

## Unit 1

# NUMBER SYSTEMS

Any number in one base system can be converted into another base system

Types

- 1) decimal to any base
- 2) Any base to decimal
- 3) Any base to Any base

# Number Systems

Decimal number:  $123.45 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$  •

Base  $b$  number:  $N = a_{q-1}b^{q-1} + \dots + a_0b^0 + \dots + a_{-p}b^{-p}$   
 $b > 1, \quad 0 \leq a_i \leq b-1$

Integer part:  $a_{q-1}a_{q-2} \dots a_0$

Fractional part:  $a_{-1}a_{-2} \dots a_{-p}$  •

Most significant digit:  $a_{q-1} \dots$

Least significant digit:  $a_{-p}$

Binary number ( $b=2$ ):  $1101.01 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}$

Representing number  $N$  in base  $b$ :  $(N)_b$  • • • • •

Complement of digit  $a$ :  $a' = (b-1)-a$

Decimal system: 9's complement of 3 =  $9-3 = 6$

Binary system: 1's complement of 1 =  $1-1 = 0$

# Representation of Integers

	<i>Base</i>				
	2	4	8	10	12
0000	0	0	0	0	0
0001	1	1	1	1	1
0010	2	2	2	2	2
0011	3	3	3	3	3
0100	10	4	4	4	4
0101	11	5	5	5	5
0110	12	6	6	6	6
0111	13	7	7	7	7
1000	20	10	8	8	8
1001	21	11	9	9	9
1010	22	12	10	$\alpha$	
1011	23	13	11	$\beta$	
1100	30	14	12	10	
1101	31	15	13	11	
1110	32	16	14	12	
1111	33	17	15	13	



# Base Conversions

**Example: Base 8 to base 10**

$$(432.2)_8 = 4 \cdot 8^2 + 3 \cdot 8^1 + 2 \cdot 8^0 + 2 \cdot 8^{-1} = (282.25)_{10}$$

**Example: Base 2 to base 10**

$$(1101.01)_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = (13.25)_{10}$$

**Base  $b_1$  to  $b_2$ , where  $b_1 > b_2$ :**

$$(N)_{b_1} = a_{q-1}b_2^{q-1} + a_{q-2}b_2^{q-2} + \cdots + a_1b_2^1 + a_0b_2^0$$

$$\frac{(N)_{b_1}}{b_2} = \underbrace{a_{q-1}b_2^{q-2} + a_{q-2}b_2^{q-3} + \cdots + a_1}_{Q_0} + \frac{a_0}{b_2}$$

$$\left(\frac{Q_0}{b_2}\right)_{b_1} = \underbrace{a_{q-1}b_2^{q-3} + a_{q-2}b_2^{q-4} + \cdots}_{Q_1} + \frac{a_1}{b_2}$$

# Conversion of Bases (Contd.)

**Example: Convert  $(548)_{10}$  to base 8**

$Q_i$	$r_i$
68	$4 = a_0$
8	$4 = a_1$
1	$0 = a_2$
	$1 = a_3$

Thus,  $(548)_{10} = (1044)_8$

**Example: Convert  $(345)_{10}$  to base 6**

$Q_i$	$r_i$
57	$3 = a_0$
9	$3 = a_1$
1	$3 = a_2$
	$1 = a_3$

5 Thus,  $(345)_{10} = (1333)_6$

# Conversions of fractional numbers

**Fractional number:**

$$(N)_{b_1} = a_{-1}b_2^{-1} + a_{-2}b_2^{-2} + \cdots + a_{-p}b_2^{-p}$$

$$b_2 \cdot (N)_{b_1} = a_{-1} + a_{-2}b_2^{-1} + \cdots + a_{-p}b_2^{-p+1}$$

**Example: Convert  $(0.3125)_{10}$  to base 8**

$$0.3125 \times 8 = 2.5000 \text{ hence } a_{-1} = 2$$

$$0.5000 \times 8 = 4.0000 \text{ hence } a_{-2} = 4$$

**Thus,  $(0.3125)_{10} = (0.24)_8$**

# Decimal to Binary

**Example: Convert  $(432.354)_{10}$  to binary**

$Q_i$	$r_i$	
216	$0 = a_0$	<b>0.354 <math>\times 2 = 0.708</math> hence <math>a_{-1} = 0</math></b>
108	$0 = a_1$	<b>0.708 <math>\times 2 = 1.416</math> hence <math>a_{-2} = 1</math></b>
54	$0 = a_2$	<b>0.416 <math>\times 2 = 0.832</math> hence <math>a_{-3} = 0</math></b>
27	$0 = a_3$	<b>0.832 <math>\times 2 = 1.664</math> hence <math>a_{-4} = 1</math></b>
13	$1 = a_4$	<b>0.664 <math>\times 2 = 1.328</math> hence <math>a_{-5} = 1</math></b>
6	$1 = a_5$	<b>0.328 <math>\times 2 = 0.656</math> hence <math>a_{-6} = 0</math></b>
3	$0 = a_6$	<b><math>\cdot</math> <math>a_{-7} = 1</math></b>
1	$1 = a_7$	<b>etc.</b>
	$1 = a_8$	

**Thus,  $(432.354)_{10} = (110110000.0101101...)_{2}$**

# Octal to Binary Conversion

**Example: Convert  $(123.4)_8$  to binary**

$$(123.4)_8 = (001\ 010\ 011.100)_2$$

**Example: Convert  $(1010110.0101)_2$  to octal**

$$(1010110.0101)_2 = (001\ 010\ 110.010\ 100)_2 = (126.24)_8$$

# Complements

- Complements are used in digital computers to simplify the subtraction operation and for logical manipulation
- They are two types of complements
  - 1) Diminished radix complement
$$(r^n - 1) - N$$
 {r is the base of number system}
  - 2) Radix Complement
$$(r^n - 1) - N + 1$$

## **(r-1)'s complement**

- If the base = 10
- The 9's complement of 546700 is
$$999999 - 546700 = 453299.$$
- If the base = 2
- The 1's complement of 1011000 is 0100111.

# **r's complement**

- the 10's complement of 012398 is 987602
- the 1's complement of 1101100 is 0010100



# Subtraction using complements

- Discard end carry for r's complement  
Using 10's complement subtract  $72532 - 3250$ .

$$M = 72532$$

$$10\text{'s complement of } N = \underline{+ 96750}$$

$$\text{Sum} = 169282$$

Discard end carry for 10's complement

Answer =

69282

## Subtraction using (r-1)'s complement

- $X - Y = 1010100 - 1000011$

$$X = 1010100$$

$$1's \text{ comp of } Y = + \underline{0111100}$$

$$\text{Sum} = 1\ 0010000$$

$$\text{Add End-around carry} = \underline{\hspace{2cm}} + 1$$

$$\textcircled{1} \ X - Y = 0010001$$

# Binary Codes

Decimal digit	$w_4w_3w_2w_1$											
	8	4	2	1	2	4	2	1	6	4	2	-3
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1	0	1	0	1
2	0	0	1	0	0	0	1	0	0	0	1	0
3	0	0	1	1	0	0	1	1	1	0	0	1
4	0	1	0	0	0	1	0	0	0	1	0	0
5	0	1	0	1	1	0	1	1	1	0	1	1
6	0	1	1	0	1	1	0	0	0	1	1	0
7	0	1	1	1	1	1	0	1	1	1	0	1
8	1	0	0	0	1	1	1	0	1	0	1	0
9	1	0	0	1	1	1	1	1	1	1	1	1



**BCD**



**Self-complementing Codes**

**Self-complementing code:** Code word of 9's complement of  $N$  obtained by interchanging 1's and 0's in the code word of  $N$

# Non Weighted Codes

<i>Decimal digit</i>	<i>Excess-3</i>				<i>Cyclic</i>			
0	0	0	1	1	0	0	0	0
1	0	1	0	0	0	0	0	1
2	0	1	0	1	0	0	1	1
3	0	1	1	0	0	0	1	0
4	0	1	1	1	0	1	1	0
5	1	0	0	0	1	1	1	0
6	1	0	0	1	1	0	1	0
7	1	0	1	0	1	0	0	0
8	1	0	1	1	1	1	0	0
9	1	1	0	0	0	1	0	0

**Add 3 to  
BCD**

**Successive code words  
differ in only one digit**

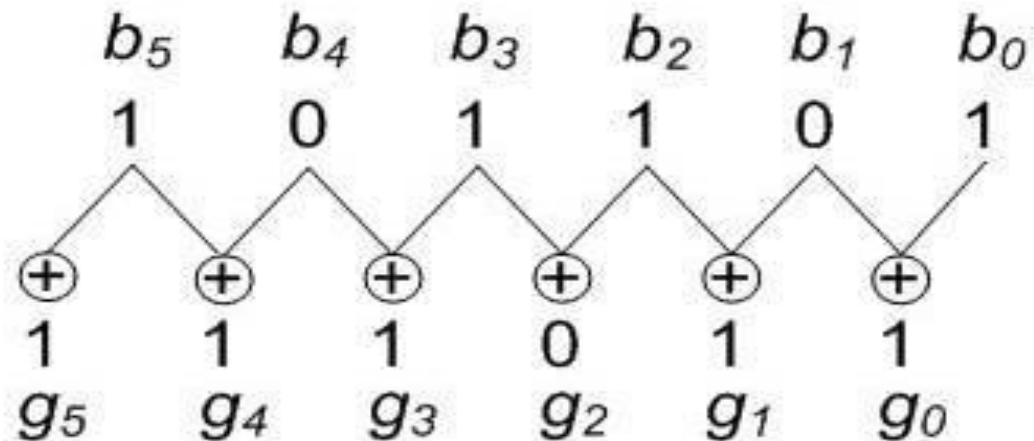
# Gray Code(Unit distance Code)

<i>Decimal number</i>	<i>Gray</i>				<i>Binary</i>			
	<i>g<sub>3</sub></i>	<i>g<sub>2</sub></i>	<i>g<sub>1</sub></i>	<i>g<sub>0</sub></i>	<i>b<sub>3</sub></i>	<i>b<sub>2</sub></i>	<i>b<sub>1</sub></i>	<i>b<sub>0</sub></i>
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	1	0	0	1	0
3	0	0	1	0	0	0	1	1
4	0	1	1	0	0	1	0	0
5	0	1	1	1	0	1	0	1
6	0	1	0	1	0	1	1	0
7	0	1	0	0	0	1	1	1
8	1	1	0	0	1	0	0	0
9	1	1	0	1	1	0	0	1
10	1	1	1	1	1	0	1	0
11	1	1	1	0	1	0	1	1
12	1	0	1	0	1	1	0	0
13	1	0	1	1	1	1	0	1
14	1	0	0	1	1	1	1	0
15	1	0	0	0	1	1	1	1

# Binary to Gray Conversion

**Example:**

**Binary:**



**Gray:**

**Gray-to-binary:**

- $b_i = g_i$  if no. of 1's preceding  $g_i$  is even
- $b_i = g_i'$  if no. of 1's preceding  $g_i$  is odd

# Mirror Image Representation in Gray Code

00	0	00	0	000
01	0	01	0	001
11	0	11	0	011
10	0	10	0	010
	1	10	0	110
	1	11	0	111
	1	01	0	101
	1	00	0	100
			1	100
			1	101
			1	111
			1	110
			1	010
			1	011
			1	001
			1	000

# Error Detection and Correction

- No communication channel or storage device is completely error-free
- As the number of bits per area or the transmission rate increases, more errors occur.
- Impossible to detect or correct 100% of the errors



# Types of Error Detection

- 3 Types of Error Detection/Correction Methods
  - Cyclic Redundancy Check (CRC)
  - Hamming Codes
  - Reed-Solomon (RS)

$$\begin{array}{ccccc} 10011001011 & = & 1001100 & + & 1011 \\ \wedge & & \wedge & & \wedge \\ \text{Code word} & & \text{information} & & \text{error-checking bits/} \\ & & \text{bits} & & \text{parity bits/} \\ & & & & \text{syndrome/} \\ & & & & \text{redundant bits} \end{array}$$

# Hamming Codes

1. One of the most effective codes for error-recovery
2. Used in situations where random errors are likely to occur
3. Error detection and correction increases in proportion to the number of **parity bits (error-checking bits)** added to the end of the information bits

**code word** = information bits + parity bits

**Hamming distance:** the number of bit positions in which two code words differ.

```
10001001
10110001
  * * *
```

**Minimum Hamming distance or  $D(\min)$**  : determines its error detecting and correcting capability.

4. Hamming codes can always detect  $D(\min) - 1$  errors, but can only correct half of those errors.

# Hamming Codes

EX. Data

Parity

Code

Bits

Bit

Word

00

0

000

01

1

011

10

1

101

11

0

110

000\* 100

001 101\*

010 110\*

011\* 111

- Single parity bit can only detect error, not correct it
- Error-correcting codes require more than a single parity bit

EX.    0 0 0 0 0  
         0 1 0 1 1  
         1 0 1 1 0  
         1 1 1 0 1

Minimum Hamming distance = 3

Can detect up to 2 errors and correct 1 error

# Cyclic Redundancy Check

1. Let the information byte  $F = 1001011$
2. The sender and receiver agree on an arbitrary binary pattern  $P$ . Let  $P = 1011$ .
3. Shift  $F$  to the left by 1 less than the number of bits in  $P$ . Now,  $F = 1001011000$ .
4. Let  $F$  be the dividend and  $P$  be the divisor. Perform "modulo 2 division".
5. After performing the division, we ignore the quotient. We got 100 for the remainder, which becomes the actual CRC checksum.
6. Add the remainder to  $F$ , giving the message  $M$ :

$$1001011 + 100 = 1001011100 = M$$

# Calculating and Using CRCs

7. M is decoded and checked by the message receiver using the reverse process.

$$\begin{array}{r} \underline{\hspace{2cm} 1010100} \\ 1011 \mid 1001011100 \\ \quad \underline{1011} \\ \quad 001001 \\ \quad 1001 \\ \quad \underline{0010} \\ \quad 001011 \\ \quad \quad \underline{1011} \\ \quad \quad 0000 \end{array}$$

← Remainder

# Canonical and Standard Forms

- We need to consider formal techniques for the simplification of Boolean functions.
  - Identical functions will have exactly the same canonical form.
  - Minterms and Maxterms
  - Sum-of-Minterms and Product-of- Maxterms
  - Product and Sum terms
  - Sum-of-Products (SOP) and Product-of-Sums (POS)

# Definitions

- *Literal*: A variable or its complement
- *Product term*: literals connected by •
- *Sum term*: literals connected by +
- *Minterm*: a product term in which all the variables appear exactly once, either complemented or uncomplemented
- *Maxterm*: a sum term in which all the variables appear exactly once, either complemented or uncomplemented



# Truth Table notation for Minterms and Maxterms

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

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

# Canonical Forms (Unique)

- Any Boolean function  $F()$  can be expressed as a *unique* **sum** of **minterms** and a unique **product** of **maxterms** (under a fixed variable ordering).
- In other words, every function  $F()$  has two canonical forms:
  - Canonical Sum-Of-Products (sum of minterms)
  - Canonical Product-Of-Sums (product of maxterms)

# Canonical Forms (cont.)

- Canonical Sum-Of-Products:  
The minterms included are those  $m_j$  such that  $F(\ ) = 1$  in row  $j$  of the truth table for  $F(\ )$ .
- Canonical Product-Of-Sums:  
The maxterms included are those  $M_j$  such that  $F(\ ) = 0$  in row  $j$  of the truth table for  $F(\ )$ .

# Example

- Truth table for  $f_1(a,b,c)$  at right
- The canonical sum-of-products form for  $f_1$  is
$$f_1(a,b,c) = m_1 + m_2 + m_4 + m_6$$
$$= \underline{a'b'c} + \underline{a'bc'} + \underline{ab'c'} + \underline{abc'}$$
- The canonical product-of-sums form for  $f_1$  is
$$f_1(a,b,c) = M_0 \cdot M_3 \cdot M_5 \cdot M_7$$
$$= (\underline{a+b+c}) \cdot (\underline{a+b'+c'}) \cdot$$
$$(\underline{a'+b+c'}) \cdot (\underline{a'+b'+c'}).$$
- Observe that:  $\underline{m_j} = \underline{M_j'}$

a	b	c	$f_1$
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	1
1	1	1	0

# Conversion Between Canonical Forms

- Replace  $\Sigma$  with  $\Pi$  (or *vice versa*) and replace those  $j$ 's that appeared in the original form with those that do not.

- Example:

$$\begin{aligned}f_1(a,b,c) &= a'b'c + a'bc' + ab'c' + abc' \\&= m_1 + m_2 + m_4 + m_6 \\&= \Sigma(1,2,4,6) \\&= \Pi(0,3,5,7) \\&= \\&= (a+b+c) \cdot (a+b'+c') \cdot (a'+b+c') \cdot (a'+b'+c')\end{aligned}$$

# Conversion of SOP from standard to canonical form

- Expand *non-canonical* terms by inserting equivalent of 1 in each missing variable  $x$ :  
 $(x + x') = 1$
- Remove duplicate minterms
- $f_1(a,b,c) = a'b'c + bc' + ac'$   
 $= a'b'c + (a+a')bc' + a(b+b')c'$   
 $= a'b'c + abc' + a'bc' + abc' +$   
 $ab'c'$   
 $= a'b'c + abc' + a'bc + ab'c'$

# Conversion of POS from standard to canonical form

- Expand noncanonical terms by adding 0 in terms of missing variables (e.g.,  $xx' = 0$ ) and using the distributive law
- Remove duplicate maxterms
- $$\begin{aligned}f_1(a,b,c) &= (a+b+c) \bullet (b'+c') \bullet (a'+c') \\&= (a+b+c) \bullet (\textcolor{red}{a}a' + b' + c') \bullet (a' + \textcolor{red}{b}b' + c') \\&= (a+b+c) \bullet (a+b'+c') \bullet (\textcolor{orange}{a'} + \textcolor{orange}{b'} + \textcolor{orange}{c'}) \bullet \\&\quad (a' + b + c') \bullet (\textcolor{orange}{a'} + \textcolor{orange}{b'} + \textcolor{orange}{c'}) \\&= \\&= (a+b+c) \bullet (a+b'+c') \bullet (a'+b'+c') \bullet (a'+b+c')\end{aligned}$$

# **Boolean Algebra and Basic Gates**



# LOGIC GATES

**Formal logic:** In formal logic, a statement (proposition) is a declarative sentence that is either

true(1) or false (0).

It is easier to communicate with computers using formal logic.

- **Boolean variable:** Takes only two values – either

true (1) or false (0).

They are used as basic units of formal logic.

# Boolean function and logic diagram




- **Boolean function:** Mapping from Boolean variables to a Boolean value.
- **Truth table:**
  - Represents relationship between a Boolean function and its binary variables.
  - It enumerates all possible combinations of arguments and the corresponding function values.

# Boolean function and logic diagram

- **Boolean algebra:** Deals with binary variables and logic operations operating on those variables.
- **Logic diagram:** Composed of graphic symbols for logic gates. A simple circuit sketch that represents inputs and outputs of Boolean functions.

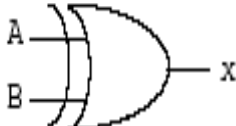
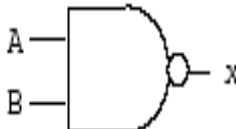
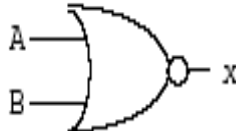
# Gates

- Refer to the hardware to implement Boolean operators.
- The most basic gates are

Name	Graphic symbol	Algebraic function	Truth table															
Inverter	A  x	$x = A'$	<table><tr><th>A</th><th>x</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	x	0	1	1	0									
A	x																	
0	1																	
1	0																	
AND	A  B x	$x = AB$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table> <p>True if both are true.</p>	A	B	x	0	0	0	0	1	0	1	0	0	1	1	1
A	B	x																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR	A  B x	$x = A + B$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table> <p>True if either one is true.</p>	A	B	x	0	0	0	0	1	1	1	0	1	1	1	1
A	B	x																
0	0	0																
0	1	1																
1	0	1																
1	1	1																

# Boolean function and truth table

- Other common gates include:

Name	Graphic symbol	Algebraic function	Truth table															
Exclusive-OR (XOR)		$x = A \oplus B$ $= A'B + AB'$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	x	0	0	0	0	1	1	1	0	1	1	1	0
A	B	x																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
NAND		$x = (AB)'$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	x	0	0	1	0	1	1	1	0	1	1	1	0
A	B	x																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$x = A + B$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	x	0	0	1	0	1	0	1	0	0	1	1	0
A	B	x																
0	0	1																
0	1	0																
1	0	0																
1	1	0																

Parity check: True if only one is true.

Inversion of AND.

Inversion of OR.

# BASIC IDENTITIES OF BOOLEAN ALGEBRA

- ***Postulate 1 (Definition)***: A Boolean algebra is a closed algebraic system containing a set  $K$  of two or more elements and the two operators  $\cdot$  and  $+$  which refer to logical AND and logical OR

## Basic Identities of Boolean Algebra (Existence of 1 and 0 element)

$$(1) x + 0 = x$$

$$(2) x \cdot 0 = 0$$

$$(3) x + 1 = 1$$

$$(4) x \cdot 1 = x$$

$$(5) x + x = x$$

$$(6) x \cdot x = x$$

$$(7) x + x' = 1$$

$$(8) x \cdot x' = 0$$

## Basic Identities of Boolean Algebra (Commutatively):

$$(9) \ x + y = y + x$$

$$(10) \ xy = yx$$

$$(11) \ x + (y + z) = (x + y) + z$$

$$(12) \ x(yz) = (xy)z$$

$$(13) \ x(y + z) = xy + xz$$

$$(14) \ x + yz = (x + y)(x + z)$$

$$(15) \ (x + y)' = x'y'$$

$$(16) \ (xy)' = x' + y'$$

$$(17) \ (x')' = x$$



# Function Minimization using Boolean Algebra

- **Examples:**

$$(a) \ a + ab = a(1+b)=a$$

$$(b) \ a(a + b) = a.a + ab = a + ab = a(1+b) = a.$$

$$(c) \ a + a'b = (a + a')(a + b) = 1(a + b) = a + b$$

$$(d) \ a(a' + b) = a.a' + ab = 0 + ab = ab$$

# *DeMorgan's Theorem*

$$(a) \ (a + b)' = a'b'$$

$$(b) \ (ab)' = a' + b'$$

Generalized DeMorgan's Theorem

$$(a) \ (a + b + \dots z)' = a'b' \dots z'$$

$$(b) \ (a.b \dots z)' = a' + b' + \dots z'$$

# *DeMorgan's Theorem*

- $F = ab + c'd'$
- $F' = ??$
  
- $F = ab + c'd' + b'd$
- $F' = ??$

## More DeMorgan's example

*Show that:*  $(a(b + z(x + a')))' = a' + b' (z' + x')$

$$\begin{aligned}(a(b + z(x + a')))' &= a' + (b + z(x + a'))' \\&= a' + b' (z(x + a'))' \\&= a' + b' (z' + (x + a'))' \\&= a' + b' (z' + x'(a'))' \\&= a' + b' (z' + x'a) \\&= a' + b' z' + b'x'a \\&= (a' + b'x'a) + b' z' \\&= (a' + b'x')(a + a') + b' z' \\&= a' + b'x' + b' z' \\&= a' + b' (z' + x')\end{aligned}$$

# Two Level implantation

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

# NAND-AND

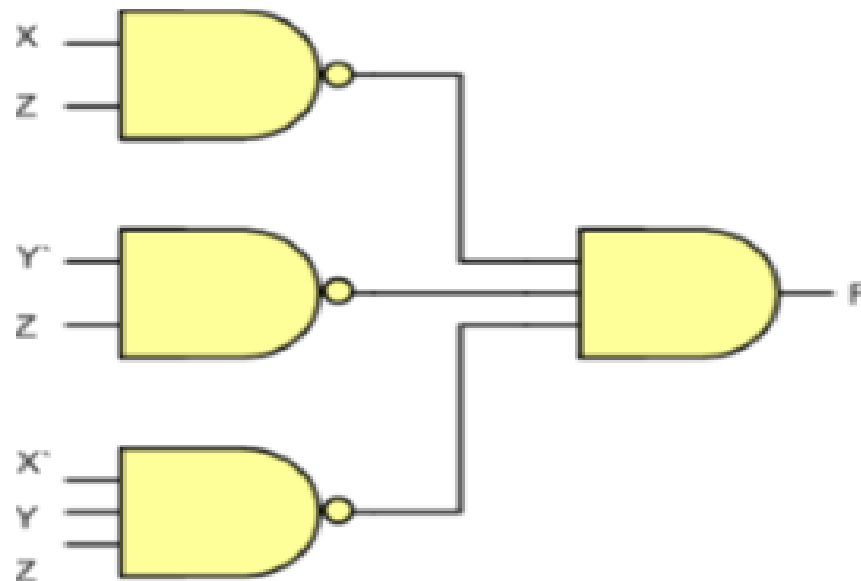
## AND-NOR functions:

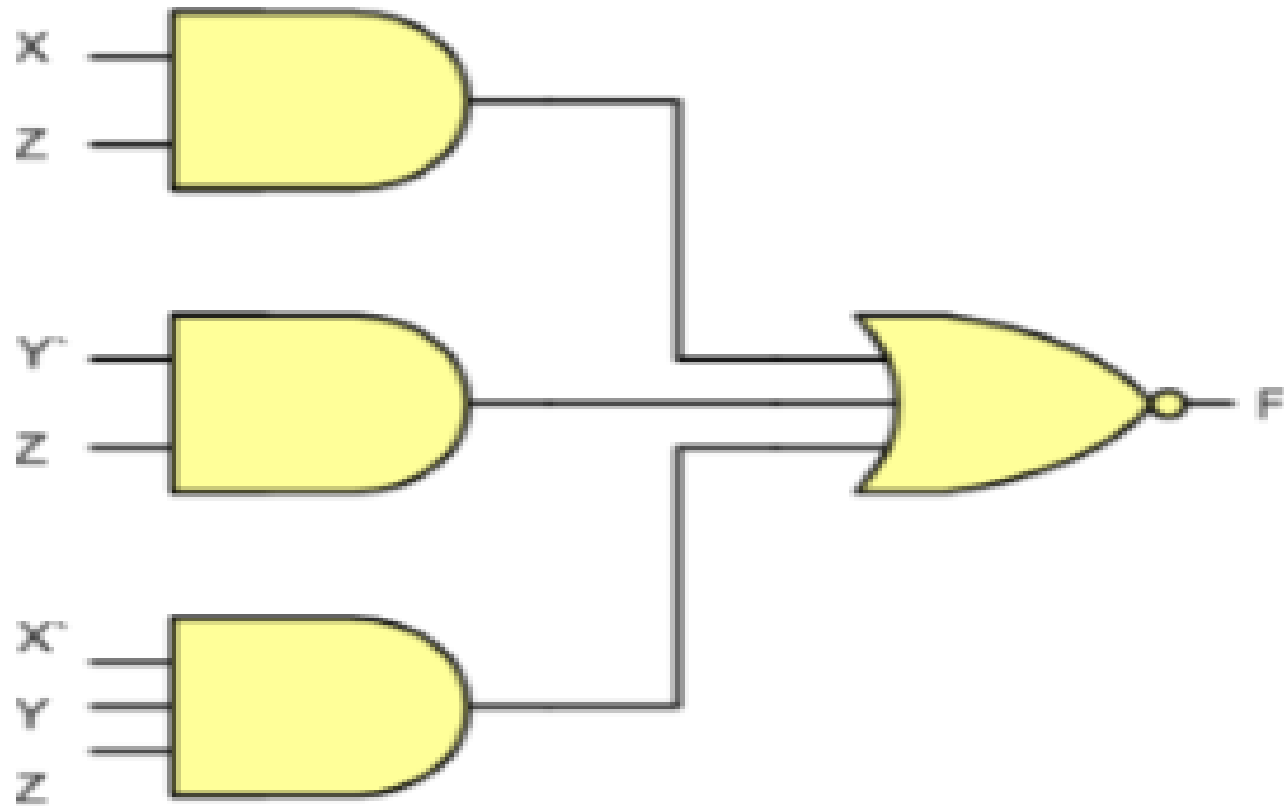
**Example 3:** Implement the following function

$$F = \overline{XZ} + \overline{YZ} + \overline{XYZ} \text{ or}$$

$$\overline{F} = XZ + YZ + XYZ$$

Since  $F'$  is in SOP form, it can be implemented by using NAND-NAND circuit. By complementing the output we can get  $F$ , or by using *NAND-AND* circuit as shown in the figure.





**It can also be implemented using AND-NOR circuit as it is equivalent to NAND- AND circuit**

## OR-NAND functions:

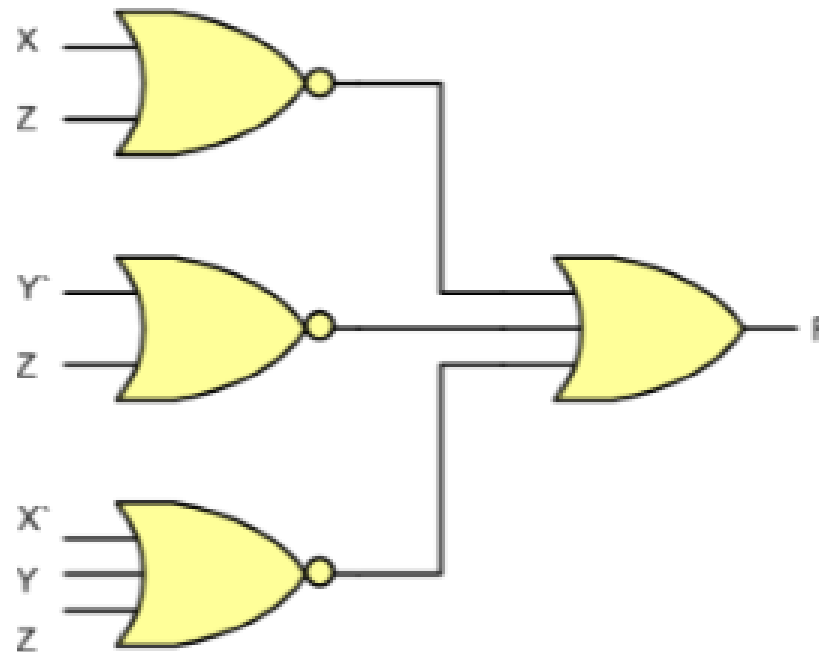
**Example 4:** Implement the following function

$$F = (X + Z) \cdot (\bar{Y} + Z) \cdot (\bar{X} + Y + Z) \text{ or}$$

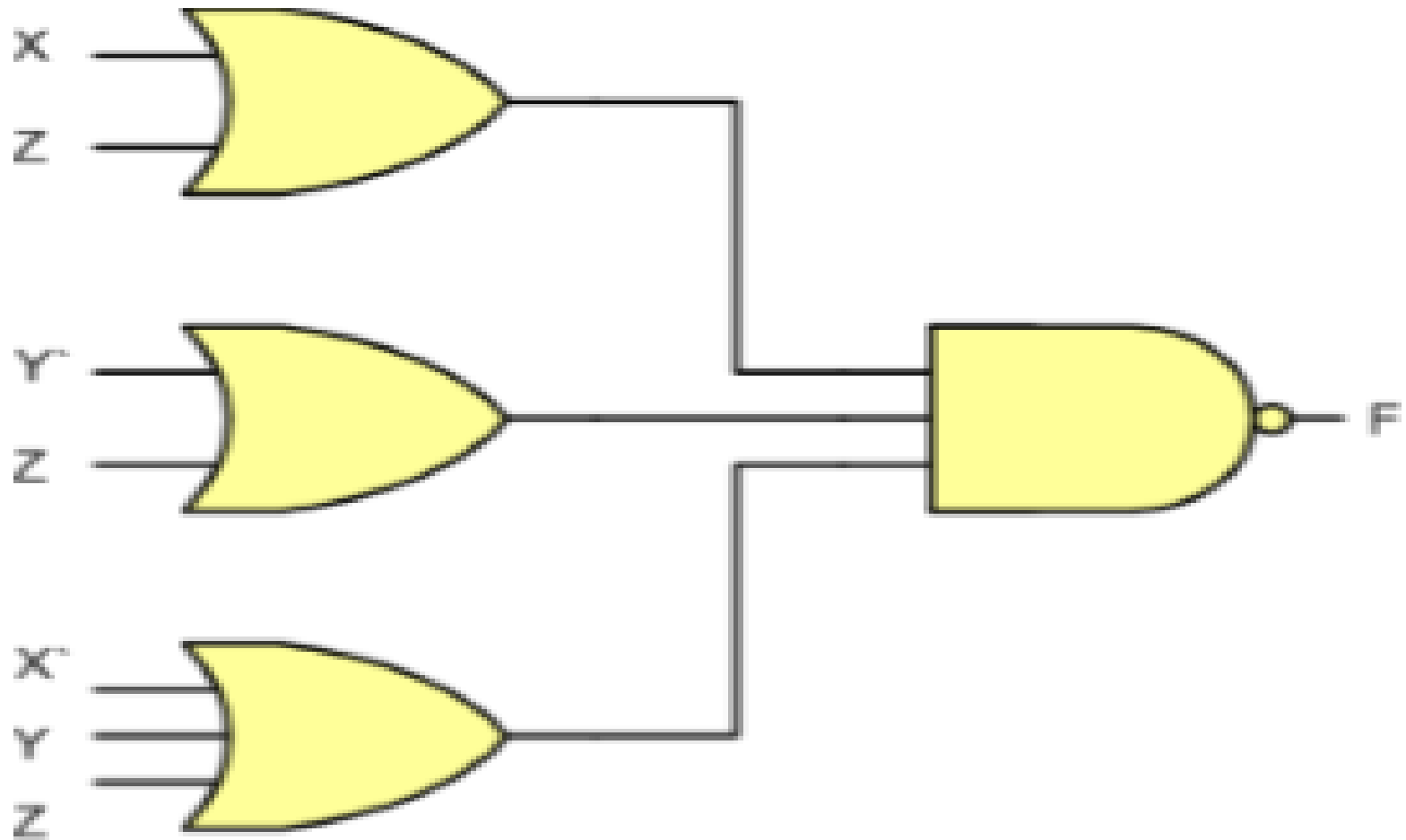
$$\bar{F} = (X + Z)(\bar{Y} + Z)(\bar{X} + Y + Z)$$

Since  $\bar{F}$  is in POS form, it can be implemented by using NOR-NOR circuit.

By complementing the output we can get  $F$ , or by using **NOR-OR** circuit as shown in the figure.





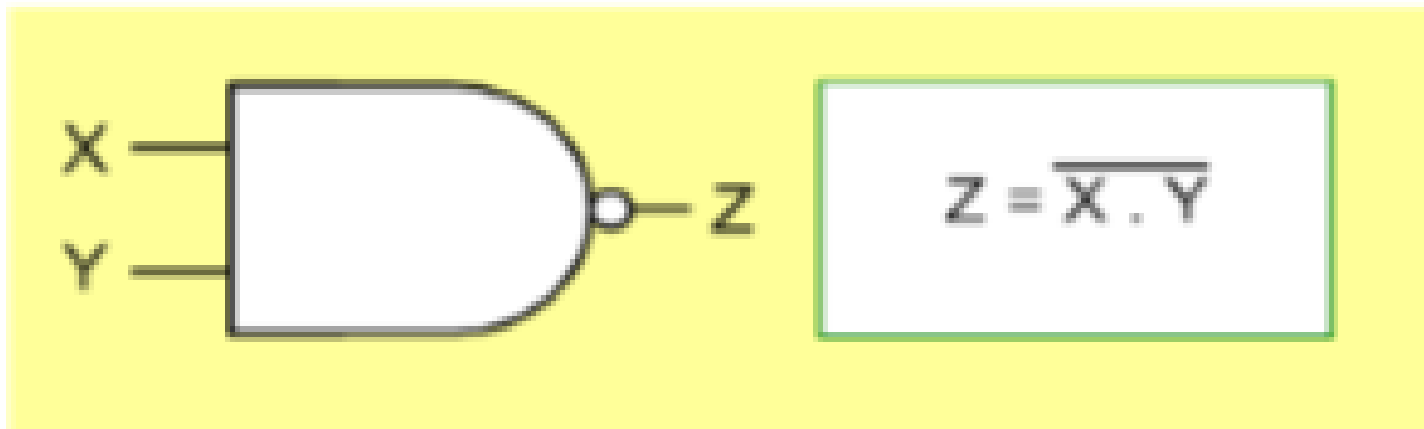


**It can also be implemented using OR-NAND circuit as it is equivalent to NOR-OR circuit**

# Universal Gates

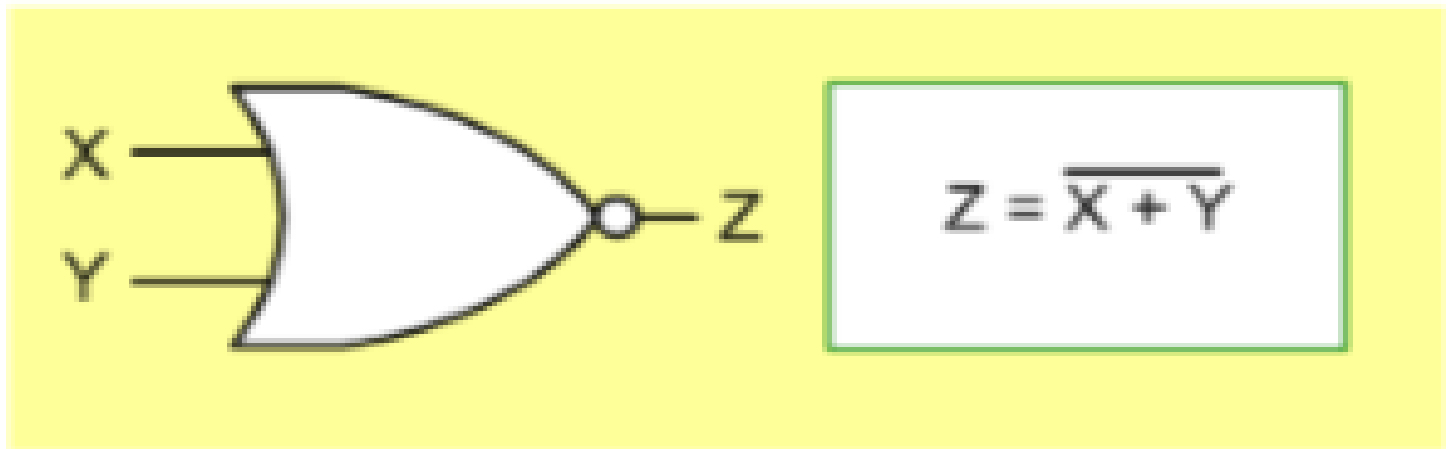
- The objectives of this lesson are to learn about:
  1. Universal gates - NAND and NOR.
  2. How to implement NOT, AND, and OR gate using NAND gates only.
  3. How to implement NOT, AND, and OR gate using NOR gates only.
  4. Equivalent gates.

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



**NAND GATE**

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



**NOR GATE**

1. All NAND input pins connect to the input signal **A** gives an output **A'**.



2. One NAND input pin is connected to the input signal **A** while all other input pins are connected to logic **1**. The output will be **A'**.



### Implementing AND Using only NAND Gates

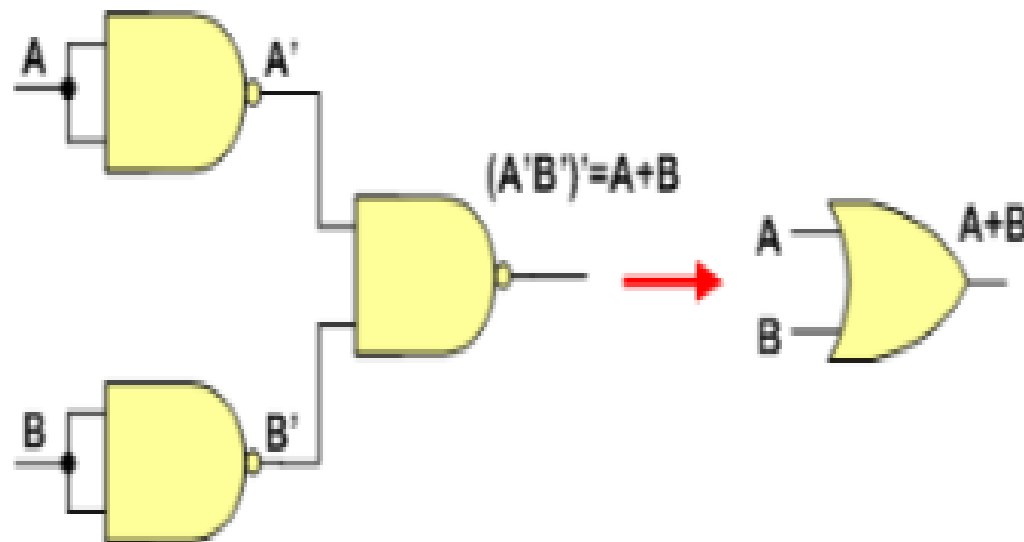
An **AND** gate can be replaced by NAND gates as shown in the figure (The AND is replaced by a NAND gate with its output complemented by a NAND gate inverter).



# NAND AS A UNIVERSAL GATE

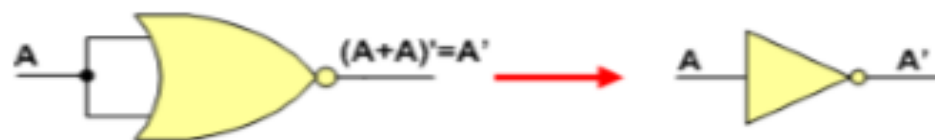
## Implementing OR Using only NAND Gates

An OR gate can be replaced by NAND gates as shown in the figure (The OR gate is replaced by a NAND gate with all its inputs complemented by NAND gate inverters).

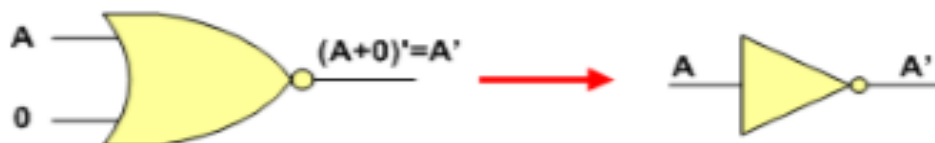


Thus, the NAND gate is a universal gate since it can implement the AND, OR and NOT functions.

1. All NOR input pins connect to the input signal **A** gives an output **A'**.



2. One NOR input pin is connected to the input signal **A** while all other input pins are connected to logic **0**. The output will be **A'**.



### Implementing OR Using only NOR Gates

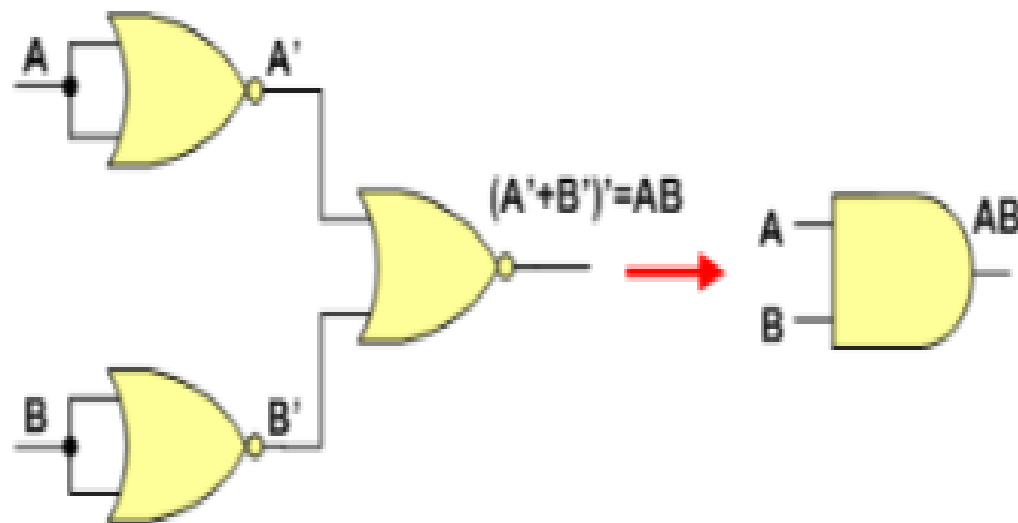
An **OR gate** can be replaced by NOR gates as shown in the figure (The OR is replaced by a NOR gate with its output complemented by a NOR gate inverter)



# NOR AS A UNIVERSAL GATE

## Implementing AND Using only NOR Gates

An AND gate can be replaced by NOR gates as shown in the figure (The AND gate is replaced by a NOR gate with all its inputs complemented by NOR gate inverters)



Thus, the NOR gate is a universal gate since it can implement the AND, OR and NOT functions.



## UNIT 2

# Gate Level Minimization of Logic Circuits

# Combinational Logic Design

- A process with 5 steps
  - Specification
  - Formulation
  - Optimization
  - Technology mapping
  - Verification
- 1<sup>st</sup> three steps and last best illustrated by example

# Functional Blocks

- Fundamental circuits that are the base building blocks of most larger digital circuits
- They are reusable and are common to many systems.
- Examples of functional logic circuits
  - Decoders
  - Encoders
  - Code converters
  - Multiplexers

# Where they are used

- Multiplexers
  - Selectors for routing data to the processor, memory, I/O
  - Multiplexers route the data to the correct bus or port.
- Decoders
  - are used for selecting things like a bank of memory and then the address within the bank. This is also the function needed to 'decode' the instruction to determine the operation to perform.
- Encoders
  - are used in various components such as keyboards.

# BCD-to-Excess-3 Code converter

- BCD is a code for the decimal digits 0-9
- Excess-3 is also a code for the decimal digits

Decimal Digit	Input BCD	Output Excess-3
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0

# Specification of BCD-to-Excess3

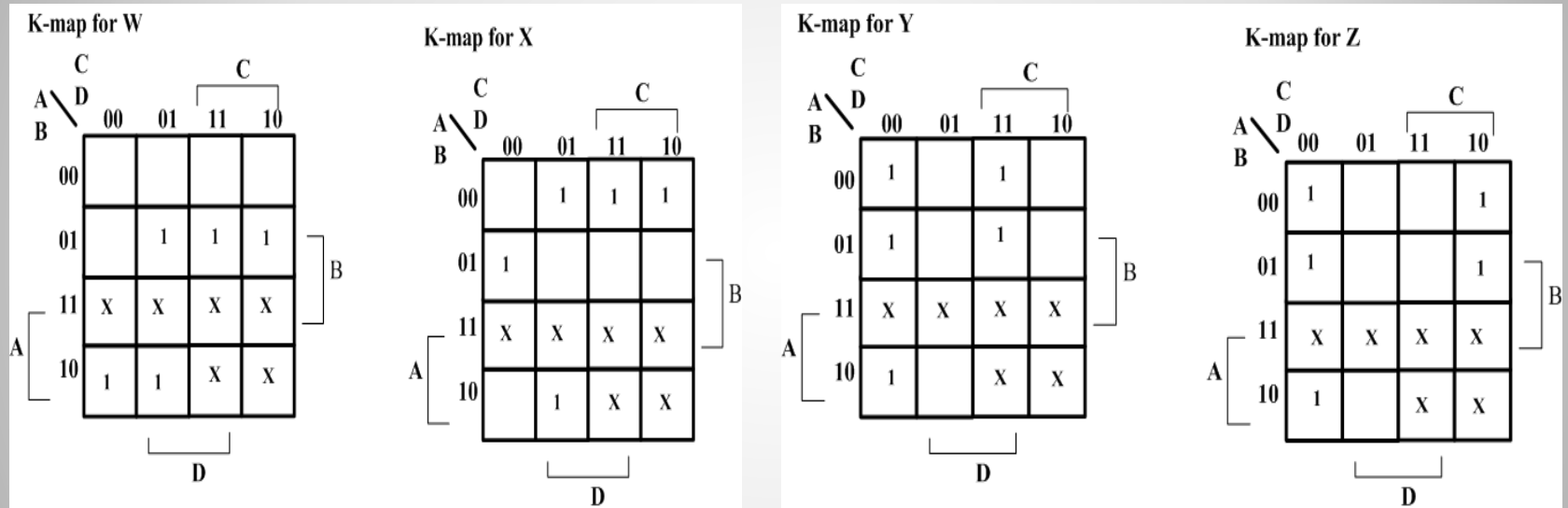
- Inputs: a BCD input, A,B,C,D with A as the most significant bit and D as the least significant bit.
- Outputs: an Excess-3 output W,X,Y,Z that corresponds to the BCD input.
- Internal operation – circuit to do the conversion in combinational logic.

## Formulation of BCD-to-Excess-3

- Excess-3 code is easily formed by adding a binary 3 to the binary or BCD for the digit.
- There are 16 possible inputs for both BCD and Excess-3.
- It can be assumed that only valid BCD inputs will appear so the six combinations not used can be treated as don't cares.

# Optimization – BCD-to-Excess-3

- Lay out K-maps for each output, W X Y Z



- A step in the digital circuit design process.



- Where are the minterms located on a K-Map?

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

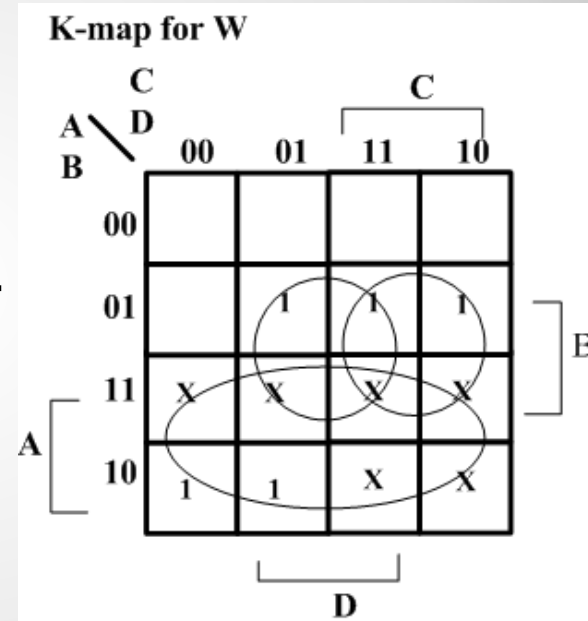
**Placing 1 on K-maps**

- $W(A,B,C,D) = \Sigma m(5,6,7,8,9)$   
 $+d(10,11,12,13,14,15)$
- $X(A,B,C,D) = \Sigma m(1,2,3,4,9)$   
 $+d(10,11,12,13,14,15)$
- $Y(A,B,C,D) = \Sigma m(0,3,4,7,8)$   
 $+d(10,11,12,13,14,15)$
- $Z(A,B,C,D) = \Sigma m(0,2,4,6,8)$   
 $+d(10,11,12,13,14,15)$

**Expressions for W X Y Z**

- W minimization

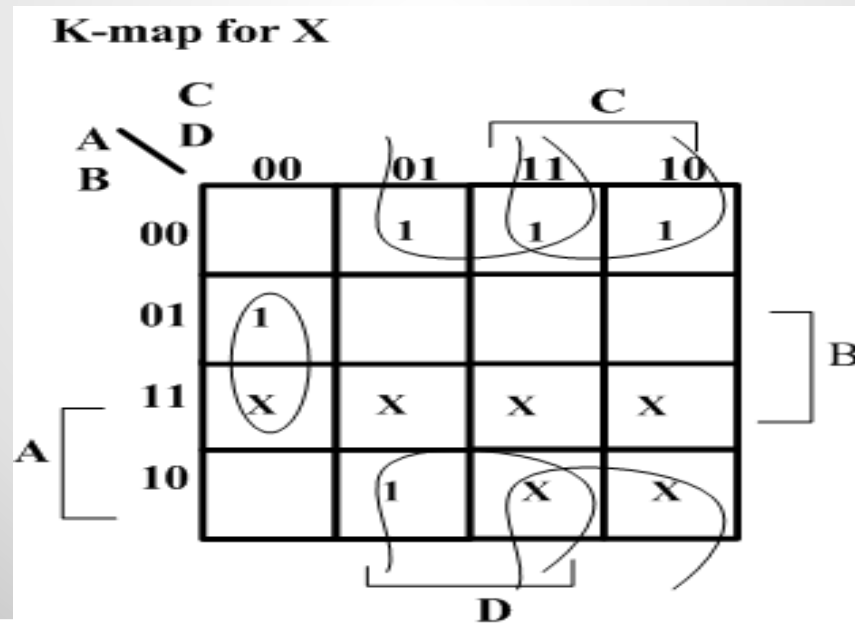
- Find  $W = A + BC +$



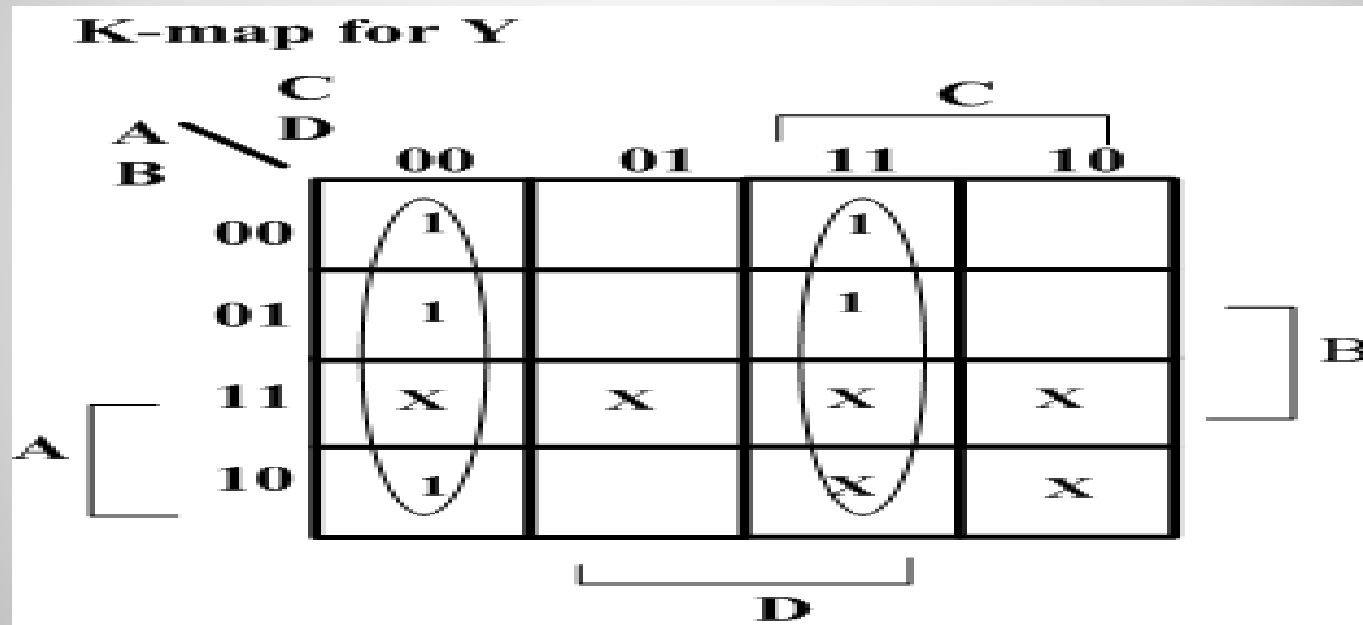
**Minimize K-Maps**

# Minimize K-Maps

- X minimization
- Find  $X = BC'D' + B'C + B'D$



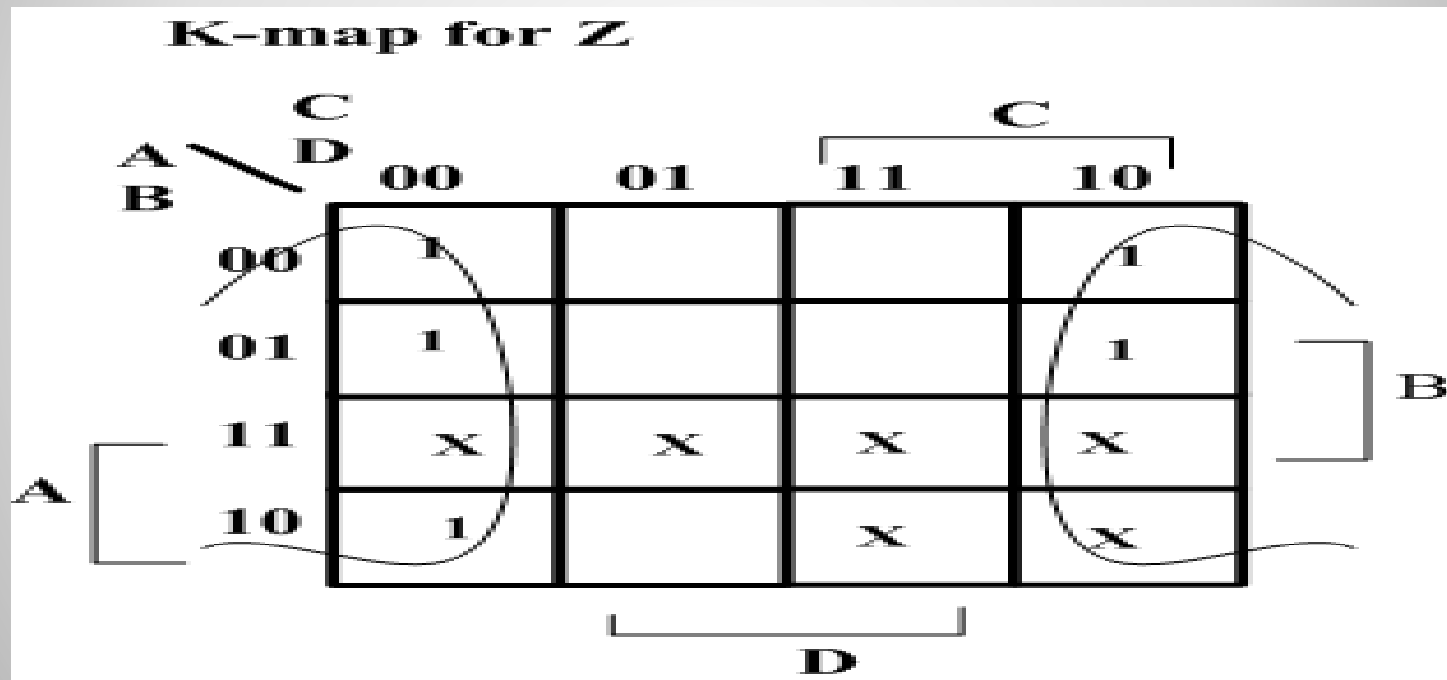
- Y minimization
- Find  $Y = CD + C'D'$



**Minimize K-Maps**

- Z minimization

- Find  $Z = D'$



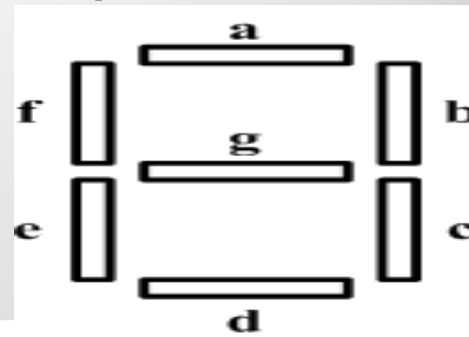
**Minimize K-Maps**

# BCD-to-Seven-Segment Decoder

- Specification
  - Digital readouts on many digital products often use LED seven-segment displays.
  - Each digit is created by lighting the appropriate segments. The segments are labeled a,b,c,d,e,f,g
  - The decoder takes a BCD input and outputs the correct code for the seven-segment display.

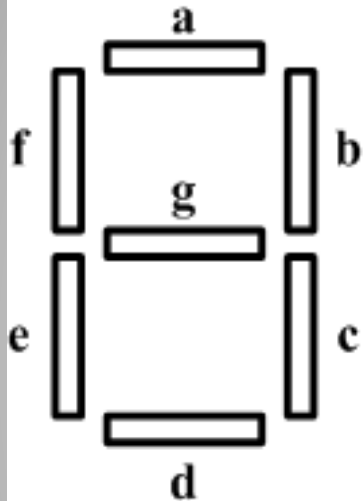
# Specification

- Input: A 4-bit binary value that is a BCD coded input.
- Outputs: 7 bits, a through g for each of the segments of the display.
- Operation: Decode the input to activate the correct segments.





- Construct a truth table



Decimal Digit	Input BCD	Seven-Segment Decoder Outputs a b c d e f g						
0	0 0 0 0	1	1	1	1	1	1	0
1	0 0 0 1	0	1	1	0	0	0	0
2	0 0 1 0	1	1	0	1	1	0	1
3	0 0 1 1	1	1	1	1	0	0	1
4	0 1 0 0	1	0	1	1	0	1	1
5	0 1 0 1	1	0	1	1	0	1	1
6	0 1 1 0	1	0	1	1	1	1	1
7	0 1 1 1	1	1	1	0	0	0	0
8	1 0 0 0	1	1	1	1	1	1	1
9	1 0 0 1	1	1	1	1	0	1	1
All other inputs		0	0	0	0	0	0	0

# Formulation

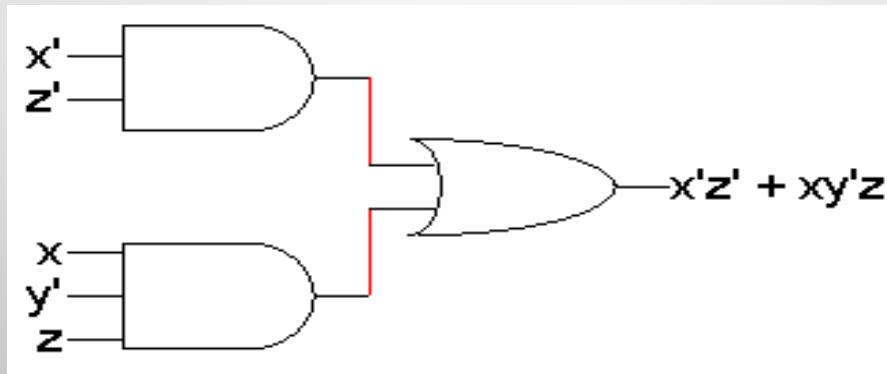
# Optimization

- Create a K-map for each output and get
  - $A = A'C + A'BD + B'C'D' + AB'C'$
  - $B = A'B' + A'C'D' + A'CD + AB'C'$
  - $C = A'B + A'D + B'C'D' + AB'C'$
  - $D = A'CD' + A'B'C + B'C'D' + AB'C' + A'BC'D$
  - $E = A'CD' + B'C'D'$
  - $F = A'BC' + A'C'D' + A'BD' + AB'C'$
  - $G = A'CD' + A'B'C + A'BC' + AB'C'$

# Karnaugh Maps for Simplification

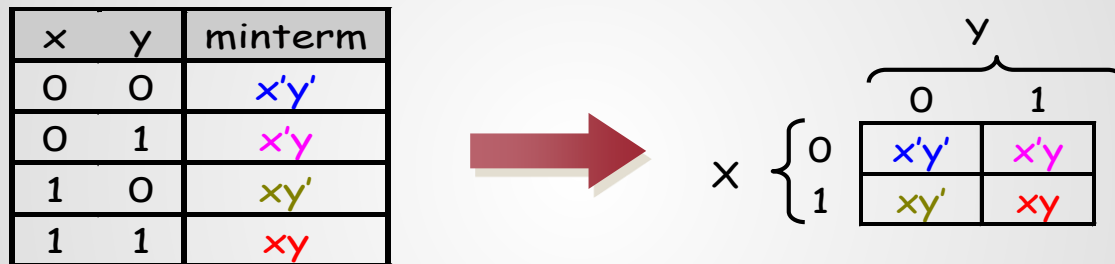
# Karnaugh Maps

- Boolean algebra helps us simplify expressions and circuits
- Karnaugh Map: A graphical technique for simplifying a Boolean expression into either form:
  - minimal sum of products (MSP)
  - minimal product of sums (MPS)
- Goal of the simplification.
  - There are a minimal number of product/sum terms
  - Each term has a minimal number of literals
- Circuit-wise, this leads to a *minimal* two-level implementation

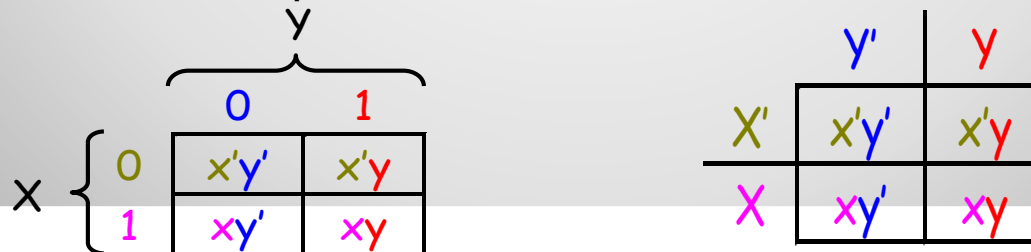


# Re-arranging the Truth Table

- A two-variable function has four possible minterms. We can re-arrange these minterms into a **Karnaugh map**



- Now we can easily see which minterms contain common literals
  - Minterms on the left and right sides contain  $y'$  and  $y$  respectively
  - Minterms in the top and bottom rows contain  $x'$  and  $x$  respectively



# Karnaugh Map Simplifications

- Imagine a two-variable sum of minterms:

$$x'y' + x'y$$

		y
	$x'y'$	$x'y$
x	$xy'$	$xy$

- Both of these minterms appear in the top row of a Karnaugh map, which means that they both contain the literal  $x'$

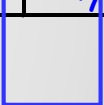
$$\begin{aligned}
 x'y' + x'y &= x'(y' + y) && [ \text{Distributive} ] \\
 &= x' \cdot 1 && [ y + y' = 1 ] \\
 &= x' && [ x \cdot 1 = x ]
 \end{aligned}$$

- What happens if you simplify this expression using Boolean algebra?

# More Two-Variable Examples

- Another example expression is  $x'y + xy$ 
  - Both minterms appear in the right side, where  $y$  is uncomplemented
  - Thus, we can reduce  $x'y + xy$  to just  $y$

		y
	$x'y'$	$x'y$
x	$xy'$	$xy$



- How about  $x'y' + x'y + xy$ ?
  - We have  $x'y' + x'y$  in the top row, corresponding to  $x'$
  - There's also  $x'y + xy$  in the right side, corresponding to  $y$
  - This whole expression can be reduced to  $x' + y$

		y
	$x'y'$	$x'y$
x	$xy'$	$xy$

- For a three-variable expression with inputs  $x$ ,  $y$ ,  $z$ , the arrangement of minterms is more tricky:

		YZ			
		00	01	11	10
X	0	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
	1	$xy'z'$	$xy'z$	$xyz$	$xyz'$

		y			
		$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
X		$xy'z'$	$xy'z$	$xyz$	$xyz'$

		Z			
		$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
X		$xy'z'$	$xy'z$	$xyz$	$xyz'$

		YZ			
		00	01	11	10
X	0	$m_0$	$m_1$	$m_3$	$m_2$
	1	$m_4$	$m_5$	$m_7$	$m_6$

		y			
		$m_0$	$m_1$	$m_3$	$m_2$
X		$m_4$	$m_5$	$m_7$	$m_6$

		Z			
		$m_0$	$m_1$	$m_3$	$m_2$
X		$m_4$	$m_5$	$m_7$	$m_6$

- Another way to label the K-map (use whichever you like):

## A Three-Variable Karnaugh Map



- With this ordering, any group of 2, 4 or 8 adjacent squares on the map contains common literals that can be factored out

				Y
	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
X	$xy'z'$	$xy'z$	$xyz$	$xyz'$
				Z

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

- "Adjacency" includes wrapping around the left and right sides:

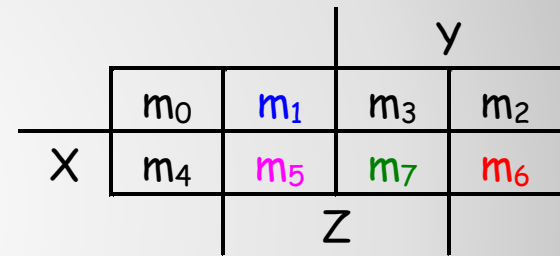
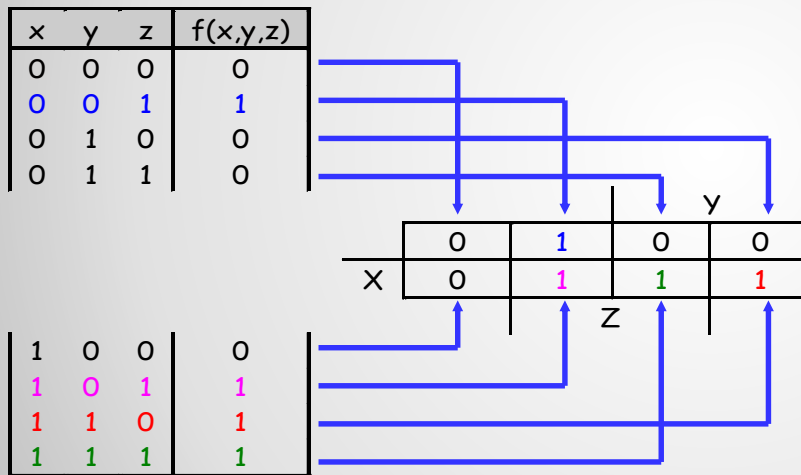
				Y
	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
X	$xy'z'$	$xy'z$	$xyz$	$xyz'$
				Z

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

- We'll use this property of adjacent squares to do our simplifications.

## Why the funny ordering?


- We can fill in the K-map directly from a truth table
  - The output in row  $i$  of the table goes into square  $m_i$  of the K-map
  - Remember that the rightmost columns of the K-map are “switched”




# K-maps From Truth Tables

- You can find the minimal SoP expression
  - Each rectangle corresponds to one product term
  - The product is determined by finding the common literals in that rectangle

		y	
	0	1	0
X	0	1	1
		z	


 $y'z$

		y	
	$x'y'z'$	$x'y'z$	$x'yz$
X	$xy'z'$	$xy'z$	$xyz$
		z	


 $xy$

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

# Reading the MSP from the K-map

- The most difficult step is grouping together all the 1s in the K-map
  - Make **rectangles** around groups of one, two, four or eight 1s
  - All of the 1s in the map should be included in at least one rectangle
  - Do *not* include any of the 0s
  - Each group corresponds to one product term

			y	
	0	1	0	0
x	0	1	1	1
		z		

# Grouping the Minterms Together

## For the Simplest Result

- Make as few rectangles as possible, to minimize the number of products in the final expression.
- Make each rectangle as large as possible, to minimize the number of literals in each term.
- Rectangles can be overlapped, if that makes them larger.

- Let's consider simplifying  $f(x,y,z) = xy + y'z + xz$
- You should convert the expression into a sum of minterms form,
  - The easiest way to do this is to make a truth table for the function, and then read off the minterms
  - You can either write out the literals or use the minterm shorthand
- Here is the truth table and sum of minterms for our example:

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

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

## K-map Simplification of SoP Expressions

# Unsimplifying Expressions

- You can also convert the expression to a sum of minterms with Boolean algebra
  - Apply the distributive law in reverse to add in missing variables.
  - Very few people actually do this, but it's occasionally useful.
$$\begin{aligned}xy + y'z + xz &= (xy \bullet 1) + (y'z \bullet 1) + (xz \bullet 1) \\&= (xy \bullet (z' + z)) + (y'z \bullet (x' + x)) + (xz \bullet (y' + y)) \\&= (xyz' + xyz) + (x'y'z + xy'z) + (xy'z + xyz) \\&= xyz' + xyz + x'y'z + xy'z \\&= m_1 + m_5 + m_6 + m_7\end{aligned}$$
- In both cases, we're actually "unsimplifying" our example expression
  - The resulting expression is larger than the original one!
  - But having all the individual minterms makes it easy to combine them together with the K-map

- In our example, we can write  $f(x,y,z)$  in two equivalent ways

$$f(x,y,z) = x'y'z + xy'z + xyz' + xyz \quad f(x,y,z) = m_1 + m_5 + m_6 + m_7$$

	y			
	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
x	$xy'z'$	$xy'z$	$xyz$	$xyz'$
	z			

	y			
	$m_0$	$m_1$	$m_3$	$m_2$
x	$m_4$	$m_5$	$m_7$	$m_6$
	z			

- In either case, the resulting K-map is shown below

	y			
	0	1	0	0
x	0	1	1	1
	z			

**Making the Example K-map**



# Practice K-map 1

- Simplify the sum of minterms  $m_1 + m_3 + m_5 + m_6$

A 2D coordinate system with a horizontal axis labeled  $x$  and a vertical axis labeled  $y$ . A shaded rectangular region is defined by the lines  $x=1$ ,  $x=3$ ,  $y=1$ , and  $y=3$ . This region is divided into four smaller quadrants by the lines  $x=2$  and  $y=2$ . The label  $z$  is positioned in the bottom-right quadrant of the shaded area.

			y	
	$m_0$	$m_1$	$m_3$	$m_2$
x	$m_4$	$m_5$	$m_7$	$m_6$
		z		

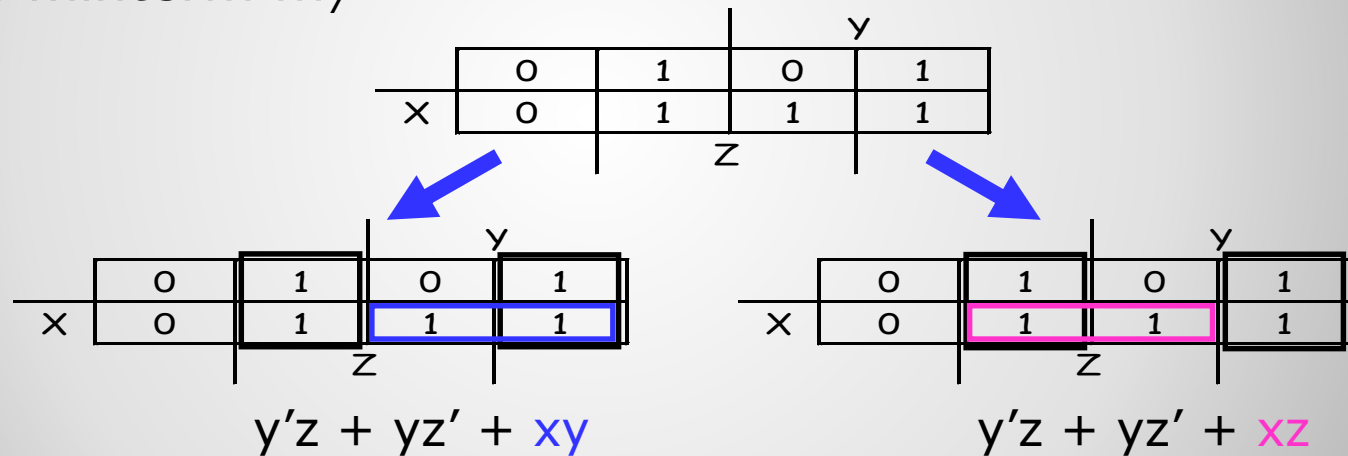
# Solutions for Practice K-map 1

- Here is the filled in K-map, with all groups shown
  - The magenta and green groups overlap, which makes each of them as large as possible
  - Minterm  $m_6$  is in a group all by its lonesome

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

- The final MSP here is  $x'z + y'z + xyz'$

- There may not necessarily be a *unique* MSP. The K-map below yields two valid and equivalent MSPs, because there are two possible ways to include minterm  $m_7$

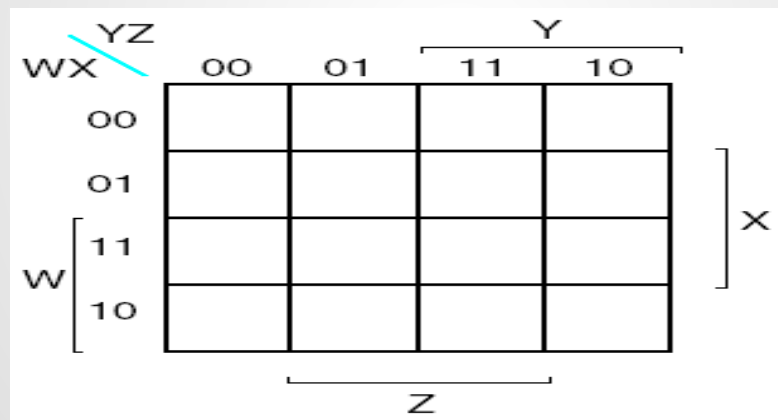


- Remember that overlapping groups is possible, as shown above

**K-maps can be tricky!**

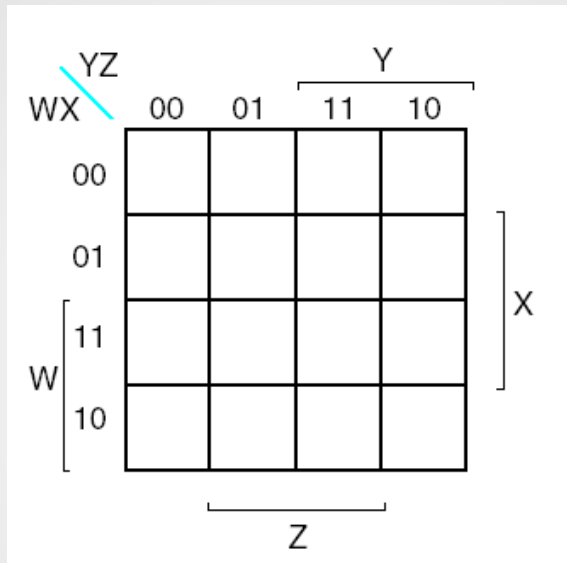
# Four-variable K-maps – $f(W,X,Y,Z)$

- We can do four-variable expressions too!
  - The minterms in the third and fourth columns, *and* in the third and fourth rows, are switched around.
  - Again, this ensures that adjacent squares have common literals



- Grouping minterms is similar to the three-variable case, but:
  - You can have rectangular groups of 1, 2, 4, 8 or 16 minterms
  - You can wrap around *all four* sides

# Four-variable K-maps



		y			
		$w'x'y'z'$	$w'x'y'z$	$w'x'yz$	$w'x'yz'$
W		$w'xy'z'$	$w'xy'z$	$w'xyz$	$w'xyz'$
		$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$
		$wx'y'z'$	$wx'y'z$	$wx'yz$	$wx'yz'$
		$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$
		Z			

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

- The expression is already a sum of minterms, so here's the K-map:

		y			
		1	0	0	1
W	X	0	1	0	0
		0	1	0	0
	Z	1	0	0	1

		y			
		$m_0$	$m_1$	$m_3$	$m_2$
W	X	$m_4$	$m_5$	$m_7$	$m_6$
		$m_{12}$	$m_{13}$	$m_{15}$	$m_{14}$
	Z	$m_8$	$m_9$	$m_{11}$	$m_{10}$

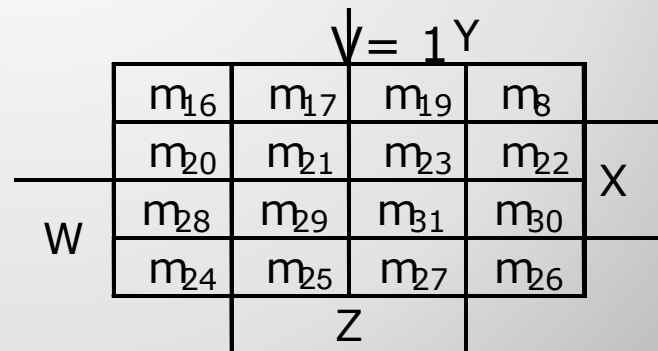
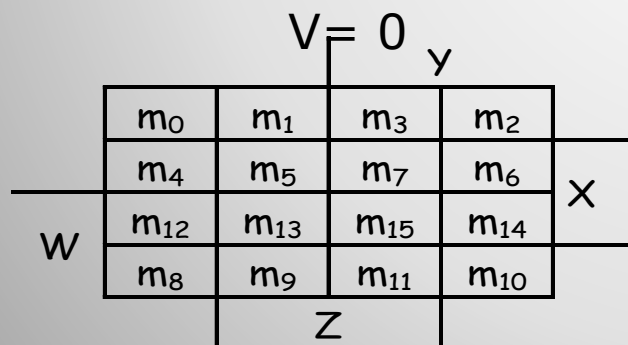
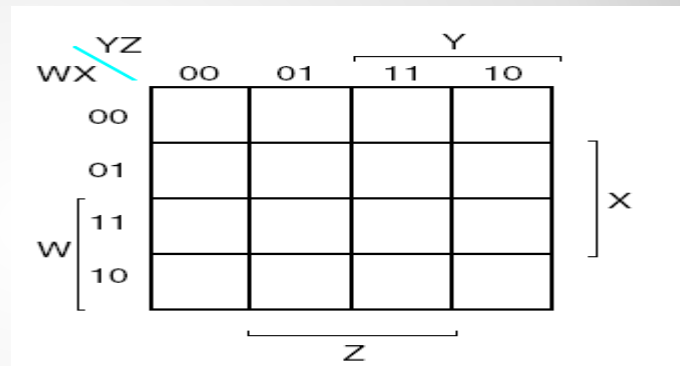
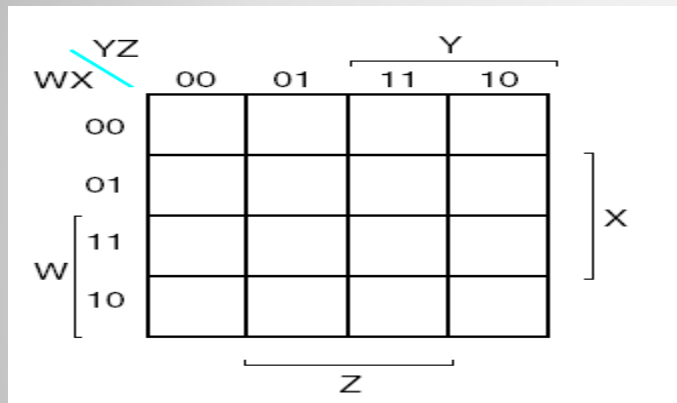
- We can make the following groups, resulting in the MSP  $x'z' + xy'z$

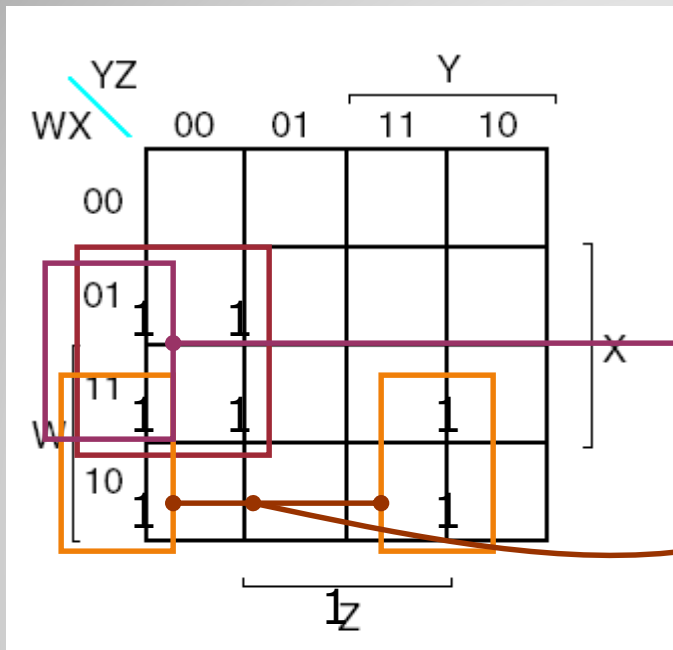
		y			
		1	0	0	1
W	X	0	1	0	0
		0	1	0	0
	Z	1	0	0	1

		y			
		$w'x'y'z'$	$w'x'y'z$	$w'x'yz$	$w'x'yz'$
W	X	$w'xy'z'$	$w'xy'z$	$w'xyz$	$w'xyz'$
		$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$
	Z	$wx'y'z'$	$wx'y'z$	$wx'yz$	$wx'yz'$

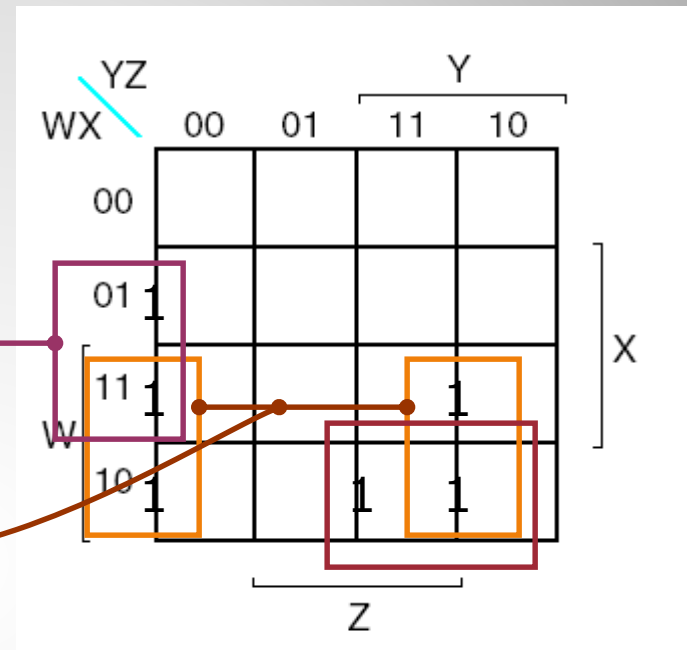
**Example: Simplify**  
 $m_0 + m_2 + m_5 + m_8 + m_{10} + m_{13}$

# Five-variable K-maps - $f(V, W, X, Y, Z)$





$V = 0$



$V = 1$

$$f = XZ'$$

$$\Sigma m(4,6,12,14,20,22,28,30)$$

$$+ V'W'Y' \quad \Sigma m(0,1,4,5)$$

$$+ W'Y'Z' \quad \Sigma m(0,4,16,20)$$

$$+ VWXY \quad \Sigma m(30,31)$$

$$+ V'WX'YZ \quad m11$$

**Simplify  $f(V,W,X,Y,Z) = \Sigma m(0,1,4,5,6,11,12,14,16,20,22,28,30,31)$**



# PoS Optimization

- Maxterms are grouped to find minimal PoS expression

		yz			
		00	01	11	10
x	0	$x + y + z$	$x + y + z'$	$x + y' + z'$	$x + y' + z$
	1	$x' + y + z$	$x' + y + z'$	$x' + y' + z'$	$x' + y' + z$

- $F(W,X,Y,Z) = \prod M(0,1,2,4,5)$

0	$x+y+z$	$x+y+z'$	$x+y'+z'$	$x+y'+z$
1	$x'+y+z$	$x'+y+z'$	$x'+y'+z'$	$x'+y'+z$
	00	01	11	
	10			

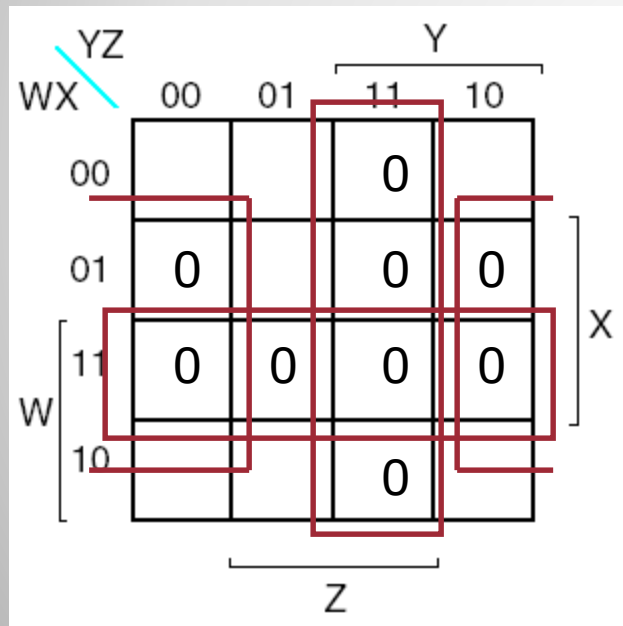
$$F(W,X,Y,Z) = Y \cdot (X + Z)$$

0	0	0	1	0
1	1	0	1	1
	00	01	11	
	10			

**PoS Optimization**

$$F(W,X,Y,Z) = \sum m(0,1,2,5,8,9,10)$$

$$= \prod M(3,4,6,7,11,12,13,14,15)$$



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

Or,

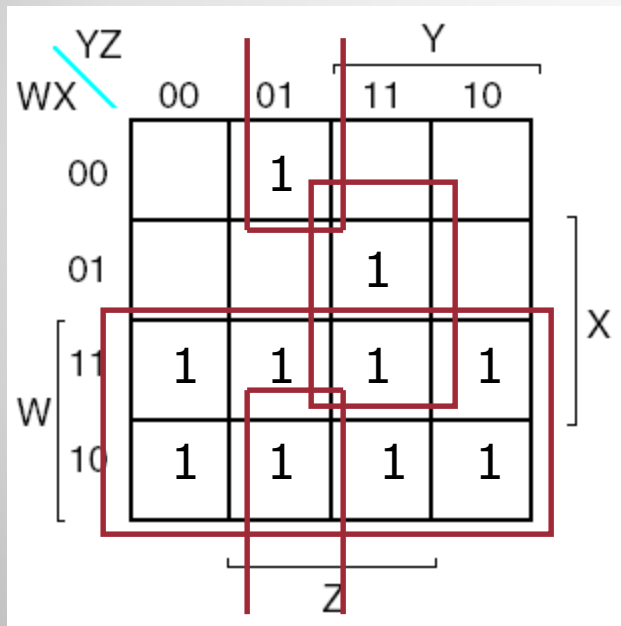
$$F(W,X,Y,Z) = X'Y' + X'Z' + W'Y'Z$$

Which one is the minimal one?

# PoS Optimization from SoP

$$F(W,X,Y,Z) = \prod M(0,2,3,4,5,6)$$

$$= \sum m(1,7,8,9,10,11,12,13,14,15)$$



$$F(W,X,Y,Z) = W + XYZ + X'Y'Z$$

# SoP Optimization from PoS

- You don't always need all  $2^n$  input combinations in an  $n$ -variable function
  - If you can guarantee that certain input combinations never occur
  - If some outputs aren't used in the rest of the circuit
- We mark don't-care outputs in truth tables and K-maps with Xs.

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	1
0	1	0	X
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	X
1	1	1	1

- Within a K-map, each X can be considered as either 0 or 1. You should pick the interpretation that allows for the most simplification.

**I don't care!**

- Find a MSP for

$$f(w,x,y,z) = \sum m(0,2,4,5,8,14,15), d(w,x,y,z) = \sum m(7,10,13)$$

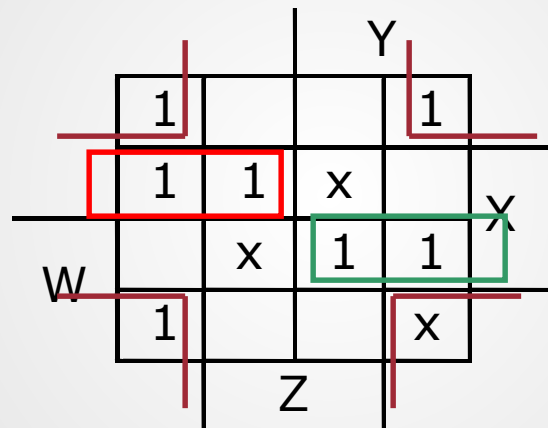
This notation means that input combinations  $wxyz = 0111, 1010$  and  $1101$  (corresponding to minterms  $m_7, m_{10}$  and  $m_{13}$ ) are unused.

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

**Practice K-map**

- Find a MSP for:

$$f(w,x,y,z) = \sum m(0,2,4,5,8,14,15), d(w,x,y,z) = \sum m(7,10,13)$$



$$f(w,x,y,z) = x'z' + w'xy' + wxy$$

# Solutions for Practice K-map

# K-map Summary

- K-maps are an alternative to algebra for simplifying expressions
  - The result is a MSP/MPS, which leads to a minimal two-level circuit
  - It's easy to handle don't-care conditions
  - K-maps are really only good for manual simplification of small expressions...
- Things to keep in mind:
  - Remember the correct order of minterms/maxterms on the K-map
  - When grouping, you can wrap around all sides of the K-map, and your groups can overlap
  - Make as few rectangles as possible, but make each of them as large as possible. This leads to fewer, but simpler, product terms
  - There may be more than one valid solution



# Combinational Logic

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

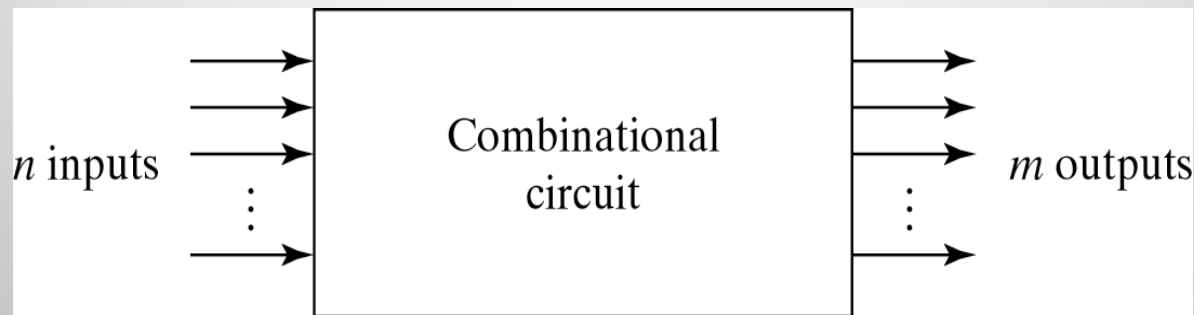


Fig. 4-1 Block Diagram of Combinational Circuit

## 2. Analysis procedure

- To obtain the output Boolean functions from a logic diagram, proceed as follows:
  1. Label all gate outputs that are a function of input variables with arbitrary symbols. Determine the Boolean functions for each gate output.
  2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the Boolean functions for these gates.

3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

# Example

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

$$F_1 = T_3 + T_2$$

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

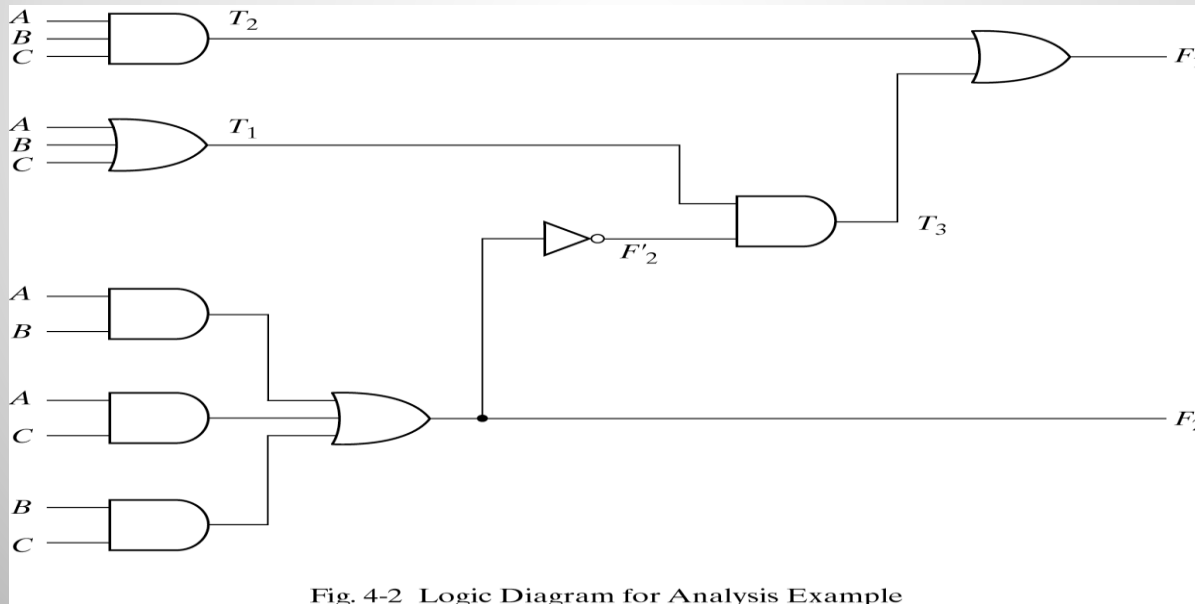


Fig. 4-2 Logic Diagram for Analysis Example

# Derive truth table from logic diagram

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

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

<i>A</i>	<i>B</i>	<i>C</i>	<i>F</i> <sub>2</sub>	<i>F</i> <sub>2</sub>	<i>T</i> <sub>1</sub>	<i>T</i> <sub>2</sub>	<i>T</i> <sub>3</sub>	<i>F</i> <sub>1</sub>
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

### 3. Design procedure

1. Table 4-2 is a Code-Conversion example, first, we can list the relation of the BCD and Excess-3 codes in the truth table.

**Table 4-2**

*Truth Table for Code-Conversion Example*

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

# Karnaugh map

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

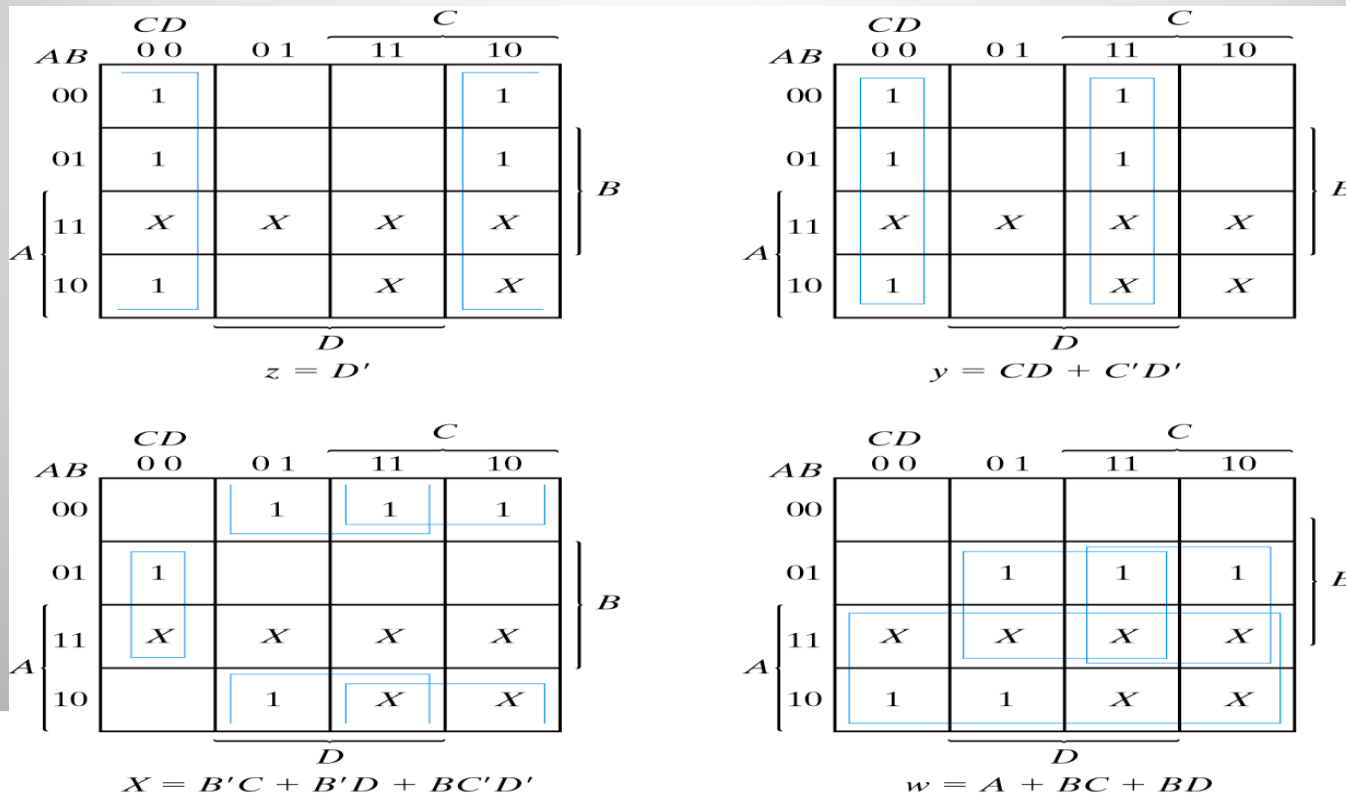


Fig. 4-3 Maps for BCD to Excess-3 Code Converter

# Circuit implementation

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

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

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

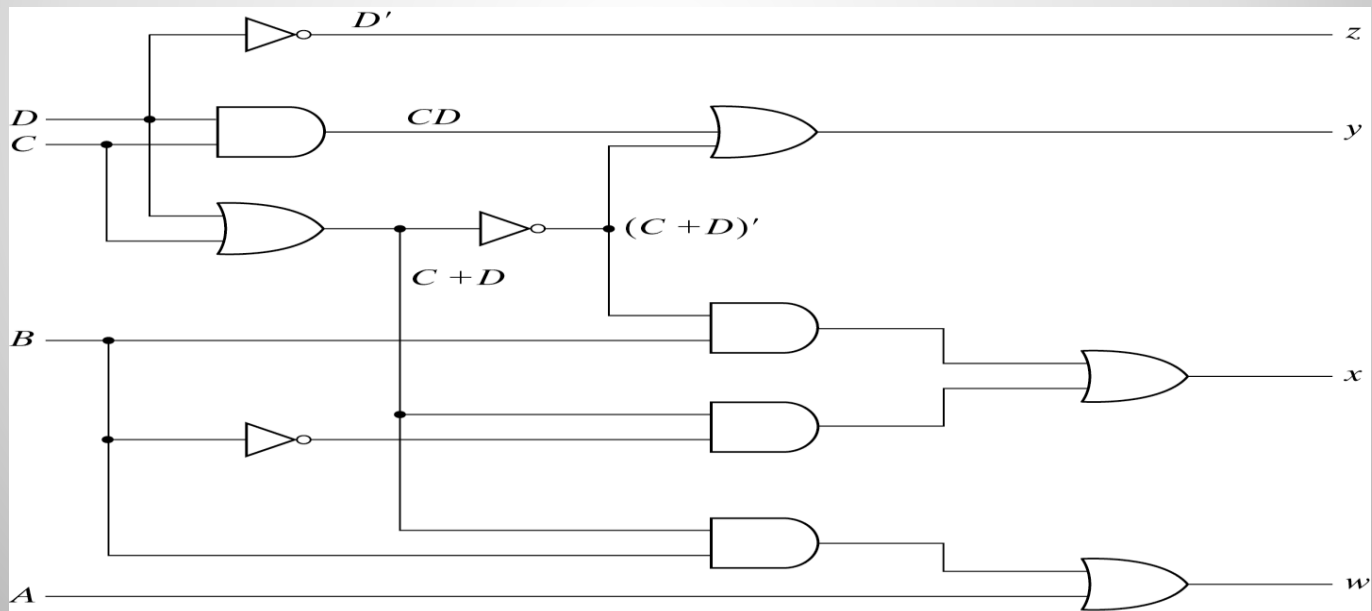


Fig. 4-4 Logic Diagram for BCD to Excess-3 Code Converter



## 4. Binary Adder-Subtractor

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

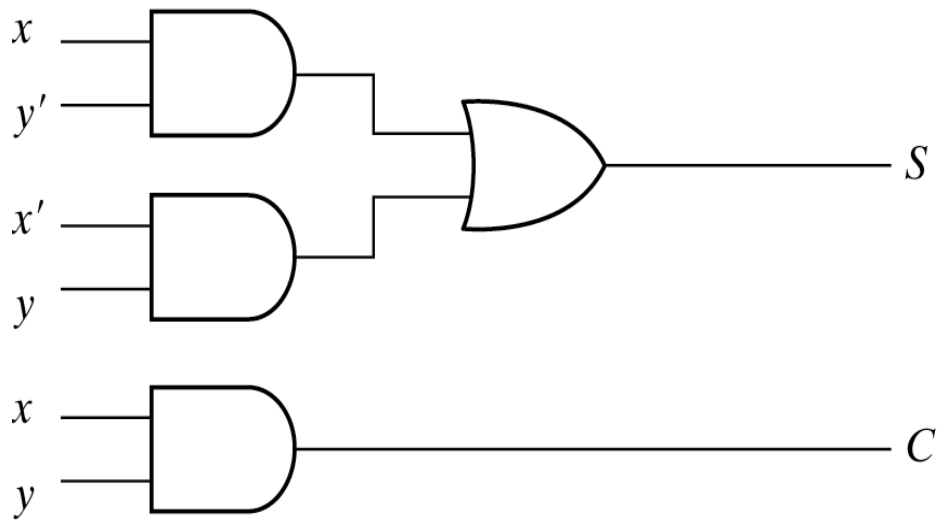
**Table 4-3**  
*Half Adder*

<i>x</i>	<i>y</i>	<i>C</i>	<i>S</i>
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

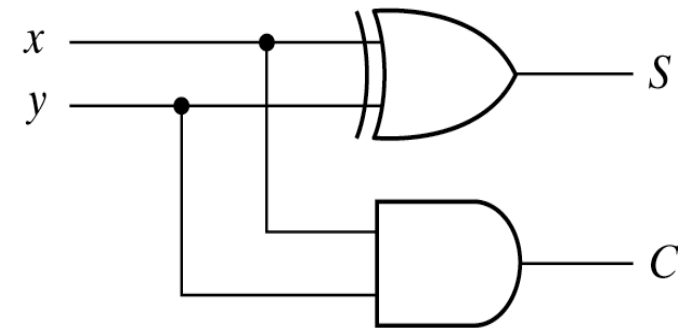
S: Sum  
C: Carry

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

$$C = xy$$



$$(a) \begin{aligned} S &= xy' + x'y \\ C &= xy \end{aligned}$$



$$(b) \begin{aligned} S &= x \oplus y \\ C &= xy \end{aligned}$$

Fig. 4-5 Implementation of Half-Adder

# Implementation of Half-Adder

# Full-Adder

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

**Table 4-4**  
*Full Adder*

<i>x</i>	<i>y</i>	<i>z</i>	<i>C</i>	<i>S</i>
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

# Simplified Expressions

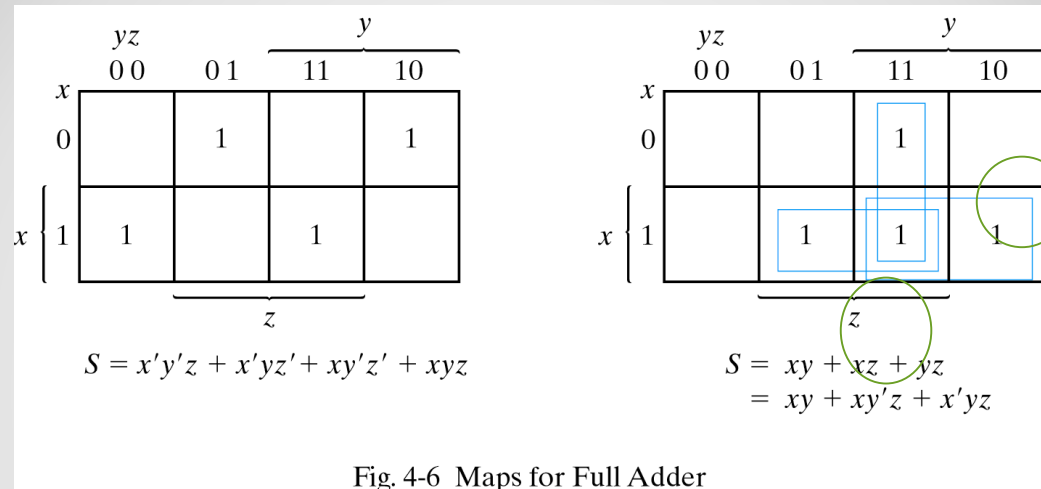


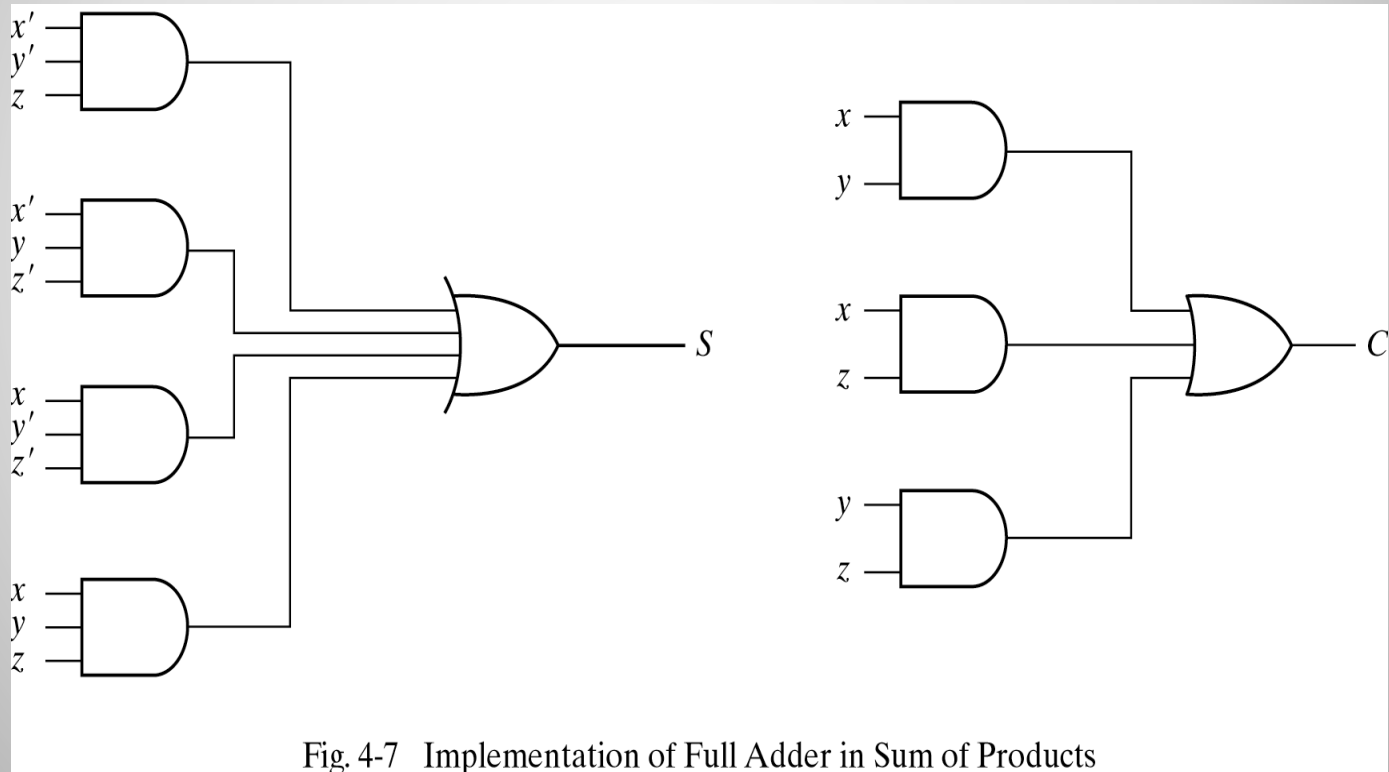
Fig. 4-6 Maps for Full Adder

C

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

$$C = xy + xz + yz$$

# Full adder implemented in SOP



# Another implementation

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

$$S = z \oplus (x \oplus y)$$

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

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

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

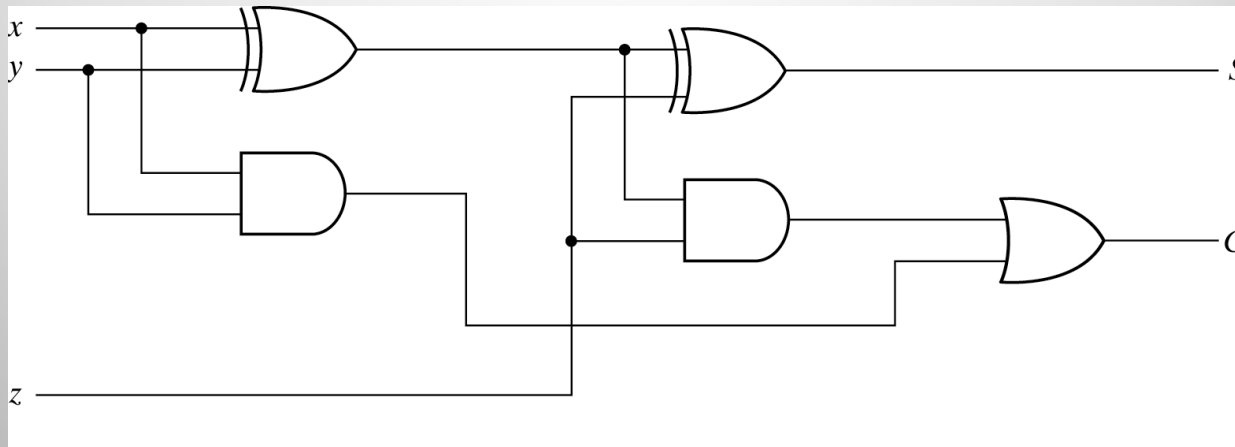


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

# Binary adder

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

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

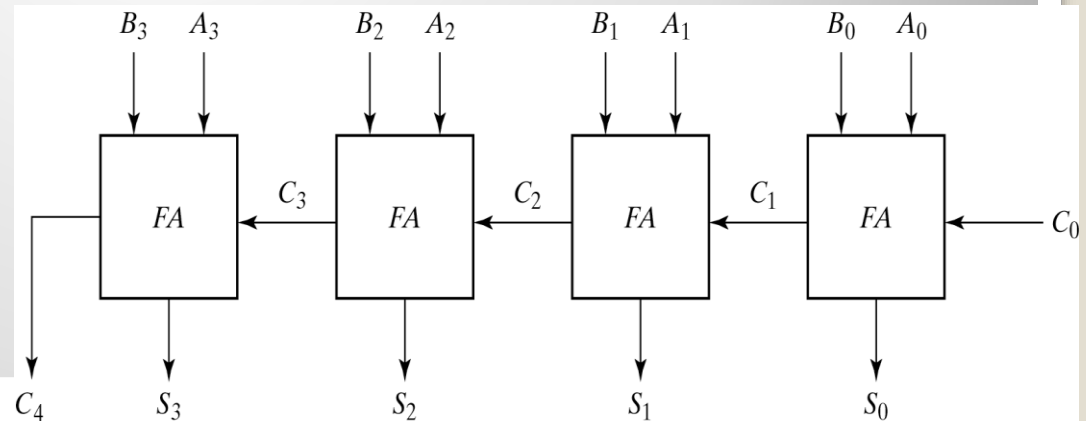


Fig. 4-9 4-Bit Adder

# Carry Propagation

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



# Carry Propagation

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

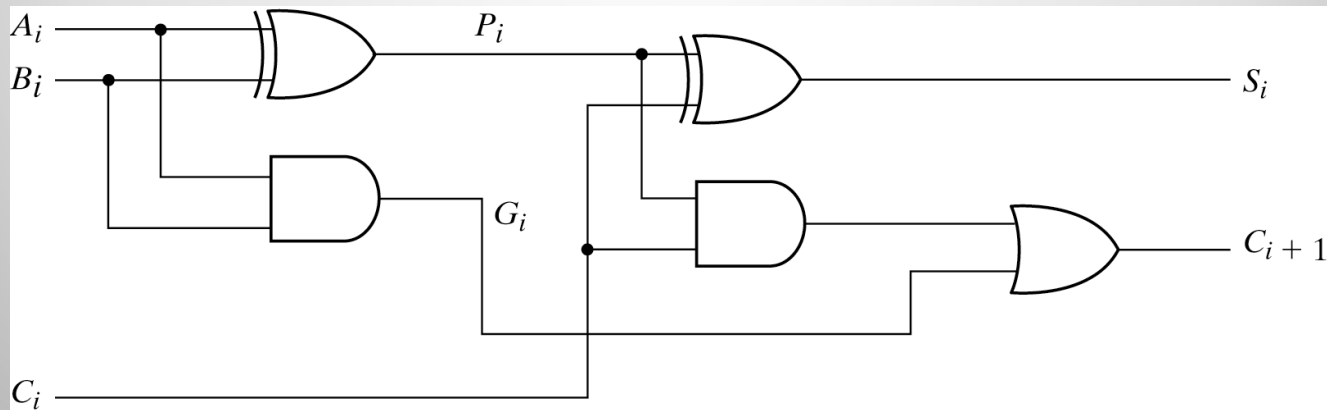


Fig. 4-10 Full Adder with P and G Shown

# Boolean functions

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

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

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

$C_0$  = input carry

$$C_1 = G_0 + P_0 C_0$$

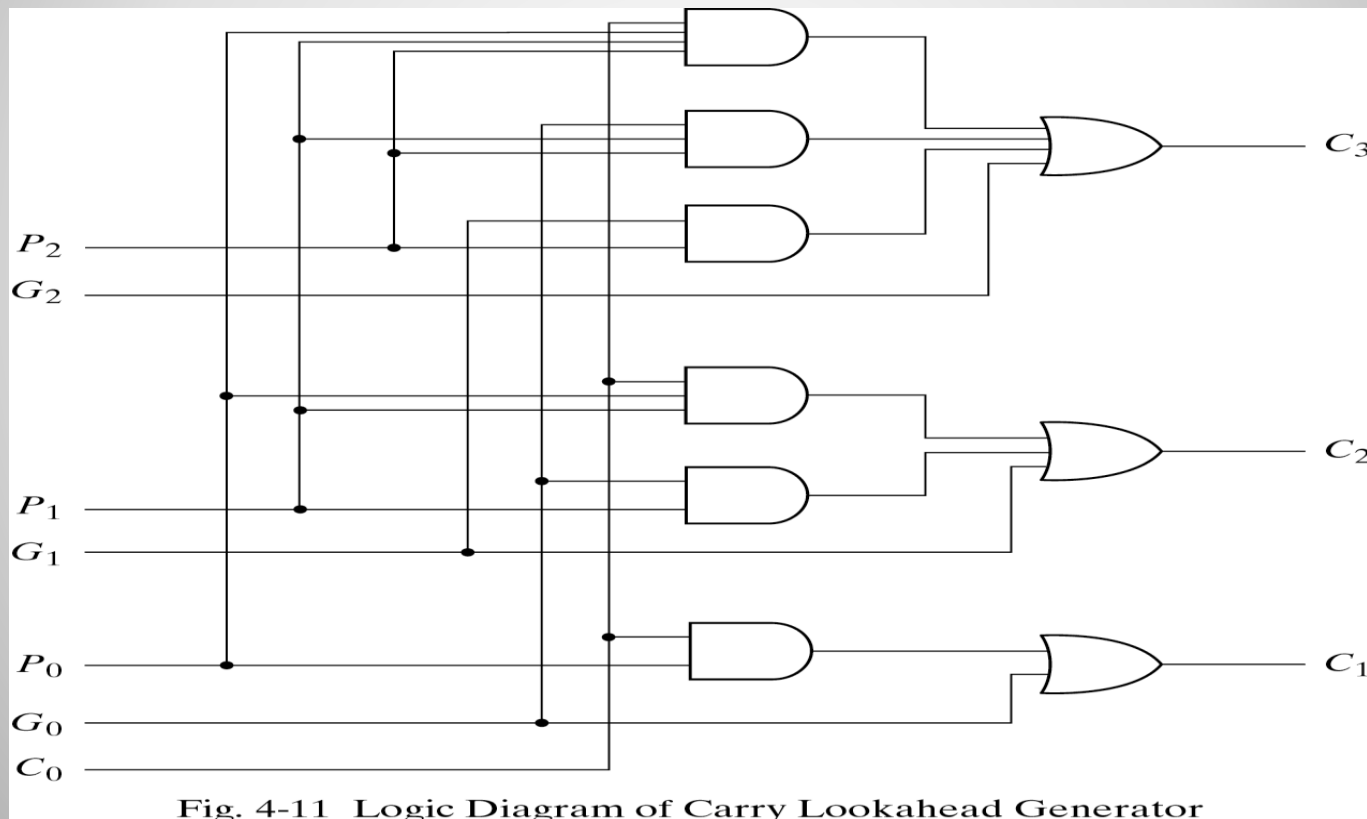
$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

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

- $C_3$  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  $C_1$ .



# 4-bit adder with carry lookahead

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

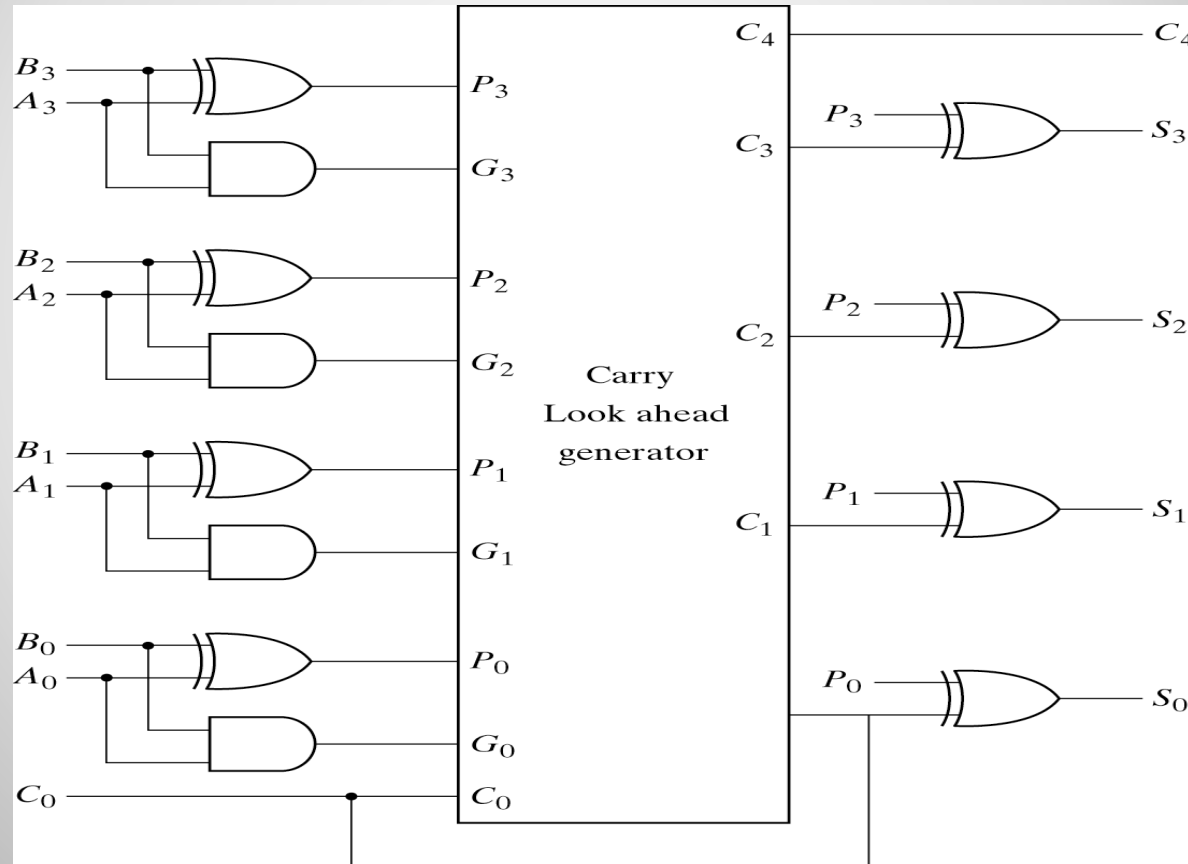


Fig. 4-12 4-Bit Adder with Carry Lookahead

M = 1  $\rightarrow$  subtractor ; M = 0  $\rightarrow$  adder



# Overflow

- It is **worth** noting Fig.4-13 that binary numbers in the **signed-complement system are added and subtracted** by the same basic addition and subtraction rules **as unsigned numbers**.
- Overflow is a problem in digital computers because the number of bits that hold the number is finite and a result that contains  $n+1$  bits cannot be accommodated.

# Overflow on signed and unsigned

- When two **unsigned** numbers are added, an overflow is detected from the **end carry out of the MSB position**.
- 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 can't occur** after an addition if one number is **positive** and the other is **negative**.
- An overflow may occur if the two numbers added are both positive or both negative.

# 5 Decimal adder

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

**Table 4-5**  
*Derivation of BCD Adder*

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



## Rules of BCD adder

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

$$C = K + Z_8Z_4 + Z_8Z_2$$

# Implementation of BCD adder

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

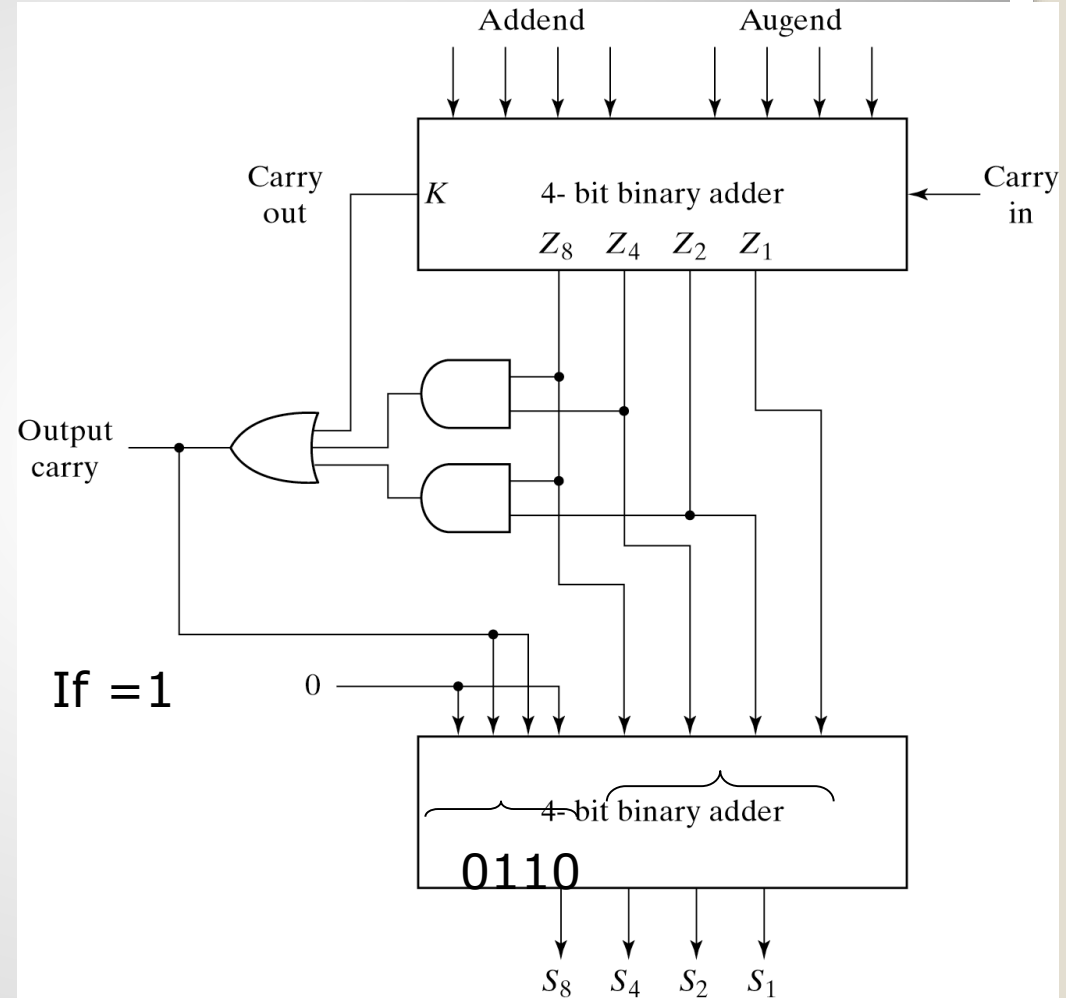


Fig. 4-14 Block Diagram of a BCD Adder

# 6. Binary multiplier

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

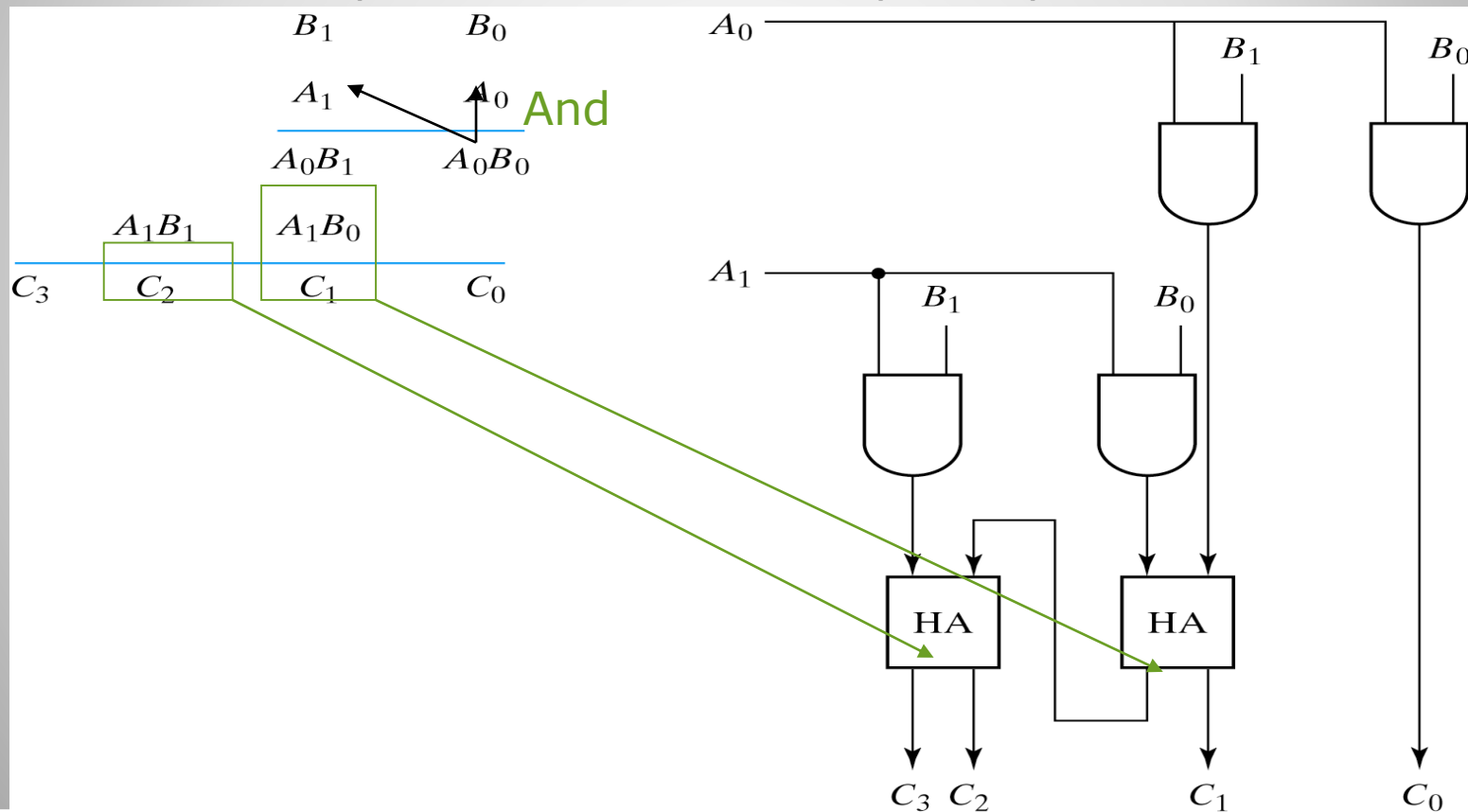


Fig. 4-15 2-Bit by 2-Bit Binary Multiplier

# 4-bit by 3-bit binary multiplier

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

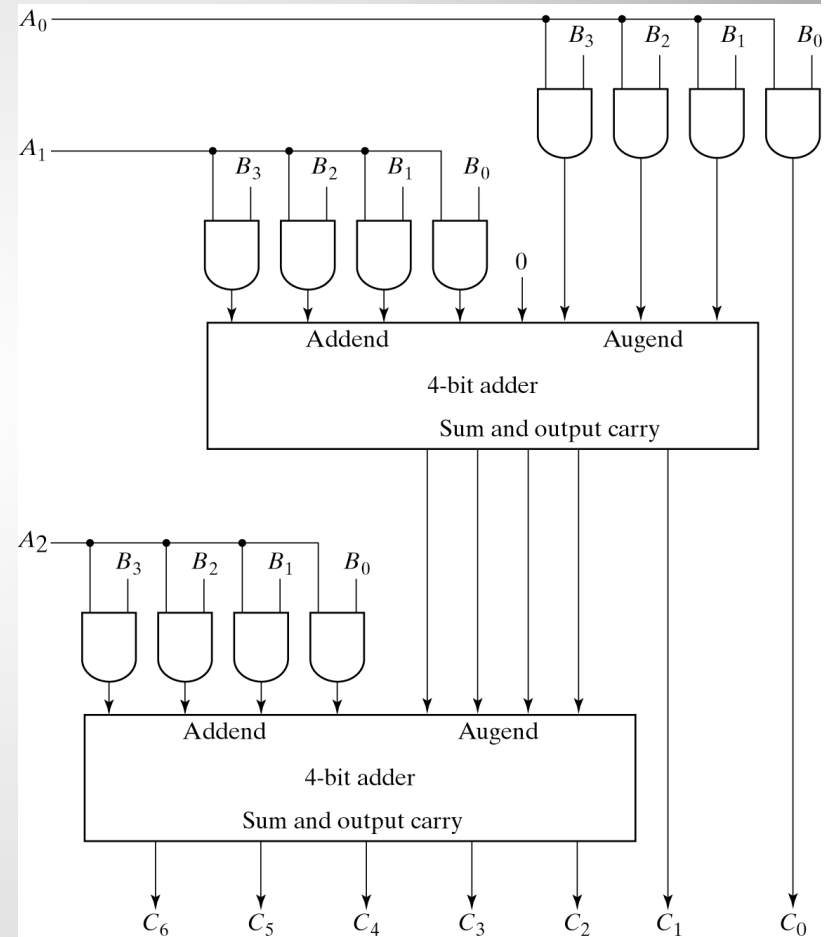


Fig. 4-16 4-Bit by 3-Bit Binary Multiplier

# 7. Magnitude comparator

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

$$A = A_3A_2A_1A_0 ; B = B_3B_2B_1B_0$$

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

$$(A = B) = x_3x_2x_1x_0$$

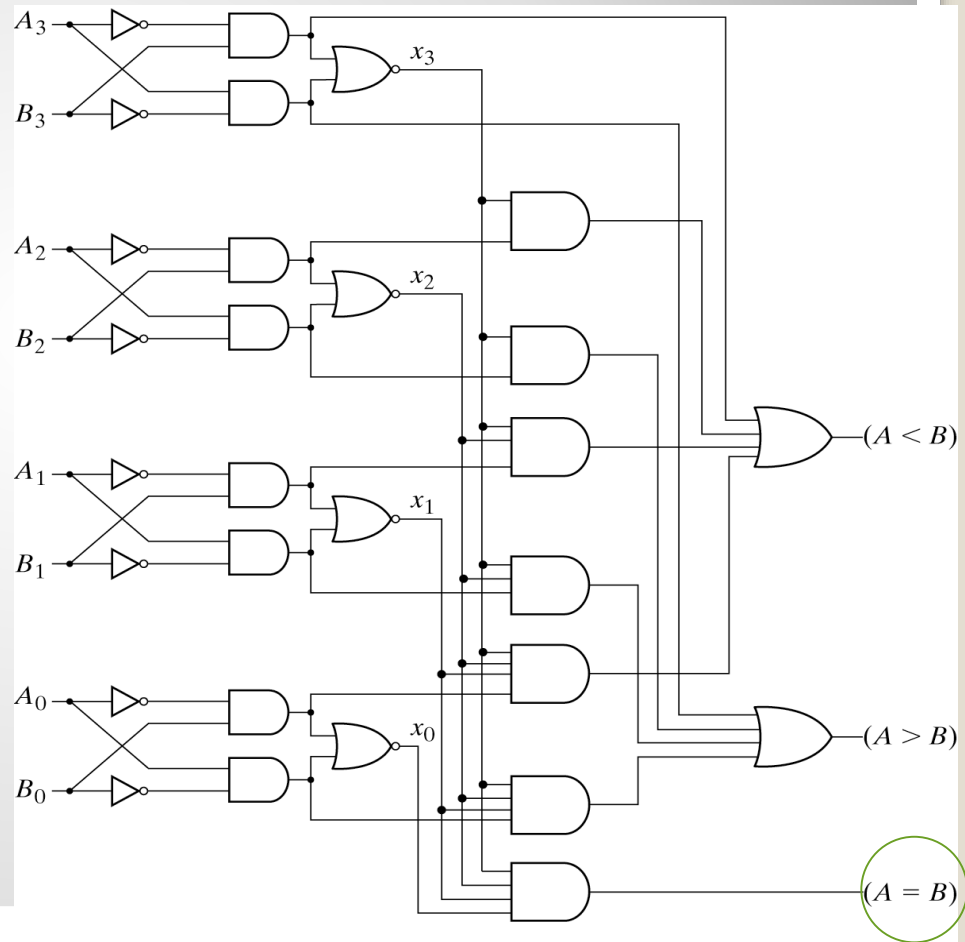


Fig. 4-17 4-Bit Magnitude Comparator

# Magnitude comparator

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

$(A > B) =$

$$A_3B'_3 + x_3A_2B'_2 + x_3x_2A_1B'_1 + x_3x_2x_1A_0B'_0$$

$(A < B) =$

$$A'_3B_3 + x_3A'_2B_2 + x_3x_2A'_1B_1 + x_3x_2x_1A'_0B_0$$

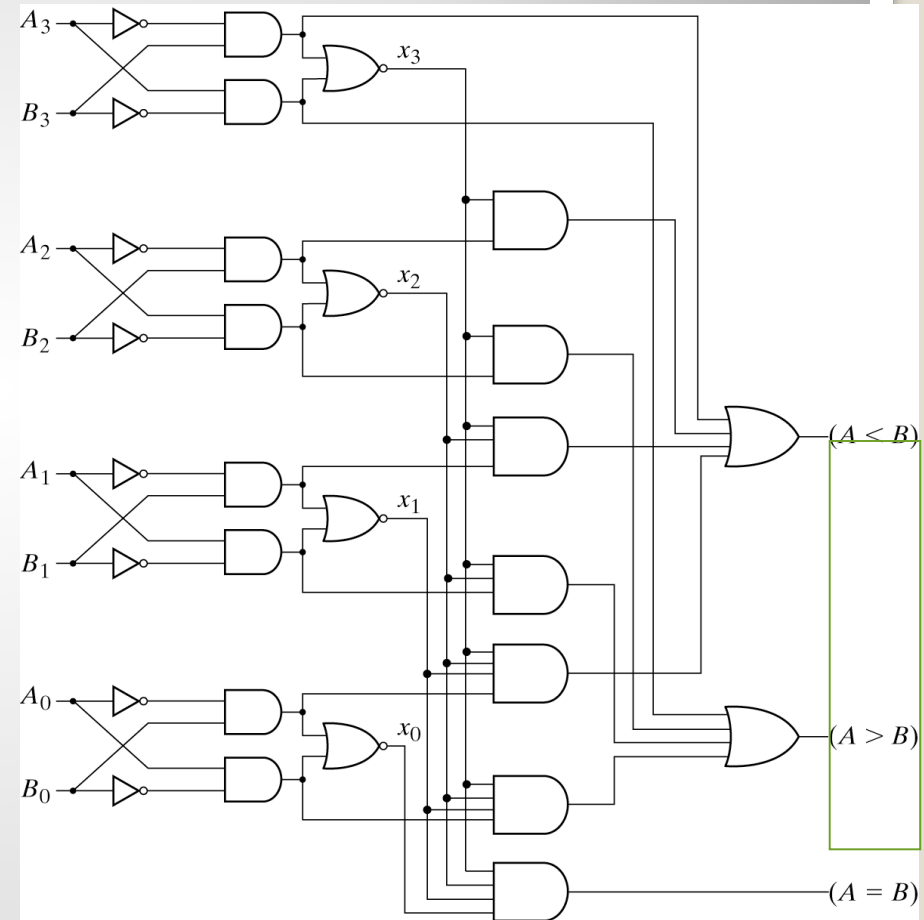


Fig. 4-17 4-Bit Magnitude Comparator

## 8. Decoders

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

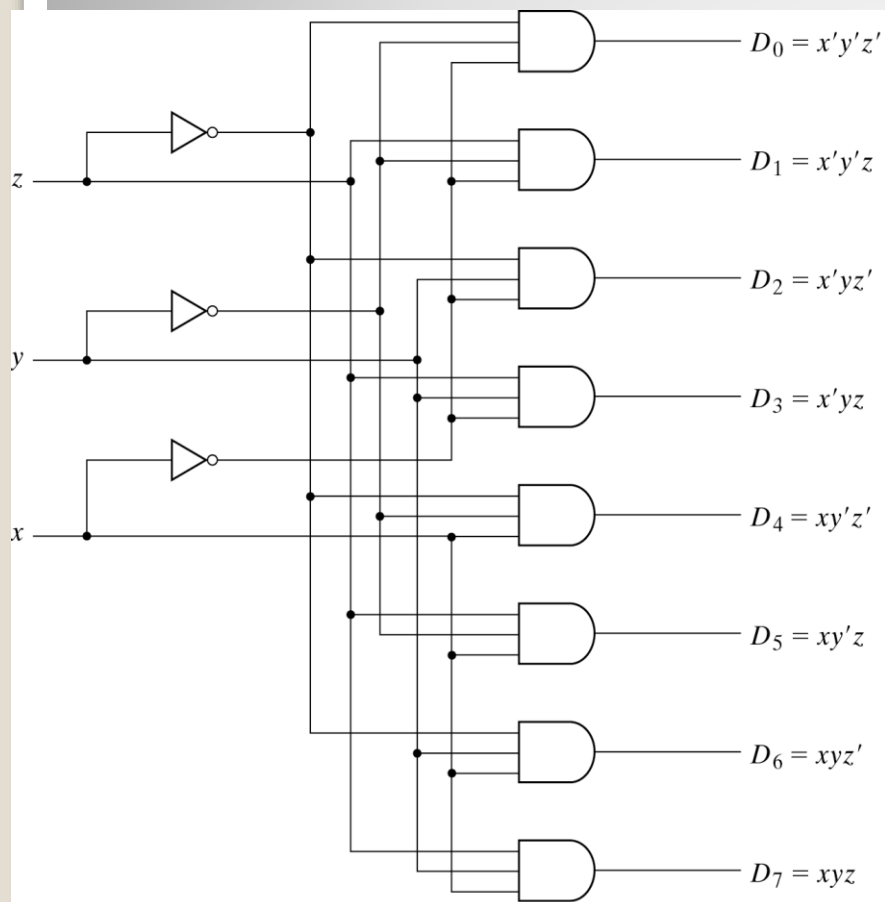


Fig. 4-18 3-to-8-Line Decoder

**Table 4-6**

*Truth Table of a 3-to-8-Line Decoder*

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



# Decoder with enable input

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

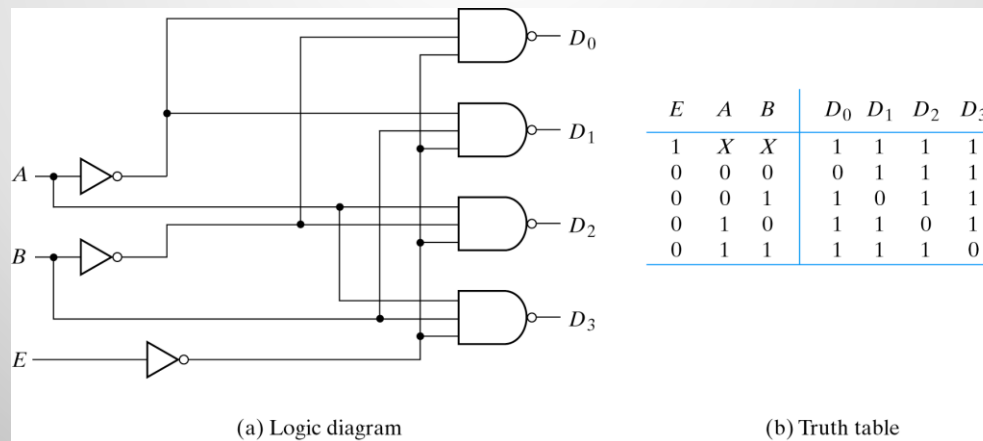
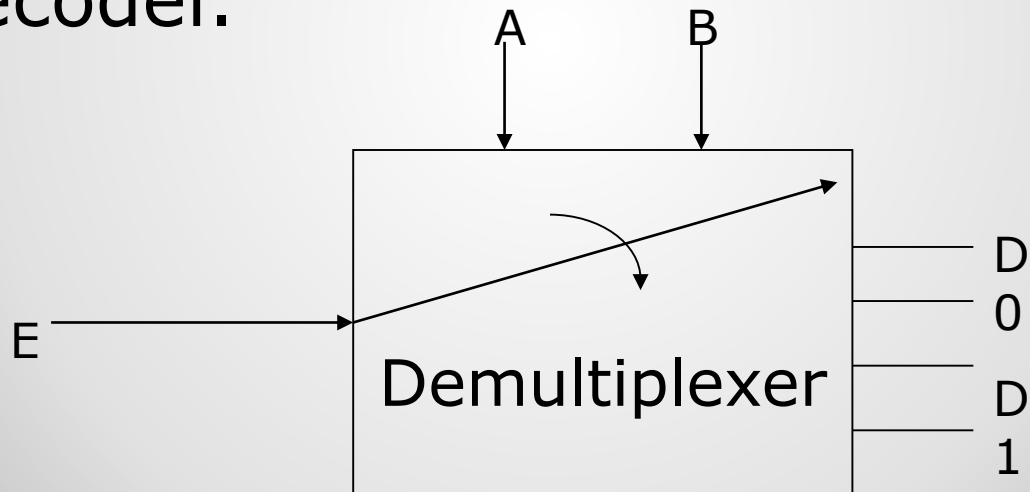


Fig. 4-19 2-to-4-Line Decoder with Enable Input

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



**Demultiplexer**

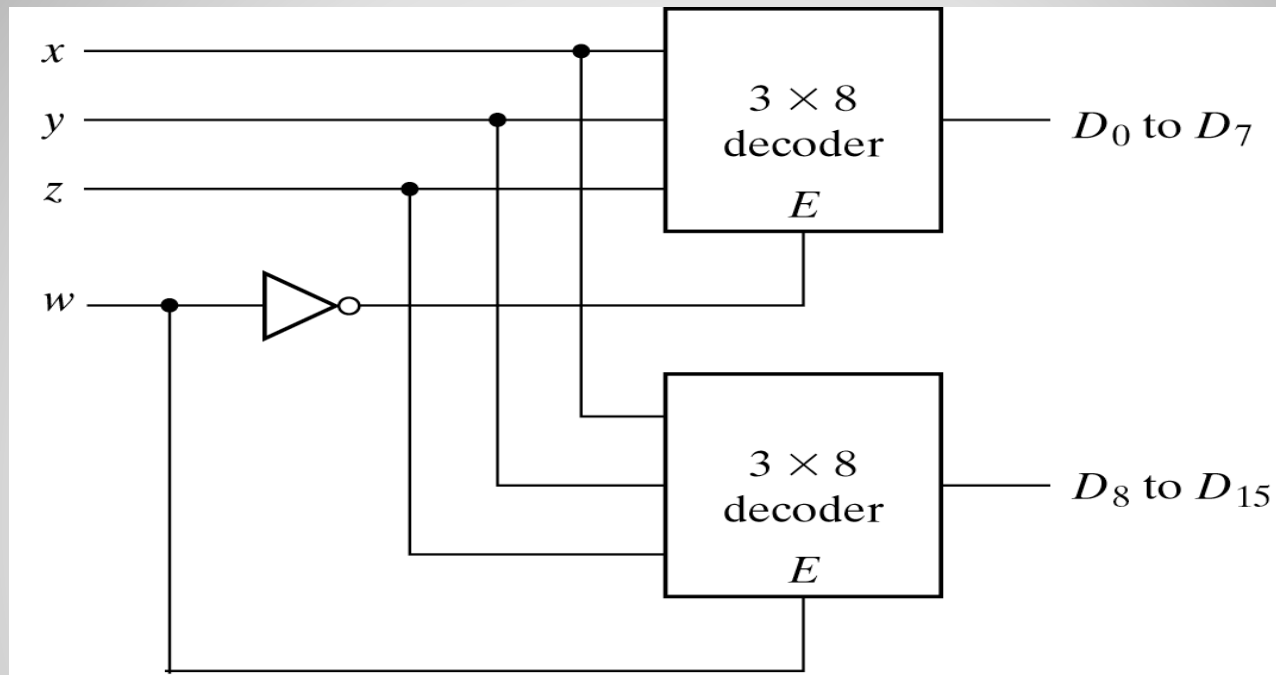


Fig. 4-20  $4 \times 16$  Decoder Constructed with Two  $3 \times 8$  Decoders

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

# Implementation of a Full Adder with a Decoder

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

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

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

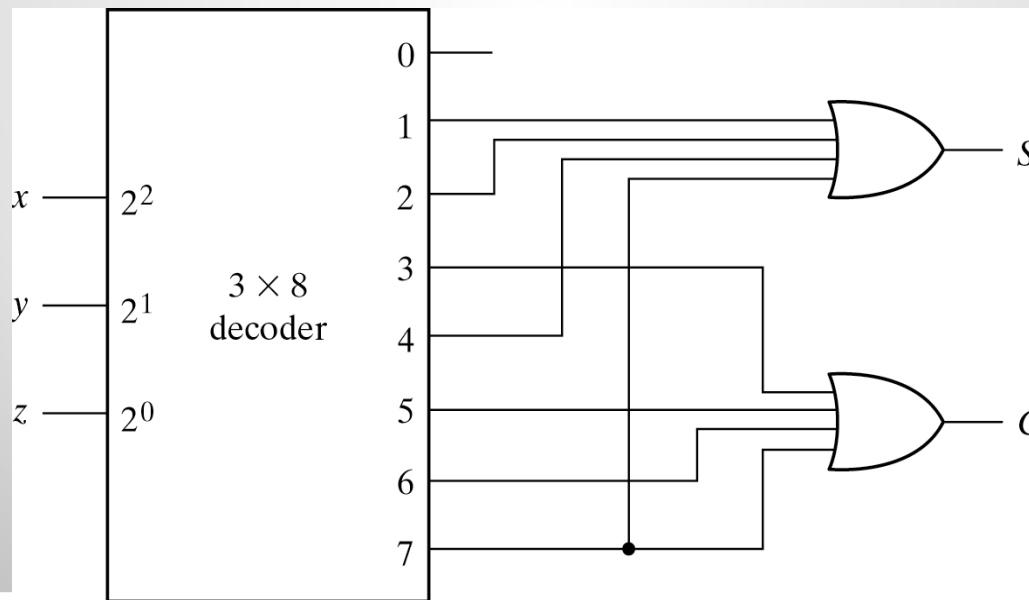


Fig. 4-21 Implementation of a Full Adder with a Decoder

## 9. Encoders

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

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

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

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

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

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



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

## Priority encoder

# Priority encoder

$V=0 \rightarrow$  no valid inputs

$V=1 \rightarrow$  valid inputs

$X$ 's in output columns  
represent

don't-care conditions

$X$ 's in the input columns are  
useful for representing a  
truth

table in condensed form.

Instead of listing all 16  
minterms of four variables.

**Table 4-8**

*Truth Table of a Priority Encoder*

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

# 4-input priority encoder

- Implementation of table 4-8

$$x = D_2 + D_3$$

$$y = D_3 + D_1 D'_2$$

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

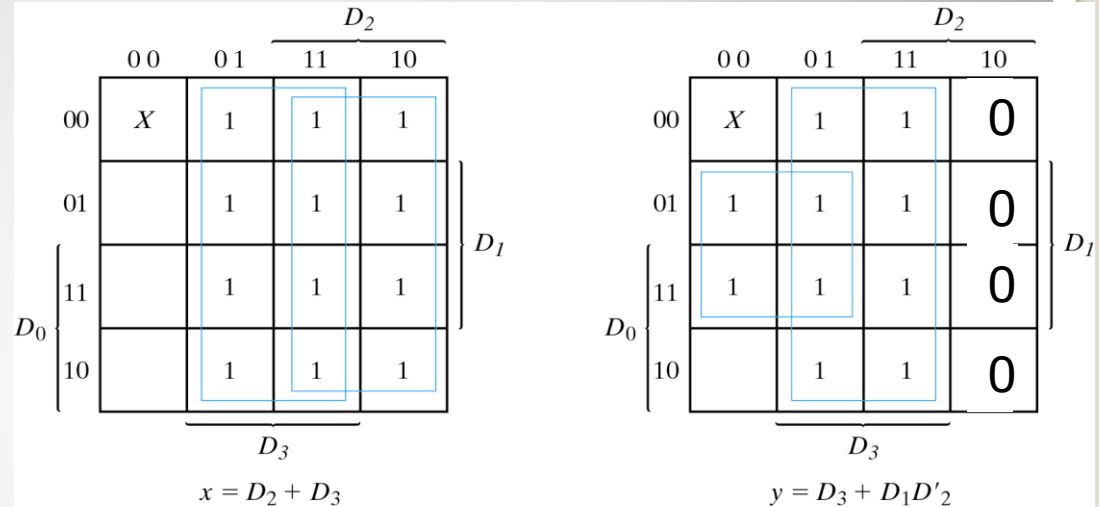


Fig. 4-22 Maps for a Priority Encoder

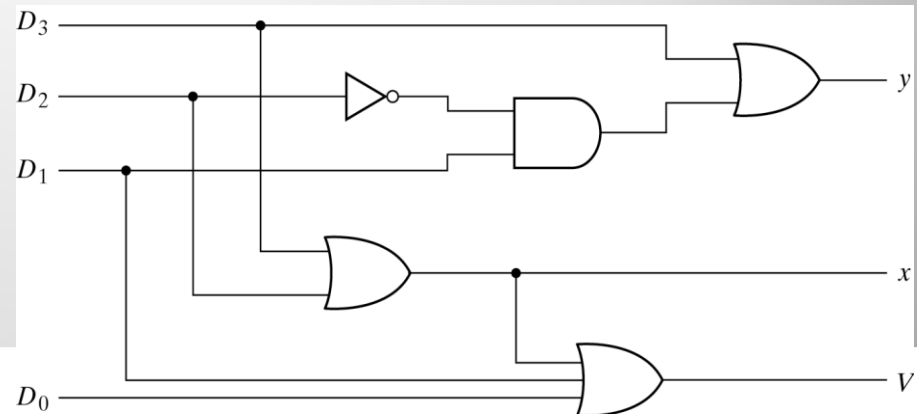


Fig. 4-23 4-Input Priority Encoder



# 10. Multiplexers

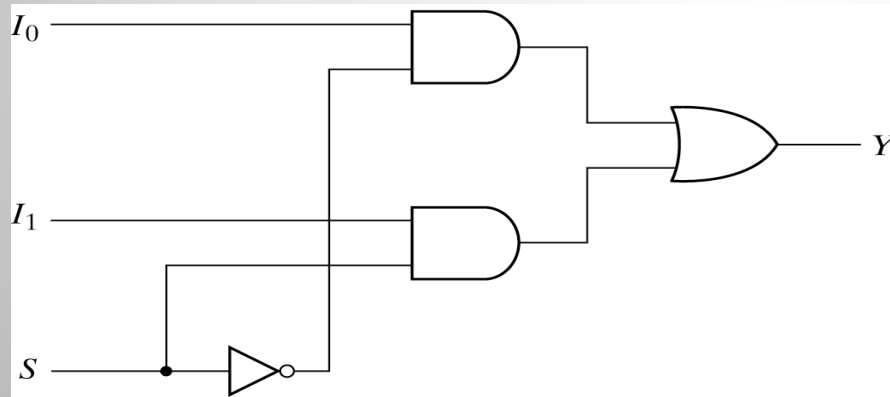
$S = 0, Y = I_0$

$S = 1, Y = I_1$

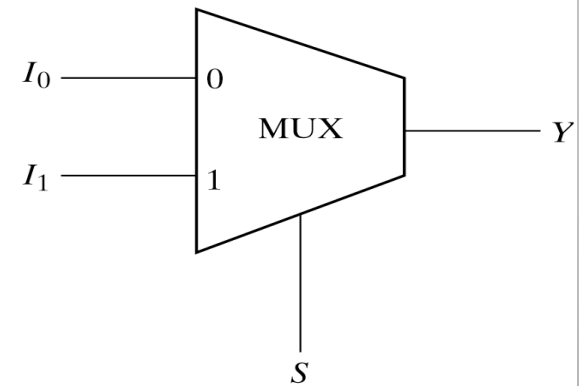
Truth Table  $\rightarrow$

S	Y
0	$I_0$
1	$I_1$

$$Y = S'I_0 + SI_1$$

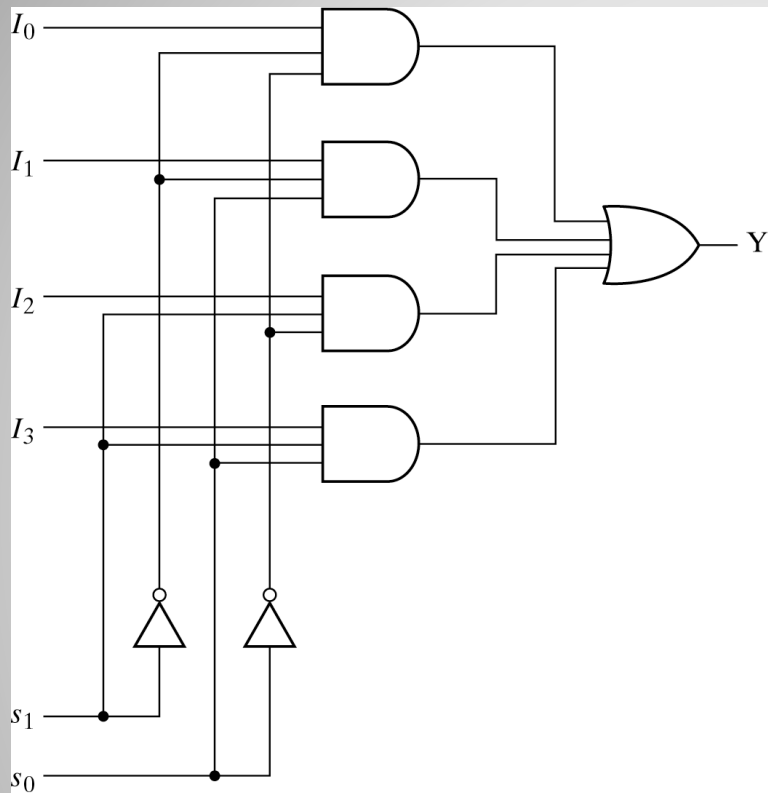


(a) Logic diagram



(b) Block diagram

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



(a) Logic diagram

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

(b) Function table

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

# 4-to-1 Line Multiplexer

# Quadruple 2-to-1 Line Multiplexer

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

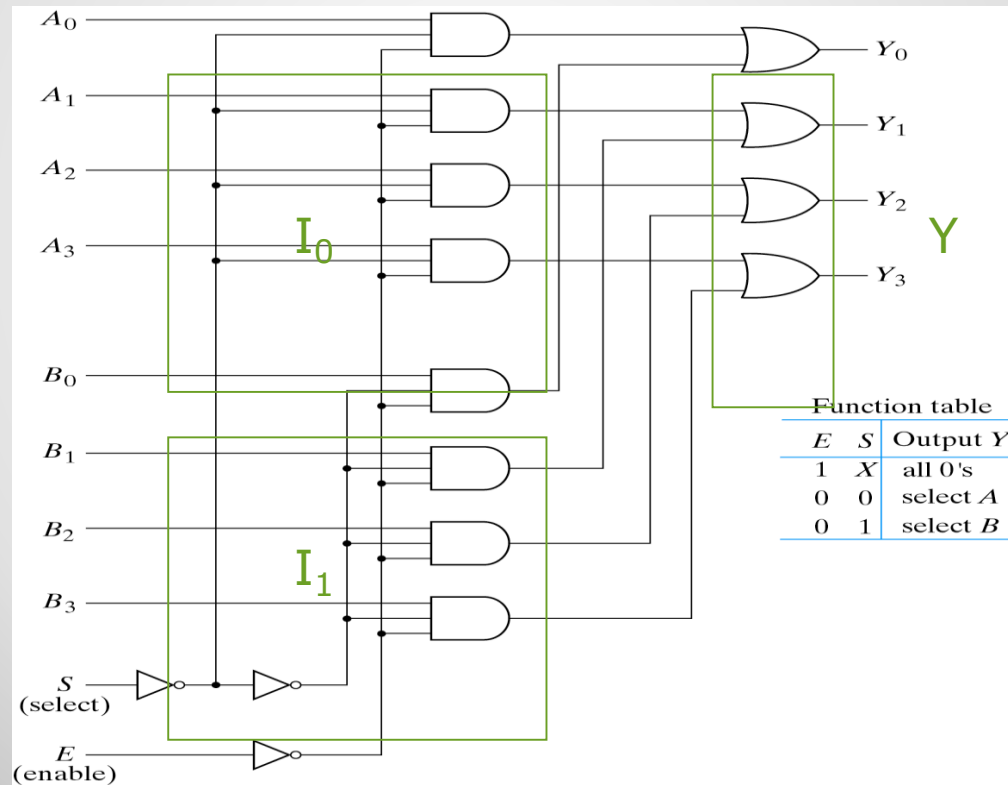


Fig. 4-26 Quadruple 2-to-1-Line Multiplexer

# Boolean function implementation

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

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

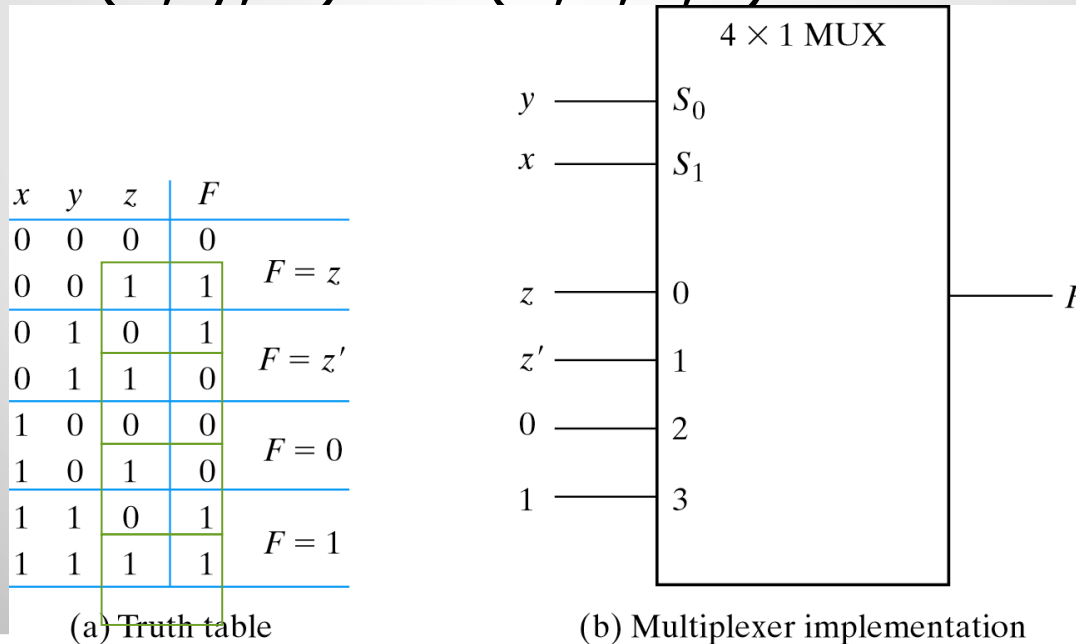


Fig. 4-27 Implementing a Boolean Function with a Multiplexer

# 4-input function with a multiplexer

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

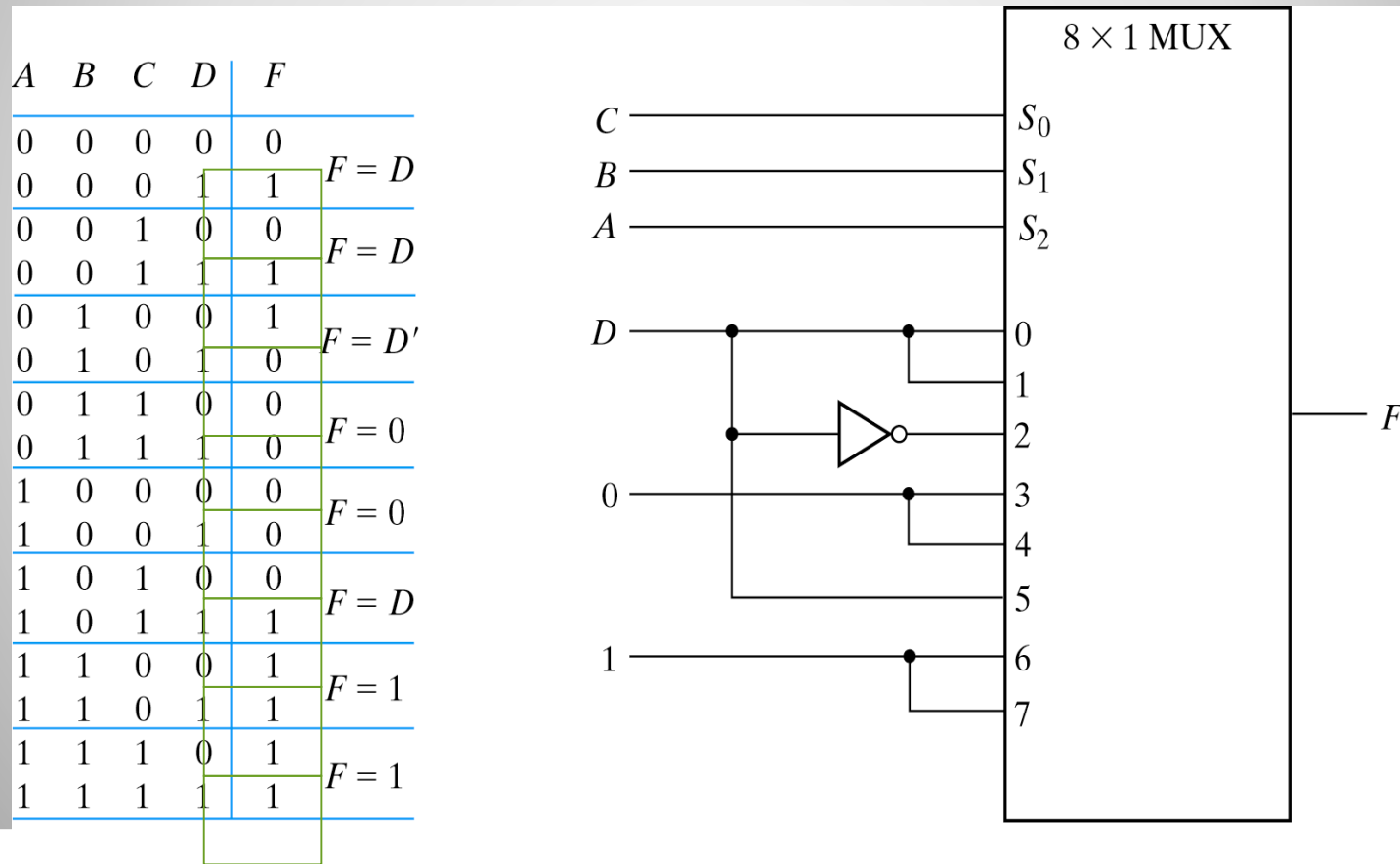


Fig. 4-28 Implementing a 4-Input Function with a Multiplexer

# Three-State Gates

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

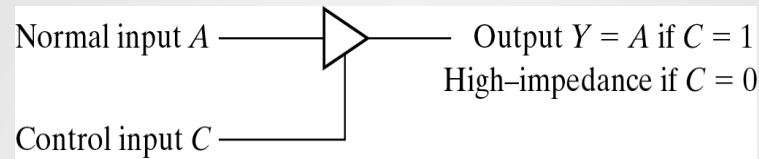
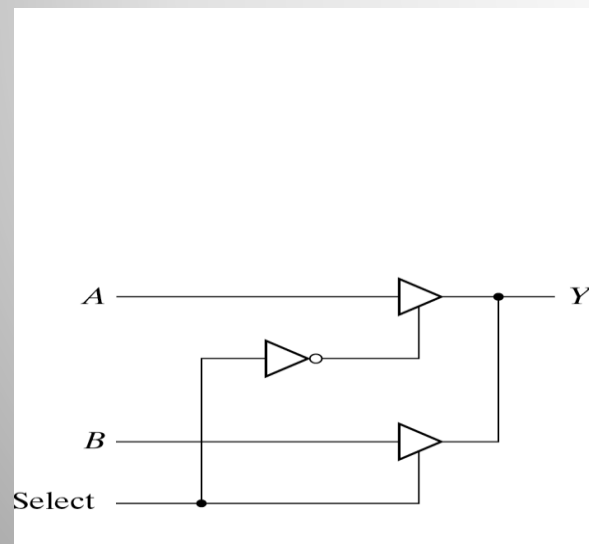
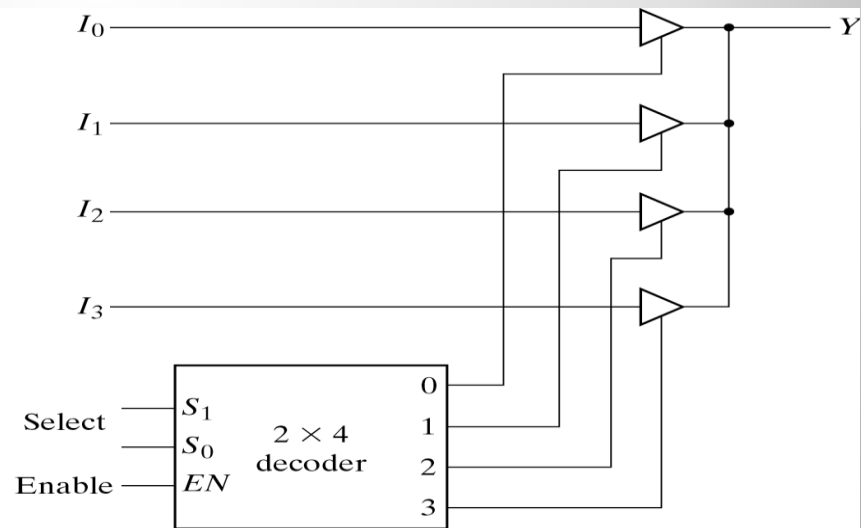


Fig. 4-29 Graphic Symbol for a Three-State Buffer



(a) 2-to-1- line mux



(b) 4 - to - 1 line mux

Fig. 4-30 Multiplexers with Three-State Gates

# 11. HDL for combinational circuits

- A module can be described in any one of the following modeling techniques:
  1. Gate-level modeling using instantiation of primitive gates and user-defined modules.
  2. Dataflow modeling using continuous assignment statements with keyword assign.
  3. Behavioral modeling using procedural assignment statements with keyword always.

# Gate-level Modeling

- A circuit is specified by its logic gates and their interconnection.
- Verilog recognizes 12 basic gates as predefined primitives.
- The logic values of each gate may be 1, 0, x(unknown), z(high-impedance).

**Table 4-9**

*Truth Table for Predefined Primitive Gates*

<b>and</b>	0	1	x	z	<b>or</b>	0	1	x	z
0	0	0	0	0	0	0	1	x	x
1	0	1	x	x	1	1	1	1	1
x	0	x	x	x	x	x	1	x	x
z	0	x	x	x	z	x	1	x	x

<b>xor</b>	0	1	x	z	<b>not</b>	input	output
0	0	1	x	x		0	1
1	1	0	x	x		1	0
x	x	x	x	x		x	x
z	x	x	x	x		z	x

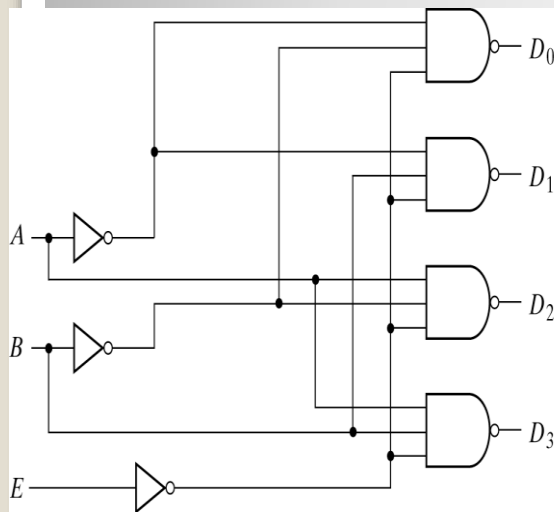


# Gate-level description on Verilog code

The **wire** declaration is for internal

## HDL Example 4-1

```
//Gate-level description of a 2-to-4-line decoder
//Figure 4-19
module decoder_g1 (A,B,E,D);
    input A,B,E;
    output [0:3]D;
    wire Anot,Bnot,Enot;
    not
        n1 (Anot,A),
        n2 (Bnot,B),
        n3 (Enot,E);
    nand
        n4 (D[0],Anot,Bnot,Enot),
        n5 (D[1],Anot,B,Enot),
        n6 (D[2],A,Bnot,Enot),
        n7 (D[3],A,B,Enot);
endmodule
```



E	A	B	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(a) Logic diagram

(b) Truth table

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

# Design methodologies

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

# A bottom-up hierarchical description

## HDL Example 4-2

---

```
//Gate-level hierarchical description of 4-bit adder
// Description of half adder (see Fig 4-5b)
module halfadder (S,C,x,y);
    input x,y;
    output S,C;
    //Instantiate primitive gates
    xor (S,x,y);
    and (C,x,y);
endmodule
```

# Full-adder

```
//Description of full adder (see Fig 4-8)
module fulladder (S,C,x,y,z);
    input x,y,z;
    output S,C;
    wire S1,D1,D2; //Outputs of first XOR and two AND gates
    //Instantiate the halfadder
        halfadder HA1 (S1,D1,x,y),
                    HA2 (S,D2,S1,z);
    or g1(C,D2,D1);
endmodule
```

# 4-bit adder

```
//Description of 4-bit adder (see Fig 4-9)
module _4bit_adder (S,C4,A,B,C0);
    input [3:0] A,B;
    input C0;
    output [3:0] S;
    output C4;
    wire C1,C2,C3; //Intermediate carries
    //Instantiate the fulladder
    fulladder FA0 (S[0],C1,A[0],B[0],C0),
               FA1 (S[1],C2,A[1],B[1],C1),
               FA2 (S[2],C3,A[2],B[2],C2),
               FA3 (S[3],C4,A[3],B[3],C3);

endmodule
```

# Three state gates

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

>> **bufif1**(OUT, A, control);

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

>> **notif0**(Y, B, enable);

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

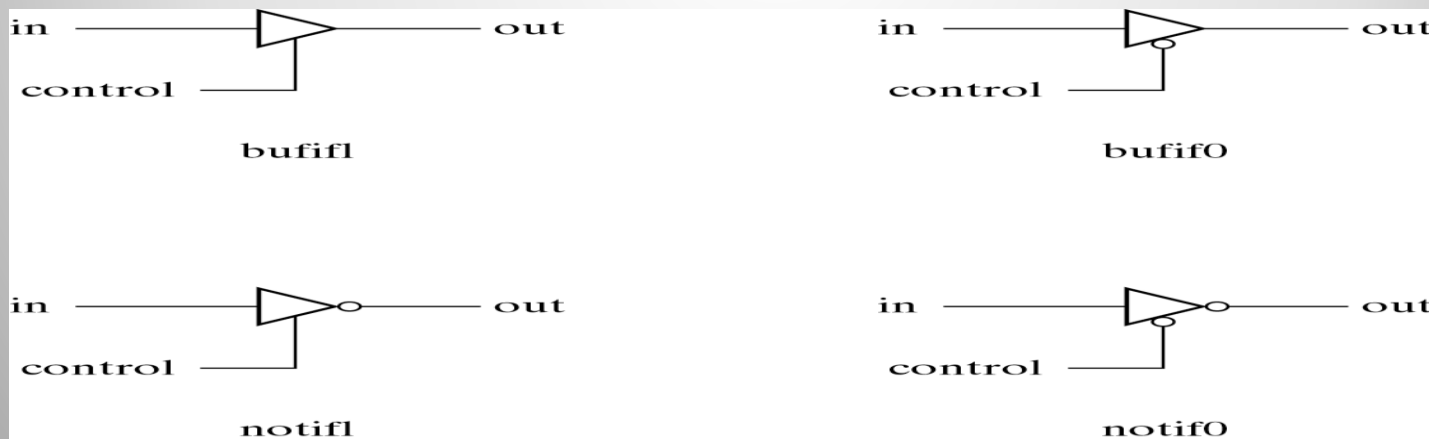


Fig. 4-31 Three-State Gates

# 2-to-1 multiplexer

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

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

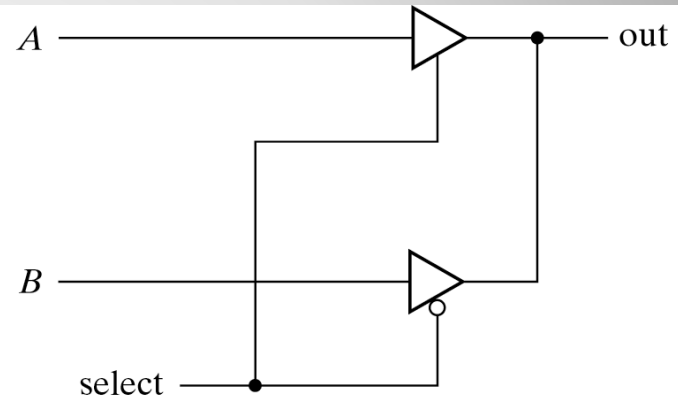


Fig. 4-32 2-to-1-Line Multiplexer with Three-State Buffers

# UNIT 4

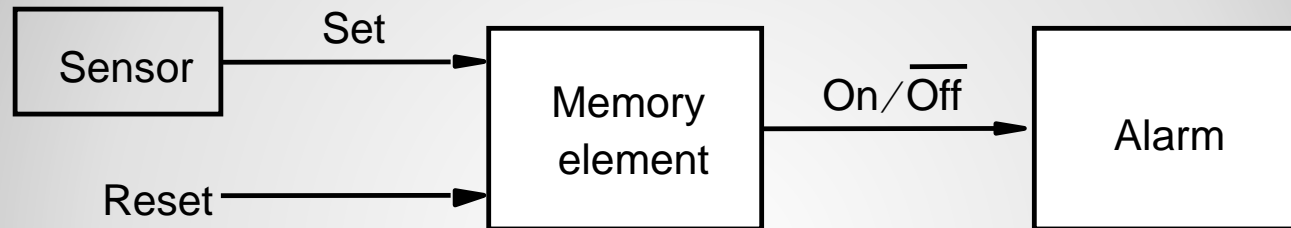
## SEQUENTIAL CIRCUITS



# Objectives

- In this chapter you will learn about:
  - Logic circuits that can store information
  - Flip-flops, which store a single bit
  - Registers, which store multiple bits
  - Shift registers, which shift the contents of a register
  - Counters of various types

# Motivation: Control of an Alarm System

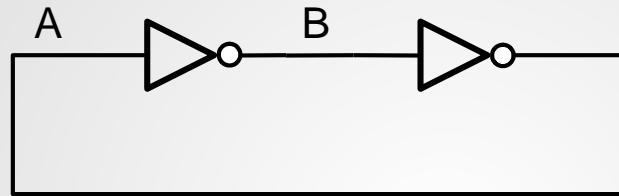


- Alarm turned on when On/Off = 1
- Alarm turned off when On/Off = 0
- Once triggered, alarm stays on until manually reset
- The circuit requires a memory element

# The Basic Latch

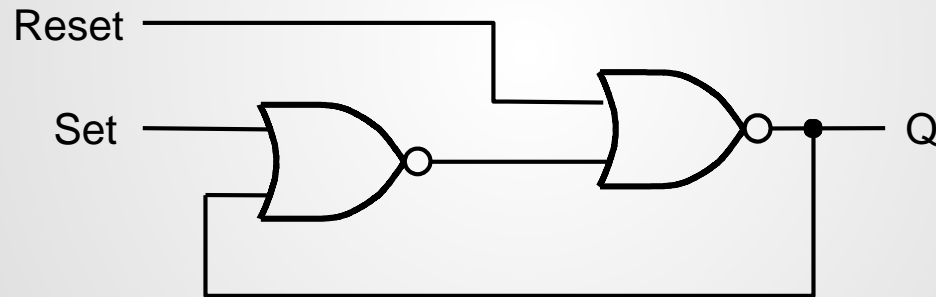
- **Basic latch** is a feedback connection of two NOR gates or two NAND gates
- It can store one bit of information
- It can be set to 1 using the *S* input and reset to 0 using the *R* input.

# A Simple Memory Element



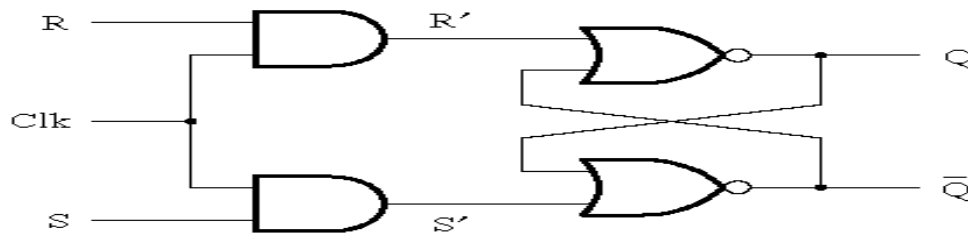
- A feedback loop with even number of inverters
- If  $A = 0$ ,  $B = 1$  or when  $A = 1$ ,  $B = 0$
- This circuit is not useful due to the lack of a mechanism for changing its state

# A Memory Element with NOR Gates



# The Gated Latch

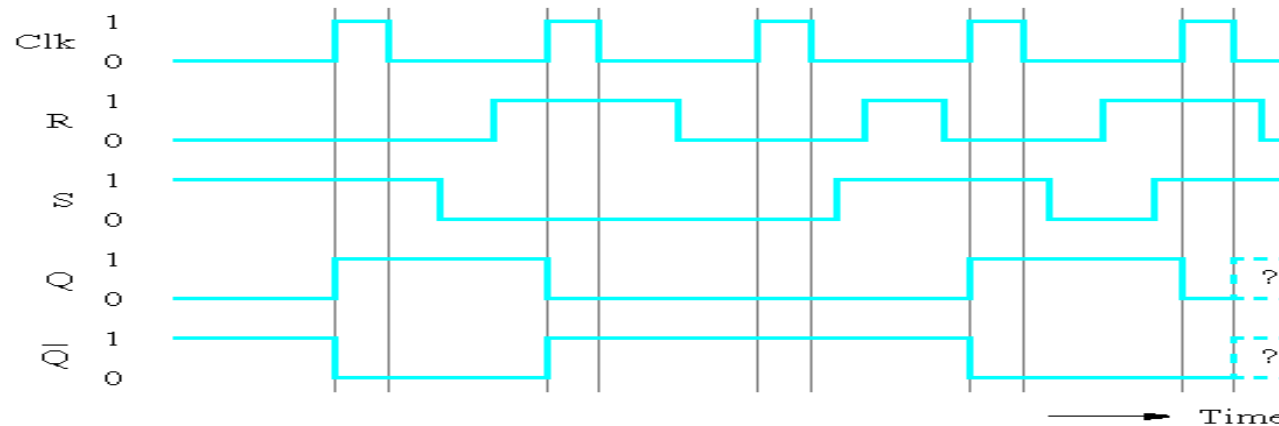
- **Gated latch** is a basic latch that includes input gating and a control signal
- The latch retains its existing state when the control input is equal to 0
- Its state may be changed when the control signal is equal to 1. In our discussion we referred to the control input as the clock
- We consider two types of gated latches:
  - **Gated SR latch** uses the  $S$  and  $R$  inputs to set the latch to 1 or reset it to 0, respectively.
  - **Gated D latch** uses the  $D$  input to force the latch into a state that has the same logic value as the  $D$  input.



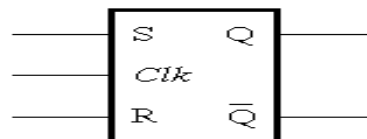
(a) Circuit

Clk	S	R	$Q(t+1)$
0	x	x	$Q(t)$ (no change)
1	0	0	$Q(t)$ (no change)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Characteristic table

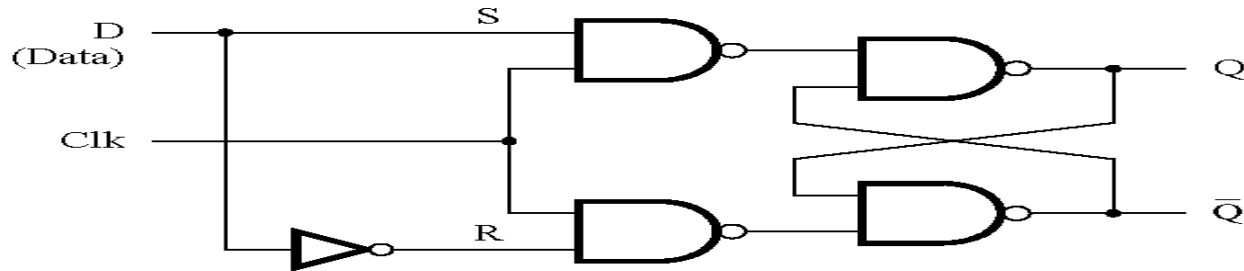


(c) Timing diagram



(d) Graphical symbol

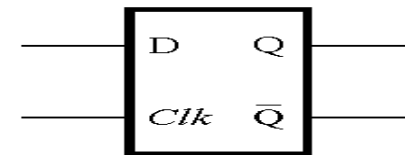
# Gated S/R Latch



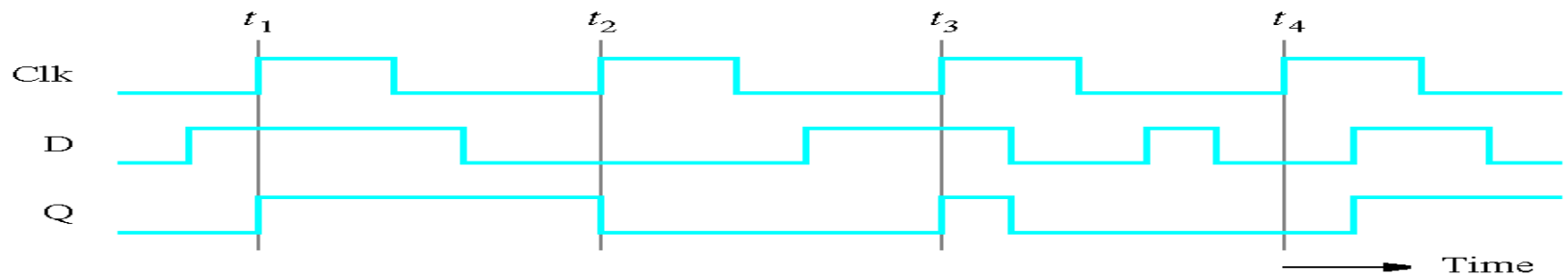
(a) Circuit

Clk	D	$Q(t+1)$
0	x	$Q(t)$
1	0	0
1	1	1

(b) Characteristic table



(c) Graphical symbol



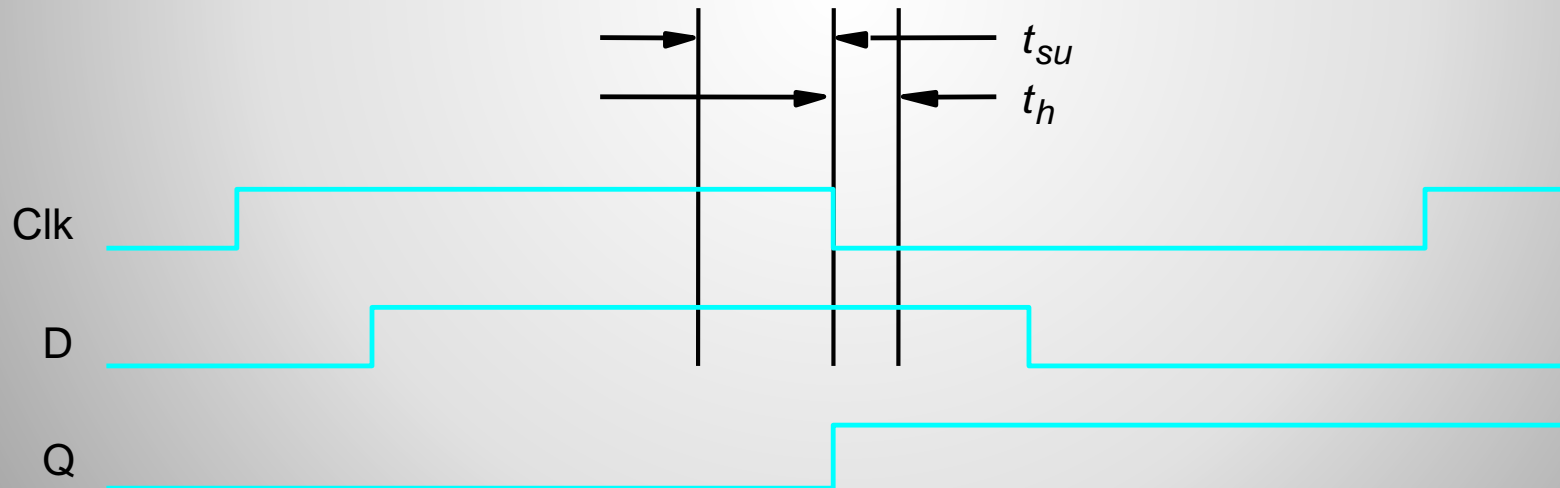
(d) Timing diagram

# Gated D Latch



# Setup and Hold Times

- Setup Time  $t_{su}$ 
  - The minimum time that the input signal must be stable prior to the edge of the clock signal.
- Hold Time  $t_h$ 
  - The minimum time that the input signal must be stable after the edge of the clock signal.

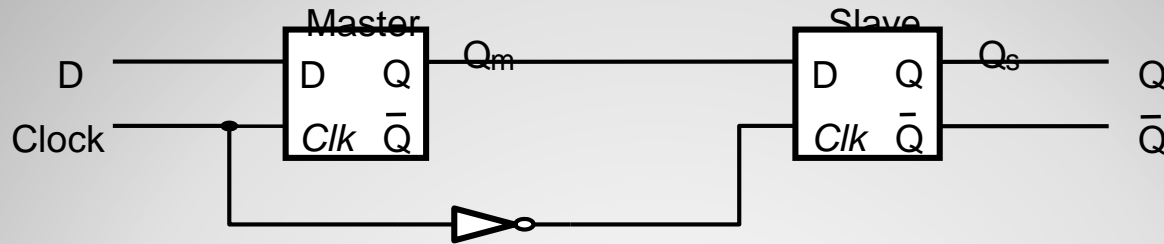


# Flip-Flops

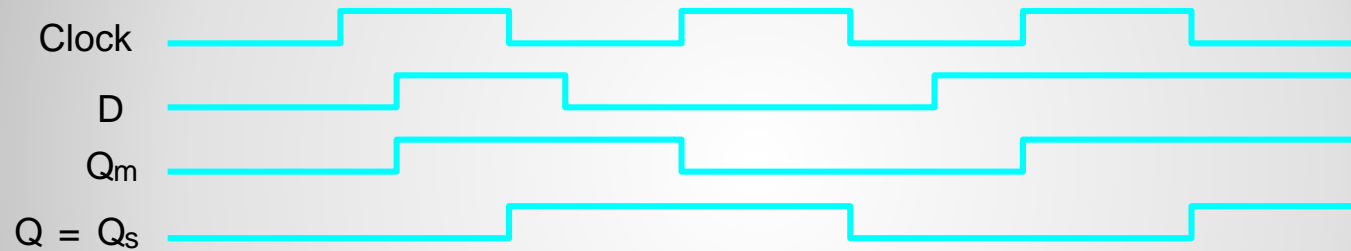
- A **flip-flop** is a storage element based on the gated latch principle
- It can have its output state changed only on the edge of the controlling clock signal

# Flip-Flops

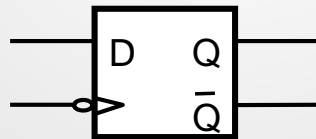
- We consider two types:
  - **Edge-triggered flip-flop** is affected only by the input values present when the active edge of the clock occurs
  - **Master-slave flip-flop** is built with two gated latches
    - The master stage is active during half of the clock cycle, and the slave stage is active during the other half.
    - The output value of the flip-flop changes on the edge of the clock that activates the transfer into the slave stage.



(a) Circuit



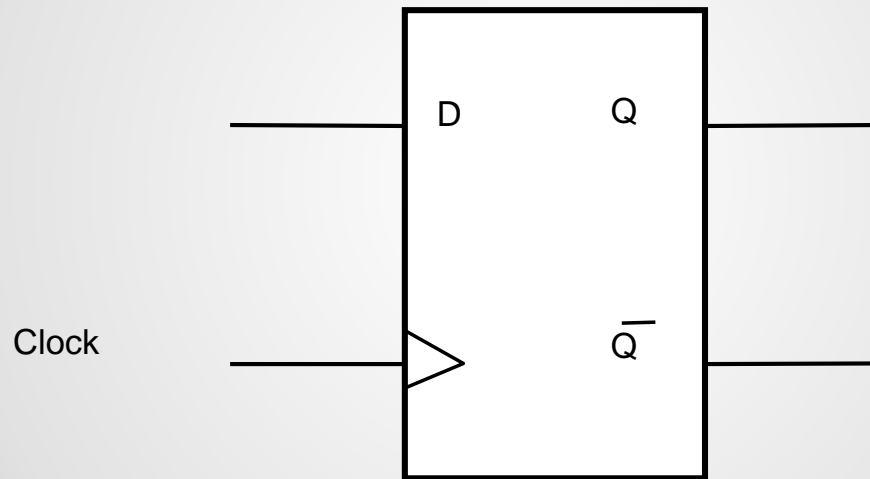
(b) Timing diagram



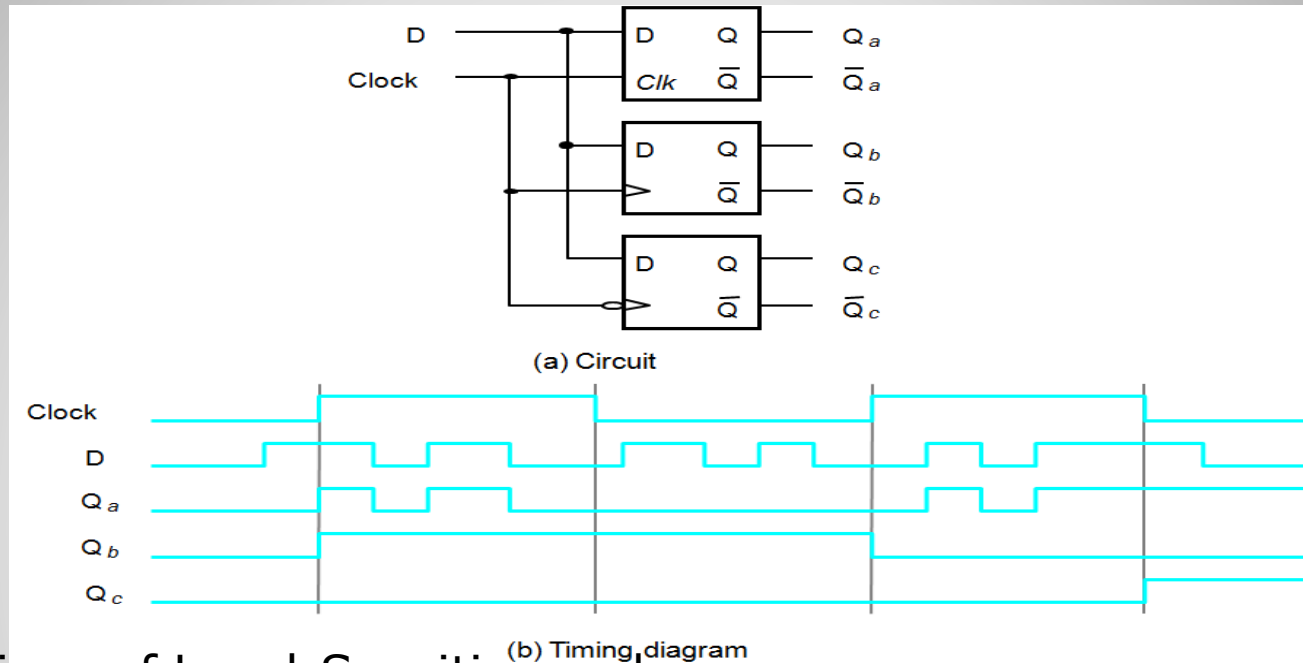
(c) Graphical symbol

# Master-Slave D Flip-Flop

# A Positive-Edge-Triggered D Flip-Flop

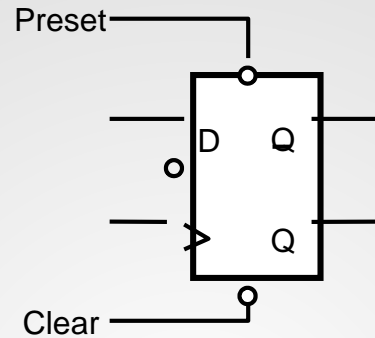


Graphical symbol

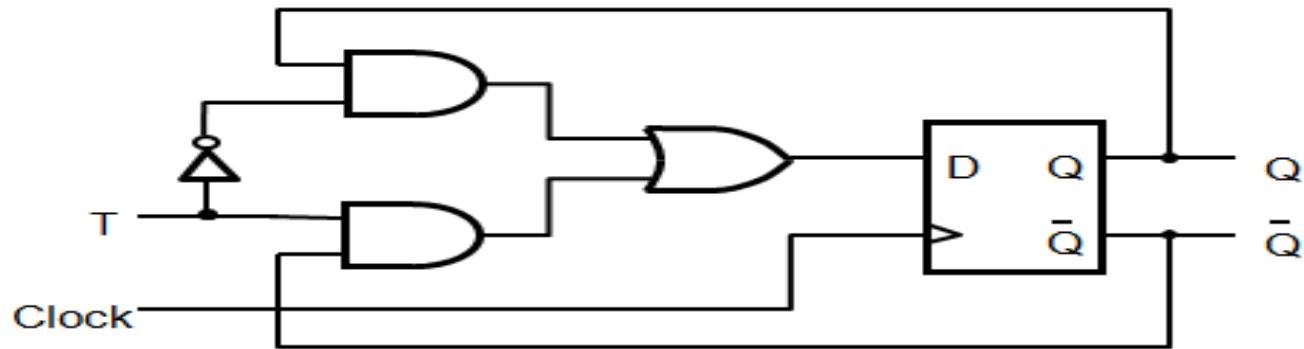


Comparison of Level-Sensitive and  
Edge-Triggered D Storage Elements

# Comparison of Level-Sensitive and Edge-Triggered D Storage Elements



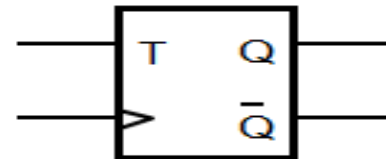
## **Master-Slave D Flip-Flop with *Clear and Preset***



(a) Circuit

T	$Q(t+1)$
0	$Q(t)$
1	$\bar{Q}(t)$

(b) Characteristic table



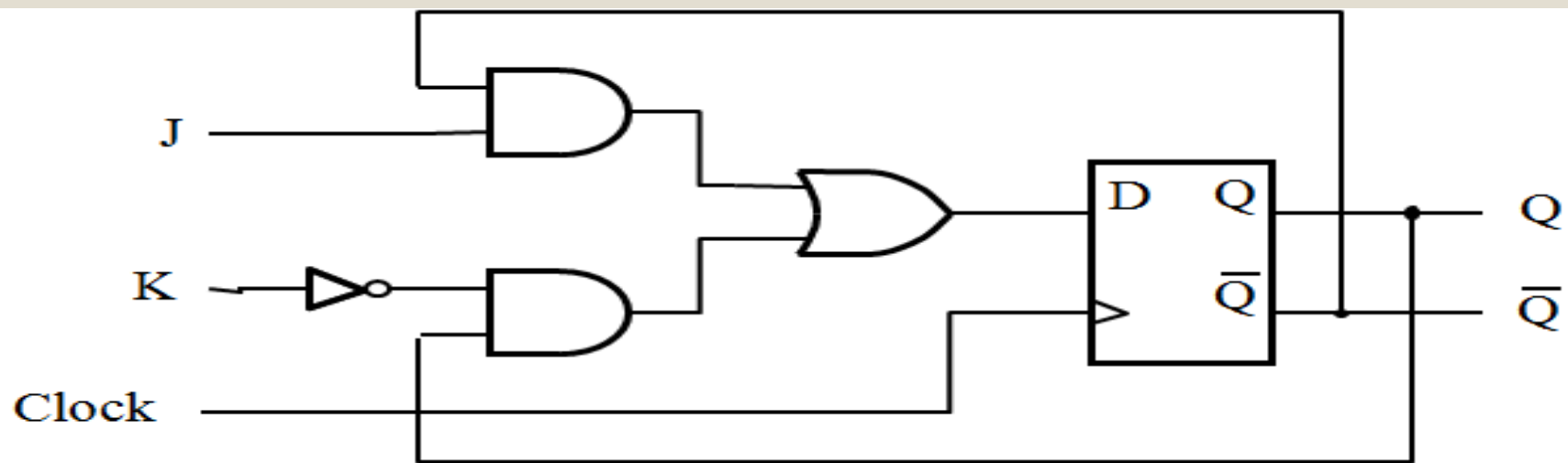
(c) Graphical symbol



(d) Timing diagram

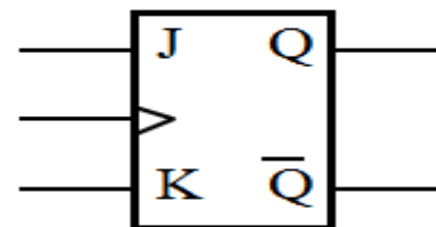
# T Flip-Flop





(a) Circuit

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\bar{Q}(t)$



(b) Characteristic table

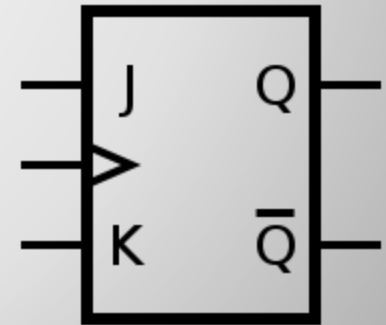
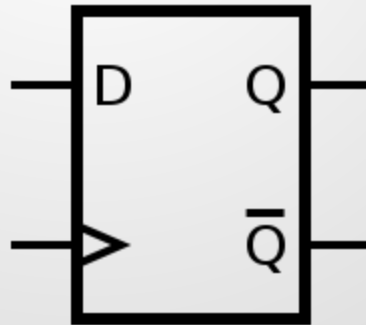
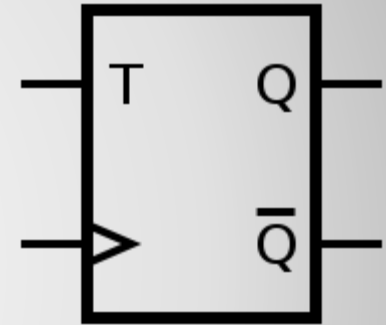
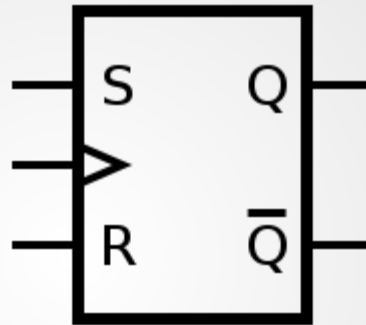
(c) Graphical symbol

# JK Flip-Flop

# Flip-flop excitation tables

# Types of Flip-flops

- SR flip-flop (Set, Reset)
- T flip-flop (Toggle)
- D flip-flop (Delay)
- JK flip-flop



# Excitation Tables

Previous State -> Present State	S	R
0 -> 0	0	X
0 -> 1	1	0
1 -> 0	0	1
1 -> 1	X	0

Previous State -> Present State	T
0 -> 0	0
0 -> 1	1
1 -> 0	1
1 -> 1	0

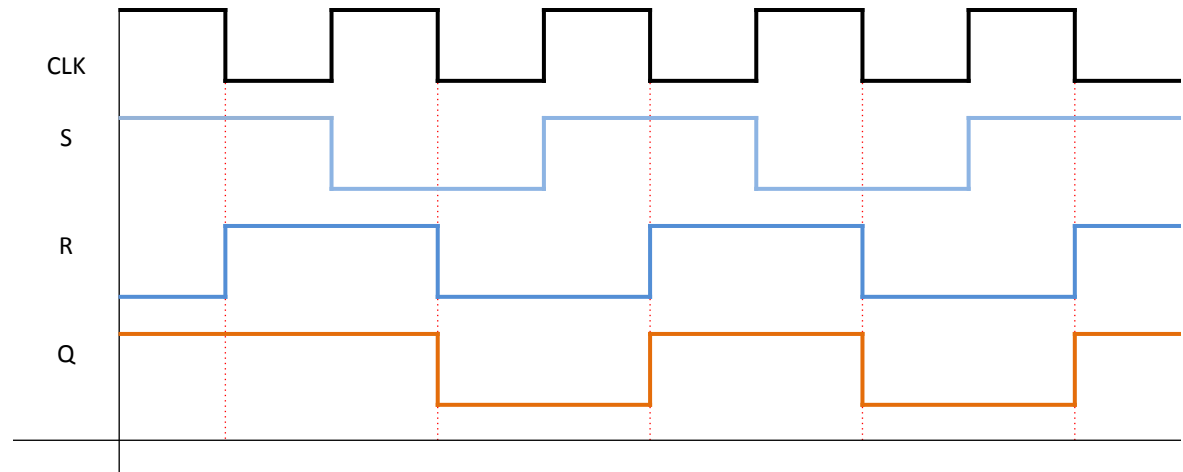
# Excitation Tables

Previous State -> Present State	D
0 -> 0	0
0 -> 1	1
1 -> 0	0
1 -> 1	1

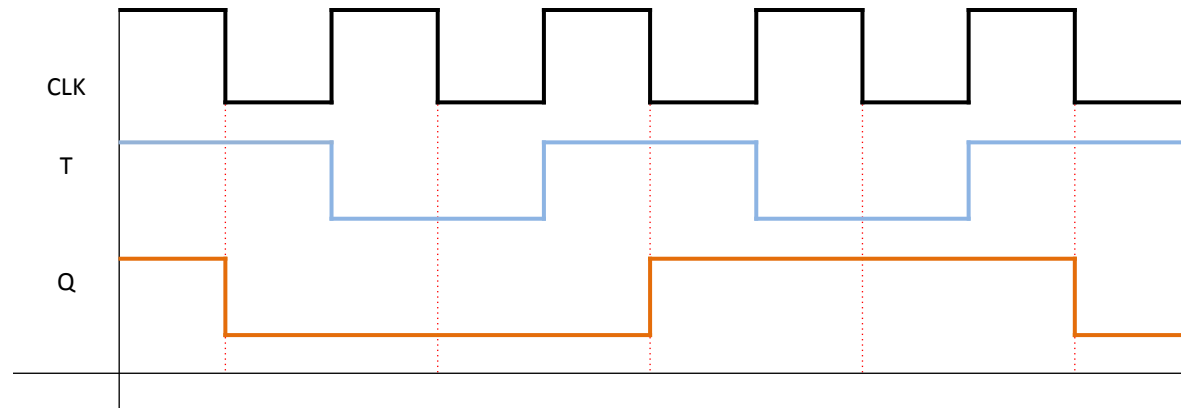
Previous State -> Present State	J	K
0 -> 0	0	X
0 -> 1	1	X
1 -> 0	X	1
1 -> 1	X	0

# Timing Diagrams

	S	R
0->0	0	X
0->1	1	0
1->0	0	1
1->1	X	0

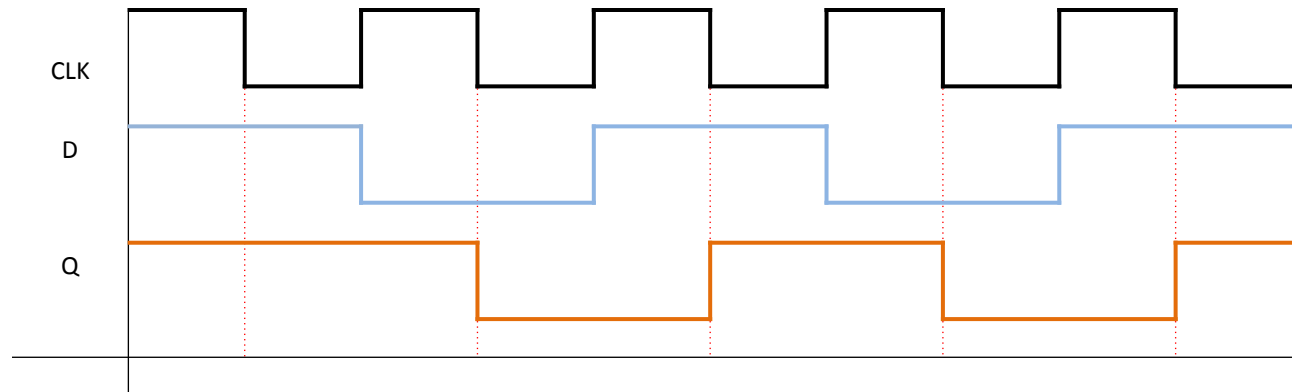


	T
0->0	0
0->1	1
1->0	1
1->1	0

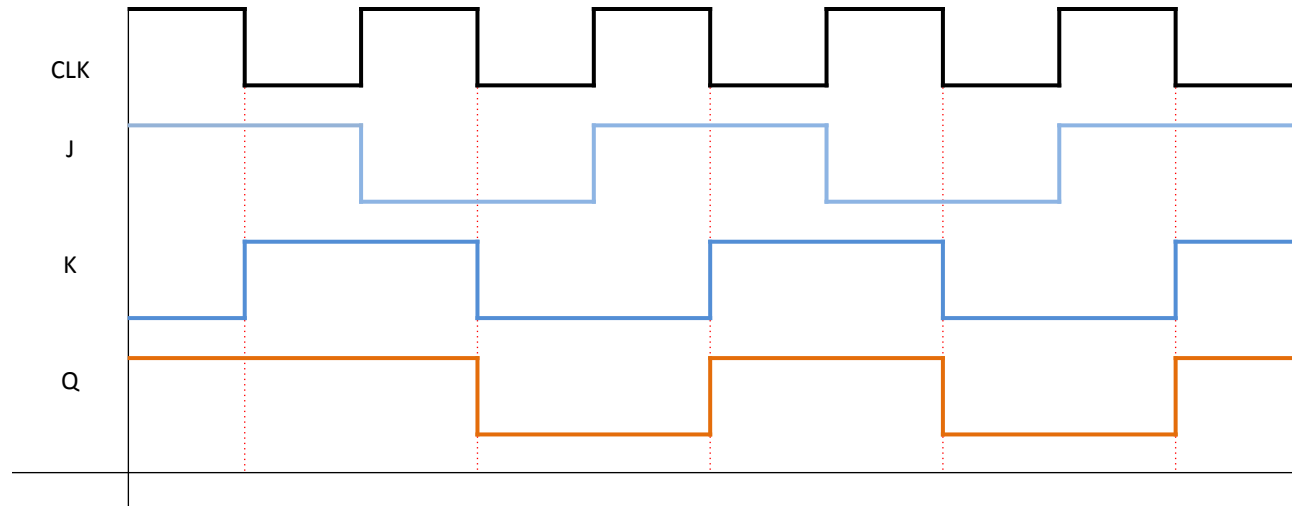


# Timing Diagrams

	D
0->0	0
0->1	1
1->0	0
1->1	1



	J	K
0->0	0	X
0->1	1	X
1->0	X	1
1->1	X	0

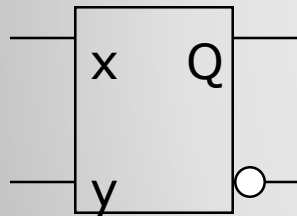


# Conversions of flipflops

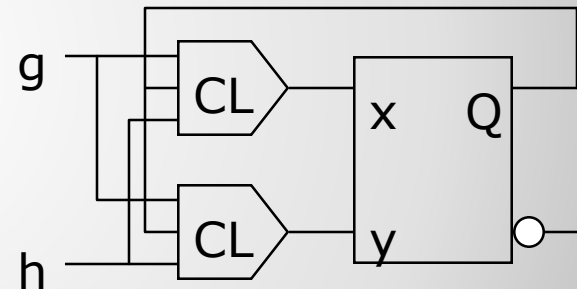
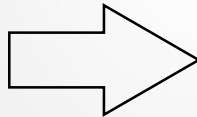
## Procedure uses excitation tables

**Method:** to realize a type **A** flipflop using a type **B** flipflop:

1. Start with the K-map or state-table for the A-flipflop.
2. Express B-flipflop inputs as a function of the inputs and present state of A-flipflop such that the required state transitions of A-flipflop are realized.



Type B



Type A

1. Find  $Q^+ = f(g,h,Q)$  for type A (using type A state-table)
2. Compute  $x = f1(g,h,Q)$  and  $y = f2(g,h,Q)$  to realize  $Q^+$ .



### Example: Use JK-FF to realize D-FF

- 1) Start transition table for D-FF
- 2) Create K-maps to express J and K as functions of inputs (D, Q)
- 3) Fill in K-maps with appropriate values for J and K to cause the same state transition as in the D-FF transition table

D	Q	Q <sup>+</sup>	J	K
0	0	0	0	X
0	1	0	X	1
1	0	1	1	X
1	1	1	X	0

#### State-Table

e.g.  
when  $D=Q=0$ , then  $Q^+=0$   
the same transition  $Q \rightarrow Q^+$   
is realized with  $J=0, K=X$

Q	Q <sup>+</sup>	R	S	J	K	T	D
0	0	X	0	0	X	0	0
0	1	0	1	1	X	1	1
1	0	1	0	X	1	1	0
1	1	0	X	X	0	0	1

		D	
		0	1
Q	0	0	1
1	X	X	

$$J = D$$

		D	
		0	1
Q	0	X	X
1	1	1	0

$$K = \overline{D}$$

## Example: Implement JK-FF using a D-FF

J	K	Q	Q+	D	T
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	1

JK \ Q	J			
	00	01	11	10
0	0	0	1	1
1	1	0	0	1

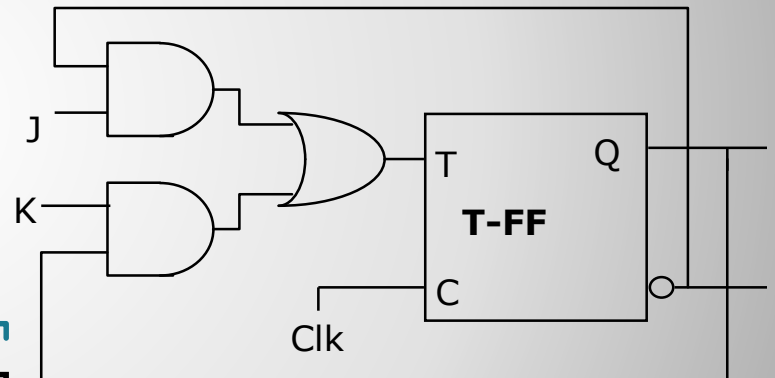
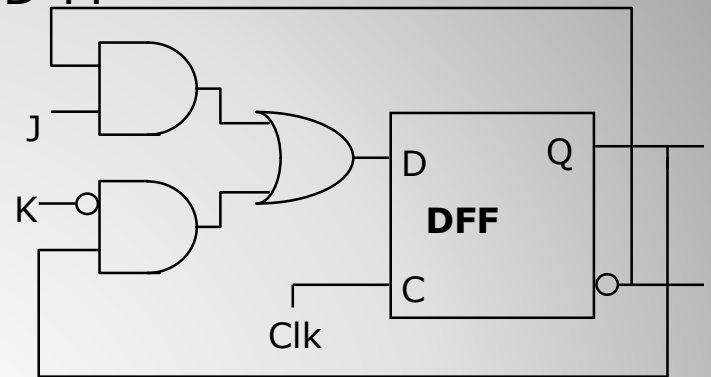
K

$$d = jQ + Kq$$

JK \ Q	J			
	00	01	11	10
0	0	0	1	1
1	0	1	1	0

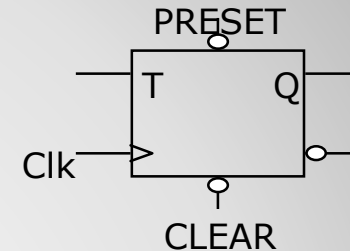
K

$$t = jQ + kq$$



# Asynchronous inputs

PRESET and CLEAR:  
asynchronous, level-sensitive inputs  
used to initialize a flipflop.

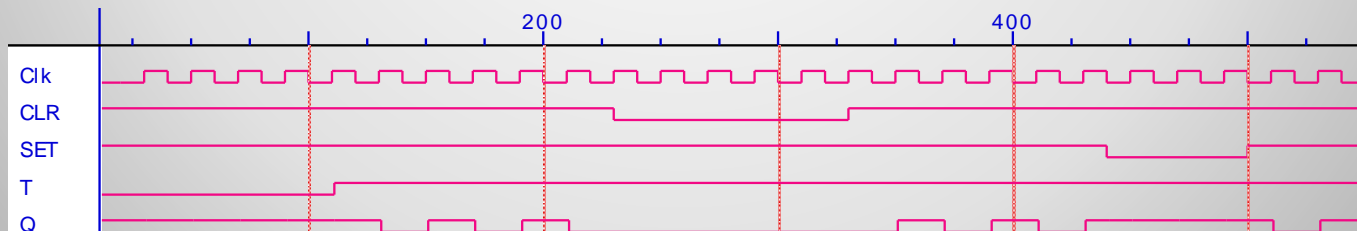
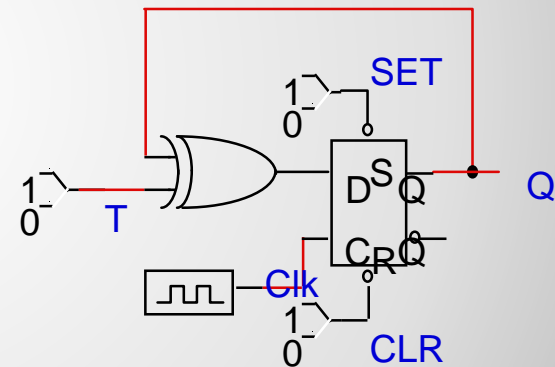


PRESET, CLEAR: active low inputs

PRESET = 0 --> Q = 1

CLEAR = 0 --> Q = 0

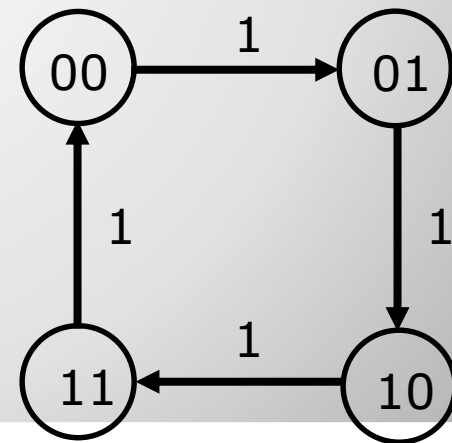
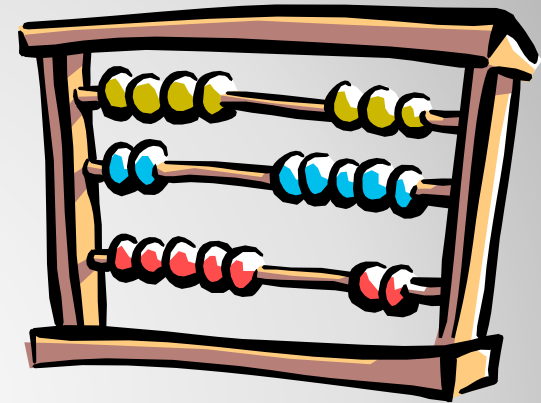
## LogicWorks Simulation



# Counters

- Counters are a specific type of sequential circuit.
- Like registers, the state, or the flip-flop values themselves, serves as the "output."
- The output value increases by one on each clock cycle.
- After the largest value, the output "wraps around" back to 0.
- Using two bits, we'd get something like this:

Present State		Next State	
A	B	A	B
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0



# Benefits of counters

- Counters can act as simple clocks to keep track of “time.”
- You may need to record how many times something has happened.
  - How many bits have been sent or received?
  - How many steps have been performed in some computation?
- All processors contain a **program counter**, or **PC**.
  - Programs consist of a list of instructions that are to be executed one after another (for the most part).
  - The PC keeps track of the instruction currently being executed.
  - The PC increments once on each clock cycle, and the next program instruction is then executed.

- Let's try to design a slightly different two-bit counter:
  - Again, the counter outputs will be 00, 01, 10 and 11.
  - Now, there is a single input, X. When X=0, the counter value should *increment* on each clock cycle. But when X=1, the value should *decrement* on successive cycles.
- We'll need two flip-flops again. Here are the four possible states:

00

01

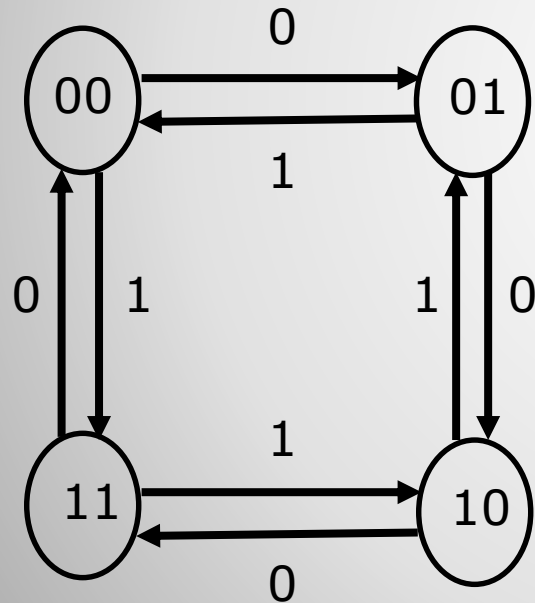
11

10

**A slightly fancier counter**

# The complete state diagram and table

- Here's the complete state diagram and state table for this circuit.



Present State		Inputs X	Next State	
Q <sub>1</sub>	Q <sub>0</sub>		Q <sub>1</sub>	Q <sub>0</sub>
0	0	0	0	1
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0

# D flip-flop inputs

- If we use D flip-flops, then the D inputs will just be the same as the desired next states.
- Equations for the D flip-flop inputs are shown at the right.
- Why does  $D_0 = Q_0'$  make sense?

Present State		Inputs X	Next State	
$Q_1$	$Q_0$		$Q_1$	$Q_0$
0	0	0	0	1
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0

			$Q_0$	
	0	1	0	1
$Q_1$	1	0	1	0
		X		

$$D_1 = Q_1 \oplus Q_0 \oplus X$$

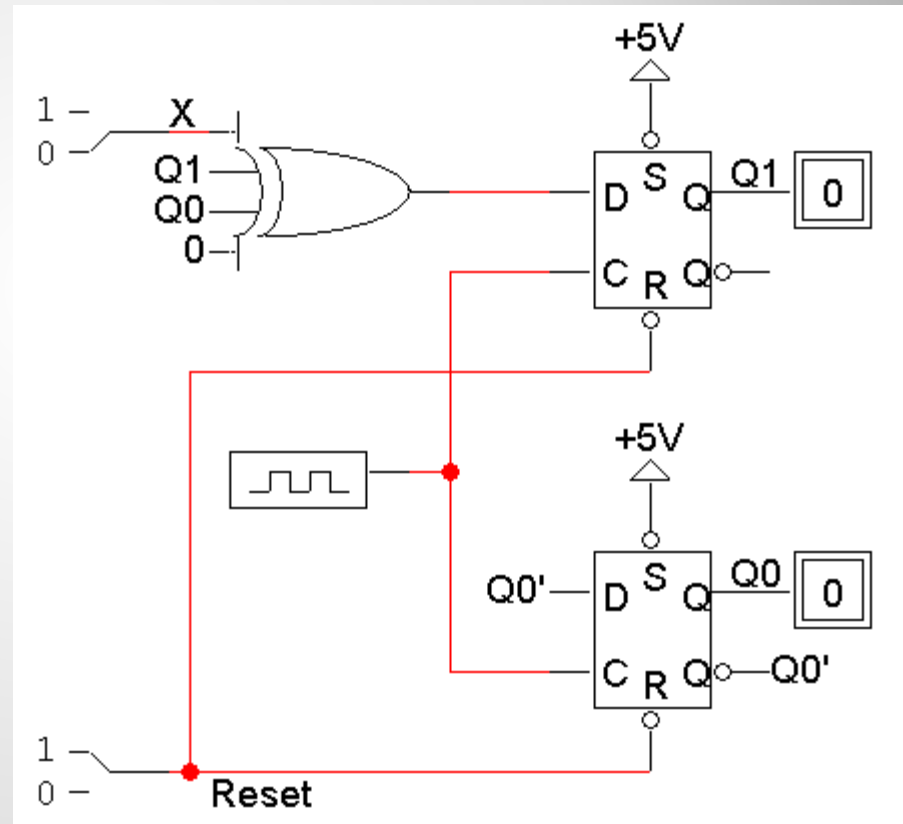
			$Q_0$	
	1	1	0	0
$Q_1$	1	1	0	0
		X		

$$D_0 = Q_0'$$



# The counter in LogicWorks

- Here are some **D Flip Flop** devices from LogicWorks.
- They have both normal and complemented outputs, so we can access **Q0'** directly without using an inverter. (Q1' is not needed in this example.)
- This circuit counts normally when **Reset = 1**. But when Reset is 0, the flip-flop outputs are cleared to 00 immediately.
- There is no three-input XOR gate in LogicWorks so we've used a four-input version instead, with one of the inputs connected to 0.



- If we use JK flip-flops instead, then we have to compute the JK inputs for each flip-flop.
- Look at the present and desired next state, and use the excitation table on the right.

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

Present State		Inputs	Next State		Flip flop inputs			
Q <sub>1</sub>	Q <sub>0</sub>	X	Q <sub>1</sub>	Q <sub>0</sub>	J <sub>1</sub>	K <sub>1</sub>	J <sub>0</sub>	K <sub>0</sub>
0	0	0	0	1	0	x	1	x
0	0	1	1	1	1	x	1	x
0	1	0	1	0	1	x	x	1
0	1	1	0	0	0	x	x	1
1	0	0	1	1	x	0	1	x
1	0	1	0	1	x	1	1	x
1	1	0	0	0	x	1	x	1
1	1	1	1	0	x	0	x	1

**JK flip-flop inputs**

# JK flip-flop input equations

Present State		Inputs	Next State		Flip flop inputs			
$Q_1$	$Q_0$		$Q_1$	$Q_0$	$J_1$	$K_1$	$J_0$	$K_0$
0	0	0	0	1	0	x	1	x
0	0	1	1	1	1	x	1	x
0	1	0	1	0	1	x	x	1
0	1	1	0	0	0	x	x	1
1	0	0	1	1	x	0	1	x
1	0	1	0	1	x	1	1	x
1	1	0	0	0	x	1	x	1
1	1	1	1	0	x	0	x	1

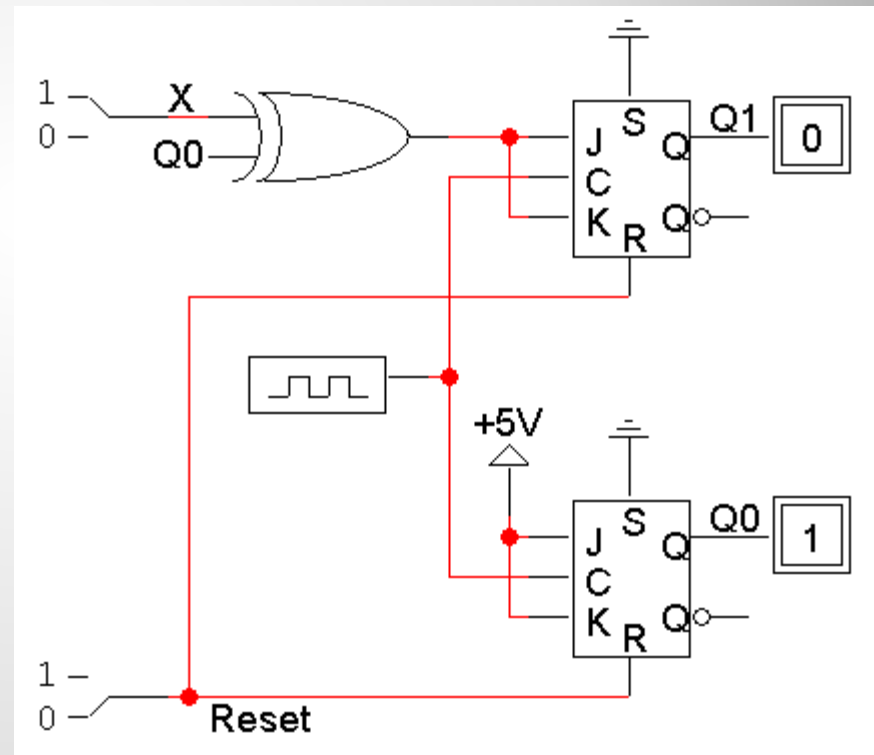
- We can then find equations for all four flip-flop inputs, in terms of the present state and inputs. Here, it turns out  $J_1 = K_1$  and  $J_0 = K_0$ .

$$J_1 = K_1 = Q_0' X + Q_0 X'$$

$$J_0 = K_0 = 1$$

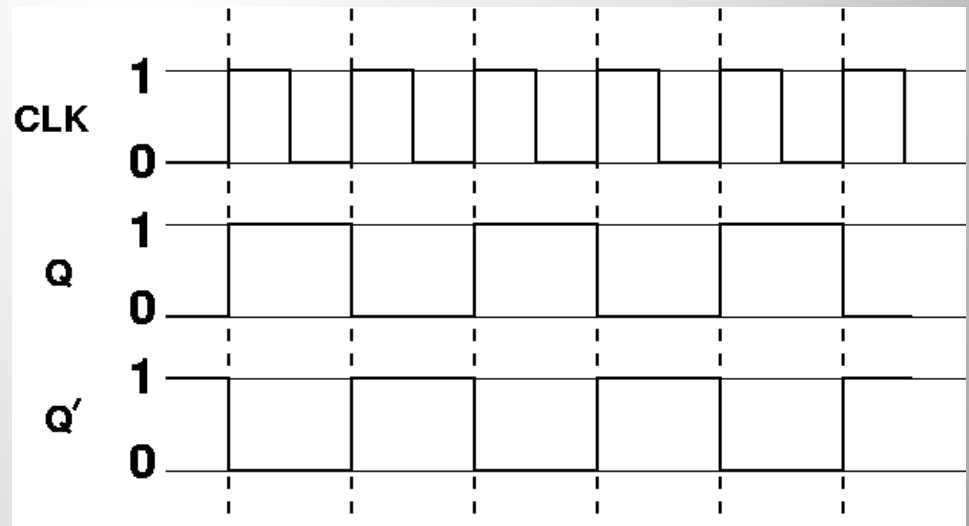
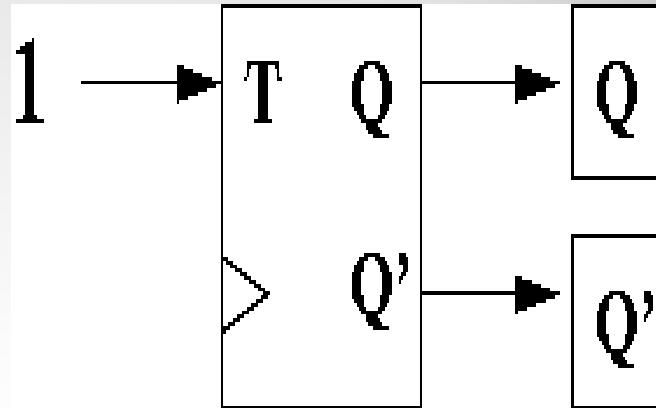
# The counter in LogicWorks again

- Here is the counter again, but using **JK Flip Flop n.i. RS** devices instead.
- The direct inputs R and S are non-inverted, or active-high.
- So this version of the circuit counts normally when **Reset = 0**, but initializes to 00 when Reset is 1.



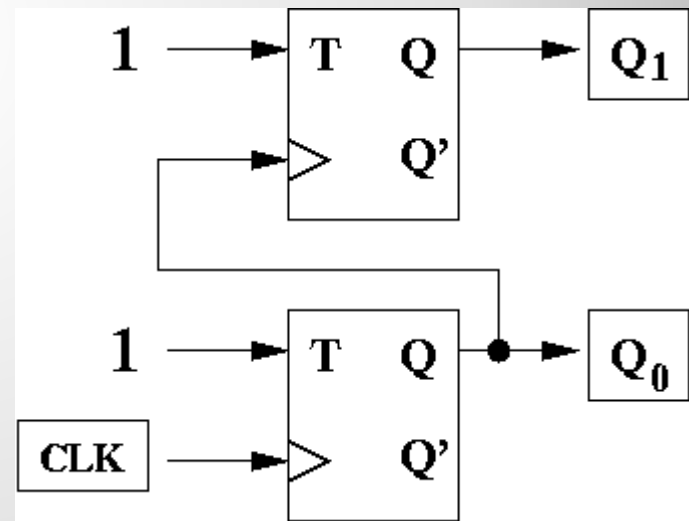
# Asynchronous Counters

- This counter is called *asynchronous* because not all flip flops are hooked to the same clock.
- Look at the waveform of the output, **Q**, in the timing diagram. It resembles a clock as well. If the period of the clock is  $T$ , then what is the period of **Q**, the output of the flip flop? It's  $2T$ !
- We have a way to create a clock that runs twice as slow. We feed the clock into a T flip flop, where T is hardwired to 1. The output will be a clock whose period is twice as long.



# Asynchronous counters

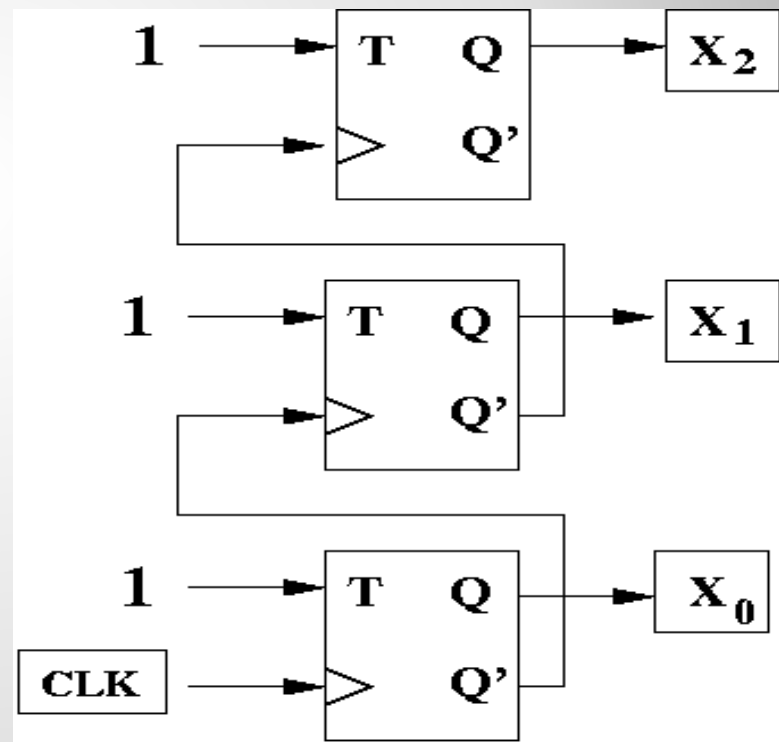
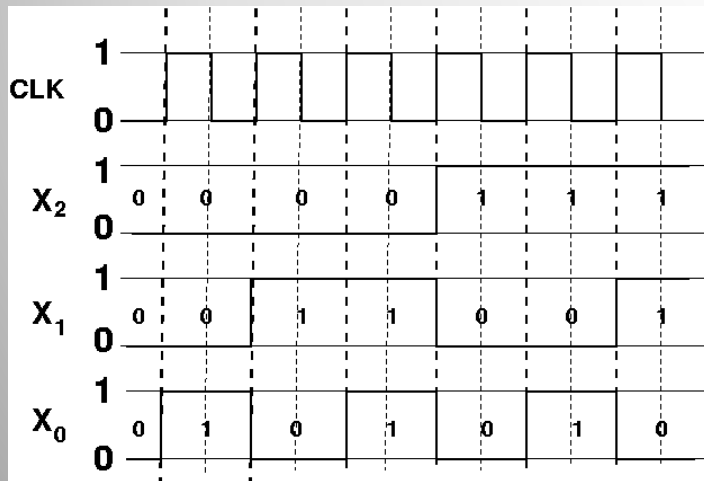
If the clock has period  $T$ . **Q0** has period  $2T$ . **Q1** period is  $4T$   
With  $n$  flip flops the period is  $2^n$ .



# Registers, Counters, State Reduction

# 3 bit asynchronous "ripple" counter using T flip flops

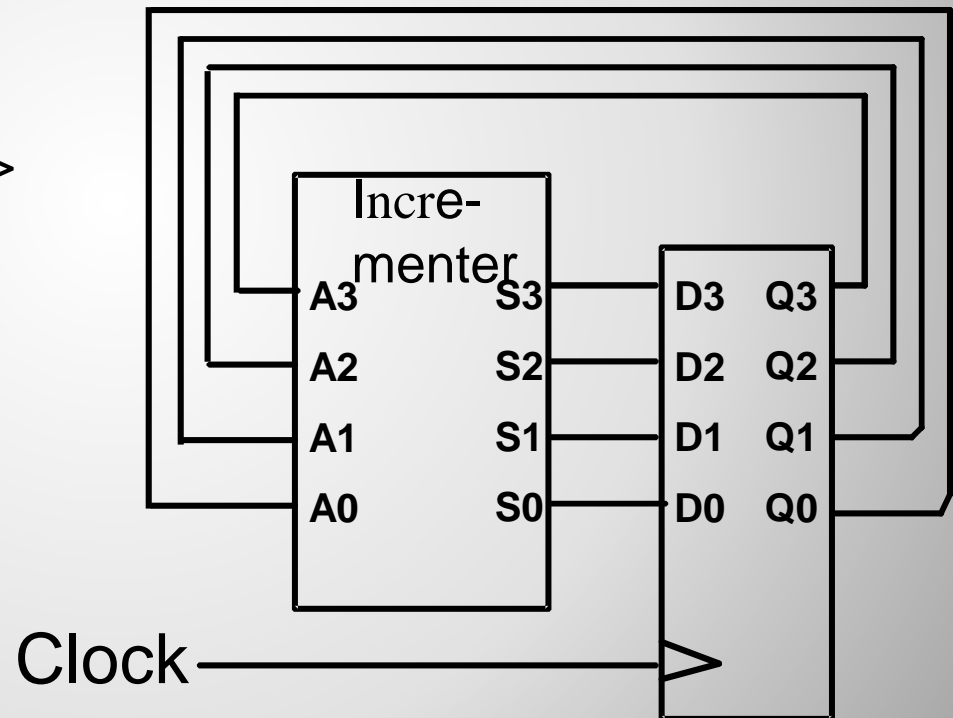
- This is called as a *ripple counter* due to the way the FFs respond one after another in a kind of rippling effect.



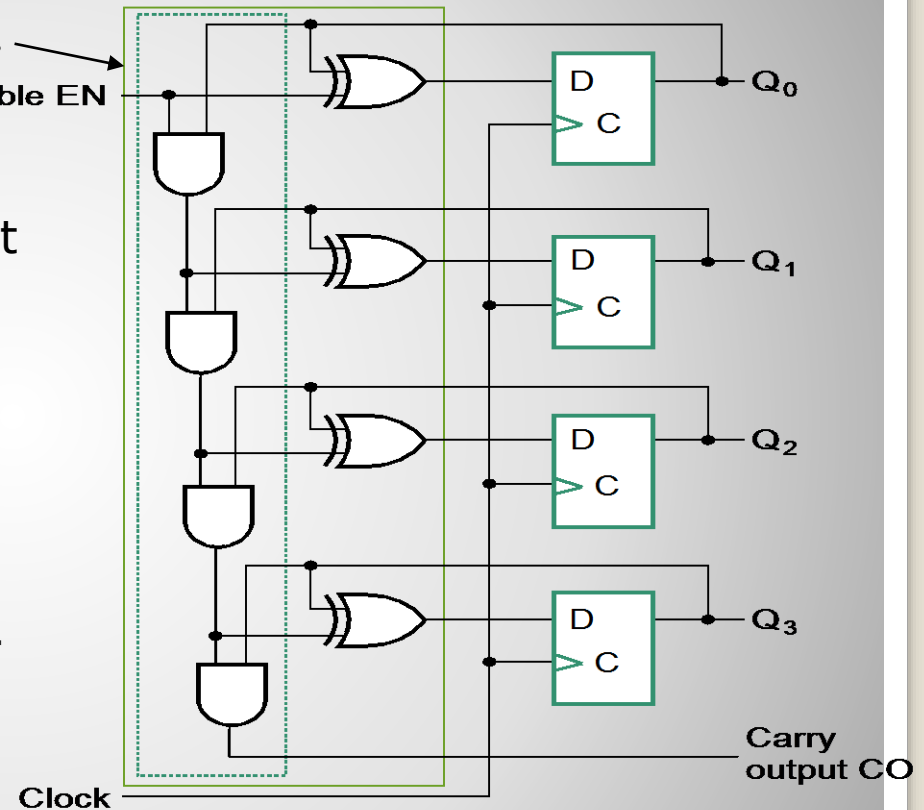


# Synchronous Counters

- To eliminate the "ripple" effects, use a common clock for each flip-flop and a combinational circuit to generate the next state.
- For an up-counter, use an incrementer =>



- Internal details =>
- Internal Logic
  - XOR complements each bit
  - AND chain causes complement of a bit if all bits toward LSB from it equal 1
- Count Enable
  - Forces all outputs of AND chain to 0 to "hold" the state
- Carry Out
  - Added as part of incrementer
  - Connect to Count Enable of additional 4-bit counters to form larger counters



(a) Logic Diagram-Serial Gating

## Synchronous Counters (continued)

# Design Example: Synchronous BCD

- Use the sequential logic model to design a synchronous BCD counter with D flip-flops
- State Table =>
- Input combinations 1010 through 1111 are don't cares

Current State				Next State			
Q8	Q4	Q2	Q1	Q8	Q4	Q2	Q1
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0

- Use K-Maps to two-level optimize the next state equations and manipulate into forms containing XOR gates:

$$D1 = Q1'$$

- $D2 = Q2 + Q1Q8'$

$$D4 = Q4 + Q1Q2$$

$$D8 = Q8 + (Q1Q8 + Q1Q2Q4)$$

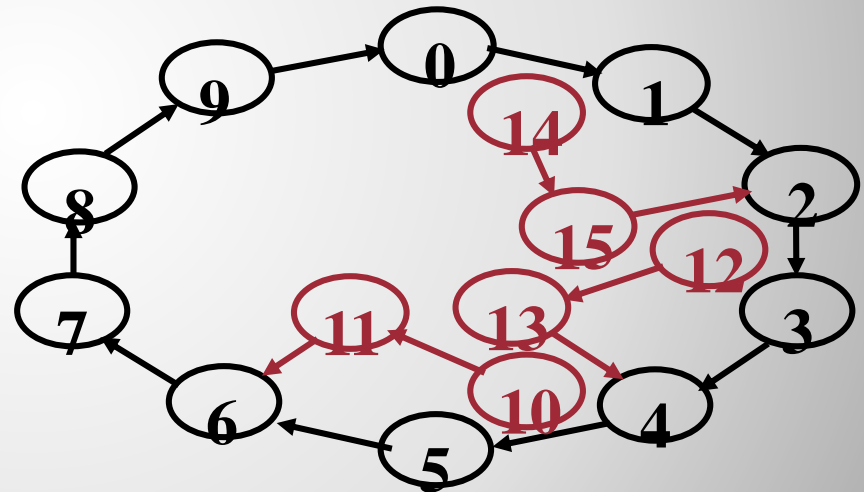
- $Y = Q1Q8$

- The logic diagram can be drawn from these equations
  - An asynchronous or synchronous reset should be added
- What happens if the counter is perturbed by a power disturbance or other interference and it enters a state other than 0000 through 1001?

## Synchronous BCD (continued)

- Find the actual values of the six next states for the don't care combinations from the equations
- Find the overall state diagram to assess behavior for the don't care states (states in decimal)

Present State	Next State
Q8 Q4 Q2 Q1	Q8 Q4 Q2 Q1
1 0 1 0	1 0 1 1
1 0 1 1	0 1 1 0
1 1 0 0	1 1 0 1
1 1 0 1	0 1 0 0
1 1 1 0	1 1 1 1
1 1 1 1	0 0 1 0



# Synchronous BCD (continued)

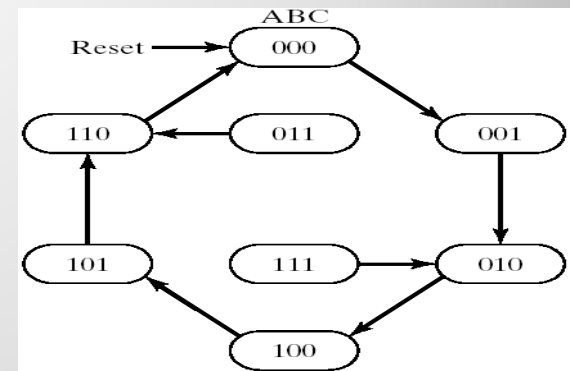
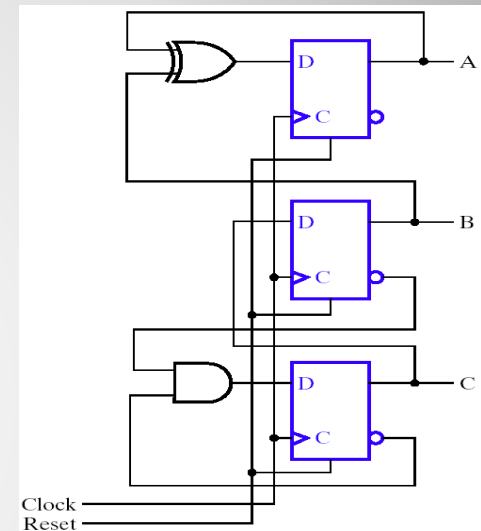
- For the BCD counter design, if an invalid state is entered, return to a valid state occurs within two clock cycles
- Is this adequate?!

**Synchronous BCD (continued)**

# Counting an arbitrary sequence

□ **TABLE 7-10**  
**State Table and Flip-Flop Inputs for Counter**

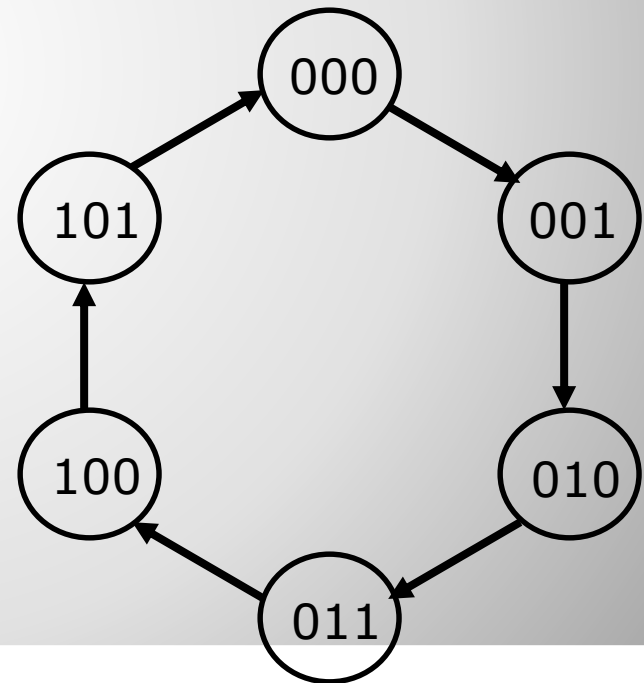
Present State			Next State		
A	B	C	DA = A(t+1)	DB = B(t+1)	DC = C(t+1)
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	0	0	0



# Unused states

- The examples shown so far have all had  $2^n$  states, and used  $n$  flip-flops. But sometimes you may have unused, leftover states.
- For example, here is a state table and diagram for a counter that repeatedly counts from 0 (000) to 5 (101).
- What should we put in the table for the two unused states?

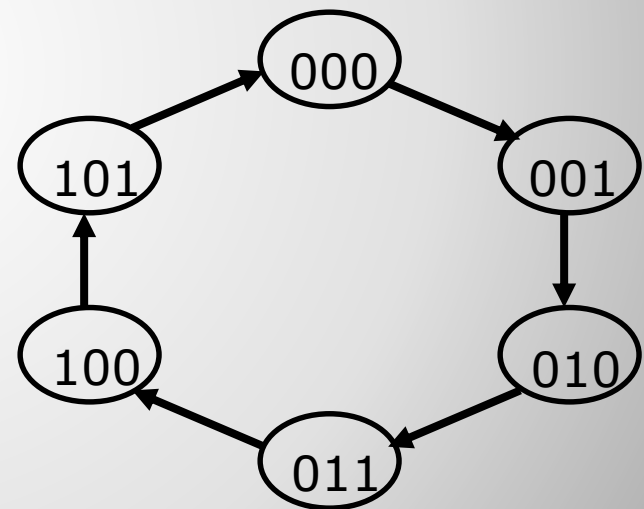
Present State			Next State		
$Q_2$	$Q_1$	$Q_0$	$Q_2$	$Q_1$	$Q_0$
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	?	?	?
1	1	1	?	?	?





- To get the *simplest* possible circuit, you can fill in don't cares for the next states. This will also result in don't cares for the flip-flop inputs, which can simplify the hardware.
- If the circuit somehow ends up in one of the unused states (110 or 111), its behavior will depend on exactly what the don't cares were filled in with.

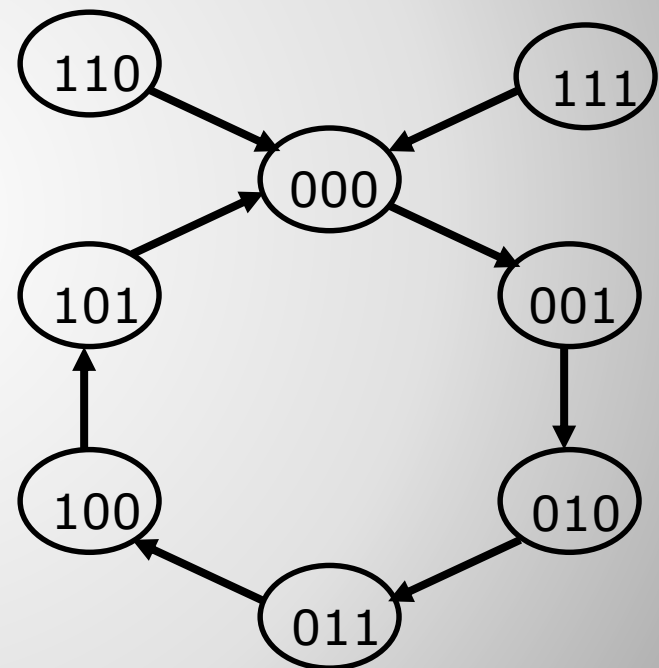
Present State			Next State		
$Q_2$	$Q_1$	$Q_0$	$Q_2$	$Q_1$	$Q_0$
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	x	x	x
1	1	1	x	x	x



**Unused states can be don't cares...**

- To get the *safest* possible circuit, you can explicitly fill in next states for the unused states 110 and 111.
- This guarantees that even if the circuit somehow enters an unused state, it will eventually end up in a valid state.
- This is called a **self-starting counter**.

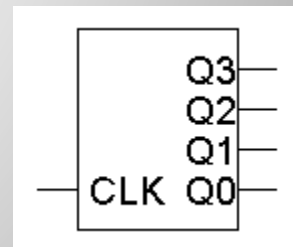
Present State			Next State		
$Q_2$	$Q_1$	$Q_0$	$Q_2$	$Q_1$	$Q_0$
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	0	0



...or maybe you do care

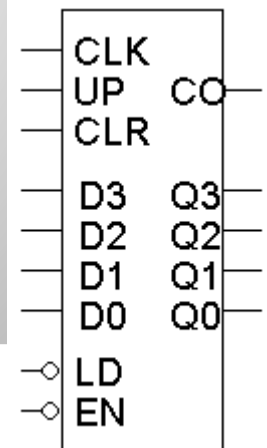
# LogicWorks counters

- There are a couple of different counters available in LogicWorks.
- The simplest one, the **Counter-4 Min**, just increments once on each clock cycle.
  - This is a four-bit counter, with values ranging from 0000 to 1111.
  - The only "input" is the clock signal.



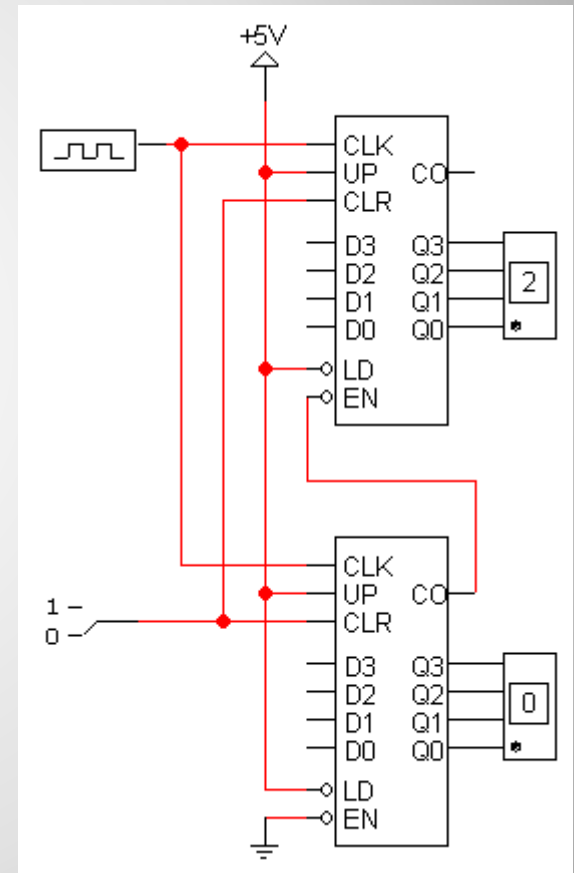
# More complex counters

- More complex counters are also possible. The full-featured LogicWorks **Counter-4** device below has several functions.
  - It can increment or decrement, by setting the **UP** input to 1 or 0.
  - You can immediately (asynchronously) clear the counter to 0000 by setting **CLR** = 1.
  - You can specify the counter's next output by setting **D<sub>3</sub>-D<sub>0</sub>** to any four-bit value and clearing **LD**.
  - The active-low **EN** input enables or disables the counter.
    - When the counter is disabled, it continues to output the same value without incrementing, decrementing, loading, or clearing.
  - The "counter out" **CO** is normally 1, but becomes 0 when the counter reaches its maximum value, 1111.



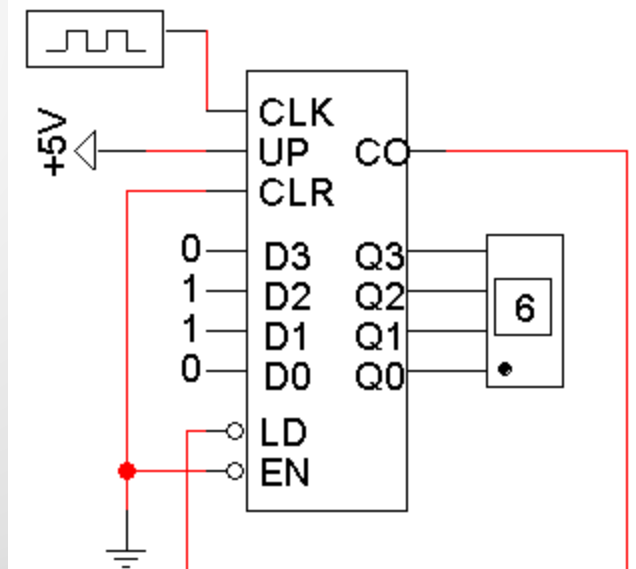
# An 8-bit counter

- As you might expect by now, we can use these general counters to build other counters.
- Here is an 8-bit counter made from two 4-bit counters.
  - The bottom device represents the least significant four bits, while the top counter represents the most significant four bits.
  - When the bottom counter reaches 1111 (i.e., when  $CO = 0$ ), it enables the top counter for one cycle.
- Other implementation notes:
  - The counters share clock and clear signals.



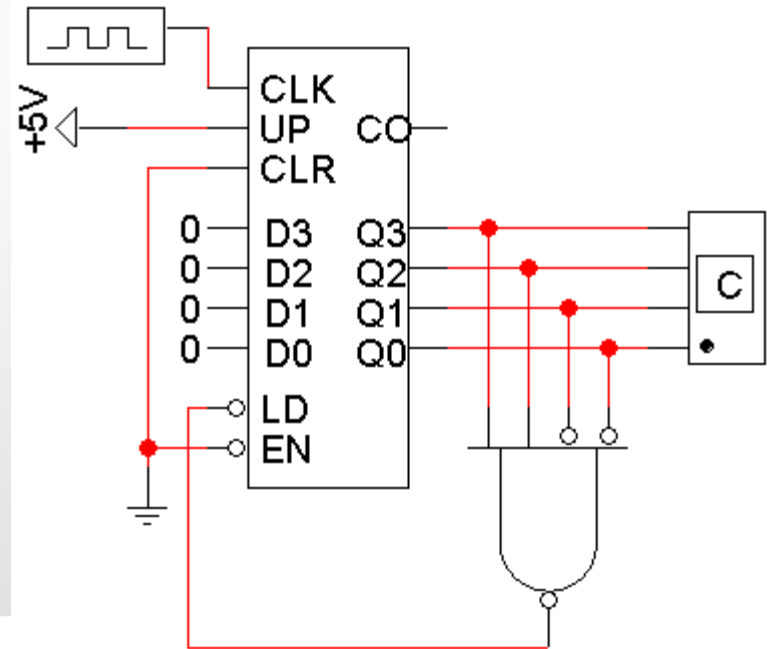
# A restricted 4-bit counter

- We can also make a counter that “starts” at some value besides 0000.
- In the diagram below, when  $CO=0$  the LD signal forces the next state to be loaded from  $D_3$ - $D_0$ .
- The result is this counter wraps from 1111 to 0110 (instead of 0000).



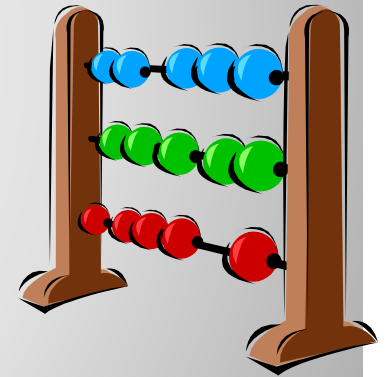
# Another restricted counter

- We can also make a circuit that counts up to only 1100, instead of 1111.
- Here, when the counter value reaches 1100, the NAND gate forces the counter to load, so the next state becomes 0000.



# Summary of Counters

- Counters serve many purposes in sequential logic design.
- There are lots of variations on the basic counter.
  - Some can increment or decrement.
  - An enable signal can be added.
  - The counter's value may be explicitly set.
- There are also several ways to make counters.
  - You can follow the sequential design principles to build counters from scratch.
  - You could also modify or combine existing counter devices.





# Sequential Circuit Design

Creating a sequential circuit to address a design need.

# Sequential Circuit Design

- Steps in the design process for sequential circuits
- State Diagrams and State Tables
- Examples

# Sequential Circuit Design Process

- Steps in Design of a Sequential Circuit
  - 1. Specification – A description of the sequential circuit. Should include a detailing of the inputs, the outputs, and the operation. Possibly assumes that you have knowledge of digital system basics.
  - 2. Formulation: Generate a state diagram and/or a state table from the statement of the problem.
  - 3. State Assignment: From a state table assign binary codes to the states.
  - 4. Flip-flop Input Equation Generation: Select the type of flip-flop for the circuit and generate the needed input for the required state transitions

# Sequential Circuit Design Process

## 2

- 5. Output Equation Generation: Derive output logic equations for generation of the output from the inputs and current state.
- 6. Optimization: Optimize the input and output equations. Today, CAD systems are typically used for this in real systems.
- 7. Technology Mapping: Generate a logic diagram of the circuit using ANDs, ORs, Inverters, and F/Fs.
- 8. Verification: Use a HDL to verify the design.

# Mealy and Moore

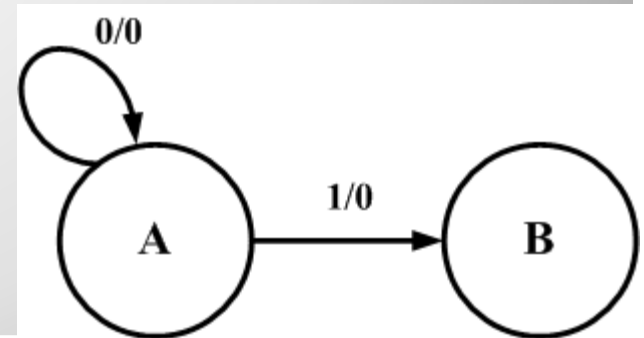
- Sequential machines are typically classified as either a Mealy machine or a Moore machine implementation.
- Moore machine: The outputs of the circuit depend only upon the current state of the circuit.
- Mealy machine: The outputs of the circuit depend upon both the current state of the circuit and the inputs.

## An example to go through the steps

- The specification: The circuit will have one input,  $X$ , and one output,  $Z$ . The output  $Z$  will be 0 except when the input sequence 1101 are the last 4 inputs received on  $X$ . In that case it will be a 1.

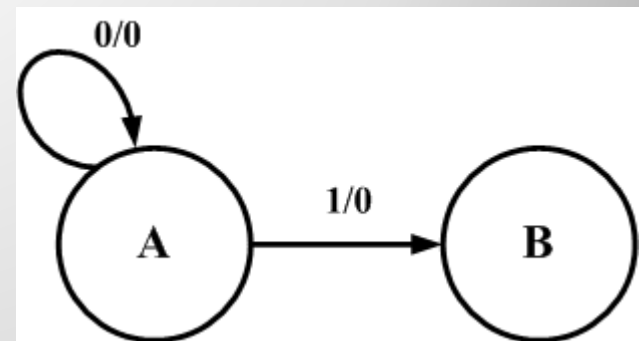
# Generation of a state diagram

- Create states and meaning for them.
  - State A – the last input was a 0 and previous inputs unknown. Can also be the reset state.
  - State B – the last input was a 1 and the previous input was a 0. The start of a new sequence possibly.
- Capture this in a state diagram



# Notes on State diagrams

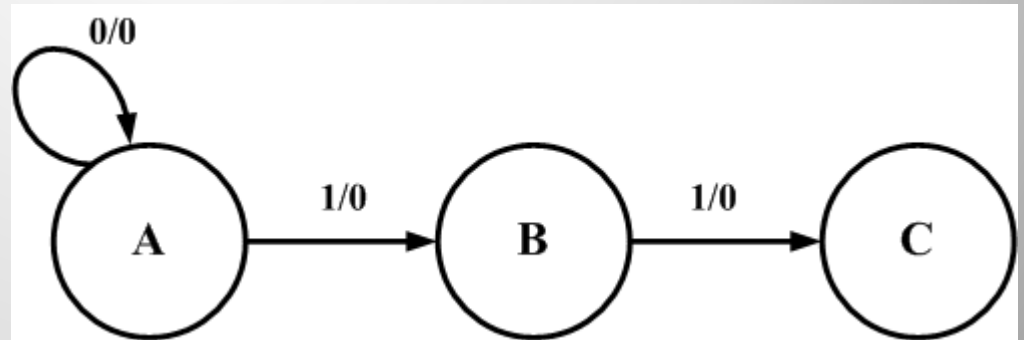
- Capture this in a state diagram
  - Circles represent the states
  - Lines and arcs represent the transition between state.
  - The notation Input/Output on the line or arc specifies the input that causes this transition and the output for this change of state.





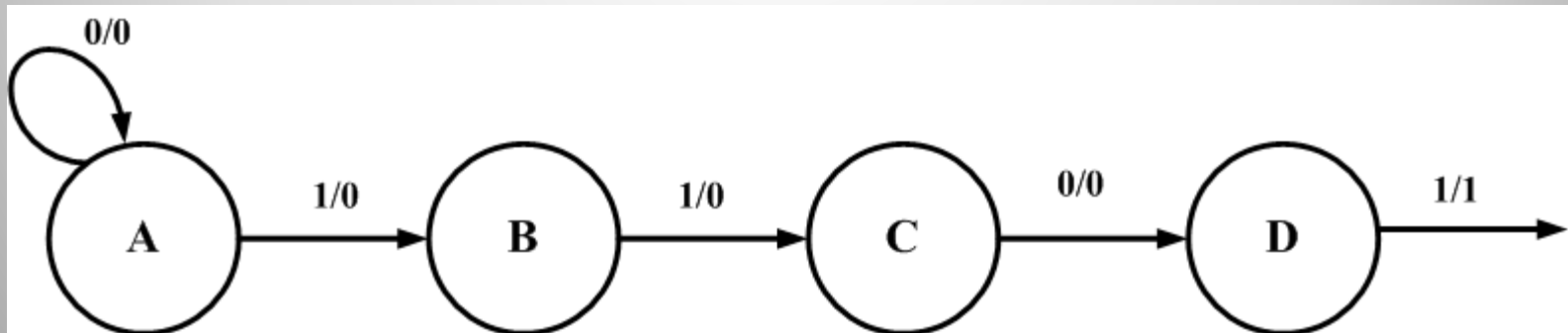
## Continue to build up the diagram

- Add a state C
  - State C – Have detected the input sequence 11 which is the start of the sequence.



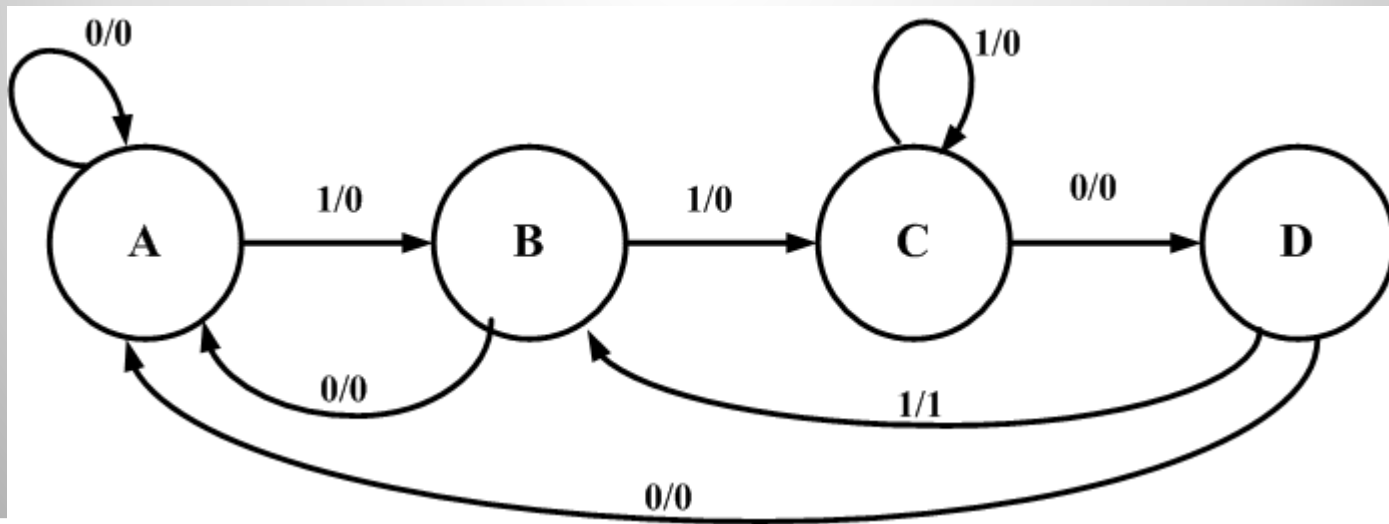
# Continue

- Add a state D
  - State D – have detected the 3<sup>rd</sup> input in the start of a sequence, a 0, now having 110. From State D, if the next input is a 1 the sequence has been detected and a 1 is output.



# Add remaining transitions

- The previous diagram was incomplete.
- In each state the next input could be a 0 or a 1. This must be included.



## Now generate a state table

- The state table
- This can be done directly from the state diagram.
- Now need to do a state assignment

Prresent State	Next State		Output	
	X =0	X=1	X=0	X=1
A	A	B	0	0
B	A	C	0	0
C	D	C	0	0
D	A	B	0	1

# Select a state assignment

- Will select a gray encoding
- For this state A will be encoded 00, state B 01, state C 11 and state D 10

Prresent State	Next State		Output	
	X =0	X=1	X=0	X=1
00	00	01	0	0
01	00	11	0	0
11	10	11	0	0
10	00	01	0	1

# Flip-flop input equations

- Generate the equations for the flip-flop inputs
- Generate the  $D_0$  equation

$Q_0Q_1$		00	01	11	10
X	0			1	
	1		1	1	

$D_0 = Q_0 Q_1 + X Q_1$

- Generate the  $D_1$  equation

$Q_0Q_1$		00	01	11	10
X	0				
	1	1	1	1	1

$D_1 = X$

# The output equation

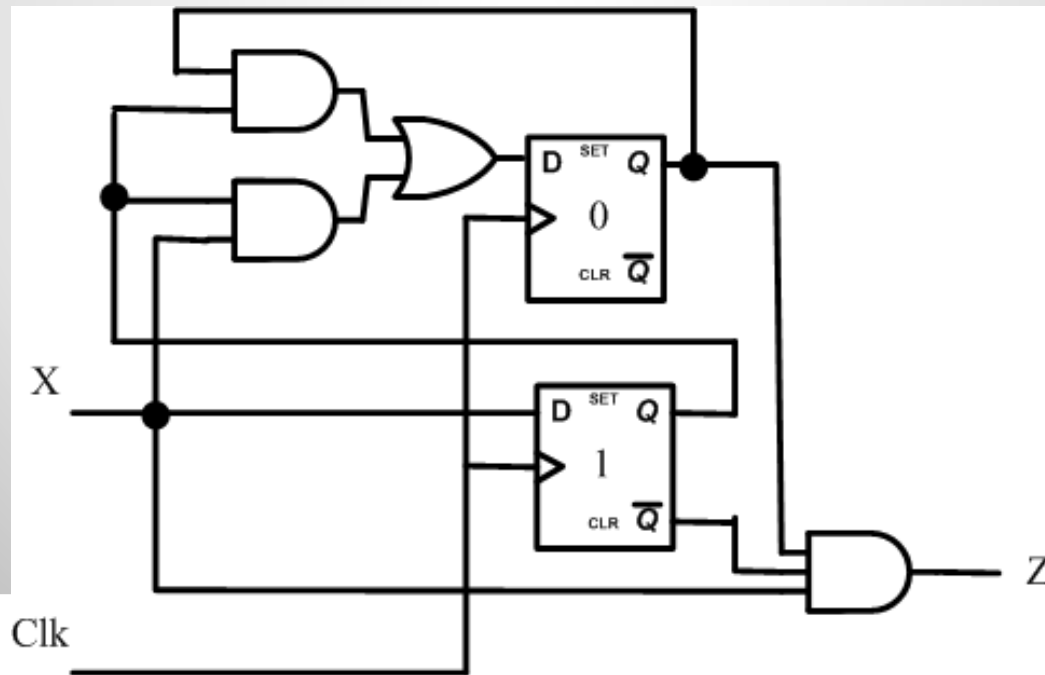
- The next step is to generate the equation for the output Z and what is needed to generate it.
- Create a K-map from the truth table.

		$Q_0Q_1$			
$X$	0				
	1				1

$$Z = X Q_0 \bar{Q}_1$$

# Now map to a circuit

- The circuit has 2 D type F/Fs





# UNIT-5

## MEMORY

# INTRODUCTION

- Memories are made up of registers
- Each register in the memory is one storage location also called memory location.
- Each storage element is called a cell .
- These cells are made up of flip-flops or capacitors in semiconductor memories.
- Data stored in a memory by a process called writing

# Classification of memory

- Memory is mainly classified into  
    Volatile memory  
    non-volatile memory
- In volatile memory data will be erased once the power is switched off
- In non-volatile memory data is retained even after the power is switched off
- RAM is volatile
- ROM is non-volatile

# RAM (Random access memory)

RAM is of two types

- SRAM (Static RAM) (flip-flop gates)

SRAM is made of flip-flops and the data is retained until the power is on

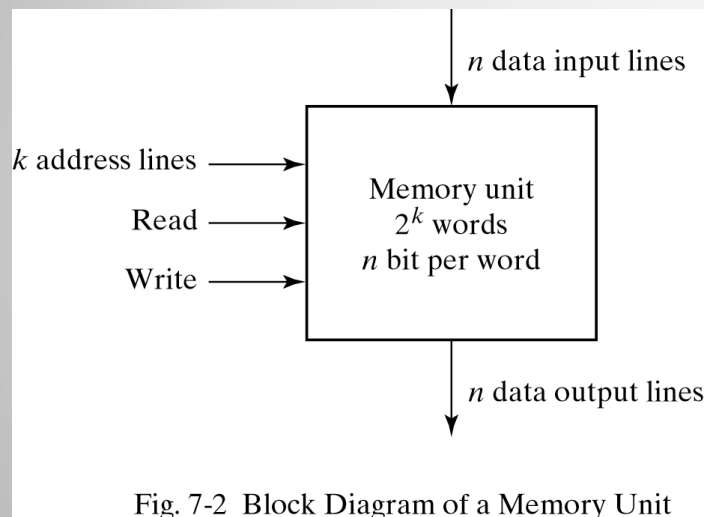
- DRAM (Dynamic RAM)

DRAM is made of capacitors and the data needs to be refreshed to retain the contents of the memory. For that purpose we need additional circuit called refresh circuitry to refresh the data after a particular time interval.

# ROM (Read only memory)

- ROM is classified as
  - ROM
  - PROM (programmable)
  - EPROM (erasable programmable)
  - EEPROM (electronically erasable programmable)

# Block diagram of memory unit



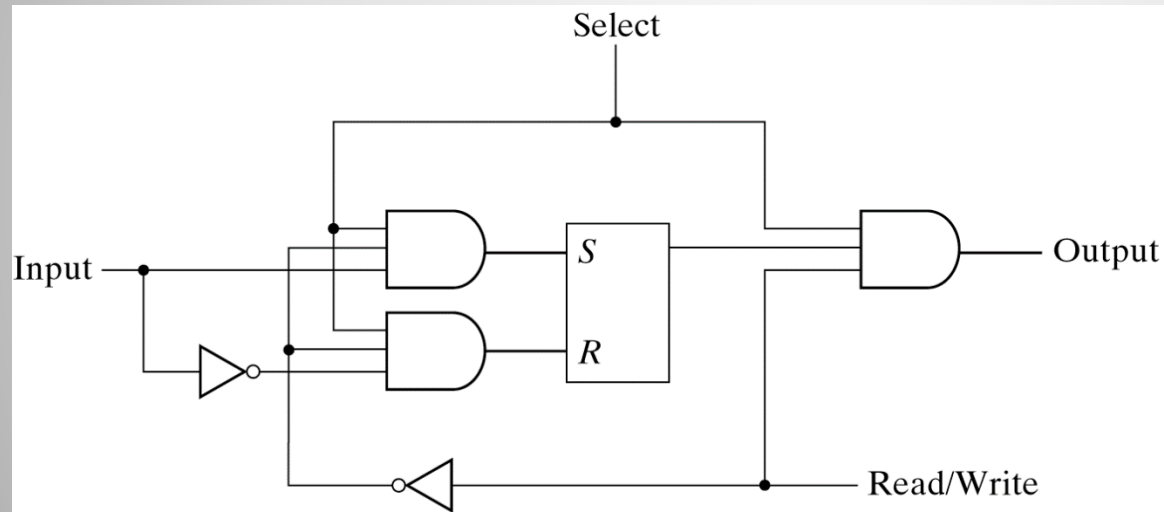
**$k$  address lines : select one particular word**

**read, write : specify the direction of transfer**

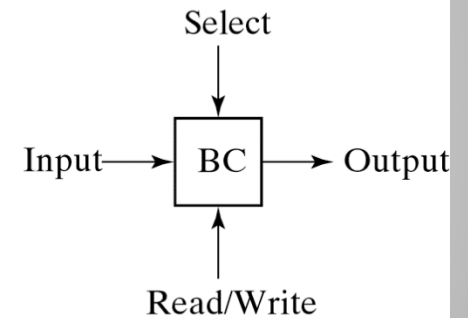
**$n$  data input line : provide the information to be stored in memory**

**$n$  data output line : supplying the information coming out of memory**

# One bit memory cell

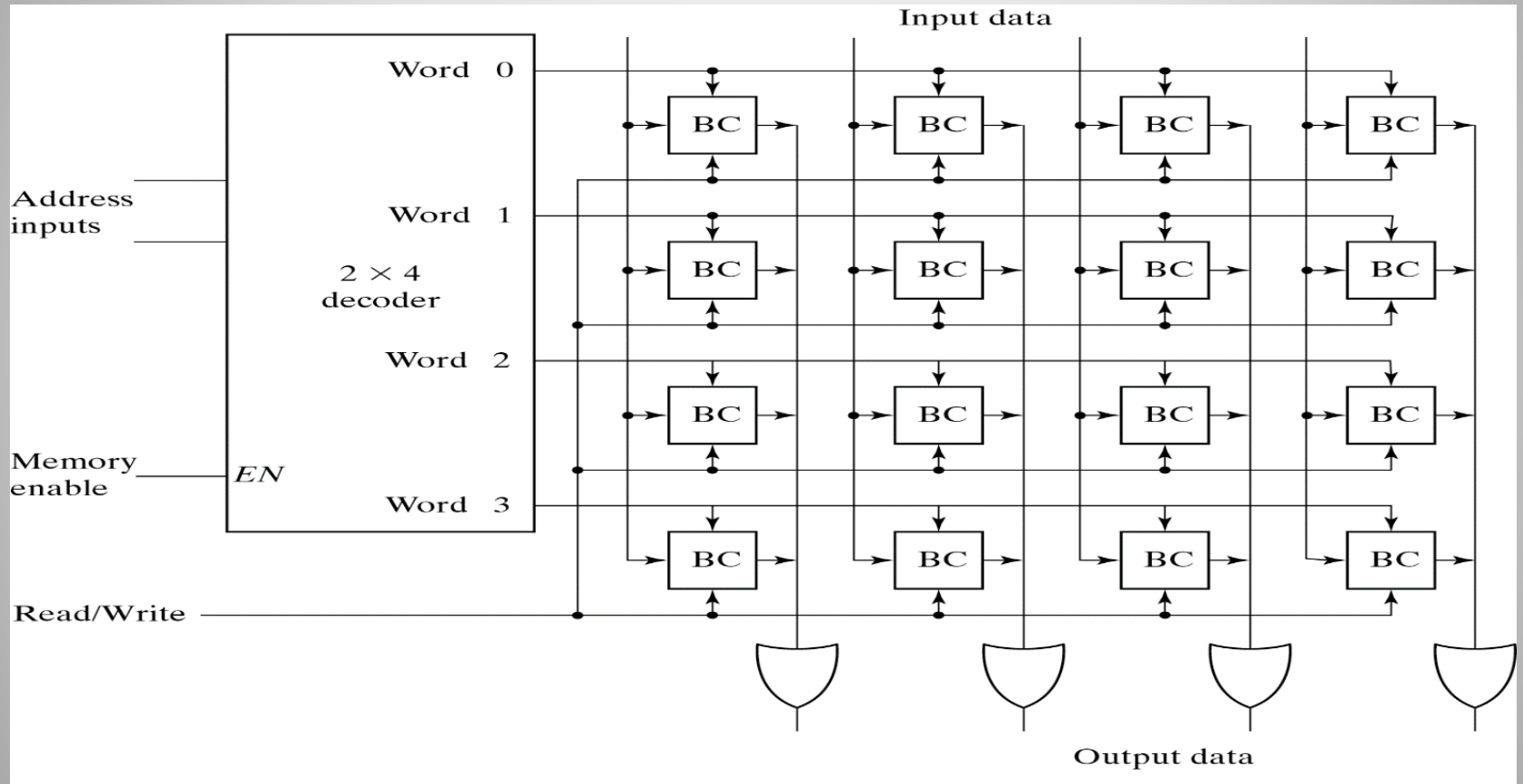


(a) Logic diagram



(b) Block diagram

# 4x4 RAM



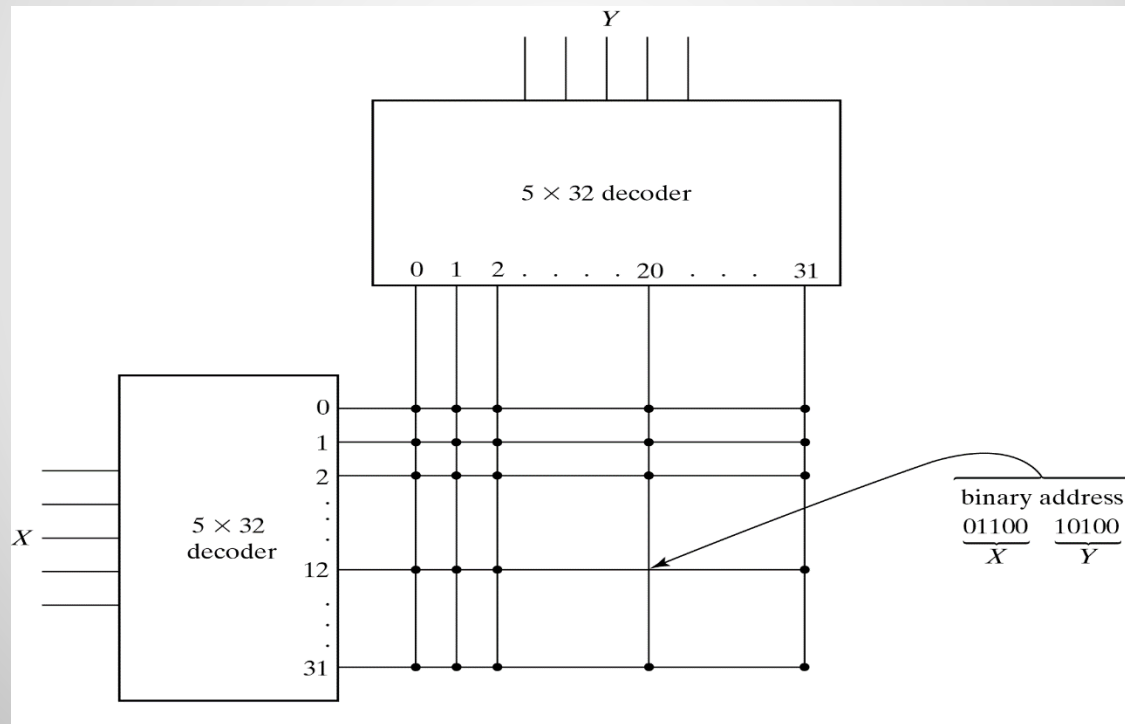


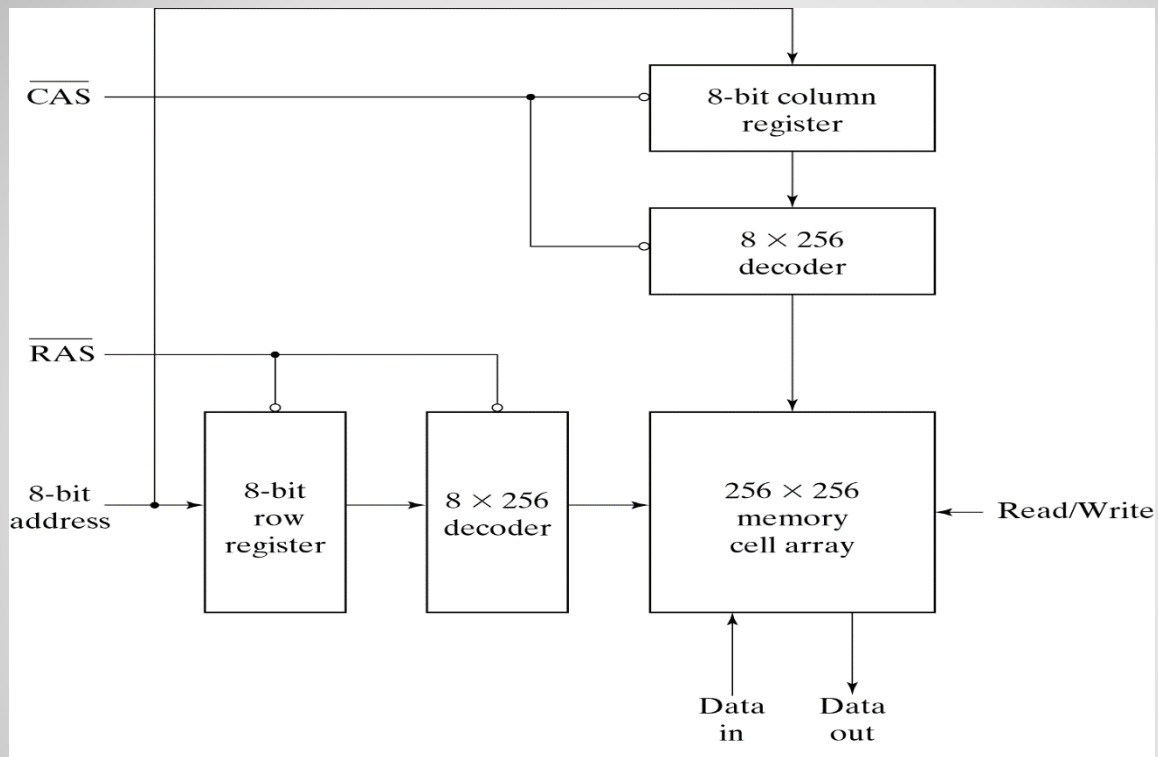
# Coincident Decoding

( two-dimensional selection scheme)

- Decoder with  $k$  input and  $2^k$  output requires  $2^k$  AND gates with  $k$  input
- $k$  input decoder can be implemented by two  $k/2$  input decoders with one for column and another for row
- e.g.,  $10 \times 1024$  decoder can be implemented by two  $5 \times 32$  decoders

# Two dimensional decoding structure for a 1k-word memory





## Address multiplexing 64K-word memory

★ Range of memory size

- $2^{10} \sim 2^{32}$  words

★ bytes

- $K=2^{10}$  、  $M=2^{20}$  、  $G=2^{30}$  。
- $64K=2^{16}$  、  $2M=2^{21}$  、  $4G=2^{32}$  。

★ Memory 1K x 16

- 10 bits address , 16 bits in each word

★ Determine the no. of bits for address

k: no. of address bits

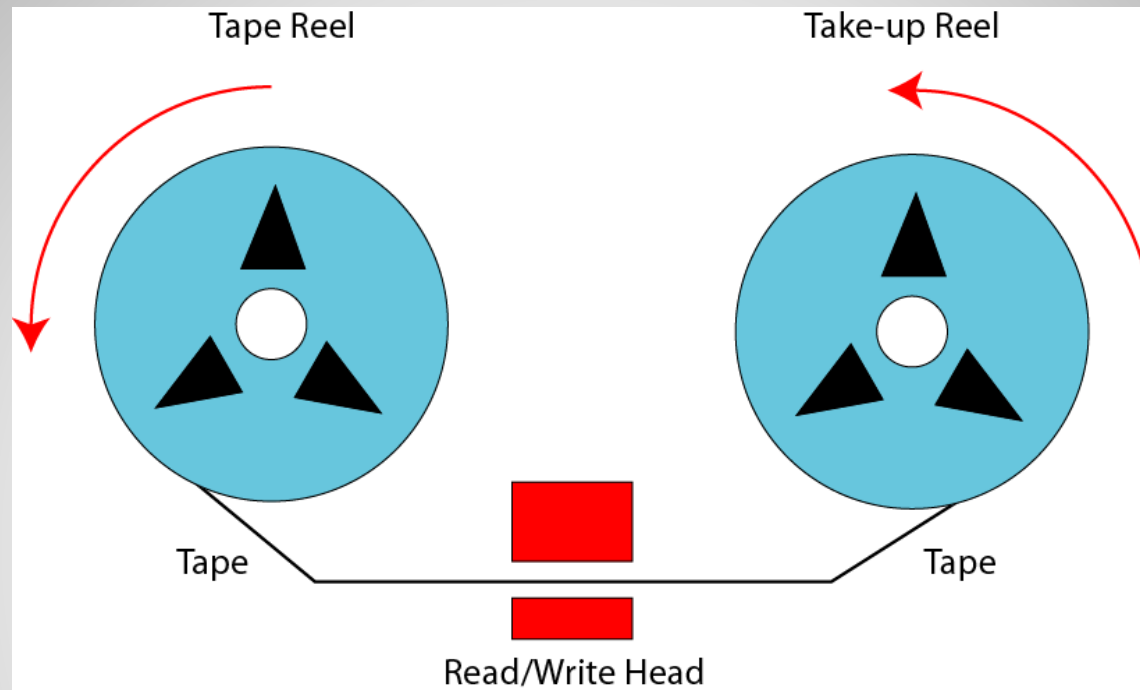
m: total number of words

# Control inputs to memory chip

Memory enable	Read/write	Memory operation
0	X	None
1	0	Write to select word
1	1	Read from selected word

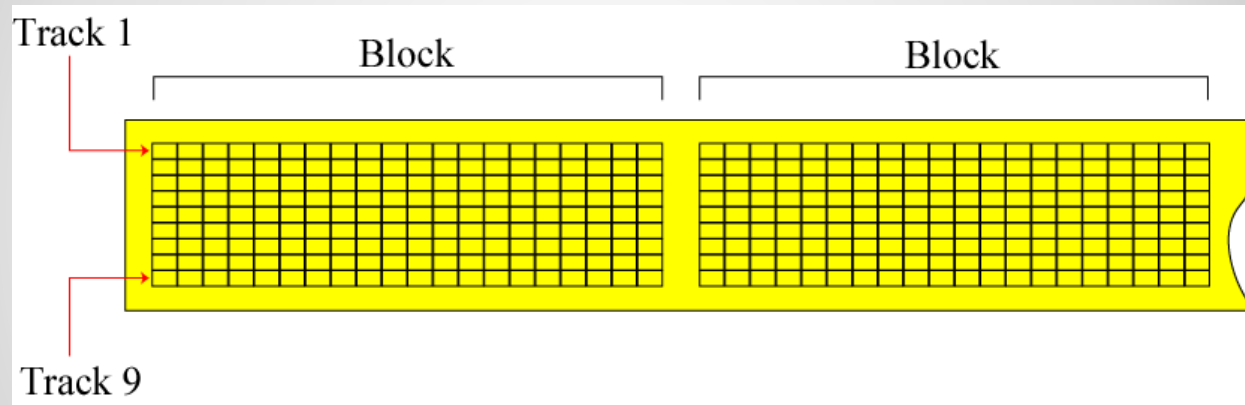
# Memory cycle timing waveforms

- access time
  - the time required to select a word and read it
- cycle time
  - the time required to complete a write cycle
- access time 、 cycle time
  - equal to a fixed number of CPU clock



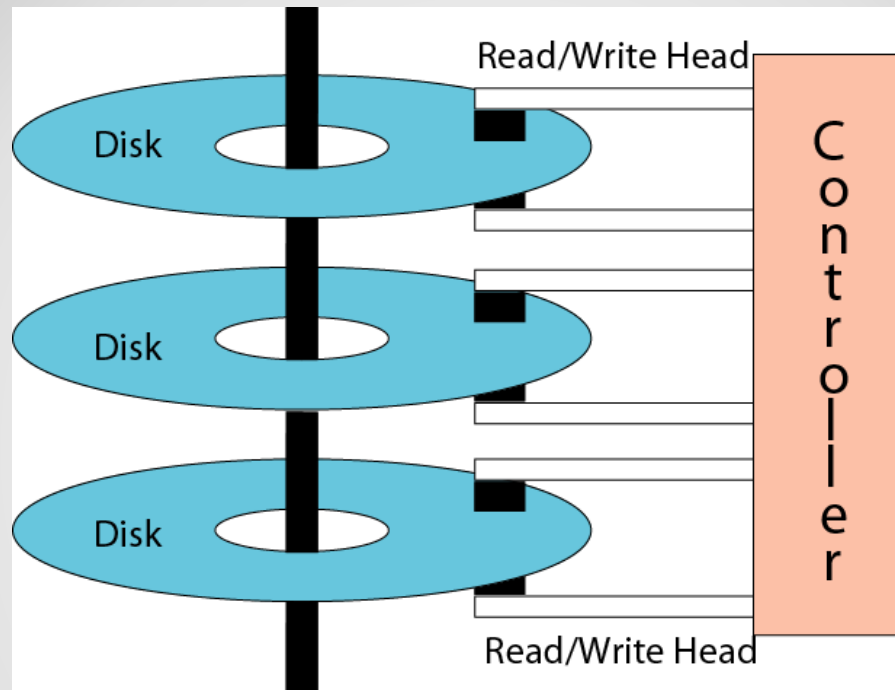
# SEQUENTIAL MEMORY-MAGNETIC TAPE

**Mechanical configuration of a tape:**

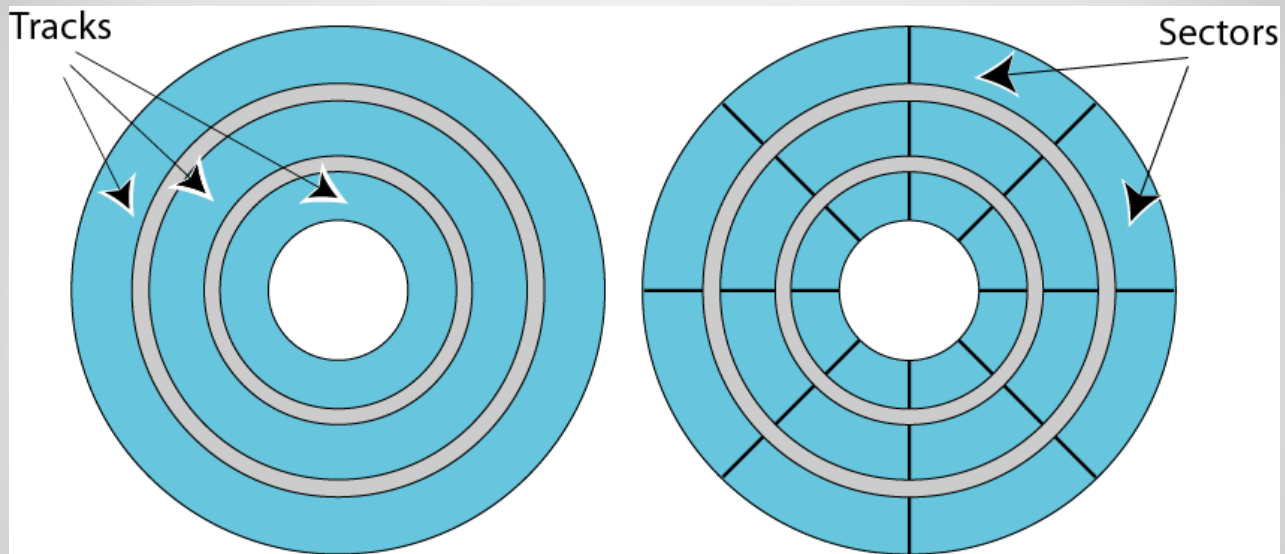


**Surface organization of a tape**

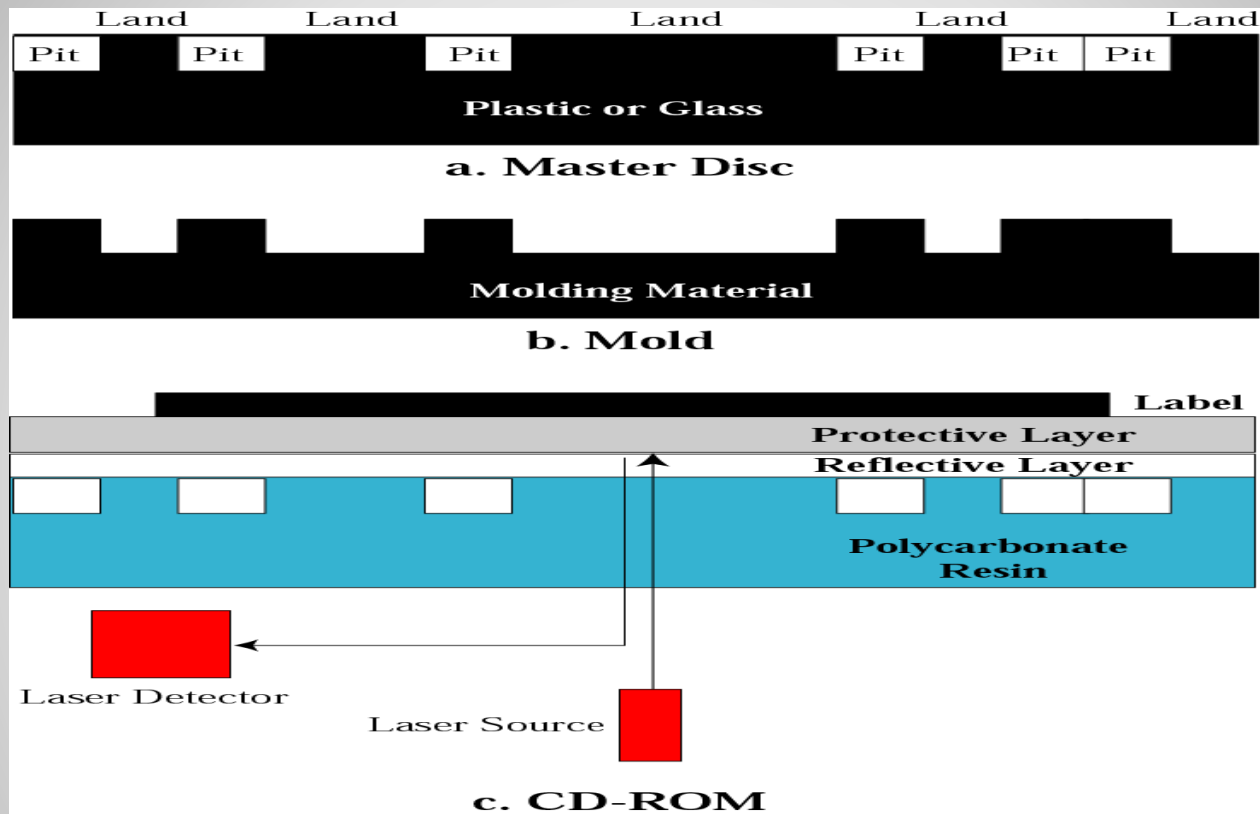




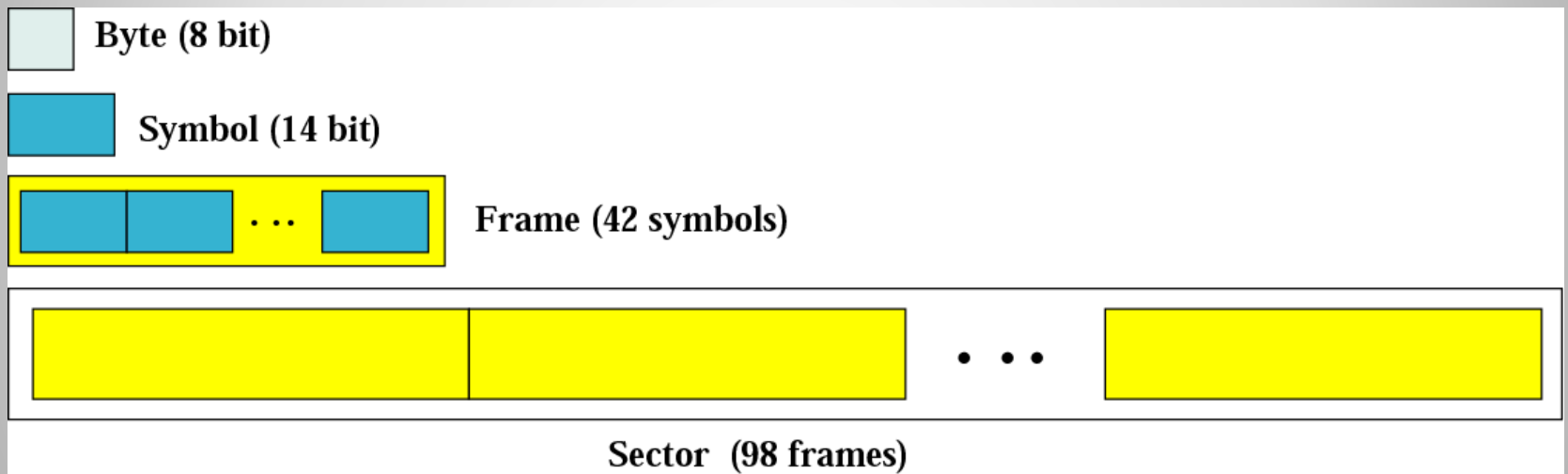
**Physical layout of a magnetic disk**



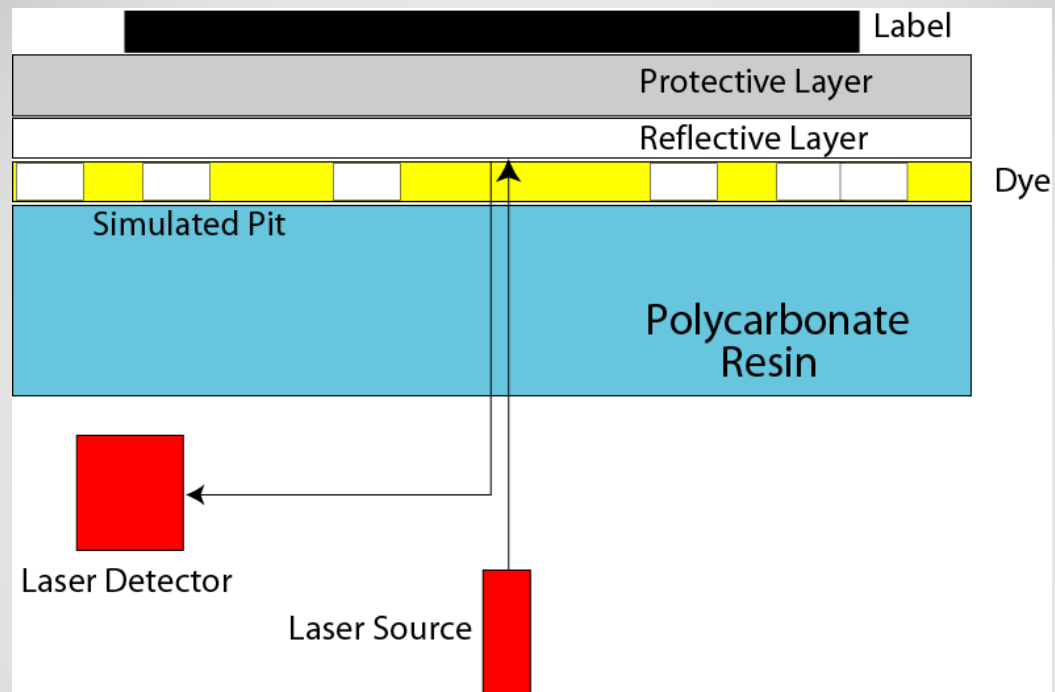
**Surface organization of a disk**



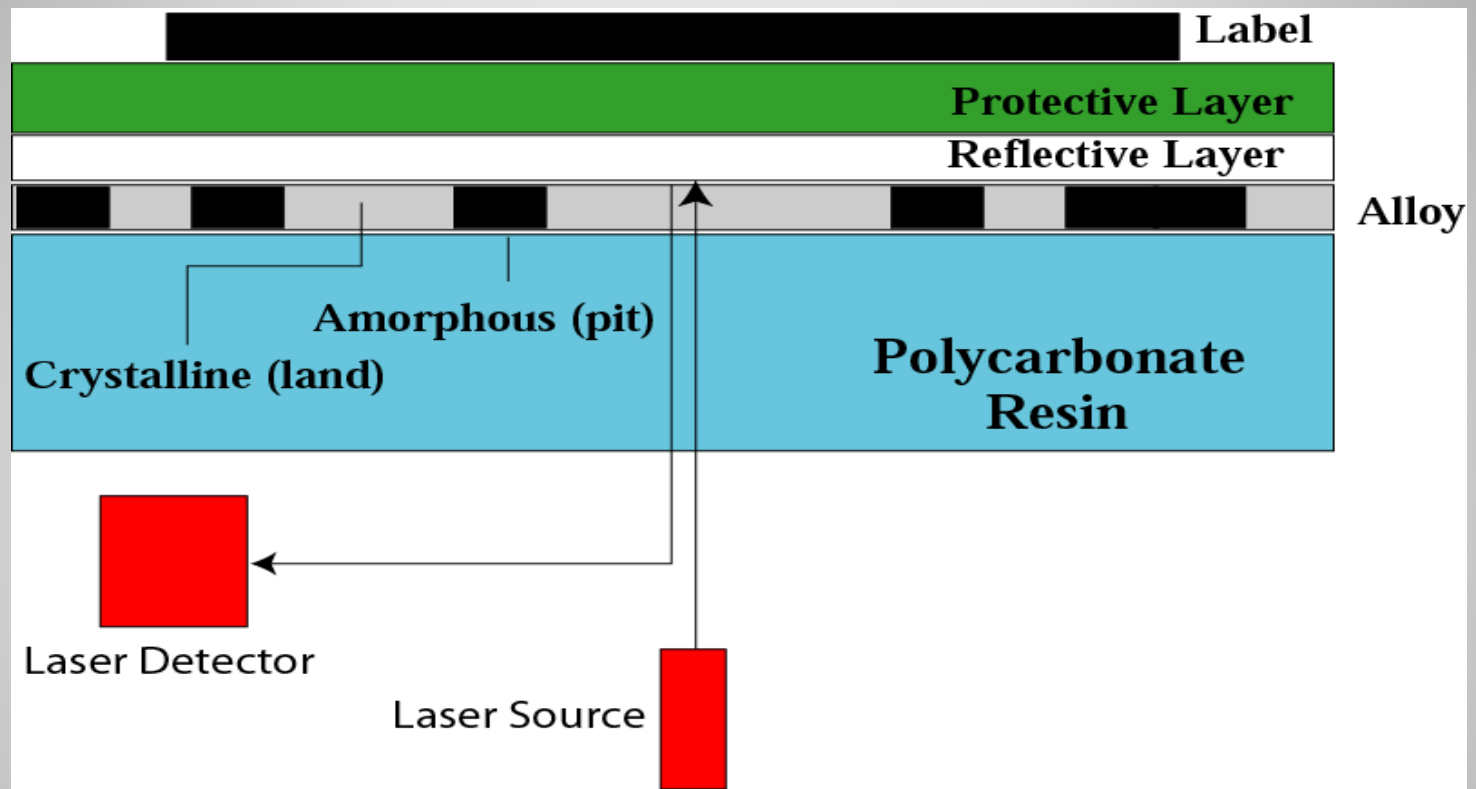
# Creation and use of CD-ROM



# CD-ROM format



## Making a CD-R



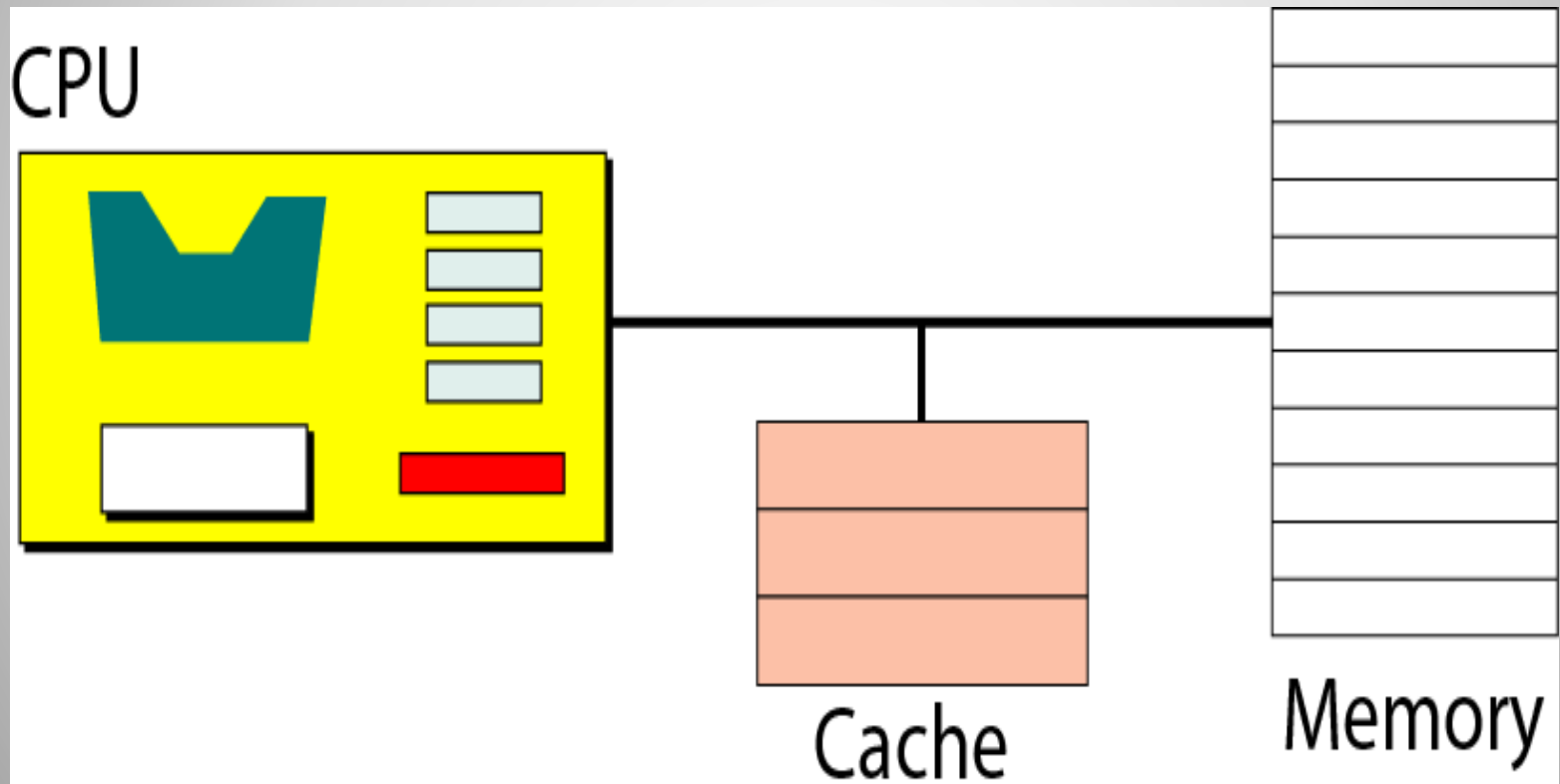
## Making a CD-RW

# DVD capacity:

<i>Feature</i>	<i>Capacity</i>
single-sided, single-layer	4.7 GB
single-sided, dual-layer	8.5 GB
double-sided, single-layer	9.4 GB
double-sided, dual-layer	17 GB

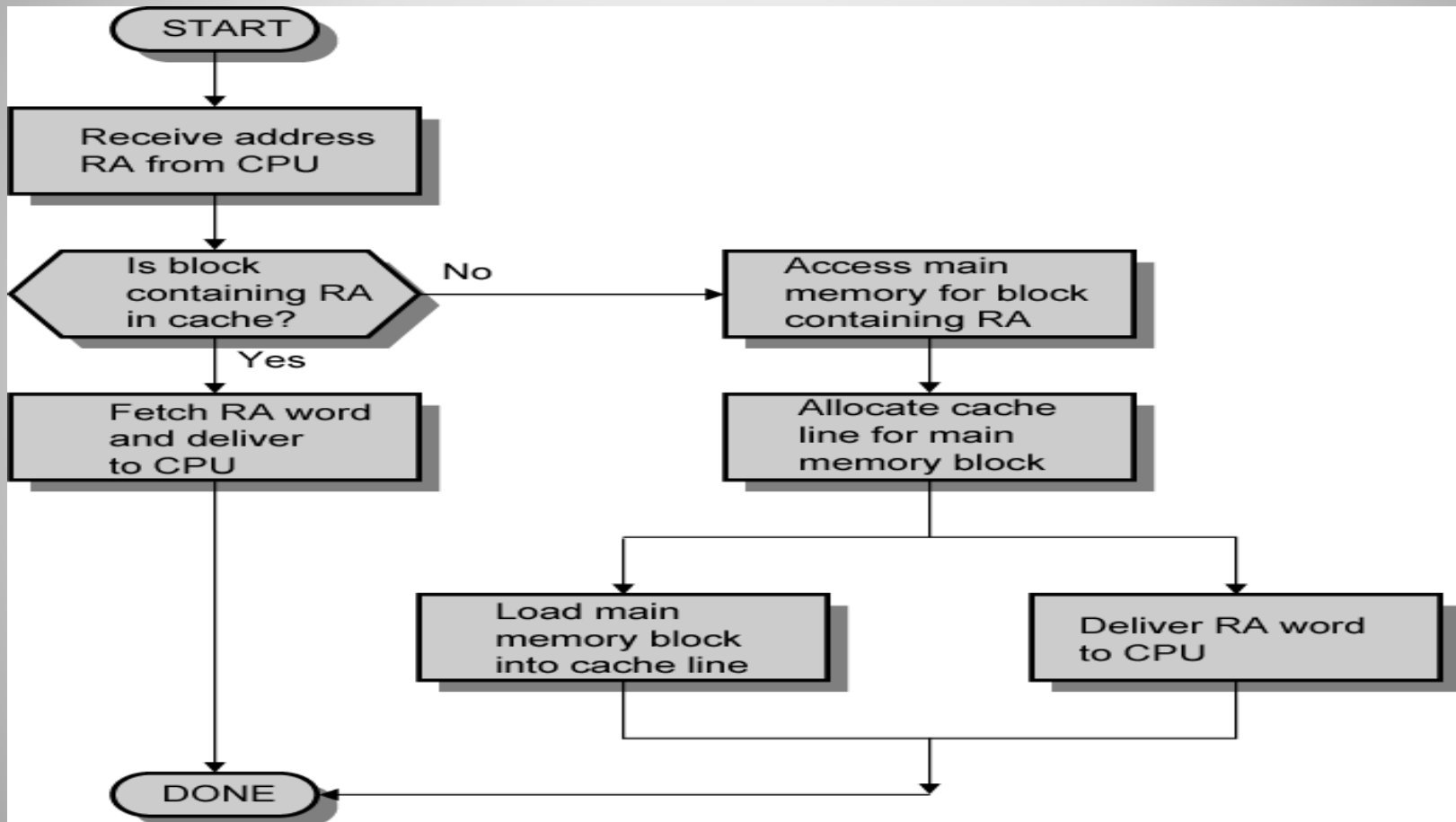
***DVD capacities***

# Cache memory





# Cache Read Operation - Flowchart

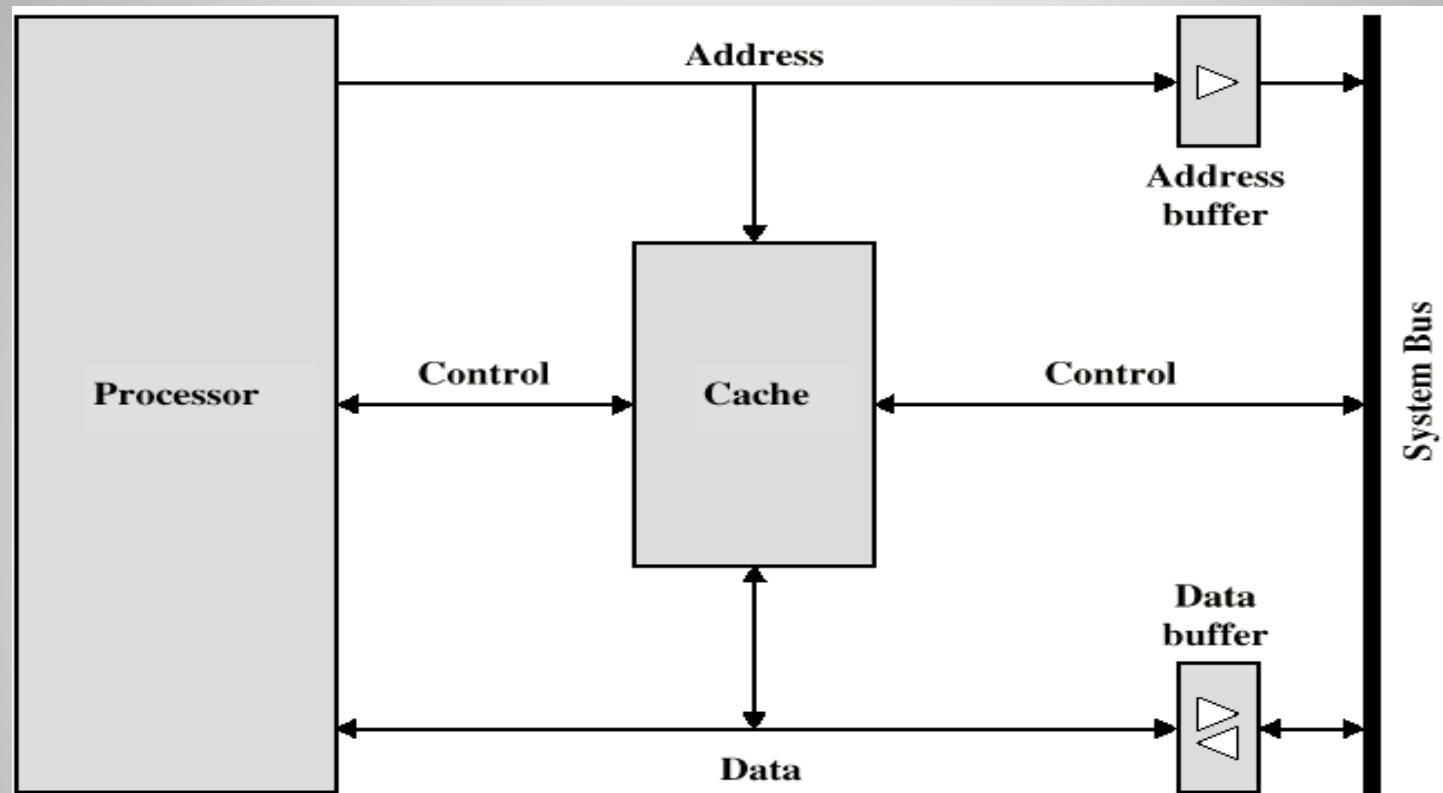


# Cache Design

- Size
- Mapping Function
- Replacement Algorithm
- Write Policy
- Block Size
- Number of Caches

# Size does matter

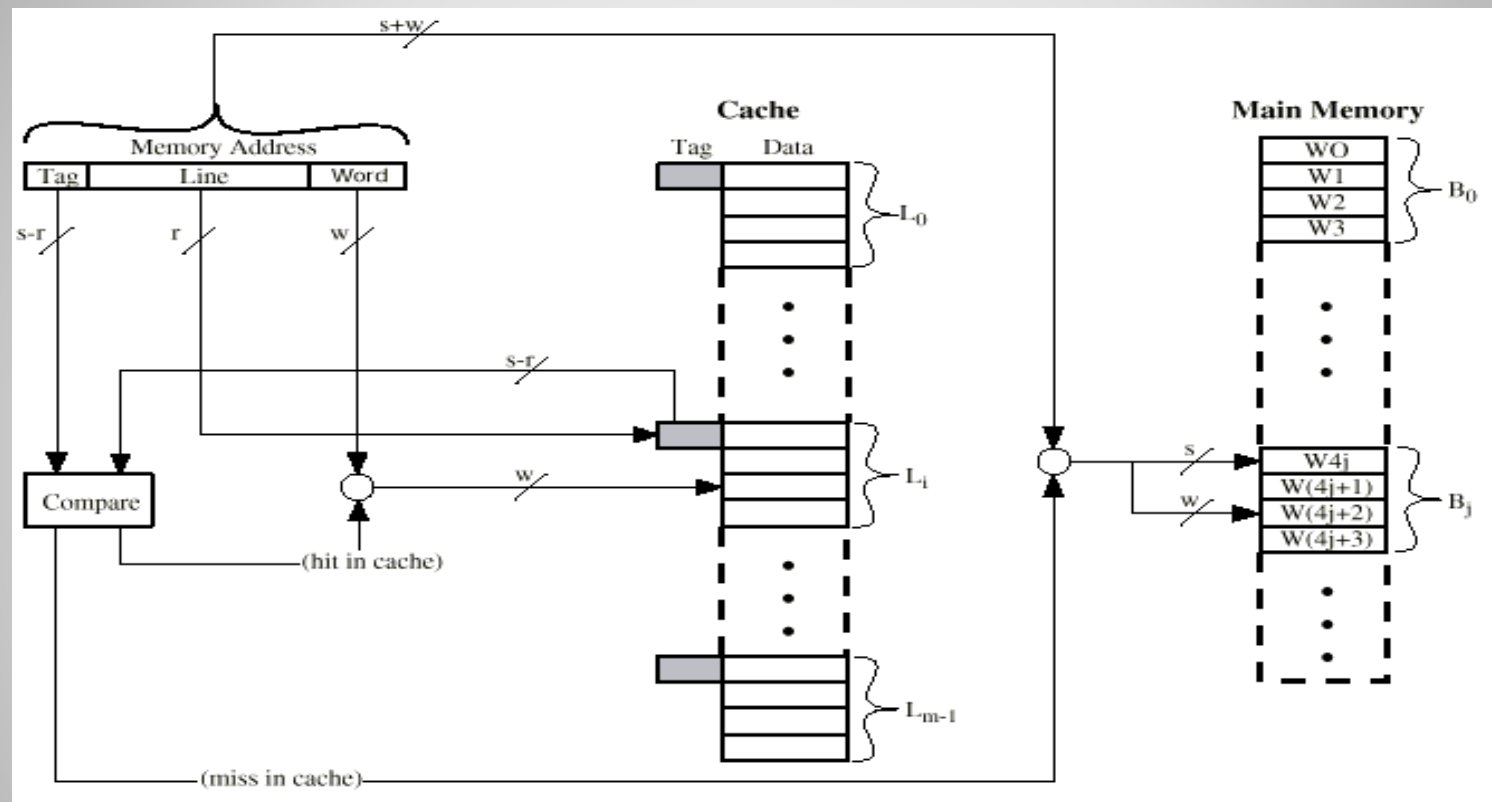
- Cost
  - More cache is expensive
- Speed
  - More cache is faster (up to a point)
  - Checking cache for data takes time



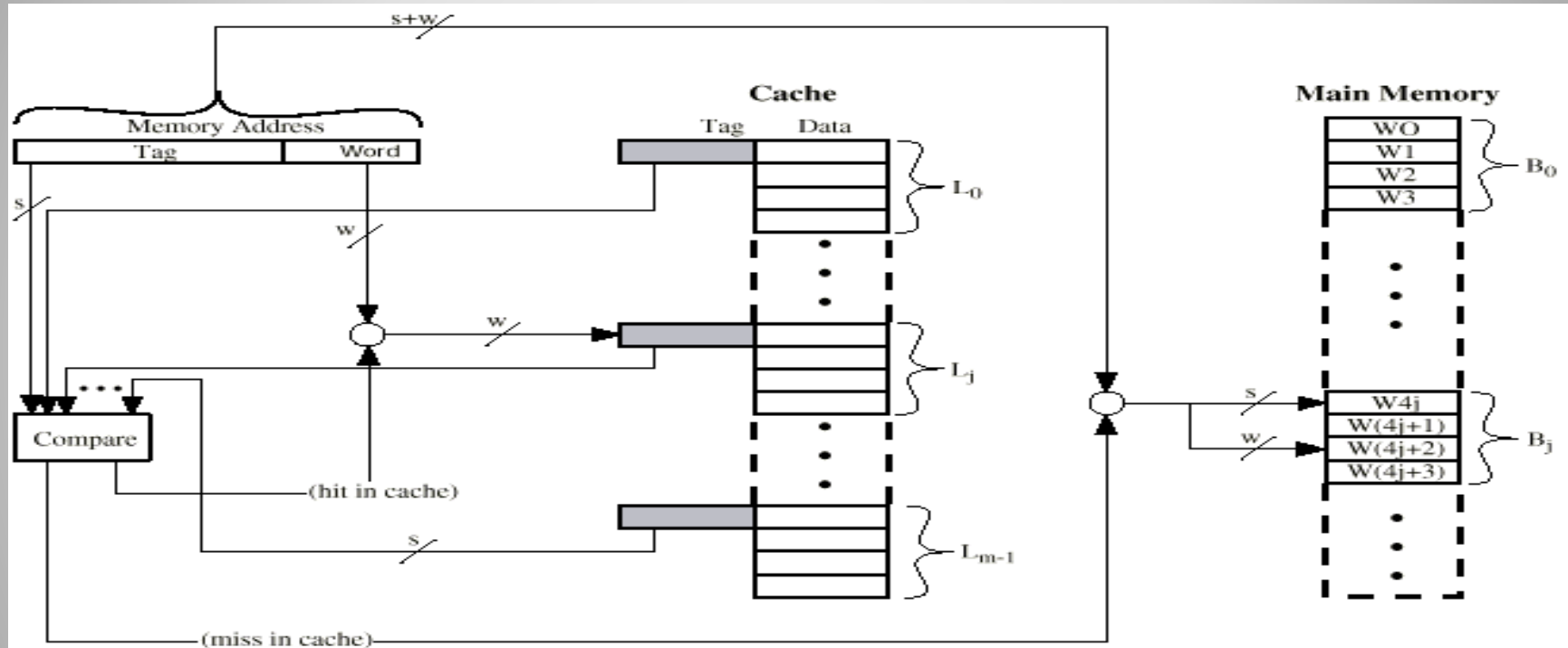
# Typical Cache Organization

# Mapping Function

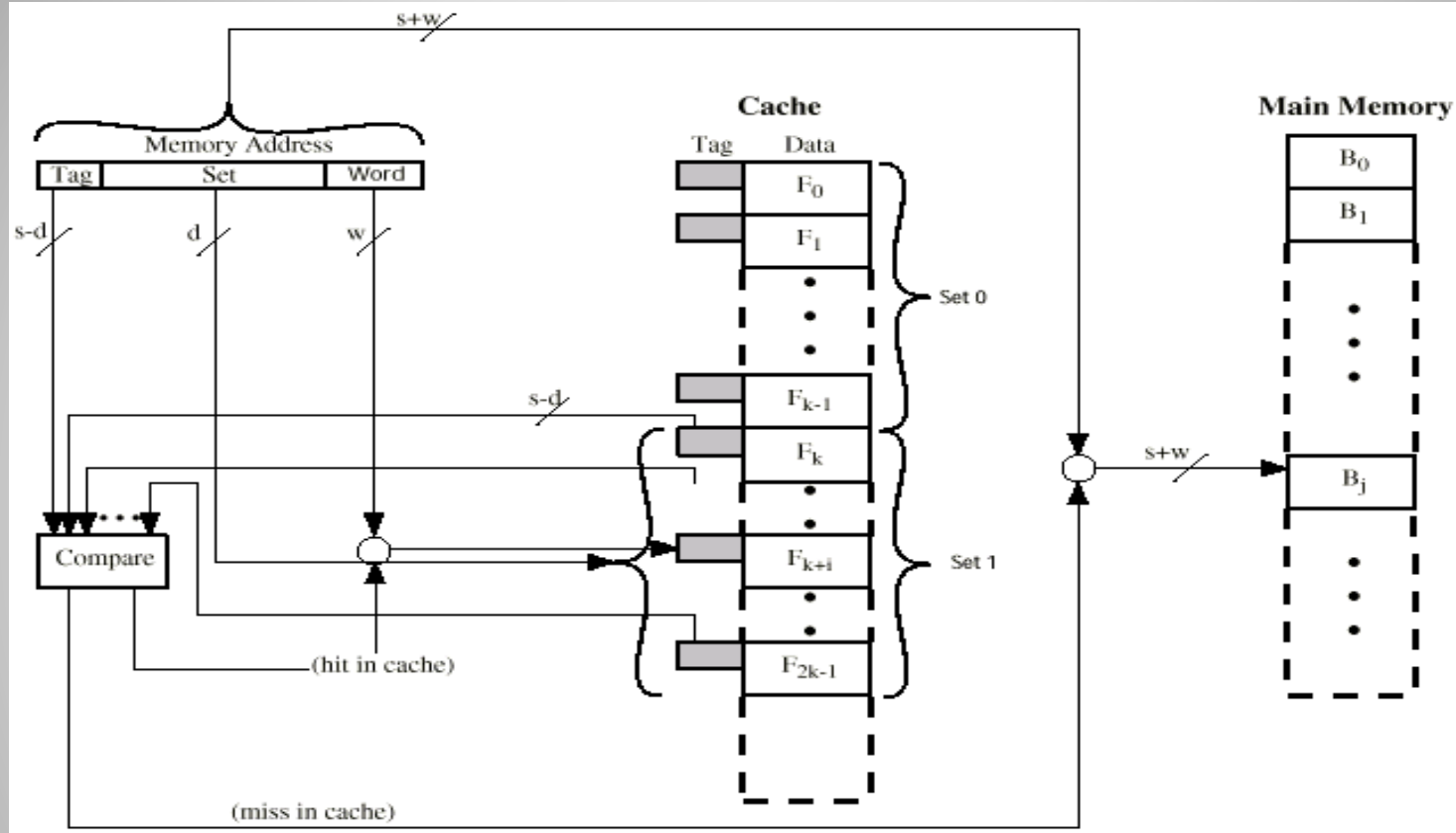
- Cache of 64kByte
- Cache block of 4 bytes
  - i.e. cache is 16k ( $2^{14}$ ) lines of 4 bytes
- 16MBytes main memory
- 24 bit address
  - ( $2^{24}=16\text{M}$ )



# Direct Mapping Cache Organization



# Fully Associative Cache Organization



## Two Way Set Associative Cache Organization



# Types of PLD (Programmable Logic Device)

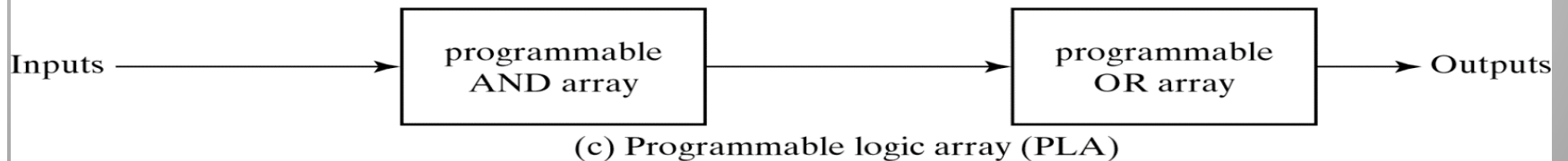
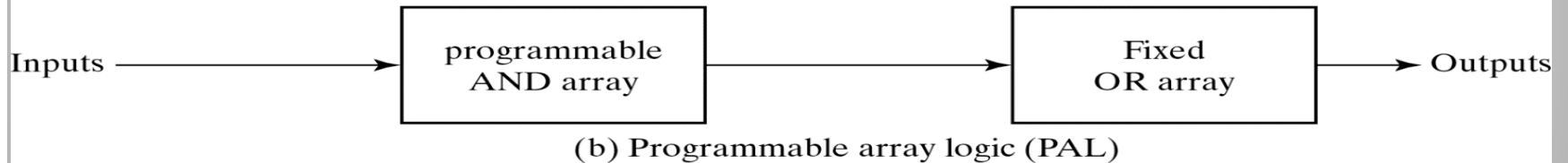
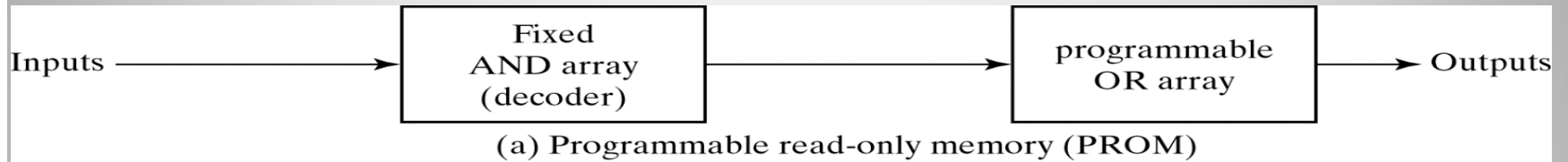
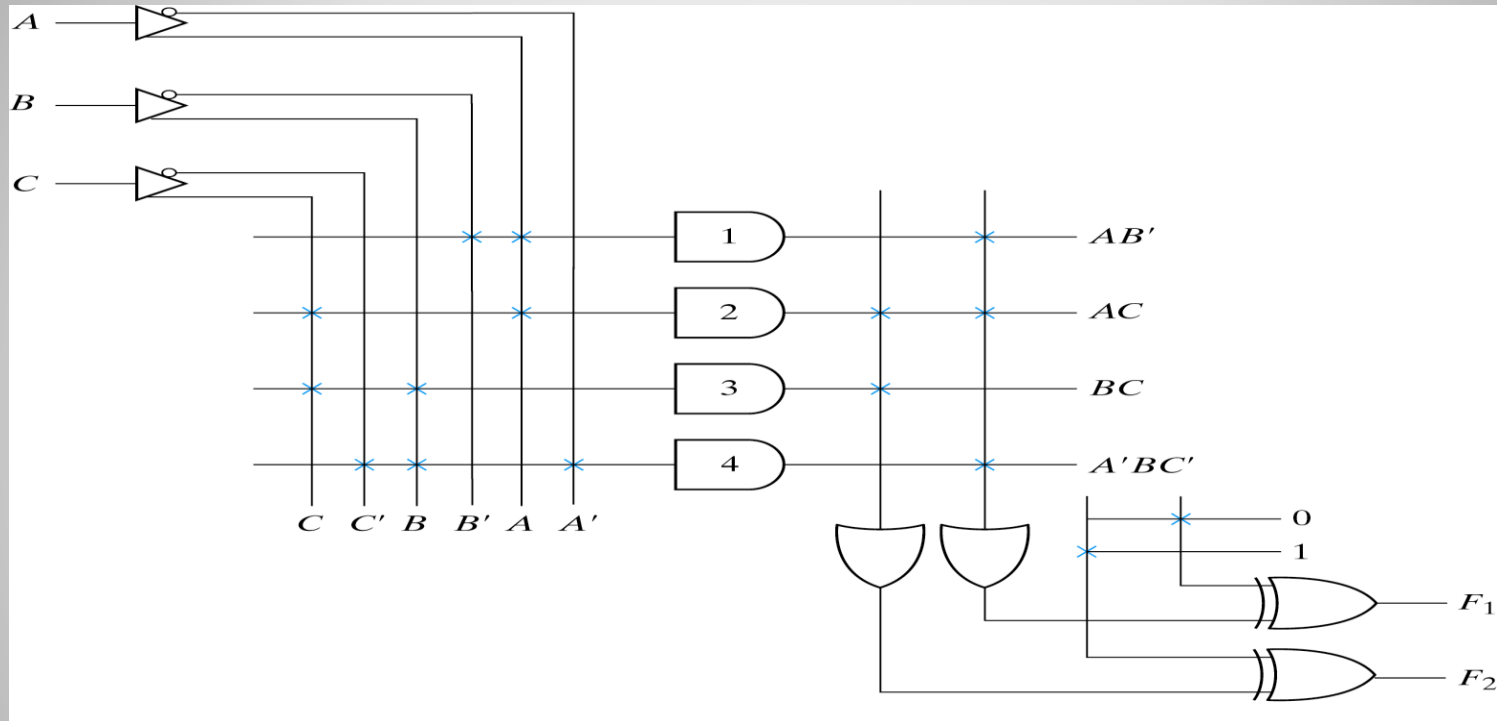


Fig. 7-13 Basic Configuration of Three PLDs

# Programmable Logic Array (PLA)

- similar to PROM
- does not provide full decoding and does not generate all the minterms
- decoder is replaced by an array of AND gate



**PLA with 3 inputs, 4 product terms,  
and two outputs**

# PLA Programming Table

- ★ PLA Programming Table consists of three sections
  - 1st, list the product terms numerically
  - 2nd, specify the required path between inputs and AND gates
  - 3rd, specifies the paths between the AND and OR gates

	Product term	inputs			outputs	
		A	B	C	(T)	(C)
					F <sub>1</sub>	F <sub>2</sub>
$AB'$	1	1	0	-	1	-
$AC$	2	1	-	1	1	1
$BC$	3	-	1	1	-	1
$A'BC'$	4	0	1	0	1	-

★ Implement the following two Boolean functions with a PLA:

$$F_1 = (A, B, C) = \sum (0, 1, 2, 4)$$

$$F_2 = (A, B, C) = \sum (0, 5, 6, 7)$$

★ Simplified by K-map :

$$F_1 = (AB + AC + BC)'$$

$$F_2 = AB + AC + A'B'C'$$

**Example**

		<i>BC</i>		<i>B</i>	
		00	01	11	10
<i>A</i>	0	1	1	0	1
<i>A</i>	1	1	0	0	0

$\underbrace{\hspace{10em}}_C$

$$F_1 = A'B' + A'C' + B'C'$$

$$F_1 = (AB + AC + BC)'$$

		<i>BC</i>		<i>B</i>	
		00	01	11	10
<i>A</i>	0	1	0	0	0
<i>A</i>	1	0	1	1	1

$\underbrace{\hspace{10em}}_C$

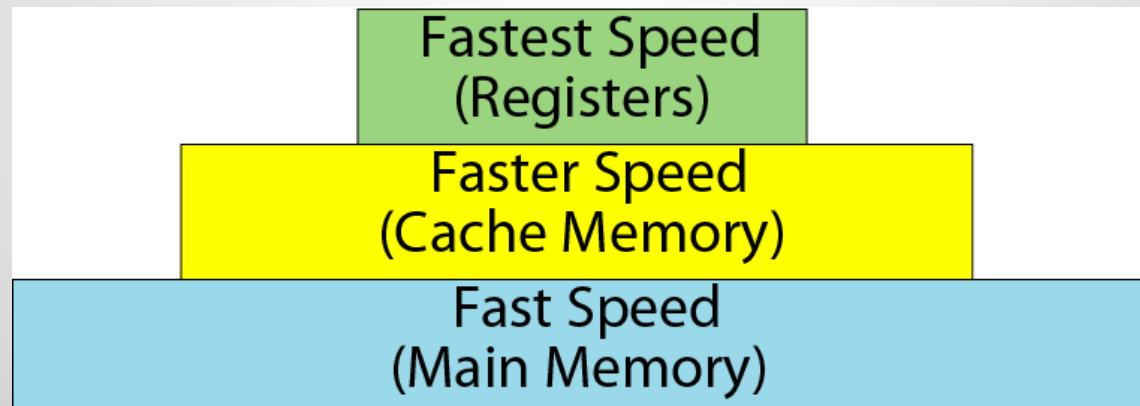
$$F_2 = AB + AC + A'B'C'$$

$$F_2 = (A'C + A'B + AB'C')'$$

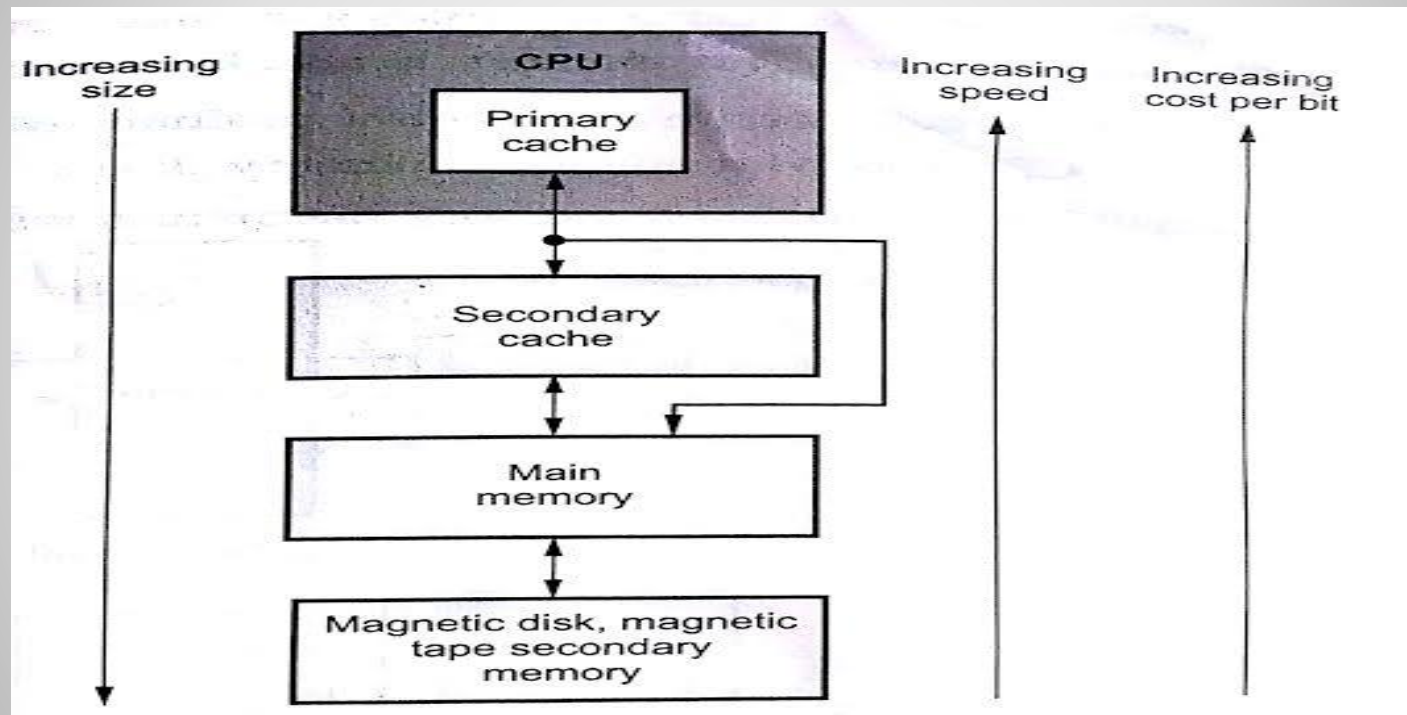
PLA programming table

	Outputs					
	Product term	Inputs			(C)	(T)
		<i>A</i>	<i>B</i>	<i>C</i>	<i>F</i> <sub>1</sub>	<i>F</i> <sub>2</sub>
<i>AB</i>	1	1	1	–	1	1
<i>AC</i>	2	1	–	1	1	1
<i>BC</i>	3	–	1	1	1	–
<i>A'B'C'</i>	4	0	0	0	–	1

**solution**



## **Memory hierarchy**



# Memory hierarchy



# Memory hierarchy in a computer system

