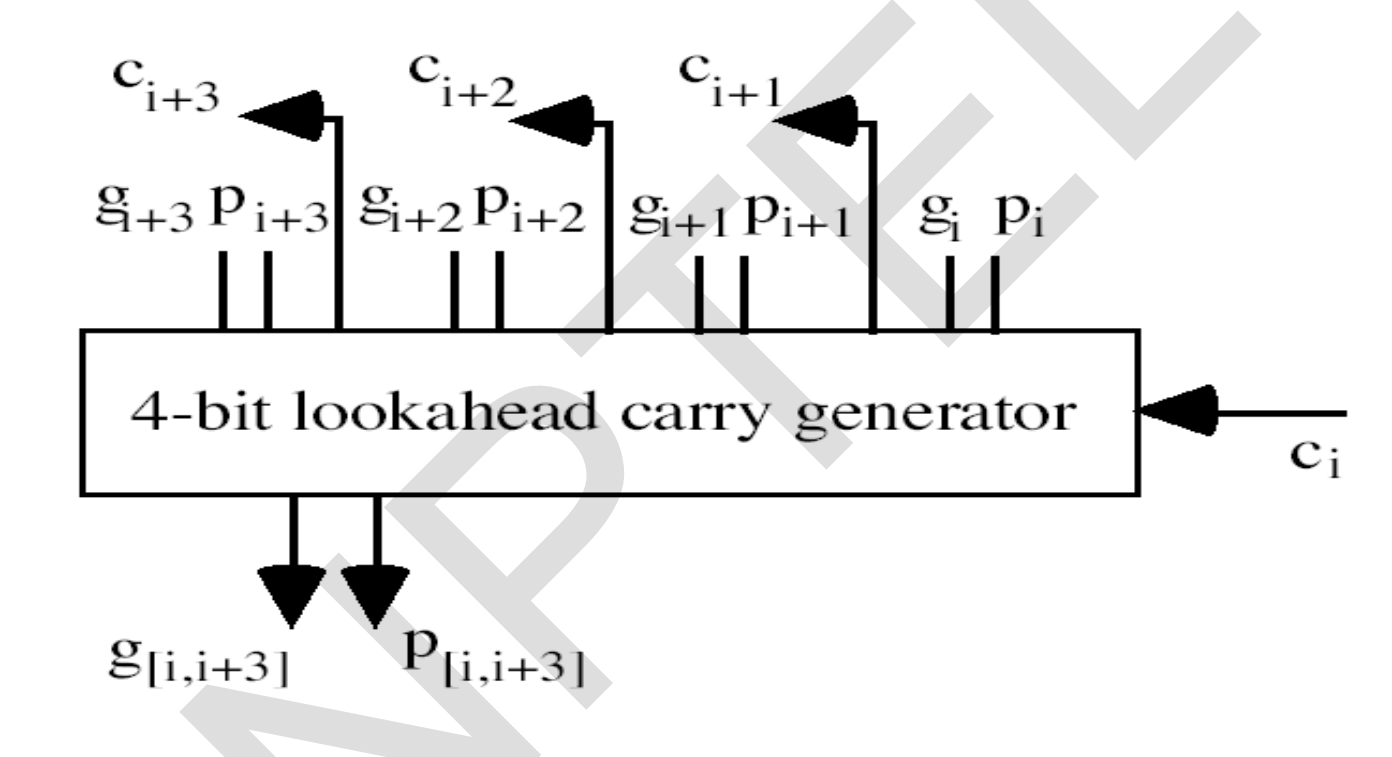
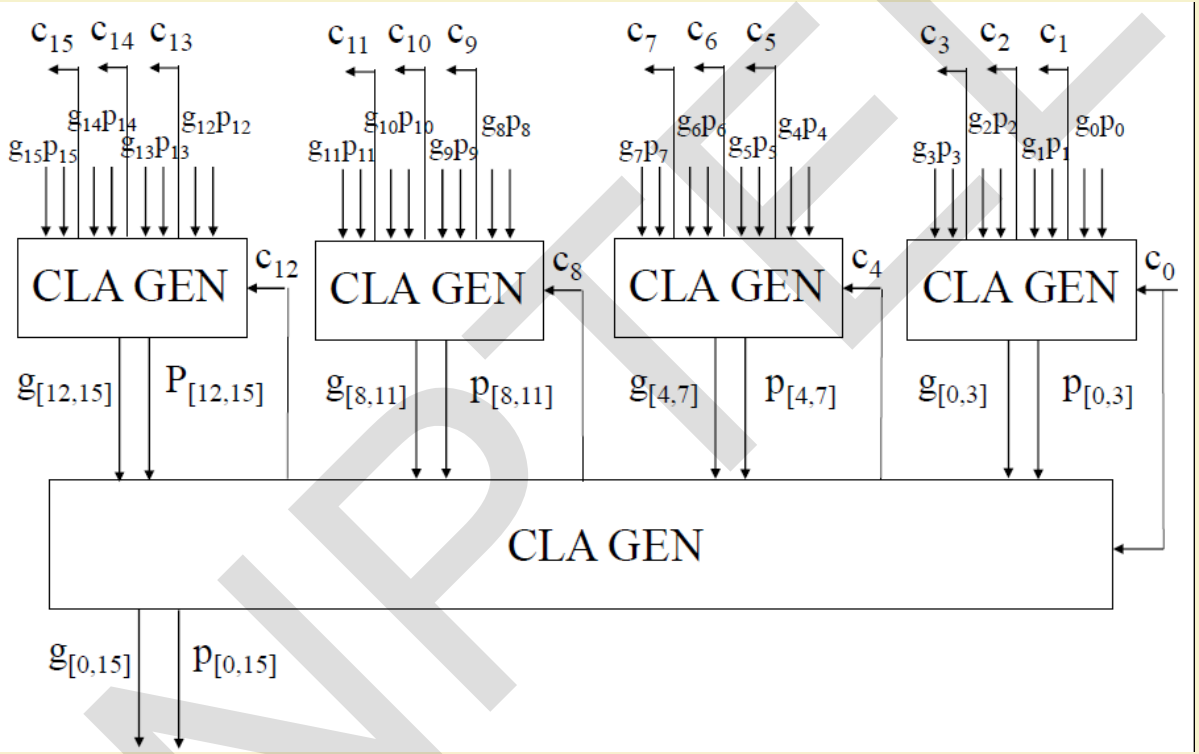
4-bit Lookahead Carry Generator Symbol



16-bit 2-level Carry Lookahead Adder



Operation of 16-bit 2-level Carry Lookahead Adder

Signals computed	Formulas	Delay
g_i, p_i i=0..15	$g_i = x_i y_i$ $p_i = x_i \oplus y_i$	1 gate delay
$g_{[i..i+3]}, p_{[i..i+3]}$ i=0, 4, 8, 12	$g_{[i..i+3]} = g_{i+3} + g_{i+2}\, p_{i+3} + g_{i+1}\, p_{i+2}\, p_{i+3} + g_i\, p_{i+1}\, p_{i+2}\, p_{i+3}$ $p_{[i..i+3]} = p_i\, p_{i+1}\, p_{i+2}\, p_{i+3}$	2 gate delays

16-bit 2-level Carry Lookahead Adder

Signals computed	Formulas	Delay
c_4, c_8, c_{12}		2 gate delays
$g_{[0..15]}, p_{[0..15]}$		

$$c_4 = g_{[0..3]} + c_0\, p_{[0..3]} \qquad c_8 = g_{[4..7]} + g_{[0..3]}\, p_{[4..7]} + c_0\, p_{[0..3]}\, p_{[4..7]}$$

$$c_{12} = g_{[8..11]} + g_{[4..7]}\, p_{[8..11]} + g_{[0..3]}\, p_{[4..7]} p_{[8..11]} + c_0 p_{[0..3]} p_{[4..7]} p_{[8..11]}$$

$$g_{[0..15]} = g_{[12..15]} + g_{[8..11]}\, p_{[12..15]} + g_{[4..7]}\, p_{[8..11]} p_{[12..15]} + g_{[0..3]} p_{[4..7]} p_{[8..11]}\, p_{[12..15]}$$

$$p_{[0..15]} = p_{[0..3]} p_{[4..7]} p_{[8..11]}\, p_{[12..15]}$$

16-bit 2-level Carry Lookahead Adder

Signals computed	Formulas	Delay
$c_{i+1}, c_{i+2}, c_{i+3}$ i = 4, 8, 12		2 gate delays

i.e., $c_5, c_6, c_7, c_9, c_{10}, c_{11}, c_{13}, c_{14}, c_{15}$

$$c_{i+3} = g_{i+2} + g_{i+1} p_{i+2} + g_i p_{i+1} p_{i+2} + c_i p_i p_{i+1} p_{i+2}$$

$$c_{i+2} = g_{i+1} + g_i p_{i+1} + c_i p_i p_{i+1}$$

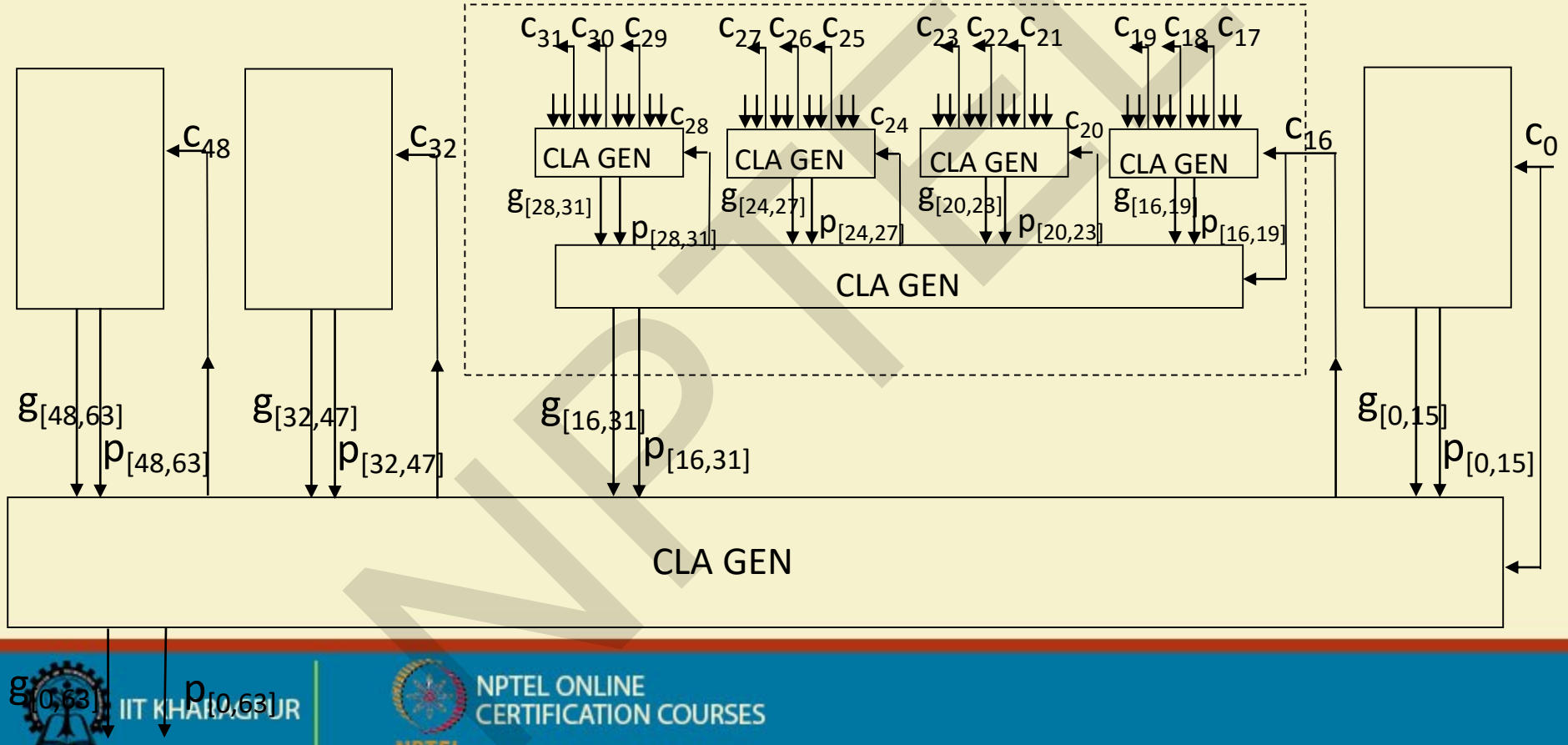
$$c_{i+1} = g_i + c_i p_i$$

16-bit 2-level Carry Lookahead Adder

Signals computed	Formulas	Delay
$s_{i+1}, s_{i+2}, s_{i+3}$		1 gate delay
	i = 4, 8, 12	
i.e., $s_5, s_6, s_7, s_9, s_{10}, s_{11}, s_{13}, s_{14}, s_{15}$		
	$s_i = p_i \oplus c_i$	

Total: 8 gate levels in the CLA adder vs. 32 gate levels in the ripple carry adder

64-bit 3-level Carry Lookahead Adder



64-bit 3-level Carry Lookahead Adder

Level	Signals computed	Delay
PRE	g_i, p_i i=0..63	1 gate delay
1	$g_{[i..i+3]}$, $p_{[i..i+3]}$ i=0, 4, 8, 12, …, 56, 60	2 gate delays
2	$g_{[i..i+15]}$, $p_{[i..i+15]}$ i=0, 16, 32, 48	2 gate delays
3	$g_{[0..63]}$, $p_{[0..63]}$ c_{16}, c_{32}, c_{48},	2 gate delays
2	c_{20}, c_{24}, c_{28}, c_{36}, c_{40}, c_{44}, c_{52}, c_{56}, c_{60}	2 gate delays
1	c_{21}, c_{22}, c_{23}, c_{25}, c_{26}, c_{27}, …, c_{61}, c_{62}, c_{63}	2 gate delays
POST	s_{21}, s_{22}, s_{23}, s_{25}, s_{26}, s_{27}, …, s_{61}, s_{62}, s_{63}	1 gate delay

Delay of a k-bit Carry-Lookahead Adder

$$T_{\text{lookahead-adder}} = 4\lceil \log_4 k \rceil$$

k	$T_{\text{lookahead-adder}}$	$T_{\text{ripple-carry-adder}}$
4	4	8
16	8	32
32	12	64
64	12	128
128	16	256
256	16	512

Decimal Adders

8421 weighted coding scheme or BCD Code

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

Forbidden codes: 1010, 1011, 1100, 1101, 1110, 1111

Decimal Adder

- Inputs: $A_3A_2A_1A_0$, $B_3B_2B_1B_0, C_{in}$ from previous decade.
- Output: C_{out} (carry to next decade), $Z_3Z_2Z_1Z_0$.
- Idea: Perform regular binary addition and then apply a corrective procedure.

Comparing Binary and BCD Sums

Decimal Sum	K	P_3	P_2	P_1	P_0	C_{out}	Z_3	Z_2	Z_1	Z_0
0-9										
10	0	1	0	1	0	1	0	0	0	0
11	0	1	0	1	1	1	0	0	0	1
12	0	1	1	0	0	1	0	0	1	0
13	0	1	1	0	1	1	0	0	1	1
14	0	1	1	1	0	1	0	1	0	0
15	0	1	1	1	1	1	0	1	0	1
16	1	0	0	0	0	1	0	1	1	0
17	1	0	0	0	1	1	0	1	1	1

Same

C_{out} is set to 0

Decimal Adder

- No correction needed when the decimal sum is between 0-9.
- Must apply a correction when the sum is between 10-19.
- Case 1:
  - 16-19: K is set to 1. Add binary quantity 0110 to $P_3P_2P_1P_0$.
  - 10-15: $KP_3P_2P_1P_0$ are set to 01010, 01011, . . , 01111. Need to add 6. Use a K-map to obtain a Boolean expression to detect these six binary combinations.

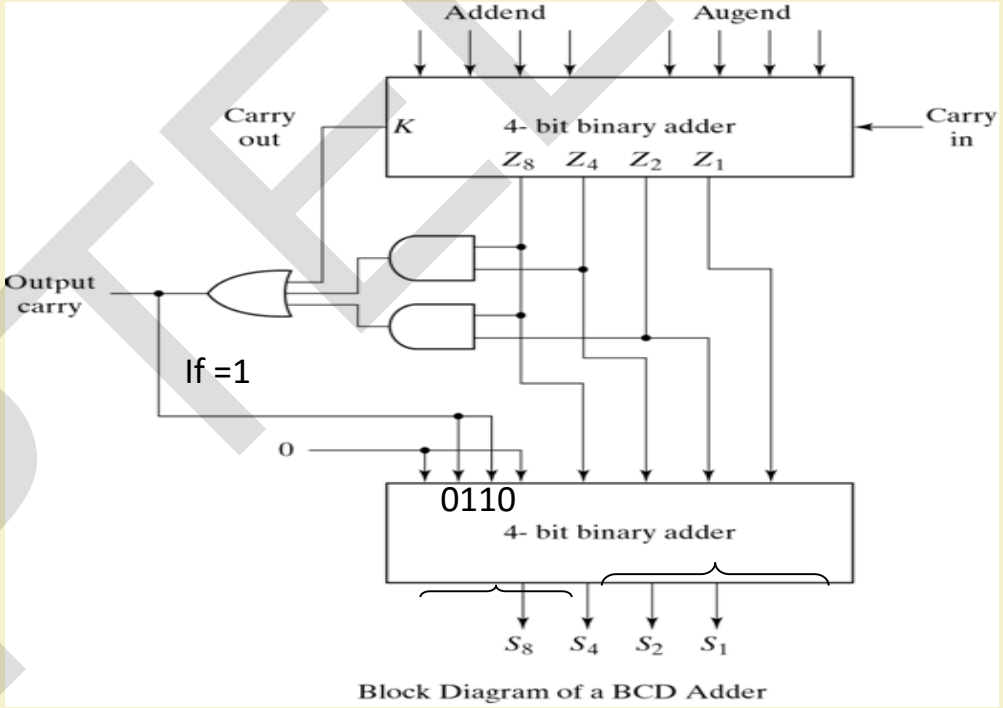
Rules of BCD adder

- 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.



Block Diagram of a BCD Adder

Binary Multiplier

Two bit Multiplication

a	b	ab
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a×b
0	0	0
0	1	0
1	0	0
1	1	1

Three bit Multiplication

- Partial products are: 101 × 1, 101 × 1, and 101 × 0
- Note that the partial product summation for *n* digit, base 2 numbers requires adding up to *n* digits (with carries) in a column.
- Note also *n* × *m* digit multiply generates up to an *m + n* digit result (same as decimal).

			1	0	1	
		×	0	1	1	
			1	0	1	
		1	0	1		
	0	0	0			
0	0	1	1	1	1	

Multiplier Boolean Equations

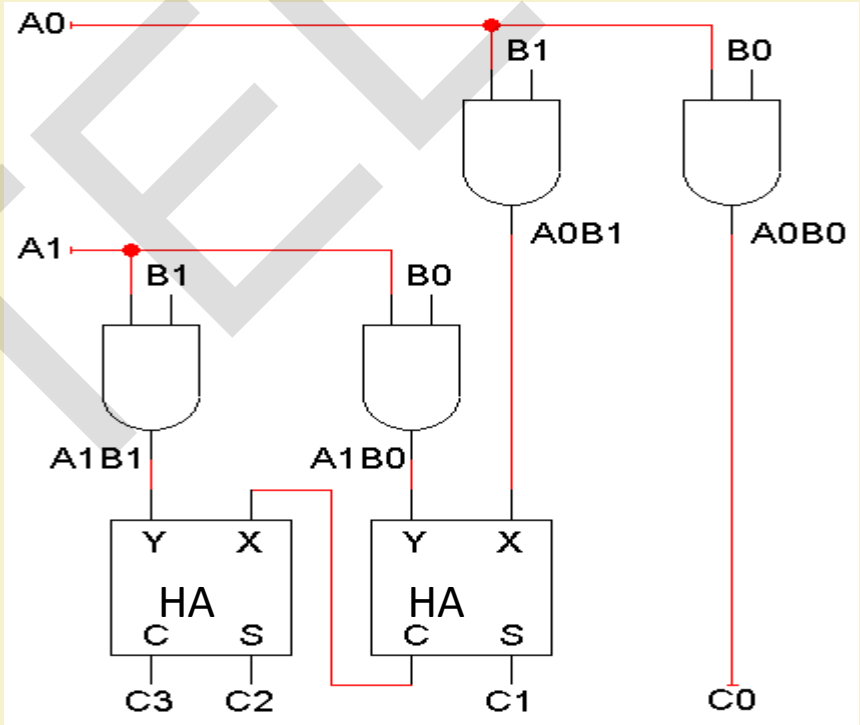
- We can also make an $n \times m$ "block" multiplier and use that to form partial products.
- Example: 2×2 – The logic equations for each partial-product binary digit are shown below:
- We need to "add" the columns to get the product bits P0, P1, P2, and P3.
- Note that some columns may generate carries.

$$
\begin{array}{cccc}
 & & b_1 & b_0 \\
 & \times & a_1 & a_0 \\
\hline
 & & (a_0 \cdot b_1) & (a_0 \cdot b_0) \\
+ & (a_1 \cdot b_1) & (a_1 \cdot b_0) & \\
\hline
P_3 & P_2 & P_1 & P_0
\end{array}
$$

A 2x2 binary multiplier

- The AND gates produce the partial products.
- For a 2-bit by 2-bit multiplier, we can just use two half adders to sum the partial products. In general, though, we'll need full adders.
- Here C_3-C_0 are the product, not carries!

			B_1	B_0
		×	A_1	A_0
			A_0B_1	A_0B_0
	+	A_1B_1	A_1B_0	
	C_3	C_2	C_1	C_0



Multiplication: a special case

- In decimal, an easy way to multiply by 10 is to shift all the digits to the left, and tack a 0 to the right end.

  128 x 10 = 1280

- We can do the same thing in binary. Shifting left is equivalent to multiplying by 2:

  11 x 10 = 110 (in decimal, 3 x 2 = 6)

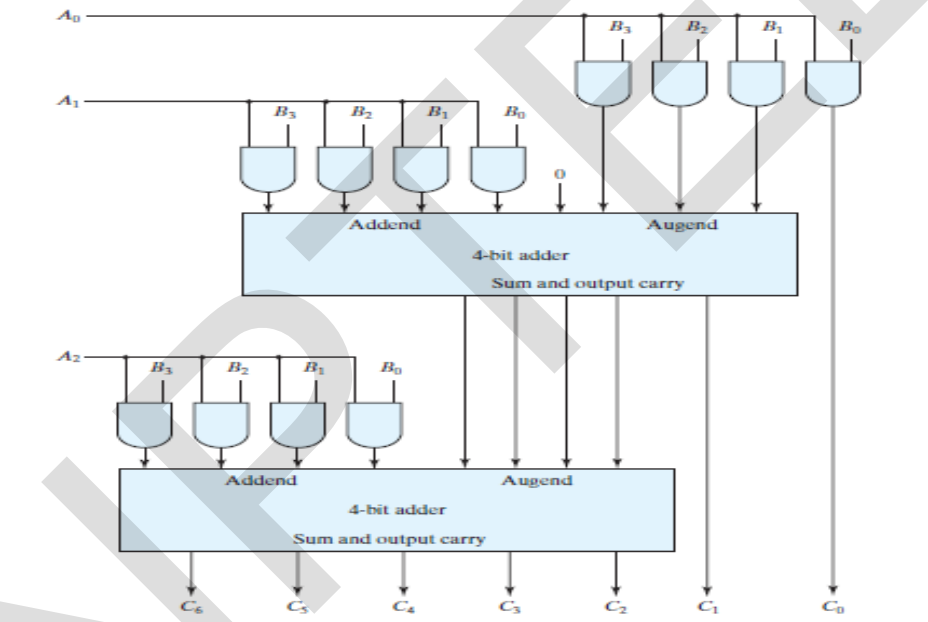
- Shifting left twice is equivalent to multiplying by 4:

  11 x 100 = 1100 (in decimal, 3 x 4 = 12)

- As an aside, shifting to the *right* is equivalent to *dividing* by 2.

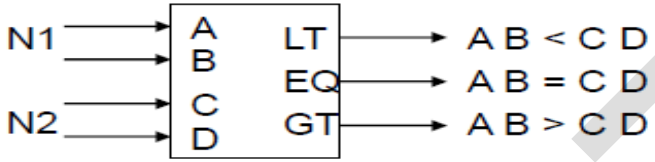
  110 ÷ 10 = 11 (in decimal, 6 ÷ 2 = 3)

4x3 Binary Multiplier



Magnitude Comparator

Two-Bit Comparator

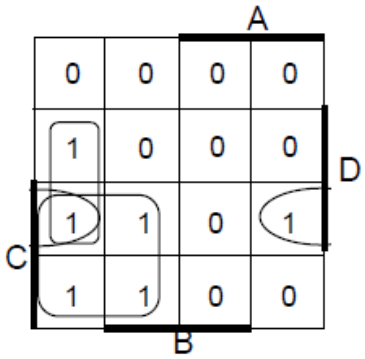


block diagram
and
truth table

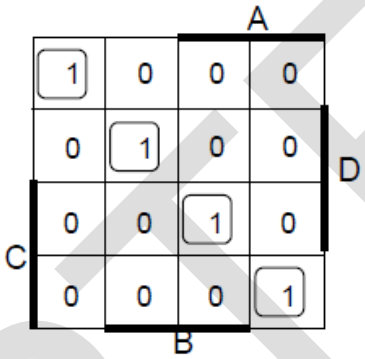
A	B	C	D	LT	EQ	GT
0	0	0	0	0	1	0
		0	1	1	0	0
		1	0	1	0	0
		1	1	1	0	0
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	1	0	0
		1	1	1	0	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	0	1	0
		1	1	1	0	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	0	1	0

we'll need a 4-variable Karnaugh map
for each of the 3 output functions

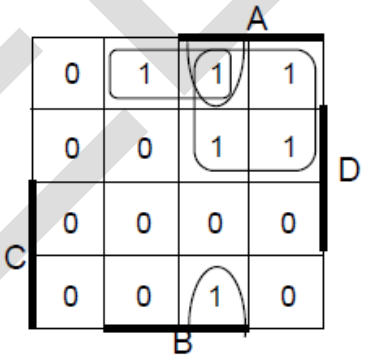
Two-Bit Comparator (cont'd)



K-map for LT



K-map for EQ



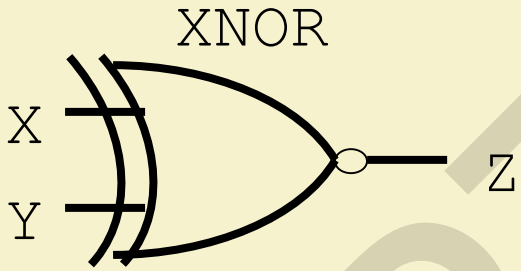
K-map for GT

LT = A' B' D + A' C + B' C D

EQ = A'B'C'D' + A'BC'D + ABCD + AB'CD' = (A xnor C) • (B xnor D)

GT = B C' D' + A C' + A B D'

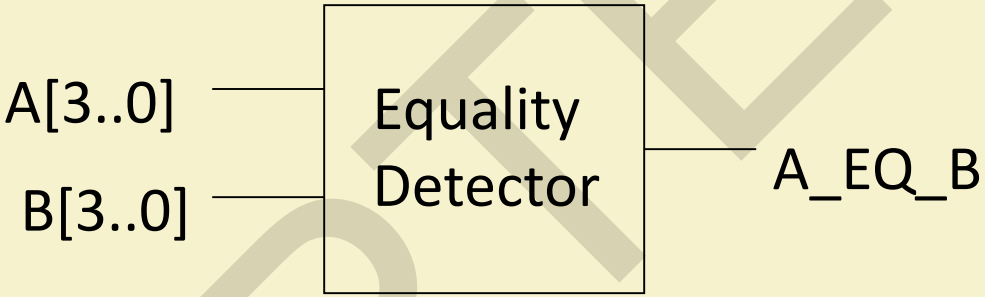
Equality Comparator



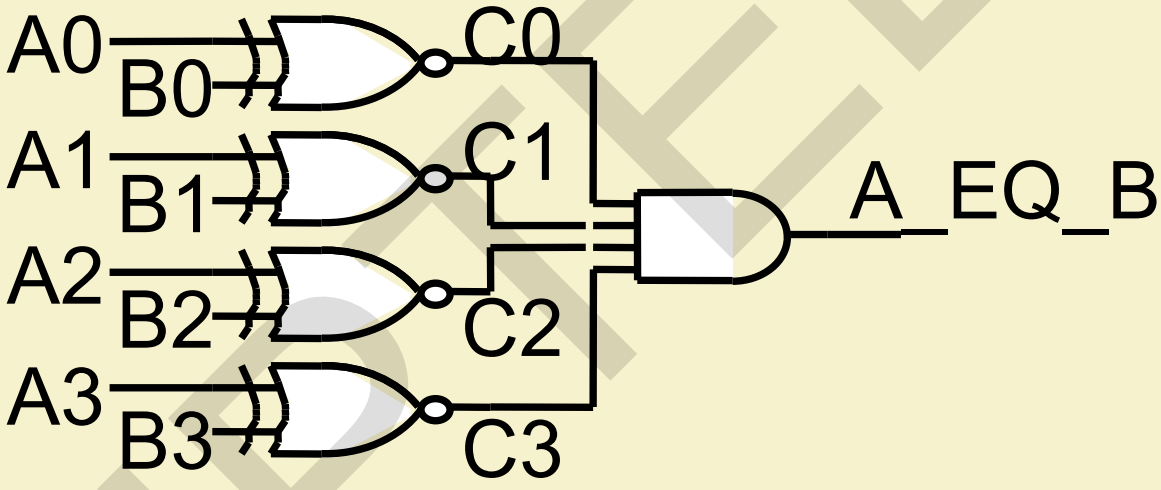
Z = X XNOR Y

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

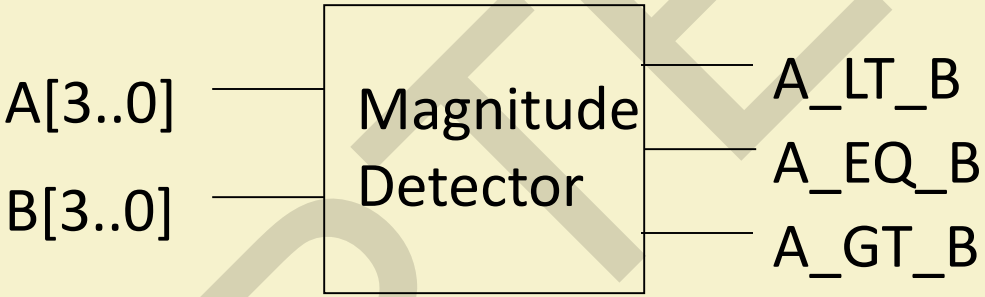
4-bit Equality Detector



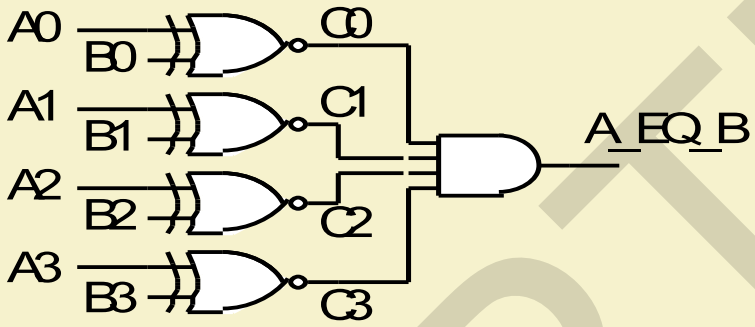
4-Bit Equality Comparator



4-bit Magnitude Comparator



Magnitude Comparator

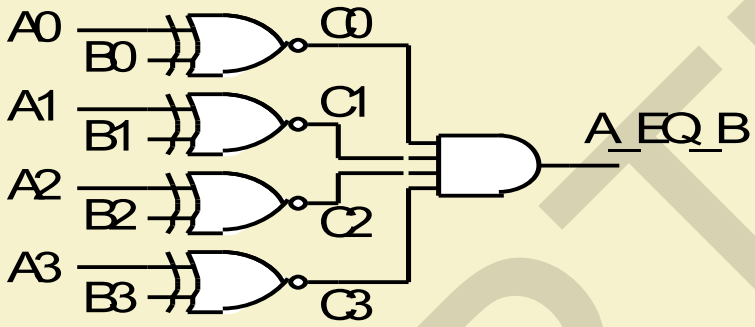


How can we find A_GT_B?

How many rows would a truth table have?

$2^8 = 256$!

Magnitude Comparator



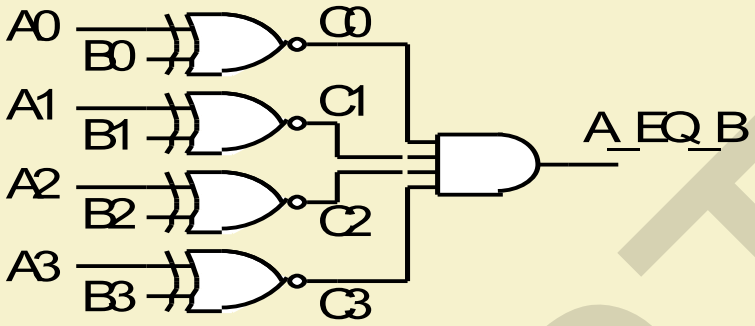
If A = 1001 and
B = 0111
is A > B?
Why?

Find A_GT_B

Because A3 > B3
i.e. A3 . B3' = 1

Therefore, one term in the logic equation for A_GT_B is A3 . B3'

Magnitude Comparator



If A = 1101 and
B = 1011
is A > B?
Why?

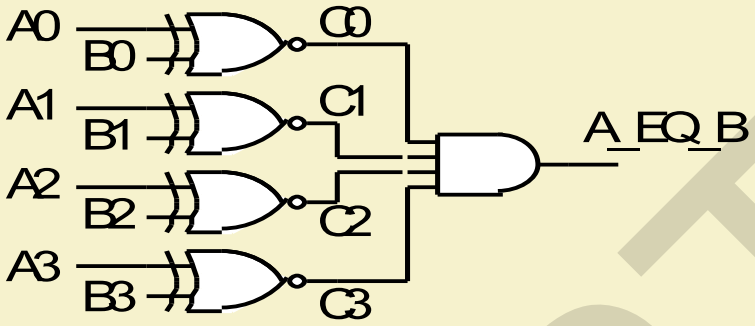
A_GT_B = A3 . B3’
+ …..

Because A3 = B3 and
A2 > B2

i.e. C3 = 1 and
A2 . B2’ = 1

Therefore, the next term in the logic equation for A_GT_B is C3 . A2 . B2’

Magnitude Comparator



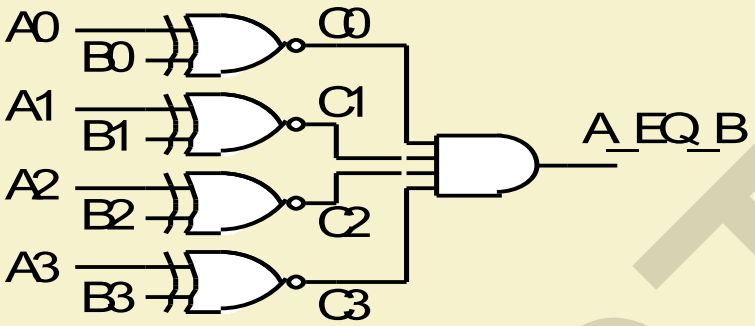
A_GT_B = A3 . B3'
+ C3 . A2 . B2'
+ .....

Because A3 = B3 and
A2 = B2 and
A1 > B1
i.e. C3 = 1 and C2 = 1 and
A1 . B1' = 1

If A = 1010 and
B = 1001
is A > B?
Why?

Therefore, the next term in the logic equation for A_GT_B is C3 . C2 . A1 . B1'

Magnitude Comparator



If A = 1011 and
B = 1010
is A > B?
Why?

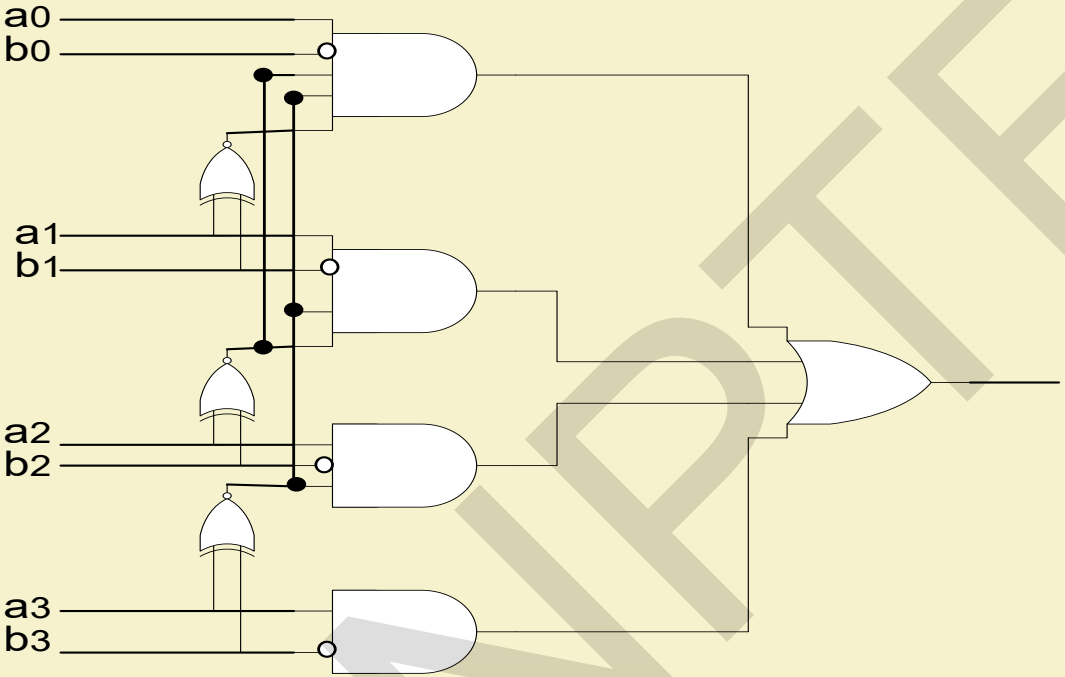
A_GT_B = A3 . B3'
+ C3 . A2 . B2'
+ C3 . C2 . A1 . B1'
+ .....

Because A3 = B3 and
A2 = B2 and
A1 = B1 and
A0 > B0

i.e. C3 = 1 and C2 = 1 and
C1 = 1 and A0 . B0' = 1

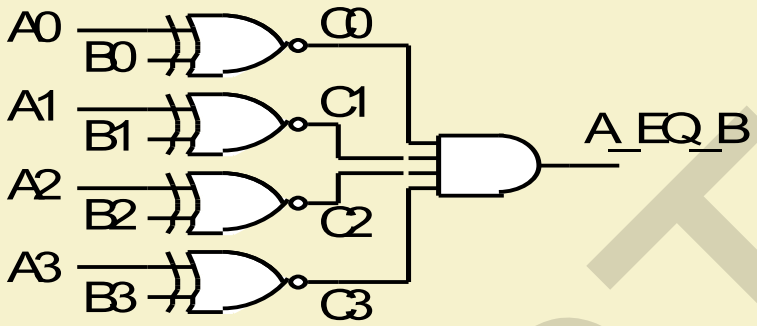
Therefore, the last term in the logic equation for A_GT_B is C3 . C2 . C1 . A0 . B0'

Magnitude Comparator



A_GT_B = A3 . B3’
+ C3 . A2 . B2’
+ C3 . C2 . A1 . B1’
+ C3 . C2 . C1 . A0 . B0’

Magnitude Comparator



Find A_LT_B

A_LT_B = A3’ . B3
+ C3 . A2’ . B2
+ C3 . C2 . A1’ . B1
+ C3 . C2 . C1 . A0’ . B0

Code Converters

CODE CONVERTERS

- A code converter is a logic circuit that changes data presented in one type of binary code to another type of binary code, such as BCD to binary, BCD to 7–segment, binary to BCD, BCD to XS3, binary to Gray code, and Gray code to binary.
- We know that, two digit decimal values ranging from 00 to 99 can be represented in BCD by two 4–bit code groups.

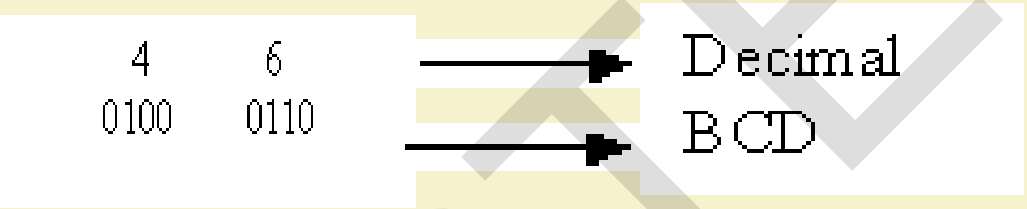
BCD-to-Binary Conversion

One method of BCD-to-Binary code conversion uses adder circuits :

1. The value, or weight, of each bit in the BCD number is represented by a binary number
2. All of the binary representations of the weights of bits that are 1s in the BCD number are added
3. The result of this addition is the binary equivalent of the BCD number

Contd...

For example, 46_{10} is represented as



- The MSB has a weight of 10, and the LSB has a weight of 1.
- So the most significant 4–bit group represents 40, and the least significant 4–bit group represents 6 as in Table.

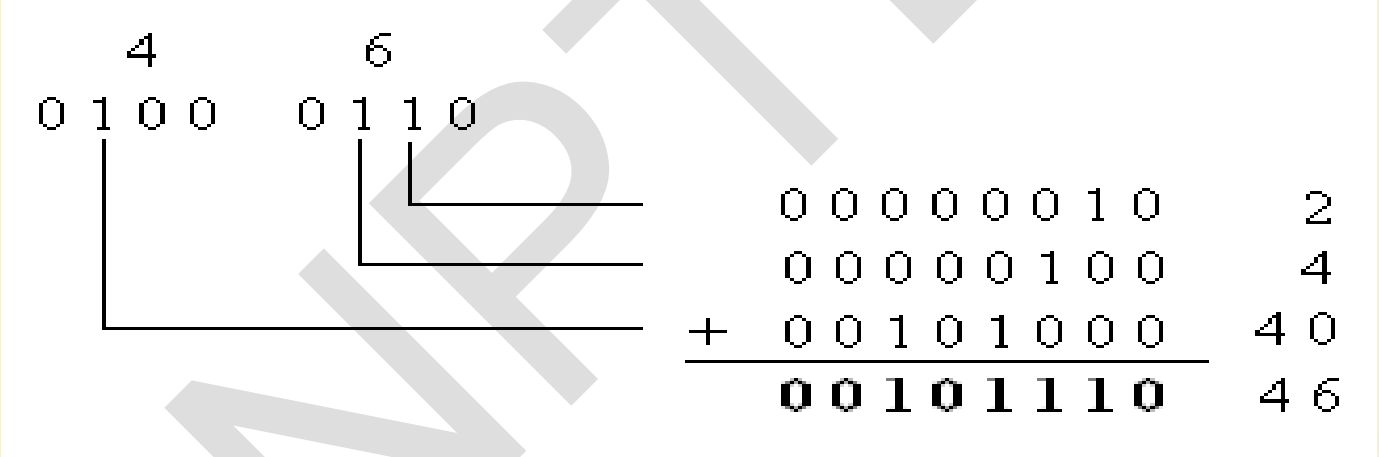
	Bit position							
Decimal weight	$10^1 = 10$				$10^0 = 1$			
Binary weight	2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0
BCD bit Weight	2^3 x 10	2^2 x 10	2^1 x 10	2^0 x 10	2^3 x 1	2^2 x 1	2^1 x 1	2^0 x 1
	80	40	20	10	8	4	2	1
Bit designation	h	g	f	e	d	c	b	a
46_{10}	0	1	0	0	0	1	1	0
		40				4	2	

Weight Table

The binary equivalent of each BCD bit is a binary number representing the BCD bit weight

BCD bit position	BCD bit weight	Binary representation
a	1	1
b	2	10
c	4	100
d	8	1000
e	10	1010
f	20	10100
g	40	101000
h	80	1010000

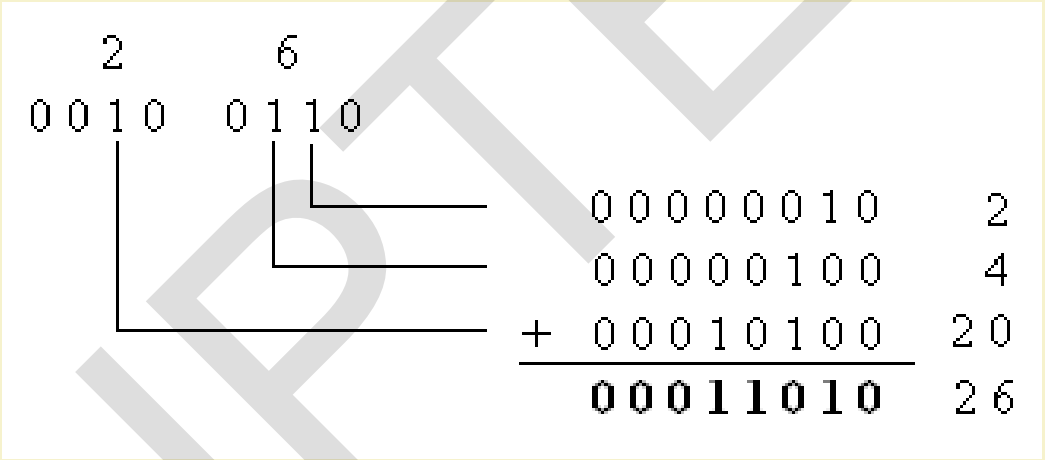
The result from the addition of the binary representation for the weights of all the 1s in the BCD number is the binary number that corresponds to the BCD number.



Example :

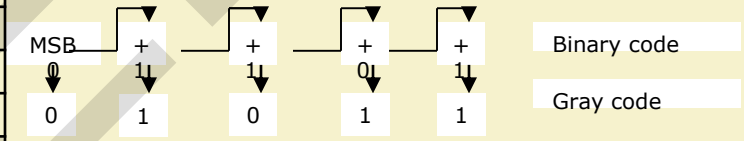
Convert the BCD equivalent of 26 to binary.

Solution



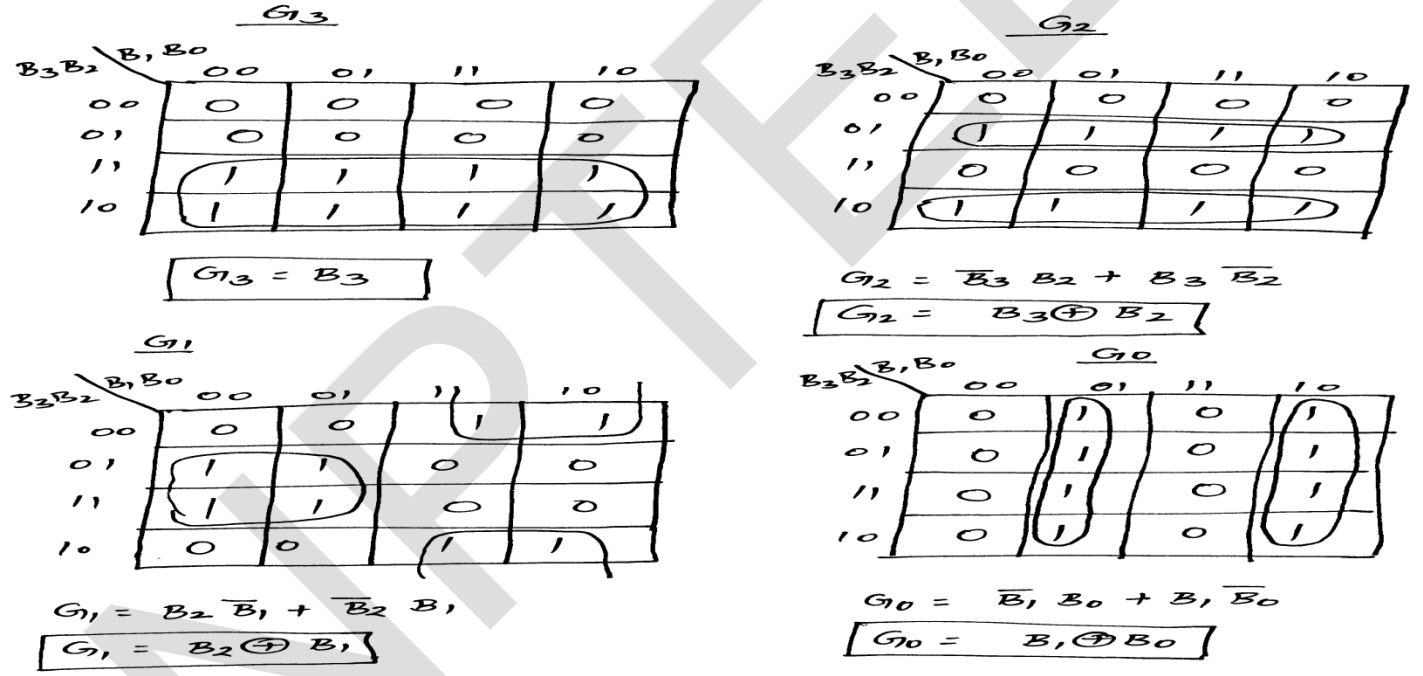
FOUR BIT BINARY TO GRAY CODE CONVERTER –DESIGN (1)…

INPUT (BINARY)					OUTPUTS (GRAY CODE)			
B3	B2	B1	B0		G3	G2	G1	G0
0	0	0	0		0	0	0	0
0	0	0	1		0	0	0	1
0	0	1	0		0	0	1	1
0	0	1	1		0	0	1	0
0	1	0	0		0	1	1	0
0	1	0	1		0	1	1	1
0	1	1	0		0	1	0	1
0	1	1	1		0	1	0	0
1	0	0	0		1	1	0	0
1	0	0	1		1	1	0	1
1	0	1	0		1	1	1	1
1	0	1	1		1	1	1	0
1	1	0	0		1	0	1	0
1	1	0	1		1	0	1	1
1	1	1	0		1	0	0	1
1	1	1	1		1	0	0	0



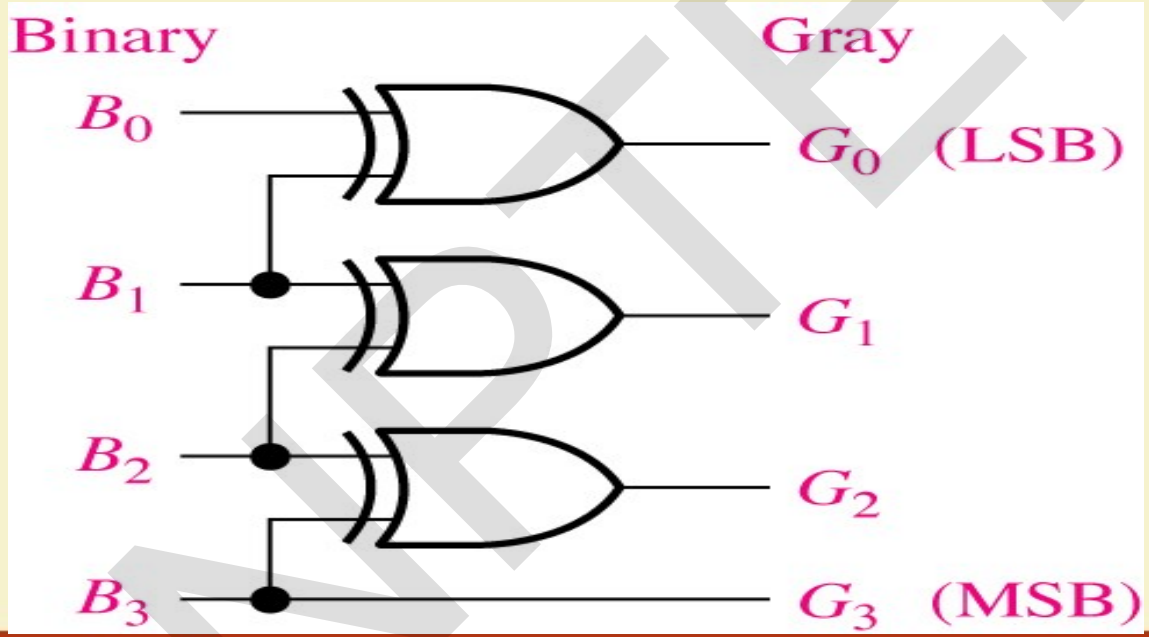
FOUR BIT BINARY TO GRAY CODE CONVERTER –DESIGN (2)…

Simplification using K-maps:



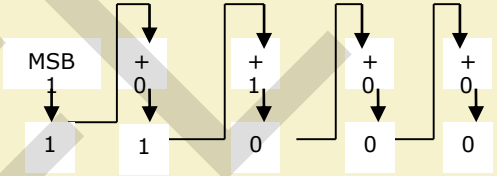
FOUR BIT BINARY TO GRAY CODE CONVERTER –DESIGN (3)

Logic Diagram:

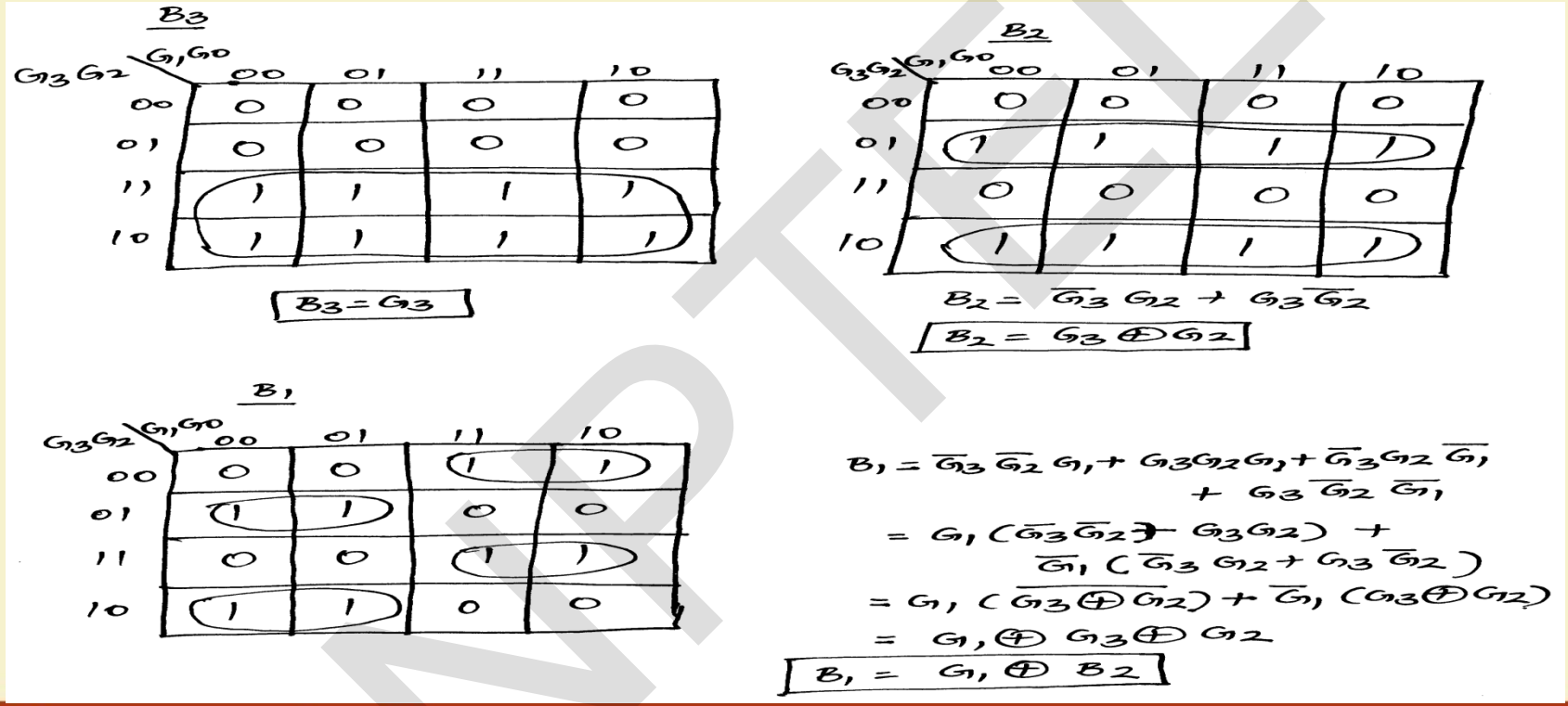


FOUR BIT GRAY CODE TO BINARY CONVERTER –DESIGN (1)…

INPUT (GRAY CODE)					OUTPUTS (BINARY)			
G3	G2	G1	G0		B3	B2	B1	B0
0	0	0	0		0	0	0	0
0	0	0	1		0	0	0	1
0	0	1	0		0	0	1	1
0	0	1	1		0	0	1	0
0	1	0	0		0	1	1	1
0	1	0	1		0	1	1	0
0	1	1	0		0	1	0	0
0	1	1	1		0	1	0	1
1	0	0	0		1	1	1	1
1	0	0	1		1	1	1	0
1	0	1	0		1	1	0	0
1	0	1	1		1	1	0	1
1	1	0	0		1	0	0	0
1	1	0	1		1	0	0	1
1	1	1	0		1	0	1	1
1	1	1	1		1	0	1	0



FOUR BIT GRAY CODE TO BINARY CONVERTER –DESIGN (2)…



FOUR BIT GRAY CODE TO BINARY CONVERTER –DESIGN (3)…

B_0

G_3G_2 \ G_1G_0	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

$$B_0 = \overline{G_3}\,\overline{G_2}\,\overline{G_1}G_0 + \overline{G_3}\,\overline{G_2}G_1\overline{G_0} + \overline{G_3}G_2\overline{G_1}\,\overline{G_0} + \overline{G_3}G_2G_1G_0 + G_3G_2\overline{G_1}G_0 + G_3G_2G_1\overline{G_0} + G_3\overline{G_2}\overline{G_1}\,\overline{G_0} + G_3\overline{G_2}G_1G_0$$

$$= \overline{G_3}\,\overline{G_2}(\overline{G_1}G_0 + G_1\overline{G_0}) + \overline{G_3}G_2(\overline{G_1}\,\overline{G_0} + G_1G_0) + G_3G_2(\overline{G_1}G_0 + G_1\overline{G_0}) + G_3\overline{G_2}(\overline{G_1}\,\overline{G_0} + G_1G_0)$$

$$= \overline{G_3}\,\overline{G_2}(G_1 \oplus G_0) + \overline{G_3}G_2(\overline{G_1 \oplus G_0}) + G_3G_2(G_1 \oplus G_0) + G_3\overline{G_2}(\overline{G_1 \oplus G_0})$$

$$= (G_1 \oplus G_0)(\overline{G_3}\,\overline{G_2} + G_3G_2) + (\overline{G_1 \oplus G_0})(G_3\overline{G_2} + \overline{G_3}G_2)$$

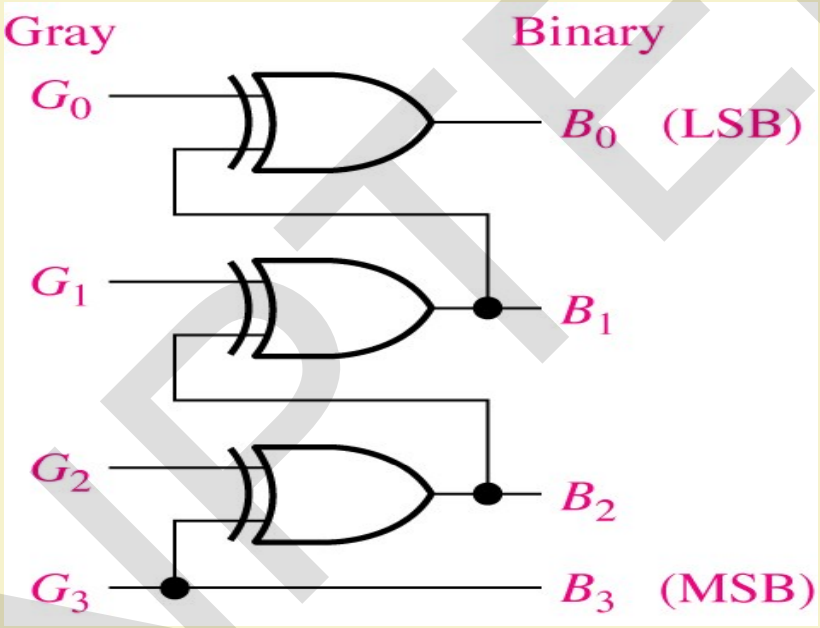
$$= (G_1 \oplus G_0)(\overline{G_3 \oplus G_2}) + (\overline{G_1 \oplus G_0})(G_3 \oplus G_2)$$

$$= G_0 \oplus G_1 \oplus G_2 \oplus G_3$$

$$\boxed{B_0 = G_0 \oplus B_1}$$

FOUR BIT GRAY CODE TO BINARY CONVERTER –DESIGN (4)

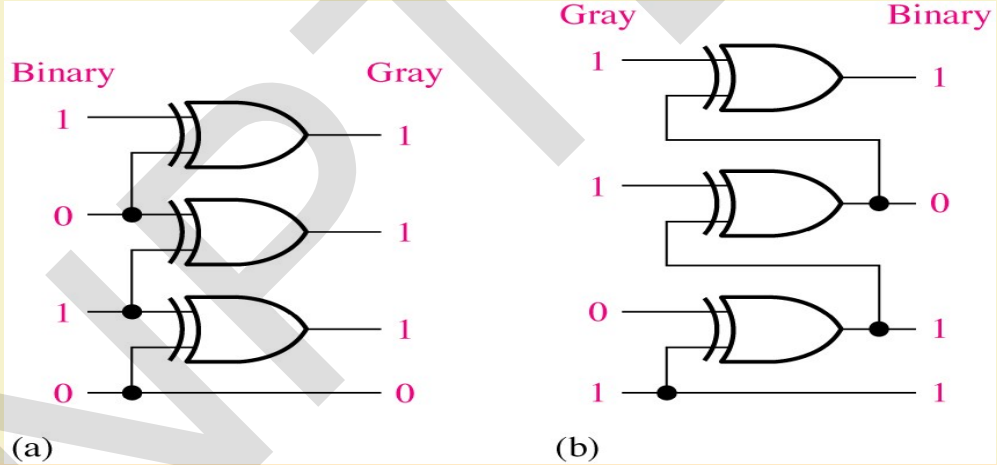
Logic Diagram:



Exercise

1. Convert the binary number 0101 to Gray code with XOR gates
2. Convert the gray code 1011 to binary with XOR gates

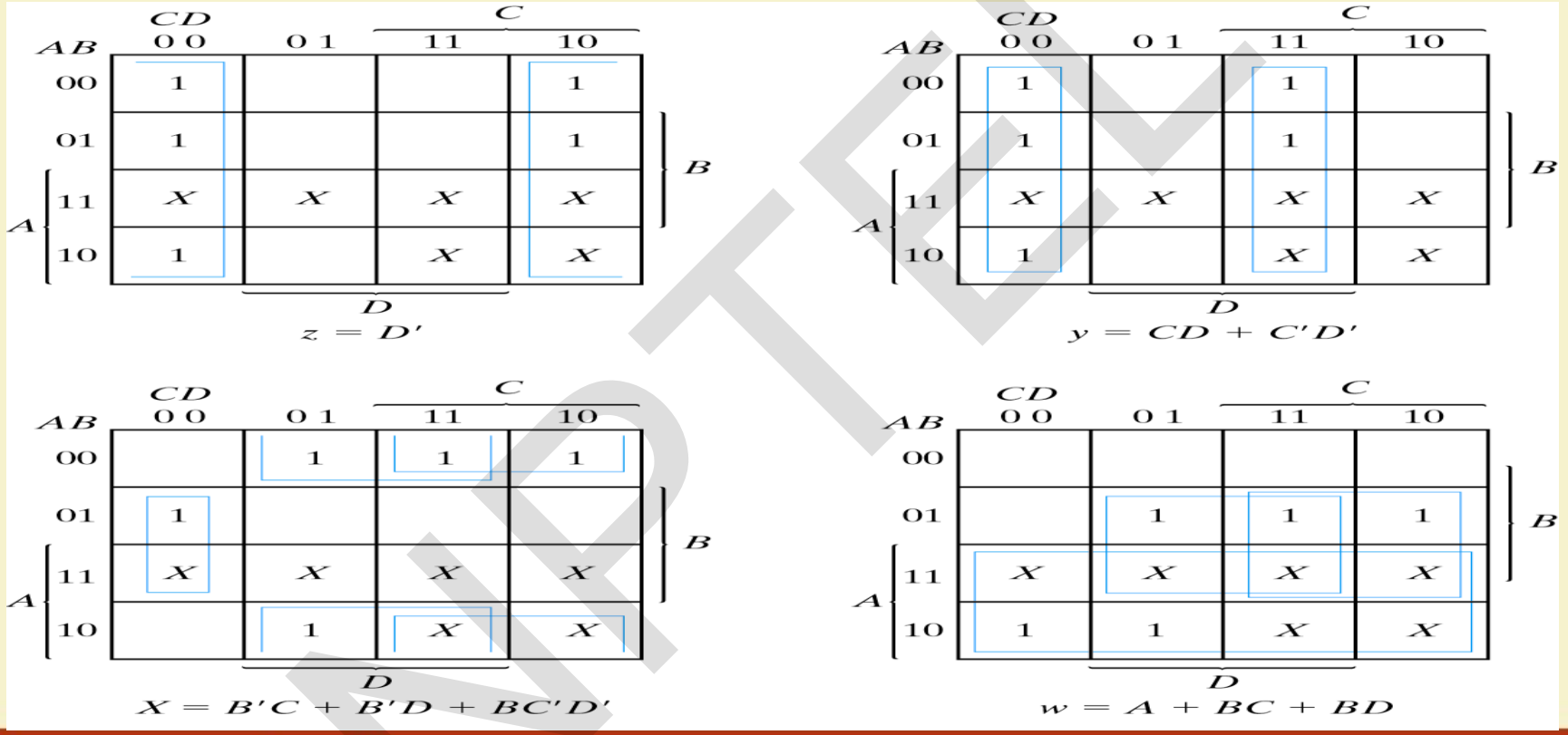
Solution:



BCD to XS 3 code converter- Design (1)...

Input (Std BCD code)					Output (XS3 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
1	0	1	0		X	X	X	X
1	0	1	1		X	X	X	X
1	1	0	1		X	X	X	X
1	1	1	0		X	X	X	X
1	1	1	1		X	X	X	X

BCD to XS 3 code converter- Design (2)...



BCD to XS 3 code converter- Design (3)...

- After the manipulation of the Boolean expressions for using common gates for two or more outputs, logic expressions can be given by

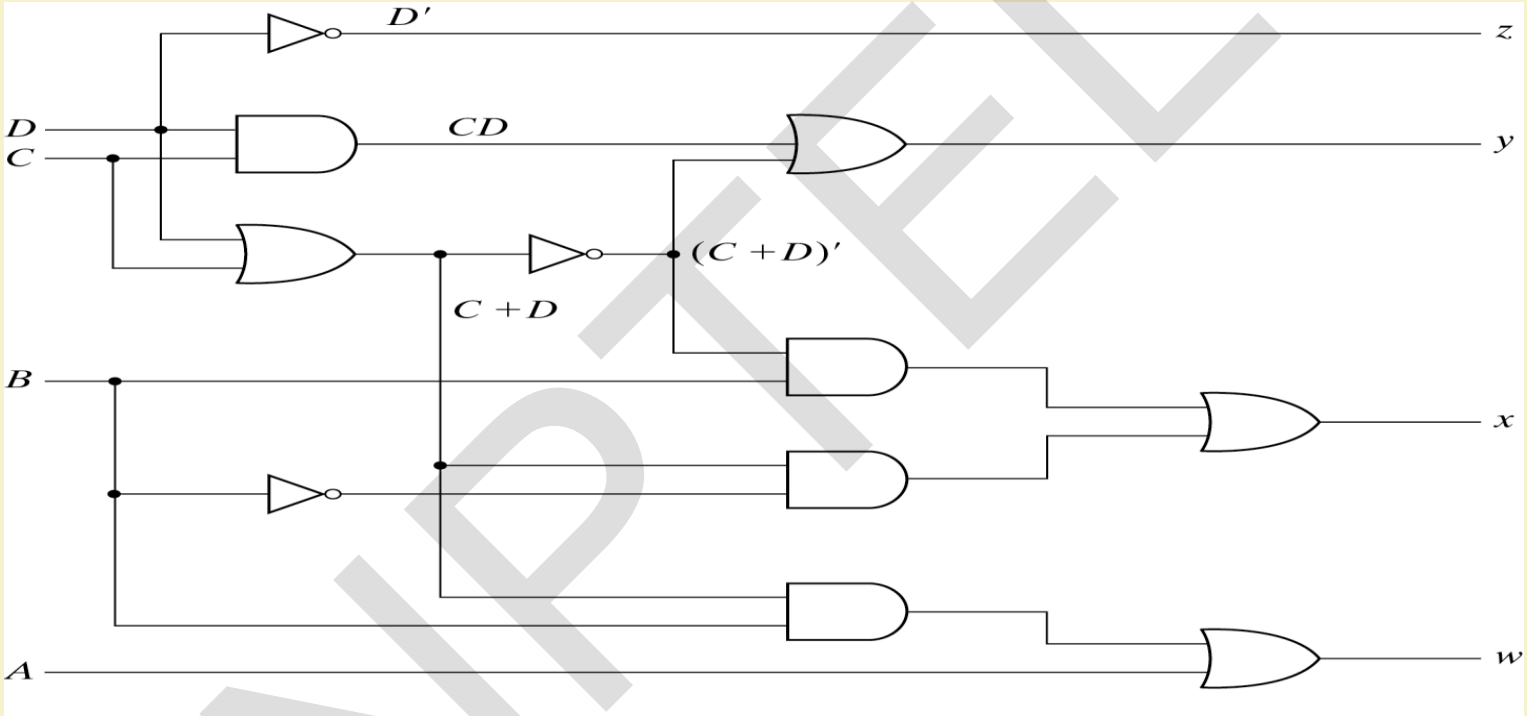
$z = D'$

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

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

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

BCD to XS 3 code converter- Design (4)



Decoders, Multiplexers, PLAs

Santanu Chattopadhyay
Electronics and Electrical Communication Engineering

Decoders

DECODER

- A decoder is a logic circuit that accepts a set of inputs that represents a binary number and activates only the output that corresponds to the input number.

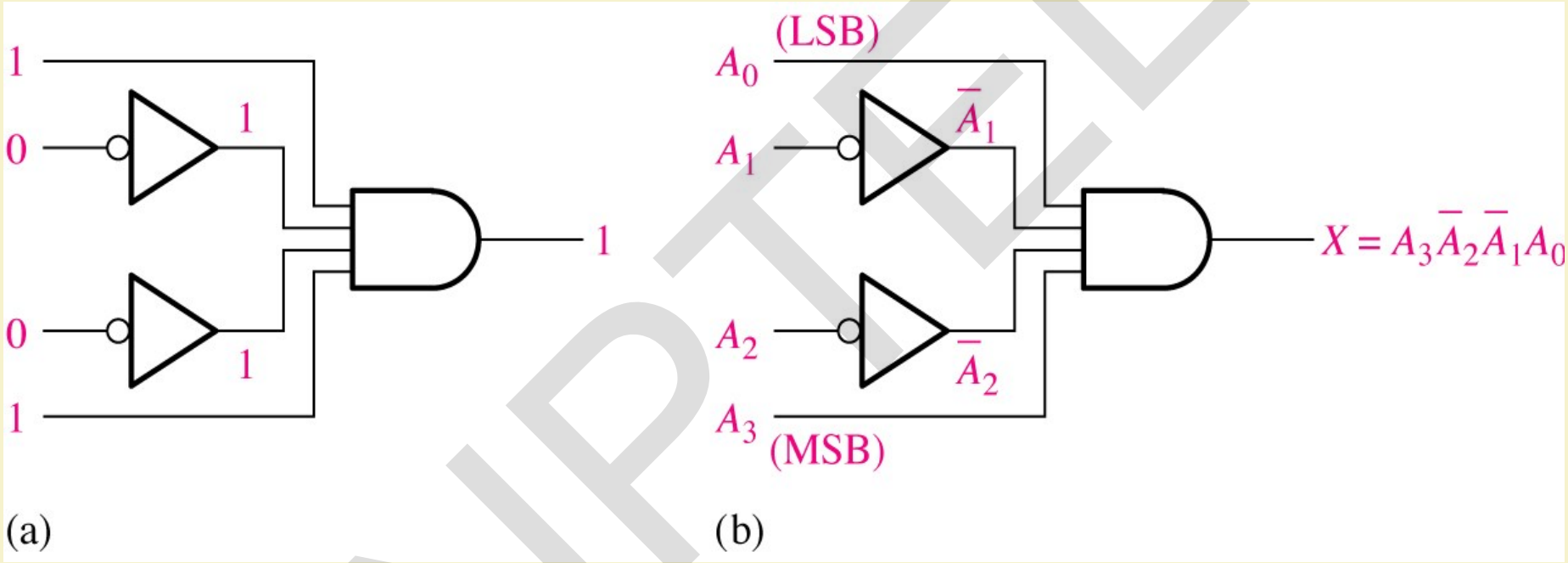
- In other words, a decoder circuit looks at its inputs, determines which binary number is present there, and activates the one output that corresponds to that number ; all other outputs remain inactive

In its general form, a decoder has N input lines to handle N bits and form one to 2^N output lines to indicate the presence of one or more N-bit combinations.

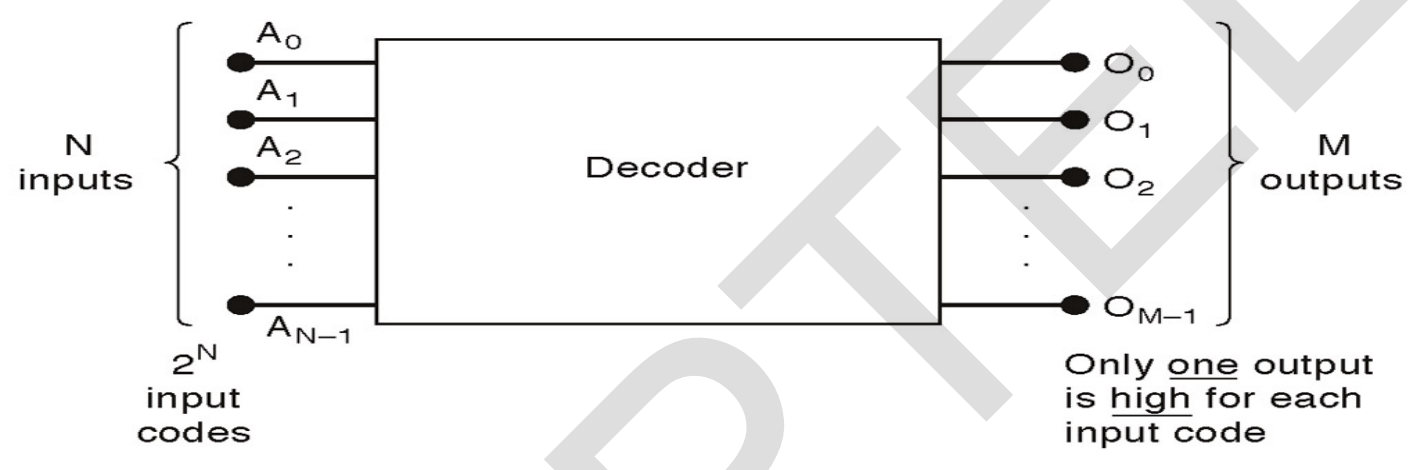
The basic binary function

- An AND gate can be used as the basic decoding element because it produces a HIGH output only when all inputs are HIGH

Decoding logic for the binary code 1001 with an active-HIGH output.



General decoder diagram

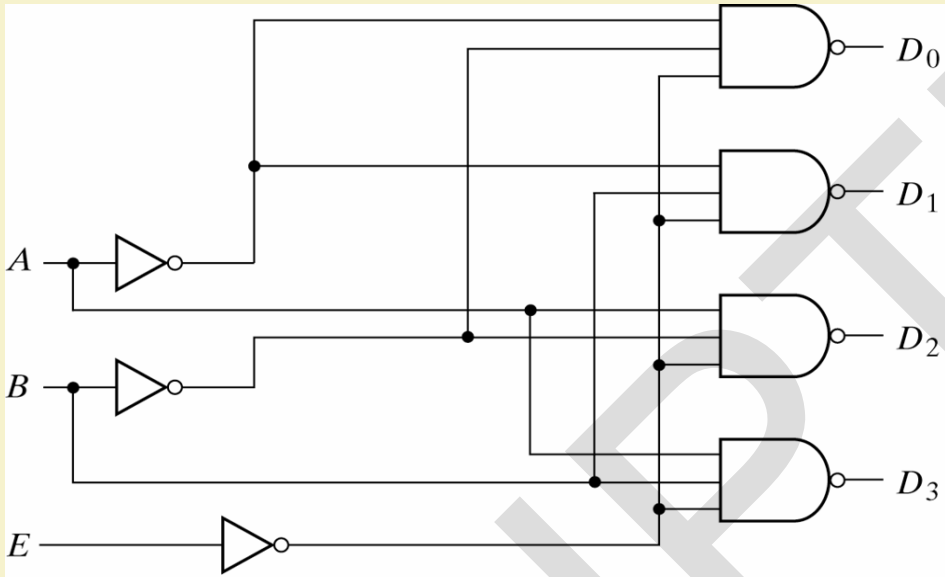


There are 2^N possible input combinations, from A_0 to A_{N-1}.

For each of these input combinations only one of the *M* outputs will be active *HIGH* (1), all the other outputs are *LOW* (0).

- If an active-LOW output (74138, one of the output will low and the rest will be high) is required for each decoded number, the entire decoder can be implemented with
    1. NAND gates
    2. Inverters
- If an active-HIGH output (74139, one of the output will high and the rest will be low) is required for each decoded number, the entire decoder can be implemented with
    - AND gates
    - Inverters

2-to-4-Line Decoder
(with Enable input)-Active LOW output (1)...



(a) Logic diagram

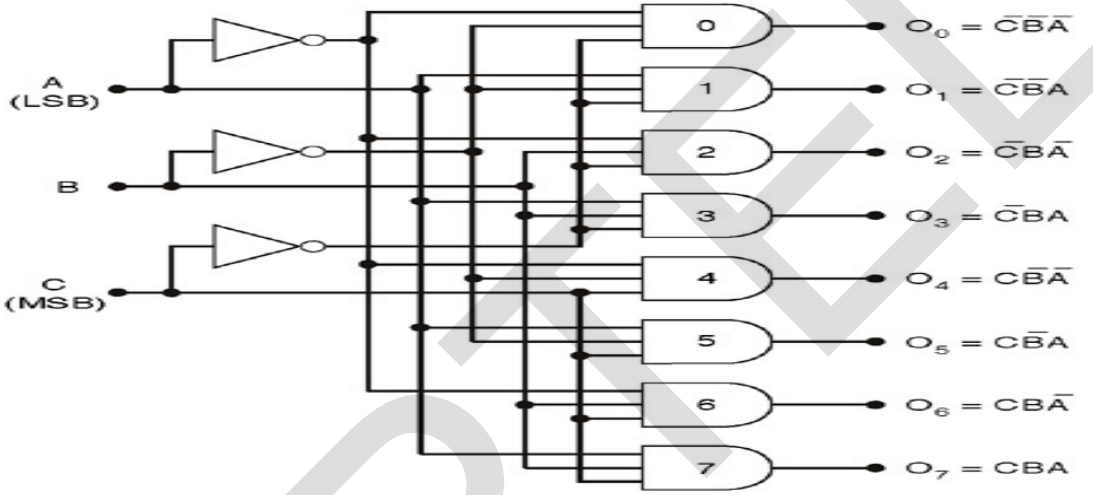
E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) Truth table

2-to-4-Line Decoder (with Enable input)-Active LOW output (2)

- The circuit operates with complemented outputs and a complement enable input. The decoder is enabled when E is equal to 0.
- Only one output can be equal to 0 at any given time, all other outputs are equal to 1.
- The output whose value is equal to 0 represents the minterm selected by inputs A and B
- The circuit is disabled when E is equal to 1.

3-8 line decoder (active-HIGH)

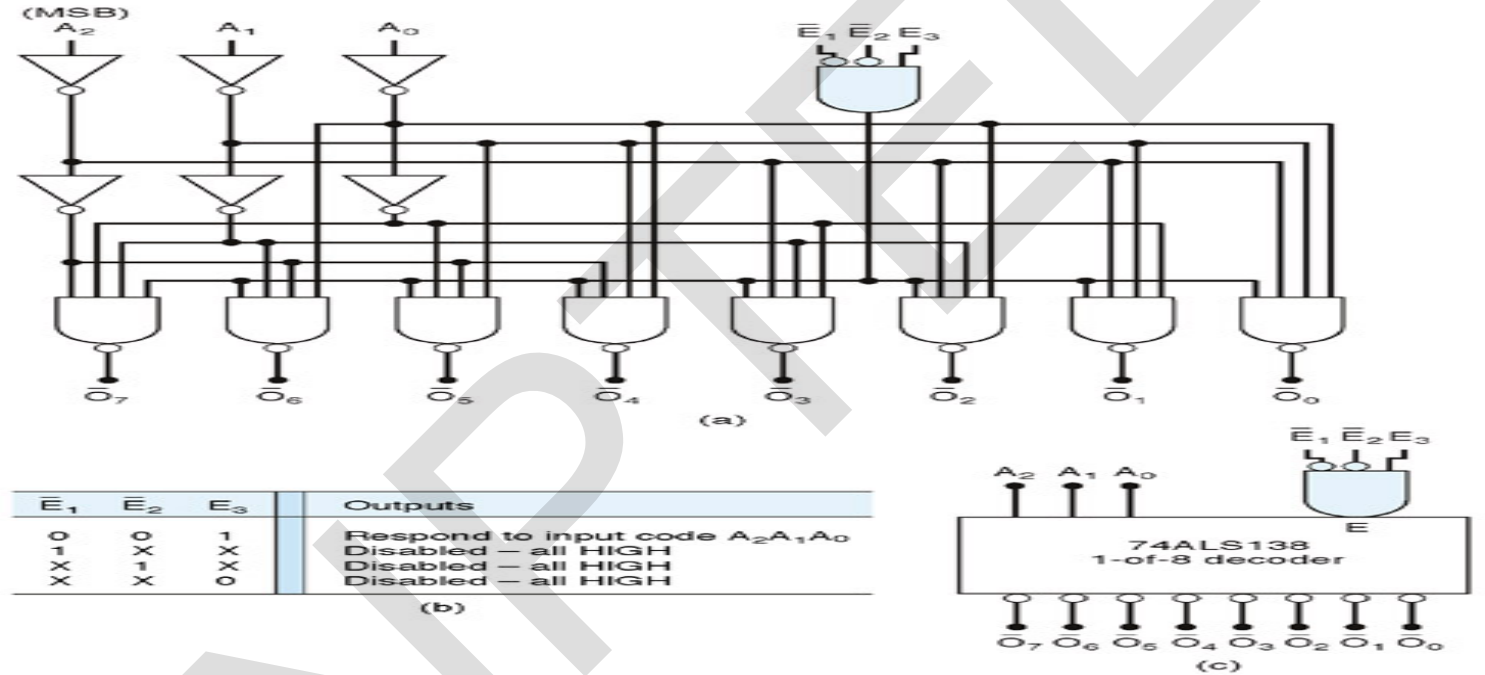


C	B	A	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

•This decoder can be referred to in several ways. It can be called a 3-line-to- 8-line decoder, because it has three input lines and eight output lines.

•It could also be called a binary-octal decoder or converters because it takes a three bit binary input code and activates the one of the eight outputs corresponding to that code. It is also referred to as a 1-of-8 decoder, because only 1 of the 8 outputs is activated at one time.

Logic diagram of 74138 (Example of a 3–Bit Decoder)



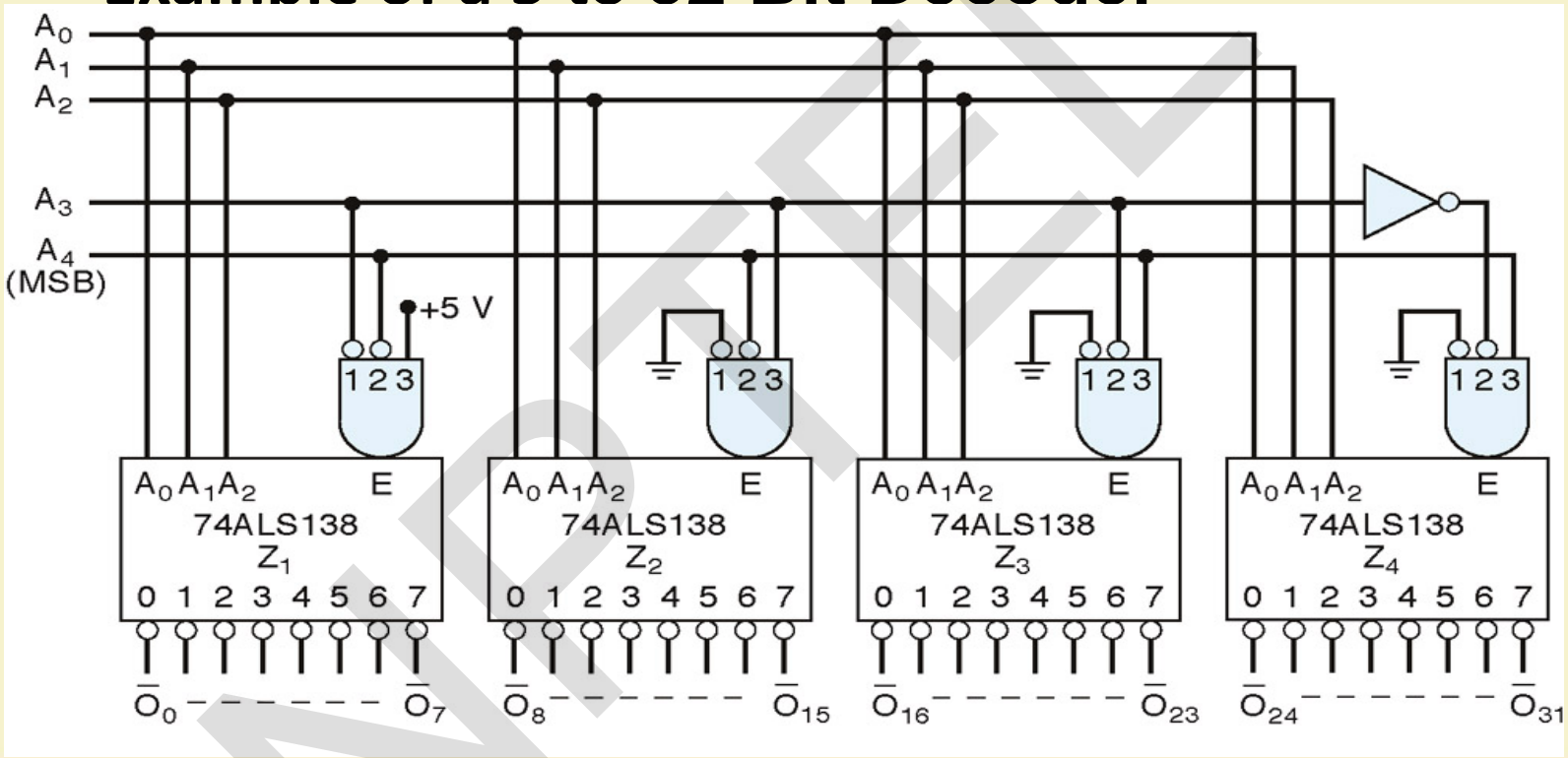
Truth table of 74138 (Example of a 3– 8 Bit Decoder) active-LOW

Inputs						Outputs							
Enables			2^2	2^1	2^0	Active–*LOW*							
E_3	$\overline{E}_1$	$\overline{E}_2$	A_2	A_1	A_0	$\overline{O}_7$	$\overline{O}_6$	$\overline{O}_5$	$\overline{O}_4$	$\overline{O}_3$	$\overline{O}_2$	$\overline{O}_1$	$\overline{O}_0$
X	X	H	X	X	X	H	H	H	H	H	H	H	H
X	H	X	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H	**L**
H	L	L	L	L	H	H	H	H	H	H	H	**L**	H
H	L	L	L	H	L	H	H	H	H	H	**L**	H	H
H	L	L	L	H	H	H	H	H	H	**L**	H	H	H
H	L	L	H	L	L	H	H	H	**L**	H	H	H	H
H	L	L	H	L	H	H	H	**L**	H	H	H	H	H
H	L	L	H	H	L	H	**L**	H	H	H	H	H	H
H	L	L	H	H	H	**L**	H	H	H	H	H	H	H

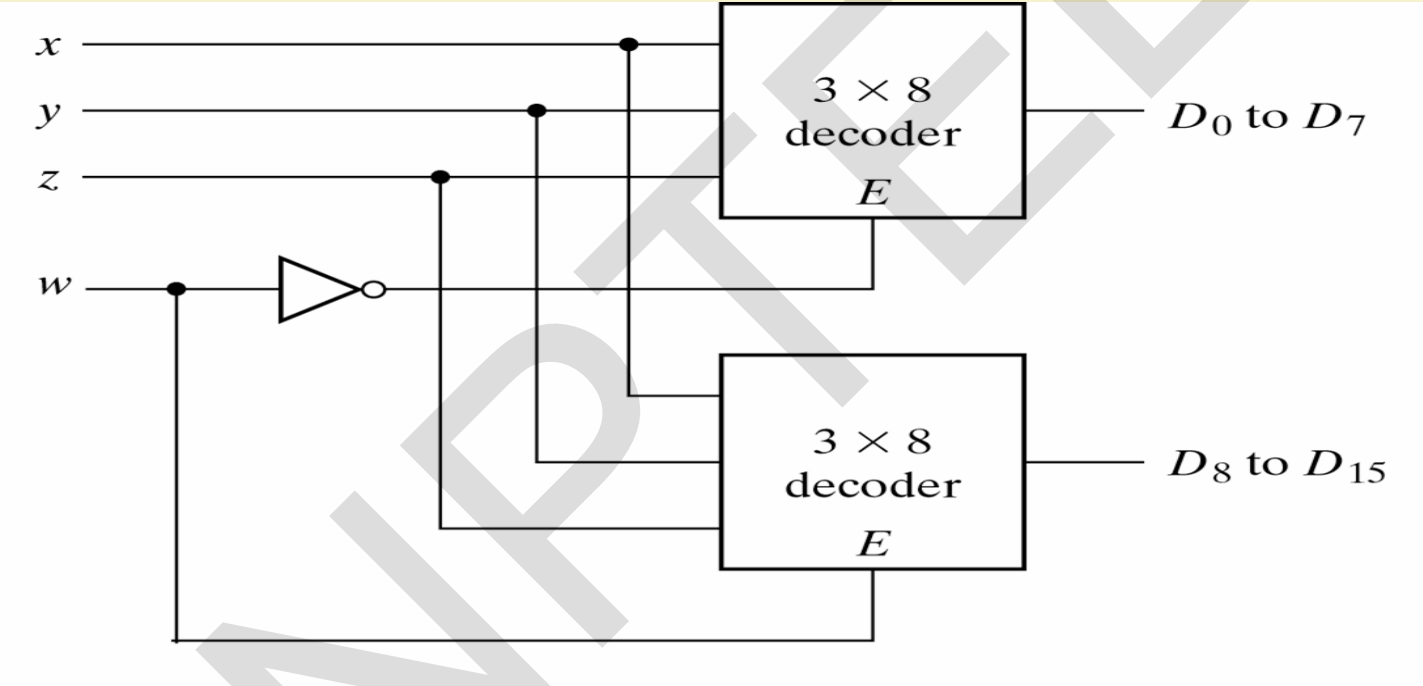
74138 (Example of a 3– 8 Bit Decoder)

- There is an enable function on this device, a *LOW* level on each input E'_1, and E'_2, and a *HIGH* level on input E_3, is required in order to make the enable gate output *HIGH*.
- The enable is connected to an input of each NAND gate in the decoder, so it must be *HIGH* for the NAND gate to be enabled.
- If the enable gate is not activated then all eight decoder outputs will be *HIGH* regardless of the states of the three input variables A_0, A_1, and A_2 .

Example of a 5 to 32 Bit Decoder



4-line-to-16 line Decoder constructed with two 3-line-to-8 line decoders (1)...



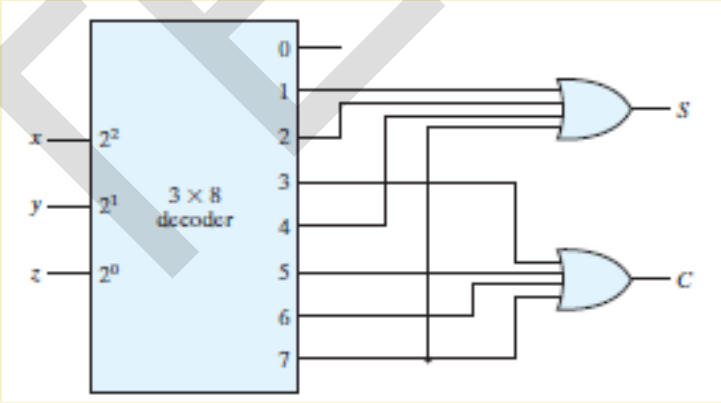
4-line-to-16 line Decoder constructed with two 3-line-to-8 line decoders (2)

- When w=0, the top decoder is enabled and the other is disabled. The bottom decoder outputs are all 0's , and the top eight outputs generate min-terms 0000 to 0111.
- When w=1, the enable conditions are reversed. The bottom decoder outputs generate min-terms 1000 to 1111, while the outputs of the top decoder are all 0's.

Combinational logic implementation

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

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



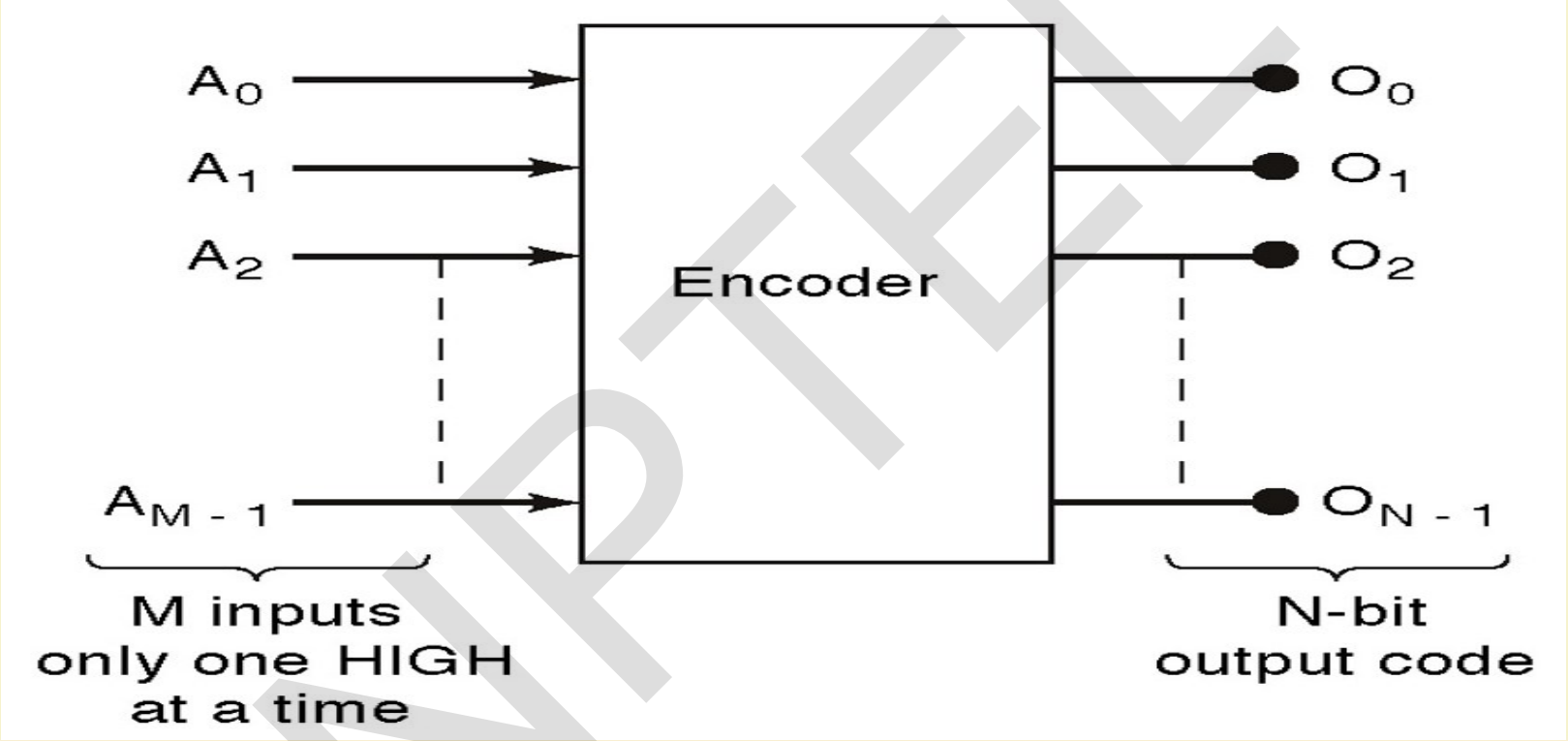
Encoders

Encoder

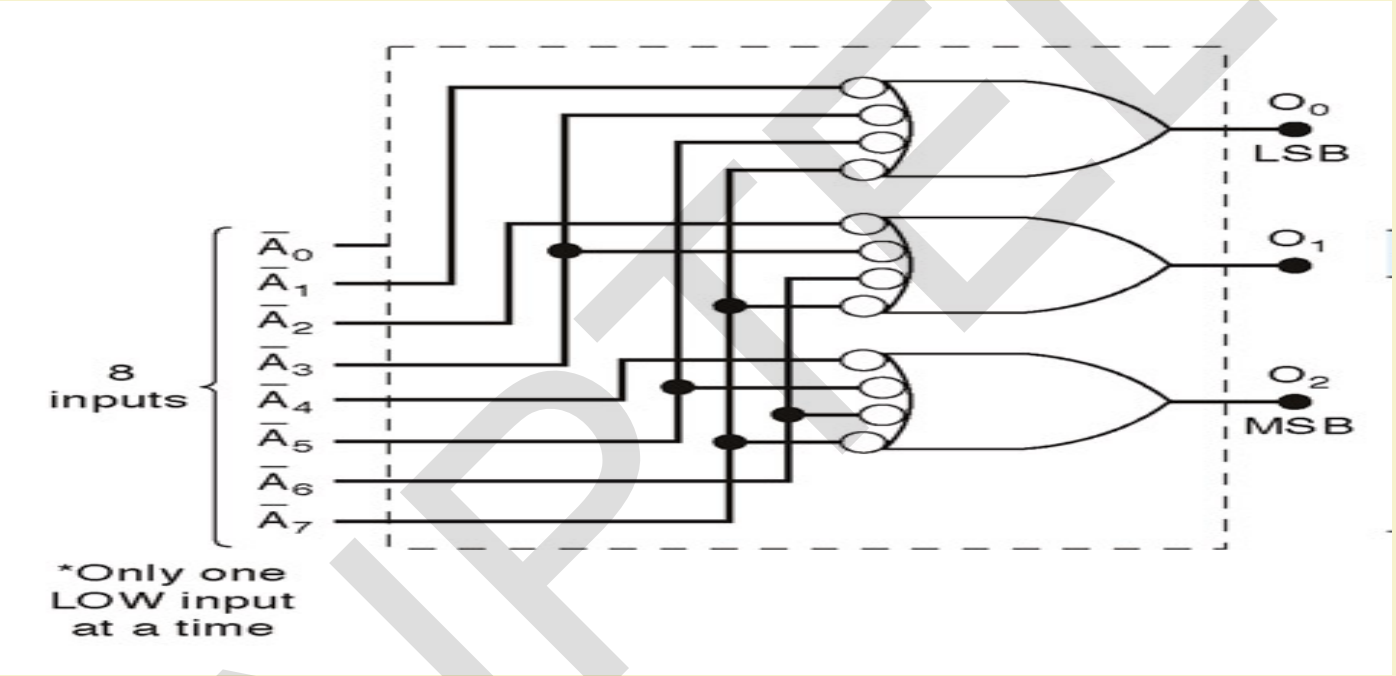
- An encoder is a combinational logic circuit that essentially performs a "reverse" of decoder functions.
- An encoder accepts an active level on one of its inputs, representing digit, such as a decimal or octal digits, and converts it to a coded output such as BCD or binary.
- Encoders can also be devised to encode various symbols and alphabetic characters.
- The process of converting from familiar symbols or numbers to a coded format is called ***encoding***.

- Most decoders accept an input code and produce a HIGH (or a LOW) at one and only one output line.
- In otherworlds , a decoder identifies, recognizes, or detects a particular code. The opposite of this decoding process is called encoding and is performed by a logic circuit called an encoder.
- An encoder has a number of input lines, only one of which input is activated at a given time and produces an N-bit output code,depending on which input is activated.

General encoder diagram



Logic circuit for octal-to binary encoder [8-line- 3-line]



Truth table for octal-to binary encoder [8-line- 3-line]

Inputs								Outputs		
$\bar{A}_0$	$\bar{A}_1$	$\bar{A}_2$	$\bar{A}_3$	$\bar{A}_4$	$\bar{A}_5$	$\bar{A}_6$	$\bar{A}_7$	O_2	O_1	O_0
X	1	1	1	1	1	1	1	0	0	0
X	0	1	1	1	1	1	1	0	0	1
X	1	0	1	1	1	1	1	0	1	0
X	1	1	0	1	1	1	1	0	1	1
X	1	1	1	0	1	1	1	1	0	0
X	1	1	1	1	0	1	1	1	0	1
X	1	1	1	1	1	0	1	1	1	0
X	1	1	1	1	1	1	0	1	1	1

A low at any single input will produce the output binary code corresponding to that input. For instance , a low at A_3' will produce O_2 =0, O_1=1 and O_0 =1, which is binary code for 3. A_o' is not connected to the logic gates because the encoder outputs always be normally at 000 when none of the inputs is LOW

Design of 4-input Priority Encoder
(4-line-to 2 line priority encoder) (1)...

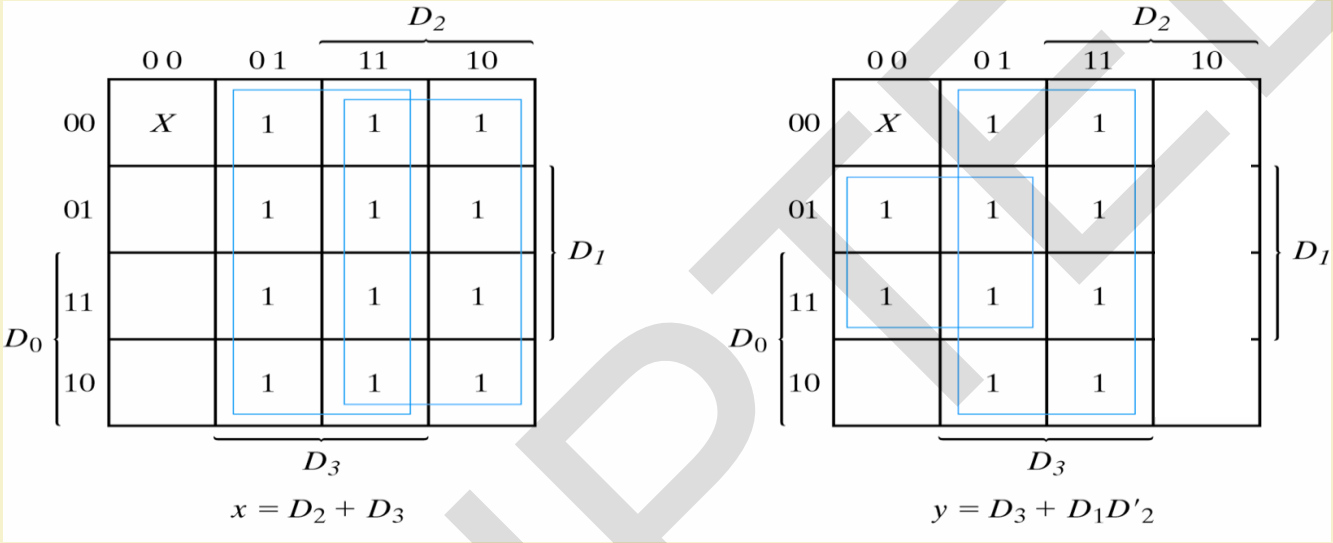
- A priority encoder is an encoder that includes the priority function
- If two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.
- **Truth Table of a 4-input Priority Encoder:**

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

Design of 4-input Priority Encoder (4-line-to 2 line priority encoder) (2)...

- In addition to two outputs x, and y, the truth table has a third output designated by V, which is a valid bit indicator that is set 1 when one or more inputs are equal to 1. If all inputs are 0, there is no valid input and V is equal to 0.
- X's in the output column indicate don't care conditions, the X's in the input columns are useful for representing a truth table in condensed form.
- The higher the subscript number, the higher the priority of the input. Input D3 has the highest priority, so regardless of the values of the other inputs, when this input is 1, the output for xy is 11 (binary 3)

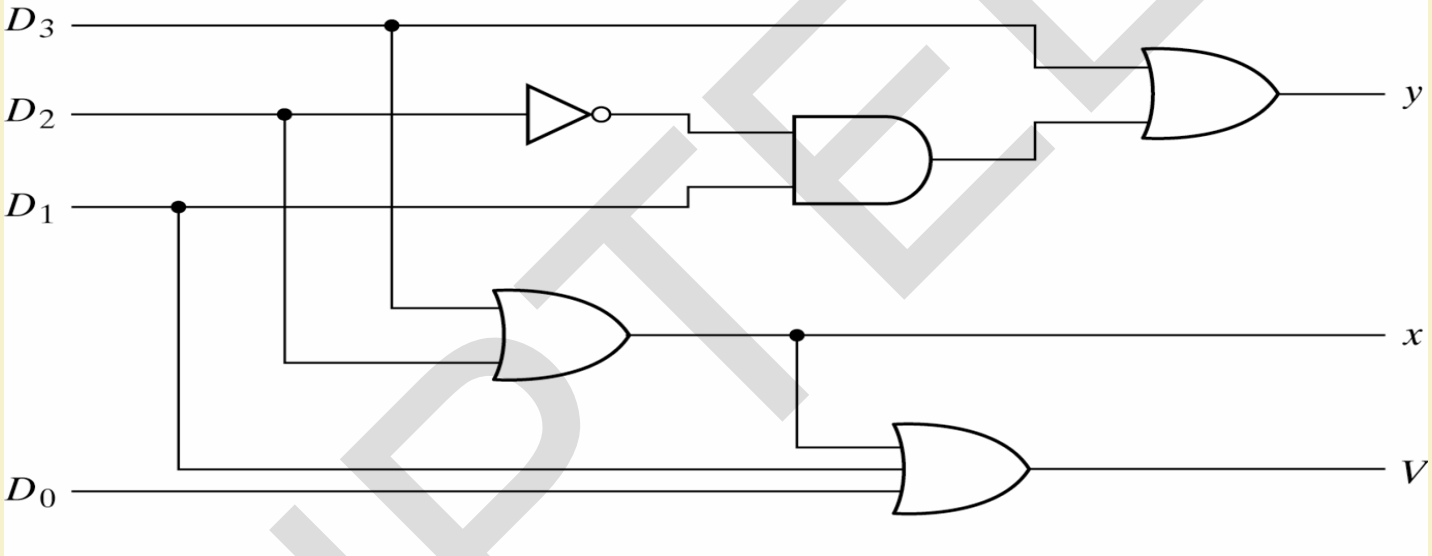
Design of 4-input Priority Encoder (4-line-to 2 line priority encoder) (3)...



$V=D_0+D_1+D_2+D_3$

K-Maps for 4-input Priority Encoder

Design of 4-input Priority Encoder
(4-line-to 2 line priority encoder) (4)



Logic Diagram for 4-input priority encoder

Multiplexers

MULTIPLEXERS (Data Selectors)

A multiplexers (MUX) is a device that allows digital information from several sources to be routed onto a single line for transmission over that line to a common destination.

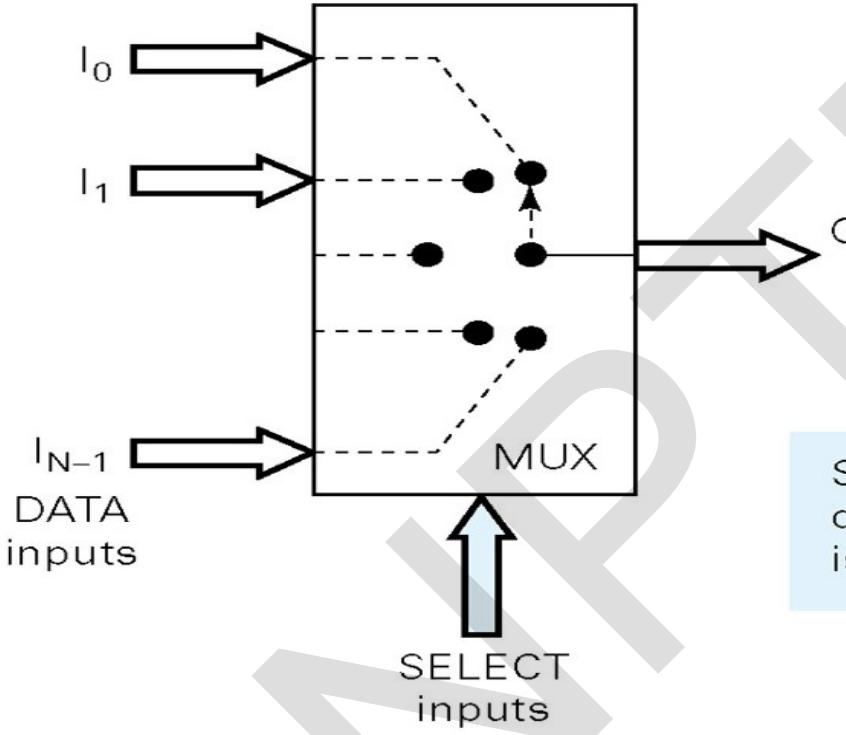
The basic multiplexers has several data input lines and a single output line. It also has data- select inputs, which permit digital data on any one of the inputs to be switched to the output line.

MUX-continued...

A modern stereo system may have a switch that selects music from one of four sources: a cassette tape, CD, a radio tuner , or an auxiliary input such as audio from a VCR or DVD. The switch selects one of the electronic signals from one of these four sources and sends it to the power amplifier and speakers.

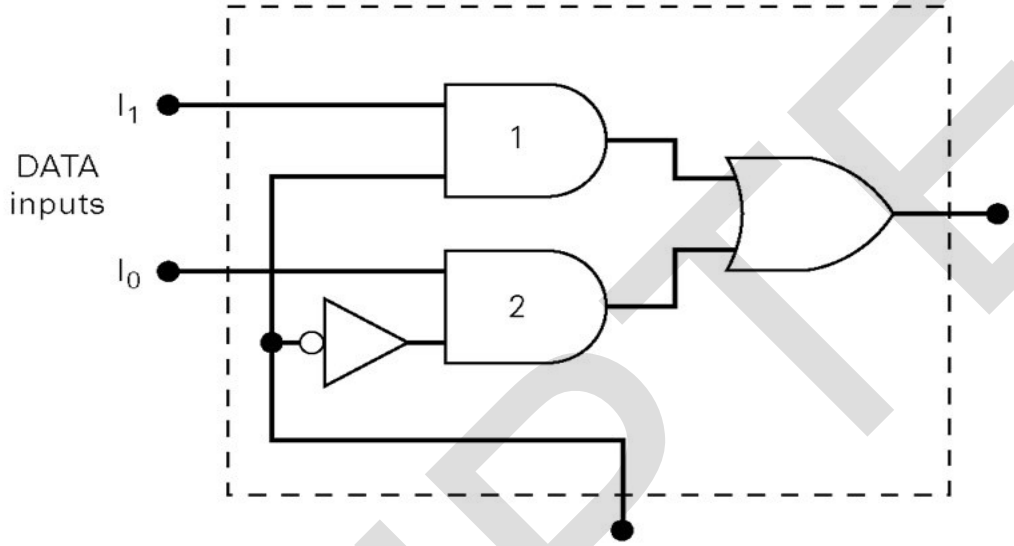
In simple terms, this is what a multiplexer (MUX) does; it selects one of several input signals and passes it on to the output.

Functional diagram of MUX



SELECT input code determines which input is transmitted to output Z.

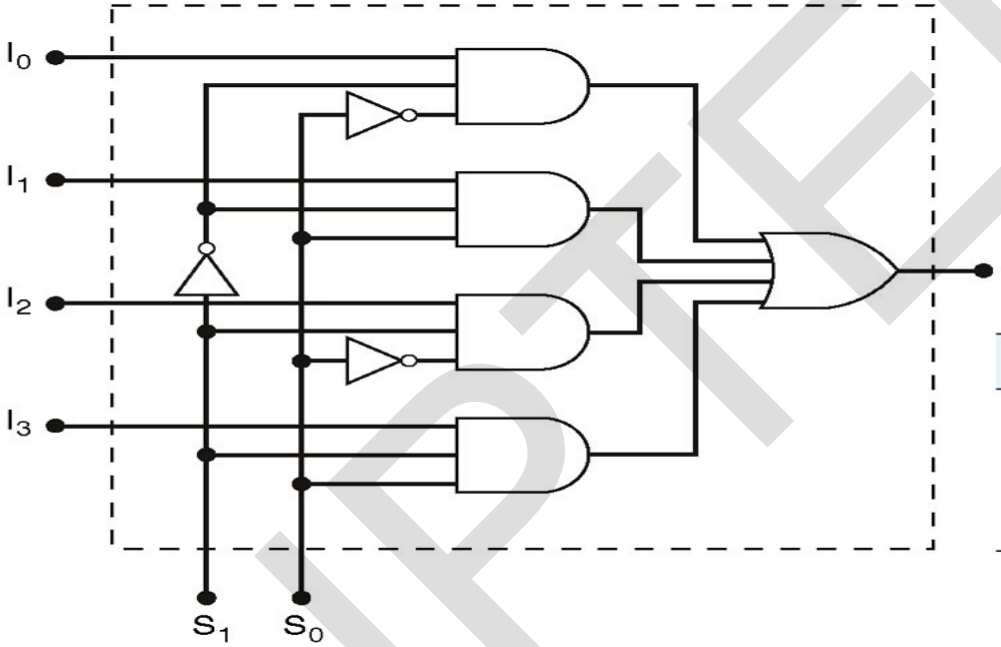
Two-input multiplexer



$$Z = I_0 \cdot \overline{S} + I_1 \cdot S$$

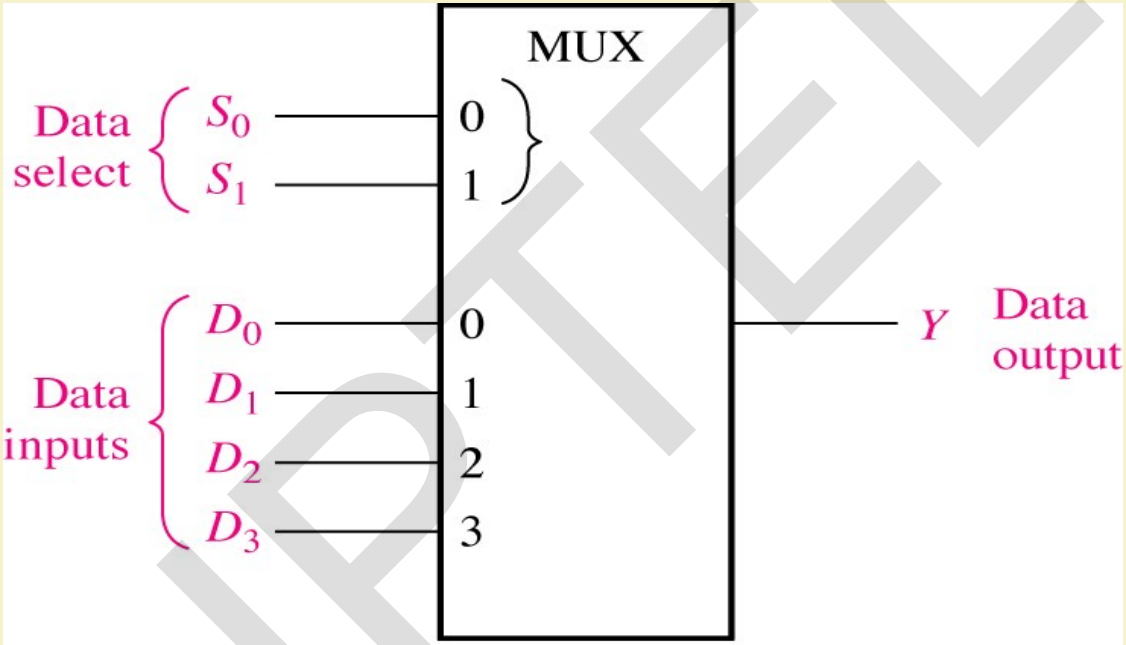
S	Output
0	$Z = I_0$
1	$Z = I_1$

Four-input multiplexer



S_1	S_0	Output
0	0	$Z = I_0$
0	1	$Z = I_1$
1	0	$Z = I_2$
1	1	$Z = I_3$

Logic symbol for a 1-of-4 data selector/multiplexer.



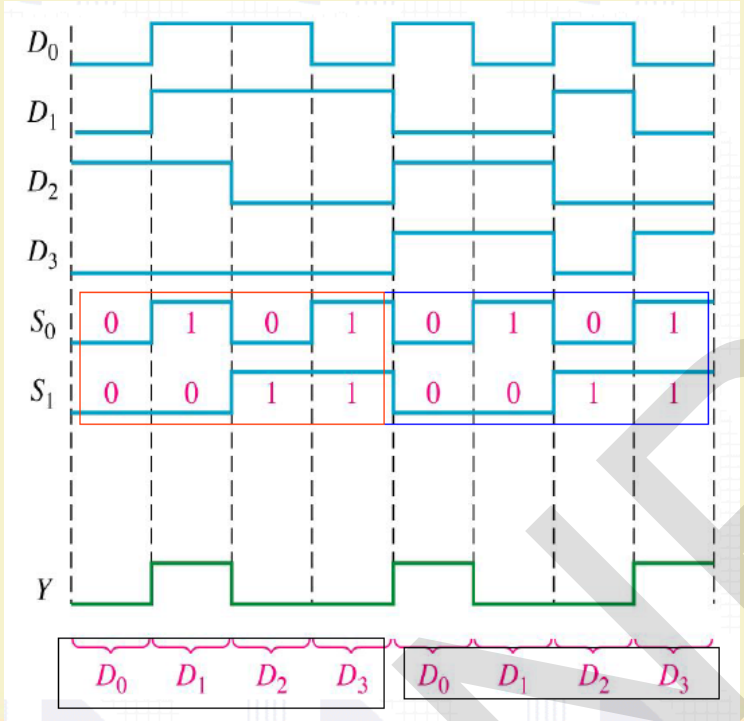
Logic symbol for a 1-of-4 data selector/multiplexer.

FUNCTION TABLE

INPUTS						STROBE $\overline{G}$	OUTPUT Y
SELECT		DATA					
B	A	C0	C1	C2	C3		
S1	S0	X	X	X	X	H	Z
[illegible]	[illegible]	L	X	X	X	L	L
L	L	H	X	X	X	L	H
L	H	X	L	X	X	L	L
L	H	X	H	X	X	L	H
H	L	X	X	L	X	L	L
H	L	X	X	H	X	L	H
H	H	X	X	X	L	L	L
H	H	X	X	X	H	L	H

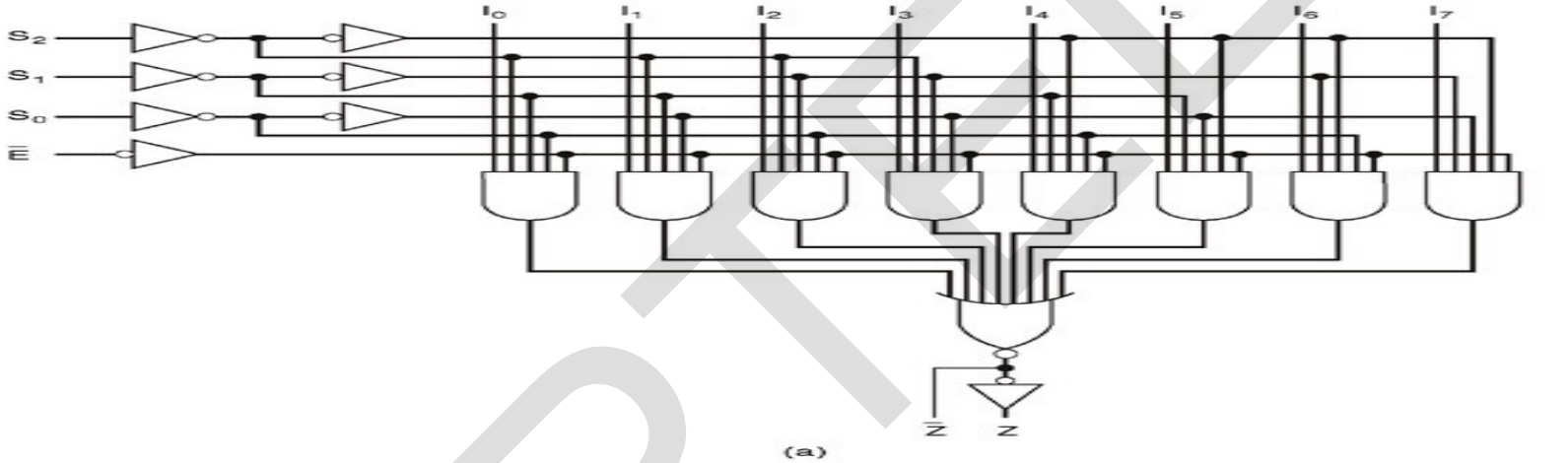
Select inputs A and B are common to both sections.

Output Waveforms in relation with the Data-Input and Data-Select waveforms - 4-input MUX



The binary state of the data-select inputs during each interval determines which data input is selected. Here the data-select inputs go through a repetitive binary sequence 00,01,10,11,00, and so on. The resulting output waveform is shown.

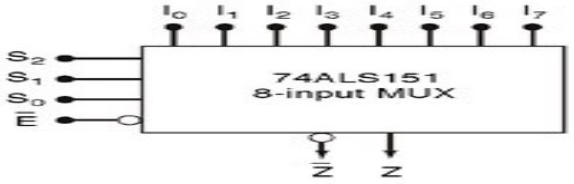
8-input multiplexer



(a)

Inputs				Outputs	
$\overline{E}$	S_2	S_1	S_0	$\overline{Z}$	Z
H	X	X	X	H	L
L	L	L	L	$\overline{I_0}$	I_0
L	L	L	H	$\overline{I_1}$	I_1
L	L	H	L	$\overline{I_2}$	I_2
L	L	H	H	$\overline{I_3}$	I_3
L	H	L	L	$\overline{I_4}$	I_4
L	H	L	H	$\overline{I_5}$	I_5
L	H	H	L	$\overline{I_6}$	I_6
L	H	H	H	$\overline{I_7}$	I_7

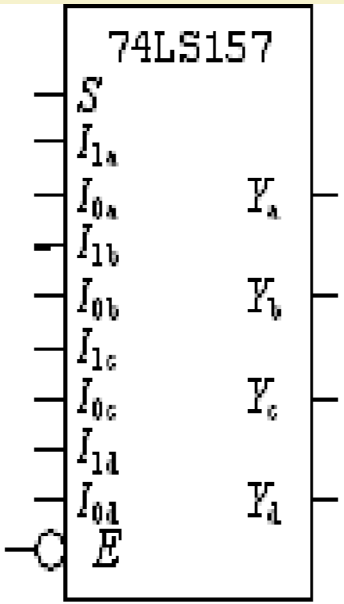
(b)



(c)

MSI Quad Two–input Multiplexer

- The **74LS157** contain of quad two–input multiplexers, $I_{0a}\, I_{0b}\, I_{0c}\, I_{0d}$ and $I_{1a}\, I_{1b}\, I_{1c}\, I_{1d}$.
- The logic symbol and truth table is shown in Figure.
- Notice that each of the four multiplexer shares a common data select line and a common *Enable.*
- Each multiplexer has only one data select input because there are only two groups of inputs to be selected.



Inputs		Outputs			
$\overline{E}$	S	Y_a	Y_b	Y_c	Y_d
H	X	L	L	L	L
L	L	I_{0a}	I_{0b}	I_{0c}	I_{0d}
L	H	I_{1a}	I_{1b}	I_{1c}	I_{1d}

Logic symbol and truth table for 74LS157 multiplexer.

MSI Quad Two–input Multiplexer

- $\overline{E}$ Input is *LOW* – allows the selected input data to pass through to the output.

- $\overline{E}$ Input is *HIGH* – will disable the multiplexers, all of the outputs will be *LOW*.

- When $\overline{E}$ = 0 and S = 1, the Y outputs will follow the set of I_1 inputs, that is $Y_a = I_{1a}$, $Y_b = I_{1b}$, $Y_c = I_{1c,}$ and $Y_d = I_{1d}$.

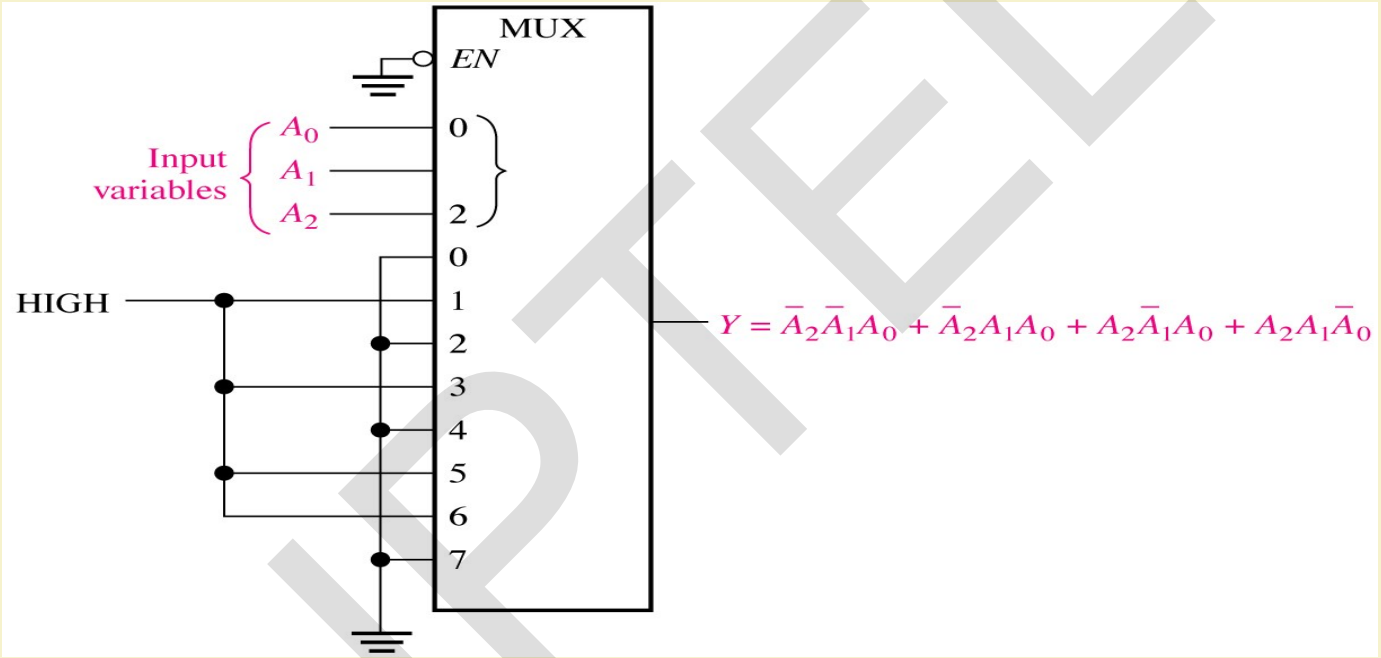
LOGIC FUNCTION GENERATION USING MUX

Exercise 1:

Implement the logic circuit function specified in the table given below by using 74LS151 8-input data selector/multiplexer.

Input			Output	
A2	A1	A0	Y	
0	0	0	0	0
0	0	1	1	1
0	1	0	0	2
0	1	1	1	3
1	0	0	0	4
1	0	1	1	5
1	1	0	1	6
1	1	1	0	7

Solution :



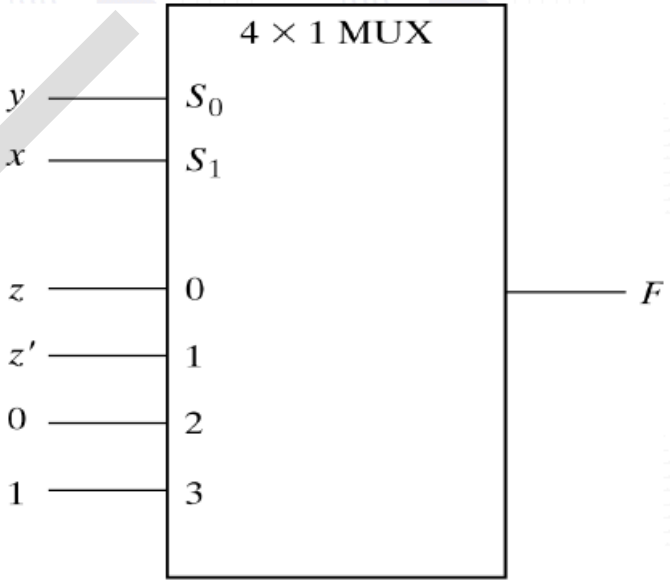
LOGIC FUNCTION GENERATION USING MUX-Method

- An efficient method for implementing a Boolean function of n variables with a MUX that has n-1 selection inputs and 2^{n-1} data inputs is given below:
  - List the Boolean function in a truth table
  - Apply the first n-1 variables in the table to the selection inputs of the MUX.
  - For each combination of the selection variables, evaluate the output as a function of the last variable. This function can be 0,1, the variable, or the complement of the variable. Apply these values to the data inputs in the proper order.

LOGIC FUNCTION GENERATION USING MUX-Example 2

Implement the Boolean function F=x'y'z +x'yz' + xyz' + xyz using a suitable MUX

x	y	z	F		
0	0	0	0	$F = z$	0
0	0	1	1		
0	1	0	1	$F = z'$	1
0	1	1	0		
1	0	0	0	$F = 0$	2
1	0	1	0		
1	1	0	1	$F = 1$	3
1	1	1	1		



LOGIC FUNCTION GENERATION USING MUX-Example 2

- The two variables x and y are applied to the selection lines in that order; x is connected to the S_1 input and y to the S_0 input.
- The values for the data input lines are determined from the truth table of the function
  - For ex., when xy=00, output F is equal to z because F=0 when z=0 and F=1 when z=1. This requires that variable z is applied to the data input 0

LOGIC FUNCTION GENERATION USING MUX-Example 3

Implement the Boolean function F=A'B'C'D+A'B'CD+A'BC'D'
+AB'CD+ABC'D'+ABC'D+ABCD'+ABCD using a suitable MUX

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