Digital Logic Design Lab practical (CS236)

Submitted To	:	Palak Jain
Submitted By	:	Harsh Chandravanshi
Roll No.	:	27
Enrolment No.	:	2019/CTAE/145
Section	:	A
Semester	:	3rd
Department	:	Computer Science Engineering

INDEX

S. No.	Experiment Name	Date of submission	Teachers remarks/sign
1	To implement basic logic gates	15/12/20	
2	To implement exclusive logic gates	15/12/20	
3	To implement universal logic gates	15/12/20	
4	To implement XOR logic gate using NAND & Nor gate	15/12/20	
5	To implement XNOR logic gate using NAND & Nor gate	15/12/20	
6	To implement half adder & full adder	15/12/20	
7	To implement half Subtractor & full Subtractor	15/12/20	
8	To implement 4:1 mux & 8:1 mux	15/12/20	
9	To implement 1:4 demux & 1:8 demux	15/12/20	

10	To implement 4:2 encoder & 8:3 encoder	15/12/20	
11	To implement 2:4 decoder & 3:8 decoder	15/12/20	
12	To implement S-R flip Flop	15/12/20	
13	To implement J-K flip Flop	15/12/20	
14	To implement D flip Flop	15/12/20	
15	To implement T flip Flop	15/12/20	

EXPERIMENT- 1

Aim - To implement basic logic gates

- Logic Gates are a block of hardware that produces signals of binary 1 or 0 when input logic requirements are satisfied.
- Each gate has a distinct graphic symbol, and its operation can be described by means of algebraic expressions.
- The relationship between the input-output binary variables for each gate can be represented in tabular form by a truth table.

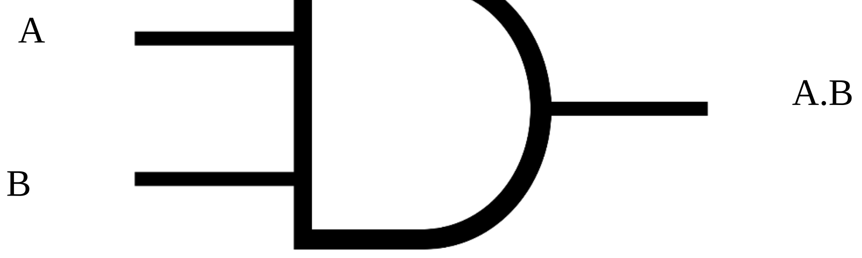
There are 3 basic logic gates -: AND GATE, OR GATE, NOT GATE

(1) AND GATE-

- An AND gate is a logic gate having two or more inputs and a single output.
- An AND gate operates on logical multiplication rules.
- The AND gate is an electronic circuit which gives a high output only if all its inputs are high.
- The AND operation is represented by a dot (.) sign.
- The AND operation can be represented by the equation as

$$Y=A.B$$

(ii) Diagram of AND GATE-



(iii) Truth Table-

INPUT		**OUTPUT**
A	**B**	Y
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

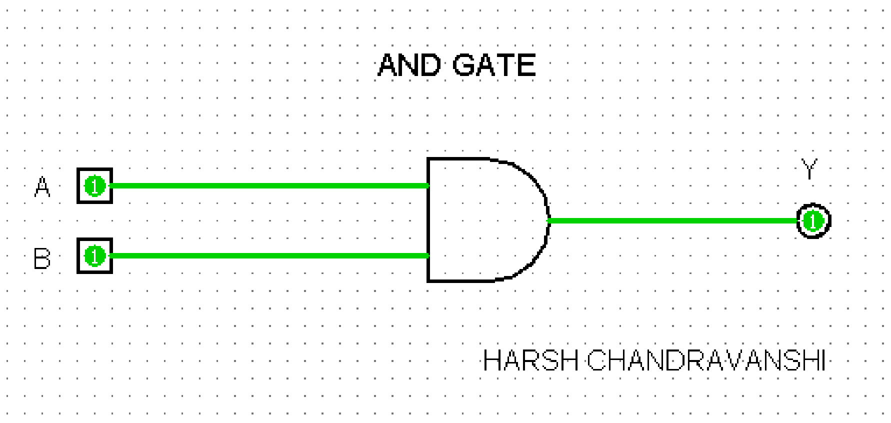
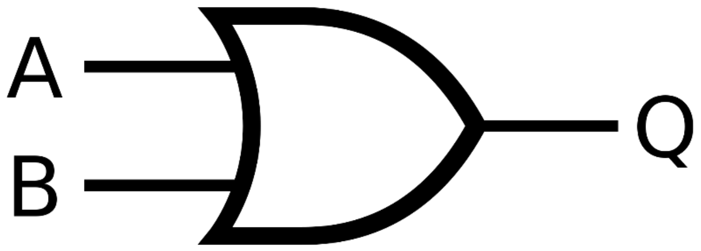


Fig-1 AND GATE using logisim

(2) OR GATE-

- An OR gate is a logic gate having two or more inputs and a single output.
- An OR gate operates on logical addition rules.
- The OR gate is an electronic circuit that gives a high output (1) if **one or more** of its inputs are high.
- A plus (+) is used to show the OR operation.
- The OR operation can be represented by the equation as Y=A+B

(ii) Diagram of OR GATE-



(iii) Truth Table-

INPUT		**OUTPUT**
A	**B**	Y
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

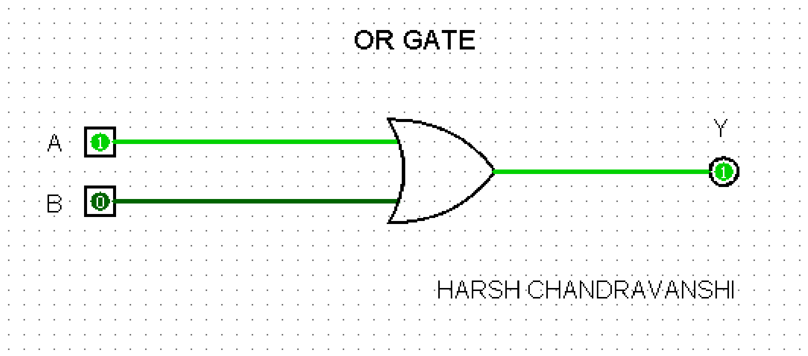
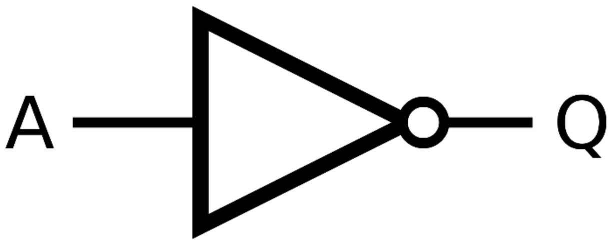


Fig-2 OR GATE using logisim

(3) NOT GATE-

- The NOT gate is an electronic circuit that produces an inverted version of the input at its output.
- It is also known as an *inverter*.
- If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top.
- The NOT gate is a forward arrow with a small circle at the output. The circle part of the symbol is what says that the output is negating the input.

(ii) Diagram of NOT GATE-



(ii) Truth Table-

INPUT	OUTPUT
A	Y
0	1
1	0

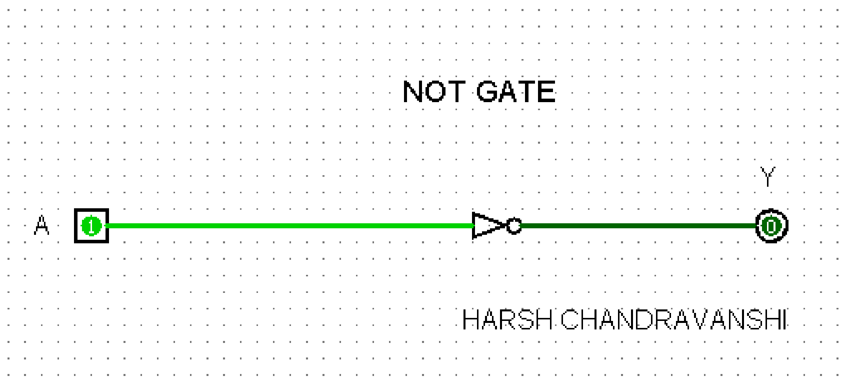


Fig-3 NOT GATE using logisim

EXPERIMENT- 2

Aim - To implement exclusive logic gates

- Logic Gates are a block of hardware that produces signals of binary 1 or 0 when input logic requirements are satisfied.
- Each gate has a distinct graphic symbol, and its operation can be described by means of algebraic expressions.
- The relationship between the input-output binary variables for each gate can be represented in tabular form by a truth table.

There are 2 exclusive logic gates -: XOR GATE, XNOR GATE

(1) XOR GATE

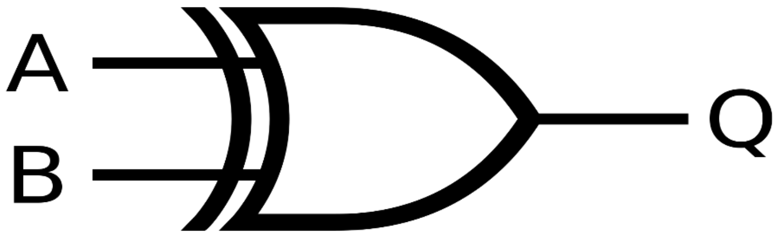
- The XOR logic function (which stands for "Exclusive OR") returns true if either of its inputs differ, and false if they are all the same.
- In other words, if its inputs are a combination of true and false, the output of XOR is true. If its inputs are all true or all false, the output of XOR is false (0).
- In Boolean algebra, the XOR value of two inputs A and B can be written as $A \oplus B$ (the XOR symbol, $\oplus$, resembles a plus sign inside a circle).

$$Y=A\oplus B$$

Or

$$Y=A'B + AB'$$

(ii) Diagram of XOR GATE-



(iii) Truth Table-

INPUT		OUTPUT
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

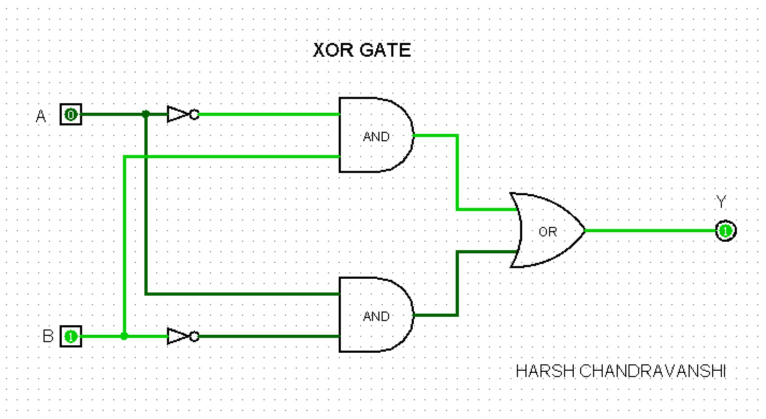


Fig-1 XOR GATE using logisim

(2) XNOR GATE

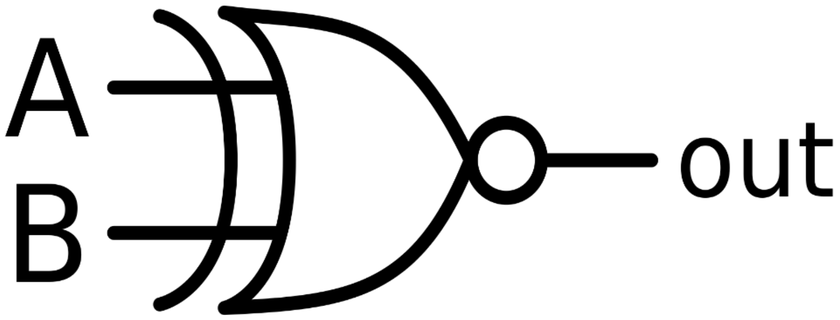
- The XNOR logic operation (which stands for "Exclusive NOT OR") returns true (1) if both the inputs are same, and false (0) if inputs are different.
- In other words, if its inputs are a combination of true (1) and false (0), then the output of XNOR is false (0). If inputs are all true of all false, the output of XNOR is true(1).
- In Boolean Algebra, the XNOR value of two inputs A and B can be written as $(A\oplus B)'$

$$Y=(A\oplus B)'$$

Or

$$Y=AB + A'B'$$

(ii) Diagram of XNOR GATE



(iii) Truth Table-

INPUT		OUTPUT
A	**B**	Y
0	**0**	**1**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

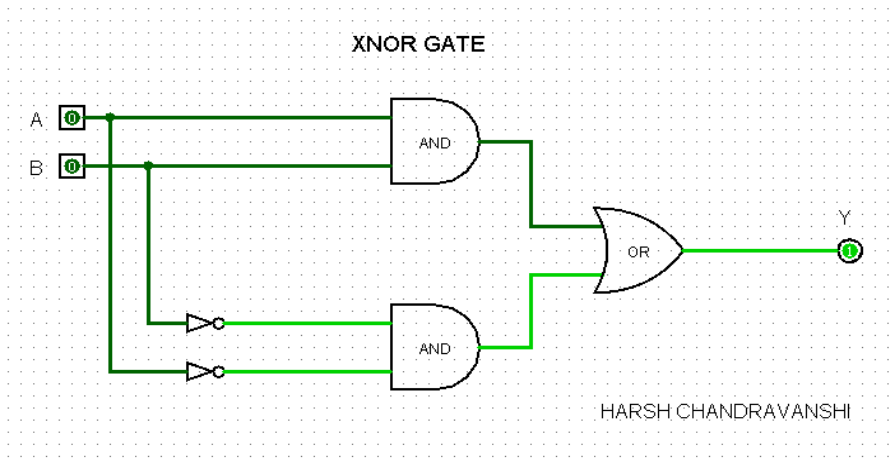


Fig-2   XNOR GATE using logisim

EXPERIMENT- 3

Aim - To implement universal logic gates

- NAND & NOR gates are called universal gates.
- We can implement any Boolean function, which is in the sum of products, by using NAND gate alone.
- We can implement any Boolean function, which is in product of sums from by using NOR gate alone.

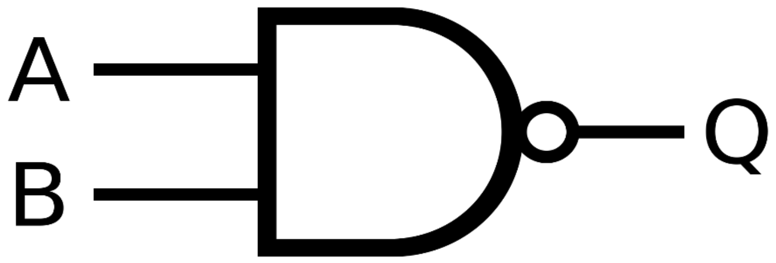
There are 2 universal logic gates -: NAND GATE, NOR GATE

(1) NAND GATE-

- This is a NOT- AND gate which is equal to an AND gate followed by a NOT gate.
- The outputs of all NAND gates are high (1) if any of the inputs are low (0).
- The symbol is an AND gate with a small circle on the output. The small circle represents inversion.
- In Boolean algebra, the NAND value of two inputs A and B can be written as (AB)’ (AB with an over score).
- NAND Gate is also called as the “Universal Logic Gate” because any other logic operation can be created by using only NAND Gates.

$$Y = (A.B)'$$

(ii) Diagram of NAND GATE-



(iii) Truth Table-

INPUT		**OUTPUT**
A	**B**	Y
0	**0**	**1**
0	**1**	**1**
1	**0**	**1**
1	**1**	**0**

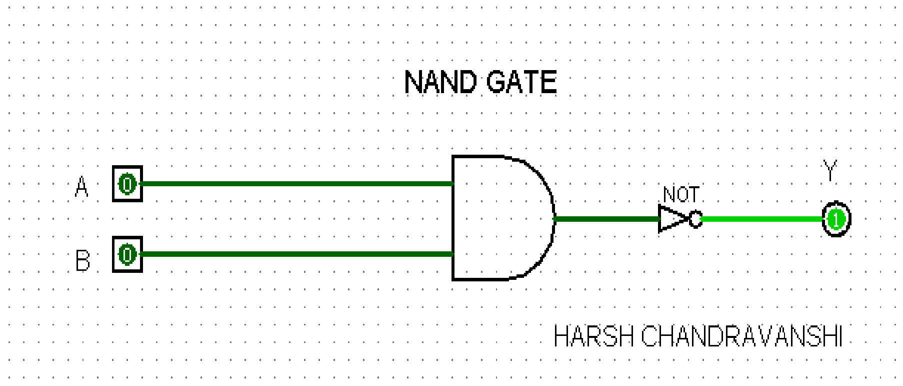


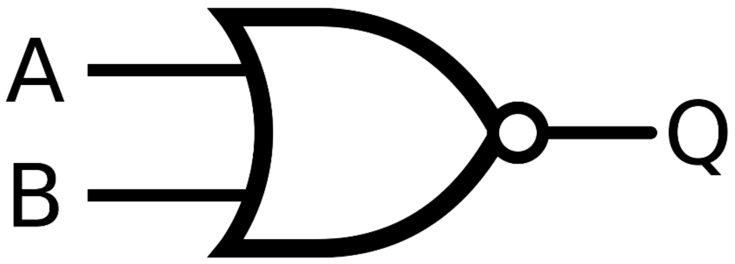
Fig-1 NAND GATE using Logisim

(2) NOR GATE-

- This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate.
- The outputs od all NOR gates are low (0) if any of the inputs are high (1).
- The symbol is an OR gate with a small circle on the output. The small circle represents inversion.
- In Boolean algebra, the NOR value of two inputs A and B can be written as (A+B)’ (A+B with an over score).
- NOR Gate is also called “Universal Logic Gate” because any other logic operation can be created using only NOR Gates.

$$Y= (A+B)'$$

(ii) Diagram of NOR GATE-



(iii) Truth Table-

INPUT		**OUTPUT**
A	**B**	Y
0	**0**	**1**
0	**1**	**0**
1	**0**	**0**
1	**1**	**0**

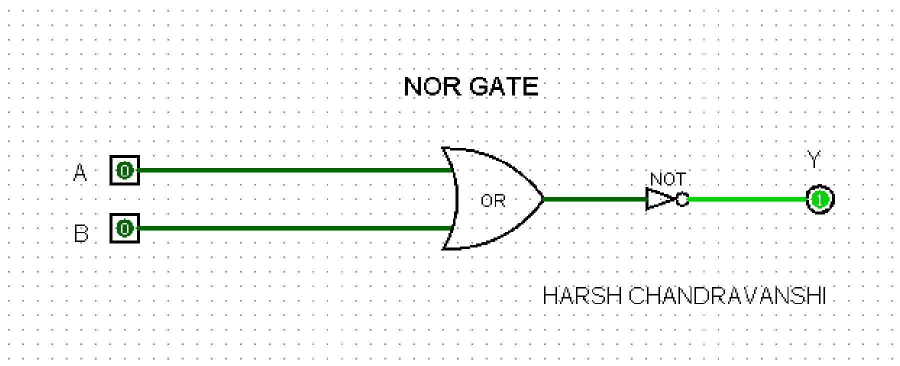


Fig-2 NOR GATE using logisim

EXPERIMENT- 4

Aim - To implement XOR logic gate using NAND & Nor gate

(1) XOR GATE USING NAND

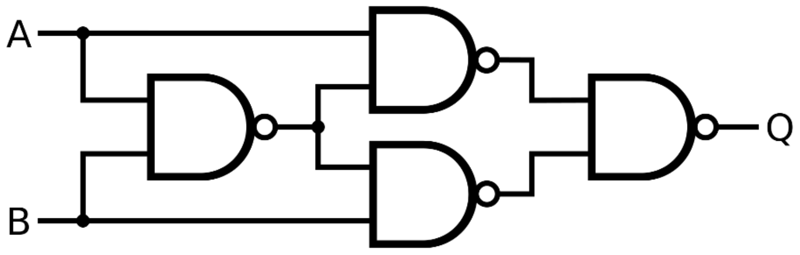
- The XOR logic function (which stands for "Exclusive OR") returns true if either of its inputs differ, and false if they are all the same.
- In other words, if its inputs are a combination of true and false, the output of XOR is true. If its inputs are all true or all false, the output of XOR is false (0).
- In Boolean algebra, the XOR value of two inputs A and B can be written as A ⊕ B (the XOR symbol, ⊕, resembles a plus sign inside a circle).

$$Y=A\oplus B$$

Or

$$Y=A'B + AB'$$

(ii) Diagram of XOR GATE USING NAND



(iii) Truth Table-

INPUT		**OUTPUT**
A	**B**	Y
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**0**

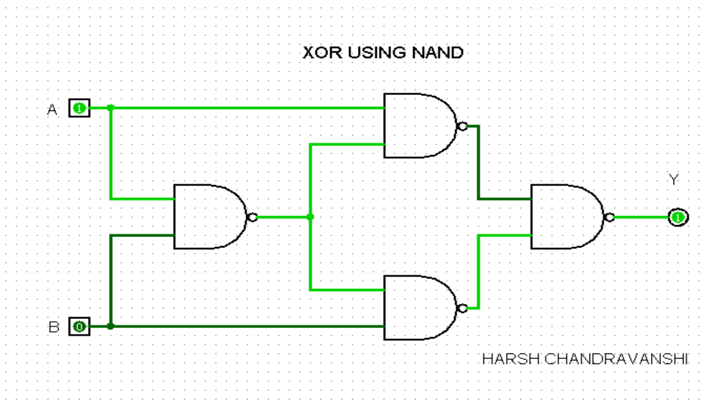


Fig-1 XOR GATE USING NAND using logisim

(2) XOR GATE USING NOR

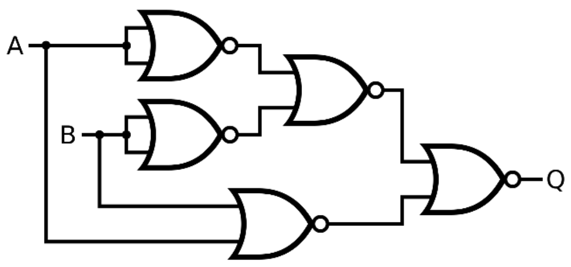
- The XOR logic function (which stands for "Exclusive OR") returns true if either of its inputs differ, and false if they are all the same.
- In other words, if its inputs are a combination of true and false, the output of XOR is true. If its inputs are all true or all false, the output of XOR is false (0).
- In Boolean algebra, the XOR value of two inputs A and B can be written as $A \oplus B$ (the XOR symbol, $\oplus$, resembles a plus sign inside a circle).

$$Y=A\oplus B$$

Or

$$Y=A'B + AB'$$

(ii) Diagram of XOR GATE USING NOR



(iii) Truth Table-

INPUT		**OUTPUT**
A	**B**	Y
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**0**

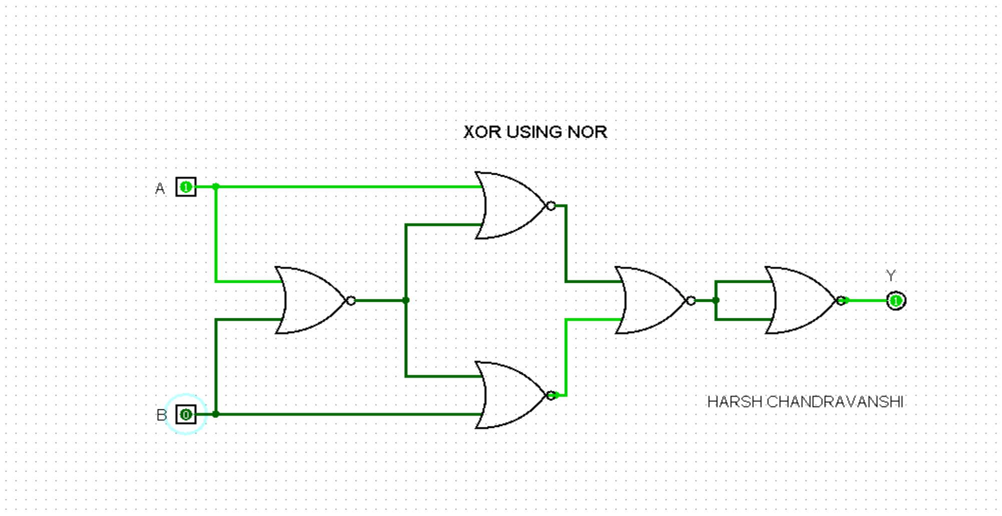


Fig-2 XOR GATE USING NOR using logisim

EXPERIMENT- 5

Aim - To implement XNOR logic gate using NAND & NOR gate

(1) XNOR GATE USING NAND

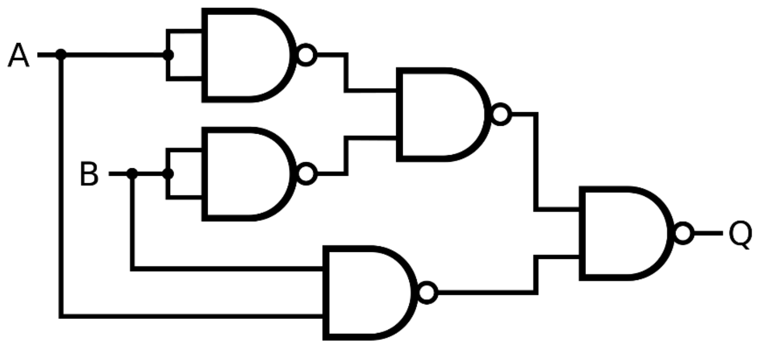
- The XNOR logic operation (which stands for "Exclusive NOT OR") returns true (1) if both the inputs are same, and false (0) if inputs are different.
- In other words, if its inputs are a combination of true (1) and false (0), then the output of XNOR is false (0). If inputs are all true of all false, the output of XNOR is true(1).
- In Boolean Algebra, the XNOR value of two inputs A and B can be written as $(A \oplus B)'$

$$Y=(A \oplus B)'$$

Or

$$Y=AB + A'B'$$

(ii) Diagram of XNOR GATE



(iii) Truth Table-

INPUT		OUTPUT
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

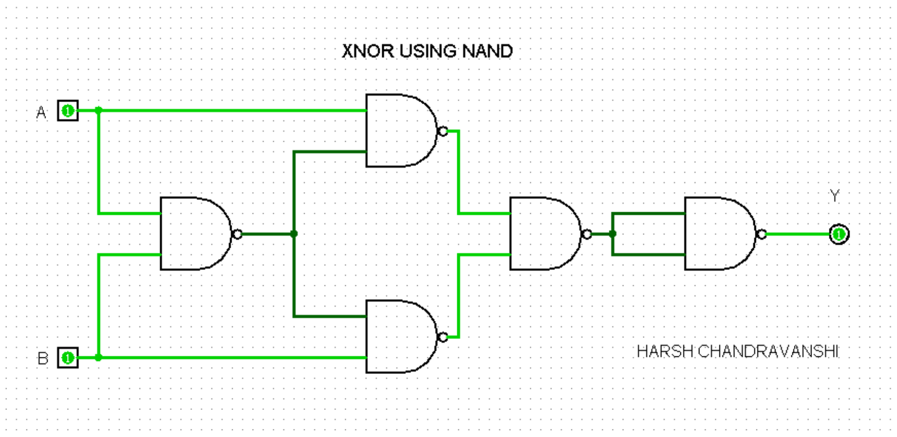


Fig-1 XNOR GATE USING NAND using logisim

(2) XNOR GATE USING NOR

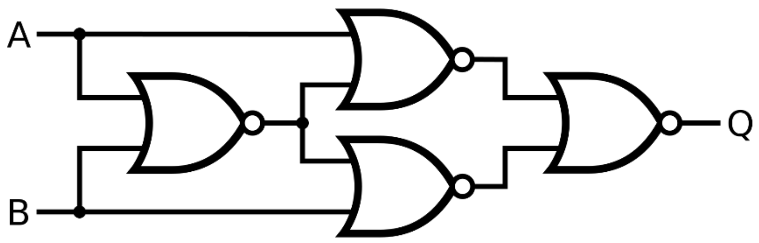
- The XNOR logic operation (which stands for “Exclusive NOT OR”) returns true (1) if both the inputs are same, and false (0) if inputs are different.
- In other words, if its inputs are a combination of true (1) and false (0), then the output of XNOR is false (0). If inputs are all true of all false, the output of XNOR is true (1).
- In Boolean Algebra, the XNOR value of two inputs A and B can be written as $(A \oplus B)'$

$$Y=(A \oplus B)'$$

Or

$$Y=AB + A'B'$$

(iv) Diagram of XNOR GATE



(v) Truth Table-

INPUT		OUTPUT
A	**B**	Y
0	**0**	**1**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

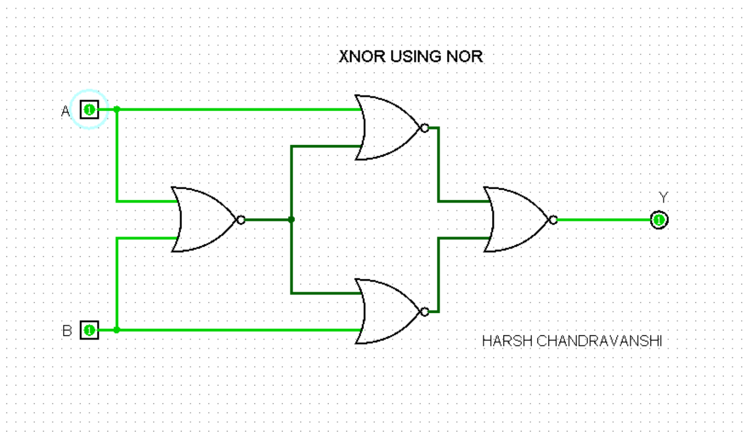


Fig-2 XNOR GATE USING NOR using logisim

EXPERIMENT- 6

Aim - To implement half adder & full adder.

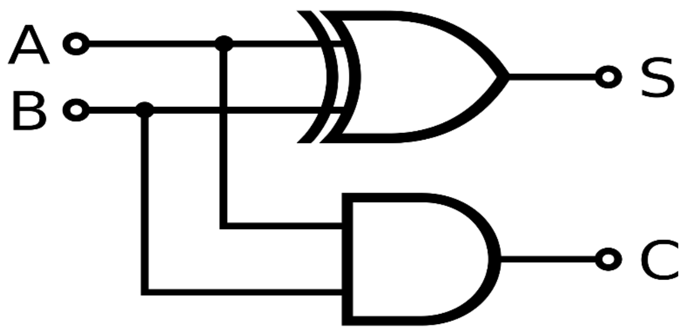
- A combinational logic circuit that performs the addition of two single bits is called Half Adder.
- A combinational logic circuit that performs the addition of three single bits is called Full adder

(1) Half Adder

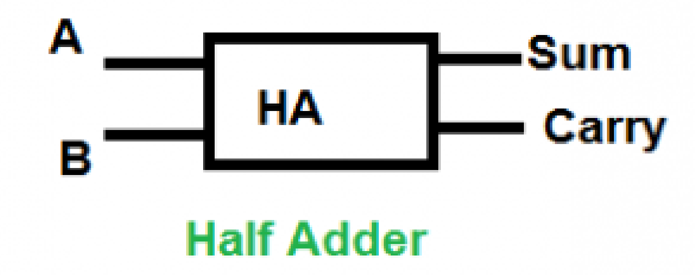
- The addition of 2-bits is called Half Adder.
- The input variables are augend and addend bits and output variables are sum and carry bits. A and B are two input bits.

SUM=A XOR B
Carry= A.B

(ii) Diagram of HALF ADDER



IMPLEMENTATION



BLOCK DIAGRAM

(ii) Truth Table-

A	**B**	**SUM**	CARRY
0	**0**	**0**	**0**
0	**1**	**1**	**0**
1	**0**	**1**	**0**
1	**1**	**0**	**1**

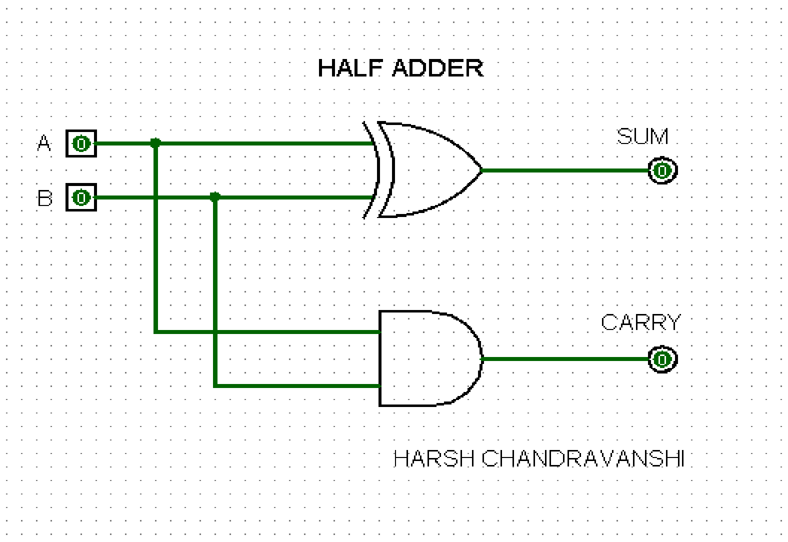


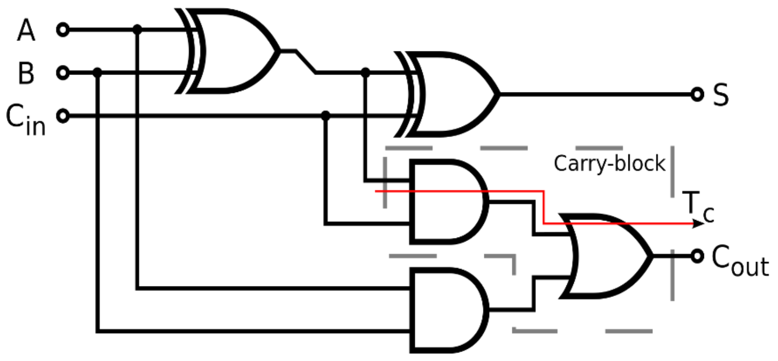
Fig-1 HALF ADDER using logisim

(2) FULL Adder

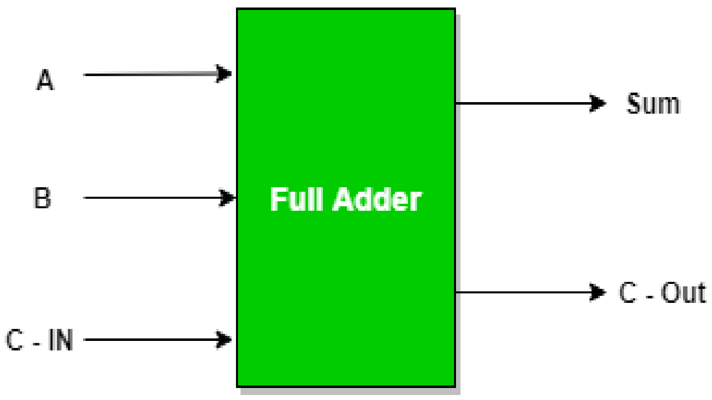
- Full Adder is the adder which adds three inputs and produces two outputs.
- The first two inputs are A and B and the third input is an input carry such as C-IN.
- The output carry is designated as C-OUT and the normal output is designated as S which is SUM.
- A full adder logic is designed in such a manner that can take eight inputs together to create a byte-wide adder and cascade the carry bit from one adder to the another.

SUM=A XOR B XOR C
Carry= AB + BC + CA

(ii) Diagram of FULL ADDER



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

A	B	C	SUM	CARRY
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

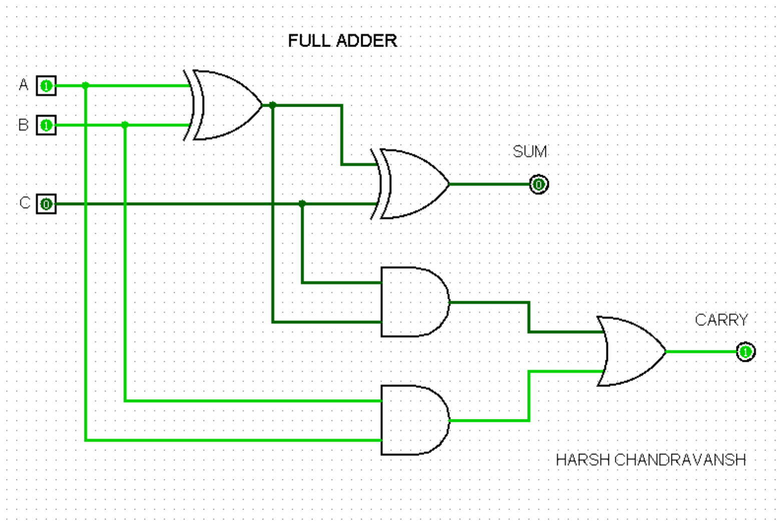


Fig-2 FULL ADDER using logisim

EXPERIMENT- 7

Aim - To implement half subtractor & full subtractor.

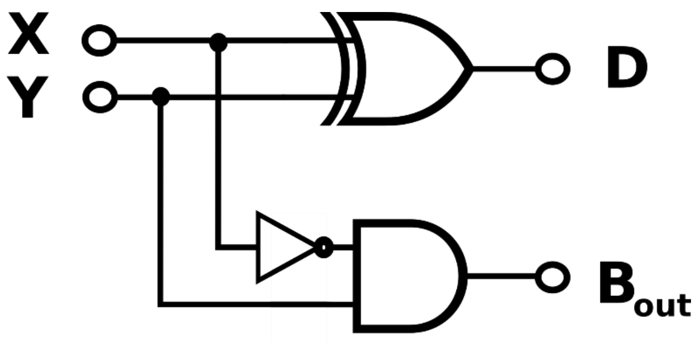
- As their name implies, a **Binary Subtractor** is a decision-making circuit that subtracts two binary numbers from each other, for example, X – Y to find the resulting difference between the two numbers.
- Unlike the Binary Adder which produces a SUM and a CARRY bit when two binary numbers are added together, the *binary subtractor* produces a DIFFERENCE, D by using a BORROW bit, B from the previous column. Then obviously, the operation of subtraction is the opposite to that of addition.

(1) Half Subtractor

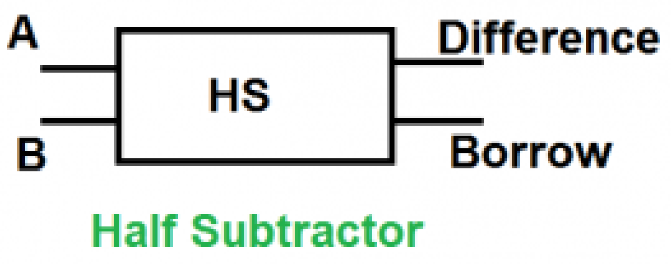
- It is a combinational logic circuit designed to perform subtraction of two single bits.
- It contains two inputs (A and B) and produces two outputs (Difference and Borrow output).

Difference=A XOR B
Borrow= A'. B

(ii) Diagram of HALF SUBTRACTOR



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

A	**B**	**Difference**	Borrow
0	**0**	**0**	**0**
0	**1**	**1**	**1**
1	**0**	**1**	**0**
1	**1**	**0**	**0**

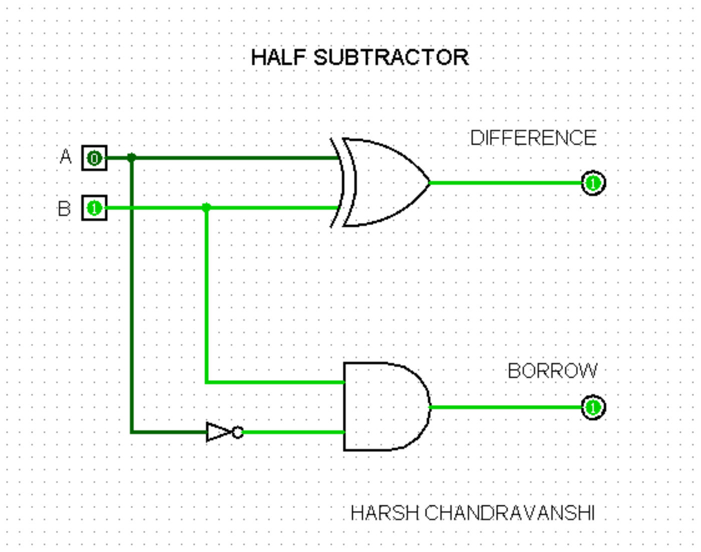


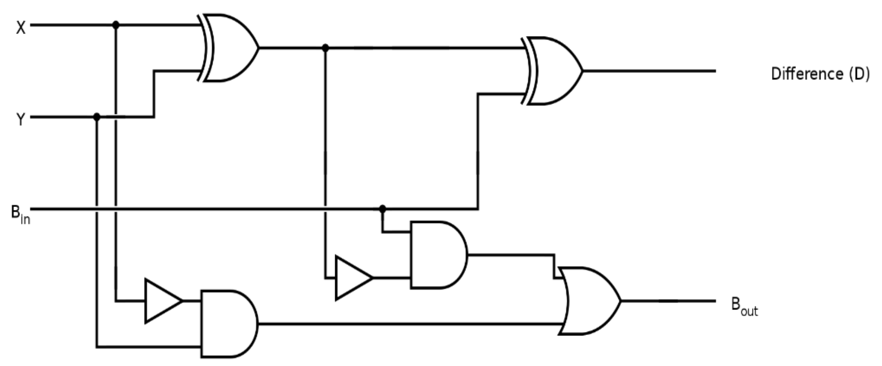
Fig-1 HALF SUBTRACTOR using logisim

(2) Full Subtractor

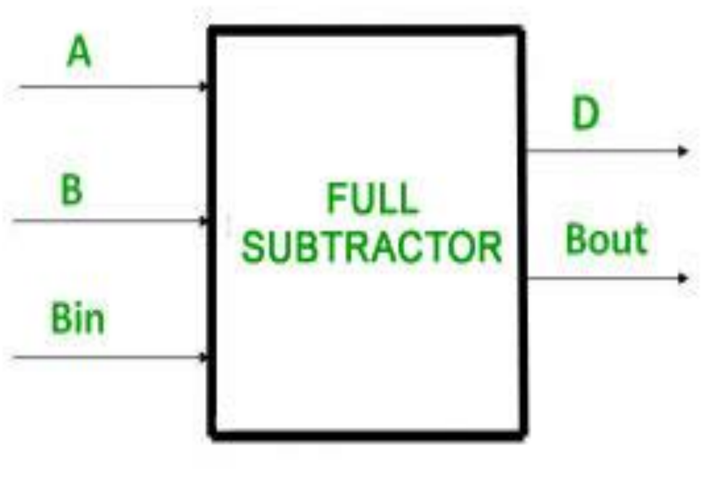
- It is a combinational logic circuit designed to perform subtraction of three single bits.
- It contains three inputs (A, B and Bin) and produces two outputs (Difference and Borrow output).
- Where A and B are called Minuend and Subtrahend bits.
- And, Bin -> Borrow-In and Bout -> Borrow-Out

Difference=A XOR B XOR C
Borrow= A’B + A’C + BC

(ii) Diagram of FULL SUBTRACTOR



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

A	B	C(Bin)	Difference	Borrow
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

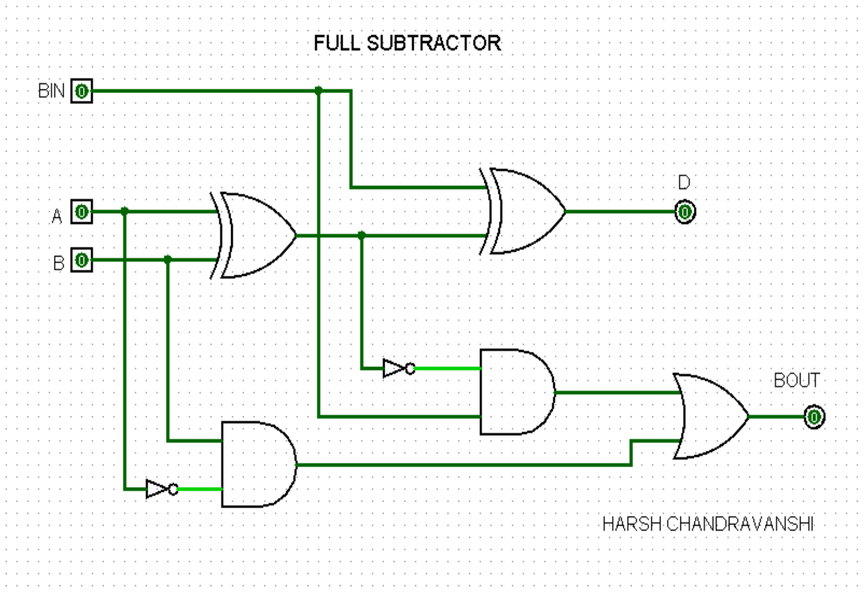


Fig-2 FULL SUBTRACTOR using logisim

EXPERIMENT- 8

Aim - To implement 4:1 mux & 8:1 mux.

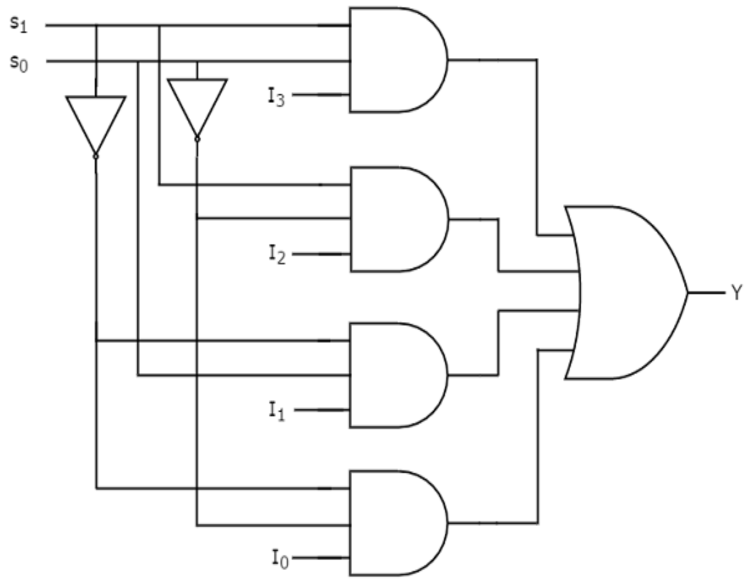
- Multiplexer means many-to-one
- Multiplexer is a data selector which takes several inputs and gives a single output.
- In Multiplexer we have 2^n input lines and 1 output lines where n is the number of selection lines.
- The multiplexer used for digital applications, also called digital multiplexer, is a circuit with many inputs but only one output.
- Few types of multiplexer are 2-to-1, 4-to-1, 8-to-1, 16-to-1 multiplexer.

(1) 4:1 Multiplexer

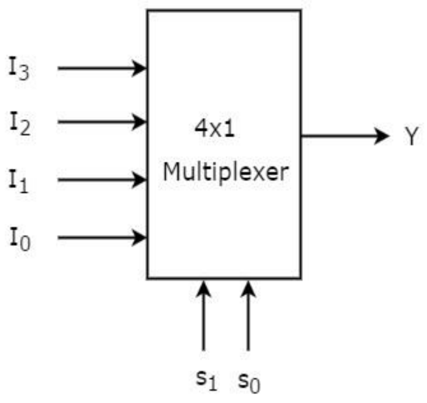
- 4x1 Multiplexer has four data inputs I_3, I_2, I_1 & I_0, two selection lines s_1 & s_0 and one output Y.
- One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines.

$$Y=S_1'S_0'I_0+S_1'S_0I_1+S_1S_0'I_2+S_1S_0I_3$$

(ii) Diagram of 4:1 Multiplexer



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

Selection Lines		**OUTPUT**
S_1	**S_0**	Y
0	**0**	**I_1**
0	**1**	**I_1**
1	**0**	**I_2**
1	**1**	**I_3**

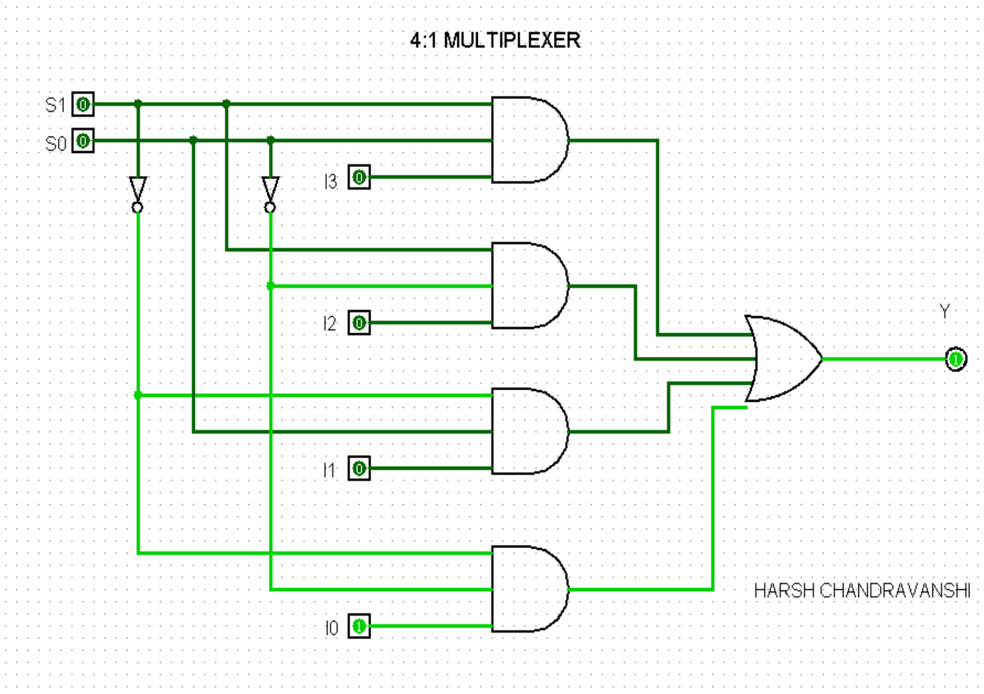


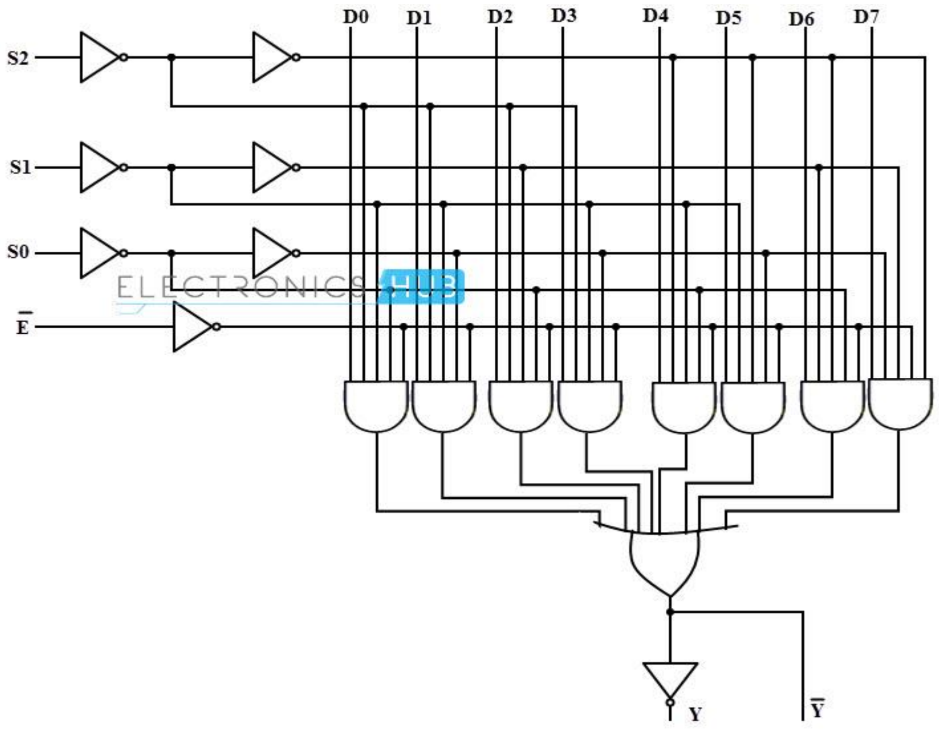
Fig-1 4:1 MUX using logisim

(2) 8:1 Multiplexer

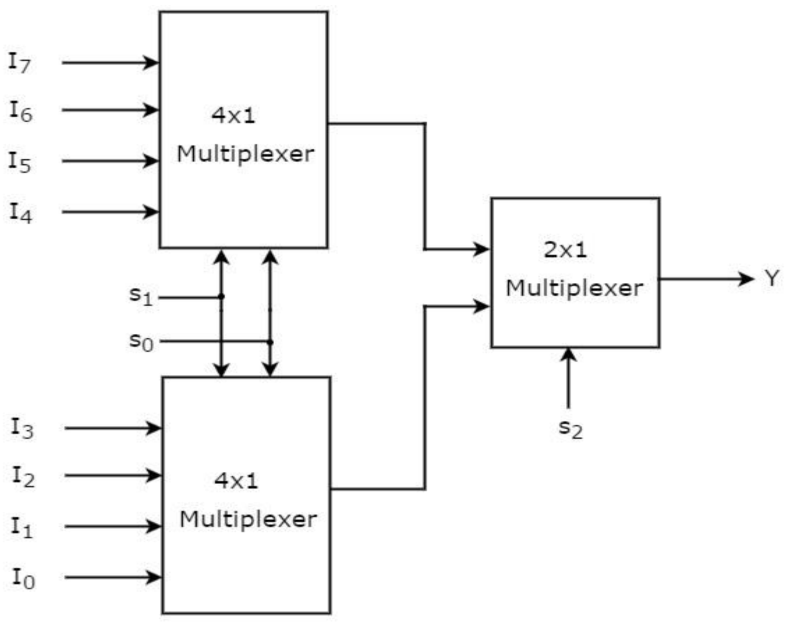
- To implement 8x1 Multiplexer using 4x1 Multiplexers and 2x1 Multiplexer. We know that 4x1 Multiplexer has 4 data inputs, 2 selection lines and one output. Whereas, 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output.
- We require two **4x1 Multiplexers** in first stage in order to get the 8 data inputs. Since, each 4x1 Multiplexer produces one output, we require a **2x1 Multiplexer** in second stage by considering the outputs of first stage as inputs and to produce the final output.
- Let the 8x1 Multiplexer has eight data inputs I_7 to I_0, three selection lines s_2, s_1 & s0 and one output Y.

$$\mathbf{Y=S_2'S_1'S_0'I_0+ S_2'S_1'S_0I_1+ S_2'S_1S_0'I_2+ S_2'S_1S_0I_3 + S_2\, S_1'\, S_0'\, I_4 + S_2\, S_1'\, S_0\, I_5 + S_2\, S_1\, S_0'\, I_6 + S_2\, S_1\, S_0'\, I_6}$$

(ii) Diagram of 8:1 Multiplexer



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

Selection Lines			**OUTPUT**
S_2	**S_1**	**S_0**	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**

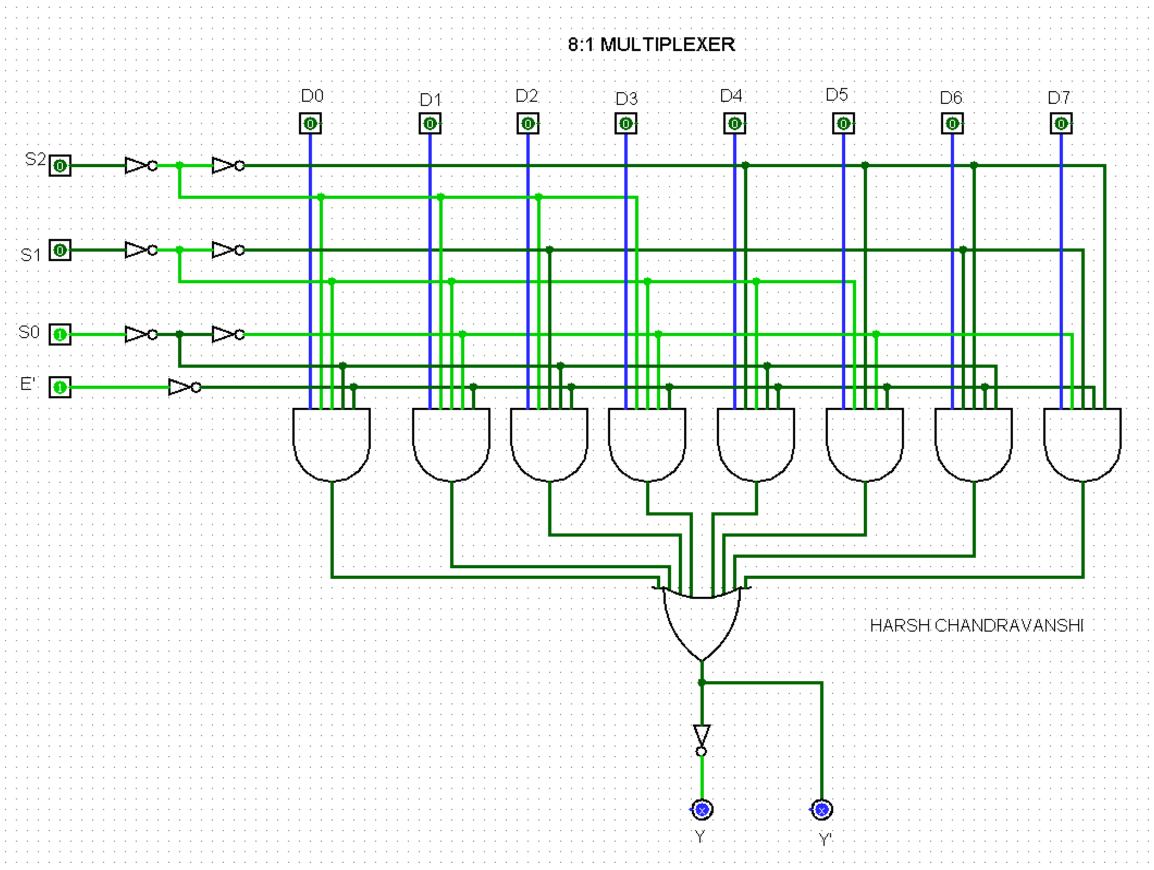


Fig-2 8:1 MUX using logisim

EXPERIMENT- 9

Aim - To implement 1:4 demux & 1:8 demux.

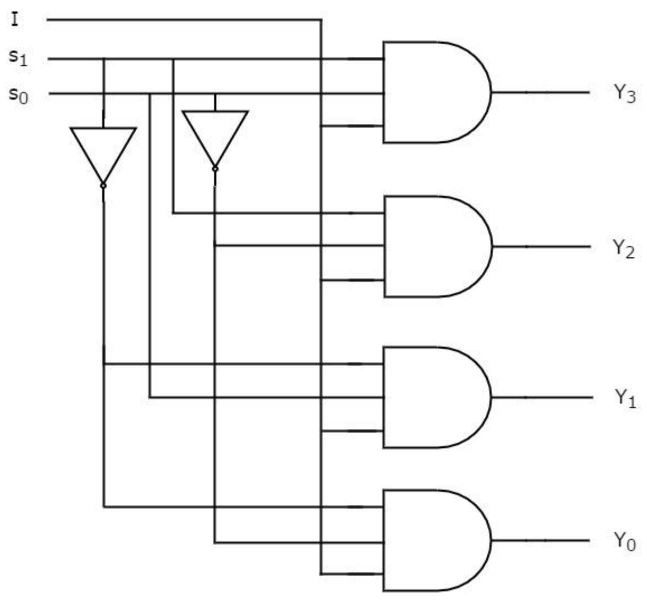
- Multiplexer means one-to-many.
- A Demultiplexer is a circuit with one input and many outputs.
- By applying control signal, we can steer any input to the output.
- Demultiplexer is a data distributor which takes a single input and gives several outputs.
- In demultiplexer we have 1 input and 2^n output lines where n is the selection line.
- Few types of demultiplexer are 1-to-2, 1-to-4, 1-to-8, 1-to-16 demultiplexer.

(1) 1:4 Multiplexer

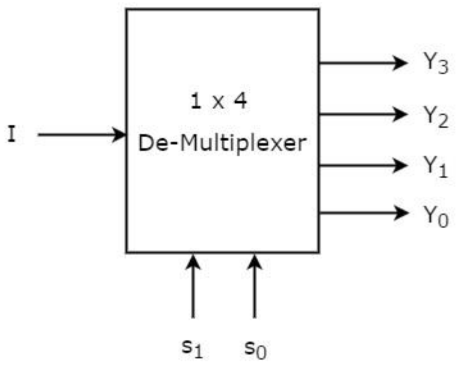
- 1x4 De-Multiplexer has one input I, two selection lines, s1 & s0 and four outputs Y3, Y2, Y1 &Y0.
- The single input 'I' will be connected to one of the four outputs, Y3 to Y0 based on the values of selection lines s1 & s0.

Y3=S1S0I
Y2=S1S0′I
Y1=S1′S0I
Y0=S1′S0′I

(ii) Diagram of 4:1 Multiplexer



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

Selection Lines		**OUTPUT**			
S_1	**S_0**	Y_3	Y_2	Y_1	Y_0
0	**0**	**0**	**0**	**0**	**I**
0	**1**	**0**	**0**	**I**	**0**
1	**0**	**0**	**I**	**0**	**0**
1	**1**	**I**	**0**	**0**	**0**

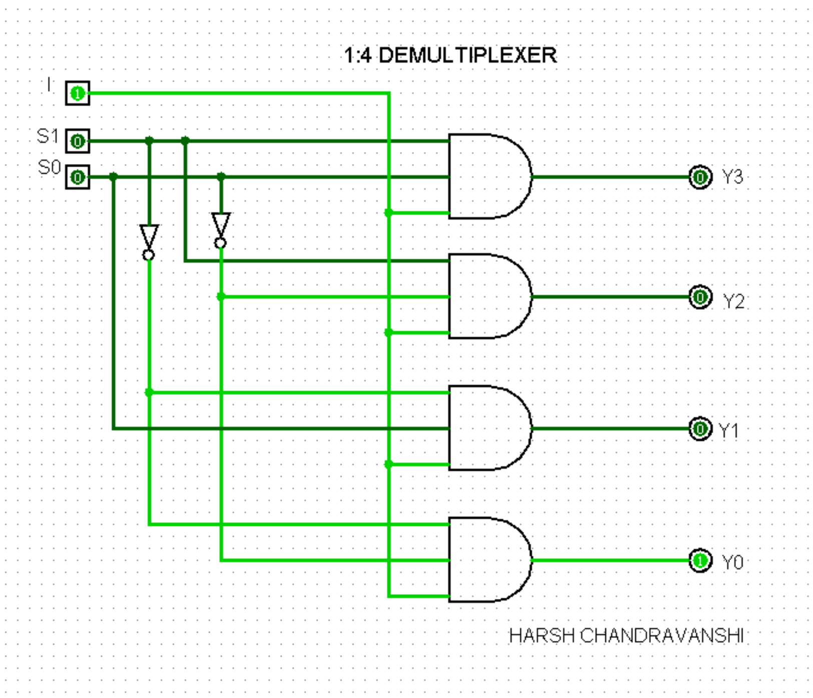


Fig-1 1:4 DEMUX using logisim

(2) 1:8 Multiplexer

- To implement 1x8 De-Multiplexer using 1x4 De-Multiplexers and 1x2 De-Multiplexer.
- We know that 1x4 De-Multiplexer has single input, two selection lines and four outputs.
- Whereas, 1x8 De-Multiplexer has single input, three selection lines and eight outputs.
- So, we require two **1x4 De-Multiplexers** in second stage in order to get the final eight outputs. Since, the number of inputs in second stage is two, we require **1x2 De Multiplexer** in first stage so that the outputs of first stage will be the inputs of second stage.
- Input of this 1x2 De-Multiplexer will be the overall input of 1x8 De-Multiplexer.

$$Y0 = D\ \overline{S2}\ \overline{S1}\ \overline{S0}$$

$$Y1 = D\ \overline{S2}\ \overline{S1}\ S0$$

$$Y2 = D\ \overline{S2}\ S1\ \overline{S0}$$

$$Y3 = D\ \overline{S2}\ S1\ S0$$

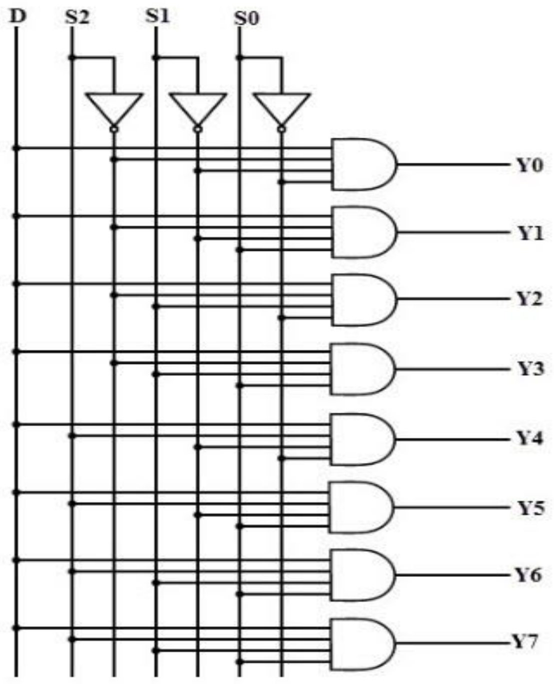
$$Y4 = D\ S2\ \overline{S1}\ \overline{S0}$$

$$Y5 = D\ S2\ \overline{S1}\ S0$$

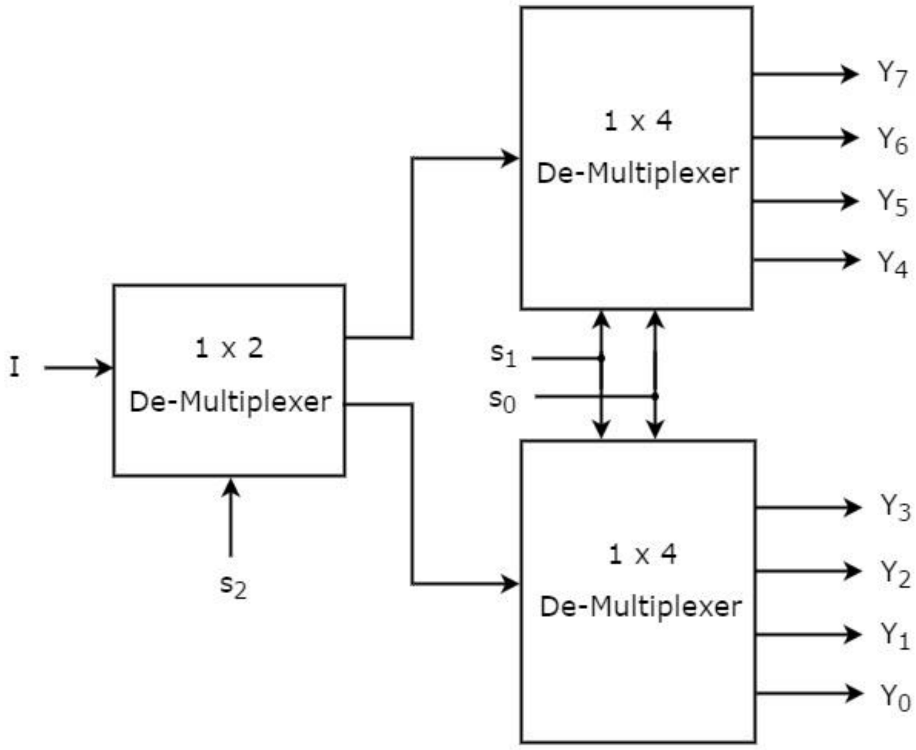
$$Y6 = D\ S2\ S1\ \overline{S0}$$

$$Y7 = D\ S2\ S1\ S0$$

(ii) Diagram of 4:1 Multiplexer



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

Select Inputs			Outputs							
S_2	S_1	S_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	D
0	0	1	0	0	0	0	0	0	D	0
0	1	0	0	0	0	0	0	D	0	0
0	1	1	0	0	0	0	D	0	0	0
1	0	0	0	0	0	D	0	0	0	0
1	0	1	0	0	D	0	0	0	0	0
1	1	0	0	D	0	0	0	0	0	0
1	1	1	D	0	0	0	0	0	0	0

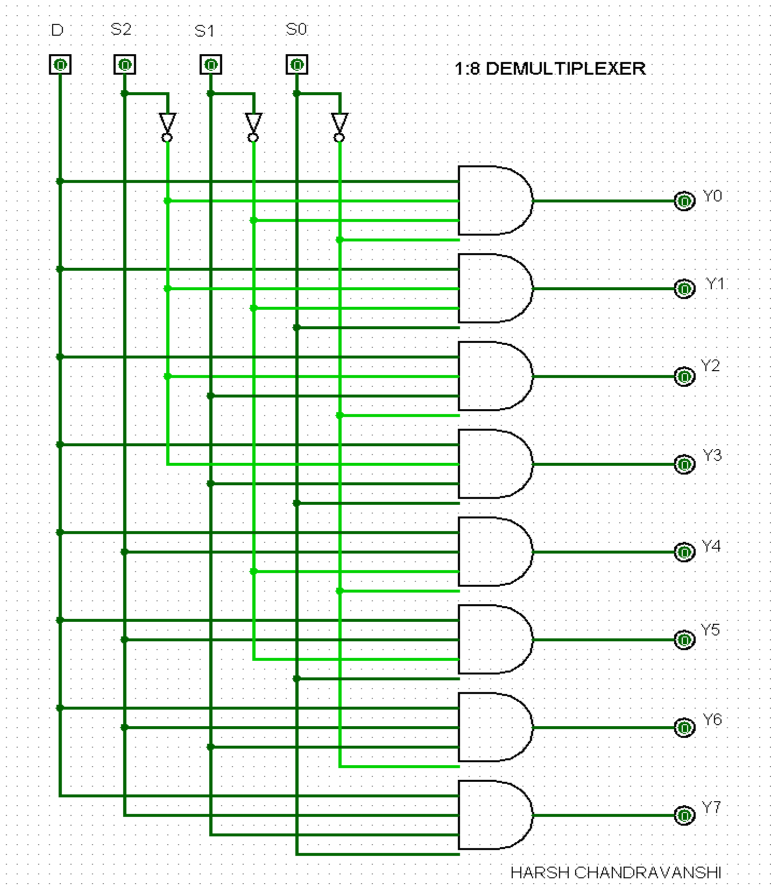


Fig-2 1:8 DEMUX using logisim

EXPERIMENT- 10

Aim - To implement 4:2 encoder & 8:3 encoder.

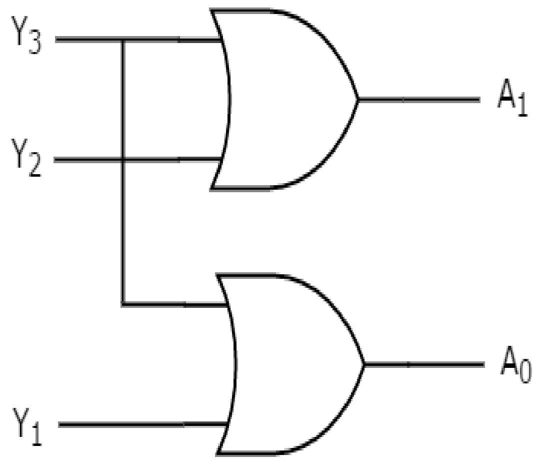
- An encoder is a combinational circuit that converts binary information in the form of a 2N input lines into N output lines, which represent N bit code for the input. For simple encoders, it is assumed that only one input line is active at a time.
- As an example, let's consider Octal to Binary encoder. Octal-to-binary encoder takes 8 input lines and generates 3 output lines.

(1) 4:2 Encoder

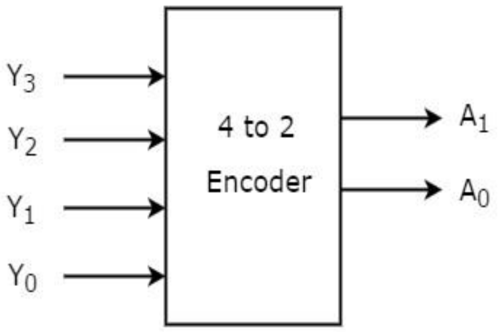
- 4 to 2 Encoder has four inputs Y3, Y2, Y1 & Y0 and two outputs A1 & A0.
- At any time, only one of these 4 inputs can be '1' in order to get the respective binary code at the output.
- We can implement the above 4:2 ENCODER by using two input OR gates.

A1=Y3+Y2
A0=Y3+Y1

(ii) Diagram of 4:2 ENCODER



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

Inputs				**Outputs**	
Y_3	**Y_2**	**Y_1**	**Y_0**	**A_1**	**A_0**
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

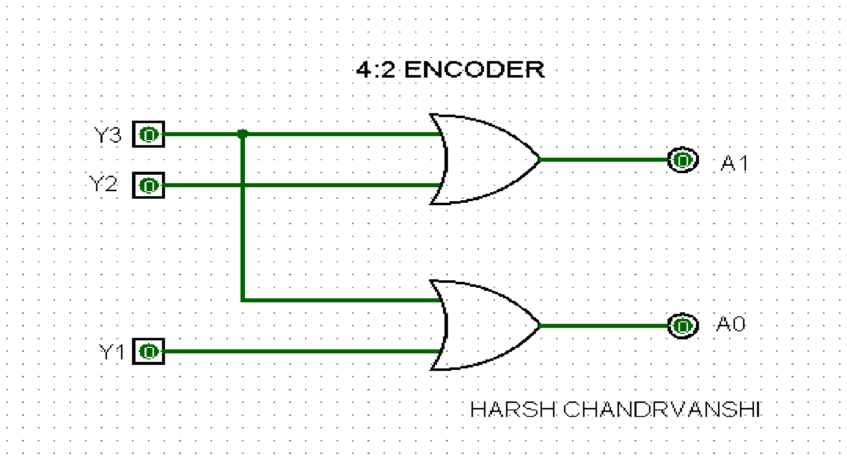


Fig-1 4:2 ENCODER using logisim

(2) 8:3 Encoder

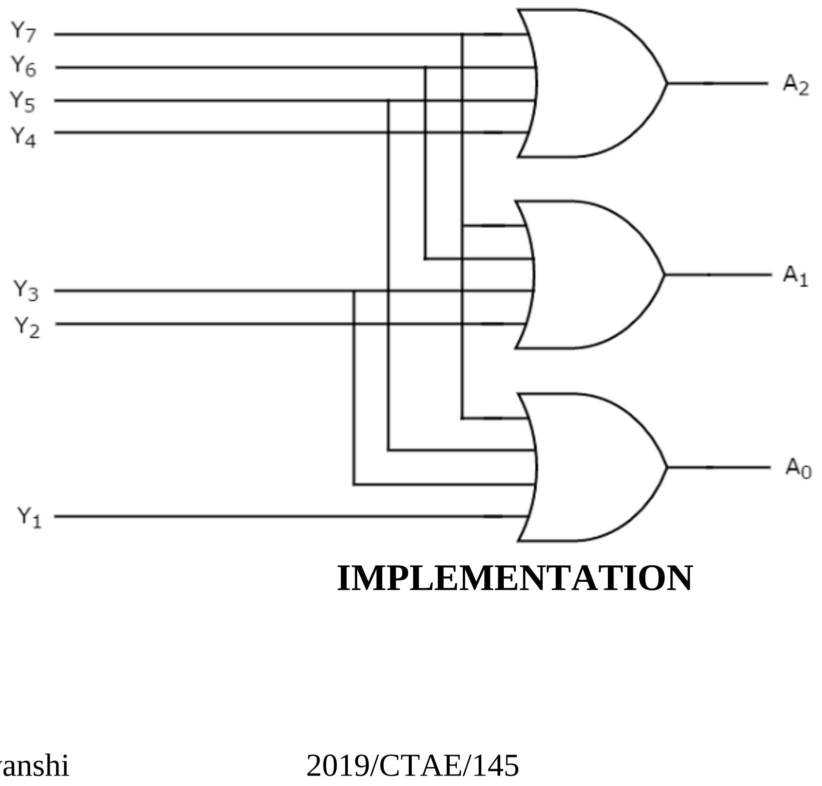
- Octal to binary Encoder has eight inputs, Y_7 to Y_0 and three outputs A_2, A_1 & A_0. Octal to binary encoder is nothing but 8 to 3 encoder.
- At any time, only one of these eight inputs can be '1' in order to get the respective binary code.
- One limitation of this encoder is that only one input can be active at any given time. If more than one inputs are active, then the output is undefined.
- Another ambiguity arises when all inputs are 0. In this case, encoder outputs 000 which actually is the output for D0 active. In order to avoid this, an extra bit can be added to the output, called the valid bit which is 0 when all inputs are 0 and 1 otherwise.
- When we consider its truth table, the output line Z is active when the input octal digit is 1, 3, 5 or 7. Similarly, Y is 1 when input octal digit is 2, 3, 6 or 7 and X is 1 for input octal digits 4, 5, 6 or 7. Hence, the Boolean functions would be:

$$\mathbf{A_2=Y_7+Y_6+Y_5+Y_4}$$

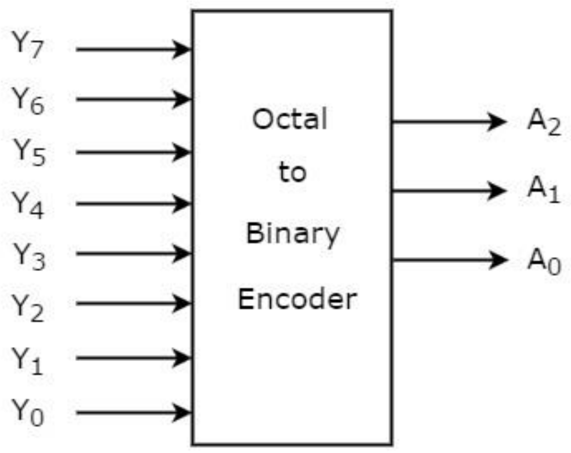
$$\mathbf{A_1=Y_7+Y_6+Y_3+Y_2}$$

$$\mathbf{A_0=Y_7+Y_5+Y_3+Y_1}$$

(ii) Diagram of 8:3 ENCODER



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

Inputs								**Outputs**		
Y_7	**Y_6**	**Y_5**	**Y_4**	**Y_3**	**Y_2**	**Y_1**	**Y_0**	**A_2**	**A_1**	**A_0**
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

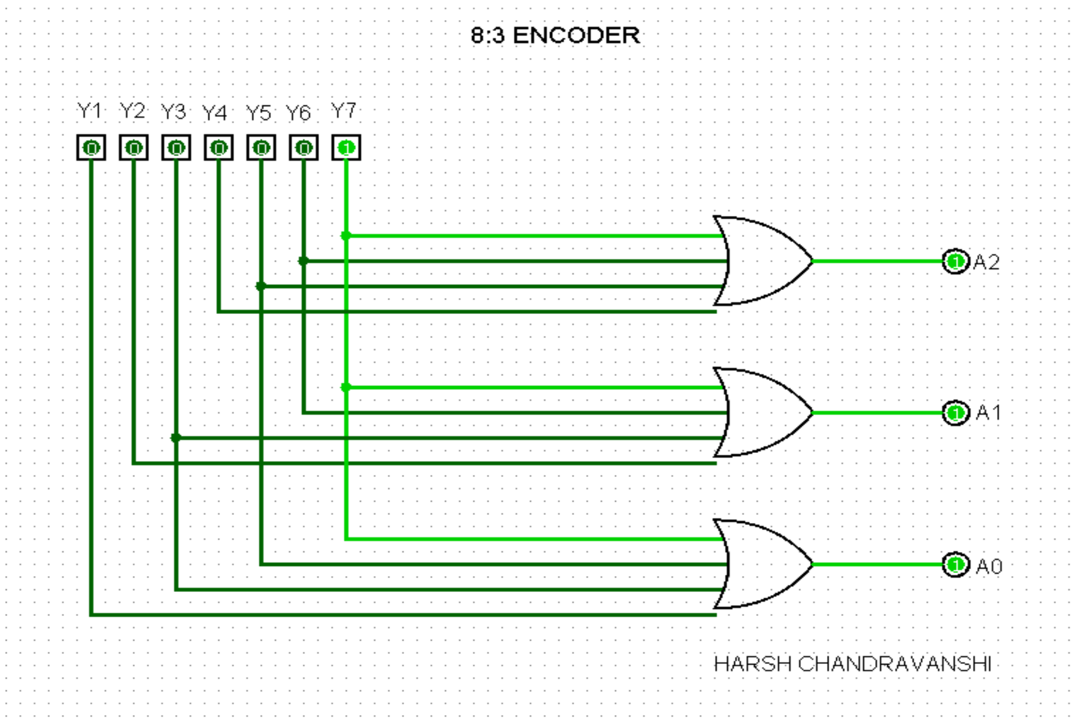


Fig-2 8:3 ENCODER using logisim

EXPERIMENT- 11

Aim - To implement 2:4 decoder & 3:8 decoder.

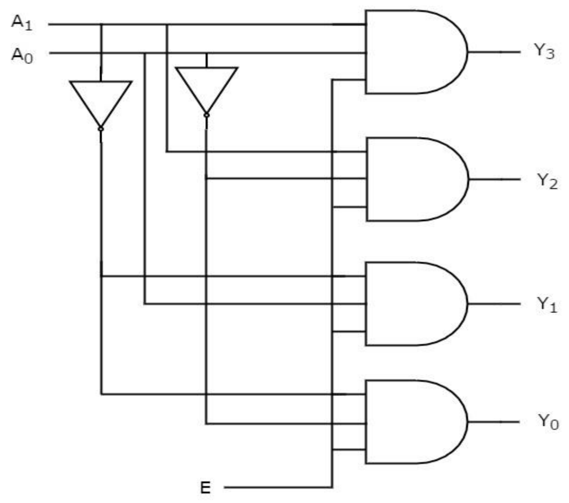
- Decoder is a combinational circuit that has ‘n’ input lines and maximum of 2n output lines.
- One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. That means decoder detects a particular code.
- The outputs of the decoder are nothing but the min terms of ‘n’ input variables lines, when it is enabled.

(1) 2:4 Decoder

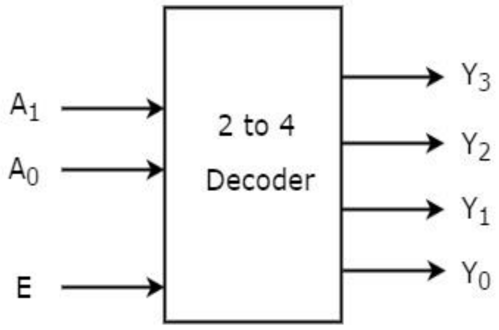
- 2 to 4 Decoder has two inputs A_1 & A_0 and four outputs Y_3, Y_2, Y_1 & Y_0.
- One of these four outputs will be ‘1’ for each combination of inputs when enable, E is ‘1’.
- We can implement the above 2:4 DECODER by using three input AND gates and two not gates.

$$Y_3=E.A_1.A_0$$
$$Y_2=E.A_1.A_0'$$
$$Y_1=E.A_1'.A_0$$
$$Y_0=E.A_1'.A_0'$$

(ii) Diagram of 2:4 DECODER



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

Enable	**Inputs**		**Outputs**			
E	**A_1**	**A_0**	**Y_3**	**Y_2**	**Y_1**	**Y_0**
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

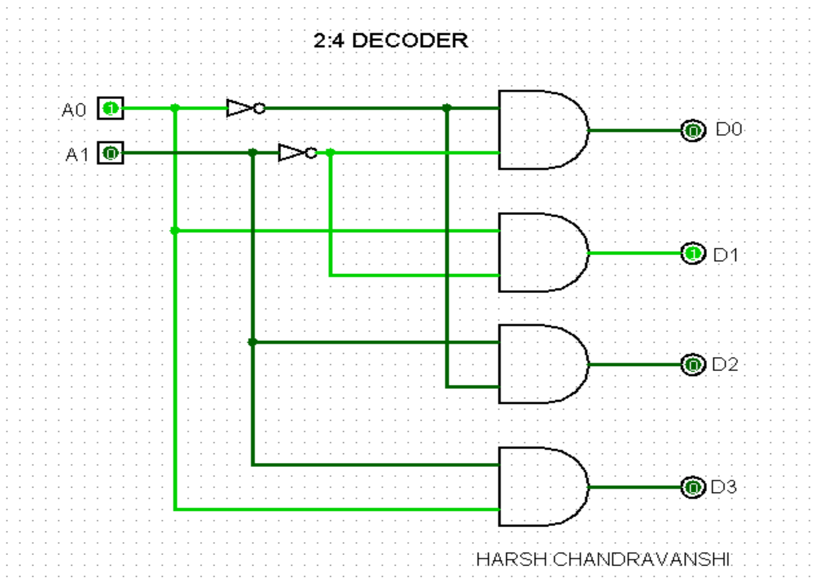


Fig-1 2:4 DECODER using logisim

(2) 3:8 Decoder

- To implement 3 to 8 decoder using 2 to 4 decoders. We know that 2 to 4 Decoder has two inputs, A1 & A0 and four outputs, Y3 to Y0. Whereas, 3 to 8 Decoder has three inputs A2, A1 & A0 and eight outputs, Y7 to Y0.
- We can find the number of lower order decoders required for implementing higher order decoder using the following formula.

Required number of lower order decoders=$m_2 m_1$

Where,

m_1 is the number of outputs of lower order decoder.

m_2 is the number of outputs of higher order decoder.

Y0=A0'.A1'.A2'

Y1=A0.A1'.A2'

Y2=A0'.A1.A2'

Y3=A0.A1.A2'

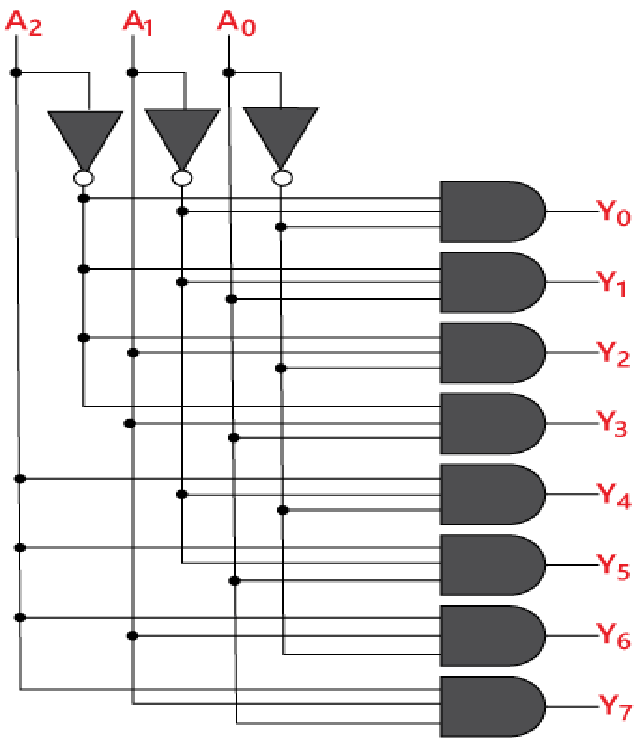
Y4=A0'.A1'.A2

Y5=A0.A1'.A2

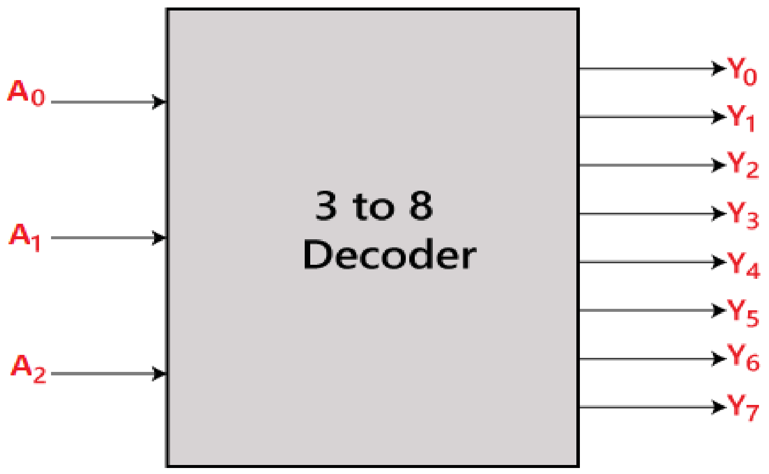
Y6=A0'.A1.A2

Y7=A0.A1.A2

(ii) Diagram of 3:8 DECODER



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

INPUTS			OUTPUTS							
A0	**A1**	**A2**	**Y0**	**Y1**	**Y2**	**Y3**	**Y4**	**Y5**	**Y6**	**Y7**
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

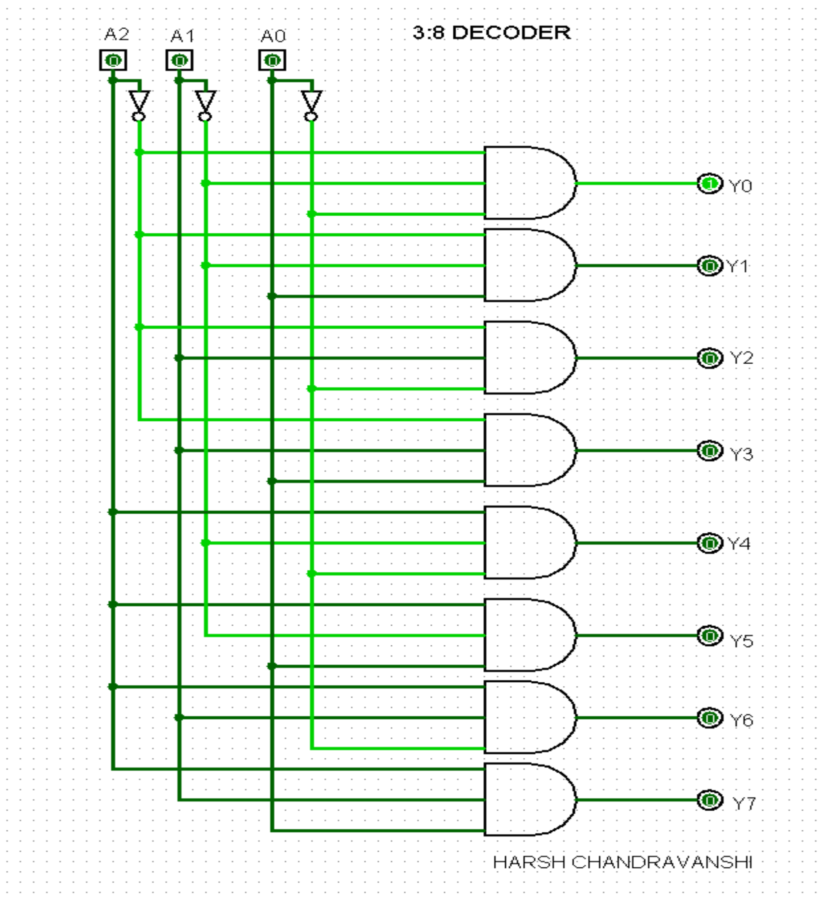


Fig-2 3:8 DECODER using logisim

EXPERIMENT- 12

Aim - To implement S-R flip Flop.

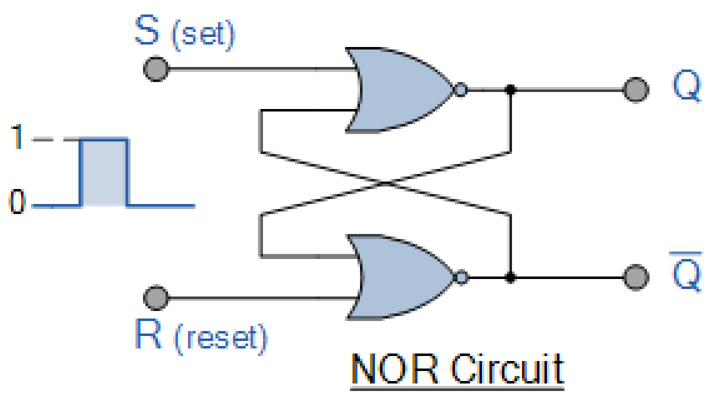
- A flip-flop is a device which stores a single *bit* (binary digit) of data; one of its two states represents a "one" and the other represents a "zero".
- Such data storage can be used for storage of *state*, and such a circuit is described as sequential logic in electronics.
- When used in a finite-state machine, the output and next state depend not only on its current input, but also on its current state (and hence, previous inputs).
- It can also be used for counting of pulses, and for synchronizing variably-timed input signals to some reference timing signal.

SR FLIP-FLOP:

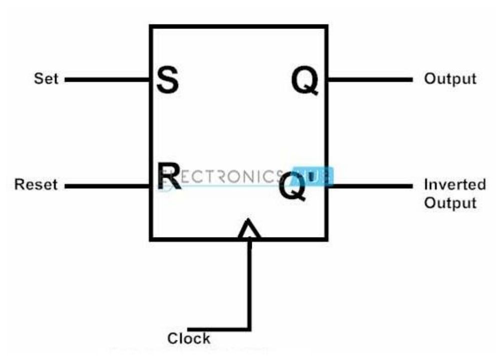
- The **SR flip-flop**, also known as a *SR Latch*, can be considered as one of the most basic sequential logic circuit possible. This simple flip-flop is basically a one-bit memory bistable device that has two inputs, one which will “SET” the device (meaning the output = “1”), and is labelled **S** and one which will “RESET” the device (meaning the output = “0”), labelled **R**.
- Then the SR description stands for “Set-Reset”. The reset input resets the flip-flop back to its original state with an output Q that will be either at a logic level “1” or logic “0” depending upon this set/reset condition.

$$Q_{(next)} = S + R'Q \qquad SR = 0$$

(ii) Diagram of SR FLIP FLOP



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

S	R	Q	STATE
0	0	PREVIOUS STATE	NO CHANGE
0	1	0	RESET
1	0	1	SET
1	1	?	FORBIDDEN

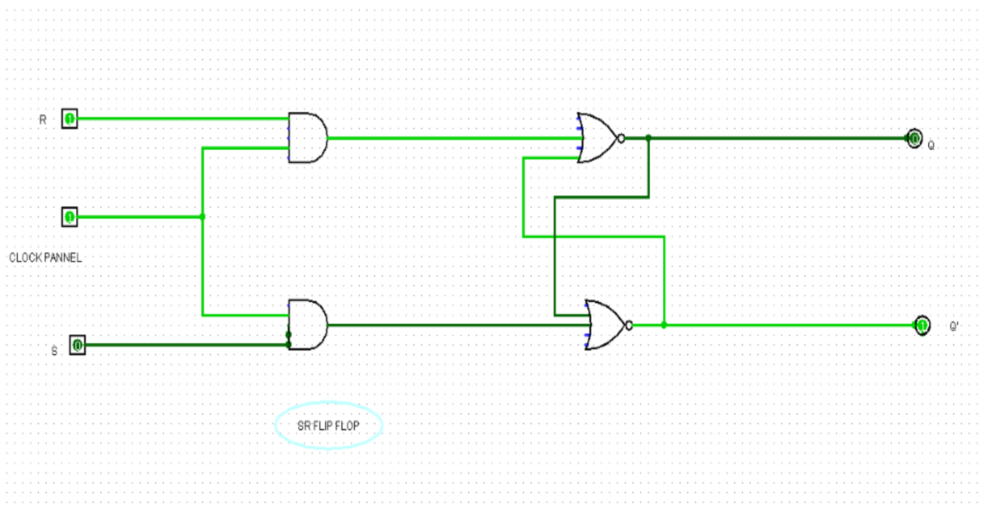


Fig-1 SR FLIP FLOP using Logisim

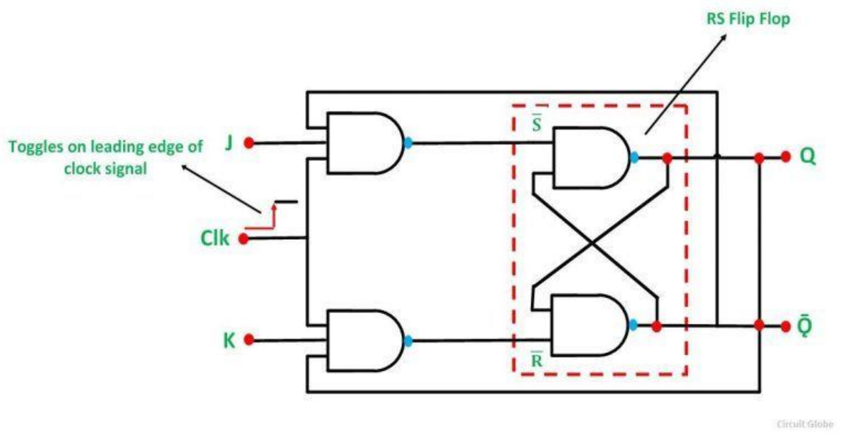
EXPERIMENT- 13

Aim - To implement J-K flip Flop.

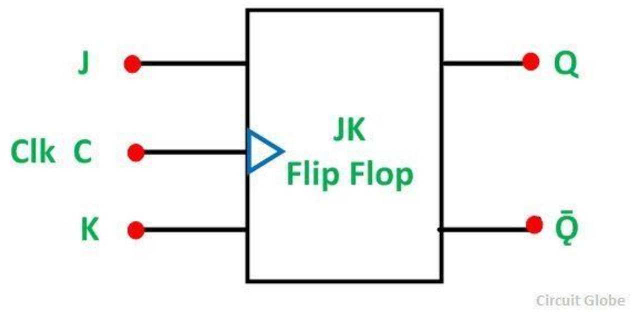
- The JK Flip Flop is the most widely used flip flop. It is considered to be a universal flip-flop circuit. The sequential operation of the JK Flip Flop is the same as for the RS flip-flop with the same SET and RESET input.
- The difference is that the JK Flip Flop does not the invalid input states of the RS Latch (when S and R are both 1). The JK Flip Flop name has been kept on the inventor name of the circuit known as Jack Kilby.
- The basic NAND gate RS flip-flop suffers from two main problems.
    - Firstly, the condition when S = 0 and R = 0 should be avoided.
    - Secondly, if the state of S or R changes its state while the input which is enabled is high, the correct latching action does not occur.
    - Thus, to overcome these two problems of the RS Flip-Flop, the JK Flip Flop was designed.
- The JK Flip Flop is basically a gated RS flip flop with the addition of the clock input circuitry. When both the inputs S and R are equal to logic “1”, the invalid condition takes place.
- Thus, to prevent this invalid condition, a clock circuit is introduced. The JK Flip Flop has four possible input combinations because of the addition of the clocked input. The four inputs are “logic 1”, ‘logic 0”. “No change’ and “Toggle”.

$$Q_{(next)} = JQ' + K'Q$$

(iv) Diagram of JK FLIP FLOP



IMPLEMENTATION



BLOCK DIAGRAM

(v) Truth Table-

	J	**K**	**Q**	**Q̄**	**DESCRIPTION**
Same as for the RS Latch	0	0	0	0	Memory No Change
	0	0	0	1	
	0	1	1	0	Reset Q >> 0
	0	1	0	1	
	1	0	0	1	Set Q >> 1
	1	0	1	0	
Toggle	1	1	0	1	Toggle
	1	1	1	0	

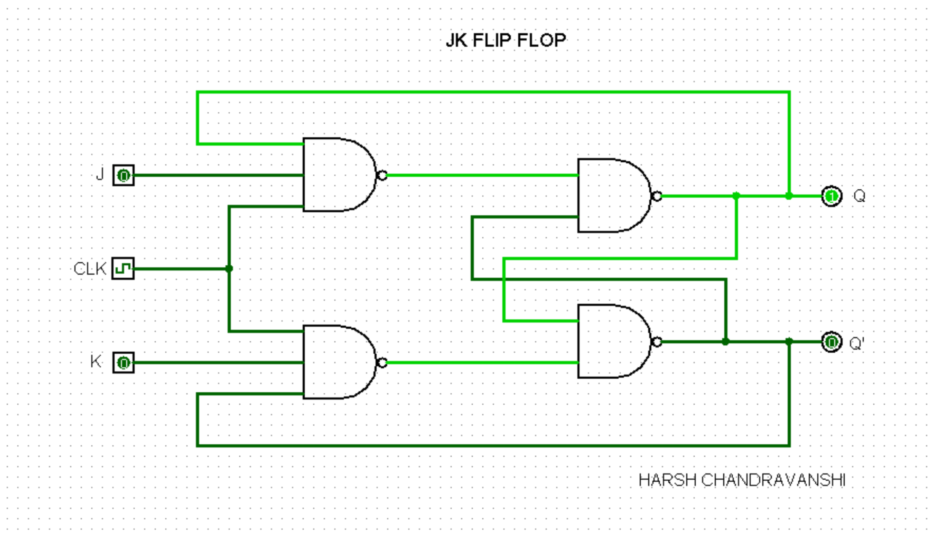


Fig-1 JK FLIP FLOP using logisim

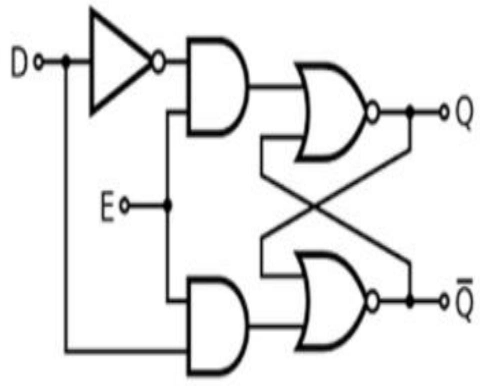
EXPERIMENT- 14

Aim - To implement D flip Flop.

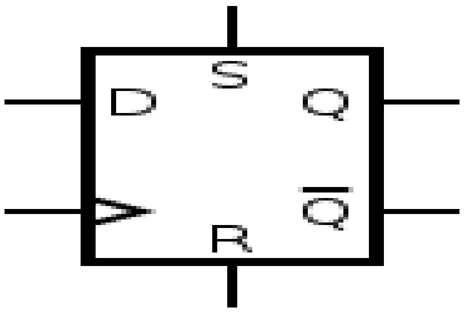
- The D flip-flop is widely used. It is also known as a "data" or "delay" flip-flop.
- The D flip-flop captures the value of the D-input at a definite portion of the clock cycle (such as the rising edge of the clock). That captured value becomes the Q output. At other times, the output Q does not change. The D flip-flop can be viewed as a memory cell, a zero-order hold, or a delay line.
- The Q output always takes on the state of the D input at the moment of a rising clock edge. (or falling edge if the clock input is active low) It is called the D flip-flop for this reason, since the output takes the value of the D input or Data input, and Delays it by one clock count.

Q(next) = D

(ii) Diagram of D FLIP FLOP



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

D	**S**	**R**	**Q**	**STATE**
	0	0	PREVIOUS STATE	NO CHANGE
0	0	1	0	RESET
1	1	0	1	SET
	1	1	?	FORBIDDEN

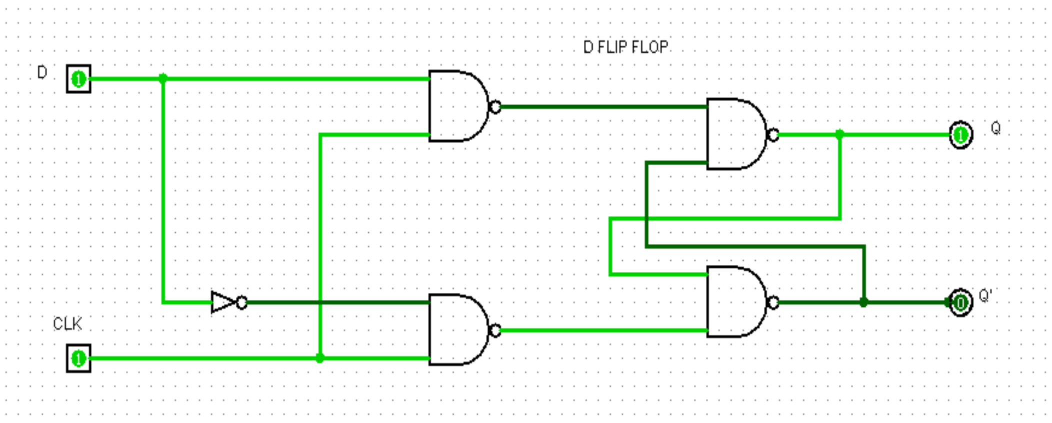


Fig-1 D FLIP FLOP using logisim

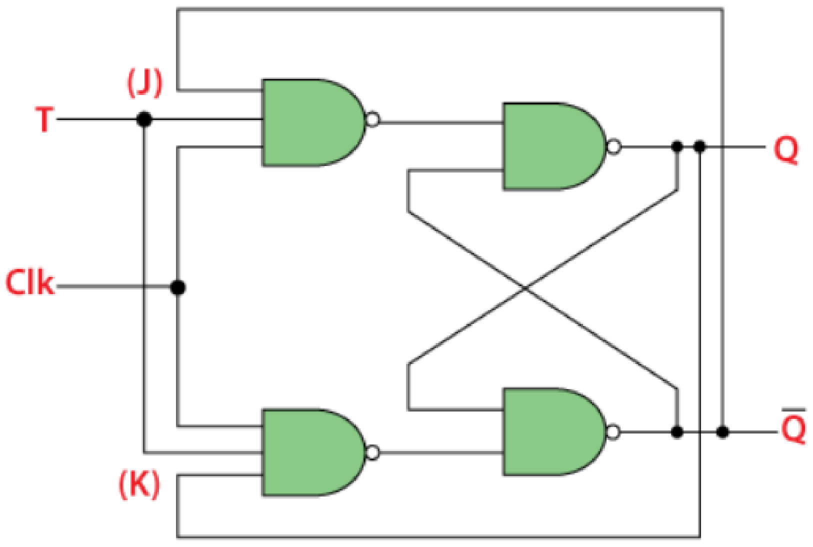
EXPERIMENT- 15

Aim - To implement D flip Flop.

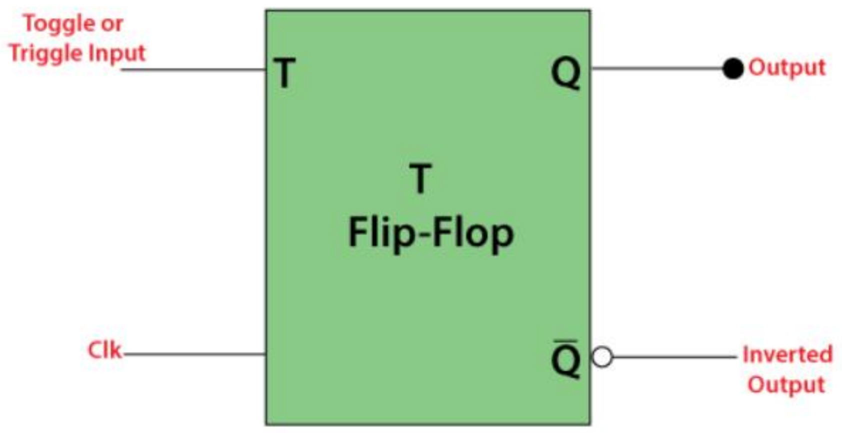
- In T flip flop, "T" defines the term "Toggle". In SR Flip Flop, we provide only a single input called "Toggle" or "Trigger" input to avoid an intermediate state occurrence. Now, this flip-flop works as a Toggle switch. The next output state is changed with the complement of the present state output. This process is known as "Toggling"'.
- The next state of the T flip flop is similar to the current state when the T input is set to false or 0.
- If toggle input is set to 0 and the present state is also 0, the next state will be 0.
- If toggle input is set to 0 and the present state is 1, the next state will be 1.
- The next state of the flip flop is opposite to the current state when the toggle input is set to 1.
- If toggle input is set to 1 and the present state is 0, the next state will be 1.
- If toggle input is set to 1 and the present state is 1, the next state will be 0.
- In "T Flip Flop", the state at an applied trigger pulse is defined only when the previous state is defined. It is the main drawback of the "T Flip Flop".

$$Q' = T \oplus Q$$

(ii) Diagram of T FLIP FLOP



IMPLEMENTATION



BLOCK DIAGRAM

(iii) Truth Table-

	PREVIOUS		**NEXT**	
T	Q	Q’	Q	Q’
0	0	1	0	1
0	1	0	1	0
1	0	1	1	0
1	1	0	0	1

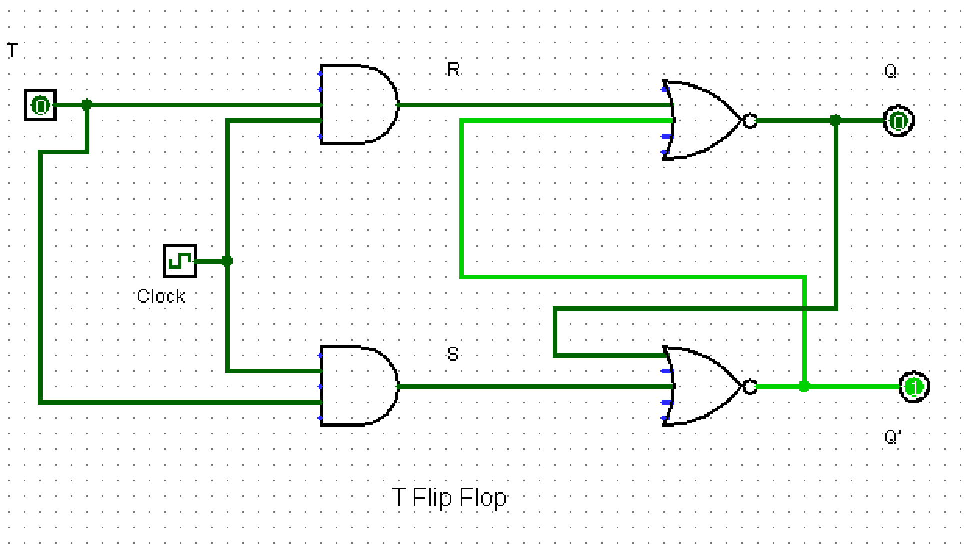


Fig-1 T FLIP FLOP using logisim