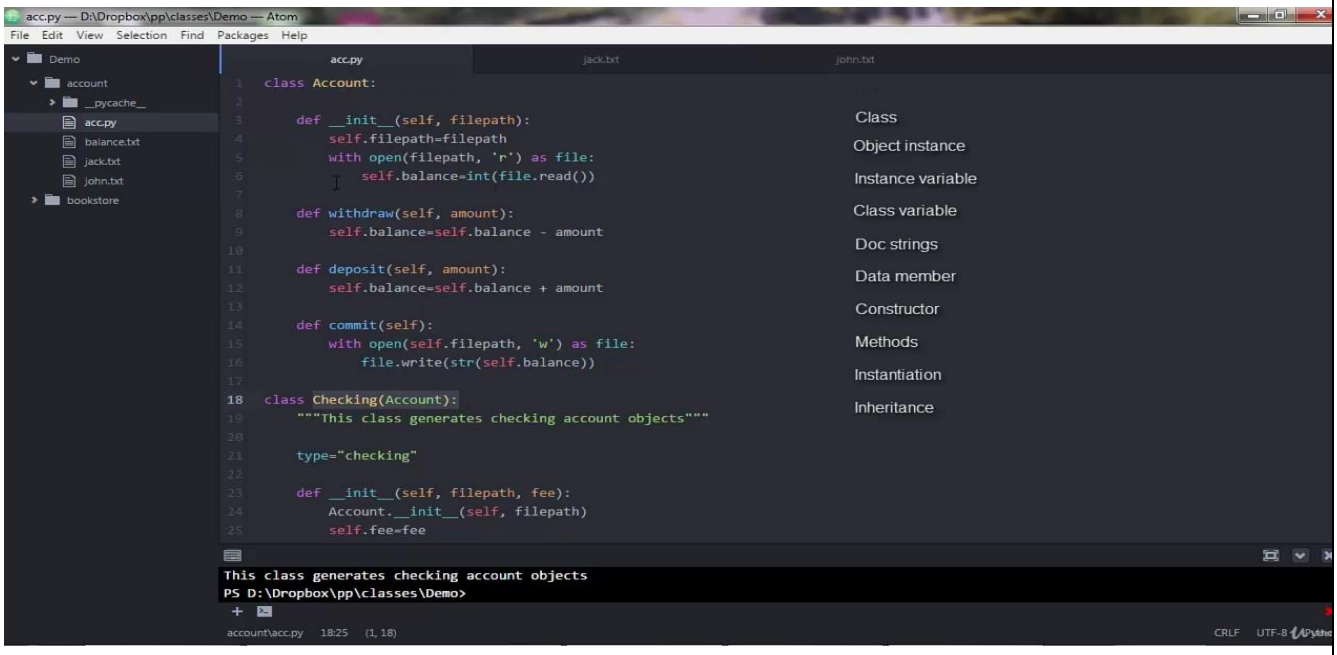
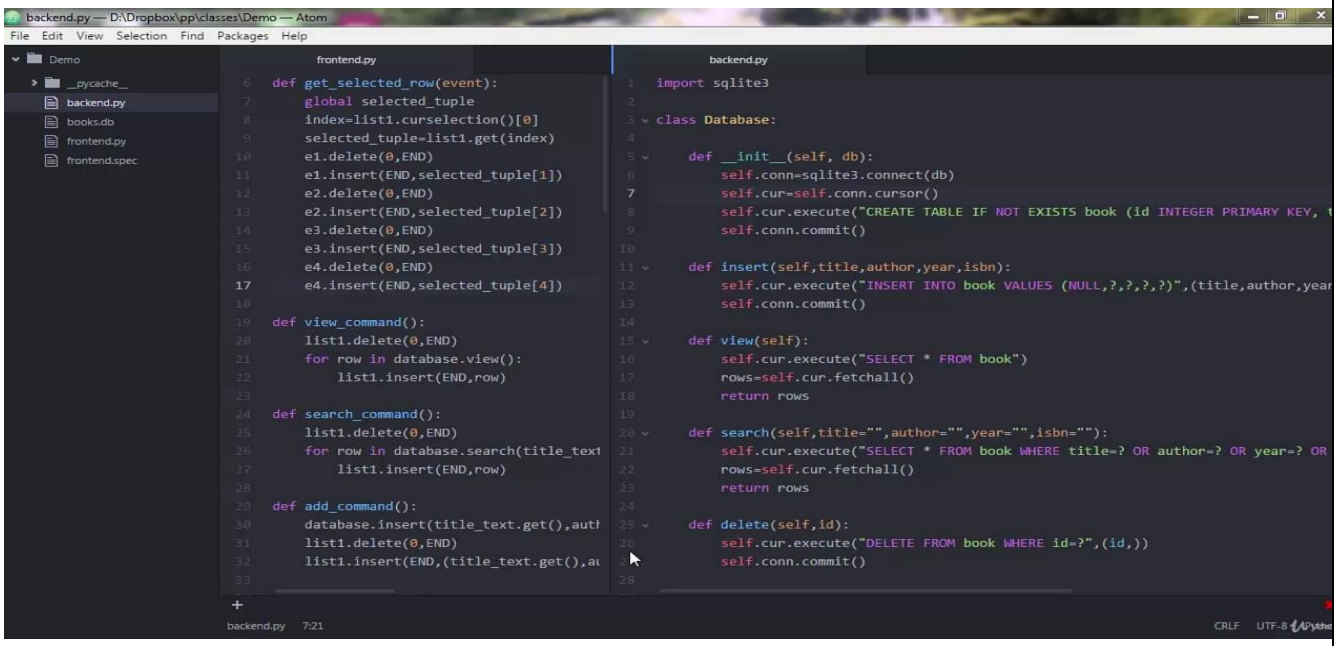
DAILY ASSESSMENT FORMAT

Date:	28th May 2020	**Name:**	Soundarya NA
Course:	UDEMY	**USN:**	4AL16EC077
Topic:	PYTHON: Object Oriented Programming	**Semester & Section:**	8th - B

FORENOON SESSION DETAILS

Image of session





Report:

Object Oriented Programming:

Introduction to oops in Python:

Python is a multi-paradigm programming language. Meaning, it supports different programming approach. One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- Attributes
- behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself). In Python, the concept of OOP follows some basic principles:

Inheritance	A process of using details from a new class without modifying existing class.
Encapsulation	Hiding the private details of a class from other objects.
Polymorphism	A concept of using common operation in different ways for different data input.

Class:

A class is a blueprint for the object.

We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, parrot is an object.

The example for class of parrot can be:

```
class Parrot:
pass
```

Here, we use **class** keyword to define an empty class **Parrot**. From class, we construct instances. An instance is a specific object created from a particular class.

Object:

An object (instance) is an instantiation of a class. When class is defined, only the description for the

object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

obj = Parrot()

Here, **obj** is object of class **Parrot**.

Creating Class and Object in Python:

```
class Parrot:
   # class attribute
   species = "bird"
   # instance attribute
   def __init__(self, name, age):
      self.name = name
      self.age = age
# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)
# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))
# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

When we run the program output will be:

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

Methods:

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Creating methods in Python:

```
class Parrot:
  # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)
    def dance(self):
        return "{} is now dancing".format(self.name)
# instantiate the object
blu = Parrot("Blu", 10)
# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

When we run the program output will be:

```
Blu sings 'Happy'
Blu is now dancing
```

Inheritance:

Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Use of inheritance in Python:

```
# parent class
class Bird:
    def __init__(self):
        print("Bird is ready")
```

```
    def whoisThis(self):
        print("Bird")
    def swim(self):
        print("Swim faster")
# child class
class Penguin(Bird):
    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")
    def whoisThis(self):
        print("Penguin")
    def run(self):
        print("Run faster")
peggy = Penguin()
peggy.whoisThis()
peggy.swim()
```

When we run the program output will be:

Bird is ready

Penguin is ready

Penguin

Swim faster

Run faster

Encapsulation:

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single " _ " or double " __".

Data Encapsulation in Python:

```
class Computer:
    def __init__(self):
        self.__maxprice = 900
    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))
    def setMaxPrice(self, price):
        self.__maxprice = price
c = Computer()
c.sell()
# change the price
c.__maxprice = 1000
c.sell()
# using setter function
c.setMaxPrice(1000)
c.sell()
```

When we run the program output will be:

Selling Price: 900

Selling Price: 900

Selling Price: 1000

Polymorphism:

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However, we could use same method to color any shape. This concept is called Polymorphism.

Using Polymorphism in Python:

```
class Parrot:
    def fly(self):
        print("Parrot can fly")
    def swim(self):
```

```
        print("Parrot can't swim")
class Penguin:
    def fly(self):
        print("Penguin can't fly")
    def swim(self):
        print("Penguin can swim")
# common interface
def flying_test(bird):
    bird.fly()
#instantiate objects
blu = Parrot()
peggy = Penguin()
# passing the object
flying_test(blu)
flying_test(peggy)
```

When we run the program output will be:

```
Parrot can fly
Penguin can't fly
```

Creating a Bank Account Object:

```
# Python program to create Bankaccount class
# with both a deposit() and a withdraw() function
class Bank_Account:
    def __init__(self):
        self.balance=0
        print("Hello!!! Welcome to the Deposit & Withdrawal Machine")
    def deposit(self):
        amount=float(input("Enter amount to be Deposited: "))
        self.balance += amount
```

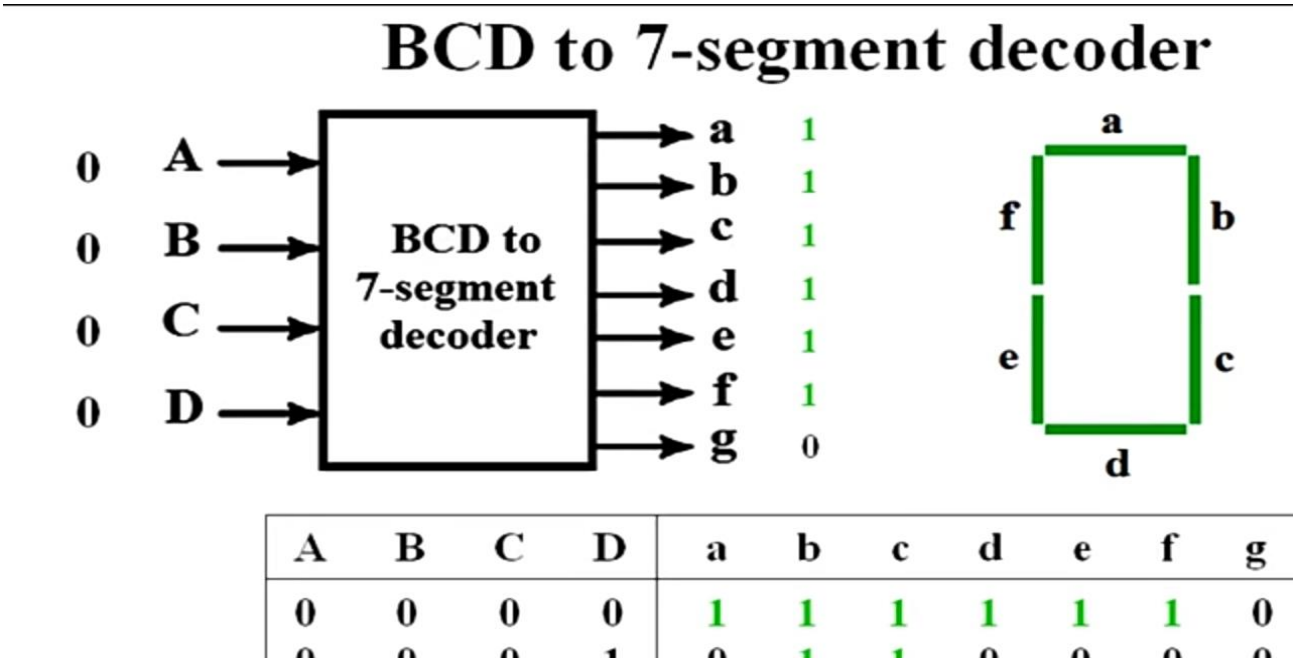
```
        print("\n Amount Deposited:",amount)
    def withdraw(self):
        amount = float(input("Enter amount to be Withdrawn: "))
        if self.balance>=amount:
            self.balance-=amount
            print("\n You Withdrew:", amount)
        else:
            print("\n Insufficient balance  ")
    def display(self):
        print("\n Net Available Balance=",self.balance)
# Driver code
# creating an object of class
s = Bank_Account()
# Calling functions with that class object
s.deposit()
s.withdraw()
s.display()
```

Output:

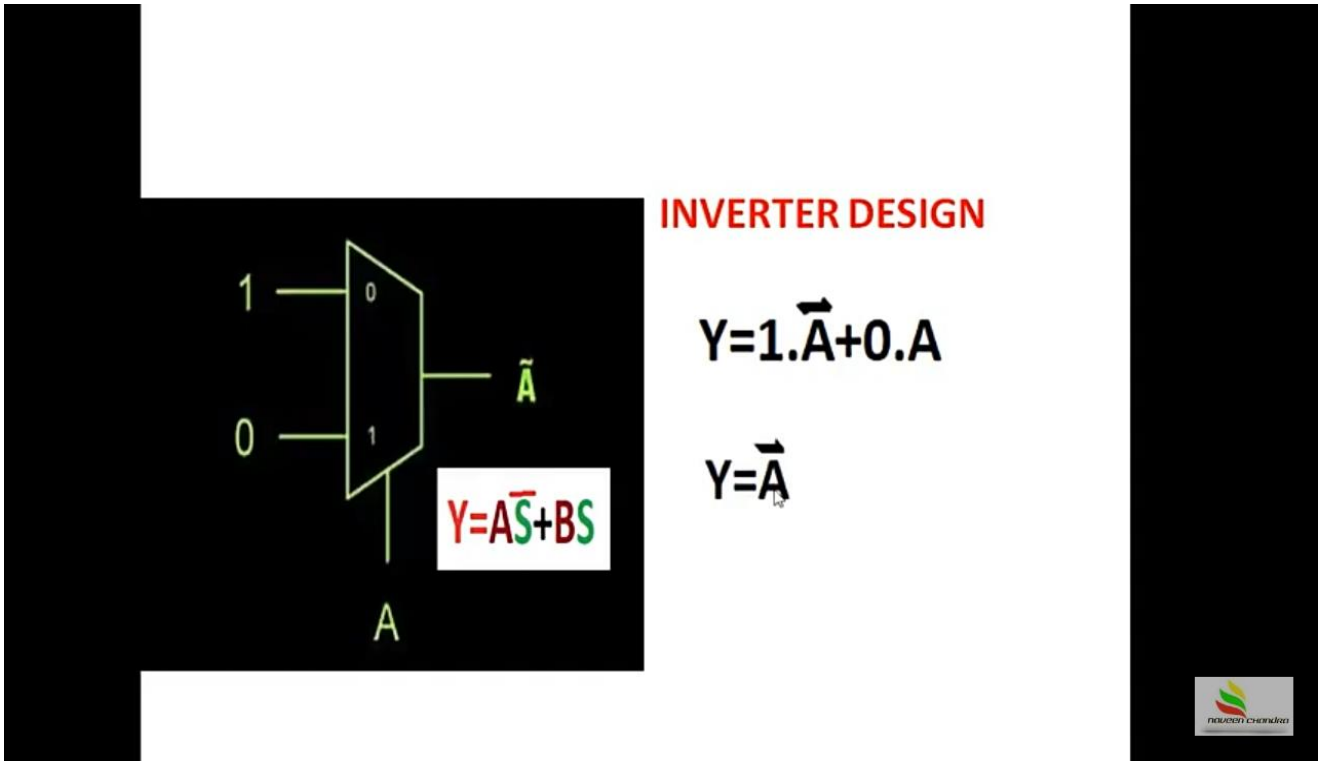
```
Hello !!! Welcome to Deposit & Withdrawal Machine
Enter amount to be deposited:
 Amount Deposited: 1000.0
Enter amount to be withdrawn:
 You Withdrew: 500.0
 Net Available Balance = 500.0
```

Date:	28th May 2020	**Name:**	Soundarya NA
Course:	Logic Design	**USN:**	4AL16EC077
Topic:	Logic Design Day 1	**Semester & Section:**	8th - B

Image:



A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0



Report:

Boolean Algebra:

Boolean Algebra is an algebra, which deals with binary numbers & binary variables. Hence, it is also called as Binary Algebra or logical Algebra. A mathematician, named George Boole had developed this algebra in 1854. The variables used in this algebra are also called as Boolean variables.

The range of voltages corresponding to Logic 'High' is represented with '1' and the range of voltages corresponding to logic 'Low' is represented with '0'.

Postulates and Basic Laws of Boolean Algebra:

Boolean Postulates:

Consider the binary numbers 0 and 1, Boolean variable x and its complement x'. Either the Boolean variable or complement of it is known as literal. The four possible logical OR operations among these literals and binary numbers are shown below.

$$x + 0 = x$$

$$x + 1 = 1$$

$$x + x = x$$

$$x + x' = 1$$

Similarly, the four possible logical AND operations among those literals and binary numbers are shown below.

$$x.1 = x$$

$$x.0 = 0$$

$$x.x = x$$

$$x.x' = 0$$

These are the simple Boolean postulates. We can verify these postulates easily, by substituting the Boolean variable with '0' or '1'.

Basic Laws of Boolean Algebra:

Following are the three basic laws of Boolean Algebra:

Commutative Law: If any logical operation of two Boolean variables give the same result irrespective of the order of those two variables, then that logical operation is said to be Commutative. The logical OR & logical AND operations of two Boolean variables x & y are shown below

$$x + y = y + x$$

$$x.y = y.x$$

The symbol '+' indicates logical OR operation. Similarly, the symbol '.' indicates logical AND operation and it is optional to represent. Commutative law obeys for logical OR & logical AND operations.

Associative Law: If a logical operation of any two Boolean variables is performed first and then the same operation is performed with the remaining variable gives the same result, then that logical operation is said to be Associative. The logical OR & logical AND operations of three Boolean variables x, y & z are shown below.

$$x + y+zy+z = x+yx+y + z$$

$$x.y.zy.z = x.yx.y.z$$

Associative law obeys for logical OR & logical AND operations.

Distributive Law: If any logical operation can be distributed to all the terms present in the Boolean function, then that logical operation is said to be Distributive. The distribution of logical OR & logical AND operations of three Boolean variables x, y & z are shown below.

$$x.y+zy+z = x.y + x.z$$

$$x + y.zy.z = x+yx+y.x+zx+z$$

Distributive law obeys for logical OR and logical AND operations.

These are the Basic laws of Boolean algebra. We can verify these laws easily, by substituting the Boolean variables with '0' or '1'.

Theorems of Boolean Algebra:

The following two theorems are used in Boolean algebra.

- **Duality Theorem:** This theorem states that the dual of the Boolean function is obtained by interchanging the logical AND operator with logical OR operator and zeros with ones. For every Boolean function, there will be a corresponding Dual function.

Group1	Group2
x + 0 = x	x.1 = x
x + 1 = 1	x.0 = 0
x + x = x	x.x = x
x + x' = 1	x.x' = 0
x + y = y + x	x.y = y.x
x + y+zy+z = x+yx+y + z	x.y.zy.z = x.yx.y.z
x.y+zy+z = x.y + x.z	x + y.zy.z = x+yx+y.x+z

- **DeMorgan's Theorem:** This theorem is useful in finding the complement of Boolean function. It states that the complement of logical OR of at least two Boolean variables is equal to the logical AND of each complemented variable. DeMorgan's theorem with 2 Boolean variables x and y can be represented as

  x+yx+y' = x'.y'

  The dual of the above Boolean function is

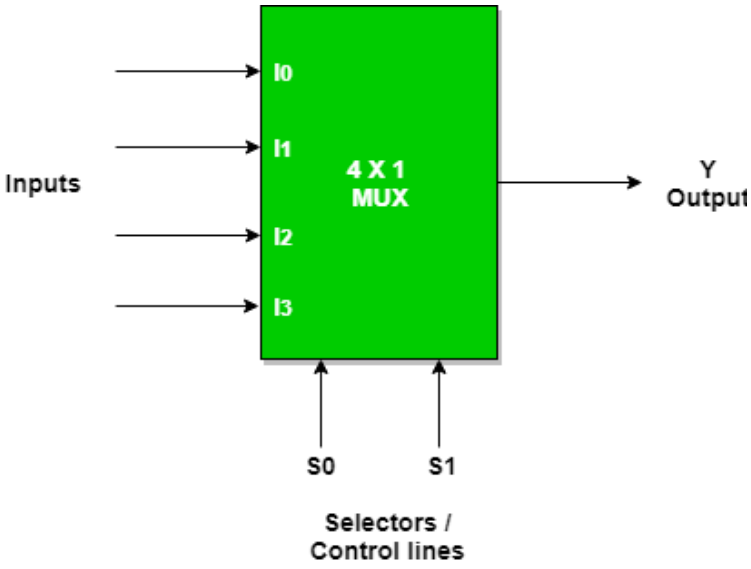
  x.yx.y' = x' + y'

Therefore, the complement of logical AND of two Boolean variables is equal to the logical OR of each complemented variable. Similarly, we can apply DeMorgan's theorem for more than 2 Boolean variables also.

MUX to logic gates:

It is a combinational circuit which have many data inputs and single output depending on control or select inputs. For N input lines, log n (base2) selection lines, or we can say that for 2n input lines, n selection lines are required. Multiplexers are also known as "Data n selector, parallel to serial

convertor, many to one circuit, universal logic circuit". Multiplexers are mainly used to increase amount of the data that can be sent over the network within certain amount of time and bandwidth.

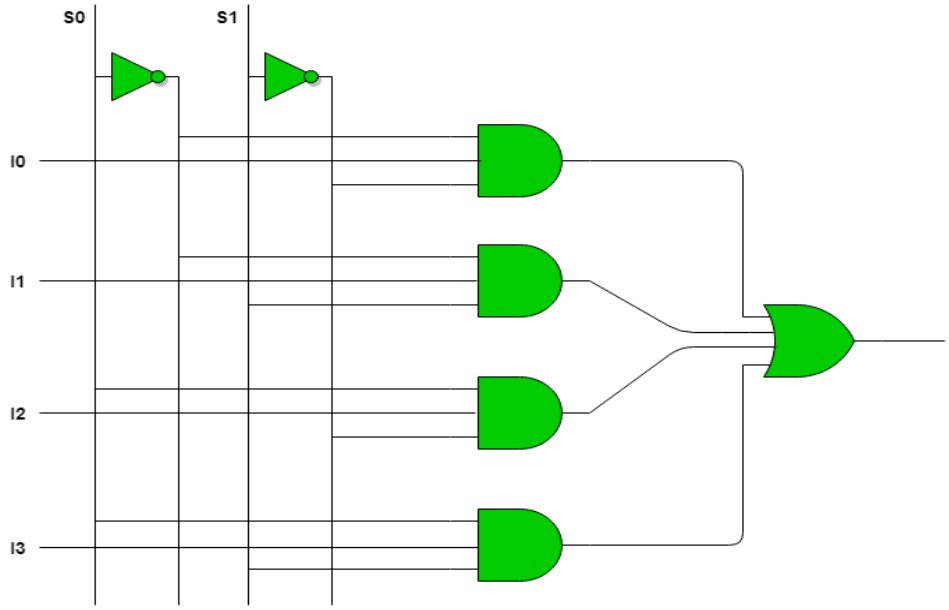


Implementation of 4:1 Multiplexer using truth table and gates.

Truth Table

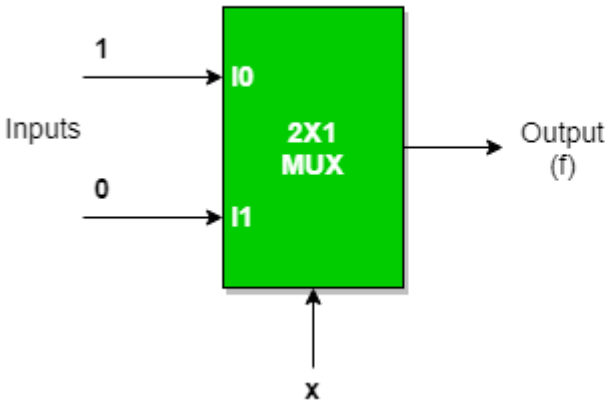
S0	S1	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

So, final equation,
Y = S0'.S1'.I0 + S0'.S1.I1 + S0.S1'.I2 + S0.S1.I3



Multiplexer can act as universal combinational circuit. All the standard logic gates can be implemented with multiplexers.

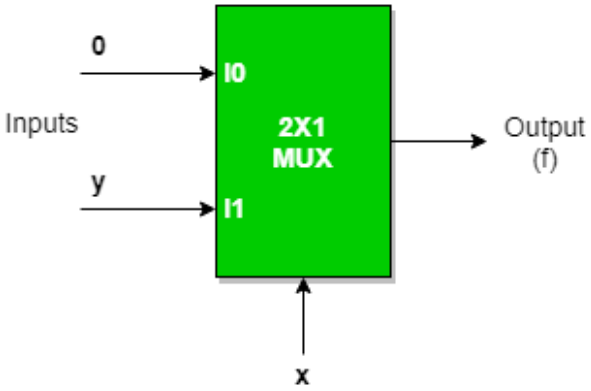
NOT gate:



Truth Table

x	f
0	1
1	0

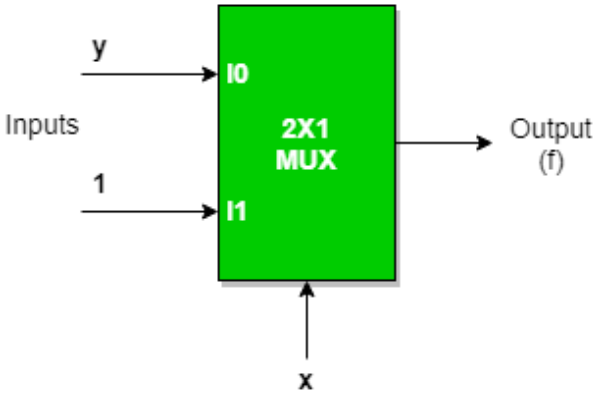
AND gate:



Truth Table

x	y	f	f→y
0	0	0	f = 0
0	1	0	
1	0	0	f = y
1	1	1	

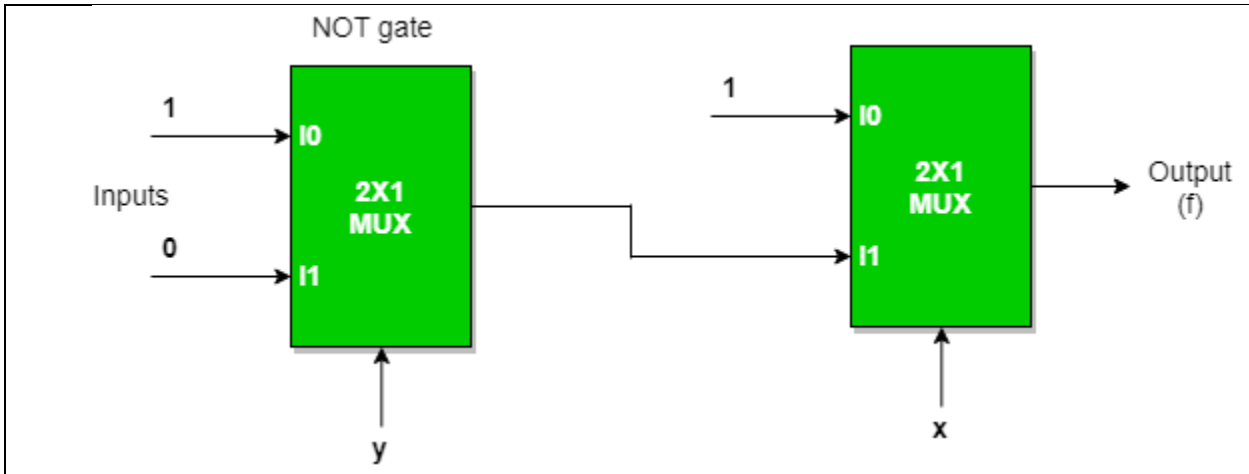
OR gate:



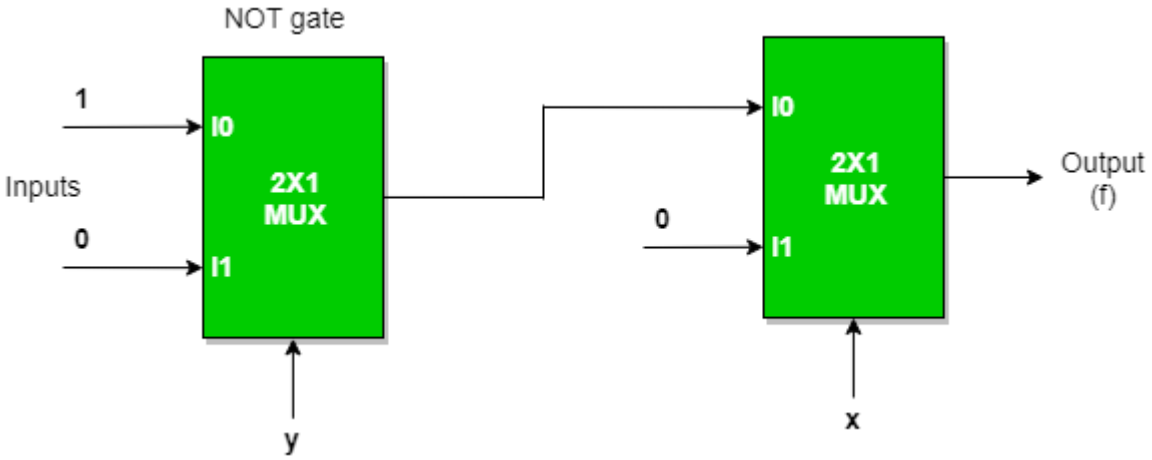
Truth Table

x	y	f	f→y
0	0	0	f = y
0	1	1	
1	0	1	f = 1
1	1	1	

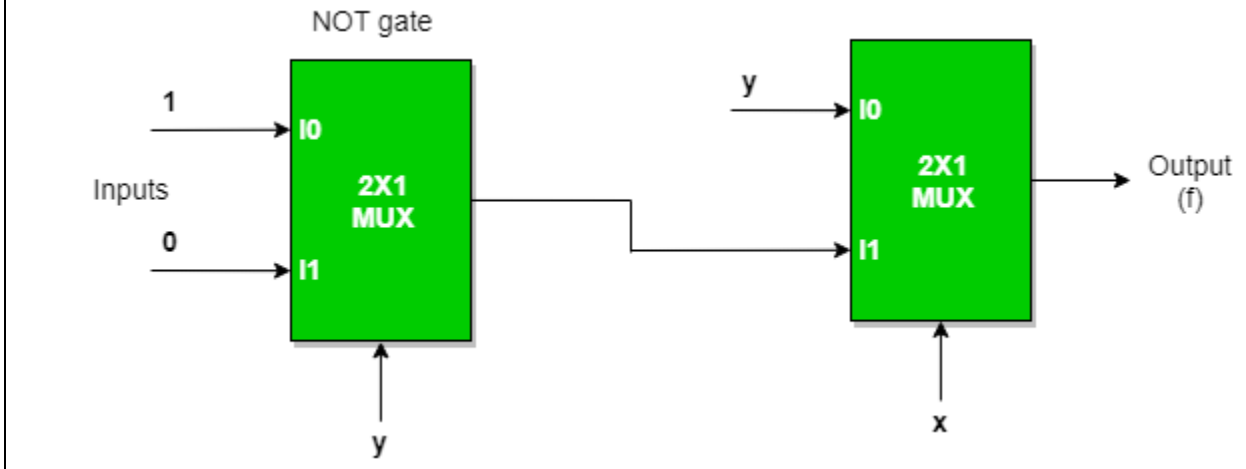
NAND gate:



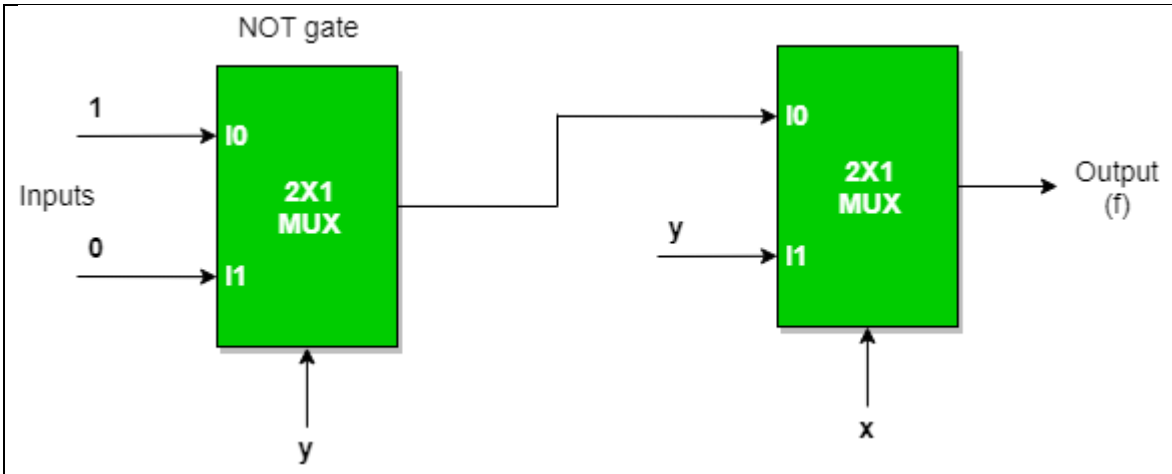
NOR gate:



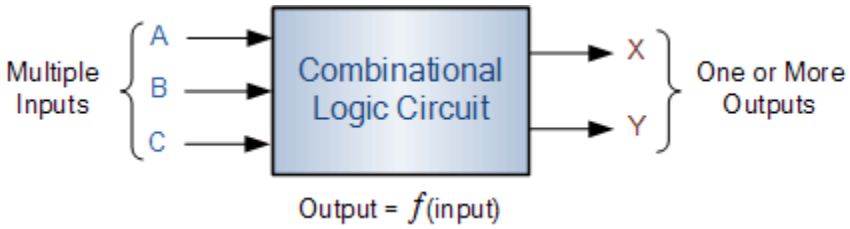
EX-OR gate:



EX-NOR gate:



Combinational Logic:



Combinational Logic Circuits are made up from basic logic NAND, NOR or NOT gates that are "combined" or connected together to produce more complicated switching circuits. These logic gates are the building blocks of combinational logic circuits. An example of a combinational circuit is a decoder, which converts the binary code data present at its input into a number of different output lines, one at a time producing an equivalent decimal code at its output. Combinational logic circuits can be very simple or very complicated and any combinational circuit can be implemented with only NAND and NOR gates as these are classed as "universal" gates.

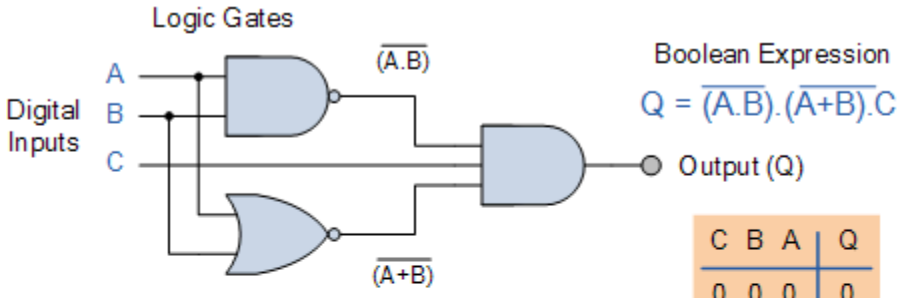
The three main ways of specifying the function of a combinational logic circuit are:

1. **Boolean Algebra** – This forms the algebraic expression showing the operation of the logic circuit for each input variable either True or False that results in a logic "1" output.

2. Truth Table – A truth table defines the function of a logic gate by providing a concise list that shows all the output states in tabular form for each possible combination of input variable that the gate could encounter.

3. Logic Diagram – This is a graphical representation of a logic circuit that shows the wiring and connections of each individual logic gate, represented by a specific graphical symbol, that implements the logic circuit.

and all three of these logic circuit representations are shown below:

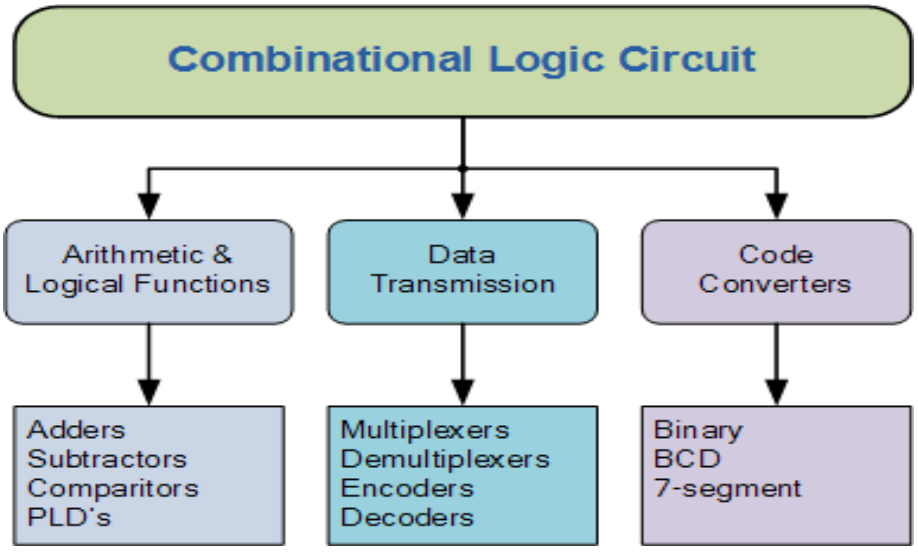


Typical Truth Table

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

As combinational logic circuits are made up from individual logic gates only, they can also be considered as "decision making circuits" and combinational logic is about combining logic gates together to process two or more signals in order to produce at least one output signal according to the logical function of each logic gate. Common combinational circuits made up from individual logic gates that carry out a desired application include Multiplexers, De-multiplexers, Encoders, Decoders, Full and Half Adders etc.

Classification of Combinational Logic:



One of the most common uses of combinational logic is in Multiplexer and De-multiplexer type circuits. Here, multiple inputs or outputs are connected to a common signal line and logic gates are used to decode an address to select a single data input or output switch.

A multiplexer consists of two separate components, a logic decoder and some solid state switches, but before we can discuss multiplexers, decoders and de-multiplexers in more detail we first need to understand how these devices use these "solid state switches" in their design.

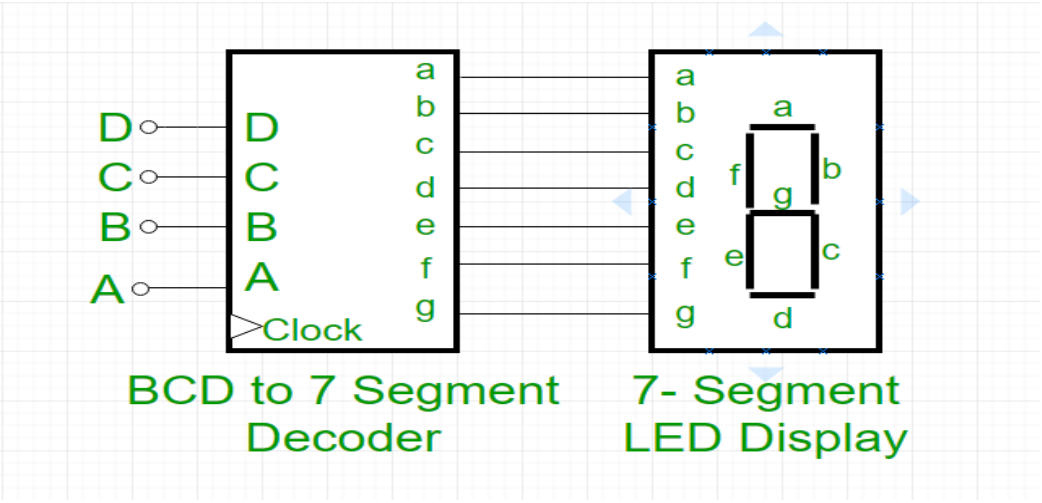
BCD to 7 Segment LED Display Decoder Circuit:

In Binary Coded Decimal (BCD) encoding scheme each of the decimal numbers (0-9) is represented by its equivalent binary pattern (which is generally of 4-bits).

Whereas, seven segment display is an electronic device which consists of seven Light Emitting Diodes (LEDs) arranged in some definite pattern (common cathode or common anode type), which is used to display Hexadecimal numerals (in this case decimal numbers, as input is BCD i.e., 0-9).

Two types of seven segment LED display:

- **Common Cathode Type:** In this type of display all cathodes of the seven LEDs are connected together to the ground or -Vcc (hence, common cathode) and LED displays digits when some 'HIGH' signal is supplied to the individual anodes.
- **Common Anode Type:** In this type of display all the anodes of the seven LEDs are connected to battery or +Vcc and LED displays digits when some 'LOW' signal is supplied to the individual cathodes.



But, seven segment display does not work by directly supplying voltage to different segments of LEDs. First, our decimal number is changed to its BCD equivalent signal then BCD to seven segment decoder converts that signals to the form which is fed to seven segment display.

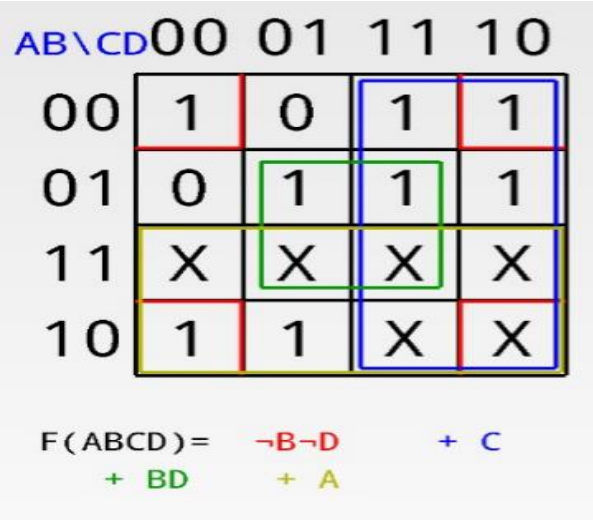
This BCD to seven segment decoder has four input lines (A, B, C and D) and 7 output lines (a, b, c, d, e, f and g), this output is given to seven segment LED display which displays the decimal number depending upon inputs.

Truth table:

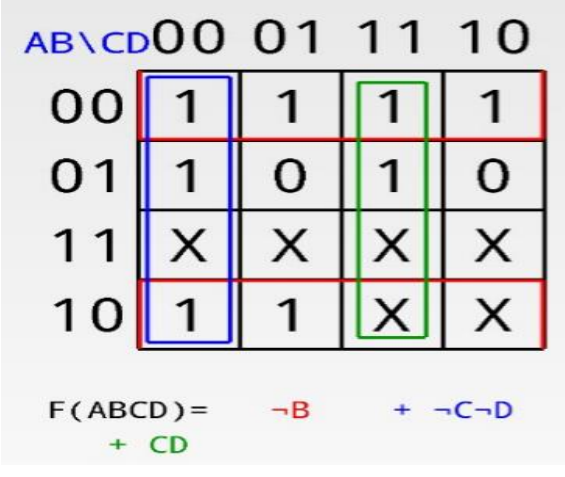
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

K maps:

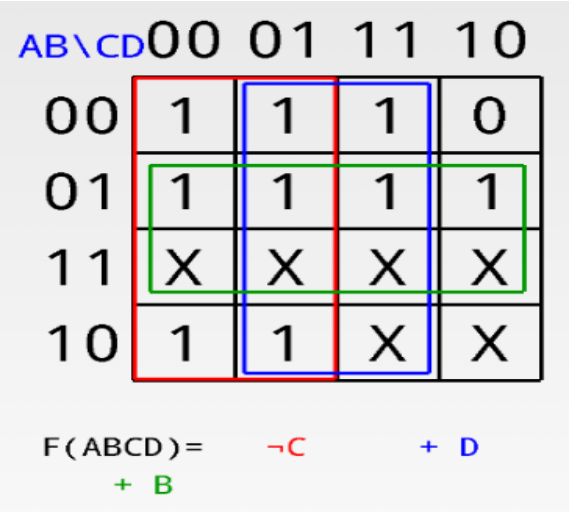
For a:



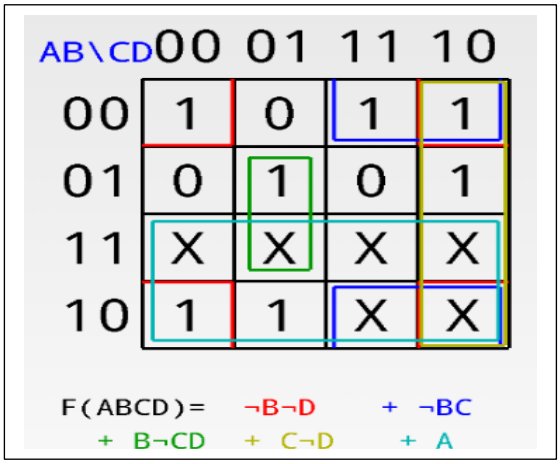
For b:



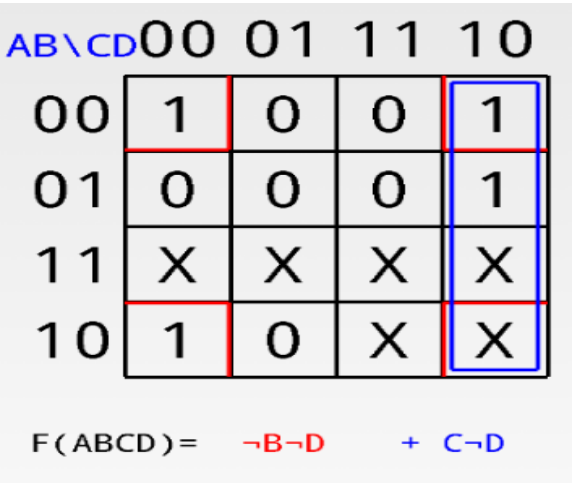
For c:



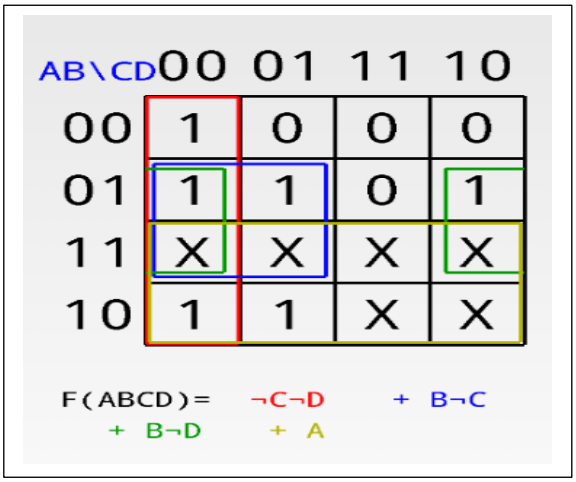
For d:



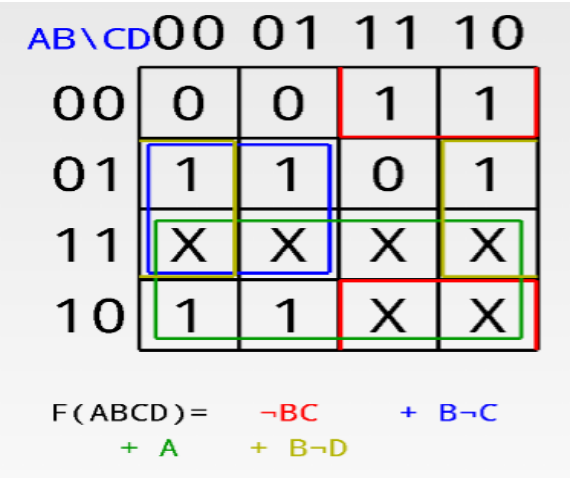
For e:

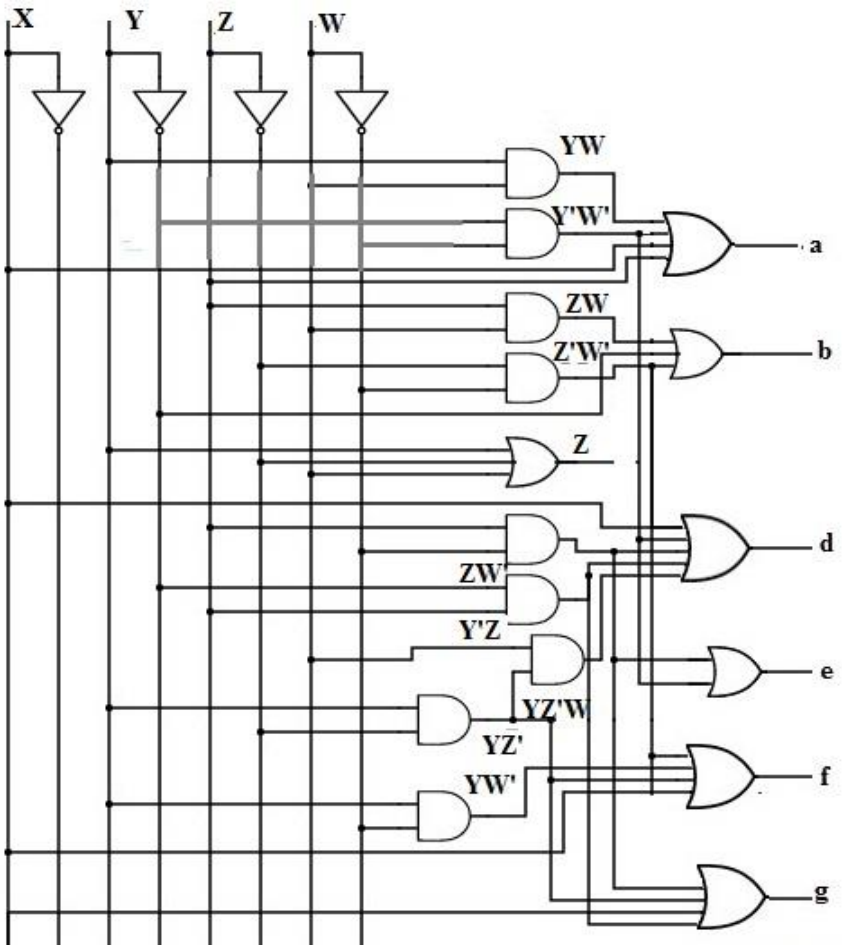


For f:



For g:





BCD to 7 segment Decoder circuit

Applications:

Seven-segment displays are used to display the digits in calculators, clocks, various measuring instruments, digital watches and digital counters.