

INTRODUCTION TO LOGIC

Chapter 4

Combinational Logic

OUTLINE OF CHAPTER 4

- Combinational Circuits
- Analysis Procedure
- Design Procedure
- Binary Adder Subtractor
- Decimal Adder
- Binary Multiplier
- Magnitude Comparator
- Decoders
- Encoders
- Multiplexers



4.1 COMBINATIONAL CIRCUITS

COMBINATIONAL CIRCUITS

- Logic circuits for digital systems
  - Combinational
  - Sequential
- A combinational circuit
  - outputs at any time are determined from the present combination of inputs.
  - Performs operation specified by a set of Boolean functions.

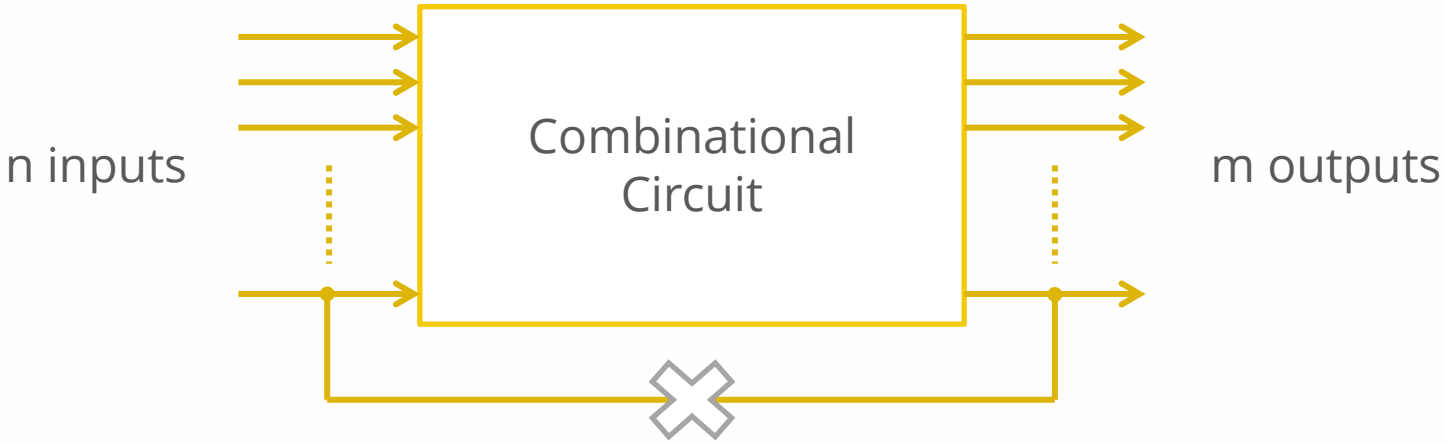
COMBINATIONAL CIRCUITS

- Sequential circuits
  - Employ storage elements in addition to logic gates.
  - Their outputs are a function of inputs and state of the storage elements.
  - Not only present values of inputs, but also on past inputs.
  - Circuit behaviour must be specified by a time sequence of inputs and internal states.
  - Sequential circuits are the building blocks of digital systems.

COMBINATIONAL CIRCUITS

- Combinational circuit consist of
  - Input variables
  - Logic gates
    - Accept signals from the inputs
    - Generate signals to the output
  - Output variables

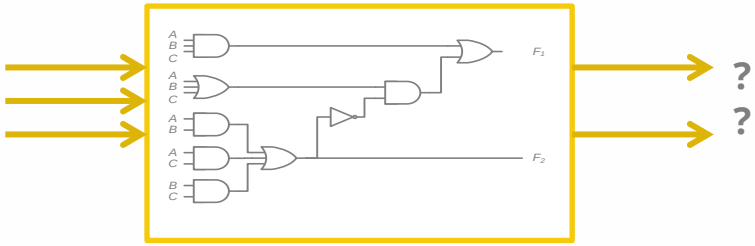
COMBINATIONAL CIRCUITS



- For n input variable, there are 2n possible binary input combinations.
- For each possible input combination there is one possible input output value.

COMBINATIONAL CIRCUITS

- Analysis
  - Given a circuit, find out its ***function***
  - Function may be expressed as:
    - Boolean function
    - Truth table



- Design
  - Given a desired function, determine its ***circuit***
  - Function may be expressed as:
    - Boolean function
    - Truth table





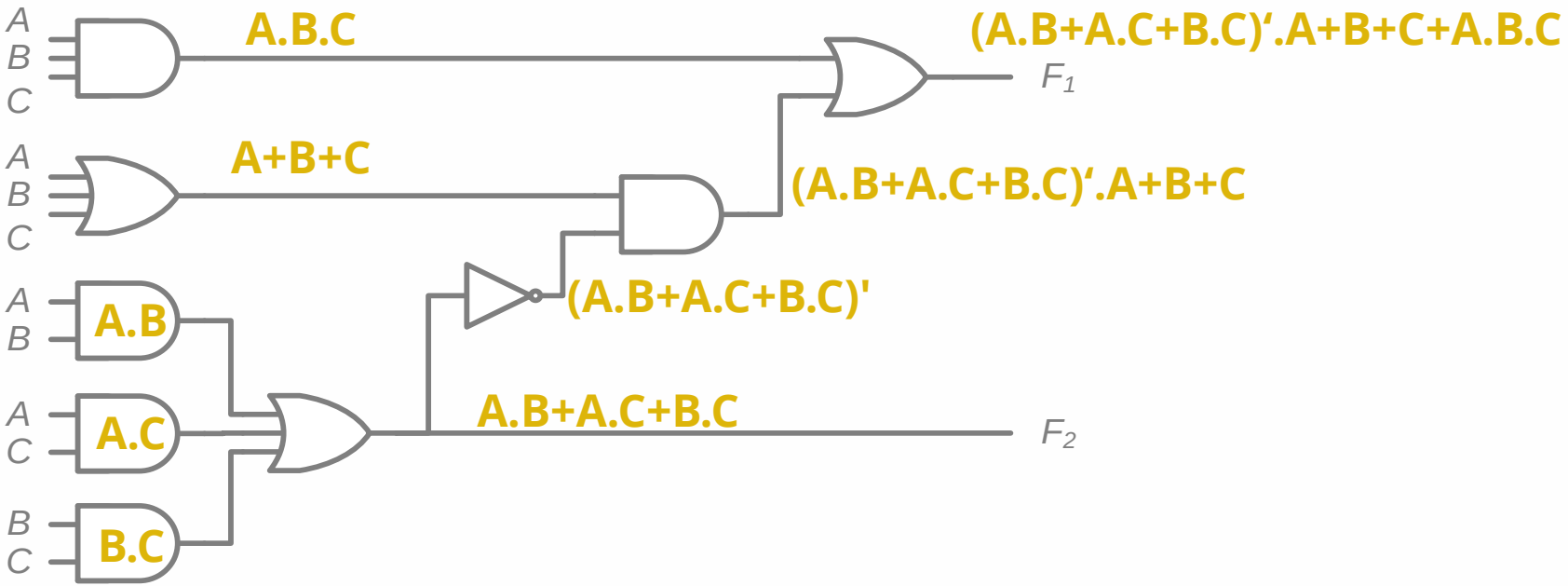
4.2 ANALYSIS PROCEDURE

ANALYSIS PROCEDURE

- To obtain the output Boolean functions from a logic diagram:
  1. Label all gate outputs that are function of input variables
     - Determine the Boolean functions for each gate output
  2. Label the gates that are a function of input variables and previously labelled gates.
     1. Find the Boolean function for these gates.
  3. Repeat the process outlined in step2 until the outputs are obtained.
  4. Obtain the output Boolean functions in terms of input variables.

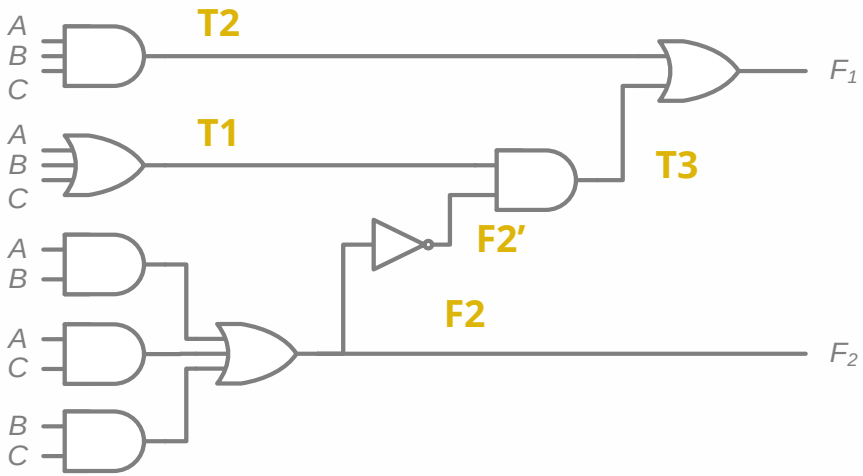
ANALYSIS PROCEDURE

- Boolean expression approach



ANALYSIS PROCEDURE

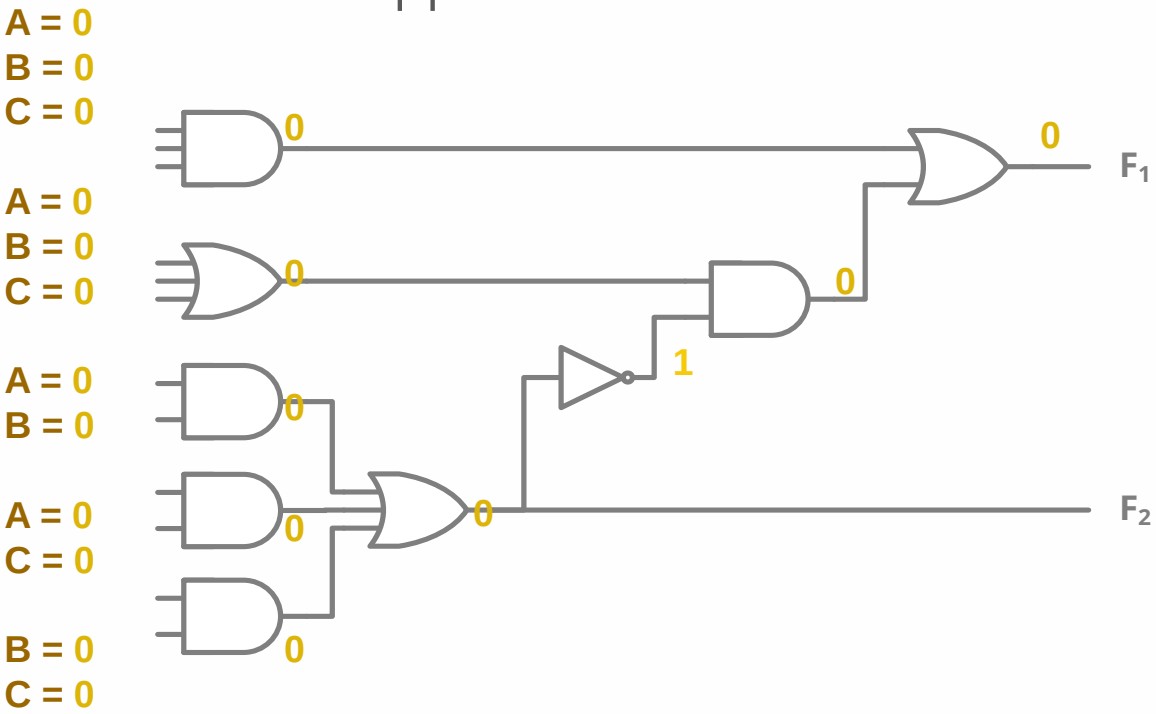
- Boolean expression approach



- T1 = A+B+C
- T2 = A.B.C
- F2 = A.B+A.C+B.C
- F2′ = (A′+B′)(A′+C′)(B′+C′) = (A′ + B′C′).(B′+C′) post4b
- T3 = F2′ . T1
- T3 = (A′+B′C′).(B′+C′).(A+B+C)
- T3 = (A′+B′C′).(AB′+B′C+AC′+BC′)
- T3 = (A′.A.B′+A′.B′.C+A′.A.C′+A′.B.C′+
- A.B′.B′.C′+B′.B′.C′.C+A.B′.C′.C′+B′.B.C′.C′)
- T3 =A′.B′.C + A′.B.C′ + A.B′.C′ + A.B′.C′
- T3 = A′.B′.C + A′.B.C′ + A.B′C′
- F1= T3+T2
- F1 = A′.B′.C + A′.B.C′ + A.B′C′ + A.B.C

ANALYSIS PROCEDURE

- Truth table approach



A	B	C	F_1	F_2
0	0	0	0	0

ANALYSIS PROCEDURE

- Truth table approach

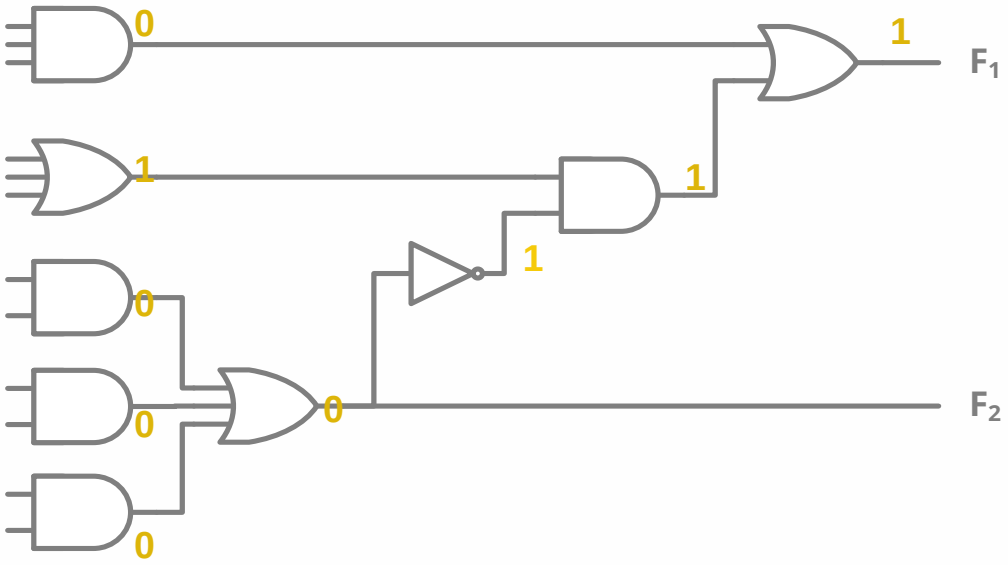
A = 0
B = 0
C = 1

A = 0
B = 0
C = 1

A = 0
B = 0

A = 0
C = 1

B = 0
C = 1



A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	1	0

ANALYSIS PROCEDURE

- Truth table approach

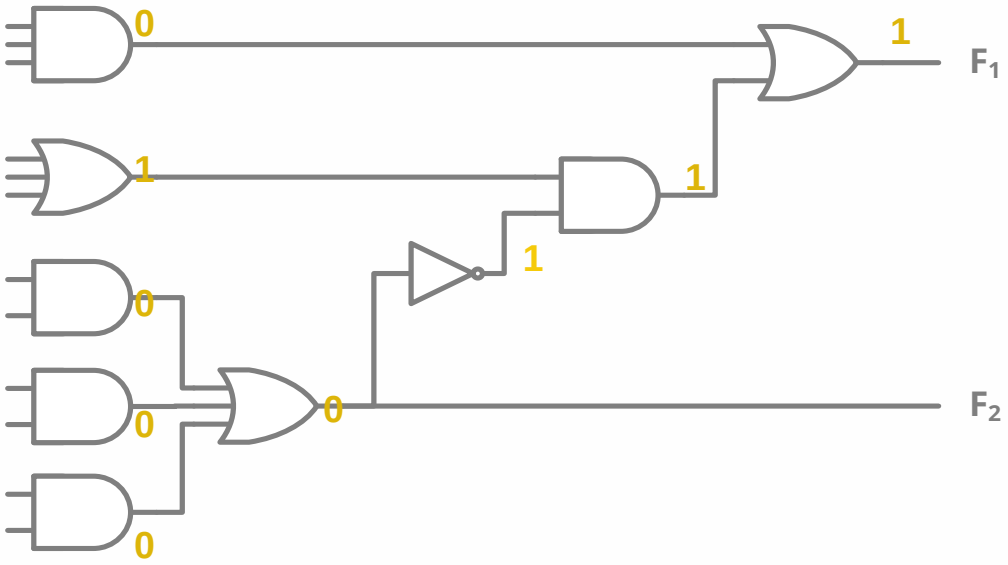
A = 0
B = 1
C = 0

A = 0
B = 1
C = 0

A = 0
B = 1

A = 0
C = 0

B = 1
C = 0



A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0

ANALYSIS PROCEDURE

- Truth table approach

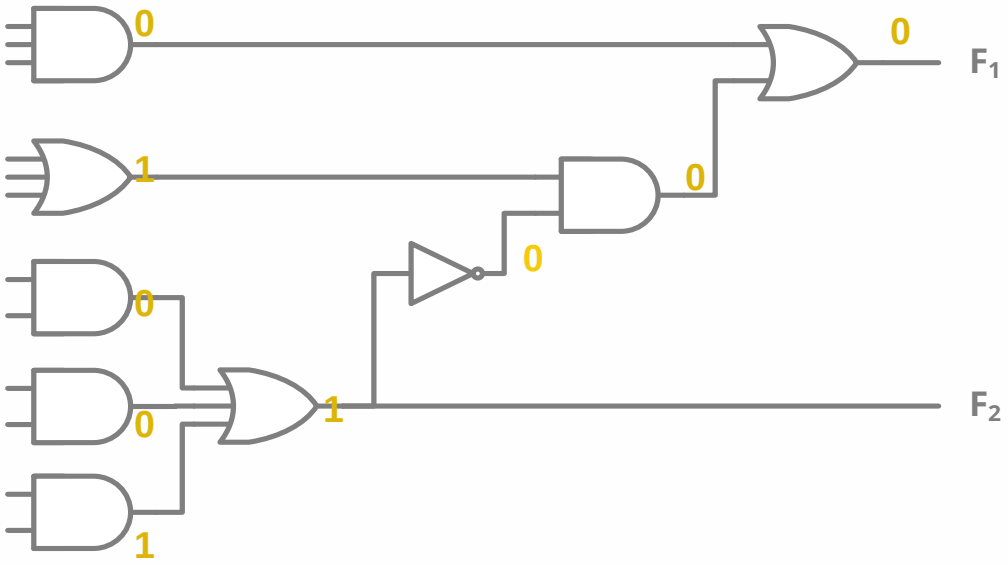
A = 0
B = 1
C = 1

A = 0
B = 1
C = 1

A = 0
B = 1

A = 0
C = 1

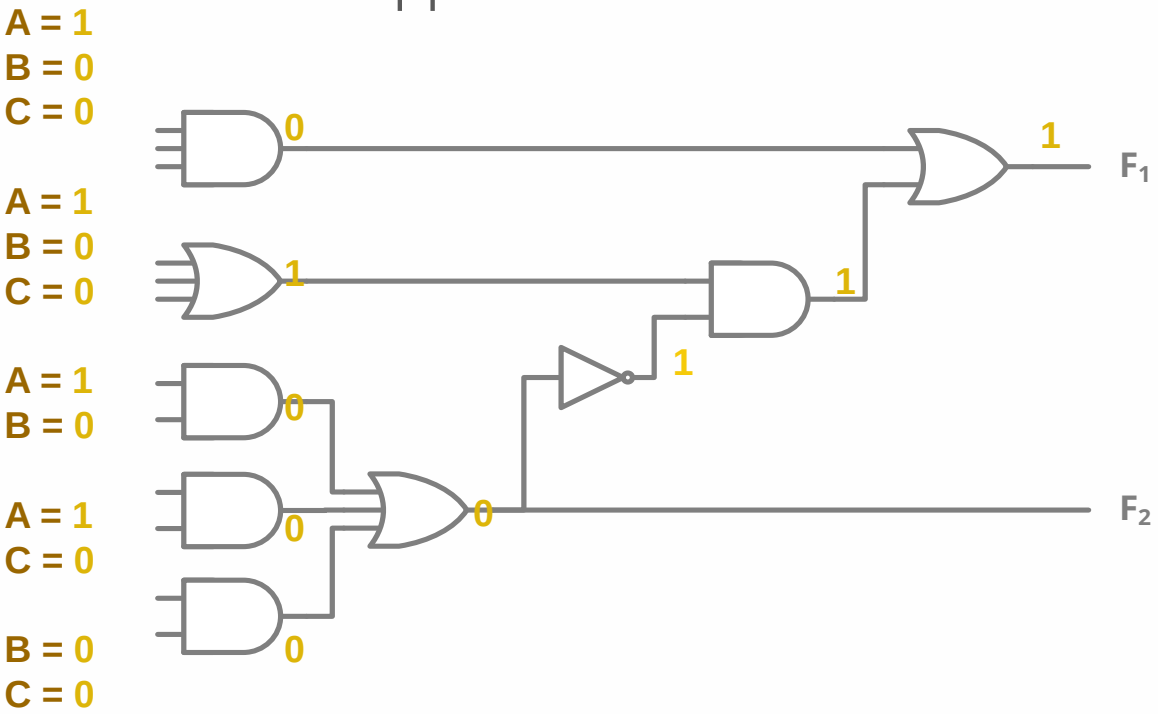
B = 1
C = 1



A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	0	0	1

ANALYSIS PROCEDURE

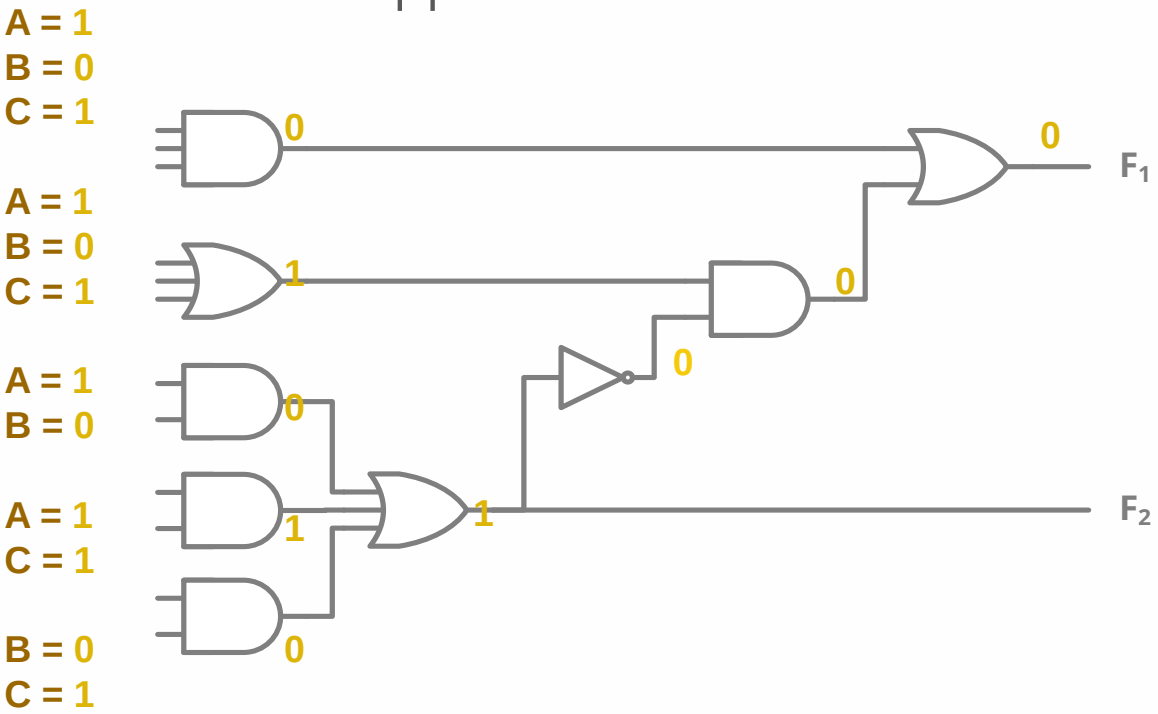
- Truth table approach



A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0

ANALYSIS PROCEDURE

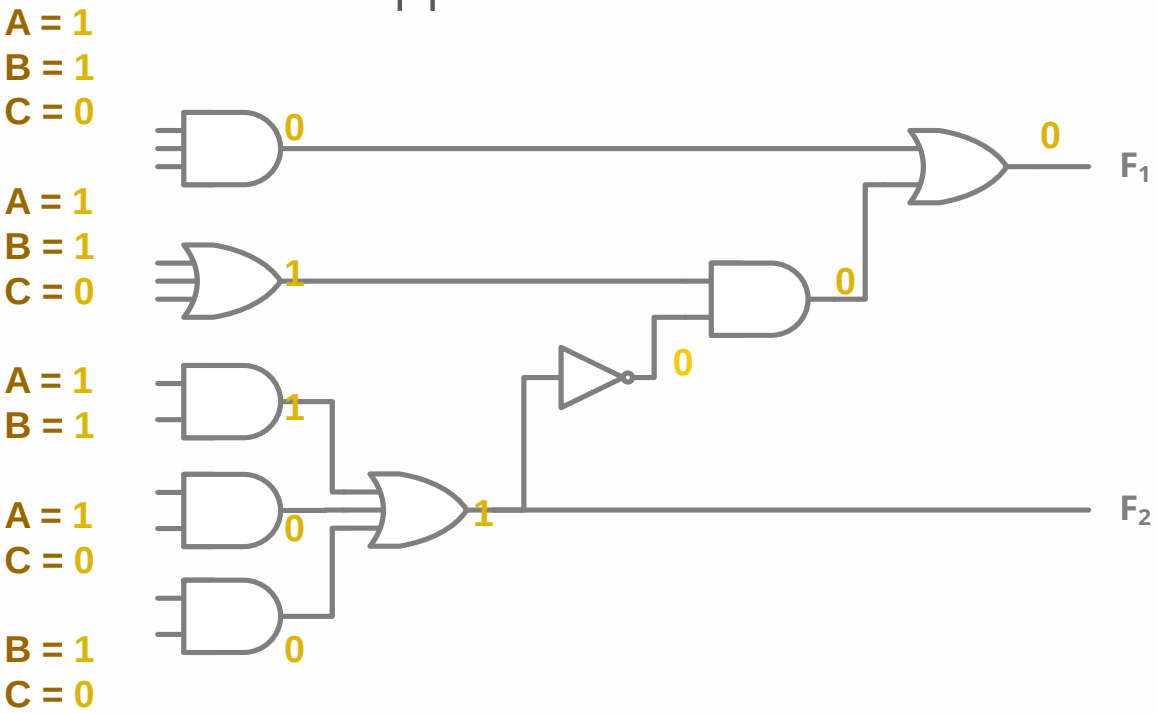
- Truth table approach



A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1

ANALYSIS PROCEDURE

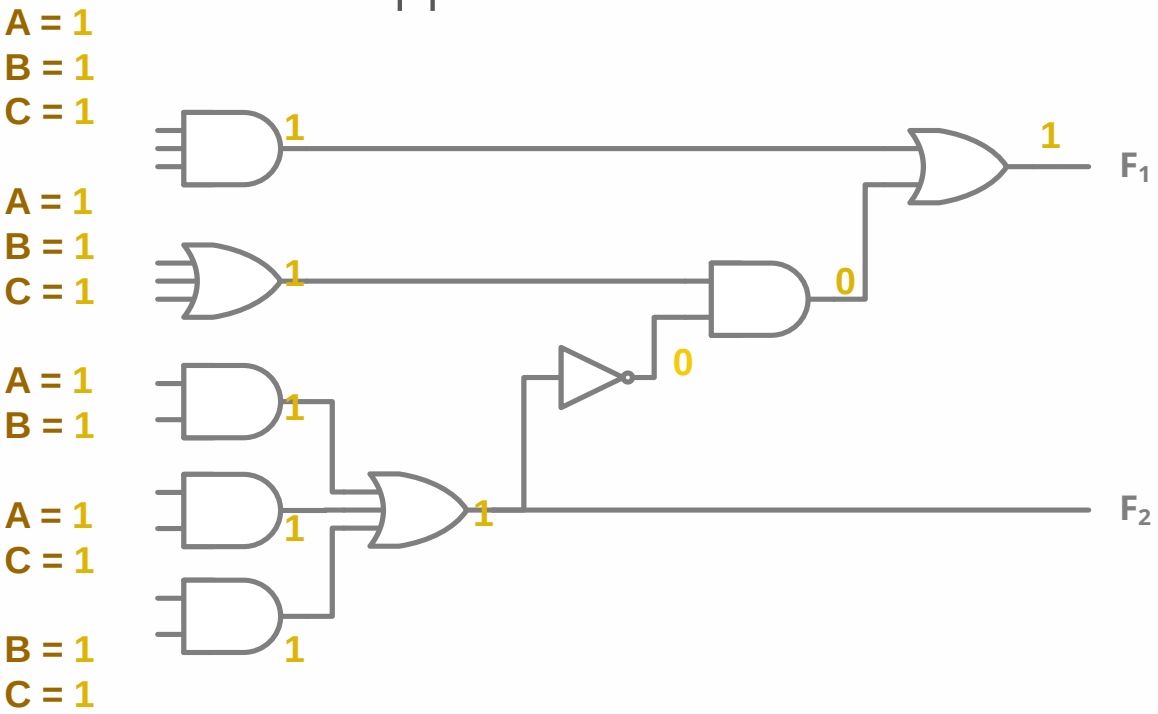
- Truth table approach



A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1

ANALYSIS PROCEDURE

- Truth table approach

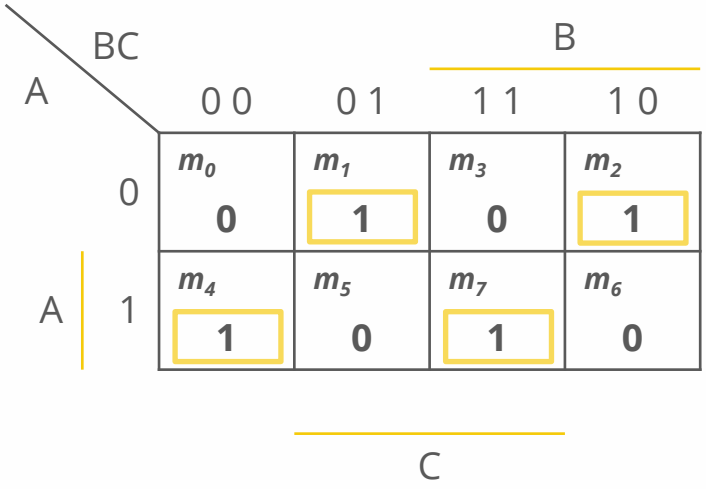


A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

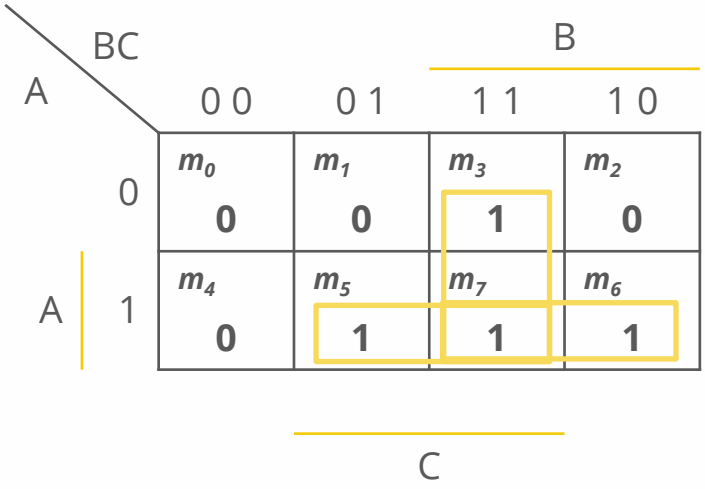
ANALYSIS PROCEDURE

- Truth table approach

A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



F1 = A'B'C + A'BC' + AB'C' + ABC



F2 = BC + AC + AB

4.3 DESIGN PROCEDURE

DESIGN PROCEDURE

- For a given a problem statement:
  - Determine the number of **inputs** and **outputs**
  - Derive the truth table
  - Simplify the Boolean expression for each output
  - Produce the required circuit

DESIGN PROCEDURE

- A truth table for a combinational circuit consist of:
  - Input columns
    - Obtained from 2^n binary numbers for the *n* input variables.
  - Output columns
    - Determined from the stated specifications.
    - Output functions specified in the truth table give exact definition of the combinational circuit.

DESIGN PROCEDURE

- The output binary functions listed in the truth table are simplified by any method:
  - Algebraic manipulation
  - The map method
  - Computer – based simplification program
- There is a variety of simplified expressions from which to choose.

DESIGN PROCEDURE

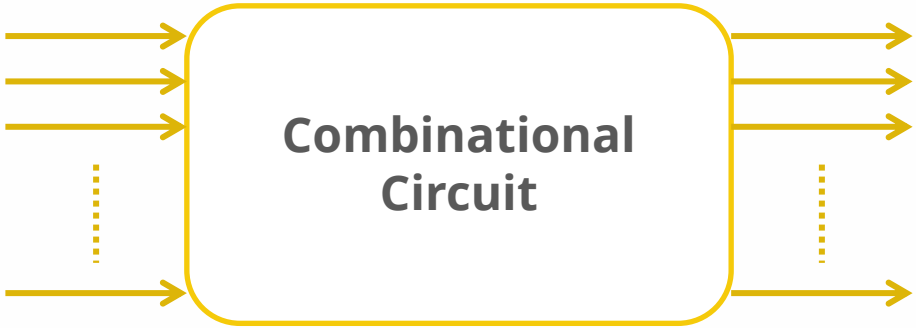
- Practical design must consider such constraints:
  - The number of gates
  - Number of inputs to a gate
  - Propagation time of the signal through the gates
  - Number of interconnections
  - Limitations of the driving capability of each gate
  - Etc.

DESIGN PROCEDURE

Code conversion example

- Code converter is a circuit that makes the two systems compatible even though each uses a different binary code.
- To convert from binary code A to binary code B;

Input:
supply the bit combination of elements specified by code A

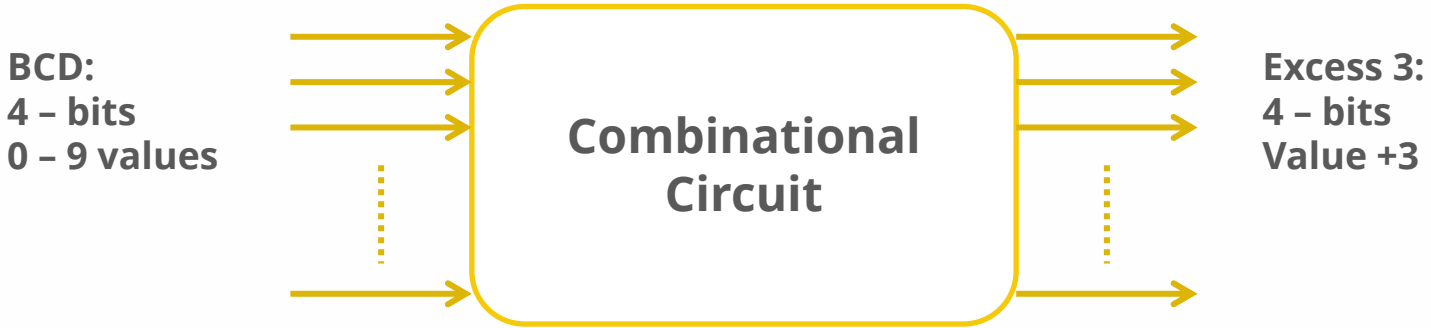


Output:
generate the corresponding bit combination of code B.

DESIGN PROCEDURE

Example:

- Design a circuit to convert a “BCD” code to “Excess 3” code.



DESIGN PROCEDURE

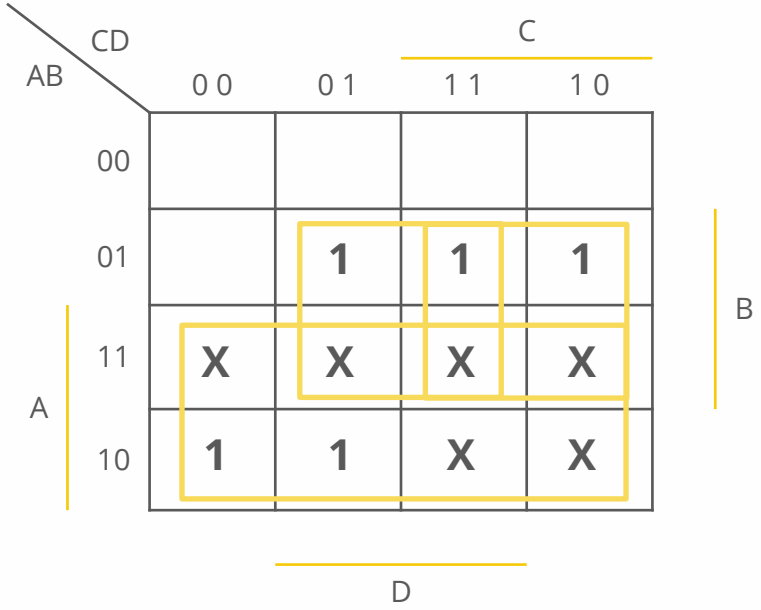
- Each code uses 4-bits to represent a decimal digit.
  - 4 input variables
    - A, B, C, D
  - 4 output variables
    - w, x, y, z
  - 2^4 = 16 bit combinations but only 10 have meaning in BCD.
  - Rest 6 bit combinations are don't-care combinations.

Input BCD	Output Excess-3
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 0	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

DESIGN PROCEDURE

Input BCD	Output Excess-3
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 0	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

w's karnaugh map

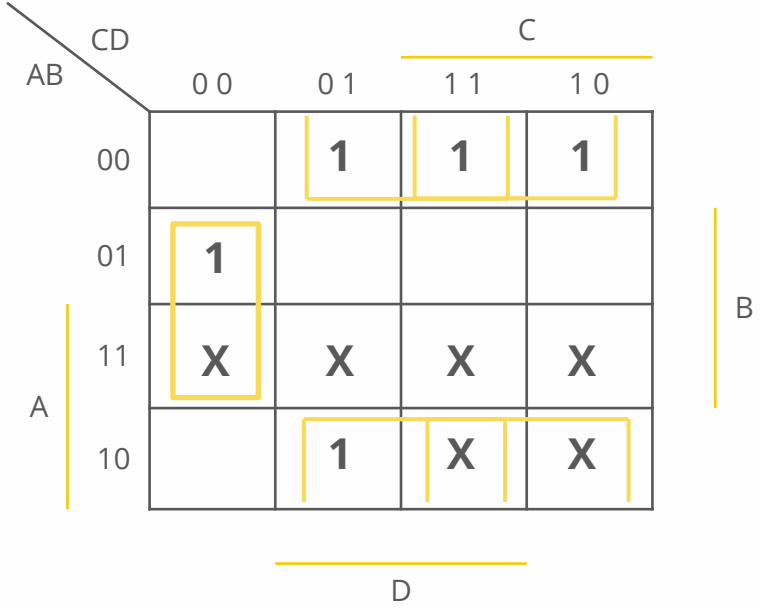


w = A + BC + BD

DESIGN PROCEDURE

Input BCD	Output Excess-3
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 0	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

x's karnaugh map

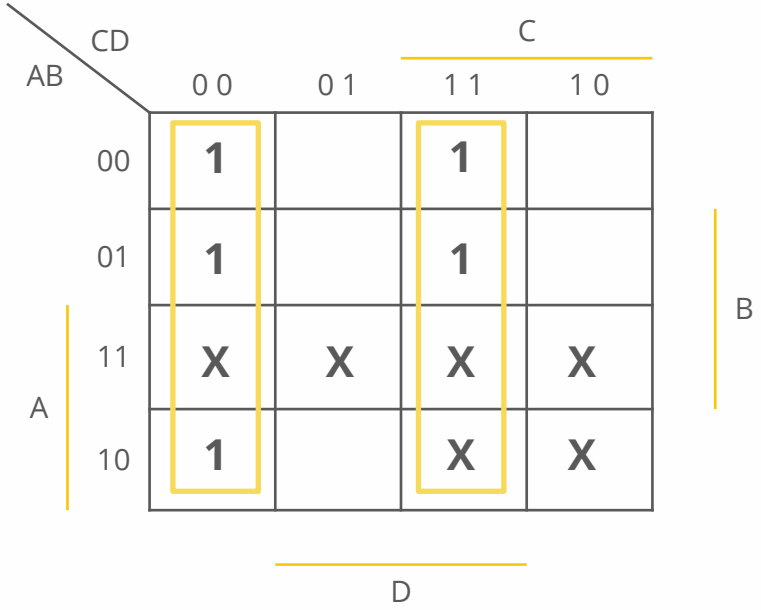


x = B'C + B'D + BC'D'

DESIGN PROCEDURE

Input BCD	Output Excess-3
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 0	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

y's karnaugh map

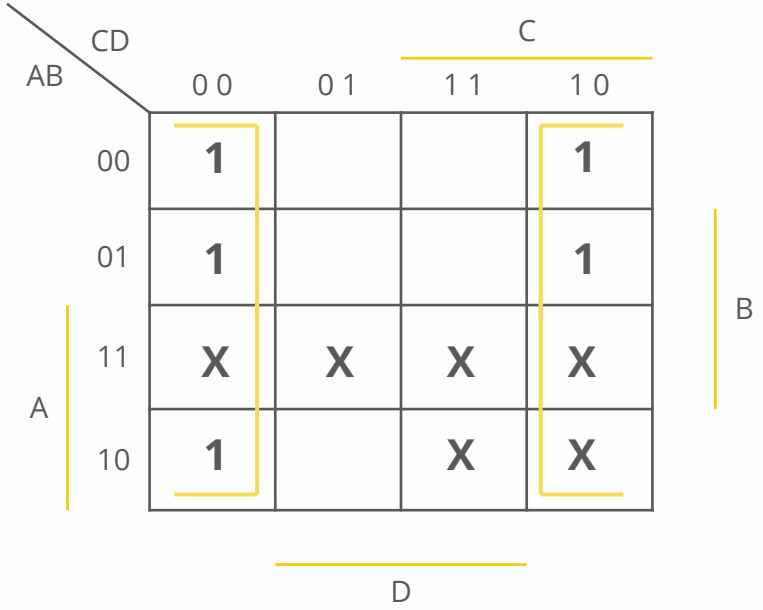


y = CD + C'D'

DESIGN PROCEDURE

Input BCD	Output Excess-3
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 0	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

z's karnaugh map

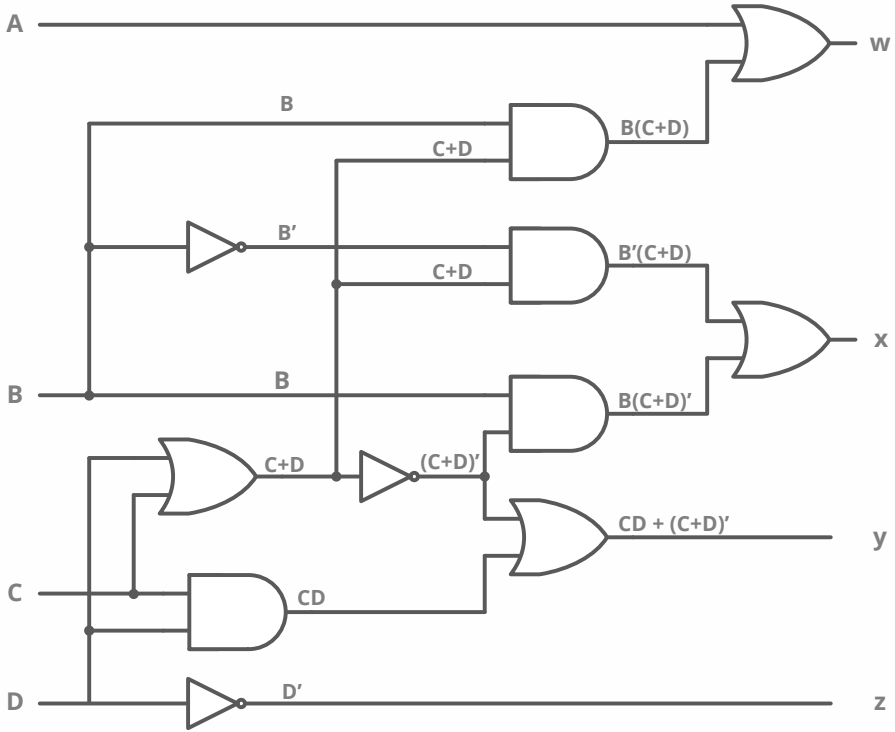


z = D'

DESIGN PROCEDURE

- w = A + BC + BD = A + B (C + D)
- x = B′C + B′D + BC′D′ = B′ (C+D) + BC′D′

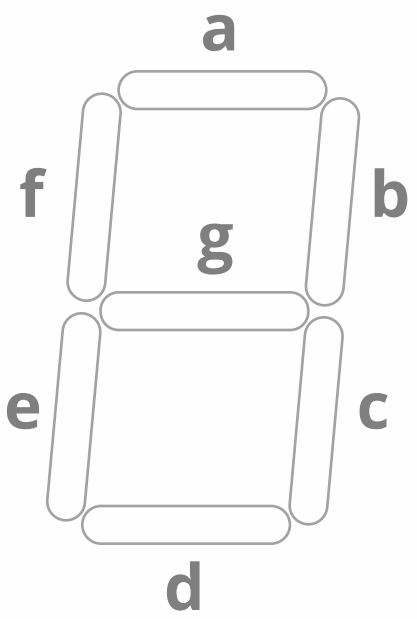
  x = B′ (C + D) + B (C+D)′
- y = CD + C′D′ = CD + (C + D)′
- z = D′



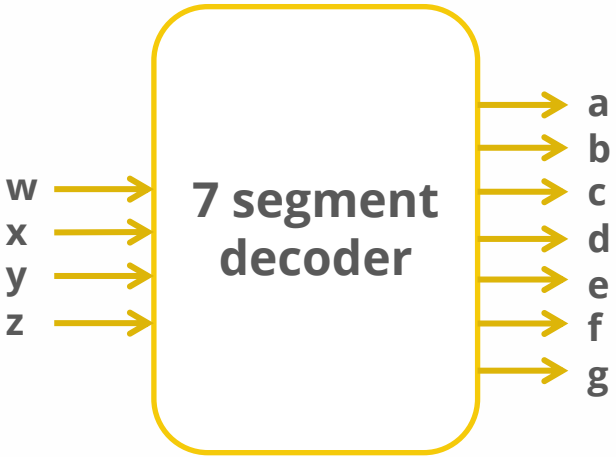
DESIGN PROCEDURE

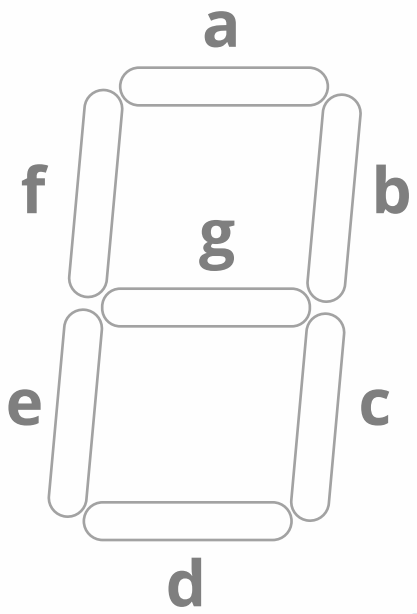
7 segment decoder

- a, b, c, d, e, f, g are the outputs.
- Outputs are in segments.
- Each segment is turned on based on the input number.
- Ex:
- If BCD 5 is entered, then segments (outputs) a, f, g, c, d should be turned on.



DESIGN PROCEDURE

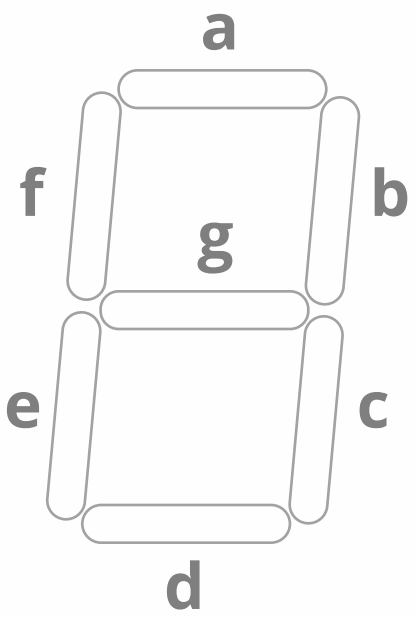






DESIGN PROCEDURE

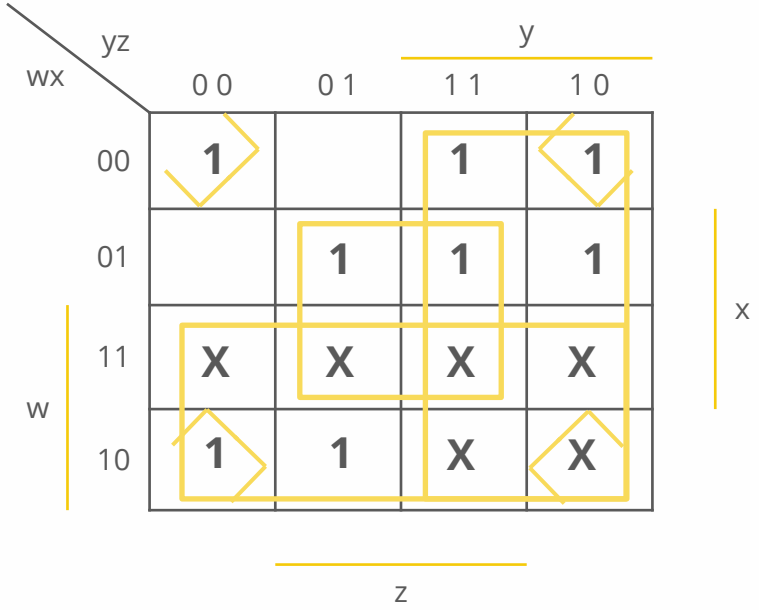
Input BCD				7 - segment							
w	x	y	z	a	b	c	d	e	f	g	
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	1	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	1	0	1	1	9
1	0	1	0	X	X	X	X	X	X	X	
1	0	1	1	X	X	X	X	X	X	X	
1	1	0	0	X	X	X	X	X	X	X	
1	1	0	1	X	X	X	X	X	X	X	
1	1	1	0	X	X	X	X	X	X	X	
1	1	1	1	X	X	X	X	X	X	X	



DESIGN PROCEDURE

Input BCD				7 - segment						
w	x	y	z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X

a's karnaugh map

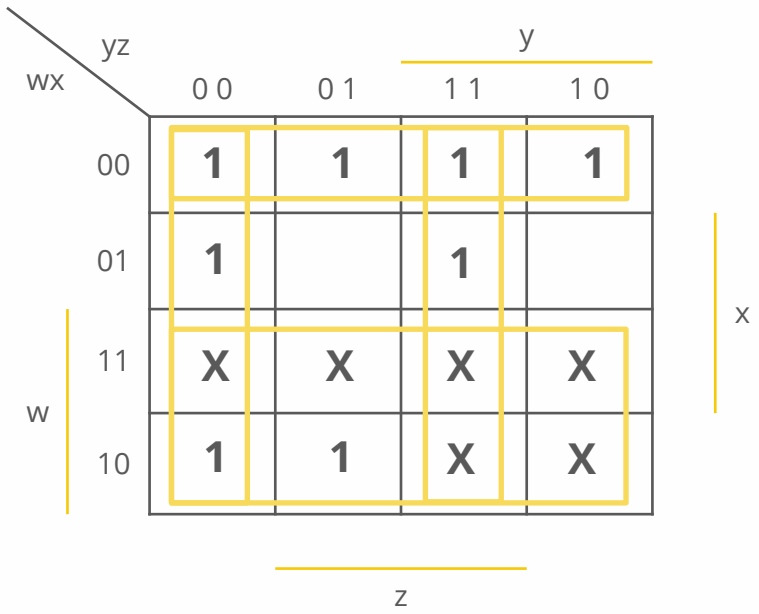


$$a = w + y + x\,z + x'\,z'$$

DESIGN PROCEDURE

Input BCD				7 - segment						
w	x	y	z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X

b's karnaugh map

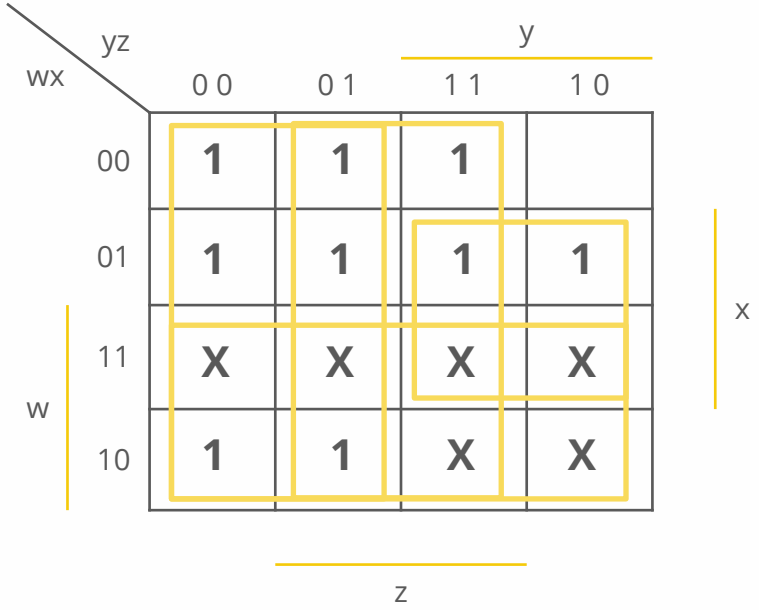


$$b = w + w'x' + y\,z + y'\,z'$$

DESIGN PROCEDURE

Input BCD				7 - segment						
w	x	y	z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X

c's karnaugh map

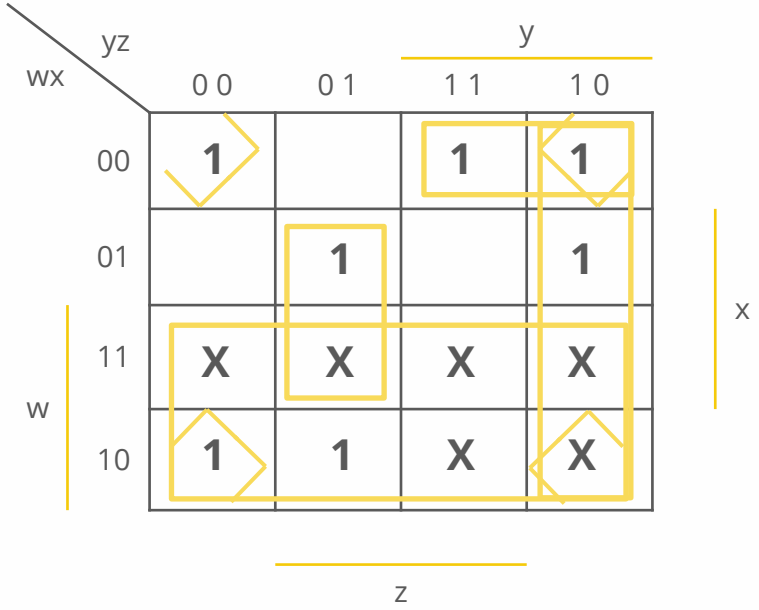


$$c = w + xy + x'z + x'y'$$

DESIGN PROCEDURE

Input BCD				7 - segment						
w	x	y	z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X

d's karnaugh map

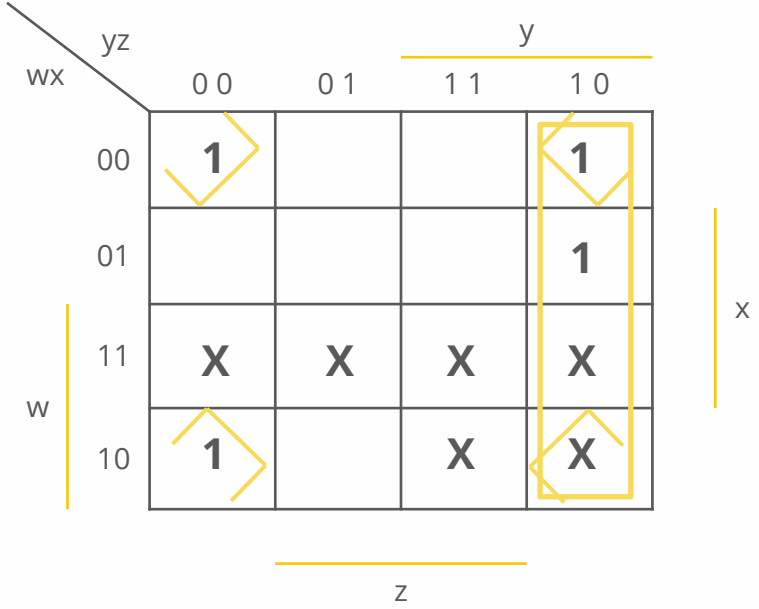


$$d = w + yz' + w'x'y + xy'z + x'z'$$

DESIGN PROCEDURE

Input BCD				7 - segment						
w	x	y	z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X

e's karnaugh map

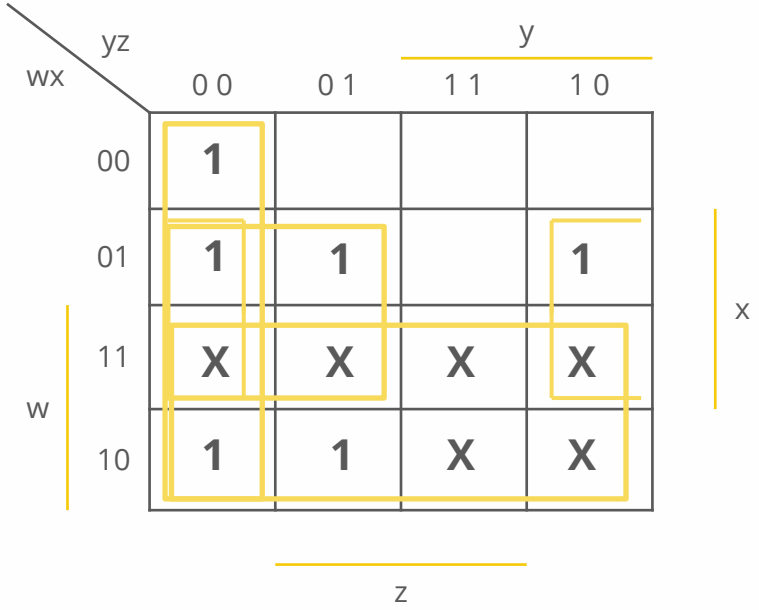


e = yz′ + x′z′

DESIGN PROCEDURE

Input BCD				7 - segment						
w	x	y	z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X

f's karnaugh map

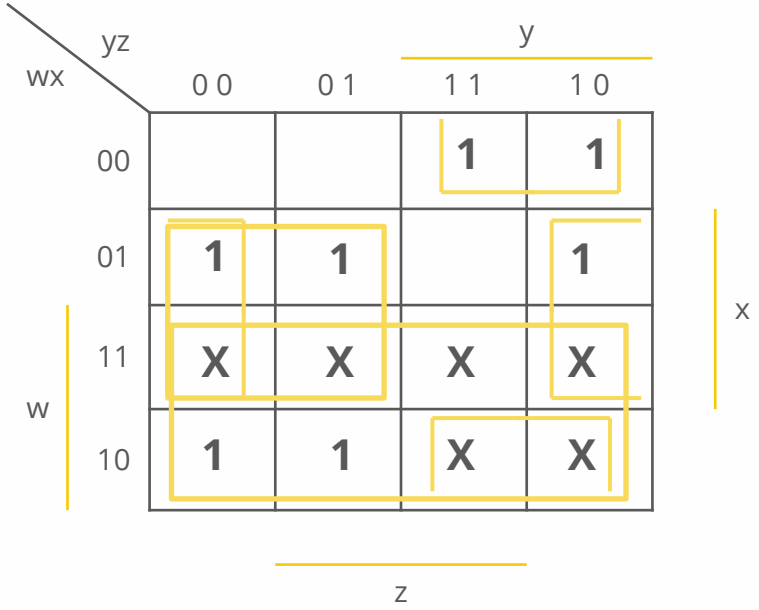


$$f = w + xy' + x z' + y'z'$$

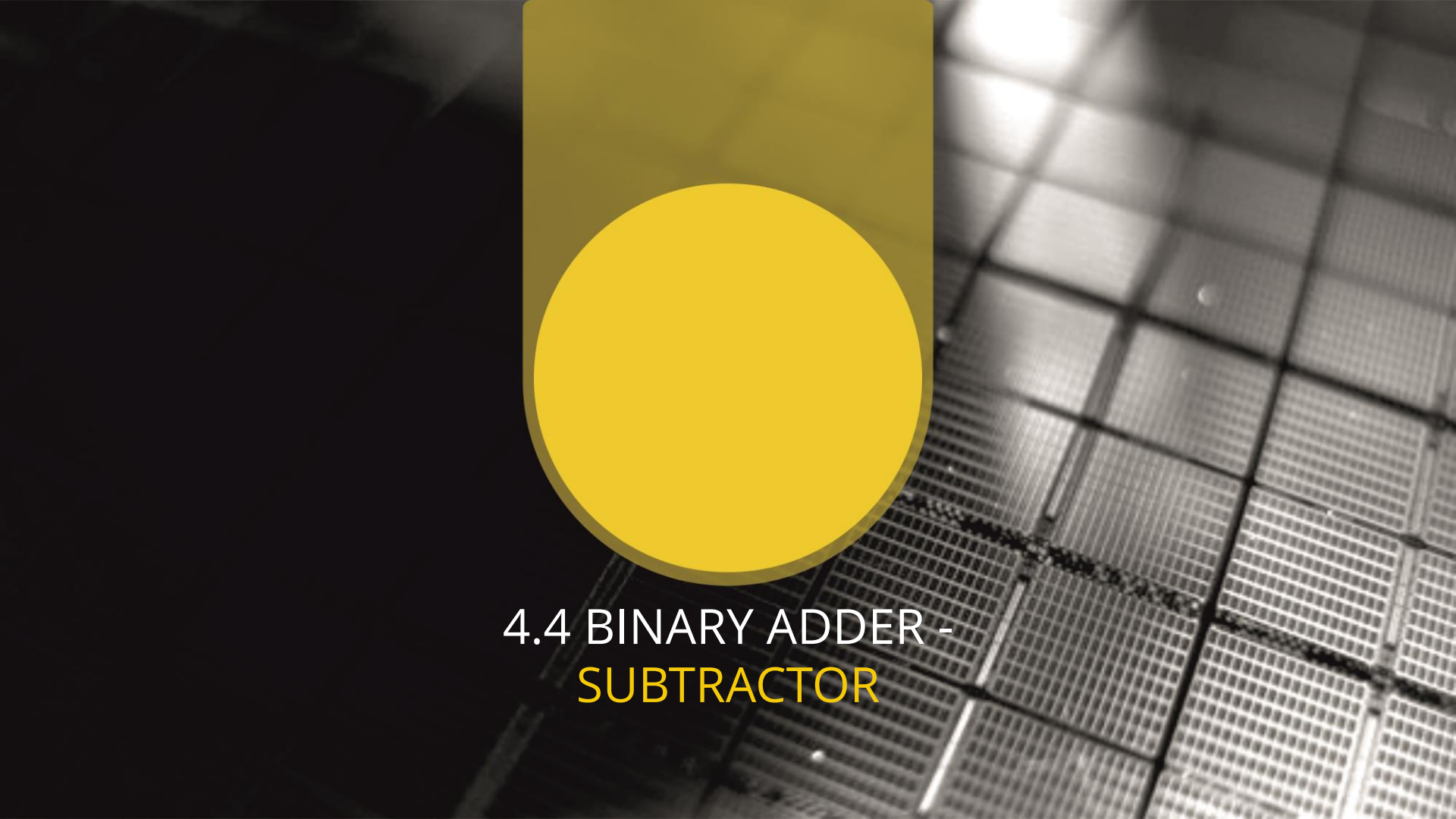
DESIGN PROCEDURE

Input BCD				7 - segment						
w	x	y	z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X

g's karnaugh map



$$g = w + xy' + x\,z' + x'y$$



4.4 BINARY ADDER - SUBTRACTOR

BINARY ADDER - SUBTRACTOR

- The most basic arithmetic operation is the addition of two binary digits.
- Simple addition consists of 4 possible elementary operations:
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = 10
- First 3 operations produce a sum of one digit.
- In the 4^{th} operation, the binary sum consists of two digits.

BINARY ADDER - SUBTRACTOR

- The higher significant bit is called a **carry**.
- A combinational circuit that performs the addition of two bits is called a **half adder**.
- One that performs the addition of three bits (two significant bits and a previous carry) is a **full adder**.
- Two half adders can be employed to implement a full adder.

BINARY ADDER - SUBTRACTOR

Half Adder

- Adds 2 bits
  - 2 inputs
  - 2 outputs
- Produces **SUM and CARRY.**



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

S = x'y + xy'

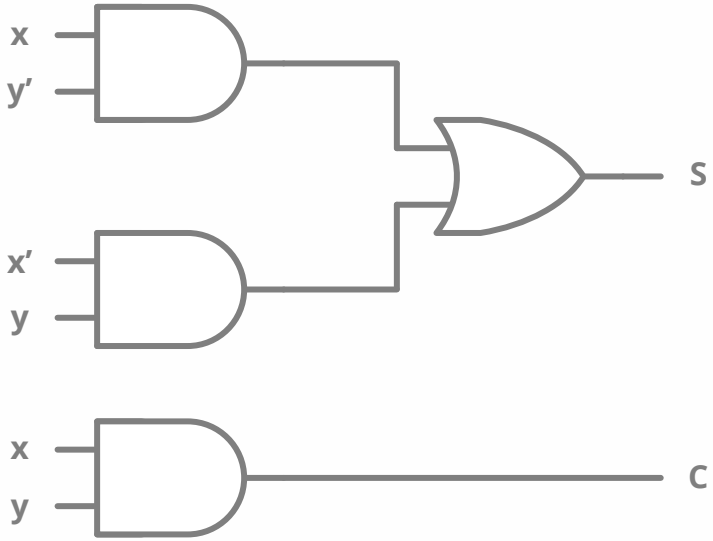
C = xy

BINARY ADDER - SUBTRACTOR

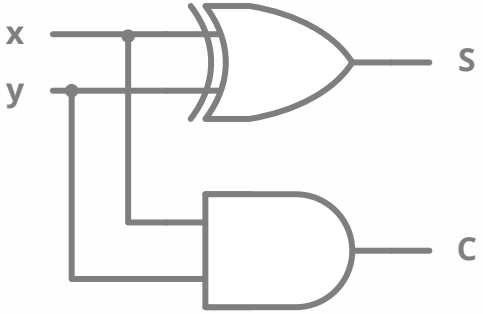
Half Adder

S = x'y + xy'

C = xy



(a) S = xy'+x'y, C = xy

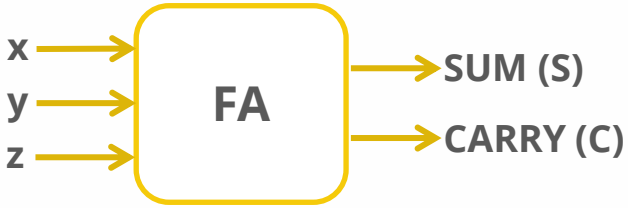


(b) S = x⊕y, C = xy

BINARY ADDER - SUBTRACTOR

Full Adder

- Adds 3 bits
  - 3 inputs
  - 2 outputs
- Produces **SUM and CARRY.**



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

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

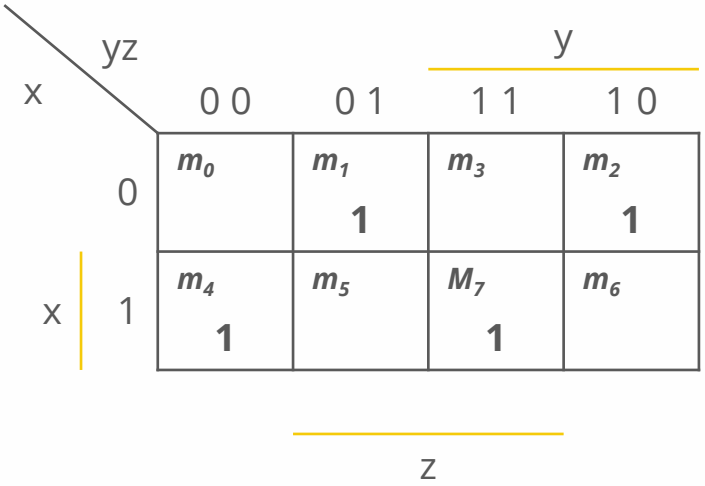
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

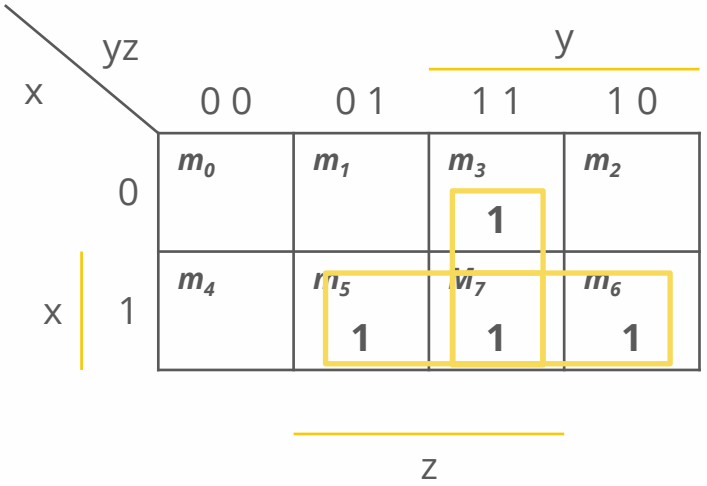
BINARY ADDER - SUBTRACTOR

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

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

$S = x \oplus y \oplus z$



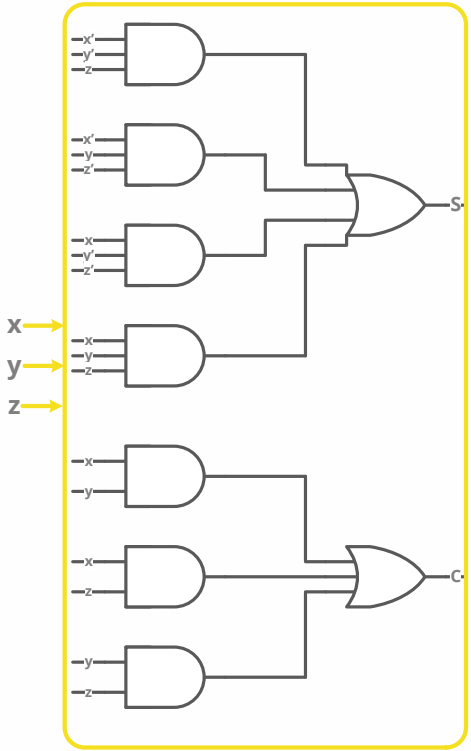


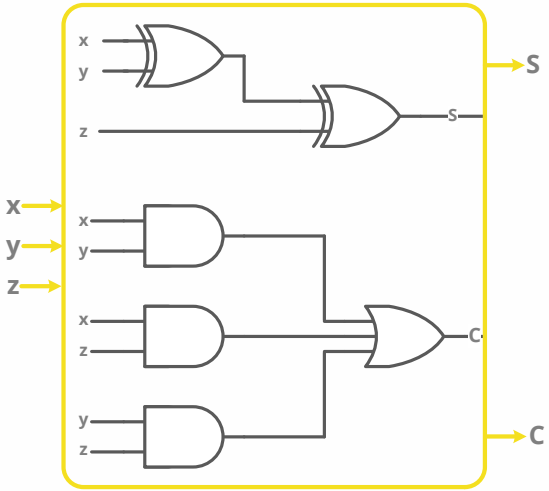
$C = xy + xz + yz$

BINARY ADDER - SUBTRACTOR

Full adder implementation:

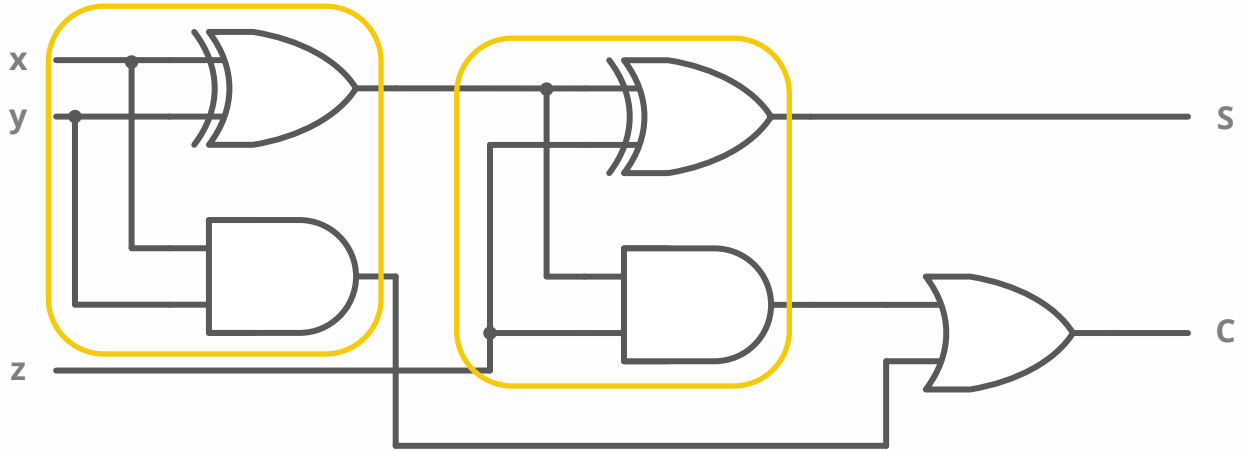
- S = x'y'z + x'yz + xy'z' + xyz
- S = x ⊕ y ⊕ z
- C = x'yz + xy'z + xyz' + xyz





BINARY ADDER - SUBTRACTOR

- Implementation of Full Adder with Two Half Adders and an OR gate

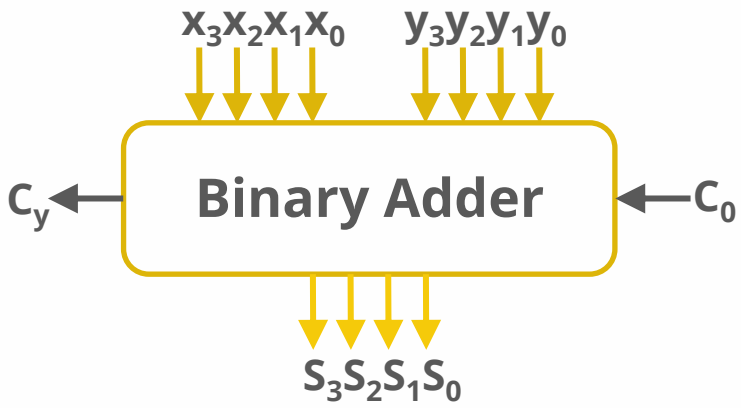


BINARY ADDER - SUBTRACTOR

Binary Adder:

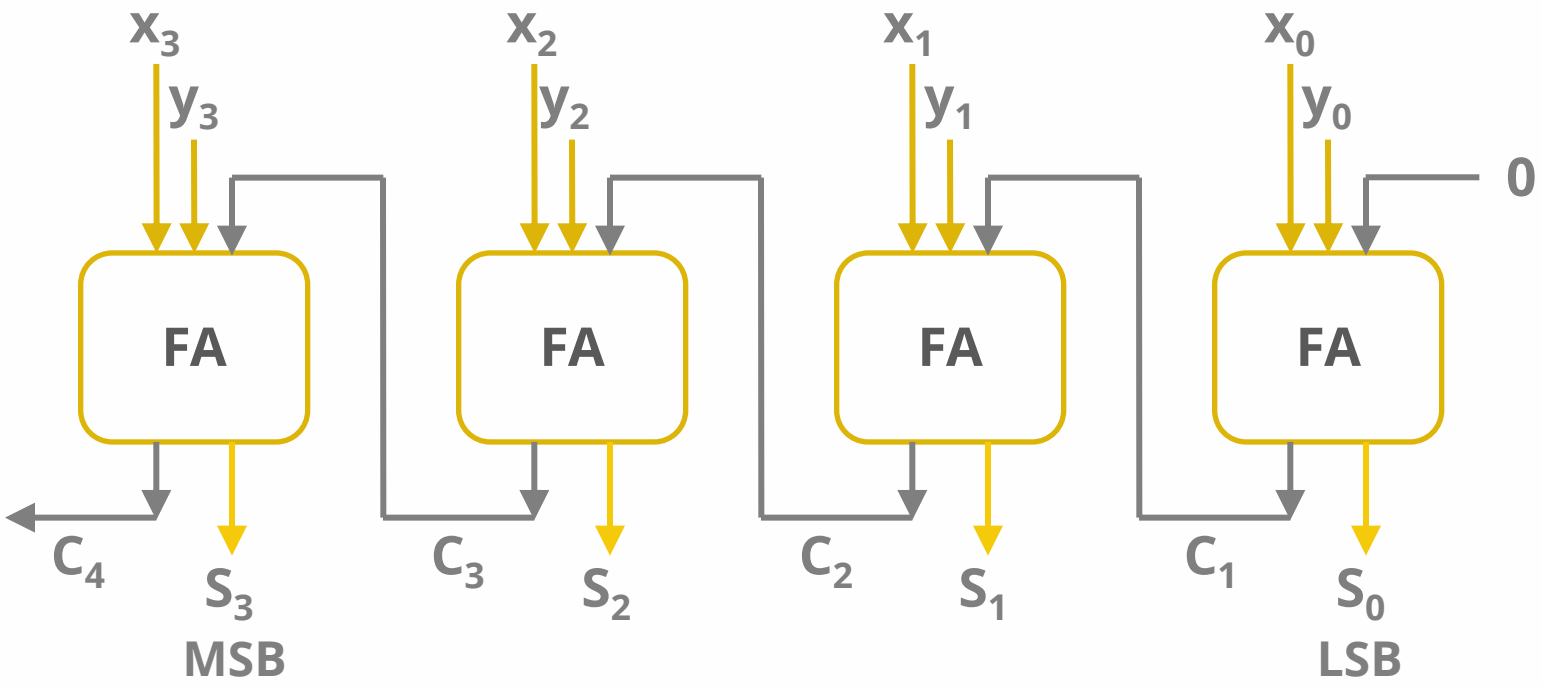
- Digital circuit that produces the arithmetic sum of two binary numbers.
- It can be constructed with full adders connected in cascade.
- The output carry from each full adder connected to the input carry of the next full adder in the chain.

$$
\begin{array}{rcccc}
 & c_3 & c_2 & c_1 & \\
+ & x_3 & x_2 & x_1 & x_0 \\
+ & y_3 & y_2 & y_1 & y_0 \\
\hline
Cy & S_3 & S_2 & S_1 & S_0
\end{array}
$$



BINARY ADDER - SUBTRACTOR

Binary Adder: 4-bit Ripple Carry Adder



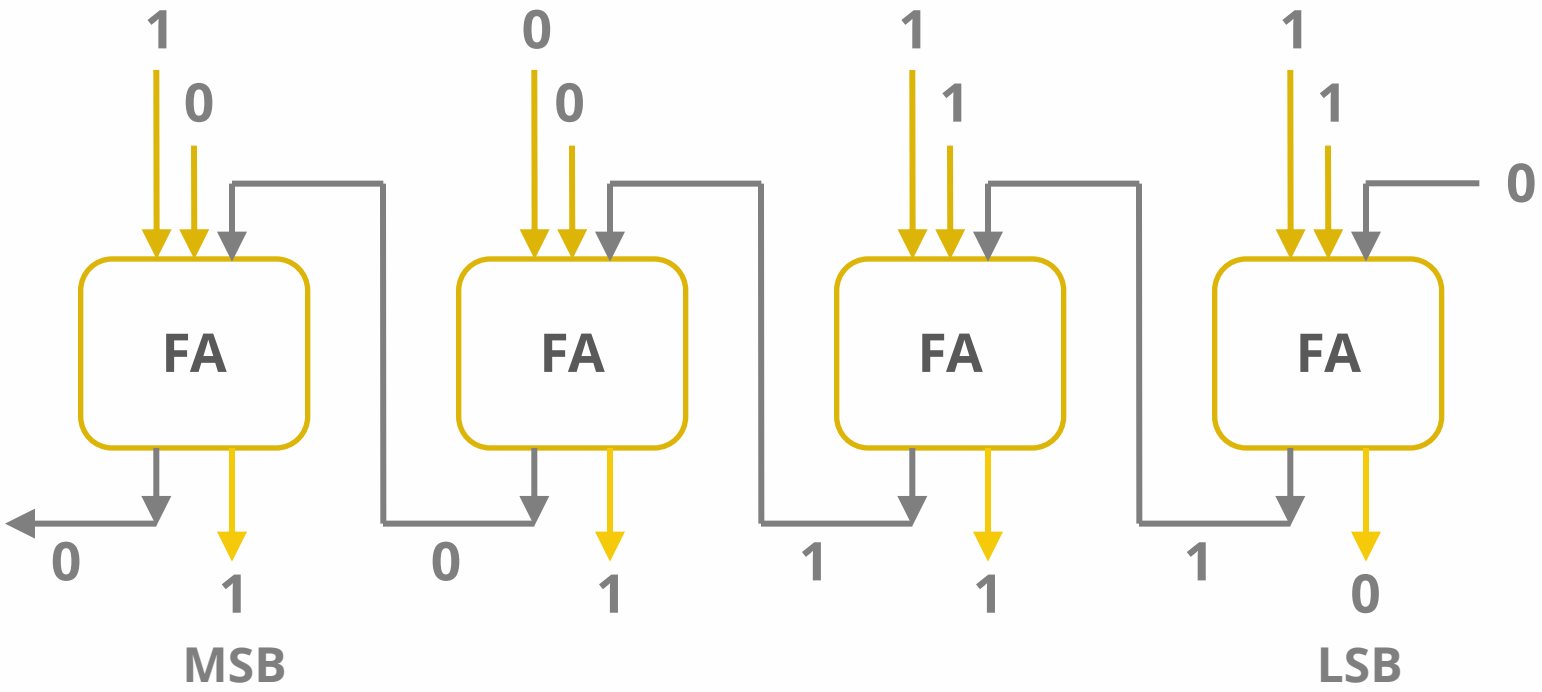
BINARY ADDER - SUBTRACTOR

- The **S** outputs generate the required sum bits.
- An **n-bit** adder requires **n** full adders with each output carry connected to the input carry of the next higher-order full adder.
- Example: A = 1011 and B = 0011.

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

BINARY ADDER - SUBTRACTOR

Binary Adder: 4-bit Ripple Carry Adder



BINARY ADDER - SUBTRACTOR

Carry propagation

- The addition of two binary numbers in parallel implies that all the bits of the augend and addend are available at the same time.
- Signals must propagate through the gates before the correct output sum is available in the output terminals.
- The total propagation time
  - = propagation delay of a typical gate x the number of gate levels in the circuit.
- Longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders.

BINARY ADDER - SUBTRACTOR

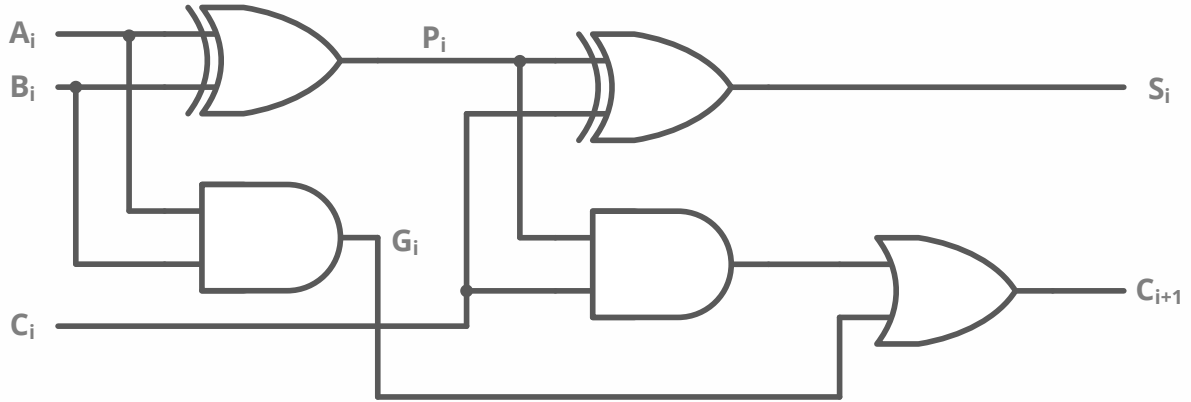
Carry propagation

- Since each bit of the sum output depends on the value of the input carry, the value of S_i in any given stage in the adder will in its steady state final value only after the input carry to that stage has been propagated.

BINARY ADDER - SUBTRACTOR

Carry propagation

- The number of gate levels for the carry propagation can be found from the circuit of the full adder.



BINARY ADDER - SUBTRACTOR

Carry propagation

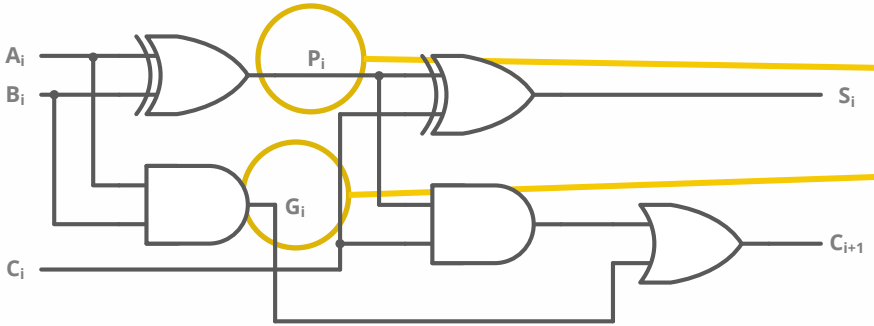
- The signal from the input carry C_i to the output carry C_{i+1}, propagates through
  - an AND gate and an OR gate,
  - which constitute two gate levels.
- If there are 4 full adders in the adder, the output carry C_4 would have
  - 2 x 4 = 8 gate levels from C_0 to C_4.
- For an **n-bit** adder, there are **2n** gate levels for the carry to propagate from input to output.

BINARY ADDER - SUBTRACTOR

- The carry propagation time is a limiting factor on the speed with which two numbers are added.
- Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is very critical.
- Reduce the carry propagation delay
  - Employ faster gates
  - Look-ahead carry (more complex mechanism, yet faster)

BINARY ADDER - SUBTRACTOR

- Consider



- Two new variables:
  - $P_i = A_i \oplus B_i$
  - $G_i = A_i \cdot B_i$
- Sum = $P_i \oplus C_i$
- Carry = $G_i + P_i \cdot C_i$

BINARY ADDER - SUBTRACTOR

- G_i is called **carry generate**
  - Produces a carry of 1 when both A_i and B_i are 1, regardless of the input carry.
- P_i . C_i is called a **carry propagate**
  - It is the term associated with the propagation of the carry from C_i to $C_i + 1$.

BINARY ADDER - SUBTRACTOR

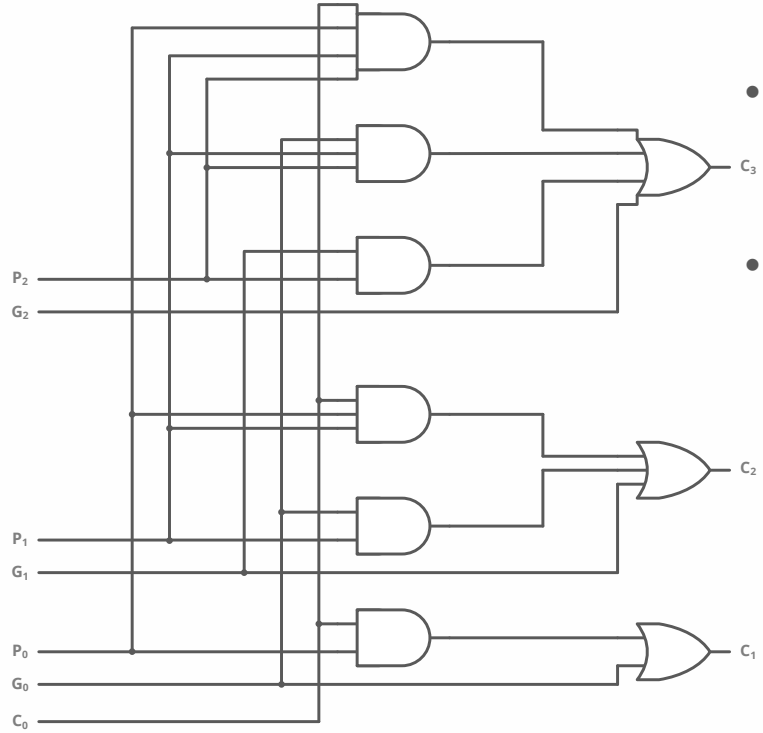
- Boolean functions for the carry outputs of each stage:
  - C_0 = input carry
  - $C_1 = \mathbf{G_0 + P_0 . C_0}$
  - $C_2 = G_1 + P_1 . C_1 = G_1 + P_1 (G_0 + P_0 . C_0)$

    $= \mathbf{G_1 + P_1 . G_0 + P_1 . P_0 . C_0}$
  - $C_3 = G_2 + P_2 . C_2 = G_2 + P_2 (G_1 + P_1 . G_0 + P_1 . P_0 . C_O)$

    $= \mathbf{G_2 + P_2 . G_1 + P_2 . P_1 . G_0 + P_2 . P_1 . P_0 . C_0}$

BINARY ADDER - SUBTRACTOR

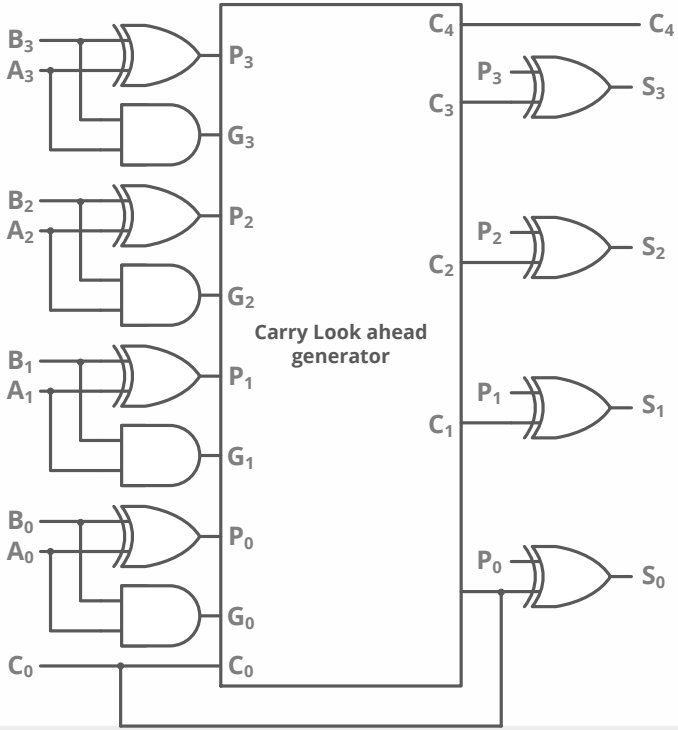
- Carry Lookahead Generator



- C_3 does not have to wait for C_2 and C_1 to propagate.
- C_3 is propagated at the same time as C_1 and C_2.

BINARY ADDER - SUBTRACTOR

- 4-Bit Adder with Carry Lookahead Generator



- Each sum output requires two XOR gates.
  - First XOR gate generates the P_i
  - AND gate generates the Gi variable.
  - Carries are propagated through CLA and applied as an input to the second XOR gate.
  - All output carries are generated after a delay though two levels of gates. S_1 through S_3 have equal propagating delay times.

BINARY ADDER - SUBTRACTOR

Binary Subtactor

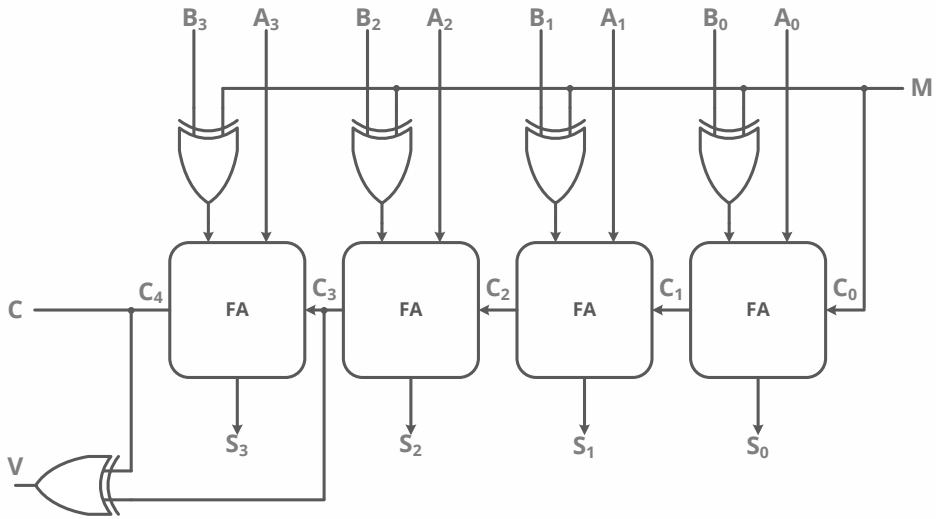
- The subtraction of unsigned binary numbers can be done most conveniently by means of complements.
- A – B can be performed by taking 2’s complement of B and adding it to A.
- 2’s complement can be done by taking 1’s complement and adding 1.
- 1’s complement can be implemented with inverters.

BINARY ADDER - SUBTRACTOR

Binary Subtactor

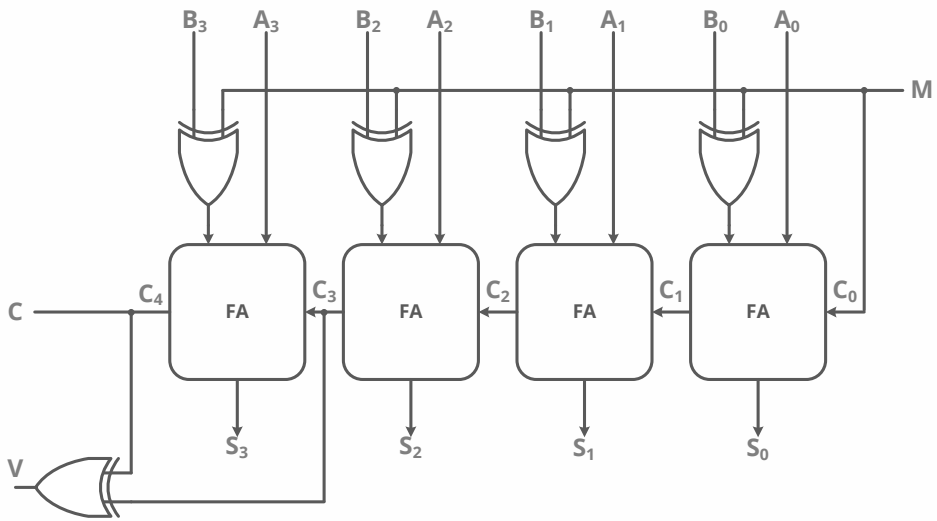
- Circuit of subtractor consist of:
  - An adder
  - Inverters
- The input carry $C_0 = 1$ when performing subtraction
- The operation performed becomes:
  - A + 1's complement of B + 1 = A + 2's complement of B.
  - **A – B = A + (-B) = A +B' + 1.**
- For unsigned numbers,
  - A – B if A $\geq$ B or 2's complement of (B – A) if A < B.

BINARY ADDER - SUBTRACTOR



- XOR gate combines addition and subtraction into one circuit with common binary adder.
- Mode input **M** controls the operation.
  - **M = 0 = Addition.**
  - **M = 1 = Subtraction.**

BINARY ADDER - SUBTRACTOR



XOR with output V is for detecting an overflow.

- When **M = 0** (Addition).
  - **B ⊕ 0 = B.**
  - **Full adders receive the value of B**
  - **C_0 = 0.**
  - **Circuit performs A + B.**
- When **M = 1** (Subtraction).
  - **B ⊕ 0 = B'.**
  - **C_0 = 1.**
  - **Circuit performs A + B' + 1**

BINARY ADDER - SUBTRACTOR

Overflow

	1	n=1	1-bit register
+	1	n=1	1-bit register
1	0	n=2	2-bit register

- There is a need for an overflow detection.
- The detection of an overflow after addition of two binary numbers depends on whether the number is to be:
  - Signed
  - Unsigned

BINARY ADDER - SUBTRACTOR

Overflow

- Signed
  - Left most bit represents the sign bit.
  - Negative numbers are in 2’s complement format.
  - Sign bit treated as part of the number and the end carry does not indicate an overflow.
- Unsigned

BINARY ADDER - SUBTRACTOR

Overflow

- Unsigned
  - When added, overflow is detected from the end carry out of the most significant position.
- Overflow cannot occur if positive and negative numbers are added.
- Overflow occurs if both numbers are positive or negative.

BINARY ADDER - SUBTRACTOR

Overflow

- Ex: +70 , +80 in 8-bit registers.
- 8-bit register = 2^7 = 256.
  - (-128) to (+127)

		0	**1**	**0**	**0**	**0**	**0**	**0**	**0**
	+70		**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**
+	+80		**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**
	+150		**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**

BINARY ADDER - SUBTRACTOR

Overflow

		0	1	0	0	0	0	0	0
	+70		0	1	0	0	0	1	1	0
+	+80		0	1	0	1	0	0	0	0
	+150		1	0	0	1	0	1	1	0

- 8-bit result that should have been positive has a negative sign bit and vice versa.
- If carry out of sign bit is taken as the sign bit of the result then the 8-bit answer will be correct.
- Since it can not be accommodated within 8-bit register, then there is a overflow.

BINARY ADDER - SUBTRACTOR

- Overflow can be detected by:
  - Observing the carry into the sign bit.
  - Observing the carry out of the sign bit.
    - If they are not equal, an overflow has occurred.
- If two carries are applied to an XOR gate overflow is detected.
    - When the output is 1.
  - 2's complement must be computed for this method to work correctly.
    - This take care of the condition when the maximum negative numbers is complemented.

BINARY ADDER - SUBTRACTOR

- If two binary numbers are unsigned
  - the C bit detects
    - a carry after addition or
    - a borrow after subtraction.
- If the numbers are signed, then the V bit detects an overflow.
  - If V = 0, then no overflow.
    - The n-bit result is correct.
  - If V = 1, then result contains n+1 bits only the right most n-bits of the number.
  - Overflow has occurred. The (n+1)th bit is the actual sign and has been shifted out of position.

4.5 DECIMAL ADDER

DECIMAL ADDER

- BCD Adder
- Consider the arithmetic addition of two decimal digits in BCD and an input carry.
- In BCD each input digit does not exceed 9
- Output sum cannot be greater than 9 + 9 + 1 = 19.
  - 1 is an input carry.

DECIMAL ADDER

- Suppose:
  - Two BCD digits applied to a 4-bit binary adder.

$$
\begin{array}{rcccc}
 & c_3 & c_2 & c_1 & \\
+ & x_3 & x_2 & x_1 & x_0 \\
+ & y_3 & y_2 & y_1 & y_0 \\
\hline
Cy & S_3 & S_2 & S_1 & S_0
\end{array}
$$

  - The output produces a result that ranges from 0 through 19.

DECIMAL ADDER

X +Y	x_3 x_2 x_1 x_0	y_3 y_2 y_1 y_0	Sum	Cy	S_3 S_2 S_1 S_0
0 + 0	0 0 0 0	0 0 0 0	= 0	0	0 0 0 0
0 + 1	0 0 0 0	0 0 0 1	= 1	0	0 0 0 1
0 + 2	0 0 0 0	0 0 1 0	= 2	0	0 0 1 0
0 + 9	0 0 0 0	1 0 0 1	= 9	0	1 0 0 1
1 + 0	0 0 0 1	0 0 0 0	= 1	0	0 0 0 1
1 + 1	0 0 0 1	0 0 0 1	= 2	0	0 0 1 0
1 + 8	0 0 0 1	1 0 0 0	= 9	0	1 0 0 1
1 + 9	0 0 0 1	1 0 0 1	= A	0	1 0 1 0
2 + 0	0 0 1 0	0 0 0 0	= 2	0	0 0 1 0
9 + 9	1 0 0 1	1 0 0 1	= 12	1	0 0 1 0

Invalid Code

Wrong BCD Value

0001 1000

DECIMAL ADDER

X +Y	$x_3\ x_2\ x_1\ x_0$	$y_3\ y_2\ y_1\ y_0$	Sum	Cy	$S_3\ S_2\ S_1\ S_0$	Required BCD Output	Value	
9 + 0	1 0 0 1	0 0 0 0	= 9	0	1 0 0 1	0 0 0 0 1 0 0 1	= 9	
9 + 1	1 0 0 1	0 0 0 1	= 10	0	1 0 1 0	0 0 0 1 0 0 0 0	= 16	✘
9 + 2	1 0 0 1	0 0 1 0	= 11	0	1 0 1 1	0 0 0 1 0 0 0 1	= 17	✘
9 + 3	1 0 0 1	0 0 1 1	= 12	0	1 1 0 0	0 0 0 1 0 0 1 0	= 18	✘
9 + 4	1 0 0 1	0 1 0 0	= 13	0	1 1 0 1	0 0 0 1 0 0 1 1	= 19	✘
9 + 5	1 0 0 1	0 1 0 1	= 14	0	1 1 1 0	0 0 0 1 0 1 0 0	= 20	✘
9 + 6	1 0 0 1	0 1 1 0	= 15	0	1 1 1 1	0 0 0 1 0 1 0 1	= 21	✘
9 + 7	1 0 0 1	0 1 1 1	= 16	1	0 0 0 0	0 0 0 1 0 1 1 0	= 22	✘
9 + 8	1 0 0 1	1 0 0 0	= 17	1	0 0 0 1	0 0 0 1 0 1 1 1	= 23	✘
9 + 9	1 0 0 1	1 0 0 1	= 18	1	0 0 1 0	0 0 0 1 1 0 0 0	= 24	



DECIMAL ADDER

- The problem is to find a rule by which the binary sum is to be converted to the correct BCD digit representation of the number in the BCD sum.

DECIMAL ADDER

Binary Sum					BCD Sum					Decimal
K	Z3	Z2	Z1	Z0	C	S3	S2	S1	S0	
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

DECIMAL ADDER

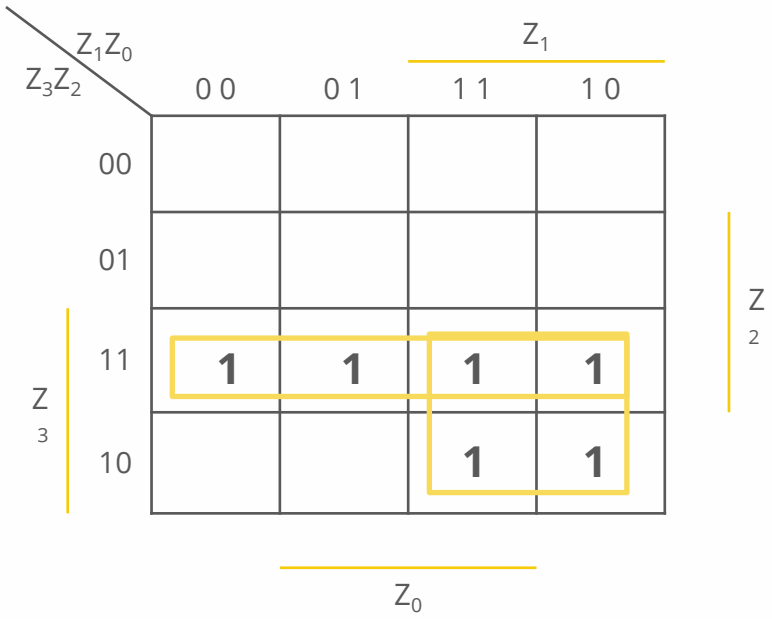
Binary Sum						BCD Sum				Decimal
K	Z3	Z2	Z1	Z0	C	S3	S2	S1	S0	
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

DECIMAL ADDER

- In examining the contents of the table;
  - When the binary SUM = 1001, the corresponding BCD number is identical.
  - When the binary SUM >1001, the BCD number is invalid.
  - The addition of 6 (0110) is required.
  - Correction is needed when K = 1.
  - Correction is needed from 1010 – 1111.
    - Z4 = 1,
    - Z3 and Z2 must = 1 to distinguish from 1000 and 1001

DECIMAL ADDER

Z_3	Z_2	Z_1	Z_0	Err
0	0	0	0	0
.				.
.				.
.				.
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

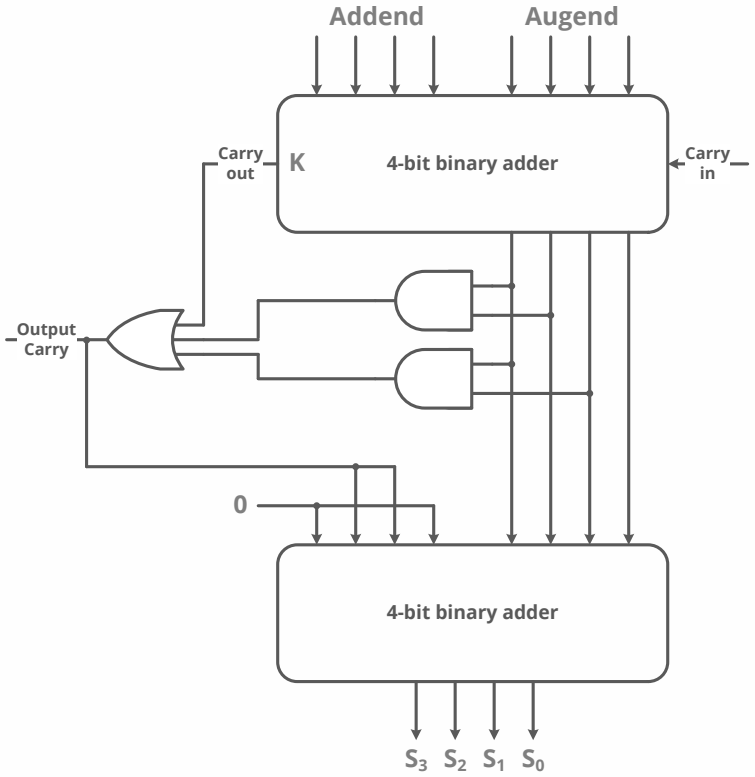


$$Err = Z_3Z_2 + Z_3Z_1$$

$$Output\ Carry = K + Z_3Z_2 + Z_3Z_1$$

DECIMAL ADDER

- When Output carry = 0,
  - Nothing is added.
- When Output carry = 1,
  - add 0110 to the binary sum.
  - provide an output carry for the next stage.



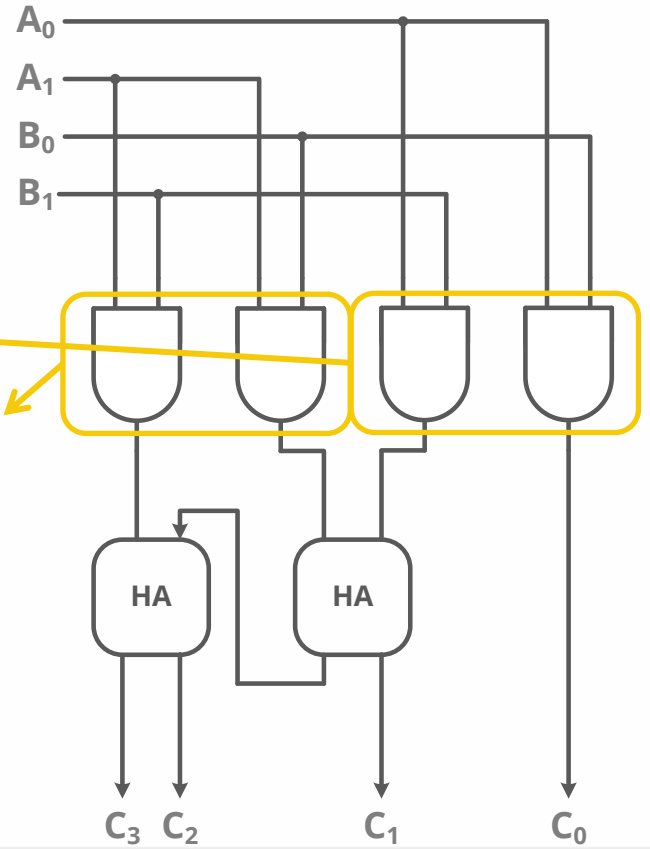


4.6 BINARY MULTIPLIER

BINARY MULTIPLIER

- Consider; 2-bit number

$$
\begin{array}{rcl}
 & B_1 \; B_0 & \leftarrow \text{Multiplicand bits} \\
\times & A_1 \; A_0 & \leftarrow \text{Multiplier bits} \\
\hline
 & A_0B_1 \; A_0B_0 & \leftarrow \text{First partial product (AND gate)} \\
+ & A_1B_1 \; A_1B_0 & \leftarrow \text{Second partial product (AND gate)} \\
\hline
 & C3 \; C2 \; C1 \; C0 & \leftarrow \text{Product}
\end{array}
$$



BINARY MULTIPLIER

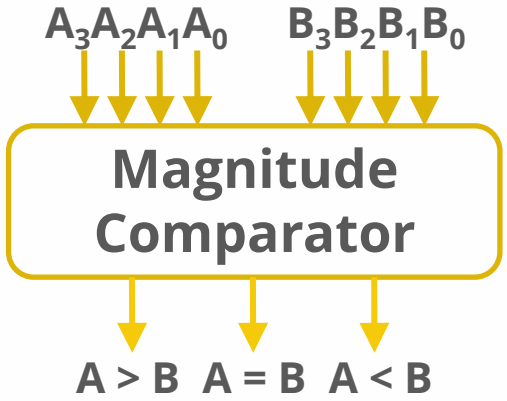
- For more bits;
  - A bit of multiplier is ANDed with each bit of multiplicand in as many levels as there are bits.
  - The binary output of AND gate in each level is added with the partial product of previous level.
  - For J multiplier bits and K multiplicand bits we need
    - J x K AND gates.



4.7 MAGNITUDE COMPARATOR

MAGNITUDE COMPARATOR

- Compare two numbers (A and B)
- 3 outputs <, =, >
  - A > B, A = B, A<B
- Consider compare 4-bit number to 4-bit number
  - A = A_3 A_2 A_1 A_0
  - B = B_3 B_2 B_1 B_0



MAGNITUDE COMPARATOR

- A = B if:
  - $A_3 = B_3$ **AND** $A_2 = B_2$ **AND** $A_1 = B_1$ **AND** $A_0 = B_0$

A_i	B_i	x_i (A = B)
0	0	1
0	1	0
1	0	0
1	1	1

$$x_i = A_i' \cdot B_i' + A_i \cdot B_i$$

$$x = x_3 \cdot x_2 \cdot x_1 \cdot x_0 = (A = B)$$

$$x_3 = A_3' \cdot B_3' + A_3 \cdot B_3$$

$$x_2 = A_2' \cdot B_2' + A_2 \cdot B_2$$

$$x_1 = A_1' \cdot B_1' + A_1 \cdot B_1$$

$$x_0 = A_0' \cdot B_0' + A_0 \cdot B_0$$

MAGNITUDE COMPARATOR

- A > B if
  - $A_i = 1$ and $B_i = 0$

A_i	B_i	y_i (A > B)
0	0	0
0	1	0
1	0	1
1	1	0

$y_i = A_i \cdot B_i'$

$y = y_3 + y_2 + y_1 + x_0 = (A > B)$

$y_3 = A_3 \cdot B_3'$

$y_2 = y3 \cdot A_2 \cdot B_2'$

$y_1 = y3 \cdot y2 \cdot A_1 \cdot B_1'$

$y_0 = y3 \cdot y2 \cdot y1 \cdot A_0 \cdot B_0'$

MAGNITUDE COMPARATOR

- A < B if
  - $A_i = 0$ and $B_i = 1$

A_i	B_i	z_i (A < B)
0	0	0
0	1	1
1	0	0
1	1	0

$z_i = A_i' \cdot B_i$

$z = z_3 + z_2 + z_1 + z_0 = (A < B)$

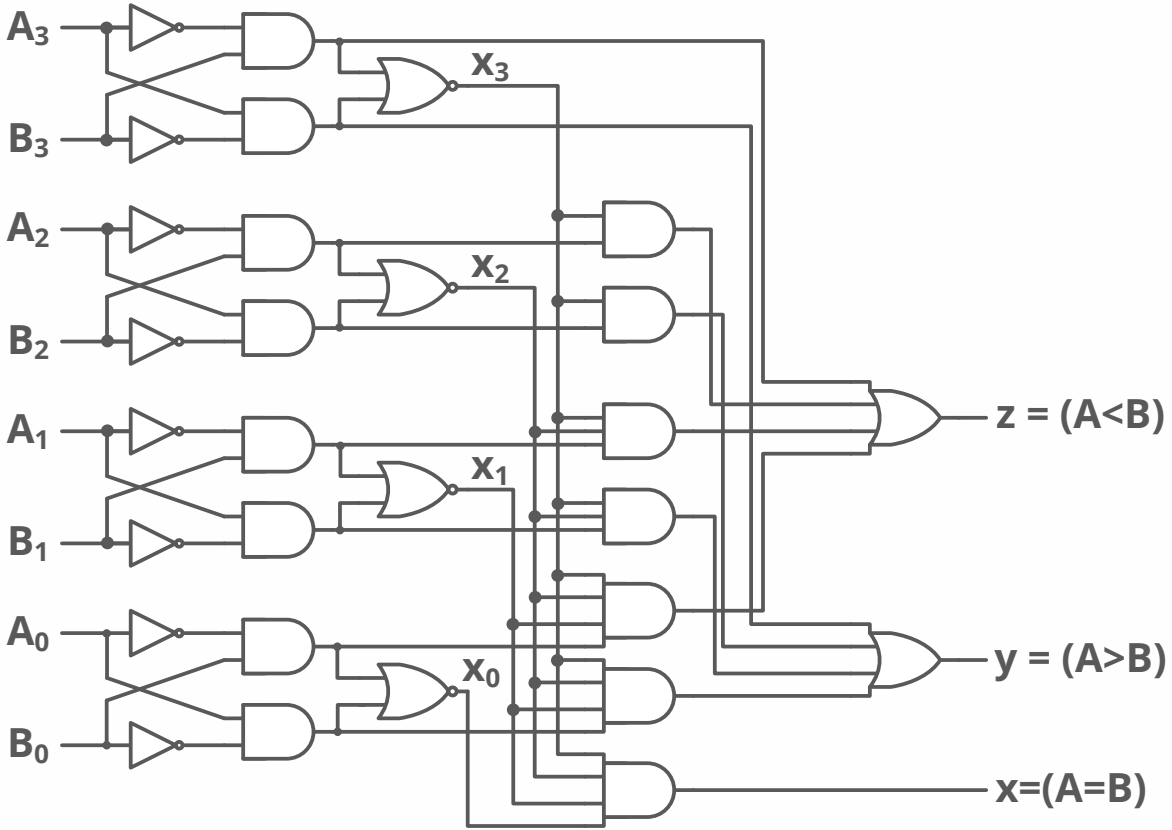
$z_3 = A_3' \cdot B_3$

$z_2 = y3 \cdot A_2' \cdot B_2$

$z_1 = y3 \cdot y2 \cdot A_1' \cdot B_1$

$z_0 = y3 \cdot y2 \cdot y1 \cdot A_0' \cdot B_0$

MAGNITUDE COMPARATOR





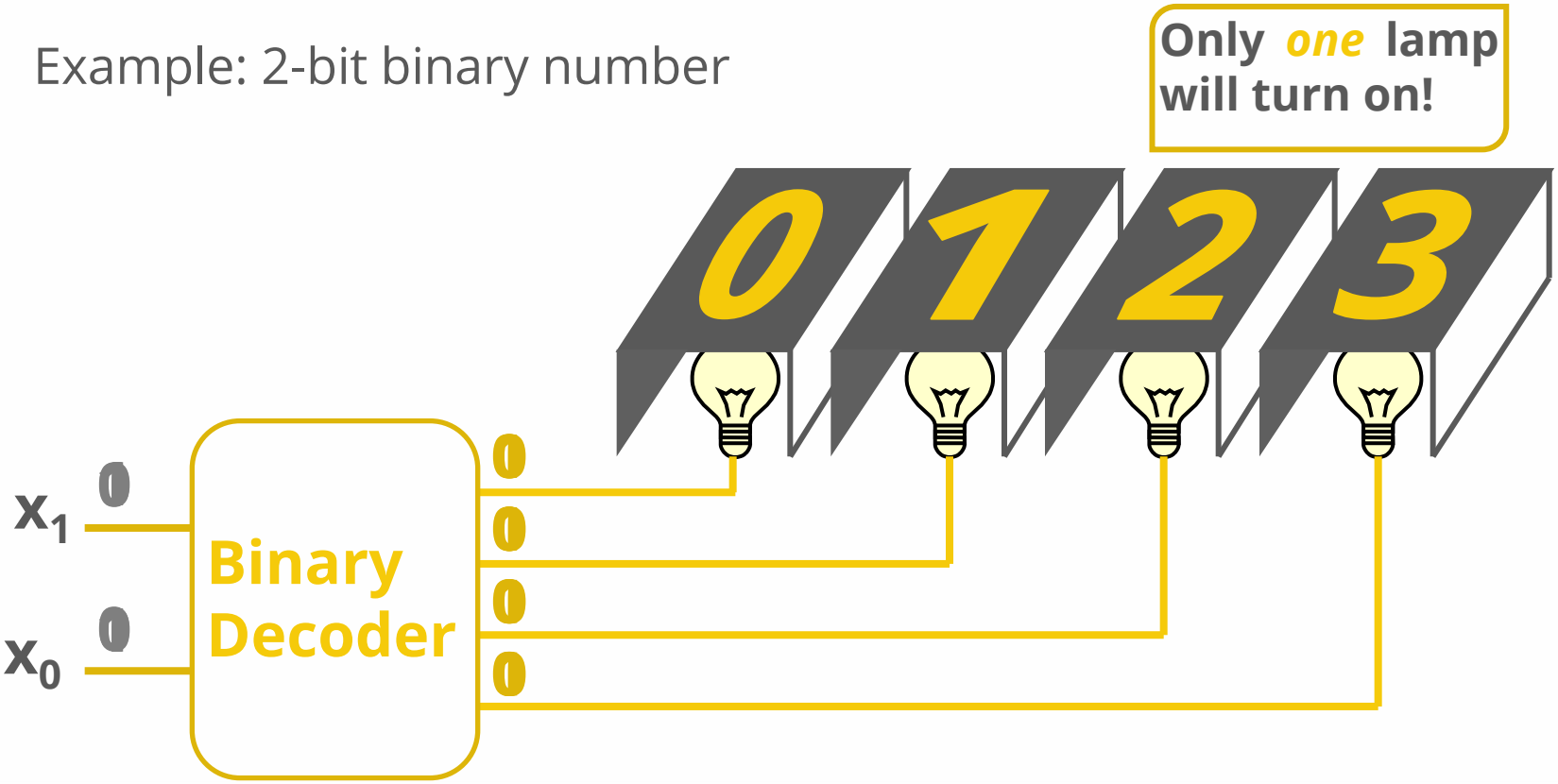
4.8 DECODERS

DECODERS

- Discrete quantities of information are represented in digital systems by binary codes.
- A binary code of **n** bits is capable of representing up to 2^n distinct elements of coded information.
- A **decoder** is a combinational circuit that converts binary information from **n** input lines to a maximum of 2^n unique output lines.

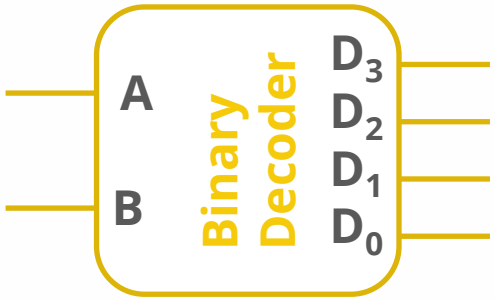
DECODERS

- Example: 2-bit binary number



DECODERS

- 2-to-4 Line Decoder



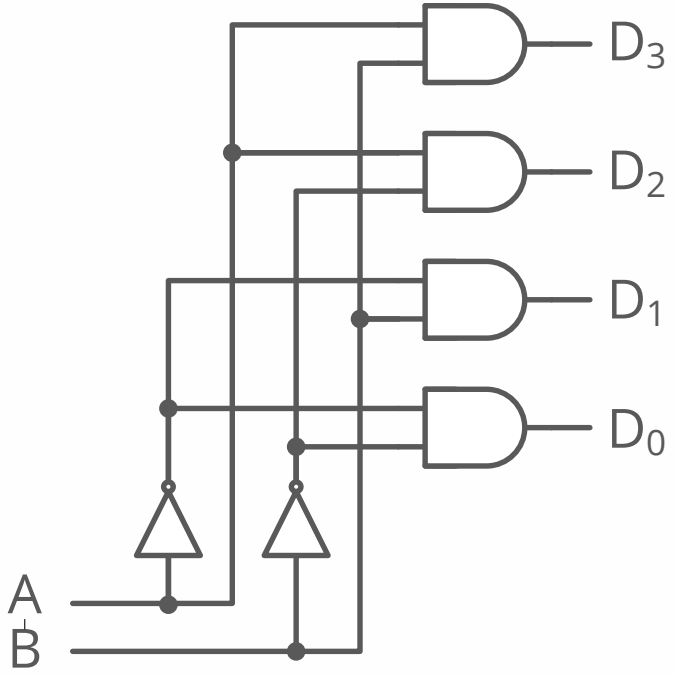
A	B	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

$D_3 = A'B'$

$D_2 = A'B$

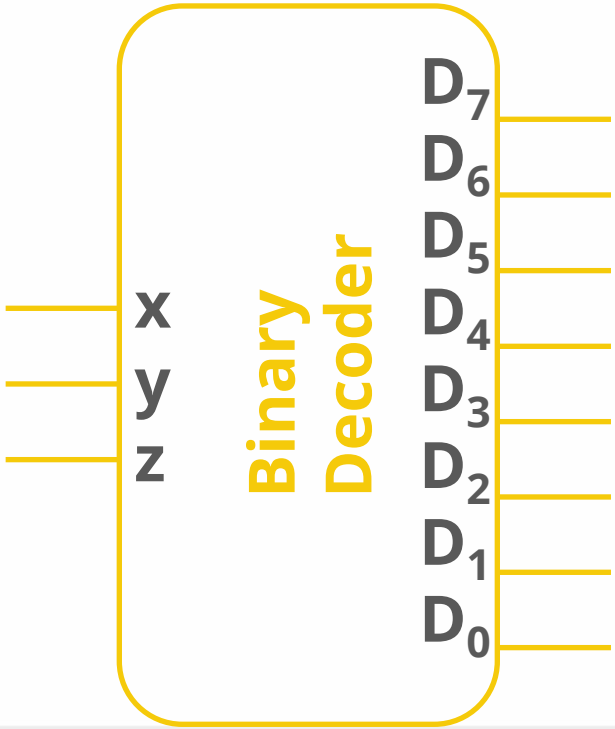
$D_1 = AB'$

$D_0 = AB$



DECODERS

- 3-to-8 Line Decoder (Binary to Octal Conversion)



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

DECODERS

- 3-to-8 Line Decoder (Binary to Octal Conversion)

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

$D_0 = x' y' z'$

$D_1 = x' y' z$

$D_2 = x' y z'$

$D_3 = x' y z$

$D_4 = x y' z'$

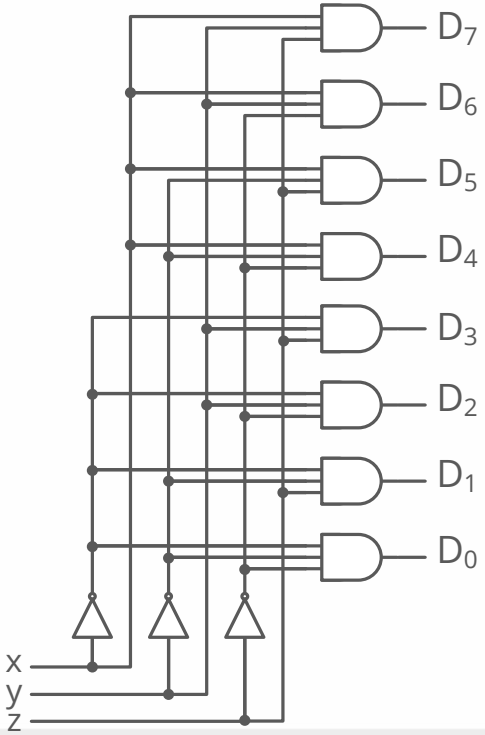
$D_5 = x y' z$

$D_6 = x y z'$

$D_7 = x y z$

DECODERS

- 3-to-8 Line Decoder (Binary to Octal Conversion)



$D_7 = x\ y\ z$

$D_6 = x\ y\ z'$

$D_5 = x\ y'\ z$

$D_4 = x\ y'\ z'$

$D_3 = x'\ y\ z$

$D_2 = x'\ y\ z'$

$D_1 = x'\ y'\ z$

$D_0 = x'\ y'\ z'$

DECODERS

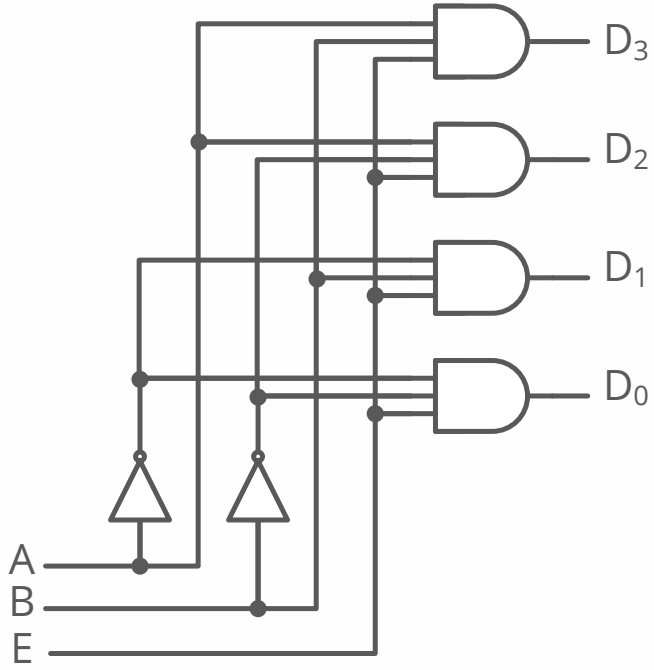
- Some decoders are with NAND gates.
- NAND becomes more economical to generate the decoder minterms in their complemented form.
- Decoders include one or more **ENABLE** inputs to control the circuit operation.

DECODERS

- 2-to-4 Line Decoder with **ENABLE** input



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



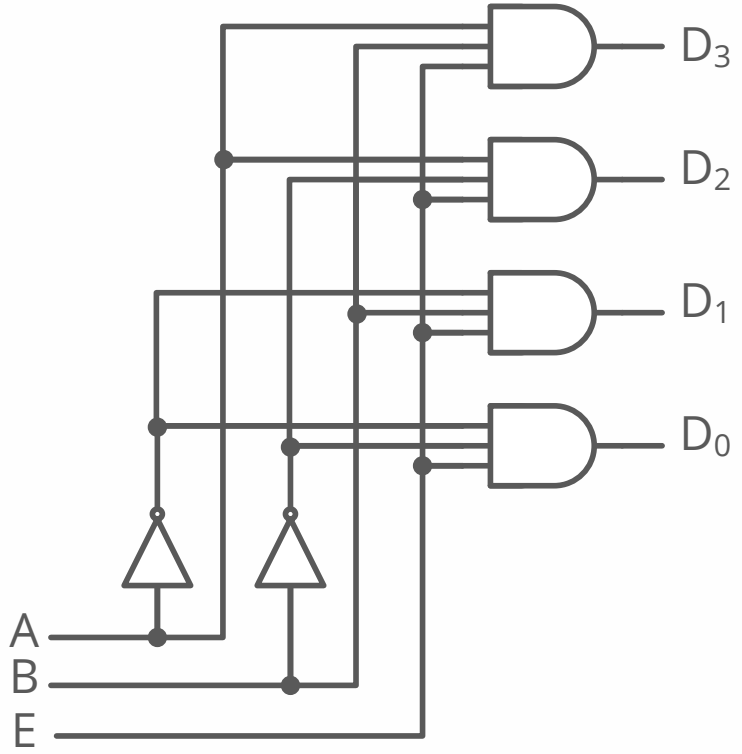
DECODERS

- 2-to-4 line decoder with **ENABLE** input can function as a **demultiplexer**.
- A **demultiplexer** is a circuit that receives information from single line and directs it to one of 2^n possible output lines.
- The selection of a specific output is controlled by the bit combination of n selected lines.

DECODERS

- The decoder can function as 1-to-4 line demultiplexer.
  - E is taken as a data input line.
  - A and B are taken as the selection inputs.

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

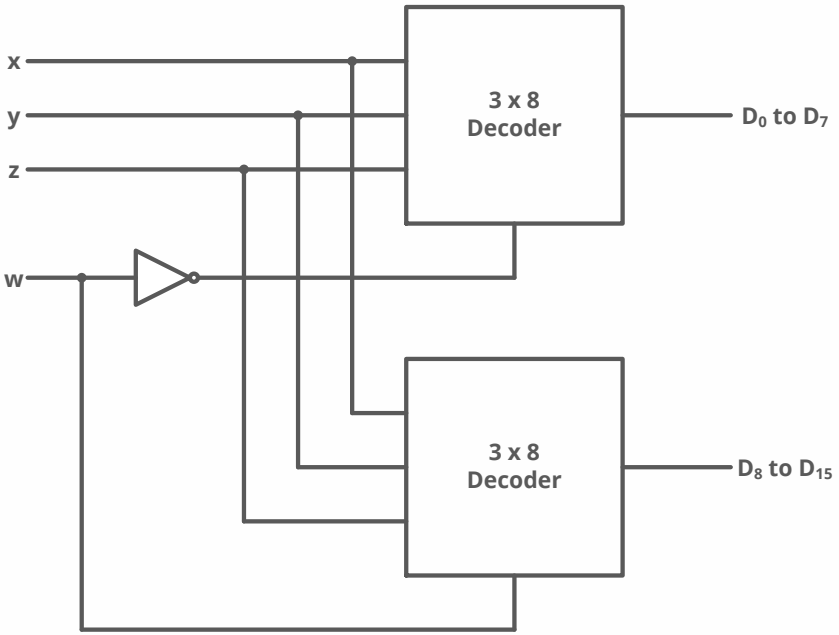


DECODERS

- Decoders with enable inputs can be connected together to form a larger decoder circuit.
- 3-to-8 line decoders with enable inputs can be connected to form a 4-to-16 line decoder.

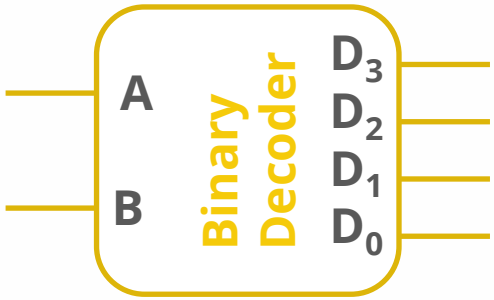
DECODERS

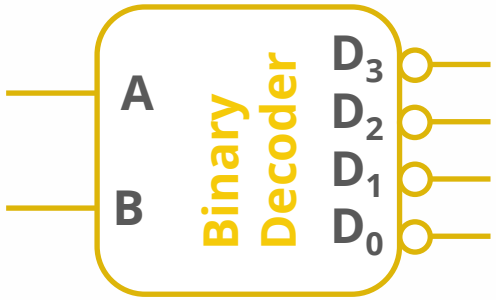
- When w = 0, top decoder enabled and other disabled.
- The bottom decoder outputs all 0's.
- The top eight outputs generate minterms 0000 to 0111.
- When w = 1, bottom decoder enabled and other disabled.



DECODERS

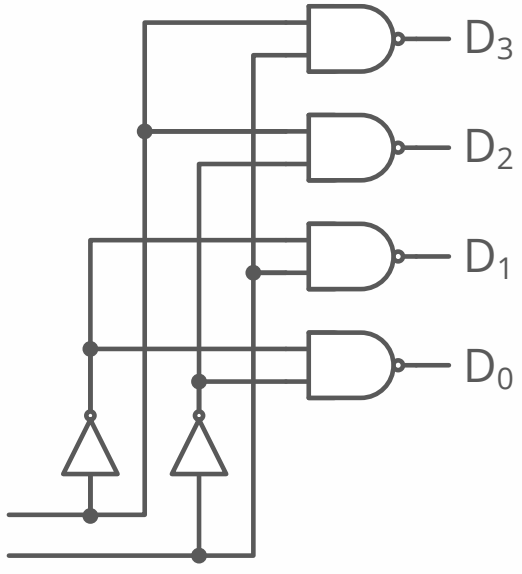
- Active – High / Active – Low





A	B	D_0	D_1	D_2	D_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

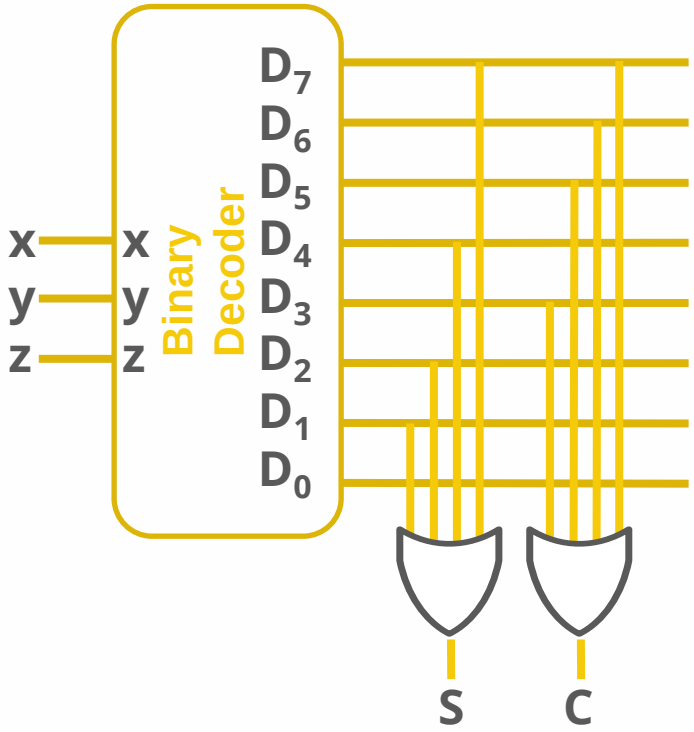
A	B	D_0	D_1	D_2	D_3
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0



DECODERS

- A decoder provides the 2n minterm of n input variables.
- Since any Boolean function can be expressed in sum of minterms
  - Use decoder to generate the mintersm
  - An external OR gate to form the logical sum.
- Any combinational circuit with n inputs and m outputs can be implemented with an n-to-2^n- line decoder and m OR gates.

DECODERS



Example: Full Adder

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

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



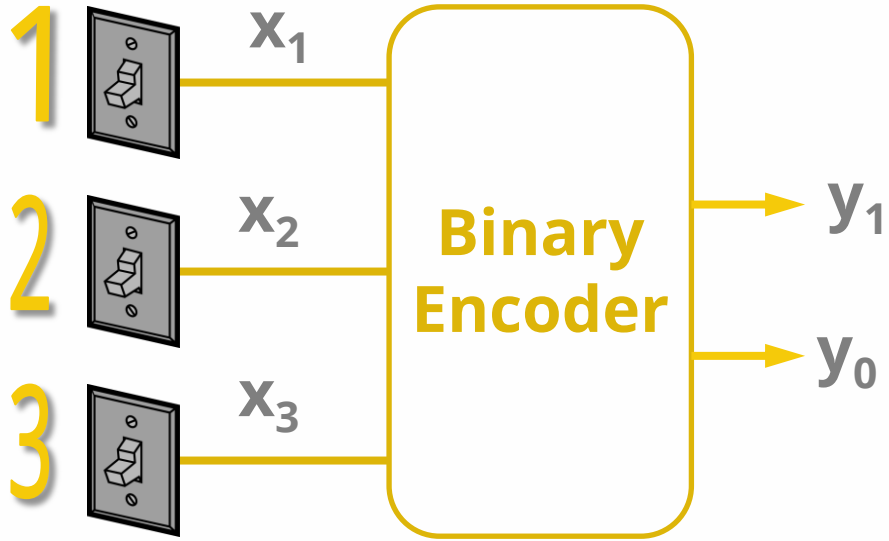
4.9 ENCODERS

ENCODERS

- An encoder is a digital circuit that performs the inverse operation of a decoder.
- An encoder has 2n (or fewer) input lines and n output lines.
- The output lines generate the binary code corresponding to the input value.

ENCODERS

- Example: 4-to-2 Binary Encoder



Only one switch should be activated at a time

x_3	x_2	x_1	y_1	y_0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
1	0	0	1	1

ENCODERS

- Example: Octal-to-Binary Encoder

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

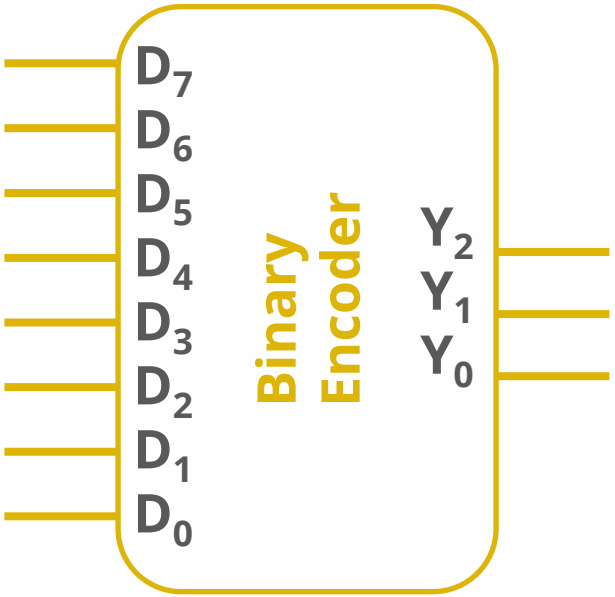
$$Y_2 = D_7 + D_6 + D_5 + D_4$$

$$Y_1 = D_7 + D_6 + D_3 + D_2$$

$$Y_0 = D_7 + D_5 + D_3 + D_1$$

ENCODERS

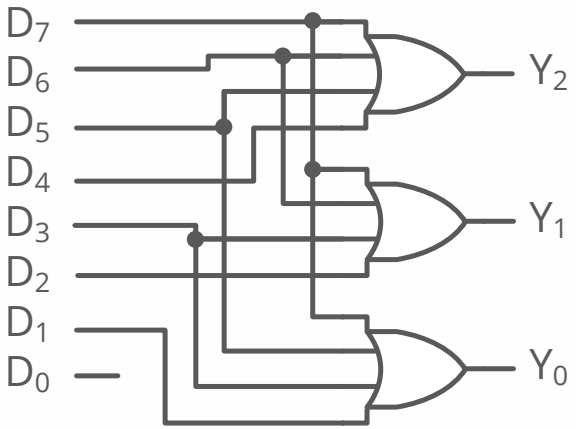
- Example: Octal-to-Binary Encoder



$$Y_2 = D_7 + D_6 + D_5 + D_4$$

$$Y_1 = D_7 + D_6 + D_3 + D_2$$

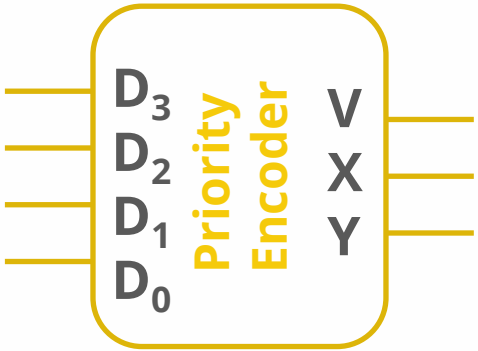
$$Y_0 = D_7 + D_5 + D_3 + D_1$$



ENCODERS

4-Input Priority Encoder

- Encoder circuit that includes the priority function.
- If two or more inputs are equal 1 at the same time,
  - The input having the highest priority will take precedence.



Inputs				Outputs		
D_3	D_2	D_1	D_0	X	Y	**V**
0	0	0	0	x	x	**0**
0	0	0	1	0	0	**1**
0	0	1	X	0	1	**1**
0	1	X	X	1	0	**1**
1	X	X	X	1	1	**1**

ENCODERS

4-Input Priority Encoder

Inputs				Outputs		
D_3	D_2	D_1	D_0	X	Y	**V**
0	0	0	0	X	X	**0**
0	0	0	1	0	0	**1**
0	0	1	X	0	1	**1**
0	1	X	X	1	0	**1**
1	X	X	X	1	1	**1**

← Valid ouput = 1, when one or more inputs =1

- X's in the output represent don't care conditions.
- X's in the input are useful fir representing a truth table in condensed form.

ENCODERS

4-Input Priority Encoder

Inputs				Outputs		
D_3	D_2	D_1	D_0	X	Y	**V**
0	0	0	0	X	X	**0**
0	0	0	1	0	0	**1**
0	0	1	X	0	1	**1**
0	1	X	X	1	0	**1**
1	X	X	X	1	1	**1**

- X's in the input are useful for representing a truth table in condensed form.
- Truth table uses an X to represent either 1 or 0.
- Example: X100 represents two minterms 0100 and 1100.

ENCODERS

4-Input Priority Encoder

Inputs				Outputs		
D_3	D_2	D_1	D_0	X	Y	**V**
0	0	0	0	X	X	**0**
0	0	0	1	0	0	**1**
0	0	1	X	0	1	**1**
0	1	X	X	1	0	**1**
1	X	X	X	1	1	**1**

- According to the table:
  - Higher the subscript number,
  - The higher the priority of the input.
  - Input D_3 has the highest priority.
  - When $D_3 = 1$, the xy is 111, regardless of the values of the other inputs.

ENCODERS

4-Input Priority Encoder

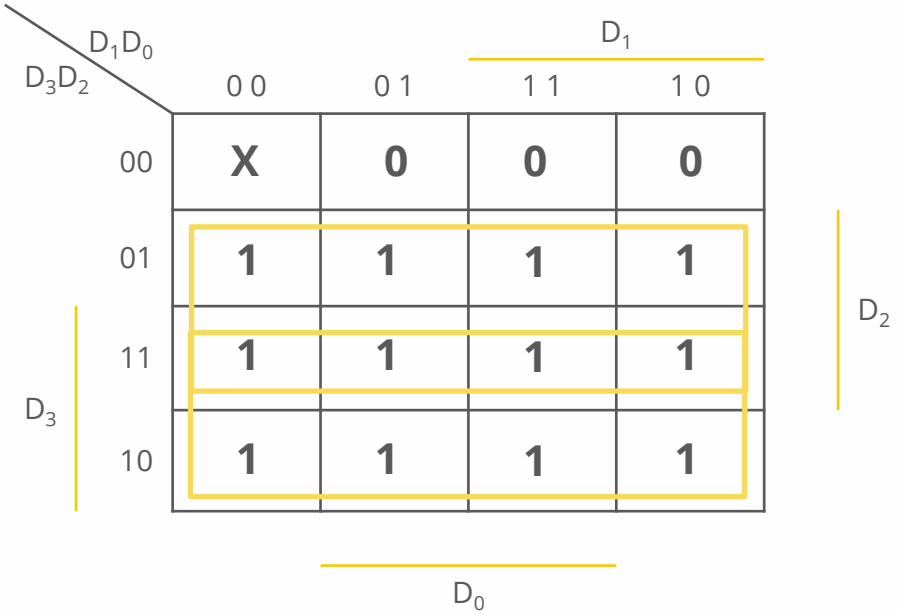
- When each X in a row is replaced first by 0 and then by 1
  - We obtain all 16 possible input combinations.
- 001X = 0010 and 0011
- 01XX = 0100 and 0101 and 0110 and 0111
- 1XXXX = 1000 and 1001 and 1010 and 1011 and 1100 and 1101 and 1110 and 1111

ENCODERS

4-Input Priority Encoder

Inputs				Outputs		
D_3	D_2	D_1	D_0	X	Y	**V**
0	0	0	0	X	X	**0**
0	0	0	1	0	0	**1**
0	0	1	X	0	1	**1**
0	1	X	X	1	0	**1**
1	X	X	X	1	1	**1**

X's K-MAP



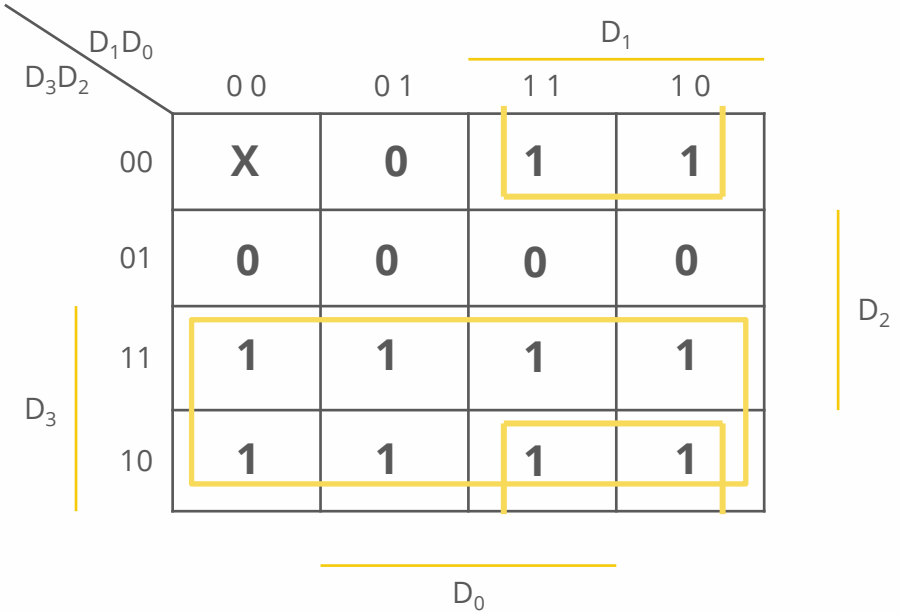
$$X = D_2 + D_3$$

ENCODERS

4-Input Priority Encoder

Inputs				Outputs		
D_3	D_2	D_1	D_0	X	Y	**V**
0	0	0	0	X	X	**0**
0	0	0	1	0	0	**1**
0	0	1	X	0	1	**1**
0	1	X	X	1	0	**1**
1	X	X	X	1	1	**1**

Y's K-MAP



$$Y = D_3 + D_1D_2'$$

ENCODERS

4-Input Priority Encoder

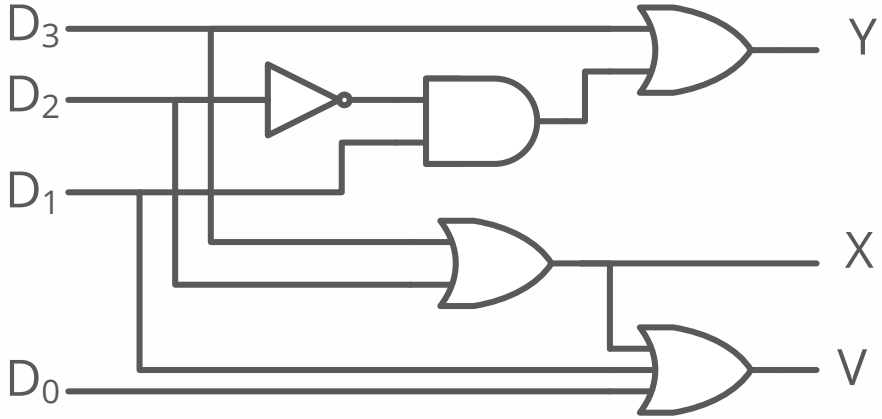
Inputs				Outputs		
D_3	D_2	D_1	D_0	X	Y	**V**
0	0	0	0	X	X	**0**
0	0	0	1	0	0	**1**
0	0	1	X	0	1	**1**
0	1	X	X	1	0	**1**
1	X	X	X	1	1	**1**

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

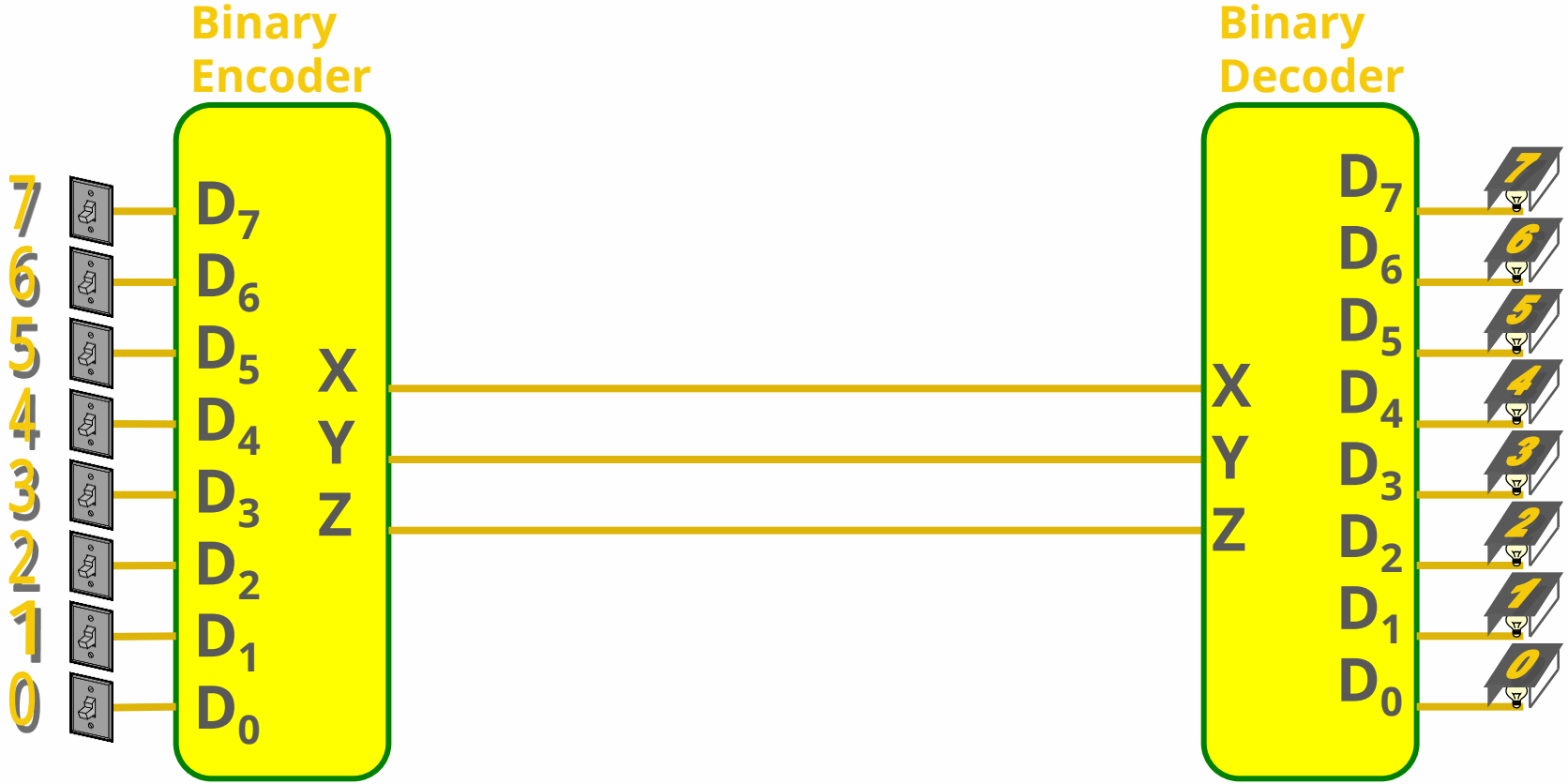
ENCODERS

4-Input Priority Encoder

- $X = D_2 + D_3$
- $Y = D_3 + D_1 D_2'$
- $V = D_0 + D_1 + D_2 + D_3$



ENCODERS

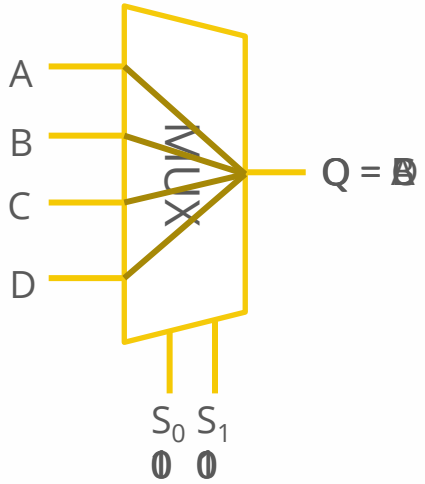

Binary
Encoder
7
6
5
4
3
2
1
0
D7
D6
D5
D4
D3
D2
D1
D0
X
Y
Z
Binary
Decoder
X
Y
Z
D7
D6
D5
D4
D3
D2
D1
D0
7
6
5
4
3
2
1
0



4.10 MULTIPLEXERS

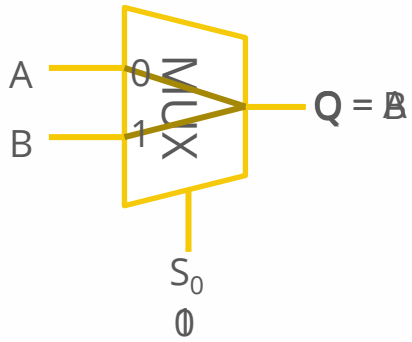
MULTIPLEXERS

- A multiplexer (MUX) is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line.
- The selection of a particular input line is controlled by a set of selection lines.
- There are 2^n input lines and n selected lines.

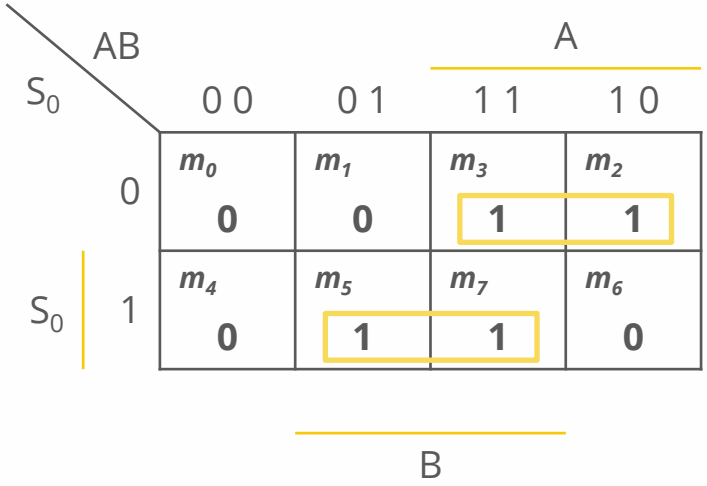


MULTIPLEXERS

- Consider 2-to-1 MUX



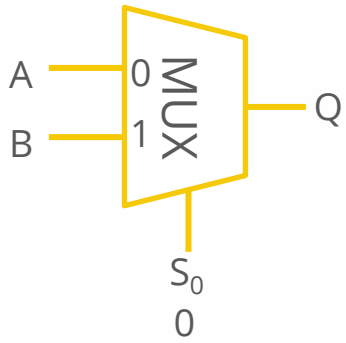
S_0	A	B	Q
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

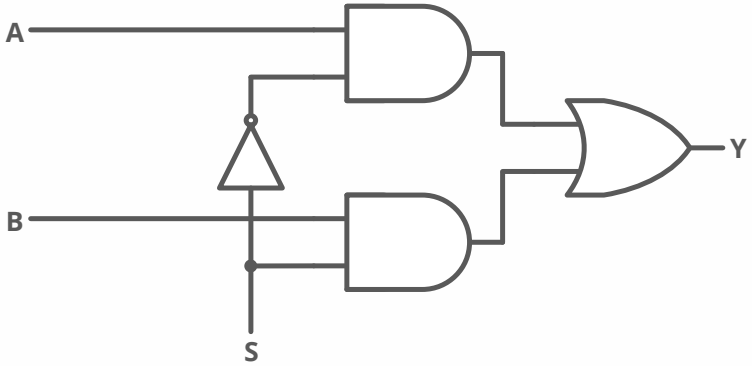


$$Q = S_0' A + S_0 B$$

MULTIPLEXERS

- Consider 2-to-1 MUX

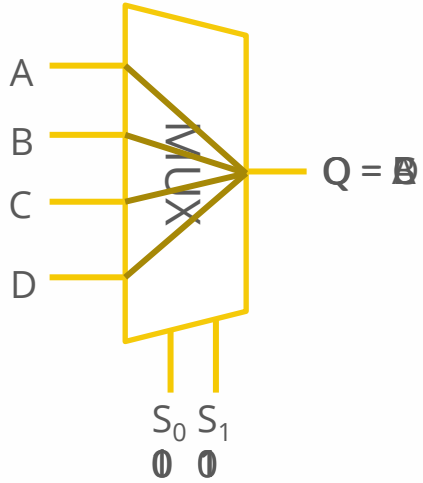




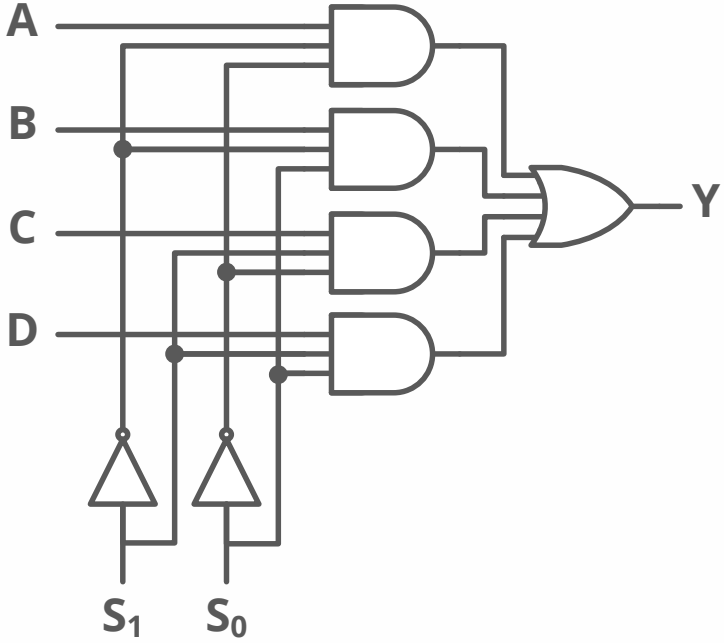
$$Q = S_0' A + S_0 B$$

MULTIPLEXERS

- 4-to-1 Multiplexer

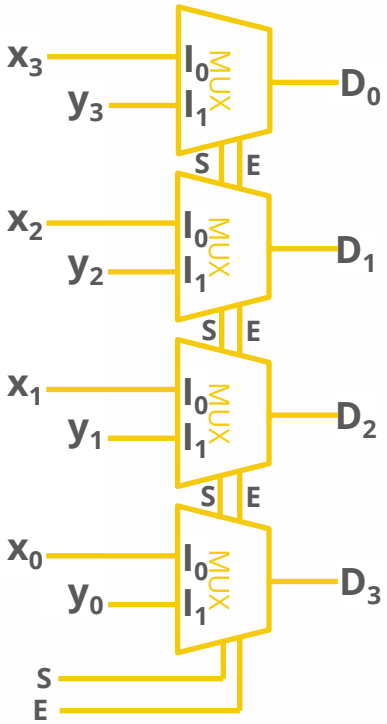


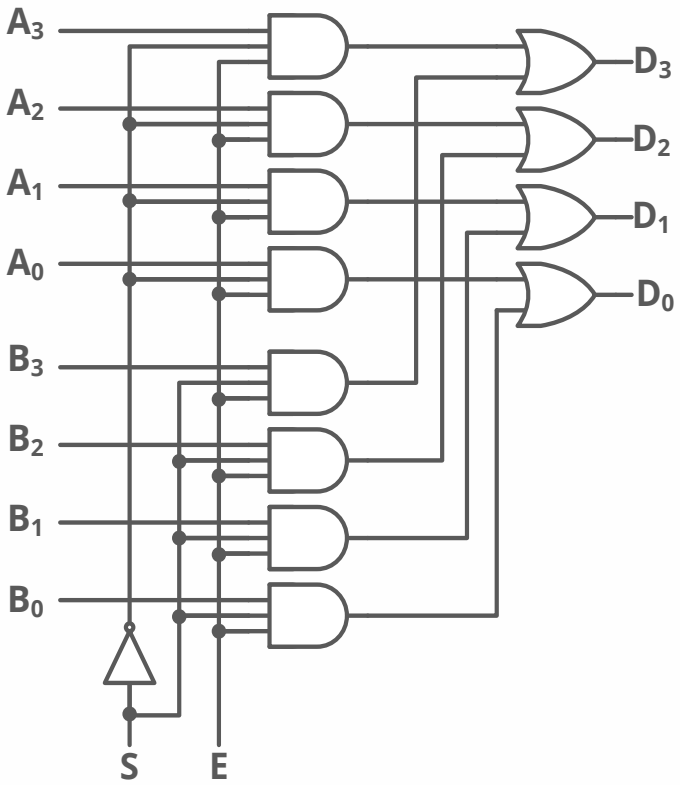
S_0	S_1	Q
0	0	A
0	1	B
1	0	C
1	1	D



MULTIPLEXERS

- 4-to-1 Multiplexer

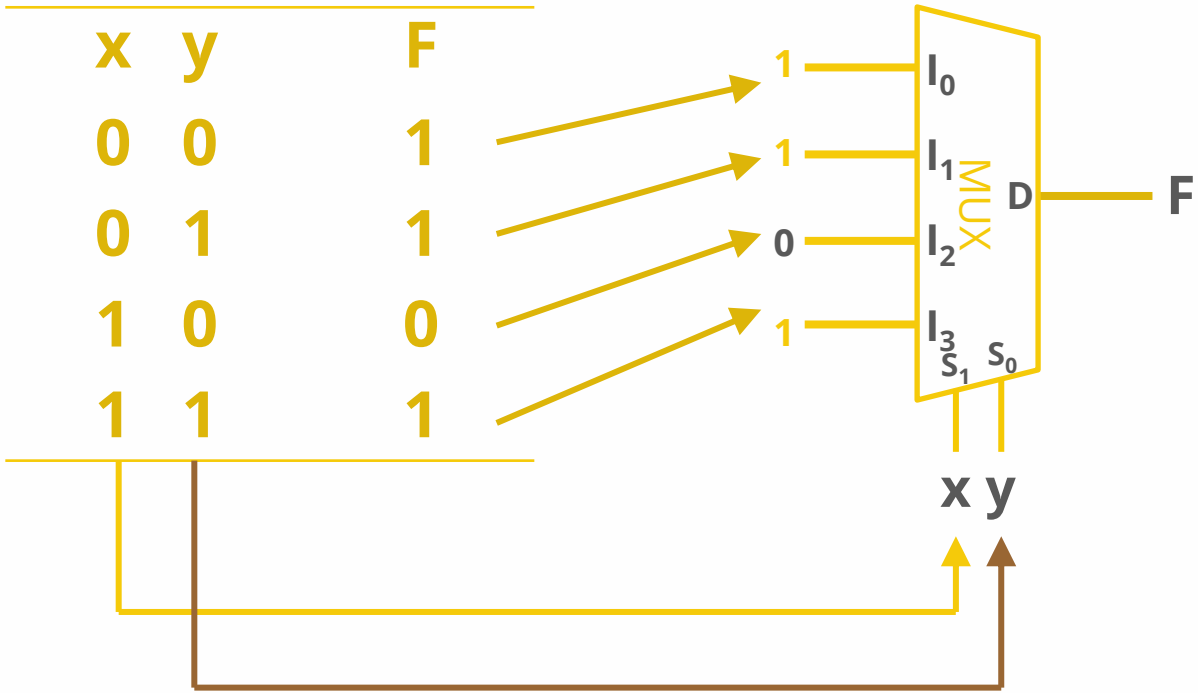




MULTIPLEXERS

- Example: F(x, y) = ∑ (0, 1, 3)

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



MULTIPLEXERS

- Boolean function implementation
  - Decoder can be used to implement Boolean functions by employing external OR gates. (see slide 114- DECODERS).
  - Multiplexer is essentially a decoder that includes the OR gate within the unit.
  - The minterms of a function are generated in multiplexer by the circuit associated with the selection inputs.

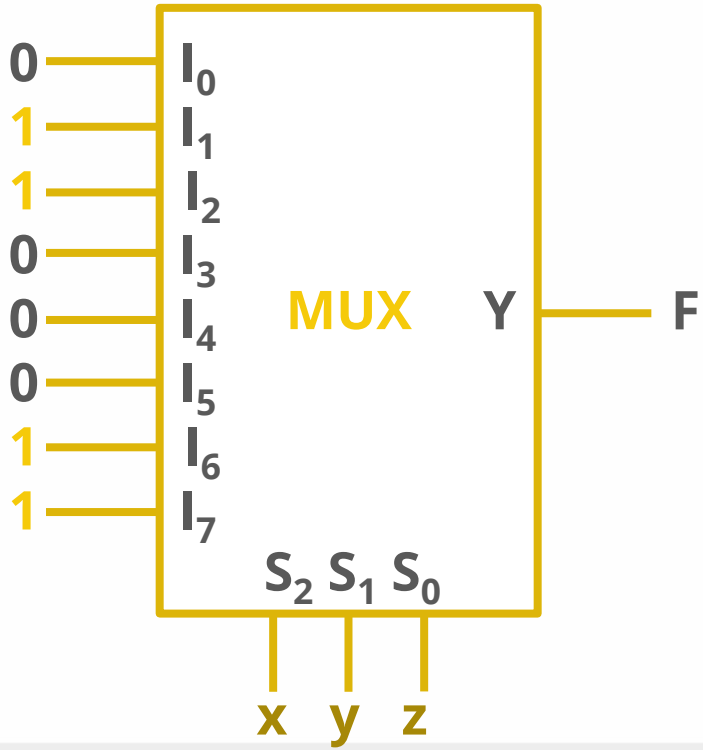
MULTIPLEXERS

- Boolean function implementation
  - This provides a method of implementing a Boolean function of n variables with a multiplexer that has n – 1 selection inputs.
  - The first n – 1 variables of the function are connected to the selection inputs of the multiplexer.
  - The remaining single variable of the function is used for the data inputs.

MULTIPLEXERS

- Example:F(x, y, z) = ∑(1, 2, 6, 7)

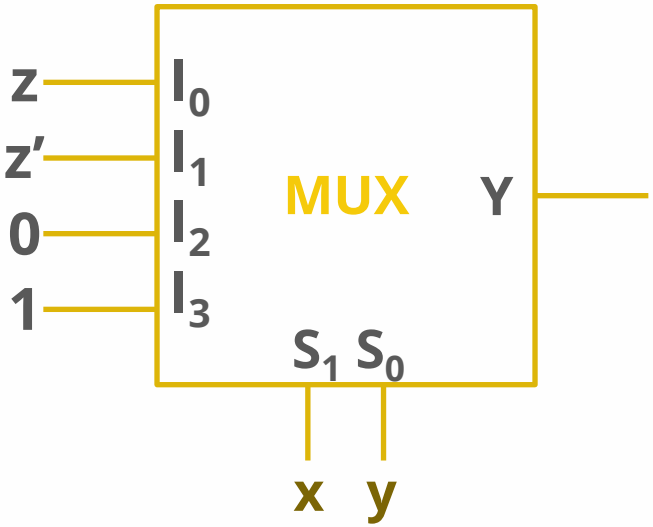
x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



MULTIPLEXERS

- Example:F(x, y, z) = ∑(1, 2, 6, 7)

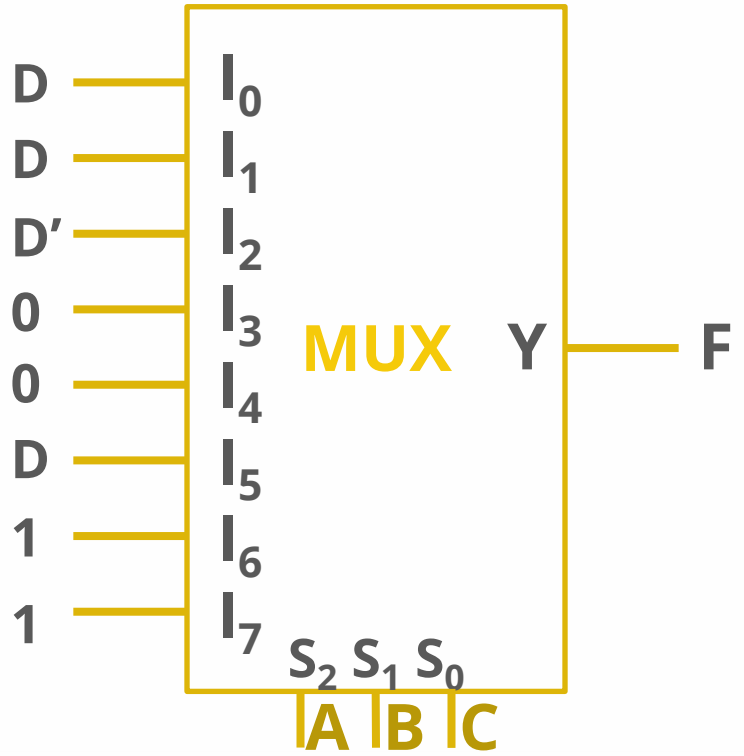
x	y	z	F	
0	0	0	0	F = z
0	0	1	1	
0	1	0	1	F = z’
0	1	1	0	
1	0	0	0	F = 0
1	0	1	0	
1	1	0	1	F = 1
1	1	1	1	



MULTIPLEXERS

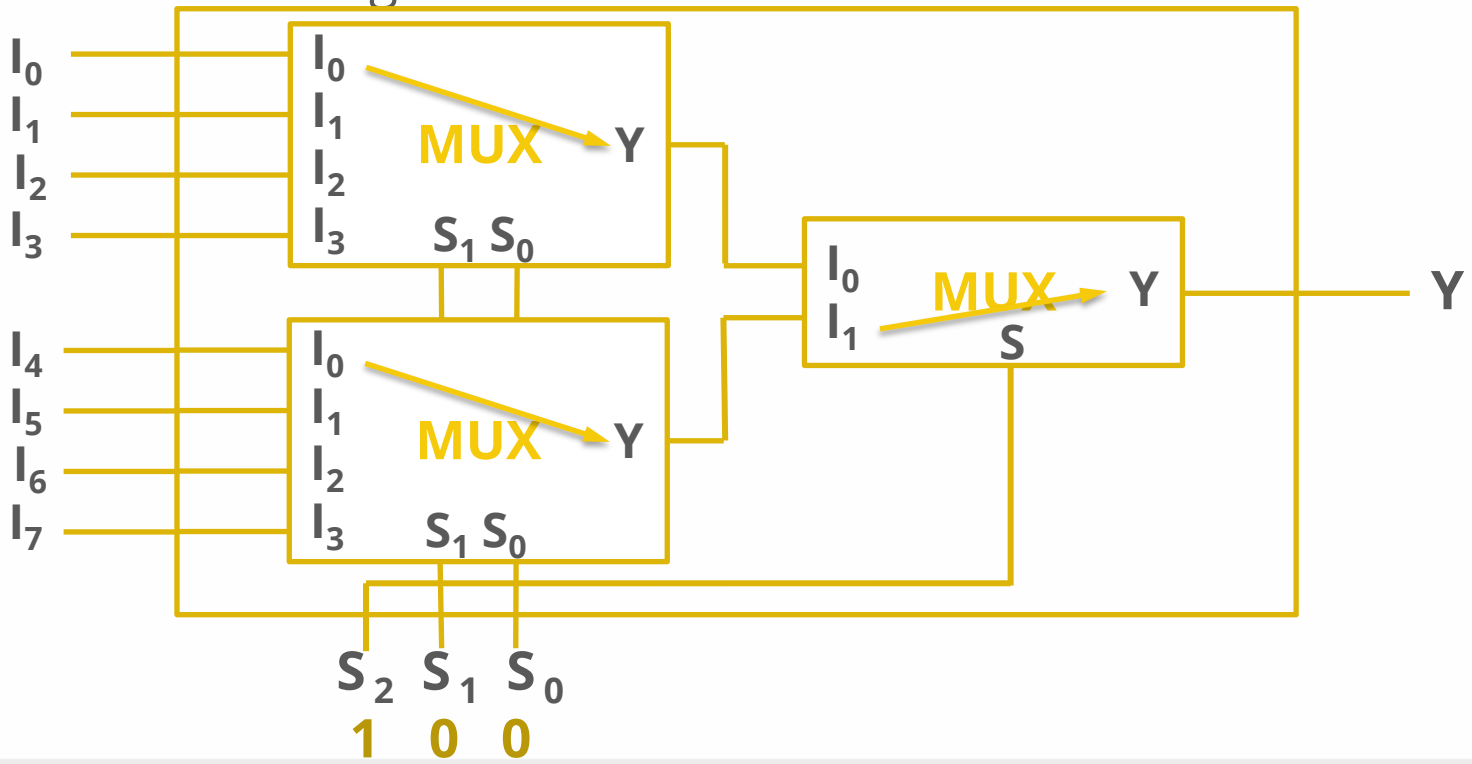
- Example:F(A, B, C, D) = ∑(1,3,4,11,12,13,14,15)

A	B	C	D	F	
0	0	0	0	0	F = D
0	0	0	1	1	
0	0	1	0	0	F = D
0	0	1	1	1	
0	1	0	0	1	F = D’
0	1	0	1	0	
0	1	1	0	0	F = 0
0	1	1	1	0	
1	0	0	0	0	F = 0
1	0	0	1	0	
1	0	1	0	0	F = D
1	0	1	1	1	
1	1	0	0	1	F = 1
1	1	0	1	1	
1	1	1	0	1	F = 1
1	1	1	1	1	

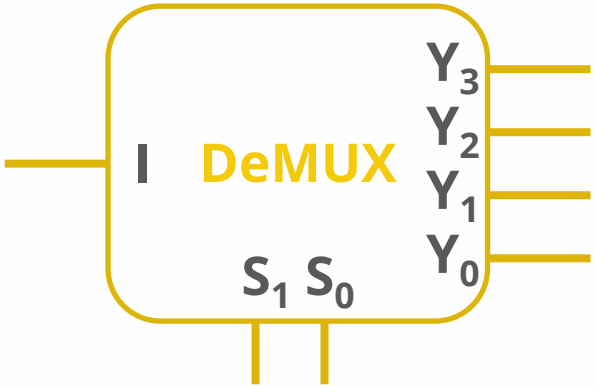


MULTIPLEXERS

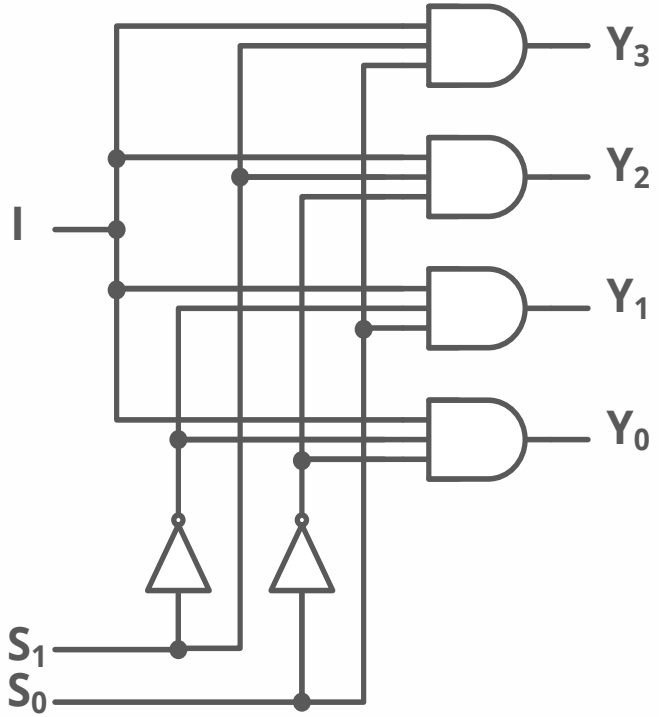
- 8-to-1 MUX using Dual 4-to-1 MUX



DEMULTIPLEXERS



S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0



DEMULTIPLEXER





S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0

E	A	B	Y_3	Y_2	Y_1	Y_0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

MULTIPLEXERS

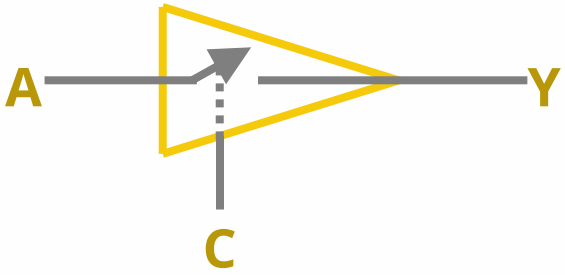
- A multiplexer can be constructed with three – state gates.
- A three – state gate is a digital circuit that exhibits three states.
- Two of the states are signals equivalent to logic 1 and 0.
- The third state is **high – impedance** state.
- Hight – impedance state behaves like an open circuit.
  - The output appears to be disconnected.
  - The circuit has no logic significance.

MULTIPLEXERS

- Three – state gates may perform any conventional logic such as AND or NAND.
- The most commonly used is the buffer gate.
- The high – impedance state of a three – state gate provides a special feature not available in other gates.
- Large number of three – state gate outputs can be connected with wires to form a common line without endangering loading effects.

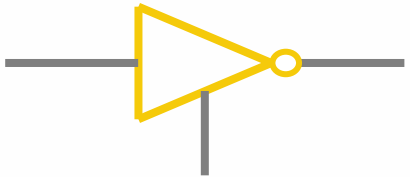
MULTIPLEXERS

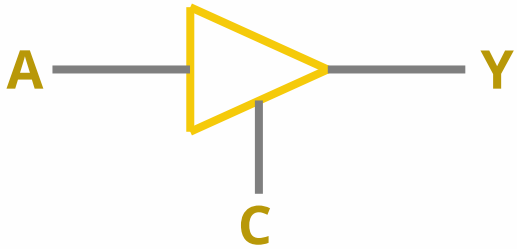
- Three state gates
  - Tri-state buffer



C	A	Y
0	x	Hi-Z
1	0	0
1	1	1

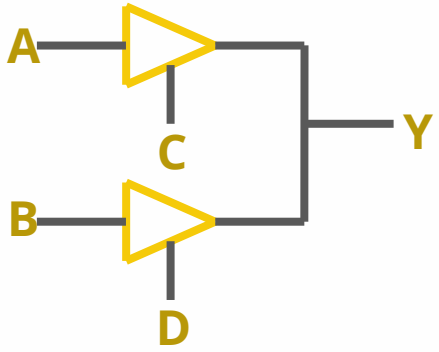
  - Tri-state inverter

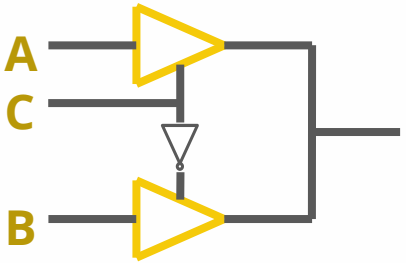




MULTIPLEXERS

- Three state gates





$$Y = \begin{cases} A & \text{if } C = 1 \\ B & \text{if } C = 0 \end{cases}$$

2-to-1 MUX

C	D	Y
0	0	Hi-Z
0	1	B
1	0	A
1	1	?

Not Allowed

MULTIPLEXERS

- 4-to-1 MUX

