Practical 1

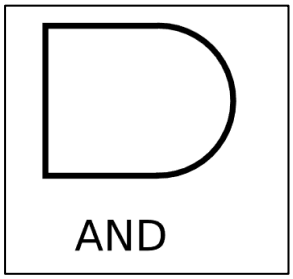
AIM: A) Demonstrate the working of basic logic gates and verify with the truth table.

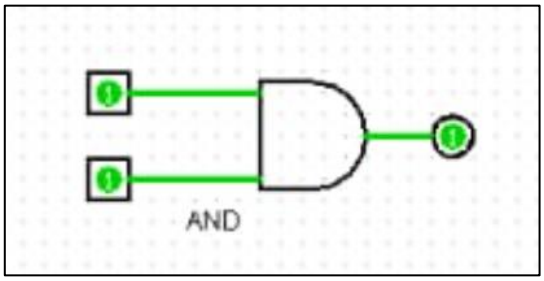
B) Demonstrate the working of basic logic gates using NAND and NOR Gates.

A) Study of Basic Logic Gates and Verification of Their Truth Tables

1. AND Gate

- **Definition**: The AND gate outputs true (1) only if all its inputs are true (1).
- **Symbol and Implementation**:



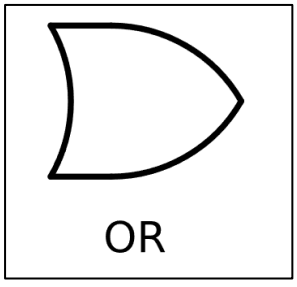


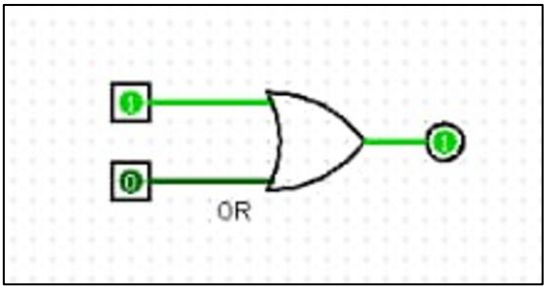
- **Truth Table**:

A	B	Q (A AND B)
0	0	0
0	1	0
1	0	0
1	1	1

2. OR Gate

- **Definition**: The OR gate outputs true (1) if at least one of its inputs is true (1).
- **Symbol and Implementation**:





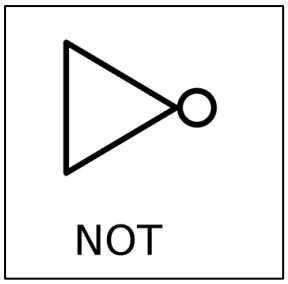
- **Truth Table**:

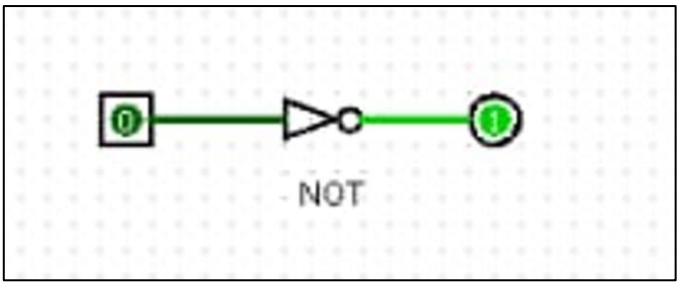
A	B	Q (A OR B)
0	0	0

0	1	1
1	0	1
1	1	1

3. NOT Gate (Inverter)

- **Definition**: The NOT gate outputs the opposite of its input. If the input is true (1), the output is false (0), and vice versa.
- **Symbol and Implementation**:



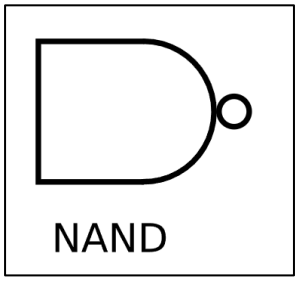


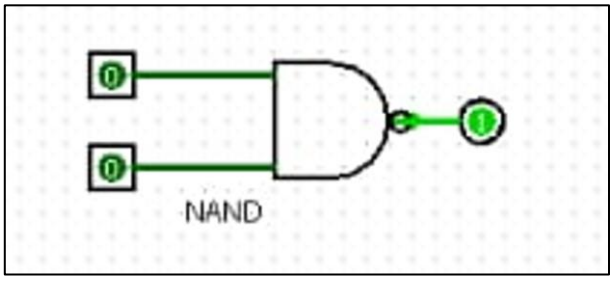
- **Truth Table**:

A	Q (NOT A)
0	1
1	0

4. NAND Gate

- **Definition**: The NAND gate is the inverse of the AND gate. It outputs false (0) only if all its inputs are true (1).
- **Symbol and Implementation**:





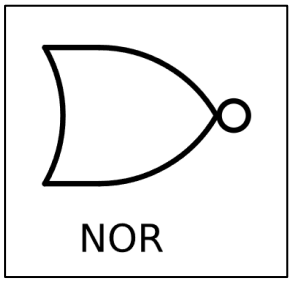
- **Truth Table**:

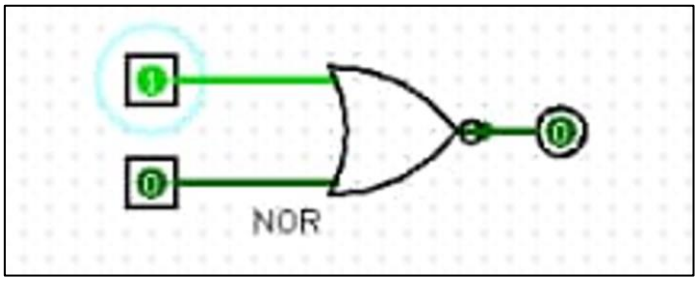
A	B	Q (A NAND B)
0	0	1
0	1	1

1	0	1
1	1	0

5. NOR Gate

- **Definition**: The NOR gate is the inverse of the OR gate. It outputs true (1) only if all its inputs are false (0).
- **Symbol and Implementation**:



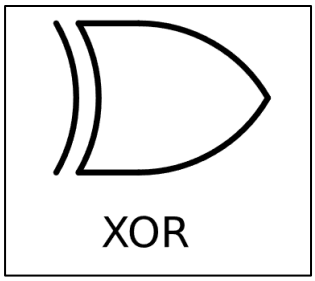


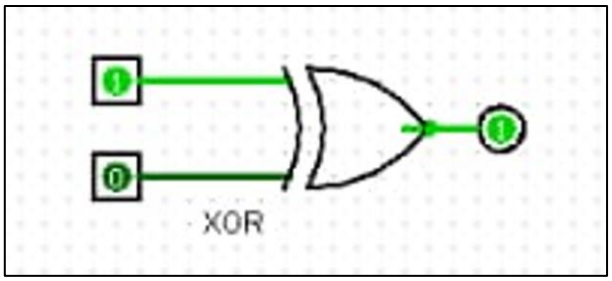
- **Truth Table**:

A	B	Q (A NOR B)
0	0	1
0	1	0
1	0	0
1	1	0

6. XOR Gate (Exclusive OR)

- **Definition**: The XOR gate outputs true (1) if an odd number of its inputs are true (1).
- **Symbol and Implementation**:





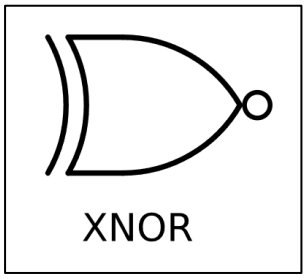
- **Truth Table**:

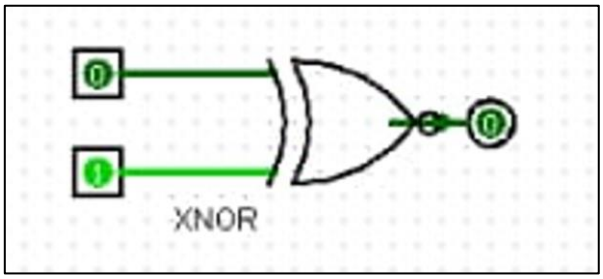
A	B	Q (A XOR B)
0	0	0
0	1	1

1	0	1
1	1	0

7. XNOR Gate (Exclusive NOR)

- **Definition**: The XNOR gate is the inverse of the XOR gate. It outputs true (1) if an even number of its inputs are true (1).
- **Symbol and Implementation**:





- **Truth Table**:

A	B	Q (A XNOR B)
0	0	1
0	1	0
1	0	0
1	1	1

Implementation of Boolean Expressions Using Basic Gates

1. AND Gate: Q = A · B (The output will be true only if both A and B are true.)

2. OR Gate: Q = A + B (The output will be true if either A or B is true.)

3. NOT Gate: $Q = \overline{A}$ (The output will be the inverse of A.)

4. NAND Gate: $Q = \overline{A \cdot B}$ (The output will be the inverse of the AND gate output.)

5. NOR Gate: $Q = \overline{A + B}$ (The output will be the inverse of the OR gate output.)

6. XOR Gate: $Q = A \oplus B$ (The output will be true if either A or B is true, but not both.)

7. XNOR Gate: $Q = \overline{A \oplus B}$ (The output will be true if both A and B are either true or false.)

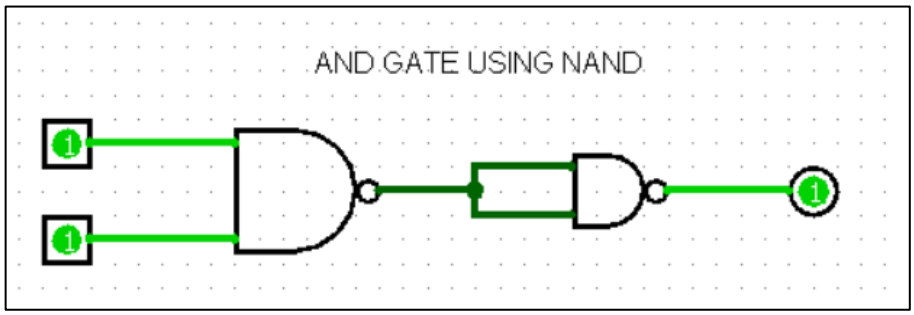
B) Working of basic logic gates using NAND and NOR Gates.

1. AND Gate Using NAND

- To construct an AND gate using only NAND gates, we can take advantage of De Morgan's laws and the fact that NAND gates are universal gates.
- The circuit involves two steps:
    - The first NAND gate receives the two inputs, A and B, and produces the output AB'.
    - The output of the first NAND gate is then fed into another NAND gate configured as an inverter (by tying its inputs together). This results in the final output, which is ABABAB, replicating the AND operation.

Truth Table:

A	B	Output (AB)
0	0	0
0	1	0
1	0	0
1	1	1



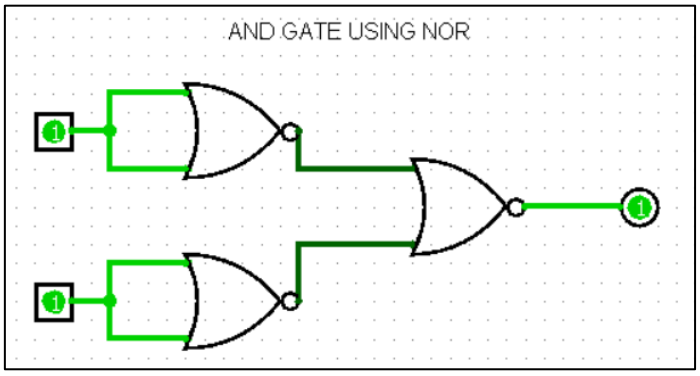
---

2. AND Gate Using NOR

- To construct an AND gate using only NOR gates, we use the fact that NOR gates can be arranged to simulate other logic gates, including AND gates.
- The circuit involves the following steps:
    - Each input A and B is first passed through a NOR gate with itself, effectively creating NOT gates. The outputs are A' and B'.
    - The outputs A' and B' are then fed into a third NOR gate. The output of this gate is (A'+B')', which, according to De Morgan's theorem, simplifies to AB, simulating the AND operation.

Truth Table:

A	B	Output (AB)
0	0	0
0	1	0
1	0	0
1	1	1



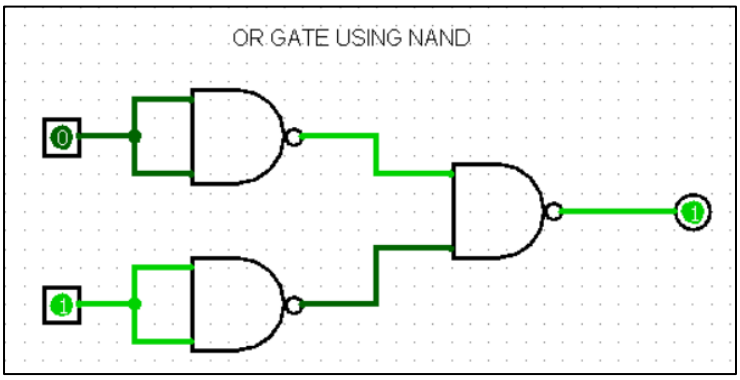
---

3. OR Gate Using NAND

- An OR gate can be implemented using NAND gates by first inverting each input and then NAND the results.
- The steps involved are:
    - Each input A and B is passed through a NAND gate with itself, effectively creating NOT gates. The outputs are A' and B'.
    - These outputs are then fed into a third NAND gate. The output is (A'·B')', which simplifies to A+B, replicating the OR operation.

Truth Table:

A	B	Output (A+B)
0	0	0
0	1	1
1	0	1
1	1	1



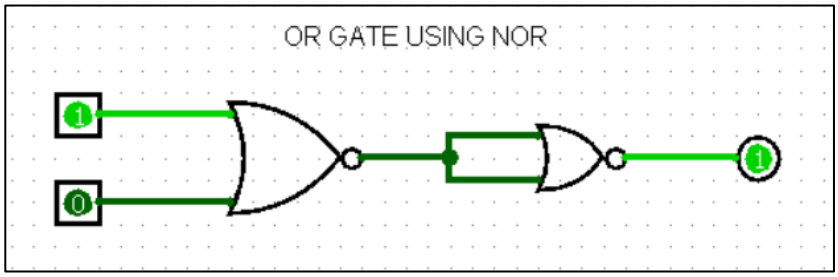
---

4. OR Gate Using NOR

- An OR gate can also be constructed using only NOR gates by using the following method:
    - The two inputs A and B are first NORed together, producing the output (A+B)'.
    - This output is then inverted by feeding it into another NOR gate where both inputs are the same, resulting in the final output A+B, which is the desired OR operation.

Truth Table:

A	B	Output (A+B)
0	0	0
0	1	1
1	0	1
1	1	1

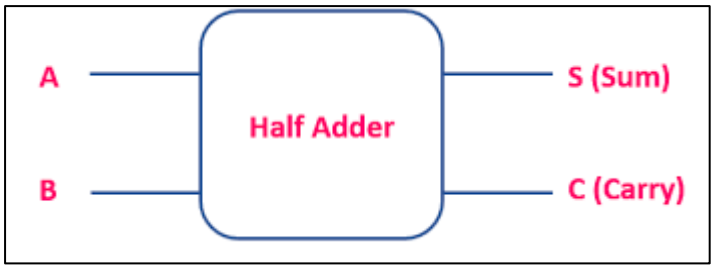


Practical 2

AIM: Design and implementation of half adder, full adder, half subtractor and full subtractor circuits using Logisim and Verilog HDL.

1. Half Adder

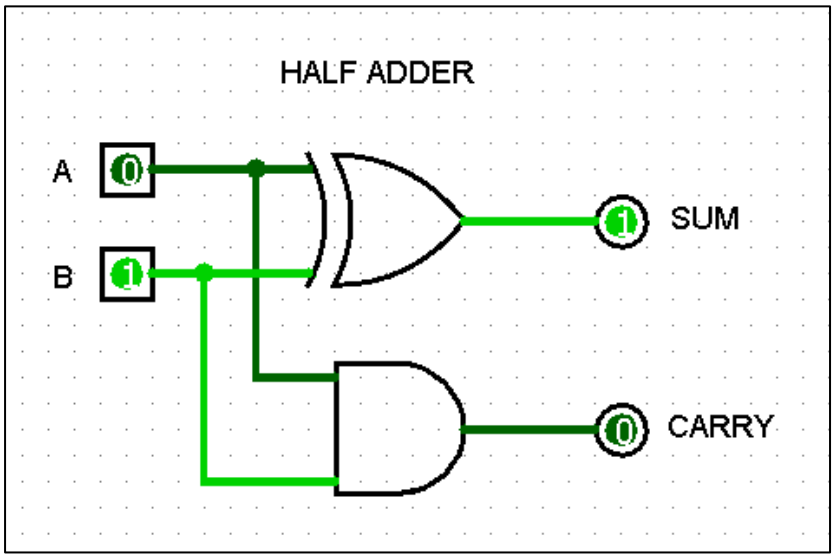
- **Definition**: A half adder is a fundamental digital circuit used in computing to add two single-bit binary numbers. It has two inputs, typically labelled A and B, and two outputs: the sum (S) and the carry (C).
- **Representation**:



- **Truth Table**:

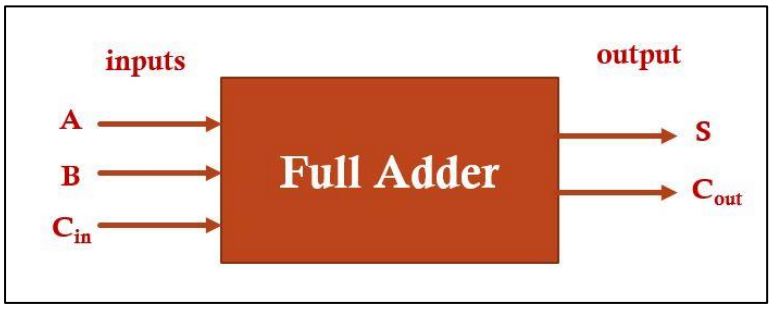
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- **Boolean Expression**: **S = A'B + AB', C = AB**
- **Implementation in Logisim**:



2. Full Adder

- **Definition**: A full adder is an advanced digital circuit used in computing to add three single-bit binary numbers: two significant bits and a carry-in bit. It has three inputs and two outputs: the sum and the carry-out.
- **Representation**:



- **Truth Table**:

A	B	Cin	S	Cout
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

- **Boolean Expression**: **S = A'B + AB', C = AB**
- **Simplification Process of Expressions:**

$$S = \overline{A}\,\overline{B}\,C_{in} + \overline{A}\,B\,\overline{C_{in}} + A\,\overline{B}\,\overline{C_{in}} + A\,B\,C_{in}$$

$$= \overline{A}\,(\overline{B}\,C_{in} + B\,\overline{C_{in}}) + A\,(\overline{B}\,\overline{C_{in}} + B\,C_{in})$$

$$= \overline{A}\,(B \oplus C_{in}) + A\,\overline{(B \oplus C_{in})}$$

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = \overline{A}\,B\,C_{in} + A\,\overline{B}\,C_{in} + A\,B\,\overline{C_{in}} + A\,B\,C_{in}$$

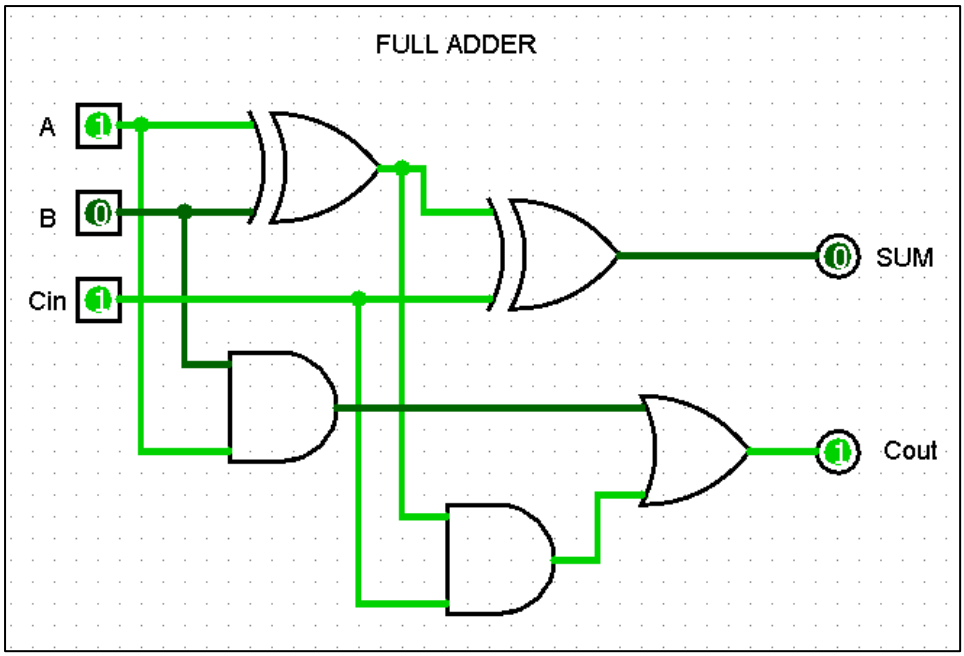
$$= AB(C_{in} + \overline{C_{in}}) + C_{in}(\overline{A}\,B + A\,\overline{B})$$

$$= AB + C_{in}(A \oplus B)$$

$$C_{out} = AB + BC_{in} + AC_{in}$$

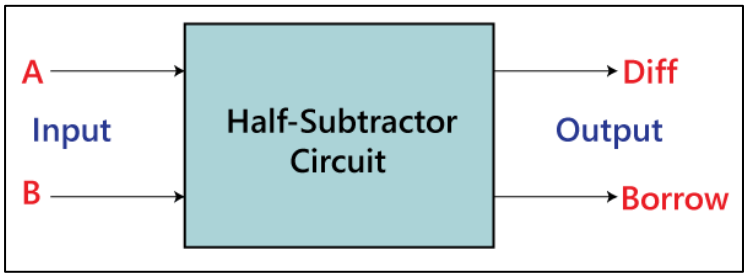
$$S = A \oplus B \oplus C_{in}$$

- **Implementation in Logisim**:



3. Half Subtractor

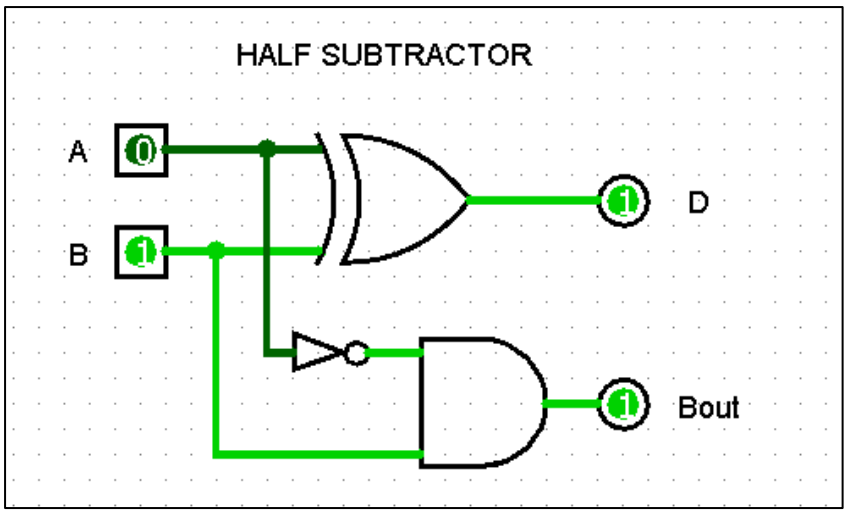
- **Definition**: A half subtractor is a digital circuit used to subtract two single-bit binary numbers. It has two inputs, typically labelled A (minuend) and B (subtrahend), and two outputs: the difference (D) and the borrow (Bout).
- **Representation**:



- **Truth Table**:

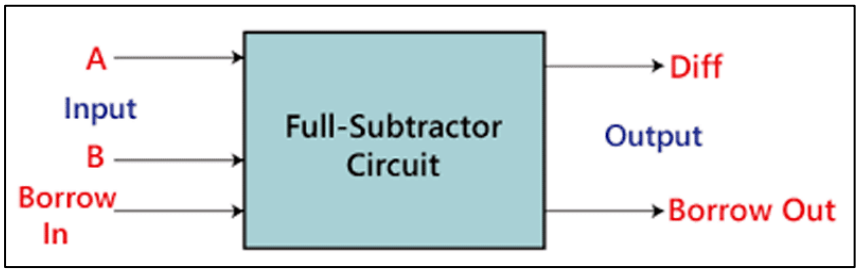
A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

- **Boolean Expression**: **Difference (D) = $A \oplus B$ (XOR gate), Borrow (Bout) = $A'.\ B$ (AND gate)**
- **Implementation in Logisim**:



4. Full Subtractor

- **Definition**: A full subtractor is a digital circuit used to subtract three binary bits: two significant bits and a borrow-in bit. It has three inputs - A (minuend), B (subtrahend), and Bin - and two outputs: the difference (D) and the borrow-out (Bout).
- **Representation**:



- **Truth Table**:

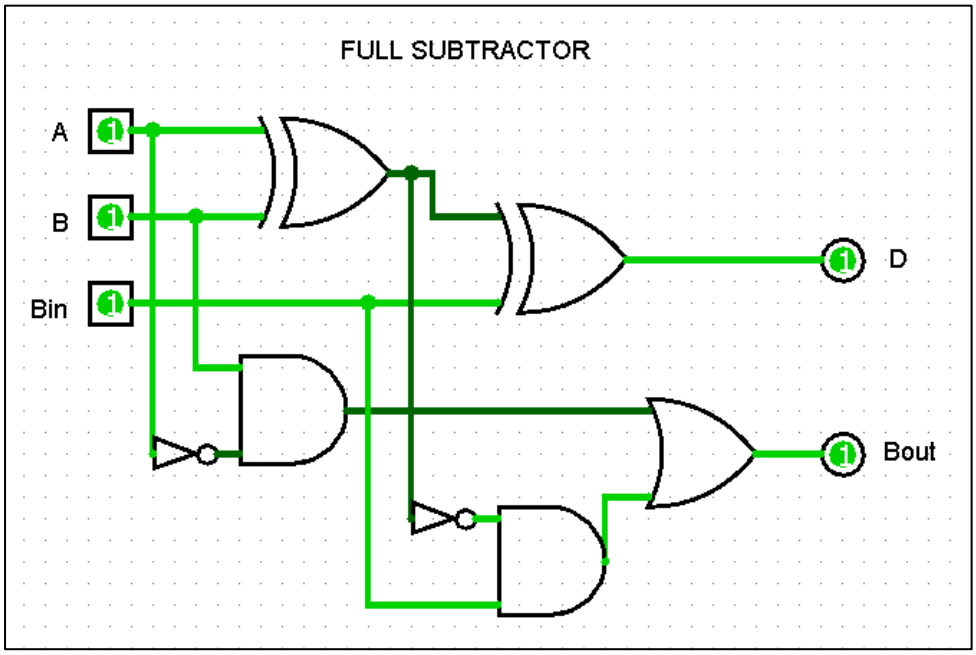
A	B	Cin	D	Cout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

- **Boolean Expression**: **Difference (D) = A ⊕ B ⊕ Bin, Borrow (Bout) = Bin (A XOR B)' + A'B**
- **Simplification Process of Expressions:**

```
D   = A'B'Bin + A'BBin' + AB'Bin' + ABBin
    = Bin(A'B' + AB)  + Bin'(AB' + A'B)
    = Bin( A XNOR B) + Bin'(A XOR B)
    = Bin (A XOR B)'  +  Bin'(A XOR B)
    = Bin XOR (A XOR B)
    = (A XOR B) XOR Bin
```

```
Bout = A'B'Bin + A'BBin' + A'BBin + ABBin
     = Bin(AB + A'B') + A'B(Bin + Bin')
     = Bin( A XNOR B) + A'B
     = Bin (A XOR B)' + A'B
```

- **Implementation in Logisim**:



Practical 3

AIM: Using in Verilog HDL

A. Design and implementation of binary to Gray and vice-versa code converter.
B. Design and implementation of a combinational circuit that converts the given BCD number into an equivalent 84-2-1 number.

A. Design and implementation of binary to Gray and vice-versa code converter.

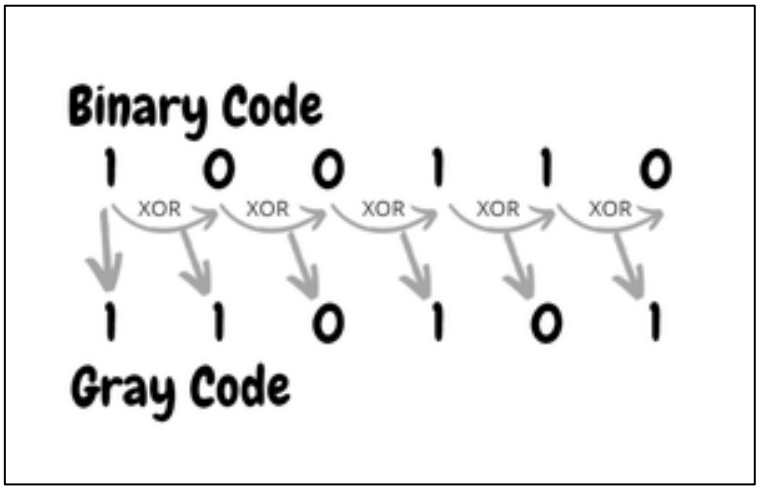
Gray Code:

- A binary numbering system in which two successive values only differ by one bit is called Gray code, often referred to as reflected binary code or unit distance code.
- In the typical sequence of binary numbers produced by the hardware that could provide an error or ambiguity during the change from one number to the next, Gray codes are highly helpful.

i) Binary to Gray Conversion

How to Convert Binary to Gray Code:

1. In the Gray code, the MSB will always be the same as the 1'st bit of the given binary number.
2. In order to perform the 2nd bit of the gray code, we perform the exclusive-or (XOR) of the 1'st and 2nd bit of the binary number. It means that if both the bits are different, the result will be one else the result will be 0.

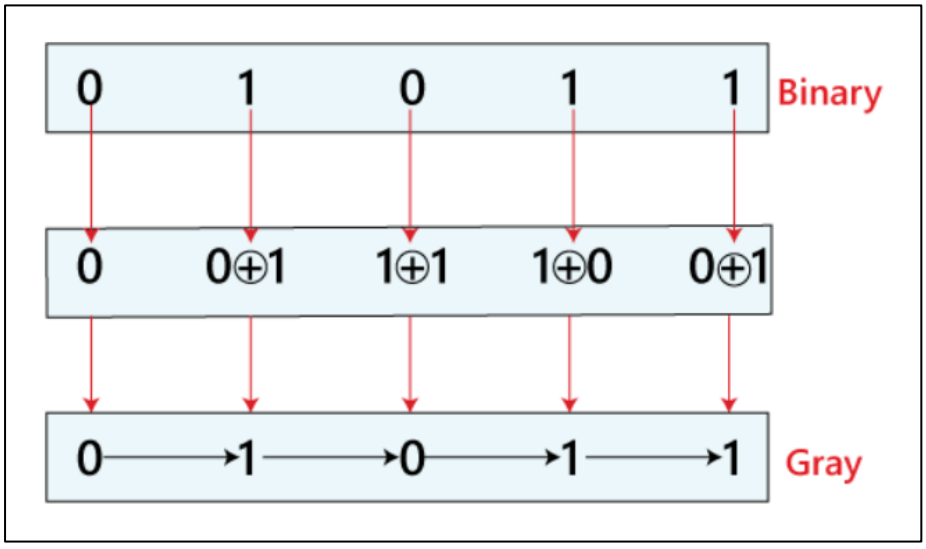


3. In order to get the 3rd bit of the gray code, we need to perform the exclusive-or (XOR) of the 2nd and 3rd bit of the binary number. The process remains the same for the 4th bit of the Gray code. Let's take an example to understand these steps.

The 4-bit binary to gray code conversion table is as follows:

Decimal Number	4-bit Binary Code	4-bit Gray Code
	B3B2B1B0	**G3G2G1G0**
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Example: The gray code of the binary number 01101 is 01011.



Boolean Expressions:

G3 = B3

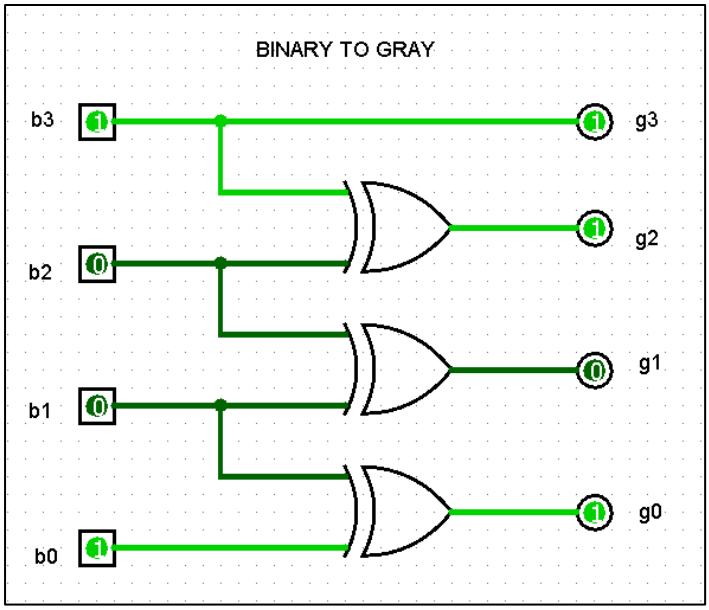
G2 = B3 $\oplus$ B2

$G1 = B2 \oplus B1$

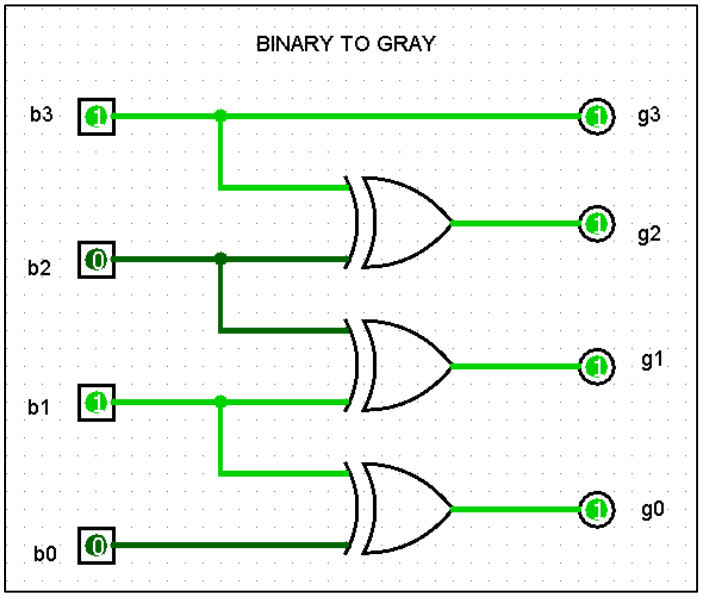
$G0 = B1 \oplus B0$

Digital Circuit Implementation in Logisim:

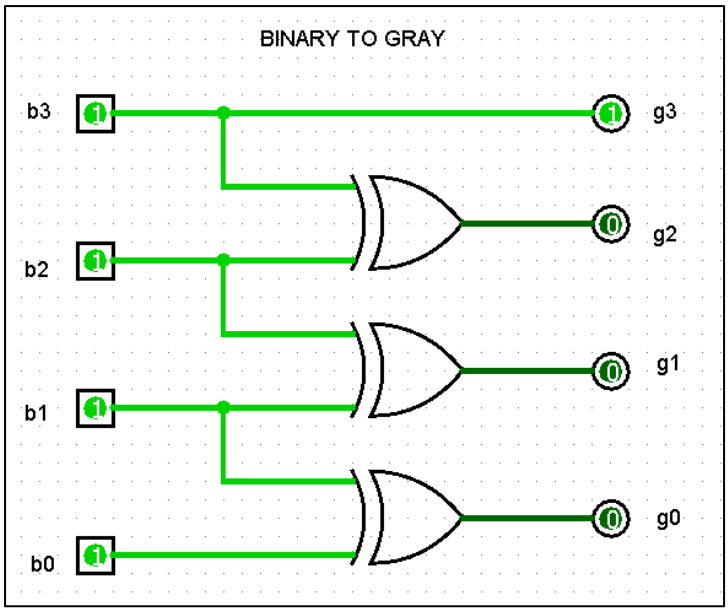
Question 1: 1001 -> 1101



Question 2: 1010 -> 1111



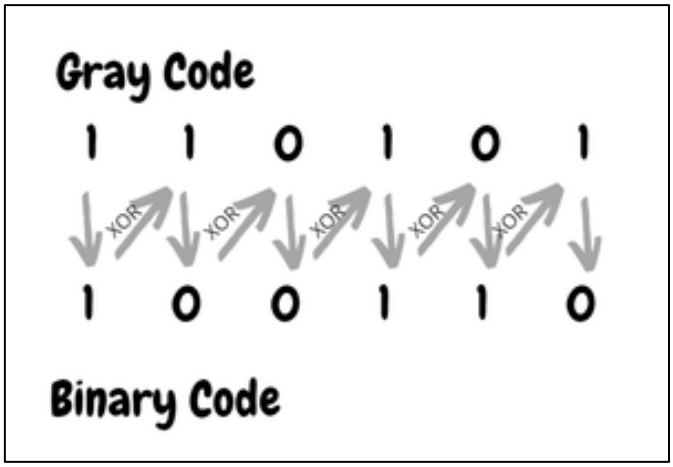
Question 3: 1111 -> 1000



ii) Gray to Binary Conversion

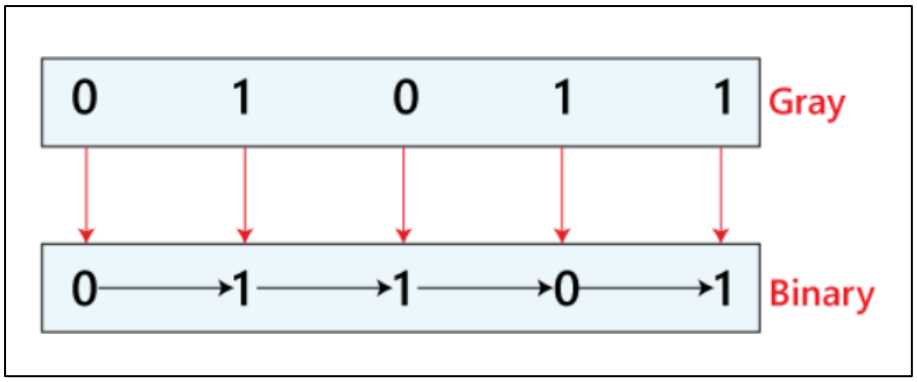
How to Convert Gray to Binary Code:

1. Just like binary to gray, in gray to binary, the 1st bit of the binary number is similar to the MSB of the Gray code.



2. The 2nd bit of the binary number is the same as the 1st bit of the binary number when the 2nd bit of the Gray code is 0; otherwise, the 2nd bit is altered bit of the 1st bit of binary number. It means if the 1st bit of the binary is 1, then the 2nd bit is 0, and if it is 0, then the 2nd bit be 1.
3. The 2nd step continues for all the bits of the binary number.

Example: The binary number of the gray code 01011 is 01101.



Boolean Expressions:

B3 = G3

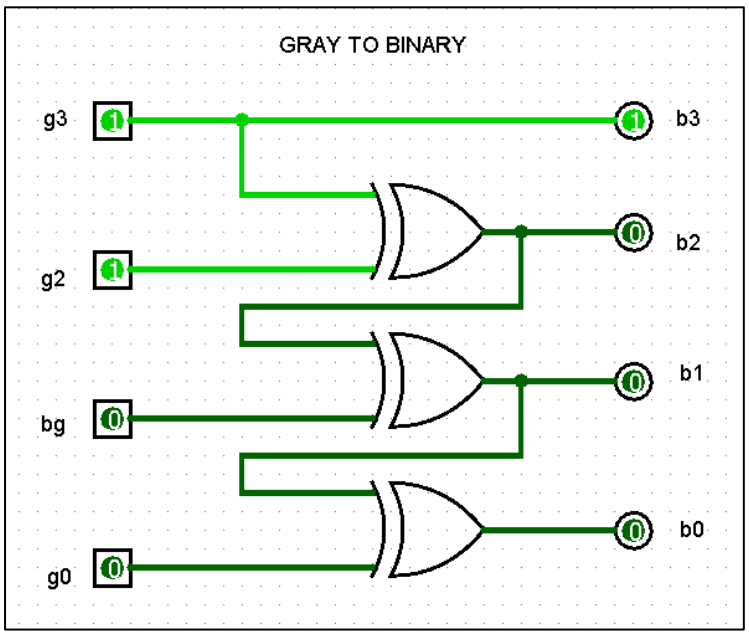
B2 = B3 $\oplus$ G2

$B1 = B2 \oplus G1$

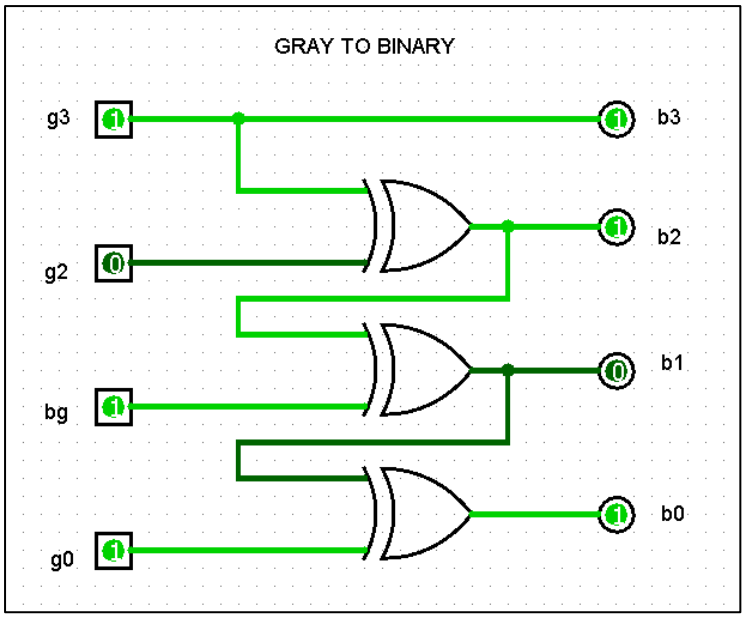
$B0 = B1 \oplus G0$

Digital Circuit Implementation in Logisim

Question 1: 1100 -> 1000



Question 2: 1011 -> 1101



Question 3: 1000 -> 1111