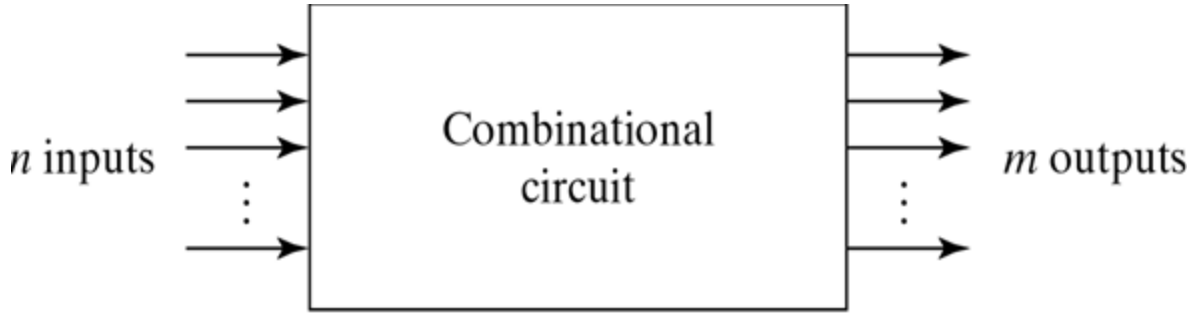
Combinational Logic Design

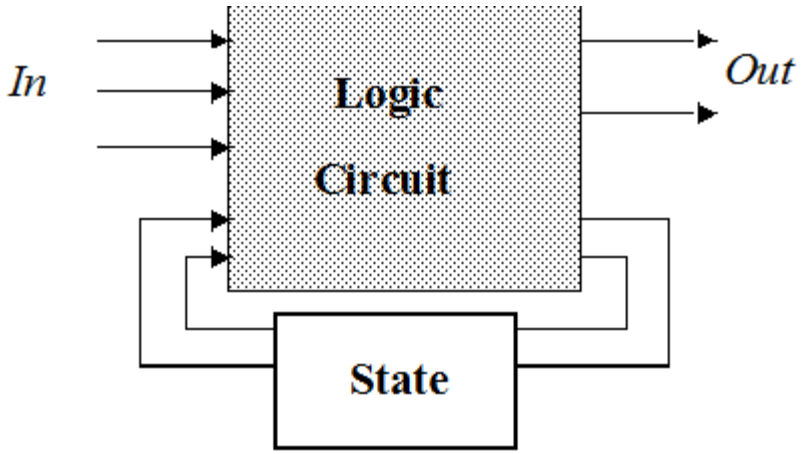
Combinational Circuits

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



  - The outputs depend only on the current input values
  - It uses only logic gates

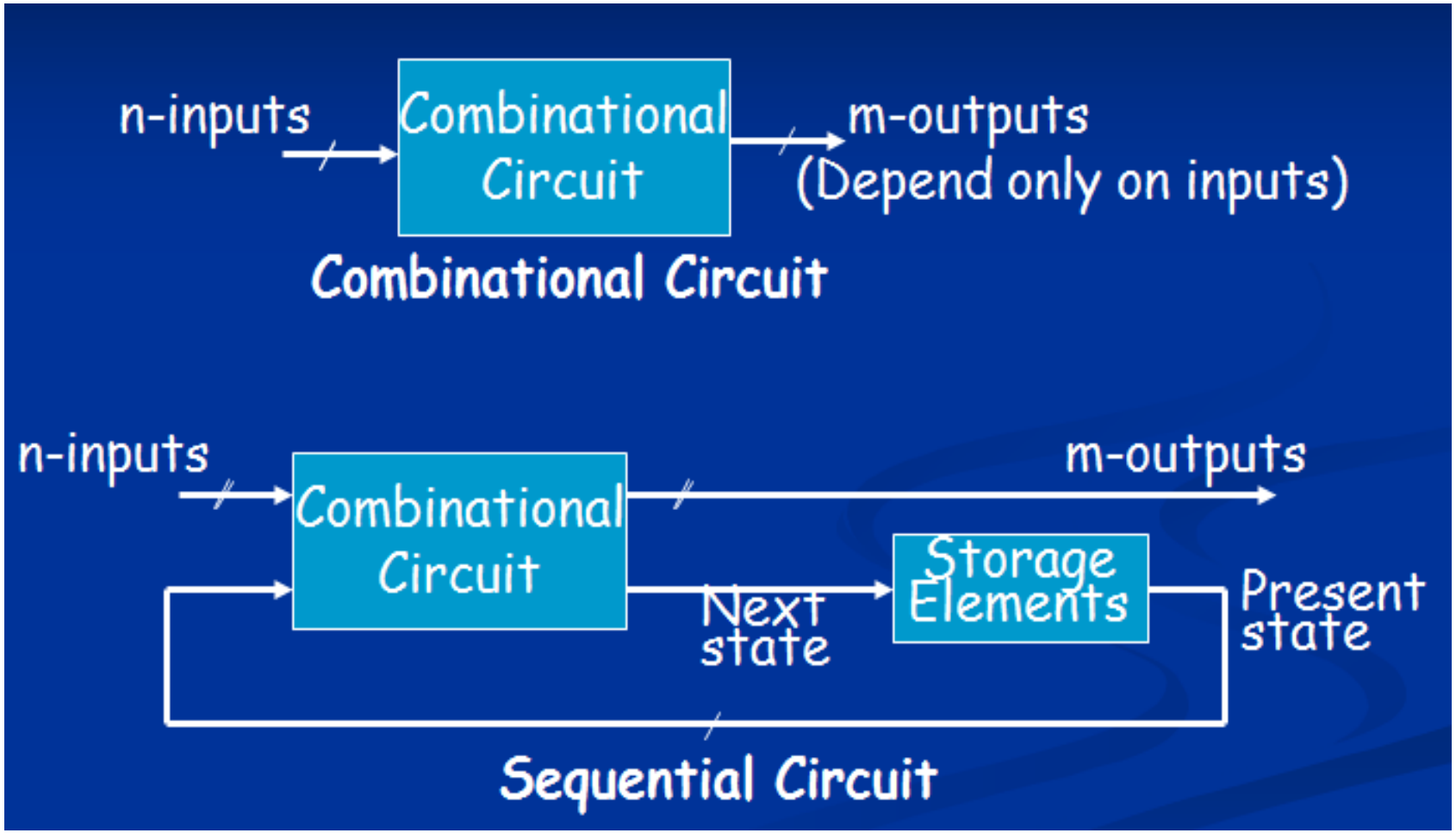
- Sequential circuit
  - The outputs depend on the current and past input values
  - It uses logic gates and storage elements



Output = *f*(*In, Previous In*)

Combinational vs. Sequential Circuits

- Combinational circuits are memory-less. Thus, the output value depends ONLY on the current input values.
- Sequential circuits consist of combinational logic as well as memory elements (used to store certain circuit states). Outputs depend on BOTH current input values and previous input values (kept in the storage elements).


n-inputs
Combinational Circuit
m-outputs
(Depend only on inputs)
Combinational Circuit
n-inputs
Combinational Circuit
m-outputs
Storage Elements
Next state
Present state
Sequential Circuit

Design Procedure of Combinational Circuits

1. Determine number of required inputs and outputs.
2. Assign names to input and output variables
3. Derive truth table
4. Obtain simplified Boolean functions
5. Draw logic diagram and verify correctness

Notes

- If there are *n* input variables, there are 2^n input combinations
- For each input combination, there is one output value
- Truth tables are used to list all possible combinations of inputs and corresponding output values

Examples of Combinational Circuits

- Adder/Subtractor
- Code converters
- Comparators
- Decoders
- Encoders
- Multiplexers
- Demultiplexer

Adder

- Are of 2 types:
  - Half adder
  - Full adder

Half Adder

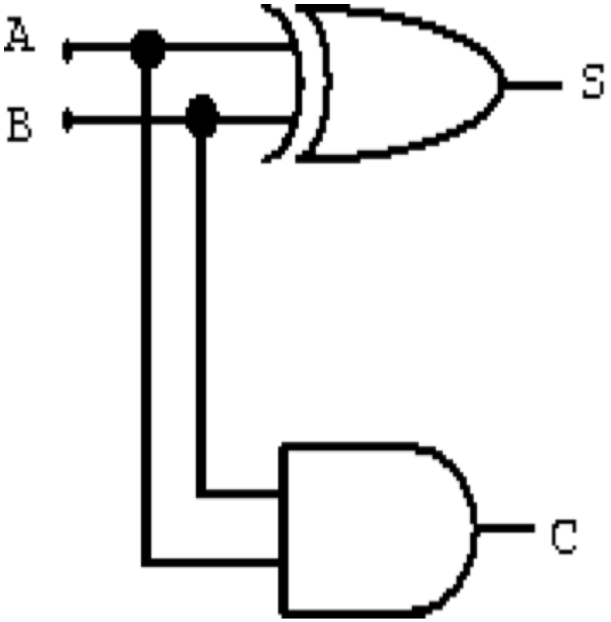
A combinational circuit that adds 2 input bits to generate a Sum bit and a Carry bit

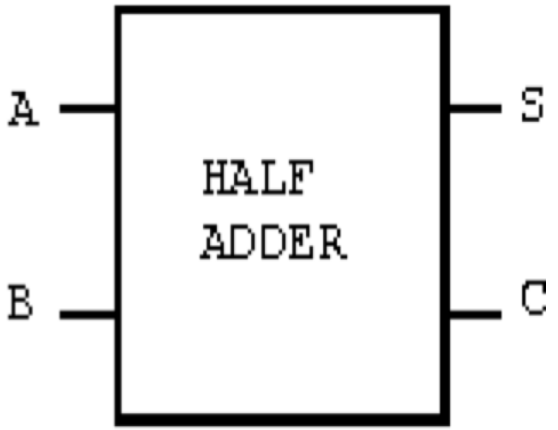
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$Sum = \overline{A}B + A\overline{B} = A \oplus B$$

$$Carry = AB$$

Half Adder





Full Adder

- A combinational circuit that adds 3 input bits to generate a Sum bit and a Carry bit

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Sum

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

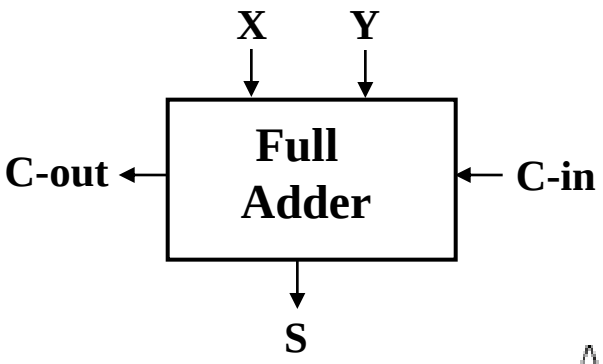
S = X'Y'Z + X'YZ' + XY'Z' +XYZ

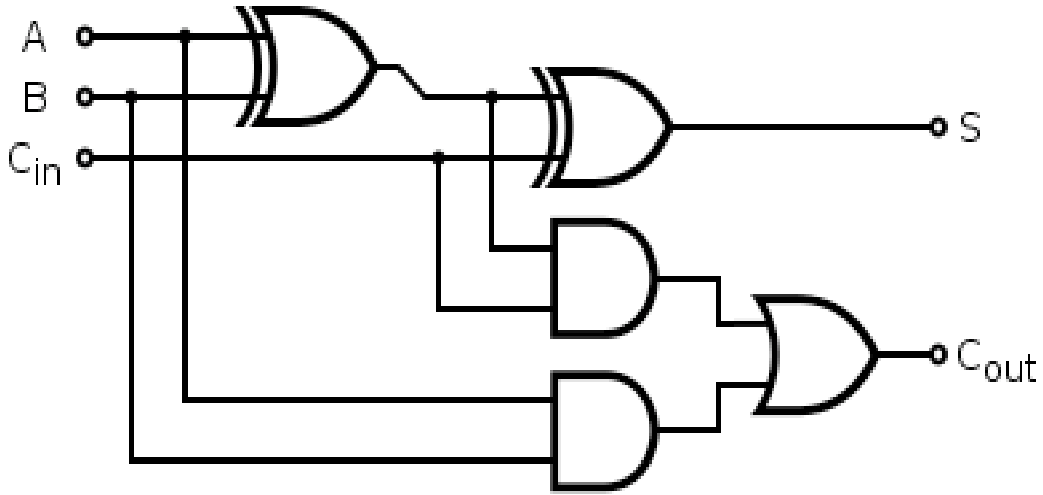
= X ⊕ Y ⊕ Z

Carry

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

C = XY + YZ + XZ


X
Y
Full
Adder
C-out
C-in
S


A
B
Cin
S
Cout

Full Adder = 2 Half Adders

Manipulating the Equations:

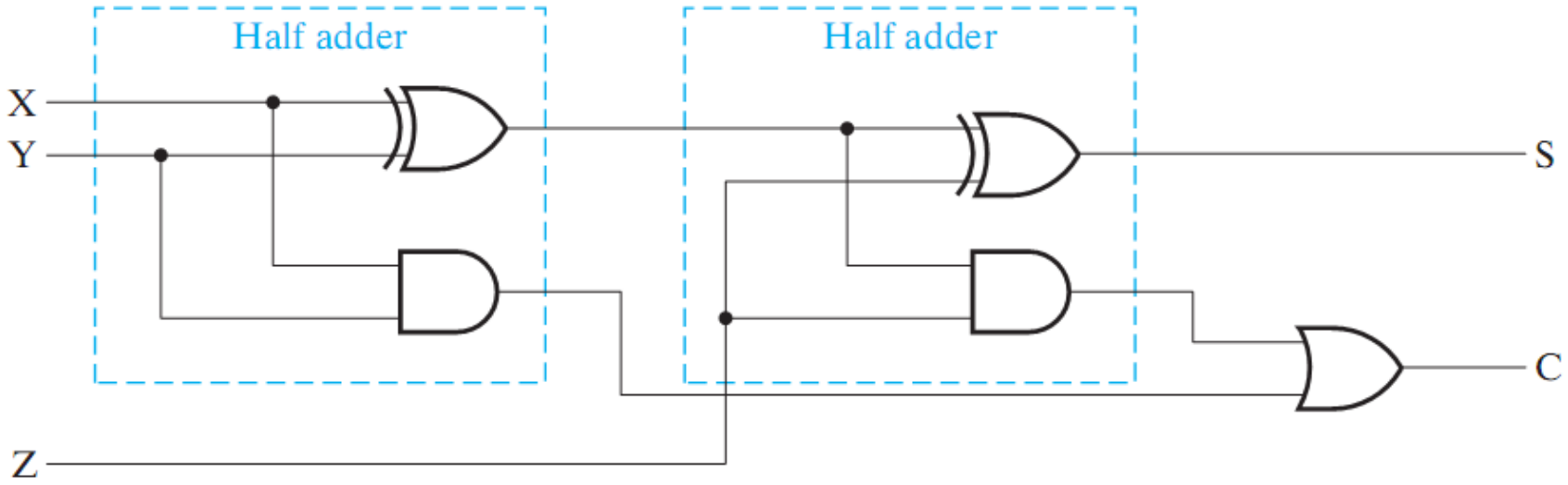
$$S = X \oplus Y \oplus Z$$

$$C = XY + XZ + YZ$$

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

$$= XY(1 + Z) + Z(XY' + X'Y)$$

$$= XY + Z(X \oplus Y)$$


Half adder
Half adder
X
Y
Z
S
C

Half Subtractor

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

Half Subtractor

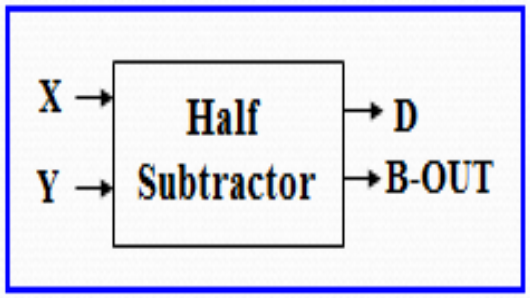
Half Subtractor Truth Table:

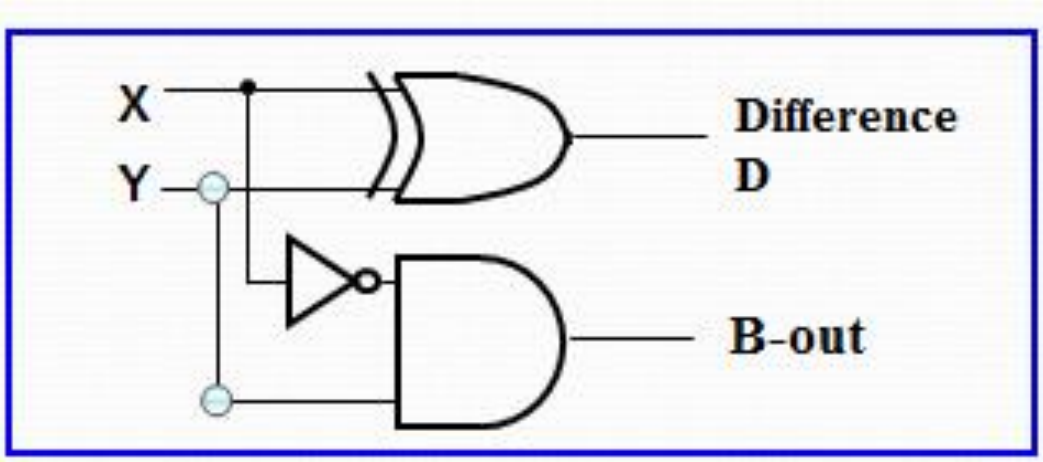
Inputs		Outputs	
X	**Y**	**D**	**B-out**
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

$$D(X,Y) = \Sigma(1,2)$$
$$D = X'Y + XY'$$
$$D = X \oplus Y$$

$$B\text{-}out(X,Y) = \Sigma(1)$$
$$B\text{-}out = X'Y$$

Half Subtractor



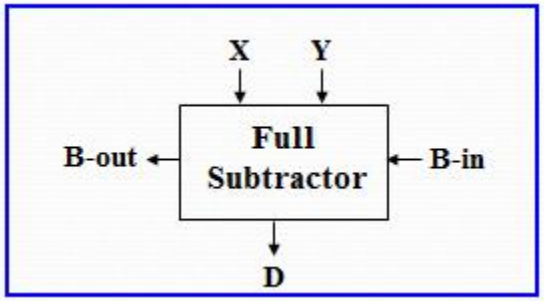


Full Subtractor

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

Full Subtractor

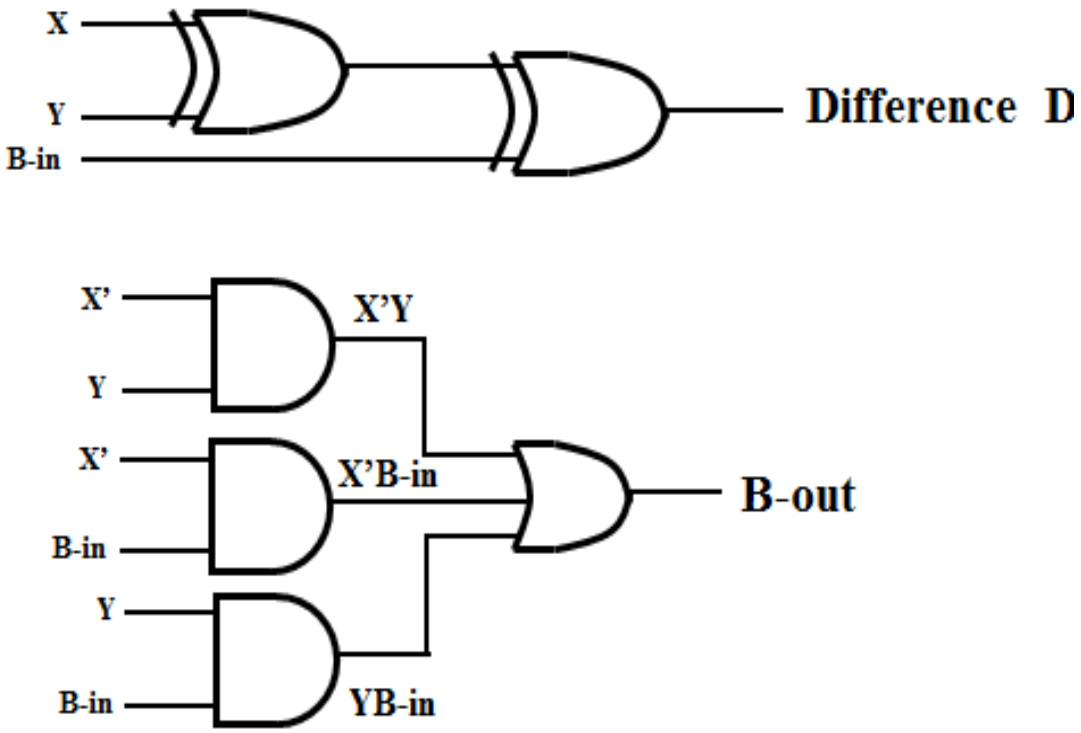
X	Y	B-in	D	B-out
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



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

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

Full Subtractor



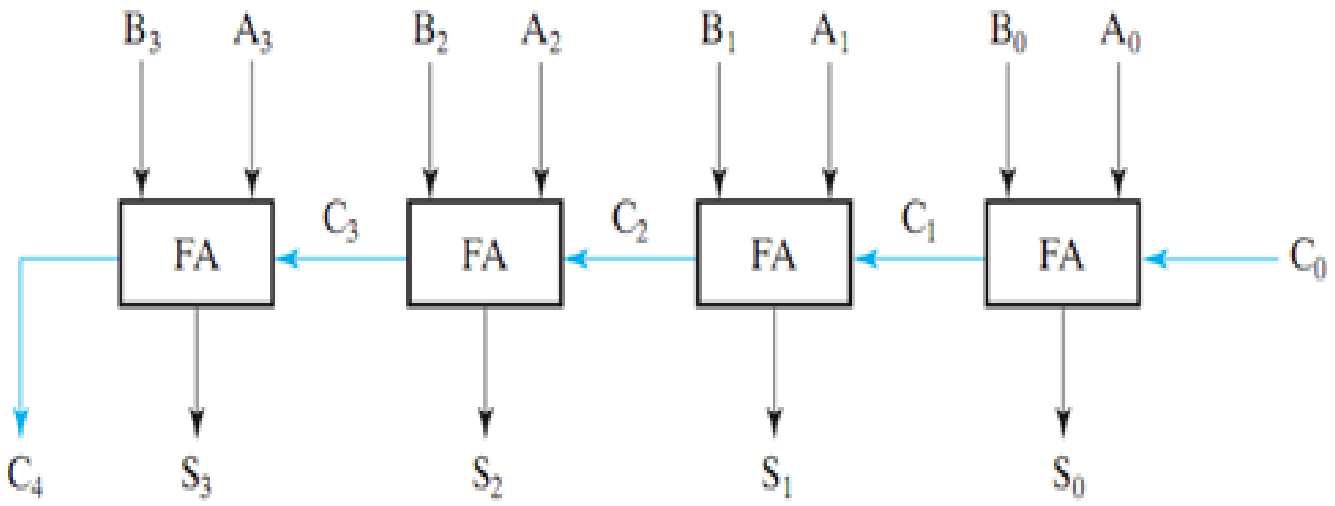
Bigger Adders

- How to build an adder for n-bit numbers?
  - Example: 4-Bit Adder
    - Inputs ? 9 inputs
    - Outputs ? 5 outputs
    - What is the size of the truth table? 512 rows!
    - How many functions to optimize? 5 functions

Binary Parallel Adder

- To add n-bit numbers:
  - Use n Full-Adders in parallel
  - The carries propagates as in addition by hand

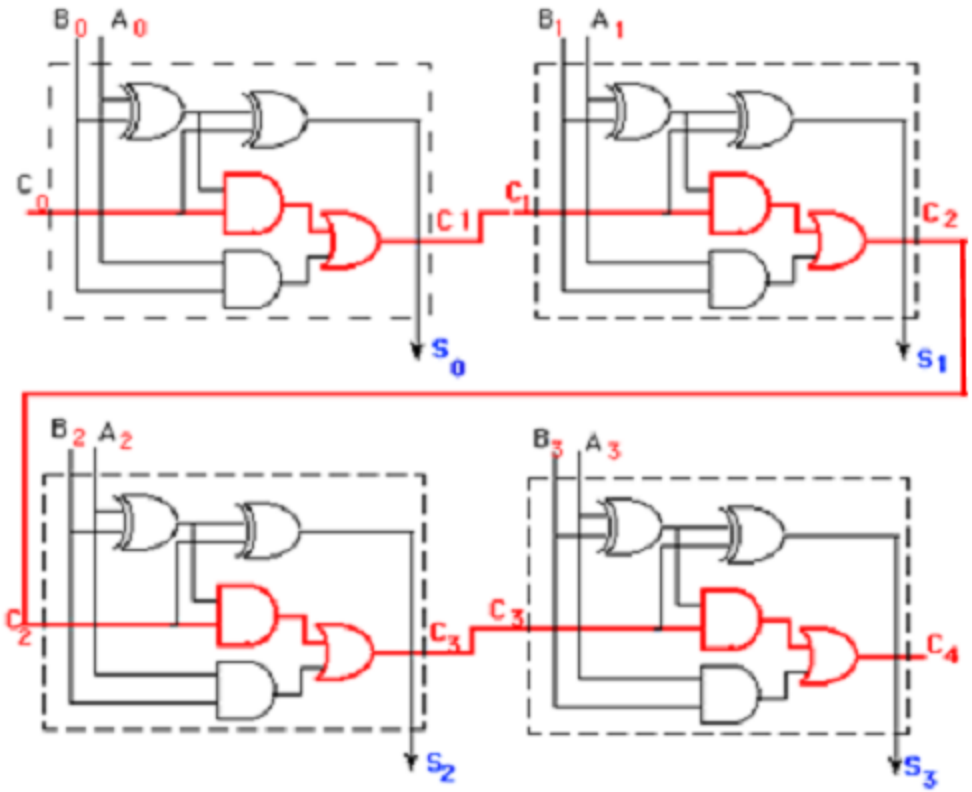
```
  1 0 0 0
  0 1 0 1
  0 1 1 0
-----------
  1 0 1 1
```



This adder is called ***ripple carry adder***

Ripple Adder Delay

- Assume gate delay = T
- 8 T to compute the last carry
- Total delay = 8 + 1 = 9T
- 1 delay form first half adder
- Delay = (2n+1)T



How to improve?

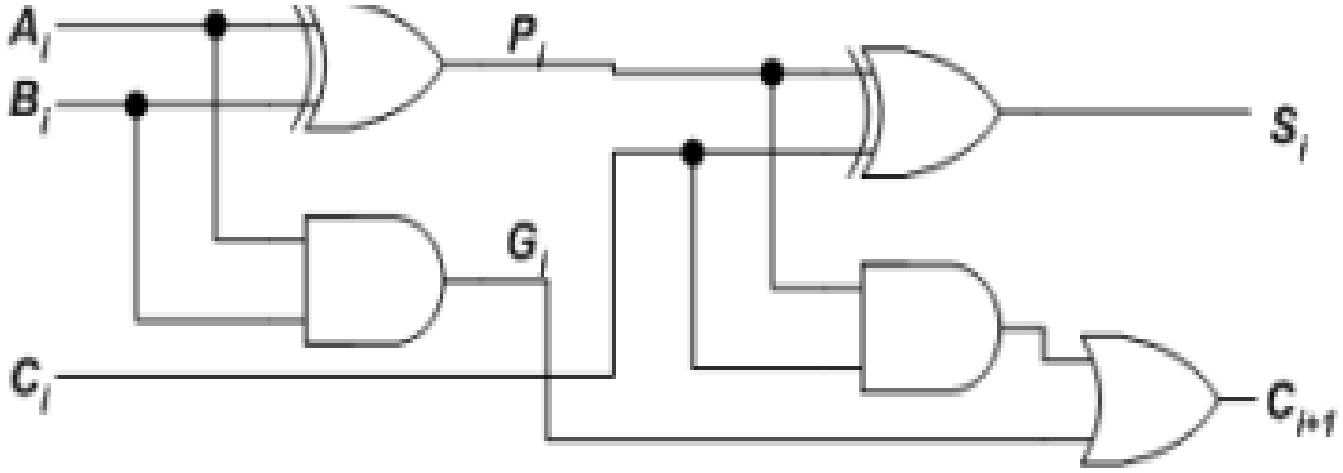
Carry Look Ahead Adder

- One problem with the addition of binary numbers is the length of time to propagate the ripple carry from the least significant bit to the most significant bit.
- Objective of Carry Look Ahead Adder is to generate all incoming carries in parallel

Carry Look Ahead Adder

- Based on two addition functions of FA, called Carry Generate (G_i) and Carry propagate (P_i)

- Defining the equations for the Full Adder in term of the P_i and G_i:



$$P_i = A_i \oplus B_i \qquad G_i = A_i B_i$$

$$S_i = P_i \oplus C_i \qquad C_{i+1} = G_i + P_i C_i$$

G_i is known as the ***carry Generate*** signal since a carry (C_{i+1}) is generated whenever G_i =1, regardless of the input carry (C_i).

P_i is known as the ***carry propagate*** signal since whenever P_i =1, the input carry is propagated to the output carry, i.e., C_{i+1}. = C_i (note that whenever P_i =1, G_i =0).

- We can write the Boolean functions of carry-out of each stage and substitute for each C_i with the value from previous equation.

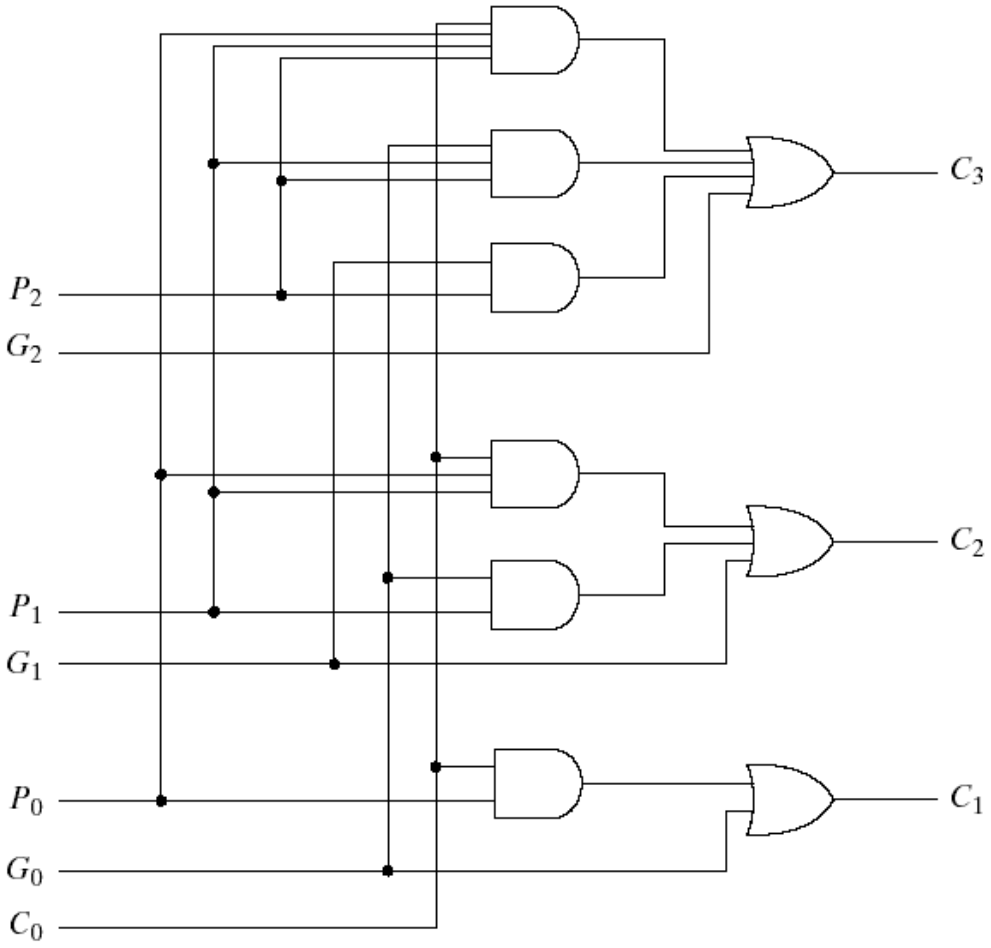
$$C_1 = G_0 + P_0 C_0$$

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

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

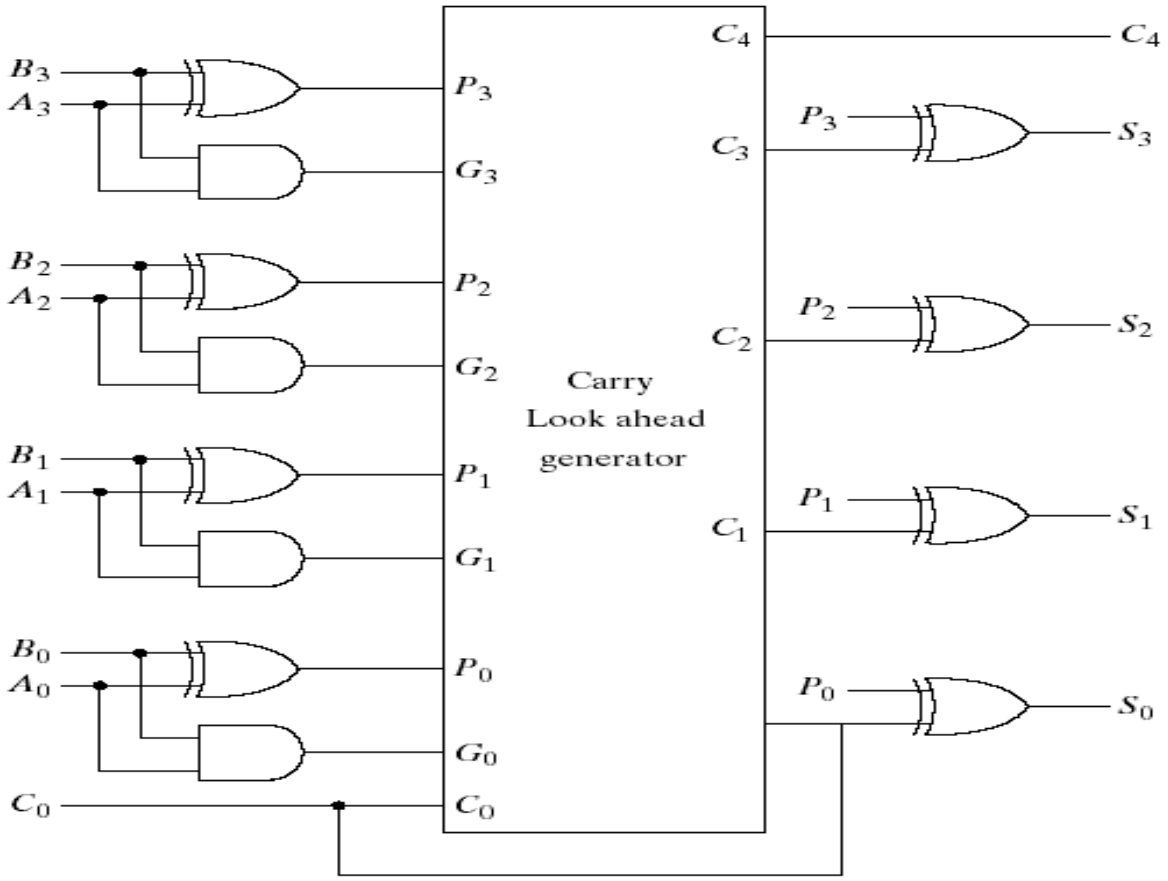
$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1$$
$$+ P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

CLA generator



4-bit carry look-ahead adder

- The 4-bit carry look-ahead (CLA) adder consists of 3 levels of logic:



- First level: Generates all the P & G signals. Four sets of P & G logic (each consists of an XOR gate and an AND gate). Output signals of this level (P's & G's) will be valid after 1T.
- Second level: The Carry Look-Ahead (CLA) logic block which consists of four 2-level implementation logic circuits. It generates the carry signals (C1, C2, C3, and C4) as defined by the above expressions. Output signals of this level (C1, C2, C3, and C4) will be valid after 3T.

- Third level: Four XOR gates which generate the sum signals (Si) (Si = Pi $\oplus$ Ci). Output signals of this level (S0, S1, S2, and S3) will be valid after 4T.
- The 4 Sum signals (S0, S1, S2, S3) will all be valid after a total delay of 4T compared to a delay of (2n+1)T for Ripple Carry adders.

- The disadvantage of the CLA adders is that the carry expressions become quite complex for more than 4 bits.

BCD Adder

- BCD digits are valid for decimal numbers 0-9
- Addition of two BCD numbers will generate an output, that may be greater than 1001 (9).
- In such cases, the BCD number 0110 is added to the result as a correction step

- When adding two BCD numbers, the maximum result that can be obtained is:

  9 + 9 = 18

  – If we include a carry in bit, then the maximum result that can be obtained is: 19 (10011)
  – Both numbers 18 and 19 are invalid BCD digits. Therefore, a 6 needs to be added to bring them to correct BCD format.

BCD Addition

- **Procedure:**
  - **Step 1:** Add the two BCD numbers, using the rules for binary addition
  - **Step 2:** If a 4-bit sum is equal to or less than 9, it is a valid BCD number.
  - **Step 3:** If a 4-bit sum is greater than 9, or if a carry out of the 4-bit group is generated, it is an invalid result. Add 6 (0110) to the 4-bit sum in order to skip the six invalid states and return the code to 8421. If a carry results when 6 is added, simply add the carry to the next 4-bit group.

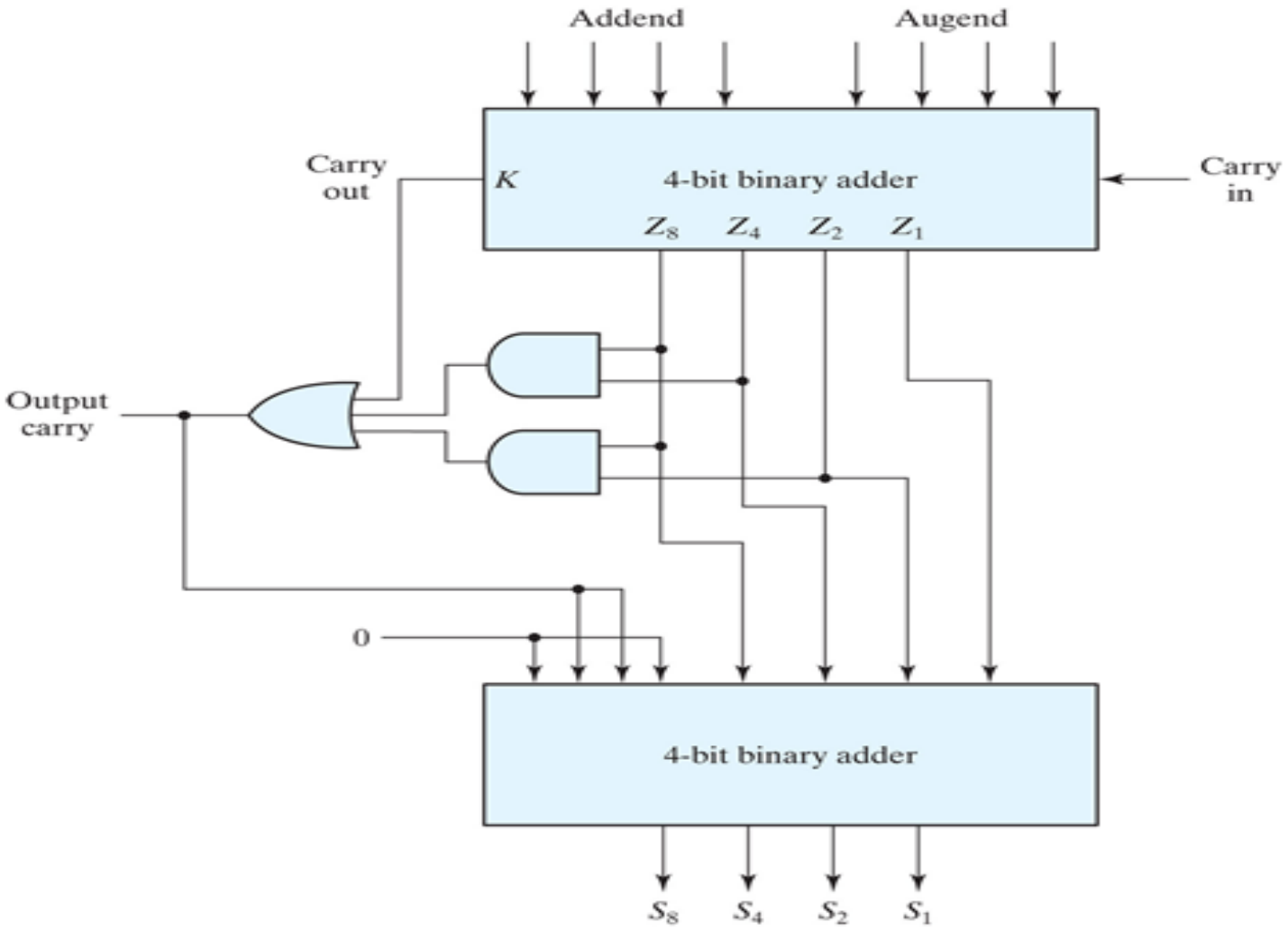
Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0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

- The condition for correction is

$$C=K+Z_8Z_4+Z_8Z_2$$

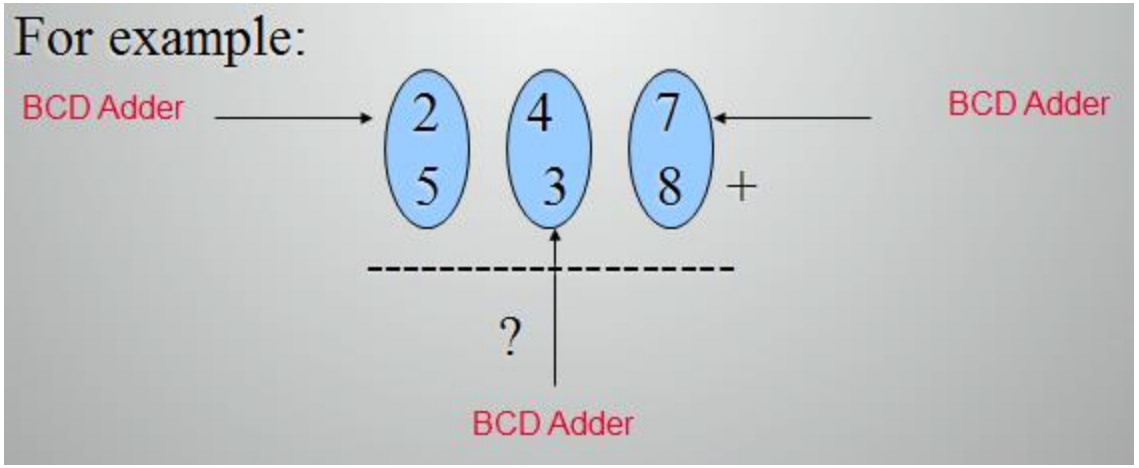
- When C=1, it is necessary to add 0110 to the binary sum and provide the output carry for the next stage
- When C=0, nothing is added to the binary sum

Block diagram of a BCD adder

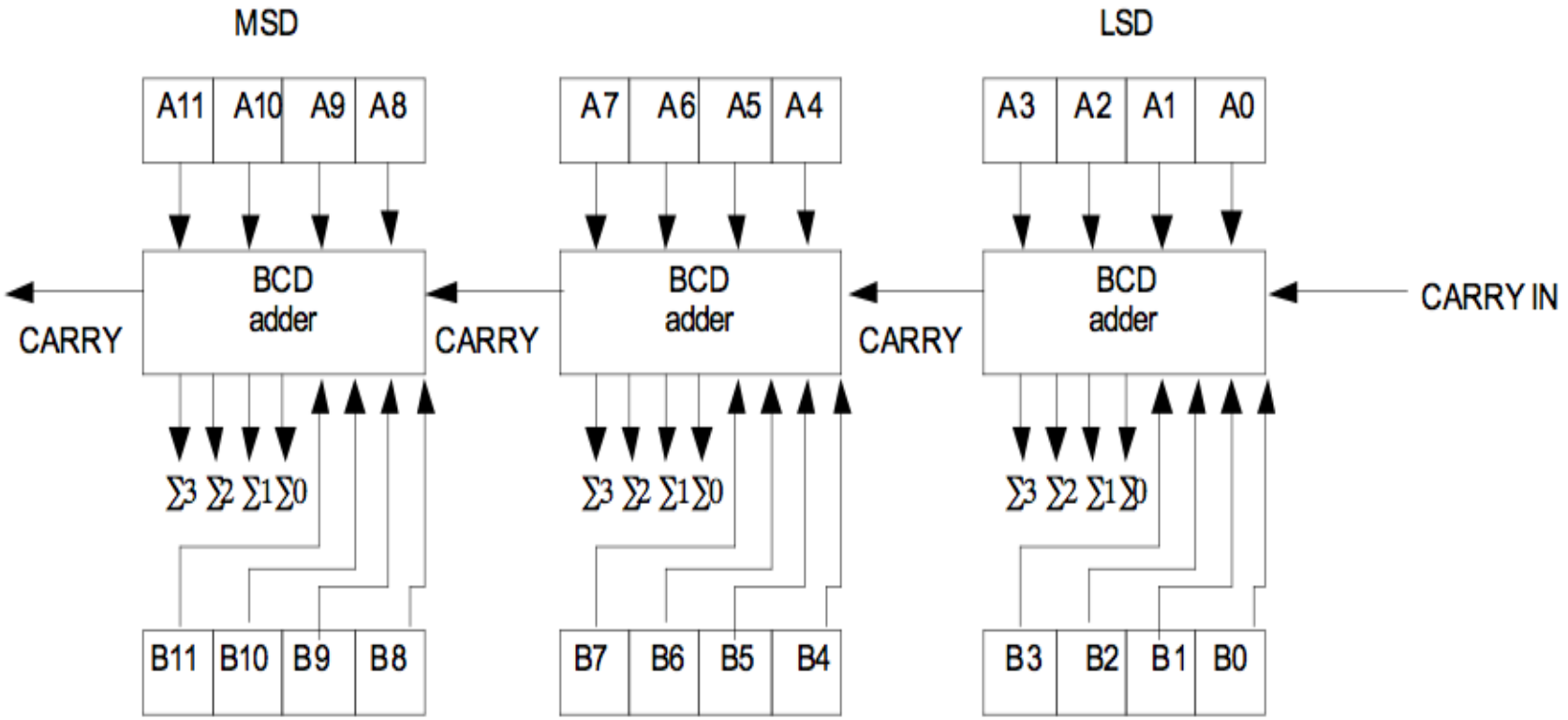


Cascading BCD Adders

- The previous circuit is used for adding two decimal digits only. That is, “ 7 + 6 = 13”.
- For adding numbers with several digits, a separate BCD adder for each digit position must be used.



Cascading BCD Adders



Example

- Determine the inputs and the outputs when the above circuit is used to add 538 to 247. Assume a CARRY IN = 0
- Solution:
  - Represent the decimal numbers in BCD

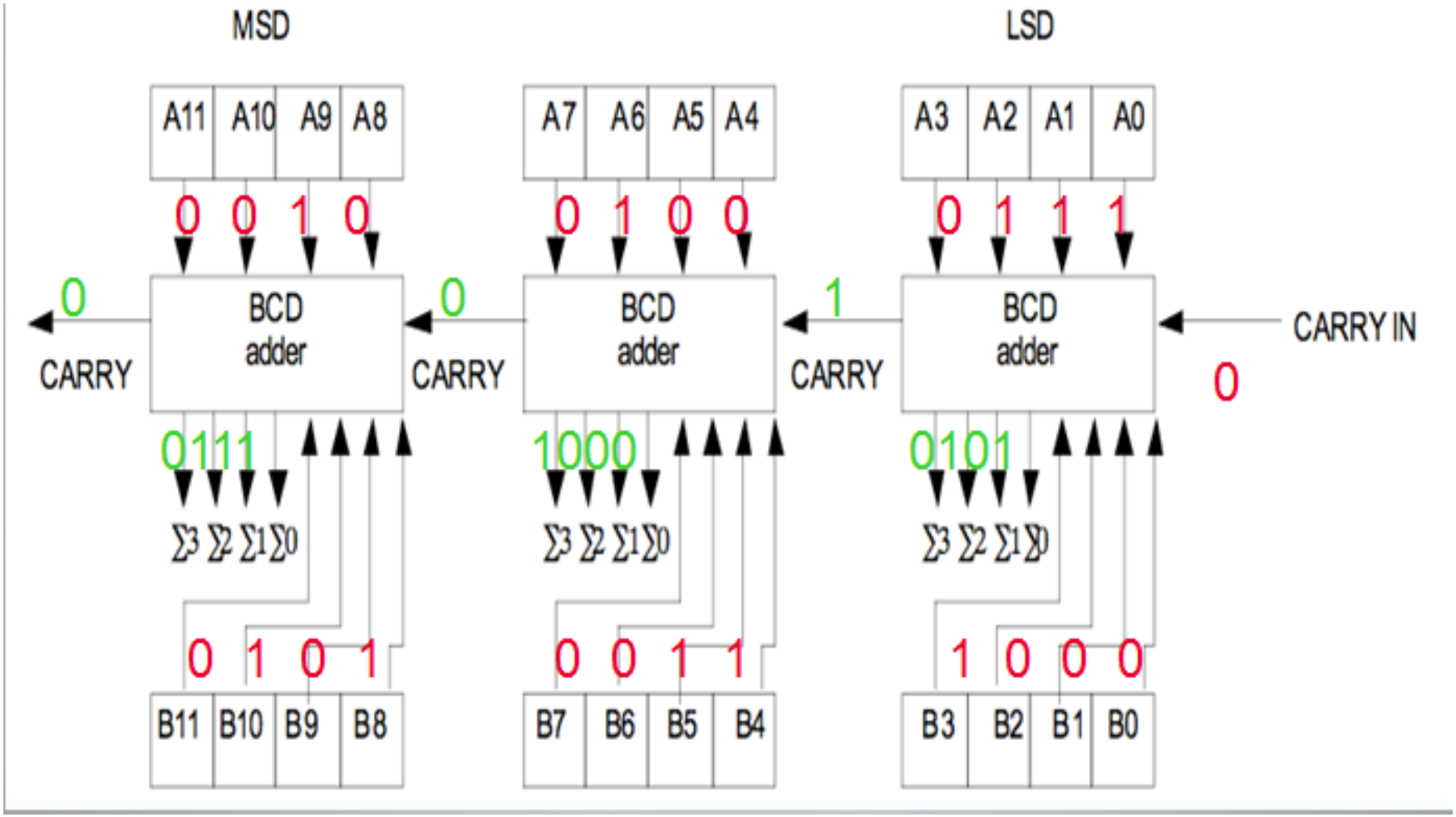
    247 = 0010 0100 0111

    538 = 0101 0011 1000

    Put these numbers in registers [A] and [B]

    [A] = 0010 0100 0111

    [B] = 0101 0011 1000


MSD
LSD
A11 A10 A9 A8
A7 A6 A5 A4
A3 A2 A1 A0
0 0 1 0
0 1 0 0
0 1 1 1
0
CARRY
BCD adder
0
CARRY
BCD adder
1
CARRY
BCD adder
CARRY IN
0
0111
1000
0101
∑3 ∑2 ∑1 ∑0
∑3 ∑2 ∑1 ∑0
∑3 ∑2 ∑1 ∑0
0 1 0 1
0 0 1 1
1 0 0 0
B11 B10 B9 B8
B7 B6 B5 B4
B3 B2 B1 B0

Magnitude Comparator

- Compare the magnitude of two binary numbers for the purpose of establishing whether one is greater than, equal to, or less than the other.

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

$$A = A_3A_2A_1A_0 \; ; \; B = B_3B_2B_1B_0$$

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

$$(A = B) = x_3x_2x_1x_0$$

- 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.

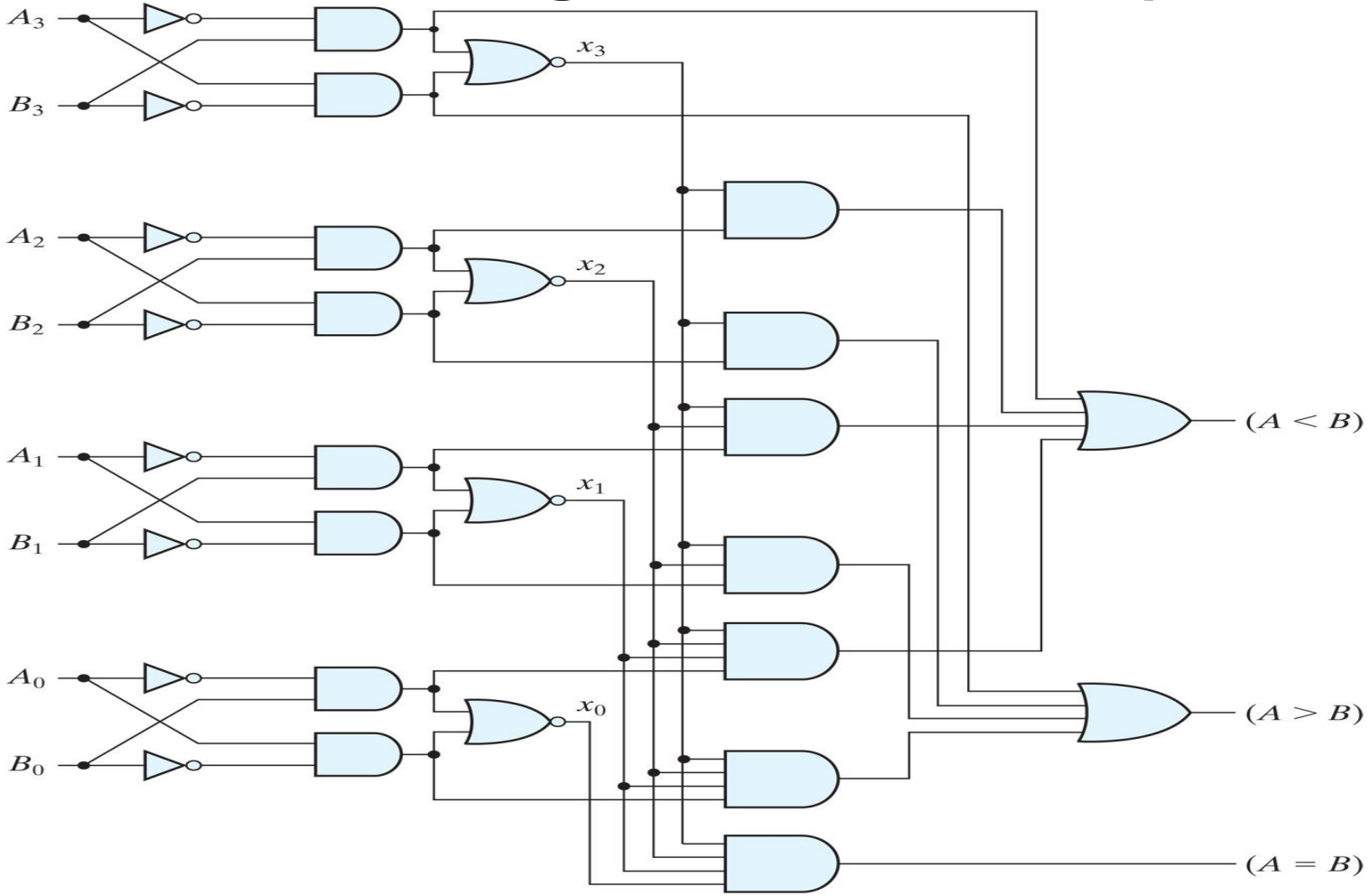
$$(A>B)= A_3B'_3+x_3A_2B'_2+x_3x_2A_1B'_1+x_3x_2x_1A_0B'_0$$

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

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

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

Four-bit magnitude comparator



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
- Examples: BCD to binary, BCD to 7–segment, binary to BCD, BCD to XS3, binary to Gray code, and Gray code to binary.

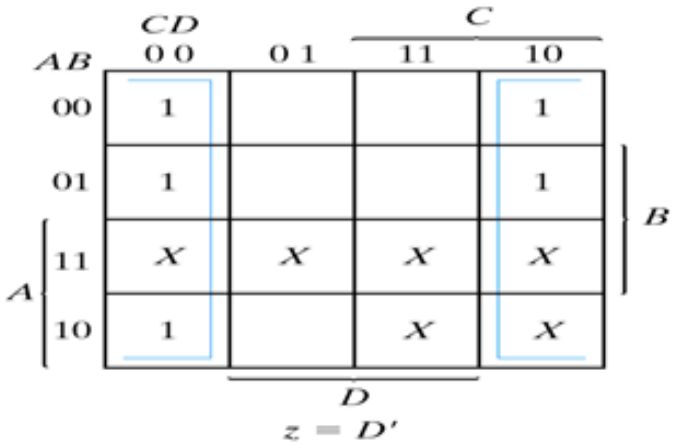
BCD to Excess 3 code converter

- 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.

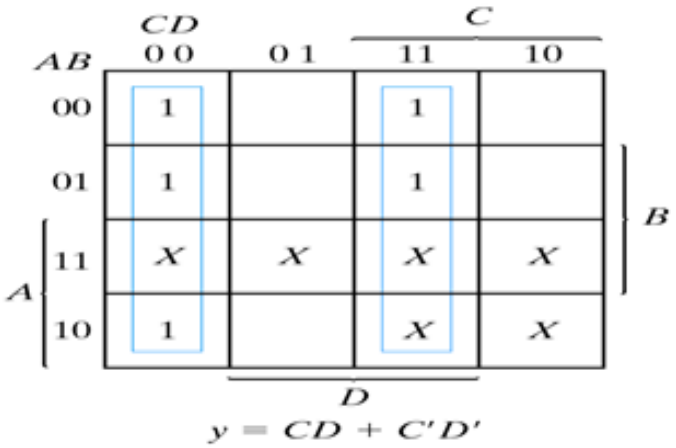
BCD to Excess 3 code converter

Input (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**

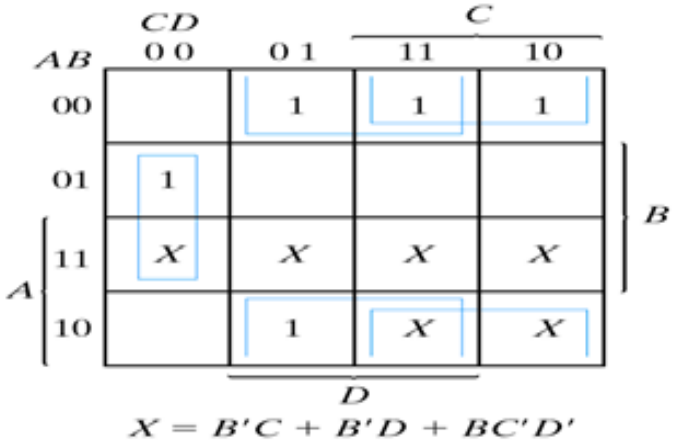
K-maps for simplification and simplified Boolean expressions



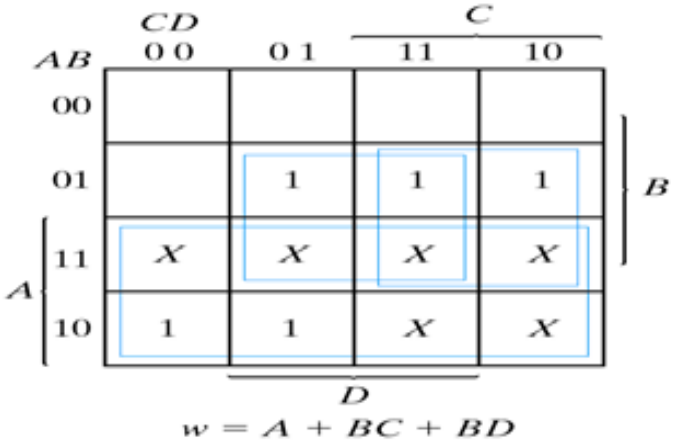
$z = D'$



$y = CD + C'D'$



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



$w = A + BC + BD$

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

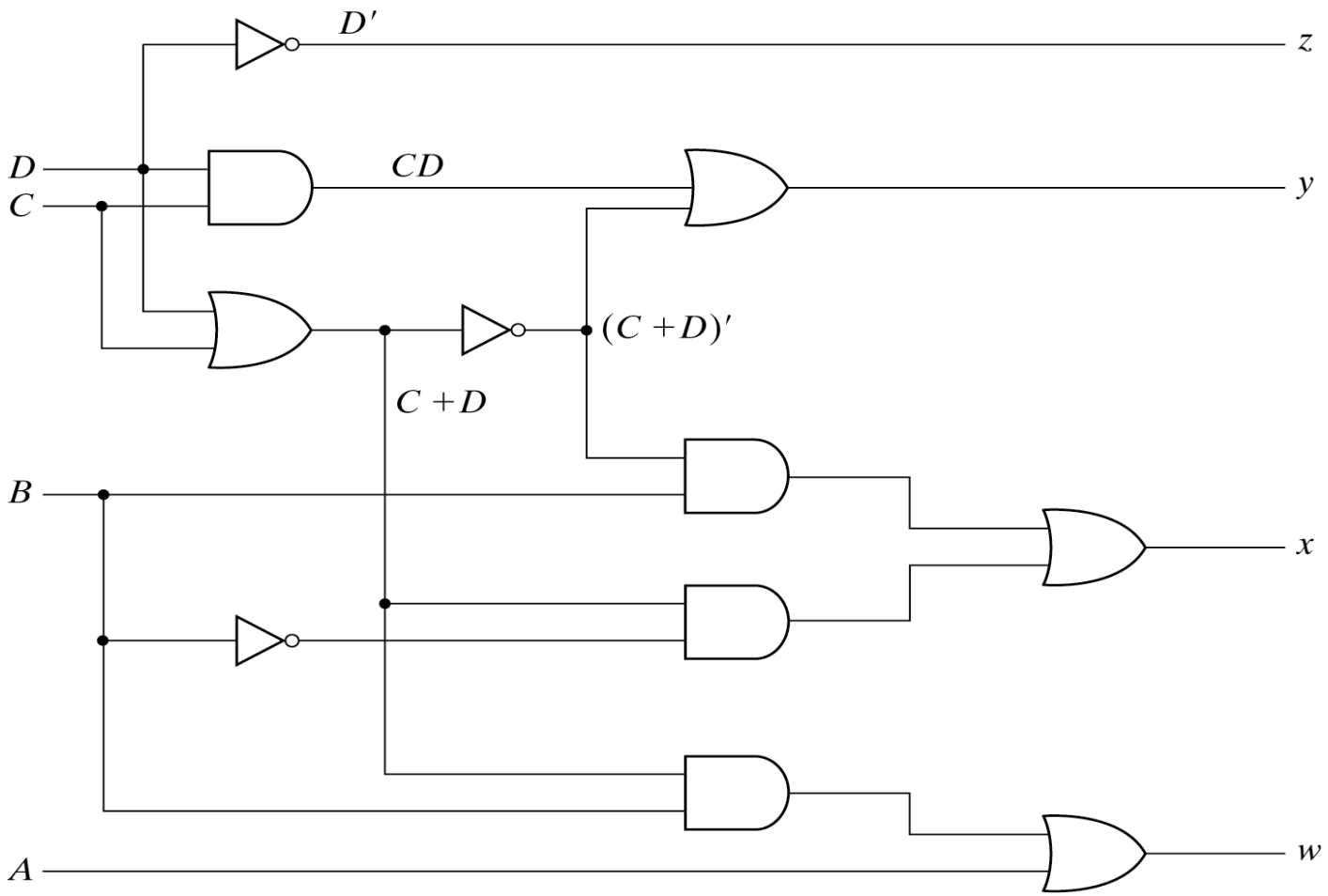
  z=D’

  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)

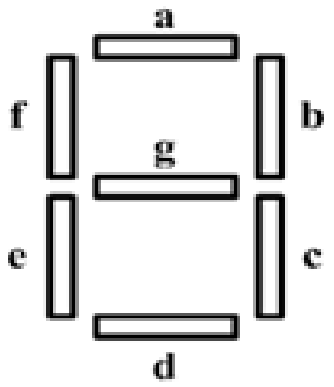
Logic Diagram



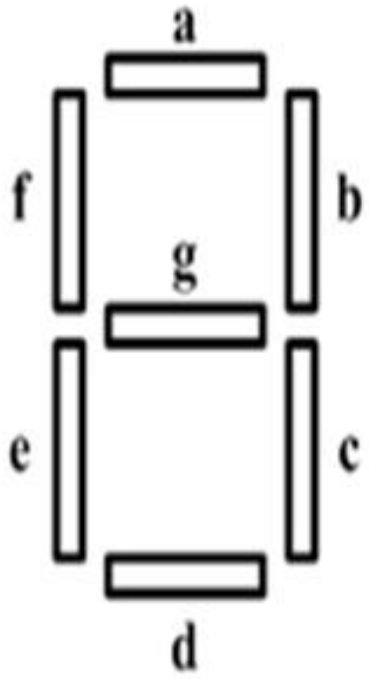
BCD-to-Seven-Segment Decoder

- 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.

- 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.



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

Binary to Gray Code Converter

Decimal	Binary				Gray			
	A	B	C	D	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

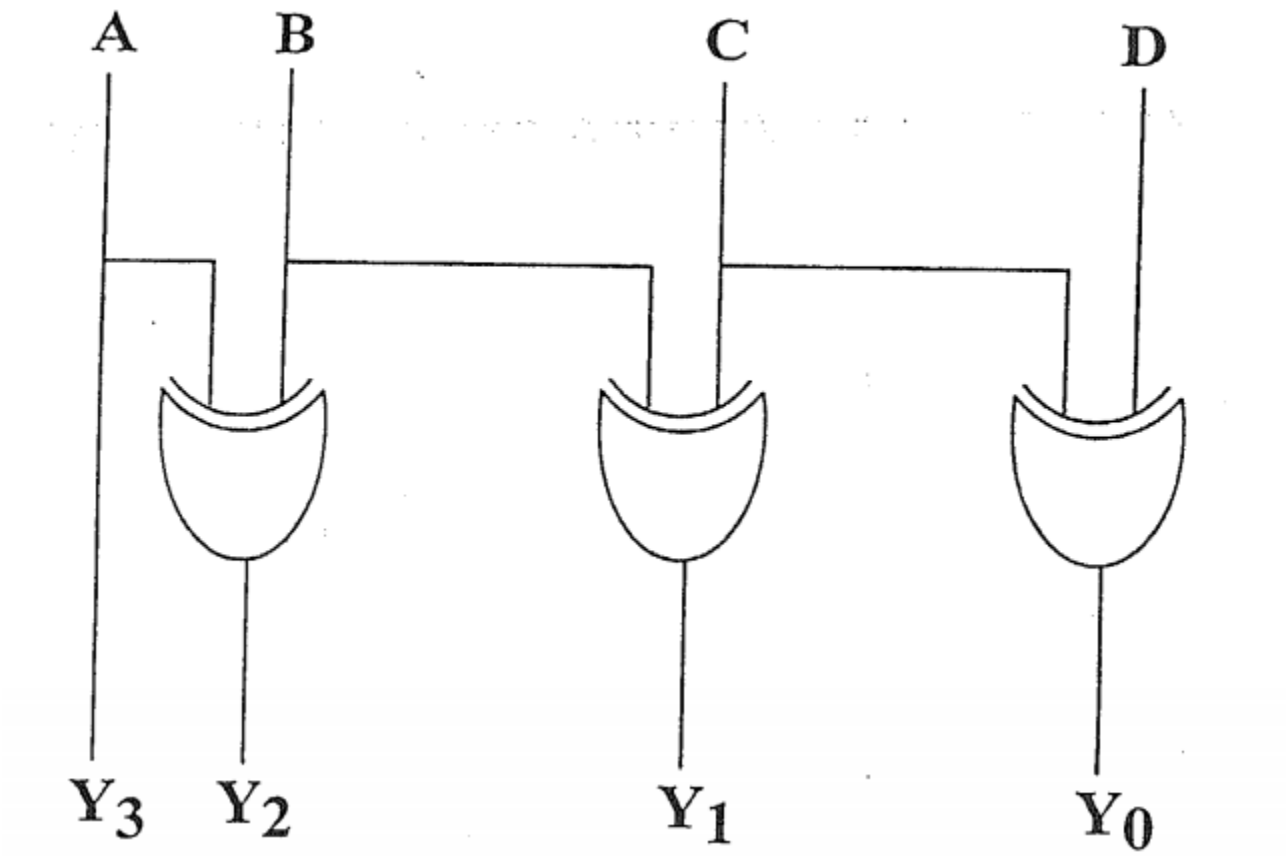
- After minimization,

  $Y3=A$

  $Y2=A \oplus B$

  $Y1=B \oplus C$

  $Y0=C \oplus D$


A
B
C
D
Y3
Y2
Y1
Y0

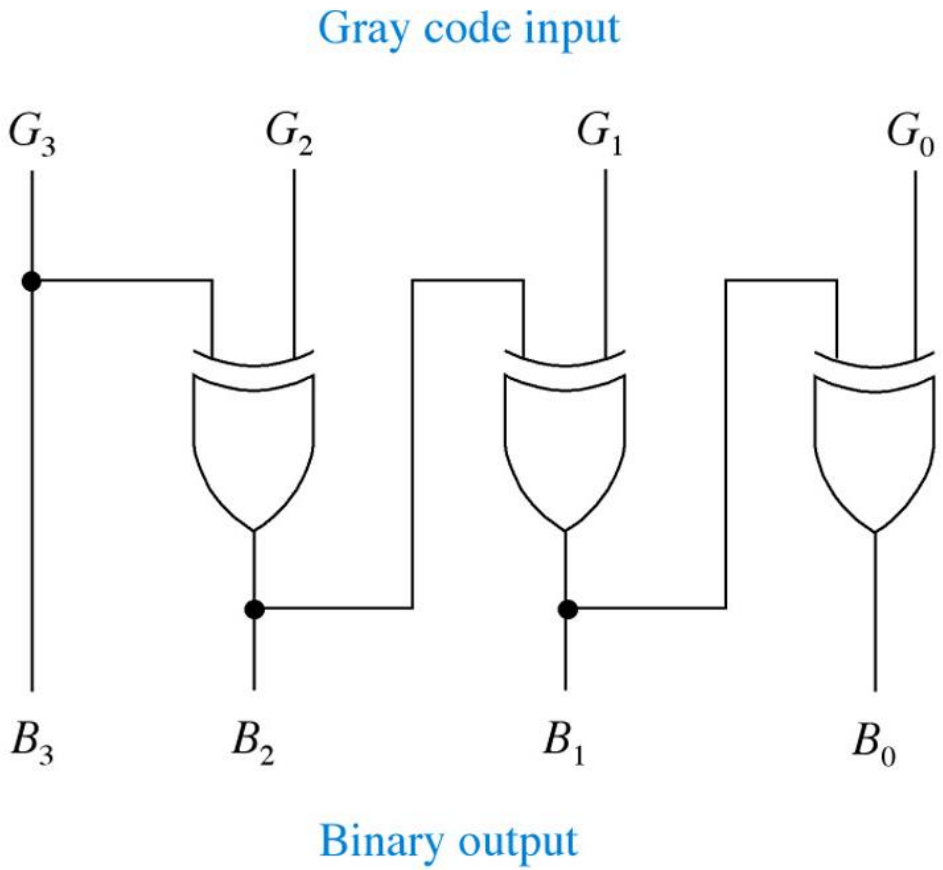
Gray to Binary Converter

$B3 = G3$

$B2 = G2 \oplus G3$

$B1 = G1 \oplus G2 \oplus G3$

$B0 = G0 \oplus G1 \oplus G2 \oplus G3$


Gray code input
G_3
G_2
G_1
G_0
B_3
B_2
B_1
B_0
Binary output

Assignment

- Design a Gray Code to BCD converter

Truth Table

Decimal	Gray Code				BCD			
	A	B	C	D	W	X	Y	Z
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	D	D	D	D
11	1	1	1	0	D	D	D	D
12	1	0	1	0	D	D	D	D
13	1	0	1	1	D	D	D	D
14	1	0	0	1	D	D	D	D
15	1	0	0	0	D	D	D	D

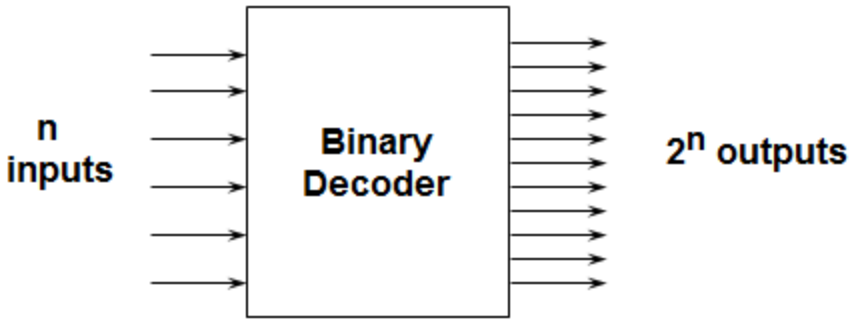
Decoder

Decoder

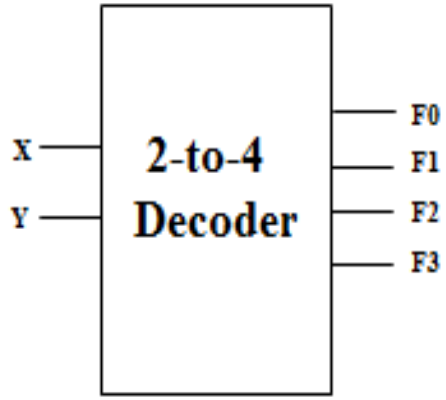
- It is a digital circuit that converts an input binary code into a single numeric output.
  - A decoder has
    - n inputs
    - 2^n outputs

- n to 2^n-line decoder.

- A decoder selects one of 2^n outputs by decoding the binary value on the N inputs.
- The decoder generates all of the minterms of the n input variables.
  - Exactly one output will be active for each combination of the inputs.



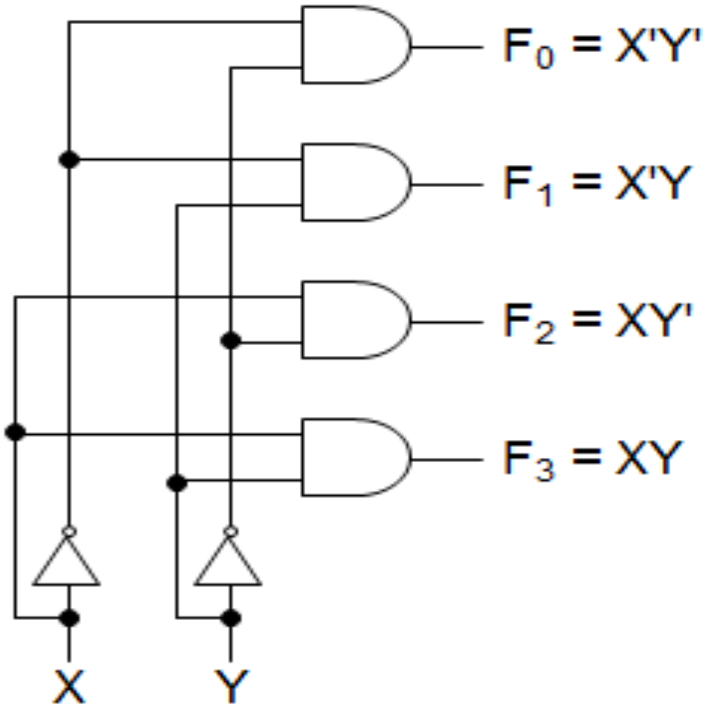
2-to-4 Binary Decoder



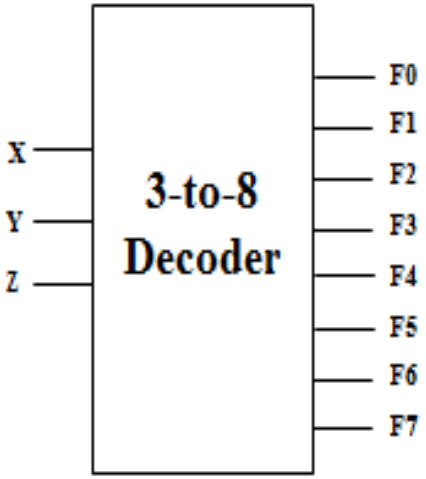
Truth Table:

X	Y	F_0	F_1	F_2	F_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Note: Each output is a 2-variable minterm (X'Y', X'Y, XY' or XY)

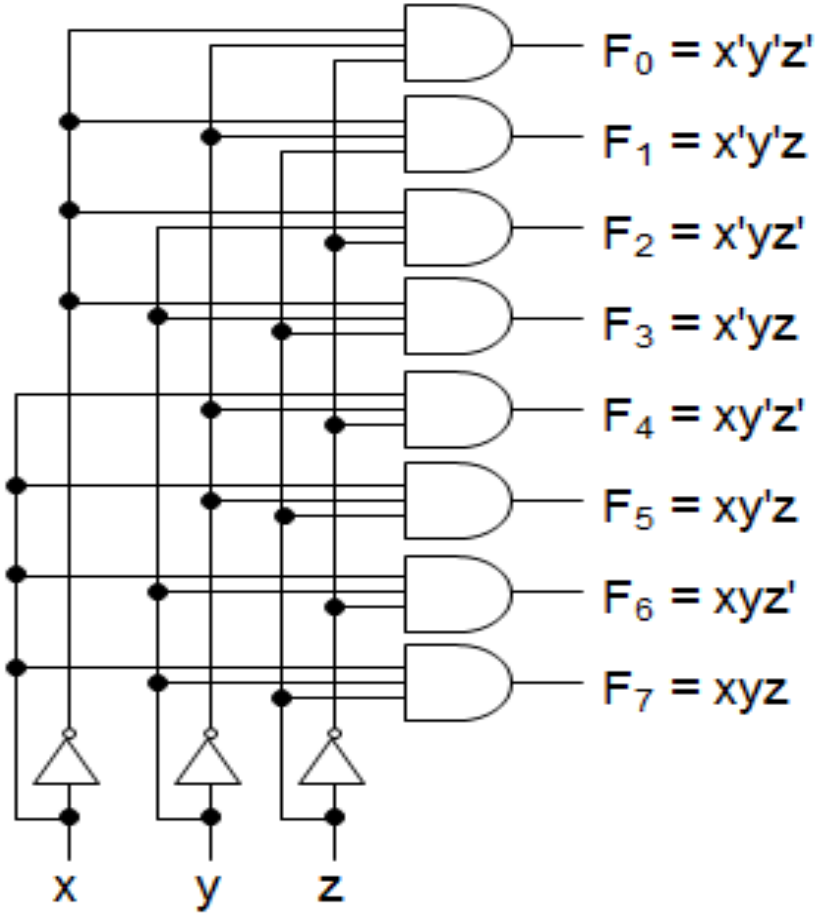

F0 = X'Y'
F1 = X'Y
F2 = XY'
F3 = XY
X
Y

3-to-8 Binary Decoder



Truth Table:

x	y	z	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_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


F0 = x'y'z'
F1 = x'y'z
F2 = x'yz'
F3 = x'yz
F4 = xy'z'
F5 = xy'z
F6 = xyz'
F7 = xyz
x
y
z

Implementing Functions Using Decoders

- Any n-variable logic function can be implemented using a single n-to-2^n decoder to generate the minterms
  - OR gate forms the sum.
  - The output lines of the decoder corresponding to the minterms of the function are used as inputs to the or gate.

- Any combinational circuit with *n* inputs and *m* outputs can be implemented with an n-to-2^n decoder with *m* OR gates.
- Suitable when a circuit has many outputs, and each output function is expressed with few minterms.

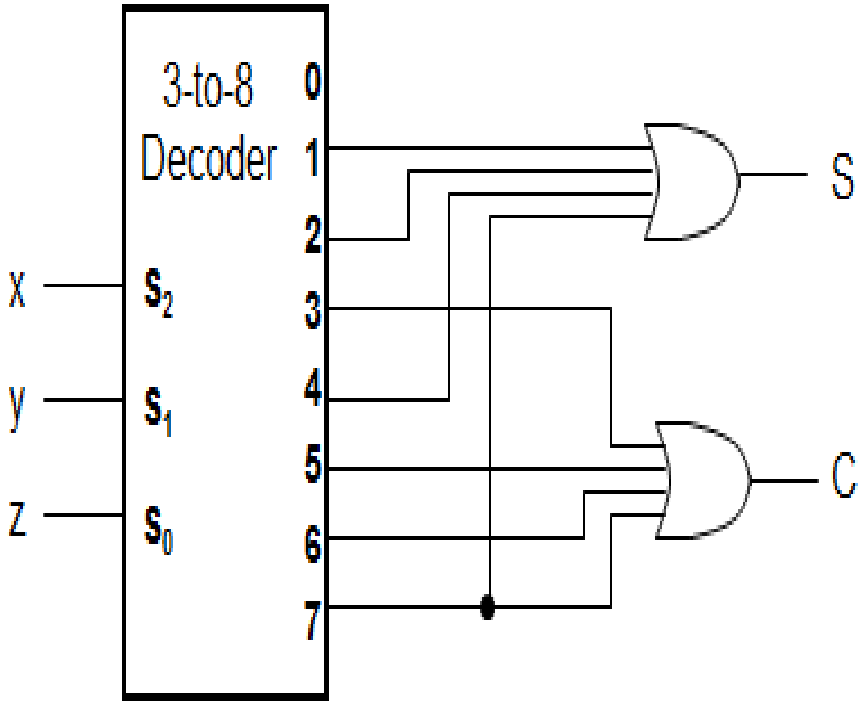
Example

- Implement the following full adder with a decoder and logic gates

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

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

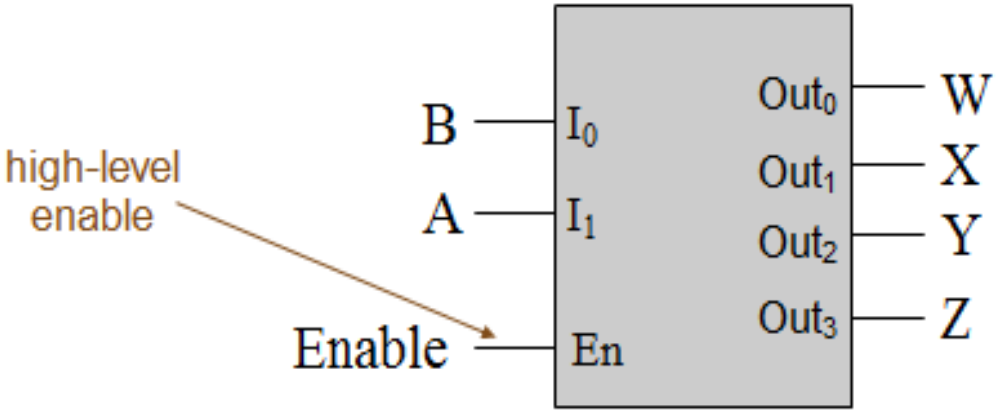
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Decoders with enable inputs

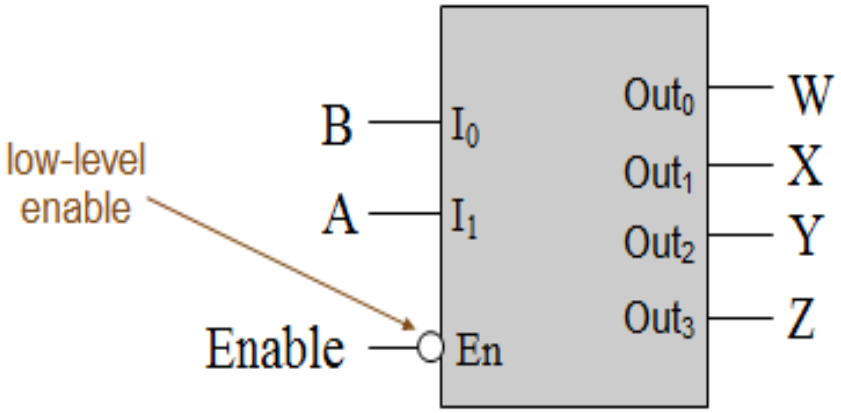
- Enable inputs are useful when constructing larger decoders from smaller decoders.
- When disabled, all outputs of the decoder can either be at logic-0 or logic-1.

Decoder with Enable



	En	A	B	W	X	Y	Z
enabled	1	0	0	1	0	0	0
	1	0	1	0	1	0	0
	1	1	0	0	0	1	0
	1	1	1	0	0	0	1
disabled	0	x	x	0	0	0	0

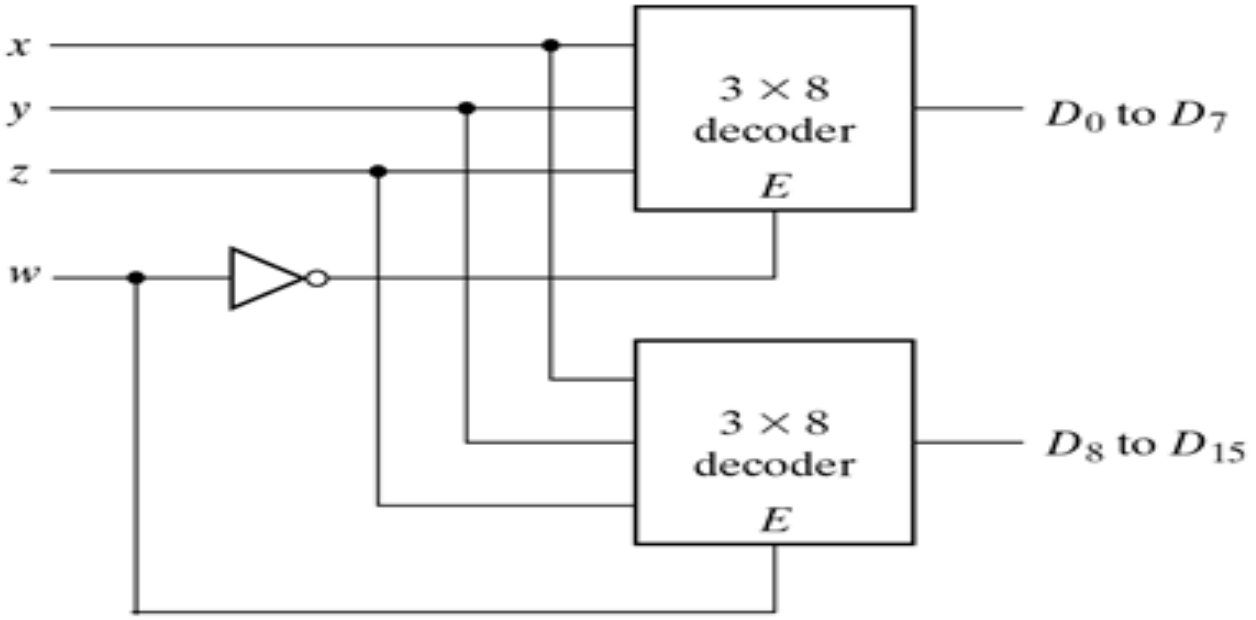
Decoder with Enable



	En	A	B	W	X	Y	Z
enabled	0	0	0	1	0	0	0
	0	0	1	0	1	0	0
	0	1	0	0	0	1	0
	0	1	1	0	0	0	1
disabled	1	x	x	0	0	0	0

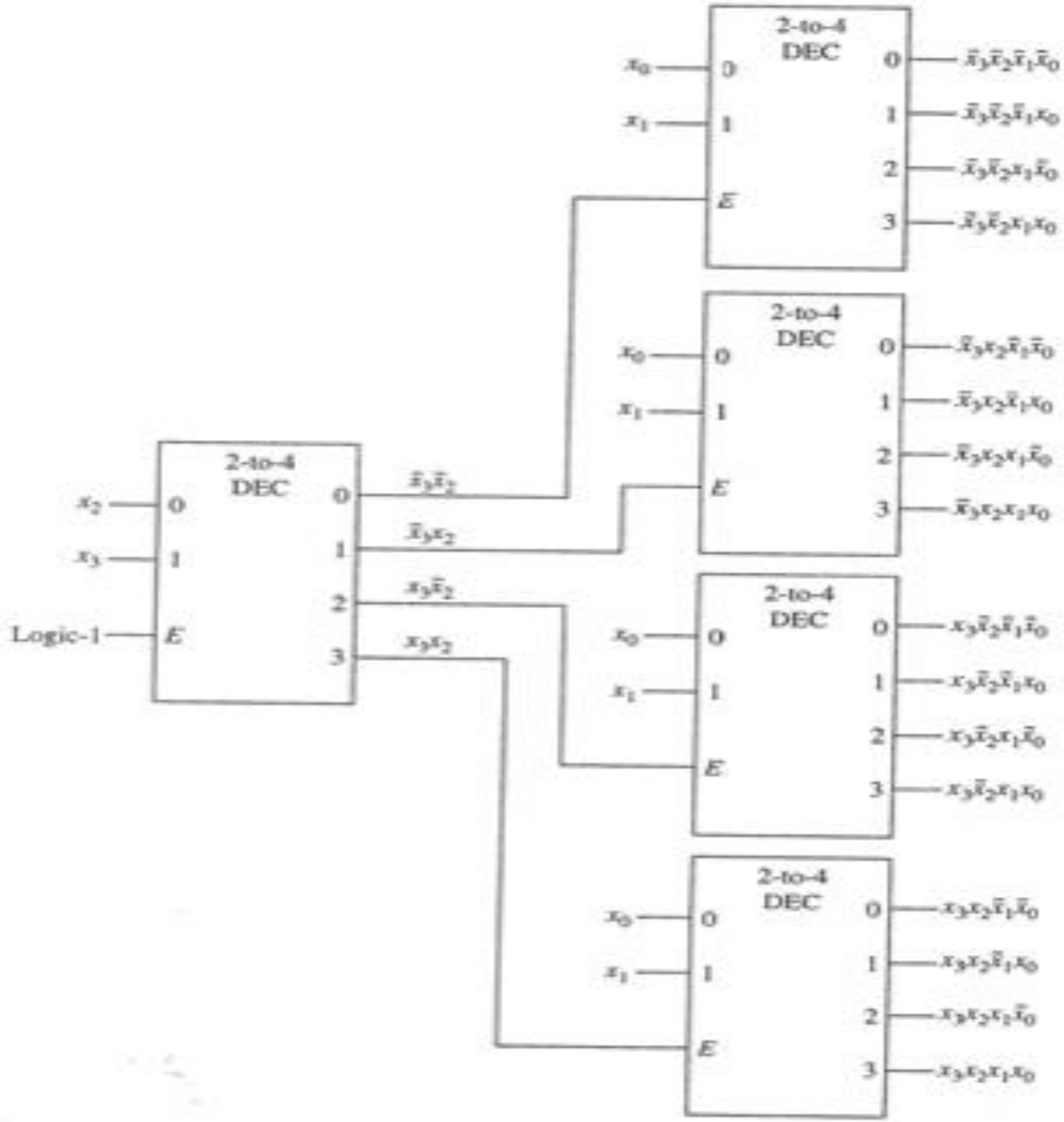
Use two 3 to 8 decoders to make 4 to 16 decoder

- Enable can also be active high
- In this example, only one decoder can be active at a time.
- x, y, z effectively select output line for w



Assignment:

Design a 4-to-16 decoder using
2-to-4 decoders only.


2-to-4
DEC
x_2
x_3
Logic-1
E
0
1
2
3
$\bar{x}_3\bar{x}_2$
$\bar{x}_3x_2$
$x_3\bar{x}_2$
x_3x_2
x_0
x_1
$\bar{x}_3\bar{x}_2\bar{x}_1\bar{x}_0$
$\bar{x}_3\bar{x}_2\bar{x}_1x_0$
$\bar{x}_3\bar{x}_2x_1\bar{x}_0$
$\bar{x}_3\bar{x}_2x_1x_0$
$\bar{x}_3x_2\bar{x}_1\bar{x}_0$
$\bar{x}_3x_2\bar{x}_1x_0$
$\bar{x}_3x_2x_1\bar{x}_0$
$\bar{x}_3x_2x_1x_0$
$x_3\bar{x}_2\bar{x}_1\bar{x}_0$
$x_3\bar{x}_2\bar{x}_1x_0$
$x_3\bar{x}_2x_1\bar{x}_0$
$x_3\bar{x}_2x_1x_0$
$x_3x_2\bar{x}_1\bar{x}_0$
$x_3x_2\bar{x}_1x_0$
$x_3x_2x_1\bar{x}_0$
$x_3x_2x_1x_0$

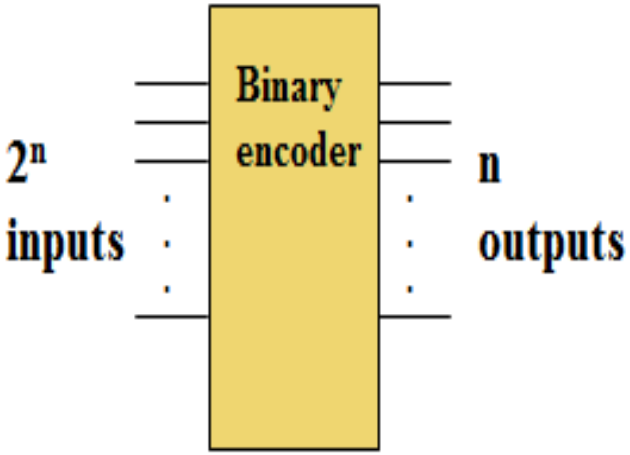
Encoders

Encoders

- An encoder performs the inverse operation of a decoder.
- At any one time, only one input line has a value of 1.
- If the a decoder's output code has fewer bits than the input code, the device is usually called an encoder.

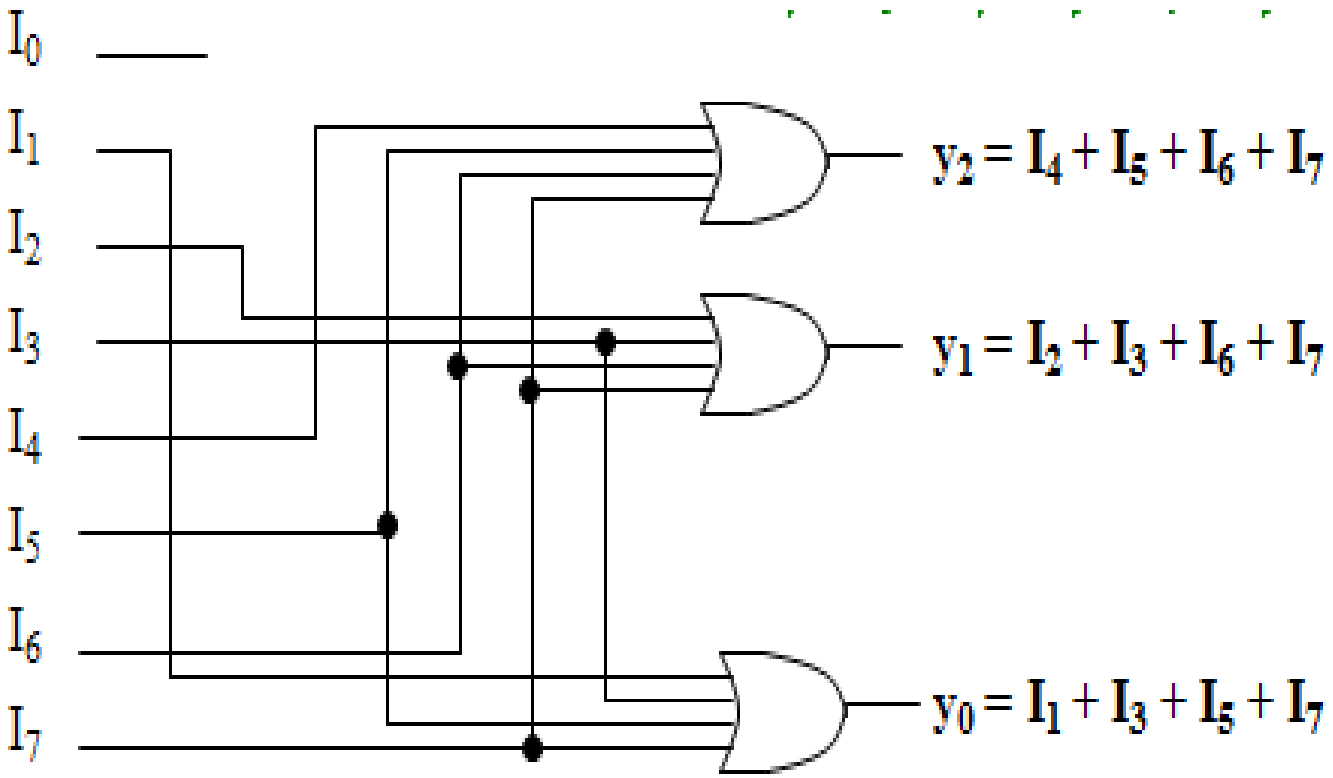
  e.g. 2^n-to-n

- The simplest encoder is a 2^n-to-n binary encoder
  - One of 2^n inputs = 1
  - Output is an n-bit binary number



8-to-3 Binary Encoder

Inputs								Outputs		
I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	y_2	y_1	y_0
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


I0
I1
I2
I3
I4
I5
I6
I7
y2 = I4 + I5 + I6 + I7
y1 = I2 + I3 + I6 + I7
y0 = I1 + I3 + I5 + I7

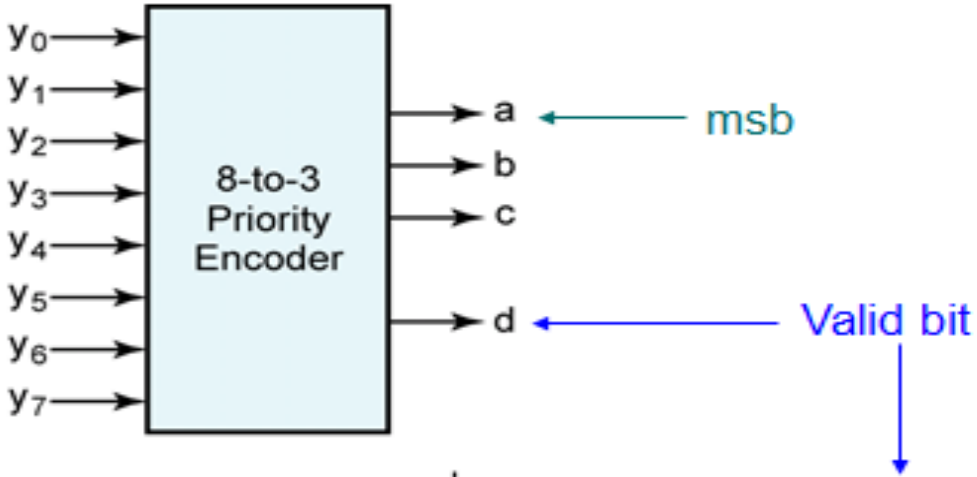
Priority Encoder

- A priority scheme is assigned to the input lines so that whenever more than one input line is asserted at any time, the output is determined by the input line having the highest priority.

Priority Encoder

- If more than one input is active, the higher-order input has priority over the lower-order input.
  --The higher value is encoded on the output
- A valid indicator, d, is included to indicate whether or not the output is valid.
  - Output is invalid when no inputs are active
    - d = 0
  - Output is valid when at least one input is active
    - d = 1
- Valid indicates that at least one input line is asserted.

Priority Encoder



y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	a	b	c	d
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
X	1	0	0	0	0	0	0	0	0	1	1
X	X	1	0	0	0	0	0	0	1	0	1
X	X	X	1	0	0	0	0	0	1	1	1
X	X	X	X	1	0	0	0	1	0	0	1
X	X	X	X	X	1	0	0	1	0	1	1
X	X	X	X	X	X	1	0	1	1	0	1
X	X	X	X	X	X	X	1	1	1	1	1

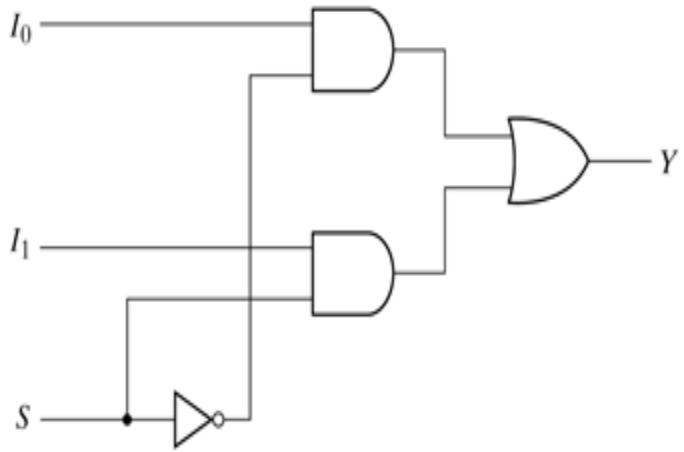
Multiplexer

Multiplexers

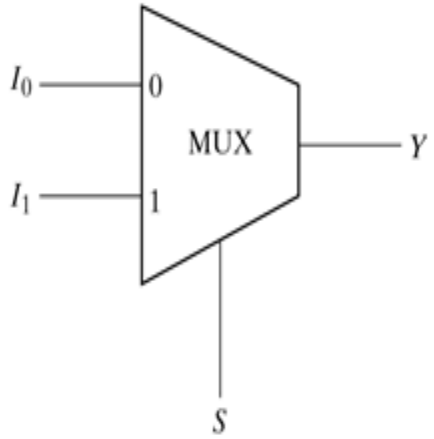
- Allows for conditional transfer of data
- Sometimes called a mux
  - Select one out of several bits
  - Some inputs used for selection
  - Also can be used to implement logic

- Normally, there are 2^n input lines and n selection lines

2:1 MUX



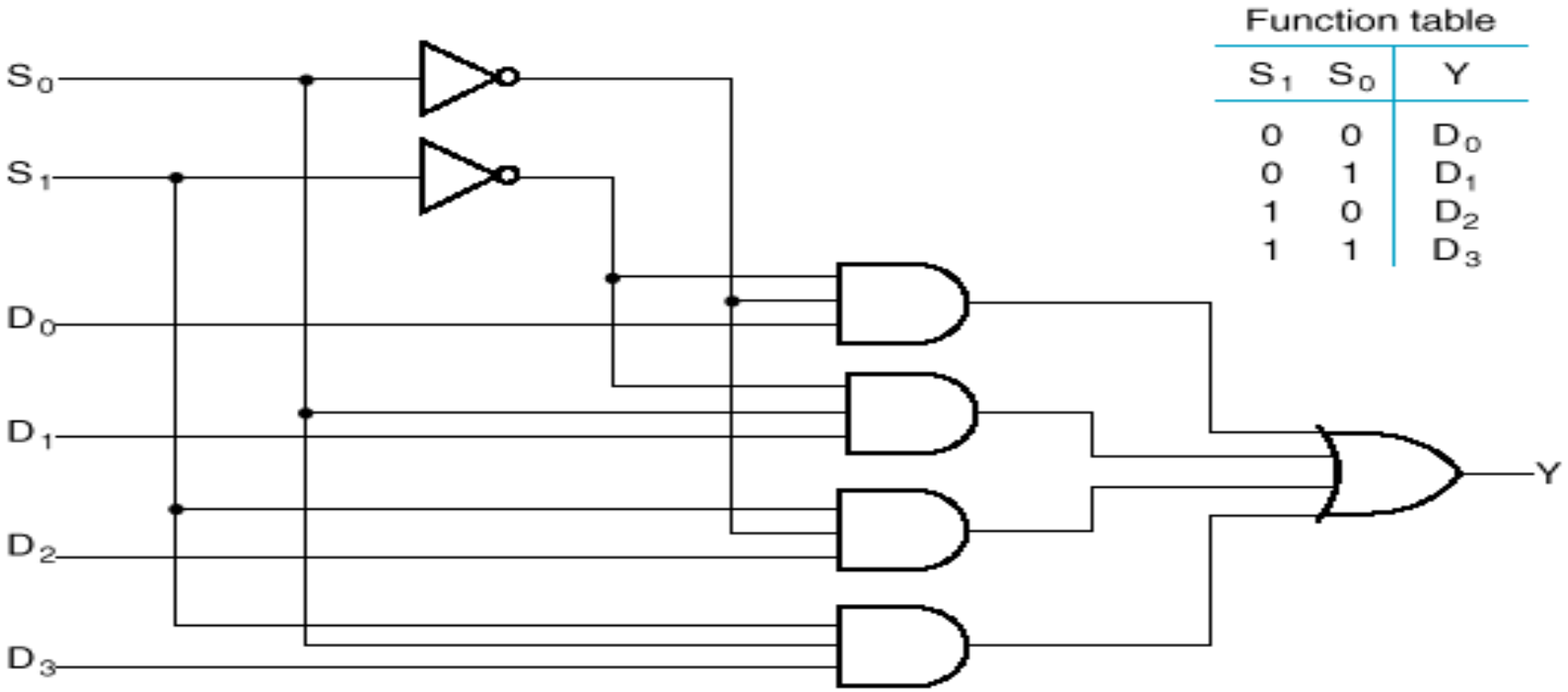
(a) Logic diagram



(b) Block diagram

4:1 MUX

- Data select control determines which input is transmitted



Function table

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Multiplexers

- Provide Combination Logic Functions
  - Multiplexers can be used to implement combinational logic circuits.
  - A multiplexer can replace several SSI logic gates

Implementing Boolean function using MUX

- This is a general procedure for implementing any Boolean function with n variables with a 2^{n-1}: 1 Mux
  - Express the function in SOP form
  - Out of n variables, one is used as input and remaining (n-1) are used as selection lines
  - The input variable is complemented in minterms 0 to 2^{n-1}-1 and uncomplemented in remaining minterms
  -

- Arrange all minterms in 2 rows.
- First row contains all minterms in which the input variable (suppose A) is complemented and 2nd row lists all minterms with A uncomlemented
- Circle available minterms and inspect each culumn separately
  - If 2 minterms in a column are not circled, apply 0 to the MUX input
  - If 2 minterms in a column are circled, apply 1
  - If bottom minterm in a column is circled, apply A
  - If top minterm in a column is circled, apply A’

Example

- $F(A, B, C)=\sum(1, 3, 5, 6)$
- $F(A, B, C, D)=\sum(0, 1, 3, 4, 8, 9, 15)$

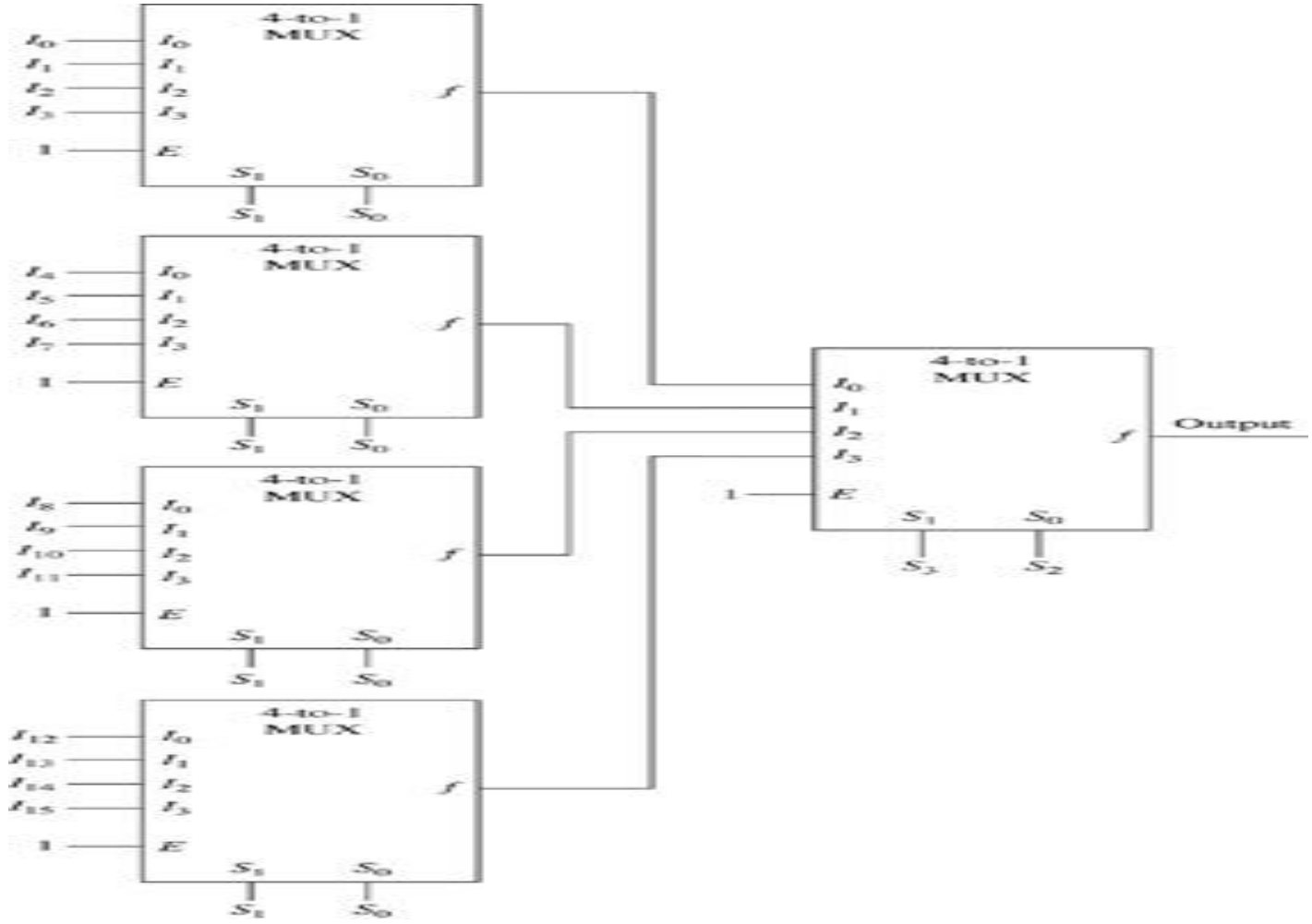
Multiplexer Tree

- A larger MUX can be implemented using a tree of smaller MUXes.
- A 16:1 MUX can be designed by
  - Two 8:1 MUXes and one 2:1 MUX or an OR gate
  - Four 4:1 MUXes
  - Eight 2:1 MUXes and one 8:1 MUX
- A 32:1 MUX can be constructed by using
  - Two 16:1 MUXes and one 2:1 MUX or an OR gate
  - Four 8:1 MUXes and a 4:1 MUX

Example

- Design an 8:1 MUX using 4:1 MUXes
- Design a 32:1 MUX using 16:1 MUXes

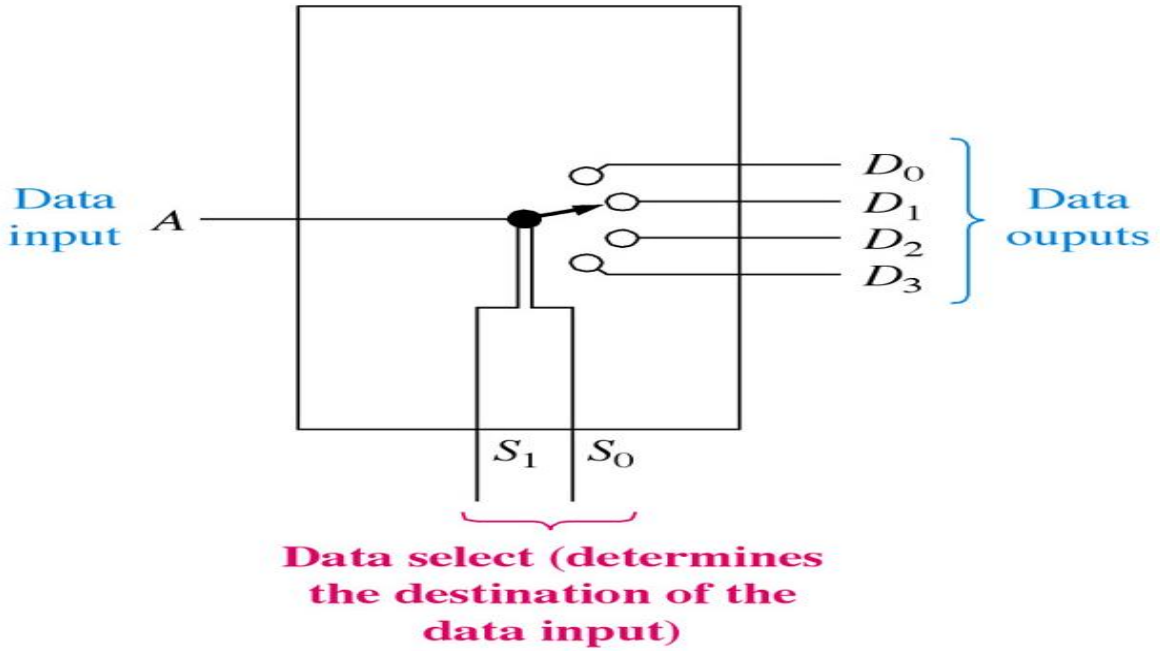
A multiplexer tree to form a 16-to-1-multiplexer.



Demultiplexers

Demultiplexers

- Opposite procedure from multiplexing
- Route a single input to one of many outputs
- Data distributor
- Single data input routed to one of several outputs



Example

- Design a 1:8 DEMUX using 1:4 DEMUXes