**Experiment – 1:  To Study and Verify the Truth Table of Various Logic Gates  and Realize Boolean Expressions Using Gates.**

              Date: 30th July 2020

**1.  Aim 1. Aim:**

1. To study logic gates & verify the truth tables.

2. Verify the logical expression using truth table.

**2.  Requirements**

Java, Simulator to use Virtual lab

**3.  Pre-Experiment Exercise**

**Brief Theory**

1. **Basic Gates**

**1. AND gate**

   The output Q is true if input A AND input B are both true: **Q = A AND B**   
   An AND gate can have two or more inputs, its output is true if all inputs   are  true.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| traditional AND gate symbol | |  |  |  | | --- | --- | --- | | Input A | Input B | Output Q | | 0 | 0 | 0 | | 0 | 1 | 0 | | 1 | 0 | 0 | | 1 | 1 | 1 | |
| Traditional symbol | Truth Table |

**2. OR gate**

The output Q is true if input A OR input B is true (or both of them are true): **Q = A OR B**. An OR gate can have two or more inputs, its output is true if at least one input is true.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| traditional OR gate symbol | |  |  |  | | --- | --- | --- | | Input A | Input B | Output Q | | 0 | 0 | 0 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 1 | |
| Traditional symbol | Truth Table |

**3. EX-OR (EXclusive-OR) gate**

The output Q is true if either input A is true OR input B is true, **but not when both of them are true**: **Q = (A AND NOT B) OR (B AND NOT A)**.This is like an [OR](http://www.kpsec.freeuk.com/gates.htm#or#or) gate but excluding both inputs being true. The output is true if inputs A and B are **DIFFERENT**.   
EX-OR gates can only have 2 inputs.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| traditional EX-OR gate symbol | |  |  |  | | --- | --- | --- | | Input A | Input B | Output Q | | 0 | 0 | 0 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 0 | |
| Traditional symbol | Truth Table |

**4. EX-NOR (EXclusive-NOR) gate**

This is an EX-OR gate with the output inverted, as shown by the 'o' on the output.   
The output Q is true if inputs A and B are the **SAME** (both true or both false): **Q = (A AND B) OR (NOT A AND NOT B)**. EX-NOR gates can only have 2 inputs.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| traditional EX-NOR gate symbol | |  |  |  | | --- | --- | --- | | Input A | Input B | Output Q | | 0 | 0 | 1 | | 0 | 1 | 0 | | 1 | 0 | 0 | | 1 | 1 | 1 | |
| Traditional symbol | Truth Table |

**5. NOT gate (inverter)**

 The output Q is true when the input A is NOT true, the output is the inverse of the input: **Q = NOT A**.   A NOT gate can only have one input. A NOT gate is also called an inverter.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| traditional NOT gate symbol |  | |  |  | | --- | --- | | Input A | Output Q | | 0 | 1 | | 1 | 0 | |
| Traditional symbol |  | Truth Table |

**Universal Gates**

**1. NOR gate (NOR = Not OR)**

This is an OR gate with the output inverted, as shown by the 'o' on the output.   
The output Q is true if NOT inputs A OR B are true: **Q = NOT (A OR B)**   
A NOR gate can have two or more inputs, its output is true if no inputs are true.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| traditional NOR gate symbol | |  |  |  | | --- | --- | --- | | Input A | Input B | Output Q | | 0 | 0 | 1 | | 0 | 1 | 0 | | 1 | 0 | 0 | | 1 | 1 | 0 | |
| Traditional symbol | Truth Table |

**2. NAND gate (NAND = Not AND)**

This is an AND gate with the output inverted, as shown by the 'o' on the output.   
The output is true if input A AND input B are NOT both true: **Q = NOT (A AND B)**   
A NAND gate can have two or more inputs, its output is true if NOT all inputs are true.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| traditional NAND gate symbol | |  |  |  | | --- | --- | --- | | Input A | Input B | Output Q | | 0 | 0 | 1 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 0 | |
| Traditional symbol | Truth Table |

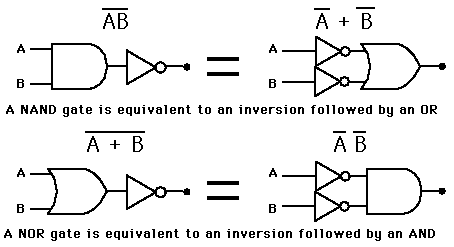
1. **De Morgan's Theorem: following are the expressions for De Morgan’s Law.**

|  |  |
| --- | --- |
| demorgan |  |

The most important [logic theorem](http://hyperphysics.phy-astr.gsu.edu/hbase/electronic/diglog.html#c2) for digital electronics, this theorem says that any logical binary expression remains unchanged if we

1. Change all variables to their complements.
2. Change all AND operations to ORs.
3. Change all OR operations to ANDs.
4. Take the complement of the entire expression.

A practical operational way to look at De Morgan's Theorem is that the inversion bar of an expression may be broken at any point and the operation at that point replaced by its opposite (i.e., AND replaced by OR or vice versa). Two forms of [De Morgan's Theorem](http://hyperphysics.phy-astr.gsu.edu/hbase/electronic/demorgan.html#c1#c1) implemented with [basic gates](http://hyperphysics.phy-astr.gsu.edu/hbase/electronic/gate.html#c1).



**Truth Table for De Morgan’s Theorem( NAND = Inversion followed by an OR)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** |  |  |  |  |
| **0** | **0** | **1** | **1** | **1** | **1** |
| **0** | **1** | **1** | **0** | **1** | **1** |
| **1** | **0** | **0** | **1** | **1** | **1** |
| **1** | **1** | **0** | **0** | **0** | **0** |

**Truth Table for De Morgan’s Theorem(  NOR  =  inversion followed by an AND)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** |  |  |  |  |
| **0** | **0** | **1** | **1** | **1** | **1** |
| **0** | **1** | **1** | **0** | **0** | **0** |
| **1** | **0** | **0** | **1** | **0** | **0** |
| **1** | **1** | **0** | **0** | **0** | **0** |

**Questions**

1. What are Universal gates? Why they are called as universal gates?

An universal gate is a gate which can implement any Boolean function without need to use any other gate type. NAND and NOR gates are called universal gates because they can perform all the three basic logic functions OR, AND and NOT.

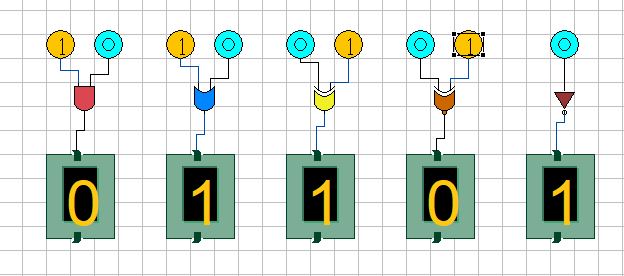
                2. Explain laws of Boolean algebra.

A set of rules or Laws of Boolean Algebra expressions have been invented to help reduce the number of logic gates needed to perform a particular logic operation resulting in a list of functions or theorems known commonly as the Laws of Boolean Algebra.

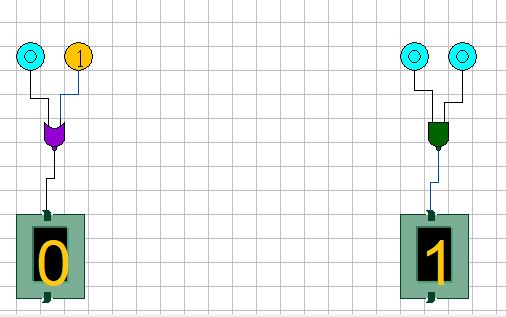
|  |  |
| --- | --- |
| Boolean Expression | Name of Law of Boolean Algebra |
| A.B=B.A  A+B=B+A | Commutative |
| (A.B).C=A.(B.C)  (A+B)+C=A+(B+C) | Associative |
| ~(A+B)=~A.~B  ~(A.B)=~A+~B | De Morgan’s Law |
| A+(B.C)=(A+B).(A+C)  A.(B+C)=(A.B)+(A.C) | Distributive |
| A+A=A  A.A=A | Idempotent |
| A+~A=T  A.~A=F  ~F=T  ~T=F | Complement |
| A+0=A  A.1=A | Identity |
| ~(~A)=A | Negation |
| P+(P.Q)=P  P.(P+Q)=P | Absorption |

**Outputs:**

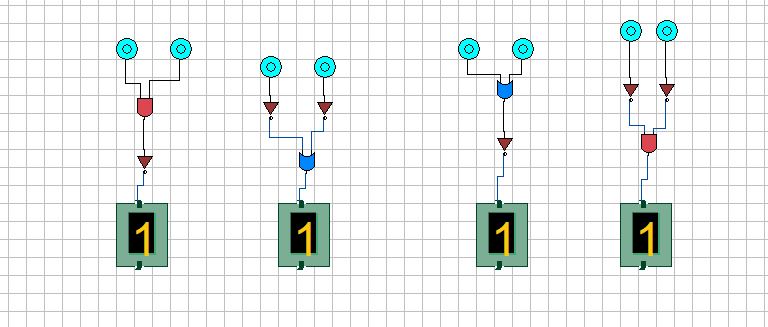
**Logic Gates:**



**Universal Gates:**



**De Morgan’s Law:**



**Experiment – 2: Realization of Basic Gates using Universal Gates**

              Date: 6th August 2020

**1.  Aim:** Verify the truth tables using Universal gates and Basic gates.

**2.  Requirements**

JAVA, Simulator

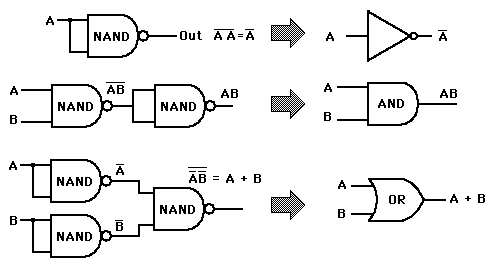
**3.  Pre-Experiment Exercise**

**Brief Theory**

1. **Using Universal Gates:** All the basic gates (e.g. AND, OR, Not etc.) can be implemented using universal gates (NAND, NOR).

**A) NAND Gate Operations**

The [NAND gate](http://hyperphysics.phy-astr.gsu.edu/hbase/electronic/nand.html#c1#c1) is called a universal gate because combinations of it can be used to accomplish all the basic functions.



**Truth Table for NOT Using NAND**

|  |  |  |
| --- | --- | --- |
| **A** |  |  |
| **0** | **1** | **1** |
| **1** | **0** | **0** |

**Truth Table for AND Using NAND**

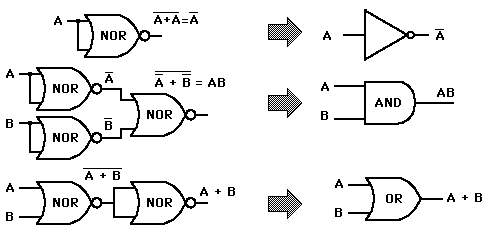
|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **B** |  | **AB** |
| **0** | **0** | **1** | **0** |
| **0** | **1** | **1** | **0** |
| **1** | **0** | **1** | **0** |
| **1** | **1** | **0** | **1** |

**Truth Table for OR Using NAND**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** |  |  |  | **A+B** |
| **0** | **0** | **1** | **1** | **0** | **0** |
| **0** | **1** | **1** | **0** | **1** | **1** |
| **1** | **0** | **0** | **1** | **1** | **1** |
| **1** | **1** | **0** | **0** | **1** | **1** |

**B) NOR Gate Operations**

The [NOR gate](http://hyperphysics.phy-astr.gsu.edu/hbase/electronic/nor.html#c1#c1) is called a universal gate because combinations of it can be used to accomplish all the basic functions.



**Truth Table for NOT Using NOR**

|  |  |  |
| --- | --- | --- |
| **A** | **A+A** |  |
| **0** | **1** | **1** |
| **1** | **0** | **0** |

**Truth Table for AND Using NOR**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** |  |  |  | **AB** |
| **0** | **0** | **1** | **1** | **0** | **0** |
| **0** | **1** | **1** | **0** | **0** | **0** |
| **1** | **0** | **0** | **1** | **0** | **0** |
| **1** | **1** | **0** | **0** | **1** | **1** |

**Truth Table for OR Using NOR**

|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **B** | **A+B** | **A+B** |
| **0** | **0** | **0** | **0** |
| **0** | **1** | **1** | **1** |
| **1** | **0** | **1** | **1** |
| **1** | **1** | **1** | **1** |

**4.  Laboratory Exercise**

**A) Procedure**

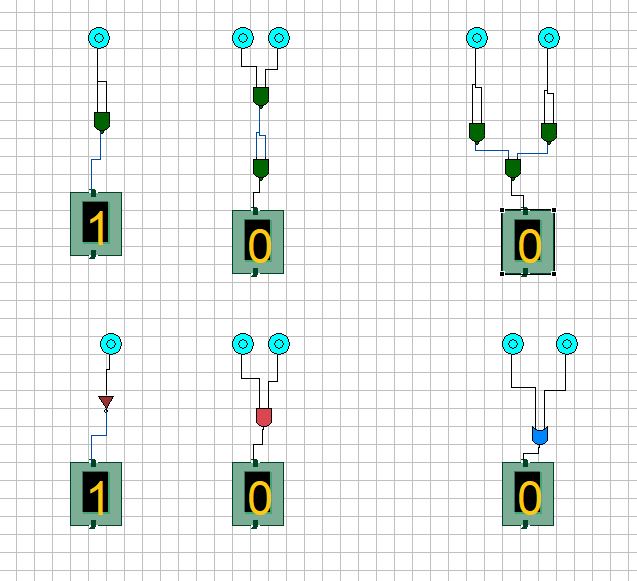
..

1. **Observations Table**

**Universal Gates Operations:**

**1. NAND Gate Operations**

1. **Not Using NAND**
2. **AND Using NAND**
3. **OR Using NAND**

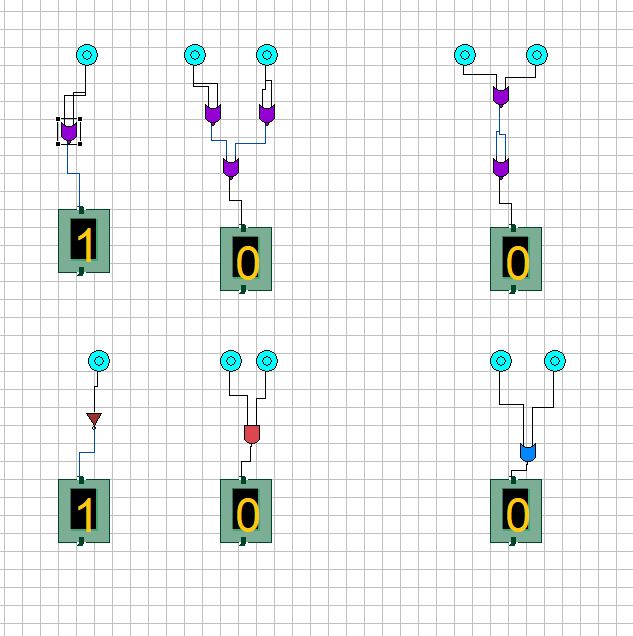


**2. NOR Gate Operations:**

**a) Not Using NOR**

**b) AND Using NOR**

**c) OR Using NOR**

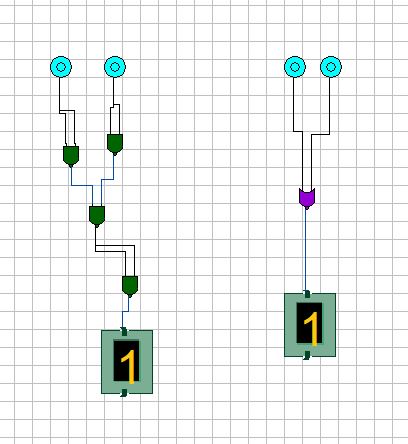


**5.  Post-Experiment Exercise**

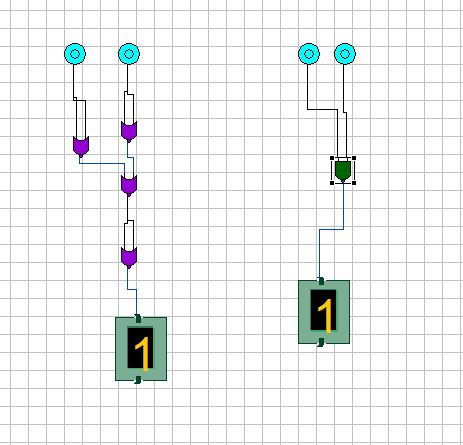
**A) Conclusion/Comments**

**B) Questions**

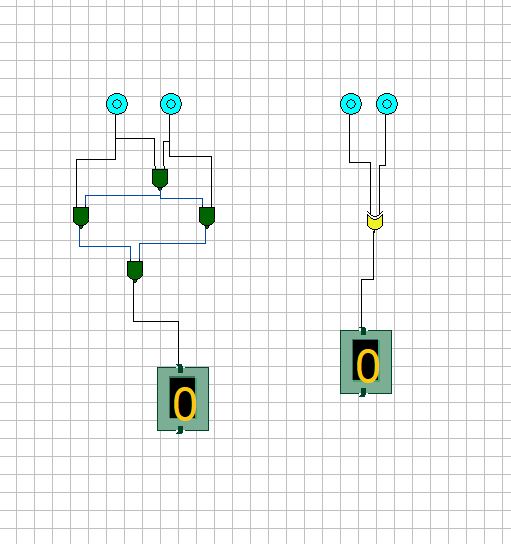
1. Implement NOR function by using NAND gates.



1. Implement NAND function by using NOR gates.



1. Implement Ex-OR function using Universal gates.



**Experiment – 3: To Realize Binary to Gray Code and Gray Code to Binary Converter.**

              Date: 16th October 2020

**1.  Aim:** Design and implement code conversion circuit

 i) Binary to gray code converter

             ii) Gray to binary converter

1. **Requirements**

JAVA, Simulator

**3.  Pre-Experiment Exercise**

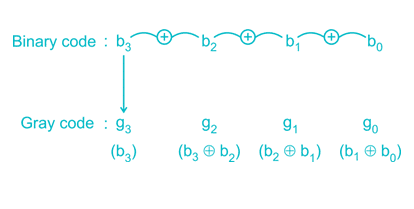
**Brief Theory**

**Gray Code** is one of the most important codes. It is a non-weighted code which belongs to a class of codes called minimum change codes. In this codes while traversing from one step to another step only one bit in the code group changes.

## Conversion from Binary to Gray code: **Following are the steps for converting binary into gray code.**

1. The left most significant bit of given binary code number is same as most left significant bit of gray number

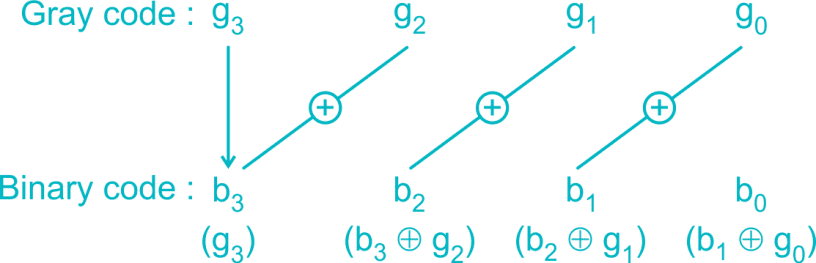
2. To obtain the successive gray bits for given binary code, add the first bit of binary number to second one and write down the result next to the first bit, add second bit and repeat the same operation until last bit

****

## Conversion from Gray to Binary code: **Following are the steps for converting gray to binary code.**

1. The left most significant bit of given gray code number is same as most left significant bit of binary number

2. To obtain the successive  binary bits for given gray code, add the second bit of gray code number to first bit of binary and write down the result next to the first bit and repeat the same operation until last bit



**Truth Table (Binary to Gray Code)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Binary code input** | | | | **Gray code output** | | | |
| B3 | B2 | B1 | B1 | G3 | G2 | G1 | G0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

**Truth Table (Gray to Binary Code)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Gray code input** | | | | **Binary code output** | | | |
| G3 | G2 | G1 | G0 | B3 | B2 | B1 | B1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

**4.  Laboratory Exercise**

**A) Procedure**

**B) Observation Table**

**Binary to Gray Code**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Binary Code Input** | | | | **Gray Code Output(Logic State)** | | | |
| B3 | B2 | B1 | B0 | G3 | G2 | G1 | G0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

**Gray to Binary Code**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Gray Code Input** | | | | **Binary Code Output(Logic State)** | | | |
| G3 | G2 | G1 | G0 | B3 | B2 | B1 | B0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

**C) Code**

**Grey To Binary:**

#include<stdio.h>

#include<conio.h>

void main()

{

int a[10],i=0,c=0,n;

printf("\n enter the gray code");

scanf("%d",&n);

while(n!=0)

{

a[i]=n%10;

n/=10;

i++;

c++;

}

for(i=c-1;i>=0;i--)

{

if(a[i]==1)

{

if(a[i-1]==1)

a[i-1]=0;

else

a[i-1]=1;

}

}

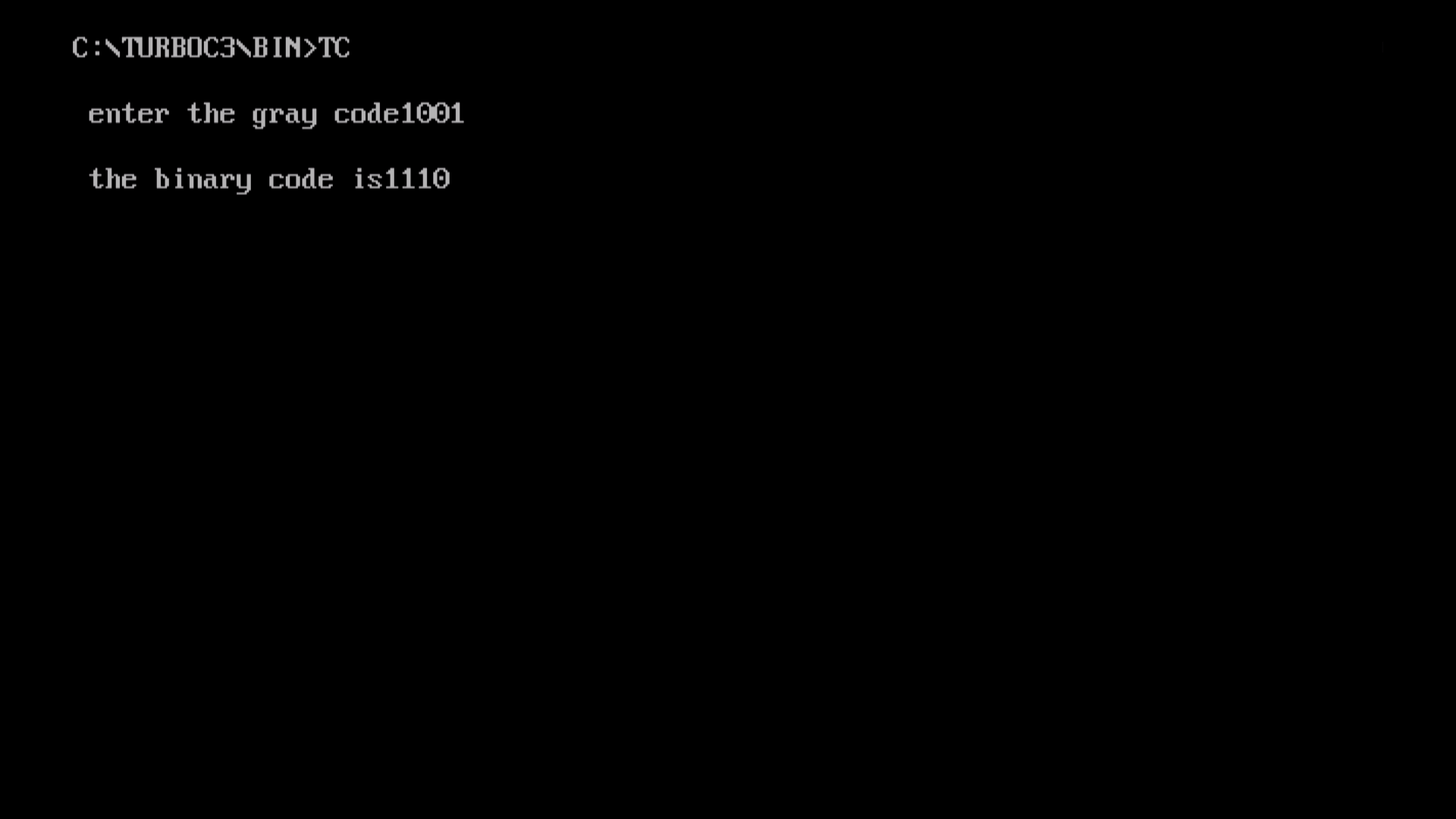
printf("\n the binary code is");

for(i=c-1;i>=0;i--)

printf("%d",a[i]);

getch();

}



**Binary to Gray Code:**

#include<stdio.h>

#include<conio.h>

void main()

{

int a[10],i=0,c=0,n,b[10];

printf("\n enter the binary code");

scanf("%d",&n);

while(n!=0)

{

a[i]=n%10;

n/=10;

i++;

c++;

}

for(i=c-1;i>=0;i--)

{

b[i]=a[i];

}

for(i=c-1;i>=0;i--)

{

if(b[i]==1)

{

if(a[i-1]==1)

a[i-1]=0;

else

a[i-1]=1;

}

}

printf("\n the gray code is");

for(i=c-1;i>=0;i--)

printf("%d",a[i]);

getch();

}



**5.  Post-Experiment Exercise**

1. **Conclusion/Comments**

 From the above experiment we are able to convert binary code to grey code and vice

versa. We also learn that the EX-OR condition is necessary for these conversions.

**B) Questions**

1. What is gray code? Why is it called as unit distance code?

Gray Code is a non-weighted code which belongs to a class of

codes called minimum change codes. In this codes while traversing

from one step to another step only one bit in the code group

changes. Adjacent codes differ only in one bit. Therefore they are known as grey codes

1. Write short note on application of gray code (Shaft encoder).

Gray codes are widely used to prevent spurious output from electromechanical switches and to facilitate error correction in digital communications such as digital terrestrial television and some cable TV systems. The Gray Code is used to eliminate the error problem which is inherent in the binary code. The gray code assures that only one bit will change between adjacent sectors.

1. Represent (35)10 in gray code.

(35)10=(110010)

**Experiment4: To design and verify the circuit of i) Half Adder ii) Full Adder**

              Date: 16th October 2020

**1.  Aim** 1.Design and verify the circuit for following:

                i) Half Adder and Full Adder.

**2.  Requirements**

Java, Simulator

**3.  Pre-Experiment Exercise**

**Brief Theory**

## Types of adders: **For single bit adders, there are two general types.**

A **half adder** has two inputs, generally labelled *A* and B, and two outputs, the [sum](http://en.wikipedia.org/wiki/Sum) S and [carry](http://en.wikipedia.org/wiki/Carry_flag) C. S is the two-bit [XOR](http://en.wikipedia.org/wiki/XOR_gate) of A and B, and C is the [AND](http://en.wikipedia.org/wiki/AND_gate) of A and B. Essentially the output of a half adder is the sum of two one-bit numbers, with C being the most significant of these two outputs.

A **full adder** has three inputs - *A*, *B*, and a carry in *C*, such that multiple adders can be used to add larger numbers. To remove ambiguity between the input and output carry lines, the carry in is labelled *Ci* or *Cin* while the carry out is labelled *Co* or *Cout*.

### Half adder: A half adder is a logical circuit that performs an addition operation on two binary digits. The half adder produces a sum and a carry value which are both binary digits. The drawback of this circuit is that in case of a multibit addition, it cannot include a carry.A half adder takes in two inputs A and B, adds, and returns the resulting sum S and a carry out C. Its logic table and Truth table is shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | | **Output** | |
| **A** | **B** | **CARRY(S)** | **SUM(S)** |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

### b) Full adder: Inputs: {A, B, CarryIn} → Outputs: {Sum, CarryOut}

A **full adder** is a logical circuit that performs an addition operation on three binary digits. The full adder produces a sum and carry value, which are both binary digits.  A full adder is more powerful than a half adder. Unlike a half adder, it can also take in a carry-in bit, and uses that in a calculation. A full adder takes in three inputs, Cin, A, and B, adds, and outputs the resulting sum S, and a carry out Cout. Its logic table and Truth table is shown below.

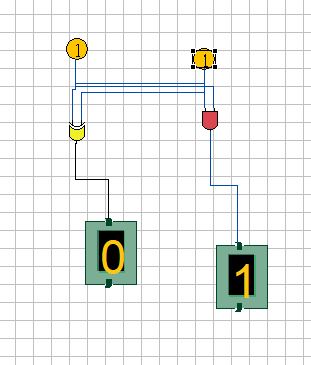
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | | | **Output** | |
| **A** | **B** | **CarryIn(Cin)** | **CarryOut(Cout)** | **Sum (S)** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

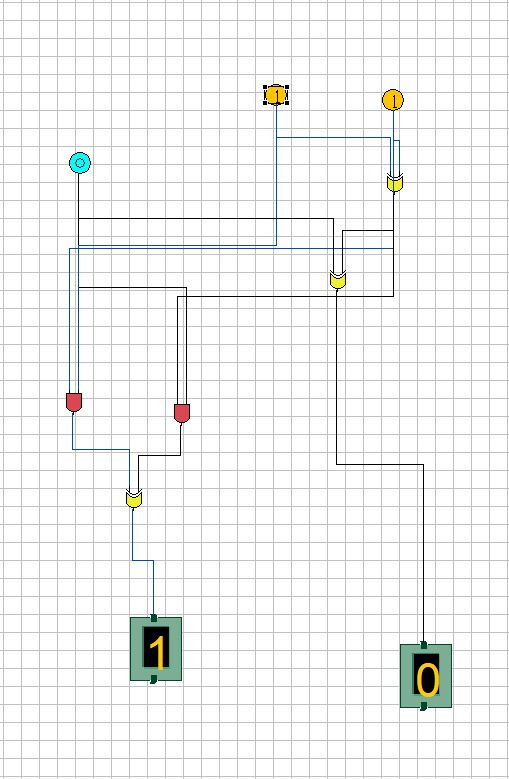
A full adder can be constructed from two half adders by connecting *A* and *B* to the input of one half adder, connecting the sum from that to an input to the second adder, connecting *Ci* to the other input and [OR](http://en.wikipedia.org/wiki/Logical_disjunction) the two carry outputs. Equivalently, *S* could be made the three-bit xor of *A*, *B*, and *Ci* and *Co* could be made the three-bit [majority function](http://en.wikipedia.org/wiki/Majority_function) of *A*, *B*, and *Ci*. The output of the full adder is the two-bit arithmetic sum of three one-bit numbers.

**A) Procedure**

**B) Implementation**

**Half Adder**

  
  
  
**Full Adder**

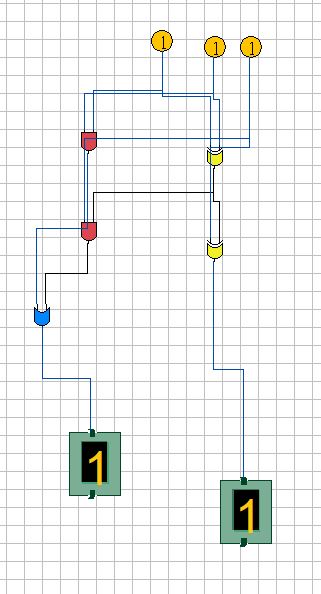


**Post-Experiment Exercise**

**A) Conclusion/Comments**

**B) Questions**

            1)Realize a full adder using two HALF adders and one OR gate.



2)Perform division of decimal 9 and 3 using binary arithmetic

(9)10 = (1001)2

(3)10 = (0011)2

1001 ➗ 11 = 0011

Quotient = 0011

Remainder = 0000

(0011)2= (3)10

3) Subtract using 1’s and 2’s complement method

a) (56)10 – (76)10

56 = 0011 1000

76 = 0100 1100

-76 = 1011 0011

+1

=1011 0100

0011 1000

+1011 0100

1110 1100

b)(15)10 – (21)10

15 = 01111

21 = 10101

-21 = 01010

+1

=01011

01111

+01011

11010