Visvesvaraya National Institute of Technology (VNIT), Nagpur

Digital Hardware Design (ECP313)

Lab Report

Submitted by :
Rajesh Nagula (BT18ECE059)
Shruti Murarka (BT18ECE099)
Dhruvi Shah (BT18ECE115)
Semester VI

Group No (9)

Submitted to :
Dr. Anamika Singh
(Lab Instructor)
Department of Electronics and Communication Engineering,
VNIT Nagpur

Contents

Experiment 1 - Introduction to VHDL (15-01-2021)

1.1 Theory: VHDL stands for very high-speed integrated circuit hardware description language. VHDL stands for very high-speed integrated circuit hardware description language. It is a programming language used to model a digital system. In this experiment, a basic code consisting of all gates is implemented was aimed to teach entity declaration, syntax & architecture.

1.2 Codes:

```
--first VHDL coding class
--adding libraries to the code

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--Defining the entity

entity all_gates is
Port(
a: in STD_LOGIC;
b: in STD_LOGIC;
c: out STD_LOGIC;
c1: out STD_LOGIC;
c2: out STD_LOGIC;
c3: out STD_LOGIC;
c4: out STD_LOGIC;
c5: out STD_LOGIC;
c6: out STD_LOGIC
);
end all_gates;

--Defining THE Functionality
architecture Behavioral of all_gates is
begin
c<= a and b;
c1<= a or b;
c2<= a nand b;
c3<= a nor b;
```

```
c4<= a xor b;
c5<= a xnor b;
c6<= not b;
end Behavioral;
```

1.3 RTL Schematic: Quartus II is used for RTL View of –

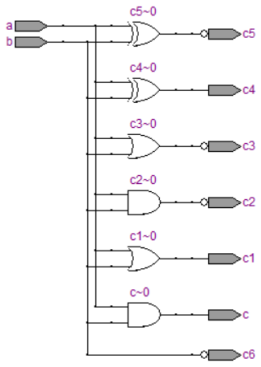

Figure 1: RTL View of All gates

1.4 Simulation Waverform: ModelSim is used for Simulation of –

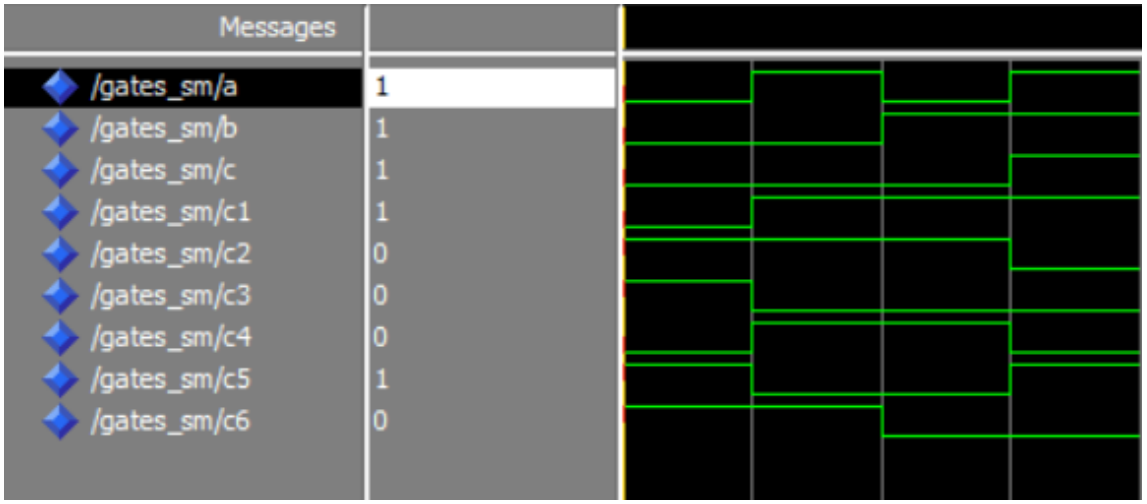

Figure 2: Simulation of All gates.

1.5 Conclusion: All gates has been designed and implemented using Quartus and Modelsim.

Experiment 2 - Data Flow Modelling, Behavioral Modelling, Structural Modelling (21-01-2021)

2.1 Theory: A digital system can be represented at different levels of abstraction. This keeps the description and design of complex systems manageable.

- Data flow modeling can be described based on the Boolean expression. It shows how the data flows from input to output.
- Behavioral modeling is used to execute statements sequentially. It shows that how the system performs according to the current statement.
- Structural modeling is used to specify the functionality and structure of the circuit.

In this experiment, Half Adder is implemented using three models.

2.2 Codes:

Dataflow Model

```
--Half adder usind Dataflow modeling
library ieee;
use ieee.std_logic_1164.all;

entity HA_dataflow is
port(a ,b: in std_logic;
sum , carry_out: out std_logic);
end HA_dataflow;

architecture dataflow of HA_dataflow is
begin
sum <= a xor b;
carry_out <= a and b;
end dataflow;
```

Behavioral Model

```
--Half Adder using Behavioral  modeling

library ieee;
use ieee.std_logic_1164.all;
```

```
entity HA_behave is
port( a,b : in std_logic; s, c : out std_logic);
end HA_behave;

architecture behaviour of HA_behave is
begin
ha : process(a,b)

begin
if a = '1' then
s <= not b;
c <= b;
else
s <= b;
c <= '0';
end if;

end process ha;
end behaviour;
```

Structural Model

```
-- Half adder using Strutural modeling approach
library ieee;
use ieee.std_logic_1164.all;

entity andgate is
port(a, b: in std_logic;
z: out std_logic);
end andgate;

architecture e1  of andgate is
begin
z<= a and b;
end e1;

library ieee;
use ieee.std_logic_1164.all;

entity xorgate is
port(a, b: in std_logic;
```

```
z: out std_logic);
end xorgate;

architecture e2 of xorgate is
begin
z <= a xor b;
end e2;

library ieee;
use ieee.std_logic_1164.all;

entity HA_Structural is
port(a, b : in std_logic; s, c: out std_logic);
end HA_Structural;

architecture structural of HA_Structural is
component andgate
port(a, b: in std_logic; z: out std_logic);
end component;

component xorgate
port(a, b: in std_logic; z: out std_logic);
end component;

begin
u1: andgate port map(a, b, c);
u2: xorgate port map(a, b, s);

end structural;
```

2.3 Testbench:

```
library ieee;
use ieee.std_logic_1164.all;

entity tb_ha_struct is
end tb_ha_struct;

architecture tb of tb_ha_struct is

component HA_Structural
```

```
port (a : in std_logic;
b : in std_logic;
s, c: out std_logic);
end component;

signal a : std_logic;
signal b : std_logic;
signal s : std_logic;
signal c : std_logic;
begin

dut : HA_Structural
port map (a => a,
b => b,
s => s,
c => c);

stimuli : process
begin
a <= '0';
b <= '0';
wait for 100ps;

a<='0';
b<='1';
wait for 100ps;

a<='1';
b<='0';
wait for 100ps;

a<='1';
b<='1';
wait for 100ps;
wait;
end process;

end tb;
```

2.4 RTL Schematic: Quartus II is used for RTL View of –

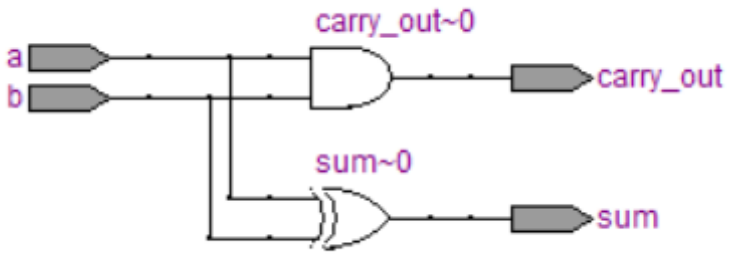

Figure 3: RTL View of Half adder : Data-flow model

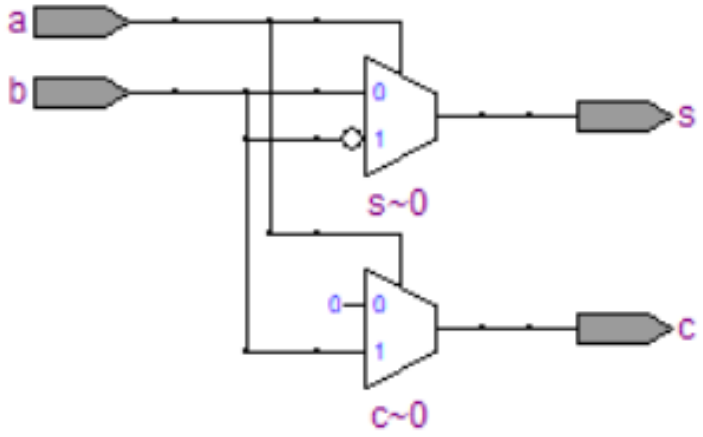

Figure 4: RTL View of Half adder : Behavioural model

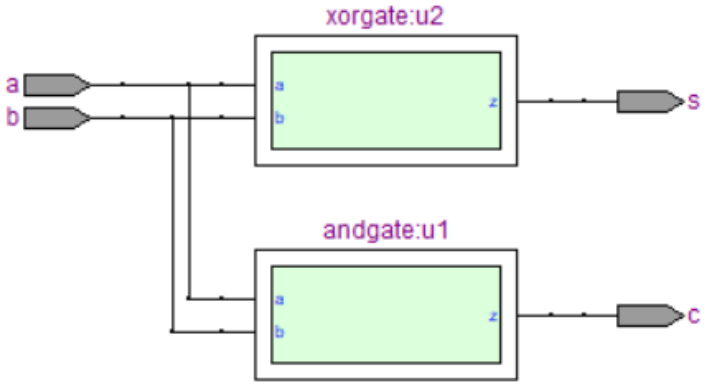

Figure 5: RTL View of Half adder : Structural model

2.5 Simulation Waverform: ModelSim is used for Simulation of –

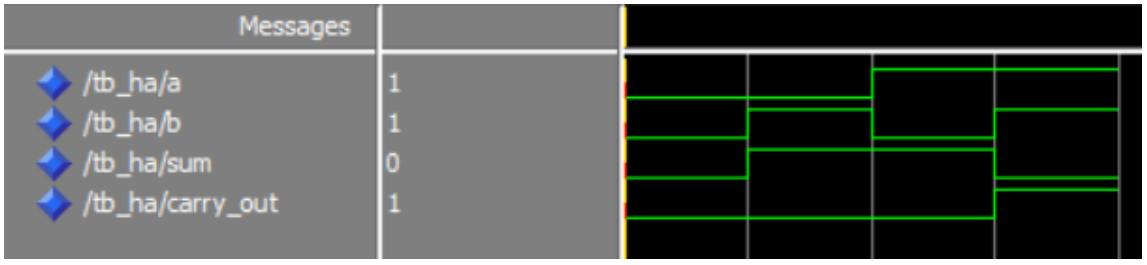

Figure 6: Simulation of HA Dataflow Model.

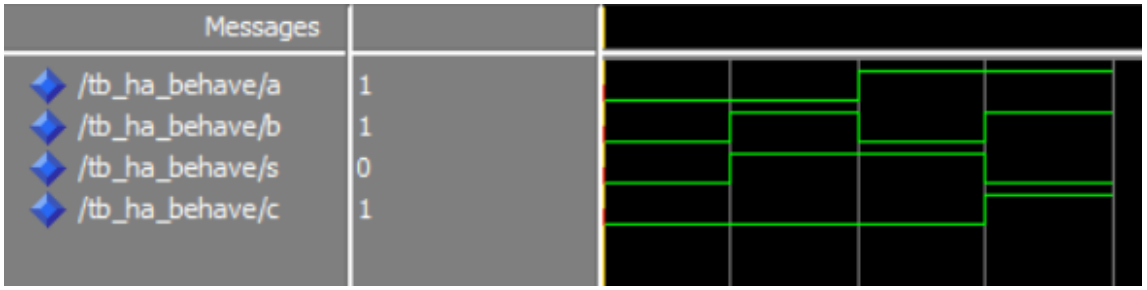

Figure 7: Simulation of HA Behavioral Model.

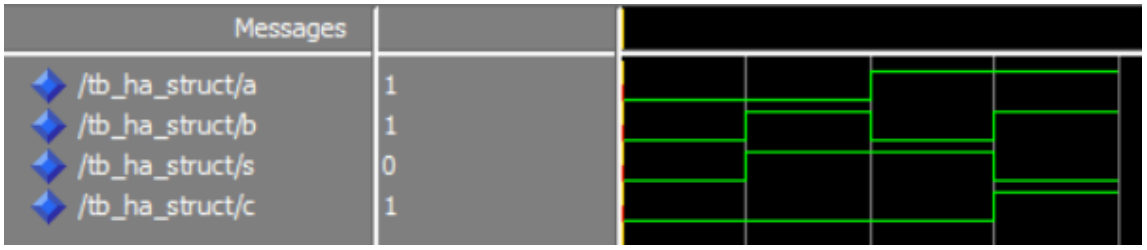

Figure 8: Simulation of HA Structural Model.

2.6 Conclusion: Dataflow, Behavioral, Structural Models for Half Adder along with their testbench has been designed and implemented using Quartus and Modelsim.

Experiment 3 - Concurrent and Sequential Statements (28-01-2021)

3.1 Theory: A VHDL description has two domains: a sequential domain and a concurrent domain. The sequential domain is represented by a process or subprogram that contains sequential statements. These statements are executed in the order in which they appear within the process or subprogram, as in programming languages. The concurrent domain is represented by an architecture that contains processes, concurrent procedure calls, concurrent signal assignments, and component instantiations.
In this experiment, various sequential and concurrent statements are implemented for MUX & Full Adder.

3.2 Codes:

Concurrent: WITH SELECT

```
library IEEE;
use IEEE.std_logic_1164.all;
entity Mux4to1_withselect is
port(
d0 : in std_logic;
d1 : in std_logic;
d2 : in std_logic;
d3 : in std_logic;
s : in std_logic_vector(1 downto 0);
y : out std_logic
);
end Mux4to1_withselect;
architecture m1 of Mux4to1_withselect is begin
 with s select
 y <= d0 when "00",
 d1 when "01",
 d2 when "10",
 d3 when "11",
 '0' when others;
end m1;
```

Concurrent: WHEN ELSE

```
library ieee;
use ieee.std_logic_1164.all;

--4:1 mux using when-else statement
entity Mux4to1_whenelse is
port (D: in std_logic_vector(0 to 3);
S: in std_logic_vector (0 to 1);
O: out std_logic);
end Mux4to1_whenelse;

architecture m1 of Mux4to1_whenelse is
begin
O <= D(0) when S="00" else
D(1) when S="01" else
D(2) when S="10" else
D(3) when S="11" else
'0';
end m1;
```

Sequential: IF THEN ELSE

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Mux4to1_ifelse is
port(
        A, B, C, D: in STD_LOGIC;
        S0, S1: in STD_LOGIC;
        Z: out STD_LOGIC
);
end Mux4to1_ifelse;

architecture bhv of Mux4to1_ifelse is
begin
process (A, B, C, D, S0, S1) is
begin
if (S0 = '1' and S1 = '0') then
        z <= A;
elsif (S0 = '1' and S1 = '0') then
```

```
        z <= B;
elsif (S0 = '0' and S1 = '1') then
        z <= C;
else
        z <= D;
end if;
end process;
end bhv;
```

Sequential: CASE

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity Mux4to1_case is
     port(
         data_in : in STD_LOGIC_VECTOR(3 downto 0);
         sel : in STD_LOGIC_VECTOR(1 downto 0);
         data_out : out STD_LOGIC
         );
end Mux4to1_case;
architecture m1 of Mux4to1_case is
begin
    mux : process (data_in,sel) is
    begin
        case sel is
            when "00" => data_out <= data_in(0);
            when "01" => data_out <= data_in(1);
            when "10" => data_out <= data_in(2);
            when others => data_out <= data_in(3);
        end case;
    end process mux;
end m1;
```

Concurrent: FOR-GENERATE

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

--2 bit full adder
entity FA_ForLoop is
```

```
port(
  a, b, c: in  std_logic_vector(1 downto 0);
  sum, carry: out std_logic_vector(1 downto 0));
end  FA_ForLoop;
architecture fl1 of  FA_ForLoop is
begin
    gen1: for i in 0 to 1 generate
        sum(i)<= a(i) xor b(i) xor c(i);
        carry(i)<=(a(i) and b(i)) or (b(i) and c(i)) or (c(i) and a(i));
    end generate gen1;
end fl1;
```

3.3 Testbench:

4:1 MUX

```
library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_Std.all;

entity Mux4to1_case_tb is
end;

architecture bench of Mux4to1_case_tb is

component Mux4to1_case
        port(
        data_in : in STD_LOGIC_VECTOR(3 downto 0);
        sel : in STD_LOGIC_VECTOR(1 downto 0);
        data_out : out STD_LOGIC
        );
end component;

signal data_in: STD_LOGIC_VECTOR(3 downto 0);
signal sel: STD_LOGIC_VECTOR(1 downto 0);
signal data_out: STD_LOGIC ;

begin

uut: Mux4to1_case port map ( data_in => data_in,
sel => sel,
```

```
data_out => data_out );

stimulus: process
begin

data_in<="1110";
sel<="00";
wait for 100ps;

data_in<="0001";
sel<="00";
wait for 100ps;

data_in<="1101";
sel<="01";
wait for 100ps;

data_in<="0010";
sel<="01";
wait for 100ps;

data_in<="1011";
sel<="10";
wait for 100ps;

data_in<="0100";
sel<="10";
wait for 100ps;

data_in<="0111";
sel<="11";
wait for 100ps;

data_in<="1000";
sel<="11";
wait for 100ps;
end process;
end;
```

Full Adder

```
library ieee;
use ieee.std_logic_1164.all;

entity tb_fa is
end tb_fa;

architecture tb of tb_fa is

component FA_ForLoop
port (a, b, c: in  std_logic_vector(1 downto 0);
  sum, carry: out std_logic_vector(1 downto 0));
end component;

signal a, b, c : std_logic_vector(1 downto 0);
signal sum, carry : std_logic_vector(1 downto 0);

begin

dut : FA_ForLoop
port map (a => a,
b => b,
c => c,
sum => sum,
carry => carry);

stimuli : process
begin
a <= "00";
b <= "01";
c <= "00";
wait for 100ps;

a <= "10";
b <= "01";
c <= "10";
wait for 100ps;

a <= "01";
b <= "00";
```

```
c <= "01";
wait for 100ps;

a <= "11";
b <= "11";
c <= "01";
wait for 100ps;
wait;
wait;
end process;

end tb;
```

3.4 RTL Schematic: Quartus II is used for RTL View of –

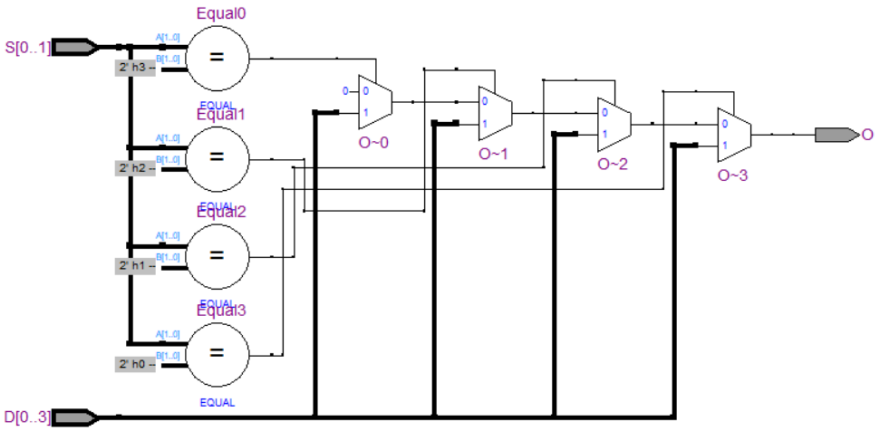

Figure 9: RTL View of 4:1 MUX - When else

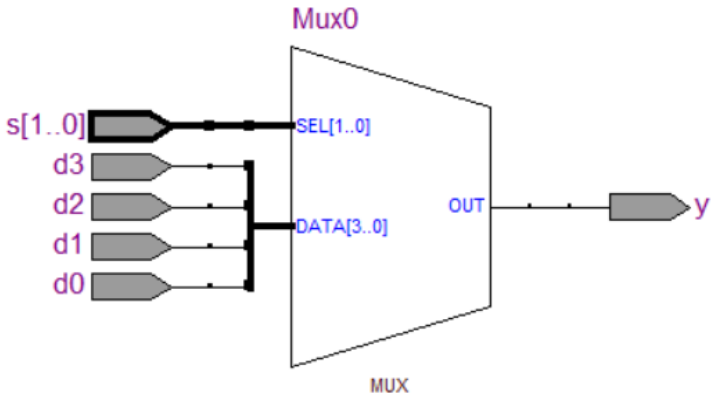

Figure 10: RTL View of 4:1 MUX - With select

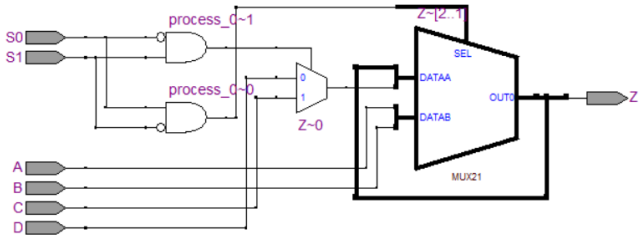

Figure 11: RTL View of 4:1 MUX - If Then Else

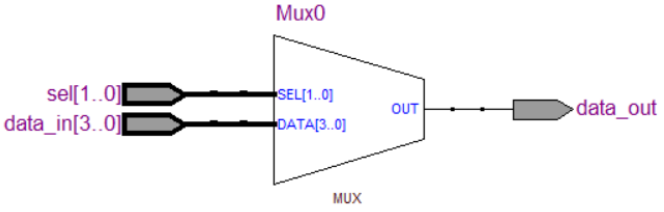

Figure 12: RTL View of 4:1 MUX - CASE

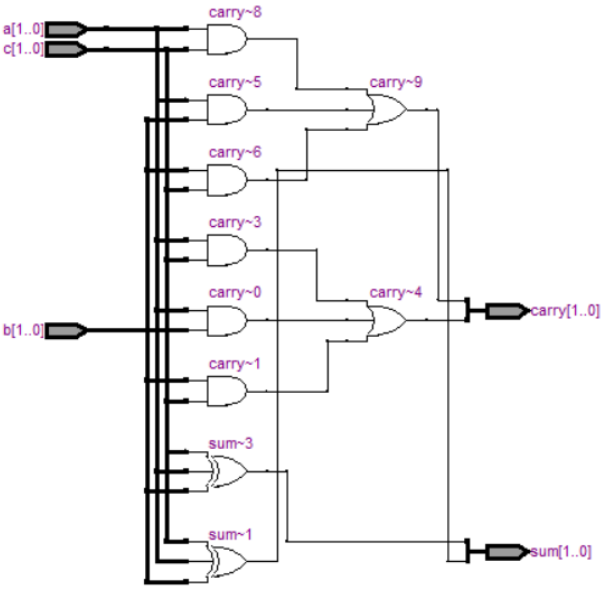

Figure 13: RTL View of Full Adder - For

3.5 Simulation Waverform: ModelSim is used for Simulation of –

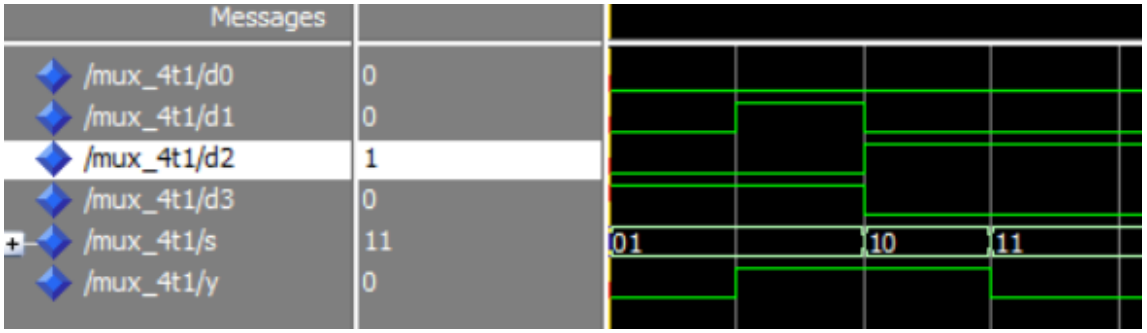

Figure 14: Simulation of 4:1 MUX - WITH SELECT.

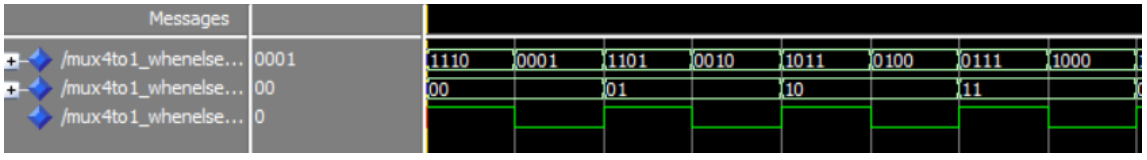

Figure 15: Simulation of 4:1 MUX - WHEN ELSE.

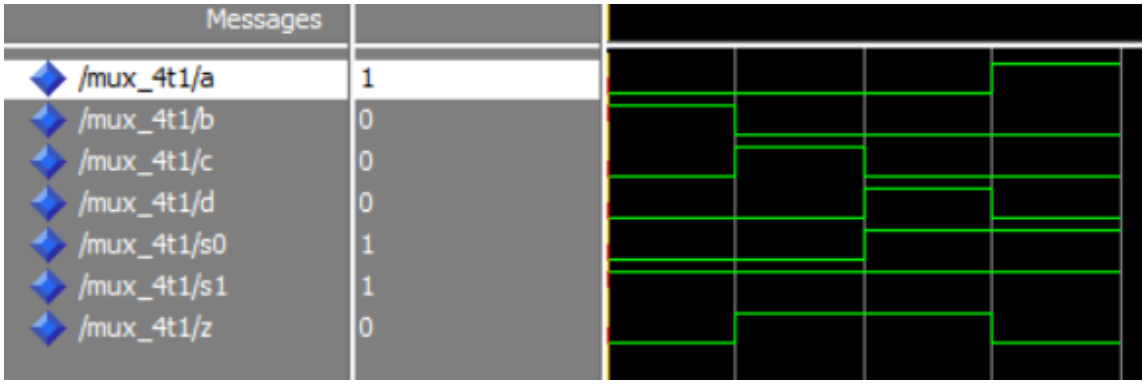

Figure 16: Simulation of 4:1 MUX - IF THEN ELSE.

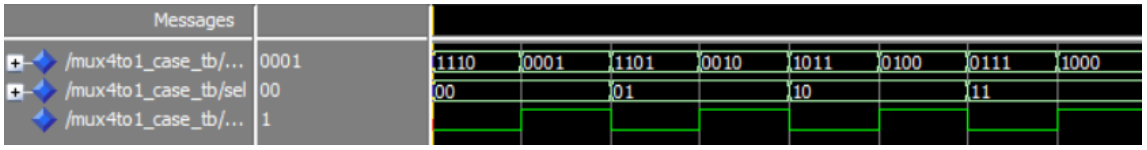

Figure 17: Simulation of 4:1 MUX - CASE.

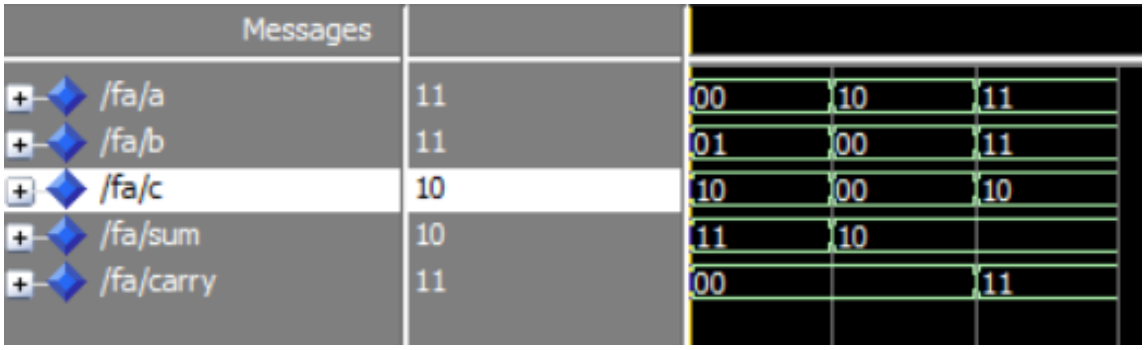

Figure 18: Simulation of Full Adder - For.

3.6 Conclusion: Various Sequential and concurrent statements for 4:1 MUX & Full Adder has been designed and implemented using Quartus and Modelsim.

Experiment 4 - Registers and Counters (11-02-2021)

4.1 Theory: Circuits that include flip-flops are usually classified by the function they perform.Register is a group of flip-flops.Each flip-flop is capable of storing one bit of information.An n-bit register consists of a group of n flip-flops.Register is a group of binary cells suitable for holding binary information.A counter is essentially a register that goes through a predetermined sequence of states.Synchronous digital systems have a master clock generator that supplies a continuous train of clock pulses.

4.2 Codes:

Shift Register with Asynchronous Reset

```
--4 bit shift register with async reset
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity sipo_asyncreset is
port(
    clk, clear : in std_logic;
    Input_Data: in std_logic;
    Q: buffer std_logic_vector(3 downto 0) );
end sipo_asyncreset;
architecture arch of sipo_asyncreset is begin
process (clk, clear)
begin
    if clear = '1' then
        Q <= "0000";
    elsif (CLK'event and CLK='1') then
        Q(3 downto 1) <= Q(2 downto 0);
        Q(0) <= Input_Data;
    end if;
end process;
end arch;
```

Shift Register with Synchronous Reset

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

```
entity sipo_syncreset is
port(
clk, clear : in std_logic;
Input_Data: in std_logic;
Q: buffer std_logic_vector(7 downto 0) );
end sipo_syncreset;

architecture arch of sipo_syncreset is
begin

process (clk)
begin
    if (CLK'event and CLK='1') then
        if clear = '1' then
            Q <= "00000000";
        elsif (clear='0') then
            Q(7 downto 1) <= Q(6 downto 0);
            Q(0) <= Input_Data;
        end if;
    end if;
end process;
end arch;
```

Up Counter using Asynchronous Clear

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity counter is
port ( clock, resetn, E: in std_logic;
Q: out integer range 0 to 255);
end counter;

architecture bhv of counter is
signal Qt: integer range 0 to 255;
begin
process (resetn,clock, E)
    begin
```

```
    if resetn = '1' then
        Qt <= 12;
    elsif (clock'event and clock='1') then
        if E = '1' then
            if Qt<255 then --line nos 50-53 to turn Qt=0 when it reaches 255
                    Qt <= Qt + 1;
            else Qt<=0;
            end if;
        end if;
    end if;
    end process;
Q <= Qt;
end bhv;
```

Up Counter with Synchronous Clear

```
--8-bit unsigned up counter with synchronous clear
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
entity counter is
port
(
    clk : in std_logic;
    reset : in std_logic;
    enable : in std_logic;
    q : out integer range 0 to 255
);
end entity;
architecture rtl of counter is
begin
process (clk)
    variable cnt : integer range 0 to 255;
    begin
        if (clk='1') then
            if reset = '1' then
            -- Reset the counter to 12
                cnt := 12;
            elsif enable = '1' then
            -- Increment the counter if counting is enabled
```

```
                cnt := cnt + 1;
            end if;
        end if;
    -- Output the current count
    q <= cnt;
end process;
end rtl;
```

4.3 Testbench:

Shift Register with synchronous Reset

```
library ieee;
use ieee.std_logic_1164.all;

entity sipo_syncreset_tb is
end sipo_syncreset_tb;

architecture tb of sipo_syncreset_tb is

 component sipo_syncreset
 port (clk, clear, input_data: in std_logic;
 Q: buffer std_logic_vector(7 downto 0) );
 end component;

 signal clk, clear: std_logic:='1';
 signal input_data: std_logic:='1';
 signal Q: std_logic_vector(7 downto 0):="00000000";
begin
 uut : sipo_syncreset
 port map (clk => clk, clear => clear, Input_Data => Input_Data,  Q=>Q);
 clock : process
 begin
 clear<='0';   --1000 1101 is the sequence  Input_Data <= '1';
 clk<='1';
 wait for 50 ps;
 clk<='0';
 wait for 50 ps;

 Input_Data <= '0';
 clk<='1';
```

```
wait for 50 ps;
clk<='0';
wait for 50 ps;

Input_Data <= '0';
clk<='1';
wait for 50 ps;
clk<='0';
wait for 50 ps;

Input_Data <= '0';
clk<='1';
wait for 50 ps;
clk<='0';
wait for 50 ps;

Input_Data <= '1';
clk<='1';
wait for 50 ps;
clk<='0';
wait for 50 ps;

Input_Data <= '1';
clk<='1';
wait for 50 ps;
clk<='0';
wait for 50 ps;

Input_Data <= '0';
clk<='1';
wait for 50 ps;
clk<='0';
wait for 50 ps;

Input_Data <= '1';
clk<='1';
wait for 50 ps;
clk<='0';
wait for 50 ps;
```

```
end process;
end tb;
```

Up Counter with Asynchronous Reset

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity upcounter_tb is
end entity;

architecture tb of upcounter_tb is
component counter is
Port ( clock,resetn,E : in STD_LOGIC;
Q : out integer range 0 to 255);
end component;

signal clock,resetn,E : STD_LOGIC := '1';
signal Qt : integer range 0 to 255;
signal Q : integer range 0 to 255;

begin

uut: counter port map(
clock => clock,
resetn => resetn,
E => E,
Q=>Q);

clk: process
begin
resetn <= '0';
clock <= '1';
wait for 50 ps;
clock <= '0';
wait for 50 ps;
end process;
end tb;
```

4.4 RTL Schematic: Quartus II is used for RTL View of –

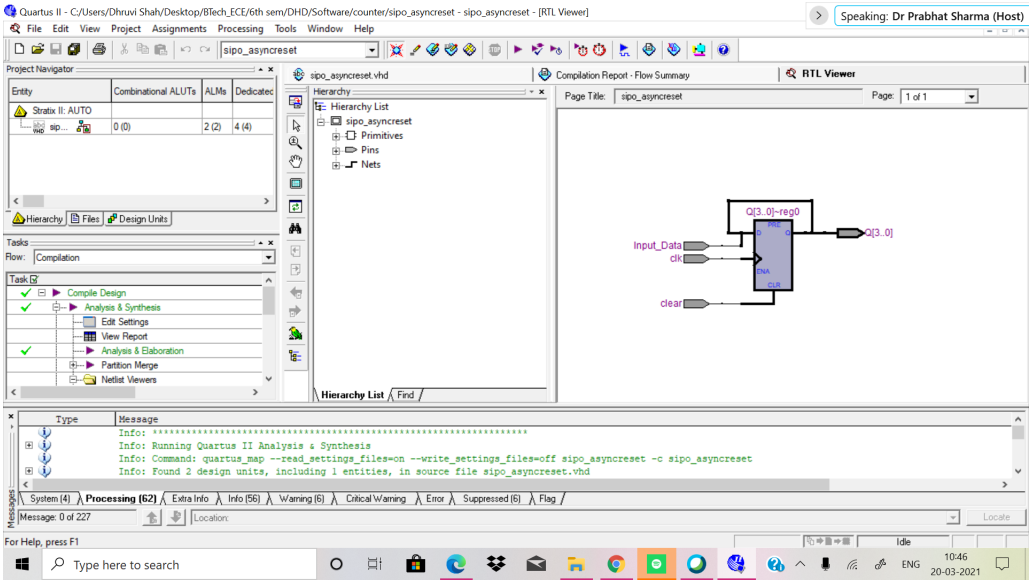

Figure 19: RTL View of 4 bit shift register with async reset

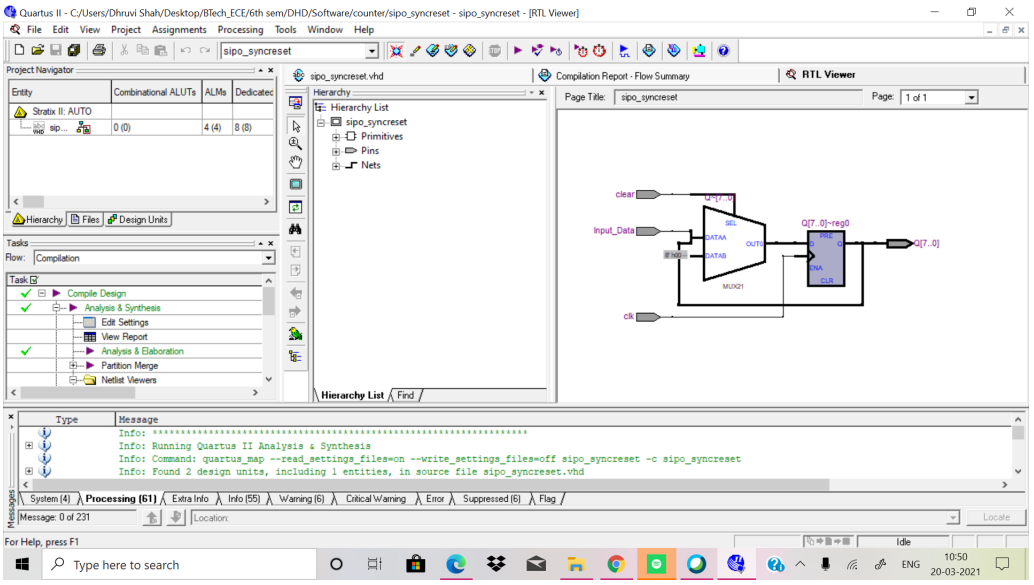

Figure 20: RTL View of 4 bit shift register with sync reset

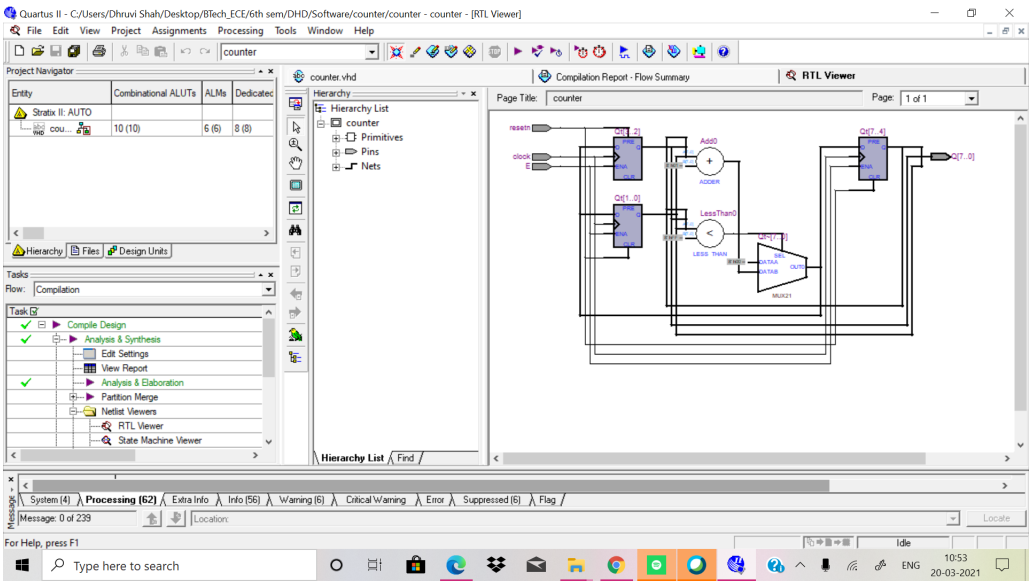

Figure 21: RTL View of up counter with asynchronous clear.

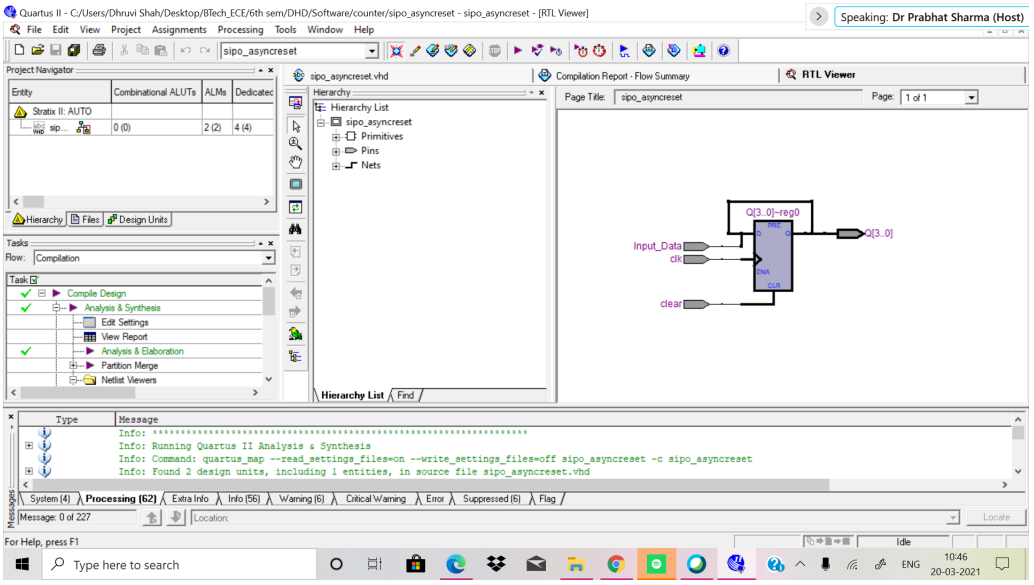

Figure 22: RTL View of 8-bit unsigned up counter with synchronous clear

4.5 <u>Simulation Waverform</u>: ModelSim is used for Simulation of –

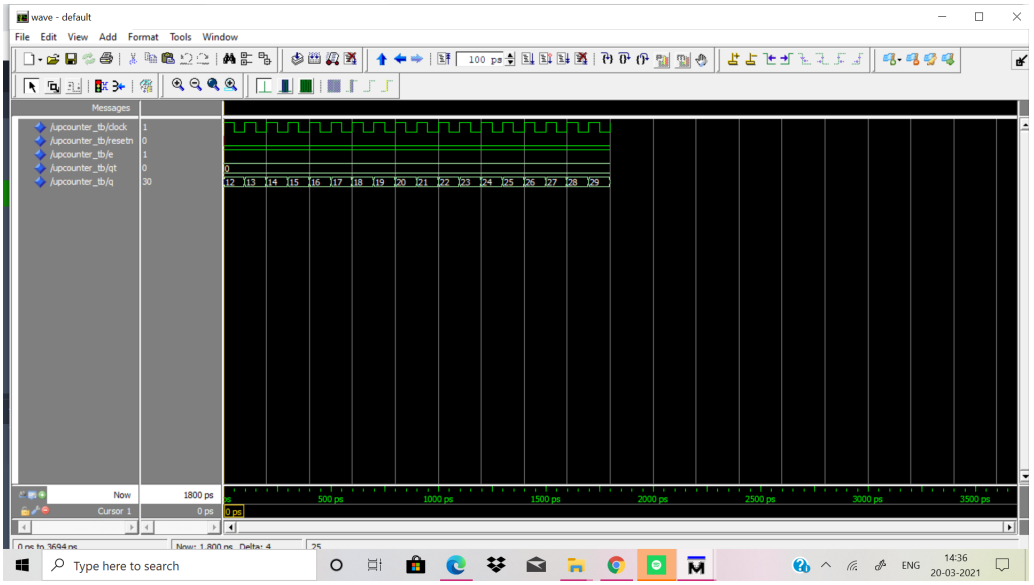

Figure 23: Simulation of upcounter with asynchronous reset.

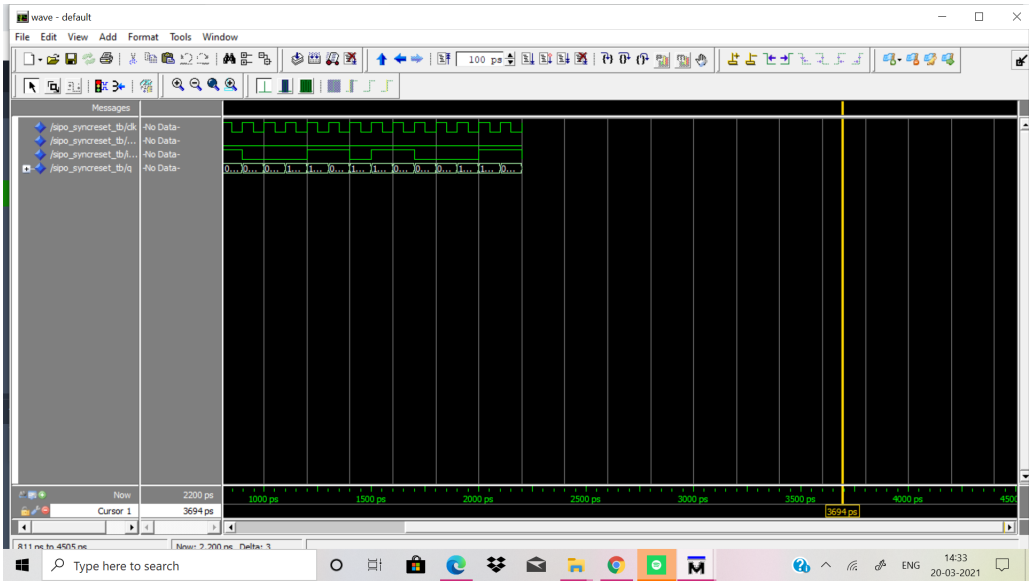

Figure 24: Simulation of Shift register with synchronous reset.

4.6 Conclusion: Shift register with synchronous and asynchronous clear , up-counter with asynchronous clear has been designed and implemented using Quartus and modelsim.

Experiment 5 - Finite State Machine (18-02-2021)

5.1 Theory: A Finite State Machine, or FSM, is a computation model that can be used to simulate sequential logic, or, in other words, to represent and control execution flow. Finite State Machines can be used to model problems in many fields, including mathematics, artificial intelligence, games or linguistics.

5.2 Codes:

Moore VHDL for detecting "11"

```
--moore code
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity moore_11 is
Port ( clk : in STD_LOGIC;
din : in STD_LOGIC;
rst : in STD_LOGIC;
dout : out STD_LOGIC);
end moore_11;

architecture Behavioral of moore_11 is
type state is (st0, st1, st2);
signal present_state, next_state : state;
begin

synchronous_process: process (clk)
begin
if rising_edge(clk) then
if (rst = '1') then
present_state <= st0;
else
present_state <= next_state;
end if;
end if;
end process;

output_decoder : process(present_state, din)
begin
next_state <= st0;
```

5.2.0

```
case (present_state) is
when st0 =>
if (din = '1') then
next_state <= st1;
else
next_state <= st0;
end if;
when st1 =>
if (din = '1') then
next_state <= st2;
else
next_state <= st0;
end if;
when st2 =>
if (din = '1') then
next_state <= st2;
else
next_state <= st0;
end if;
when others =>
next_state <= st0;
end case;
end process;

next_state_decoder : process(present_state)
begin case (present_state) is
when st0 =>
dout <= '0';
when st1 =>
dout <= '0';
when st2 =>
dout <= '1';
when others =>
dout <= '0';
end case;
end process;

end Behavioral;
```

Mealy VHDL for detectiong "11"

```
--mealy code

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity mealy_11 is
  port(clk, din, rst: in std_logic;
  dout: out std_logic);
end mealy_11;
architecture behav of mealy_11 is
type state is (st0, st1);
signal present_state, next_state: state;
begin
  sync_process: process(clk)
  begin
    if rising_edge(clk) then
      if rst='1' then
        present_state<=st0;
      else
        present_state<=next_state;
      end if;
    end if;
  end process;

  state_output_process: process(present_state, din)
  begin
    dout<='0';
    case(present_state) is
  when st0=>
    if din='0' then
      next_state<=st0;
      dout<='0';
    else
      next_state<=st1;
      dout<='0';
    end if;
  when st1=>
    if din='0' then
      next_state<=st0;
      dout<='0';
```

```
    else
      next_state<=st1;
      dout<='1';
    end if;
  end case;
end process;
end behav;
```

5.3 Testbench:

Moore Testbench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY tb_moore_11 IS
END tb_moore_11;
ARCHITECTURE behavior OF tb_moore_11 IS
--Component Declaration for the Unit Under Test (UUT)
COMPONENT moore_11
PORT(
clk : IN std_logic;
din : IN std_logic;
rst : IN std_logic;
dout : OUT std_logic
);
END COMPONENT;
--Inputs
signal clk : std_logic := '0';
signal din : std_logic := '0';
signal rst : std_logic := '0';
--Outputssignal dout : std_logic;
--Clock period definitions
constant clk_period : time := 100 ps;
BEGIN
--Instantiate the Unit Under Test (UUT)
uut: moore_11 PORT MAP (
clk => clk,
din => din,
rst => rst,
dout => dout
);
```

```
--Clock process definitions
clk_process :process
begin
clk <= '1';
wait for clk_period/2;
clk <= '0';
wait for clk_period/2;
end process;
--Stimulus process
stim_proc: process
begin
rst <= '1';wait for 100 ps;
rst <= '0';
din <= '0';
wait for 100 ps;
din <= '1';
wait for 100 ps;
din <= '0';
wait for 100 ps;
din <= '1';
wait for 100 ps;
din <= '1';
wait for 100 ps;
din <= '0';
wait for 100 ps;
din <= '1';
wait for 100 ps;
din <= '1';
wait for 100 ps;
din <= '1';
wait for 100 ps;
din <= '1';wait for 100 ps;
din <= '0';
wait for 100 ps;
din <= '1';
end process;
END;
```

Mealy Testbench

```
-- testbench for mealy
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

5.3.0

```
ENTITY tb_mealy_11 IS
END tb_mealy_11;
ARCHITECTURE behavior OF tb_mealy_11 IS
--Component Declaration for the Unit Under Test (UUT)
COMPONENT mealy_11
PORT(
clk : IN std_logic;
din : IN std_logic;
rst : IN std_logic;
dout : OUT std_logic
);
END COMPONENT;
--Inputs
signal clk : std_logic := '0';
signal din : std_logic := '0';
signal rst : std_logic := '0';
--Outputs
signal dout : std_logic;
--Clock period definitions
constant clk_period : time := 100 ps;
BEGIN
--Instantiate the Unit Under Test (UUT)
uut: mealy_11 PORT MAP (
clk => clk,
din => din,
rst => rst,
dout => dout
);
--Clock process definitions
clk_process :process
beginclk <= '1';
wait for clk_period/2;
clk <= '0';
wait for clk_period/2;
end process;
--Stimulus process
stim_proc: process
begin
rst <= '1';
wait for 100 ps;
```

```
rst <= '0';
din <= '0';
wait for 100 ps;
din <= '1';
wait for 100 ps;
din <= '0';
wait for 100 ps;
din <= '1';
wait for 100 ps;
din <= '1';
wait for 100 ps;
din <= '0';
wait for 100 ps;
din <= '1';
wait for 100 ps;
din <= '1';
wait for 100 ps;
din <= '1';
wait for 100 ps;
din <= '0';
wait for 100 ps;
din <= '1';
wait for 100 ps;
end process;
END;
```

5.4 RTL Schematic: Quartus II is used for RTL View of –

Moore RTL

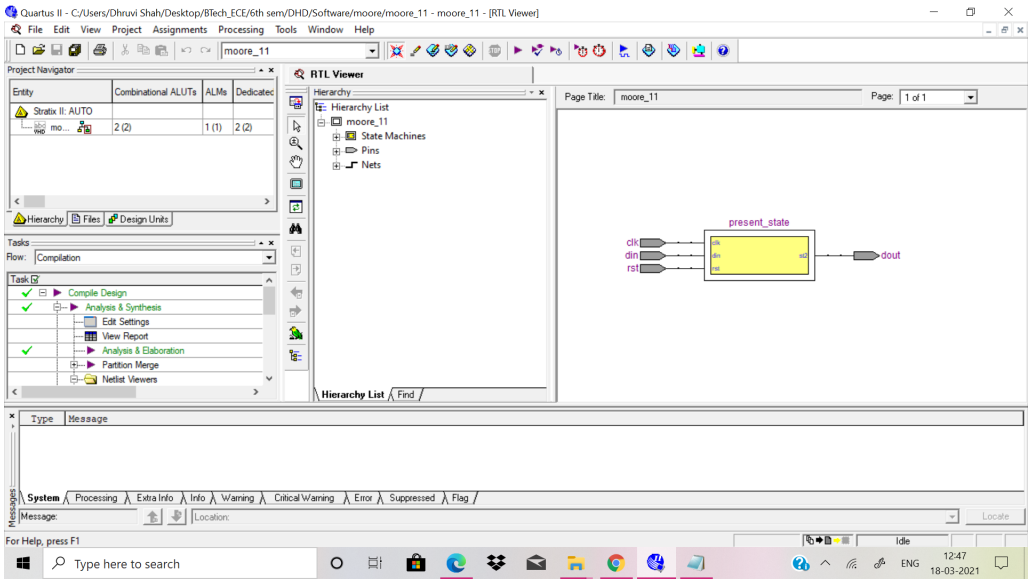

Figure 25: RTL View of sequence detector("11") using Moore approch

Mealy RTL

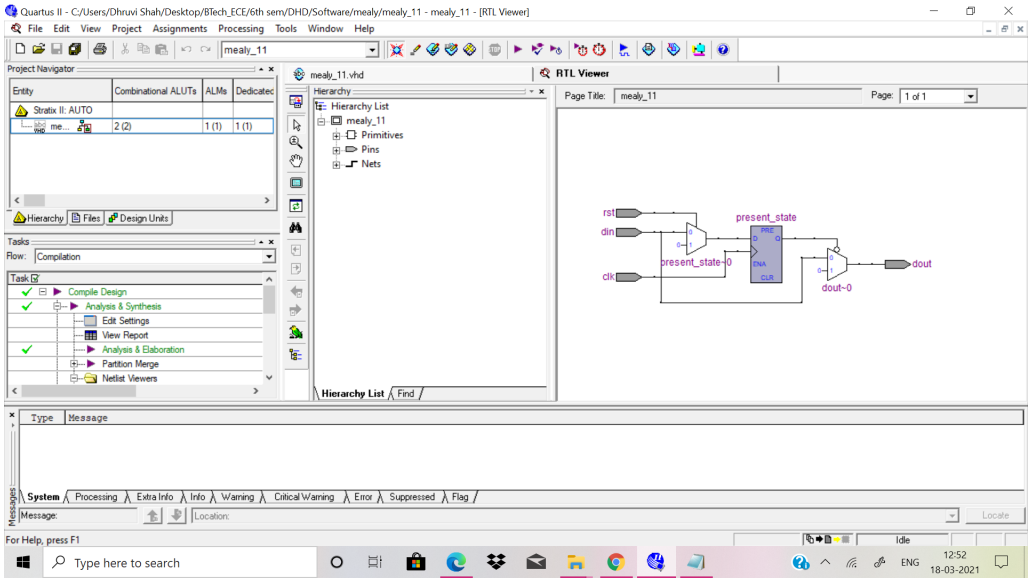

Figure 26: RTL View of sequence detector("11") using mealy approach

5.5 Simulation Waverform: ModelSim is used for Simulation of –

Moore

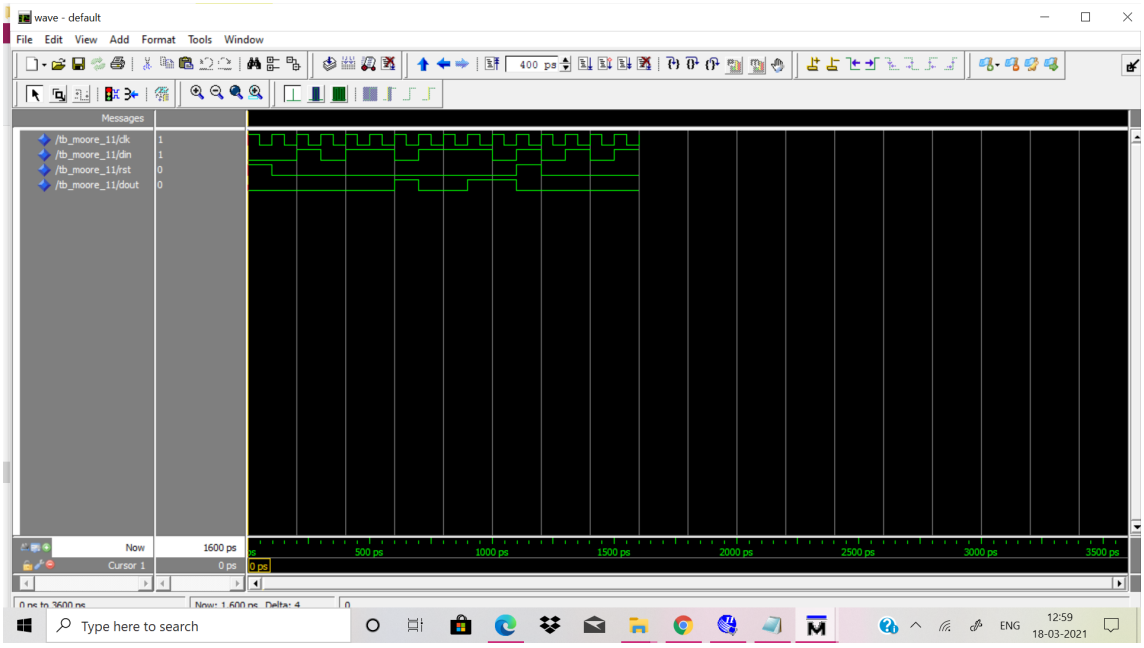

Figure 27: Simulation of sequence detector(”11”) using moore approach.

Mealy

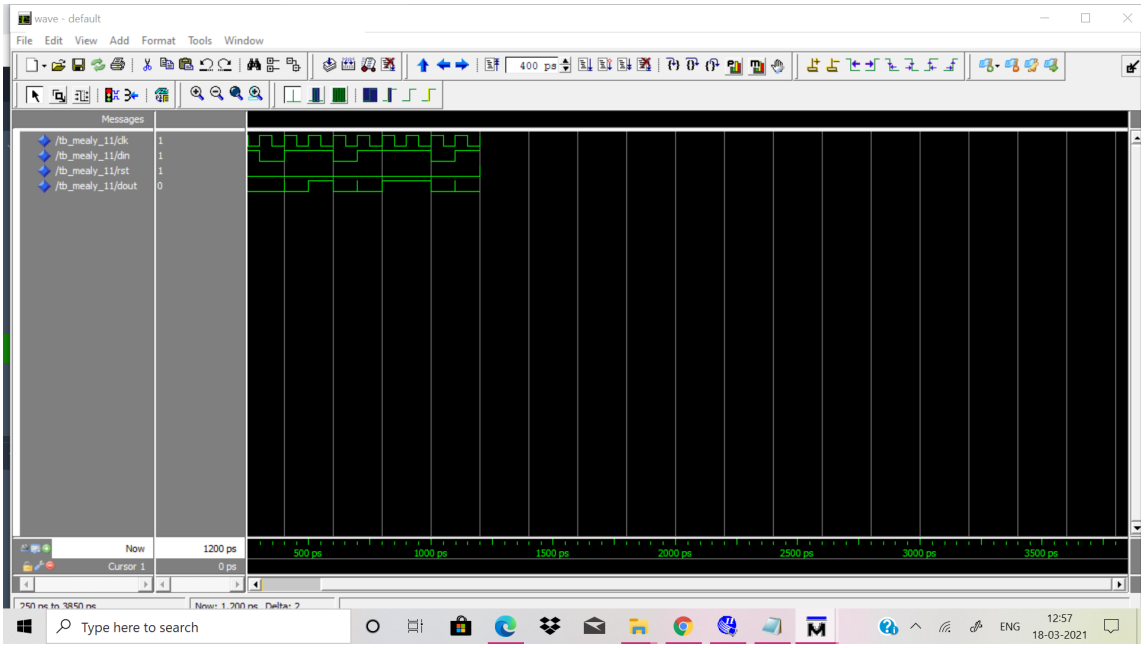

Figure 28: Simulation of sequence detector("11") using mealy approach.

5.6 Conclusion: The advantages of Finite State Machine include the following.

- Finite state machines are flexible.
- Easy to move from a significant abstract to a code execution.
- Low processor overhead.
- Easy determination of reachability of a state.

Experiment 6 - Configuration and Packages (25-02-2021)

6.1 Theory: In VHDL, components can be used in order to make other components, or they can be called in order to design more complex systems. This can be done by describing each component every time it is used, but this leads to extremely verbose and repetitive code. Instead, components are encapsulated into packages that can then be called by any other component at any time without a full description. In this experiment, we design a 4-bit full adder using a 1-bit full adder, which is declared and defined in a package.

6.2 Codes:

1 bit Full Adder

```
--fulladd.vhd (fulladd entity)
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY fulladd IS
PORT (Cin, x, y: IN STD_LOGIC ;
s, Cout: OUT STD_LOGIC ) ;
END fulladd ;
ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
s <= x XOR y XOR Cin ;
Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y) ;
END LogicFunc ;
```

Full Adder Package

```
--fulladd_package.vhd (package file)
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
PACKAGE fulladd_package IS
COMPONENT fulladd
PORT ( Cin, x, y: IN STD_LOGIC ;
s, Cout: OUT STD_LOGIC ) ;
END COMPONENT ;
END fulladd_package ;
```

4 bit Full Adder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;
ENTITY adder4 IS
PORT ( Cin : IN STD_LOGIC ;
x3, x2, x1, x0 : IN STD_LOGIC ;
y3, y2, y1, y0 : IN STD_LOGIC ;
s3, s2, s1, s0 : OUT STD_LOGIC ;
Cout : OUT STD_LOGIC ) ;
END adder4 ;
ARCHITECTURE Structure OF adder4 IS
SIGNAL c1, c2, c3 : STD_LOGIC ;
BEGIN
stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;
stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;
stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;
stage3: fulladd PORT MAP (
Cin => c3, Cout => Cout, x => x3, y => y3, s => s3 ) ;
END Structure ;
```

6.3 Testbench:

```
--TestBench for adder4
library ieee;
use ieee.std_logic_1164.all;
entity tb_adder4 is
end tb_adder4;
architecture tb of tb_adder4 is
component adder4
port (Cin : in std_logic;
x3 : in std_logic;
x2 : in std_logic;
x1 : in std_logic;
x0 : in std_logic;
y3 : in std_logic;
y2 : in std_logic;
y1 : in std_logic;
y0 : in std_logic;
s3 : out std_logic;
```

```
s2 : out std_logic;
s1 : out std_logic;
s0 : out std_logic;
Cout : out std_logic);
end component;
signal Cin : std_logic;
signal x3 : std_logic;
signal x2 : std_logic;
signal x1 : std_logic;
signal x0 : std_logic;
signal y3 : std_logic;
signal y2 : std_logic;
signal y1 : std_logic;
signal y0 : std_logic;
signal s3 : std_logic;
signal s2 : std_logic;
signal s1 : std_logic;
signal s0 : std_logic;
signal Cout : std_logic;

begin
dut : adder4
port map (Cin => Cin,
x3 => x3,
x2 => x2,
x1 => x1,
x0 => x0,
y3 => y3,
y2 => y2,
y1 => y1,
y0 => y0,
s3 => s3,
s2 => s2,
s1 => s1,
s0 => s0,
Cout => Cout);
stimuli : process
begin
Cin <= '0';
x3 <= '0';
```

```
x2 <= '0';
x1 <= '0';
x0 <= '0';
y3 <= '0';
y2 <= '0';
y1 <= '0';
y0 <= '0';
wait for 100 ps;
Cin <= '1';
x3 <= '0';
x2 <= '0';
x1 <= '0';
x0 <= '0';
y3 <= '0';
y2 <= '0';
y1 <= '0';
y0 <= '0';
wait for 100 ps;

Cin <= '0';
x3 <= '1';
x2 <= '0';
x1 <= '0';
x0 <= '0';
y3 <= '1';
y2 <= '0';
y1 <= '0';
y0 <= '0';
wait for 100 ps;
end process;
end tb;
-- Configuration block below is required by some simulators.
-- Usually no need to edit.
configuration cfg_tb_adder4 of tb_adder4 is
for tb
end for;
end cfg_tb_adder4;
```

6.4 RTL Schematic: Quartus Prime is used for RTL View

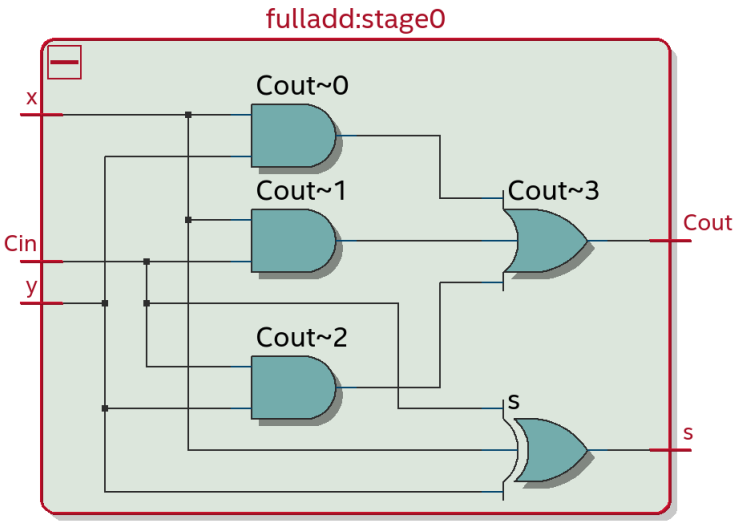

Figure 29: RTL View of 1 Bit Full Adder.

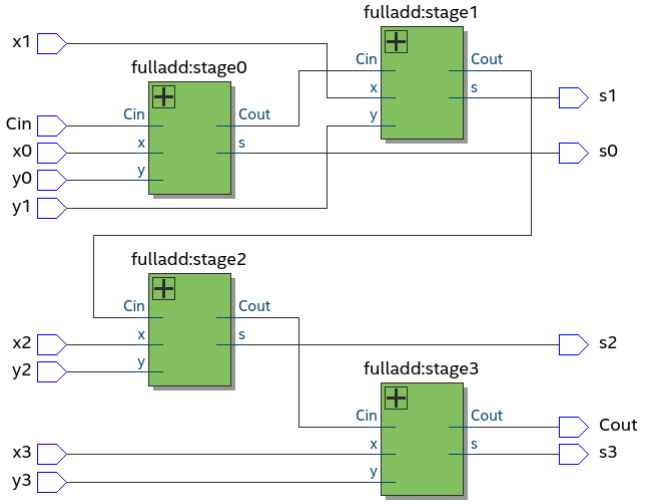

Figure 30: RTL View of 4 Bit Full Adder.

6.5 Simulation Waveform: Modelsim is used for Simulation

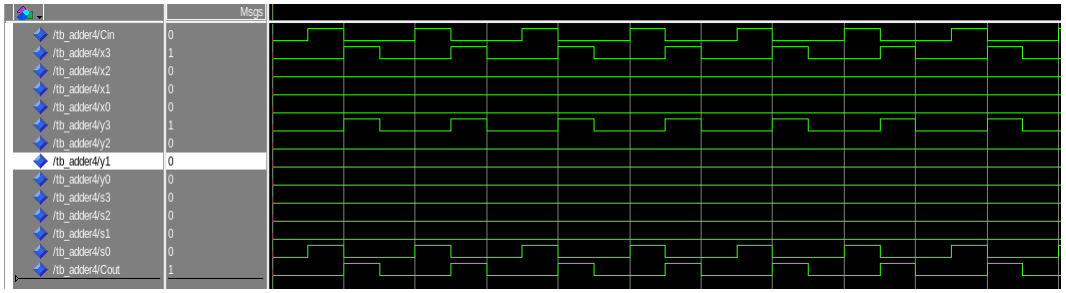

Figure 31: Simulation Waveform.

6.6 Conclusion: By using user packages, verbose and repetitive code are prevented. In this experiment 1 bit Full adder is designed by using package style of VHDL programming. Testbench are programmed to simulated the waveforms.

Experiment 7 - Verilog - 1 (19-03-2021)

7.1 Theory: Verilog is a hardware description language similar to VHDL. It supports similar modelling styles as well, including dataflow, structual and behavioural modelling. Verilog is weakly typed and has more C-like notation, while being relatively less verbose. In this experiment, two basic devices (full adder and 4:1 multiplexer) are implemented using the three types of modelling.

7.2 Codes:

Full Adder : Dataflow

```
module FA_dataflow(
input A, B, Cin,
output S, Cout);

assign S= A ^ B ^ Cin;
assign Cout= (A & B) | (B & Cin) | (A & Cin);
endmodule
```

Full Adder: Behavioural

```
module FA_behav(
input A, B, Cin,
output reg S, Cout
);

always @(*)
begin
S = A^B^Cin;
Cout = (A&B)|(B&Cin)|(A&Cin);
end
endmodule
```

Full Adder: Structural

```
module FA_struct(
input A, B, Cin,
output S, Cout);

wire a1, a2, a3;
```

```
xor u1(a1,A,B);
and u2(a2,A,B);
and u3(a3,a1,Cin);
or u4(Cout,a2,a3);
xor u5(S,a1,Cin);
endmodule
```

4:1 Mux - Dataflow

4:1 Mux- Behavioural

```
module Mux_behav(
input i0,i1,i2,i3,s0,s1,
output reg y);

always @(i0 or i1 or i2 or i3 or s0, s1)
begin
if(s1 & s0)
y<=i3;
else if(s1 & (~(s0)))
y<=i2;
else if(~(s1) & s0)
y<=i1;
else
y<=i0;
end
endmodule
```

4:1 Mux - Structural

```
module Mux_struct(out, a, b, c, d, s0, s1);
output out;
input a, b, c, d, s0, s1;
wire s0bar, s1bar, T1, T2, T3;
not_gate u1(s1bar, s1);
not_gate u2(s0bar, s0);
and_gate u3(T1, a, s0bar, s1bar);
and_gate u4(T2, b, s0, s1bar);
and_gate u5(T3, c, s0bar, s1);
and_gate u6(T4, d, s0, s1);
or_gate u7(out, T1, T2, T3, T4);
endmodule
```

```
module and_gate(output a, input b, c, d);
assign a = b & c & d;
endmodule

module not_gate(output e, input f);
assign e = ~ f;
endmodule

module or_gate(output l, input m, n, o, p);
assign l = m | n | o | p;
endmodule
```

7.3 Testbench:

Full Adder

```
--TestBench for Full Adder for all 3 approaches
--entity name needs to be changed for every approach
`timescale 100ps/ 10ps
module FA_dataflow_tb();
reg A,B,Cin;
wire S,Cout;

FA_dataflow tb(
.A(A),
.B(B),
.Cin(Cin),
.S(S),
.Cout(Cout)
);


initial
begin
$monitor($time, "A=%b, B=%b, Cin=%b, S=%b, Cout=%b \n", A,B,Cin,S,Cout);
end


initial begin
A = 0;
```

```
B = 0;
Cin = 0;
#5;
A = 0;
B = 0;
Cin = 1;
#5;
A = 0;
B = 1;
Cin = 0;
#5;
A = 0;
B = 1;
Cin = 1;
#5;
A = 1;
B = 0;
Cin = 0;
#5;
A = 1;
B = 0;
Cin = 1;
#5;
A = 1;
B = 1;
Cin = 0;
#5;
A = 1;
B = 1;
Cin = 1;
#5;
end

endmodule
```

4:1 Mux

```
`timescale 1ps/1ps
module Mux_tb;

wire out;
```

```
reg a;
reg b;
reg c;
reg d;
reg s0, s1;
wire y;
reg i0,i1,i2,i3;


Mux_struct tb_struct(.out(out), .a(a), .b(b), .c(c), .d(d), .s0(s0), .s1(s1));
initial
begin

a=1'b0; b=1'b0; c=1'b0; d=1'b0;
s0=1'b0; s1=1'b0;
#500 $finish;

end

always #40 a=~a;
always #20 b=~b;
always #10 c=~c;
always #5 d=~d;
always #5 s0=~s0;
always #10 s1=~s1;

always@(a or b or c or d or s0 or s1)
$monitor("At time = %t, Output = %d", $time, out);

// for data flow
Mux_dataflow tb_df(.y(y), .s0(s0), .s1(s1),
.i0(i0), .i1(i1), .i2(i2), .i3(i3));
initial
begin

i0=1'b0; i1=1'b0; i2=1'b0; i3=1'b0;
s0=1'b0; s1=1'b0;
#500 $finish;

end
```

```
always #40 i0=~i0;
always #20 i1=~i2;
always #10 i2=~i2;
always #5 i3=~i3;
always #5 s0=~s0;
always #10 s1=~s1;

always@(i0 or i1 or i2 or i3 or s0 or s1)
$monitor("At time = %t, Output = %d", $time, out);

// for behav
reg s01,s11,i01,i11,i21,i31;
wire y1;
Mux_behav tb_behav(.y(y1), .s0(s01), .s1(s11),
.i0(i01), .i1(i11), .i2(i21), .i3(i31));
initial
begin

i01=1'b0; i11=1'b0; i21=1'b0; i31=1'b0;
s01=1'b0; s11=1'b0;
#500 $finish;

end

always #40 i01=~i01;
always #20 i11=~i11;
always #10 i21=~i21;
always #5 i31=~i31;
always #5 s01=~s01;
always #10 s11=~s11;

always@(i01 or i11 or i21 or i31 or s01 or s11)
$monitor("At time = %t, Output = %d", $time, y1);

endmodule;
```

7.4 RTL Schematic: Quartus prime is used for RTL View of –

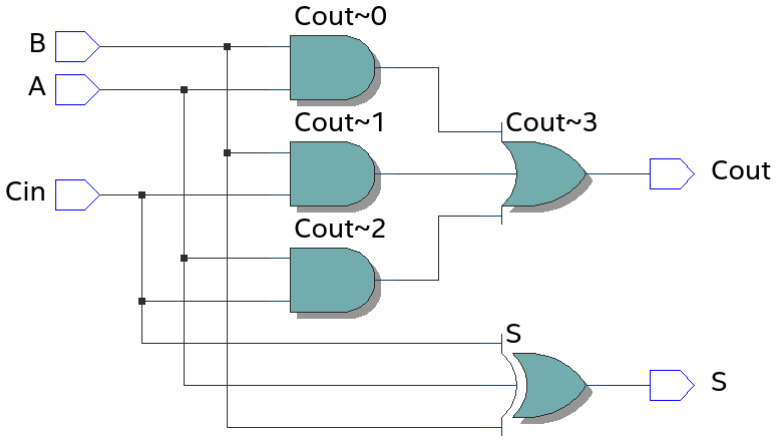

Figure 32: RTL View of Full Adder (Dataflow).

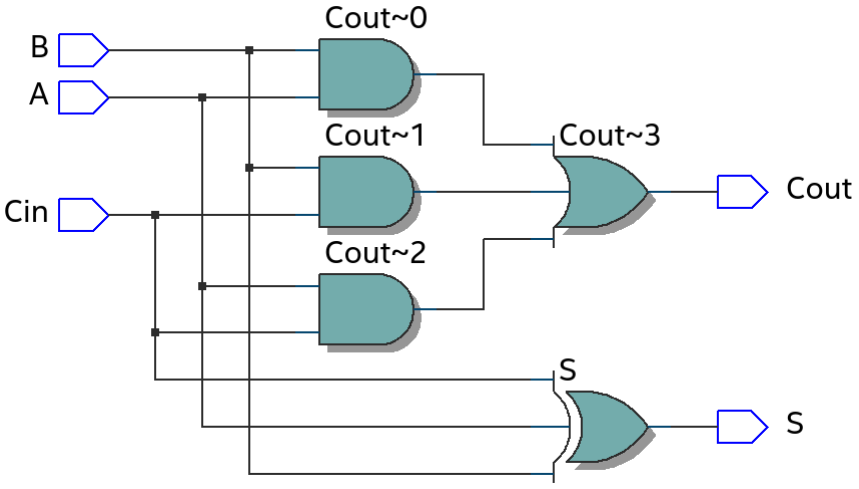

Figure 33: RTL View of Full Adder (behavorial).

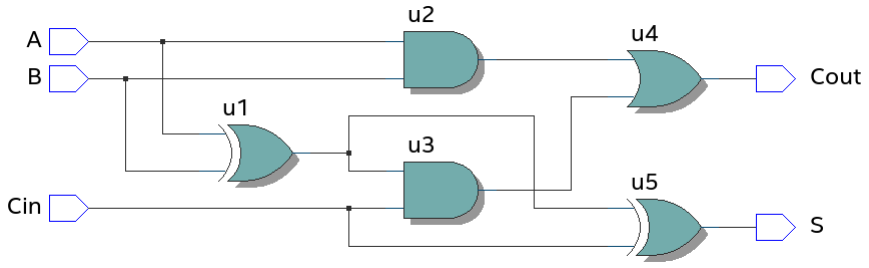

Figure 34: RTL View of Full Adder (Structural).

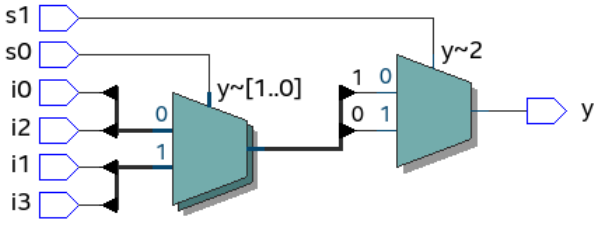

Figure 35: RTL View of MUX (Dataflow).

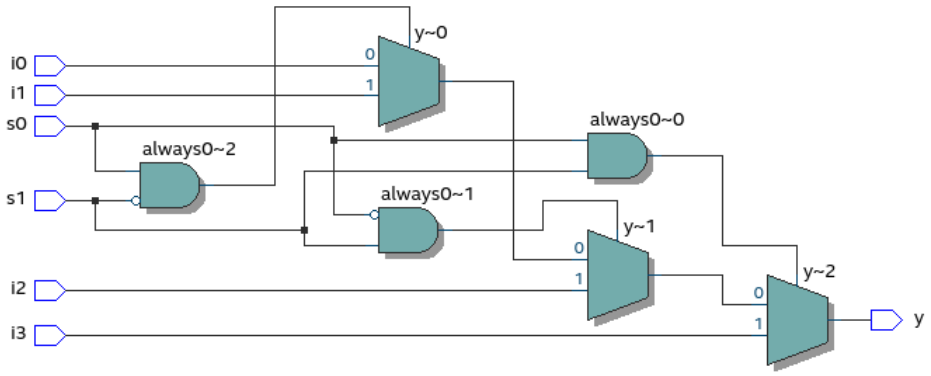

Figure 36: RTL View of MUX (behavorial).

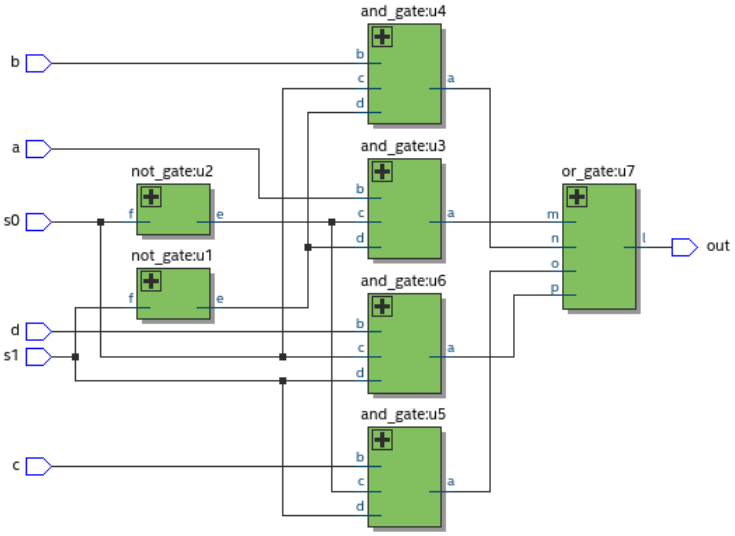

Figure 37: RTL View of MUX (Structural).

7.5 Simulation Waverform: ModelSim is used for Simulation of –

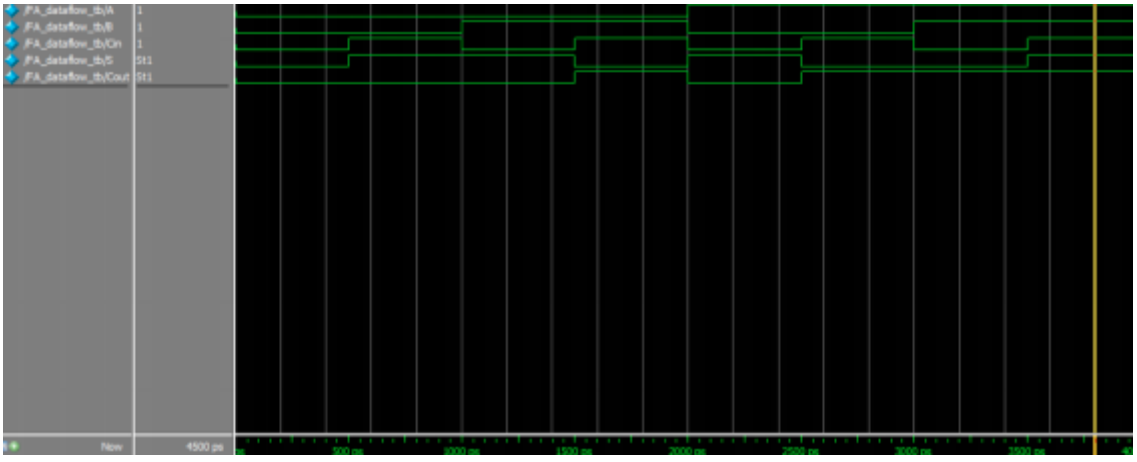

Figure 38: Simulation Full Adder (Dataflow).

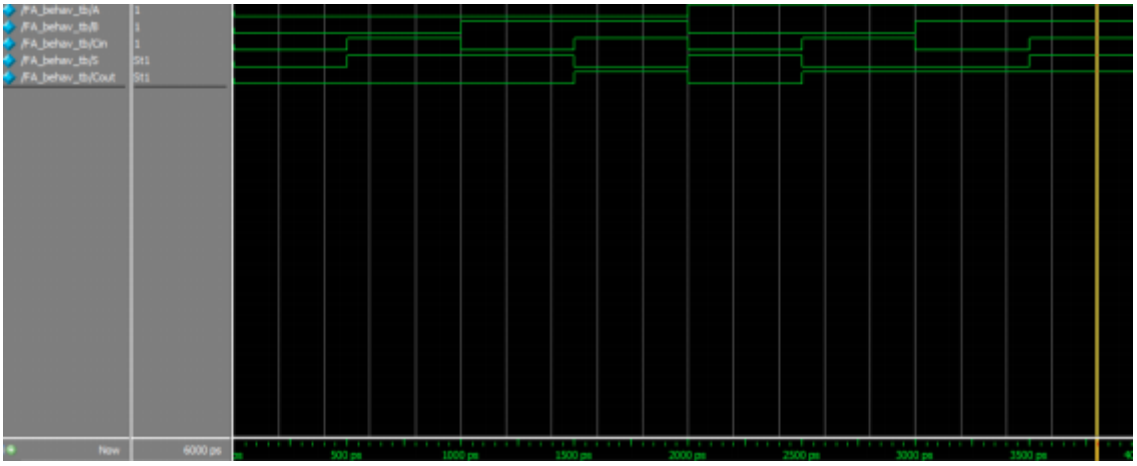

Figure 39: Simulation Full Adder (Behavorial).

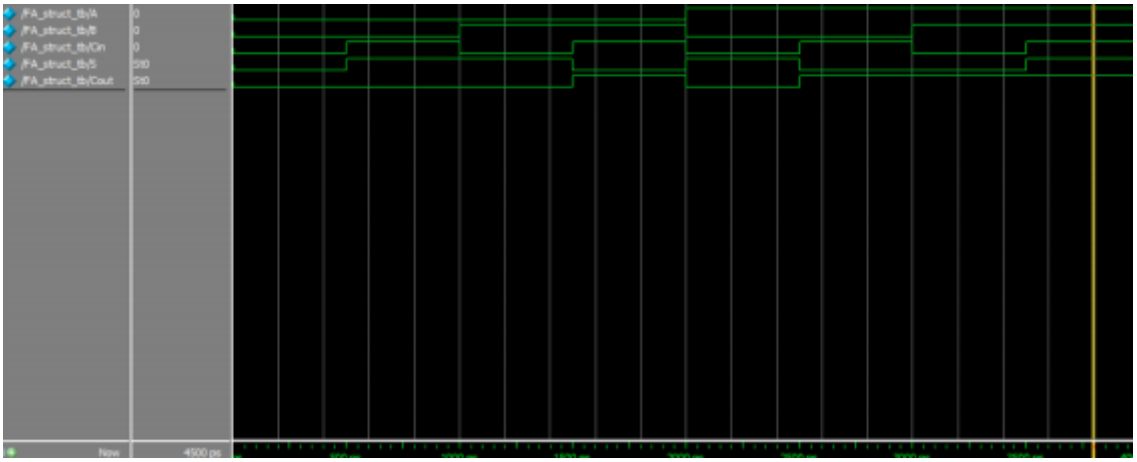

Figure 40: Simulation Full Adder (Structural).

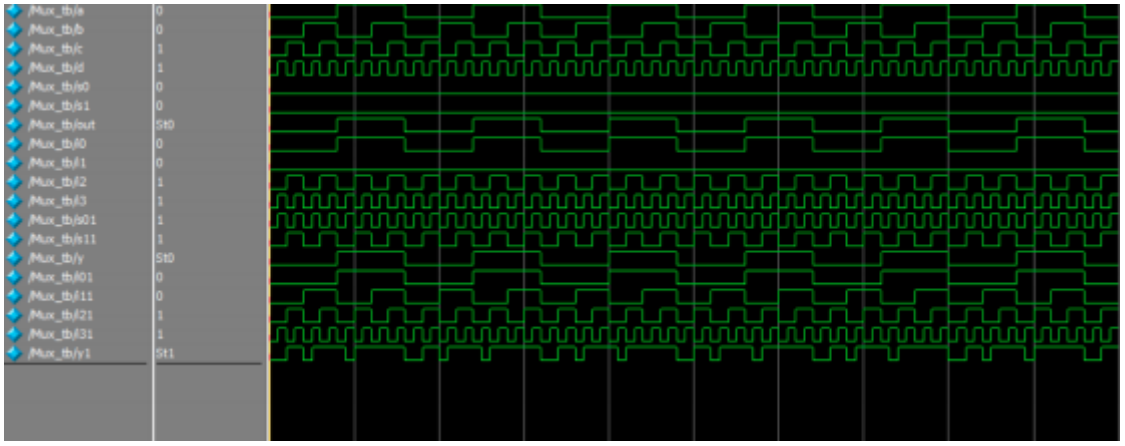

Figure 41: Simulation MUX.

7.6 Conclusion: In this experiment, Full-adder and 4:1 MUX is designed using Verilog language. The output obtained was verified using the truth table and with the help of a bunch of test-cases in the test-bench.

Experiment 8 - Verilog - 2 (25-03-2021)

8.1 Theory: Similar to every other hardware description language, Verilog allows for both synchronous and asynchronous designs. The primary difference between them is that the synchronous circuits perform all their operations, including reset checks, only on the rising edge of the input clock while an asynchronous circuit constantly checks and performs certain operations regardless of clock cycle, instead responding immediately to certain stimuli.

8.2 Codes:

shift register with synchronous reset

```
module SR_sync_reset
#(parameter N=8)
(
input wire clk, reset,
input wire s_in,
output wire s_out
);

reg [N-1:0] r_reg;
wire [N-1:0] r_next;


always @(posedge clk)
begin
if (~reset)
r_reg <= 0;
else
r_reg <= r_next;
end

assign r_next = {s_in, r_reg[N-1:1]};
assign s_out = r_reg[0];


endmodule
```

SR with Asynchronous reset

```
module SR_async_reset
#(parameter N=8)
(
input wire clk, reset,
input wire s_in,
output wire s_out
);

reg [N-1:0] r_reg;
wire [N-1:0] r_next;


always @(posedge clk, negedge reset)
begin
if (~reset)
r_reg <= 0;
else
r_reg <= r_next;
end

assign r_next = {s_in, r_reg[N-1:1]};
assign s_out = r_reg[0];


endmodule
```

Counter with Synchronous reset

```
module Counter_sync #(parameter N=4) (
input clock,
input reset,
output reg [3:0] out
);

always @(posedge clock)
begin
if (!reset)
out <= 0;
else
```

```
out <= out+1;
end
endmodule
```

Counter with Asynchronous reset

```
module Counter_async #(parameter N=4) (
input clock,
input reset,
output reg [3:0] out
);

always @(posedge clock or negedge reset)
begin
if (!reset)
out <= 0;
else
out <= out+1;
end
endmodule
```

8.3 Testbench:

shift register with synchronous reset

```
`timescale 10ps / 1ps
module SR_sync_reset_tb;
// Inputs
reg clk ;
reg reset;
// Outputs
reg s_in;
wire s_out;
// Instantiate the Unit Under Test (UUT)
SR_sync_reset #(4) uut (
.clk(clk),
.reset(reset),
.s_in(s_in),
.s_out(s_out)
);
```

8.3.0

```
integer i, j;
initial
begin

clk = 0;
for(i =0; i<=40; i=i+1)
begin
#10 clk = ~clk;
end
end

initial
begin

$dumpfile("test.vcd");
$dumpvars(0,SR_sync_reset_tb);

s_in = 0; reset =1;
#2 s_in = 0 ; reset = 0;
#2 reset =1;
for(i =0; i<=10; i=i+1)
begin
#20 s_in = ~s_in;
end
#20 s_in =1;
#20 s_in = 1;
#20 s_in =0;
#20 s_in =1;
#20 s_in = 1;
#20 s_in =0;
#20 s_in =1;
#20 s_in = 1;
#20 s_in =0;
end

initial begin
$monitor("clk=%d s_in=%d,s_out=%d",clk,s_in, s_out);
end

endmodule
```

SR with Asynchronous reset

```
`timescale 10ps / 1ps
module SR_async_reset_tb;
// Inputs
reg clk ;
reg reset;
// Outputs
reg s_in;
wire s_out;
// Instantiate the Unit Under Test (UUT)
SR_async_reset #(4) uut (
.clk(clk),
.reset(reset),
.s_in(s_in),
.s_out(s_out)
);

integer i, j;
initial
begin

clk = 0;
for(i =0; i<=40; i=i+1)
begin
#10 clk = ~clk;
end
end

initial
begin

$dumpfile("test.vcd");
$dumpvars(0,SR_async_reset_tb);

s_in = 0; reset =1;
#2 s_in = 0 ; reset = 0;
#2 reset =1;
for(i =0; i<=10; i=i+1)
begin
#20 s_in = ~s_in;
```

```
end
#20 s_in =1;
#20 s_in = 1;
#20 s_in =0;
#20 s_in =1;
#20 s_in = 1;
#20 s_in =0;
#20 s_in =1;
#20 s_in = 1;
#20 s_in =0;
end

initial begin
$monitor("clk=%d s_in=%d,s_out=%d",clk,s_in, s_out);
end

endmodule
```

Counter with synchronous reset

```
`timescale 10ps / 1ps
module Counter_sync_tb;
// Inputs
reg clock ;
reg reset;
// Output
wire[3:0] out;
// Instantiate the Binary Counter
Counter_sync #(4) uut1 (
.clock(clock),
.reset(reset),
.out(out)
);

integer i;
initial
begin

clock = 0;
for(i =0; i<=40; i=i+1)
begin
```

```
#10 clock = ~clock;
end
end

initial
begin

$dumpfile("test1.vcd");
$dumpvars(0,Counter_sync_tb);

reset = 0;
//reset =1;
//#2 reset = 0;
#20 reset =1;
end

initial begin
$monitor("clock=%d binary=%4b",clock,out);
end

endmodule
```

Counter with Asynchronous Reset

```
`timescale 10ps / 1ps
module Counter_async_tb;
// Inputs
reg clock ;
reg reset;
// Output
wire[3:0] out;
// Instantiate the Binary Counter
Counter_async #(4) uut (
.clock(clock),
.reset(reset),
.out(out)
);

integer i;
initial
begin
```

```
clock = 0;
for(i =0; i<=40; i=i+1)
begin
#10 clock = ~clock;
end
end

initial
begin

$dumpfile("test.vcd");
$dumpvars(0,Counter_async_tb);

reset =1;
#2 reset = 0;
#2 reset =1;
end

initial begin
$monitor("clock=%d binary=%4b",clock,out);
end

endmodule
```

8.4 RTL Schematic: Quartus prime is used for RTL View of –

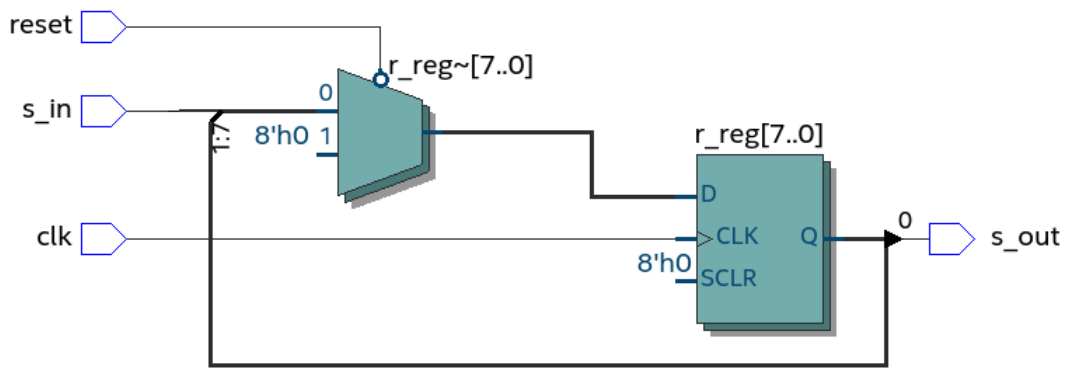

Figure 42: RTL View of Synchronous Shift register.

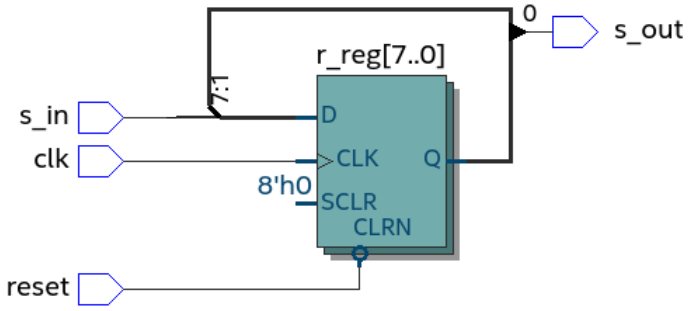

Figure 43: RTL View of Asynchronous Shift Register.

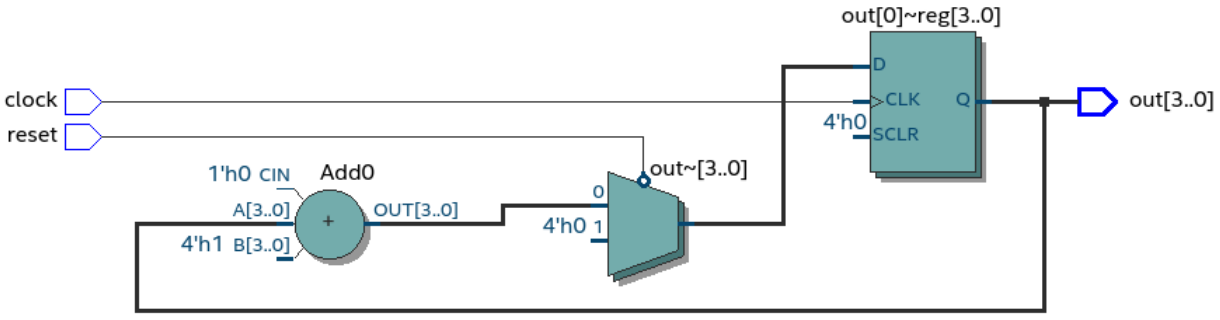

Figure 44: RTL View of Synchronous Counter.

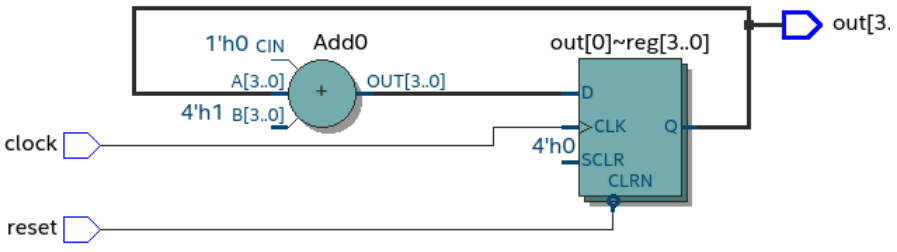

Figure 45: RTL View of Asynchronous Counter.

8.5 Simulation Waverform: ModelSim is used for Simulation of –

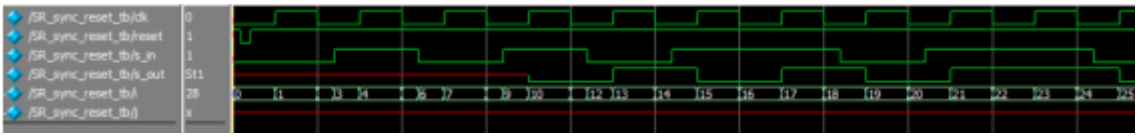

Figure 46: Simulation of Synchronous Shift register.

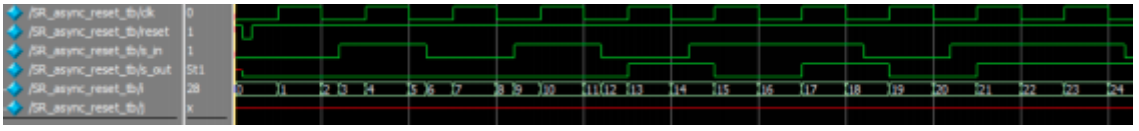

Figure 47: Simulation of Asynchronous Shift Register.

Figure 48: Simulation of Synchronous Counter.

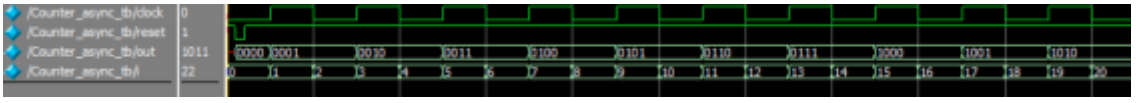

Figure 49: Simulation of Asynchronous Counter.

8.6 Conclusion: In this experiment, synchronous asynchronous Shift registers and counters are designed using Verilog. The output was verified using test-bench.

Experiment 9 - FPGA - 1 (08-04-2021)

9.1 Theory: Field programmable gate arrays (FPGAs) are used by industry to implement logic when the system is complex, the time-to-market is short, the performance (e.g., speed) of an FPGA is acceptable, and the volume of potential sales does not warrant the investment in a standard cell-based ASIC. Circuits can be rapidly prototyped into an FPGA using an HDL. Once the HDL model is verified, the description is synthesized and mapped into the FPGA. FPGA vendors provide software tools for synthesizing the HDL description of a circuit into an optimized gate-level description and mapping (fitting) the resulting netlist into the resources of their FPGA. This process avoids the detailed assembly of ICs that is required by composing a circuit on a breadboard, and the process involves significantly less risk of failure, because it is easier and faster to edit an HDL description than to rewire a breadboard.

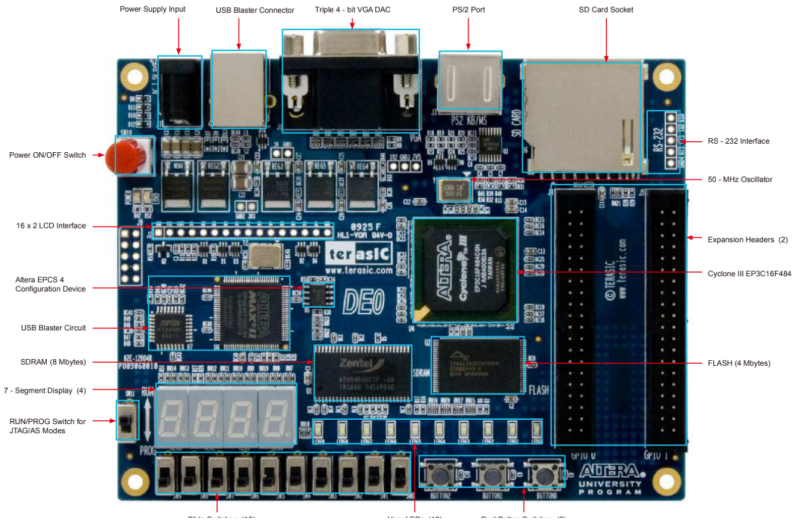

Figure 50: FPGA board used : The DE0 Borad.

9.2 Screenshots:

Half Adder

The following screenshots are for the Half Adder Experiment.

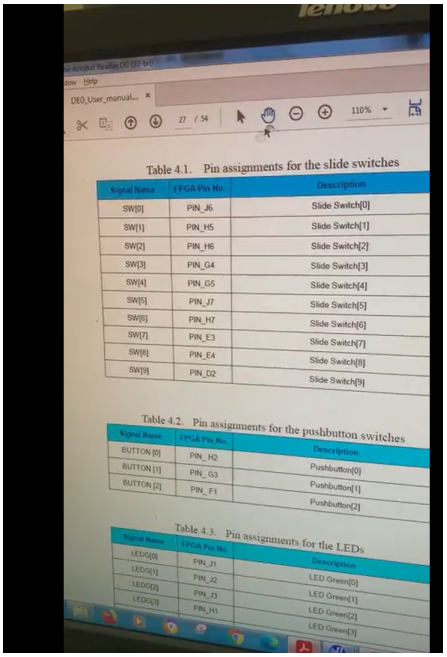

Table 4.1. Pin assignments for the slide switches

Signal Name	FPGA Pin No.	Description
SW[0]	PIN_J6	Slide Switch[0]
SW[1]	PIN_H5	Slide Switch[1]
SW[2]	PIN_H6	Slide Switch[2]
SW[3]	PIN_G4	Slide Switch[3]
SW[4]	PIN_G5	Slide Switch[4]
SW[5]	PIN_J7	Slide Switch[5]
SW[6]	PIN_H7	Slide Switch[6]
SW[7]	PIN_E3	Slide Switch[7]
SW[8]	PIN_E4	Slide Switch[8]
SW[9]	PIN_D2	Slide Switch[9]

Table 4.2. Pin assignments for the pushbutton switches

Signal Name	FPGA Pin No.	Description
BUTTON[0]	PIN_H2	Pushbutton[0]
BUTTON[1]	PIN_G3	Pushbutton[1]
BUTTON[2]	PIN_F1	Pushbutton[2]

Table 4.3. Pin assignments for the LEDs

Signal Name	FPGA Pin No.	Description
LEDG[0]	PIN_J1	LED Green[0]
LEDG[1]	PIN_J2	LED Green[1]
LEDG[2]	PIN_J3	LED Green[2]
LEDG[3]	PIN_H1	LED Green[3]

Figure 51: Shows the manual for the pin assignment.

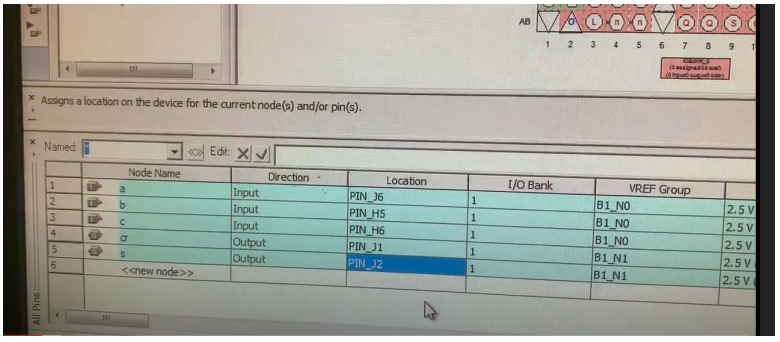

Figure 52: Pins assigned.

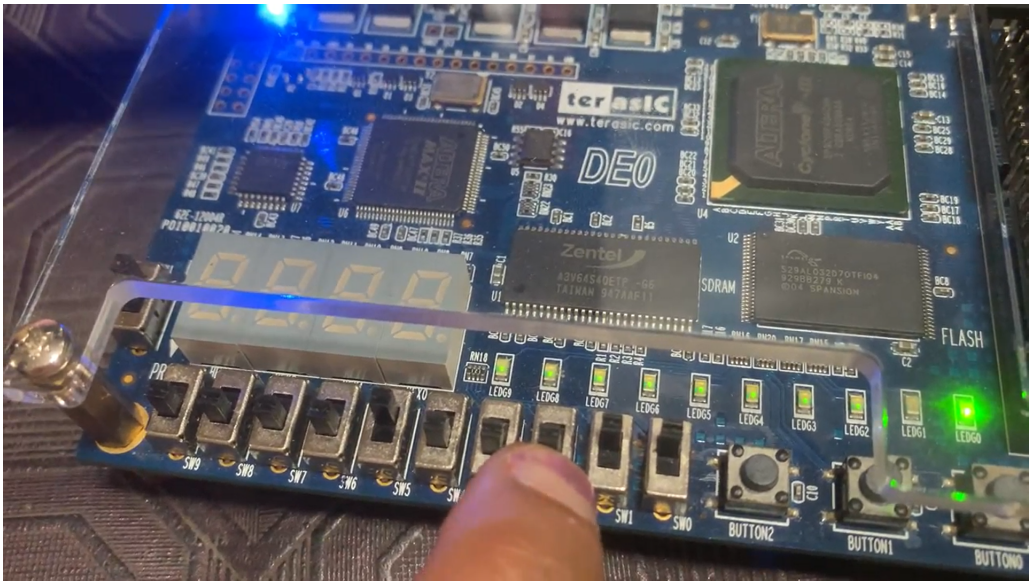

Figure 53: The green light in the figure indicates the outputs "sum" and "carry" for the input a=1, b=0 , cin = 0.

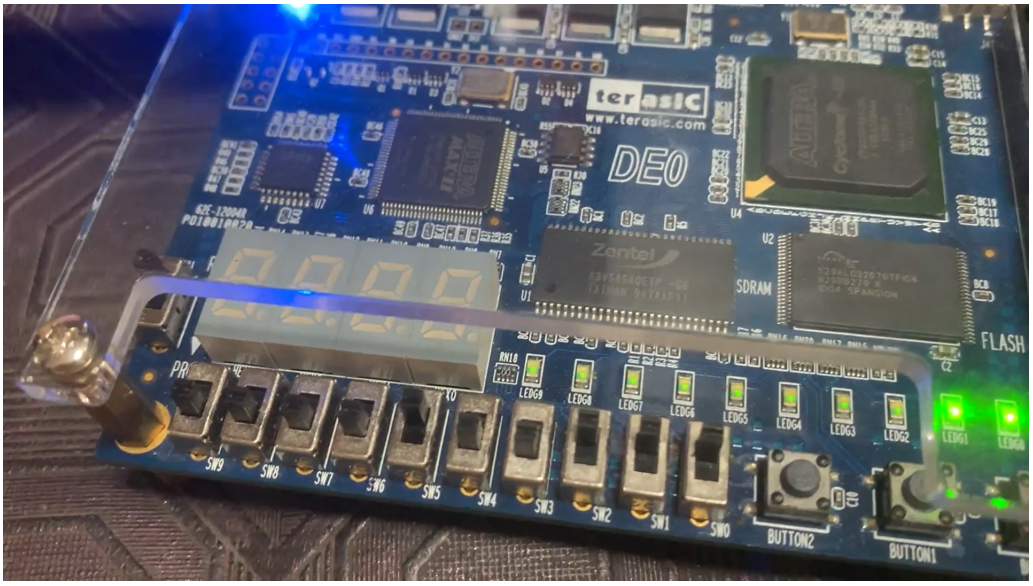

Figure 54: The green light in the figure indicates the outputs "sum" and "carry" for the input a=1, b=1 , cin = 0.

Mux 4:1

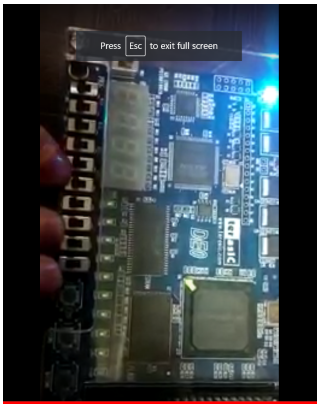

Figure 55: The 6 switches (two for the select lines) are used as the input.

9.3 Conclusion: Hence the burning of the Half Adder code and the Mux 4:1 code on the FPGA was successfully observed. Once the code is wriiten Select the board from the list of devices whose serial number should match the serial number of the FPGA board on which you will burn the code. In this experiment we used a FPGA board by altera. For pin assignment in Quartus II, Go to Assignments Tab -¿ Pin Planner. Assign the pins by referring to the User Manual of the FPGA board.For burining the code:
Go To Tools Tab -¿ Programmer -¿ select .sof file -¿ Click Start.
After burning the code on the board inputs were given via the switches and the correct outputs were obsereved on the green LEDs.

Experiment 10 - FPGA - 2 (09-04-2021)

10.1 Theory: Field programmable gate arrays (FPGAs) are used by industry to implement logic when the system is complex, the time-to-market is short, the performance (e.g., speed) of an FPGA is acceptable, and the volume of potential sales does not warrant the investment in a standard cell-based ASIC. Circuits can be rapidly prototyped into an FPGA using an HDL. Once the HDL model is verified, the description is synthesized and mapped into the FPGA. FPGA vendors provide software tools for synthesizing the HDL description of a circuit into an optimized gate-level description and mapping (fitting) the resulting netlist into the resources of their FPGA. This process avoids the detailed assembly of ICs that is required by composing a circuit on a breadboard, and the process involves significantly less risk of failure, because it is easier and faster to edit an HDL description than to rewire a breadboard.

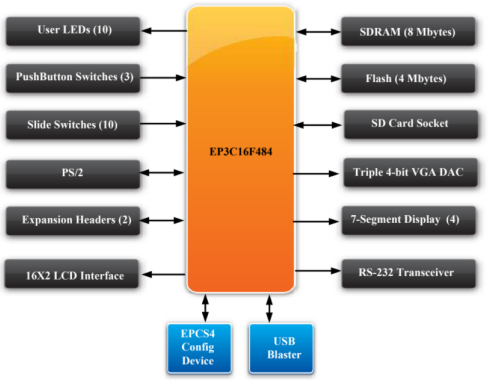

Figure 56: Block Diagram of DE0 board.

The following hardware is provided on the DE0 board:

- Altera Cyclone III 3C16 FPGA device
- Altera Serial Configuration device – EPCS4
- USB Blaster (on board) for programming and user API control; both JTAG and Active Serial (AS) programming modes are supported
- 8-Mbyte SDRAM
- 4-Mbyte Flash memory
- SD Card socket
- 3 pushbutton switches

- 10 toggle switches
- 10 green user LEDs
- 50-MHz oscillator for clock sources
- VGA DAC (4-bit resistor network) with VGA-out connector
- RS-232 transceiver
- PS/2 mouse/keyboard connector
- Two 40-pin Expansion Headers

10.2 Screenshots: The Screenshots are..

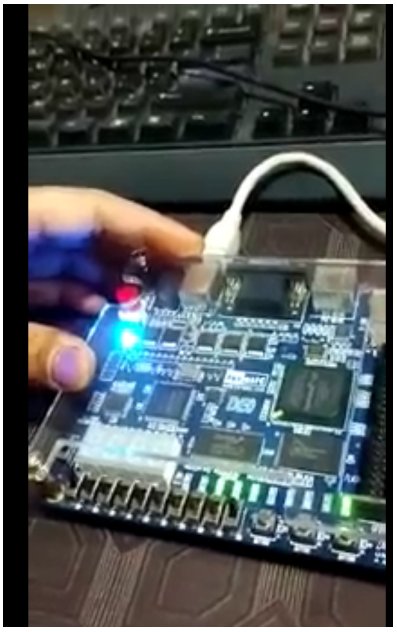

Figure 57: The green LED glows one by one showing the output of the shift register.

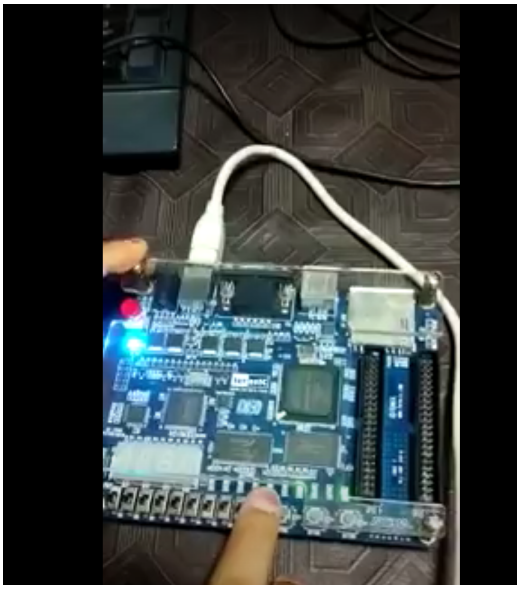

Figure 58: Output of the Counter, once the reset switch is made high the counting stops and then starts from 0.

10.3 Conclusion: The codes for the shift register and and counter implementation were burned and observed correctly on the FPGA board. As the clock frequency of the FPGA board was 50 MHz (can be seen in the clock section of the manual pin G_21 is 50MHz clock input) , the shif tand the counts for the above two experiments were not visible in every clock cycle. To reduce the clock frequency,a temporary signal names temp is introduces which is a vector of length 26 following code was implemented in Quartus II.