Visvesvaraya National Institute of Technology (VNIT), Nagpur

Digital Hardware Design (ECP313)

Lab Report

Submitted by :
Suraj Kaple (BT18ECE021)
Viraj Patel (BT18ECE015)
Ankit (BT18ECE007)
Semester VI

Group No 27

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

Contents

Experiment 1 - Introduction to VHDL (15-01-201)

1.1 Theory: VHDL stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language.A digital system in VHDL consists of a design entity that can contain other entities that are then considered components of the top-level entity. Each entity is modeled by an entity declaration and an architecture body. One can consider the entity declaration as the interface to the outside world that defines the input and output signals, while the architecture body contains the description of the entity and is composed of interconnected entities, processes and components, all operating concurrently, as schematically shown in Fig. 1

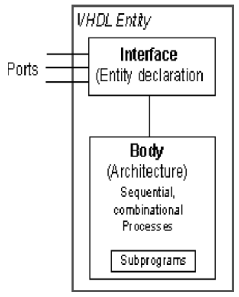


Figure 1: VHDL Block Implementation

In this experiment all the basic logic gates such as AND, OR, NAND, XOR, NOR, XNOR, NOT are implemented.

1.2 Codes:

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

entity all_gates is
Port(
a: in STD_LOGIC;
b: in STD_LOGIC;
c0: 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;

architecture dataflow of all_gates is
begin
c0 <= 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 dataflow;
```

1.3 Testbench:

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

entity all_gates_tb is
end all_gates_tb;

architecture tb of all_gates_tb is
  component all_gates
    port(
      a: in STD_LOGIC;
      b: in STD_LOGIC;
      c0: 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 component;

signal a: std_logic:='0';
signal b: std_logic:='0';
signal c0: std_logic:='0';
signal c1: std_logic:='0';
signal c2: std_logic:='0';
signal c3: std_logic:='0';
signal c4: std_logic:='0';
signal c5: std_logic:='0';
signal c6: std_logic:='0';

constant period : time := 100 ns;

begin
  uut: all_gates port map(
  a => a,
  b => b,
  c0 => c0,
  c1 => c1,
  c2 => c2,
  c3 => c3,
  c4 => c4,
  c5 => c5,
  c6 => c6
  );
  clock: process
  begin
    a <= '0';
    b <= '0';
    wait for period;
    a <= '0';
    b <= '1';
    wait for period;
    a <= '1';
    b <= '0';
    wait for period;
    a <= '1';
```

```
        b <= '1';
        wait for period;
    end process;
end;
```

1.4 RTL Schematic: - Here a,b are input ports and c0, c1, c2, c3, c4, c5, c6 are output ports. All input and output ports have std_logic as their datatype.

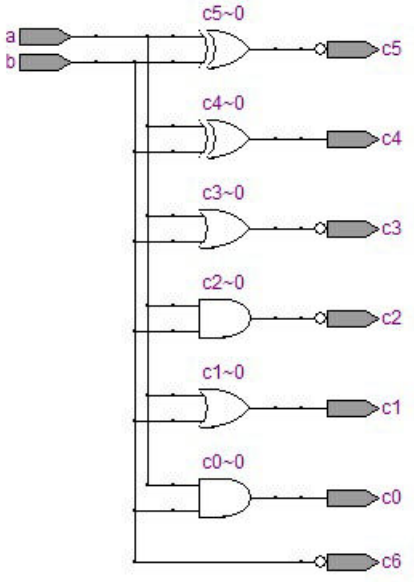


Figure 2: RTL View of logic gate implementation

It can be seen from RTL view that
c0 = 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

1.5 Simulation Waverform: -

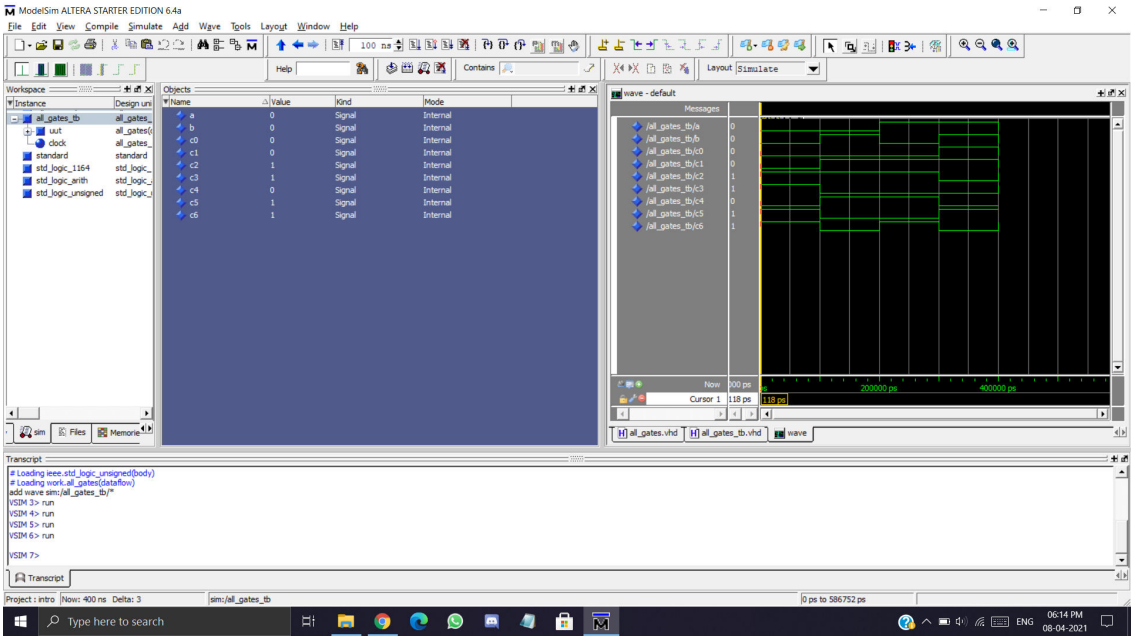
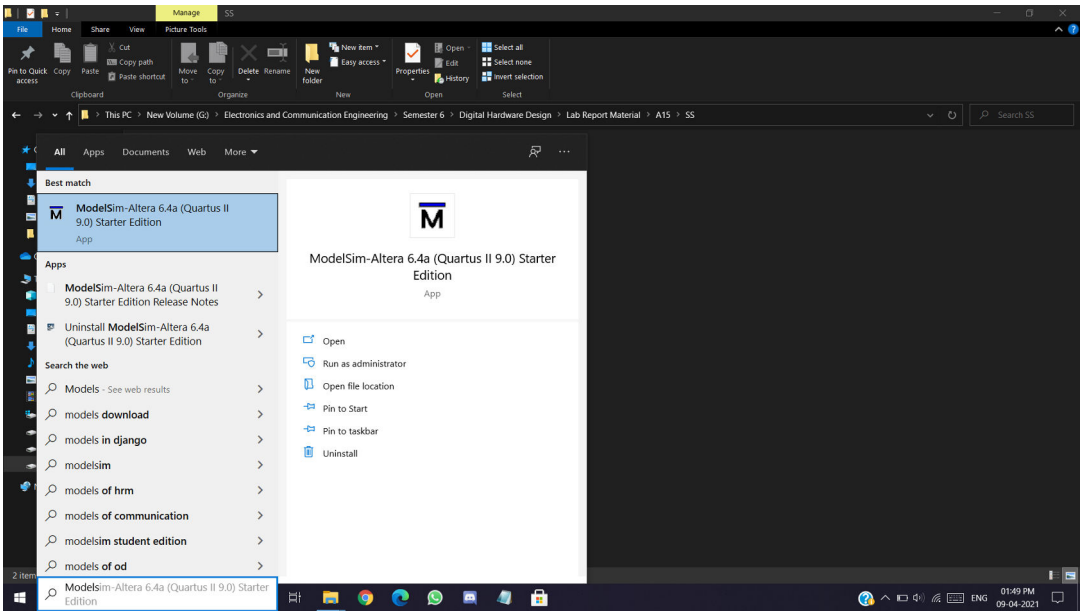


Figure 3: Simulated waveform

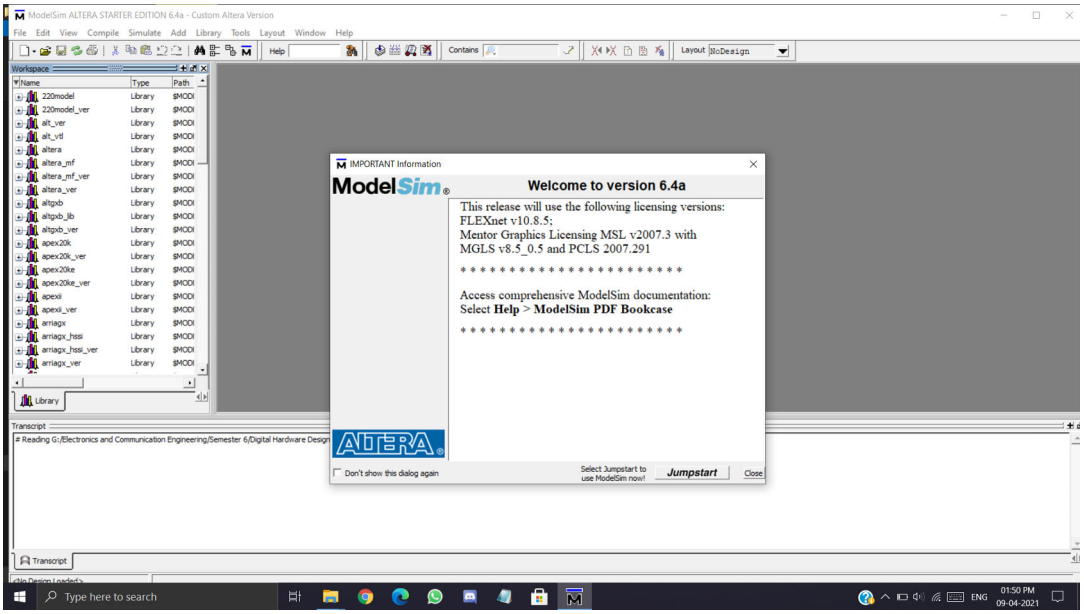
1.6 Conclusion: In above experiment we implemented data flow model of all basic logic gates in Quartus II 9.0 Web Edition. and simulated same in Modelsim - Altera 6.4a. Test bench was written in order to test the code and waveform was generated which can be seen in Fig. 3.

1.7 Screenshots: -

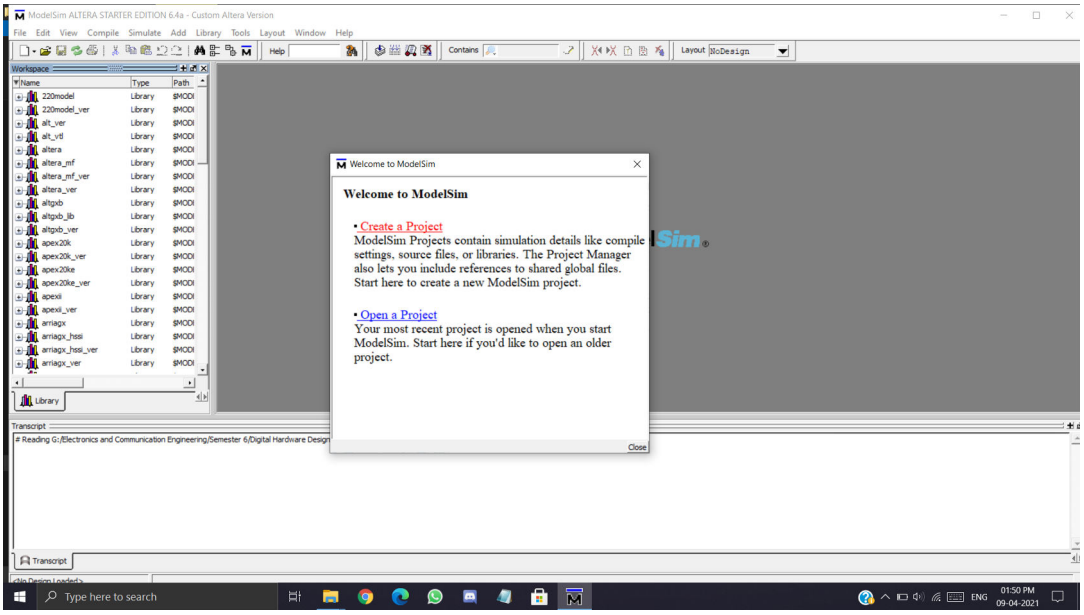
1. Open Modelsim by typing Modelsim in search bar and press **enter**.



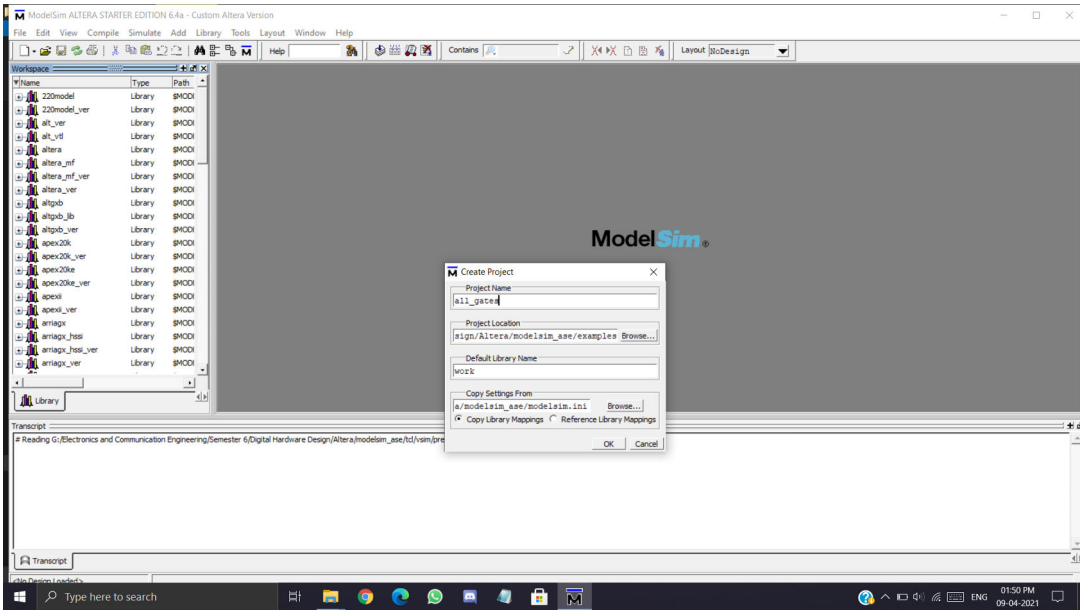
2. Dialog box will pop up click on **Jumpstart**.



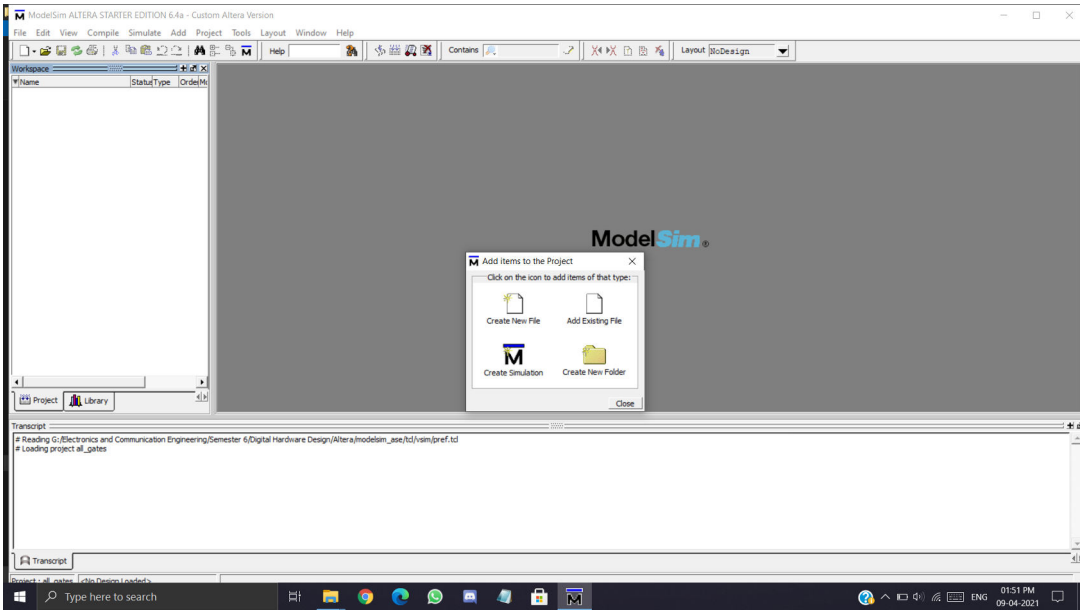
3. Click on **Create a Project**.



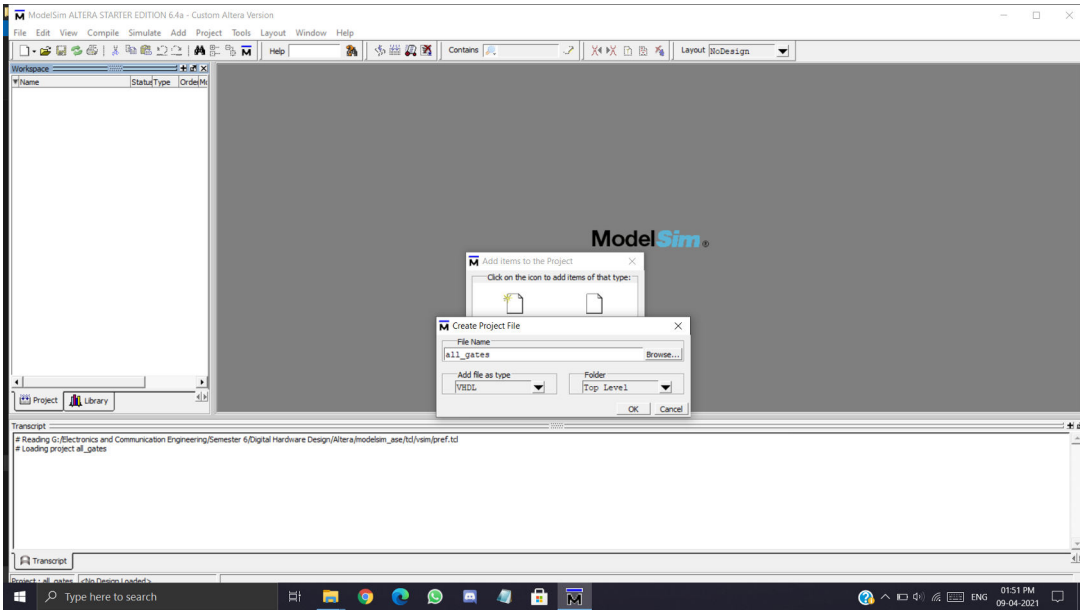
4. Write name of your project, select project location and click **ok**.



5. Click on **Create new file**.

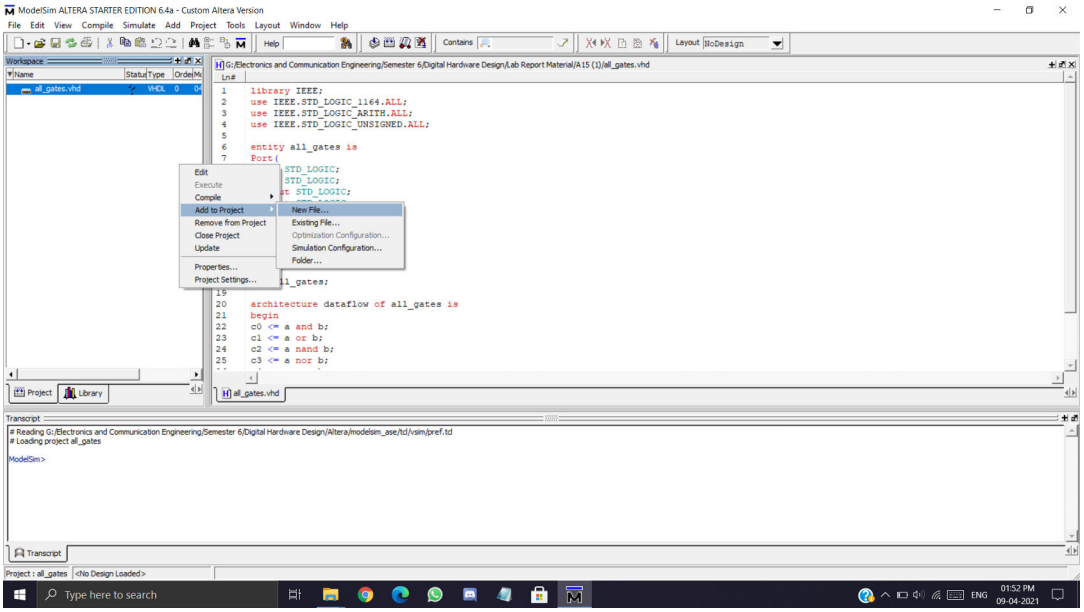


6. Write file name. **Note:** filename should be same as **entity name**.

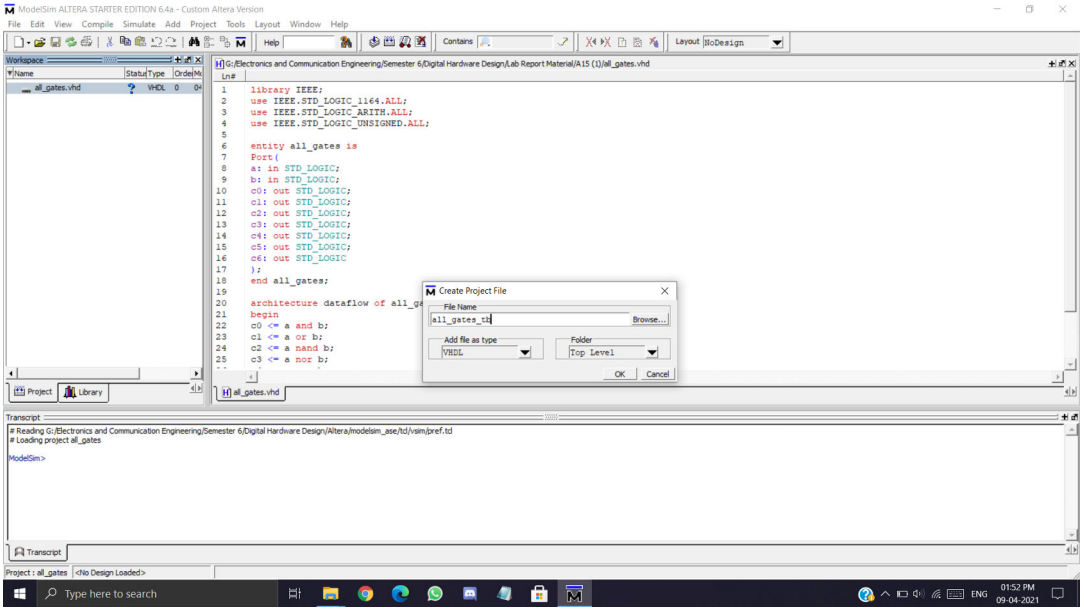


7. Write your VHDL code

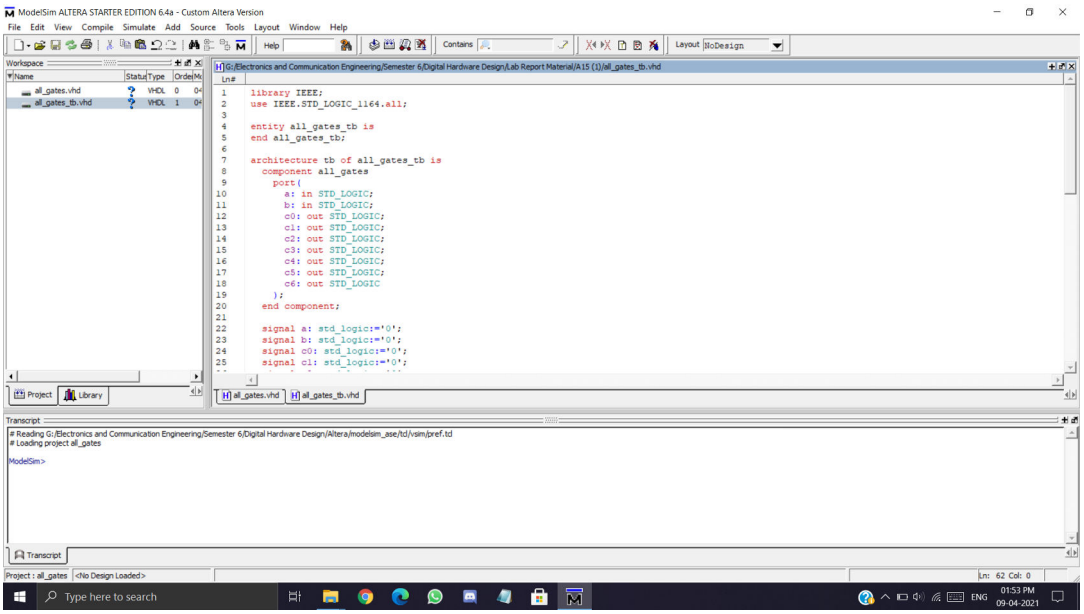
8. To create test bench, Right click anywhere in the workspace window → add to project → create new file.



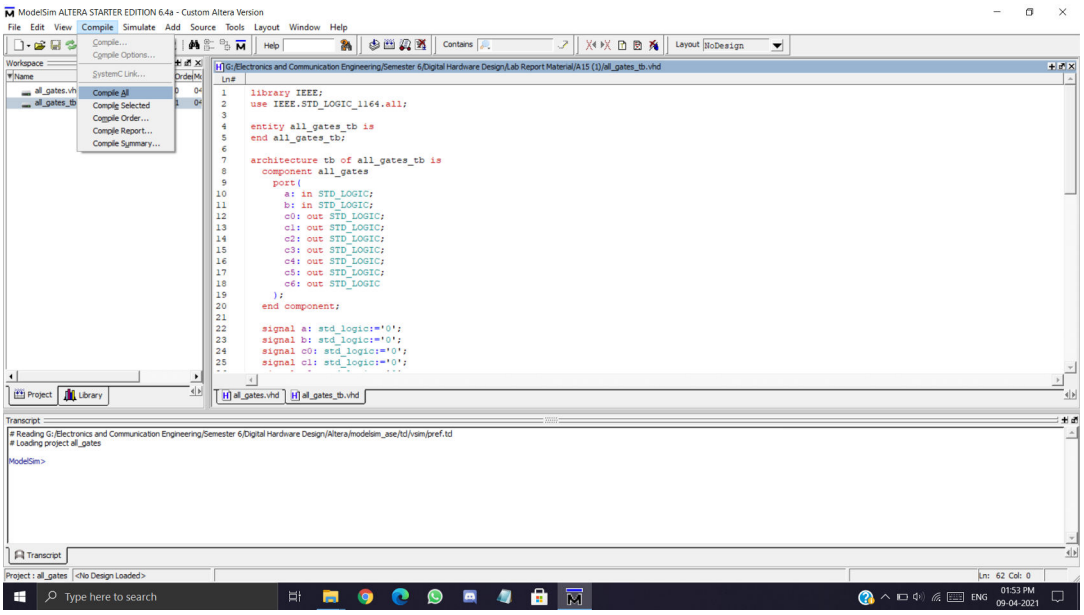
9. Write file name. **Note:** filename should be same as **entity name**



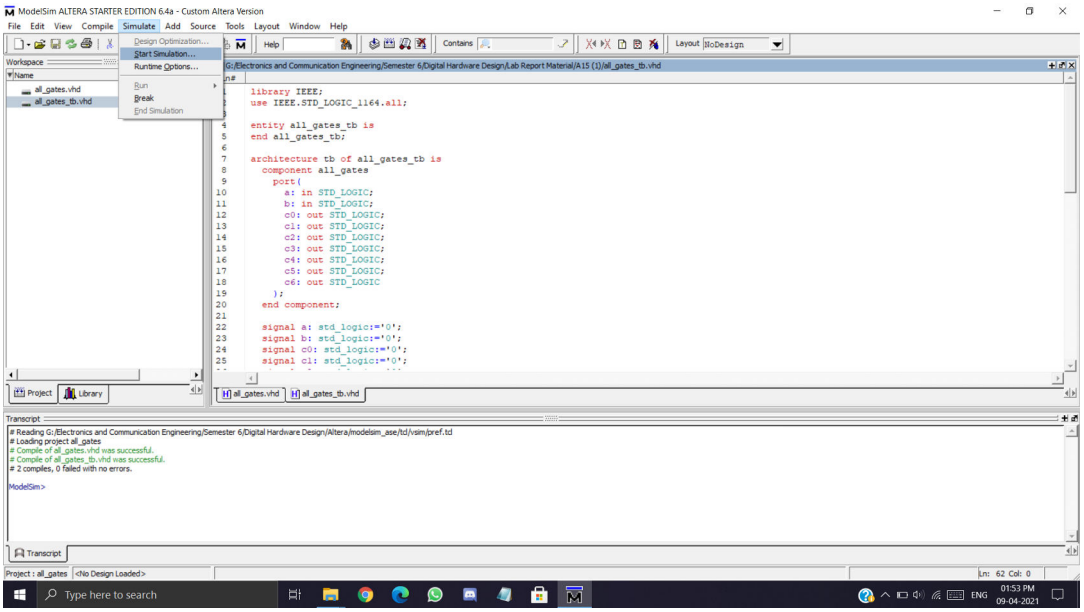
10. Write code for test bench.



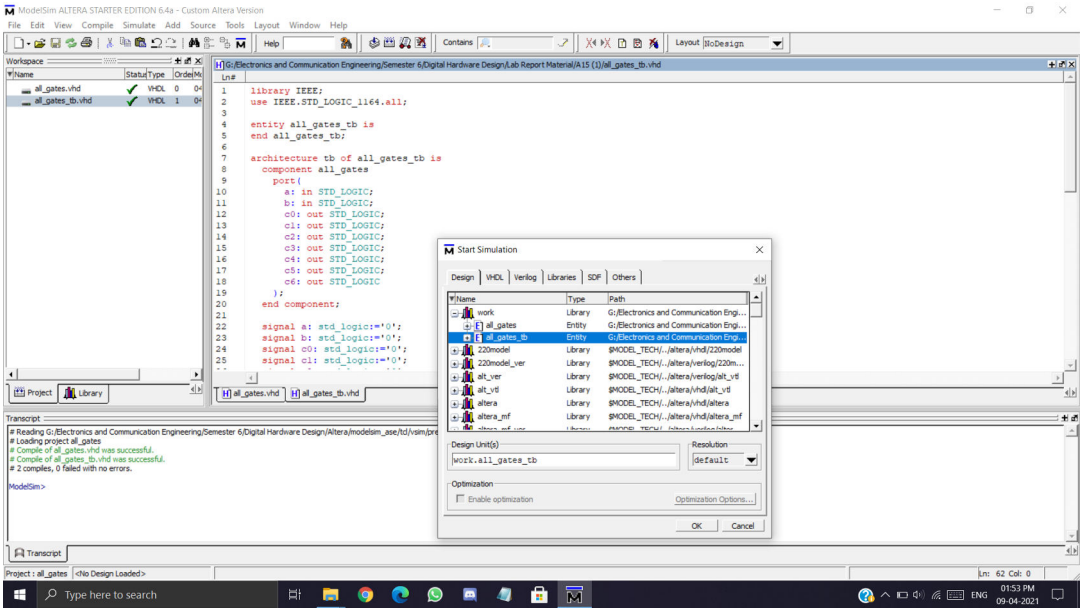
11. From menu bar click on **Compile→Compile All**.



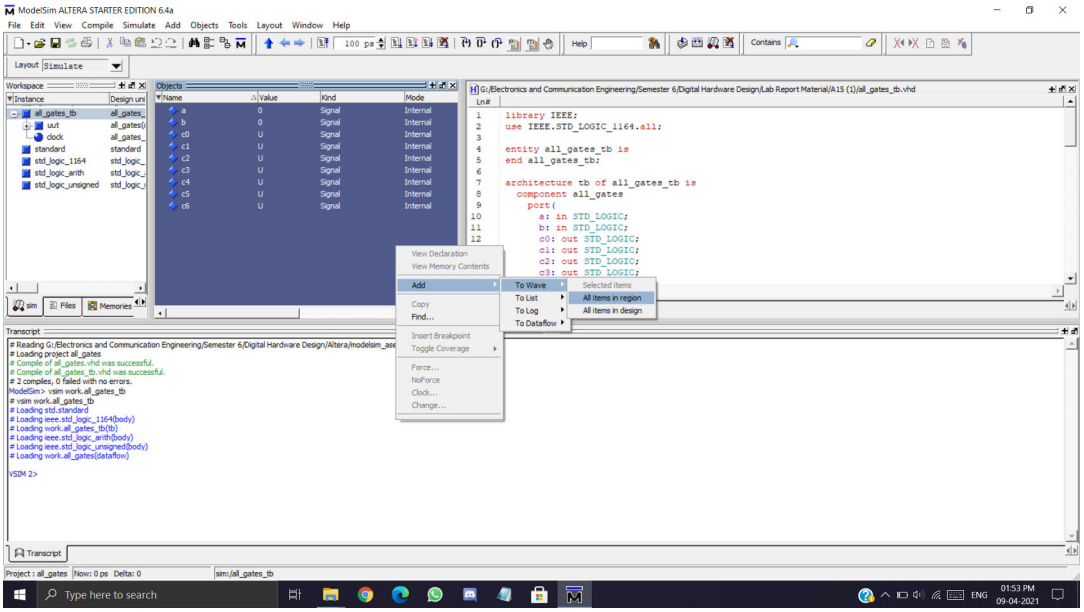
12. After successful compilation of both the codes, click on simulate from menu bar and start simulation.



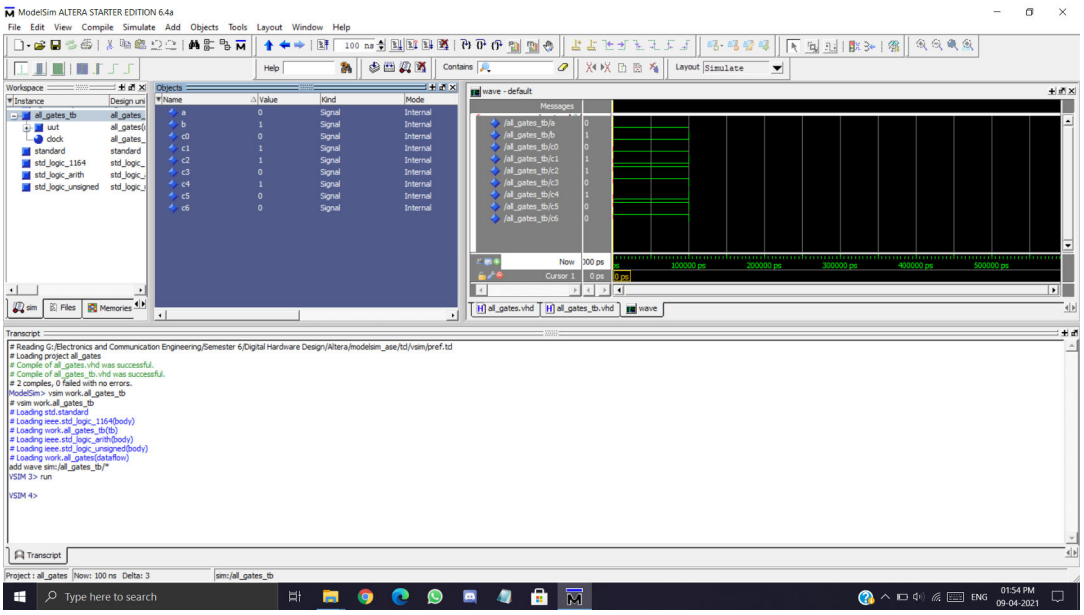
13. From pop-up window, click **work→test_bench file→ok**.



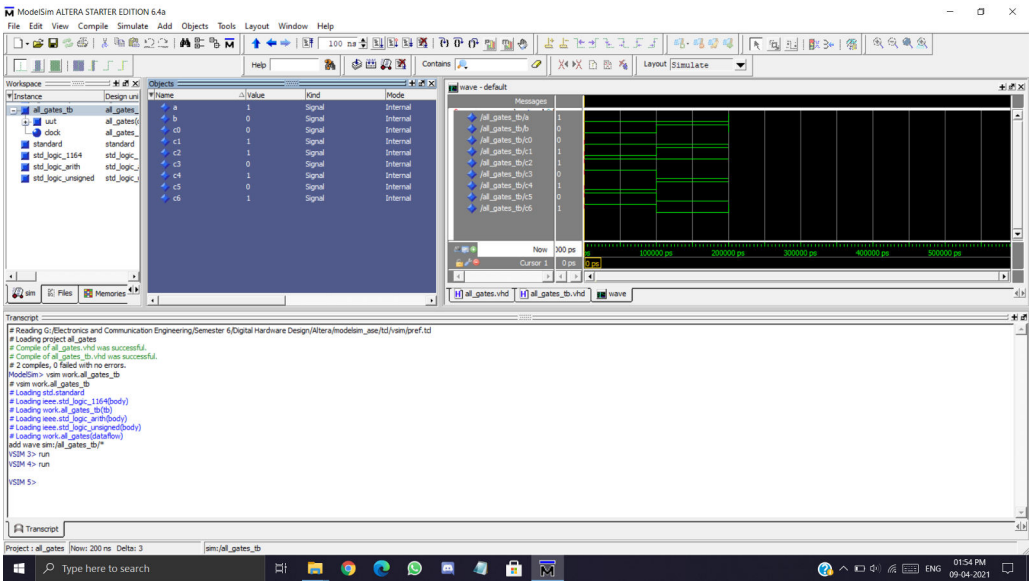
14. Right click anywhere in the objects window → Add→ To Wave→ All items in region.



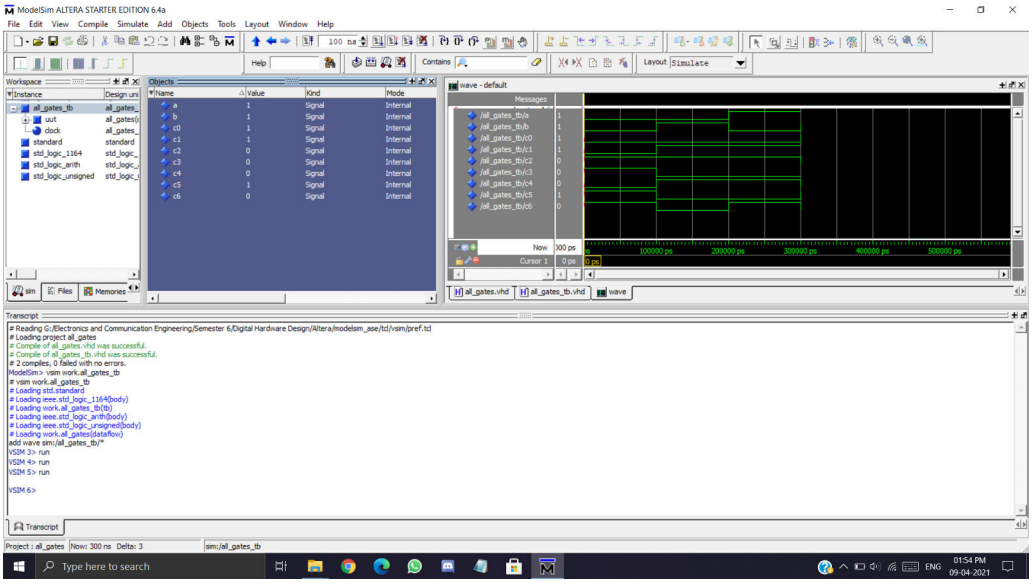
15. Select proper timescale click run.



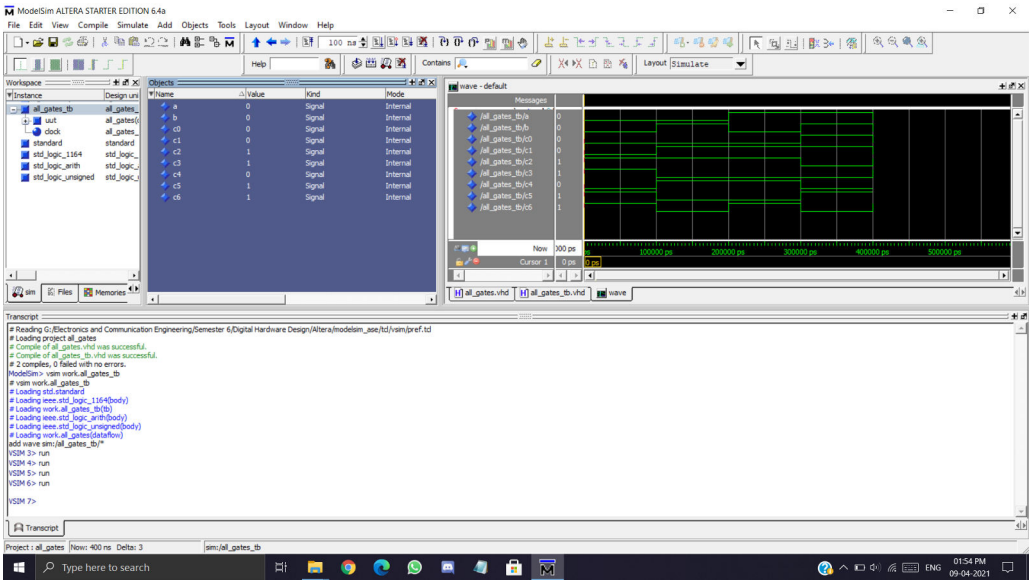
16. Click run.



17. Click run.



18. Continue clicking run till you cover all the test cases.



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

2.1 Theory: There are three types of modelling in which system designing in VHDL. They are

Data flow Model

The data flow representation describes how data moves through the system. This is typically done in terms of data flow between registers (Register Transfer level). The data flow model makes use of concurrent statements that are executed in parallel as soon as data arrives at the input.

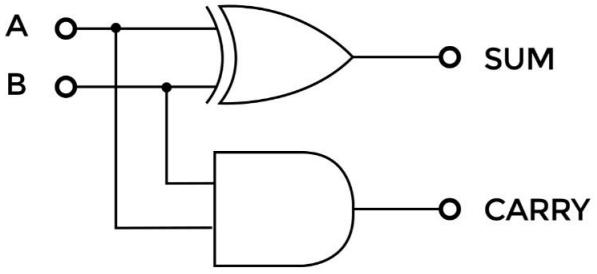
Behavioral Model

It is the behavioral model that describes a system in terms of what it does (or how it behaves) rather than in terms of its components and interconnection between them. A behavioral description specifies the relationship between the input and output signals. This could be a Boolean expression or a more abstract description such as the Register Transfer or Algorithmic level.

Structural Model

The structural model describes a system as a collection of gates and components that are interconnected to perform a desired function. A structural description could be compared to a schematic of interconnected logic gates. It is a representation that is usually closer to the physical realization of a system.

In this experiment we implemented Half adder circuit using all three models.



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

Figure 4: HAlf Adder

2.2 Codes:

Dataflow

```
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

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

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

architecture behavior of HA_behav is
    begin
    ha: process (a, b)
    begin
         if a = '1' then
            sum <= not b;
            carry_out <= b;
        else
            sum <= b;
            carry_out <= '0';
        end if;
    end process ha;
```

```
end behavior;
```

Structural

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

architecture structural of HA_struct 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:

Testbench for Dataflow

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

entity dataflow_tb is
end dataflow_tb;

architecture tb of dataflow_tb is
component HA_dataflow
 port (a, b: in std_logic;
 sum, carry_out: out std_logic);
end component;

signal a, b: std_logic;
signal sum, carry_out: std_logic;
constant period : time := 100 ns;

begin
  uut: HA_dataflow port map (
  a=>a, b=>b, sum=>sum,
  carry_out=>carry_out
  );
```

```
  clock: process
  begin
      a <= '0';
      b <= '0';
      wait for period;
      a <= '0';
      b <= '1';
      wait for period;
      a <= '1';
      b <= '0';
      wait for period;
      a <= '1';
      b <= '1';
      wait for period;
  end process;
end;
```

Testbench for Behavioral

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

entity behavioral_tb is
end behavioral_tb;

architecture tb of behavioral_tb is
component HA_behav
 port (a, b: in std_logic;
 sum, carry_out: out std_logic);
end component;

signal a, b: std_logic;
signal sum, carry_out: std_logic;
constant period : time := 100 ns;

begin
  uut: HA_behav port map (
  a=>a, b=>b, sum=>sum,
```

```
  carry_out=>carry_out
  );

  clock: process
  begin
      a <= '0';
      b <= '0';
      wait for period;
      a <= '0';
      b <= '1';
      wait for period;
      a <= '1';
      b <= '0';
      wait for period;
      a <= '1';
      b <= '1';
      wait for period;
  end process;
end;
```

Testbench for Structural

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

entity structural_tb is
end structural_tb;

architecture tb of structural_tb is
component HA_struct
 port (a, b: in std_logic;
 s, c: out std_logic);
end component;

signal a, b: std_logic;
signal s, c: std_logic;
constant period : time := 100 ns;

begin
```

```
  uut: HA_struct port map (
  a=>a, b=>b, s=>s, c=>c
  );

  clock: process
  begin
      a <= '0';
      b <= '0';
      wait for period;
      a <= '0';
      b <= '1';
      wait for period;
      a <= '1';
      b <= '0';
      wait for period;
      a <= '1';
      b <= '1';
      wait for period;
  end process;
end;
```

2.4 RTL Schematic: -

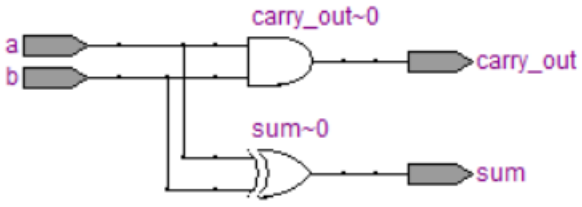


Figure 5: RTL view of Dataflow Half Adder

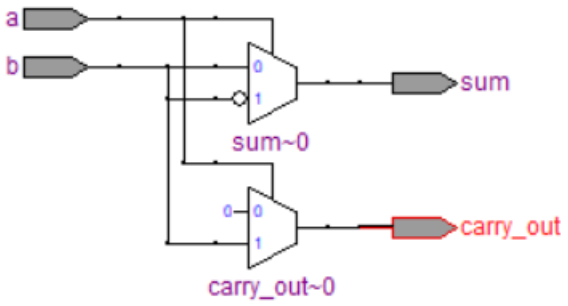


Figure 6: RTL view of Behavioural Half Adder

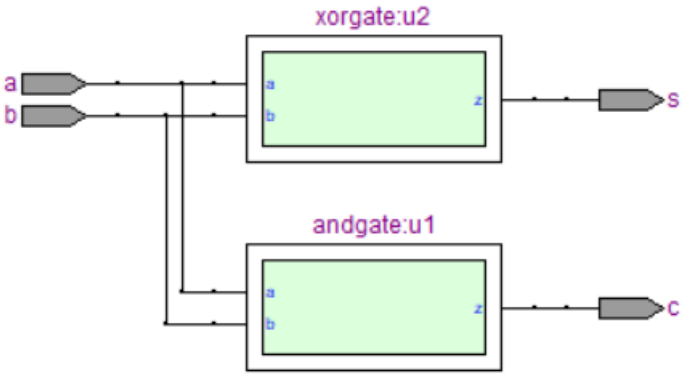


Figure 7: RTL view of Structural Half Adder

2.5 Simulation Waverform: -

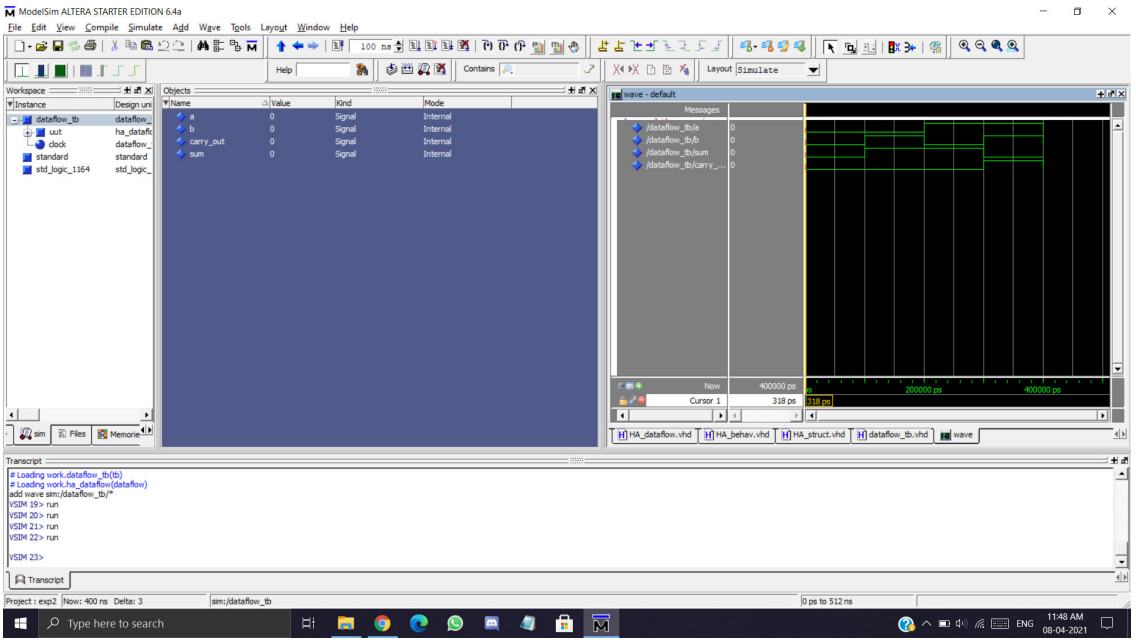


Figure 8: Simulation Waveform of Data flow Model

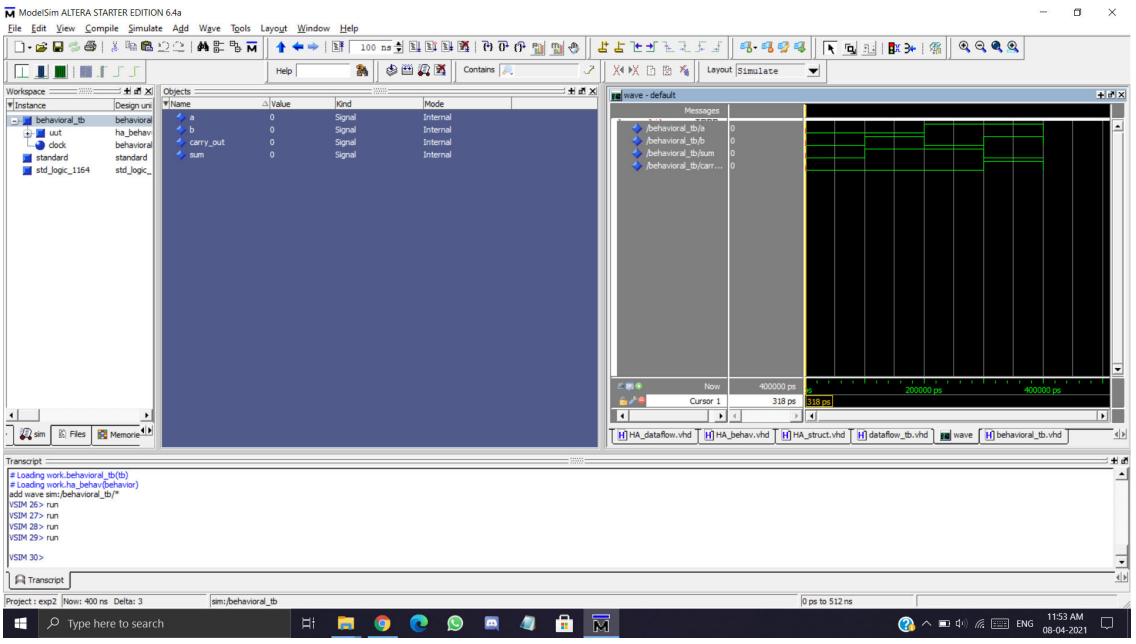


Figure 9: Simulation Waveform of Behavioral Model

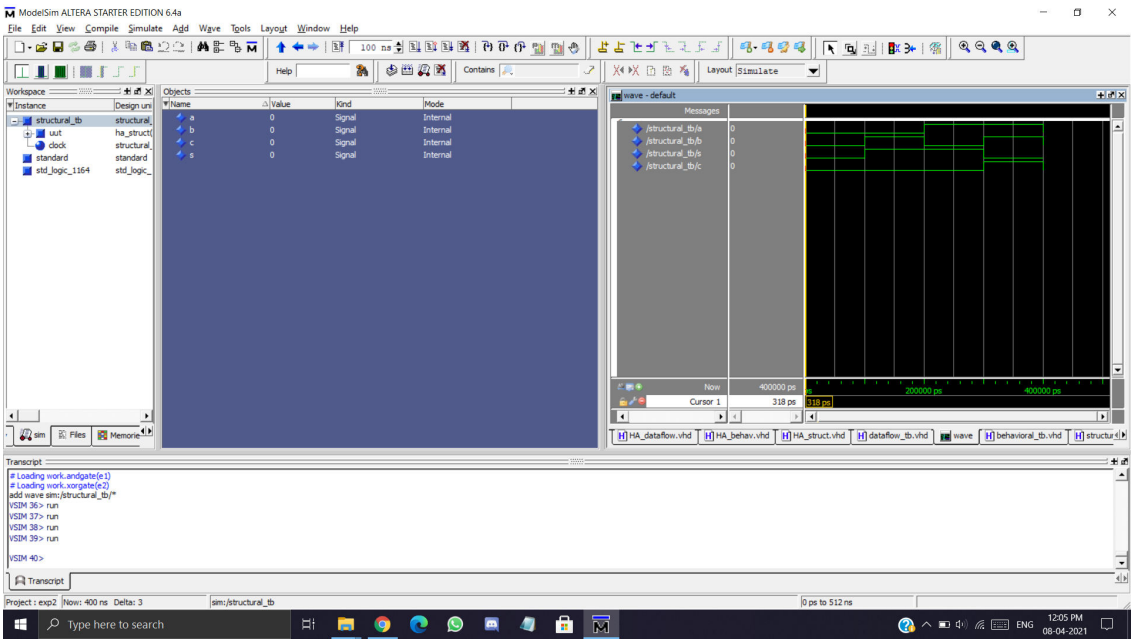


Figure 10: Simulation Waveform of Structural Model

2.6 Conclusion: All three models i.e. data flow, behavioral and structural of half adder were implemented using VHDL in Quartus II 9.0 and test bench of each model was simulated in ModelSim Altera-6.4a. The waveform obtained for data flow, behavioral, structural model simulation is shown in figure 6, figure 7, figure 8 respectively.

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

3.1 Theory:

Concurrent Statement

Concurrent statements are executed when one or more of the signals on the right hand side change their value (i.e. an event occurs on one of the signals). Let us consider an example

```
entity XNOR2 is
    port (A, B: in std_logic;
          Z: out std_logic);
end XNOR2;

architecture behavioral_xnor of XNOR2 is
    -- signal declaration (of internal signals X, Y)
    signal X, Y: std_logic;
    begin
        X <= A and B;
        Y <= (not A) and (not B);
        Z <= X or Y;
    end behavioral_xnor;
```

For instance, when the input A changes, the internal signals X and Y change values that in turn causes the last statement to update the output Z. There may be a propagation delay associated with this change. Digital systems are basically data-driven and an event which occurs on one signal will lead to an event on another signal, etc. The execution of the statements is determined by the flow of signal values. As a result, the order in which these statements are given does not matter (i.e., moving the statement for the output Z ahead of that for X and Y does not change the outcome).

Sequential Statement

A process statement is the main construct that allows you to use sequential statements to describe the behavior of a system over time. A *process* is declared within an architecture and is a *concurrent* statement. However, the statements inside a process are executed *sequentially*.

The syntax for a process statement is

```
[process_label:] process [ (sensitivity_list) ] [is]
    [ process_declarations]
begin
    list of sequential statements
end process [process_label];
```

The sensitivity list is a set of signals to which the process is sensitive. Any change in the value of the signals in the sensitivity list will cause immediate execution of the process. Sequential statements are:

- variable assignments
- case statement

```
case expression is
    when choices =>
        sequential statements
    when choices =>
        sequential statements
        -- branches are allowed
    [ when others => sequential statements ]
end case;
```

- if statement

```
if condition then
    sequential statements
[elsif condition then
    sequential statements ]
[else
    sequential statements ]
end if;
```

- loop statement

```
[ loop_label :]iteration_scheme loop
    sequential statements
    [next  [label] [when condition];
    [exit  [label] [when condition];
end loop [loop_label];
```

- wait statement

In this experiment we designed **4:1 MUX** using **concurrent** (with select when, when else, for loop) as well as **sequential** (if else, case is when) statements.

3.2 Codes:

if 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 ='0' 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;
```

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;
```

when else

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

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;
```

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 others;
end m1;
```

for loop

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

entity ForLoop is
port(
  a, b, c: in std_logic_vector(1 downto 0);
  sum, carry: out std_logic_vector(1 downto 0)
);
end ForLoop;

architecture fl1 of 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:

Test bench for if else

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

entity ifelse_tb is
end ifelse_tb;

architecture tb of ifelse_tb is
component Mux4to1_ifelse
port(
 A,B,C,D : in STD_LOGIC;
 S0,S1: in STD_LOGIC;
 Z: out STD_LOGIC
);
end component;

signal A,B,C,D,S0,S1,Z: std_logic;
constant period : time := 100 ns;

begin
  uut: Mux4to1_ifelse port map (
  A=>A, B=>B, C=>C, D=>D, S0=>S0, S1=>S1, Z=>Z
  );
  clock: process
  begin
      A <= '1';
      B <= '1';
      C <= '0';
      D <= '1';
```

```
      S0 <= '0';
      S1 <= '0';
      wait for period;
      S0 <= '1';
      S1 <= '0';
      wait for period;
      S0 <= '0';
      S1 <= '1';
      wait for period;
      S0 <= '1';
      S1 <= '1';
      wait for period;

      A <= '0';
      B <= '1';
      C <= '1';
      D <= '0';

      S0 <= '0';
      S1 <= '0';
      wait for period;
      S0 <= '1';
      S1 <= '0';
      wait for period;
      S0 <= '0';
      S1 <= '1';
      wait for period;
      S0 <= '1';
      S1 <= '1';
      wait for period;
  end process;
end;
```

Test bench for case

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

3.3.0

```
entity case_tb is
end case_tb;

architecture tb of 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) := "1101";
signal sel: std_logic_vector(1 downto 0);
signal data_out: std_logic;
constant period : time := 100 ns;

begin
  uut: Mux4to1_case port map (
  data_in=>data_in, sel=>sel, data_out=>data_out
  );
  clock: process
  begin
      sel <= "00";wait for period;
      sel <= "01";wait for period;
      sel <= "10";wait for period;
      sel <= "11";wait for period;

      data_in <= "0101";

      sel <= "00";wait for period;
      sel <= "01";wait for period;
      sel <= "10";wait for period;
      sel <= "11";wait for period;

  end process;
end;
```

Test bench for when else

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

entity whenelse_tb is
end whenelse_tb;

architecture tb of whenelse_tb is
component Mux4to1_whenelse
 port(
 D : in STD_LOGIC_VECTOR(0 to 3);
 S : in STD_LOGIC_VECTOR(0 to 1);
 O : out STD_LOGIC
 );
end component;

signal D: std_logic_vector(0 to 3) := "1101";
signal S: std_logic_vector(0 to 1);
signal O: std_logic;
constant period : time := 100 ns;

begin
  uut: Mux4to1_whenelse port map (
  D=>D, S=>S, O=>O
  );
  clock: process
  begin
      S <= "00";wait for period;
      S <= "01";wait for period;
      S <= "10";wait for period;
      S <= "11";wait for period;

      D <= "0101";

      S <= "00";wait for period;
      S <= "01";wait for period;
      S <= "10";wait for period;
      S <= "11";wait for period;
```

```
  end process;
end;
```

Test bench for with select

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

entity withselect_tb is
end withselect_tb;

architecture tb of withselect_tb is
component Mux4to1_withselect
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 component;

signal d0,d1,d2,d3: std_logic;
signal s: std_logic_vector(1 downto 0);
signal y: std_logic;
constant period : time := 100 ns;


begin
  uut: Mux4to1_withselect port map(
  d0=>d0, d1=>d1, d2=>d2, d3=>d3, s=>s, y=>y
  );
  clock: process
  begin
  d0 <= '1';
  d1 <= '1';
  d2 <= '0';
  d3 <= '1';
```

```
  S <= "00";wait for period;
  S <= "01";wait for period;
  S <= "10";wait for period;
  S <= "11";wait for period;

  d0 <= '0';
  d1 <= '1';
  d2 <= '1';
  d3 <= '0';

  S <= "00";wait for period;
  S <= "01";wait for period;
  S <= "10";wait for period;
  S <= "11";wait for period;
  end process;
end;
```

Test bench for "for" loop

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

entity for_tb is
end for_tb;

architecture tb of for_tb is
component 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, sum, carry: std_logic_vector(1 downto 0);
constant period : time := 100 ns;

begin
```

```
  uut: ForLoop port map(
  a=>a, b=>b, c=>c, sum=>sum, carry=>carry
  );
  clock: process
  begin
    wait for period;
    a <= "11";
    b <= "01";
    c <= "01";
    wait for period;
  end process;
end;
```

3.4 RTL Schematic: -

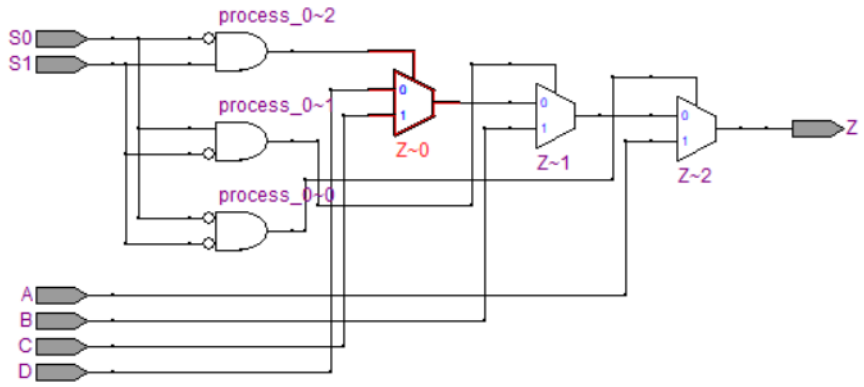


Figure 11: RTL view of 4:1 Mux using If else

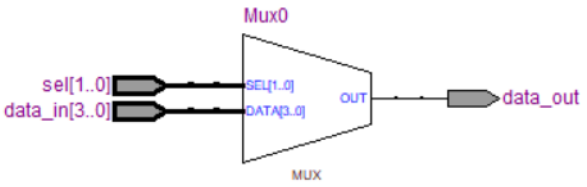


Figure 12: RTL view of 4:1 Mux using case when

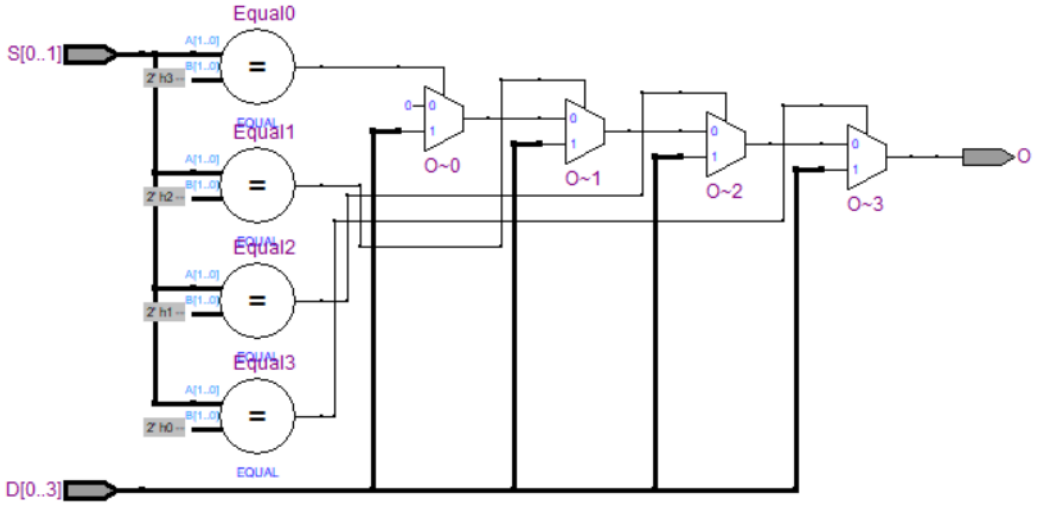


Figure 13: RTL view of 4:1 Mux using When else

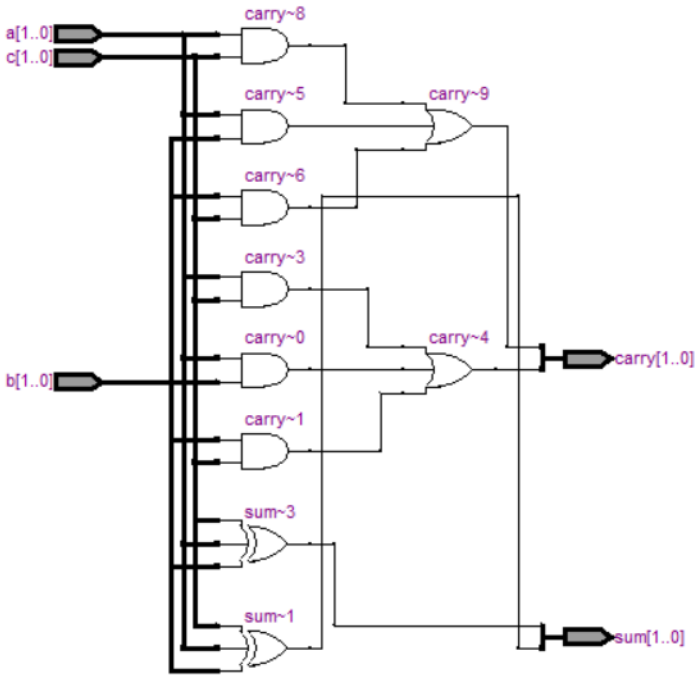


Figure 14: RTL view of 2 bit adder using Forloop

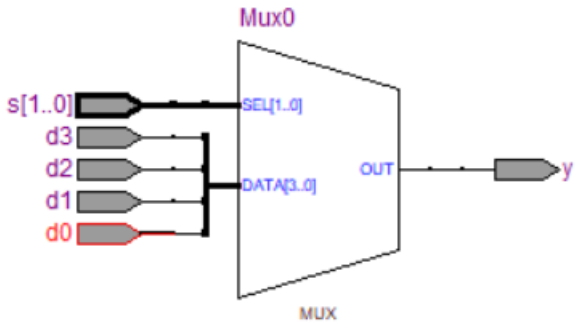


Figure 15: RTL view of 4:1 Mux using with select

3.5 Simulation Waverform: -

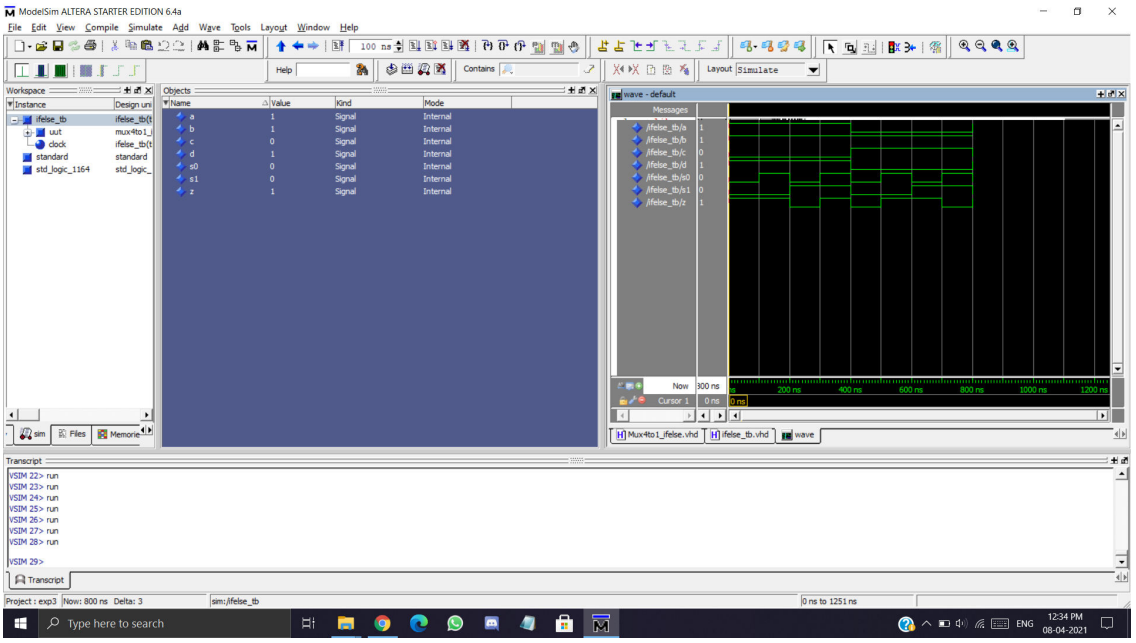


Figure 16: Waveform for **4:1 MUX** using **if-else** statement

3.5

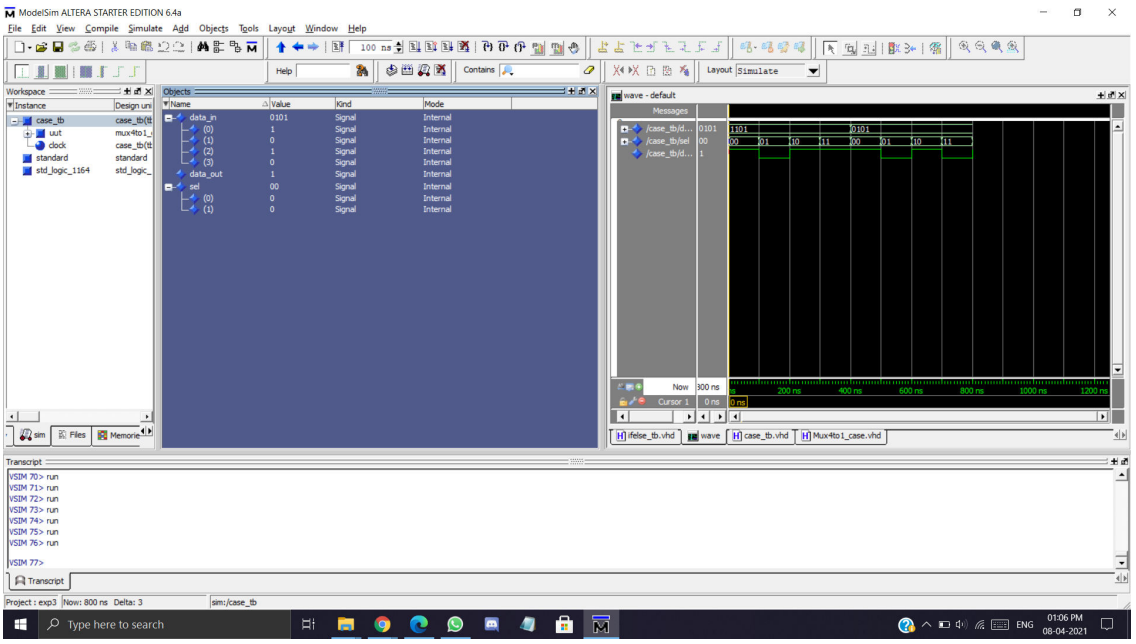


Figure 17: Waveform for **4:1 MUX** using **case** statement

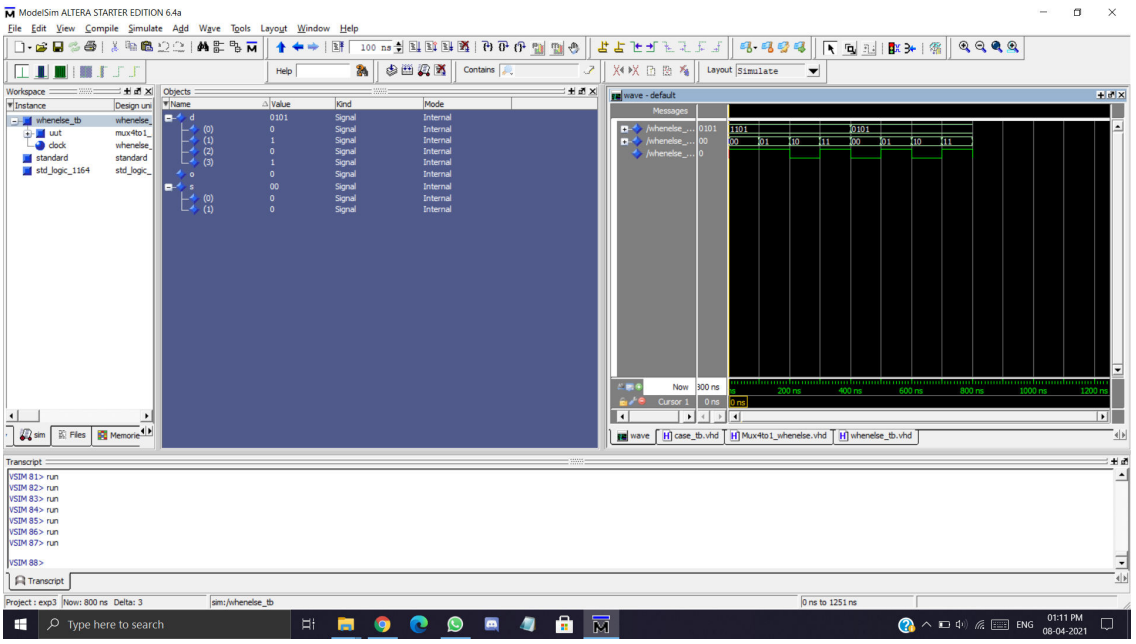


Figure 18: Waveform for **4:1 MUX** using **when-else** statement

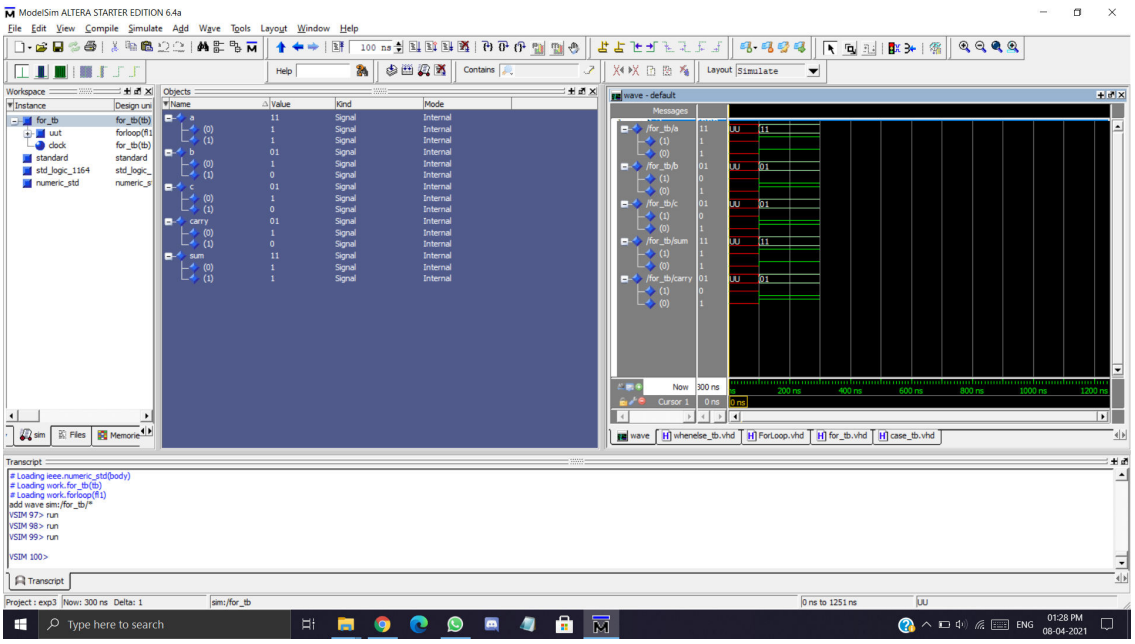


Figure 19: Waveform for **2 bit full adder** using **for generate** statement

3.6 Conclusion: 4:1 Multiplexer was designed, synthesized and simulated using Quartus II 0.0 and ModelSim Altera-6.4a. 4:1 MUX was implemented using concurrent and sequential statements. Test bench for each implementation was written and simulated in ModelSim to obtain the waveform shown in section above.

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

4.1 Theory:

A **shift register** is made up of many single-bit "D-Type Data Latches," one for each data bit, either a logic "0" or logic "1," linked in a serial type daisy-chain structure, with the output of one latch being the input of the next, and so on. Shift Registers are used in calculators and computers for data storage or data movement. They are commonly used to store data such as two binary numbers until they are connected together, or to convert data from serial to parallel or parallel to serial formats.

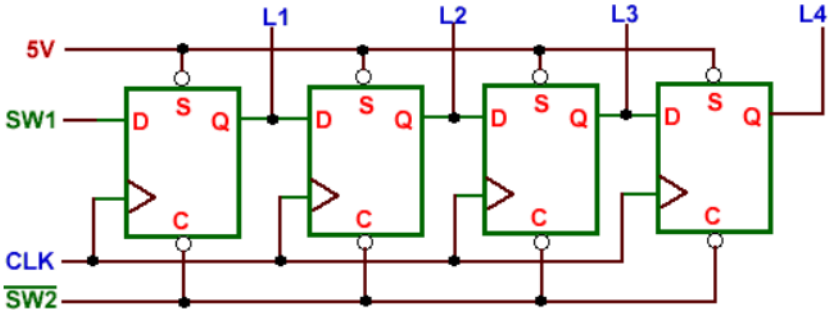


Figure 20: 4 bit Shift Register

Counters use sequential logic to count clock pulses. A counter can be implemented implicitly with a Register Inference. The Quartus II software can infer a counter from an If Statement that specifies a clock edge together with logic that adds or subtracts a value from the signal or variable. The If Statement and additional logic should be inside a Process Statement.

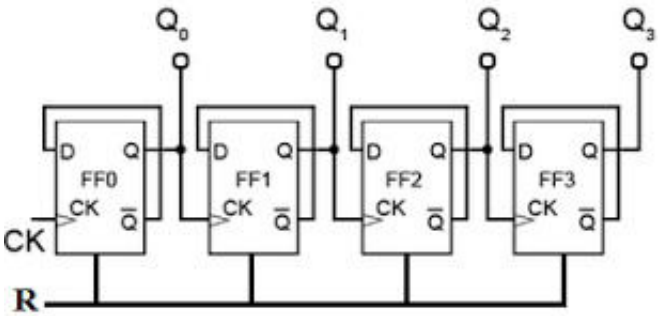


Figure 21: 4 bit Up Counter

4.2 Codes:

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_sync is
port
(
    clk : in std_logic;
    reset : in std_logic;
    enable : in std_logic;
    q : out integer range 0 to 255
);
end counter_sync;

architecture rtl of counter_sync 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;
```

Up-counter with asynchronous clear

```
--8-bit unsigned up counter with asynchronous clear & clock enable
--counter with asynchronous reset (active low) and enable
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

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

architecture bhv of counter_async 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
                Qt <= Qt + 1;
            end if;
        end if;
    end process;
    Q <= Qt;
end bhv;
```

Shift register with synchronous reset

```
--8 bit 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;
```

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;
```

4.3 Testbench:

Testbench for Up-counter with synchronous clear

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

entity counter_sync_tb is
end entity;

architecture tb of counter_sync_tb is
component counter_sync is
Port ( clock,reset,enable : in STD_LOGIC;
       q : out integer range 0 to 255);
end component;

signal clock: STD_LOGIC := '0';
signal reset,enable : STD_LOGIC;
signal q : integer range 0 to 255;
constant period: time:= 100ns;

begin
    uut: counter_sync port map(
    clock => clock,
    reset => reset,
    enable => enable,
    q=>q);
    clk: process
    begin
```

```
        clock <= '1';
        enable <= '0';
        reset <= '1';
        wait for period/2;
        clock <= '0';
        reset <= '0';
        enable <= '1';
        wait for period/2;

        for i in 0 to 9 loop
            clock <= not clock;
            wait for period/2;
        end loop;
    end process;
end tb;
```

Testbench for Up-counter with asynchronous clear

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

entity counter_async_tb is
end entity;

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

signal clock: STD_LOGIC := '0';
signal resetn,E : STD_LOGIC;
signal Q : integer range 0 to 255;
constant period: time := 100 ns;

begin
    uut: counter_async port map(
```

```
    clock => clock,
    resetn => resetn,
    E => E,
    Q=>Q);
    clk: process
    begin
        resetn <= '1';
        clock <= '1';
        wait for period/2;
        resetn <= '0';
        clock <= '0';
        E <= '1';
        wait for period/2;

        for i in 0 to 9 loop
            clock <= not clock;
            wait for period/2;
        end loop;
    end process;
end tb;
```

Testbench for Shift register with synchronous reset

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

entity sipo_sync_tb is
end sipo_sync_tb;

architecture tb of sipo_sync_tb is
component sipo_syncreset
port(
    clk, clear : in std_logic;
    Input_Data: in std_logic;
    Q: buffer std_logic_vector(7 downto 0)
);
end component;
```

```
signal clk, clear: std_logic;
signal Input_data: std_logic;
signal Q: std_logic_vector(7 downto 0);
constant period: time := 100ns;

begin
    uut: sipo_syncreset port map(
        clk=>clk, clear=>clear,
        Input_data=>Input_data, Q=>Q
    );

    clock: process
    begin
        clk <= '1';
        wait for period/2;
        clk <= '0';
        wait for period/2;
    end process;

    simu: process
    begin
        clear <= '1';
        wait for period/2;
        clear <= '0';
        wait for period/2;

        Input_data <= '1';
        wait for period;
        Input_data <= '1';
        wait for period;
        Input_data <= '0';
        wait for period;
        Input_data <= '1';
        wait for period;
        Input_data <= '0';
        wait for period;
        Input_data <= '1';
        wait for period;
        Input_data <= '0';
```

```
        wait for period;
        Input_data <= '1';
        wait for period;
    end process;
end;
```

Testbench for Shift register with asynchronous reset

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

entity sipo_async_tb is
end sipo_async_tb;

architecture tb of sipo_async_tb is
component sipo_asyncreset
port(
    clk, clear : in std_logic;
    Input_Data: in std_logic;
    Q: buffer std_logic_vector(3 downto 0)
);
end component;

signal clk, clear: std_logic;
signal Input_data: std_logic;
signal Q: std_logic_vector(3 downto 0);
constant period: time := 100ns;

begin
    uut: sipo_asyncreset port map(
        clk=>clk, clear=>clear,
        Input_data=>Input_data, Q=>Q
    );

    clock: process
    begin
        clk <= '1';
        wait for period/2;
```

```
        clk <= '0';
        wait for period/2;
    end process;

    simu: process
    begin
        clear <= '1';
        wait for period/2;
        clear <= '0';
        wait for period/2;


        Input_data <= '1';
        wait for period;
        Input_data <= '1';
        wait for period;
        Input_data <= '0';
        wait for period;
        Input_data <= '1';
        wait for period;
    end process;
end;
```

4.4 RTL Schematic:

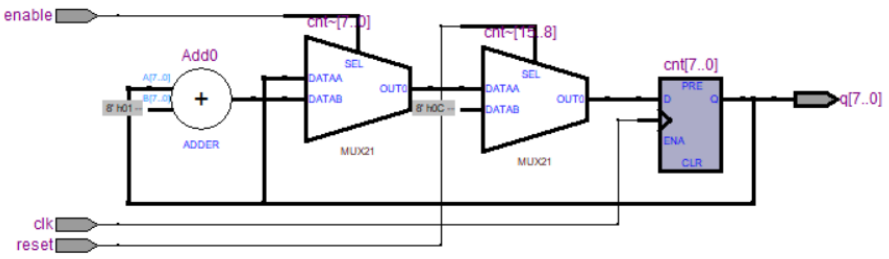


Figure 22: Synchronous counter

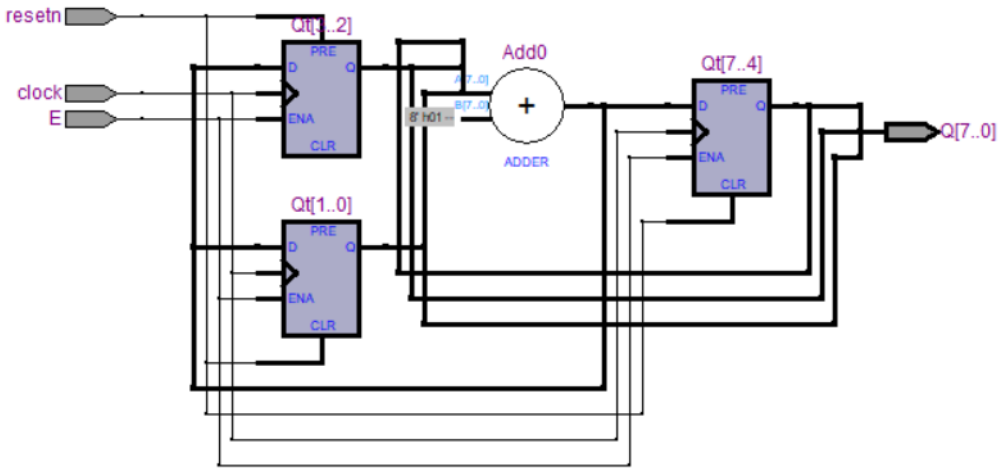


Figure 23: asynchronous counter

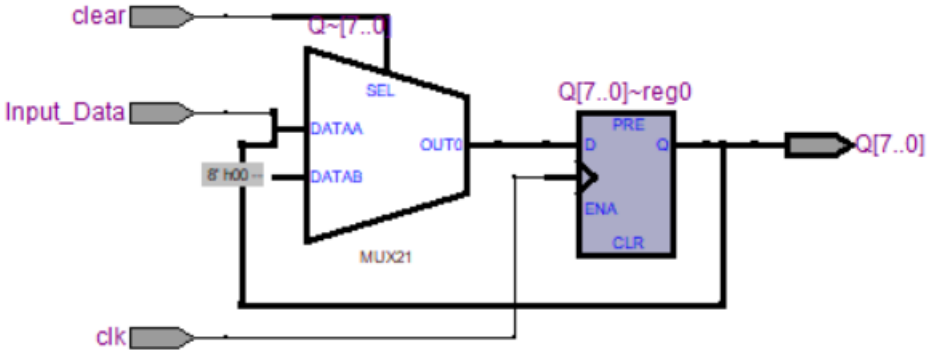


Figure 24: 8 bit Shift Register with Synchronous Reset

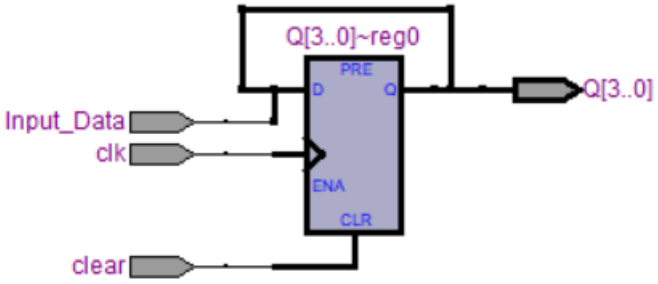


Figure 25: 4 bit Shift Register with Asynchronous Reset

4.5 Simulation Waverform:

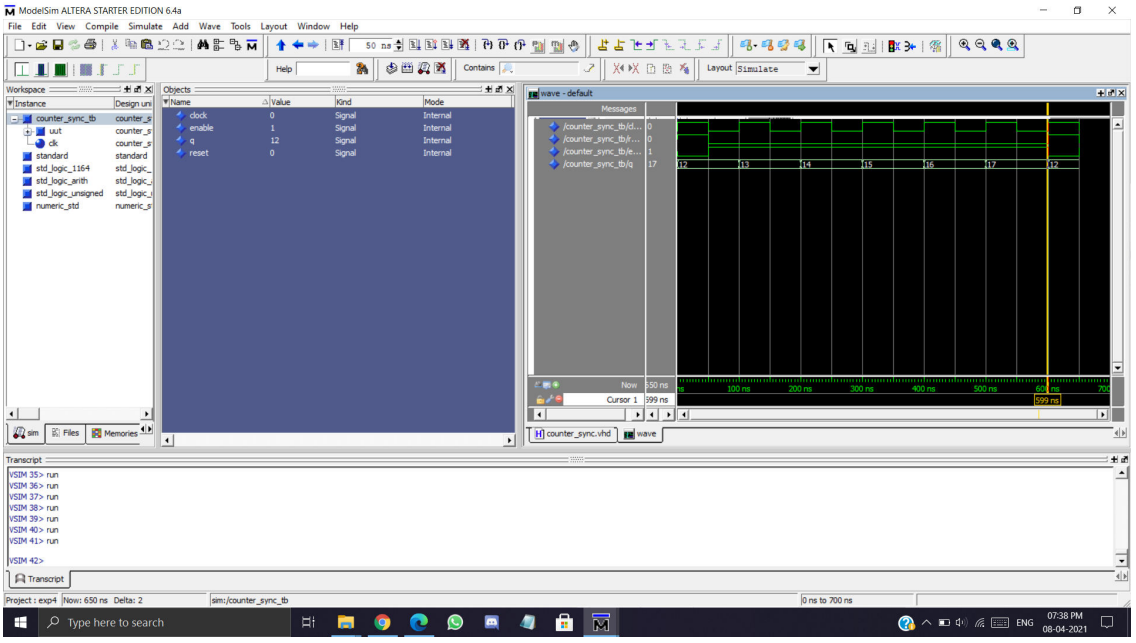


Figure 26: Waveform for **Synchronous Up counter**

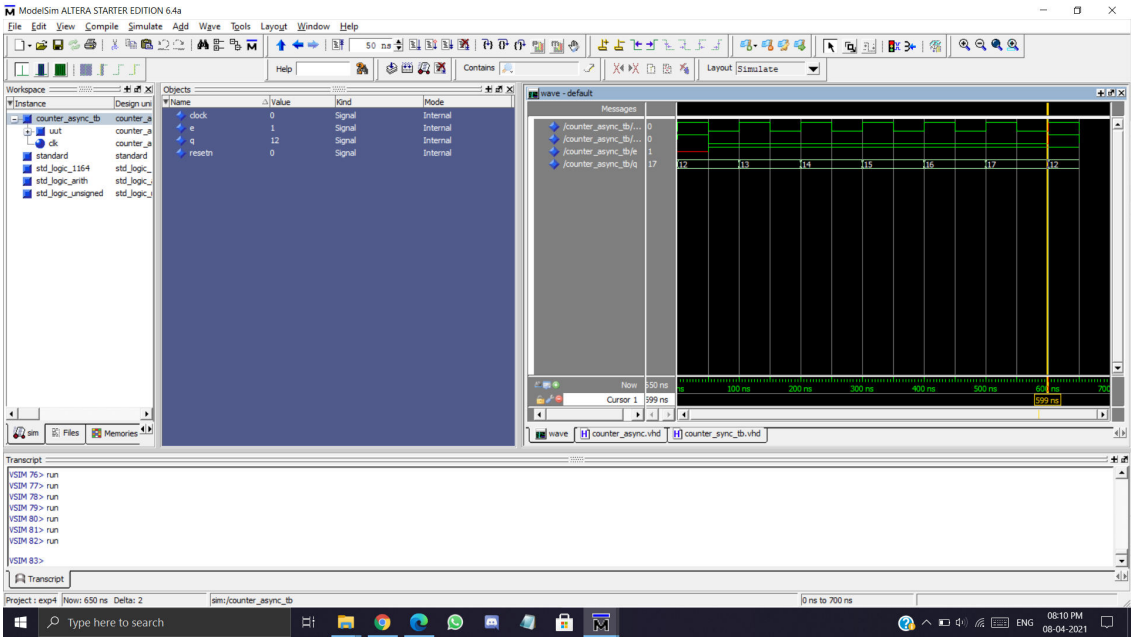


Figure 27: Waveform for **Up counter** with **Asynchronous Clear**

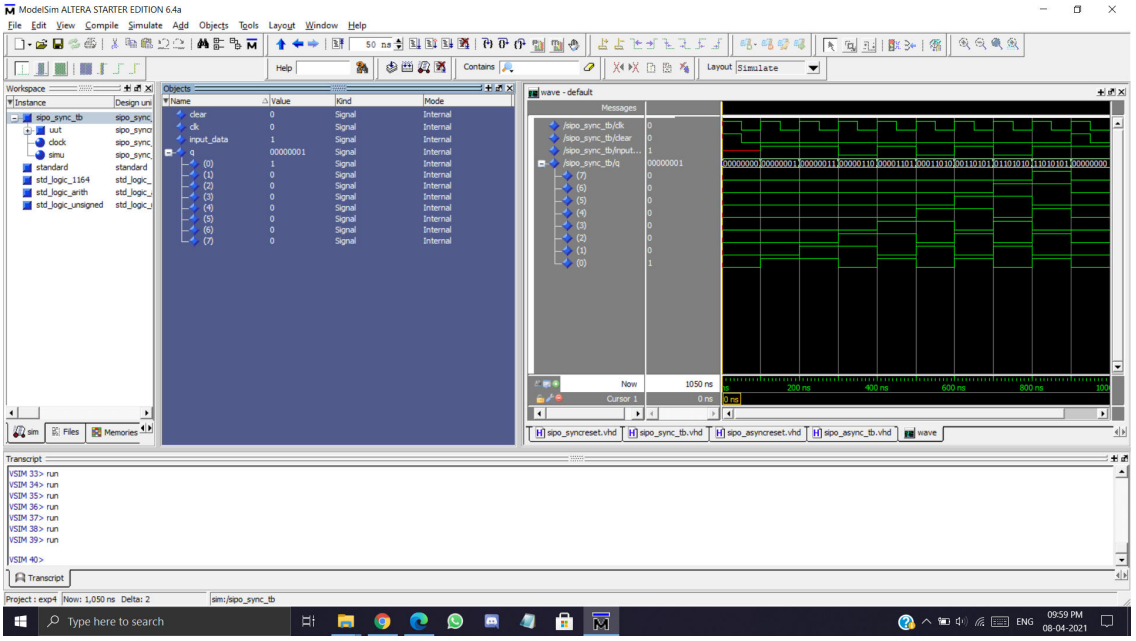


Figure 28: Waveform for **8 bit Shift Register** with **synchronous reset**

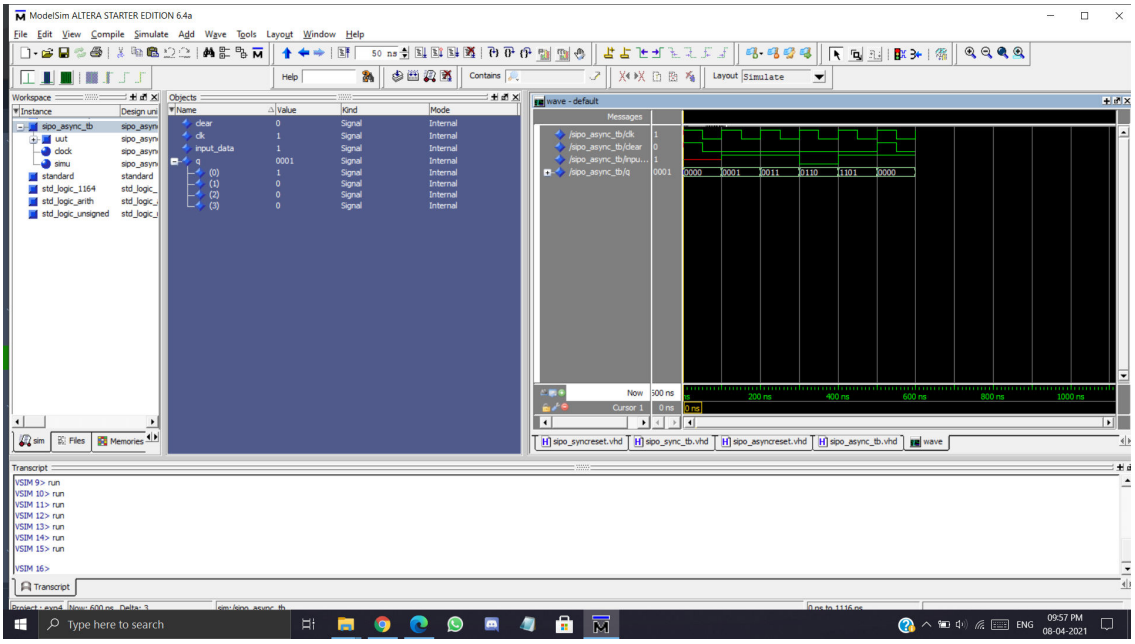


Figure 29: Waveform for **4 bit shift register** with **asynchronous reset**

4.6 Conclusion:

- **Synchronous Up-counter**: This counter counts sequentially on every clock pulse when enable is high from 0 to 255. The process block triggers when there is event on clk. The counter resets only when reset and clk both are high.
- **Asynchronous Up-counter**: This counter counts sequentially on every rising edge of clock provided reset is low and E is high. Here process is triggered whenever there occurs event on any one of reset, clock and E. The counter resets when reset is high, there is no need of clock to be high hence known as counter with asynchronous reset.
- **Shift Register with synchronous reset**: Whenever there is event on clk and clear is high, register is cleared and initialized to "00000000". If clear is low then value of register is shifted right by 1 bit and LSB is loaded with present input.
- **Shift Register with asynchronous reset**: It is similar to above one only difference is reset is not synchronized with clk i.e. if reset is high register is initialized to "00000000" irrespective of clock, high or low.

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

5.1 Theory: A finite-state machine (FSM) or finite state automaton (FSA) is a mathematical model of computation. It's a machine that can only be in one of a finite number of states at any given moment. In response to such inputs, the FSM will change from one state to another; this change is referred to as a transition.

If a synchronous sequential circuit has a finite number of states, it is also known as a Finite State Machine (FSM). FSMs is divided into two categories.

- Mealy State Machine
- Moore State Machine

A **Finite State Machine** is said to be **Mealy state machine**, if outputs depend on both present inputs present states.

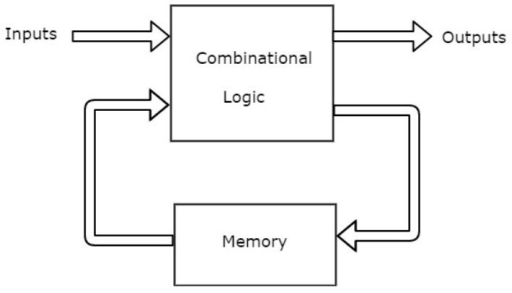


Figure 30: Mealy Block diagram

A **Finite State Machine** is said to be **Moore state machine**, if outputs depend only on present states.

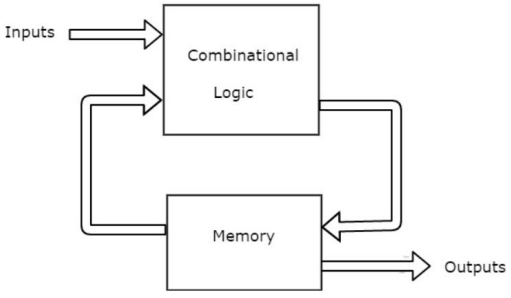


Figure 31: Moore block diagram

5.2 Codes:

Mealy machine to detect 11

```
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;
```

Moore machine to detect 11

```
--moore machine seq detector 11
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;
        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;
```

5.3 Testbench:

TB for mealy machine to detect 11

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

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
    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';
        wait for 100 ps;
    end process;
END;
```

TB for moore machine to detect 11

```
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';
--Outputs
signal 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 <= '0';
        wait for 100 ps;
        din <= '1';
    end process;
END;
```

5.4 RTL Schematic:

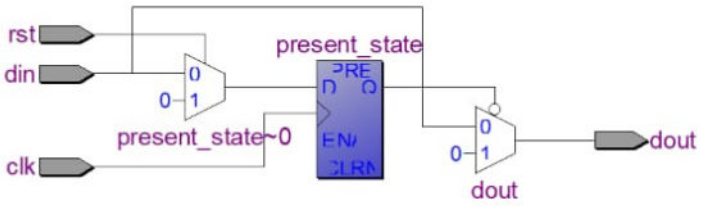


Figure 32: RTL view of **Mealey Machine**

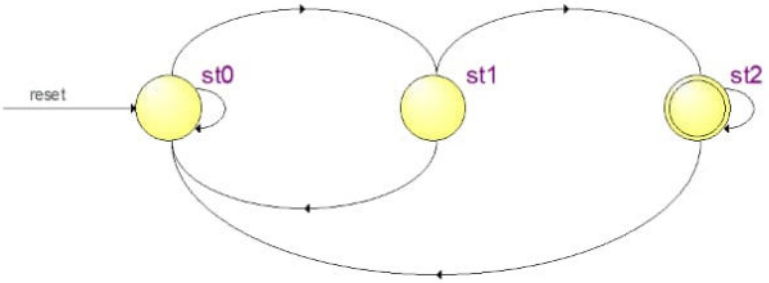


Figure 33: FSM view of **Moore Machine**

Figure 34: RTL view of **Moore Machine**

5.5 Simulation Waverform:

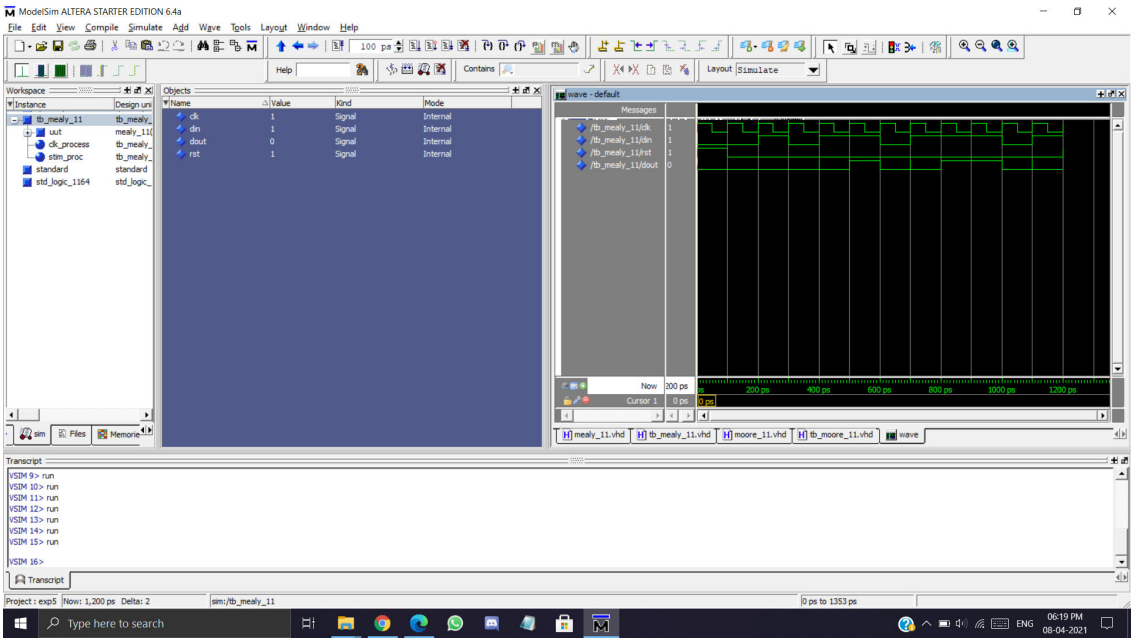


Figure 35: **Mealey machine** waveform to detect **sequence '11'**

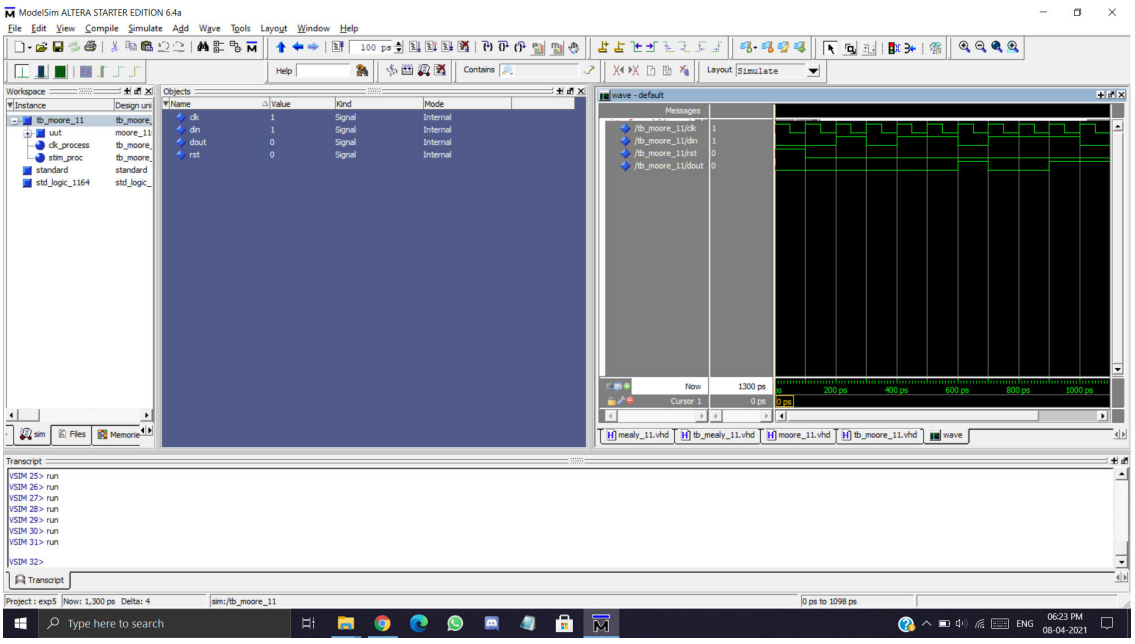


Figure 36: **Moore machine** waveform to detect **sequence '11'**

5.6 Conclusion: Sequence detector was designed, implemented, and simulated using Quartus II 9.0 and ModelSim 6.4a. This detector was used to detect sequence '11'. Both **Moore** and **Mealey** machine were modelled. From this experiment it can be concluded that in moore machine output depends only on present states whereas in case of mealey machine output depends on both present states as well as present inputs.

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

6.1 Theory: A **VHDL package** is a file or module that contains declarations of commonly used objects, data type, component declarations, signal, procedures and functions that can be shared among different VHDL models.

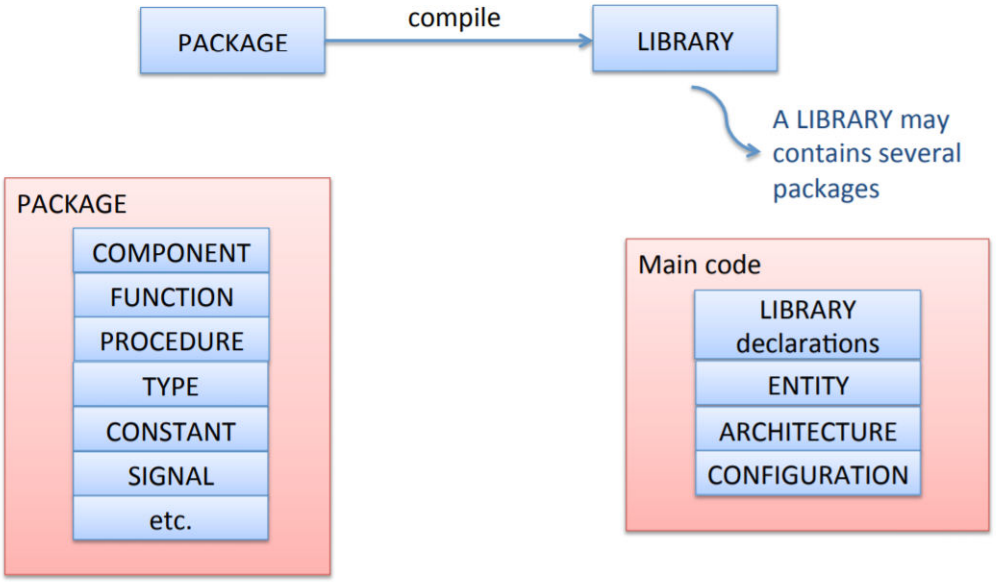


Figure 37:

The IEEE library comes with several packages.

- std_logic_1164 package: defines the standard data types
- std_logic_arith package: provides arithmetic, conversion and comparison functions for the signed, unsigned, integer, std_ulogic, std_logic and std_logic_vector types
- std_logic_unsigned
- std_logic_misc package: defines supplemental types, subtypes, constants and functions for the std_logic_1164 package.

To use any of these one must include the library and use clause:
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

The syntax to declare a package is as follows:
– Package declaration
package name_of_package is
package declarations
end package name_of_package;

– Package body declarations
package body name_of_package is
package body declarations
end package body name_of_package;

6.2 Codes:

1 bit full adder

```
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

```
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:

```
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: -

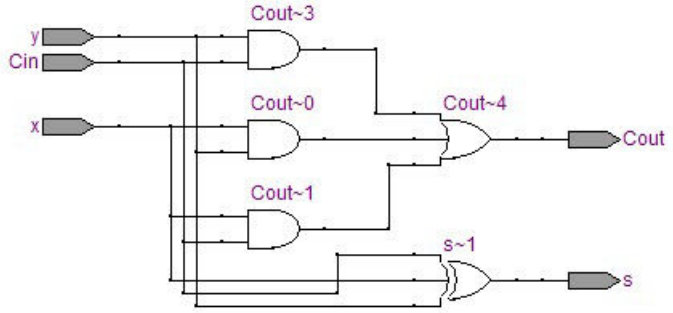


Figure 38: RTL view of **full adder**

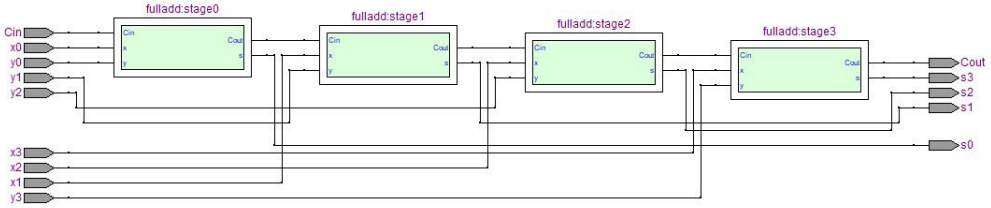


Figure 39: RTL view of **4 bit full adder**

6.5 Simulation Waverform: -

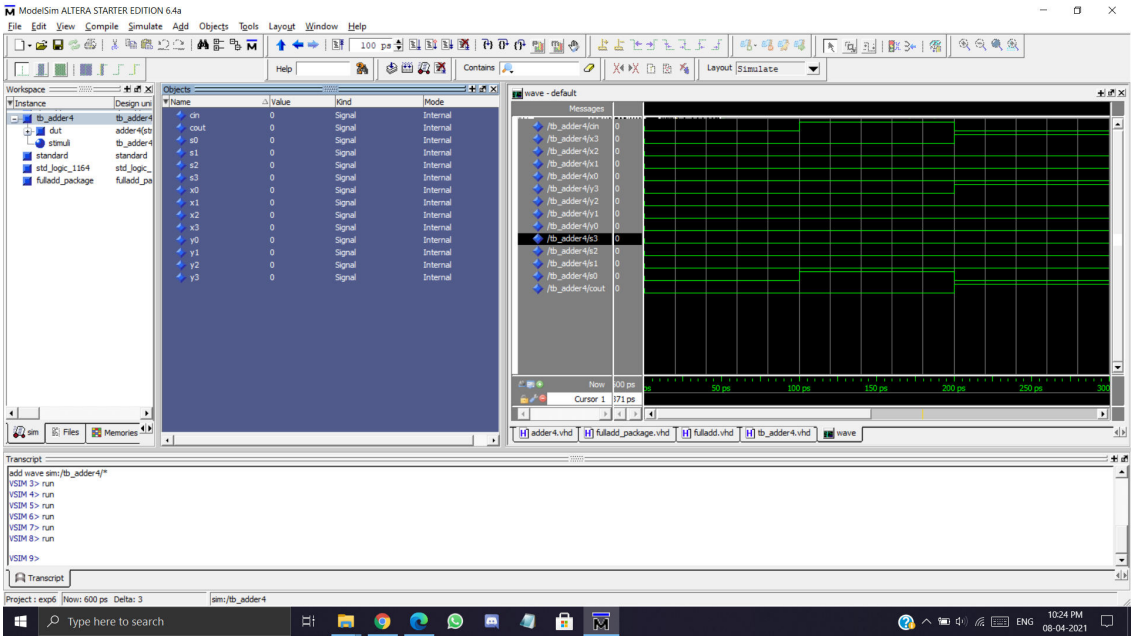


Figure 40: Waveform of **4 bit full adder**

6.6 Conclusion: We made use of the predefined type std_logic that is declared in the package std_logic_1164 while implementing full adder package. This package was compiled and stored in work library. The components of this package was used in implementing 4 bit full adder using "USE work.fulladd_package.all" statement.

Experiment 7 - Introduction to Verilog (19-03-2021)

7.1 Theory: Verilog is a **HARDWARE DESCRIPTION LANGUAGE** (HDL). It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flip−flop. It means, by using a HDL we can describe any digital hardware at any level.

Verilog supports a design at many levels of **abstraction**. The major three are −

- Behavioral level
- Register-transfer level or dataflow
- Gate/Structural level

In this experiment we implemented 1 bit full adder and 4:1 Multiplexer in all three levels of abstraction mentioned above.

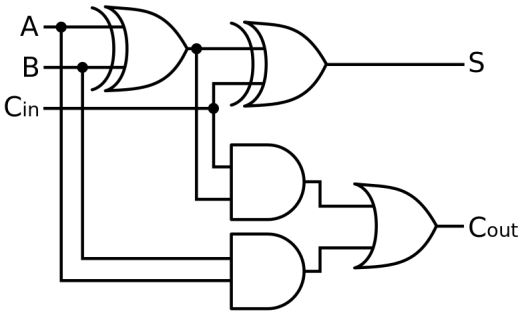


Figure 41: Full Adder

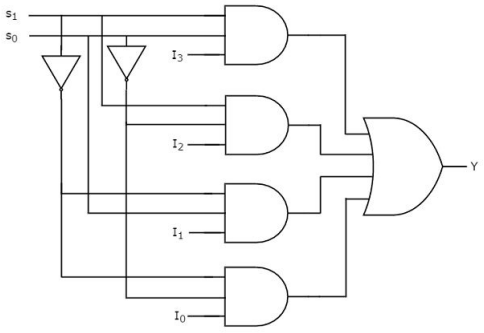


Figure 42: 4:1 Multiplexer

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 behavioral

```
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 Multiplexer dataflow

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

    assign y= s1?(s0?i3:i2) : (s0?i1:i0);

endmodule
```

4:1 Multiplexer behavioral

```
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 Multiplexer 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:

Testbench for full adder dataflow

```
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
```

Testbench for full adder behavioral

```
‘timescale 100ps/ 10ps
module FA_behav_tb();
    reg A,B,Cin;
    wire S,Cout;
    FA_behav 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
```

Testbench for full adder structural

```
‘timescale 10ns/ 10ps
module FA_struct_tb();
    reg A,B,Cin;
    wire S,Cout;
    FA_struct 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;
        #10;
        A = 0;
        B = 0;
        Cin = 1;
        #10;
        A = 0;
```

```
        B = 1;
        Cin = 0;
        #10;
        A = 0;
        B = 1;
        Cin = 1;
        #10;
        A = 1;
        B = 0;
        Cin = 0; #10;
        A = 1;
        B = 0;
        Cin = 1;
        #10;
        A = 1;
        B = 1;
        Cin = 0;
        #10;
        A = 1;
        B = 1;
        Cin = 1;
        #10;
    end
endmodule
```

Common testbench for 4:1 Multiplexer

```
‘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:

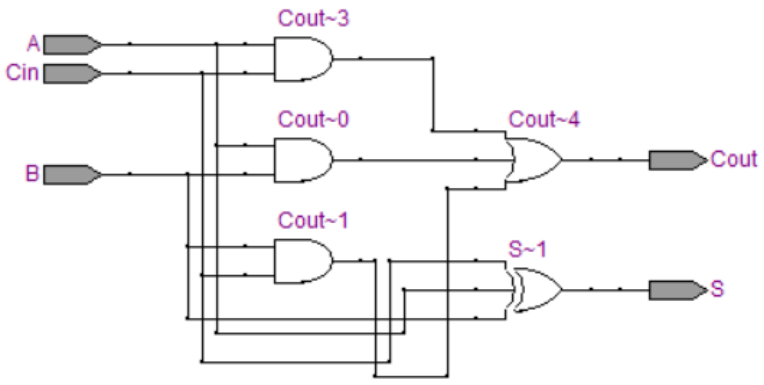


Figure 43: RTL view of **Dataflow Half adder**

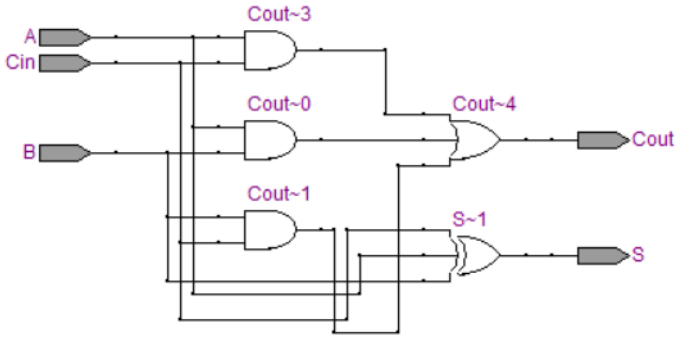


Figure 44: RTL view of **Behaviour Half adder**

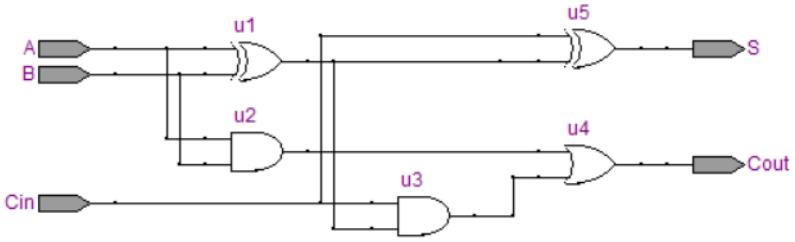


Figure 45: RTL view of **Structural Half adder**

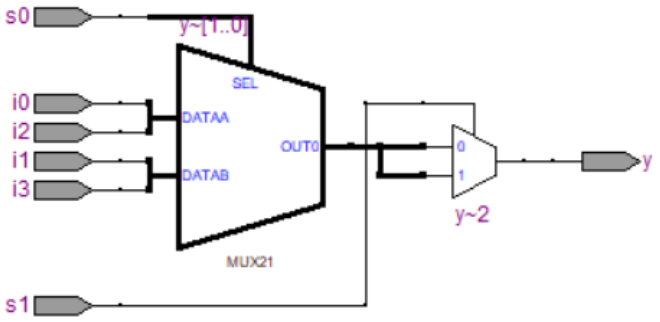


Figure 46: RTL view of **Dataflow 4:1 Mux**

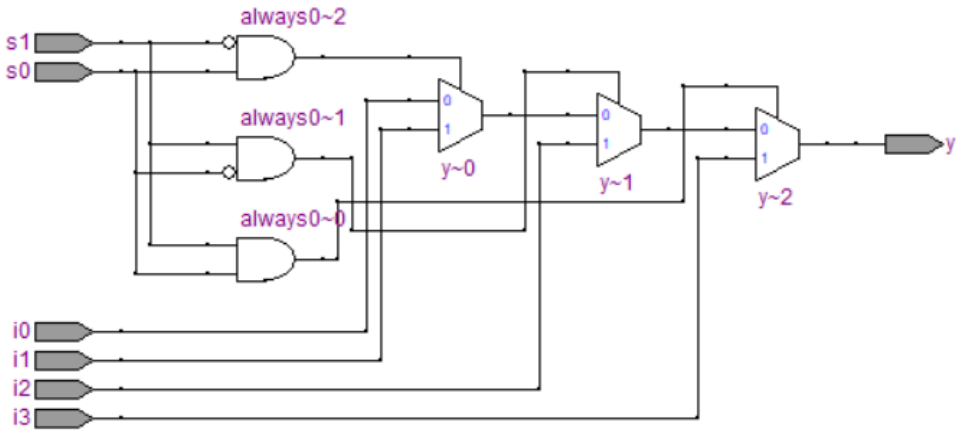


Figure 47: RTL view of **Behaviour 4:1 Mux**

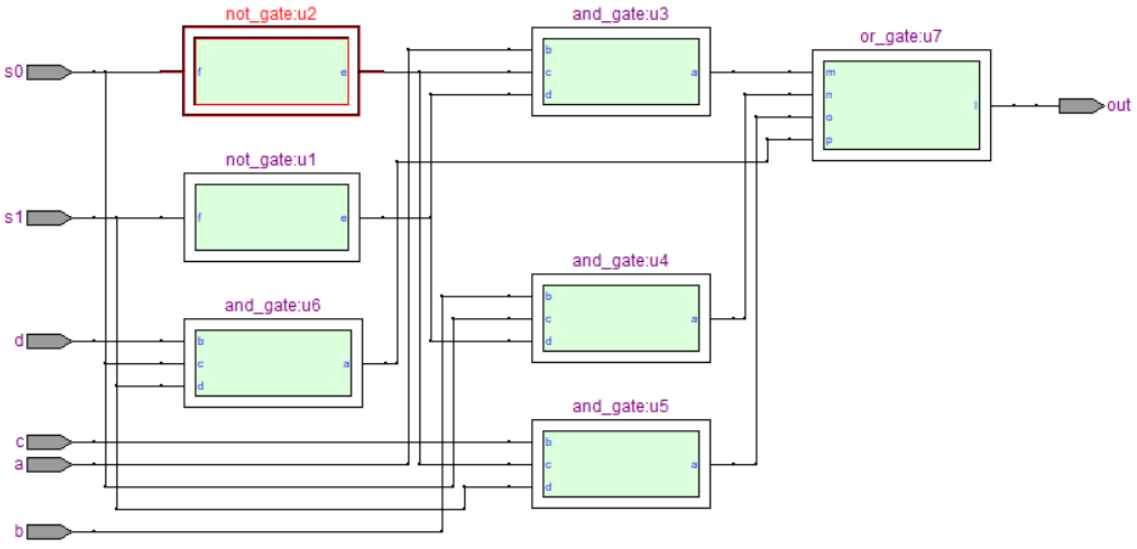


Figure 48: RTL view of **Structural 4:1 Mux**

7.5 Simulation Waverform: -

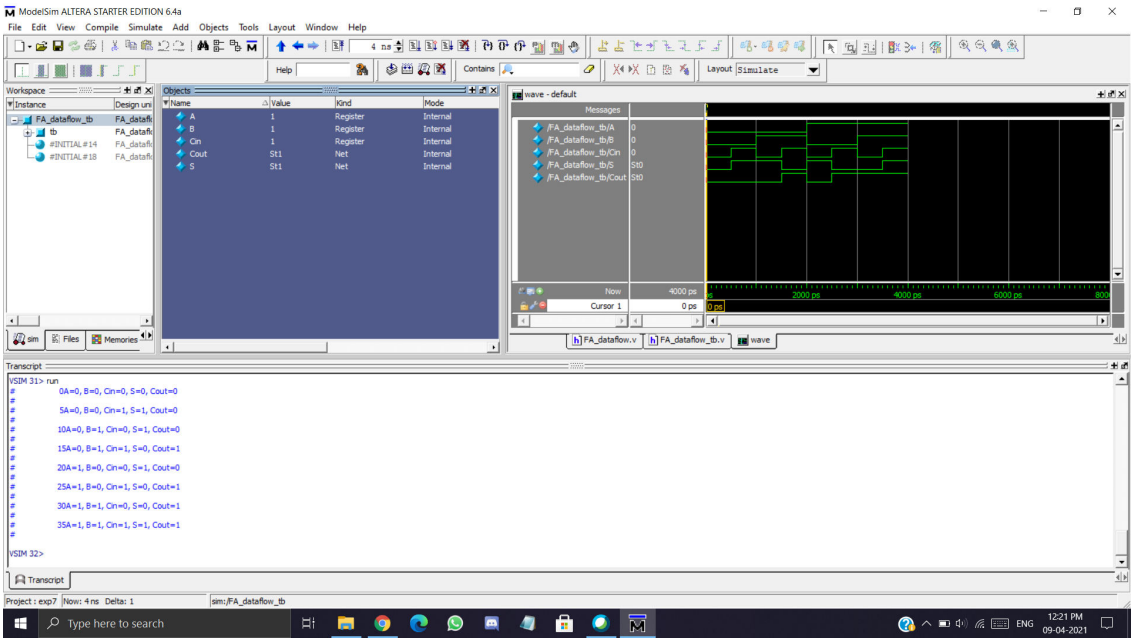


Figure 49: **Dataflow Model** of **full adder**

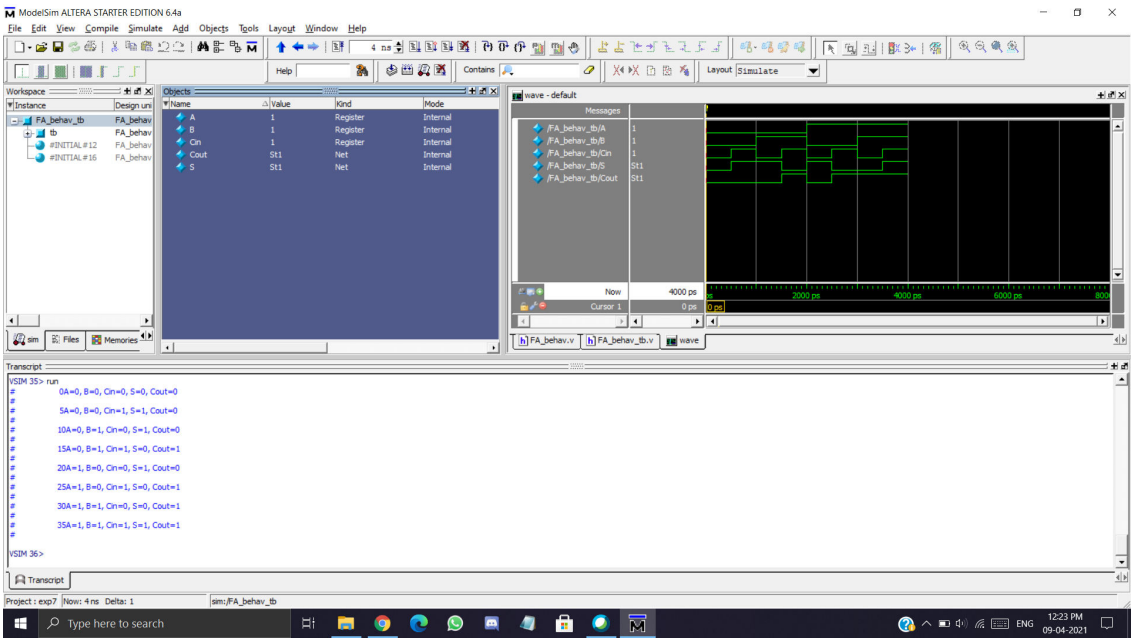


Figure 50: **Behavioral Model** of **full adder**

7.5

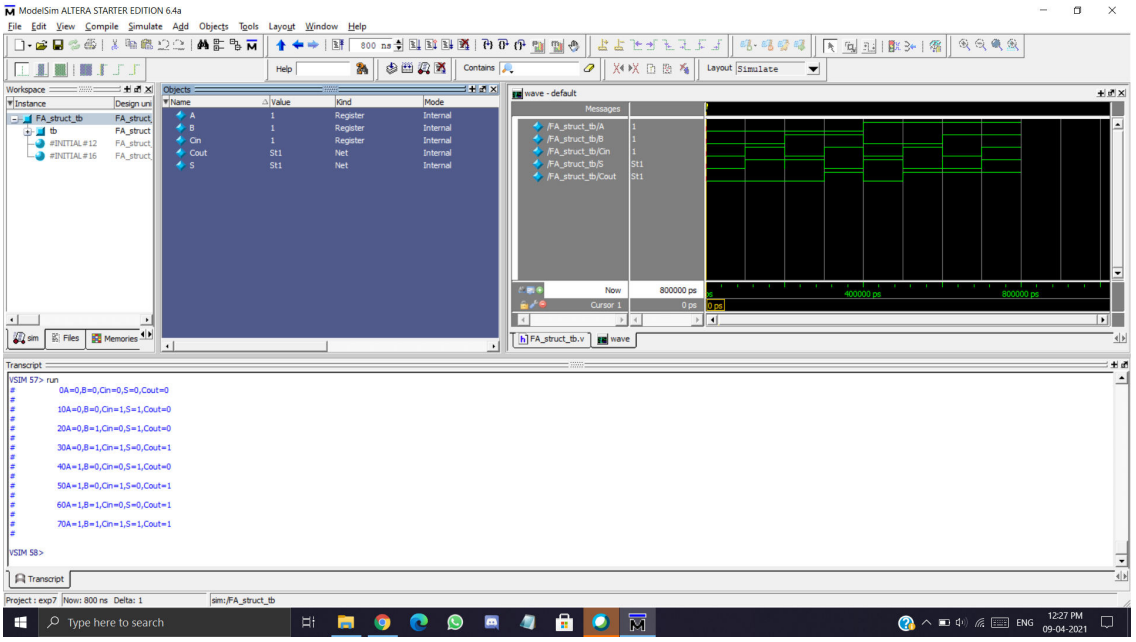


Figure 51: **Structural Model** of **full adder**

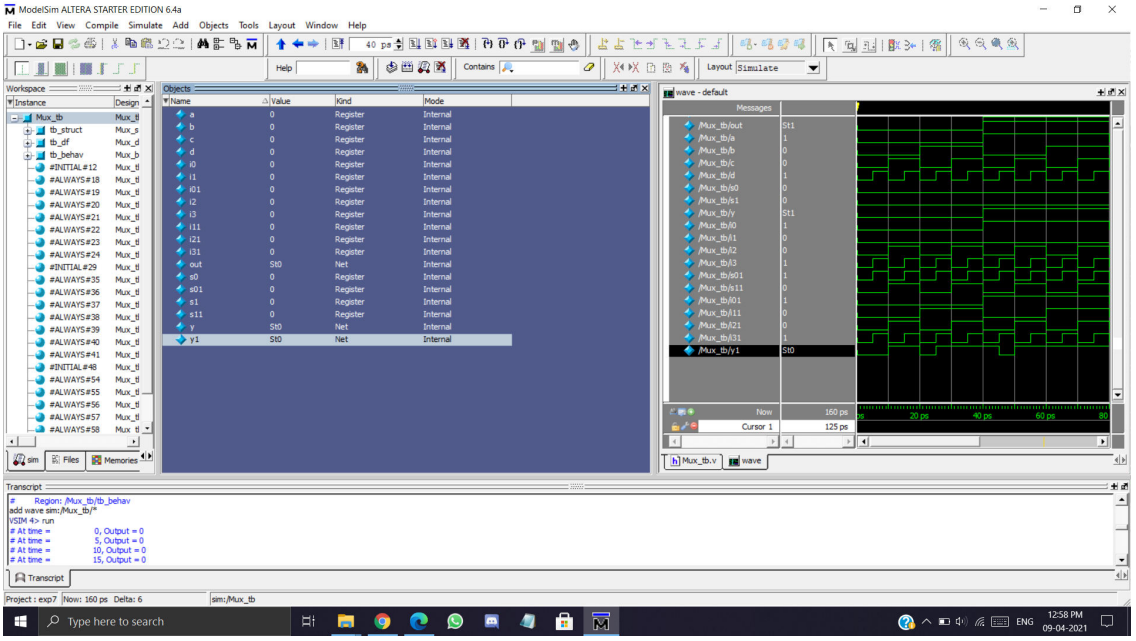


Figure 52: Waveform of **4:1 MUX**

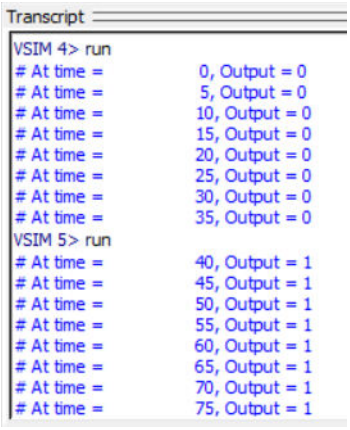


Figure 53: Console output of **4:1 MUX**

7.6 Conclusion: Full adder and 4:1 Multiplexer was designed,implemented and stimulated using Quartus II and ModelSim 6.4a in Verilog HDL. All three models i.e. structural, behavioral, and data flow were implemented.

Experiment 8 - Registers and Counters in Verilog (25-03-2021)

8.1 Theory:

A **shift register** is made up of many single-bit "D-Type Data Latches," one for each data bit, either a logic "0" or logic "1," linked in a serial type daisy-chain structure, with the output of one latch being the input of the next, and so on. Shift Registers are used in calculators and computers for data storage or data movement. They are commonly used to store data such as two binary numbers until they are connected together, or to convert data from serial to parallel or parallel to serial formats.

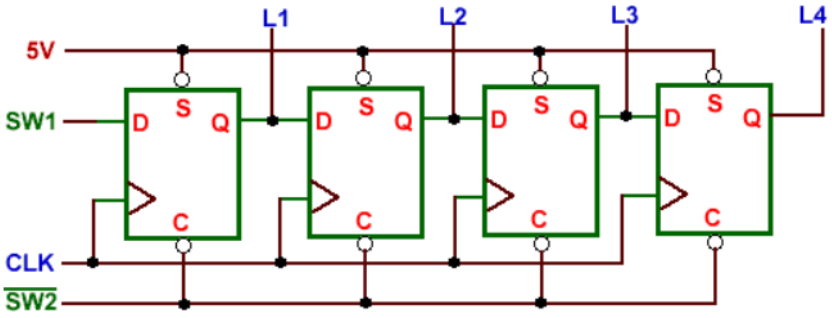


Figure 54: 4 bit Shift Register

Counters use sequential logic to count clock pulses. A counter can be implemented implicitly with a Register Inference. The Quartus II software can infer a counter from an If Statement that specifies a clock edge together with logic that adds or subtracts a value from the signal or variable. The If Statement and additional logic should be inside a Process Statement.

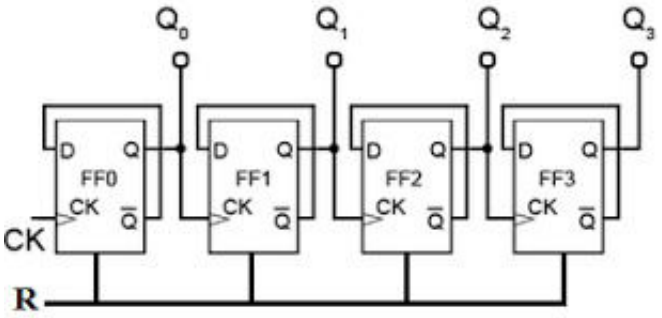


Figure 55: 4 bit Up Counter

8.2 Codes:

Shift register 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
```

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
```

counter synchronous

```
module Counter_sync #(parameter N=4) (
    input clock,
    input reset,
    output reg [3:0] out
);
    always @(posedge clock)
    begin
        if (!reset) \\active low
            out <= 0;
        else
            out <= out+1;
    end
endmodule
```

counter asynchronous

```
module Counter_async #(parameter N=4) (
    input clock,
    input reset,
    output reg [3:0] out
);
    always @(posedge clock or negedge reset)
    begin
        if (!reset) \\active low
            out <= 0;
        else
            out <= out+1;
```

```
    end
endmodule
```

8.3 Testbench:

shift reg with async tb

```
‘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
```

shift reg with sync reset tb

```
‘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));

    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
```

counter sync tb

```
‘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 async tb

```
‘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: -

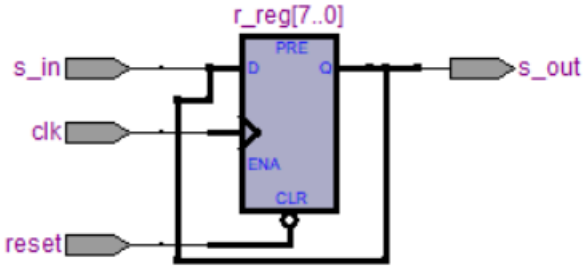


Figure 56: RTL view of **Shift register** with **synchronous reset**

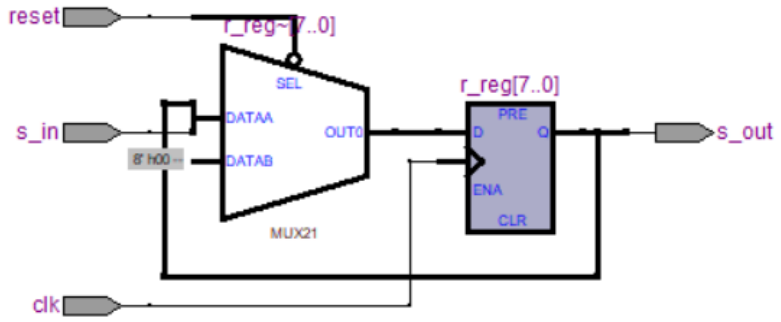


Figure 57: RTL view of **Shift register** with **asynchronous reset**

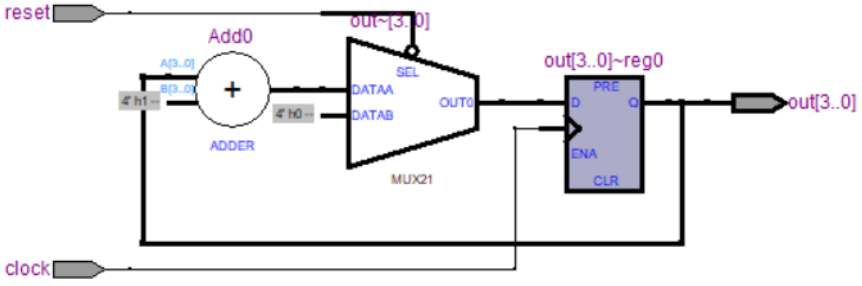


Figure 58: RTL view of **Counter** with **synchronous reset**

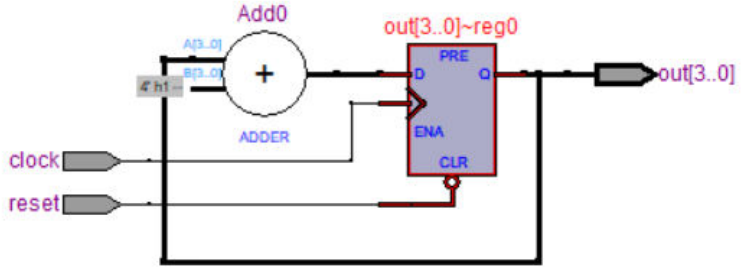


Figure 59: RTL view of **Counter** with **asynchronous reset**

8.5 Simulation Waverform: -

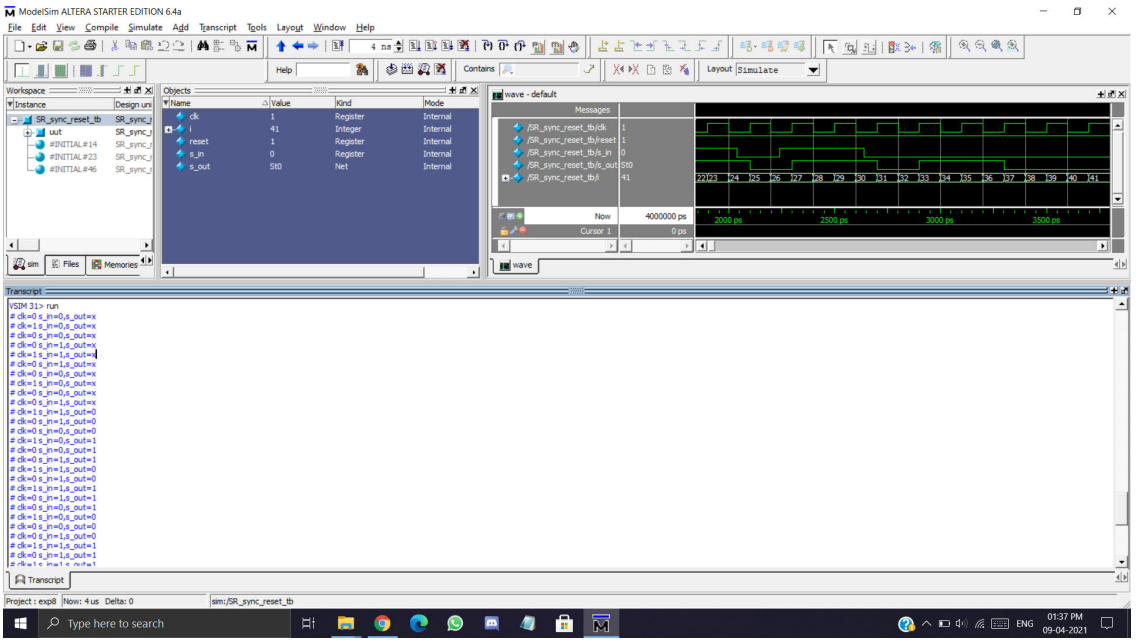


Figure 60: **Shift Register** with **synchronous reset**

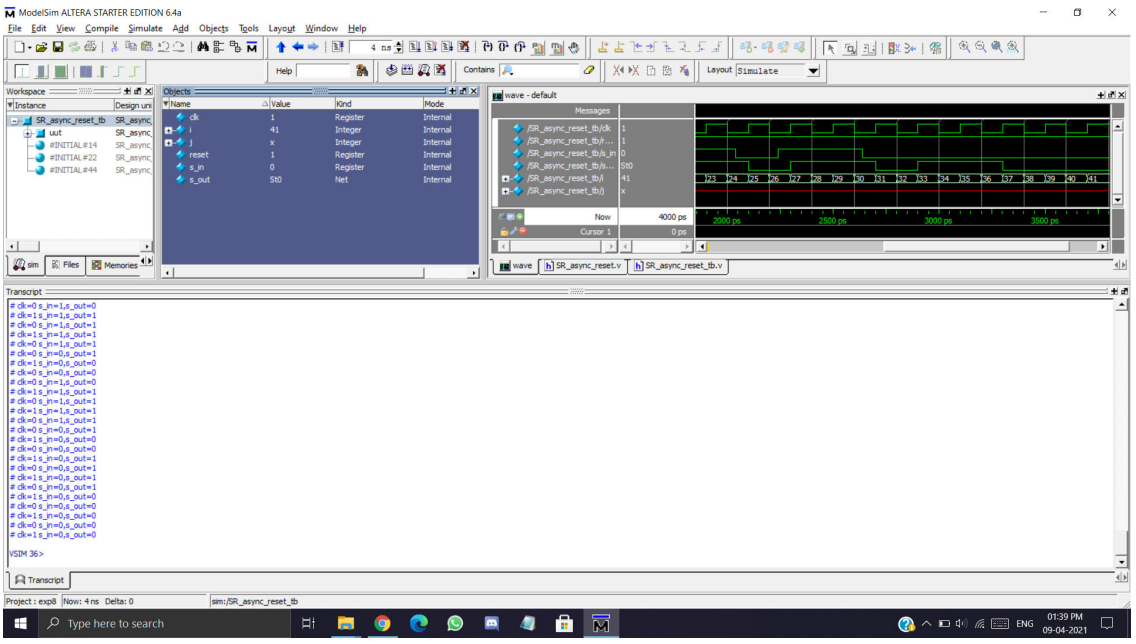


Figure 61: **Shift Register** with **asynchronous reset**

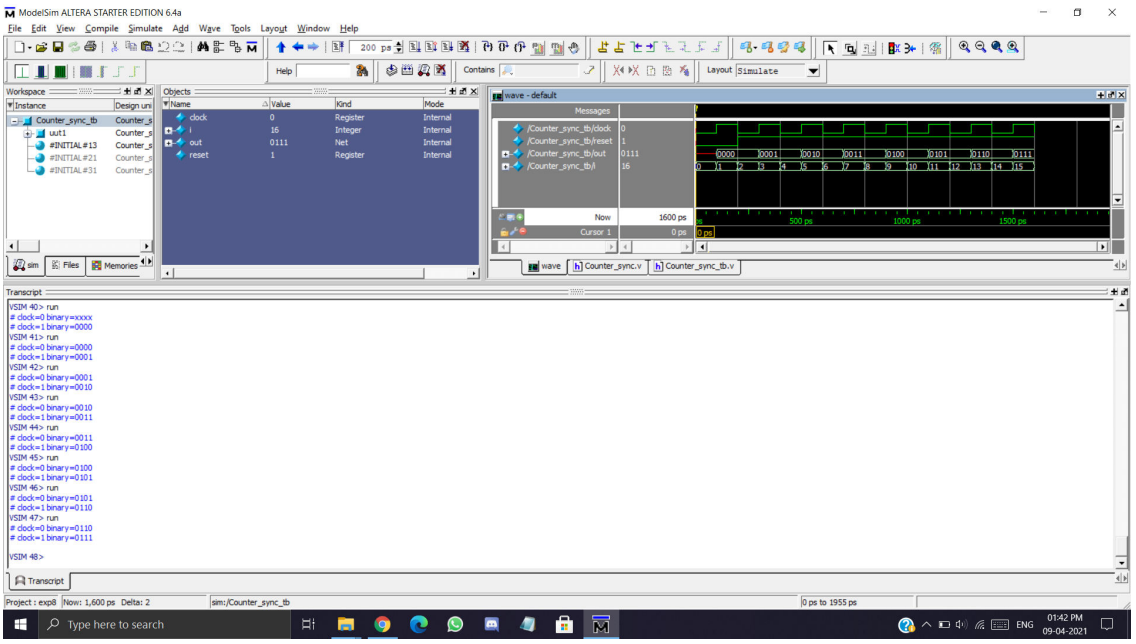


Figure 62: **Counter** with **synchronous reset**

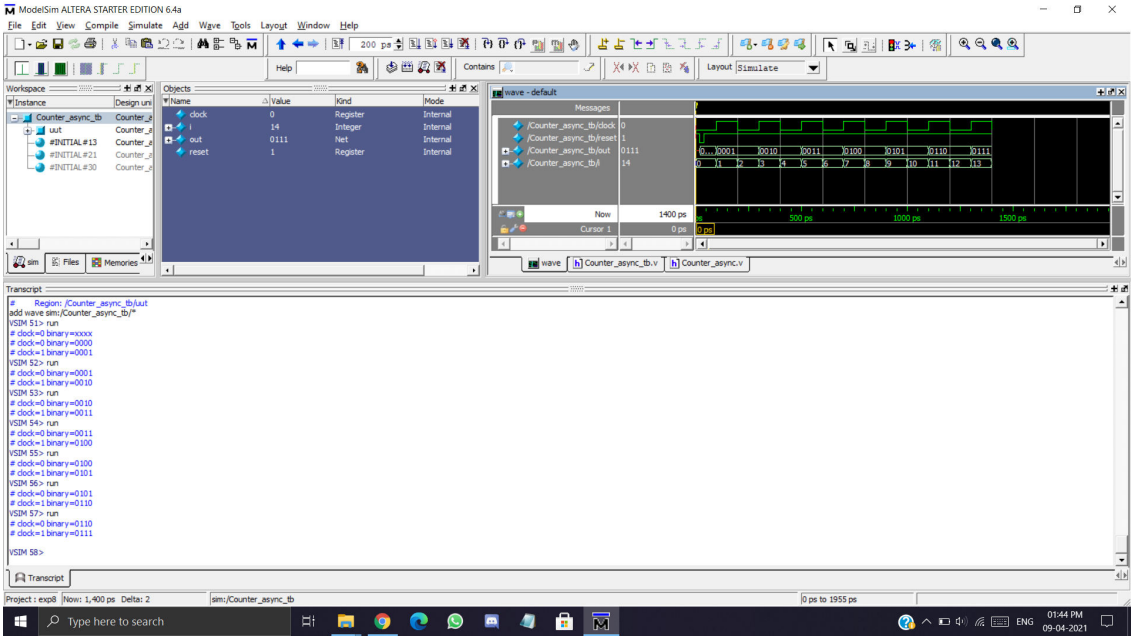


Figure 63: **Counter** with **asynchronous reset**

8.6 Conclusion: In this experiment we observed that verilog coding is similar to C programming and is easy to implement than VHDL since syntax is quite familiar. 8 bit shift register both synchronous and asynchronous also 4 bit counter(synchronous and asynchronous) was implemented and simulated using Quartus II 9.0 and ModelSim 6.4a. Following things were observed

- **Synchronous Up-counter**: This counter counts sequentially on every positive edge of clock 0 to 15. The always block triggers at every positive edge of clock. The counter resets only when reset is low and clock is high.
- **Asynchronous Up-counter**: This counter counts sequentially on every rising edge of clock provided reset is low . Here always is triggered at rising edge of clock or falling edge of reset. The counter resets when reset is low. It does not depends on clock state hence known as counter with asynchronous reset.
- **Shift Register with synchronous reset**: Whenever there is positive edge of clk and reset(active low) is low, register is cleared and initialized to 0. If clear is low then value of register is shifted right by 1 bit and LSB is loaded with present input.
- **Shift Register with asynchronous reset**: It is similar to above one only difference is reset is not synchronized with clk i.e. if reset is high register is initialized to 0 irrespective of clock, high or low.

Experiment 9 - Introduction to FPGA & combinational circuits' implementation on FPGA board (08-04-2021)

9.1 Theory: A **field-programmable gate array (FPGA)** is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence the term "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC).
FPGAs contain an array of programmable logic blocks, and a hierarchy of "reconfigurable interconnects" allowing blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. Many FPGAs can be reprogrammed to implement different logic functions,[2] allowing flexible reconfigurable computing as performed in computer software.

Altera's Cyclone III FPGA

Terasic technologies design FPGA devlopment kits for Intel's Altera Cyclone III. Altera's Cyclone III FPGA Development Kit combines the largest density low-cost, low-power FPGA available with a robust set of memories and user interfaces.

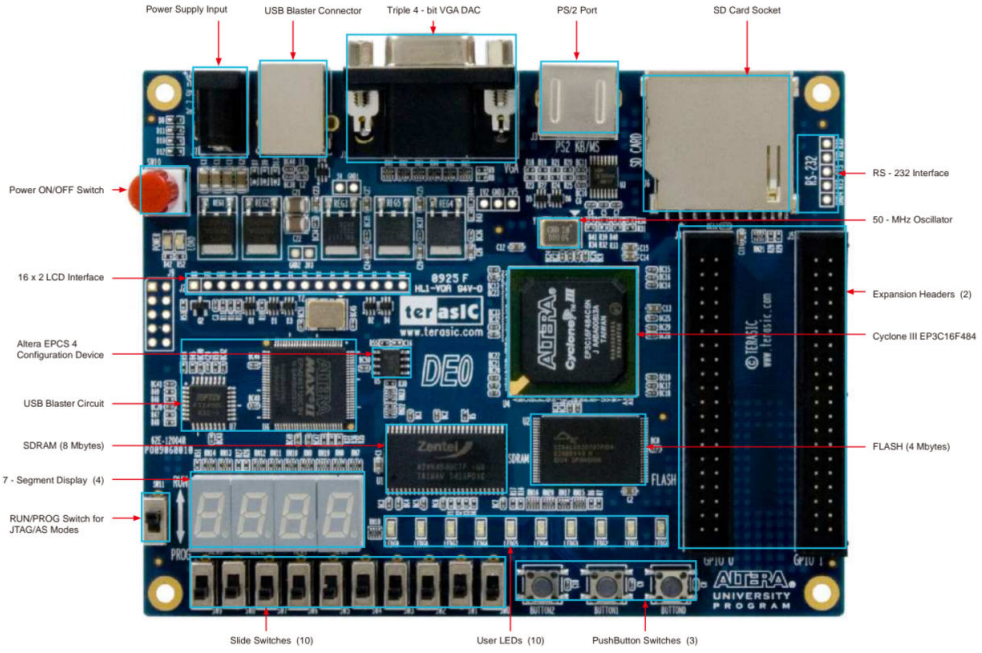


Figure 64: The development board

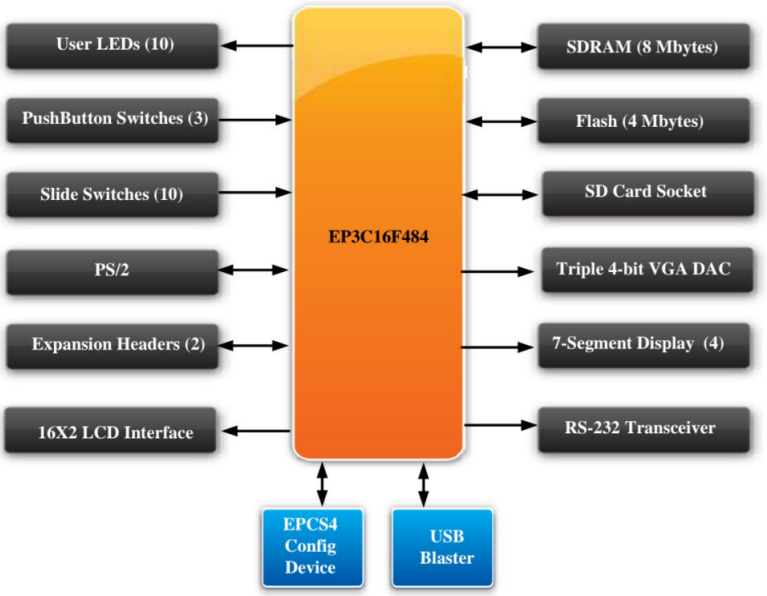


Figure 65: Block diagram of the development board

Steps to burn code in FPGA

1. In Quartus II, open the .qpf file of the project.
2. Select the board from list of devices whose serial number **should match** the serial number of the FPGA board. In our case we select Cyclone III as the device family and EP3C16F484C6 as the device. Click on OK.
3. Compile the code.
4. Assign the pins of FGPA board to the input/output ports defined in the entity. For pin assignment in Quartus II, go to assignments tab →Pin Planner.
5. Assign the pins by referring to the user manual of the FPGA board in use.
6. Compile the code again after pin assignment is done.
7. To burn the code on the board go to Tools →Programmer →select .sof file →start.
8. Once the progress bar reads 100%, we can check the working of our code on the FPGA by changing the input using the switches and observe the output with the help of LEDs.

9.2 Screenshots: -

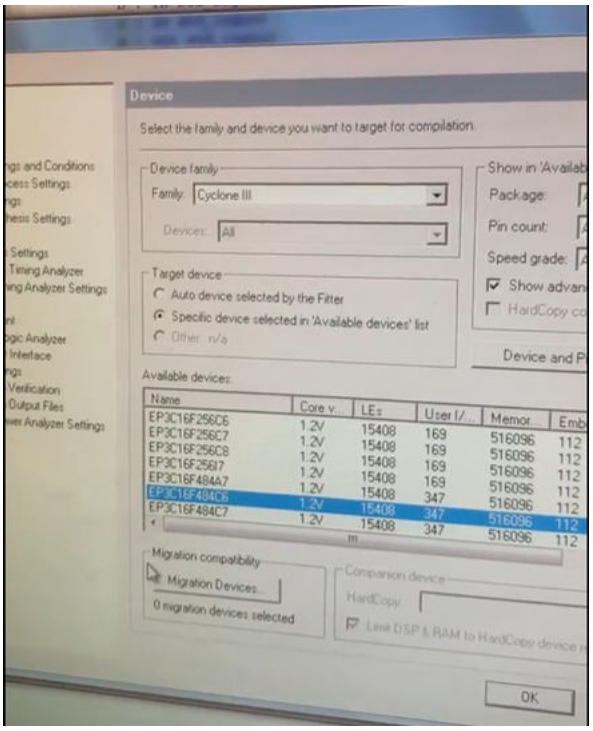
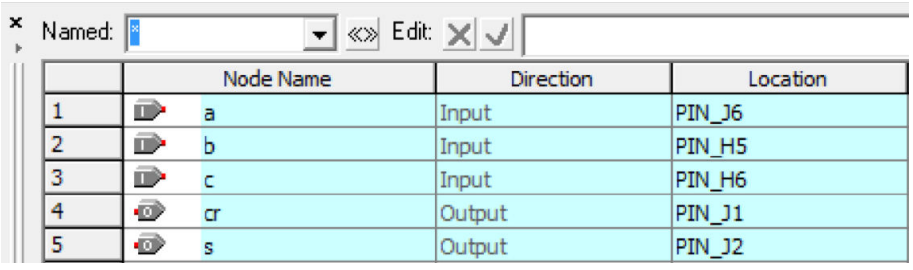


Figure 66: Selection of FPGA board

Full adder



	Node Name	Direction	Location
1	a	Input	PIN_J6
2	b	Input	PIN_H5
3	c	Input	PIN_H6
4	cr	Output	PIN_J1
5	s	Output	PIN_J2

Figure 67: Pin assignment for full adder

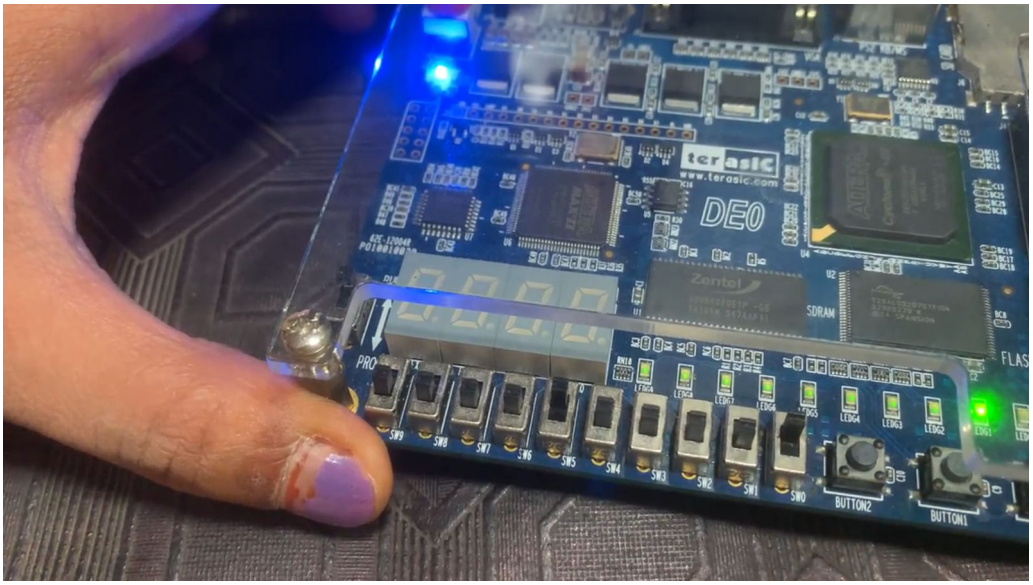


Figure 68: *a=1, b=0, c=0, sum=1, cout=0*

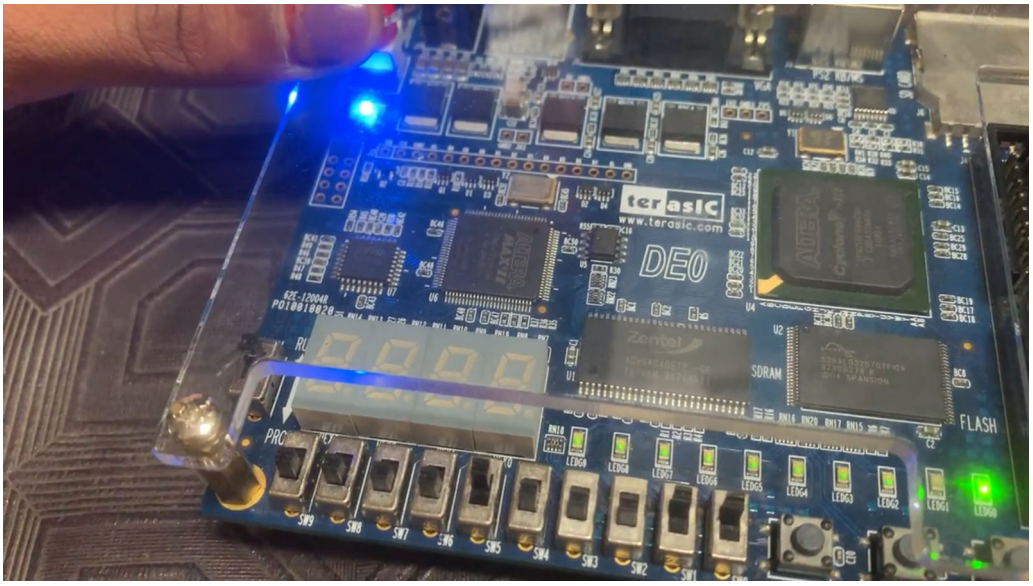


Figure 69: *a=1, b=1, c=0, sum=0, cout=1*

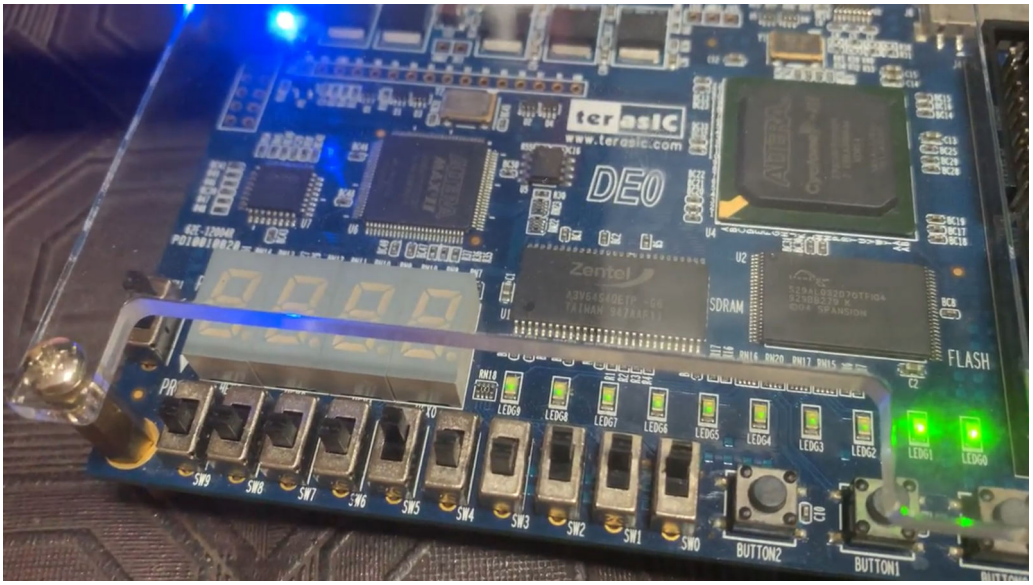


Figure 70: *a=1*, *b=1*, *c=1*, *sum=1*, *cout=1*

4:1 Multiplexer

Named: Edit:

	Node Name	Direction	Location	I/O Bank	VREF Group
1	A	Input	PIN_J6	1	B1_N0
2	B	Input	PIN_H5	1	B1_N0
3	C	Input	PIN_H6	1	B1_N0
4	D	Input	PIN_G4	1	B1_N0
5	S0	Input	PIN_G5	1	B1_N0
6	S1	Input	PIN_J7	1	B1_N1
7	Z	Output	PIN_J1	1	B1_N1

Figure 71: Pin assignment for full adder

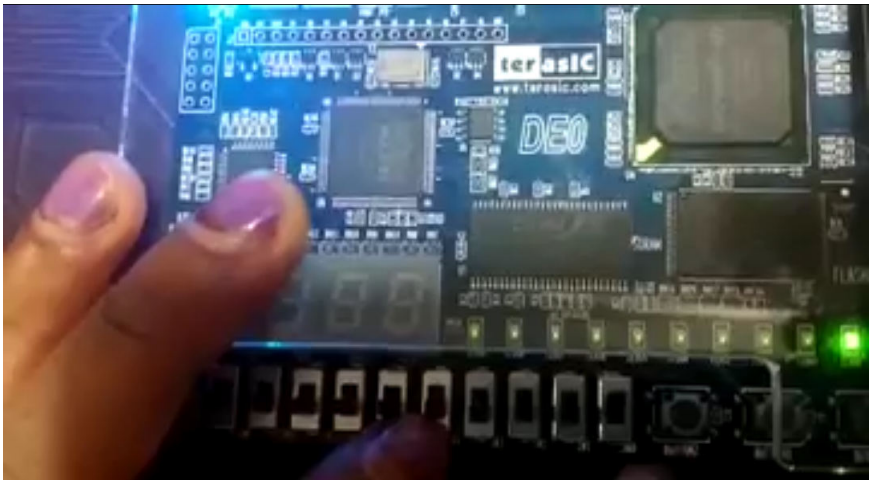


Figure 72: *s0=0*, *s1=0*, *z=a*

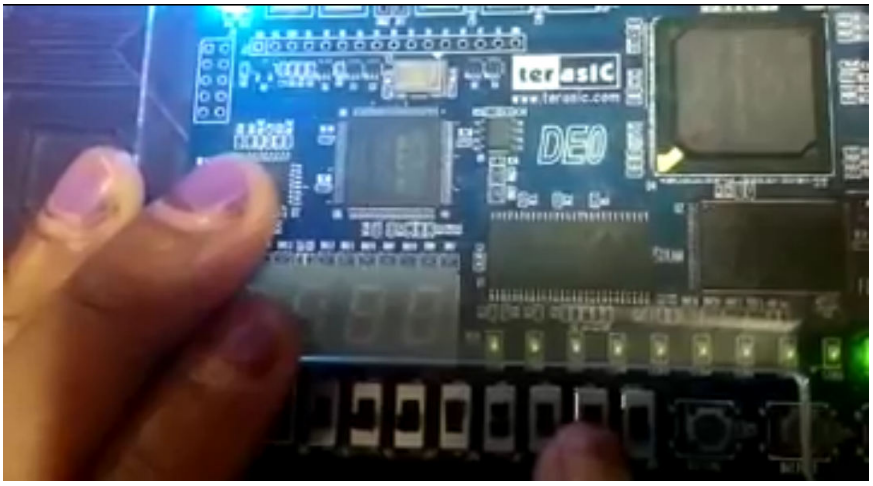


Figure 73: *s0=1*, *s1=0*, *z=b*

9.3 Conclusion: It can be concluded from our first experiment on a FPGA board that FPGAs are useful for prototyping application-specific integrated circuits (ASICs) or processors. Fristly, we programmed the FPGA as a full adder and secondly as a 4:1 multiplexer. We reprogrammed the FPGA until our design was final and bug-free.

Experiment 10 - Sequential circuits' implementation on FPGA board (09-04-2021)

10.1 Theory:

Steps to burn code in FPGA

1. In Quartus II, open the .qpf file of the project.
2. Select the board from list of devices whose serial number **should match** the serial number of the FPGA board. In our case we select Cyclone III as the device family and EP3C16F484C6 as the device. Click on OK.
3. Compile the code.
4. On successful compilation, assign the pins of FGPA board to the input/output ports defined in the entity. For pin assignment in Quartus II, go to assignments tab →Pin Planner.
5. Assign the pins by referring to the user manual of the FPGA board in use.
6. Compile the code again after pin assignment is done.
7. To burn the code on the board go to Tools →Programmer →select .sof file →start.
8. Once the progress bar reads 100%, we can check the working of our code on the FPGA by altering the input using the switches and observe the output with the help of LEDs acting as output .

In order to change the clock frequency we add a delay using **temp**. Code for the same is as follows

Code for modification of clock frequency

```
process(clock,reset)
begin
    if( clock'event and clock='1') then
        if reset='1' then
            temp<=(others =>'0');
            srst<='1';
        else
            temp<=temp+1;
            sclk<=temp(24);
```

```
            if temp(25)<='1' then
                srst<='0';
            else null;
            end if;
        end if;
    end if;
end process;
```

10.2 Screenshots: -

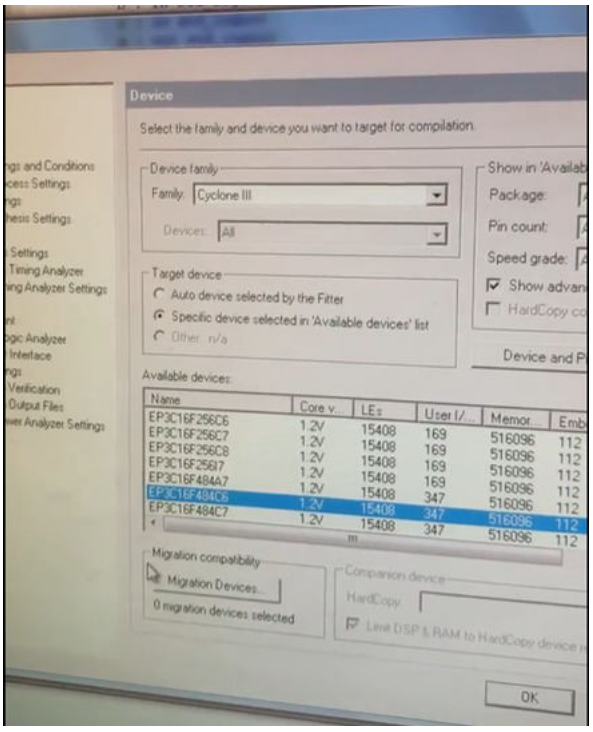


Figure 74: Selection of FPGA board

Counter

Named: Edit:

	Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard
1	clock	Input	PIN_G21	6	B6_N1	2.5 V (default)
2	cout[3]	Output	PIN_J1	1	B1_N1	2.5 V (default)
3	cout[2]	Output	PIN_J2	1	B1_N1	2.5 V (default)
4	cout[1]	Output	PIN_J3	1	B1_N1	2.5 V (default)
5	cout[0]	Output	PIN_H1	1	B1_N1	2.5 V (default)
6	reset	Input	PIN_J6	1	B1_N0	2.5 V (default)
7	<<new node>>					

Figure 75: Pin assignment for counter

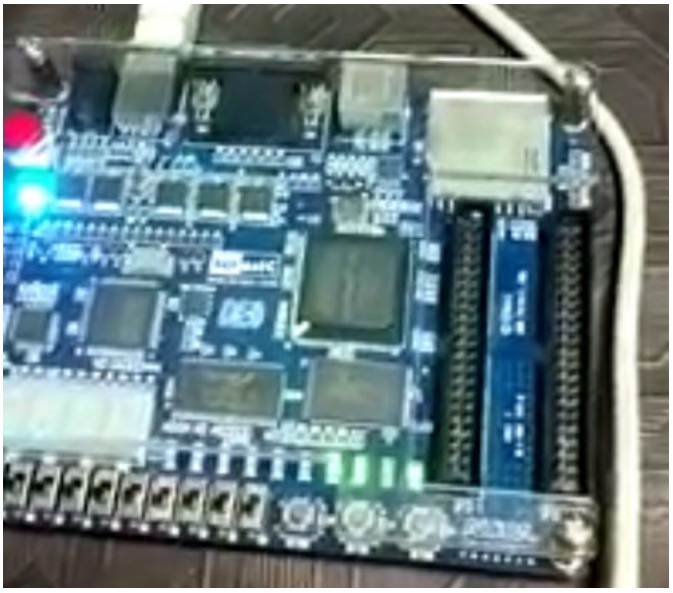


Figure 76: Counter

8 bit Shift register

Named: Edit: PIN_G21

	Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard
1	clk	Input	PIN_G21	6	B6_N1	2.5 V (default)
2	Input_Data	Input	PIN_D2	1	B1_N0	2.5 V (default)
3	Q[7]	Output	PIN_J1	1	B1_N1	2.5 V (default)
4	Q[6]	Output	PIN_J2	1	B1_N1	2.5 V (default)
5	Q[5]	Output	PIN_J3	1	B1_N1	2.5 V (default)
6	Q[4]	Output	PIN_H1	1	B1_N1	2.5 V (default)
7	Q[3]	Output	PIN_F2	1	B1_N0	2.5 V (default)
8	Q[2]	Output	PIN_E1	1	B1_N0	2.5 V (default)
9	Q[1]	Output	PIN_C1	1	B1_N0	2.5 V (default)
10	Q[0]	Output	PIN_C2	1	B1_N0	2.5 V (default)
11	reset	Input	PIN_J6	1	B1_N0	2.5 V (default)

Figure 77: Pin assignment for 8 bit shift register

Figure 78: Shift register

10.3 Conclusion: Counter and Shift register were implemented in Quartus II and same was burnt on FPGA board. Switch **SW0** was mapped as **reset pin**. Corresponding outputs were observed on first 4 LEDs in case of counter and first 8 LEDs in case of shift register as they were 4 bits and 8 bits respectively. Since the frequency of inbuilt clock on FPGA board is **50 MHz** we were not able to observe output. In order to observe the change in output we applied delay using **temp**. This creates a delay of $(2^{25} - 1)/50 * 10^{-6} = 0.67sec$