

NUMBER-BASE CONVERSIONS

Number Systems

A **number system** is defined as a **system** of writing for expressing **numbers**. It is the mathematical notation for representing **numbers** of a given set by using digits or other symbols in a consistent manner.

■ Decimal Number System

- Uses 10 digits from 0 to 9= { 0, 1, 2, 3, ..., 9 }
- Example-35,64,135,2345 etc

■ Binary Number System

- Uses two digits, 0 and 1.
- Also called base 2 number system
- Example-100,011,101,1100,10 etc

Number Systems

■ Octal Number System:

- Uses eight digits, 0,1,2,3,4,5,6,7.
- Also called base 8 number system
- Example-35,64 ,71,135 etc

■ Hexadecimal Number System:

- Uses 10 digits and 6 letters, 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.
- Letters represents numbers starting from 10.
 $A = 10, B = 11, C = 12, D = 13, E = 14, F = 15.$
- Also called base 16 number system.
- Example-135,782,18D,6FC etc

Decimal Number System

- Decimal number system, symbols = { 0, 1, 2, 3, ..., 9 }
- Example: $(7594)_{10} = (7 \times 10^3) + (5 \times 10^2) + (9 \times 10^1) + (4 \times 10^0)$

A decimal number such as 7,594 represents a quantity equal to 7 thousands, plus 5 hundreds, plus 9 tens, plus 4 units. The thousands, hundreds, etc., are powers of 10 implied by the position of the coefficients (symbols) in the number.
- In general, $(a_n a_{n-1} \dots a_0)_{10} = (a_n \times 10^n) + (a_{n-1} \times 10^{n-1}) + \dots + (a_0 \times 10^0)$
- $(2.75)_{10} = (2 \times 10^0) + (7 \times 10^{-1}) + (5 \times 10^{-2})$
- In general, $(a_n a_{n-1} \dots a_0 . f_1 f_2 \dots f_m)_{10} = (a_n \times 10^n) + (a_{n-1} \times 10^{n-1}) + \dots + (a_0 \times 10^0) + (f_1 \times 10^{-1}) + (f_2 \times 10^{-2}) + \dots + (f_m \times 10^{-m})$

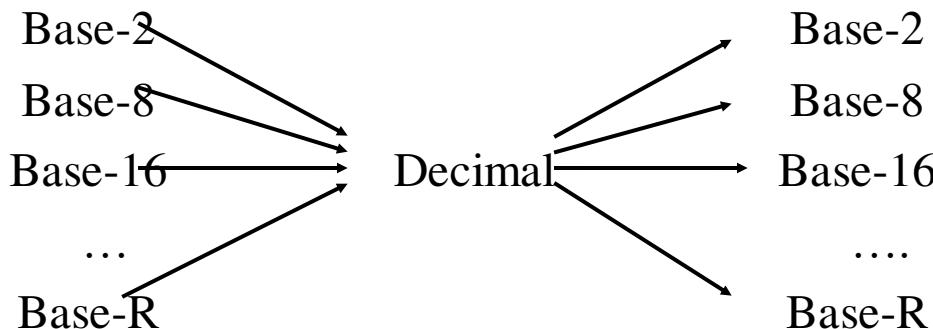
NUMBER-BASE CONVERSIONS

- Representations of a number in a different radix are said to be equivalent if they have the same decimal representation. For example, $(0011)_8$ and $(1001)_2$ are equivalent—both have decimal value 9.
- The conversion of a number in base r to decimal is done by expanding the number in a power series and adding all the terms as shown previously.
- We now present a general procedure for the reverse operation of converting a decimal number to a number in base r .
- If the number includes a radix point, it is necessary to separate the number into an integer part and a fraction part, since each part must be converted differently.
- The conversion of a decimal integer to a number in base r is done by dividing the number and all successive quotients by r and accumulating the remainders.

Conversion between Decimal and other Bases

■ Decimal to base-R

- ❖ whole numbers: repeated division-by-R
- ❖ fractions: repeated multiplication-by-R



Decimal-to-Binary Conversion

- ❖ *Repeated Division-by-2 Method* (for whole numbers)
- ❖ *Repeated Multiplication-by-2 Method* (for fractions)

Repeated Division-by-2 Method

- To convert a whole number to binary, use successive division by 2 until the quotient is 0. The remainders form the answer, with the first remainder as the *least significant bit (LSB)* and the last as the *most significant bit (MSB)*.

$$(43)_{10} = (101011)_2$$

2	43		
2	21	rem 1	← LSB
2	10	rem 1	
2	5	rem 0	
2	2	rem 1	
2	1	rem 0	
	0	rem 1	← MSB

Decimal to Binary Conversion

Divide by 2 Process

Decimal #

$$13 \div 2 = 6 \text{ remainder } 1$$

$$6 \div 2 = 3 \text{ remainder } 0$$

$$3 \div 2 = 1 \text{ remainder } 1$$

$$1 \div 2 = 0 \text{ remainder } 1$$

Divide-by-2 Process
Stops When
Quotient Reaches 0

1 1 0 1

Repeated Multiplication-by-2 Method

To convert decimal fractions to binary, repeated multiplication by 2 is used.

The process is continued until the fraction becomes 0 or until the number of digits has sufficient accuracy.

The coefficients of the binary number are obtained from the integers as follows: The first integer is written as the MSB, and the last as the LSB.

$$(0.3125)_{10} = (.0101)_2$$

Coefficient
$0.3125 \times 2 = 0.625$
$0.625 \times 2 = 1.25$
$0.25 \times 2 = 0.50$
$0.5 \times 2 = 1.00$

Decimal-to-Binary Conversion

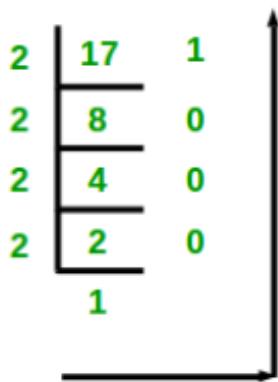
Convert $(0.6875)_{10}$ to Binary

	<u>integer</u>	<u>fraction</u>	<u>coefficient</u>
$0.6875 \times 2 =$	1	+	$a_{-1} = 1$
$0.3750 \times 2 =$	0	+	$a_{-2} = 0$
$0.7500 \times 2 =$	1	+	$a_{-3} = 1$
$0.5000 \times 2 =$	1	+	$a_{-4} = 1$

$$(0.6875)_{10} = (0.a_{-1}a_{-2}a_{-3}a_{-4})_2 = (0.1011)_2$$

Decimal to Binary Conversion of 17.65

Decimal number : 17



Binary number: 10001

$$0.65 * 2 = 1.3 \longrightarrow 1$$

$$0.3 * 2 = 0.6 \longrightarrow 0$$

$$0.6 * 2 = 1.2 \longrightarrow 1$$

$$0.2 * 2 = 0.4 \longrightarrow 0$$

$$0.4 * 2 = 0.8 \longrightarrow 0$$

$$(17.65)_{10} = (10001.10100)_2$$

Practice

Convert the following decimal numbers into binary:

Decimal $(11)_{10}$

Decimal $(4)_{10}$

Decimal $(17)_{10}$

Decimal $(47.2)_{10}$

Binary to Decimal Conversion

$$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}$$

$$(1010.011)_2 = (?)_{10}$$

Place Value

- *Example - Place value in binary system:*

	2^3	2^2	2^1	2^0
Place Value	8	4	2	1
Binary Number	1	1	0	0

RESULT: $(1100)_2 = \text{decimal } 8 + 4 + 0 + 0 = (12)_{10}$

Binary to Decimal Conversion

**Convert Binary Number 110011
to a Decimal Number:**

	2^5	2^4	2^3	2^2	2^1	2^0
Binary	1	1	0	0	1	1
	↓	↓	↓	↓	↓	↓

Decimal $32 + 16 + 0 + 0 + 2 + 1 = 51$

1-Bit Binary Numbers	2-Bit Binary Numbers	3-Bit Binary Numbers	4-Bit Binary Numbers	Decimal Equivalents
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

Hexadecimal Number System

Uses 16 symbols - Base 16 System
0-9, A, B, C, D, E, F

<u>Decimal</u>	<u>Binary</u>	<u>Hexadecimal</u>
1	0001	1
9	1001	9
10	1010	A
15	1111	F
16	10000	10

Decimal to Hexadecimal Conversion

Divide by 16 Process

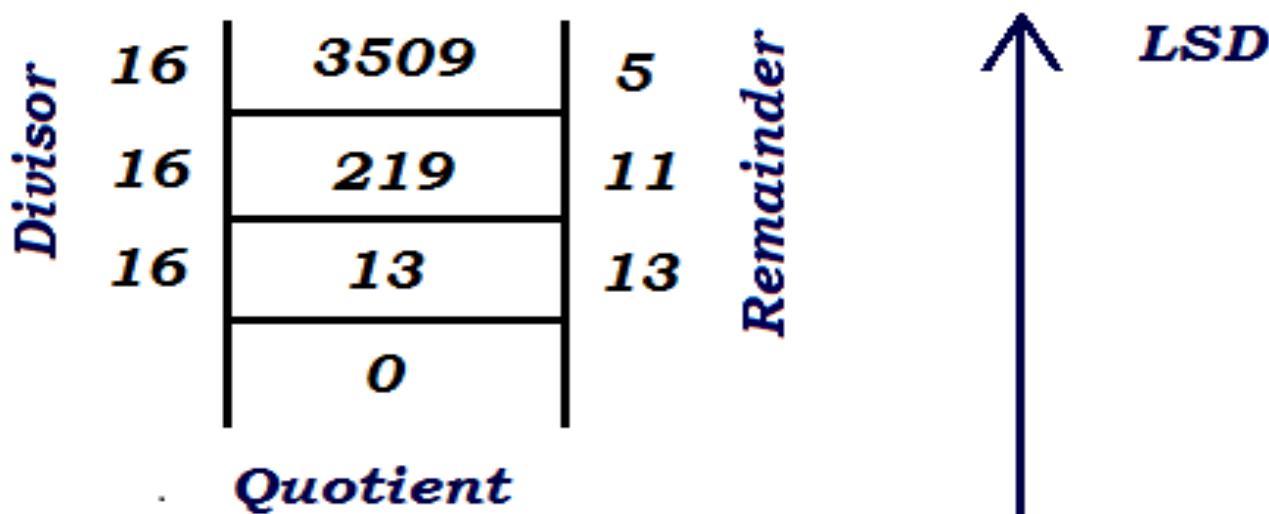
Decimal # $47 \div 16 = 2$ remainder 15

Divide-by-16 Process
Stops When
Quotient Reaches 0

0 remainder 2

2 F

Find the Hex equivalent for the Decimal 3509



MSD - most significant digit LSD - least significant digit

*For Hex value 13 = D, 11 = B & 5 = 5
Therefore, the equivalent Hex
number for decimal 3509 is DB5*

Hexadecimal to Decimal Conversion

Convert hexadecimal number **2DB**
to a decimal number

Place Value	256	16	1
	16^2	16^1	16^0

Hexadecimal	2	D	B
-------------	---	---	---

$$(256 \times 2) \quad (16 \times 13) \quad (1 \times 11)$$

$$\text{Decimal} \quad 512 \quad + \quad 208 \quad + \quad 11 = 731$$

$$2ED_{16} = 2 \times 16^2 + E \times 16^1 + D \times 16^0 = 749_{10}$$

two
two hundred
fifty six's

fourteen
sixteens

thirteen
ones

Practice

Convert Hexadecimal number A6 to Binary

$$A6 = \boxed{1010\ 0110 \text{ (Binary)}}$$

Convert Hexadecimal number 16 to Decimal

$$16 = \boxed{22 \text{ (Decimal)}}$$

Convert Decimal 63 to Hexadecimal

$$63 = \boxed{3F \text{ (Hexadecimal)}}$$

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Octal Numbers

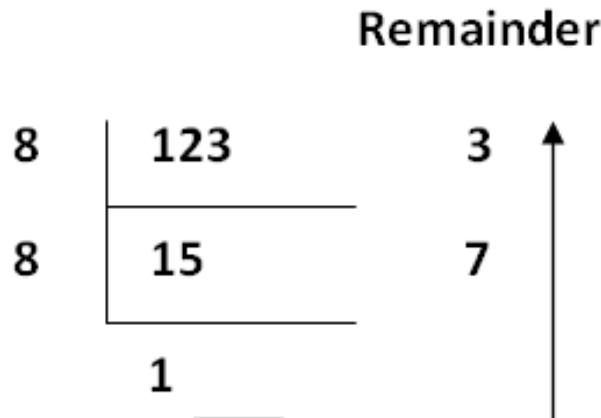
Uses 8 symbols - Base 8 System

0, 1, 2, 3, 4, 5, 6, 7

<u>Decimal</u>	<u>Binary</u>	<u>Octal</u>
1	001	1
6	110	6
7	111	7
8	001 000	10
9	001 001	11

Octal \leftrightarrow Decimal conversion

To Convert Decimal To Octal



$$123_{10} = 173_8$$

Octal Number to Decimal

$$2754_8 = \begin{array}{r} 2 \times 8^3 \\ 7 \times 8^2 \\ 5 \times 8^1 \\ 4 \times 8^0 \end{array} \rightarrow 1024 \quad 448 \quad 40 \quad 4$$

1516_{10}

$$(630.4)_8 = 6 \times 8^2 + 3 \times 8 + 4 \times 8^{-1} = (408.5)_{10}$$

Decimal to Octal Conversion

Convert $(0.513)_{10}$ to octal.

$$0.513 \times 8 = 4.104$$

$$0.104 \times 8 = 0.832$$

$$0.832 \times 8 = 6.656$$

$$0.656 \times 8 = 5.248$$

$$0.248 \times 8 = 1.984$$

$$0.984 \times 8 = 7.872$$

$$(0.513)_{10} = (0.406517\ldots)_8$$

Binary-Octal/Hexadecimal Conversion

- Binary → Octal: Partition in groups of 3

$$(10\text{ }111\text{ }011\text{ }001 . 101\text{ }110)_2 = (2731.56)_8$$

- Octal → Binary: reverse

$$(2731.56)_8 = (10\text{ }111\text{ }011\text{ }001 . 101\text{ }110)_2$$

- Binary → Hexadecimal: Partition in groups of 4

$$(101\text{ }1101\text{ }1001 . 1011\text{ }1000)_2 = (5D9.B8)_{16}$$

- Hexadecimal → Binary: reverse

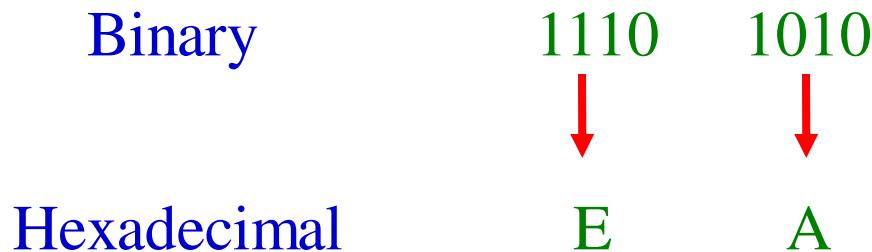
$$(5D9.B8)_{16} = (101\text{ }1101\text{ }1001 . 1011\text{ }1000)_2$$

Hexadecimal and Binary Conversions

- Hexadecimal to Binary Conversion



- Binary to Hexadecimal Conversion



Binary-Octal/Hexadecimal Conversion

$$(\underline{\underline{10}}\underline{\underline{110}}\underline{\underline{001}}\underline{\underline{101}}\underline{\underline{011}} \cdot \underline{\underline{111}}\underline{\underline{100}}\underline{\underline{000}}\underline{\underline{110}})_2 = (26153.7406)_8$$

2 6 1 5 3 7 4 0 6

$$(\underline{\underline{10}}\underline{\underline{1100}}\underline{\underline{0110}}\underline{\underline{1011}} \cdot \underline{\underline{1111}}\underline{\underline{0010}})_2 = (2C6B.F2)_{16}$$

2 C 6 B F 2

$$(673.124)_8 = (\underline{\underline{110}}\underline{\underline{111}}\underline{\underline{011}} \cdot \underline{\underline{001}}\underline{\underline{010}}\underline{\underline{100}})_2$$

6 7 3 1 2 4

$$(306. D)_{16} = (\underline{\underline{0011}}\underline{\underline{0000}}\underline{\underline{0110}} \cdot \underline{\underline{1101}})_2$$

3 0 6 D

Base-R to Decimal Conversion

$$\begin{aligned}\blacktriangleright (1101.101)_2 &= 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-3} \\ &= 8 + 4 + 1 + 0.5 + 0.125 \\ &= (13.625)_{10}\end{aligned}$$

$$\begin{aligned}\blacktriangleright (572.6)_8 &= 5 \times 8^2 + 7 \times 8^1 + 2 \times 8^0 + 6 \times 8^{-1} \\ &= 320 + 56 + 2 + 0.75 = (378.75)_{10}\end{aligned}$$

$$\begin{aligned}\blacktriangleright (2A.8)_{16} &= 2 \times 16^1 + 10 \times 16^0 + 8 \times 16^{-1} \\ &= 32 + 10 + 0.5 = (42.5)_{10}\end{aligned}$$

$$\begin{aligned}\blacktriangleright (341.24)_5 &= 3 \times 5^2 + 4 \times 5^1 + 1 \times 5^0 + 2 \times 5^{-1} + 4 \times 5^{-2} \\ &= 75 + 20 + 1 + 0.4 + 0.16 = (96.56)_{10}\end{aligned}$$

Numbers with Different Bases

Decimal (base 10)	Binary (base 2)	Octal (base 8)	Hexadecimal (base 16)
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Problems

1. Convert the following numbers with the indicated bases to decimal:

- (a) $(4310)_5$
- (b) $(198)_{12}$
- (c) $(435)_8$
- (d) $(345)_6$

2. Convert the hexadecimal number 64CD to binary, and then convert it from binary to octal.

3. Express the following numbers in decimal:

(a) $(10110.0101)_2$

(b) $(16.5)_{16}$

(c) $(26.24)_8$

(d) $(DADA.B)_{16}$

(e) $(1010.1101)_2$

4. Convert the following binary numbers to hexadecimal
and to decimal:

(a) 1.10010

(b) 110.010.

5. Convert the decimal number 431 to binary in two
ways:

(a) convert directly to binary;

(b) convert first to hexadecimal and then from
hexadecimal to binary.

Which method is faster?

6. Determine the base of the numbers in each case for the following operations to be correct:

$$(a) 14/2 = 5 \quad (b) 54/4 = 13 \quad (c) 24+17=40$$

Solution:

The base of the numbers in each case for the following operations to be correct:

$$14/2 = 5;$$

Find decimal equivalent

$$14 = 1 \times r^1 + 4 \times r^0 = r + 4$$

$$2 = 2 \times r^0 = 2$$

$$5 = 5 \times r^0 = 5$$

$$(4+r)/2 = 5$$

Solving this equation, we get $r=6$, base 6

$$54/4 = 13;$$

Find decimal equivalent

$$54=5 \times r^1 + 4 \times r^0 = 5r + 4$$

$$4=4 \times r^0=4$$

$$13=1 \times r^1 + 3 \times r^0 = r + 3$$

$$(5r+4)/4= r + 3$$

Solving this equation, we get $r=8$, base 8

$$24+17=40;$$

Find decimal equivalent

$$24=2 \times r^1 + 4 \times r^0 = 2r + 4$$

$$17=1 \times r^1 + 7 \times r^0 = r + 7$$

$$40=4 \times r^1 + 0 \times r^0 = 4r + 0$$

$$(2r + 4) + (r + 7) = 4r$$

Solving this equation, we get $r=11$, base 11

Binary arithmetic(addition and subtraction)

Complements of Numbers (r and r-1complement)

Binary Addition

Basic mathematical operations with binary numbers works similar to the decimal system. However there are a few rules specific to the binary system.

Case	A + B	Sum	Carry
1	0 + 0	0	0
2	0 + 1	1	0
3	1 + 0	1	0
4	1 + 1	0	1

$$\begin{array}{r} \textcolor{red}{0\ 1\ 1\ 1} \\ 00111 \quad 7 \\ \hline 10101 \quad 21 \\ \hline 11100 = 28 \end{array}$$

Binary Addition



Addition Rules:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 0 \text{ carry } 1$$

$$1 + 1 + 1 = 1 \text{ carry } 1$$

OVERFLOW

Example 1:

$$\begin{array}{r} 0101 \\ + 0110 \\ \hline 1011 \end{array}$$

Example 2:

$$\begin{array}{r} 11 \\ 10111 \\ + 10010 \\ \hline 101001 \end{array}$$

Binary Subtraction

Binary subtraction is also similar to that of decimal subtraction with the difference that when 1 is subtracted from 0, it is necessary to borrow 1 from the next higher order bit and that bit is reduced by 1 (or 1 is added to the next bit of subtrahend) and the remainder is 1.

Binary Subtraction Rule Chart

Rules and tricks: Binary subtraction is much easier than the decimal subtraction when you remember the following rules:

- $0 - 0 = 0$
- $0 - 1 = 1$ (with a borrow of 1)
- $1 - 0 = 1$
- $1 - 1 = 0$

$$\begin{array}{r} & 0 & 10 \\ & 1 & 1 & 0 & 0 \\ (-) & 1 & 0 & 1 & 0 \\ \hline & 0 & 0 & 1 & 0 \end{array}$$

borrow

$$\begin{array}{r} 110 \\ - 101 \\ \hline 001 \end{array}$$

110 - 101 = 1

COMPLEMENTS OF NUMBERS

- Complements are used in digital computers to **simplify the subtraction operation** and for **logical manipulation**.
- Simplifying operations leads to simpler, less expensive circuits to implement the operations.
- There are two types of complements for each **base- r** system: the **radix complement** and the **diminished radix complement**.
- The first is referred to as the **r 's complement** and the second as the **$(r - 1)$'s complement**.
- *When the value of the base r is substituted in the name, the two types are referred to as the 2's complement and 1's complement for binary numbers and the 10's complement and 9's complement for decimal numbers.*

Diminished Radix Complement

- Given a number N in base r having n digits, the $(r - 1)$'s complement of N , i.e., its diminished radix complement, is defined as $(r^n - 1) - N$.
- For decimal numbers, $r = 10$ and $r - 1 = 9$, so the 9's complement of N is $(10^n - 1) - N$.
- In this case, 10^n represents a number that consists of a single 1 followed by n 0's.

Diminished Radix Complement

- $10^n - 1$ is a number represented by n 9's.
- For example, if $n = 4$, we have $10^4 = 10,000$ and $10^4 - 1 = 9999$.
- It follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9.

Calculation of the 9's complement of 546700

999999

-546700

453299

Calculation of the 9's complement of 012398

999999

-012398

987601

Diminished Radix Complement

- For **binary numbers**, $r = 2$ and $r - 1 = 1$, so the 1's complement of N is $(2^n - 1) - N$.
- Again, 2^n is represented by a binary number that consists of a 1 followed by n 0's.
- $2^n - 1$ is a binary number represented by **n 1's**.

Diminished Radix Complement

- For example, if $n = 4$, we have $2^4 = (10000)_2$ and $2^4 - 1 = (1111)_2$.
- Thus, the 1's complement of a binary number is obtained by **subtracting each digit from 1**. However, when subtracting binary digits from 1, we can have either $1 - 0 = 1$ or $1 - 1 = 0$, which causes the bit to change from 0 to 1 or from 1 to 0, respectively.
- Therefore, **the 1's complement of a binary number is formed by changing 1's to 0's and 0's to 1's**.

1's complement of 1011000

1	1	1	1	1	1	1
-	1	0	1	1	0	0
<hr/>						
0	1	0	0	1	1	1

The 1's complement of 0101101

$$\begin{array}{r} 1111111 \\ - 0101101 \\ \hline 1010010 \end{array}$$

- The $(r - 1)$'s complement of octal or hexadecimal numbers is obtained by subtracting each digit from 7 or F (decimal 15), respectively.

Radix Complement

- The r 's complement of an n -digit number N in base r is defined as $r^n - N$ for $N \neq 0$ and as 0 for $N = 0$.
- Comparing with the $(r - 1)$'s complement, we note that the r 's complement is obtained by adding 1 to the $(r - 1)$'s complement, since $r^n - N = [(r^n - 1) - N] + 1$.

10's complement of decimal 2389

- Thus, the 10's complement of decimal 2389 is $7610 + 1 = 7611$ and is obtained by adding 1 to the 9's complement value.

$$\begin{array}{r} 9999 \\ - 2389 \\ \hline 7610 \\ + \quad 1 \\ \hline 7611 \end{array}$$

2's complement of binary 101100

- The 2's complement of binary 101100 is $010011 + 1 = 010100$ and is obtained by adding 1 to the 1's-complement value.

1's-complement of 101100= 010011

$$\begin{array}{r} + \\ 010011 \\ \hline 010100 \end{array}$$

Radix Complement

Since 10^n is a number represented by a 1 followed by n 0's, $10^n - N$, which is the 10's complement of N , can be formed also by leaving all least significant 0's unchanged, subtracting the first nonzero least significant digit from 10, and subtracting all higher significant digits from 9.

10's complement of 012398 is 987602

1000000

- 012398

987602

10's complement of 246700 is 753300

$$\begin{array}{r} 1000000 \\ - 246700 \\ \hline 753300 \end{array}$$

- The 10's complement of the first number is obtained by subtracting 8 from 10 in the least significant position and subtracting all other digits from 9.
- The 10's complement of the second number is obtained by leaving the two least significant 0's unchanged, subtracting 7 from 10, and subtracting the other three digits from 9.
- Similarly, the 2's complement can be formed by leaving all least significant 0's and the first 1 unchanged and replacing 1's with 0's and 0's with 1's in all other higher significant digits.

Obtain 2's complement of the binary number 1101100

1's complement of 1101100=0010011

2's complement of 1101100= 1's complement of
1101100+1

$$= 0010011$$

$$\begin{array}{r} + \\ 0010011 \\ \hline \end{array}$$

$$= 0010100$$

Obtain 2's complement of the binary number 0110111

1's complement of 0110111 = 1001000

2's complement of 0110111 = 1's complement of
0110111 + 1

$$= 1001000$$

$$\begin{array}{r} + \\ 1 \end{array}$$

$$= 1001001$$

- The 2's complement of the first number is obtained by leaving the two least significant 0's and the first 1 unchanged and then replacing **1's with 0's and 0's with 1's** in the other four most significant digits.
- The 2's complement of the second number is obtained by leaving the least significant 1 unchanged and **complementing all other digits**.

- It is also worth mentioning that **the complement of the complement restores the number to its original value .**
- To see this relationship, note that the r 's complement of N is $r^n - N$, so that the complement of the complement is
 $r^n - (r^n - N) = N$ and is equal to the original number.

Assignment

1. Add and multiply the following numbers without converting them to decimal.

- (a) Binary numbers 1011 and 101.
- (b) Hexadecimal numbers 2E and 34.

2. Obtain the 1's and 2's complements of the following binary numbers:

- | | |
|--------------|---------------|
| (a) 00010000 | (b) 00000000 |
| (c) 11011010 | (d) 10101010 |
| (e) 10000101 | (f) 11111111. |

3. Find the 9's and the 10's complement of the following decimal numbers:

- (a) 25,478,03
- (b) 63, 325, 600
- (c) 25,000,000
- (d) 00,000,000.

4. (a) Find the 16's complement of C3DF.

- (b) Convert C3DF to binary.
- (c) Find the 2's complement of the result in (b).
- (d) Convert the answer in (c) to hexadecimal and compare with the answer in (a).

LECTURE 4

Subtraction with Complements (1's and 2's)

Subtraction with Complements

- The **direct method** of subtraction taught in elementary schools uses the borrow concept.
- In this method, we borrow a 1 from a higher significant position when the minuend digit is smaller than the subtrahend digit.
- The method works well when people perform subtraction with paper and pencil.
- However, when subtraction is implemented with digital hardware, the method is **less efficient** than the method that uses complements.

Subtraction of two *n-digit unsigned numbers* $M - N$ in base r

- The subtraction of two *n-digit unsigned numbers* $M - N$ in base r can be done as follows:
 1. Add the minuend M to the r 's complement of the subtrahend N . Mathematically,
$$M + (r^n - N) = M - N + r^n.$$
 2. If $M \geq N$, the sum will produce an end carry r^n , which can be discarded; what is left is the result $M - N$.

Subtraction of two *n*-digit unsigned numbers $M - N$ in base r

3. If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the r 's complement of $(N - M)$.

To obtain the answer in a familiar form, take the r 's complement of the sum and place a negative sign in front.

Subtraction of two *n*-digit unsigned numbers $M - N$ in base r

- Both numbers must have the same number of digits
1. Calculate the r 's complement of the subtrahend N .
Add the minuend M to the r 's complement of the subtrahend N .
- Inspect the result obtained in step (1) for an end carry:
- If an end carry occurs, discard it.
 - If an end carry does not occur, take the r 's complement of the number obtained in step (1) and place a negative sign in front.

Subtraction with r's Complement

The subtraction of two positive numbers ($M - N$), both of base r , may be done as follows:

- (1) Add the minuend M to the r 's complement of the subtrahend N .
- (2) Inspect the result obtained in step (1) for an end carry:
 - (a) If an end carry occurs, discard it.
 - (b) If an end carry does not occur, take the r 's complement of the number obtained in step (1) and place a negative sign in front.

Subtraction Using 10's complement

Using 10's complement, subtract $72532 - 3250$.

$$M = \begin{array}{r} 72532 \\ + 96750 \\ \hline \end{array}$$

$$\text{10's complement of } N = \begin{array}{r} + 96750 \\ \hline \end{array}$$

Sum = 169282

$$\text{Discard end carry } 10^5 = \begin{array}{r} - 100000 \\ \hline \end{array}$$

$$\text{Answer} = 69282$$

Subtraction Using 10's complement

Using 10's complement, subtract $3250 - 72532$.

$$\begin{array}{r} M = 03250 \\ \text{10's complement of } N = +27468 \\ \text{Sum} = 30718 \end{array}$$

There is no end carry. Therefore, the answer is $-(\text{10's complement of } 30718) = -69282$.

Subtraction Using 2's complement

Given the two binary numbers $X = 1010100$ and $Y = 1000011$, perform the subtraction

(a) $X - Y$ and (b) $Y - X$ by using 2's complements.

(a)
$$X = \begin{array}{r} 1010100 \\ + 0111101 \\ \hline \end{array}$$

$$\begin{array}{r} 2\text{'s complement of } Y = + \\ \hline \text{Sum} = 10010001 \end{array}$$

$$\text{Discard end carry } 2^7 = - \underline{\underline{10000000}}$$

Answer: $X - Y = 0010001$

(b)
$$Y = \begin{array}{r} 1000011 \\ + 0101100 \\ \hline \end{array}$$

$$\begin{array}{r} 2\text{'s complement of } X = + \\ \hline \text{Sum} = 1101111 \end{array}$$

There is no end carry. Therefore, the answer is $Y - X = -(2\text{'s complement of } 1101111) = -0010001$.

Subtraction with $(r - 1)$'s Complement

Both numbers must have the same number of digits

- (1) Add the minuend M to the $(r - 1)$'s complement of the subtrahend N .
- (2) Inspect the result obtained in step (1) for an end carry.
 - (a) If an end carry occurs, add 1 to the least significant digit (end around carry).
 - (b) If an end carry does not occur, take the $(r - 1)$'s complement of the number obtained in step (1) and place a negative sign in front.

Subtraction Using 9's complement

(a) $M = 72532$
 $N = 03250$

end around carry

$$\begin{array}{r} 72532 \\ + 96749 \\ \hline 1\cancel{6}9281 \\ \hline 69282 \end{array}$$

9's complement of $N = 96749$

ANSWER: 69282

(b) $M = 03250$
 $N = 72532$

no carry

$$\begin{array}{r} 03250 \\ + 27467 \\ \hline \cancel{3}0717 \end{array}$$

9's complement of $N = 27467$

ANSWER: $-69282 = -$ (9's complement of 30717)

Subtraction Using 1's complement

(a) $X - Y = 1010100 - 1000011$

$$\begin{array}{r} X = 1010100 \\ \text{1's complement of } Y = + 0111100 \\ \text{Sum} = 10010000 \\ \text{End-around carry} = + \underline{\hspace{2cm}} 1 \\ \text{Answer: } X - Y = 0010001 \end{array}$$

(b) $Y - X = 1000011 - 1010100$

$$\begin{array}{r} Y = 1000011 \\ \text{1's complement of } X = + \underline{0101011} \\ \text{Sum} = 1101110 \end{array}$$

There is no end carry. Therefore, the answer is $Y - X = -(1\text{'s complement of } 1101110) = -0010001$.

Some additional problems:

- Perform subtraction on the given unsigned numbers using the **10's** and **9's** complement of the subtrahend. Where the result should be negative, find its **10's/9's** complement and affix a minus sign. Verify your answers.

(a) $4,637 - 2,579$

(b) $125 - 1,800$

(c) $2,043 - 4,361$

(d) $1,631 -$

Additional Problems:

- Perform subtraction on the given unsigned binary numbers using the 2's and 1's complement of the subtrahend. Where the result should be negative, find its 2's/1's complement and affix a minus sign.
 - (a) $10011 - 10010$
 - (b) $100010 - 100110$
 - (c) $1001 - 110101$
 - (d) $101000 - 10101$

LECTURE 5

Binary Codes

Binary Codes

- Digital systems use signals that have two distinct values and circuit elements that have two stable states.
- There is a direct analogy among binary signals, binary circuit elements, and binary digits.
- A binary number of n digits, for example, may be represented by n binary circuit elements, each having an output signal equivalent to 0 or 1.

Binary Codes

- Digital systems represent and manipulate not only **binary numbers**, but also many other discrete elements of information.
- Any discrete element of information that is distinct among a group of quantities can be represented with a **binary code** (i.e., a pattern of 0's and 1's).
- The codes must be in binary because, in today's technology, only circuits that represent and manipulate patterns of 0's and 1's can be **manufactured economically** for use in computers.

Binary Codes

- However, it must be realized that **binary codes** merely change the symbols, not the meaning of the elements of information that they represent.
- If we inspect the bits of a computer at random, we will find that most of the time they represent some type of coded information rather than binary numbers.
- An *n-bit binary code is a group of n bits that assumes up to r^n distinct combinations of 1's and 0's, with each combination representing one element of the set that is being coded.*

Binary Codes

- A set of four elements can be coded with **two bits**, with each element assigned one of the following bit combinations: 00, 01, 10, 11.
- A set of eight elements requires a **three-bit code** and a set of 16 elements requires a **four-bit code**.
- The bit combination of an *n-bit code is determined from the count in binary from 0 to $2^n - 1$. Each element* must be assigned a unique binary bit combination, and no two elements can have the same value; otherwise, the code assignment will be ambiguous.

Binary Coded Decimal (BCD)

Binary-Coded Decimal (BCD)

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

A decimal number in BCD is the same as its equivalent binary number only when the number is between 0 and 9.

The binary combinations 1010 through 1111 are not used and have no meaning in BCD.

Binary Coded Decimal (BCD)

- A number with k decimal digits will require $4k$ bits in BCD. Decimal 396 is represented in BCD with 12 bits as 0011 1001 0110, with each group of 4 bits representing one decimal digit.
- A decimal number in BCD is the same as its equivalent binary number only when the number is between 0 and 9. A BCD number greater than 10 looks different from its equivalent binary number, even though both contain 1's and 0's.
- Moreover, the binary combinations 1010 through 1111 are not used and have no meaning in BCD. Consider decimal 185 and its corresponding value in BCD and binary:

$$(185)_{10} = (0001\ 1000\ 0101)_{BCD} = (10111001)_2$$

BCD Addition

$$\begin{array}{r} 4 \quad 0100 \\ +5 \quad +0101 \\ \hline 9 \quad 1001 \end{array} \quad \begin{array}{r} 4 \quad 0100 \\ +8 \quad +1000 \\ \hline 12 \quad 1100 \end{array} \quad \begin{array}{r} 8 \quad 1000 \\ +9 \quad +1001 \\ \hline 17 \quad 10001 \end{array}$$
$$\begin{array}{r} \\ \\ \hline +0110 \\ \hline 10010 \end{array} \quad \begin{array}{r} \\ \\ \hline +0110 \\ \hline 10111 \end{array}$$

BCD Addition

BCD	1	1		
	0001	1000	0100	184
	+0101	0111	0110	+576
Binary sum	<u>0111</u>	<u>10000</u>	<u>1010</u>	
Add 6		0110	0110	
BCD sum	<u>0111</u>	<u>0110</u>	<u>0000</u>	<u>760</u>

Other Binary Codes

Four Different Binary Codes for the Decimal Digits

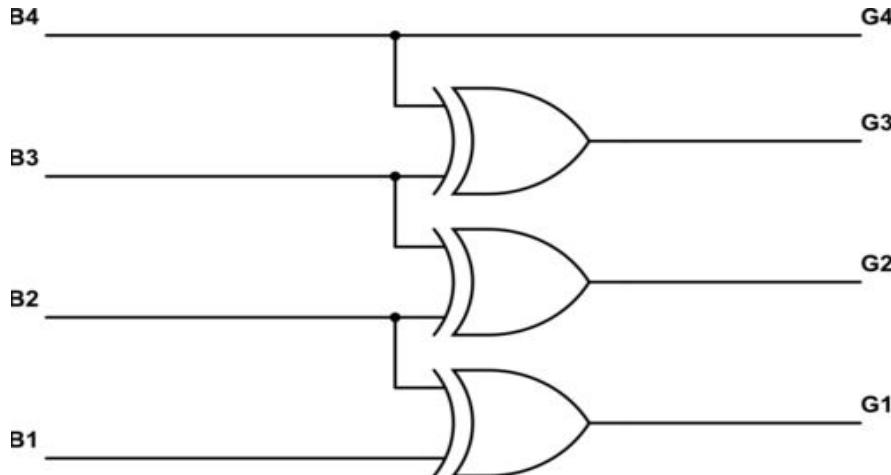
Decimal Digit	BCD 8421	2421	Excess-3	8, 4, -2, -1
0	0000	0000	0011	0000
1	0001	0001	0100	0111
2	0010	0010	0101	0110
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1011
6	0110	1100	1001	1010
7	0111	1101	1010	1001
8	1000	1110	1011	1000
9	1001	1111	1100	1111
Unused bit combi- nations	1010	0101	0000	0001
	1011	0110	0001	0010
	1100	0111	0010	0011
	1101	1000	1101	1100
	1110	1001	1110	1101
	1111	1010	1111	1110

Binary Codes

- The 2421 and the excess-3 codes are examples of **self-complementing codes**. Such codes have the property that the 9's complement of a decimal number is obtained directly by changing 1's to 0's and 0's to 1's (i.e., by complementing each bit in the pattern).
- For example, decimal 395 is represented in the excess-3 code as **0110 1100 1000**. The 9's complement of 395 is 604 .Now excess-3 code of 604 is **1001 0011 0111**, which is obtained simply by complementing (1's)each bit of excess-3 code of 395.
- The excess-3 code has been used in some older computers because of its self complementing property.
- **Excess-3 is an unweighted code in which each coded combination is obtained from the corresponding binary value plus 3.**
- **Note that the BCD code is not self-complementing.**

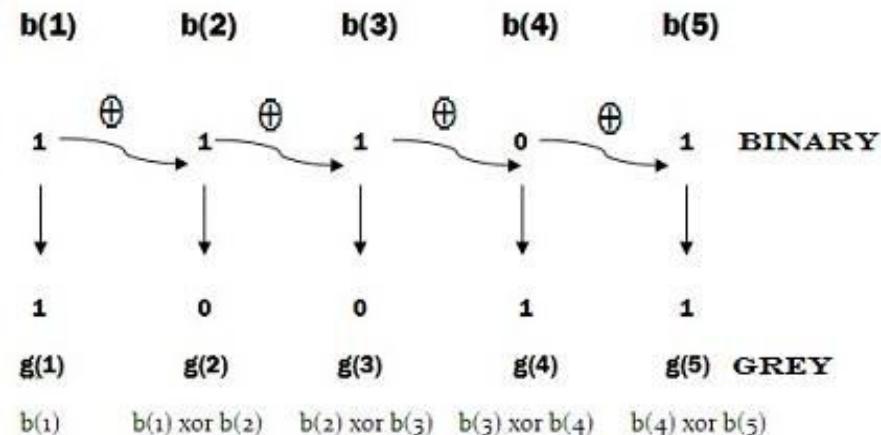
Gray code

Gray code – also known as **Cyclic Code**, **Reflected Binary Code (RBC)**, **Reflected Binary** (RB) or **Grey code** – is defined as an ordering of the binary number system such that each incremental value can only differ by one bit. In gray code, while traversing from one step to another step only one bit in the code group changes. That is to say that two adjacent code numbers differ from each other by only one bit.



Binary to Grey Code Conversion

Convert the binary 11101_2 to its equivalent Grey code

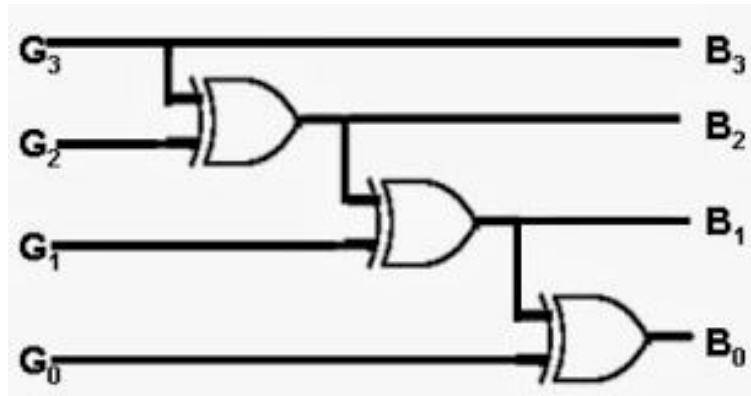


Gray code

- The Gray code is used in applications in which the normal sequence of binary numbers generated by the hardware may produce an error or ambiguity during the transition from one number to the next.
- If binary numbers are used, a change, for example, from 0111 to 1000 may produce an intermediate erroneous number 1001 if the value of the rightmost bit takes longer to change than do the values of the other three bits.
- This could have **serious consequences** for the machine using the information. The **Gray code eliminates this problem, since only one bit changes its value during any transition between two numbers.**

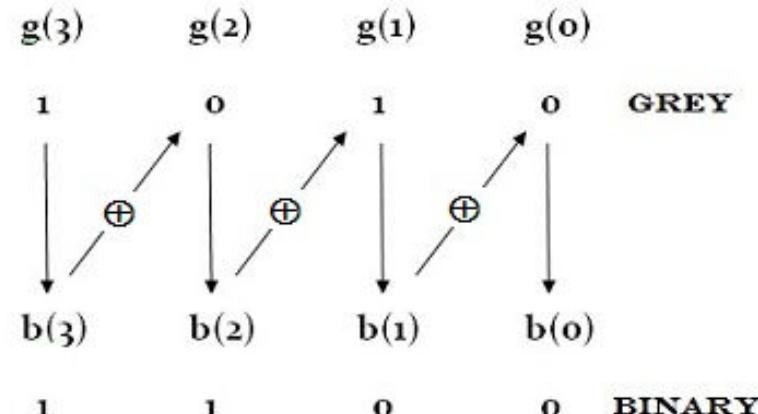
BINARY INPUT				GRAY CODE OUTPUT			
B3	B2	B1	B0	G3	G2	G1	G0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Gray to Binary Conversion



Grey Code to Binary Conversion

Convert the Grey code 1010 to its equivalent Binary



$$\text{i.e } b(3) = g(3)$$

$$b(2) = b(3) \oplus g(2)$$

$$b(1) = b(2) \oplus g(1)$$

$$b(0) = b(1) \oplus g(0)$$

GRAY CODE INPUT				BINARY OUTPUT			
G3	G2	G1	G0	B3	B2	B1	B0
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

Parity Bit

The parity generating technique is one of the most widely used error detection techniques for the data transmission. In digital systems, when binary data is transmitted and processed , data may be subjected to noise so that such noise can alter 0s (of data bits) to 1s and 1s to 0s. **parity bit** is added to the word containing data in order to make number of 1s either even or odd. Thus it is used to detect errors , during the transmission of binary data .

Parity Generator

It is combinational circuit that accepts an $n-1$ bit stream data and generates the additional bit that is to be transmitted with the bit stream. This additional or extra bit is termed as a parity bit.

In **even parity** bit scheme, the parity bit is ‘**0**’ if there are **even number of 1s** in the data stream and the parity bit is ‘**1**’ if there are **odd number of 1s** in the data stream.

In **odd parity** bit scheme, the parity bit is ‘**1**’ if there are **even number of 1s** in the data stream and the parity bit is ‘**0**’ if there are **odd number of 1s** in the data stream. Let us discuss both even and odd parity generators.

Even Parity Generator

Let us assume that a 3-bit message is to be transmitted with an even parity bit.

Let the three inputs A, B and C are applied to the circuits and output bit is the parity bit P.

The total number of 1s must be even, to generate the even parity bit P. The figure shows the truth table of even parity generator in which 1 is placed as parity bit in order to make all 1s as even when the number of 1s in the truth table is odd.

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

Odd Parity Generator

Let us consider that the 3-bit data is to be transmitted with an odd parity bit. The three inputs are A, B and C and P is the output parity bit. The total number of bits must be odd in order to generate the odd parity bit.

In the given truth table below, 1 is placed in the parity bit in order to make the total number of bits odd when the total number of 1s in the truth table is even.

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

Some Additional Problems on Binary Codes:

- Represent the unsigned decimal numbers 791 and 658 in BCD, and then show the steps necessary to form their sum.
- Represent the decimal number 6,248 in (a) BCD, (b) excess-3 code, (c) 2421 code, and (d) a 6311 code.
- The state of a 12-bit register is 100010010111. What is its content if it represents
 - (a) Three decimal digits in BCD?
 - (b) Three decimal digits in the excess-3 code?
 - (c) Three decimal digits in the 84-2-1 code?
 - (d) A binary number?

Contd.

- Express the decimal 5280 in Excess-3 code.
- What is the decimal equivalent of this excess-3 code:

0110 1001 1100 0111

- Convert the Gray Code 1110 to its Binary equivalent?
- Generate the even parity bit table for a 4-bit message.
- Generate an odd parity bit table for a 2-bit message.

Contd.

- Draw truth table for 3-bit Binary to Gray code and Gray Code to Binary code.
- Prove that why 8 4 -2 -1 code is not a self complementing code?
- Do the BCD addition of the followings BCD nos:
 - a. 86 and 39
 - b. 99 and 0
 - c. 123 and 37

Boolean Algebra

INTRODUCTION

- Because binary logic is used in all of today's digital computers and devices, the **cost of the circuits** that implement it is an important factor addressed by designers—be they computer engineers, electrical engineers, or computer scientists.
- Finding simpler and cheaper, but equivalent, realizations of a circuit can reap huge payoffs in reducing the overall cost of the design.
- Mathematical methods that simplify circuits rely primarily on Boolean algebra.
- Therefore, this topic provides a basic vocabulary and a brief foundation in Boolean algebra that will enable you to optimize simple circuits and to understand the purpose of algorithms used by software tools to optimize complex circuits involving millions of logic gates.

Boolean Algebra

Boolean algebra, like any other deductive mathematical system, may be defined with a set of elements, a set of operators, and a number of unproved axioms or postulates.

BASIC DEFINITIONS

- The most common postulates used to formulate various algebraic structures are:
 - 1. Closure.** A set S is closed with respect to a binary operator if, for every pair of elements of S , the binary operator specifies a rule for obtaining a unique element of S .
For Example: $N=\{1,2,3,4\dots\}$, is closed with respect to the binary operator $+$ by the rules of arithmetic addition, since for any $a,b \in N$ we obtain a unique $c \in N$ by the operation $a+b=c$.
 - The set of natural numbers is *not closed with respect to the binary operator - by the rules of arithmetic* subtraction, because $2-3=-1$ and $2,3 \in N$, while $(-1) \notin N$.

- 2. Associative law.** A binary operator $*$ on a set S is said to be associative whenever

$$(x * y) * z = x * (y * z) \text{ for all } x, y, z, \in S$$

- 3. Commutative law.** A binary operator $*$ on a set S is said to be commutative whenever

$$x * y = y * x \text{ for all } x, y \in S$$

Identity element

4. **Identity element.** A set S is said to have an identity element with respect to a binary operation $*$ on S if there exists an element $e \in S$ with the property that

$$e * x = x * e = x \text{ for every } x \in S$$

- Example: The element 0 is an identity element with respect to the binary operator + on the set of integers
 $I = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$,
since $x + 0 = 0 + x = x$ for any $x \in I$
- The set of natural numbers, N , has no identity element, since 0 is excluded from the set.

Inverse

5. 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

$$x * y = e$$

- Example: In the set of integers, I , and the operator $+$, with $e = 0$, the inverse of an element a is $(-a)$, since $a + (-a) = 0$.

Distributive law

6. Distributive law. If $*$ and \cdot are two binary operators on a set S , $*$ is said to be distributive over \cdot .

$$x * (y \cdot z) = (x * y) \cdot (x * z)$$

Basic Definitions

- The operators and postulates have the following meanings:

The binary operator + defines addition. The **additive identity** is 0.

Ex. $a + 0 = 0 + a$

The **additive inverse** defines subtraction.

The binary operator \cdot defines multiplication.

The **multiplicative identity** is 1. The **multiplicative inverse** of $a = 1/a$ defines division, i.e., $a \cdot 1/a = 1$

The only **distributive law** applicable is that of \cdot over $+$:

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

Axiomatic Definition of Boolean Algebra

- Boolean algebra is defined by a set of elements, B , provided following postulates with two binary operators, $+$ and \cdot , are satisfied:
 1. (a) The structure is closed with respect to the operator $+$.
(b) The structure is closed with respect to the operator \cdot .

x	y	$x \cdot y$	x	y	$x + y$
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	1	0	1
1	1	1	1	1	1

The structure *is closed with respect to the two operators is obvious from the tables*, since the result of each operation is either 1 or 0 and $1, 0 \in B$.

2. (a) The element 0 is an **identity element** with respect to +;
that is, $x + 0 = 0 + x = x$.

(b) The element 1 is an **identity element** with respect to . ;
that is, $x \cdot 1 = 1 \cdot x = x$.

3. (a) The structure is **commutative** with respect to +;
that is, $x + y = y + x$.

(b) The structure is **commutative** with respect to . ;
that is, $x \cdot y = y \cdot x$.

4. (a) The operator . is **distributive** over +;
that is, $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$.

(b) The operator + is **distributive** over . ;
that is, $x + (y \cdot z) = (x + y) \cdot (x + z)$.

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

$y + z$	$x \cdot (y + z)$
0	0
1	0
1	0
1	0
0	0
1	1
1	1
1	1

$x \cdot y$	$x \cdot z$	$(x \cdot y) + (x \cdot z)$
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	1	1
1	0	1
1	1	1

5. For every element $x \in B$, there exists an element $x' \in B$ (called the complement of x) such that

(a) $x + x' = 1$

since $0 + 0' = 0 + 1 = 1$ and $1 + 1' = 1 + 0 = 1$.

(b) $x \cdot x' = 0$.

since $0 \cdot 0' = 0 \cdot 1 = 0$ and $1 \cdot 1' = 1 \cdot 0 = 0$.

6. There exist at least two elements $x, y \in B$ such that $x \neq y$.

Duality principle

- *Duality principle states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged.*
- In a two-valued Boolean algebra, the identity elements and the elements of the set B are the same: 1 and 0.
- *The duality principle has many applications. If the dual of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's.*

Postulates and Theorems

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)	$x(x + y) = x$

Basic Theorems

Theorem 1(a): $x + x = x$

$= (x + x) \cdot 1$ $= (x + x) \cdot (x + x')$ $= x + xx'$ $= x + 0$ $= x$
Dual Dual back

Theorem 1(b): $x \cdot x = x$

$$\begin{aligned} &= x \cdot x + 0 \\ &= xx + xx' \\ &= x(x + x') \\ &= x \cdot 1 \\ &= x \end{aligned} \quad \begin{array}{l} \text{by postulate 2(a)} \\ \text{5(b)} \\ \text{4(a)} \\ \text{5(a)} \\ \text{2(b)} \end{array}$$

THEOREM 2(a): $x + 1 = 1$.

Statement	Justification
$x + 1 = 1 \cdot (x + 1)$	<i>postulate 2(b)</i>
$= (x + x')(x + 1)$	<i>5(a)</i>
$= x + x' \cdot 1$	<i>4(b)</i>
$= x + x'$	<i>2(b)</i>
$= 1$	<i>5(a)</i>

THEOREM 2(b): $x \cdot 0 = 0$

by applying duality property.

THEOREM 6(a): $x + x y = x$.

(Method-1)

Statement	Justification
$x + x y = x \cdot 1 + x y$	<i>postulate 2(b)</i>
$= x(1 + y)$	<i>4(a)</i>
$= x(y + 1)$	<i>3(a)</i>
$= x \cdot 1$	<i>2(a)</i>
$= x$	<i>2(b)</i>

(Method-2)

x	y	xy	$x + xy$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

THEOREM 6(b): $x(x + y) = x$

by applying duality property.

DeMorgan's theorem

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

x	y	$x + y$	$(x + y)'$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

x'	y'	$x'y'$
1	1	1
1	0	0
0	1	0
0	0	0

Operator Precedence

- The operator precedence for evaluating Boolean expressions is
 - (1) parentheses
 - (2) NOT
 - (3) AND
 - (4) OR

In other words, expressions inside parentheses must be evaluated before all other operations. The next operation that holds precedence is the complement, and then follows the AND and, finally, the OR.

Boolean Functions

Boolean function

- Boolean algebra is an algebra that deals with binary variables and logic operations.
- A **Boolean function** described by an algebraic expression consists of binary variables, the constants 0 and 1, and the logic operation symbols.
- For a given value of the binary variables, the function can be equal to either 1 or 0.

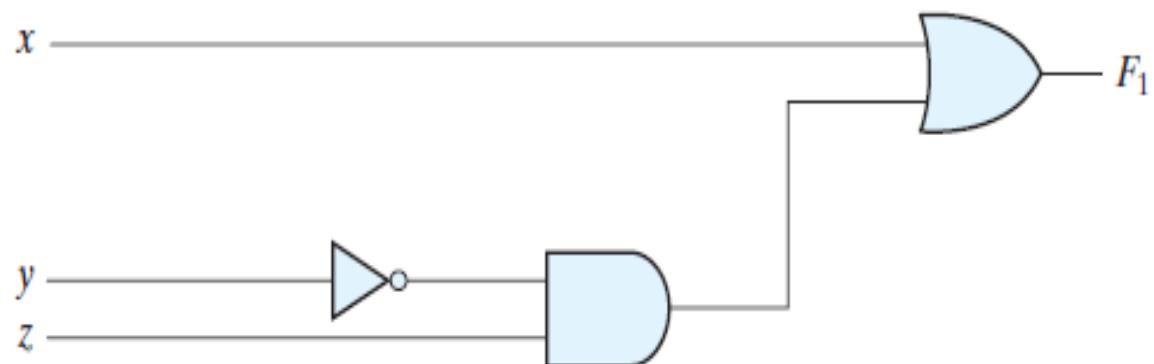
Boolean Functions

Consider the following Boolean function:

$$F_1 = x + y'z$$

A Boolean function can be represented in a truth table. The binary combinations for the truth table are obtained by counting from 0 through $2^n - 1$ see 0 to 7.

x	y	z	F_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	1



Simplification of Boolean Functions

- There is **only one way** that a Boolean function can be represented in a **truth table**.
- In **algebraic form**, it can be expressed in a **variety of ways and** all of them have equivalent logic.
- The particular expression used to represent the function will dictate the interconnection of gates in the logic-circuit diagram. Conversely, the interconnection of gates will dictate the logic expression.

Simplification of Boolean Functions

- Here is a key fact that motivates our use of Boolean algebra: By manipulating a Boolean expression according to the rules of Boolean algebra, it is sometimes possible to obtain a simpler expression for the same function and thus reduce the number of gates in the circuit and the number of inputs to the gate.
- Designers are motivated to reduce the complexity and number of gates because their effort can significantly reduce the cost of a circuit.

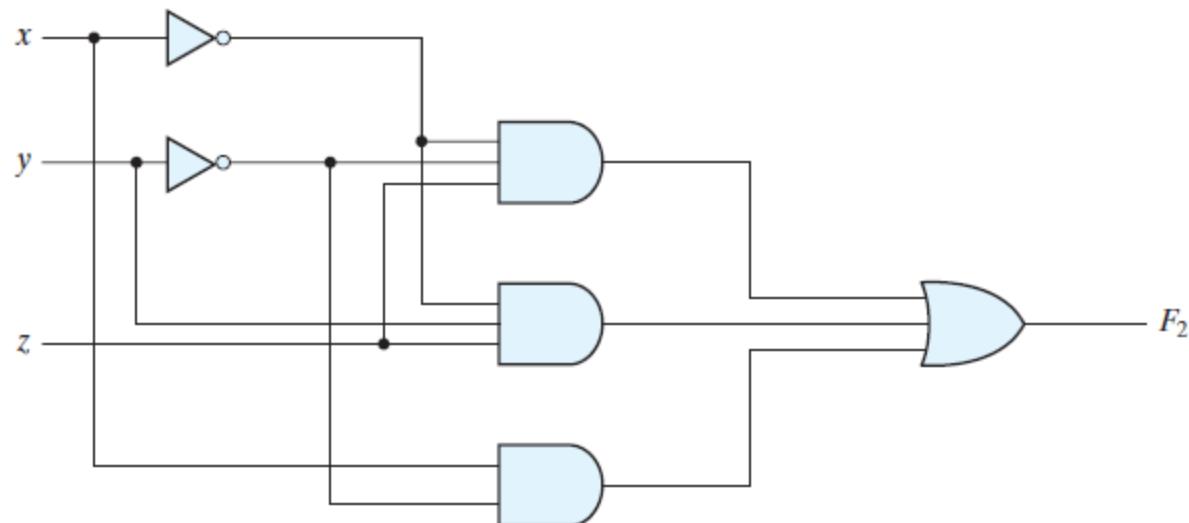
Before simplification of Boolean function

- Consider the following Boolean function:

$$F_2 = x'y'z + x'yz + xy'$$

This function with logic gates is shown in Fig. The function is equal to 1 when xyz = 001 or 011 or when xyz = 100,101 .

x	y	z	F ₂
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

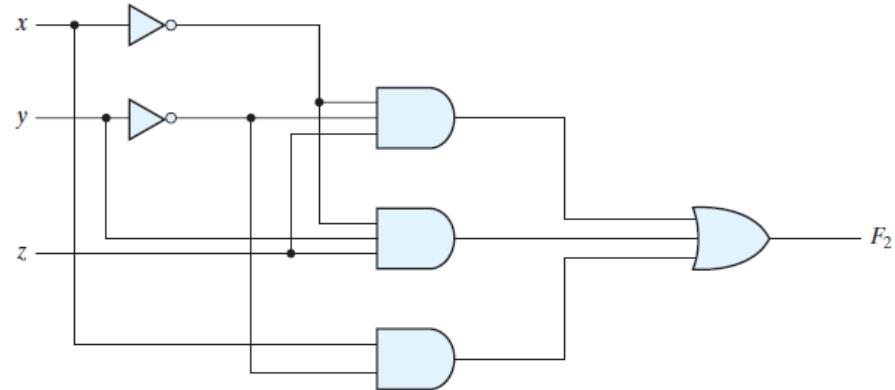


After simplification of Boolean function

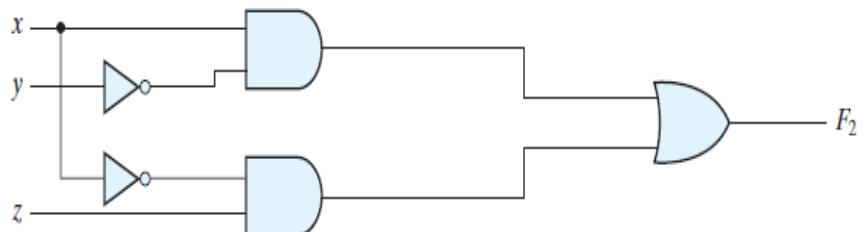
- Simplify the following Boolean function:

$$\begin{aligned}F_2 &= x'y'z + x'yz + xy' \\&= x'z(y' + y) + xy' \\&= x'z + xy'\end{aligned}$$

- The Reduced function would be preferable because it requires less wires and components.



simplified



Equivalent Expressions

$$F_2 = x'y'z + x'yz + xy' \text{ (primitive)}$$

$F_2 = 1$ when $xyz=001$ or 011 or

when $xy=100, 101$

$$F_2 = x'z + xy'$$

$F=1$ when $xz=01$ or

when $xy=10$

(simplified)

- Since both expression produce the same truth table, they are said to be equivalent.

x	y	z	F_2
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

- Therefore, the two circuits have the same outputs **for all possible binary combinations** of inputs of the three variables.
- Each circuit implements the same identical function, but the one with fewer gates and fewer inputs to gates is preferable because it requires **fewer wires and components**.
- In general, there are many equivalent representations of a logic function. Finding the most economic representation of the logic is an important design task.

Algebraic Manipulation

- When a Boolean expression is implemented with logic gates, each term requires a gate and each variable within the term designates an input to the gate.
- We define a *literal* to be a single variable within a term, in complemented or uncomplemented form.

- $F_2 = x'y'z + x'yz + xy'$ has three terms and five literals
- $F_2 = x'z + xy'$ has two terms and four literals
- By reducing the number of terms, the number of literals, or both in a Boolean expression, it is often possible to obtain a simpler circuit.
- The manipulation of Boolean algebra consists mostly of reducing an expression for the purpose of obtaining a simpler circuit.

Simplification of Boolean functions

Simplify the following Boolean functions to a minimum number of literals.

$$1. \quad x(x' + y) = xx' + xy = 0 + xy = xy.$$

$$2. \quad x + x'y = (x + x')(x + y) = 1(x + y) = x + y.$$

$$3. \quad (x + y)(x + y') = x + xy + xy' + yy' = x(1 + y + y') = x.$$

$$\begin{aligned}4. \quad xy + x'z + yz &= xy + x'z + yz(x + x') \\&= xy + x'z + xyz + x'yz \\&= xy(1 + z) + x'z(1 + y) \\&= xy + x'z.\end{aligned}$$

$$5. \quad (x + y)(x' + z)(y + z) = (x + y)(x' + z), \text{ by duality from function 4.}$$

Simplification of Boolean functions

The fourth function illustrates the fact that an increase in the number of literals sometimes leads to a simpler final expression.

Function 5 is not minimized directly, but can be derived from the dual of the steps used to derive function 4.

Functions 4 and 5 are together known as the ***consensus theorem***.

Assignment

1 Demonstrate the validity of the following identities by means of truth tables:

- (a) DeMorgan's theorem for three variables: $(x + y + z)' = x'y'z'$ and $(xyz)' = x' + y' + z'$
- (b) The distributive law: $x + yz = (x + y)(x + z)$
- (c) The distributive law: $x(y + z) = xy + xz$
- (d) The associative law: $x + (y + z) = (x + y) + z$
- (e) The associative law and $x(yz) = (xy)z$

2 Simplify the following Boolean expressions to a minimum number of literals:

- | | |
|------------------------------|---------------------------------|
| (a)* $xy + xy'$ | (b)* $(x + y)(x + y')$ |
| (c)* $xyz + x'y + xyz'$ | (d)* $(A + B)'(A' + B')'$ |
| (e) $(a + b + c')(a'b' + c)$ | (f) $a'bc + abc' + abc + a'bc'$ |

Assignment

- 3 Simplify the following Boolean expressions to a minimum number of literals:
- (a)* $ABC + A'B + ABC'$ (b)* $x'yz + xz$
(c)* $(x + y)'(x' + y')$ (d)* $xy + x(wz + wz')$
(e)* $(BC' + A'D)(AB' + CD')$ (f) $(a' + c')(a + b' + c')$
- 4 Reduce the following Boolean expressions to the indicated number of literals:
- (a)* $A'C' + ABC + AC'$ to three literals
(b)* $(x'y' + z)' + z + xy + wz$ to three literals
(c)* $A'B(D' + C'D) + B(A + A'CD)$ to one literal
(d)* $(A' + C)(A' + C')(A + B + C'D)$ to four literals
(e) $ABC'D + A'BD + ABCD$ to two literals

HARDWARE DESCRIPTION LANGUAGE

Contents

Hardware Descriptive Language

Design Entry

Logic Simulation

Logic Synthesis

Timing Verification

Fault Simulation

VHDL & Verilog

Module Declaration

Boolean Expression

Combinational Logic Modeled with Boolean Equations

Dataflow Modeling

Behavioral Modeling

Gate level Modeling

HARDWARE DESCRIPTION LANGUAGE

- Manual methods for designing logic circuits are feasible only when the circuit is small.
- For anything else (i.e., a practical circuit), designers use computer-based design tools. Coupled with the correct-by-construction methodology, computer-based design tools leverage the creativity and the effort of a designer and reduce the **risk of producing a flawed design**.
- Prototype **integrated circuits** are too expensive and time consuming to build, so all modern design tools rely on a hardware description language to **describe, design, and test a circuit** in software before it is ever manufactured.

HARDWARE DESCRIPTION LANGUAGE

- A hardware description language (HDL) is a computer-based language that describes the hardware of digital systems in a textual form. It resembles an ordinary computer programming language, such as C, but is specifically oriented to describing hardware structures and the behavior of logic circuits.
- It can be used to represent logic diagrams, truth tables, Boolean expressions, and complex abstractions of the behavior of a digital system. One way to view an HDL is to observe that it describes a relationship between signals that are the inputs to a circuit and the signals that are the outputs of the circuit.
- For example, an HDL description of an AND gate describes how the logic value of the gate's output is determined by the logic values of its inputs.

HARDWARE DESCRIPTION LANGUAGE

- As a documentation language, an 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, retrieved, edited, and transmitted easily and processed by computer software in an efficient manner.
- HDLs are used in several major steps in the design flow of an integrated circuit design entry, functional simulation or verification, logic synthesis, timing verification, and fault simulation.

Design Entry

- *Design entry creates an HDL-based description of the functionality that is to be implemented in hardware.*
- Depending on the HDL, the description can be in a variety of forms: Boolean logic equations, truth tables, a netlist of interconnected gates, or an abstract behavioral model.
- The HDL model may also represent a **partition of a larger circuit into smaller interconnected and interacting functional units.**

Logic simulation

- *Logic simulation* displays the behavior of a digital system through the use of a computer.
- A simulator interprets the HDL description and either produces readable output, such as a time-ordered sequence of input and output signal values, or displays waveforms of the signals.
- The simulation of a circuit predicts how the hardware will behave before it is actually fabricated. Simulation detects functional errors in a design without having to physically create and operate the circuit.
- Errors that are detected during a simulation can be corrected by modifying the appropriate HDL statements. The stimulus (i.e., the logic values of the inputs to a circuit) that tests the functionality of the design is called a *test bench*.
- *Thus, to simulate a digital system, the design is first described in an HDL and then verified by simulating the design and checking it with a test bench, which is also written in the HDL.* An alternative and more complex approach relies on formal mathematical methods to prove that a circuit is functionally correct. We will focus exclusively on simulation.

Logic Synthesis

- *Logic synthesis is the process of deriving a list of physical components and their interconnections (called a netlist) from the model of a digital system described in an HDL.*
- The netlist can be used to fabricate an integrated circuit or to lay out a printed circuit board with the hardware counterparts of the gates in the list. **Logic synthesis is similar to compiling a program in a conventional high-level language.**
- The difference is that, instead of producing an object code, logic synthesis produces a database describing the elements and structure of a circuit. The database specifies how to fabricate a physical integrated circuit that implements in silicon the functionality described by statements made in an HDL.
- Logic synthesis is based on formal exact procedures that implement digital circuits and addresses that part of a digital design which can be automated with computer software. The design of today's large, complex circuits is made possible by logic synthesis software.

Timing Verification

- **Timing verification** confirms that the fabricated, integrated circuit will operate at a specified speed. Because each logic gate in a circuit has a propagation delay, a signal transition at the input of a circuit cannot immediately cause a change in the logic value of the output of a circuit.
- Propagation delays ultimately **limit the speed** at which a circuit can operate. Timing verification checks each signal path to verify that it is not compromised by propagation delay.
- This step is done after logic synthesis specifies the actual devices that will compose a circuit and before the circuit is released for production.

Fault Simulation

- In VLSI circuit design, ***fault simulation*** compares the behavior of an ideal circuit with the behavior of a circuit that contains a process-induced flaw. Dust and other particulates in the atmosphere of the clean room can cause a circuit to be fabricated with a fault.
- A circuit with a fault will not exhibit the same functionality as a fault-free circuit. **Fault simulation is used to identify input stimuli that can be used to reveal the difference between the faulty circuit and the fault-free circuit.**
- These test patterns will be used to test fabricated devices to ensure that **only good devices are shipped to the customer**. Test generation and fault simulation may occur at different steps in the design process, but they are always done before production in order to avoid the disaster of producing a circuit whose internal logic cannot be tested.

VHDL and Verilog

- Companies that design integrated circuits use proprietary and public HDLs. In the public domain, there are two standard HDLs that are supported by the IEEE: **VHDL and Verilog**. VHDL is a Department of Defense–mandated language. (The *V* in VHDL stands for the first letter in VHSIC, an acronym for very high-speed integrated circuit.)
- Verilog began as a proprietary HDL of **Cadence Design Systems**, but Cadence transferred control of Verilog to a consortium of companies and universities known as Open Verilog International (OVI) as a step leading to its adoption as an IEEE standard.

VHDL and Verilog

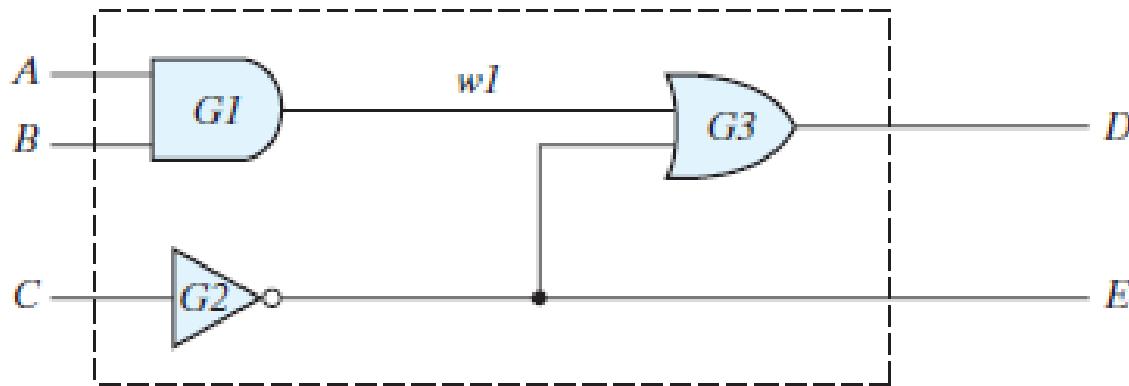
- Our emphasis will be on the **modeling, verification, and synthesis** (both manual and automated) of Verilog models of circuits having specified behavior.
- The Verilog HDL was initially approved as a standard HDL in 1995; revised and enhanced versions of the language were approved in 2001 and 2005.
- We will address only those features of Verilog, including the latest standard, that support our discussion of HDL-based design methodology for integrated circuits.

Module Declaration

- In particular, a Verilog model is composed of text using keywords, of which there are about 100. Keywords are predefined lowercase identifiers that define the language constructs.
- Examples of keywords are **module**, **endmodule**, **input**, **output**, **wire**, **and**, **or**, and **not**.

- Any text between two forward slashes (`//`) *and the end of the line is interpreted as a comment and will have no effect* on a simulation using the model. Multiline comments begin with `/ *` and terminate with `* /`.
- Blank spaces are ignored, but they may not appear within the text of a keyword, a user-specified identifier, an operator, or the representation of a number.
- Verilog is **case sensitive**, which means that uppercase and lowercase letters are distinguishable (e.g., **not** **is not the same as NOT**).
- The term **module** *refers to the text* enclosed by the keyword pair **module .. . endmodule**. **A module is the fundamental descriptive unit** in the Verilog language. It is declared by the keyword **module** and must always be terminated by the keyword **endmodule**.

Circuit to demonstrate an HDL



```
// Verilog model of circuit of Figure. IEEE 1364-1995 Syntax
module Simple_Circuit (A, B, C, D, E);
    Output    D, E;
    input     A, B, C;
    wire      w1;
    and      G1 (w1, A, B); // Optional gate instance name
    not      G2 (E, C);
    or       G3 (D, w1, E);
endmodule
```

Circuit to demonstrate an HDL

- The *port list of a module is the interface between the module and its environment.*
- In this example, the ports are the inputs and outputs of the circuit. the keywords **input** and **output** specify which of the ports are inputs and which are outputs.
- Internal connections are declared as wires. The circuit in this example has one internal connection, at terminal *w1* , and is declared with the keyword **wire**.

Circuit to demonstrate an HDL

- The structure of the circuit is specified by a list of (predefined) *primitive* gates, each identified by a descriptive keyword (**and**, **not**, **or**). The **elements of the list** are referred to as *instantiations of a gate*, each of which is referred to as a *gate instance* .
- Each *gate instantiation* consists of an optional name (such as *G1*, *G2* , etc.) followed by the gate output and inputs separated by commas and enclosed within parentheses.
- The output of a primitive gate is always listed first, followed by the inputs.

Circuit to demonstrate an HDL

- The output of a primitive must be **listed first**, but the inputs and outputs of a module may be listed in any order.
- The module description ends with the keyword **endmodule**.
- **Each statement must be terminated with a semicolon**, but there is no semicolon after **endmodule**.

Circuit to demonstrate an HDL

- It is important to understand the distinction between the terms *declaration and instantiation*. A Verilog module is declared. Its declaration specifies the input–output behavior of the hardware that it represents.
- Predefined **primitives** are not declared, because their definition is specified by the language and is not subject to change by the user.
- Primitives are used (i.e., instantiated), just as gates are used to populate a printed circuit board.

Boolean Expressions

- Boolean equations describing combinational logic are specified in Verilog with a continuous assignment statement consisting of the keyword **assign** followed by a Boolean expression.

Combinational Logic Modeled with Boolean Equations

$$E = A + BC + B'D$$

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

// Verilog model: Circuit with Boolean expressions

```
module Circuit_Boolean_CA (E, F, A, B, C, D);
```

```
output E, F;
```

```
input A, B, C, D;
```

```
assign E = A || (B && C) || ((!B) && D);
```

```
assign F = ((!B) && C) || (B && (!C) && (!D));
```

```
endmodule
```

Dataflow Modeling

- Dataflow modeling of combinational logic uses a number of operators that act on binary operands to produce a binary result.
- Verilog HDL provides about 30 different operators.
- It is necessary to distinguish between arithmetic and logic operations, so different symbols are used for each.

HDL Operators

Some Verilog HDL Operators

Symbol	Operation	Symbol	Operation
+	binary addition		
-	binary subtraction		
&	bitwise AND	&&	logical AND
	bitwise OR		logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
==	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

Dataflow: Four-Bit Comparator

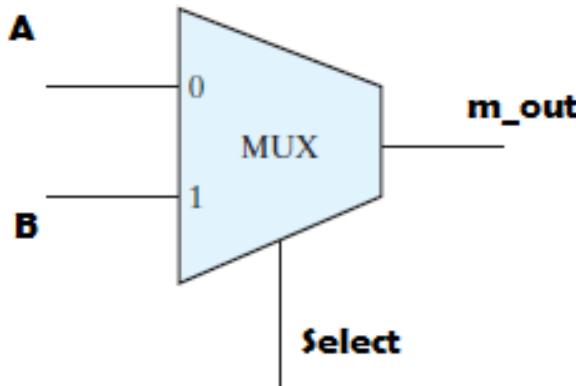
```
// Dataflow description of a four-bit comparator IN2001, 2005 syntax

module mag_compare
( output A_lt_B, A_eq_B, A_gt_B,
  input [3: 0] A, B
);
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (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 an optional event control expression** and a list of procedural assignment statements.
- The event control expression specifies when the statements will execute. The target output of a procedural assignment statement must be of the **reg data type**.
- **Contrary to the wire data type**, whereby the target output of an assignment may be continuously updated, a **reg data** type retains its value until a new value is assigned.

Behavioral: Two-to-One Line Multiplexer



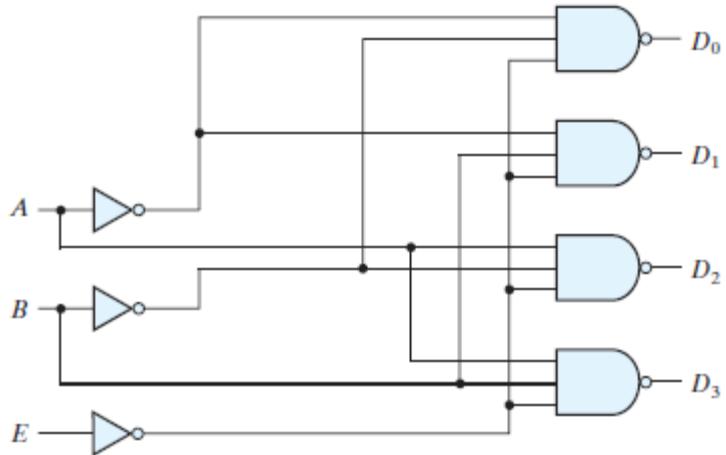
```
// Behavioral description of two-to-one-line multiplexer
module mux_2x1_beh (m_out, A, B, select);
    output      m_out;
    input       A, B, select;
    reg        m_out;

    always      @(A or B or select)
        if (select == 1) m_out = A;
        else m_out = B;
endmodule
```

Gate-Level Modeling

- In this type of representation, a circuit is specified by its logic gates and their interconnections. Gatelevel modeling provides a textual description of a schematic diagram.
- The Verilog HDL includes 12 basic gates as predefined primitives. Four of these primitive gates are of the three-state type.
- They are all declared with the lowercase keywords **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, and **buf**. Primitives such as **and** are ***n*-input primitives**.
- *They can have any number of scalar inputs* (e.g., a three-input **and** primitive). The **buf** and **not** primitives are ***n*-output primitives**. A single input can drive multiple output lines distinguished by their identifiers.

Gate-Level Modeling



// Gate-level description of two-to-four-line decoder
// Refer to Fig. 4.19 with symbol *E* replaced by *enable*, for clarity.

```
module decoder_2x4_gates (D, A, B, enable);  
    output [0:3] D;  
    input A, B;  
    input enable;  
    wire A_not, B_not, enable_not;  
  
    not  
        G1 (A_not, A),  
        G2 (B_not, B),  
        G3 (enable_not, enable);  
    nand  
        G4 (D[0], A_not, B_not, enable_not),  
        G5 (D[1], A_not, B, enable_not),  
        G6 (D[2], A, B_not, enable_not),  
        G7 (D[3], A, B, enable_not);  
endmodule
```

Combinational Logic Modeled with Boolean Equations

$$F = (A'C)AB + BCD'$$

- module Circuit_A (A, B, C, D, F);
- input A, B, C, D;
- output F;
- wire w, x, y, z, a, d;
- or (x, B, C, d);
- and (y, a ,C);
- and (w, z ,B);
- and (z, y, A);
- or (F, x, w);
- not (a, A);
- not (d, D);
- endmodule

```
> module Circuit_C (y1, y2, y3, a, b);  
> output y1, y2, y3;  
> input a, b;  
> assign y1 = a || b;  
> assign y2 = a && b;  
> endmodule
```

Canonical and Standard form

Contents

- Complement of Boolean function
- HDL Description of Boolean Function
- Canonical form representation of Boolean Functions
- Standard form representation of Boolean Functions

Complement of Boolean function

- The complement of a function F is F' and is obtained from an interchange of 0's for 1's and 1's for 0's in the value of F .
- The complement of a function may be derived algebraically through DeMorgan's theorems.
- DeMorgan's theorems can be extended to three or more variables.

$$\begin{aligned}(A + B + C)' &= (A + x)' \quad \text{let } B + C = x \\ &= A'x' \quad \text{by theorem 5(a) (DeMorgan)} \\ &= A'(B + C)' \quad \text{substitute } B + C = x \\ &= A'(B'C') \quad \text{by theorem 5(a) (DeMorgan)} \\ &= A'B'C' \quad \text{by theorem 4(b) (associative)}\end{aligned}$$

Complement of Boolean function

DeMorgan's theorems for any number of variables resemble the two-variable case in form and can be derived by successive substitutions similar to the method used in the preceding derivation. These theorems can be generalized as follows:

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

$$(ABCD\dots F)' = A' + B' + C' + D' + \dots + F'$$

The generalized form of DeMorgan's theorems states that the complement of a function is obtained by interchanging AND and OR operators and complementing each literal.

Complement of Boolean function

EX-1 $F_1 = x'yz' + x'y'z$ by
 applying DeMorgan's theorem.

$$\begin{aligned} F_1' &= (x'y'z' + x'y'z) ' \\ &= (x'y'z') ' \cdot (x'y'z) ' \\ &= (x + y' + z)(x + y + z') \end{aligned}$$

A simpler procedure for deriving the complement of a function is to take the dual of the function and complement each literal.

Ex2: $F_2 = x'yz' + x'y'z$

$$\begin{aligned} \text{Dual of } F_2 &= (x' + y + z')(x' + y' + z) \\ \text{complement each literal} &= (x + y' + z)(x + y + z') = F_2' \end{aligned}$$

Complement of Boolean function

Find the complement of the function F_2 by Applying DeMorgan's theorems

$$\begin{aligned}F'_2 &= [x(y'z' + yz)]' = x' + (y'z' + yz)' = x' + (y'z')'(yz)' \\&= x' + (y + z)(y' + z') \\&= x' + yz' + y'z\end{aligned}$$

Find the complement of the function F_2 by taking dual of the function and complementing each literal.

$$F_2 = x(y'z' + yz).$$

The dual of F_2 is $x + (y' + z')(y + z)$.

Complement each literal: $x' + (y + z)(y' + z') = F'_2$.

HDL Description of Boolean Function

- Boolean equations describing combinational logic are specified in Verilog with a **continuous assignment statement** consisting of the keyword **assign** followed by a **Boolean expression**.

Example: Write HDL Description of the function

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

- // Verilog model: Circuit with Boolean expressions

```
module Circuit_Boolean_CA ( F, X, Y, Z);
output F;
input X, Y, Z;
assign F = ((!X) && Y && (!Z) ) || ((!X) && (!Y) && Z);
endmodule
```

Minterms

- A binary variable may appear either in its normal form (x) or in its complement form (x').
- Now consider two binary variables x and y combined with an AND operation. Since each variable may appear in either form, there are four possible combinations: $x'y'$, $x'y$, xy' , and xy . Each of these four AND terms is called a **min-term**, or a standard product. In a similar manner, n variables can be combined to form 2^n min-terms.
- Each min-term is obtained from an AND term of the n variables, with each variable being primed if the corresponding bit of the binary number is a 0 and un-primed if a 1.

Maxterms

- *n variables forming an OR term, with each variable being primed or unprimed, provide 2^n possible combinations, called max-terms, or standard sums.*
- Now consider two binary variables x and y combined with an OR operation. Since each variable may appear in either form, there are four possible combinations: $x' + y'$, $x' + y$, $x + y'$, and $x + y$. Each of these four OR terms is called a maxterm. In a similar manner, n variables can be combined to form 2^n maxterms.
- Each maxterm is obtained from an OR term of the n variables, with each variable being primed if the corresponding bit of the binary number is a 1 and unprimed if a 0.

Min-terms and Max-terms for Three Binary Variables

x	y	z	Minterms		Maxterms	
			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	$x'y z'$	m_2	$x + y' + z$	M_2
0	1	1	$x'y z$	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 truth table consists of a set of inputs and outputs. If there are ‘n’ input variables, then there will be 2^n possible combinations with zeros and ones. So the value of each output variable depends on the combination of input variables. So, each output variable will have ‘1’ for some combination of input variables and ‘0’ for some other combination of input variables.
- Therefore, we can express each output variable in following two ways.
- Sum of minterms form
- Product of maxterms form
- Boolean functions expressed as a sum of minterms or product of maxterms are said to be in *canonical form*.

Sum of Minterms

A Boolean function can be expressed algebraically from a given truth table by forming a min-term for each combination of the variables that produces a 1 in the function and then taking the OR of all those terms.

$$f_1 = x' y' z + x y' z' + x y z$$

$$= m_1 + m_4 + m_7$$

$$= \sum(m_1, m_4, m_7)$$

$$= \Sigma m(1,4,7)$$

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

Example

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

Ans.

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

Combining all term

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

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

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

Product of Maxterms

Now consider the complement of a Boolean function.

The complement of f_1 is

$$f_1' = x'y'z' + x'y z' + x'yz + xy'z + xyz'$$

If we take the complement of f_1' ,

$$f_1 = (x + y + z)(x + y' + z)(x + y' + z')(x' + y + z')(x' + y' + z)$$

$$= M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6$$

$$= \prod (M_0, M_2, M_3, M_5, M_6)$$

$$= \prod (0, 2, 3, 5, 6)$$

Form a maxterm for each combination of the variables that produces a 0 in the function, and then form the AND of all those maxterms.

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

Example

Ex. Express the Boolean function $F = xy + x'z$ as a product of maxterms.

using distributive law $\rightarrow F = xy + x'z = (xy + x')(xy + z)$
 $= (x + x')(y + x')(x + z)(y + z)$
 $= (x' + y)(x + z)(y + z)$

Each OR term missing one variable

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

Combining all the terms

$$\begin{aligned}F &= (x + y + z)(x + y' + z)(x' + y + z)(x' + y + z') \\&= M_0 M_2 M_4 M_5 \\F(x, y, z) &= \prod(0, 2, 4, 5)\end{aligned}$$

Conversion between canonical forms

Ex. Boolean expression: $F = xy + x'z$

$$xy = 11 \text{ or } xz = 01$$

sum of minterms is

$$F(x, y, z) = \Sigma(1, 3, 6, 7)$$

product of maxterms is

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

Take complement of F'
by DeMorgan's
theorem

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

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

Standard SOP and POS forms

- We discussed two canonical forms of representing the Boolean outputs.
- Another way to express Boolean functions is in *standard form*. In this configuration, the terms that form the function may contain one, two, or any number of literals. There are two types of standard forms: **the sum of products and products of sums**.

Standard SOP form

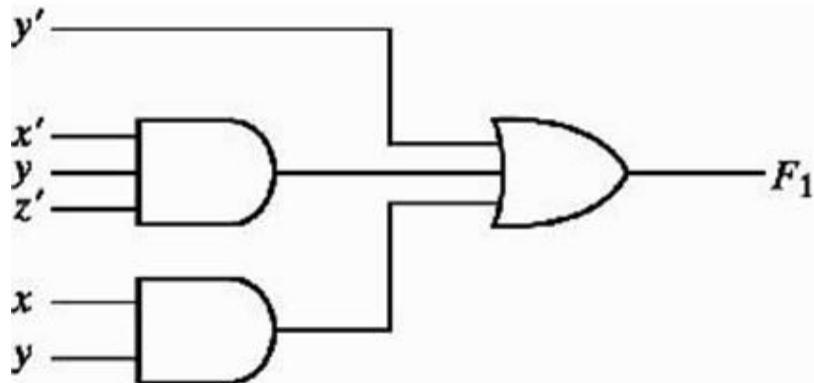
- Standard SOP form means **Standard Sum of Products** form. In this form, each product term need not contain all literals. So, the product terms may or may not be the minterms.
- The sum of products is a Boolean expression containing AND terms, called product terms, with one or more literals each. The sum denotes the ORing of these terms. An example of a function expressed as a sum of products is
 $F1 = y + xy + xy'z$
- The expression has three product terms, with one, two, and three literals. Their sum is, in effect, an OR operation.

Standard POS form

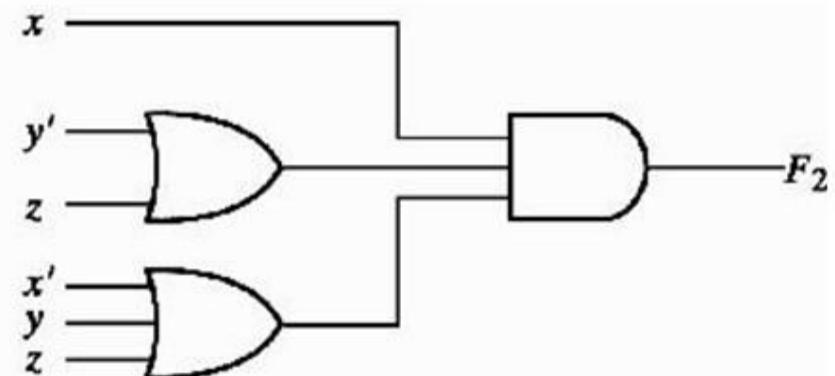
- Standard POS form means **Standard Product of Sums** form.
- *A product of sums is a Boolean expression containing OR terms, called sum terms.*
- Each term may have any number of literals. The *product denotes the ANDing of these terms.*

Three- and two-level implementation of Standard forms

- Another way to express Boolean functions is in **standard form**.
 1. Sum of products(SOP): $F_1 = y' + xy + x'yz'$
 2. Product of sums(POS): $F_2 = x(y' + z)(x' + y + z')$



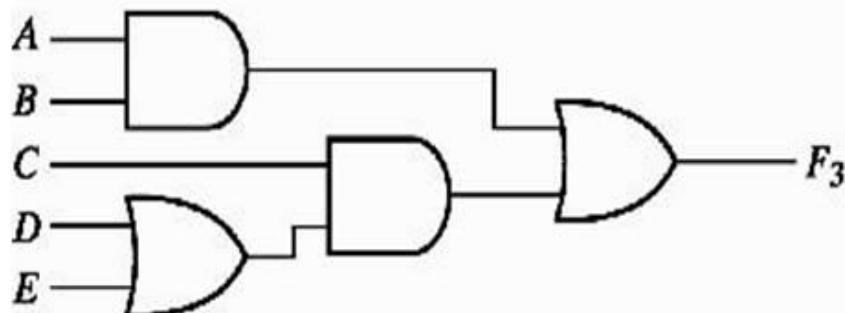
(a) Sum of Products



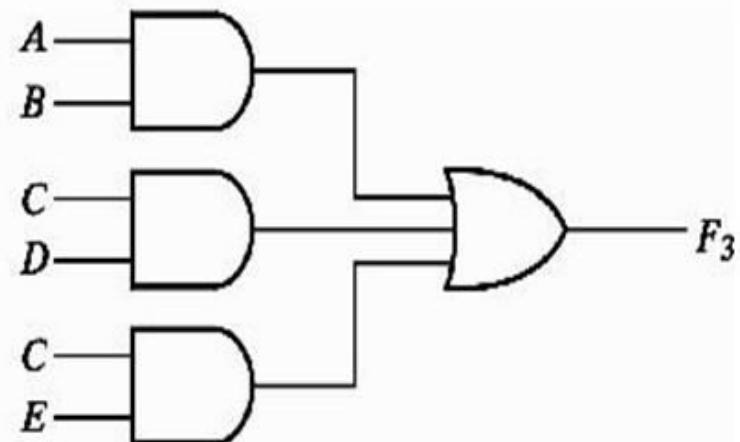
(b) Product of Sums

Standard forms

- F_3 is a non-standard form, neither in SOP nor in POS.
- F_3 can change to a standard form by using distributive law.



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



$$(b) AB + CD + CE$$

HDL Description of Boolean Function

- Write HDL Description of the function

$$F = AB + CD + CE$$

- // Verilog model: Circuit with Boolean expressions

```
module Circuit_CA ( F, A, B, C,D,E);
output F;
input A, B, C, D, E;
assign F = (A && B) || (C && D) || (C && E) ;
endmodule
```

HDL Description of Boolean Function

- Write HDL Description of the function

$$F = AB + C(D+E)$$

- // Verilog model: Circuit with Boolean expressions

```
module Circuit_CA ( F, A, B, C, D, E);
output F;
input A, B, C, D, E;
assign F = (A && B) || (C && (D || E)) ;
endmodule
```

INTEGRATED CIRCUITS

INTEGRATED CIRCUITS

- An **integrated circuit (IC)** is fabricated on a die of a silicon semiconductor crystal, called a *chip*, *containing the electronic components for constructing digital gates*.
- The various gates are interconnected inside the chip to form the required circuit. The chip is mounted in a ceramic or plastic container, and connections are welded to external pins to form the integrated circuit.

INTEGRATED CIRCUITS

- The number of pins may range from 14 on a small IC package to several thousand on a larger package.
- Each IC has a numeric designation printed on the surface of the package for identification.
- Vendors provide data books, catalogs, and Internet websites that contain descriptions and information about the ICs that they manufacture.

Levels of Integration

- Digital ICs are often categorized according to the complexity of their circuits, as measured by the number of logic gates in a single package.
- The differentiation between those chips which have a few internal gates and those having hundreds of thousands of gates is made by customary reference to a package as being either a **small-, medium-, large-, or very large-scale integration device**.
- ***Small-scale integration (SSI)*** devices contain several independent gates in a single package. The inputs and outputs of the gates are connected directly to the pins in the package. The number of gates is usually fewer than 10 and is limited by the number of pins available in the IC.

Levels of Integration

- **Medium-scale integration (MSI)** devices have a complexity of approximately 10 to 1,000 gates in a single package. They usually perform specific elementary digital operations.
- Example of MSI digital functions are decoders, adders, multiplexers , registers and counters.
- **Large-scale integration (LSI)** devices contain thousands of gates in a single package. They include digital systems such as processors, memory chips, and programmable logic devices.

Levels of Integration

- ***Very large-scale integration (VLSI)*** devices now contain millions of gates within a single package.
- Examples are large memory arrays and complex microcomputer chips.
- Because of their small size and low cost, VLSI devices have revolutionized the computer system design technology, giving the designer the capability to create structures that were previously uneconomical to build.

Digital Logic Families

- Digital integrated circuits are classified not only by their complexity or logical operation, but also by the **specific circuit technology** to which they belong.
- The circuit technology is referred to as a ***digital logic family***. *Each logic family has its own basic electronic circuit upon which more complex digital circuits and components are developed. The basic circuit in each technology is a NAND, NOR, or inverter gate.*

Digital Logic Families

- The electronic components employed in the construction of the basic circuit are usually used to name the technology. Many different logic families of digital integrated circuits have been introduced commercially. The following are the most popular:
 - TTL transistor-transistor logic;
 - ECL emitter-coupled logic;
 - MOS metal-oxide semiconductor;
 - CMOS complementary metal-oxide semiconductor

Digital Logic Families

- TTL is a logic family that has been in use for 50 years and is considered to be standard.
- ECL has an advantage in systems requiring high-speed operation.
- MOS is suitable for circuits that need **high component density**, and CMOS is preferable in systems requiring **low power consumption**, such as digital cameras, personal media players, and other handheld portable devices. Low power consumption is essential for VLSI design; therefore, CMOS has become the dominant logic family, while TTL and ECL continue to decline in use.

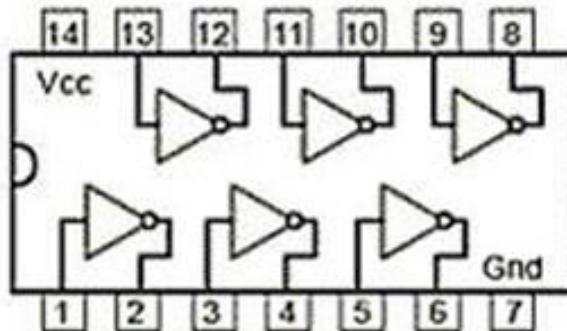
DIGITAL LOGIC GATES

- Since Boolean functions are expressed in terms of **AND, OR, and NOT operations**, it is easier to implement a Boolean function with these type of gates. Still, the possibility of constructing gates for the other logic operations is of practical interest.
- Factors to be weighed in considering the construction of other types of logic gates are
 - (1) The feasibility and economy of producing the gate with physical components
 - (2) The possibility of extending the gate to more than two inputs
 - (3) The basic properties of the binary operator, such as commutativity and associativity
 - (4) The ability of the gate to implement Boolean functions alone or in conjunction with other gates.

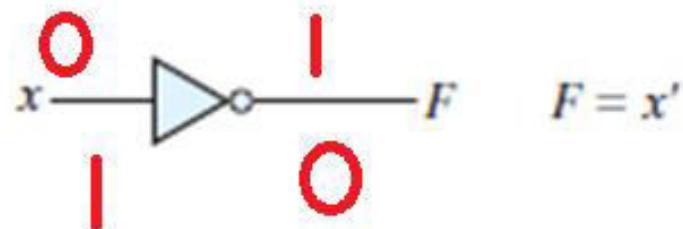
Digital Logic gates

NOT Gate

x	$F = x'$
0	1
1	0



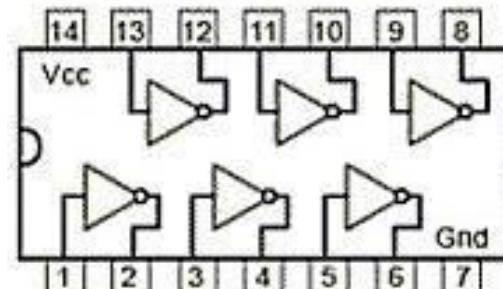
7404 Hex NOT Gates
(Inverters)



Digital Logic gates

NOT Gate

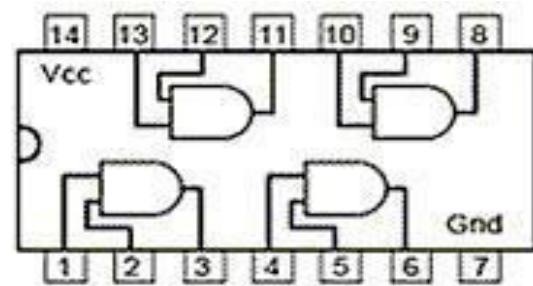
x	$F = x'$
0	1
1	0



7404 Hex NOT Gates
(Inverters)



A	B	$F = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

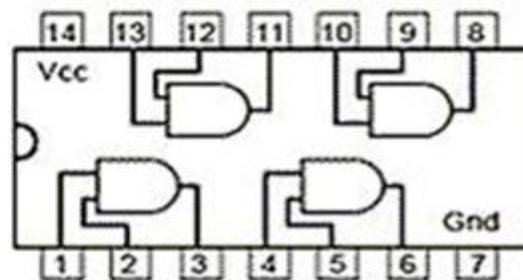


7408 Quad 2 input
AND Gates

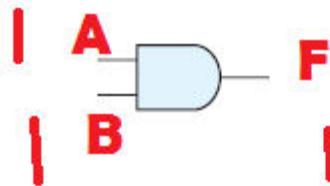
Digital Logic gates

AND Gate

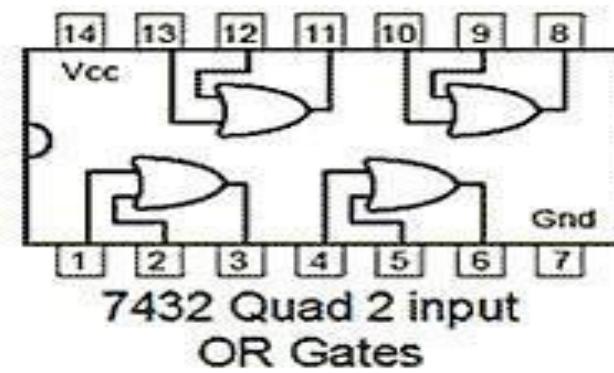
A	B	$F = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1



7408 Quad 2 input
AND Gates



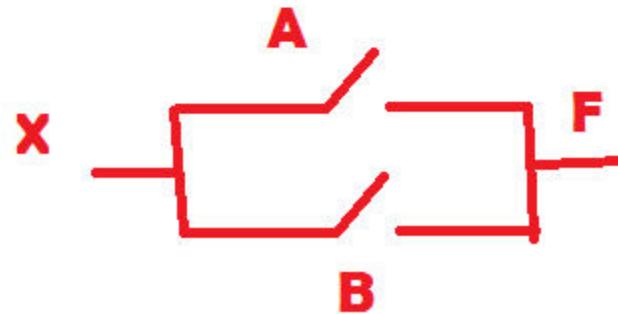
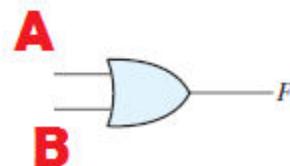
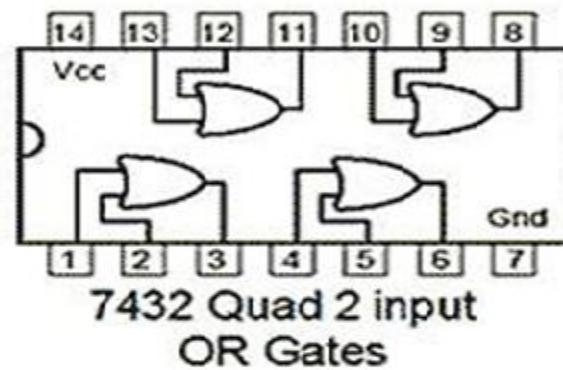
A	B	$F = A+B$
0	0	0
0	1	1
1	0	1
1	1	1



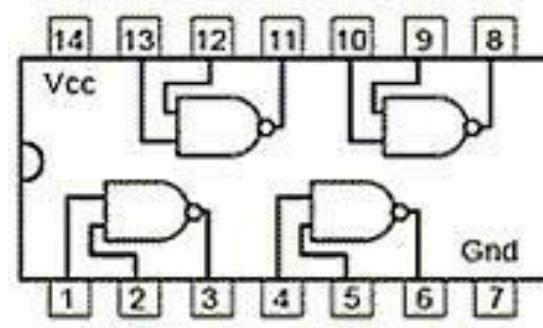
Digital Logic gates

2 Input OR Gate

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



A	B	F = (A.B)'
0	0	1
0	1	1
1	0	1
1	1	0

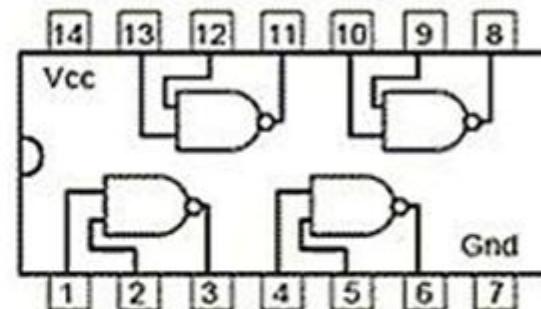


**7400 Quad 2 input
NAND Gates**

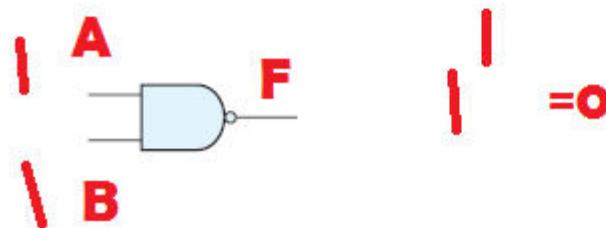
Digital Logic gates

2 Input NAND Gate

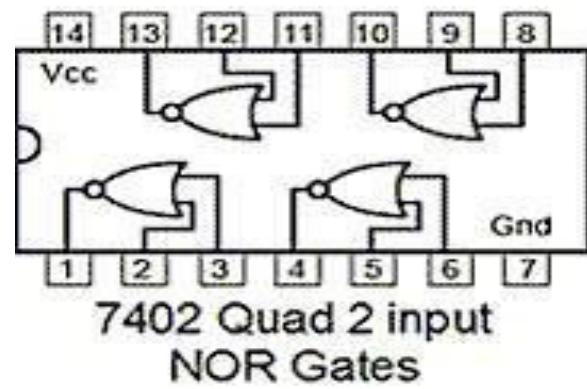
A	B	$F = (A \cdot B)'$
0	0	1
0	1	1
1	0	1
1	1	0



7400 Quad 2 input
NAND Gates



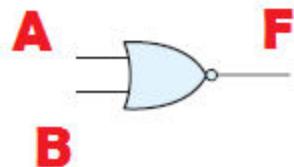
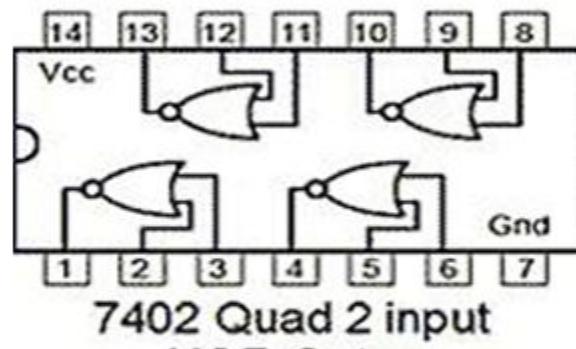
A	B	$F = (A+B)'$
0	0	1
0	1	0
1	0	0
1	1	0



Digital Logic gates

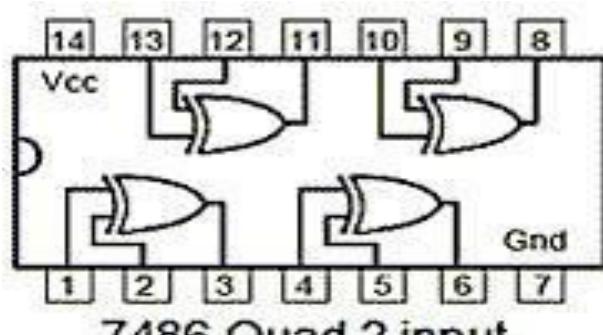
2 Input NOR Gate

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



$$\oplus$$

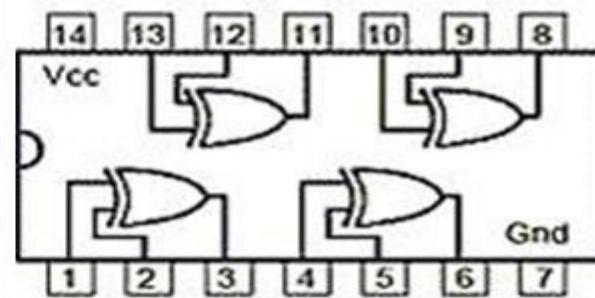
A	B	$F = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0



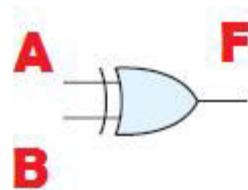
Digital Logic gates

2 Input XOR Gate

A	B	$F = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0



7486 Quad 2 input
XOR Gates



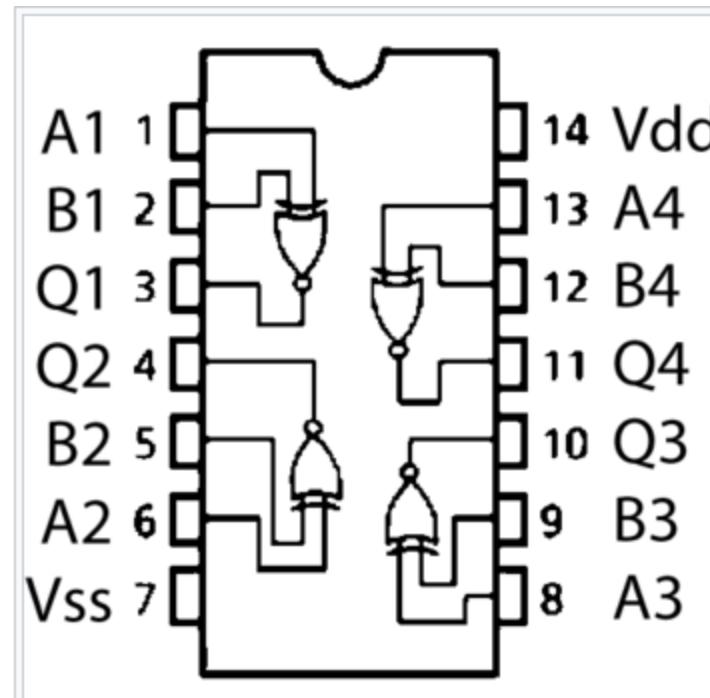
$$A' B + A B'$$

Exclusive-NOR or equivalence



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

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



SN74LS266

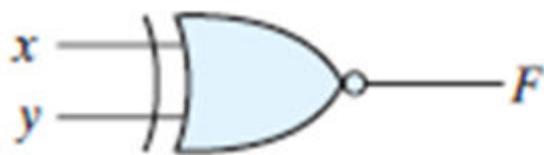
Quad 2 input XNOR gates

Digital Logic gates

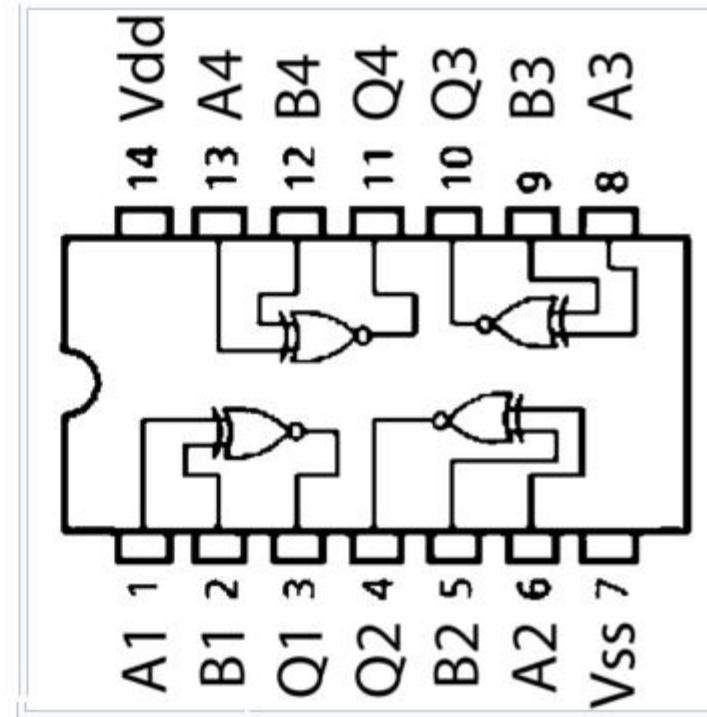
2 Input XNOR Gate

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

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



SN74LS266

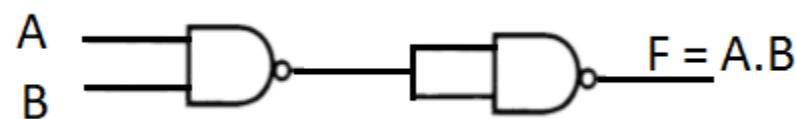


Quad 2 input XNOR gates

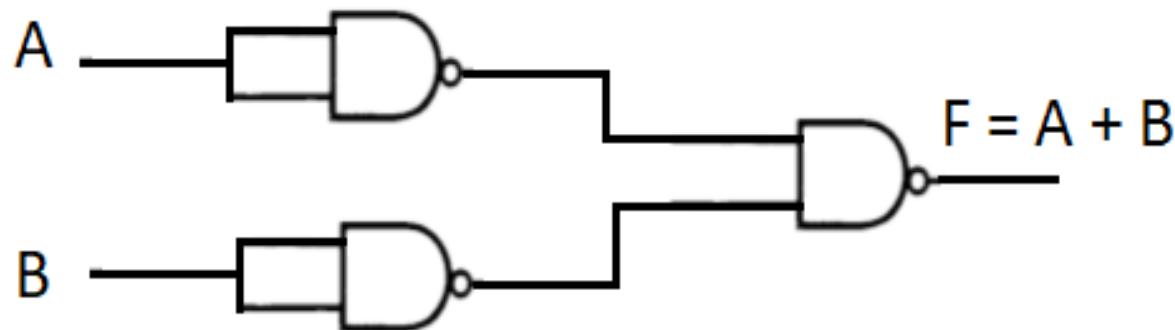
Using a single 7400 IC, connect a circuit that produces an inverter



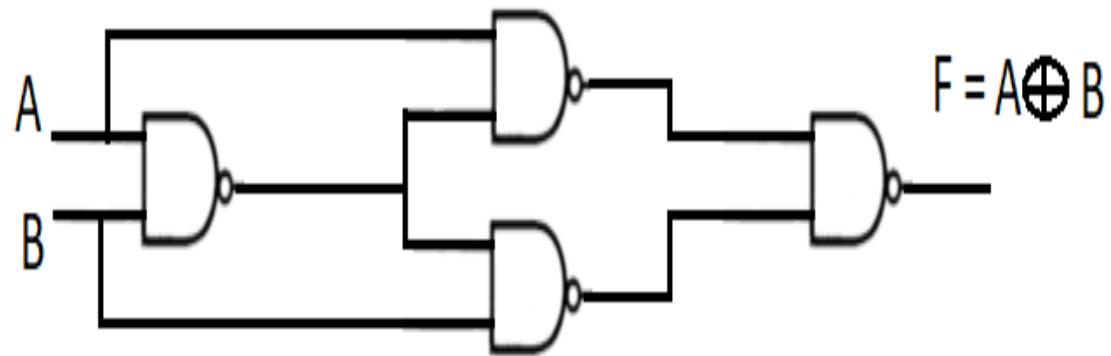
Using a single 7400 IC, connect a circuit that produces a two-input AND gate



Using a single 7400 IC, connect a circuit that produces a two-input OR gate



Using a single 7400 IC, connect a circuit
that produces a two-input EX-OR gate

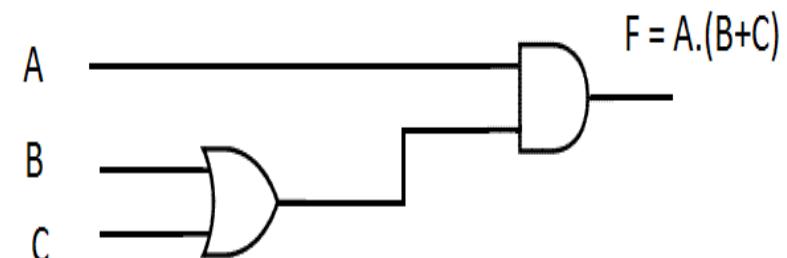


Implement the Boolean function:

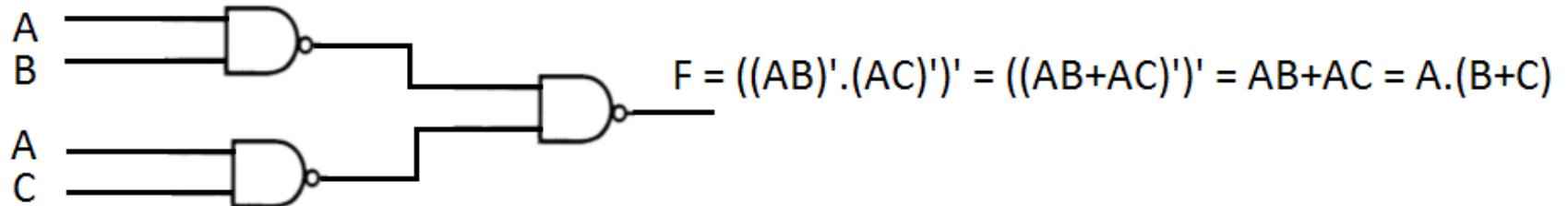
$$F = A \cdot (B + C)$$

Circuit Diagram

A	B	C	F=A.(B+C)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Circuit Diagram using NAND gates



A	B	C	$F=((AB)' \cdot (AC)')'$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Optimization of Digital Circuits using K Map

INTRODUCTION

- *Gate-level minimization is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit.* This task is well understood, but is difficult to execute by manual methods when the logic has more than a few inputs.
- Fortunately, **computer-based logic synthesis tools** can minimize a large set of Boolean equations efficiently and quickly.
- Nevertheless, it is important that a designer understand the underlying **mathematical description and solution** of the problem.
- This lecture serves as a foundation for your understanding of that important topic and will enable you to execute a **manual design** of simple circuits, preparing you for skilled use of modern design tools.

Karnaugh map or K-map

- The complexity of the digital logic gates that implement a Boolean function is directly related to the **complexity of the algebraic expression** from which the function is implemented.
- Although the truth table representation of a function is unique, when it is expressed algebraically it can appear in many different, but equivalent, forms.
- Boolean expressions may be simplified by algebraic means but this procedure of minimization is awkward because it **lacks specific rules to predict each succeeding step** in the manipulative process.
- The **map method** presented here provides a **simple, straightforward procedure** for minimizing Boolean functions.
- This method may be regarded as a pictorial form of a truth table. The map method is also known as the **Karnaugh map or K-map** .

Karnaugh map or K-map

- A **K-map** is a diagram made up of **squares**, with each square representing one minterm of the function that is to be minimized.
- Since any Boolean function can be expressed as a **sum of minterms**, it follows that a Boolean function is recognized graphically in the map from the **area enclosed by those squares whose minterms** are included in the function.
- In fact, the map presents a **visual diagram** of all possible ways a function may be expressed in **standard form**.
- By recognizing various patterns, the user can derive alternative algebraic expressions for the same function, from which the simplest can be selected.

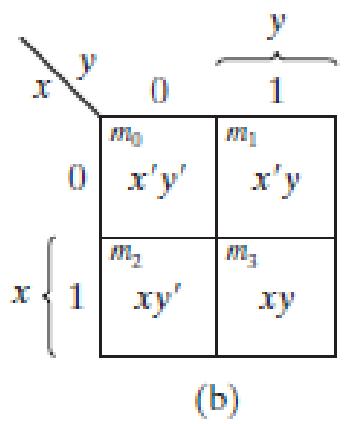
Karnaugh map or K-map

- The simplified expressions produced by the map are always in one of the two standard forms: **sum of products** or **product of sums**.
- It will be assumed that the simplest algebraic expression is an algebraic expression with a **minimum number of terms** and with the **smallest possible number of literals** in each term.
- This expression produces a circuit diagram with a **minimum number of gates** and the **minimum number of inputs to each gate**.

Two-Variable K-Map

m_0	m_1
m_2	m_3

(a)



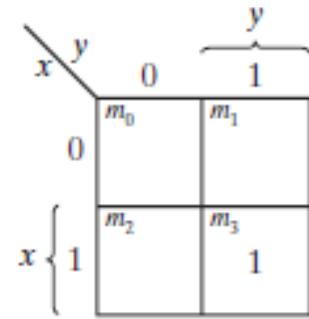
(b)

- There are four minterms for two variables; hence, the map consists of four squares, one for each minterm.
- The map drawn in figure shows the relationship between the squares and the two variables x and y .
- *The 0 and 1 marked in each row and column designate the values of variables.*
- Variable x appears primed in row 0 and unprimed in row 1. Similarly, y appears primed in column 0 and unprimed in column 1.

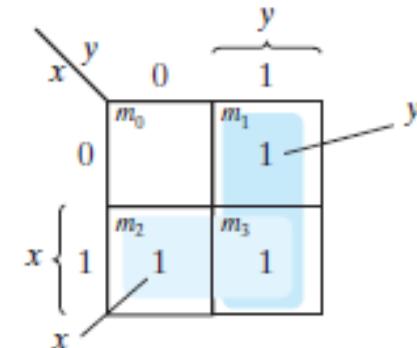
Representation of functions in the map

- If we mark the squares whose minterms belong to a given function, the two-variable map becomes another useful way to represent any one of the 16 Boolean functions of two variables.
- As an example, the function xy is shown in Fig. (a). Since xy is equal to m_3 , a 1 is placed inside the square that belongs to m_3 .
- Similarly, the function $x + y$ is represented in the map of Fig. (b) by three squares marked with 1's. These squares are found from the minterms of the function:

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



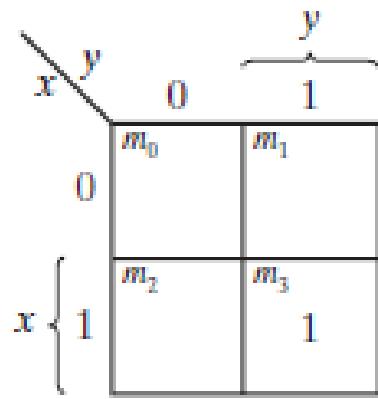
(a) xy



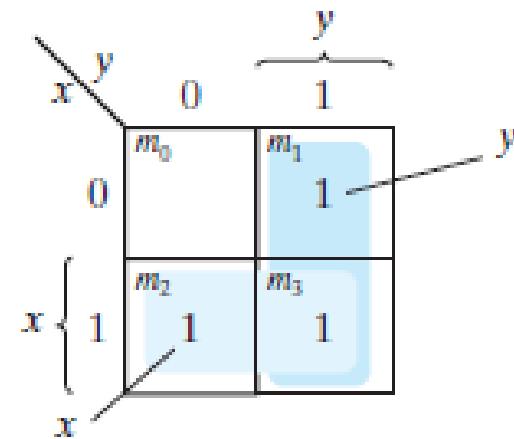
(b) $x + y$

Two-Variable K-Map

- $F = X'Y + XY' + XY = Y(X' + X) + XY' = Y + XY' = (X + Y)(Y + Y') = X + Y$

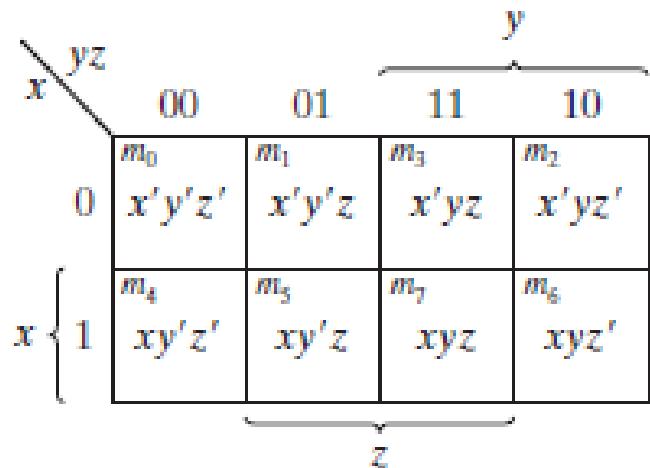


(a) xy



(b) $x + y$

Three-Variable K-Map



- There are eight minterms for three binary variables; therefore, the map consists of eight squares.
- Note that the minterms are arranged, not in a binary sequence, but in a sequence similar to the Gray code.
- The characteristic of this sequence is that **only one bit changes in value from one adjacent column to the next**.
- The map drawn is marked with numbers in each row and each column to show the relationship between the squares and the three variables.

Three-Variable K-Map

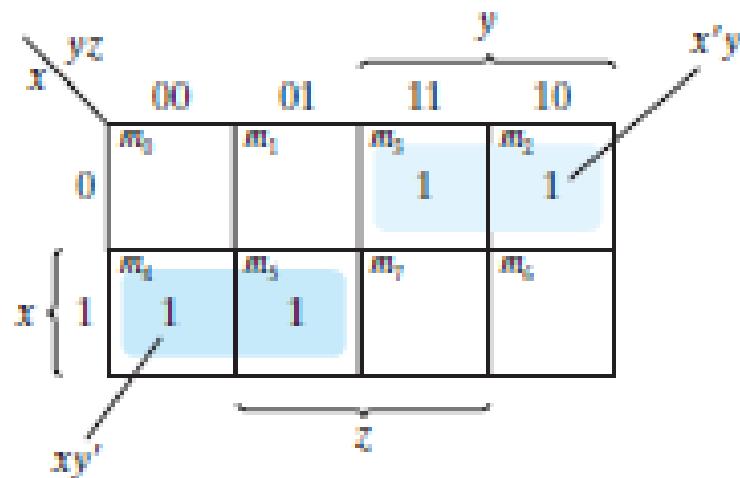
- For example, the square assigned to m_5 corresponds to row 1 and column 01.
- When these two numbers are concatenated, they give the binary number 101, whose decimal equivalent is 5. Each cell of the map corresponds to a unique minterm, so another way of looking at square $m_5 = xy'z$ is to consider it to be in the row marked x and the column belonging to $y'z$ (column 01).
- Note that there are four squares in which each variable is equal to 1 and four in which each is equal to 0.
- The variable appears unprimed in the former four squares and primed in the latter. For convenience, we write the variable with its letter symbol under the four squares in which it is unprimed.

Three-Variable K-Map

- The number of adjacent squares that may be combined must always represent a number that is a power of two, such as 1, 2, 4, and 8. As more adjacent squares are combined, we obtain a product term with fewer literals.
- **One square represents one minterm, giving a term with three literals.**
- **Two adjacent squares represent a term with two literals.**
- **Four adjacent squares represent a term with one literal.**
- **Eight adjacent squares encompass the entire map and produce a function that is always equal to 1.**

Simplify the Boolean function

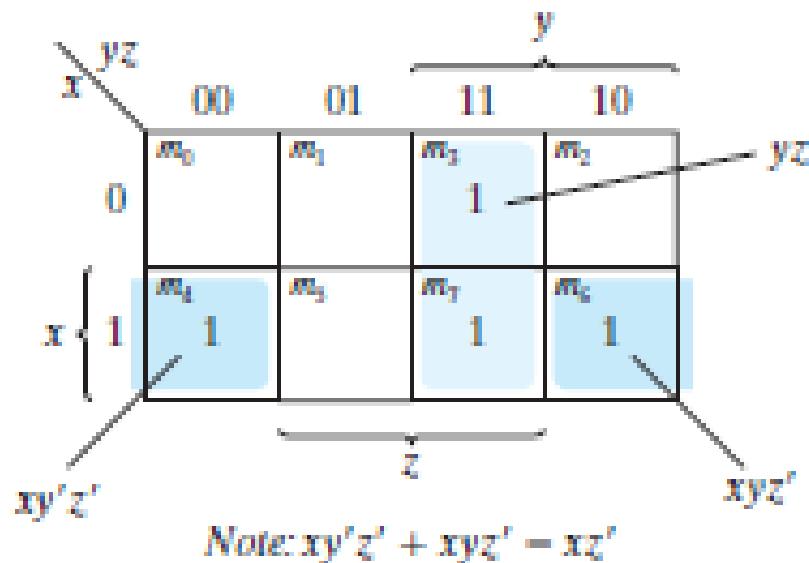
$$F(x, y, z) = \Sigma(2, 3, 4, 5)$$



$$F(x, y, z) = \Sigma(2, 3, 4, 5) = x'y + xy'$$

Simplify the Boolean function

$$F(x, y, z) = \Sigma(3, 4, 6, 7)$$

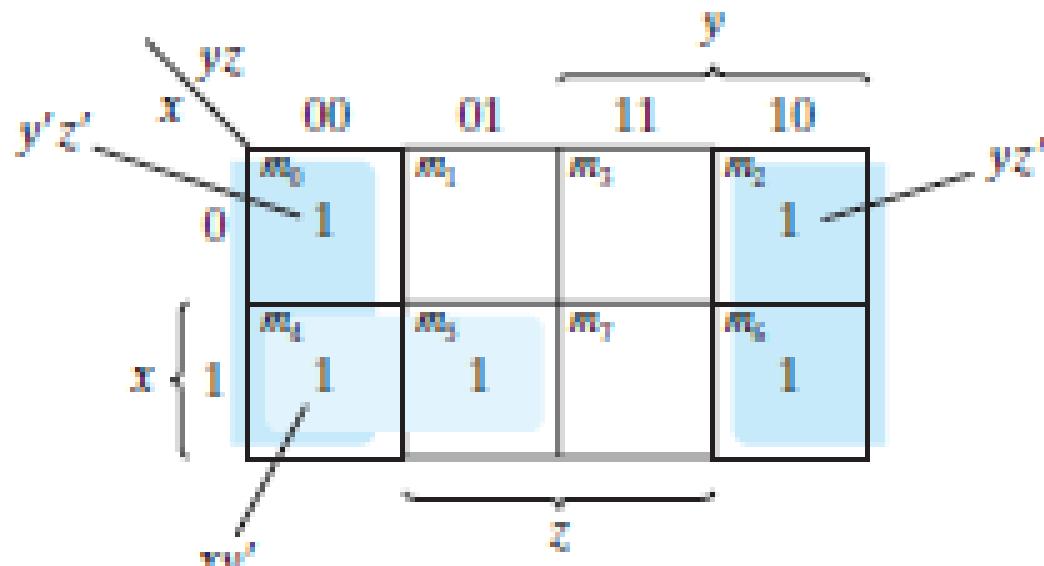


$$F = yz + xz'$$

Simplify the Boolean function

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

$$F = z' + xy'$$



$$\text{Note: } y'z' + yz' = z'$$

For the Boolean function

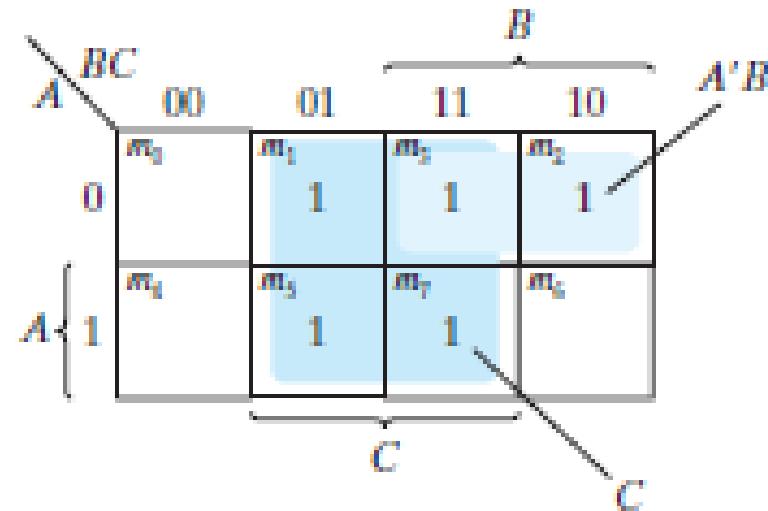
$$F = A'C + A'B + AB'C + BC$$

- (a) Express this function as a sum of minterms.
- (b) Find the minimal sum-of-products expression.

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

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

$$F = C + A'B$$

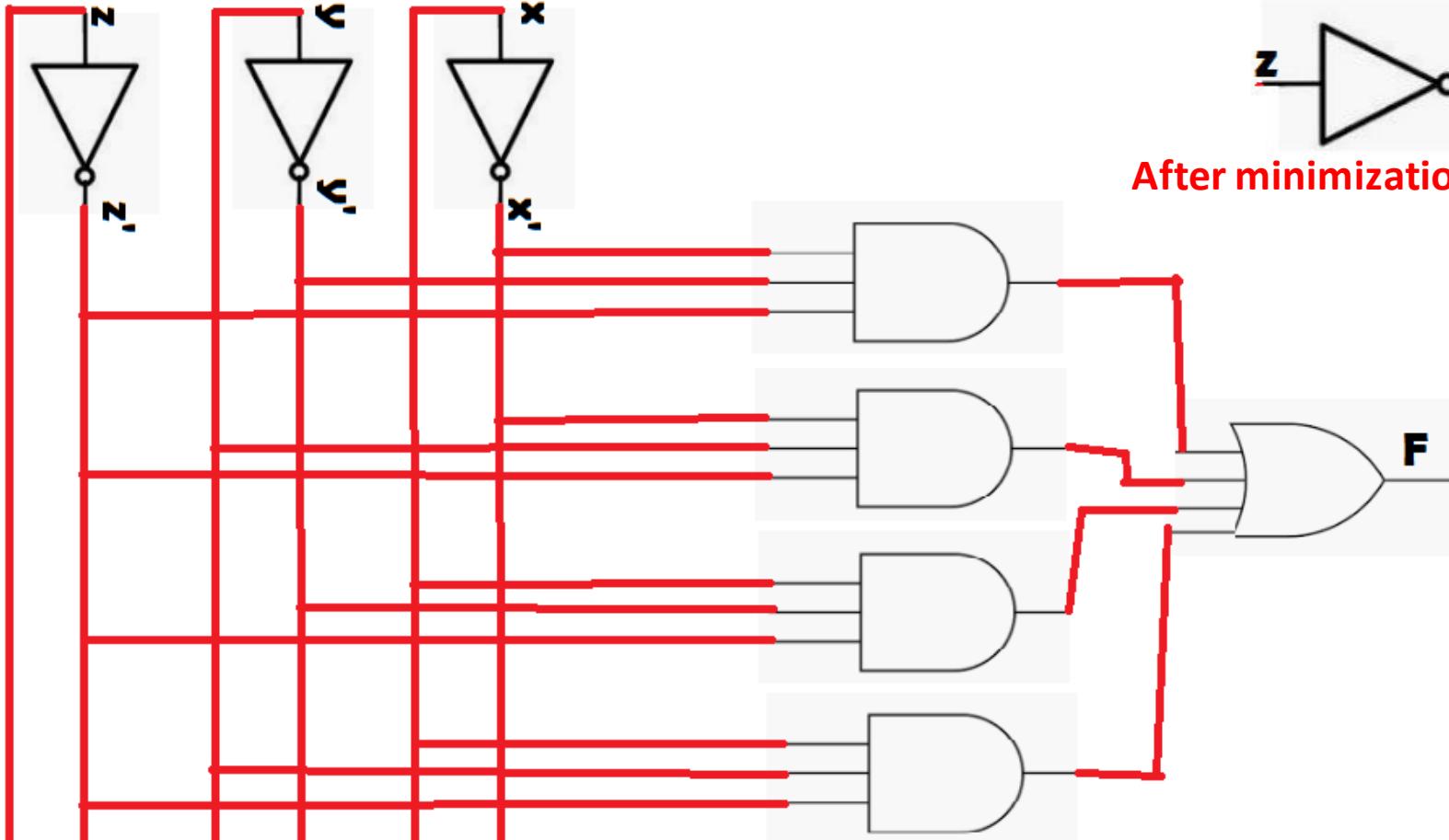


Example: Minimize the logical sum of the four adjacent minterms 0, 2, 4, and 6

$$\begin{aligned}m_0 + m_2 + m_4 + m_6 &= x'y'z' + x'yz' + xy'z' + xyz' \\&= x'z'(y' + y) + xz'(y' + y) \\&= x'z' + xz' = z'(x' + x) = z'\end{aligned}$$

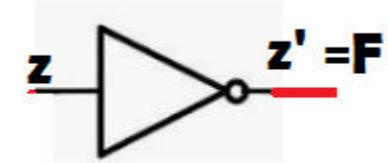
		00	01	11	10	
		0	m ₀	m ₁	m ₃	m ₂
		1	m ₄	m ₅	m ₇	m ₆

CIRCUIT OPTIMIZATION



Before minimization of function

After minimization of function



HDL Description of circuit

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

// Verilog model: Circuit with Boolean expressions

```
module Circuit_Boolean (F, X, Y, Z);
```

```
Output F;
```

```
Input X, Y, Z ;
```

```
assign F = ((!X) && (!Y) && (!Z) ) || ((!X) && Y && (!Z) ) || (X &&  
    (!Y) && (!Z) ) || (X && Y && (!Z) );
```

```
endmodule
```

FOUR-VARIABLE K-MAP

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

- The map for Boolean functions of four binary variables (w, x, y, z) is shown in Figure. The map is drawn to show the relationship between the squares and the four variables.
- The rows and columns are numbered in a **Gray code** sequence, with only one digit changing value between two adjacent rows or columns.
- The minterm corresponding to each square can be obtained from the concatenation of the row number with the column number. For example, the numbers of the third row (11) and the second column (01), when concatenated, give the binary number 1101, the binary equivalent of decimal 13. Thus, the square in the third row and second column represents minterm m_{13} .

FOUR-VARIABLE K-MAP

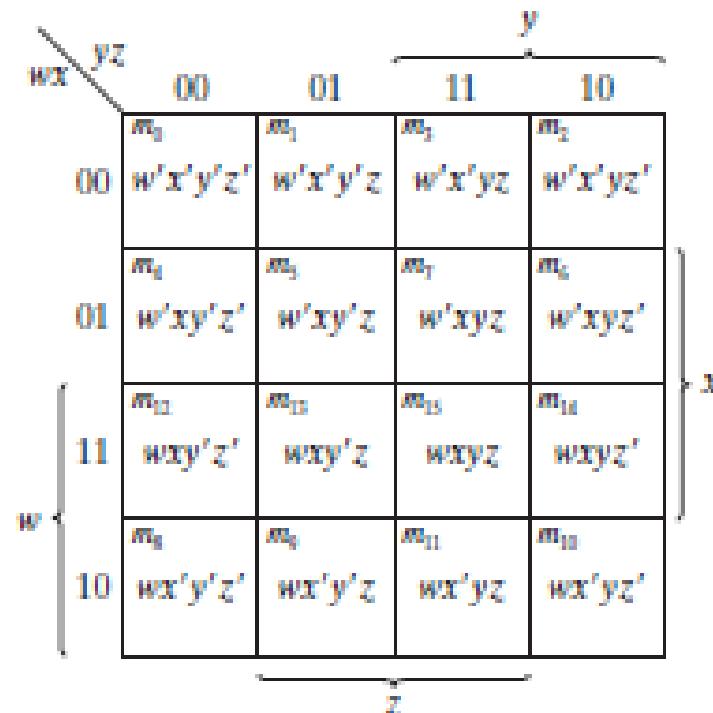
- The map minimization of four-variable Boolean functions is similar to the method used to minimize three-variable functions.
- Adjacent squares are defined to be squares next to each other. In addition, the map is considered to lie on a surface with the top and bottom edges, as well as the right and left edges, touching each other to form adjacent squares. For example, *m0 and m2 form adjacent squares, as do m3 and m11.*
- *The combination of adjacent squares that is useful during the simplification process is easily determined from inspection of the four-variable map:*

FOUR-VARIABLE K-MAP

- One square represents one minterm, giving a term with four literals.
- Two adjacent squares represent a term with three literals.
- Four adjacent squares represent a term with two literals.
- Eight adjacent squares represent a term with one literal.
- Sixteen adjacent squares produce a function that is always equal to 1.

FOUR-VARIABLE K-MAP

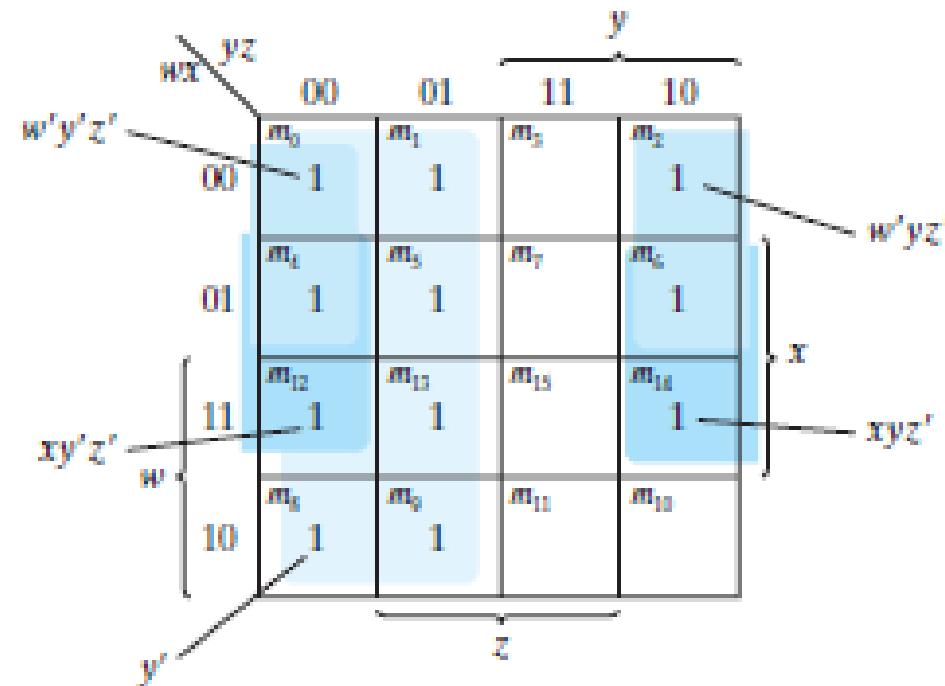
w	x	y	z	
0	0	0	0	m0
0	0	0	1	m1
0	0	1	0	m2
0	0	1	1	m3
0	1	0	0	m4
0	1	0	1	m5
0	1	1	0	m6
0	1	1	1	m7
1	0	0	0	m8
1	0	0	1	m9
1	0	1	0	m10
1	0	1	1	m11
1	1	0	0	m12
1	1	0	1	m13
1	1	1	0	m14
1	1	1	1	m15



Simplify the Boolean function

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

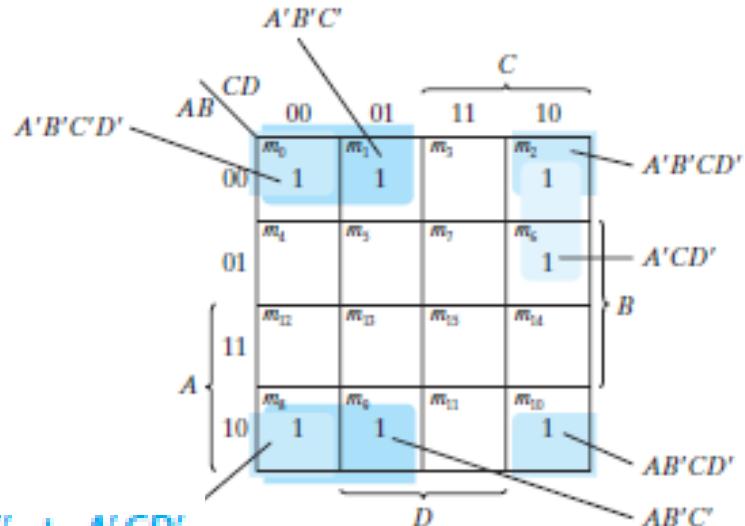
$$F = y' + w'z' + xz'$$



Note: $w'y'z' + w'yz' = w'z'$
 $xy'z' + xyz' = xz'$

Simplify the Boolean function

$$F = A'B'C' + B'CD' + A'BCD' + AB'C'$$

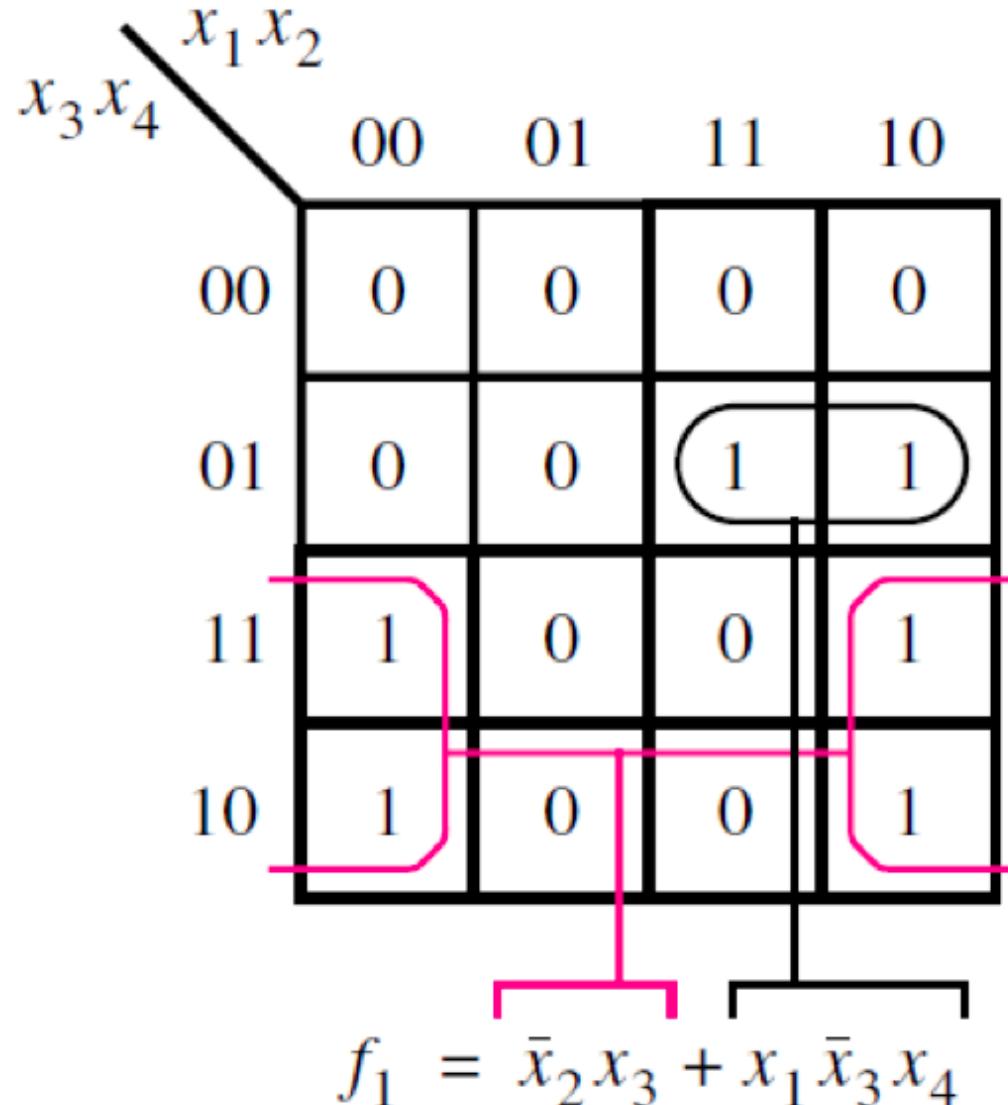


$$A'B'C' + B'CD' + A'BCD' + AB'C' = B'D' + B'C' + A'CD'$$

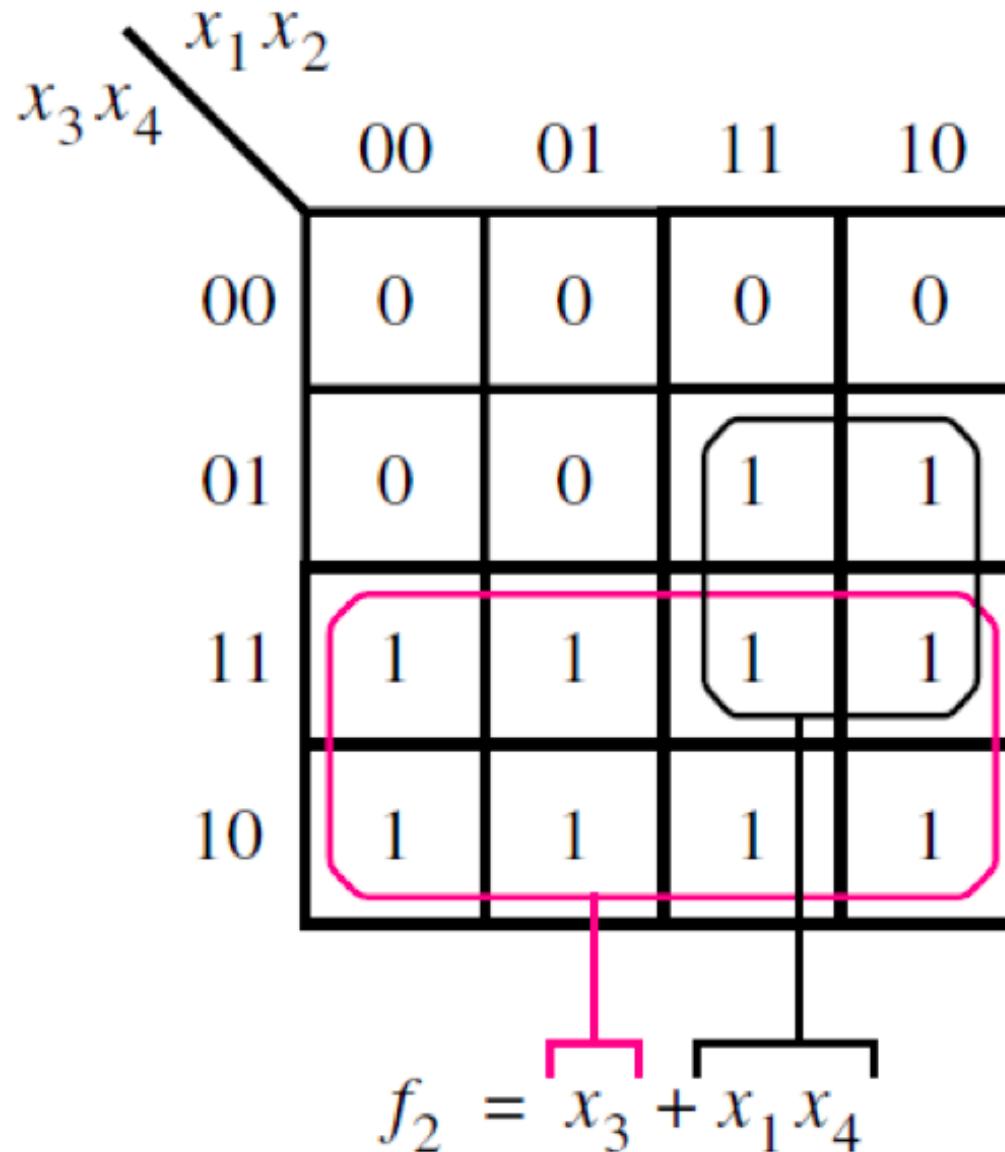
...ie:
A'B'C'D' + A'B'CD' - A'B'D'
AB'C'D' + AB'CD' - AB'D'
A'B'D' + AB'D' - B'D'
A'B'C' + AB'C' - B'C'

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

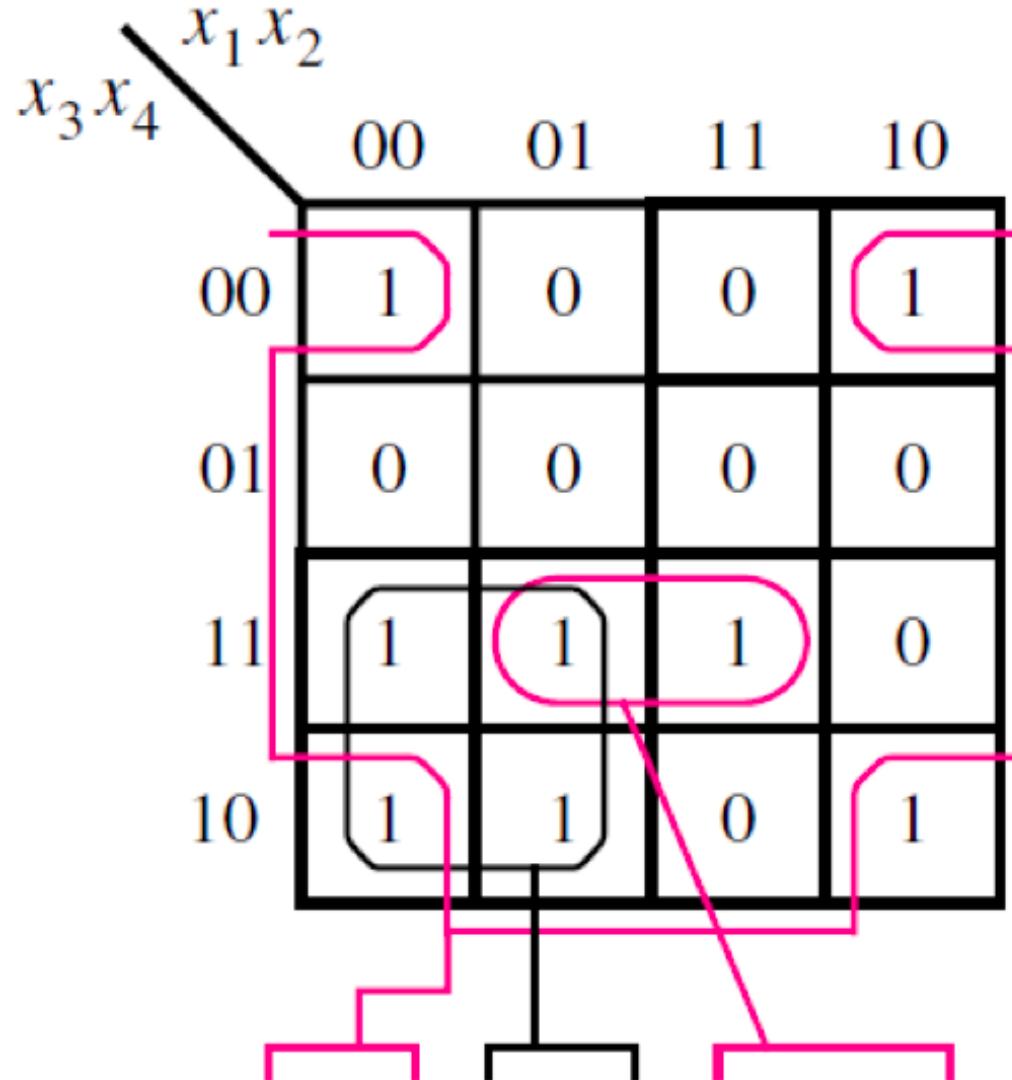
Example of a four-variable Karnaugh map



Example of a four-variable Karnaugh map

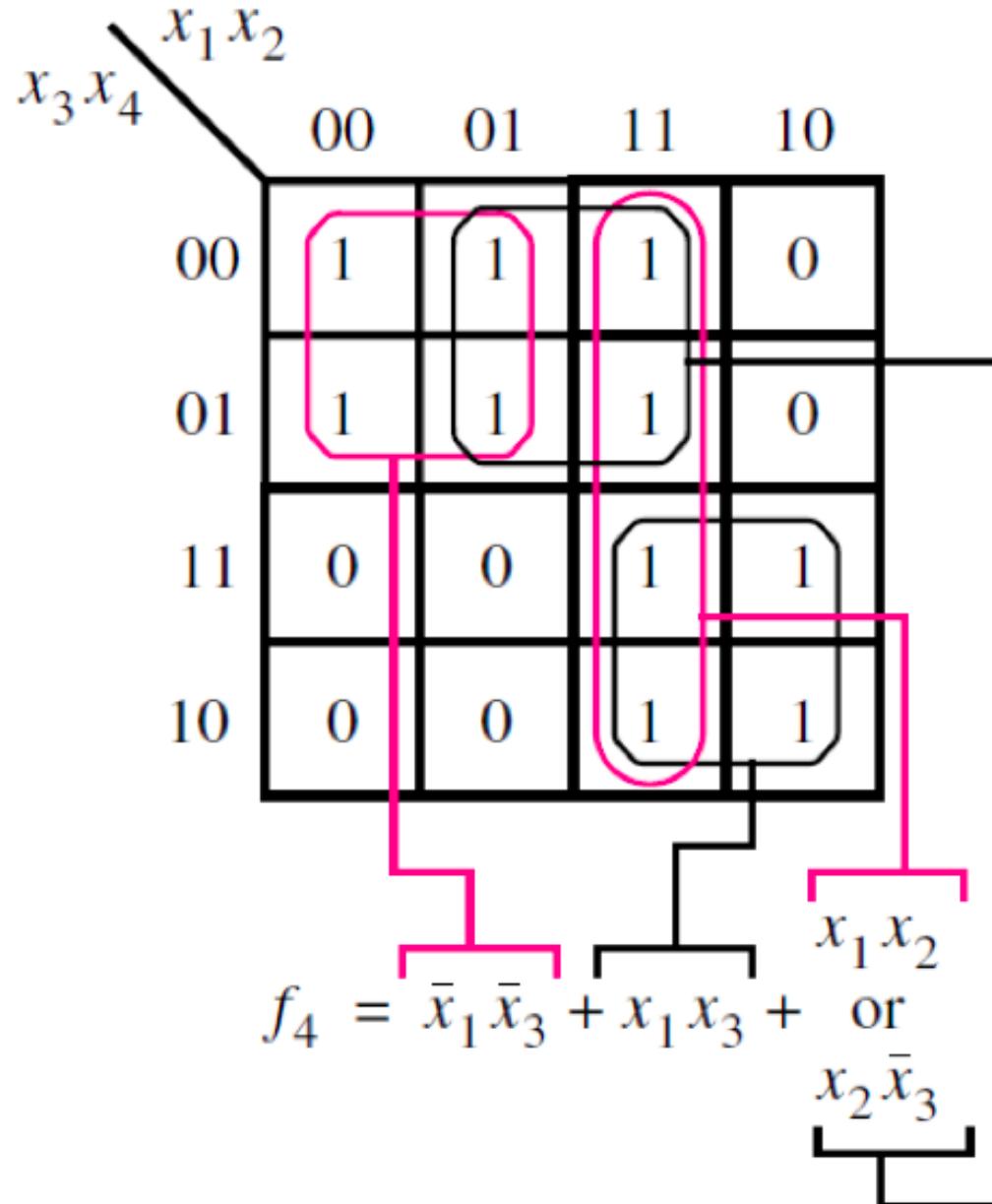


Example of a four-variable Karnaugh map



$$f_3 = \bar{x}_2 \bar{x}_4 + \bar{x}_1 x_3 + x_2 x_3 x_4$$

Example of a four-variable Karnaugh map



[Figure 2.54 from the textbook]

HDL Description of The simplified Function

$$F = X_1'X_3' + X_1X_3 + X_1X_2$$

// Verilog model: Circuit with Boolean expressions

module Circuit (F, X_1 , X_2 , X_3);

Output F;

Input X_1 , X_2 , X_3 ;

assign F = ((! X_1) && (! X_3)) || (X_1 && X_3) || (X_1 && X_2);

endmodule

HDL Description of The Simplified Function

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

```
// Verilog model: Circuit with Boolean expressions
module Circuit (F, A,B,C,D);
Output F;
Input A, B, C, D;
assign F = ((! B) && (! D)) || ((! B) && (! C)) || ((! A) && C && (! D));
endmodule
```

HDL Description of circuit

$$F = X_1'X_3' + X_1X_3 + X_1X_2$$

// Verilog model: Circuit with Boolean expressions

module Circuit (**F**, X_1 , X_2 , X_3);

Output **F**;

Input X_1 , X_2 , X_3 ;

assign **F** = ($(! X_1)$ **&&** ($! X_3$)) **||** (X_1 **&&** X_3) **||** (X_1 **&&** X_2);

endmodule

$$F = ABC' + AB'C'D + ACD + ACD'$$

$$\begin{aligned} F &= ABC'(D' + D) + AB'C'D + ACD(B + B') \\ &\quad + ACD'(B + B') \\ &= ABC' + AB'C'D + ACD + ACD' \quad \text{Using Property-1} \\ &= ABC' + AB'C'D + AC(D + D') \\ &= ABC' + AB'C'D + AC \quad \text{Using Property-1} \\ &= A(BC' + C) + AB'C'D \\ &= A(B + C) + AB'C'D \quad \text{Using Property-2} \\ &= AB + AC + AB'C'D \\ &= AB + AC + AC'D \quad \text{Using Property-2} \\ &= AB + AC + AD \quad \text{Using Property-2} \end{aligned}$$

Prime Implicants

- In choosing adjacent squares in a map, we must ensure that
 - (1) all the minterms of the function are covered when we combine the squares,
 - (2) the number of terms in the expression is minimized, and
 - (3) there are no redundant terms (i.e., minterms already covered by other terms).
- Sometimes there may be two or more expressions that satisfy the simplification criteria. The procedure for combining squares in the map may be made more systematic if we understand the meaning of two special types of terms.

Prime Implicants

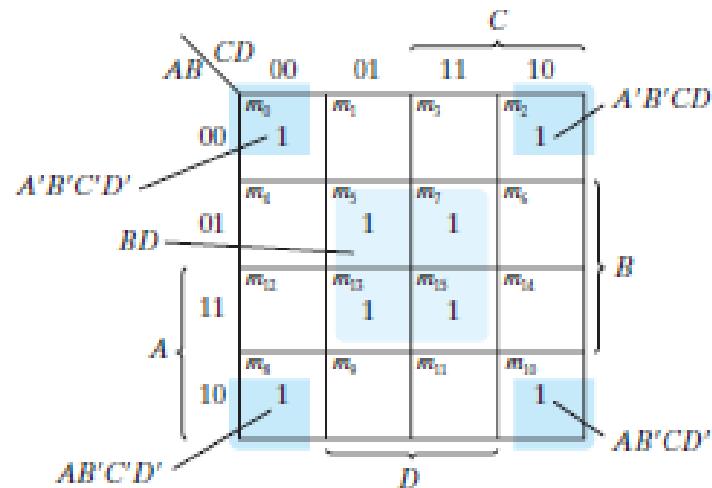
- A *prime implicant is a product term obtained by combining the maximum possible number of adjacent squares in the map. If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be essential.*
- The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares.

Prime Implicants

- This means that 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 if they are not within a group of eight adjacent squares, and so on.
- The essential prime implicants are found by looking at each square marked with a 1 and checking the number of prime implicants that cover it. The prime implicant is essential if it is the only prime implicant that covers the minterm.

Consider the following four-variable Boolean function:

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

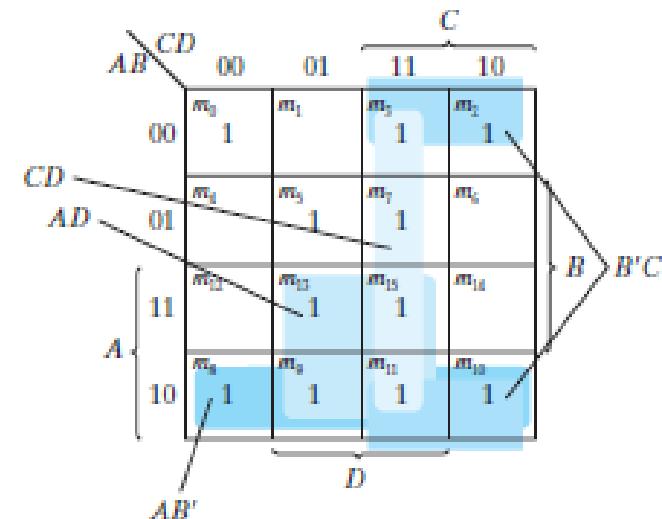


Note: $A'B'C'D' + A'B'CD' = A'B'D'$

$AB'C'D' + AB'CD' = AB'D'$

$A'B'D' + AB'D' = B'D'$

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



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

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

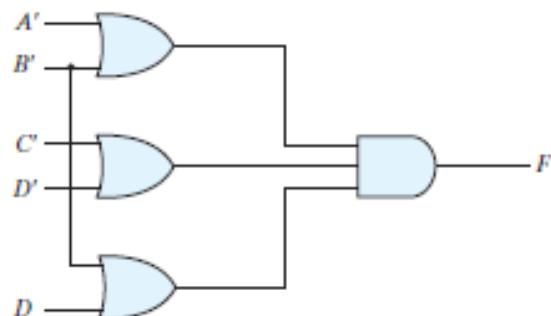
PRODUCT-OF-SUMS SIMPLIFICATION

- The procedure for obtaining a minimized function in product-of-sums form follows from the basic properties of Boolean functions. The 1's placed in the squares of the map represent the minterms of the function. The minterms not included in the standard sum-of-products form of a function denote the complement of the function.
- From this observation, we see that the complement of a function is represented in the map by the squares not marked by 1's. If we mark the empty squares by 0's and combine them into valid adjacent squares, we obtain a simplified sum-of-products expression of the complement of the function (i.e., of F).
- *The complement of F gives us back the function F in product-of-sums form (a consequence of DeMorgan's theorem). Because of the generalized DeMorgan's theorem, the function so obtained is automatically in product-of-sums form. The best way to show this is by example.*

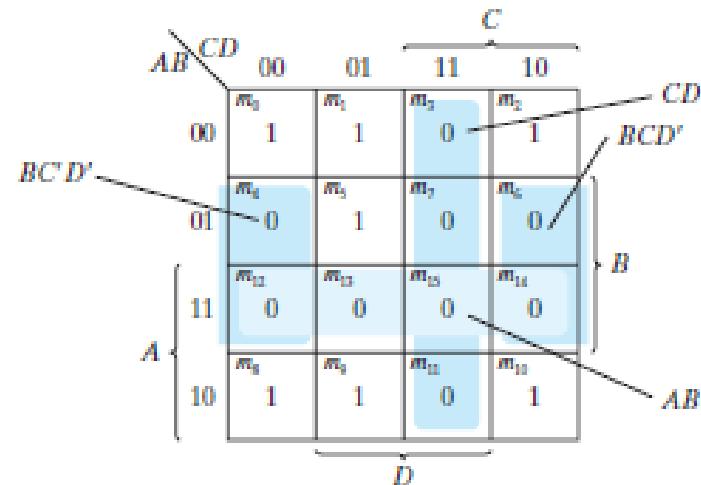
Simplify the following Boolean function into product-of-sums form

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

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



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



$$\text{Note: } BC'D' + BCD' = BD'$$

Practice Questions

1. Reduce the following equations

(a) $Y = AB'C + AB'C' + A'BC'$

(b) $Y = m(0,2,4,6,7)$

(c) $F(A,B,C) = A'C' + A'B + AB'C$

(d) $F(X,Y,Z) = m_0 + m_2 + m_5 + m_6 + m_7$

(e) $Y = A'BC' + ABC + ABC' + A'BC + AB'C$

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

Practice Questions

Reduce the following equations

(a) $F(W,X,Y,Z) = \sum (0,1,2,3,5,6,11,13)$

(b) $F(P,Q,R,S) = \sum (0,2,5,7,8,10,13,15)$

(c) $Y(A,B,C,D) = \sum (2,3,12,13,14,15)$

(d) $F(A,B,C,D) = \sum (7,13,14,15)$

(e) $Y(A,B,C,D) = \sum (4,6,7,15)$

LECTURE 13

DIGITAL LOGIC

Topics to be discussed

- Boolean function Simplification using K-map with don't care conditions
- HDL Description

DON'T-CARE CONDITIONS

➤ It often occurs that for certain input combinations, the value of output is unspecified either because the input combinations are invalid or because the precise value of the output is of no consequence. The combinations for which the values of the expression are not specified are called don't care combinations or optional combinations.

Examples:- In excess-3 code, 0000, 0001, 0010, 1101, 1110 and 1111 are unspecified and never occur. These are called don't cares. The four-bit binary code for the decimal digits has six combinations that are not used and consequently are considered to be unspecified.

- A don't-care minterm is a combination of variables whose logical value is not specified.
- Such a minterm cannot be marked with a 1 in the map, because it would require that the function always be a 1 for such a combination. Likewise, putting a 0 on the square requires the function to be 0.
- To distinguish the don't-care condition from 1's and 0's, an X is used. Thus, an X inside a square in the map indicates that we don't care whether the value of 0 or 1 is assigned to F for the particular minterm.

- During the process of design, using an SOP map, each don't care is treated as 1 if it is helpful in map reduction, otherwise it is treated as 0 and left alone.
- During the process of design using a POS map, each don't care is treated as 0, if it is useful in map reduction, otherwise it is treated as 1 and left alone.
- In choosing adjacent squares to simplify the function in a map, the don't-care minterms may be assumed to be either 0 or 1.
- When simplifying the function, we can choose to include each don't-care minterm with either the 1's or the 0's, depending on which combination gives the **simplest expression**.

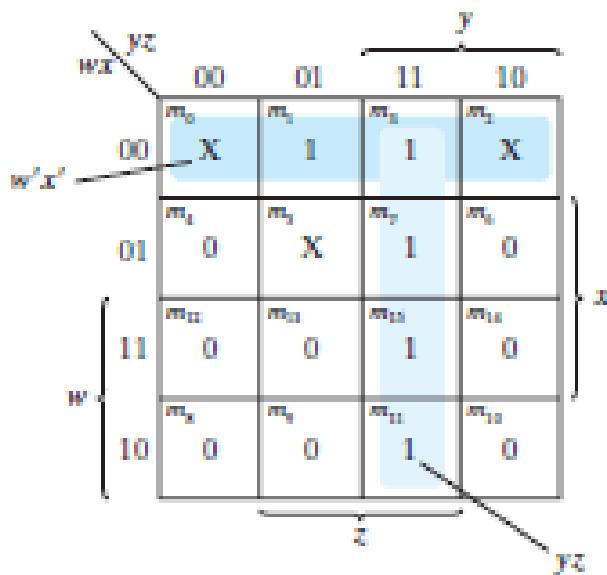
Example 1

Simplify the Boolean function

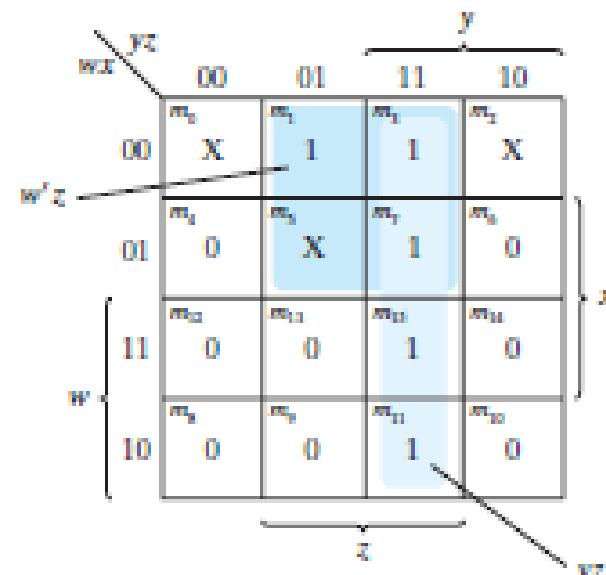
$$F(w, x, y, z) = \Sigma(1, 3, 7, 11, 15)$$

which has the don't-care conditions

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



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



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

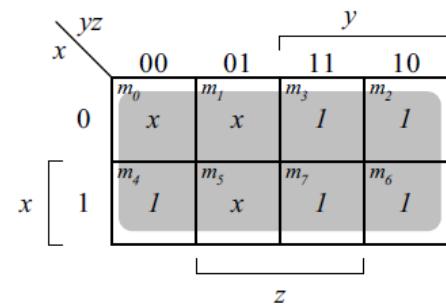
Either one of the preceding two expressions satisfies the conditions stated for this Example.

Example 2

Simplify the Boolean function F, together with don't-care conditions d, and then express the simplified function in sum-of-minterms forms.

$$F(x,y,z) = \sum(2, 3, 4, 6, 7)$$

$$d(x,y,z) = \sum(0, 1, 5)$$



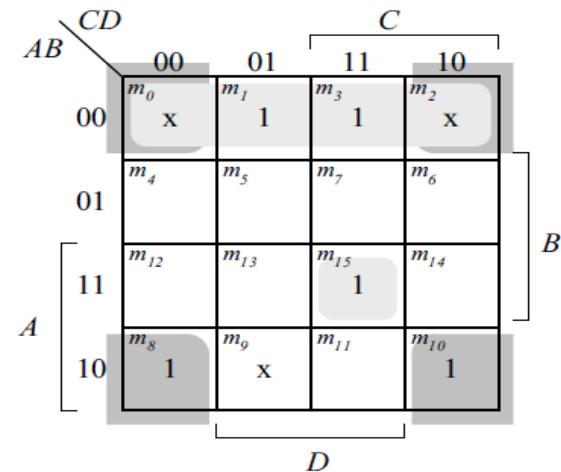
$$F=1$$

Example 3

Simplify the Boolean function F , together with don't-care conditions d , and then express the simplified function in sum-of-minterms forms.

$$F(A,B,C,D) = \sum(1, 3, 8, 10, 15)$$

$$d(A,B,C,D) = \sum(0, 2, 9)$$



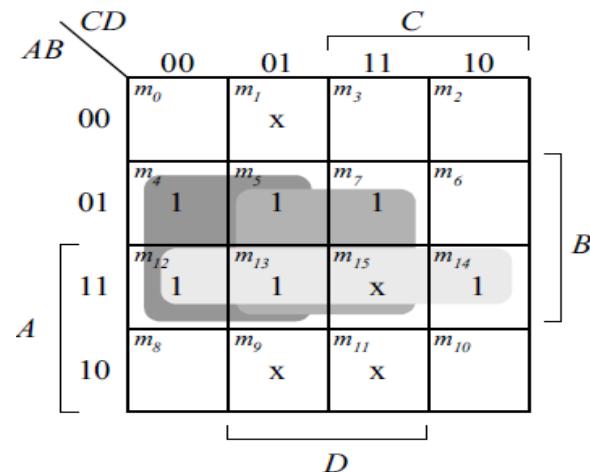
$$F = B'D' + A'B' + ABCD$$

Example 4

Simplify the Boolean function F , together with don't-care conditions d , and then express the simplified function in sum-of-minterms forms.

$$F(A,B,C,D) = \sum(4, 5, 7, 12, 13, 14)$$

$$d(A,B,C,D) = \sum(1, 9, 11, 15)$$



$$F = BC' + BD + AB$$

HDL Description of The Simplified Function

$F = YZ + W'X'$ (*From Example 1*)

```
// Verilog model: Circuit with Boolean expressions
module Example_1 (F, W, X, Y, Z);
Output F;
Input W, X, Y, Z;
assign F = (Y && Z) || (!W) && (!X);
endmodule
```

HDL Description of The Simplified Function

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

```
// Verilog model:with Boolean expressions
```

```
module Circuit (F,A,B,C,D);
```

```
Output F;
```

```
Input A, B, C, D;
```

```
assign F = ((! A) || (! B)) && ((! C) || (! D)) && ((! B) || D);
```

```
endmodule
```

HDL Description of The Simplified Function (*From Example 3*)

```
module Example_3 (F, A, B, C, D);
Output F;
Input A, B, C, D;
assign F = ((!B) && (!D)) || ((!A) && (!B)) || (A && B && C && D);
endmodule
```

NAND and NOR Implementation



Lecture-14
By
Bhagyalaxmi Behera
Asst. Professor (Dept. of ECE)

Contents

Introduction

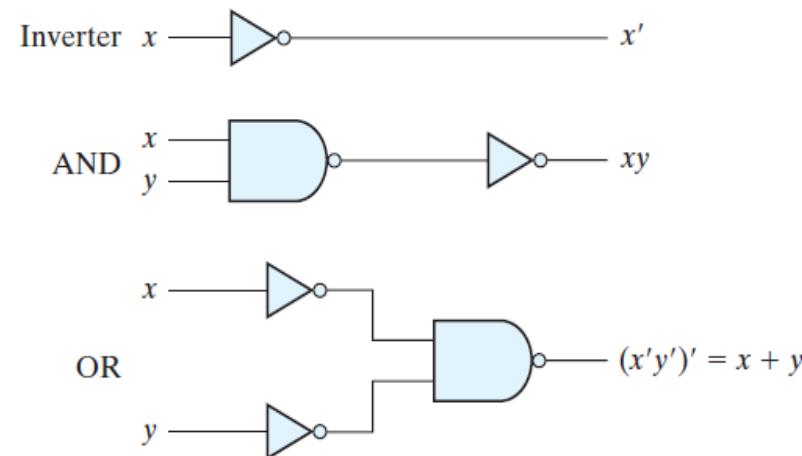
NAND and NOR implementation

Introduction

- Digital circuits are frequently constructed with **NAND** or **NOR** gates rather than with AND and OR gates.
- NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families.
- Because of the prominence of NAND and NOR gates in the design of digital circuits, rules and procedures have been developed for the conversion from Boolean functions given in terms of AND, OR, and NOT into equivalent NAND and NOR logic diagrams.

NAND Circuits

- The NAND gate is said to be a *universal gate because any logic circuit can be implemented* with it. To show that any Boolean function can be implemented with NAND gates, we need only show that the logical operations of AND, OR, and complement can be obtained with NAND gates alone. (As shown below)

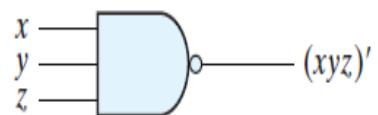


Logic operations with NAND gates

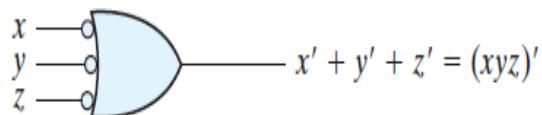
Three-input NAND gate

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

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



(a) AND-invert



(b) Invert-OR

Two graphic symbols for a three-input NAND gate

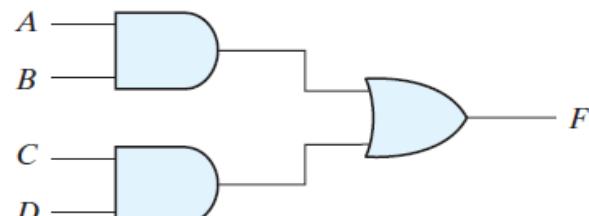
Two-Level Implementation

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

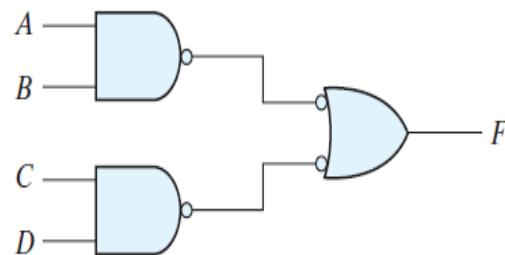
Example 1

$$F = AB + CD$$

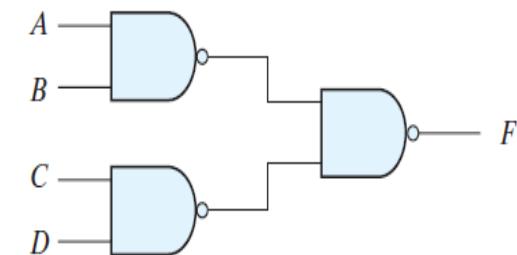
$$F = ((AB + CD)')' = ((AB)'(CD)')'$$



(a)



(b)



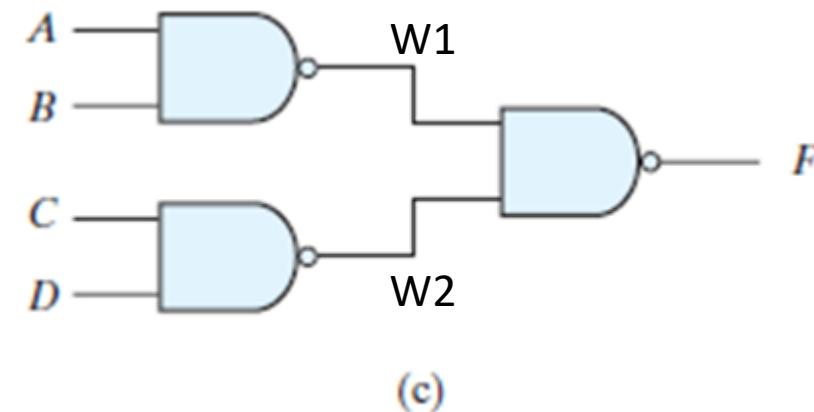
(c)

Three ways to implement $F = AB + CD$

HDL for the above circuit:

```
// Verilog model of two level  
implementation circuit using NAND gates.
```

```
module two-level-gf (A, B, C, D, F);  
output F;  
input A, B, C, D;  
wire w1, w2;  
nand G1 (w1, A, B);  
nand G2 (w2, C, D);  
nand G3 (F, w1, w2);  
endmodule
```



Example 2

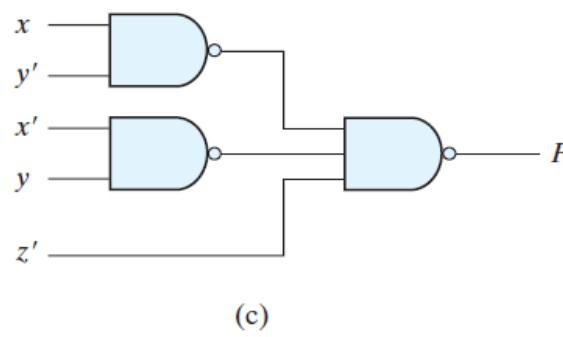
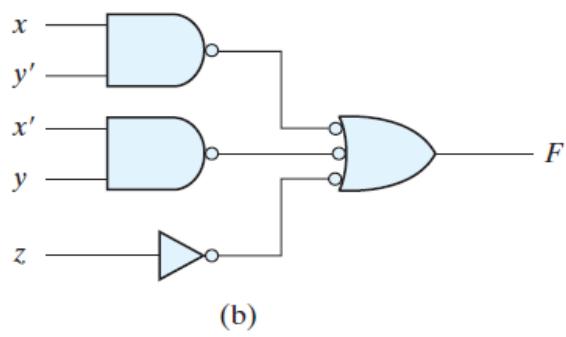
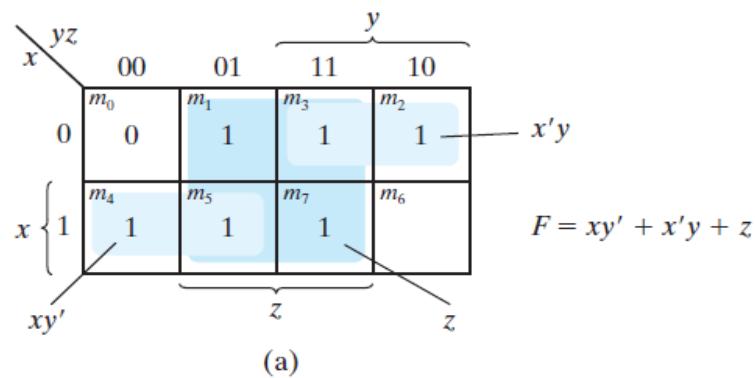
Implement the following Boolean function with NAND gates: $F(x, y, z) = \sum (1, 2, 3, 4, 5, 7)$

The procedure described in the previous example indicates that a Boolean function can be implemented with two levels of NAND gates. The procedure for obtaining the logic diagram from a Boolean function is as follows:

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

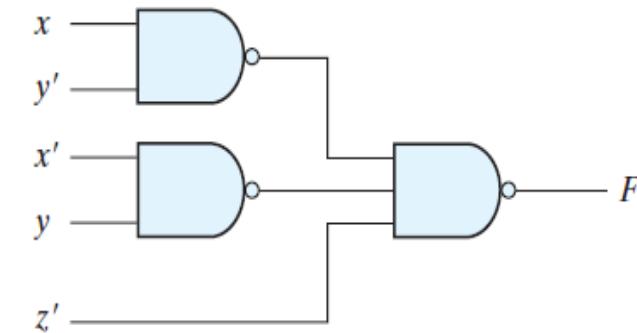
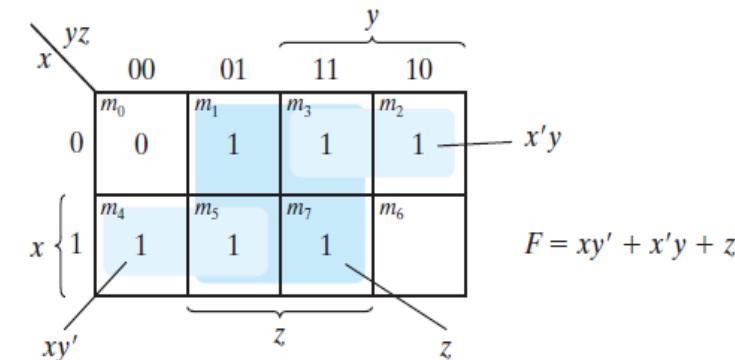
Answer

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



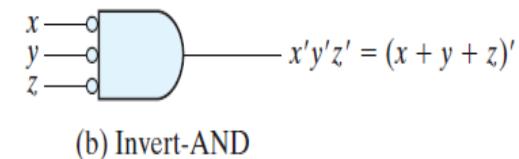
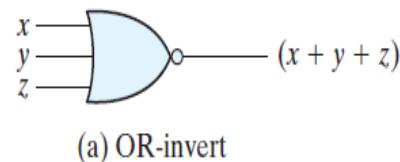
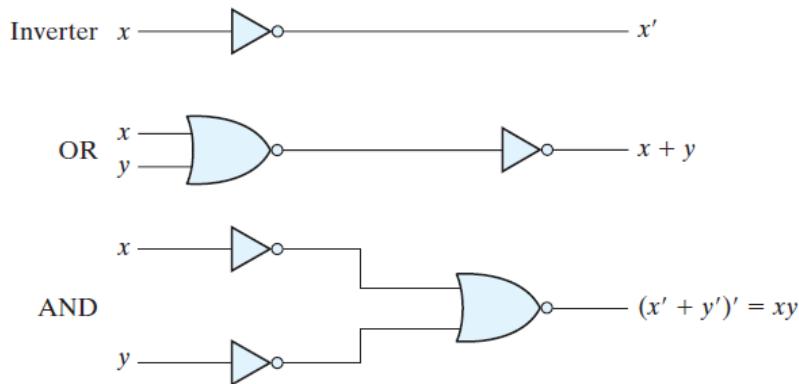
HDL for the above Boolean function

```
// Verilog model: Circuit with Boolean  
expressions  
module Circuit_Boolean_df (F, x, y, z);  
output F;  
input x, y, z;  
assign F = (x && (!y)) || ((!x) && y) || z ;  
endmodule
```



NOR Implementation

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



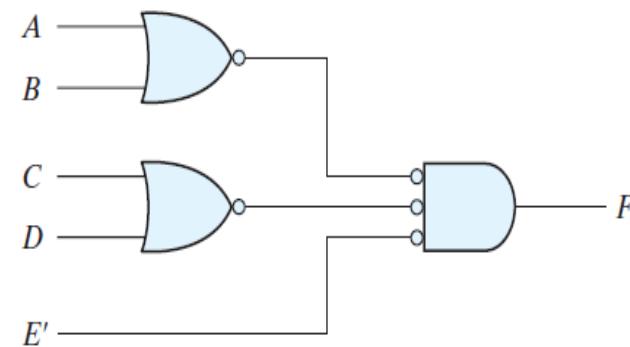
Two graphic symbols for the NOR gate

Logic operations with NOR gates

Example1

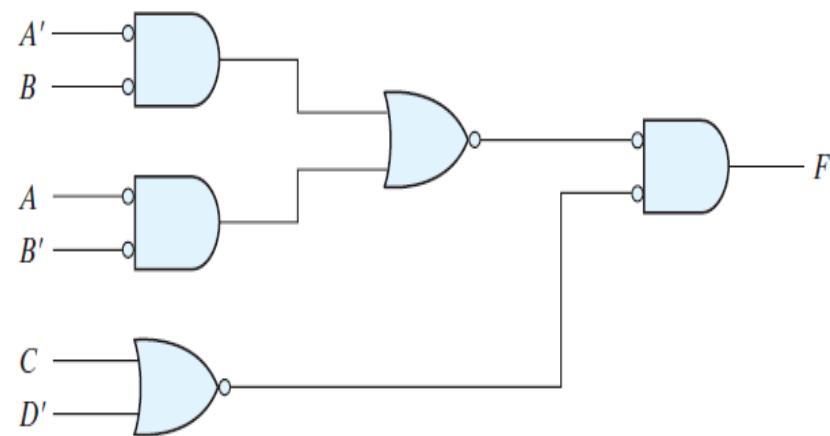
$$F = (A + B)(C + D)E$$

The OR-AND pattern can easily be detected by the removal of the bubbles along the same line. Variable E is complemented to compensate for the third bubble at the input of the second-level gate.



Example 2

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

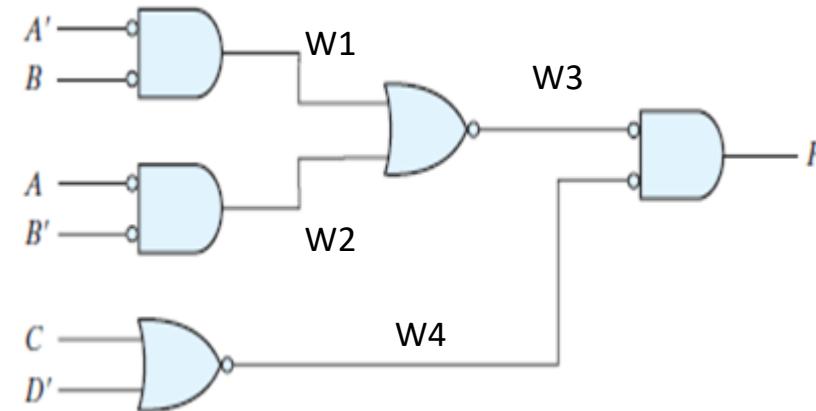


HDL for the above Boolean function using NOR gates

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

```
// Verilog model of Boolean function using  
NOR gates.
```

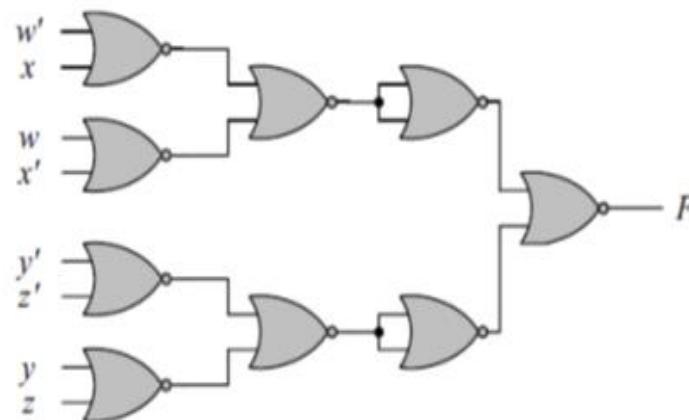
```
module boolean-usingnor-gf(F, A, B, C,  
D);  
output F;  
input A, B, C, D;  
wire A', B', D', w1, w2, w3, w4;  
nor (A', A, A),  
      (B', B, B),  
      (D', D, D);  
nor G1 (w1, A', B),  
      G2 (w2, A, B'),  
      G3 (w4, C, D'),  
      G4 (w3, w1, w2),  
      G5 (F, w3, w4);  
endmodule
```



Example 3

Implement the following Boolean function with NOR gates: $F(w,x,y,z) = \sum(1, 2, 13, 14)$

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



$$F = \Sigma(1, 2, 13, 14)$$

$$F' = w'x + wx' + y'z' + yz = [(w+x')(w'+x)(y+z)(y'+z')]'$$

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

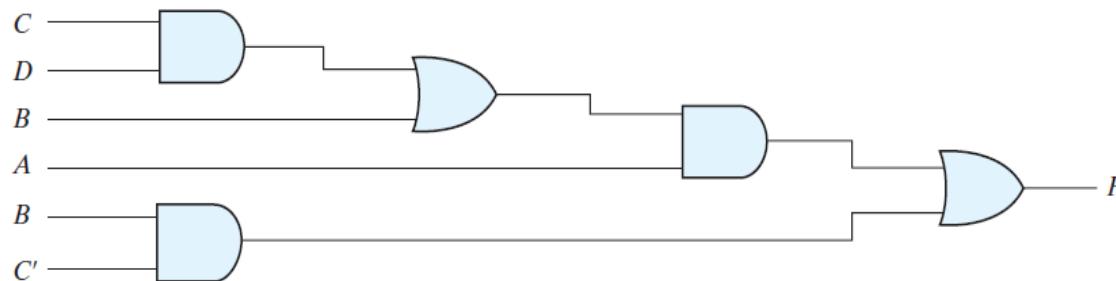
Multilevel NAND Circuits (SOP)



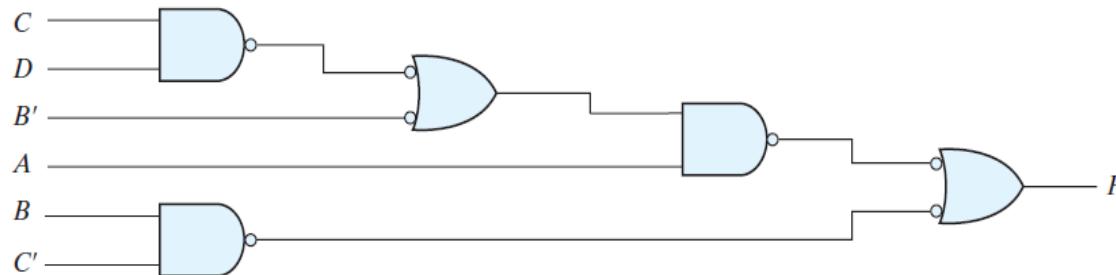
Lecture-15
By
Bhagyalaxmi Behera
Asst. Professor (Dept. of ECE)

Multilevel NAND Circuits (SOP)

$$F = A(CD + B) + BC'$$



(a) AND-OR gates



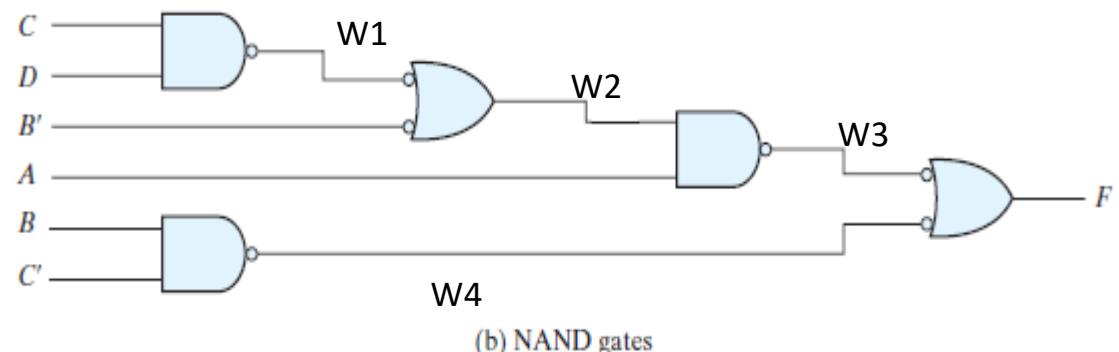
(b) NAND gates

The general procedure for converting a multilevel AND–OR diagram into an all-NAND diagram using mixed notation is as follows:

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

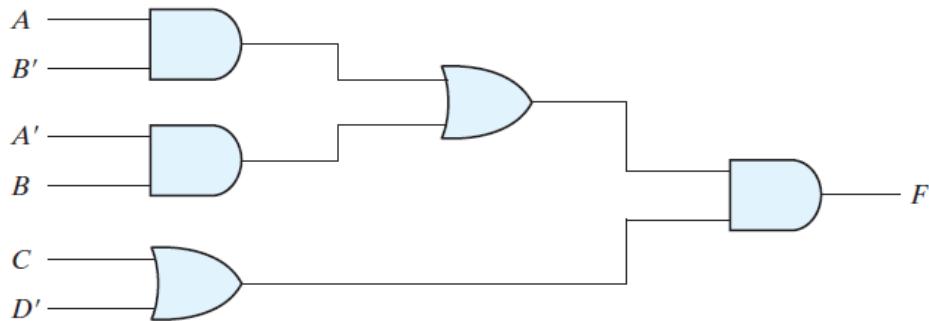
HDL for the above Multilevel NAND Circuit (SOP)

```
// Verilog model of Multilevel level  
implementation circuit using NAND gates.  
module multi-level-gf (F, A, B, C, D);  
output F;  
input A, B, C, D;  
wire B', C', w1, w2, w3, w4;  
nand (B', B, B),  
      (C', C, C);  
nand G1 (w1, C, D),  
      G2 (w2, w1, B'),  
      G3 (w3, A, w2),  
      G4 (w4, B, C'),  
      G5 (F, w3, w4);  
endmodule
```

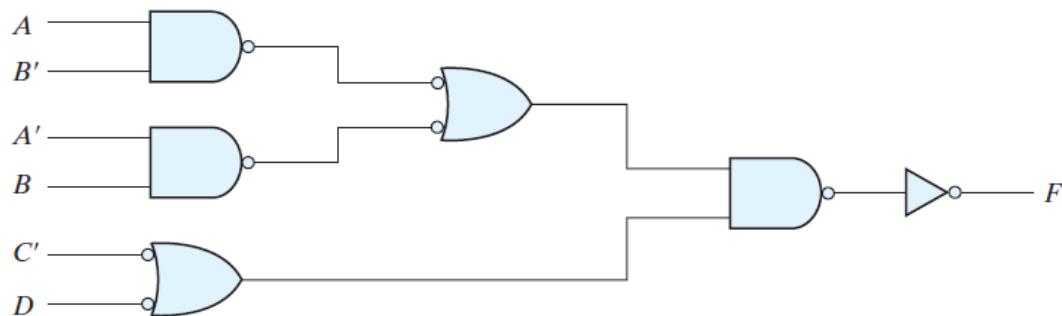


Multilevel NAND Circuits (POS)

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



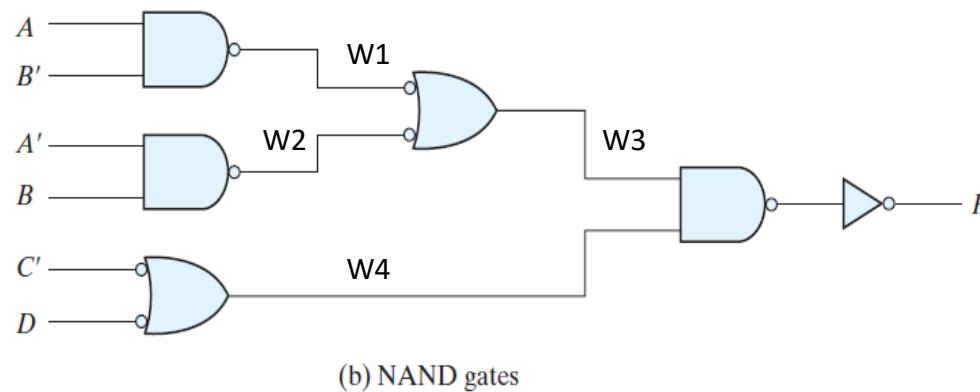
(a) AND-OR gates



(b) NAND gates

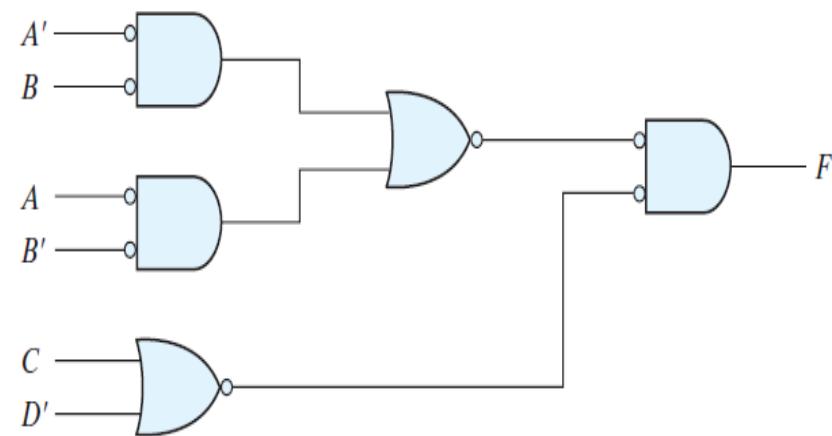
HDL for the above Multilevel NAND Circuit (POS)

```
module Fig_3_23b_gates (F, A, A_bar, B, B_bar, C_bar, D);
output F;
input A, A_bar, B, B_bar, C_bar, D;
wire w1, w2, w3, w4;
nand (w1, A, B_bar);
nand (w2, A_bar, B);
not (w1_bar, w1);
not (w2_bar, w2);
or (w3, w1_bar, w2_bar);
or (w4, C, D_bar);
not (w5, C_bar);
not (w6, D);
nand (F_bar, w3, w4);
not (F, F_bar);
endmodule
```



Multilevel NOR Implementation Example

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

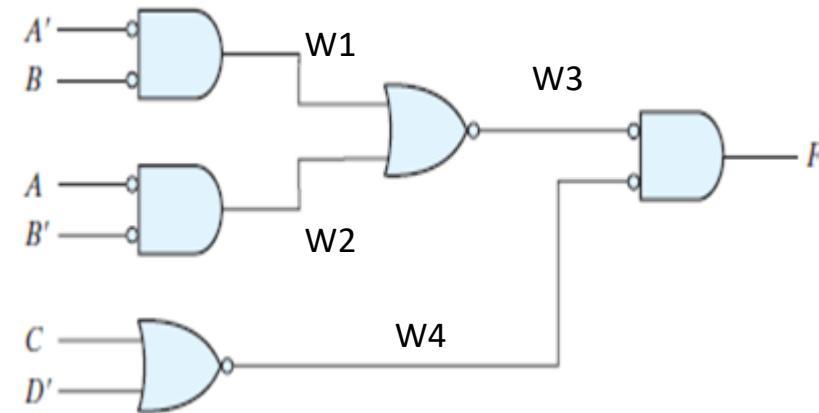


HDL for the above Boolean function using NOR gates

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

// Verilog model of Boolean function using
NOR gates.

```
module boolean-usingnor-gf(F, A, B, C,  
D);  
output F;  
input A, B, C, D;  
wire A', B', D', w1, w2, w3, w4;  
nor (A', A, A),  
      (B', B, B),  
      (D', D, D);  
nor G1 (w1, A', B),  
      G2 (w2, A, B'),  
      G3 (w4, C, D'),  
      G4 (w3, w1, w2),  
      G5 (F, w3, w4);  
endmodule
```



Example

Draw a multilevel NOR circuit for the following expression:

$$(AB' + CD')E + BC(A+B)$$

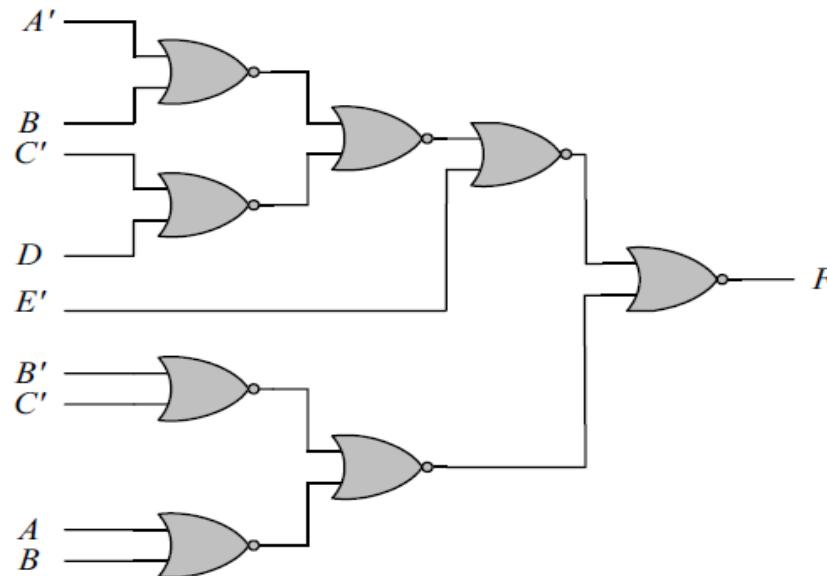
Multi-level NOR:

$$F = (AB' + CD'E) + BC(A+B)$$

$$F' = [(AB' + CD'E) + BC(A+B)]'$$

$$F' = [[(AB' + CD'E)' + E]'+ [(BC)' + (A+B)']']'$$

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



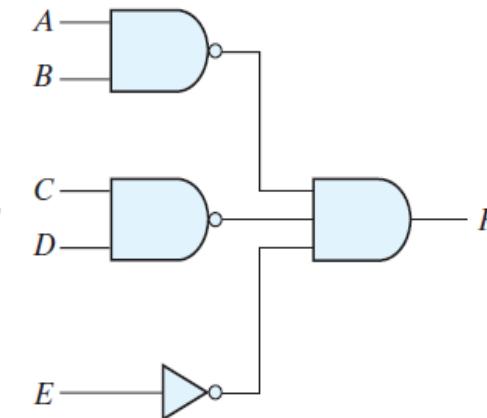
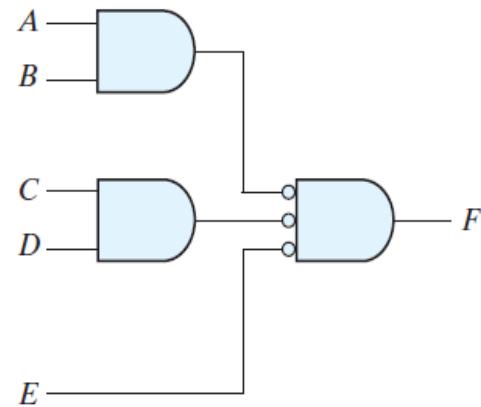
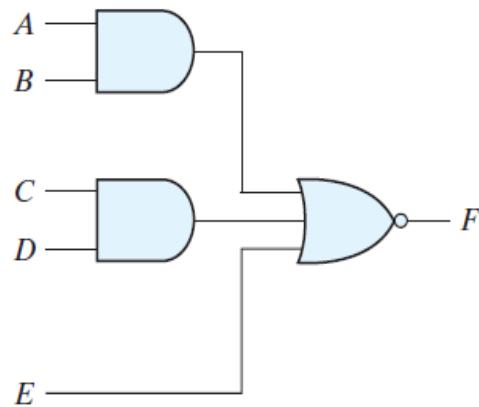
Nondegenerate Forms

The eight *nondegenerate forms* produce an implementation in sum-of-products form or product-of-sums form. The eight *nondegenerate forms* are as follows:

- AND–OR
- OR–AND
- NAND–NAND
- NOR–NOR
- NOR–OR
- NAND–AND
- OR–NAND
- AND–NOR

AND-OR-INVERT Implementation

$$F = (AB + CD + E)'$$



(a) AND-NOR

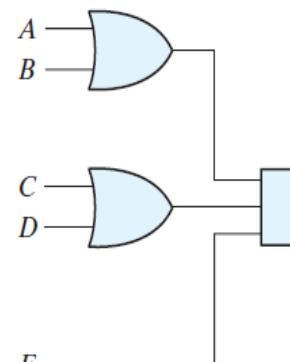
(b) AND-NOR

(c) NAND-AND

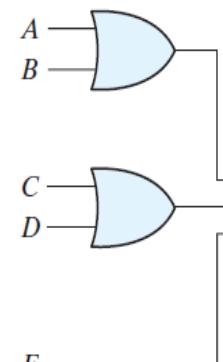
OR-AND-INVERT Implementation

The OR-NAND and NOR-OR forms perform the OR-AND-INVERT function.

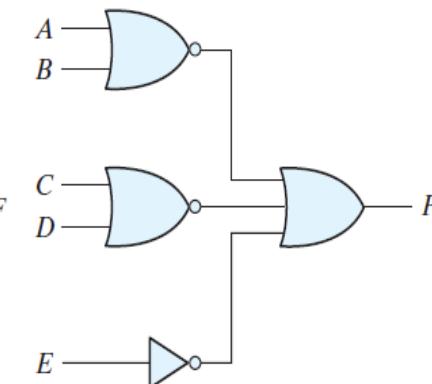
$$F = [(A + B)(C + D)E]'$$



(a) OR-NAND



(b) OR-NAND



(c) NOR-OR

HDL for the above AND-OR-INVERT & OR-AND-INVERT Circuit

$$F = (AB + CD + E)'$$

```
// Verilog model: Circuit with Boolean  
expressions  
module Circuit_Boolean_df (F, A, B, C,  
D, E);  
    output F;  
    input A, B, C, D, E;  
    assign F = !((A && B) || (C&& D) || E);  
endmodule
```

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

```
// Verilog model: Circuit with Boolean  
expressions  
module Circuit_Boolean_df (F, A, B, C,  
D, E);  
    output F;  
    input A, B, C, D, E;  
    assign F = (A || B) && (C || D) && (!E) ;  
endmodule
```

Exclusive OR and Equivalence Functions

- The exclusive-OR (XOR), denoted by the symbol \oplus , is a logical operation that performs the following Boolean operation:

$$x \oplus y = xy' + x'y$$

- It is particularly useful in arithmetic operations and error detection and correction circuits.
- The **exclusive-OR** is equal to 1 if only x is equal to 1 or if only y is equal to 1 (i.e., x and y differ in value), but not when both are equal to 1 or when both are equal to 0. The **exclusive NOR**, also known as **equivalence**, performs the following Boolean operation:

$$(x \oplus y)' = xy + x'y'$$

- The exclusive-NOR is equal to 1 if both x and y are equal to 1 or if both are equal to 0. The exclusive-NOR can be shown to be the complement of the exclusive-OR by means of a truth table or by algebraic manipulation:

$$(x \oplus y)' = (xy' + x'y)' = (x' + y)(x + y') = xy + x'y'$$

➤ The following identities apply to the exclusive-OR operation:

$$x \oplus 0 = x$$

$$x \oplus 1 = x'$$

$$x \oplus x = 0$$

$$x \oplus x' = 1$$

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

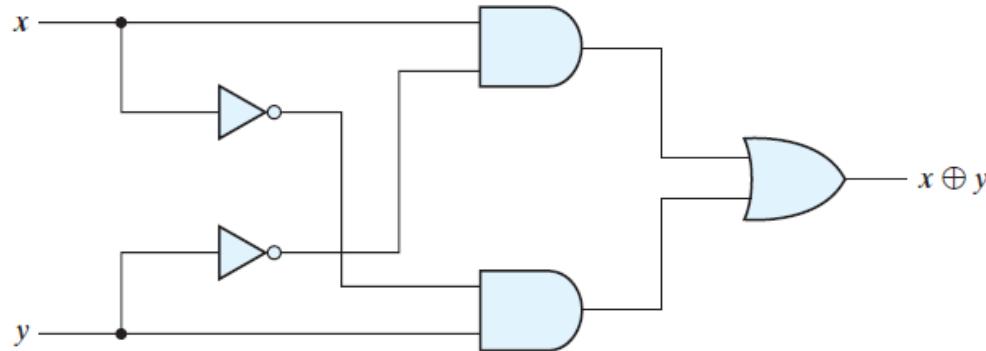
➤ Also, it can be shown that the exclusive-OR operation is both **commutative** and **associative**; that is,

$$A \oplus B = B \oplus A$$

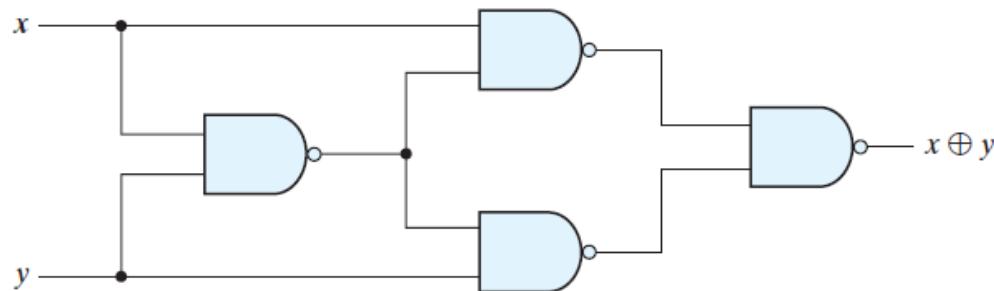
$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$$

➤ This means that the two inputs to an exclusive-OR gate can be interchanged without affecting the operation.

Construction of a two-input exclusive-OR function:



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

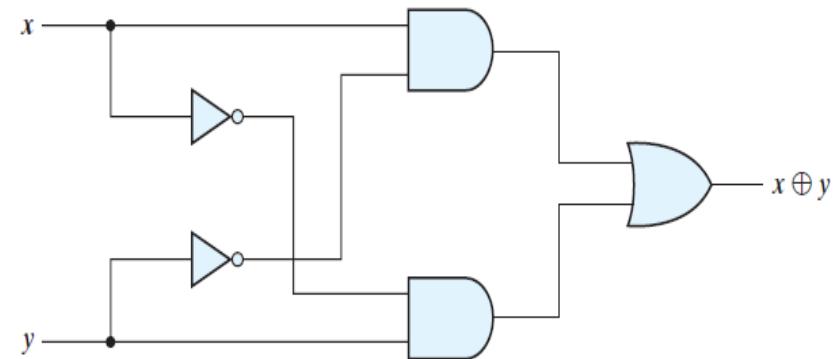


(b) Exclusive-OR with NAND gates

HDL For XOR Gate

```
// Verilog model for XOR gate.  
module exclusiveor-df (F, x, y);  
  output F;  
  input x, y;  
  assign F = (x && (!y)) || ((!x) && y);  
endmodule
```

$$x \oplus y = xy' + x'y$$



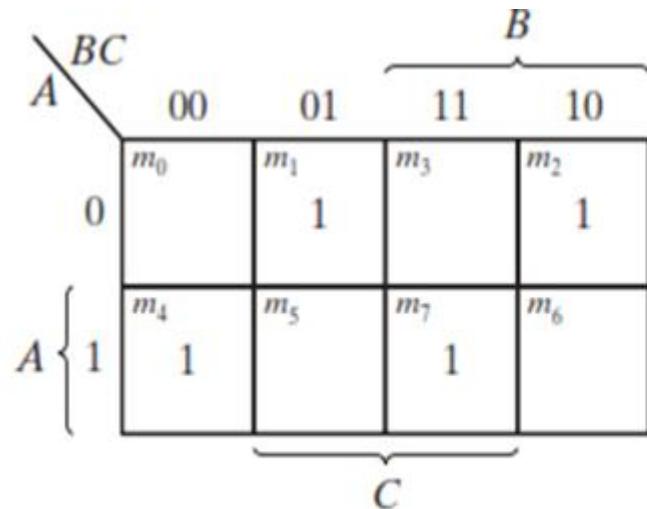
Odd Function

The exclusive-OR operation with three or more variables can be converted into an ordinary Boolean function by replacing the \oplus symbol with its equivalent Boolean expression. In particular, the three-variable case can be converted to a Boolean expression as follows:

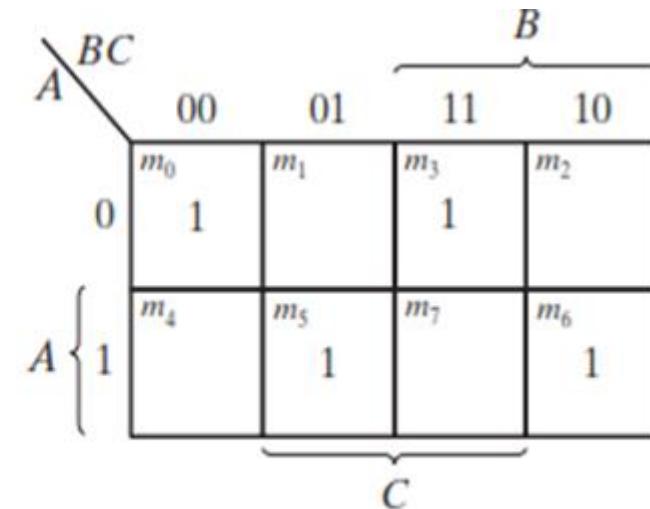
$$\begin{aligned} A \oplus B \oplus C &= (AB' + A'B)C' + (AB + A'B')C \\ &= AB'C' + A'BC' + ABC + A'B'C \\ &= \Sigma(1, 2, 4, 7) \end{aligned}$$

The multiple-variable exclusive-OR operation is defined as an **odd function** because in the case of three or more variables the requirement is that an odd number of variables be equal to 1.

- In general, an n -variable exclusive-OR function is an odd function defined as the logical sum of the $2^n/2$ minterms whose binary numerical values have an odd number of 1's.
- The definition of an odd function can be clarified by plotting it in a map.



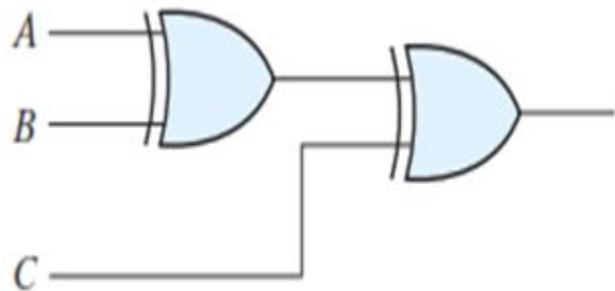
(a) Odd function $F = A \oplus B \oplus C$



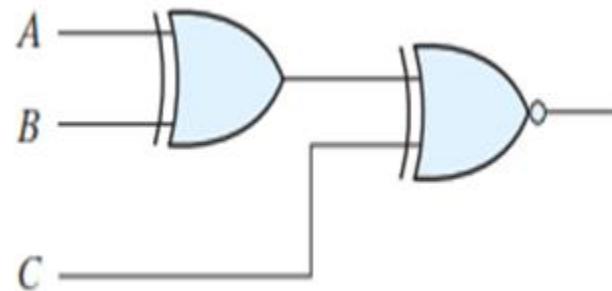
(b) Even function $F = (A \oplus B \oplus C)'$

Map for a three-variable exclusive-OR function

Logic diagram of odd and even functions



(a) 3-input odd function



(b) 3-input even function

The three-input odd function is implemented by means of two-input exclusive-OR gates, as shown in Fig (a). The complement of an odd function is obtained by replacing the output gate with an exclusive-NOR gate, as shown in Fig.(b).

- Consider now the **four-variable exclusive-OR** operation. By algebraic manipulation, we can obtain the sum of minterms for this function:

$$\begin{aligned}
 A \oplus B \oplus C \oplus D &= (AB' + A'B) \oplus (CD' + C'D) \\
 &= (AB' + A'B)(CD + C'D') + (AB + A'B')(CD' + C'D) \\
 &= \Sigma(1, 2, 4, 7, 8, 11, 13, 14)
 \end{aligned}$$

There are 16 minterms for a four-variable Boolean function. Half of the minterms have binary numerical values with an odd number of 1's; the other half of the minterms have binary numerical values with an even number of 1's.

- In plotting the function in the map, the binary numerical value for a minterm is determined from the row and column numbers of the square that represents the minterm.
- The map of Fig.(a) is a plot of the 4-variable exclusive-OR function. This is an odd function because the binary values of all the minterms have an odd number of 1's. The complement of an odd function is an even function. As shown in Fig.(b), the 4-variable even function is equal to 1 when an even number of its variables is equal to 1.

		CD		C			
		00	01	11	10		
		00	m_0	m_1	m_3	m_2	1
A	B	01	m_4	m_5	m_7	1	m_6
		11	m_{12}	m_{13}	1	m_{15}	m_{14}
		10	m_8	m_9	1	m_{11}	m_{10}
			D				

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

		CD		C			
		00	01	11	10		
		00	m_0	m_1	m_3	m_2	1
A	B	01	m_4	m_5	m_7	m_6	1
		11	m_{12}	m_{13}	m_{15}	m_{14}	1
		10	m_8	m_9	m_{11}	m_{10}	1
			D				

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

Combinational Logic Circuits



Lecture-16

By
Bhagyalaxmi Behera
Asst. Professor (Dept. of ECE)

Combinational Logic

Introduction

- Logic circuits for digital systems may be **combinational** or **sequential**.
- A **combinational** circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs.
- In contrast, **sequential** circuits employ storage elements in addition to logic gates. Their outputs are a function of the inputs and the state of the storage elements. The circuit behaviour must be specified by a time sequence of inputs and internal states.

Combinational circuits

- A combinational circuit consists of an interconnection of logic gates. Combinational logic gates react to the values of the signals at their inputs and produce the value of the output signal, transforming binary information from the given input data to a required output data.
- Combinational circuit is a circuit in which we combine the different gates in the circuit, for example encoder, decoder, multiplexer and demultiplexer.

- Some of the characteristics of combinational circuits are following:
- The output of combinational circuit at any instant of time, depends only on the levels present at input terminals.
- The combinational circuit do not use any memory. The previous state of input does not have any effect on the present state of the circuit.
- A combinational circuit can have an n number of inputs and m number of outputs.

- The n input binary variables come from an external source; the m output variables are produced by the internal combinational logic circuit and go to an external destination.
- For n input variables, there are 2^n possible combinations of the binary inputs.
- Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables.
- A combinational circuit also can be described by m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

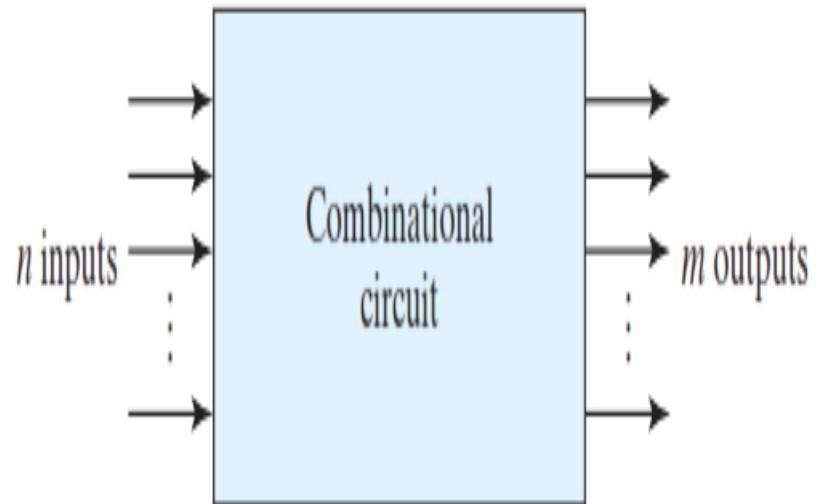
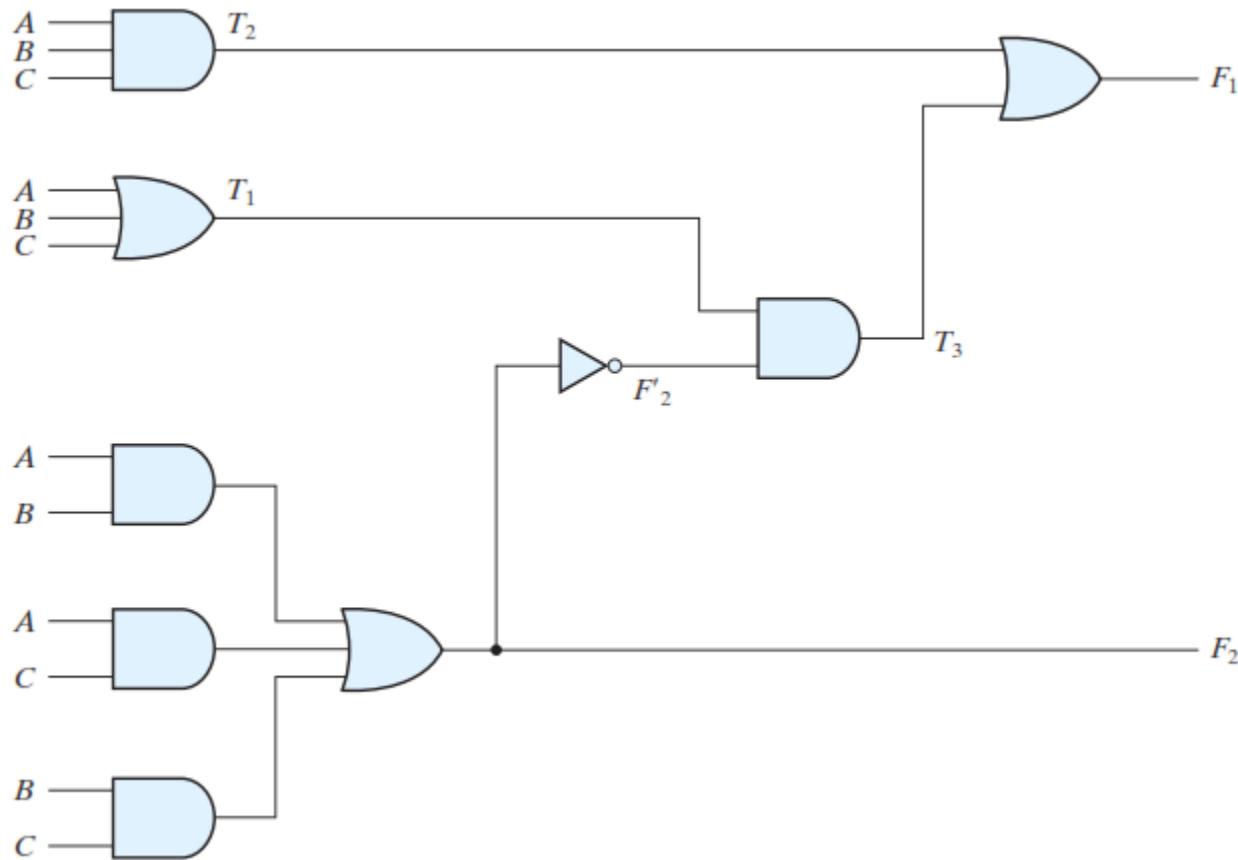


Fig 1 Block diagram of combinational circuit

Analysis Procedure

- The analysis of a combinational circuit requires that we determine the function that the circuit implements.
- To obtain the output Boolean functions from a logic diagram, we proceed as follows:
 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.



$$F_2 = AB + AC + BC$$

$$T_1 = A + B + C$$

$$T_2 = ABC$$

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:

$$\begin{aligned} F_1 &= T_3 + T_2 = F_2 T_1 + ABC = (AB + AC + BC)'(A + B + C) + ABC \\ &= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\ &= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\ &= A'BC' + A'B'C + AB'C' + ABC \end{aligned}$$

➤ To obtain the truth table directly from the logic diagram without going through the derivations of the Boolean functions, we proceed as follows:

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.

Truth Table

A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Design Procedure

- The design of combinational circuits starts from the specification of the design objective and culminates in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be obtained. The procedure involves the following steps:
- From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each.
- Derive the truth table that defines the required relationship between inputs and outputs.

- Obtain the simplified Boolean functions for each output as a function of the input variables.
- Draw the logic diagram and verify the correctness of the design (manually or by simulation).

Note:

- A truth table for a combinational circuit consists of input columns and output columns. The input columns are obtained from the 2^n binary numbers for the n *input* variables. The binary values for the outputs are determined from the stated specifications.
- The output functions specified in the truth table give the exact definition of the combinational circuit. The output binary functions listed in the truth table are simplified by any available method, such as algebraic manipulation, the map method, or a computer-based simplification program.
- A practical design must consider such constraints as the 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 (i.e., the number of gates to which the output of the circuit may be connected), and various other criteria that must be taken into consideration when designing integrated circuits.

- Since the importance of each constraint is dictated by the particular application, it is difficult to make a general statement about what constitutes an acceptable implementation.

- In most cases, the simplification begins by satisfying an elementary objective, such as producing the simplified Boolean functions in a standard form.

- Then the simplification proceeds with further steps to meet other performance criteria.

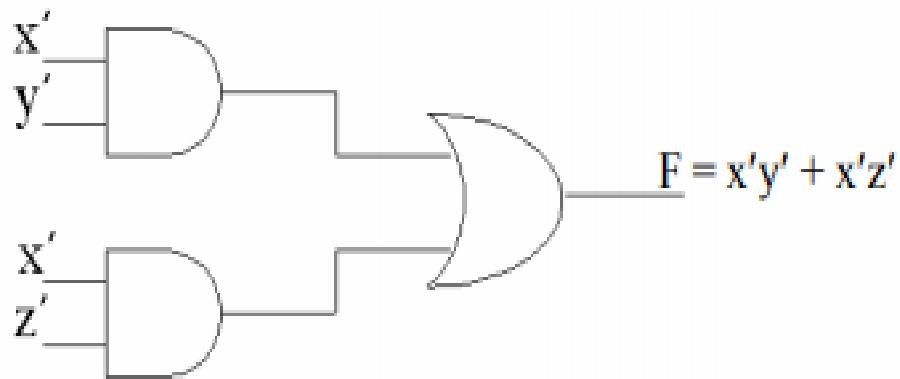
Some Examples:

1. Design a combinational circuit with three inputs and one output. The output is 1 when the binary value of the inputs is less than 3. The output is 0 otherwise.

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

A Karnaugh map for three variables x, y, and z. The columns are labeled x\yz (x-bar, y, z-bar) and the rows are labeled x (0, 1). The cells (00, 01, 11) are circled in red.

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



HDL for the above circuit:

```
// Verilog model for combinational circuit
module circuit1-df (F, x, y, z);
output F;
input x, y, z;
assign F = ((!x) && (!y)) || ((!x) && (!z));
endmodule
```

2) Design a combinational circuit with 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.

x	y	z	A	B	C
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

↔ y

$x \downarrow$

$x \setminus xyz$	00	01	11	10
0			1	
1		1	1	1

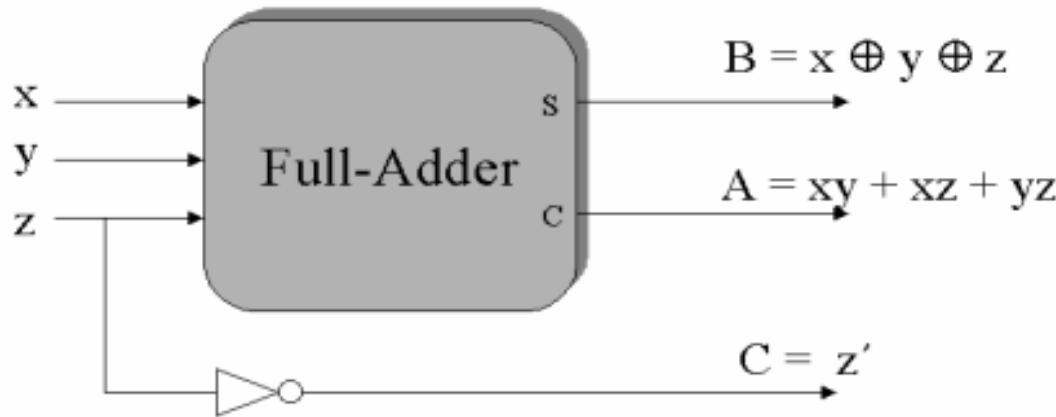
$$A = xy + xz + yz$$

↔ y

$x \downarrow$

$x \setminus xyz$	00	01	11	10
0		1		1
1	1		1	

$$B = x \oplus y \oplus z \quad C = z'$$



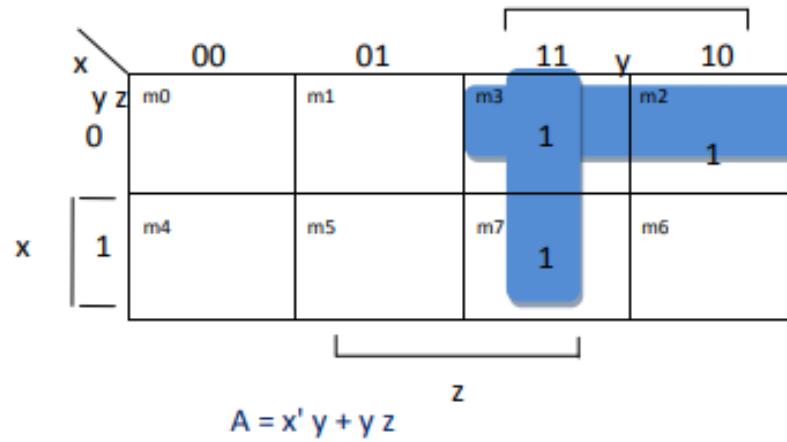
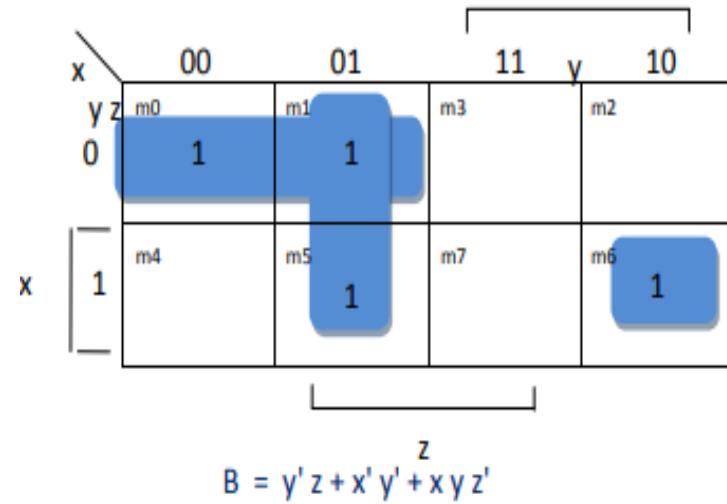
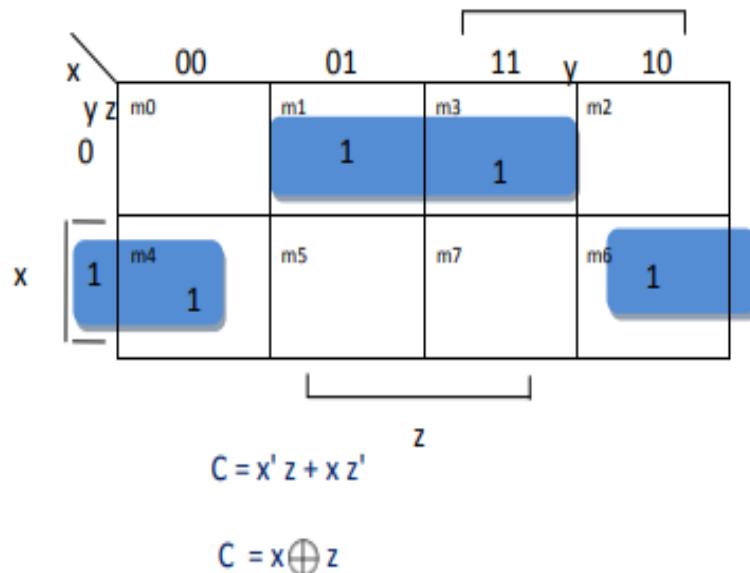
HDL for the above circuit:

```
// Verilog model for combinational circuit
module circuit2-df (A, B, C, x, y, z);
output A, B, C;
input x, y, z;
assign A = (x && y) || (x && z) || (y && z);
assign B = ((!x) && (!y) && (z)) || ((!x) && (y) && (!z)) || ((x) && (!y) && (!z)) ||
(x && y && z);
Assign C = (!z);
endmodule
```

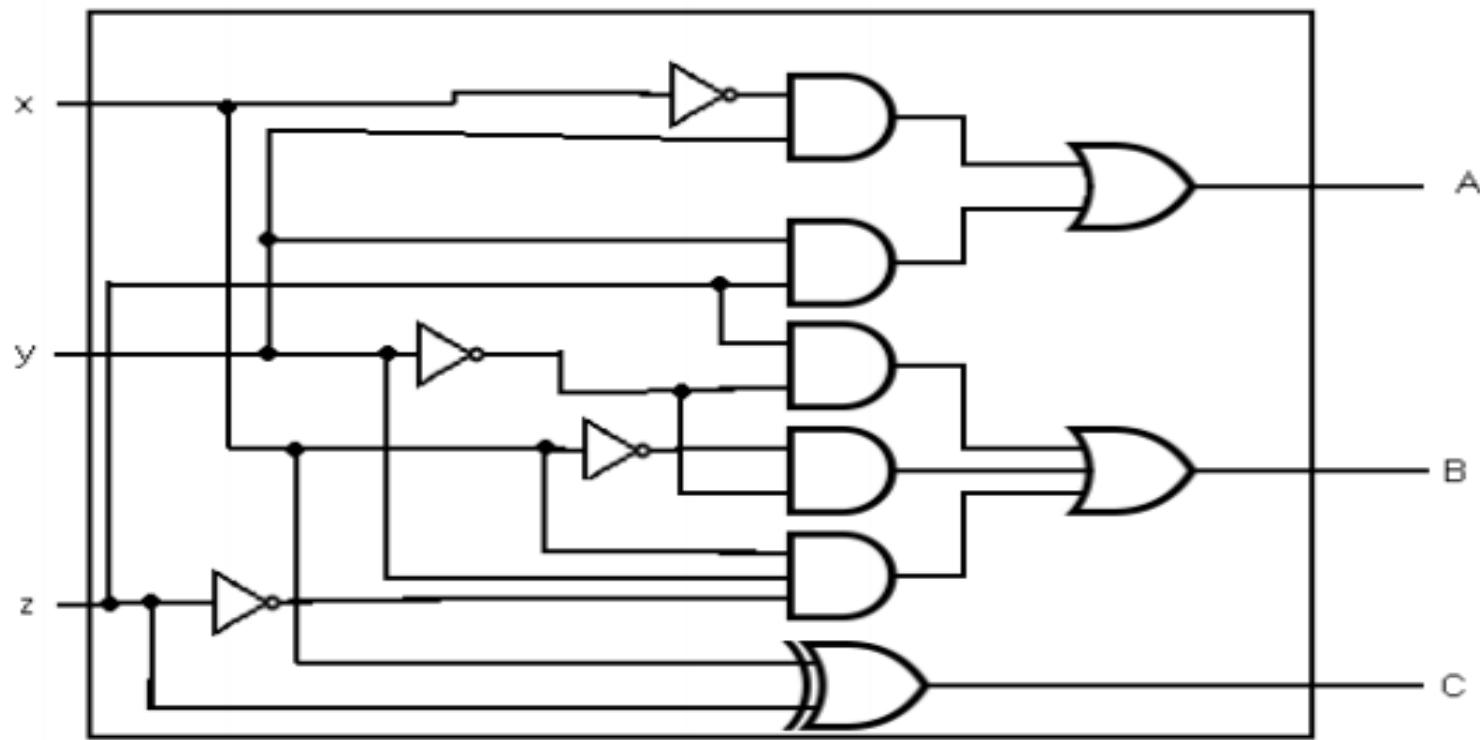
- Design a combinational circuit with three inputs, x, y and z and three outputs A,B, and C. When Binary inputs is 0,1,2 or 3, the binary outputs is two greater than the input. When the binary input is 4,5,6 or 7, the binary output is three less than the input.

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

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



The final circuit



HDL for the above circuit:

```
// Verilog model for combinational circuit
module circuit3-df (A, B, C, x, y, z);
output A, B, C;
input x, y, z;
assign A = (!x) && y || (y && z);
assign B = (!y) && (z) || (!x) && (!y) || (x) && (y) && (!z));
Assign C = (!x) && z || (x && (!z));
endmodule
```

THANK YOU

Combinational Logic Circuits



Lecture-17
By
Bhagyalaxmi Behera
Asst. Professor (Dept. of ECE)

Combinational Logic

Overview of previous lecture

- What is a Combinational Circuit.
- How is it different from Sequential Circuit.
- What is the analysis procedure for a Combinational Circuit.
- What is the design procedure for a Combinational Circuit.

Binary Adder

- The simple binary addition consists of four possible elementary operations:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$1 + 1 = 10$ (The higher significant bit of this result is called a *carry*).

When the augend and addend numbers contain more significant digits, the carry obtained from the addition of two bits is added to the next higher order pair of significant bits.

- A combinational circuit that performs the addition of two bits is called a *half adder* .
- One that performs the addition of three bits (two significant bits and a previous carry) is a *full adder* .
- A *binary adder-subtractor* is a combinational circuit that performs the arithmetic operations of addition and subtraction with binary numbers.

➤ Half Adder

From the verbal explanation of a half adder, we find that this circuit needs two binary inputs and two binary outputs. The input variables designate the augend and addend bits; the output variables produce the sum and carry. We assign symbols x and y to the two inputs and S (for sum) and C (for carry) to the outputs.

- The truth table for the half adder is listed below.
- The C output is 1 only when both inputs are 1.
- The S output represents the least significant bit of the sum.

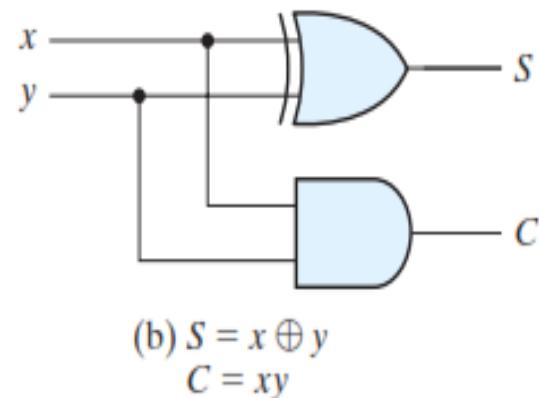
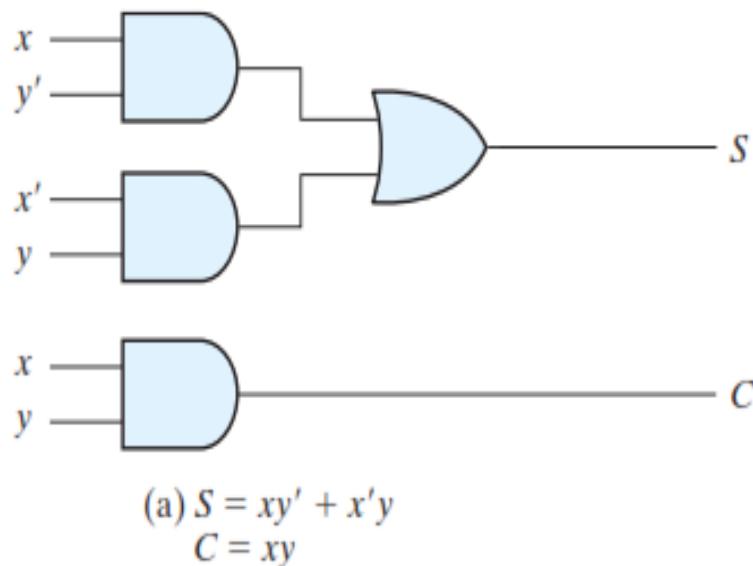
Half Adder

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

The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum-of-products expressions are

$$S = x'y + xy'$$

$$C = xy$$

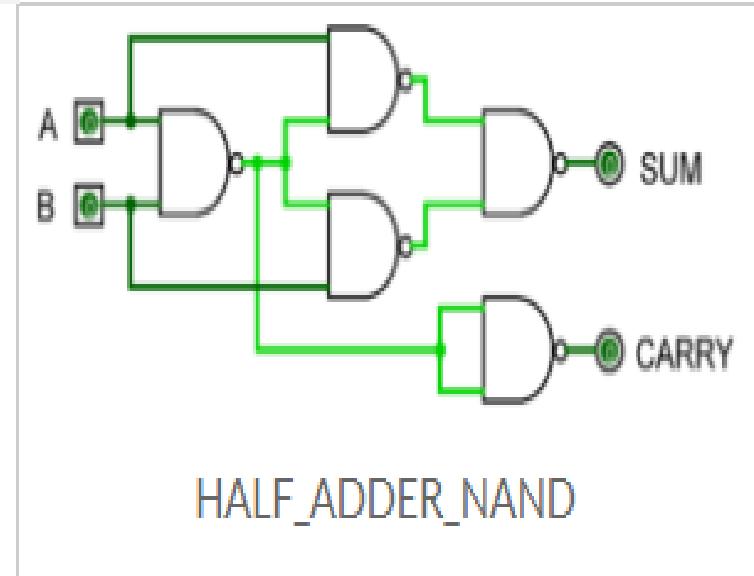
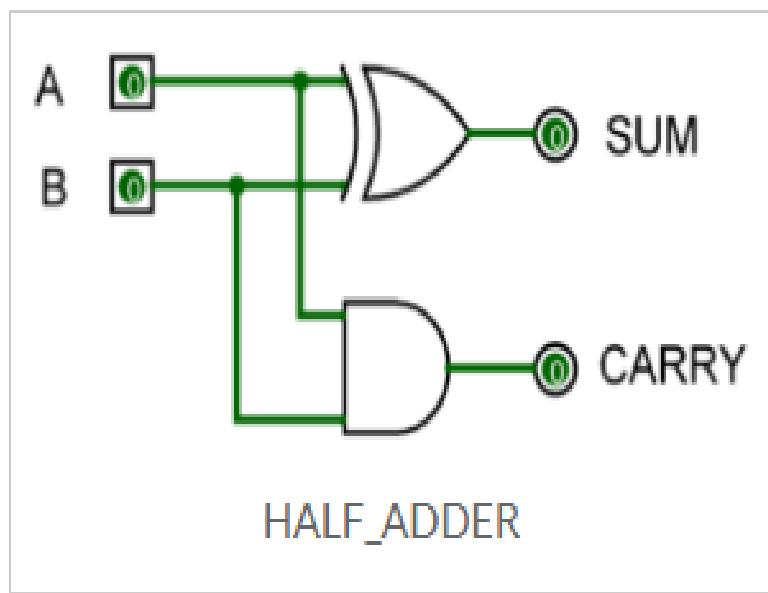


Implementation of half adder

Implement HALF ADDER using NAND GATES

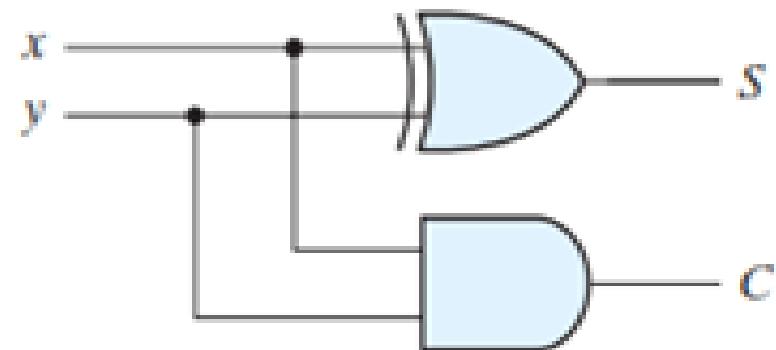
$$S = A \oplus B$$

$$C = AB$$



- HDL for Half Adder

```
module half_adder ( output S, C, input x, y);  
xor (S, x, y);  
and (C, x, y);  
endmodule
```



Full Adder

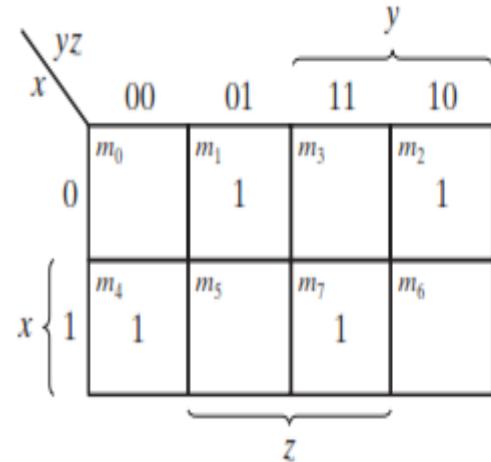
- A full adder is a combinational circuit that forms the arithmetic sum of three bits. It consists of three inputs and two outputs. Two of the input variables, denoted by x and y , represent the two significant bits to be added. The third input, z , represents the carry from the previous lower significant position.
- The two outputs are designated by the symbols S for sum and C for carry. The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry formed by adding the input carry and the bits of the words.

The truth table of the full adder is listed below.

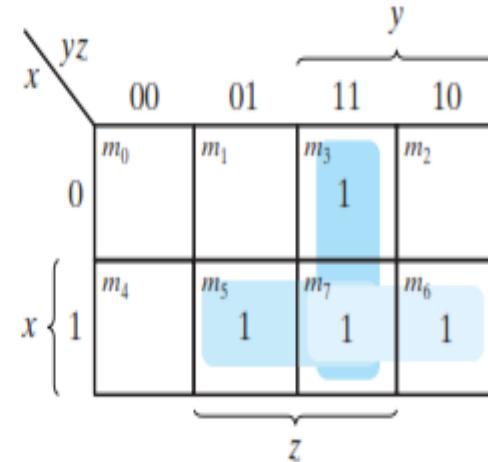
Full Adder

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

K-Maps for full adder

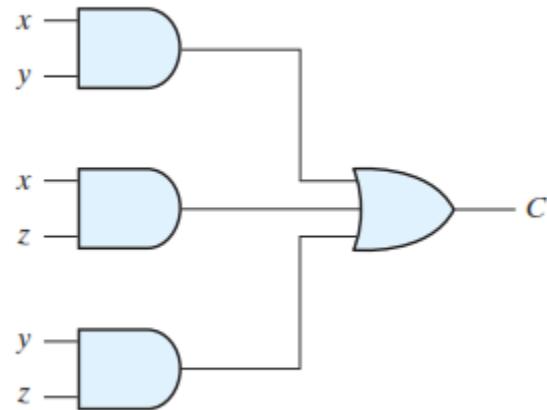
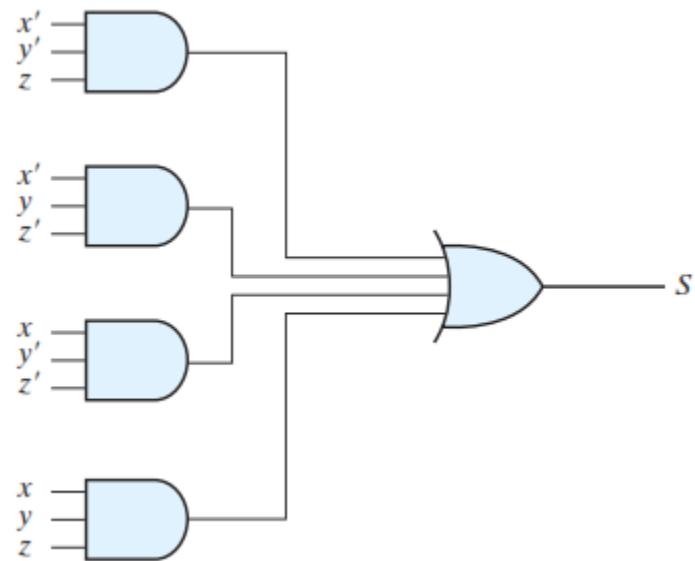


$$(a) S = x'y'z + x'y'z' + xy'z' + xyz$$



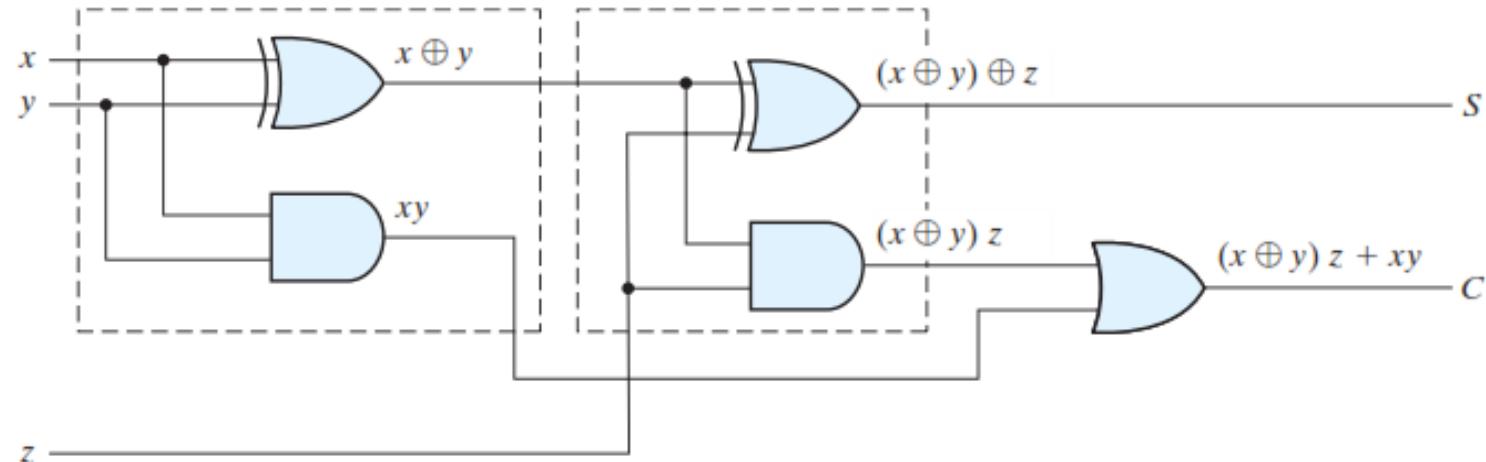
$$(b) C = xy + xz + yz$$

The logic diagram for the full adder implemented in sum-of-products form.



Implementation of full adder in sum-of-products form

It can also be implemented with two half adders and one OR gate,



The S output from the second half adder is the exclusive-OR of z and the output of the first half adder, giving \rightarrow

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

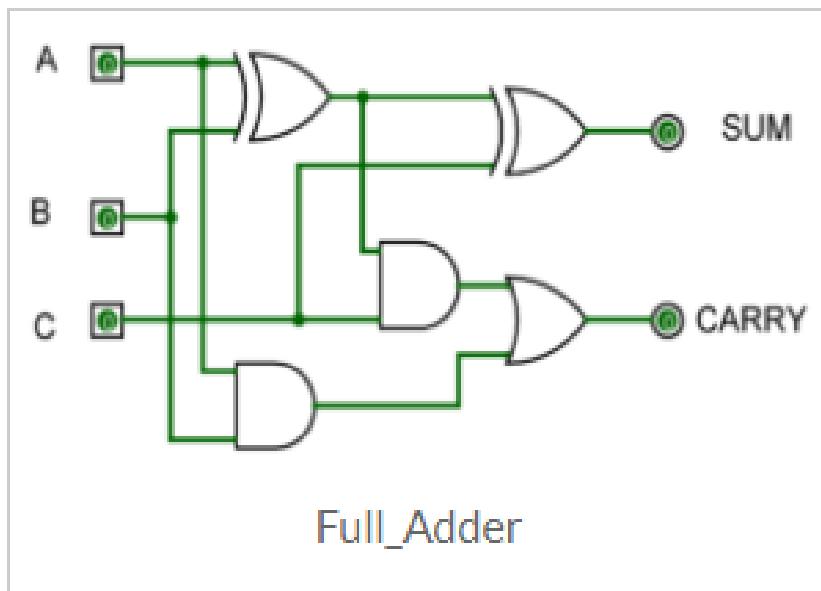
The carry output is

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

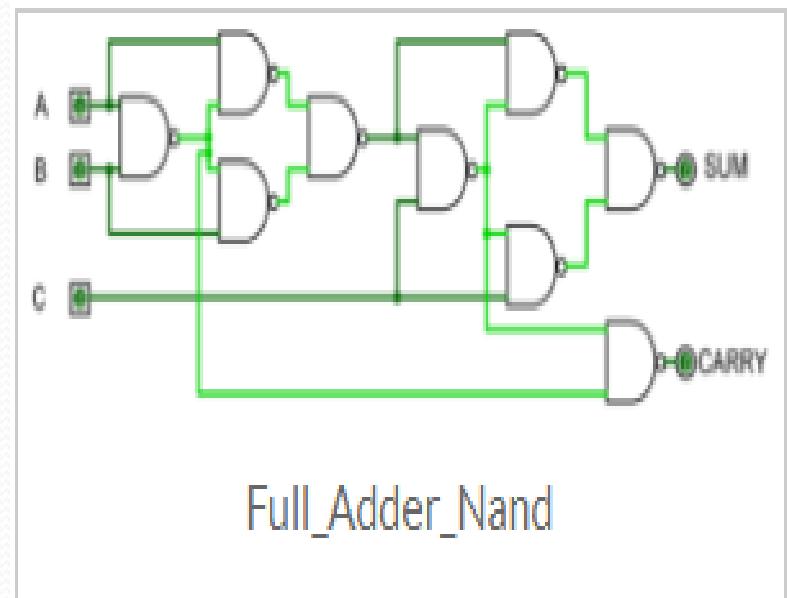
Realize Full ADDER using NAND gates only

$$\text{Sum} = A \oplus B \oplus C$$

$$\text{Carry} = A \cdot B + (A \oplus B) \cdot C$$



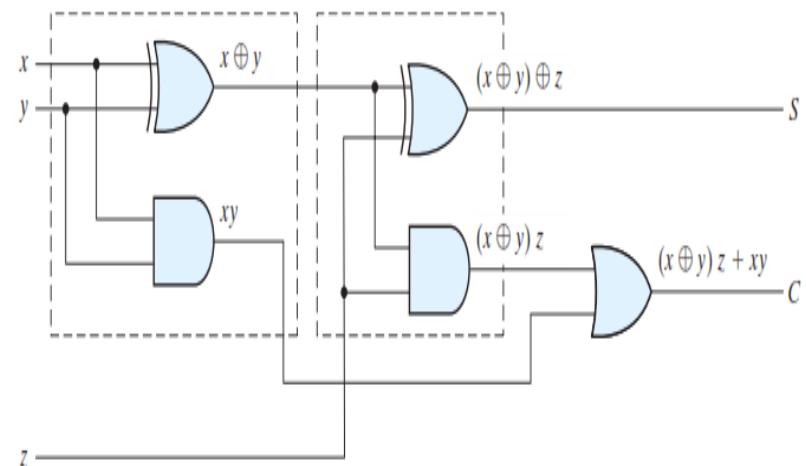
Full_Adder



Full_Adder_Nand

- HDL for Full Adder

```
// Description of full adder
// module full_adder (S, C, x, y, z);
// output S, C;
// input x, y, z;
module full_adder ( output S, C, input x, y, z);
wire S1, C1, C2;
// Instantiate half adders
half_adder HA1 (S1, C1, x, y);
half_adder HA2 (S, C2, S1, z);
or G1 (C, C2, C1);
endmodule
```



Binary adder

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

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

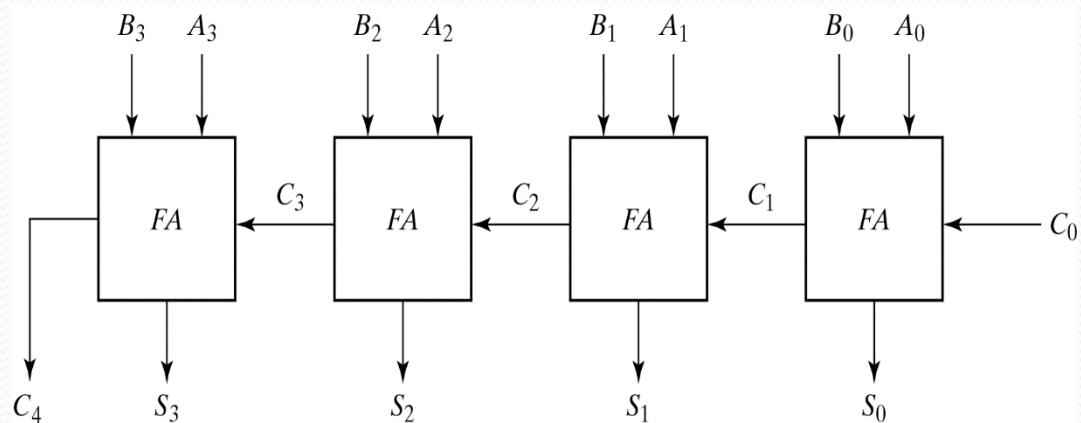


Fig. 4-9 4-Bit Adder

HDL for Ripple Carry Adder

```
module ripple_carry_4_bit_adder ( output [3: 0] Sum, output C4,  
input [3: 0] A, B,  
input Co);  
wire C1, C2, C3;  
// Intermediate carries  
// Instantiate chain of full adders  
full_adder FAo (Sum[0], C1, A[0], B[0], Co),  
FA1 (Sum[1], C2, A[1], B[1], C1),  
FA2 (Sum[2], C3, A[2], B[2], C2),  
FA3 (Sum[3], C4, A[3], B[3], C3);  
endmodule
```

Carry Propagation

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

Carry Propagation

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

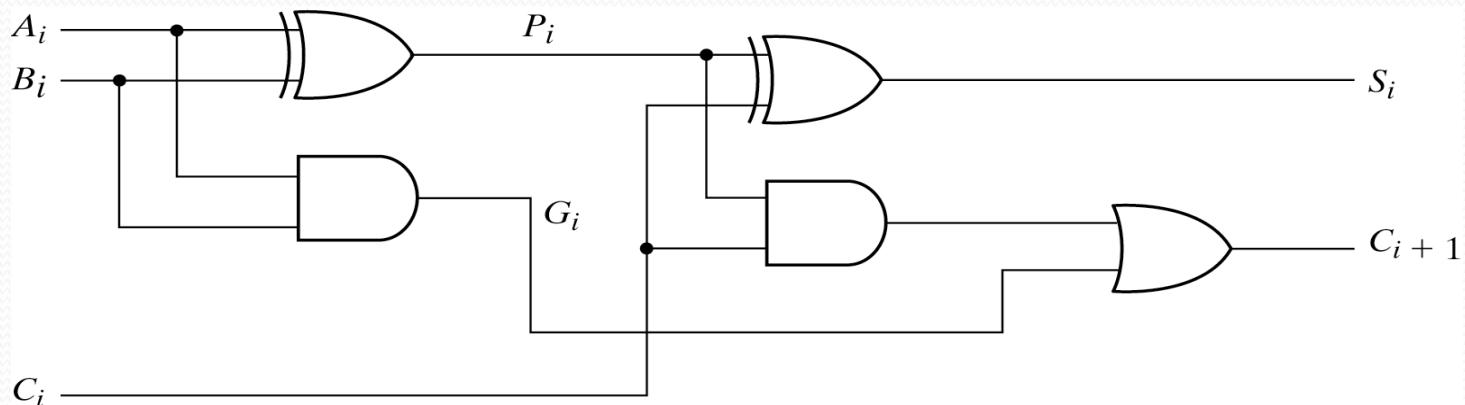


Fig. 4-10 Full Adder with P and G Shown

Boolean functions

$$P_i = A_i \oplus B_i \quad \text{steady state value}$$

$$G_i = A_i B_i \quad \text{steady state value}$$

Output sum and carry

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

G_i : carry generate P_i : carry propagate

C_o = input carry

$$C_1 = G_o + P_o C_o$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_o + P_1 P_o C_o$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_o + P_2 P_1 P_o C_o$$

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

Logic diagram of carry look-ahead generator

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

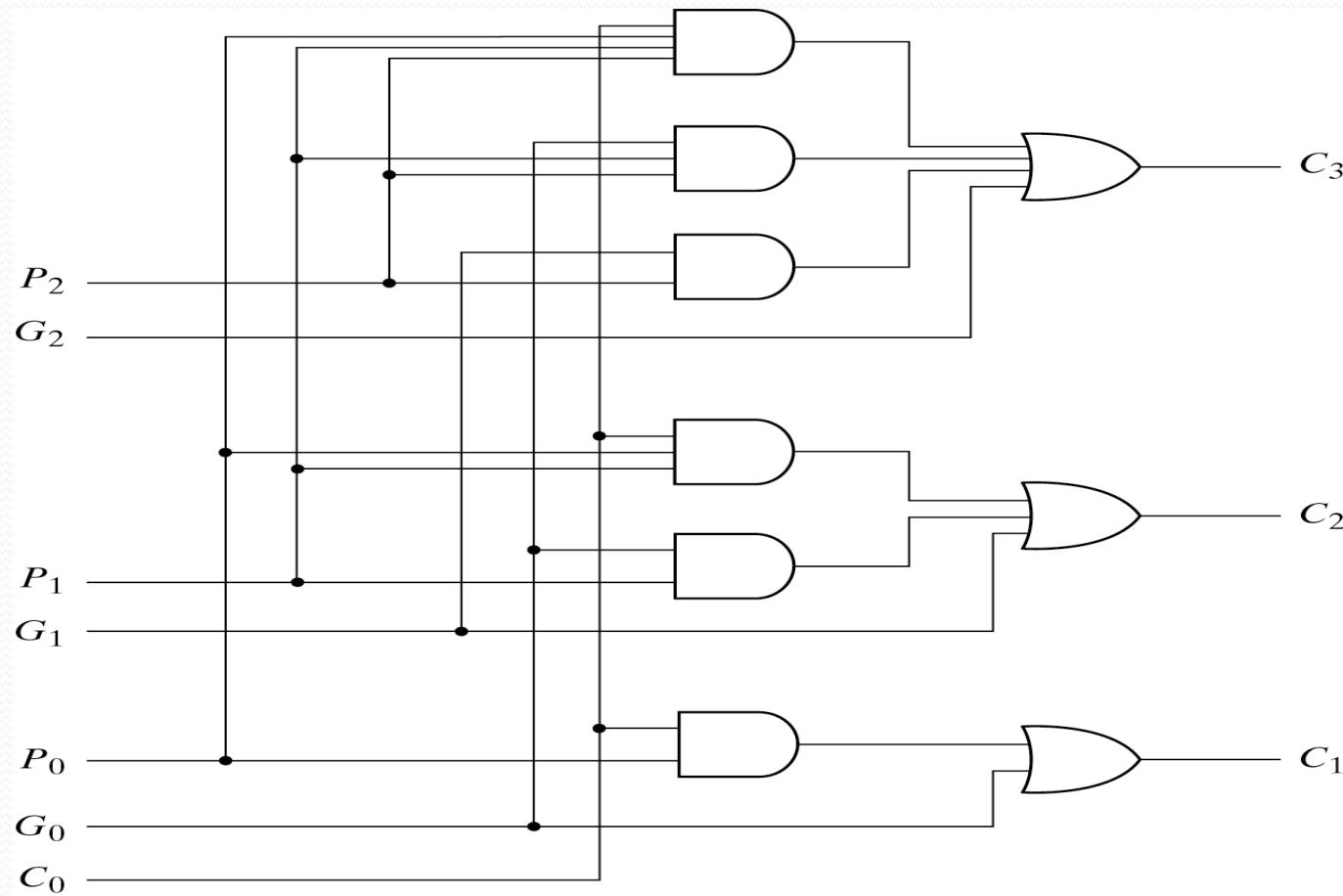


Fig. 4-11 Logic Diagram of Carry Lookahead Generator

4-bit adder with Carry Look ahead

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

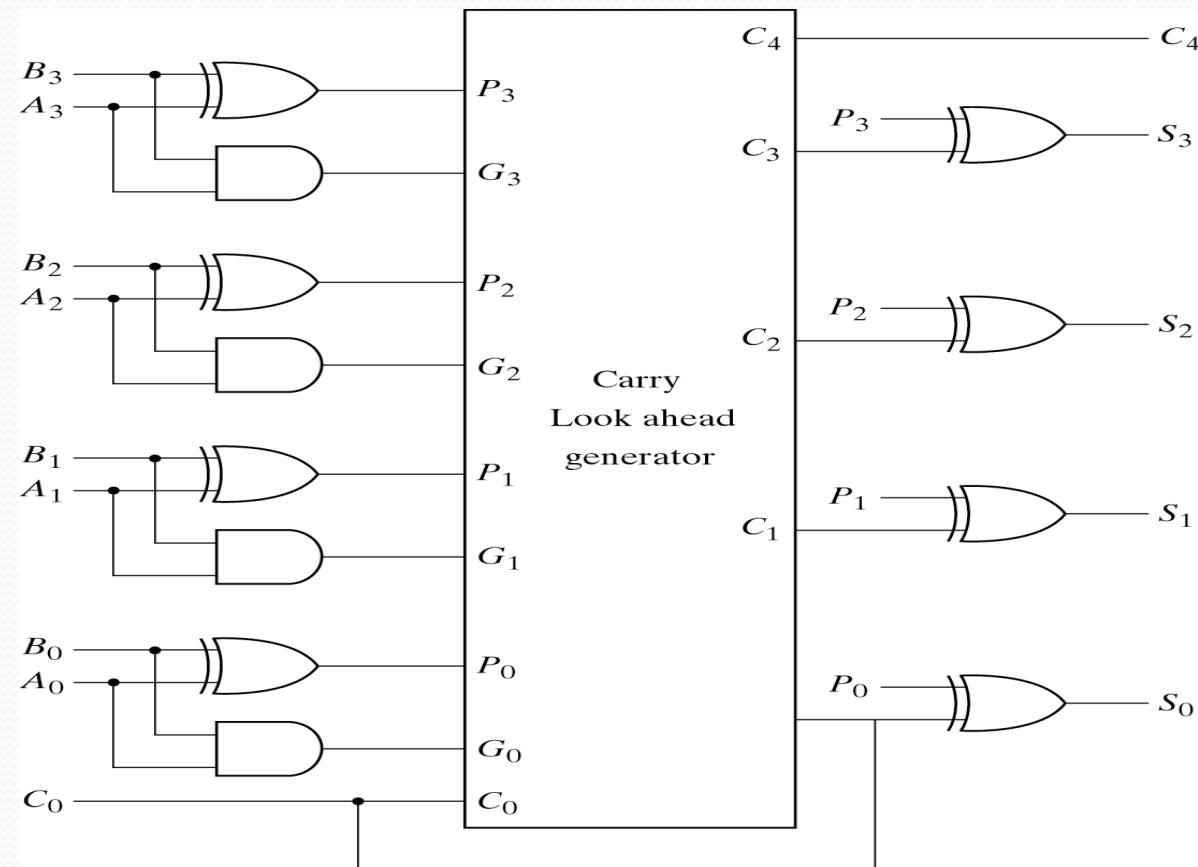


Fig. 4-12 4-Bit Adder with Carry Lookahead

THANK YOU

Digital Logic

CHAPTER 04

Lecture 18

Combinational Logic

Overview of previous lecture

- What is a Binary addition.
- What is Binary Adder

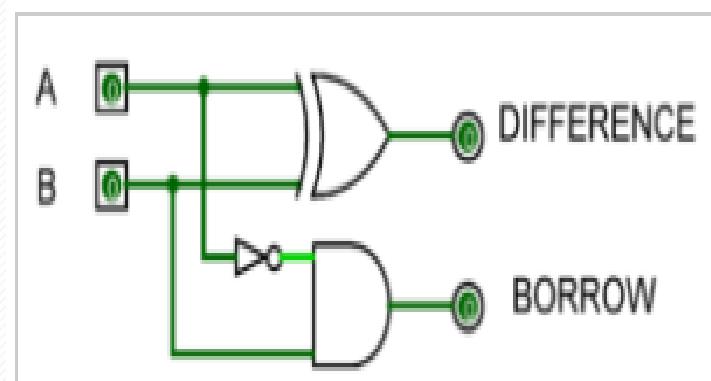
Half Subtractor

Half subtractor is a combination circuit with two inputs and two outputs (difference and borrow). It produces the difference between the two binary bits at the input and also produces an output (Borrow) to indicate if a 1 has been borrowed. In the subtraction (A-B), A is called as Minuend bit and B is called as Subtrahend bit.

Truth Table

Inputs		Output	
A	B	(A - B)	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Circuit Diagram

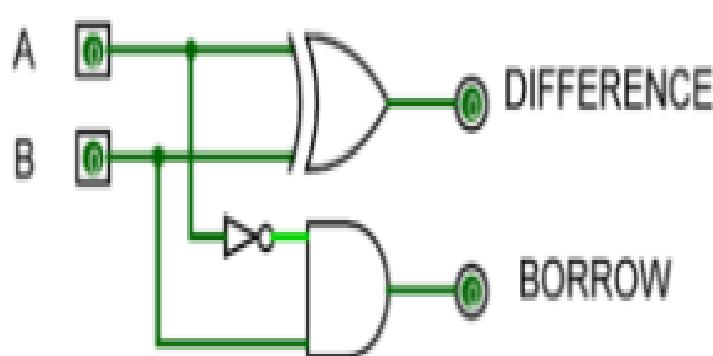


Half_Subtractor

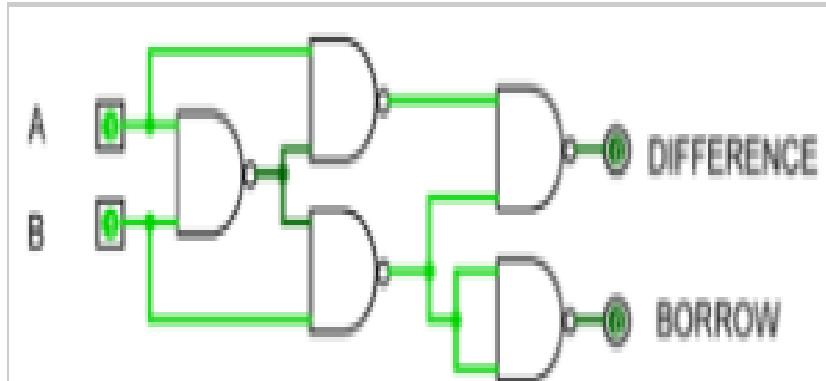
Implement HALF SUBTRACTOR using NAND GATES

$$D = A \oplus B$$

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



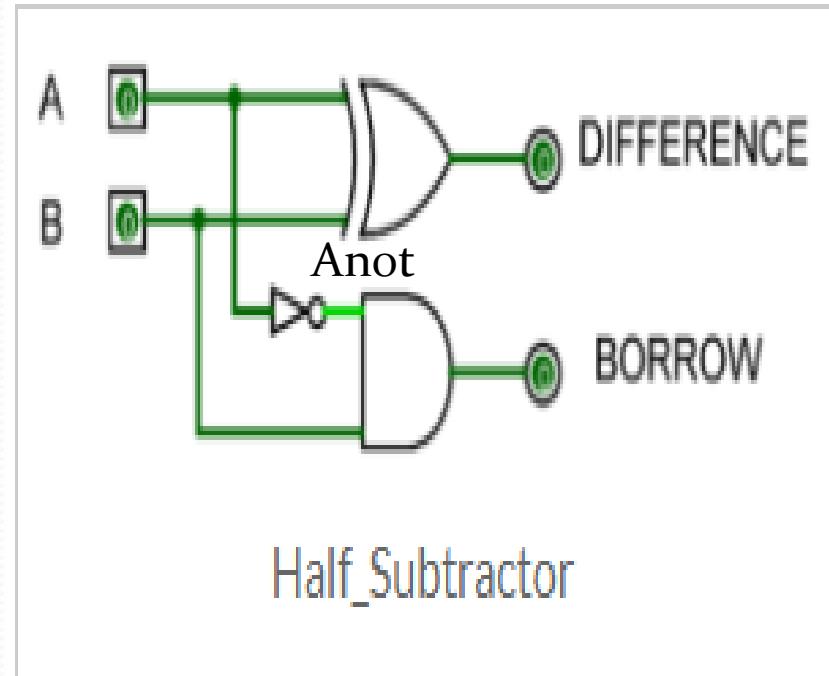
Half_Subtractor



Half_Subtractor_Nand

HDL for Half Subtractor

```
module half_subtractor
(output Difference, Borrow,
input A, B
);
wire Anot;
xor (Difference, A, B);
not (Anot, A);
and (Borrow, Anot, B);
endmodule
```



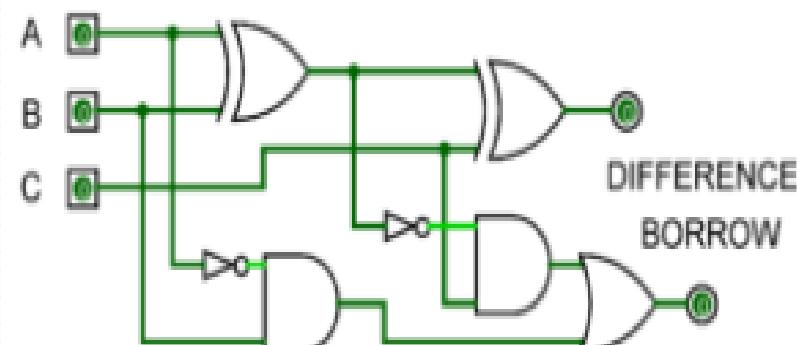
Full Subtractors

- The disadvantage of a half subtractor is overcome by full subtractor. The full subtractor is a combinational circuit with three inputs A,B,C and two output D and C'. A is the 'minuend', B is 'subtrahend', C is the 'borrow' produced by the previous stage, D is the difference output and C' is the borrow output.

Truth Table

Inputs			Output	
A	B	C	(A-B-C)	C'
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

Circuit Diagram

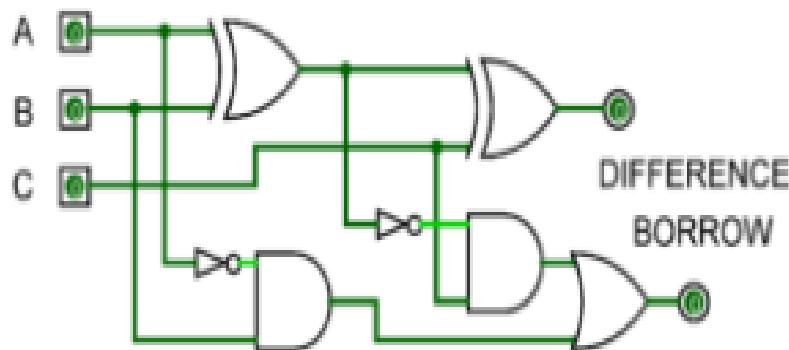


Full_Subtractor

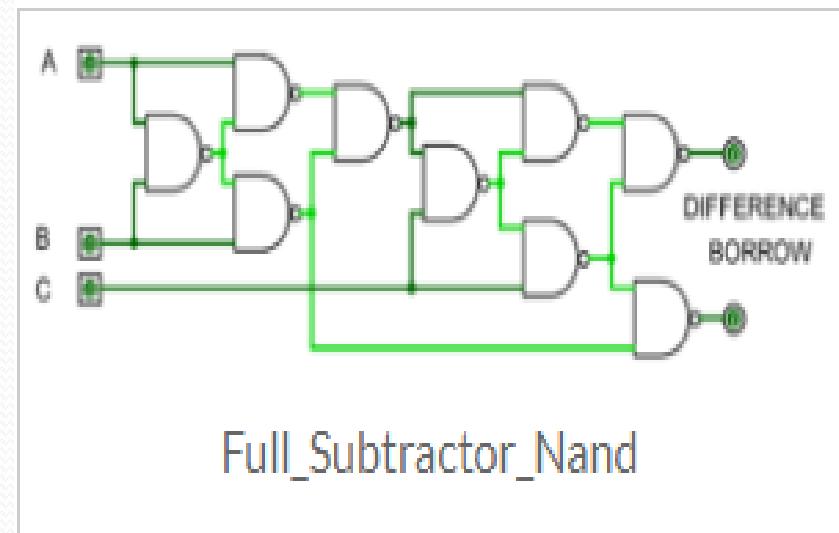
Realize Full SUBTRACTOR using NAND gates only

$$D = A \oplus B \oplus C$$

$$B = \bar{A}B + BC + \bar{A}C$$



Full_Subtractor



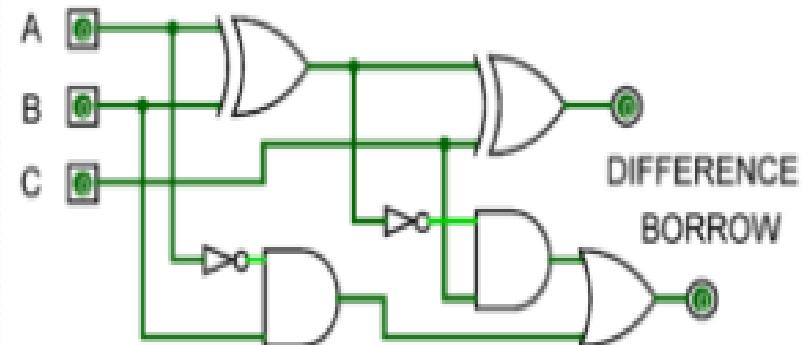
Full_Subtractor_Nand

HDL for Full Subtractor

```
// Verilog model Full Subtractor circuit
module Full-subtractor-df (difference,
borrow, A, B, C);
output difference, borrow;
input A, B, C;
assign difference = ((!A) && (!B) && (C))
|| ((!A) && (B) && (!C)) || ((A) && (!B) &&
(!C)) || (A && B && C);
assign borrow = ((!A) && B) || (B && C) ||
((!A) && C );
endmodule
```

$$D = A \oplus B \oplus C$$

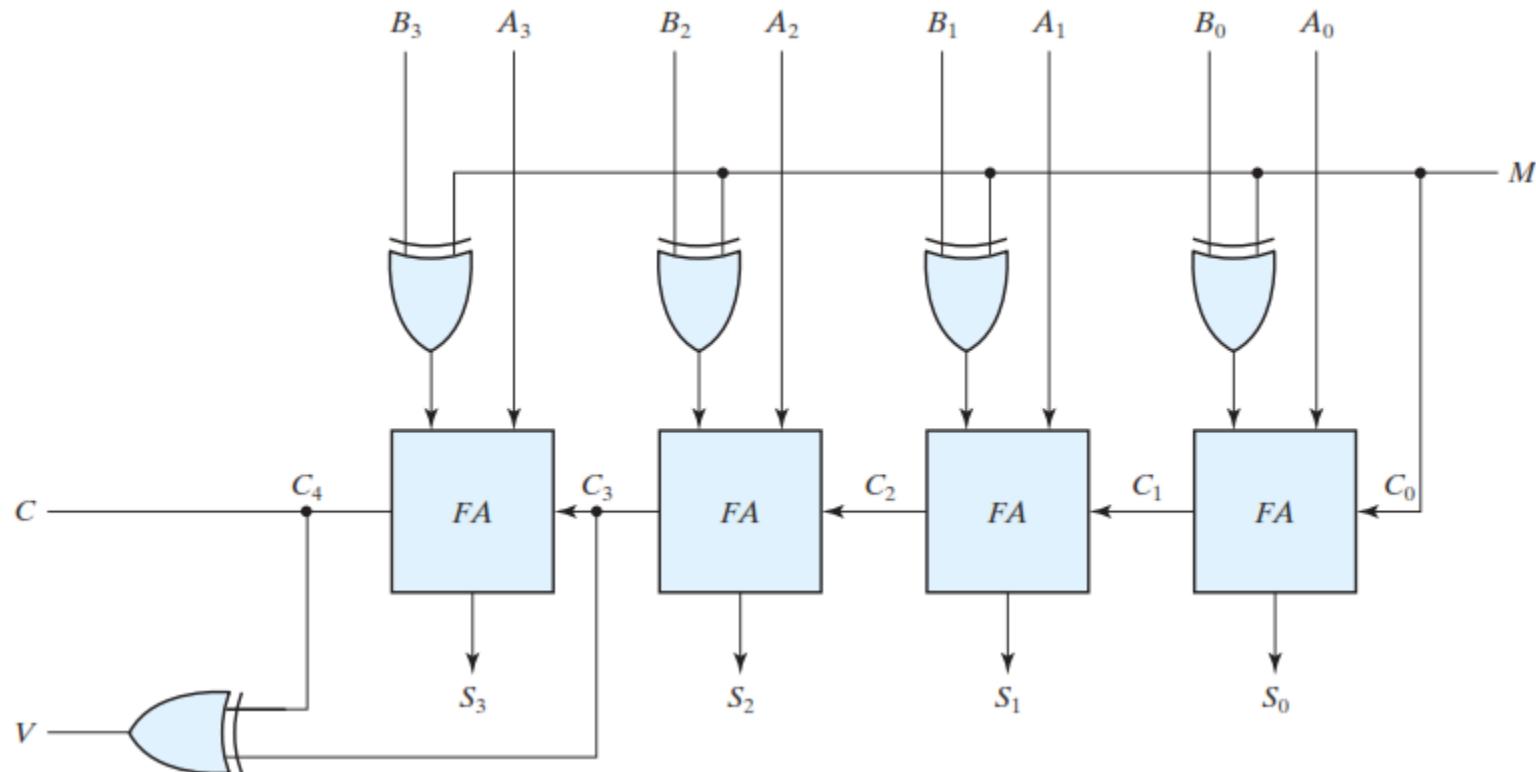
$$B = \overline{AB} + BC + \overline{AC}$$



Full_Subtractor

Binary Subtractor

The subtraction of unsigned binary numbers can be done most conveniently by means of complements. Remember that the subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A .



Four-bit adder–subtractor (with overflow detection)

- The circuit for subtracting $A - B$ consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C_o must be equal to 1 when subtraction is performed. The operation thus performed becomes A , plus the 1's complement of B , plus 1. This is equal to A plus the 2's complement of B .
- The addition and subtraction operations can be combined into one circuit with one common binary adder by including an exclusive-OR gate with each full adder. A four-bit adder-subtractor circuit as shown above.
- The mode input M controls the operation. When $M = 0$, the circuit is an adder, and when $M = 1$, the circuit becomes a subtractor.

Overflow

- When two numbers with n digits each are added and the sum is a number occupying $n + 1$ digits, we say that an **overflow** occurred.
- The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be **signed or unsigned**. When two **unsigned** numbers are added, an overflow is detected from the end carry out of the most significant position. In the case of **signed** numbers, two details are important:
- The leftmost bit always represents the sign, and negative numbers are in 2's-complement form. When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

- An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result whose magnitude is smaller than the larger of the two original numbers.
- An overflow may occur if the two numbers added are both positive or both negative.
- An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow has occurred.

carries: $+70$ $+80$ <hr/> $+150$	0 1 $0\ 1000110$ $0\ 1010000$ <hr/> $1\ 0010110$	carries: -70 -80 <hr/> -150	1 0 $1\ 0111010$ $1\ 0110000$ <hr/> $0\ 1101010$
---	--	---	--

- This is indicated in the examples in which the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow is detected when the output of the gate is equal to 1.
- The binary adder-subtractor circuit with outputs C and V is shown in above Fig. If the two binary numbers are considered to be unsigned, then the C bit detects a carry after addition or a borrow after subtraction. If the numbers are considered to be signed, then the V bit detects an overflow.

THANK YOU

DIGITAL LOGIC

LECTURE-19



Code conversion

- The availability of a large variety of codes for the same discrete elements of information results in the use of different codes by different digital systems.
- A **conversion circuit** must be inserted between the two systems if each uses different codes for the same information.
- Thus, a **code converter** is a circuit that makes the two systems compatible even though each uses a different binary code.



- To convert from binary code A to binary code B, the input lines must supply the bit combination of elements as specified by code A and the output lines must generate the corresponding bit combination of code B. A combinational circuit performs this transformation by means of logic gates.
- The design procedure will be illustrated by an example that converts binary coded decimal (BCD) to the excess-3 code for the decimal digits.

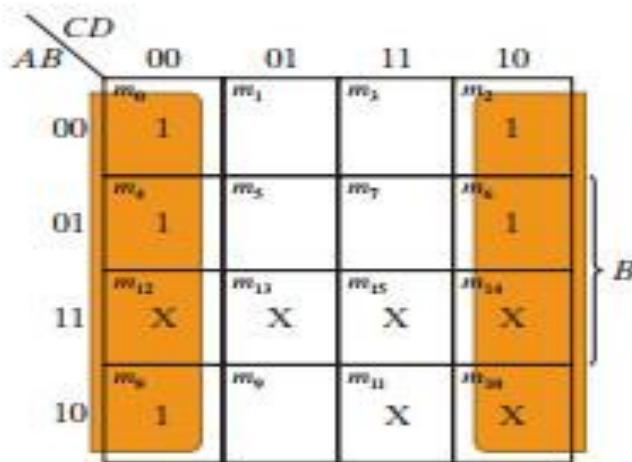


Truth Table for Code Conversion Example

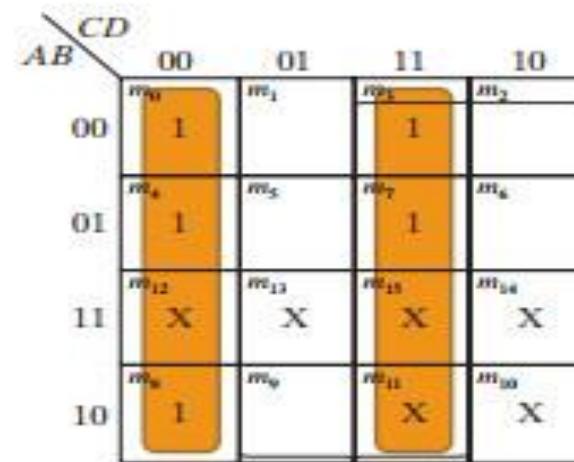
Input BCD				Output Excess-3 Code			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

We designate the four input binary variables by the symbols A, B, C, and D, and the four output variables by w, x, y , and z .

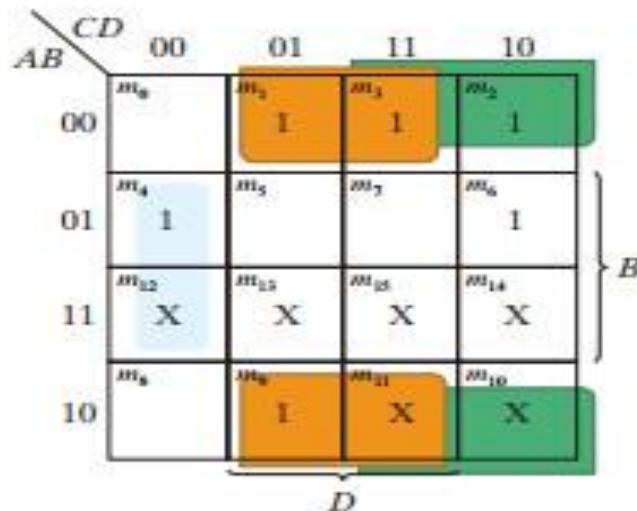
Maps for BCD-to-excess-3 code converter



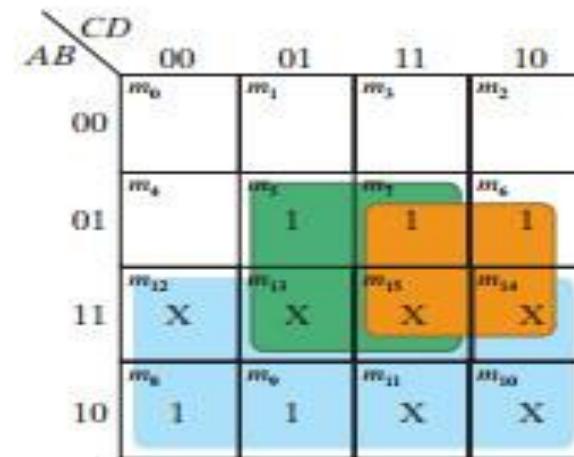
$$z = D'$$



$$y = CD + C'D'$$



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



$$w = A + BC + BD$$

So, finally the output expression are as follows:

$$z = D'$$

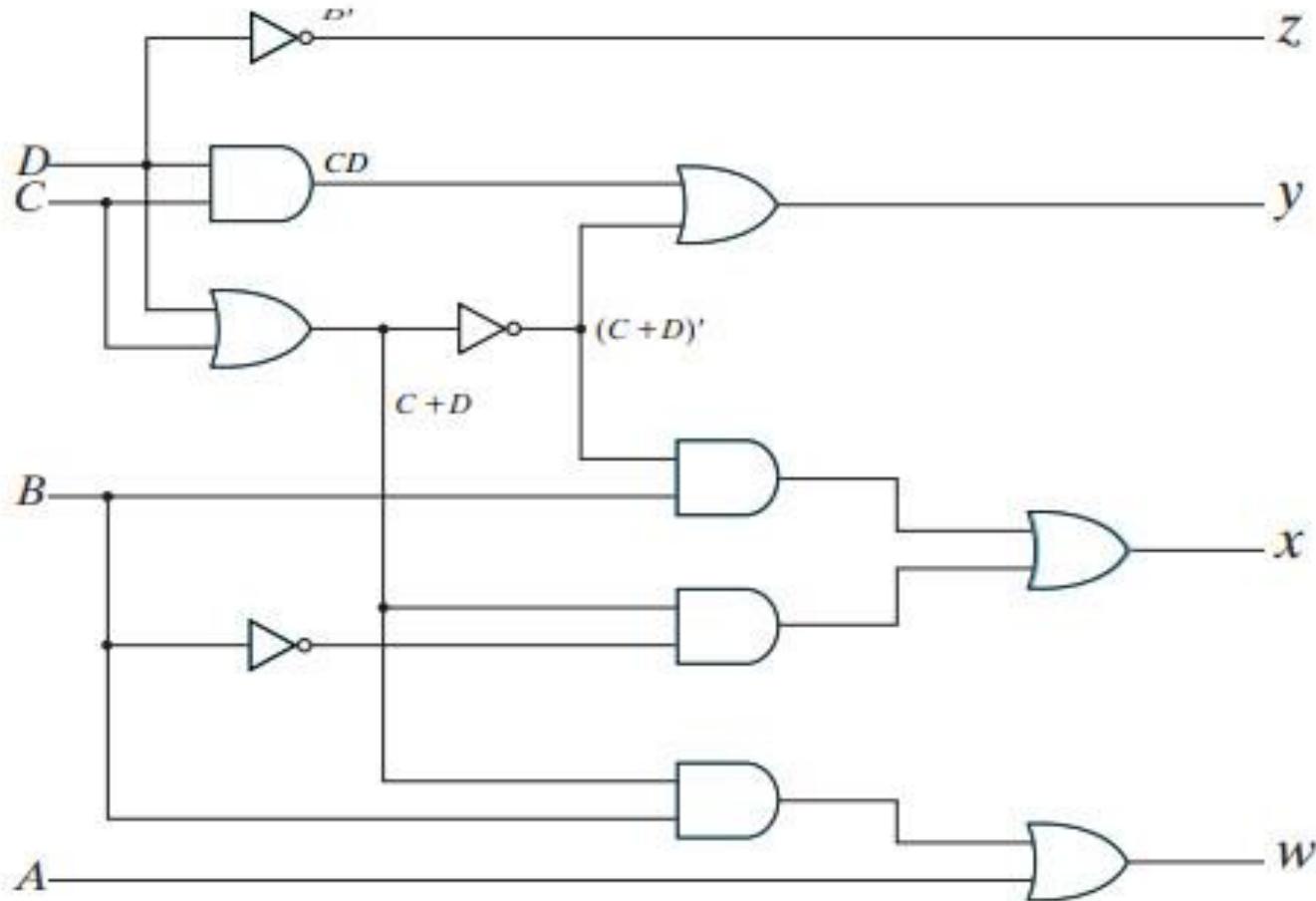
$$y = CD + C'D' = CD + (C + D)'$$

$$\begin{aligned}x &= B'C + B'D + BC'D' = B'(C + D) + BC'D' \\&= B'(C + D) + B(C + D)'\end{aligned}$$

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



Logic diagram for BCD-to-excess-3 code converter



Design a combinational circuit that converts a four-bit binary number to a four bit Gray code .

The gray code is a **non-weighted code**. The successive gray code differs in one-bit position only that means it is a **unit distance code**. It is also referred as a **cyclic code**. It is not suitable for arithmetic operations. It is the most popular of the unit distance codes. It is also a **reflective code**. An n-bit Gray code can be obtained by reflecting an n-1 bit code about an axis after 2^{n-1} rows and putting the MSB of 0 above the axis and the MSB of 1 below the axis. Reflection of the 4 bits binary to gray code conversion table is given below:

The Truth Table for binary –Gray Code Convertor Circuit

Decimal Number	4 bit Binary Number <u>ABCD</u>	4 bit Gray Code <u>G₁G₂G₃G₄</u>
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1
10	1 0 1 0	1 1 1 1
11	1 0 1 1	1 1 1 0
12	1 1 0 0	1 0 1 0
13	1 1 0 1	1 0 1 1
14	1 1 1 0	1 0 0 1
15	1 1 1 1	1 0 0 0



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

$$G_4 = A$$

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

$$G_3 = \bar{A}\bar{B} + A\bar{B} = A \oplus B$$

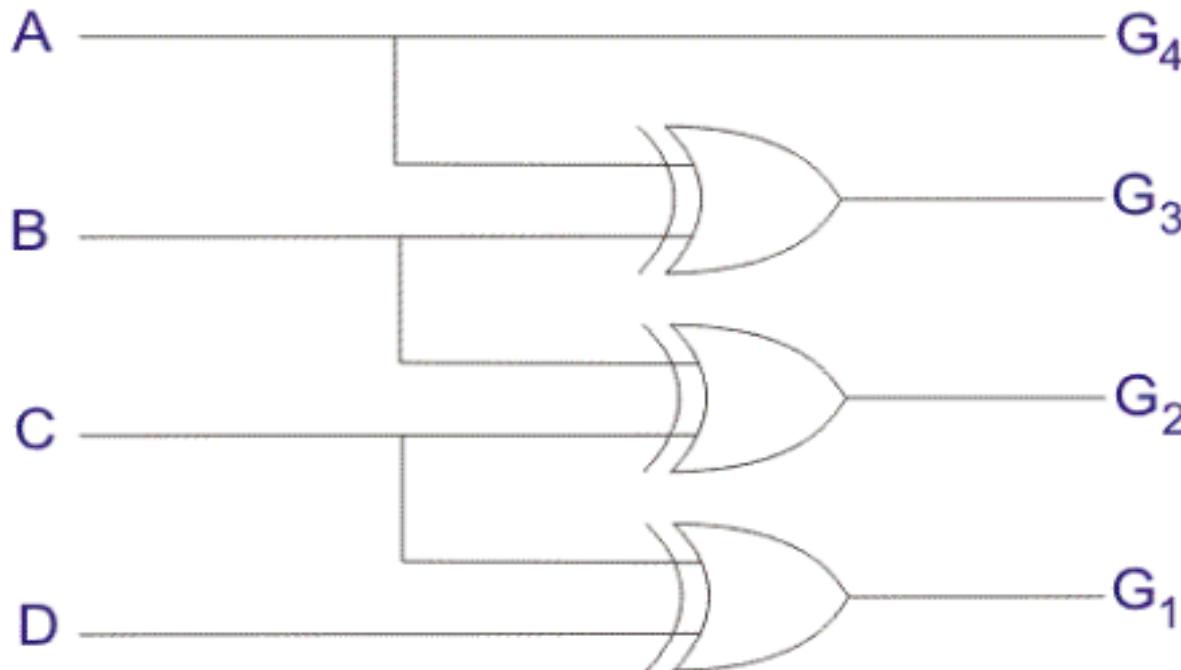
	CD	00	01	11	10
AB	00	0	1	1 3	1 2
	01	1 4	1 5	7	6
	11	1 12	1 13	15	14
	10	8	9	1 11	1 10

$$G_2 = BC + \bar{B}C = B \oplus C$$

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

$$G_1 = \bar{C}\bar{D} + C\bar{D} = C \oplus D$$

Logic Diagram for Binary – Gray Code Convertor



Design a code converter that converts a decimal digit from “8, 4, -2, -1” code to BCD.

The Truth Table



	8	4	-2	-1	8	4	2	1
	A	B	C	D	w	x	y	z
0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	1
2	0	1	1	0	0	0	1	0
3	0	1	0	1	0	0	1	1
4	0	1	0	0	0	1	0	0
5	1	0	1	1	0	1	0	1
6	1	0	1	0	0	1	1	0
7	1	0	0	1	0	1	1	1
8	1	0	0	0	1	0	0	0
9	1	1	1	1	1	0	0	1

↔ C ↔

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

A ↓

$$w = AB + AC'D'$$

↔ C ↔

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

A ↓

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

↔ C ↔

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

A ↓

$$y = C'D + CD'$$

and $z = D$



**THANK
YOU**



DIGITAL LOGIC

LECTURE-20

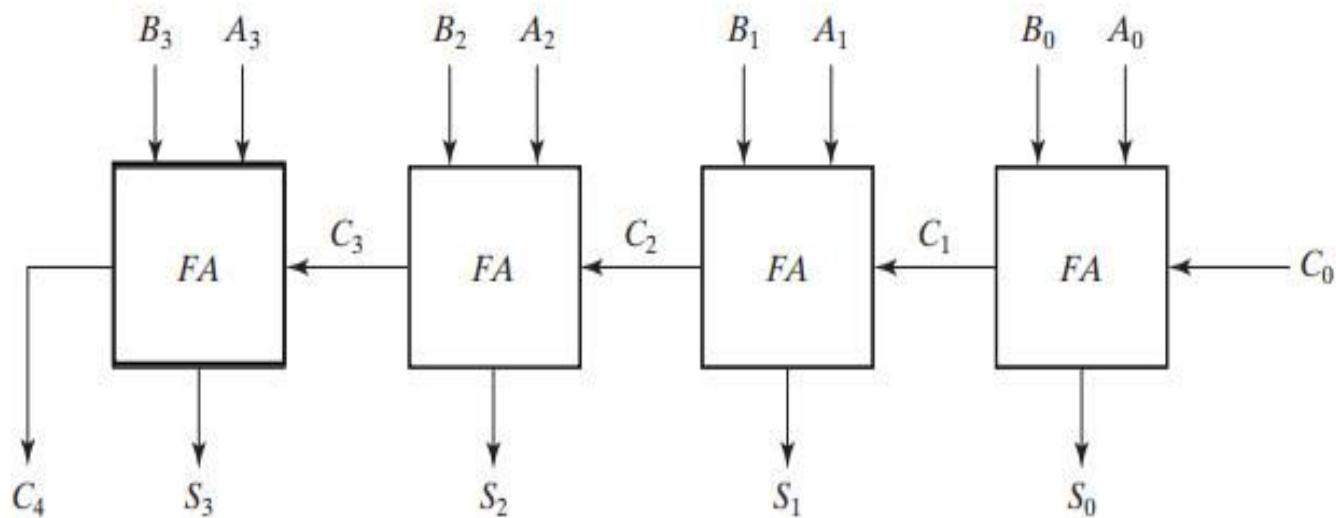


Combinational Logic with MSI & LSI: Binary Parallel adder

- A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers.
- It 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.



- Addition of **n-bit numbers** requires a chain of **n** full adders or a chain of **one-half adder** and **n - 1** full adders.



Four-bit adder

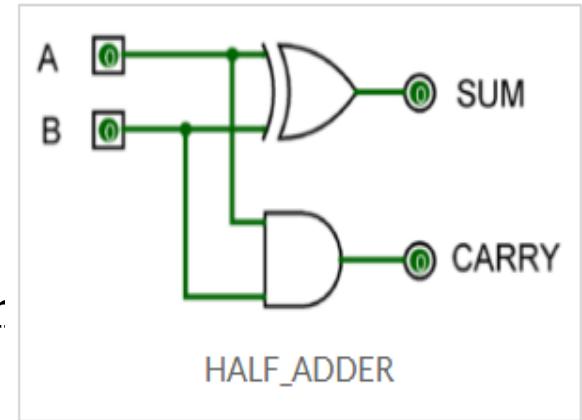
- The above figure shows the interconnection of 4 full-adder (FA) circuits to provide a **4-bit binary ripple carry** adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the least significant bit.
- The carries are connected in a chain through the full adders.
- The input carry to the adder is C_0 , and it ripples through the full adders to the output carry C_4 .
- The S outputs generate the required sum bits.
- **An n -bit adder requires n full adders**, with each output carry connected to the input carry of the next higher order full adder.

- To demonstrate with a specific example, consider the two binary numbers $A = 1011$ and $B = 0011$. Their sum $S = 1110$ is formed with the four-bit adder as follows:

Subscript <i>i</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}

HDL for Binary Adder

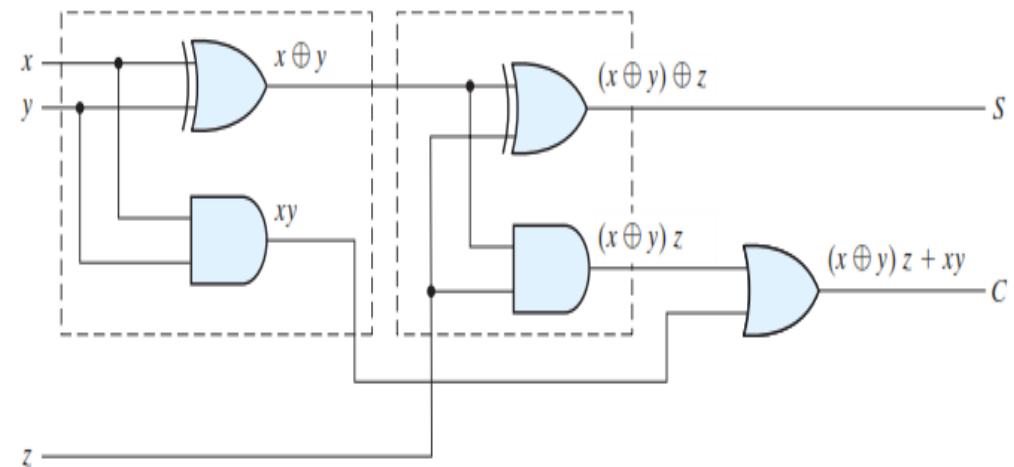
```
// Gate-level description of four-bit ripple carry adder
// Description of half adder
// module half_adder (S, C, x, y);
// output S, C;
// input x, y;
module half_adder ( output S, C, ir
// Instantiate primitive gates
xor (S, x, y);
and (C, x, y);
endmodule
// Description of full adder
// module full_adder (S, C, x, y, z);
// output S, C;
// input x, y, z;
```



```

module full_adder (output S, C, input x, y, z);
wire S1, C1, C2;
// Instantiate half adders
half_adder HA1 (S1, C1, x, y)
half_adder HA2 (S, C2, S1, z
or G1 (C, C2, C1);
endmodule

```



```

// Description of four-bit adder
// module ripple_carry_4_bit_adder (Sum, C4, A, B, C0);
// output [3: 0] Sum;
// output C4;
// input [3: 0] A, B;
// input C0;

```

```
// Alternative Verilog 2001, 2005 syntax:  
module ripple_carry_4_bit_adder ( output [3: 0] Sum,  
    output C4,  
    input [3: 0] A, B, input C0);  
wire C1, C2, C3; // Intermediate carries  
// Instantiate chain of full adders  
full_adder FA0 (Sum[0], C1, A[0], B[0], C0),  
    FA1 (Sum[1], C2, A[1], B[1], C1),  
    FA2 (Sum[2], C3, A[2], B[2], C2),  
    FA3 (Sum[3], C4, A[3], B[3], C3);  
endmodule
```



Advantage

The four-bit adder is a typical example of a standard component. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit by the classical method would require a truth table with $2^9 = 512$ entries, since there are nine inputs to the circuit. By using an iterative method of cascading a standard function, it is possible to obtain a simple and straightforward implementation.

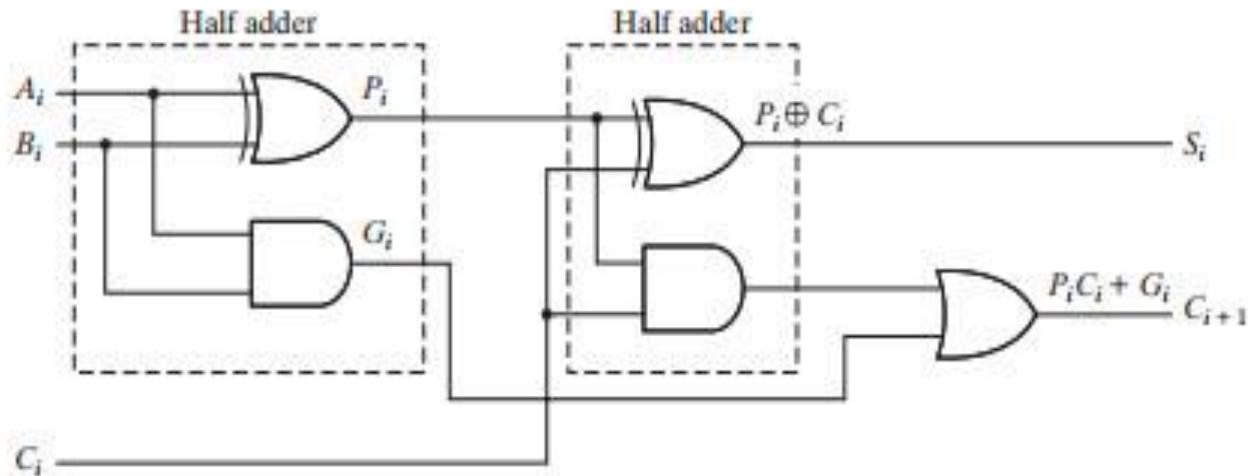


Disadvantage

- The addition of two binary numbers in parallel implies that all the bits of the augend and addend are available for computation at the same time. The longest propagation delaytime in an adder is the time it takes the carry to propagate through the full adders.
- Since each bit of the sum output depends on the value of the input carry, the value of S_i at any given stage in the adder will be in its steady-state final value only after the input carry to that stage has been propagated.



Carry Propagation



Full adder with P and G shown

- The input and output variables use the subscript i to denote a typical stage of the adder. The signals at P_i and G_i settle to their steady-state values after they propagate through their respective gates.
- 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.
- If there are four full adders in the adder, the output carry C_{i+1} would have $2 * 4 = 8$ gate levels from C_0 to C_4 .
- For an n -bit adder, there are 2^n gate levels for the carry to propagate from input to output.



- The carry propagation time is an important attribute of the adder because it limits the speed with which two numbers are added.
- Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is critical.
- An obvious solution for reducing the carry propagation delay time is to employ faster gates with reduced delays.
- Solution is to increase the complexity of the equipment in such a way that the carry delay time is reduced.
- The most widely used technique employs the principle of *carry lookahead logic*.

If we define two new binary

$$P_i = A_i \oplus B_i$$

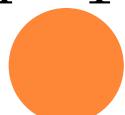
$$G_i = A_i B_i$$

the output sum and carry can respectively be expressed as

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

G_i is called a *carry generate*, and it produces a carry of 1 *when both* A_i and B_i are 1, regardless of the input carry C_i . P_i is called a *carry propagate*, because it determines whether a carry into stage i will propagate into stage $i + 1$.



- We now write the Boolean functions for the carry outputs of each stage and substitute the value of each C_i from the previous equations:

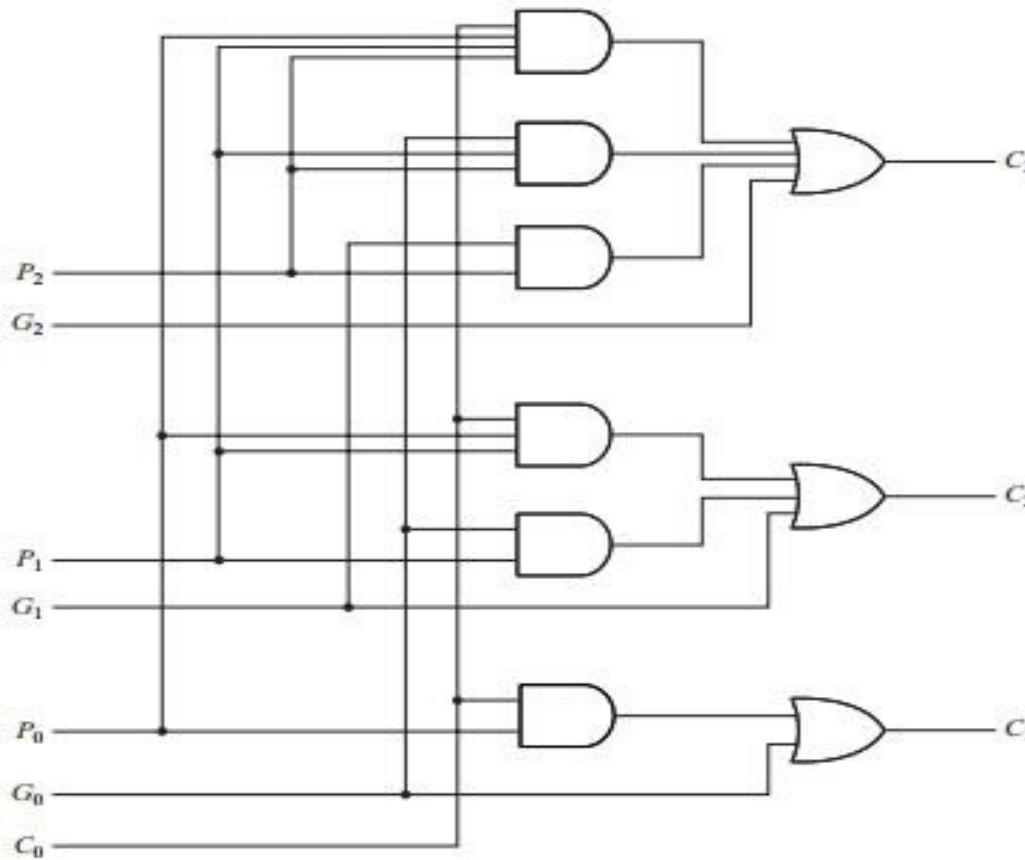
$$C_0 = \text{input carry}$$

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$

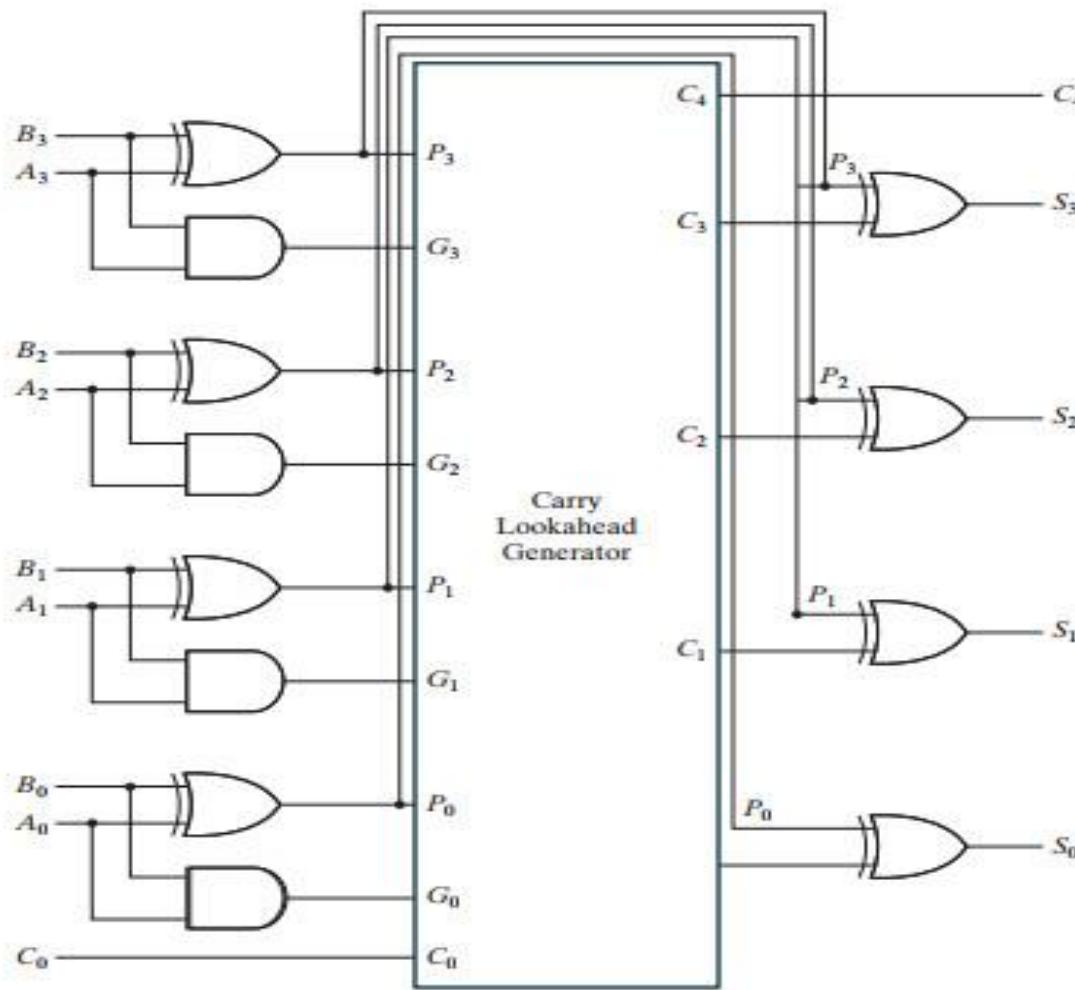
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 = P_2 P_1 P_0 C_0$$

- The three Boolean functions for C_1 , C_2 , and C_3 are implemented in the carry lookahead generator shown in Fig. below



Logic diagram of carry lookahead generator

The construction of a four-bit adder with a carry lookahead scheme is shown below:



Four-bit adder with carry lookahead

THANK YOU



DIGITAL LOGIC

LECTURE-21



○ Decimal Adder

- An adder for a computer that employ arithmetic circuits that accept coded decimal numbers and present results in the same code.
- A decimal adder requires a minimum of nine inputs and five outputs, since four bits are required to code each decimal digit and the circuit must have an input and output carry.
- There is a wide variety of possible decimal adder circuits, depending upon the code used to represent the decimal digits.
- Here we examine a decimal adder for the BCD code.



BCD Adder

- Consider the arithmetic addition of two decimal digits in BCD, together with an input carry from a previous stage.
- Since each input digit does not exceed 9, the output sum cannot be greater than $9 + 9 + 1 = 19$, the 1 in the sum being an input carry.
- Suppose we apply two BCD digits to a four-bit binary adder.



Derivation of BCD Adder

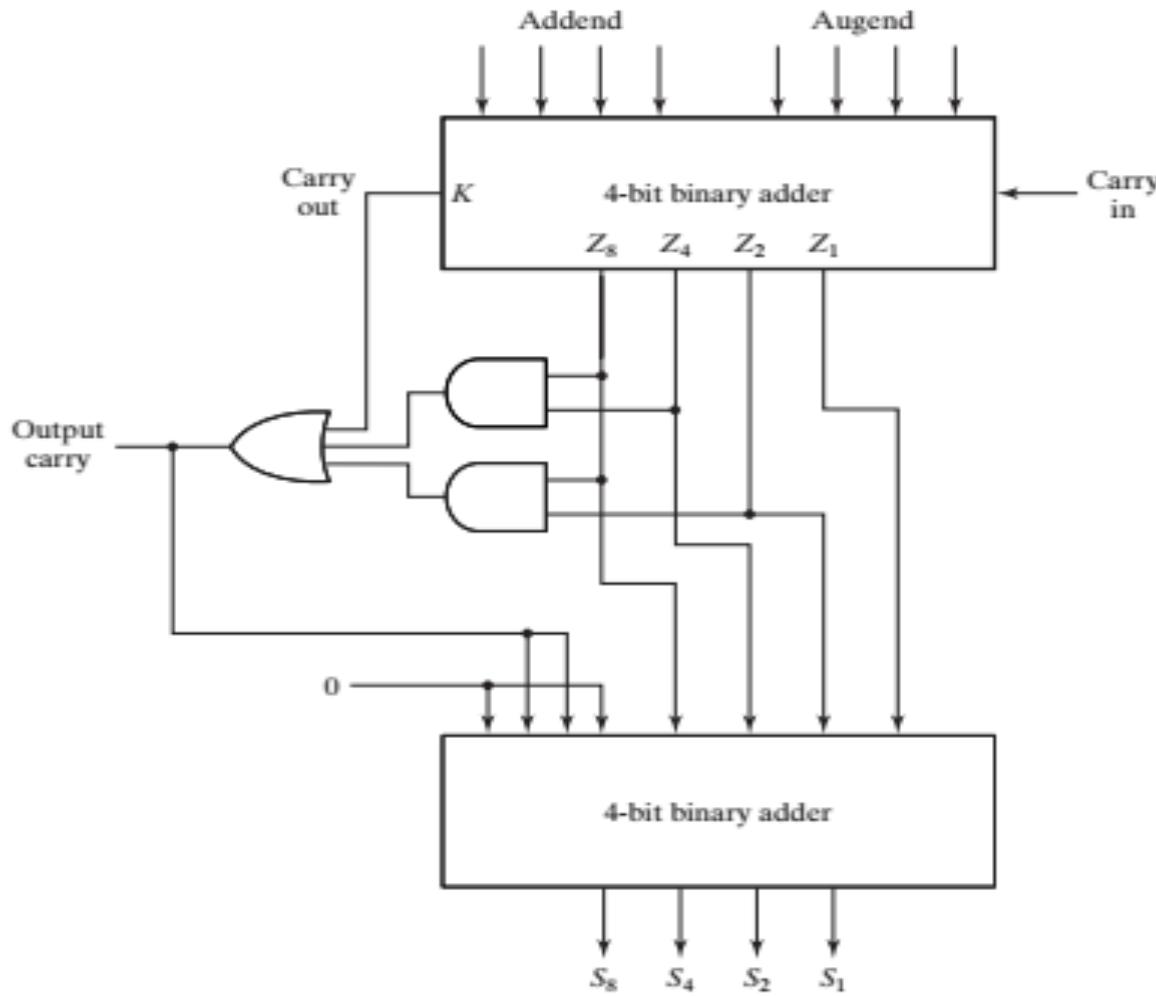
K	Binary Sum					BCD Sum					Decimal
	Z ₈	Z ₄	Z ₂	Z ₁	C	S ₈	S ₄	S ₂	S ₁		
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1	1
0	0	0	1	0	0	0	0	1	0	2	2
0	0	0	1	1	0	0	0	1	1	3	3
0	0	1	0	0	0	0	1	0	0	4	4
0	0	1	0	1	0	0	1	0	1	5	5
0	0	1	1	0	0	0	1	1	0	6	6
0	0	1	1	1	0	0	1	1	1	7	7
0	1	0	0	0	0	1	0	0	0	8	8
0	1	0	0	1	0	1	0	0	1	9	9
0	1	0	1	0	1	0	0	0	0	10	10
0	1	0	1	1	1	0	0	0	1	11	11
0	1	1	0	0	1	0	0	1	0	12	12
0	1	1	0	1	1	0	0	1	1	13	13
0	1	1	1	0	1	0	1	0	0	14	14
0	1	1	1	1	1	0	1	0	1	15	15
1	0	0	0	0	1	0	1	1	0	16	16
1	0	0	0	1	1	0	1	1	1	17	17
1	0	0	1	0	1	1	0	0	0	18	18
1	0	0	1	1	1	1	0	0	1	19	19



- The logic circuit that detects the necessary correction can be derived from the entries in the table. It is obvious that a correction is needed when the binary sum has an output carry $K = 1$.
- The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_8Z_4 + Z_8Z_2$$





Block diagram of a BCD adder

Magnitude Comparator

- A *magnitude comparator* is a combinational circuit that compares two numbers A and B and determines their relative magnitudes.
- The outcome of the comparison is specified by three binary variables that indicate whether $A > B$, $A = B$ or $A < B$.
- Consider two numbers, A and B , with four digits each. Write the coefficients of the numbers in descending order of significance:
$$A = A_3 \ A_2 \ A_1 \ A_0$$

$$B = B_3 \ B_2 \ B_1 \ B_0$$



The two numbers are **equal** if all pairs of significant digits are equal: $A_3=B_3, A_2=B_2, A_1=B_1$ and $A_0=B_0$

$$x_i = A_i B_i + A'_i B'_i \quad \text{for } i = 0, 1, 2, 3$$

The binary variable ($A = B$) is equal to 1 only if all pairs of digits of the two numbers are equal.

$$(A = B) = x_3 x_2 x_1 x_0$$



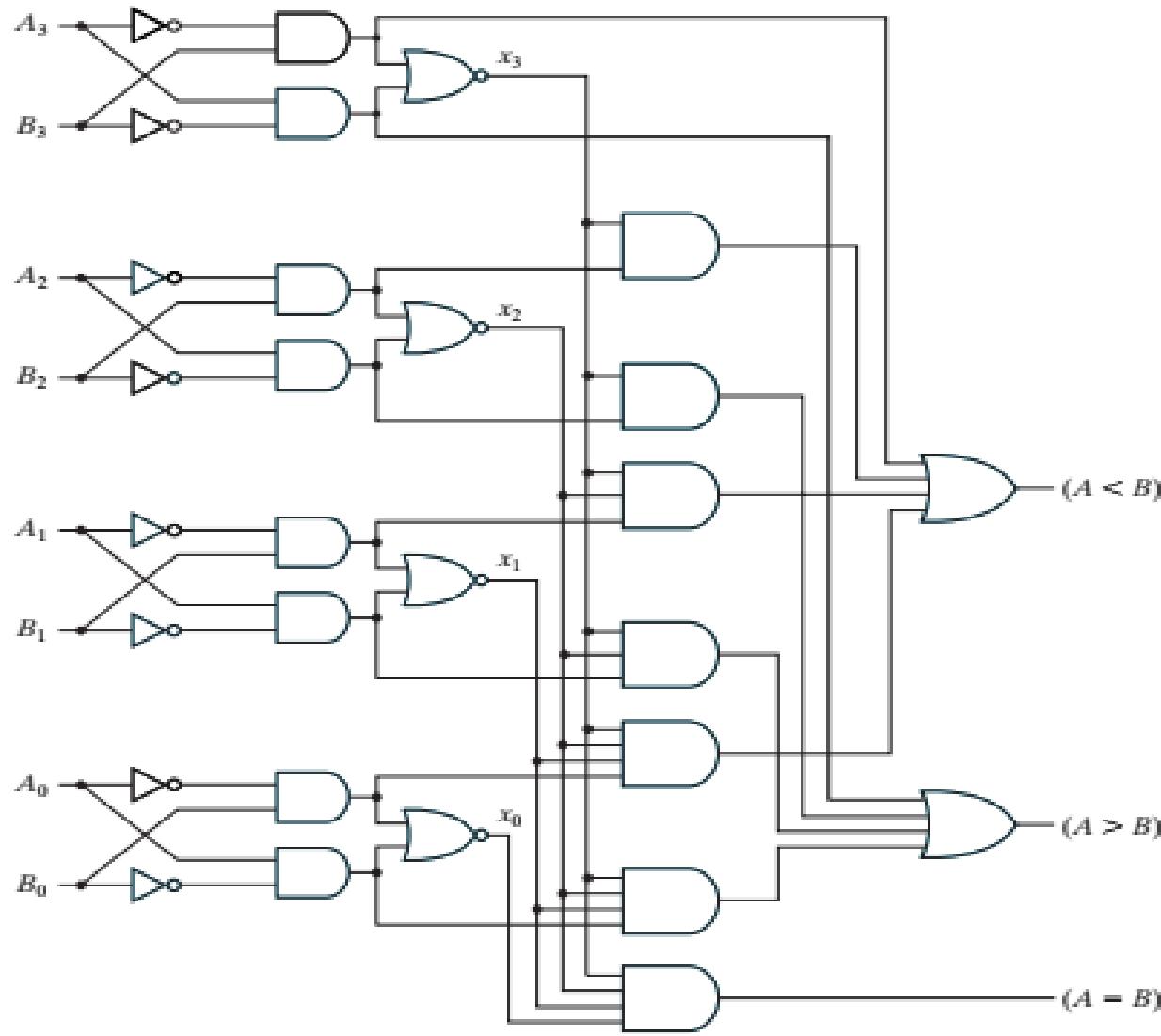
To determine whether A is greater or less than B , we inspect the relative magnitudes of pairs of significant digits, starting from the most significant position.

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

$$(A < B) = A'_3B_3 + x_3A'_2B_2 + x_3x_2A'_1B_1 + x_3x_2x_1A'n_0B'_0$$

The symbols $A > B$ and $A < B$ are binary output variables that are equal to 1 when $A > B$ and $A < B$, respectively.





Four-bit magnitude comparator

Example

Design a combinational circuit that compares two 4-bit numbers to check if they are equal. The circuit output is equal to 1 if the two numbers are equal and 0 otherwise.

$$x_i = A_i B_i + A'_i B'_i \quad \text{for } i = 0, 1, 2, 3$$

$$(A = B) = x_3 x_2 x_1 x_0$$



HDL for 4 bits Magnitude Comparator

```
// Dataflow description of a four-bit comparator
module mag_compare
( output A_lt_B, A_eq_B, A_gt_B,
input [3: 0] A, B);
assign A_lt_B = (A < B);
assign A_gt_B = (A > B);
assign A_eq_B = (A == B);
endmodule
```



THANK YOU



LOGIC DESIGN

EET-1021

CHAPTER 04

Lecture 22

Combinational Logic

Overview of previous lecture

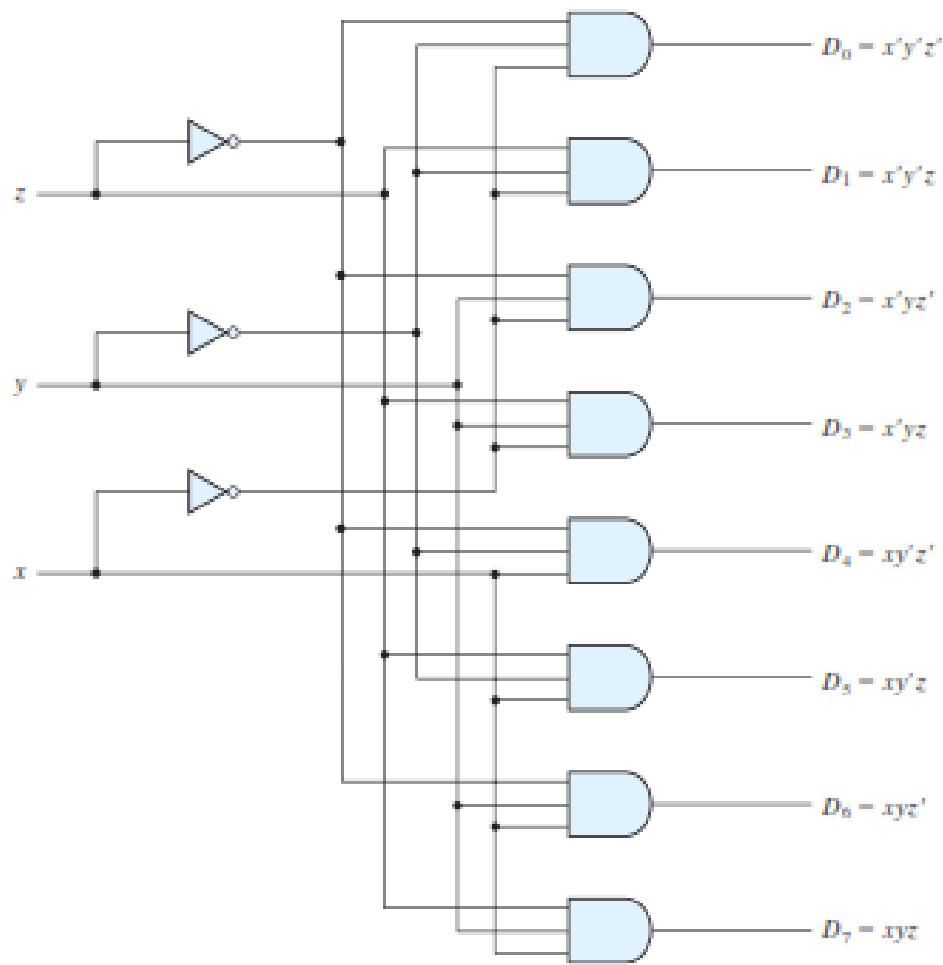
- **What is Decimal adder**
- **How to construct this adder circuit**
- **What is a Magnitude Comparator circuit**

Decoder

- A *decoder* is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines.
- If the n -bit coded information has unused combinations, the decoder may have fewer than 2^n outputs.
- The decoders presented here are called *n -to- m -line decoders*, where $m \leq 2^n$
- The name *decoder* is also used in conjunction with other code converters, such as a BCD-to-seven-segment decoder.

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

For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1. The output whose value is equal to 1 represents the minterm equivalent of the binary number currently available in the input lines.



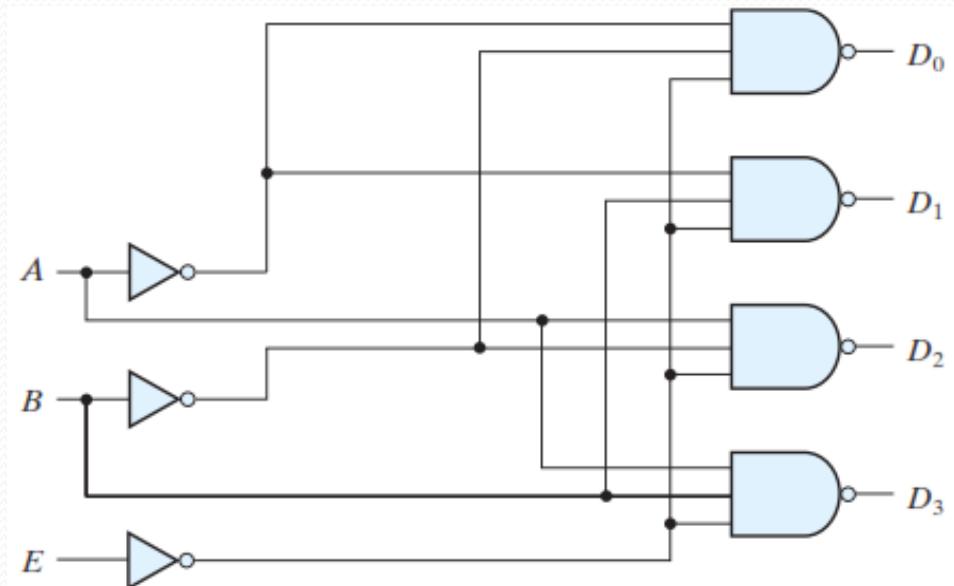
Three-to-eight-line decoder

A decoders include one or more *enable inputs* to control the circuit operation. A two-to-four-line decoder with an enable input is constructed with NAND gates.

The circuit operates with complemented outputs and a complement enable input. The decoder is enabled when E is equal to 0 (i.e., active-low enable).

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	1

Truth table



Logic Diagram

- A decoder with enable input can function as a *demultiplexer*—
a circuit that receives information from a single line and directs it to one of 2^n possible output lines. The selection of a specific output is controlled by the bit combination of n selection lines.
- The decoder above shown can function as a one-to-four-line demultiplexer when E is taken as a data input line and A and B are taken as the selection inputs.
- *The single input variable E has a path to all four outputs, but the input information is directed to only one of the output lines, as specified by the binary combination of the two selection lines A and B .*
- As decoder and demultiplexer operations are obtained from the same circuit, a decoder with an enable input is referred to as a *decoder – demultiplexer*.

HDL for 2 to 4 line Decoder:

```
// Gate-level description of two-to-four-line decoder
```

```
module decoder_2x4_gates (D, A, B, enable);
```

```
output [o:3] D;
```

```
input A, B, enable;
```

```
wire A_not,B_not,enable_not;
```

```
not
```

```
G1 (A_not, A),
```

```
G2 (B_not, B),
```

```
G3 (enable_not, enable);
```

```
nand
```

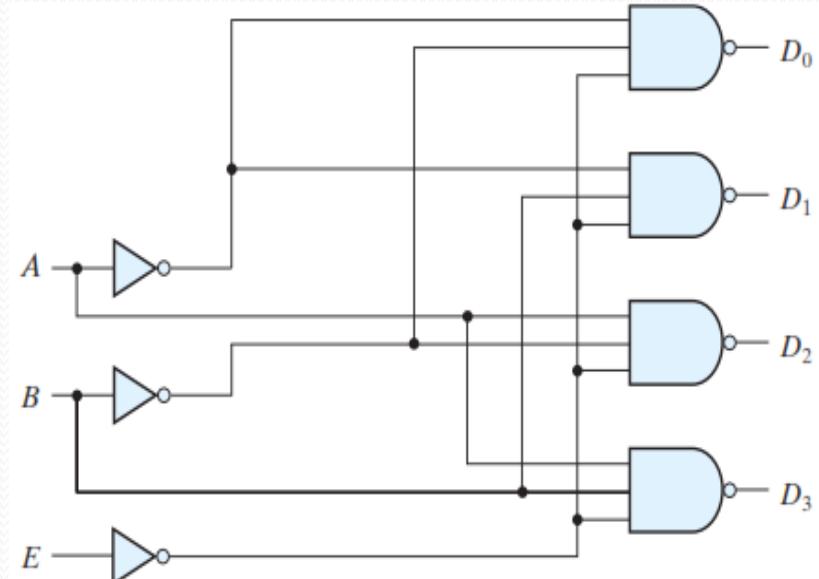
```
G4 (D[o], A_not, B_not, enable_not),
```

```
G5 (D[1], A_not, B, enable_not),
```

```
G6 (D[2], A, B_not, enable_not),
```

```
G7 (D[3], A, B, enable_not);
```

```
endmodule
```



Dataflow Modeling

```
//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)
```

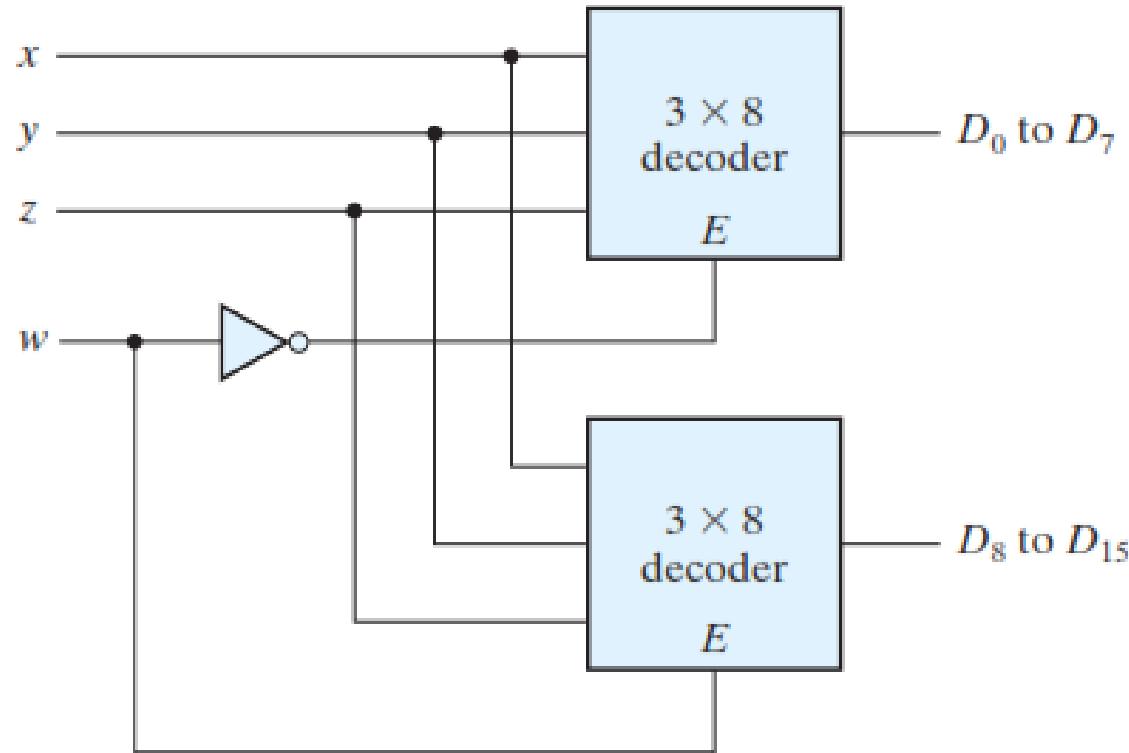
Behavioral Model

```
module decoder(sel,en,y);
    input [1:0] sel;
    input en;
    output [3:0] y;

    reg y;

    always @ (sel, en)
    begin // required if multiple sequential statements
        if (en == 0)
            y = 4'b1111;
        else
            if (sel == 2'b00)
                y = 4'b1110;
            else if (sel == 2'b01)
                y = 4'b1101;
            else if (sel == 2'b10)
                y = 4'b1011;
            else
                y = 4'b0111;
    end
endmodule
```

Decoders with enable inputs can be connected together to form a larger decoder circuit



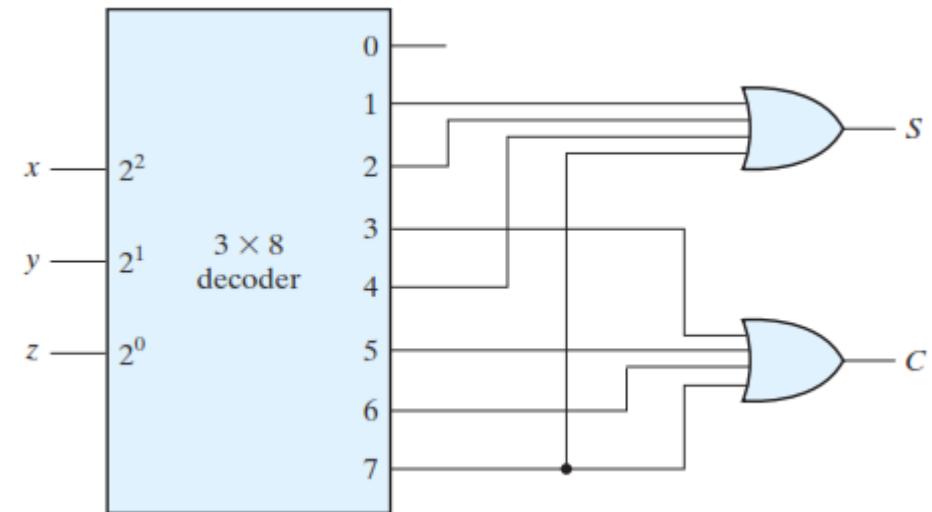
4×16 decoder constructed with two 3×8 decoders

Combinational Logic Implementation

Since any Boolean function can be expressed in sum-of-minterms form, a decoder that generates the minterms of the function, together with an external OR gate that forms their logical sum, provides a hardware implementation of the function. In this way, any combinational circuit with n inputs and m outputs can be implemented with an n -to- 2^n -line decoder and m OR gates.

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

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



Implementation of a full adder with a decoder

Encoder

- An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and n output lines. The output lines, as an aggregate, generate the binary code corresponding to the input value.
- Let us consider an example of an encoder i.e the **octal-to-binary encoder**

It has eight inputs (one for each of the octal digits) and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time.

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

Truth Table of an Octal-to-Binary Encoder

The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. These conditions can be expressed by the following Boolean output functions:

$$z = D_1 + D_3 + D_5 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$x = D_4 + D_5 + D_6 + D_7$$

The encoder can be implemented with three OR gates.

Limitation

- The encoder defined above has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination.
- To resolve this ambiguity, encoder circuits must establish an input priority to ensure that only one input is encoded.
- Another ambiguity in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0; but this output is the same as when D is equal to 1. The discrepancy can be resolved by providing one more output to indicate whether at least one input is equal to 1.

Priority Encoder

- A priority encoder is an encoder circuit that includes the priority function.
- The truth table of a four-input priority encoder is given below.

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

In addition to the two outputs x and y , the circuit has a third output designated by V ; this is a valid bit indicator that is set to 1 when one or more inputs are equal to 1. Note that whereas X 's in output columns represent don't-care conditions, the X 's in the input columns are useful for representing a truth table in condensed form

		D_2		D_3	
		00	01	11	10
		m_0	m_1	m_3	m_2
D_0	00	X	1	1	1
	01	m_4	m_5	m_7	m_6
	11	m_{12}	m_{13}	m_{15}	m_{14}
	10	m_8	m_9	m_{11}	m_{10}

$$x = D_2 + D_3$$

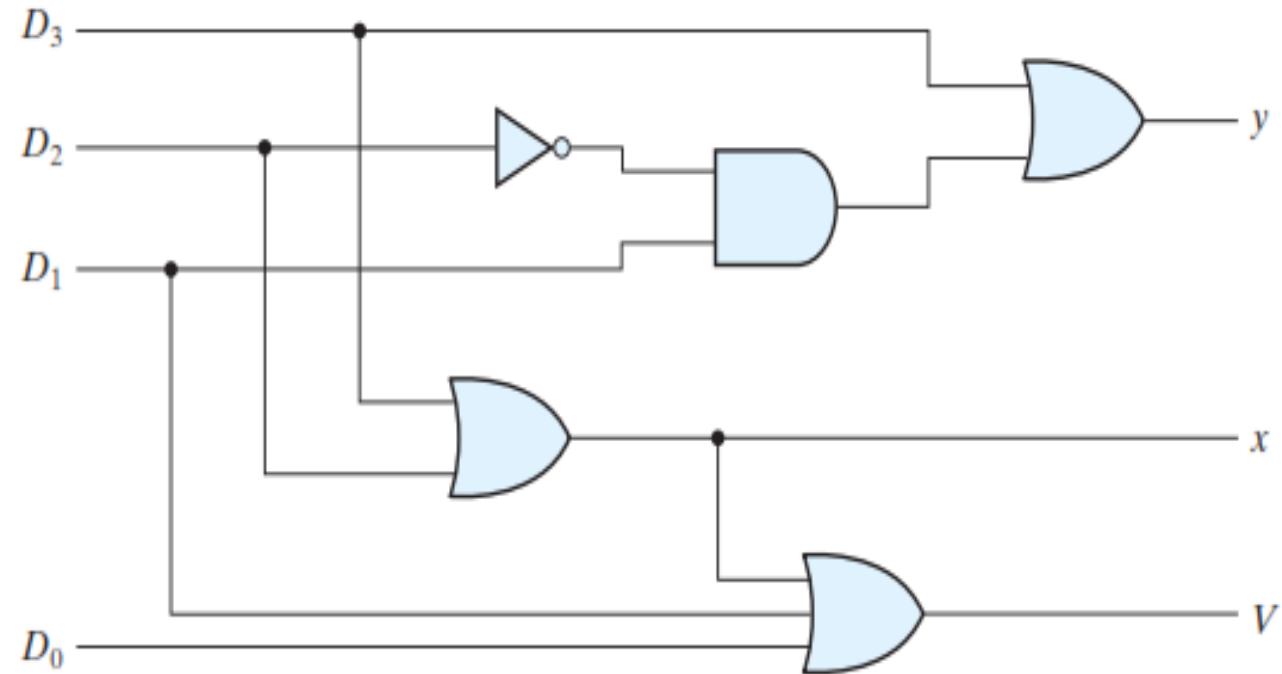
		D_2		D_3	
		00	01	11	10
		m_0	m_1	m_3	m_2
D_0	00	X	1	1	1
	01	m_4	m_5	m_7	m_6
	11	m_{12}	m_{13}	m_{15}	m_{14}
	10	m_8	m_9	m_{11}	m_{10}

$$y = D_3 + D_1 D'_2$$

$$x = D_2 + D_3$$

$$y = D_3 + D_1 D'_2$$

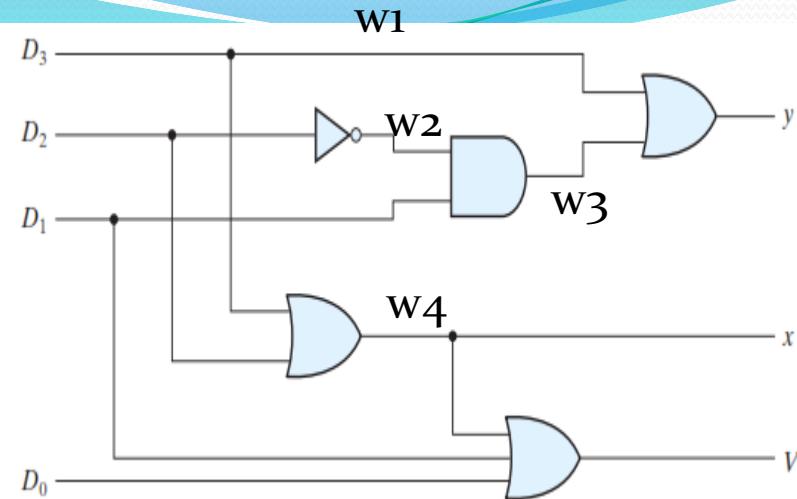
$$V = D_0 + D_1 + D_2 + D_3$$



Four-input priority encoder

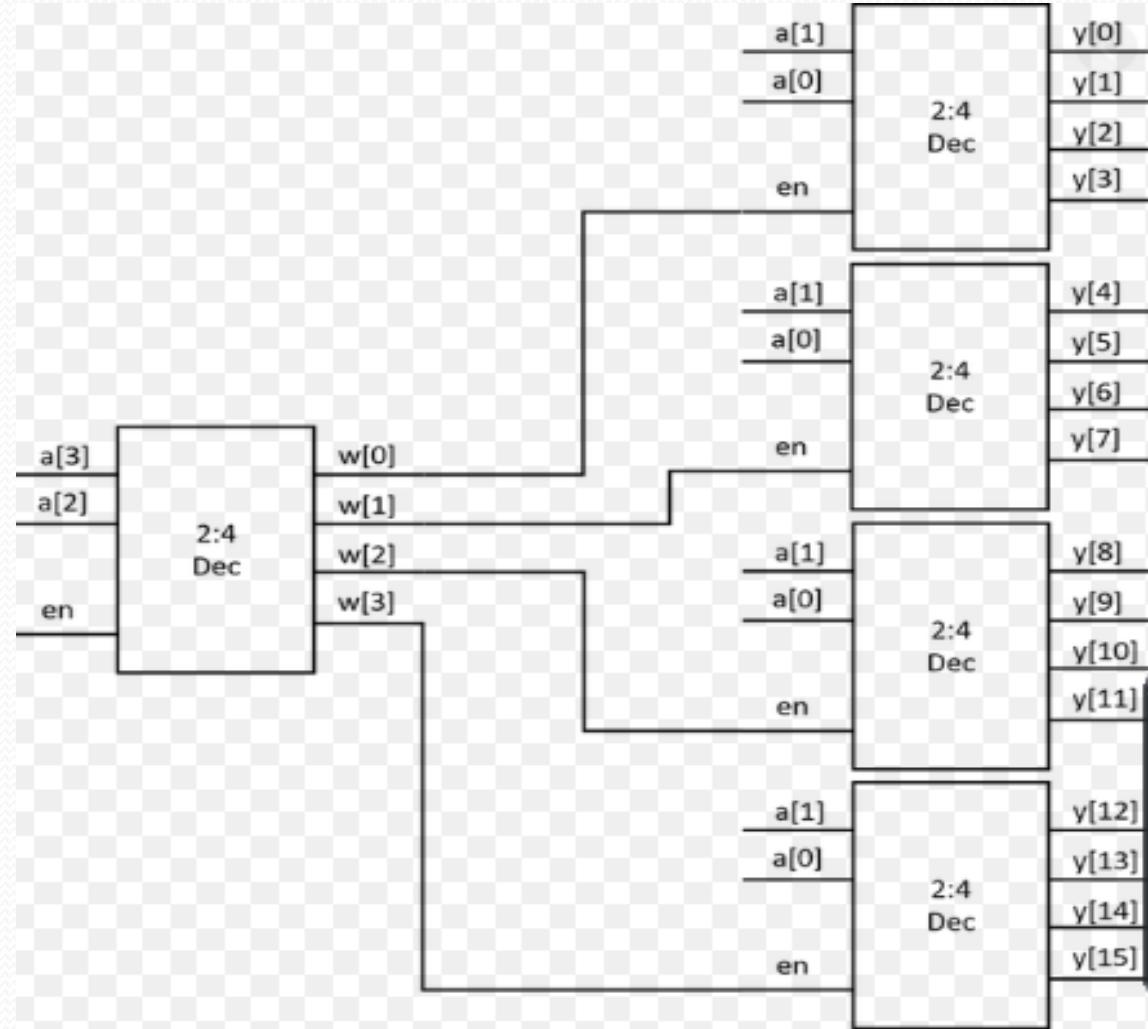
HDL for Priority Encoder

```
module PriorityEncoder_4Bit(  
    input [0:3] D,  
    output [1:0] Y,  
    output V  
);  
    reg [1:0] Y;  
    reg V;  
    always @(D)  
    begin  
        Y[1] <= D[2] | D[3];  
        Y[0] <= D[3] | D[1] & ~D[2];  
        V = D[0] | D[1] | D[2] | D[3];  
    end  
endmodule
```

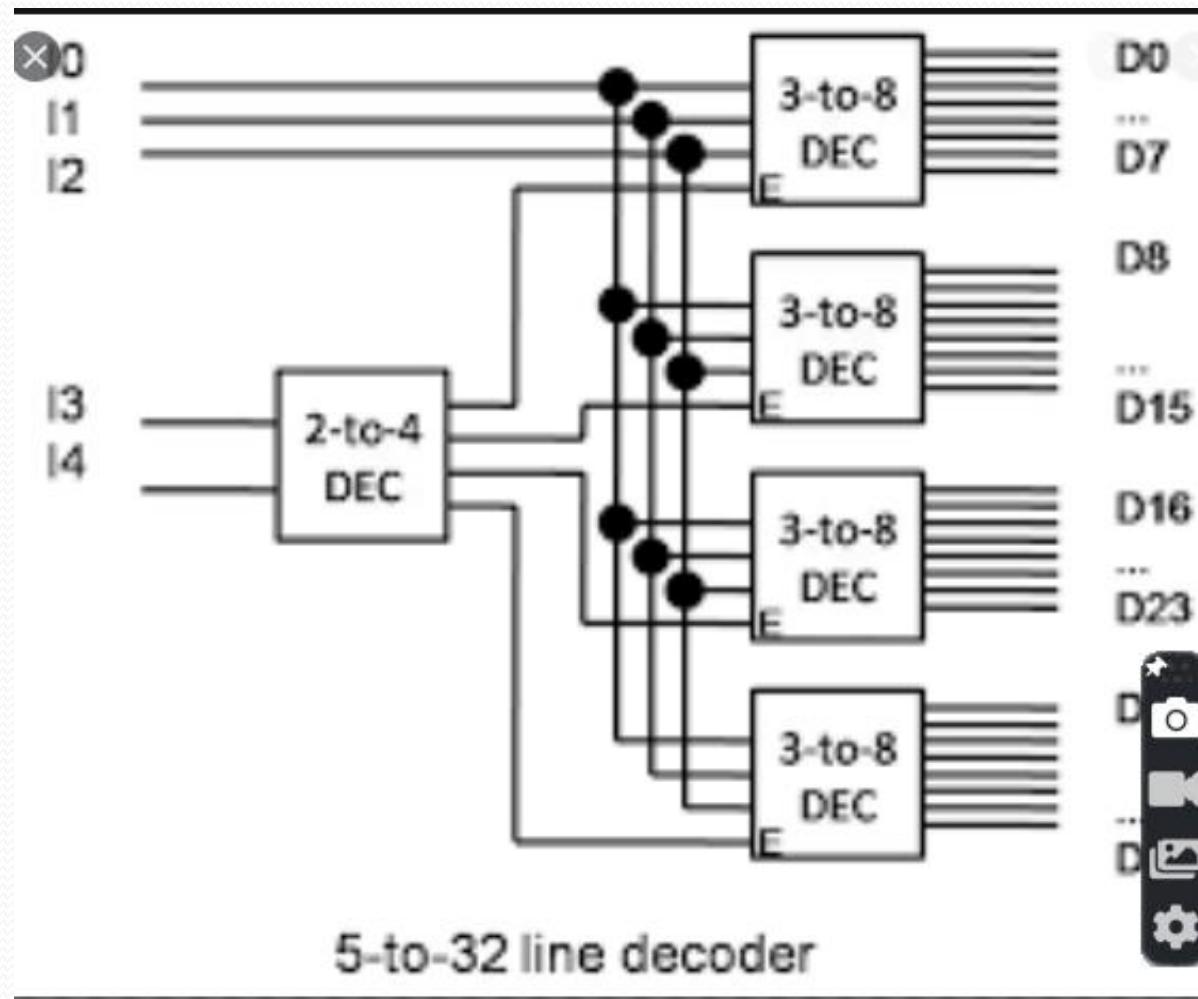


Example

Construct a 4-to-16-line decoder with five 2-to-4-line decoders with enable.



Construct a 5-to-32-line decoder with four 3-to-8-line decoders with enable and a 2-to4-line decoder. Use block diagrams for the components.





THANK YOU

LOGIC DESIGN

EET-1021

CHAPTER 04

Lecture 23

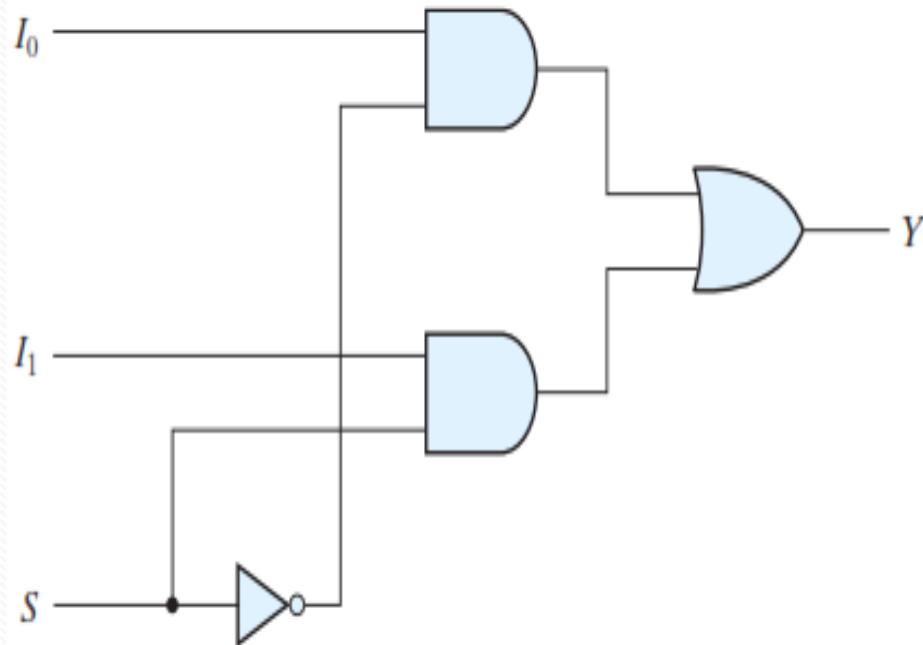
Combinational Logic

Overview of previous lecture

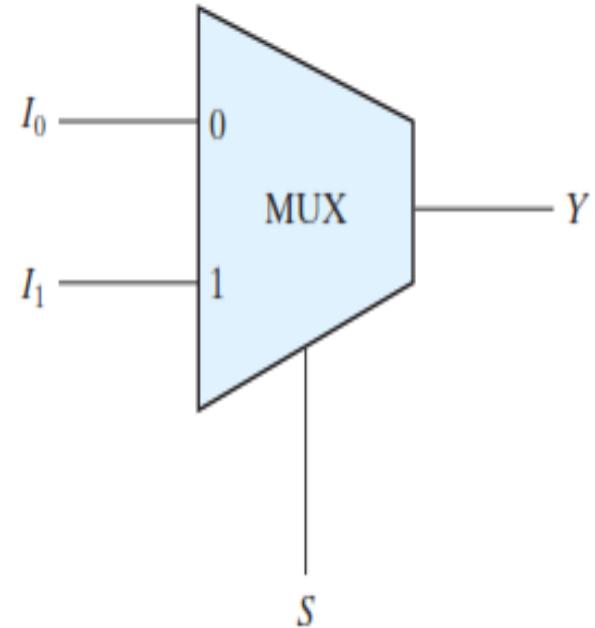
- **What is Decoder**
- **What is Encoder**
- **Limitation of Encoder**
- **Priority Encoder**

Multiplexers

- A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2^n input lines and n *selection* lines whose bit combinations determine which input is selected.
- In general, a 2^n -to-1-line multiplexer is constructed from an n -to- 2^n decoder by adding 2^n input lines to it, one to each AND gate. The outputs of the AND gates are applied to a single OR gate.

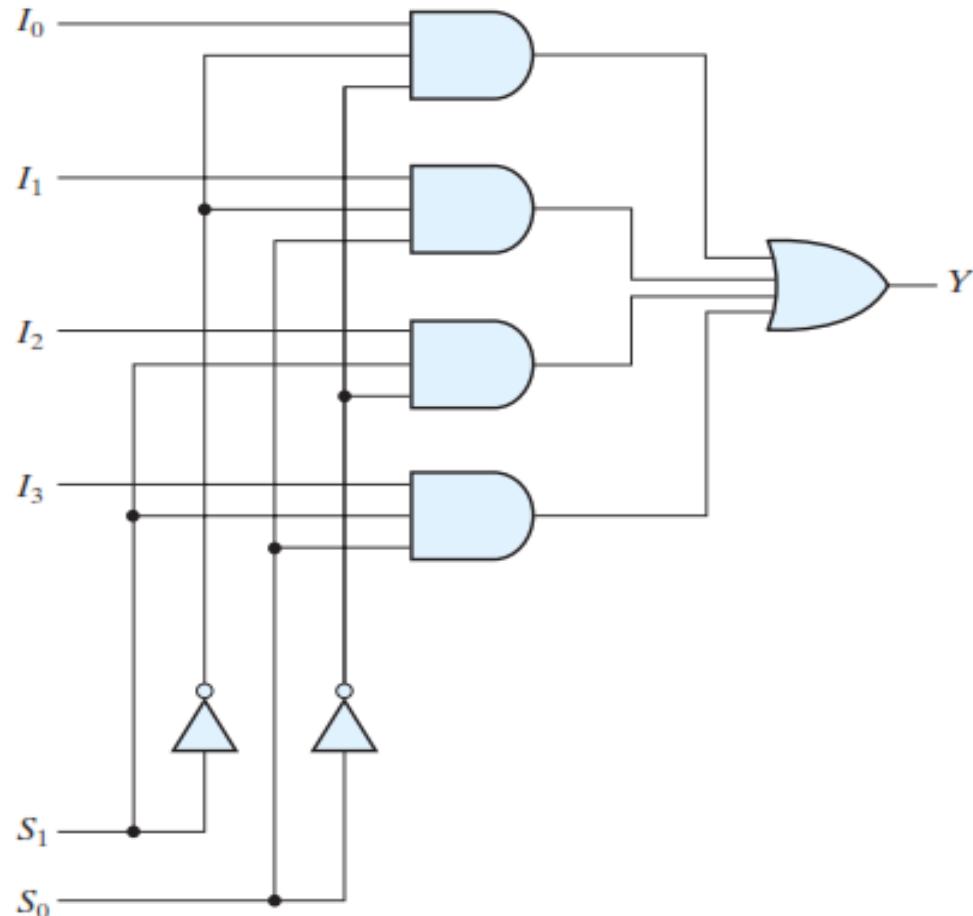


(a) Logic diagram



(b) Block diagram

Two-to-one-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

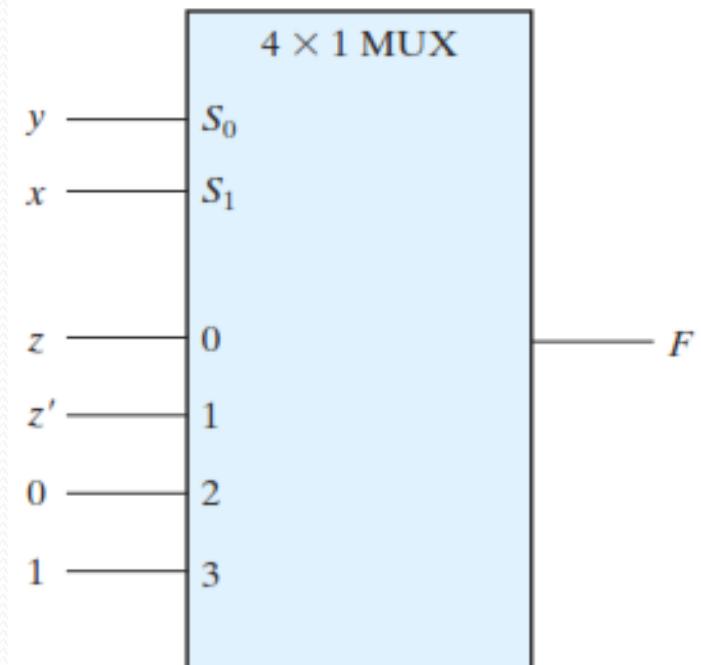
Four-to-one-line multiplexer

Boolean Function Implementation

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

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

(a) Truth table

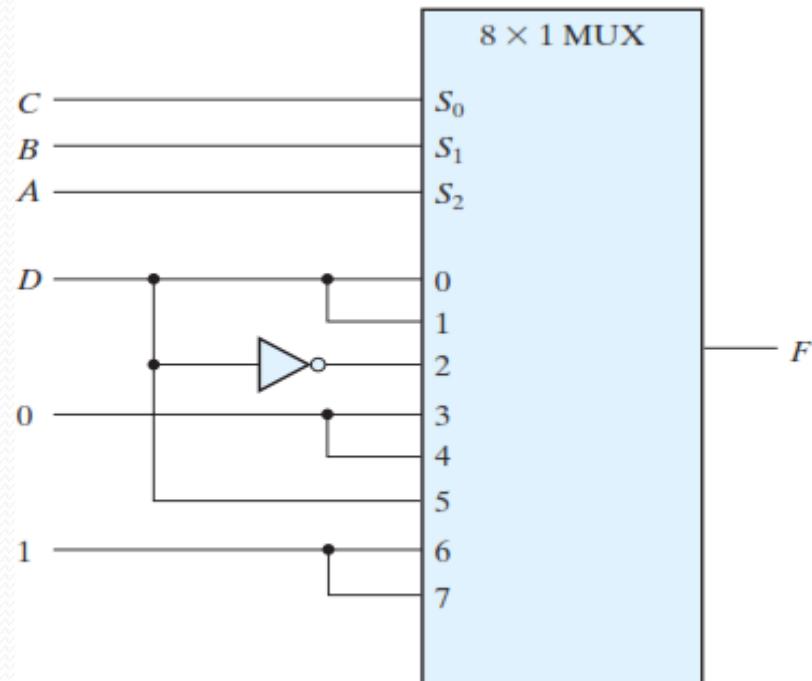


(b) Multiplexer implementation

As a second example, consider the implementation of the Boolean function

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

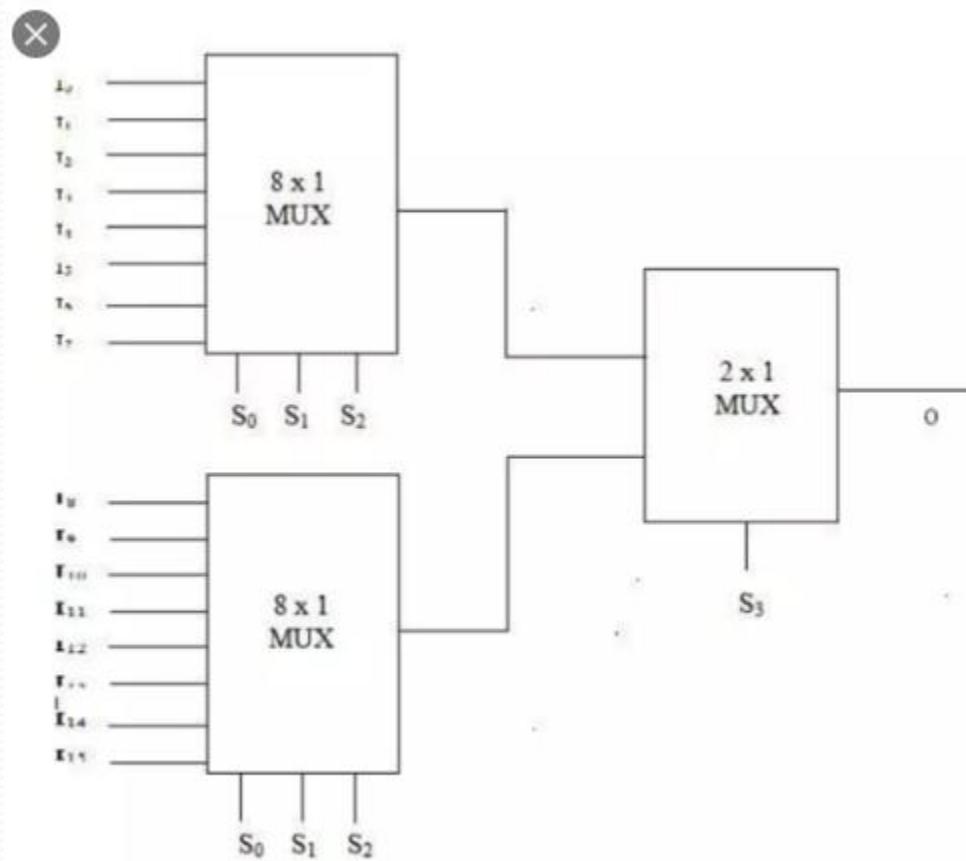
A	B	C	D	F
0	0	0	0	0 $F = D$
0	0	0	1	1
0	0	1	0	0 $F = D$
0	0	1	1	1
0	1	0	0	1 $F = D'$
0	1	0	1	0
0	1	1	0	0 $F = 0$
0	1	1	1	0
1	0	0	0	0 $F = 0$
1	0	0	1	0
1	0	1	0	0 $F = D$
1	0	1	1	1
1	1	0	0	1 $F = 1$
1	1	0	1	1
1	1	1	0	1 $F = 1$
1	1	1	1	1



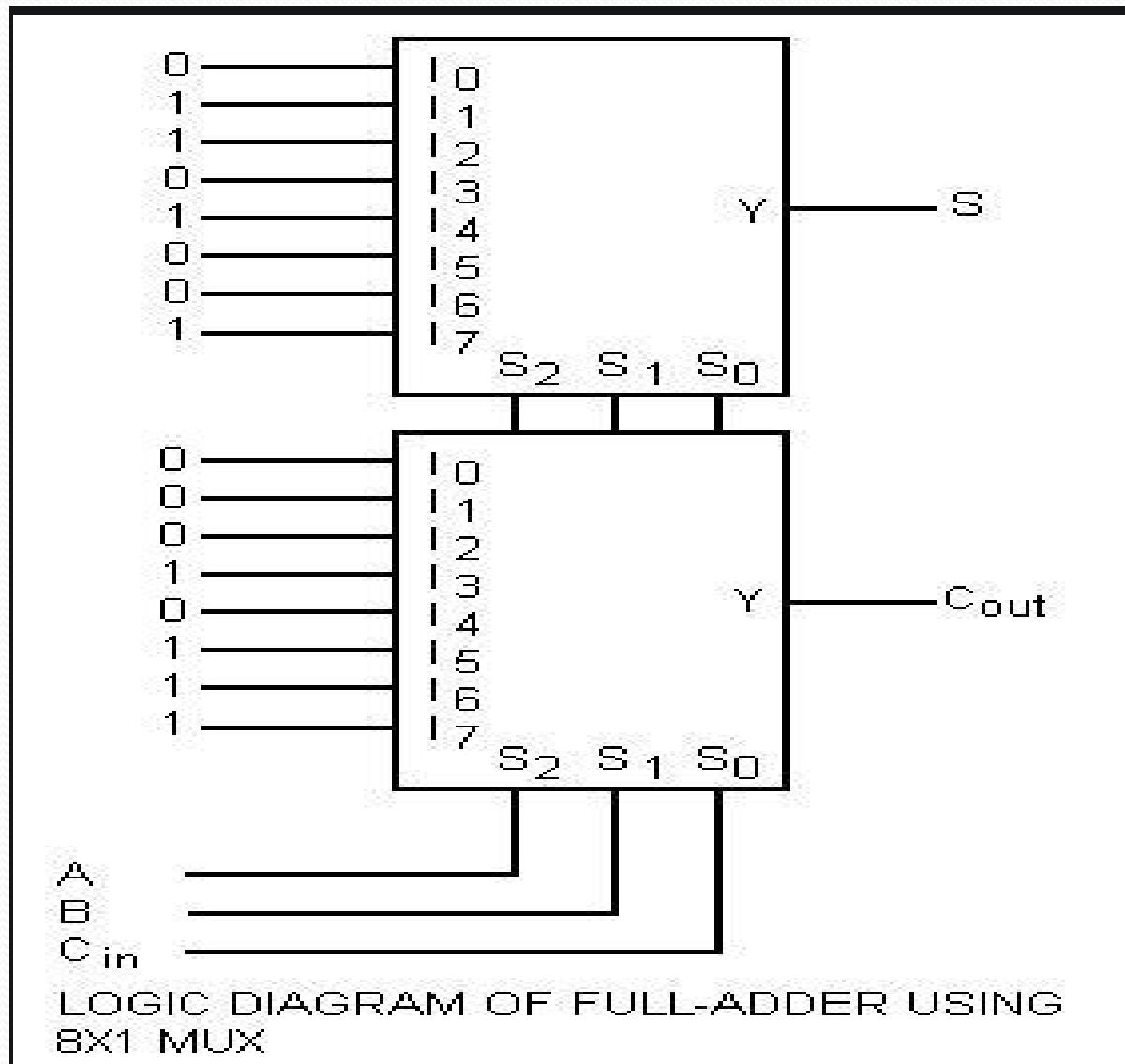
Implementing a four-input function with a multiplexer

Example

Construct a 16×1 multiplexer with two 8×1 and one 2×1 multiplexers.



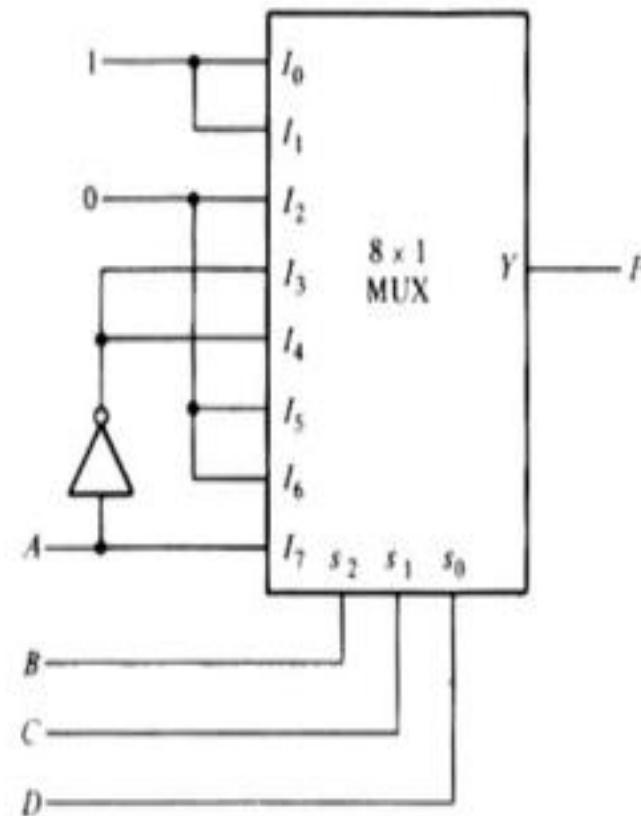
Implement a full adder with two 8×1 multiplexers.



Implement the given Boolean function by using 8×1 multiplexer.

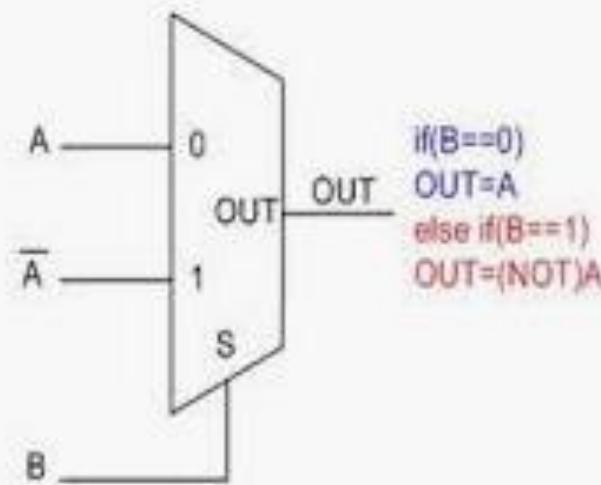
$$F(A,B,C,D) = \sum(0,1,3,4,8,9,15)$$

	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
A'	0	1	2	3	4	5	6	7
A	8	9	10	11	12	13	14	15
	1	1	0	A'	A'	0	0	A



Implement XOR gate by using 2×1 Multiplier

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





THANK YOU

LOGIC DESIGN

EET-1021

CHAPTER 04

Lecture 24

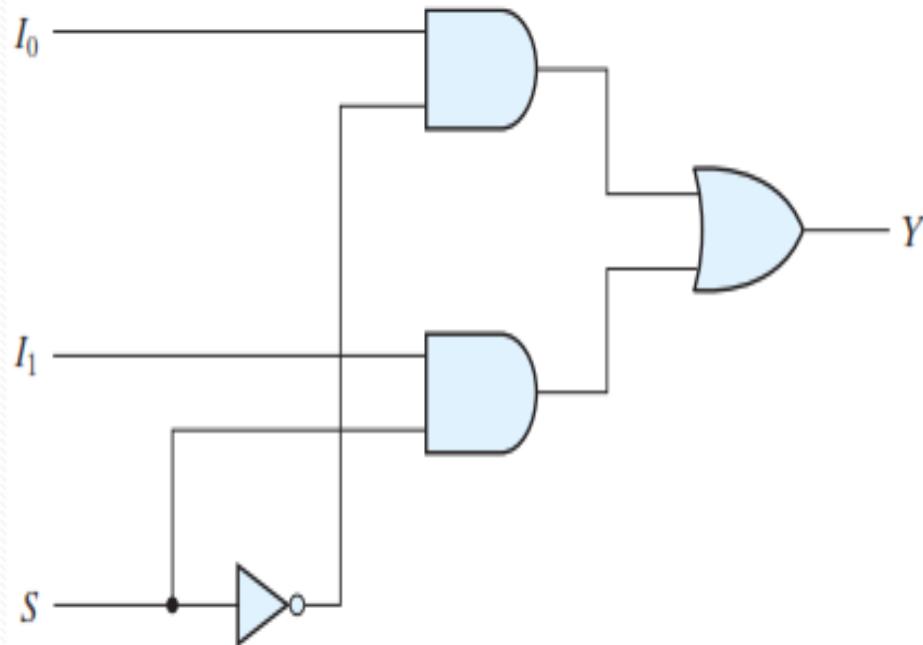
Combinational Logic

Overview of previous lecture

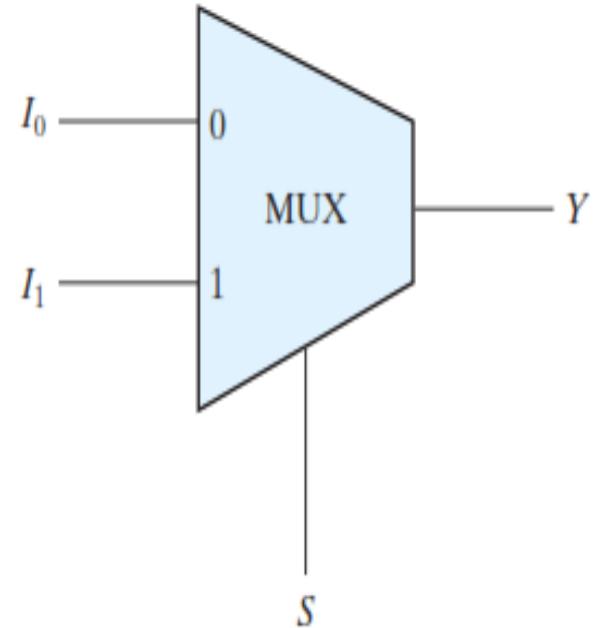
- **What is a Multiplexer**

Multiplexers

- A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2^n input lines and n *selection* lines whose bit combinations determine which input is selected.
- In general, a 2^n -to-1-line multiplexer is constructed from an n -to- 2^n decoder by adding 2^n input lines to it, one to each AND gate. The outputs of the AND gates are applied to a single OR gate.



(a) Logic diagram

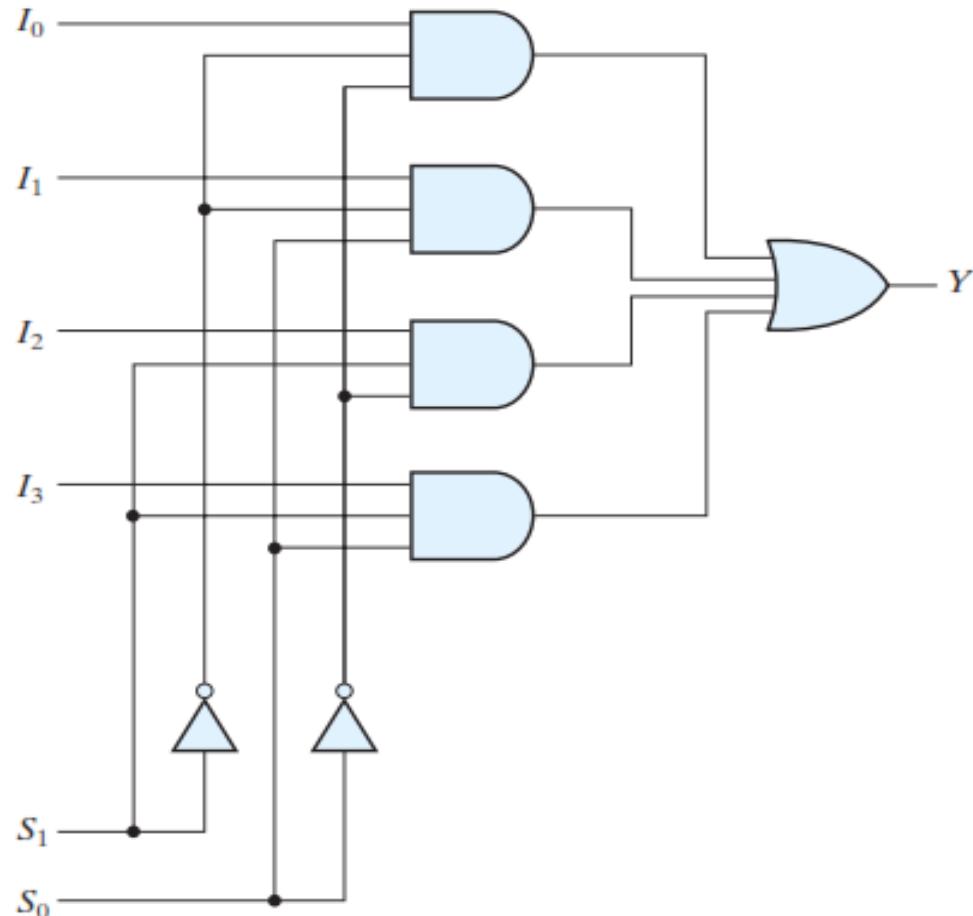


(b) Block diagram

Two-to-one-line multiplexer

HDL for Two-to-One Multiplexer

```
// Dataflow description of two-to-one-line multiplexer
module mux_2x1_df(Y, Io, I1, s);
    output Y;
    input Io, I1;
    input s;
    assign Y = (s)? Io : I1;
endmodule
```



(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

Four-to-one-line multiplexer

- **HDL for Four-to-One Multiplexer**

// Behavioral description of four-to-one-line multiplexer

```
module mux_4x1_beh (Y, I0, I1, I2, I3, s);
```

```
output Y;
```

```
input I0, I1, I2, I3;
```

```
input [1:0]s;
```

```
reg Y;
```

```
always @ (I0 or I1 or I2 or I3 or s)
```

```
begin
```

```
Case (s)
```

```
2' boo: Y<= I0;
```

```
2' bo1: Y<= I1;
```

```
2' b10: Y<= I2;
```

```
2' b11: Y<= I3;
```

```
endcase
```

```
end
```

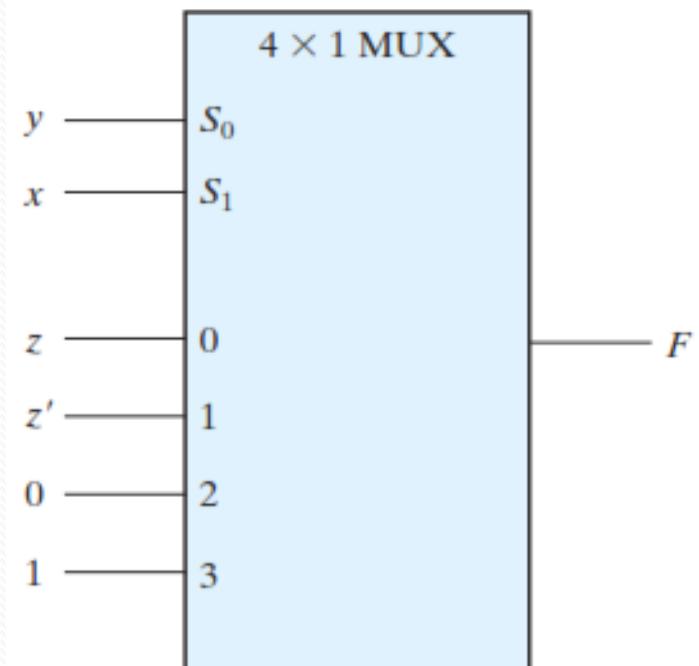
```
endmodule
```

Boolean Function Implementation

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

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

(a) Truth table



(b) Multiplexer implementation

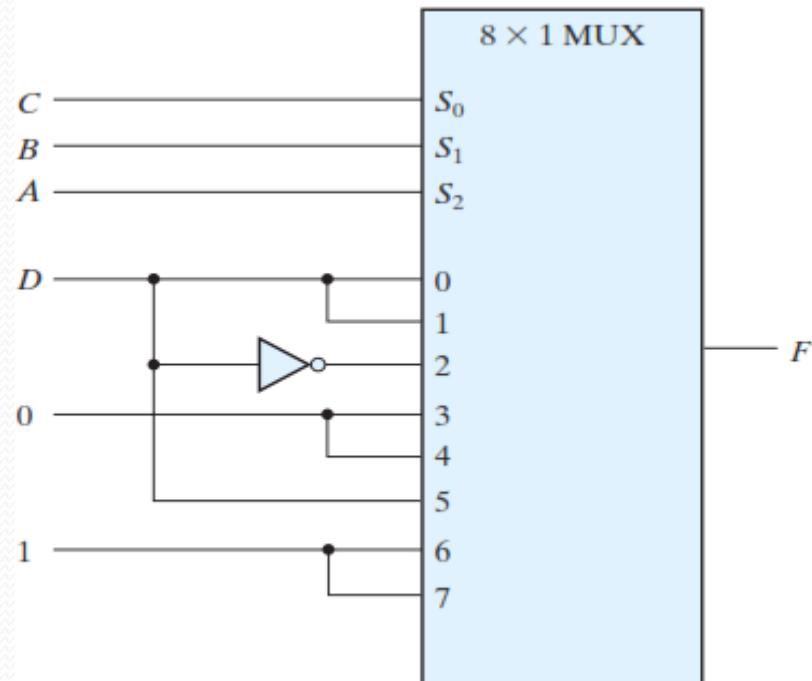
HDL for implementing a Boolean Function using Multiplexer :

```
module booleanfunction (F, z, s);
input z;
input [1:0]s;
output F;
wire z';
not (z', z);
mux_4x1_beh (F, z, z', o, 1, s);
endmodule
```

As a second example, consider the implementation of the Boolean function

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

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



Implementing a four-input function with a multiplexer



THANK YOU

LOGIC DESIGN

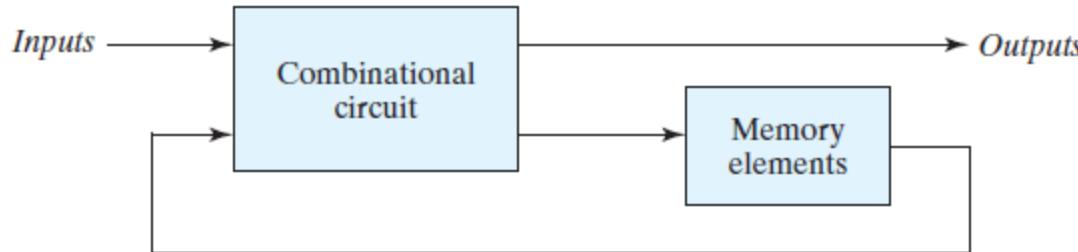
EET-1021

CHAPTER 05

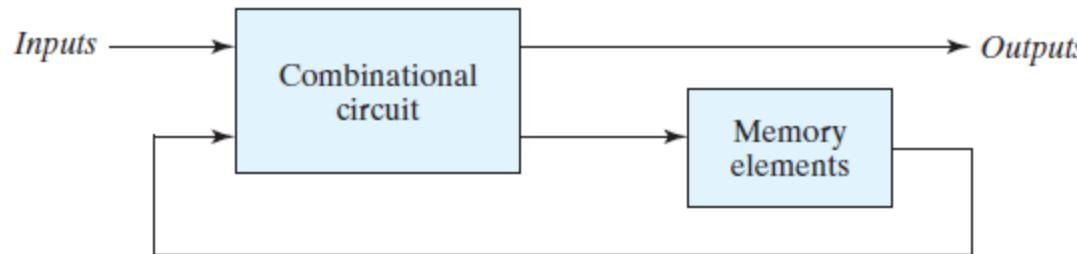
Lecture 25

SEQUENTIAL LOGIC CIRCUITS

Block diagram of sequential circuit



- It consists of a combinational circuit to which storage elements are connected to form a feedback path.
- The storage elements are devices capable of storing binary information.
- The binary information stored in these elements at any given time defines the *state of the sequential circuit* at that time.
- The sequential circuit receives binary information from external inputs that, together with the present state of the storage elements, determine the binary value of the outputs.



- These external inputs also determine the condition for changing the state in the storage elements.
- The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the **inputs**, but also of the **present state of the storage elements**.
- The next state of the storage elements is also a function of external inputs and the present state.
- Thus, **a sequential circuit is specified by a time sequence of inputs, outputs, and internal states** .
- In contrast, **the outputs of combinational logic depend only** on the present values of the inputs.

Classification of sequential circuit

- There are **two** main types of sequential circuits, and their classification is a function of the timing of their signals.
- A *synchronous sequential circuit* is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time.
- The behavior of an *asynchronous sequential circuit* depends upon the input signals at any instant of time and the order in which the inputs change.

Asynchronous sequential circuit

- The **storage elements** commonly used in asynchronous sequential circuits are **time-delay devices**.
- The storage capability of a time-delay device varies with the time it takes for the signal to propagate through the device. In practice, the internal **propagation delay** of logic gates is of sufficient duration to produce the needed delay, so that actual delay units may not be necessary.
- In **gate-type asynchronous systems**, the storage elements consist of **logic gates** whose propagation delay provides the required storage.
- Thus, an asynchronous sequential circuit may be regarded as a combinational circuit with feedback. Because of the feedback among logic gates, an asynchronous sequential circuit may become unstable at times.

Synchronous sequential circuit

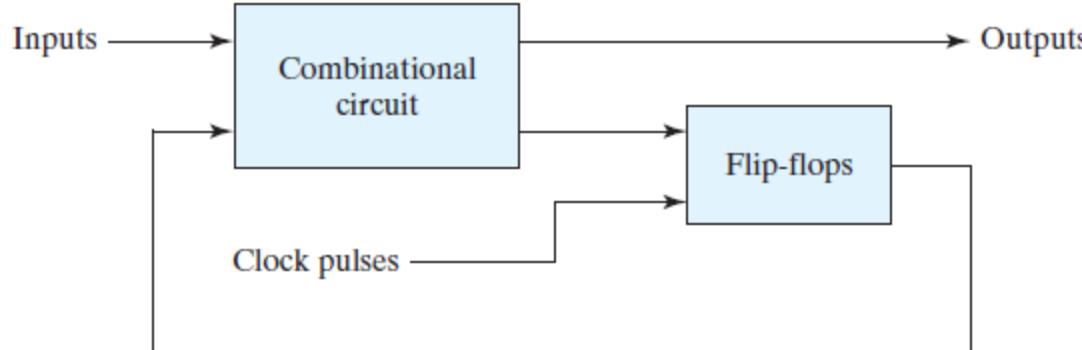
- A **synchronous sequential** circuit employs signals that affect the storage elements at only discrete instants of time.
- **Synchronization** is achieved by a timing device called a **clock generator**, which provides a *clock signal having the form of a periodic train of clock pulses*.
- *The clock signal is commonly denoted by the identifiers **clock** and **clk** .*
- *The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of each pulse.*
- In practice, the **clock pulses** determine *when computational activity will occur* within the circuit, and other signals (external inputs and otherwise) determine *what changes will take place affecting the storage elements and the outputs*.



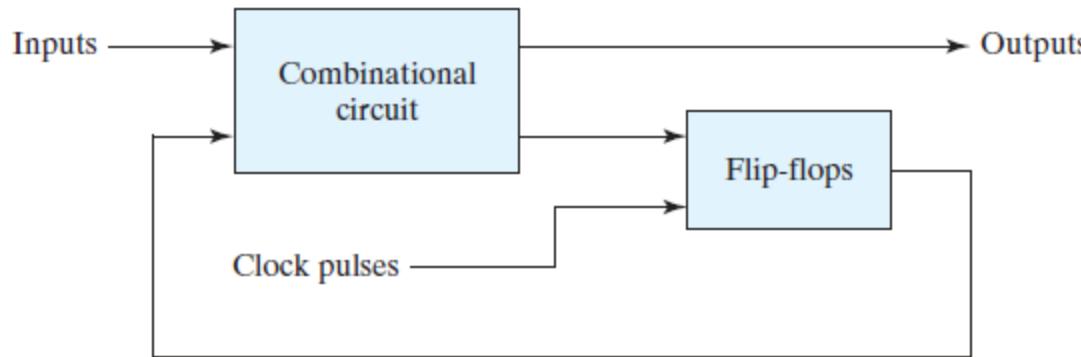
Timing diagram of clock pulses

- For example, a circuit that is to add and store two binary numbers would compute their sum from the values of the numbers and store the sum at the **occurrence of a clock pulse**.
- Synchronous sequential circuits that use clock pulses to control storage elements are called *clocked sequential circuits* and are the type **most frequently** encountered in practice.
- They are called *synchronous circuits because the activity within the circuit and the resulting updating of stored values is synchronized to the occurrence of clock pulses.*
- The design of synchronous circuits is feasible because they seldom manifest instability problems and their timing is easily broken down into independent discrete steps, each of which can be considered separately.

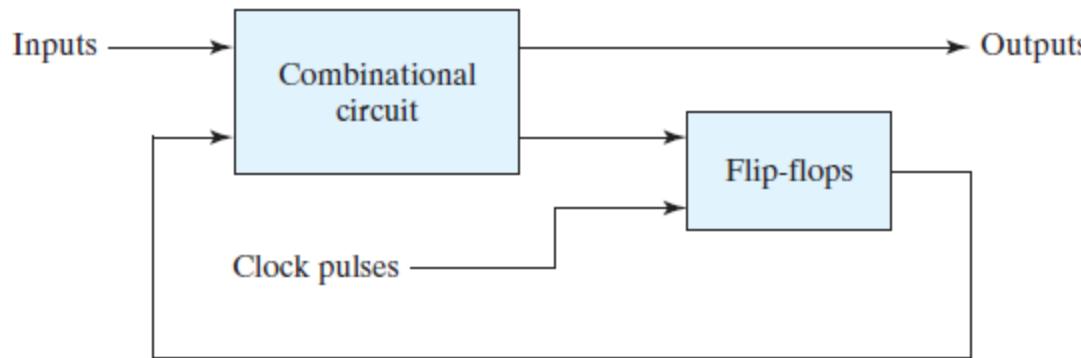
Block diagram of Synchronous clocked sequential circuit



- The storage elements (memory) used in clocked sequential circuits are called **flipflops**.
- A flip-flop is a binary storage device capable of storing **one bit** of information.
- In a stable state, the output of a flip-flop is either 0 or 1. A sequential circuit may use many flip-flops to store as many bits as necessary.
- The **outputs** are formed by a *combinational logic* function of the inputs to the circuit or the values stored in the flip-flops (or both).



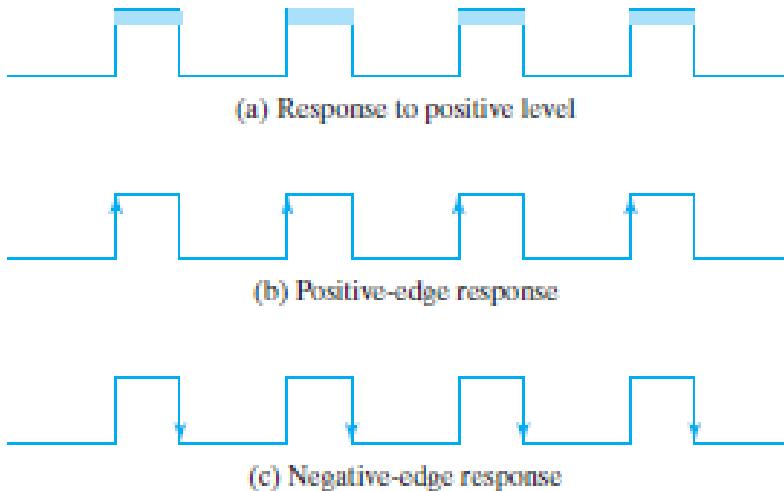
- The value that is stored in a flip-flop when the clock pulse occurs is also determined by the **inputs to the circuit** or the **values presently stored** in the flip-flop (or both).
- The new value is stored (i.e., the flip-flop is updated) when a **pulse** of the clock signal occurs.
- Prior to the occurrence of the clock pulse, the combinational logic forming the next value of the flip-flop must have reached a stable value. Consequently, the **speed** at which the combinational logic circuits operate is critical.
- If the clock (synchronizing) pulses arrive at a **regular interval**, the combinational logic must respond to a change in the state of the flip-flop in time to be updated before the next pulse arrives.
- Propagation delays** play an important role in determining the minimum interval between clock pulses that will allow the circuit to operate correctly.



- A change in state of the flip-flops is initiated only by a **clock pulse transition**—for example, when the value of the clock signals changes from 0 to 1.
- When a clock pulse is **not active**, the feedback loop between the value stored in the flip-flop and the value formed at the input to the flip-flop is effectively broken because the flipflop outputs cannot change even if the outputs of the combinational circuit driving their inputs change in value.
- Thus, the transition from one state to the next occurs only at predetermined intervals dictated by the **clock pulses**.

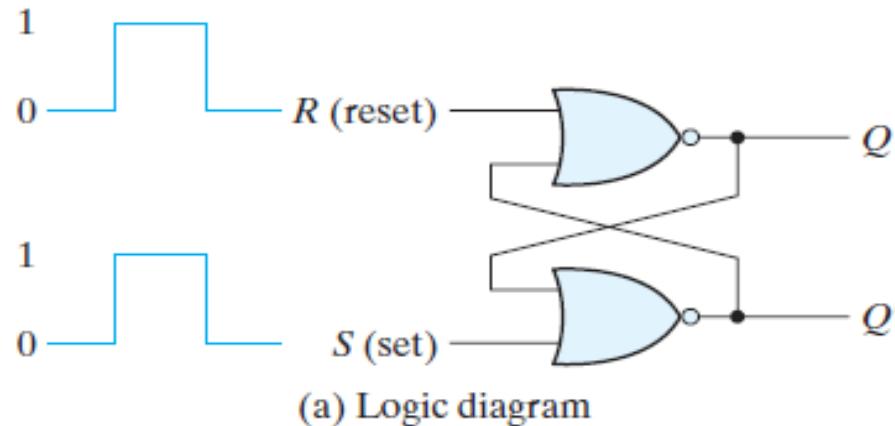
STORAGE ELEMENTS: LATCHES

- A storage element in a digital circuit can maintain a **binary state indefinitely** (as long as power is delivered to the circuit), until directed by an input signal to switch states.
- The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state.
- *Storage elements that operate with signal levels (rather than signal transitions) are referred to as **latches** ; those controlled by a clock transition are **flip-flops** .*
- *Latches are said to be level sensitive devices; flip-flops are edge-sensitive devices.*



- The latch responds to a change in the **level** of a clock pulse.
- The key to the proper operation of a flip-flop is to trigger it only during a **signal transition**.
- *This can be accomplished* by eliminating the feedback path that is inherent in the operation of the sequential circuit using latches.
- A clock pulse goes through two transitions: from 0 to 1 and the return from 1 to 0.
- As shown in Figure , the **positive transition** is defined as the **positive edge** and the **negative transition** as the **negative edge**.

SR Latch with NOR gates



S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	(forbidden)

(b) Function table

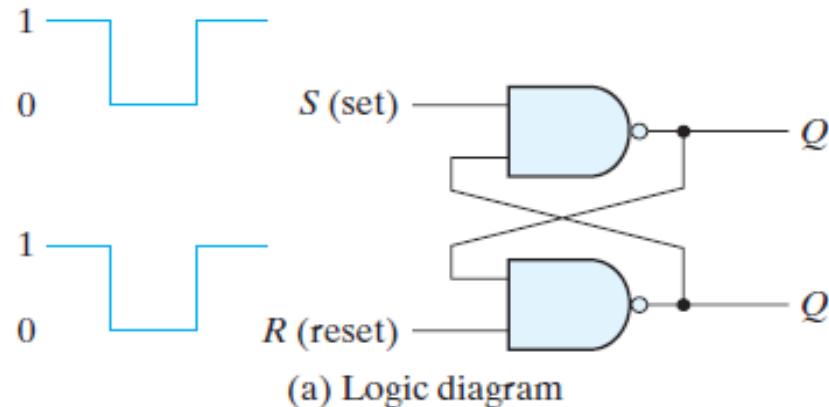
The *SR latch* is a circuit with two cross-coupled NOR gates or two cross-coupled NAND

gates, and two inputs labeled S for set and R for reset.

The latch has **two useful states**.

When output $Q = 1$ and $Q' = 0$, the latch is said to be in the **set state**. When $Q = 0$ and $Q' = 1$, it is in the **reset state**.

SR latch with NAND gates



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

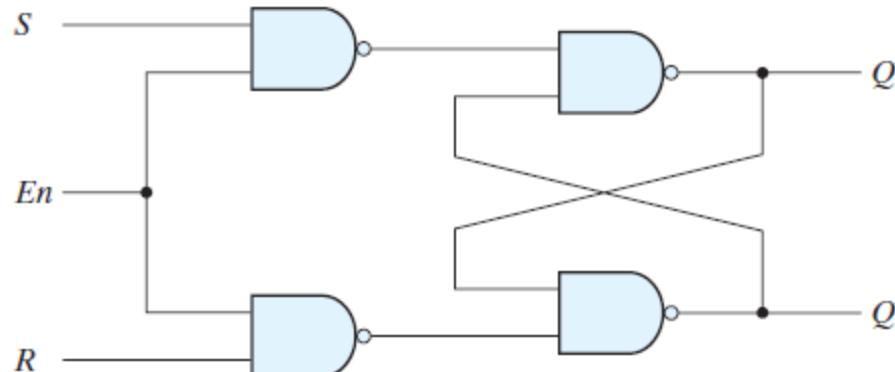
(b) Function table

Annotations for the function table:

- Row 2: (after $S = 1, R = 0$)
- Row 4: (after $S = 0, R = 1$)
- Row 5: (forbidden)

- It operates with both inputs normally at 1, unless the state of the latch has to be changed.
- The application of 0 to the *S input* causes output *Q* to go to 1, putting the latch in the **set state**. When the *S* input goes back to 1, the circuit remains in the set state.
- After both inputs go back to 1, we are allowed to change the state of the latch by placing a 0 in the *R input*. This action causes the circuit to go to the **reset state** and stay there even after both inputs return to 1.
- The condition that is forbidden for the NAND latch is **both inputs being equal to 0** at the same time, an input combination that should be avoided.

SR latch with control input

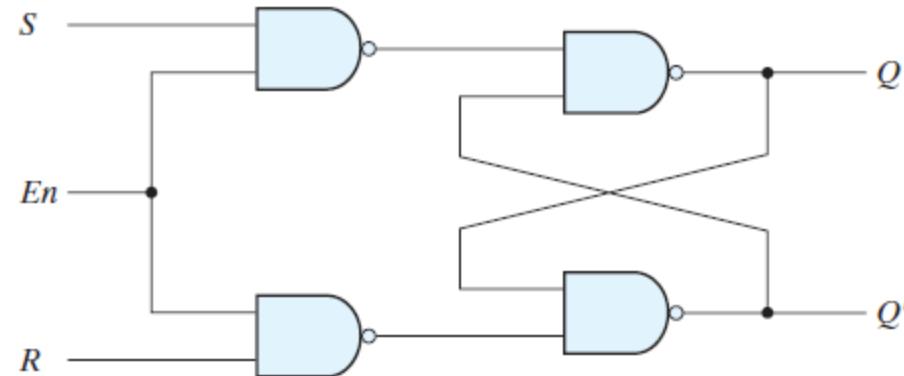


(a) Logic diagram

En	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

(b) Function table

- It consists of the basic SR latch and two additional NAND gates. **The control input En acts as an enable signal for the other two inputs.**
- The outputs of the NAND gates stay at the logic-1 level as long as the enable signal remains at 0. This is the quiescent condition for the SR latch.**
- When the enable input goes to 1, information from the S or R input is allowed to affect the latch. **The set state is reached with $S = 1$, $R = 0$, and $En = 1$ (active-high enabled).**



(a) Logic diagram

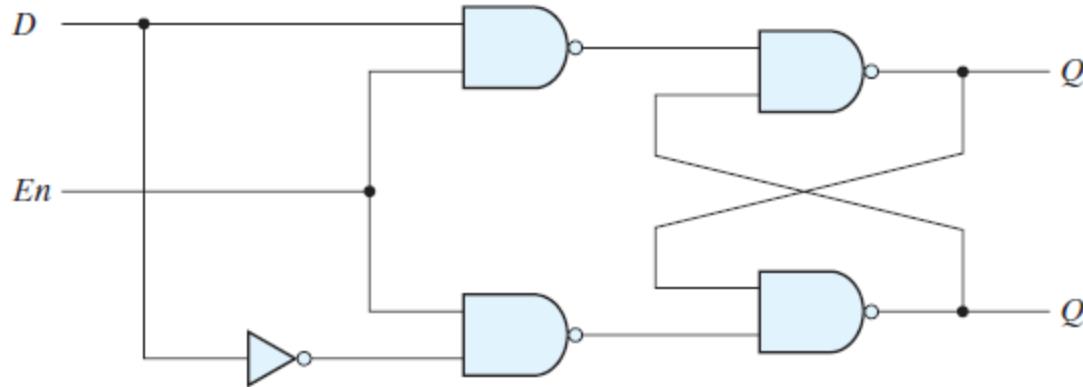
En	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

(b) Function table

- To change to the **reset state**, the inputs must be $S = 0$, $R = 1$, and $En = 1$. In either case, when En returns to 0, the circuit remains in its current state.
- The control input disables the circuit by applying 0 to En , so that the state of the output does not change regardless of the values of S and R .
- Moreover, when $En = 1$ and both the S and R inputs are equal to 0, the state of the circuit does not change.
- An **indeterminate condition occurs when all three inputs are equal to 1**.

```
module set-reset (Q, Q', S, R, EN);
    input S, R, EN;
    output Q, Q';
    wire a, b;
    nand n1(a, S, EN),
        n2(b, R, EN),
        n3(Q, a, Q'),
        n4(Q', b, Q);
endmodule
```

D Latch (Transparent Latch)



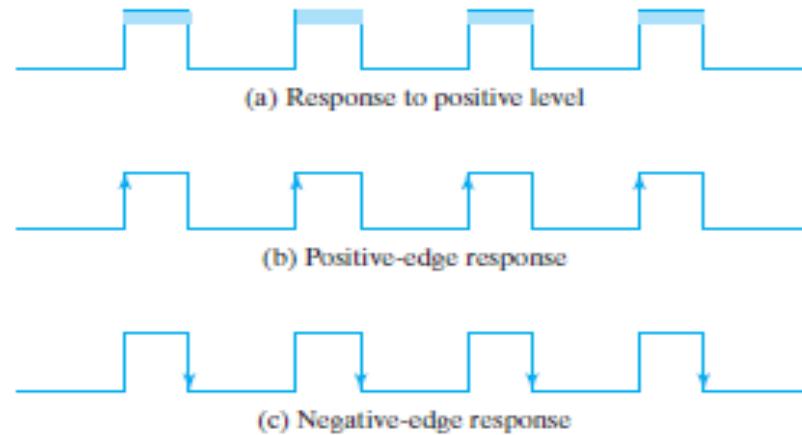
(a) Logic diagram

En	D	Next state of Q
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

(b) Function table

- One way to eliminate the undesirable condition of the indeterminate state in the SR latch is to ensure that inputs **S and R are never equal to 1 at the same time**. This is done in the D latch, shown in Figure .
- This latch has only two inputs: D (data) and En (enable). **The D input goes directly to the S input, and its complement is applied to the R input.**
- As long as the enable input is at 0, the cross-coupled SR latch has both inputs at the 1 level and the circuit cannot change state regardless of the value of D .
- The D input is sampled when En = 1. If D = 1, the Q output goes to 1, placing the circuit in **the set state**.
- If D = 0, output Q goes to 0, placing the circuit in the **reset state**.

Clock response in latch and flip-flop



- The latch responds to a change in the *level* of a clock pulse.
- The key to the proper operation of a flip-flop is to trigger it only during a *signal transition*.
- *This can be accomplished by eliminating the feedback path that is inherent in the operation of the sequential circuit using latches.*
- A clock pulse goes through two transitions: from 0 to 1 and the return from 1 to 0.
- As shown in Figure , the **positive transition** is defined as the **positive edge** and the **negative transition** as the **negative edge**.

```
// Description of D latch
module D_latch (Q, D, enable);
output Q;
input D, enable;
reg Q;
always @ (enable or D)
  if (enable) Q <= D; // Same as: if (enable == 1)
endmodule

// Alternative syntax
module D_latch (output reg Q, input enable, D);
always @ (enable, D)
  if (enable) Q <= D; // No action if enable not asserted
endmodule
```



THANK YOU

DIGITAL LOGIC

CHAPTER 5

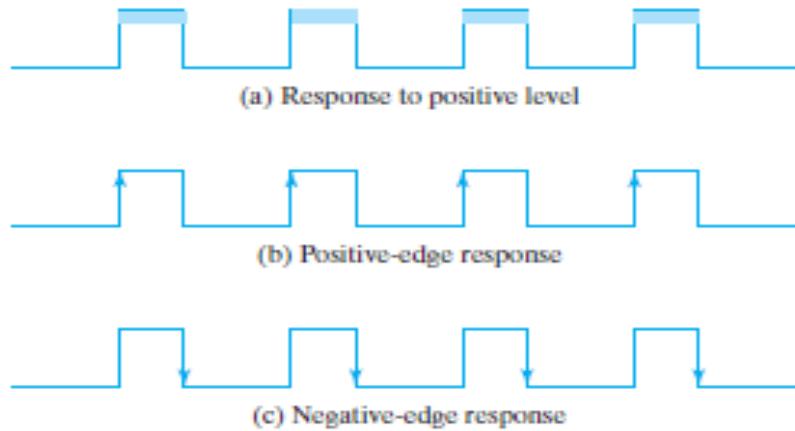
Lecture 26

SEQUENTIAL LOGIC CIRCUITS

STORAGE ELEMENTS: FLIP-FLOPS

- Latches and flip-flops are the basic elements for storing information.
- One latch or flip-flop can store one bit of information.
- The main difference between latches and flip-flops is that for latches, their outputs are constantly affected by their inputs as long as the enable signal is asserted.
- In other words, when they are enabled, their content changes immediately when their inputs change.
- Flip-flops, on the other hand, have their content change only either at the rising or falling edge of the enable signal.
- This enable signal is usually the controlling clock signal.
- There are basically four main types of latches and flip-flops: **SR**, **D**, **JK**, and **T**.
- The major differences in these flip-flop types are the number of inputs they have and how they change state.

Clock response in latch and flip-flop

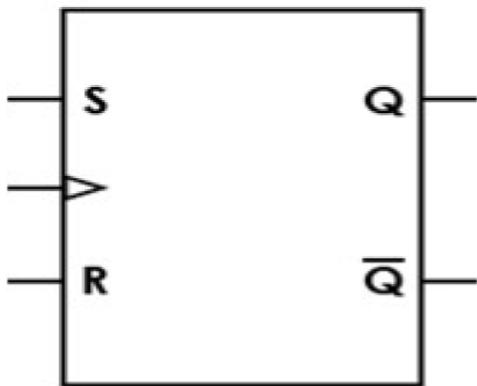


- The latch responds to a change in the *level of a clock pulse*.
- The key to the proper operation of a flip-flop is to trigger it only during a *signal transition*.
- *This can be accomplished by eliminating the feedback path that is inherent in the operation of the sequential circuit using latches.*
- A clock pulse goes through two transitions: from 0 to 1 and the return from 1 to 0.
- As shown in Figure , the **positive transition** is defined as the **positive edge** and the **negative transition** as the **negative edge**.

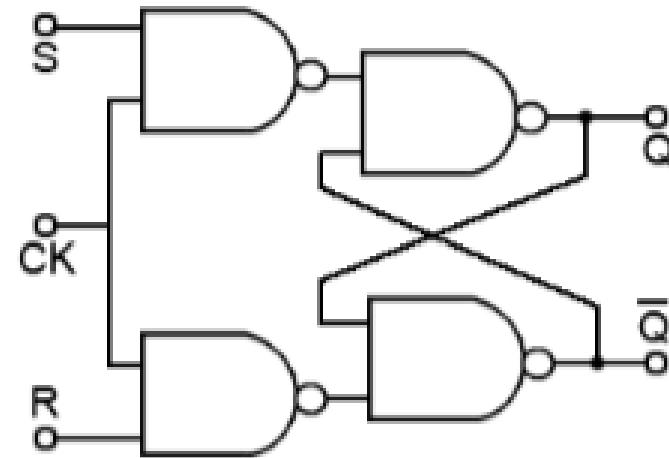
SR FLIP FLOP

- The **SR flip-flop** can be considered as one of the most basic sequential logic circuit possible.
- This simple flip-flop is basically a one-bit memory bi-stable device that has two inputs, one which will “SET” the device (meaning the output = “1”), and is labelled **S** and one which will “RESET” the device (meaning the output = “0”), labelled **R**.
- Then the SR description stands for “Set-Reset”.
- The reset input resets the flip-flop back to its original state with an output Q that will be either at a logic level “1” or logic “0” depending upon this set/reset condition.

SR Flip Flop



FLIP-FLOP SYMBOL



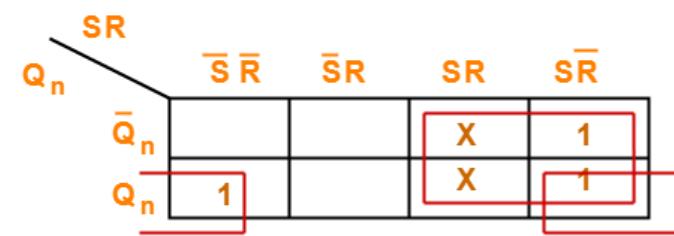
LOGIC DIAGRAM

CLK	S	R	Q
0	X	X	Unchanged
1	0	0	Unchanged
1	0	0	Reset(0)
1	1	0	Set(1)
1	1	1	Indetermined

FUNCTIONAL TABLE

SR Flip Flop

Q_n	S	R	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	X
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	X



CHARACTERISTIC TABLE

$$Q_{n+1} = S + Q_n R'$$

CHARACTERISTIC EQUATION

SR Flip Flop

$$Q_{n+1} = S + Q_n R'$$

CHARACTERISTIC EQUATION

Q_n	Q_{n+1}	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

EXCITATION TABLE

HDL for SR Flip Flop:

```
module sr_ff (input s, input r, input clk, output reg q, output q_not);
assign q_not = !q;
always @ (posedge clk)
begin
if(clk==1)
q <= s||(!r&&q);
end;
endmodule;
```

THANK YOU

DIGITAL LOGIC

CHAPTER 5

Lecture 27

SEQUENTIAL LOGIC CIRCUITS

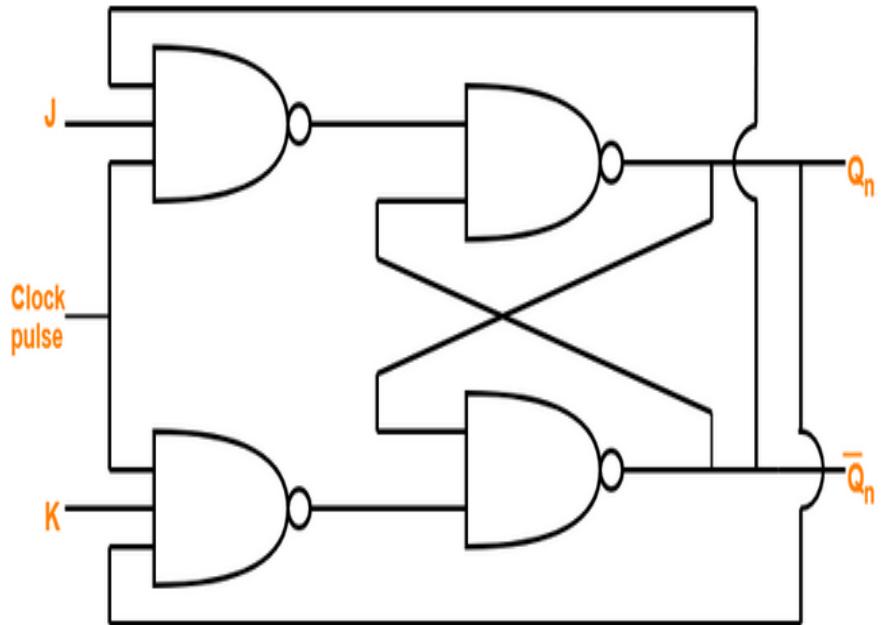
Clocked JK Flip-Flop

- The JK flip flop is the most versatile flip-flop, and the most commonly used flip flop.
- Like the RS flip-flop,it has two data inputs, J and K, and a clock pulse input (CP).
- In the following circuit diagram NAND gates are used instead of NOR gates.
- It has no undefined states, however.
- The fundamental difference of this device is the feedback paths.
- If the J and K inputs at 1 and the clock pulse is applied ,then the output will change state regardless of the previous condition.

Clocked JK Flip-Flop



Logic Symbol



Logic Circuit For JK Flip Flop Using SR Flip Flop

JK Flip Flop

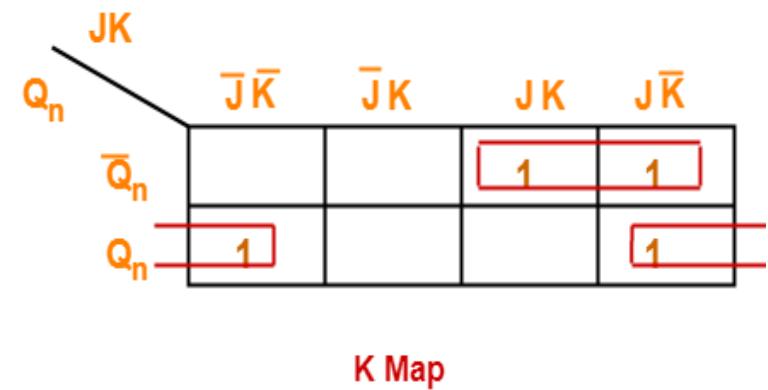
CLK	J	K	Q
0	X	X	Unchanged
1	0	0	Unchanged
1	0	0	Reset(0)
1	1	0	Set(1)
1	1	1	Toggle(Q?)

FUNCTIONAL TABLE

JK Flip Flop

Q_n	J	K	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

CHARACTERISTIC TABLE



$$Q_{n+1} = Q'_n J + Q_n K'$$

CHARACTERISTIC EQUATION

JK Flip Flop

$$Q_{n+1} = Q'_n J + Q_n K'$$

CHARACTERISTIC EQUATION

Q_n	Q_{n+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

EXCITATION TABLE

HDL for JK Flip Flop:

```
module JKFLIPFLOP(input j, input k, input clk, output reg q, output q_not);
assign q_not = !q;
always@(posedge clk)
begin
if(clk==1) q <= (q_not&&j) || (q&&(!k));
end;
endmodule;
```

THANK YOU

Subject name: Logic Design

Subject code: FET 1021

Semester: 3rd

Lecture 28:

Storage Elements : D and T flip-flops



**Department of Electronics & Communication Engineering,
Institute of Technical Education & Research,
S 'O' A (Deemed to be) University,
Khandagiri, Bhubaneswar, Orissa-751030**

Topics to be covered:

- D flip-flops, Graphical symbol, Logic diagram, Function Table, Characteristics Table, Characteristics Equation, Excitation table, and HDL description.
- T flip flops, Graphical symbol, Logic diagram, Function Table, Characteristics Table, Characteristics Equation, Excitation table, and HDL description.

Learning outcome:

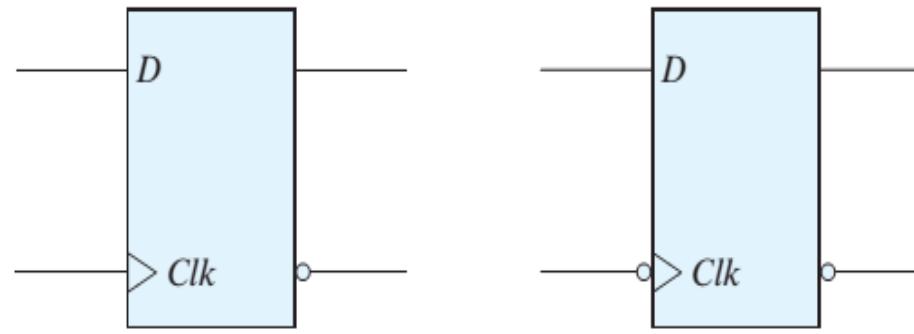
- Able to gather knowledge about the D and T Flip-flops.
- Able to design D & T FFs using HDL description

D flip-flop

- An SR flip-flop is rarely used in actual sequential logic because of its undefined outputs for inputs $R = S = 1$.
- It can be modified to form a more useful circuit called D flip-flop.
- The D flip-flop has only a single data input D as shown in the circuit diagram.
- That data input is connected to the S input of an SR flip-flop, while the inverse of D is connected to the R input.
- The D flip-flop also has a clock pulse as an input.
- It can be +ve edge triggered or -ve edge triggered.

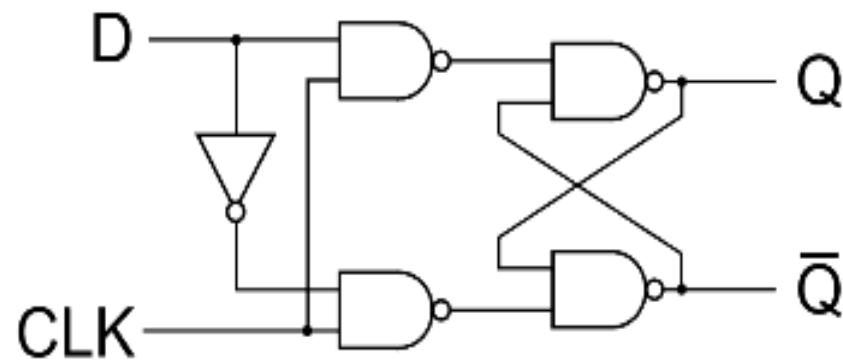
D flip-flop

Graphical symbol:



(a) Positive edge

(b) Negative edge

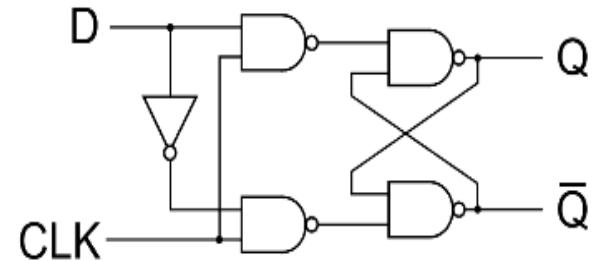


Logic Diagram of D flip-flop using NAND logic

D flip-flop

Operation:

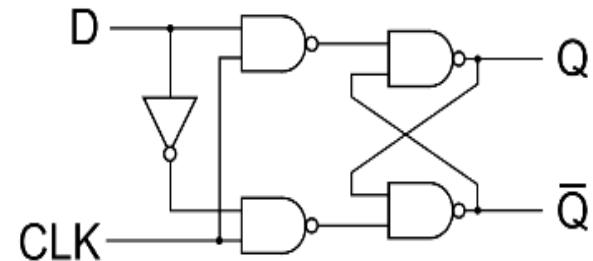
- By connecting an inverter (NOT gate) to the SR flip-flop we can SET and RESET the flip-flop.
- This complement avoids the ambiguity inherent in the SR latch when both inputs are LOW.
- If this data input is held HIGH the flip flop would be SET and when it is LOW the flip flop would change and become RESET. However, this would be rather pointless since the output of the flip flop would always change on every pulse applied to this data input.
- To avoid this additional input called the CLOCK is used to isolate the data input from the flip-flop's latching circuitry after the desired data has been stored.
- The effect is that D input condition is only copied to the output Q when the clock input is active.
- The D flip flop will store and output whatever logic level is applied to its data terminal so long as the clock input is HIGH



D flip-flop

Function or truth table:

Clock	D	Q_{t+1}	Description
$\downarrow(0)$	X	Q_t	No change
$\uparrow(1)$	0	0	Reset $Q=0$
$\uparrow(1)$	1	1	Set $Q=1$



Characteristic Table:

State		Input
Present state	Next state	D
0	0	0
0	1	1
1	0	0
1	1	1

Excitation Table:

Q_t	D	Q_{t+1}
0	0	0
0	1	1
1	0	0
1	1	1

Characteristic Equation: $Q_{(t+1)} = D$

Note: \downarrow and \uparrow indicates direction of clock pulse as it is assumed D-type flip-flops are edge triggered.

D flip-flop

D flip-flop vs D clocked latch

D flip flop:

- Triggered on the +ve edge of the clock.
- Output Q (and state) changes on a time instant.

D clocked latch:

- Output changes with D while clock is High.
- Output changes during a window of time.

D flip-flop

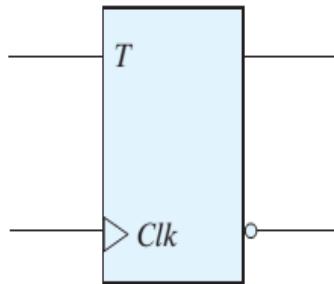
HDL code

```
module d_flip_flop(q,qb,clk,rst,d);
    input clk, rst, d;
    output q, qb;
    reg q,qb;
    always@(posedge clk)
        begin
            if (rst==1)
                q<=0;
            else
                q<=d;
            qb<=~q;
        end
    end module
```

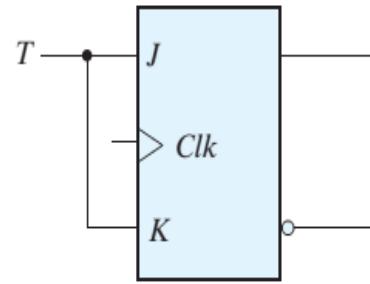
T flip-flop

- The T flip-flop has one input in addition to the clock
- T stands for toggle for the obvious reason.
- When T is asserted ($T = 1$), the flip-flop state toggles back and forth, and when T is de-asserted, the flip-flop keeps its current state.
- The T flip-flop can be constructed using a D flip-flop with the two outputs Q and Q' feedback to the D input through a multiplexer that is controlled by the T input.
- The T flip-flop is a single input version of the JK flip-flop.
- The T flip-flop is obtained from the JK type if both inputs are tied together

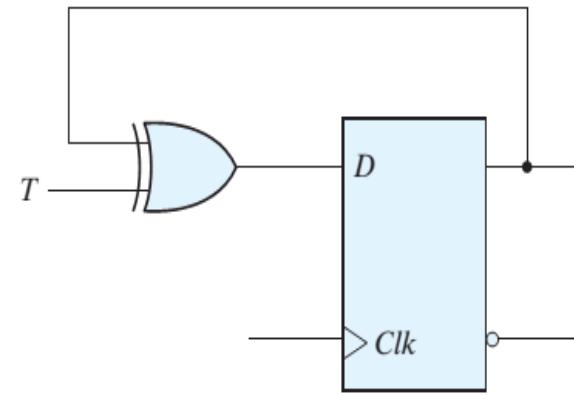
T flip-flop



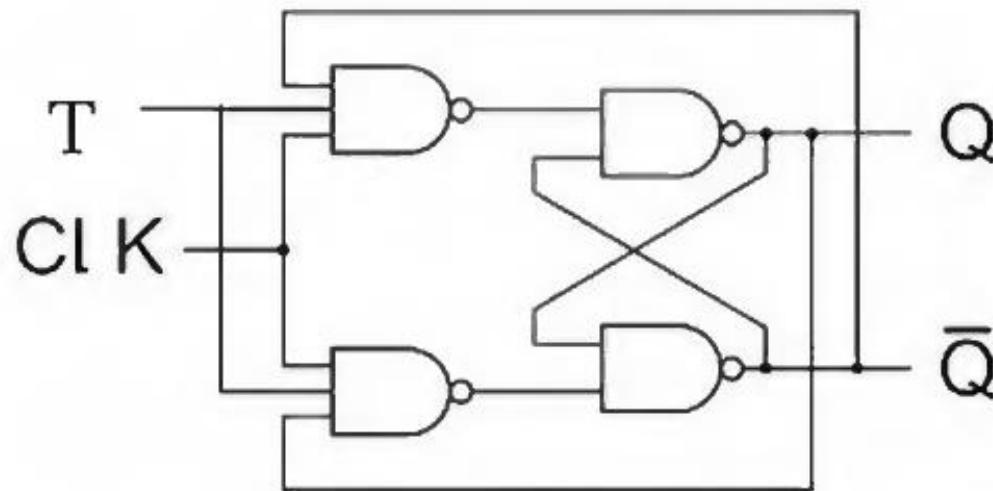
(a) Graphical Symbol



(b) From JK flip-flop



(c) From D flip-flop

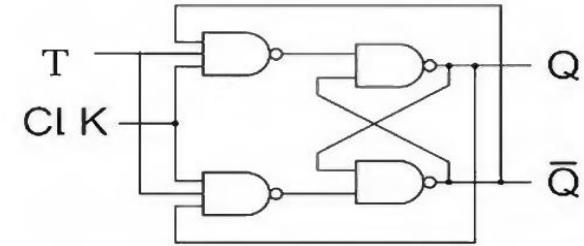


Logic Diagram of T flip-flop using NAND logic

T flip-flop

Operation:

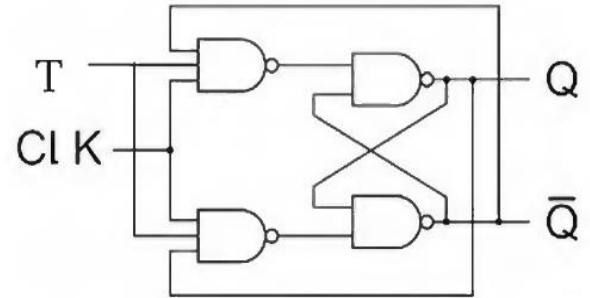
- If the output $Q = 0$, then the upper NAND is in enable state and lower NAND gate is in disable condition.
- This allows the trigger to pass the S inputs to make the flip –flop in SET state i.e. $Q = 1$.
- If the output $Q = 1$, then the upper NAND is in disable state and lower NAND gate is in enable condition. This allows the trigger to pass the R inputs to make the flip –flop in RESET state i.e. $Q=0$.
- In simple terms, the operation of the T flip –flop is
- When the T input is low, then the next state of the T flip flop is same as the present state.
- $T = 0$ and present state = 0 then the next state =0
- $T = 1$ and present state = 1 then the next state =1
- When the T input is high and during the positive transition of the clock signal, the next state of the T flip –flop is the inverse of present state.
- $T = 1$ and present state = 0 then the next state =1
- $T = 1$ and present state = 1 then the next state =0



T flip-flop

Function or truth table:

Clock	T	Q_{t+1}	Description
$\downarrow(0)$	X	Q_t	Memory
$\uparrow(1)$	0	Q_t	Memory
$\uparrow(1)$	1	Q'_t	Toggle



Characteristic Table:

Q_t	T	Q_{t+1}	Description
0	0	0	No change
0	1	1	Toggle
1	0	1	No change
1	1	0	Toggle

Excitation Table:

State		Input
Present state (Q_t)	Next state (Q_{t+1})	T
0	0	0
0	1	1
1	0	1
1	1	0

Characteristic Equation: $Q_{(t+1)} = T \cdot Q' + \bar{T} \cdot Q$

Note: \downarrow and \uparrow indicates direction of clock pulse as it is assumed T-type flip-flops are edge triggered.

T flip-flop

HDL code

```
module tff(t,clk,rst, q,qb);
    input t,clk,rst;
    output q,qb;
    reg q,qb;
    reg temp=0;
    always@(posedge clk,posedge rst)
    begin
        If (rst==0)
            begin
                if(t==1) begin
                    temp=~ temp;
                end
            else
                temp=temp;
        end
        q=temp;qb=~temp;
    end
end module
```

Assignment

1. Design T flip-flop using JK flip-flop.
2. Design D flip-flop using JK flip-flop.
3. What are the applications of D and T flip-flops?

**Thank you
End of Lecture-28**

State Reduction & Assignment

(Lecture-31)

State Reduction & Assignment

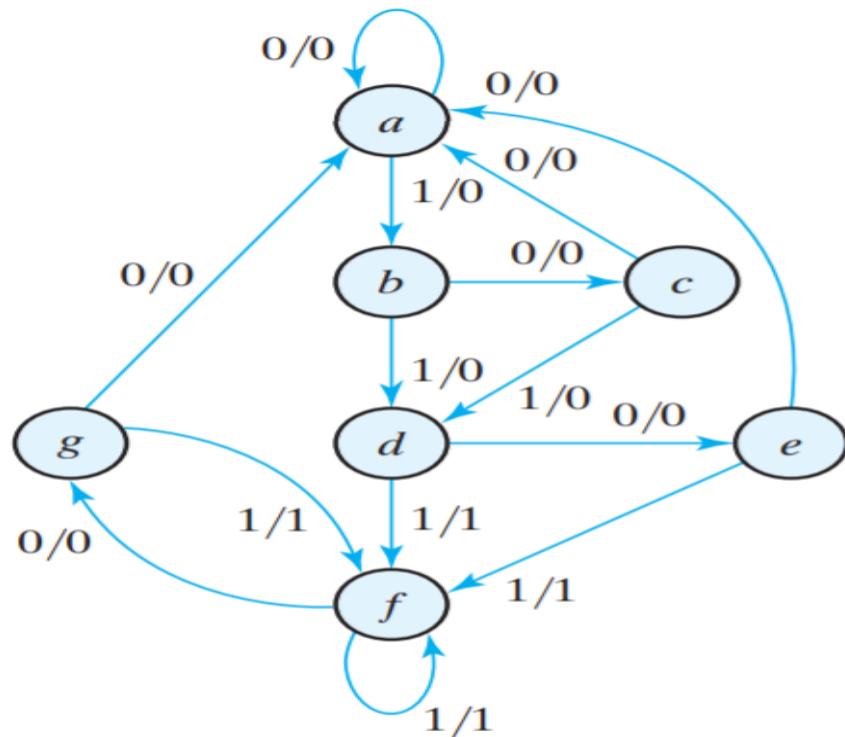
- The analysis of sequential circuits starts from a circuit diagram and culminates in a state table or diagram.
- The design (synthesis) of a sequential circuit starts from a set of specifications and culminates in a logic diagram.
- Two sequential circuits may exhibit the same input–output behaviour, but have a different number of internal states in their state diagram.
- State reduction and assignment discusses certain properties of sequential circuits that may simplify a design by reducing the number of gates and flip-flops it uses. In general, reducing the number of flipflops reduces the cost of a circuit.

State Reduction

- The reduction in the number of flip-flops in a sequential circuit is referred to as the state-reduction problem.
- State-reduction algorithms are concerned with procedures for reducing the number of states in a state table, while keeping the external input–output requirements unchanged.
- Since m flip-flops produce $2m$ states, a reduction in the number of states may (or may not) result in a reduction in the number of flip-flops.
- An unpredictable effect in reducing the number of flip-flops is that sometimes the equivalent circuit (with fewer flip-flops) may require more combinational gates to realize its next state and output logic.

State-reduction procedure example

- We start with a sequential circuit whose specification is given in the state diagram of Fig. below .



State-reduction procedure example

- In our example, only the input–output sequences are important; the internal states are used merely to provide the required sequences. For that reason, the states marked inside the circles are denoted by letter symbols instead of their binary values
- This is in contrast to a binary counter, wherein the binary value sequence of the states themselves is taken as the outputs.
- There are an infinite number of input sequences that may be applied to the circuit; each results in a unique output sequence.

Explanation of Example

- As an example, consider the input sequence 01010110100 starting from the initial state a . Each input of 0 or 1 produces an output of 0 or 1 and causes the circuit to go to the next state.
- From the state diagram, we obtain the output and state sequence for the given input sequence as follows: With the circuit in initial state a , an input of 0 produces an output of 0 and the circuit remains in state a . With present state a and an input of 1, the output is 0 and the next state is b . With present state b and an input of 0, the output is 0 and the next state is c . Continuing this process, we find the complete sequence to be as follows:

Explanation of Example

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

- In each column, we have the present state, input value, and output value.
- The next state is written on top of the next column.
- It is important to realize that in this circuit the states themselves are of secondary importance, because we are interested only in output sequences caused by input sequences.
- The problem of state reduction is to find ways of reducing the number of states in a sequential circuit without altering the input–output relationships.

Reduction of number of states

1. First, we need the state table; it is more convenient to apply procedures for state reduction with the use of a table rather than a diagram.
2. The state table of the circuit is listed in Table 5.6 and is obtained directly from the state diagram.
3. The following algorithm for the state reduction of a completely specified state table is given here without proof: “Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an equivalent state.” When two states are equivalent, one of them can be removed without altering the input–output relationships.
4. Now apply this algorithm to Table 5.6 . Going through the state table, we look for two present states that go to the same next state and have the same output for both input combinations.

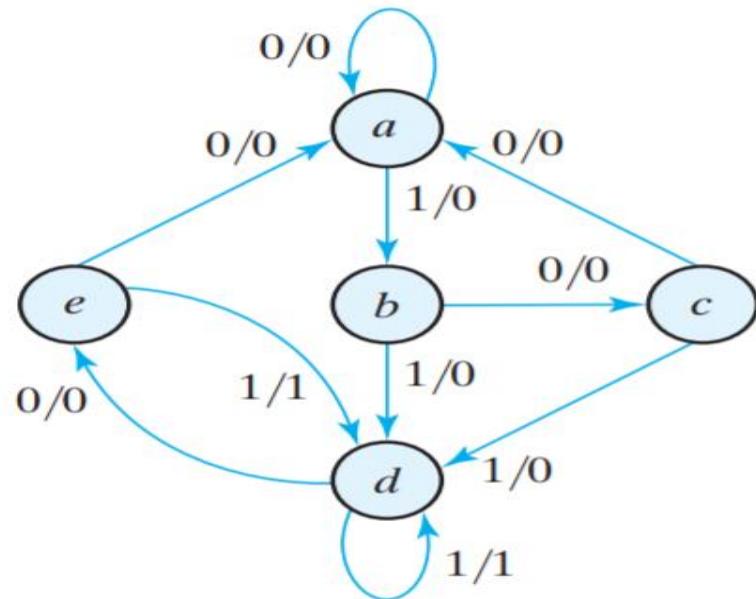
Reduction of number of states

5. States e and g are two such states: They both go to states a and f and have outputs of 0 and 1 for $x = 0$ and $x = 1$, respectively. Therefore, states g and e are equivalent, and one of these states can be removed.
6. The procedure of removing a state and replacing it by its equivalent is demonstrated in Table 5.7 .
7. The row with present state g is removed, and state g is replaced by state e each time it occurs in the columns headed “Next State.”
8. Present state f now has next states e and f and outputs 0 and 1 for $x = 0$ and $x = 1$, respectively . The same next states and outputs appear in the row with present state d . Therefore, states f and d are equivalent, and state f can be removed and replaced by d .

Reduction of number of states

State Table

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



Reducing the State Table

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

Reduced State Table

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

The state diagram for the reduced table consists of only five states

State Reduction

- The sequential circuit of this example was reduced from seven to five states.
- In general, reducing the number of states in a state table may result in a circuit with less equipment.
- However, the fact that a state table has been reduced to fewer states does not guarantee a saving in the number of flip-flops or the number of gates.
- In actual practice designers may skip this step because target devices are rich in resources.

State Assignment

- In order to design a sequential circuit with physical components, it is necessary to assign unique coded binary values to the states.
- For a circuit with m states, the codes must contain n bits, where $2^n \geq m$.
- For example, with three bits, it is possible to assign codes to eight states, denoted by binary numbers 000 through 111.
- If the state table is used, we must assign binary values to seven states; the remaining state is unused.
- If the reduced state table is used, only five states need binary assignment, and we are left with three unused states.

State Assignment

- Unused states are treated as don't-care conditions during the design.
- Since don't-care conditions usually help in obtaining a simpler circuit, it is more likely but not certain that the circuit with five states will require fewer combinational gates than the one with seven states.
- The simplest way to code five states is to use the first five integers in binary counting order, as shown in the first assignment of Table below . *Three Possible Binary State Assignments*

State	Assignment 1, Binary	Assignment 2, Gray Code	Assignment 3, One-Hot
<i>a</i>	000	000	00001
<i>b</i>	001	001	00010
<i>c</i>	010	011	00100
<i>d</i>	011	010	01000
<i>e</i>	100	110	10000

State Assignment

- The reduced state table with binary assignment 1 substituted for the letter symbols of the states.

Reduced State Table with Binary Assignment 1

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
000	000	001	0	0
001	010	011	0	0
010	000	011	0	0
011	100	011	0	1
100	000	011	0	1

State Assignment

- A different assignment will result in a state table with different binary values for the states.
- The binary form of the state table is used to derive the next state and output-forming combinational logic part of the sequential circuit.
- The complexity of the combinational circuit depends on the binary state assignment chosen.
- Sometimes, the name transition table is used for a state table with a binary assignment. This convention distinguishes it from a state table with symbolic names for the states.

Design Procedure

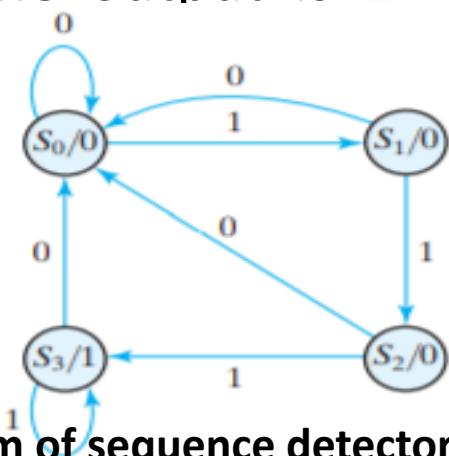
- From the word description and specifications of the desired operation, derive a state diagram for the circuit.
- Reduce the number of states if necessary.
- Assign binary values to the states.
- Obtain the binary-coded state table.
- Choose the type of flip-flops to be used.
- Derive the simplified flip-flop input equations and output equations.
- Draw the logic diagram.

Design Procedure

- The word specification of the circuit behaviour usually assumes that the reader is familiar with digital logic terminology.
- It is necessary that the designer use intuition and experience to arrive at the correct interpretation of the circuit specifications, because word descriptions may be incomplete and inexact.
- Once such a specification has been set down and the state diagram obtained, it is possible to use known synthesis procedures to complete the design.
- Although there are formal procedures for state reduction and assignment (steps 2 and 3), they are seldom used by experienced designers. Steps 4 through 7 in the design can be implemented by exact algorithms and therefore can be automated.
- The part of the design that follows a well-defined procedure is referred to as synthesis .

Design Procedure

- It is derived by starting with state S0, the reset state. If the input is 0, the circuit stays in S0, but if the input is 1, it goes to state S1 to indicate that a 1 was detected.
- If the next input is 1, the change is to state S2 to indicate the arrival of two consecutive 1's, but if the input is 0, the state goes back to S0. The third consecutive 1 sends the circuit to state S3.
- If more 1's are detected, the circuit stays in S3. Any 0 input sends the circuit back to S0. In this way, the circuit stays in S3 as long as there are three or more consecutive 1's received.
- This is a Moore model sequential circuit, since the output is 1 when the circuit is in state S3 and is 0 otherwise.



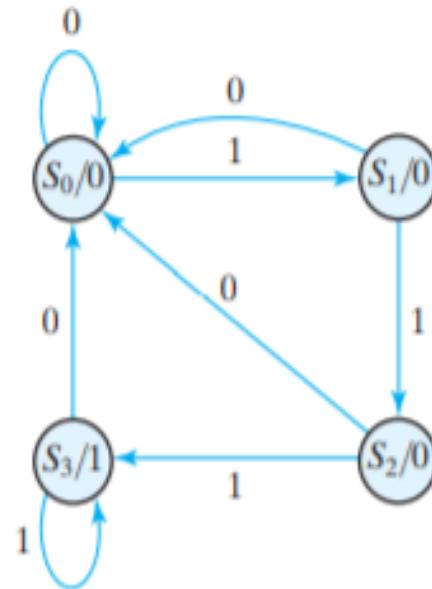
State diagram of sequence detector

Synthesis Using D Flip-Flops

- To design the circuit by hand, we need to assign binary codes to the states and list the state table.

State Table for Sequence Detector

Present State		Input <i>x</i>	Next State		Output <i>y</i>
<i>A</i>	<i>B</i>		<i>A</i>	<i>B</i>	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1



- The table is derived from the state diagram with a sequential binary assignment.

Synthesis Using D Flip-Flops

- We choose two D flip-flops to represent the four states, and we label their outputs A and B . There is one input x and one output y .
- The characteristic equation of the D flip-flop is $Q(t + 1) = DQ$, which means that the next-state values in the state table specify the D input condition for the flip-flop.
- The flip-flop input equations can be obtained directly from the next-state columns of A and B and expressed in sum-of-minterms form as

$$A(t + 1) = D_A(A, B, x) = \sum(3,5,7)$$

$$B(t + 1) = D_B(A, B, x) = \sum(1,5,7)$$

$$y(A, B, x) = \sum(6,7)$$

where A and B are the present-state values of flip-flops A and B, x is the input, and D_A and D_B are the input equations. The minterms for output y are obtained from the output column in the state table

Synthesis Using D Flip-Flops

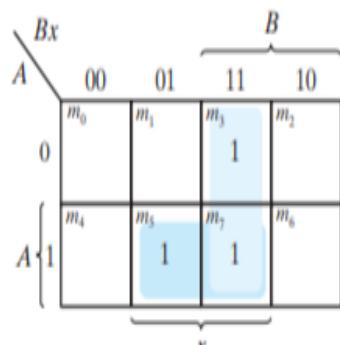
- The Boolean equations are simplified by means of the maps plotted .

The simplified equations are

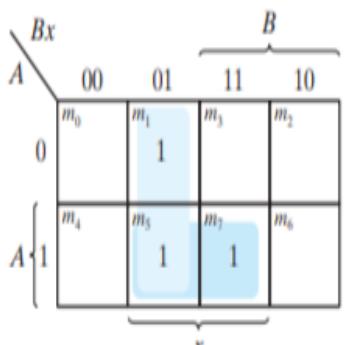
$$D_A = Ax + Bx$$

$$D_B = Ax + B'x$$

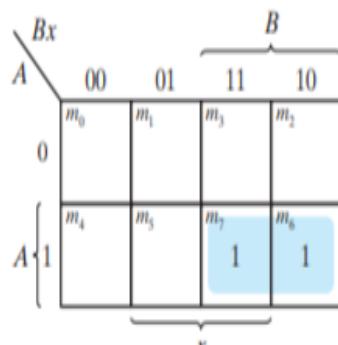
$$Y=AB$$



$$D_A = Ax + Bx$$



$$D_B = Ax + B'x$$

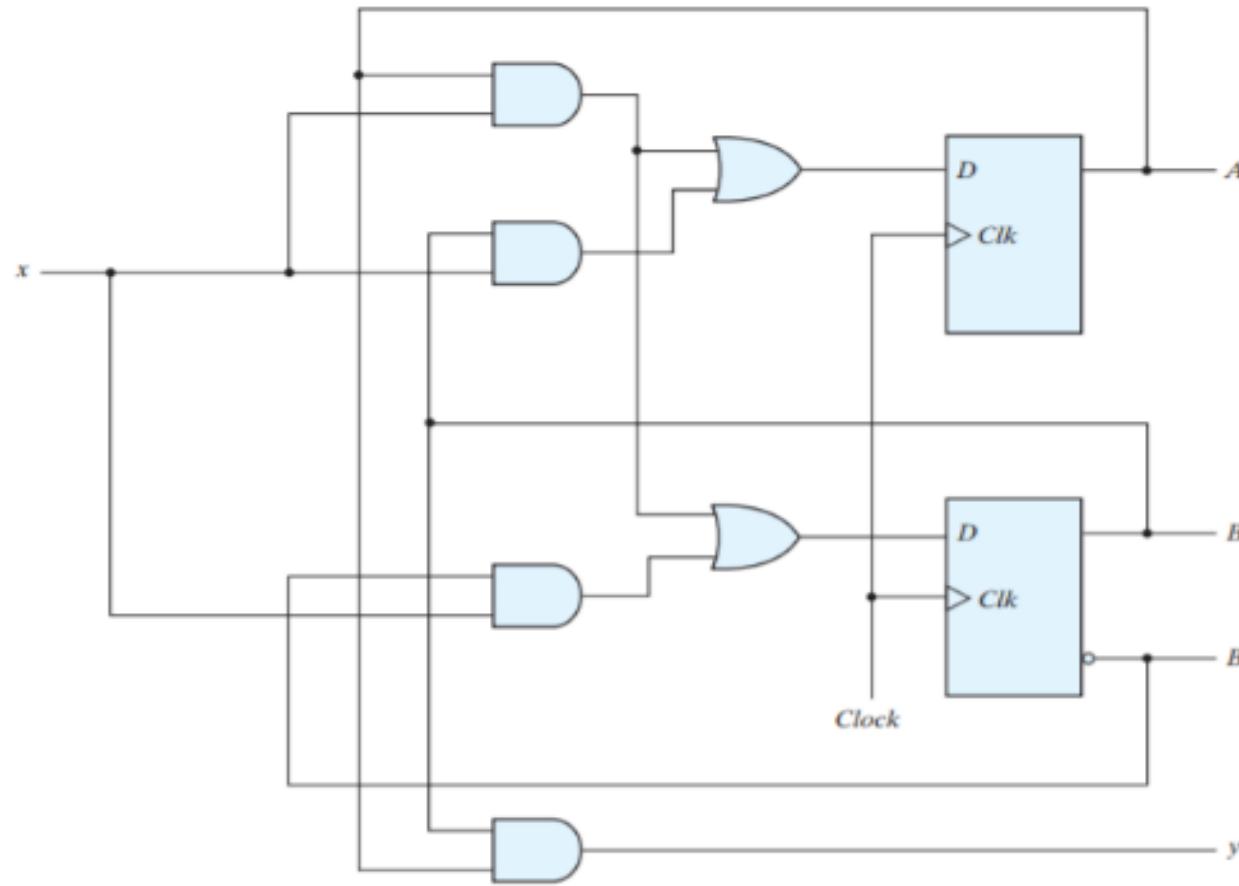


$$y = AB$$

K maps for sequence detector

- The advantage of designing with D flip-flops is that the Boolean equations describing the inputs to the flip-flops can be obtained directly from the state table.

Synthesis Using D Flip-Flops



The schematic of the sequential circuit

Synthesis Using D Flip-Flops

- The design of a sequential circuit with flip-flops other than the D type is complicated by the fact that the input equations for the circuit must be derived indirectly from the state table.
- When D -type flip-flops are employed, the input equations are obtained directly from the next state. This is not the case for the JK and T types of flip-flops.
- In order to determine the input equations for these flip-flops, it is necessary to derive a functional relationship between the state table and the input equations.
- The flip-flop characteristic tables provide the value of the next state when the inputs and the present state are known.
- These tables are useful for analysing sequential circuits and for defining the operation of the flip-flops.

Synthesis Using D Flip-Flops

- During the design process, we usually know the transition from the present state to the next state and wish to find the flip-flop input conditions that will cause the required transition.
- For this reason, we need a table that lists the required inputs for a given change of state. Such a table is called an excitation table .

Synthesis Using D Flip-Flops

- The excitation tables for the two flip-flops (JK and T).

Flip-Flop Characteristic Tables		
JK Flip-Flop		
J	K	$Q(t + 1)$
0	0	$Q(t)$ No change
0	1	0 Reset
1	0	1 Set
1	1	$Q'(t)$ Complement



Flip-Flop Excitation Tables						
$Q(t)$	$Q(t = 1)$	J	K	$Q(t)$	$Q(t = 1)$	T
0	0	0	X	0	0	0
0	1	1	X	0	1	1
1	0	X	1	1	0	1
1	1	X	0	1	1	0

(a) JK Flip-Flop

(b) T Flip-Flop

D Flip-Flop		
D	$Q(t + 1)$	T
0	0	Reset
1	1	Set

T Flip-Flop		
T	$Q(t + 1)$	$Q(t)$
0	$Q(t)$ No change	0
1	$Q'(t)$ Complement	1

- Each table has a column for the present state Q (t), a column for the next state Q(t + 1), and a column for each input to show how the required transition is achieved.

Synthesis Using D Flip-Flops

- There are four possible transitions from the present state to the next state. The required input conditions for each of the four transitions are derived from the information available in the characteristic table.
- The symbol X in the tables represents a don't-care condition, which means that it does not matter whether the input is 1 or 0.

Synthesis Using JK Flip-Flops

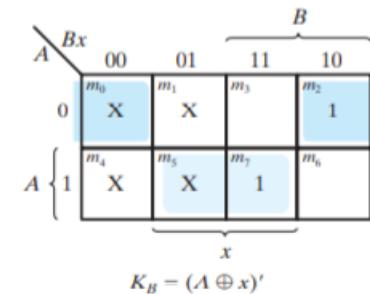
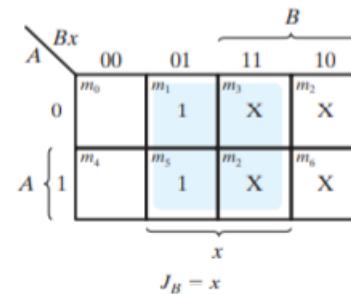
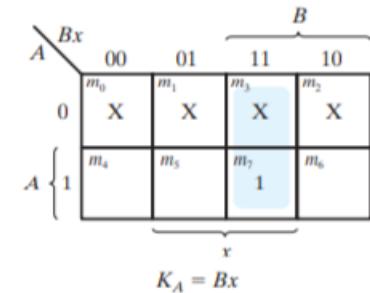
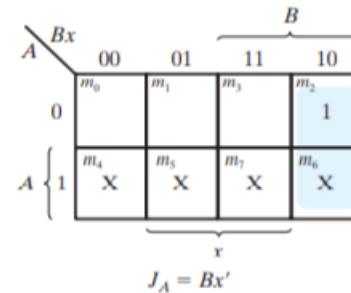
- The manual synthesis procedure for sequential circuits with JK flip-flops is the same as with D flip-flops, except that the input equations must be evaluated from the present state to the next-state transition derived from the excitation table.
- In addition to having columns for the present state, input, and next state, as in a conventional state table, the table shows the flip-flop input conditions from which the input equations are derived. The flip-flop inputs are derived from the state table in conjunction with the excitation table for the JK flip-flop.

Synthesis Using JK Flip-Flops

- Synthesis the sequential circuit

State Table and JK Flip-Flop Inputs

Present State		Input <i>x</i>	Next State		Flip-Flop Inputs			
<i>A</i>	<i>B</i>		<i>A</i>	<i>B</i>	<i>J_A</i>	<i>K_A</i>	<i>J_B</i>	<i>K_B</i>
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1



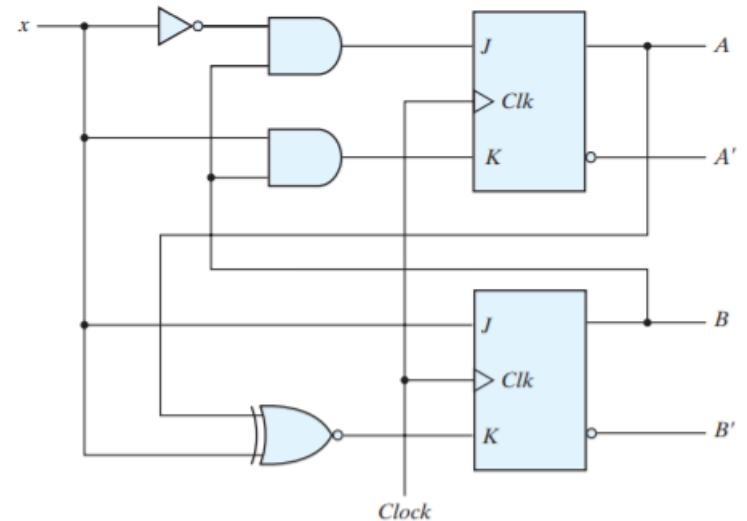
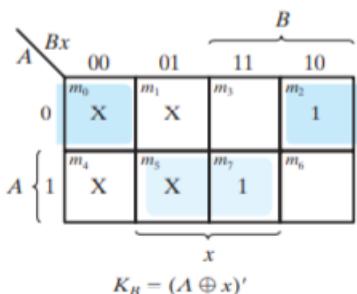
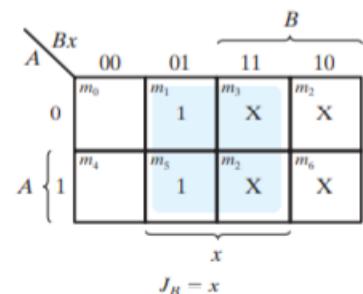
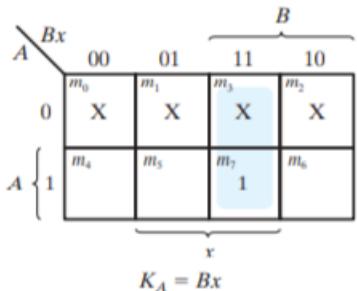
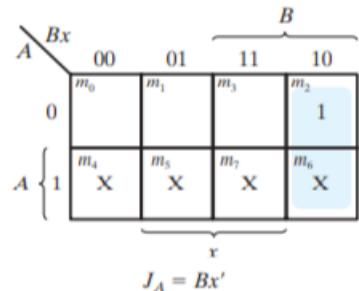
Maps for j and k input equations

Advantage of using JK -type flip-flops

- The fact that there are so many don't-care entries indicates that the combinational circuit for the input equations is likely to be simpler, because don't-care minterms usually help in obtaining simpler expressions.
- If there are unused states in the state table, there will be additional don't-care conditions in the map. Nonetheless, D-type flip-flops are more amenable to an automated design flow.

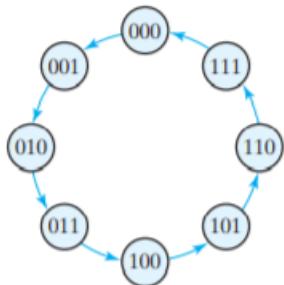
Advantage of using JK -type flip-flops

- The four input equations for the pair of JK flip-flops are listed under the maps of Fig. in LHS.



- The logic diagram (schematic) of the sequential circuit is drawn in RHS

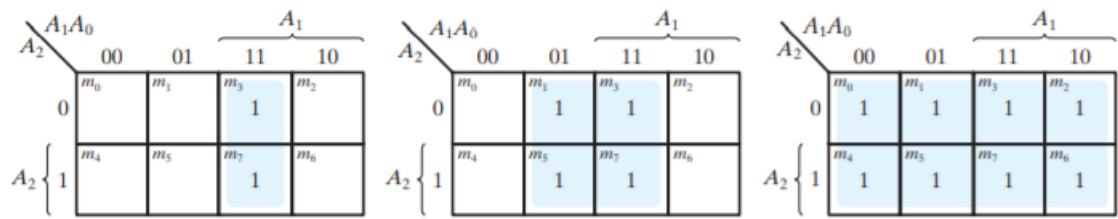
Synthesis using T flip-flops



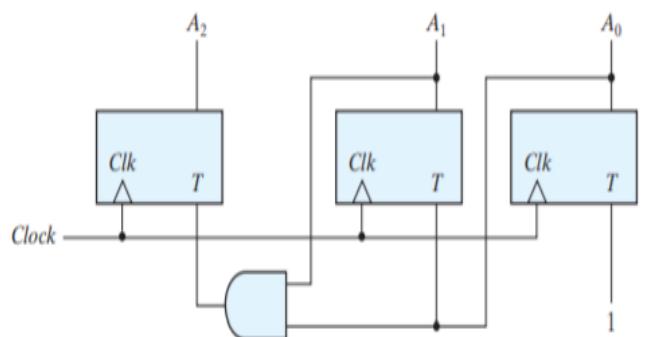
State Diagram of 3 bit binary counter

State Table for Three-Bit Counter

Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A2}	T_{A1}	T_{A0}
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	1	1
1	1	1	0	0	0	1	1	1



Maps for 3 bit binary counter



Logic Diagram of 3 bit binary counter

Assignment

1. A sequential circuit has two JK flip-flops A and B and one input x . The circuit is described by the following flip-flop input equations:

$$J_A = x \quad K_A = B$$

$$J_B = x \quad K_B = A'$$

- (a) Derive the state equations $A(t + 1)$ and $B(t + 1)$ by substituting the input equations for the J and K variables.
- (b) Draw the state diagram of the circuit.
2. A sequential circuit has two JK flip-flops A and B , two inputs x and y , and one output z . The flip-flop input equations and circuit output equation are

$$J_A = Bx + B'y' \quad K_A = B'xy'$$

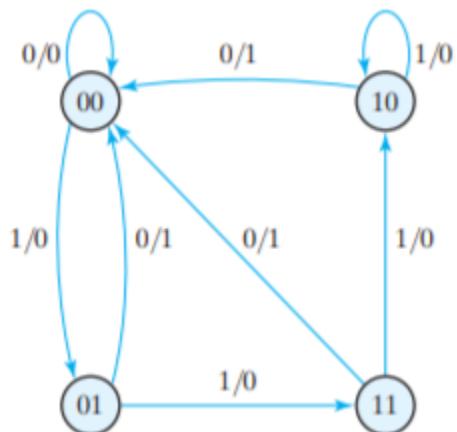
$$J_B = A'x \quad K_B = A + xy'$$

$$z = Ax'y' + Bx'y'$$

- (a) Draw the logic diagram of the circuit.
- (b) Tabulate the state table.
- (c) Derive the state equations for A and B .

Assignment

3. For the circuit described by the state diagram of Fig.
- Determine the state transitions and output sequence that will be generated when an input sequence of 010110111011110 is applied to the circuit and it is initially in the state 00.
 - Find all of the equivalent states in Fig. 5.16 and draw a simpler, but equivalent, state diagram.
 - Using D flip-flops, design the equivalent machine (including its logic diagram) described by the state diagram in (b).



Assignment

3. For the following state table

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

- (a) Draw the corresponding state diagram.
- (b)* Tabulate the reduced state table.
- (c) Draw the state diagram corresponding to the reduced state table.

Assignment

4.

Starting from state a , and the input sequence 01110010011, determine the output sequence for

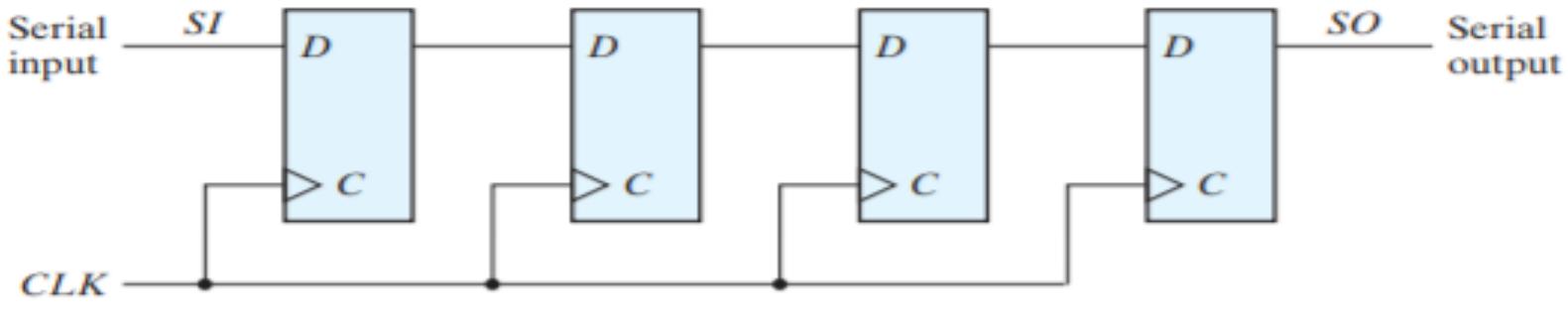
- (a) The state table of the previous problem.
- (b) The reduced state table from the previous problem. Show that the same output sequence is obtained for both.

Shift Registers

(Lecture-32)

Shift Registers

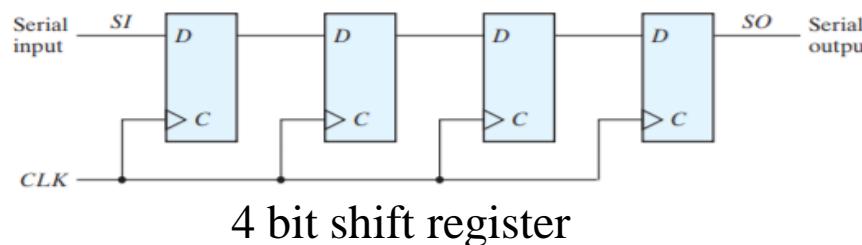
- A register capable of shifting the binary information held in each cell to its neighbouring cell, in a selected direction, is called a shift register.
- The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the input of the next flip-flop.
- All flip-flops receive common clock pulses, which activate the shift of data from one stage to the next.
- The simplest possible shift register is one that uses only flip-flops, as shown in Fig.



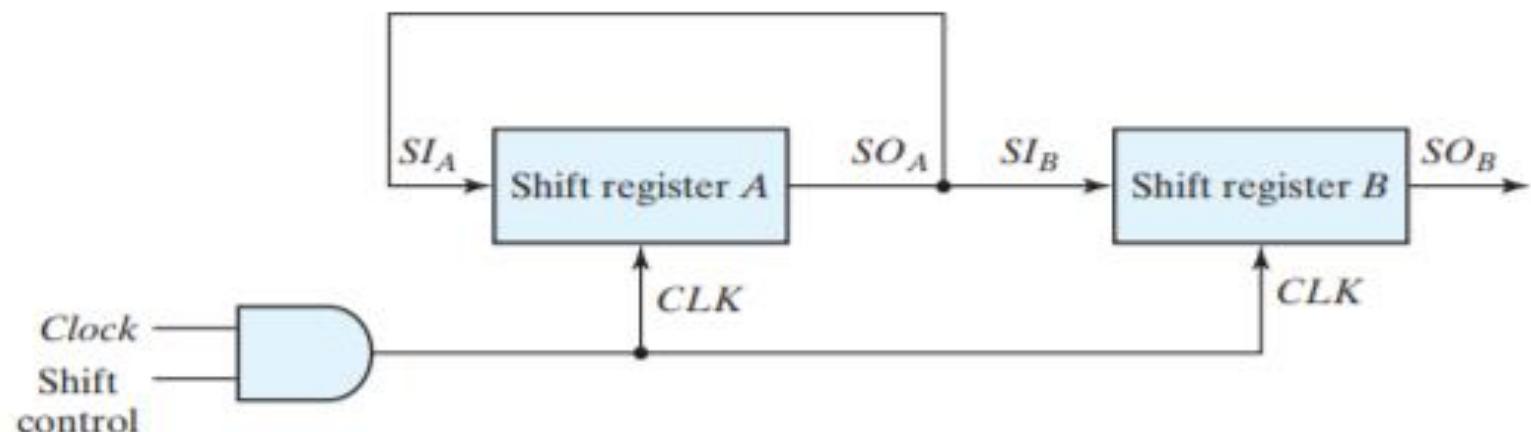
4 bit shift register

Shift Registers

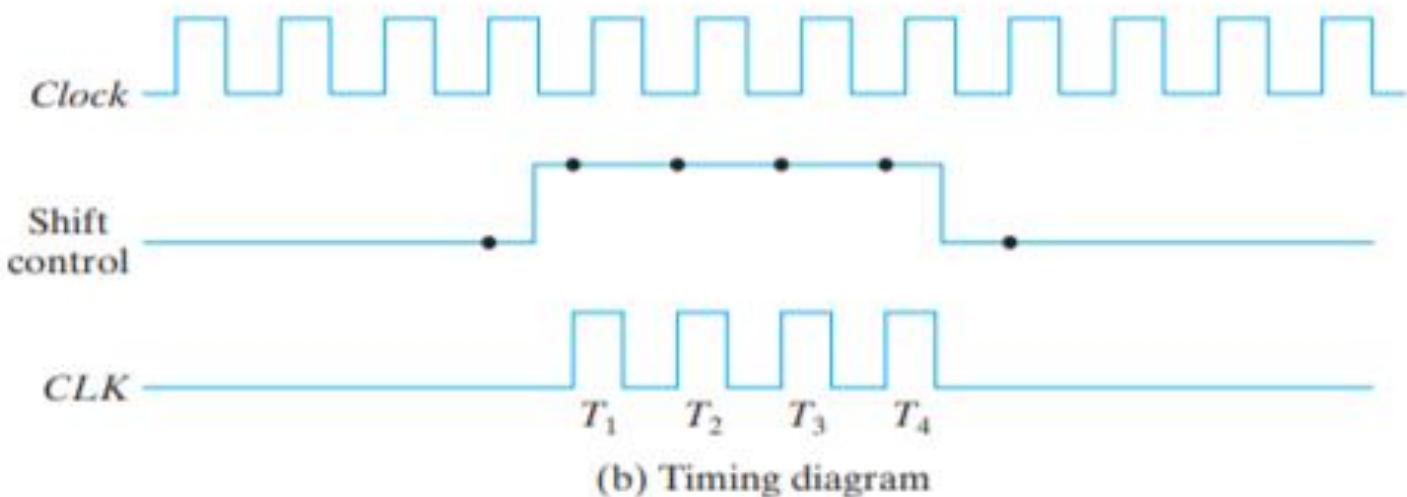
- The output of a given flip-flop is connected to the D input of the flip-flop at its right.
- This shift register is unidirectional (left-to-right).
- Each clock pulse shifts the contents of the register one bit position to the right.
- The configuration does not support a left shift.
- The serial input determines what goes into the leftmost flip-flop during the shift.
- The serial output is taken from the output of the rightmost flip-flop.
- Sometimes it is necessary to control the shift so that it occurs only with certain pulses, but not with others.



Serial transfer from register A to register B



(a) Block diagram



(b) Timing diagram

Shift Registers

- Flip flops can be used to store a single bit of binary data (1 or 0). However, in order to store multiple bits of data, we need multiple flip flops.
- N flip flops are to be connected in an order to store n bits of data.
- A **Register** is a device which is used to store such information. It is a group of flip flops connected in series used to store multiple bits of data . The information stored within these registers can be transferred with the help of **shift registers**.

Shift Registers

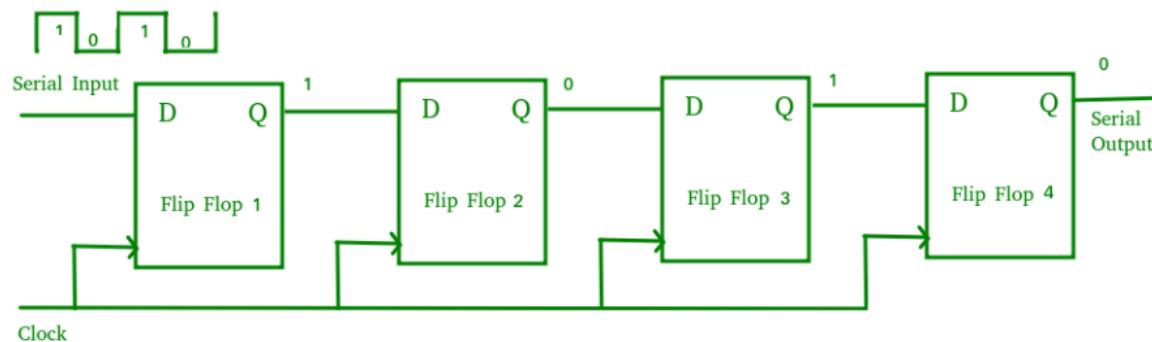
- Shift Register is a group of flip flops used to store multiple bits of data. The bits stored in such registers can be made to move within the registers and in/out of the registers by applying clock pulses.
- An n-bit shift register can be formed by connecting n flip-flops where each flip flop stores a single bit of data. The registers which will shift the bits to left are called “Shift left registers”.
- The registers which will shift the bits to right are called “Shift right registers”.

Shift Registers Types

- Serial In Serial Out shift register
- Serial In parallel Out shift register
- Parallel In Serial Out shift register
- Parallel In parallel Out shift register

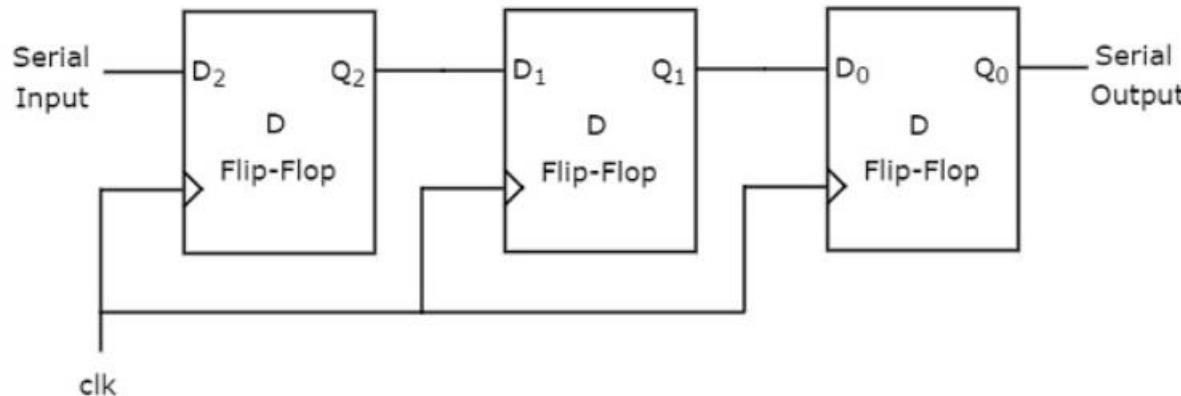
SISO Shift Registers

- The shift register, which allows serial input (one bit after the other through a single data line) and produces a serial output is known as Serial-In Serial-Out shift register.
- Since there is only one output, the data leaves the shift register one bit at a time in a serial pattern, thus the name Serial-In Serial-Out Shift Register.
- The circuit consists of four D flip-flops which are connected in a serial manner. All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.



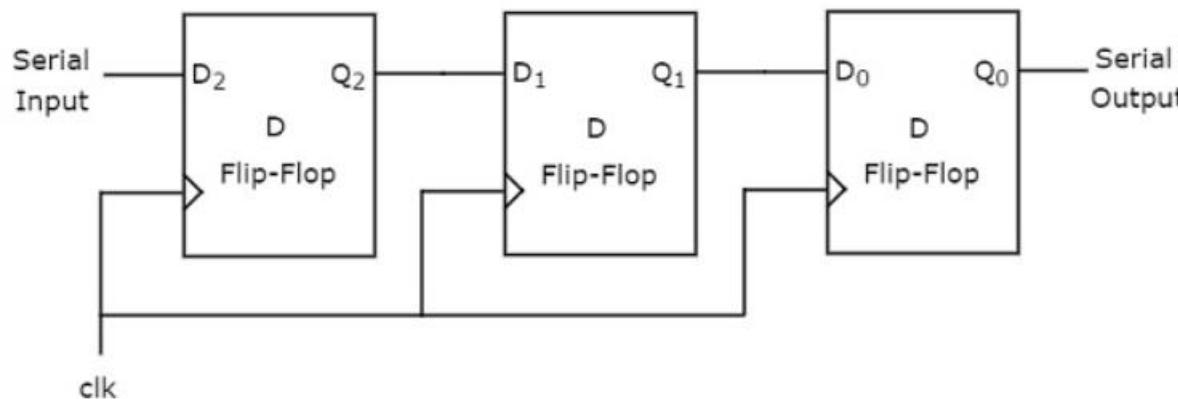
SISO Shift Registers

- The above circuit is an example of shift right register, taking the serial data input from the left side of the flip flop.
- The main use of a SISO is to act as a delay element.
- The shift register, which allows serial input and produces serial output is known as Serial In – Serial Out SISO shift register.
- The **block diagram** of 3-bit SISO shift register is shown in the following figure.



SISO Shift Registers

- This block diagram consists of three D flip-flops, which are **cascaded**. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.
- In this shift register, we can send the bits serially from the input of left most D flip-flop. Hence, this input is also called as **serial input**. For every positive edge triggering of clock signal, the data shifts from one stage to the next. So, we can receive the bits serially from the output of right most D flip-flop. Hence, this output is also called as **serial output**.



Example

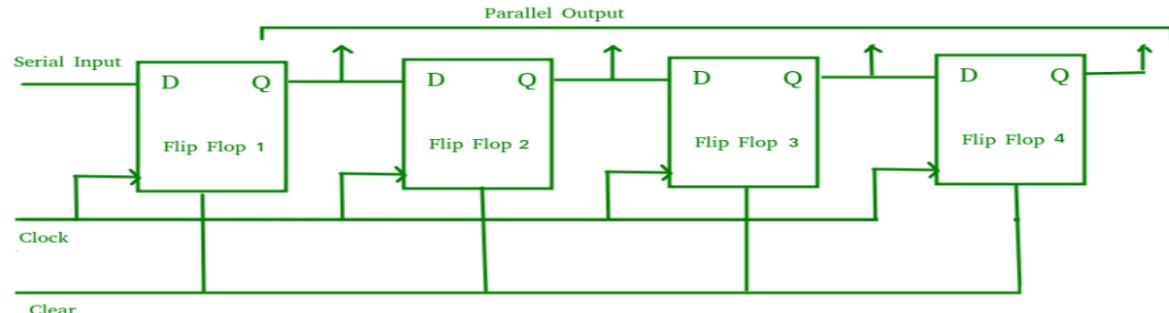
- Let us see the working of 3-bit SISO shift register by sending the binary information “**011**” from LSB to MSB serially at the input.
- Assume, initial status of the D flip-flops from leftmost to rightmost is Q₂Q₁Q₀=000 Q₂Q₁Q₀=000. We can understand the **working of 3-bit SISO shift register** from the following table.
- The initial status of the D flip-flops in the absence of clock signal is Q₂Q₁Q₀=000Q₂Q₁Q₀=000. Here, the serial output is coming from Q₀Q₀. So, the LSB 11 is received at 3rd positive edge of clock and the MSB 00 is received at 5th positive edge of clock.
- Therefore, the 3-bit SISO shift register requires five clock pulses in order to produce the valid output. Similarly, the **N-bit SISO shift register** requires **2N-1** clock pulses in order to shift ‘N’ bit information.

Example

No of positive edge of Clock	Serial Input	Q_2	Q_1	Q_0
0	-	0	0	0
1	1 <i>LSB</i>	1	0	0
2	1	1	1	0
3	0 <i>MSB</i>	0	1	1 <i>LSB</i>
4	-	-	0	1
5	-	-	-	0 <i>MSB</i>

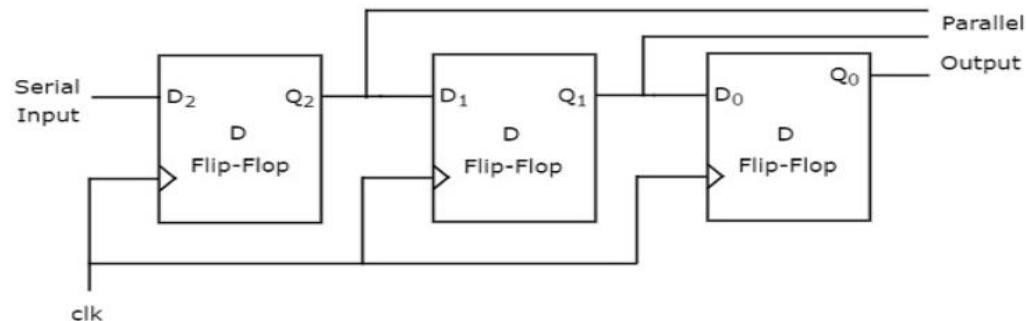
SIFO Shift Registers

- The shift register, which allows serial input (one bit after the other through a single data line) and produces a parallel output is known as Serial-In Parallel-Out shift register.
- The logic circuit given below shows a serial-in-parallel-out shift register.
- The circuit consists of four D flip-flops which are connected. The clear (CLR) signal is connected in addition to the clock signal to all the 4 flip flops in order to RESET them.
- The output of the first flip flop is connected to the input of the next flip flop and so on.
- All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.



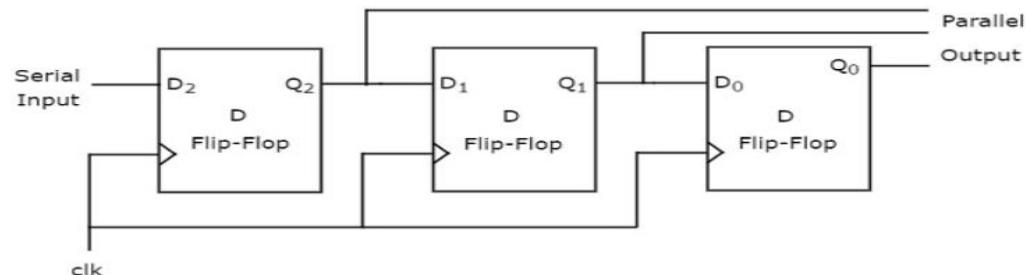
SIPO Shift Registers

- The above circuit is an example of shift right register, taking the serial data input from the left side of the flip flop and producing a parallel output.
- They are used in communication lines where demultiplexing of a data line into several parallel lines is required because the main use of the SIPO register is to convert serial data into parallel data.
- The shift register, which allows serial input and produces parallel output is known as Serial In – Parallel Out SIPOSIP0 shift register.
- The **block diagram** of 3-bit SIPO shift register is shown in the following figure.



SISO Shift Registers

- This circuit consists of three D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop.
- All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.
- In this shift register, we can send the bits serially from the input of left most D flip-flop. Hence, this input is also called as **serial input**.
- For every positive edge triggering of clock signal, the data shifts from one stage to the next. In this case, we can access the outputs of each D flip-flop in parallel. So, we will get **parallel outputs** from this shift register.



Example

- Let us see the working of 3-bit SIPO shift register by sending the binary information “**011**” from LSB to MSB serially at the input.
- Assume, initial status of the D flip-flops from leftmost to rightmost is $Q_2 Q_1 Q_0 = 000$ $Q_2 Q_1 Q_0 = 000$. Here, $Q_2 Q_2$ & $Q_0 Q_0$ are MSB & LSB respectively. We can understand the **working of 3-bit SIPO shift register** from the following table.
- The initial status of the D flip-flops in the absence of clock signal is $Q_2 Q_1 Q_0 = 000$ $Q_2 Q_1 Q_0 = 000$. The binary information “**011**” is obtained in parallel at the outputs of D flip-flops for third positive edge of clock.
- So, the 3-bit SIPO shift register requires three clock pulses in order to produce the valid output. Similarly, the **N-bit SIPO shift register** requires **N** clock pulses in order to shift ‘N’ bit information.

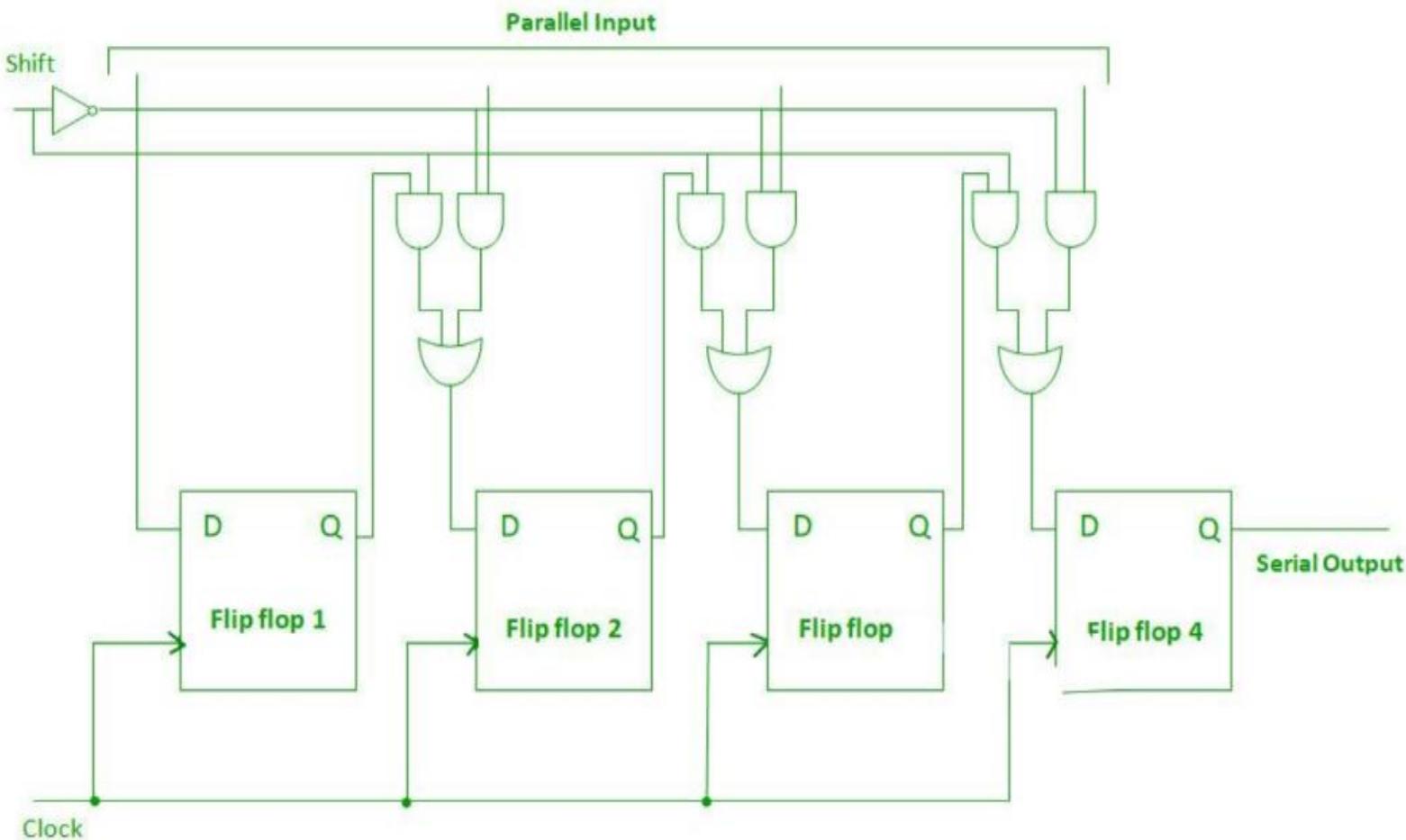
Example

No of positive edge of Clock	Serial Input	Q_2 MSB	Q_1	Q_0 LSB
0	-	0	0	0
1	1 LSB	1	0	0
2	1	1	1	0
3	0 MSB	0	1	1

PISO Shift Registers

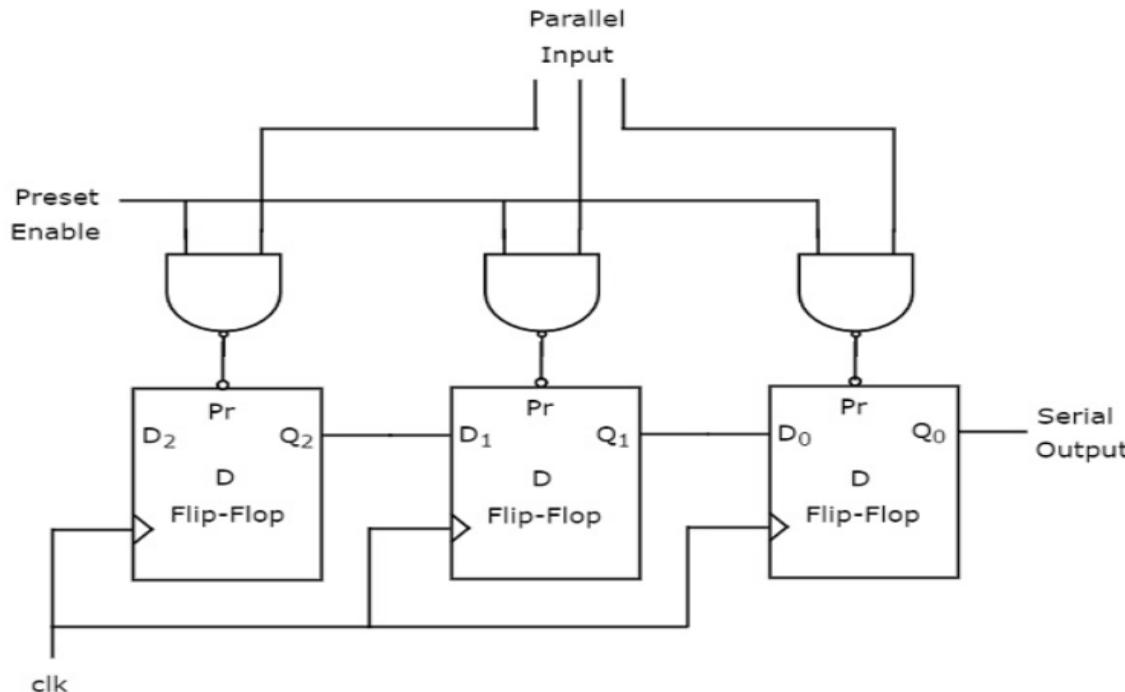
- The shift register, which allows parallel input (data is given separately to each flip flop and in a simultaneous manner) and produces a serial output is known as Parallel-In Serial-Out shift register.
- The logic circuit given below shows a parallel-in-serial-out shift register. The circuit consists of four D flip-flops which are connected.
- The clock input is directly connected to all the flip flops but the input data is connected individually to each flip flop through a multiplexer at the input of every flip flop.
- The output of the previous flip flop and parallel data input are connected to the input of the MUX and the output of MUX is connected to the next flip flop.
- All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.

PISO Shift Registers



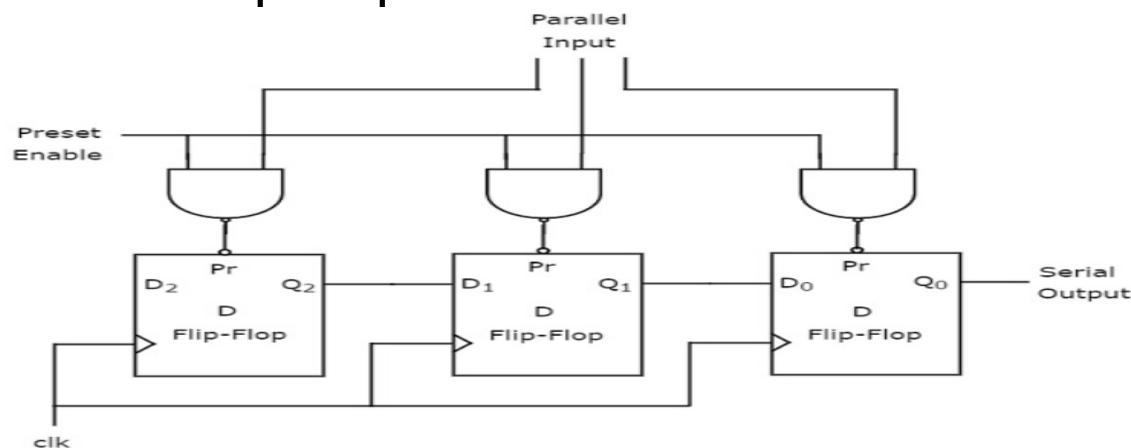
PISO Shift Registers

- A Parallel in Serial out (PISO) shift register us used to convert parallel data to serial data.
- The shift register, which allows parallel input and produces serial output is known as Parallel In – Serial Out PISO shift register.
- The **block diagram** of 3-bit PISO shift register is shown in the following figure



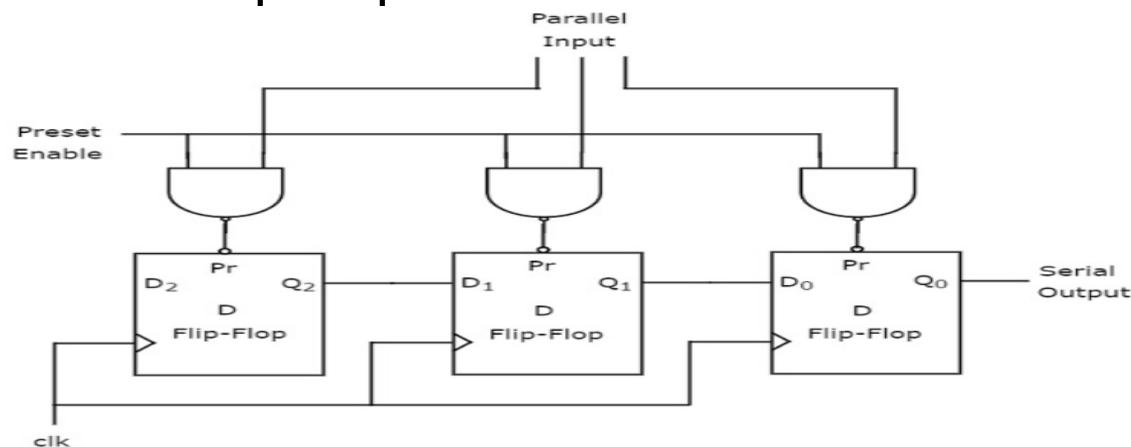
PISO Shift Registers

- This circuit consists of three D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop.
- All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.
- In this shift register, we can apply the **parallel inputs** to each D flip-flop by making Preset Enable to 1.
- For every positive edge triggering of clock signal, the data shifts from one stage to the next. So, we will get the **serial output** from the right most D flip-flop.



Example

- This circuit consists of three D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop.
- All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.
- In this shift register, we can apply the **parallel inputs** to each D flip-flop by making Preset Enable to 1.
- For every positive edge triggering of clock signal, the data shifts from one stage to the next. So, we will get the **serial output** from the right most D flip-flop.



Example

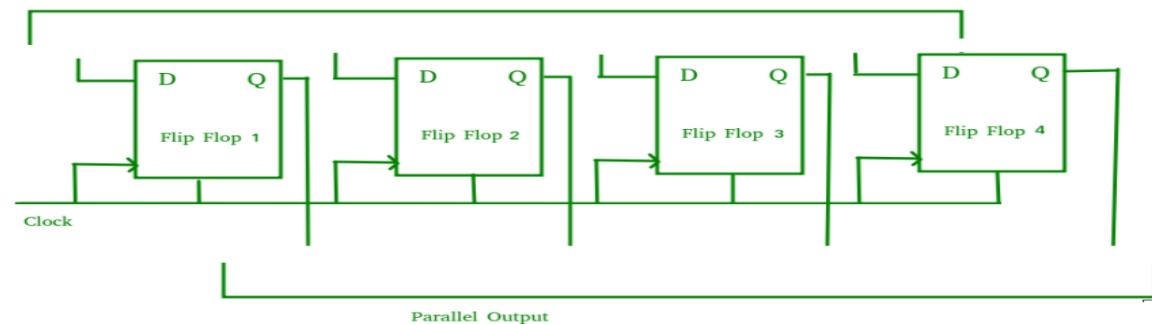
- Let us see the working of 3-bit PISO shift register by applying the binary information “**011**” in parallel through preset inputs.
- Since the preset inputs are applied before positive edge of Clock, the initial status of the D flip-flops from leftmost to rightmost will be $Q_2Q_1Q_0=011Q_2Q_1Q_0=011$. We can understand the **working of 3-bit PISO shift register** from the following table.
- Here, the serial output is coming from Q_0Q_0 . So, the LSB 11 is received before applying positive edge of clock and the MSB 00 is received at 2nd positive edge of clock.
- Therefore, the 3-bit PISO shift register requires two clock pulses in order to produce the valid output. Similarly, the **N-bit PISO shift register** requires **N-1** clock pulses in order to shift ‘N’ bit information.

Example

No of positive edge of Clock	Q_2	Q_1	Q_0
0	0	1	1 LSB
1	-	0	1
2	-	-	0 LSB

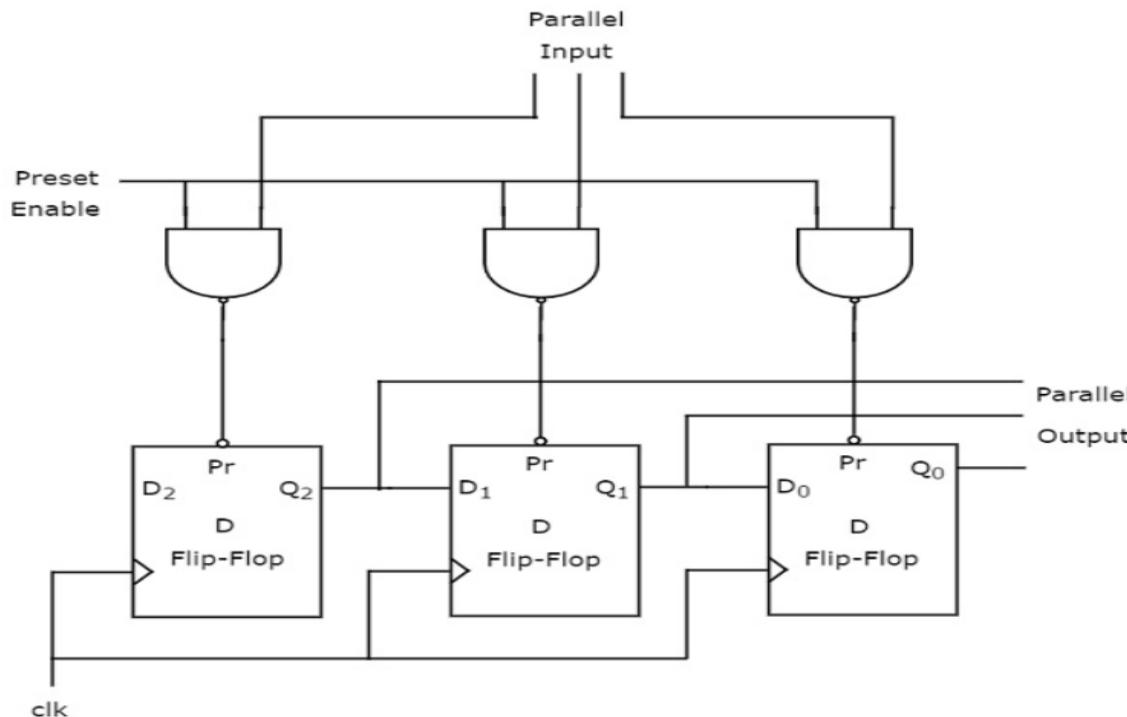
PIPO Shift Registers

- The shift register, which allows parallel input (data is given separately to each flip flop and in a simultaneous manner) and also produces a parallel output is known as Parallel-In parallel-Out shift register.
- The logic circuit given below shows a parallel-in-parallel-out shift register. The circuit consists of four D flip-flops which are connected. The clear (CLR) signal and clock signals are connected to all the 4 flip flops.
- In this type of register, there are no interconnections between the individual flip-flops since no serial shifting of the data is required. Data is given as input separately for each flip flop and in the same way, output also collected individually from each flip flop.



PIPO Shift Registers

- A Parallel in Parallel out (PIPO) shift register is used as a temporary storage device and like SISO Shift register it acts as a delay element.
- The shift register, which allows parallel input and produces parallel output is known as Parallel In – Parallel Out PIPOPIPO shift register. The **block diagram** of 3-bit PIPO shift register is shown in the following figure



PIPO Shift Registers

- This circuit consists of three D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop.
- All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.
- In this shift register, we can apply the **parallel inputs** to each D flip-flop by making Preset Enable to 1. We can apply the parallel inputs through preset or clear. These two are asynchronous inputs. That means, the flip-flops produce the corresponding outputs, based on the values of asynchronous inputs.
- In this case, the effect of outputs is independent of clock transition. So, we will get the **parallel outputs** from each D flip-flop.

Example

- Let us see the working of 3-bit PIPO shift register by applying the binary information “**011**” in parallel through preset inputs.
- Since the preset inputs are applied before positive edge of Clock, the initial status of the D flip-flops from leftmost to rightmost will be $Q_2Q_1Q_0=011$ $Q_2Q_1Q_0=011$. So, the binary information “**011**” is obtained in parallel at the outputs of D flip-flops before applying positive edge of clock.
- Therefore, the 3-bit PIPO shift register requires zero clock pulses in order to produce the valid output. Similarly, the **N-bit PIPO shift register** doesn’t require any clock pulse in order to shift ‘N’ bit information.

Example

- Let us see the working of 3-bit PIPO shift register by applying the binary information “**011**” in parallel through preset inputs.
- Since the preset inputs are applied before positive edge of Clock, the initial status of the D flip-flops from leftmost to rightmost will be $Q_2Q_1Q_0=011$ $Q_2Q_1Q_0=011$. So, the binary information “**011**” is obtained in parallel at the outputs of D flip-flops before applying positive edge of clock.
- Therefore, the 3-bit PIPO shift register requires zero clock pulses in order to produce the valid output. Similarly, the **N-bit PIPO shift register** doesn’t require any clock pulse in order to shift ‘N’ bit information.

Assignment

1. 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 Answer: c
 - a) 01110 Explanation: LSB bit is inverted and feed back to MSB:
 - b) 00001 01110->initial
 - c) 00101 10111->first clock pulse
 - d) 00110 01011->second
2. Assume that a 4-bit serial in/serial out shift register is initially clear. We wish to store the nibble 1100. What will be the 4-bit pattern after the second clock pulse? (Right-most bit first)
 - a) 1100 Answer: c Explanation: In Serial-In/Serial-Out shift register, data will be shifted one at a time with every clock pulse. Therefore,
Wait | Store
1100 | 0000
110 | 0000 1st clock
11 | 0000 2nd clock.
 - b) 0011
 - c) 0000
 - d) 1111

Assignment

1. A serial in/parallel out, 4-bit shift register initially contains all 1s. The data nibble 0111 is waiting to enter. After four clock pulses, the register contains _____
 - a) 0000 Answer: c
 - b) 1111 Explanation: In Serial-In/Parallel-Out shift register, data will be shifted all at a time with every clock pulse. Therefore,
 - c) 0111 Wait | Store
 - d) 1000

0111 | 0000
011 | 1000 1st clk
01 | 1100 2nd clk
0 | 1110 3rd clk
X | 1111 4th clk.
2. 1. Based on how binary information is entered or shifted out, shift registers are classified into _____ categories.
 - a) 2
 - b) 3
 - c) 4 Answer:c
 - d) 5

HDL FOR REGISTER

(Lecture-33)

Introduction

- Registers and counters can be described in Verilog at either the behavioural or the structural level.
- Behavioural modelling describes only the operations of the register, as prescribed by a function table, without a preconceived structure.
- 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 multilevel logic diagram. Both are useful.
- When a machine is complex, a hierarchical description creates a physical partition of the machine into simpler and more easily described units.

HDL Example (Universal Shift Register-Behavioral Model)

```
// Behavioral description of a 4-bit universal shift register
// Fig. 6.7 and Table 6.3
module Shift_Register_4_beh (          // V2001, 2005
    output reg      [3: 0]  A_par,        // Register output
    input   [3: 0]  I_par,        // Parallel input
    input   s1, s0,           // Select inputs
           MSB_in, LSB_in,       // Serial inputs
           CLK, Clear_b         // Clock and Clear
);
always @ (posedge CLK, negedge Clear_b) // V2001, 2005
if (Clear_b == 0) A_par <= 4'b0000;
else
  case ({s1, s0})
    2'b00: A_par <= A_par;           // No change
    2'b01: A_par <= {MSB_in, A_par[3: 1]}; // Shift right
    2'b10: A_par <= {A_par[2: 0], LSB_in}; // Shift left
    2'b11: A_par <= I_par;           // Parallel load of input
  endcase
endmodule
```

Variables of type **reg** retain their value until they are assigned a new value by an assignment statement. Consider the following alternative **case** statement for the shift register model:

```
case ({s1, s0})
  // 2'b00: A_par <= A_par;           // No change
  2'b01: A_par <= {MSB_in, A_par [3: 1]}; // Shift right
  2'b10: A_par <= {A_par [2: 0], LSB_in}; // Shift left
  2'b11: A_par <= I_par;             // Parallel load of input
endcase
```

HDL Example (Universal Shift Register-Structural Model)

```
// Structural description of a 4-bit universal shift register
module Shift_Register_4_str (
    output [3: 0] A_par,           // V2001, 2005
    input [3: 0] I_par,            // Parallel output
    input      s1, s0,             // Parallel input
    input      MSB_in, LSB_in, CLK, Clear_b
);
    // Mode select
    // Serial inputs, clock, clear
    // bus for mode control
    assign [1:0] select = {s1, s0};

    // Instantiate the four stages
    stage ST0 (A_par[0], A_par[1], LSB_in, I_par[0], A_par[0], select, CLK, Clear_b);
    stage ST1 (A_par[1], A_par[2], A_par[0], I_par[1], A_par[1], select, CLK, Clear_b);
    stage ST2 (A_par[2], A_par[3], A_par[1], I_par[2], A_par[2], select, CLK, Clear_b);
    stage ST3 (A_par[3], MSB_in, A_par[2], I_par[3], A_par[3], select, CLK, Clear_b);
endmodule

// One stage of shift register
module stage (i0, i1, i2, i3, Q, select, CLK, Clr_b);
    input      i0,                // circulation bit selection
              i1,                // data from left neighbor or serial input for shift-right
              i2,                // data from right neighbor or serial input for shift-left
              i3;                // data from parallel input
    output     Q;
    input [1: 0] select;          // stage mode control bus
    input      CLK, Clr_b;        // Clock, Clear for flip-flops
    wire       mux_out;

    // instantiate mux and flip-flop
    Mux_4_x_1 M0      (mux_out, i0, i1, i2, i3, select);
    D_flip_flop M1      (Q, mux_out, CLK, Clr_b);
endmodule

// 4x1 multiplexer      // behavioral model
module Mux_4_x_1 (mux_out, i0, i1, i2, i3, select);
    output     mux_out;
    input      i0, i1, i2, i3;
    input [1: 0] select;
    reg       mux_out;
    always @ (select, i0, i1, i2, i3)
        case (select)
            2'b00: mux_out = i0;
            2'b01: mux_out = i1;
            2'b10: mux_out = i2;
            2'b11: mux_out = i3;
        endcase
endmodule

// Behavioral model of D flip-flop
module D_flip_flop (Q, D, CLK, Clr_b);
    output     Q;
    input      D, CLK, Clr;
    reg       Q;
    always @ (posedge CLK, negedge Clr_b)
        if (!Clr) Q <= 1'b0; else Q <= D;
endmodule
```