Experiment 1

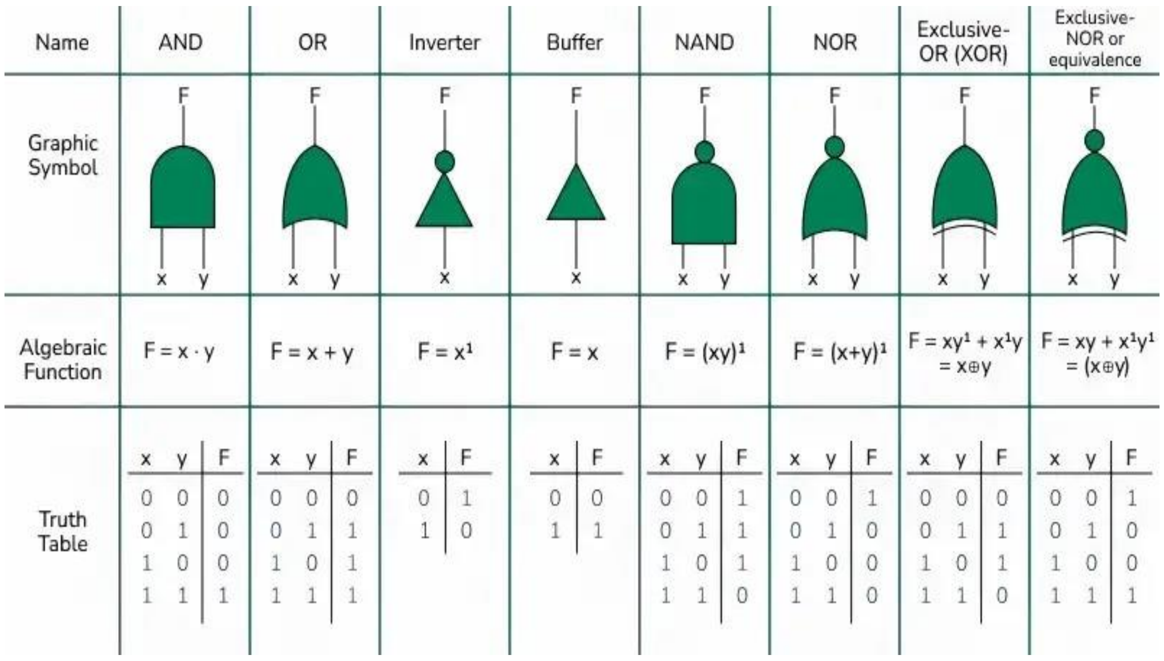
Aim : Write a VHDL Code for all basic logic gates and verify its functionality using simulation.

Software Used : MATLAB

Theory : Logic gates are the fundamental building blocks of digital circuits. These gates operate on binary inputs (0 and 1) to produce a single binary output based on specific logical functions.

Basic Logic Gates:

1. AND Gate: Produces a HIGH output (1) only when all its inputs are HIGH.
2. OR Gate: Produces a HIGH output (1) when at least one input is HIGH.
3. NOT Gate: Inverts the input (0 becomes 1, and 1 becomes 0).
4. NAND Gate: The complement of the AND gate (NOT AND).
5. NOR Gate: The complement of the OR gate (NOT OR).
6. XOR Gate: Produces a HIGH output (1) when the number of HIGH inputs is odd.
7. XNOR Gate: The complement of the XOR gate (produces HIGH when inputs are equal).



Name	AND	OR	Inverter	Buffer	NAND	NOR	Exclusive-OR (XOR)	Exclusive-NOR or equivalence
Graphic Symbol	F; x y	F; x y	F; x	F; x	F; x y	F; x y	F; x y	F; x y
Algebraic Function	$F = x \cdot y$	$F = x + y$	$F = x^1$	$F = x$	$F = (xy)^1$	$F = (x+y)^1$	$F = xy^1 + x^1y = x \oplus y$	$F = xy + x^1y^1 = (x \oplus y)$

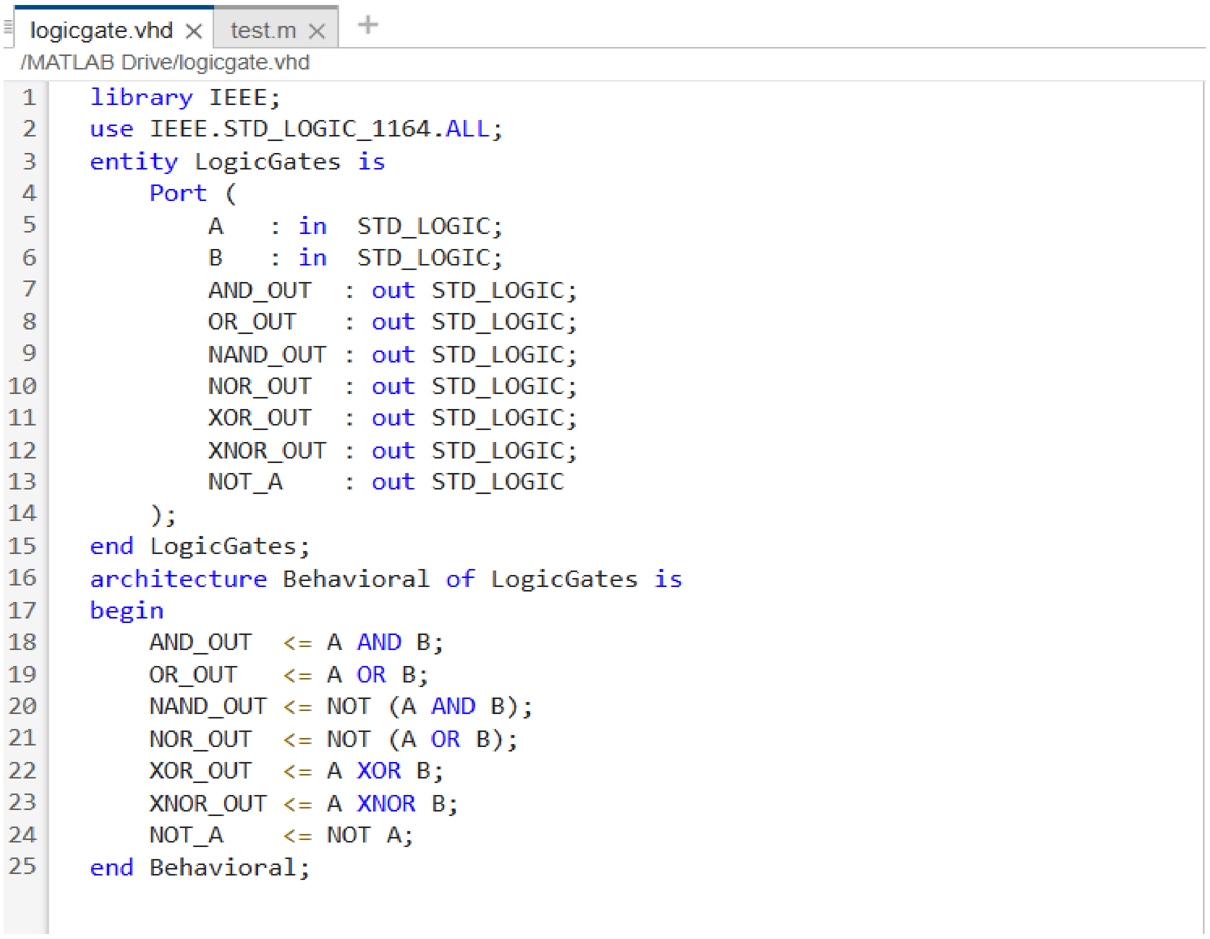
Truth Table

x	y	AND F	OR F	NAND F	NOR F	XOR F	XNOR F
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1

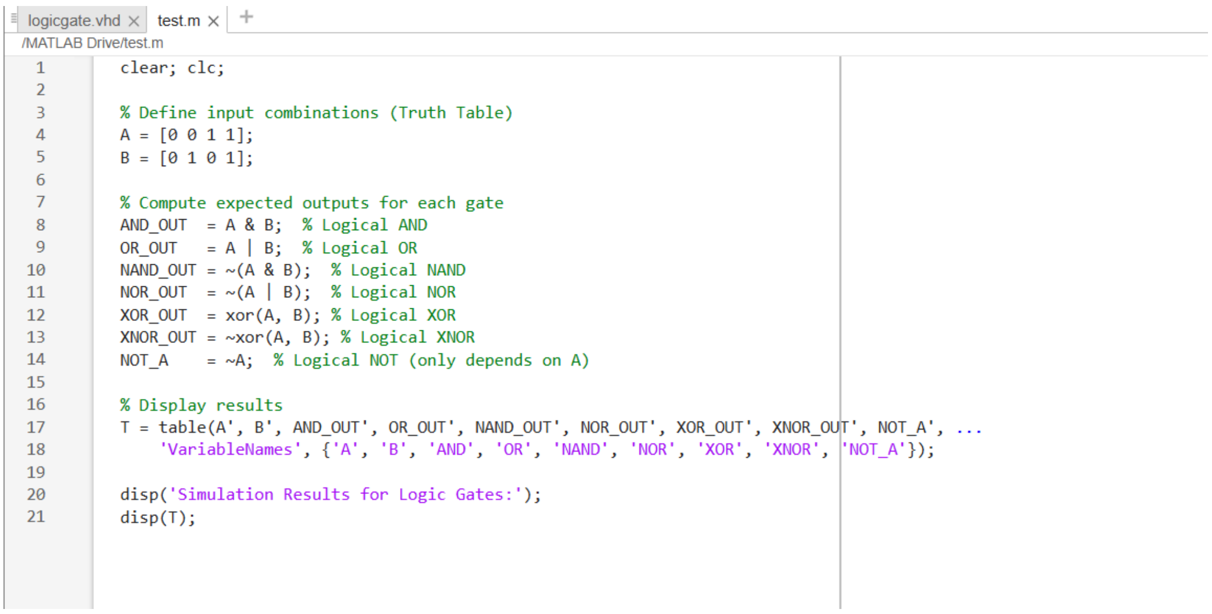
x	Inverter F	Buffer F
0	1	0
1	0	1

Circuit Diagram and its Algebraic Function

VHDL Code :

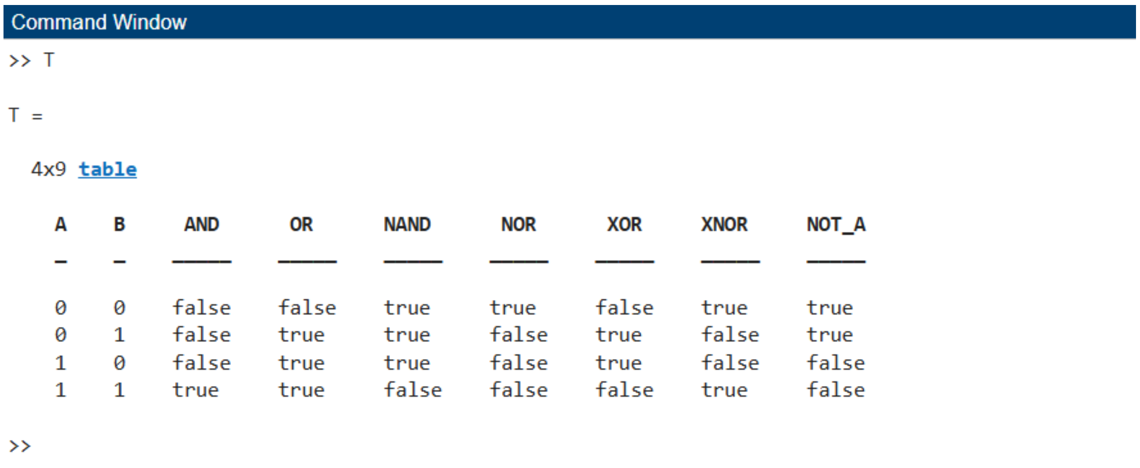


```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity LogicGates is
    Port (
        A    : in   STD_LOGIC;
        B    : in   STD_LOGIC;
        AND_OUT   : out STD_LOGIC;
        OR_OUT    : out STD_LOGIC;
        NAND_OUT  : out STD_LOGIC;
        NOR_OUT   : out STD_LOGIC;
        XOR_OUT   : out STD_LOGIC;
        XNOR_OUT  : out STD_LOGIC;
        NOT_A     : out STD_LOGIC
    );
end LogicGates;
architecture Behavioral of LogicGates is
begin
    AND_OUT   <= A AND B;
    OR_OUT    <= A OR B;
    NAND_OUT  <= NOT (A AND B);
    NOR_OUT   <= NOT (A OR B);
    XOR_OUT   <= A XOR B;
    XNOR_OUT  <= A XNOR B;
    NOT_A     <= NOT A;
end Behavioral;
```



```matlab
clear; clc;

% Define input combinations (Truth Table)
A = [0 0 1 1];
B = [0 1 0 1];

% Compute expected outputs for each gate
AND_OUT  = A & B;  % Logical AND
OR_OUT   = A | B;  % Logical OR
NAND_OUT = ~(A & B);  % Logical NAND
NOR_OUT  = ~(A | B);  % Logical NOR
XOR_OUT  = xor(A, B); % Logical XOR
XNOR_OUT = ~xor(A, B); % Logical XNOR
NOT_A    = ~A;  % Logical NOT (only depends on A)

% Display results
T = table(A', B', AND_OUT', OR_OUT', NAND_OUT', NOR_OUT', XOR_OUT', XNOR_OUT', NOT_A', ...
    'VariableNames', {'A', 'B', 'AND', 'OR', 'NAND', 'NOR', 'XOR', 'XNOR', 'NOT_A'});

disp('Simulation Results for Logic Gates:');
disp(T);
```

Result:



```
>> T

T =

  4x9 table

    A    B     AND      OR      NAND     NOR      XOR     XNOR    NOT_A
    _    _    _____    _____    _____    _____    _____    _____    _____

    0    0    false    false    true     true     false    true     true 
    0    1    false    true     true     false    true     false    true 
    1    0    false    true     true     false    true     false    false
    1    1    true     true     false    false    false    true     false

>>
```

Conclusion : The successful implementation and verification of basic logic gates demonstrate the principles of Boolean logic. VHDL allows for precise hardware design, while MATLAB provides an efficient platform for functional verification. This experiment bridges the gap between theory and practical application.

Application of Logic Gates :

1. Digital Circuits: Used in arithmetic and logic units, memory, and control systems.
2. Data Processing: Employed in data transmission, storage, and retrieval.
3. Embedded Systems: Form the foundation of microcontrollers and microprocessors.

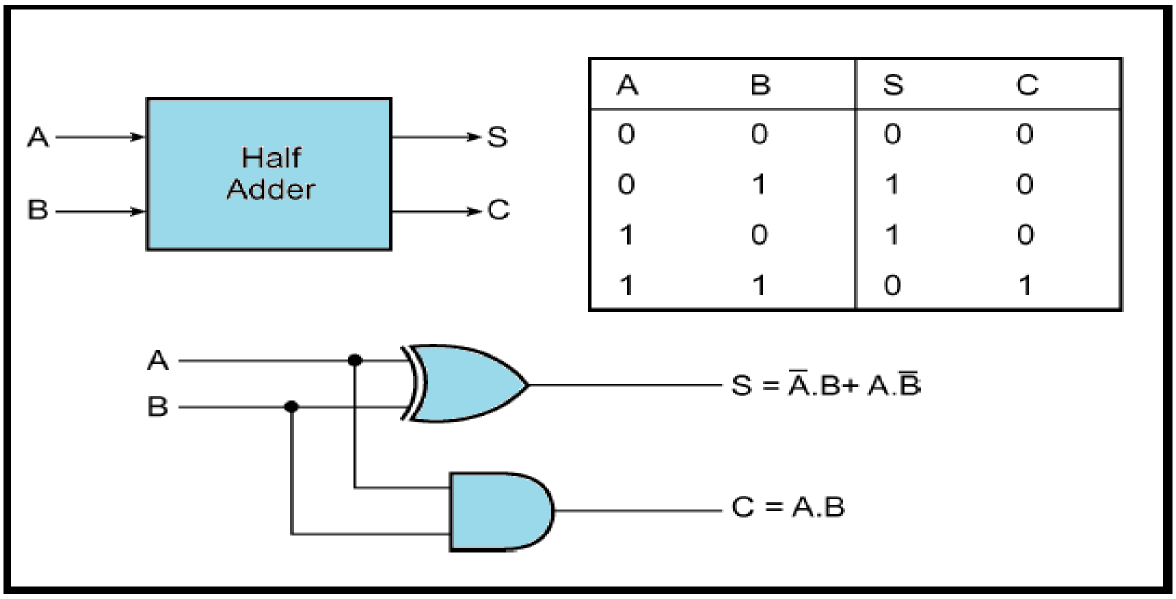
Experiment 2

Aim : Write a VHDL code for Half adder and Full adder circuits and simulate its output.

Software Used : MATLAB

Theory :

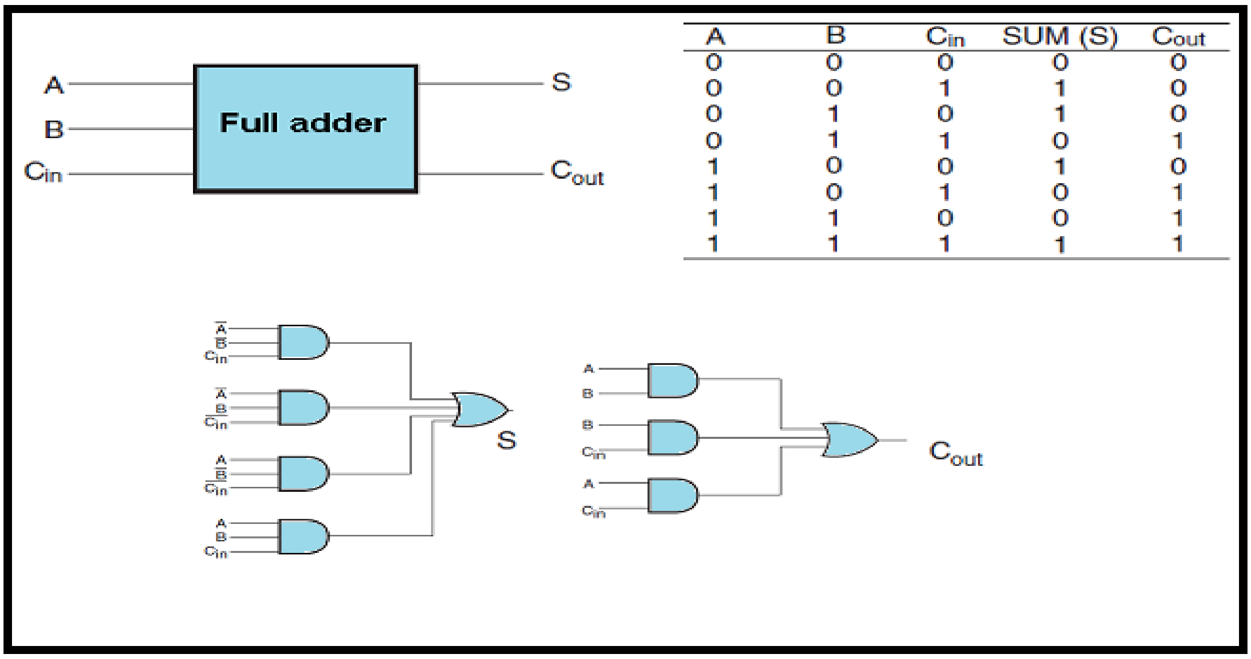
Half Adder: A Half Adder is a basic digital circuit that performs the addition of two single-bit binary numbers. The circuit outputs two values: Sum and Carry. The Sum is the result of the addition, and the Carry represents any overflow into the next bit position. The Half Adder uses basic logic gates: XOR for the Sum and AND for the Carry.



A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Circuit diagram and truth table for half adder

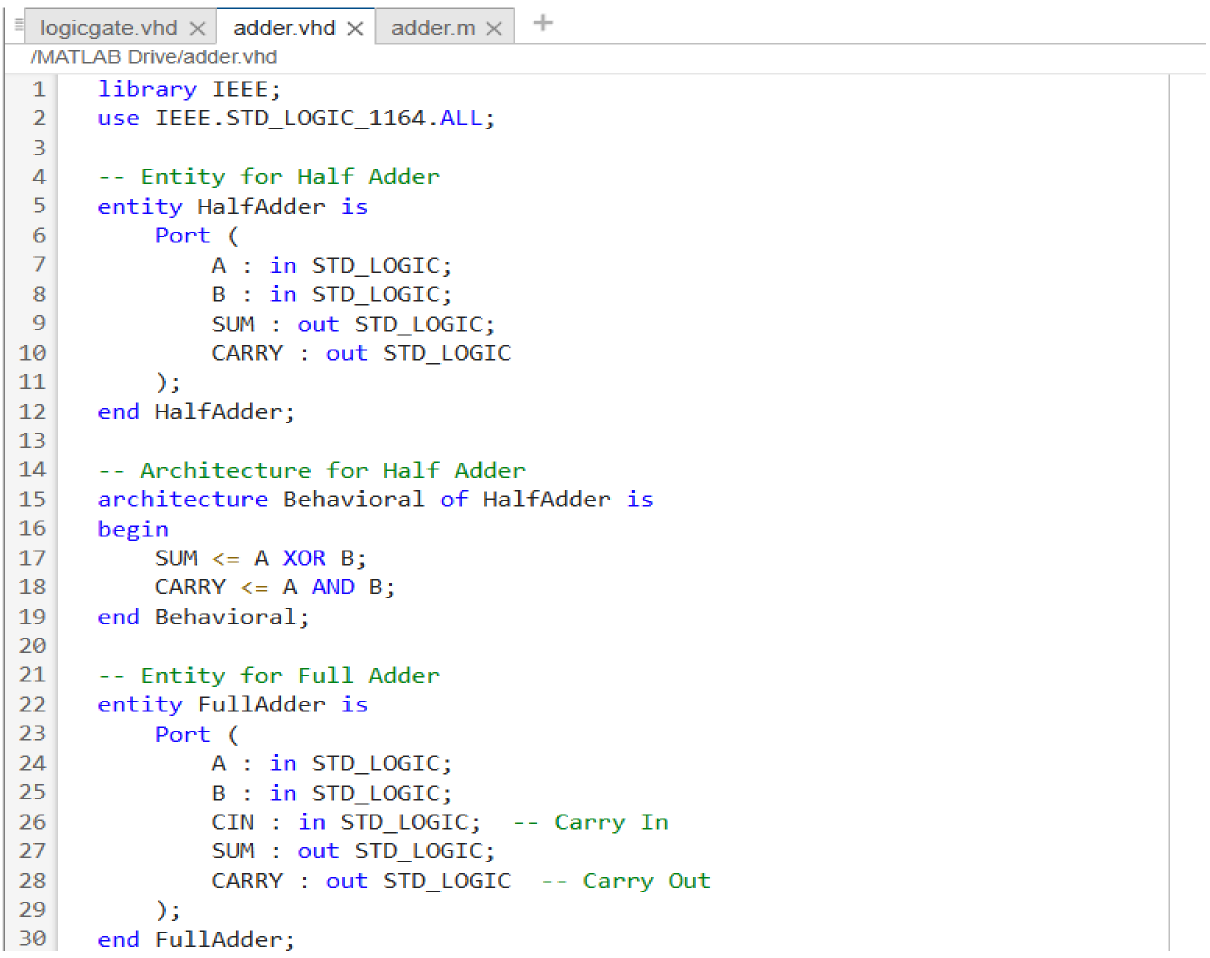
Full Adder: A Full Adder extends the concept of the Half Adder to include an additional input, Cin (Carry-in), allowing it to add three single-bit binary numbers (two significant bits and a carry-in). It outputs a Sum and a Carry. The Sum is computed using the XOR operation on all three inputs, while the Carry is determined using a combination of AND and OR gates to account for all possible carry-out condition.



A	B	C_{in}	SUM (S)	C_{out}
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

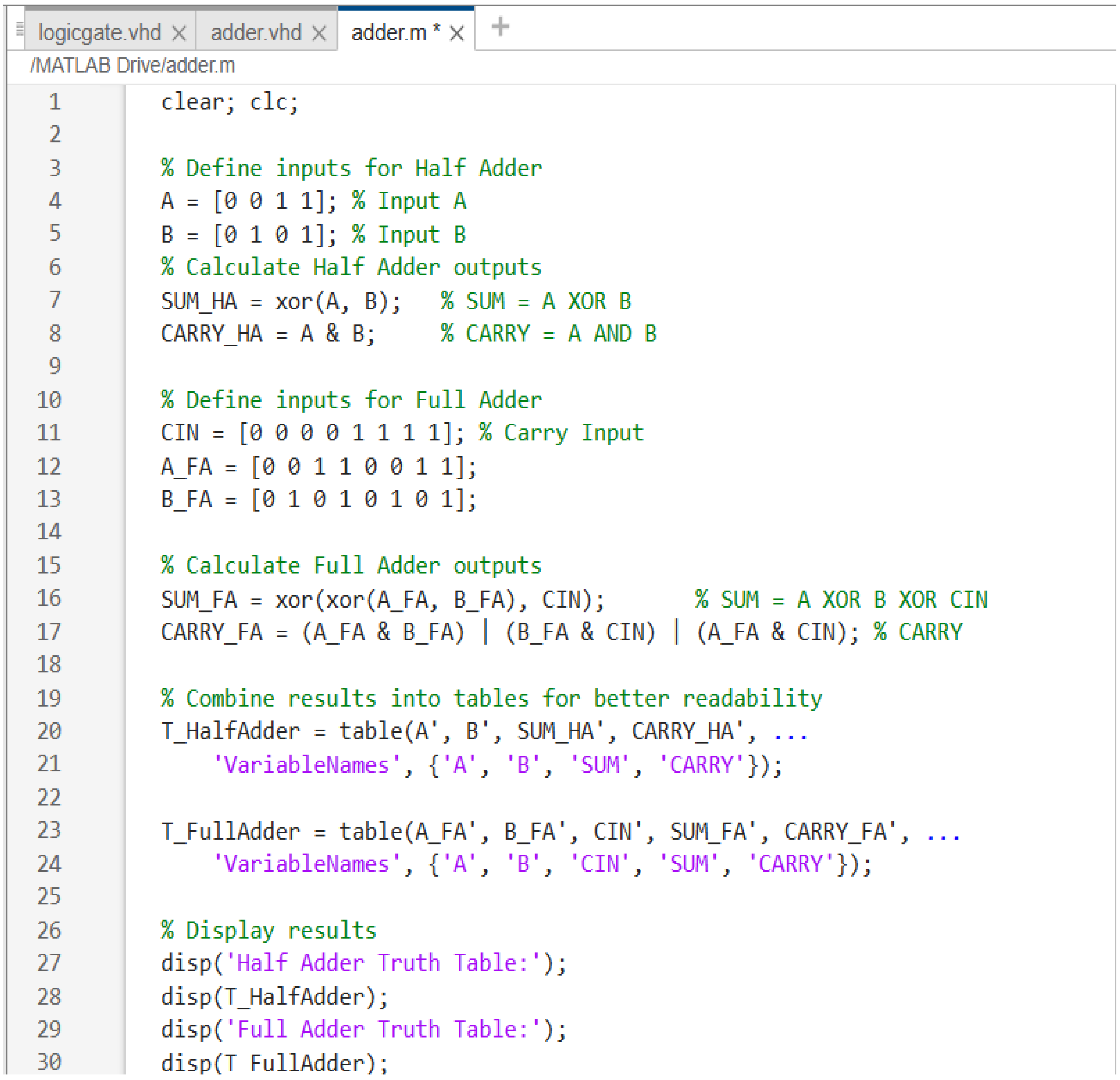
Circuit diagram and truth table for full adder.

VHDL Code:



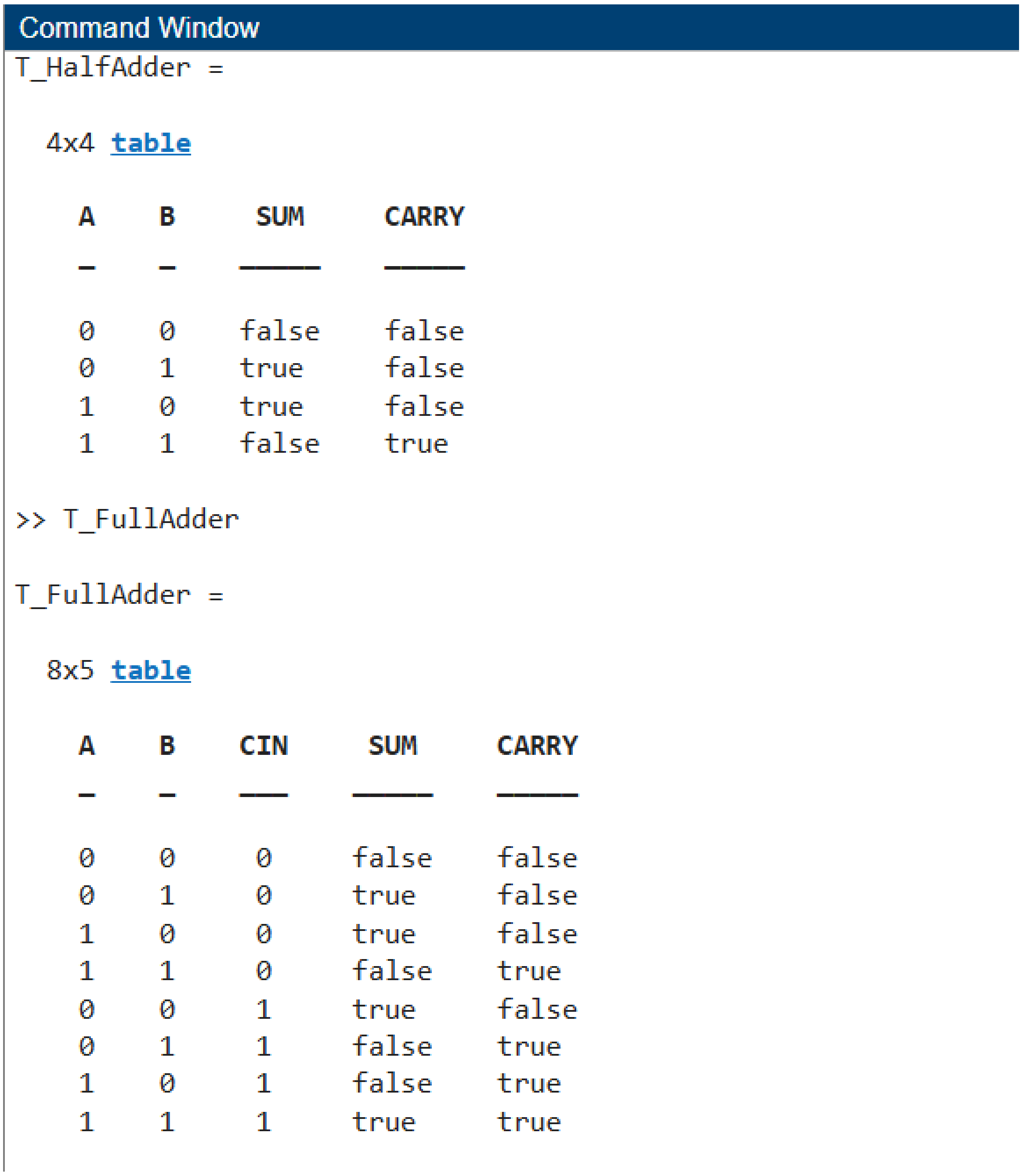
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity for Half Adder
entity HalfAdder is
    Port (
        A : in STD_LOGIC;
        B : in STD_LOGIC;
        SUM : out STD_LOGIC;
        CARRY : out STD_LOGIC
    );
end HalfAdder;

-- Architecture for Half Adder
architecture Behavioral of HalfAdder is
begin
    SUM <= A XOR B;
    CARRY <= A AND B;
end Behavioral;

-- Entity for Full Adder
entity FullAdder is
    Port (
        A : in STD_LOGIC;
        B : in STD_LOGIC;
        CIN : in STD_LOGIC;   -- Carry In
        SUM : out STD_LOGIC;
        CARRY : out STD_LOGIC   -- Carry Out
    );
end FullAdder;
```

```vhdl
-- Architecture for Full Adder
architecture Behavioral of FullAdder is
begin
    SUM <= A XOR B XOR CIN;
    CARRY <= (A AND B) OR (B AND CIN) OR (A AND CIN);
end Behavioral;
```



```matlab
clear; clc;

% Define inputs for Half Adder
A = [0 0 1 1]; % Input A
B = [0 1 0 1]; % Input B
% Calculate Half Adder outputs
SUM_HA = xor(A, B);    % SUM = A XOR B
CARRY_HA = A & B;      % CARRY = A AND B

% Define inputs for Full Adder
CIN = [0 0 0 0 1 1 1 1]; % Carry Input
A_FA = [0 0 1 1 0 0 1 1];
B_FA = [0 1 0 1 0 1 0 1];

% Calculate Full Adder outputs
SUM_FA = xor(xor(A_FA, B_FA), CIN);       % SUM = A XOR B XOR CIN
CARRY_FA = (A_FA & B_FA) | (B_FA & CIN) | (A_FA & CIN); % CARRY

% Combine results into tables for better readability
T_HalfAdder = table(A', B', SUM_HA', CARRY_HA', ...
    'VariableNames', {'A', 'B', 'SUM', 'CARRY'});

T_FullAdder = table(A_FA', B_FA', CIN', SUM_FA', CARRY_FA', ...
    'VariableNames', {'A', 'B', 'CIN', 'SUM', 'CARRY'});

% Display results
disp('Half Adder Truth Table:');
disp(T_HalfAdder);
disp('Full Adder Truth Table:');
disp(T FullAdder);
```

Result:



```
T_HalfAdder =

  4x4 table

    A    B    SUM      CARRY
    _    _    _____    _____

    0    0    false    false
    0    1    true     false
    1    0    true     false
    1    1    false    true

>> T_FullAdder

T_FullAdder =

  8x5 table

    A    B    CIN    SUM      CARRY
    _    _    ___    _____    _____

    0    0     0     false    false
    0    1     0     true     false
    1    0     0     true     false
    1    1     0     false    true
    0    0     1     true     false
    0    1     1     false    true
    1    0     1     false    true
    1    1     1     true     true
```

Conclusion : In this experiment, we successfully designed and simulated both Half Adder and Full Adder circuits using VHDL. The Half Adder demonstrated the basic functionality of adding two single-bit binary numbers, producing a Sum and a Carry. The Full Adder extended this capability by including a Carry-in input, enabling it to add three single-bit binary numbers, further showcasing the essential building blocks of binary addition in digital systems. The simulation results validated the correct operation of both adders, emphasizing their fundamental role in arithmetic operations within digital logic design.

Application of Half Adder :

1. Binary Calculators: Half Adders are integral components in binary calculators, where they perform the addition of two binary digits.
2. Digital Circuits: They are used in various digital circuits where simple addition of two bits is required, such as in encoders, decoders, and multiplexers.

Application of Full Adder:

1. Arithmetic Logic Units (ALUs): Full Adders are crucial components of ALUs in microprocessors and microcontrollers, performing multi-bit binary addition operations.
2. Digital Signal Processing: They are used in digital signal processing systems for executing arithmetic operations on digital signals, such as in FIR and IIR filters.

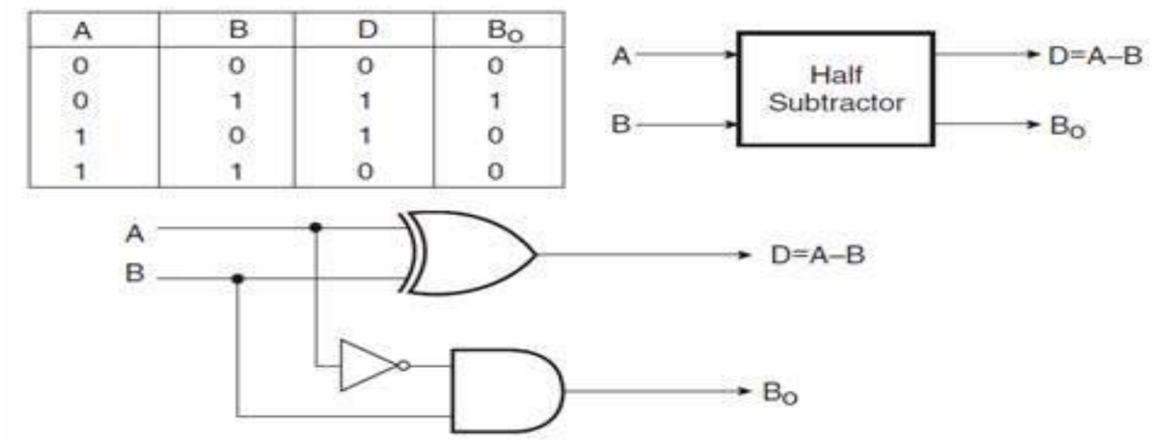
Experiment 3

Aim : Write a VHDL Code for Half Subtractor and Full Subtractor circuits and verify its functionality using simulation.

Software Used : MATLAB

Theory :

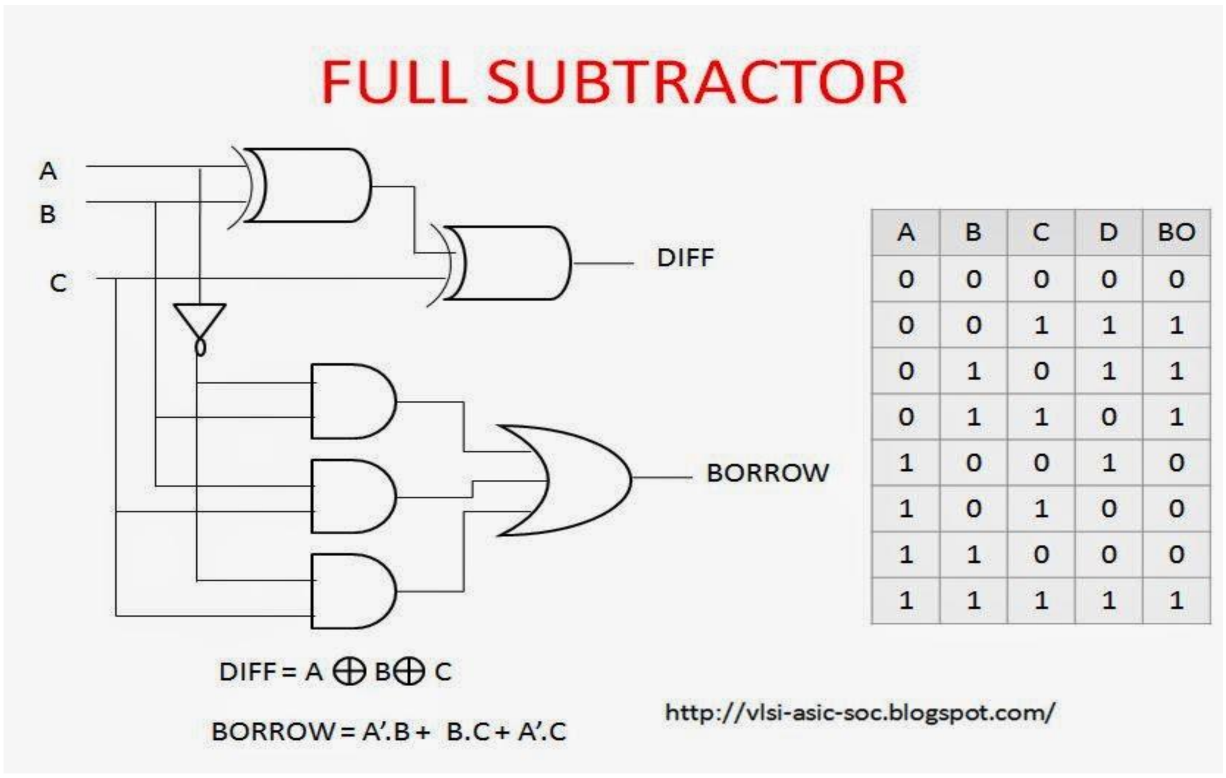
Half Subtractor: A half subtractor is a combinational circuit used in digital electronics to subtract one binary digit from another. It performs the subtraction of two bits, resulting in a difference and a borrow output. The half subtractor takes two inputs (the minuend and the subtrahend) and produces two outputs: the difference and the borrow. The difference indicates the result of the subtraction, while the borrow indicates if a 1 has been borrowed from a higher bit position to perform the subtraction. The half subtractor is fundamental in building more complex subtraction circuits and is often used in arithmetic logic units (ALUs) of digital systems.



A	B	D	B_O
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Circuit diagram and truth table of half subtractor

Full Subtractor: A full subtractor is an extension of the half subtractor that handles the subtraction of three binary digits: two significant bits and an input borrow from a previous subtraction. This makes the full subtractor more versatile and suitable for multi-bit binary subtraction. The full subtractor has three inputs (minuend, subtrahend, and borrow-in) and produces two outputs: the difference and the borrow-out. The difference represents the result of the subtraction, considering the borrow-in, while the borrow-out indicates whether a borrow is needed for the next higher bit position. The full subtractor is an essential building block for creating binary subtractors that can handle larger binary numbers.



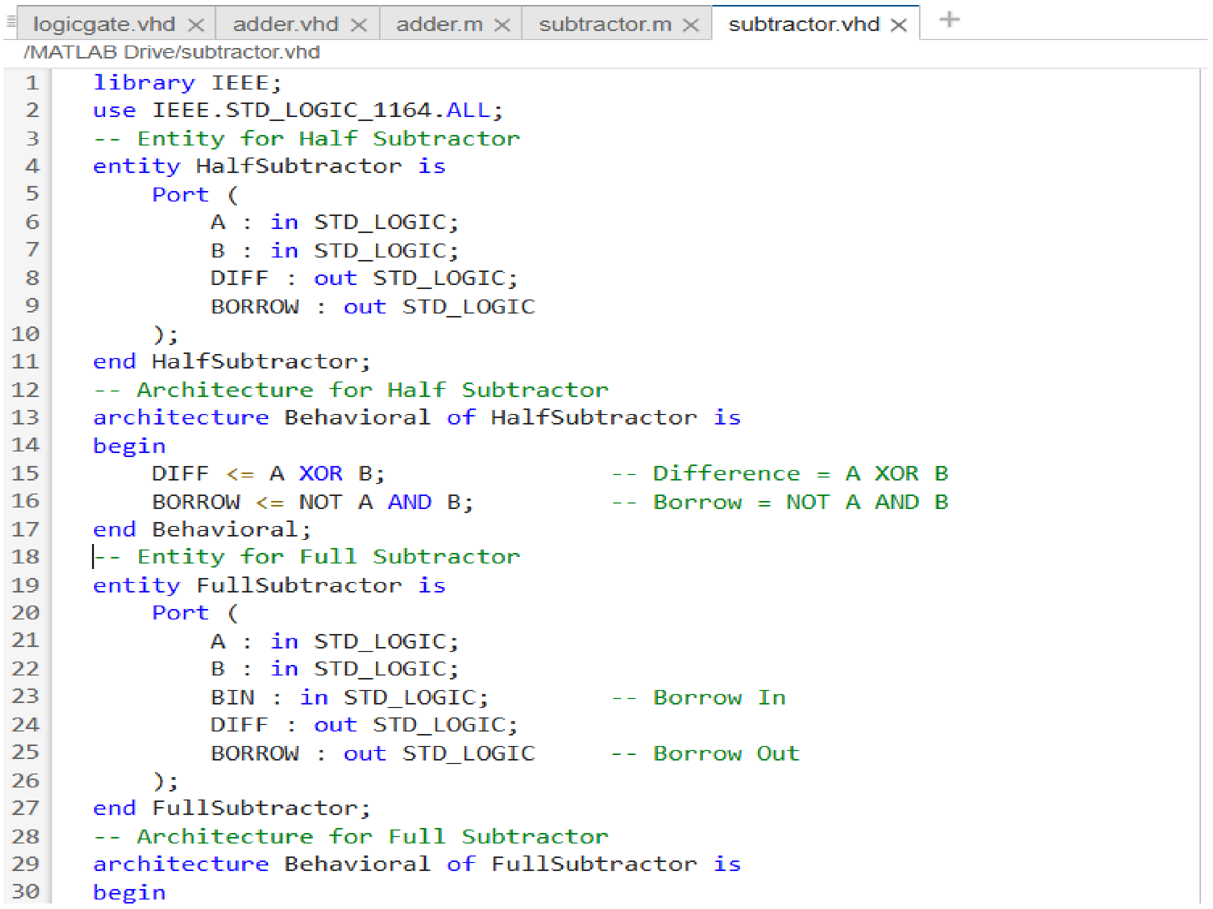
A	B	C	D	BO
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$$DIFF = A \oplus B \oplus C$$

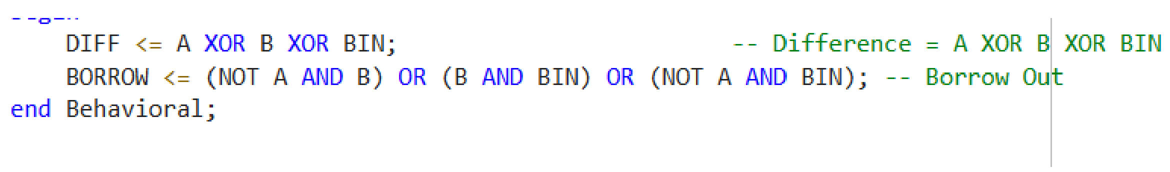
$$BORROW = A'.B + B.C + A'.C$$

Circuit diagram and truth table of full subtractor

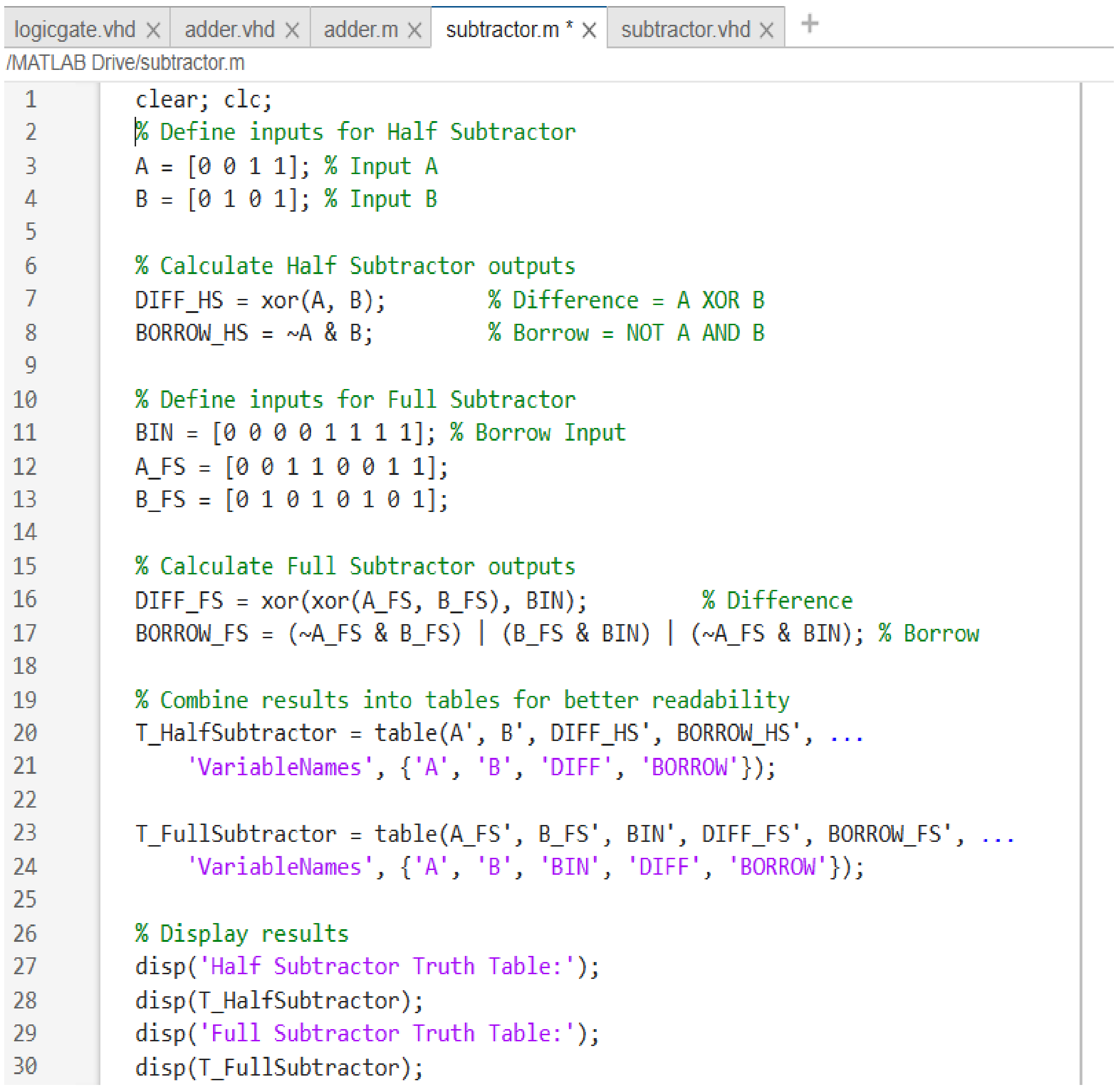
VHDL Code:



```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- Entity for Half Subtractor
entity HalfSubtractor is
    Port (
        A : in STD_LOGIC;
        B : in STD_LOGIC;
        DIFF : out STD_LOGIC;
        BORROW : out STD_LOGIC
    );
end HalfSubtractor;
-- Architecture for Half Subtractor
architecture Behavioral of HalfSubtractor is
begin
    DIFF <= A XOR B;             -- Difference = A XOR B
    BORROW <= NOT A AND B;       -- Borrow = NOT A AND B
end Behavioral;
-- Entity for Full Subtractor
entity FullSubtractor is
    Port (
        A : in STD_LOGIC;
        B : in STD_LOGIC;
        BIN : in STD_LOGIC;      -- Borrow In
        DIFF : out STD_LOGIC;
        BORROW : out STD_LOGIC   -- Borrow Out
    );
end FullSubtractor;
-- Architecture for Full Subtractor
architecture Behavioral of FullSubtractor is
begin
```

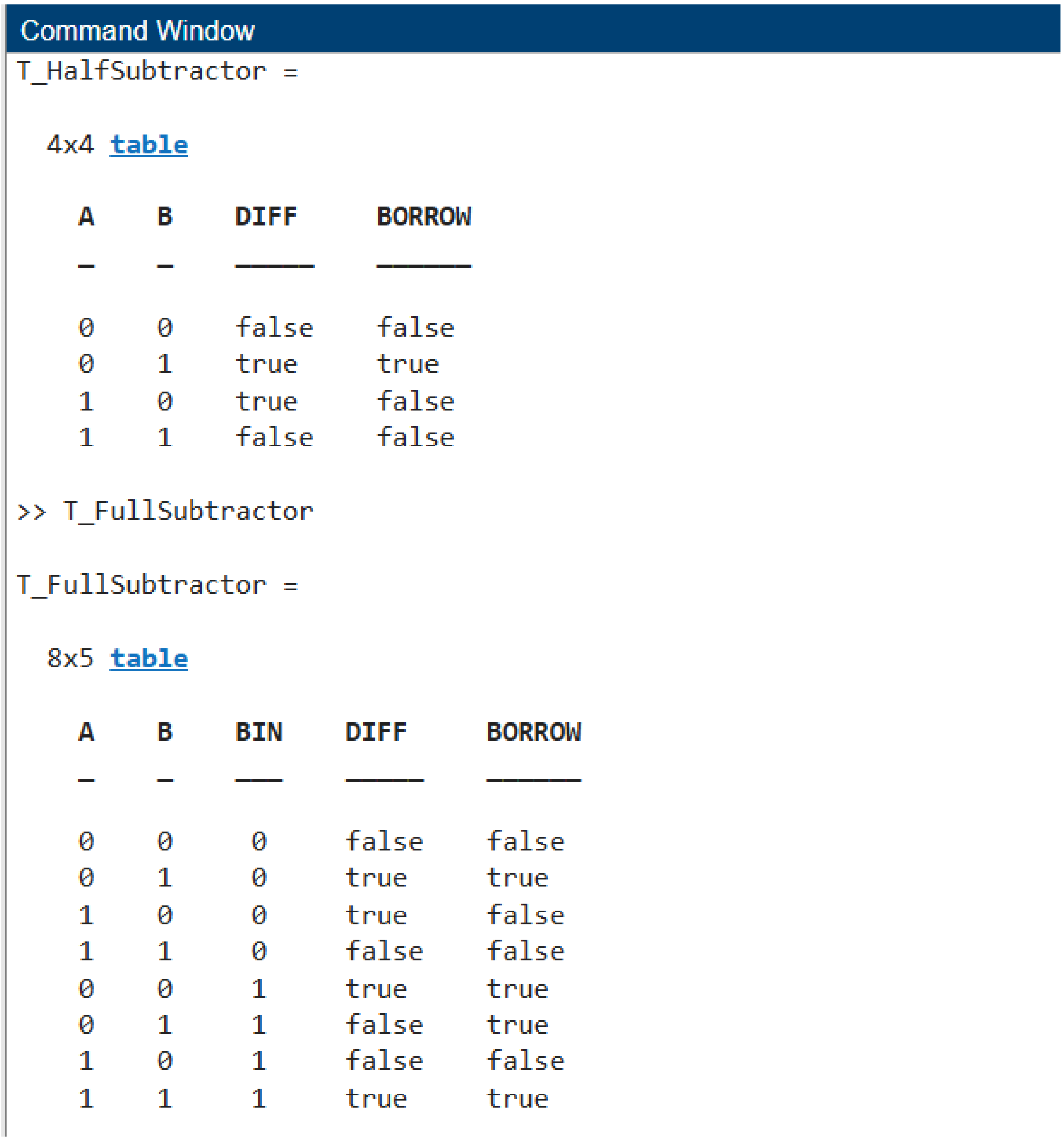


```
    DIFF <= A XOR B XOR BIN;                          -- Difference = A XOR B XOR BIN
    BORROW <= (NOT A AND B) OR (B AND BIN) OR (NOT A AND BIN); -- Borrow Out
end Behavioral;
```



```
clear; clc;
% Define inputs for Half Subtractor
A = [0 0 1 1]; % Input A
B = [0 1 0 1]; % Input B

% Calculate Half Subtractor outputs
DIFF_HS = xor(A, B);        % Difference = A XOR B
BORROW_HS = ~A & B;         % Borrow = NOT A AND B

% Define inputs for Full Subtractor
BIN = [0 0 0 0 1 1 1 1]; % Borrow Input
A_FS = [0 0 1 1 0 0 1 1];
B_FS = [0 1 0 1 0 1 0 1];

% Calculate Full Subtractor outputs
DIFF_FS = xor(xor(A_FS, B_FS), BIN);         % Difference
BORROW_FS = (~A_FS & B_FS) | (B_FS & BIN) | (~A_FS & BIN); % Borrow

% Combine results into tables for better readability
T_HalfSubtractor = table(A', B', DIFF_HS', BORROW_HS', ...
    'VariableNames', {'A', 'B', 'DIFF', 'BORROW'});

T_FullSubtractor = table(A_FS', B_FS', BIN', DIFF_FS', BORROW_FS', ...
    'VariableNames', {'A', 'B', 'BIN', 'DIFF', 'BORROW'});

% Display results
disp('Half Subtractor Truth Table:');
disp(T_HalfSubtractor);
disp('Full Subtractor Truth Table:');
disp(T_FullSubtractor);
```

Result:



```
Command Window
T_HalfSubtractor =

  4x4 table

    A    B    DIFF     BORROW
    _    _    _____    ______

    0    0    false    false 
    0    1    true     true  
    1    0    true     false 
    1    1    false    false 

>> T_FullSubtractor

T_FullSubtractor =

  8x5 table

    A    B    BIN    DIFF     BORROW
    _    _    ___    _____    ______

    0    0     0     false    false 
    0    1     0     true     true  
    1    0     0     true     false 
    1    1     0     false    false 
    0    0     1     true     true  
    0    1     1     false    true  
    1    0     1     false    false 
    1    1     1     true     true  
```

Conclusion: In this experiment, we examined the workings of half and full subtractors, crucial components for binary subtraction in digital electronics. The half subtractor allowed us to understand the basic concept of subtracting two binary digits and producing a difference and borrow output.

Building on this, the full subtractor introduced a third input for borrowing, enabling more complex multi-bit binary subtraction. These concepts are foundational for designing arithmetic circuits in digital systems like ALUs.This experiment highlighted the importance of half and full subtractors in performing essential arithmetic operations within digital systems.

Application of Half Subtractor:

1. Digital Circuits: Used in simple arithmetic operations within digital circuits, such as calculators and small-scale computing devices.
2. Data Processing: Utilized in data processing systems where basic subtraction of binary digits is required.

Application of Full Subtractor:

1. Multi-bit Subtraction: Essential in multi-bit binary subtraction in digital systems, such as computer processors and arithmetic logic units (ALUs).
2. Digital Systems Design: Integral in designing complex digital systems that require efficient handling of binary subtraction with borrow operations, like digital signal processors (DSPs).

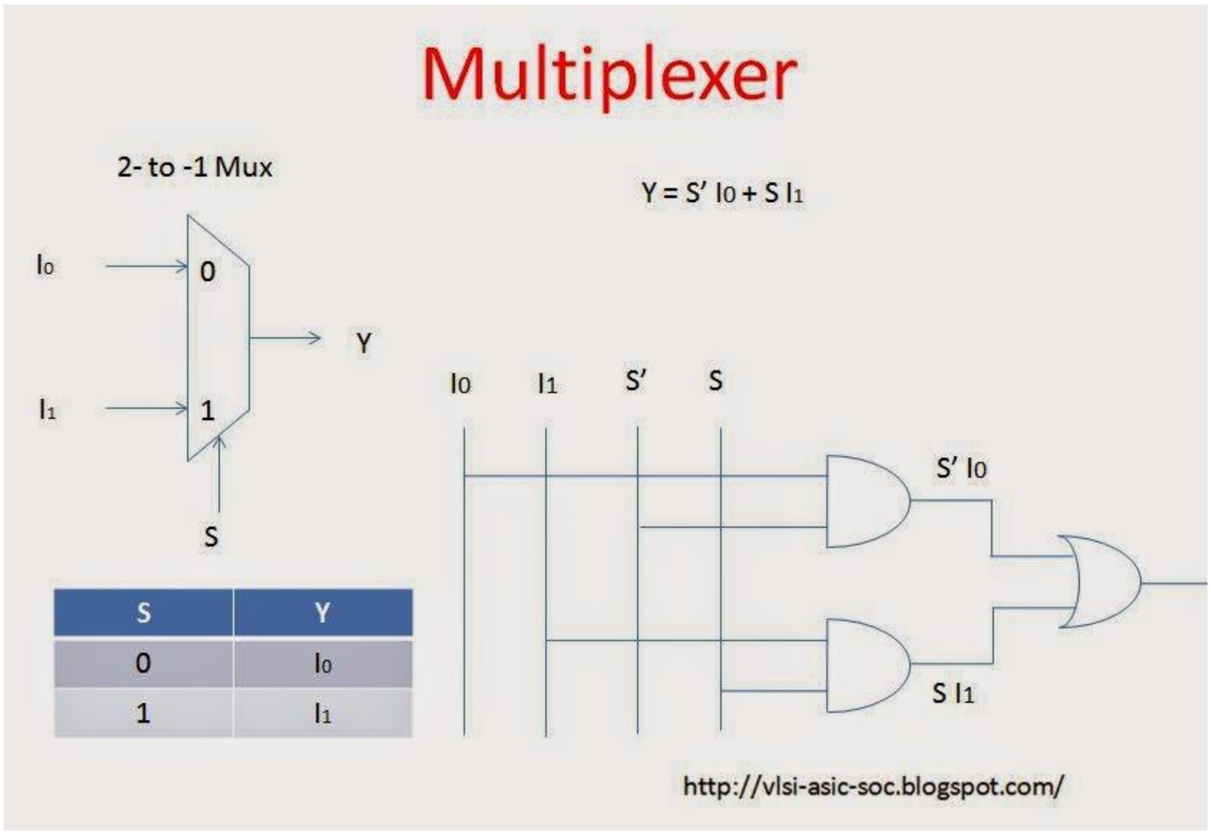
Experiment 4

Aim : Write a VHDL Code for 2x1.4x1,8x1 Multiplexer and verify it's working.

Software Used : MATLAB

Theory : A multiplexer (MUX) is a digital circuit that selects one of several input signals and forwards the chosen input to a single output line. The selection is controlled by a set of selection inputs.

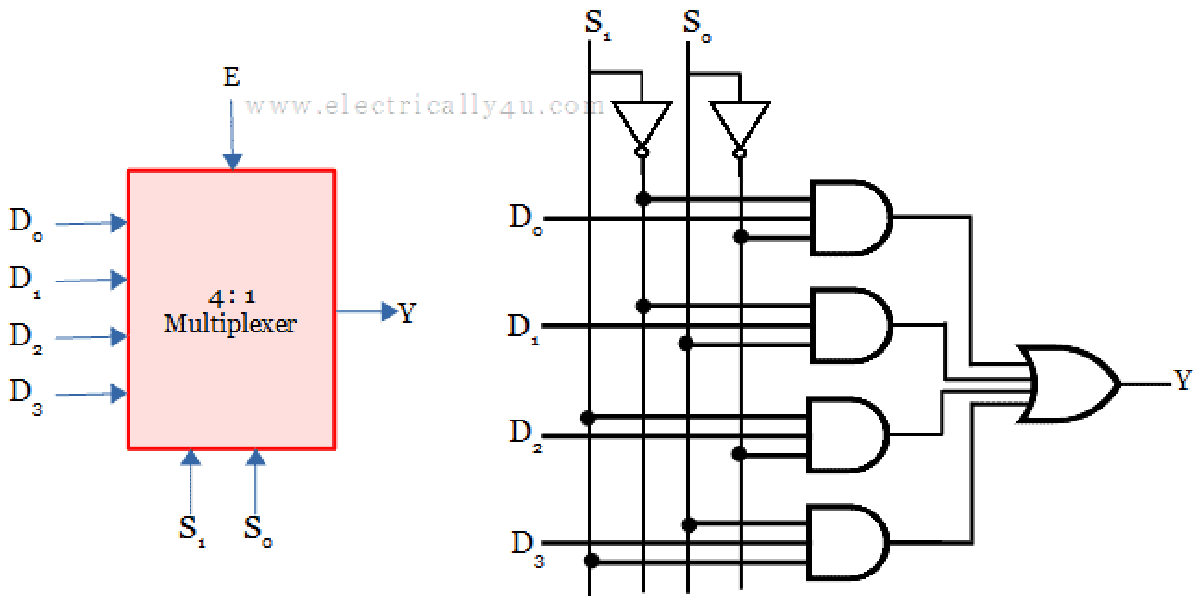
1. 2x1 MUX: Has two data inputs (I0, I1) and one selection input (S). It directs the input (I0 or I1) based on the value of S.



S	Y
0	I_0
1	I_1

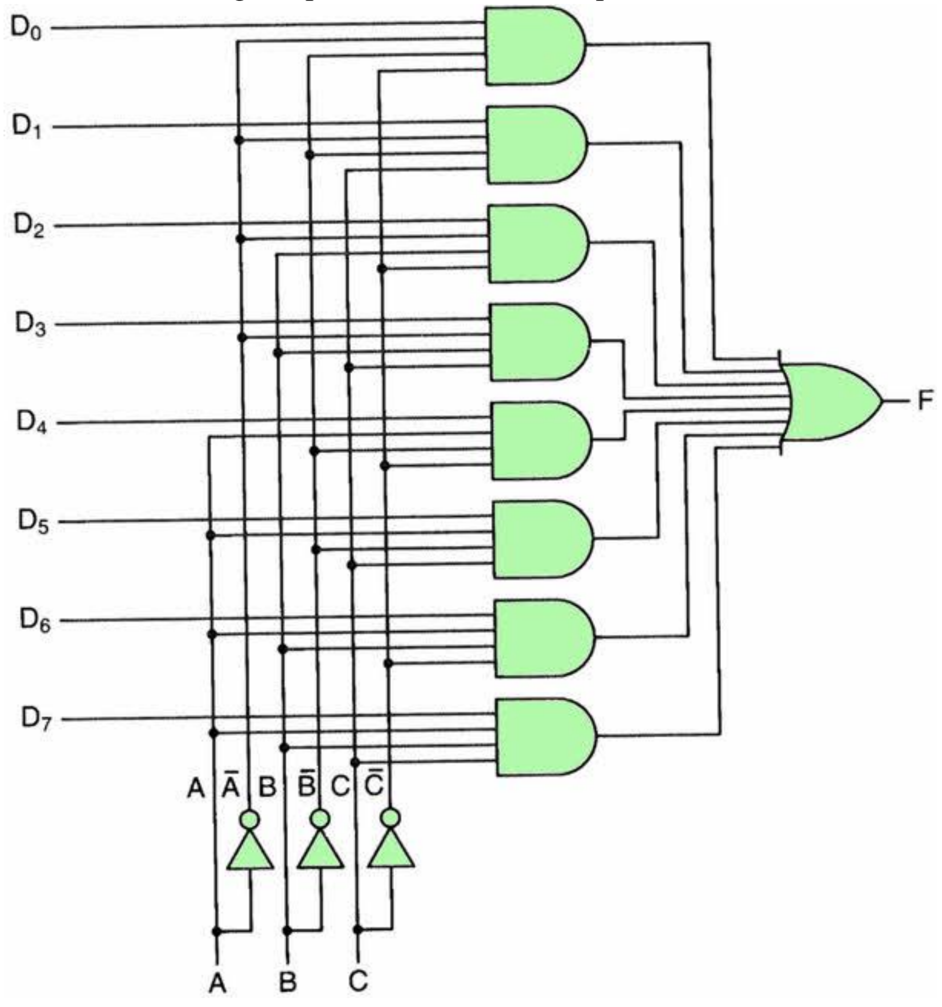
Circuit diagram and block diagram of 2 x 1 Multiplexer

2. 4x1 MUX: Has four data inputs (I0, I1, I2, I3) and two selection inputs (S0, S1). It selects one of the inputs to route to the output based on the combination of S0 and S1.

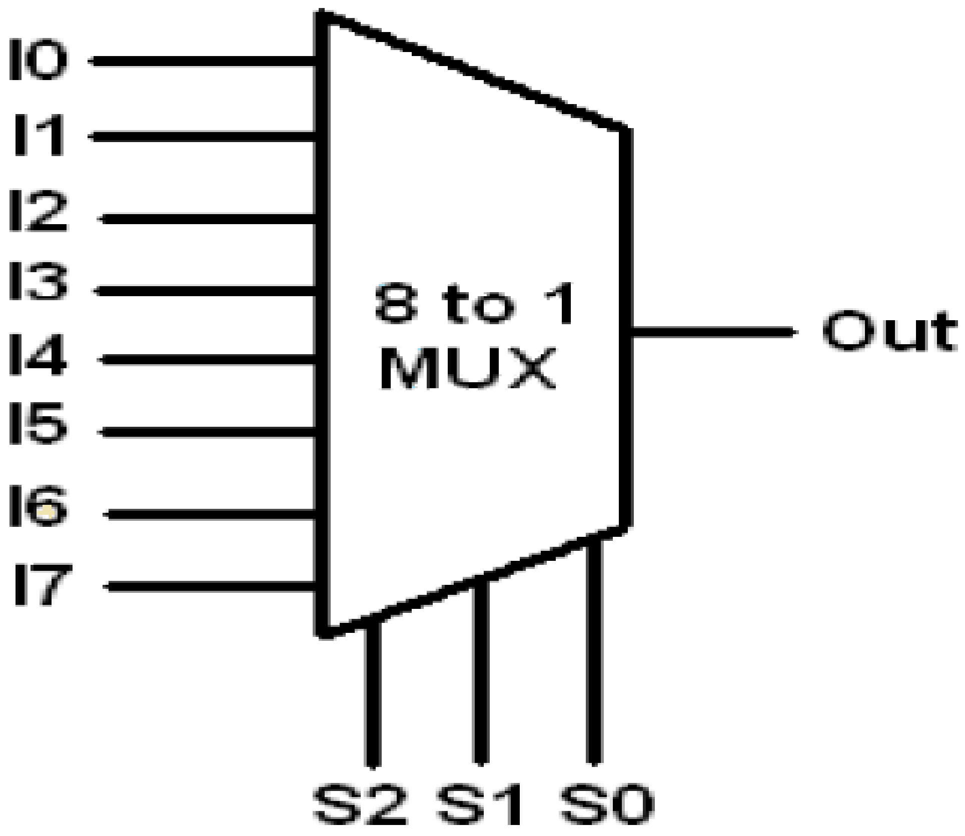


Circuit diagram and block diagramof 4 x 1 Multiplexer

3. 8x1 MUX: Has eight data inputs (I0 to I7) and three selection inputs (S0, S1, S2). It selects one of the eight inputs to route to the output based on the combination of S0,S1,S2

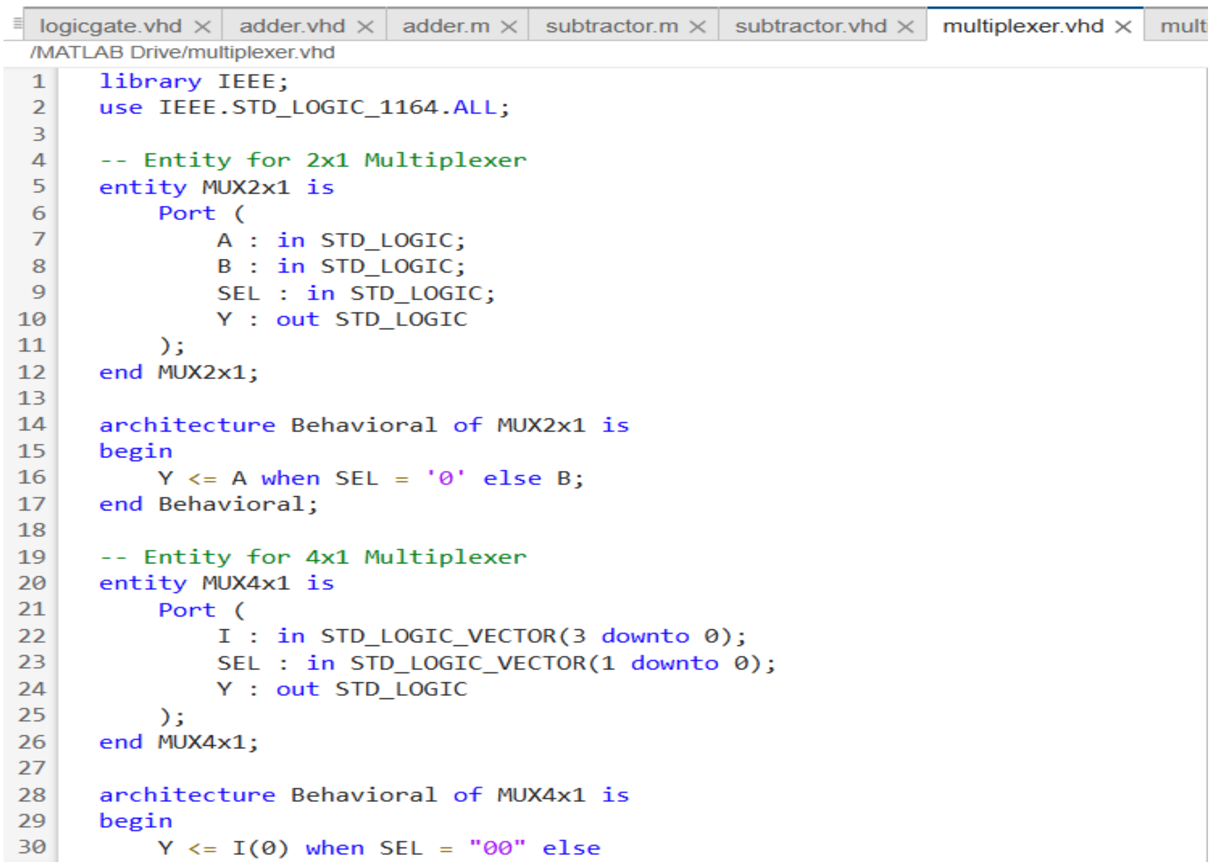


.

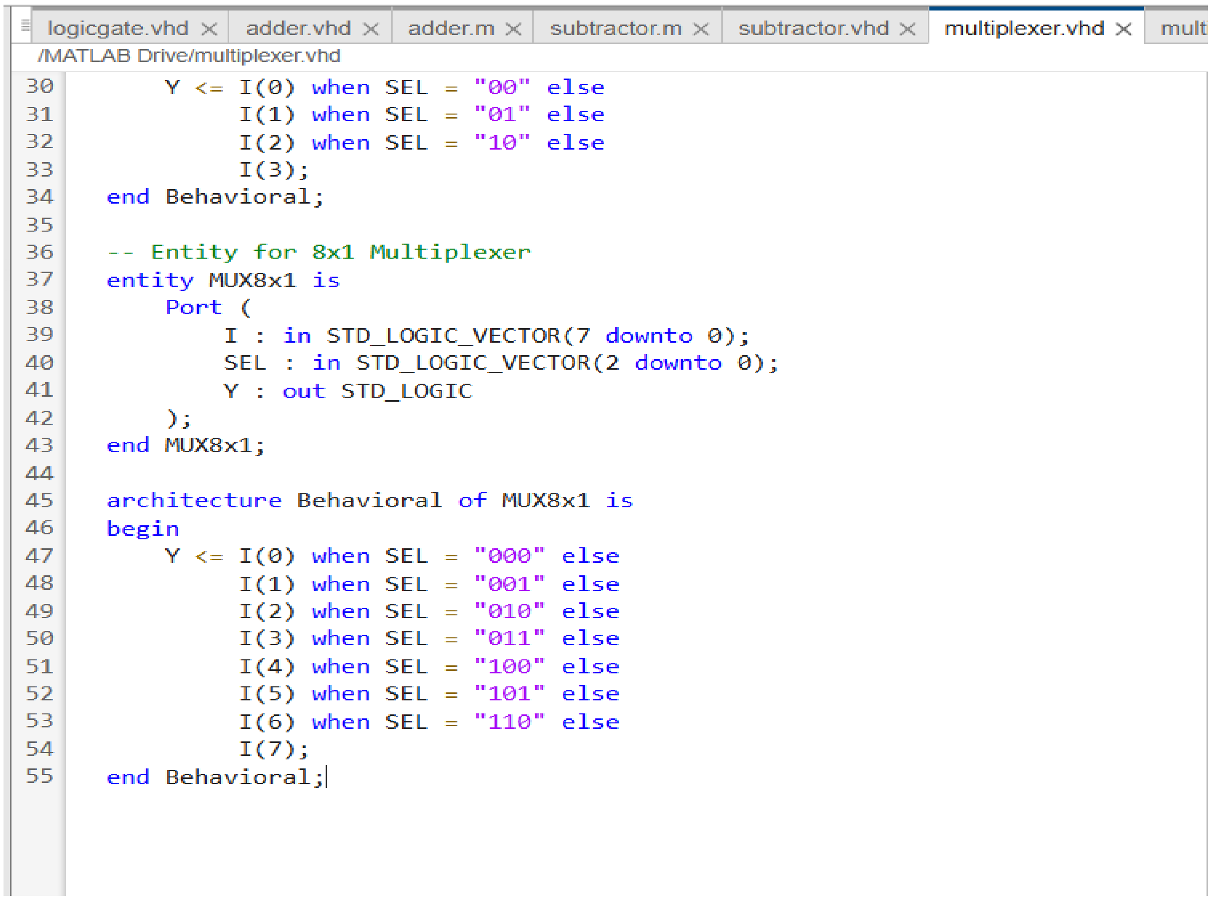


Circuit diagram and block diagram of 8 x1 Multiplexer

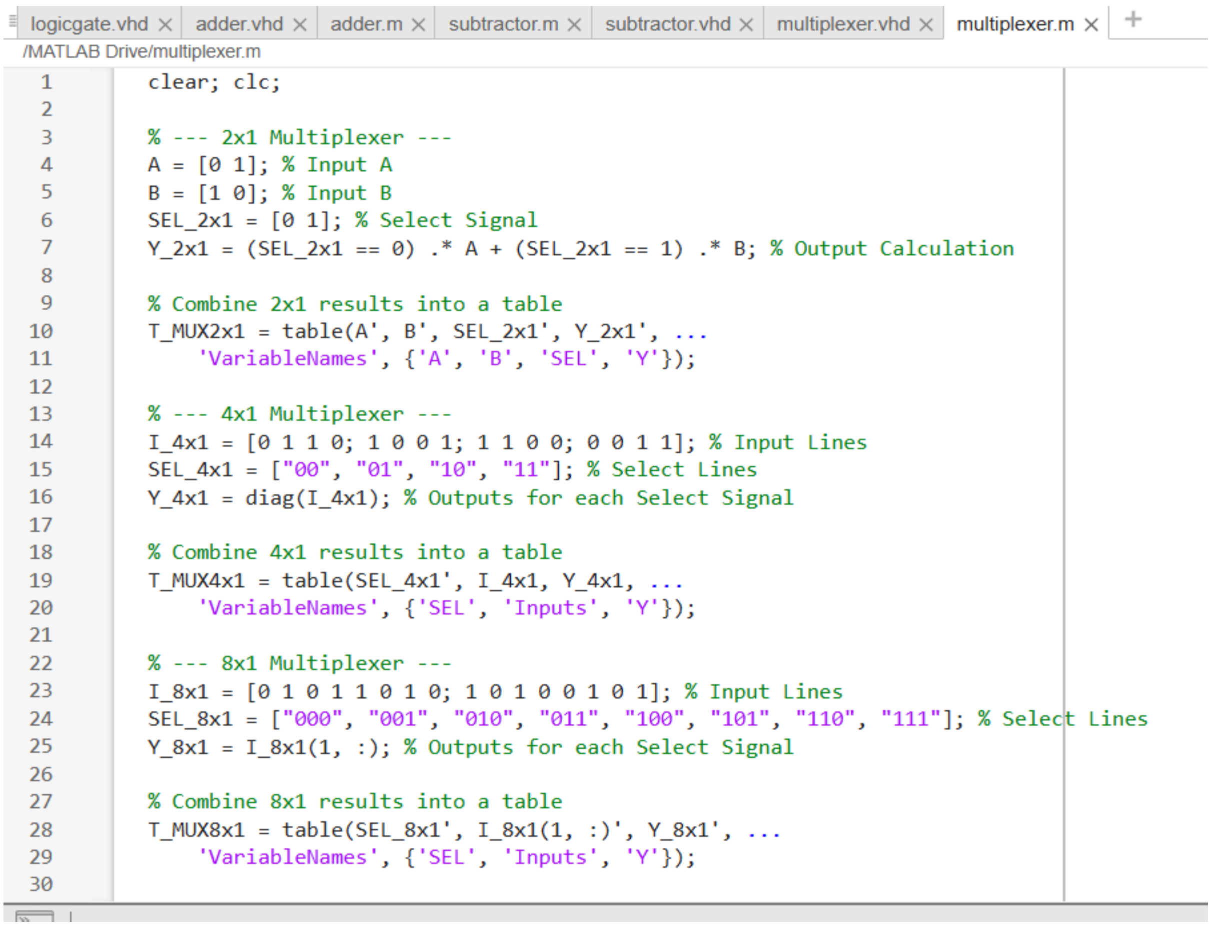
VHDL Code :



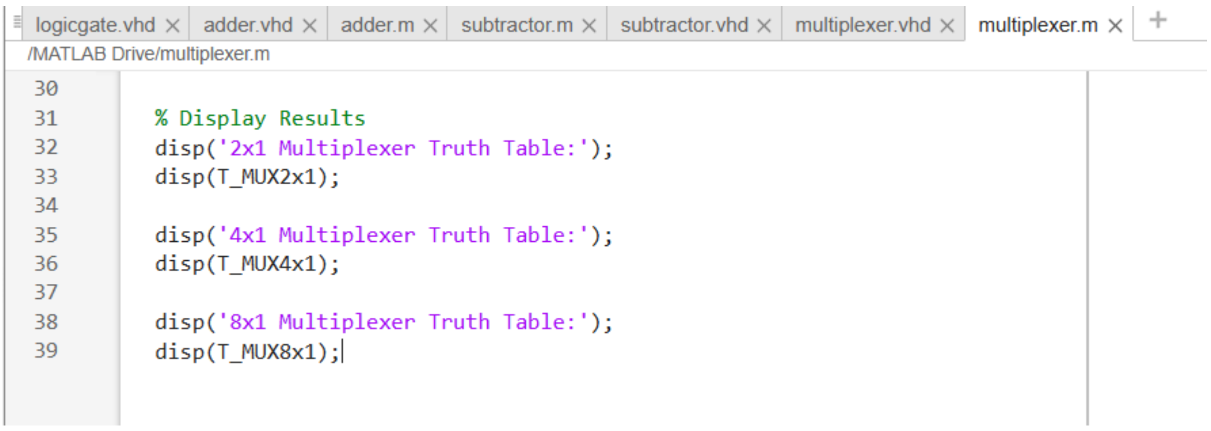
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity for 2x1 Multiplexer
entity MUX2x1 is
    Port (
        A : in STD_LOGIC;
        B : in STD_LOGIC;
        SEL : in STD_LOGIC;
        Y : out STD_LOGIC
    );
end MUX2x1;

architecture Behavioral of MUX2x1 is
begin
    Y <= A when SEL = '0' else B;
end Behavioral;

-- Entity for 4x1 Multiplexer
entity MUX4x1 is
    Port (
        I : in STD_LOGIC_VECTOR(3 downto 0);
        SEL : in STD_LOGIC_VECTOR(1 downto 0);
        Y : out STD_LOGIC
    );
end MUX4x1;

architecture Behavioral of MUX4x1 is
begin
    Y <= I(0) when SEL = "00" else
```



```vhdl
    Y <= I(0) when SEL = "00" else
         I(1) when SEL = "01" else
         I(2) when SEL = "10" else
         I(3);
end Behavioral;

-- Entity for 8x1 Multiplexer
entity MUX8x1 is
    Port (
        I : in STD_LOGIC_VECTOR(7 downto 0);
        SEL : in STD_LOGIC_VECTOR(2 downto 0);
        Y : out STD_LOGIC
    );
end MUX8x1;

architecture Behavioral of MUX8x1 is
begin
    Y <= I(0) when SEL = "000" else
         I(1) when SEL = "001" else
         I(2) when SEL = "010" else
         I(3) when SEL = "011" else
         I(4) when SEL = "100" else
         I(5) when SEL = "101" else
         I(6) when SEL = "110" else
         I(7);
end Behavioral;
```

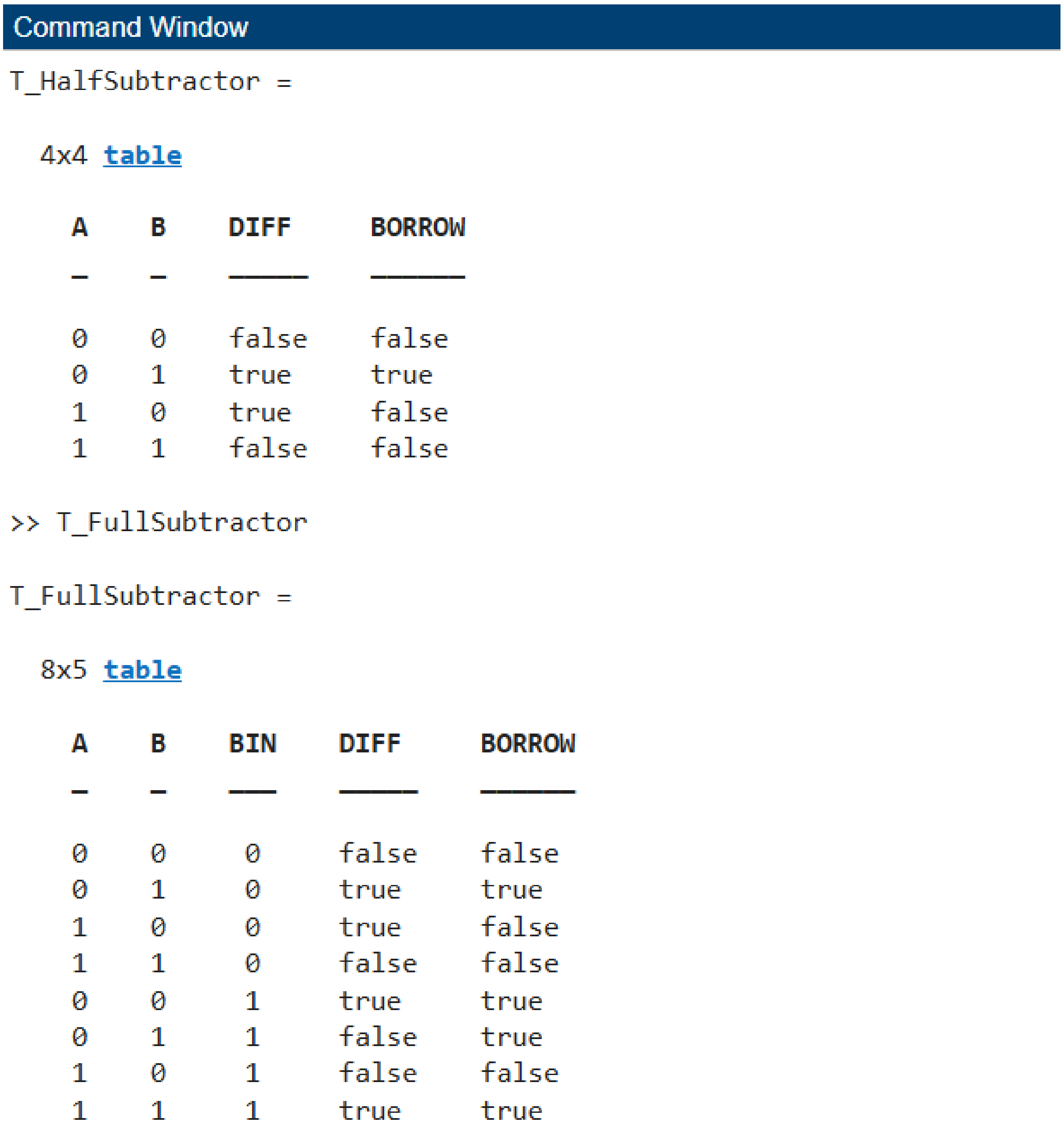


```matlab
clear; clc;

% --- 2x1 Multiplexer ---
A = [0 1]; % Input A
B = [1 0]; % Input B
SEL_2x1 = [0 1]; % Select Signal
Y_2x1 = (SEL_2x1 == 0) .* A + (SEL_2x1 == 1) .* B; % Output Calculation

% Combine 2x1 results into a table
T_MUX2x1 = table(A', B', SEL_2x1', Y_2x1', ...
    'VariableNames', {'A', 'B', 'SEL', 'Y'});

% --- 4x1 Multiplexer ---
I_4x1 = [0 1 1 0; 1 0 0 1; 1 1 0 0; 0 0 1 1]; % Input Lines
SEL_4x1 = ["00", "01", "10", "11"]; % Select Lines
Y_4x1 = diag(I_4x1); % Outputs for each Select Signal

% Combine 4x1 results into a table
T_MUX4x1 = table(SEL_4x1', I_4x1, Y_4x1, ...
    'VariableNames', {'SEL', 'Inputs', 'Y'});

% --- 8x1 Multiplexer ---
I_8x1 = [0 1 0 1 1 0 1 0; 1 0 1 0 0 1 0 1]; % Input Lines
SEL_8x1 = ["000", "001", "010", "011", "100", "101", "110", "111"]; % Select Lines
Y_8x1 = I_8x1(1, :); % Outputs for each Select Signal

% Combine 8x1 results into a table
T_MUX8x1 = table(SEL_8x1', I_8x1(1, :)', Y_8x1', ...
    'VariableNames', {'SEL', 'Inputs', 'Y'});

```



```

    % Display Results
    disp('2x1 Multiplexer Truth Table:');
    disp(T_MUX2x1);

    disp('4x1 Multiplexer Truth Table:');
    disp(T_MUX4x1);

    disp('8x1 Multiplexer Truth Table:');
    disp(T_MUX8x1);
```

Result :



```
T_HalfSubtractor =

  4x4 table

    A    B    DIFF     BORROW
    _    _    _____    ______

    0    0    false    false 
    0    1    true     true  
    1    0    true     false 
    1    1    false    false 

>> T_FullSubtractor

T_FullSubtractor =

  8x5 table

    A    B    BIN    DIFF     BORROW
    _    _    ___    _____    ______

    0    0     0     false    false 
    0    1     0     true     true  
    1    0     0     true     false 
    1    1     0     false    false 
    0    0     1     true     true  
    0    1     1     false    true  
    1    0     1     false    false 
    1    1     1     true     true  
```

Conclusion : In this experiment, we designed and implemented 2x1, 4x1, and 8x1 multiplexers using VHDL. By simulating these circuits, we verified their functionality. Each multiplexer correctly routed the selected input to the output based on the selection inputs, demonstrating the fundamental operation of multiplexers in digital circuits. This experiment provided insight into how multiplexers can be used for data routing, signal selection, and implementing logical functions in more complex digital systems. Understanding these principles is essential for designing efficient digital systems and applications.

Application of 2x1 Multiplexer :

1. Data Selection: Selects between two data inputs to be sent to a single output line.
2. Signal Routing: Used in digital communication systems to route signals from multiple sources to a single destination.

Application of 4 x1 Multiplexer :

1. Data Path Multiplexing: Used in processors to select data from different registers or memory locations for a specific operation.
2. Control Circuits: Used in control units to select control signals based on the current state of the system.

Application of 8 x 1 Multiplexer :

1. Data Selector: Chooses one of eight input data lines to route to a single output line.
2. Function Generator: Selects different logic functions based on control signals in digital processors.