

BECE102L

Digital Systems Design

Faculty-Dr.Deepika Rani Sona

Assistant Prof.Senior

Dept.of Embedded Technology

School of Electronics Engineering

FALL SEMESTER 2023-24

Open Hours

Tue:11:00AM-12:00 Noon

Wed:10:30AM-11:30AM

Digital Systems Design

L T P C
3 0 0 3

OUTLINE:

- Course Objective & Outcomes
- Syllabus
- Evaluation Procedure/Rubrics
- Introduction to Digital Logic
- Boolean Algebra: Basic definitions

Course Objective

1. Provide an understanding of Boolean algebra and logic functions.
2. Develop the knowledge of combinational and sequential logic circuit design.
3. Design and model the data path circuits for digital systems.
4. Establish a strong understanding of programmable logic.
5. Enable the student to design and model the logic circuits using Verilog HDL.

Course Outcome

1. Optimize the logic functions using Boolean principles and K-map
2. Model the Combinational and Sequential logic circuits using Verilog HDL
3. Design the various combinational logic circuits and data path circuits
4. Analyze and apply the design aspects of sequential logic circuits
5. Analyze and apply the design aspects of Finite state machines
6. Examine the basic architectures of programmable logic devices

SYLLABUS

Module:1	Digital Logic	8 hours
Boolean Algebra: Basic definitions, Axiomatic definition of Boolean Algebra, Basic Theorems and Properties of Boolean Algebra, Boolean Functions, Canonical and Standard Forms, Simplification of Boolean functions. Gate-Level Minimization: The Map Method (K-map up to 4 variable), Product of Sums and Sum of Products Simplification, NAND and NOR Implementation. Logic Families: Digital Logic Gates, TTL and CMOS logic families.		
Module:2	Verilog HDL	5 hours
Lexical Conventions, Ports and Modules, Operators, Dataflow Modelling, Gate Level Modelling, Behavioural Modeling, Test Bench		
Module:3	Design of Combinational Logic Circuits	8 hours
Design Procedure, Half Adder, Full Adder, Half Subtractor, Full Subtractor, Decoders, Encoders, Multiplexers, De-multiplexers, Parity generator and checker, Applications of Decoder, Multiplexer and De-multiplexer. Modeling of Combinational logic circuits using Verilog HDL.		
Module:4	Design of data path circuits	6 hours
N-bit Parallel Adder/Subtractor, Carry Look Ahead Adder, Unsigned Array Multiplier, Booth Multiplier, 4-Bit Magnitude comparator. Modeling of data path circuits using Verilog HDL		

Module:5	Design of Sequential Logic Circuits	8 hours
Latches, Flip-Flops - SR, D, JK & T, Buffer Registers, Shift Registers - SISO, SIPO, PISO, PIPO, Design of synchronous sequential circuits: state table and state diagrams, Design of counters: Modulo-n, Johnson, Ring, Up/Down, Asynchronous counter. Modeling of sequential logic circuits using Verilog HDL		
Module:6	Design of FSM	4 hours
Finite state Machine(FSM):Mealy FSM and Moore FSM , Design Example : Sequence detection, Modeling of FSM using Verilog HDL		
Module:7	Programmable Logic Devices	4 hours
Types of Programmable Logic Devices: PLA, PAL, CPLD, FPGA Generic Architecture		
Module:8	Contemporary issues	2 hours
Guest lecture from Industries and R & D Organizations		
Total Lecture hours:		45 hours

Textbook(s)	
1.	M. Morris Mano and Michael D. Ciletti, Digital Design: With an Introduction to the Verilog HDL and System Verilog, 2018, 6 th Edition, Pearson Pvt. Ltd.
Reference Books	
1.	Ming-Bo Lin, Digital Systems Design and Practice: Using Verilog HDL and FPGAs, 2015, 2nd Edition, Create Space Independent Publishing Platform.
2.	Samir Palnitkar, Verilog HDL: A Guide to Digital Design and Synthesis, 2009, 2nd edition, Prentice Hall of India Pvt. Ltd.
3.	Stephen Brown and ZvonkoVranesic, Fundamentals of Digital Logic with Verilog Design, 2013, 3rd Edition, McGraw-Hill Higher Education.

Evaluation Procedure/Rubrics

Components	Marks	Weightage
CAT-1	50	15
CAT-2	50	15
QUIZ-1(Before CAT-1-Tentative)	10	10
QUIZ-2(After CAT-2-Tentative)	10	10
DIGITAL ASSIGNMENT-1	10	10
FAT	100	40
	TOTAL	100

Laws of Boolean Algebra

- **Commutative Law**
- (a) $A + B = B + A$
- (b) $A \cdot B = B \cdot A$
- **Associate Law**
- (a) $(A + B) + C = A + (B + C)$
- (b) $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
- **Distributive Law**
- (a) $A \cdot (B + C) = A \cdot B + A \cdot C$
- (b) $A + (B \cdot C) = (A + B) \cdot (A + C)$
- **Identity Law**
- (a) $A + A = A$
- (b) $A \cdot A = A$
- **Redundance Law**
- (a) $A + A \cdot B = A$
- (b) $A \cdot (A + B) = A$

Distributive Law

*These DO NOT hold good in ordinary algebra
but are true in Boolean algebra.*

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\bar{A} = 0$	$A + \bar{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}\bar{B}$

Implementation of Boolean Functions using Logic Gates.

Using the theorems of Boolean Algebra, the algebraic forms of functions can often be simplified, which leads to simpler (and cheaper) implementations.



Simplification of Boolean Functions:

- An implementation of a Boolean Function requires the use of logic gates.
- A smaller number of gates, with each gate (other than Inverter) having less number of inputs, may reduce the cost of the implementation.
- There are 2 methods for simplification of Boolean functions.

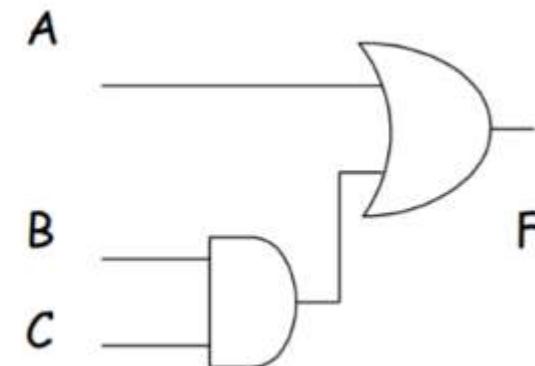


Simplification of Boolean Functions: Two Methods

- The algebraic method by using Identities
- The graphical method by using Karnaugh Map method
- The K-map method is easy and straightforward.
- A K-map for a function of n variables
 - consists of 2^n cells, and,
 - in every row and column, two adjacent cells should differ in the value of only one of the logic variables.

EXAMPLE 1:

$$\begin{aligned} F &= A \cdot \overline{B} + A \cdot B + B \cdot C \\ &= A \cdot (\overline{B} + B) + B \cdot C \\ &= A \cdot 1 + B \cdot C \\ &= A + B \cdot C \end{aligned}$$



EXAMPLE 2:

Show that $A + A \cdot B = A$

$$\begin{aligned} & A + AB \\ = & A \cdot 1 + A \cdot B \\ = & A \cdot (1 + B) \\ = & A \cdot 1 \\ = & A \end{aligned}$$

$$\begin{aligned} (A + B)(A + C) &= AA + AC + AB + BC \\ &= A + AC + AB + BC \\ &= A(1 + C + B) + BC \\ &= A \cdot 1 + BC \\ &= A + BC \end{aligned}$$

$$\begin{aligned} A\bar{C} + AB\bar{C} &= A\bar{C} (1 + B) \\ &= A\bar{C} (1) \\ &= A\bar{C} \end{aligned}$$

$$\begin{aligned} A + BC &= A \cdot 1 + BC && (\because A \cdot 1 = 1) \\ &= A(1 + B) + BC && (\because 1 + B = 1) \\ &= A \cdot 1 + AB + BC && (\because A(B + C) = AB + BC) \\ &= A \cdot (1 + C) + AB + BC && (\because 1 + C = 1) \\ &= A \cdot 1 + AC + AB + BC \\ &= A \cdot A + AC + AB + BC && (\because A \cdot A = A) \\ &= A(A + C) + B(A + C) \\ &= (A + B)(A + C) \end{aligned}$$

SOP Boolean Function Implementation using Logic Gates

Sum of Products (SOP)	
Input	AND
Output	OR

Product

A
B

B

C
D

D

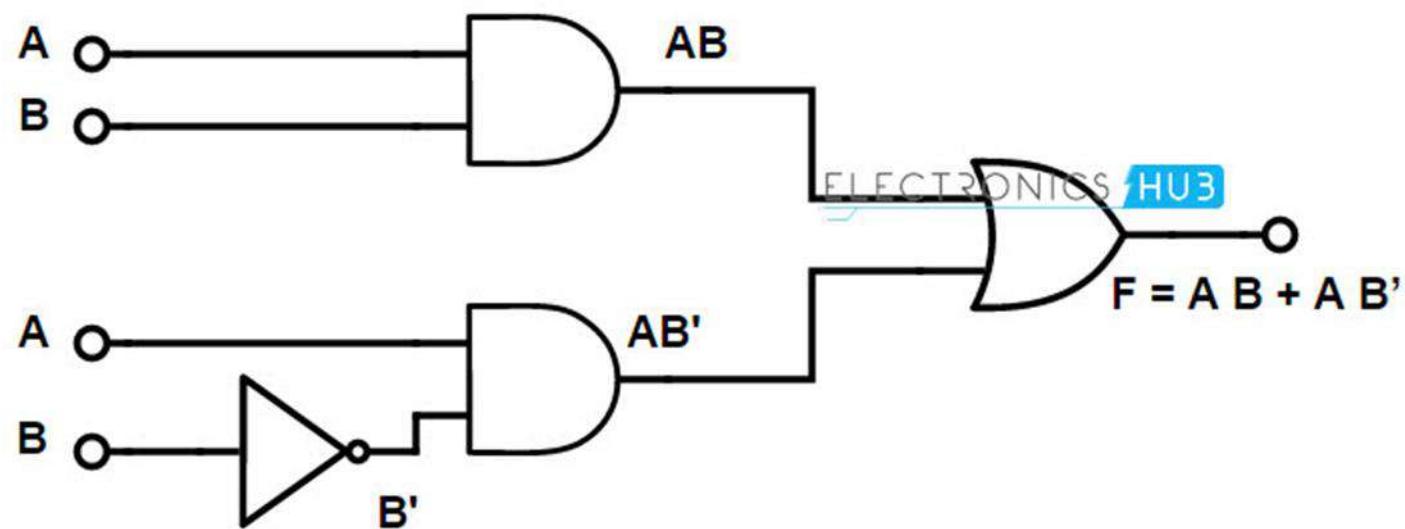
Product

Sum

$$Y = AB + CD \quad \text{SOP}$$

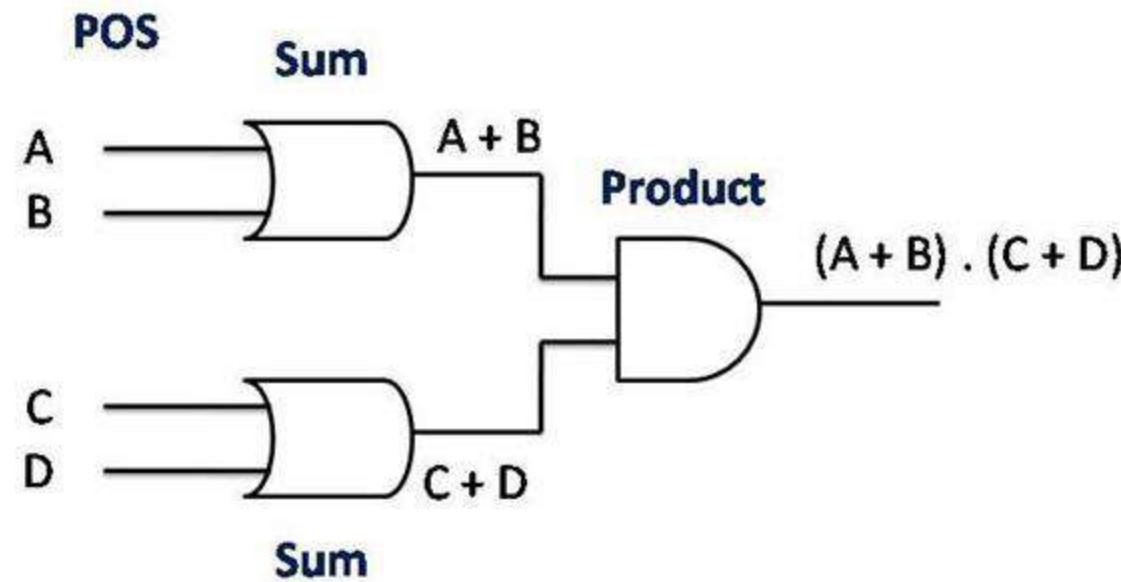
SOP

SOP

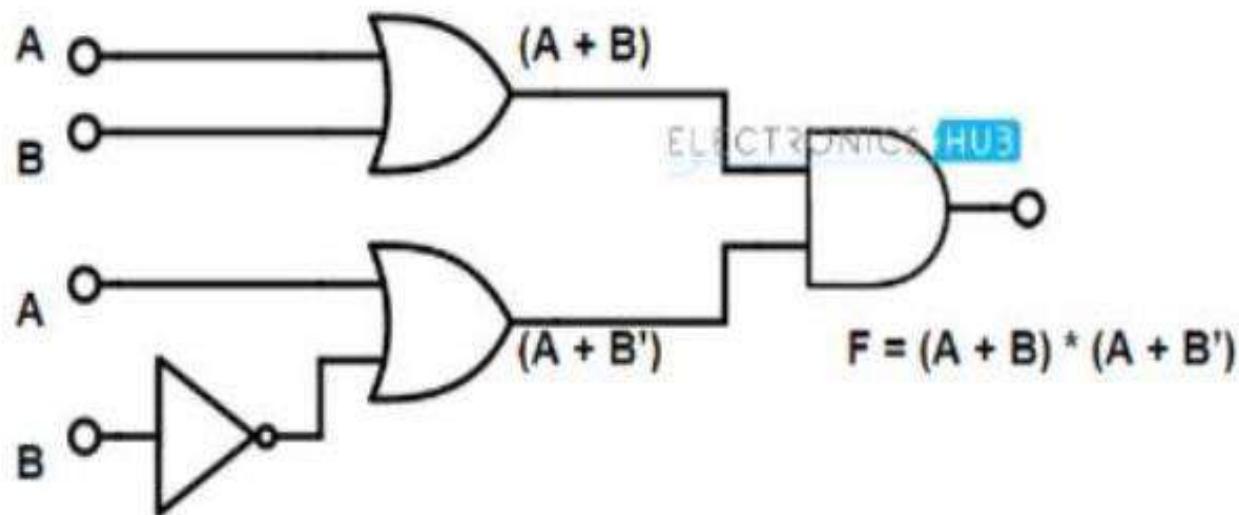


POS Boolean Function Implementation using Logic Gates

Product of Sums (POS)	
Input	OR
Output	AND



$$F = (A + B) \cdot (A + \bar{B})$$



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

Module - I

BOOLEAN ALGEBRA

FALL 23-24

Courtesy: M. Morris Mano, "Digital Design", 5th Edition, *Prentice Hall of India Pvt. Ltd.*, 2008 and its "e-book" version.

BOOLEAN ALGEBRA

Contents

- Basic Definitions, Axiomatic Definition of Boolean Algebra
- Basic Theorems and Properties of Boolean Algebra,
- Boolean Functions
- Canonical and Standard Forms
- Digital Logic Gates, timing concepts

Basic definitions

- Boolean algebra is defined with a set of elements, a set of operators, and a number of axioms and postulates

Postulates

- **Closure:** A set S is closed with respect to a binary operator if, for every pair of elements of S , the binary operator specifies for obtaining an unique element of S .

Basic definitions

- **Associative law:** A binary operator * on a set **S** is said to **be associative** whenever
$$(x * y) * z = x * (y * z)$$
- **Commutative law:** A binary operator * on on a set S is said lo **be commutative** when
$$x * y = y * x$$
- **Identity element:** A set S is said to have identity element with respect to the binary operator * on S if there exists an element $e \in S$ with property that
$$e * x = x * e = x \text{ for any } x$$

Basic definitions

- **Inverse:** A set S having the identity element e with respect to a binary operator $*$ is said to have an inverse whenever for every $x \in S$, there exists an element $y \in S$ such that $x * y = e$
- **Distributive law:** If $*$ and $.$ Are two binary operators on set S , $*$ is said to be distributive over $.$ Whenever

$$x^*(y.z) = (x^*y).(x^*z)$$

Postulates and theorems of Boolean algebra

Postulates and Theorems of Boolean Algebra

Postulate 2	(a) $x + 0 = x$	(b) $x \cdot 1 = x$
Postulate 5	(a) $x + x' = 1$	(b) $x \cdot x' = 0$
Theorem 1	(a) $x + x = x$	(b) $x \cdot x = x$
Theorem 2	(a) $x + 1 = 1$	(b) $x \cdot 0 = 0$
Theorem 3, involution	$(x')' = x$	
Postulate 3, commutative	(a) $x + y = y + x$	(b) $xy = yx$
Theorem 4, associative	(a) $x + (y + z) = (x + y) + z$	(b) $x(yz) = (xy)z$
Postulate 4, distributive	(a) $x(y + z) = xy + xz$	(b) $x + yz = (x + y)(x + z)$
Theorem 5, DeMorgan	(a) $(x + y)' = x'y'$	(b) $(xy)' = x' + y'$
Theorem 6, absorption	(a) $x + xy = x$	(b) $x(x + y) = x$

Boolean functions

- A Boolean function is described by an algebraic expression consisting of binary variables, the constants 0 and 1, and the logic operation symbols.
- A Boolean function can be represented in a truth table.
- The number of rows in the truth table is 2^n , where n is the number of variables in the function.

Boolean functions

Consider function F

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

The truth table for the function is given as

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

Minterms and Maxterms

- For n variables, there are 2^n AND and 2^n OR combinations possible.
- Each of the AND term is a minterm or standard product
- Each of the OR term is a maxterm or standard sums

Eg: variables x, y

Minterms: $xy, x'y, xy', x'y'$.

Maxterms: $x+y, x+y', x'+y, x'+y'$

Minterms and Maxterms

Minterms and maxterms for three binary variables

			Minterms		Maxterms	
x	y	z	Term	Designation	Term	Designation
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

Canonical form

- A function expressed as sum of minterms (SOP) or product of max terms (POS) is said to be in canonical form.
- To convert a Boolean function into canonical form, identity element theorem has to be used.

Canonical form

Eg: Express the function $F(x,y,z) = x+y'z$ in canonical form.

- The first term is not a minterm. So, convert it into a minterm

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

- The second term is not a minterm. So, convert it into a minterm

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

$$\text{Finally } F(x,y,z) = x+y'z$$

$$\begin{aligned} &= xyz + xyz' + xy'z + xy'z' + xy'z + x'y'z \\ &= m_1 + m_4 + m_5 + m_6 + m_7 \end{aligned}$$

Canonical form

Eg Express the function $F(x,y,z) = xy + x'z$ in POS form.

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

$$\begin{aligned}x' + y &= x' + y + zz' = (x' + y + z)(x' + y + z') \\x + z &= x + z + yy' = (x + y + z)(x + y' + z) \\y + z &= y + z + xx' = (x + y + z)(x' + y + z)\end{aligned}$$

$$\begin{aligned}F &= (x + y + z)(x + y' + z)(x' + y + z)(x' + y + z') \\&= M_0M_2M_4M_5\end{aligned}$$

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

Standard Form

- In this, the Boolean function may contain any number of literals (no stringent rules as in the case of canonical form)

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

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

The above two functions are in standard form but not in canonical form.

Standard Form

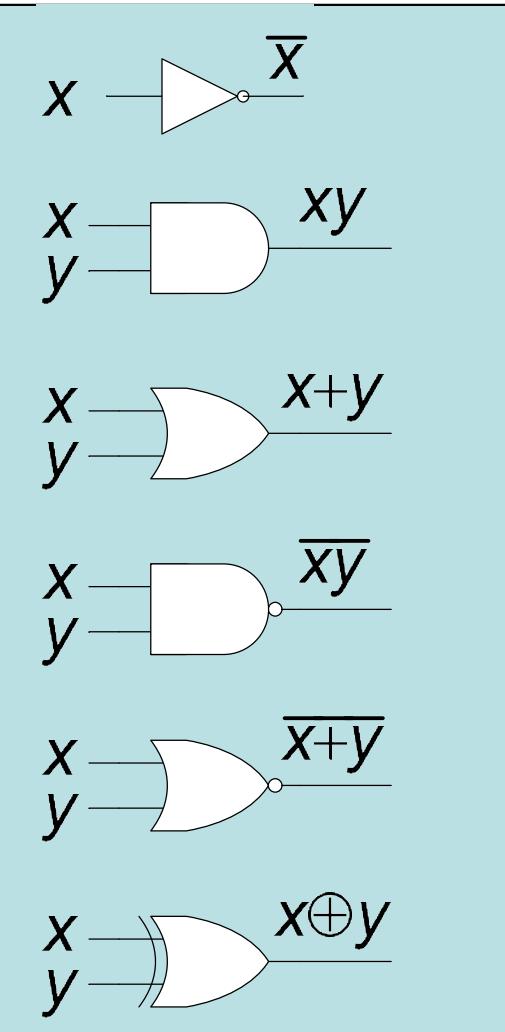
Two types of standard forms :

- a. Sum of products
- b. Product of sum

The only difference is that, the terms may or may not contain all the literals

Logic gates

NOT



VIT UNIVERSITY

GATE LEVEL MINIMIZATION

Courtesy: M. Morris Mano, "Digital Design", 3rd Edition, *Prentice Hall of India Pvt. Ltd.*, 2008 and its "e-book" version.

Gate-level Minimization

- 2-1 The Map Method Two-variable map and Three-variable map
- 2-2 Four-Variable Map
- 2-3 Product of Sums Simplification
- 2-4 Don't-care Conditions
- 2-5 NAND and NOR Implementation
- 2-6 Other Two-Level Implementations

The Map Method

- Simplification of Boolean Expression
 - Minimum # of terms, minimum # of literals
 - To reduce complexity of digital logic gates
 - The simplest expression is not unique
- Methods:
 - Algebraic minimization \Rightarrow lack of specific rules
 - Karnaugh map or K-map
 - Combination of 2, 4, ... adjacent squares

Logic circuit \Leftrightarrow Boolean function \Leftrightarrow Truth table \Leftrightarrow **K-map**
 \Leftrightarrow Canonical form (sum of minterms, product of maxterms)
 \Leftrightarrow **(Simplified) standard form** (sum of products, product of sums)

Two-variable Map

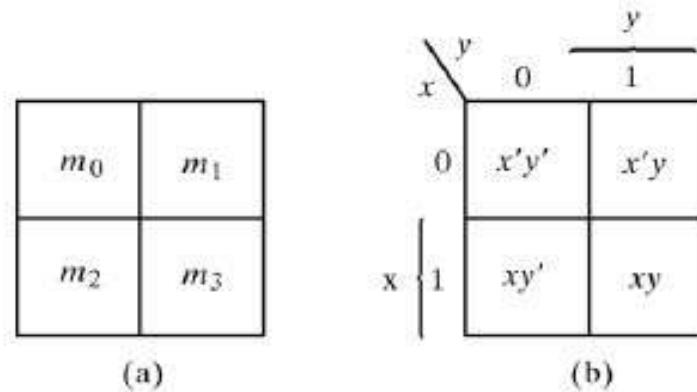


Fig. 3-1 Two-variable Map

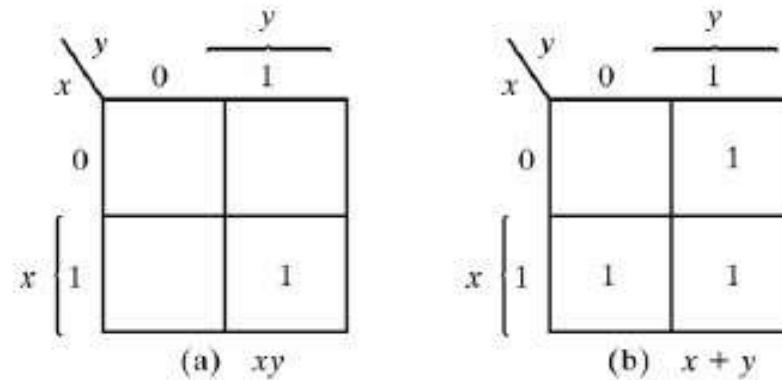


Fig. 3-2 Representation of Functions in the Map

$$m_1 + m_2 + m_3 = x'y' + xy' + xy = x + y$$

Three-Variable Map

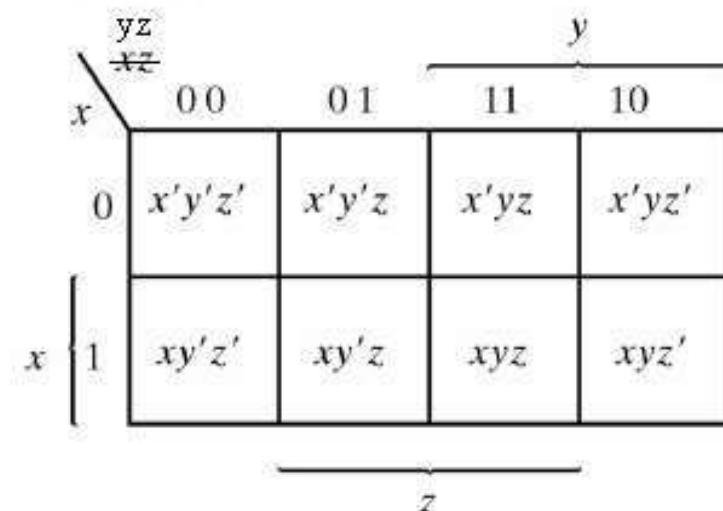
- 8 minterms for 3 binary variables
- Any two adjacent squares differ by only one variable (Gray code; wrap)

$$m_0 + m_2 = x'y'z' + x'yz' = x'z'(y' + y) = x'z'$$

$$m_4 + m_6 = xy'z' + xyz' = xz' + (y' + y) = xz'$$

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6

(a)



(b)

Examples

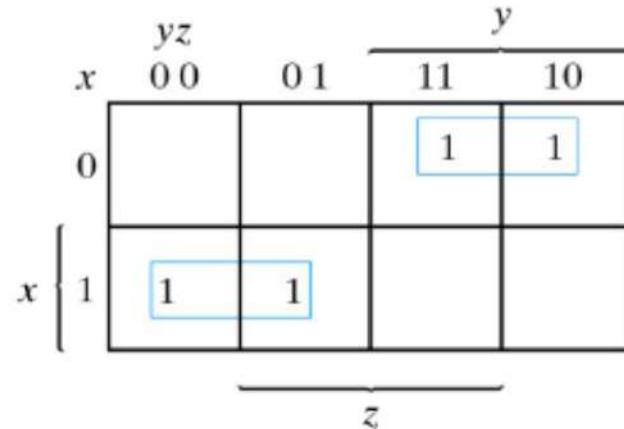


Fig. 3-4 Map for Example 3-1; $F(x, y, z) = \Sigma(2, 3, 4, 5) = x'y + xy'$

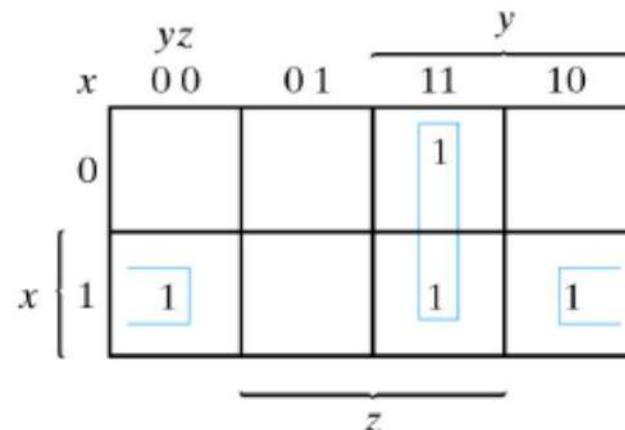
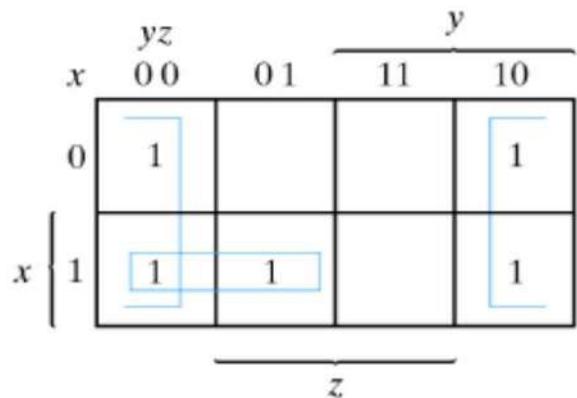
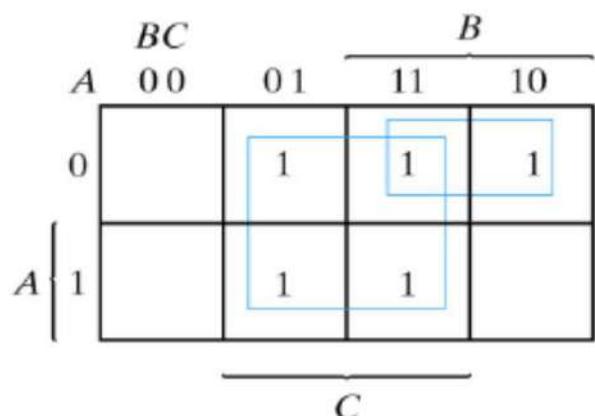


Fig. 3-5 Map for Example 3-2; $F(x, y, z) = \Sigma(3, 4, 6, 7) = yz + xz'$

Examples



Map for Example 3-3; $F(x, y, z) = \Sigma(0, 2, 4, 5, 6) = z' + xy'$



Map for Example 3-4; $A'C + A'B + AB'C + BC = C + A'B$

One square represents one minterm, giving a term of three literals

- Two adjacent squares represent a term of two literals
- Four adjacent squares represent a term of one literal

Four-Variable Map

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

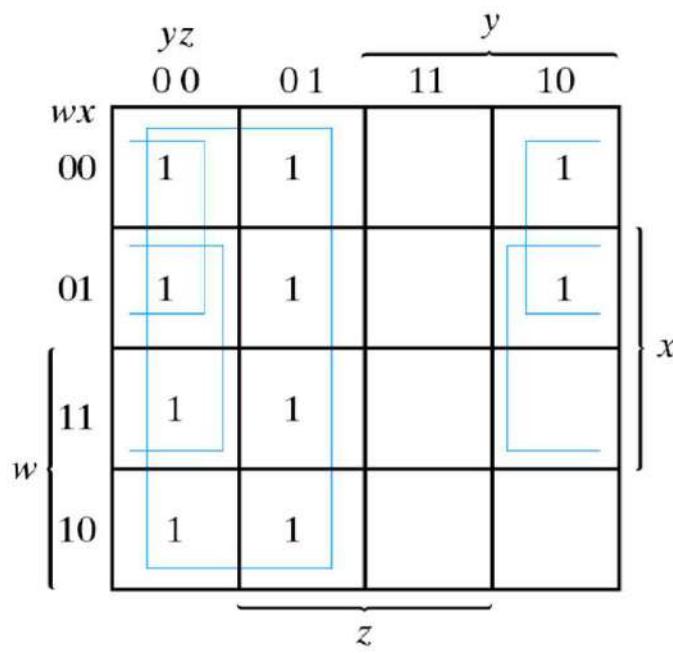
(a)

yz		y	
0 0		0 1	1 1
wx	w	x	z
w'x'y'z'	w'x'y'z	w'x'yz	w'x'yz'
w'xy'z'	w'xy'z	w'xyz	w'xyz'
wxy'z'	wxy'z	wxyz	wxyz'
wx'y'z'	wx'y'z	wx'yz	wx'yz'

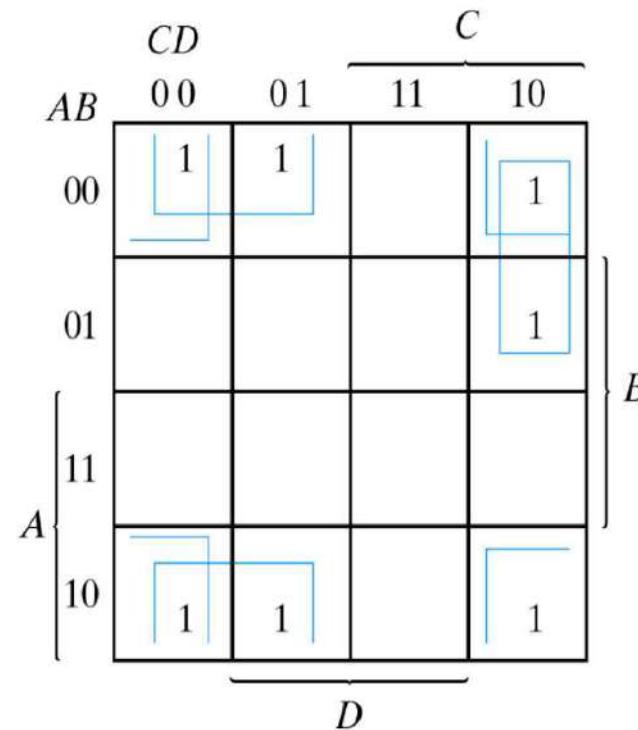
(b)

- Two adjacent squares represent a term of three literals
 - Four adjacent squares represent a term of two literals
 - Eight adjacent squares represent a term of one literal
- The larger the number of squares combined, the smaller the number of literals in the term

Examples



Map for Example 3-5; $F(w, x, y, z)$
 $= \Sigma (0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14) = y' + w'z' + xz'$



Map for Example 3-6; $A'B'C + B'CD' + A'BCD'$
 $+ AB'C' = B'D' + B'C' + A'CD'$

Simplification Using Prime Implicants

- **prime implicant**: a product term obtained by combining the maximum possible number of adjacent squares in the map
 - A single 1 on a map represents a prime implicant if it is not adjacent to any other 1's
 - Two adjacent 1's form a prime implicant, provided that they are not within a group of four adjacent squares
 - Four adjacent 1's form a prime implicant, provided that they are not within a group of eight adjacent squares
 - and so on
- If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be **essential**

		CD		C		
		00	01	11	10	
AB		00				1
A	01		1	1		
	11		1	1		
B	10	1				1
			D			

(a) Essential prime implicants
BD and B'D'

		CD		C		
		00	01	11	10	
AB		00	1			1
A	01		1	1		
	11		1	1		
B	10	1	1	1		1
			D			

(b) Prime implicants CD, B'C
AD, and AB'
(covering m₁₁)

$$\begin{aligned}
 F &= BD + B'D' + CD + AD \\
 &= BD + B'D' + CD + AB' \\
 &= BD + B'D' + B'C + AD \\
 &= BD + B'D' + B'C + AB'
 \end{aligned}$$

First determine all the essential prime implicants

- prime implicant: combining max # of adjacent squares
- essential prime implicant: containing a minterm that is covered by only one prime implicant

The simplified expression is obtained from

- the logical sum of all the essential prime implicants
- plus other prime implicants that may be needed to cover any remaining minterms not covered by the essential prime implicants

Product of Sums Simplification

- Boolean functions can be expressed in sum of products or product of sums

- Recall from Table 2-4

$$\begin{aligned}f_1 &= m_1 + m_4 + m_7 = M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6 \\&= x'y'z + xy'z' + xyz \\&= (x+y+z)(x+y'+z)(x+y'+z')(x'+y+z')(x'+y'+z) \\&= \Sigma(1, 4, 7) = \Pi(0, 2, 3, 5, 6)\end{aligned}$$

$$\begin{aligned}f_1' &= m_0 + m_2 + m_3 + m_5 + m_6 = M_1 \cdot M_4 \cdot M_7 \\&= x'y'z' + x'yz' + x'yz + xy'z + xyz' \\&= (x+y+z')(x'+y+z)(x'+y'+z') \\&= \Sigma(0, 2, 3, 5, 6) = \Pi(1, 4, 7)\end{aligned}$$

- Complement of f' (by DeMorgan's Theorem)
- Simplifying Boolean function in product of sums
 1. Derive f' in sum of products from the map
 2. Complement of f'

Example

Simply Boolean function $F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10)$ in (a) sum of products and (b) product of sums

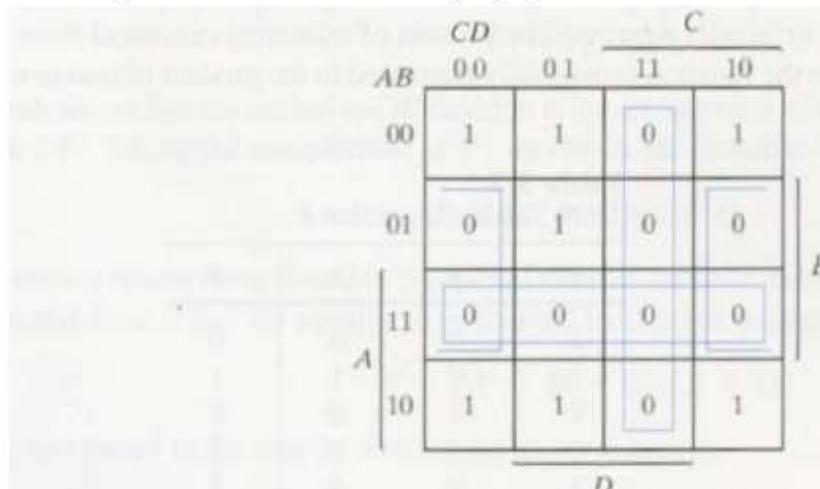


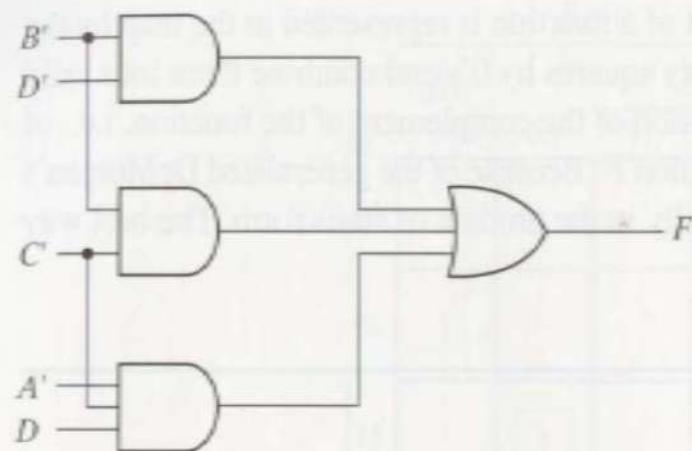
FIGURE 3-14

Map for Example 3-8; $F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10)$
 $= B'D' + B'C + A'C'D = (A' + B)(C + D')(B' + D)$

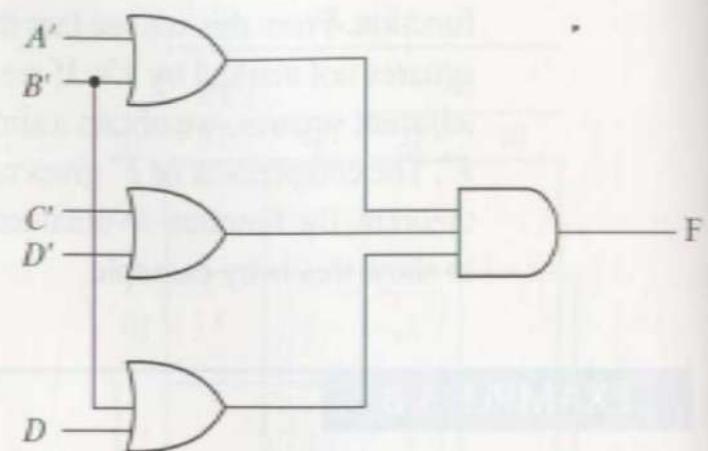
$$\begin{aligned}(a) F &= B'D' \\&\quad + B'C' \\&\quad + A'C'D\end{aligned}$$

- (b) 1. Obtain simplified complemented function:
 $F' = AB + CD + BD'$
2. Applying DeMorgan's theorem to obtain F
 $F = (F')' = (A' + B') (C' + D') (B' + D)$

Gate Implementation for Previous Example



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



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

FIGURE 3-15
Gate Implementation of the Function of Example 3-8

$$F(x, y, z) = \Sigma(1, 3, 4, 6) = \Pi(0, 2, 5, 7)$$

Table 3-2
Truth Table of Function F

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

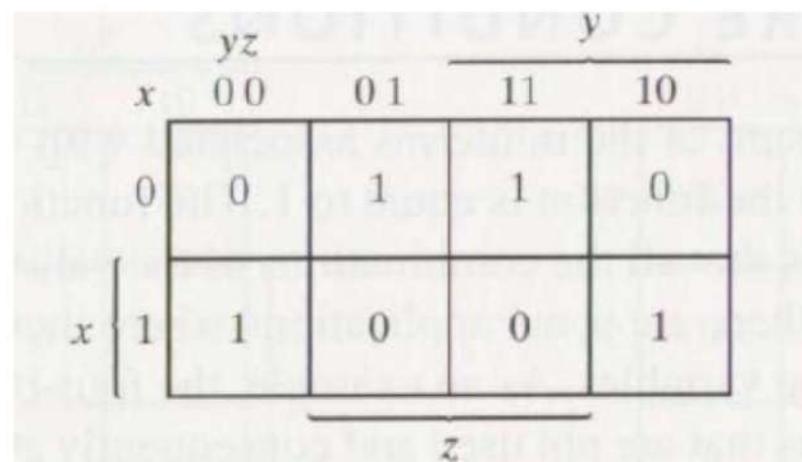


FIGURE 3-16
Map for the Function of Table 3-2

$$F(x, y, z) = \Sigma(1, 3, 4, 6) = \Pi(0, 2, 5, 7)$$

$$F = x'z + xz'$$

$$F' = xz + x'z' \Rightarrow F = (x' + z')(x + z)$$

Don't-Care Condition

- In practice, there are some applications where the function is not specified for certain combinations of the variables
- Functions that have unspecified outputs for some input combinations are called **incompletely specified functions**
- It's customary to call the unspecified minterms of a function **don't-care** conditions
 - marked as X, indicating that we don't care where 0 or 1 is assigned to F for the particular minterm
 - can be used on a map to provide further simplification
 - may be assumed to be either 0 or 1

Example

EXAMPLE 3-9

Simplify the Boolean function

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

which has the don't-care conditions

$$d(w, x, y, z) = \Sigma(0, 2, 5)$$

		yz		y	
		00	01	11	10
wx		X	1	1	X
w	00	0	X	1	0
	01	0	0	1	0
	11	0	0	1	0
	10	0	0	1	0

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

		yz		y	
		00	01	11	10
wx		X	1	1	X
w	00	0	X	1	0
	01	0	X	1	0
	11	0	0	1	0
	10	0	0	1	0

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

FIGURE 3-17

Example with don't-care Conditions

$$F(w, x, y, z) = yz + w'x' = \Sigma(0, 1, 2, 3, 7, 11, 15)$$

$$F(w, x, y, z) = yz + w'z = \Sigma(1, 3, 5, 7, 11, 15)$$

product of sums?

Introduction to Logic Gates

- Logic Gates
 - ❖ The Inverter
 - ❖ The AND Gate
 - ❖ The OR Gate
 - ❖ The NAND Gate
 - ❖ The NOR Gate
 - ❖ The XOR Gate
 - ❖ The XNOR Gate

- Drawing Logic Circuit
- Analysing Logic Circuit

Introduction to Logic Gates

- Universal Gates: NAND and NOR
 - ❖ NAND Gate
 - ❖ NOR Gate
- Implementation using NAND Gates
- Implementation using NOR Gates
- Implementation of SOP Expressions
- Implementation of POS Expressions
- Positive and Negative Logic
- Integrated Circuit Logic Families

Logic Gates

■ Gate Symbols

AND

Symbol set 1



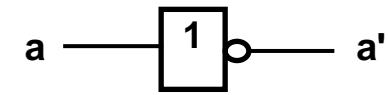
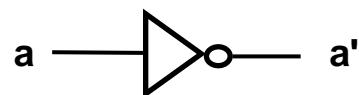
Symbol set 2
(ANSI/IEEE Standard 91-1984)



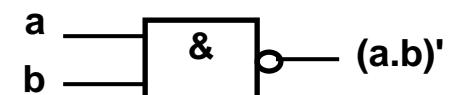
OR



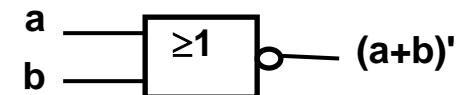
NOT



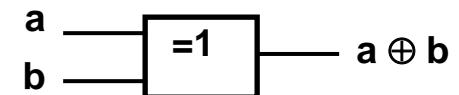
NAND



NOR

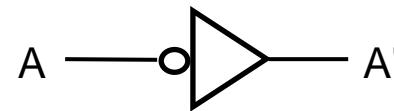
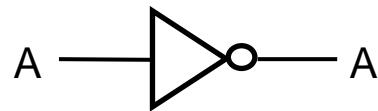


EXCLUSIVE OR



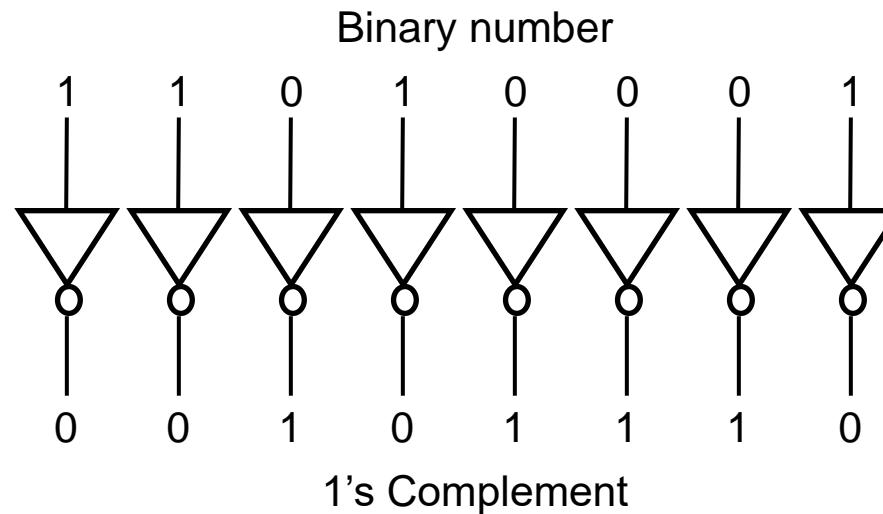
Logic Gates: The Inverter

■ The Inverter



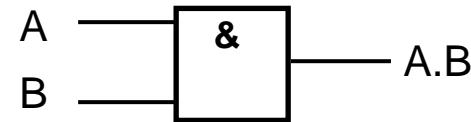
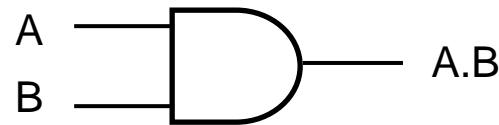
A	A'
0	1
1	0

Application of the inverter: complement.



Logic Gates: The AND Gate

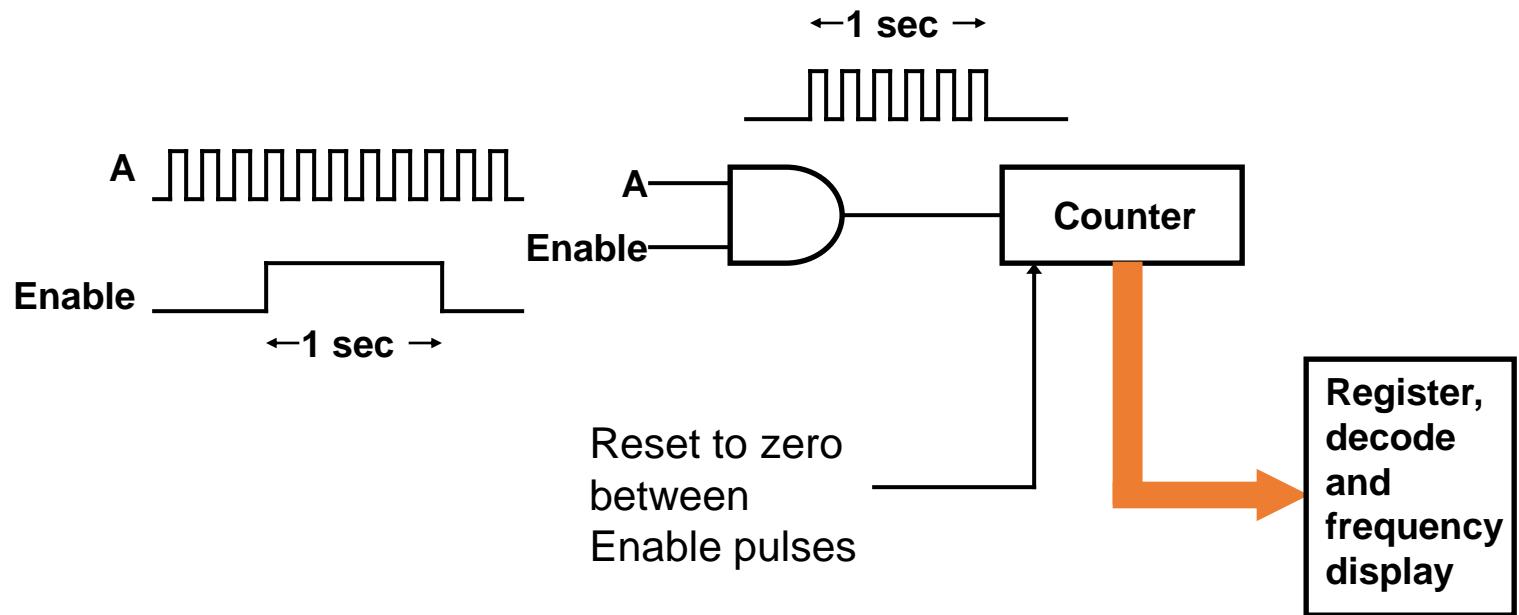
- The **AND** Gate



A	B	A . B
0	0	0
0	1	0
1	0	0
1	1	1

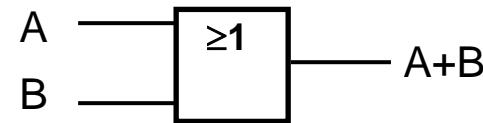
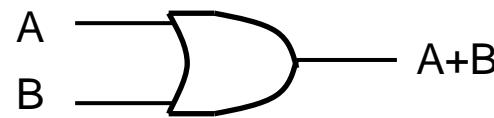
Logic Gates: The AND Gate

■ Application of the AND Gate



Logic Gates: The OR Gate

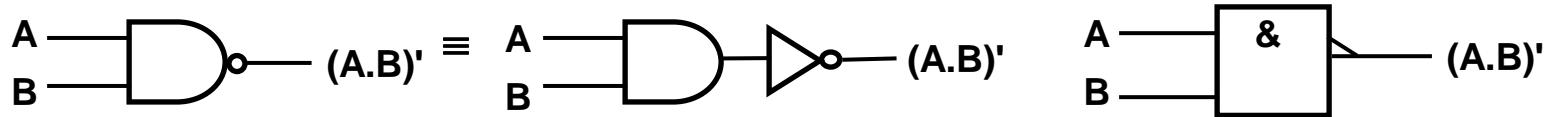
- The **OR** Gate



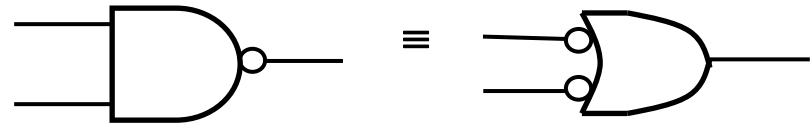
A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

Logic Gates: The NAND Gate

■ The **NAND** Gate



A	B	$(A \cdot B)'$
0	0	1
0	1	1
1	0	1
1	1	0

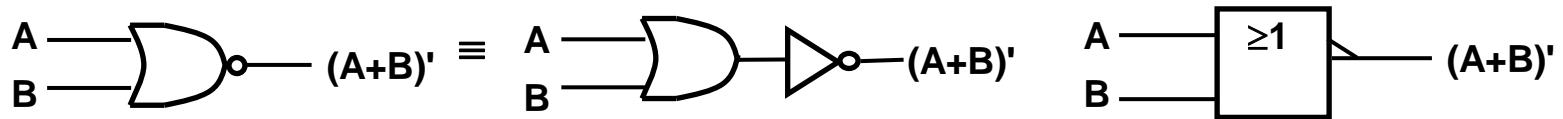


NAND

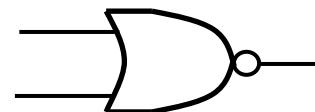
Negative-OR

Logic Gates: The NOR Gate

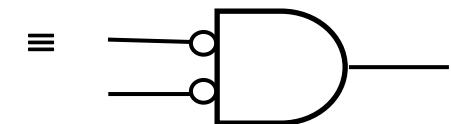
■ The **NOR** Gate



A	B	$(A+B)'$
0	0	1
0	1	0
1	0	0
1	1	0



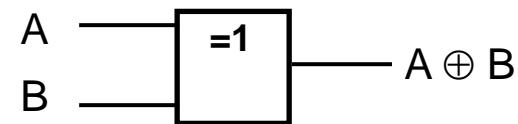
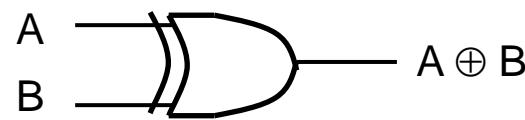
NOR



Negative-AND

Logic Gates: The XOR Gate

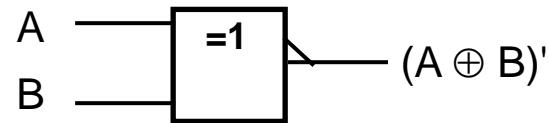
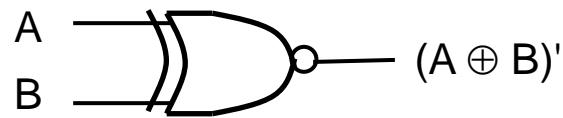
■ The **XOR** Gate



A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Logic Gates: The XNOR Gate

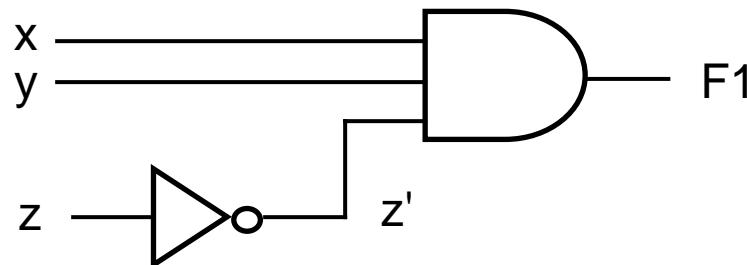
- The **XNOR** Gate



A	B	$(A \oplus B)'$
0	0	1
0	1	0
1	0	0
1	1	1

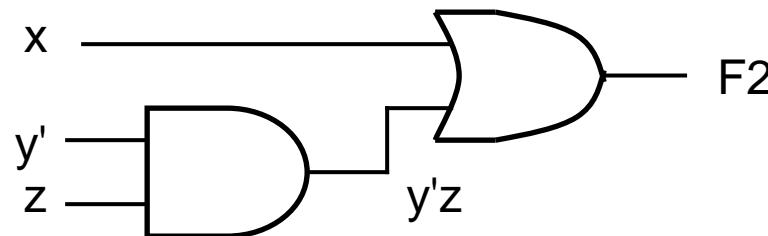
Drawing Logic Circuit

- When a Boolean expression is provided, we can easily draw the logic circuit.
- Examples:
 - (i) $F1 = xyz'$ (note the use of a 3-input AND gate)

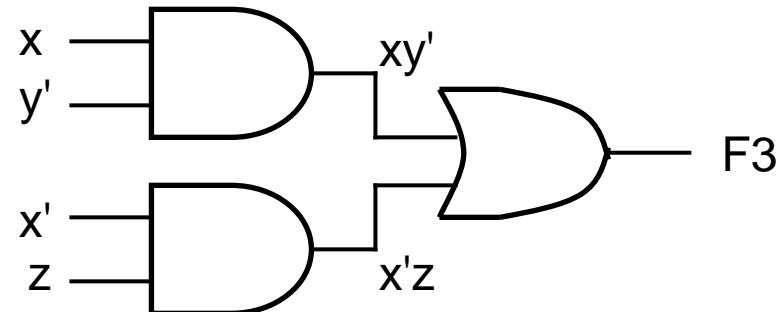


Drawing Logic Circuit

(ii) $F_2 = x + y'z$ (can assume that variables and their complements are available)



(iii) $F_3 = xy' + x'z$



Problem

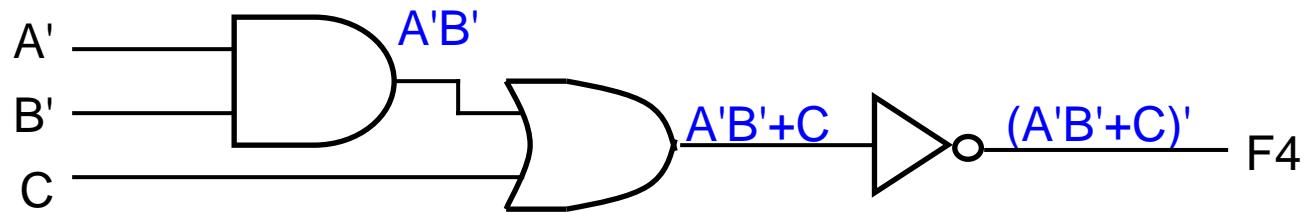
Q1. Draw a logic circuit for $BD + BE + D'F$

Q2. Draw a logic circuit for

$$A'BC + B'CD + BC'D + ABD'$$

Analysing Logic Circuit

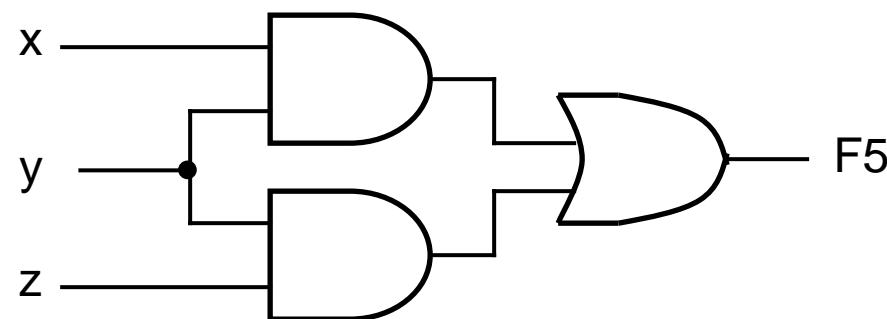
- When a logic circuit is provided, we can analyse the circuit to obtain the logic expression.
- Example: What is the Boolean expression of F4?



$$F4 = (A'B'+C)' = (A+B).C'$$

Problem

- What is Boolean expression of F5?



Universal Gates: NAND and NOR

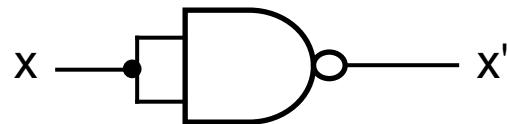
- AND/OR/NOT gates are sufficient for building any Boolean functions.
- However, other gates are also used because:
 - (i) usefulness
 - (ii) economical on transistors
 - (iii) self-sufficient

NAND/NOR: economical, self-sufficient

XOR: useful (e.g. parity bit generation)

NAND Gate

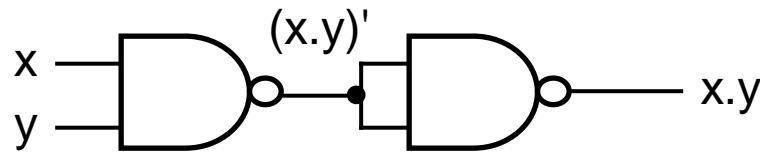
- NAND gate is **self-sufficient** (can build any logic circuit with it).
- Can be used to implement AND/OR/NOT.
- Implementing an inverter using NAND gate:



$$(x \cdot x)' = x' \quad (\text{T1: idempotency})$$

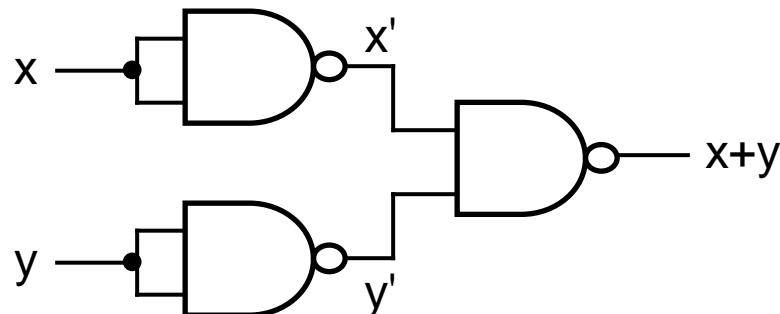
NAND Gate

- Implementing AND using NAND gates:



$$\begin{aligned} ((xy)'(xy))' &= ((xy)')' \quad \text{idempotency} \\ &= (xy) \quad \text{involution} \end{aligned}$$

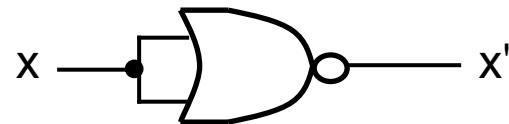
- Implementing OR using NAND gates:



$$\begin{aligned} ((xx)'(yy))' &= (x'y')' \quad \text{idempotency} \\ &= x''+y'' \quad \text{DeMorgan} \\ &= x+y \quad \text{involution} \end{aligned}$$

NOR Gate

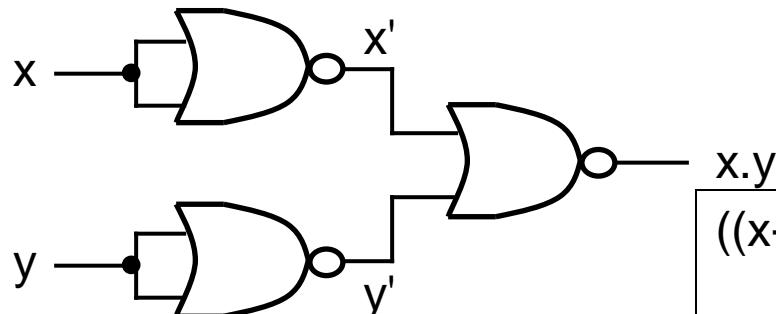
- NOR gate is also self-sufficient.
- Can be used to implement AND/OR/NOT.
- Implementing an inverter using NOR gate:



$$(x+x)' = x' \quad (\text{T1: idempotency})$$

NOR Gate

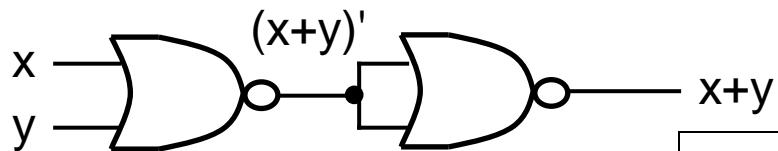
Implementing AND using NOR gates:



$$\begin{aligned} & ((x+x)' + (y+y)')' = (x'+y')' \\ & \quad = x''.y'' \\ & \quad = x.y \end{aligned}$$

idempotency
DeMorgan
involution

Implementing OR using NOR gates:



$$\begin{aligned} & ((x+y)' + (x+y)')' = ((x+y)')' \\ & \quad = (x+y) \end{aligned}$$

idempotency
involution

Implementation using NAND gates

- Possible to implement any Boolean expression using NAND gates.

Procedure:

- (i) Obtain sum-of-products Boolean expression:

$$\text{e.g. } F_3 = xy' + x'z$$

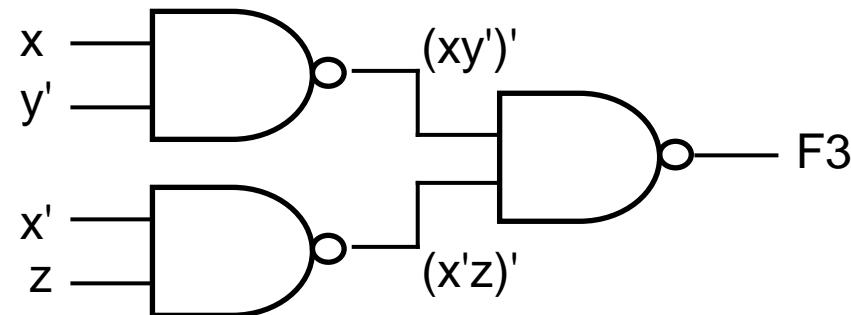
- (ii) Use DeMorgan theorem to obtain expression using 2-level NAND gates

$$\text{e.g. } F_3 = xy' + x'z$$

$$= (xy' + x'z)'' \quad \text{involution}$$

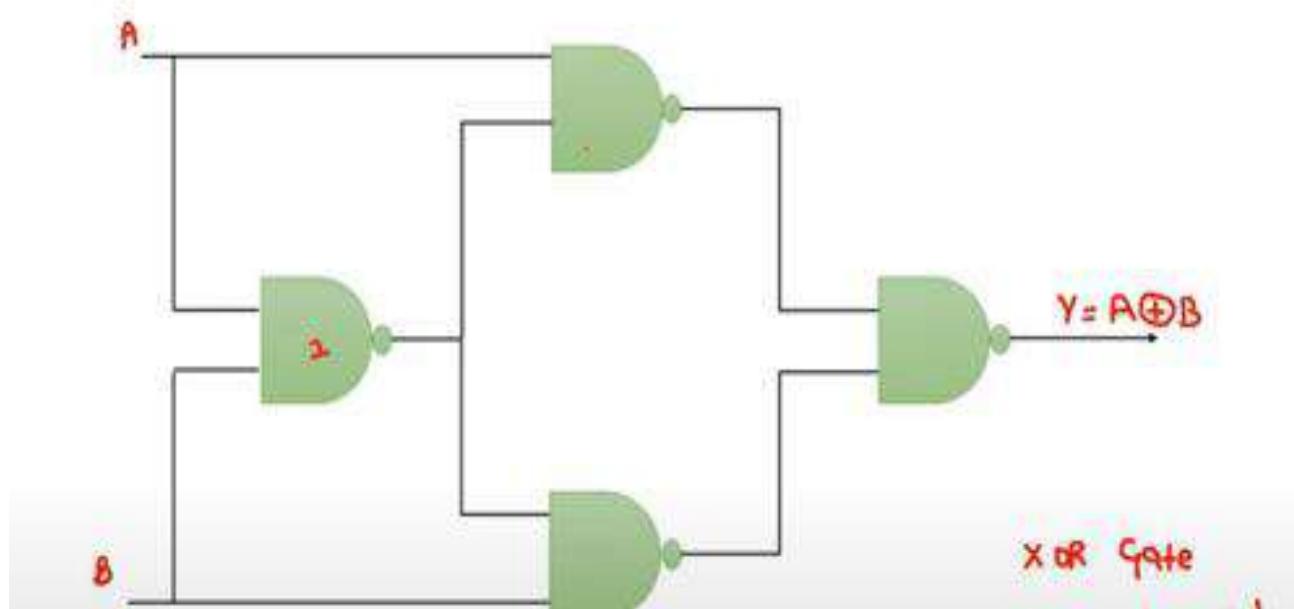
$$= ((xy')' \cdot (x'z)')' \quad \text{DeMorgan}$$

Implementation using NAND gates



$$F3 = ((xy')' \cdot (x'z)')' = xy' + x'z$$

Implement XOR gate with NAND gate



Implementation using NOR gates

- Possible to implement boolean expression using NOR gates.

Procedure:

- Obtain product-of-sums Boolean expression:

$$\text{e.g. } F_6 = (x+y').(x'+z)$$

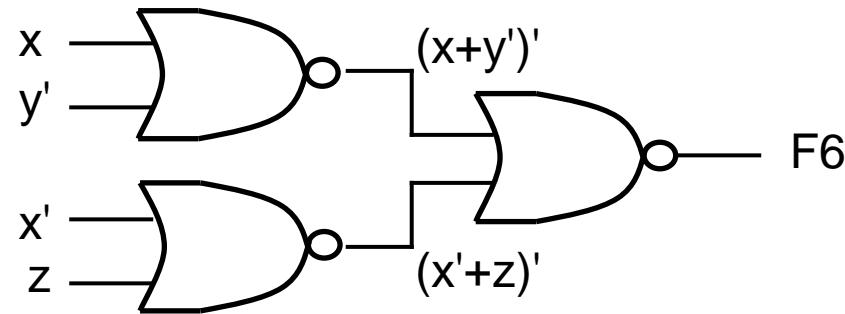
- Use DeMorgan theorem to obtain expression using 2-level NOR gates.

$$\text{e.g. } F_6 = (x+y').(x'+z)$$

$$= ((x+y').(x'+z))'' \quad \text{involution}$$

$$= ((x+y')' + (x'+z)')' \quad \text{DeMorgan}$$

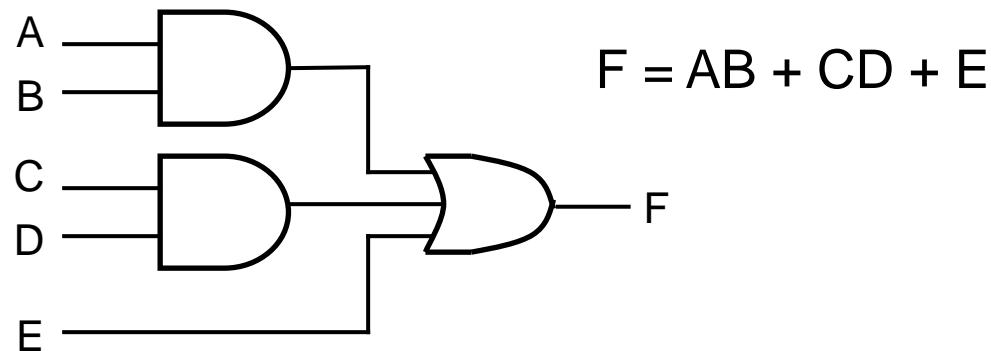
Implementation using NOR gates



$$F6 = ((x+y')' + (x'+z)')' = (x+y').(x'+z)$$

Implementation of SOP Expressions

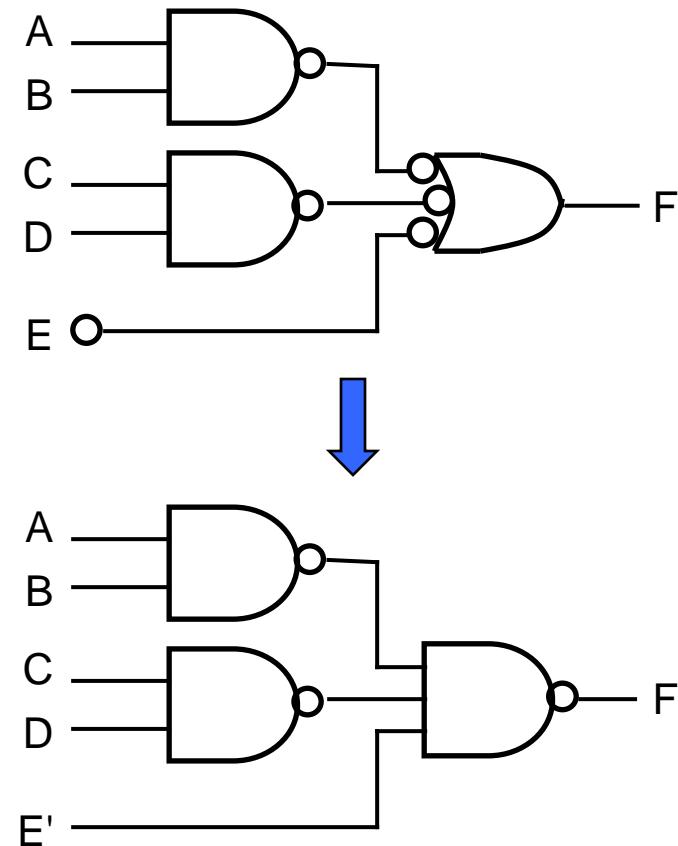
- Sum-of-Products expressions can be implemented using:
 - ❖ 2-level AND-OR logic circuits
 - ❖ 2-level NAND logic circuits
- AND-OR logic circuit



Implementation of SOP Expressions

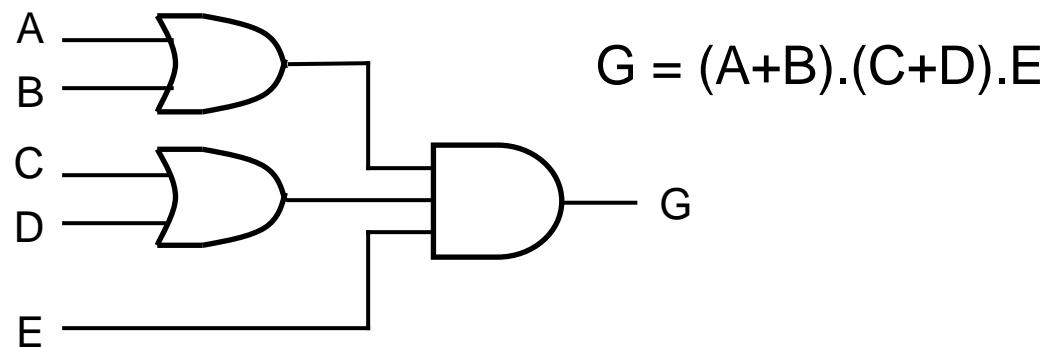
- NAND-NAND circuit (by circuit transformation)

- add double bubbles
- change OR-with-inverted-inputs to NAND & bubbles at inputs to their complements



Implementation of POS Expressions

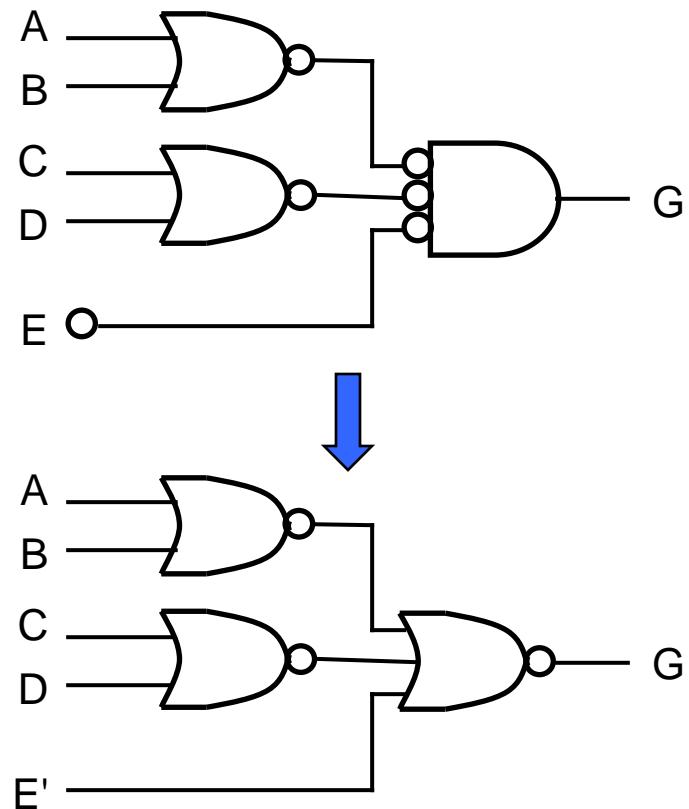
- Product-of-Sums expressions can be implemented using:
 - ❖ 2-level OR-AND logic circuits
 - ❖ 2-level NOR logic circuits
- OR-AND logic circuit



Implementation of POS Expressions

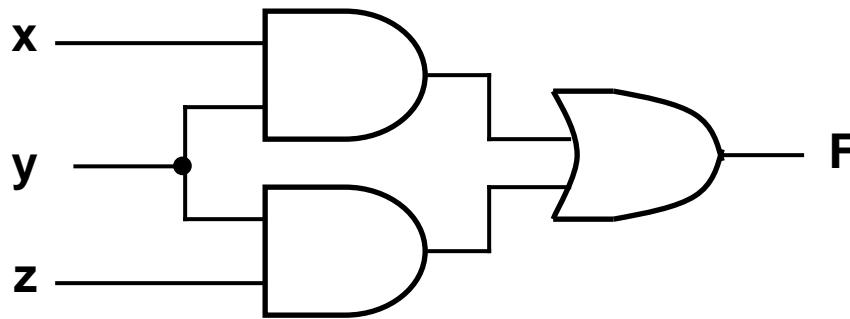
- NOR-NOR circuit (by circuit transformation):

- add double bubbles
- changed AND-with-inverted-inputs to NOR & bubbles at inputs to their complements



Solve it yourself (Exercise 4.3)

- Q1. Draw a logic circuit for $BD + BE + D'F$ using only NAND gates. Use both DeMorgan method and SOP method.
- Q2. Transform the following AND-OR Circuit to NAND circuit.



- Q3. Using only NOR gates, draw a logic circuit using POS method for $(A+B+C')(B'+C'+D)$

Positive & Negative Logic

- In logic gates, usually:
 - ❖ H (high voltage, 5V) = 1
 - ❖ L (low voltage, 0V) = 0
- This convention – **positive logic**.
- However, the reverse convention, **negative logic** possible:
 - ❖ H (high voltage) = 0
 - ❖ L (low voltage) = 1
- Depending on convention, same gate may denote different Boolean function.

Positive & Negative Logic

A signal that is set to logic 1 is said to be *asserted*, or *active*, or *true*.

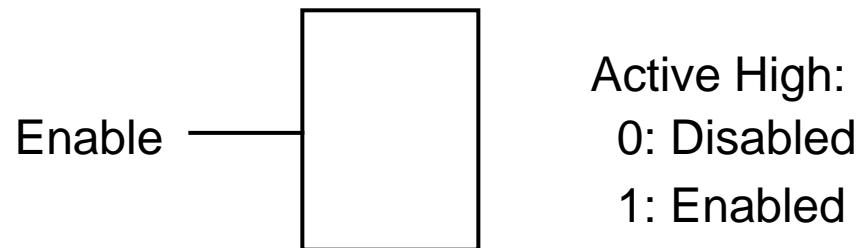
A signal that is set to logic 0 is said to be *deasserted*, or *negated*, or *false*.

Active-high signal names are usually written in uncomplemented form.

Active-low signal names are usually written in complemented form.

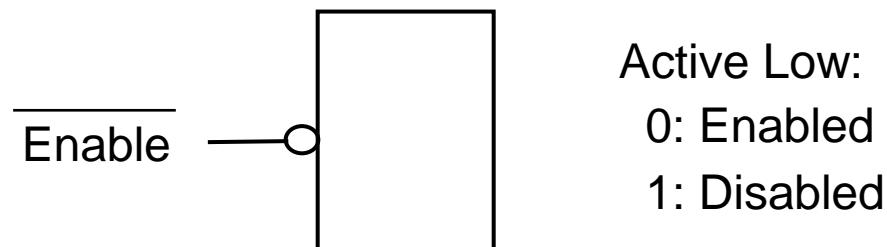
Positive & Negative Logic

Positive logic:



Active High:
0: Disabled
1: Enabled

Negative logic:



Active Low:
0: Enabled
1: Disabled

Integrated Circuit Logic Families

Some digital integrated circuit families: TTL, CMOS, ECL.

TTL: Transistor-Transistor Logic.

Uses bipolar junction transistors

Consists of a series of logic circuits: standard TTL, low-power TTL, Schottky TTL, low-power Schottky TTL, advanced Schottky TTL, etc.

Integrated Circuit Logic Families

TTL Series	Prefix Designation	Example of Device
Standard TTL	54 or 74	7400 (quad NAND gates)
Low-power TTL	54L or 74L	74L00 (quad NAND gates)
Schottky TTL	54S or 74S	74S00 (quad NAND gates)
Low-power Schottky TTL	54LS or 74LS	74LS00 (quad NAND gates)

Integrated Circuit Logic Families

CMOS: Complementary Metal-Oxide Semiconductor.

Uses field-effect transistors

ECL: Emitter Coupled Logic.

Uses bipolar circuit technology.

Has fastest switching speed but high power consumption.

Integrated Circuit Logic Families

Performance characteristics

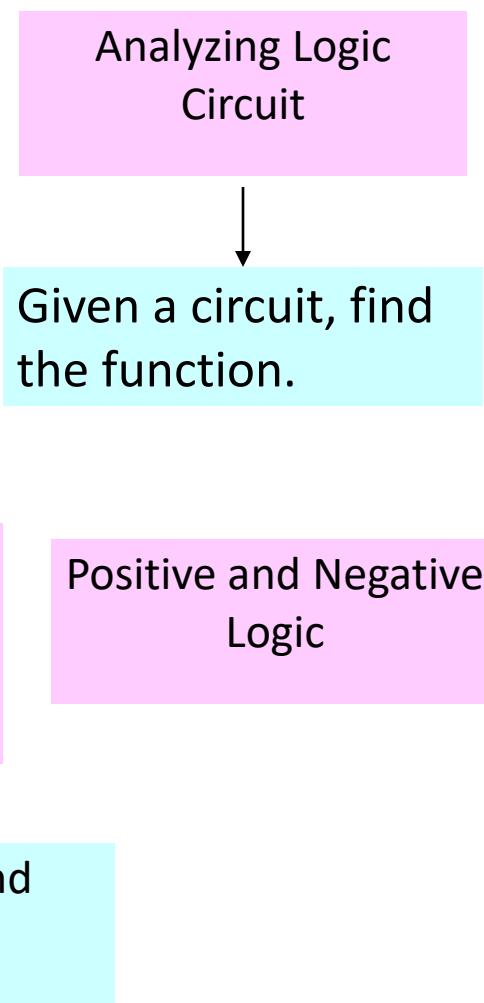
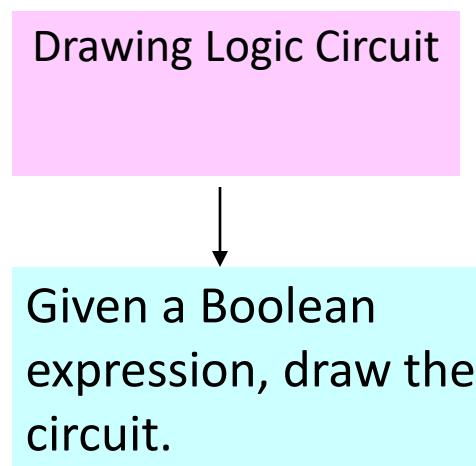
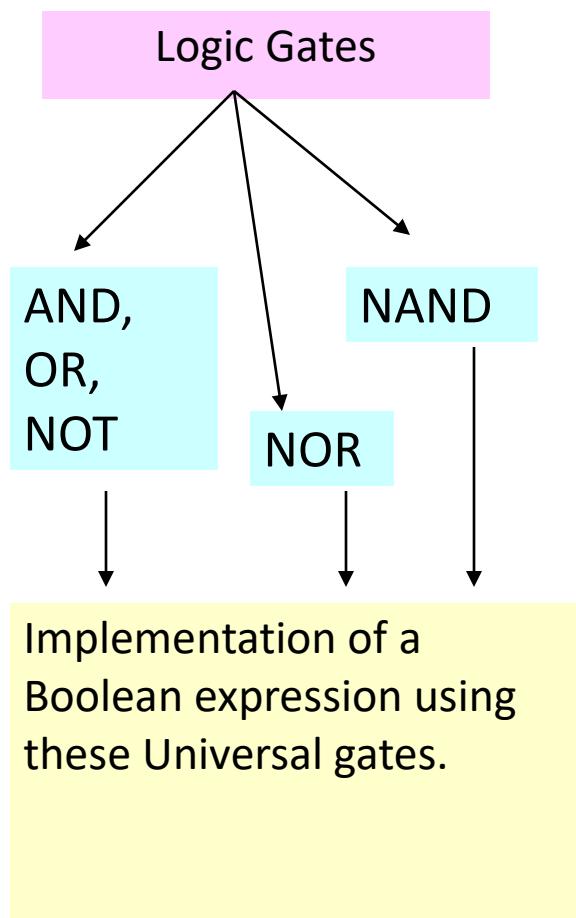
Propagation delay time.

Power dissipation.

Fan-out: Fan-out of a gate is the maximum number of inputs that the gate can drive.

Speed-power product (SPP): product of the propagation delay time and the power dissipation.

Summary





VIT
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

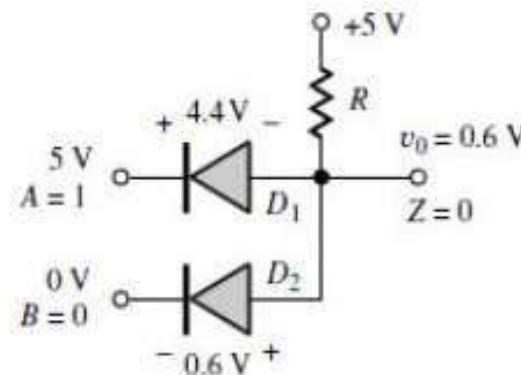
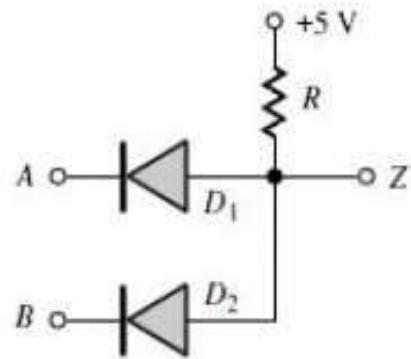
Logic Families for Implementation of Logic Gates



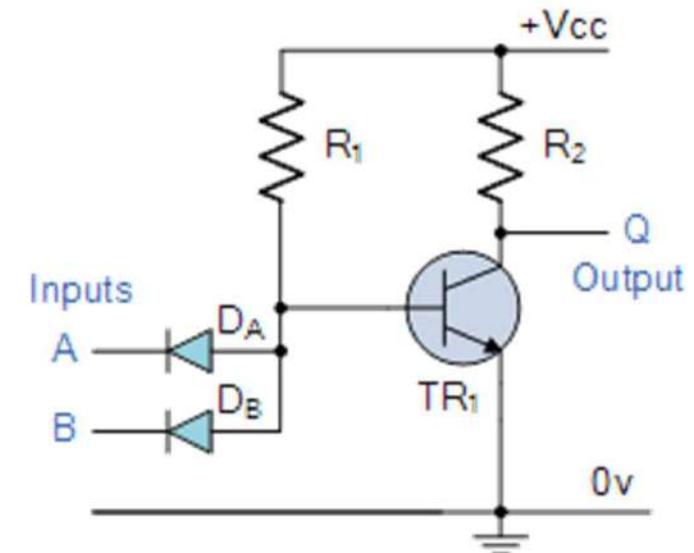
Logic Families

- Ways to design logic gates
 - Diode Transistor Logic (DTL)
 - Transistor – Transistor Logic (TTL)
 - Emitter Coupled Logic (ECL)
 - Complementary Metal Oxide Semiconductor (CMOS) Logic

Logic Families – Diode Transistor Logic

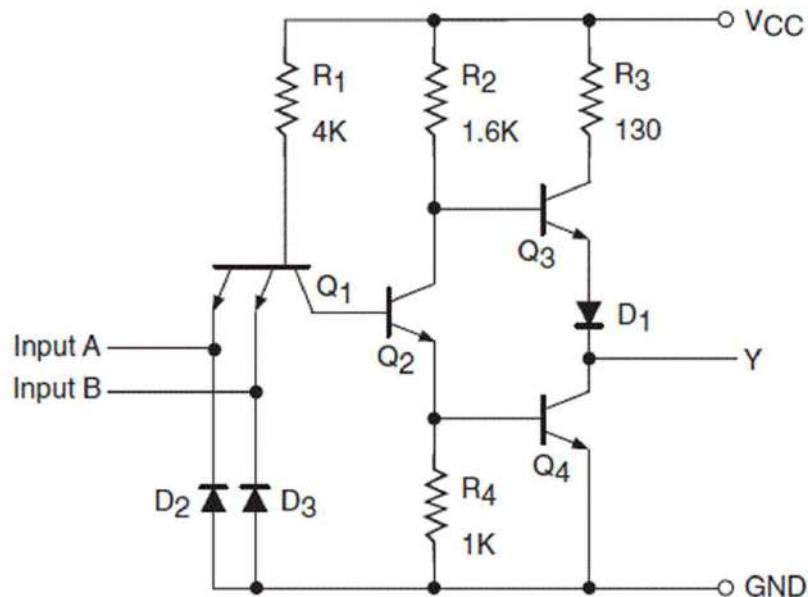


2-input AND gate

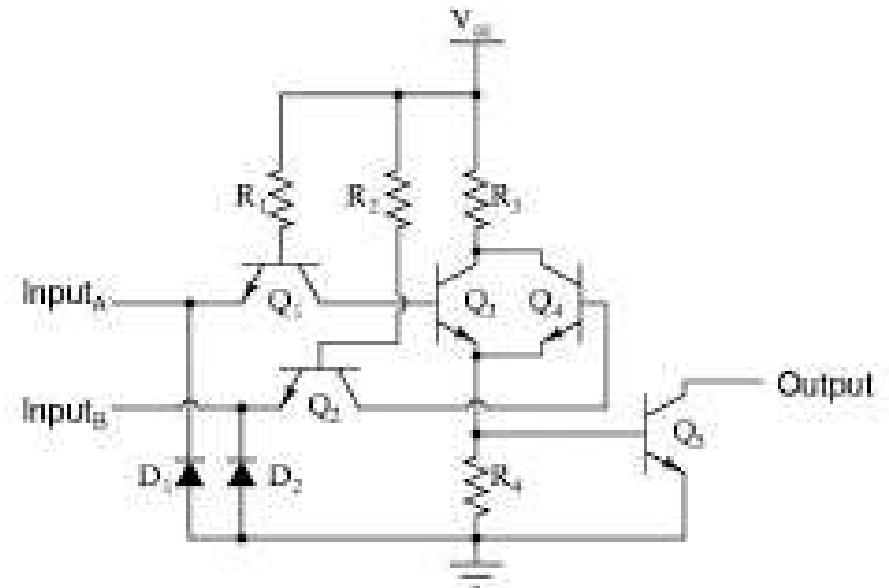


2-input NAND gate

Logic Families – Transistor Transistor Logic

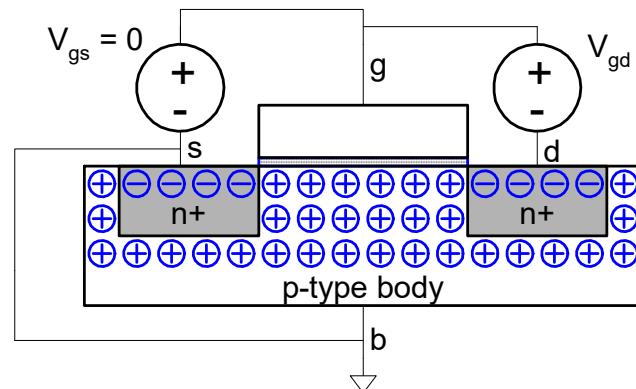


2-input NAND gate

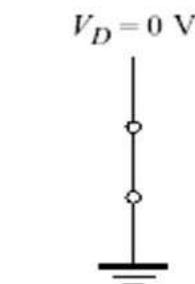
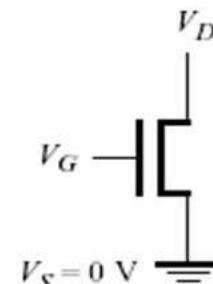
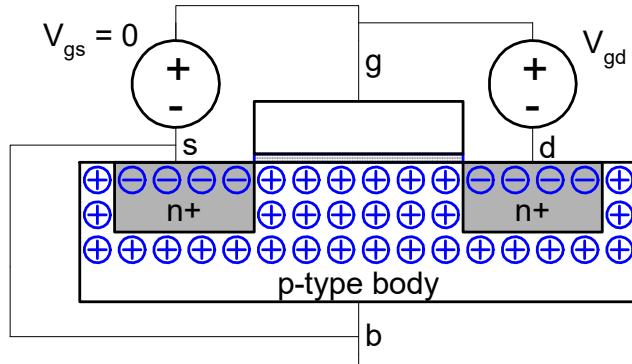


2-input NOR gate

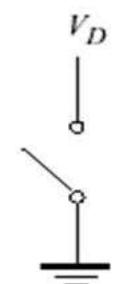
- An MOS transistor is a majority-carrier device, in which the current in a conducting channel between the source and the drain is modulated by a voltage applied to the gate.



- Majority carrier = electrons
- A positive voltage applied on the gate with respect to the substrate enhances the number of electrons in the channel and hence increases the conductivity of the channel.
- If gate voltage is less than a threshold voltage V_t , the channel is cut-off (very low current between source & drain).



Closed switch
when $V_G = V_{DD}$



Open switch
when $V_G = 0 \text{ V}$

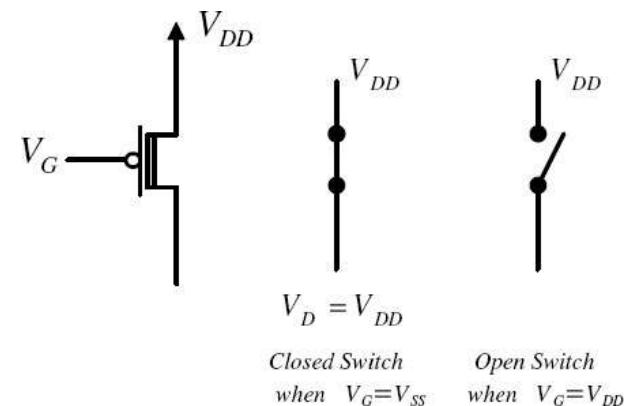


VIT[®]

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

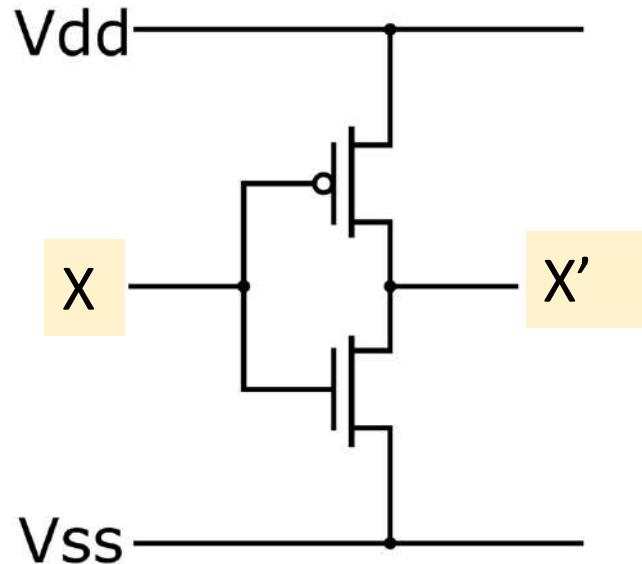
P-MOS Transistor

- Majority carrier = holes
- A negative voltage applied on the gate with respect to the substrate enhances the number of holes in the channel and hence increases the conductivity of the channel.
- If gate voltage is less than a threshold voltage V_t , the channel is cut-off (very low current between source & drain).

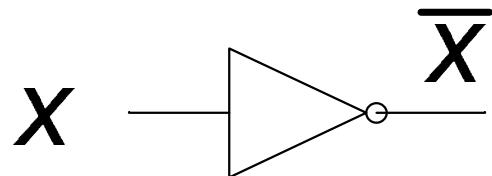




Complementary MOS (CMOS) NOT gate



X	X'
0	1
1	0



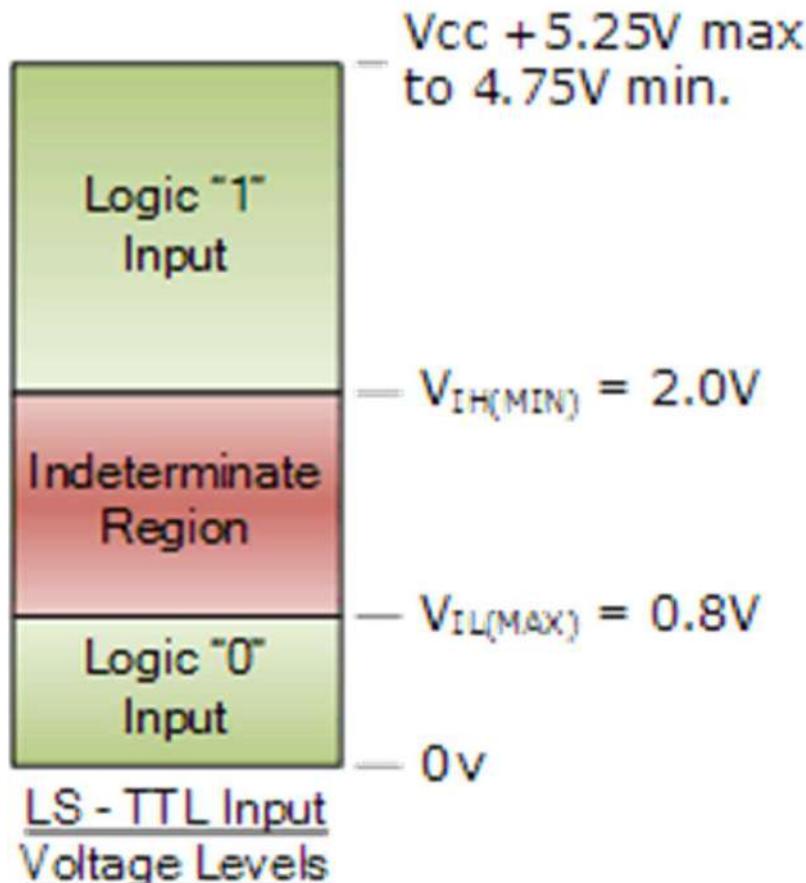
Voltage Levels



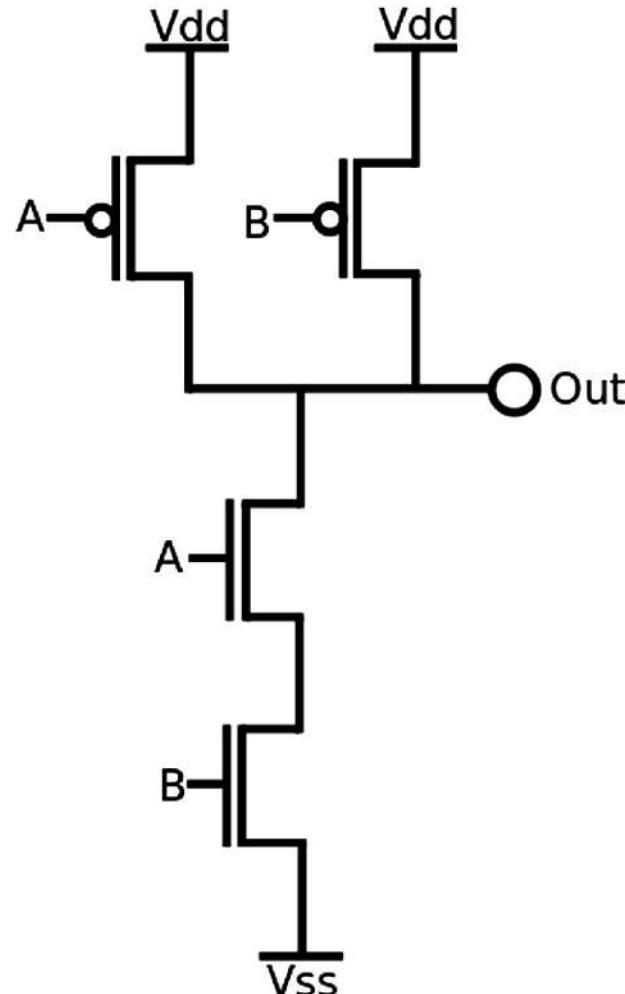
VIT[®]

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

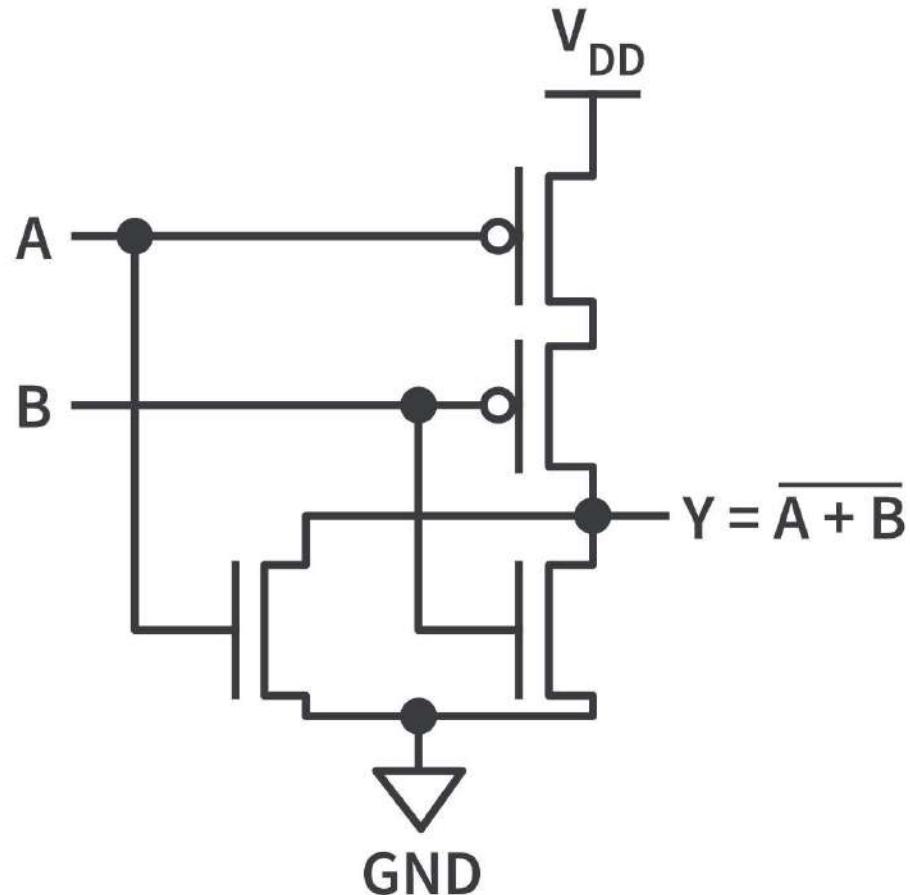


Complementary MOS (CMOS) NAND gate



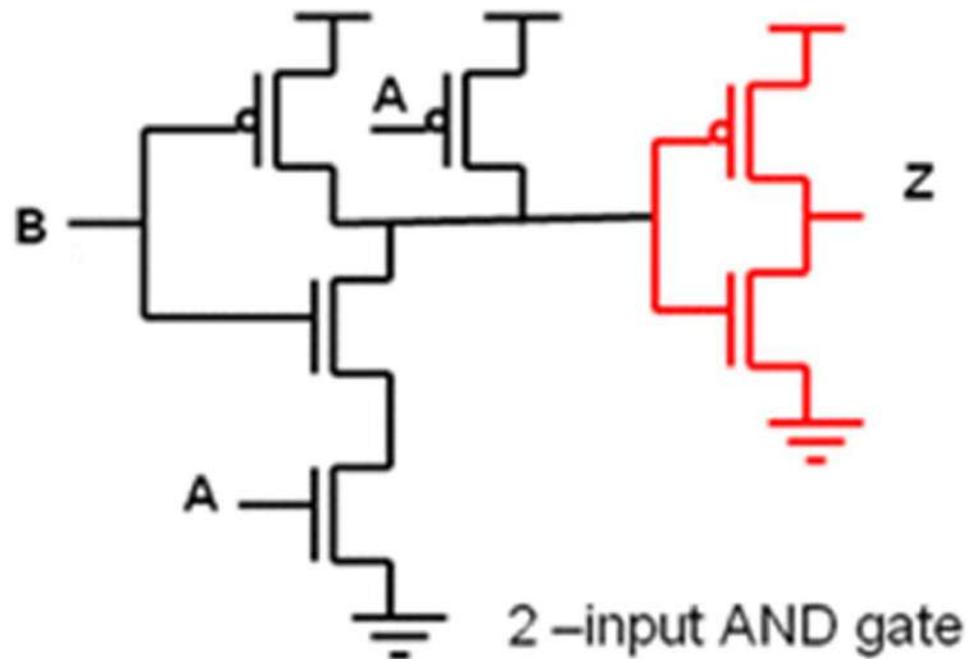
A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

Complementary MOS (CMOS) NOR gate



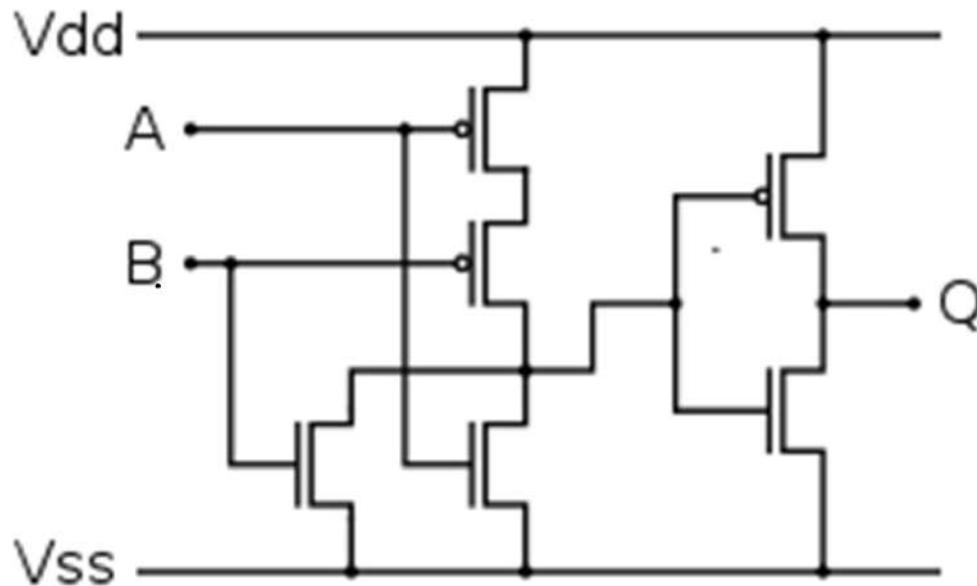
A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

Complementary MOS (CMOS) AND gate



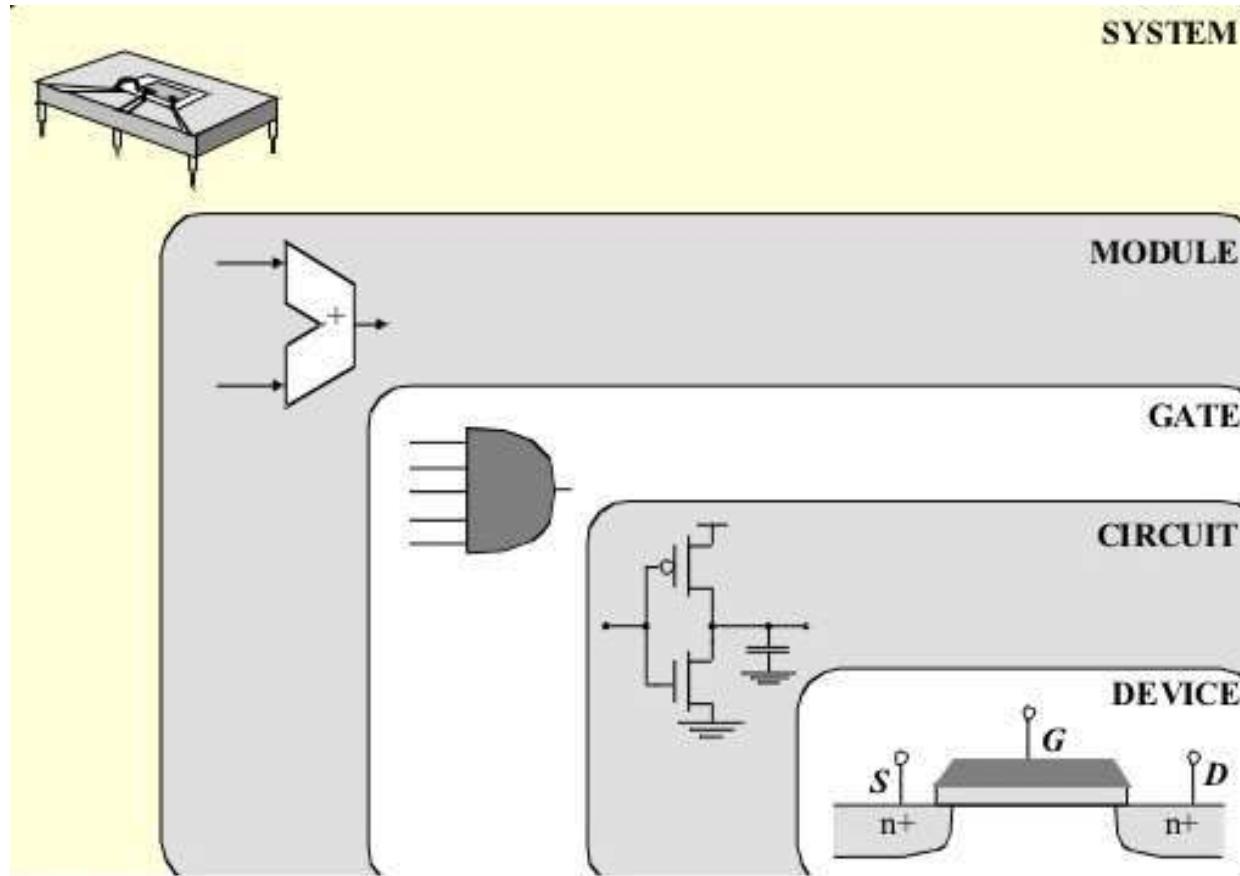
A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

Complementary MOS (CMOS) OR gate



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

Design Abstraction and CAD Tools



<https://www.youtube.com/watch?v=jVHq7V2-gOs>

Verilog HDL

Session -I

VERILOG HDL

A Guide to Digital Design and Synthesis

by

Samir Palnitkar



Outline

- ❖ VHDL Vs Verilog HDL

EVOLUTION OF COMPUTER AIDED DIGITAL DESIGN

- The first Integrated Circuit (IC) or silicon chip was fabricated in 1960s.
- IC chip evolution -> SSI, MSI, LSI, VLSI...
- Designing single chip with more than 100,000 transistors - VLSI.
- Complicated design processes.
- Traditional / conventional design method includes manual translation of design description into logical equations and then to schematic.

EVOLUTION OF COMPUTER AIDED DIGITAL DESIGN

- Verification through bread-boarding?
- CAD (back-end) tools became critical.
- Graphic packages (PSpice, Workbench, OrCAD) for gate level / schematic representation.
 - Cannot handle higher complexities.
 - Poor portability.
 - Poor readability for high complex designs.
- In all the above design methods the functional bugs cannot be identified till the design is implemented in hardware, and hence the design time is very long.

HDL?

- In electronics, a hardware description language or HDL is any language from a class of computer languages for formal description of electronic circuits. It can describe the circuit's operation, its design, and tests to verify its operation by means of simulation
- Popular HDLs are Verilog HDL & VHDL (for any complexity).

Emergence of HDLs

- Logic Simulators for verifying functionality thus removing functional bugs at an early stage in the design.
- High level languages such as FORTRAN, PASCAL, C, C++, etc., are sequential in nature.
- Digital designers felt the need for a standard language to describe digital systems / hardware.
- Hardware Description Languages (HDLs) comes into existence and these have special constructs to model the concurrency of processes found in digital systems.

Features of HDLs

- Easy development, verification and debugging through HDLs.
- HDL descriptions are easily portable, and is also compatible to all design tools.
- HDLs can describe the digital systems at various abstraction levels & also supports hierarchical modeling.
- HDL descriptions can be functionally simulated with Logic Simulators .

Features of HDLs

- Advent of Logic synthesis tools in late 1980's pushes HDLs to the forefront of digital design.
- Digital circuits described at Register Transfer Level (RTL) using HDLs, can also be synthesized through automated logic synthesis tools.
- Logic Synthesis tools can extract gate level details automatically from HDL (RTL) description.

VHDL Vs Verilog HDL

- ❖ VHDL : Very High Speed Integrated Circuit Hardware Description Language
- ❖ Verilog HDL: Verification Logic Hardware Description Language

Differences:

- VHDL was designed to support system level design and specification.
- Verilog HDL was designed primarily for digital hardware designers developing FPGAs and ASICs.

The differences becomes clear when one analyze the language features.

VHDL Vs Verilog HDL

❖ VHDL

- Provides some high level constructs not available in Verilog (User defined types, Configurations etc.,)

❖ Verilog

- Provides comprehensive support for low-level digital design
- Not available in native VHDL
 - Range of type definitions and supporting functions(called packages need to be included)

History of Verilog

- Developed by Philip Moorby in 1984-1985.
- Gateway Design Automation introduced Verilog in 1984 as their proprietary HDL.
- Cadence took over in 1989.
- Cadence made Verilog HDL public in 1990.
- Verilog HDL becomes IEEE 1364-1995.

Popularity of Verilog HDL

- Verilog HDL is modeled after ‘C’ language.
- Allows different levels of abstraction to be mixed in the same model.
- Easy to learn and easy to use.
- Almost all logic simulation & synthesis tools support Verilog.
- Verilog HDL is an IEEE 1364 standard.

Verilog basics

Lexical conventions

- White space
- Operators
- Comments
- Number specification
- Strings
- Identifiers and Keywords
- Escaped Identifiers

White space

- Comprise of the white space.
- Blank spaces ---> \b
- Tabs ---> \t
- New lines ---> \n
- White space is not ignored in strings.

Types of Operators

- Unary operators.
 - operates on a single operand.

```
assign out = ~ a; //In dataflow abstraction level
```

- Binary operators.
 - operates on two operands.

```
assign out = a & b; //In dataflow abstraction level
```

- Ternary operators.
 - operates on three operands.

```
assign out = s ? a : b; //In dataflow abstraction level
```

Comments

- Improve readability and helps good documentation.
- Two comment structures are available in verilog:
 - single / one line comment
 - `z = x + y; // arithmetic operation`
 - multiple line / block comment
 - `/* this logic performs
the reversal
of bits */`
- Multiple line comments cannot be nested.
 - `/* This is /* a wrong
comment */ structure */`

Format

- Verilog HDL is case-sensitive.
- All the keywords in Verilog must be in lower case.
- Verilog constructs may be written across multiple lines, or on one line.

```
module and_gate(a,b,y);input a,b; output y; assign y= a & b; endmodule

module and_gate(a,b,y);
input a,b;
output y;
assign y= a & b;
endmodule
```

1/3 Identifiers and key words

- Identifiers and keywords are used to define language constructs.
- Identifiers refer objects to be referenced in the design.
- Identifiers are made of alphabets (both cases), numbers, the underscore '_' and the dollar sign '\$'.
- They start with an alphabetic character or underscore.

Identifiers and key words

- They cannot start with a number or with ‘\$’ which is reserved for system tasks.
- Identifiers are case sensitive i.e., identifiers differing in their case are distinct.
- An identifier say count is different from COUNT, count and cOuNT.

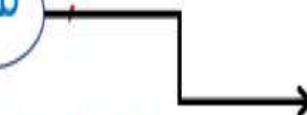
Examples of Identifiers

Identifier used
during simulation



```
module and_gate(a,b,y);
  input a,b;
  output y;
  assign y= a & b;
endmodule
```

Identifier used
in ports



```
module and_gate(a,b,y);
  input a,b;
  output y;
  assign y= a & b;
endmodule
```

Examples of Identifiers

Count

COUNT

_R2_D2

R56_68

FIVE\$

\$count

Illegal

12six_b

Illegal

- Example:

```
reg sum;
```

```
input data;
```

Identifiers(3/3)

- ⇒ **name, _name. Name, name1, name_\$, ...** all these are allowed as identifiers.
- ⇒ **name aa** not allowed as an identifier because of the blank (“name” and “aa” are interpreted as two different identifiers)
- ⇒ **\$name** not allowed as an identifier because of the presence of “\$” as the first character.
- ⇒ **1_name** not allowed as an identifier, since the numeral “1” is the first character.
- ⇒ **@name** not allowed as an identifier because of the presence of the character “@”.
- ⇒ **A+b** not allowed as an identifier because of the presence of the character “+”.
- ⇒ Keywords are not allowed

Keywords

- The keywords define the language constructs.
- A keyword signifies an activity to be carried out, initiated, or terminated.
- As such, a programmer cannot use a keyword for any purpose other than that it is intended for.
- All keywords in Verilog are in small letters and require to be used as such.

Examples

module → signifies the beginning of a module definition.

endmodule → signifies the end of a module definition.

begin → signifies the beginning of a block of statements.

end → signifies the end of a block of statements.

if → signifies a conditional activity to be checked

while → signifies a conditional activity to be carried out.

Number specification

- Sized numbers.
 - <size> '<base format> <number>
- Unsized numbers.
 - '<base format> <number>
- <size> in decimal
- <base format> can be b or B, d or D, o or O and h or H.
- Numbers without <base format> are decimal by default.

Number specification

- Examples:

Sized numbers :

4'b1111 // This is a 4-bit binary number

12'habc // This is a 12-bit hexadecimal number

16'd255 // This is a 16-bit decimal number.

Unsized numbers :

23456 // This is a 32-bit decimal number by default

'hc3 // This is a 32-bit hexadecimal number

'o21 // This is a 32-bit octal number

Number Specification-Example

<i>Format</i>	<i>Explanation</i>
9'o123	The binary equivalents are 001 010 011
35 'd35	Both of these represent decimal numbers of unspecified size – normally interpreted by Verilog as 32 bitwide, i.e., 0000 0000 0000 0000 0000 0000 0000 0010 0011
7'Hx	7-bit x (x extended), i.e.... xxxxxxx
4'hz	4-bit z (z extended), i.e... zzzz
4'd-4	Not legal
8 'h 2A	Spaces allowed between size & ' character & between <i>base</i> and <i>value</i>
3' b001	Not legal: no space allowed between ' and base b
10'b10	Padded with 0 to the left, 0000000010
3'b1001_0011	Represented as 3'b011

Number Specification-Example

<i>Format</i>	<i>Explanation</i>
9'b1_1011_1x01	All these represent binary numbers of value 11011x01. B & b specify binary numbers. “_” is ignored. x signifies the concerned bit to be of unknown value.
9'b11011x01	
9'B11011x01	
11'hb0	A 11 bit number with a hex assignment. Its value is 000 1011 0000. The number of bits specified is more than that indicated in the value field. Enough zeros are padded to the left as shown.
9'hza	A hex number of 9 bits. Its value is taken as zzzzz 1010.
-4'd7	A 4 bit negative number. Its value in 2's complement form is 7. Thus the number is actually – (16 – 7) = –9.

Number Representation

Examples:

- **6'b010_111** gives **010111**
- **8'b0110** gives **00000110**
- **4'bx01** gives **xx01**
- **16'H3AB** gives **0000001110101011**
- **24** gives **0...0011000**
- **5'036** gives **11110**
- **16'Hx** gives **xxxxxxxxxxxxxxxx**
- **8'hz** gives **zzzzzzzz**

Unknown & high impedance values

- X or x for unknown values.
- Z or z for high impedance values.
- X or Z at the MSB has the self padding property.

Examples:

32'B z // this is a 32-bit high impedance number

6'h X // this is a 6-bit hex number

12'H 13x // this is a 12-bit hex number

Numbers - more examples

- 5'O37 5-bit octal which gives 11111
- 4'D2 4-bit decimal
- 7'Hx 7-bit x (x extended), i.e.... xxxxxxx
- 4'hz 4-bit z (z extended), i.e... zzzz
- 4'd-4 Not legal
- 8 'h 2A Spaces allowed between size & ' character
& between *base* and *value*
- 3' b001 Not legal: no space allowed between ' and
base b
- 10'b10 Padded with 0 to the left, 0000000010
- 3'b1001_0011 is same as 3'b011

Integer Number

- ☞ Integers can be represented as a decimal number –signed or unsigned; an unsigned number is automatically taken as a positive number.
- ☞ By default circuit synthesizer will assign 32 bits of width.

Valid Number Specification:

2

25

253

-253

The following are invalid since non-decimal representations are not permissible.

2a

B8

-2a

-B8

Real Number

- ☞ Real numbers can be specified in decimal or scientific notation.
- ☞ The decimal notation has the form 3.2
- ☞ A number can be specified in scientific notation as $4.3\text{e}2$ where 4.3 is the mantissa and 2 the exponent.

The decimal equivalent of this number is 430.

Examples:

$-4.3\text{e}2$, $-4.3\text{e}-2$, and $4.3\text{e}-2$.

Strings

- A string is a sequence of characters that are enclosed by double quotes.
- “Verilog classes are very interesting???”
- Spaces are not ignored in strings.
- Strings cannot be on multiple lines.

Strings in expressions

Example:

P = “numb”

assigns the binary value 0110 1110 0111 0101 0110 1101 0110 0010 to P

(0110 1110, 0111 0101, 0110 1101 and 0110 0010 are the 8-bit equivalents of the letters n, u, m, and b, respectively).

Test for Understanding

- Multiple line comments can be nested – True/False
- List out rules for choosing an identifier name
- Verilog code is case insensitive – True/False
- Identify the valid number specification
 - a. 1'b0 b. -4'b10 c. 3' b101 d. 3'bx e. 4'b10
- Strings can be given in multiple lines – True / False

Concept of a 'module'

- A module is a basic building block in Verilog and can specify a system of any complexity.
- The module definition is the same for a system of any complexity.
- Provides functionality through its port interface.
- It can be an element or a collection of lower-level design (macro or leaf cells or primitive cells) blocks.

Components of a Verilog Module

Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),

Declarations of **wires**,
regs and other variables

Data flow statements
(**assign**)

Instantiation of lower
level modules

always and **initial** blocks.
All behavioral statements
go in these blocks.

Tasks and functions

endmodule statement

How to declare a module ?

- A module in Verilog is declared using the keyword **module** and a corresponding keyword **endmodule** must appear at the end of the module.
- Each module must have a module name, which acts as an identifier.
- A module can have an optional port list which describes the input, output & inout terminals of the module.

Module declaration - Examples

```
module ex1( );
```



Module name: ex1

```
endmodule
```



No. of ports: 0

```
module ex2(y,a,b,c,d);
```



Module name: ex2

```
output y;
```



No. of ports: 5

```
input a,b,c,d;
```

```
wire f1,f2;
```

```
or o1(f1,a,b)
```

```
and a1(f2,c,d);
```

```
xor x1(y,f1,f2);
```

```
endmodule
```

Nesting of modules

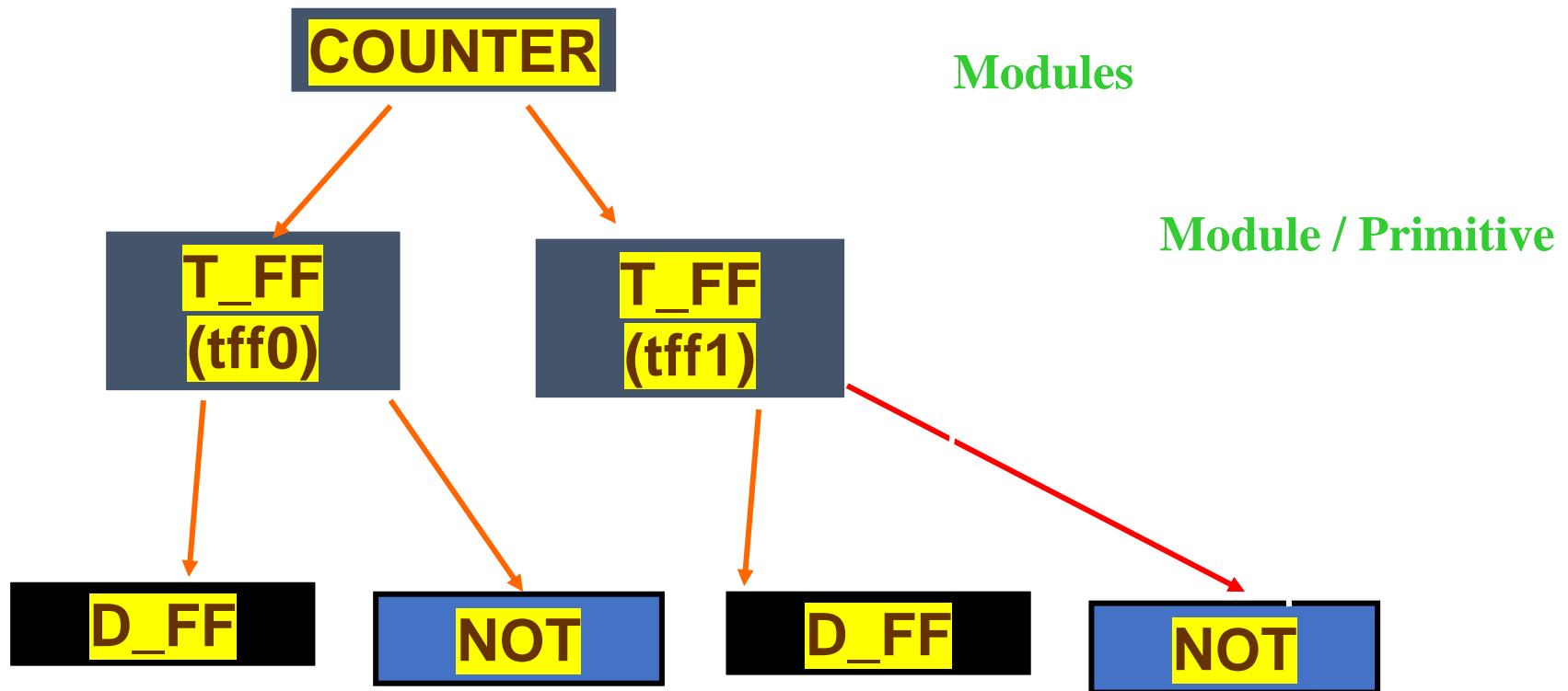
- In Verilog nesting of modules is not permitted i.e., one module definition cannot contain another module definition within the module and endmodule statements.

```
module counter(q, clk, reset);  
output [3:0]q;  
input clk, reset;
```

```
    module T_FF(q, clock, reset) // Illegal
```

```
        endmodule  
endmodule
```

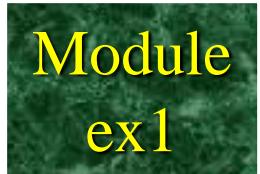
Illustration of a 'module'



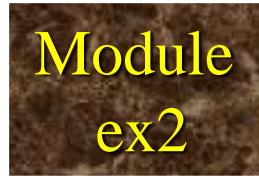
Module instances

- A module provides a template from which one can create actual objects.
- Each object has its own name, variables, parameters and I/O interface.
- The process of creating objects from a module template is called instantiation, and the objects are called instances.
- Primitives and modules can be instantiated.

Module instantiation

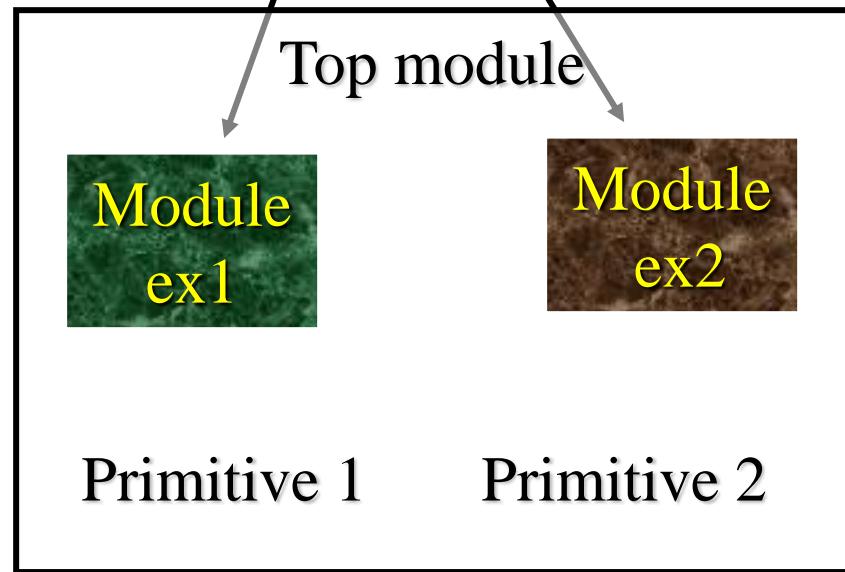


module templates



ex1 and ex2 as objects

in top module

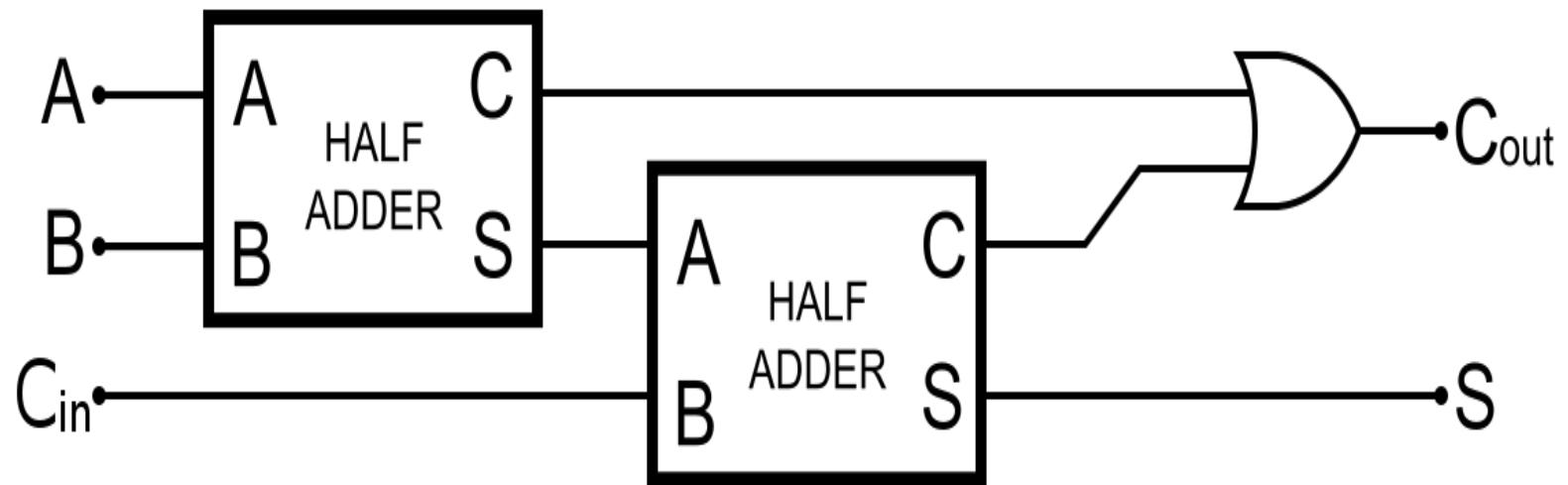


Ports

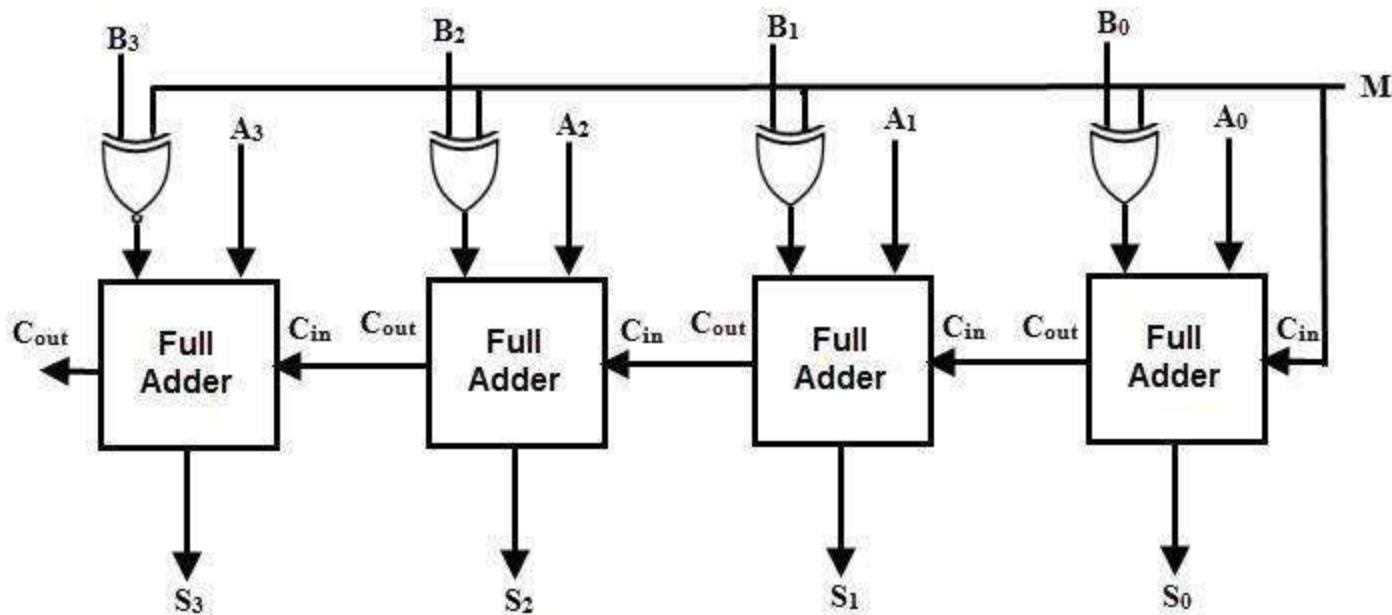
All ports in the list of ports must be declared in the module. Ports can be declared as follows:

Verilog Keyword	Type of Port
input	input port
output	output port
inout	bidirectional

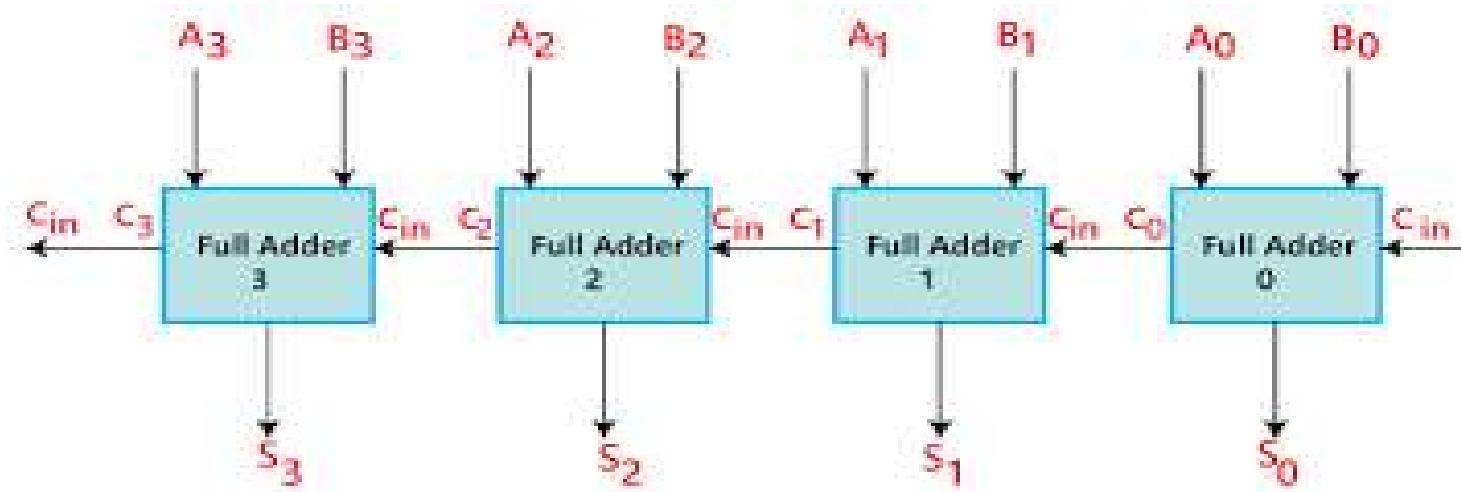
FULL ADDER



4 BIT PARALLEL ADDER

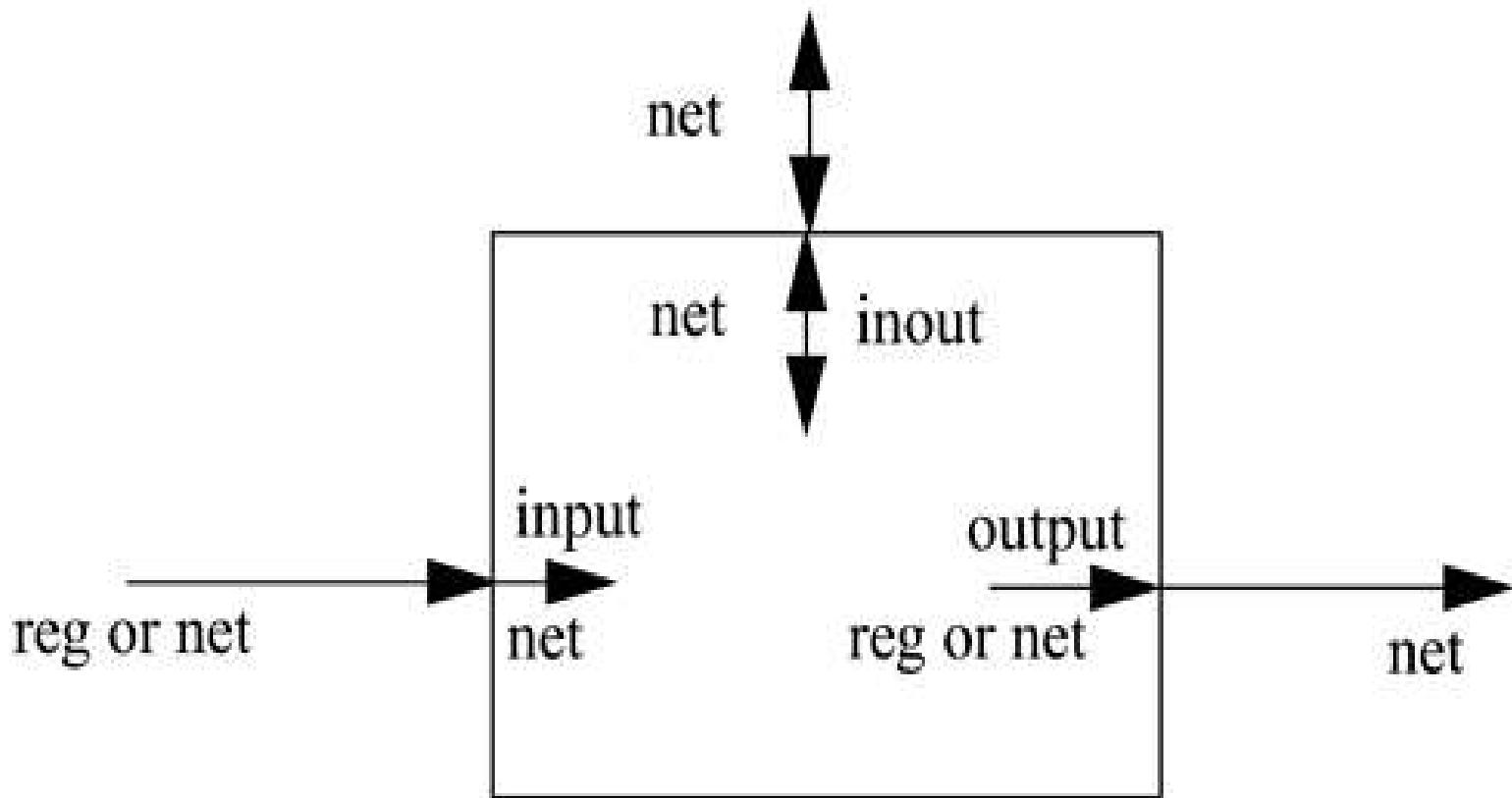


Port Declarations



```
module fulladd4(sum, c_out, a, b, c_in);
//Begin port declarations section
output [3:0] sum;
output c_cout;
input [3:0] a, b;
input c_in; //End port declarations section ...
<module internals> ...
endmodule
```

Port Connection Rules



Port Connection Rules

Inputs

Internally, input ports must always be of the type net.
Externally, the inputs can be connected to a variable
which is a reg or a net.

Outputs

Internally, outputs ports can be of the type reg or net.
Externally, outputs must always be connected to a net.
They cannot be connected to a reg.

Inouts

Internally, inout ports must always be of the type net.
Externally, inout ports must always be connected to a net.

Port Connection Rules

Width matching

It is legal to connect internal and external items of different sizes when making inter-module port connections. However, a warning is typically issued that the widths do not match.

Unconnected ports

Verilog allows ports to remain unconnected. For example, certain output ports might be simply for debugging, and you might not be interested in connecting them to the external signals. You can let a port remain unconnected by instantiating a module.

Data Types

Data types

- ❖ Net Data type
- ❖ Registers Data type

Data Type

A variable belongs to one of two data types

Net:

- Must be continuously driven
- Used to model connection between continuous assignments and Instantiations.

Register:

- Retains the last value assigned to it.
- Often used to represent storage element.

Net Data Type

- Nets represent connections / physical wires between hardware elements.
- Nets will not store / hold any value.
- Different net types supported for synthesis:
 - **wire , wor, wand, tri , supply0, supply1**
 - wire and tri are equivalent ; when there are multiple drivers, driving them, the output of the drivers are shorted together.
 - wor / wand inserts an OR/AND gate at the connection.
 - supply0 / supply1 model power supply connections.
- Default Size : 1-bit / scalar
- Default Value : z

Example for wire and wand - Net data type

```
module wired (a,b,f);  
input a,b;  
output f;  
wire f;  
assign f= a & b;  
assign f= a | b;  
endmodule
```

```
module wired_a (a,b,f);  
input a,b;  
output f;  
wand f;  
assign f= a & b;  
assign f= a | b;  
endmodule
```

Example for supply – Net data type

```
module supply_wire (a,b,c,f);
input a,b;
output f;
wire t1,t2;
supply0 gnd;
supply1 vdd;
nand G1(t1,vdd,a,b);
xor G2(t2,c,gnd);
and G3(f,t1,t2);
endmodule
```

Wire declaration examples

- wire a; // signal 'a' declared as wire
- wire out; // signal 'out' declared as wire

Ex: assign out = a | b; or o1(out, a, b);

- wire a, b; // signals 'a' & 'b' declared as wires
- wire d = 1'b0; /*net 'b' is fixed to logic value '0' at declaration*/

Registers

- ✓ In Verilog registers represent data storage elements.
- ✓ Used to model hardware memory elements / registers.
- ✓ Registers can hold / store a value.
- ✓ Declared by the keyword **reg , integer**
- ✓ Default Size : 1-bit / scalar
- ✓ Default Value : x

Other Differences

In arithmetic expression,

- An integer is treated as a 2's complement signed integer
- A reg is treated as an unsigned quantity

General Rule

- “**reg**” is used to model actual hardware registers such as counters, accumulators etc.,
- “**integer**” is used for situation like loop counting
- ❖ The reg declaration explicitly specifies the size either in scalar or vector quantity.
- ❖ For integer it takes default size, usually 32-bits

Vectors

- Nets or register data types can be declared as vectors (more no. of bits).
- If bit width is not specified then the default value is 1-bit (scalar).

```
wire a; // default scalar net value
```

```
wire [7:0] bus; // 8-bit bus
```

```
wire [31:0] busA, busB, busC; //32- bit bus
```

```
reg clock; // scalar register(default)
```

```
reg [0:40] virtual_addr; //virtual address 41 bits
```

Register declaration examples

- **reg p;**
- **reg w,y;**

```
reg reset;  
initial  
begin  
    reset = 1 'b1; //initialize reset to 1  
    #100 reset = 1 'b0; /* after 100 times units reset  
                        is de asserted */  
end
```

Addressing Vectors

- **wire** [15:0]busA;

busA[9]; // bit # 9 or 10th bit of vector busA from LSB

- **wire** [0:15]busB;

busB[9]; // bit # 9 or 7th bit of vector busB from LSB

- **reg** [31:0]cnt_out;

cnt_out[14:7]; // group of 8 bits of a vector register

cnt_out[7:14]; // is illegal addressing

Reference

1. Samir Palnitkar,"Verilog HDL: A Guide to Digital Design and Synthesis" Prentice Hall, Second Edition,2003

OPERATORS

Operator types

- The operators can be :
 - arithmetic
 - logical
 - relational
 - equality
 - bit wise
 - reduction
 - shift
 - concatenation
 - replication
 - conditional

Arithmetic operators

Expressions constitute operators and operands.

operation	symbol	operand
Multiply	*	binary
Divide	/	binary
Add	+	binary
Subtract	-	binary
Modulus	%	binary

Arithmetic operator examples

Eg: a * b // multiply a and b

a / b // divide a by b

a+b // add a and b

a - b // subtract b from a

a%b // modulus of a by b

a=3'b011 b=3'b010 d=4 e=3

c=a * b // c= 3'b110

c= a / b // c= 1

c= a+b // c= 3'b101

c= a-b // c=3'b001

c=d/e // c=1

Arithmetic operators

`13 % 4` // evaluates to 1.

`-9 % 2` // evaluates to -1, takes sign of the
first // operand

- In arithmetic operations, if any operand bit has a value x , then the result of the entire expression is x .
- The size of the result is determined by the size of the largest operand.

Logical operators

Logical operator evaluates always to a one bit value either true(1) or false (0) or x (unambiguous) . If any operand bit is either x or z it is equivalent to x

operation	symbol	operand
logical and	&&	binary
logical or		binary
logical not	!	binary

Logical operator examples

```
a1 = 1'b0; // 0 is false;
```

```
a2 = 1'b1; // 1 is true
```

a1 && a2 is 0 (false)

a1 || a2 is 1 (true)

!a2 is 0 (false)

- For vector operands, a non-zero vector is treated as logical 1.

Logical operator examples

Example:

a=10 b=00

a && b // evaluates to 0 (1 && 0)

a=2'b1x b=2'b11

a || b // is unknown, evaluates to x.

Relational operators

- Relational operations return logical 0 or 1. If there is any x or z bit in operand then it will return x.

Operation	Symbol	Operand
greater	>	Binary
less than	<	Binary
Greater than or equal to	\geq	Binary
Less than or equal to	\leq	Binary

Relational operator examples

a = 5 b = 6 c = 2'1x

a >b // evaluates to 0

a <= b // evaluates to 1

b >= c // evaluates to x

Equality operators

Equality operators are the following

Operation	Symbol	Operand
logical equality	<code>==</code>	binary
logical inequality	<code>!=</code>	binary
case equality	<code>====</code>	binary
case inequality	<code>!==</code>	binary

Equality operators

- Equality operator can return 1 or 0.
- Logical equality operator (`== !=`) will return x if any of the operand bit has x.
- Case equality operator compares both operand bit by bit including x and z bit. If it matches then returns 1 or else it returns 0. It doesn't return x.

Contd...

```
a=3; b=5; c=3'b100; d=3'b101; e=4'b1xxx;  
f=4'b1xxx;
```

```
g=3'b1xxz
```

```
a !=b // evaluates to 1.
```

```
e==f // evaluates to 1.
```

```
f==g // evaluates to 0. d == e // evaluates to x
```

Bitwise operators

Bitwise operations are performed on each bit of the operand

Operation	Symbol	Operand
Bitwise and	&	Binary
Bitwise or		Binary
Bitwise negation	~	Unary
Bitwise xor	^	Binary
Bitwise xnor	$\sim\wedge$ or $\wedge\sim$	Binary

Bitwise operators

```
module fulladd_1(sum,carry,a,b,c);
input a,b,c;
output sum,carry;
wire sum,carry;

assign sum = (a^b )^ c;
assign carry = (a&b) | (b&c) | (c&a);
endmodule
```

Bitwise operator examples

```
a = 3'b111;  b = 3'b101;  d = 3'b1x1;
```

```
c = ~a;      // c = 3'b000
```

```
c = a & b;  // c = 3'b101
```

```
c = a | b;  // c = 3'b111
```

```
c = a ^ b;  // c = 3'b010
```

```
c = a | d;  // c = 3'b1x1
```

Reduction operators

Reduction operators are unary operators

Operation	Symbol	Operand
reduction and	&	unary
reduction nand	$\sim\&$	unary
reduction or		unary
reduction nor	$\sim $	unary
reduction xor	$^$	unary
reduction xnor	$\sim^$ or $\sim\wedge$	unary

Reduction operator examples

x = 4'b01100

c = &x // c= 0 & 1&1&0&0 c=0

c = |x // c= 0|1|1|0|0 c=1

c = ^x // c=0^1^1^0^0 c=0

Shift operators

Shift operator can be shift left or shift right

Operation	Symbol	Operand
shift right	>>	unary
shift left	<<	unary

Example:

a = 4'b1011;

y = a >> 2; // y = 4'b0010, 0's filled in MSB

y = a << 2; // y = 4'b1100, 0's filled in LSB

Concatenation operators

- Concatenation operator is used to append multiple operands.
- The operand must be sized.

```
a=3'b101; b=3'b111;
```

```
y = {a,b};           // y = 6'b101111
```

```
y = {a,b,3'b010};   // y =101111010
```

Replication operators

Replication operator is used to concatenate same number.

a=3'b101 b =2'b10

y = {2{a}}; // result of y is 6'b101101

y = {2{a},2{b} }; // result of y is 10'b1011011010

y = { 2{a},2'b10}; // result of y is 6'b10110110

Conditional operators

Conditional operator ? :

format:

conditional_expr ? true_expr : false_expr;

eg:

assign out = control ? I1 : I2;

control	out
1	I1
0	I2

Conditional Operator examples

```
module mux_con(out,s0,s1,i);
input s0,s1;
input [3:0]i;
output out;
wire out;

assign out = s1 ? ( s0 ? i[3]:i[2]) : (s0 ? i[1]:i[0]) ;

endmodule
```

Operator precedence

+ - ~ !	unary
* / %	arithmetic
+ -	binary
<< >>	shift
< <= > >=	relational
& ~&	reduction and , nand
^ ~^	reduction exor, exnor
~	reduction or , nor
&&	logical and
	logical or
? :	conditional

STRUCTURAL / GATE LEVEL MODELING

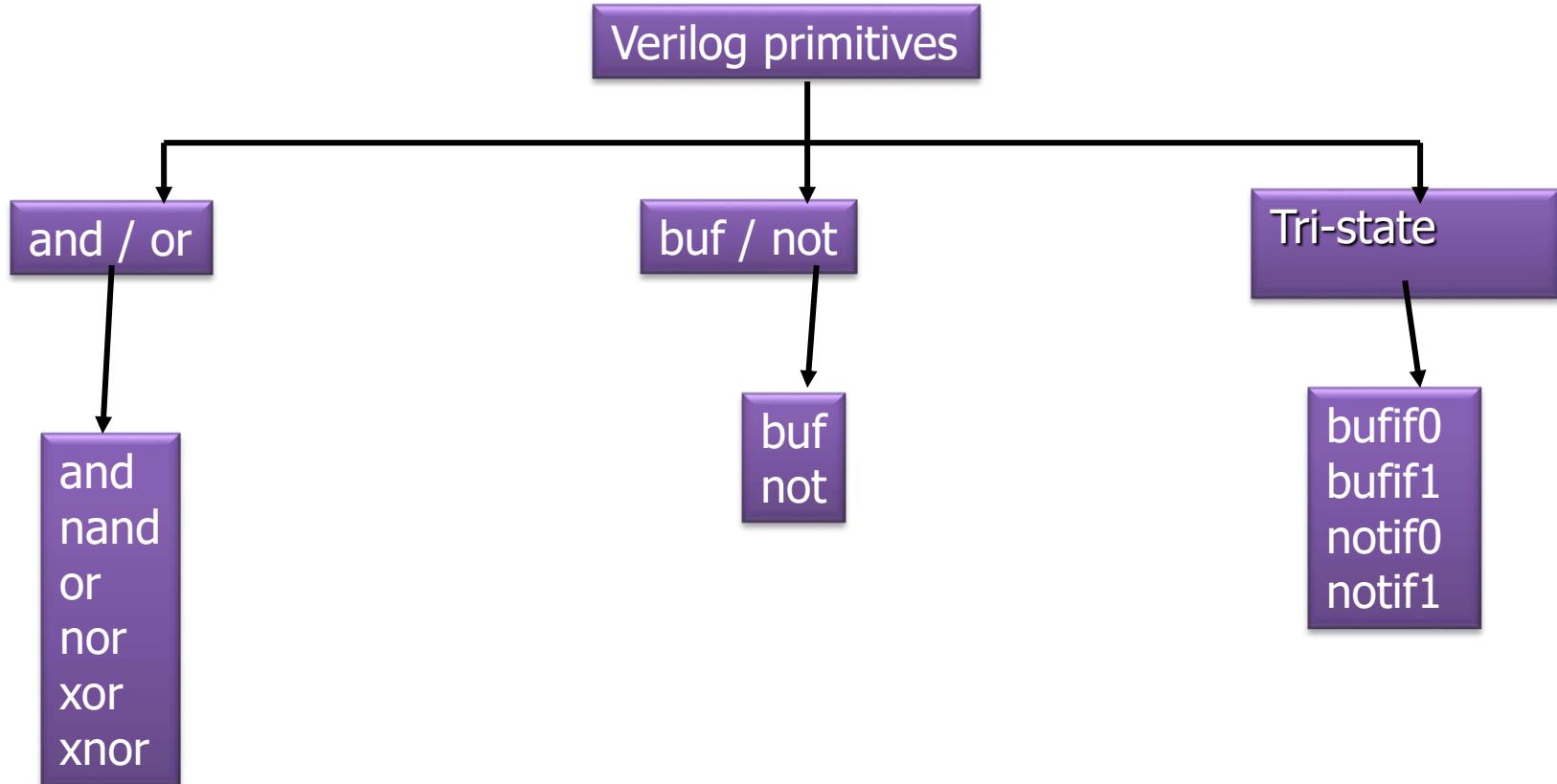
Structural / Gate Level Modeling

The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gate-level logic diagram.

Structural Level Modeling

- Features:
 - Hardware design is described using instantiations of both primitives as well as modules.
 - Logic independent of the ordering of instantiations of both primitives and modules.
 - Concurrent execution of both primitives and modules.
 - Instance name is mandatory for modules but optional for primitives.

Classification of primitives



Verilog Logic Gate Primitives

- Verilog supports basic logic gates as predefined primitives.
- These primitives are instantiated like modules except that they are predefined in Verilog.
- No module definition is needed for using the primitives.



and



nand



or



nor



xor



xnor

Primitive gates

Features:

- 1-output, multiple inputs.
 - Output transitions (0, 1, x).
-
- **and** i1 (output, input_1, input_2, ..., input_n);
 - **nand** i2 (output, input_1, input_2, ..., input_n);
 - **or** i3 (output, input_1, input_2, ..., input_n);
 - **nor** i4 (output, input_1, input_2, ..., input_n);
 - **xor** i5 (output, input_1, input_2, ..., input_n);
 - **xnor** i6 (output, input_1, input_2, ..., input_n);

AND/OR PRIMITIVES TRUTH TABLE

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

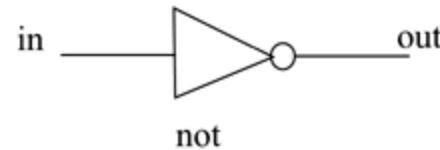
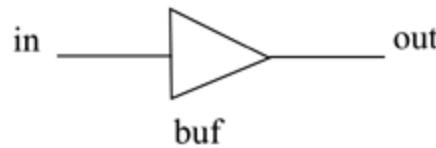
nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

buf / not primitives

Features:

- 1-input, multiple outputs.
- Output transitions (0, 1, x).



- **buf i1 (output_1, output_2, ..., output_n, input);**
- **not i2 (output_1, output_2, ..., output_n, input);**

BUF / NOT PRIMITIVES TRUTH TABLE

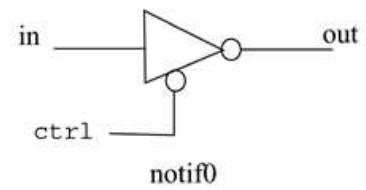
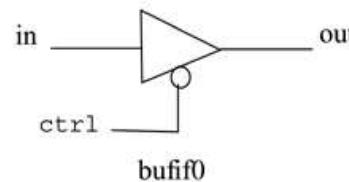
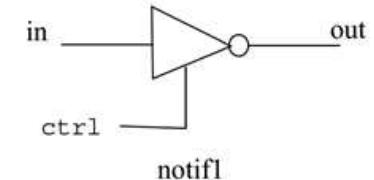
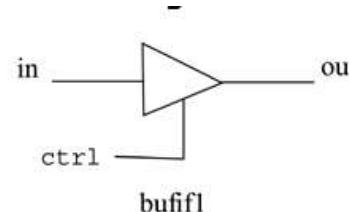
buf	
input	output
0	0
1	1
X	X
Z	X

not	
input	output
0	1
1	0
X	X
Z	X

Tri-state Primitives

Features:

- Has only 3 terminals.
- Output transitions (0, 1, x, z).



- **bufif0** i1 (output, data input, control)
- **bufif1** i2 (output, data input, control input);
- **notif0** i3 (output, data input, control input);
- **notif1** i4 (output, data input, control input);

Types of gate delays

- Gate delay: signal propagation delay from any gate input to the gate output.
- Rise delay: refers to the transition to the '1' value from any other value.
- Fall delay: refers to the transition to the '0' value from any other value.
- Turn-off delay: refers to the transition to the high-impedance value (z), from any other value.
- If the value changes to x, the minimum of the three delays is taken.

Gate delay specifications

- One delay specification: If specified, it is used for all transitions.
and #(delay time) a1 (out, i1, i2);
and #(4) a1 (out, i1, i2);
- Two delay specification: If specified, they refer to rise and fall times.
or #(rise_del, fall_del) o1 (out, i1, i2);
or #(5, 6) o1 (out, i1, i2);
- Three delay specification: If specified, they refer to rise, fall and turn-off times.
bufif1 #(rise_del, fall_del, turn_off_del) b1 (out, in, cnt);
bufif1 #(2, 3, 5) b1 (out, in, ctrl);

MIN : TYP : MAX VALUES

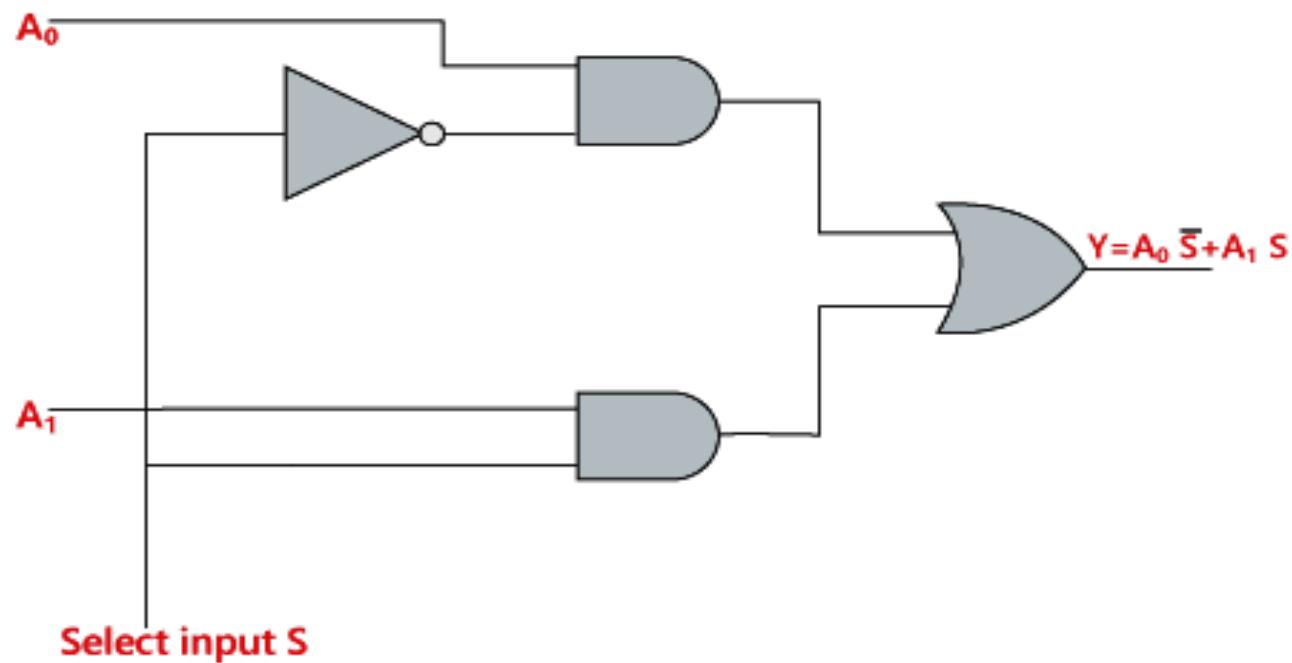
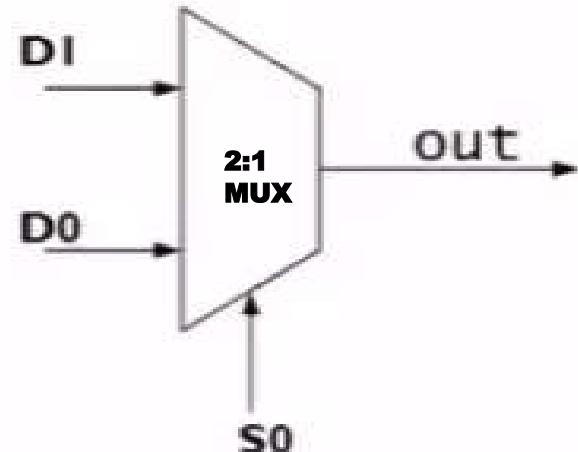
- Primitive gate delays allow three values each for the rise, fall and turn-off delays.
- The three values are minimum, typical and maximum, and the three are separated by colons.
- Either of the three values can be selected at the start of the simulation (run time). If no value is selected, typical value is the default.

EXAMPLES OF MIN : TYP : MAX VALUES

- One delay specification with min:typ:max values.
and #(2:4:5) a1 (out, i1, i2);
 $\text{rise}_{\min}, \text{fall}_{\min} = 2$, $\text{rise}_{\text{typ}}, \text{fall}_{\text{typ}} = 4$, $\text{rise}_{\max}, \text{fall}_{\max} = 5$.
- Two delay specification with min:typ:max values.
or #(1:5:3, 2:6:4) o1 (out, i1, i2);
 $\text{rise}_{\min} = 1$, $\text{rise}_{\text{typ}} = 5$, $\text{rise}_{\max} = 3$, $\text{fall}_{\min} = 2$, $\text{fall}_{\text{typ}} = 6$, $\text{fall}_{\max} = 4$.
- Three delay specification with min:typ:max values.
bufif1 #(1:2:4, 1:3:5, 3:5:6) b1 (out, i1, i2);
 $\text{rise}_{\min} = 1$, $\text{rise}_{\text{typ}} = 2$, $\text{rise}_{\max} = 4$, $\text{fall}_{\min} = 1$, $\text{fall}_{\text{typ}} = 3$, $\text{fall}_{\max} = 5$, $\text{turn-off}_{\min} = 3$, $\text{turn-off}_{\text{typ}} = 5$, $\text{turn-off}_{\max} = 6$.

2:1 MULTIPLEXER

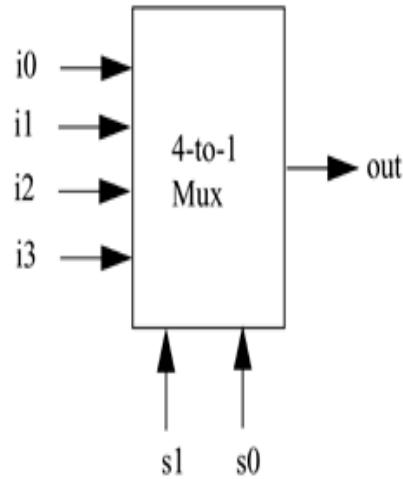
- 2:1 Multiplexer has
 - two data inputs, D0 and DI
 - one select line S0.
 - One output Z.



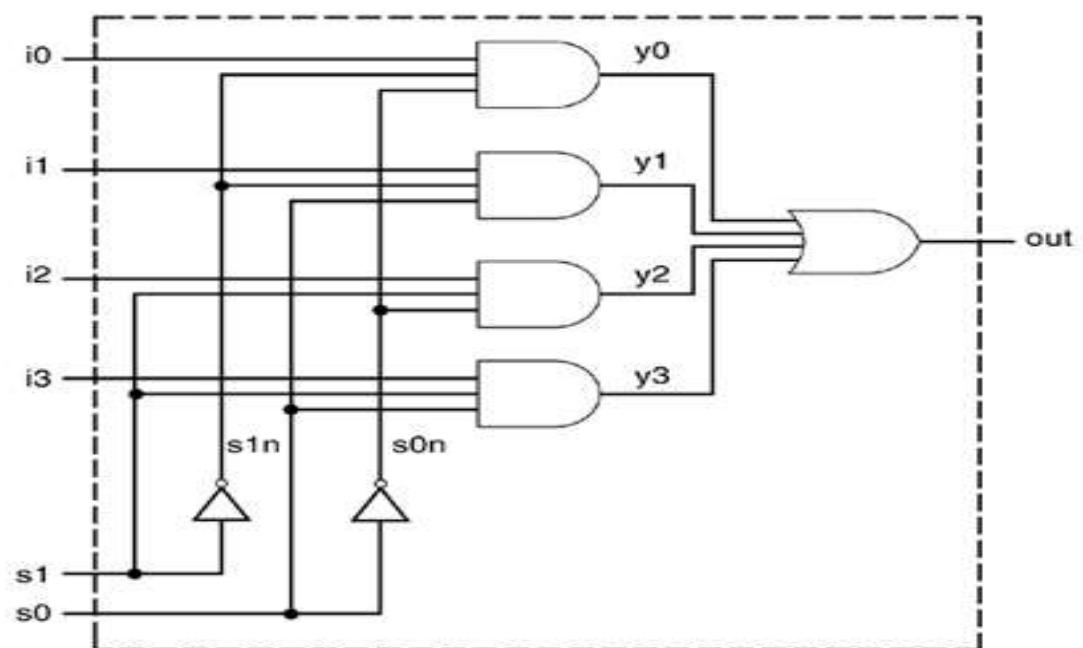
Module Instantiation - Example

```
module twmux (a,b,s,y);
input a,b,s;
output y;
wire y,s1,w1,w2;
not n1(s1,s);
and a1(w1,a,s);
and a2 (w2,b,s1);
or o1(y,w1,w2);
endmodule
```

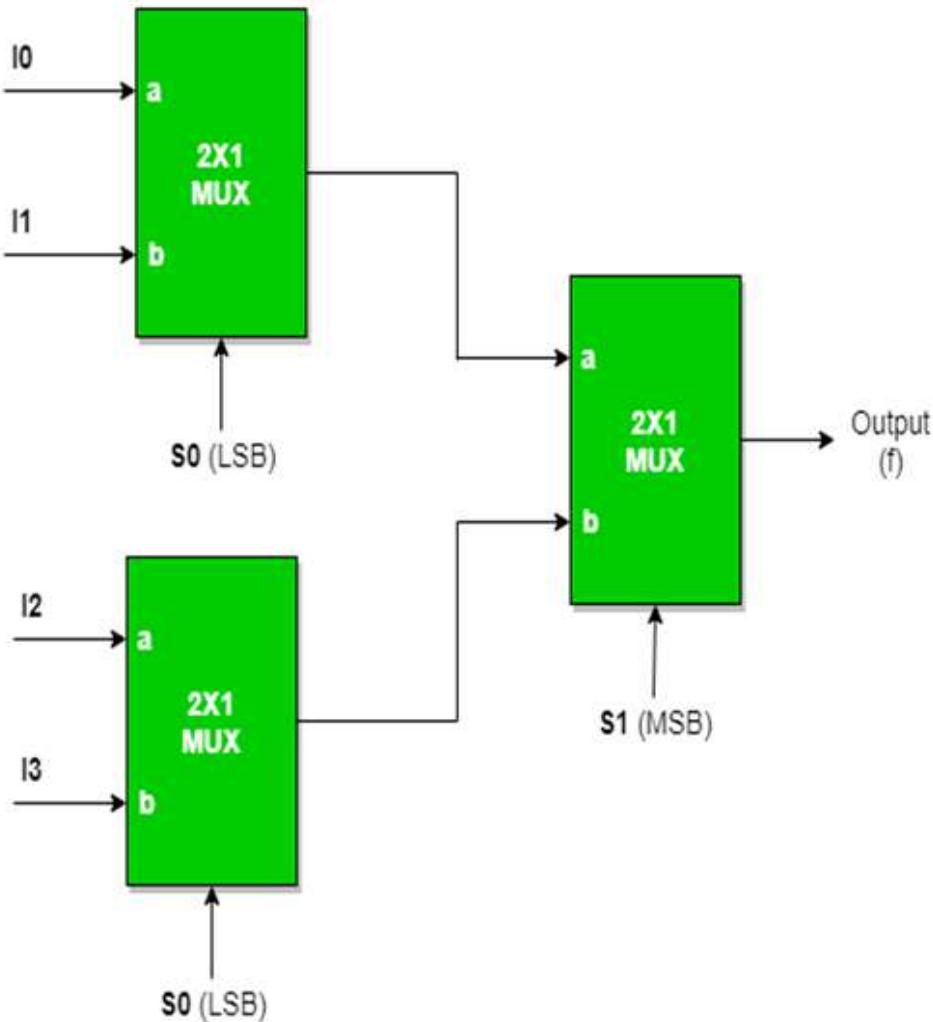
Example - Primitive Instantiation



s_1	s_0	out
0	0	i_0
0	1	i_1
1	0	i_2



Inputs



Truth Table

S ₀	S ₁	f
0	0	I ₀
0	1	I ₁
1	0	I ₂
1	1	I ₃

Primitive Instantiation - Example

```
// Module 4-to-1 multiplexer. Port list is taken exactly from the  
I/O diagram.
```

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);  
output out;  
input i0, i1, i2, i3;  
input s1, s0;  
wire s1n, s0n;  
wire y0, y1, y2, y3;  
not (s1n, s1); not (s0n, s0);  
and (y0, i0, s1n, s0n);  
and (y1, i1, s1n, s0);  
and (y2, i2, s1, s0n);  
and (y3, i3, s1, s0);  
or (out, y0, y1, y2, y3);  
endmodule
```

Connecting Ports to External Signals

- There are two methods of making connections between signals specified in the module instantiation and the ports in a module definition. These two methods cannot be mixed.
 - Connecting by ordered list
 - Connecting ports by name

Connecting by ordered list

- Connecting by ordered list is the most intuitive method for most beginners.
- The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition

Connecting by ordered list-Example

```
module twmux (a,b,s,y);  
input a,b,s;  
output y;  
wire y,s1,w1,w2;  
not n1(s1,s);  
and a1(w1,a,s);  
and a2 (w2,b,s1);  
or o1(y,w1,w2);  
endmodule
```

```
module frmux (a,b,c,d,sel, y);  
input a,b,c,d,sel;  
output y;  
wire y,sel,w1,w2;  
twmux t1(a,b,sel,w1);  
twmux t2(c,d,sel,w2);  
twmux t3(w1,w2,sel,y);  
endmodule
```

Connecting Ports by name

- For large designs where modules have, say, 50 ports, remembering the order of the ports in the module definition is impractical and error-prone.
- Verilog provides the capability to connect external signals to ports by the port names, rather than by position.

Connecting Ports by name -Example

```
module twmux (a,b,s,y);  
input a,b,s;  
output y;  
wire y,s1,w1,w2;  
not n1(s1,s);  
and a1(w1,a,s);  
and a2 (w2,b,s1);  
or o1(y,w1,w2);  
endmodule
```

```
module frmux (a,b,c,d,sel, y);  
input a,b,c,d,sel;  
output y;  
wire y,sel1,w1,w2;  
twmux t1(.a(a), .b(b), .s(sel), .y(w1) );  
twmux t2(.a (c), .b(d), .s(sel), .y(w2) );  
twmux 3(.a(w1), .b(w2), .s(sel), .y(y));  
endmodule
```

Behavioral timing controls

Procedural timing controls

- Delay based timing control
- Event based timing control
- Level sensitive timing control

Delay based timing control

- Regular delay control
- Intra assignment delay control
- Zero delay control

Regular delay control

Format:

```
#<delay_value> <target> = RHS expression;
```

- It is used to delay the execution of the corresponding procedural statement by the defined delay value.
- Regular delays defer the execution of the entire assignment.

e.g. : #10 c = a + b;
#(2:4:6,3:5:7) or_out = a | b;

Intra-assignment delay control

Format:

<target> = #<delay_value> RHS expression;

- Intra-assignment delays compute the RHS expression at the current time and defer the assignment of the computed value to the LHS target by the defined delay value.

e.g. : c = #10 a + b;

Zero delay control

- Statements with zero delay execute at the end of the current simulation time, when all other statements in the current simulation time have executed.

```
initial  
begin  
    a = 1'b1;  
    b = 1'b0;  
end
```

```
initial  
begin  
    #0 a = 1'b0;  
    #0 b = 1'b1;  
end
```

Event based timing control

- An event is the change in the value on a register or a net.
- Implicit event: The value changes on nets and registers can be used as events to trigger the execution of a statement.
- The event can also be based on the direction of the change.
 - **posedge** : change towards the value 1
 - **negedge** : change towards the value 0

Types of event based timing control

- Regular event control
- Named event control
- Event **or** control

Regular event control

Format:

@(event) statement;

e.g.: @ (ena) q = d;

@ (**posedge** t_in) q = ~q;

q = @ (**negedge** clk) d;

@ (**posedge** clk_in) q_out = d_in;

Named event control

- Events can be declared with an identifier.
- The declared can be triggered conditionally or unconditionally, & these doesn't hold any data.
- The triggered events can be made to execute the a block of statements / assignments waiting for this trigger.
- Event is declared by keyword: **event** event_name;
- Event triggered as: -> event_name;
- Usage: @ <event_name>

Example of named event control

```
module ex1(out, sig);
output out;
input sig;
reg out;
wire sig;
event shoot;
always @ (negedge sig)
begin
if (sig == 1'b0)
-> shoot;
else
out = 1'b0;
end
always @ (shoot)
//if (shoot == 1'b1)
out = 1'b1;
```

Event **or** control

- The logical or of any number of events can be expressed such that the occurrence of any one of the events triggers the execution of the procedural statement that follows it.

Format:

always @ (event1 **or** event2 **or** event3)

always @ (**posedge** event1 **or negedge** event2)

Event or control -Example

```
always @ (in1 or in0 or  
sel_in)
```

```
begin
```

```
if (sel_in == 1'b0)
```

```
mux_out = in0;
```

```
else
```

```
mux_out = in1;
```

```
end
```

```
always @ (posedge clk_in or  
negedge set_in_n)
```

```
begin
```

```
if (set_in_n == 1'b0)
```

```
q_out = 1'b1;
```

```
else
```

```
q_out = d_in;
```

```
end
```

Level sensitive control

- Execution of statements will be delayed (wait) till a condition becomes true.
- The nature of the wait statement is level-sensitive, as opposed to basic event control (specified by the @ character), which is edge-sensitive.

Format:

wait (condition) <statement>

Level sensitive control - Example

```
module lat(d_in, ena_in_n, q_out);
output q_out;
input d_in, ena_in_n;

reg q_out;
wire d_in, ena_in_n;

always @(d_in or ena_in_n)
begin
  wait (ena_in_n == 1'b0)
  q_out = d_in;
end
endmodule
```

Behavioral constructs

if-else statements

- The if construct checks a specific condition and decides execution based on the result.

assignment1;

if(condition) **assignment2;**

assignment3;

assignment4;

- After execution of **assignment1**, the condition specified is checked . If it is satisfied , **assignment2** is executed; if not it is skipped.
- In either case the execution continues through **assignment3**, **assignment4**, etc.
- Execution of **assignment2** alone is dependent on the condition. The rest of the sequence remains.

if-else and if-elseif-else statements

```
if (<expression>)
begin
    true_statements;
end
```

```
if (<expression>)
begin
    true_statements;
end
else
begin
    false statements;
end
```

```
if (<expression 1>)
begin
    true_statements 1;
end
:
:
else if (<expression n>)
begin
    true_statements n;
end
else
begin
    default_statements;
end
```

Conditional statement - Examples

```
if (!ena = 1'b1)
begin
    out = in1 & in2;
end
```

```
if (sel == 1'b0)
begin
    mux_out = in0;
end
else
begin
    mux_out = in1;
end
```

```
if (in1 == 1'b0 && in2 == 1'b0)
dec_o1 = 1'b1;

else if (in1 == 1'b0 && in2 == 1'b1)
dec_o2 = 1'b1;

else if (in1 == 1'b1 && in2 == 1'b0)
dec_o3 = 1'b1;

else
dec_o4 = 1'b1;
```

case statement

- This is a multiway decision statement that tests whether an expression matches one of a number of other alternatives.
- Doesn't treat 'x' & 'z' as don't cares.

```
case (expression)
alternative 1:begin
    end
alternative 2:begin
    end
alternative 3:begin
    end
default: begin
    end
endcase
```

case statement - Example

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0); output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0})
2'd0 : out = i0;
2'd1 : out = i1;
2'd2 : out = i2;
2'd3 : out = i3;
default: $display("Invalid control signals");
endcase
endmodule
```

casez statements

- **casez**: same as **case** statement but treats 'z' as don't cares.
- All bit positions with 'z' can also be represented by '?'

casez - Examples

module

casez_ex(counter,cmd);

input[0:3] counter;

output cmd;

reg cmd;

always@(counter)

casez(counter)

4'b???1:cmd=0;

4'b??10:cmd=1;

4'b?100:cmd=2;

4'b1000:cmd=3;

default : cmd=0;

endcase

endmodule

casex statement

- **casex**: same as **case** / **casez** statement but treats all 'x' and 'z' as don't cares.

casex - Example

```
module casex_ex(sel,Bitposition);
input [5:1] sel;
output [2:0] Bitposition;
reg [2:0] Bitposition;
always @ (sel)
casex(sel)
5'bxxxx1 : Bitposition=1 ;
5'bxxx1x : Bitposition=2;
5'bxx1xx : Bitposition=3;
5'bx1xxx : Bitposition=4;
5'b1xxxx : Bitposition=5;
default : Bitposition=0;
endcase
endmodule
```

Looping statements

- Executes the statements for zero, one or more number of times.
- Four types of looping statements.
 - while
 - for
 - repeat
 - forever

for loop construct

The for loop in Verilog is quite similar to the for loop in C
Format:

```
for (initialization; expression; incr/dcr)
    begin
        statements
    end
```

The sequence of execution as follows

1. Execute **initialization**
2. Evaluate **expression**
3. If the **expression** evaluates to the true state(1), carry out **statements**. Go to step 5.
4. If the **expression** evaluates to false state (0), exit the loop.
5. Execute **incr/dcr**. Go to step 2.

for loop - Example

```
module addfor (s,co,a,b,cin,en);
output [7:0] s;
output co;
input [7:0] a,b;
input en,cin;
reg [8:0] c;
reg co;
reg [7:0] s;
integer i;
always @ (posedge en)
begin
c[0] = cin;
```

```
for (i=0; i<=7; i=i+1)
begin
{c[i+1],s[i]} = (a[i] + b[i] +
c[i]);
end
co=c[8]
end
endmodule
```

repeat construct

- The repeat construct is used to repeat specified block a specified number of times.
- The format is show below , the quantity ‘a’ can be a number or an expression evaluated to a number.
- As soon as the repeat statement is encountered, a is evaluated, the block will be executed ‘a’ times.
- If ‘a’ evaluates to 0 or x or z, the block is not executed.

Format:

repeat (a)

begin

statements

end

repeat construct - Example

```
module clkgen (en, cnt);
input en;
output cnt;
integer cnt;
always @ (en)
Begin
if (en)
cnt=0;
repeat (105)

#10 cnt = cnt + 1'b1;
end
endmodule
```

While loop

The *format* of while loop is shown below.

- The **expression** is evaluated. If it is *true* , the **statements** executed and **expression** evaluated and checked again.
- If the expression evaluates to *false*, the loop is terminated and the following statement is taken for execution. If the expression evaluates to *true*, execution of **statement** is repeated.
- The loop is terminated and broken only if the expression evaluates to false.

Format:

```
while (expression)
begin
    statements
end
```

While loop - Example

```
module while2 (b,n,en,clk);      @(posedge clk)
input [7:0] n;                  a=a-1'b1;
input clk,en;                  end
output b;                      b=1'b0;
reg [7:0] a;                  end
reg b;                         initial
always@(posedge en)           b=1'b0;
begin                           endmodule
a=n;
while( | a)
begin
b=1'b1;
```

forever loop

Repeated execution of a block in an endless manner is best done with the **forever** loop (compare with repeat where the repetition is for a fixed number of times)

Format:

```
forever  
begin  
    statements;  
end
```

forever LOOP - Example

```
module clkgen (clk);
output clk;
reg clk;
initial
begin
    clk=1'b0;
    forever #10 clk=~clk;
end
initial
#1000 $finish;

endmodule
```

DATAFLOW MODELING

Dataflow Modeling (1/2)

- Gate level Modeling approach works well for small circuits but not for complex designs.
- Designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level.
- Dataflow modeling provides a powerful way to implement a design.
- Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates.

Dataflow Modeling (2/2)

- With gate densities on chips increasing rapidly, dataflow modeling has assumed great importance.
- Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called logic synthesis.
- The data flow modeling allows the designer to concentrate on optimizing the circuit in terms of data flow.
- In the digital design community, the term RTL (Register Transfer Level) design is commonly used for a combination of dataflow modeling and behavioral modeling.

Continuous assignment (1/2)

- This level of abstraction level resembles like that of boolean equation representation of digital systems.
- The dataflow assignments are called “Continuous Assignments”.
- Continuous assignments are always active.
- Syntax: *assign #(delay) target = expression;*
- A Verilog module can contain any number of continuous assignment statements, and all staements execute concurrently.

Continuous assignment (2/2)

- LHS target -> always a net, *not a register*.
- RHS -> registers, nets or function calls.
- Delay values can be specified in terms of time units.
- Whenever an event (change of value) occurs on any operand used on the RHS of an expression, the expression is evaluated and assigned to the LHS target.

Types of continuous assignments

- Regular continuous assignment.

```
wire mux_out, a_in, b_in, sel_in;  
assign mux_out = sel_in ? b_in : a_in;
```

- Implicit continuous assignment.

```
wire a_in, b_in, sel_in;  
wire mux_out = sel_in ? b_in : a_in;
```

Continuous assignment examples

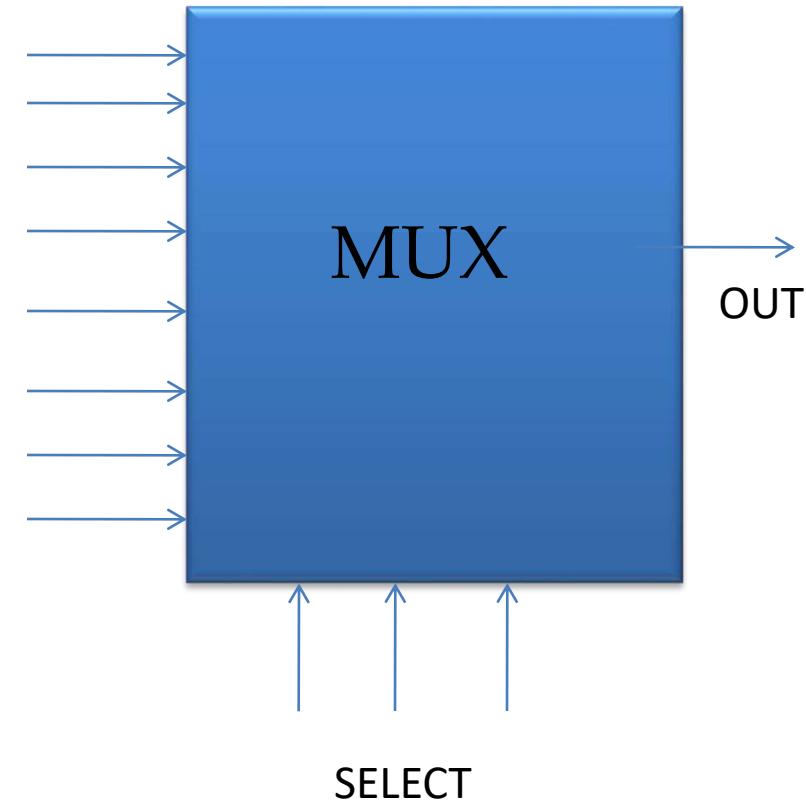
wire and_out;

assign and_out = i1 & i2; // regular continuous assignment

wire exor_out = i1[31:0] ^ i2[31:0] // implicit assignment

Example - Mux Inference

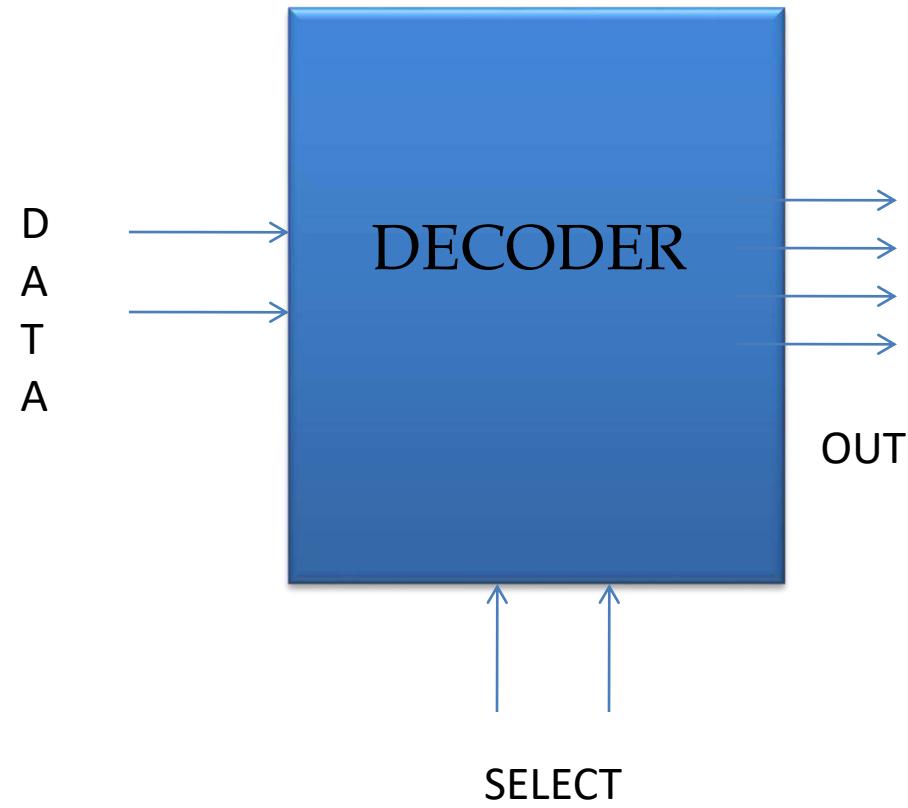
```
module generate_mux  
  (data,select,out);  
  
  input [0:7] data;  
  input [0:2]select;  
  output out;  
  
  wire out;  
  
  assign out = data[select];  
  
endmodule
```



Note: Non-constant index in expression on RHS generates MUX

Example - Decoder Inference

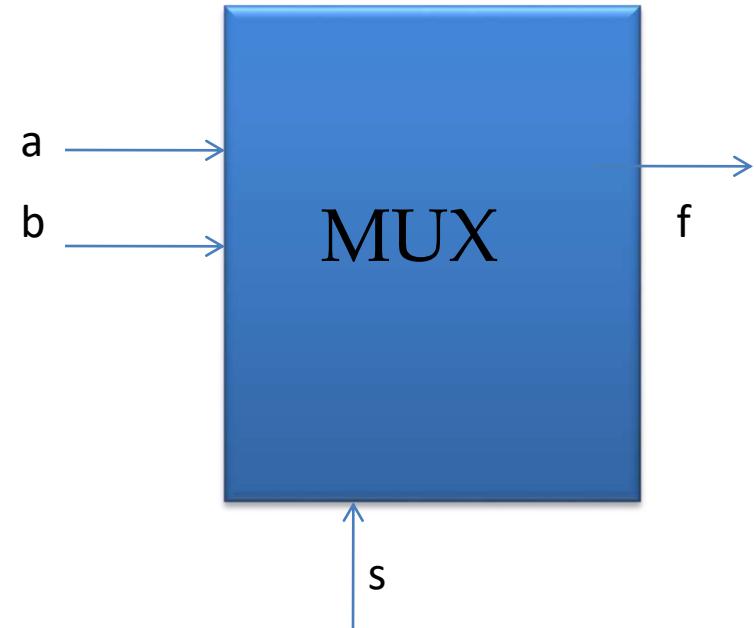
```
module generate decoder  
  (data,select,out);  
  
  input data;  
  input [0:1]select;  
  output [0:3] out;  
  
  wire [0:3] out;  
  
  assign out[select]=data;  
  
endmodule
```



Note: Non- constant index in expression on LHS generates DECODER

Example

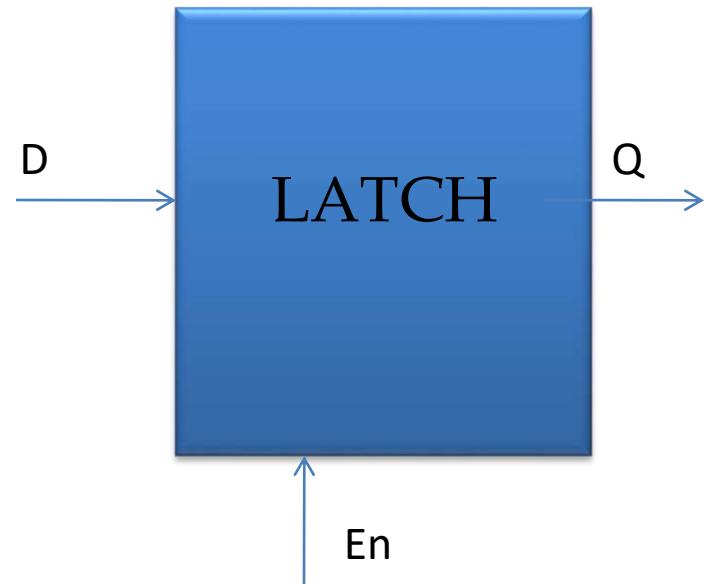
```
module generate_mux  
  (a,b,f,s);  
  input a,b;  
  input s;  
  output f;  
  wire f;  
  assign f = s? a : b;  
 endmodule
```



Note: Conditional Operator generate MUX

Example

```
module latch (D,Q,En);
input D,En;
output Q;
wire Q;
assign Q=En?D:Q;
endmodule
```



Note: Using “assign” to describe sequential logic using conditional operator.
Cyclic dependency of net also infers latch.

Ways to specifying delays

Delay values control the time between the change in a right hand-side operand and when the new value is assigned to the left-hand side.

- Regular assignment delay
- Implicit continuous assignment delay
- Net declaration delay

Regular assignment delay

- The delay value is specified after the keyword **assign**.
- Any change in values of in1 or in2 will result in a delay of 10 time units before recomputation of the expression in1 & in2, and the result will be assigned to out.

Example:

```
module and_ex(and_out , a_in, b_in);
  input a_in, b_in;
  output and_out;
  wire and_out;
  assign #10 and_out = a_in & b_in;
endmodule
```

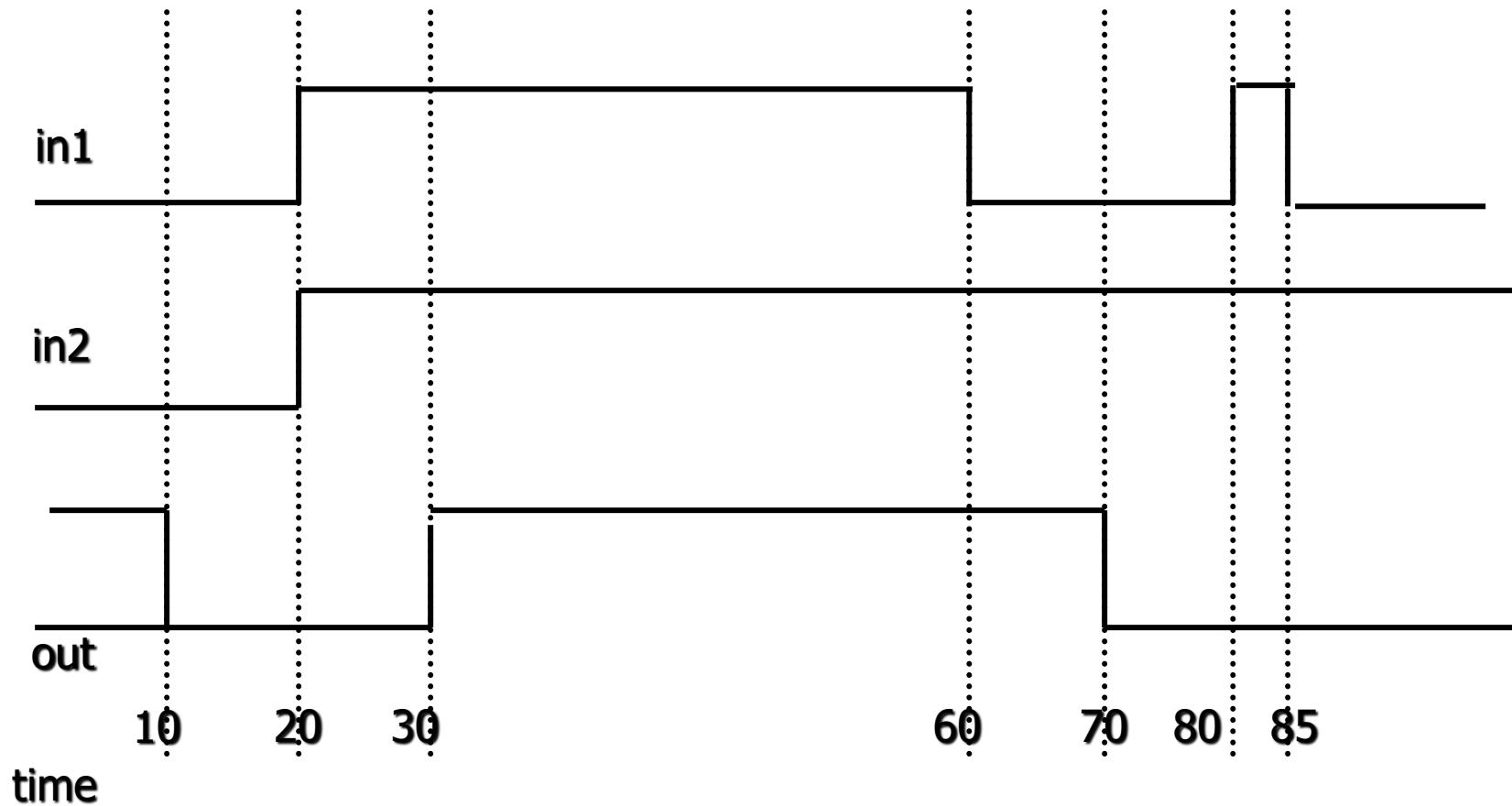
Inertial delay

wire out;

```
assign #10 out = i1 & i2; // Regular assignment delay
```

- The inertial delay of real circuits is modeled through regular assignment delay.
- Any event on the RHS signals which is not lasting for the amount of inertial delay specified will not have any effect on the LHS target.

Inertial delay



Implicit continuous assignment delay

An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.

Example:

```
module and_ex(and_out , a_in, b_in);  
  input a_in, b_in;  
  output and_out;  
  wire #10 and_out = a_in & b_in;  
endmodule
```

Net declaration delay

- A delay can be specified on a net when it is declared without putting a continuous assignment on the net.
- If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly.

Example:

```
module and_ex(and_out , a_in, b_in);
  input a_in, b_in;
  output and_out;
  wire #10 and_out;
  assign and_out = a_in & b_in;
endmodule
```

More “assign” Statements

DATAFLOW MODELING

Dataflow Modeling (1/2)

- Gate level Modeling approach works well for small circuits but not for complex designs.
- Designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level.
- Dataflow modeling provides a powerful way to implement a design.
- Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates.

Dataflow Modeling (2/2)

- With gate densities on chips increasing rapidly, dataflow modeling has assumed great importance.
- Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called logic synthesis.
- The data flow modeling allows the designer to concentrate on optimizing the circuit in terms of data flow.
- In the digital design community, the term RTL (Register Transfer Level) design is commonly used for a combination of dataflow modeling and behavioral modeling.

Continuous assignment (1/2)

- This level of abstraction level resembles like that of boolean equation representation of digital systems.
- The dataflow assignments are called “Continuous Assignments”.
- Continuous assignments are always active.
- Syntax: *assign #(delay) target = expression;*
- A Verilog module can contain any number of continuous assignment statements, and all staements execute concurrently.

Continuous assignment (2/2)

- LHS target -> always a net, *not a register*.
- RHS -> registers, nets or function calls.
- Delay values can be specified in terms of time units.
- Whenever an event (change of value) occurs on any operand used on the RHS of an expression, the expression is evaluated and assigned to the LHS target.

Types of continuous assignments

- Regular continuous assignment.

```
wire mux_out, a_in, b_in, sel_in;  
assign mux_out = sel_in ? b_in : a_in;
```

- Implicit continuous assignment.

```
wire a_in, b_in, sel_in;  
wire mux_out = sel_in ? b_in : a_in;
```

Continuous assignment examples

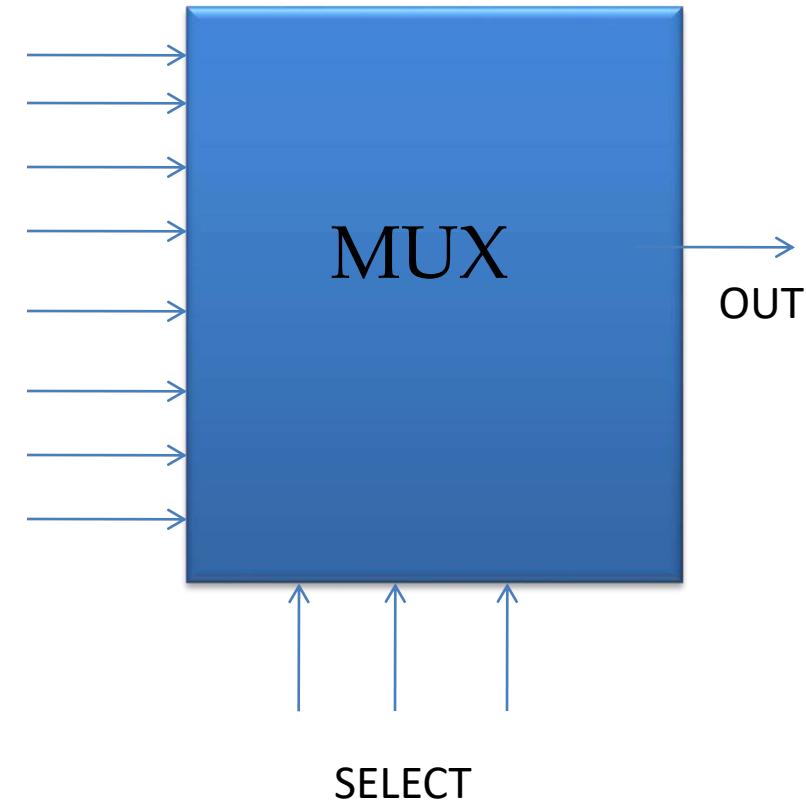
wire and_out;

assign and_out = i1 & i2; // regular continuous assignment

wire exor_out = i1[31:0] ^ i2[31:0] // implicit assignment

Example - Mux Inference

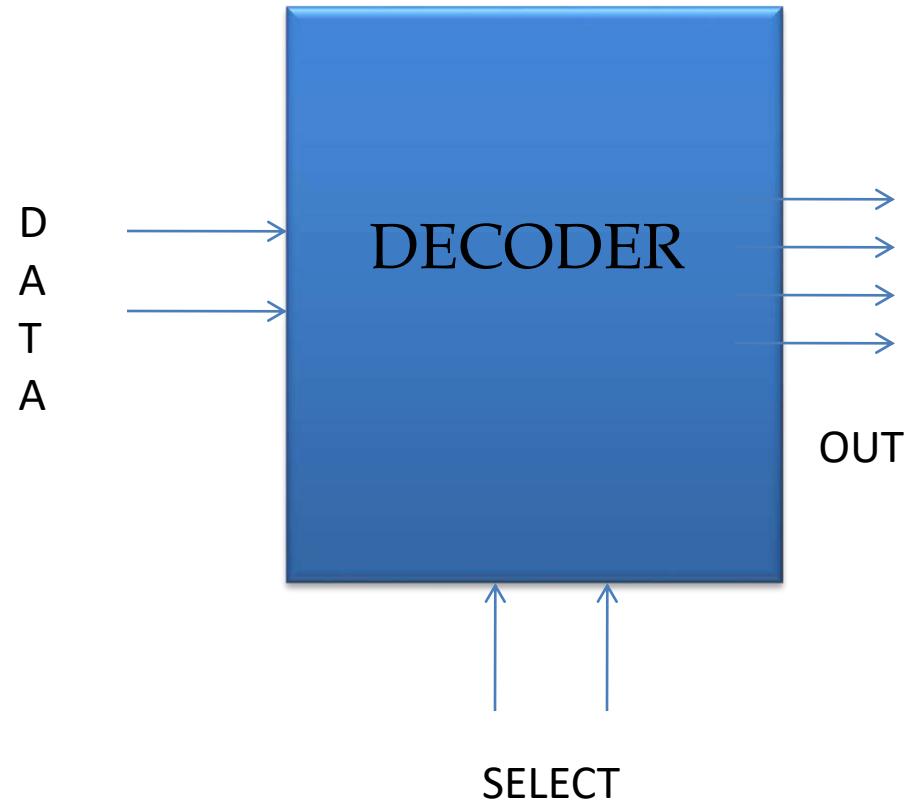
```
module generate_mux  
  (data,select,out);  
  
  input [0:7] data;  
  input [0:2]select;  
  output out;  
  
  wire out;  
  
  assign out = data[select];  
  
endmodule
```



Note: Non-constant index in expression on RHS generates MUX

Example - Decoder Inference

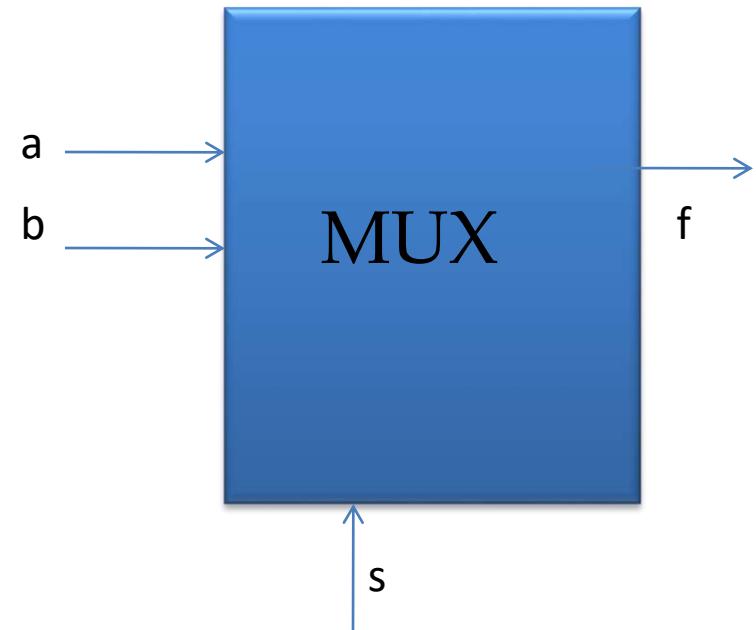
```
module generate decoder  
  (data,select,out);  
  
  input data;  
  input [0:1]select;  
  output [0:3] out;  
  
  wire [0:3] out;  
  
  assign out[select]=data;  
  
endmodule
```



Note: Non- constant index in expression on LHS generates DECODER

Example

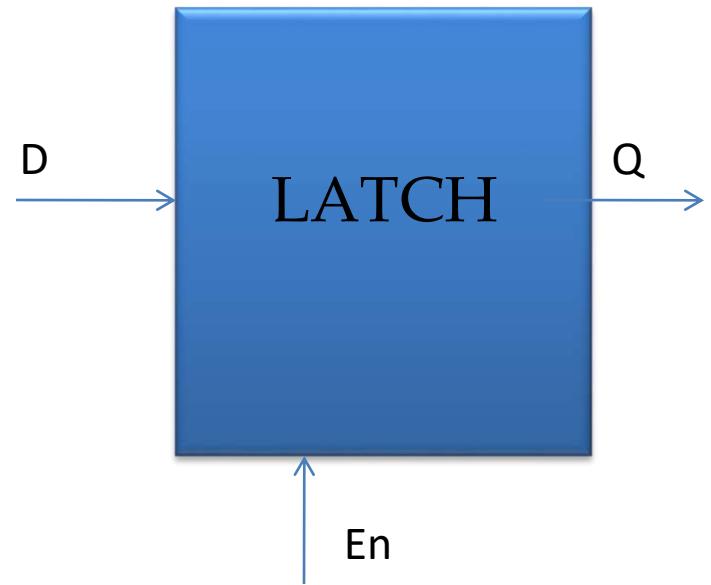
```
module generate_mux  
  (a,b,f,s);  
  input a,b;  
  input s;  
  output f;  
  wire f;  
  assign f = s? a : b;  
endmodule
```



Note: Conditional Operator generate MUX

Example

```
module latch (D,Q,En);
input D,En;
output Q;
wire Q;
assign Q=En?D:Q;
endmodule
```



Note: Using “assign” to describe sequential logic using conditional operator.
Cyclic dependency of net also infers latch.

Ways to specifying delays

Delay values control the time between the change in a right hand-side operand and when the new value is assigned to the left-hand side.

- Regular assignment delay
- Implicit continuous assignment delay
- Net declaration delay

Regular assignment delay

- The delay value is specified after the keyword **assign**.
- Any change in values of in1 or in2 will result in a delay of 10 time units before recomputation of the expression in1 & in2, and the result will be assigned to out.

Example:

```
module and_ex(and_out , a_in, b_in);
  input a_in, b_in;
  output and_out;
  wire and_out;
  assign #10 and_out = a_in & b_in;
endmodule
```

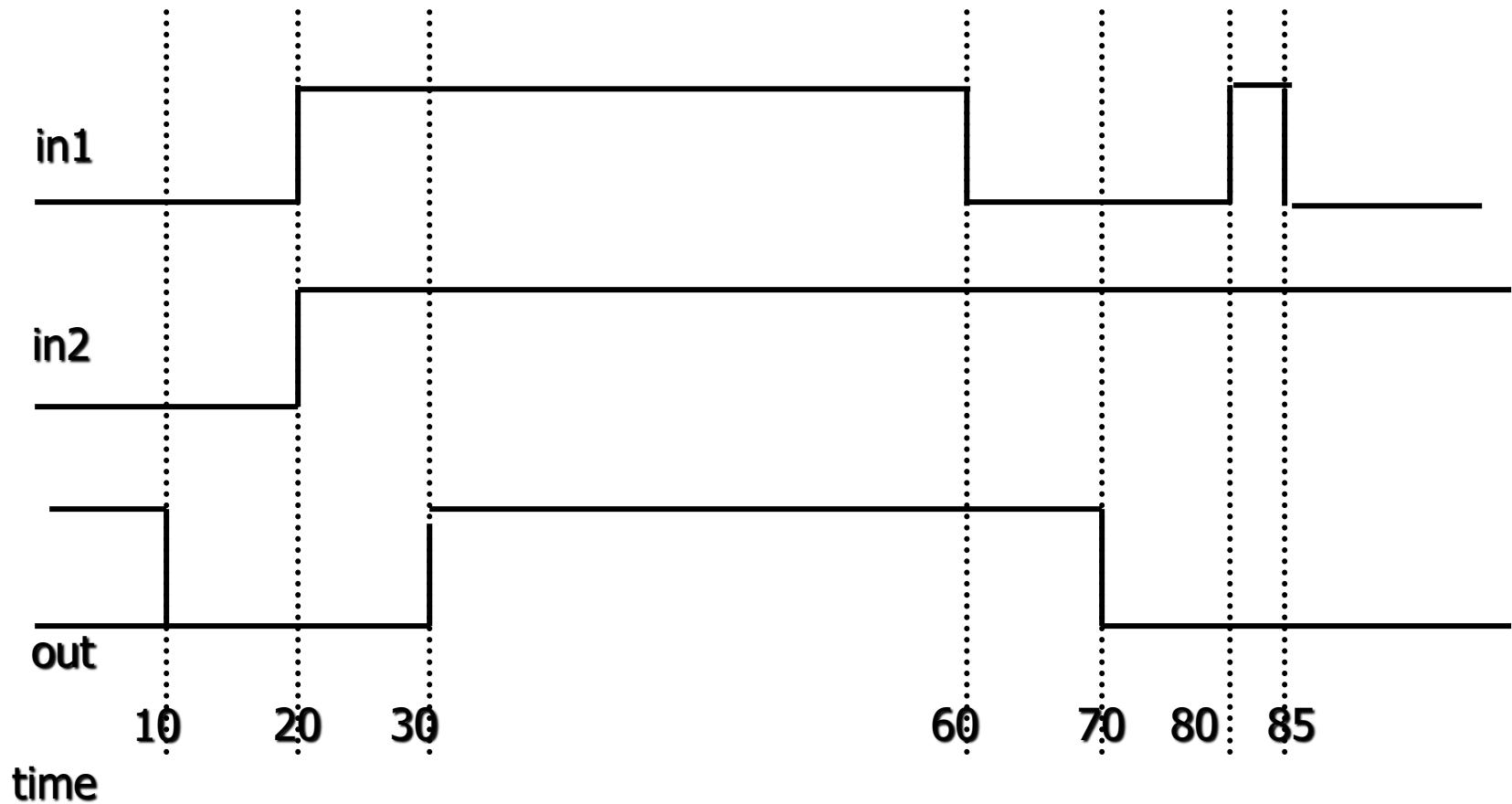
Inertial delay

wire out;

```
assign #10 out = i1 & i2; // Regular assignment delay
```

- The inertial delay of real circuits is modeled through regular assignment delay.
- Any event on the RHS signals which is not lasting for the amount of inertial delay specified will not have any effect on the LHS target.

Inertial delay



Implicit continuous assignment delay

An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.

Example:

```
module and_ex(and_out , a_in, b_in);  
  input a_in, b_in;  
  output and_out;  
  wire #10 and_out = a_in & b_in;  
endmodule
```

Net declaration delay

- A delay can be specified on a net when it is declared without putting a continuous assignment on the net.
- If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly.

Example:

```
module and_ex(and_out , a_in, b_in);
  input a_in, b_in;
  output and_out;
  wire #10 and_out;
  assign and_out = a_in & b_in;
endmodule
```

More “assign” Statements

Behavioral Modeling

Behavioral or algorithmic level:

- This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.

Behavioral modeling

- There are two structured procedures in Verilog:
 - **initial**
 - **always**
- Concurrent execution is observed in between these procedures.
- Sequential / concurrent execution can be realized within these procedures.
- Only registers can be assigned in these procedures.
- The assignments in these procedures are called “procedural assignments”.

initial statement

- Starts execution at ‘0’ simulation time and executes only once during the entire simulation.
- Multiple statements in initial block can be grouped with (**begin & end**) or (**fork & join**) keywords.
- These blocks are not synthesizable.

initial statement

- initial blocks cannot be nested.
- Each initial block represent a separate and independent activity.
- initial blocks are used in generating test benches.

initial block structures

initial

```
xor_out = in1 ^ in2;
```

initial
begin

```
enable = 1'b0;  
rst   = 1'b0;  
#100  rst = 1'b1;  
#20 enable = 1'b1;  
end
```

initial

begin

```
and_out = a_in & b_in;
```

end

initial

begin

```
clk = 1'b0;
```

```
reset = 1'b0;
```

initial

begin

```
#100 reset = 1'b1;
```

```
#20 clk = 1'b1;
```

end

end

always statement

- Starts execution at ‘0’ simulation time and is active all through out the entire simulation.
- Multiple statements inside always block can be grouped with (**begin & end**) or (**fork & join**) keywords.
- Execution of always blocks is controlled by using the timing control.
- always blocks cannot be nested.

always statement

- An always block without any sensitivity control will create an infinite loop and execute forever.
- Each always block represent a separate and independent activity.
- These blocks can synthesize to different hardware depending on their usage.
- always block with timing control are synthesizable.

always block structures

always

```
xor_out = in1 ^ in2;
```

always @(posedge reset)
begin
 if (reset == 1'b1)
 q_out = 1'b0;
 else
 q_out = d_in;
end

always @(a_in or b_in)
begin
 and_out = a_in & b_in;
end

always
begin
 cnt = 1'b0;
 reset = 1'b0;
 always @(posedge clk)
 begin
 #100 cnt = 1'b1;
 #20 enable = 1'b1;
 end
end

always block - Examples

always statement with timing control - Example

```
module cntr(cnt_out,  
             clk_in);  
input clk_in;  
output cnt_out;  
  
reg cnt_out;  
wire clk_in;  
  
always @(posedge clk_in)  
cnt_out = cnt_out + 1'b1;  
  
endmodule
```

always statement without timing control -Example

```
module clk_gen(clk_out);  
  
output clk_out;  
reg clk_out;  
  
always  
#5 clk_out = ~clk_out;  
initial  
clk_out = 1'b1;  
endmodule
```

Block statements

- Provides a means of grouping two or more procedural statements together.
- Types of blocks:
 - *sequential block / begin-end block*
 - *parallel block / fork-join block*
- These blocks can be nested.
- These blocks can be mixed.

Sequential block

Format:

```
begin  
  <statements>  
end
```

- Statements are executed sequentially one after the other.
- If delay is specified, it is relative to the time when the previous statement in the block is executed.
- Control leaves the block after executing the last statement.

Parallel block

Format:

fork

<statements>

join

- Statements are executed concurrently.
- If delay is specified, it is relative to the time, when the block has started its execution.
- Control leaves the block after executing the last time ordered statement.

Continuous Vs. procedural assignments

Continuous assignment	Procedural assignment
Occurs within a module.	Occurs inside an always or an initial statement.
Executes concurrently with other statements.	Execution is w.r.t. other statements surrounding it.
Drives nets.	Drives registers.
Uses “=” assignment operator.	Uses “=” or “<=” assignment operator.
Uses assign keyword.	No assign keyword (exception).

Procedural assignments

- These are for updating **reg**, **integer**, **real**, **time** and their bit / part selects.
- The values of the variables can get changed only by another procedural assignment statement.
- Procedural assignments are of two types
 - blocking procedural assignment
 - non-blocking procedural assignment

Blocking Assignment

- Most Commonly used type
- The target of assignment gets updated before the next sequential statement in procedural block is executed.
- A statement using blocking assignment blocks the execution of the statements following it, until it gets completed.
- Recommended style for modeling combinational logic.(data dependency)

Non-Blocking Assignment

- The assignments to the target gets scheduled for the end of the simulation cycle.
 - Normally occurs at the end of the sequential block (**begin** **end**)
 - Statements subsequent to the instruction under the consideration are not blocked by the assignment.
- Recommended style for modeling sequential logic
 - Can be used to assign several '**reg**' type variables synchronously, under the control of a common block

Rules to be followed

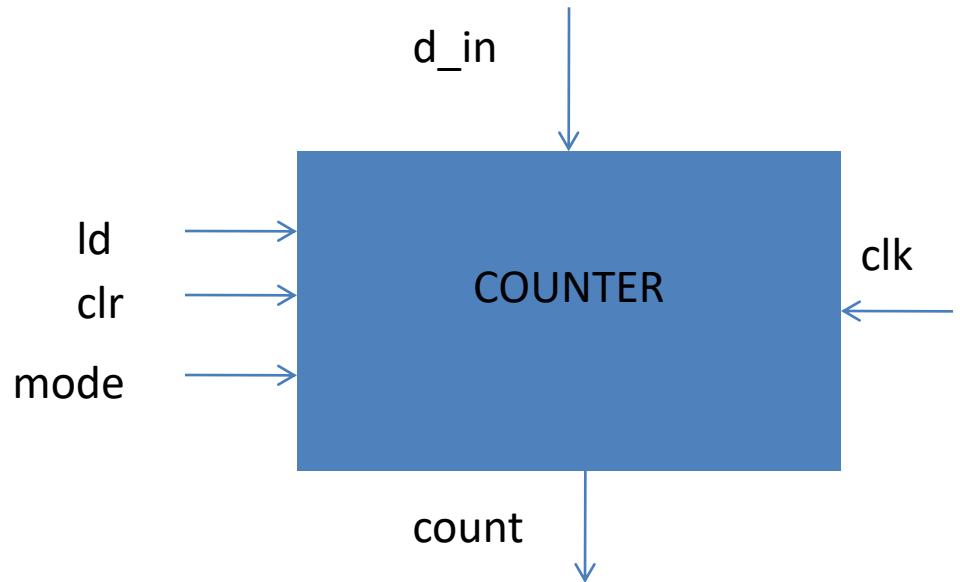
- Verilog synthesizer ignores the delays specified in a procedural assignment statement.
 - May lead to functional mismatch between the design model and the synthesized netlist.
- A variable cannot appear as the target of both a blocking and a non-blocking assignment.

Following is not permissible

```
value = value +1;  
value <=init
```

Example- up-down counter(synchronous clear)

```
module counter (clk, clr, ld, mode, d_in, count);
    input clk, mode, clr, ld;
    input [0:7] d_in;
    output [0:7] count;
    reg [0:7] count;
    always @(posedge clk)
        if (ld)
            count<=d_in;
        else if (clr)
            count<=1'b0;
        else if (mode)
            count <=count + 1;
        else
            count<=count-1
endmodule
```



Module:3

INTRODUCTION TO COMBINATIONAL CIRCUIT

Design of combinational circuits, Adder, Subtractor, Code Converter, Analyzing a Combinational Circuit.

Logic Circuits

Combinational Circuits

- Consist of *logic gates* whose outputs at any time are determined from the present combination of inputs
 - input variables, logic gates, and output variables
- The logic gates accept signals from the inputs and generate signals to the output

Sequential Circuits

- Consist of *memory storage elements* and *logic gates*
 - Their outputs are a function of the inputs and the state of the storage elements
 - The state of storage elements is a function of previous inputs
- The outputs of a sequential circuit also depend on the past inputs

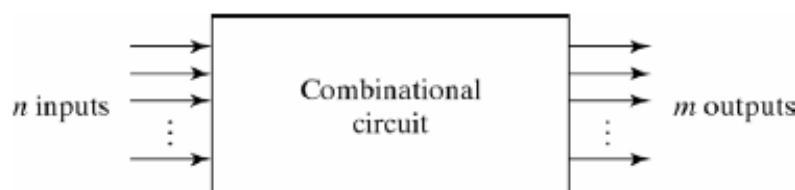


Fig. 4-1 Block Diagram of Combinational Circuit

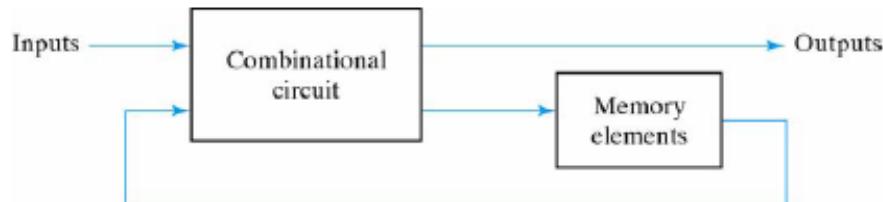


Fig. 5-1 Block Diagram of Sequential Circuit

Combinational Circuits

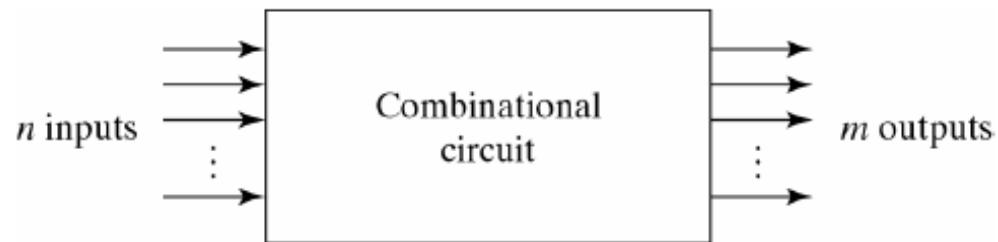


Fig. 4-1 Block Diagram of Combinational Circuit

Transform binary information from the given input data to a required output data

- n input variables
 - 2^n possible binary input combinations
 - $(2^n)^2 = 2^{2n}$ possible Boolean functions
- m output variables
 - each output function is expressed in terms of the n input variables
 - described by m Boolean functions

Standard combinational circuits

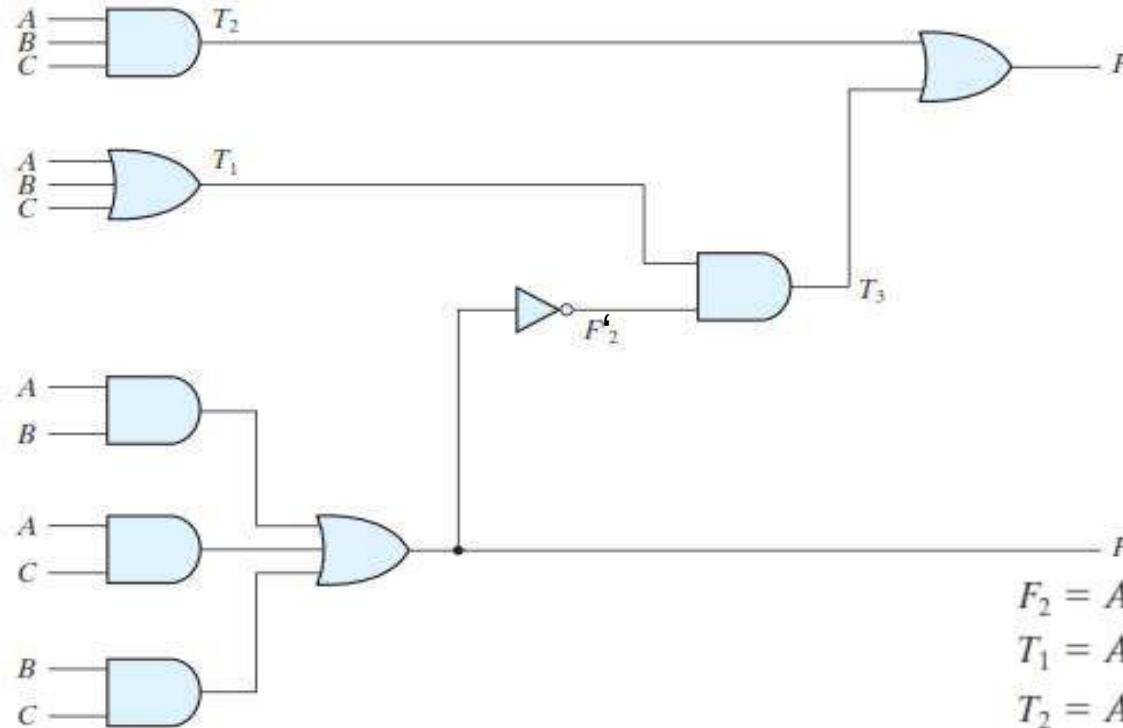
- available in MSI, standard cells in complex VLSI circuits
- i.e. adders, subtractors, comparators, decoders, encoders, multiplexers

Analysis Procedure

Start with a given logic diagram and culminate with a set of Boolean functions, a truth table, or a possible explanation of the circuit operation

- Make sure the given circuit is combinational and not sequential
 - no feedback paths or memory elements
 - A feedback path is a connection from the output of one gate to the input of a second gate that forms part of the input to the first gate
- Obtain the output Boolean functions or the truth table

Obtaining Boolean Functions



First Obtain functions of input variables

$$F_2 = AB + AC + BC$$

$$T_1 = A + B + C$$

$$T_2 = ABC$$

FIGURE 4.2
Logic diagram for analysis example

Next, we consider outputs of gates that are a function of already defined symbols:

$$T_3 = F'_2 T_1$$

$$F_1 = T_3 + T_2$$

To obtain F_1 as a function of A , B , and C , we form a series of substitutions as follows:

$$\begin{aligned} F_1 &= T_3 + T_2 = F'_2 T_1 + ABC = (AB + AC + BC)'(A + B + C) + ABC \\ &= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\ &= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\ &= A'BC' + A'B'C + AB'C' + ABC \end{aligned}$$

Design Procedure

The design of combinational circuits:

- Starts from the specification of the problem and culminates in a logic circuit diagram or a set of Boolean functions.
1. From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each.
 2. Derive the truth table that defines the required relationship between inputs and outputs.
 3. Obtain the simplified Boolean functions for each output as a function of the input variables.
 4. Draw the logic diagram and verify the correctness of the design (manually or by simulation).

Constraints in a practical design:

- number of gates
- number of inputs to a gate
- propagation time of the signal through the gates
- number of interconnections
- limitations of the driving capability of each gate
- etc.

Half Adder - Addition of Two Bits

Table 4.3

Half Adder

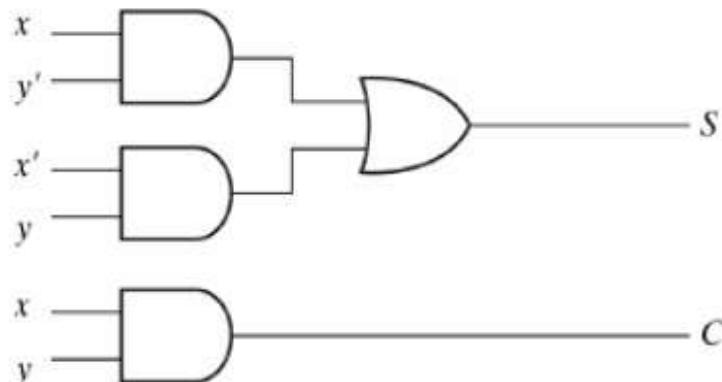
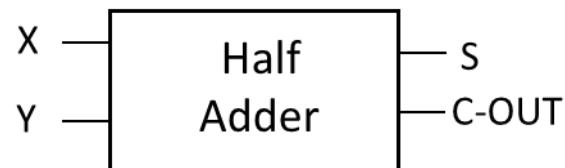
x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Two inputs: x and y

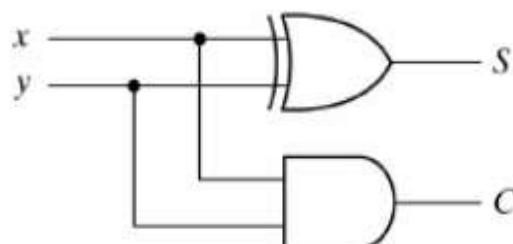
Two outputs:

- Sum S = $x'y + xy'$
- Carry C = xy

It can also be implemented with an exclusive-OR and an AND gate



$$(a) S = xy' + x'y \\ C = xy$$

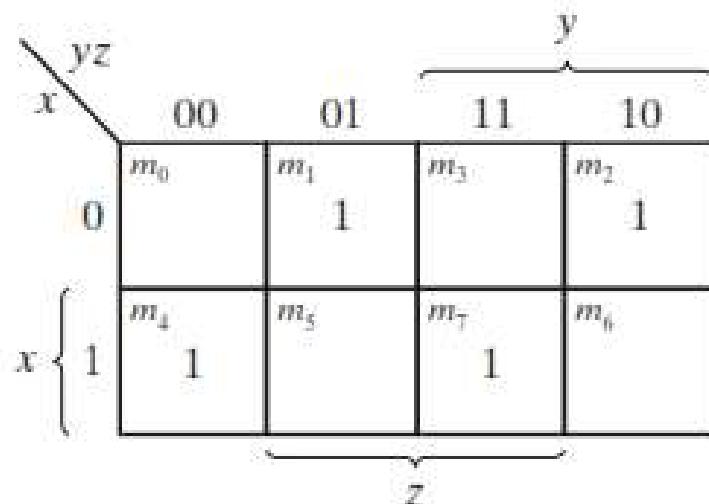
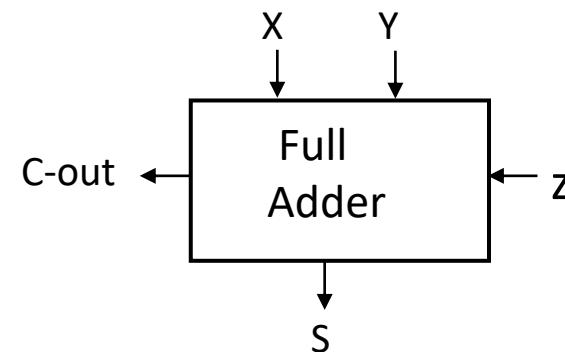


$$(b) S = x \oplus y \\ C = xy$$

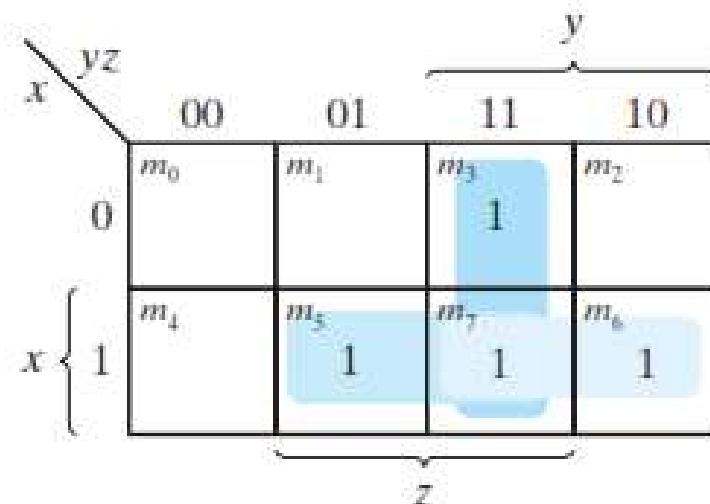
Table 4.4
Full Adder

Full Adder-Sum of Three Bits

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



$$(a) S = x'y'z + x'yz' + xy'z' + xyz$$



$$(b) C = xy + xz + yz$$

Full Adder

- Adding two single-bit binary values, X, Y with a carry input bit C-in produces a sum bit S and a carry out C-out bit.

Full Adder Truth Table

Inputs			Outputs	
X	Y	C-in	S	C-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S(X, Y, C\text{-in}) = \sum (1, 2, 4, 7)$$

$$\text{C-out}(x, y, C\text{-in}) = \sum (3, 5, 6, 7)$$

Sum S

A Karnaugh map for the sum bit S. The columns are labeled XY (00, 01, 11, 10) and the rows are labeled C-in (0, 1). The minterms 1, 3, 5, and 7 are marked with 1's. The map is annotated with X and Y at the top right, and C-in at the bottom right.

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

$$S = X'Y'(C\text{-in}) + XY'(C\text{-in})' + XY'(C\text{-in})' + XY(C\text{-in})$$

$$S = X \oplus Y \oplus (C\text{-in})$$

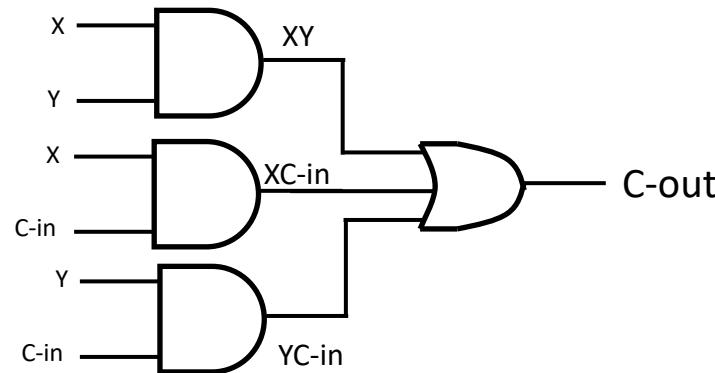
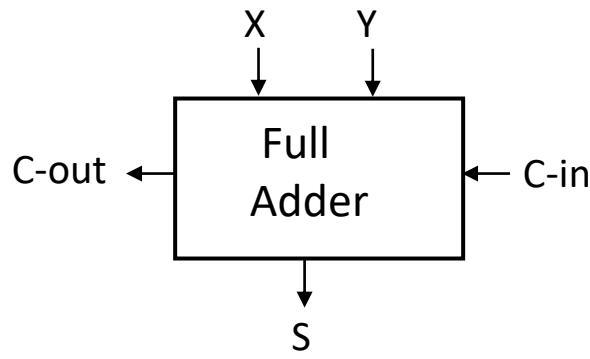
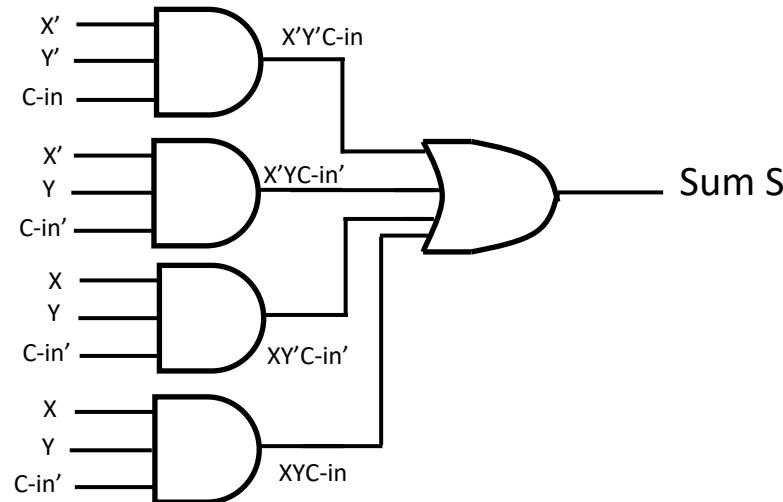
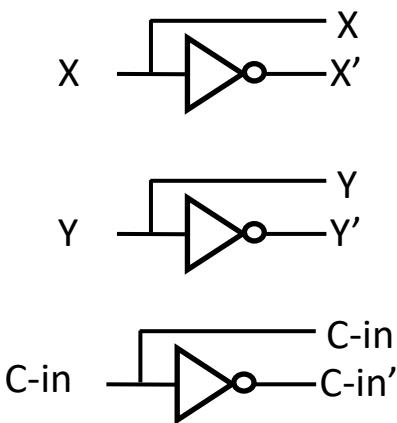
Carry C-out

A Karnaugh map for the carry bit C-out. The columns are labeled XY (00, 01, 11, 10) and the rows are labeled C-in (0, 1). The minterms 3, 5, 6, and 7 are marked with 1's. The map is annotated with X and Y at the top right, and C-in at the bottom right.

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

$$\text{C-out} = XY + X(C\text{-in}) + Y(C\text{-in})$$

Full Adder Circuit Using AND-OR

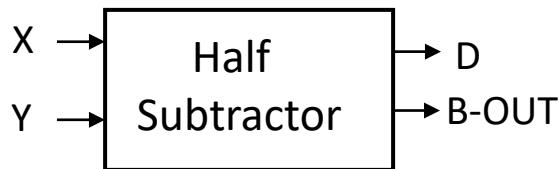


Half Subtractor

- Subtracting a single-bit binary value Y from another X (I.e. $X - Y$) produces a difference bit D and a borrow out bit B-out.
- This operation is called half subtraction and the circuit to realize it is called a half subtractor.

Half Subtractor Truth Table

Inputs		Outputs	
X	Y	D	B-out
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0



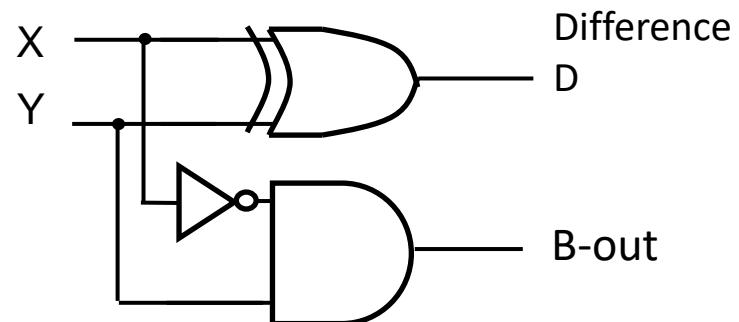
$$D(X,Y) = \Sigma (1,2)$$

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

$$D = X \oplus Y$$

$$B\text{-out}(x, y, C\text{-in}) = \Sigma (1)$$

$$B\text{-out} = X'Y$$



Full Subtractor

- Subtracting two single-bit binary values, Y, B-in from a single-bit value X produces a difference bit D and a borrow out B-out bit. This is called full subtraction.

Full Subtractor Truth Table

Inputs			Outputs	
X	Y	B-in	D	B-out
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$$D(X, Y, B\text{-in}) = \sum (1, 2, 4, 7)$$

$$B\text{-out}(x, y, B\text{-in}) = \sum (1, 2, 3, 7)$$

Difference D

		X			
	XY	00	01	11	10
B-in	0	0	2	6	4
1	1	1	3	7	5

$$D = X'Y'(B\text{-in}) + XY'(B\text{-in})' + XY'(B\text{-in})' + XY(B\text{-in})$$

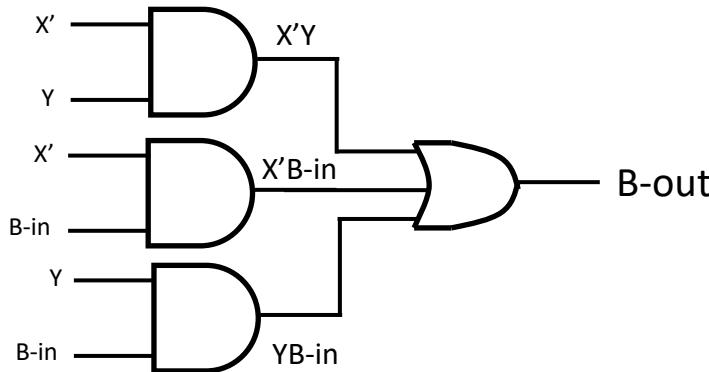
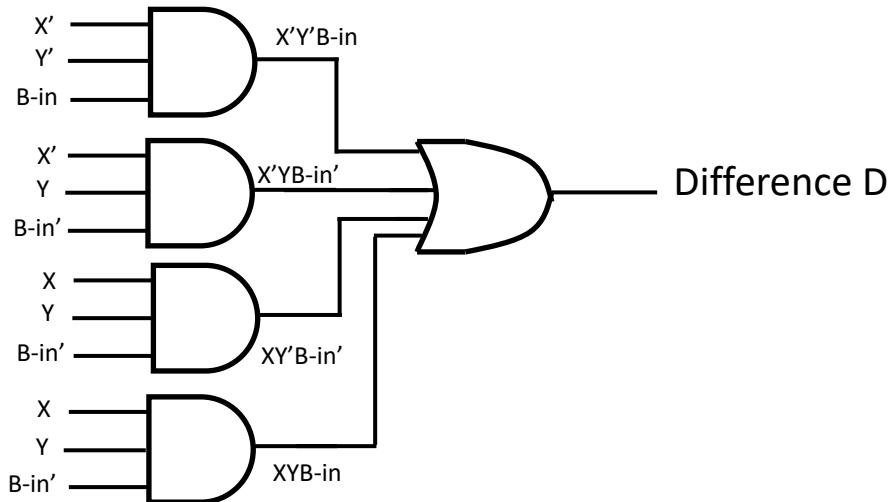
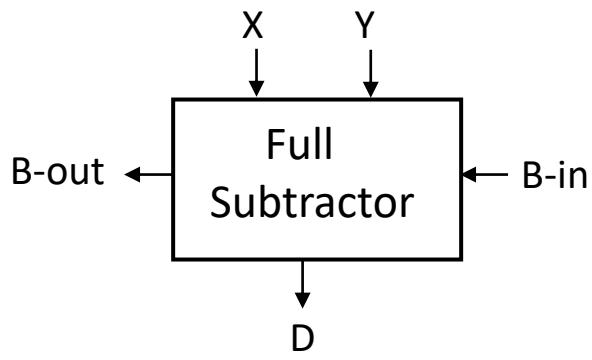
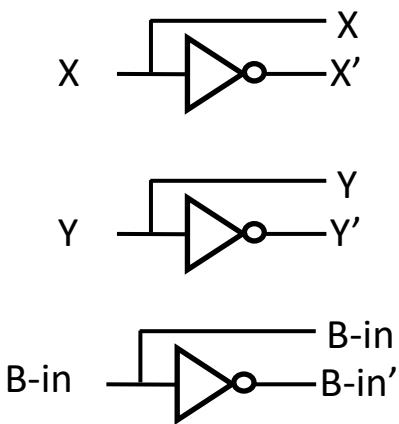
$$D = X \oplus Y \oplus (B\text{-in})$$

Borrow B-out

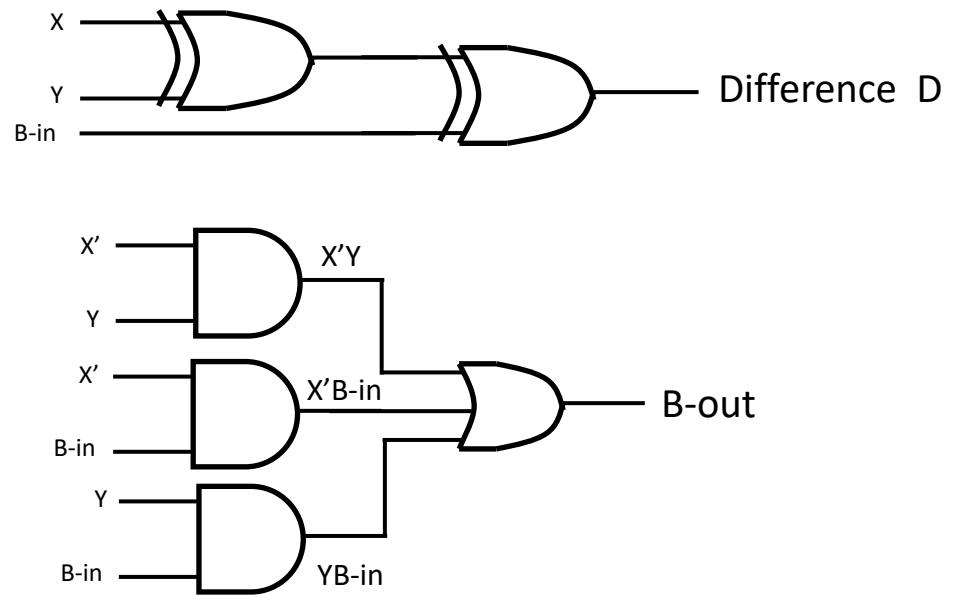
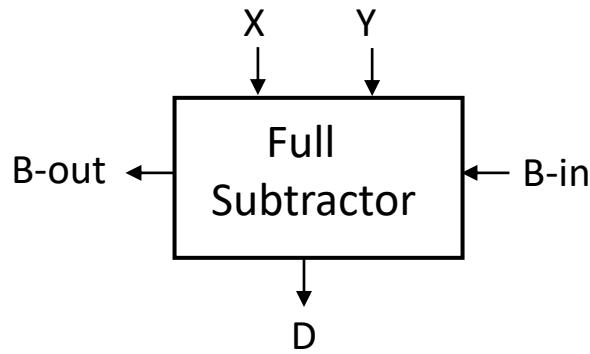
		X			
	XY	00	01	11	10
B-in	0	0	1	6	4
1	1	1	3	7	5

$$B\text{-out} = X'Y + X'(B\text{-in}) + Y(B\text{-in})$$

Full Subtractor Circuit Using AND-OR



Full Subtractor Circuit Using XOR



full subtractor

x	y	z/Bin	D	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$$\text{Diff} = \bar{x}\bar{y}z + \bar{x}y\bar{z} + x\bar{y}z + x\bar{y}\bar{z}$$

$$= \bar{x}(\bar{y}z + \bar{y}\bar{z}) + x(\bar{y}\bar{z} + yz)$$

$$= \bar{x}(\frac{y \oplus z}{K}) + x(\frac{\bar{y} \oplus z}{K})$$

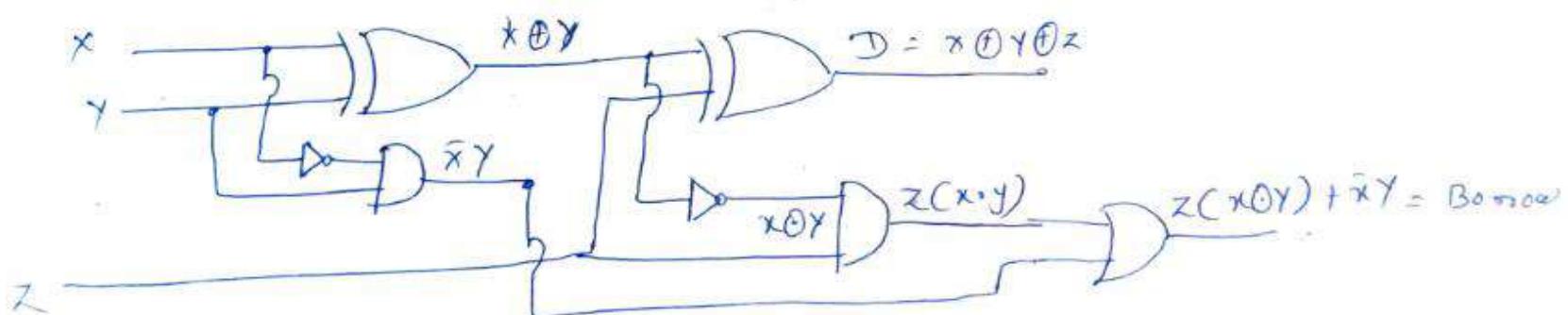
$$= \bar{x}K + x\bar{K}$$

$$= x \oplus y \oplus z$$

$$\text{Borrow}_{\text{out}} = \bar{x}\bar{y}z + \bar{x}y\bar{z} + \bar{x}yz + xy\bar{z}$$

$$= z(\bar{x}\bar{y} + xy) + \bar{x}y(\bar{z} + z)$$

$$= z(x \odot y) + \bar{x}y$$



Obtaining Boolean Functions and Truth Table from a Logic Diagram

Obtain output Boolean Functions from a logic diagram :

1. Label all gate outputs that are a function of input variables with arbitrary symbols—but with meaningful names. 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.

Obtain the Truth Table directly from a logic diagram :

1. Determine the number of input variables in the circuit. For n inputs, form the 2^n possible input combinations and list the binary numbers from 0 to $(2^n - 1)$ in a table.
2. Label the outputs of selected gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates which are a function of the input variables only.
4. Proceed to obtain the truth table for the outputs of those gates which are a function of previously defined values until the columns for all outputs are determined.

Problem

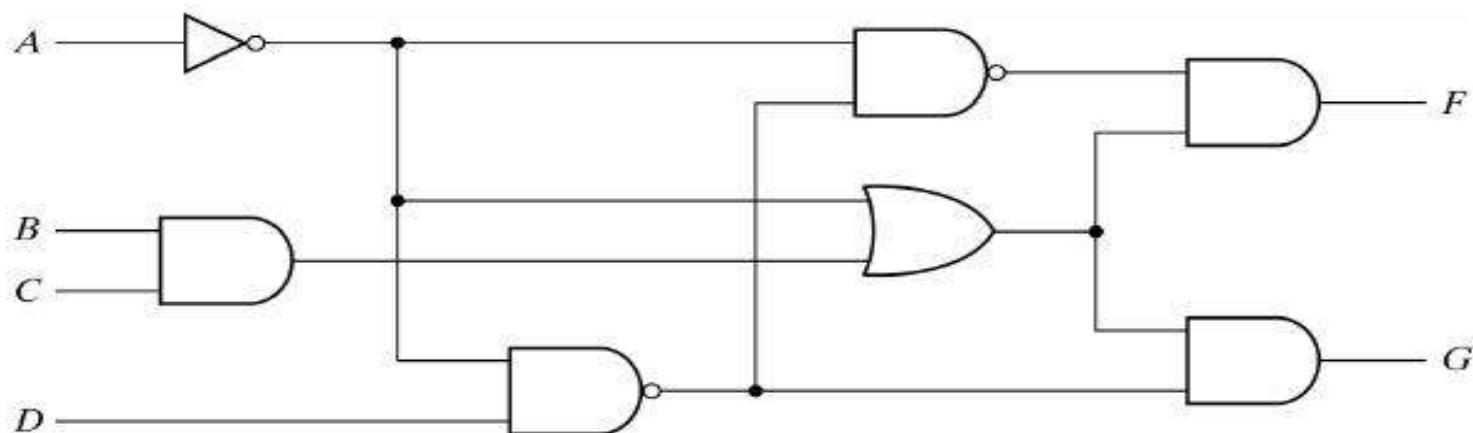


Fig. P4-2

$$F = (A + D)(A' + BC) = A'D + ABC + BCD = A'D + ABC$$

$$G = (A + D')(A' + BC) = A'D' + ABC + BCD' = A'D' + ABC$$

AB	CD	00	01	11	10
00		1	1		
01		1	1		
11			1	1	
10					1

AB	CD	00	01	11	10
00		1			1
01		1			1
11			1	1	
10					1

$$A'D + ABC + BCD = A'D + ABC$$

$$A'D' + ABC + BCD' = A'D' + ABC$$

Problem Majority Circuit

A majority circuit is a combinational circuit whose output is equal to 1 if the input variables have more 1's than 0's. The output is 0 otherwise. Design a 3-input majority circuit.

xy ₃	F
000	0
001	0
010	0
011	1
100	0
101	1
110	1
111	1

x	y ₃	01	11	10
0			1	
,		1	1	1

$$F = xy + xz + yz$$

Addition of Two 2-digit's

$$\bullet A_1 A_0 + B_1 B_0 = CS_1 S_0$$

$$\begin{array}{r} 00 \\ + 00 \\ \hline 000 \end{array}$$

$$\begin{array}{r} 11 \\ + 11 \\ \hline 110 \end{array}$$

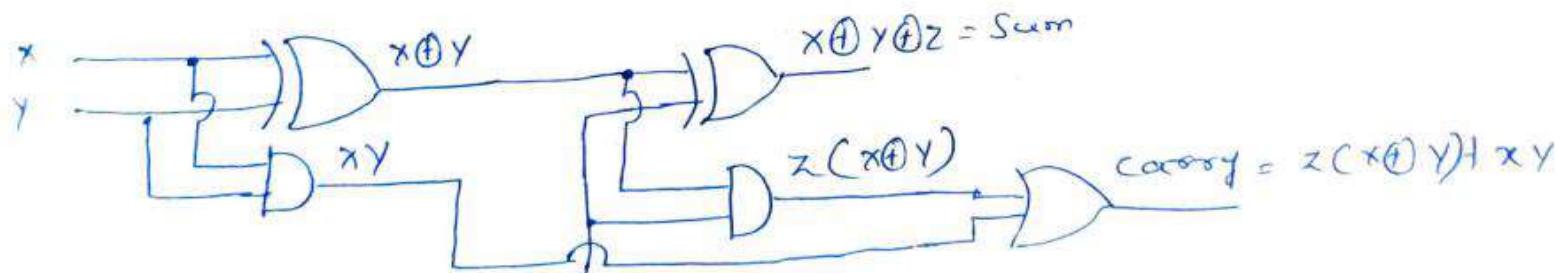
Full Adder

x	y	z/cin	s	c
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\begin{aligned}
 S &= \bar{x}\bar{y}z + \bar{x}y\bar{z} + x\bar{y}\bar{z} + xy\bar{z} \\
 &= \bar{x}(\bar{y}z + y\bar{z}) + x(\bar{y}\bar{z} + y\bar{z}) \\
 &= \cancel{x}(\cancel{y\oplus z}) + x(\cancel{y\oplus z}) \\
 &= \cancel{x}K + x(\cancel{K}) \\
 &= x \oplus K = x \oplus y \oplus z
 \end{aligned}$$

$$\begin{aligned}
 C &= \cancel{\bar{x}\bar{y}z} + \cancel{x\bar{y}z} + \cancel{xy\bar{z}} + \cancel{xyz} \\
 &= z(\bar{x}y + x\bar{y}) + xy(\bar{z} + z) = z(x \oplus y) + xy
 \end{aligned}$$

Implementation of full adder by two Half adder and an OR gate



Implementations of Full-Adder

Two-level AND-OR Implementation

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

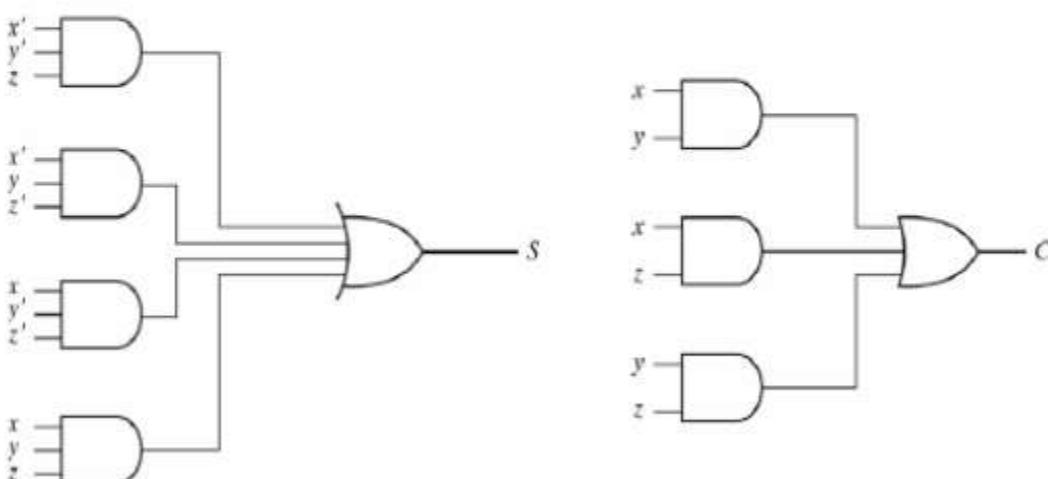


Fig. 4-7 Implementation of Full Adder in Sum of Products

Implemented with two half adders and one OR gate

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

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

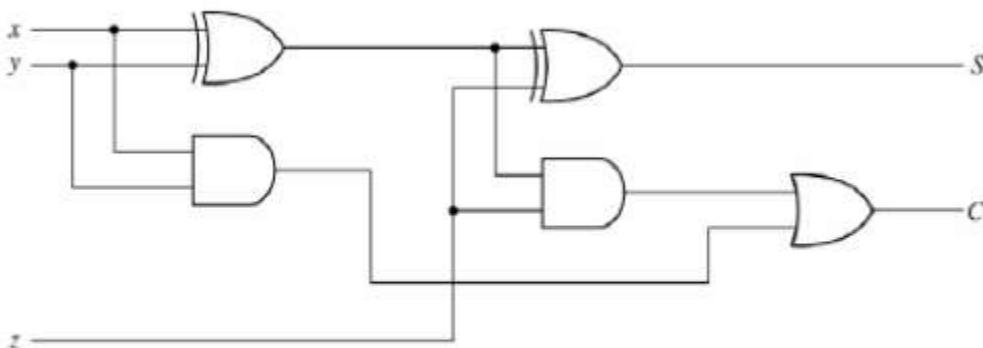


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

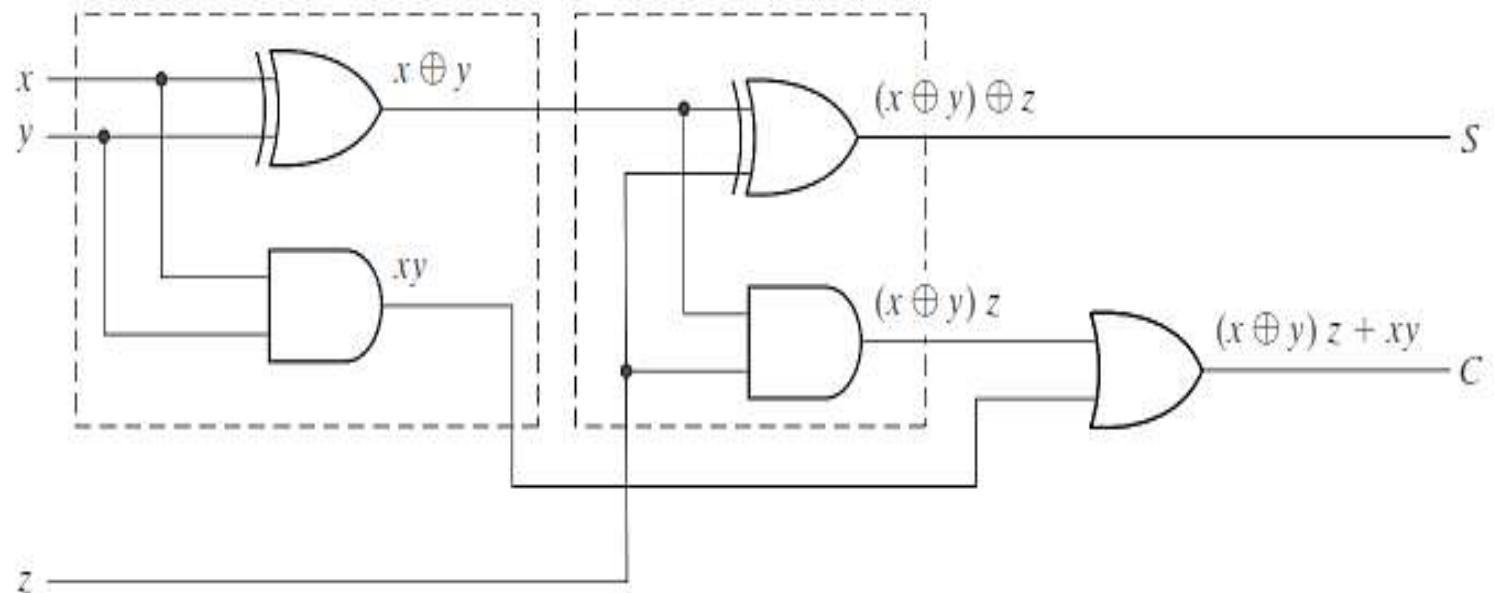


FIGURE 4.8

Implementation of full adder with two half adders and an OR gate

Binary adder

- Binary adder that produces the arithmetic sum of binary numbers can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain
- Note that the input carry C_0 in the least significant position must be 0.

Binary Adder

- For example to add $A = 1011$ and $B = 0011$

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}

Binary Adder

Example: $10+6=16$

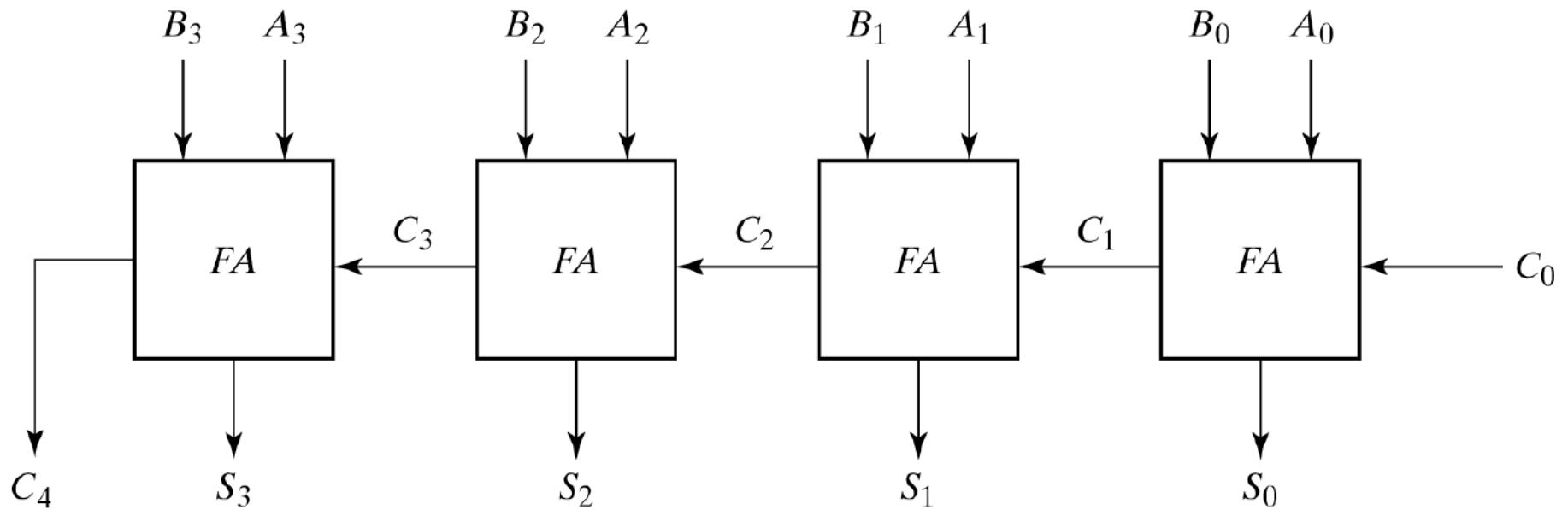


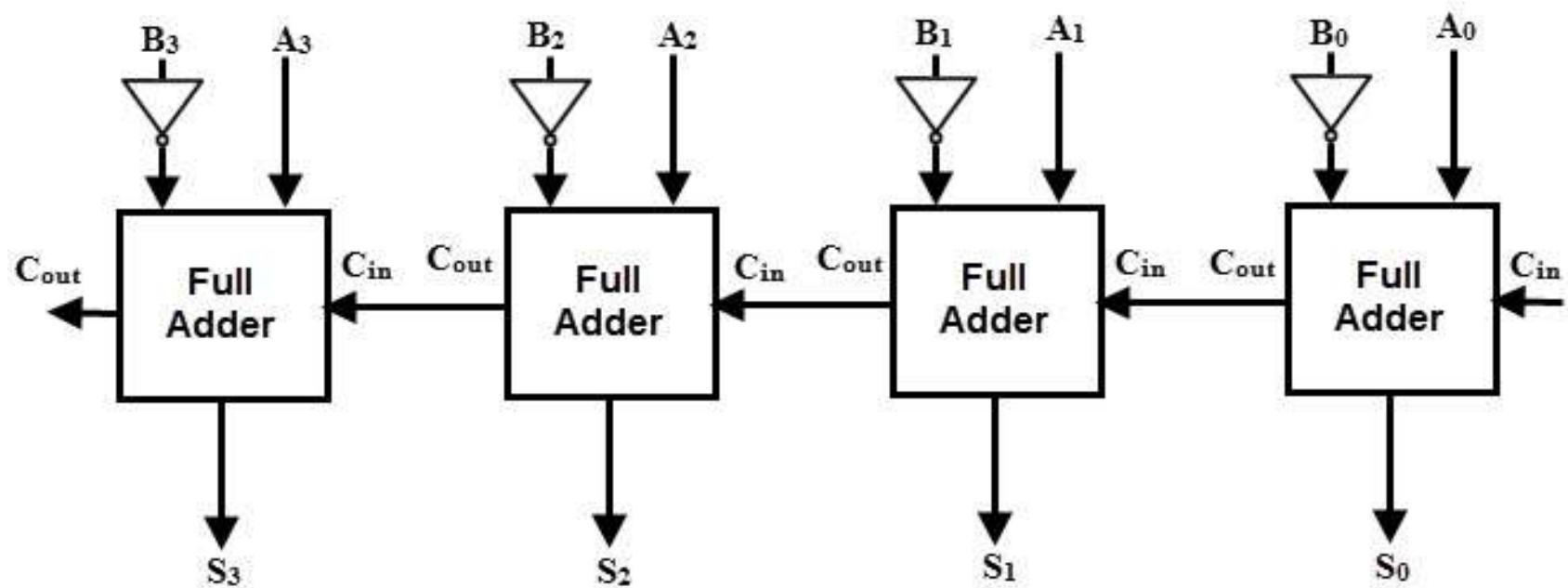
Fig. 4-9 4-Bit Adder

Binary Subtractor

- The subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A because $A - B = A + (-B) = A + ((B' + 1))$
- It means if we use the inverters to make 1's complement of B (connecting each B_i to an inverter) and then add 1 to the least significant bit (by setting carry C_0 to 1) of binary adder, then we can make a binary subtractor.

4 bit 2's complement Subtractor

Example: $10-6=10-((1\text{'s of } 6)+1)=4$



Adder Subtractor

- The addition and subtraction can be combined into one circuit with one common binary adder (see next slide).
- The mode M controls the operation. When M=0 the circuit is an **adder** when M=1 the circuit is a **subtractor**. It can be done by using exclusive-OR for each Bi and M. Note that $1 \oplus x = x'1 + x1' = x'$ and $0 \oplus x = x'0 + x0' = x$

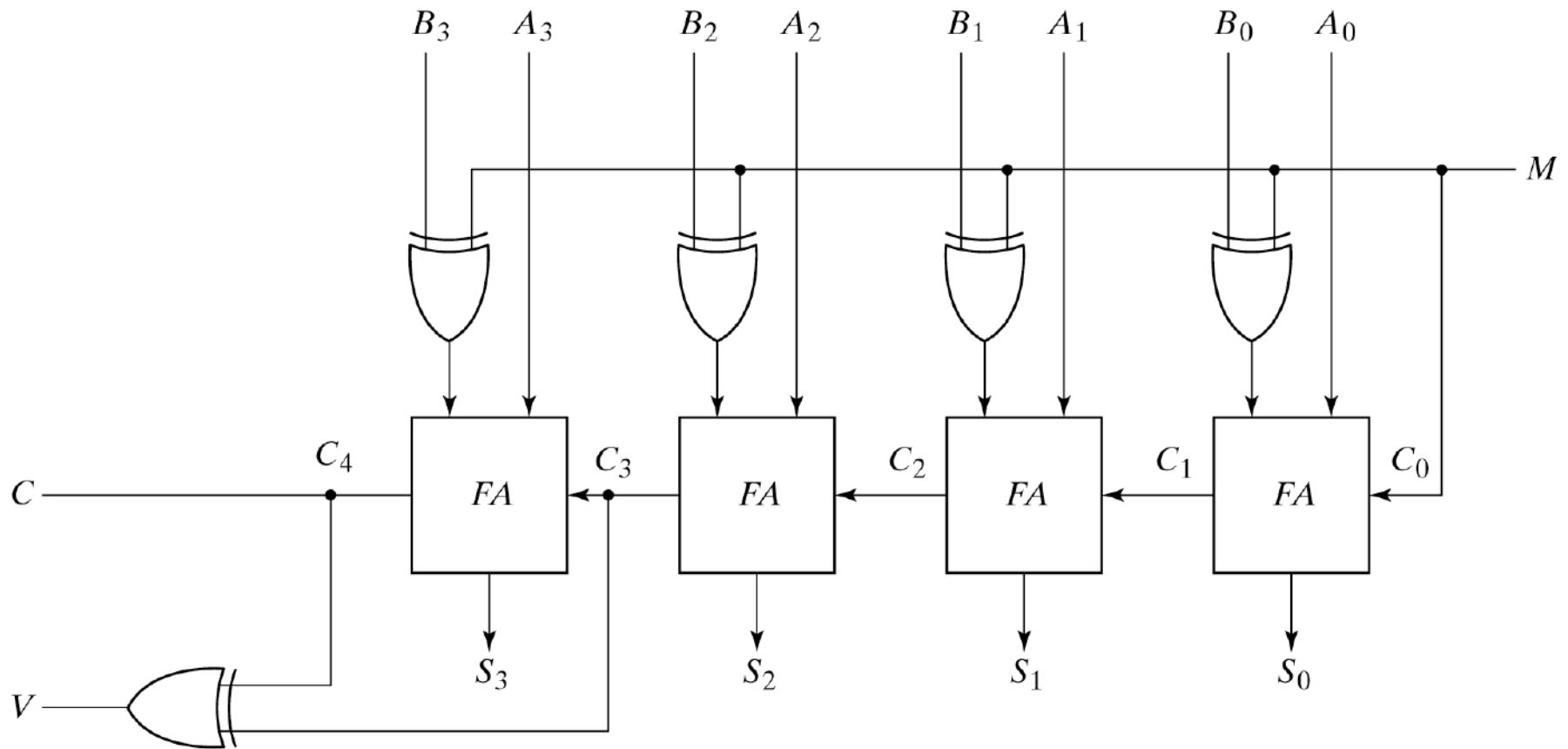


Fig. 4-13 4-Bit Adder Subtractor

Checking Overflow

- Note that in the previous slide if the numbers considered to be signed V detects overflow. V=0 means no overflow and V=1 means the result is wrong because of overflow
- Overflow can be happened when adding two numbers of the same sign (both negative or positive) and result can not be shown with the available bits. It can be detected by observing the carry into sign bit and carry out of sign bit position. If these two carries are not equal an overflow occurred. That is why these two carries are applied to exclusive-OR gate to generate V.

Example: Find the squares of 3-bit numbers.

Decimal Equivalent	Input variables			Decimal Equivalent	Output variables					
	X	Y	Z		A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	0	0	1
2	0	1	0	4	0	0	0	1	0	0
3	0	1	1	9	0	0	1	0	0	1
4	1	0	0	16	0	1	0	0	0	0
5	1	0	1	25	0	1	1	0	0	1
6	1	1	0	36	1	0	0	1	0	0
7	1	1	1	49	1	1	0	0	0	1

Module:3

Design And Analyses Of Combinational Circuits

Binary Parallel Adder, Magnitude Comparator,
Decoders, Encoders, Multiplexers, De-multiplexers

Binary adder

- Binary adder that produces the arithmetic sum of binary numbers can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain
- Note that the input carry C_0 in the least significant position must be 0.

Binary Adder

- For example to add $A = 1011$ and $B = 0011$

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}

Binary Adder

Example: $10+6=16$

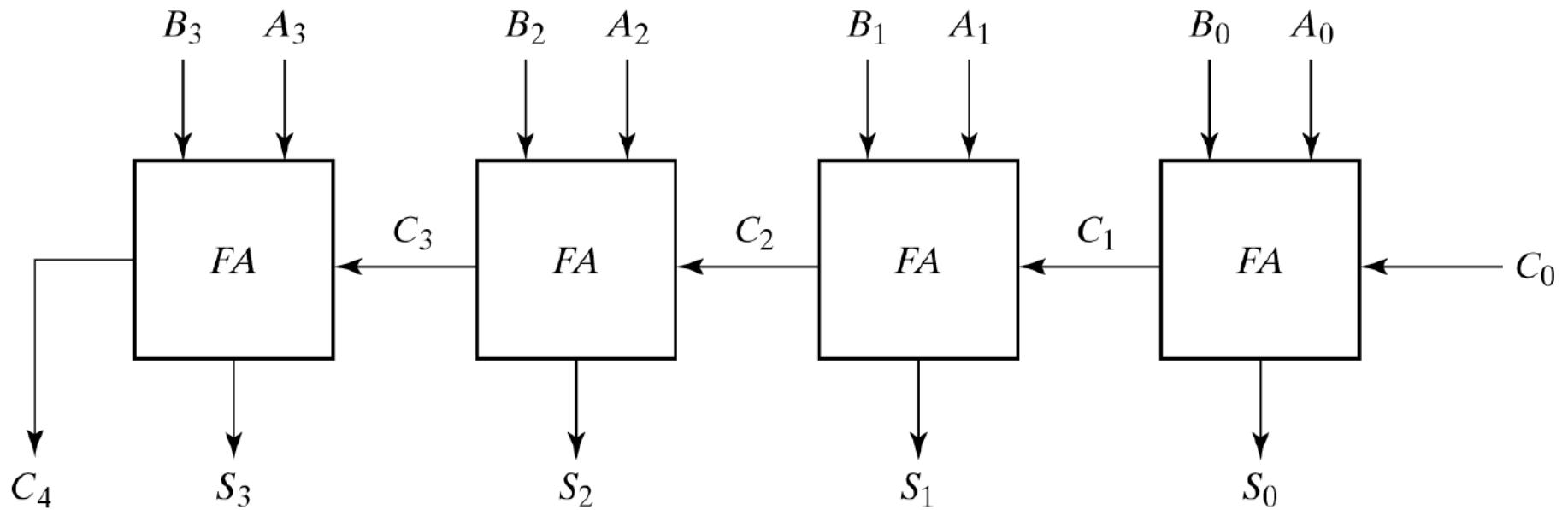


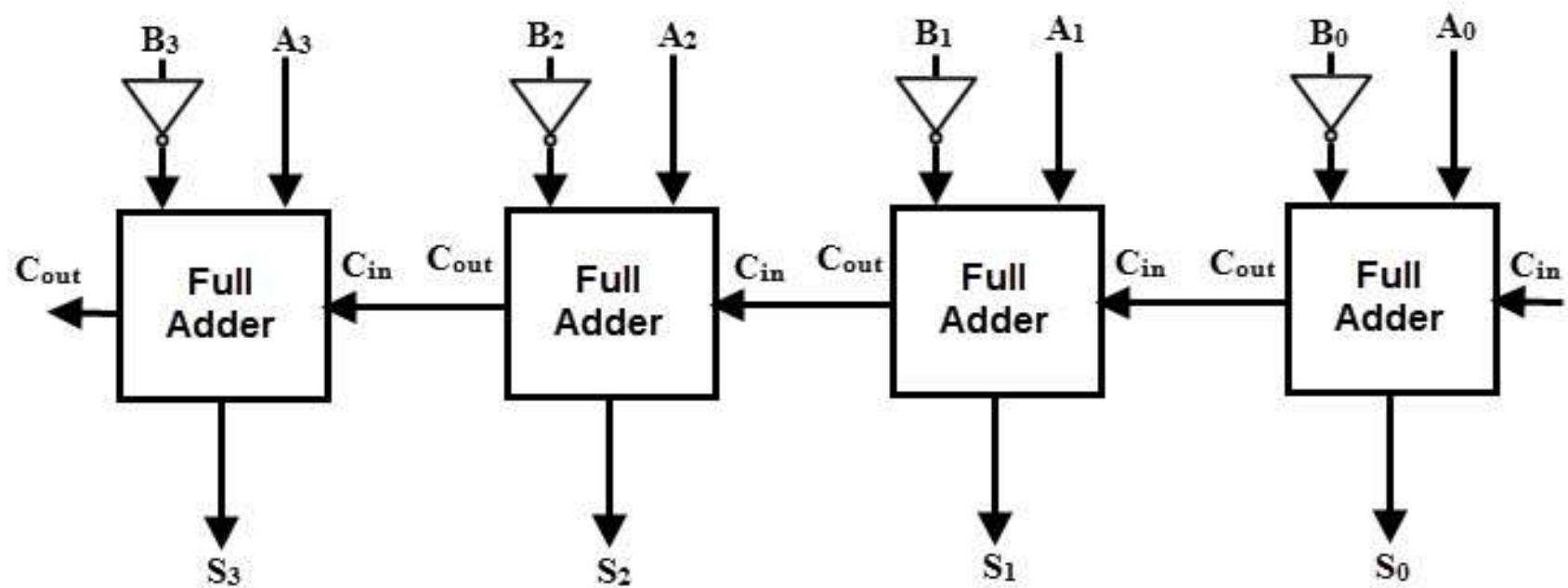
Fig. 4-9 4-Bit Adder

Binary Subtractor

- The subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A because $A - B = A + (-B) = A + ((B' + 1))$
- It means if we use the inverters to make 1's complement of B (connecting each B_i to an inverter) and then add 1 to the least significant bit (by setting carry C_0 to 1) of binary adder, then we can make a binary subtractor.

4 bit 2's complement Subtractor

Example: $10-6=10-((1\text{'s of } 6)+1)=4$



Adder Subtractor

- The addition and subtraction can be combined into one circuit with one common binary adder (see next slide).
- The mode M controls the operation. When M=0 the circuit is an **adder** when M=1 the circuit is a **subtractor**. It can be done by using exclusive-OR for each Bi and M. Note that $1 \oplus x = x'1 + x1' = x'$ and $0 \oplus x = x'0 + x0' = x$

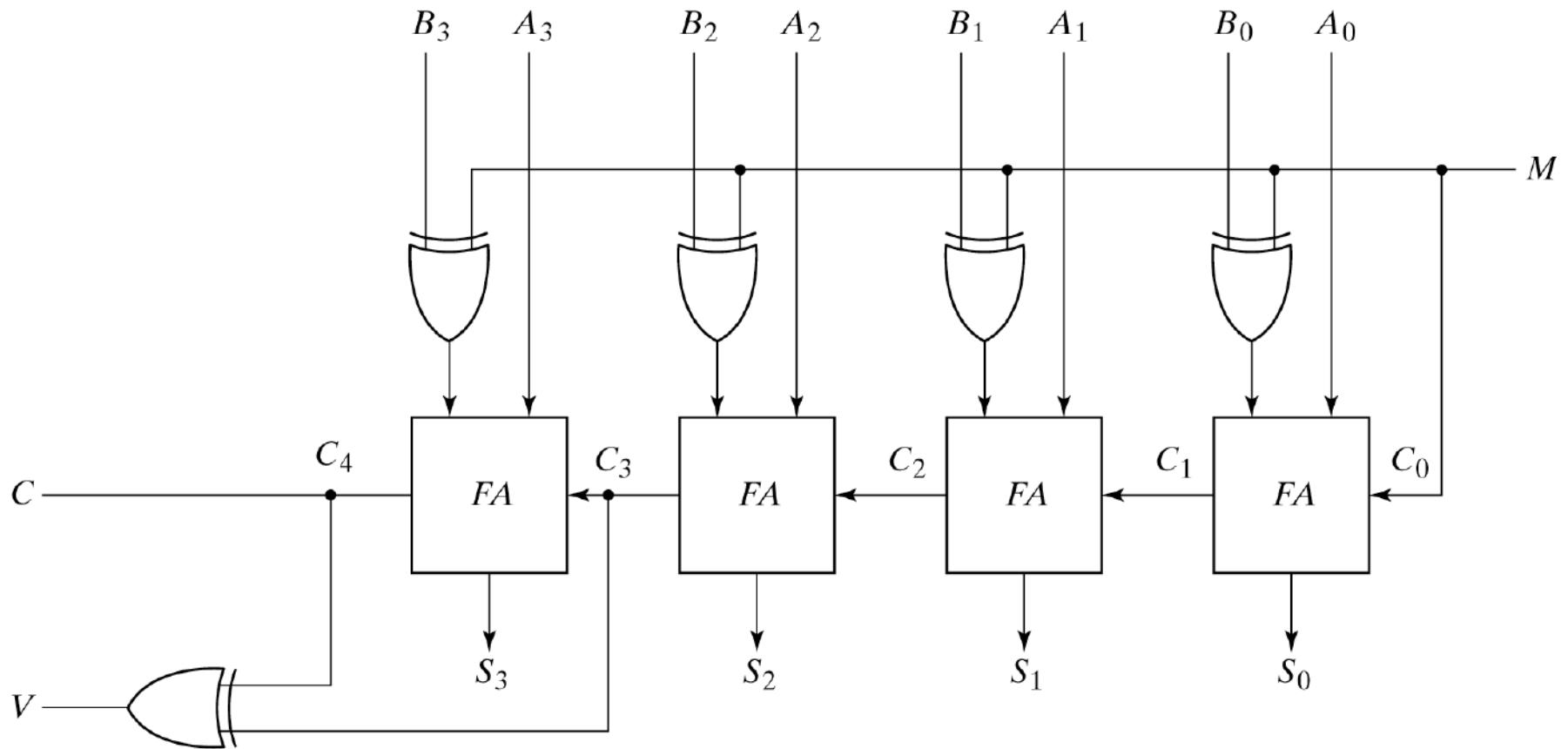


Fig. 4-13 4-Bit Adder Subtractor

Checking Overflow

- Note that in the previous slide if the numbers considered to be signed V detects overflow. V=0 means no overflow and V=1 means the result is wrong because of overflow
- Overflow can be happened when adding two numbers of the same sign (both negative or positive) and result can not be shown with the available bits. It can be detected by observing the carry into sign bit and carry out of sign bit position. If these two carries are not equal an overflow occurred. That is why these two carries are applied to exclusive-OR gate to generate V.

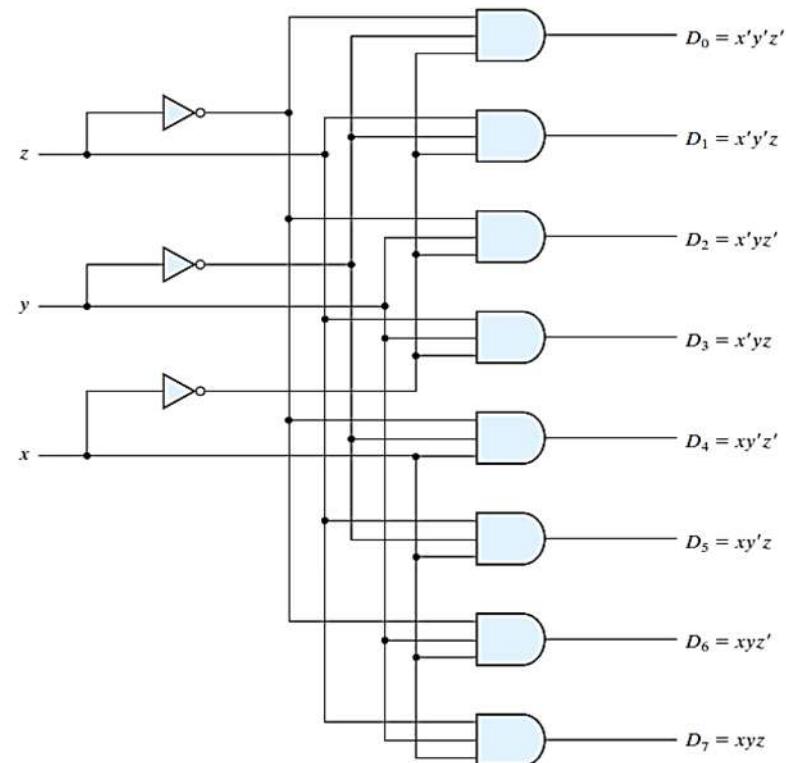
Decoders

A combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines

n-to-m-line decoders: generate m ($=2^n$ or fewer) minterms of n input variables

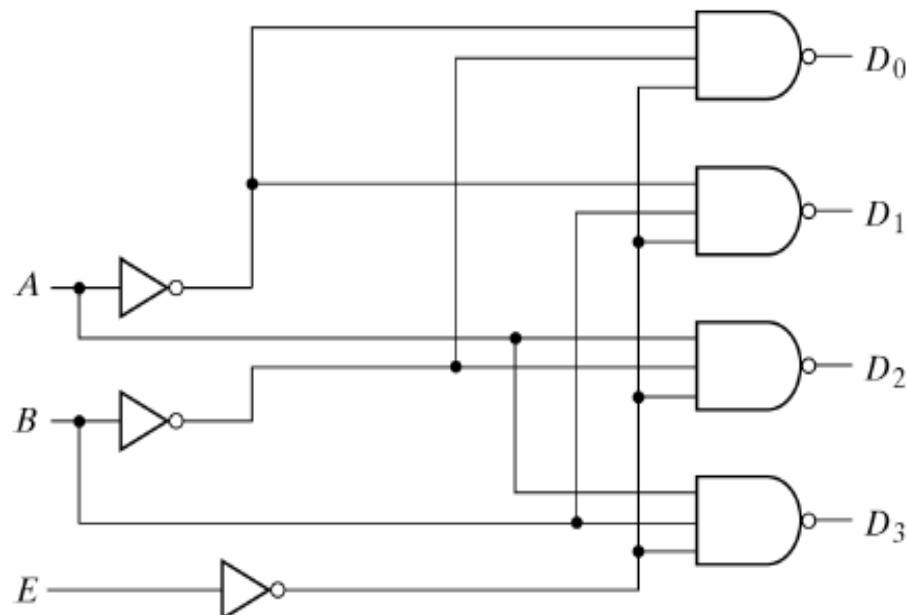
Truth Table of a Three-to-Eight-Line Decoder

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 using NAND

- More economical to generate the decoder minterms in their complemented form using NAND gates
- Support one or more enable inputs to control the circuit operation
 - e.g. disabled when E is equal to 1 (no outputs)
 - same as a **Demultiplexer**: a circuit that receives information from a single line and directs it to one of 2^n possible output lines
 - Selection of a specific output is controlled by bit combination of n selection lines

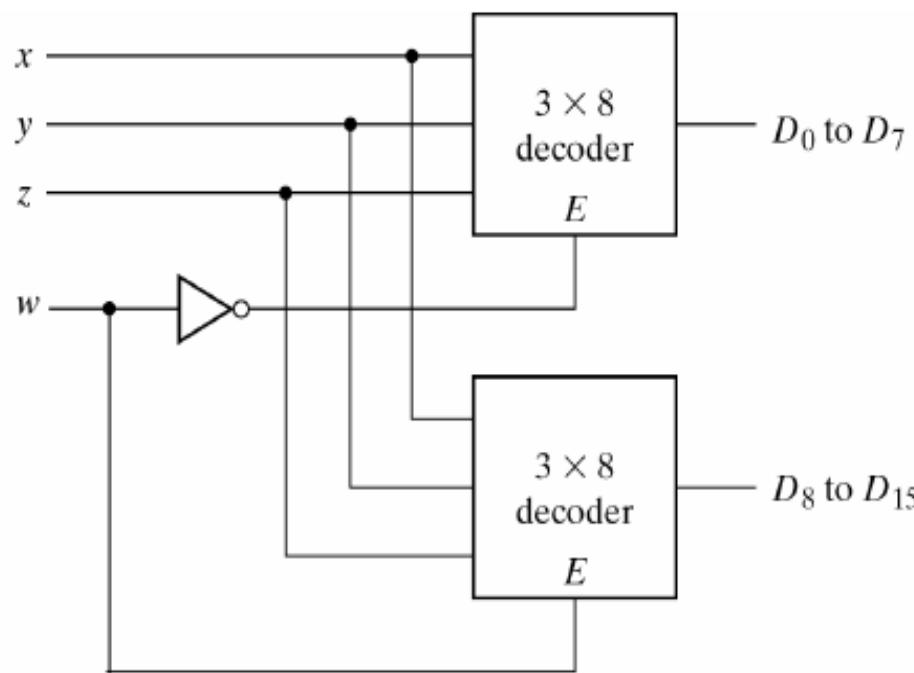


(a) Logic diagram

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

Connecting multiple decoders to form a larger one



wxyz	D ₀ D ₁ ***** D ₇	D ₈ ***** D ₁₅
0000	0111 1111	1111 1111
0001	1011 1111	1111 1111
0010	1101 1111	1111 1111
0011	1110 1111	1111 1111
0100	1111 0111	1111 1111
0101	1111 1011	1111 1111
0110	1111 1101	1111 1111
0111	1111 1110	1111 1111
1000	1111 1111	0111 1111
1001	1111 1111	1011 1111
1010	1111 1111	1101 1111
1011	1111 1111	1110 1111
1100	1111 1111	1111 0111
1101	1111 1111	1111 1011
1110	1111 1111	1111 1101
1111	1111 1111	1111 1110

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

w=0: the top decoder is enabled (0000 ... 0111: D₀ to D₇)

w=1: the bottom decoder is enabled (1000 ... 1111: D₈ to D₁₅)

- A decoder provides the 2^n minterm of n input variable
- Any combinational circuit with n inputs and m outputs can be implemented with an n -to- 2^n -line decoder and m OR gates

Combinational Logic Implementation – Full Adder with a Decoder

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

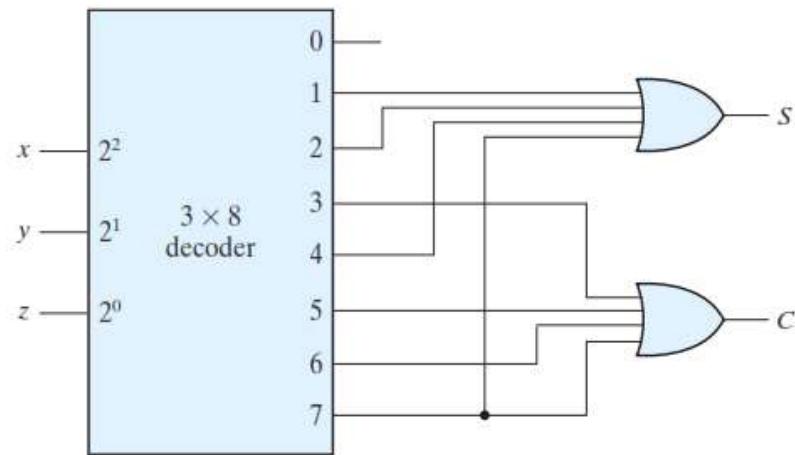


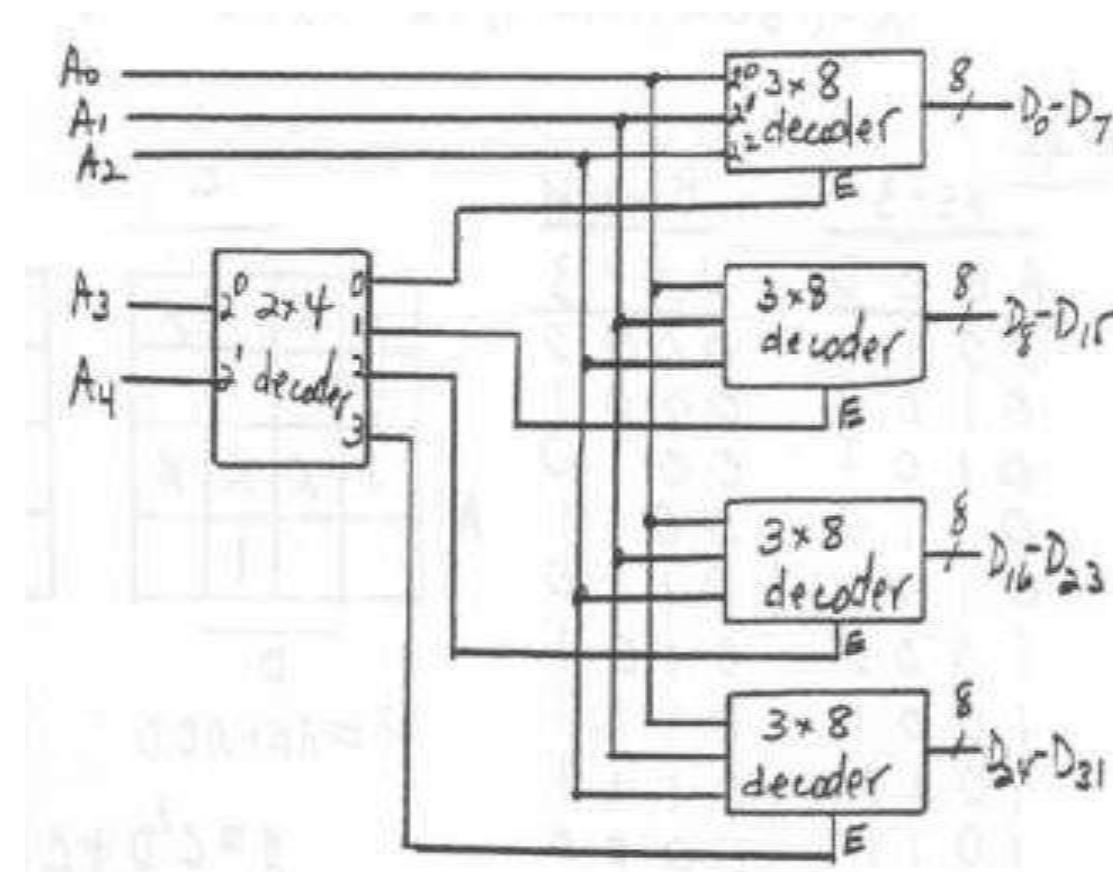
FIGURE 4.21
Implementation of a full adder with a decoder

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

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

Large input with NOR: if the number of minterms is greater than $2^n/2$, then F' can be expressed with fewer minterms

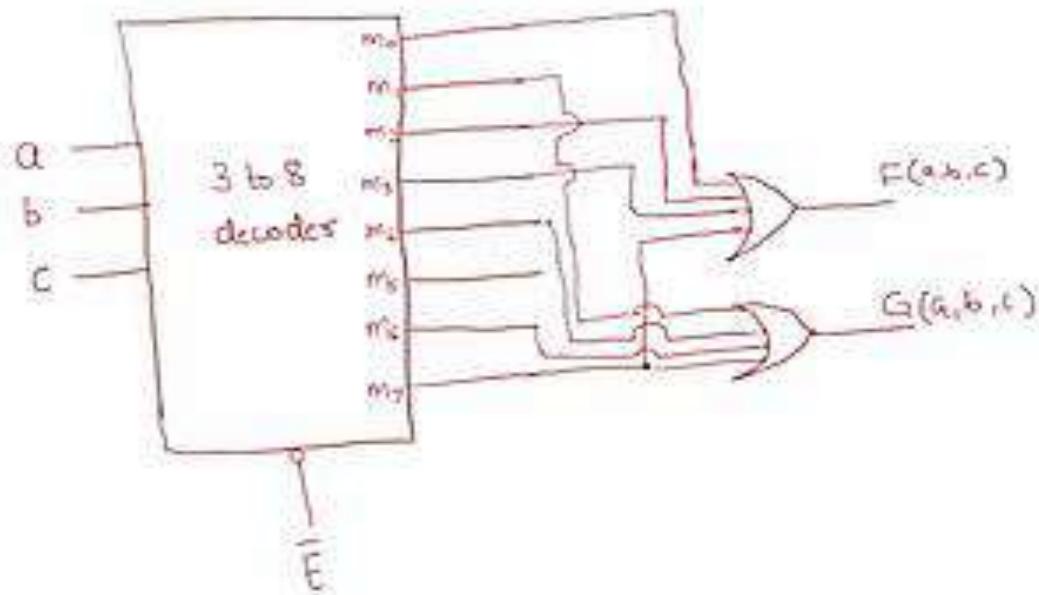
Problem : Construct a 5-to-32-line decoder with four 3-to-8-line decoders with enable and a 2-to-4-line decoder. Use block diagrams for the components



Example:

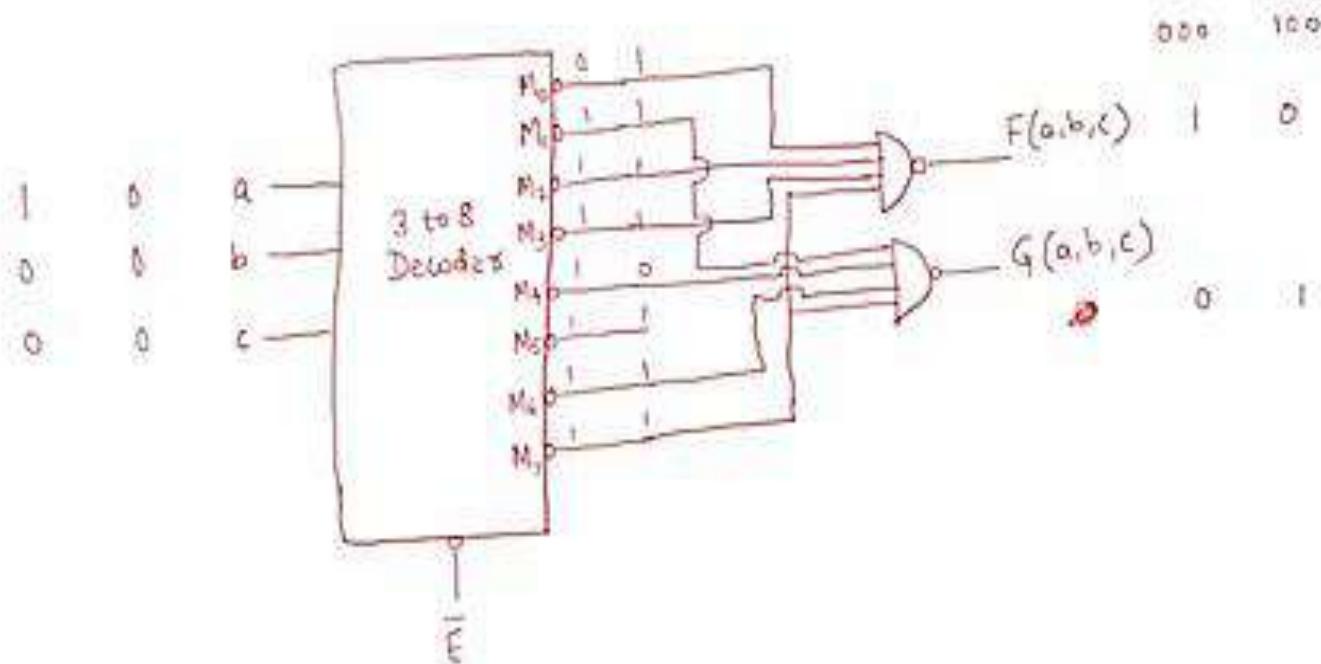
Case 1: Active HIGH output

- $F(a,b,c) = \sum m(0,2,3,7)$ and $G(a,b,c) = \sum m(1,4,6,7)$



Case 2: Active LOW output

- $F(a,b,c) = \sum m(0,2,3,7)$ and $G(a,b,c) = \sum m(1,4,6,7)$



Encoders

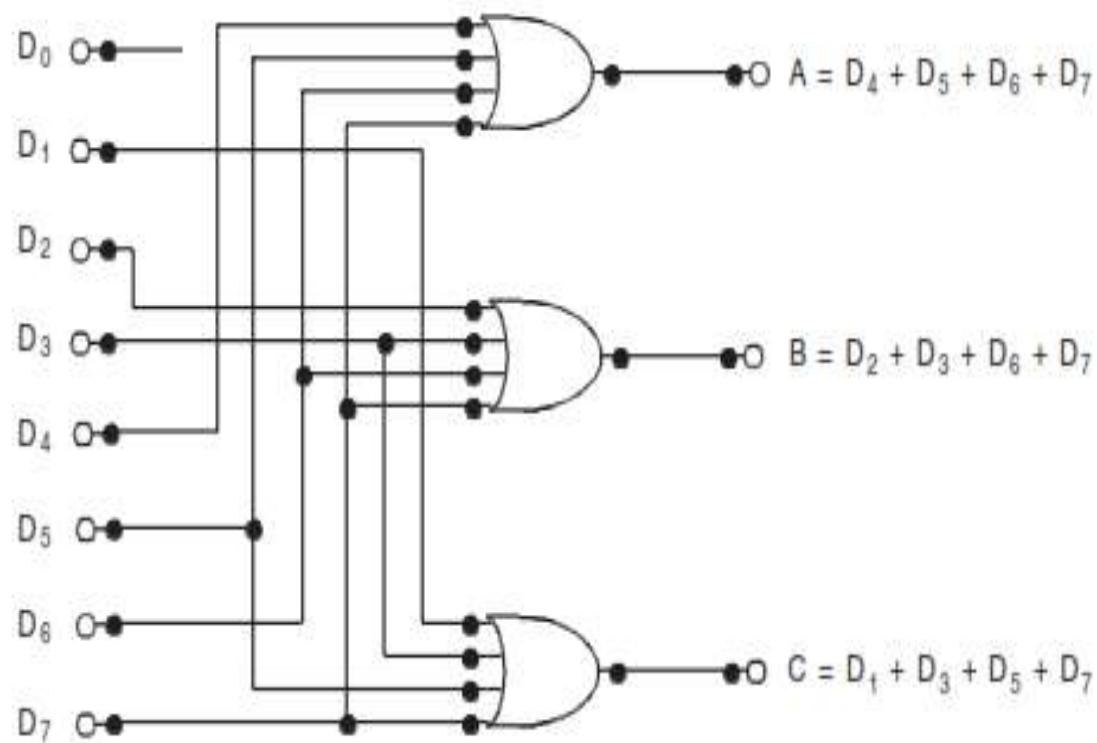
An encoder has 2^n (or fewer) input lines and n output lines, which generate the binary code corresponding to the input value

Octal-to-binary encoder: 8 inputs (one for each of the octal digits) and three outputs generating the corresponding binary number

- multiple inputs: undefined => priority encoder
- Input with all 0's = D_0 is equal to 1

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	A	B	C
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

8 to 3 line encoder



Priority Encoder

A priority encoder is an encoder circuit that includes the priority function: if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.

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

X's in inputs: condensed form, either 0 or 1

X's in outputs: don't-care

V ; valid bit indicator that is set to 1 when one or more inputs are equal to 1. If all inputs are 0, there is no valid input and V is equal to 0.

$$x = D_2 + D_3$$

$$y = D_3 + D_1 D_2'$$

$$V = D_0 + D_1 + D_2 + D_3$$

$D_0 D_1$	$D_2 D_3$	00	01	$\overbrace{11}^{D_2}$	$\overbrace{10}^{D_3}$
D_0	m_0	X	m_1	m_3	m_2
	m_4		m_5	m_7	m_6
	m_{12}	m_{13}	m_{15}	m_{14}	
	m_8	m_9	m_{11}	m_{10}	

$$x = D_2 + D_3$$

$D_0 D_1$	$D_2 D_3$	00	01	$\overbrace{11}^{D_2}$	$\overbrace{10}^{D_3}$
D_0	m_0	X	m_1	m_3	m_2
	m_4	1		m_7	m_6
	m_{12}	1	m_{13}	m_{15}	m_{14}
	m_8		m_9	m_{11}	m_{10}

$$y = D_3 + D_1 D_2'$$

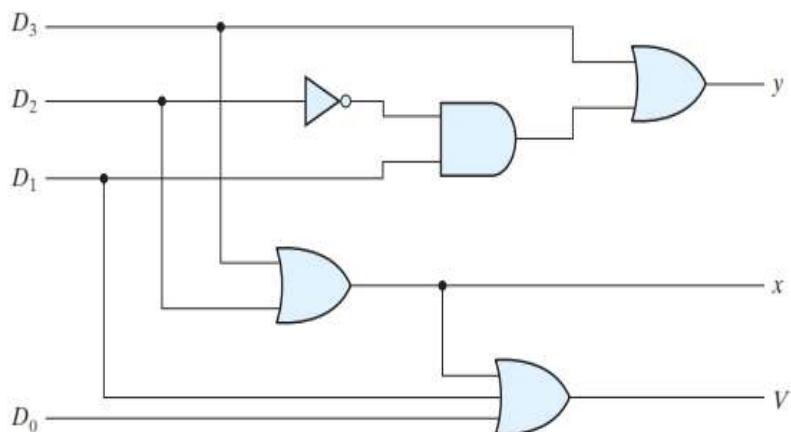


FIGURE 4.23
Four-input priority encoder

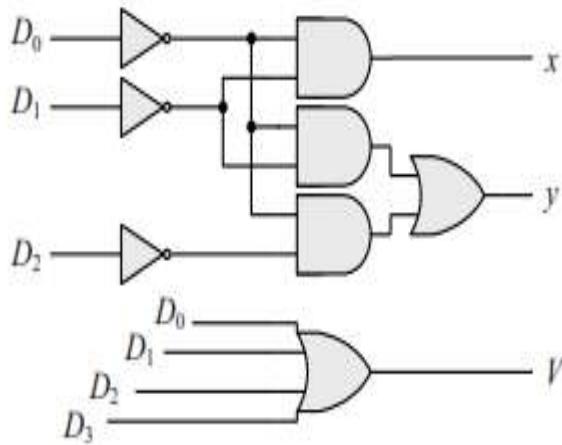
Example

Design a four-input priority encoder with inputs as in Table 4.8, but with input D_0 having the highest priority and input D_3 the lowest priority.

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

Solution:



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

D_3D_2	D_1D_0	00	01	11	10	D_1
D_3	00	m_0	m_f	1	1	m_2
	01	m_e	1	1	1	m_6
	11	m_{j2}	m_{j3}	1	m_{j4}	1
	10	m_8	m_9	m_{j1}	m_{j0}	1

$$V = D_0 + D_1 + D_2 + D_3$$

D_3D_2	D_1D_0	00	01	11	10	D_1
D_3	00	m_0	m_f			m_2
	01	m_e	m_5			m_6
	11	m_{j2}	m_{j3}	m_{j4}		
	10	m_8	m_9	m_{j1}	m_{j0}	

D_3D_2	D_1D_0	00	01	11	10	D_1
D_3	00	m_0	m_f			m_2
	01		m_5			m_6
	11	m_{j2}	m_{j3}	m_{j4}		1
	10	m_8	m_9	m_{j1}	m_{j0}	1

$$x = D_i D_0'$$

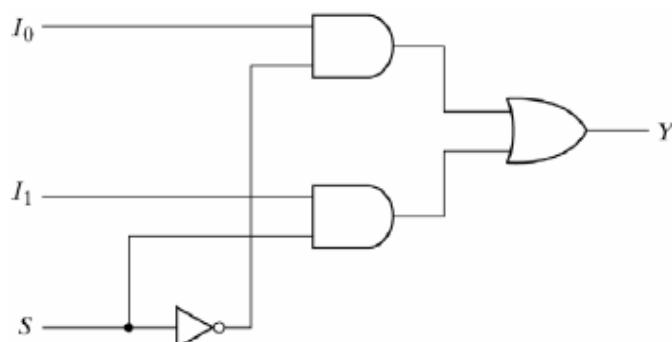
$$y = D_0' D_2' + D_1 D_0'$$

Multiplexers

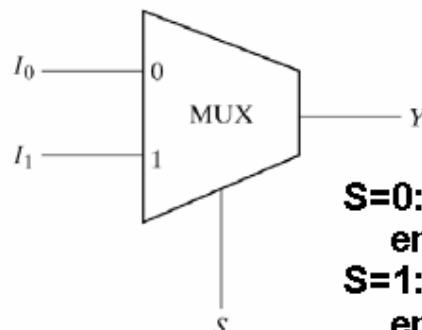
A combinational circuit that selects binary information from one of many input lines and directs it to a single output line

- Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected
- also called a **data selector**

2-to-1-line multiplexer: two data input lines, one output line, and one selection line S



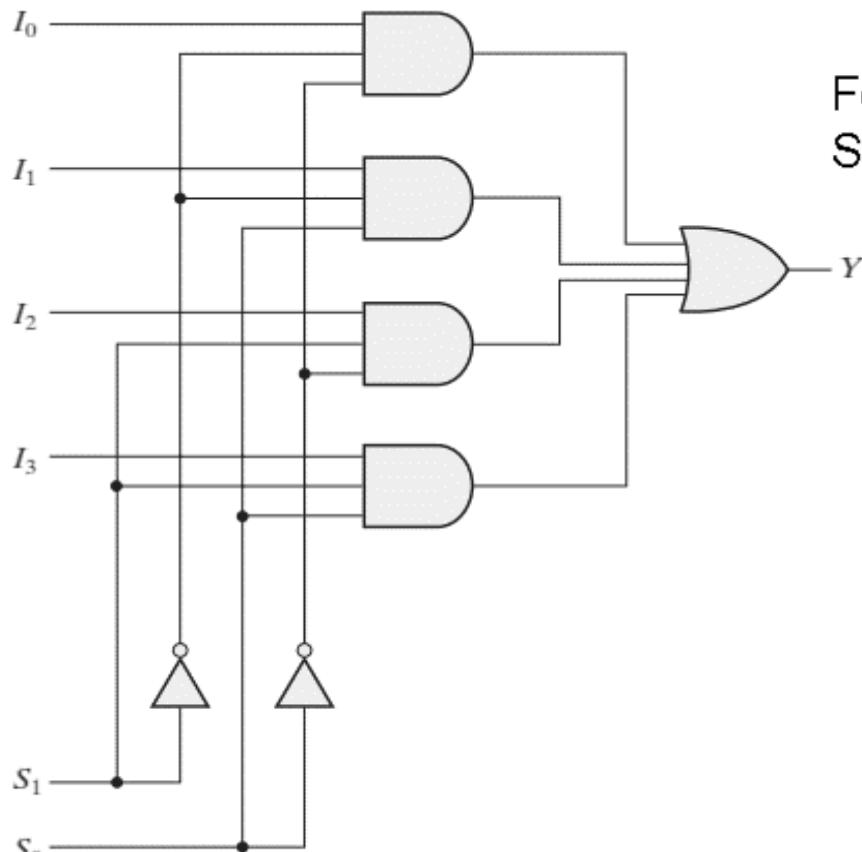
(a) Logic diagram



(b) Block diagram

S=0: the upper AND is enabled and $Y = I_0$
S=1: the lower AND is enabled and $Y = I_1$

4 to 1 line Multiplexer



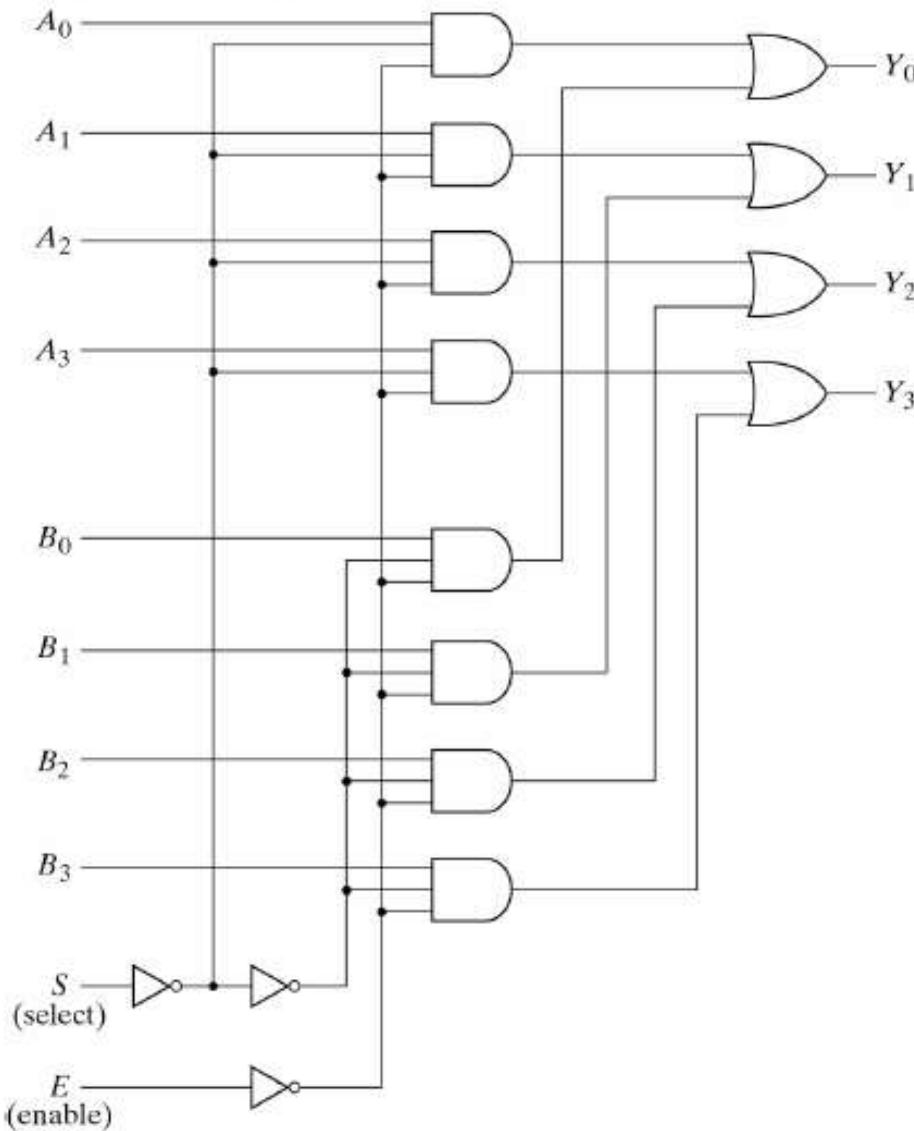
(a) Logic diagram

Four inputs: I_0 through I_3
Selection lines S_1 and S_0

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

Quadruple 2-to-1-line multiplexer with enable input



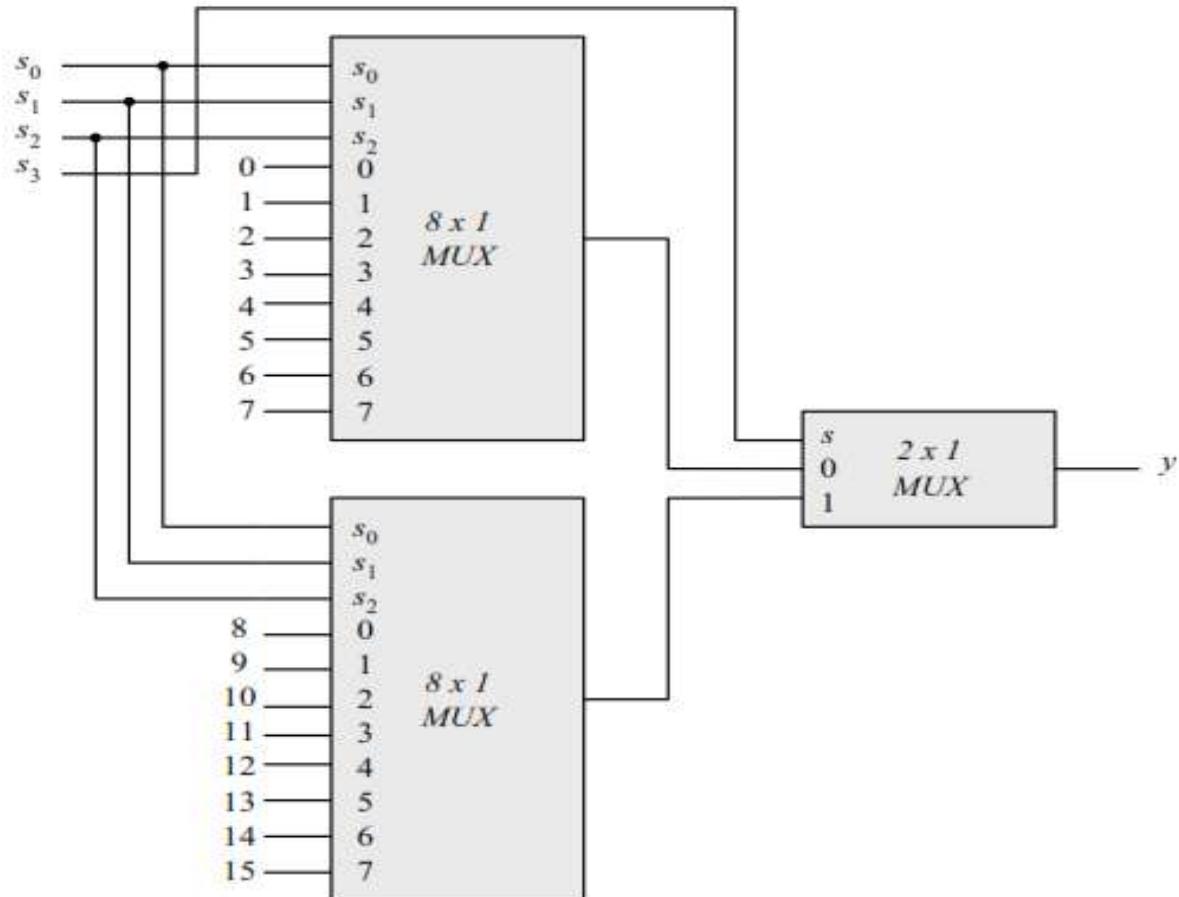
- Select one of two 4-bit sets of data lines
- 4 multiplexers, each capable of selecting one of two input lines
- Selection line S
- Enable line E

Function table

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

Problem: Construct a 16×1 multiplexer with two 8×1 and one 2×1 multiplexers. Use block diagrams

Solution:



Boolean Function Implementation

Implementing a Boolean function of n variables with a multiplexer that has $n-1$ selection lines (2^{n-1} inputs)

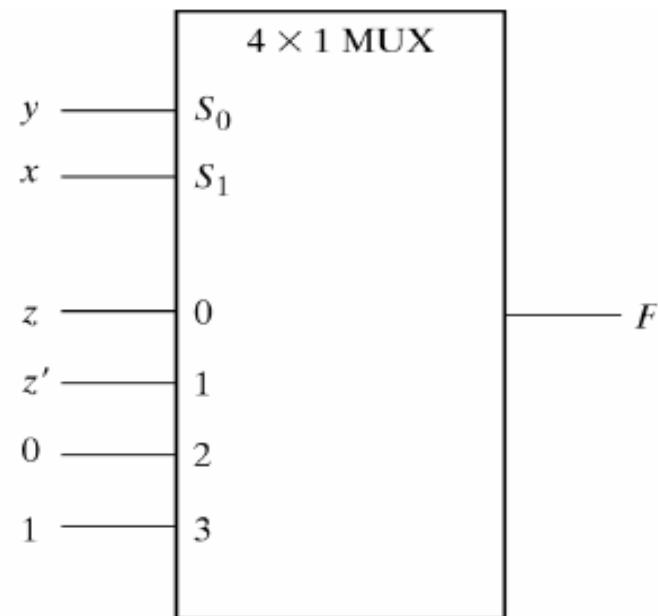
- the first $n-1$ variables are connected to the selection inputs
- the remaining single variable is used for the data input: z , z' , 1, or 0

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

- x, y : selection inputs S_1 and S_0

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

(a) Truth table



(b) Multiplexer implementation

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

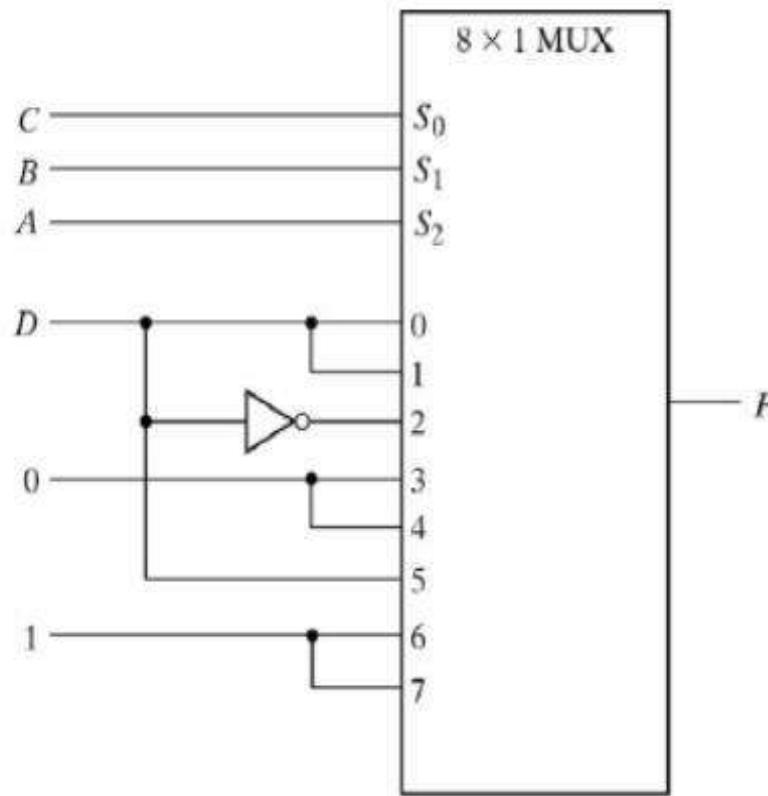
Implementing with a multiplexer of n selection lines

Implementing a 4-input function with a multiplexer

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

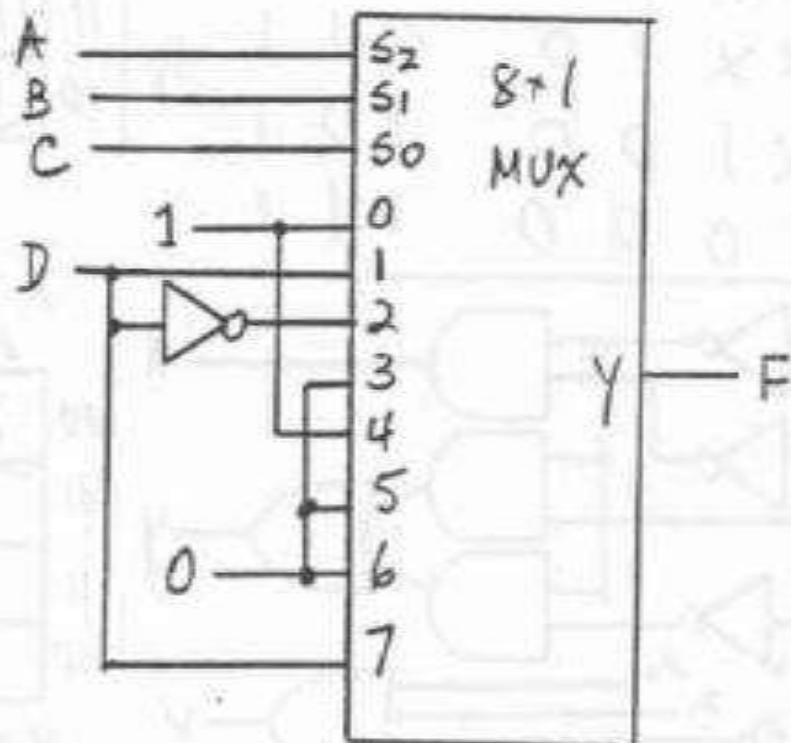
1. List the truth table of F
2. Evaluate the output as a function of the last variable: 0, 1, the variable, or the complement of the variable
3. Apply the first n-1 variables to the selection inputs
4. Connect 0, 1, the variable and the complement of the variable to the data inputs according to the results of step 2

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



Problem : Implement the following Boolean function with a multiplexer: $F(A, B, C, D) = \Sigma(0, 1, 3, 4, 8, 9, 15)$

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



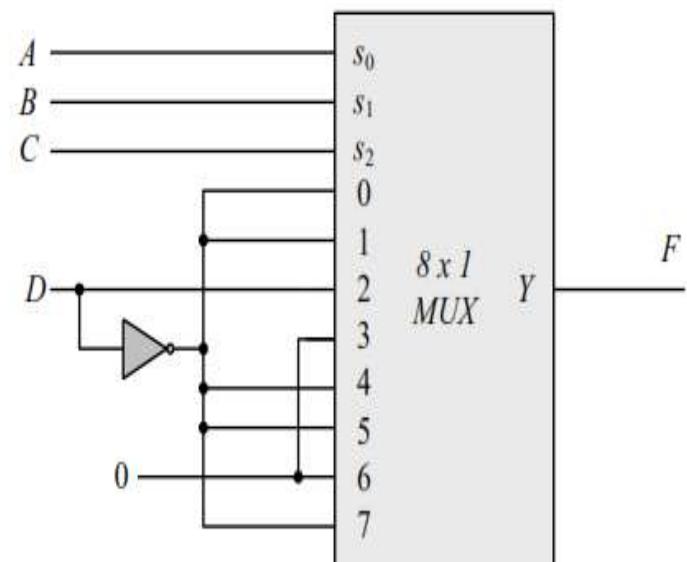
Example: Implement the following Boolean function with a multiplexer

(a) $F(A, B, C, D) = \Sigma(0, 2, 5, 8, 10, 14)$

(b) $F(A, B, C, D) = \Pi(2, 6, 11)$

Solution(a):

Inputs ABCD	$F = \Sigma(0, 2, 5, 8, 10, 14)$
000 0 0 0	1 $F = D'$
000 1 0 1	0
001 0 1 2	1
001 1 1 3	0 $F = D'$
010 0 2 4	0
010 1 2 5	1
011 0 3 6	0 $F = 0$
011 1 3 7	0
100 0 4 8	1 $F = D'$
100 1 4 9	0
101 0 5 10	1
101 1 5 11	0 $F = D'$
110 0 6 12	0 $F = 0$
110 1 6 13	0
111 0 7 14	1 $F = D'$
111 1 7 15	0



Solution(b):

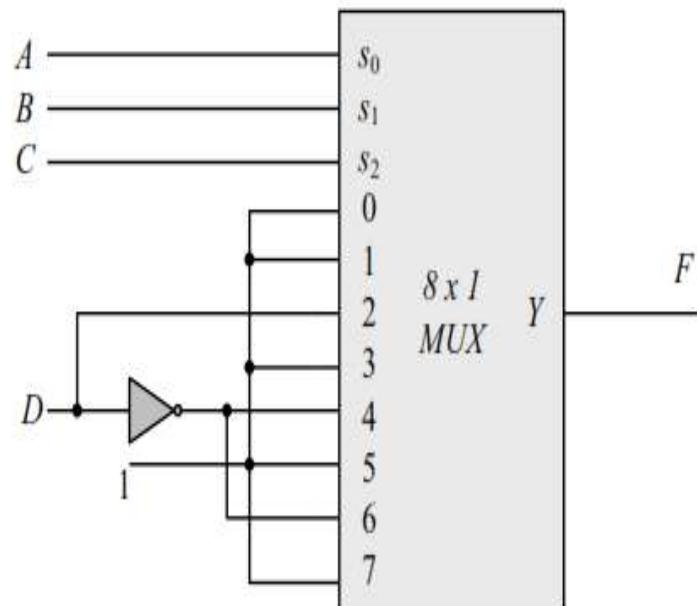
$$F = \Pi(2, 6, 11) = (A' + B' + C + D')(A' + B + C + D')(A + B' + C + D)$$

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

$$F' = (ABC'D) + (AB'C'D) + (A'BC'D') = \Sigma(13, 9, 4)$$

$$F = \Sigma(0, 1, 2, 3, 5, 6, 7, 8, 10, 11, 12, 14, 15)$$

<i>Inputs</i>				
<i>ABCD</i>				
0	0	0	0	$1 F = I$
0	0	1	1	1
0	1	0	2	$1 F = I$
0	1	1	3	1
0	1	0	4	$0 F = D$
0	1	0	5	1
0	1	1	6	$1 F = I$
0	1	1	7	1
1	0	0	8	$1 F = D'$
1	0	1	9	0
1	0	1	10	$1 F = I$
1	0	1	11	1
1	1	0	12	$1 F = D'$
1	1	0	13	0
1	1	0	14	$1 F = I$
1	1	1	15	1



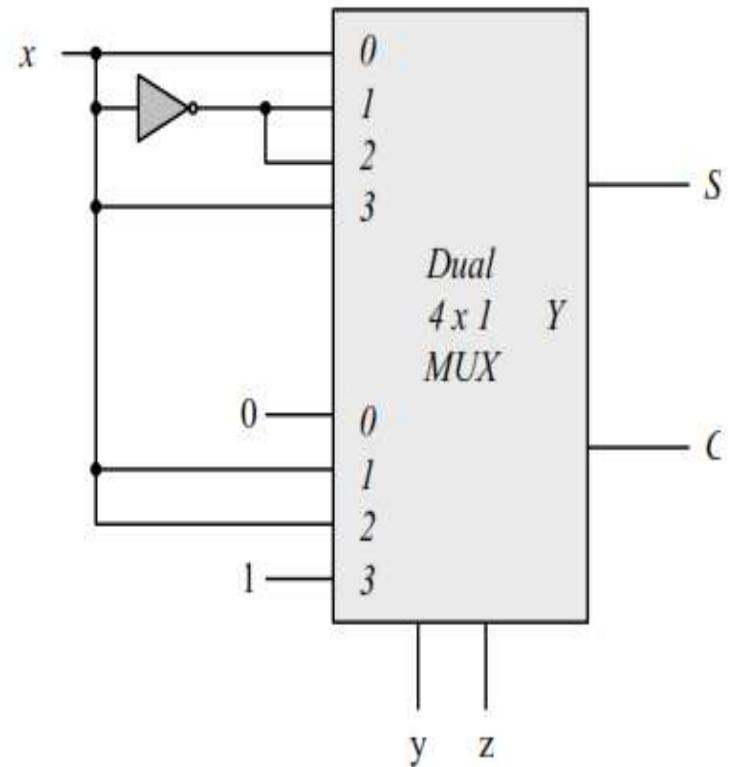
Example: Implement a full adder with two 4×1 multiplexers.

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

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

S	I_0	I_1	I_2	I_3
x'	0	1	2	3
x	4	5	6	7
	x	x'	x'	x

C	I_0	I_1	I_2	I_3
x'	0	1	2	3
x	4	5	6	7
	0	x	x	1



Example:

An 8×1 multiplexer has inputs A , B , and C connected to the selection inputs S_2 , S_1 , and S_0 , respectively. The data inputs I_0 through I_7 are as follows:

- (a)* $I_1 = I_2 = I_7 = 0; I_3 = I_5 = 1; I_0 = I_4 = D$; and $I_6 = D'$.
- (b) $I_1 = I_2 = 0; I_3 = I_7 = 1; I_4 = I_5 = D$; and $I_0 = I_6 = D'$.

Determine the Boolean function that the multiplexer implements.

Mux based designs

e.g. To implement the function the function

$$F(A, B, C) = \sum(1, 2, 5, 7) \text{ using } ① \text{ 8 to 1 MUX}$$

② 4 to 1 MUX.

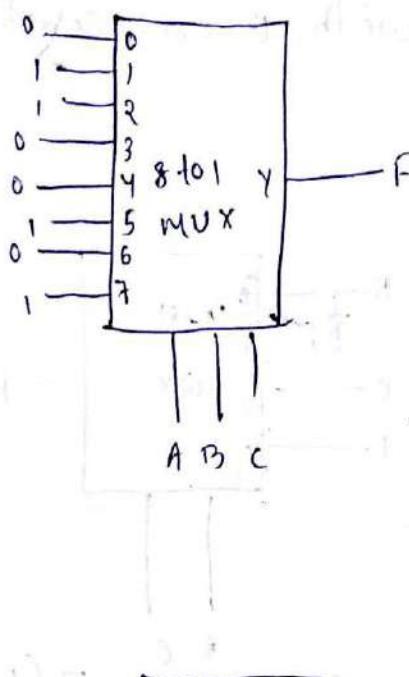
$$① F = A'B'C + AB'C' + A'B'C + ABC$$

②

NOW $N = 3$. So we use $2^{N-1} = 2^2$

= 4 to 1 MUX

BC as the Selection line



	$\bar{B}\bar{C}$	$\bar{B}C$	$B\bar{C}$	BC
	00	01	10	11
A	0	1	2	3
A'	4	5	6	7
	0	1	A'	A

$\bar{B}\bar{C}$	$\bar{B}C$	$B\bar{C}$	BC
00	0	1	2
01	0	1	2
10	0	1	2
11	0	1	2

Contd...

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

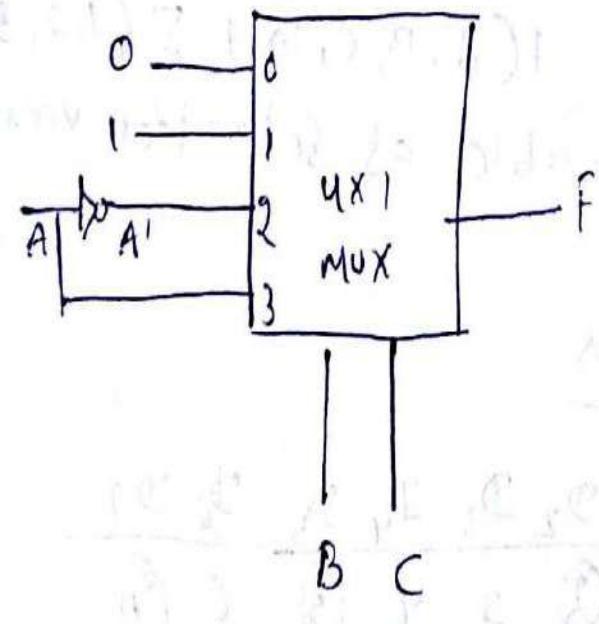
0 1 i 0 0 1 1 1

Σ

$$01 \rightarrow B'C (A' + A) = 1$$

$$10 \rightarrow Bc' \rightarrow A'$$

$$11 \rightarrow BC \rightarrow A$$



Ex To implement the function the function
 $F(A, B, C) = \Sigma(1, 2, 5, 7)$ using mux using different
variable as selection variable.

B → input

A & C as selection lines.

The minterms with B in complement form are
0, 1, 4, 5 and the minterms with B in uncomplemented
form are 2, 3, 6, 7.

	A	B	C		D ₀	D ₁	D ₂	D ₃
0	0	0	0	B'	0	1	4	5
1	0	0	1	B	2	3	6	7
2	0	1	0					
3	0	1	1					
4	1	0	0	B	B'	0	1	
5	1	0	1					
6	1	1	0					
7	1	1	1					

Ex Implement the function $F(A, B, C, D) = \sum(1, 2, 5, 7, 9, 14)$ using MUX using different variable as selection variable.

$A \rightarrow$ input

$B, C \& D \rightarrow$ Selection

$A' \rightarrow 0-7$

$A \rightarrow 8-15$

Minterms

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
A'	0	1	2	3	4	5	6	7
A	8	9	10	11	12	13	14	15
	0	1	A'	0	0	A'	A'	A'

$N =$ in the function } . $N = 4$ so MUX of $2^{N-1} = 2^3 = 8 + 01$

$N-1 \rightarrow$ selection line }

Ex, $2^{N-1} - 101$ MUX

$f = \Sigma(2, 3, 5, 7, 10, 11, 15)$. implement the function

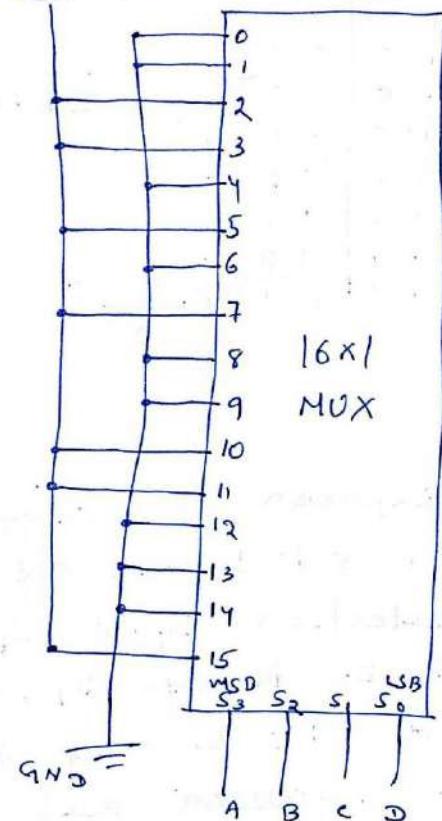
using 16×1 , 8×1 and 4×1 mux. (Hint: For 8×1 & 4×1 mux

sol?

BY 16×1 MUX

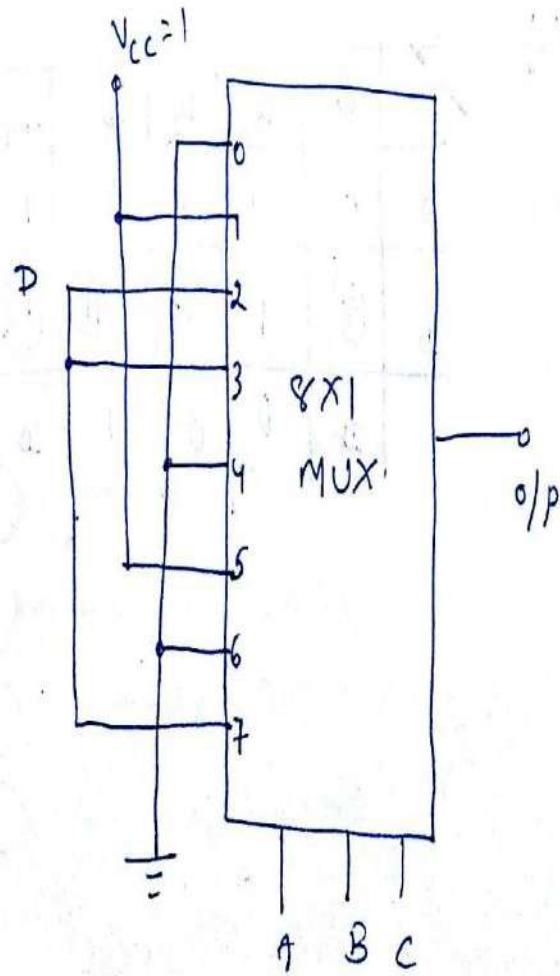
$V_{CC} = 1$

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



By 8x1 MUX.

	i/p			F
	A	B	C	D
0	0	0	0	0
	0	0	0	1
	0	0	1	0
1	0	0	1	1
	0	1	0	0
2	0	1	0	1
	0	1	1	0
3	0	1	1	1
	1	0	0	0
4	1	0	0	1
	1	0	0	0
	1	0	1	0
5	1	0	1	1
	1	1	0	0
6	1	1	0	0
	1	1	1	0
7	1	1	1	1



BY 4x1 MUX

$$F = \Sigma (2, 3, 5, 7, 10, 11, 15)$$

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

$$\Rightarrow \bar{C}\bar{D} + C\bar{D} = C(\bar{D} + D) = C$$

	0	1	0	0	0
1	0	1	0	1	1
	0	1	1	0	0
	0	1	1	1	1

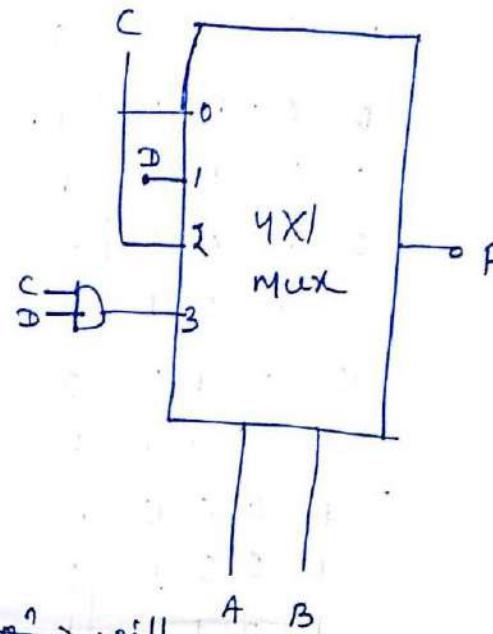
$$\Rightarrow \bar{C}D + C\bar{D} = D(\bar{C} + C) = D$$

	1	0	0	0	0
2	1	0	0	1	0
	1	0	1	0	1
	1	0	1	1	1

$$\Rightarrow C\bar{D} + C\bar{D} = C$$

	1	1	0	0	0
3	1	1	0	1	0
	1	1	1	0	0
	1	1	1	1	1

$$\Rightarrow CD$$



Ex Implement the following function:

Example : HOME WORK

- An 8x1 Multiplexer has the inputs A, B, C connected to the selection inputs S₂,S₁ and S₀ respectively. The data inputs are I₁ = I₂ = 0; I₃ = I₇ = 1; I₄ = I₅ = D and I₀ = I₆ = D'. Determine the Boolean function that the multiplexer implements.

Implementation of Boolean Function using Multiplexers

Implement :- $F(A,B,C,D) = \sum m(1,4,5,7,9,12,13)$ using 4×1 MUX.

		CD	$\bar{C}D$	$\bar{C}D$	CD	$C\bar{D}$
AB		00	01	11	10	
$\bar{A}\bar{B}$	00					
$\bar{A}\bar{B}$	01					
AB	11					
A \bar{B}	10					

→

		CD	$\bar{C}D$	$\bar{C}D$	CD	$C\bar{D}$
AB		00	01	11	10	
$\bar{A}\bar{B}$	00		1			
$\bar{A}\bar{B}$	01		1	1	1	
AB	11		1	1		
A \bar{B}	10		1			

Ex Implement $F(A, B, C, D) = \sum_m(1, 4, 5, 7, 9, 12, 13)$

using a 4-to-1 MUX.

Sel ⁿ	A ⁿ	B ⁿ	C ⁿ	D ⁿ	
(AB) ₂ 00			1		
I ₁ 01	1	1	1		$\rightarrow \bar{C}D$
I ₂ 11	1	1			$\rightarrow \bar{C}$
I ₃ 10		1			$\rightarrow \bar{C}D$

$AB \rightarrow \text{selector line}; CD \text{ input}$

$\rightarrow \bar{C}D$

$\rightarrow \bar{C}D + \bar{C}D + C\bar{D} = \bar{C} + C\bar{D} = \bar{C} + \bar{D}$

$\rightarrow \bar{C}$

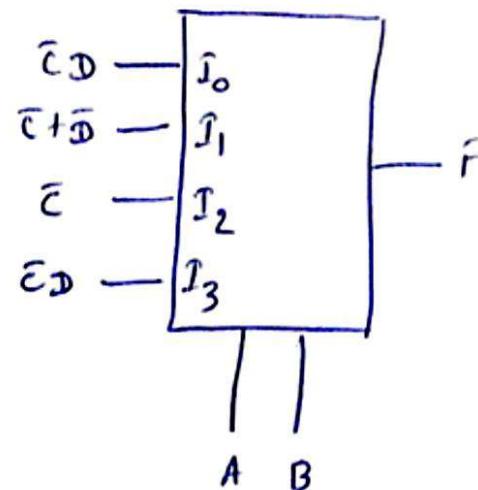
$\rightarrow \bar{C}D$

minimum no. of gates

if CD selector line:

$$B + 1 + \bar{A}B$$

$$I_0 \quad I_1 \quad I_3$$

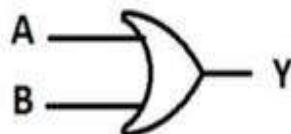


Example

A staircase light (L) is controlled by two switches one at the top of the stairs and another at the bottom of stairs, such that the light is ON when and only when one of the switch is ON and another switch is OFF. What will be the logic equation in SOP form.

IMPLEMENTATION OF LOGIC GATES BY USING MUX

OR Gate



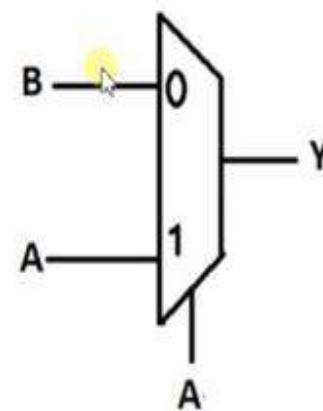
Logic Diagram

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

TRUTH TABLE

Observations from truth table

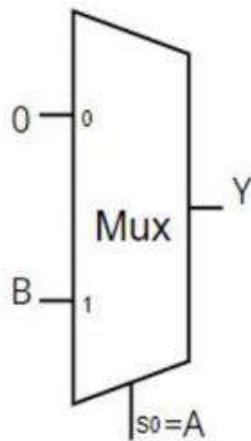
1. $Y=B$ if $A=0$
2. $Y=A$ if $A=1$



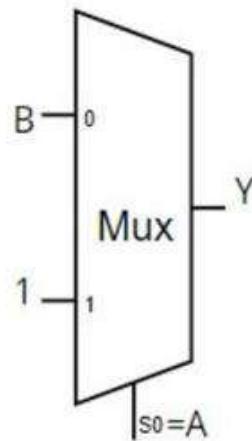
OR gate using 2:1 mux

Design of Logic Gates Using Mux

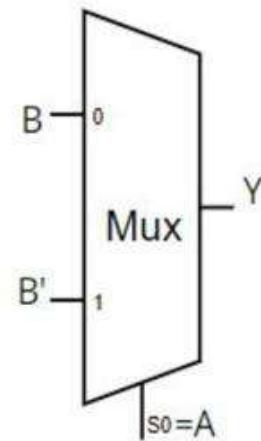
AND Gate



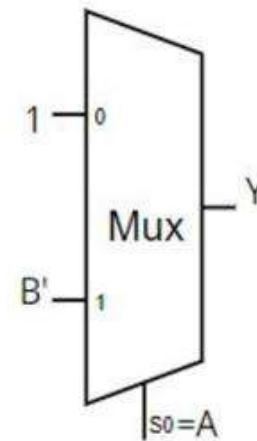
OR Gate



EXOR Gate

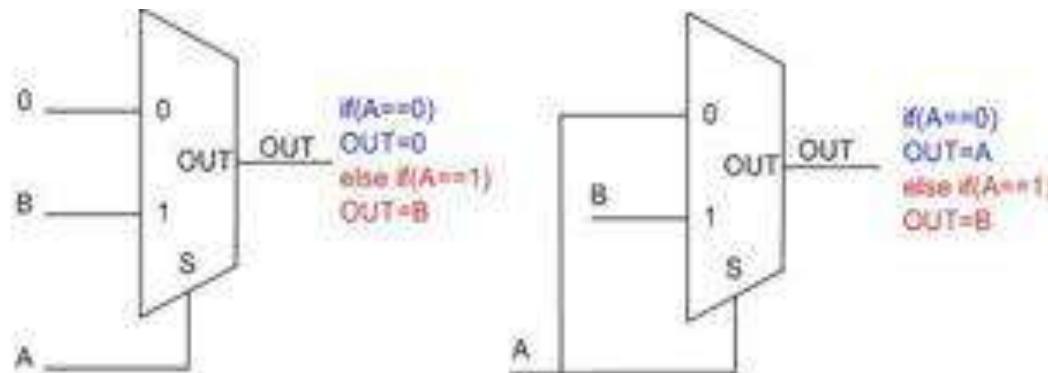


NAND Gate

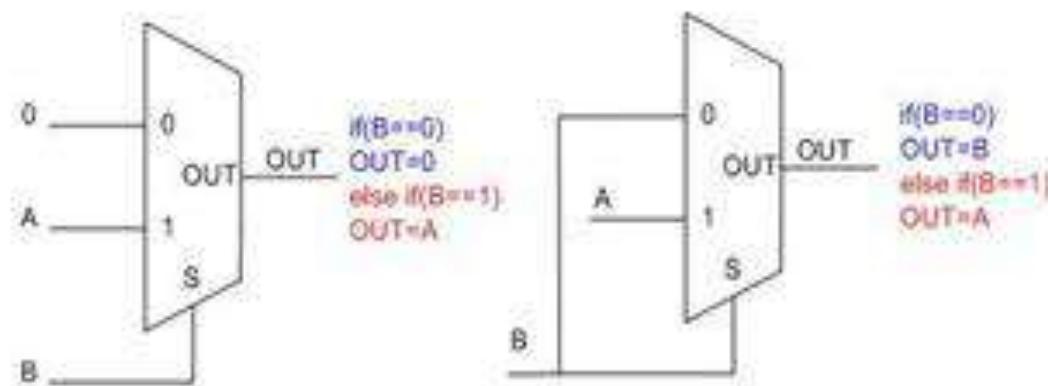


AND GATE IMPLEMENTATION WITH DIFFERENT SELECTORS

Inputs		Output
A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

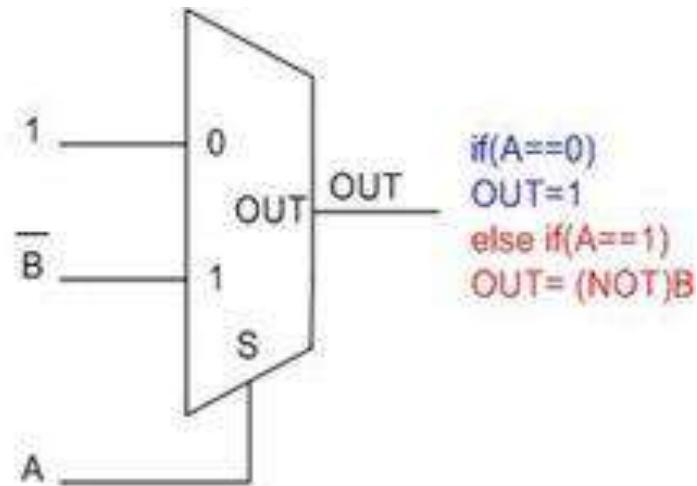


Inputs		Output
A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1



NAND GATE IMPLEMENTATION

Inputs		Output
A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0



IMPLEMENTATION OF LOGIC GATES BY USING MUX

② OR :

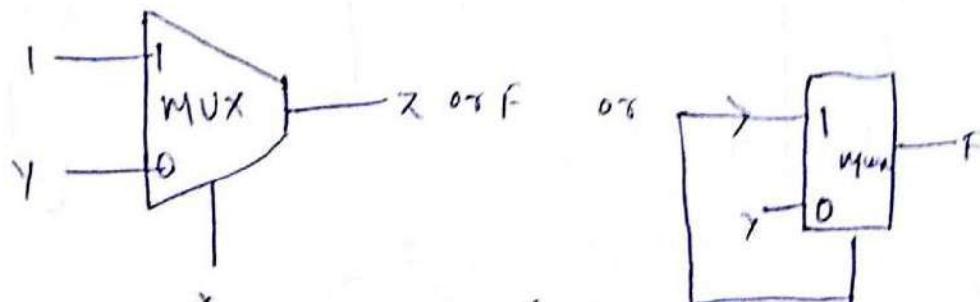
Truth Table

X	Y	O/P
0	0	0
0	1	1
1	0	1
1	1	1

$x \rightarrow$ Selection Line

$x = 0; \quad x = 1$

$x = 0; \quad \text{o/p is } y \quad (\text{i.e. same value of } x \text{ as } y \text{ column will appear at } y \text{ column}).$



$Z = X + Y; \text{ convert this to canonical form as}$

$$\begin{aligned} Z &= X(Y+Y') + Y(X+X') = XY + YY' + YX + YY' \\ &= XY + X Y' + X' Y = \Sigma(1, 2, 3) \end{aligned}$$

Half Adder using 4 to 1MUX

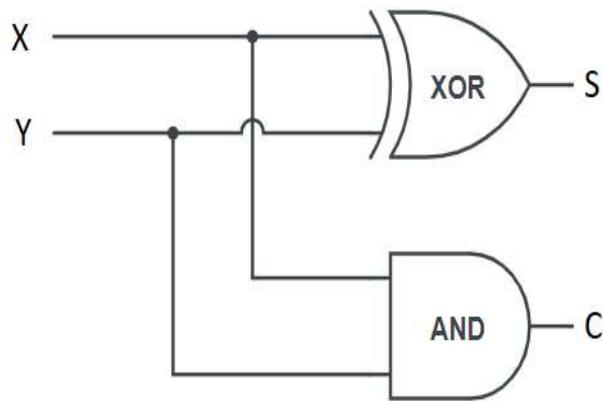
Truth Table of Half Adder and 4 to 1 MUX

INPUT		OUTPUT	
a	b	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

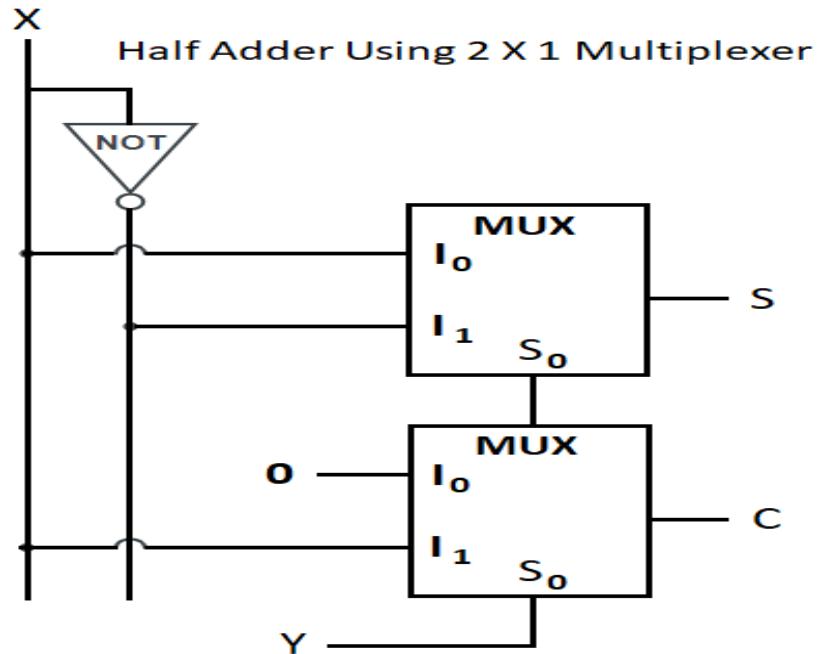
INPUT		OUTPUT
s1	s0	y
0	0	0 ₂
0	1	0 ₁
1	1	0 ₂
1	0	0 ₃

Implementation of half adder by using 2:1 MUX

Half Adder Logic Diagram



$$S = X'Y + XY' \text{ and } C = XY$$



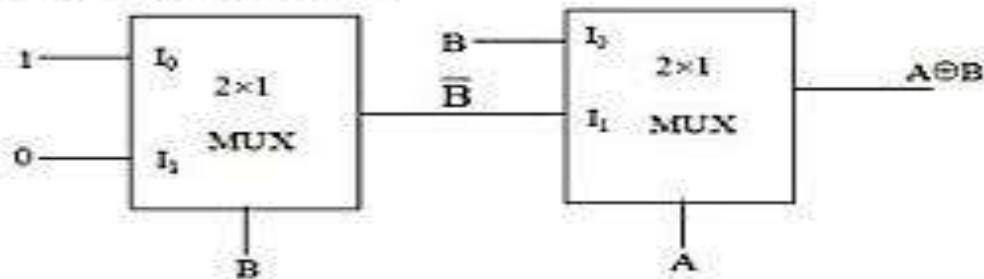
Half Adder Truth Table

Input		Output	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

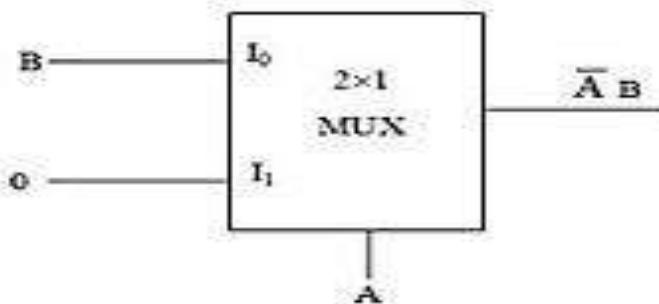
Q. What is minimum num of 2:1 MUX needs to implement half subtractor



For XOR (Difference)



For Borrow:

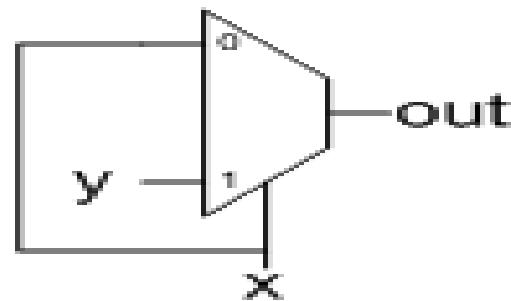
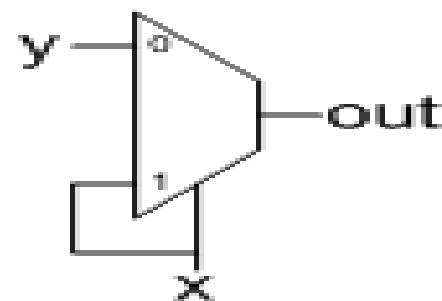
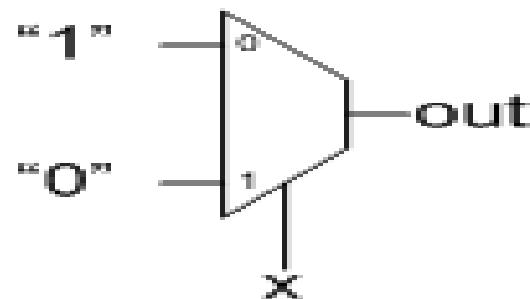


For difference two 2x1 MUX, for Borrow one 2x1 MUX

Total three 2x1 MUX required

NOT GATE

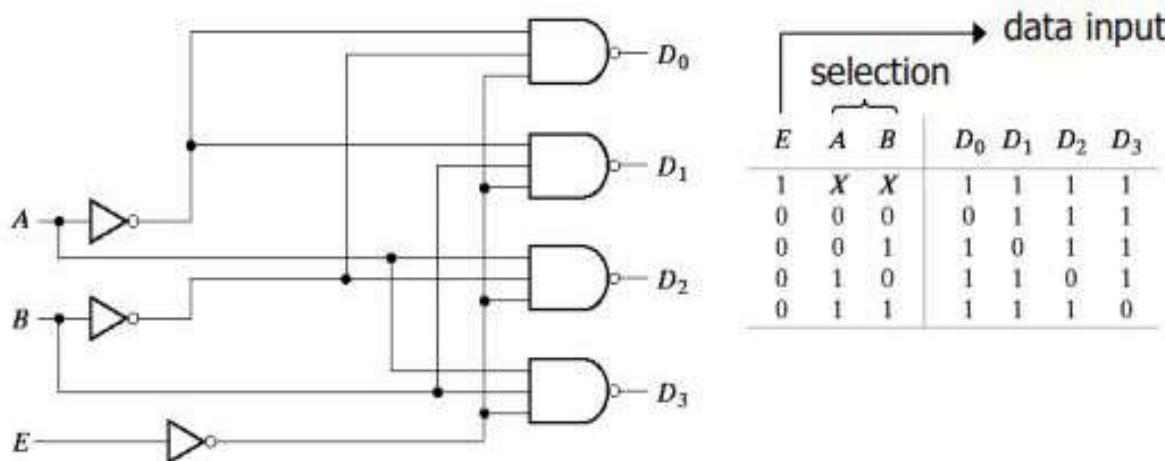
X	INPUT
0	1
1	0



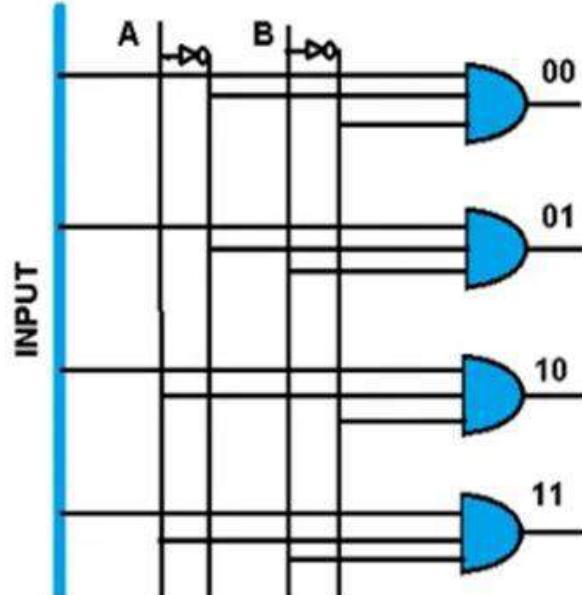
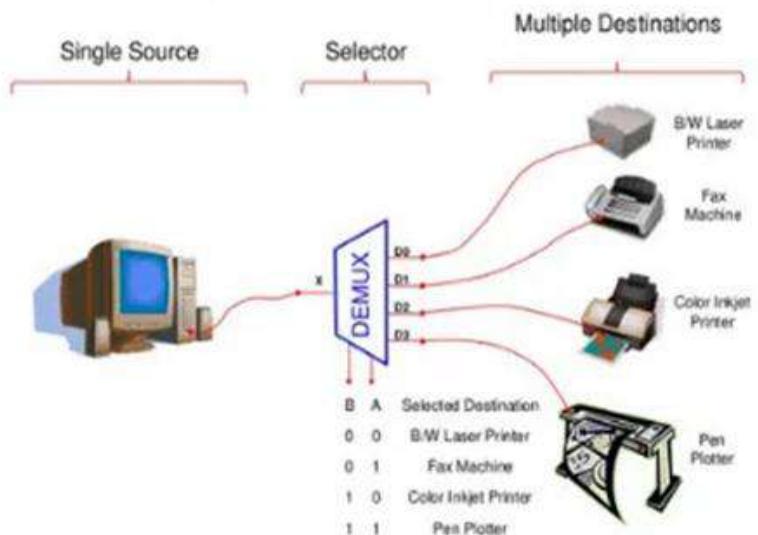
Demultiplexer

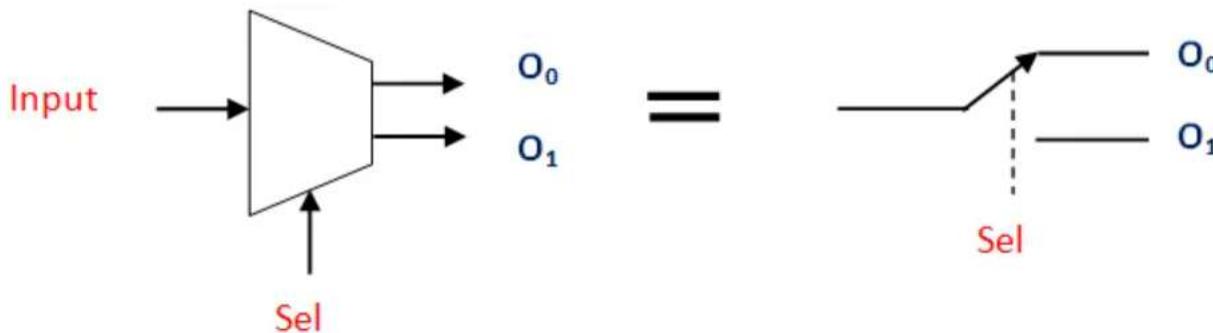
Demux or data Distributor

- A circuit that receives information from a single line and directs it to one of 2^n possible output lines
- A decoder with enable input can function as a demultiplexer
 - Often referred to as a ***decoder/demultiplexer***



What is a Demultiplexer?

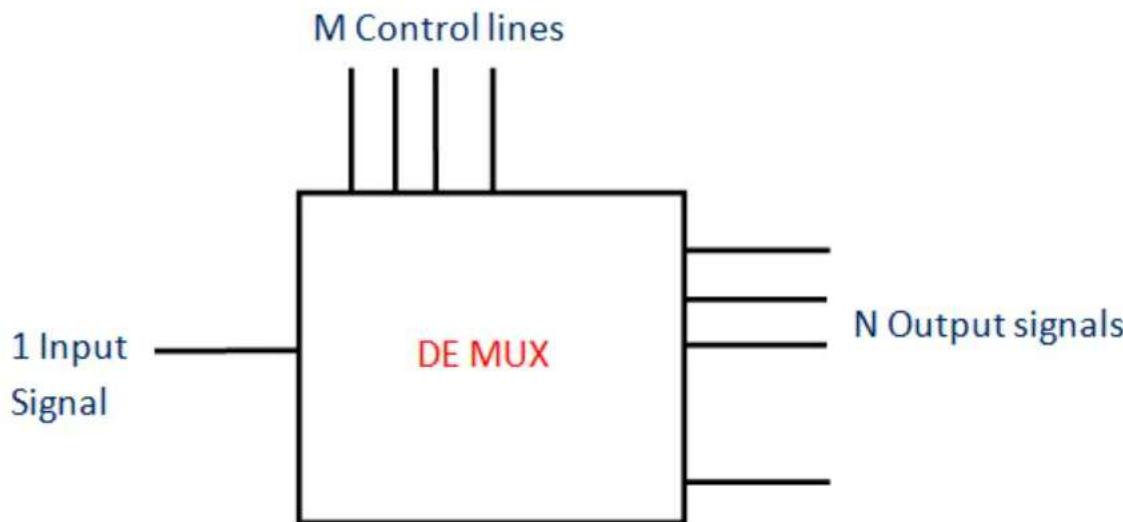




1-to-2 Demultiplexer Schematic

Controlled Switch

The pin diagram of the demultiplexer is in the figure below.



PARITY GENERATOR AND CHECKER

by Dr.Deepika Rani Sona

Parity error

- Irregular changes to data, as it is recorded when it is entered in memory.
- Different types of parity errors can require the retransmission of data or cause serious system errors, such as system crashes.

The most common error detection code used is the parity bit.

A parity bit is an extra bit included with a binary message to make the total number of 1's either odd or even.

In case of even parity, the parity bit is chosen so that the total number of 1's in the coded message is even.

Alternatively, odd parity can be used in which the total number of 1's in the coded message is made odd.

During transfer of information, the message at the sending-end is applied to a parity generator where the parity bit is generated.

Computers can sometimes make errors when they transmit data.

Even/odd parity:

Is basic method for detecting if an odd number of bits has been switched by accident.

Odd parity:

The number of 1-bit must add up to
an odd number

3-bit message			Odd parity bit generator (P)
A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Even parity:

The number of 1-bit must add up to an even number

3-bit message			Even parity bit generator (P)
A	B	C	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Boolean Expression

Even Pair

$$\begin{aligned} P &= \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ \\ &= \bar{X}(\bar{Y}Z + Y\bar{Z}) + X(\bar{Y}\bar{Z} + YZ) \\ &= \bar{X}(Y \oplus Z) + X(\bar{Y} \oplus Z) \\ &= X \oplus (Y \oplus Z) \end{aligned}$$

Odd Pair

$$\begin{aligned} P &= \bar{X}\bar{Y}\bar{Z} + \bar{X}YZ + X\bar{Y}Z + XY\bar{Z} \\ &= \bar{X}(\bar{Y}\bar{Z} + YZ) + X(\bar{Y}Z + Y\bar{Z}) \\ &= \bar{X}(\bar{Y} \oplus Z) + X(Y \oplus Z) \\ &= \bar{X} \oplus (Y \oplus Z) \end{aligned}$$

K-Map Simplification

X \ YZ	00	01	11	10
0	0	1	0	1
1	1	0	1	0

X \ YZ	00	01	11	10
0	1	0	1	0
1	0	1	0	1

Parity checker

It is a logic circuit that checks for possible errors in the transmission. This circuit can be an even parity checker or odd parity checker depending on the type of parity generated at the transmission end. When this circuit is used as even parity checker, the number of input bits must always be even.

Even Parity Checker

Consider that three input message along with even parity bit is generated at the transmitting end. These 4 bits are applied as input to the parity checker circuit, which checks the possibility of error on the data. Since the data is transmitted with even parity, four bits received at circuit must have an even number of 1s.

If any error occurs, the received message consists of odd number of 1s. The output of the parity checker is denoted by PEC (Parity Error Check).

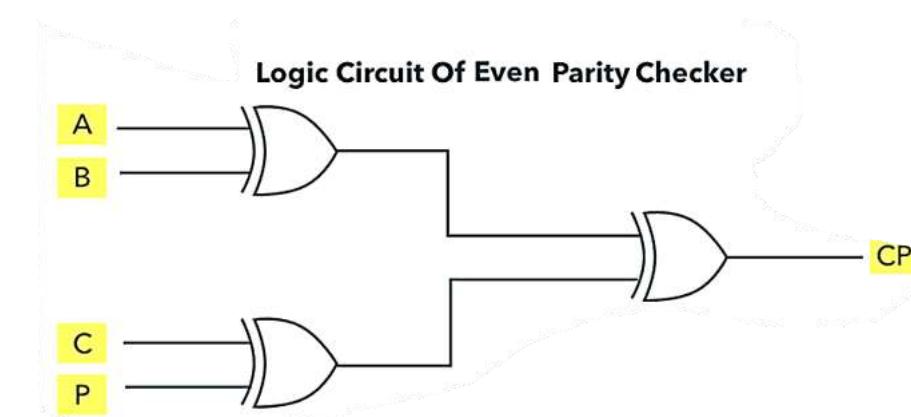
The below table shows the truth table for the Even Parity Checker in which PEC = 1 if the error occurs, i.e., the four bits received have odd number of 1s and PEC = 0 if no error occurs, i.e., if the 4-bit message has even number of 1s.

4-bit received message				Parity error check C_p
A	B	C	P	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

K-Map For Even Parity Checker

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

$$\begin{aligned}
 PEC &= \overline{A} \overline{B} (\overline{C}P + C\overline{P}) + \overline{A}B(\overline{C}\overline{P} + CP) + AB(\overline{C}P + C\overline{P}) + A\overline{B}(\overline{C}\overline{P} + CP) \\
 &= \overline{A} \overline{B} (C \oplus P) + \overline{A}B(\overline{C} \oplus P) + AB(C \oplus P) + A\overline{B}(\overline{C} \oplus P) \\
 &= (\overline{A} \overline{B} + AB)(C \oplus P) + (\overline{A}B + A\overline{B})(\overline{C} \oplus P) \\
 &= (A \oplus B) \oplus (C \oplus P)
 \end{aligned}$$



Odd Parity Checker

Consider that a three bit message along with odd parity bit is transmitted at the transmitting end. Odd parity checker circuit receives these 4 bits and checks whether any error are present in the data.

If the total number of 1s in the data is odd, then it indicates no error, whereas if the total number of 1s is even then it indicates the error since the data is transmitted with odd parity at transmitting end.

The below figure shows the truth table for odd parity generator where PEC = 1 if the 4-bit message received consists of even number of 1s (hence the error occurred) and PEC= 0 if the message contains odd number of 1s (that means no error).

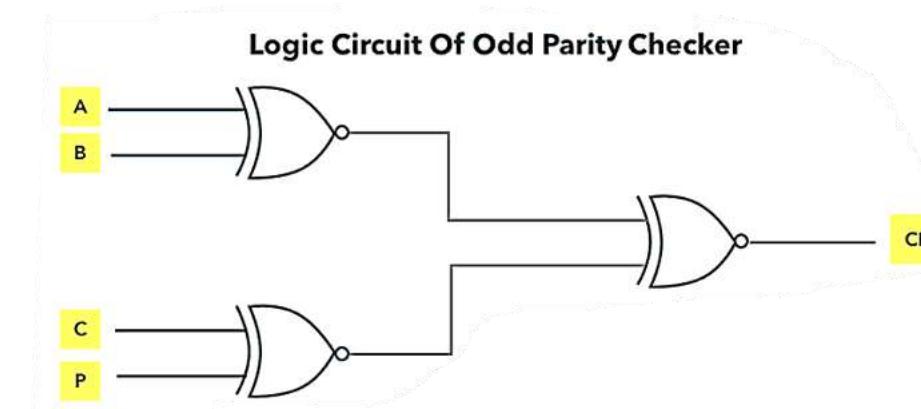
4-bit received message				Parity error check C_p
A	B	C	P	
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

K-Map for 3-bit Odd Parity Checker

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

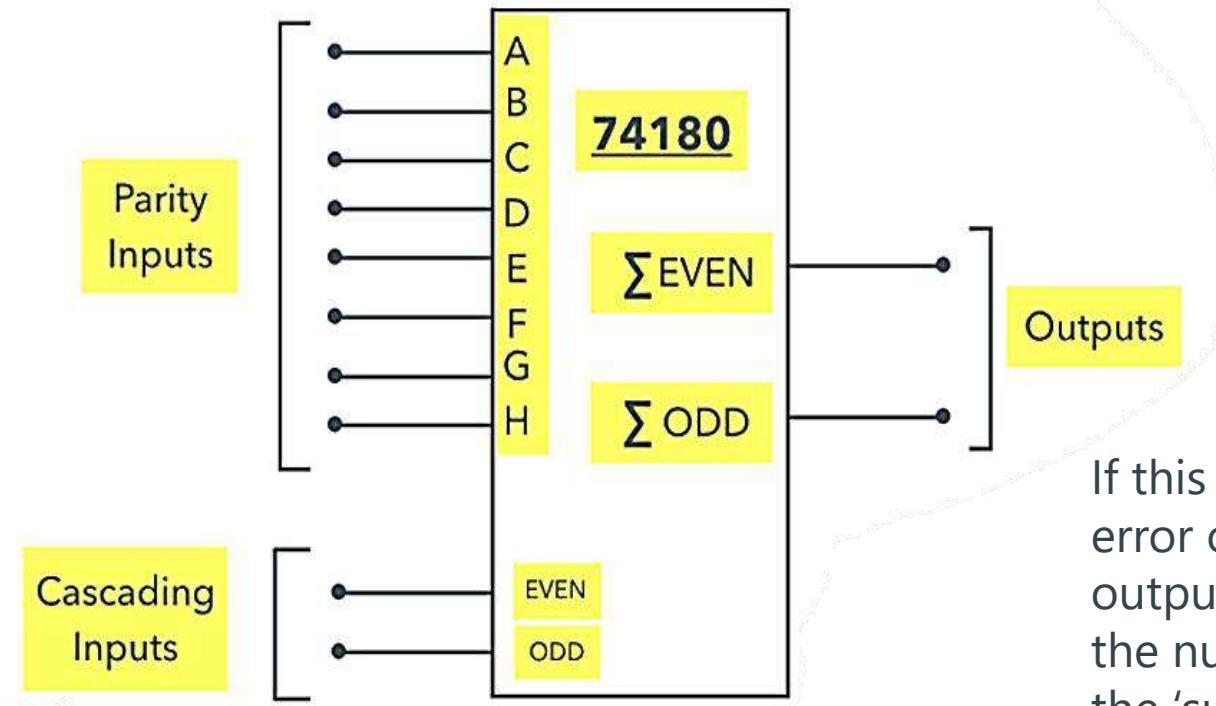
After simplification, the final expression for the PEC is obtained as

$$\text{PEC} = (\text{A Ex-NOR B}) \text{ Ex-NOR } (\text{C Ex-NOR P})$$



Parity Generator/Checker ICs

Parity Generator / Checker IC 74180



If this IC is used as an Even Parity Checker and when a parity error occurs, the 'sum even' output goes low and 'sum odd' output goes high. If this IC is used as an Odd Parity Checker, the number of input bits should be odd, but if an error occurs the 'sum odd' output goes low and 'sum even' output goes high.

Parity Generator/Checker applications:

- One important application of the use of an Exclusive-OR gate is to generate parity.
- Parity is used to detect errors in transmitted data caused by noise or other disturbances.
- A parity bit is an extra bit that is added to a data word and can be either odd or even parity.
- In an even parity system, the sum of all the bits (including the parity bit) is an even number
In an odd parity system the sum of all the bits must be an odd number.

- The circuit that creates the parity bit at the transmitter is called the parity generator. The circuit that determines if the received data is correct is the parity checker.
- Parity is good for detecting a single bit error only.
- The parity generator and the parity checker can both be built using Exclusive-OR gates.

THANK YOU!

Hardware Description Language - Introduction

- ◆ HDL is a language that describes the hardware of digital systems in a textual form.
- ◆ It resembles a programming language, but is specifically oriented to describing hardware structures and behaviors.
- ◆ The main difference with the traditional programming languages is HDL's representation of extensive parallel operations whereas traditional ones represents mostly serial operations.
- ◆ The most common use of a HDL is to provide an alternative to schematics.

HDL – Introduction (2)

- ◆ When a language is used for the above purpose (i.e. to provide an alternative to schematics), it is referred to as a *structural description* in which the language describes an interconnection of components.
- ◆ Such a structural description can be used as input to logic simulation just as a schematic is used.
- ◆ Models for each of the primitive components are required.
- ◆ If an HDL is used, then these models can also be written in the HDL providing a more uniform, portable representation for simulation input.

HDL – Introduction (3)

- ◆ HDL can be used to represent logic diagrams, Boolean expressions, and other more complex digital circuits.
- ◆ Thus, in top down design, a very high-level description of a entire system can be precisely specified using an HDL.
- ◆ This high-level description can then be refined and partitioned into lower-level descriptions as a part of the design process.

HDL – Introduction (4)

- ◆ As a documentation language, HDL is used to represent and document digital systems in a form that can be read by both humans and computers and is suitable as an exchange language between designers.
- ◆ The language content can be stored and retrieved easily and processed by computer software in an efficient manner.
- ◆ There are two applications of HDL processing: *Simulation* and *Synthesis*

Logic Simulation

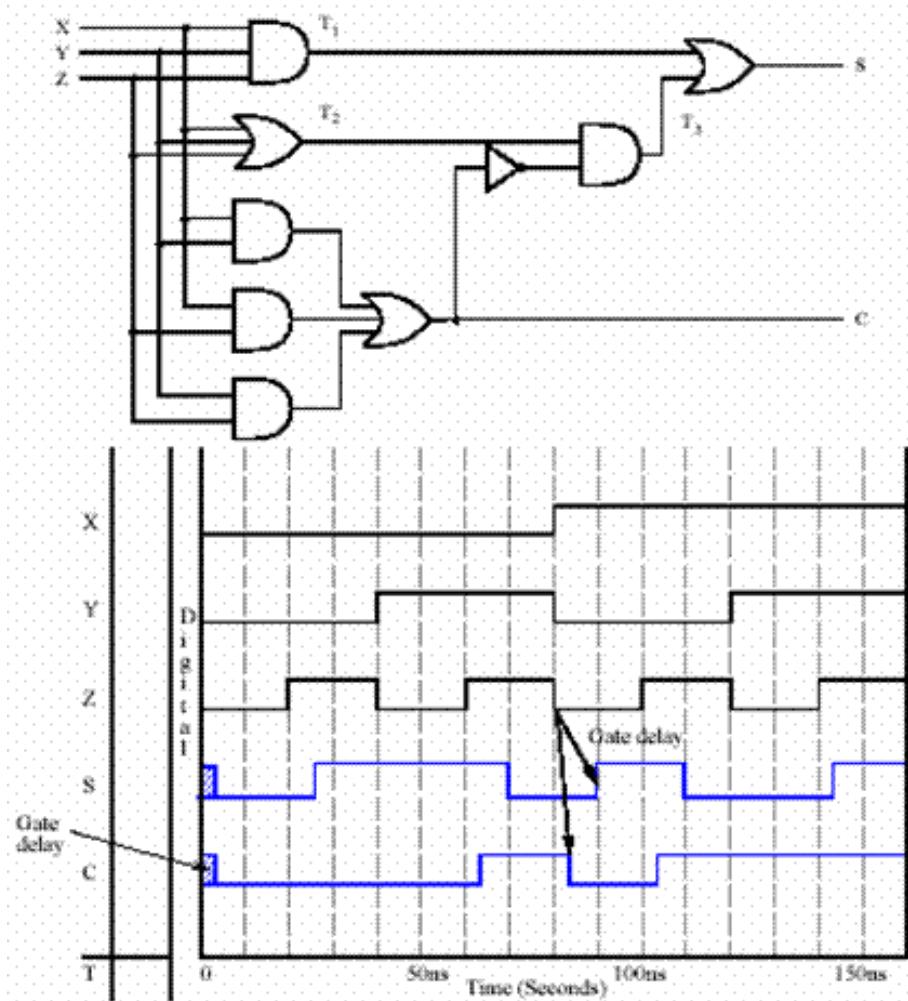
- ◆ A simulator interprets the HDL description and produces a readable output, such as a timing diagram, that predicts how the hardware will behave before its is actually fabricated.
- ◆ Simulation allows the detection of functional errors in a design without having to physically create the circuit.

Logic Simulation (2)

- ◆ The stimulus that tests the functionality of the design is called a test bench.
- ◆ To simulate a digital system
 - Design is first described in HDL
 - Verified by simulating the design and checking it with a test bench which is also written in HDL.

Logic Simulation

- ♦ Logic simulation is a fast, accurate method of analyzing a circuit to see its waveforms



Types of HDL

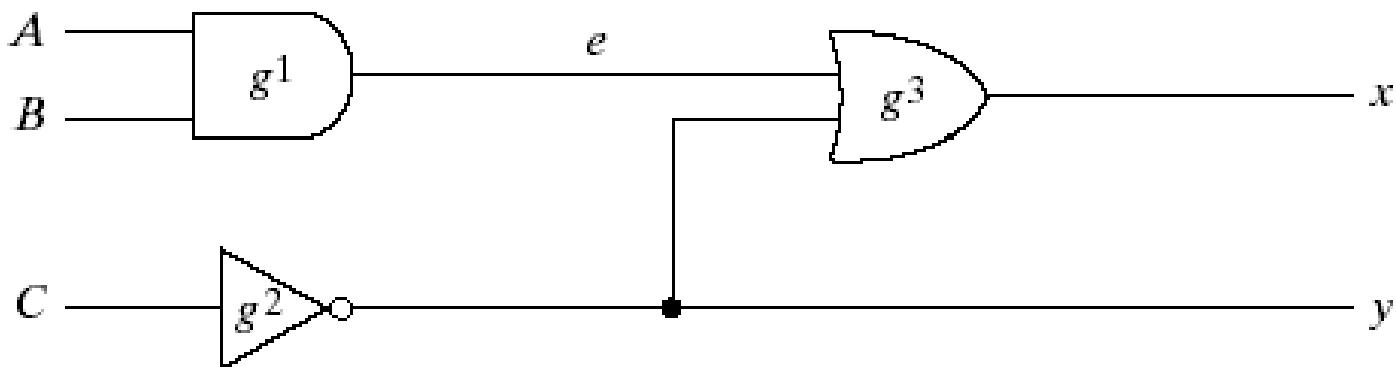
- ◆ There are two standard HDL's that are supported by IEEE.
 - **VHDL** (*Very-High-Speed Integrated Circuits Hardware Description Language*) - Sometimes referred to as VHSIC HDL, this was developed from an initiative by US. Dept. of Defense.
 - **Verilog HDL** – developed by Cadence Data systems and later transferred to a consortium called *Open Verilog International* (OVI).

Verilog

- ◆ Verilog HDL has a syntax that describes precisely the legal constructs that can be used in the language.
- ◆ It uses about 100 keywords pre-defined, lowercase, identifiers that define the language constructs.
- ◆ Example of keywords: *module, endmodule, input, output wire, and, or, not* , etc.,
- ◆ Any text between two slashes (//) and the end of line is interpreted as a comment.
- ◆ Blank spaces are ignored and names are case sensitive.

Verilog - Module

- ◆ A *module* is the building block in Verilog.
- ◆ It is declared by the keyword *module* and is always terminated by the keyword *endmodule*.
- ◆ Each statement is terminated with a semicolon, but there is no semi-colon after *endmodule*.



Verilog – Module (2)

HDL Example

```
module smpl_circuit(A,B,C,x,y);  
    input A,B,C;  
    output x,y;  
    wire e;  
    and g1(e,A,B);  
    not g2(y,C);  
    or  g3(x,e,y);  
endmodule
```

Verilog – Gate Delays

- ◆ Sometimes it is necessary to specify the amount of delay from the input to the output of gates.
- ◆ In Verilog, the delay is specified in terms of time units and the symbol #.
- ◆ The association of a time unit with physical time is made using ***timescale*** compiler directive.
- ◆ Compiler directive starts with the “backquote (`)” symbol.
`**timescale** 1ns/100ps
- ◆ The first number specifies the *unit of measurement* for time delays.
- ◆ The second number specifies the *precision* for which the delays are rounded off, in this case to 0.1ns.

Verilog – Module (4)

```
//Description of circuit with delay
module circuit_with_delay (A,B,C,x,y);
    input A,B,C;
    output x,y;
    wire e;
    and #(30) g1(e,A,B);
    or # (20) g3(x,e,y);
    not #(10) g2(y,C);
endmodule
```

Verilog – Module (5)

- ◆ In order to simulate a circuit with HDL, it is necessary to apply inputs to the circuit for the simulator to generate an output response.
- ◆ An HDL description that provides the stimulus to a design is called a **test bench**.
- ◆ The *initial* statement specifies inputs between the keyword **begin** and **end**.
- ◆ Initially ABC=000 (A,B and C are each set to 1'b0 (one binary digit with a value 0)).
- ◆ **\$finish** is a *system task*.

Verilog – Module (6)

```
module circuit_with_delay
(A,B,C,x,y);
input A,B,C;
output x,y;
wire e;
and #(30) g1(e,A,B);
or #(20) g3(x,e,y);
not #(10) g2(y,C);
endmodule
```

```
//Stimulus for simple circuit
module stimcrct;
reg A,B,C;
wire x,y;
circuit_with_delay cwd(A,B,C,x,y);
initial
begin
A = 1'b0; B = 1'b0; C = 1'b0;
#100
A = 1'b1; B = 1'b1; C = 1'b1;
#100 $finish;
end
endmodule
```

Verilog – Module (6)

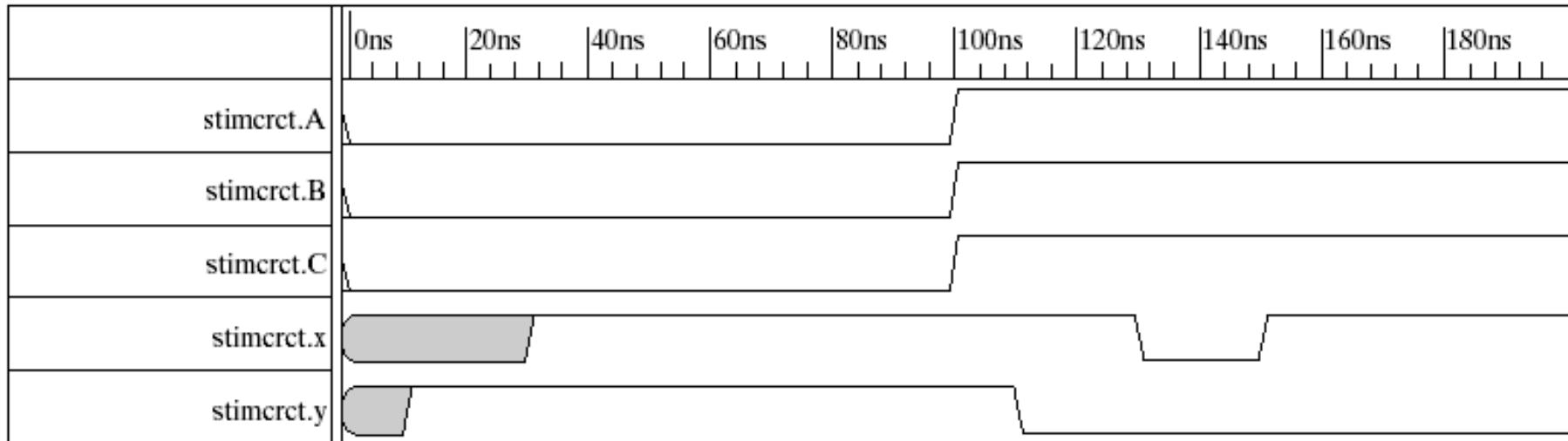


Fig. 3-38 Simulation Output of HDL Example 3-3

In the above example, **cwd** is declared as one instance **circuit_with_delay**. (similar in concept to object<->class relationship)

Verilog – Module (7)

Bitwise operators

- Bitwise NOT : \sim
- Bitwise AND: $\&$
- Bitwise OR: $|$
- Bitwise XOR: \wedge
- Bitwise XNOR: $\sim\wedge$ or $\wedge\sim$

Verilog – Module (8)

Boolean Expressions:

- ◆ These are specified in Verilog HDL with a continuous assignment statement consisting of the keyword *assign* followed by a Boolean Expression.
- ◆ The earlier circuit can be specified using the statement:

assign x = (A&B) | ~C)

E.g. $x = A + BC + B'D$

$y = B'C + BC'D'$

Verilog – Module (9)

```
//Circuit specified with Boolean equations
module circuit_bln (x,y,A,B,C,D);
    input A,B,C,D;
    output x,y;
    assign x = A | (B & C) | (~B & C);
    assign y = (~B & C) | (B & ~C & ~D);
endmodule
```

Verilog – Module (10)

User Defined Primitives (UDP):

- ◆ The logic gates used in HDL descriptions with keywords *and*, *or*, etc., are defined by the system and are referred to as *system primitives*.
- ◆ The user can create additional primitives by defining them in tabular form.
- ◆ These type of circuits are referred to as *user-defined primitives*.

Verilog – Module (12)

UDP features

- ◆ UDP's do not use the keyword module. Instead they are declared with the keyword ***primitive***.
- ◆ There can be **only one output** and it must be listed first in the port list and declared with an *output* keyword.
- ◆ There can be any number of inputs. The order in which they are listed in the ***input*** declaration must conform to the order in which they are given values in the table that follows.
- ◆ The truth table is enclosed within the keywords ***table*** and ***endtable***.
- ◆ The values of the inputs are listed with a colon (:). The output is always the last entry in a row followed by a semicolon (;).
- ◆ It ends with the keyword ***endprimitive***.

Verilog – Module (13)

```
//User defined primitive(UDP)
primitive crctp (x,A,B,C);
    output x;
    input A,B,C;
//Truth table for x(A,B,C) = Minterms (0,2,4,6,7)
    table
//      A   B   C   :   x   (Note that this is only a
comment)
      0   0   0   :   1;
      0   0   1   :   0;
      0   1   0   :   1;
      0   1   1   :   0;
      1   0   0   :   1;
      1   0   1   :   0;
      1   1   0   :   1;
      1   1   1   :   1;
endtable
endprimitive
```

```
// Instantiate primitive
module declare_crctp;
    reg     x,y,z;
    wire   w;
    crctp (w,x,y,z);
endmodule
```

Verilog – Module (14)

- ◆ A module can be described in any one (or a combination) of the following modeling techniques.
 - **Gate-level modeling** using instantiation of primitive gates and user defined modules.
 - This describes the circuit by specifying the gates and how they are connected with each other.
 - **Dataflow modeling** using continuous assignment statements with the keyword **assign**.
 - This is mostly used for describing combinational circuits.
 - **Behavioral modeling** using procedural assignment statements with keyword **always**.
 - This is used to describe digital systems at a higher level of abstraction.

Gate-Level Modeling

- ◆ Here a circuit is specified by its **logic gates** and their **interconnections**.
- ◆ It provides a textual description of a schematic diagram.
- ◆ Verilog recognizes 12 basic gates as predefined primitives.
 - 4 primitive gates of 3-state type.
 - Other 8 are: **and, nand, or, nor, xor, xnor, not, buf**
- ◆ When the gates are simulated, the system assigns a four-valued logic set to each gate – *0, 1, unknown (x) and high impedance (z)*

Gate-level modeling (2)

- ◆ When a primitive gate is incorporated into a module, we say it is *instantiated* in the module.
- ◆ In general, component instantiations are statements that reference lower-level components in the design, essentially creating unique copies (or *instances*) of those components in the higher-level module.
- ◆ Thus, a module that uses a gate in its description is said to *instantiate* the gate.

Gate-level Modeling (3)

- ◆ Modeling with vector data (multiple bit widths):
 - A vector is specified within square brackets and two numbers separated with a colon.

e.g. **output** [0:3] D; - This declares an output vector D with 4 bits, 0 through 3.

wire [7:0] SUM; – This declares a wire vector SUM with 8 bits numbered 7 through 0.

The first number listed is the most significant bit of the vector.

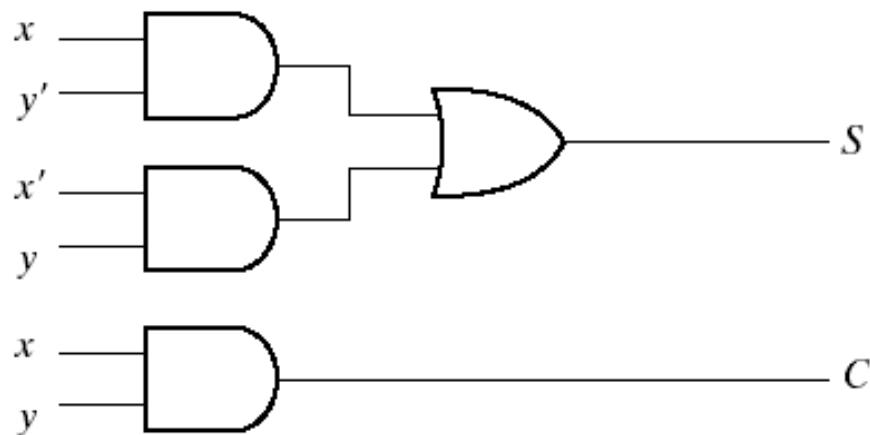
Gate-level Modeling

- ◆ Two or more modules can be combined to build a hierarchical description of a design.
- ◆ There are two basic types of design methodologies.
 - **Top down:** In top-down design, the top level block is defined and then sub-blocks necessary to build the top level block are identified.
 - **Bottom up:** Here the building blocks are first identified and then combine to build the top level block.
- ◆ In a top-down design, a 4-bit binary adder is defined as top-level block with 4 full adder blocks. Then we describe two half-adders that are required to create the full adder.
- ◆ In a bottom-up design, the half-adder is defined, then the full adder is constructed and the 4-bit adder is built from the full adders.

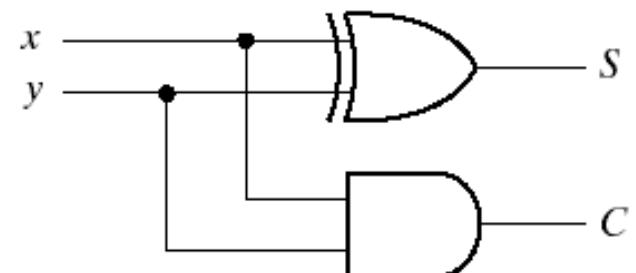
Gate-level Modeling

- ◆ A bottom-up hierarchical description of a 4-bit adder is described in Verilog as
 - Half adder: defined by instantiating primitive gates.
 - Then define the full adder by instantiating two half-adders.
 - Finally the third module describes 4-bit adder by instantiating 4 full adders.
- ◆ **Note:** In Verilog, one module definition cannot be placed within another module description.

4-bit Half Adder



$$(a) S = xy' + x'y \\ C = xy$$



$$(b) S = x \oplus y \\ C = xy$$

4-bit Full Adder

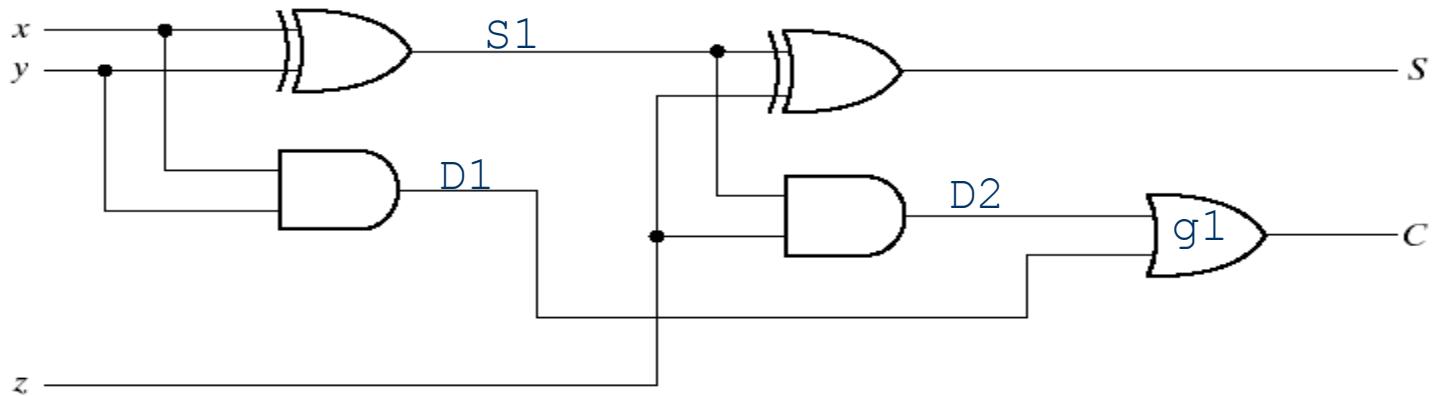


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

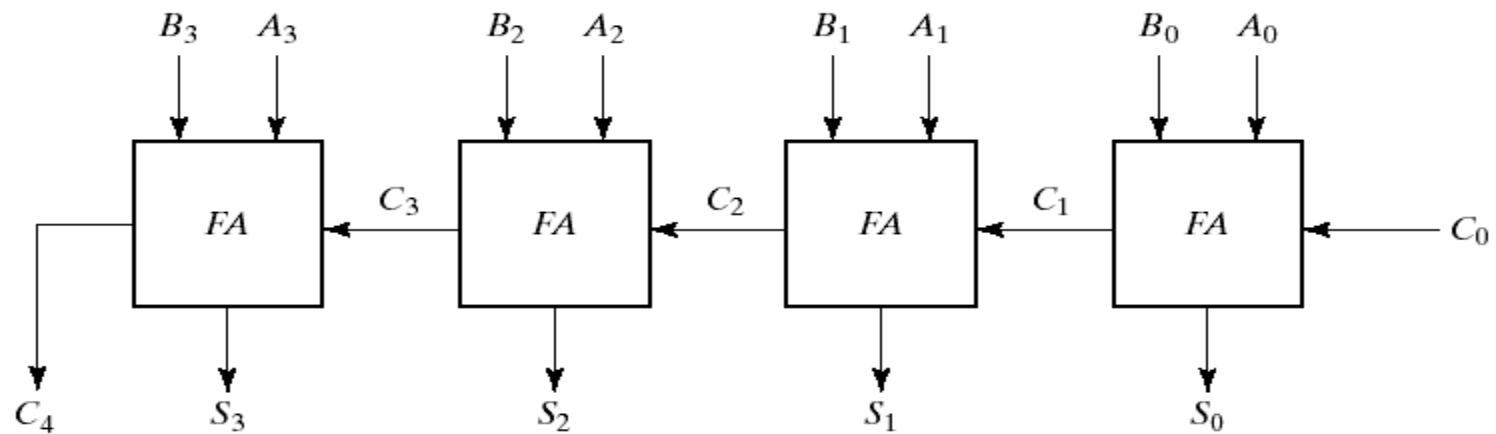


Fig. 4-9 4-Bit Adder

4-bit Full Adder

```
//Gate-level hierarchical description of 4-bit adder
module halfadder (S,C,x,y);
    input x,y;
    output S,C;
    //Instantiate primitive gates
    xor (S,x,y);
    and (C,x,y);
endmodule

module fulladder (S,C,x,y,z);
    input x,y,z;
    output S,C;
    wire S1,D1,D2; //Outputs of first XOR and two AND
    gates
    //Instantiate the half adders
    halfadder HA1(S1,D1,x,y), HA2(S,D2,S1,z);
    or g1(C,D2,D1);
endmodule
```

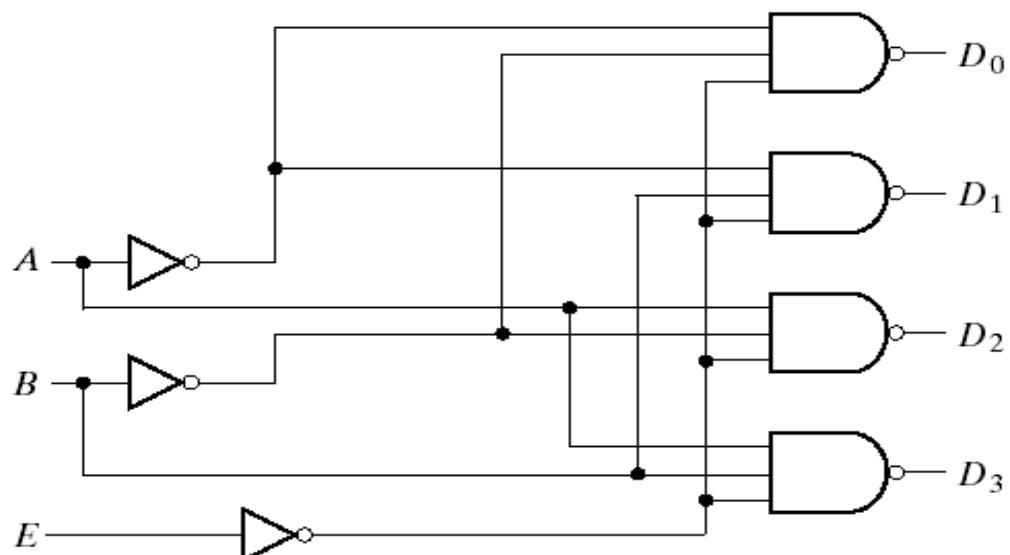
4-bit Full Adder

```
module _4bit_adder (S,C4,A,B,C0);
    input [3:0] A,B;
    input C0;
    output [3:0] S;
    output C4;
    wire C1,C2,C3; //Intermediate carries

    //Instantiate the full adder
    fulladder FA0 (S[0],C1,A[0],B[0],C0),
                FA1 (S[1],C2,A[1],B[1],C1),
                FA2 (S[2],C3,A[2],B[2],C2),
                FA3 (S[3],C4,A[3],B[3],C3);

endmodule
```

2 to 4 Decoder



(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

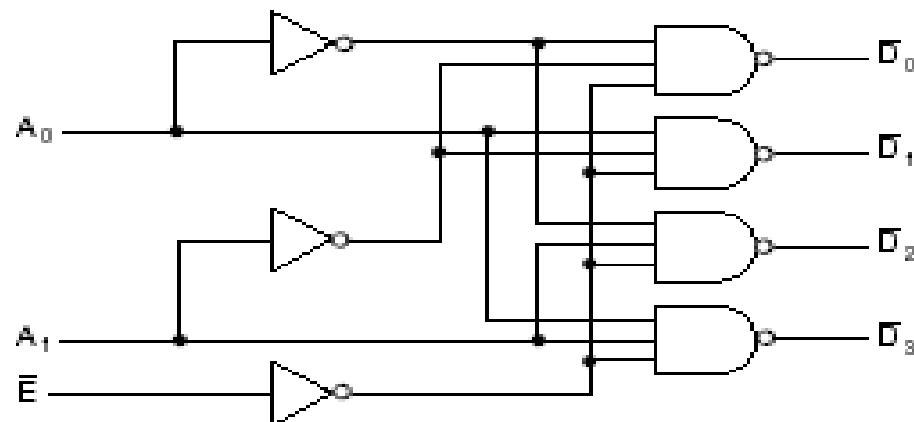
(b) Truth table

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

2 to 4 Decoder

```
//Gate-level description of a 2-to-4-line decoder
module decoder_gl (A,B,E,D);
    input A,B,E;
    output[0:3]D;
    wire Anot,Bnot,Enot;
    not
        n1 (Anot,A),
        n2 (Bnot,B),
        n3 (Enot,E);
    nand
        n4 (D[0],Anot,Bnot,Enot),
        n5 (D[1],Anot,B,Enot),
        n6 (D[2],A,Bnot,Enot),
        n7 (D[3],A,B,Enot);
endmodule
```

2-to-4 Line Decoder



(a) Logic diagram

E	A_1	A_0	D_0	D_1	D_2	D_3
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
1	X	X	1	1	1	1

(b) Truth table

$$D_0 = \overline{E} \overline{A}_1 A_0$$

$$D_1 = \overline{E} \overline{A}_1 \overline{A}_0$$

$$D_2 = \overline{E} A_1 \overline{A}_0$$

$$D_3 = \overline{E} A_1 A_0$$

(c) Logic Equations

Fig.3-14 A 2-to-4-Line Decoder

2-to-4 Line Decoder

```
//2 to 4 line decoder
module decoder_2_to_4_st_v(E_n, A0, A1, D0_n, D1_n,
D2_n, D3_n);
input E_n, A0, A1;
output D0_n, D1_n, D2_n, D3_n;

wire A0_n, A1_n, E;
not g0(A0_n, A0), g1(A1_n, A1), g2(E, E_n);
nand g3(D0_n, A0_n, A1_n, E), g4(D1_n, A0, A1_n, E),
g5(D2_n, A0_n, A1, E), g6(D3_n, A0, A1, E);
endmodule
```

Three-State Gates

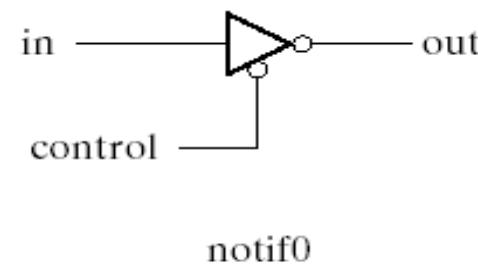
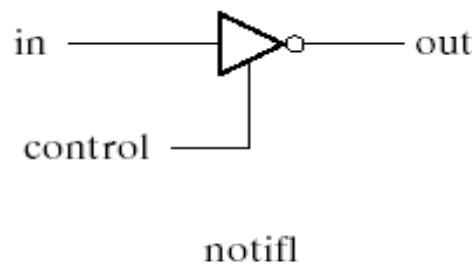
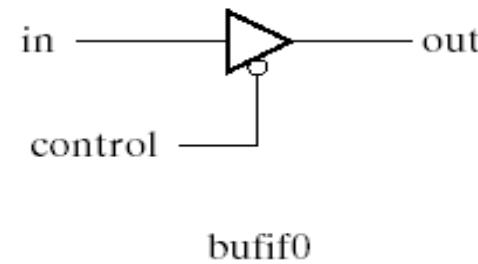
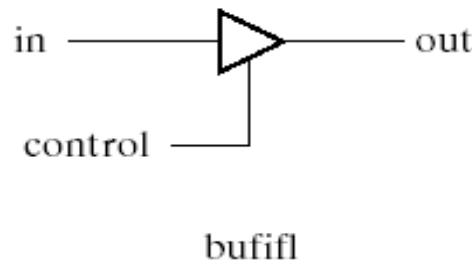


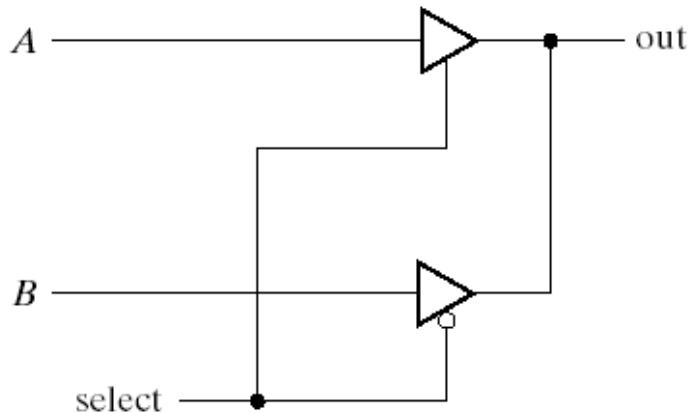
Fig. 4-31 Three-State Gates

Three-State Gates

- ◆ Three-state gates have a control input that can place the gate into a high-impedance state. (symbolized by **z** in HDL).
- ◆ The **bufif1** gate behaves like a normal buffer if `control=1`. The output goes to a high-impedance state **z** when `control=0`.
- ◆ **bufif0** gate behaves in a similar way except that the high-impedance state occurs when `control=1`
- ◆ Two **not** gates operate in a similar manner except that the o/p is the complement of the input when the gate is not in a high impedance state.
- ◆ The gates are instantiated with the statement
 - `gate name (output, input, control);`

Three-State Gates

The output of 3-state gates can be connected together to form a common output line. To identify such connections, HDL uses the keyword tri (for tri-state) to indicate that the output has multiple drivers.



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

Three-State Gates

- Keywords **wire** and **tri** are examples of *net* data type.
- Nets represent connections between hardware elements. Their value is continuously driven by the output of the device that they represent.
- The word *net* is not a keyword, but represents a class of data types such as **wire**, **wor**, **wand**, **tri**, **supply1** and **supply0**.
- The **wire** declaration is used most frequently.
- The net **wor** models the hardware implementation of the wired-OR configuration.
- The **wand** models the wired-AND configuration.
- The nets **supply1** and **supply0** represent power supply and ground.

Dataflow Modeling

- ◆ Dataflow modeling uses a number of operators that act on operands to produce desired results.
- ◆ Verilog HDL provides about 30 operator types.
- ◆ Dataflow modeling uses continuous assignments and the keyword **assign**.
- ◆ A continuous assignment is a statement that assigns a value to a net.
- ◆ The value assigned to the net is specified by an expression that uses operands and operators.

Dataflow Modeling (2)

```
//Dataflow description of a 2-to-4-line decoder
module decoder_df (A,B,E,D);

input A,B,E;
output [0:3] D;

assign D[0] = ~ (~A & ~B & ~E) ,
          D[1] = ~ (~A & B & ~E) ,
          D[2] = ~ (A & ~B & ~E) ,
          D[3] = ~ (A & B & ~E) ;

endmodule
```

A 2-to-1 line multiplexer with data inputs A and B, select input S, and output Y is described with the continuous assignment

```
assign Y = (A & S) | (B & ~S)
```

Dataflow Modeling (3)

```
//Dataflow description of 4-bit adder
module binary_adder (A,B,Cin,SUM,Cout);
    input [3:0] A,B;
    input Cin;
    output [3:0] SUM;
    output Cout;
    assign {Cout,SUM} = A + B + Cin;
endmodule
```

```
//Dataflow description of a 4-bit comparator.
module magcomp (A,B,ALTB,AGTB,AEQB);
    input [3:0] A,B;
    output ALTB,AGTB,AEQB;
    assign ALTB = (A < B),
            AGTB = (A > B),
            AEQB = (A == B);
endmodule
```

Dataflow Modeling (4)

- ◆ The addition logic of 4 bit adder is described by a single statement using the operators of addition and concatenation.
- ◆ The plus symbol (+) specifies the binary addition of the 4 bits of **A** with the 4 bits of **B** and the one bit of **Cin**.
- ◆ The target output is the concatenation of the output carry **Cout** and the four bits of **SUM**.
- ◆ Concatenation of operands is expressed within braces and a comma separating the operands. Thus, **{Cout, SUM}** represents the 5-bit result of the addition operation.

Dataflow Modeling (5)

- ◆ Dataflow Modeling provides the means of describing combinational circuits by their function rather than by their gate structure.
- ◆ **Conditional operator (?:)**
condition ? true-expression : false-expression;
- ◆ A 2-to-1 line multiplexer
assign OUT = select ? A : B;

```
//Dataflow description of 2-to-1-line mux
module mux2x1_df (A,B,select,OUT);
    input A,B,select;
    output OUT;
    assign OUT = select ? A : B;
endmodule
```

Behavioral Modeling

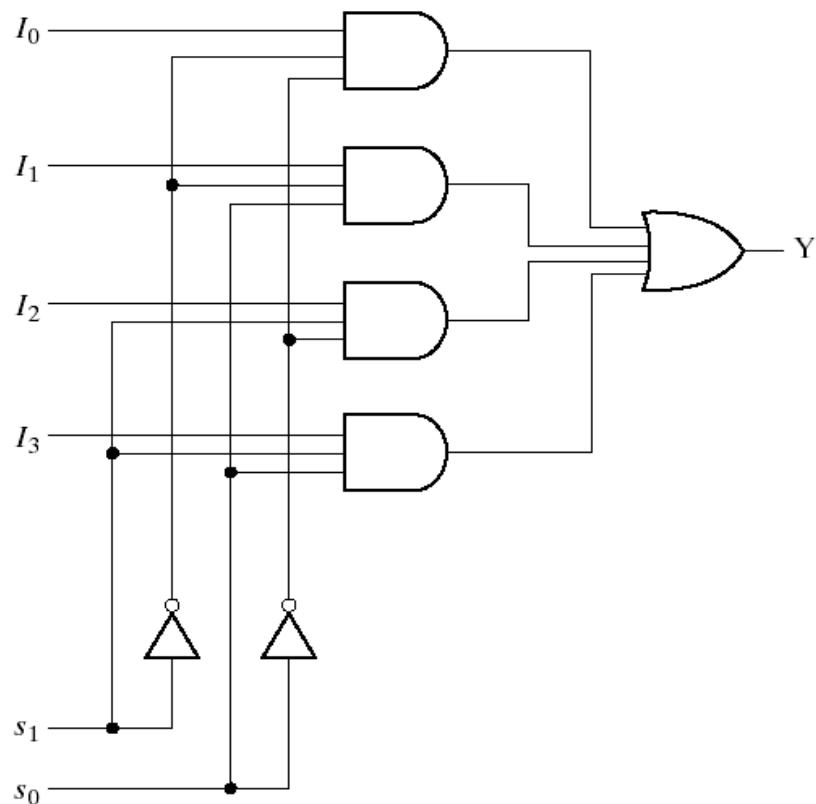
- ◆ Behavioral modeling represents digital circuits at a functional and algorithmic level.
- ◆ It is used mostly to describe sequential circuits, but can also be used to describe combinational circuits.
- ◆ Behavioral descriptions use the keyword **always** followed by a list of procedural assignment statements.
- ◆ The target output of procedural assignment statements must be of the **reg** data type.
- ◆ A **reg** data type retains its value until a new value is assigned.

Behavioral Modeling (2)

- ◆ The procedural assignment statements inside the **always** block are executed every time there is a change in any of the variable listed after the @ symbol. (Note that there is no “;” at the end of **always** statement)

```
//Behavioral description of 2-to-1-line multiplexer
module mux2x1_bh(A,B,select,OUT);
    input A,B,select;
    output OUT;
    reg OUT;
    always @(select or A or B)
        if (select == 1) OUT = A;
        else OUT = B;
endmodule
```

Behavioral Modeling (3)



(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

4-to-1 line
multiplexer

Behavioral Modeling (4)

```
//Behavioral description of 4-to-1 line mux

module mux4x1_bh (i0,i1,i2,i3,select,y);
    input i0,i1,i2,i3;
    input [1:0] select;
    output y;
    reg y;
    always @ (i0 or i1 or i2 or i3 or select)
        case (select)
            2'b00: y = i0;
            2'b01: y = i1;
            2'b10: y = i2;
            2'b11: y = i3;
        endcase
endmodule
```

Behavioral Modeling (5)

- ◆ In 4-to-1 line multiplexer, the select input is defined as a 2-bit vector and output y is declared as a reg data.
- ◆ The always block has a sequential block enclosed between the keywords **case** and **endcase**.
- ◆ The block is executed whenever any of the inputs listed after the @ symbol changes in value.

Writing a Test Bench

- ◆ A test bench is an HDL program used for applying stimulus to an HDL design in order to test it and observe its response during simulation.
- ◆ In addition to the **always** statement, test benches use the **initial** statement to provide a stimulus to the circuit under test.
- ◆ The **always** statement executes repeatedly in a loop. The initial statement executes only once starting from simulation time=0 and may continue with any operations that are delayed by a given number of units as specified by the symbol #.

Writing a Test Bench (2)

```
initial begin  
    A=0;  B=0;  #10  A=1;  #20  A=0;  B=1;  
end
```

- ◆ The block is enclosed between **begin** and **end**. At time=0, A and B are set to 0. 10 time units later, A is changed to 1. 20 time units later (at t=30) a is changed to 0 and B to 1.

Writing a Test Bench (2)

- ◆ Inputs to a 3-bit truth table can be generated with the initial block

```
initial begin  
    D = 3'b000; repeat (7); #10 D = D +  
    3'b001;  
end
```

- ◆ The 3-bit vector D is initialized to 000 at time=0. The keyword **repeat** specifies looping statement: one is added to D seven times, once every 10 time units.

Writing a Test-Bench (3)

- ◆ A stimulus module is an HDL program that has the following form.

module testname

*Declare local **reg** and **wire** identifiers*

Instantiate the design module under test.

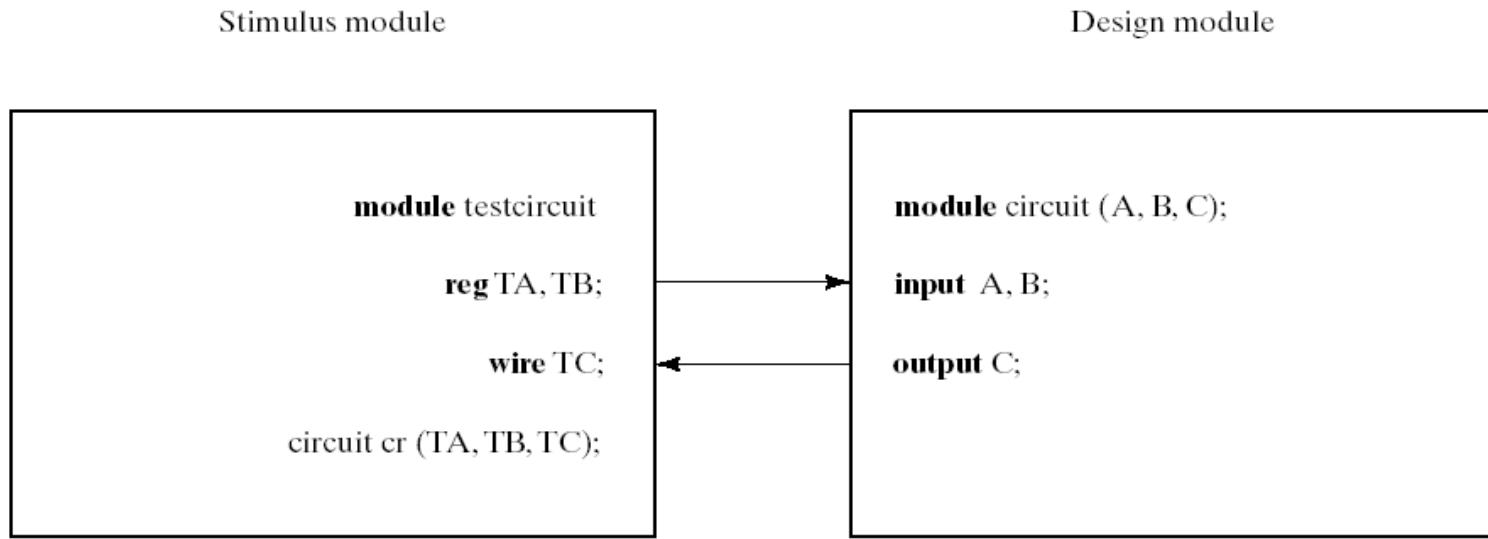
*Generate stimulus using **initial** and **always** statements*

Display the output response.

endmodule

- ◆ A test module typically has no inputs or outputs.
- ◆ The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local reg data type.
- ◆ The outputs of the design module that are displayed for testing are declared in the stimulus model as local wire data type.
- ◆ The module under test is then instantiated using the local identifiers.

Writing a Test-Bench (4)



The stimulus model generates inputs for the design module by declaring identifiers *TA* and *TB* as **reg** data type, and checks the output of the design unit with the **wire** identifier *TC*. The local identifiers are then used to instantiate the design module under test.

Writing a Test-Bench (5)

- ◆ The response to the stimulus generated by the **initial** and **always** blocks will appear at the output of the simulator as timing diagrams.
- ◆ It is also possible to display numerical outputs using Verilog *system tasks*.
 - **\$display** – display one-time value of variables or strings with end-of-line return,
 - **\$write** – same \$display but without going to next line.
 - **\$monitor** – display variables whenever a value changes during simulation run.
 - **\$time** – displays simulation time
 - **\$finish** – terminates the simulation
- ◆ The syntax for **\$display**, **\$write** and **\$monitor** is of the form
Task-name (format-specification, argument list);
E.g. **\$display(%d %b %b, C,A,B);**
\$display("time = %0d A = %b B=%b",\$time,A,B);

Writing a Test-Bench (6)

```
//Stimulus for mux2x1_df
module testmux;
    reg TA,TB,TS; //inputs for mux
    wire Y;          //output from mux
    mux2x1_df mx (TA,TB,TS,Y); // instantiate mux
    initial begin
        $monitor("select=%b A=%b B=%b OUT=%b", TS, TA, TB, Y);
        TS = 1; TA = 0; TB = 1;
        #10 TA = 1; TB = 0;
        #10 TS = 0;
        #10 TA = 0; TB = 1;
    end
endmodule
```

Writing a Test-Bench (7)

```
//Dataflow description of 2-to-1-line multiplexer
module mux2x1_df (A,B,select,OUT);

input A,B,select;
output OUT;

assign OUT = select ? A : B;

endmodule
```

Descriptions of Circuits

- ◆ ***Structural Description*** – This is directly equivalent to the schematic of a circuit and is specifically oriented to describing hardware structures using the components of a circuit.
- ◆ ***Dataflow Description*** – This describes a circuit in terms of function rather than structure and is made up of concurrent assignment statements or their equivalent. Concurrent assignments statements are executed concurrently, i.e. in parallel whenever one of the values on the right hand side of the statement changes.

Descriptions of Circuits (2)

- ◆ ***Hierarchical Description*** – Descriptions that represent circuits using hierarchy have multiple entities, one for each element of the Hierarchy.
- ◆ ***Behavioral Description*** – This refers to a description of a circuit at a level higher than the logic level. This type of description is also referred to as the *register transfers level*.

2-to-4 Line Decoder – Data flow description

```
//2-to-4 Line Decoder: Dataflow
module dec_2_to_4_df(E_n,A0,A1,D0_n,D1_n,D2_n,D3_n);
    input E_n, A0, A1;
    output D0_n,D1_n,D2_n,D3_n;
    assign D0_n=~ (~E_n&~A1&~A0);
    assign D1_n=~ (~E_n&~A1& A0);
    assign D2_n=~ (~E_n& A1&~A0);
    assign D3_n=~ (~E_n& A1& A0);
endmodule
```

4-to-1 Multiplexer

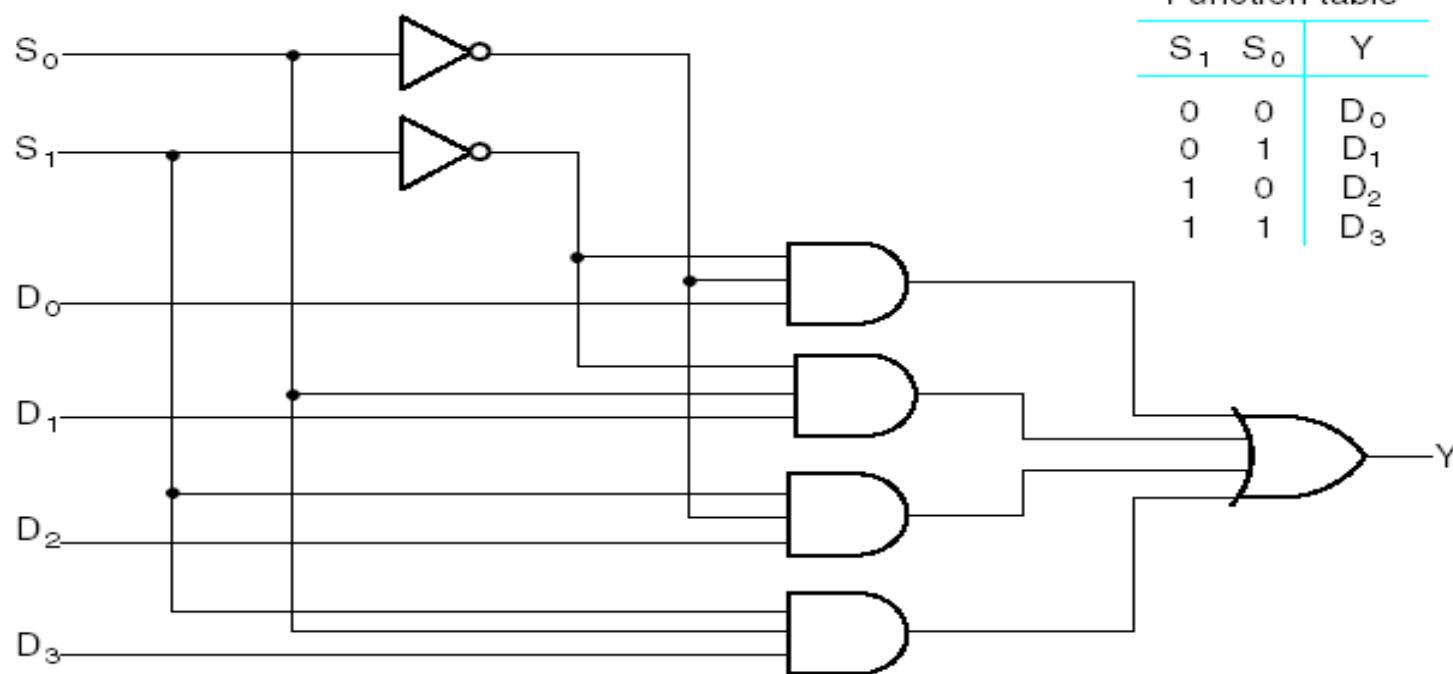


Fig. 3-19 4-to-1-Line Multiplexer

4-to-1 Multiplexer

```
//4-to-1 Mux: Structural Verilog
module mux_4_to_1_st_v(S,D,Y);
    input [1:0]S;
    input [3:0]D;
    output Y;
    wire [1:0]not_s;
    wire [0:3]N;
    not g0(not_s[0],S[0]),g1(not_s[1],S[1]);
    and g2(N[0],not_s[0],not_s[1],D[0]),
          g3(N[1],S[0],not_s[1],D[0]),
          g4(N[2],not_s[0],S[1],D[0]),
          g5(N[3],S[0],S[1],D[0]);
    or g5(Y,N[0],N[1],N[2],N[3]);
endmodule
```

4-to-1 Multiplexer – Data Flow

```
//4-to-1 Mux: Dataflow description
module mux_4_to_1(S,D,Y);
    input [1:0]S;
    input [3:0]D;
    output Y;
    assign Y = (~S[1]&~S[0]&D[0]) | (~S[1]&S[0]&D[1])
                | (S[1]&~S[0]&D[2]) | (S[1]&S[0]&D[3]);
endmodule
```

```
//4-to-1 Mux: Conditional Dataflow description
module mux_4_to_1(S,D,Y);
    input [1:0]S;
    input [3:0]D;
    output Y;
    assign Y = (S==2'b00)?D[0] : (S==2'b01)?D[1] :
                (S==2'b10)?D[2] : (S==2'b11)?D[3]:1'bx;;
endmodule
```

4-to-1 Multiplexer

```
//4-to-1 Mux: Dataflow Verilog Description
module mux_4_to_1(S,D,Y);
    input [1:0]S;
    input [3:0]D;
    output Y;
    assign Y=S[1]?(S[0]?D[3]:D[2]):(S[0]?D[1]:D[0]);
endmodule
```

Adder

□ TABLE 3-7
Truth Table of Half Adder

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 3-7 Truth Table of Half Adder

□ TABLE 3-8
Truth Table of Full Adder

Inputs			Outputs	
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

Table 3-8 Truth Table of Full Adder

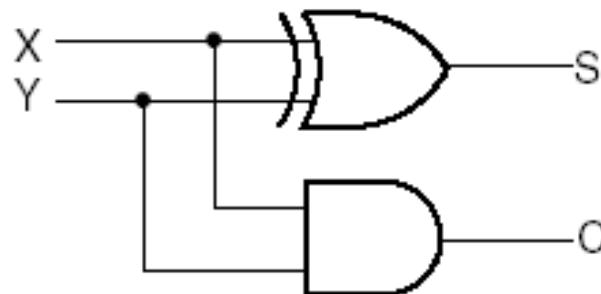
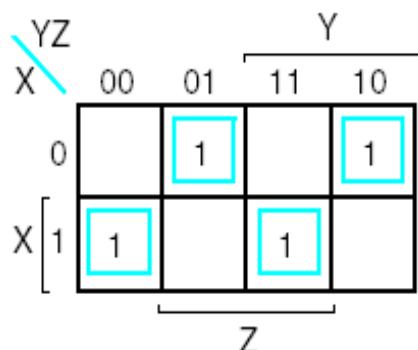
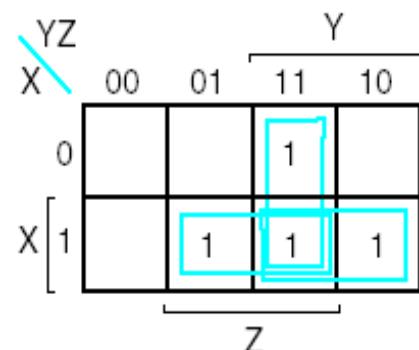


Fig. 3-25 Logic Diagram of Half Adder



$$\begin{aligned}S &= \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ \\&= X \oplus Y \oplus Z\end{aligned}$$



$$\begin{aligned}C &= XY + XZ + YZ \\&= XY + Z(X\bar{Y} + \bar{X}Y) \\&= XY + Z(X \oplus Y)\end{aligned}$$

Fig. 3-26 Maps for Full Adder

4-bit Adder

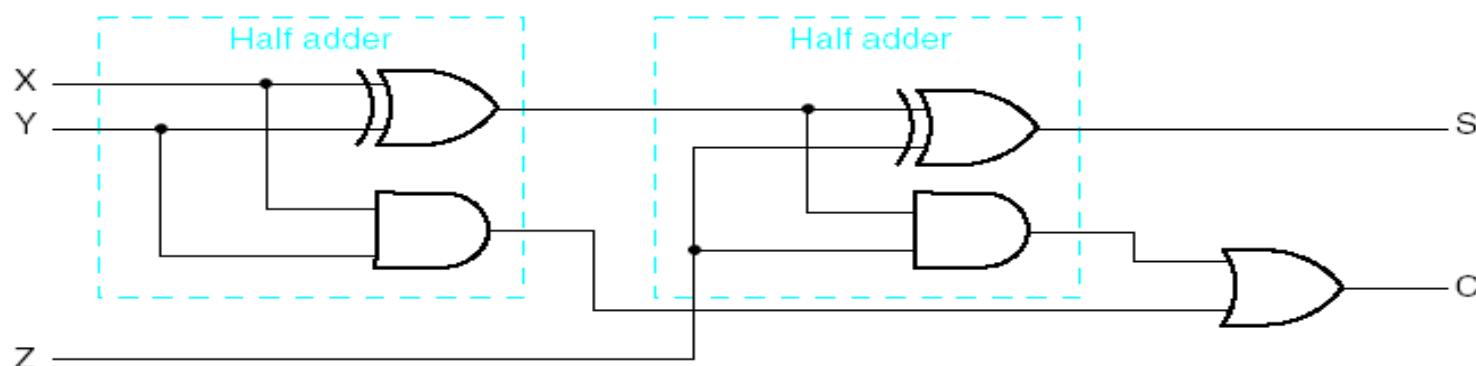


Fig. 3-27 Logic Diagram of Full Adder

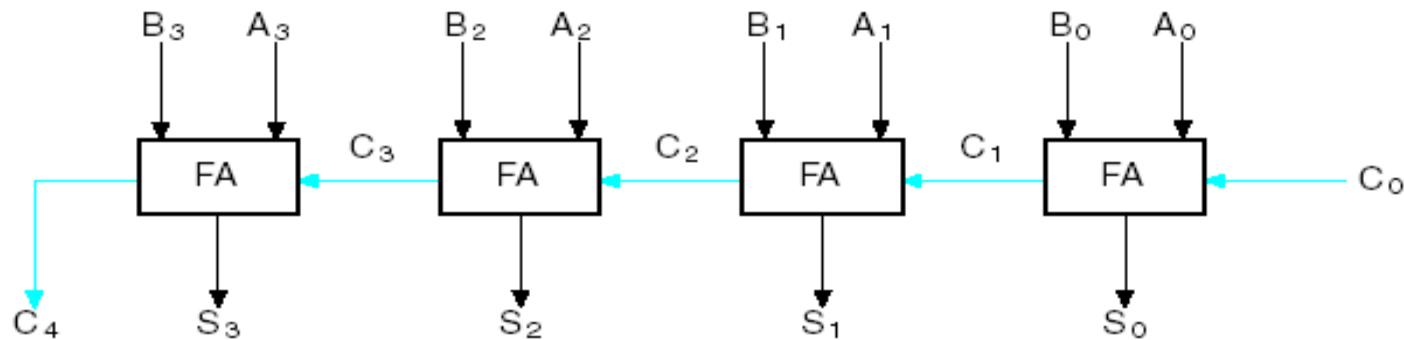


Fig. 3-28 4-Bit Ripple Carry Adder

4-bit-Adder

```
// 4-bit Adder: Hierarchical Dataflow/Structural
// (See Figures 3-27 and 3-28 for logic diagrams)

module half_adder_v(x, y, s, c);
    input x, y;
    output s, c;

    assign s = x ^ y;
    assign c = x & y;

endmodule

module full_adder_v(x, y, z, s, c);
    input x, y, z;
    output s, c;

    wire hs, hc, tc;

    half_adder_v HA1(x, y, hs, hc),
              HA2(hs, z, s, tc);
    assign c = tc | hc;

endmodule

module adder_4_v(B, A, C0, S, C4);
    input[3:0] B, A;
    input C0;
    output[3:0] S;
    output C4;

    wire[3:1] C;

    full_adder_v Bit0(B[0], A[0], C0, S[0], C[1]),
                  Bit1(B[1], A[1], C[1], S[1], C[2]),
                  Bit2(B[2], A[2], C[2], S[2], C[3]),
                  Bit3(B[3], A[3], C[3], S[3], C4);

endmodule
```

4-bit Adder

```
//4-bit adder : dataflow description
module adder_4bit (A,B,C0,S,C4);
    input [3:0] A,B;
    input C0;
    output [3:0] S;
    output C4;
    assign {C4,S} = A + B + C0;
endmodule
```

Sequential System Design

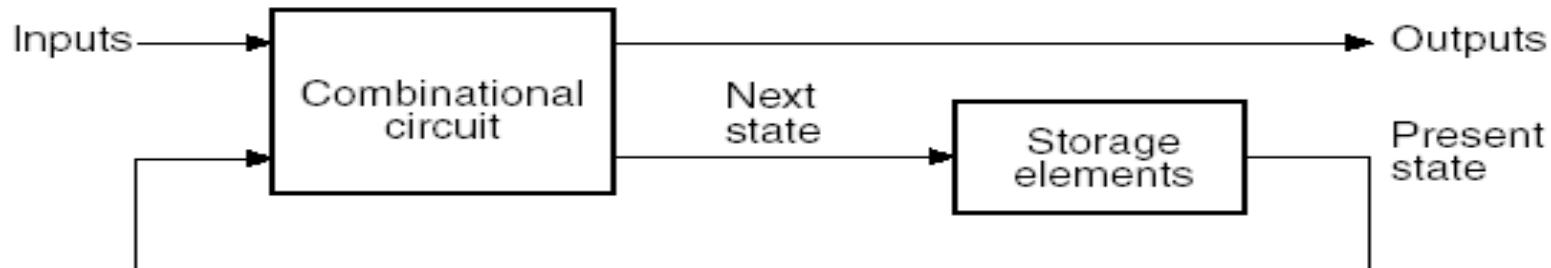
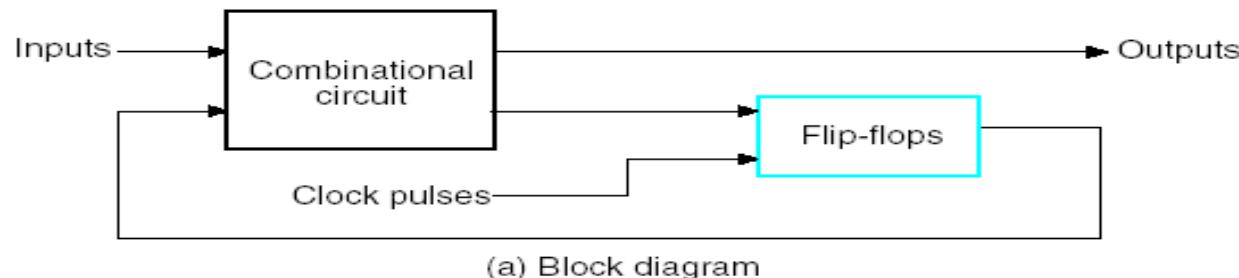


Fig. 4-1 Block Diagram of a Sequential Circuit



(b) Timing diagram of clock pulses

Fig. 4-3 Synchronous Clocked Sequential Circuit

Sequential System Design (2)

1. Obtain either the state diagram or the state table from the statement of the problem.
2. If only a state diagram is available from step 1, obtain state table.
3. Assign binary codes to the states.
4. Derive the flip-flop input equations from the next-state entries in the encoded state table.
5. Derive output equations from the output entries in the state table.
6. Simplify the flip-flop input and output equations.
7. Draw the logic diagram with D flip-flops and combinational gates, as specified by the flip-flop I/O equations.

Behavioral Modeling in SSD

- ◆ There are two kinds of behavioral statements in Verilog HDL: **initial** and **always**.
- ◆ The **initial** behavior executes once beginning at time=0.
- ◆ The **always** behavior executes repeatedly and re-executes until the simulation terminates.
- ◆ A behavior is declared within a module by using the keywords **initial** or **always**, followed by a statement or a block of statements enclosed by the keywords **begin** and **end**.

Behavioral Modeling in SSD (2)

- ◆ An example of a free-running clock

```
initial begin
    clock = 1'b0;
    repeat (30);
        #10 clock = ~clock;
end
```

```
initial begin
    clock = 1'b0;
    #300 $finish;
end
always #10 clock = ~clock
```

Behavioral Modeling in SSD (3)

- ◆ The always statement can be controlled by delays that wait for a certain time or by certain conditions to become true or by events to occur.
- ◆ This type of statement is of the form:
always @ (event control expression)
Procedural assignment statements
- ◆ The event control expression specifies the condition that must occur to activate the execution of the procedural assignment statements.
- ◆ The variables in the left-hand side of the procedural statements must be of the **reg** data type and must be declared as such.

Behavioral Modeling in SSD (4)

- ◆ The statements within the block, after the event control expression, execute sequentially and the execution suspends after the last statement has executed.
- ◆ Then the always statement waits again for an event to occur.
- ◆ Two kind of events:
 - *Level sensitive* (E.g. in combinational circuits and in latches)
always @ (A or B or Reset) will cause the execution of the procedural statements in the **always** block if changes occur in **A** or **B** or **Reset**.
 - *Edge-triggered* (In synchronous sequential circuits, changes in flip-flops must occur only in response to a transition of a clock pulse.
always @ (**posedge** clock or **negedge** reset) will cause the execution of the procedural statements only if the **clock** goes through a positive transition or if the **reset** goes through a negative transition.

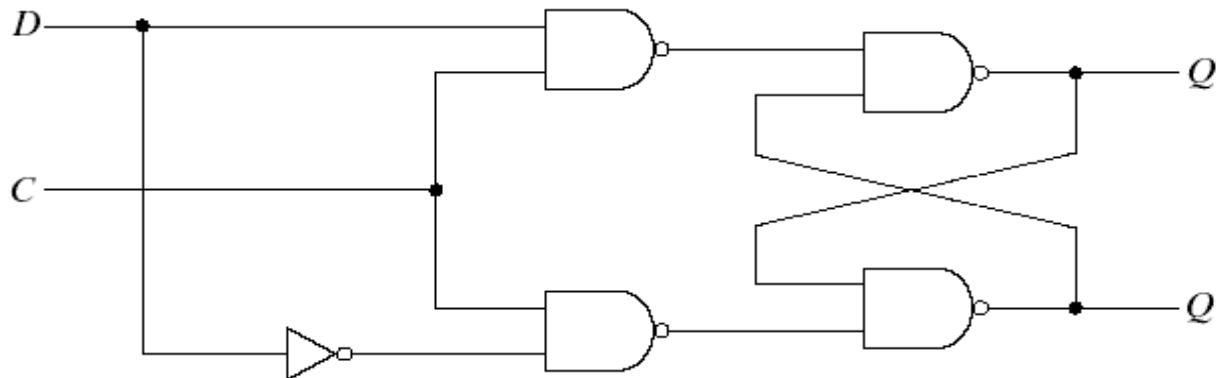
Behavioral Modeling in SSD (5)

- ◆ A procedural assignment is an assignment within an **initial** or **always** statement.
- ◆ There are two kinds of procedural assignments: *blocking* and *non-blocking*
 - **Blocking assignments** (executed sequentially in the order they are listed in a sequential block)
 - $B = A$
 - $C = B + 1$
 - **Non-blocking assignments** (evaluate the expressions on the right hand side, but do not make the assignment to the left hand side until all expressions are evaluated.)
 - $B <= A$
 - $C <= B + 1$

Flip-Flops and Latches

- ◆ The D-latch is transparent and responds to a change in data input with a change in output as long as control input is enabled.
- ◆ It has two inputs, D and control, and one output Q. Since Q is evaluated in a procedural statement it must be declared as **reg** type.
- ◆ Latches respond to input signals so the two inputs are listed without edge qualifiers in the event control expression following the @ symbol in the **always** statement.
- ◆ There is one blocking procedural assignment statement and it specifies the transfer of input D to output Q if control is true.

Flip-Flops and Latches



(a) Logic diagram

C	D	Next state of Q
0	X	No change
1	0	$Q = 0$; Reset state
1	1	$Q = 1$; Set state

(b) Function table

```
module D_latch(Q,D,control);
    output Q;
    input D,control;
    reg Q;
    always @ (control or D)
        if(control) Q = D; //Same as: if(control=1)
endmodule
```

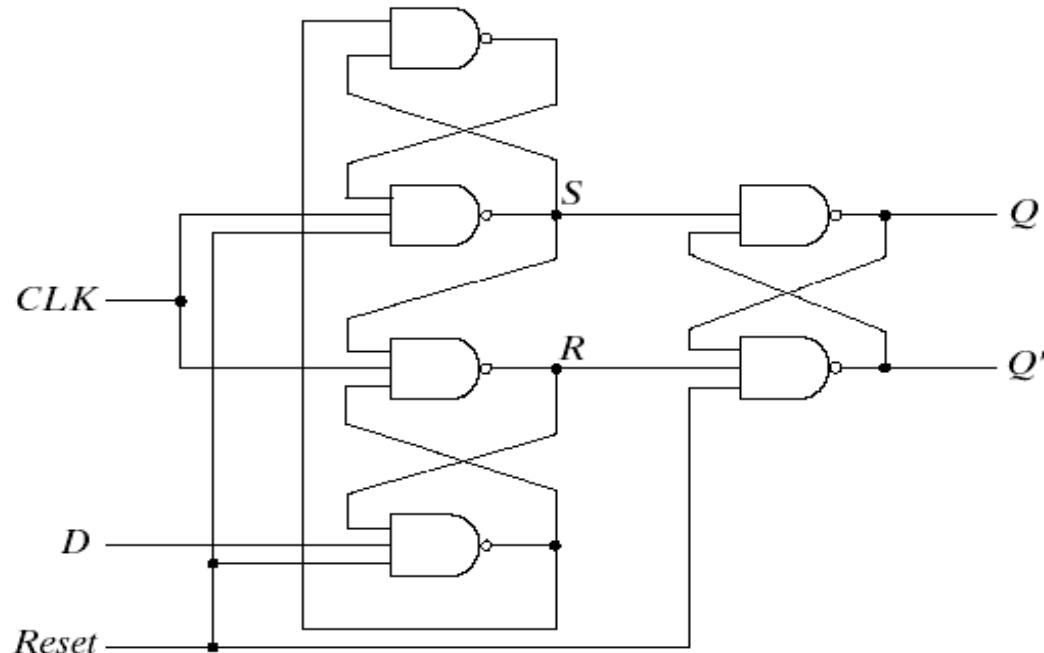
Flip-Flops and Latches

```
//D flip-flop  
module D_FF (Q,D,CLK);  
    output Q;  
    input D,CLK;  
    reg Q;  
    always @ (posedge CLK)  
        Q = D;  
    endmodule
```

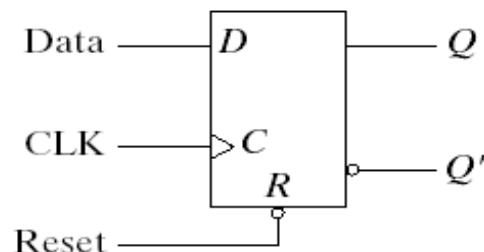
```
//D flip-flop with asynchronous reset.  
module DFF (Q,D,CLK,RST);  
    output Q;  
    input D,CLK,RST;  
    reg Q;  
    always @ (posedge CLK or negedge RST)  
        if (~RST) Q = 1'b0; // Same as: if (RST = 0)  
        else Q = D;  
    endmodule
```

D Flip-Flop with Reset

D Flip-Flop with Asynchronous Reset



(a) Circuit diagram

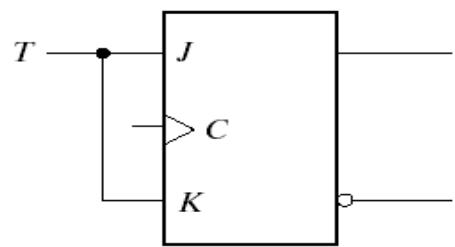


(b) Graphic symbol

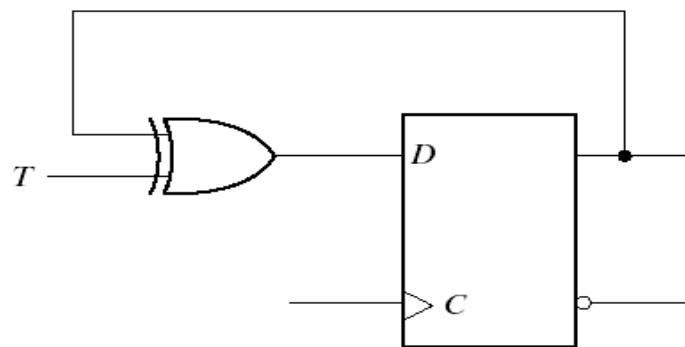
R	C	D	Q	Q'
0	X	X	0	1
1	↑	0	0	1
1	↑	1	1	0

(b) Function table

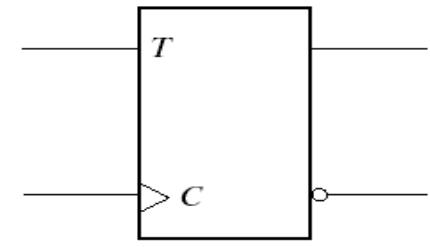
T & J-K Flip-Flops



(a) From *JK* flip-flop



(b) From *D* flip-flop



(c) Graphic symbol

Fig. 5-13 T Flip-Flop

T & J-K Flip-Flops

```
//T flip-flop from D flip-flop and gates  
module TFF (Q,T,CLK,RST);  
    output Q;  
    input T,CLK,RST;  
    wire DT;  
    assign DT = Q ^ T ;  
//Instantiate the D flip-flop  
    DFF TF1 (Q,DT,CLK,RST);  
endmodule
```

```
//JK flip-flop from D flip-flop and gates  
module JKFF (Q,J,K,CLK,RST);  
    output Q;  
    input J,K,CLK,RST;  
    wire JK;  
    assign JK = (J & ~Q) | (~K & Q);  
//Instantiate D flipflop  
    DFF JK1 (Q,JK,CLK,RST);  
endmodule
```

Characteristic equations of the flip-flops:

$$Q(t+1) = Q \oplus T \quad \text{for a T flip - flop}$$

$$Q(t+1) = JQ' + K'Q \quad \text{for a JK flip - flop}$$

J-K Flip-Flop

```
// Functional description of JK
// flip-flop
module JK_FF (J,K,CLK,Q,Qnot);
    output Q,Qnot;
    input  J,K,CLK;
    reg   Q;
    assign Qnot = ~ Q ;
    always @(posedge CLK)
        case ({J,K})
            2'b00: Q = Q;
            2'b01: Q = 1'b0;
            2'b10: Q = 1'b1;
            2'b11: Q = ~ Q;
        endcase
endmodule
```

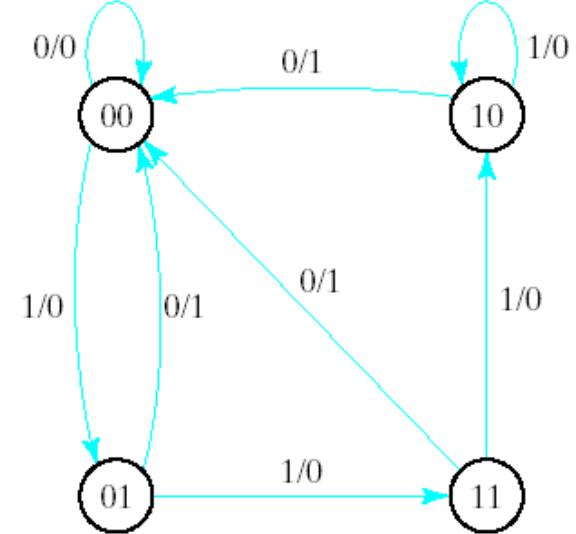
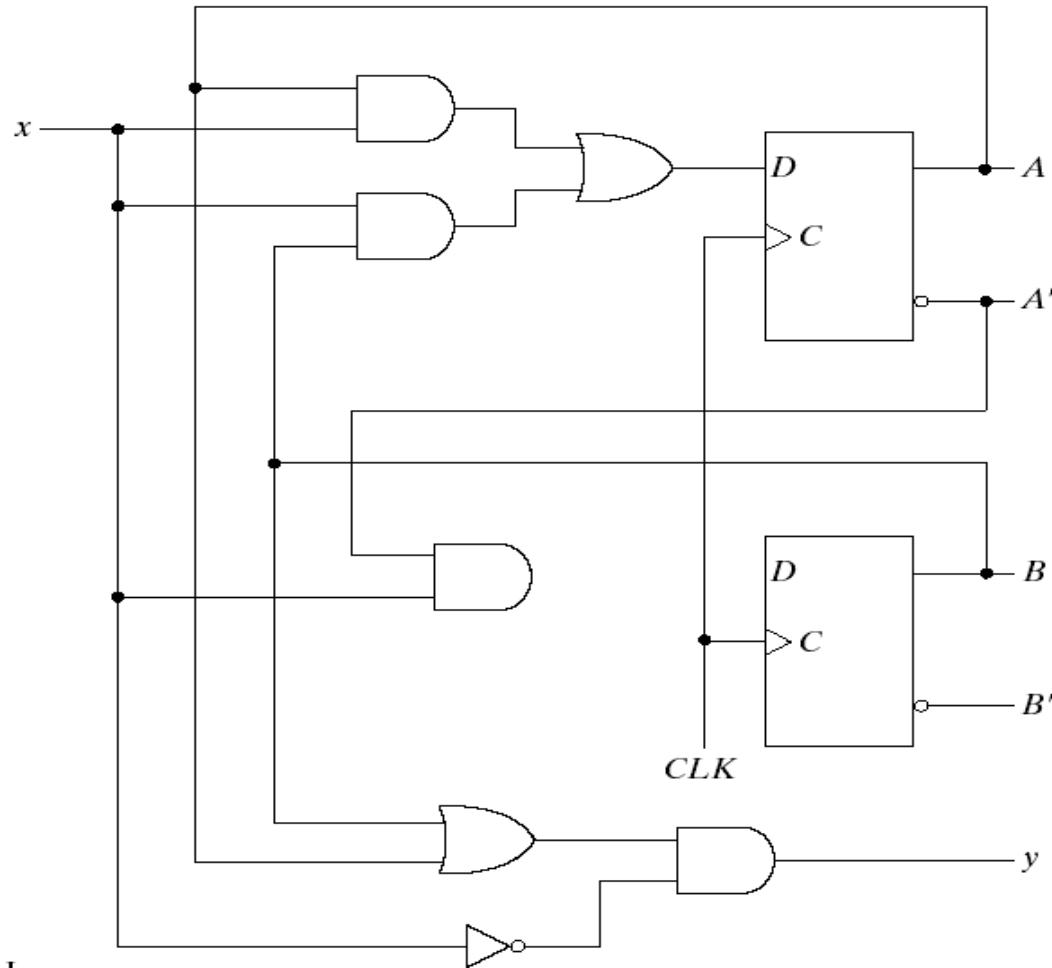
- Here the flip-flop is described using the characteristic table rather than the characteristic equation.
- The **case** multiway branch condition checks the 2-bit number obtained by concatenating the bits of J and K.
- The **case** value ({J,K}) is evaluated and compared with the values in the list of statements that follow.

D-Flip-Flop

```
//Positive Edge triggered DFF with Reset
module DFF(CLK,RST,D,Q);
    input CLK,RST,D;
    output Q;
    reg Q;

    always@(posedge CLK or posedge RST)
        if (RST) Q<=0;
        else     Q<=D;
endmodule
```

Sequential Circuit



Sequential Circuit (2)

```
//Mealy state diagram for the circuit
module Mealy_mdl (x,y,CLK,RST);
    input x,CLK,RST;
    output y;
    reg y;
    reg [1:0] Prstate,Nxtstate;
    parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
    always@(posedge CLK or negedge RST)
        if (~RST) Prstate = S0; //Initialize to
state S0
        else Prstate = Nxtstate; //Clock operations
```

Sequential Circuit (3)

```
always @(Prstate or x)      //Determine next state
  case (Prstate)
    S0: if (x) Nxtstate = S1;
    S1: if (x) Nxtstate = S3;
          else Nxtstate = S0;
    S2: if (~x) Nxtstate = S0;
    S3: if (x) Nxtstate = S2;
          else Nxtstate = S0;
  endcase
always @(Prstate or x)      //Evaluate output
  case (Prstate)
    S0: y = 0;
    S1: if (x) y = 1'b0; else y = 1'b1;
    S2: if (x) y = 1'b0; else y = 1'b1;
    S3: if (x) y = 1'b0; else y = 1'b1;
  endcase
endmodule
```

Sequential Circuit (4)

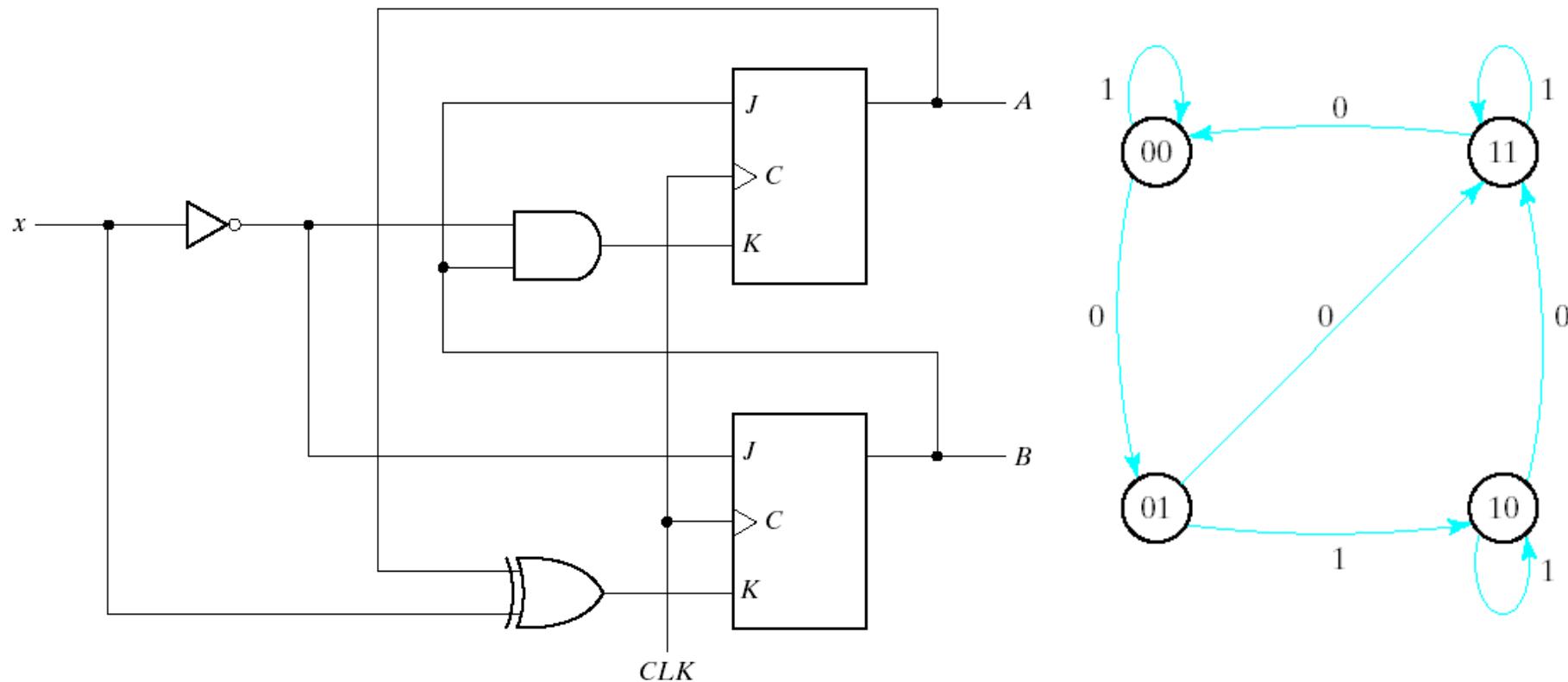
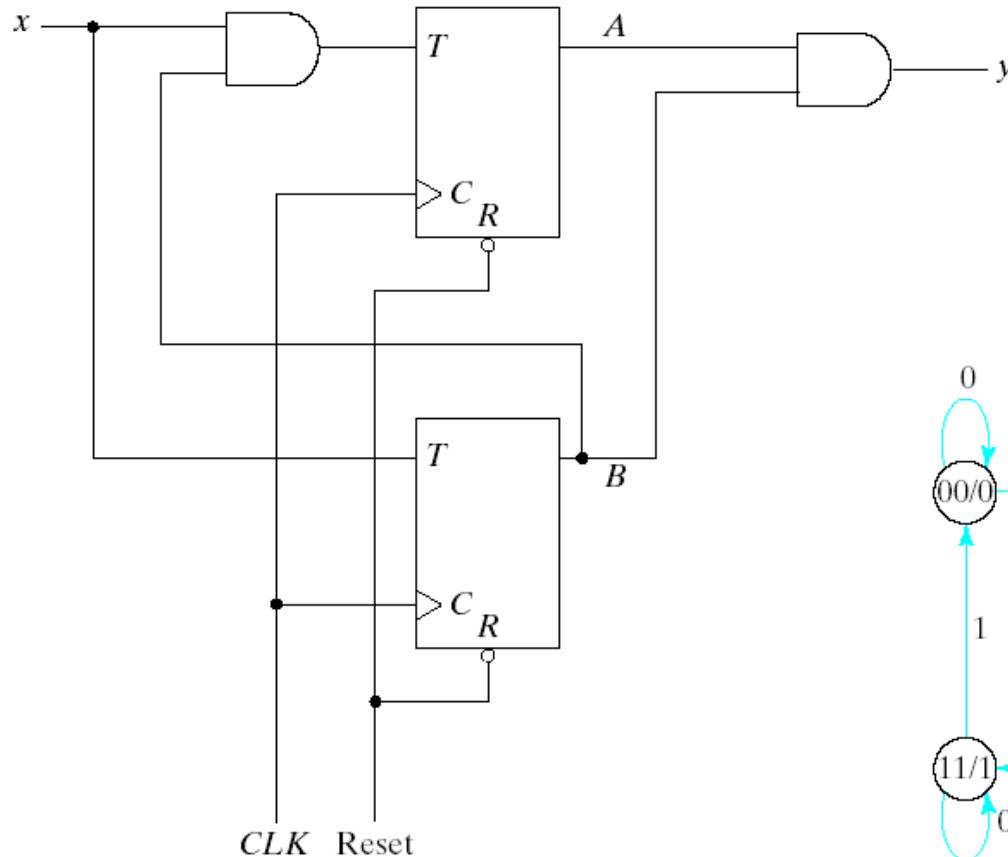


Fig. 5-18 Sequential Circuit with JK Flip-Flop

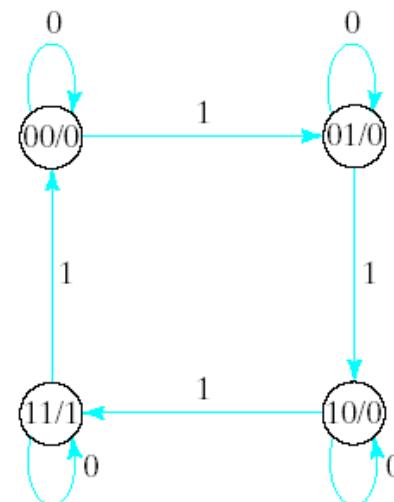
Sequential Circuit (5)

```
//Moore state diagram (Fig. 5-19)
module Moore_mdl (x,AB,CLK,RST);
    input x,CLK,RST;
    output [1:0]AB;
    reg [1:0] state;
    parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
    always @(posedge CLK or negedge RST)
        if (~RST) state = S0; //Initialize to state S0
        else
            case(state)
                S0: if (~x) state = S1;
                S1: if (x) state = S2; else state = S3;
                S2: if (~x) state = S3;
                S3: if (~x) state = S0;
            endcase
            assign AB = state;           //Output of flip-flops
    endmodule
```

Sequential Circuit (6)



(a) Circuit diagram



(b) State diagram

Fig. 5-20 Sequential Circuit with T Flip-Flops

Sequential Circuit (7)

```
//Structural description of sequential circuit
//See Fig. 5-20(a)

module Tcircuit (x,y,A,B,CLK,RST);

    input x,CLK,RST;
    output y,A,B;
    wire TA,TB;

//Flip-flop input equations

    assign TB = x,
            TA = x & B;

//Output equation

    assign y = A & B;

//Instantiate T flip-flops

    T_FF BF (B,TB,CLK,RST);
    T_FF AF (A,TA,CLK,RST);

endmodule
```

Sequential Circuit (8)

```
//T flip-flop
module T_FF (Q,T,CLK,RST);
  output Q;
  input T,CLK,RST;
  reg Q;
  always@(posedge CLK or
negedge RST)
    if(~RST) Q=1'b0;
    else Q=Q^T;
endmodule
```

```
//Stimulus for testing seq. cir
module testTcircuit;
  reg x,CLK,RST; //inputs for
                        circuit
  wire y,A,B; //output from
                        circuit
  Tcircuit TC(x,y,A,B,CLK,RST);
  initial begin
    RST = 0; CLK = 0;
    #5 RST = 1;
    repeat (16)
      #5 CLK = ~CLK;
    end
    initial begin
      x = 0; #15 x = 1;
      repeat (8)
        #10 x = ~ x;
    end
endmodule
```

Sequence Recognizer

Identify the sequence 1101, regardless of where it occurs in a longer sequence.

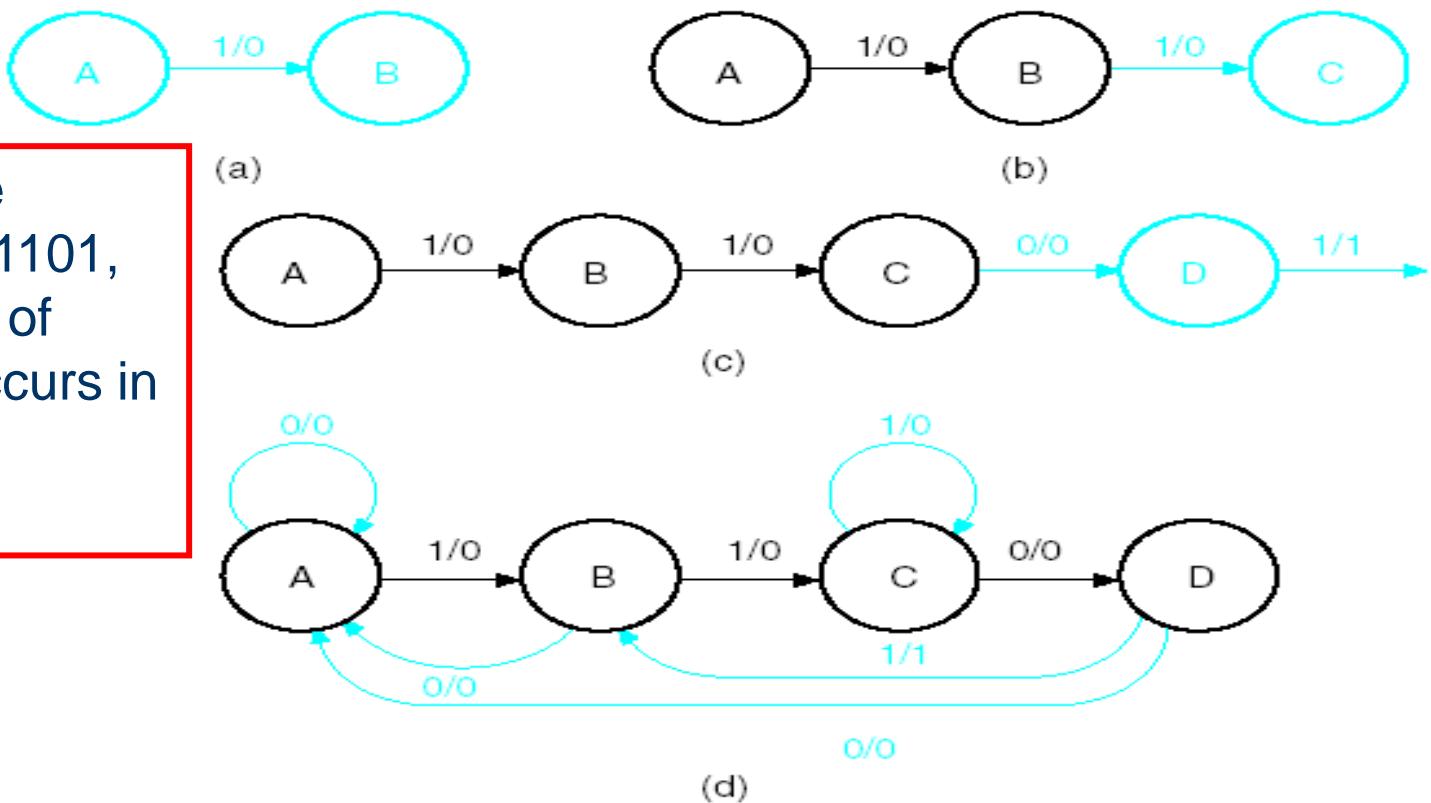


Fig. 4-21 Construction of a State Diagram

Sequence Recognizer (2)

□ TABLE 4-5
State Table for State Diagram in Figure 4-21

Present State	Next State		Output Z	
	X = 0	X = 1	X = 0	X = 1
A	A	B	0	0
B	A	C	0	0
C	D	C	0	0
D	A	B	0	1

Table 4-5 State Table for State Diagram in Figure 4-21

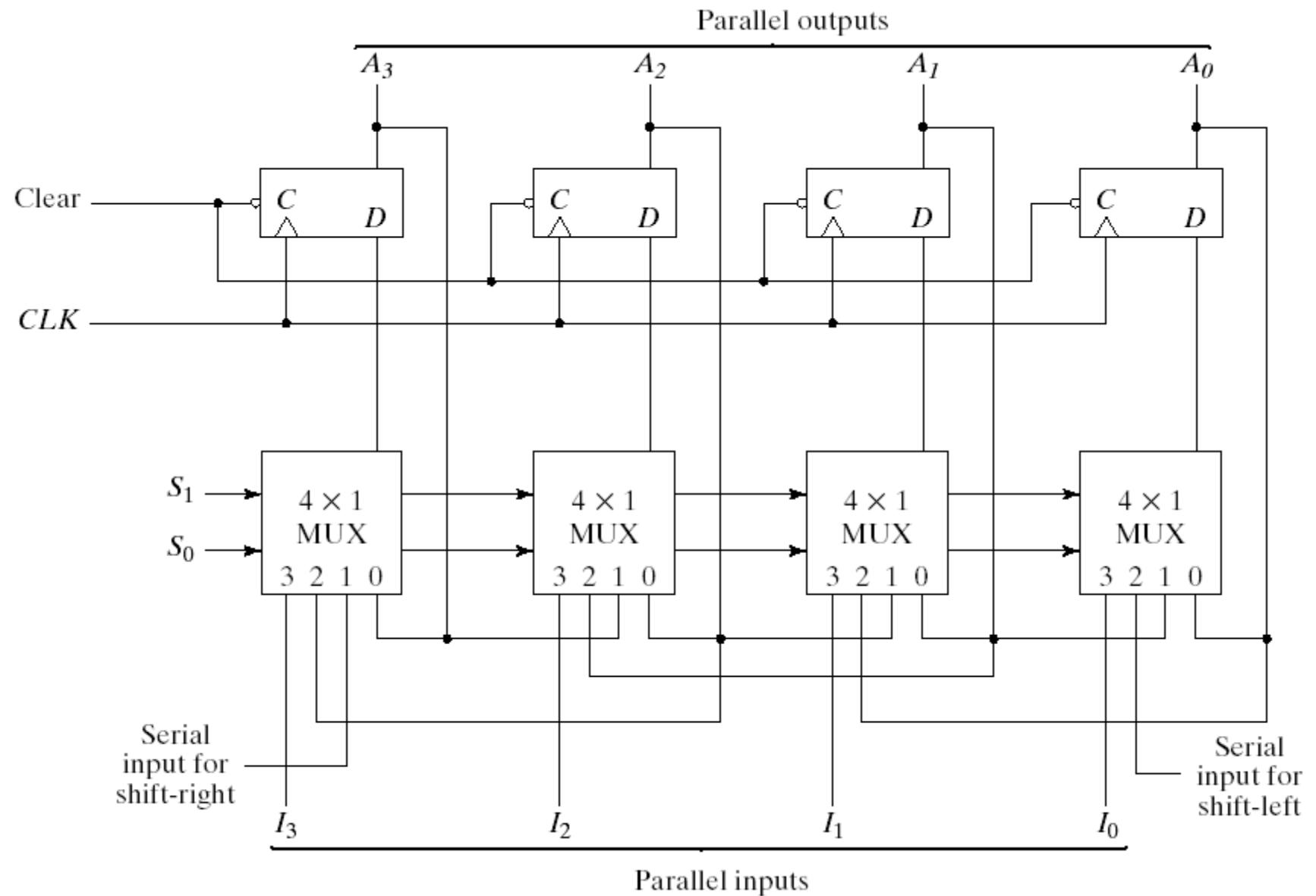
Sequence Recognizer

```
module seq_recognizer(CLK,RST,X,Z);
    input CLK,RST,X;
    output Z;
    reg [1:0]state, next_state;
    parameter A=2'b00,B=2'b01,C=2'b10,D=2'b11;
    reg Z;
    always@ (posedge CLK or posedge RST) begin
        if(RST==1) state <= A;
        else state <= next_state;
    end
endmodule
```

```
always@(X or state) begin
    case(state)
        A:if(X) next_state <= B; else next_state <= A;
        B:if(X) next_state <= C; else next_state <= A;
        C:if(X) next_state <= C; else next_state <= D;
        D:if(X) next_state <= B; else next_state <= A;
    endcase
end
always@(X or state) begin
    case(state)
        A:Z<=0;
        B:Z<=0;
        C:Z<=0;
        D:Z<=X?1:0;
    endcase
end
endmodule
```

HDL for Registers and Counters

- ◆ Registers and counters can be described in HDL at either the behavioral or the structural level.
- ◆ In the behavioral, the register is specified by a description of the various operations that it performs similar to a function table.
- ◆ A structural level description shows the circuit in terms of a collection of components such as gates, flip-flops and multiplexers.
- ◆ The various components are instantiated to form a hierarchical description of the design similar to a representation of a logic diagram.



HDL for Registers and Counters (3)

Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	+ve	1	X	Load inputs
1	+ve	0	1	Count next binary state
1	+ve	0	0	No change

4-bit binary counter with parallel load

Mode Control		Register Operation
S_1	S_0	Register Operation
0	0	No Change
0	1	Shift Right
1	0	Shift Left
1	1	Parallel Load

Function table for 4-bit Universal Shift Register

HDL for Registers and Counters (4)

```
//Behavioral description of Universal shift register
module shftreg (s1,s0,Pin,lfin,rtin,A,CLK,Clr);
    input s1,s0;          //Select inputs
    input lfin, rtin;    //Serial inputs
    input CLK,Clr;      //Clock and Clear
    input [3:0] Pin;     //Parallel input
    output [3:0] A;      //Register output
    reg [3:0] A;

always @ (posedge CLK or negedge Clr)
    if (~Clr) A = 4'b0000;
    else
        case ({s1,s0})
            2'b00: A = A;           //No change
            2'b01: A = {rtin,A[3:1]}; //Shift right
            2'b10: A = {A[2:0],lfin}; //Shift left
            //Parallel load input
            2'b11: A = Pin;
        endcase
endmodule
```

HDL for Registers and Counters (5)

```
//Structural description of Universal shift register
module SHFTREG (I,select,lfin,rtin,A,CLK,Clr);
    input [3:0] I;                      //Parallel input
    input [1:0] select;                 //Mode select
    input lfin,rtin,CLK,Clr;          //Serial input,clock,clear
    output [3:0] A;                   //Parallel output

//Instantiate the four stages
    stage ST0 (A[0],A[1],lfin,I[0],A[0],select,CLK,Clr);
    stage ST1 (A[1],A[2],A[0],I[1],A[1],select,CLK,Clr);
    stage ST2 (A[2],A[3],A[1],I[2],A[2],select,CLK,Clr);
    stage ST3 (A[3],rtin,A[2],I[3],A[3],select,CLK,Clr);
endmodule
```

HDL for Registers and Counters (6)

```
//One stage of shift register

module stage(i0,i1,i2,i3,Q,select,CLK,Clr);

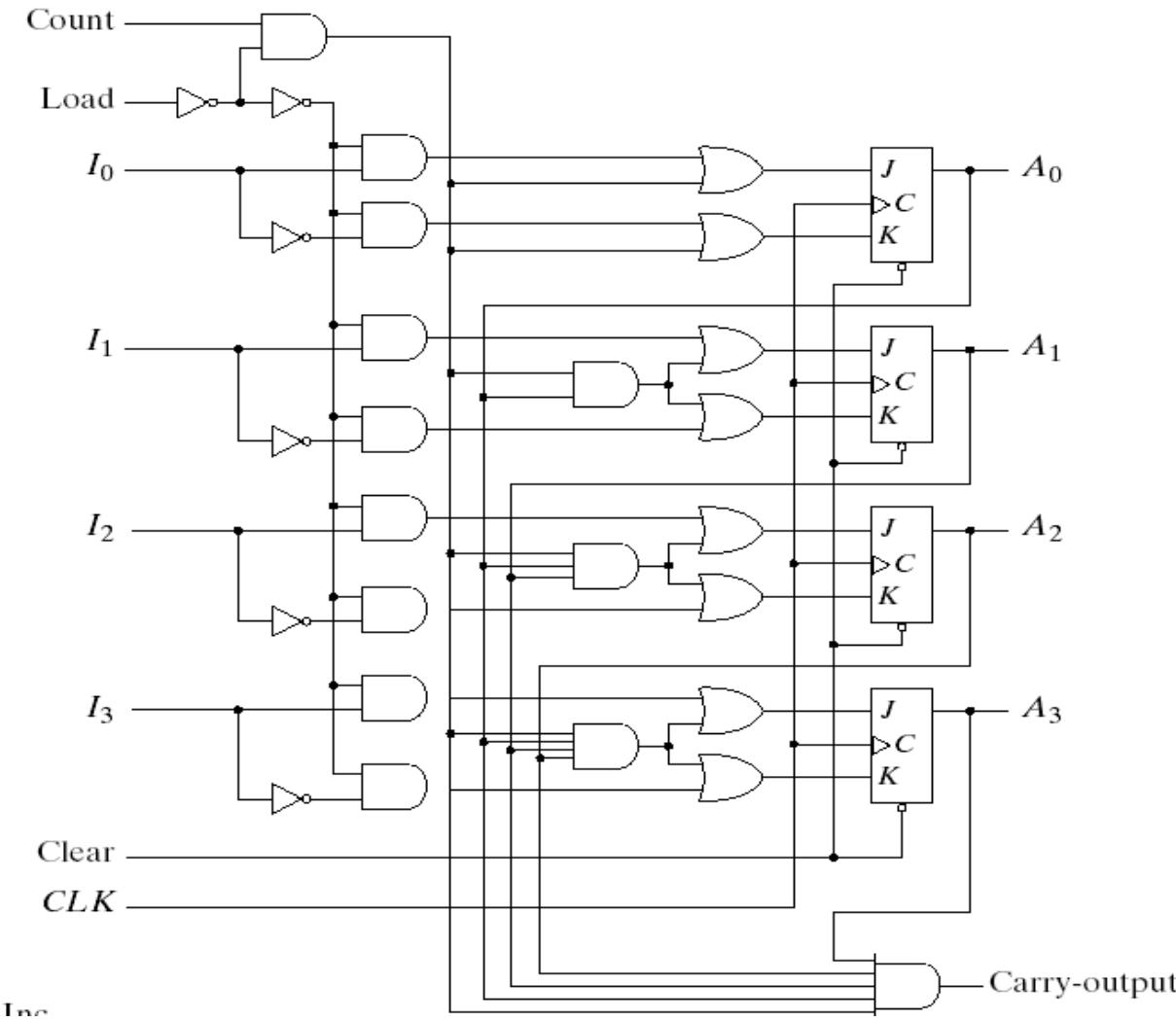
input i0,i1,i2,i3,CLK,Clr;
input [1:0] select;
output Q;
reg Q,D;

//4x1 multiplexer

always @ (i0 or i1 or i2 or i3 or select)
case (select)
    2'b00: D = i0;
    2'b01: D = i1;
    2'b10: D = i2;
    2'b11: D = i3;
endcase

//D flip-flop
always@(posedge CLK or negedge Clr)
    if (~Clr) Q = 1'b0;
    else Q = D;
endmodule
```

HDL for Registers and Counters (7)



HDL for Registers and Counters (8)

```
//Binary counter with parallel load

module counter (Count,Load,IN,CLK,Clr,A,CO);

input Count,Load,CLK,Cr;

input [3:0] IN;           //Data input

output CO;              //Output carry

output [3:0] A;          //Data output

reg [3:0] A;

assign CO = Count & ~Load & (A == 4'b1111);

always @(posedge CLK or negedge Clr)

    if (~Clr) A = 4'b0000;

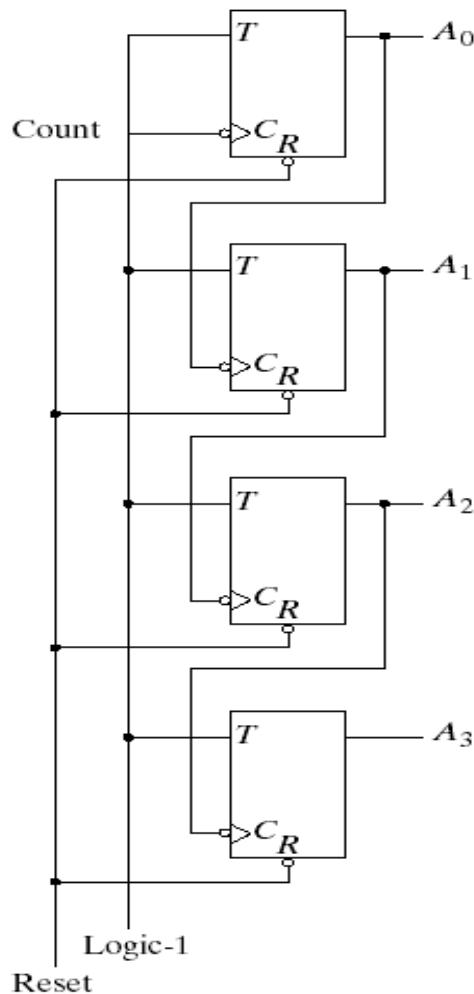
    else if (Load) A = IN;

    else if (Count) A = A + 1'b1;

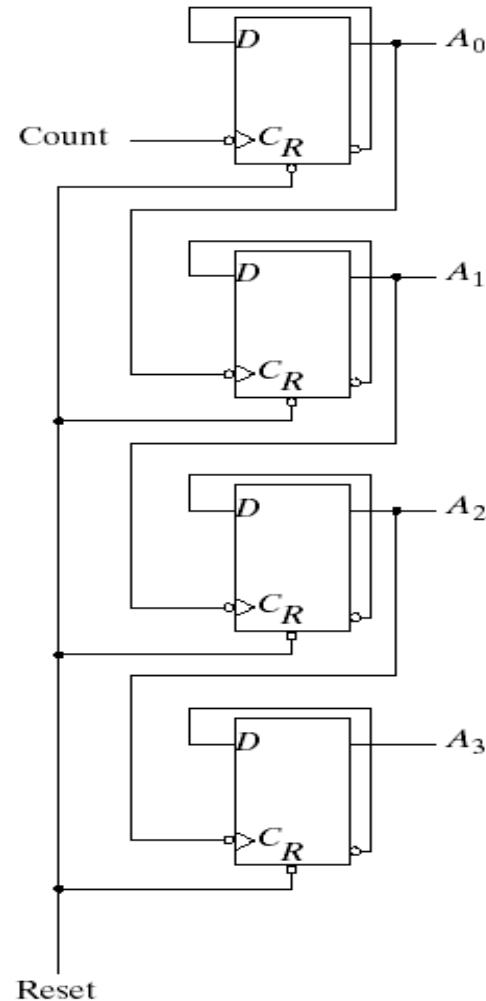
    else A = A;           // no change, default condition

endmodule
```

HDL for Registers and Counters (9)



nc. (a) With T flip-flops



(b) With D flip-flops

4-bit Binary
Ripple
Counter

HDL for Registers and Counters (10)

```
//Ripple counter
module ripplecounter(A0,A1,A2,A3,Count,Reset);
    output A0,A1,A2,A3;
    input Count,Reset;
//Instantiate complementing flip-flop
    CF F0 (A0,Count,Reset);
    CF F1 (A1,A0,Reset);
    CF F2 (A2,A1,Reset);
    CF F3 (A3,A2,Reset);
endmodule
```

```
//Complementing flip-flop with delay
//Input to D flip-flop = Q'
module CF (Q,CLK,Reset);
    output Q;
    input CLK,Reset;
    reg Q;
always@ (negedge CLK or posedge Reset)
    if(Reset) Q=1'b0;
    else Q=#2 (~Q); //Delay of 2 time
units
endmodule
```

HDL for Registers and Counters (11)

```
//Stimulus for testing ripple counter
module testcounter;
    reg Count;
    reg Reset;
    wire A0,A1,A2,A3;
//Instantiate ripple counter
    ripplecounter RC (A0,A1,A2,A3,Count,Reset);
always
    #5 Count = ~Count;
initial
begin
    Count = 1'b0;
    Reset = 1'b1;
    #4 Reset = 1'b0;
    #165 $finish;
end
endmodule
```



Some arithmetic fundamentals

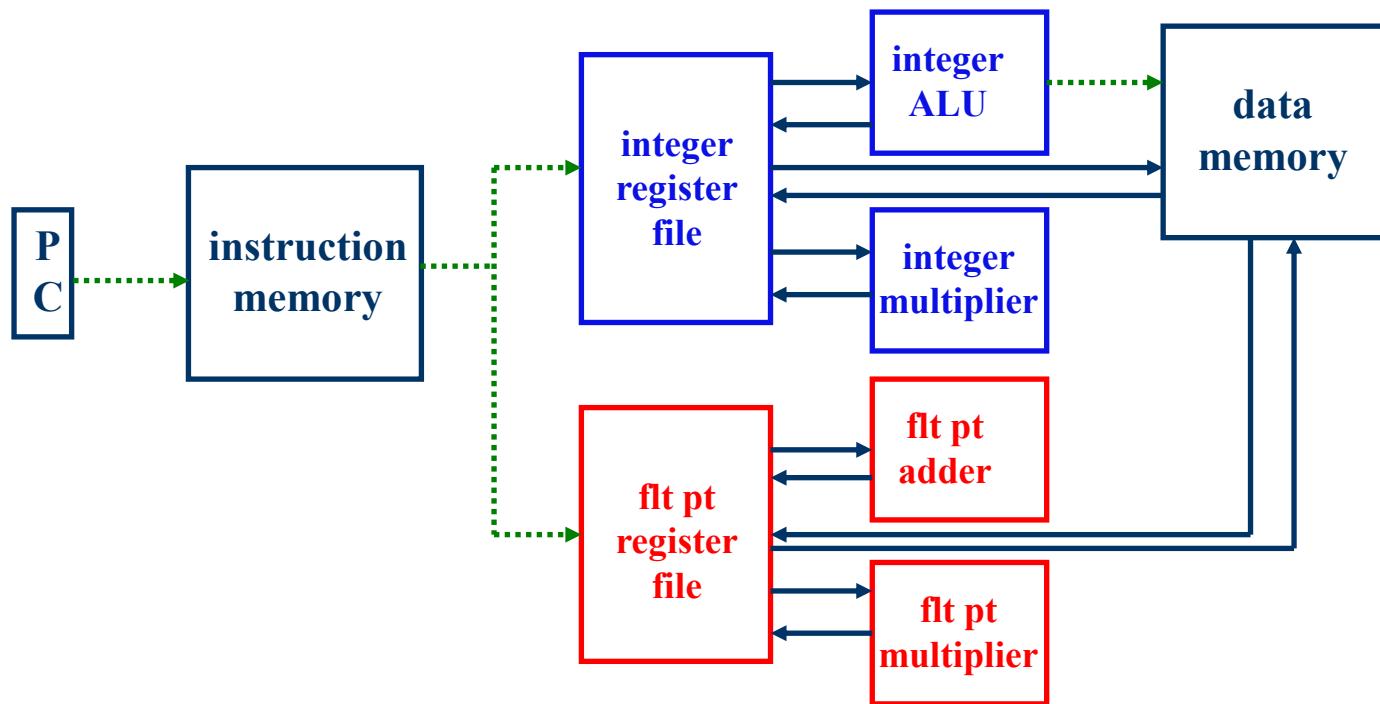
- ◆ Unsigned numbers: values are always positive
 - Example: $1000\ 1010_2 = 2^7 + 2^3 + 2^1 = 146_{10}$
- ◆ Signed numbers: two's complement notation
 - Example: $1000\ 1010_2 = -2^7 + 2^3 + 2^1 = -118_{10}$
 - Leftmost bit called the *sign bit*
 - Positive numbers have a sign bit of 0, negative numbers a sign bit of 1
- ◆ Sign extending a number: replicate the most significant bit the number of times needed
 - Example: $1000\ 1010_2$ is the same as $1111\ 1111\ 1000\ 1010_2$

Some arithmetic fundamentals

- ◆ Negating a two's complement number: invert (NOT) all bits and add 1
 - Example: negative of $1000\ 1010 = 0111\ 0101 + 1 = 0111\ 0110 = 118$
- ◆ Logical operations
 - AND, OR, NOR, XOR perform logic function on a bit-by-bit basis
 - Example: $1000\ 1010 \text{ AND } 1101\ 0110 = 1000\ 0010$
 - Also arithmetic/logical shift left/right

Integer and floating point computation

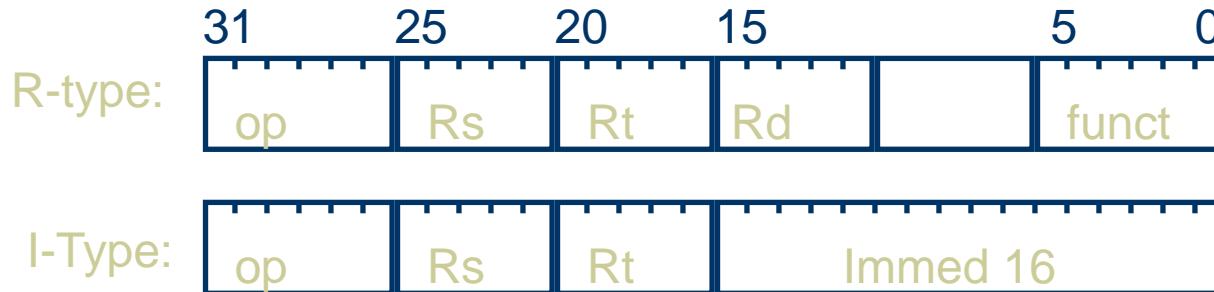
- ◆ Most general-purpose ISAs specify separate integer and floating point register files
 - Operand representation formats differ
 - Computation hardware differs
- ◆ Result is a split of the *execution core* into integer and floating point sections



MIPS ALU requirements

- ◆ add, addu, sub, subu, addi, addiu
 - => 2's complement adder/sub with overflow detection
- ◆ and, or, andi, oru, xor, xori, nor
 - => Logical AND, logical OR, XOR, nor
- ◆ SLTI, SLTIU (set less than)
 - => 2's complement adder with inverter, check sign bit of result
- ◆ ALU from CS 150 / P&H book chapter 4 supports these ops

MIPS arithmetic instruction format



Type	op	funct
ADDI	10	xx
ADDIU	11	xx
SLTI	12	xx
SLTIU	13	xx
ANDI	14	xx
ORI	15	xx
XORI	16	xx
LUI	17	xx

Type	op	funct
ADD	00	40
ADDU	00	41
SUB	00	42
SUBU	00	43
AND	00	44
OR	00	45
XOR	00	46
NOR	00	47

Type	op	funct
	00	50
	00	51
SLT	00	52
SLTU	00	53

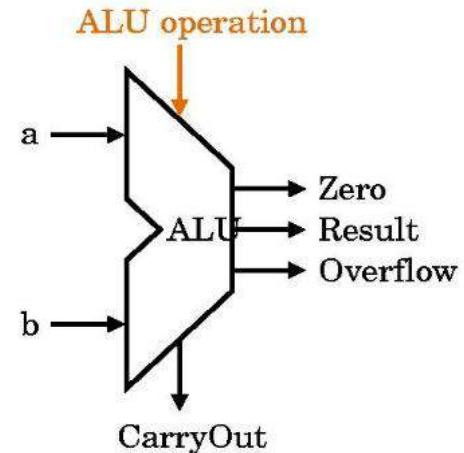
- ◆ Signed arithmetic generate overflow, no carry

Designing an integer ALU for MIPS

- ◆ ALU = Arithmetic Logic Unit
- ◆ Performs single cycle execution of simple integer instructions
- ◆ Supports add, subtract, logical, set less than, and equality test for beq and bne
 - Both signed and unsigned versions of add, sub, and slt

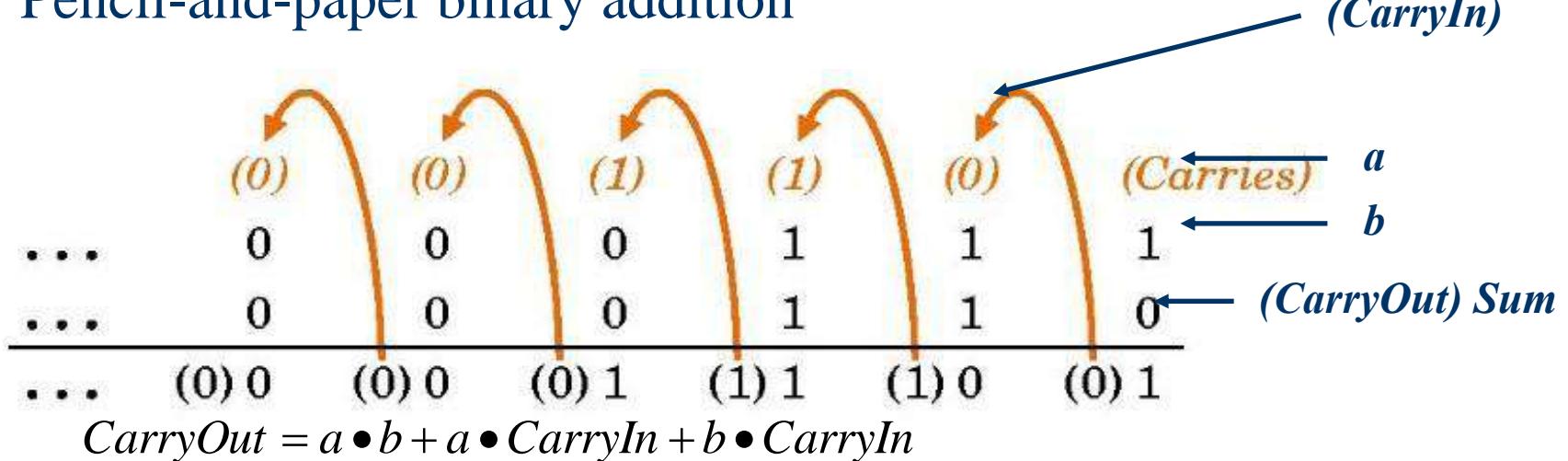
ALU block diagram

- ◆ Inputs
 - a,b: the data (operands) to be operated on
 - ALU operation: the operation to be performed
- ◆ Outputs
 - Result: the result of the operation
 - Zero: indicates if the Result = 0 (for beq, bne)
 - CarryOut: the carry out of an addition operation
 - Overflow: indicates if an add or sub had an overflow (later)



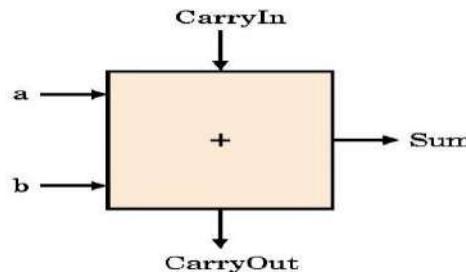
Basic integer addition

- ◆ Pencil-and-paper binary addition



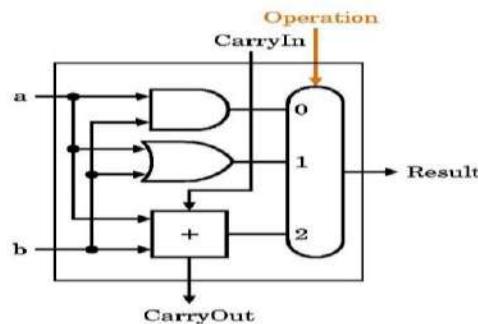
- ◆ Full adder sum and carry equations for each bit

$$\text{Sum} = a \bullet \bar{b} \bullet \overline{\text{CarryIn}} + \bar{a} \bullet b \bullet \overline{\text{CarryIn}} + \bar{a} \bullet \bar{b} \bullet \text{CarryIn} + a \bullet b \bullet \text{CarryIn} = a \oplus b \oplus \text{CarryIn}$$



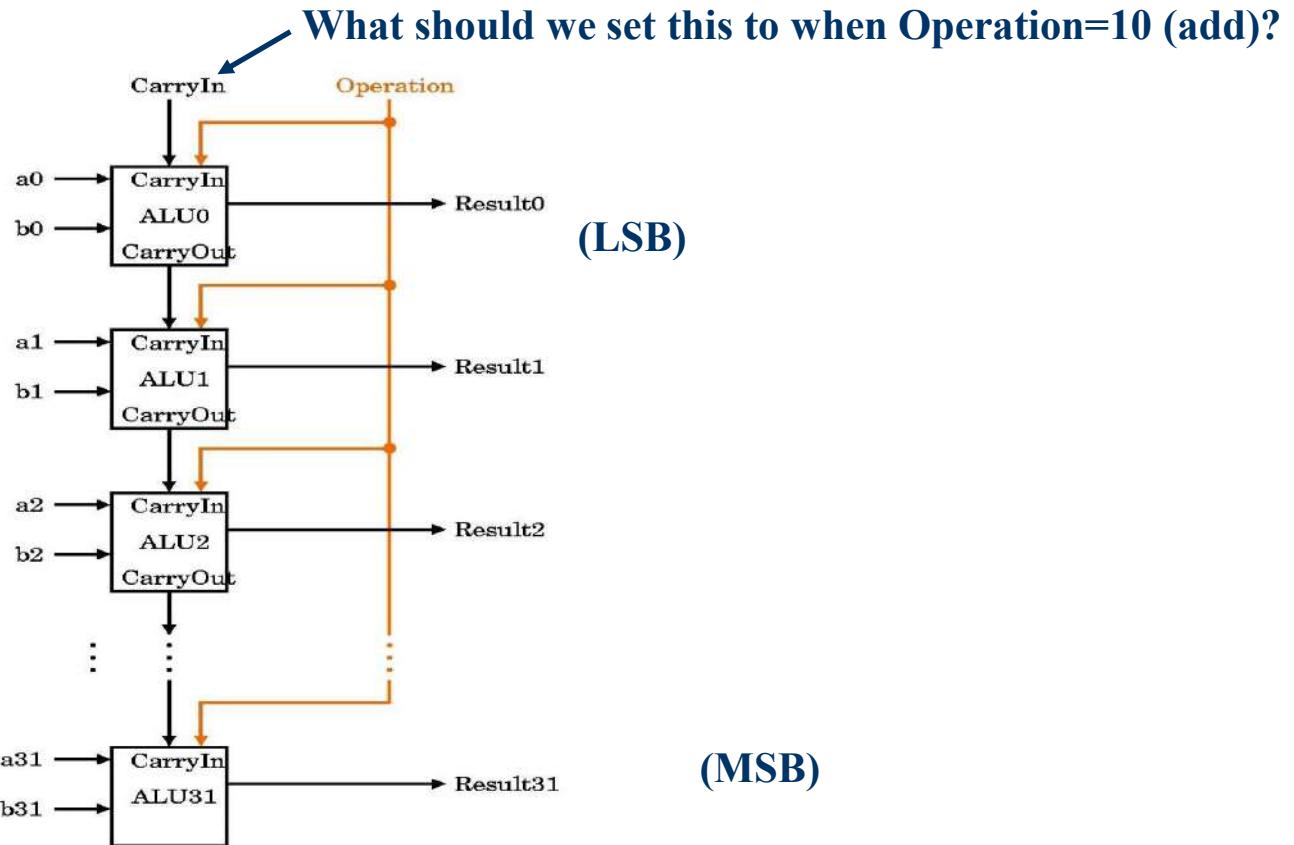
1-bit ALU *bit-slice*

- ♦ *Bit-slice design*: create a building block of part of the datapath (1 bit in our example) and replicate it to form the entire datapath
- ♦ ALU bit-slice supporting addition, AND, OR



- 1-bit AND, 1-bit OR, 1-bit full add of *a* and *b* always calculated
- *Operation* input (2 bits) determines which of these passes through the MUX and appears at *Result*
- *CarryIn* is from previous bit-slice
- *CarryOut* goes to next bit-slice

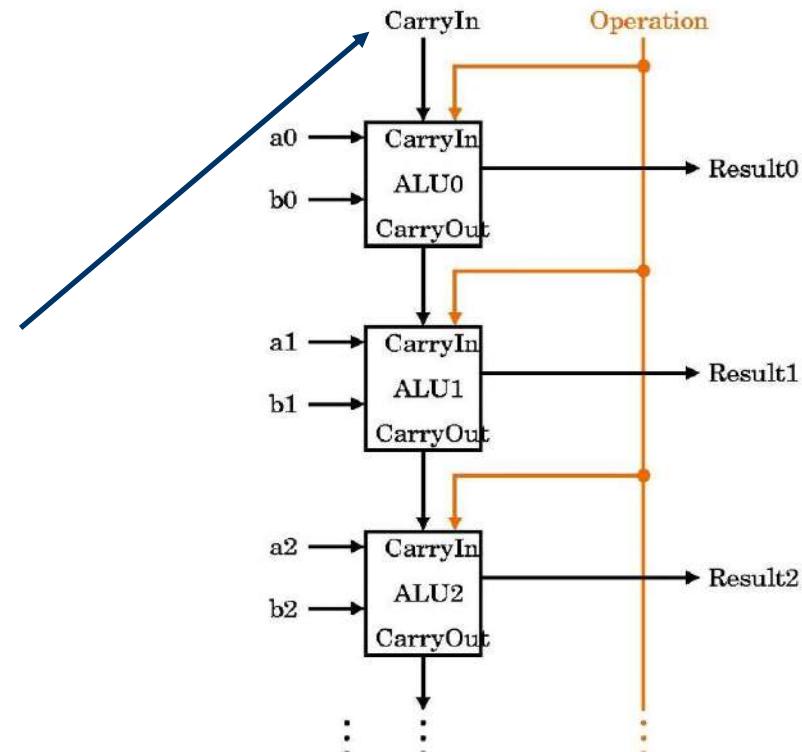
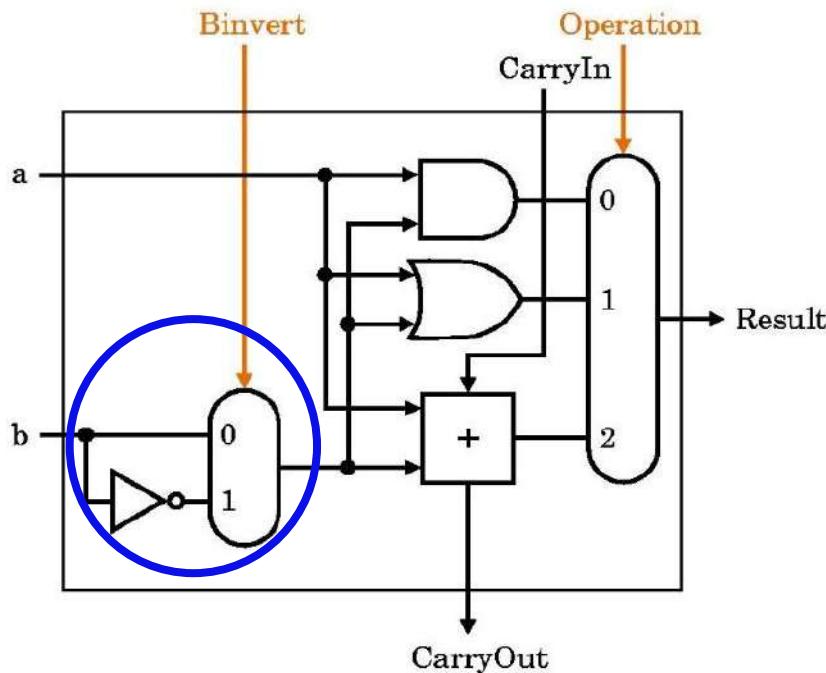
Creating a 32-bit ALU from 1-bit bit-slices



- ◆ Performs *ripple carry addition*
 - Carry follows serial path from bit 0 to bit 31 (slow)

Handling subtraction

- ◆ Perform $a+(-b)$
- ◆ Recall that to negate b we
 - Invert (NOT) all bits
 - Add a 1
 - Set CarryIn input of LSB bitslice to 1
- ◆ New bit-slice design

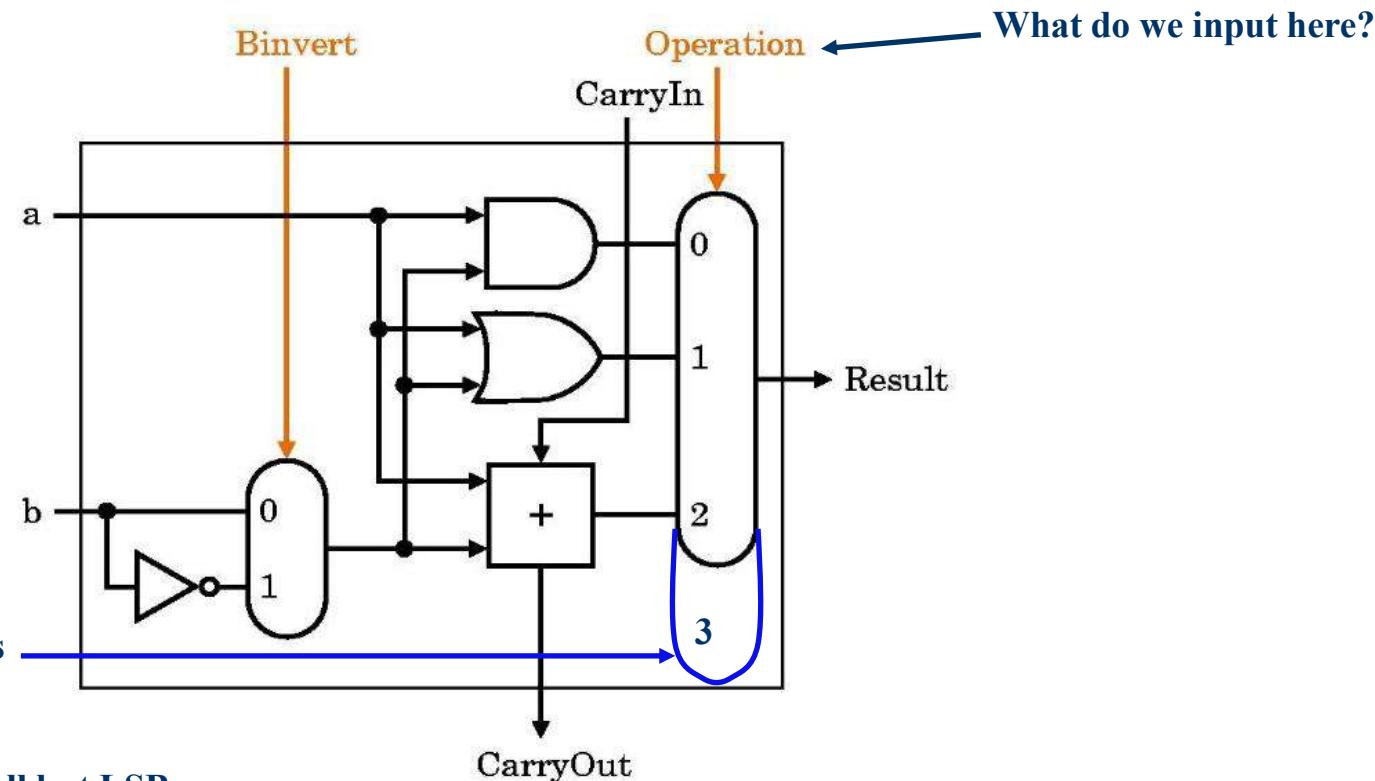


Implementing bne, beq

- ◆ Need to detect if $a=b$
- ◆ Detected by determining if $a-b=0$
 - Perform $a+(-b)$
 - NOR all Result bits to detect $\text{Result}=0$

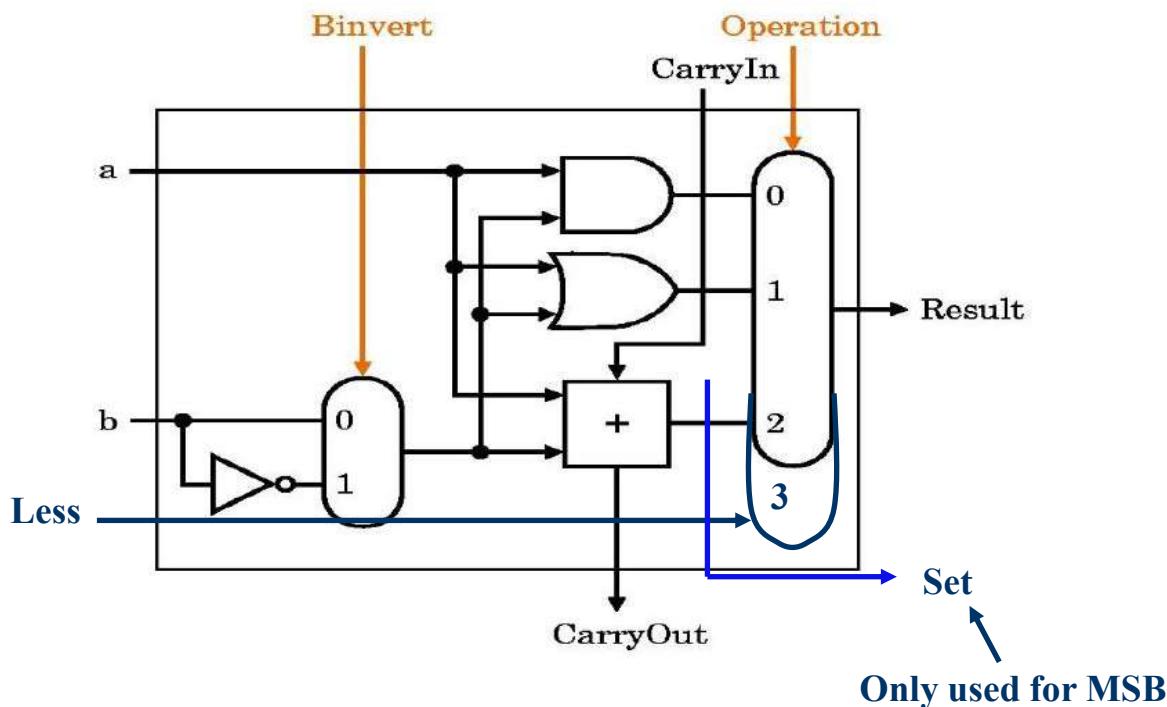
Implementing Set Less Than (slt)

- ◆ Result=1 if $a < b$, otherwise Result=0
- ◆ All bits except bit 0 are set to zero through a new bit-slice input called *Less* that goes to a 4th input on the bit-slice MUX

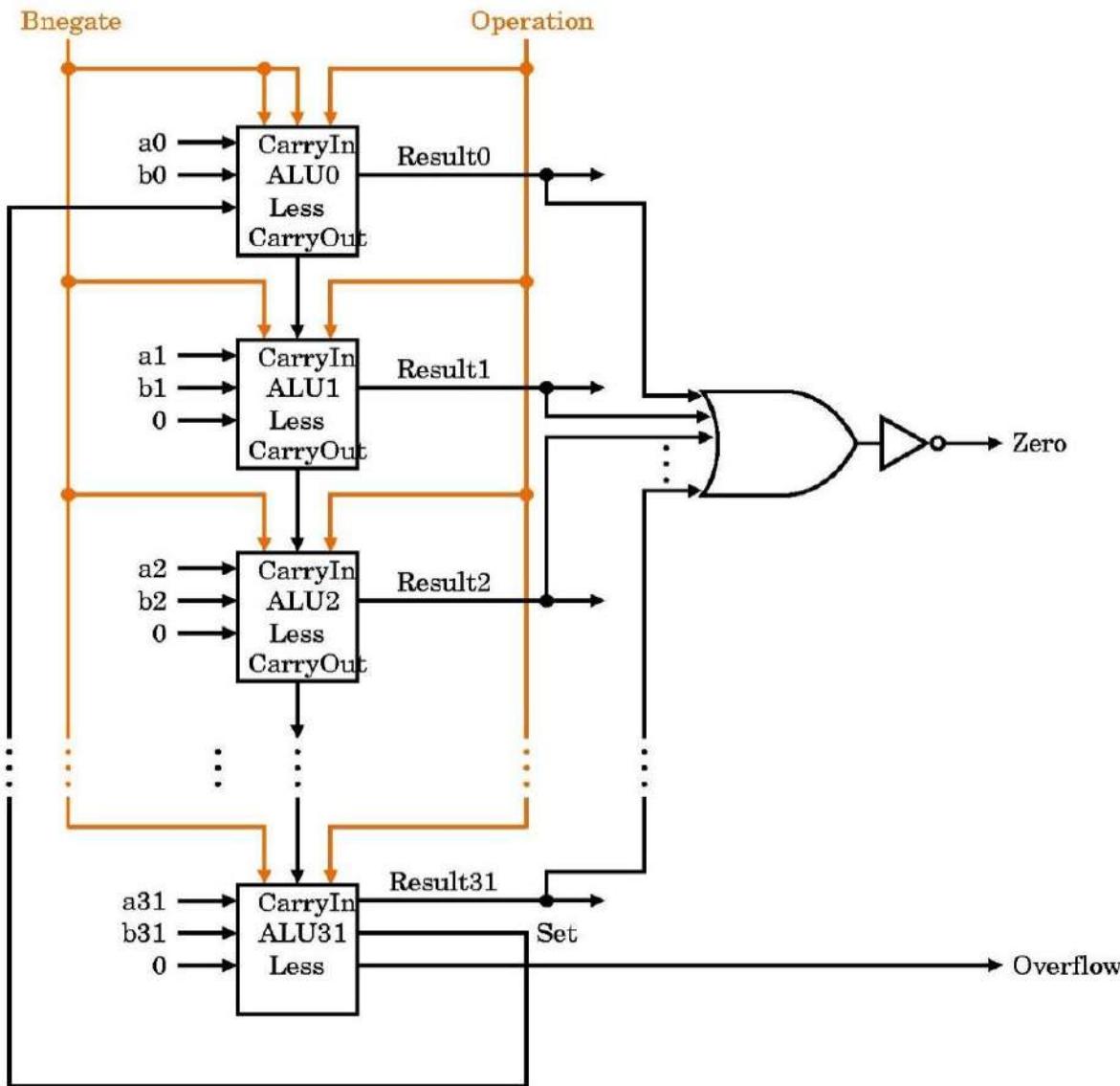


Implementing Set Less Than (slt)

- ◆ Set bit 0 to one if the result of $a-b$ is negative and to zero otherwise
 - $a-b = \text{negative number} \implies a < b$
 - Feed the adder output of bit 31 to the *Less* input of bit 0



Full 32-bit ALU design



Full 32-bit ALU design

- ◆ Bnegate controls CarryIn input to bit 0 and Binvert input to all bit-slices
 - Both are 1 for subtract and 0 otherwise, so a single signal can be used
- ◆ NOR, XOR, shift operations would also be included in a MIPS implementation

Overflow

- ◆ *Overflow* occurs when the result from an operation cannot be represented with the number of available bits (32 in our ALU)
- ◆ For signed addition, overflow occurs when
 - Adding two positive numbers gives a negative result
 - Example: 01110000...+00010000...=1000000...
 - Adding two negative numbers gives a positive result
 - Example: 10000000...+10000000...=0000000...
- ◆ For signed subtraction, overflow occurs when
 - Subtracting a negative from a positive number gives a negative result
 - Subtracting a positive from a negative number gives a positive result

Overflow

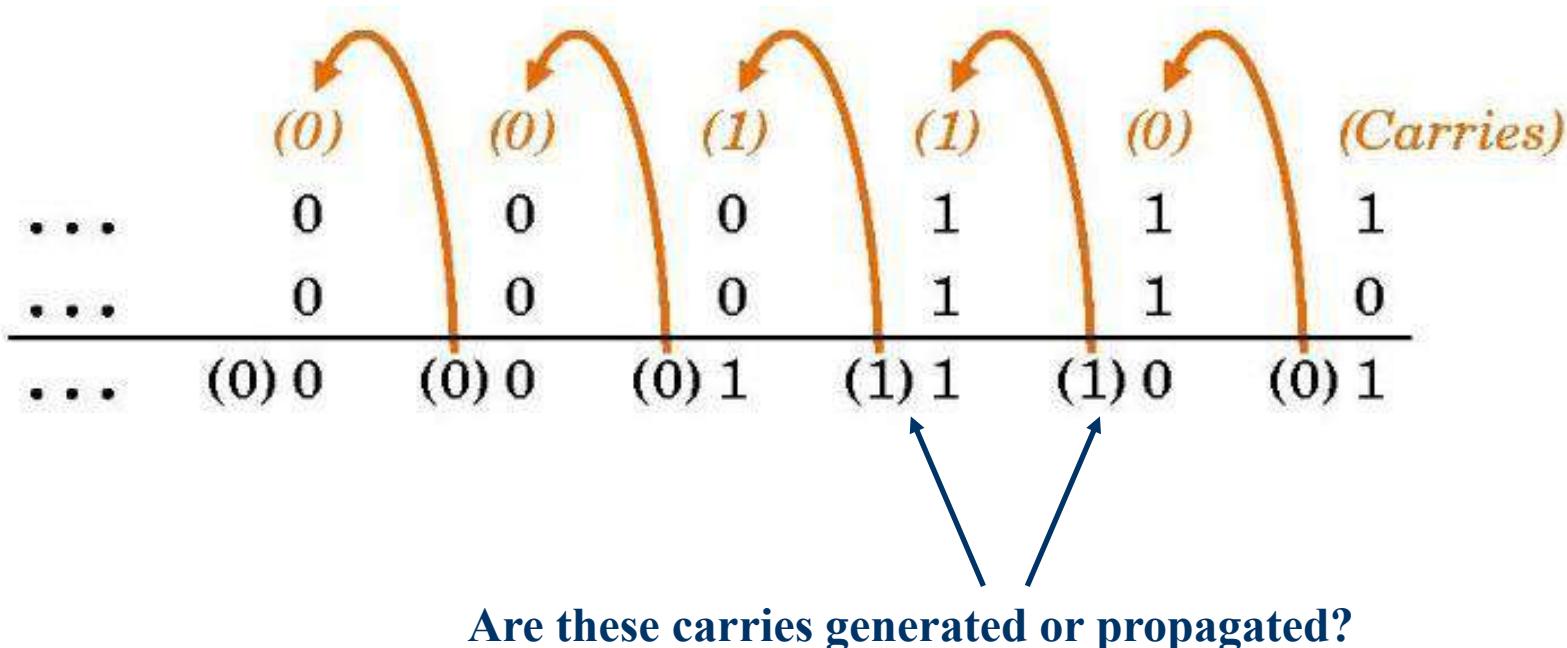
- ◆ Overflow on unsigned arithmetic, which is primarily used for manipulating addresses, is ignored in many ISAs (including MIPS)
- ◆ Overflow on signed arithmetic causes an *interrupt* to deal with the problem (Chapter 5)
- ◆ Overflow detection: XOR CarryIn of MSB with CarryOut of MSB (problem 4.42)

Faster carry generation

- ◆ Ripple carry addition is too slow for wide adders
- ◆ Some alternative faster *parallel* schemes
 - Carry lookahead
 - Carry skip
 - Carry select
- ◆ Cost is more hardware!

Carry lookahead addition

- Two ways in which carry=1 from the i th bit-slice
 - Generated* if both a_i and b_i are 1
 - $g_i = a_i \bullet b_i$
 - A CarryIn (c_i) of 1 is *propagated* if a_i or b_i are 1
 - $p_i = a_i + b_i$



Carry lookahead addition

- ◆ Two ways in which carry=1 from the i th bit-slice
 - *Generated* if both a_i and b_i are 1
 - $g_i = a_i \bullet b_i$
 - A CarryIn (c_i) of 1 is *propagated* if a_i or b_i are 1
 - $p_i = a_i + b_i$
- ◆ The carry out, c_{i+1} , is therefore
$$c_i + 1 = g_i + p_i \bullet c_i$$
- ◆ Using substitution, can get carries in parallel

$$c_1 = g_0 + p_0 \bullet c_0$$

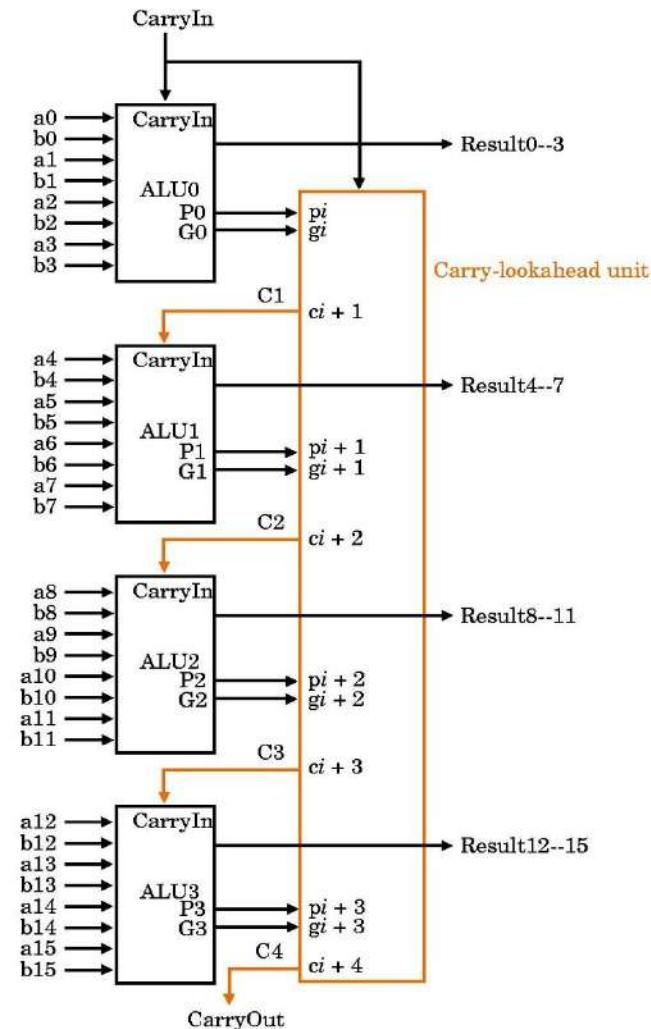
$$c_2 = g_1 + p_1 \bullet g_0 + p_1 \bullet p_0 \bullet c_0$$

$$c_3 = g_2 + p_2 \bullet g_1 + p_2 \bullet p_1 \bullet g_0 + p_2 \bullet p_1 \bullet p_0 \bullet c_0$$

$$c_4 = g_3 + p_3 \bullet g_2 + p_3 \bullet p_2 \bullet g_1 + p_3 \bullet p_2 \bullet p_1 \bullet g_0 + p_3 \bullet p_2 \bullet p_1 \bullet p_0 \bullet c_0$$
$$\vdots$$
$$\vdots$$
$$\vdots$$

Carry lookahead addition

- ◆ Drawbacks
 - $n+1$ input OR and AND gates for n th input
 - Irregular structure with many long wires
- ◆ Solution: do two levels of carry lookahead
 - First level generates Result (using carry lookahead to generate the *internal* carries) and propagate and generate signals for a *group* of 4 bits (P_i and G_i)
 - Second level generates carry out's for each group based on carry in and P_i and G_i from previous group



Carry lookahead addition

- Internal equations for group 0

$$gi = ai \bullet bi$$

$$pi = ai + bi$$

$$ci+1 = gi + pi \bullet ci$$

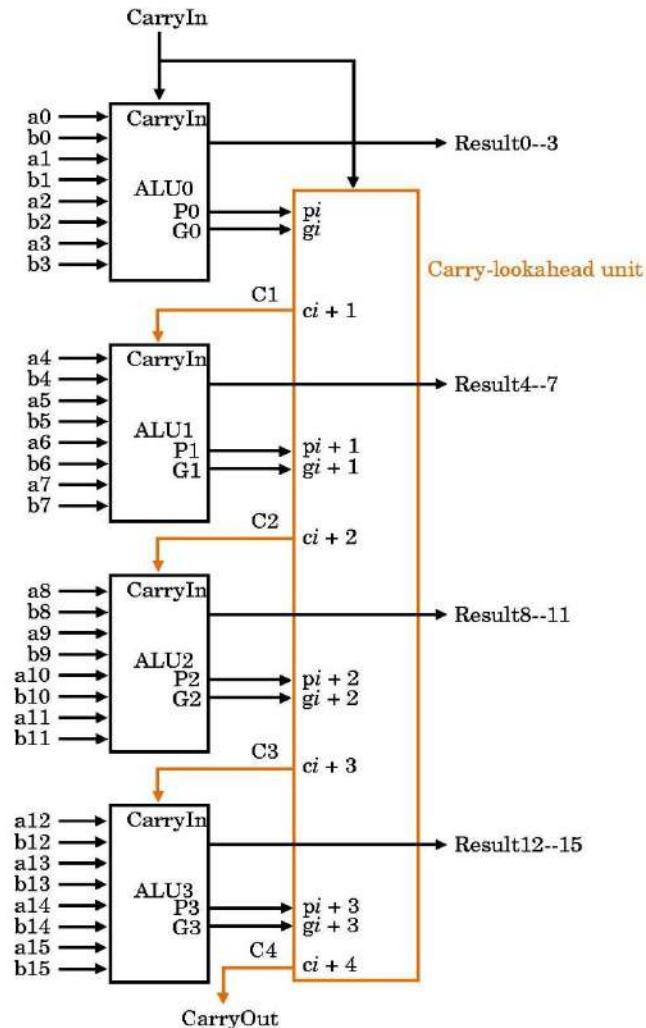
- Group equations for group 0

$$G0 = g3 + p3 \bullet g2 + p3 \bullet p2 \bullet g1 + p3 \bullet p2 \bullet p1 \bullet g0$$

$$P0 = p3 \bullet p2 \bullet p1 \bullet p0$$

- C1 output of carry lookahead unit

$$C1 = G0 + P0 \bullet c0$$

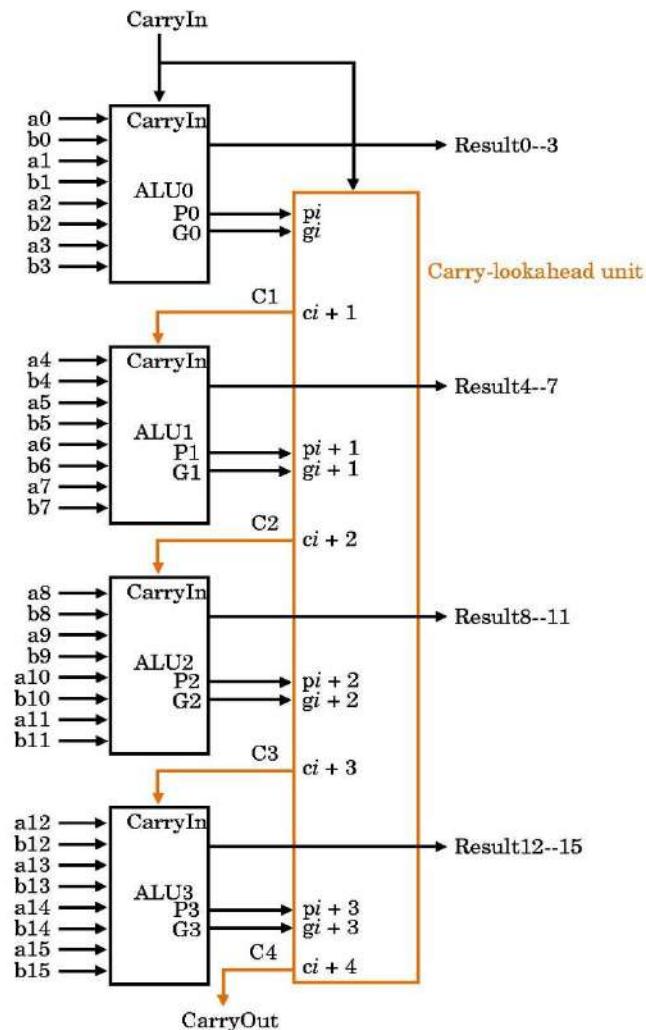


Carry lookahead addition

- ◆ Internal equations for group 1

- ◆ Group equations for group 1

- ◆ C2 output of carry lookahead unit

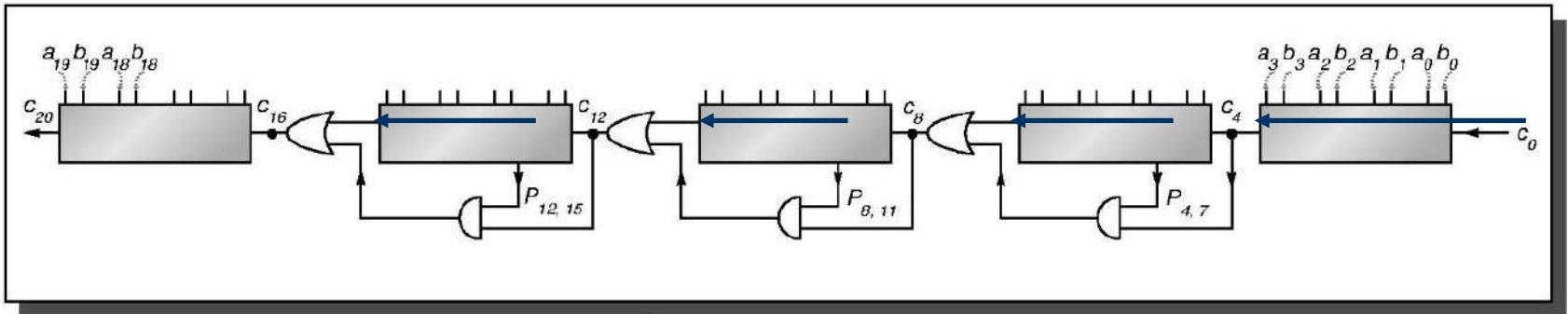


Carry skip addition

- ◆ CLA group generate (G_i) hardware is complex
- ◆ Carry skip addition
 - Generate G_i 's by ripple carry of a and b inputs with cin 's = 0 (except c_0)
Generate P_i 's as in carry lookahead
 - For each group, combine G_i , P_i , and cin as in carry lookahead to form group carries
 - Generate sums by ripple carry from a and b inputs and group carries

Carry skip addition

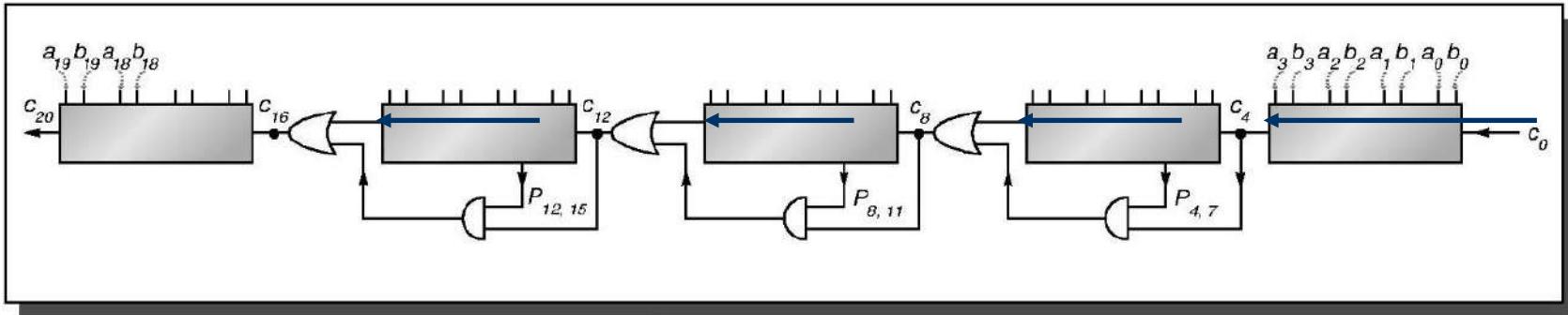
- ◆ Operation
 - Generate Gi's through ripple carry



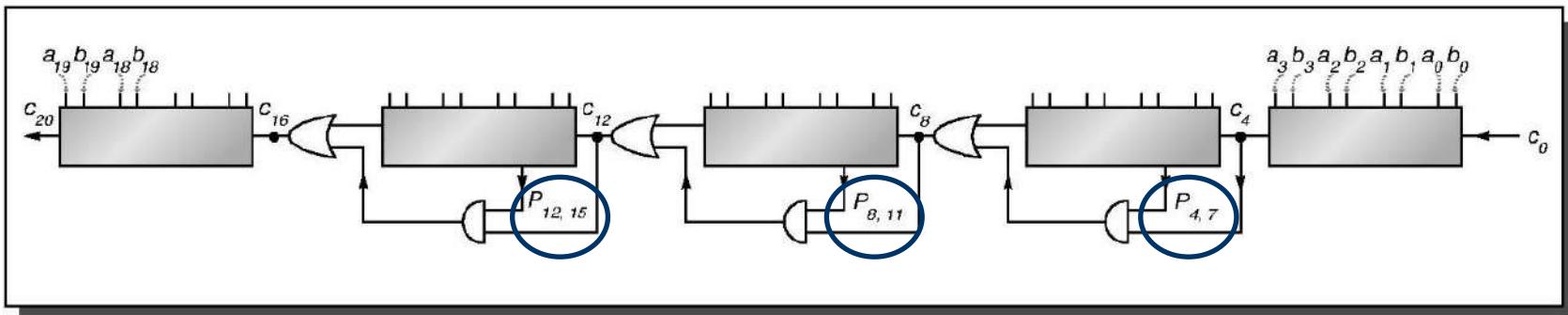
Carry skip addition

◆ Operation

- Generate Gi's through ripple carry



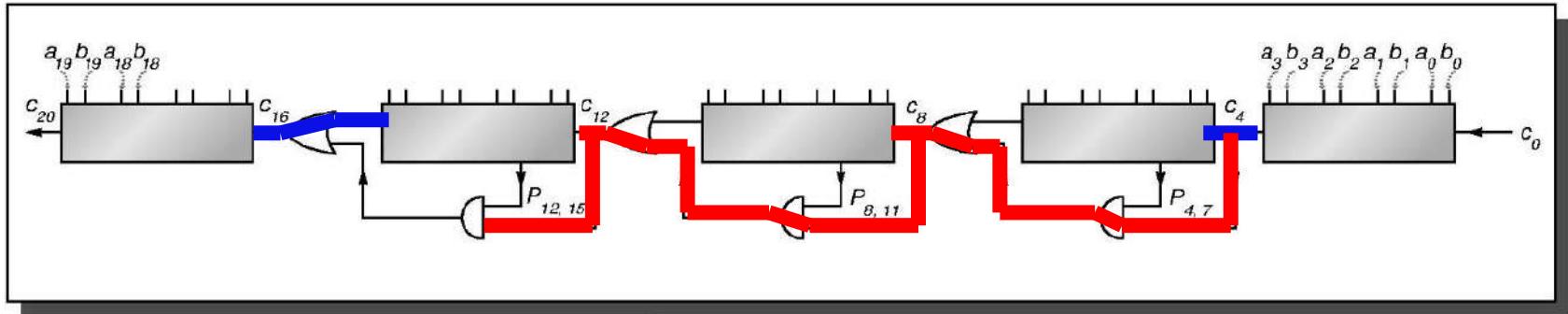
- In parallel, generate Pi's as in carry lookahead



Carry skip addition

- ◆ Operation

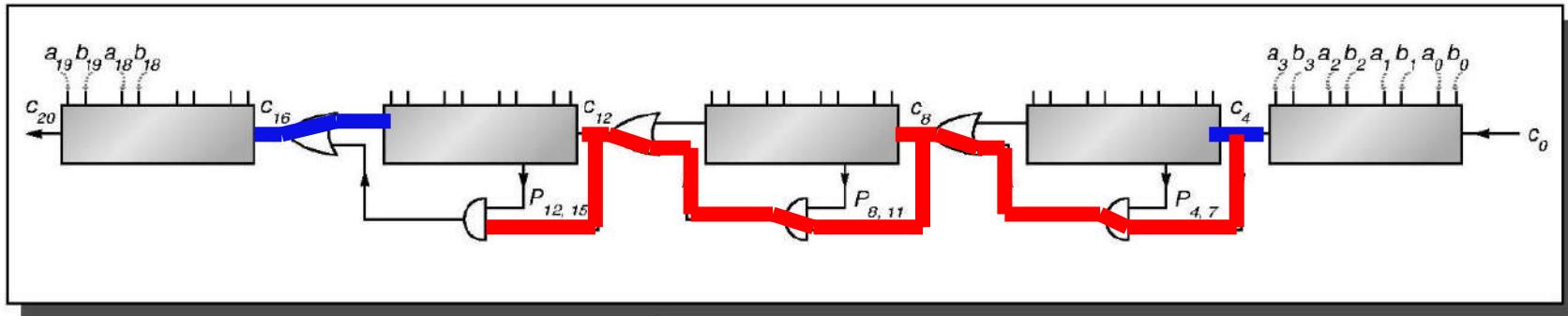
- Group carries to each block are generated or propagated



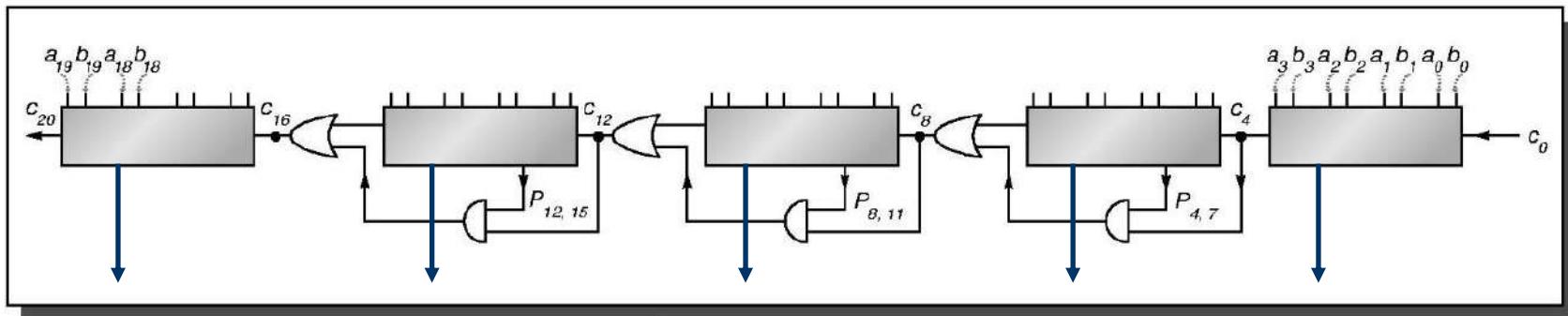
Carry skip addition

◆ Operation

- Group carries to each block are generated or propagated

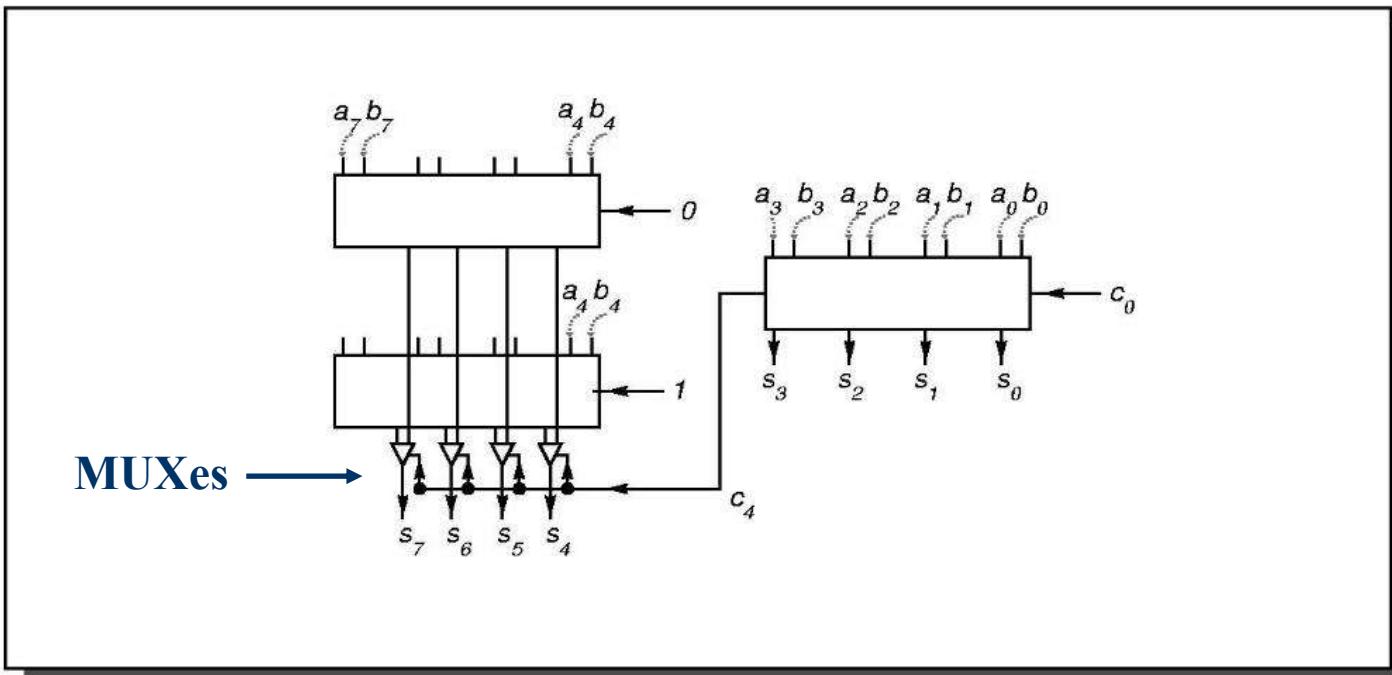


- Sums are generated in parallel from group carries



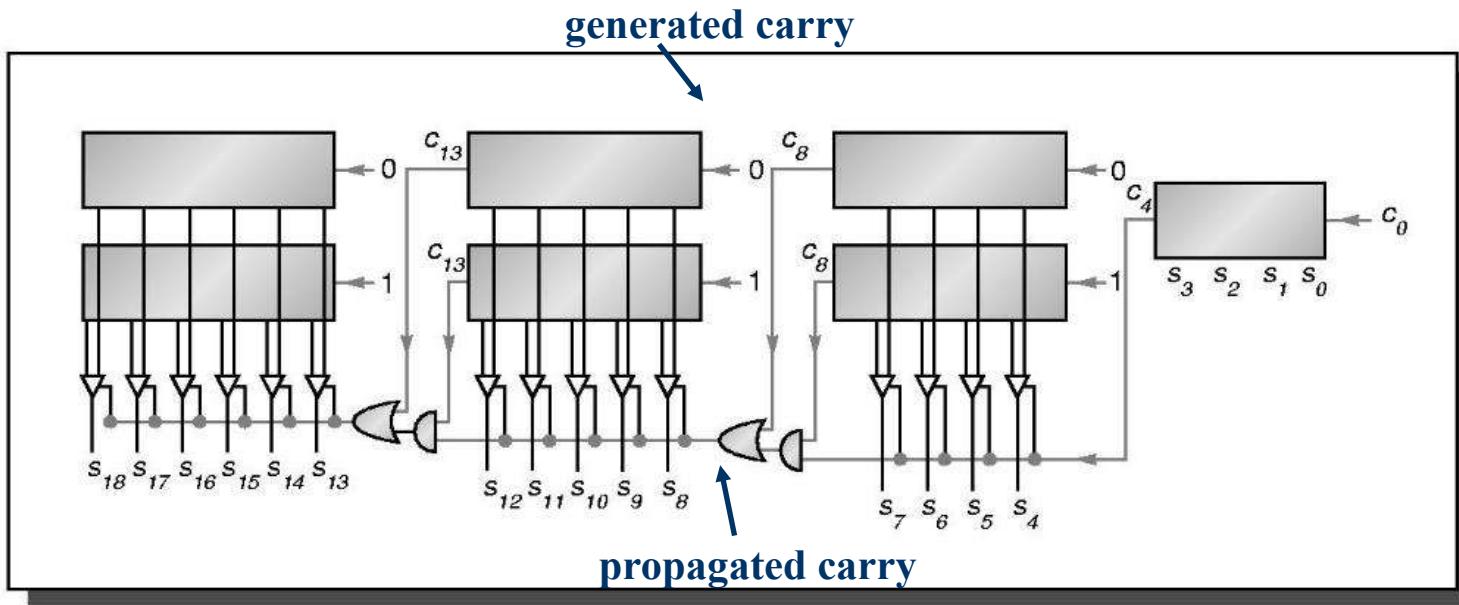
Carry select addition

- ◆ For each group, do two additions in parallel
 - One with cin forced to 0
 - One with cin forced to 1
- ◆ Generate cin in parallel and use a MUX to select the correct sum outputs
- ◆ Example for 8 bits



Carry select addition

- ◆ A larger design



- Why different numbers of bits in each block?
 - Hint1: it's to minimize the adder delay
 - Hint2: assume a k-input block has k time units of delay, and the AND-OR logic has 1 time unit of delay

Time and space comparison of adders

Adder	Time (worst)	Space (worst)
Ripple	$O(n)$	$O(n)$
CLA	$O(\log n)$	$O(n \log n)$
Carry skip	$O(\sqrt{n})$	$O(n)$
Carry select	$O(\sqrt{n})$	$O(n)$

- ◆ Differences only matter for large number of bits
- ◆ Other factors
 - Ripple carry is the most regular structure (most amenable to VLSI implementation, but any can be used in practice)
 - Carry skip requires clearing cin's at start of operation (such as dynamic CMOS logic)
 - Carry select requires driving many MUXes

Questions?



Binary Codes

Binary codes are codes which are represented in binary system with modification from the original ones. Below we will be seeing the following:

- Weighted Binary Systems
- Non Weighted Codes



Weighted Binary Systems

Weighted binary codes are those which obey the positional weighting principles, each position of the number represents a specific weight. The binary counting sequence is an example.

Decimal	8421	2421	5211	Excess-3
0	0000	0000	0000	0011
1	0001	0001	0001	0100
2	0010	0010	0011	0101
3	0011	0011	0101	0110
4	0100	0100	0111	0111
5	0101	1011	1000	1000
6	0110	1100	1010	1001
7	0111	1101	1100	1010
8	1000	1110	1110	1011
9	1001	1111	1111	1100



8421 Code/BCD Code

The BCD (Binary Coded Decimal) is a straight assignment of the binary equivalent. It is possible to assign weights to the binary bits according to their positions. The weights in the BCD code are 8,4,2,1.

Example: The bit assignment 1001, can be seen by its weights to represent the decimal 9 because:

$$1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 = 9$$



2421 Code

This is a weighted code, its weights are 2, 4, 2 and 1. A decimal number is represented in 4-bit form and the total four bits weight is $2 + 4 + 2 + 1 = 9$. Hence the 2421 code represents the decimal numbers from 0 to 9.



5211 Code

This is a weighted code, its weights are 5, 2, 1 and 1. A decimal number is represented in 4-bit form and the total four bits weight is $5 + 2 + 1 + 1 = 9$. Hence the 5211 code represents the decimal numbers from 0 to 9.

❖ Reflective Code

A code is said to be reflective when code for 9 is complement for the code for 0, and so is for 8 and 1 codes, 7 and 2, 6 and 3, 5 and 4. Codes 2421, 5211, and excess-3 are reflective, whereas the 8421 code is not.

❖ Sequential Codes

A code is said to be sequential when two subsequent codes, seen as numbers in binary representation, differ by one. This greatly aids mathematical manipulation of data. The 8421 and Excess-3 codes are sequential, whereas the 2421 and 5211 codes are not.



Non Weighted Codes

Non weighted codes are codes that are not positionally weighted. That is, each position within the binary number is not assigned a fixed value.

❖ Excess-3 Code

Excess-3 is a non weighted code used to express decimal numbers. The code derives its name from the fact that each binary code is the corresponding 8421 code plus 0011(3).

Example: 1000 of 8421 = 1011 in Excess-3

❖ Gray Code

The gray code belongs to a class of codes called minimum change codes, in which only one bit in the code changes when moving from one code to the next. The Gray code is non-weighted code, as the position of bit does not contain any weight. The gray code is a reflective digital code which has the special property that any two subsequent numbers codes differ by only one bit. This is also called a unit-distance code. In digital Gray code has got a special place.

Decimal Number	Binary Code	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010

13	1101	1011
14	1110	1001
15	1111	1000

❖ Binary to Gray Conversion

- Gray Code MSB is binary code MSB.
- Gray Code MSB-1 is the XOR of binary code MSB and MSB-1.
- MSB-2 bit of gray code is XOR of MSB-1 and MSB-2 bit of binary code.
- MSB-N bit of gray code is XOR of MSB-N-1 and MSB-N bit of binary code.

Module:3

INTRODUCTION TO COMBINATIONAL CIRCUIT

Design of combinational circuits, Adder, Subtractor, Code Converter, Analyzing a Combinational Circuit.

Logic Circuits

Combinational Circuits

- Consist of *logic gates* whose outputs at any time are determined from the present combination of inputs
 - input variables, logic gates, and output variables
- The logic gates accept signals from the inputs and generate signals to the output

Sequential Circuits

- Consist of *memory storage elements* and *logic gates*
 - Their outputs are a function of the inputs and the state of the storage elements
 - The state of storage elements is a function of previous inputs
- The outputs of a sequential circuit also depend on the past inputs

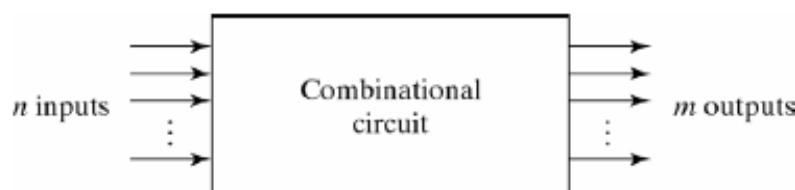


Fig. 4-1 Block Diagram of Combinational Circuit

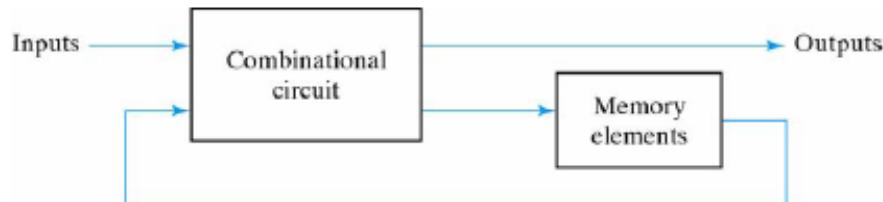


Fig. 5-1 Block Diagram of Sequential Circuit

Combinational Circuits

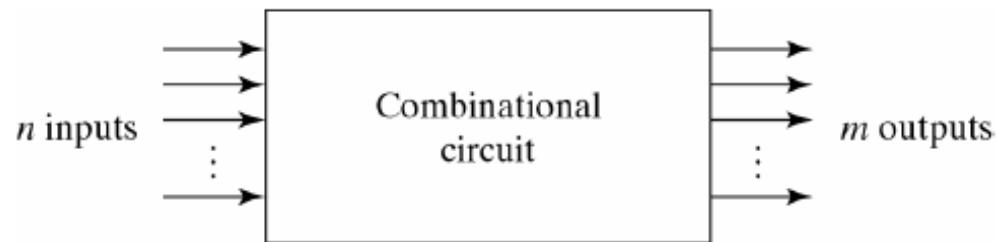


Fig. 4-1 Block Diagram of Combinational Circuit

Transform binary information from the given input data to a required output data

- n input variables
 - 2^n possible binary input combinations
 - $(2^n)^2 = 2^{2n}$ possible Boolean functions
- m output variables
 - each output function is expressed in terms of the n input variables
 - described by m Boolean functions

Standard combinational circuits

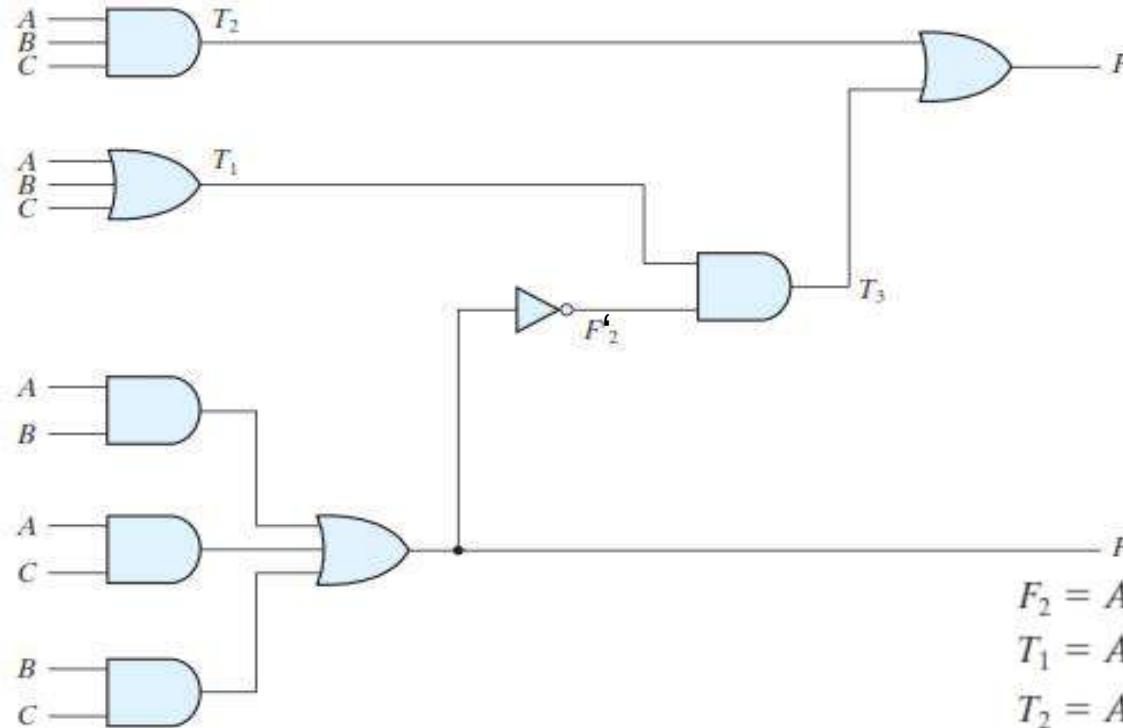
- available in MSI, standard cells in complex VLSI circuits
- i.e. adders, subtractors, comparators, decoders, encoders, multiplexers

2-9 Analysis Procedure

Start with a given logic diagram and culminate with a set of Boolean functions, a truth table, or a possible explanation of the circuit operation

- Make sure the given circuit is combinational and not sequential
 - no feedback paths or memory elements
 - A feedback path is a connection from the output of one gate to the input of a second gate that forms part of the input to the first gate
- Obtain the output Boolean functions or the truth table

Obtaining Boolean Functions



First Obtain functions of input variables

$$F_2 = AB + AC + BC$$

$$T_1 = A + B + C$$

$$T_2 = ABC$$

FIGURE 4.2
Logic diagram for analysis example

Next, we consider outputs of gates that are a function of already defined symbols:

$$T_3 = F'_2 T_1$$

$$F_1 = T_3 + T_2$$

To obtain F_1 as a function of A , B , and C , we form a series of substitutions as follows:

$$\begin{aligned} F_1 &= T_3 + T_2 = F'_2 T_1 + ABC = (AB + AC + BC)'(A + B + C) + ABC \\ &= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\ &= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\ &= A'BC' + A'B'C + AB'C' + ABC \end{aligned}$$

Half Adder - Addition of Two Bits

Table 4.3

Half Adder

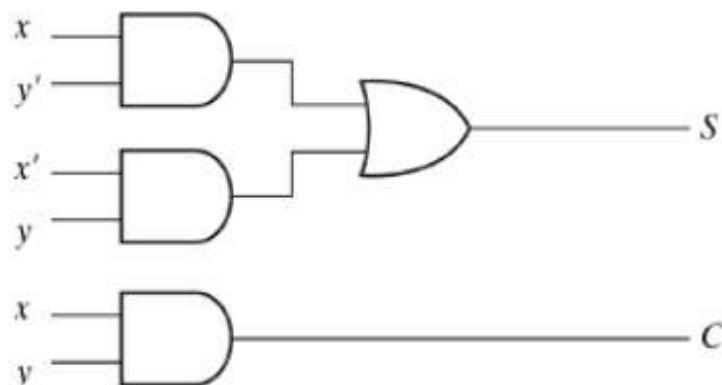
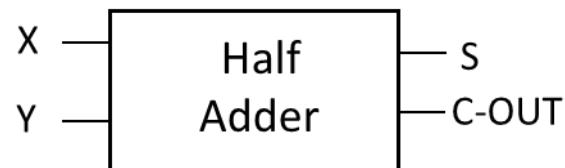
x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Two inputs: x and y

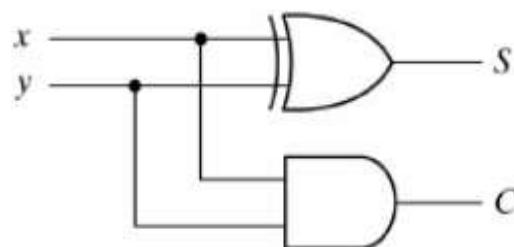
Two outputs:

- Sum S = $x'y + xy'$
- Carry C = xy

It can also be implemented with an exclusive-OR and an AND gate



$$(a) S = xy' + x'y \\ C = xy$$

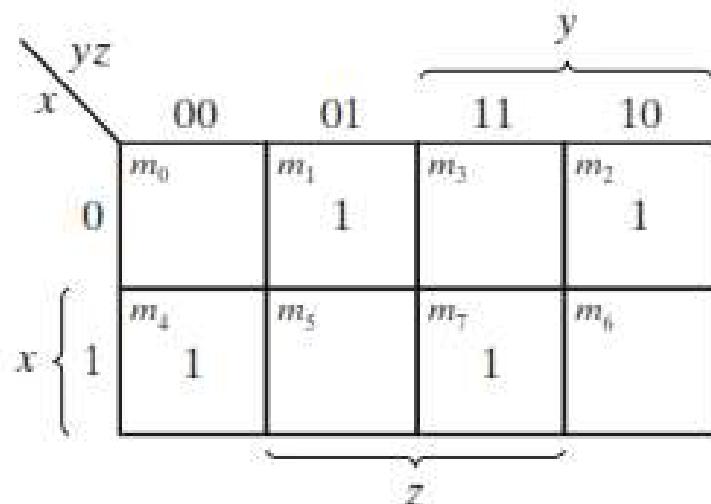
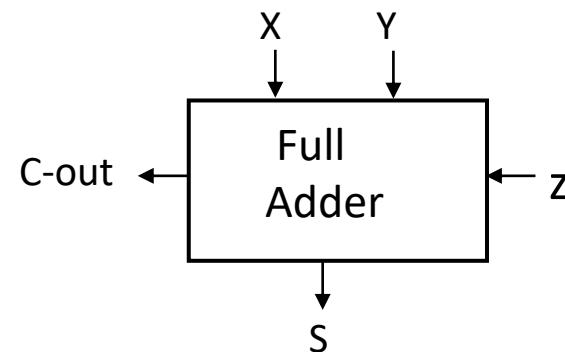


$$(b) S = x \oplus y \\ C = xy$$

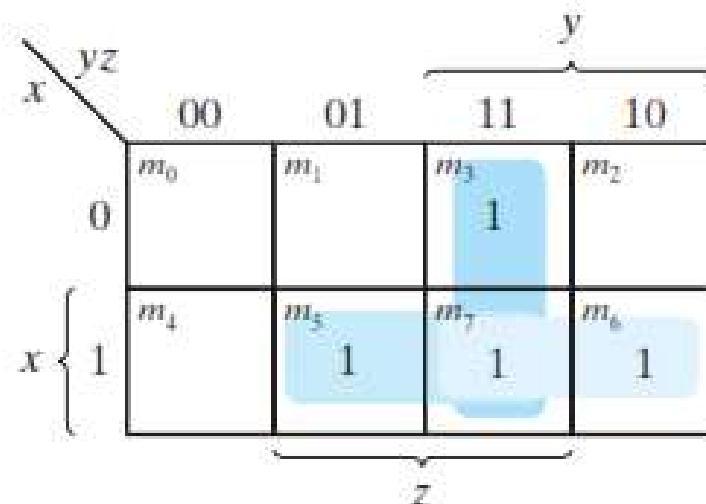
Table 4.4
Full Adder

Full Adder-Sum of Three Bits

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



(a) $S = x'y'z + x'yz' + xy'z' + xyz$



(b) $C = xy + xz + yz$

Full Adder

- Adding two single-bit binary values, X, Y with a carry input bit C-in produces a sum bit S and a carry out C-out bit.

Full Adder Truth Table

Inputs			Outputs	
X	Y	C-in	S	C-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S(X, Y, C\text{-in}) = \sum (1, 2, 4, 7)$$

$$\text{C-out}(x, y, C\text{-in}) = \sum (3, 5, 6, 7)$$

Sum S

A Karnaugh map for the sum bit S. The columns are labeled XY (00, 01, 11, 10) and the rows are labeled C-in (0, 1). The minterms 1, 3, 5, and 7 are marked with 1's. The map is annotated with X and Y at the top right, and C-in at the bottom right.

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

$$S = X'Y'(C\text{-in}) + XY'(C\text{-in})' + XY'(C\text{-in})' + XY(C\text{-in})$$

$$S = X \oplus Y \oplus (C\text{-in})$$

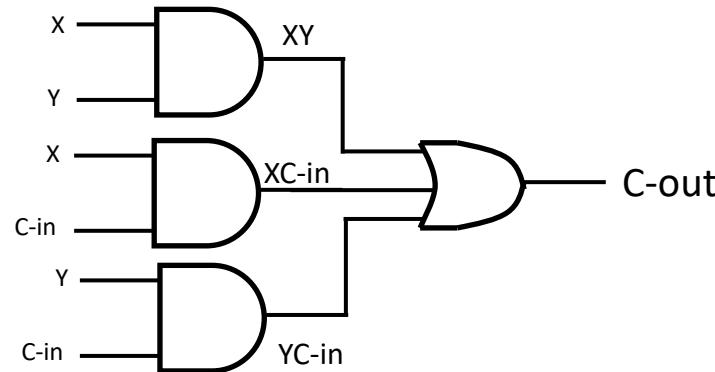
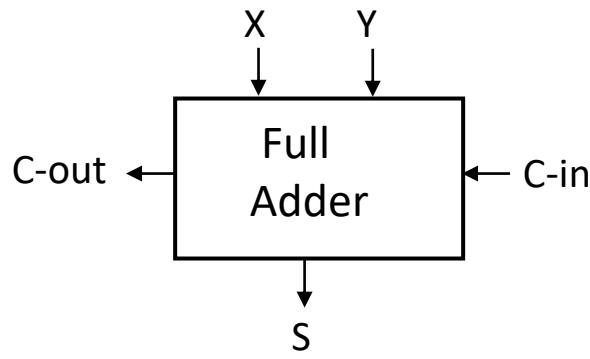
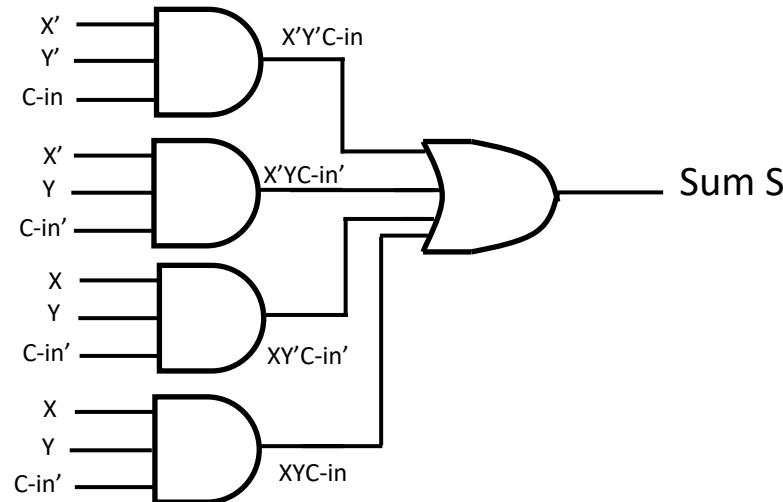
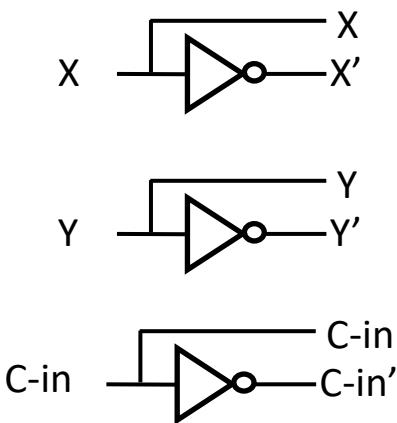
Carry C-out

A Karnaugh map for the carry bit C-out. The columns are labeled XY (00, 01, 11, 10) and the rows are labeled C-in (0, 1). The minterms 3, 5, 6, and 7 are marked with 1's. The map is annotated with X and Y at the top right, and C-in at the bottom right.

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

$$\text{C-out} = XY + X(C\text{-in}) + Y(C\text{-in})$$

Full Adder Circuit Using AND-OR

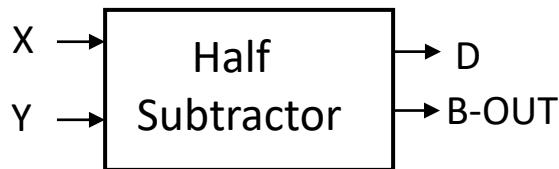


Half Subtractor

- Subtracting a single-bit binary value Y from another X (I.e. $X - Y$) produces a difference bit D and a borrow out bit B-out.
- This operation is called half subtraction and the circuit to realize it is called a half subtractor.

Half Subtractor Truth Table

Inputs		Outputs	
X	Y	D	B-out
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0



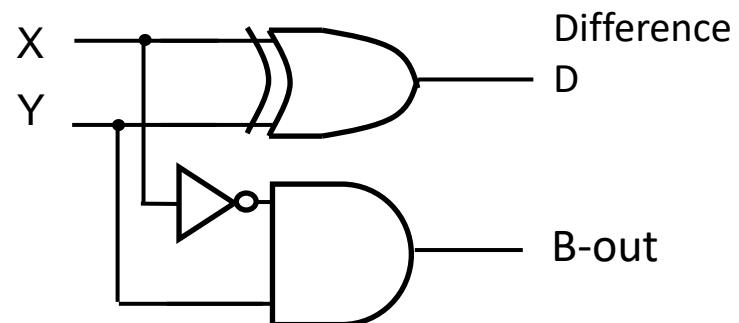
$$D(X,Y) = \Sigma (1,2)$$

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

$$D = X \oplus Y$$

$$B\text{-out}(x, y, C\text{-in}) = \Sigma (1)$$

$$B\text{-out} = X'Y$$



Full Subtractor

- Subtracting two single-bit binary values, Y, B-in from a single-bit value X produces a difference bit D and a borrow out B-out bit. This is called full subtraction.

Full Subtractor Truth Table

Inputs			Outputs	
X	Y	B-in	D	B-out
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$$D(X, Y, B\text{-in}) = \sum (1, 2, 4, 7)$$

$$B\text{-out}(x, y, B\text{-in}) = \sum (1, 2, 3, 7)$$

Difference D

		X			
	XY	00	01	11	10
B-in	0	0	2	6	4
1	1	1	3	7	5

$$D = X'Y'(B\text{-in}) + XY'(B\text{-in})' + XY'(B\text{-in})' + XY(B\text{-in})$$

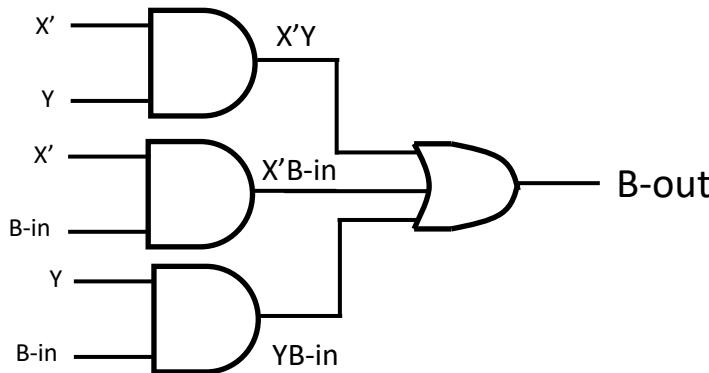
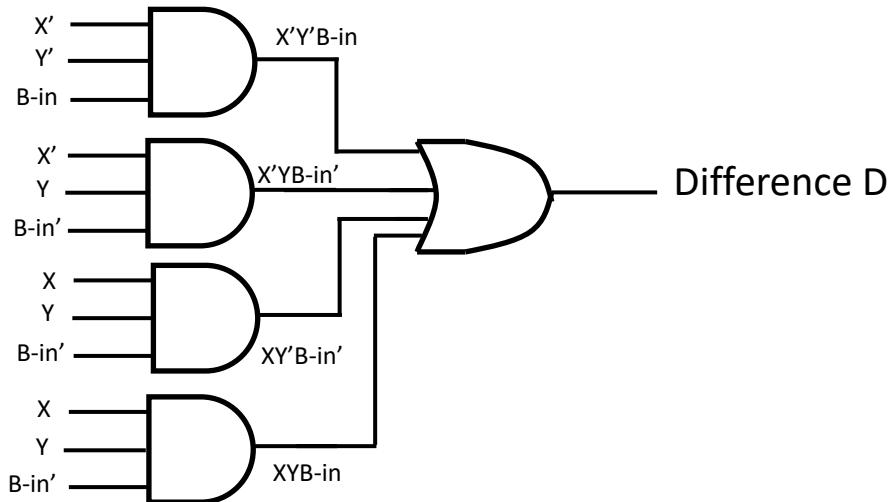
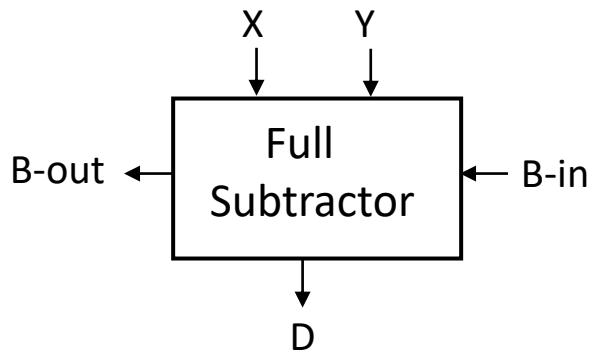
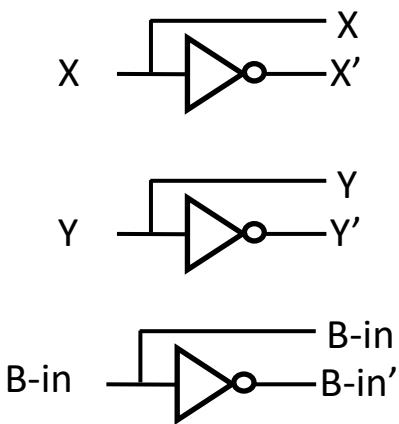
$$D = X \oplus Y \oplus (B\text{-in})$$

Borrow B-out

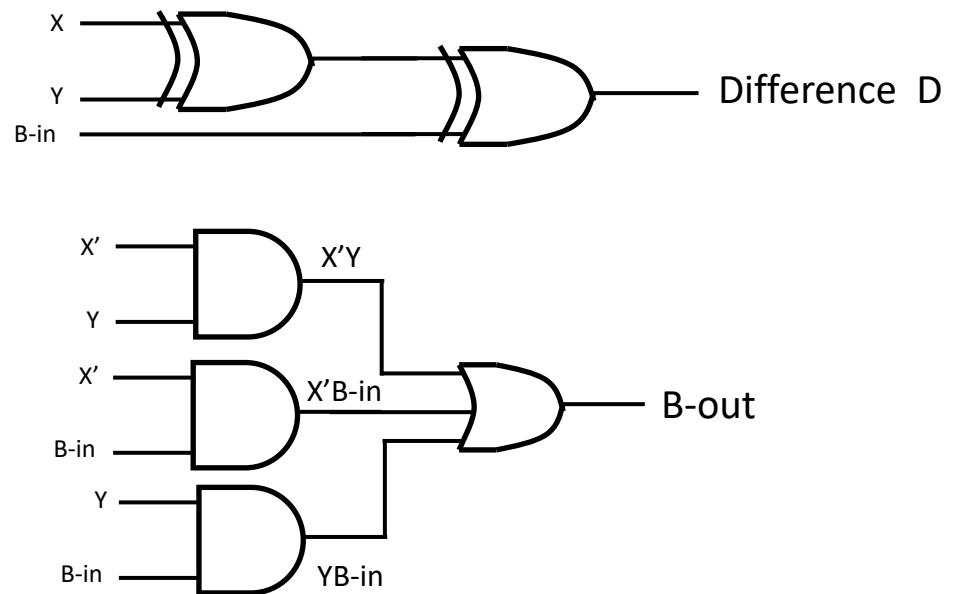
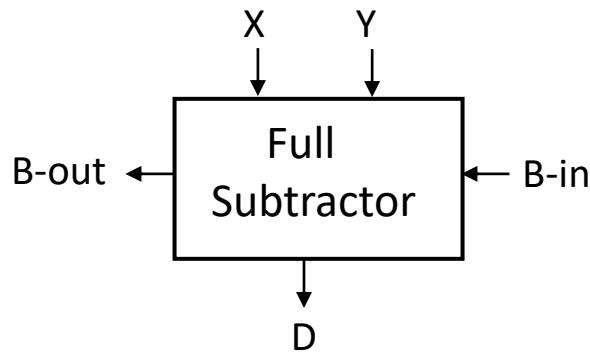
		X			
	XY	00	01	11	10
B-in	0	0	1	6	4
1	1	1	3	7	5

$$B\text{-out} = X'Y + X'(B\text{-in}) + Y(B\text{-in})$$

Full Subtractor Circuit Using AND-OR



Full Subtractor Circuit Using XOR



full subtractor

x	y	z/Bin	D	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$$\text{Diff} = \bar{x}\bar{y}z + \bar{x}y\bar{z} + x\bar{y}z + x\bar{y}\bar{z}$$

$$= \bar{x}(\bar{y}z + \bar{y}\bar{z}) + x(\bar{y}\bar{z} + yz)$$

$$= \bar{x}(\frac{y \oplus z}{K}) + x(\frac{\bar{y} \oplus z}{K})$$

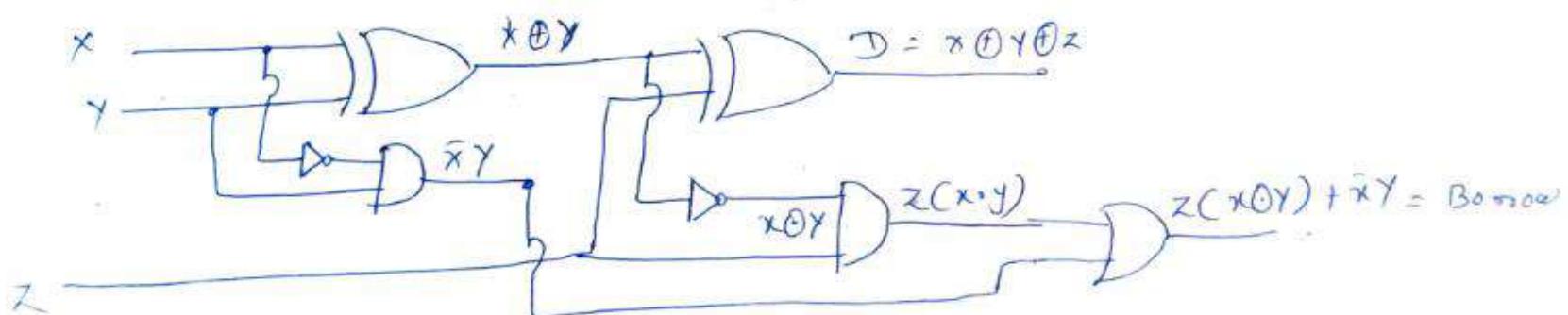
$$= \bar{x}K + x\bar{K}$$

$$= x \oplus y \oplus z$$

$$\text{Borrow}_{\text{out}} = \bar{x}\bar{y}z + \bar{x}y\bar{z} + \bar{x}yz + xy\bar{z}$$

$$= z(\bar{x}\bar{y} + xy) + \bar{x}y(\bar{z} + z)$$

$$= z(x \odot y) + \bar{x}y$$



Obtaining Boolean Functions and Truth Table from a Logic Diagram

Obtain output Boolean Functions from a logic diagram :

1. Label all gate outputs that are a function of input variables with arbitrary symbols—but with meaningful names. 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.

Obtain the Truth Table directly from a logic diagram :

1. Determine the number of input variables in the circuit. For n inputs, form the 2^n possible input combinations and list the binary numbers from 0 to $(2^n - 1)$ in a table.
2. Label the outputs of selected gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates which are a function of the input variables only.
4. Proceed to obtain the truth table for the outputs of those gates which are a function of previously defined values until the columns for all outputs are determined.

Problem

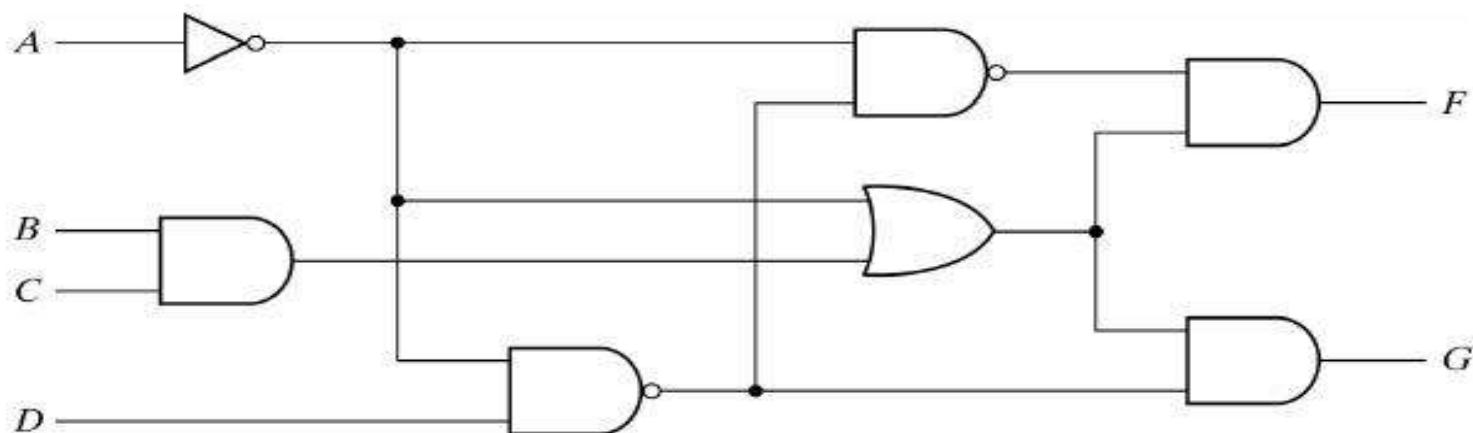


Fig. P4-2

$$F = (A + D)(A' + BC) = A'D + ABC + BCD = A'D + ABC$$

$$G = (A + D')(A' + BC) = A'D' + ABC + BCD' = A'D' + ABC$$

AB	CD	00	01	11	10
00		1	1		
01		1	1		
11			1	1	
10					1

AB	CD	00	01	11	10
00		1			1
01		1			1
11			1	1	
10					1

$$A'D + ABC + BCD = A'D + ABC$$

$$A'D' + ABC + BCD' = A'D' + ABC$$

Design Procedure

The design of combinational circuits:

- Starts from the specification of the problem and culminates in a logic circuit diagram or a set of Boolean functions.
1. From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each.
 2. Derive the truth table that defines the required relationship between inputs and outputs.
 3. Obtain the simplified Boolean functions for each output as a function of the input variables.
 4. Draw the logic diagram and verify the correctness of the design (manually or by simulation).

Constraints in a practical design:

- number of gates
- number of inputs to a gate
- propagation time of the signal through the gates
- number of interconnections
- limitations of the driving capability of each gate
- etc.

Problem Majority Circuit

A majority circuit is a combinational circuit whose output is equal to 1 if the input variables have more 1's than 0's. The output is 0 otherwise. Design a 3-input majority circuit.

xy ₃	F
000	0
001	0
010	0
011	1
100	0
101	1
110	1
111	1

x	y ₀ y ₁ y ₂	00	01	11	10
0				1	
1			1	1	1

$$F = xy + xz + yz$$

Addition of Two 2-digit's

$$\bullet A_1 A_0 + B_1 B_0 = CS_1 S_0$$

$$\begin{array}{r} 00 \\ + 00 \\ \hline 000 \end{array}$$

$$\begin{array}{r} 11 \\ + 11 \\ \hline 110 \end{array}$$

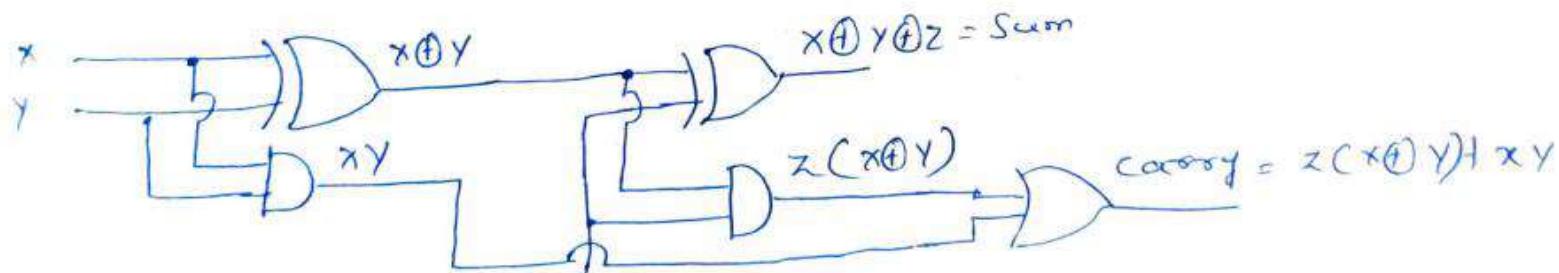
Full Adder

x	y	z/cin	s	c
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\begin{aligned}
 S &= \bar{x}\bar{y}z + \bar{x}\bar{y}\bar{z} + x\bar{y}\bar{z} + xy\bar{z} \\
 &= \bar{x}(\bar{y}z + \bar{y}\bar{z}) + x(\bar{y}\bar{z} + y\bar{z}) \\
 &= \cancel{x}(\cancel{y}\cancel{z} + \cancel{y}\cancel{\bar{z}}) + x(\cancel{y}\cancel{z} + y\cancel{z}) \\
 &= \cancel{x}\cancel{z} + \cancel{x}\cancel{z} \\
 &= x \oplus z = x \oplus y \oplus z
 \end{aligned}$$

$$\begin{aligned}
 c &= \cancel{\bar{x}\bar{y}z} + \cancel{x\bar{y}z} + \cancel{x\bar{y}\bar{z}} + \cancel{xy\bar{z}} \\
 &= z(\bar{x}y + x\bar{y}) + xy(\bar{z} + z) = z(x \oplus y) + xy
 \end{aligned}$$

Implementation of full adder by two Half adder and an OR gate



Implementations of Full-Adder

Two-level AND-OR Implementation

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

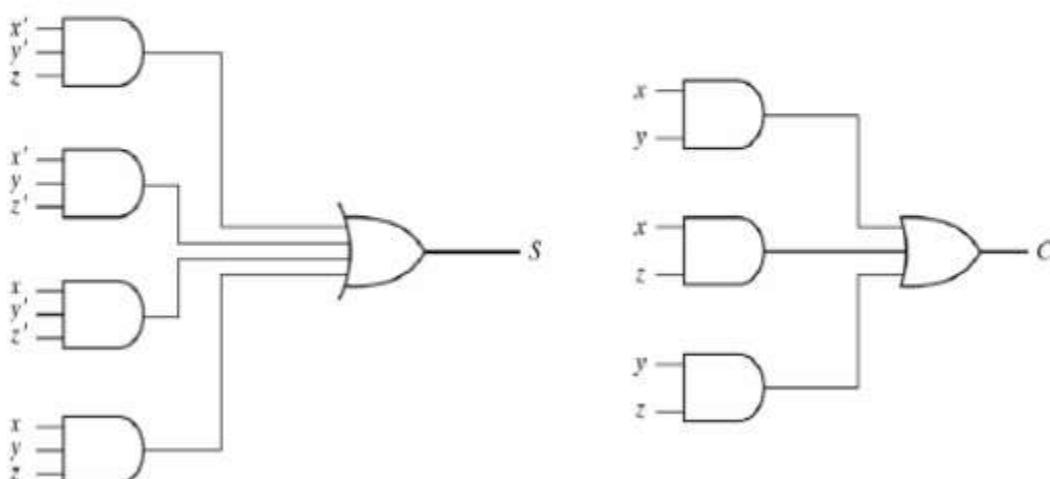


Fig. 4-7 Implementation of Full Adder in Sum of Products

Implemented with two half adders and one OR gate

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

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

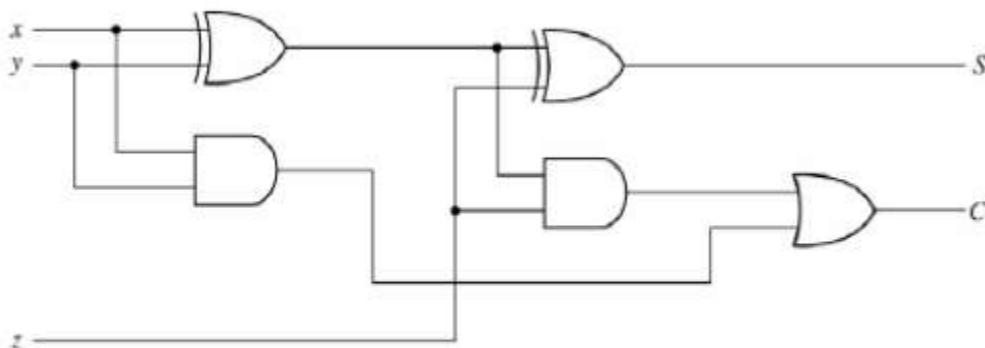


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

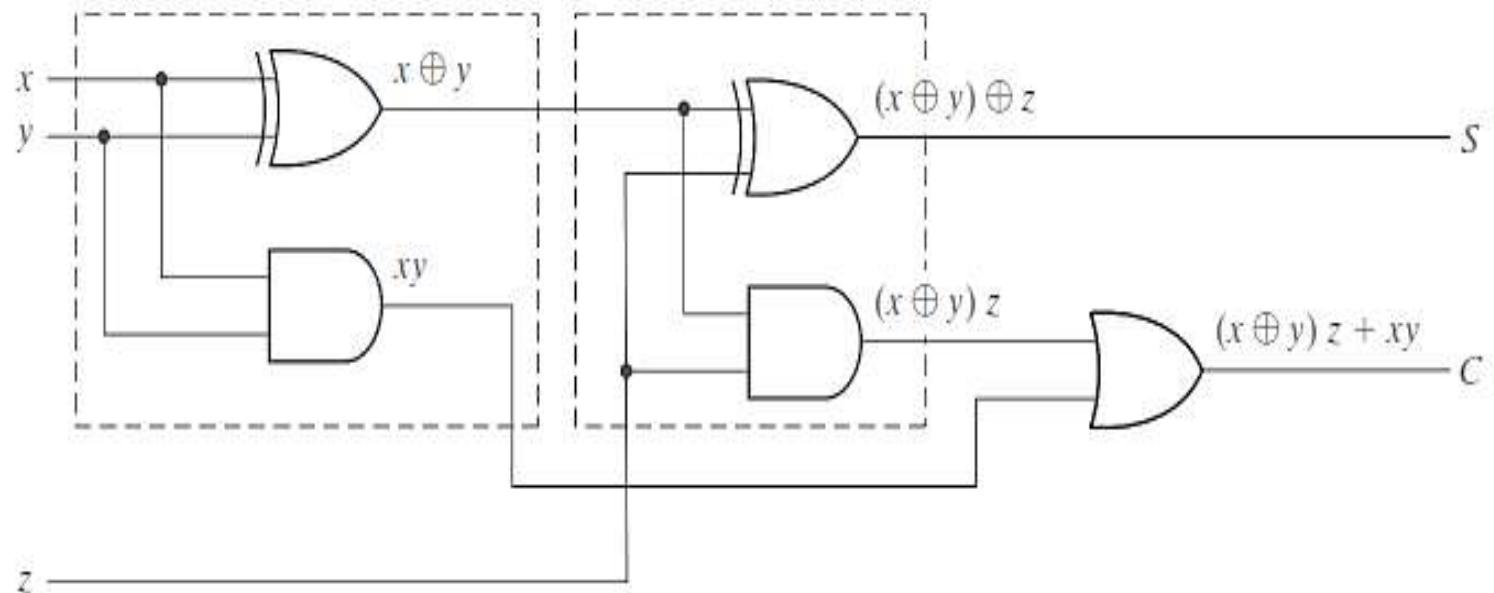


FIGURE 4.8

Implementation of full adder with two half adders and an OR gate

Binary adder

- Binary adder that produces the arithmetic sum of binary numbers can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain
- Note that the input carry C_0 in the least significant position must be 0.

Binary Adder

- For example to add $A = 1011$ and $B = 0011$

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}

Binary Adder

Example: $10+6=16$

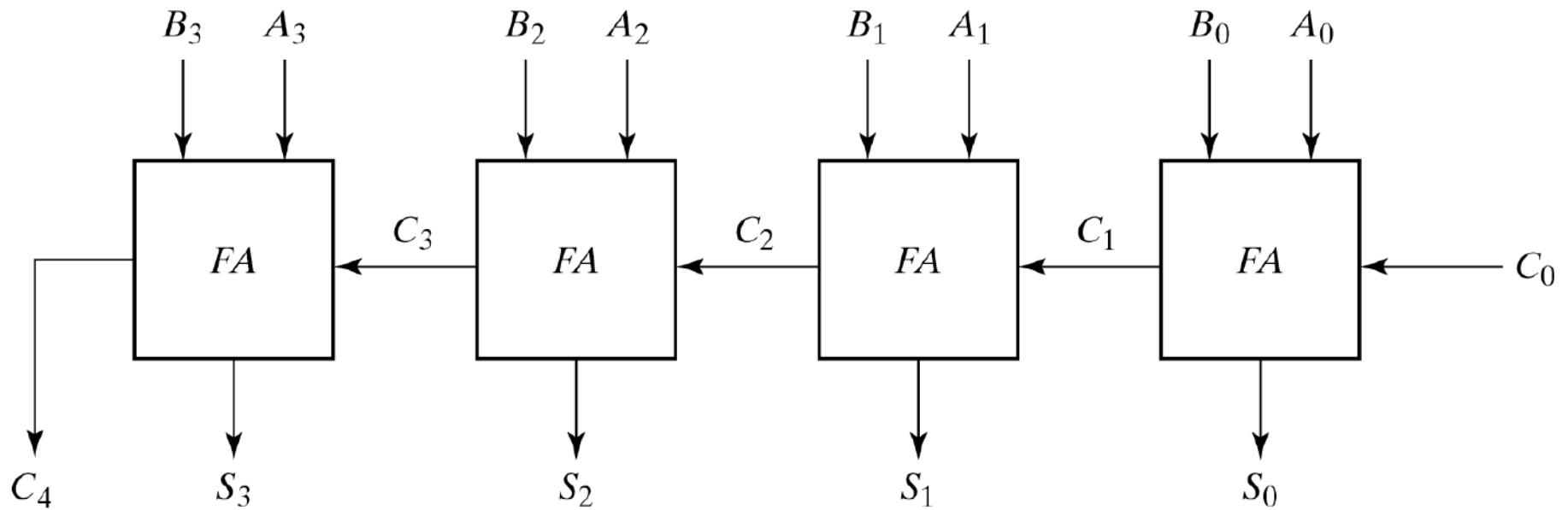


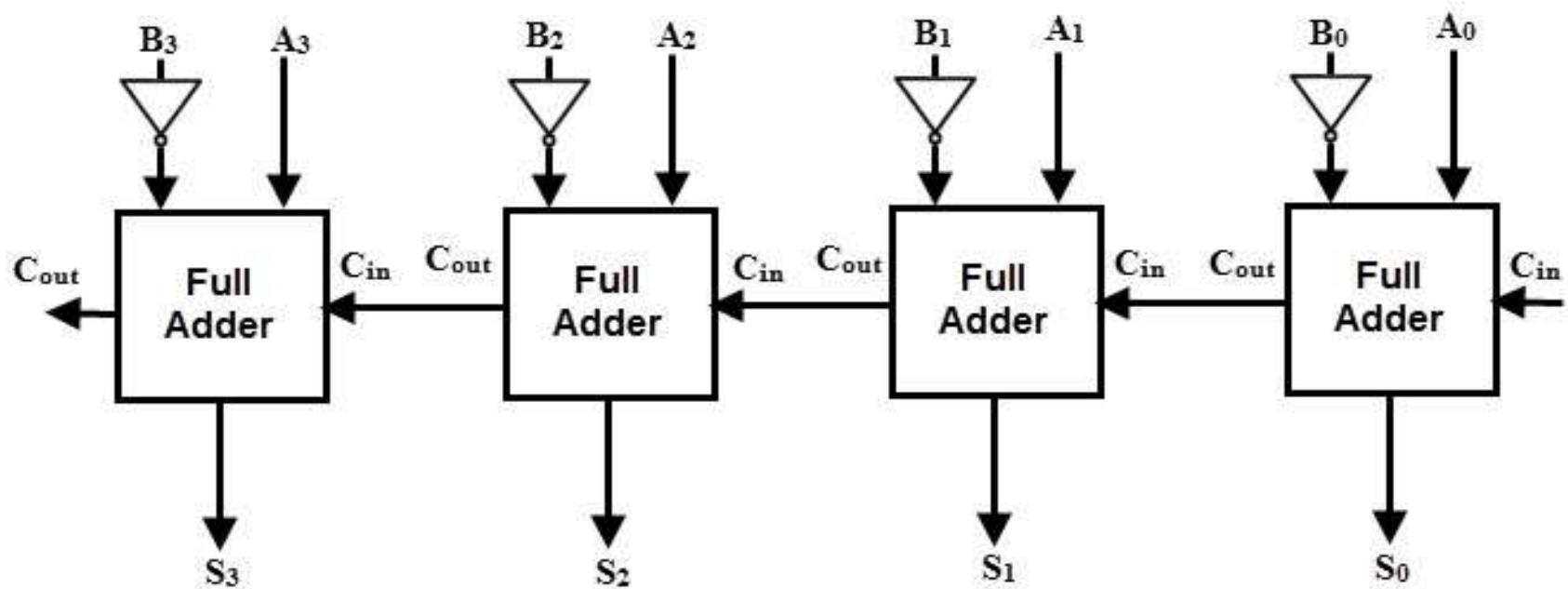
Fig. 4-9 4-Bit Adder

Binary Subtractor

- The subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A because $A - B = A + (-B) = A + ((B' + 1))$
- It means if we use the inverters to make 1's complement of B (connecting each B_i to an inverter) and then add 1 to the least significant bit (by setting carry C_0 to 1) of binary adder, then we can make a binary subtractor.

4 bit 2's complement Subtractor

Example: $10-6=10-((1\text{'s of } 6)+1)=4$



Adder Subtractor

- The addition and subtraction can be combined into one circuit with one common binary adder (see next slide).
- The mode M controls the operation. When M=0 the circuit is an **adder** when M=1 the circuit is a **subtractor**. It can be done by using exclusive-OR for each Bi and M. Note that $1 \oplus x = x'1 + x1' = x'$ and $0 \oplus x = x'0 + x0' = x$

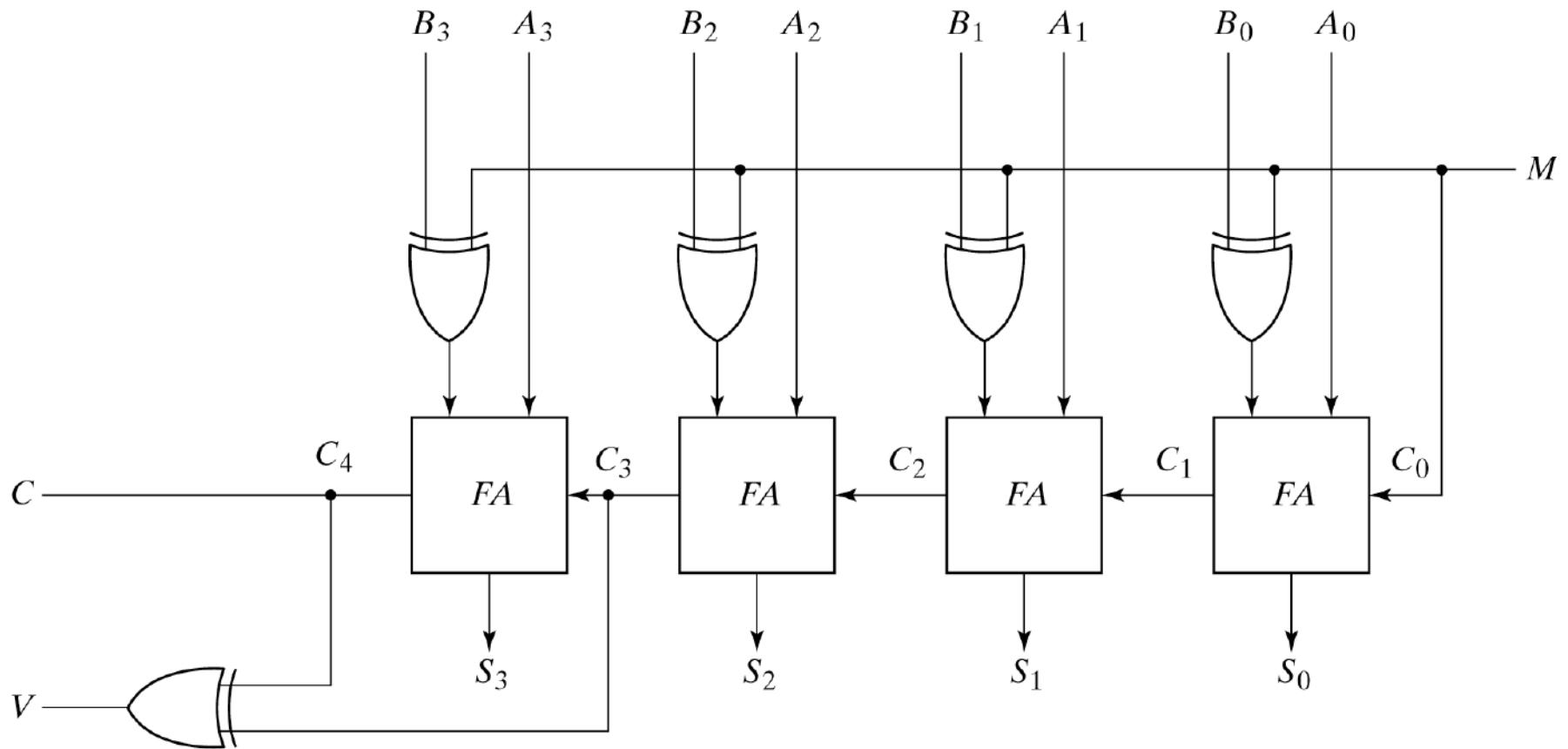


Fig. 4-13 4-Bit Adder Subtractor

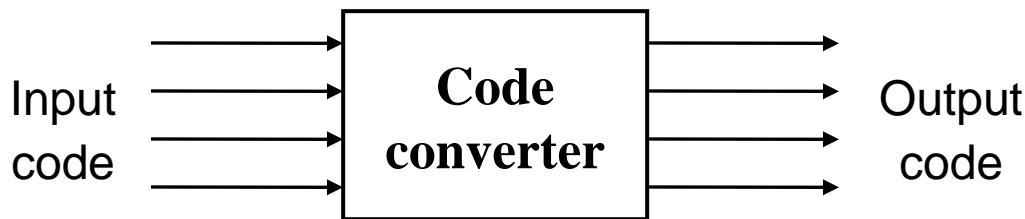
Checking Overflow

- Note that in the previous slide if the numbers considered to be signed V detects overflow. V=0 means no overflow and V=1 means the result is wrong because of overflow
- Overflow can be happened when adding two numbers of the same sign (both negative or positive) and result can not be shown with the available bits. It can be detected by observing the carry into sign bit and carry out of sign bit position. If these two carries are not equal an overflow occurred. That is why these two carries are applied to exclusive-OR gate to generate V.

CODE CONVERTERS

Code Converters

- Code converters – take an input code, translate to its equivalent output code.



Example: **BCD to Excess-3 Code Converter.**

Input: BCD digit

Output: Excess-3 digit

BCD-to-Excess-3 Code converter

- BCD is a code for the decimal digits 0-9
- Excess-3 is also a code for the decimal digits

Decimal Digit	Input BCD	Output Excess-3
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0

Specification of BCD-to-Excess3

- Inputs: a BCD input, A,B,C,D with A as the most significant bit and D as the least significant bit.
- Outputs: an Excess-3 output W,X,Y,Z that corresponds to the BCD input.
- Internal operation – circuit to do the conversion in combinational logic.

Formulation of BCD-to-Excess-3

- Excess-3 code is easily formed by adding a binary 3 to the binary or BCD for the digit.
- There are 16 possible inputs for both BCD and Excess-3.
- It can be assumed that only valid BCD inputs will appear so the six combinations not used can be treated as don't cares.

Expressions for W X Y Z

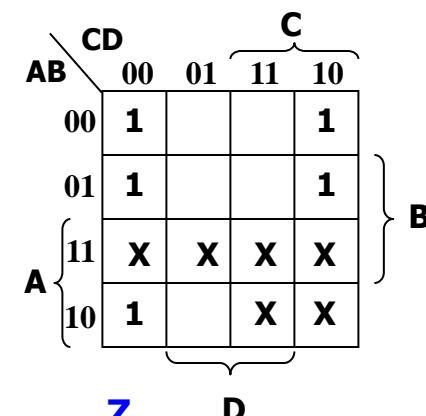
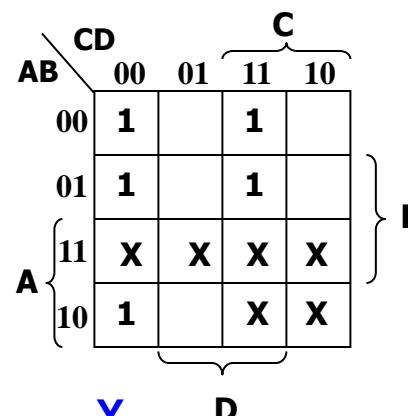
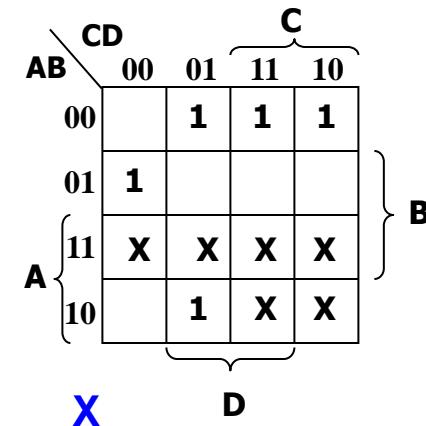
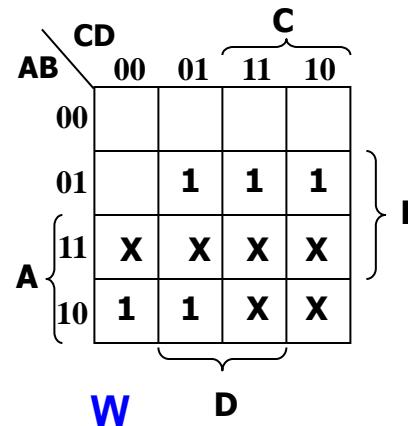
- $W(A,B,C,D) = \Sigma m(5,6,7,8,9) + d(10,11,12,13,14,15)$
- $X(A,B,C,D) = \Sigma m(1,2,3,4,9) + d(10,11,12,13,14,15)$
- $Y(A,B,C,D) = \Sigma m(0,3,4,7,8) + d(10,11,12,13,14,15)$
- $Z(A,B,C,D) = \Sigma m(0,2,4,6,8) + d(10,11,12,13,14,15)$

BCD-to-Excess-3 Code Converter

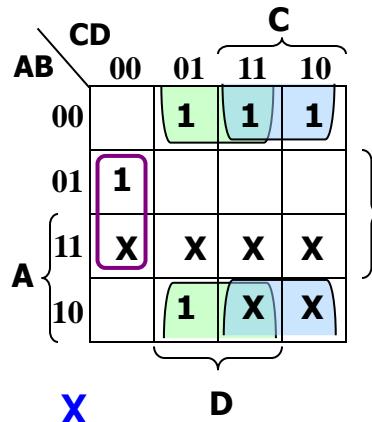
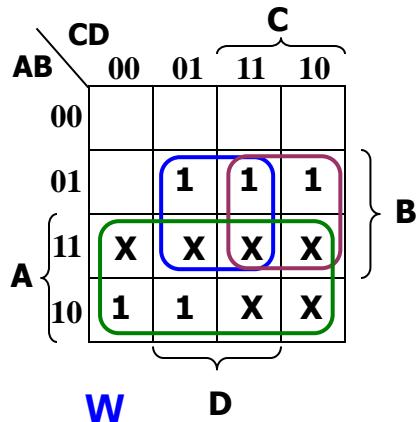
- Truth table:

	BCD				Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0
10	1	0	1	0	X	X	X	X
11	1	0	1	1	X	X	X	X
12	1	1	0	0	X	X	X	X
13	1	1	0	1	X	X	X	X
14	1	1	1	0	X	X	X	X
15	1	1	1	1	X	X	X	X

K-maps:



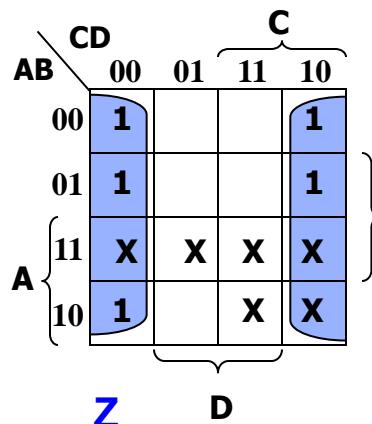
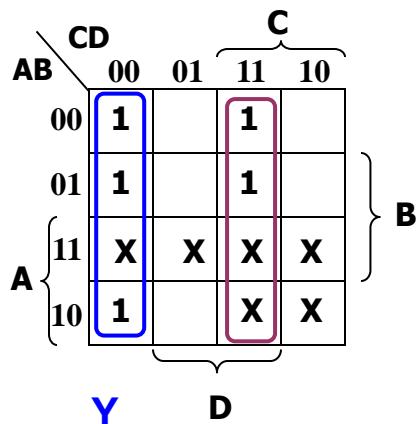
BCD-to-Excess-3 Code Converter



$$W = A + BC + BD$$

$$X = B'C + B'D + BC'D'$$

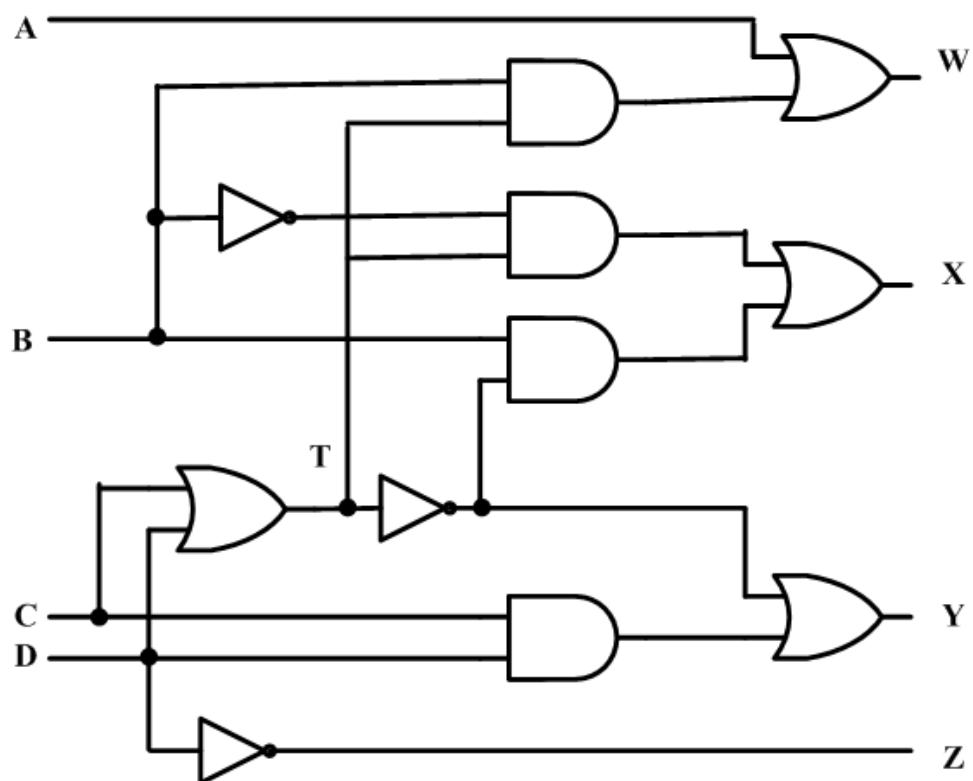
$$Y = CD + C'D'$$



$$Z = D'$$

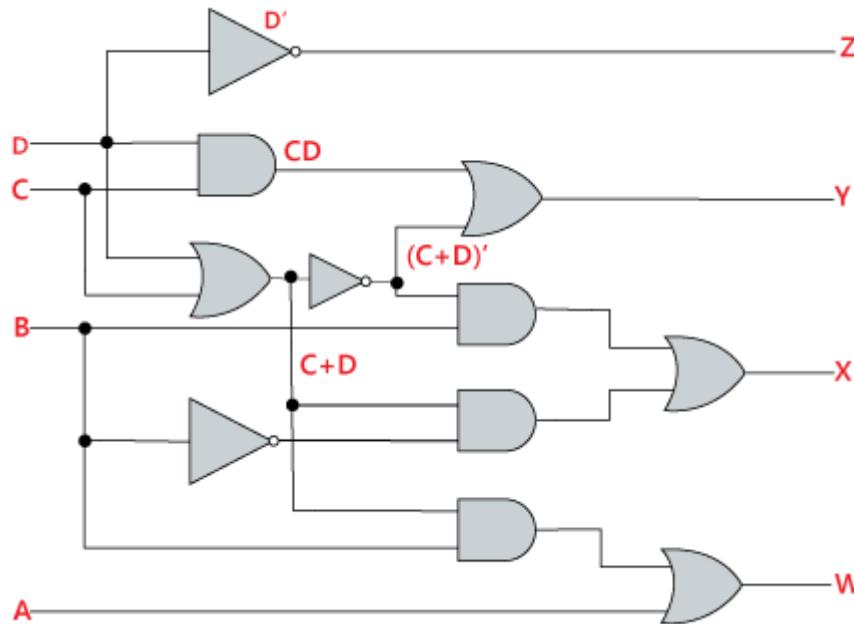
Create the digital circuit

- Implementing the second set of equations where $T=C+D$ results in a lower gate count.
- This gate has a fanout of 3

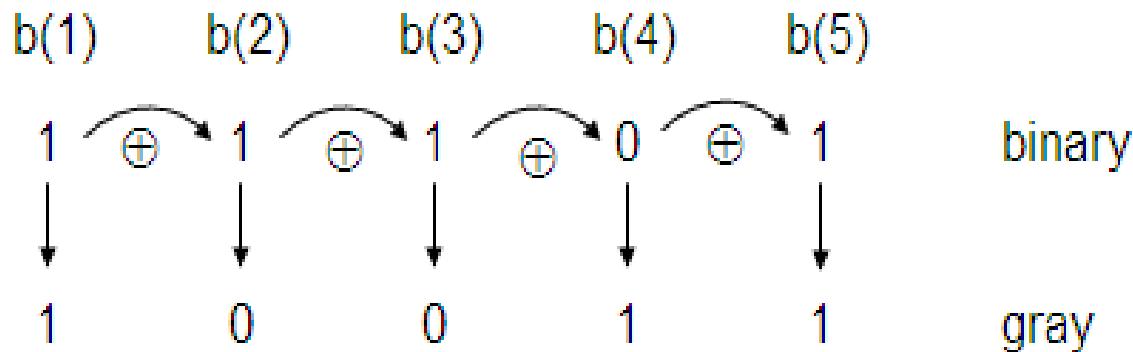


Excess-3 to BCD code converter

- $A = WX + WYZ = W(X + YZ)$
- $B = X'Y' + X'Z' + XYZ = X'(Y' + Z') + XYZ = X'(YZ)' + XYZ$
- $C = Y'Z + YZ'$
- $D = Z'$



BINARY TO GRAY CODE CONVERTER



g(1) g(2) g(3) g(4) g(5)

b(1) b(1) xor b(2) b(2) xor b(3) b(3) xor b(4) b(4) xor b(5)

BINARY TO GRAY CODE CONVERTER

Binary				Gray			
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

K-Maps for Gray code output

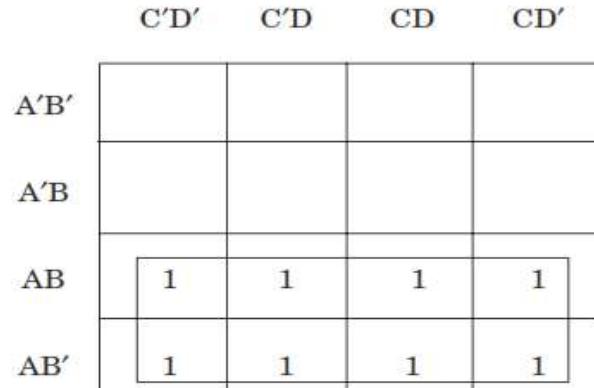


Figure 5.14(a) Karnaugh map for W.

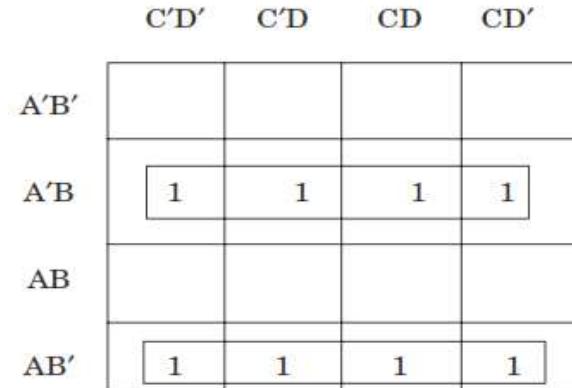


Figure 5.14(b) Karnaugh map for X.

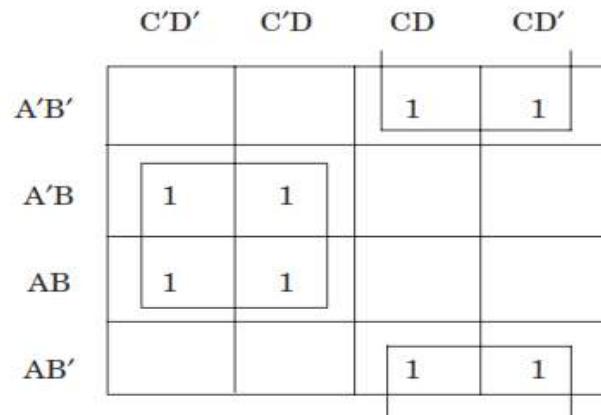


Figure 5.14(c) Karnaugh map for Y.

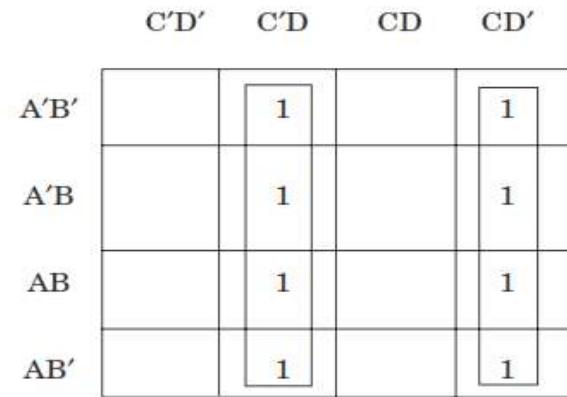


Figure 5.14(d) Karnaugh map for Z.

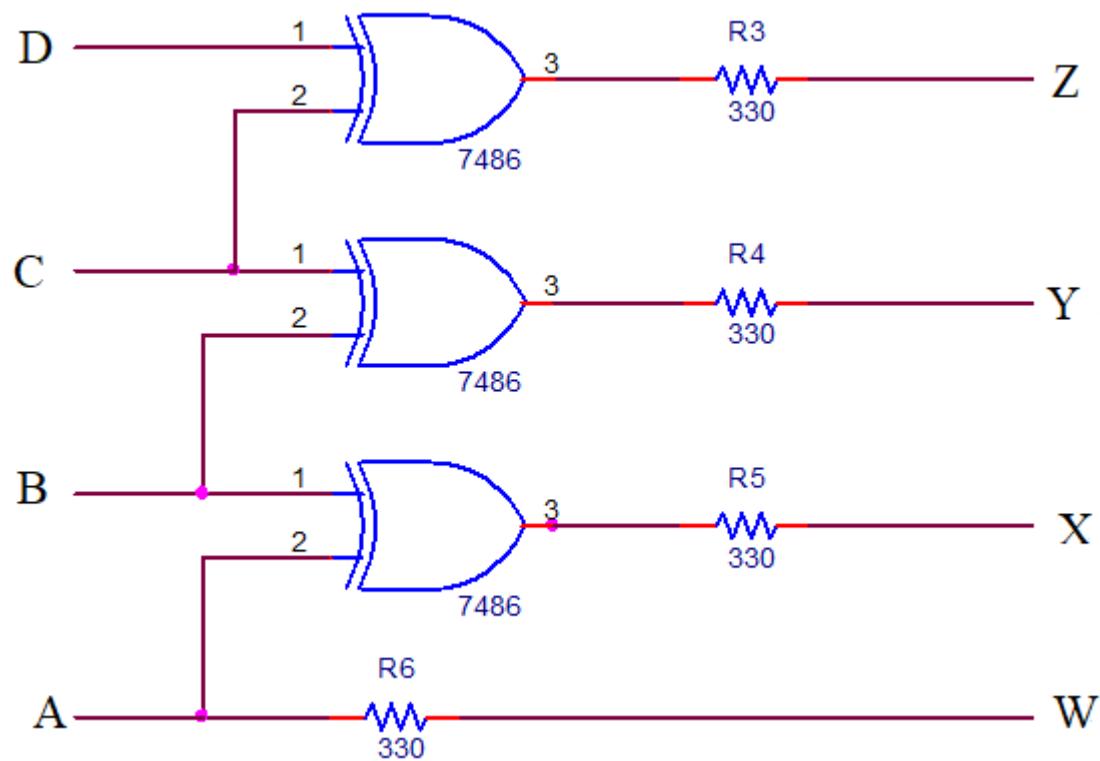
$$W = A,$$

$$Y = BC' + B'C = B \oplus C, \quad \text{and}$$

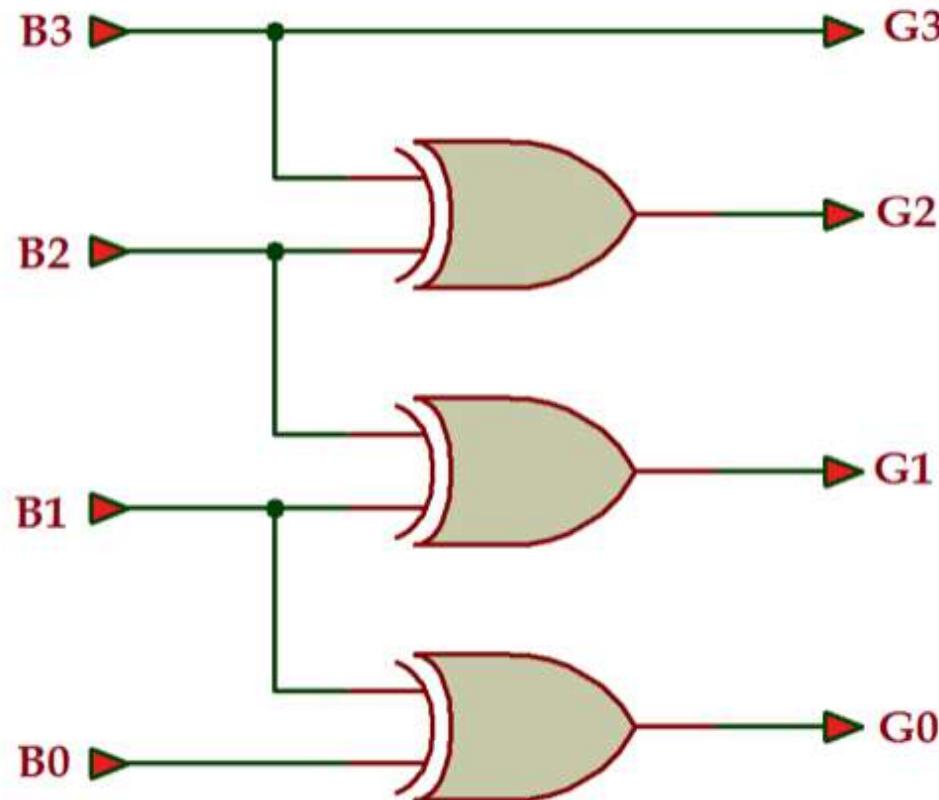
$$X = A'B + AB' = A \oplus B,$$

$$Z = C'D + CD' = C \oplus D.$$

BINARY TO GRAY CODE CONVERTER

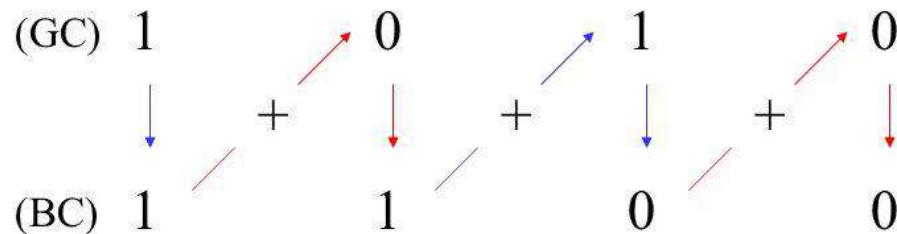


BINARY TO GRAY CODE CONVERTER



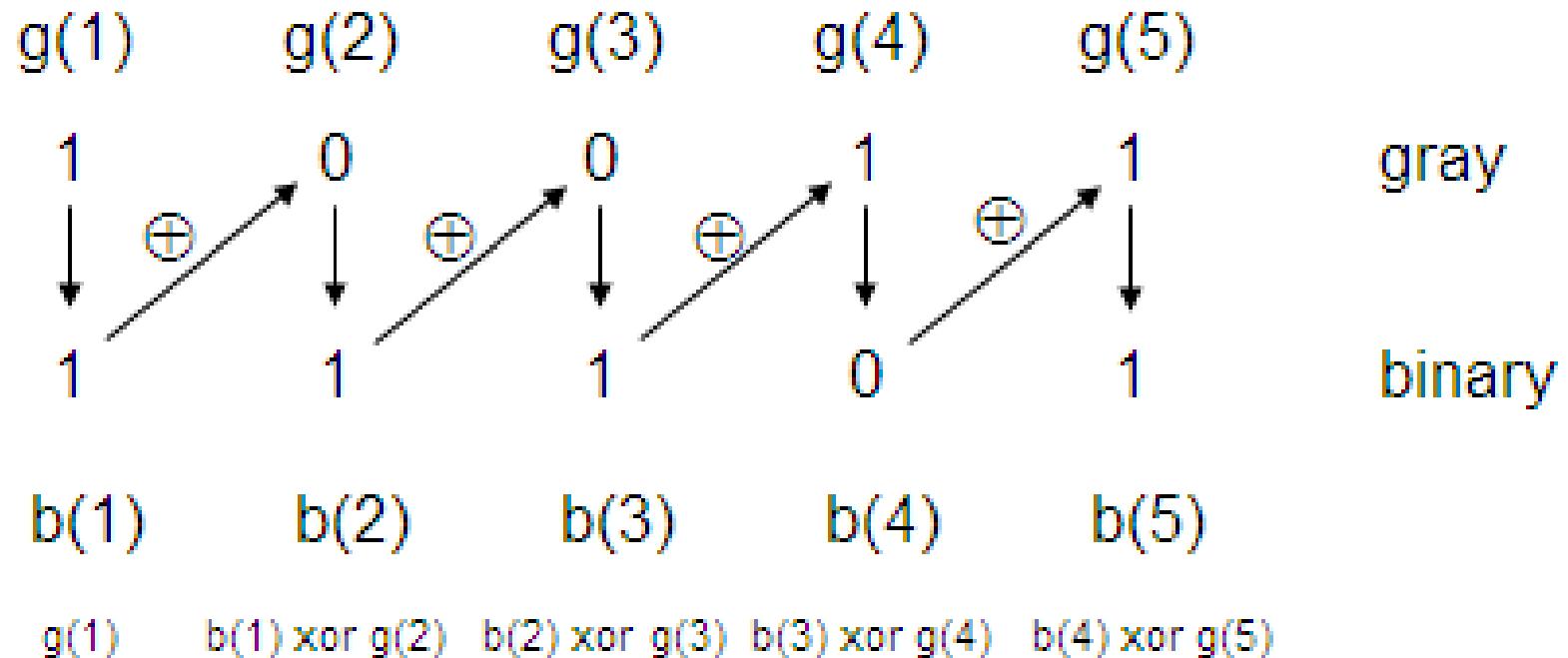
GRAY CODE TO BINARY CONVERTER

Gray to Binary Code Conversion



- MSB does not change as a result of conversion
- Start with MSB of binary number and add it to the second MSB of the Gray code to get the next binary bit
- Repeat for subsequent binary coded bits

GRAY CODE TO BINARY CONVERTER



GRAY CODE TO BINARY CONVERTER

INPUT				OUTPUT			
W _(MSB)	X	Y	Z _(MSB)	A _(MSB)	B	C	D _(LSB)
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	1	0	0	1	0
0	0	1	0	0	0	1	1
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
0	1	0	1	0	1	1	0
0	1	0	0	0	1	1	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	1	1	0	1	0
1	1	1	0	1	0	1	1
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	0	0	1	1	1	1	0
1	0	0	0	1	1	1	1

K-Maps for Binary code output

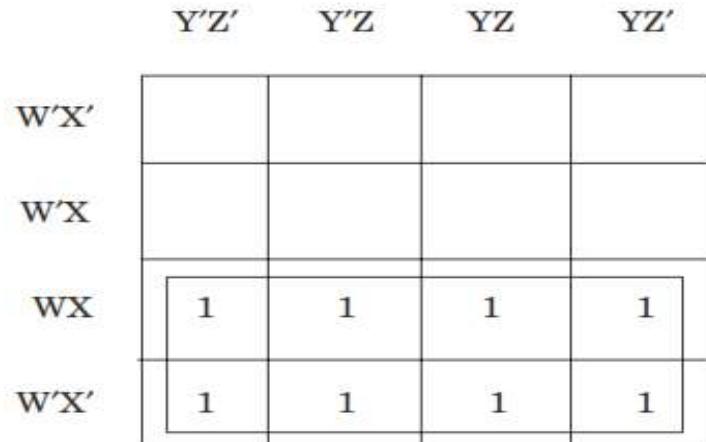


Figure 5.16(a) Karnaugh map for A.

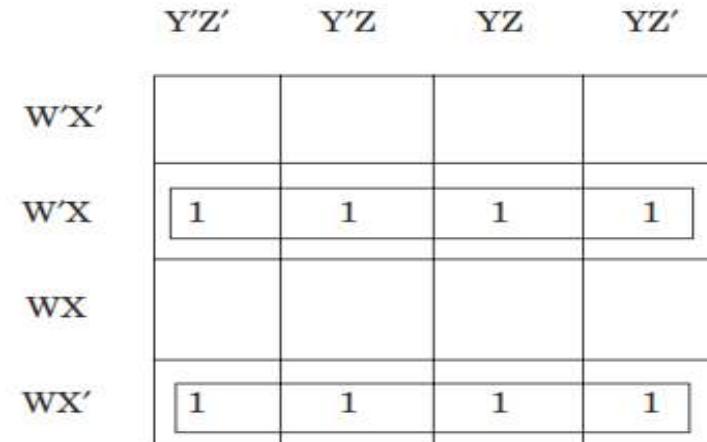


Figure 5.16(b) Karnaugh map for B.

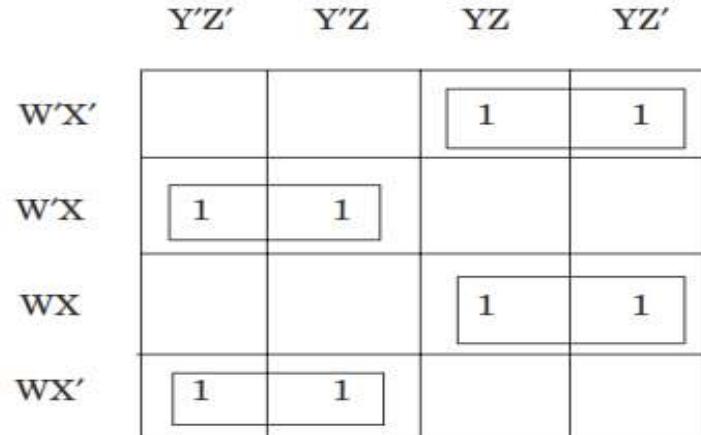


Figure 5.16(c) Karnaugh map for C.

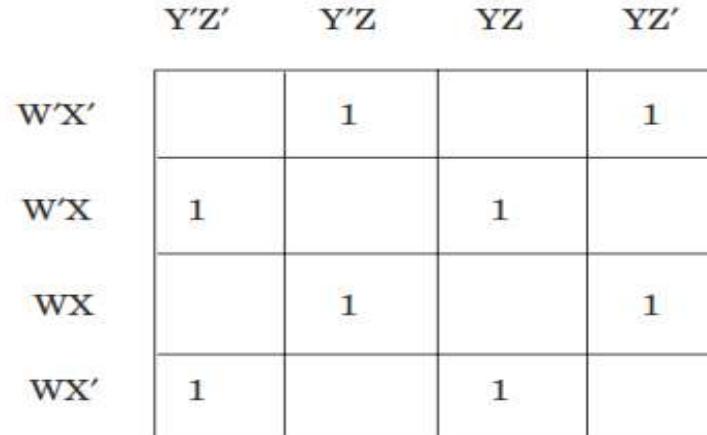


Figure 5.16(d) Karnaugh map for D.

Boolean Expressions for Binary codes

$$A = W$$

$$B = W'X + WX' = W \oplus X$$

$$C = W'X'Y + W'XY' + WXY + WX'Y'$$

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

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

$$= W \oplus X \oplus Y$$

or, $C = B \oplus Y$

$$D = W'X'Y'Z + W'X'YZ' + W'XY'Z' + W'XYZ + WXY'Z + WXYZ' + WX'Y'Z' + WX'YZ$$

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

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

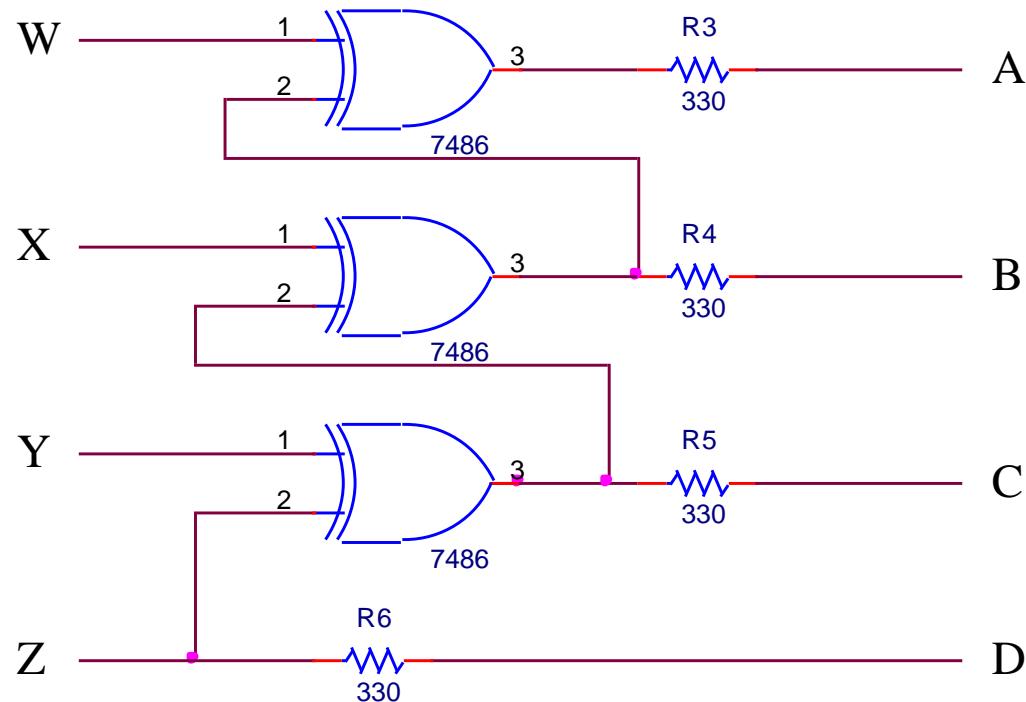
$$= (W'X + WX')(Y \oplus Z)' + (W'X' + WX)(Y \oplus Z)$$

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

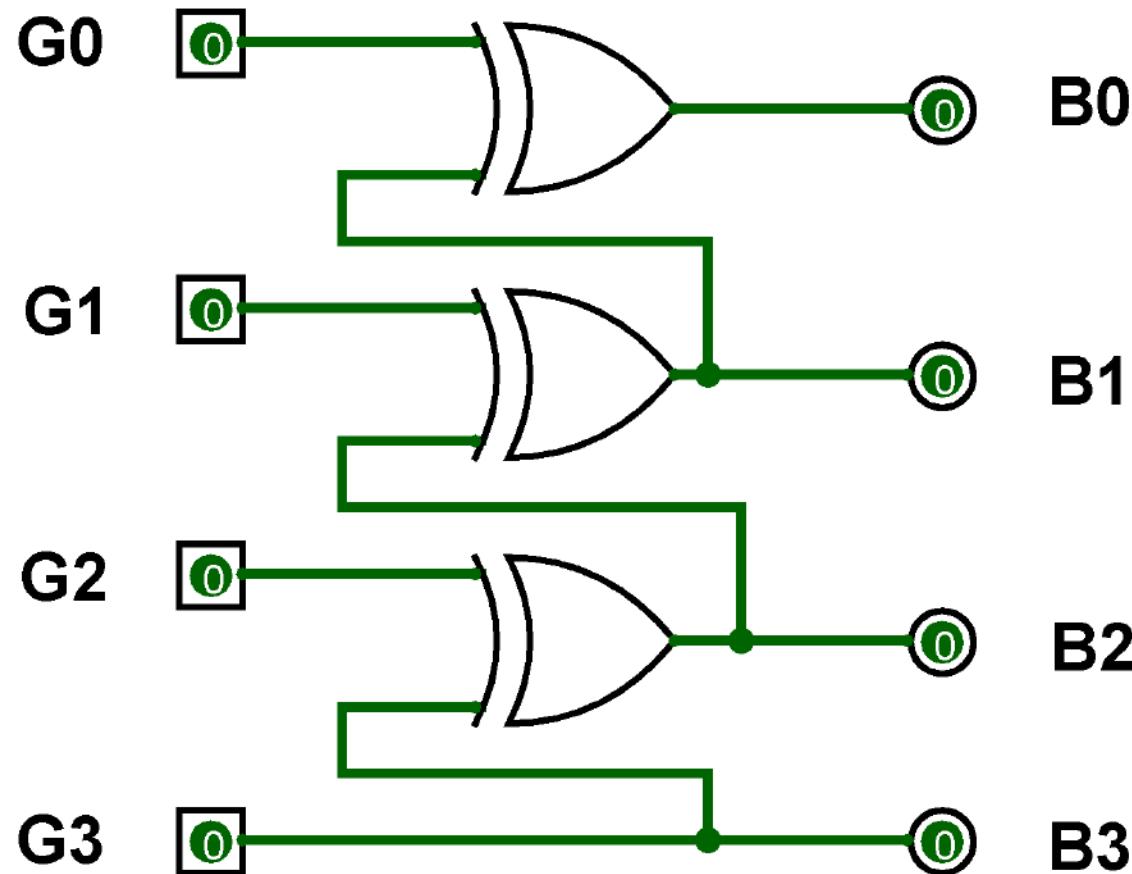
$$= W \oplus X \oplus Y \oplus Z$$

or, $D = C \oplus Z.$

GRAY CODE TO BINARY CONVERTER



GRAY CODE TO BINARY CONVERTER



Example: Find the squares of 3-bit numbers.

<i>Input variables</i>				<i>Output variables</i>						
<i>Decimal Equivalent</i>	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>Decimal Equivalent</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>
0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	0	0	1
2	0	1	0	4	0	0	0	1	0	0
3	0	1	1	9	0	0	1	0	0	1
4	1	0	0	16	0	1	0	0	0	0
5	1	0	1	25	0	1	1	0	0	1
6	1	1	0	36	1	0	0	1	0	0
7	1	1	1	49	1	1	0	0	0	1

	$Y'Z'$	$Y'Z$	YZ	YZ'
X'				
X			1	1

Figure 5.30(a) Karnaugh map for A.

	$Y'Z'$	$Y'Z$	YZ	YZ'
X'			1	
X		1		

Figure 5.30(c) Karnaugh map for C.

	$Y'Z'$	$Y'Z$	YZ	YZ'
X'	0	0	0	0
X	0	0	0	0

Figure 5.30(e) Karnaugh map for E.

	$Y'Z'$	$Y'Z$	YZ	YZ'
X'				
X	1	1	1	

Figure 5.30(b) Karnaugh map for B.

	$Y'Z'$	$Y'Z$	YZ	YZ'
X'				1
X				1

Figure 5.30(d) Karnaugh map for D.

	$Y'Z'$	$Y'Z$	YZ	YZ'
X'		1	1	
X		1	1	

Figure 5.30(f) Karnaugh map for F.

The Boolean expressions of the output variables are

$$A = XY$$

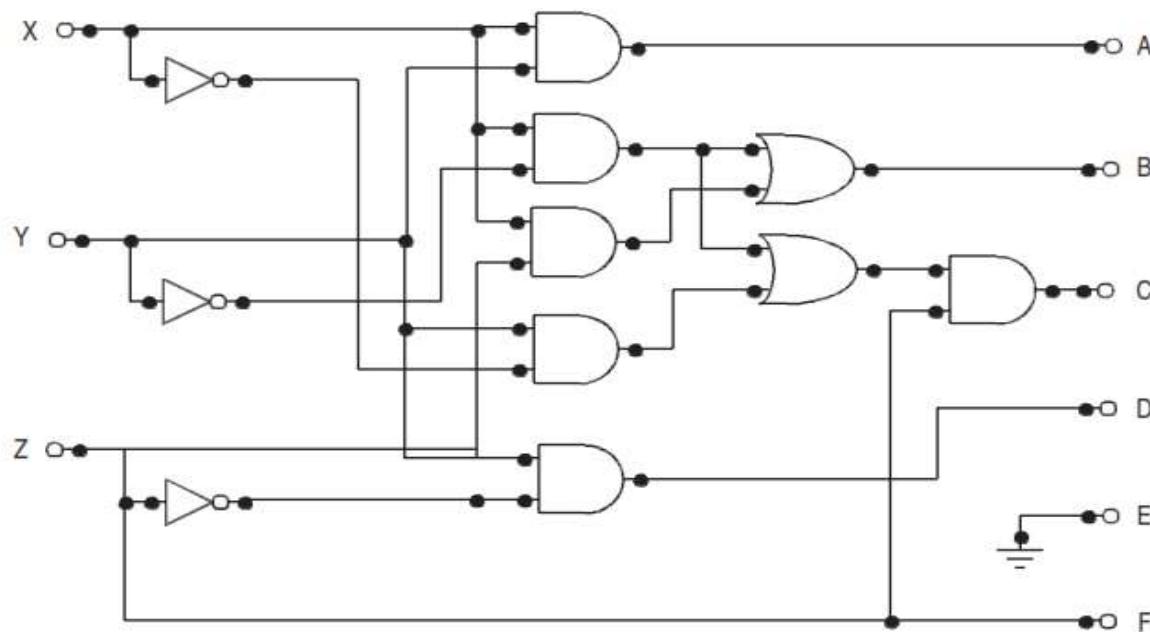
$$B = XY' + XZ$$

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

$$D = YZ'$$

$$E = 0 \quad \text{and}$$

$$F = Z.$$



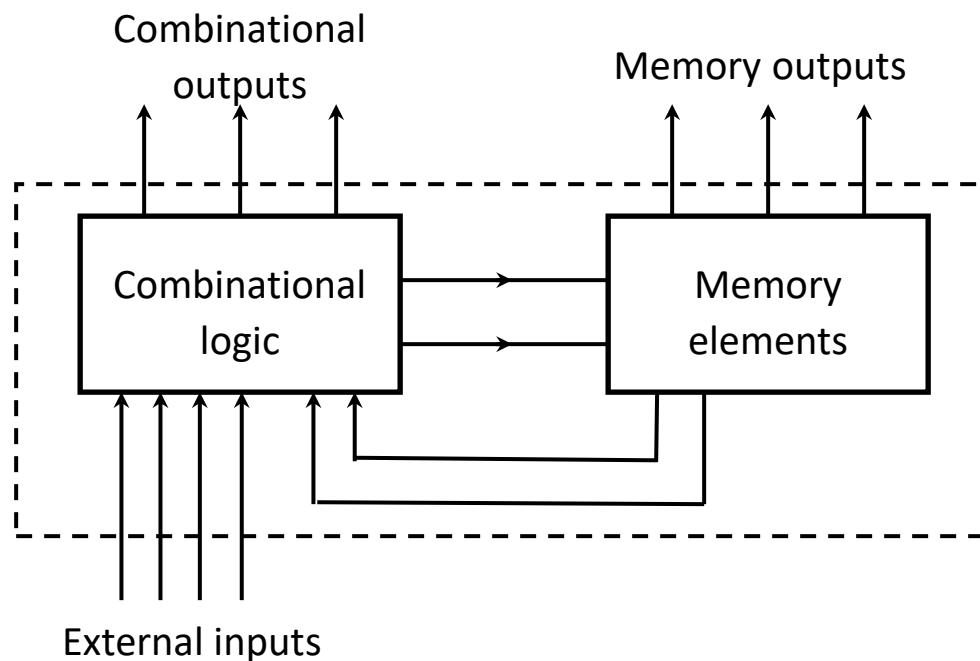
MODULE-5

Sequential Logic Latches & Flip-flops

- Introduction
- Memory Elements
- Pulse-Triggered Latch
 - ❖ S-R Latch
 - ❖ Gated S-R Latch
 - ❖ Gated D Latch
- Edge-Triggered Flip-flops
 - ❖ S-R Flip-flop
 - ❖ D Flip-flop
 - ❖ J-K Flip-flop
 - ❖ T Flip-flop

Introduction

- A **sequential circuit** consists of a *feedback path*, and employs some *memory elements*.



Sequential circuit = Combinational logic + Memory Elements

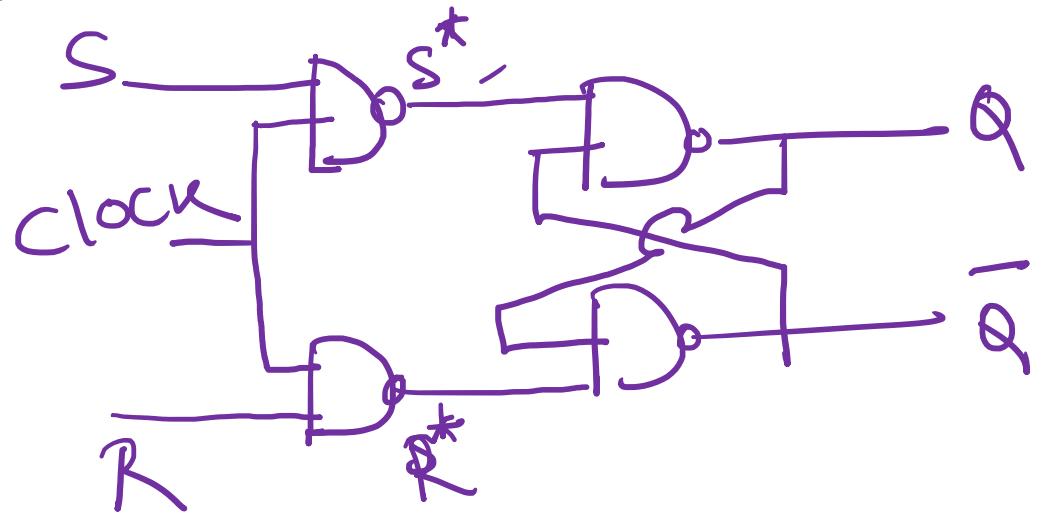
Introduction

- There are two types of sequential circuits:
 - ❖ *synchronous*: outputs change only at specific time
 - ❖ *asynchronous*: outputs change at any time
- *Multivibrator*: a class of sequential circuits. They can be:
 - ❖ *bistable* (2 stable states)
 - ❖ *monostable* or *one-shot* (1 stable state)
 - ❖ *astable* (no stable state)
- Bistable logic devices: *latches* and *flip-flops*.
- Latches and flip-flops differ in the method used for changing their state.

S^*	R^*	Q	\bar{Q}
0	0	Not used	
0	1	1	0
1	0	0	1
1	1		

Memory

SR-FF



$\text{clock} = 0 \& 1$

$$S^* = \overline{S \cdot \text{clk}} = \overline{S} + \overline{\text{clk}} = 1$$

$$R^* = 1$$

case 2 : when $\text{clk} = 1$

$$\begin{cases} S^* = \overline{S} \\ R^* = \overline{R} \end{cases}$$

Truth Table for SR FF

clk	S	R	Q	\bar{Q}
(low)	0	x x		
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

valid

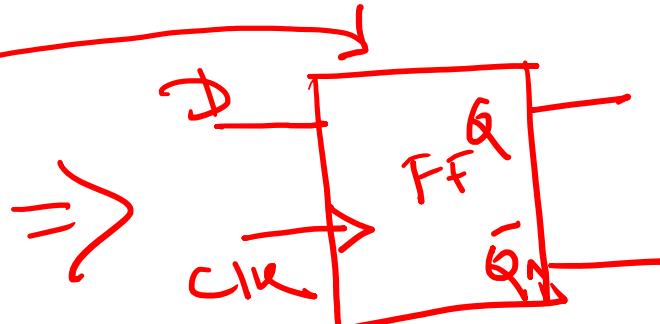
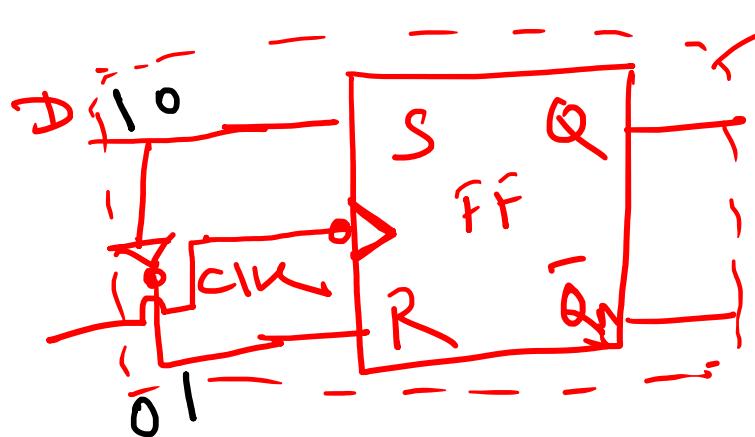
memory \rightarrow Restore
memory \rightarrow "

Not used

Truth table for SR FF :

CIN	S	R	Q _{n+1} → Next State	Q _n → Present state
0	X	X		
1	0	0	Q _n	
1	0	1	0 ✓	X
1	1	0	1	
1	1	1	X (Invalid)	

D-latch / D - FF



TT for SR

Clk	SR	Q_{n+1}
0	xx	Q_n memor
1	00	0
	01	1
	10	x
	11	x

TT for DFF

Clk	D	Q_{n+1}
		Q_n
0	x	0
1	0	1

characteristic Table of D FF

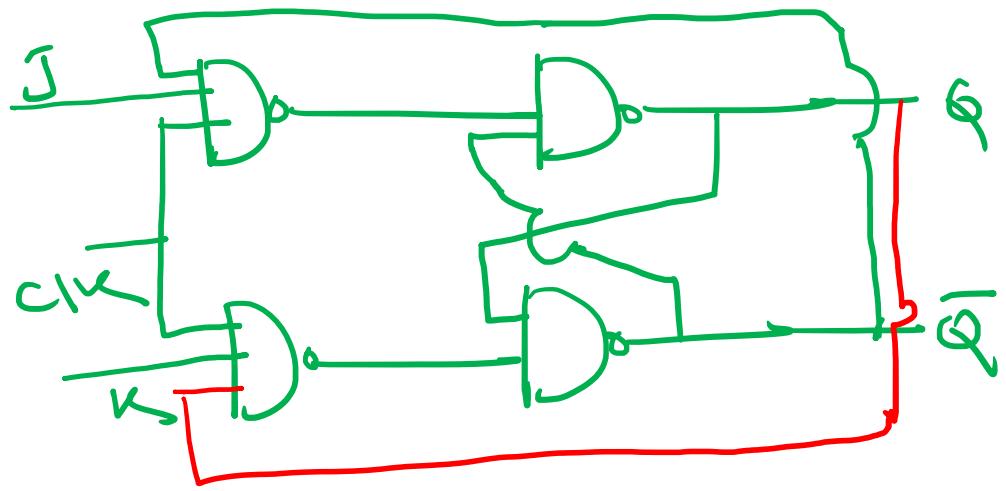
Q_n	D	Q_{n+1}
0	0	0
0	1	1
1	0	0
1	1	1

$$Q_{n+1} = D$$

Excitation Table of D FF

Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

J K Flip Flop



Characteristic: Table for JK

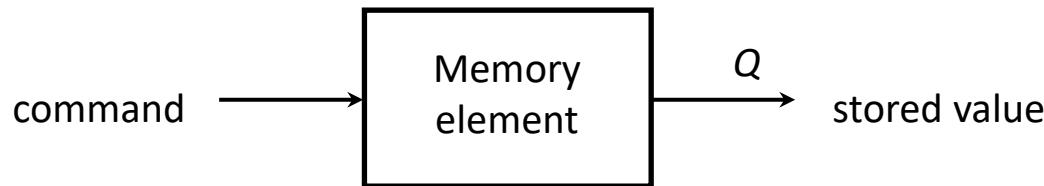
On J u		Qnt
0	0 0	
0	0 1	
0	1 0	
0	1 1	
1	0 0	
1	0 1	
1	1 0	
1	1 1	

Excitation Table

On Qnt		J u
0	0	
0	1	
1	0	
1	1	

Memory Elements

- **Memory element:** a device which can remember value indefinitely, or change value on command from its inputs.



- **Characteristic table:**

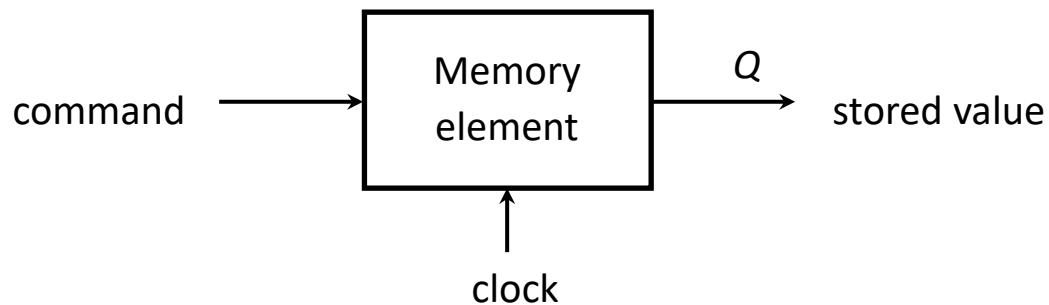
Command (at time t)	$Q(t)$	$Q(t+1)$
Set	X	1
Reset	X	0
Memorise / No Change	0	0
	1	1

$Q(t)$: current state

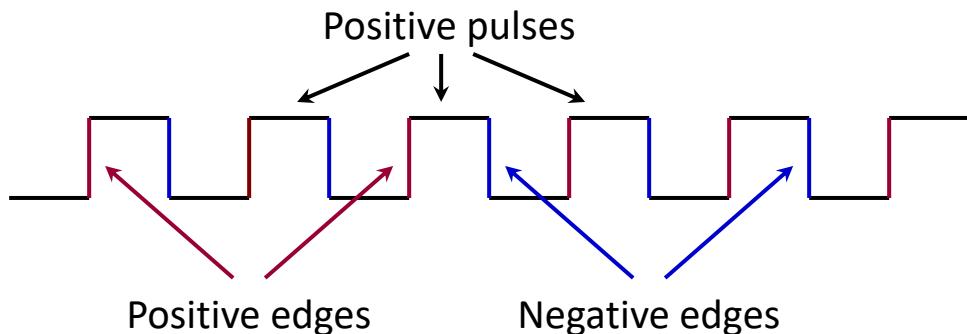
$Q(t+1)$ or Q^+ : next state

Memory Elements

- Memory element with clock. Flip-flops are memory elements that change state on clock signals.



- Clock is usually a square wave.



Memory Elements

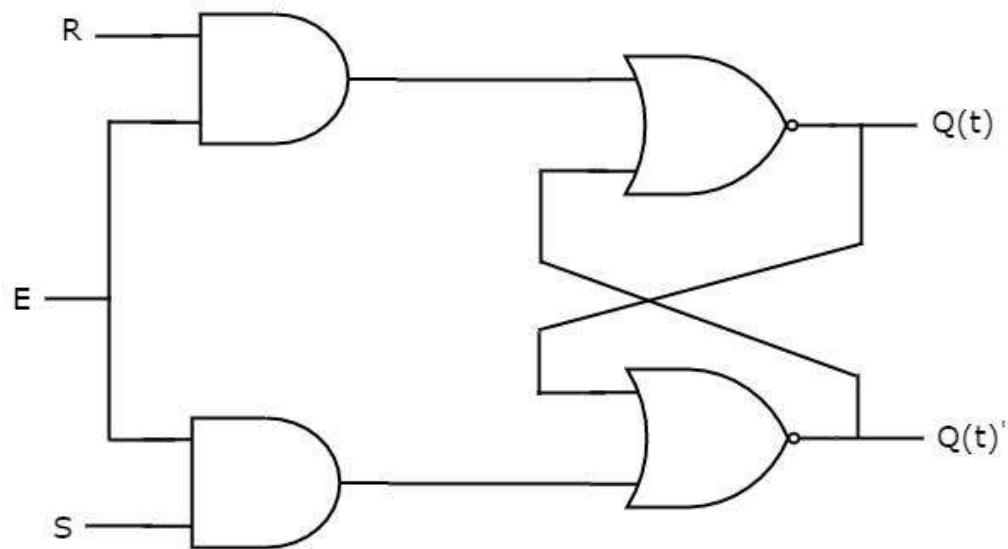
- Two types of triggering/activation:
 - ❖ pulse-triggered
 - ❖ edge-triggered
- Pulse-triggered
 - ❖ latches
 - ❖ ON = 1, OFF = 0
- Edge-triggered
 - ❖ flip-flops
 - ❖ positive edge-triggered (ON = from 0 to 1; OFF = other time)
 - ❖ negative edge-triggered (ON = from 1 to 0; OFF = other time)

INTRODUCTION

- There are two types of memory elements based on the type of triggering that is suitable to operate it.
- Latches
- Flip-flops
- Latches operate with enable signal, which is **level sensitive**. Whereas, flip-flops are edge sensitive.

SR Latch

- SR Latch is also called as **Set Reset Latch**. This latch affects the outputs as long as the enable, E is maintained at '1'.

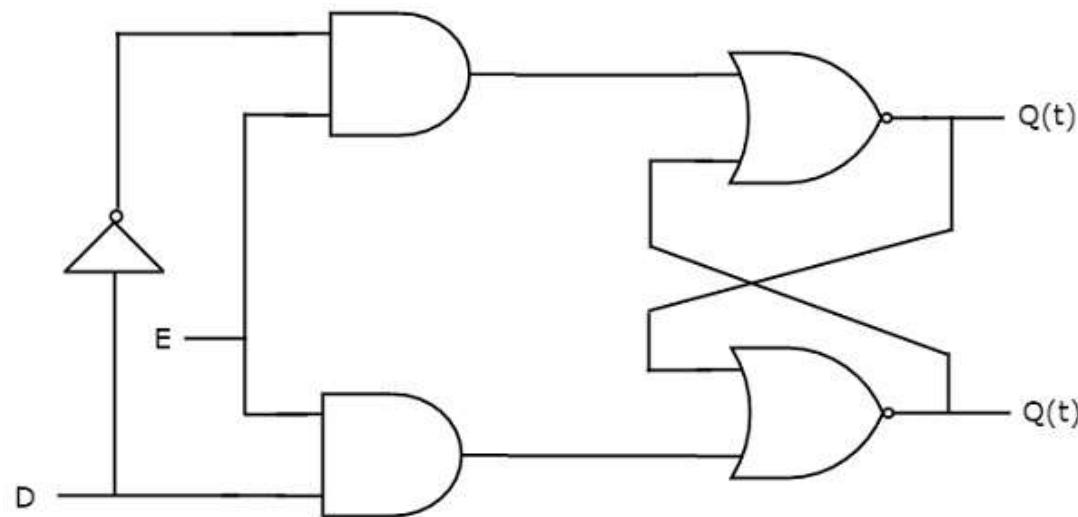


state table

S	R	Q_{t+1}
0	0	Q_t
0	1	0
1	0	1
1	1	x

D or Data Latch

There is one drawback of SR Latch. That is the next state value can't be predicted when both the inputs S & R are one. So, we can overcome this difficulty by D Latch. It is also called as Data Latch.



state table

D	Q _{t+1}
0	0
1	1

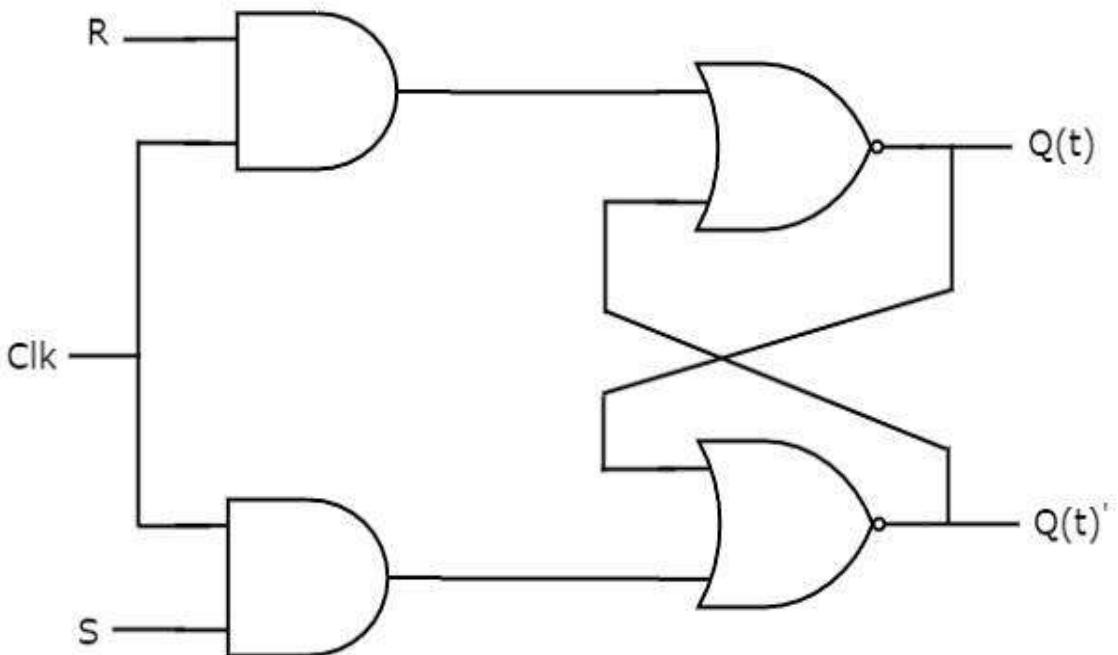
Flip-Flops

- SR Flip-Flop
- D Flip-Flop
- JK Flip-Flop
- T Flip-Flop
- SR flip-flop operates with only positive clock transitions or negative clock transitions. Whereas, SR latch operates with enable signal.

state table

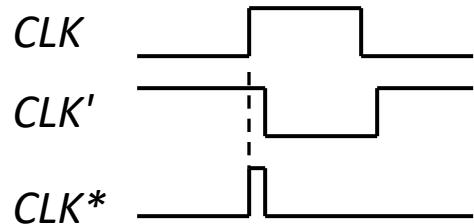
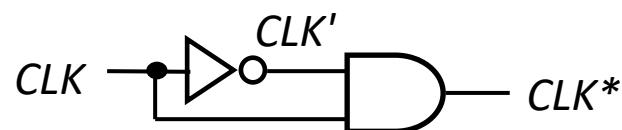
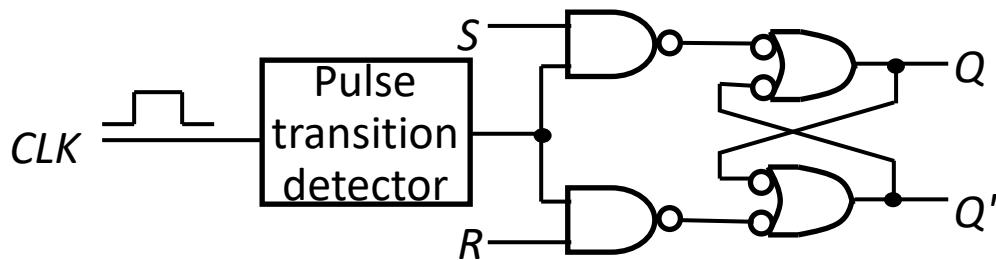
S	R	Qt+1
0	0	Qt
0	1	0
1	0	1
1	1	Invalid

- SR Flip-Flop

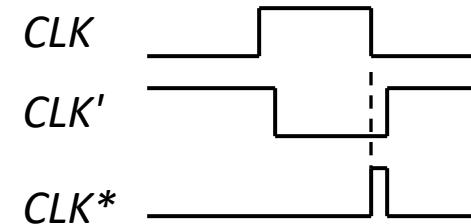
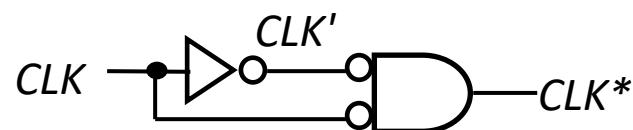


S-R Flip-flop

The pulse transition detector.



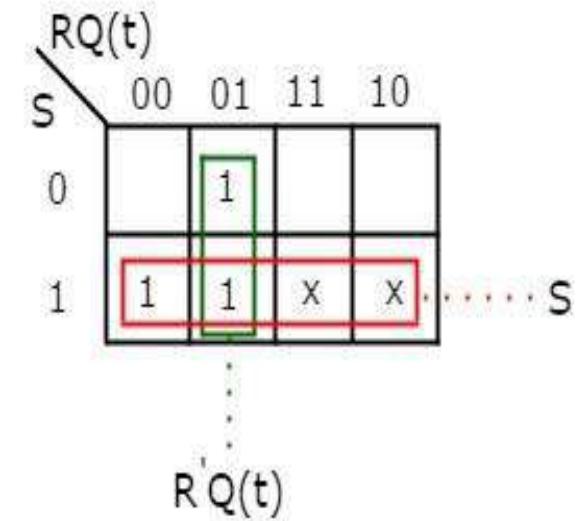
Positive-going transition
(rising edge)



Negative-going transition
(falling edge)

Characteristic Table :SR flip-flop

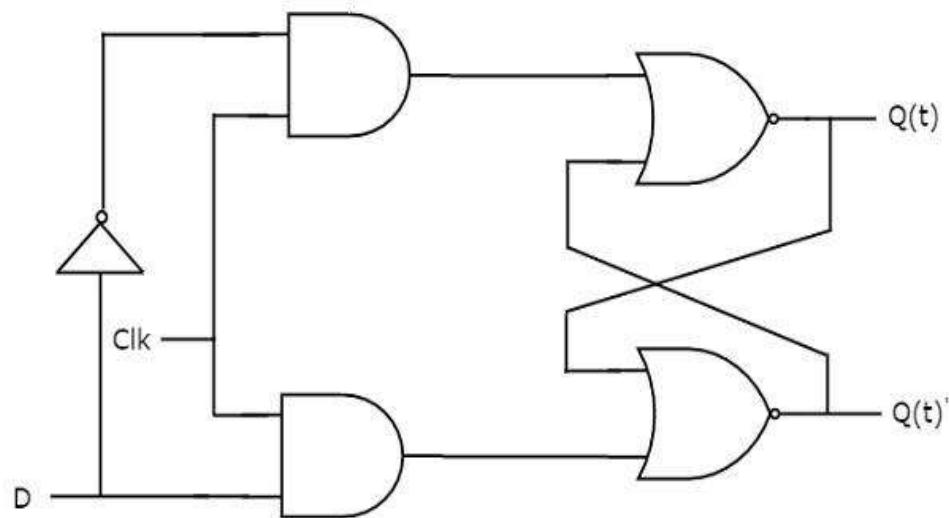
Present Inputs		Present State	Next State
S	R	Q_t	Q_{t+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	x
1	1	1	x



$$Q(t+1) = S + R'Q(t)$$

D Flip-Flop

- D flip-flop operates with only positive clock transitions or negative clock transitions.
- D latch operates with enable signal.
- The output of D flip-flop is insensitive to the changes in the input, D except for active transition of the clock signal.



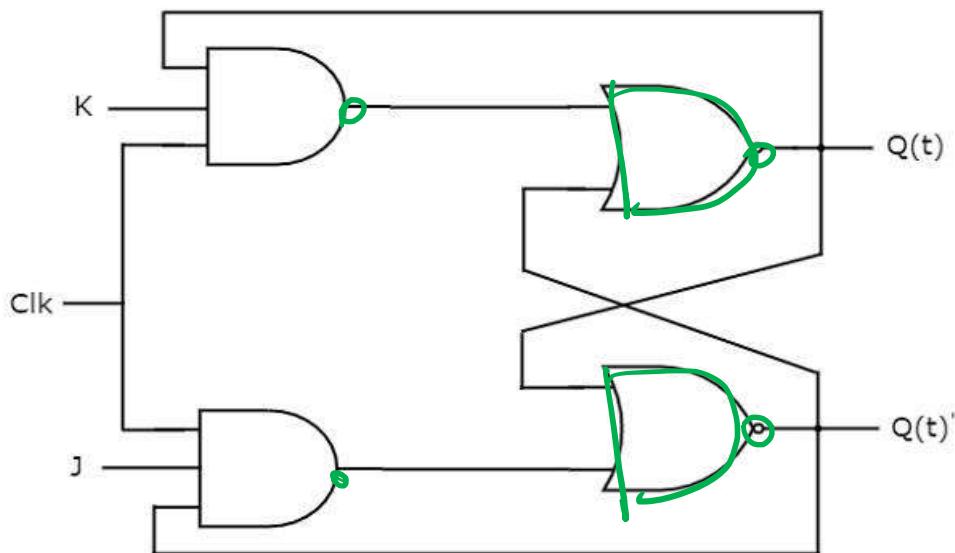
state table of D flip-flop

D	Qt + 1
0	0
1	1

$$Qt+1 = D$$

JK Flip-Flop

- JK flip-flop is the modified version of SR flip-flop. It operates with only positive clock transitions or negative clock transitions.

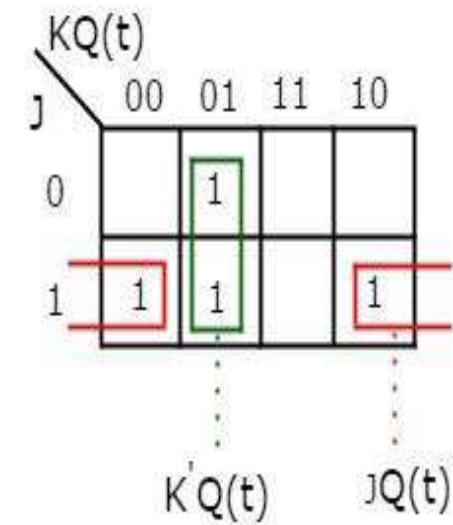


state table of JK flip-flop

J	K	Qt+1
0	0	Qt
0	1	0
1	0	1
1	1	Qt'

Characteristic Table : JK flip-flop

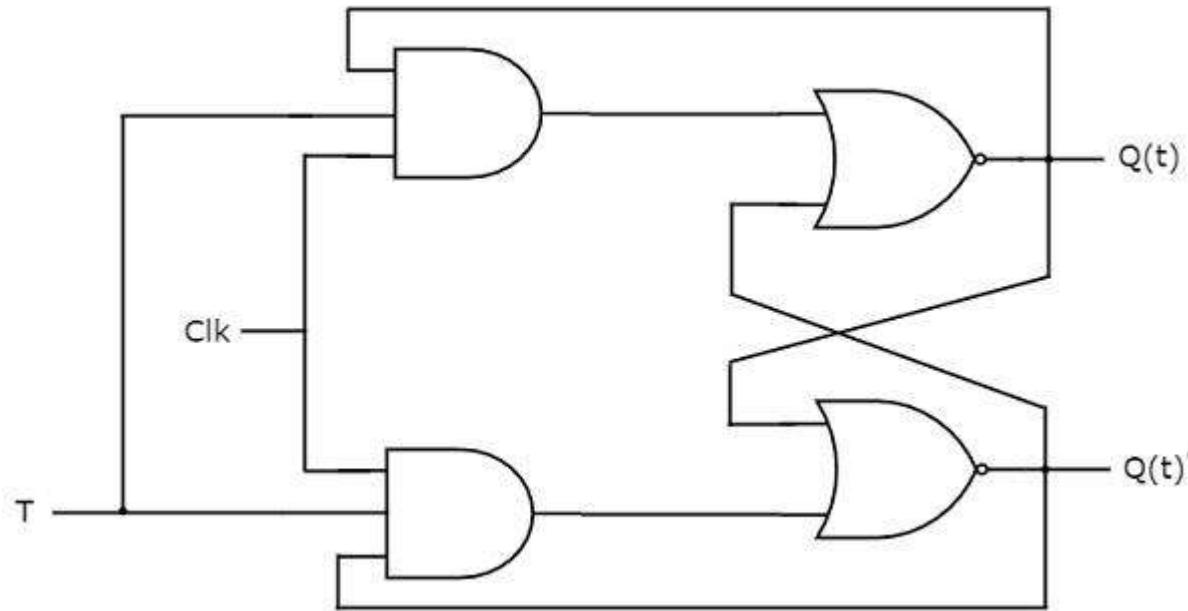
Present Inputs		Present State	Next State
J	K	Q_t	Q_{t+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



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

T Flip-Flop

- T flip-flop is the simplified version of JK flip-flop. It is obtained by connecting the same input 'T' to both inputs of JK flip-flop.



T	Qt+1
0	Qt
1	Qt'

Characteristic Table :T flip-flop

Inputs	Present State	Next State
T	Q _t	Q _{t+1}
0	0	0
0	1	1
1	0	1
1	1	0

$$Q(t+1) = T'Q(t) + TQ(t)'$$

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

5

Steps for flip-flop conversion

- Identify the Available and Desired Flip Flop.
- Prepare a characteristic table for the Desired Flip Flop.
- Prepare the Excitation table for the Available Flip Flop .
- State Boolean expression for the Available Flip Flop.
- Draw the circuit diagram of desired flip-flop according to the simplified expressions using available flip-flop and necessary logic gates.



Excitation Table for all flip-flops

Present State	Next State	SR flip-flop inputs		D flip-flop input	JK flip-flop inputs		T flip-flop input
Q_t	Q_{t+1}	S	R	D	J	K	T
0	0	0	x	0	0	x	0
0	1	1	0	1	1	x	1
1	0	0	1	0	x	1	1
1	1	x	0	1	x	0	0

JK to D FF Conversion

Ext. Table

Step 1:

$$AVI \text{ FF} = JK$$

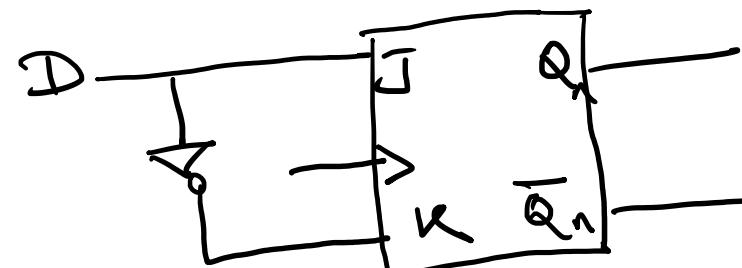
$$Des \text{ FF} = D$$

Characteristic Table
of D FF

Q_n	D	Q_{n+1}	J	K
0	0	0	0	X
0	1	1	1	X
1	0	0	X	1
1	1	1	X	0

Q_n	D	0	1
0	0	1	X
1	X	X	

$$J = D$$



JK to D FF

$\#$	Q_n	Q_{n+1}	$J \& K$
0	0	0	0 X -
0	1	1	1 X
1	0	X	1 1
1	1	0	X 0

K map for J

K map for K

Q_n	D	0	1
0	X	X	
1	1	0	

$$K = D'$$

Convert T FF to D FF

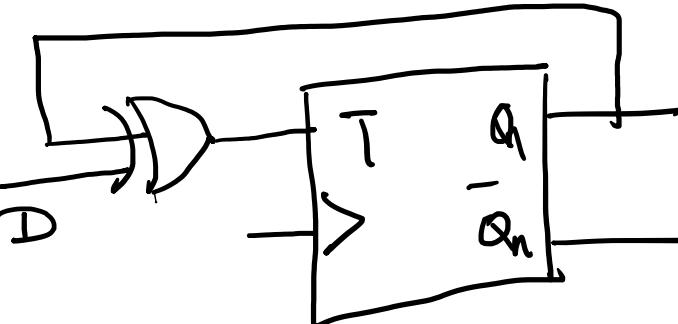
Step 1:

Anal. f^{ff} = T ; Desired = D^{ff}

char. Table for D

Q_n	D	Q_{n+1}	T
0	0	0	0
0	1	1	1
1	0	0	1
1	1	1	0

$\times Q_n$



$$T = Q_n \oplus D$$

Convert SR FF to JK FF

Step I:

$$Avt = SRFF$$

$$DFF = JK$$

Step II

Char. Table of JK FF

Q_n	J	K	Q_{n+1}	S	R
0	0	0	0	0	X
0	0	1	0	0	X
1	0	0	1	1	0
0	1	0	1	1	0
1	0	1	0	0	0
0	1	1	0	X	0
1	1	0	1	0	0
1	1	1	1	0	1

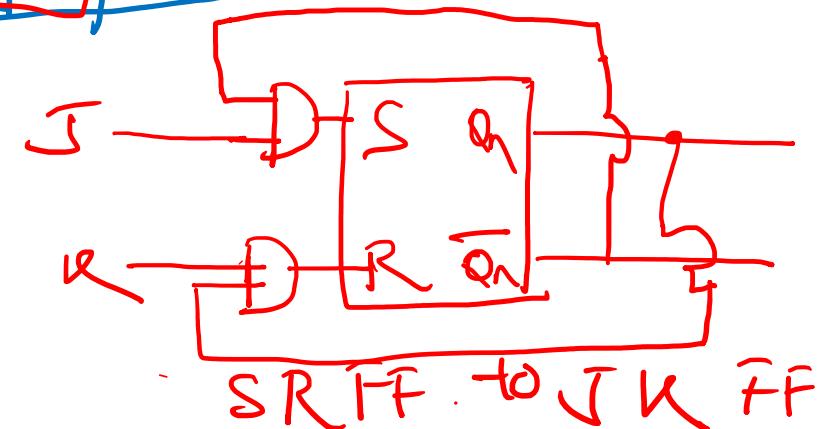
K-map - S

$Q_n \backslash J \backslash K$	00	01	11	10
0	0	0	1	1
1	X	0	0	X

$$S = Q_n \bar{J}$$

$Q_n \backslash J \backslash K$	00	01	11	10
0	X	X	0	0
1	0	1	1	0

$$R = Q_n K$$



SR Flip-Flop to other Flip-Flop Conversions

SR Flip-Flop to other Flip-Flop Conversions

Following are the three possible conversions of SR flip-flop to other flip-flops.

- SR flip-flop to D flip-flop
- SR flip-flop to JK flip-flop
- SR flip-flop to T flip-flop

SR flip-flop to D flip-flop conversion

Characteristic table of D flip-flop

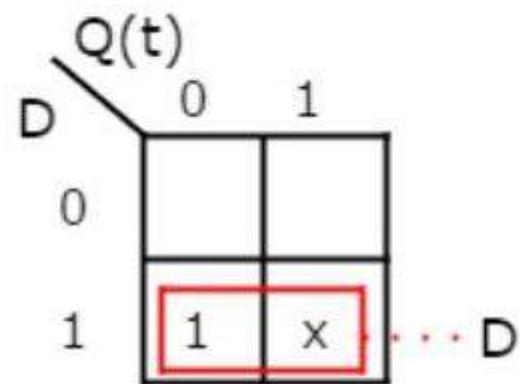
D flip-flop input	Present State	Next State
D	Q_t	Q_{t+1}
0	0	0
0	1	0
1	0	1
1	1	1

Characteristic table of D flip-flop along with the **excitation inputs of SR flip-flop**.

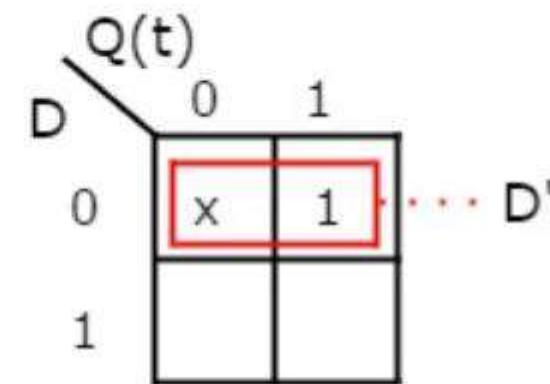
D flip-flop input	Present State	Next State	SR flip-flop inputs	
D	Q_t	Q_{t+1}	S	R
0	0	0	0	x
0	1	0	0	1
1	0	1	1	0
1	1	1	x	0

We can use 2 variable K-Maps for getting simplified expressions for these inputs. The **k-Maps** for S & R are shown below.

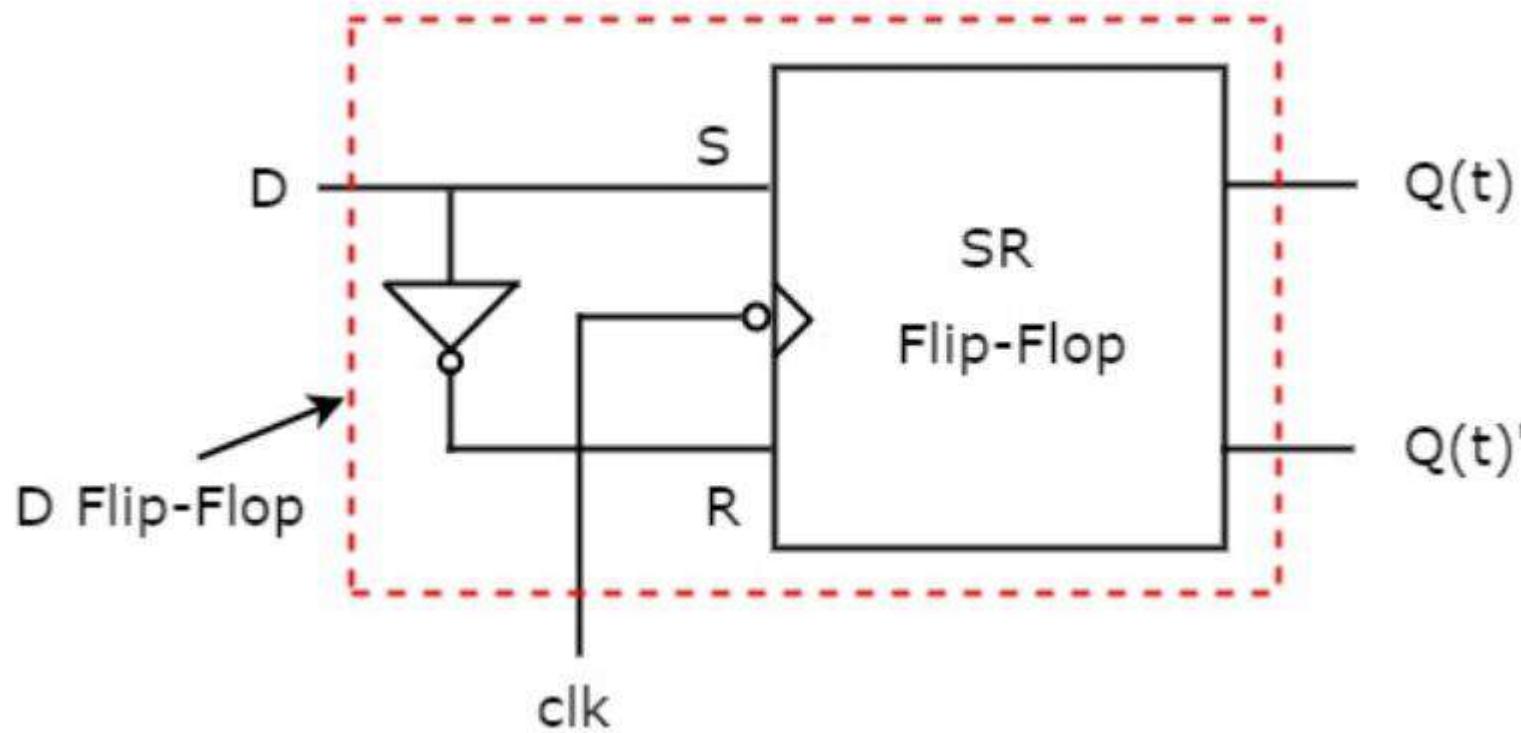
K-Map for S



K-Map for R



So, we got $S = D$ & $R = D'$ after simplifying. The **circuit diagram** of D flip-flop is shown in the following figure.



This circuit consists of SR flip-flop and an inverter. This inverter produces an output, which is complement of input, D. So, the overall circuit has single input, D and two outputs Q_t & Q_t' .

D Flip-Flop to other Flip-Flop Conversions

Following are the three possible conversions of D flip-flop to other flip-flops.

- D flip-flop to T flip-flop
- D flip-flop to SR flip-flop
- D flip-flop to JK flip-flop

D flip-flop to T flip-flop conversion

Characteristic table of T flip-flop

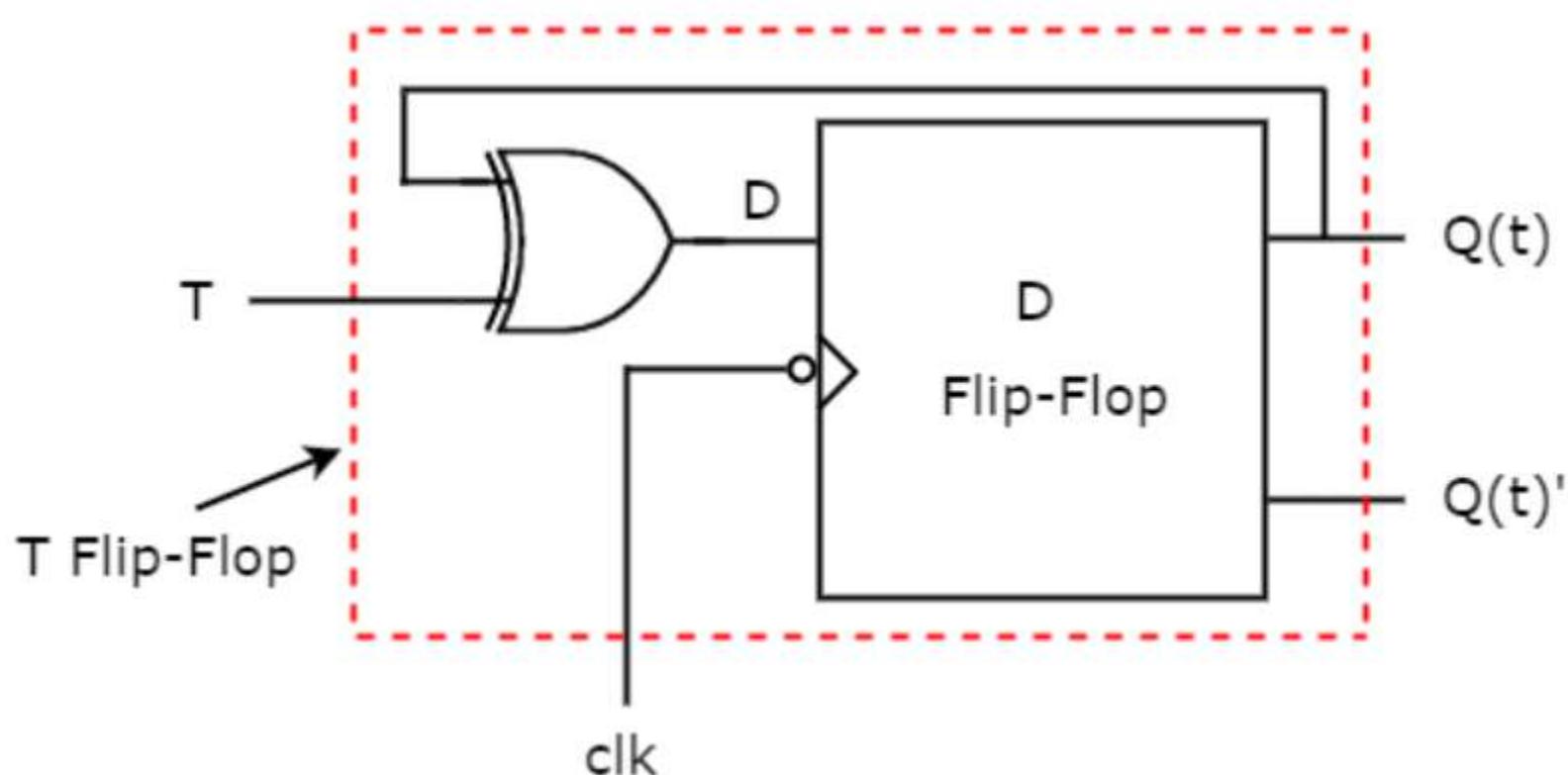
T flip-flop input	Present State	Next State
T	Q_t	Q_{t+1}
0	0	0
0	1	1
1	0	1
1	1	0

Characteristic table of T flip-flop along with the **excitation input of D flip-flop**

T flip-flop input T	Present State Q_t	Next State Q_{t+1}	D flip-flop input D
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

$$D = T \oplus Q(t)$$

The circuit diagram of T flip-flop



JK flip-flop to T flip-flop conversion

Characteristic table of T flip-flop

T flip-flop input	Present State	Next State
T	Q_t	Q_{t+1}
0	0	0
0	1	1
1	0	1
1	1	0

Characteristic table of T flip-flop along with the **excitation inputs of JK flipflop**

T flip-flop input	Present State	Next State	JK flip-flop inputs	
T	Q_t	Q_{t+1}	J	K
0	0	0	0	x
0	1	1	x	0
1	0	1	1	x
1	1	0	x	1

$$J = T \text{ & } K = T$$

K-Map for J

A Karnaugh map for output J. The vertical axis is labeled Q(t) and the horizontal axis is labeled T. The top row is labeled 0 and the bottom row is labeled 1. The left column is labeled 0 and the right column is labeled 1. The cell at the bottom-left (Q(t)=1, T=1) contains a '1' and is highlighted with a red box. The other three cells contain 'x'.

	0	1
0	x	
1	1	x

... T

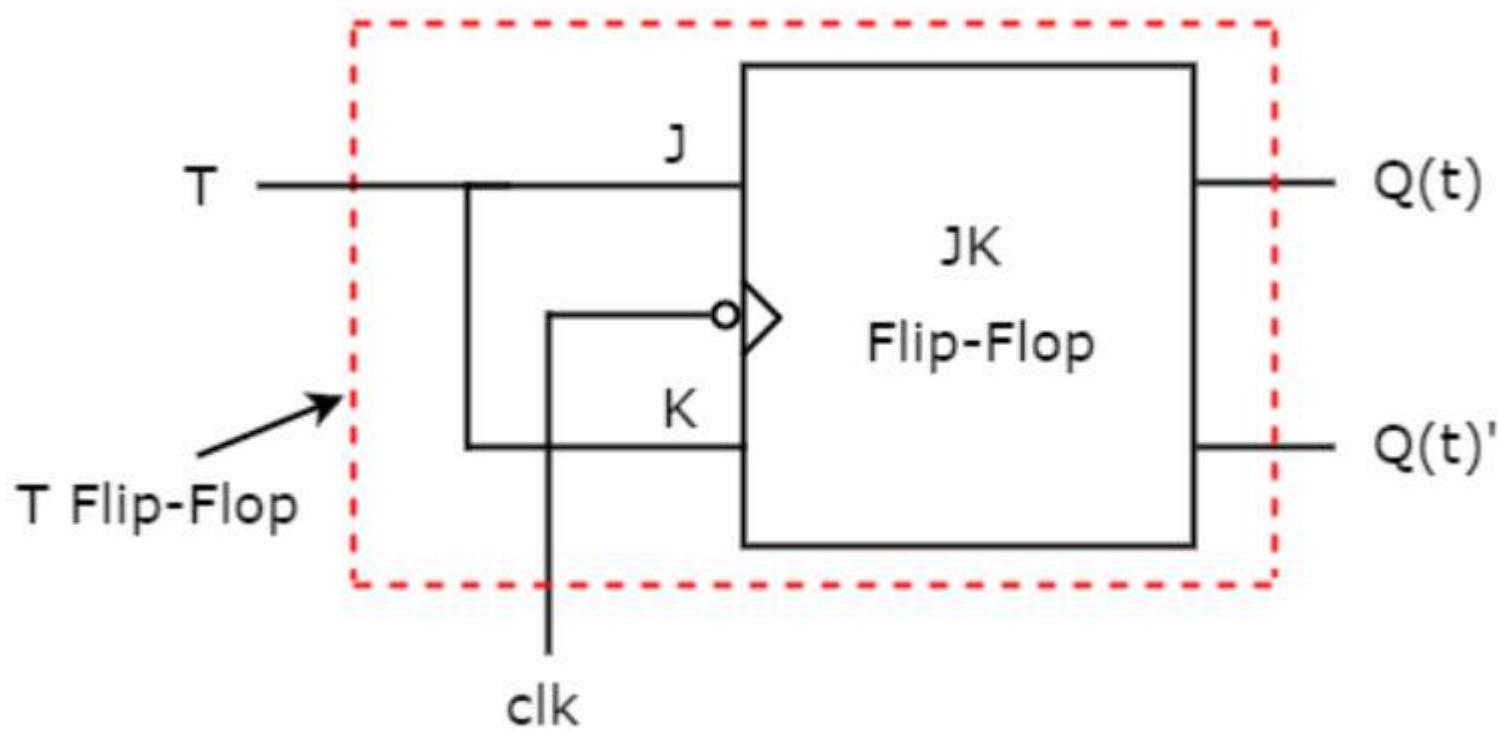
K-Map for K

A Karnaugh map for output K. The vertical axis is labeled Q(t) and the horizontal axis is labeled T. The top row is labeled 0 and the bottom row is labeled 1. The left column is labeled 0 and the right column is labeled 1. The cell at the bottom-left (Q(t)=1, T=1) contains a '1' and is highlighted with a red box. The other three cells contain 'x'.

	0	1
0	x	
1	x	1

... T

$$J = T \text{ & } K = T$$



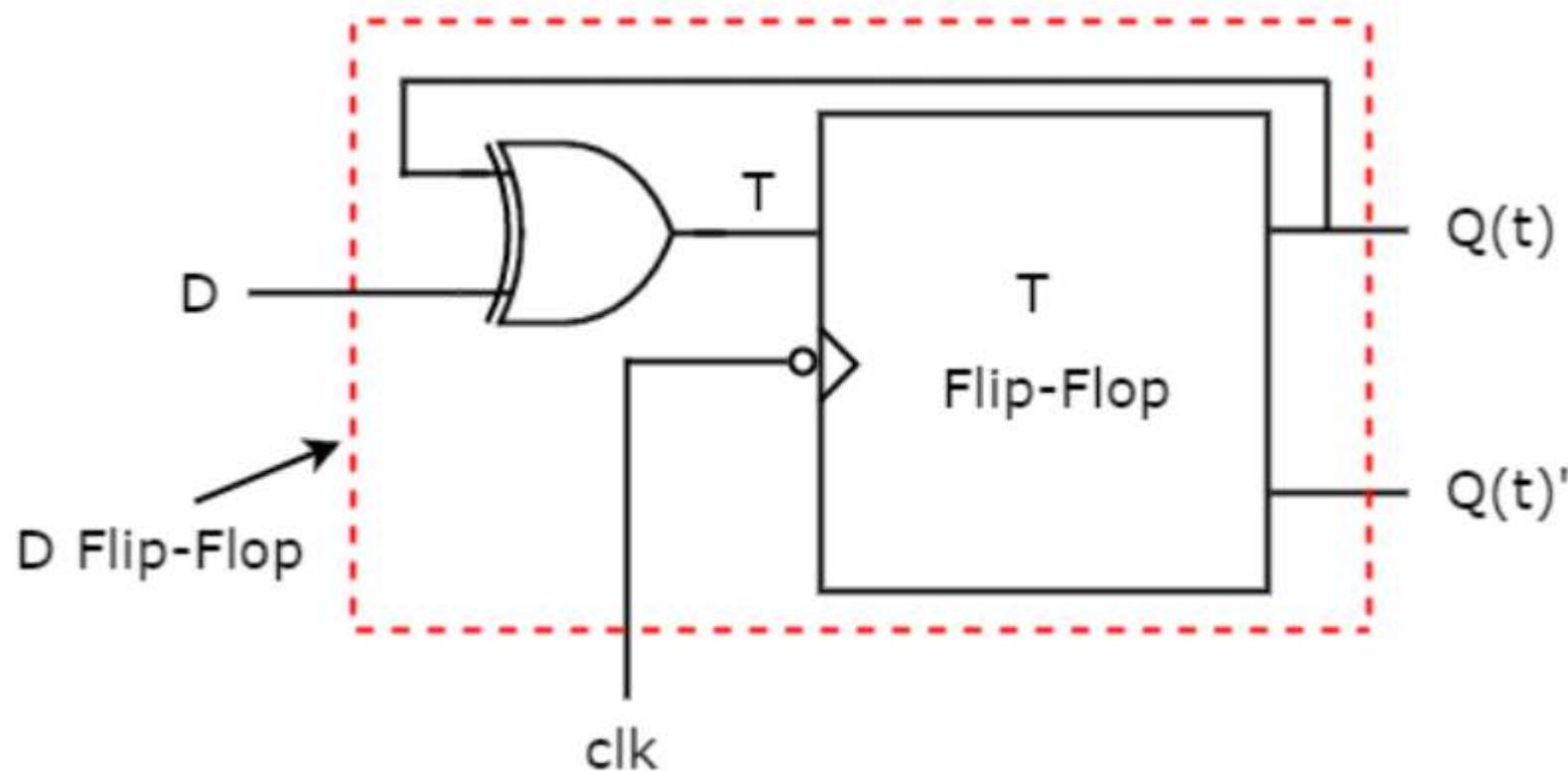
T flip-flop to D flip-flop conversion

Characteristic table of D flip-flop along with the excitation input of T flip-flop.

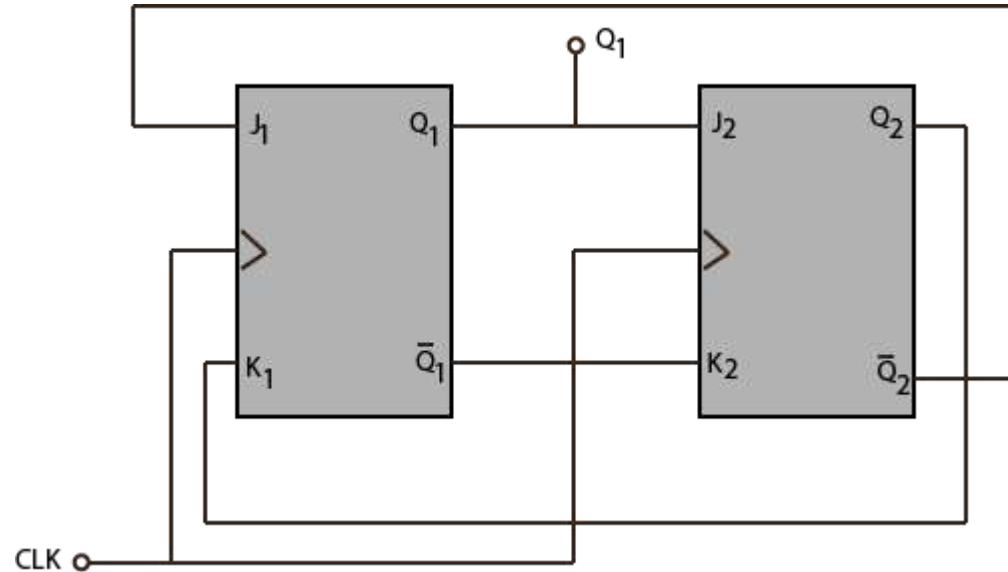
D flip-flop input	Present State	Next State	T flip-flop input
D	Q _t	Q _{t+1}	T
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

$$T = D \oplus Q(t)$$

circuit diagram of D flip-flop



Example: The outputs of the two flip-flops Q1, Q2 in the figure shown are initialized to 0, 0. Determine the sequence generated at Q1 upon application of clock signal.



Solution: Initially $Q_1 = Q_2 = 0$

Truth table of JK:

J K Q_n

0 0 Previous state

0 1 0

1 0 1

1 1 \bar{Q}_n

Case - 1: 1st clock pulse

$J_1 = \bar{Q}_2, K_1 = Q_2$

$Q_1 = 0, \bar{Q}_1 = 1, Q_2 = 0 = K_1, \bar{Q}_2 = 1$

So, $J_1 = 1, K_1 = 0$ and $J_2 = 0, K_2 = 1$

So, $\bar{Q}_1 = 1, \bar{Q}_2 = 0 = J_1$

Case - 2:

$Q_1 = 1, Q_2 = 0$

$\bar{Q}_1 = 0, \bar{Q}_2 = 1$

So, $J_1 = 1, J_2 = 1$

$K_1 = 0, K_2 = 0$

So, $\bar{Q}_1 = 1, \bar{Q}_2 = 1$

Case - 3:

$Q_1 = 1, \bar{Q}_1 = 0, Q_2 = 1, \bar{Q}_2 = 0$

$J_1 = 0, K_1 = 1, J_2 = 1, K_2 = 0$

$\bar{Q}_1 = 0, \bar{Q}_2 = 1$

So, the sequence will be 01100

Concept:

The Truth table of JK flip flop is given by:

J	K	Q_{n+1}	
0	0	Q_n	Hold state
0	1	0	Reset state
1	0	1	Set state
1	1	\bar{Q}_n	Toggle state

C l o c k	Present State				F F Inputs				Next State	
	Q_1	\bar{Q}_1	Q_2	\bar{Q}_2	J_1	K_1	J_2	K_2	Q_1^+	Q_2^+
1	0	1	0	1	1	0	0	1	1	0
2	1	0	0	1	1	0	1	0	1	1
3	1	0	1	0	0	1	1	0	0	1
:	:	:	:	:	:	:	:	:	:	:

Analysis:

Initially, Q_1 and Q_2 are 0, 0.

From the figure:

$$J_1 = \bar{Q}_2, K_1 = Q_2, J_2 = Q_1, K_2 = \bar{Q}_1$$

Hence the sequence generated at Q_1 is 0110...

Registers

Register:

- ❖ A set of flip-flops, possibly with added combinational gates, that perform data-processing tasks.

- ❖ Store and manipulate information in a digital system.

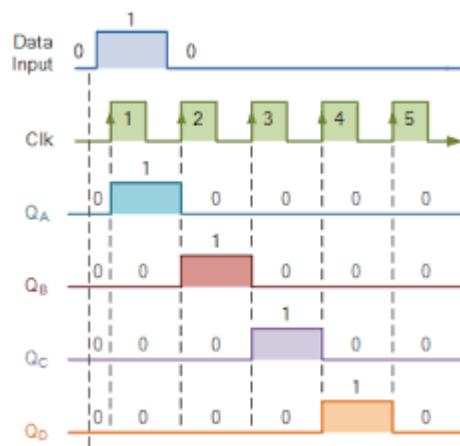
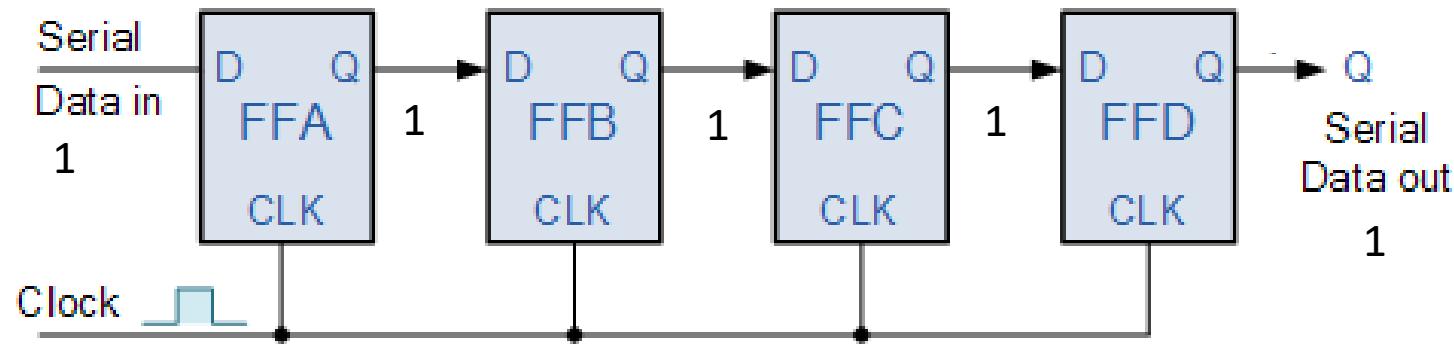
Shift Register

1. Serial In Serial Out (SISO)
2. Serial In Parallel Out (SIPO)
3. Parallel In Serial Out (PISO)
4. Parallel In Parallel Out (PIPO)

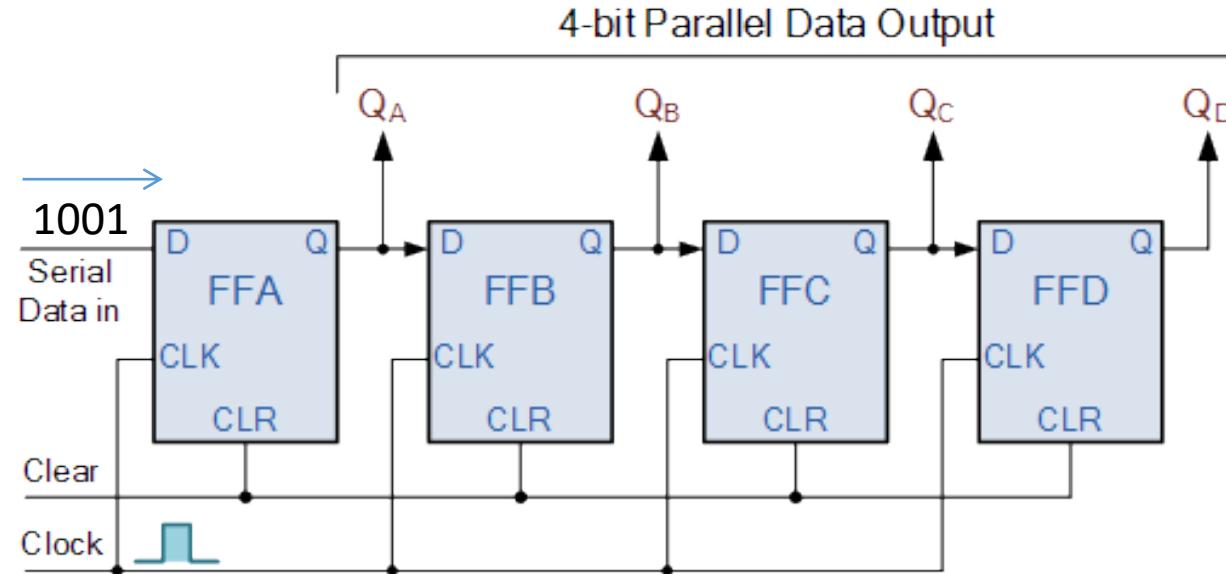
Note:

Serial → Single
Parallel → Many

Serial In Serial Out

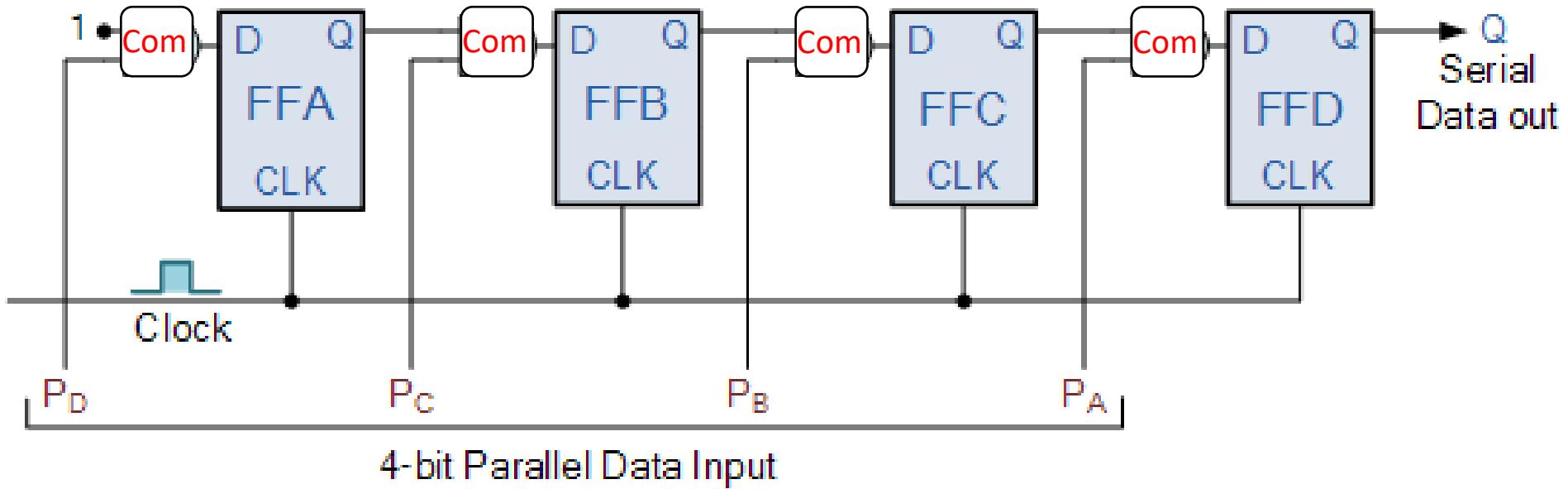


Serial In Parallel Out



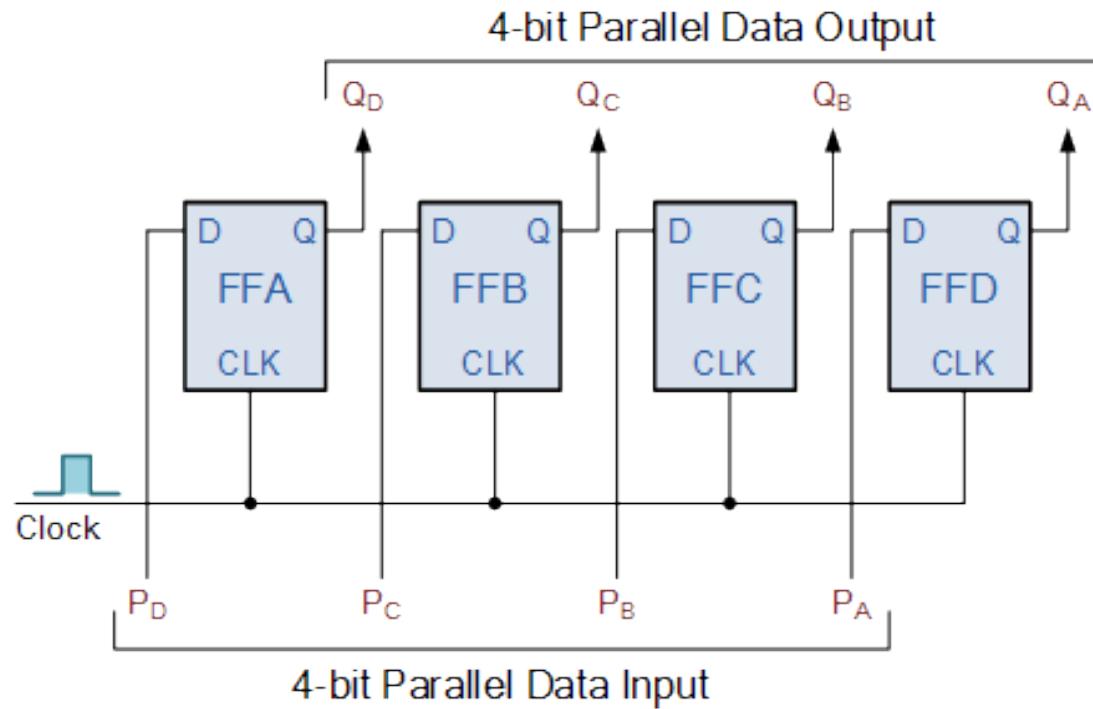
Clear	FF0	FF1	FF2	FF3
1001	0	0	0	0
	1	0	0	0
	0	1	0	0
	0	0	1	0
	1	0	0	1

Parallel In Serial Out



Com → Combinational Circuit (Any)

Parallel In Parallel Out



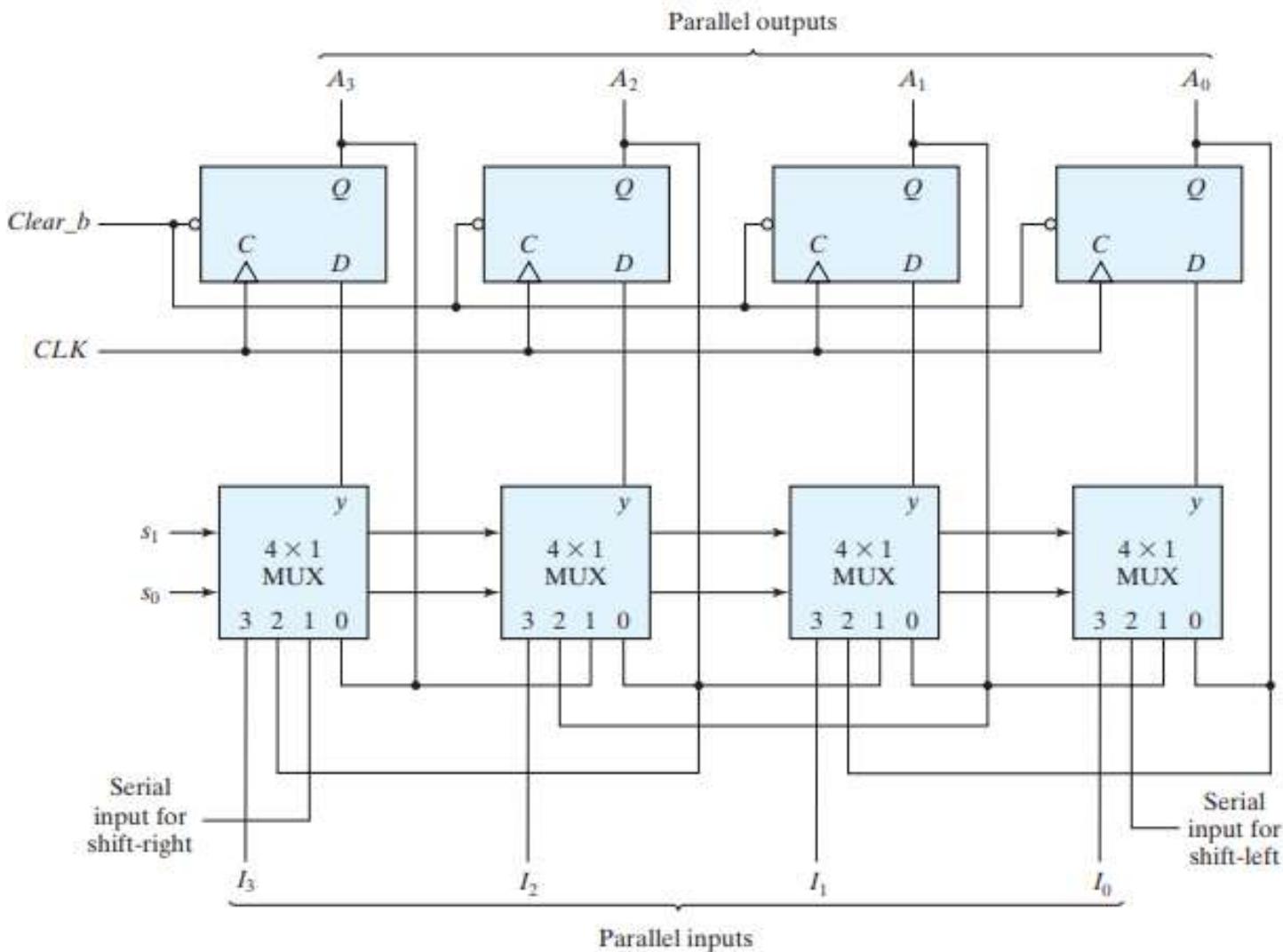
This is ordinary register and it won't perform any shift operation.

Universal Shift Register

Function Table for the Register of Fig. 6.7

Mode Control

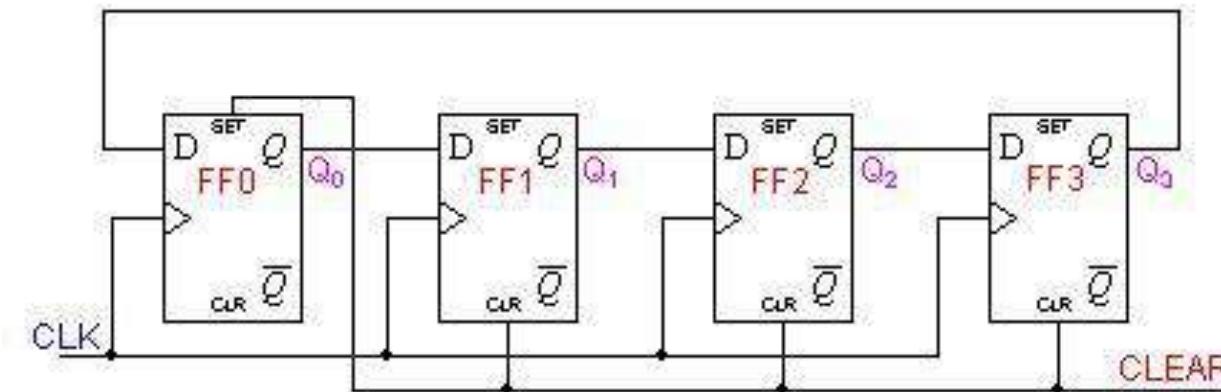
s_1	s_0	Register Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load



Counter

- ❖ The common clock pulse triggers all flip-flops simultaneously
- ❖ Hardware may be more complex but more reliable

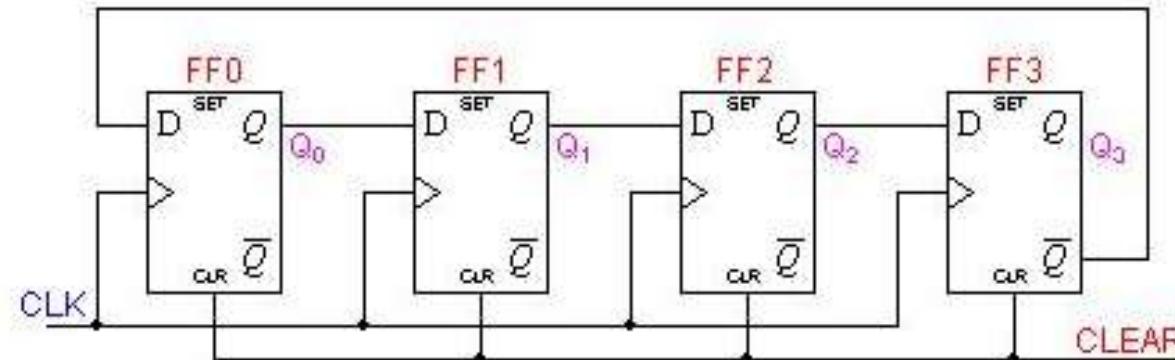
Ring Counter (Using Shift Register)



Clock pulse	Q0	Q1	Q2	Q3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

Also known as mod-4 counter (mod-n counter)

Johnson Counter (Twisted Ring Counter)



Clock pulse	Q0	Q1	Q2	Q3
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

Also known as mod- 2^n counter

BCE102L
DIGITAL SYSTEM DESIGN

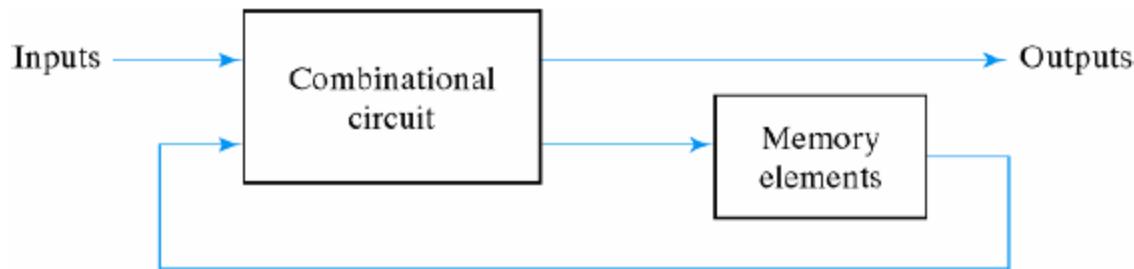
Module:5

**DESIGN OF REGISTERS AND
COUNTERS**

Courtesy: M. Morris Mano, "Digital Design", 3rd Edition, *Prentice Hall of India Pvt. Ltd.*, 2008 and its "e-book" version.

**Registers, Shift Registers, Bi-directional shift registers,
Counters, Ripple and Synchronous Counters, Ring and
Johnson counters.**

Registers



- Clocked sequential circuit
 - No flip-flops / no feedbacks → reduce to combinational circuit
 - No combinational circuit → remain a sequential circuit
 - registers and counters
- Register: a group of flip-flops capable of storing one bit of information
 - n-bit register consists of a group of n flip-flops capable of storing n bits
- Counter: a register going through a predetermined sequence of states



Registers and Counters

Register:

- A set of flip-flops, possibly with added combinational gates, that perform data-processing tasks.
- Store and manipulate information in a digital system.

Counter:

- A register that goes through a predetermined sequence of states
- A special type of register
- Employed in circuits to sequence and control operations

Shift Register

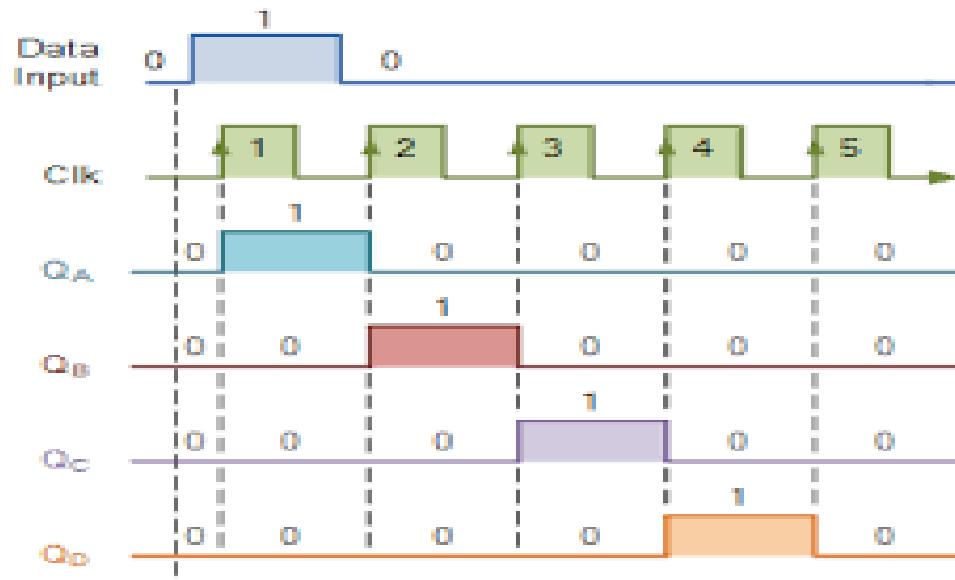
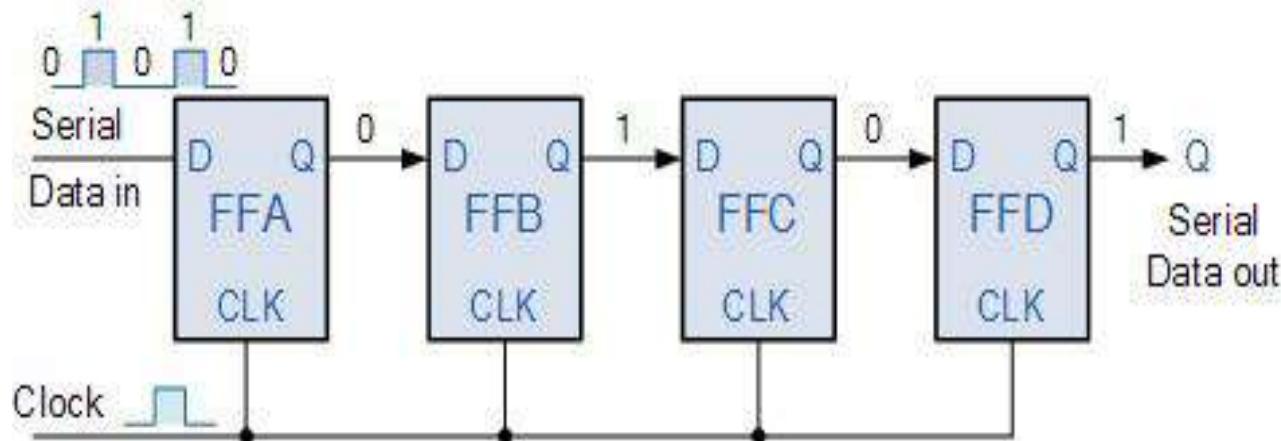
1. Serial In Serial Out (SISO)
2. Serial In Parallel Out (SIPO)
3. Parallel In Serial Out (PISO)
4. Parallel In Parallel Out (PIPO)

Note:

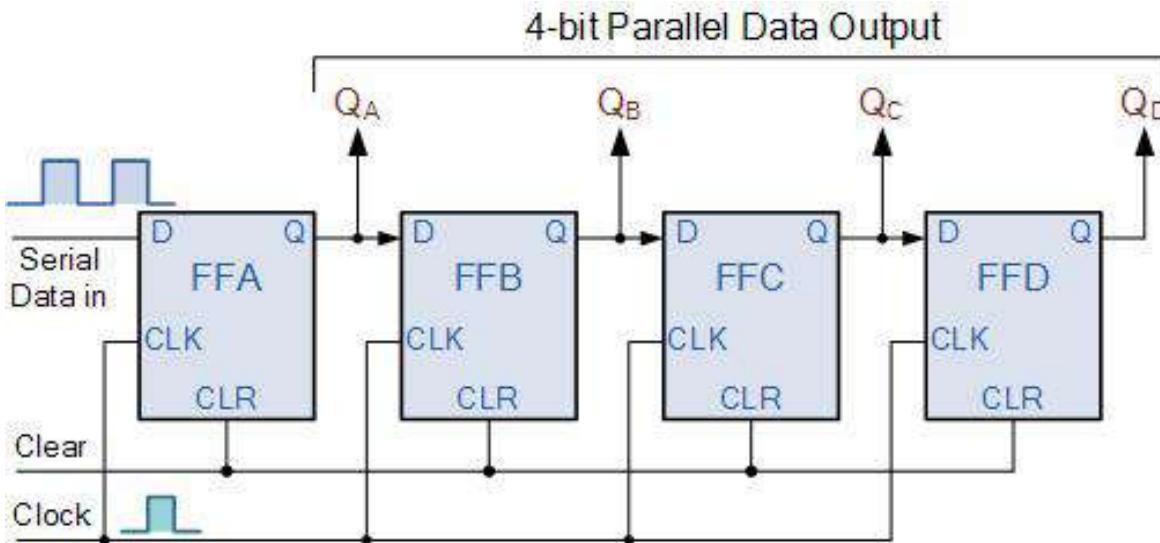
Serial → Single

Parallel → Many

Serial In Serial Out(SISO)

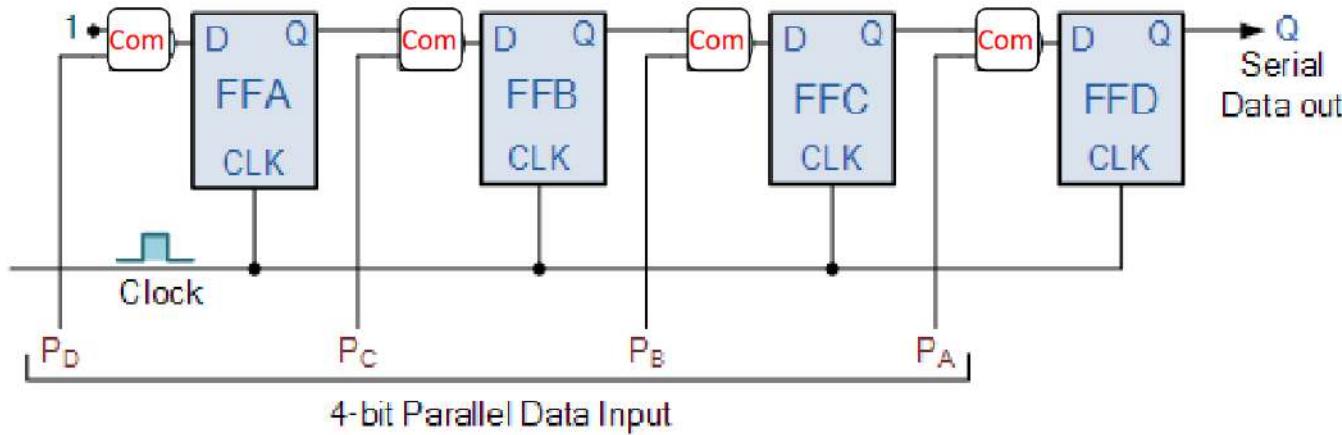


Serial In Parallel Out(SIPO)

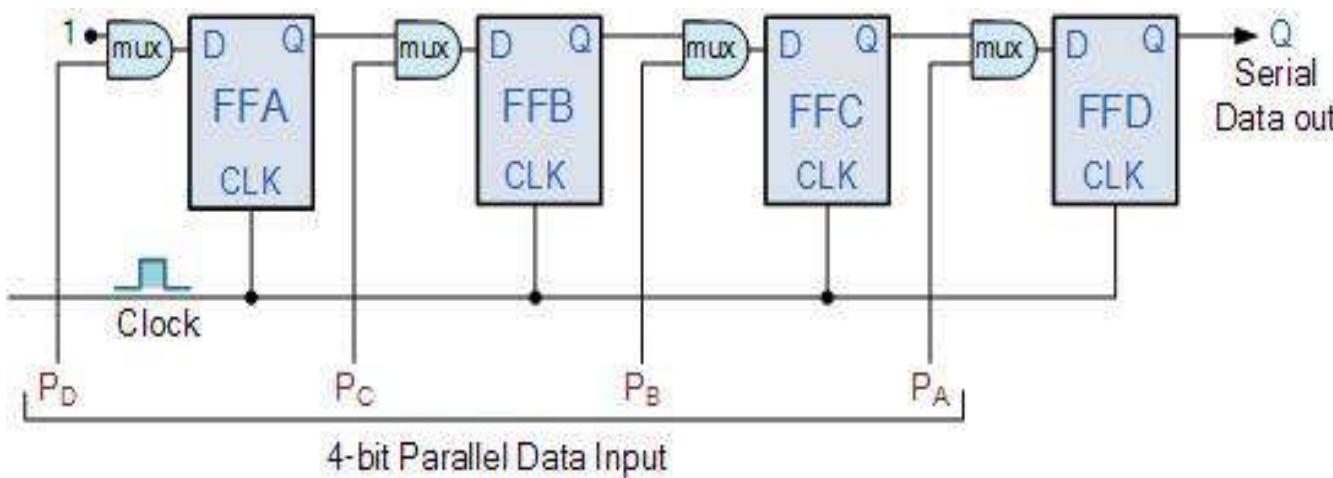


Clear	FF0	FF1	FF2	FF3
1001	0	0	0	0
	1	0	0	0
	0	1	0	0
	0	0	1	0
	1	0	0	1

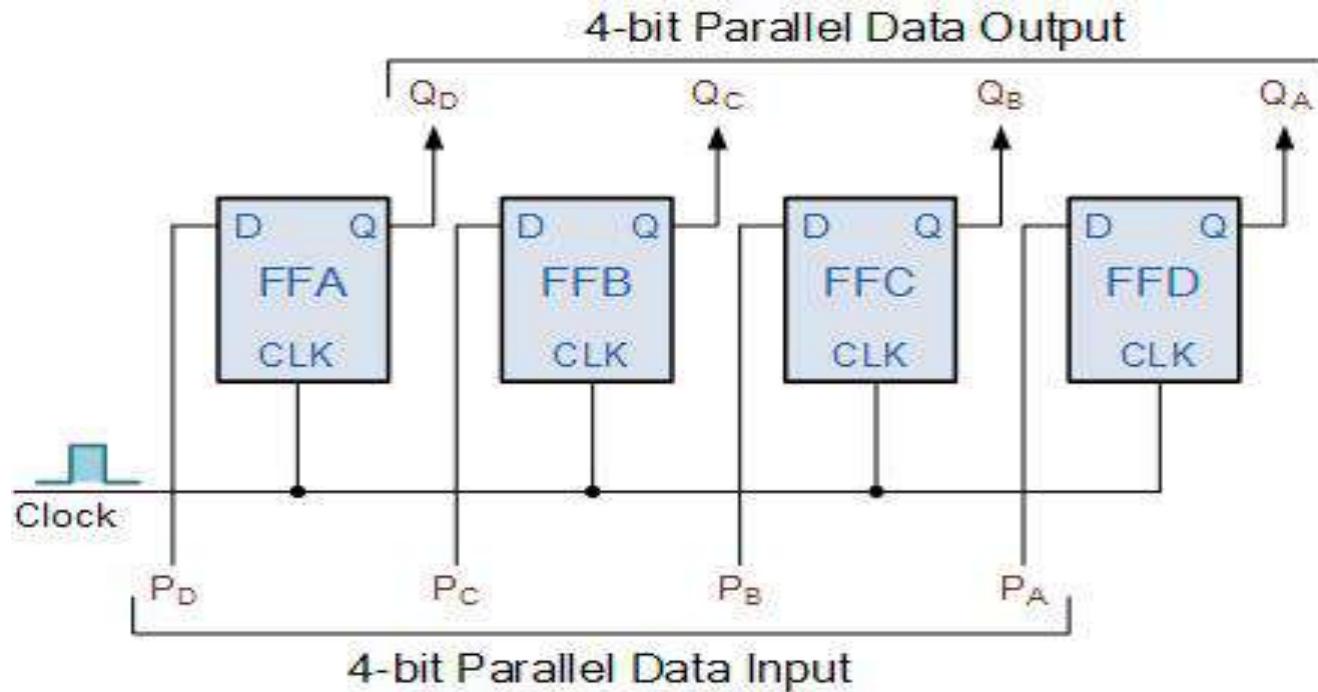
Parallel In Serial Out (PISO)



Com → Combinational Circuit (Any)

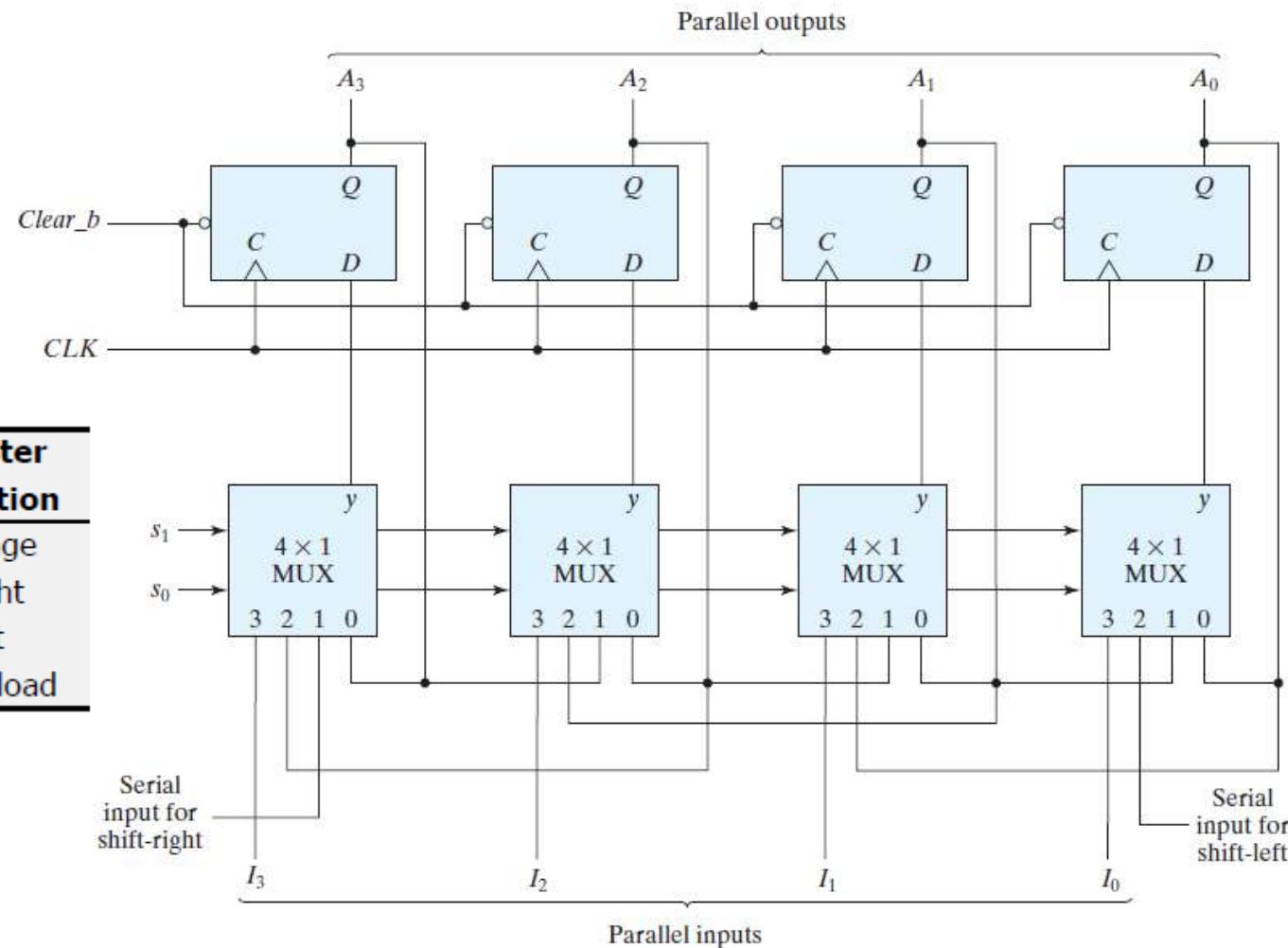


Parallel In Parallel Out (PIPO)



This is ordinary register and it won't perform any shift operation.

Universal Shift Register





Application of Shift Registers

- https://www.tutorialspoint.com/digital_circuits/digital_circuits_application_of_shift_registers.htm

EXAMPLE:The group of bits 11001 is serially shifted (right-most bit first) into a 5-bit parallel output shift register with an initial state 01110. After three clock pulses, the register contains _____

ANS: LSB bit is inverted and feed back to MSB:

01110->initial

10111->first clock pulse

01011->second

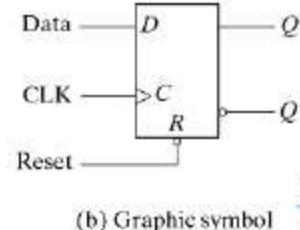
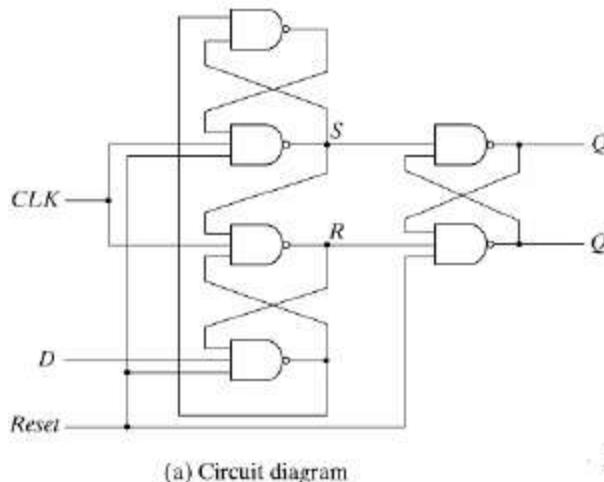
00101->third.

Q.The bit sequence 0010 is serially entered (right-most bit first) into a 4-bit parallel out shift register that is initially clear. What are the Q outputs after two clock pulses?

answer is 1000.

4-bit Register

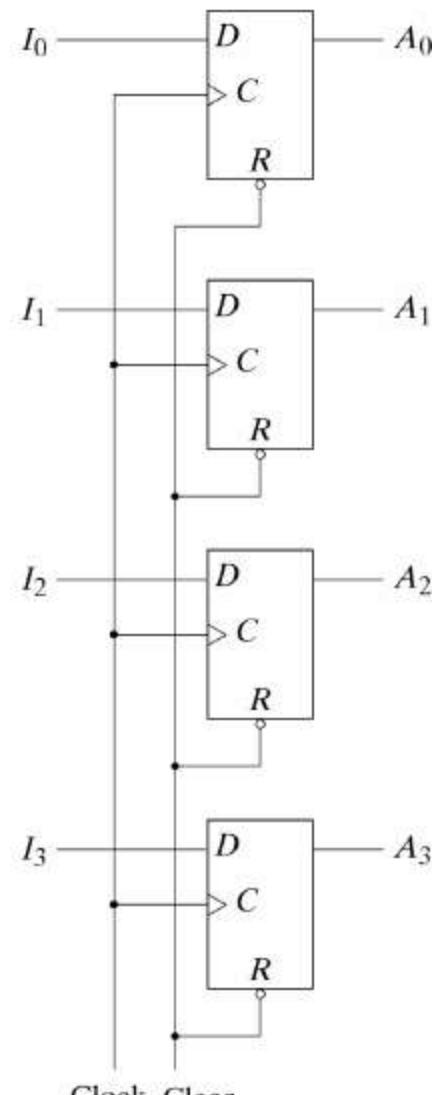
- Simplest register: consisting of only flip-flops without any gates
- Example 6-1: 4-bit register
 - positive edge trigger
 - When the clear input goes to 0, all flip-flops are reset
 - The R inputs must be maintained at logic 1 during normal clocked operation



R	C	D	Q	Q'
0	X	X	0	1
1	↑	0	0	1
1	↑	1	1	0

(c) Function table

D Flip-Flop with Asynchronous Reset



4-Bit Register

Register with Parallel Load

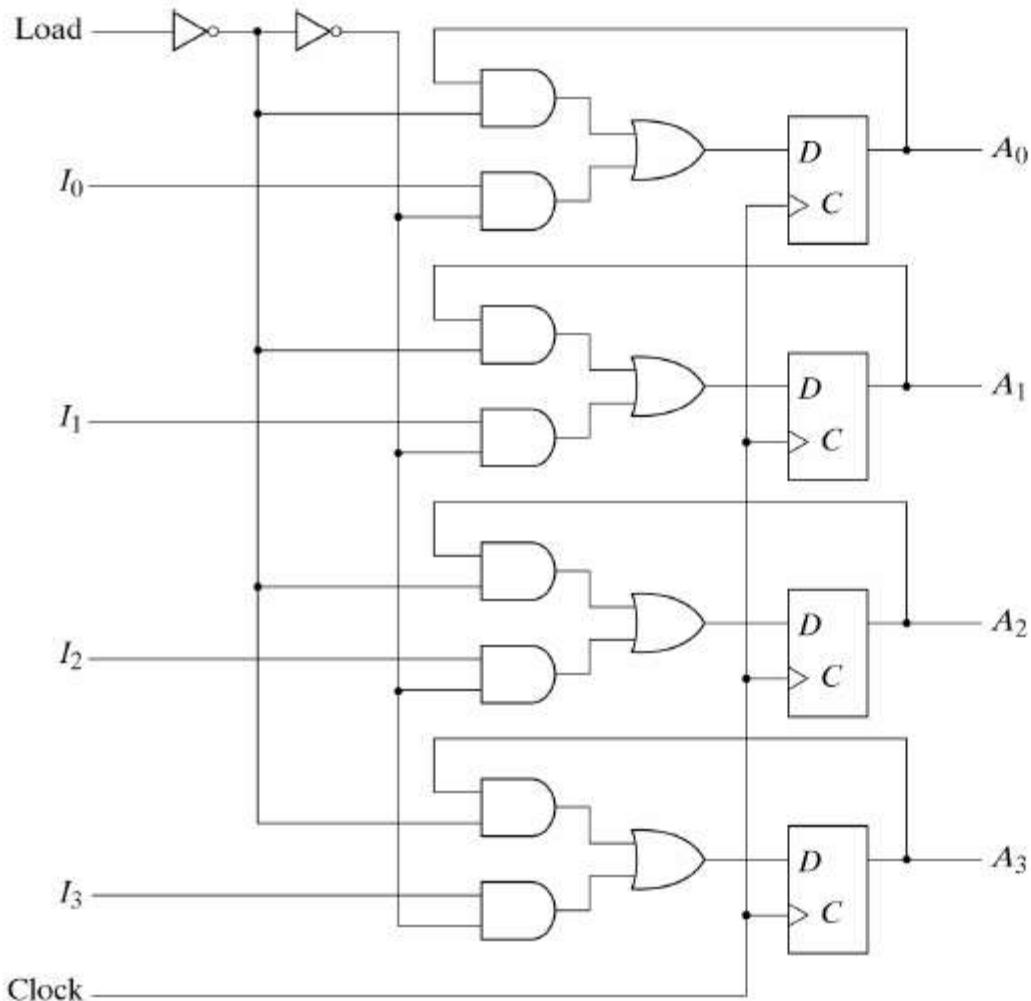
Parallel load

- loading: the transfer of new information into a register
- parallel loading: all the bits of the register are loaded simultaneously with a common clock pulse
 - **Load control**: determine when to load new information

Approaches to register with parallel load

1. controlling the clock input signal with an enabling gate:
uneven propagation delays between the master clock and the inputs of flip-flops
2. controlling the D inputs: ensure that all clock pulses arrive at the same time anywhere in the system

4-bit register with a load control input



load=1

- data are transferred into the register with the next positive edge of the lock

load=0

- outputs are connected to their respective inputs

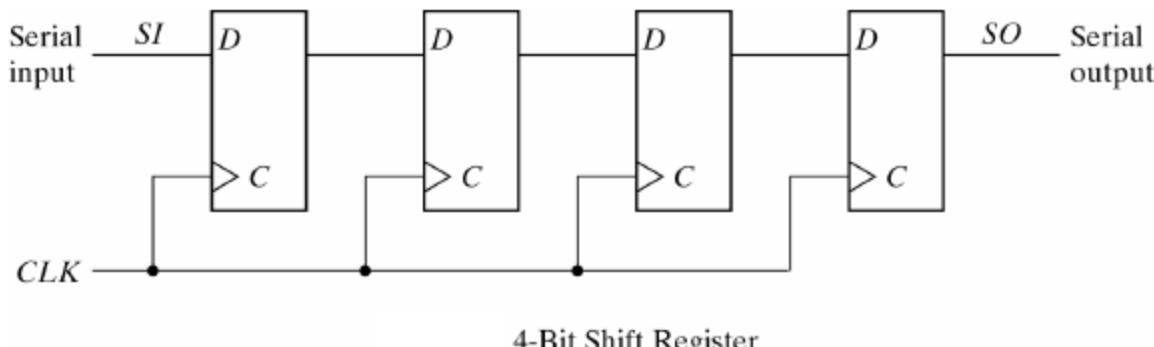
• The feedback connection is necessary because the D flip-flop does not have a “no change” condition

• The clock pulses are applied to the C inputs at all times

4-Bit Register with Parallel Load

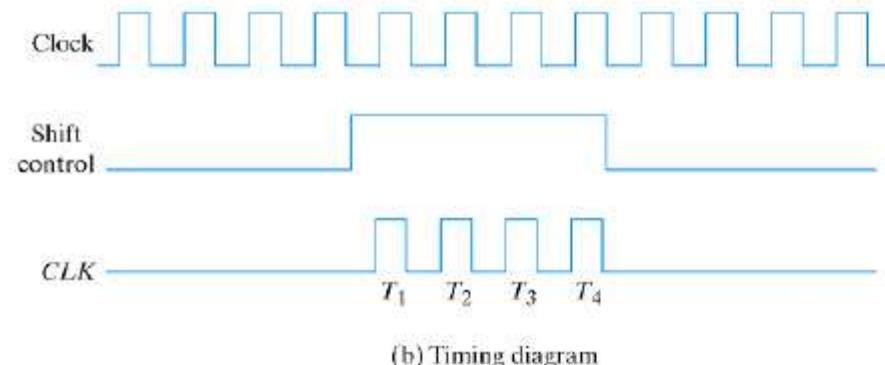
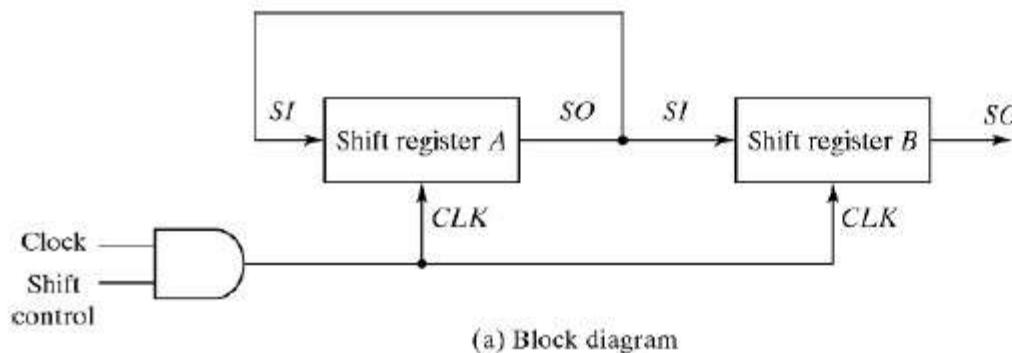
Shift Registers

- shift register: a register capable of shifting its binary information in one or both directions
- Example Fig. 6-3: each clock pulse shifts the contents of the register one bit position to the right
 - serial input: determines what goes into the leftmost flip-flop
 - serial output: taken from the output of the rightmost flip-flop
- **Shift control:** make the shift occur only with certain pulses
 - inhibiting the clock
 - control through the D inputs (shown later)



Serial Transfer

serial transfer: information is transferred one bit at a time by shifting the bits out of source register into destination register



Serial Transfer from Register A to register B

- The serial output (SO) of register A is connected to the serial input (SI) of register B and the SI of register A itself
- The shift control input determines when and how many times the registers are shifted
- serial vs. parallel

Table 6-1
Serial-Transfer Example

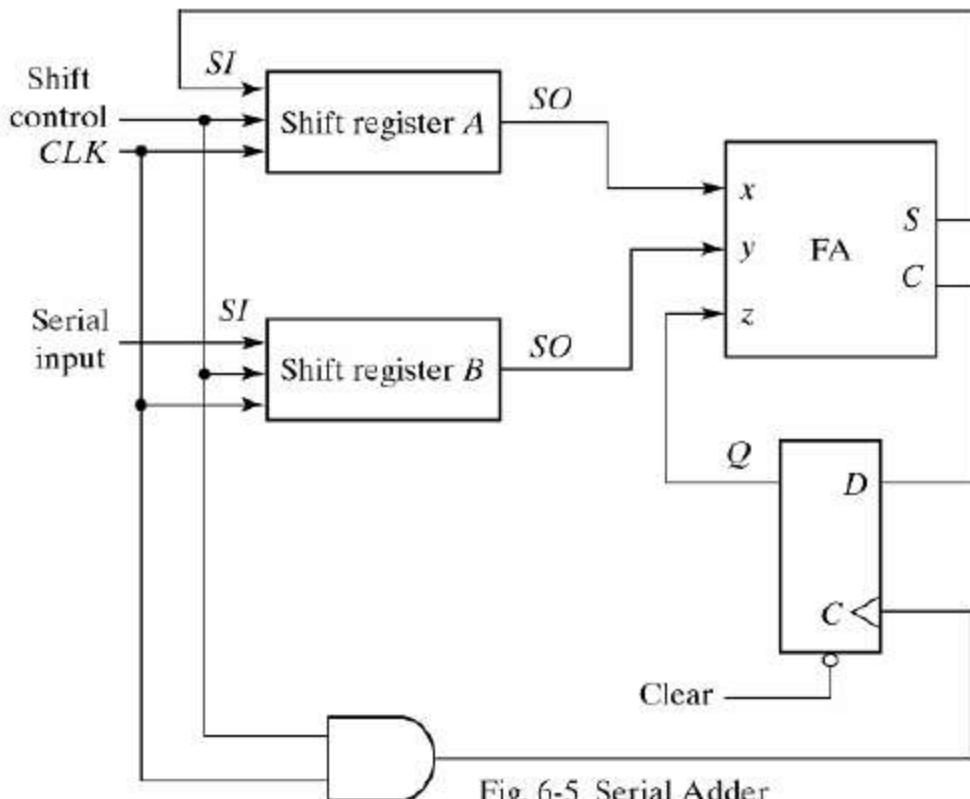
Timing Pulse	Register A	Register B
Initial value	1 0 1 1	0 0 1 0
After T_1	1 1 0 1	1 0 0 1
After T_2	1 1 1 0	1 1 0 0
After T_3	0 1 1 1	0 1 1 0
After T_4	1 0 1 1	1 0 1 1

Serial Addition

Register A holds the augend and register B holds the addend

- Initially, register A and carry flip-flop are cleared to 0

All the numbers are transferred serially into B and added to A



- parallel adder: use registers with parallel load
 - # of full adders = # of bits
 - faster
 - combinational circuit
- serial adder: use shift registers
 - requiring less equipment
 - only one full adder
 - sequential circuit

Second Form of Serial Adder

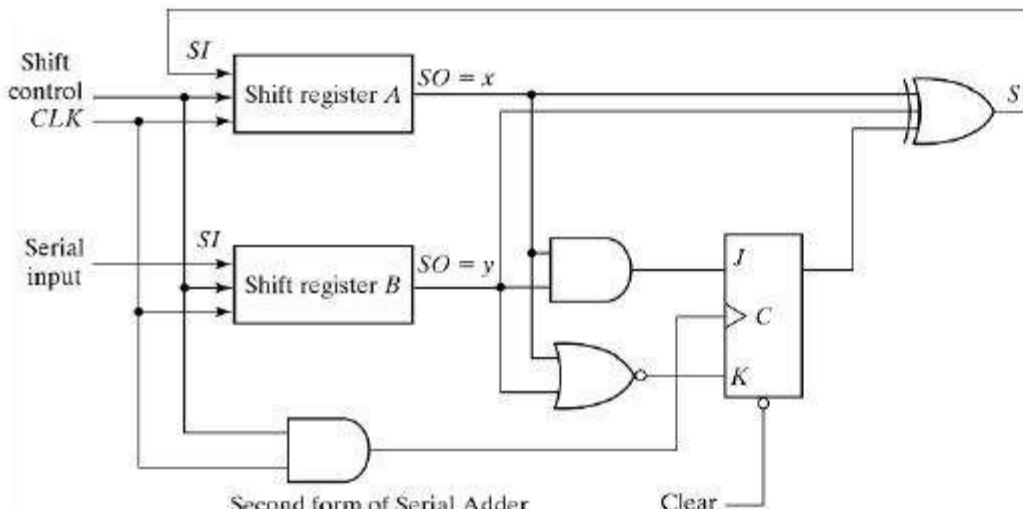
Table 6-2 State Table for Serial Adder

Present State	Inputs		Next State	Output	Flip-Flop Inputs	
Q	X	y	Q	S	J_Q	K_Q
0	0	0	0	0	0	X
0	0	1	0	1	0	X
0	1	0	0	1	0	X
0	1	1	1	0	1	X
1	0	0	0	1	X	1
1	0	1	1	0	X	0
1	1	0	1	0	X	0
1	1	1	1	1	X	0

Design a serial adder using a JK FF

- Assume 2 shift registers as input
- Obtain state table with FF input/outputs
- Obtain input and output equations
- Draw the circuit

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



Universal Shift Register

- Unidirectional shift register: capable of shifting in one direction only
- Bidirectional shift register: capable of shifting in both directions
- universal shift register: has both shifts and parallel load capabilities
- The most general shift register has the following capabilities:
 1. A *clear* control to clear the register to 0.
 2. A *clock* input to synchronize the operations.
 3. A *shift-right* control to enable the shift right operation and the *serial input* and *output* lines associated with the shift right.
 4. A *shift-left* control to enable the shift left operation and the *serial input* and *output* lines associated with the shift left.
 5. A *parallel-load* control to enable a parallel transfer and the n input lines associated with the parallel transfer.
 6. n parallel output lines.
 7. A control state that leaves the information in the register unchanged in the presence of the clock.

4-bit Universal Shift Register

Has all the capabilities listed above

Selection inputs control the mode of operation

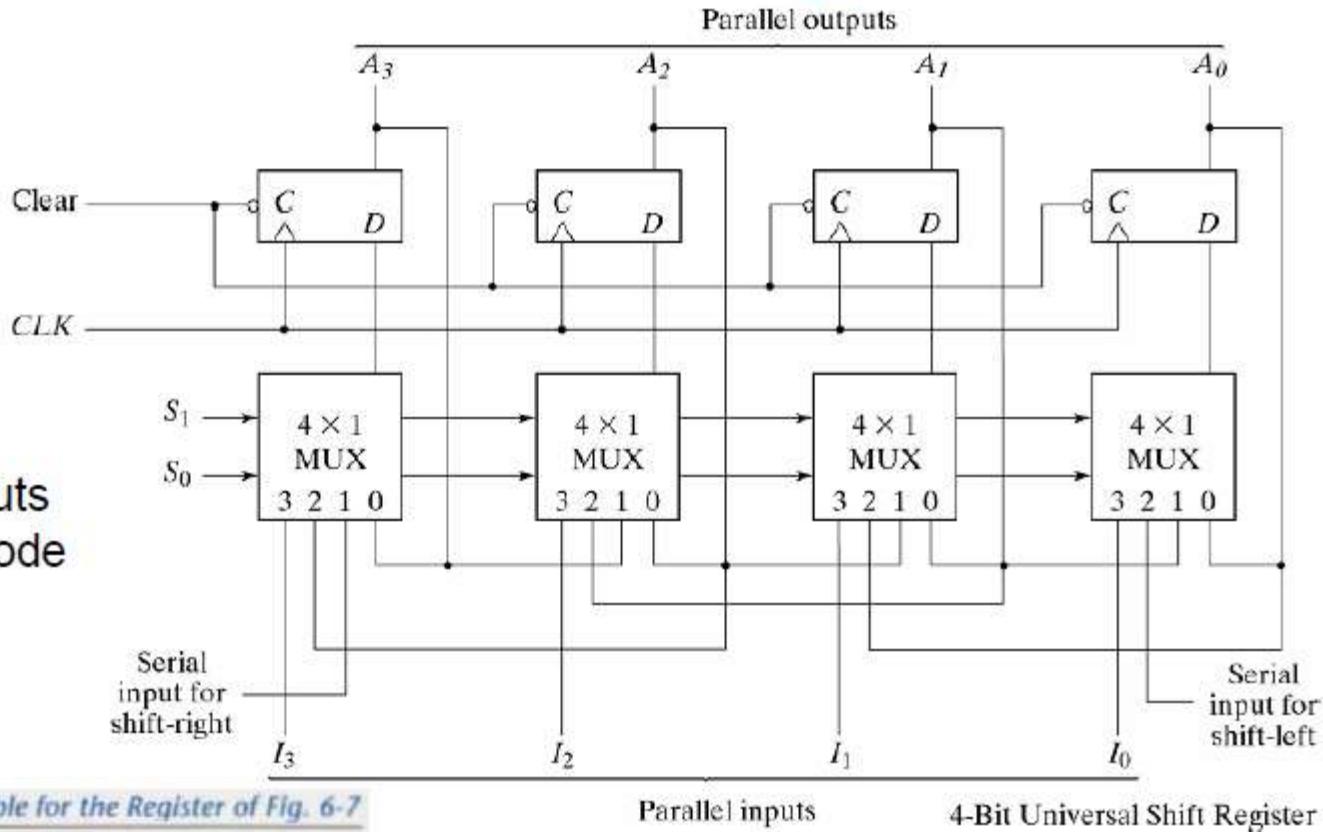


Table 6-3 Function Table for the Register of Fig. 6-7

Mode Control		Register Operation	
S_1	S_0		
0	0	No change	A _i
0	1	Shift right	A _{i+1}
1	0	Shift left	A _{i-1}
1	1	Parallel load	I _i

Shift registers are often used to interface digital systems situated remotely from each other

Ripple Counters

- **counter**: a register that goes through a prescribed sequence of states upon the application of input pulses
 - may occur at a fixed interval of time or at random
 - may follow the binary number sequence or any other sequence of states
 - n-bit **binary counter**: n flip-flops counting in binary from 0~ 2^n-1
- Two categories
 - ***Ripple counters***: FF output transition serves as a source for triggering other via the clock pin
 - Binary ripple counter
 - BCD ripple counter
 - ***Synchronous counters***: inputs of all FF receive the common clock
 - discussed in Sections 6-4 and 6-5



Sequential Circuits



Counters

- A register that goes through a prescribed sequence of states

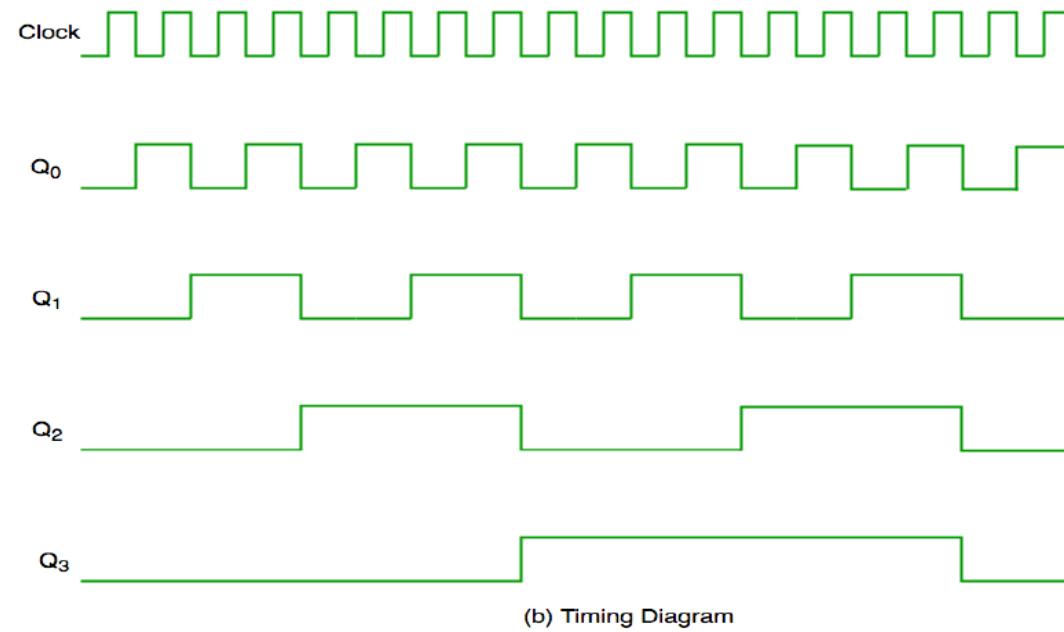
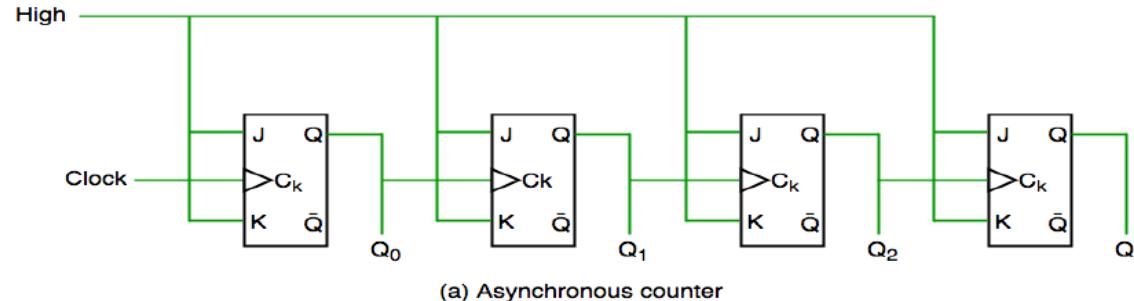
Ripple counter

- The flip-flop output transition serves as a source for triggering other flip-flops
- Hardware is much simpler
- Unreliable and delay dependent due to the asynchronous behaviors

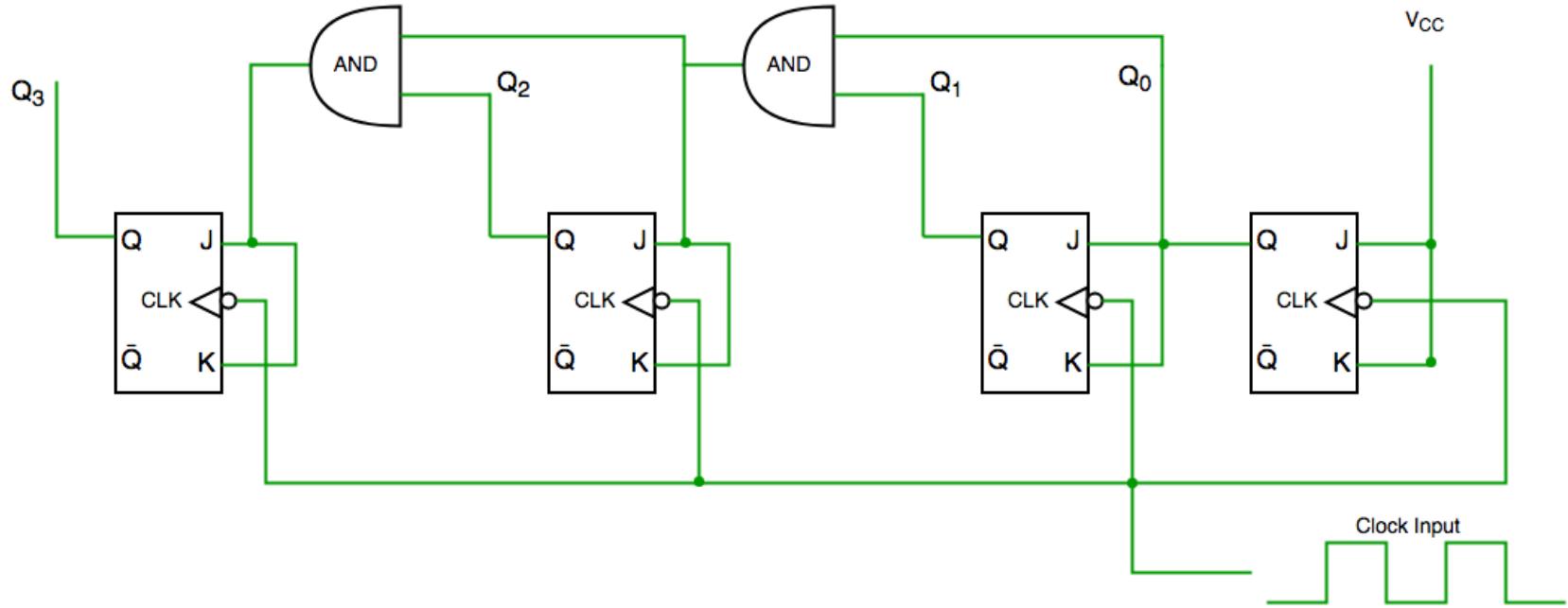
Synchronous counter

- The common clock pulse triggers all flip-flops simultaneously
- Hardware may be more complex but more reliable

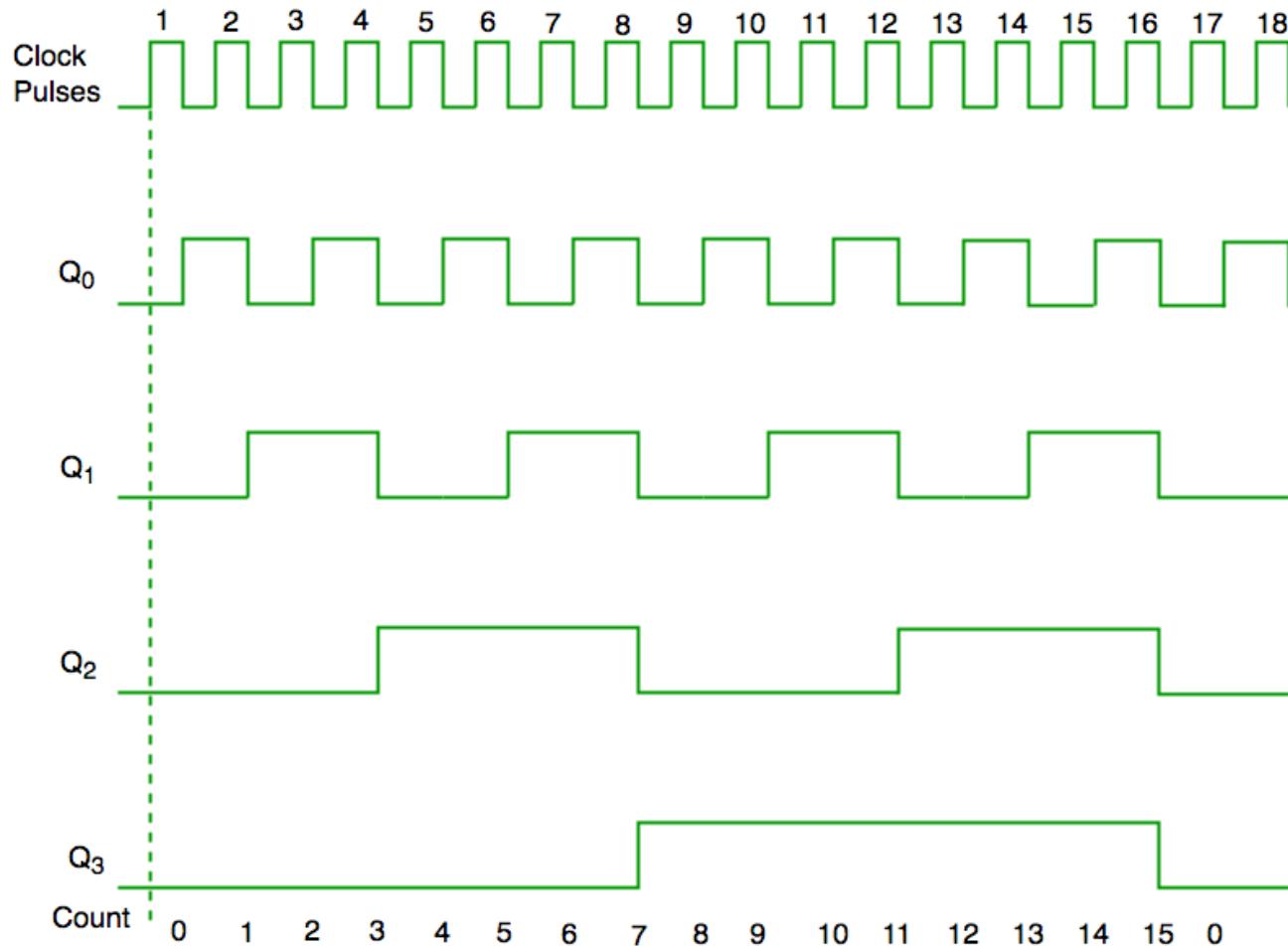
Asynchronous Counter /Serial Counter



Synchronous Counter/parallel Counter



Timing diagram synchronous counter





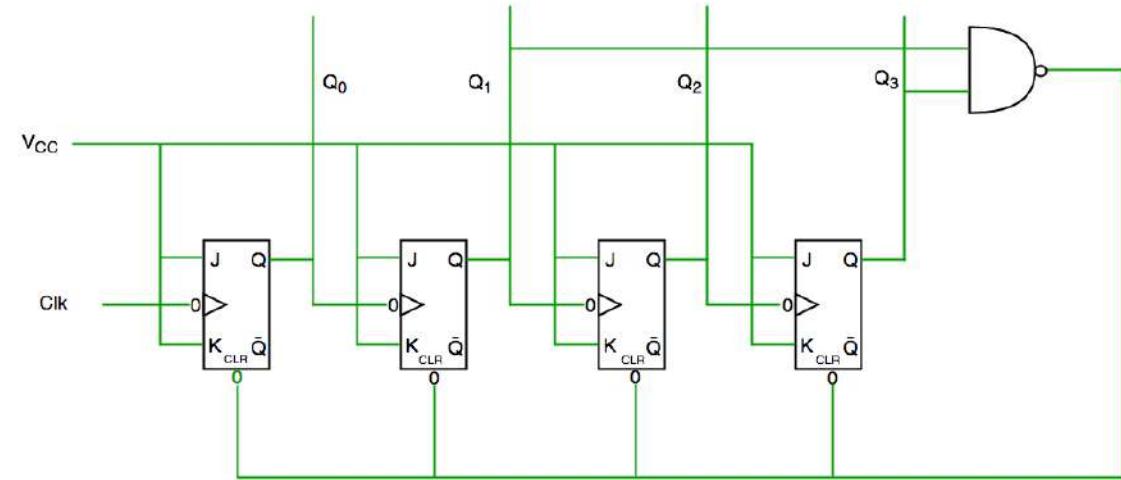
Decade Counter

- A decade counter counts ten different states and then reset to its initial states. A simple decade counter will count from 0 to 9 but we can also make the decade counters which can go through any ten states between 0 to 15(for 4 bit counter).

Decade Counter

Truth table for simple decade counter

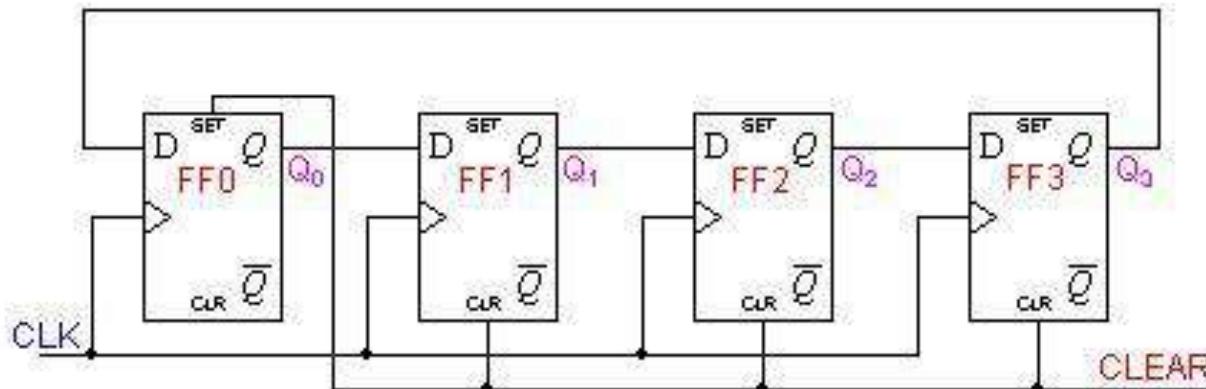
Clock pulse	Q3	Q2	Q1	Q0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	0	0	0	0



Decade counter circuit diagram

Important point: Number of flip flops used in counter are always greater than equal to $(\log_2 n)$ where n =number of states in counter.

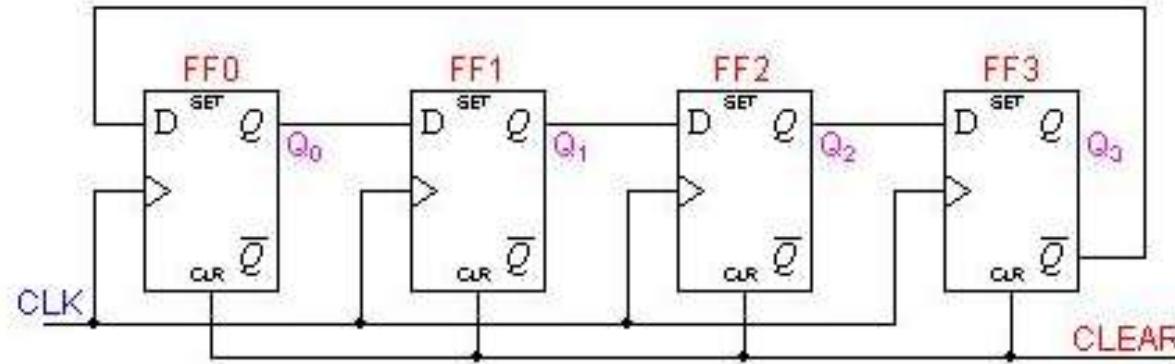
Ring Counter (Shift register based)



Clock Pulse	Q3	Q2	Q1	Q0
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	1	0	0	0

- Also known as mod-4 counter

Twisted Ring Counter (Jonshon Counter)

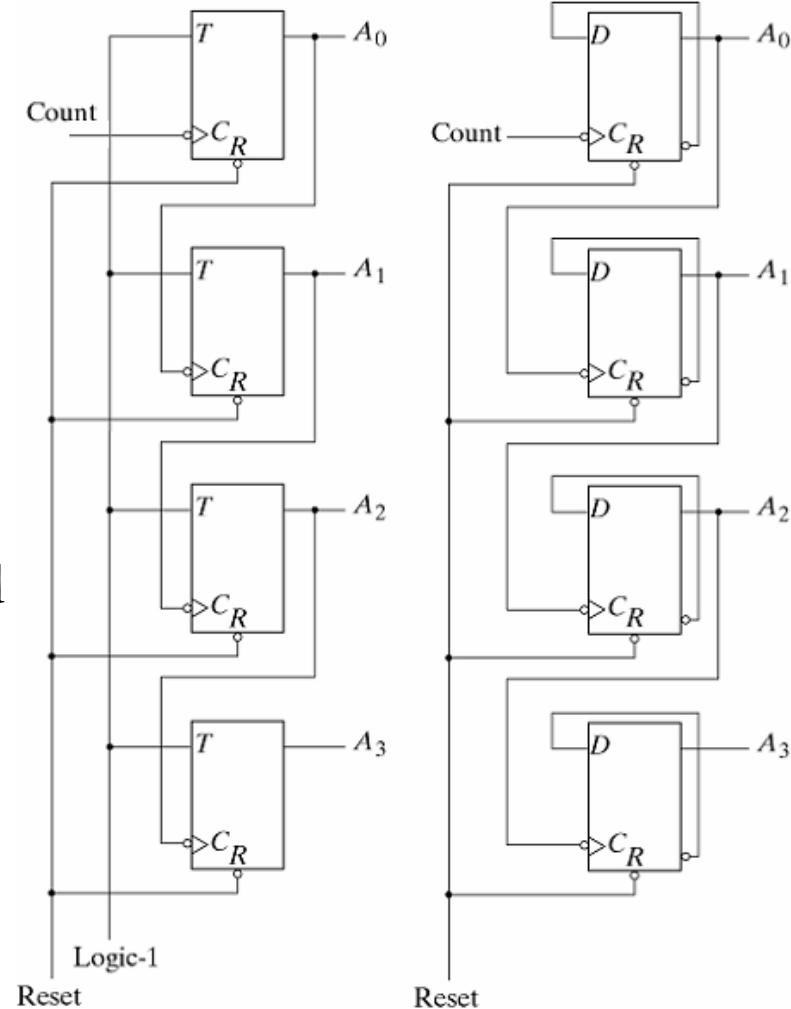


Clock Pulse	Q3	Q2	Q1	Q0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	1
3	0	1	1	1
4	1	1	1	1
5	1	1	1	0
6	1	1	0	0
7	1	0	0	0

- Also known as mod- 2^n counter

4-Bit Binary Ripple Counter (Asynchronous)

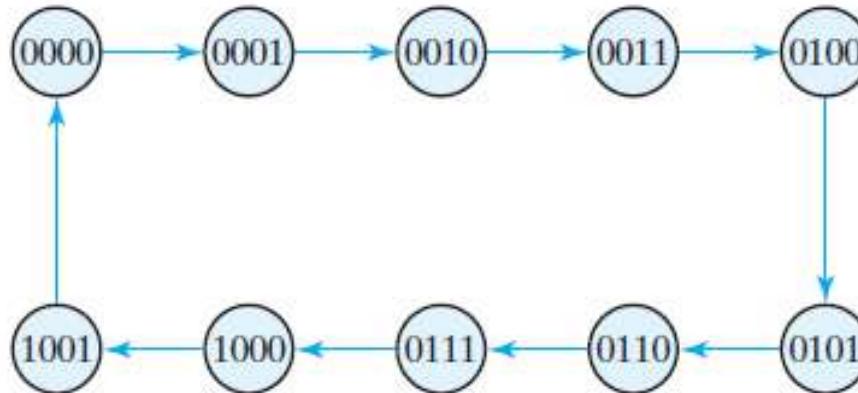
- One single count input
- Output of each FF connected to C input of next higher-order FF
- **Three approaches**
 - from T
 - from JK: J and K inputs tied together
 - from D: complement output connected to the D input
- Every time A_i goes from 1 to 0, it complements A_{i+1} (Negative trigger)
- Binary count-down counter use positive-trigger T flip-flops instead



(a) With T flip-flops

(b) With D flip-flops

BCD or Decade ripple Counter using JK

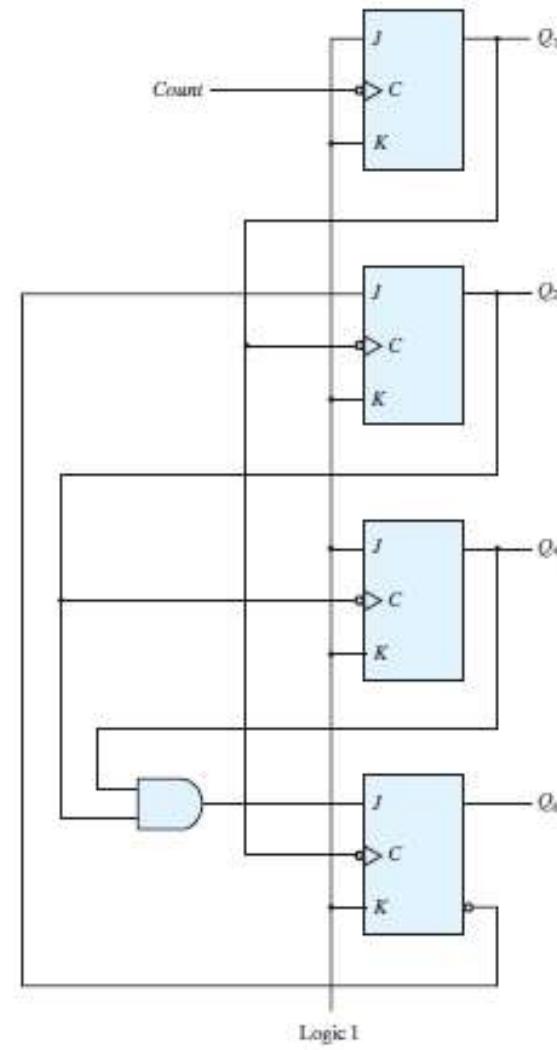
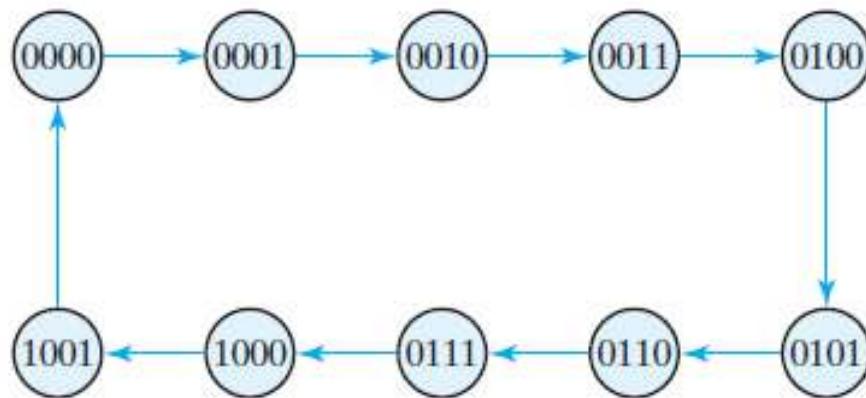


$$\begin{aligned}
 T_{Q1} &= 1 \\
 T_{Q2} &= Q_8'Q_1 \\
 T_{Q4} &= Q_2Q_1 \\
 T_{Q8} &= Q_8Q_1 + Q_4Q_2Q_1 \\
 y &= Q_8Q_1
 \end{aligned}$$

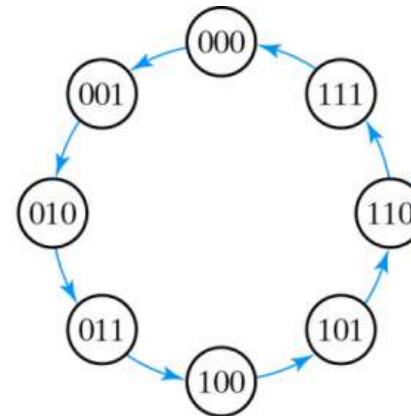
State Table for BCD Counter

Present State				Next State				Output y	Flip-Flop Inputs			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1		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

BCD or Decade ripple Counter using JK

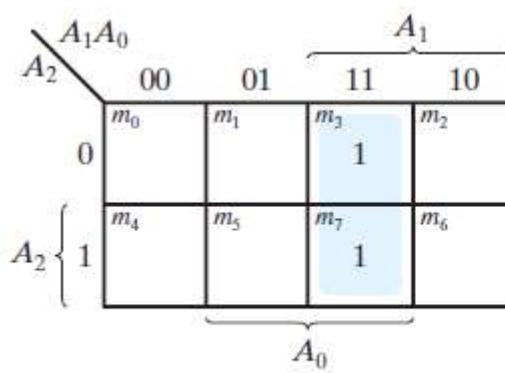
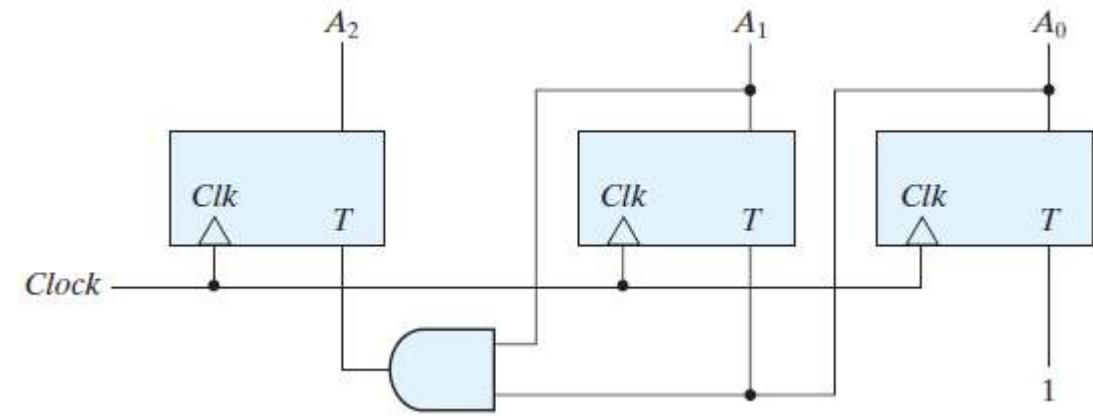
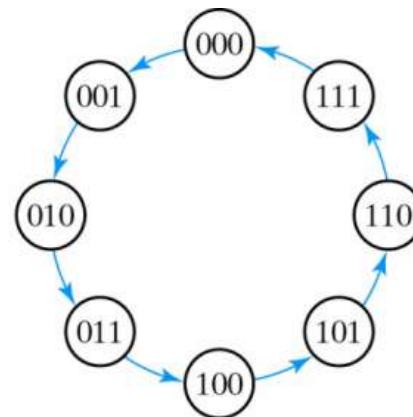


3-bit Binary Counter using T F/F

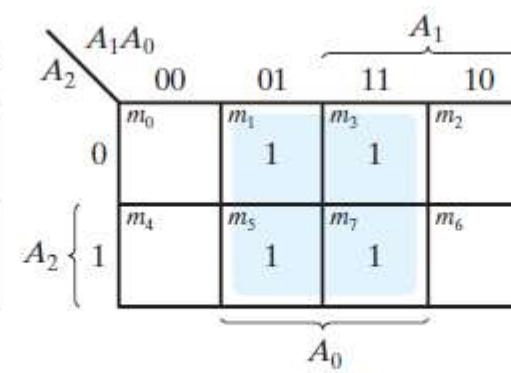


Output State Transitions			Flip-flop inputs					
Present State Q2 Q1 Q0			Next State Q2 Q1 Q0			T2 T1 T0		
0 0 0			0 0 1			0 0 1		
0 0 1			0 1 0			0 1 1		
0 1 0			0 1 1			0 0 1		
0 1 1			1 0 0			1 1 1		
1 0 0			1 0 1			0 0 1		
1 0 1			1 1 0			0 1 1		
1 1 0			1 1 1			0 0 1		
1 1 1			0 0 0			1 1 1		

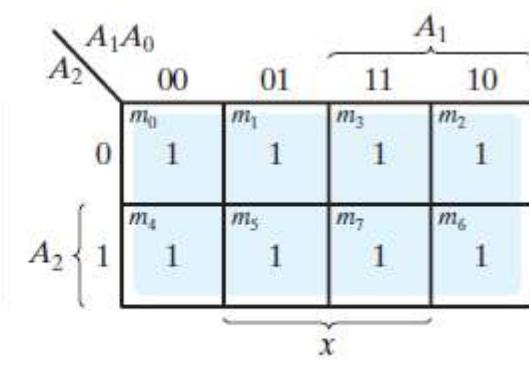
3-bit Binary Counter using T F/F



$$T_{A2} = A_1A_0$$

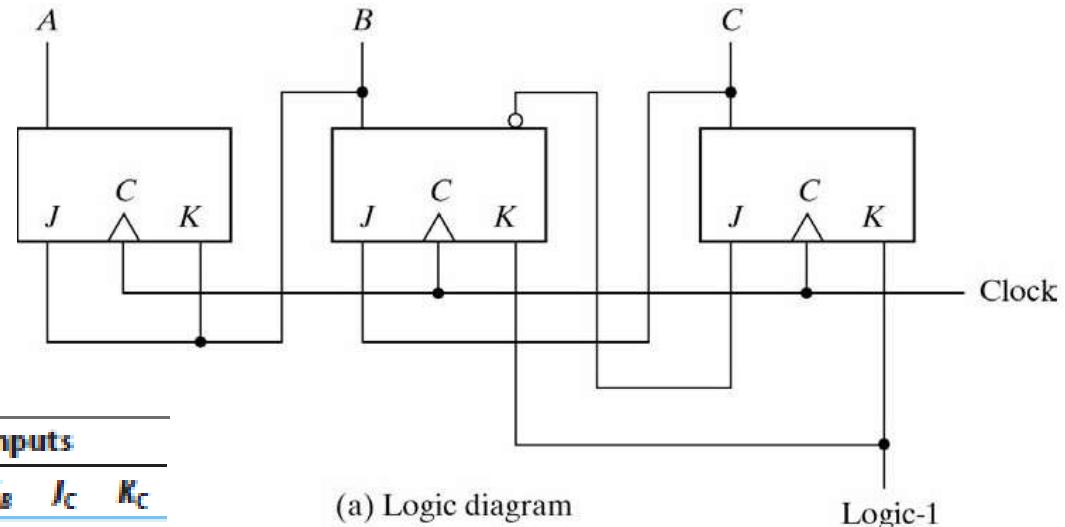


$$T_{A1} = A_0$$

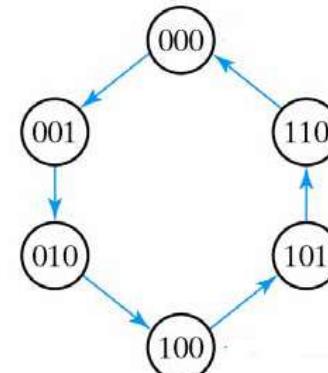


$$T_{A0} = 1$$

Counter with random States



Present State			Next State			Flip-Flop Inputs					
A	B	C	A	B	C	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	1	0	0	1	X	X	1	0	X
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	0	0	0	X	1	X	1	0	X



(b) State diagram

Simplified equations:

$$J_A = B \quad K_A = B$$

$$J_B = C \quad K_B = 1$$

$$J_C = B' \quad K_C = 1$$

4-Bit Synchronous UP Binary Counter

A_3	A_2	A_1	A_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

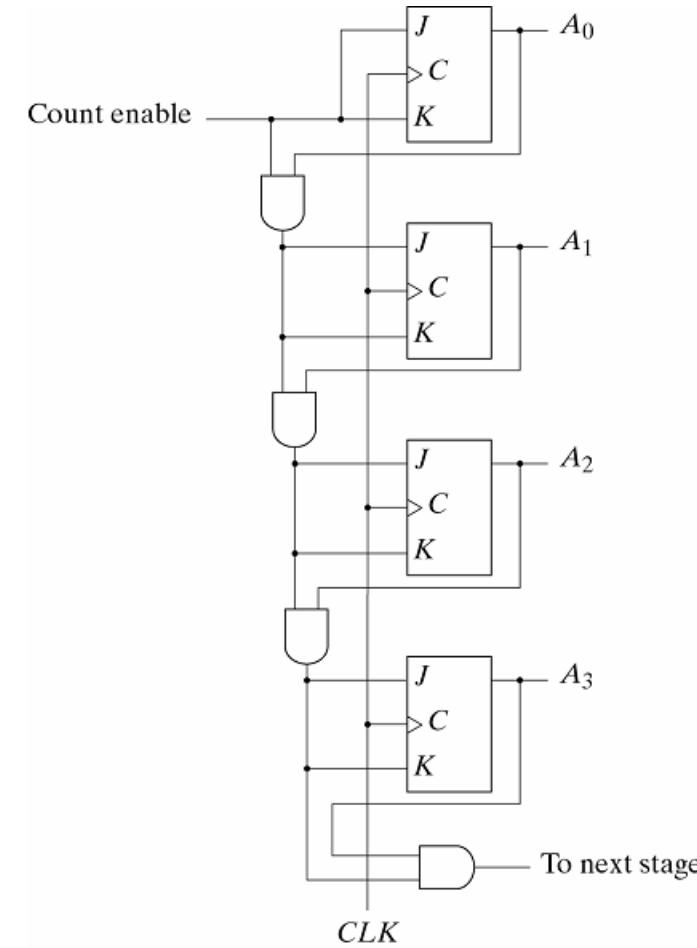
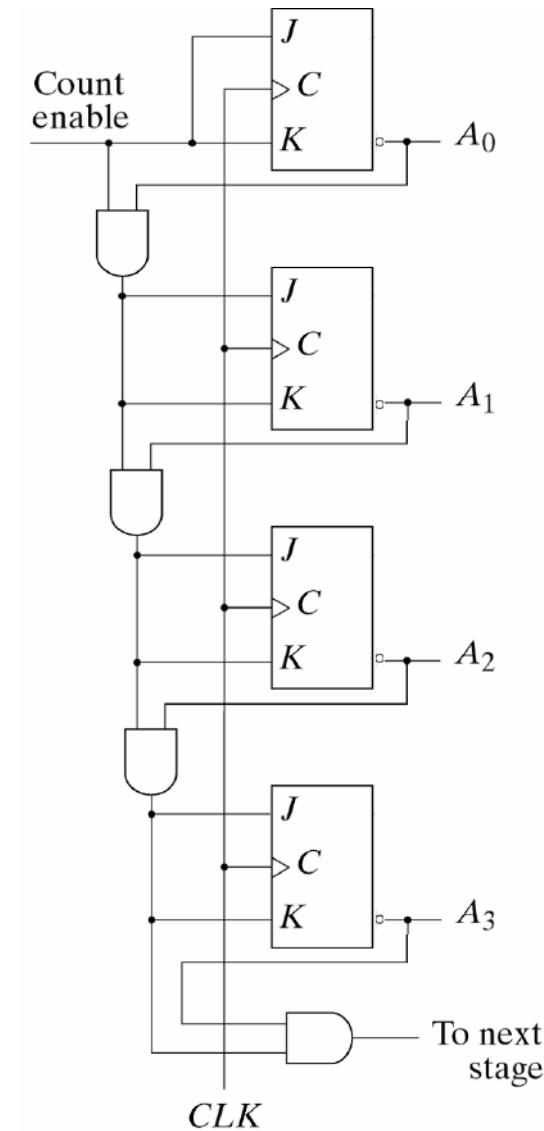


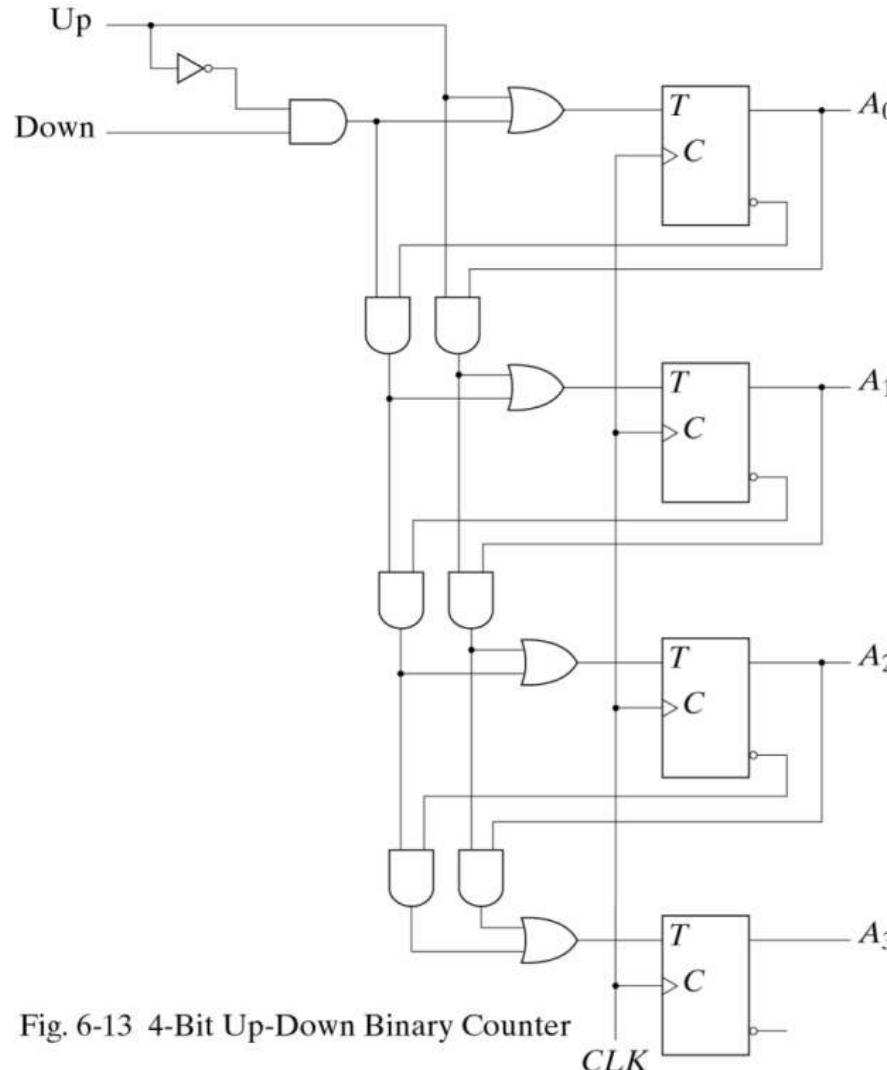
Fig. 6-12 4-Bit Synchronous Binary Counter

Synchronous Count Down Binary Counter

A_3	A_2	A_1	A_0
1	1	1	1
1	1	1	0
1	1	0	1
1	1	0	0
1	0	1	1
1	0	1	0
1	0	0	1
1	0	0	0
0	1	1	1
0	1	1	0
0	1	0	1
0	1	0	0
0	0	1	1
0	0	1	0
0	0	0	1
0	0	0	0



Up-Down Binary Counter



up	down	operation
1	x	count up
0	1	count down
0	0	no change

an up-down binary counter
using T flip-flops

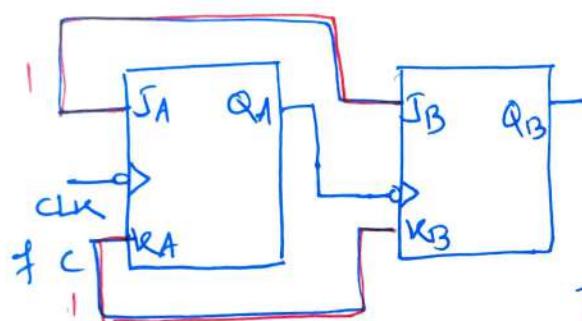
Fig. 6-13 4-Bit Up-Down Binary Counter

Counter → counts clock pulses.

counter:- counter is a sequential circuit and it is used to count like - 1, 2, 3, 4, 5...n it starts and ends counting. It is also known as frequency divider.
e.g. 2-6, 3-9

Working of counter

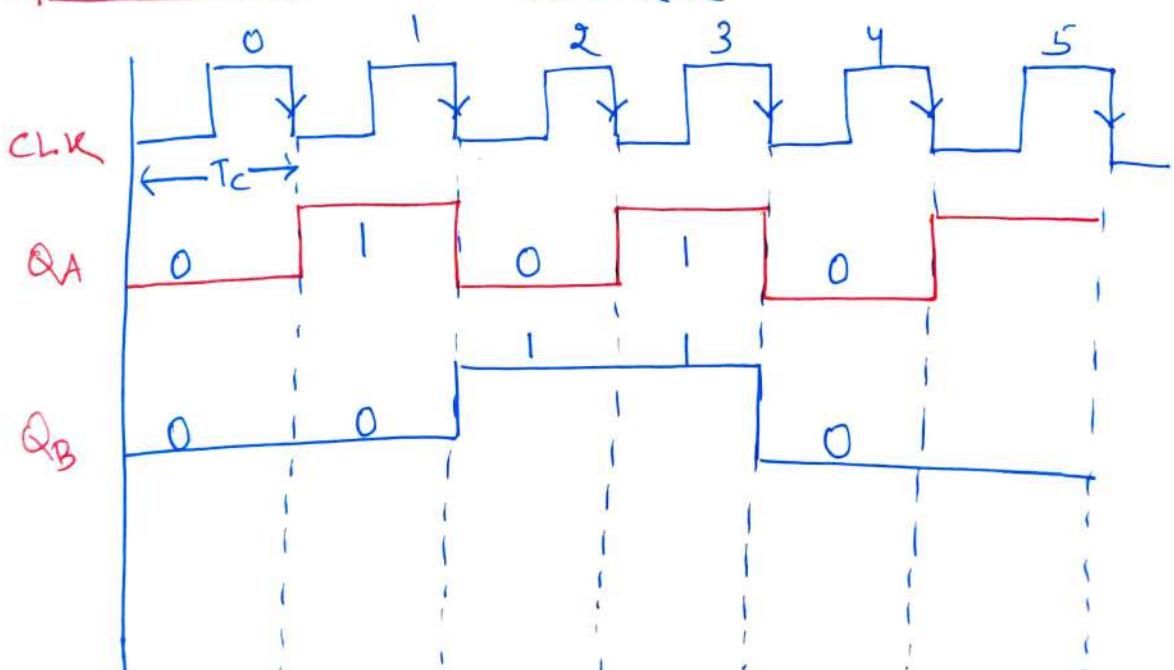
→ Toggling is used in counter
→ -ve edge triggered.



$$f_A = \frac{f_C}{2}, f_B = \frac{f_C}{4}$$

↑ P = no. of FF

$$T_A = 2T_C; \text{ if } P=2, Q=4 \therefore f_B = \frac{f_C}{4}$$



CLK	Q_B	Q_A
0	0	0
1	0	1
2	1	0
3	1	1
4	0	0

To count 0 to 15
we need $2^4 = 16$ FF

The seq → Q_D Q_C Q_B Q_A

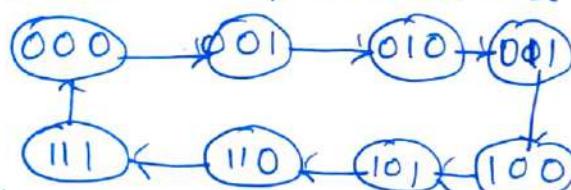
Design of 3-bit Synchronous UP Counter

Step-1: ① Find no. of FF ② Decide type of FF
for 3 bit \rightarrow no. of FF = 3 by using T FF.

Step-2: Excitation Table for T FF

Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

Step-3: To find the / draw the state diagram. $2^3 = 8$ = no. of states
maxⁿ no. of states = $2^n - 1 = 8 - 1 = 7$



state diagram

* By using state diagram find the correct excitation Table.

Present State			N.S			Input of FF		
Q_c	Q_B	Q_A	Q_c^+	Q_B^+	Q_A^+	T_c	T_B	T_A
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	0	0	0	0	1	1	1

K-map:

for T_c		
Q_c	Q_B	Q_A
00	00, 01	11, 10
10	00, 10	00, 11

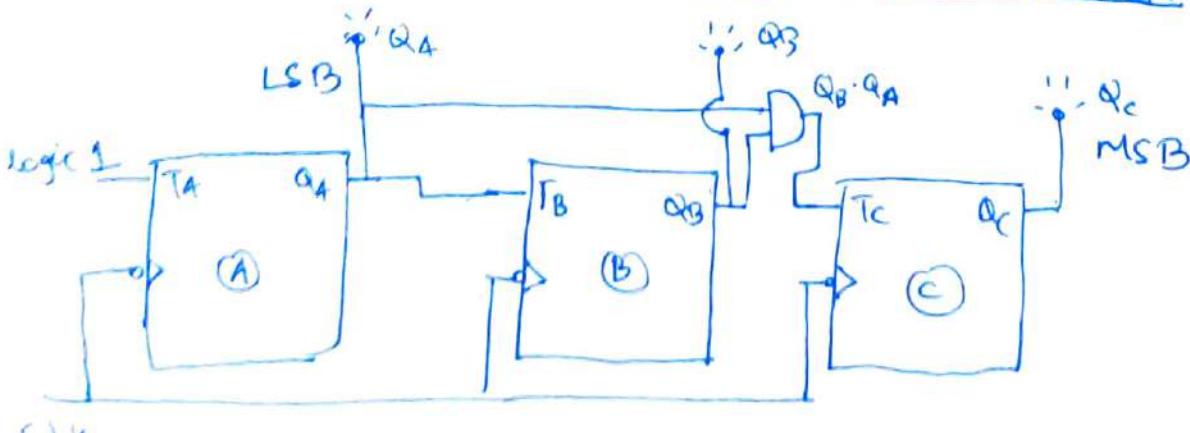
$$T_c = Q_B Q_A$$

for T_B		
Q_c	Q_B	Q_A
00	01	11, 10
10	11	00

$$T_B = Q_A$$

for T_A		
Q_c	Q_B	Q_A
00	01	11, 10
10	11	11

$$T_A = 1$$



BCE102L
DIGITAL SYSTEM DESIGN

Module:5

**DESIGN OF REGISTERS AND
COUNTERS**

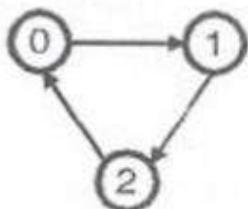
Courtesy: M. Morris Mano, "Digital Design", 3rd Edition, *Prentice Hall of India Pvt. Ltd.*, 2008 and its "e-book" version.

COUNTERS

Modulus counter

- The 2 bit counter is called as MOD-4 counter
- The 3 bit counter is called as MOD-8 counter
- MOD number = 2^n
- Modulus means no. of states through which the counter progress during its operations.

Mod -3 asynchronous counter using 2-bit ripple counter



(a) State diagram of a MOD-3 counter

Truth table of reset logic

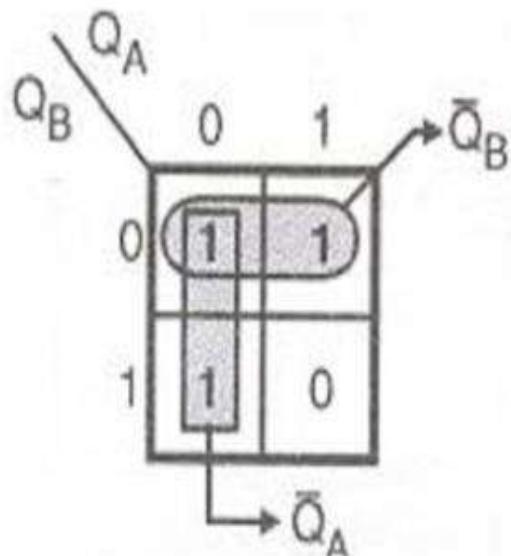
Flip-flop outputs		Output of reset logic Y
Q_B	Q_A	
0	0	1
0	1	1
1	0	1
1	1	0

} Valid states
→ Clear all the flip-flops

Mod -3 asynchronous counter using 2-bit ripple counter

K-map and simplification :

The K-map and simplifier expression for the output (Y) of the reset logic



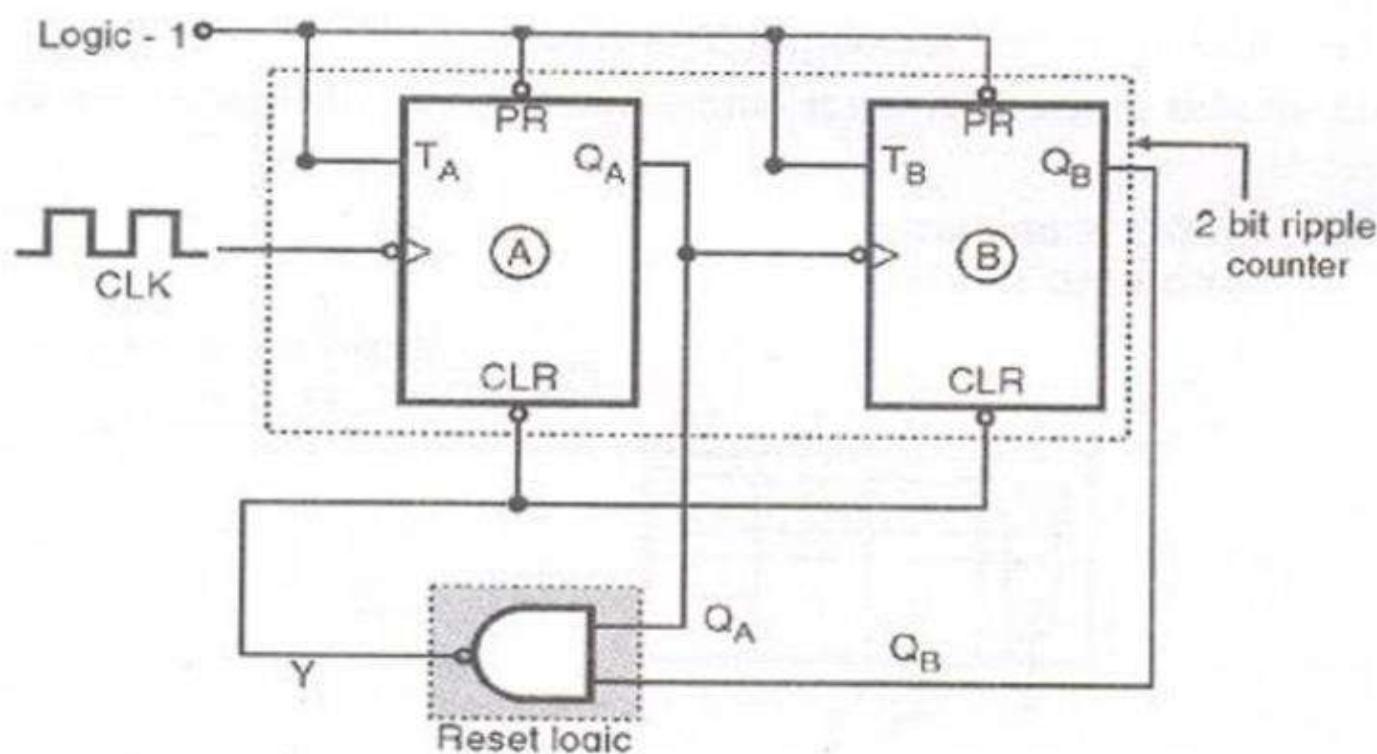
Boolean expression for Y

$$Y = \bar{Q}_B + \bar{Q}_A$$

$$\therefore Y = \overline{\bar{Q}_B \cdot \bar{Q}_A}$$

Mod -3 asynchronous counter using 2-bit ripple counter

The logic diagram of MOD-3 counter

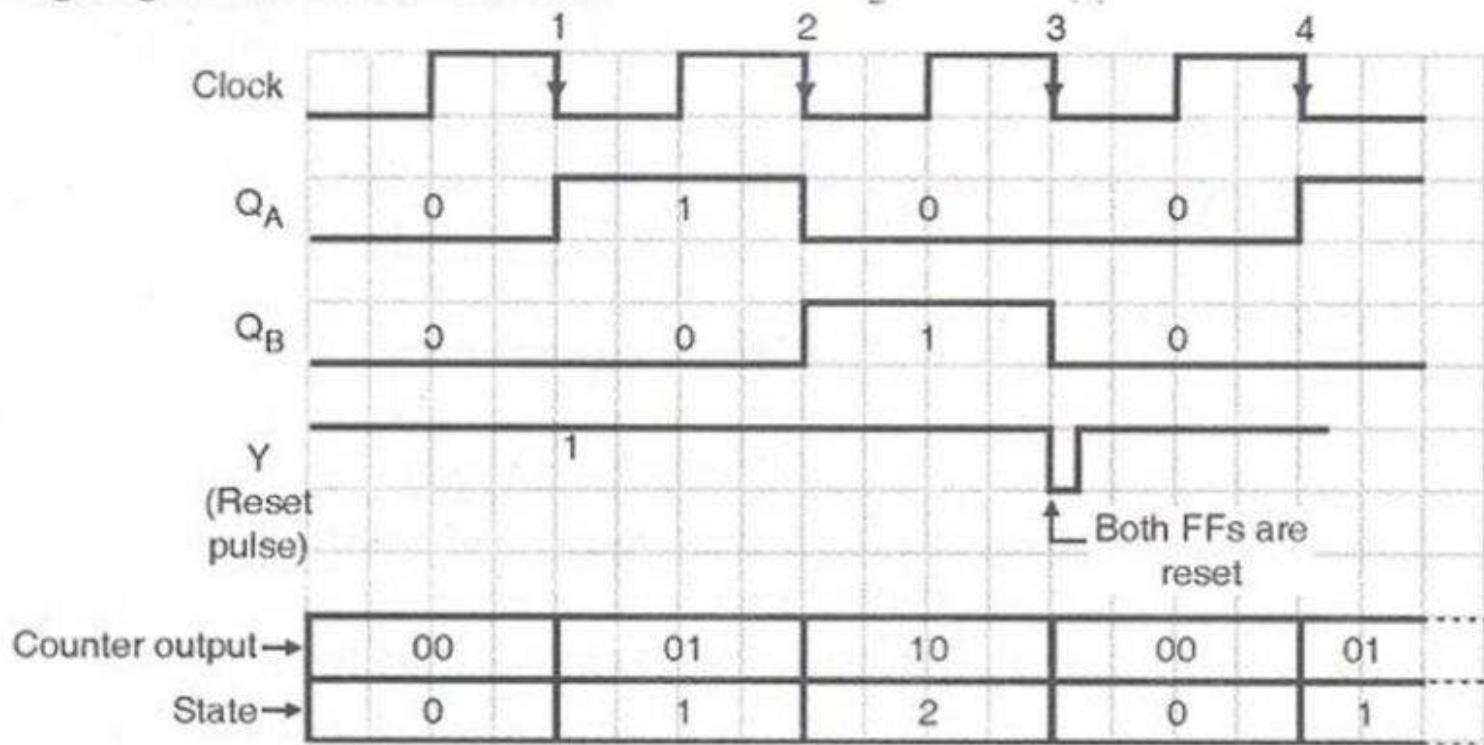


MOD-3 counter using a 2-bit ripple counter

Mod -3 asynchronous counter using 2-bit ripple counter

Timing diagram of MOD-3 counter :

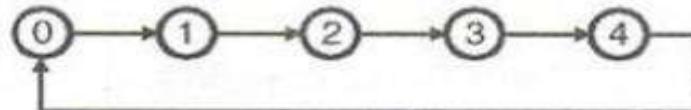
The timing diagram for a MOD-3 counter



Timing diagram for MOD-3 counter

Modulo-5 ripple counter using a 3-bit ripple counter

Step 1 : Draw the state diagram :



State diagram of a MOD-5 ripple counter

Step 2 : Write truth table for the reset logic :

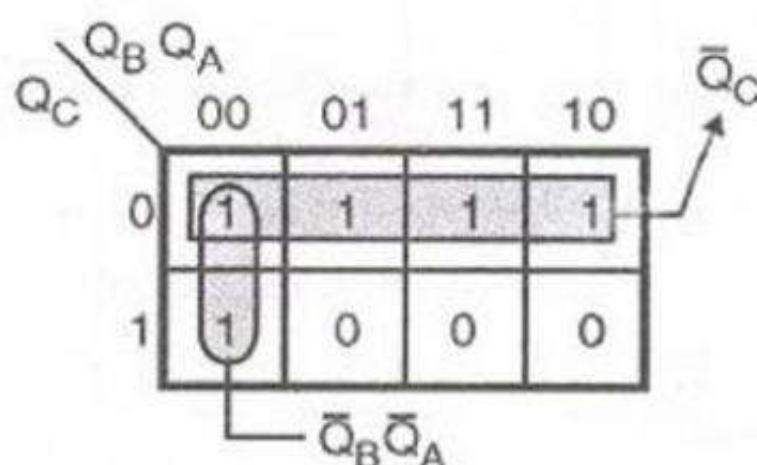
Truth table for the reset logic

State	Flip-flop outputs			Output Y of reset logic
	Q_C	Q_B	Q_A	
0	0	0	0	1
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

Valid states Invalid states

Modulo-5 ripple counter using a 3-bit ripple counter

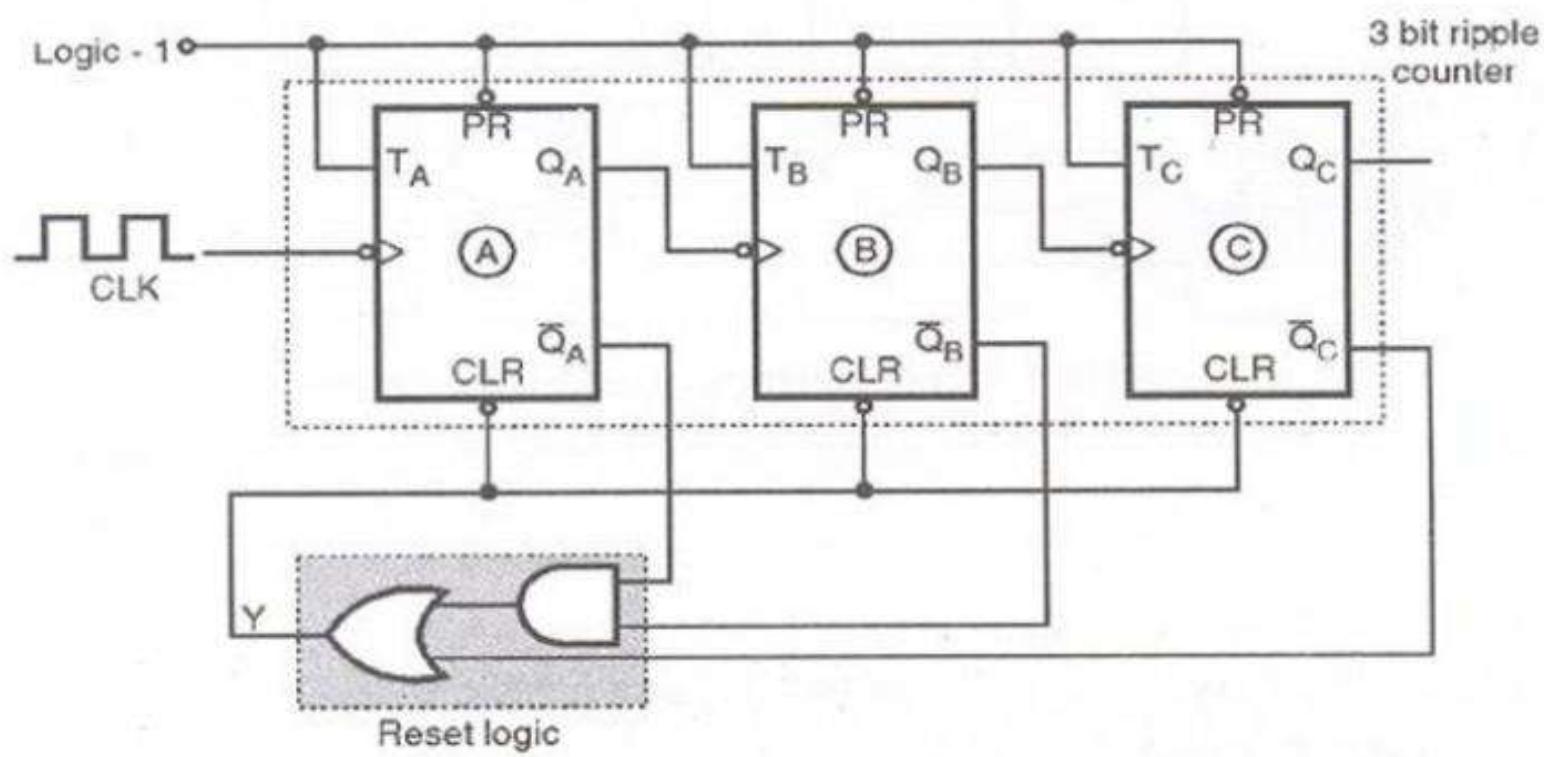
Step 3 : K-map and simplification for Y output :
For output Y



Expression for Y
 $Y = \bar{Q}_C + \bar{Q}_B \bar{Q}_A$

K-map and simplification

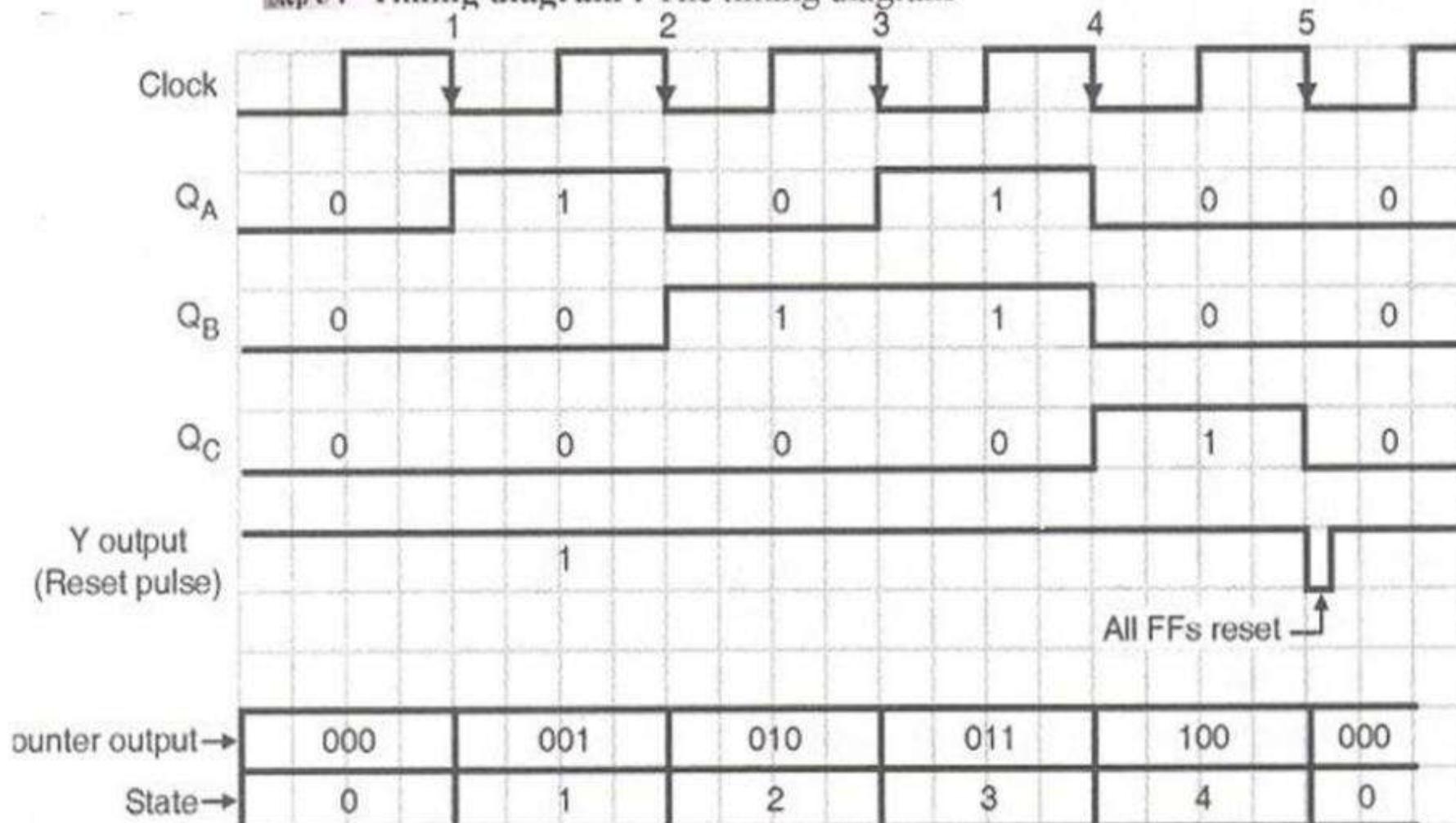
Modulo-5 ripple counter



Logic diagram of MOD-5 ripple counter

Modulo-5 ripple counter

Step 5: Timing diagram : The timing diagram



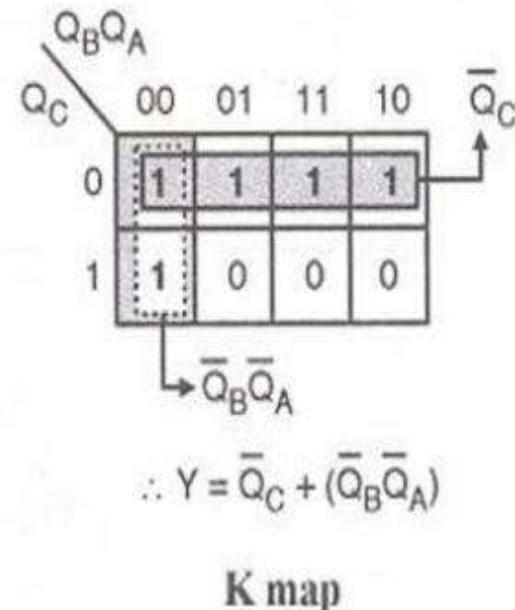
Timing diagram of MOD-5 ripple counter

MOD-6 asynchronous counter

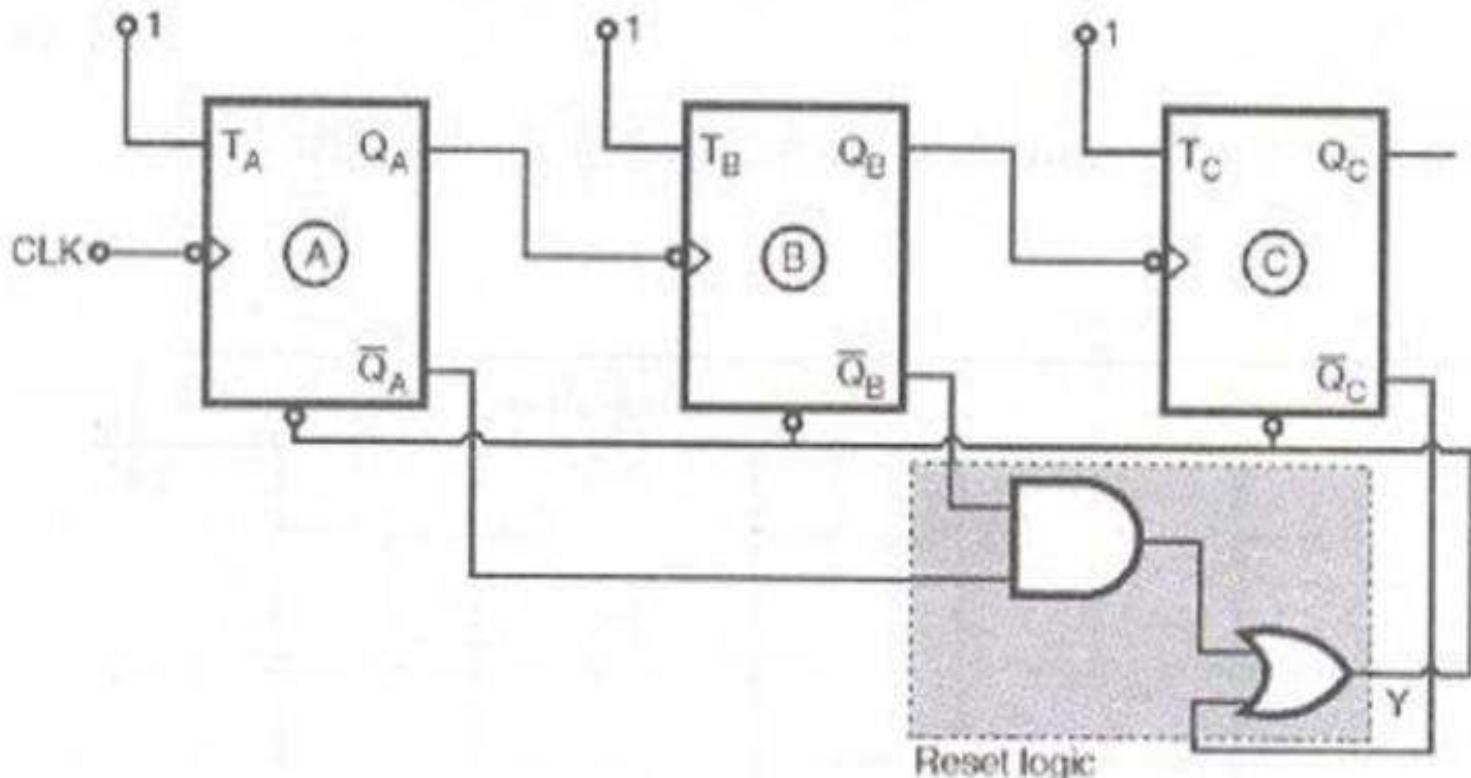
Step 1 : Write the truth table and K map for the output of reset logic.

Truth table

CLK	Q _C	Q _B	Q _A	Y
Initially	0	0	0	1
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	0	0	0	1



MOD-6 asynchronous counter



MOD-6 asynchronous counter

MOD-7 asynchronous counter up counting using T flip-flop

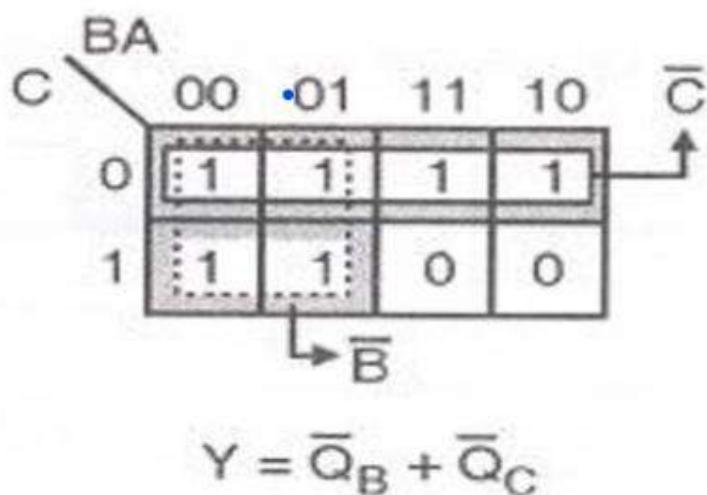
Step 1 : Truth table :

The truth table of divide by 7 asynchronous up counter

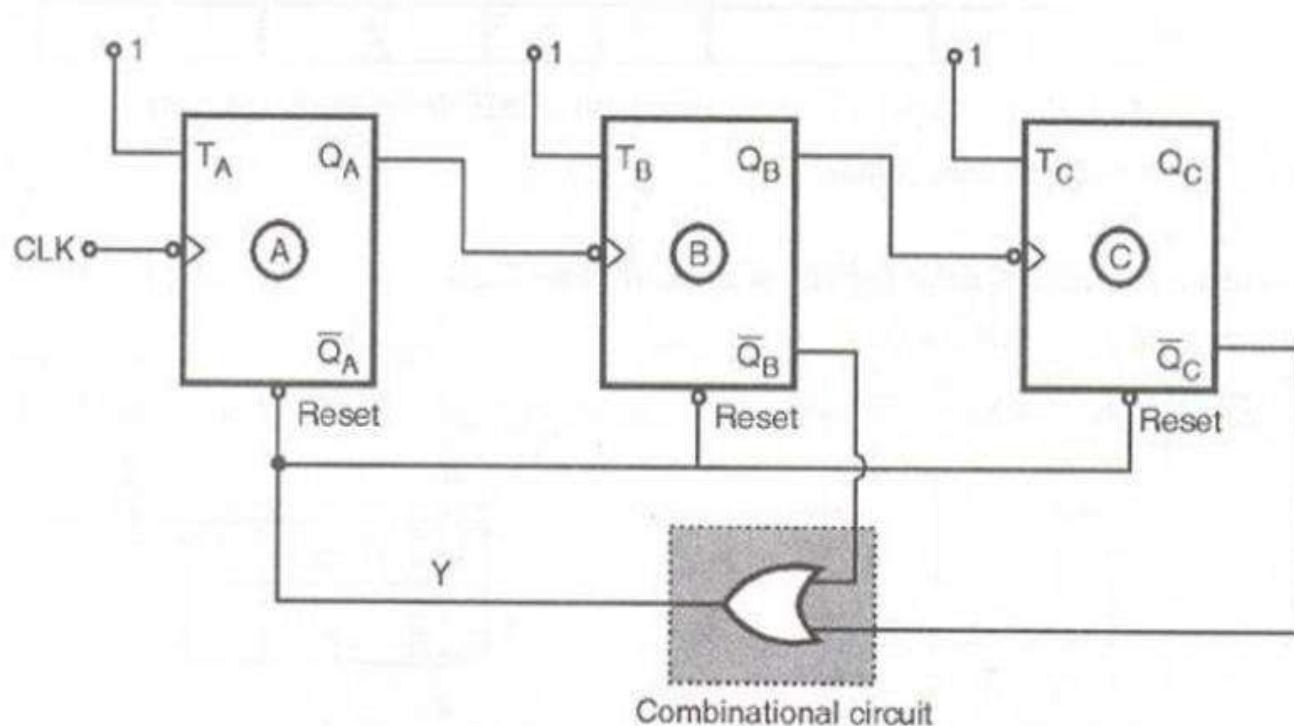
CLK	Q _C	Q _B	Q _A	Y output
Initially	0	0	0	1
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	0	0	0	1

MOD-7 asynchronous counter up counting using T flip-flop

Step 2 : K map :

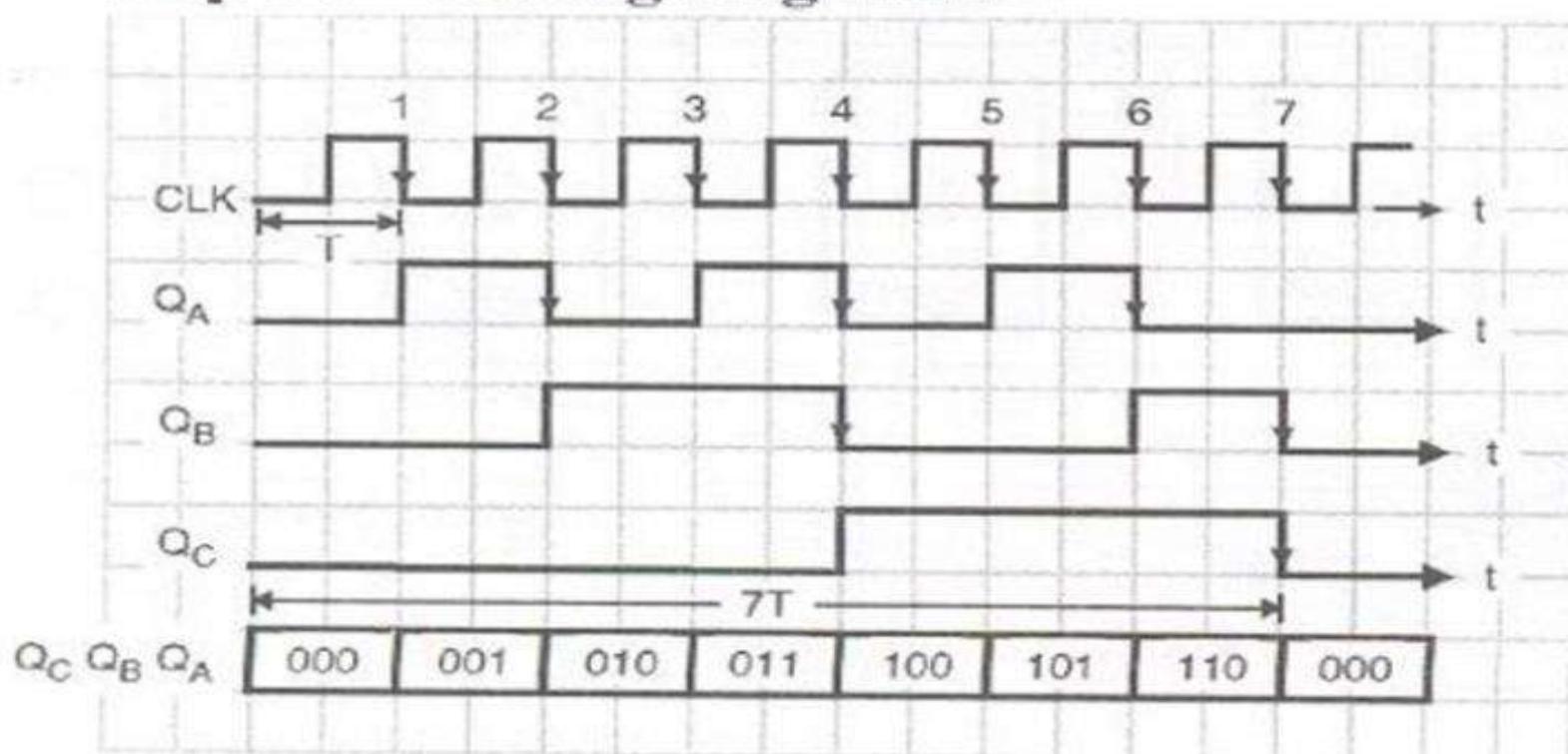


Step 3 : Draw the circuit :



MOD-7 asynchronous counter up counting using T flip-flop

Step 4 : Timing diagram :



Timing diagram of divide by 7 counter

MOD-12 asynchronous counter

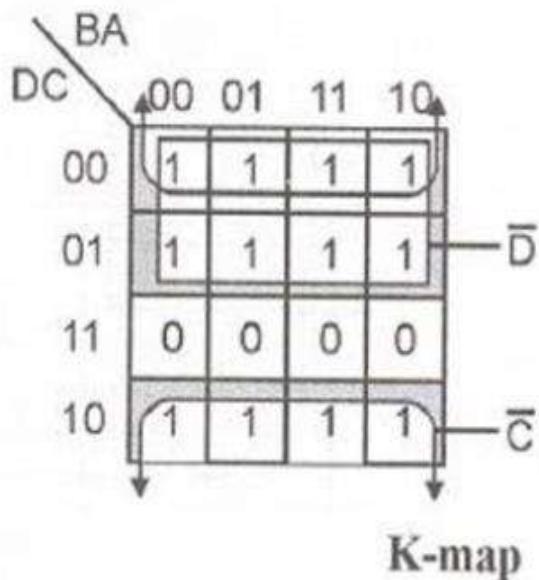
Step 1: Write truth table :

For a MOD-12 counter, there will be 12 distinct states. We have to use 4 flip flops.

CLK	D	C	B	A	Output Y
Initially	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	1
12	0	0	0	0	1

MOD-12 asynchronous counter

Step 2 : K-map for the output of the combination circuit

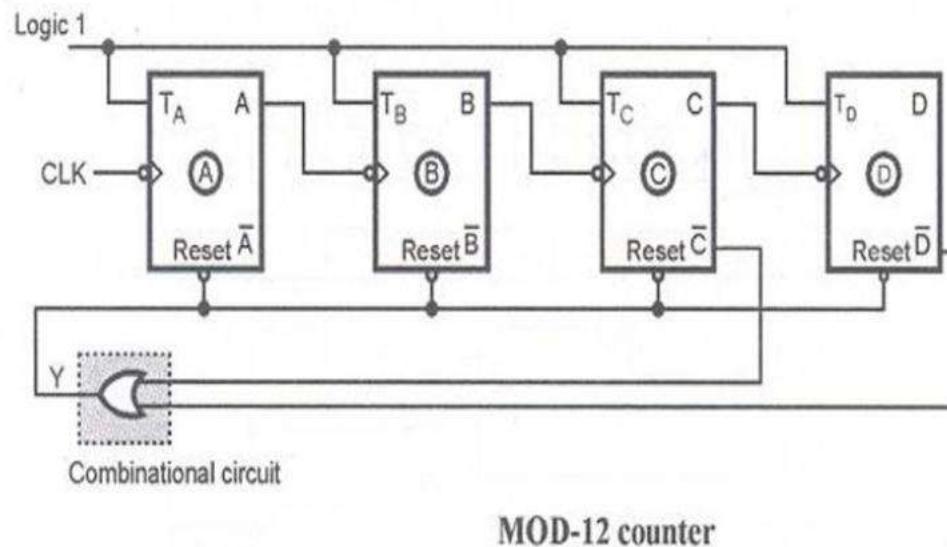


Simplified expression is given by

$$Y = \bar{C} + \bar{D}$$

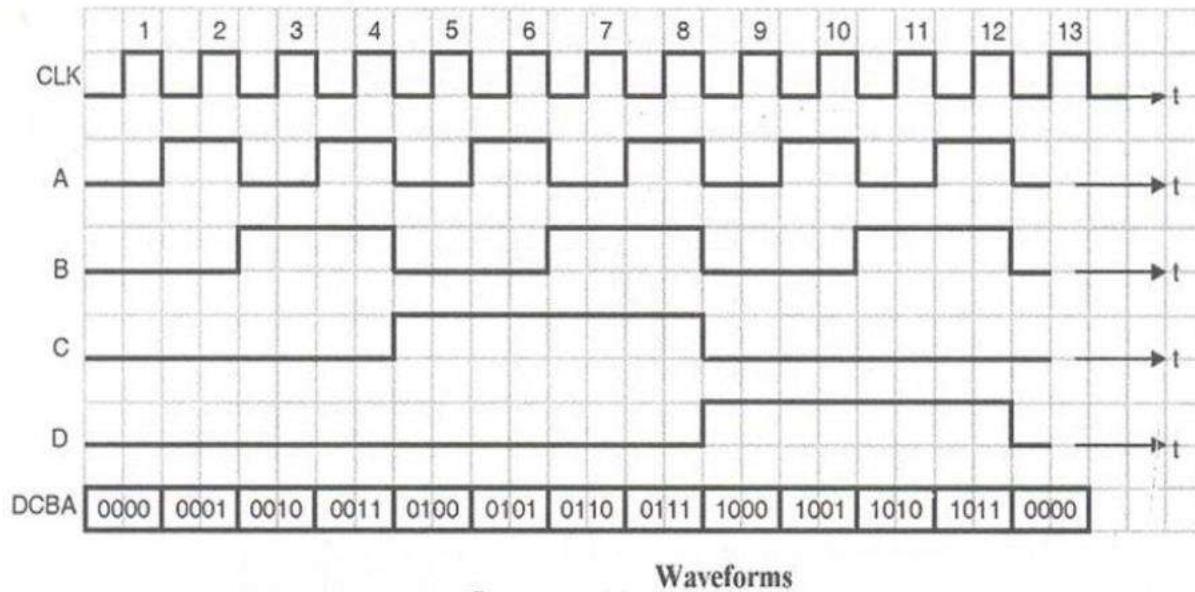
MOD-12 asynchronous counter

Step 3 : Draw the circuit :



MOD-12 asynchronous counter

Step 4 : Waveforms :



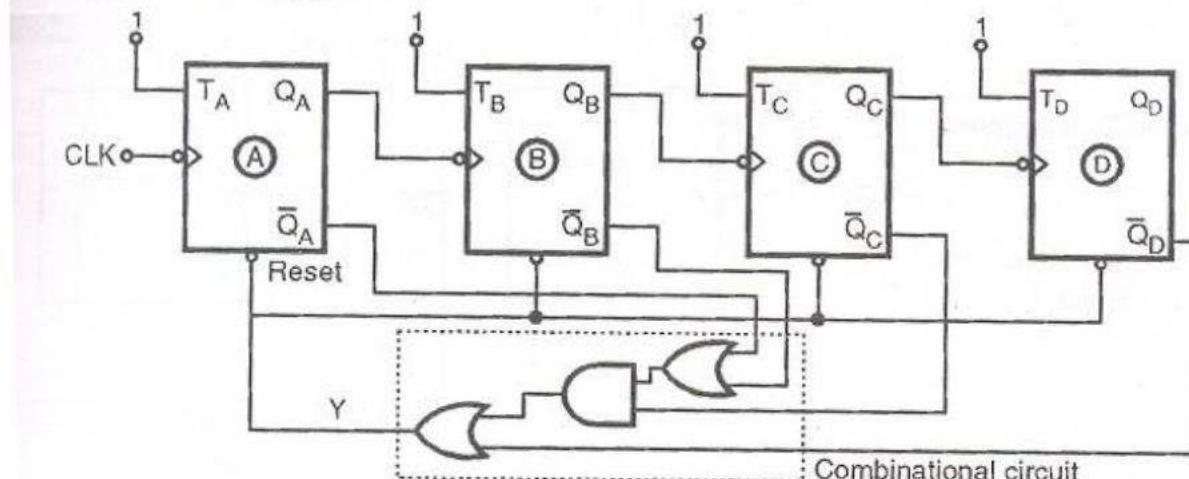
MOD-11 asynchronous counter

Step 1 : Write the truth table :

Q _D	Q _C	Q _B	Q _A	Y output
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
0	0	0	0	1

MOD-11 asynchronous counter

Draw the circuit diagram :

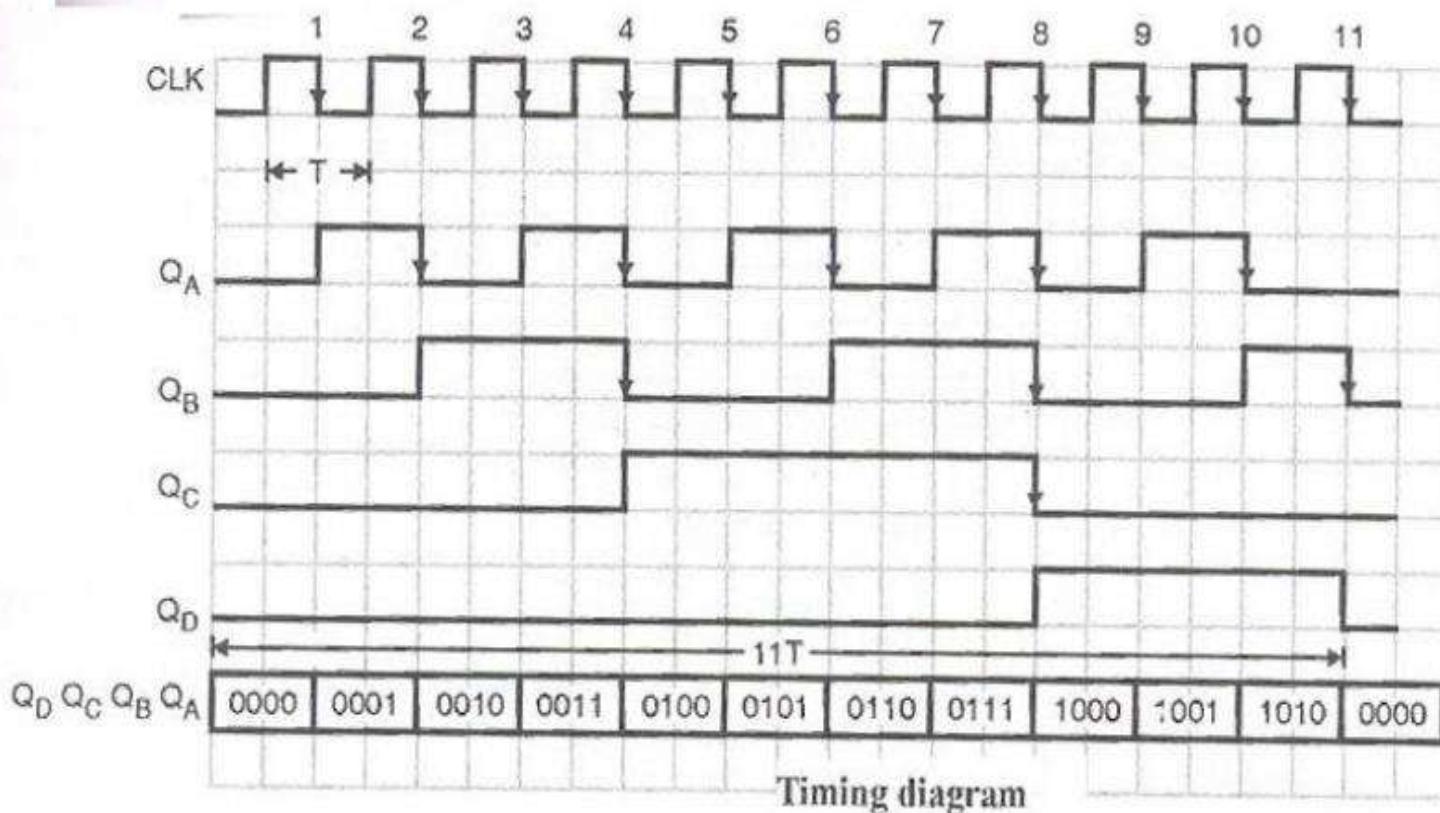


MOD 11 counter

MOD-11 asynchronous counter

Step 4 : Timing diagram :

- The timing diagram



MOD-11 asynchronous counter

$$\text{Output frequency } f_o = \frac{f_{\text{CLK}}}{11}$$

$$\therefore f_{\text{CLK}} = 11 f_o = 11 \times 11 \text{ kHz}$$

$$\therefore f_{\text{CLK}} = 121 \text{ kHz}$$

How many flip flops are required to construct the following modulus counter ? why
(A) 5 (B) 83 (C) 99 (D) 10

Soln. : In general m number of flip-flops are required to construct Mod-n counter, where

$$n \leq 2^m$$

	MOD n counter	Number of flip-flops (m)
A	5	3
B	83	7
C	99	7
D	10	4

How many flip-flops are required to count 16 clock pulses ? Why ?

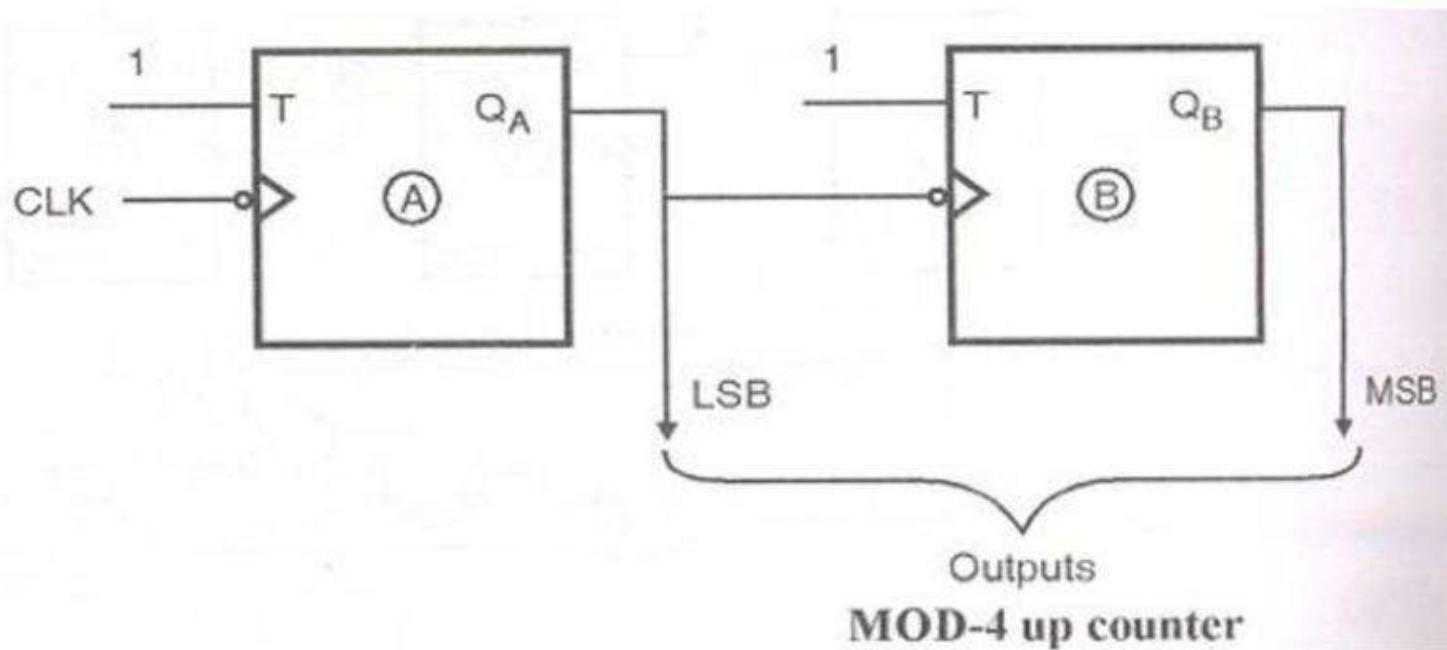
Soln. : To count n clock pulses m flip-flops are required, where,

$$n \leq 2^m$$

Hence to count 16 clock pulses 4 flip-flops are required, since $16 = 2^4$.

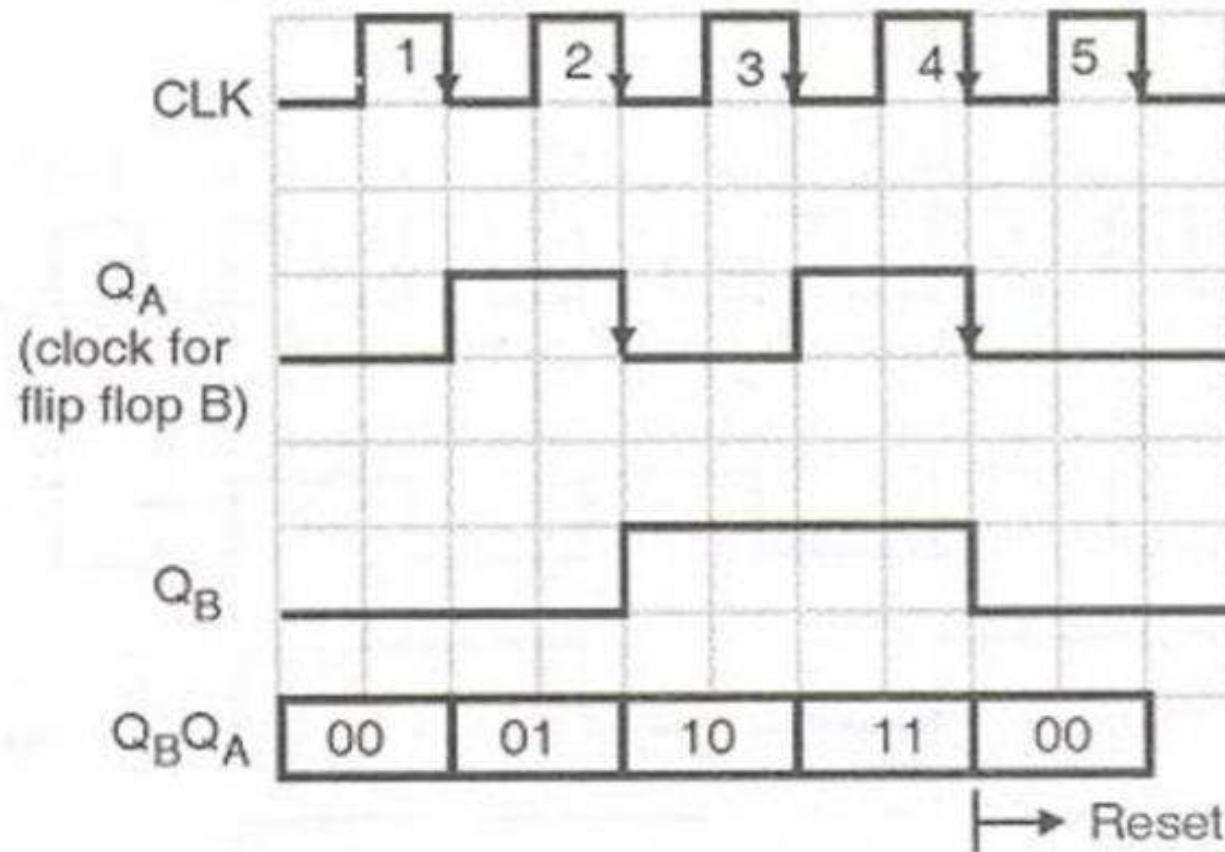
MOD-4 asynchronous counter

$$\therefore 2^n = 4$$
$$\therefore n = 2$$



MOD-4 asynchronous counter

The timing diagram



Registers

Register:

- ❖ A set of flip-flops, possibly with added combinational gates, that perform data-processing tasks.

- ❖ Store and manipulate information in a digital system.

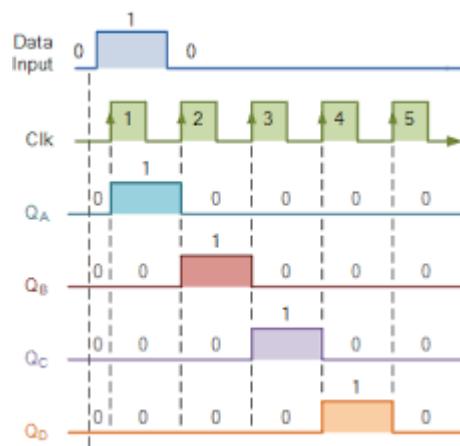
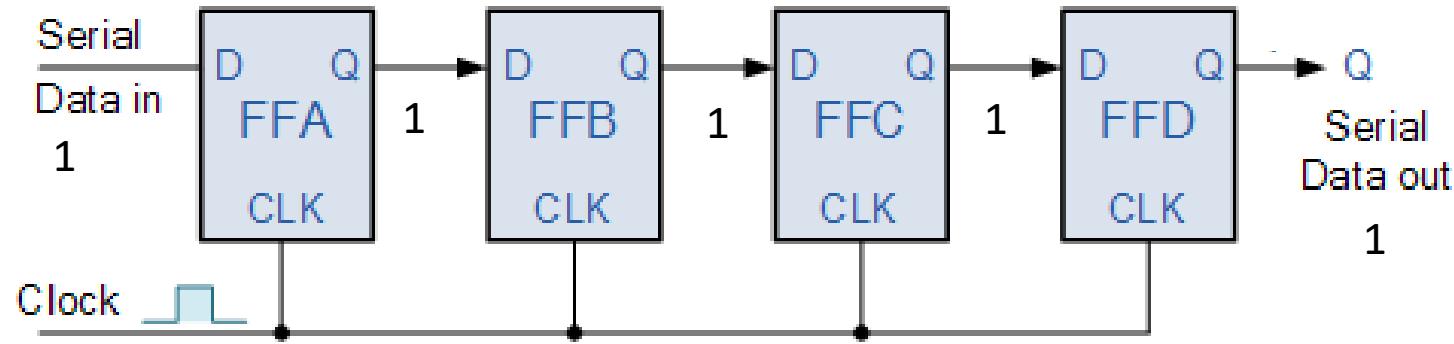
Shift Register

1. Serial In Serial Out (SISO)
2. Serial In Parallel Out (SIPO)
3. Parallel In Serial Out (PISO)
4. Parallel In Parallel Out (PIPO)

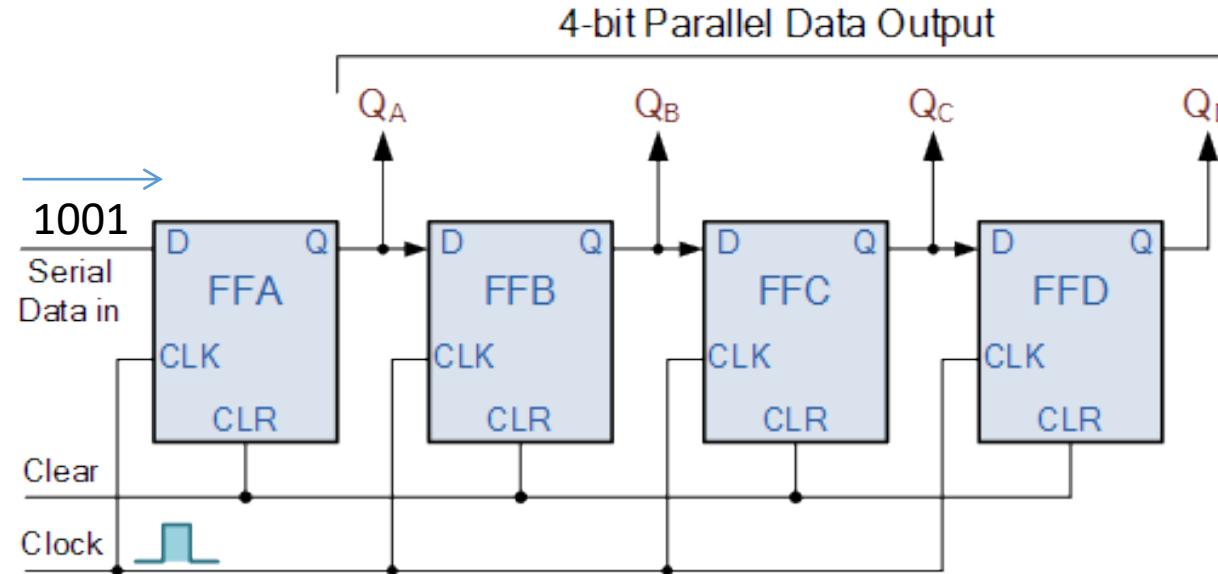
Note:

Serial → Single
Parallel → Many

Serial In Serial Out

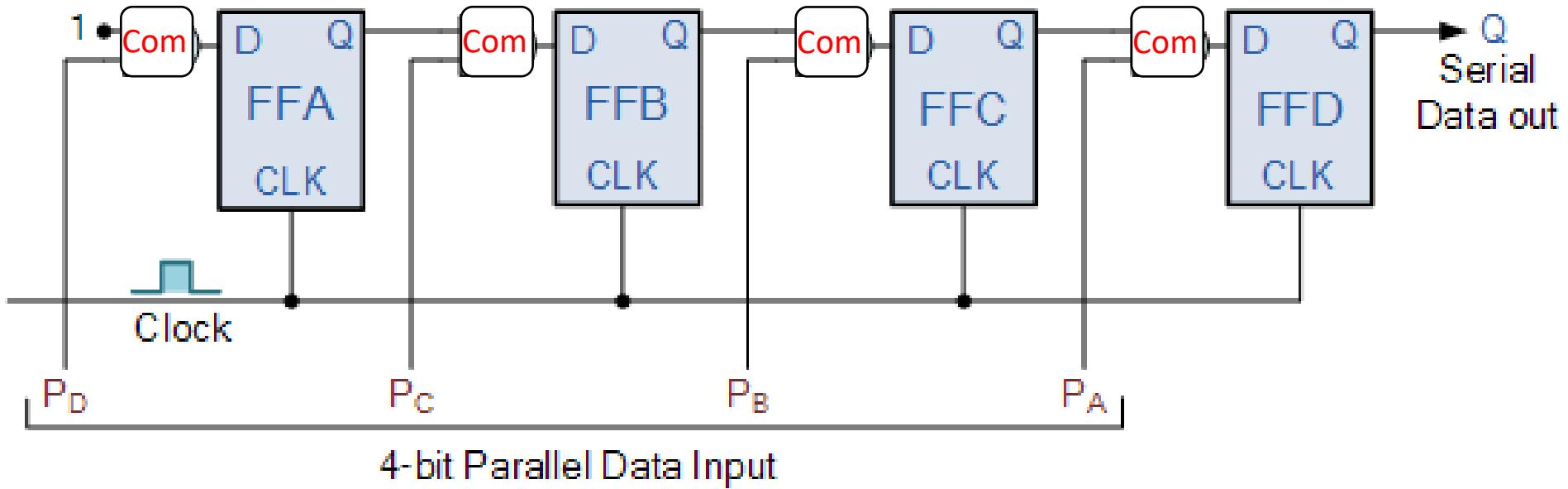


Serial In Parallel Out



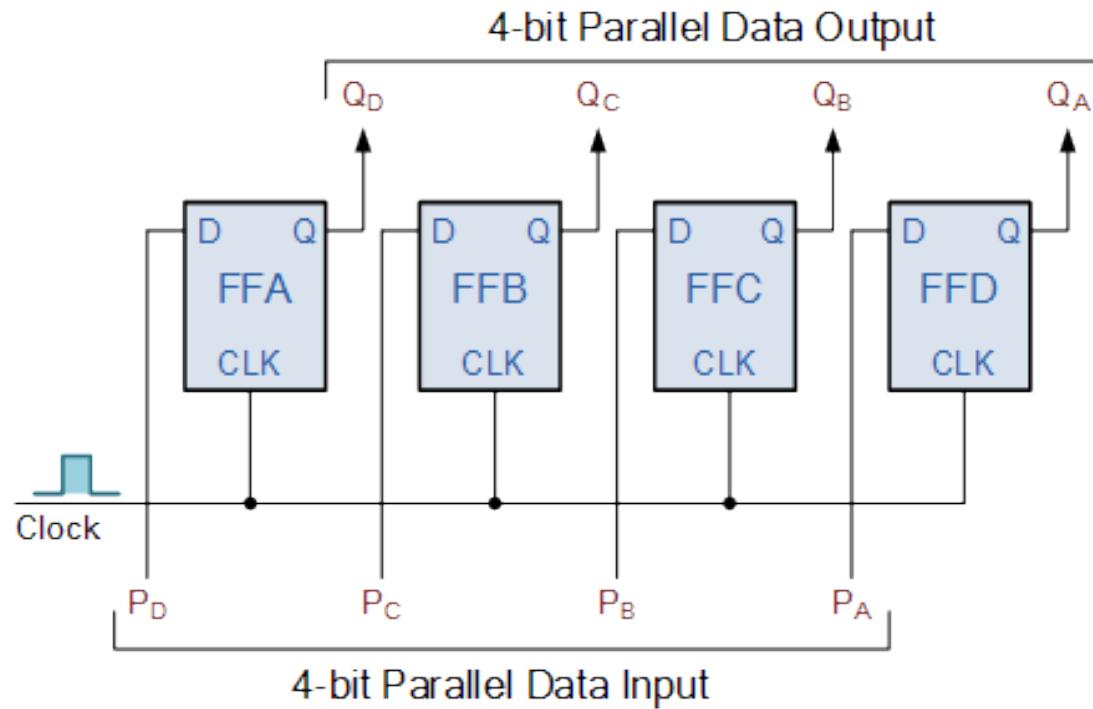
Clear	FF0	FF1	FF2	FF3
1001	0	0	0	0
	1	0	0	0
	0	1	0	0
	0	0	1	0
	1	0	0	1

Parallel In Serial Out



Com → Combinational Circuit (Any)

Parallel In Parallel Out



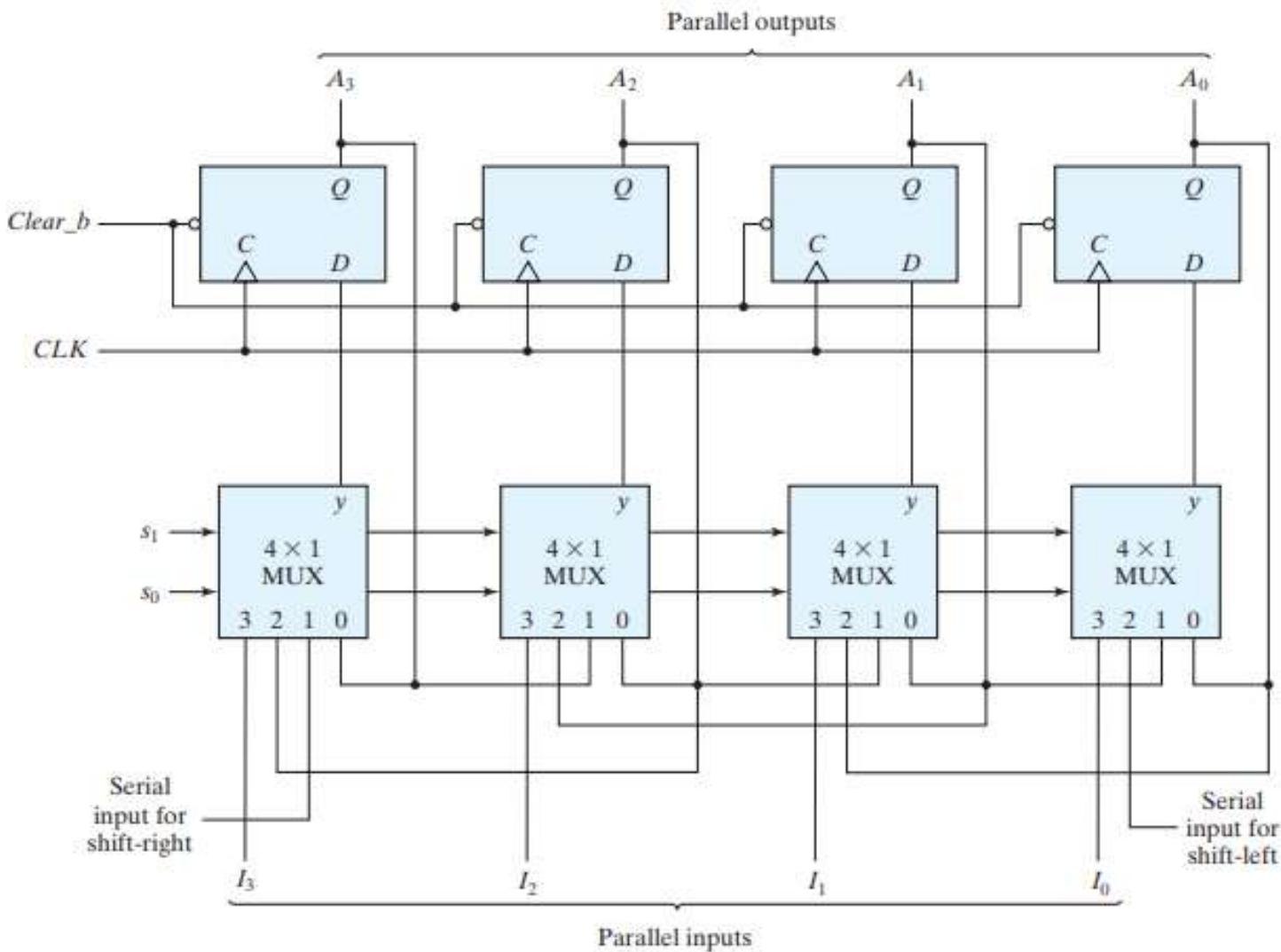
This is ordinary register and it won't perform any shift operation.

Universal Shift Register

Function Table for the Register of Fig. 6.7

Mode Control

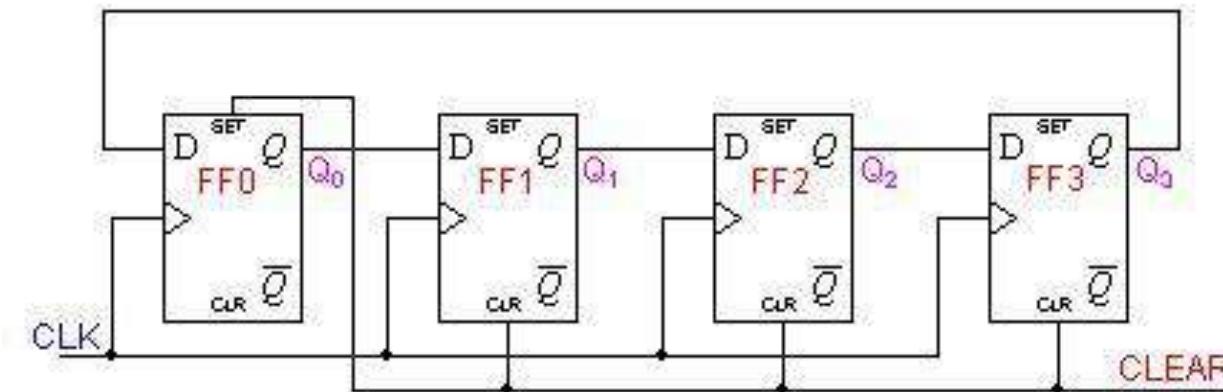
s_1	s_0	Register Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load



Counter

- ❖ The common clock pulse triggers all flip-flops simultaneously
- ❖ Hardware may be more complex but more reliable

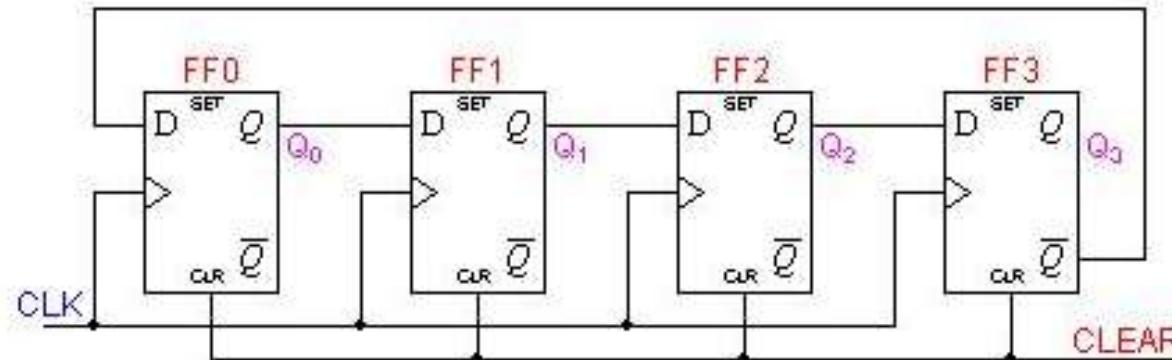
Ring Counter (Using Shift Register)



Clock pulse	Q0	Q1	Q2	Q3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

Also known as mod-4 counter (mod-n counter)

Johnson Counter (Twisted Ring Counter)



Clock pulse	Q0	Q1	Q2	Q3
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

Also known as mod- 2^n counter

BCE102L
DIGITAL SYSTEM DESIGN

Module:5

**DESIGN OF REGISTERS AND
COUNTERS**

Courtesy: M. Morris Mano, "Digital Design", 3rd Edition, *Prentice Hall of India Pvt. Ltd.*, 2008 and its "e-book" version.

COUNTERS

Types of Counter

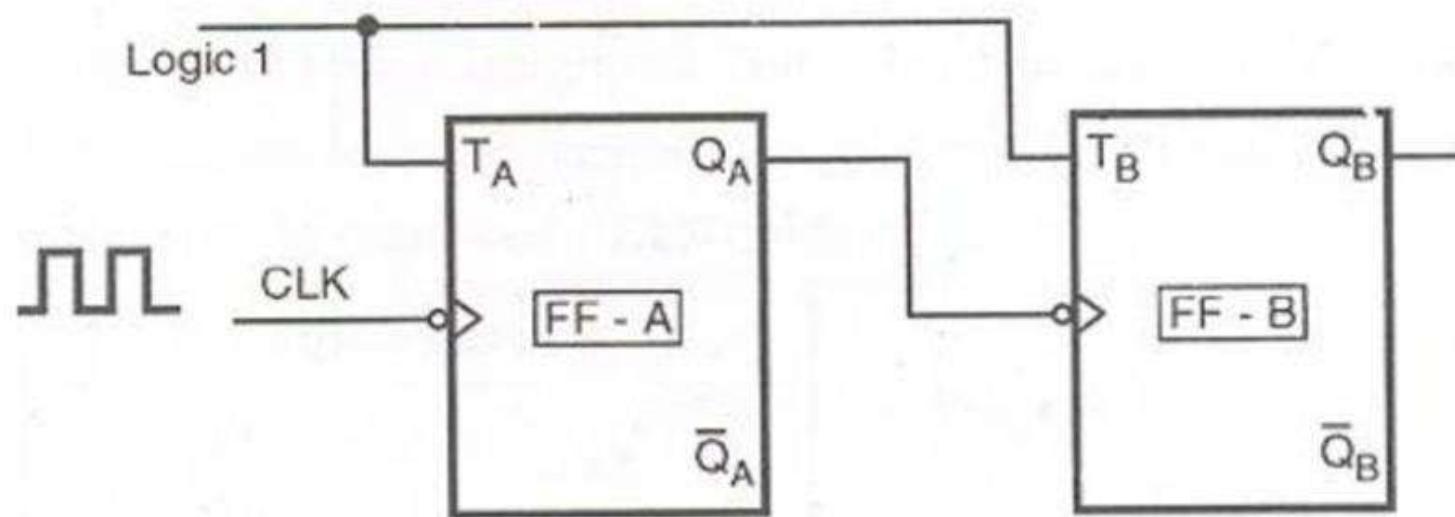
- Asynchronous counter
- Synchronous counter
- Asynchronous counter :- for these counter the external clock signal is applied to one flip-flop & then the o/p of preceding flip-flop connected to clock of next flip-flop.
- Synchronous counter :- for these counter all the flip-flop receive the external clock pulse simultaneously.

Asynchronous counter

- Counter means count no of clock pulses which counter has received.
- Up counter count up counting i.e. 0 to 3 etc.
- No. of state depends on 2^n n is no of f/fs
- The maximum count is $(2^n - 1)$

Asynchronous counter

- 2-bit Asynchronous counter Up counter (Ripple Counter)



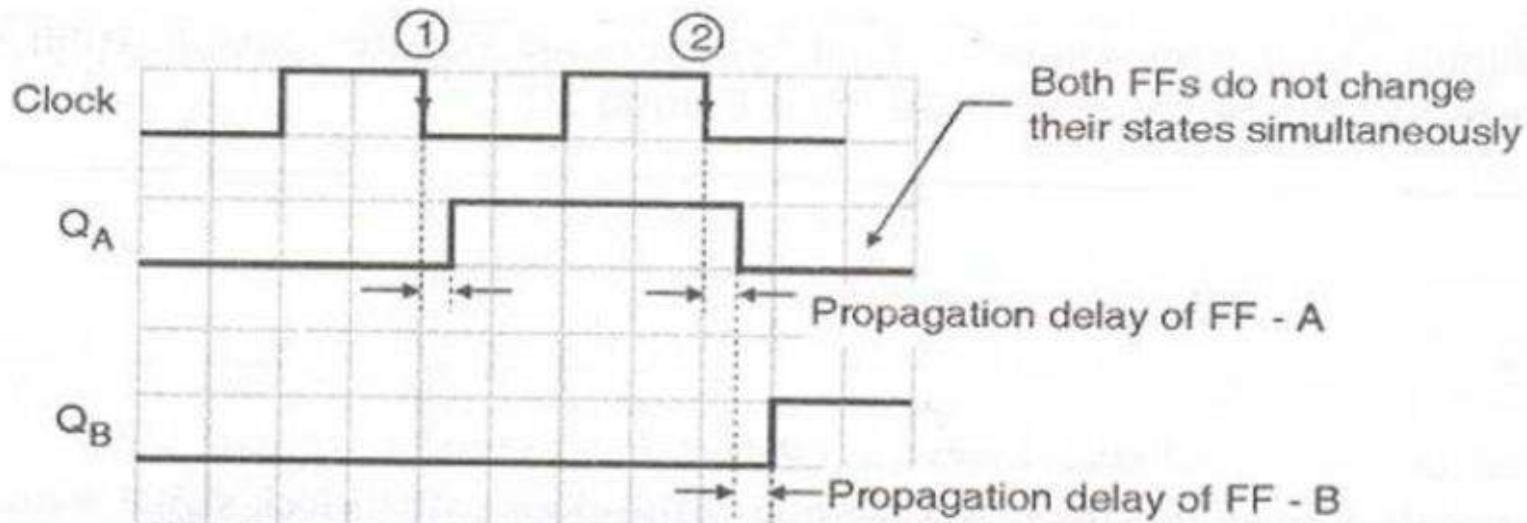
A two bit asynchronous binary counter

2-bit Asynchronous counter Up counter (Ripple Counter)

first falling edge of the clock hits FF-A, it will toggle as $T_A = 1$. Hence Q_A will be equal to 1.

$$Q_B Q_A = 00$$

$$Q_B Q_A = 10$$

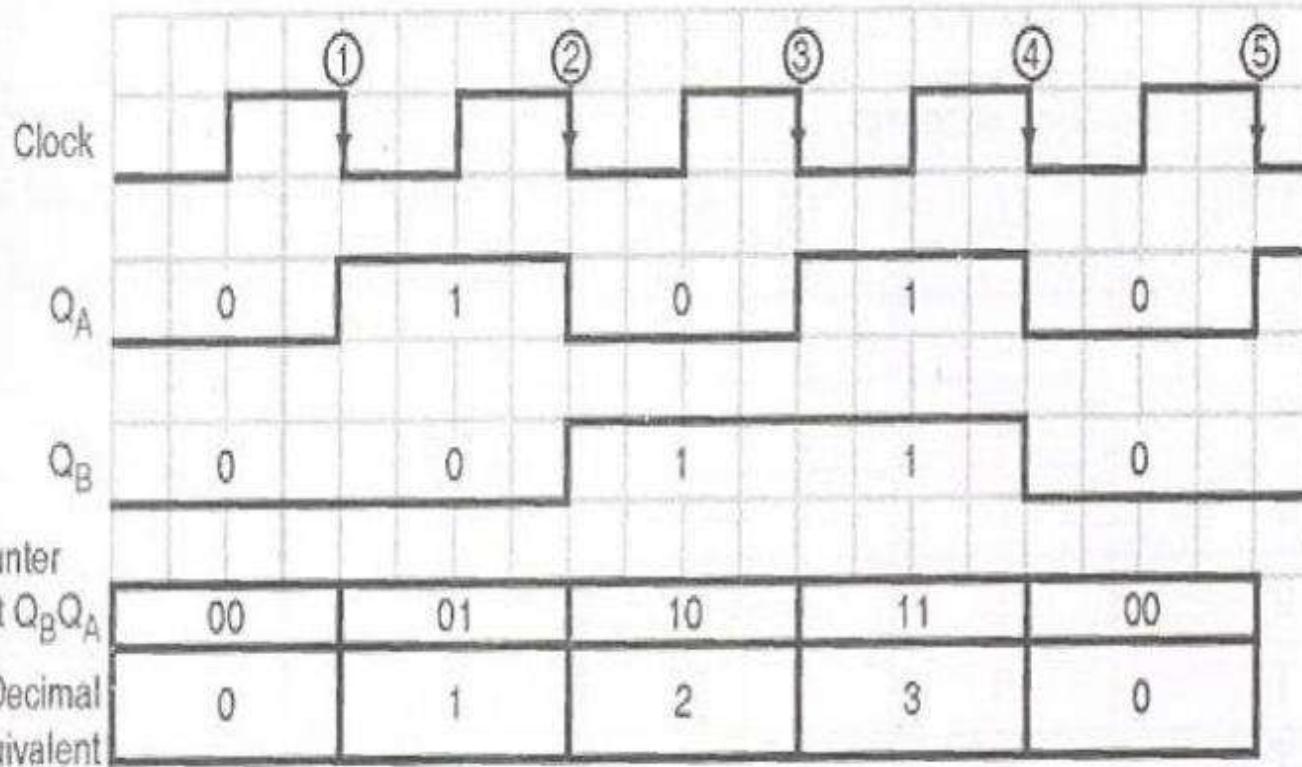


2-bit Asynchronous counter Up counter (Ripple Counter)

operation of a 2-bit binary ripple counter

Clock	Counter outputs		State number	Decimal equivalent of counter output
	Q_B (MSB)	Q_A (LSB)		
Initially	0	0	-	0
1 st (\downarrow)	0	1	1	1
2 nd (\downarrow)	1	0	2	2
3 rd (\downarrow)	1	1	3	3
4 th (\downarrow)	0	0	4	0

2-bit Asynchronous counter Up counter (Ripple Counter)

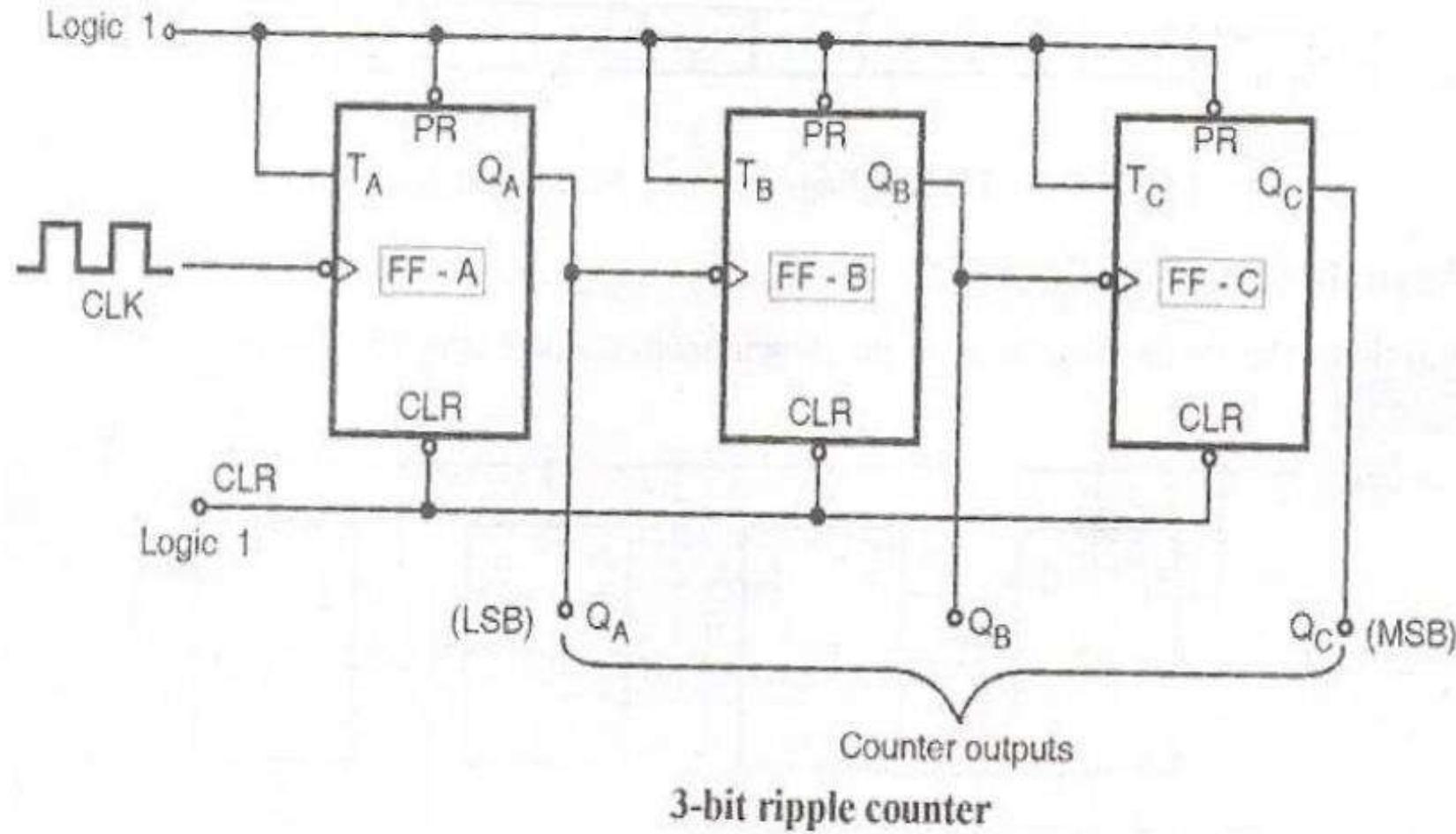


Note: Propagation delays have been ignored while drawing these waveforms

FF-A toggles at every negative CLK edge

Timing diagram for a 2 bit ripple counter

3-bit Asynchronous counter Up counter (Ripple Counter)

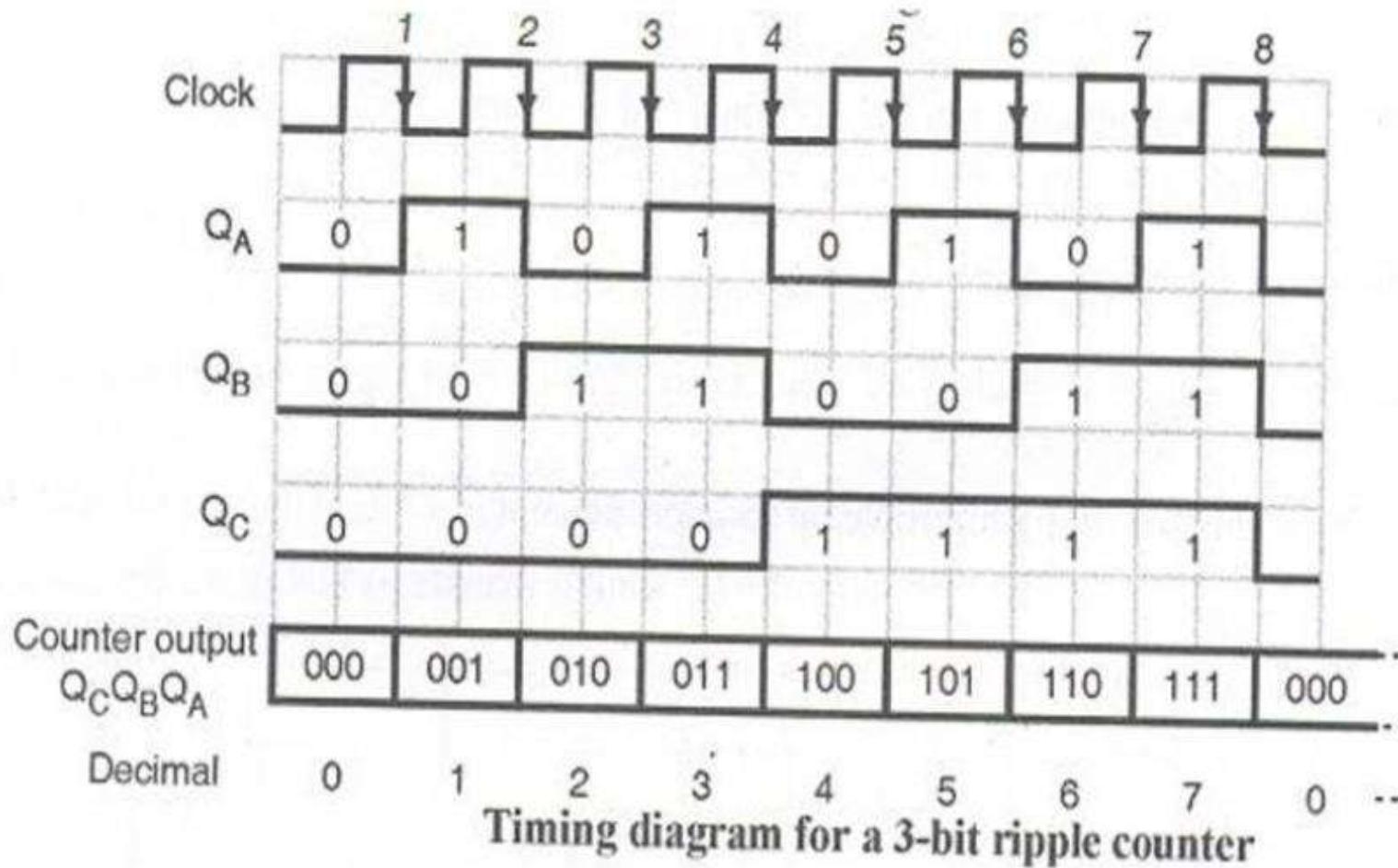


3-bit Asynchronous counter Up counter (Ripple Counter)

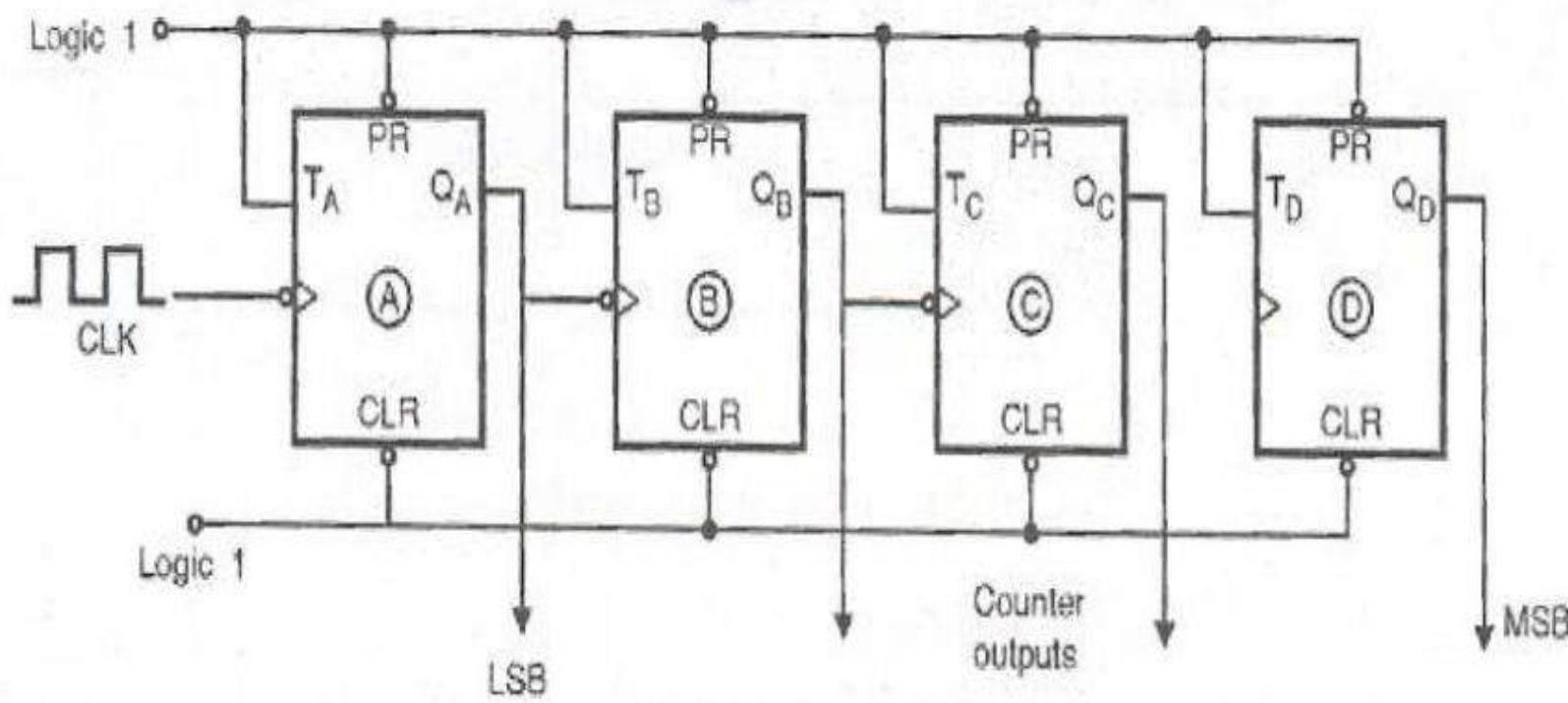
operation of a 3-bit ripple counter

Clock	Flip-flop outputs			State	Decimal equivalent
	Q_C (MSB)	Q_B	Q_A (LSB)		
Initially	0	0	0	1	0
1 st (\downarrow)	0	0	1	2	1
2 nd (\downarrow)	0	1	0	3	2
3 rd (\downarrow)	0	1	1	4	3
4 th (\downarrow)	1	0	0	5	4
5 th (\downarrow)	1	0	1	6	5
6 th (\downarrow)	1	1	0	7	6
7 th (\downarrow)	1	1	1	8	7
8 th (\downarrow)	0	0	0	1	0

3-bit Asynchronous counter Up counter (Ripple Counter)



4-bit Asynchronous counter Up counter (Ripple Counter)



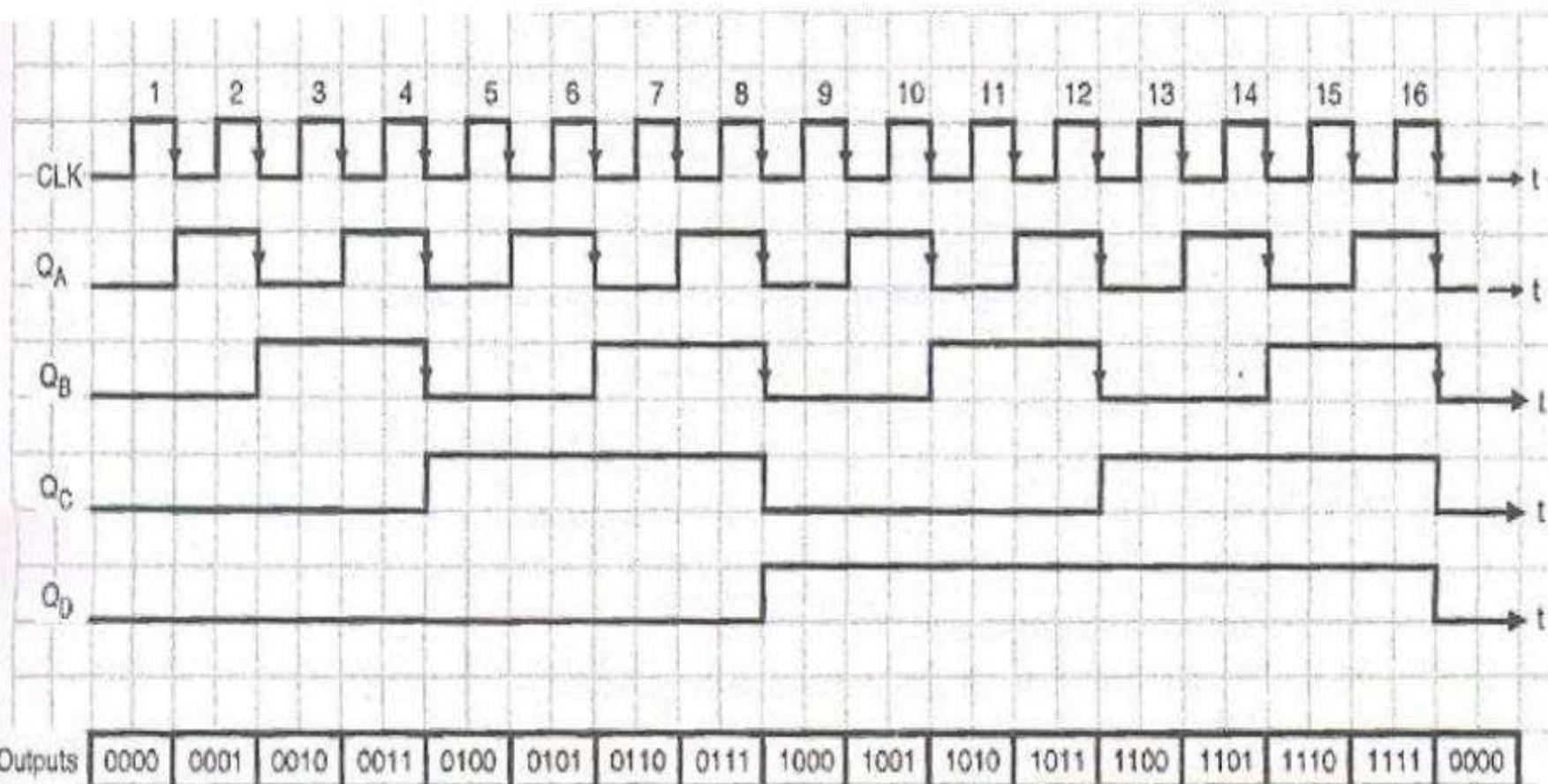
4 bit asynchronous up counter

4-bit Asynchronous counter Up counter (Ripple Counter)

Truth table for a 4-bit asynchronous up counter

Clock	FF outputs.			
	Q_D	Q_C	Q_B	Q_A
Initially	0	0	0	0
1 (\downarrow)	0	0	0	1
2 (\downarrow)	0	0	1	0
3 (\downarrow)	0	0	1	1
:				
:				
14 (\downarrow)	1	1	1	0
15 (\downarrow)	1	1	1	1
16 (\downarrow)	0	0	0	0

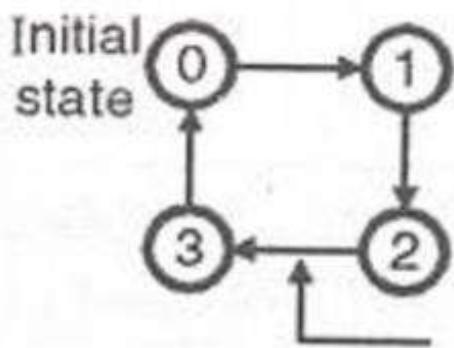
4-bit Asynchronous counter Up counter (Ripple Counter)



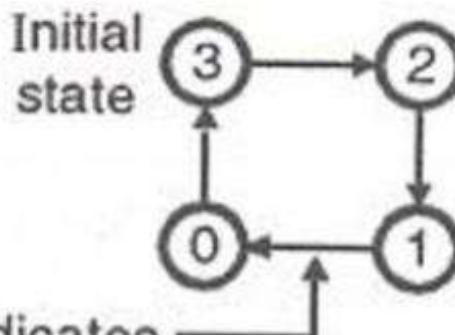
Waveforms of a 4 bit asynchronous up counter

State diagram

- The state diagram represents the states of a counter graphically.



Arrow indicates
the direction
of counting



- (a) For a 2-bit up counter (b) For a 2-bit down counter

State diagram

Module:4 Design of data path circuits

N-bit Parallel Adder/Subtractor, Carry Look Ahead Adder, Unsigned Array Multiplier, Booth Multiplier, 4-Bit Magnitude comparator. Modeling of data path circuits using Verilog HDL [6 Hours]

Binary Adder – Sum of Two Binary Numbers

- An n-bit adder requires n full adders with each output carry connected to the input carry of the next higher-order full adder
- 4-bit adder: Interconnection of four full adder (FA) circuits

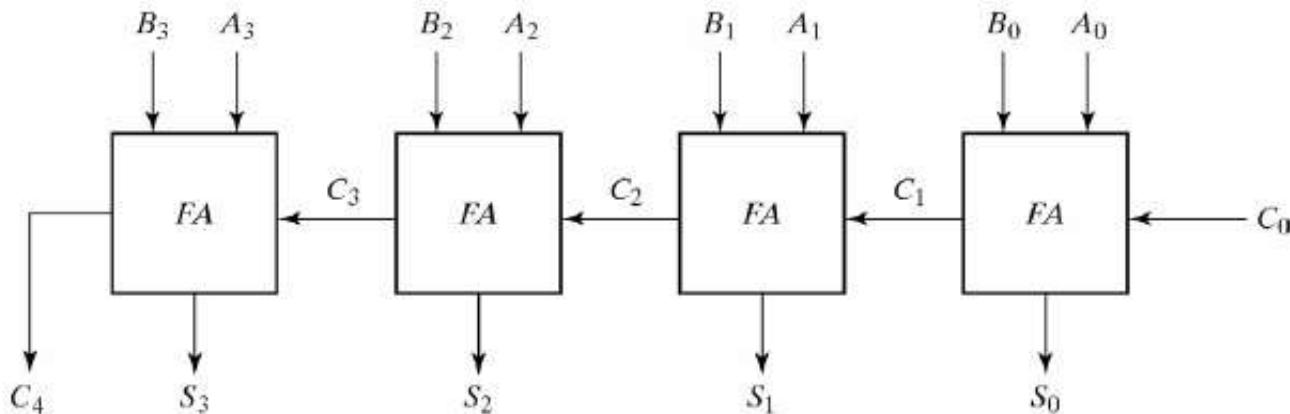


Fig. 4-9 4-Bit Adder

Example: $A=1011$
and $B=0011$

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}

Carry Propagation

- Total propagation time
 - = propagation delay of a typical gate * number of gate levels in the circuit
- The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders
 - Inputs A_i and B_i are available as soon as input signals are applied to the adder
 - The value of S_i in any given stage will be in its steady state final value only after the input carry to that stage has been propagated
 - C_3 has to wait for C_2 , C_2 has to wait for C_1 and so on down to C_0
- A limiting factor on the speed with which two numbers are added

Carry Propagation of a Full Adder

- The signal from the input carry C_i to the output carry C_{i+1} propagates through an AND gate and an OR gate, which constitute two gate levels
 - For an n -bit adder, there are $2n$ gate levels for the carry to propagate from input to output
- 4-bit adder
 - Carry propagation: $2 * 4 = 8$ gate levels from C_0 to C_4
 - Critical path: 9 gate levels (3 for C_0)

Reducing Carry Propagation Delay

1. Employ faster gates with reduced delays
 - Physical circuits have a limit to their capability
2. Complicated techniques for parallel adders
 - Principle of **carry lookahead**

Definitions of two new variables

- carry generate G_i : produces a carry of 1 when both A_i and B_i are 1, regardless of the input carry C_i
- carry propagate P_i : the term associated with the propagation of the carry from C_i to C_{i+1}

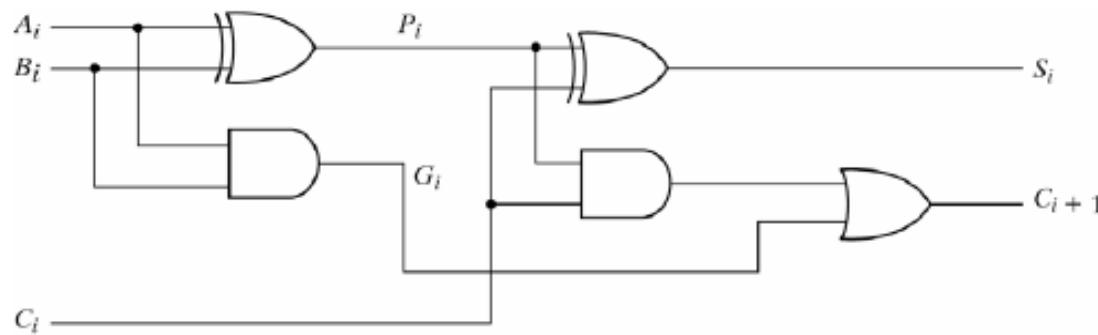


Fig. 4-10 Full Adder with P and G Shown

$$\begin{aligned}P_i &= A_i \oplus B_i \\G_i &= A_i B_i \\ \text{sum } S_i &= P_i \oplus C_i \\ \text{carry } C_{i+1} &= G_i + P_i C_i\end{aligned}$$

Carry Lookahead Generator

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

the output sum and carry can respectively be expressed as

$$S_i = P_i \oplus C_i$$

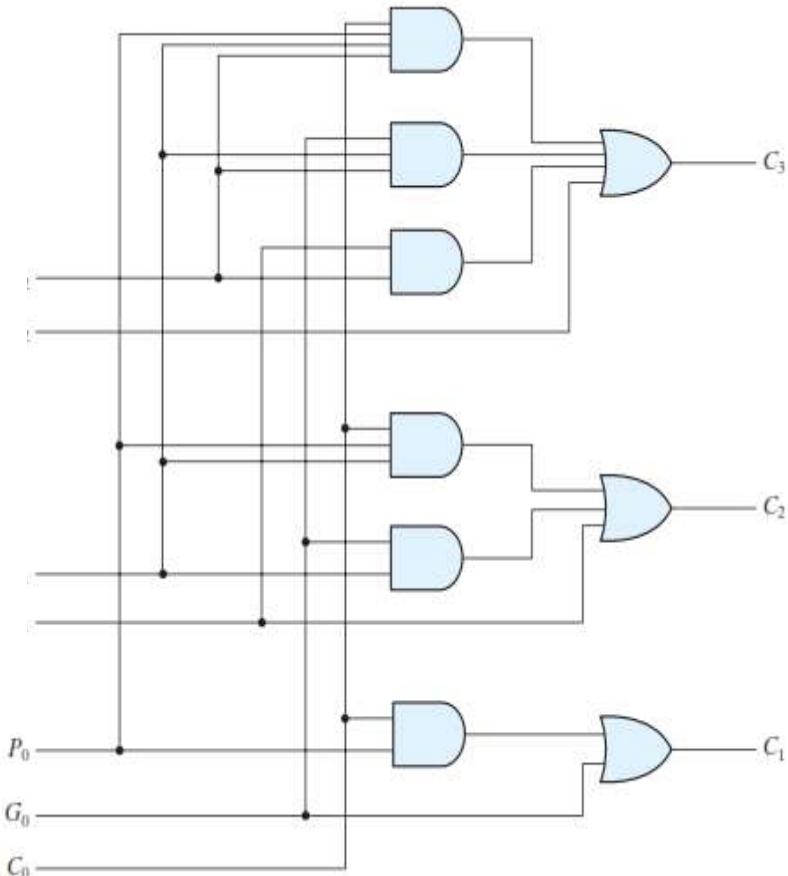
$$C_{i+1} = G_i + P_i C_i$$

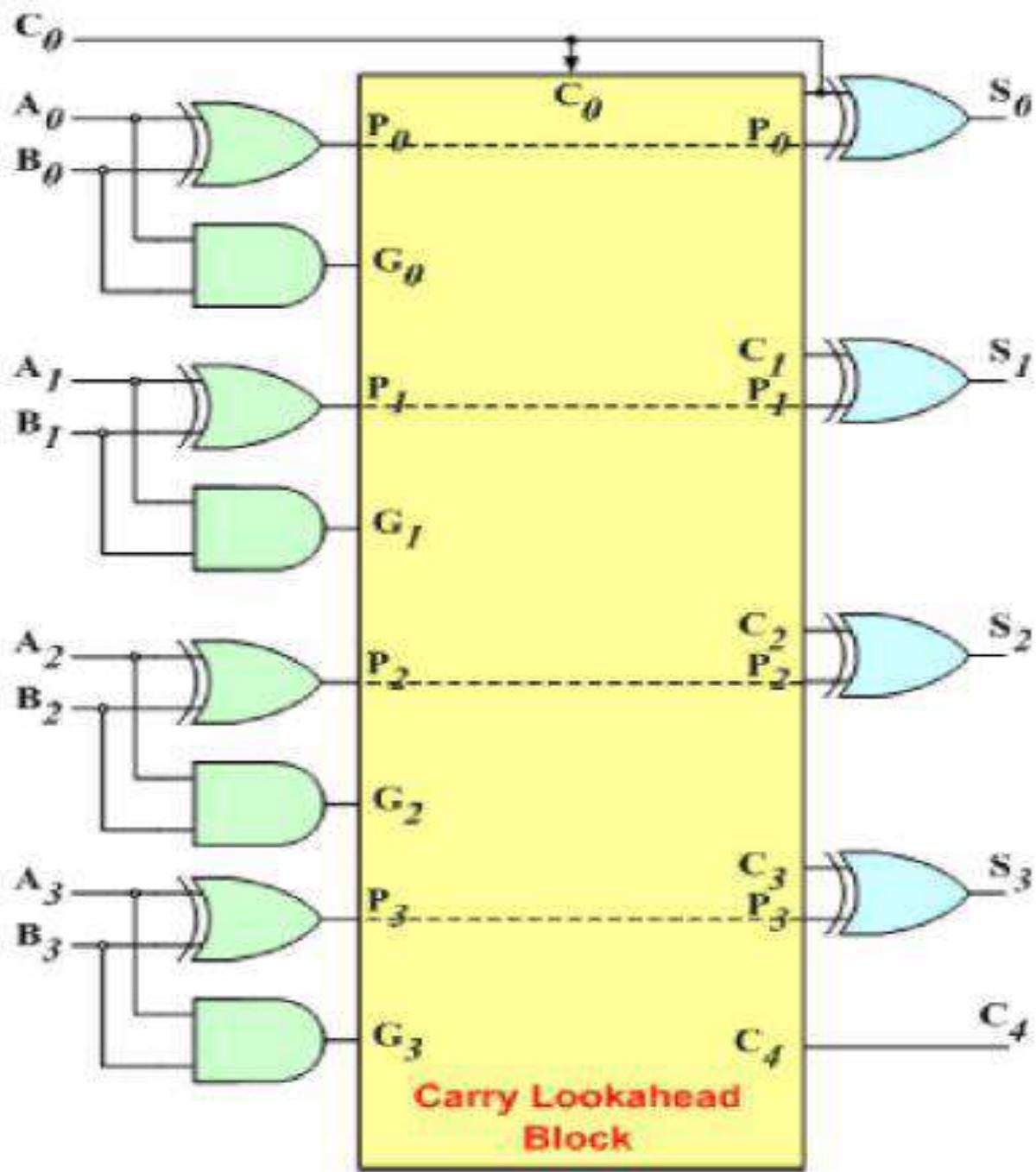
C_0 = input carry

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 = P_2 P_1 P_0 C_0$$





4-bit CARRY LOOK-AHEAD ADDER

$$P_0 = A_0 \oplus B_0$$

$$P_1 = A_1 \oplus B_1$$

$$P_2 = A_2 \oplus B_2$$

$$P_3 = A_3 \oplus B_3$$

$$S_0 = P_0 \oplus C_0$$

$$S_1 = P_1 \oplus G$$

$$S_2 = P_2 \oplus G$$

$$S_3 = P_3 \oplus C_3$$

$$G = P_0 C_0 + G_0$$

$$C_2 = P_1 G + G_{1L} = P_1 (P_0 C_0 + G_0) + G_1$$

$$C_3 = P_2 C_2 + G_3 = P_2 (G_1 + P_1 G_0 + P_1 P_0 C_0) + G_2$$

$$C_4 = P_3 C_3 + G_3$$

$$= P_3 (G_2 + P_2 G_1 + P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0) + G_3$$

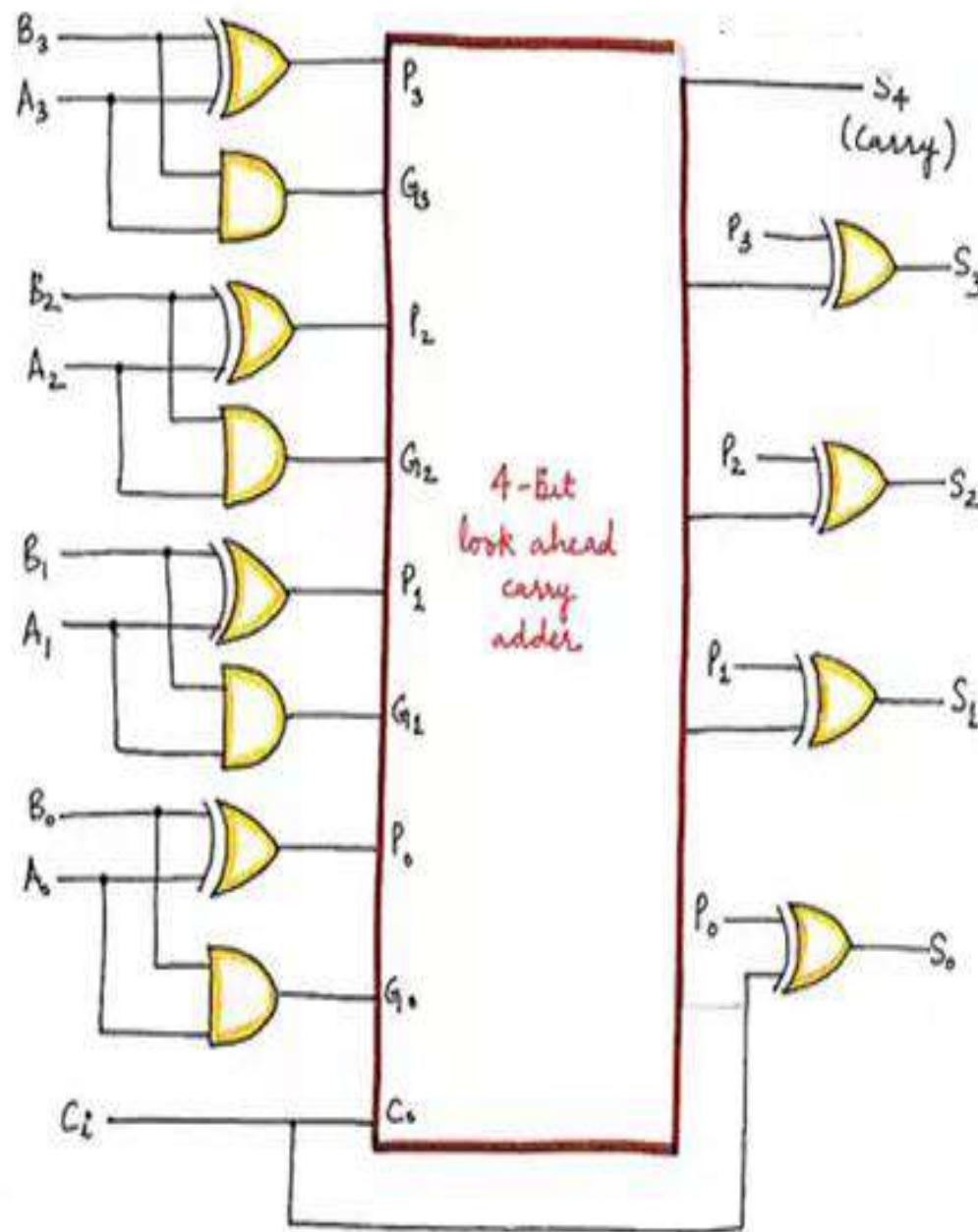
$$G_{10} = A_0 B_0$$

$$G_{11} = A_1 B_1$$

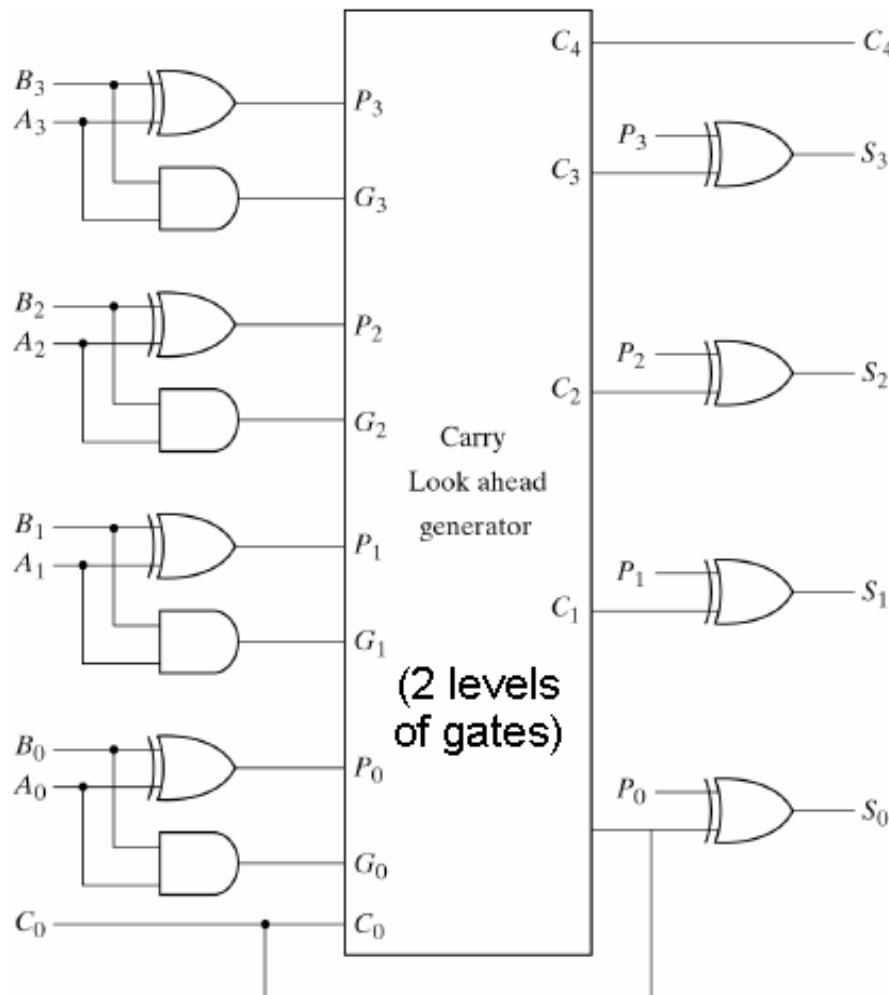
$$G_{12} = A_2 B_2$$

$$G_{13} = A_3 B_3$$

C_0 = O/p carry



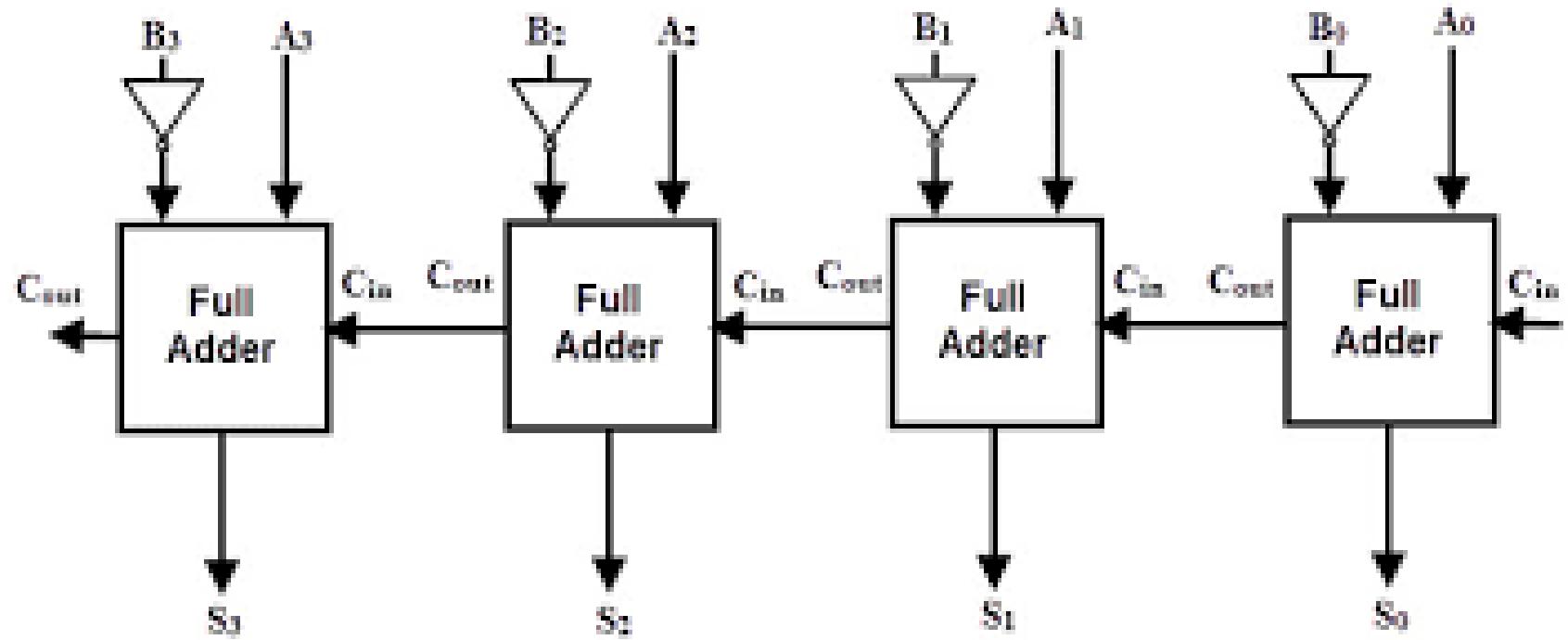
4-Bit Adder with Carry Lookahead



$$\begin{aligned}P_i &= A_i \oplus B_i \\G_i &= A_i B_i \\S_i &= P_i \oplus C_i\end{aligned}$$

All output carries are generated concurrently by the carry lookahead generator after a delay through two levels of gates

Parallel Subtractor



4-Bit Parallel Adder-Subtractor

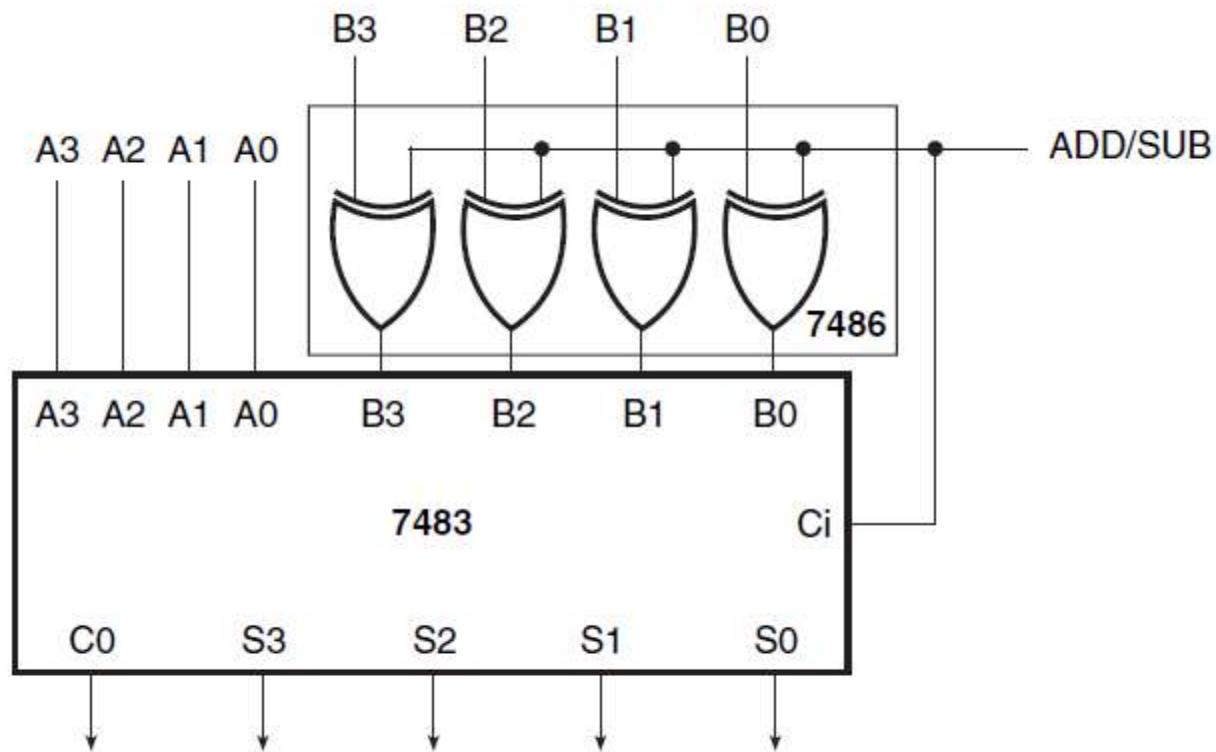
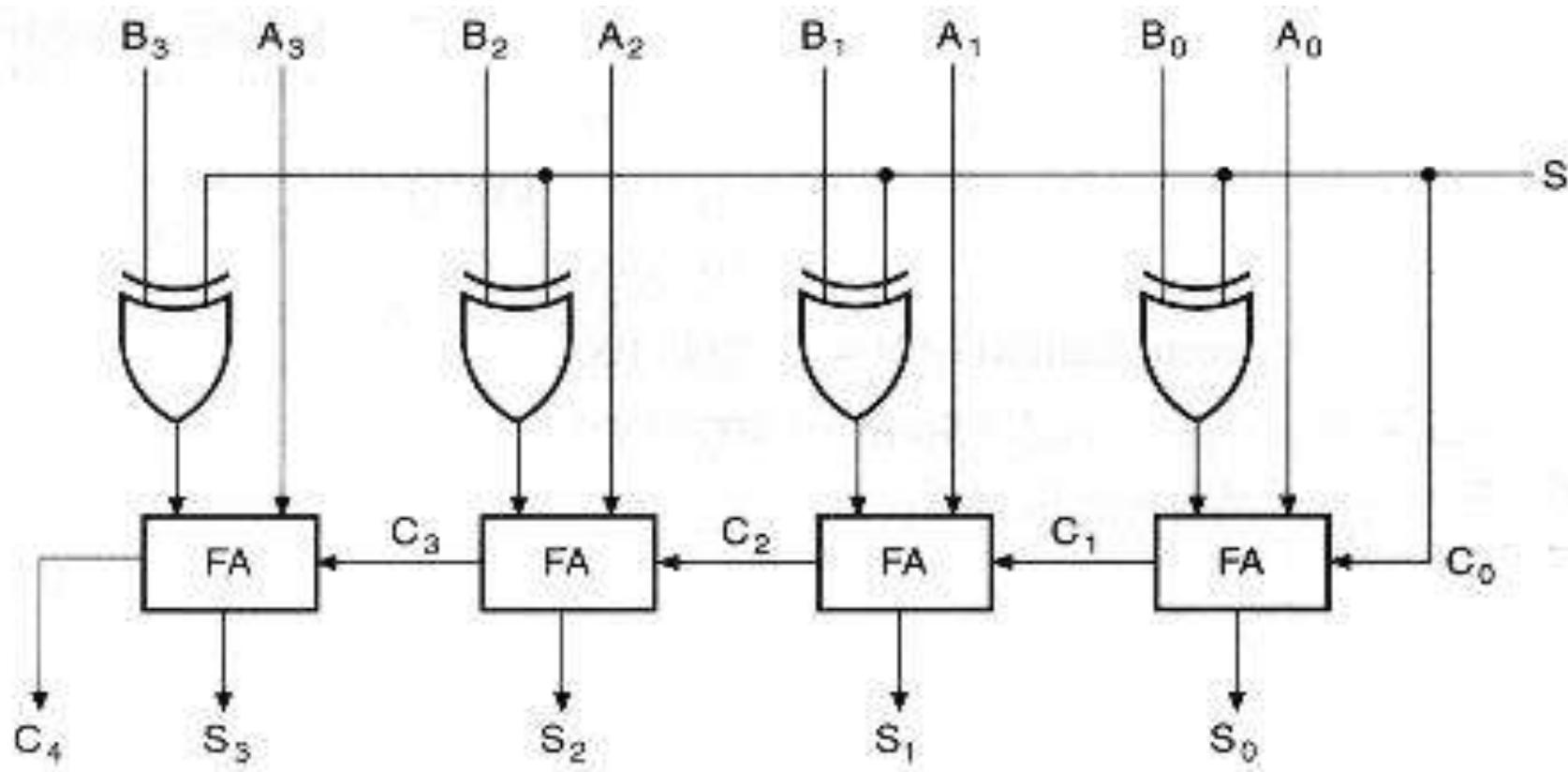


Figure 16.7 Adder–subtractor circuit (example 16.5).

4-Bit Parallel Adder-Subtractor



Binary Subtractor

Subtracting $A - B$

$$= A + (\text{1's complement of } B) + 1$$

$$= A + ((r^n - 1) - B) - r^n + 1$$

$$= A + (\text{2's complement of } B)$$

$$= A + (r^n - B) - r^n$$

V: to detect an overflow
for signed numbers

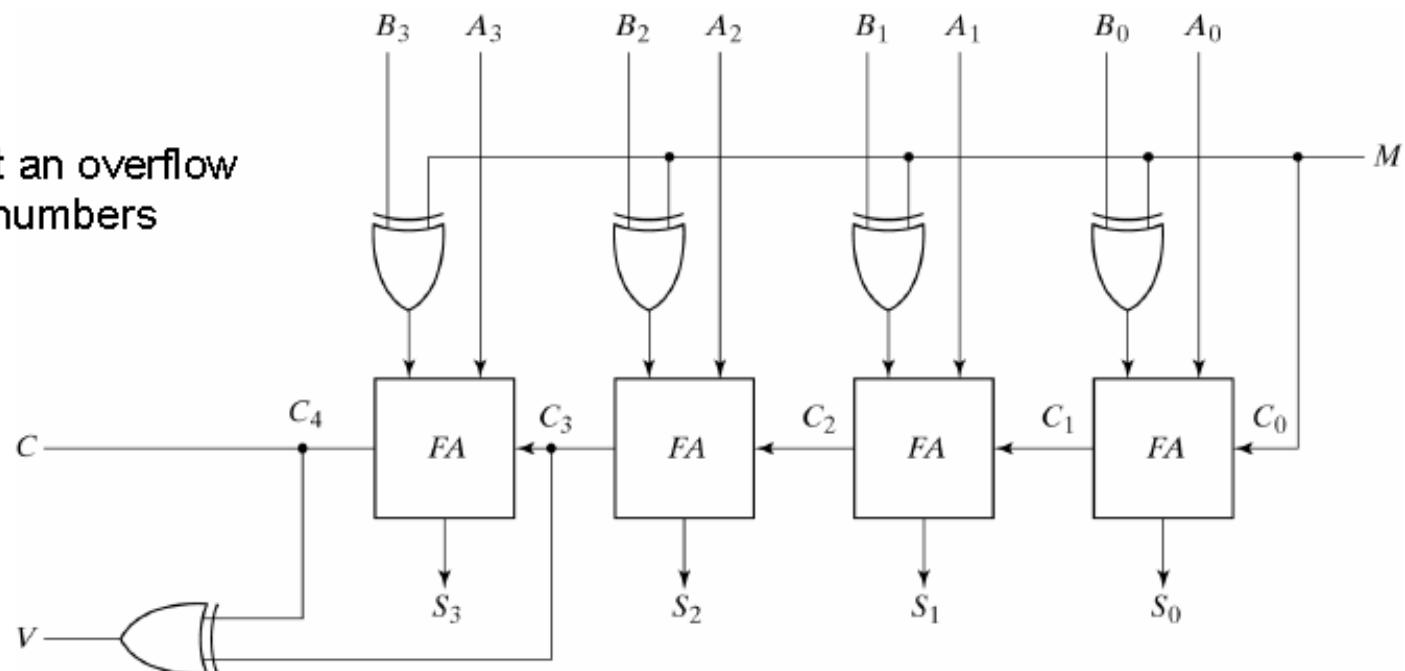


Fig. 4-13 4-Bit Adder Subtractor

4-Bit Adder Subtractor

$M=0$, the circuit is an adder ($B \oplus 0 = B$)

$M=1$, the circuit is a subtractor ($B \oplus 1 = B'$, $C_0=1$)

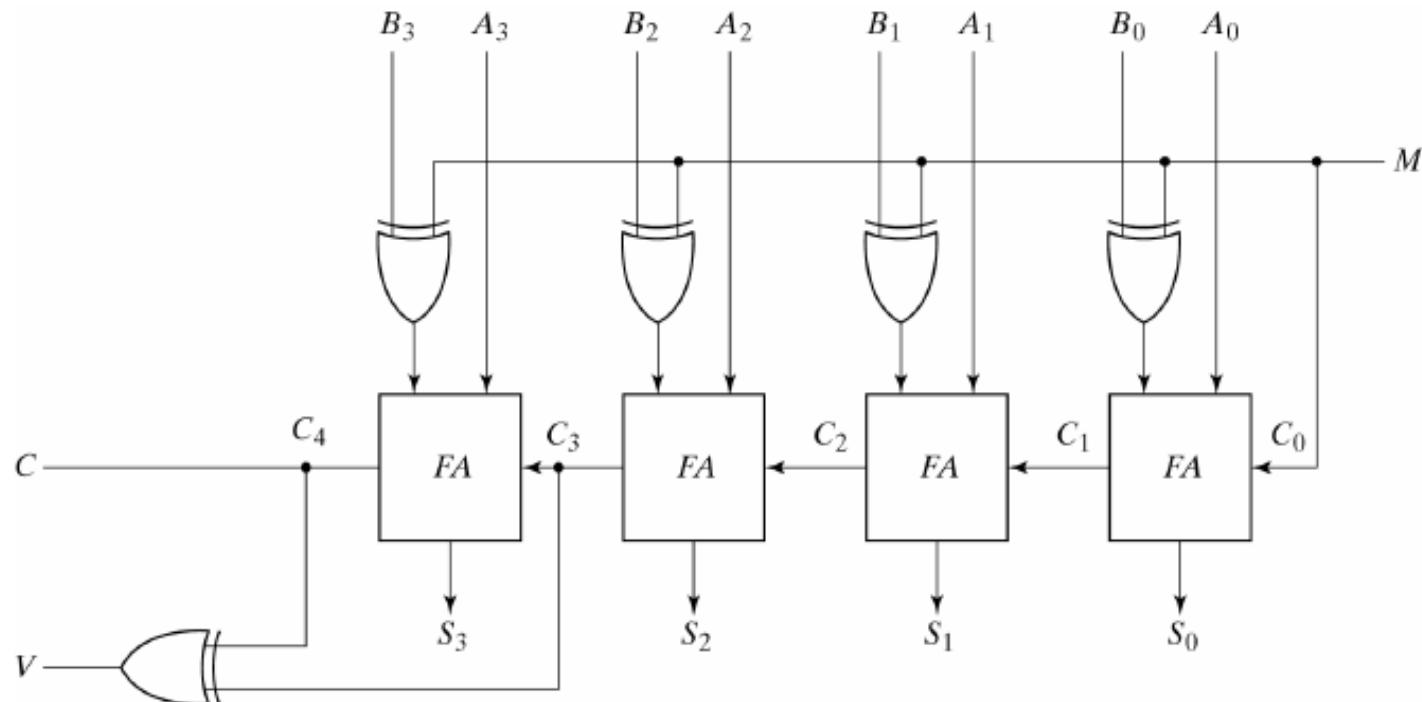


Fig. 4-13 4-Bit Adder Subtractor

Binary numbers in the signed-complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers

Overflow

When two numbers of n digits each are added and the sum occupies n+1 digits, an overflow occurs

- Addition of two unsigned numbers: detected from the end carry of the most significant position
- Addition of two signed numbers: May occur if the two numbers are both positive or both negative

•Signed numbers

- the leftmost bit always represents the sign and negative numbers are in 2's complement form
- Addition: the sign bit is treated as part of the number and the end carry does not indicate an overflow

Overflow for Signed Numbers

Examples: two signed binary numbers, +70 and +80, stored in two 8-bit register (+127 to -128)

carries:	0	1	carries:	1	0
+70	0	1000110	-70	1	0111010
+80	0	1010000	-80	1	0110000
—	—	—	—	—	—
+150	1	0010110	-150	0	1101010

Detecting overflow condition by observing

- the carry into the sign bit position
- the carry out of the sign bit position

An overflow has occurred if two carries are not equal
(V bit)

Overflow Detection

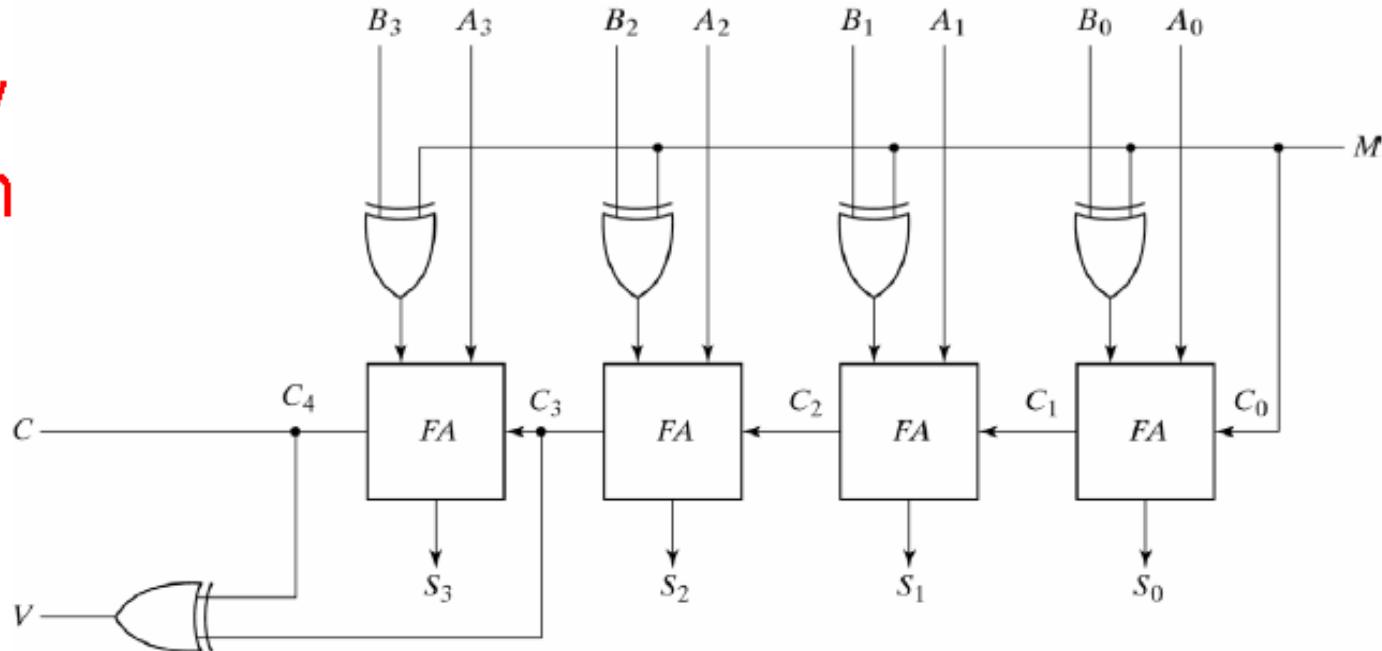


Fig. 4-13 4-Bit Adder Subtractor

- **Overflow for unsigned numbers**

C bit detects a carry after addition or a borrow after subtraction

- **Overflow for signed numbers**

$V=0$ after an addition or subtraction: indicate no overflow

$V=1$: the result of the operation contains $n+1$ bits

- An overflow has occurred
- The $(n+1)$ th bit is the actual sign and has been shifted out of position

2-12 BCD Adder

BCD Adder

Table 4-5
Derivation of BCD Adder

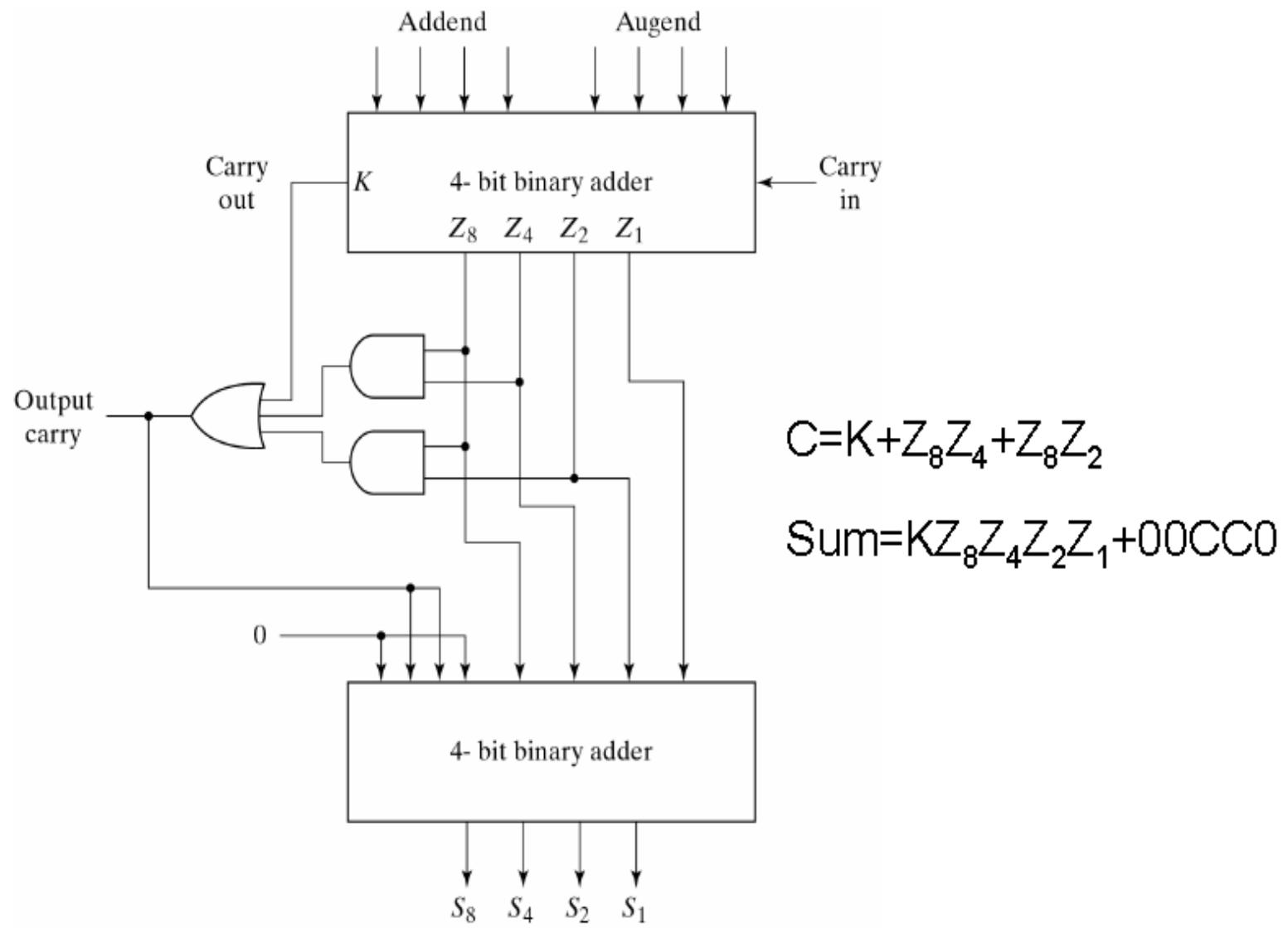
K	Binary Sum					C	BCD Sum				Decimal
	Z ₈	Z ₄	Z ₂	Z ₁			S ₈	S ₄	S ₂	S ₁	
0	0	0	0	0		0	0	0	0	0	0
0	0	0	0	1		0	0	0	0	1	1
0	0	0	1	0		0	0	0	1	0	2
0	0	0	1	1		0	0	0	1	1	3
0	0	1	0	0	=	0	0	1	0	0	4
0	0	1	0	1	+ 0	0	0	1	0	1	5
0	0	1	1	0	C=0	0	0	1	1	0	6
0	0	1	1	1		0	0	1	1	1	7
0	1	0	0	0		0	1	0	0	0	8
0	1	0	0	1		0	1	0	0	1	9
<hr/>											
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	+0110	1	0	0	1	1	13
0	1	1	1	0	C=1	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

Convert
5 bits into
2 BCD
digits

$$C = K + Z_8 Z_4 + Z_8 Z_2$$

$$\text{Sum} = KZ_8Z_4Z_2Z_1 + 00CC0$$

Adder Block diagram of a BCD



2-13 Binary Multiplier

Multiplication of binary numbers is performed in the same way as in decimal numbers

- partial product: the multiplicand is multiplied by each bit of the multiplier starting from the least significant bit

$$\begin{array}{r} & 2 & 8 \\ * & 1 & 3 \\ \hline & 6 + 2 & 4 \\ 2 & 8 \\ \hline 3 & 6 & 4 \end{array}$$

$$\begin{array}{r} & 1 & 1 & 0 \\ * & 1 & 1 & 0 & 1 \\ \hline & 1 & 1 & 0 \\ & 0 & 0 & 0 \\ & 1 & 1 & 0 \\ & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 & 1 & 1 & 0 \end{array}$$

Multiplication of two bits = A * B (AND)

$$0 * 0 = 0$$

$$0 * 1 = 0$$

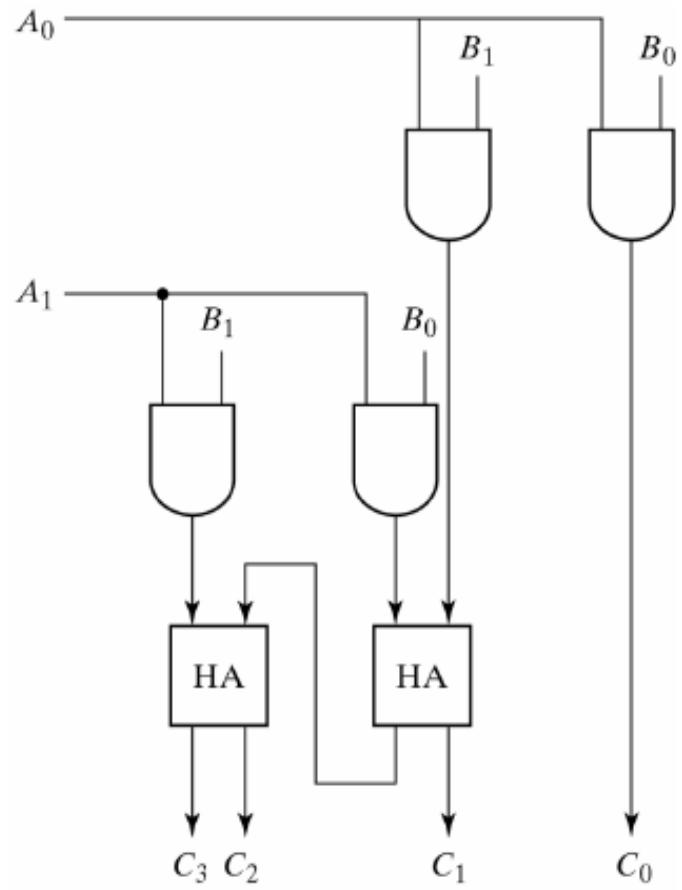
$$1 * 0 = 0$$

$$1 * 1 = 1$$

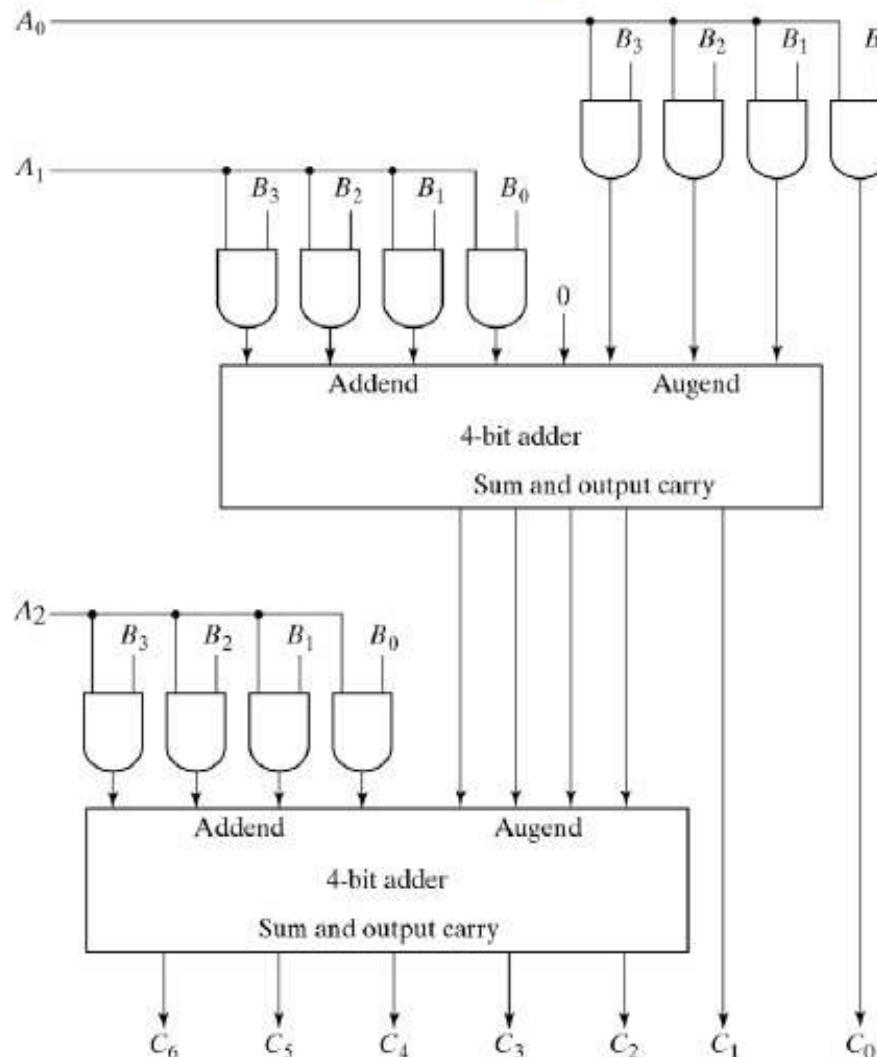
2-bit by 2-bit binary multiplier

A1 A0 B1 B0	C3 C2 C1 C0
0 0 0 0	0 0 0 0
0 0 0 1	0 0 0 0
0 0 1 0	0 0 0 0
0 0 1 1	0 0 0 0
0 1 0 0	0 0 0 0
0 1 0 1	0 0 0 1
0 1 1 0	0 0 1 0
0 1 1 1	0 0 1 1
1 0 0 0	0 0 0 0
1 0 0 1	0 0 1 0
1 0 1 0	0 1 0 0
1 0 1 1	0 1 1 0
1 1 0 0	0 0 0 0
1 1 0 1	0 0 1 1
1 1 1 0	0 1 1 0
1 1 1 1	1 0 0 1

$$\begin{array}{r} B_1 \quad B_0 \\ A_1 \quad A_0 \\ \hline A_0B_1 \quad A_0B_0 \\ A_1B_1 \quad A_1B_0 \\ \hline C_3 \quad C_2 \quad C_1 \quad C_0 \end{array}$$



4-bit by 3-bit binary multiplier



$$\begin{array}{r}
 & B_3 \ B_2 \ B_1 \ B_0 \\
 & 1 \ 1 \ 0 \ 1 \\
 * & 1 \ 1 \ 0 \\
 \hline
 & 0 \ 0 \ 0 \ 0 \\
 + & 1 \ 1 \ 0 \ 1 \\
 \hline
 & 1 \ 1 \ 0 \ 1 \ 0 \\
 + & 1 \ 1 \ 0 \ 1 \\
 \hline
 & 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0
 \end{array}
 \quad
 \begin{array}{r}
 & 1 \ 3 \\
 & 6 \\
 \hline
 & 1 \ 8 \\
 A_0 & 6 \\
 \hline
 & 7 \ 8 \\
 A_1 & \\
 \hline
 & A_2
 \end{array}$$

Magnitude Comparator

- Equal ($A = B$)

- $A_3=B_3$ and $A_2=B_2$ and $A_1=B_1$ and $A_0=B_0$

$$X_i = A_i B_i + A_i' B_i' \text{ for } i = 0, 1, 2, 3$$

$X_i = 1$ means
 A_i and B_i are equal !!

- $(A=B) = X_3 X_2 X_1 X_0$

- Greater ($A > B$) or Less ($A < B$)

- Comparison start from the MSB
 - If the two digits are equal, compare the next lower digits
 - Continues until a pair of unequal digits is reached
 - A is 1 and B is 0 $\Rightarrow A > B$
 - A is 0 and B is 1 $\Rightarrow A < B$

$$(A > B) = A_3 B_3' + X_3 A_2 B_2' + X_3 X_2 A_1 B_1' + X_3 X_2 X_1 A_0 B_0'$$

$$(A < B) = A_3' B_3 + X_3 A_2' B_2 + X_3 X_2 A_1' B_1 + X_3 X_2 X_1 A_0' B_0$$

4-47

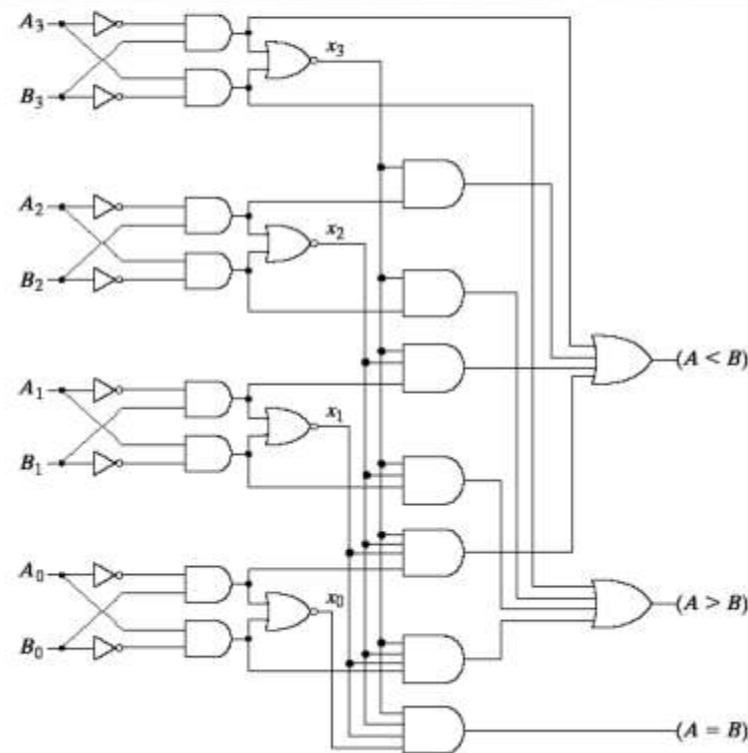
1-bit Comparator

Truth Table

A	B	A=B	A>B	A<B
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

$$\text{Output} = \overline{A}\overline{B} + AB$$

4-Bit Magnitude Comparator



4-48

Module:4

PARALLEL ADDER SUBTRACTOR

Binary Parallel Adder

Binary adder

- Binary adder that produces the arithmetic sum of binary numbers can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain
- Note that the input carry C_0 in the least significant position must be 0.

Binary Adder

- For example to add $A = 1011$ and $B = 0011$

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}

Binary Adder

Example: $10+6=16$

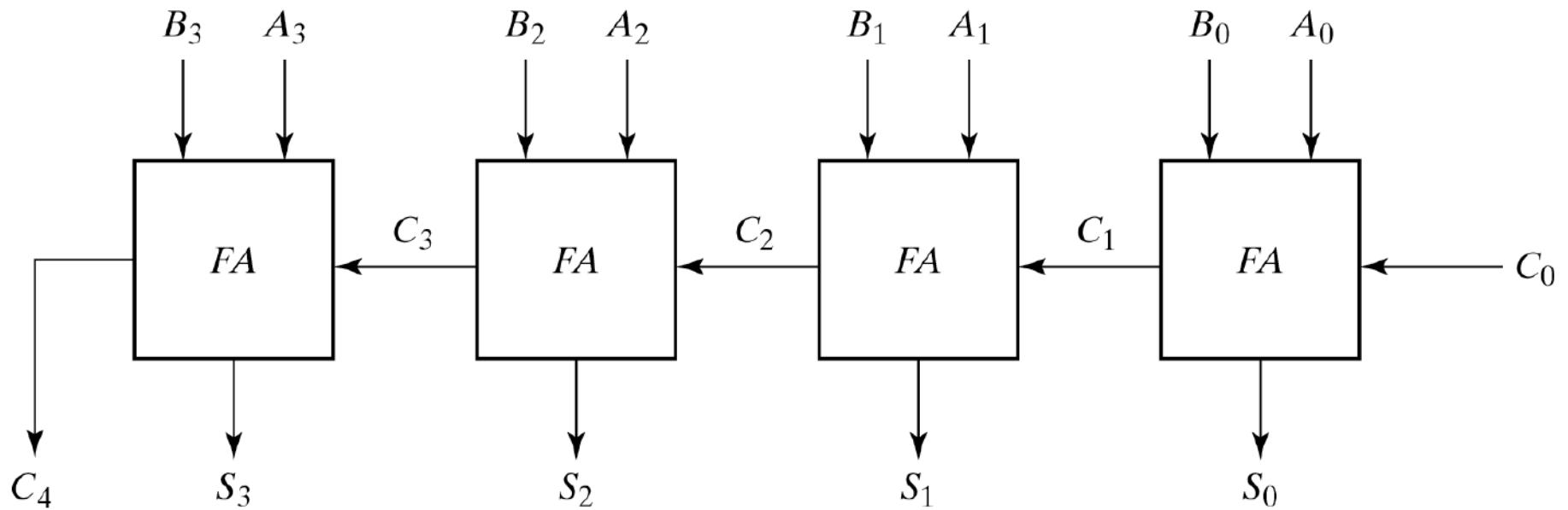


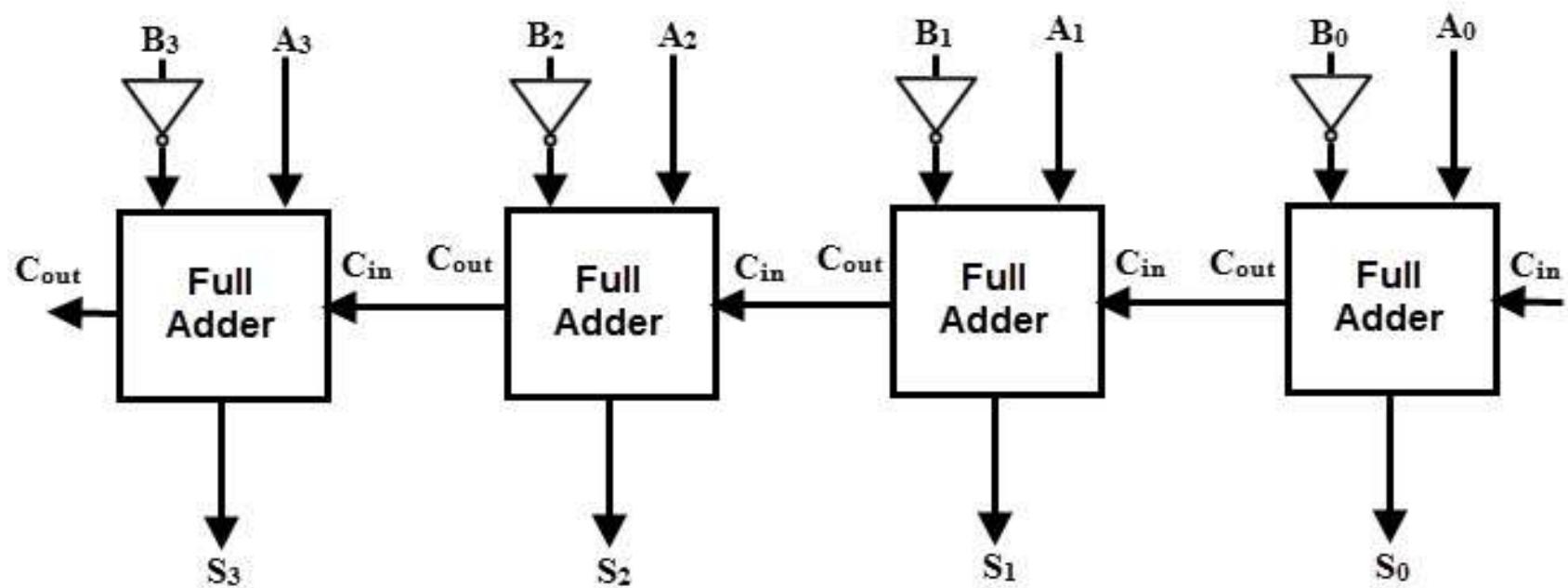
Fig. 4-9 4-Bit Adder

Binary Subtractor

- The subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A because $A - B = A + (-B) = A + ((B' + 1))$
- It means if we use the inverters to make 1's complement of B (connecting each B_i to an inverter) and then add 1 to the least significant bit (by setting carry C_0 to 1) of binary adder, then we can make a binary subtractor.

4 bit 2's complement Subtractor

Example: $10-6=10+((1\text{'s of } 6)+1)=4$



Adder Subtractor

- The addition and subtraction can be combined into one circuit with one common binary adder (see next slide).
- The mode M controls the operation. When M=0 the circuit is an **adder** when M=1 the circuit is a **subtractor**. It can be done by using exclusive-OR for each Bi and M. Note that $1 \oplus x = x'1 + x1' = x'$ and $0 \oplus x = x'0 + x0' = x$

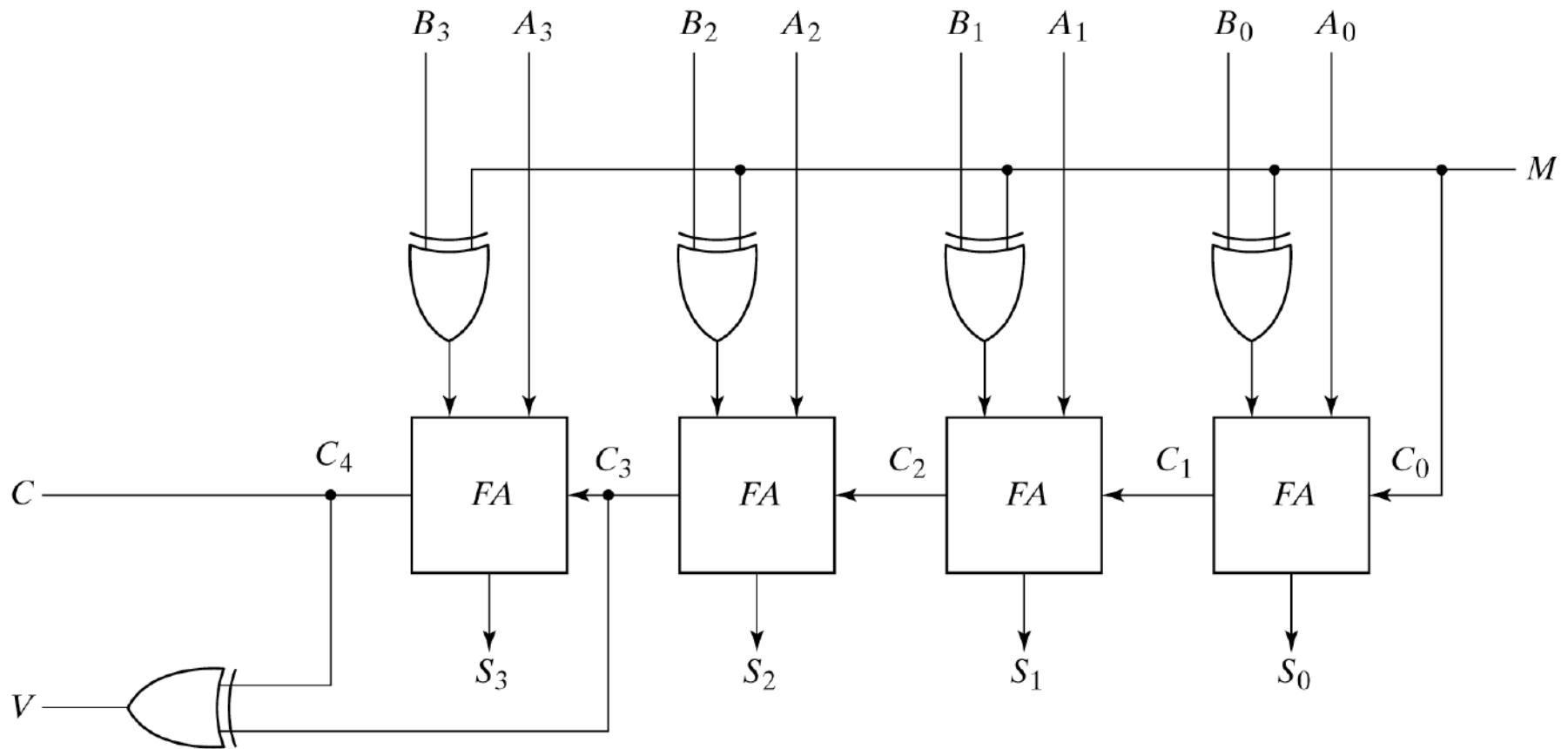


Fig. 4-13 4-Bit Adder Subtractor

Checking Overflow

- Note that in the previous slide if the numbers considered to be signed V detects overflow. V=0 means no overflow and V=1 means the result is wrong because of overflow
- Overflow can be happened when adding two numbers of the same sign (both negative or positive) and result can not be shown with the available bits. It can be detected by observing the carry into sign bit and carry out of sign bit position. If these two carries are not equal an overflow occurred. That is why these two carries are applied to exclusive-OR gate to generate V.

4-Bit Parallel Adder-Subtractor

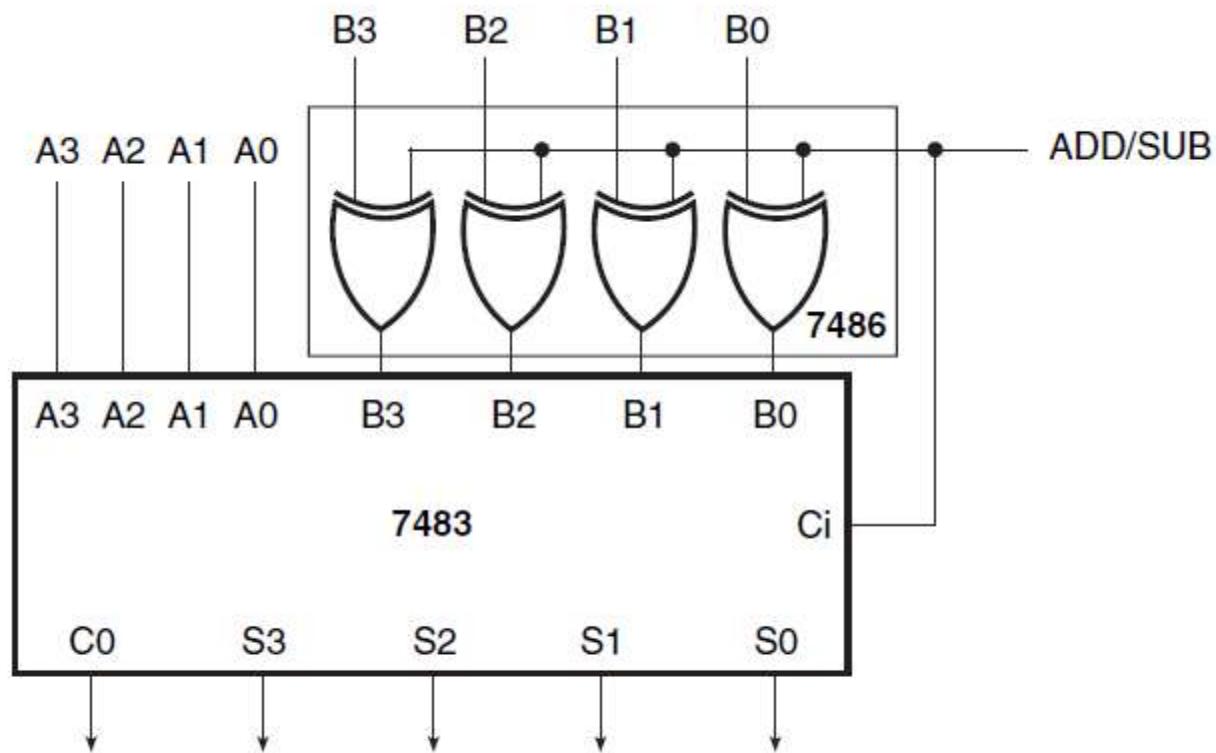


Figure 16.7 Adder–subtractor circuit (example 16.5).

4-Bit Adder Subtractor

$M=0$, the circuit is an adder ($B \oplus 0 = B$)

$M=1$, the circuit is a subtractor ($B \oplus 1 = B'$, $C_0=1$)

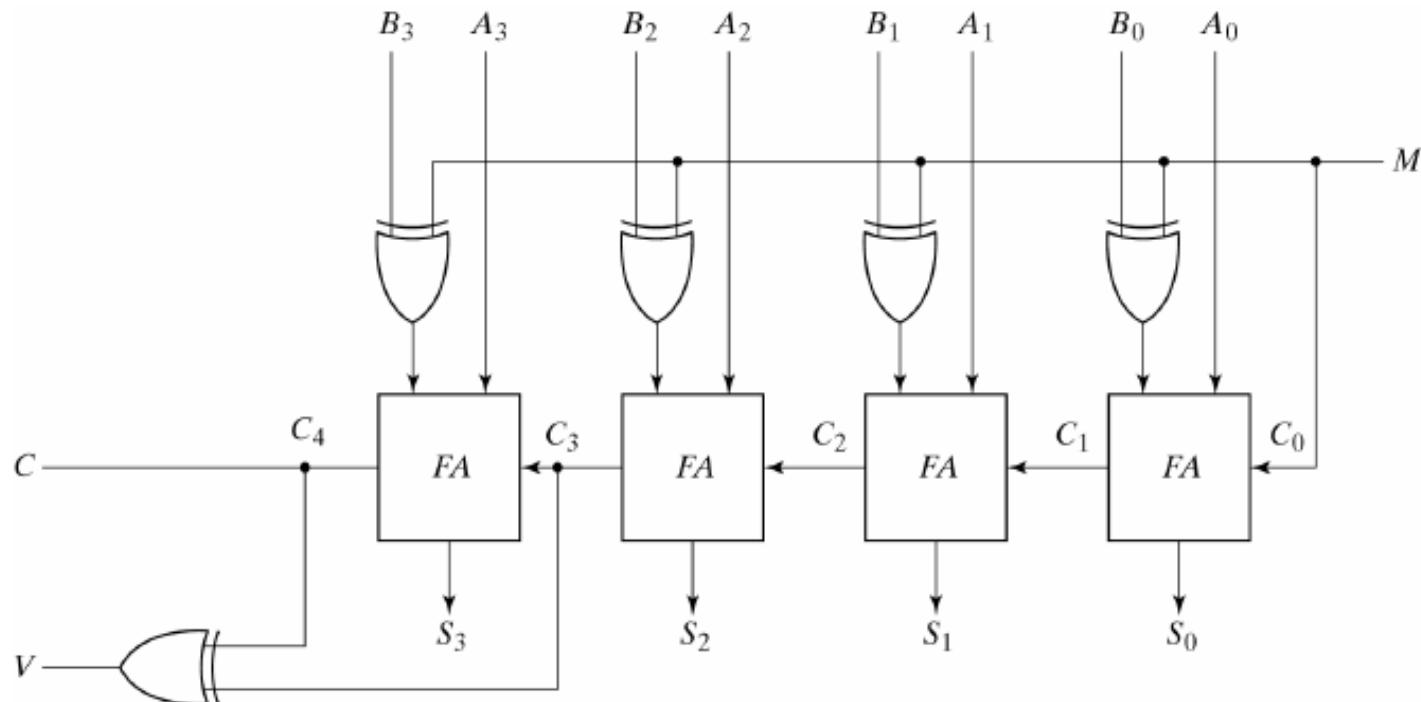


Fig. 4-13 4-Bit Adder Subtractor

Binary numbers in the signed-complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers

Overflow

When two numbers of n digits each are added and the sum occupies n+1 digits, an overflow occurs

- Addition of two unsigned numbers: detected from the end carry of the most significant position
- Addition of two signed numbers: May occur if the two numbers are both positive or both negative

•Signed numbers

- the leftmost bit always represents the sign and negative numbers are in 2's complement form
- Addition: the sign bit is treated as part of the number and the end carry does not indicate an overflow

2-12 BCD Adder

BCD Adder

Table 4-5
Derivation of BCD Adder

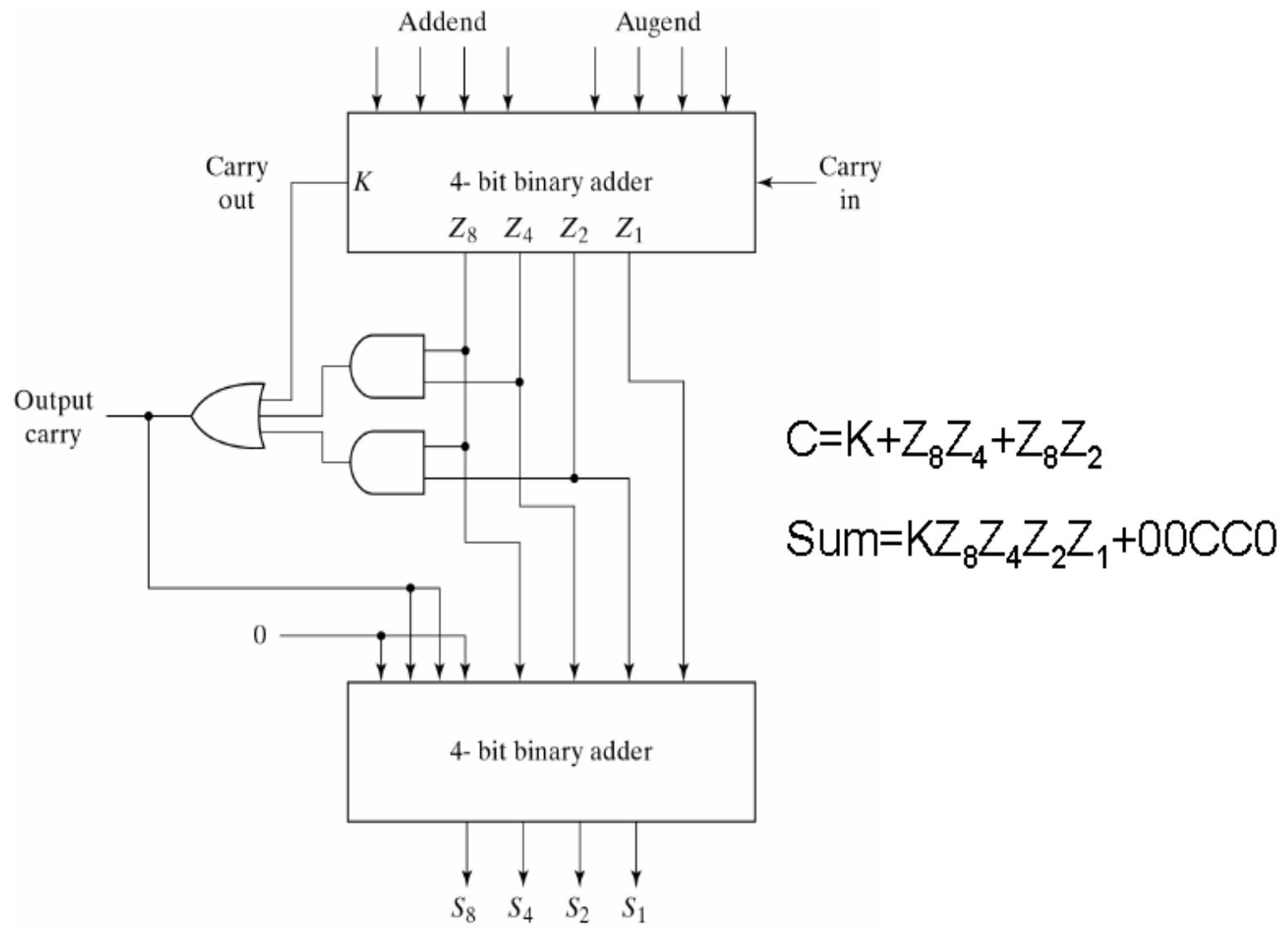
K	Binary Sum					C	BCD Sum				Decimal
	Z ₈	Z ₄	Z ₂	Z ₁			S ₈	S ₄	S ₂	S ₁	
0	0	0	0	0		0	0	0	0	0	0
0	0	0	0	1		0	0	0	0	1	1
0	0	0	1	0		0	0	0	1	0	2
0	0	0	1	1		0	0	0	1	1	3
0	0	1	0	0	=	0	0	1	0	0	4
0	0	1	0	1	+ 0	0	0	1	0	1	5
0	0	1	1	0	C=0	0	0	1	1	0	6
0	0	1	1	1		0	0	1	1	1	7
0	1	0	0	0		0	1	0	0	0	8
0	1	0	0	1		0	1	0	0	1	9
<hr/>											
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	+0110	1	0	0	1	1	13
0	1	1	1	0	C=1	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

Convert
5 bits into
2 BCD
digits

$$C = K + Z_8 Z_4 + Z_8 Z_2$$

$$\text{Sum} = KZ_8Z_4Z_2Z_1 + 00CC0$$

Adder Block diagram of a BCD



Encoding Schemes for Multipliers

Hardware multiplication is performed in the same way multiplication done by hand, first step is to partialized the products are computed then shifted appropriately and summed.

Normal multiplication Process:

The simplest multiplication operation is to directly calculate the product of two numbers by hand. This procedure can be divided into three steps:

1. Partial product generation
2. Partial product reduction
3. Addition.

Let us calculate the product of 2's complement of two numbers 1101(-3) and 5 (0101), when computing the two binary numbers product we get the result

1 1 0 1	Multiplicand
x 0 1 0 1	Multiplier

1 1 1 1 1 1 0 1	PP1
0 0 0 0 0 0 0 0	PP2
1 1 1 1 0 1	PP3
+ 0 0 0 0 0	PP4

1 1 1 1 1 0 0 1	Product

\swarrow

Discard this bit

From the above we say that 1101 is multiplicand and 0101 is multiplier. The intermediate products are partial products. The final result is product (-15). When this method is processed in hardware, the operation is to take one of the multiplier bits at a time from right to left, multiplying the multiplicand by the single bit of the multiplier and shifting the intermediate product one position to the left of the earlier intermediate products. All the bits of the partial products in each column are added to obtain two bits: sum and carry. Finally, the sum and carry bits in each column have to be summed. The two rows before the product are called sum and carry bits.

1 1 0 1	Multiplicand
x 0 1 0 1	Multiplier

$$\begin{array}{r}
 \begin{array}{ccccccccc}
 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 1 & 1 & 1 & 1 & 0 & 1 \\
 + & 0 & 0 & 0 & 0 & 0
 \end{array}
 \hline
 \begin{array}{ccccccccc}
 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0
 \end{array}
 \hline
 \begin{array}{ccccccccc}
 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1
 \end{array}
 \end{array}$$

PP1
PP2
PP3
PP4

Sum bit
Carry bit

Product

Advantage:

In this method the partial product circuit is simple and easy to implement. Therefore, it is suitable for the implementation of small multipliers.

Disadvantage:

This method is not able to efficiently handle the sign extension and it generates a number of partial products as many as the number of bits of the multiplier, which results in many adders needed so that the area and power consumption increase. This method is not applicable for large multipliers.

Booths algorithm:-

This algorithm will be slow if there are many partial products because the output must wait until each sum is calculated and performed. As speed is main important aspect while calculating the multiplication. By using the Booths algorithm, we can maximize the speed. In Booths algorithm, speed can be maximized by reducing the partial products to half. The adder circuits in Booth algorithm provide the advantage in maximizing the speed.

ALU implements Booth algorithm to multiply binary number. ALU cannot simply multiply binary number as it can do only addition, subtraction and shifting.

Working Principle:-

Accumulator	Multiplier Q	Q-1	Multiplicand	Action
0000	001 <u>0</u>	<u>0</u>	0110 2's complement of 0110	Shifting
Step1 0000	000 <u>1</u>	<u>0</u>	1010	

0000	Step 2	000 <u>1</u>	<u>0</u>	0110 2's complement of 0110 1010	Subtraction shifting
1101	Step 4	000 <u>0</u>	<u>1</u>	0110	Addition
+0110					Shifting
0011					Shifting
0001	Step 5	100 <u>0</u>	<u>0</u>		
0000	Step 6	1100	0		

The operation of Booth encoding consists of two major steps: the first one is to take one bit of the multiplier, and then to decide whether to add the multiplicand according to the current and previous bits of the multiplier.

Initially accumulator start with 0000 and we will perform shift that is arithmetic shift then we get first bit of A and the copy of first bit A and for the remaining positions we get from the second position of A. The left bit in A is shifted to **Q** and bits of **Q** is placed after it. The bit in **Q** is shifted to **Q₋₁**. Initially **Q₋₁** will be 0. Then we will compare LSB Multiplier, based on this comparision we will perform the addition, subtraction and shifting.

Q₀	Q₋₁	Action on A and multiplicand
0	0	Shifting
0	1	Adding
1	0	Subtracting
1	1	Shifting

Based on this comparision we will perform the action on Accumulator and multiplicand.

In the above table we multiplied 0010 and 0110 as an example and the result will be combination of both Accumulator and multiplier i.e 0000 1100(12).

Modified Booth Algorithm:

The encoding method is widely used to generate the partial products for the implementation of large multipliers. It adopts parallel encoding scheme. Modified Booth Algorithm can be realized by using circuits consists of Booth Encoder and multiplier array of partial product generator (Multiplexer) and adders.

Booth Encoder:

It implements Booth Algorithm encoding.

Basic Principle:-

The basic principle in modified Booth Algorithm is, consider X and Y are two fixed-point two's complement numbers, where X is multiplier and Y is multiplicand both having same n bits.

X can be represented as

$$\begin{aligned}
 X &= -X_{n-1}2^{n-1} + \sum_{i=0}^{n-2} X_i 2^i \\
 &= \sum_{i=0}^{n/2-1} (-X_{2i-1} + X_{2i} + X_{2i-1}) \cdot 2^{2i} \\
 &= \sum_{i=0}^{n/2-1} (-2X_{2i+1} + X_{2i} + X_{2i-1}) \cdot 4^i
 \end{aligned}$$

By multiplying the X with Y, results

$$XY = \sum_{i=0}^{n/2-1} (-2X_{2i+1} + X_{2i} + X_{2i-1}) \cdot 4^i \cdot Y$$

From the above equation we portioned the bits of multipliers into substrings of 3 adjacent bits and each substring consists of $(X_{2i+1}, X_{2i}, X_{2i-1})$, each corresponds to the value $\{-2, -1, 0, +1, +2\}$

The grouping of bits of multiplies can be done as



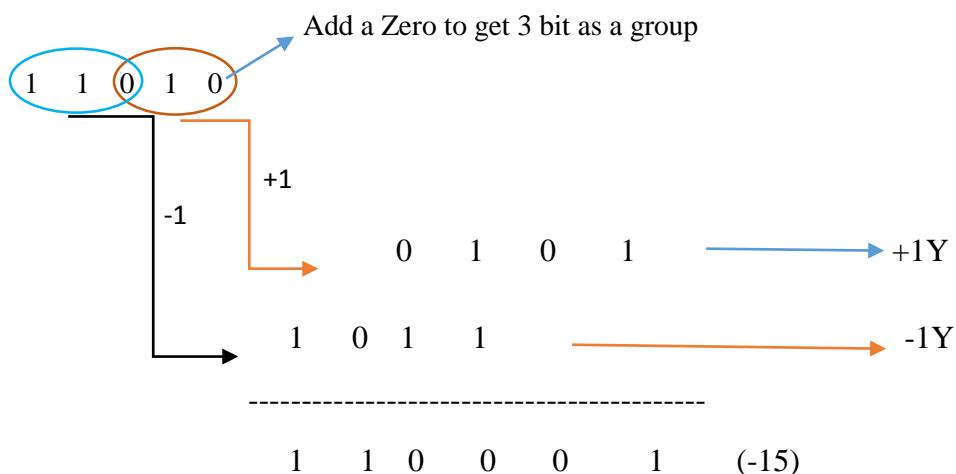
X_{2i+1}	X_{2i}	X_{2i-1}	Possible values
0	0	0	0
0	0	1	+1
0	1	0	+1
0	1	1	+2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Each three adjacent bits of the multiplier can generate a single encoding digit having the possible values

$\{-2, -1, 0, +1, +2\}$. For the $n \times n$ multiplication, the number of bits for the multiplier X is n , using the modified Booth encoding $n/2$ partial products are produced.

Multiplication:

The partial product should be shifted two positions to the left of the partial product due to the is multiplied by



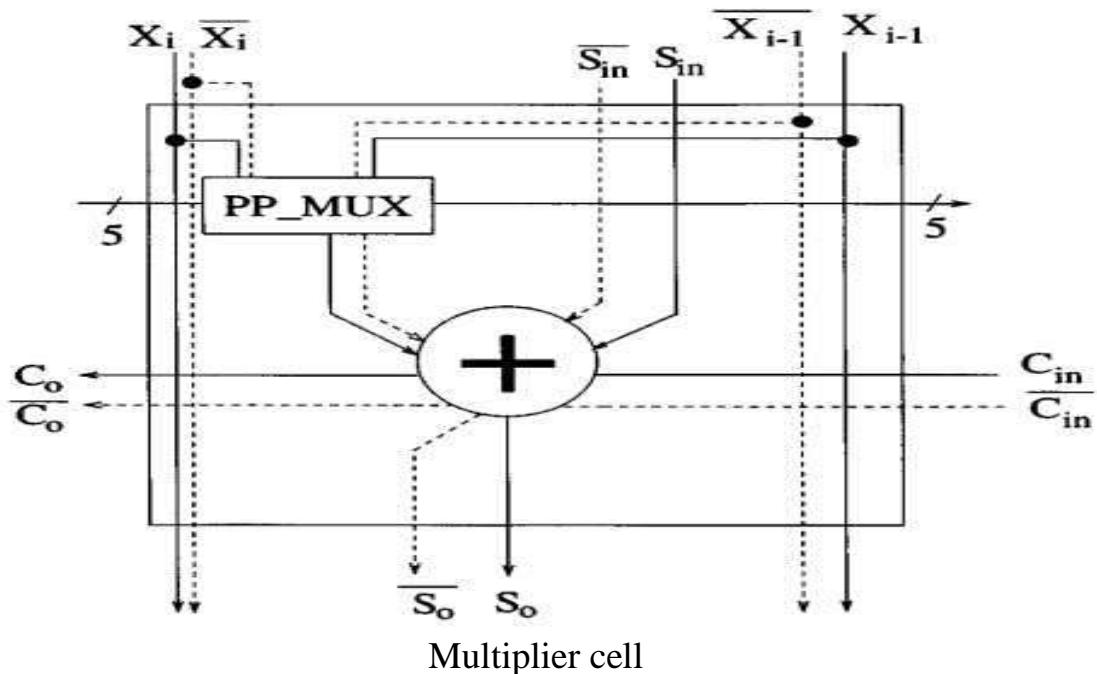
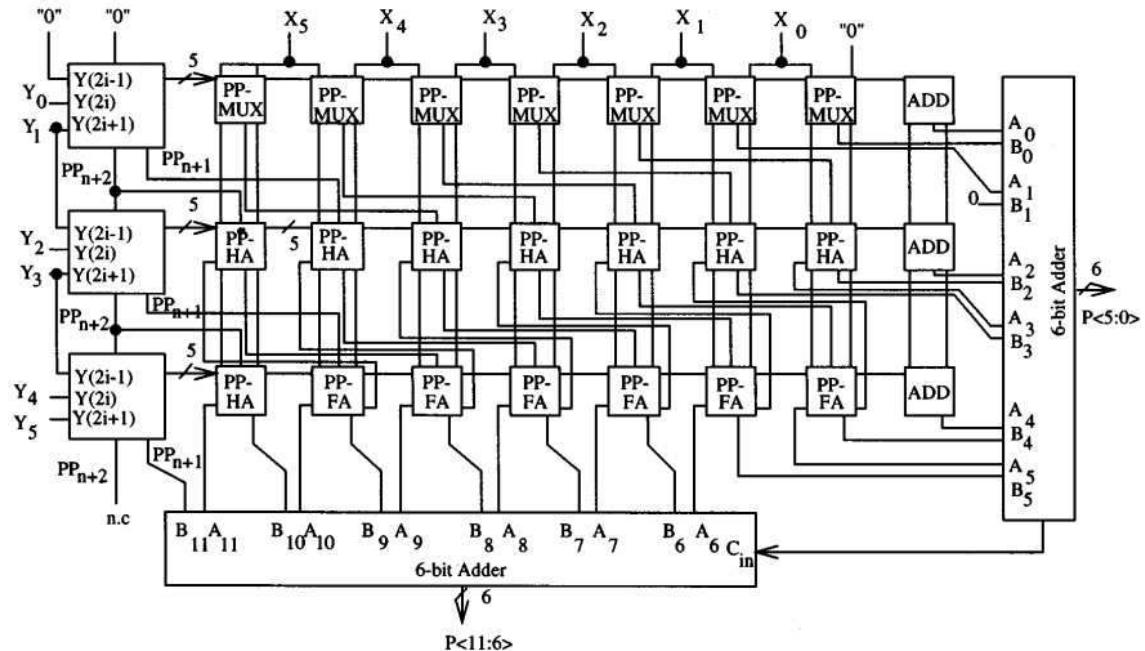
The operation is summarized as

Possible values	Operation on Y
0	$0*Y: Y \Rightarrow 0 \Rightarrow$ Product is zero
+1	$+1*Y: Y \Rightarrow$ Product (Y is the product)
+2	$+2*Y: \text{One Shift Y to the left} \Rightarrow$ Product
-1	$-1*Y: \text{Invert Y and add 1 to the LSB of Y} \Rightarrow$ Product

-2

-2*Y: One Shift to the left for Y, then inverted Y & added 1 to the LSB

Architecture: Booth Multiplier



Modified Booth Algorithm can be realized by using circuits consists of Booth Encoder and multiplier array of partial product generator (Multiplexer) and adders.

Booth Encoder:

On the left-hand side are the Booth encoders, one for each partial product. They each have three bits of as input (with “0” to the right of the LSB).They are also responsible for decoding and propagating the sign extension logic to the next encoder. The array cells then generate the appropriate bit and add it to the accumulated sum with their internal adders. In two’s complement format inverting a number consists of flipping all bits and adding one. The “ADD” cell generates the 1 if required.

Multiplier cell:

The cell consists of two components, a multiplexer to generate the partial product bit (PP-MUX) and a full adder (FA) or half adder (HA) to add this bit with the previous sum.

The multiplier cell represents one bit in a partial product and is responsible for:

- 1) Generating a bit of the correct partial product in response to the signals from the Booth encoder;
- 2) Adding this bit to the cumulative sum propagated from the row above.

Adders:

In Booth Algorithm we use Half Adder and Full Adders along with the Multiplexer. The FA is the most critical circuit in the multiplier as it ultimately determines the speed and power dissipation of the array.

The Boolean Expression for Half Adder is

$$S_0 = A \text{ XOR } B$$

$$C_0 = AB$$

The Boolean Expression for Full Adder is

$$S_0 = A \text{ XOR } B \text{ XOR } C_{in}$$

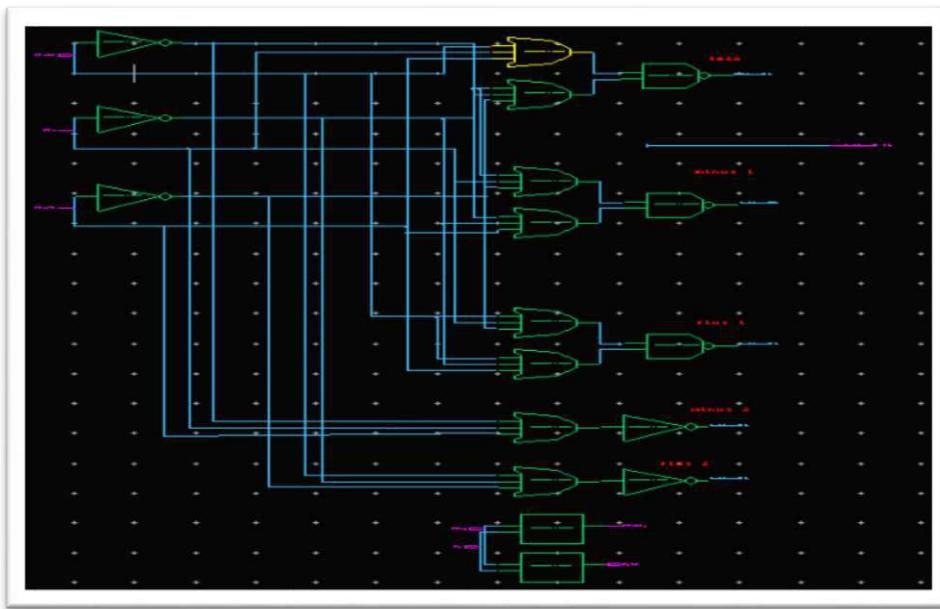
$$C_0 = A (\sim B) C_{in} + \tilde{A} B C_{in} + AB.$$

COMPARISON OF GDI BOOTH MULTIPLIER AND CMOS BOOTH MULTIPLIER

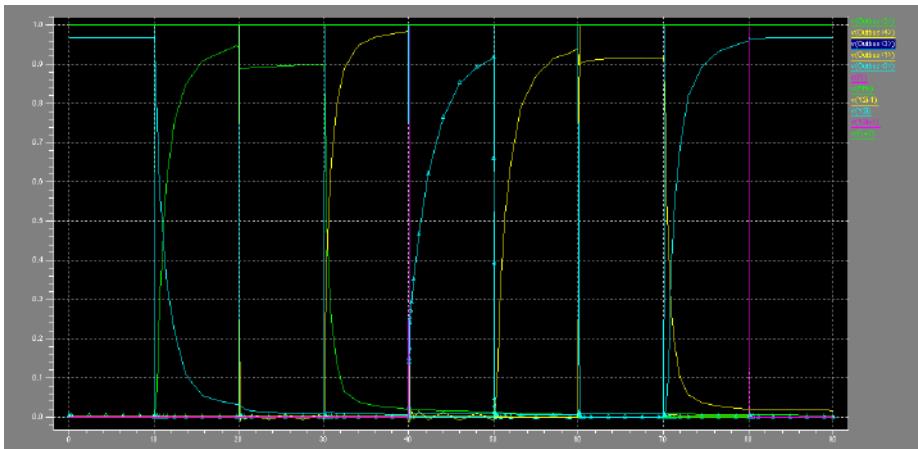
In this project we compare GDI Booth Multiplier and CMOS Booth Multiplier in terms of power dissipation. The power dissipation in a multiplier is a very important issue as it reflects the total power dissipated by the circuit and hence affects the performance for the device. We will conclude that GDI Booth Multiplier is better than CMOS Booth Multiplier. We calculate the power dissipation of basic blocks (Booth Encoder, Adder) of both GDI Booth Multiplier and CMOS Booth Multiplier. We calculate the power dissipation of each block at channel length of 45nm, by varying the width of the channel we calculate the power of each block, in GDI Booth Multiplier at channel width of PMOS = 49nm and NMOS = 45nm and in CMOS Booth Multiplier, the channel width of PMOS is 900nm and NMOS is 450nm.

Calculation of power dissipation of GDI based Booth Encoder:

The booth Encoder Circuit is shown in the Figure. The Y_{2i-1} , Y_{2i} , Y_{2i+1} are the three adjacent bits of the multiplier and its output generates five possible values which are encoded.



The output of the Booth Encoder circuit is shown in figure.



Power Dissipation of the input Source is

Source	Power in GDI
VDD	5.8902e-006
Y _{2i+1}	2.5952e-008
Y _{2i}	6.9726e-008
Y _{2i-1}	1.3979e-007
P _{P n , j}	9.6417e-013
F _j	1.6801e-006
TOTAL POWER	7.805768e-6

Time Delay of the outbus w.r.t input Voltage Sources is:

	Delay in GDI
Outbus<0> w.r.t Y _{2i-1}	7.6990e-009
Outbus<1> w.r.t Y _{2i-1}	2.5674e-008
Outbus<2> w.r.t Y _{2i-1}	5.9361e-009
Outbus<3> w.r.t Y _{2i-1}	1.8549e-010
Outbus<4> w.r.t Y _{2i-1}	4.3784e-011
Outbus<0> w.r.t Y _{2i}	2.3010e-009
Outbus<1> w.r.t Y _{2i}	4.3835e-010
Outbus<2> w.r.t Y _{2i}	4.0639e-009
Outbus<3> w.r.t Y _{2i}	1.8549e-010
Outbus<4> w.r.t Y _{2i}	3.8965e-010
Outbus<0> w.r.t Y _{2i+1}	2.7699e-008
Outbus<1> w.r.t Y _{2i+1}	1.4326e-008
Outbus<2> w.r.t Y _{2i+1}	2.5936e-008
Outbus<3> w.r.t Y _{2i+1}	1.8549e-010
Outbus<4> w.r.t Y _{2i+1}	9.6104e-009

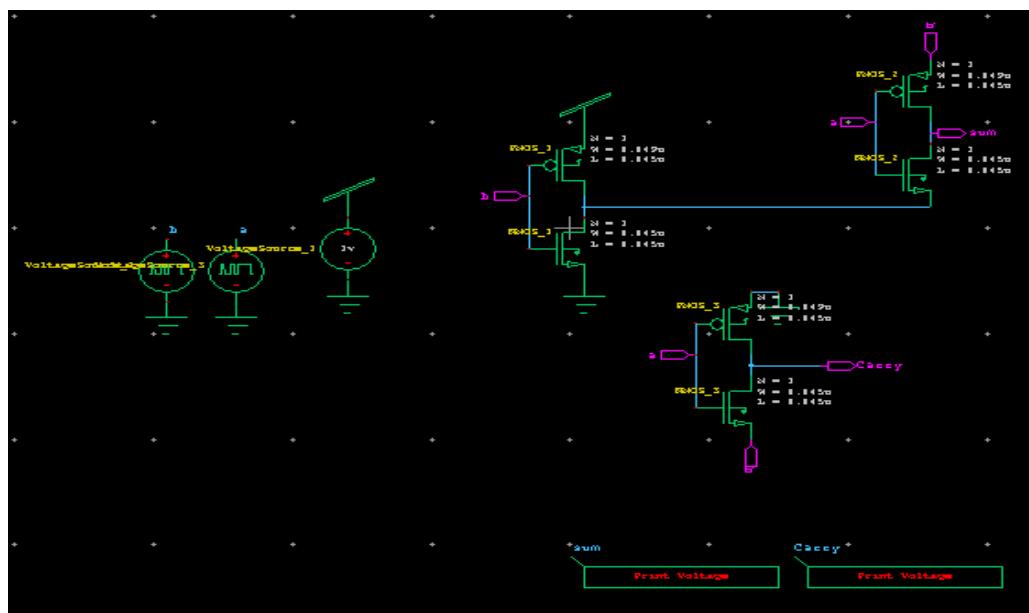
Calculation of power of Half Adder:-

We calculate the power of Half Adder for different input patterns taken in such

	Pattern	Power
Source 1(1v)	0011	6.0470e-008
Source 2(1v)	0101	1.6233e-009
TOTAL POWER		1.596313e-7

a way that satisfies the all the conditions or having all the possible cases. The power of VDD Voltage Source is 9.7538e-008

The circuit of the Half Adder in the GDI Multiplier is



Calculation of Power Dissipation of Full Adder of GDI Multiplier:

We calculate the power of Half Adder for different input patterns taken in such a way that satisfies the all the conditions or having all the possible cases. The power of VDD Voltage Source is -2.1139e-008 watts.

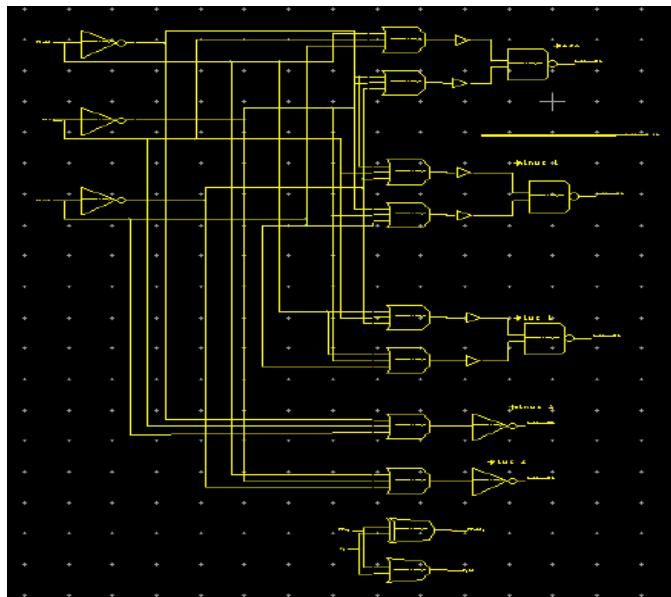
	Pattern	Power
Source1(1v)	00001111	4.1456e-006
Source2(1v)	00110011	9.5822e-007
Source3(1v)	01010101	1.1155e-007
TOTAL POWER		5.194231e-6

Calculation of Power Dissipation in Booth Encoder of CMOS Multiplier:

The booth Encoder Circuit is shown in the Figure. The Y_{2i-1} , Y_{2i} , Y_{2i+1} are the three adjacent bits of the multiplier and its output generates five possible values which are encoded. The output of the Booth Encoder is Outbus $<0, 1, 2, 3, 4>$.The power dissipated from the input Voltage Sources is

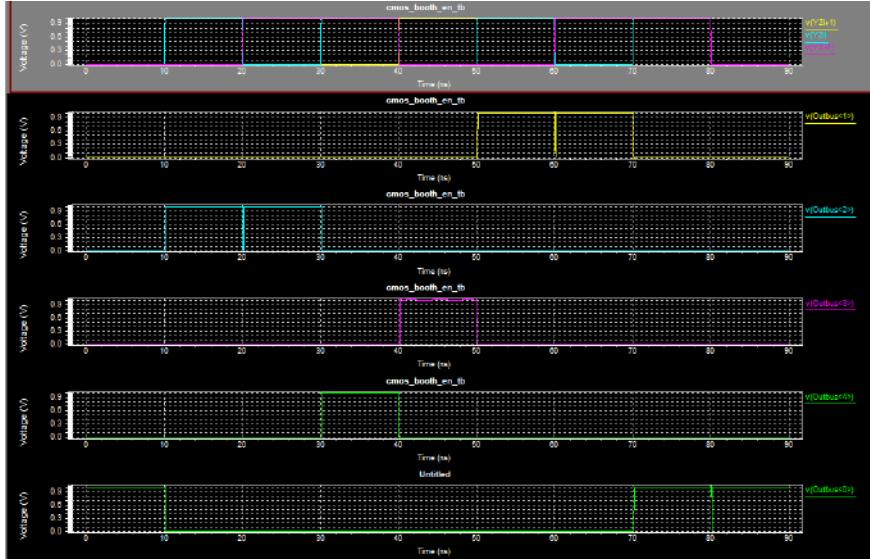
Source	Power in CMOS
VDD	8.2437e-006
Y_{2i+1}	4.5346e-008
Y_{2i}	6.6163e-008
Y_{2i-1}	1.1919e-007
$PP_{n, j}$	2.3175e-008
F_j	4.5286e-008
TOTAL POWER	8.54286e-6

The circuit Diagram of Booth Encoder of CMOS Multiplier is



Calculation of Time Delay of output of the Booth Encoder

The output of the Booth Encoder is

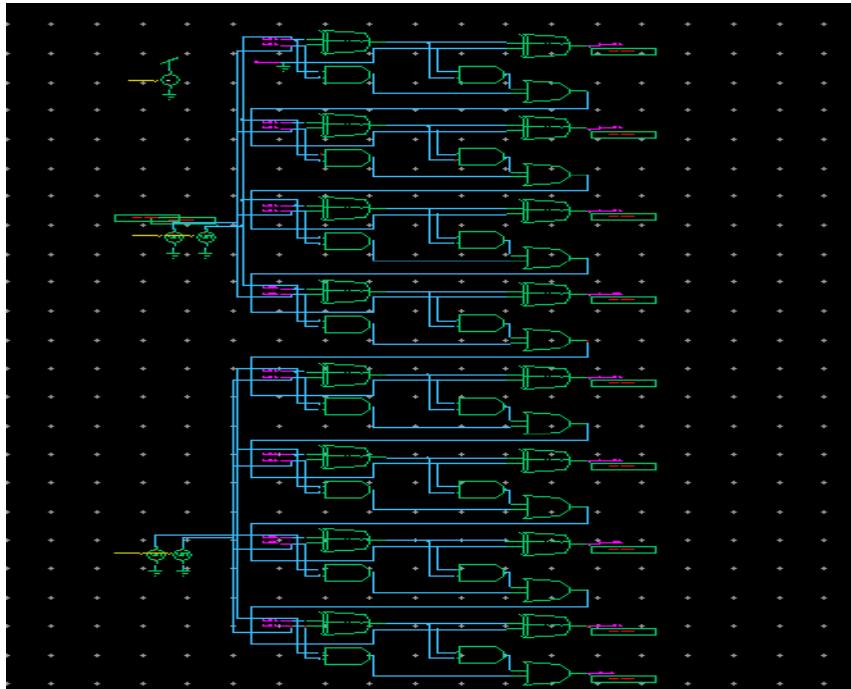


Comparison of Time Delay of output w.r.t input:

	Delay in CMOS
Outbus<0> w.r.t Y2i-1	6.8460e-011
Outbus<1> w.r.t Y2i-1	1.1024e-010
Outbus<2> w.r.t Y2i-1	1.1285e-010
Outbus<3> w.r.t Y2i-1	1.2418e-010
Outbus<4> w.r.t Y2i-1	8.3383e-011
Outbus<0> w.r.t Y2i	9.9315e-009
Outbus<1> w.r.t Y2i	1.0110e-008
Outbus<2> w.r.t Y2i	9.8871e-009
Outbus<3> w.r.t Y2i	1.2418e-010
Outbus<4> w.r.t Y2i	3.9185e-011
Outbus<0> w.r.t Y2i+1	2.9932e-008
Outbus<1> w.r.t Y2i+1	1.0110e-008
Outbus<2> w.r.t Y2i+1	1.9929e-008
Outbus<3> w.r.t Y2i+1	1.2418e-010
Outbus<4> w.r.t Y2i+1	3.9185e-011

Calculation of Power Dissipation of Adder Circuit of CMOS Multiplier:

The Circuit diagram of Adder circuit is



	Pattern	Power
Source1(1v)	0000000011111111	8.7274e-008
Source2(1v)	0000111100001111	5.8733e-008
Source3(1v)	0011001100110011	1.0371e-007
Source4(1v)	0101010101010101	2.4736e-007
TOTAL POWER		0.0004295371

We calculate the power of Half Adder for different input patterns taken in such a way that satisfies the all the conditions or having all the possible cases. The power of VDD Voltage Source is 4.2904e-004.

Comparison of time delay of both GDI & CMOS Multiplier:

	Delay in CMOS	Delay in GDI
Outbus<0> w.r.t Y2i-1	6.8460e-011	7.6990e-009
Outbus<1> w.r.t Y2i-1	1.1024e-010	2.5674e-008
Outbus<2> w.r.t Y2i-1	1.1285e-010	5.9361e-009
Outbus<3> w.r.t Y2i-1	1.2418e-010	1.8549e-010
Outbus<4> w.r.t Y2i-1	8.3383e-011	4.3784e-011
Outbus<0> w.r.t Y2i	9.9315e-009	2.3010e-009
Outbus<1> w.r.t Y2i	1.0110e-008	4.3835e-010
Outbus<2> w.r.t Y2i	9.8871e-009	4.0639e-009
Outbus<3> w.r.t Y2i	1.2418e-010	1.8549e-010
Outbus<4> w.r.t Y2i	3.9185e-011	3.8965e-010

Outbus<0> w.r.t Y2i+1	2.9932e-008	2.7699e-008
Outbus<1> w.r.t Y2i+1	1.0110e-008	1.4326e-008
Outbus<2> w.r.t Y2i+1	1.9929e-008	2.5936e-008
Outbus<3> w.r.t Y2i+1	1.2418e-010	1.8549e-010
Outbus<4> w.r.t Y2i+1	3.9185e-011	9.6104e-009

From the above table we say that the delay is more in GDI Multiplier compared to CMOS Multiplier.

Comparison of power dissipated in both GDI Multiplier and CMOS Multiplier:

Source	Power in CMOS	Power in GDI
VDD	8.2437e-006	5.8902e-006
Y2i+1	4.5346e-008	2.5952e-008
Y2i	6.6163e-008	6.9726e-008
Y2i-1	1.1919e-007	1.3979e-007
PP n , j	2.3175e-008	9.6417e-013
F j	4.5286e-008	1.6801e-006
TOTAL POWER	8.54286e-6	7.805768e-6

From the above table we say that the power dissipation is more in CMOS Multiplier compared to GDI Multiplier by 8%. This is due to less transistors are used GDI Multiplier when compared to CMOS Multiplier. So that the power is less in GDI Multiplier when compared to CMOS Multiplier.

Combinational Circuits

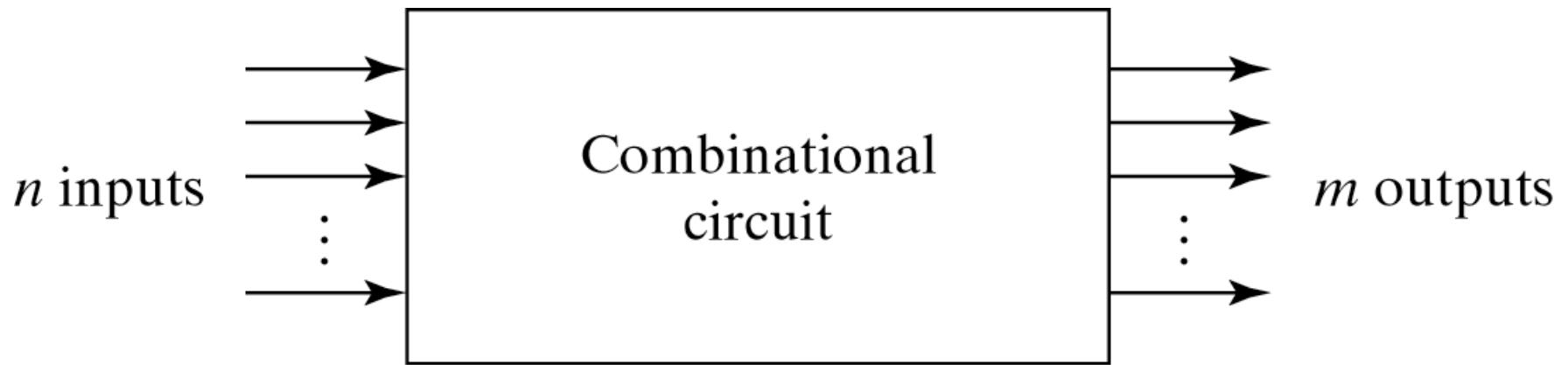


Fig. 4-1 Block Diagram of Combinational Circuit

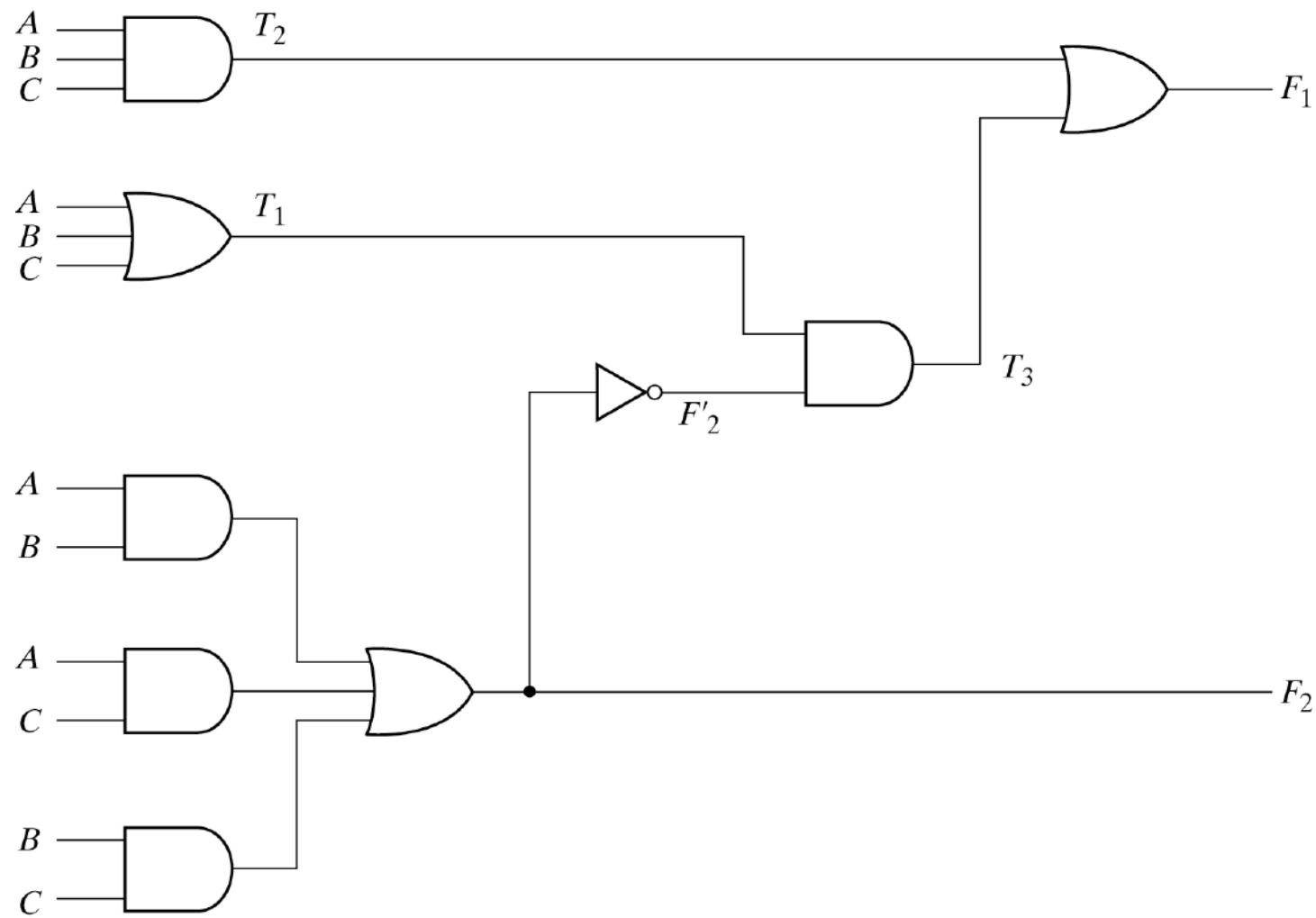
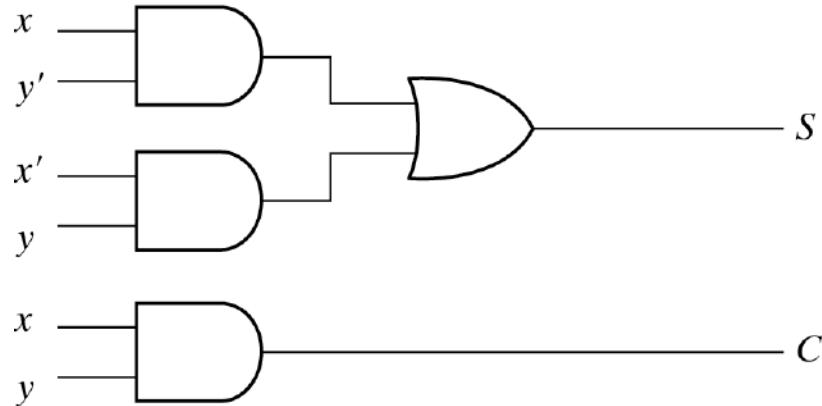


Fig. 4-2 Logic Diagram for Analysis Example

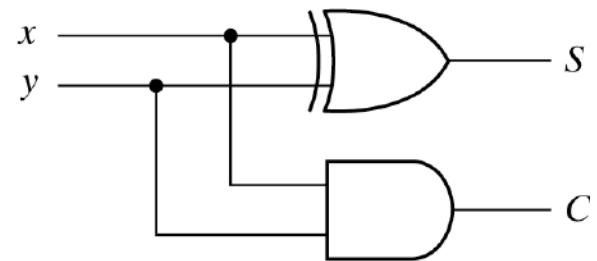
Designing Combinational Circuits

In general we have to do following steps:

1. Problem description
2. Input/output of the circuit
3. Define truth table
4. Simplification for each output
5. Draw the circuit

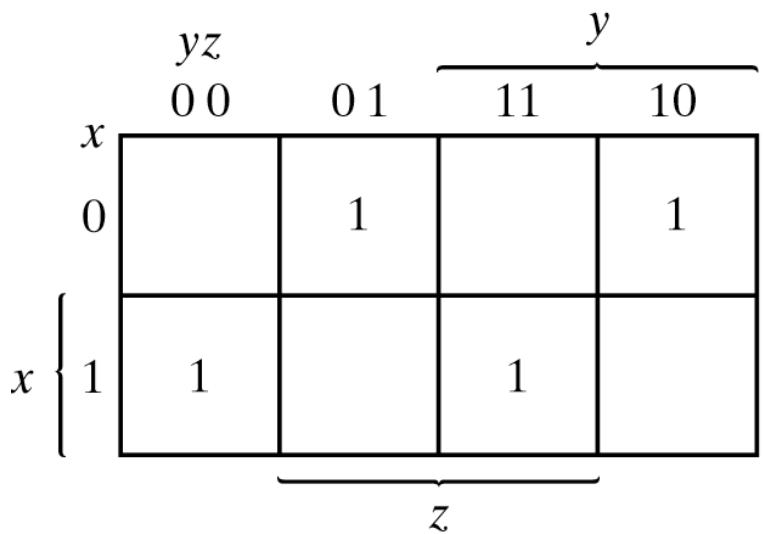


$$(a) S = xy' + x'y \\ C = xy$$

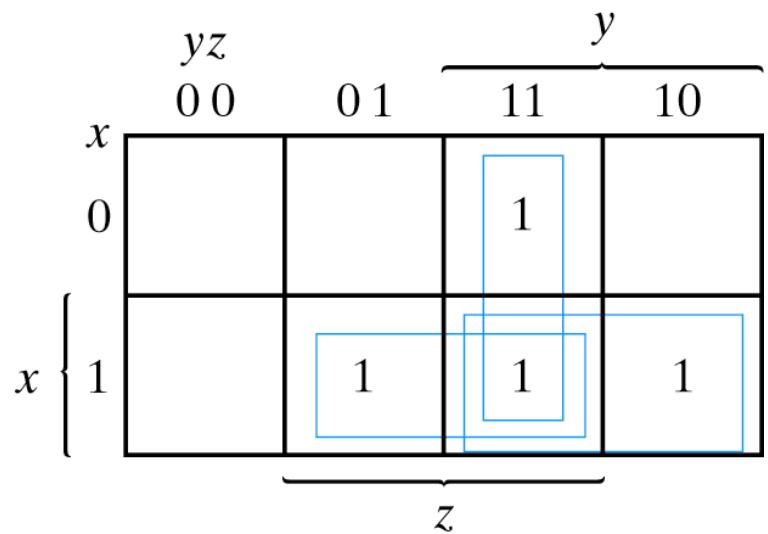


$$(b) S = x \oplus y \\ C = xy$$

Fig. 4-5 Implementation of Half-Adder



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



$$\begin{aligned} S &= xy + xz + yz \\ &= xy + xy'z + x'yz \end{aligned}$$

Fig. 4-6 Maps for Full Adder

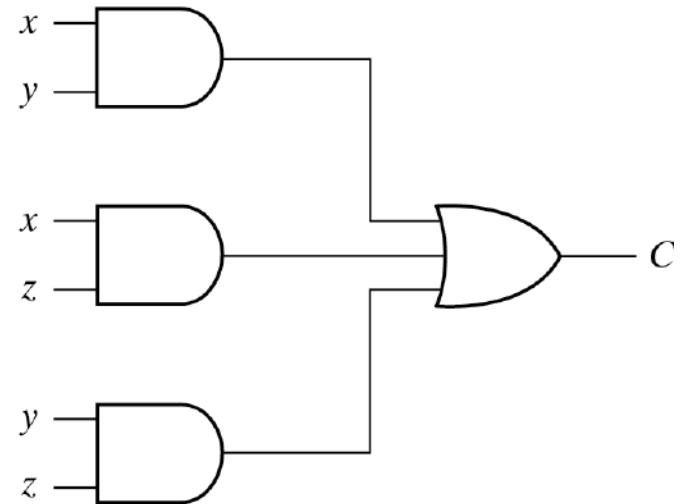
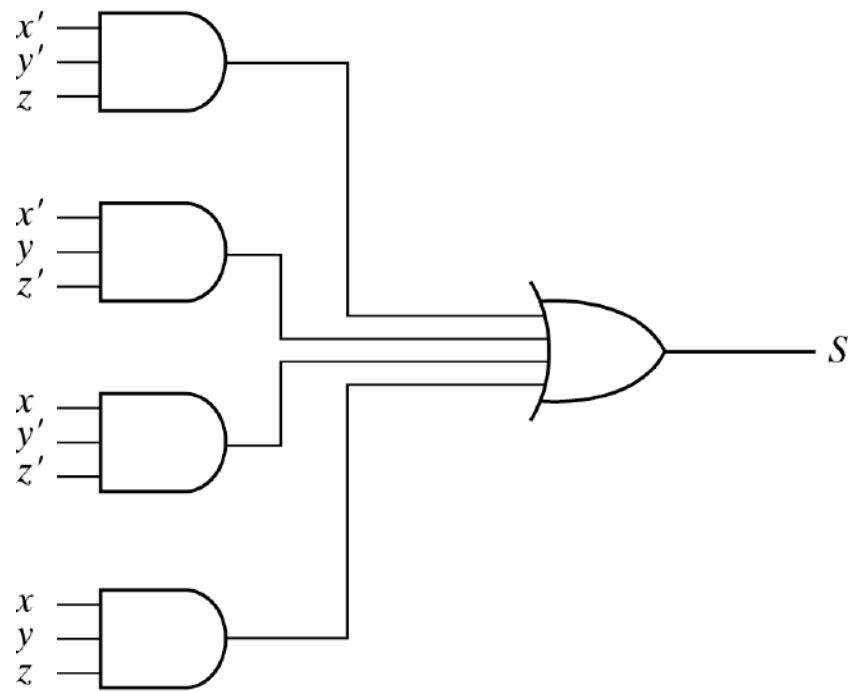


Fig. 4-7 Implementation of Full Adder in Sum of Products

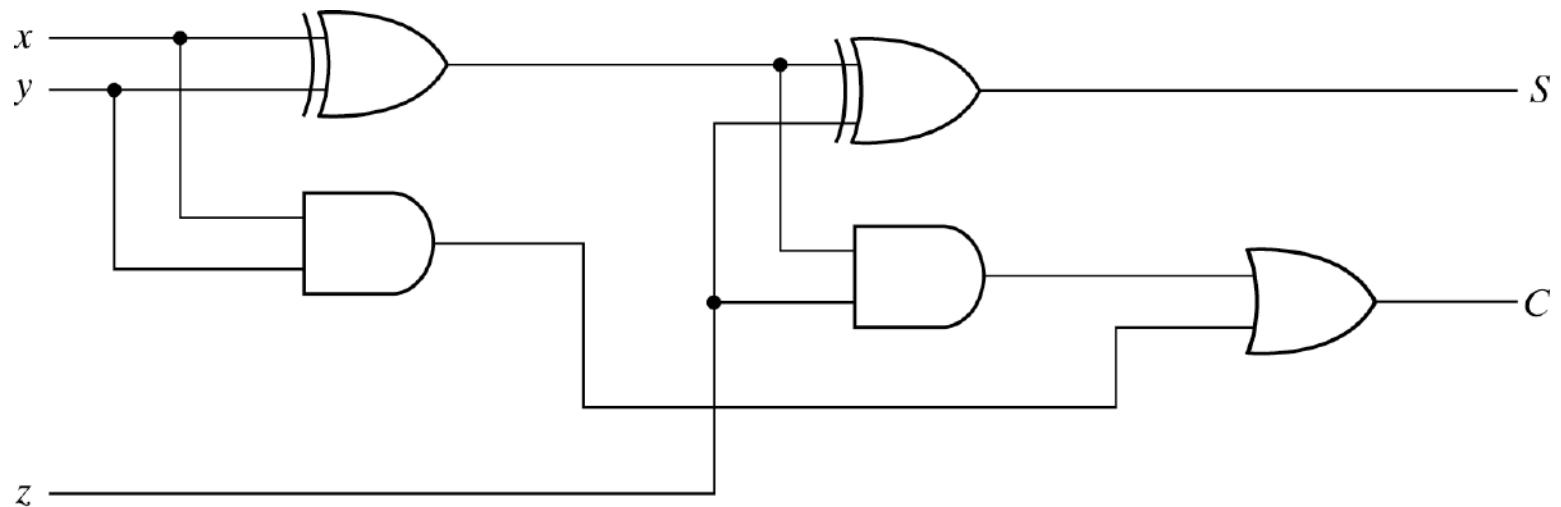


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

Binary adder

- Binary adder that produces the arithmetic sum of binary numbers can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain
- Note that the input carry C_0 in the least significant position must be 0.

Binary Adder

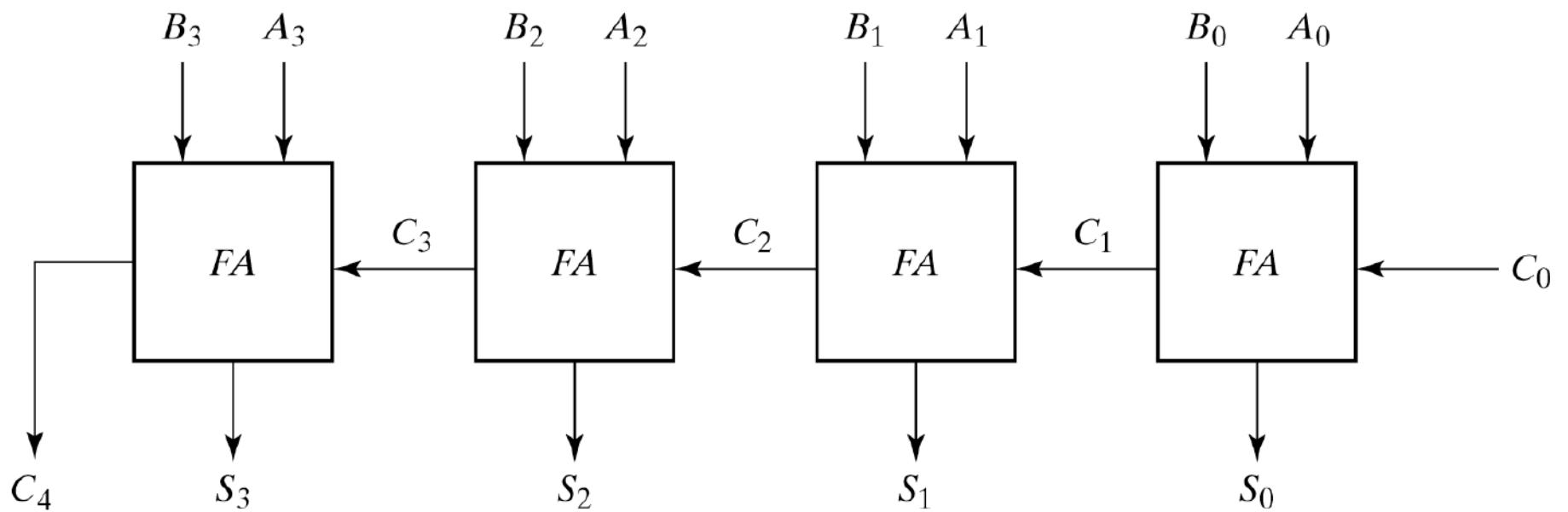


Fig. 4-9 4-Bit Adder

Binary Adder

- For example to add $A = 1011$ and $B = 0011$

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

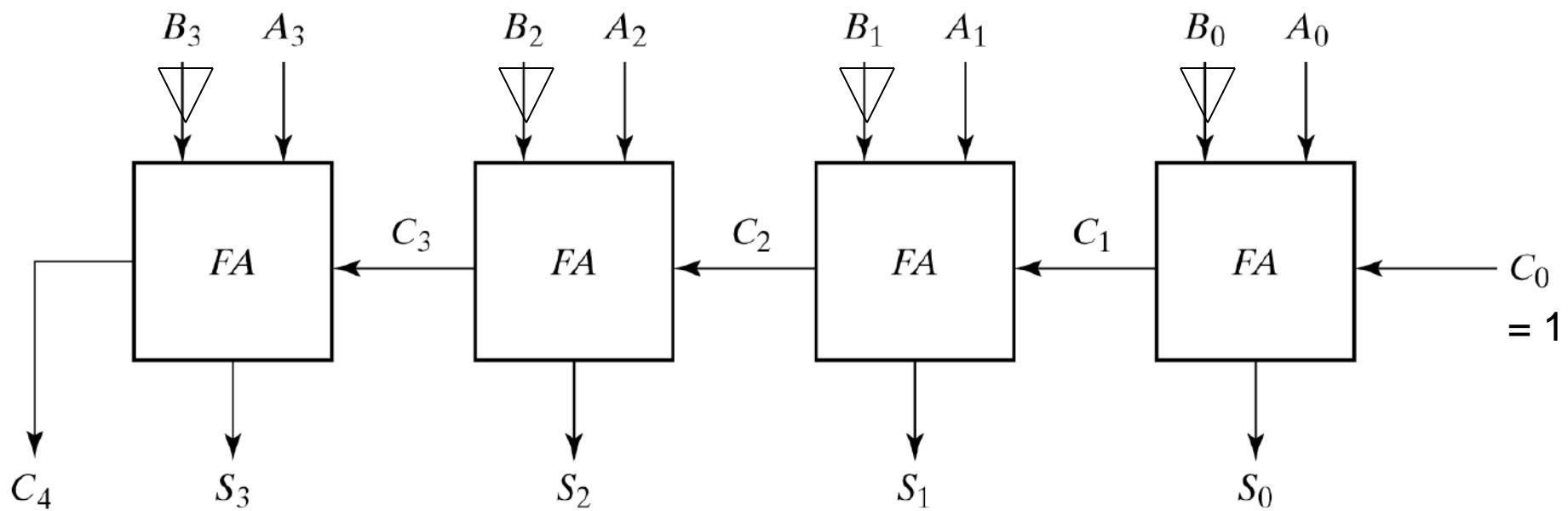
Sum: 1 1 1 0 S_i

Output carry: 0 0 1 1 C_{i+1}

Binary Subtractor

- The subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A because $A - B = A + (-B)$
- It means if we use the inverters to make 1's complement of B (connecting each B_i to an inverter) and then add 1 to the least significant bit (by setting carry C_0 to 1) of binary adder, then we can make a binary subtractor.

4 bit 2's complement Subtractor



Adder Subtractor

- The addition and subtraction can be combined into one circuit with one common binary adder (see next slide).
- The mode M controls the operation. When M=0 the circuit is an adder when M=1 the circuit is subtractor. It can be done by using exclusive-OR for each Bi and M. Note that $1 \oplus x = x'$ and $0 \oplus x = x$

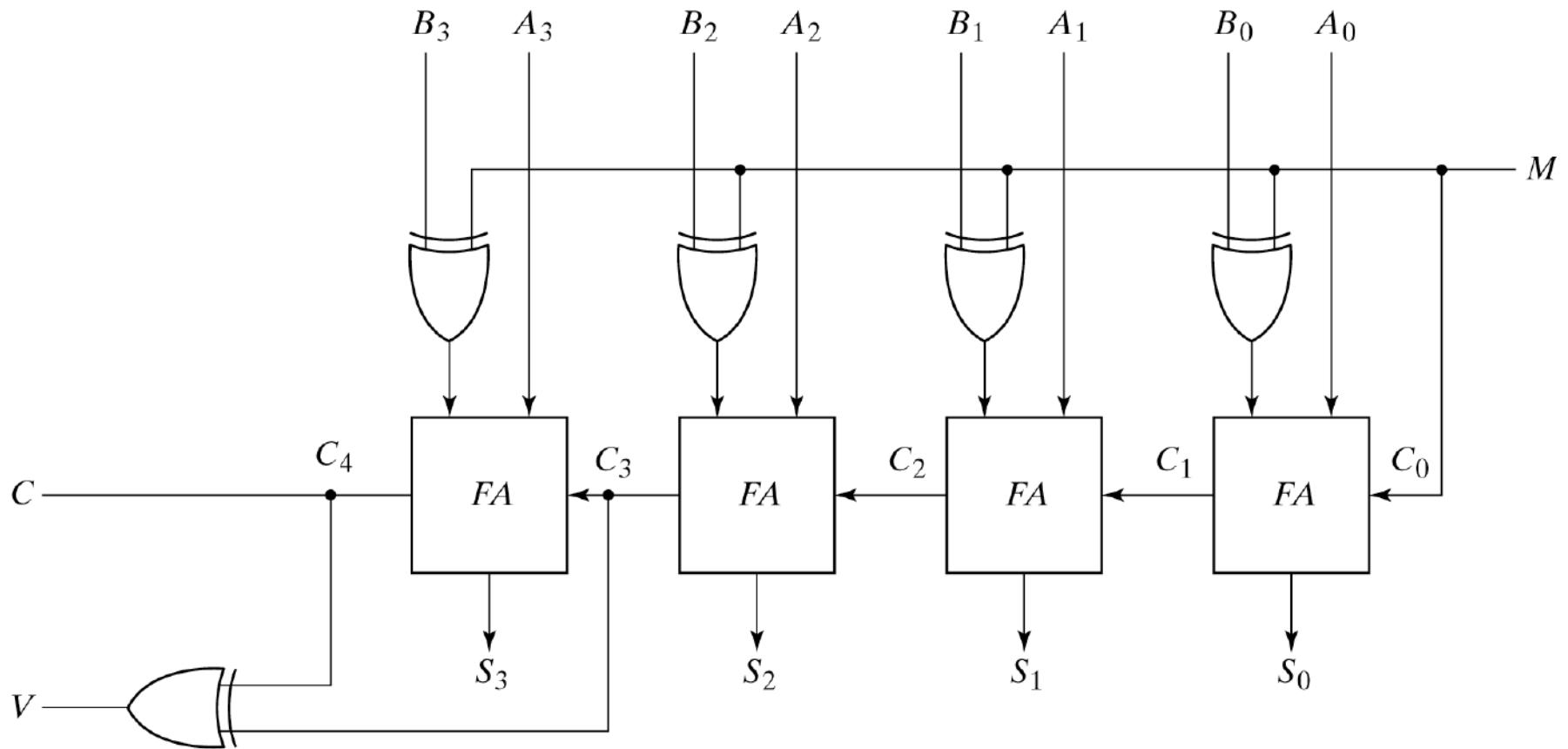


Fig. 4-13 4-Bit Adder Subtractor

Checking Overflow

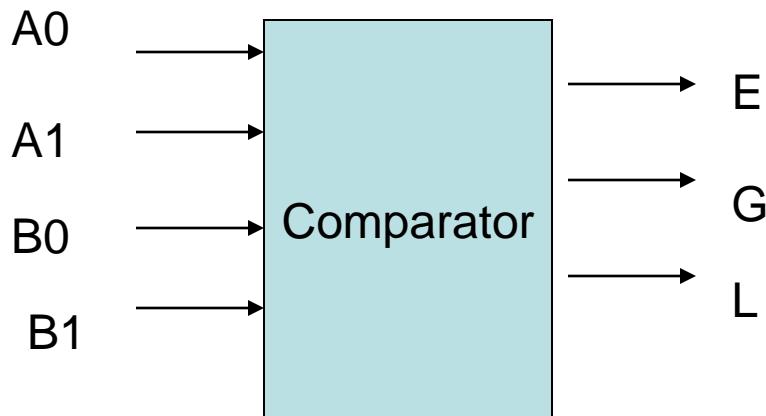
- Note that in the previous slide if the numbers considered to be signed V detects overflow. V=0 means no overflow and V=1 means the result is wrong because of overflow
- Overflow can be happened when adding two numbers of the same sign (both negative or positive) and result can not be shown with the available bits. It can be detected by observing the carry into sign bit and carry out of sign bit position. If these two carries are not equal an overflow occurred. That is why these two carries are applied to exclusive-OR gate to generate V.

Magnitude Comparator

- It is a combinational circuit that compares two numbers and determines their relative magnitude
- The output of comparator is usually 3 binary variables indicating:
 - A>B
 - A=B
 - A<B
- For example to design a comparator for 2 bit binary numbers A (A₁A₀) and B (B₁B₀) we do the following steps:

Comparators

- For a 2-bit comparator we have four inputs A1A0 and B1B0 and three output E (is 1 if two numbers are equal) G (is 1 when A > B) and L (is 1 when A < B) If we use truth table and KMAP the result is
- $E = A'1A'0B'1B'0 + A'1A0B'1B0 + A1A0B1B0 + A1A'0B1B'0$
or $E = ((A0 \oplus B0) + (A1 \oplus B1))'$ (see next slide)
- $G = A1B'1 + A0B'1B'0 + A1A0B'0$
- $L = A'1B1 + A'1A'0B0 + A'0B1B0$



2-Bit Magnitude Comparator

- Block Diagram of n-Bit Magnitude Comparator

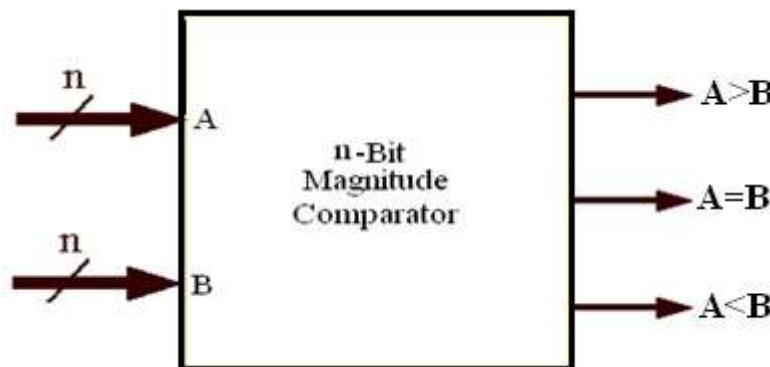
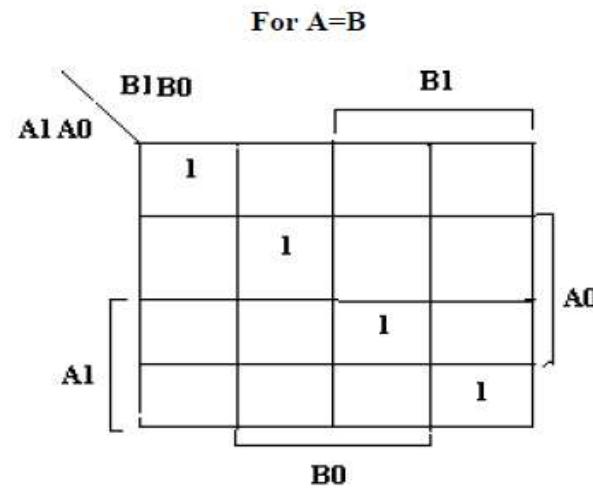
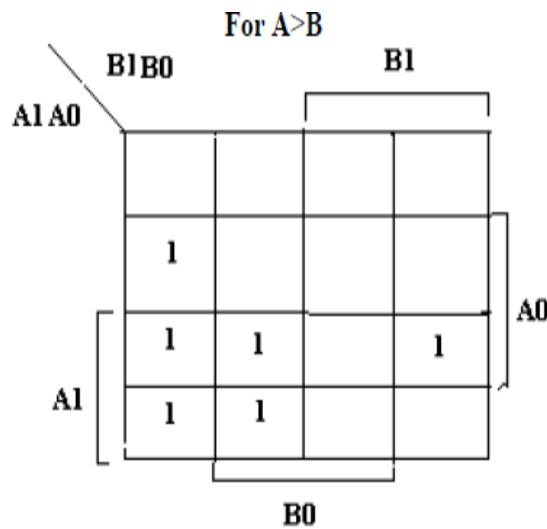


Table 1. Truth Table of 2-Bit Magnitude Comparator

INPUT				OUTPUT		
A1	A0	B1	B0	A>B	A=B	A<B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

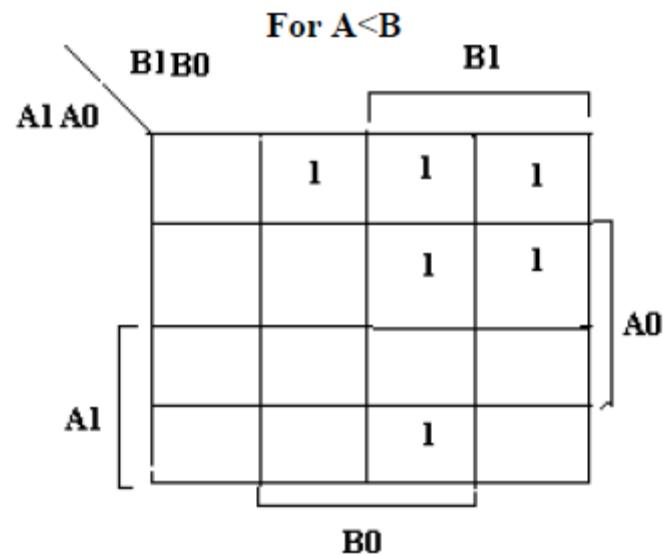
kmaps



$$\begin{aligned}
 A > B: &= A1B1' + A0B0'A1'B1' + A0B0'A1B1 \\
 &= A1B1' + A0B0'(A1'B1' + A1B1) \\
 &= A1B1' + A0B0' X1
 \end{aligned}$$

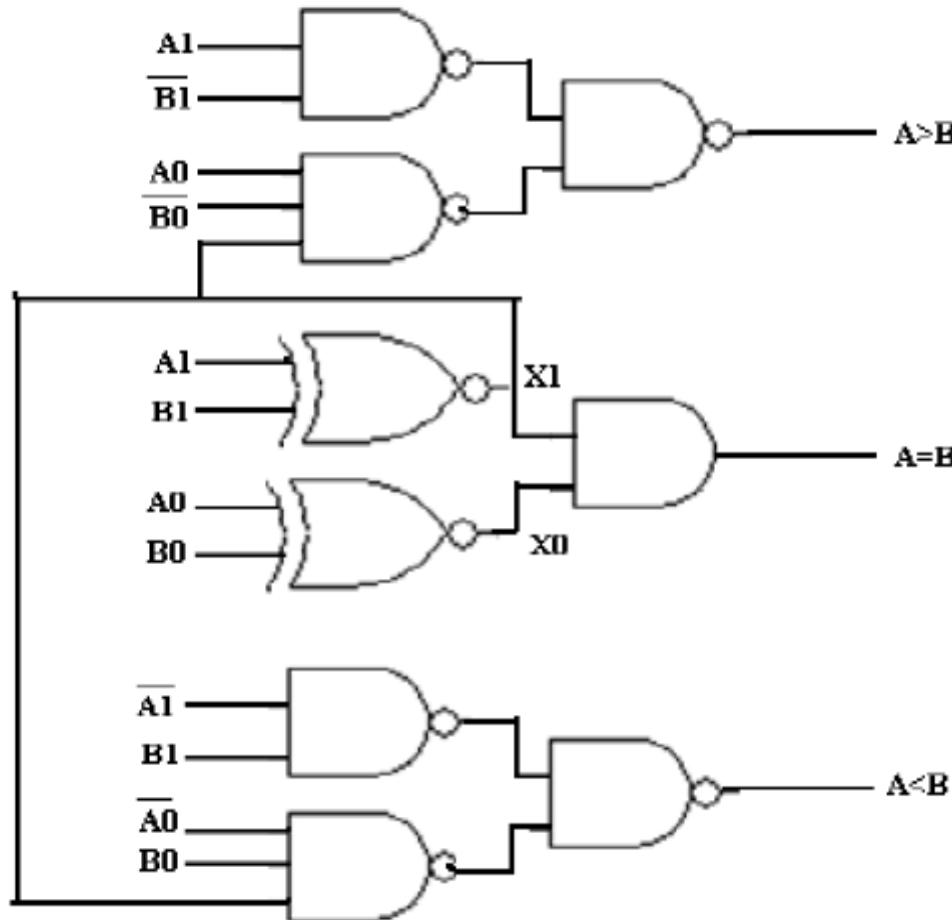
$$\begin{aligned}
 A = B: &= A1'A0'B1'B0' + A1'A0B1'B0 + A1A0'B1B0' + \bar{A}1A0B1B0 \\
 &= (A1'B1' + A1B1)(A0'B0' + A0B0) \\
 &= X1X0
 \end{aligned}$$

Contd...



$$\begin{aligned} A < B : &= A1'B1 + A0'B0A1'B1' + A0'B0A1B1 \\ &= A1'B1 + A0'B0(A1'B1' + A1B1) \\ &= A1'B1 + A0'B0 X1 \end{aligned}$$

LOGIC DIAGRAM

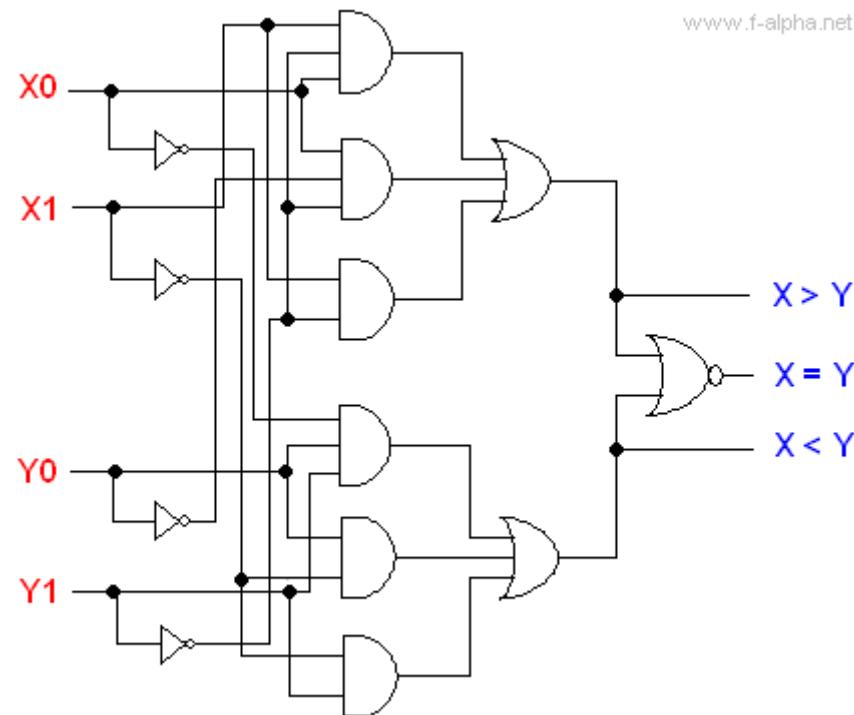


Logic Diagram of 2-Bit Magnitude Comparator

Truth table of a 2-bit magnitude comparator

x₁	x₀	y₁	y₀	x<y	x=y	x>y
0	0	0	0	0	1	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	1	1	0	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	0	1	0

LOGIC DIAGRAM



Magnitude Comparator

- Here we use simpler method to find E (called X) and G (called Y) and L (called Z)
- $A=B$ if all $A_i=B_i$

A_i	B_i	X_i
0	0	1
0	1	0
1	0	0
1	1	0

It means $X_0 = A_0B_0 + A'_0B'_0$ and

$$X_1 = A_1B_1 + A'_1B'_1$$

If $X_0=1$ and $X_1=1$ then $A_0=B_0$ and $A_1=B_1$

Thus, if $A=B$ then $X_0X_1 = 1$ it means

$$X = (A_0B_0 + A'_0B'_0)(A_1B_1 + A'_1B'_1) \text{ since } (x \oplus y)' = (xy + x'y')$$

$$X = (A_0 \oplus B_0)' (A_1 \oplus B_1)' = ((A_0 \oplus B_0) + (A_1 \oplus B_1))'$$

It means for X we can NOR the result of two exclusive-OR gates

Magnitude Comparator

- $A > B$ means $A_1 \ B_1 \ Y_1$

0	0	0
0	1	0
1	0	1
1	1	0

if $A_1 = B_1$ ($X_1 = 1$) then A_0 should be 1 and B_0 should be 0

$A_0 \ B_0 \ Y_0$

0	0	1
0	1	0
1	0	0
1	1	0

For $A > B$: $A_1 > B_1$ or $A_1 = B_1$ and $A_0 > B_0$

It means $Y = A_1 B'_1 + X_1 A_0 B'_0$ should be 1 for $A > B$

Magnitude Comparator

- For $B > A$ $B_1 > A_1$
or
 $A_1 = B_1$ and $B_0 > A_0$
- The procedure for binary numbers with more than 2 bits can also be found in the similar way. For example next slide shows the 4-bit magnitude comparator, in which

$$(A = B) = x_3x_2x_1x_0$$

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

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

TRUTH TABLE

Table 10.1 Truth table of 7485

A_3	B_3	Comparing inputs				Cascading inputs			Outputs				
		A_3	B_3	A_1	B_1	A_0	B_0	$A > B$	$A < B$	$A = B$	$A > B$	$A < B$	$A = B$
$A_3 > B_3$		x		x		x		x	x	x	1	0	0
$A_3 < B_3$		x		x		x		x	x	x	0	1	0
$A_3 = B_3$		$A_2 > B_2$		x		x		x	x	x	1	0	0
$A_3 = B_3$		$A_2 < B_2$		x		x		x	x	x	0	1	0
$A_3 = B_3$		$A_2 = B_2$		$A_1 > B_1$		x		x	x	x	1	0	0
$A_3 = B_3$		$A_2 = B_2$		$A_1 < B_1$		x		x	x	x	0	1	0
$A_3 = B_3$		$A_2 = B_2$		$A_1 = B_1$		$A_0 > B_0$		x	x	x	1	0	0
$A_3 = B_3$		$A_2 = B_2$		$A_1 = B_1$		$A_0 < B_0$		x	x	x	0	1	0
$A_3 = B_3$		$A_2 = B_2$		$A_1 = B_1$		$A_0 = B_0$		1	0	0	1	0	0
$A_3 = B_3$		$A_2 = B_2$		$A_1 = B_1$		$A_0 = B_0$		0	1	0	0	1	0
$A_3 = B_3$		$A_2 = B_2$		$A_1 = B_1$		$A_0 = B_0$		0	0	1	0	0	1
$A_3 = B_3$		$A_2 = B_2$		$A_1 = B_1$		$A_0 = B_0$		x	x	1	0	0	1
$A_3 = B_3$		$A_2 = B_2$		$A_1 = B_1$		$A_0 = B_0$		1	1	0	0	0	0
$A_3 = B_3$		$A_2 = B_2$		$A_1 = B_1$		$A_0 = B_0$		0	0	0	1	1	0

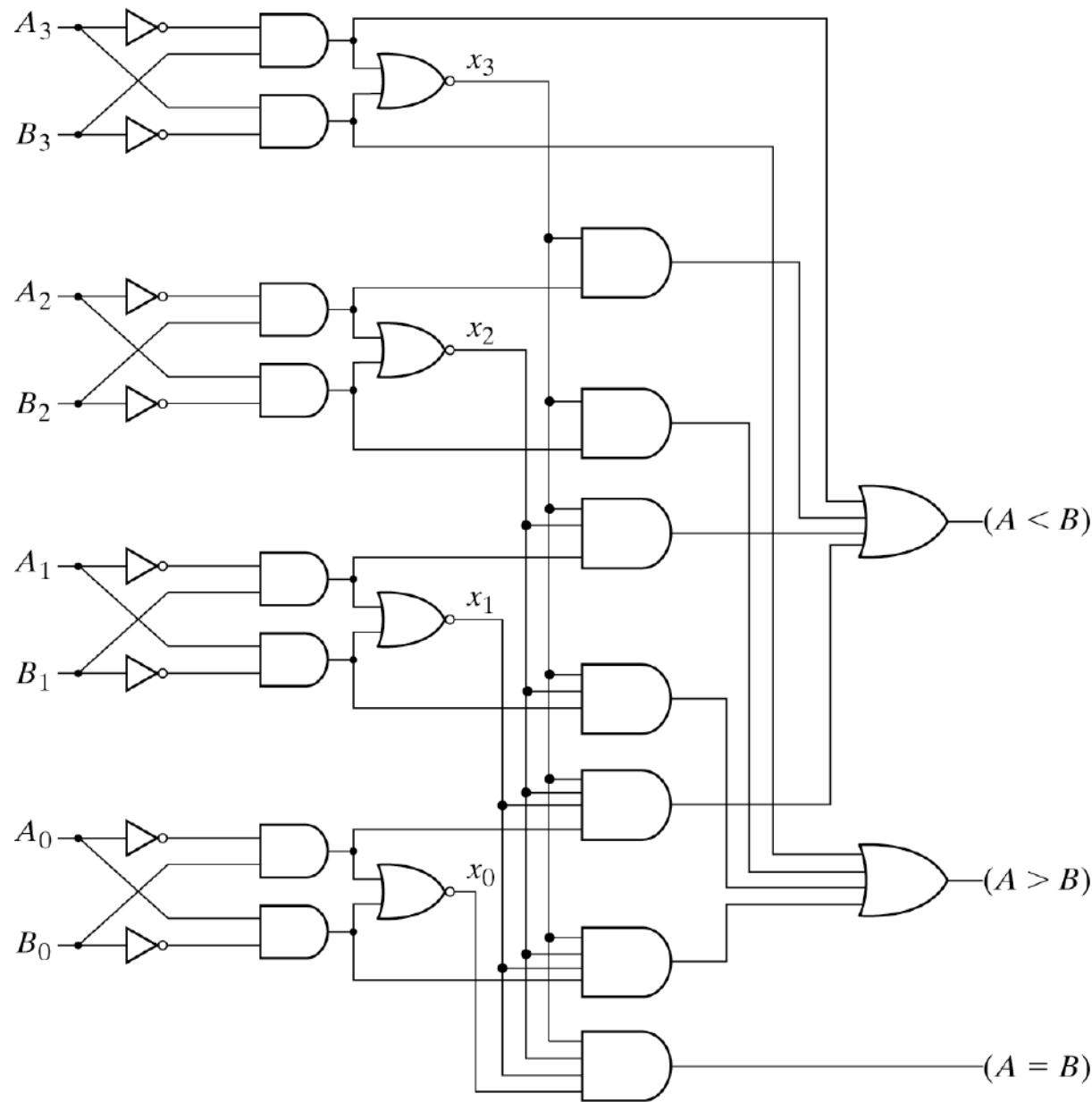
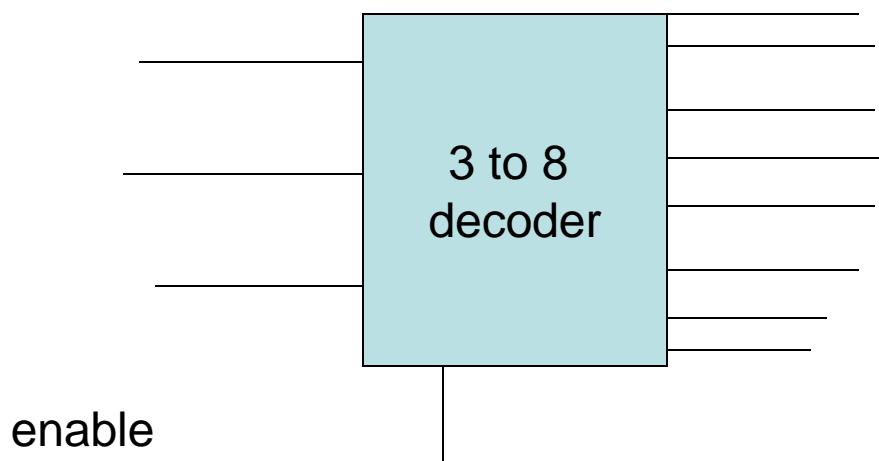


Fig. 4-17 4-Bit Magnitude Comparator

Decoder

- Is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines
For example if the number of input is $n=3$ the number of output lines can be $m=2^3$. It is also known as 1 of 8 because one output line is selected out of 8 available lines:



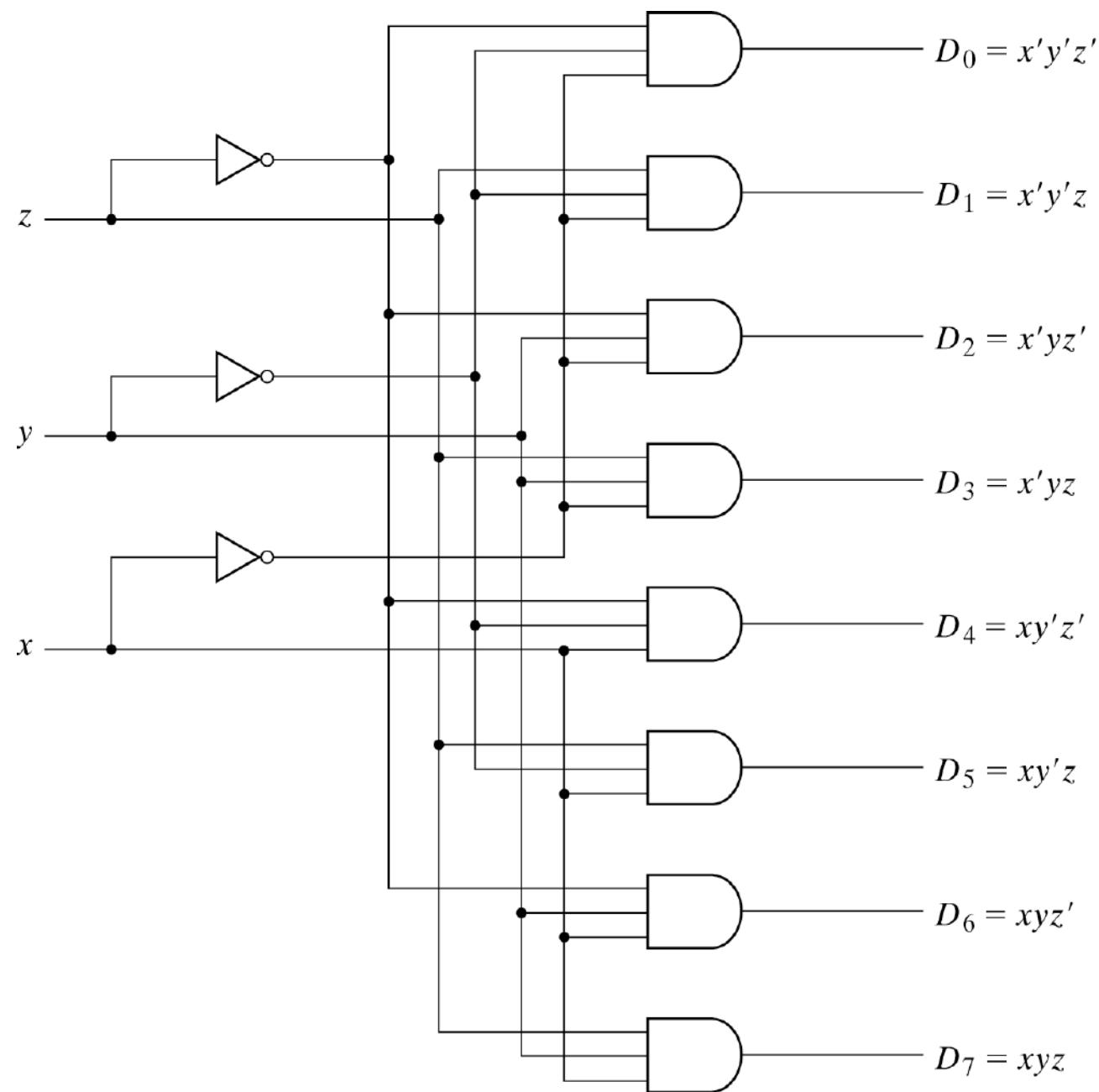


Fig. 4-18 3-to-8-Line Decoder

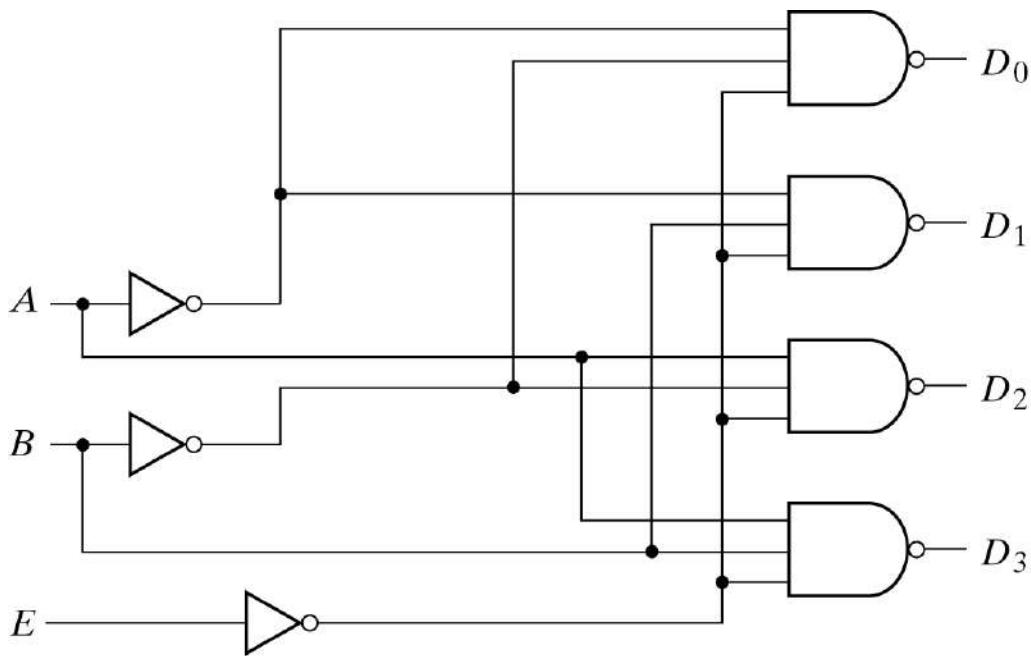
Decoder with Enable Line

- Decoders usually have an enable line,
- If $\text{enable}=0$, decoder is off. It means all output lines are zero
- If $\text{enable}=1$, decoder is on and depending on input, the corresponding output line is 1, all other lines are 0
- See the truth table in next slide

Truth table for decoder

E	a ₂	a ₁	a ₀	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
---	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

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



(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

Major application of Decoder

- Decoder is used to implement any combinational circuits (f^n)
For example the truth table for full adder is $s(x,y,z) = \sum (1,2,4,7)$ and $C(x,y,z) = \sum (3,5,6,7)$. The implementation with decoder is:

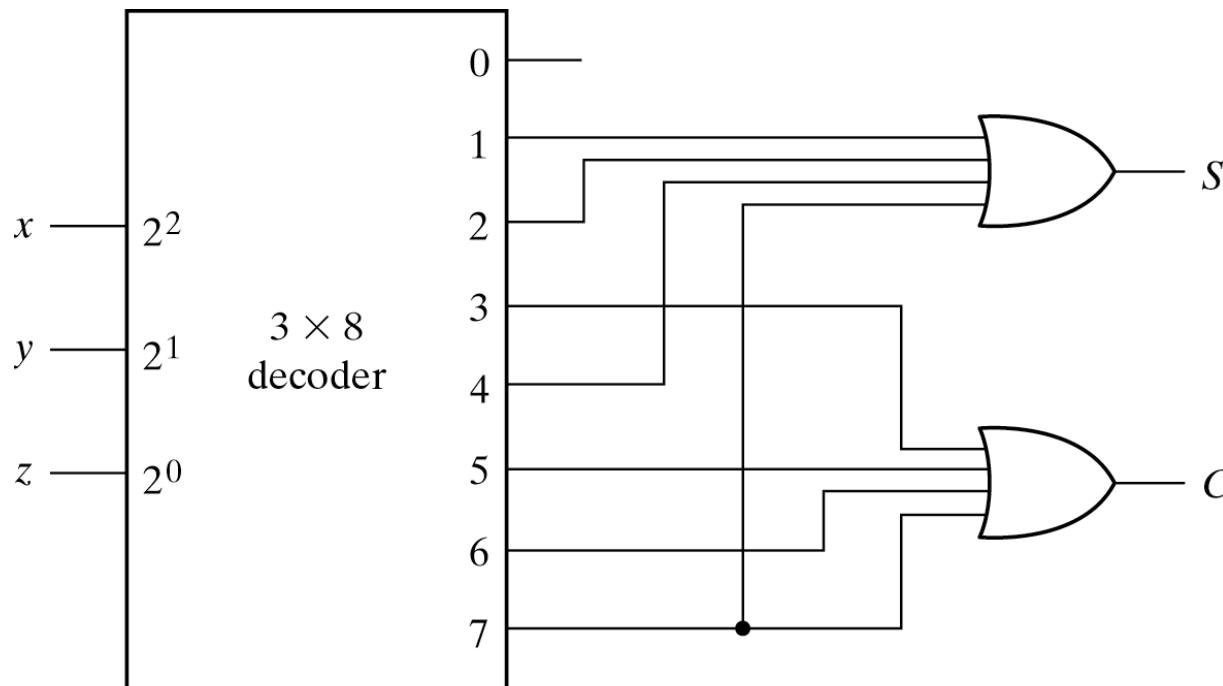
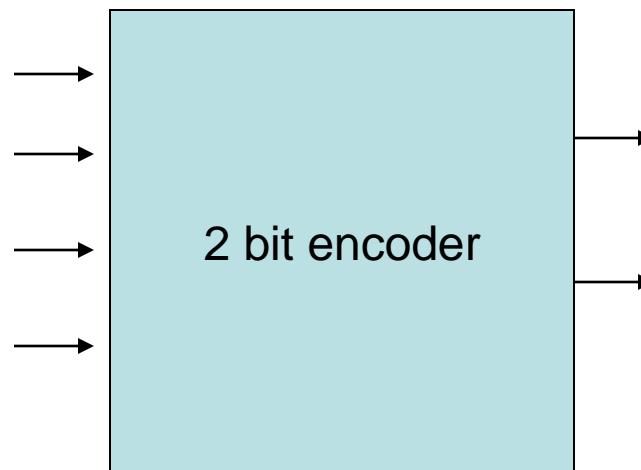


Fig. 4-21 Implementation of a Full Adder with a Decoder

Encoder

- Encoder is a digital circuit that performs the inverse operation of a decoder
- Generates a unique binary code from several input lines.
- Generally encoders produce 2-bit, 3-bit or 4-bit code. n bit encoder has 2^n input lines



2-bit encoder

- If one of the four input lines is active encoder produces the binary code corresponding to that line
- If more than one of the input lines will be activated or all the output is undefined. We can consider don't care for these situations but in general we can solve this problem by using priority encoder.

2-bit Priority Encoder

- A priority encoder is an encoder circuit that includes priority function.
- It means if two or more inputs are equal to 1 at the same time, the input having higher subscript number, considered as a higher priority. For example if D3 is 1 regardless of the value of the other input lines the result of output is 3 which is 11.
- If all inputs are 0, there is no valid input. For detecting this situation we considered a third output named V. V is equal to 0 when all input are 0 and is one for rest of the situations of TT.

2-bit Priority Encoder

- By using TT and K-map we get following boolean functions for 4-input (or 2-bit) priority encoder:
 - $X = D_2 + D_3$
 - $Y = D_3 + D_1D'_2$
 - $V = D_0 + D_1 + D_2 + D_3$
- See next two slides for K-maps and the logic circuit of 2-bit priority encoder

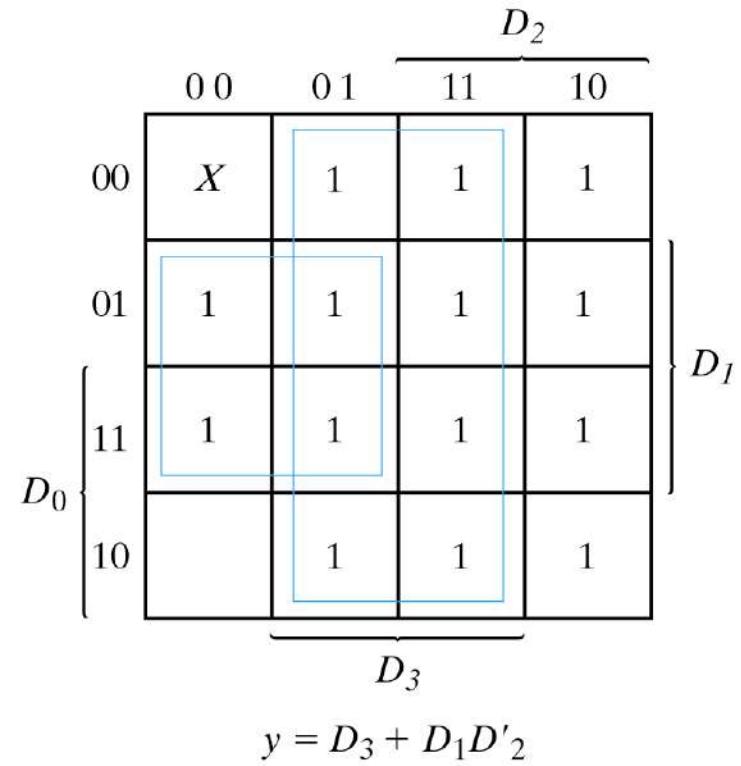
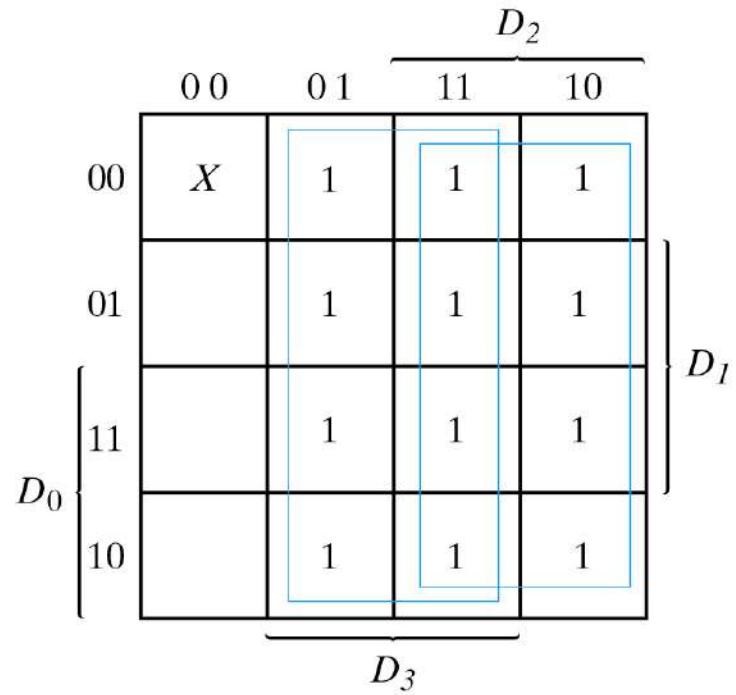


Fig. 4-22 Maps for a Priority Encoder

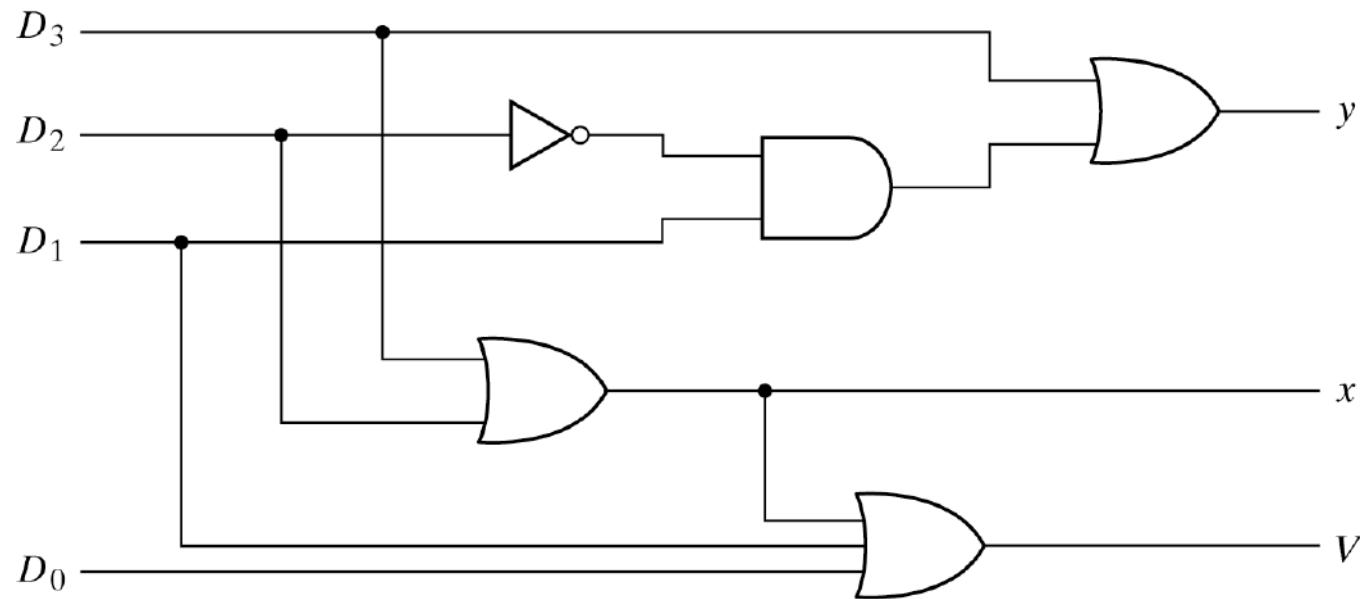
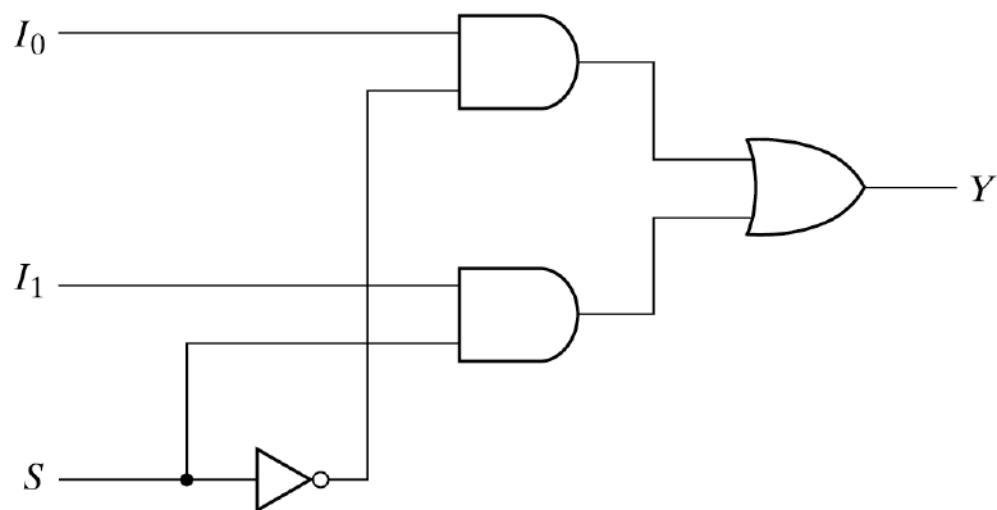


Fig. 4-23 4-Input Priority Encoder

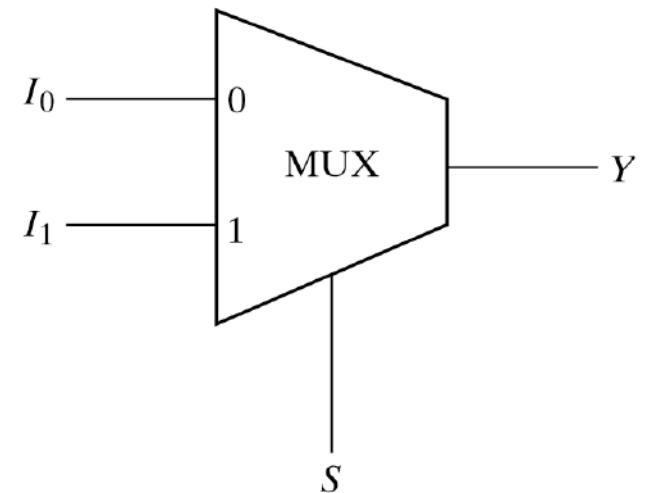
Multiplexer

- It is a combinational circuit that selects binary information from one of the input lines and directs it to a single output line
- Usually there are 2^n input lines and n selection lines whose bit combinations determine which input line is selected
- For example for 2-to-1 multiplexer if selection S is zero then I_0 has the path to output and if S is one I_1 has the path to output (see the next slide)

2-to-1 multiplexer

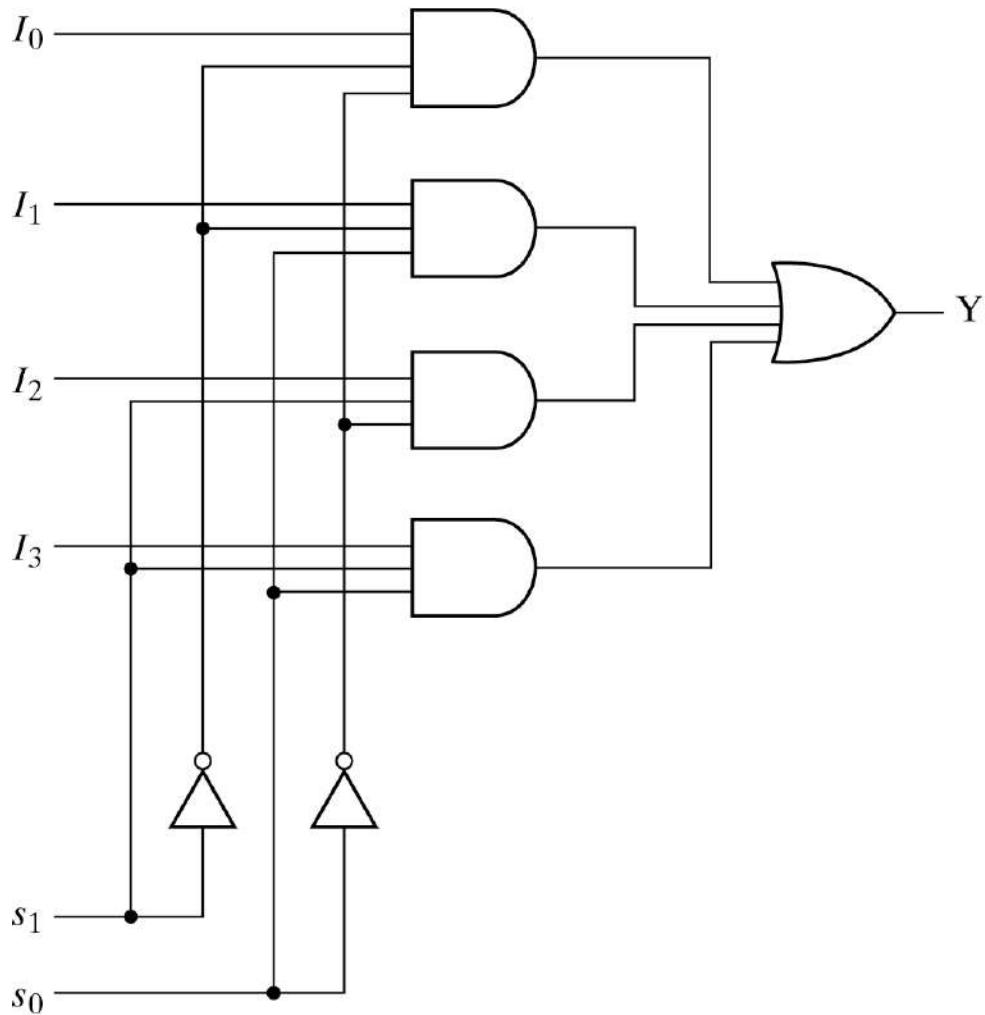


(a) Logic diagram



(b) Block diagram

Fig. 4-24 2-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

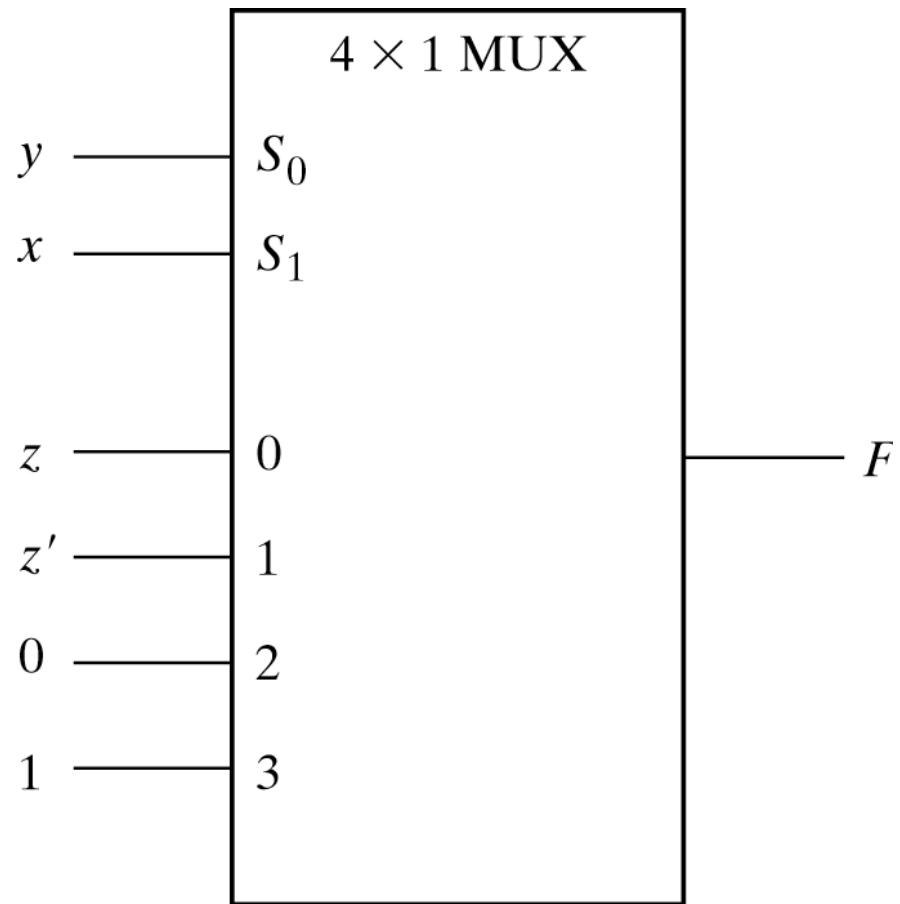
Fig. 4-25 4-to-1-Line Multiplexer

Boolean function Implementation

- Another method for implementing boolean function is using multiplexer
- For doing that assume boolean function has n variables. We have to use multiplexer with n-1 selection lines and
- 1- first n-1 variables of function is used for data input
- 2- the remaining single variable (named z)is used for data input. Each data input can be z, z' , 1 or 0. From truth table we have to find the relation of F and z to be able to design input lines. For example : $f(x,y,z) = \sum(1,2,6,7)$

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

(a) Truth table



(b) Multiplexer implementation

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

$$F_{A,B,C,D} = \sum(1,3,4,11,12,13,14,15)$$

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

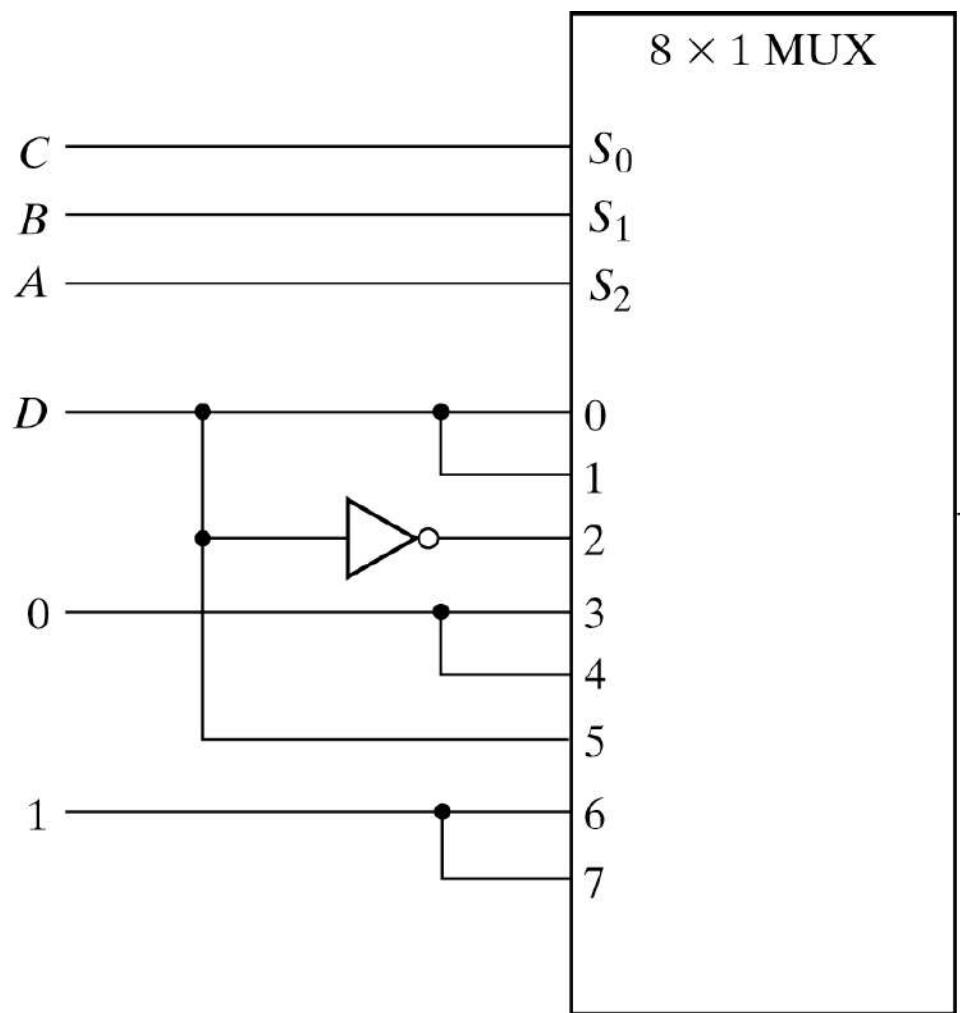


Fig. 4-28 Implementing a 4-Input Function with a Multiplexer

Three-State Gates

- Three state gates exhibit three states instead of two states. The three states are:
- high : 1
- Low : 0
- High impedance : In that state the output is disconnected which is equal to open circuit. In the other words in that state circuit has no logic significant. We can have AND or NAND three-state gates but the most common is three-state buffer gate

Three-State Gates

- We may use conventional gates such as AND or NAND as tree-state gates but the most common is three-state buffer gate.
- Note that buffer produces transfer function and can be used for power amplification. Three state buffer has extra input control line entering the bottom of the gate symbol (see next slide)

Three-state buffer

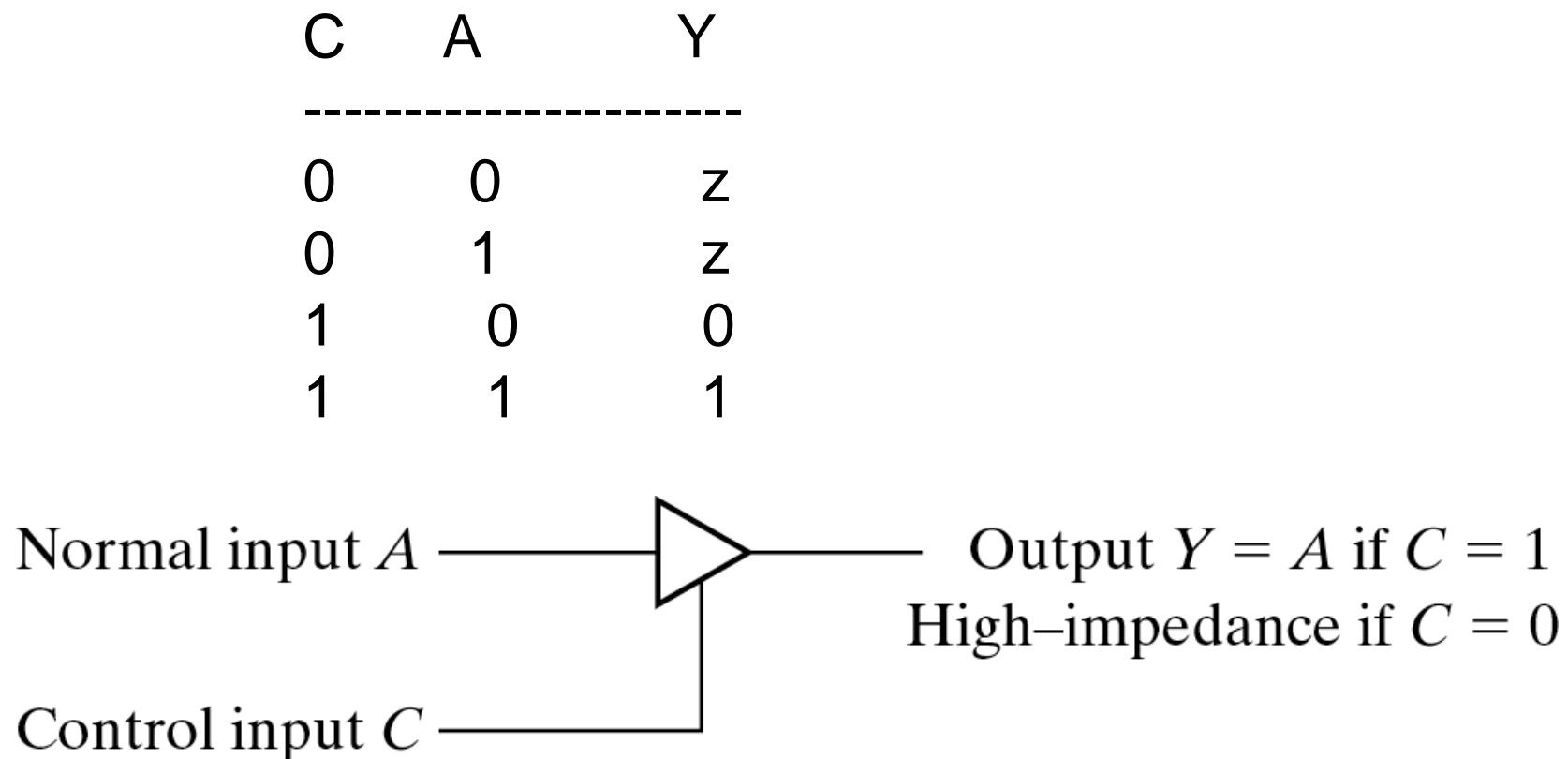


Fig. 4-29 Graphic Symbol for a Three-State Buffer

Three-state buffers can be used to implement multiplexer

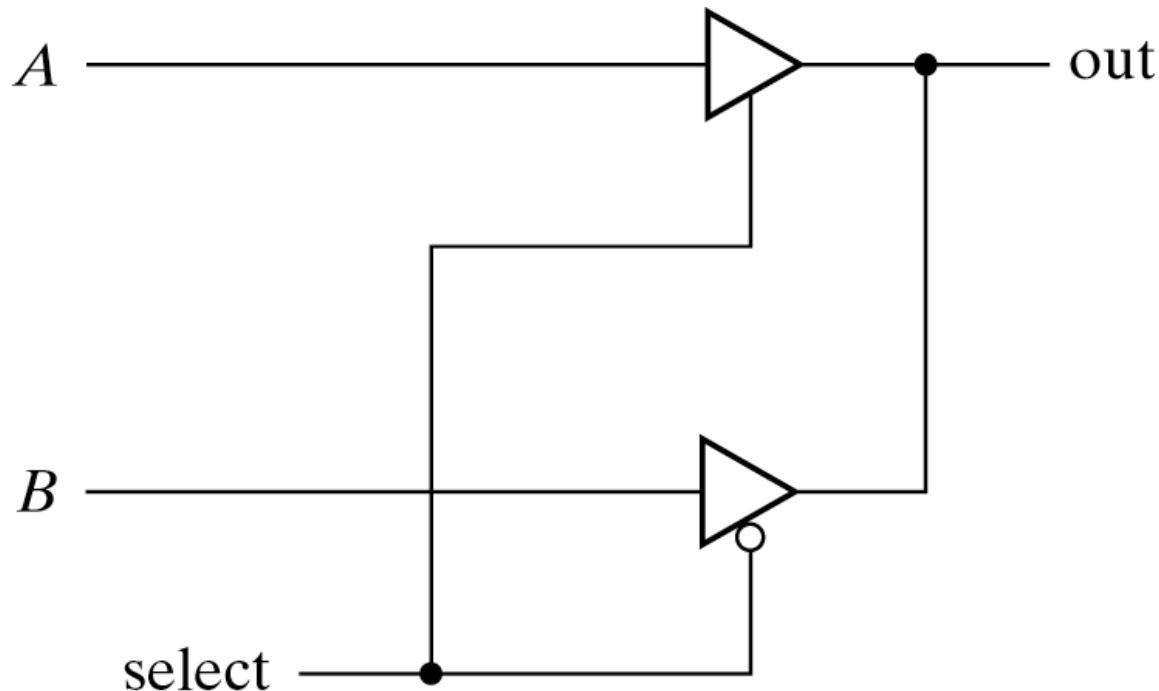


Fig. 4-32 2-to-1-Line Multiplexer with Three-State Buffers

Programmable Logic Devices

Programmable Logic Devices PLDsPLDs are the integrated circuits. They contain an array of AND gates & another array of OR gates. There are three kinds of PLDs based on the type of arrays, which has programmable feature.

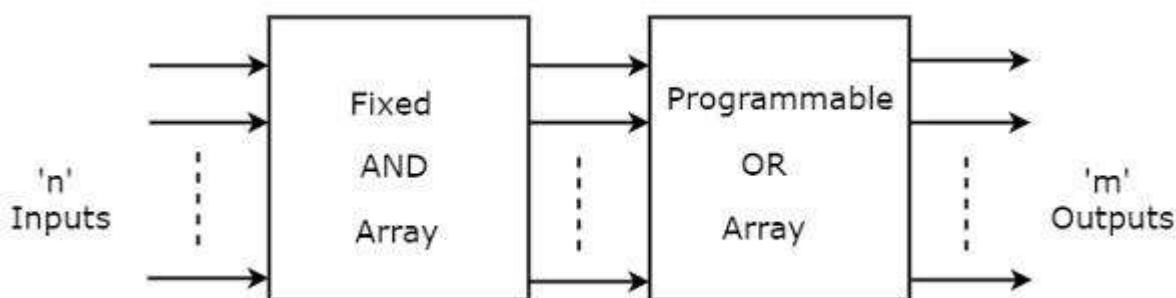
- Programmable Read Only Memory
- Programmable Array Logic
- Programmable Logic Array

The process of entering the information into these devices is known as **programming**. Basically, users can program these devices or ICs electrically in order to implement the Boolean functions based on the requirement. Here, the term programming refers to hardware programming but not software programming.

Programmable Read Only Memory PROMPROM

Read Only Memory ROMROM is a memory device, which stores the binary information permanently. That means, we can't change that stored information by any means later. If the ROM has programmable feature, then it is called as **Programmable ROM PROMPROM**. The user has the flexibility to program the binary information electrically once by using PROM programmer.

PROM is a programmable logic device that has fixed AND array & Programmable OR array. The **block diagram** of PROM is shown in the following figure.



Here, the inputs of AND gates are not of programmable type. So, we have to generate 2^n product terms by using 2^n AND gates having n inputs each. We can implement these product terms by using $n \times 2^n$ decoder. So, this decoder generates ' n ' **min terms**.

Here, the inputs of OR gates are programmable. That means, we can program any number of required product terms, since all the outputs of AND gates are applied as inputs to each OR gate. Therefore, the outputs of PROM will be in the form of **sum of min terms**.

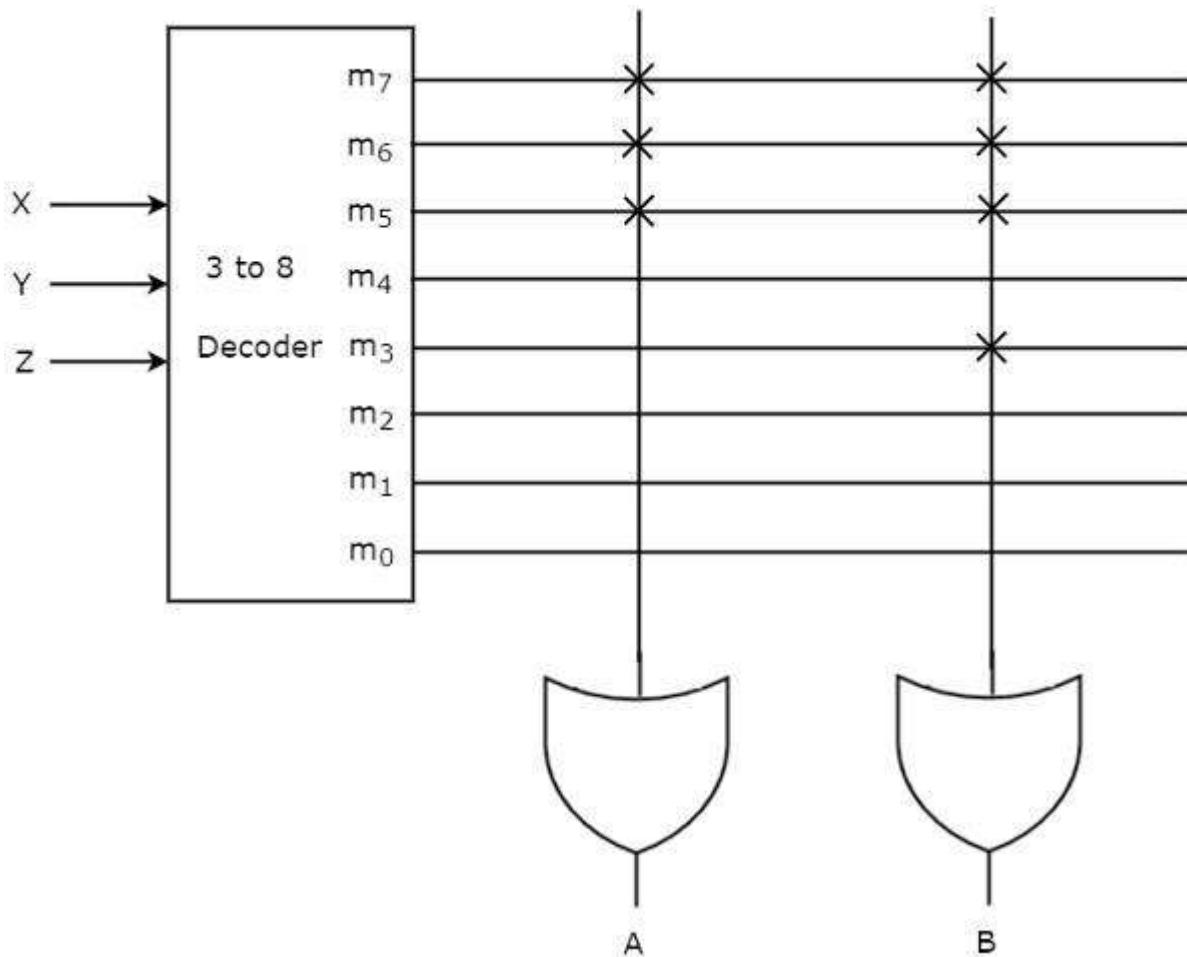
Example

Let us implement the following **Boolean functions** using PROM.

$$A(X,Y,Z) = \sum m(5,6,7) \quad A(X,Y,Z) = \sum m(5,6,7)$$

$$B(X,Y,Z) = \sum m(3,5,6,7) \quad B(X,Y,Z) = \sum m(3,5,6,7)$$

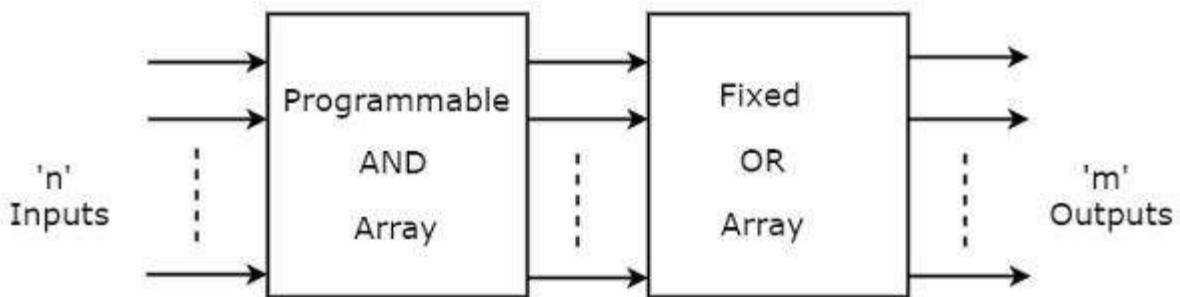
The given two functions are in sum of min terms form and each function is having three variables X, Y & Z. So, we require a 3 to 8 decoder and two programmable OR gates for producing these two functions. The corresponding **PROM** is shown in the following figure.



Here, 3 to 8 decoder generates eight min terms. The two programmable OR gates have the access of all these min terms. But, only the required min terms are programmed in order to produce the respective Boolean functions by each OR gate. The symbol 'X' is used for programmable connections.

Programmable Array Logic PAL

PAL is a programmable logic device that has Programmable AND array & fixed OR array. The advantage of PAL is that we can generate only the required product terms of Boolean function instead of generating all the min terms by using programmable AND gates. The **block diagram** of PAL is shown in the following figure.



Here, the inputs of AND gates are programmable. That means each AND gate has both normal and complemented inputs of variables. So, based on the requirement, we can program any of those inputs. So, we can generate only the required **product terms** by using these AND gates.

Here, the inputs of OR gates are not of programmable type. So, the number of inputs to each OR gate will be of fixed type. Hence, apply those required product terms to each OR gate as inputs. Therefore, the outputs of PAL will be in the form of **sum of products form**.

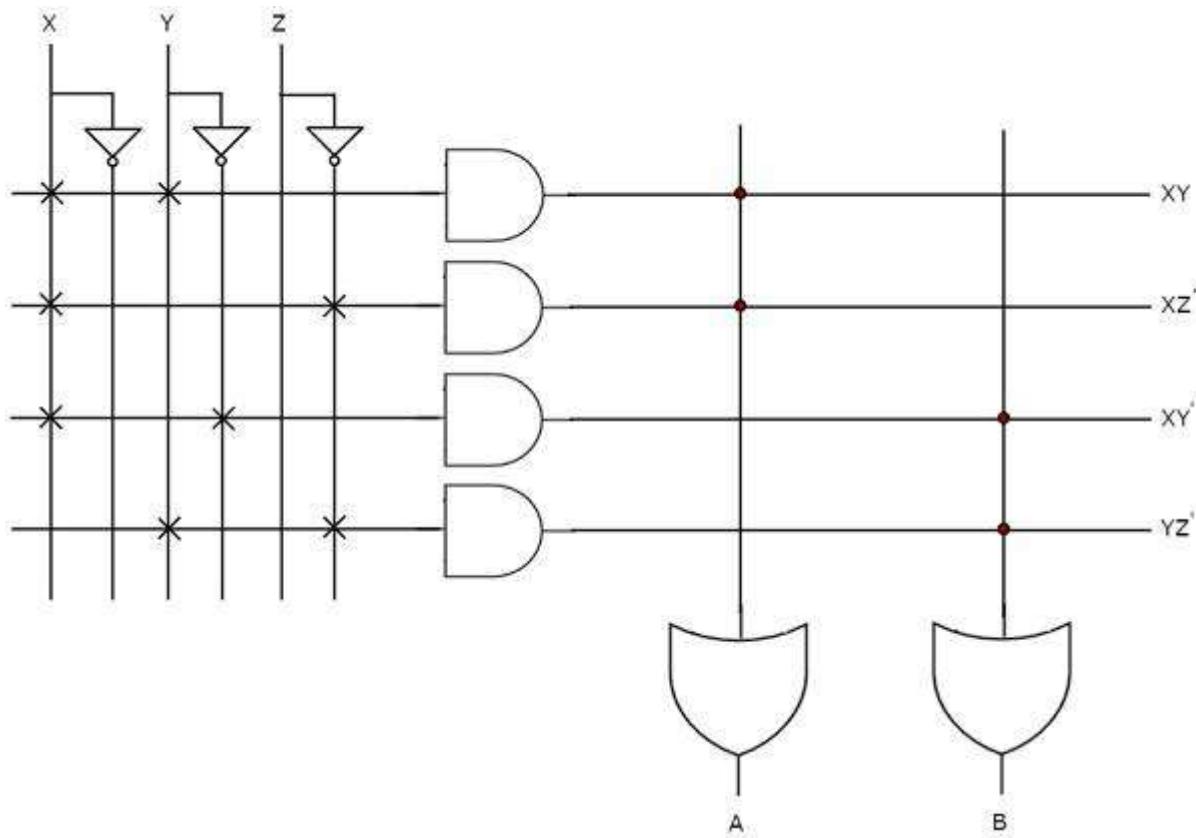
Example

Let us implement the following **Boolean functions** using PAL.

$$A = XY + XZ'$$

$$B = XY' + YZ'$$

The given two functions are in sum of products form. There are two product terms present in each Boolean function. So, we require four programmable AND gates & two fixed OR gates for producing those two functions. The corresponding **PAL** is shown in the following figure.

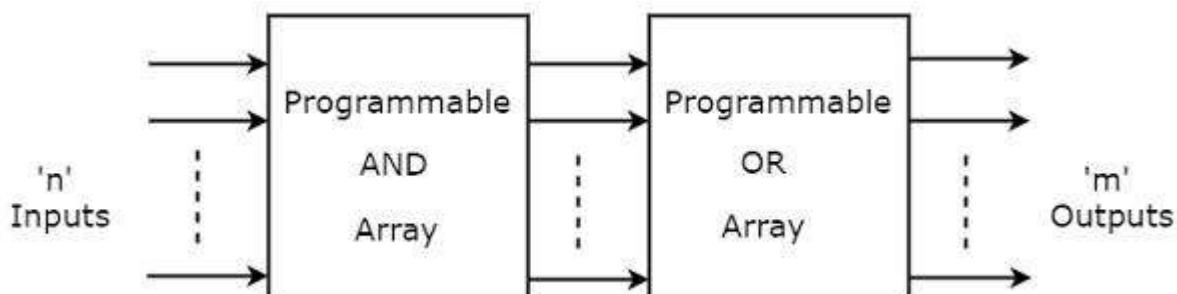


The **programmable AND gates** have the access of both normal and complemented inputs of variables. In the above figure, the inputs $X, X'X', Y, Y'Y', Z$ & $Z'Z'$, are available at the inputs of each AND gate. So, program only the required literals in order to generate one product term by each AND gate. The symbol 'X' is used for programmable connections.

Here, the inputs of OR gates are of fixed type. So, the necessary product terms are connected to inputs of each **OR gate**. So that the OR gates produce the respective Boolean functions. The symbol '.' is used for fixed connections.

Programmable Logic Array PLA

PLA is a programmable logic device that has both Programmable AND array & Programmable OR array. Hence, it is the most flexible PLD. The **block diagram** of PLA is shown in the following figure.



Here, the inputs of AND gates are programmable. That means each AND gate has both normal and complemented inputs of variables. So, based on the requirement, we can program any of those inputs. So, we can generate only the required **product terms** by using these AND gates.

Here, the inputs of OR gates are also programmable. So, we can program any number of required product terms, since all the outputs of AND gates are applied as inputs to each OR gate. Therefore, the outputs of PAL will be in the form of **sum of products form**.

Example

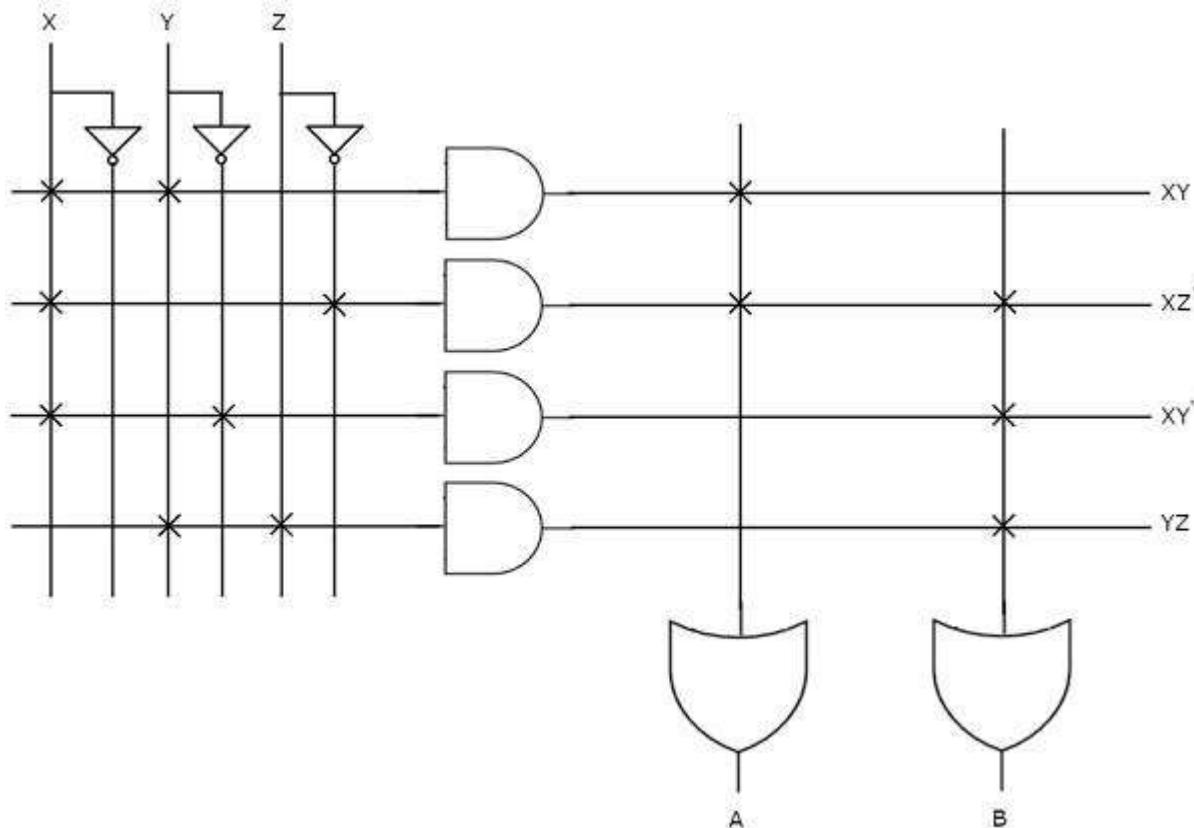
Let us implement the following **Boolean functions** using PLA.

$$A = XY + XZ'$$

$$B = XY' + YZ + XZ'$$

The given two functions are in sum of products form. The number of product terms present in the given Boolean functions A & B are two and three respectively. One product term, $Z'XZ'X$ is common in each function.

So, we require four programmable AND gates & two programmable OR gates for producing those two functions. The corresponding **PLA** is shown in the following figure.



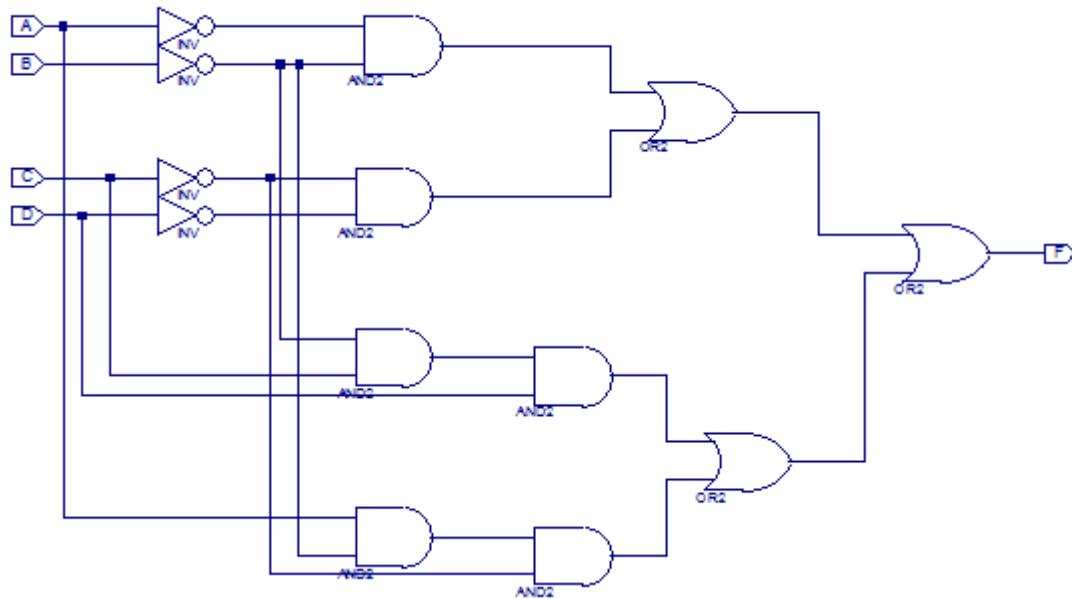
The **programmable AND gates** have the access of both normal and complemented inputs of variables. In the above figure, the inputs X, X'X', Y, Y'Y', Z & Z'Z', are available at the inputs of each AND gate. So, program only the required literals in order to generate one product term by each AND gate.

All these product terms are available at the inputs of each **programmable OR gate**. But, only program the required product terms in order to produce the respective Boolean functions by each OR gate. The symbol 'X' is used for programmable connections.

Example:

For the combinational circuit shown in figure:

- (i) Determine the minterms
- (ii) Implement output F by PLA



PROGRAMMABLE LOGIC DEVICES

Programmable logic devices (PLDs) are used for designing logic circuits. PLDs can be configured by the user to perform specific functions. The different types of PLDs available are:

- ROM and EPROM.
- Programmable Logic Arrays (PLA)
- Programmable Array Logic (PAL)
- Field Programmable Gate Arrays (FPGA)

ROM

ROM consists of an array of semiconductor devices interconnected to store an array of memory data. Data can be only read, it cannot be changed under normal operating conditions.

TYPES OF ROM.

- Mask programmable ROM
- Erasable programmable ROM (EPROM)
- Electrically erasable PROM (EEPROM)
- Flash

BASIC ROM STRUCTURE

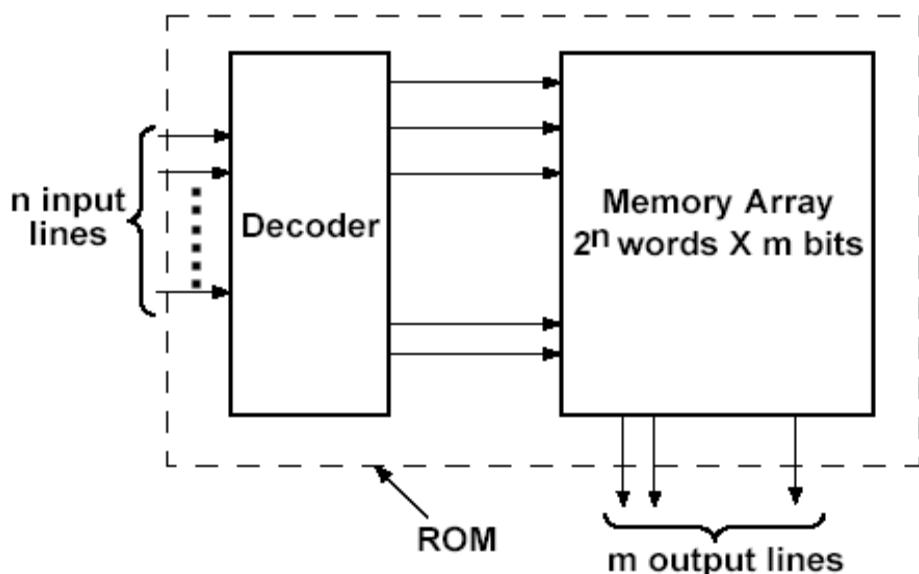


Fig.1 ROM Block diagram

Programmable Logic Array

A programmable logic array (PLA) performs the same basic function as a ROM. It is the most flexible device in the family of PLDs. The internal organization of a PLA is different from that of the ROM. The decoder of the ROM is replaced with an AND array that realizes selected product terms of the input variables. The AND array is followed by an OR array which ORs together the product terms needed to form the output functions. Both the AND and the OR arrays are programmable giving a lot of flexibility for implementing logic design.

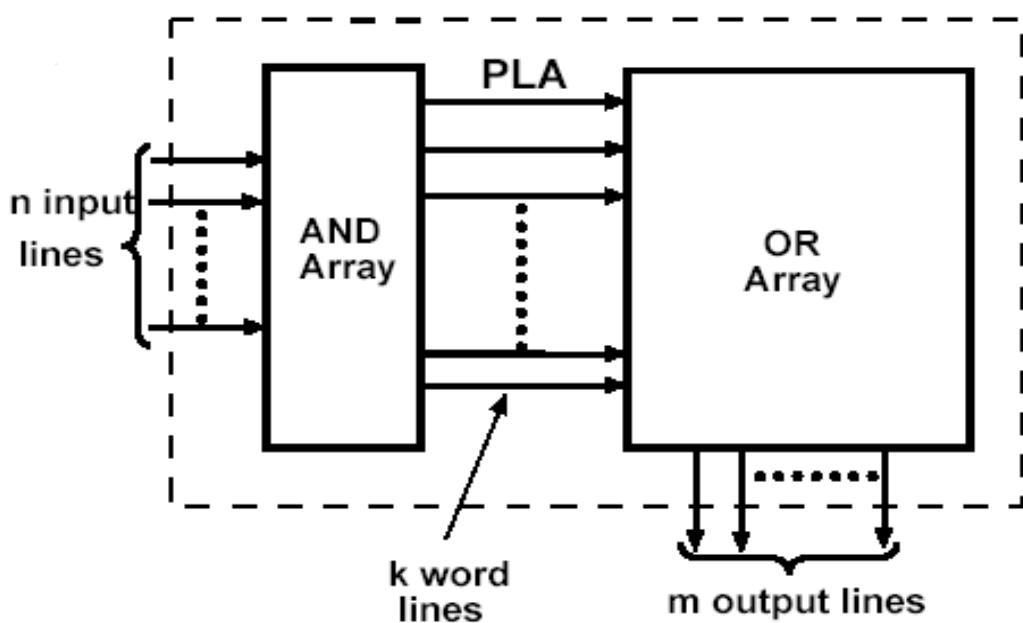


Fig 8 – Internal Logic Diagram of a PLA

Internally the PLA uses NOR-NOR logic but the added input and output inverting buffers make it equivalent to AND-OR logic. Logic gates are formed in the array by connecting NMOS switching transistors between the column line and row line. The transistors act as switches, so if the gate input is a logic zero, the transistor is turned off

whereas if the gate input is a logic one, the transistor provides a conducting path to ground.

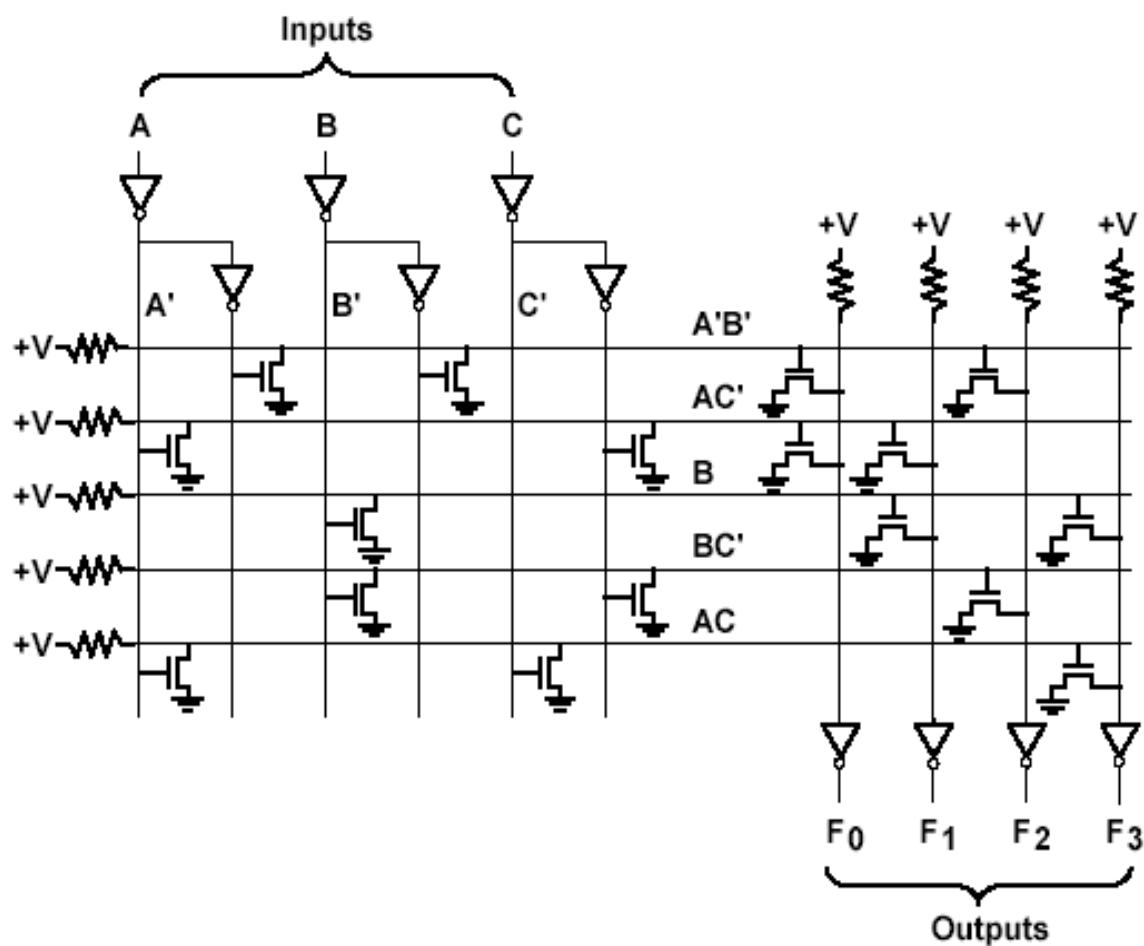


Fig 9 –PLA with 3 inputs , 5 product terms and 4 outputs

$$F_0 = \sum m(0,1,4,6) = A'B' + AC'$$

$$F_1 = \sum m(2,3,4,6,7) = B + AC'$$

$$F_2 = \sum m(0,1,2,6) = A'B' + BC'$$

$$F_3 = \sum m(2,3,5,6,7) = AC + B$$

The above set of formulas are implemented using NOR-NOR logic of PLA as shown in Fig. 9 by placing the NMOS switching transistors wherever the connection has to be established. The same set of equations can be implemented using an AND-OR array equivalent as shown in Fig. 10.

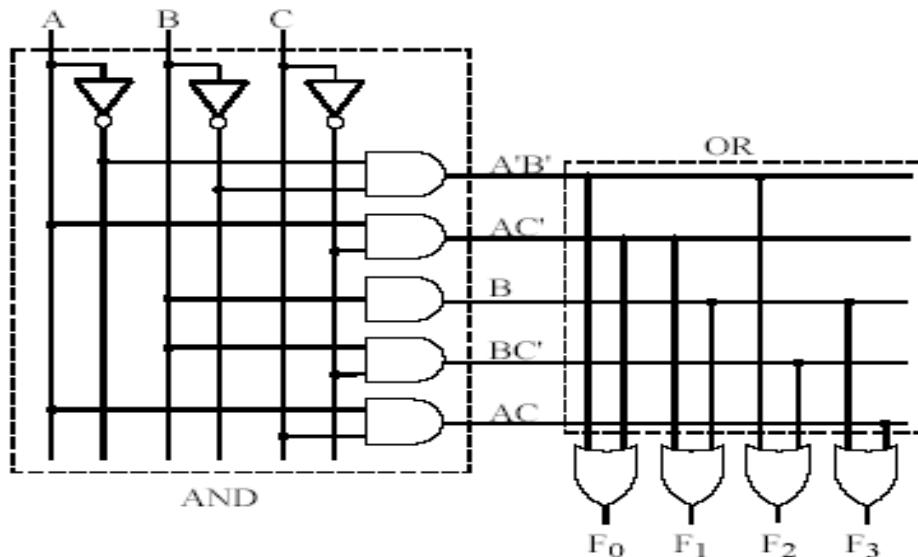


Fig 10 –AND-OR array equivalent of Fig. 9

The contents of a PLA can be specified by a modified truth table as shown in Fig.11. The input side of the table specifies the product terms . The symbols 0,1 and – indicate whether a variable is complemented, not complemented or not present in the corresponding product term. The output side of the table specifies which product terms

appear in which output function. A 1 or 0 in the output terms indicate whether a given product term is present or not present in the corresponding output function.

Product	Inputs			Outputs		
	A	B	C	F0	F1	F2
A'B'	0	0	-	1	0	1
AC'	1	-	0	1	1	0
B	-	1	-	0	1	0
BC'	-	1	0	0	0	1
AC	1	-	1	0	0	0

fig 11 – PLA Table for Fig. 10

The first row of the table indicates that the term A'B' is present in output functions F0 and F2. The second row indicates that AC' is present in F0 and F1. This PLA table can be written directly using the given set of equations to be realized using PLA logic.

Realization of a given function using min number of rows in the PLA

$$F_1 = \sum m(2,3,5,7,8,9,10,11,13,15) \quad \dots (1)$$

$$F_2 = \sum m(2,3,5,6,7,10,11,14,15) \quad \dots (2)$$

$$F_3 = \sum m(6,7,8,9,13,14,15) \quad \dots (3)$$

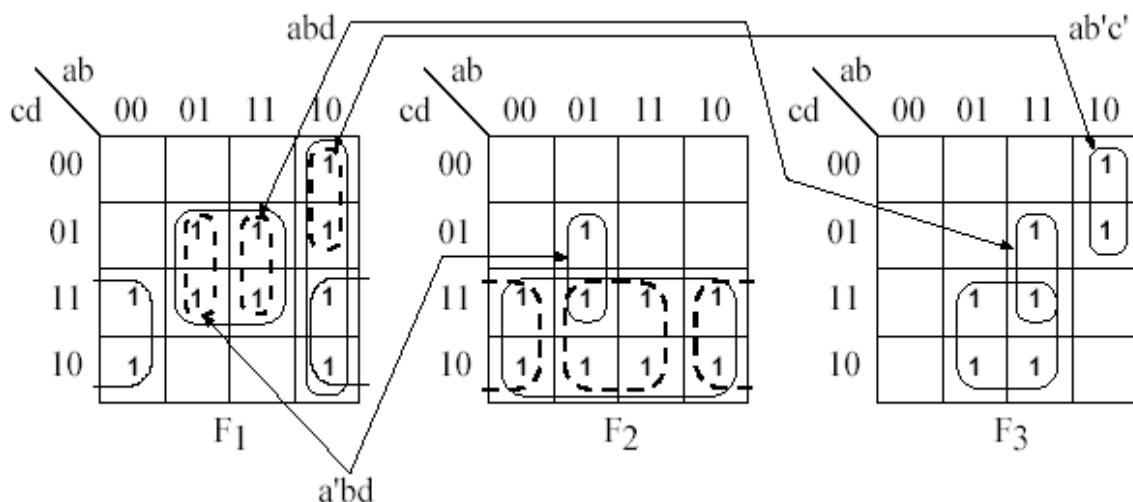


Fig 12 – Multiple Output Karnaugh Map

$$F_1 = BD + B'C + AB' \quad \dots(4)$$

$$F_2 = C + A'BD \quad \dots(5)$$

$$F_3 = BC + AB'C' + ABD \quad \dots(6)$$

Equations 1,2 and 3 can be reduced to equations 4, 5 and 6 respectively using Karnaugh map as shown in Fig.12.

If we implement these reduced equations 4,5 and 6 in a PLA then a total of 8 different product terms (including C) are required. So instead of minimizing each function separately, we have to minimize the total number of rows in the PLA table. When we are trying to design a logic using PLA, the number of terms in each equation is not important since the size of the PLA does not depend on the number of terms. The term $AB'C'$ is already present in function F3. So we can use it in F1 instead of AB' by writing AB' as $AB'(C+C')$. F1 can be now written as

$$\begin{aligned} F_1 &= BD + B'C + AB'(C+C') \\ &= BD + B'C + AB'C + A'B'C' \\ &= BD + B'C(1+A) + A'B'C' \\ &= BD + B'C + A'B'C' \end{aligned}$$

This simplification of F1 eliminates the need to use a separate row for the original term AB' present in F1 of equation (4).

Since the terms $A'BD$ and ABD are needed in F2 and F3 respectively, we can replace the term BD in F1 with $A'BD + ABD$. This eliminates the need for a row to implement the term BD in PLA. Similarly, since $B'C$ and BC are used in F1 and F3 respectively, we can replace C in F3 with $B'C + BC$. Now the equations for F1, F2 and F3 with the above said changes can be written as :

$$F_1 = BD(A+A') + B'C + AB'(C+C')$$

$$= ABD + A'BD + B'C + AB'C' \quad \dots(7)$$

$$\begin{aligned} F2 &= C(B+B') + A'BD \\ &= BC + B'C + A'BD \end{aligned} \quad \dots(8)$$

$$F3 = BC + AB'C' + ABD \quad \dots(9)$$

The current equations for F1, F2 and F3 as shown in equations 7,8 and 9 respectively have only 5 different product terms. So the PLA table can now be written with only 5 rows. This is a significant improvement over the equations 4, 5 and 6, which resulted in 8 product terms. The reduced PLA table corresponding to equations 7, 8 and 9 is as shown in the figure 13.

a	b	c	d	F1	F2	F3
0	1	-	1	1	1	0
1	1	-	1	1	0	1
1	0	0	-	1	0	1
-	0	1	-	1	1	0
-	1	1	-	0	1	1

Fig 13 – Reduced PLA table

PLA table is significantly different from that of ROM truth table. In a truth table, each row represents a minterm ; therefore, one row will be exactly selected by each combination of input values. The 0s and 1s of the output portion of the selected row determine the corresponding output values. On the other hand, each row in a PLA table represents a general product term. Therefore, 0, 1 or more rows may be selected by each combination of input values. To determine the value of F for a given input combination, the values of F in the selected rows of the PLA table must be ORed together. For example, if **abcd=0001** is given as input , no rows are selected as this combination does not exist in the PLA table; and all the F outputs are 0. If abcd = 1001, only the third row is selected resulting in F1F2F3 = 101. If **abcd = 0111**, the first and the fifth rows are selected. Therefore, to get the values for F1, F2 and F3 is got by ORing the respective values of F1, F2 and F3 in the corresponding rows resulting in F1 = 1 + 0 = 1, F2 = 1+1 = 1 and F3 = 0 + 1 = 1.

PLA Realization of Equations

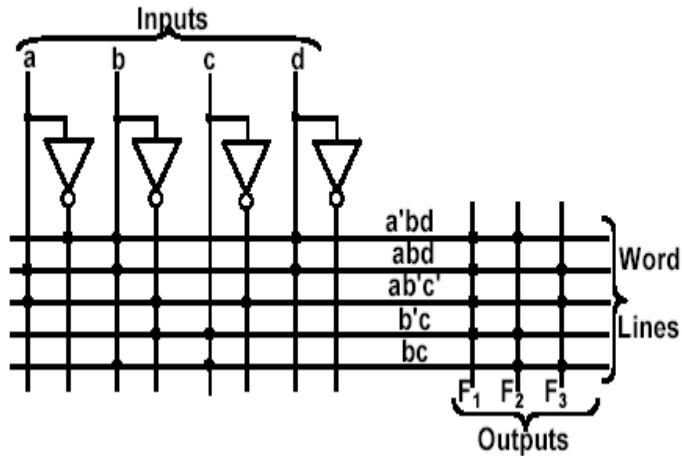


Fig 14–PLA Realization of Equations 7, 8 and 9.

Fig 14 shows the PLA structure, which has four inputs, five product terms and three outputs as shown in equation 7,8 and 9. A dot at the intersection of word line and an input or output line indicates the presence of a switching element in the array.

Implementation of BCD to Excess-3 using PLA

Product Term	Q1	Q2	Q3	X	Q1 ⁺	Q2 ⁺	Q3 ⁺	Z
Q2'	–	0	–	–	1	0	0	0
Q1	1	–	–	–	0	1	0	0
Q1Q2Q3	1	1	1	–	0	0	1	0
Q1Q3'X'	1	–	0	0	0	0	1	0
Q1'Q2'X	0	0	–	1	0	0	1	0
Q3'X'	–	–	0	0	0	0	0	1
Q3X	–	–	1	1	0	0	0	1

Fig 15 –PLA Table for BCD to Excess-3 Converter

$$\begin{aligned}
 Q1^+ &= Q2' \\
 Q2^+ &= Q1 \\
 Q3^+ &= Q1Q2 Q3 + X'Q1Q3' + XQ1'Q2' \\
 Z &= X'Q3' + XQ3
 \end{aligned}$$

Fig16 –Reduced Equations for BCD to excess-3

We can realize the sequential machine for BCD to excess 3 using a PLA and three D f/f. The network structure is same as that realized with ROM, except that the ROM is replaced by PLA. The required PLA table is as shown in figure 15 is derived from equations shown in figure 16 for BCD to excess-3.

Reading the output of the PLA in VHDL is somewhat more difficult than reading the ROM output. Since the input to the PLA can match several rows, and the output from those rows must be ORed together. The realization of the equations shown in fig 16 is as shown in Fig 17.

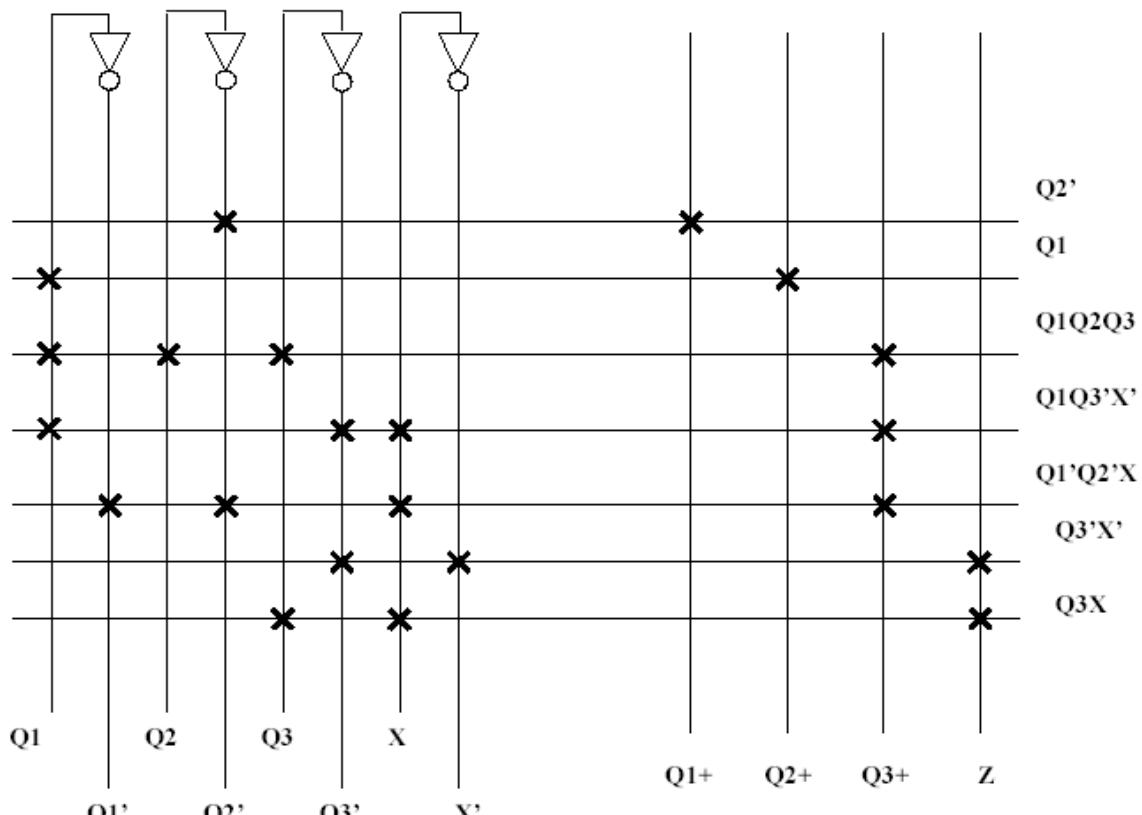


Fig : 17 Realization of BCD to Excess-3 using PLA.



School of Computer Science and Engineering

Fall Semester 2023-2024

Model QP Continuous Assessment Test – 1

Programme Name &Branch: B. Tech. Computer Science and Engineering

Course Name & code: BECE102L - Digital Systems Design

Faculty Name (s): Deepika Rani Sona

Exam Duration: 90 Min.

Maximum Marks: 50

Q.No.	Question	Max Marks	CO	BL
1.	a. Simplify the Following using Boolean Algebra $[(\bar{a} + \bar{d} + \bar{b}\bar{c})(b + d + \bar{a}\bar{c})] + \bar{b}\bar{c}\bar{d} + \bar{a}\bar{c}\bar{d}$ (reduce to three terms) b. Simplify each of the following expressions using only the consensus theorem (or its dual): $BC'D' + ABC' + AC'D + AB'D + A'BD'$ (reduce to three terms)	5	CO2	BL3
2.	Simplify the following using K – Map and design the hardware circuit using (i) only NAND gates (ii) only NOR gates. $F = \sum m(0,3,5,6,9,10,12,15) + d(1,7,8,14)$	10	CO2	BL2
3.	Design a combinational circuit using Full Adder, which has three inputs, x, y, and z, and three outputs, A, B, and C. When the binary input is 0, 1, 2, or 3, the binary output is one greater than the input. When the binary input is 4, 5, 6, or 7, the binary output is one less than the input.	10	CO3	BL6
4.	Design a 4 bit Code converter to convert 84-2 -1 to excess – 2 Code	10	CO1	BL3
5.	Simplify the following Boolean function with the help of four variables K-map $y(a,b,c,d) = \sum m(0,2,5,7,8,10,13) + d(14,15)$ <ul style="list-style-type: none"> (a) How many quad/four cell groups and what are they? (b) How many pair/two cell groups and what are they? (c) Simplified Boolean expression 			
6.	Let A, B and C are inputs and Y is a output.			

(Note: Type the answer. For typing the answer use dot, plus and single quote operators for AND, OR and NOT gate respectively.)

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

- (a) Boolean expression in Sum of product (SOP). (2 Mark)
- (b) Boolean expression in canonical form using designations in Product of sum (POS) for the given truth table (2 Mark)
- (c) Simplify the Boolean expression to four literals. (2 Mark)
- (d) Suppose the simplified Boolean is designed only with 2-input NAND gates, how many 2-input NAND gates are (minimum) needed? (4 Mark)



VIT[®]

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

BECE 102L - CAT1 - KEY

ADDITIONAL SHEET

Initial of the Hall Supervisor with Date
Emp. ID :

$$D) F = \prod M(0, 2, 4, 6, 9, 11, 13, 15)$$

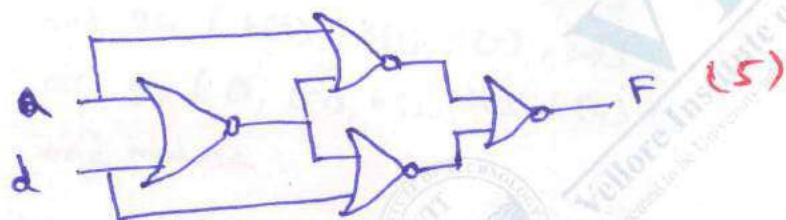
ab\cd	00	01	11	10
00	1	1		
01	1	1		
11		1		
10		1		1

$$F' = (a+d)(\bar{a}+\bar{d})$$

$$= ad + a\bar{d} + \bar{a}\bar{d} + \bar{a}d$$

$$= a\bar{d} + \bar{a}d \quad (2)$$

$$F = \bar{a}\bar{d} + ad = a \oplus d$$



$$2) a) F = \sum m(0, 2, 5, 7, 8, 10, 13, 15)$$

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

$$\text{Minimized SOP} = \bar{b}\bar{d} + bd$$

$$= b \cancel{\oplus} d \cancel{(\oplus)}$$

$$= b \oplus d \quad (1)$$

$$\overline{F'} = (\bar{b}+d)(b+\bar{d})$$

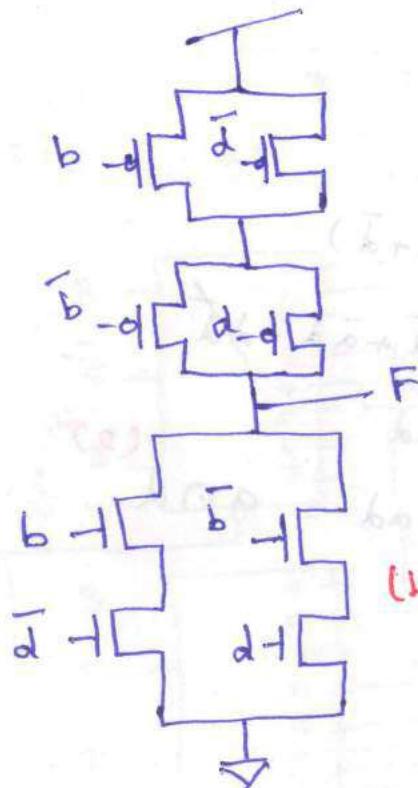
$$= \bar{b}b + \bar{b}\bar{d} + bd + d\bar{d}$$

$$= \bar{b}\bar{d} + bd$$

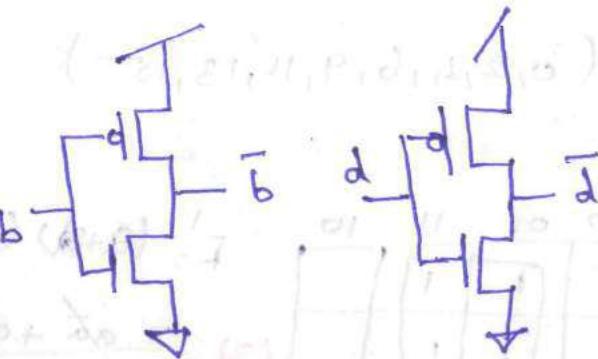
$$F = b \oplus d. \quad (1)$$

$$b) F = \overline{\overline{bd} + bd}$$

$$F = \overline{\overline{bd} + bd} = \overline{\overline{bd} + \overline{bd}}$$



(1p)



(1)

3(a)

$$c) a^nb = 4'b \times 111 \quad (1\frac{1}{2})$$

$$d) a \& b = 1 \quad (1)$$

$$e) \{ \{ a[3:2], \{ 2 \{ b[1] \} \} \} \} = 1000 \quad (1\frac{1}{2})$$

$$f) na = 4'b 0101 \quad (1\frac{1}{2})$$

$$g) a^n b = 4'b \times 0000 \quad (1\frac{1}{2})$$

$$h) (a != b) = 1 \quad (1\frac{1}{2})$$

3b

module Sample (O, S, w, x, y, z);

input [1:0] S;

input w, x, y, z;

~~Output reg~~

output O;

wire notso, notsi;

wire [3:0] t;

not g1 (notso, S[0]);

not g2 (notsi, S[1]);

and g3 (t[0], notso, notsi, w);

and g4 (t[1], S[0], notso, x);

and g5 (t[2], S[1], notso, y);

and g6 (t[3], S[1], S[0], z);

or g7 (0, t[1], t[2], t[3], t[4]);

end module.

S ₁	S ₀	O
0	0	w
0	1	x
1	0	y
1	1	z

(4)

Testbench

module Sample_tb();

reg [1:0] S;

reg w, x, y, z;

wire O;

Sample dut (O, S, w, x, y, z);

initial

begin

w = 1'b0; S[1] = 1'b0; S[0] = 1'b[0]; x = 1'b0; y = 1'b0; z = 1'b0;

#10

w = 1'b1; S[1] = 1'b0; S[0] = 1'b[0]; x = 1'b0; y = 1'b0; z = 1'b0;

#10

x = 1'b0; S[1] = 1'b0; S[0] = 1'b[3]; w = 1'b0; y = 1'b0; z = 1'b0;

\$stop;

end

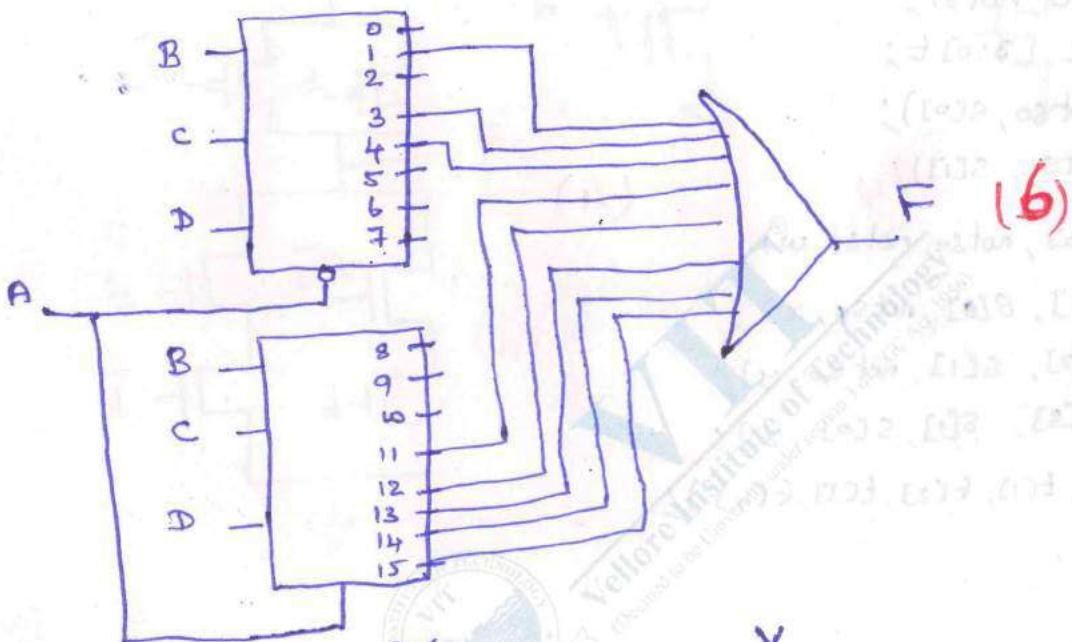
(3)

endmodule

4) F

D	I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇
0	0	0	1	0	0	0	1	1
1	1	1	0	0	0	1	1	1

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



AB	CD	00	01	11	10	AB
00	00	1				
01	01		1			
11	11			1		
10	10	1				

X	CD
1	00
1	01
1	11
1	10

$$Y = A \oplus B \oplus C \quad (2)$$

X	CD
1	00
1	01
1	11
1	10

$$X = \bar{A}B + \bar{B}A \quad (2)$$

AB	CD
11	00
11	01
11	11
11	10

$$W = A + \bar{A} \quad (1)$$

module CKT (A, B, C, D, W, X, Y, Z);

input A, B, C, D;

output W, X, Y, Z;

assign W = A, X = A' B, Y = A' B' C, Z = A' B' C' D;

endmodule. (4)



Exam : QUIZ 1 /B1

Discipline: B.Tech. CSE		Semester: FALL 2023-24	
Subject : Digital Systems Design		Full Marks: 10	
Name and REG NO. -		Time : 10Minutes	
1	What is system task to suspend simulation a) \$finish b) \$minitor c) \$display d) \$stop	7	How many inputs will a decimal-to-BCD encoder require? a) 4 b) 8 c) 10 d) 16
2	Which of the following options represent the correct reduction of $x\bar{y}z + \bar{x}\bar{y}z$ a) 0 b) $\bar{y}z$ c) $x + \bar{x}$ b) $2\bar{y}z$	8	What is a parity bit? a) An error detection is achieved by adding an extra bit b) After addition, the carry is found c) Bit generated during data transmission d) After addition, the total number of bits
3	How many AND, OR and EXOR gates are required for the configuration of full adder? a) 1, 2, 2 b) 2, 1, 2 c) 3, 1, 2 d) 4, 0, 1	9	Which is legal negative number a) 4'd-3 b) 6'-d3 c) -6d'3 d) None
4	The expression for Absorption law is given by _____ a) $A + AB = A$ b) $A + AB = B$ c) $AB + AA' = A$ d) $A + B = B + A$	10	In continuous assignment left hand side must be a) Net b) Reg c) Scalar or Vector Net d) Scalar or Vector reg

5	<p>The canonical sum of product form of the function $y(A,B) = A + B$ is</p> <hr/> <p>a) $AB + BB + A'A$ b) $AB + AB' + A'B$ c) $BA + BA' + A'B'$ d) $AB' + A'B + A'B'$</p>		
6	<p>Operator which precedes the operand</p> <p>a) Unary b) Binary c) Ternary d) None</p>		