LAB MANUAL

Group Members:

1. ***Bisma Ali(2021-BSE-008)***
2. ***Laiba Sohail(2021-BSE-016)***
3. ***Neha Amjad(2021-BSE-024)***
4. ***Tanzeela Asghar(2021-BSE-032)***

Department:Software Engineer
Submitted To:Sir.Ahsen Ilyas

1. ***Date of Submission:*** *31,JANUARY, 2023*

Instructor signature: _____

Table of Contents

EXPERIMENT # 1

FAMILIARIZATION WITH LOGIC TRAINER AND VERIFICATION OF TRUTHTABLE OF BASIC LOGIC GATES

1. Objectives:

Having completed this experiment, you will be able to:

- **i.** Understand the different options, facilities and provisions provided on the logic trainer.
- **ii.** Recognize the different logic gates.
- **iii.** Verify the truth table of basic logic gates:

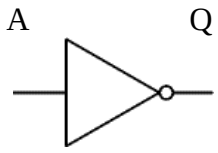
1.	AND	4081
2.	OR	4071
3.	NOT	4049
4.	NAND	4011
5.	NOR	4001
6.	X-OR	4070
7.	EX-NOR	4077

2. Basic Information:

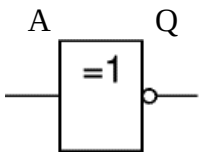
A logic gate is an elementary building block of a digital circuit. Most logic gates have two inputs and one output. At any given moment, every terminal is in one of the two binary conditions l*ow/false* (0) or *high/true* (1), represented by different voltage levels. The logic state of a terminal can, and generally does, change often, as the circuit processes data. In most logic gates, the low state is approximately zero volts (0 V), while the high state is approximately five volts positive (+5 V). There are seven logic gates: AND, OR, XOR, NOT, NAND, NOR, and XNOR.

a. Logic Gates Symbols and Truth Tables:

- **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.

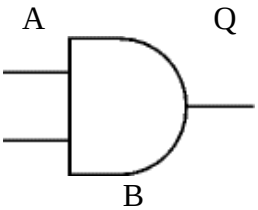


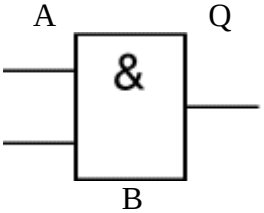
Traditional Symbol



IEC Symbol

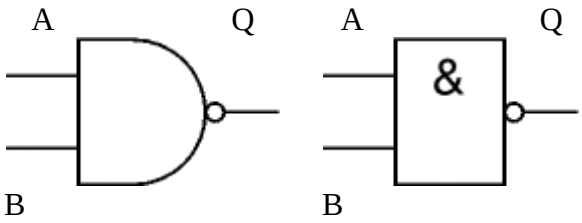
- **AND Gate:**
  The output Q is true if input A AND input B are both true: **Q = A AND B** An AND gate can have two or more inputs, its output is true if all inputs are true.





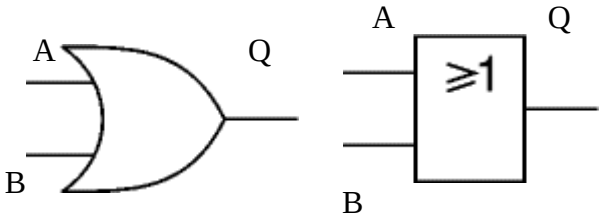
Traditional Symbol *IEC Symbol*

- **NAND gate (NAND = NOT of 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 Symbol *IEC Symbol*

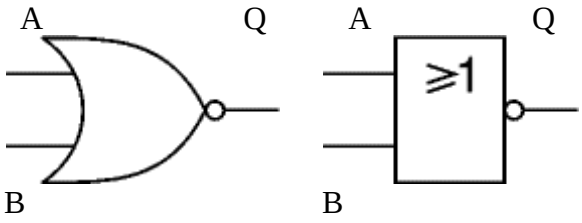
- **OR Gate:**
  The output Q is true if input A OR input B is true (or both of them are true): **Q = A OR B**. An OR gate can have two or more inputs, its output is true if at least one input is true.



Traditional Symbol *IEC Symbol*

- **NOR gate (NOR = NOT of OR):**

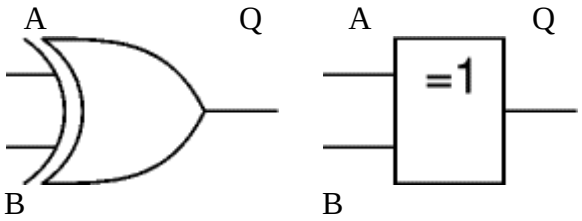
  This is an OR gate with the output inverted, as shown by the 'o' on the output. The output Q is true if NOT inputs A OR B are true: **Q = NOT (A OR B)**. A NOR gate can have two or more inputs, its output is true if no inputs are true.



Traditional Symbol *IEC Symbol*

- **EX-OR (EXclusive-OR) Gate:**

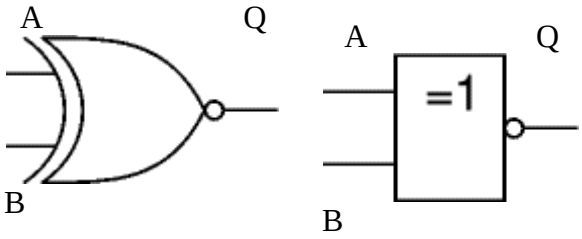
The output Q is true if either input A is true OR input B is true, **but not when both are true**: **Q = (A AND NOT B) OR (B AND NOT A)**. This is like an **OR** gate but excluding both inputs being true.
The output is true if inputs A and B are **DIFFERENT**. EX-OR gates can only have 2 inputs.



Traditional Symbol *IEC Symbol*

- **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 Symbol *IEC Symbol*

- **Universal Gates:**

The NAND and NOR gates can be said to be universal gates, since combinations of them can be used to accomplish any of the basic operations and can, thus produce an inverter, an OR gate or an AND gate. The non-inverting gates do not have this versatility since they can't produce an invert.

3. Experimental Work:

a. Material Required:

- Logic Trainer
- Connecting Wires
- Components (IC's)

b. Procedure:

- Connect the logic Trainer to 220V Ac power supply.
- Turn the Trainer On and verify the voltage of power supply by using voltmeter.
- Install the IC chip under experiment on the trainer breadboard.
- Wire the circuit according to the diagram provided; supply the +5V and ground to pin number 14 and 7 respectively.
- Use logic switches to provide "0" and "1" for input of each gate one at a time as A and B.

- For the output indication use LED Lout.
- Verify the output according to the truth tables of each gate.
- Fill the truth table in at step 3.4 according to the results obtained.
- Write down your observation & comments at step 3.4 as per your concept developed during this experimental work.

c. Experimental Results.

AND GATE (4081)

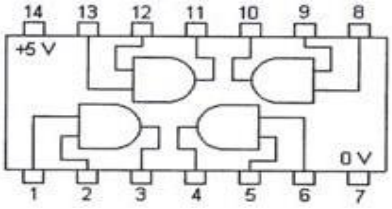
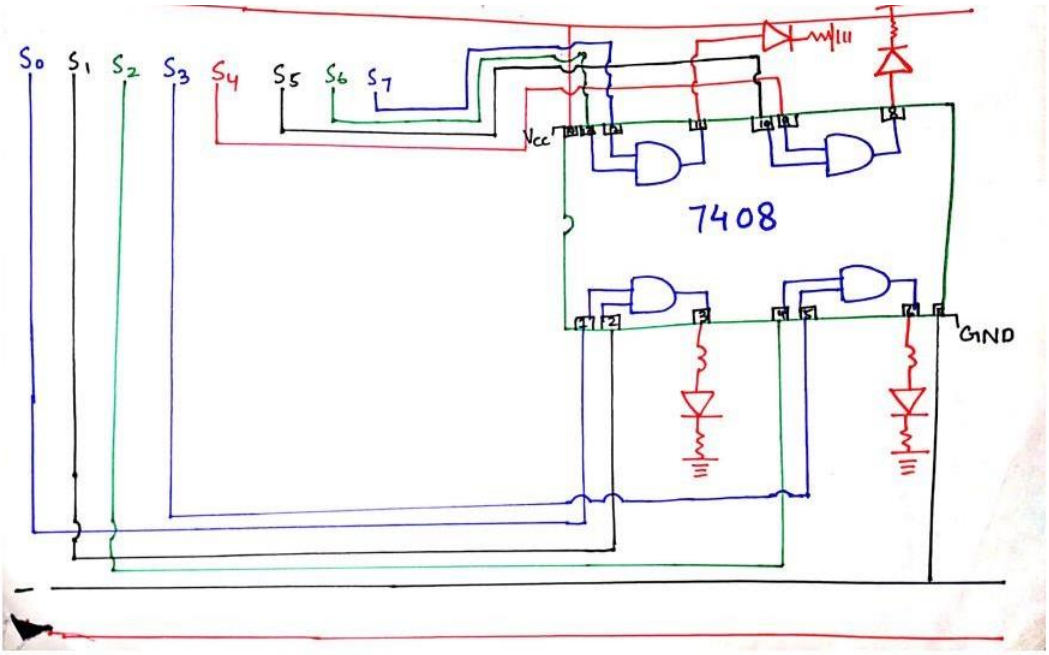


Figure 1 Schematic diagram of AND gate



TRUTH TABLE

Input A	Input B	Output Q
0	0	0
0	1	0
1	0	0
1	1	1

OR GATE (4071)

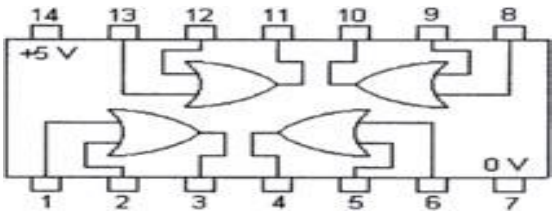
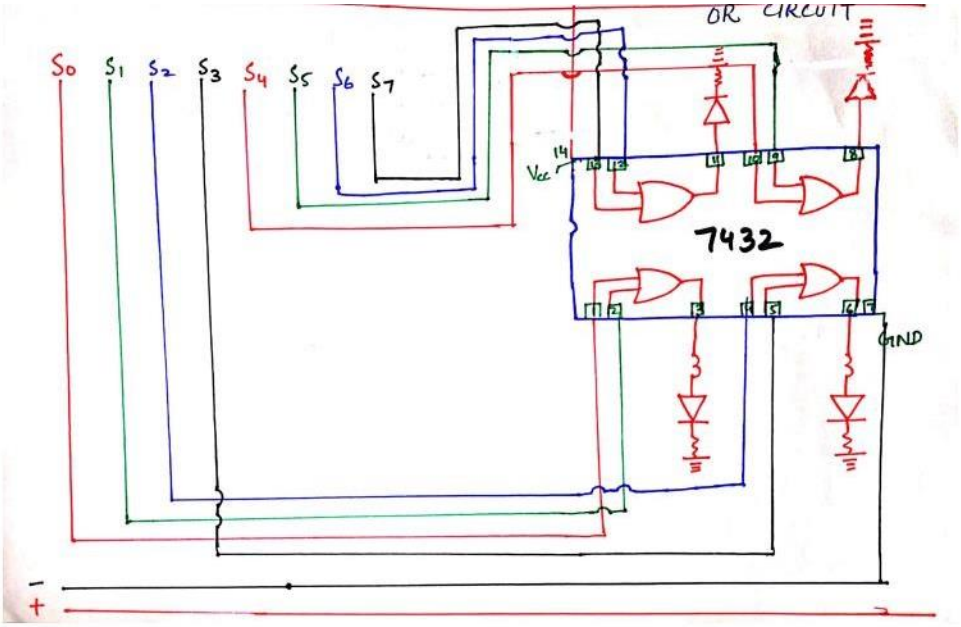


Figure 3 Schematic diagram of OR gate



TRUTH TABLE

Input A	Input B	Output Q
0	0	0
0	1	1
1	0	1
1	1	1

NAND GATE (4011)

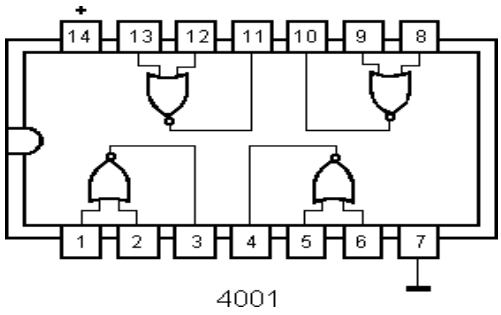
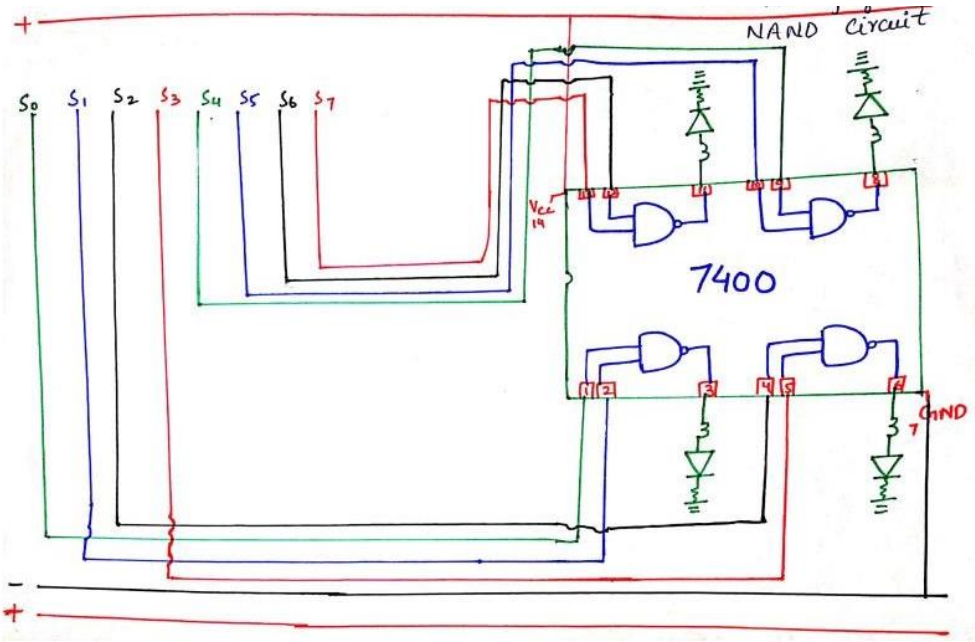


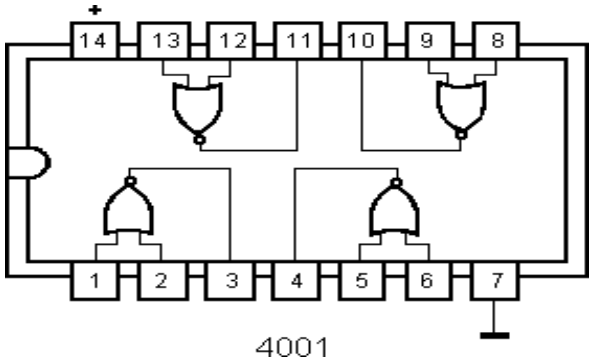
Figure 5 Schematic diagram of NAND gate



TRUTH TABLE

Input A	Input B	Output Q
0	0	1
0	1	1
1	0	1
1	1	0

NOR GATE (4001)



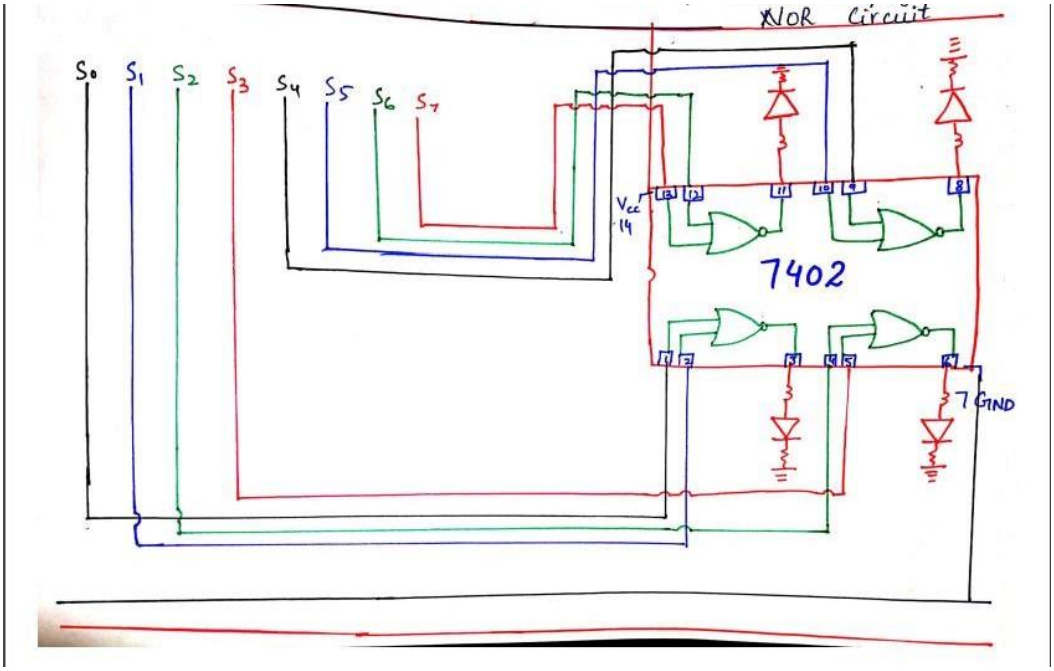


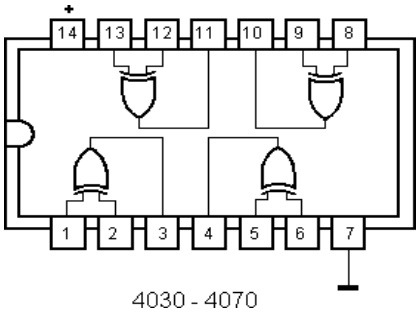
Figure 7 Schematic diagram of NOR gate

TRUTH TABLE

Input A	Input B	Output Q
0	0	1
0	1	0
1	0	0

1	1	0

X-OR GATE (4070)



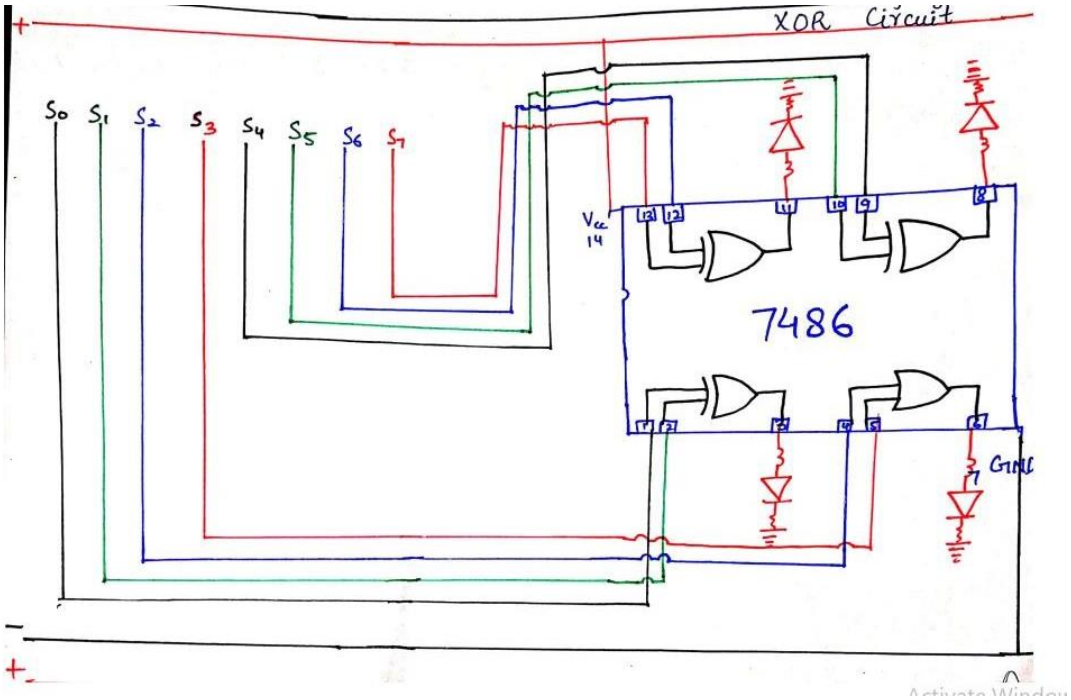
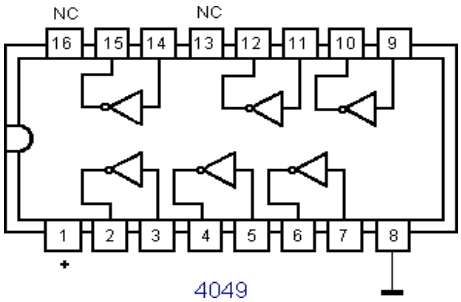


Figure 9 Schematic diagram of X-OR gate

TRUTH TABLE

Input A	Input B	Output Q
0	0	0
0	1	1
1	0	1
1	1	0

NOT GATE (4049)



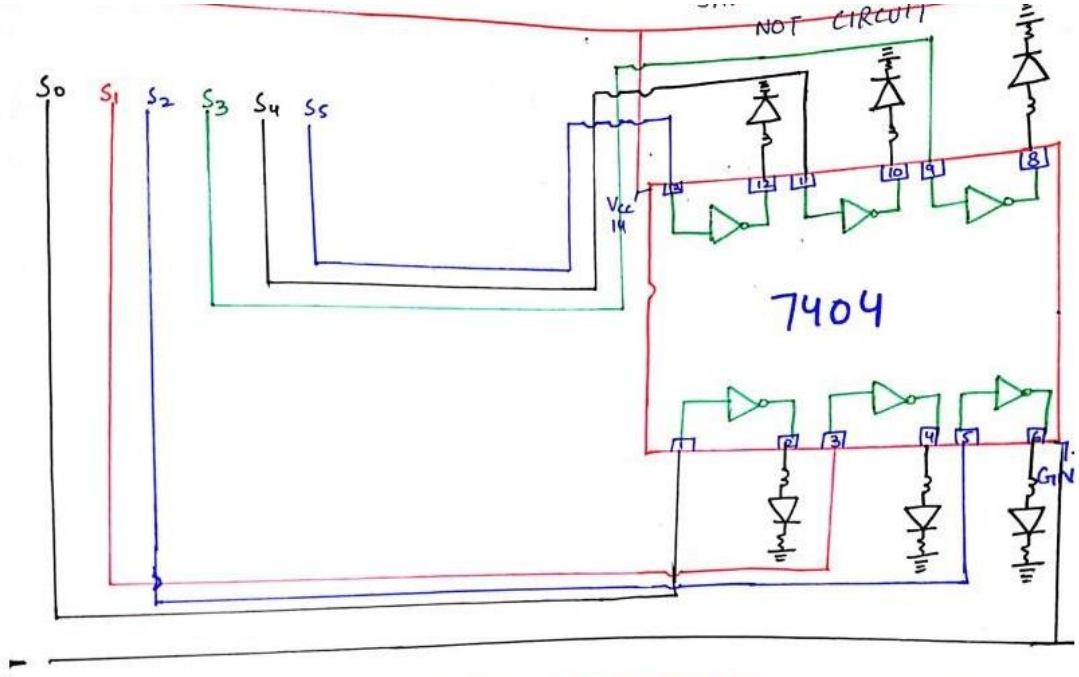


Figure 11 Schematic diagram of NOT gate

TRUTH TABLE

Input A	Output A'
0	1
1	0

In Case Of Trouble:

- Check the power supply.
- Check the Vcc and GND at pin number 14 and 7 of the IC under test.
- Check all the wire connections and remove the breaks.
- Check the IC under test using truth table.

EXPERIMENT # 2

WORKING WITH UNIVERSAL LOGIC GATES

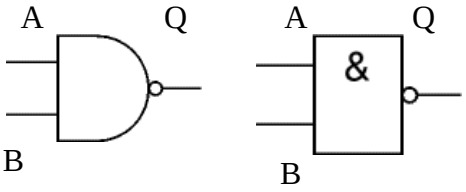
1. Objective:

After completion of this experiment Students shall be able to understand the behavior of logic NAND & NOR as the universal gates by obtaining the result like basic AND, OR, NOT gates.

2. Basic Information of Universal Gates:

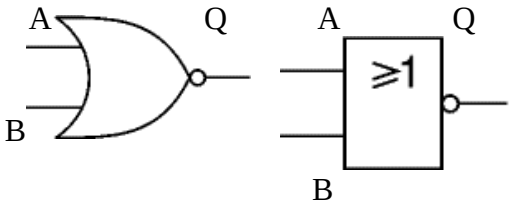
The NAND and the NOR gate can be said to be universal gates since combinations of them can be used to accomplish any of the basic operations and can thus produce an inverter, an OR gate or an AND gate. The non-inverting gates do not have this versatility since they can't produce an invert.

a. NAND Gate:



Traditional Symbol *IEC Symbol*

b. NOR Gate:



Traditional Symbol *IEC Symbol*

3. Experimental Work:

This Experiment has two parts.

Part-1: *Working with NAND Gate*

Part-2: *Working with NOR Gate*

3.1. Material Required:

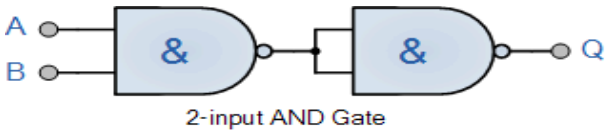
- Logic Trainer
- Connecting wires
- Components: 4011, 4001

3.2. Procedure:

- Install the ICs on trainer's breadboard.
- Wire the pins 14 &7 of IC's to +5v and GND respectively.
- Wire the circuit according to the diagrams provided in step 4.1 & 4.3 for NAND and NOR gates.

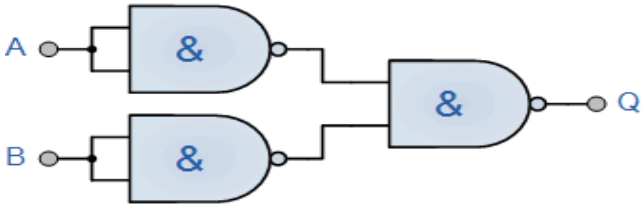
- Mention the input as "A" "B" and output as A', A+B, and AB with respective logic circuit.
- Write down the results of Boolean expression of the interconnected gates at step 4.2, 4.4 with respect to the basic logic gates in the table provided.

3.3. Working with NAND Gate:



2-input AND Gate

NOT Gate



2-input OR Gate

a. **Experimental Results:**

AND

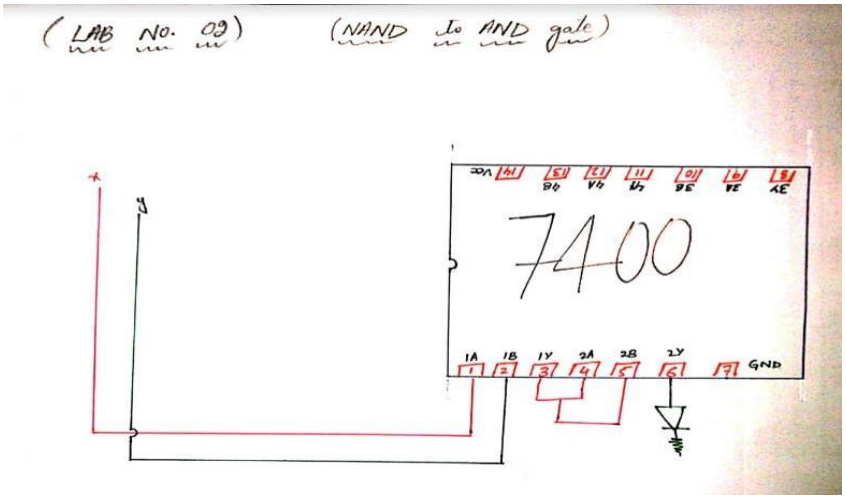


Figure 13 Schematic diagram of NAND to AND

TRUTH TABLE

Input A	Input B	Output AB
0	0	0
0	1	0
1	0	0
1	1	1

NOT

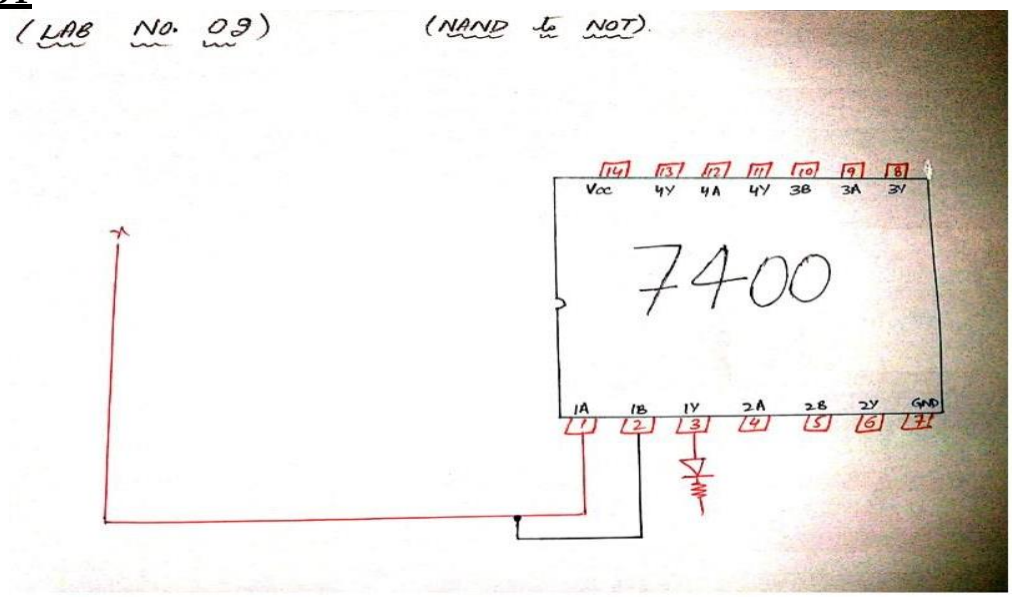


Figure 15 Schematic diagram of NAND to NOT

TRUTH TABLE

Input A	Output A’
0	1
1	0

OR

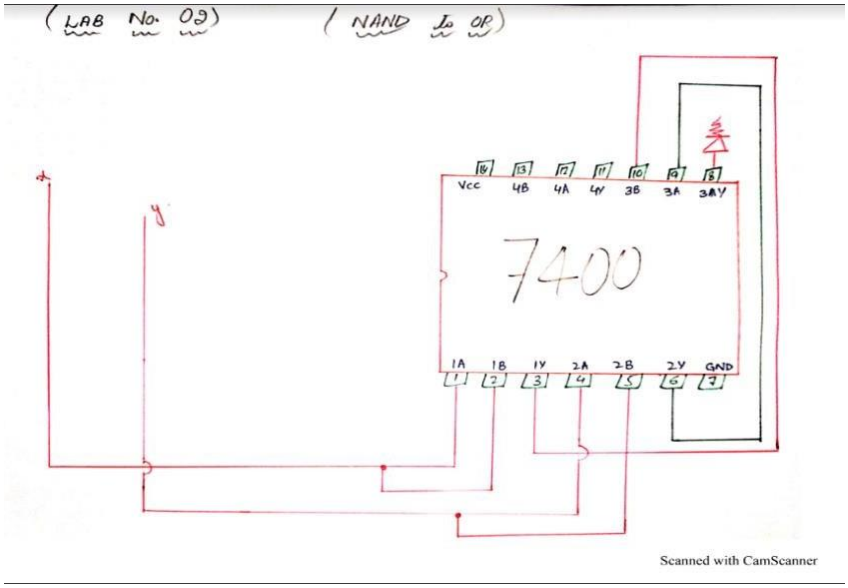
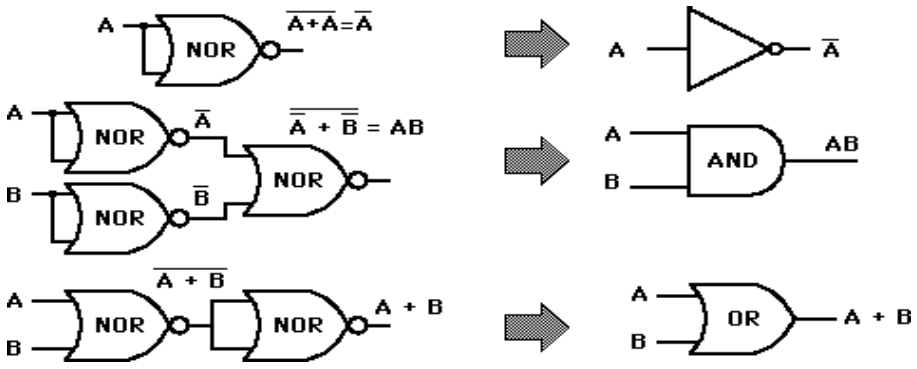


Figure 17 Schematic diagram of NAND to OR

TRUTH TABLE

Input A	Input B	Output AB
0	0	0
0	1	1
1	0	1
1	1	1

3.1. Working with NOR GATE:



a. <u>Experimental Results:</u>

AND

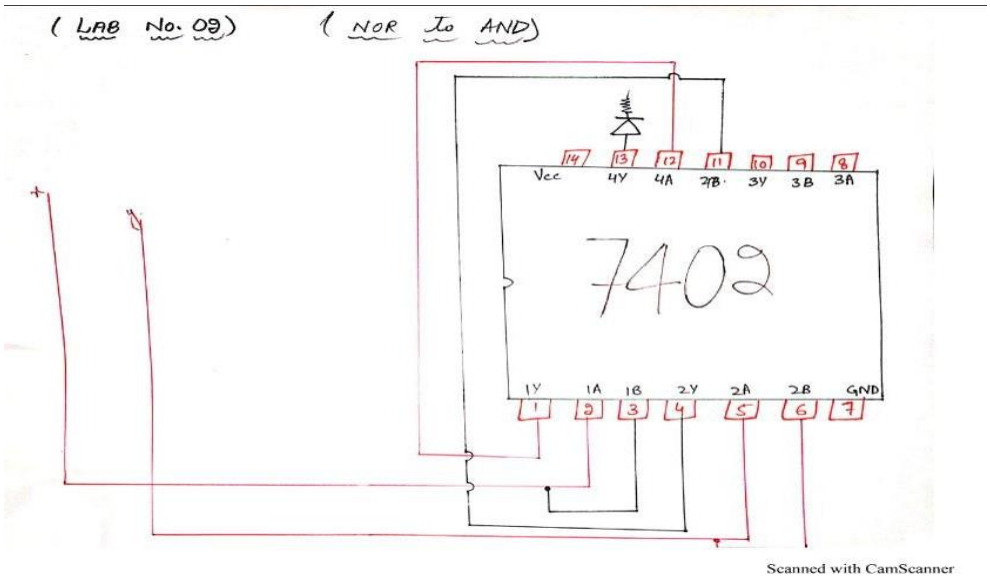


Figure 19 Schematic diagram of NOR to AND

TABLE

TRUTH

Input A	Input B	Output AB
0	0	0
0	1	0

1	**0**	**0**
1	**1**	**1**

NOT

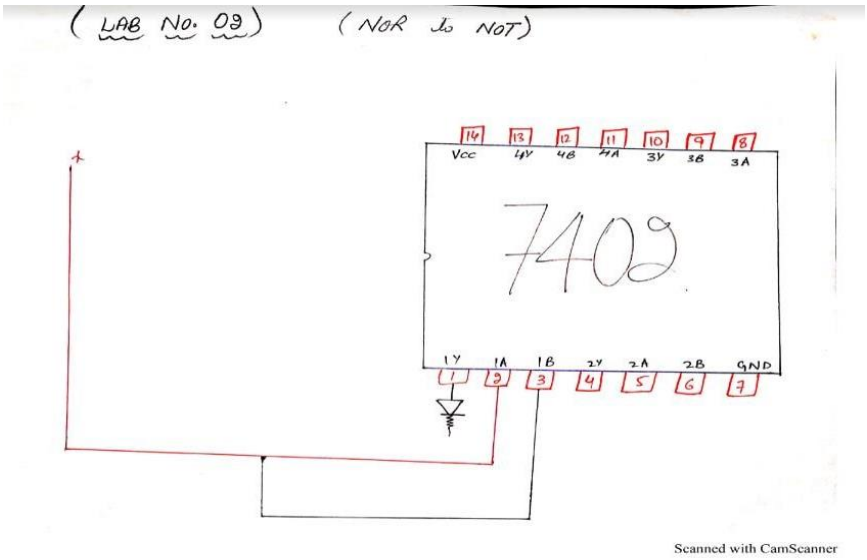


Figure 21 Schematic diagram of NOR to NOT

TRUTH TABLE

Input A	Output A’
0	**1**

1	**0**

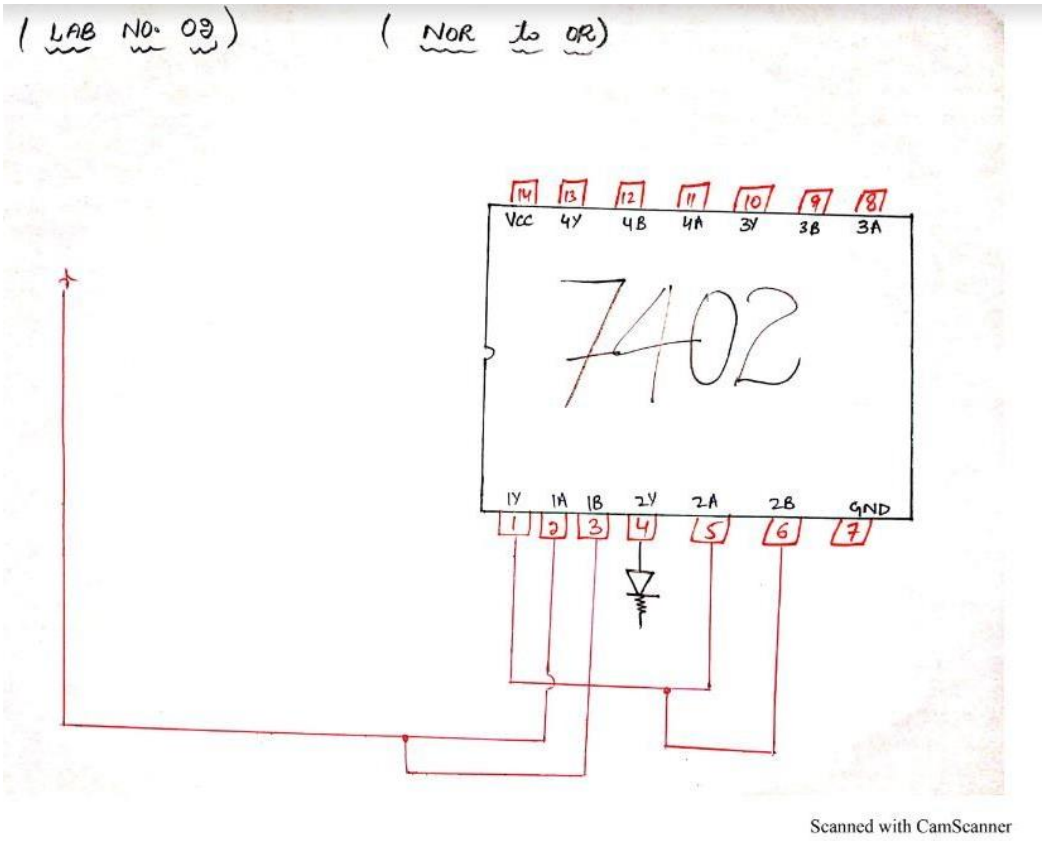


Figure 23 Tinkercad diagram of NOR to OR

TRUTH TABLE

Input A	Input B	Output AB
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

EXPERIMENT # 3(a)

IMPLEMENTATION OF BOOLEAN EXPRESSION THROUGH LOGIC GATES & ALSO VERIFICATION OF DEMORGAN'S LAW

1. Material Required:

- Logic Trainer
- Connecting Wires
- Power Supply
- Components: 4081, 4049, 4071

2. Procedure:

Consider as an example for the following Boolean function:

$$F1 = x + y'z$$

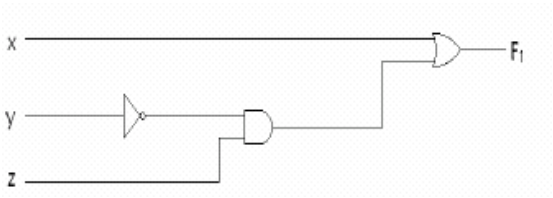
The function F1 is equal to 1 if x is equal to 1 or if both y' and z are equal to 1, F1 is equal to 0 otherwise. The complement operation dictates that when y'=1 then y=0. Therefore, we can say that F1=1 if x=1 or if y=0 and z=1. A Boolean function expresses the logical expression for all possible values of the variables.

A Boolean function can be represented in a truth table. A truth table is a lot of combinations of 1's and 0's assigned to the binary variables and a column that shows the value of the function for each binary combination. The number of rows in the truth table is 2n, where n is the number of variables in the function. The binary combinations for the truth table are obtained from binary numbers by counting from 0 through 2n-1. Following table shows the truth table for the function F1.

Truth Table for F1

Input X	Input Y	Input Z	Output F1
0	0	0	0
0	0	1	1

0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



Gate implementation of F1=x+y'z

Figure 24 Schematic diagram for F1

3. Verify Demorgan's Theorem

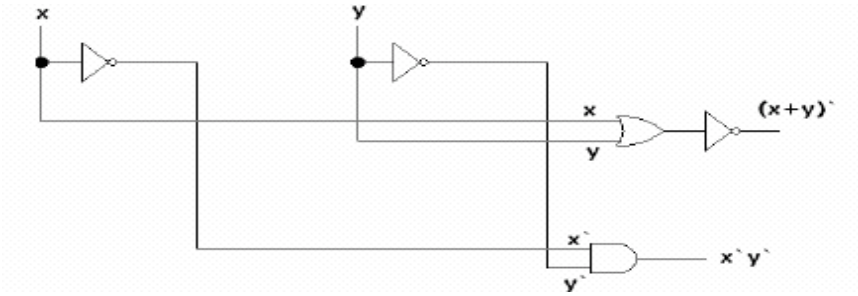
Demorgan's law can be stated in terms of logic terms, which is the 1st law states that,

$$(x+y)' = x'y' \qquad \text{Theorem 1}$$

And the second law state that

$$(xy)' = x' + y' \qquad \text{Theorem 2}$$

a) Truth Table that verifies the above given Theorem 1,



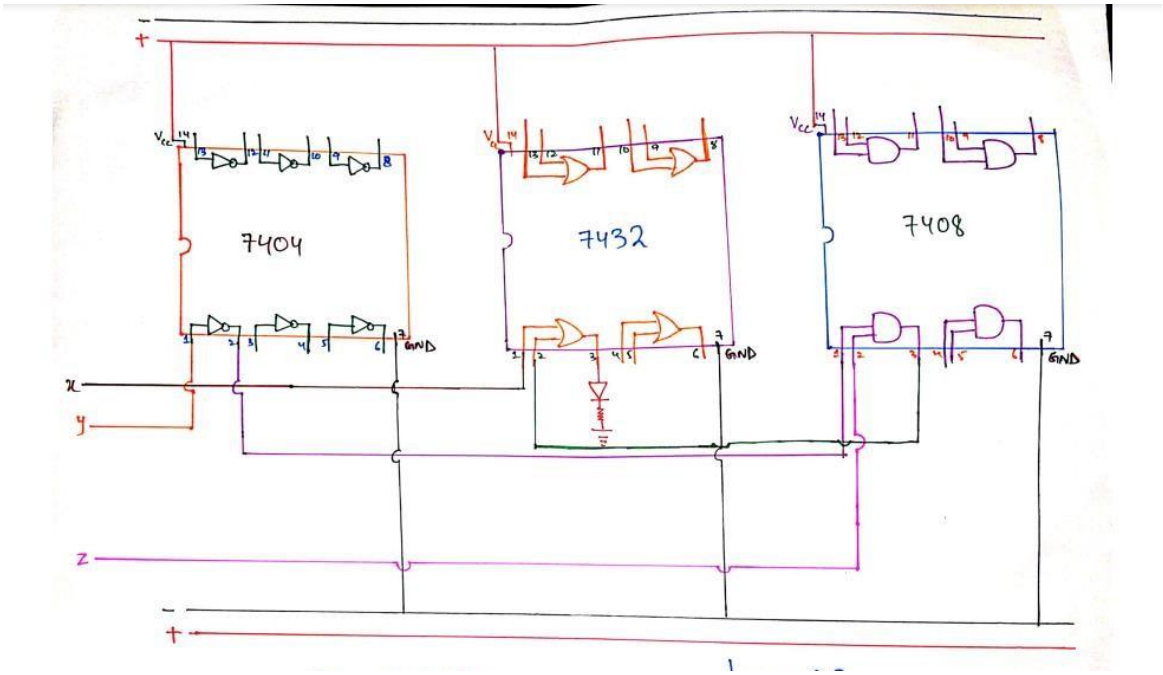


Figure 26 Schematic diagram for Demorgans law (Theorem 1)

b) Truth Table that verifies the above given Theorem 2

Input X	Input Y	(xy)’	X’+Y’
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

EXPERIMENT # 3(b)

IMPLEMENTATION OF XOR AND XNOR GATES USING NANDGATES

1. Material Required:

Logic Trainer
Connecting Wires
Components(IC’s): 4011

2. Procedure:

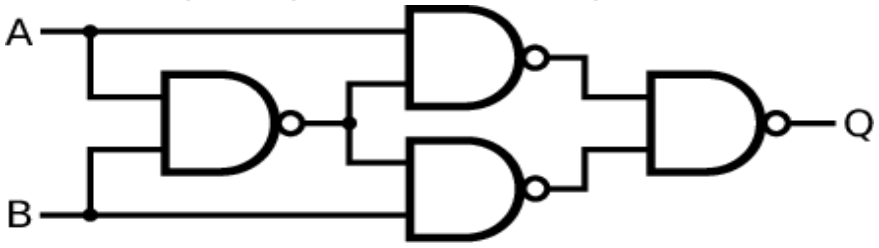
It is clear from the logic diagram that the NAND gate implementation of XOR gate requires five NAND gates. You will need two quad- 2 in NAND gate ICs to perform this experiment. Gets the required number of ICs containing NAND gates and other apparatus from the lab attendant. Plug in the ICs in the breadboard of the Logic Trainer. Connect 5Vdc power supply and ground on pins 14 and 7 respectively. For other pin configuration consult the data sheet (we have already used NAND gates in the first lab so it should not be a problem). Wire your circuit according to the logic diagram for XOR gate as given above. Once you have wired the circuit, check it with your instructor and, if approved, power up your circuit. The outputs should be connected to the LEDs on the Logic Trainer for monitoring purpose. Apply different input combinations at the input and note down the outputs and fill in the following truth table. This truth table should confirm to the one given in theory.

Repeat the same procedure for NAND gate implementation of XNOR gate.

3. EXPERIMENTAL RESULT

Fill in the following truth table in the presence of the lab instructor.

a. Draw NAND logic diagram for the XOR gate



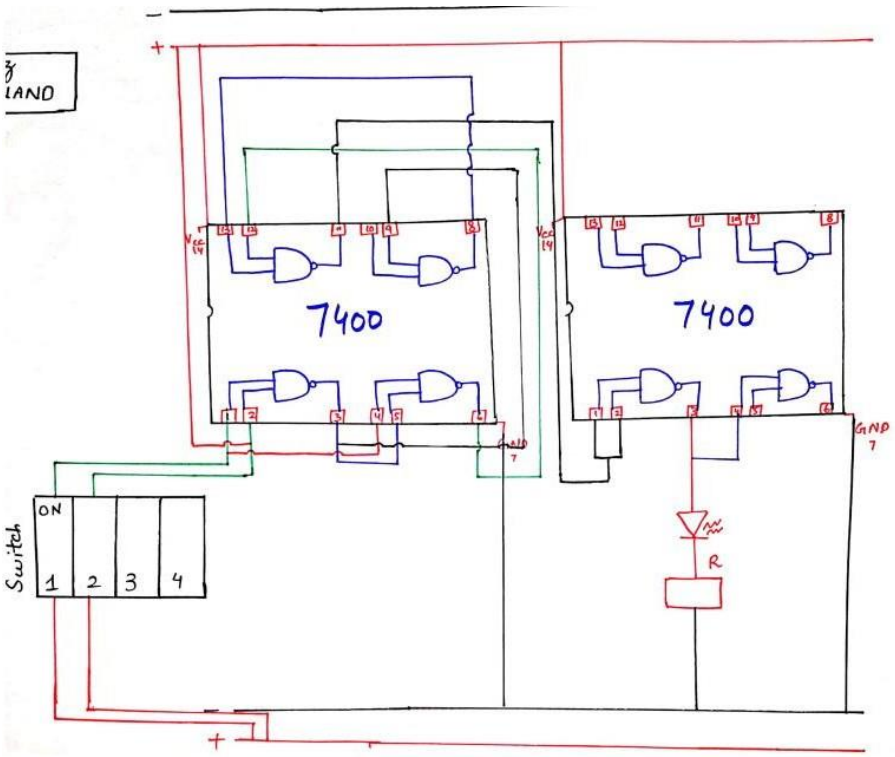


Figure 28 Schematic diagram of NAND to X-OR

b. Write Boolean function for XOR gate:

Q = (A'. B) + (A. B')

c. Truth Table for XOR Gate:

Input A	Input B	Output F
.0	0	0
0	1	1
1	0	1
1	1	0

Draw NAND logic diagram for the XNOR gate:

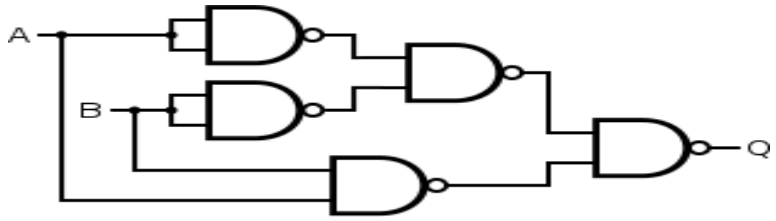
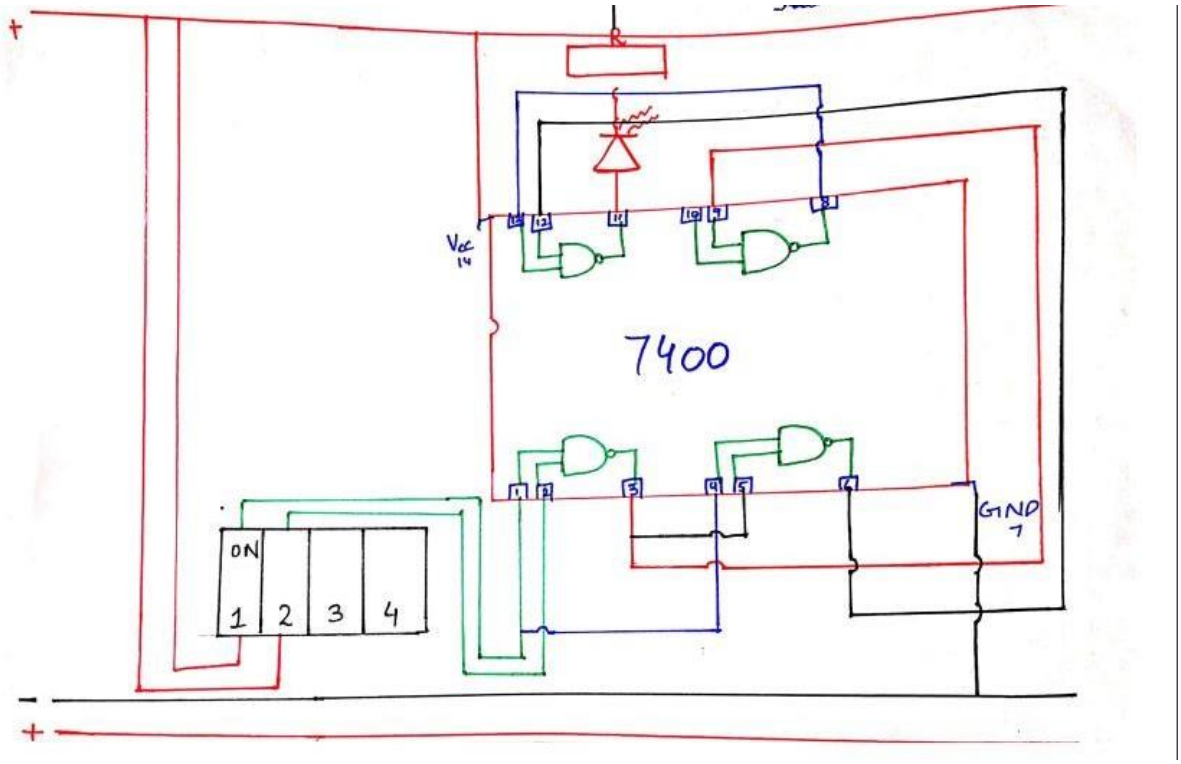


Figure 30 Schematic diagram of NAND to X-NOR



a. Write Boolean function for XNOR gate:

$$\mathbf{Q} = (A.\ (AB)')'.\ (B.\ (AB)')$$

b. Truth Table for XNOR Gate:

Input A	Input B	Output F
0	0	1
0	1	0
1	0	0
1	1	1

EXPERIMENT # 4

IMPLEMENTATION OF COMBINATIONAL FUNCTIONS USING VARIOUSLOGIC GATES

1. Material Required

- IC component: 4073, 4072, 4049
- Connecting Wires
- Logic Trainer

2. Procedure

- Connect the Logic trainer to 220 volts AC power supply.
- Install the ICs on trainer's breadboard.
- Wire the circuit according to the diagram shown as here under.
- Use logic switches to provide inputs at A, B and C as per table provided in step 3.
- Connect the output of the circuit to the LED provided on the breadboard.
- Verify the truth table with respect to the inputs and outputs to satisfy the system design.

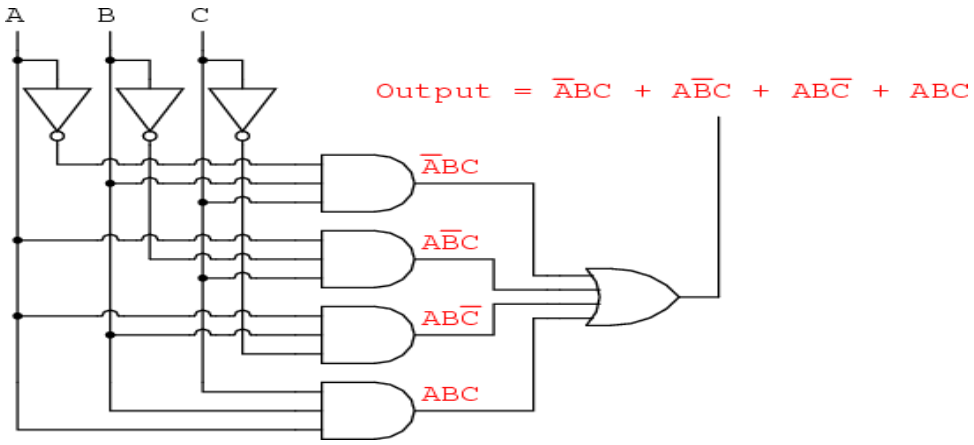
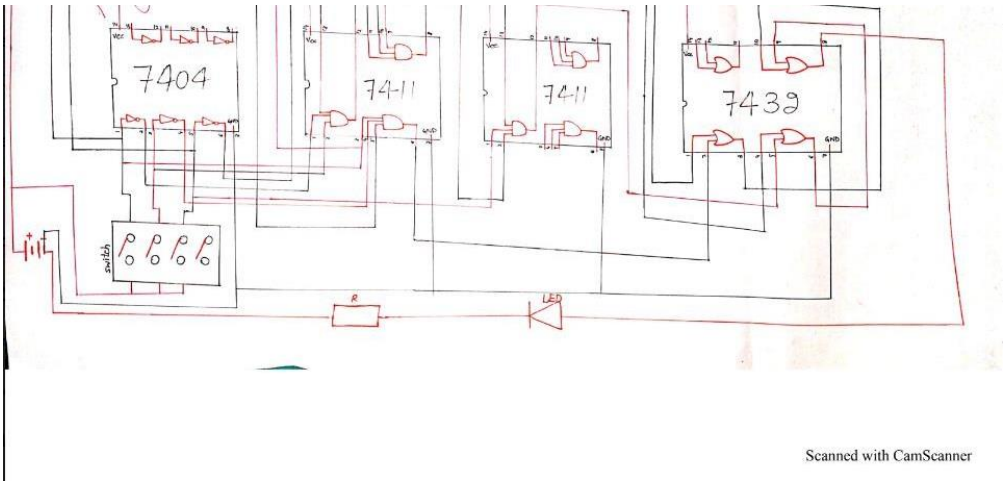


Figure 32 Schematic diagram of combinational functions using logic gates



3. Experimental Work:

In this experiment, you will Implement a combinational logic by applying the various inputs, according to the table given here under and find out the results with respect to the Boolean expression.

Input A	Input B	Input C	Input	Output F
0	0	0	A′B′C′	0
0	0	1	A′B′C	0
0	1	0	A′BC′	0
0	1	1	A′BC	0
1	0	0	AB′C′	0
1	0	1	AB′C	1
1	1	0	ABC′	1
1	1	1	ABC	1

EXPERIMENT # 5

IMPLEMENTATION OF HALF ADDER & FULL ADDER

1.1. Material Used:

- Connecting wires
- Logic Trainer
- Components (XOR Gate, AND Gate, OR Gate)

1.2. Procedure:

- Wire the circuit according to the pins supply indicator i.e. +5v to pin number 7, 14 respectively.
- Interconnect the basic logic gates as per logic diagrams given for half adder and full-adder
- Implement the given Boolean expression by basic logic gates, verify and write the result in the truth table

1.3. Experimental Results

Inputs		Output			
		S_{HA}		C_{HA}	
X	Y	Actual (Formula)	Observed (Experiment)	Actual	Observed
0	0	0	0	0	0
0	1	1	1	0	0
1	0	1	1	0	0
1	1	0	0	1	1

Schematic Diagram:

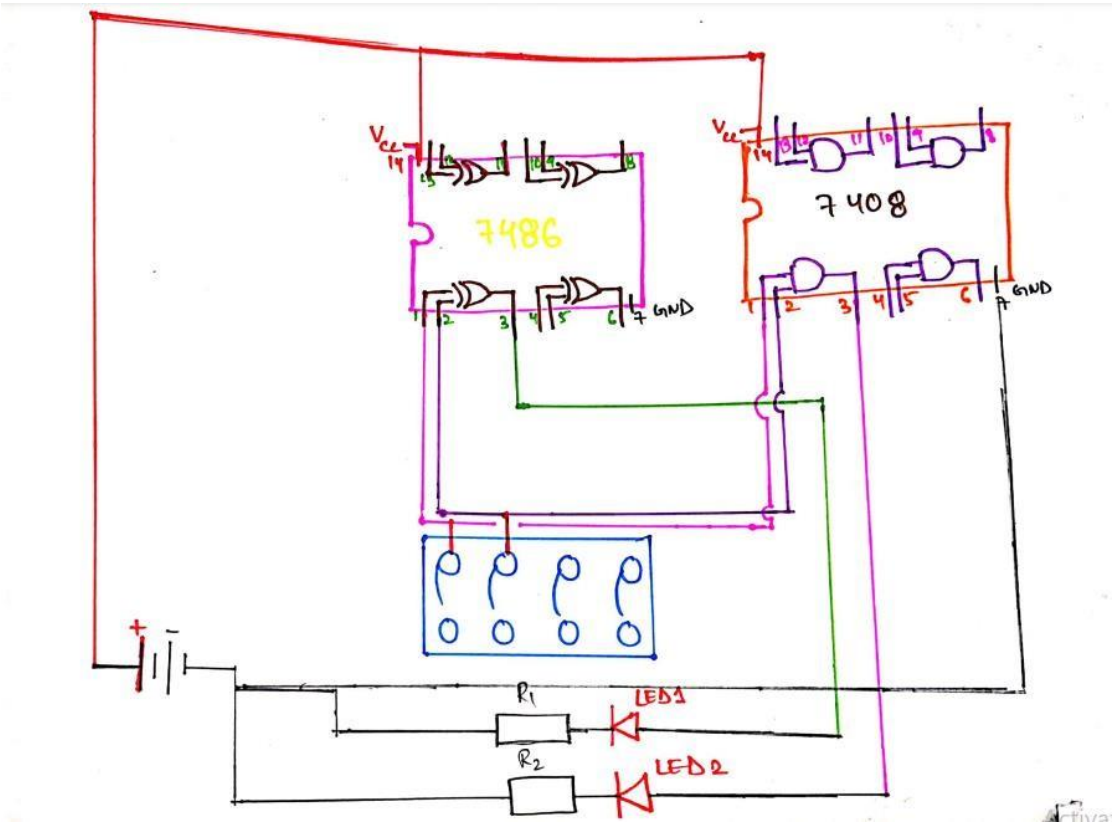


Figure 34 Schematic diagram of Half adder

Tinkercad Circuit:

Truth Table:

Inputs			Output			
			S_{FA}		C_{FA}	
x	y	z	Actual	Observed	Actual	Observed
0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	0	0	1	1
1	0	0	1	1	0	0
1	0	1	0	0	1	1
1	1	0	0	0	1	1
1	1	1	1	1	1	1

Sum = x′y′z+x′yz′+xy′z′+xyz

Simplifying by using Boolean Postulates & theorems/k-map, we get

Sum=(x′y+xy′)′.z+(x′y+xy′

SFA = (x ⊕ y) ⊕ z

Carry = x′yz + xy′z + xyz′+xyz

Simplifying by using Boolean Postulates & theorems/k-map, we get

Carry=(x′y+xy′).y

CFA =

(x⊕y)z+xy

Schematic

Diagram:

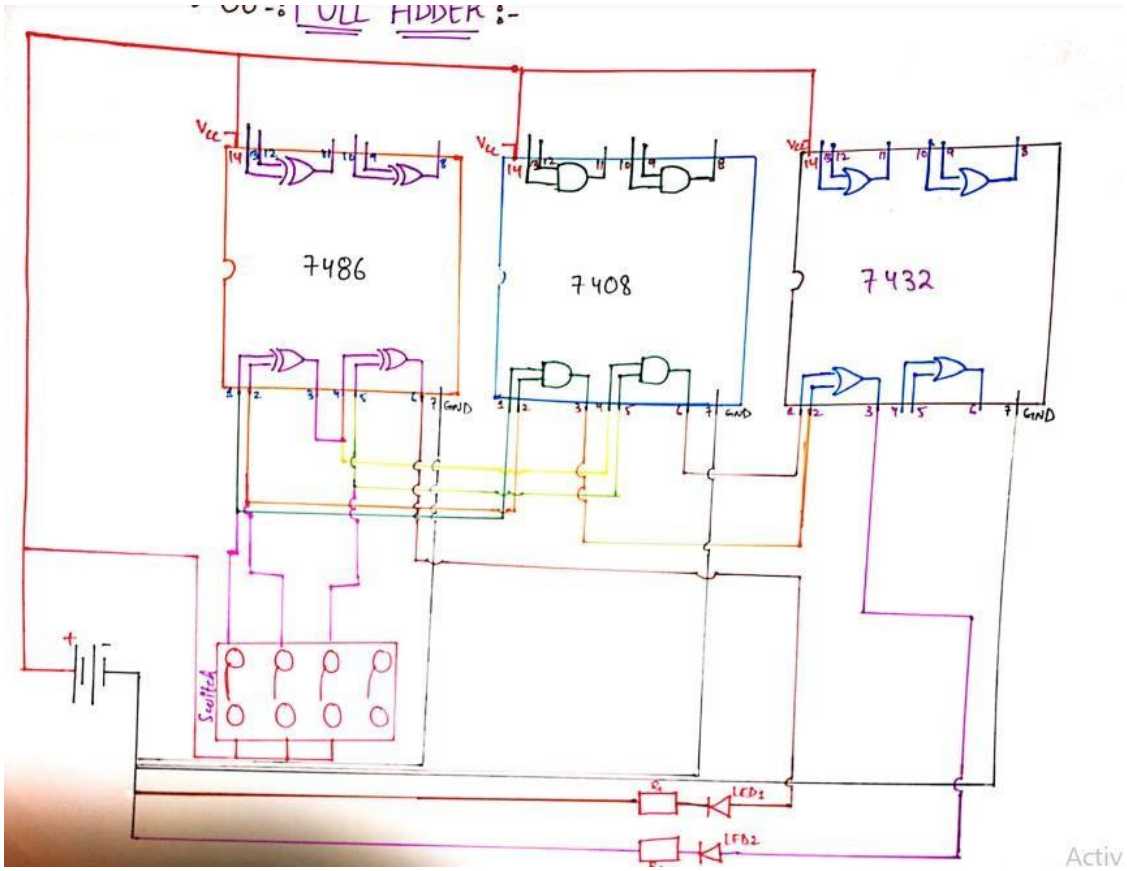


Figure 36 Schematic diagram of Full adder

Tinkerc

ad

Circuit:

Truth Table:

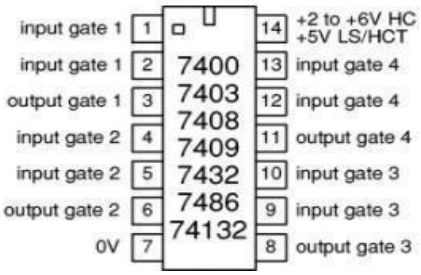
Inputs		Output			
		S_{HA}		C_{HA}	
X	Y	Actual (Formula)	Observed (Experiment)	Actual	Observed
0	0	0	0	0	0
0	1	1	1	0	0
1	0	1	1	0	0
1	1	0	0	1	1

DATASHEET:

Quad 2-input gates

- 7400 quad 2-input NAND
- 7403 quad 2-input NAND with open collector outputs
- 7408 quad 2-input AND
- 7409 quad 2-input AND with open collector outputs
- 7432 quad 2-input OR
- 7486 quad 2-input EX-OR
- 74132 quad 2-input NAND with Schmitt trigger inputs

The 74132 has Schmitt trigger inputs to provide good noise immunity. They are ideal for slowly changing or noisy signals.



EXPERIMENT # 6

DESIGN THE BCD-TO-SEVEN-SEGMENT DECODER CIRCUIT

1. Experimental Work

1.1. Material Used:

- Logic trainer
- Connecting wires
- Seven-Segment Display
- Components: 7447

- Power supply

1.2. Procedure

- Wire the IC chip and connect the +5v (Vcc) and ground the pin number 16 and 8 respectively.
- The BCD (8421) code is listed in the table given from this table. You can determine the relation between each BCD bit and the decimal digits in order to analyze the logic.
- For instance, the most significant bit of BCD code, A3 is always a "1" for decimal digits 8 and 9. An OR expression for bit A3 in terms of a decimal digits can therefore be written is
- A3= 8+9
- Bit A2 is always a one; a decimal digit is always "1" for 4, 5, 6 or 7 and can be expressed as an OR function as follows.
- A2= 4+5+6+7
- Bit A1 is always a one for decimal digit 2,3, 6 or 7 and can be expressed as
- A1 =2+3+6+7
- Finally a zero is always a 1 for decimal digit 1,3,5,7, or 9 the expression for zero is
- A0=1+3+5+7+9

Truth Table:

Inputs										
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

Schematic Diagram:

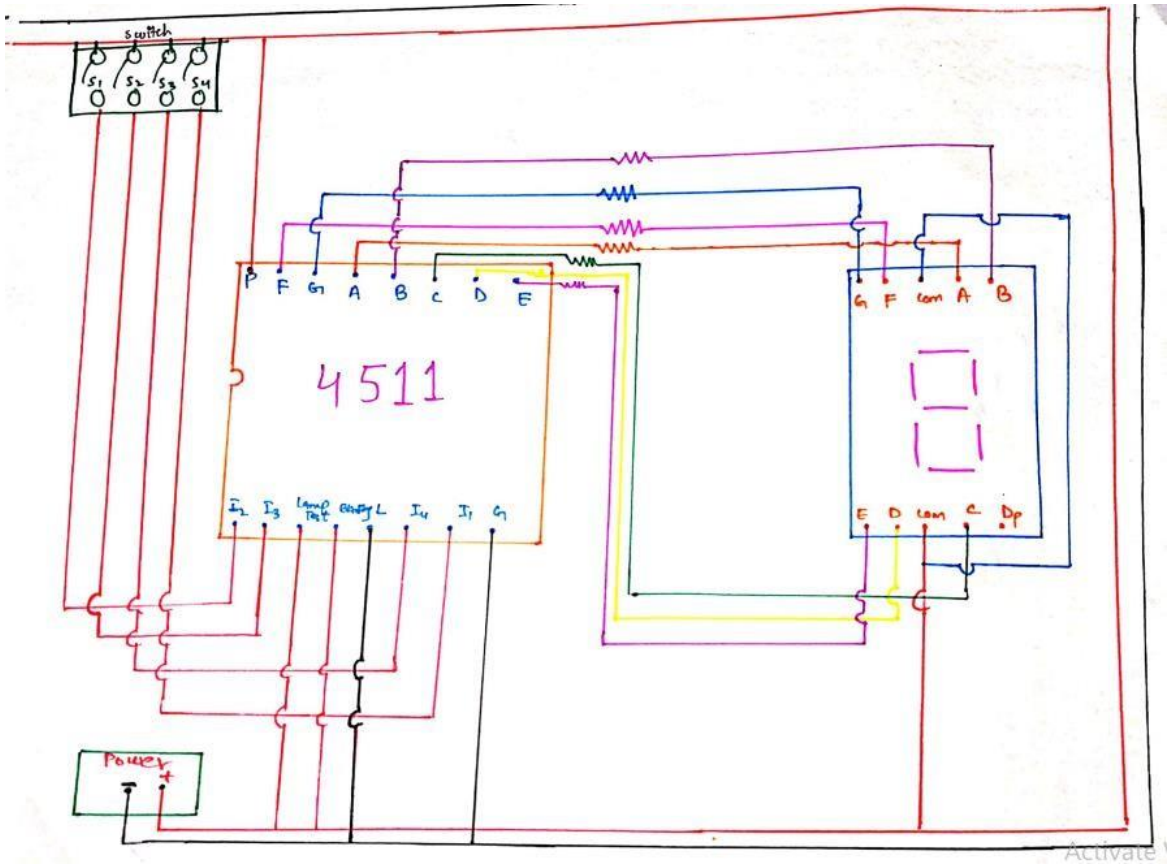


Figure 38 Schematic diagram of BCD-to-seven-segment decoder

EXPERIMENT # 7

INTRODUCTION, IMPLEMENTATION AND WORKING WITH MULTIPLEXERS,DECODERS AND ENCODERS

1. Experimental Work

1.1 Material Used

- Components: 4073, 4049, 4071
- Connecting Wires

- Logic Trainer

1.2. Procedure

We will implement the multiplexer circuit first. It is clear from the logic diagram that the AND, OR and NOT gate implementation of multiplexer requires four 3-input AND gates, one 4-input OR gate and two NOT gates. Get the required number of ICs containing above mentioned gates and other apparatus from the lab attendant. Install the ICs in the breadboard of the Logic Trainer. All three IC models used are 16 pin ICs. These are designed in such a way that pin number 8 is considered as ground and power is given to pin number 16. For other pin configurations consult the data sheet (we have already used these gates in the first lab so it should not be a problem). Wire your circuit according to the logic diagram for the multiplexer circuit as given above. Once you have wired the circuit, check it with your instructor and, if approved, power up your circuit. The outputs should be connected to the LEDs on the Logic Trainer for monitoring purposes.

Repeat the same procedure for the decoder and encoder circuit.

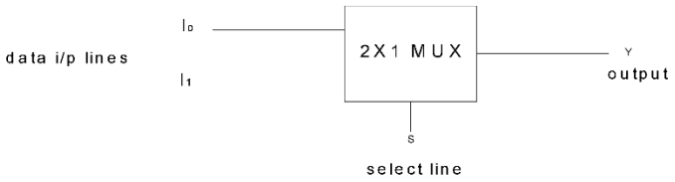
a. Multiplexer

Multiplexer, simply called Mux, is a data selector and is capable of "selecting" one of many input lines (usually 2^n) and displaying its input status on the only output line available.

A Mux has

1. Select lines
2. Data input lines
3. Output line.

Block diagram of 2x1 MUX

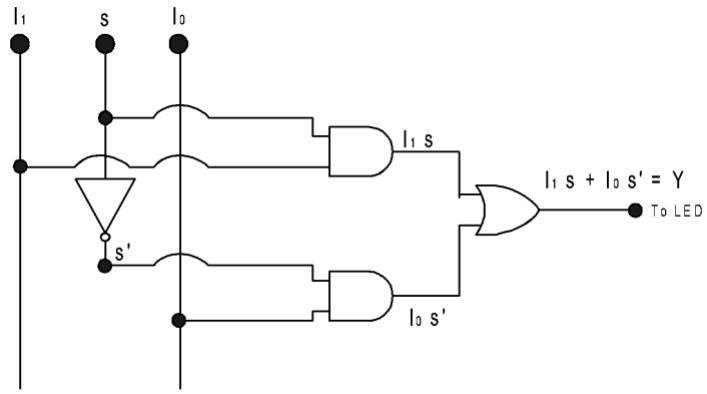


The function table of 2x1 Mux is

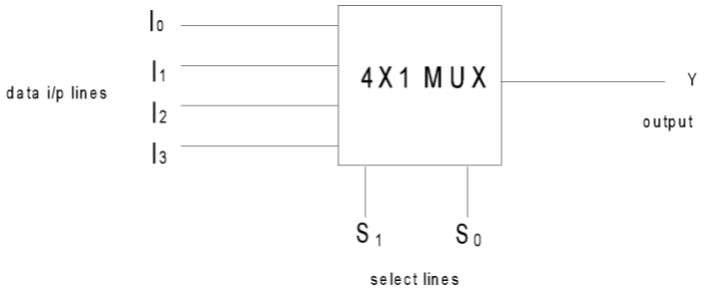
Select line	o/p
S	Y
0	I_o
1	I_1

The Boolean function for 2x1 Mux is: $\mathbf{Y = I_1 s + I_0 s'}$

Logic Diagram of 2x1 Mux is



Block diagram of 4x1 MUX



I_0, I_1, I_2 and I_3 are inputs of Mux

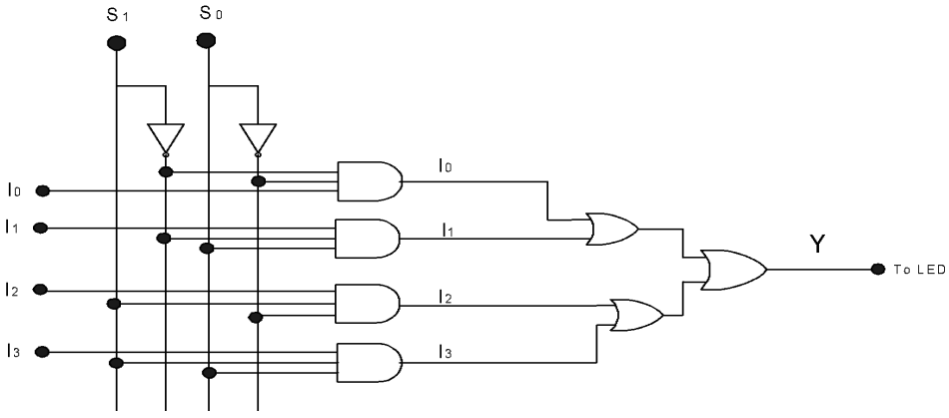
S_1 and S_0 are select lines

Y is output

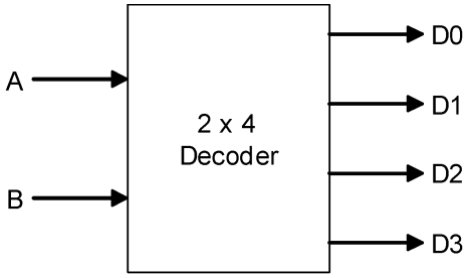
The Boolean function for 4x1 Mux is

$$Y = I_0 S_1' S_0' + S_1' S_0 I_1 + S_1 S_0' I_2 + S_1 S_0 I_3$$

Logic Diagram of 4x1 Mux is



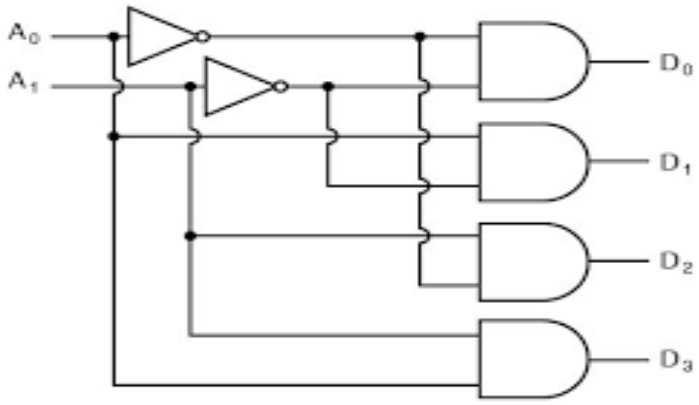
b. Block diagram of 2x 4 Decoder



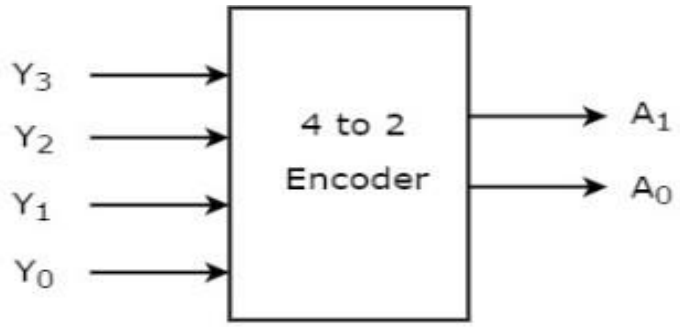
The truth table of **2-to-4 line decoder** is given below. The output variables of a decoder are mutually exclusive because only one output can be equal to 1 at any time.

Inputs		**Outputs**			
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

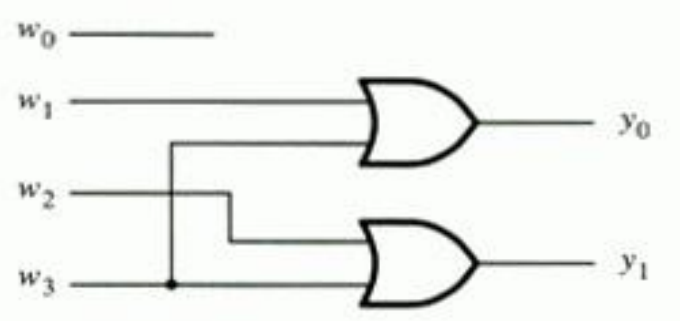
Draw the logic diagram of 2x4 decoder:



Draw the Block Diagram of 4-to-2 Encoder:



Draw the logic diagram of 4x2 Encoder



2. EXPERIMENTAL RESULT

Fill in the following truth tables while observing the outputs.

Truth table for 4x1 multiplexer:

Select inputs		**Output**
S_1	S_0	Y
0	0	1
0	1	1

1	0	1
1	1	1

Schematic Diagram:

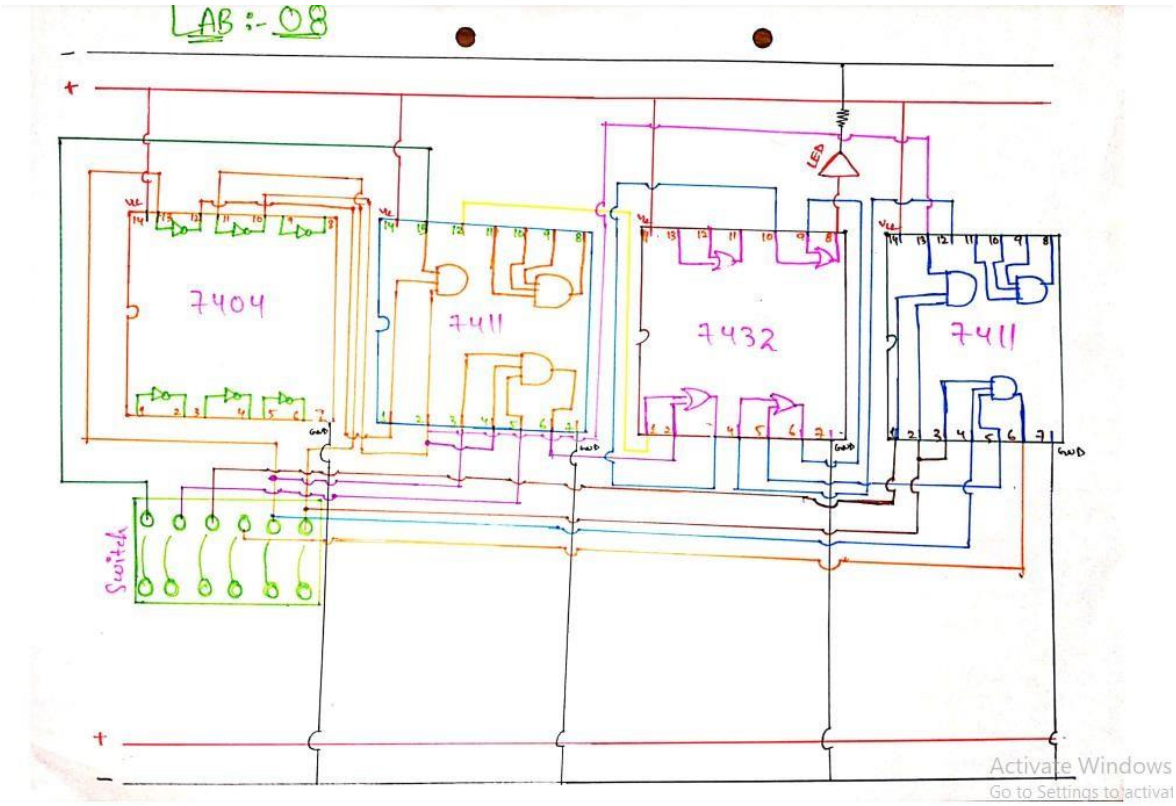


Figure 42 Schematic diagram of 4x1 multiplexer

Truth table for 2x4 decoder:

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

Schematic Diagram:

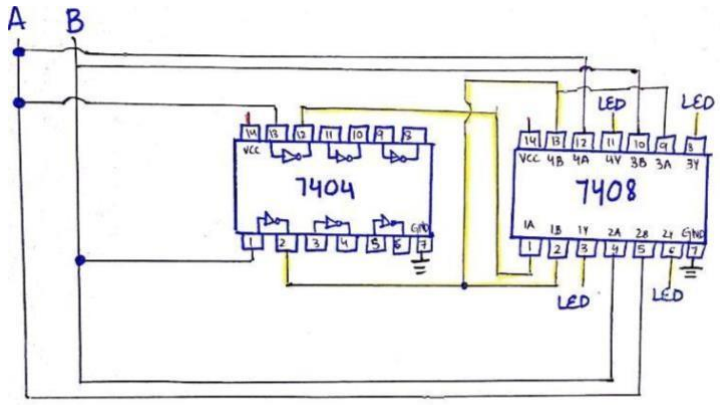


Figure 43 Schematic diagram of 2x4 decoder

Truth table for 4x2 encoder:

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

Schematic Diagram:

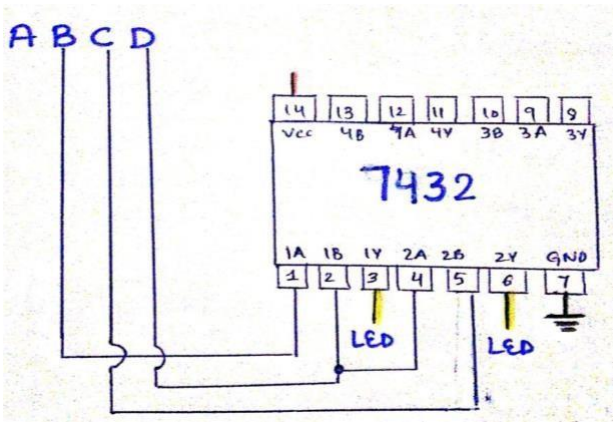


Figure 45 Schematic diagram of 4x2 encoder

EXPERIMENT # 8(a)

IMPLEMENTATION OF FULL ADDER WITH TWO, 2X4 DECODERS

1. Experimental Work

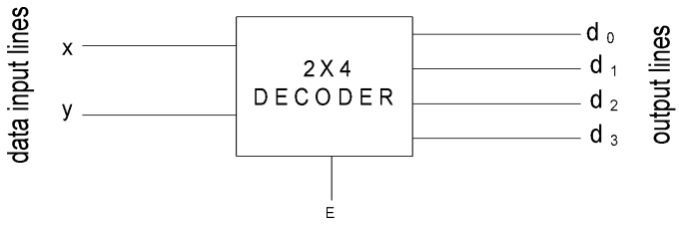
1.1.Material Used:

- Components: 4071,
- Connecting Wires
- Logic Trainer

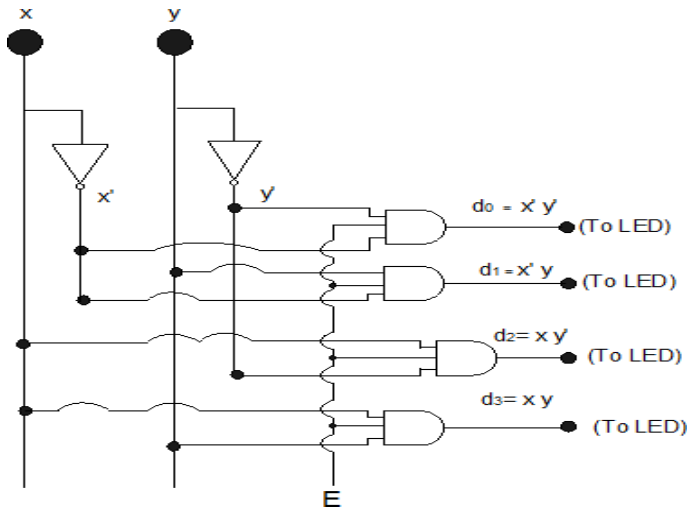
1.2.Procedure:

Implement Half Adder with 2x4 Decoder.

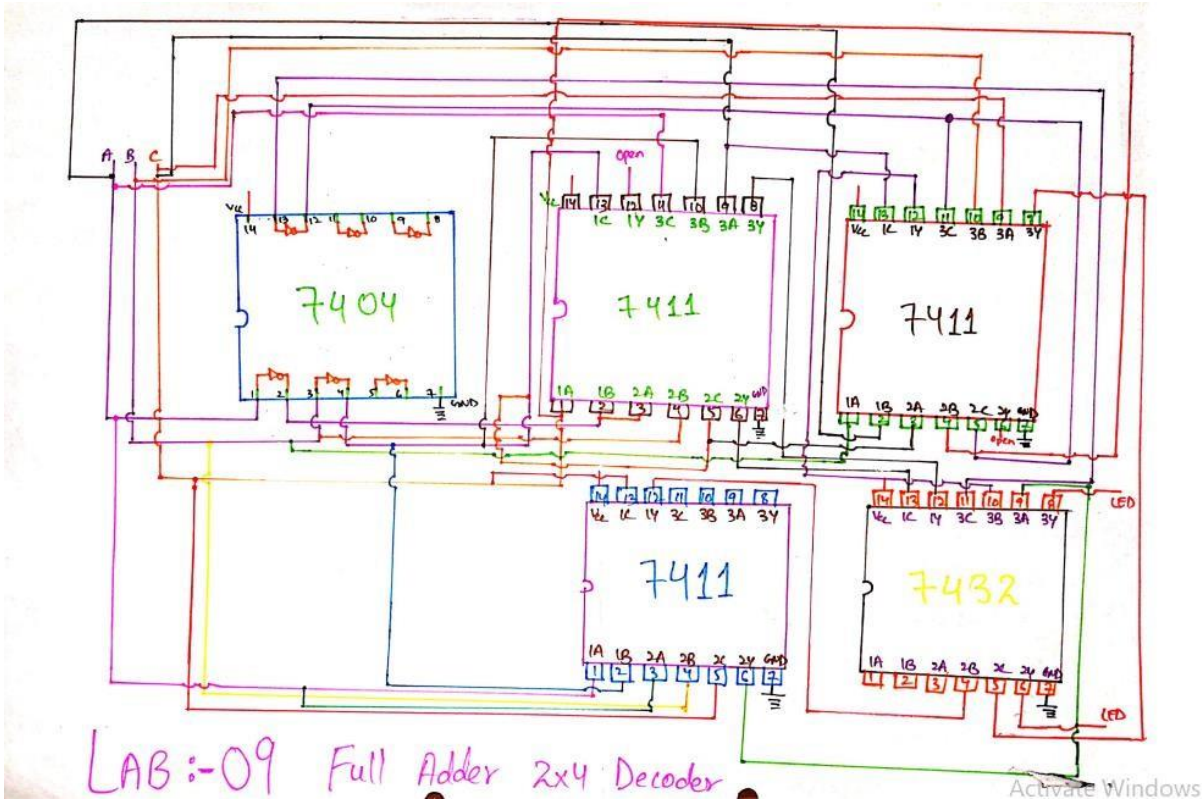
Block Diagram of 2X4 Decoder.



Logic Diagram of 2X4 Decoder:



Design Schematic diagram (A)When Buttons are ON



Design Schematic diagram (B)When Buttons are Off:

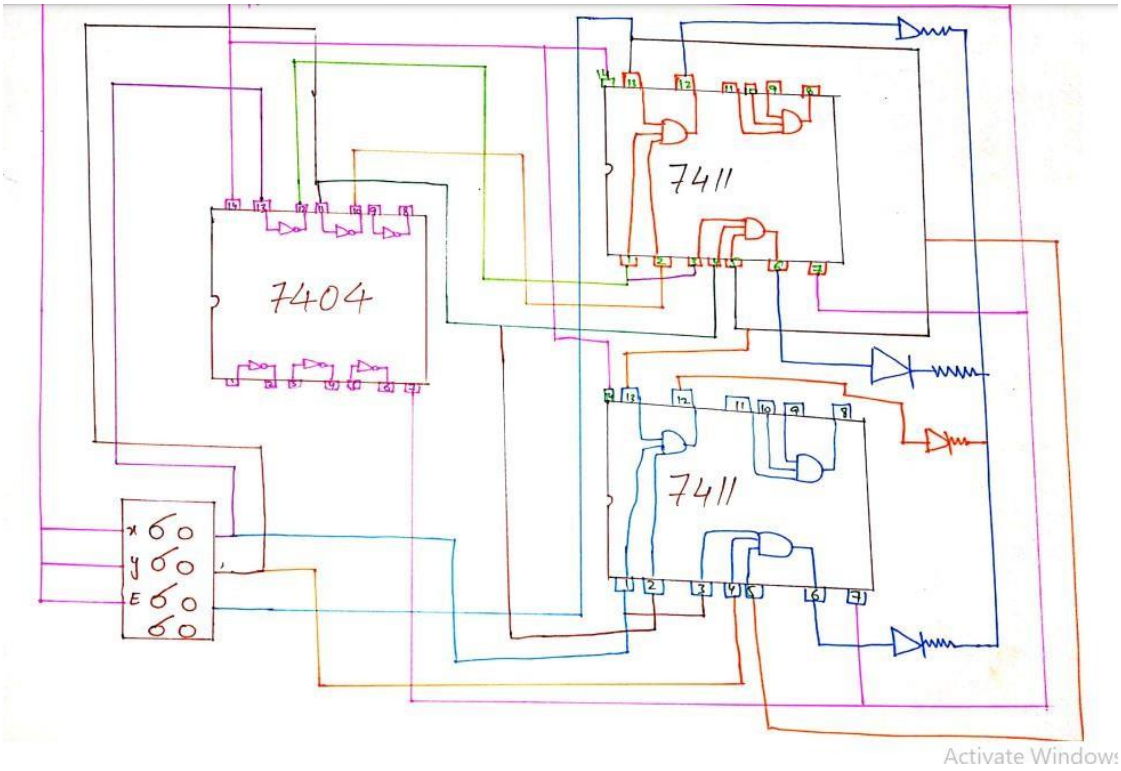


Figure 49 Schematic diagram of 2X4 Decoder when buttons are OFF

Truth Table of 2X4 Decoder

x	Y	E	d_0	d_1	d_2	d_3
0	0	1	0	0	0	1
0	1	1	0	0	1	0

1	0	1	0	1	0	0
1	1	1	1	0	0	0

Boolean Functions for 2 x 4 Decoder

$d_o = E\, x'\, y'$

$d_1 = E\, x'y$

$d_2 = E\, x\, y'$

$d_3 = E\, x\, y$

Truth Table of Half Adder

i/p's		o/p's	
x	**y**	**S_{HA}**	**C_{HA}**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Truth Table of 2X4 Decoder

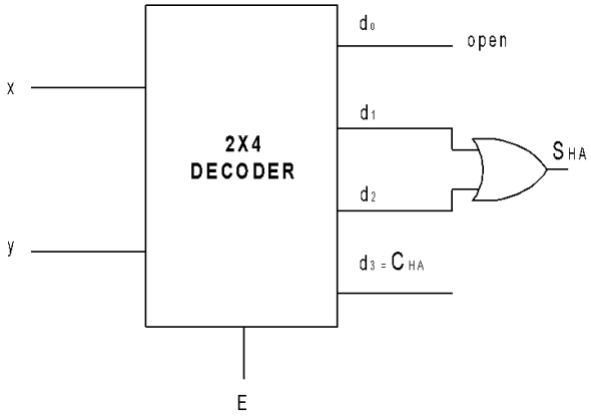
i/p's		o/p's			
x	**Y**	**d0**	**d1**	**d2**	**d3**
0	0	0	0	0	0
0	1	0	0	0	0
1	0	0	0	0	0
1	1	0	0	0	0

By comparing Truth Tables of half Adder and 2 X 4 Decoder.

We can see that $S_{HA} = d_1 + d_2$

$C_{HA} = d_3$

Block Diagram of Half Adder with Truth Table of 2X4 Decoder



Schematic Diagram:

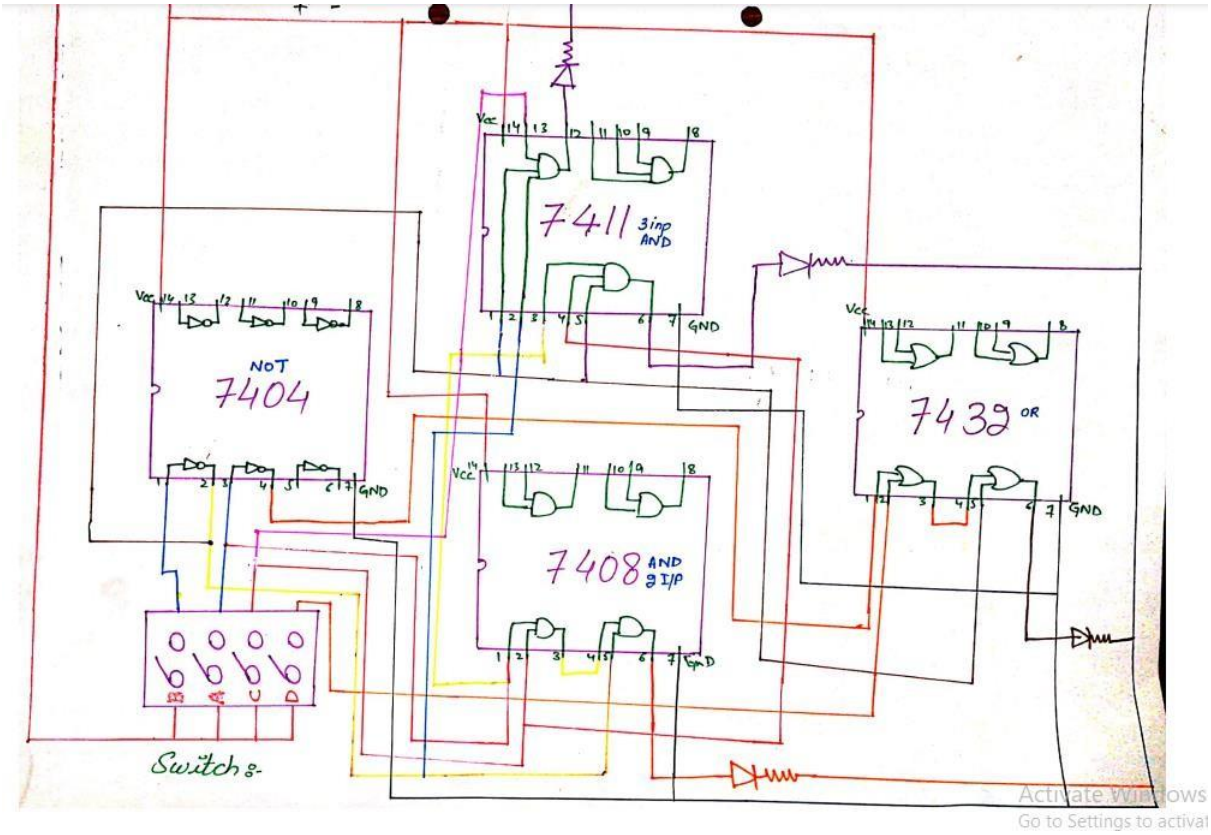


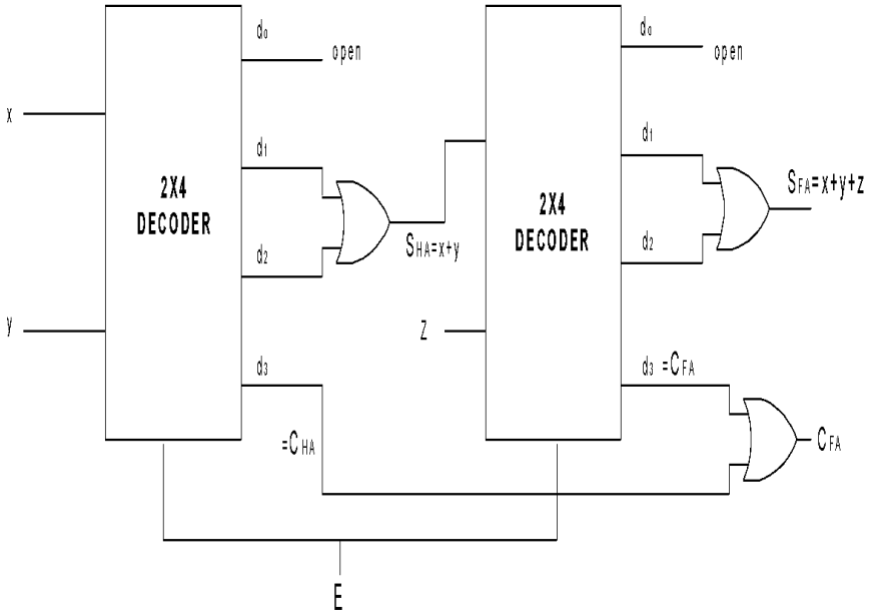
figure 51 Schematic diagram of Half Adder with 2X4 Decoder

Truth Table of Full Adder

i/p's			o/p's	
x	**Y**	**z**	**S_{HA}**	**C_{HA}**
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

Block Diagram of Full Adder with 2, 2X4 Decoders.

Using the concept of implementation of Half Adder with 2X4 Decoder, we can implement Full Adder with 2, 2 X 4 Decoders.



Schematic Diagram:

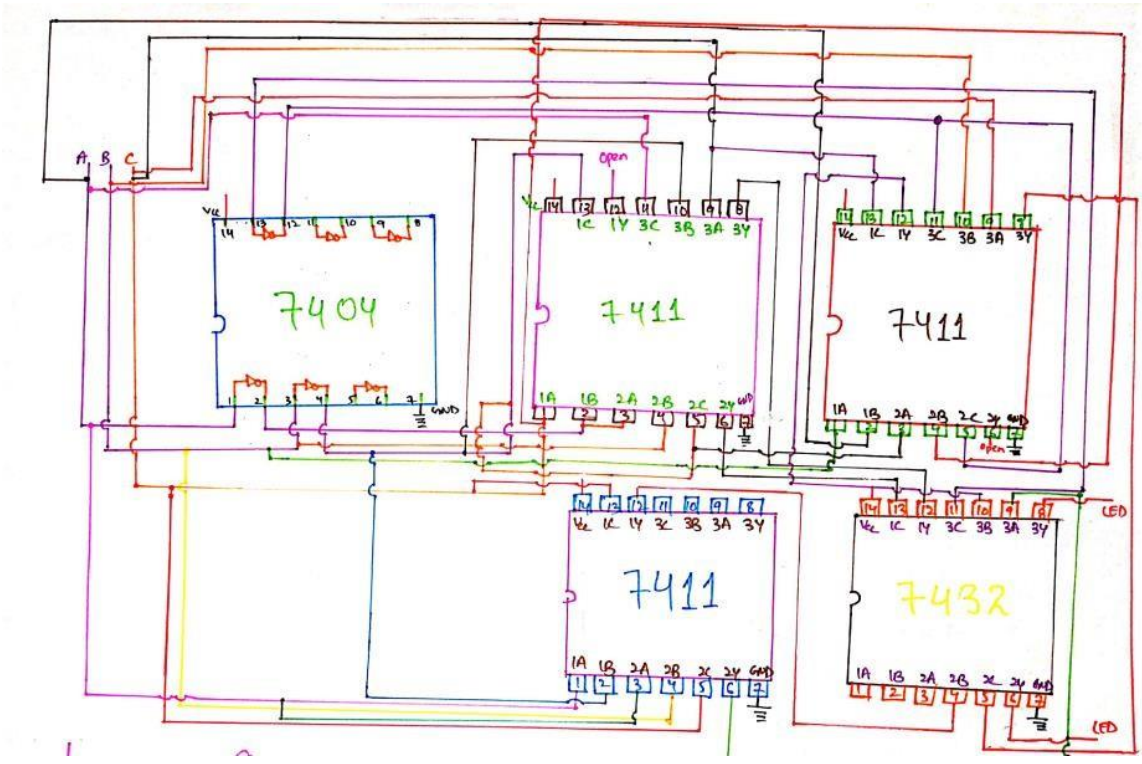


Figure 53 Schematic diagram of Full Adder with 2, 2X4 Decoders

EXPERIMENT # 8(b)

IMPLEMENTATION OF FULL ADDER WITH 8x1 MUX

1. Experimental Work:

1.1. Material Used:

- Components
- Connecting Wires
- Logic Trainer

1.2. Procedure:

Connect the selection switches of both MUX to input bits.

First of all we check/implement Carry of Full Adder (having 3 inputs) using 8X1 Mux, for this take:

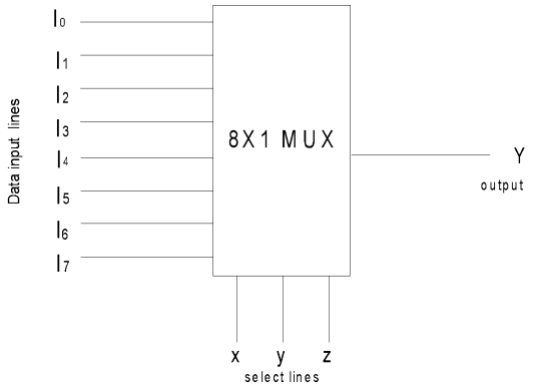
$I_0 = I_1 = I_2 = I_4 = 0$, $I_1 = 0$, $I_2 = 0$, $I_3 = I_5 = I_6 = I_7 = 1$, from Carry column of Truth table of Full Adder and then select x, y, z from Function table of 8X1 Mux and then observe outputs at Y Pin, that should be equal to Carry of Full Adder for combination of x, y, z at select lines, which is inserted through data switches, this step is repeated for all x, y, z combinations, at select lines to observe Carry of Full Adder.

Then we check/implement Sum of Full Adder for 3 input variables, using 8X1 Mux for this, we take:

$I_0 = 0$, $I_1 = 1$, $I_2 = 1$, $I_3 = 0$, $I_4 = 1$, $I_5 = 0$, $I_6 = 0$, $I_7 = 1$, from Sum column of Truth Table of Full Adder, as data inputs to 8X1 Mux, and then for each combination of x, y, z at select lines from Function table, we see output at Y Pin of the IC, which should be equal to value of Sum of Full Adder for x, y, z combination at select lines, which is inserted through data switches, this step is repeated for all x, y, z combinations, at select lines to observe Sum of Full Adder.

2. Experimental Results:

BLOCK DIAGRAM:



Function Table:

Select lines			o/p
x	**Y**	**z**	**Y**
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

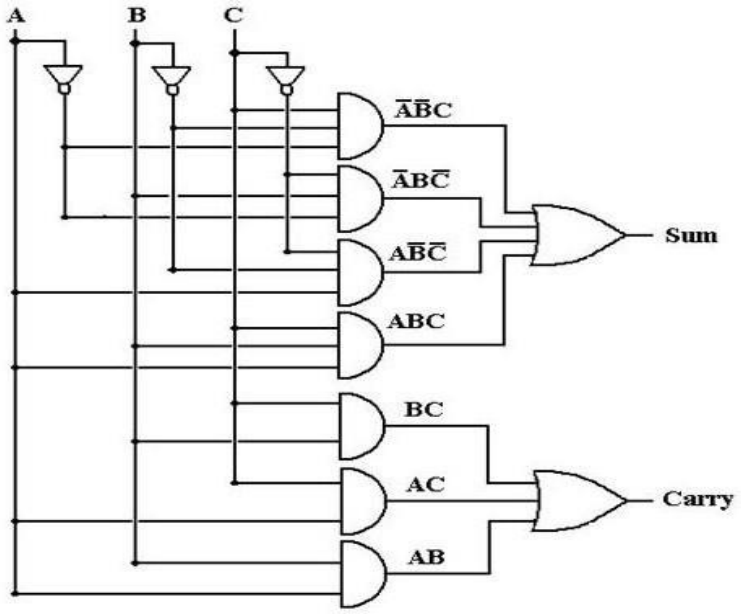
Truth Table of Full Adder

Inputs of Full Adder			Outputs	
X	**Y**	**z**	**S**	**C**
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

Function Table of 8x1 Mux

i/p of Full Adder = Select lines of MUX			o/p of 8x1 mux	o/p of 8x1 mux	o/p of 8x1 mux
x	y	z	S = Y	C = Y	
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	1	1	0
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	0	0	1
1	1	0	0	0	1
1	1	1	1	1	1

Draw the complete circuit diagram of above arrangement



Design Schematic diagram (A)When Buttons are ON

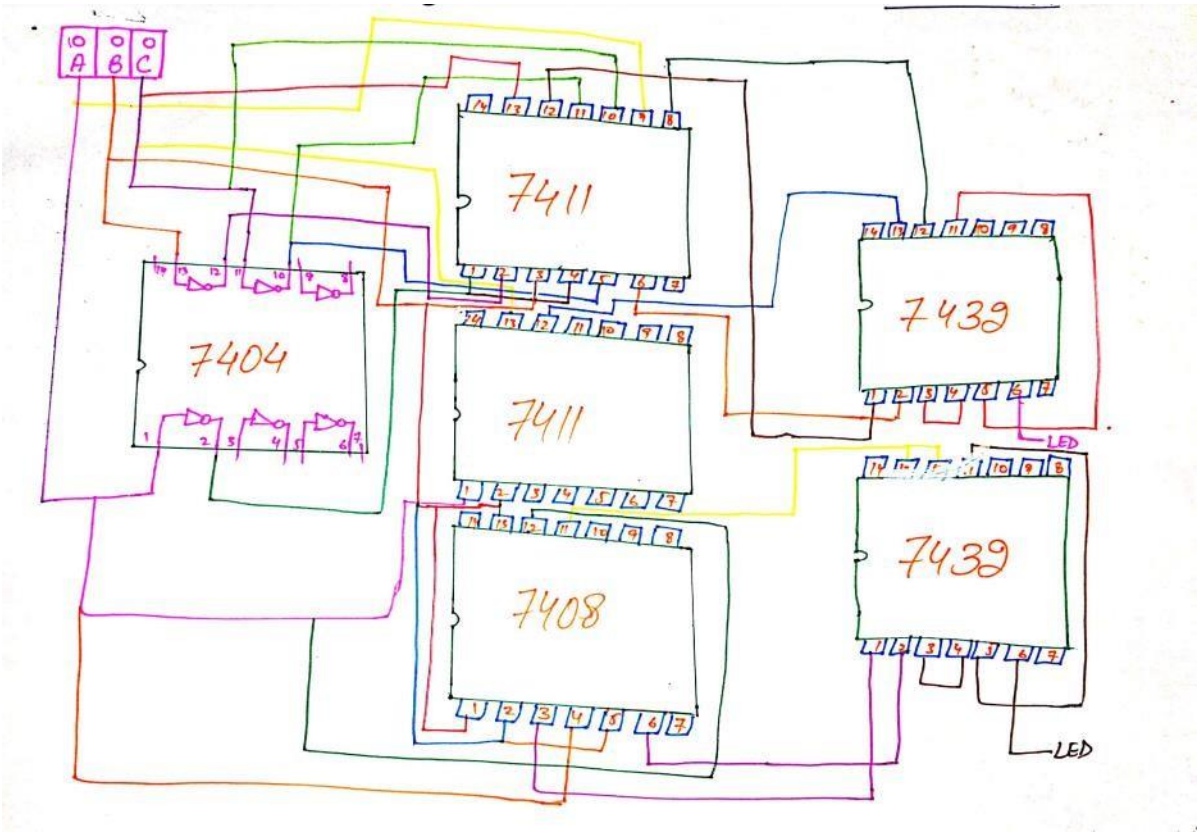


Figure 63 Schematic diagram of 8x1 Mux (i)

Design Schematic diagram (B)
When Buttons are OFF

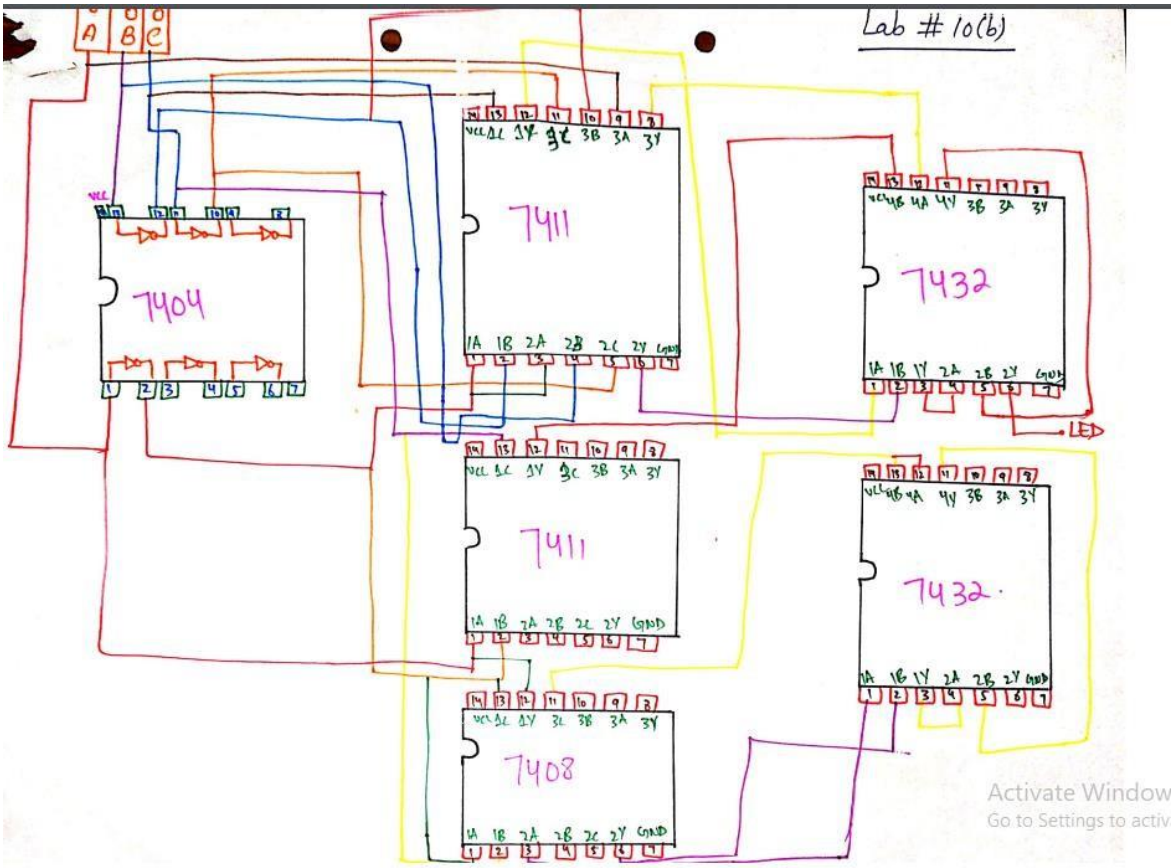


Figure 64 Schematic diagram of 8x1 Mux (ii)

EXPERIMENT # 9

VERIFICATION OF THE TRUTH TABLE OF RS FLIP FLOP

1. Experimental Work:

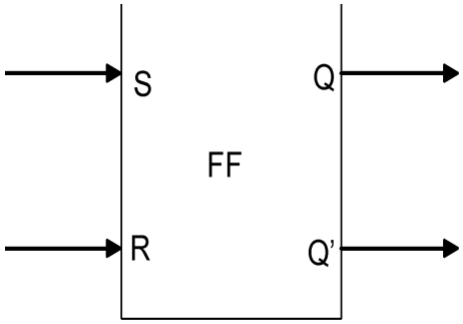
1.1 Material Used:

- Logic trainer
- Connecting wires
- IC: 4011, 4001
- Power supply

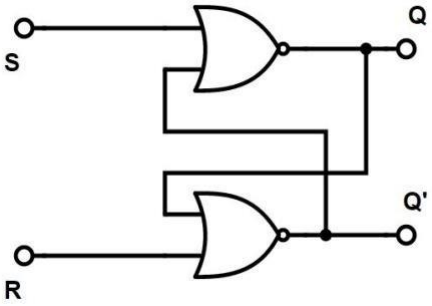
1.2 Procedure:

The NOR gate implementation of RS flip-flop requires NOR gates AND gates. Get the required ICs and other apparatus from the lab attendant. Install the IC 7400 in the breadboard of the Logic Trainer. Connect 5Vdc power supply and ground on pins 14 and 7 respectively. For other pin configuration consult the data sheet (we have already used these gates in the first lab so it should not be a problem). Wire your circuit according to the logic diagram you have drawn. Once you have wired the circuit, check it with your instructor and, if approved, power up your circuit. The outputs should be connected to the LEDs on the Logic Trainer for monitoring purpose. Apply different input combinations at the input and note down the Q and Q' outputs and fill in the following truth table. This truth table should conform to the one given in theory. If there are problems, consult the appendix on troubleshooting given at the end of lab manual. If the problem persists, request the lab supervisor for help.

Block Diagram of RS Flip Flop:



Circuit Diagram of RS Flip Flop sing NOR gates:



2.1. Draw Schematic diagram of RS flip flop using NOR gates:

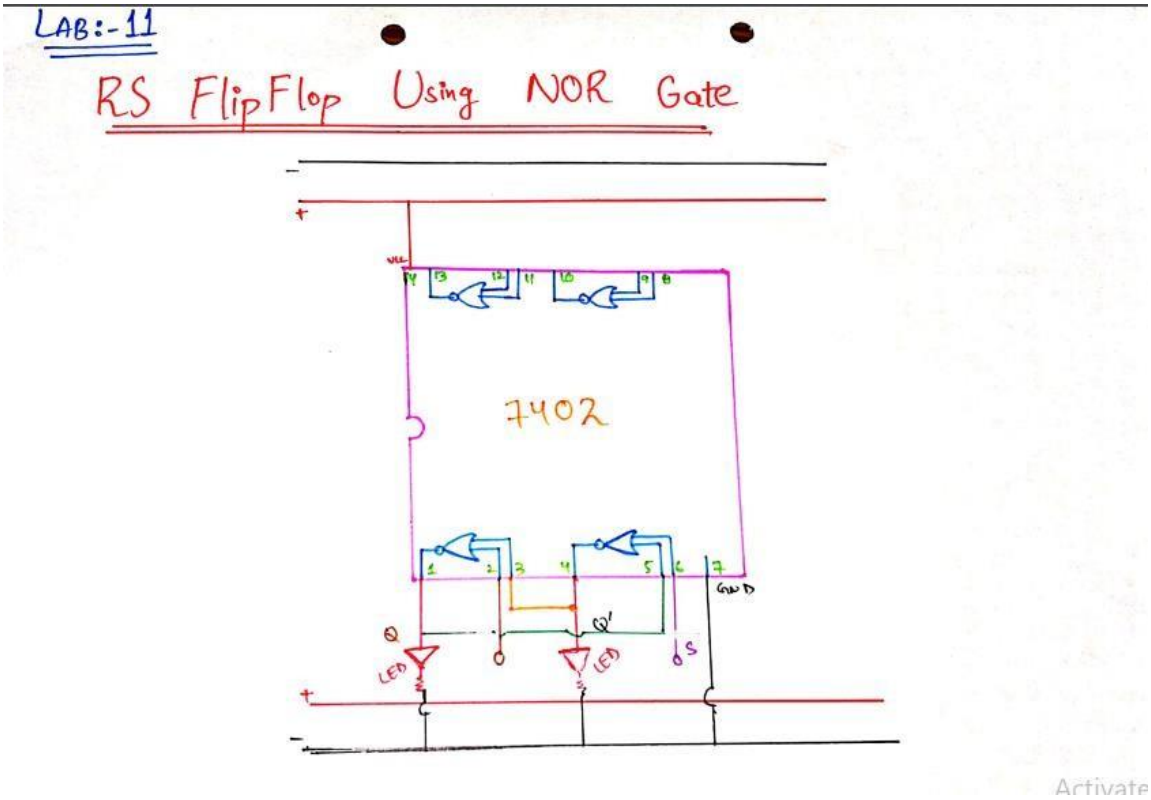


Figure 65 Schematic diagram of RS flip flop using NOR gates

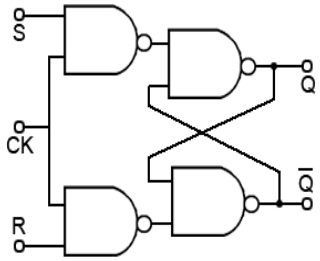
2.2.Experimental Results:

Fill in the following truth table by observing the outputs.

Truth Table for RS flip-flop with NOR Gates:

Inputs		Outputs		Comments
S	**R**	**Q**	**Q'**	
0	0	0	0	No Change
0	1	1	0	Reset
1	0	0	1	Set
1	1	-	-	Undefined

Circuit Diagram of RS Flip Flop using NAND gates:



Draw the Schematic circuit of RS Flip flop using NAND gates

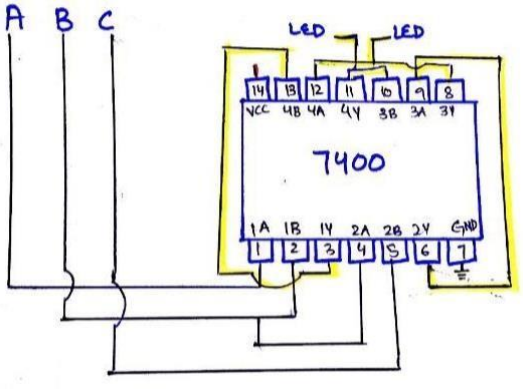


Figure 67 Schematic diagram of RS Flip flop using NAND gates

Truth Table for RS flip-flop with NAND Gates:

Inputs		Outputs		Comments
S	R	Q	Q'	
0	0	0	0	No change
0	1	0	1	Reset state
1	0	1	0	Set state
1	1	1	1	Indetermined

EXPERIMENT # 10

VERIFICATION OF THE TRUTH TABLE OF JK FLIP FLOP

1. **Experimental Work:**

1.1 Material Used.

- Logic trainer
- Connecting wires
- IC: 4011, 4001
- Power supply

1.2 Procedure:

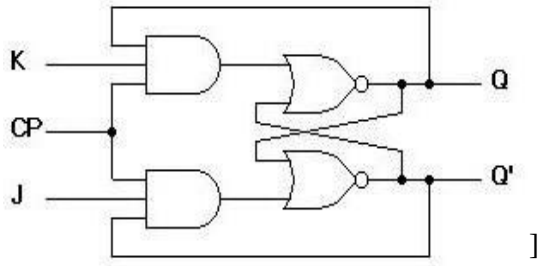
The implementation of JK flip-flop requires two 2-input NOR gates and two 3-input AND gates. Get the required ICs and other apparatus from the lab attendant. Install the ICs in the breadboard of the Logic Trainer. Connect 5Vdc power supply and ground on pins 14 and 7 respectively. For other pin configuration consult the data sheet (we have already used these gates in the first lab so it should not be a problem). Wire your circuit according to the logic diagram you have drawn. Once you have wired the circuit, check it with your instructor and, if approved, power up your circuit. The outputs should be connected to the LEDs on the Logic Trainer for monitoring purpose. Apply different input combinations at the input and note down the Q (t+1) outputs and fill in the following truth table. This truth table should conform to the one given in theory. If there are problems, consult the appendix on

troubleshooting given at the end of lab manual. If the problem persists, request lab supervisor for help.

Repeat the same procedure for implementation of JK flip-flop with NAND gates.

2. Experimental Results:

Draw the logic diagram of a JK flip flop using NOR and AND gates:



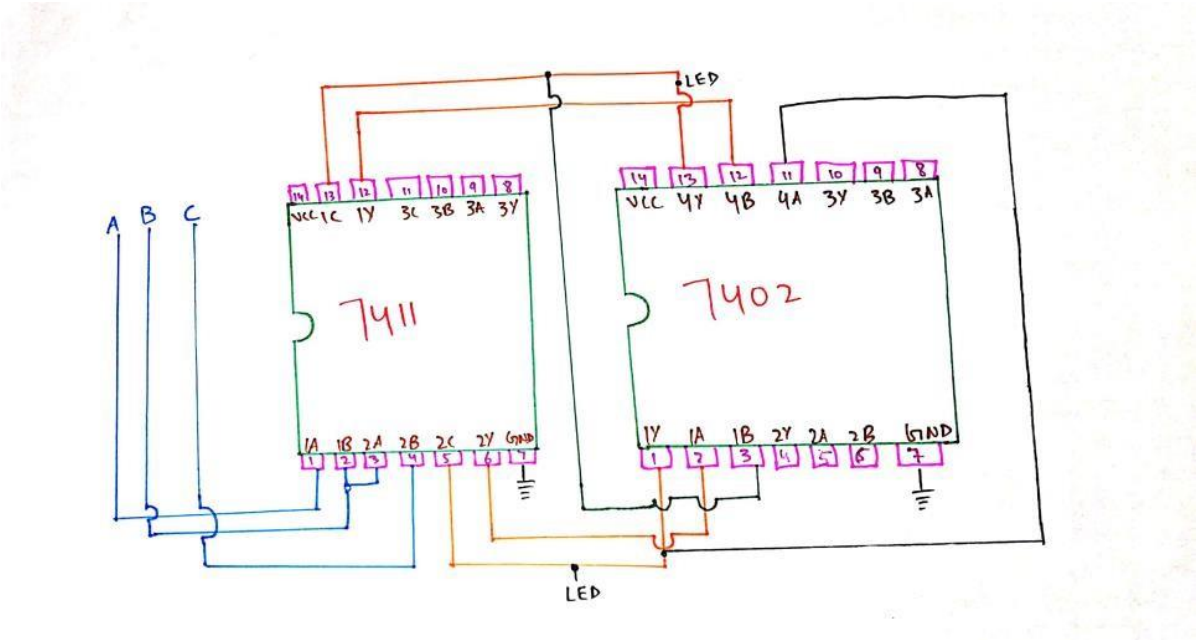


Figure 69 Schematic diagram of JK flip flop using NOR and AND gates

Fill in the following truth tables by observing the outputs.

JK with NOR gates:

Q	J	K	Q(t+1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	Invalid
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	Invalid

Observed characteristics table for NOR JK flip - flop

Draw the logic diagram of a JK flip flop using NAND gates:

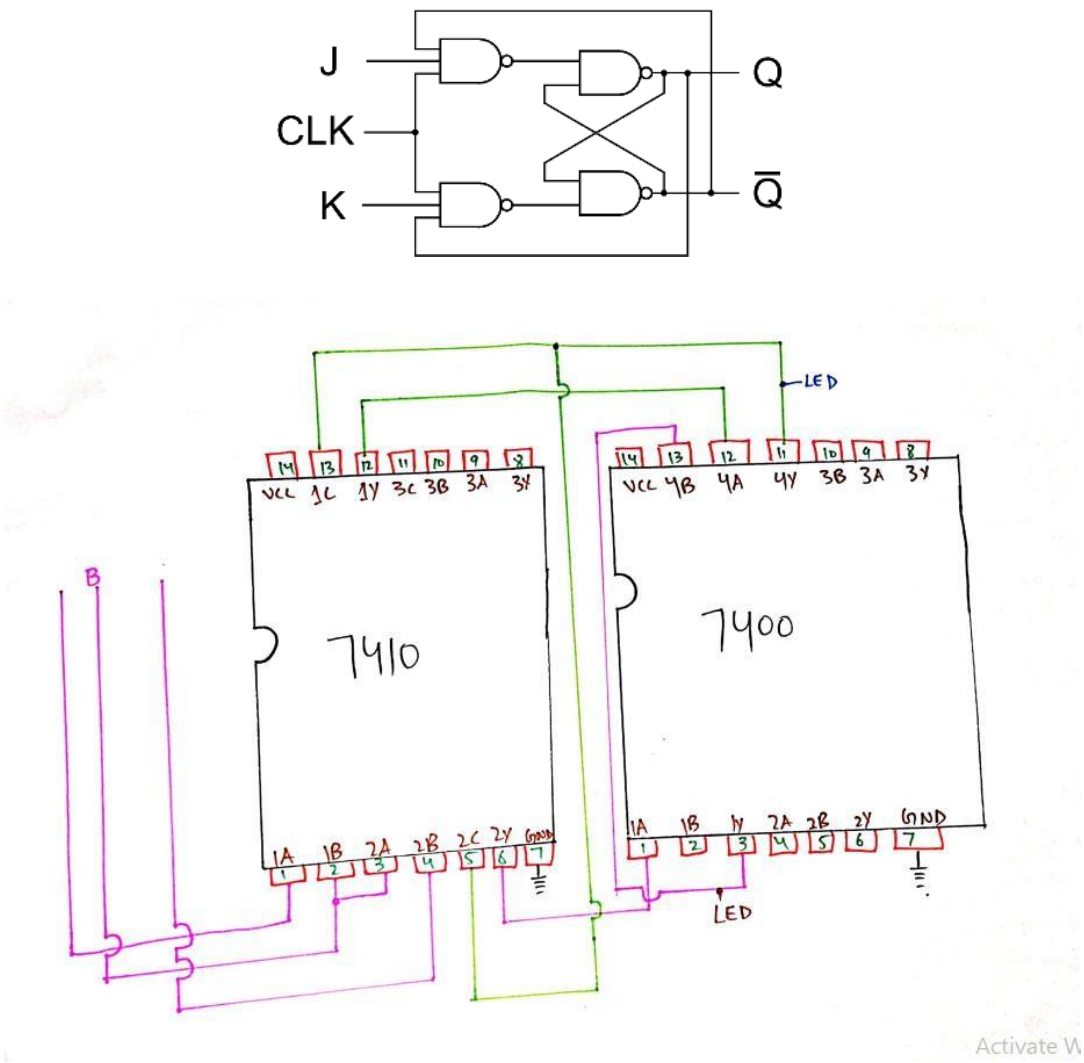


Figure 71 Schematic diagram of JK flip flop using NAND gates

JK with NAND gates:

Q	J	K	Q(t+1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	Invalid
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	Invalid

Observed characteristics table for NAND gates JK flip - flop

EXPERIMENT # 11

Introduction to Emu8086 and Assembly Language

Objectives

In this lab, an introduction of Emu8086 will be given. Also, the Hello World Program will be explained to make students understand how programs are written in Emu8086 using Assembly Language.

Theory

Introduction

Emu8086 is a program that compiles the source code (assembly language) and executes it. You can watch registers, flags and memory while your program executes. Arithmetic & Logical Unit (ALU) shows the internal work of the central processor unit (CPU). Emulator runs programs on a Virtual PC; this completely blocks your program from accessing real hardware, such as hard-drives and memory, 8086 machine code is fully compatible with all next generations of Intel's microprocessors.

Where to start?

- Start Emu8086 by selecting its icon from the start menu, or by running Emu8086.exe.
- Select "Samples" from "File" menu.
- Click [Compile and Emulate] button (or press F5 hot key).
- Click [Single Step] button (or press F8 hot key), and watch how the code is being executed.
- Try opening other samples, all samples are heavily commented, so it's a great learning tool.

Directives

ORG 100h is a compiler directive (it tells compiler how to handle the source code). This directive is very important when you work with variables. It says to compiler that the executable file will be loaded at the offset of 100h (256 bytes), so compiler should calculate the correct address for all variables when it replaces the variable names with their offsets. Directives are never converted to any real machine code.

Why executable file is loaded at offset of 100h? Operating system keeps some data about the program in the first 256 bytes of the CS (code segment), such as command line parameters and etc.

Offset is use to get the offset address of the variable in register specified

MOV instruction

- Copies the second operand (source) to the first operand (destination).
- The source operand can be an immediate value, general-purpose register or memory location.
- The destination register can be a general-purpose register, or memory location.
- Both operands must be the same size, which can be a byte or a word.

Syntax:

```
mov destination, source
```

Example:

```
mov ax, 10      ; puts the value of 10 in the register ax
mov cx, ax      ; puts the value contained in the register ax into cx
```

Register is a series of memory cells inside the CPU itself. Because registers are inside the CPU there is very little overhead in working with them. There are four general purpose registers, AX, BX, CX, and DX. These are the registers you will be using often. Each of these general registers is 16- bit. They also have 8-bit counterparts. AX is 16 bits where as AH and AL is 8bit.

Note: - AH being the high bit, and AL being the low bit. Together AH and AL make AX.

Procedure is a part of code that can be called from your program in order to make some specific task.

Procedures make program more structural and easier to understand. Generally procedure returns to the same point from where it was called.

Syntax:

```
namePROC
; here goes the code
; of the procedure ...
RET
nameENDP
```

Exercise1.1

Comment on the Architecture of intel 8086 Processor

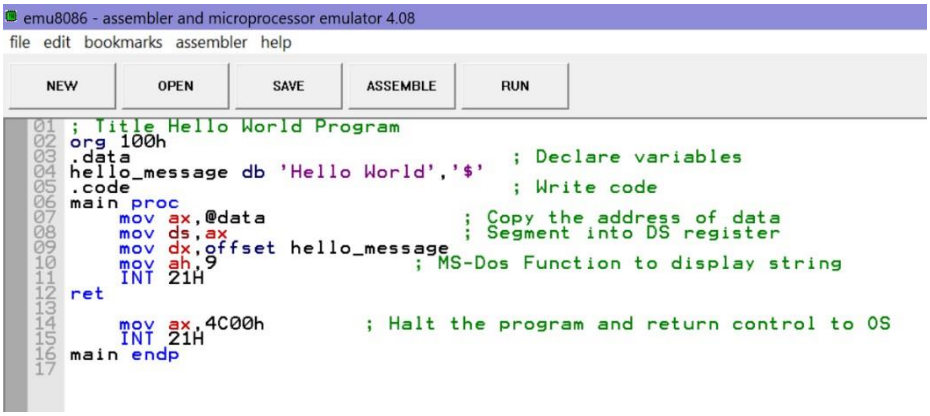
Answer:

8086 Microprocessor is an enhanced version of 8085Microprocessor that was designed by Intel in 1976. It is a 16-bit Microprocessor having 20 address lines and16 data lines that provides up to 1MB storage. It consists of powerful instruction set, which provides operations like multiplication and division easily.

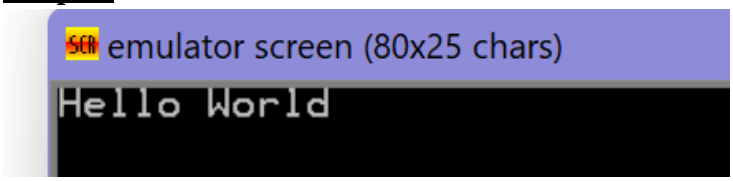
exercise 1.2 : Hello world Program

```
      ; Title Hello World Program
      org 100h
      .data                                              ;
   Declare variables
      hello_message db 'Hello World','$'
   .code                                                 ;
Write code
      main proc
            mov ax,@data                          ; Copy the
   address of data
            mov ds,ax                             ; Segment
```

Assembler:



Output:



EXPERIMENT # 12

Use Registers to Input /Output, Display and Strings in Assembly

OBJECTIVES:

To understand how to give Input, Output and Display Characters and strings

DIRECTIVES:

Assembler Directives are not assembly language instructions as they do not generate any machine code They are special codes placed in the assembly language program to instruct the assembler to perform a particular task or function. They can be used to define symbol values, reserve and initialize storage space for variables and control the placement of the program code

- **Tittle Directive**
- **.MODEL directive**
- **.STACK directive**
- **.DATA directive**
- **.CODE directive**

Title Directive:

Maximum 60 characters wide title.

For example

TITLE print a Hello Word

MODEL directive

Specify how many code and data segments are necessary for the program.

There are many several types of memory small, compact, medium, large etc.

For example

.model small

Shows that's no more than 64 k memory for code and 64k for data.

STACK DIRECTIVE:

This directive is optional and is used to define the size of the stack segment.

Syntax: .*STACK* <*size*>

.stack 100d (100 bytes stack segment)

.stack 100h (256 bytes stack segment)

.DATA directive

Used to define data segment necessary for our program.

Variables are stored in that segments.

Data can be of different types like byte, word, double word or quad word.

DB – Define Byte

DW – Define Word

DD – Define Double Word

DQ – Define Quad Word

DT – Define Ten Bytes

message db "Hello Word","$"

count db 20

<variable> <directive> < Value>

.CODE directive:

Indicate the beginning of instructions i.e. the assembly code.

The assembly language program is end with END directive.

INTERRUPTS

An interrupt is an event that causes the processor to suspend its present task and transfer control to a newprogram

called the interrupt service routine (ISR)
There are three sources of interrupts

- Processor interrupts
- Hardware interrupts generated by a special chip, for ex: 8259 Interrupt Controller.
- Software interrupts

Software Interrupt is just similar to the way the hardware interrupt actually works. The INT Instructionrequests services from the OS, usually for I/O. These services are located in the OS.

INT has a range 0 -> FFh. Before INT is executed AH usually contains a function number that identifies the subroutine.
The 8086 INT instruction can be used to cause the 8086 to do any one of 256 possible interrupt types.

The desired type is specified as part of the instruction.

Software interrupts produced by the INT instructions have many uses.
For example: INT 3 instruction to insert breakpoints in programs for debugging. Another use of Software interrupt INT is to test various interrupts-service procedures.
For example: INT 0 instruction to send execution to a divide-by-zero interrupt – service procedurewithout having to run the actual division program

REGISTERS

Registers are high-speed storage locations directly inside the CPU, designed to be accessed at much higherspeed than conventional memory.

Types of registers

General purpose Registers

8086 CPU has 8 general purpose registers, each register has its own name:
AX - the accumulator register (divided into **AH / AL**).**BX** - the base address register (divided into **BH / BL**).**CX** - the count register (divided into **CH / CL**).
DX - the data register (divided into **DH / DL**).**SI** - source index register.
DI - destination index register.
BP - base pointer.
SP - stack pointer.
Despite the name of a register, it's the programmer who determines the usage for each general purpose register. The main purpose of a register is to keep a number (variable). The size of the above registers is 16 bit, it'ssomething like: **0011000000111001b** (in binary form), or **12345** in decimal (human) form.
4 general purpose registers (AX, BX, CX, DX) are made of two separate 8 bit registers, for example if AX= **0011000000111001b**, then AH=**00110000b** and AL=**00111001b**. Therefore, when you modify any of the 8 bit registers 16 bit register is also updated, and vice-versa. The same is for other 3 registers, "H" is for high and "L" is for low part.

Because registers are located inside the CPU, they are much faster than memory. Accessing a memory locationrequires the use of a system bus, so it takes much longer. Accessing data in a register usually takes no time.

Therefore, you should try to keep variables in the registers. Register sets are very small and most registers have special purposes which limit their use as variables, but they are still an excellent place to store temporary data ofcalculations.

SEGMENT REGISTERS

CS - points at the segment containing the current program.
DS - generally points at segment where variables are defined. **ES** - extra segment register, it's up to a coder to define its usage.**SS** - points at the segment containing the stack.

Although it is possible to store any data in the segment registers, this is never a good idea. The segment registers have a very special purpose pointing at accessible blocks of memory. Segment registers work togetherwith general purpose register to access any memory value. For example if we would like to access memory at the physical address **12345h** (hexadecimal), we should set the **DS = 1230h** and **SI = 0045h**. This is good, sincethis way we can access much more memory than with a single register that is limited to 16 bit values. CPU makes a calculation of physical address by multiplying the segment register by 10h and adding general purposeregister to it (1230h * 10h +45h = 12345h): The address formed with 2 registers is called an **effective address**.By default **BX, SI** and **DI** registers work with **DS** segment register;**BP** and **SP** work with **SS** segment register. Other general purpose registers cannot form an effective address! Also, although **BX** can form an effective address, **BH** and **BL** cannot!

FLAG REGISTERS

STATUS FLAGS

- Allow the results of one instruction to influence later instructions.
- The arithmetic instructions use OF, SF, ZF, AF, PF, and CF

Zero Flag (ZF)

If the result is zero, zero flag is set. Once a flag is set, it remains in that state until another instruction that affects the flags is executed. Not all instructions affect all status flags **add** and **sub** affect all six flags **inc** and**dec** affect all but the carry flag **mov**, **push**, and **pop** do not affect any flags.

Example

Initially, assume **ZF = 0**

```
MOV AL,55H      ;  ZF is still zero
SUB AL,55H      ;  Result is 0
                ; ZF is set (ZF = 1)
```

Sign Flag (SF)

Indicates the sign of the result

Useful only when dealing with signed numbers

- MOV AL,15
- ADD AL,97

Clears the sign flag as sets the sign flag asthe result is 112 (or 0111000 in binary)

- MOV AL,15

- **SUB AL,97**

sets the sign flag as the result is **-82** (or 10101110 in binary)Carry

Flag (CF)

Records the fact that the result of an arithmetic operation on unsigned numbers is out of rangeThe carry flag is set in the following example

MOV AL,0FH
ADD AL,0F1H

MOV INSTRUCTION

Copies the second operand (source) to the first operand (destination).
The source operand can be an immediate value, general-purpose register or memory location.
The destination register can be a general-purpose register, or memory location.
Both operands must be the same size, which can be a byte or a word.These types of operands are supported:
MOV REG, memory
MOV memory, REG
MOV REG, REG
MOV memory, immediate
MOV REG, immediate

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
memory: [BX], [BX+SI+7], variable, etc...
immediate: 5, -24, 3Fh, 10001101b, etc...

For segment registers only these types of MOV are supported:MOV SREG, memory
MOV memory, SREG
MOV REG, SREG
MOV SREG, REG
SREG: DS, ES, SS, and only as second operand: CS.
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
memory: [BX], [BX+SI+7], variable, etc...
The MOV instruction cannot be used to set the value of the CS and IP registers.
Example:

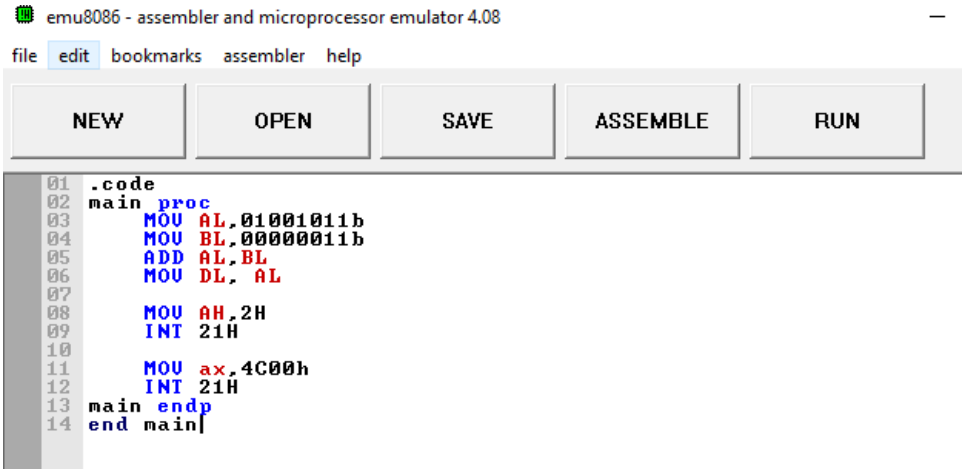
ORG 100h ; directive required for a COM program.
MOV AX, 0B800h ; set AX to hexadecimal value of B800h.MOV DS, AX ; copy value of AX to DS.
MOV CL, 'A' ; set CL to ASCII code of 'A', it is 41h.
MOV CH, 01011111b ; set CH to binary value.

MOV BX, 15Eh ; set BX to 15Eh.
MOV [BX], CX ; copy contents of CX to memory at B800:015ERET ;
returns to operating system.

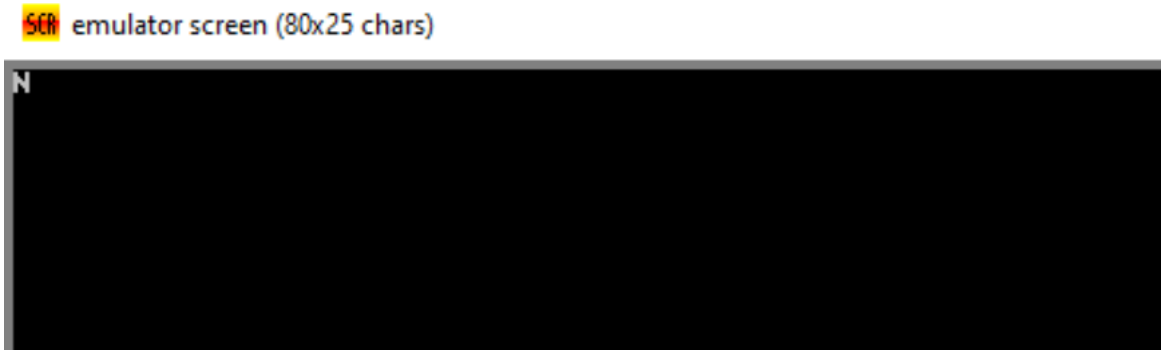
Addition of binary digits
Example # 01

```
.code
main proc
    MOV AL,01001011b
    MOV BL,00000011b
    ADD AL,BL
    MOV DL, AL

    MOV AH,2H
    INT 21H
```

Assembler:



Output:



CX (Count Register)

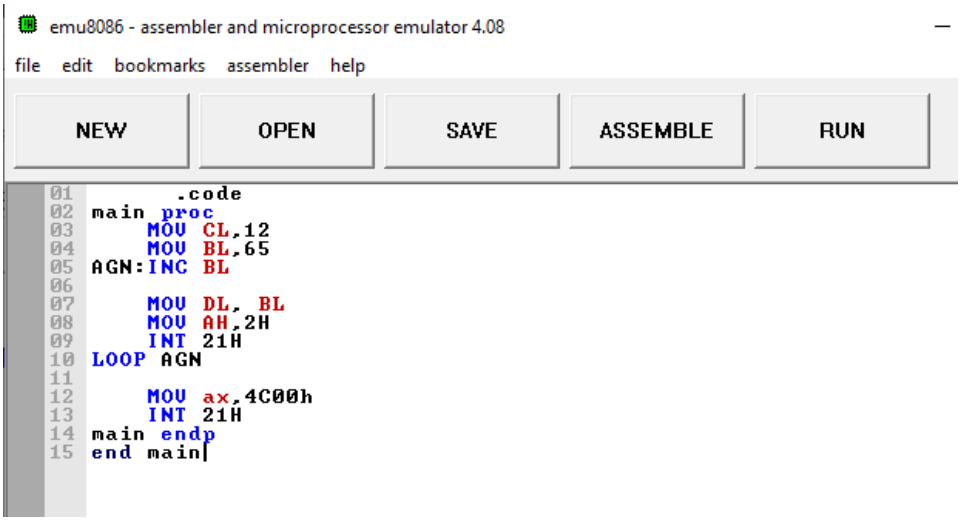
- Contains the count for certain instructions e,g shift count, rotate the number of bytes and a counter with loop instruction.
- Can be accessed as 32 bit (ECX), 16 bit (CX) or 8 bit (CH or CL) register 32 bit
- General Purpose use in ADD, MUL, DIV, MOV

- Special Purpose use in LOOP etc.

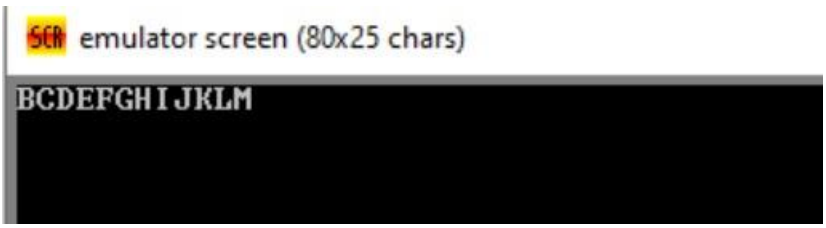
Example # 02

```
            .code
            main proc
                MOV CL,12
                MOV BL,65
            AGN:INC BL

                MOV DL, BL
                MOV AH,2H
                INT 21H
                  LOOP AGN

                MOV ax,4C00h
                INT 21H
            main endp
          end main
```

Assembler:



Output:



Introduction to Input / Output

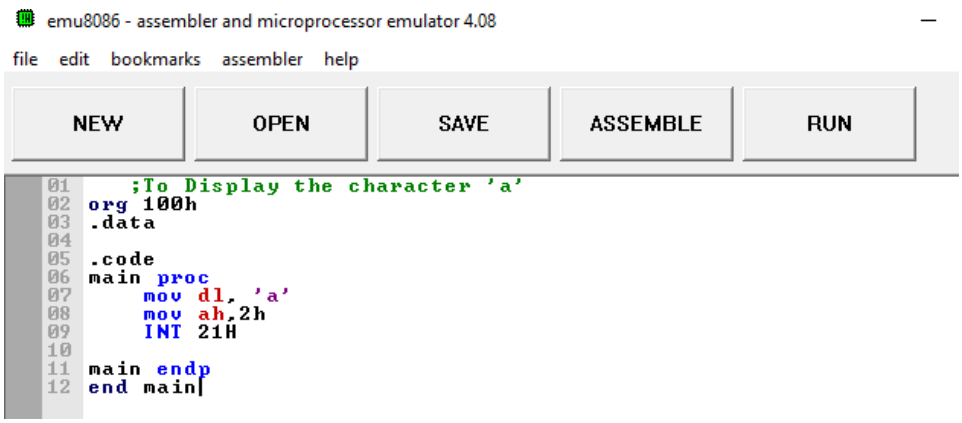
- In 8086 assembly language, we use a software interrupt mechanism for I/O.
- An interrupt signals the processor to suspend its current activity (i.e. running your program) and to pass control to an interrupt service program (i.e. part of the operating system).
- A software interrupt is one generated by a program (as opposed to one generated by hardware).

- The 8086 INT instruction generates a software interrupt.
- For I/O and some other operations, the number used is 21h.
- A Specific number is placed in the register AH to specify which I/O operation (e.g. read a character, display a character) you wish to carry out.
- When the I/O operation is finished, the interrupt service program terminates and program will be resumed at the instruction following int.

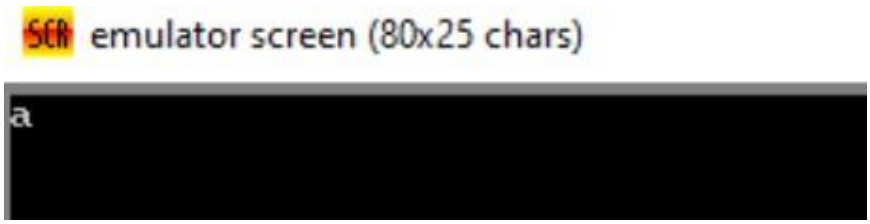
Example # 03 Write a code fragment to display the character 'a' on the screen

```
;To Display the character 'a'
org 100h
.data

.code
main proc
      mov dl, 'a'
      mov ah,2h
      INT 21H

main endp
end main
```

Assembler:



Output:



Character Input from User

To get the input from keyboard a subprogram at **1h** will be called. First 1h will be placed in **ah** and then aninterrupt **21h** generated to call the subprogram. Finally the character will be placed in **al** register.

Example

```
mov ah,1h          ; Keyboard input subprogram
INT 21H            ; Call the subprogram to get character input and  stored in al
```

Note:-

- Carriage Return ASCII (0DH) is the control character to bring the cursor to the start of a line.

```
;display Return
mov dl, 0dh
mov ah, 2h
int 21h ; display Carriage Return
```

- Line-feed ASCII (oAh) is the control character that brings the cursor down to the next line on the screen.

```
;display Line-feed
mov dl, 0ah
mov ah, 2h
int 21h ; display Line Feed
```

Example # 4: Write a program to read and display a character.

```
;To read a character from keyboard
org 100h
.data
    inMsg db "Enter a character
",'$'
    newline db 0dh,0ah,'$'
    outMsg db "User enter
character ",'$'
.code
main proc

    ;To display input message
      mov ax, @data
      mov ds, ax
      mov dx, offset inMsg
   mov ah, 9
      INT 21H

      ;Get input from keyboard
      mov ah,1h
      INT 21H
      mov bl, al

      ;Move cursor to new line
      mov ax, @data
```

Assembler:

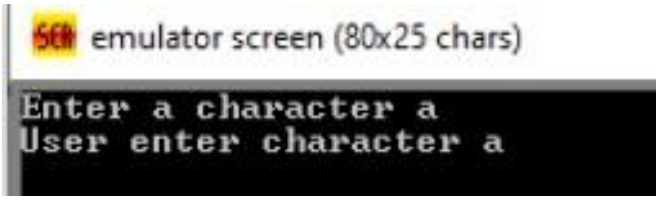
emu8086 - assembler and microprocessor emulator 4.08

file edit bookmarks assembler help

NEW OPEN SAVE ASSEMBLE

```
        ;To read a character from keyboard
org 100h
.data
    inMsg db "Enter a character ",'$'
    newline db 0dh,0ah,'$'
    outMsg db "User enter character ",'$'
.code
main proc

    ;To display input message
    mov ax, @data
    mov ds, ax
    mov dx, offset inMsg
mov ah, 9
    INT 21H

    ;Get input from keyboard
    mov ah,1h
    INT 21H
    mov bl, al

    ;Move cursor to new line
    mov ax, @data
    mov ds, ax
    mov dx, offset newline
    mov ah, 9
    INT 21H

    ;To display ouput message
    mov ax, @data
    mov ds, ax
    mov dx, offset outMsg
mov ah, 9
    INT 21H

    ;To diplay character placed in dl
    mov dl, bl
    mov ah,2h
    INT 21H

    mov ax,4C00h
    INT 21H
main endp
```

Output:



String Output

- A string is a list of characters treated as a unit.
- In 8086 assembly language, single or double quotes may be used to denote a string constant.
- For Defining String Variables following 3 definitions are equivalent ways of defining a string "abc":

- var1 db "abc" ; string constant
- var2 db ‘a’, ‘b’, ‘c’ ; character constants

var3 db 97, 98, 99 ; ASCII codes

- The first version simply encloses the string in quotes. (preferred method)
- The second version defines a string by specifying a list of the character constants that make up the string.
- The third version defines a string by specifying a list of the ASCII codes that make up the string
- In order to display string using MS-DOS subprogram (number 9h), the string must be terminated with the‘$’ character.
- In order to display a string we must know where the string begins and ends.
- The beginning of string is given by obtaining its address using the offset operator.
- The end of a string may be found by either knowing in advance the length of the string or by storing aspecial character at the end of the string.

TASK # 01

Write a program that input a character from user 2 times. The program will display the character enteredby user 2 times on screen in newline(without using a loop).

Sample input vs output:

Please enter the character? A
you have entered the character A
you have entered the character A
Please enter another character? B
you have entered the character B
you have entered the character B

TASK # 02

Write a program that input a character from user is in lowercase, the program will convert it to uppercaseand will display it on console after conversion.

Hint: - The ASCII codes for lowercase letters (a-z) are 97-122. In order to convert a lowercase letter to uppercase letter, just subtract 32 from its ASCII code.

Sample input vs output:

Please enter an alphabet in lower case? a
Upper case of the input is : A

TASK # 03

Write an Assembly code which take an alphabet from user and print the next and previous alphabets onthe screen.

Sample input vs output:

Please enter an alphabet ? d
Previous alphabet in English grammar is : c
You have entered alphabet : d
Next alphabet in English grammar is : e

Task-1:
Assembler:

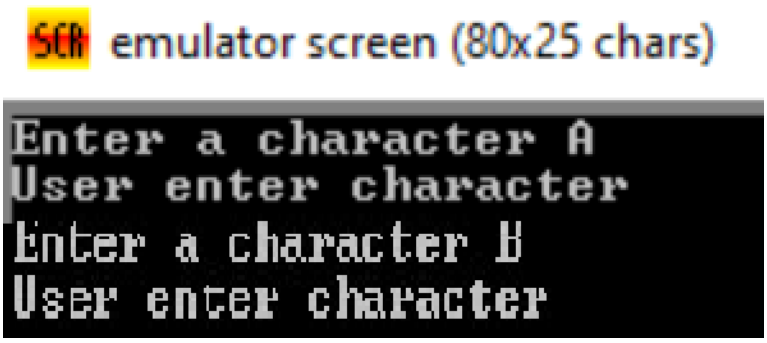
emu8086 - assembler and microprocessor emulator 4.08

file edit bookmarks assembler help

NEW OPEN SAVE ASSEMBLE

```
;To read two charactera from keyboard
org 100h
org 100h
.data
    inMsg db "Enter a character ",'$'
    newline db 0dh,0ah,'$'
    outMsg db "User enter character ",'$'

.code
main proc

    ;To display input message
    mov ax, @data
    mov ds, ax
    mov dx, offset inMsg
mov ah, 9
    INT 21H

    ;Get input of first charactera from keyboard
    mov ah,1h
    INT 21H
    mov bl, al
    ;Get input of second charactera from keyboard
    mov ah,1h
    INT 21H
    mov bl, al

    ;Move cursor to new line
    mov ax, @data
    mov ds, ax
    mov dx, offset newline
    mov ah, 9
    INT 21H

    ;To display ouput message
    mov ax, @data
    mov ds, ax
    mov dx, offset outMsg
mov ah, 9
    INT 21H

    ;To diplay character placed in dl
    mov dl, bl
    mov ah,2h
    INT 21H

    mov ax,4C00h
mov dl, bl
        mov ah,2h
        INT 21H
```

Output:



Task-2:

Assembler:

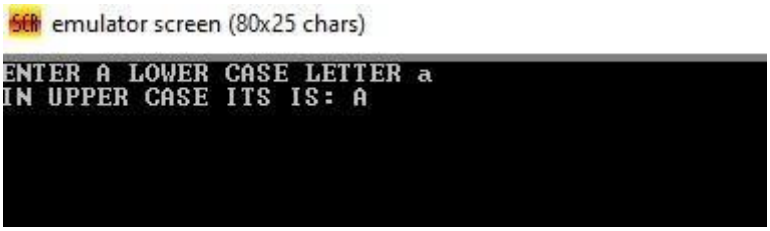
emu8086 - assembler and microprocessor emulator 4.08

file edit bookmarks assembler help

NEW OPEN SAVE ASSEMBLE RU

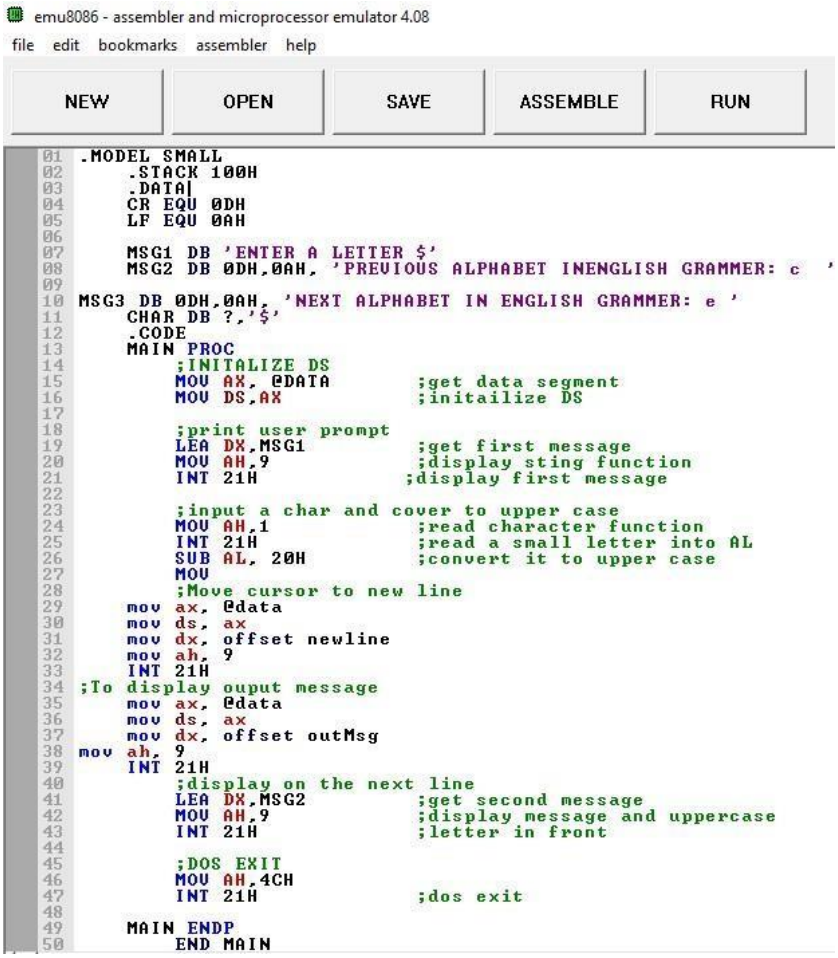
```
        .MODEL SMALL
.STACK 100H
.DATA

CR EQU 0DH
LF EQU 0AH

MSG1 DB 'ENTER A LOWER CASE LETTER $'
MSG2 DB 0DH,0AH, 'IN UPPER CASE ITS IS: '
CHAR DB ?,'$'

.CODE

MAIN PROC
    ;INITALIZE DS
    MOV AX, @DATA       ;get data segment
    MOV DS,AX           ;initailize DS

    ;print user prompt
    LEA DX,MSG1         ;get first message
    MOV AH,9            ;display sting function
    INT 21H            ;display first message

    ;input a char and cover to upper case
    MOV AH,1            ;read character function
    INT 21H             ;read a small letter into AL
    SUB AL, 20H         ;convert it to upper case
    MOV CHAR, AL        ;and store it

    ;display on the next line
    LEA DX,MSG2         ;get second message
    MOV AH,9            ;display message and uppercase
    INT 21H             ;letter in front

    ;DOS EXIT
    MOV AH,4CH
    INT 21H             ;dos exit

MAIN ENDP
    END MAIN
```

Output:



Task-3:

Assembler:




```
.MODEL SMALL
    .STACK 100H
    .DATA
    CR EQU 0DH
    LF EQU 0AH

    MSG1 DB 'ENTER A LETTER $'
    MSG2 DB 0DH,0AH, 'PREVIOUS ALPHABET INENGLISH GRAMMER: c  '

MSG3 DB 0DH,0AH, 'NEXT ALPHABET IN ENGLISH GRAMMER: e '
    CHAR DB ?,'$'
    .CODE
    MAIN PROC
        ;INITALIZE DS
        MOV AX, @DATA       ;get data segment
        MOV DS,AX           ;initailize DS

        ;print user prompt
        LEA DX,MSG1         ;get first message
        MOV AH,9            ;display sting function
        INT 21H            ;display first message

        ;input a char and cover to upper case
        MOV AH,1            ;read character function
        INT 21H             ;read a small letter into AL
        SUB AL, 20H         ;convert it to upper case
        MOV
        ;Move cursor to new line
    mov ax, @data
    mov ds, ax
    mov dx, offset newline
    mov ah, 9
    INT 21H
;To display ouput message
    mov ax, @data
    mov ds, ax
    mov dx, offset outMsg
mov ah, 9
    INT 21H
        ;display on the next line
        LEA DX,MSG2         ;get second message
        MOV AH,9            ;display message and uppercase
        INT 21H             ;letter in front

        ;DOS EXIT
        MOV AH,4CH
        INT 21H             ;dos exit

    MAIN ENDP
        END MAIN
```

Output: