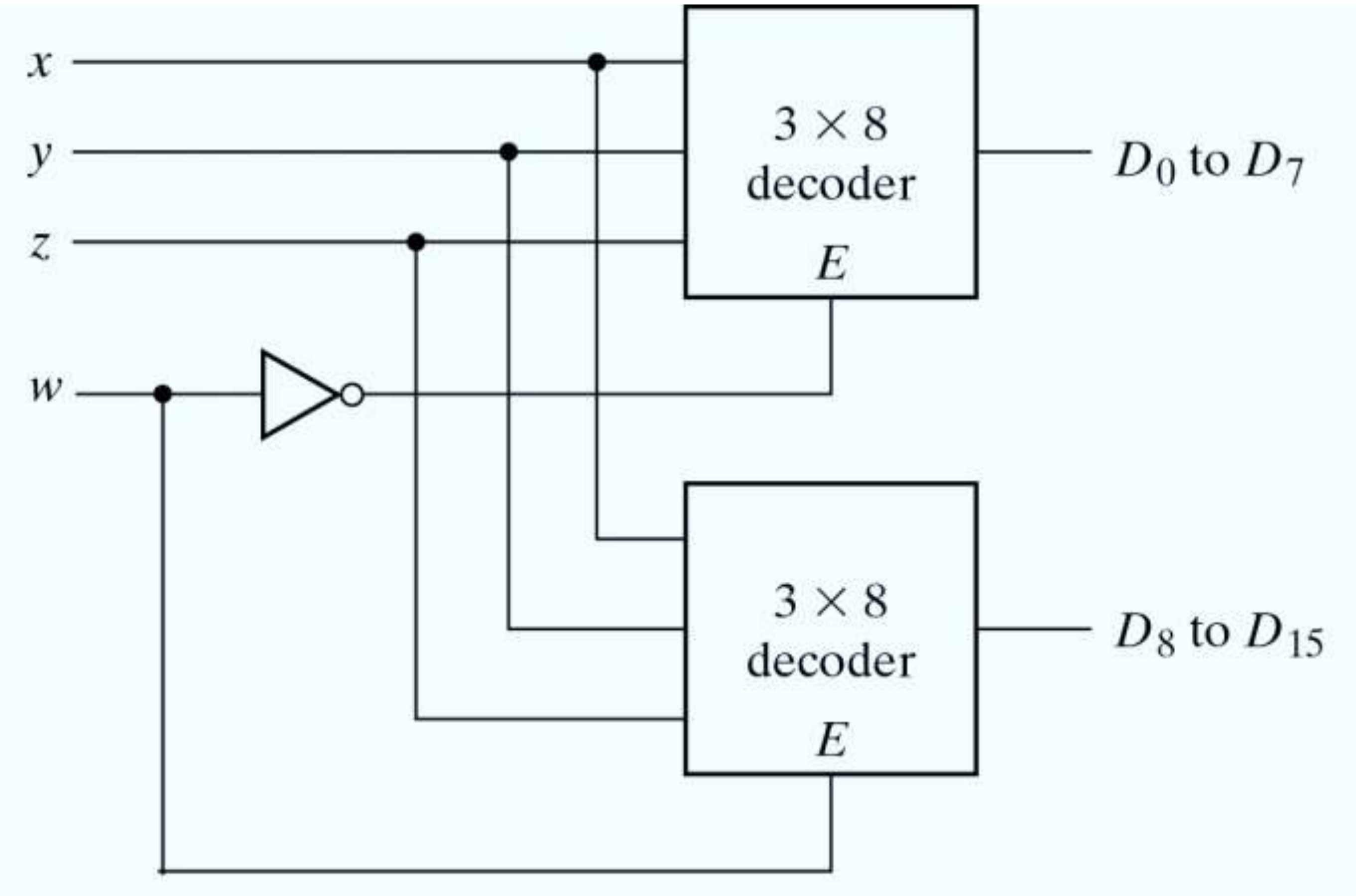


Fig. 4-20 4 × 16 Decoder Constructed with Two 3 × 8 Decoders

Implementation of a Full Adder with a Decoder

- From table 4-4, we obtain the functions for the combinational circuit in sum of minterms:

  $S(x, y, z) = \sum(1, 2, 4, 7)$

  $C(x, y, z) = \sum(3, 5, 6, 7)$

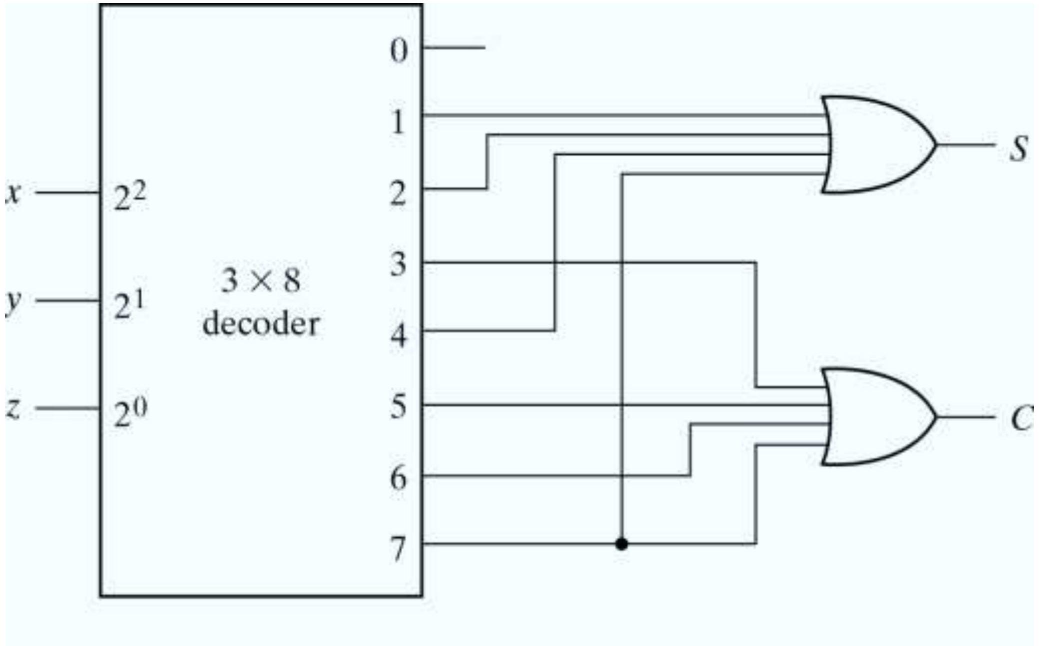


Fig. 4-21 Implementation of a Full Adder with a Decoder

Encoders

- An encoder is the inverse operation of a decoder.
- We can derive the Boolean functions by table 4-7

$$z = D_1 + D_3 + D_5 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$x = D_4 + D_5 + D_6 + D_7$$

Table 4-7
Truth Table of Octal-to-Binary Encoder

Inputs								**Outputs**		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Priority encoder

- If two inputs are active simultaneously, the output produces an undefined combination. We can establish an input priority to ensure that only one input is encoded.
- Another ambiguity in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0; the output is the same as when D_0 is equal to 1.
- The discrepancy tables on Table 4-7 and Table 4-8 can resolve aforesaid condition by providing one more output to indicate that at least one input is equal to 1.

Priority encoder

V=0→no valid inputs
V=1→valid inputs

X's in output columns represent don't-care conditions
X's in the input columns are useful for representing a truth table in condensed form. Instead of listing all 16 minterms of four variables.

Table 4-8
Truth Table of a Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

4-input priority encoder

- Implementation of table 4-8

$x = D_2 + D_3$

$y = D_3 + D_1D'_2$

$V = D_0 + D_1 + D_2 + D_3$

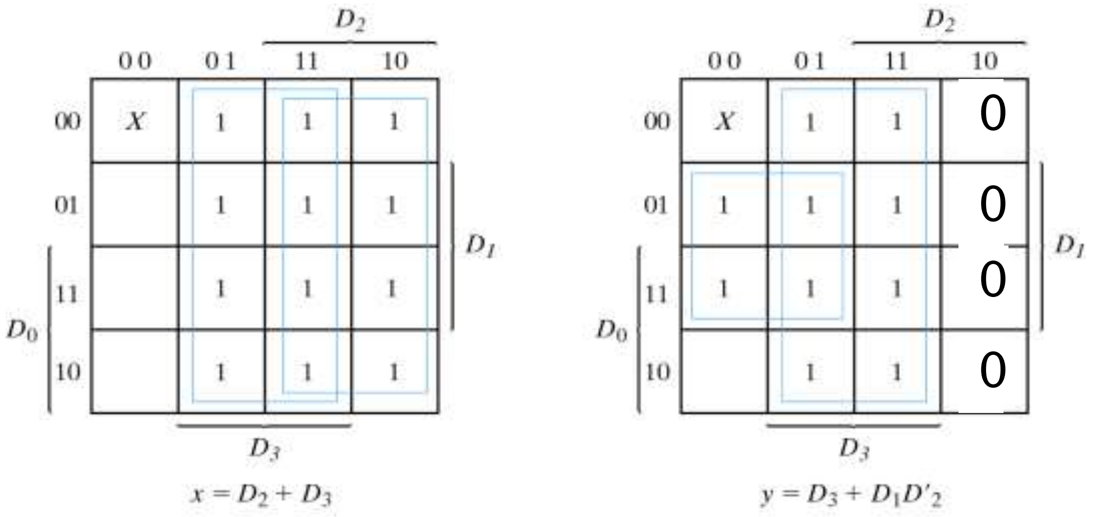


Fig. 4-22 Maps for a Priority Encoder

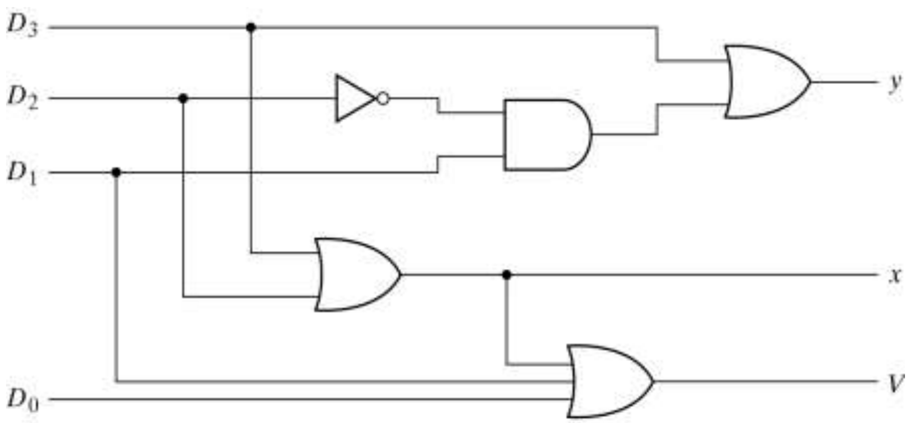


Fig. 4-23 4-Input Priority Encoder

Multiplexers

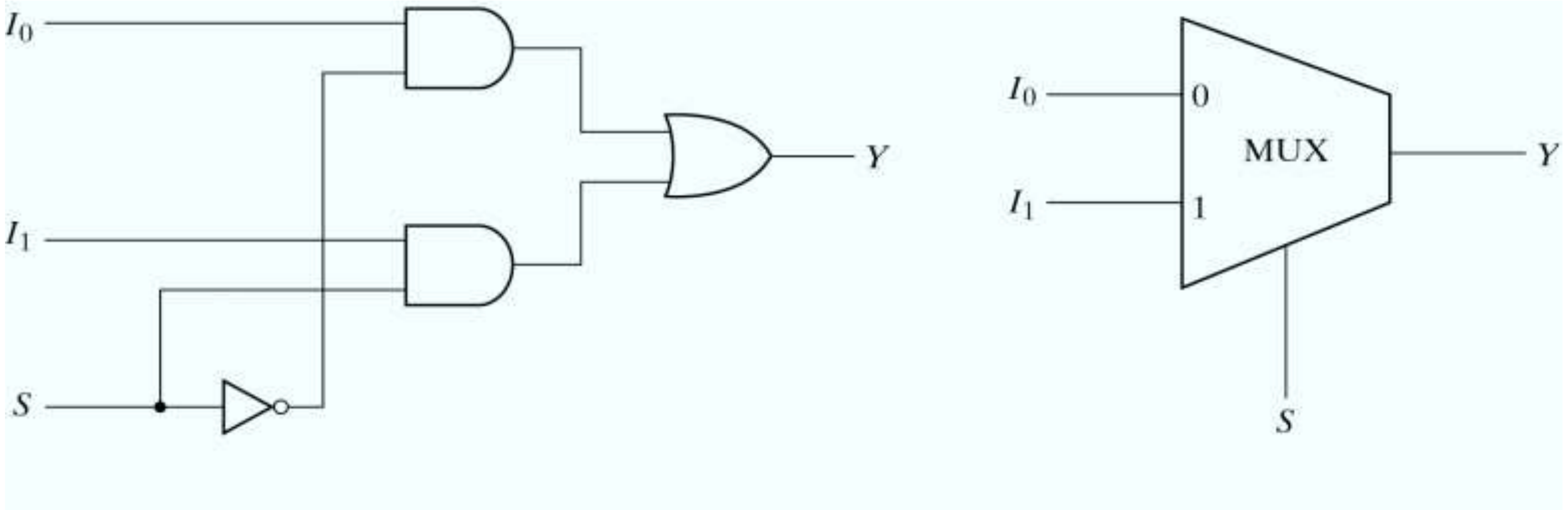
$S = 0, Y = I_0$

$S = 1, Y = I_1$

Truth Table→

S	Y
0	I_0
1	I_1

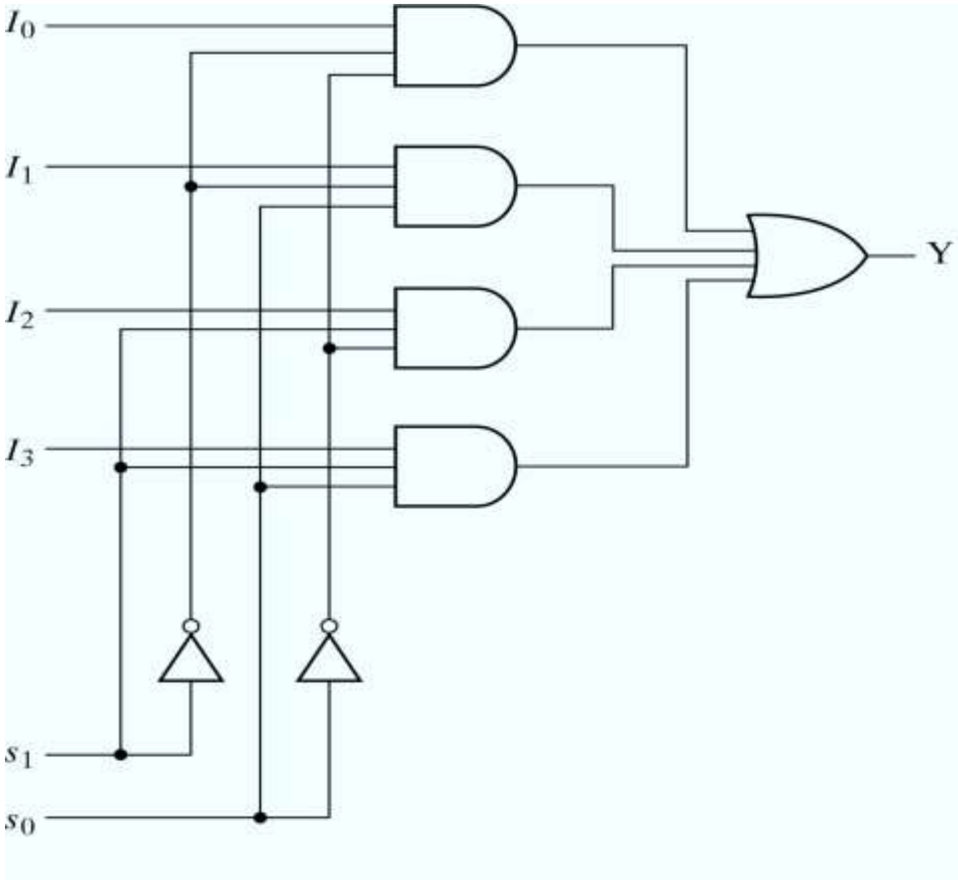
$Y = S'I_0 + SI_1$



(a) Logic diagram (b) Block diagram

Fig. 4-24 2-to-1-Line Multiplexer

4-to-1 Line Multiplexer



(a) Logic diagram

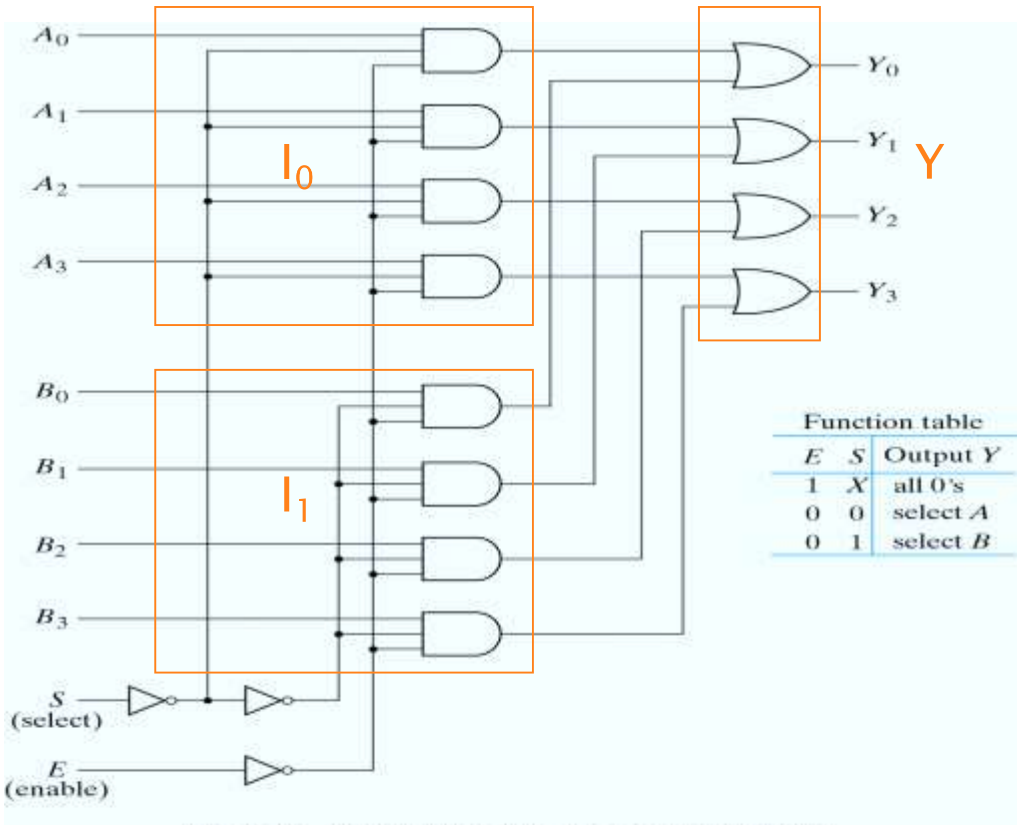
s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

Fig. 4-25 4-to-1-Line Multiplexer

Quadruple 2-to-1 Line Multiplexer

- Multiplexer circuits can be combined with common selection inputs to provide multiple-bit selection logic. Compare with Fig4-24.



E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

Fig. 4-26 Quadruple 2-to-1-Line Multiplexer

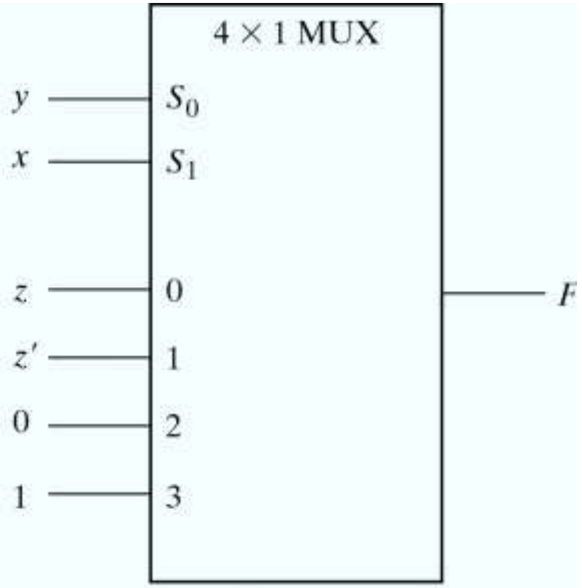
Boolean function implementation

- A more efficient method for implementing a Boolean function of n variables with a multiplexer that has n-1 selection inputs.

$$F(x, y, z) = \Sigma(1,2,6,7)$$

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table



(b) Multiplexer implementation

Fig. 4-27 Implementing a Boolean Function with a Multiplexer

4-input function with a multiplexer

$$F(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$$

A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

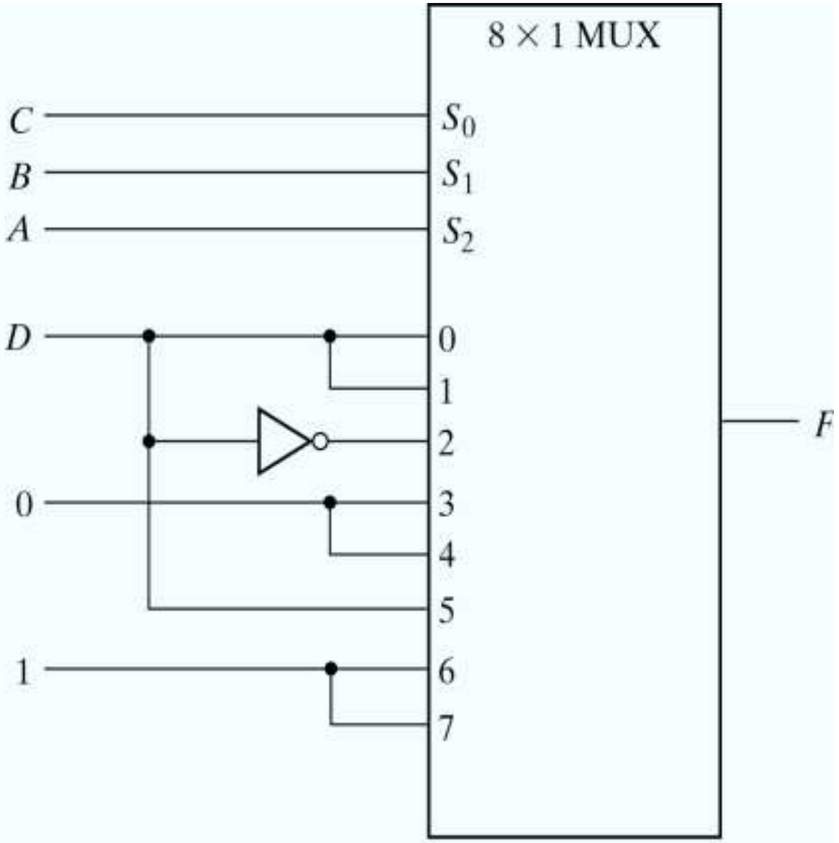


Fig. 4-28 Implementing a 4-Input Function with a Multiplexer

Three-State Gates

- A multiplexer can be constructed with three-state gates.

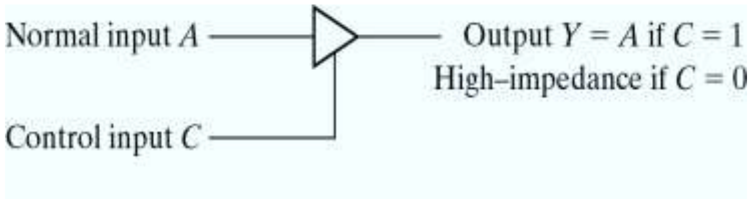


Fig. 4-29 Graphic Symbol for a Three-State Buffer

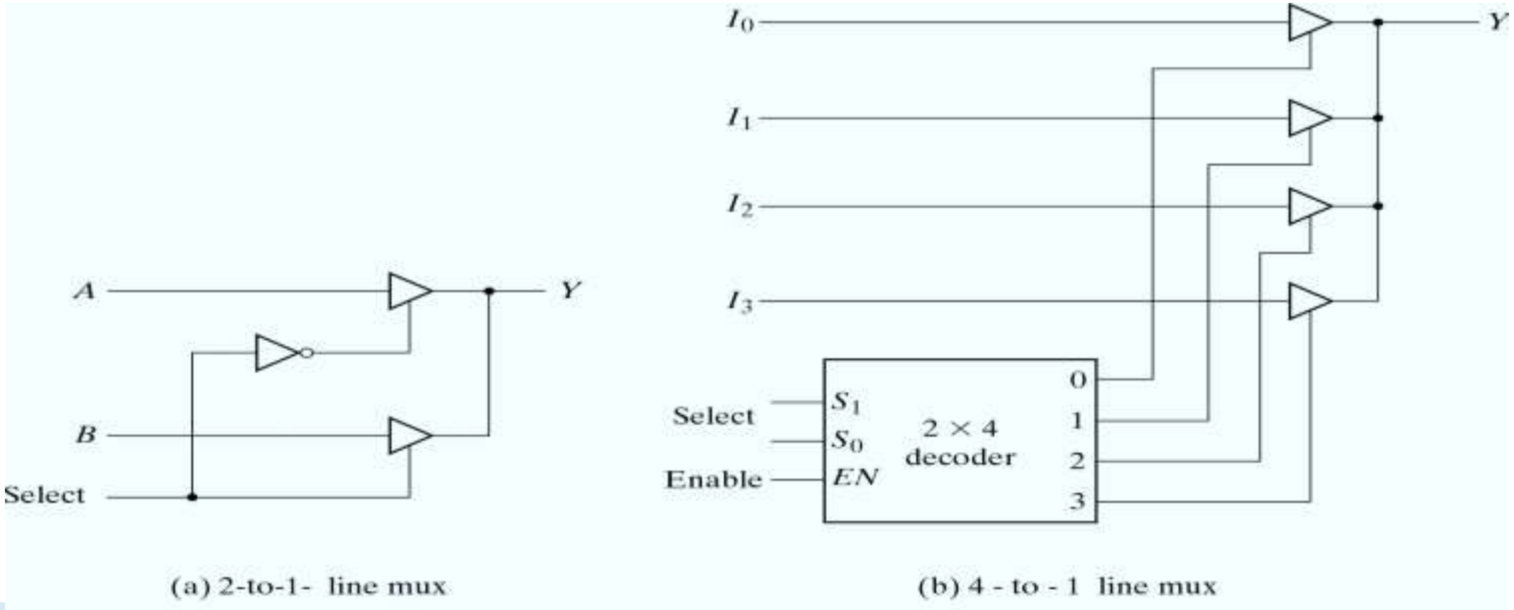


Fig. 4-30 Multiplexers with Three-State Gates

Design methodologies

- There are two basic types of design methodologies: top-down and bottom-up.
- Top-down: the top-level block is defined and then the sub-blocks necessary to build the top-level block are identified.(Fig.4-9 binary adder)
- Bottom-up: the building blocks are first identified and then combined to build the top-level block.(Example 4-2 4-bit adder)

Three state gates

Gates statement: gate name(output, input, control)

>> bufif1(OUT, A, control);

A = OUT when control = 1, OUT = z when control = 0;

>> notif0(Y, B, enable);

Y = B' when enable = 0, Y = z when enable = 1;

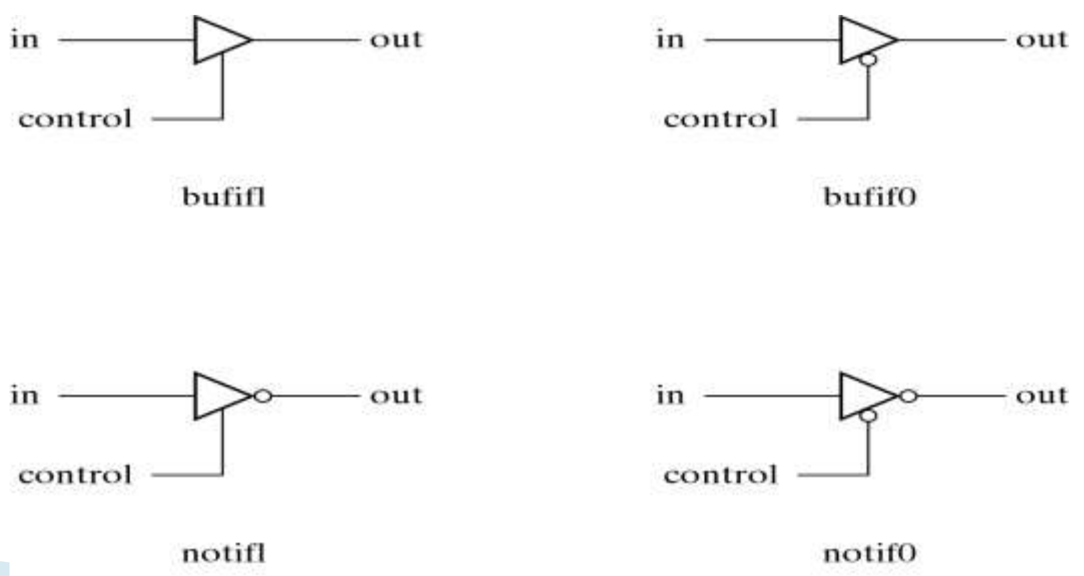


Fig. 4-31 Three-State Gates

2-to-1 multiplexer

- HDL uses the keyword tri to indicate that the output has multiple drivers.

```
module muxtri (A, B, select, OUT);
  input A,B,select;
  output OUT;
  tri OUT;
  bufif1 (OUT,A,select);
  bufif0 (OUT,B,select);
endmodule
```

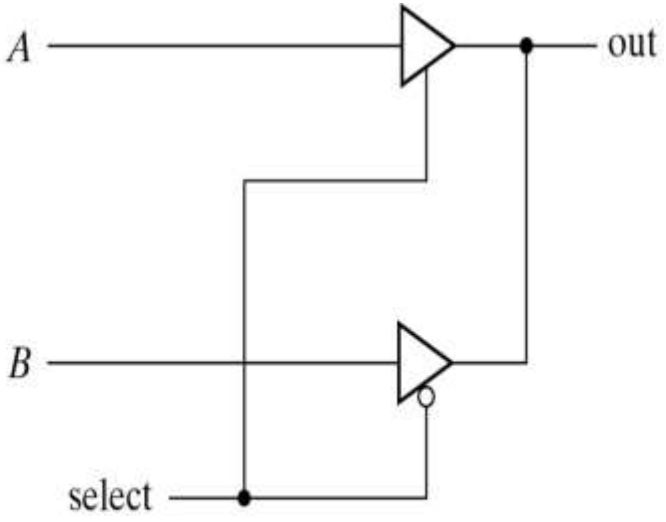


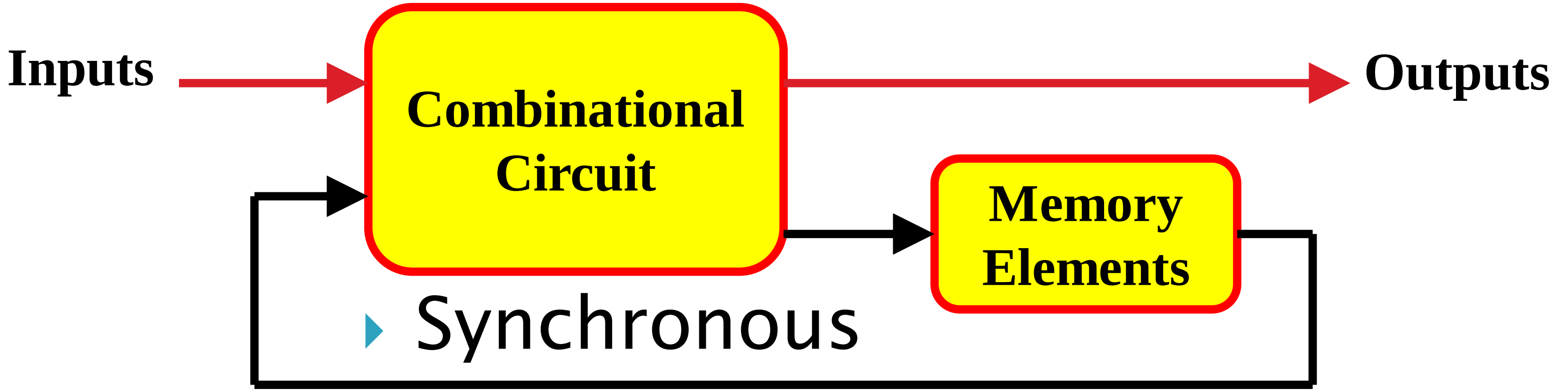
Fig. 4-32 2-to-1-Line Multiplexer with Three-State Buffers

IV UNIT

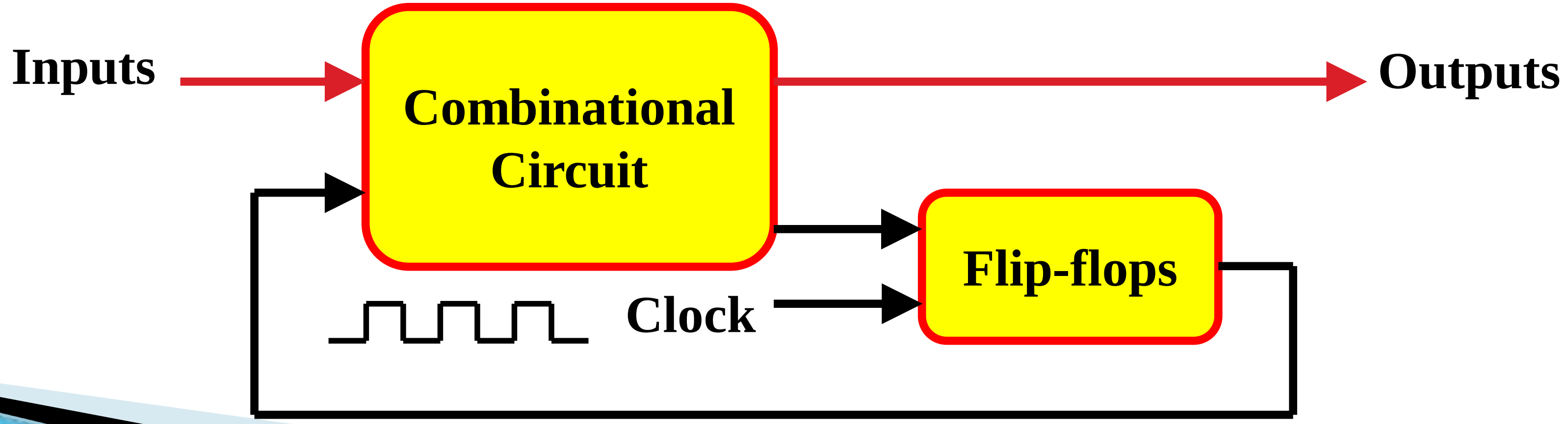
Sequential Circuits

Sequential Circuits

- Asynchronous



- Synchronous

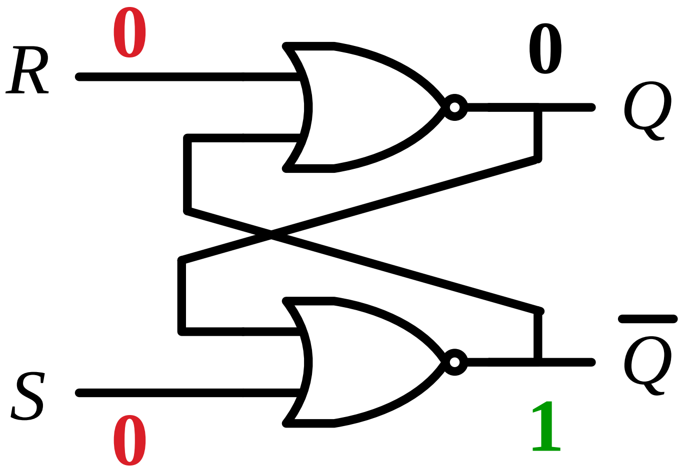


FF vs. Latch

- *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, 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.

Latches

- *SR* Latch



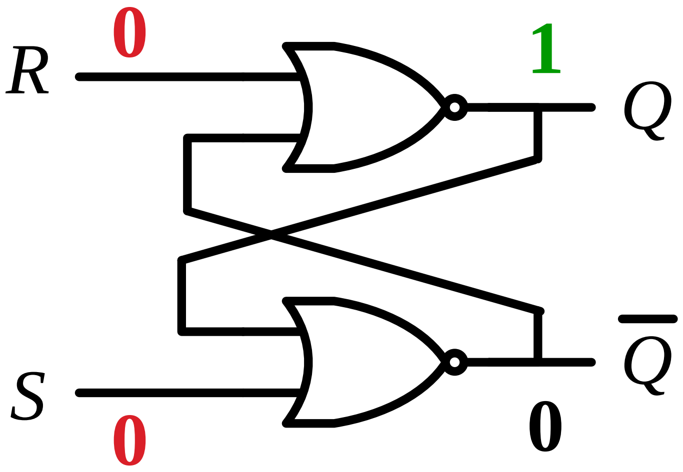
S R Q_0	Q	Q'
0 0 0	0	1

$\boldsymbol{Q} = \boldsymbol{Q_0}$

Initial Value

Latches

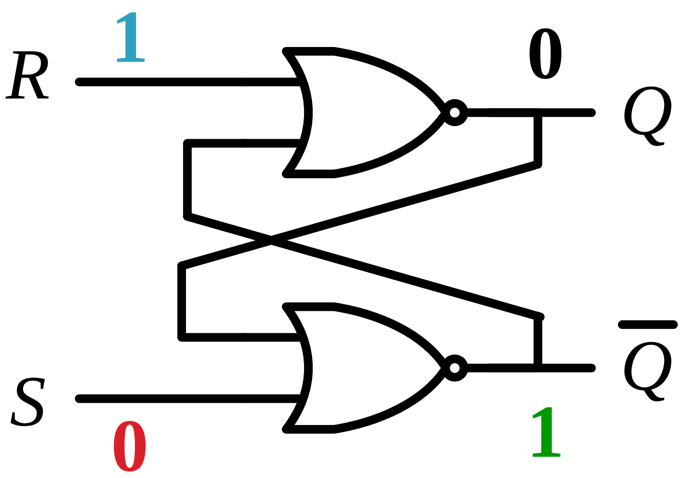
- *SR* Latch



S R Q_0	Q	Q'	
0 0 0	0	1	$Q = Q_0$
0 0 1	1	0	$Q = Q_0$

Latches

- *SR* Latch



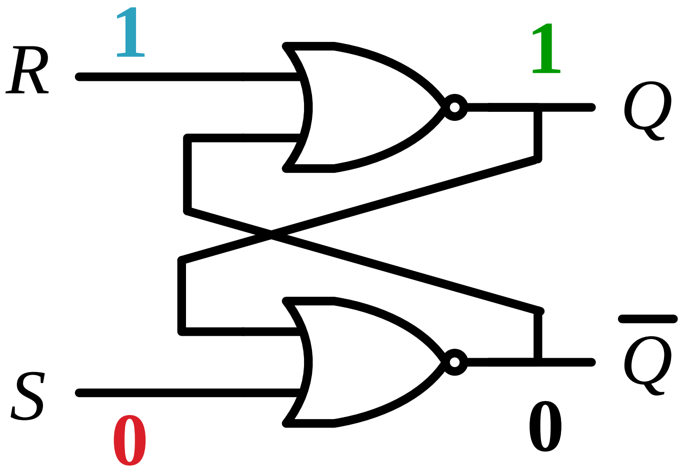
S R Q_0	Q	Q'
0 0 0	0	1
0 0 1	1	0
0 1 0	0	1

$Q = Q_0$

$Q = 0$

Latches

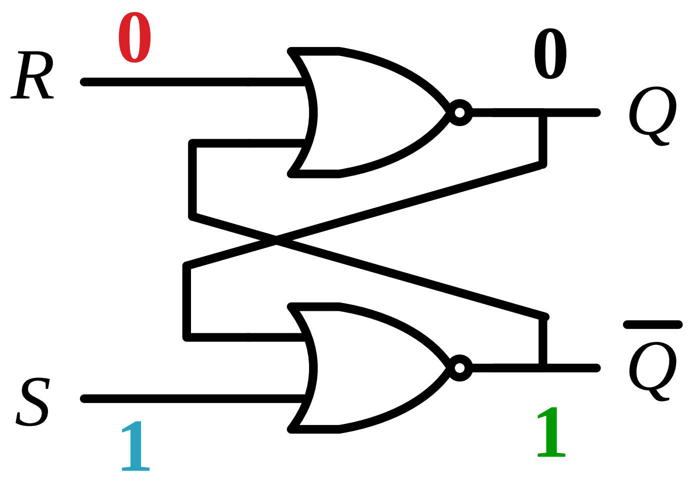
- *SR* Latch



S	R	Q_0	Q	Q'	
0	0	0	0	1	$Q = Q_0$
0	0	1	1	0	
0	1	0	0	1	$Q = 0$
0	1	1	0	1	$Q = 0$

Latches

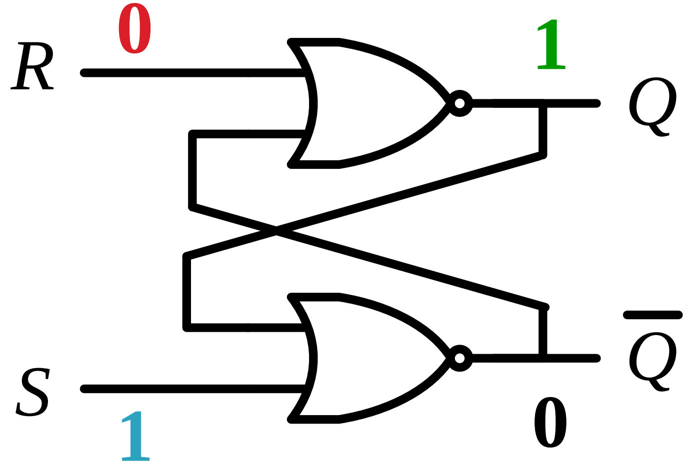
- *SR* Latch



S R Q_0	Q	Q'	
0 0 0	0	1	$Q = Q_0$
0 0 1	1	0	
0 1 0	0	1	$Q = 0$
0 1 1	0	1	
1 0 0	1	0	$Q = 1$

Latches

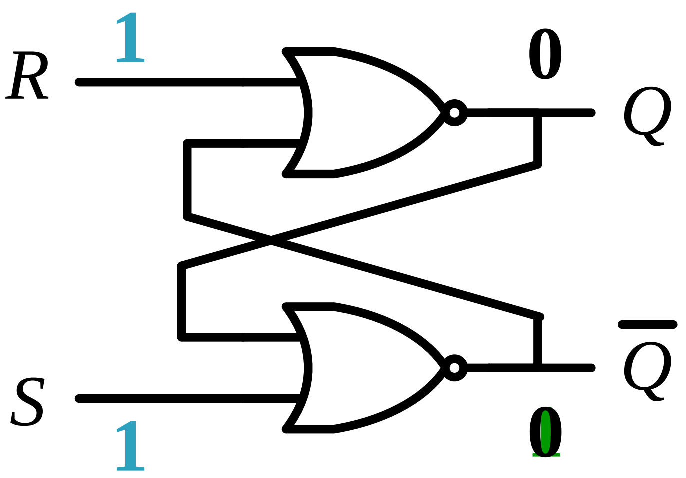
- *SR* Latch



S R Q_0	Q	Q'	
0 0 0	0	1	$Q = Q_0$
0 0 1	1	0	
0 1 0	0	1	$Q = 0$
0 1 1	0	1	
1 0 0	1	0	$Q = 1$
1 0 1	1	0	$Q = 1$

Latches

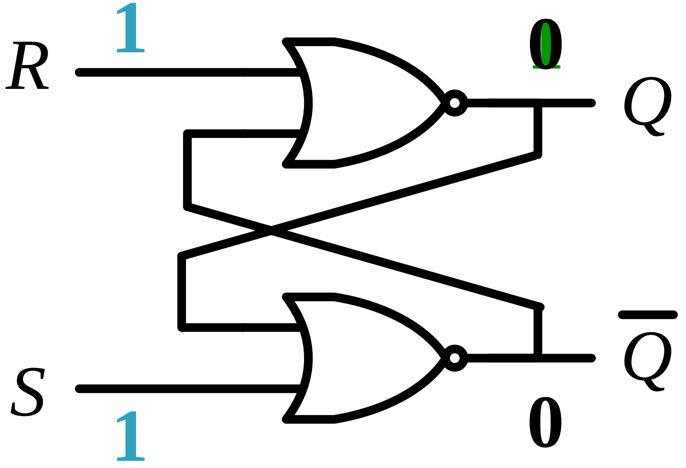
- *SR* Latch



S R Q_0	Q	Q'	
0 0 0	0	1	$Q = Q_0$
0 0 1	1	0	
0 1 0	0	1	$Q = 0$
0 1 1	0	1	
1 0 0	1	0	$Q = 1$
1 0 1	1	0	
1 1 0	0	0	$Q = Q'$

Latches

- *SR* Latch



S R Q_0	Q	Q'
0 0 0	0	1
0 0 1	1	0
0 1 0	0	1
0 1 1	0	1
1 0 0	1	0
1 0 1	1	0
1 1 0	0	0
1 1 1	0	0

$Q = Q_0$

$Q = 0$

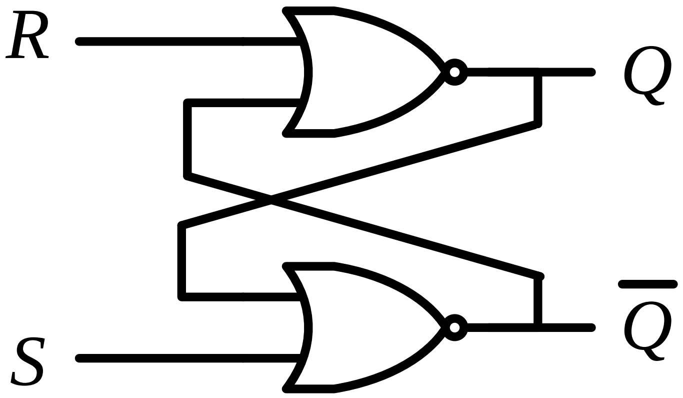
$Q = 1$

$Q = Q'$

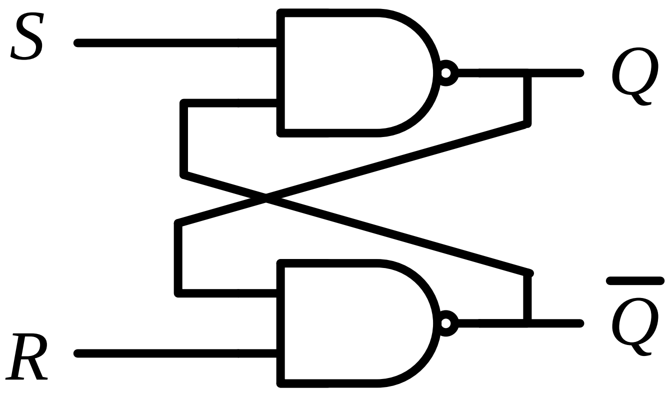
$Q = Q'$

Latches

- *SR* Latch



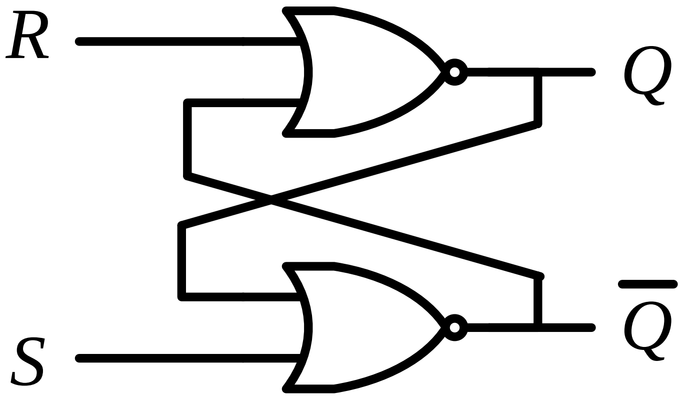
S	R	Q	
0	0	Q_0	**No change**
0	1	0	**Reset**
1	0	1	**Set**
1	1	Q=Q'=0	**Invalid**



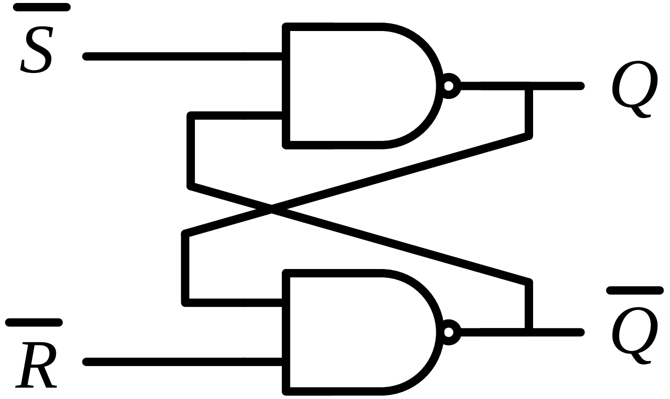
S	R	Q	
0	0	Q=Q'=1	**Invalid**
0	1	1	**Set**
1	0	0	**Reset**
1	1	Q_0	**No change**

Latches

- *SR* Latch



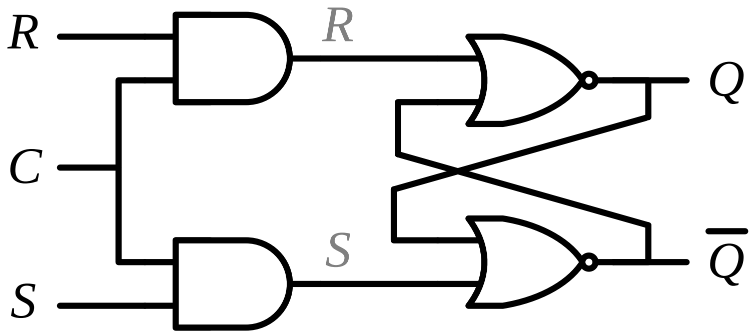
S R	Q	
0 0	Q_0	**No change**
0 1	0	**Reset**
1 0	1	**Set**
1 1	Q=Q'=0	**Invalid**

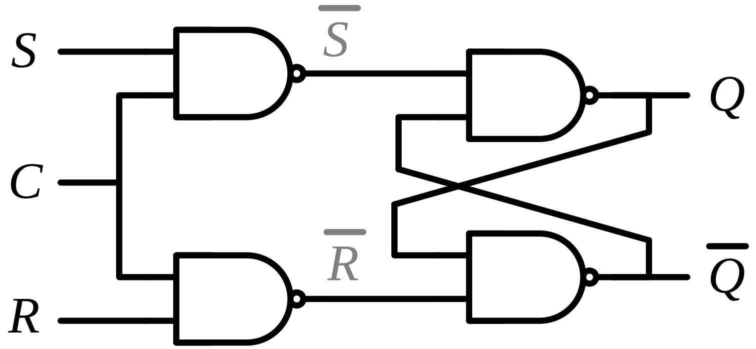


S' R'	Q	
0 0	Q=Q'=1	**Invalid**
0 1	1	**Set**
1 0	0	**Reset**
1 1	Q_0	**No change**

Controlled Latches

- *SR* Latch with Control Input

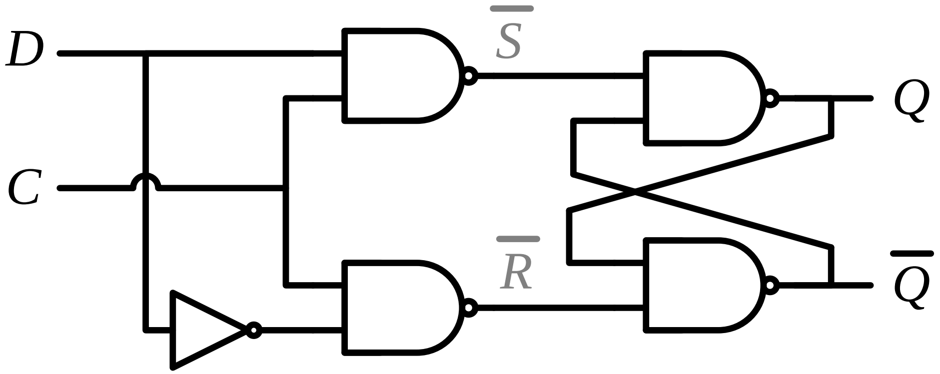




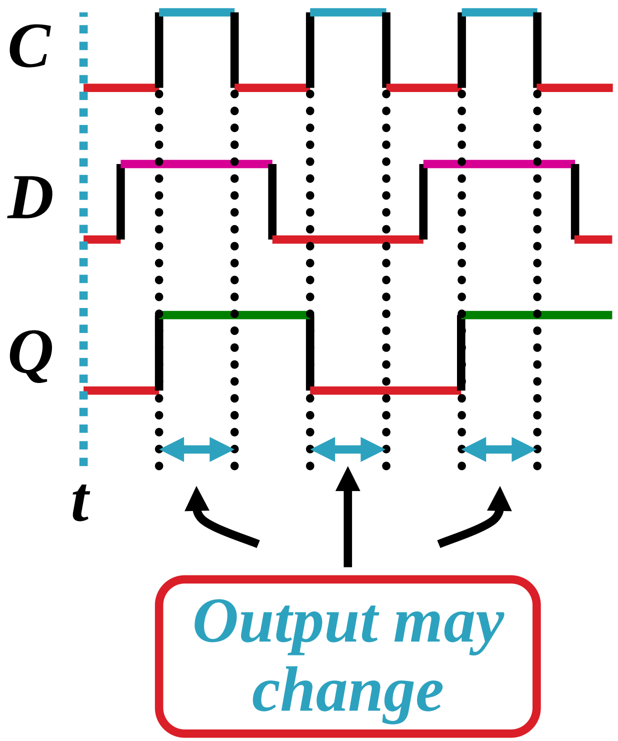
C S R	Q	
0 x x	Q_0	**No change**
1 0 0	Q_0	**No change**
1 0 1	0	**Reset**
1 1 0	1	**Set**
1 1 1	$Q=Q'$	**Invalid**

Controlled Latches

- *D* Latch (*D* = *Data*)



Timing Diagram



C D	Q
0 x	Q_0
1 0	0
1 1	1

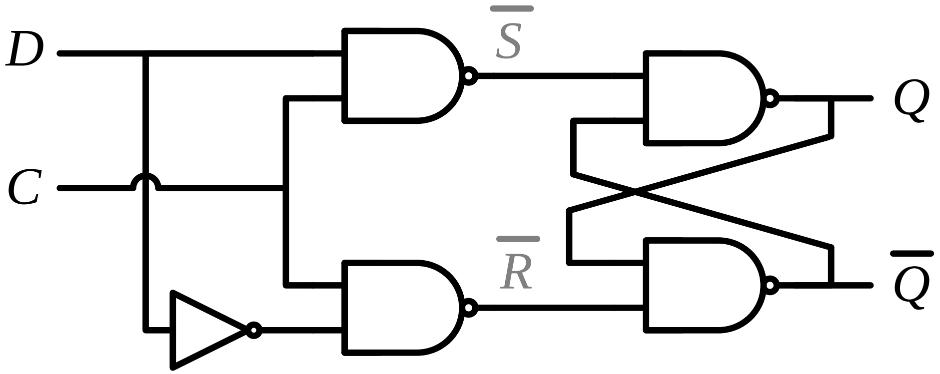
No change

Reset

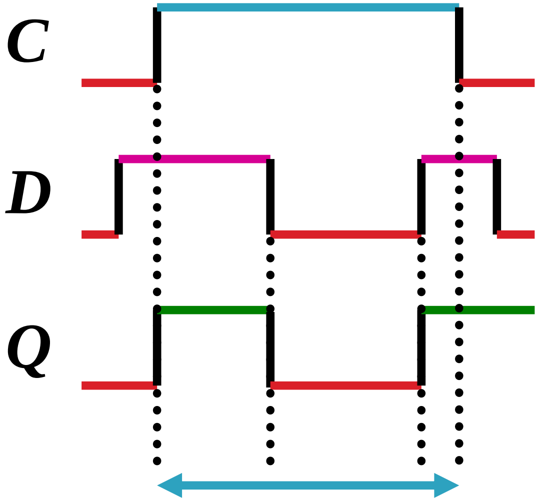
Set

Controlled Latches

- *D* Latch (*D* = *Data*)



Timing Diagram

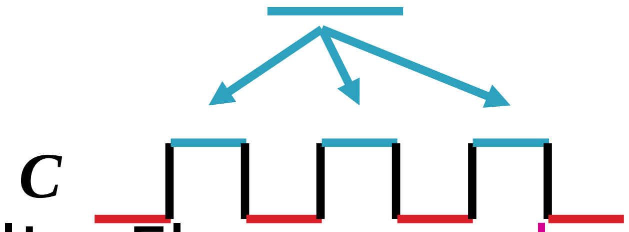


Output may change

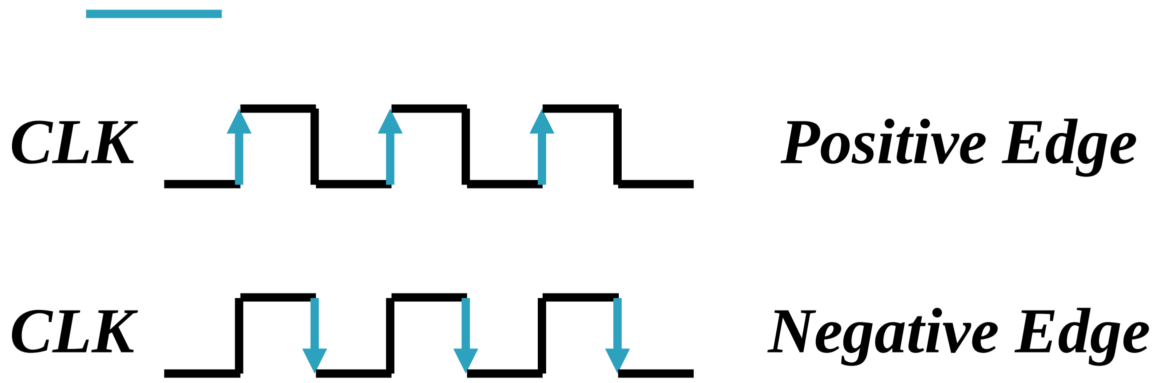
C D	Q	
0 x	Q_0	**No change**
1 0	0	**Reset**
1 1	1	**Set**

Flip-Flops

- Controlled latches are level-triggered

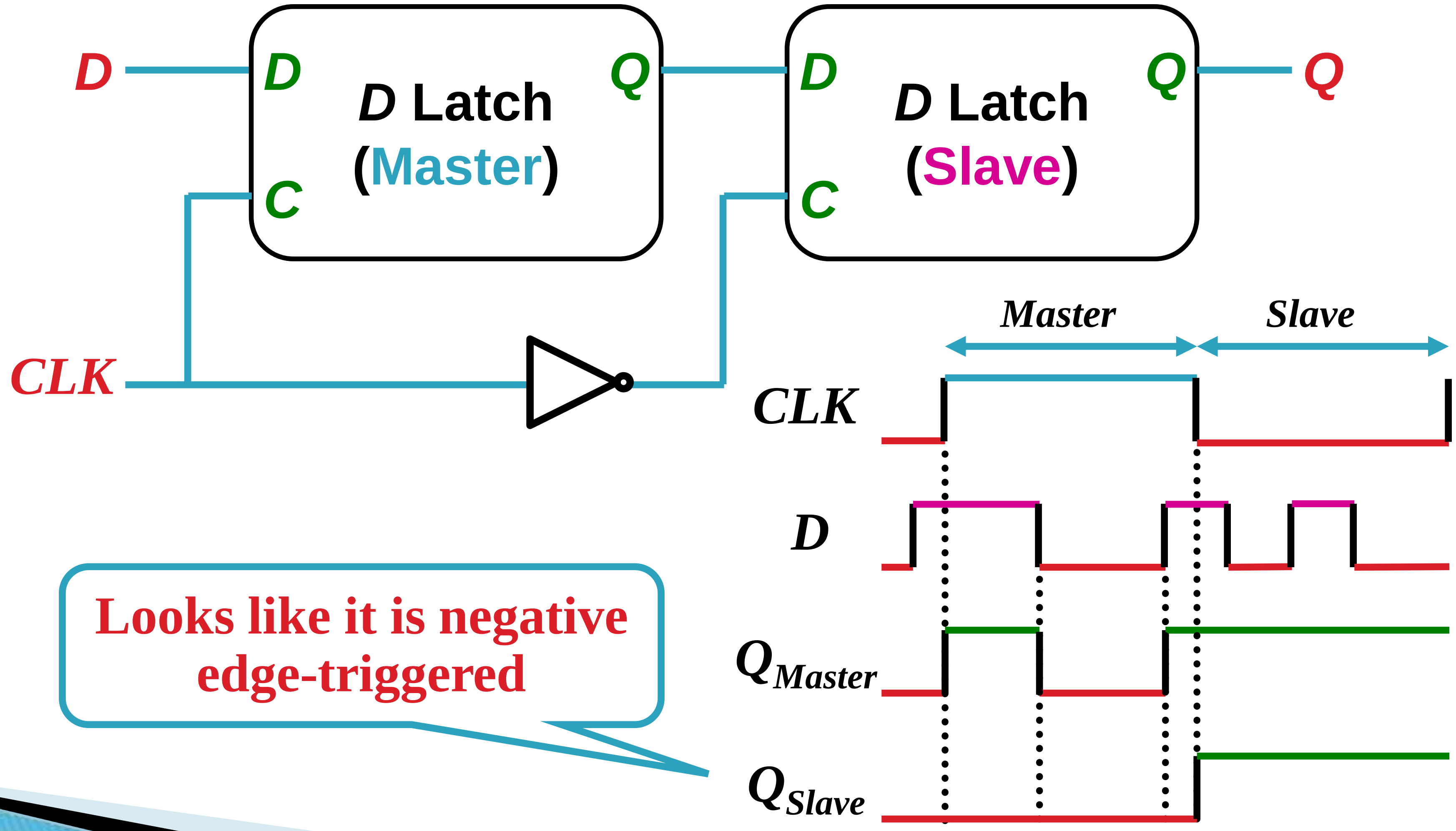


- Flip-Flops are edge-triggered



Flip-Flops

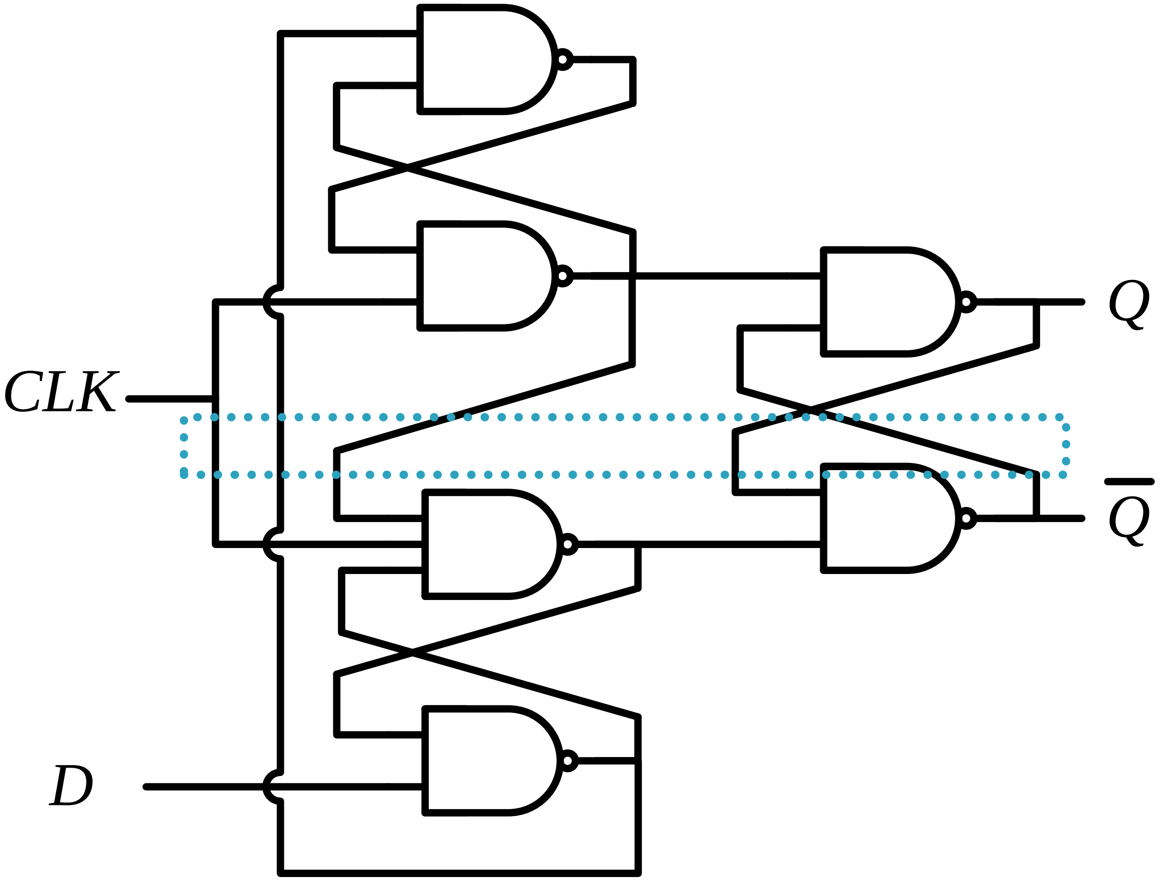
- Master-Slave *D* Flip-Flop

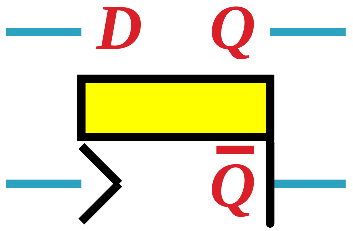


Looks like it is negative edge-triggered

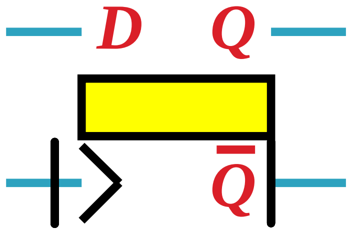
Flip-Flops

- Edge-Triggered *D* Flip-Flop





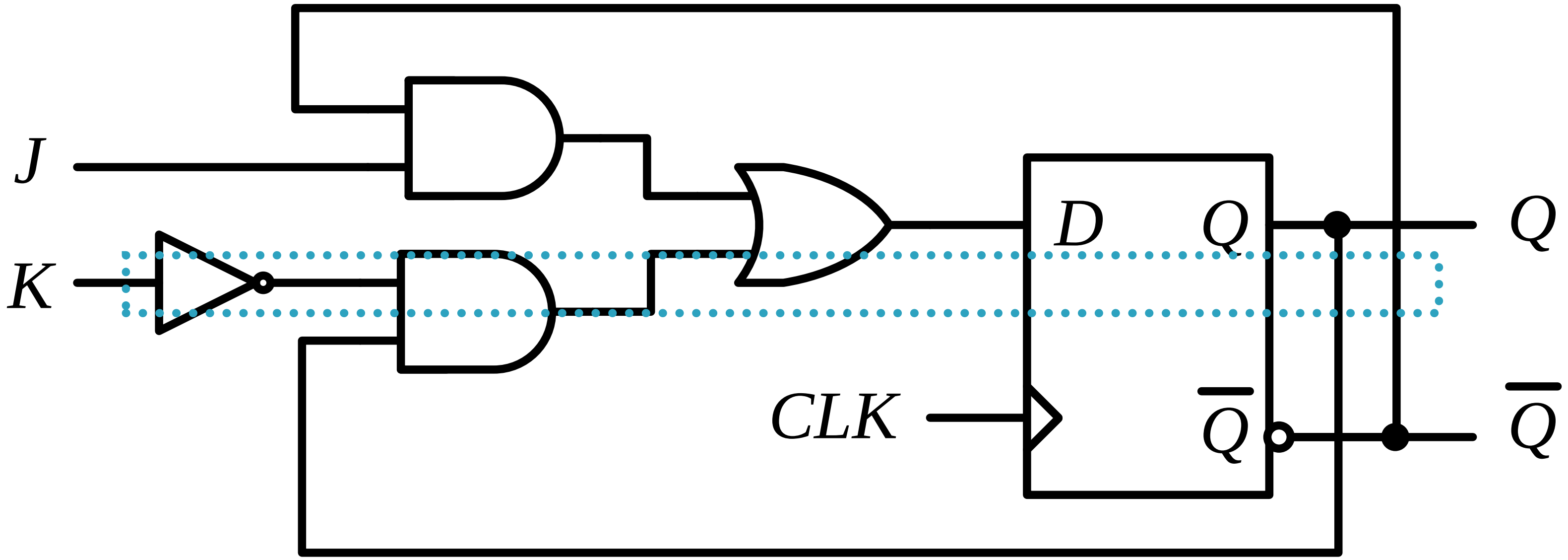
Positive Edge



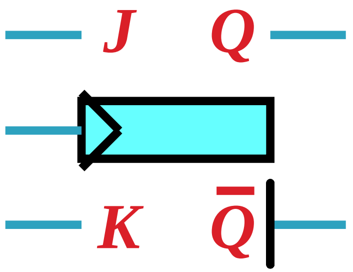
Negative Edge

Flip–Flops

- *JK* Flip–Flop

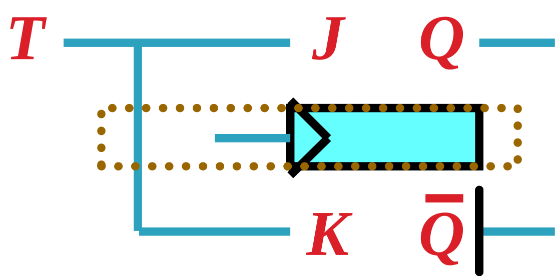


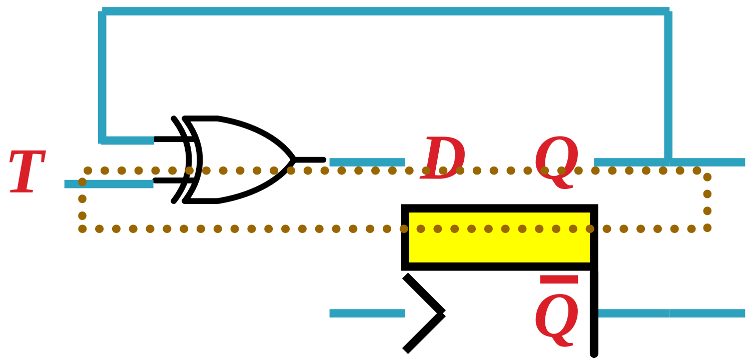
$$D = JQ' + K'Q$$



Flip-Flops

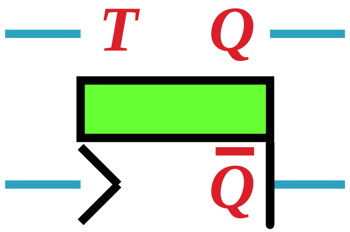
- *T* Flip-Flop



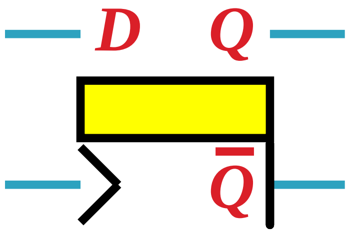


$$D = JQ' + K'Q$$

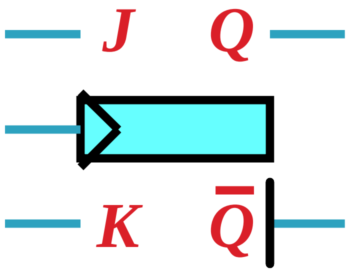
$$D = TQ' + T'Q = T \oplus Q$$



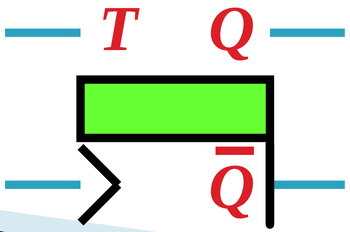
Flip-Flop Characteristic Tables



D	*Q*(*t*+1)	
0	0	**Reset**
1	1	**Set**

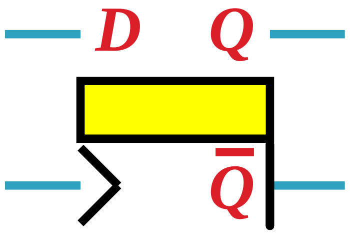


J	*K*	*Q*(*t*+1)	
0	0	*Q*(*t*)	**No change**
0	1	0	**Reset**
1	0	1	**Set**
1	1	*Q'*(*t*)	**Toggle**



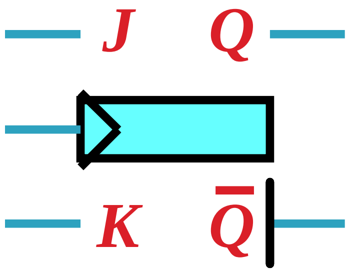
T	*Q*(*t*+1)	
0	*Q*(*t*)	**No change**
1	*Q'*(*t*)	**Toggle**

Flip-Flop Characteristic Equations



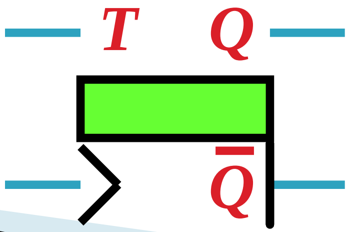
D	Q(*t*+1)
0	0
1	1

$$Q(t+1) = D$$



J	K	Q(*t*+1)
0	0	Q(*t*)
0	1	0
1	0	1
1	1	Q'(*t*)

$$Q(t+1) = JQ' + K'Q$$

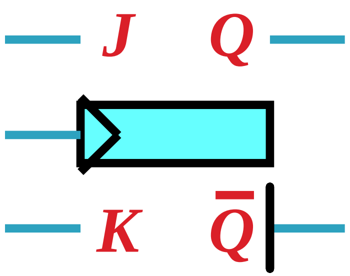


T	Q(*t*+1)
0	Q(*t*)
1	Q'(*t*)

$$Q(t+1) = T \oplus Q$$

Flip-Flop Characteristic Equations

- Analysis / Derivation

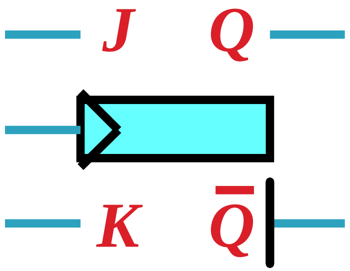


J	K	Q(t)	Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

No change

Flip-Flop Characteristic Equations

- Analysis / Derivation



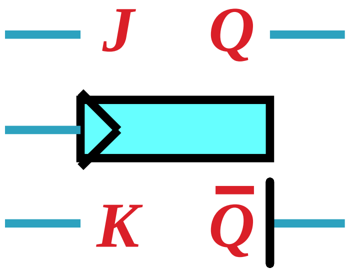
J	K	$Q(t)$	$Q(t+1)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	
1	0	1	
1	1	0	
1	1	1	

No change

Reset

Flip-Flop Characteristic Equations

- Analysis / Derivation



J	K	Q(t)	Q(t+1)
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	

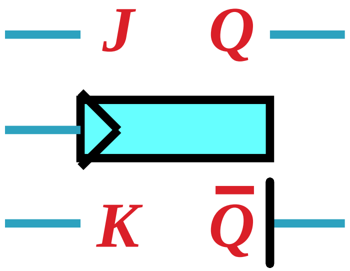
No change

Reset

Set

Flip-Flop Characteristic Equations

- Analysis / Derivation



J	K	$Q(t)$	$Q(t+1)$
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	0

No change

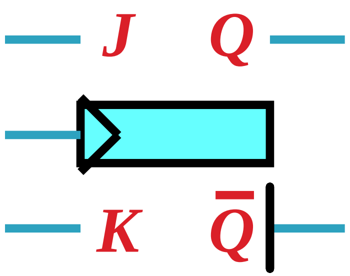
Reset

Set

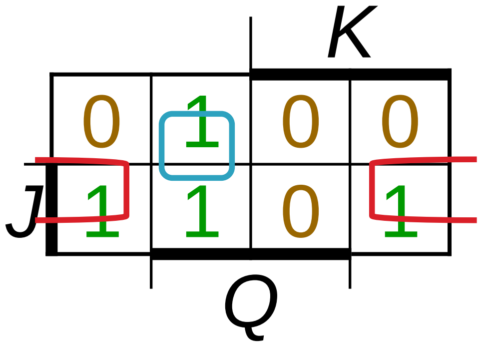
Toggle

Flip-Flop Characteristic Equations

- Analysis / Derivation



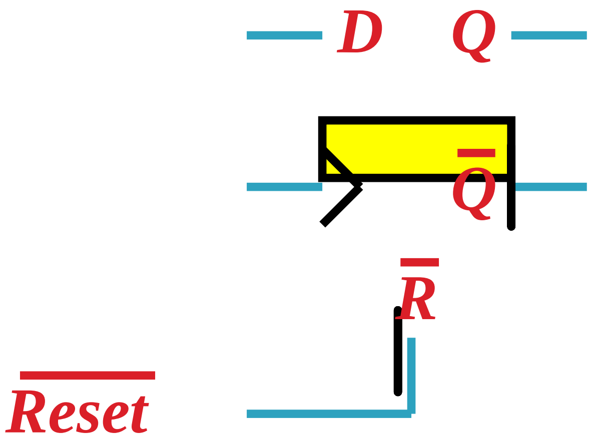
J	K	$Q(t)$	$Q(t+1)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



$$Q(t+1) = JQ' + K'Q$$

Flip-Flops with Direct Inputs

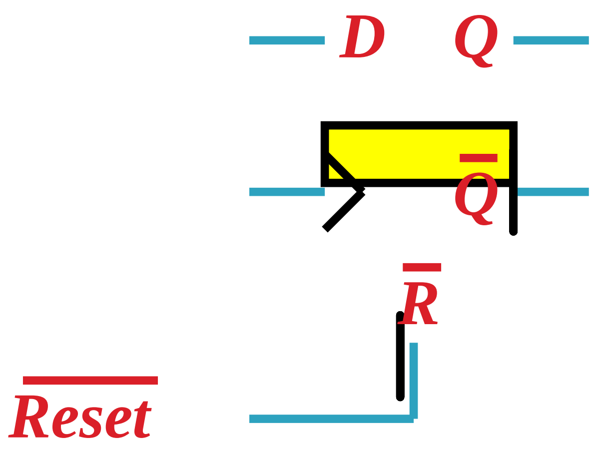
- Asynchronous Reset



R' D CLK	Q(t+1)
0 x x	0

Flip-Flops with Direct Inputs

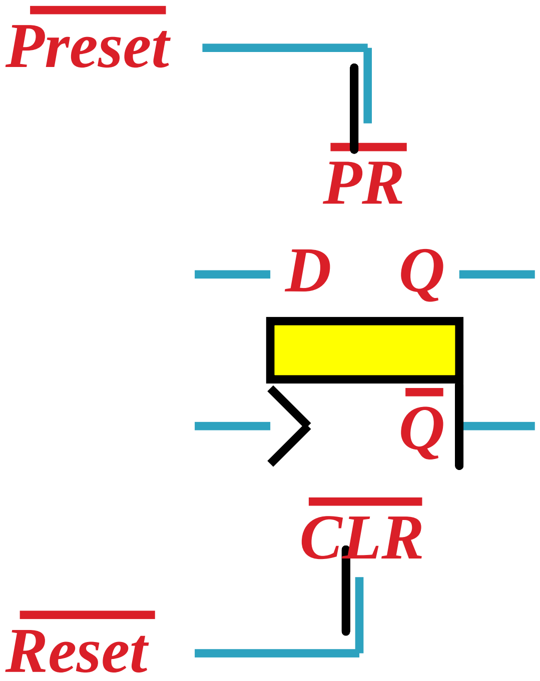
- Asynchronous Reset



R'	D	CLK	Q(t+1)
0	x	x	0
1	0	↑	0
1	1	↑	1

Flip-Flops with Direct Inputs

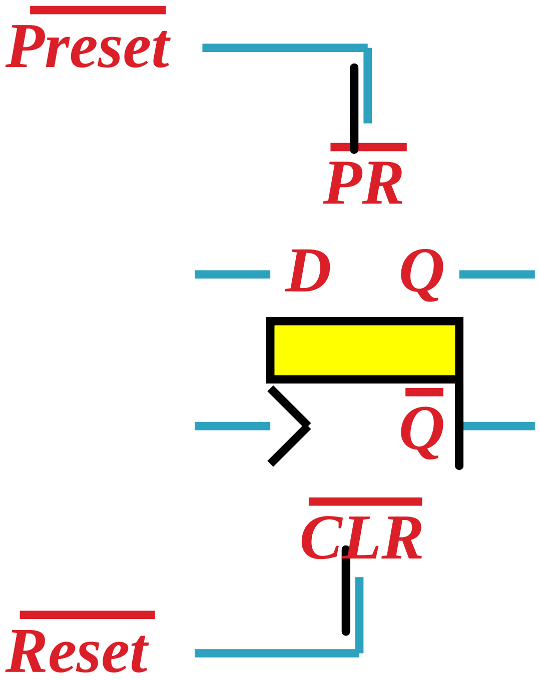
- Asynchronous Preset and Clear



PR'	CLR'	D	CLK	Q(t+1)
1	0	x	x	0

Flip-Flops with Direct Inputs

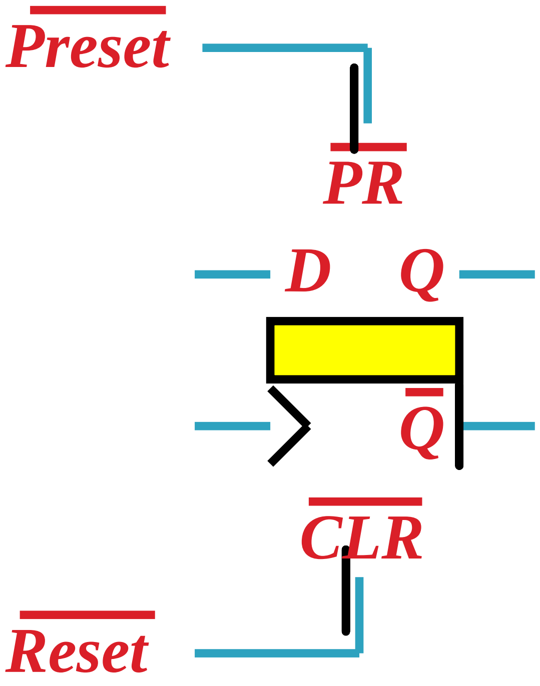
- Asynchronous Preset and Clear



PR’	CLR’	D	CLK	Q(t+1)
1	0	x	x	0
0	1	x	x	1

Flip-Flops with Direct Inputs

- Asynchronous Preset and Clear



PR’	CLR’	D	CLK	Q(t+1)
1	0	x	x	0
0	1	x	x	1
1	1	0	↑	0
1	1	1	↑	1

Mealy and Moore Models

- The Mealy model: the outputs are functions of both the present state and inputs (Fig. 5-15).
  - The outputs may change if the inputs change during the clock pulse period.
    - The outputs may have momentary false values unless the inputs are synchronized with the clocks.
- The Moore model: the outputs are functions of the present state only (Fig. 5-20).
  - The outputs are <u>synchronous</u> with the clocks.

Mealy and Moore Models

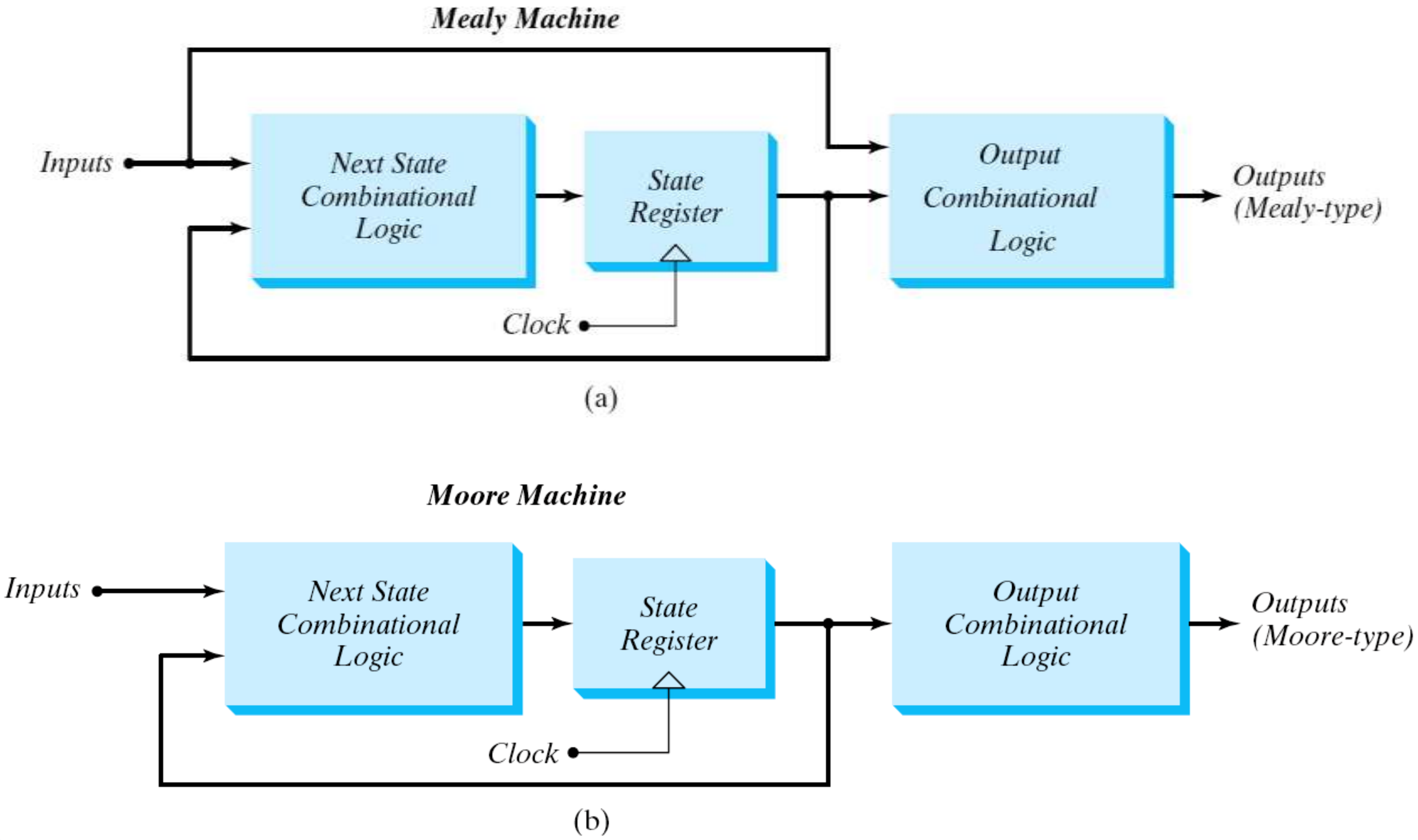


Fig. 5.21 Block diagram of Mealy and Moore state machine

State Reduction and Assignment

- State Reduction
  Reductions on the number of flip-flops and the number of gates.
  - A reduction in the number of states may result in a reduction in the number of flip-flops.
  - An example state diagram showing in Fig. 5.25.

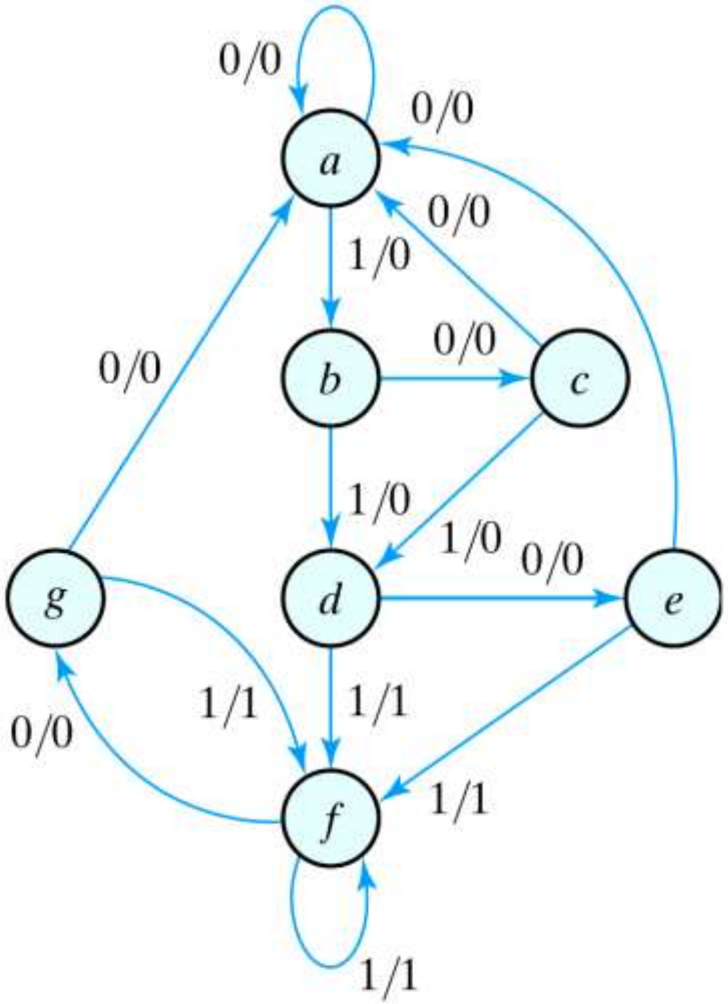


Fig. 5.25 State diagram

State Reduction

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	

- Only the input–output sequences are important.
- Two circuits are equivalent
  - Have identical outputs for all input sequences;
  - The number of states is not important.

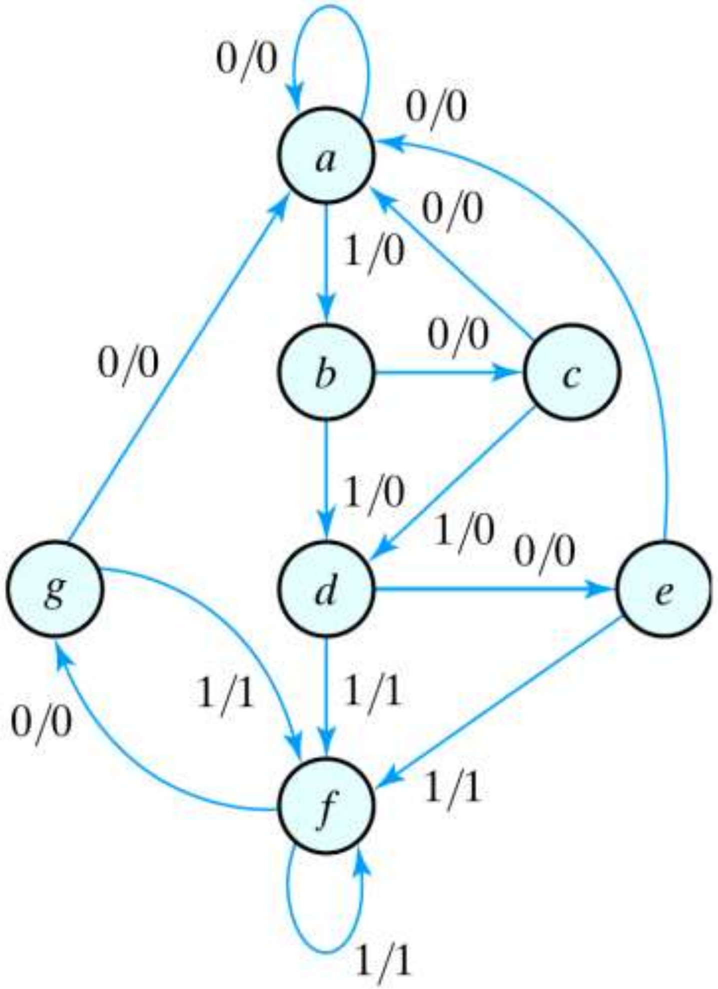
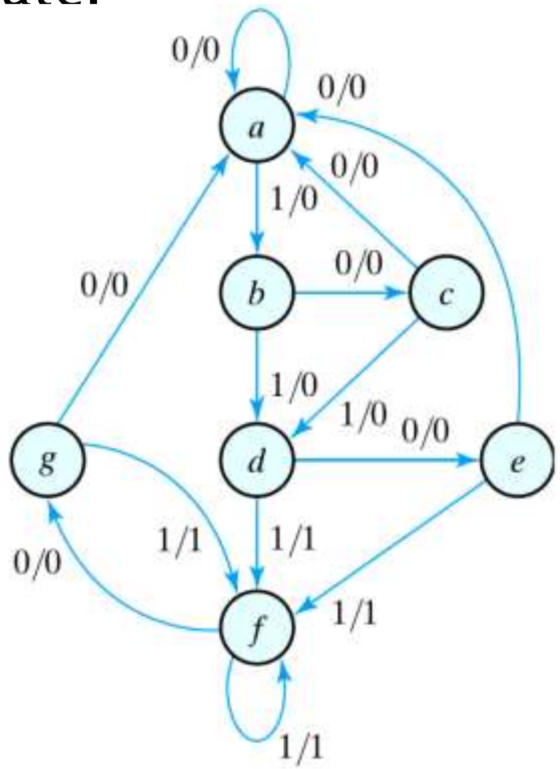


Fig. 5.25 State diagram

- Equivalent states
  - Two states are said to be equivalent
    - For each member of the set of inputs, they give exactly the same output and send the circuit to the same state or to an equivalent state.
    - One of them can be r

Table 5.6
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	g	f	0	1
g	a	f	0	1



- Reducing the state table
  - $e = g$ (remove g);
  - $d = f$ (remove f);

Table 5.7
Reducing the State Table

Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
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

◦ The reduced finite state machine

Table 5.8
Reduced State Table

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
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

State:	a	a	b	c	d	e	d	d	e	d	e	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 checking of each pair of states for possible equivalence can be done systematically using Implication Table.
- The unused states are treated as don't-care condition $\Rightarrow$ fewer combinational gates.

Table 5.8
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

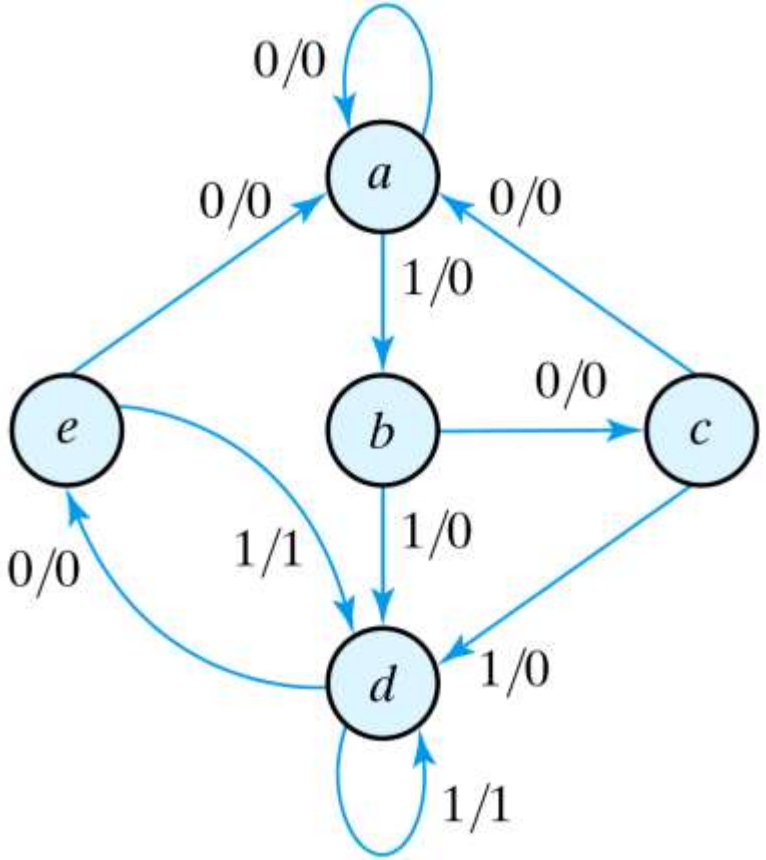


Fig. 5.26 Reduced State diagram

Implication Table

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

State Assignment

- State Assignment
- To minimize the cost of the combinational circuits.
  - Three possible binary state assignments. (m states need n–bits, where $2^n > m$)

Table 5.9
Three Possible Binary State 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

- Any binary number assignment is satisfactory as long as each state is assigned a unique number.
- Use binary assignment 1.

Table 5.10
Reduced State Table with Binary Assignment 1

Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
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

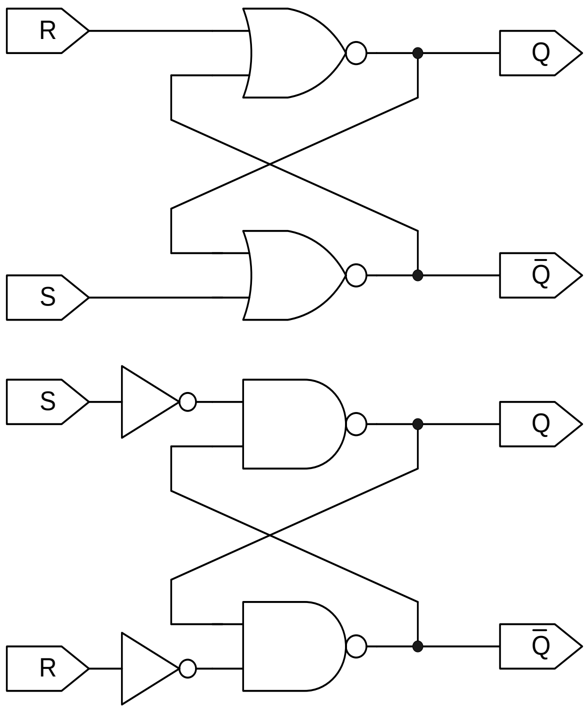
Sequential Digital Circuits

- Sequential circuits are digital circuits in which the outputs depend not only on the current inputs, but also on the previous state of the output.
- They basic sequential circuit elements can be divided in two categories:
- Level-sensitive (Latches)
  - High-level sensitive
  - Low-level sensitive
- Edge-triggered (Flip-flops)
  - Rising (positive) edge triggered
  - Falling (negative) edge triggered
  - Dual-edge triggered

The Set/Reset (SR) Latch

The Set/Reset latch is the most basic unit of sequential digital circuits. It has two inputs (S and R) and two outputs outputs Q and Q’. The two outputs must always be complementary, i.e if Q is 0 then Q’ must be 1, and vice-versa. The S input sets the Q output to a logic 1. The R input resets the Q output to a logic 0.

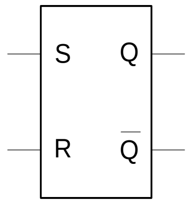
Circuit Diagram



Truth Table

S	R	Q+	Q´+	Function
0	0	Q	Q´	Latch
0	1	0	1	Reset
1	0	1	0	Set
1	1	0	0	Illegal

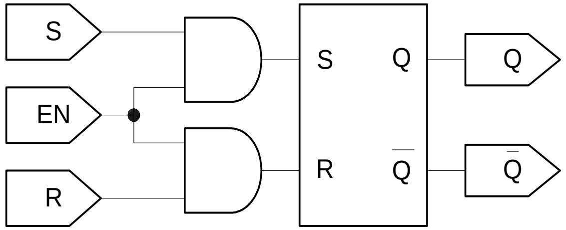
Logic Symbol



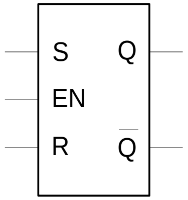
The Gated Set/Reset (SR) Latch

To be able to control when the S and R inputs of the SR latch can be applied to the latch and thus change the outputs, an extra input is used. This input is called the Enable. If the Enable is 0 then the S and R inputs have no effect on the outputs of the SR latch. If the Enable is 1 then the Gated SR latch behaves as a normal SR latch.

Circuit Diagram



Logic Symbol



Truth Table

EN	S	R	Q+
0	0	0	Q
0	0	1	Q
0	1	0	Q
0	1	1	Q
1	0	0	Q
1	0	1	0
1	1	0	1
1	1	1	U

Truth Table

EN	S	R	Q+	Function
0	X	X		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

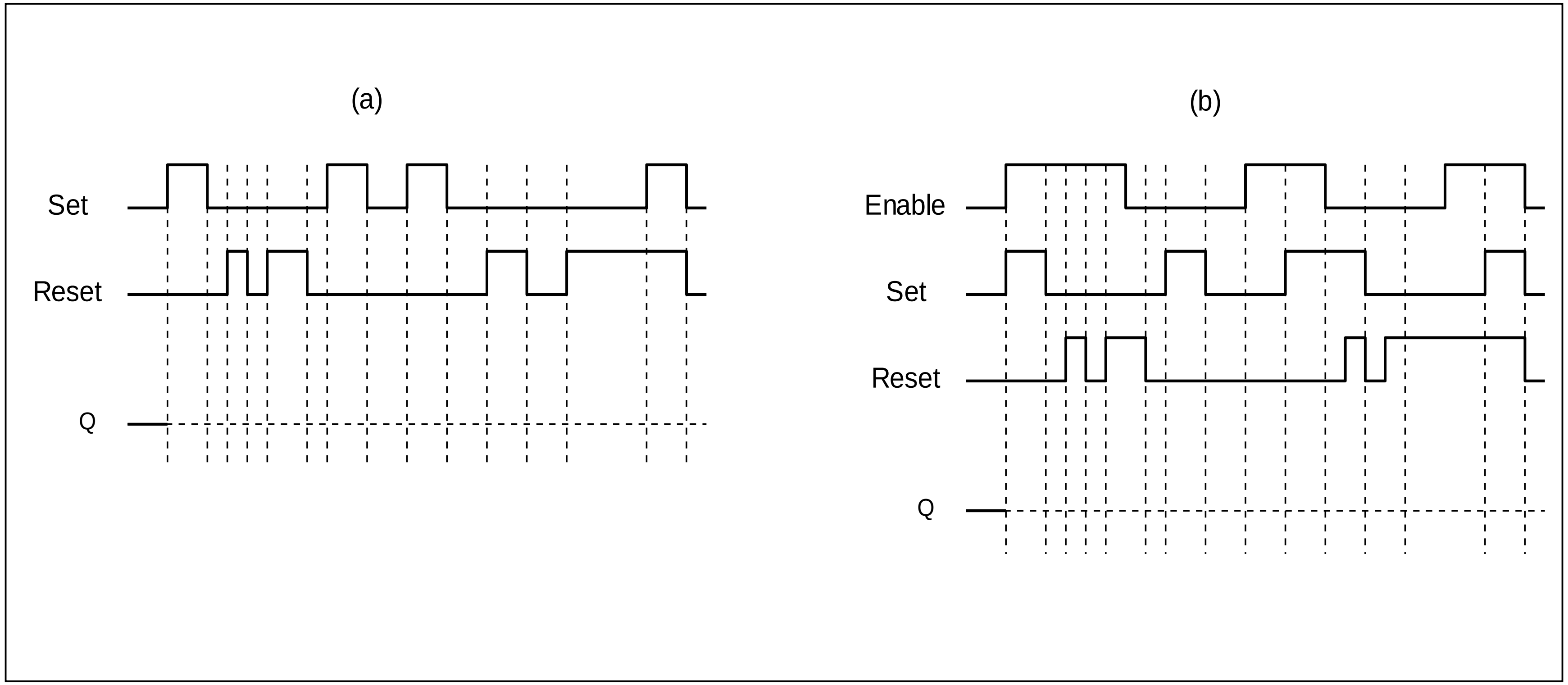
SR Latch :- Example

Complete the timing diagrams for :

(a) Simple SR Latch

(b) SR Latch with Enable input.

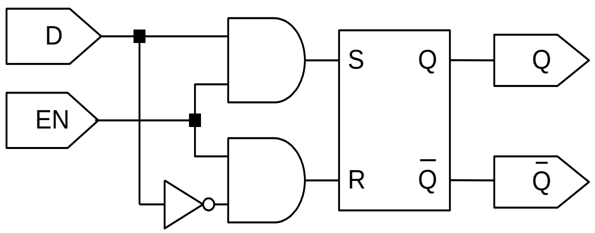
Assume that for both cases the Q output is initially at logic zero.



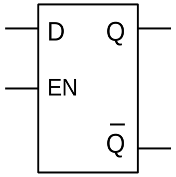
The Data (D) Latch

A problem with the SR latch is that the S and R inputs can not be at logic 1 at the same time. To ensure that this can not happen, the S and R inputs can by connected through an inverter. In this case the Q output is always the same as the input, and the latch is called the Data or D latch. The D latch is used in Registers and memory devices.

Circuit Diagram



Logic Symbol



Truth Table

EN	D	Q	Q+
0	0	0	Q
0	0	1	Q
0	1	0	Q
0	1	1	Q
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

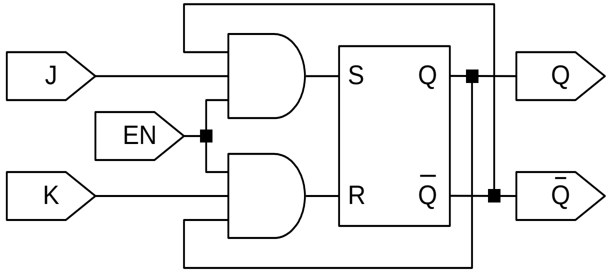
Truth Table

EN	D	Q+	Function
0	0		
0	1		
1	0		
1	1		

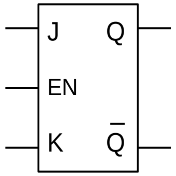
The JK Latch

Another way to ensure that the S and R inputs can not be at logic 1 simultaneously, is to cross connect the Q and Q' outputs with the S and R inputs through AND gates. The latch obtained is called the JK latch. In the J and K inputs are both 1 then the Q output will change state (Toggle) for as long as the Enable 1, thus the output will be unstable. This problem is avoided by ensuring that the Enable is at logic 1 only for a very short time, using edge detection circuits.

Circuit Diagram



Logic Symbol



Truth Table

EN	J	K	Q	Q+
0	X	X	X	Q
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Truth Table

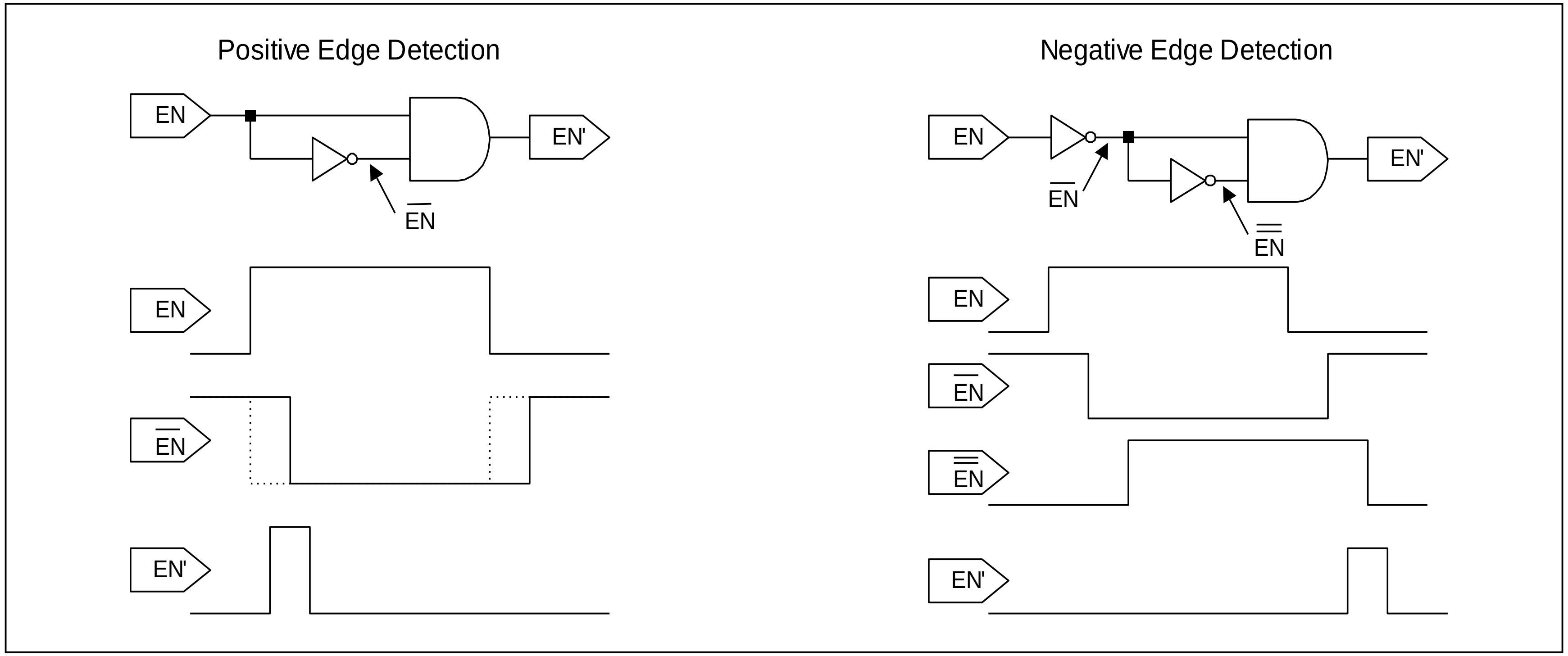
EN	J	K	Q+	Function
0	X	X		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Latches and Flip-Flops

- **Latches** are also called **transparent** or **level triggered** flip flops, because the change on the outputs will follow the changes of the inputs as long as the Enable input is set.
- **Edge triggered** flip flops are the flip flops that change there outputs only at the transition of the Enable input. The enable is called the Clock input.

Edge Detection Circuits

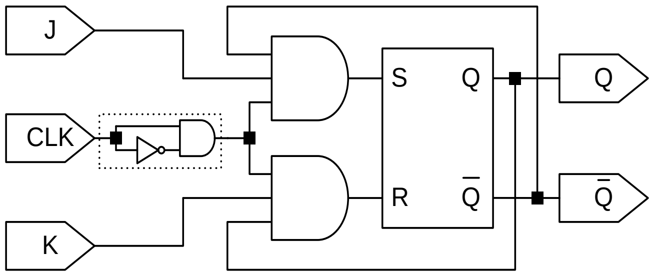
Edge detection circuits are used to detect the transition of the Enable from logic 0 to logic 1 (positive edge) or from logic 1 to logic 0 (negative edge). The operation of the edge detection circuits shown below is based on the fact that there is a time delay between the change of the input of a gate and the change at the output. This delay is in the order of a few nanoseconds. The Enable in this case is called the Clock (CLK)



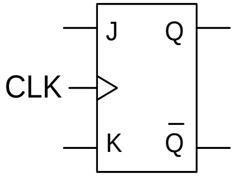
The JK Edge Triggered Flip Flop

The JK edge triggered flip flop can be obtained by inserting an edge detection circuit at the Enable (CLK) input of a JK latch. This ensures that the outputs of the flip flop will change only when the CLK changes (0 to 1 for +ve edge or 1 to 0 for –ve edge)

Positive Edge JK Flip Flop

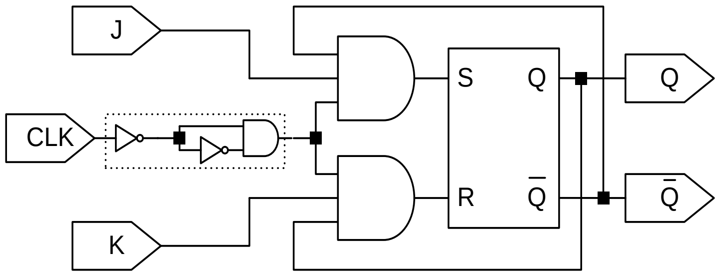


Logic Symbol

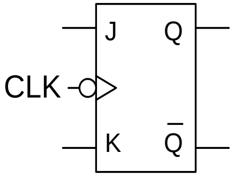


CLK	J	K	Q_{N+1}	Function
‾|_	X	X	Q	
↑	0	0	Q	
↑	0	1	0	
↑	1	0	1	
↑	1	1	Q’	

Negative Edge JK Flip Flop



Logic Symbol

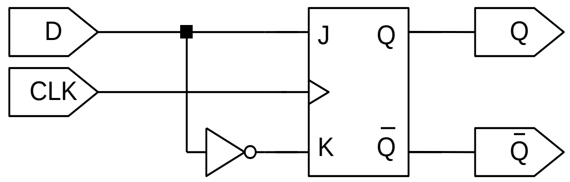


CLK	J	K	Q_{N+1}	Function
_|‾	X	X		
↓	0	0		
↓	0	1		
↓	1	0		
↓	1	1		

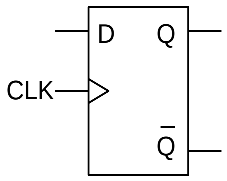
The D Edge Triggered Flip Flop

The D edge triggered flip flop can be obtained by connecting the J with the K inputs of a JK flip through an inverter as shown below. The D edge trigger can also be obtained by connecting the S with the R inputs of a SR edge triggered flip flop through an inverter.

Positive Edge D Flip Flop

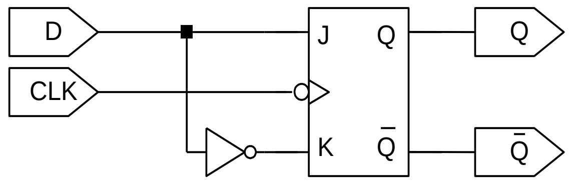


Logic Symbol

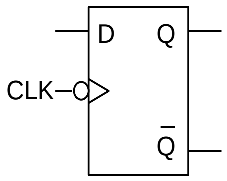


CLK	D	Q_{N+1}	Function
⎤⎣	X	Q	
↑	0	0	
↑	1	1	

Negative Edge D Flip Flop



Logic Symbol

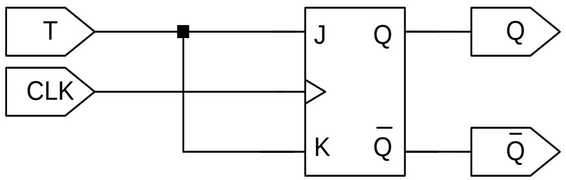


CLK	D	Q_{N+1}	Function
⎦⎡	X	Q	
↓	0	0	
↓	1	1	

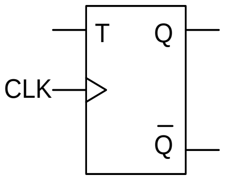
The Toggle (T) Edge Triggered Flip Flop

The T edge triggered flip flop can be obtained by connecting the J with the K inputs of a JK flip directly. When T is zero then both J and K are zero and the Q output does not change. When T is one then both J and K are one and the Q output will change to the opposite state, or toggle.

Positive Edge T Flip Flop

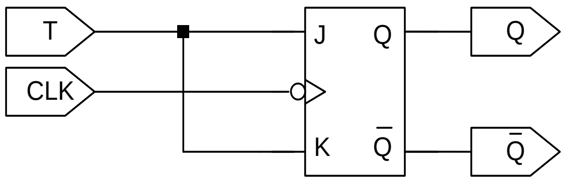


Logic Symbol

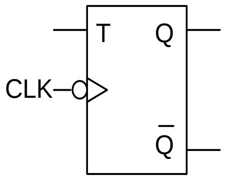


CLK	T	Q_{N+1}	Function
	X	Q	
↑	0	Q	
↑	1	Q´	

Negative Edge T Flip Flop



Logic Symbol

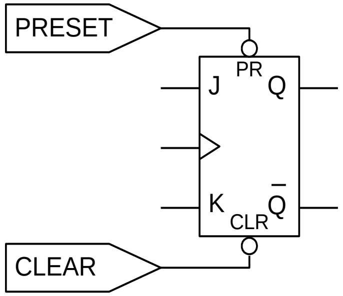


CLK	T	Q_{N+1}	Function
	X	Q	
↓	0	Q	
↓	1	Q´	

Flip Flops with asynchronous inputs (Preset and Clear)

Two extra inputs are often found on flip flops, that either clear or preset the output. These inputs are effective at any time, thus are called asynchronous. If the Clear is at logic 0 then the output is forced to 0, irrespective of the other normal inputs. If the Preset is at logic 0 then the output is forced to 1, irrespective of the other normal inputs. The preset and the clear inputs can not be 0 simultaneously. In the Preset and Clear are both 1 then the flip flop behaves according to its normal truth table.

Positive Edge JK Flip Flop with Preset and Clear



CLK	PR	CLR	J	K	Q_{N+1}	Function
‾|_	0	0	X	X		
↑	0	1	X	X	1	
↑	1	0	X	X	0	
↑	1	1	0	0	Q	
↑	1	1	0	1	0	
↑	1	1	1	0	1	
↑	1	1	1	1	Q'	

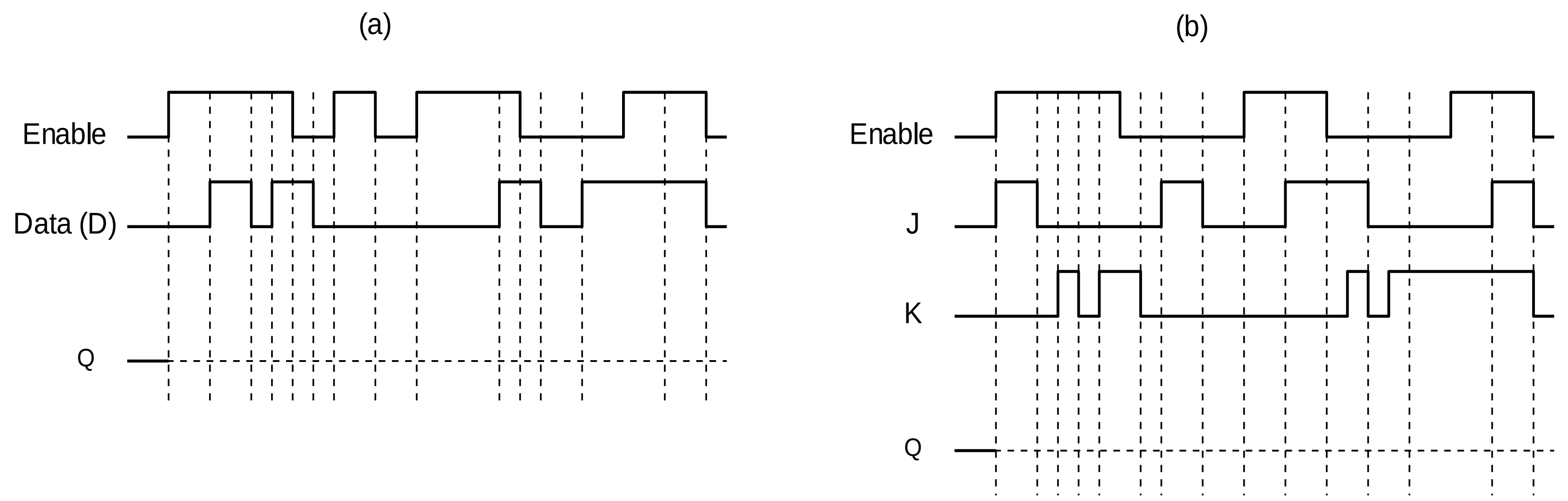
Data (D) Latch :- Example

Complete the timing diagrams for :

(a) D Latch

(b) JK Latch

Assume that for both cases the Q output is initially at logic zero.



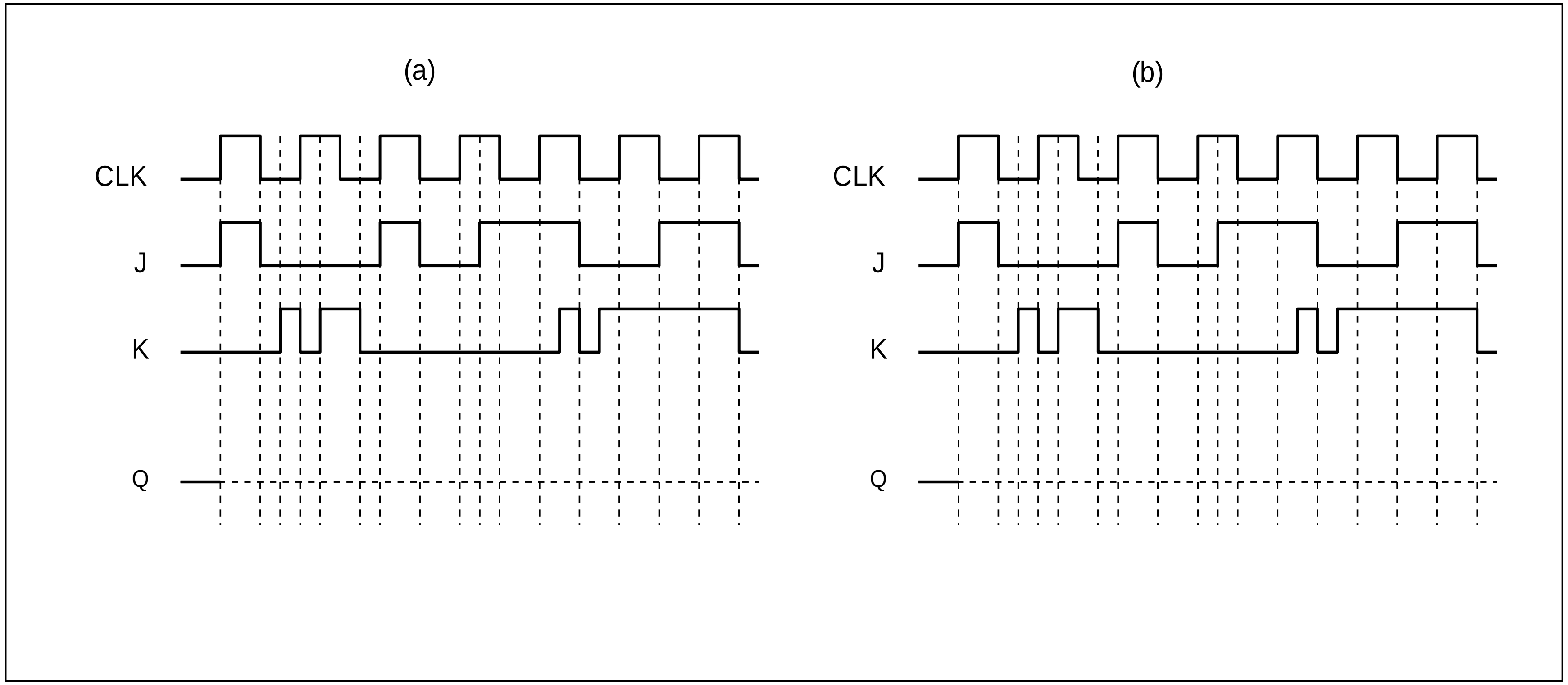
JK Edge Triggered Flip Flop :- Example

Complete the timing diagrams for :

(a) Positive Edge Triggered JK Flip Flop

(b) Negative Edge Triggered JK Flip Flop

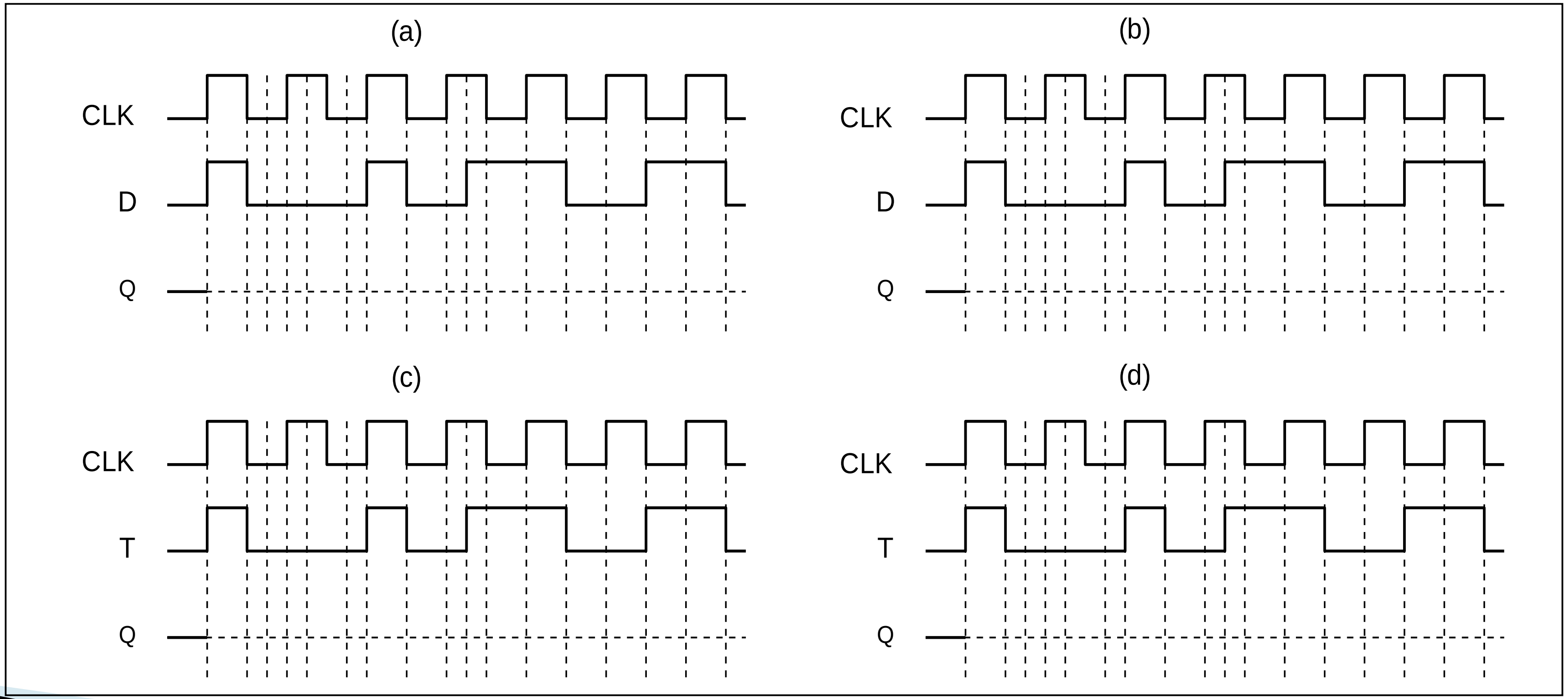
Assume that for both cases the Q output is initially at logic zero.



D and T Edge Triggered Flip Flops :- Example

Complete the timing diagrams for :

(a) Positive Edge Triggered D Flip Flop
(b) Positive Edge Triggered T Flip Flop
(c) Negative Edge Triggered T Flip Flop
(d) Negative Edge Triggered D Flip Flop



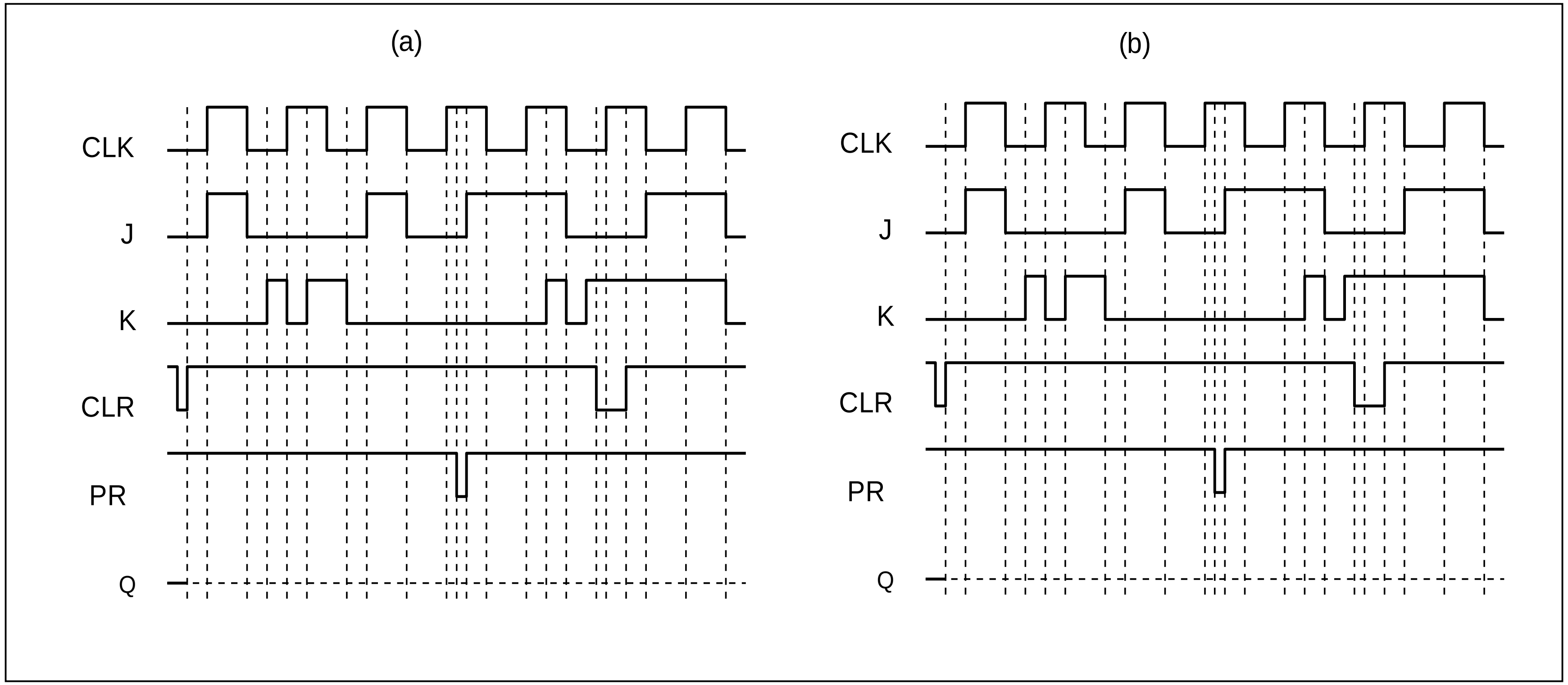
JK Flip Flop With Preset and Clear:- Example

Complete the timing diagrams for :

(a) Positive Edge Triggered JK Flip Flop

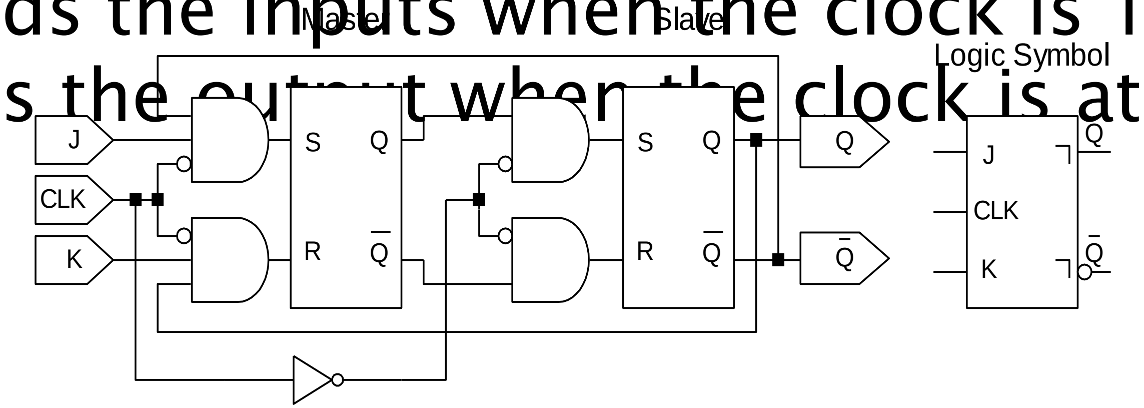
(b) Negative Edge Triggered JK Flip Flop.

Assume that for both cases the Q output is initially at logic zero.



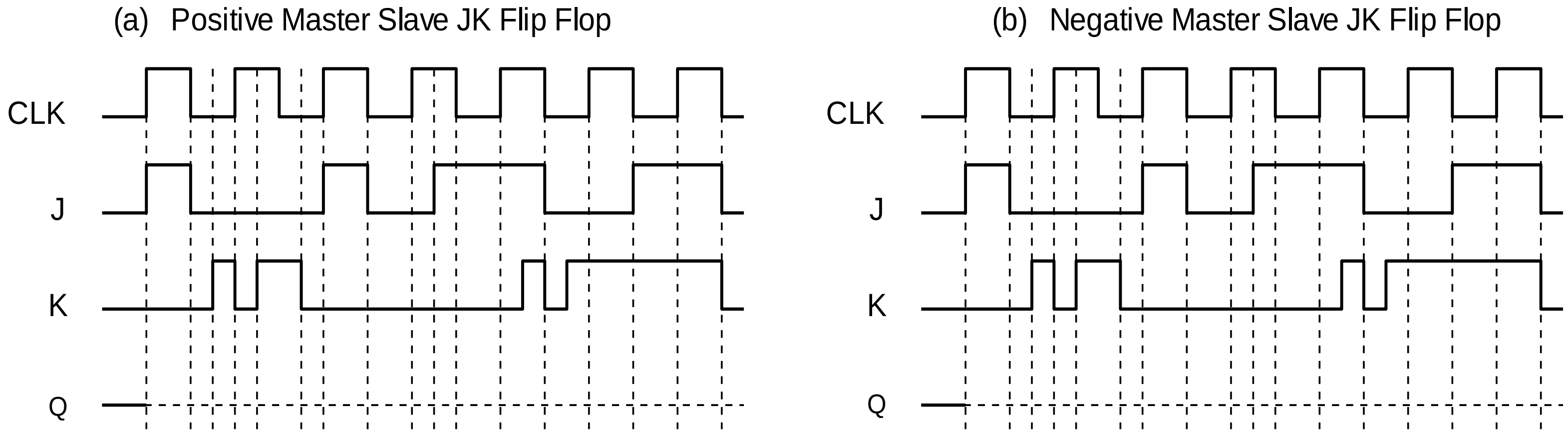
Level Triggered Master Slave JK Flip Flop

A Master Slave flip flop is obtained by connecting two SR latches as shown below. This flip flop reads the inputs when the clock is 1 and changes the output when the clock is at logic zero.



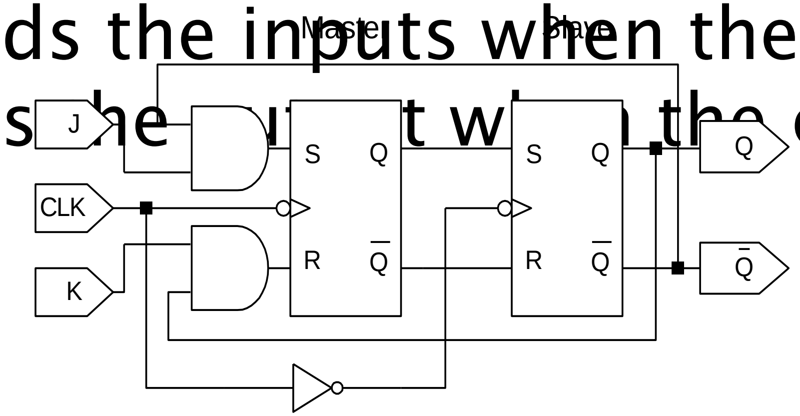
Truth Table

CLK	J	K	Q	Function
	0	0		
	0	1		
	1	0		
	1	1		

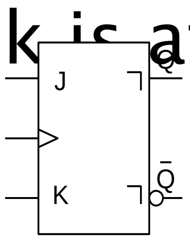


Edge Triggered Master Slave JK Flip Flop

A Master Slave flip flop is obtained by connecting two SR latches as shown below. This flip flop reads the inputs when the clock is 1 and changes the output when the clock is at logic zero.

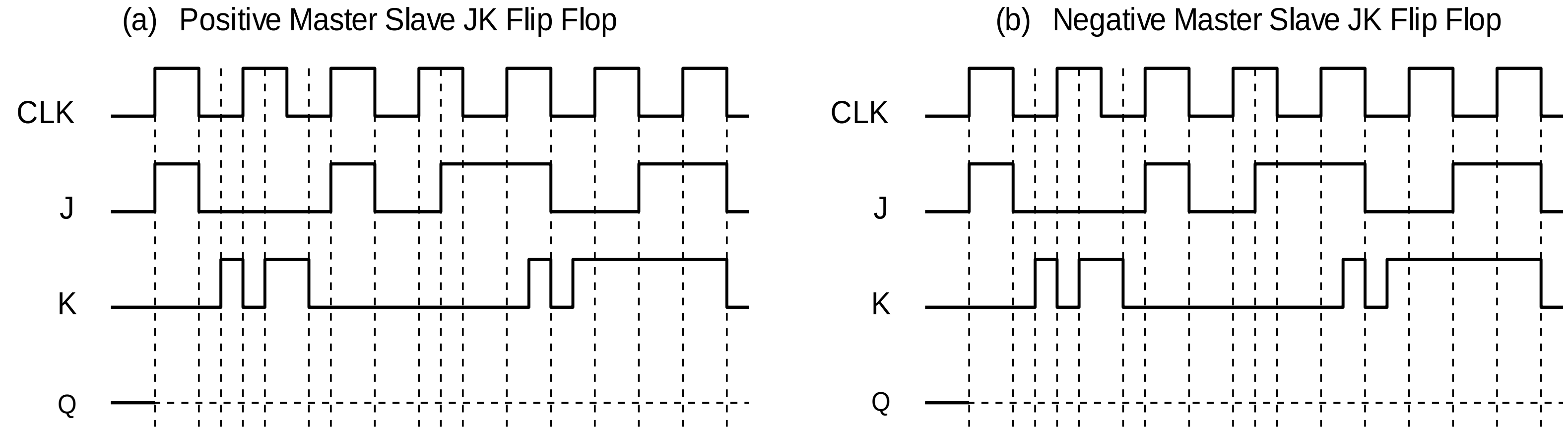


Logic Symbol

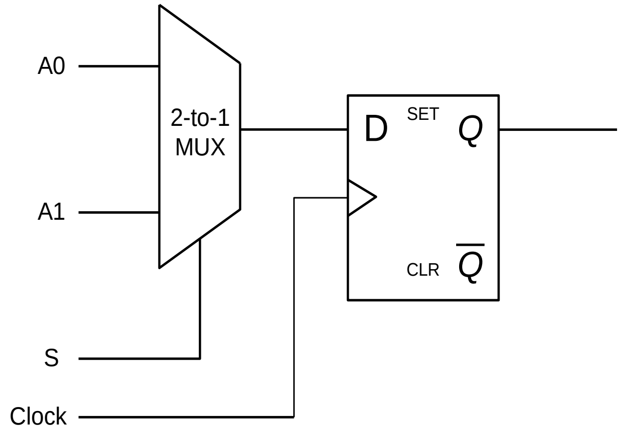


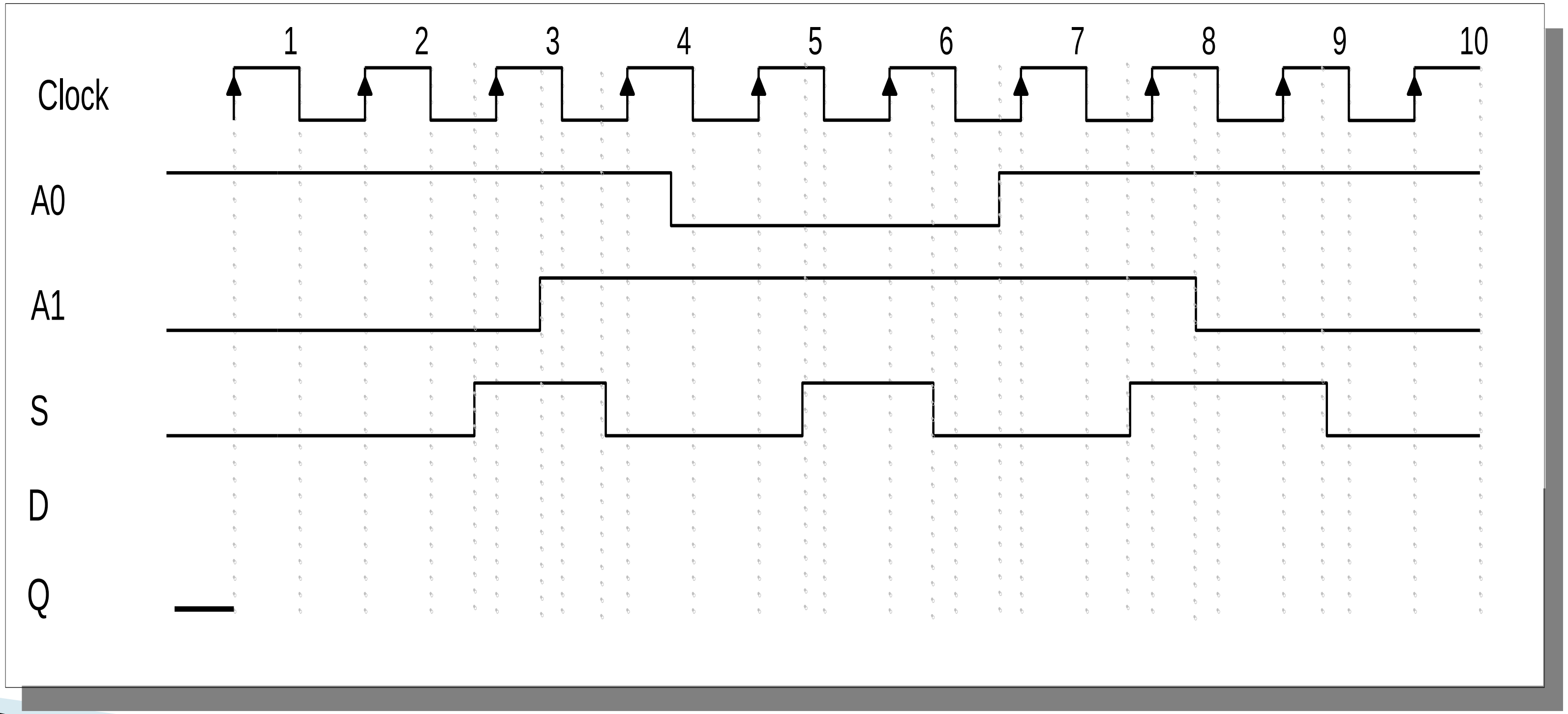
Truth Table

CLK	J	K	Q	Function
	0	0		
	0	1		
	1	0		
	1	1		

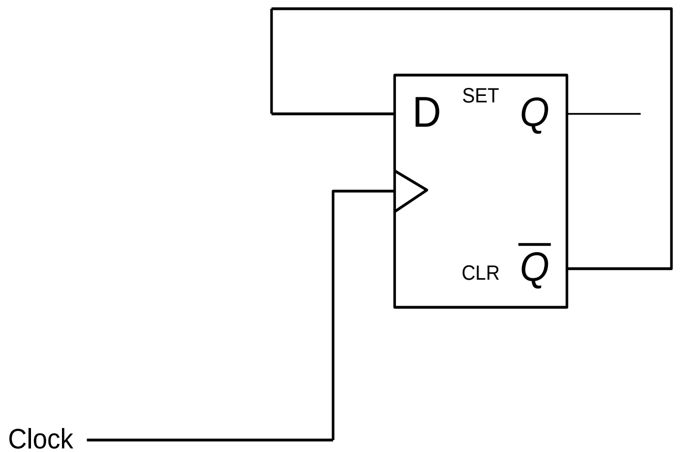


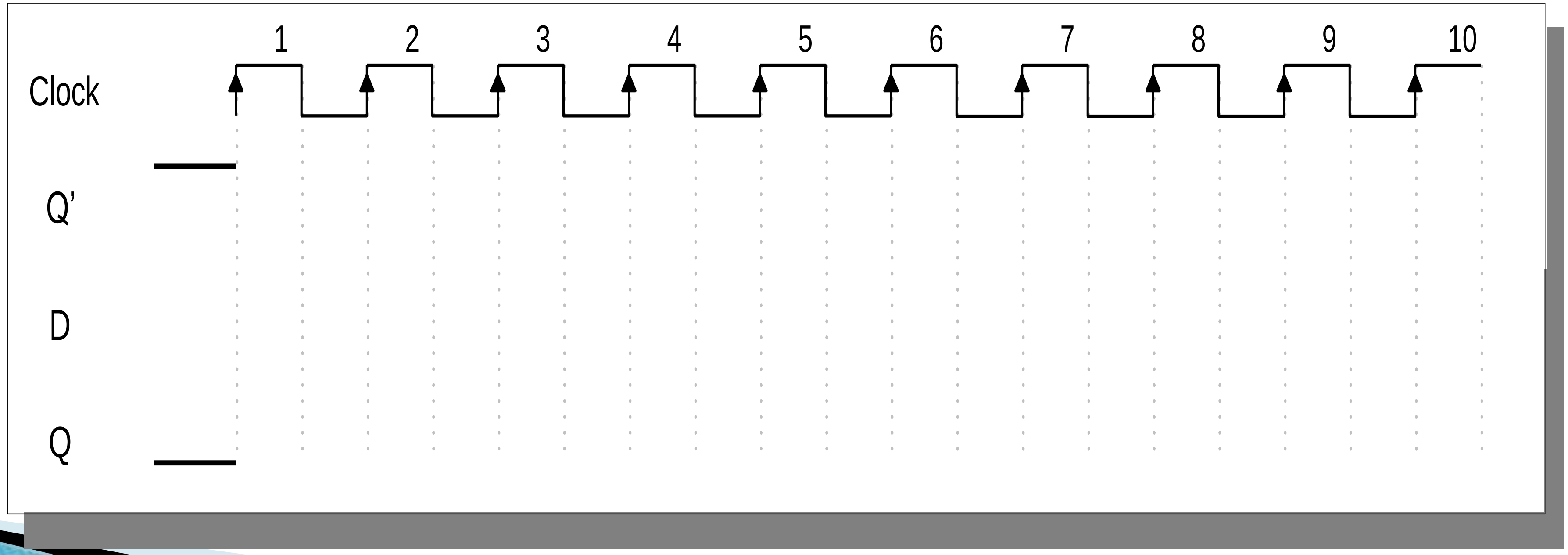
Sequential circuit example 1



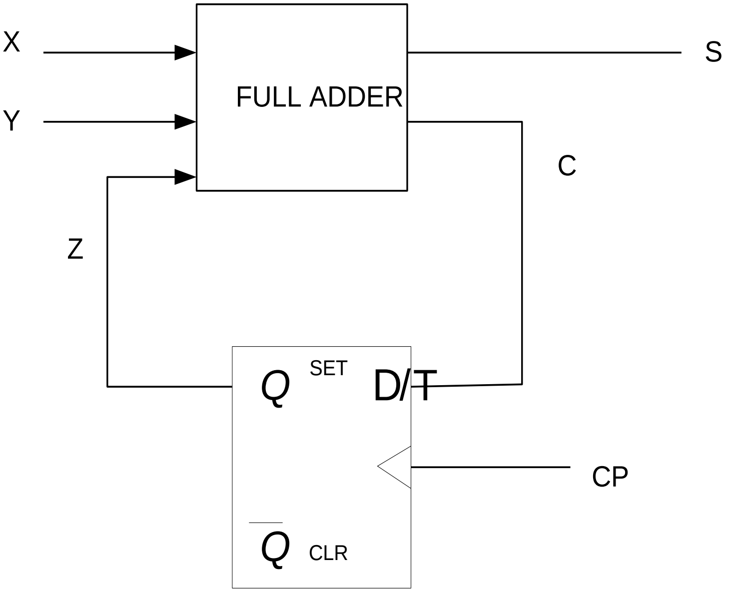


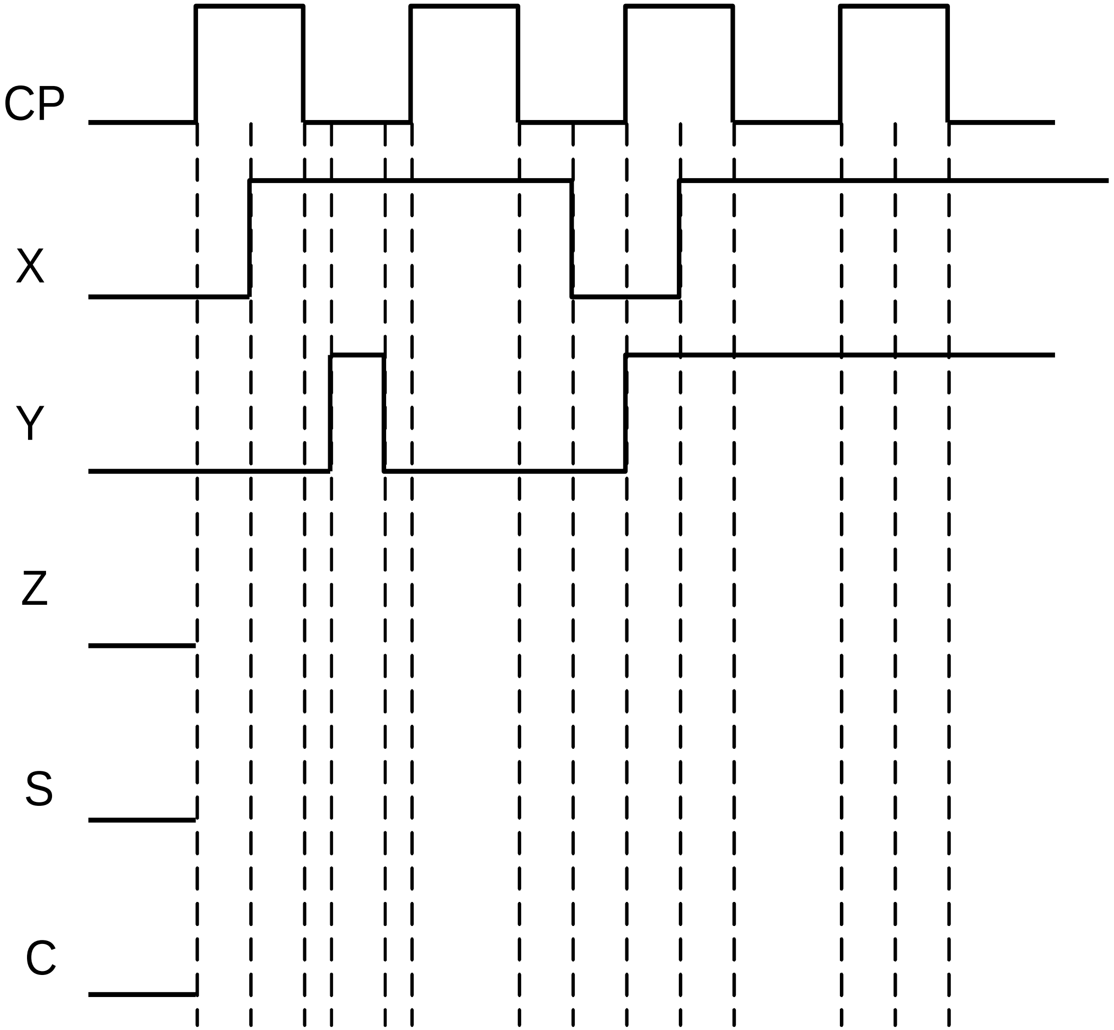
Sequential circuit example 2





Sequential circuit example 3





COUNTERS

A register that goes through a prescribed sequence of states upon the application of input pulses is called a counter.

The input pulses may be clock pulses or they may originate from some external source and may occur at a fixed interval of time or at random.

The sequence of states may follow the same binary number sequence or any other sequence of states .

A counter that follows the binary number sequence is called a binary counter.

An n-bit binary counter consists of n flip flops and can count in binary from o through (2^n)-1 .

Counters are available in 2 categories: ripple counters and synchronous counters.

In a ripple counter, the flip flop output transition serves as a source for triggering other flip flops.

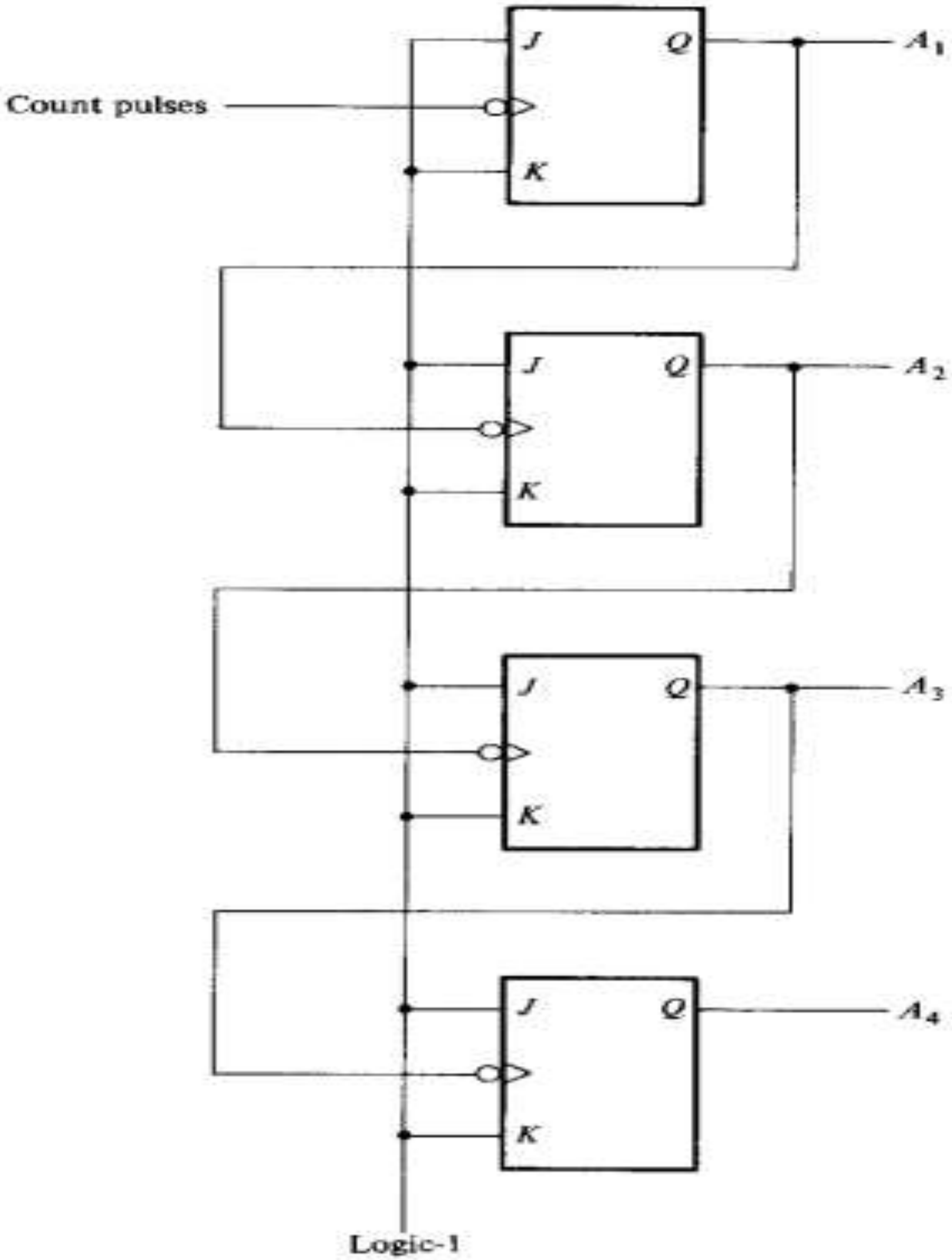
BINARY RIPPLE COUNTER

A binary ripple counter consists of a series connection of complementing flip flops(T or JK), with the output of each flip flop connected to the CP input of the next higher order flip flop.

The flip flop holding the least significant bit receives the incoming count pulses.

In the diagram of 4-bit binary ripple counter, all J and K inputs are equal to 1.

The small circle in the CP input indicates that the flip flop complements during a negative-going transition or when the output to which it is connected goes from 1 to 0.



Count pulses
J
Q
K
A1
A2
A3
A4
Logic-1

Count Sequence for a Binary Ripple Counter

Count Sequence				Conditions for Complementing Flip-Flops	
A_4	A_3	A_2	A_1		
0	0	0	0	Complement A_1	
0	0	0	1	Complement A_1	A_1 will go from 1 to 0 and complement A_2
0	0	1	0	Complement A_1	
0	0	1	1	Complement A_1	A_1 will go from 1 to 0 and complement A_2;
					A_2 will go from 1 to 0 and complement A_3
0	1	0	0	Complement A_1	
0	1	0	1	Complement A_1	A_1 will go from 1 to 0 and complement A_2
0	1	1	0	Complement A_1	
0	1	1	1	Complement A_1	A_1 will go from 1 to 0 and complement A_2;
					A_2 will go from 1 to 0 and complement A_3;
					A_3 will go from 1 to 0 and complement A_4
1	0	0	0	and so on . . .	

From the table it is obvious that the lowest-order bit A1 must be complemented with each count pulse.

Every time A1 goes from 1 to 0, it complements A2.

Every time A2 goes from 1 to 0, it complements A3, and so on.

The flip flops change once at a time in rapid succession, and the signal propagates through the counter in a ripple fashion.

Ripple counters are also called as asynchronous counters

BCD RIPPLE COUNTER

A decimal counter follows a sequence of ten states and returns to 0 after the count of 9.

Such a counter must have at least 4 flip flops to represent each decimal digit is represented by the binary code used to represent a decimal digit.

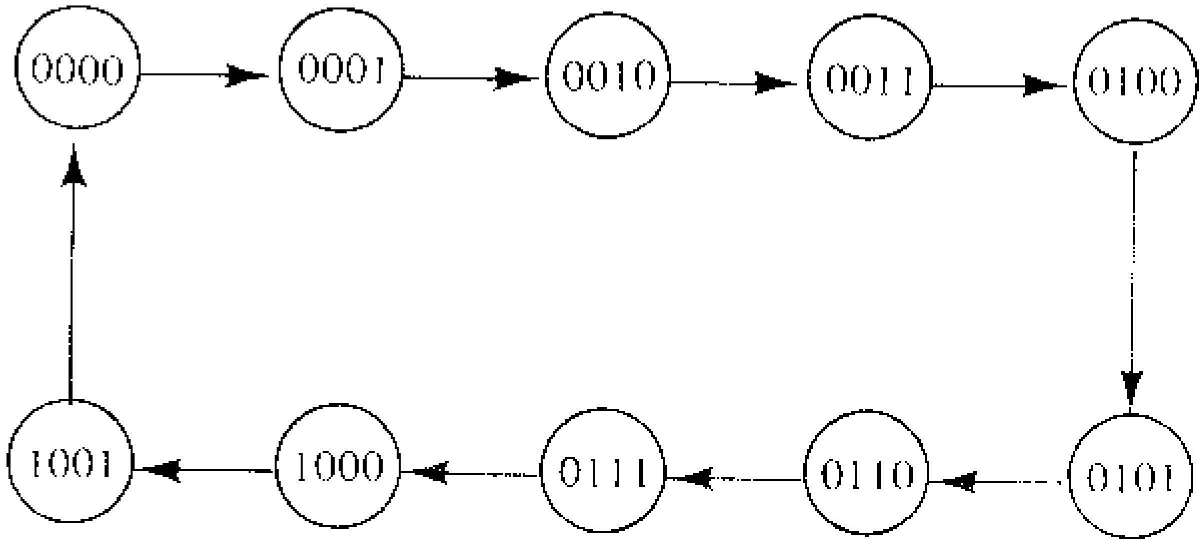
The sequence of states in a decimal counter is dictated by the binary code used to represent a decimal digit.

In the diagram, the output of Q1 is applied to the C inputs of both Q2 and Q8 and the output of Q2 is applied to the C input of Q4.

The J and K inputs are connected either to a permanent 1 signal or to outputs of other flip flops.

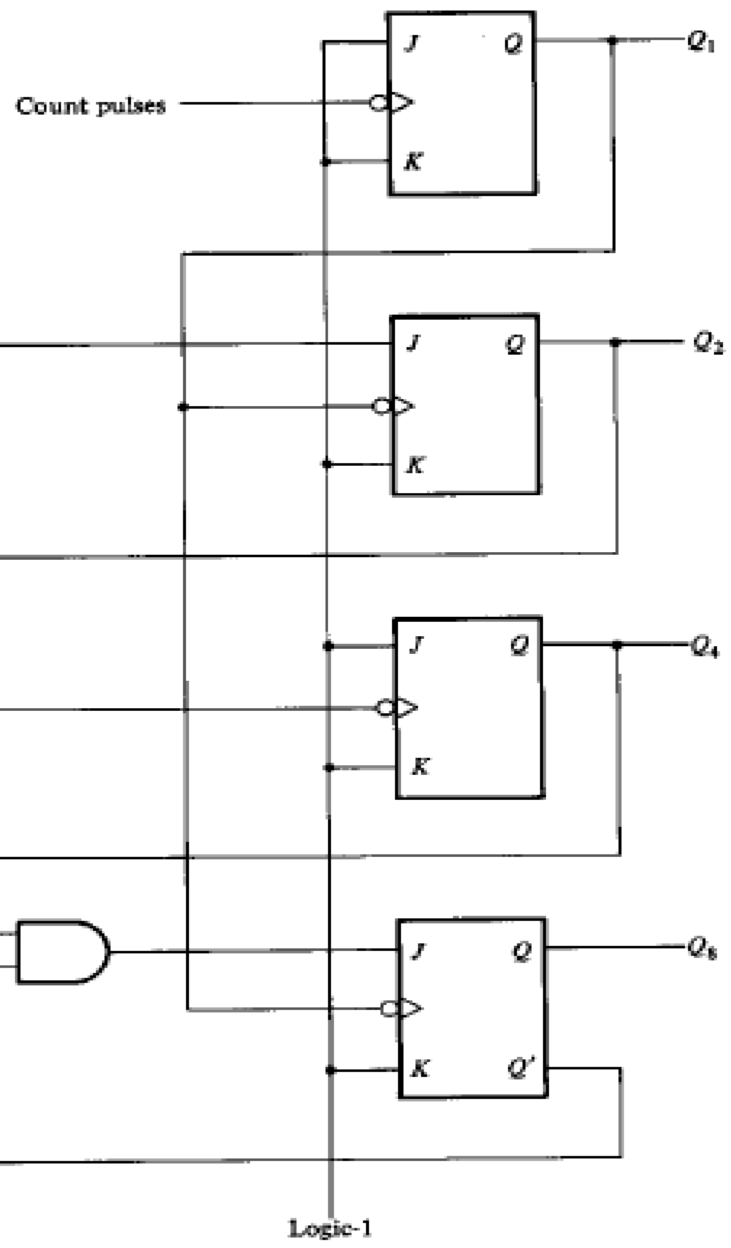
A ripple counter is an asynchronous sequential

State diagram of a decimal BCD counter



→ When the C input goes from 1 to 0, the flip flop is set if J=1, is cleared if K=1, is complemented if J=K=1, and is left unchanged if J=K=0.

BCD ripple counter



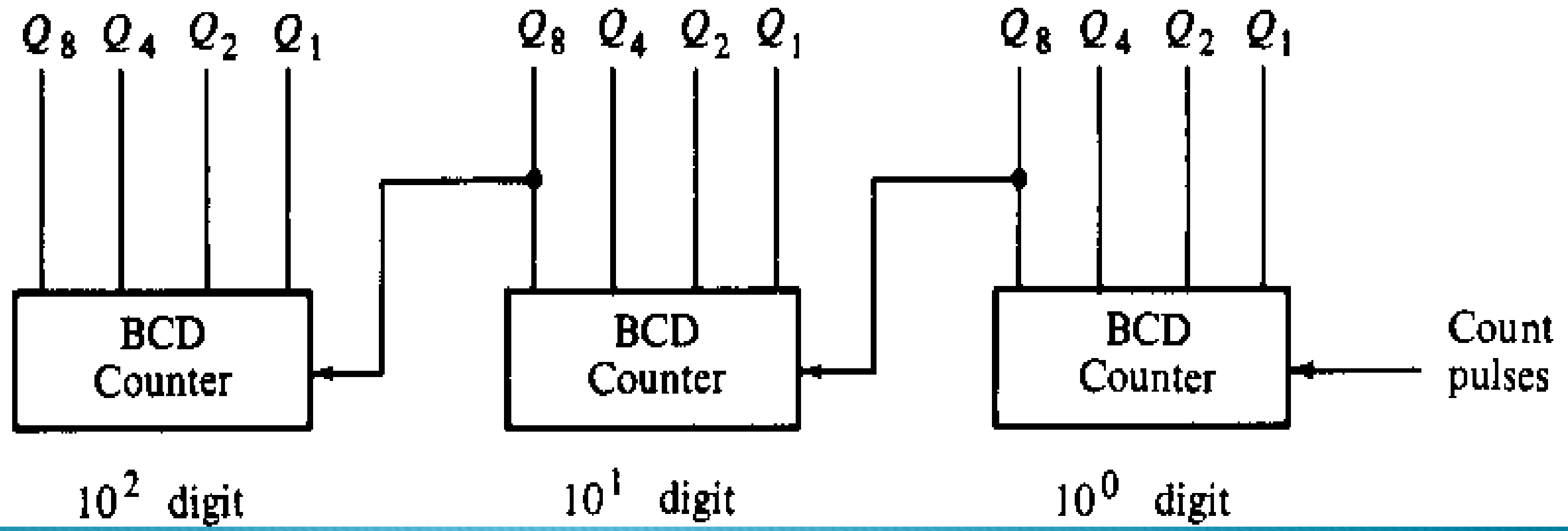
→ The BCD counter is called a decade counter, since it counts from 0 to 9.

→ To count in decimal from 0 to 99, we need a 2-decade counter.

→ To count from 0 to 199, we need a 3-decade counter.

→ In a 3-decade counter, the inputs to the second and third decades come from Q8 of the previous decade.

→ When Q8 in one decade goes from 1 to 0, it triggers the count for the next higher order decade while its own decade goes from 9 to 0.



SYNCHRONOUS COUNTERS

Synchronous counters are different from ripple counters in that clock pulses are applied to the inputs of all flip flops simultaneously rather than one at a time in succession as in a ripple counter.

The decision whether a flip flop is to be complemented or not is determined from the values of the data inputs such as T or J and K at the time of the clock edge.

If T=0 or J=K=0, the flip flop does not change state.

If T=1 or J=K=1, the flip flop complements.

BINARY COUNTER

The design of a synchronous binary counter is so simple that there is no need to go through a sequential logic design process.

In a synchronous binary counter, the flip flop in the least significant position is complemented with every pulse.

A flip flop in any other position is complemented when all the bits in the lower significant positions are equal to 1.

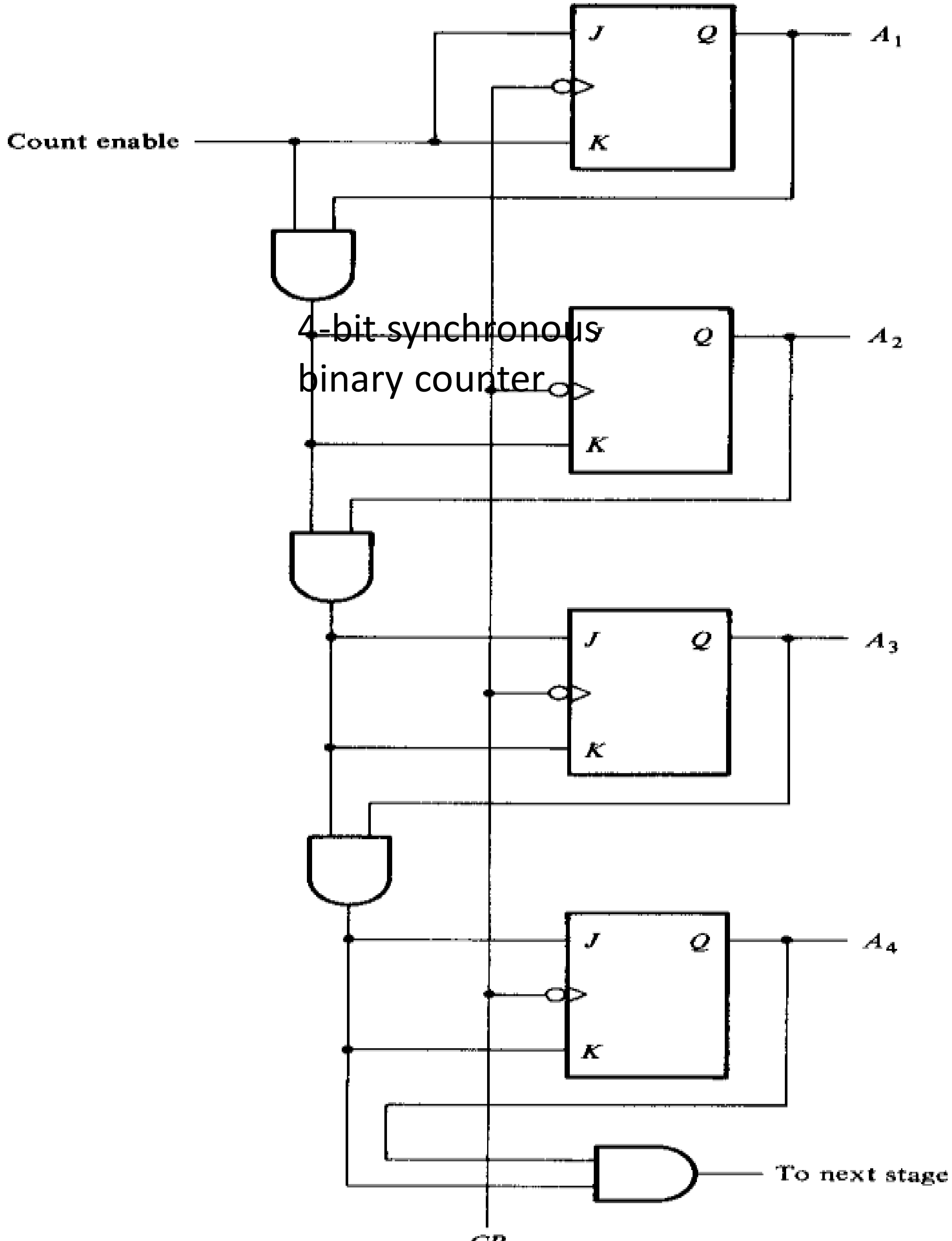
For eg, if the present state of a 4-bit counter is A3A2A1A0=0011, the next count is 0100.

A_0 is always complemented because the present state of $A_0=1$.

A_2 is complemented because the present state of $A_1 A_0 = 11$.

However A_3 is not complemented because the present state of $A_2 A_1 A_0 = 011$, which does not give an all 1's condition.

4-bit synchronous binary counter



In the diagram, the C inputs of all flip flops are connected to a common clock.

The counter is enabled with the count enable input.

If the enable input is 0, all J and K inputs are equal to 0 and the clock does not change the state of the counter.

The first stage A_0 has its J and K equal to 1 if the counter is enabled.

The other J and K inputs are equal to 1 if all previous least significant stages are equal to 1 and count is enabled.

The chain of AND gates generates the required logic for the J and K inputs in each stage.

The counter can be extended to any number of stages, with each stage having an additional flip flop and an AND gate that gives an output of 1 if all previous flip flop outputs are 1

UP-DOWN BINARY COUNTER

A synchronous count down binary counter goes through the binary states in reverse order from 1111 down to 0000 and back to 1111 to repeat the count.

The bit in the least significant position is complemented with each pulse.

A bit in any other position is complemented if all lower significant bits are equal to 0.

For eg, the next state after the present state of 0100 is 0011.

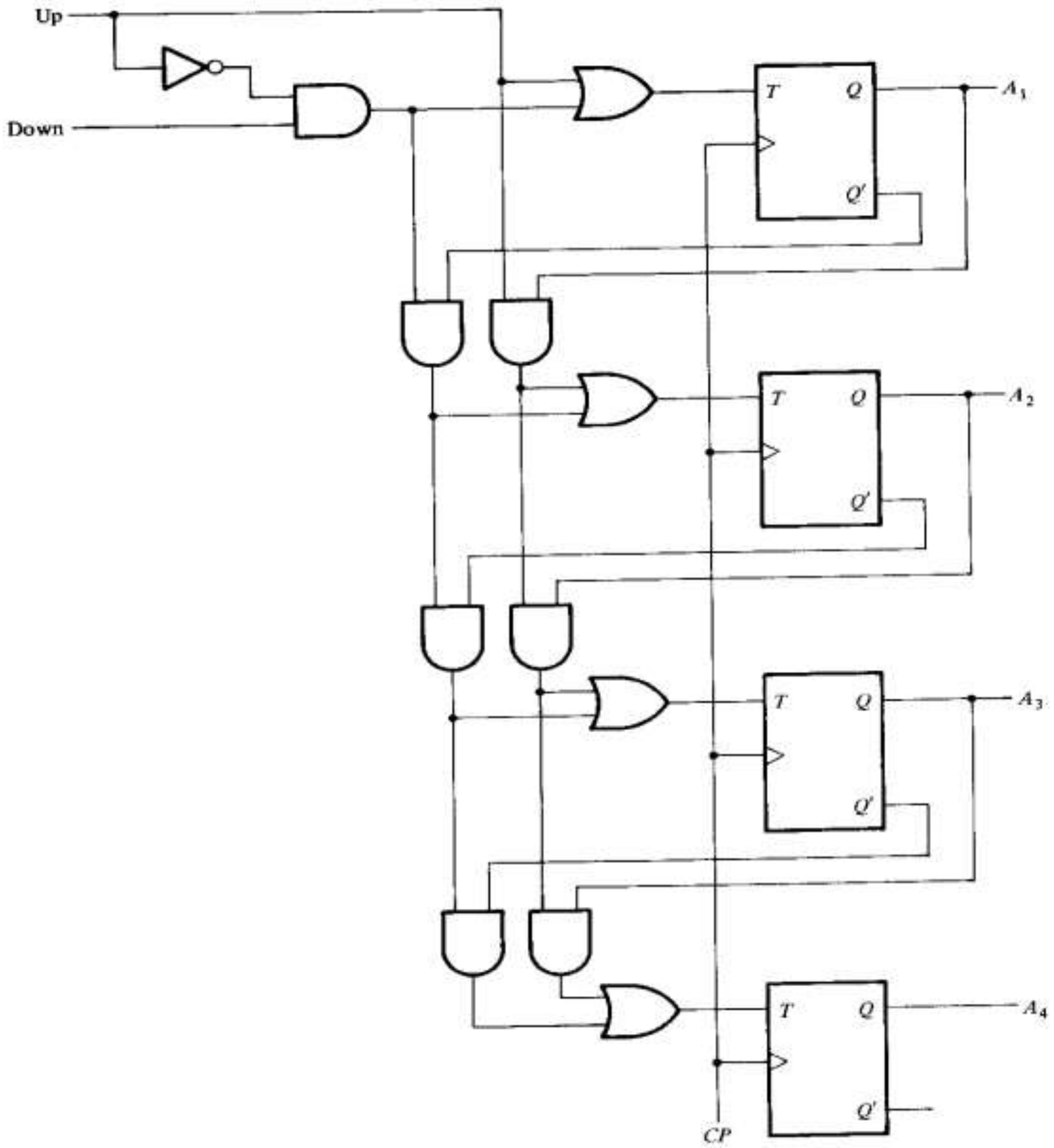
The least significant bit is always complemented.

The second significant bit is complemented because the first 2 bits are equal to 0.

The 3^{rd} significant bit is complemented because the 1^{st} 2 bits are equal to 0.

But the 4^{th} bit does not change because not all lower significant bits are equal to 0.

4-bit up-down binary counter



Both up counting and down counting can be combined in one circuit to form a counter capable of counting either up or down.

It has an up control input and a down control input.

When the up input is 1, the circuit counts up, since the T inputs receive their signals from the values of the previous normal outputs of the flip flops.

When the up input is 1 and the down input is 0, the circuit counts up, since the T inputs receive their signals from the values of the normal outputs of the flip flops.

When the down input is 1 and the up input is 0, the circuit counts down, since the complemented outputs of the previous flip flops are applied to the T inputs.

When the up and down inputs are both 0, the circuit does not change state and remains in the same count.

When the up and down inputs are both 1, the circuit counts up.

BCD COUNTER

A BCD counter counts in binary-coded decimal from 0000 to 1001 and back to 0000.

Excitation Table for BCD Counter

Present State				Next State				Output	Flip-Flop Inputs			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1	y	TQ_8	TQ_4	TQ_2	TQ_1
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	1	0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	1	0	0	1

- The flip flop input conditions for the T flip flops are obtained from the present and next state conditions.
- The output is equal to 1 when the present state is 1001.
- In this way, y can enable the count of the next higher significant decade while the same pulse switches the present decade from 1001 to 0000.
- The flip flop input equations can be simplified by means of maps.
- The unused terms are taken as don't care terms.
- The circuit can be easily drawn with 4 T flip flops, 5 AND gates and 1 OR gate.
- Synchronous BCD counters can be cascaded to form a counter for decimal numbers of any length.

$$TQ_1 = 1$$

$$TQ_2 = Q_8' Q_1$$

$$TQ_4 = Q_2 Q_1$$

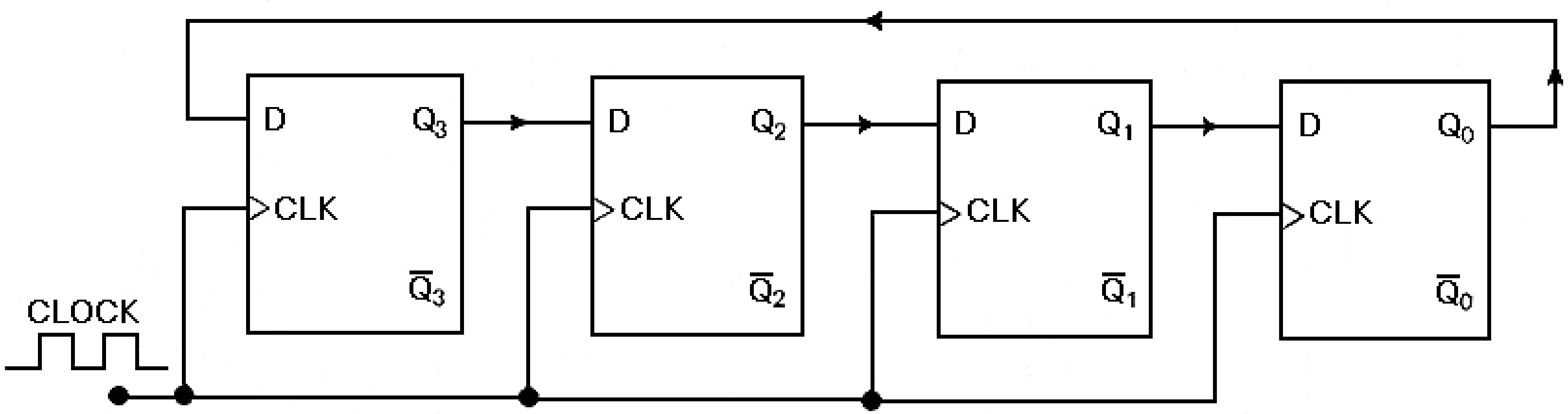
$$TQ_8 = Q_8 Q_1 + Q_4 Q_2 Q_1$$

Procedure to Design Synchronous Counters

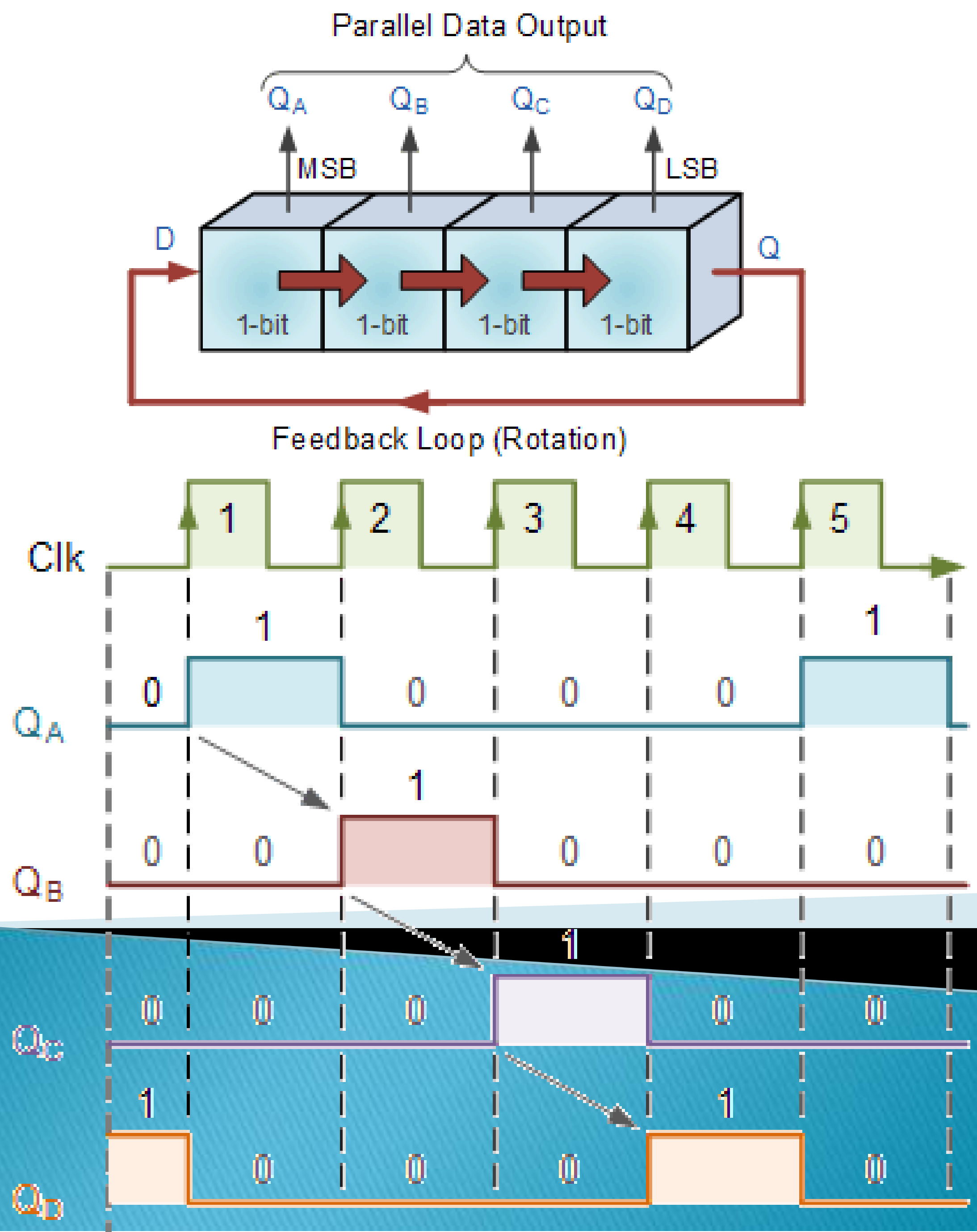
The procedure to design a synchronous counter is listed here.

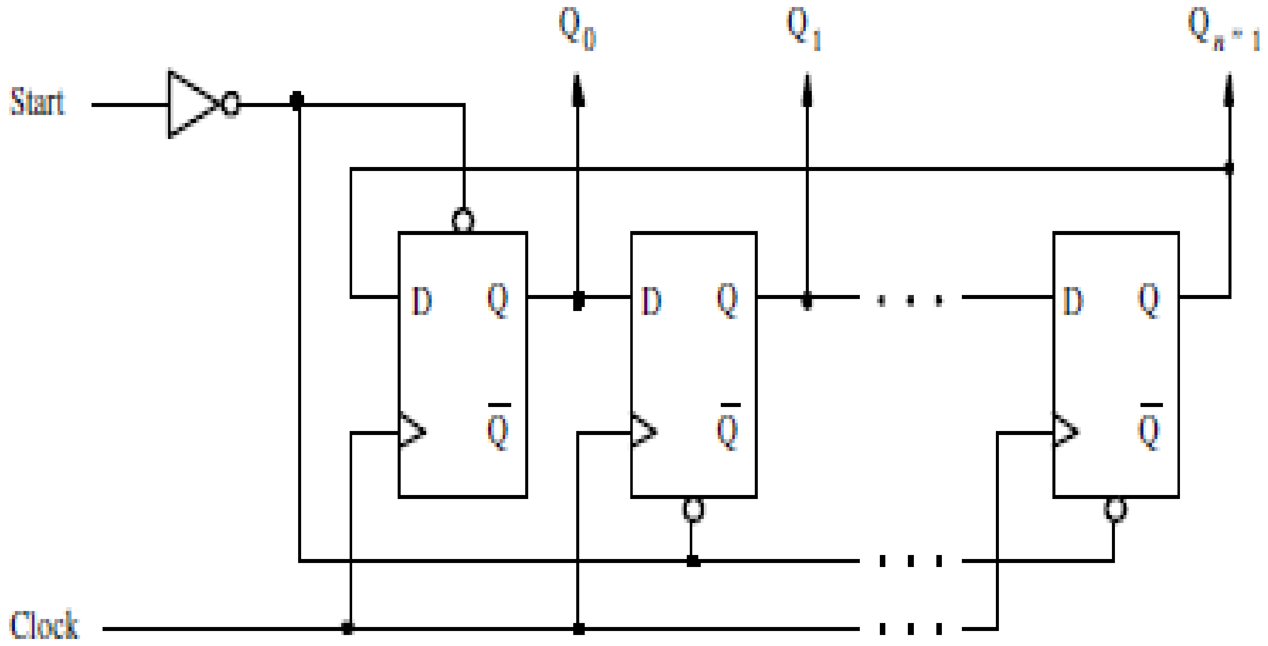
- Obtain the truth table of the logic sequence for intended counter to be designed. Alternatively obtain the state diagram of the counter.
- Determine the number and type of flip-flop to be used.
- From the excitation table of the flip-flop, determine the next state logic.
- From the output state, use Karnaugh map for simplification to derive the
circuit output functions and the flip-flop output functions.
- Draw the logic circuit diagram.

❖Ring Counter



Ring counters are implemented using shift registers. It is essentially a circulating shift register connected so that the last flip-flop shifts its value into the first flip-flop. There is usually only a single 1 circulating [redacted] applied. (Starts 1000->0100->0010->0001 repeat)

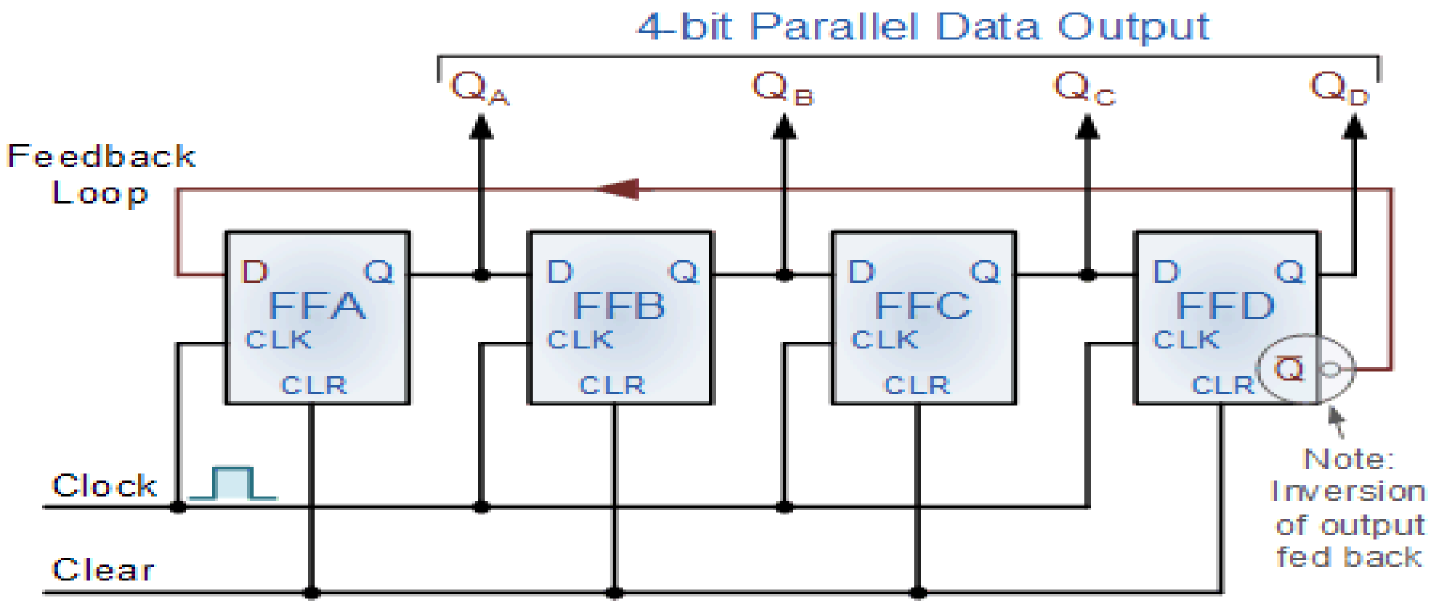

Parallel Data Output
Q_A
Q_B
Q_C
Q_D
MSB
LSB
D
Q
1-bit
1-bit
1-bit
1-bit
Feedback Loop (Rotation)
Clk
1
2
3
4
5
Q_A
Q_B
Q_C
Q_D



(a) An *n*-bit ring counter

Start control signal, which presets the left-most flip-flop to 1 and clears the others to 0.

Johnson Counter



The Johnson counter, also known as the twisted-ring counter, is exactly the same as the ring counter except that the inverted output of the last flip-flop is connected to the input of the first flip-flop.

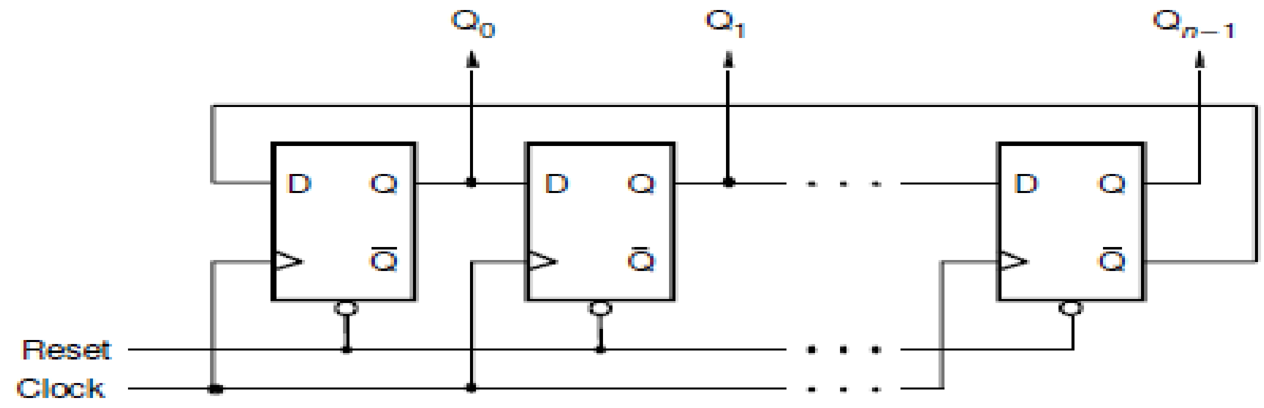
Let's say, starts from 000, 100, 110, 111, 011 and 001, and the sequence is repeated so long as there is input pulse.

Truth Table for a 4-bit Johnson Ring Counter

Clock Pulse No	FFA	FFB	FFC	FFD
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1

As well as counting or rotating d ing counters can also be used to dete or number values within a set of gic gates such as the *AND* or the *OR* gates to the outputs of the flip-flops the circuit can be made to detect a set number or value. Standard 2, 3 or 4-stage Johnson ring counters can also be used to divide the frequency of the clock signal by varying their feedback connections and divide-by-3 or divide-by-5 outputs are also available.

Johnson Counter

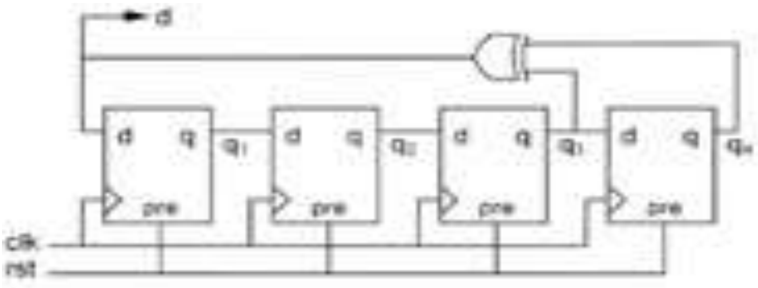


To initialize the operation of the Johnson counter, it is necessary to reset all flip-flops, as shown in the figure. [redacted] Johnson nor the ring counter will generate the desired counting sequence if not initialized properly.

Pseudo–Random Sequence Generators

- The generation of pseudo-random bit sequences is particularly useful in communication and computing systems
- Pseudo-random sequences are normally generated using a circuit called *linear-feedback shift register* (LFSR)
- it consists simply of a tapped circular shift register with the taps feeding a modulo-2 adder (XOR gate) whose output is fed back to the first flip-flop

The shift register must start from a nonzero state

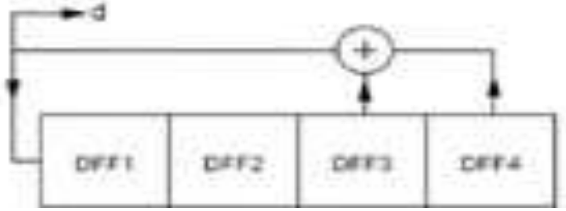


(a)

reset →

$Q_1Q_2Q_3Q_4$	d	$Q_1Q_2Q_3Q_4$	d
1111	0	1100	0
0111	0	0110	1
0011	0	1011	0
0001	1	0101	1
1000	0	1010	1
0100	0	1101	1
0010	1	1110	1
1001	1	1111	0

(b)



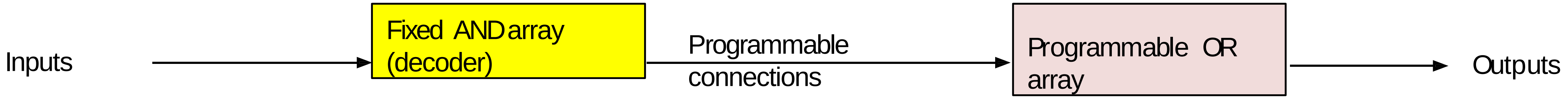
(c)

V UNIT

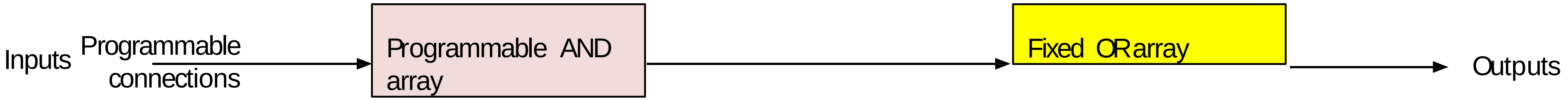
PROGRAMMABLE DEVICES

Programmable Logic Devices (PLDs)

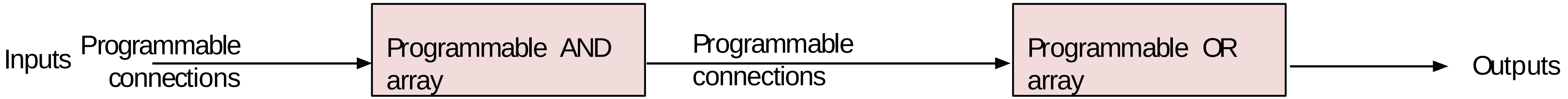
All use AND-OR structure- differ in which is programmable



Programmable read-only memory (PROM)

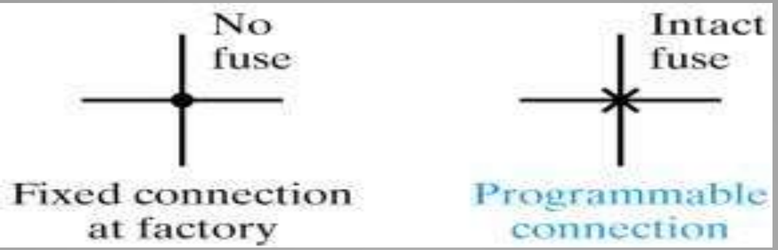


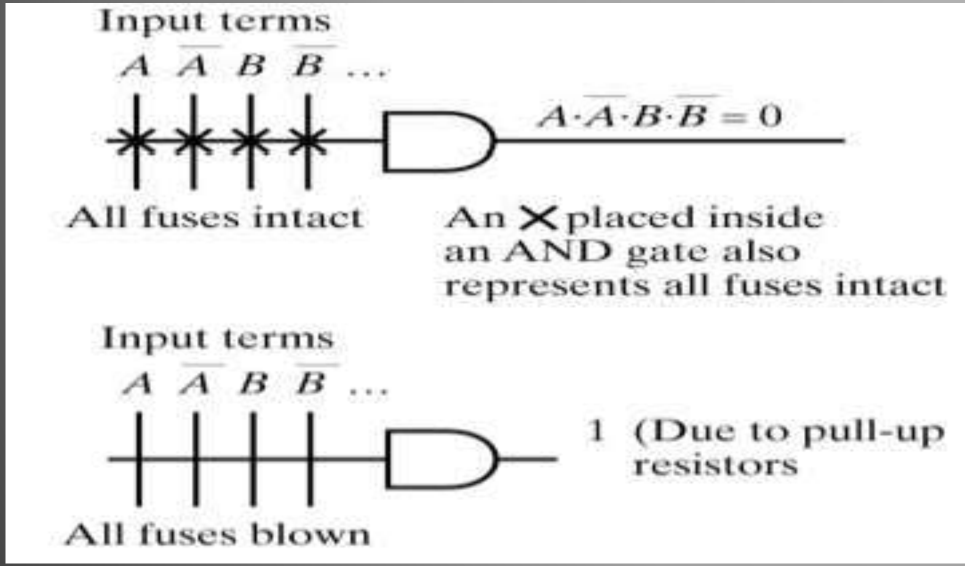
Programmable array logic (PAL) device

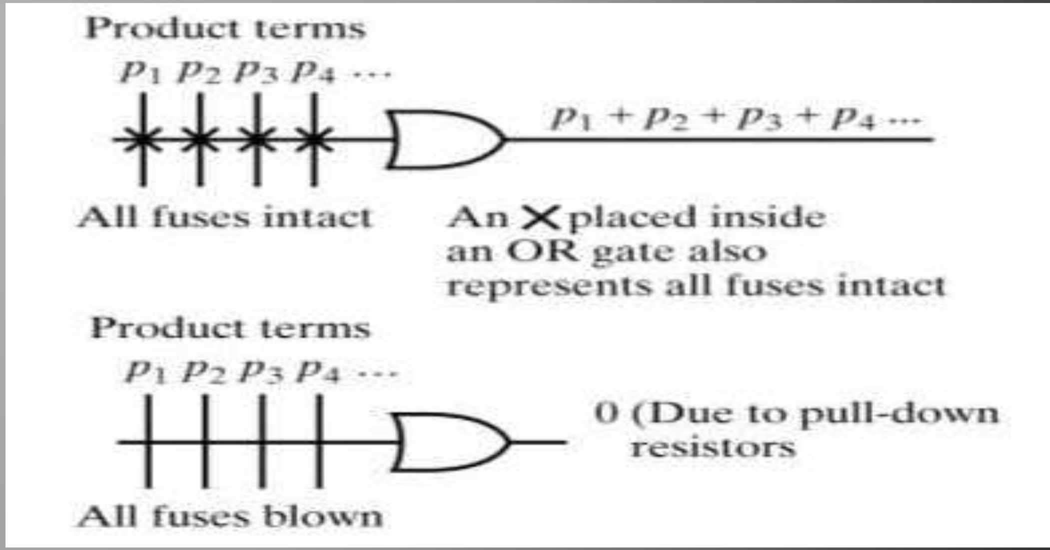


Programmable logic array (PLA)

Programmable Symbology

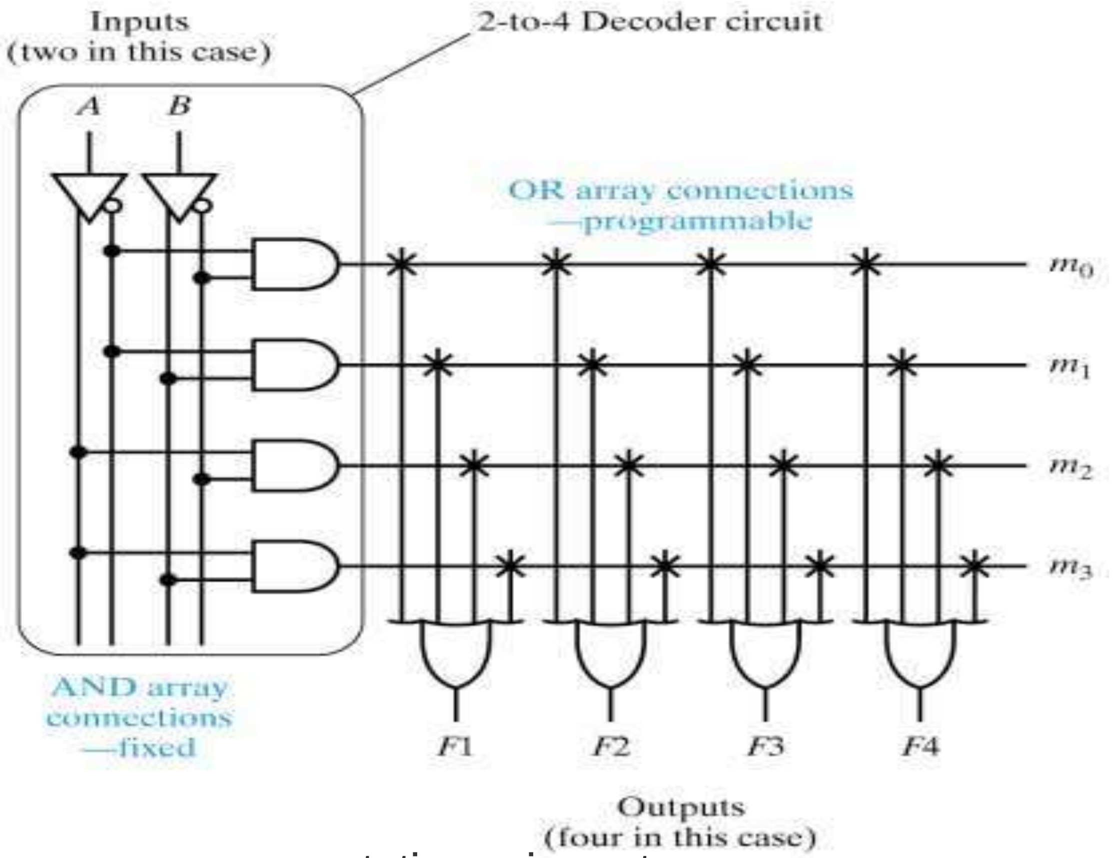






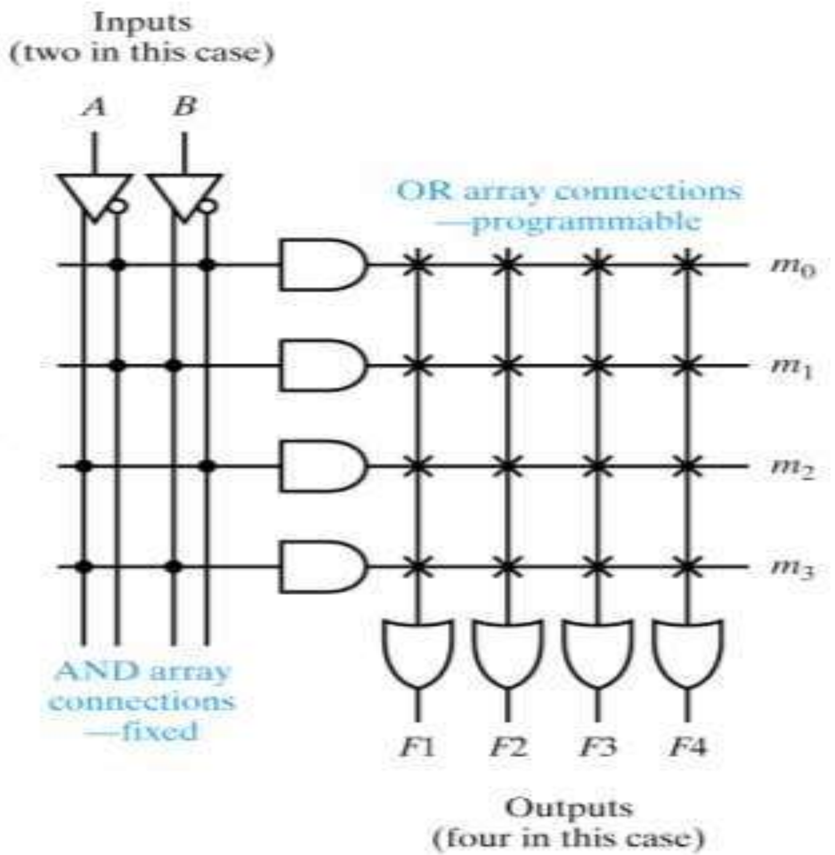
PROM

Note: This PROM has 4 memory locations of 4 bits each



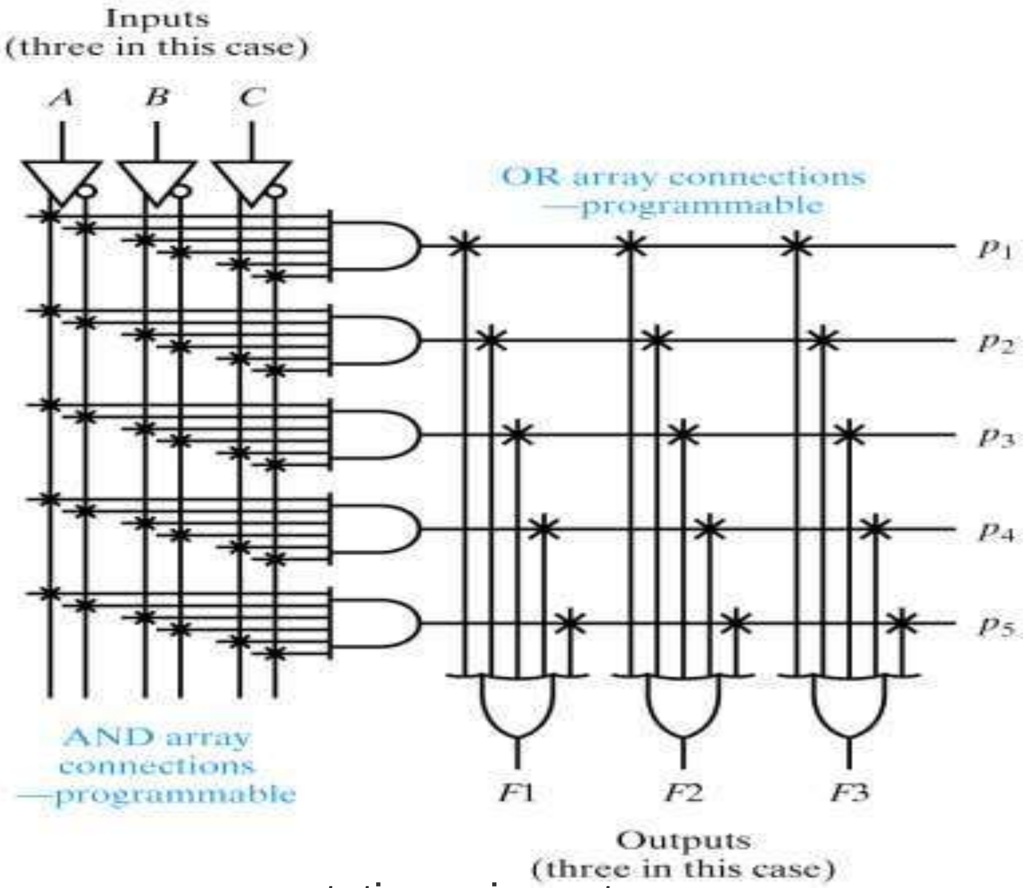
representation using gates

⇒



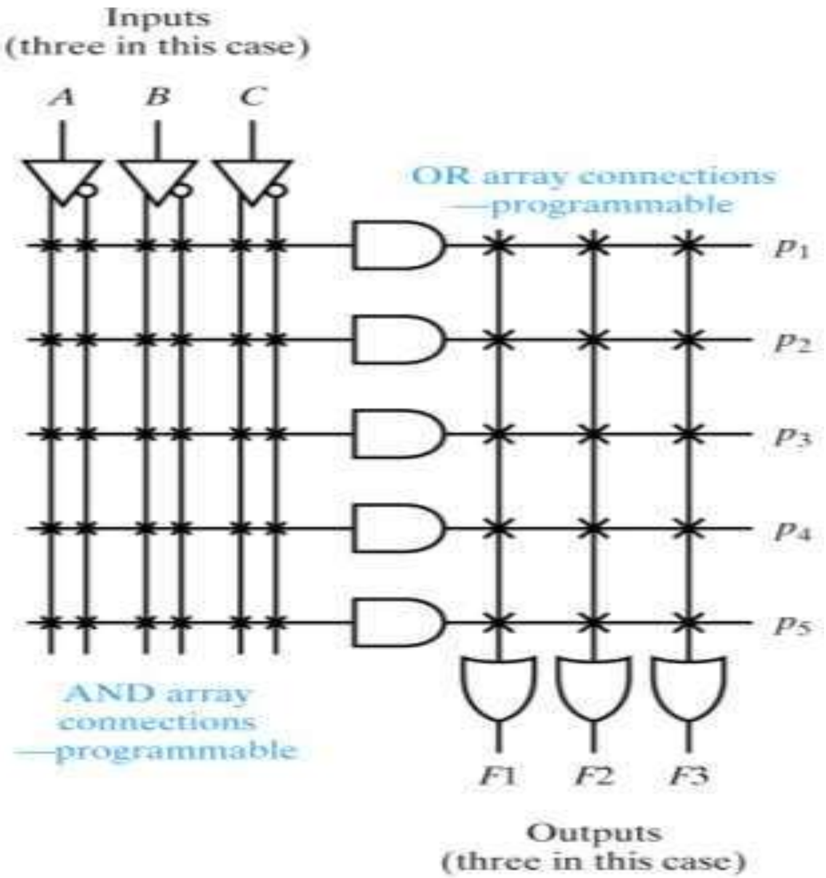
simplified representation

PLA



representation using gates

⇒

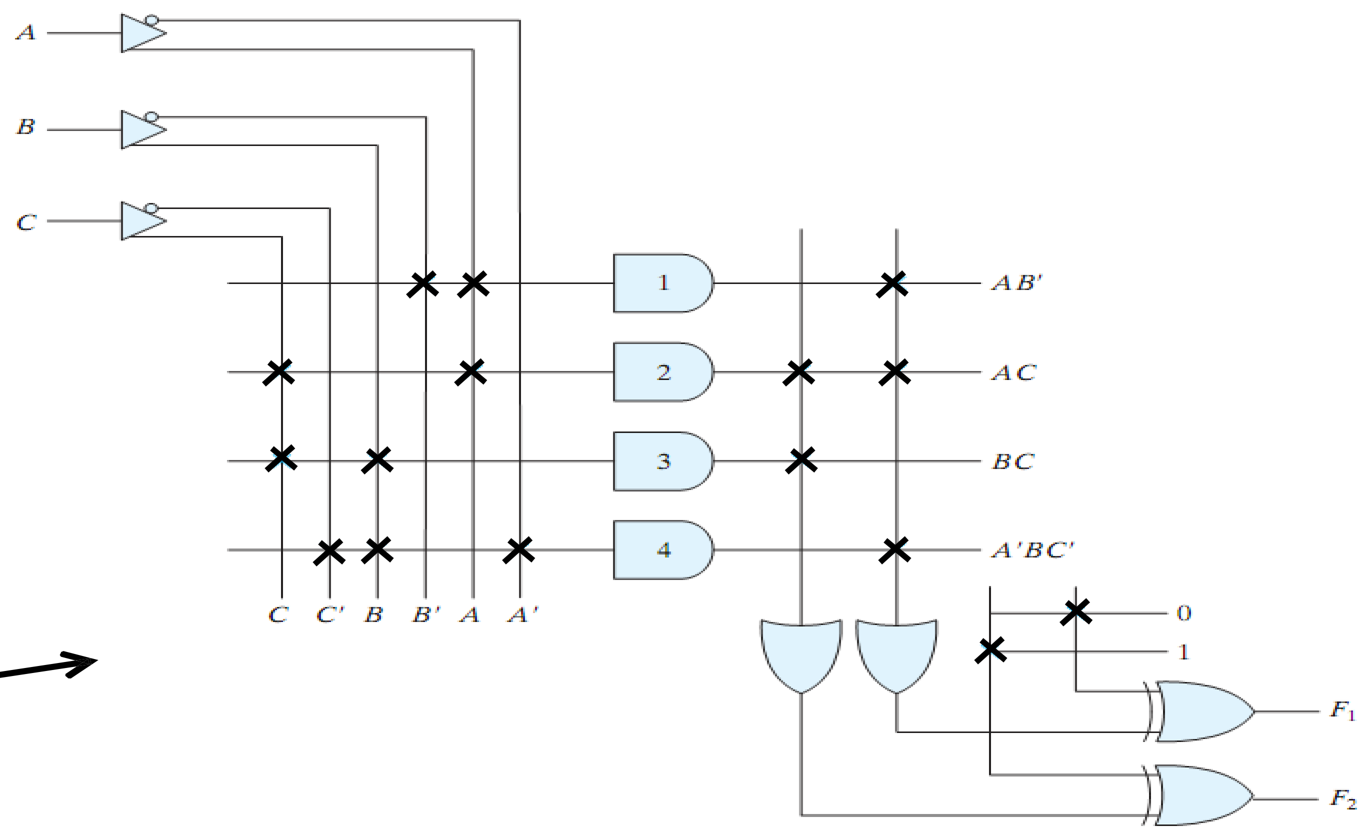


simplified representation

Programmable Logic Array (PLA)

- AND array and OR array are programmable
- XOR is available to complement an output if needed

- **Example:**
  - 3 inputs/2 outputs
  - $F_1 = AB' + AC + A'BC'$
  - $F2 = (AC + BC)'$

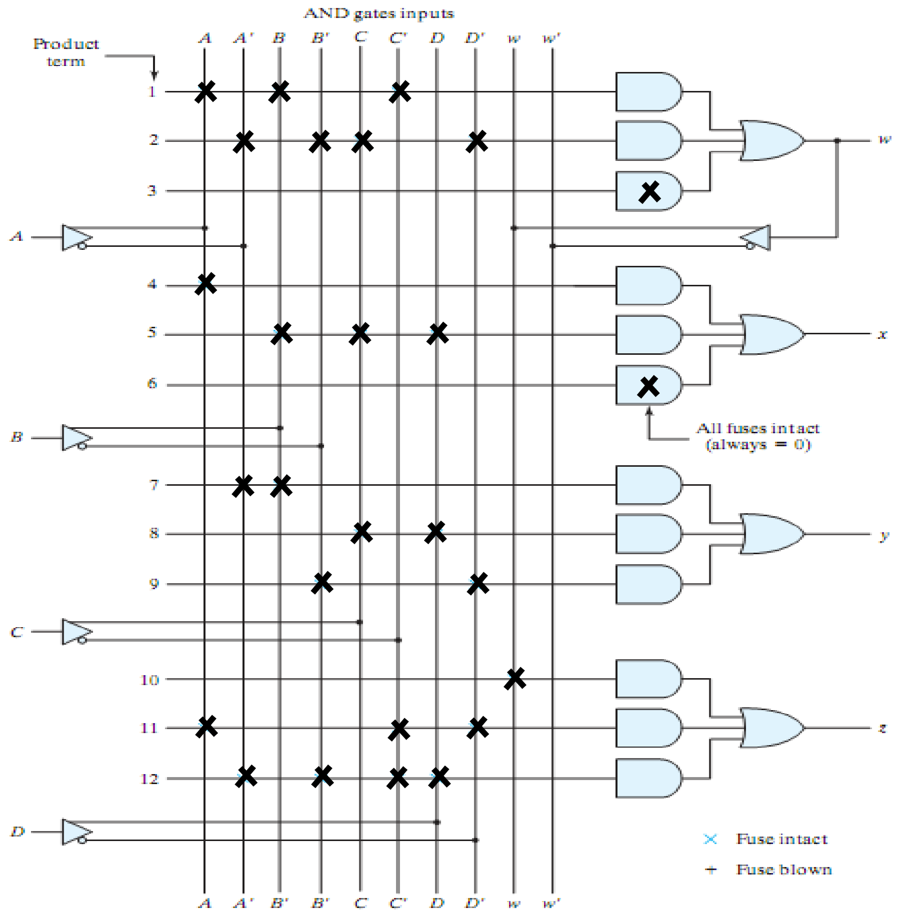


Source: Mano's textbook

Programmable Array Logic (PAL)

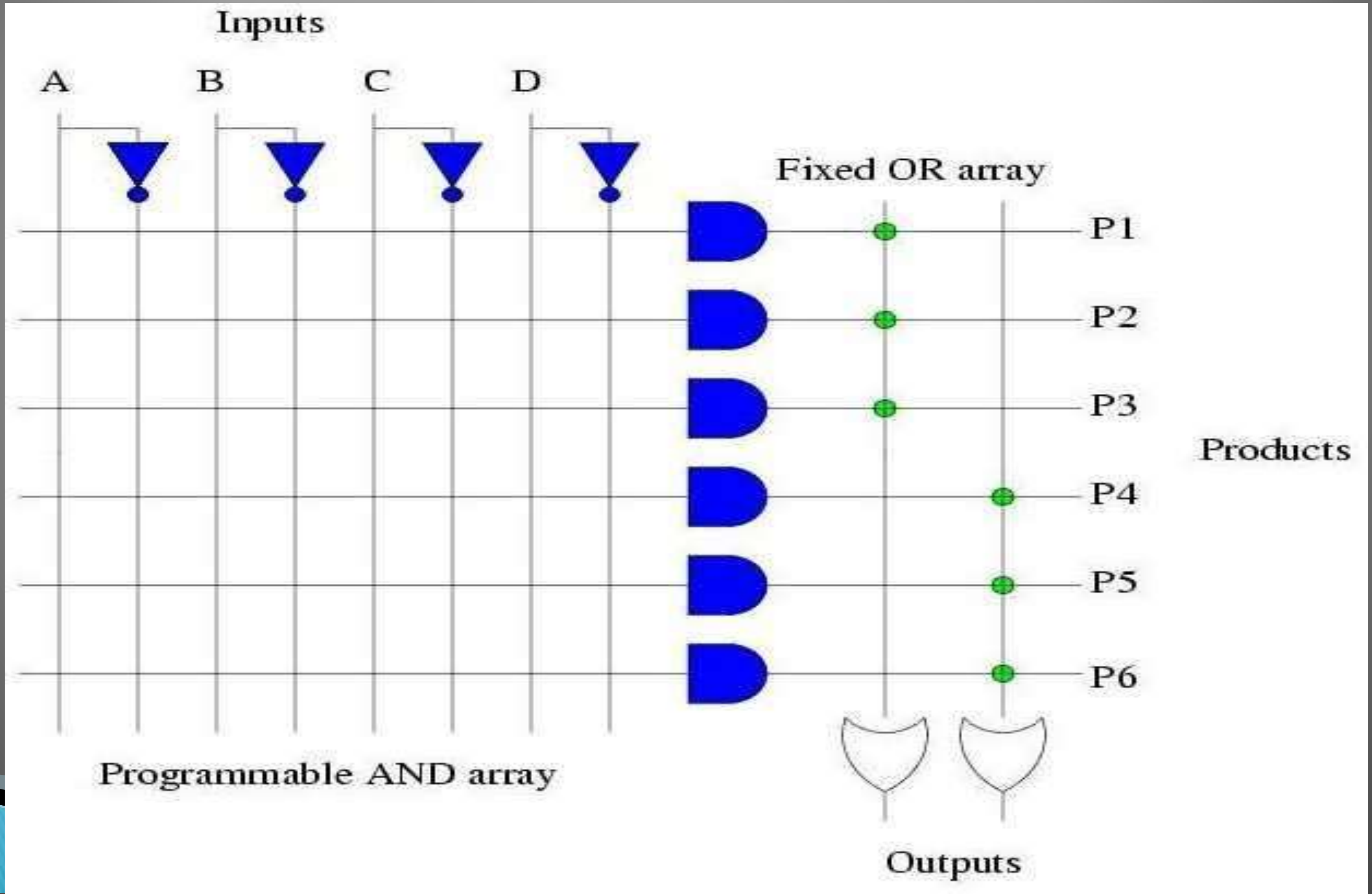
- Fixed OR array and programmable AND array
  - Opposite of ROM
- Feed back is used to support more product terms
- AND output can not be shared here!

- **Example:**
- 4 inputs/4 outputs with fixed 3- input OR gates
- W = ABC' + A' B' CD'
- X = ?
- Y = ?
- Z = ?



Source: Mano's textbook

PAL Logic Diagram



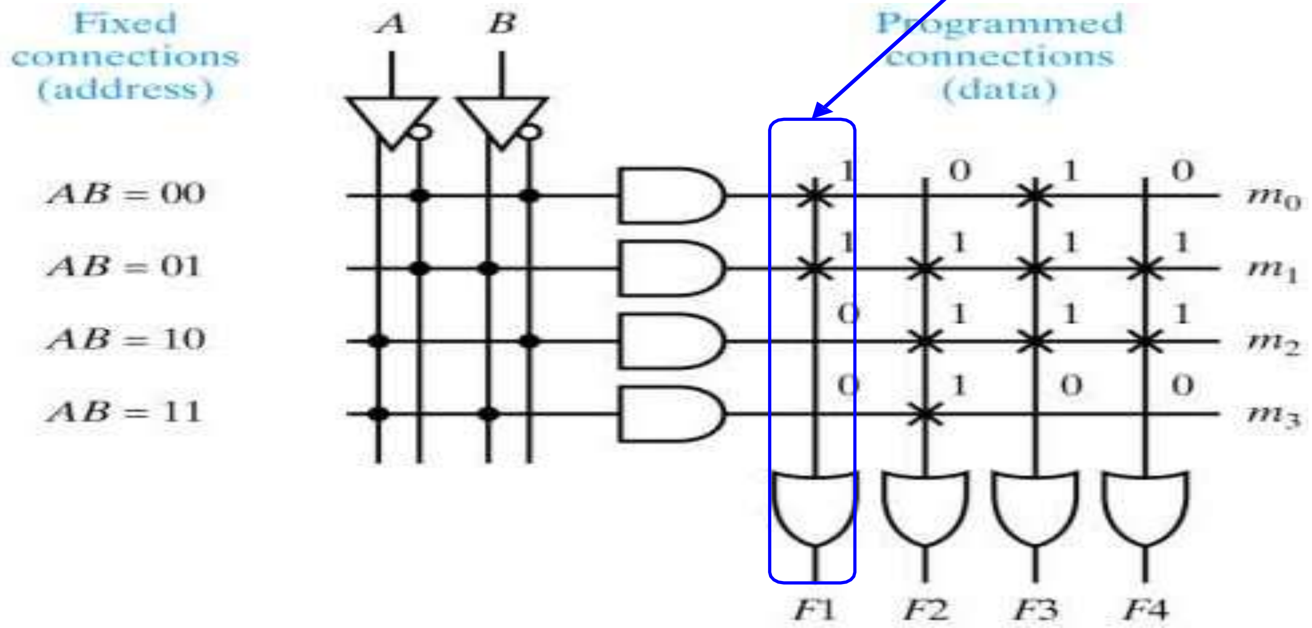
PROM Design Example

Use a PROM to implement an:

- inverter $F1 = \overline{A}$
  - OR $F2 = A+B$
  - NAND $F3 = \overline{A \cdot B}$
  - XOR $F4 = A \oplus B$

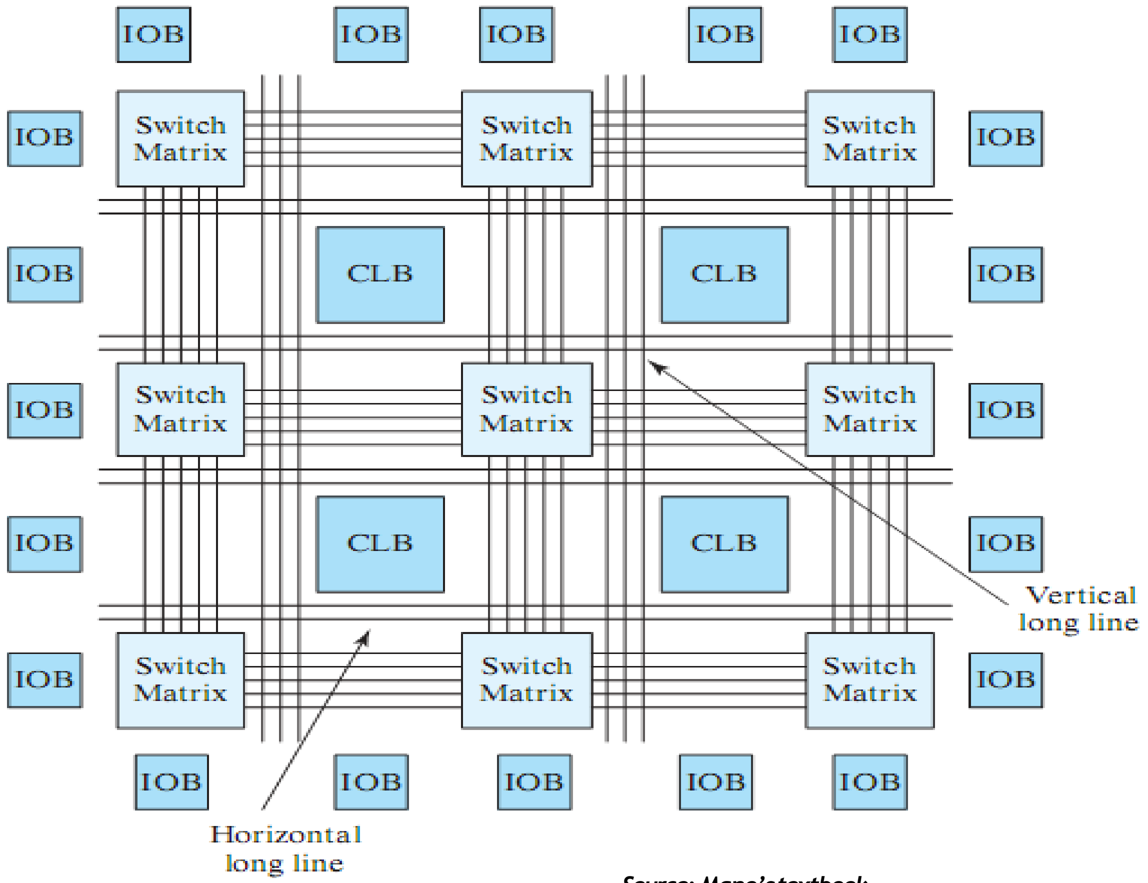
AB	F1	F2	F3	F4
00	1	0	1	0
01	1	1	1	1
10	0	1	1	1
11	0	1	0	0

Truth table is transferred directly to the PROM grid.



Field Programmable Gate Array (FPGA)

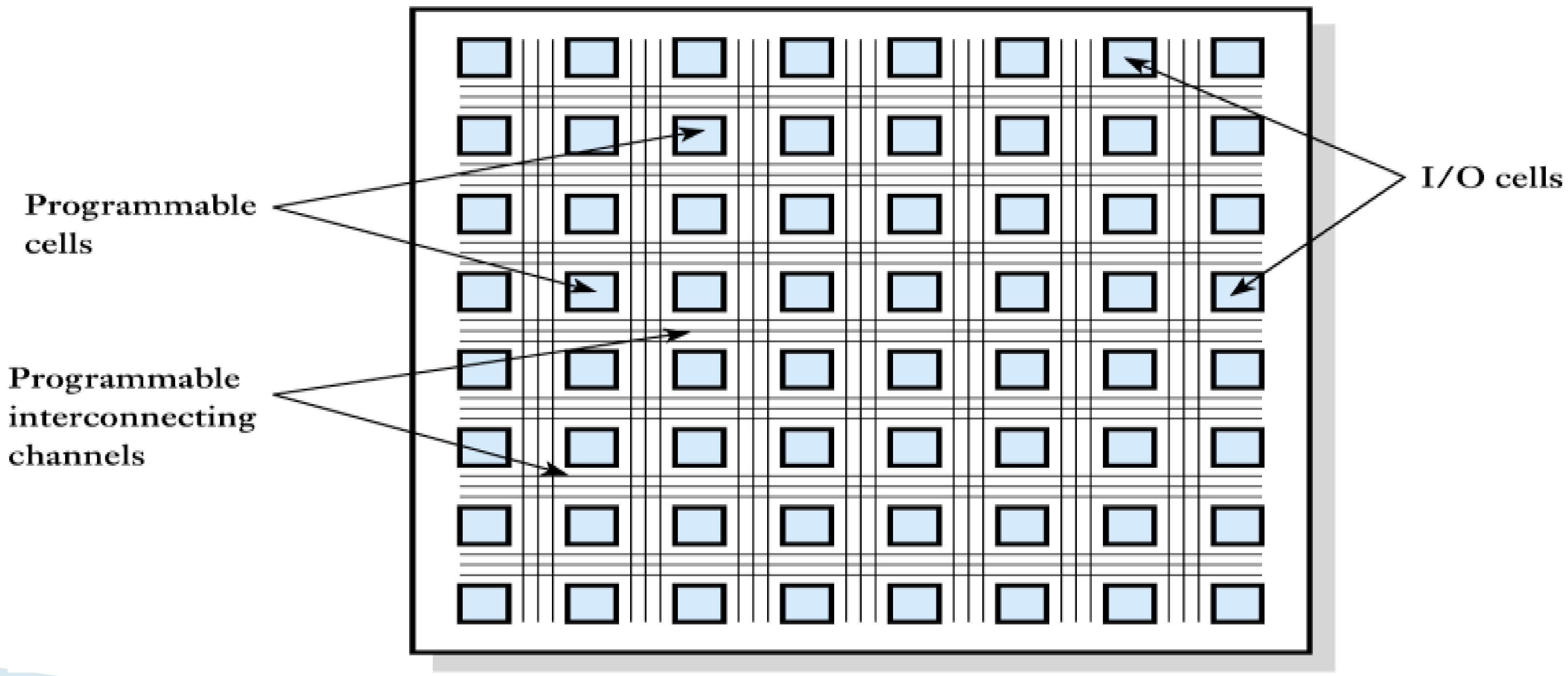
- **Xilinx FPGAs**
- Configurable Logic Block (CLB)
  - Programmable logic and FFs
- Programmable Interconnects
  - Switch Matrices
  - Horizontal/vertical lines
- I/O Block (IOB)
  - Programmable I/O pins



Source: Mano's textbook

Field programmable gate arrays

– a programmable device using more complex cells



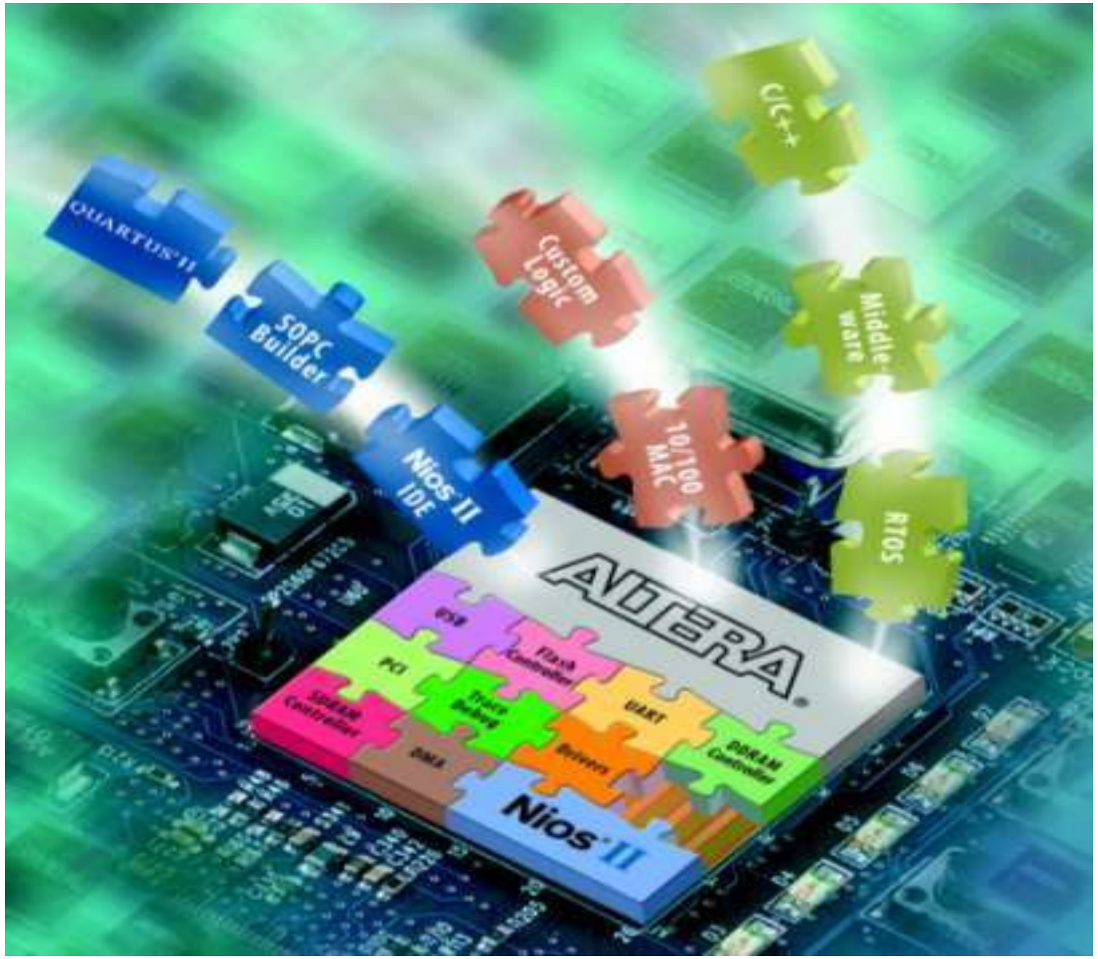
FPGA Vendors & Device Families

- Xilinx
  - Virtex-II/Virtex-4: Feature- packed high-performance SRAM-based FPGA
  - Spartan 3: low-cost feature reduced version
  - CoolRunner: CPLDs
- Altera
  - Stratix/Stratix-II
    - High-performance SRAM-based FPGAs
  - Cyclone/Cyclone-II
    - Low-cost feature reduced version for cost-critical applications
  - MAX3000/7000 CPLDs
  - MAX-II: Flash-based FPGA

- Actel
  - Anti-fuse based FPGAs
    - Radiation tolerant
  - Flash-based FPGAs
- Lattice
  - Flash-based FPGAs
  - CPLDs (EEPROM)
- QuickLogic
  - ViaLink-based FPGAs

Special FPGA functions

- Internal SRAM
- Embedded Multipliers and DSP blocks
- Embedded logic analyzer
- Embedded CPUs
- High speed I/O (~10GHz)
- DDR/DDRII/DDRIII SDRAM interfaces
- PLLs



❖ RAM (Random Access Memory)

RAM (Random Access Memory) is the hardware in a computing device where the operating system (OS), application programs and data in current use are kept so they can be quickly reached by the device's processor.

RAM is the main memory in a computer, and it is much faster to read from and write to than other kinds of storage, such as a hard disk drive (HDD), solid-state drive (SSD) or optical drive.

Random Access Memory is volatile. That means data is retained in RAM as long as the computer is on, but it is lost when the computer is turned off.

When the computer is rebooted, the OS and other files are reloaded into RAM, usually from an HDD or SSD.

HOW IT WORKS

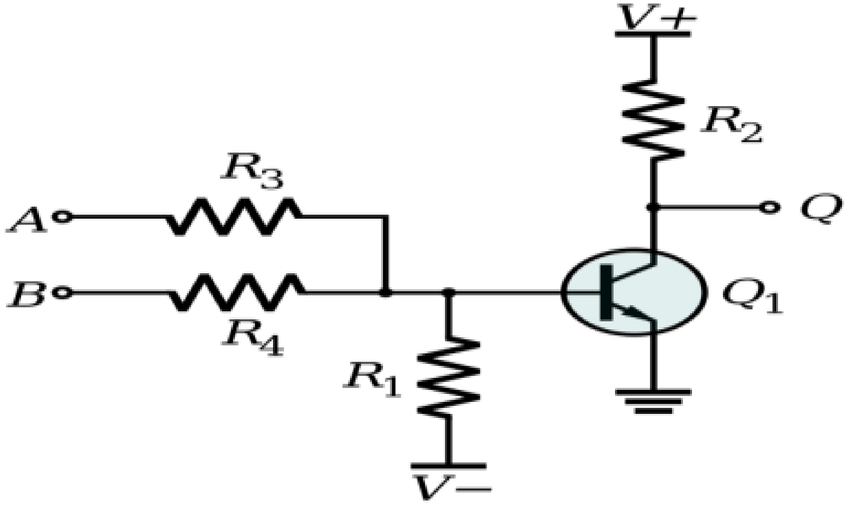
- The term *random access* as applied to RAM comes from the fact that any storage location, also known as any memory address, can be accessed directly. Originally, the term *Random Access Memory* was used to distinguish regular core memory from offline memory.
- Offline memory typically referred to magnetic tape from which a specific piece of data could only be accessed by locating the address sequentially, starting at the beginning of the tape. RAM is organized and controlled in a way that enables data to be stored and retrieved directly to and from specific locations.
- Other types of storage -- such as the hard drive and CD-ROM-- are also accessed directly or randomly, but the term *random access* isn't used to describe these other types of storage.
- RAM is similar in concept to a set of boxes in which each box can hold a 0 or a 1. Each box has a unique address that is found by counting across the columns and down the rows. A set of RAM boxes is called an array, and each box is known as a cell.
- To find a specific cell, the RAM controller sends the column and row address down a thin electrical line etched into the chip. Each row and column in a RAM array has its own address line. Any data that's read flows back on a separate data line.
- RAM is physically small and stored in microchips. It's also small in terms of the amount of data it can hold. A typical laptop computer may come with 8 gigabytes of RAM, while a hard disk can hold 10 terabytes.

TYPES OF RAM

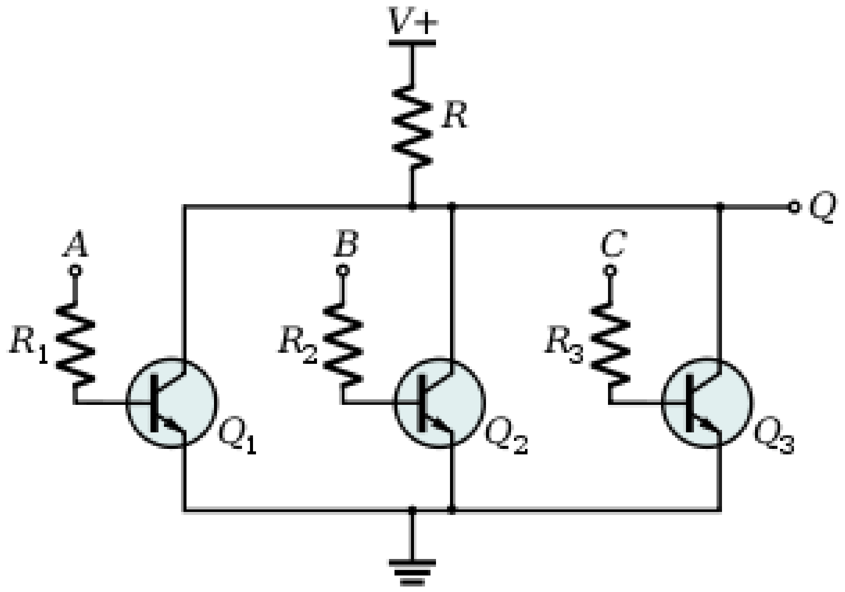
RAM comes in two primary forms:

- **Dynamic Random Access Memory (DRAM)** makes up the typical computing device's RAM and, as was previously noted, it needs that power to be on to retain stored data.
- Each DRAM cell has a charge or lack of charge held in an electrical capacitor. This data must be constantly refreshed with an electronic charge every few milliseconds to compensate for leaks from the capacitor. A transistor serves as a gate, determining whether a capacitor's value can be read or written.
- **Static Random Access Memory (SRAM)** also needs constant power to hold on to data, but it doesn't need to be continually refreshed the way DRAM does.
- In SRAM, instead of a capacitor holding the charge, the transistor acts as a switch, with one position serving as 1 and the other position as 0. Static RAM requires several transistors to retain one bit of data compared to dynamic RAM which needs only one transistor per bit. As a result, SRAM chips are much larger and more expensive than an equivalent amount of DRAM.

Resistor-transistor logic (RTL)



One-transistor RTL NOR gate



Multi-transistor RTL NOR gate

WORKING

- **Resistor–transistor logic** (**RTL**) (sometimes also **transistor–resistor logic** (**TRL**)) is a class of digital circuits built using resistors as the input network and bipolar junction transistors (BJTs) as switching devices. RTL is the earliest class of transistorized digital logic circuit used; other classes include diode–transistor logic (DTL) and transistor–transistor logic (TTL). RTL circuits were first constructed with discrete components, but in 1961 it became the first digital logic family to be produced as a monolithic integrated circuit.
- The logical operation OR is performed by applying consecutively the two arithmetic operations addition and comparison (the input resistor network acts as a parallel *voltage summer* with equally weighted inputs and the following common-emitter transistor stage as a *voltage comparator* with a threshold about 0.7 V). The equivalent resistance of all the resistors connected to logical "1" and the equivalent resistance of all the resistors connected to logical "0" form the two legs of a composed voltage divider driving the transistor. The base resistances and the number of the inputs are chosen (limited) so that only one logical "1" is sufficient to create base-emitter voltage exceeding the threshold and, as a result, saturating the transistor. If all the input voltages are low (logical "0"), the transistor is cut-off. The pull-down resistor R_1 biases the transistor to the appropriate on-off threshold. The output is inverted since the collector-emitter voltage of transistor Q_1 is taken as output, and is high when the inputs are low. Thus, the analog resistive network and the analog transistor stage perform the logic function NOR

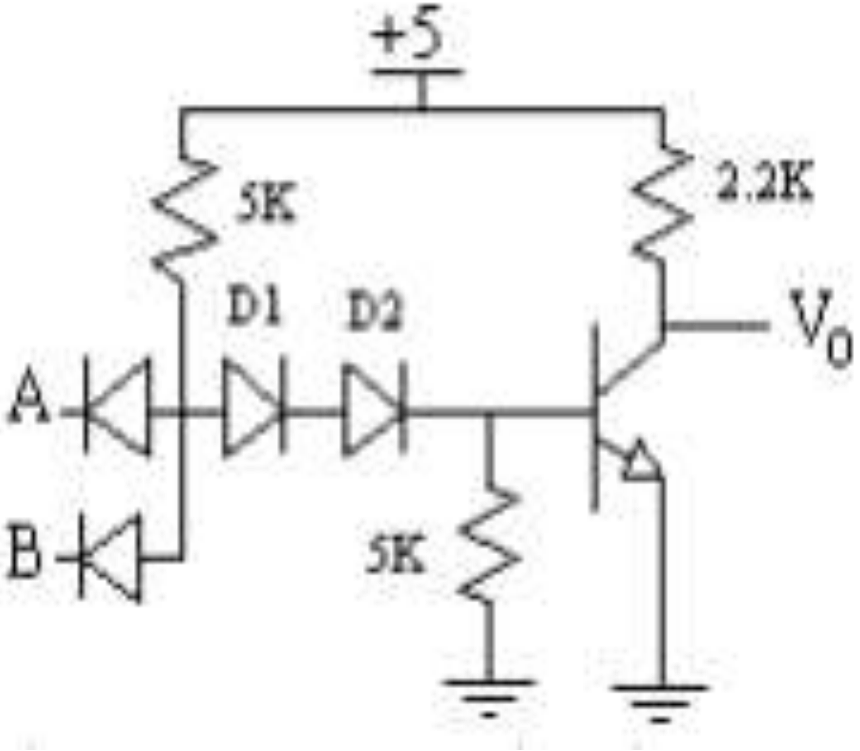
Advantages

- The primary advantage of RTL technology was that it used a minimum number of transistors. In circuits using discrete components, before integrated circuits, transistors were the most expensive component to produce. Early IC logic production (such as Fairchild's in 1961) used the same approach briefly, but quickly transitioned to higher-performance circuits such as diode–transistor logic and then transistor–transistor logic (starting in 1963 at Sylvania Electric Products), since diodes and transistors were no more expensive than resistors in the IC.

Limitations

- The disadvantage of RTL is its high power dissipation when the transistor is switched on, by current flowing in the collector and base resistors. This requires that more current be supplied to and heat be removed from RTL circuits. In contrast, TTL circuits with "totem-pole" output stage minimize both of these requirements.
- Another limitation of RTL is its limited fan-in: 3 inputs being the limit for many circuit designs, before it completely loses usable noise immunity.[*citation needed*] It has a low noise margin. Lancaster says that integrated circuit RTL NOR gates (which have one transistor per input) may be constructed with "any reasonable number" of logic inputs, and gives an example of an 8-input NOR gate.

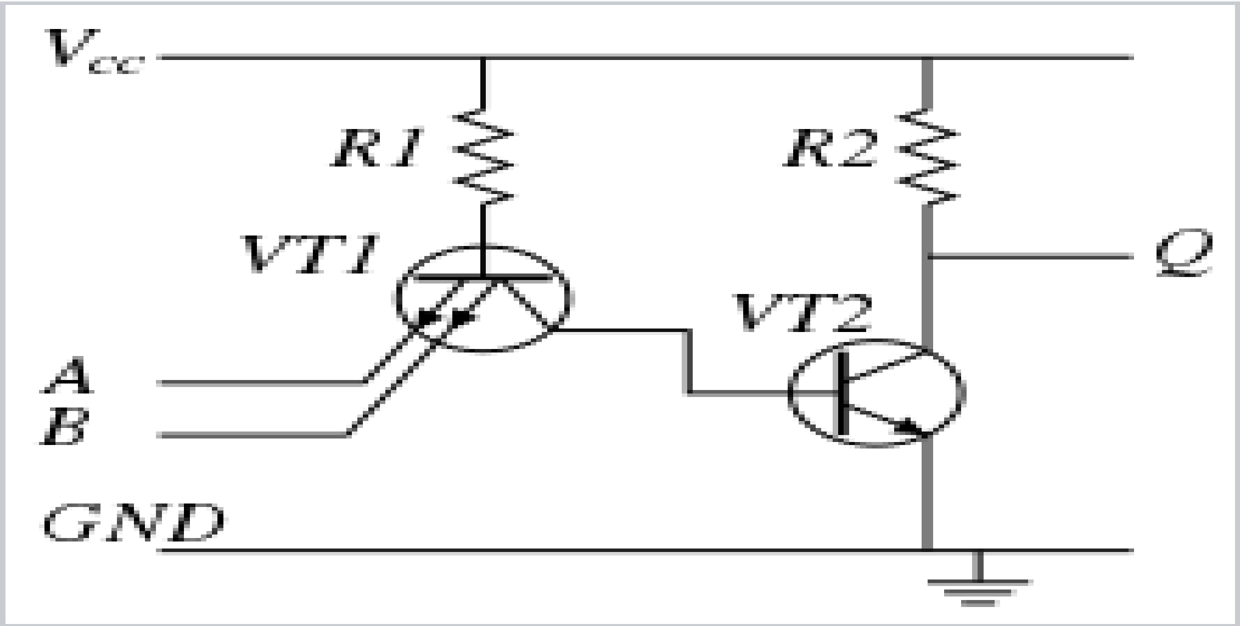
DIODE TRANSISTOR LOGIC CIRCUITS



DIODE TRANSISTOR LOGIC CIRCUITS

- DTL was initially made with discrete transistors and resistors before being integrated onto silicon.
- One early form of DTL, used by IBM Corp in the 360 family of computers, was really a hybrid technology.
- Transistor and diode chips were glued to a ceramic substrate and aluminum resistor paste was deposited on the substrate to make resistors.
- Finally the ceramic base and components were hermetically sealed in an aluminum can. This family was used extensively in IBM products in the middle to late 1960's.
- While this family was not a true integrated circuit, it was very successful and was less expensive than true integrated circuits for several years.
- By the early 1970's integrated circuits became quite common and DTL gave way to TTL which was more appropriate to integrated circuit technology.
- While DTL is no longer commercially used, we will discuss it because it is similar to and easier to understand than TTL, and because designers still find the configuration of value.
- First, however, we will discuss diode logic which is the front end of the DTL gate and performs the actual logic operation.

Transistor-Transistor logic (TTL)

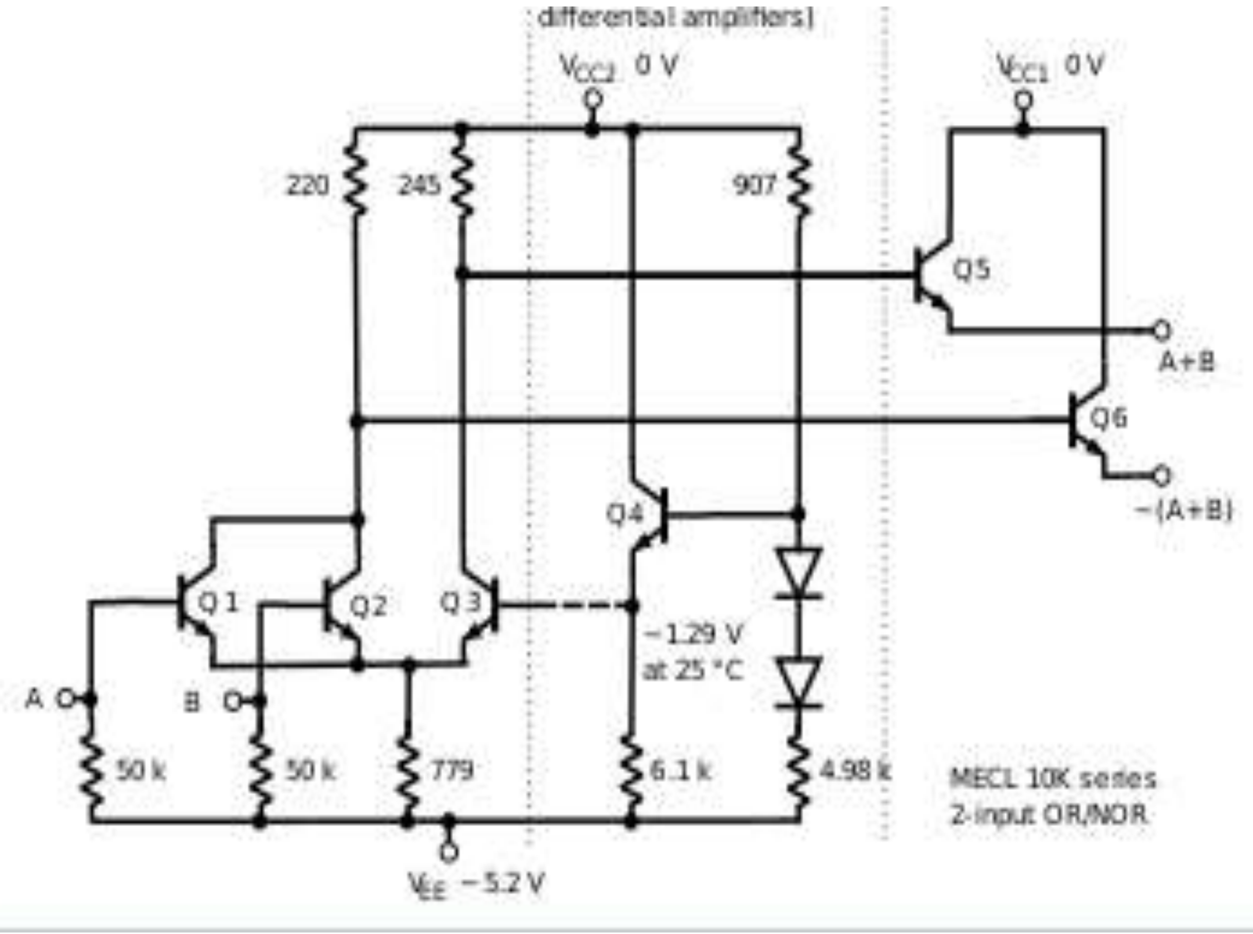


Two-input TTL NAND gate with a simple output stage (simplified)

Transistor–Transistor logic (TTL)

- Transistor–transistor logic (TTL) is a logic family built from bipolar junction transistors. Its name signifies that transistors perform both the logic function (the first "transistor") and the amplifying function (the second "transistor"), as opposed to resistor–transistor logic (RTL) or diode–transistor logic (DTL).
- TTL integrated circuits (ICs) were widely used in applications such as computers, industrial controls, test equipment and instrumentation, consumer electronics, and synthesizers. Sometimes TTL-compatible logic levels are not associated directly with TTL integrated circuits, for example, they may be used at the inputs and outputs of electronic instruments.
- After their introduction in integrated circuit form in 1963 by Sylvania Electric Products, TTL integrated circuits were manufactured by several semiconductor companies. The 7400 series by Texas Instruments became particularly popular. TTL manufacturers offered a wide range of logic gates, flip-flops, counters, and other circuits. Variations of the original TTL circuit design offered higher speed or lower power dissipation to allow design optimization. TTL devices were originally made in ceramic and plastic dual in-line package(s) and in flat-pack form. Some TTL chips are now also made in surface-mount technology packages.

EMITTER-COUPLED LOGIC (ECL)



WORKING

- The ECL circuit operation is considered below with assumption that the input voltage is applied to T1 base, while T2 input is unused or a logical "0" is applied.
- During the transition, the core of the circuit – the emitter-coupled pair (T1 and T3) – acts as a differential amplifier with single-ended input. The "long-tail" current source (R_E) sets the total current flowing through the two legs of the pair.
- The input voltage controls the current flowing through the transistors by sharing it between the two legs, steering it all to one side when not near the switching point.
- The gain is higher than at the end states (see below) and the circuit switches quickly.

Metal–Oxide–Semiconductor (MOS) Fundamentals

- The metal-oxide (SiO2)-semiconductor (Si) is the most common microelectronic structures nowadays. The two terminals of MOS-Capacitor consist of the main structures in MOS devices and it is the simplest structure of MOS devices. Therefore, it's essential to understand the mechanisms and characteristics of how MOS-C operates. The mechanisms under static biasing conditions can be visualized from two diagrams.

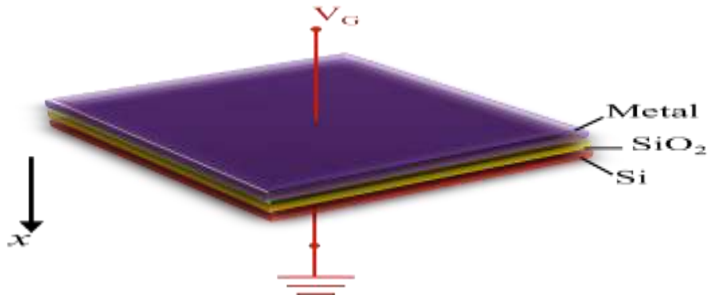
Energy band diagram

Block-charge diagram

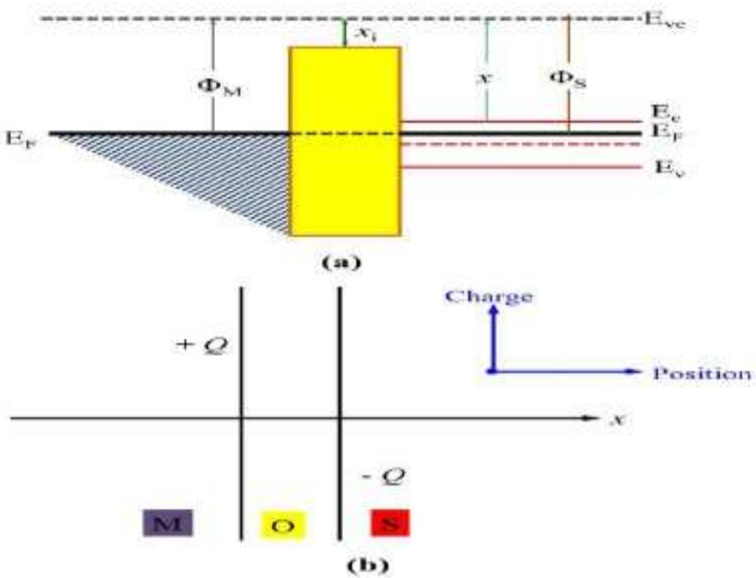
The characteristics of MOS-C can be visualized by C-V (Capacitance verses Voltage) curves.

- **Introduction**
- The principals of forming MOS structure are similar to the metal-semiconductor (MS) contact structures, but the MOS structure is like sandwich structures which have a thin layer of silicon oxides in the middle between metal and semiconductor (Si) layer. Figure 1 below shows a schematic of an ideal MOS-C device. For an ideal MOS-C structure, some properties should follow below.

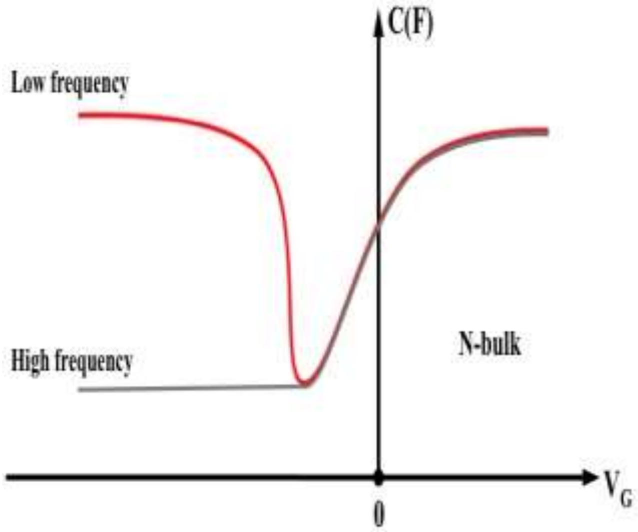
The metallic gate should thick enough to be equipotential region, where every points has the same potential in the space, under a.c and d.c biasing conditions. The oxides layer in the middle should be a perfect insulator with zero current flowing through under all static biasing conditions. There should be no charge centers located on the oxide-semiconductor interface. The semiconductor should be uniformly doped with donors or acceptors as p-type or n-type semiconductors. The semiconductor (Si) should be thick enough for charges to encounter a field free region (Si bulk) before reaching the back contact. The Ohmic contacts should be established on the backside of the MOS device.



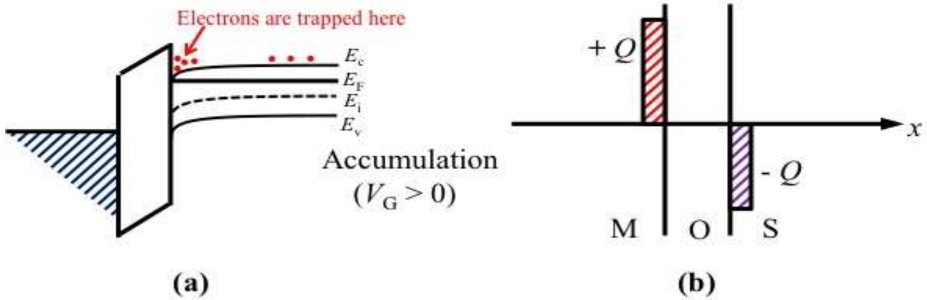
The schematic of an ideal MOS-C device
Energy Band and Block Charge Diagrams



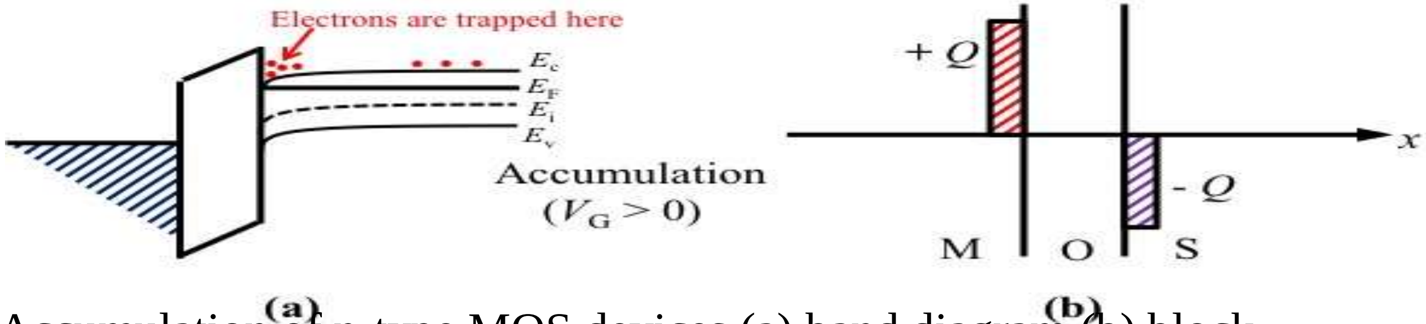
The flat band diagram of MOS-C in equilibrium with n-type semiconductor, (b) the block charge diagrams of flat band MOS-C.



Capacitance verses voltage of MOS-C device for n-type semiconductor.

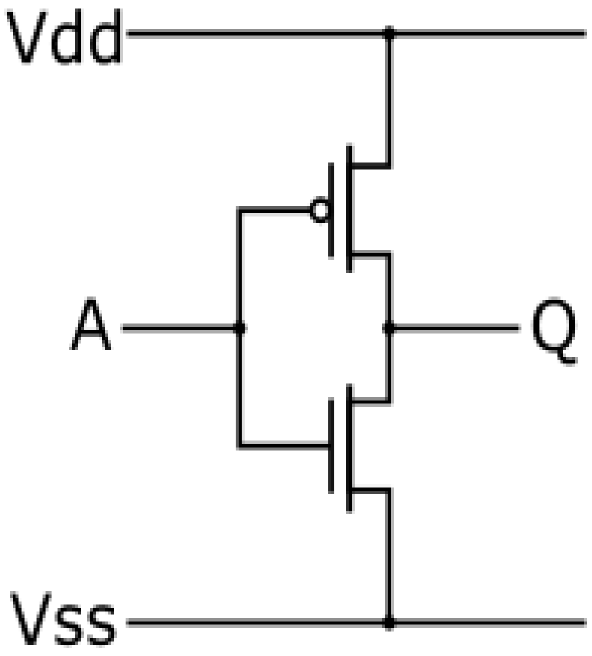


Accumulation of n-type MOS devices (a) band diagram (b) block charge diagram



Accumulation of n-type MOS devices (a) band diagram (b) block charge diagram

Complementary metal–oxide–semiconductor (CMOS)



CMOS Inverter

WORKING

- CMOS circuits are constructed in such a way that all P-type metal–oxide–semiconductor (PMOS) transistors must have either an input from the voltage source or from another PMOS transistor.
- Similarly, all NMOS transistors must have either an input from ground or from another NMOS transistor. The composition of a PMOS transistor creates low resistance between its source and drain contacts when a low gate voltage is applied and high resistance when a high gate voltage is applied.
- On the other hand, the composition of an NMOS transistor creates high resistance between source and drain when a low gate voltage is applied and low resistance when a high gate voltage is applied.
- CMOS accomplishes current reduction by complementing every NMOSFET with a PMOSFET and connecting both gates and both drains together. A high voltage on the gates will cause the NMOSFET to conduct and the PMOSFET not to conduct, while a low voltage on the gates causes the reverse.
- This arrangement greatly reduces power consumption and heat generation. However, during the switching time, both MOSFETs conduct briefly as the gate voltage goes from one state to another.
- This induces a brief spike in power consumption and becomes a serious issue at high frequencies.

Comparison of Logic Families

FAMILY	DESCRIPTION	PRPOGATION DELAY (ns)	TOGGLE SPEED (MHZ)	POWER PER GATE @ 1 MHZ (mw)	TYPICAL SUPPLY VOLTAGE RANGE	INTRODUCTION YEAR	REMARKS
CMOS	AC/ACT	3	125	0.5	3.3 or 5 (2–6 or 4.5–5.5)	1985	ACT has TTL compatible levels
CMOS	HC/HCT	9	50	0.5	5 (2–6 or 4.5–5.5)	1982	HCT has TTL compatible levels
CMOS	4000B/74C	30	5	1.2	10V (3–18)	1970	Approximately half speed and power at 5 volts
DTL	Diode-transistor logic	25		10	5	1962	Introduced by Signetics, Fairchild 930 line became industry standard in 1964
ECL	ECL III	1	500	60	−5.2(−5.19 – −5.21)	1968	Improved ECL
ECL	MECL I	8		31	-5.2	1962	first integrated logic circuit commercially produced
ECL	ECL 10K	2	125	25	−5.2(−5.19 – −5.21)	1971	Motorola
ECL	ECL 100K	0.75	350	40	−4.5(−4.2 – −5.2)	1981	
ECL	ECL 100KH	1	250	25	−5.2(−4.9 – −5.5)	1981	
PMOS	MEM 1000	300	1	9	-27 and -13	1967	Introduced by General Instrument
RTL	Resistor-transistor logic	500	4	10	3.3	1963	the first CPU built from integrated circuits (the Apollo Guidance Computer) used RTL.
TTL	Original series	10	25	10	5 (4.75–5.25)	1964	Several manufacturers
TTL	L	33	3	1	5 (4.75–5.25)	1964	Low power
TTL	H	6	43	22	5 (4.75–5.25)	1964	High speed

TTL	S	3	100	19	5 (4.75–5.25)	1969	Schottky high speed
TTL	LS	10	40	2	5 (4.75–5.25)	1976	Low power Schottky high speed
TTL	ALS	4	50	1.3	5 (4.5–5.5)	1976	Advanced Low power Schottky
TTL	F	3.5	100	5.4	5 (4.75–5.25)	1979	Fast
TTL	AS	2	105	8	5 (4.5–5.5)	1980	Advanced Schottky
TTL	G	1.5	1125 (1.125 GHz)		1.65 – 3.6	2004	First GHz 7400 series logic
TTL	Original series	10	25	10	5 (4.75–5.25)	1964	Several manufacturers
TTL	L	33	3	1	5 (4.75–5.25)	1964	Low power
TTL	H	6	43	22	5 (4.75–5.25)	1964	High speed
TTL	S	3	100	19	5 (4.75–5.25)	1969	Schottky high speed
TTL	LS	10	40	2	5 (4.75–5.25)	1976	Low power Schottky high speed
TTL	ALS	4	50	1.3	5 (4.5–5.5)	1976	Advanced Low power Schottky
TTL	F	3.5	100	5.4	5 (4.75–5.25)	1979	Fast
TTL	AS	2	105	8	5 (4.5–5.5)	1980	Advanced Schottky
TTL	G	1.5	1125 (1.125 GHz)		1.65 – 3.6	2004	First GHz 7400 series logic
RTL	Resistor–transistor logic	500	4	10	3.3	1963	the first CPU built from integrated circuits (the Apollo Guidance Computer) used RTL.
PMOS	MEM 1000	300	1	9	-27 and -13	1967	Introduced by General Instrument
ECL	ECL III	1	500	60	-5.2(-5.19 – -5.21)	1968	Improved ECL
ECL	MECL I	8		31	-5.2	1962	first integrated logic circuit commercially produced