Digital SoC Design Projects

Anirudha Behera
Illinois Institute of Technology
MSc in Electrical Engineering
ID-A20503687

CONTENTS

PROJECT-1

1. PURPOSE:

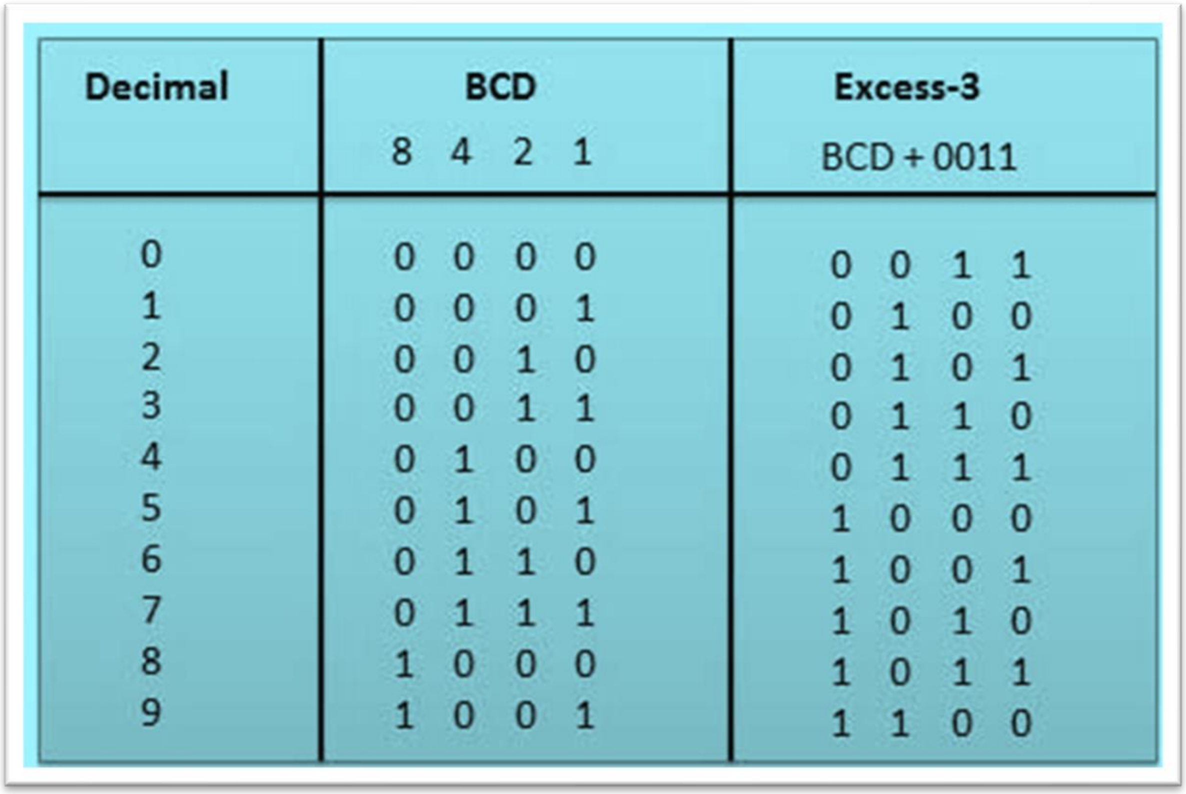
In this first project I am going to design and implement code converters circuit using ZedBoard Zynq7000 development board and Xilinx Vivado as digital circuit development tool.

2. BACKGROUND:

In a typical code conversion technique, we basically convert one type of code to another type for specific purposes like converting human friendly code format to computer understandable code format.

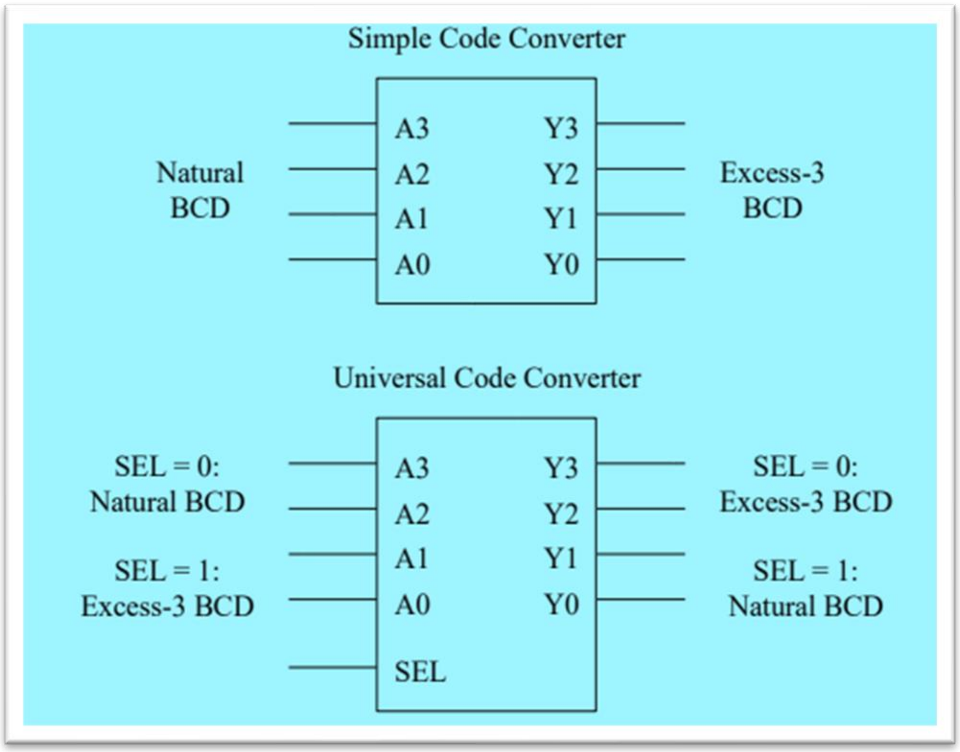
In this lab work we will be designing code converters. For first two designs we will focus on converting Binary Coded Decimal format to Excess-3 format and for the third one we will focus on designing a Universal Code Converter which will be able to convert BCD to Excess-3 and Excess-3 to BCD.

TRUTH TABLE



Decimal	BCD 8 4 2 1	Excess-3 BCD + 0011
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0

The below block diagrams will show the component arrangement with their names of respective code converters.



3. PROCEDURE:

- We are opening Xilinx Vivado suite in order to design our proposed code converter model.
- In the first page we will choose to "Create New Project" option and then I gave name to my design files and added Zedboard constrain file in next page.
- Then in the third page I selected my Zedboard from boards for testing purposes.
- All the programs are written in VHDL programming which is known as a very popular hardware description language.
- After specifying the port pins to the model using VHDL language we started defining the logic of normal code converter.
- After writing the .vhd code for code converter I started writing testbench for the proposed design model which will be used for testing purposes.
- In that testbench we will instantiate the previously designed code converter model and its components.
- Then we will add signals to assign variables and write the test code for performance testing.

- This same technique is followed for all three conde converter models. Their all-logic id defined in different ways and the Universal one is the mode advanced and complex one as compared to other two.

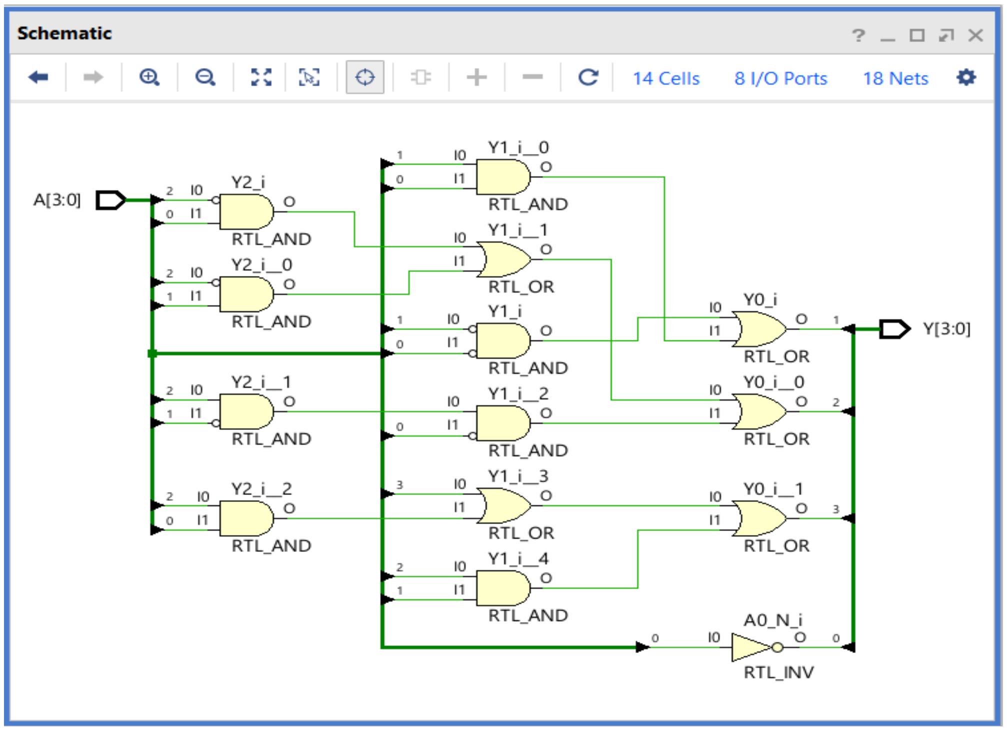
4. RESULTS:

- First, we will define the input and output pins in added Zedboard.xdc constrain file according to the defined logic in .vhd file.
- We must be very careful about the pin names in both files, they have to be the same.
- After that we can add out hardware board to the Xilinx Vivado by clicking on,

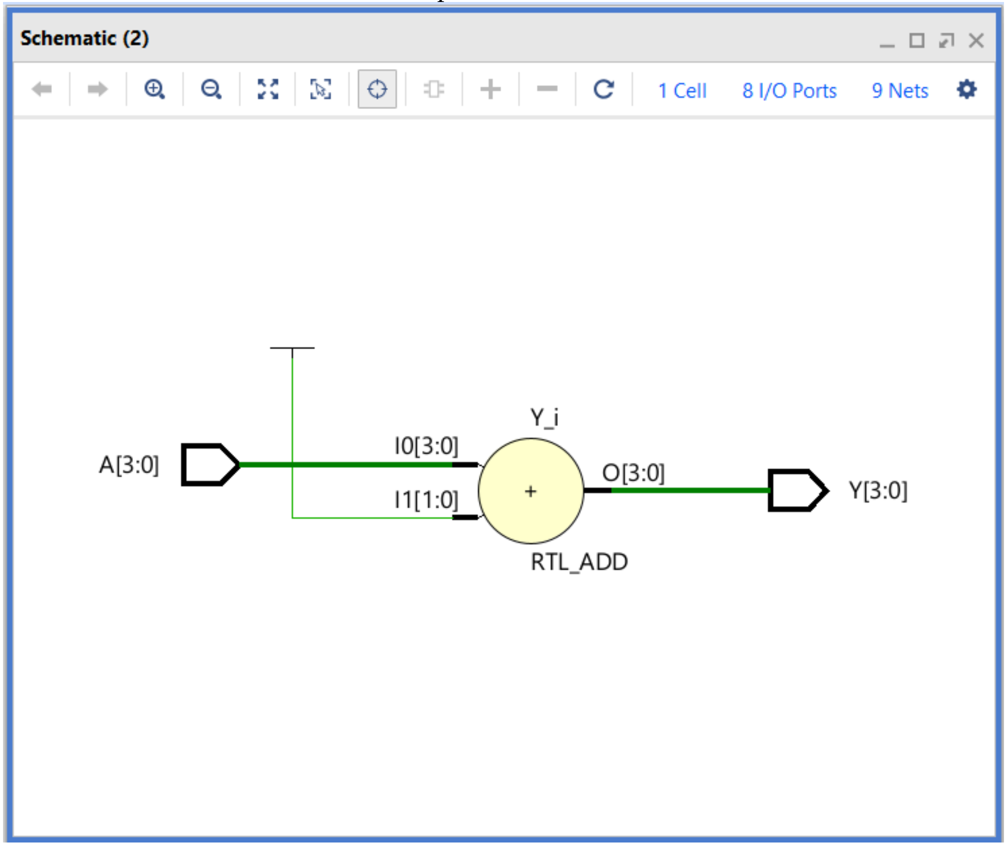
Open Hardware Manager > Open Target > Auto Connect

- After successfully connecting the board, we will generate the Bitstream for the respective model by clicking "Generate Bitstream" option on the lower left-hand side of Vivado suite.
- Now if the synthesis and Implementation is not done properly it will take ample amount of time to do it first and then only it will process the bitstream file.
- After saving the Bitstream file we can move towards the "Program Device" option which is below the "Hardware manager".
- By selecting it and loading the bitstream file to it we can successfully implement the proposed model logic to our connected external hardware.
- After implementing the logic to hardware, we can check the correctness of out logic by looking at the switch positions and LEDs lights.
- Programmed switches will act as input and LEDs will act as output to the design.
- Below I have given all three-code converter's simulation, RTL circuit, Implementation diagram and hardware output images for reference.

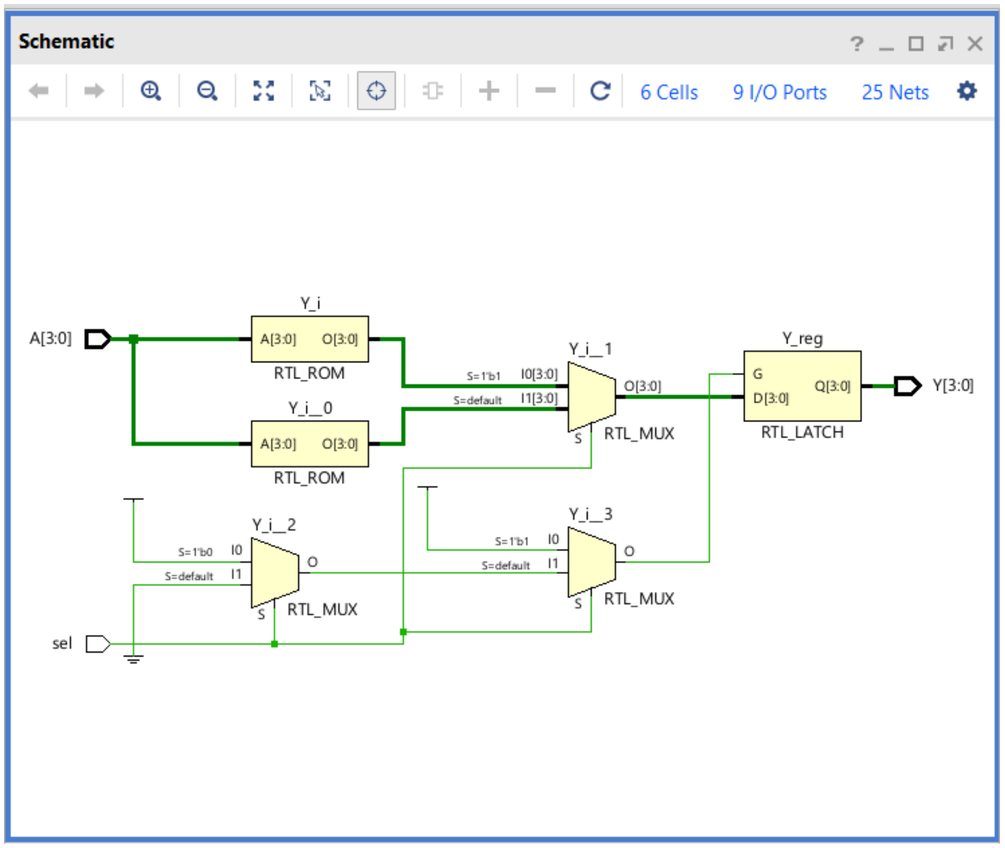
RTL Schematic:



First simple-Code Converter

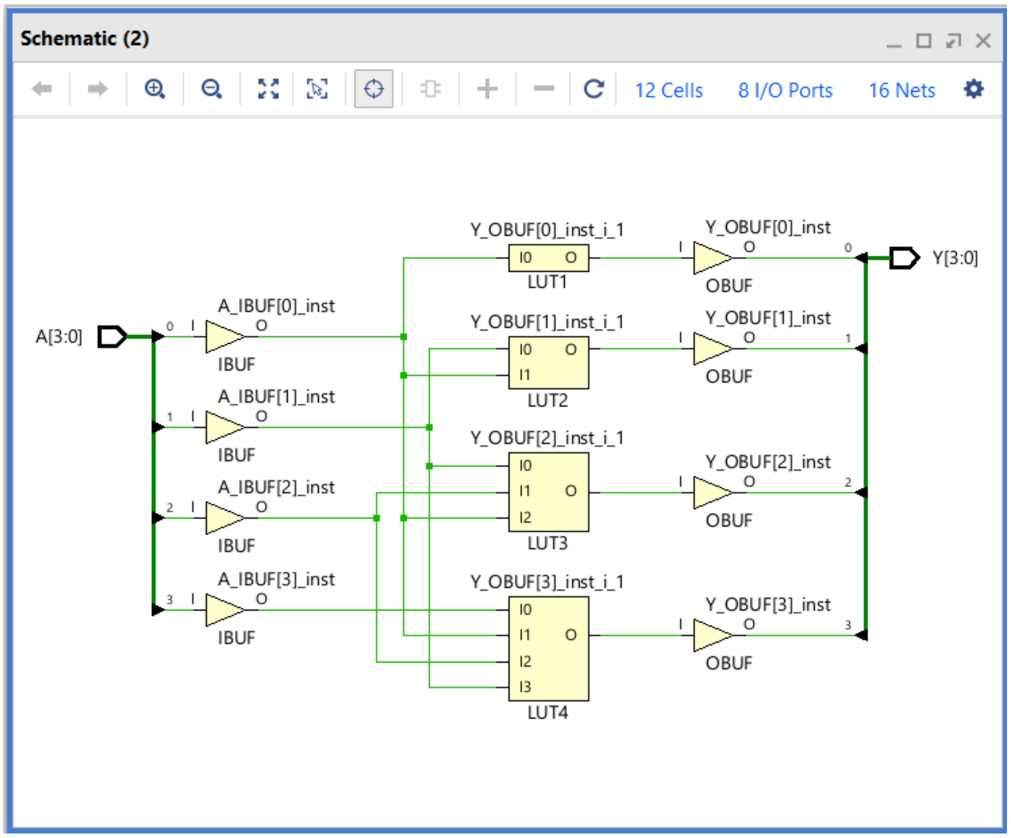


Second simple-Code Converter

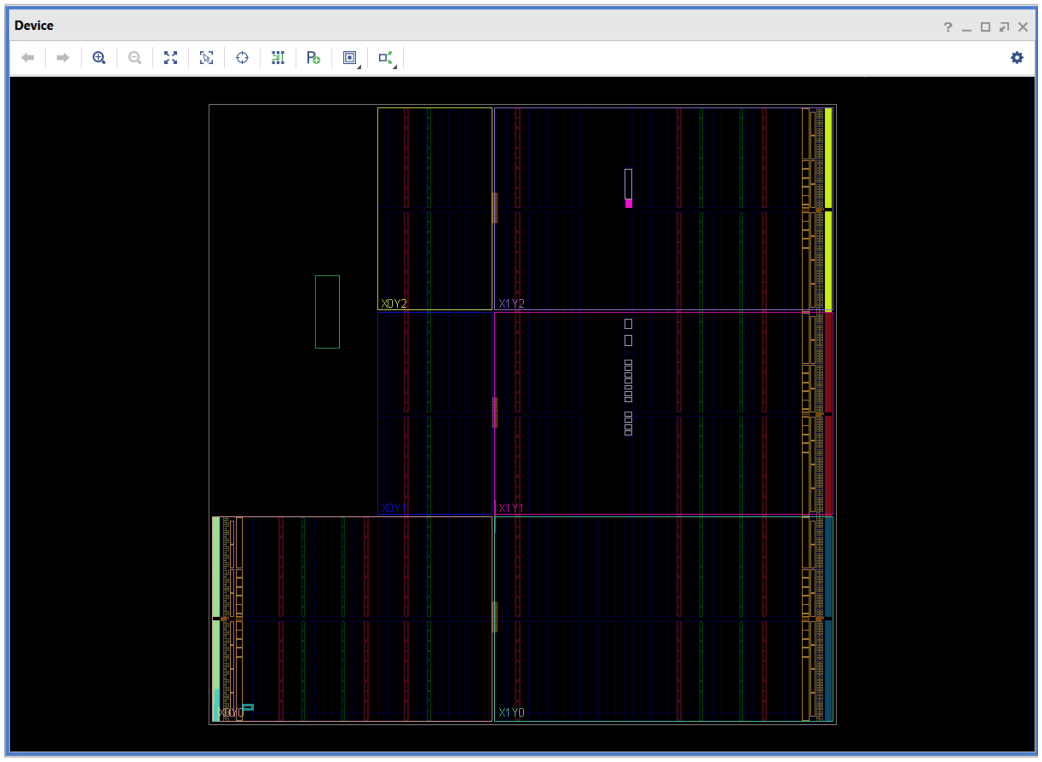


Third Universal-Code Converter

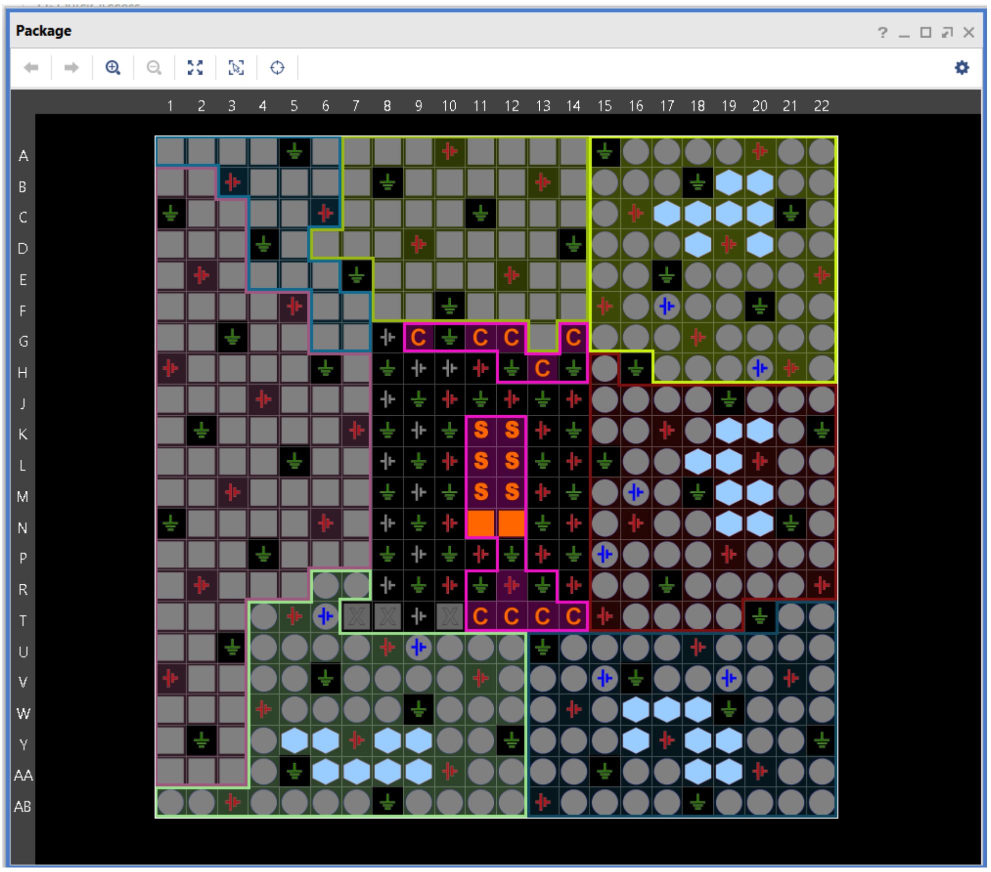
Implemented Schematic:



Floorplan:



I/O planning:

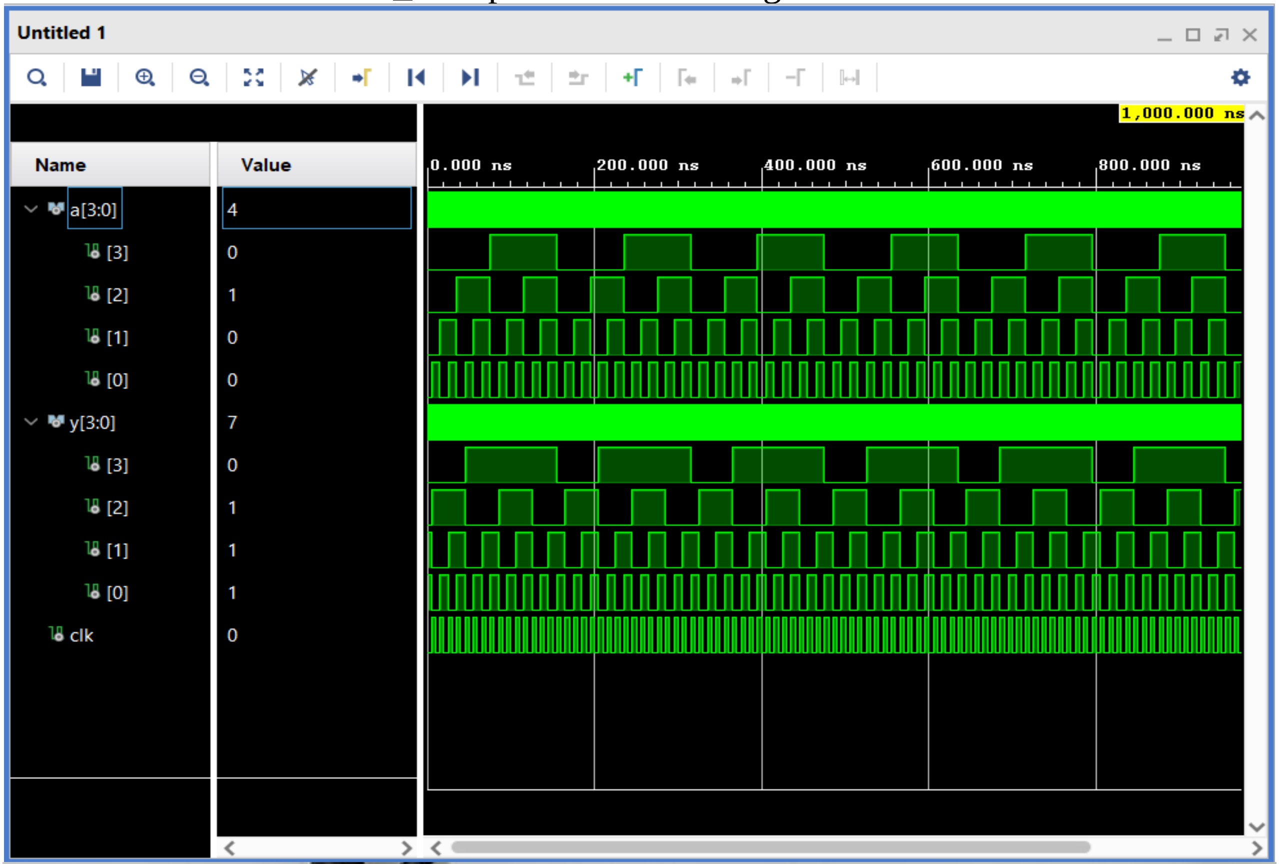


Simulation: First S!mple-Code Converter:

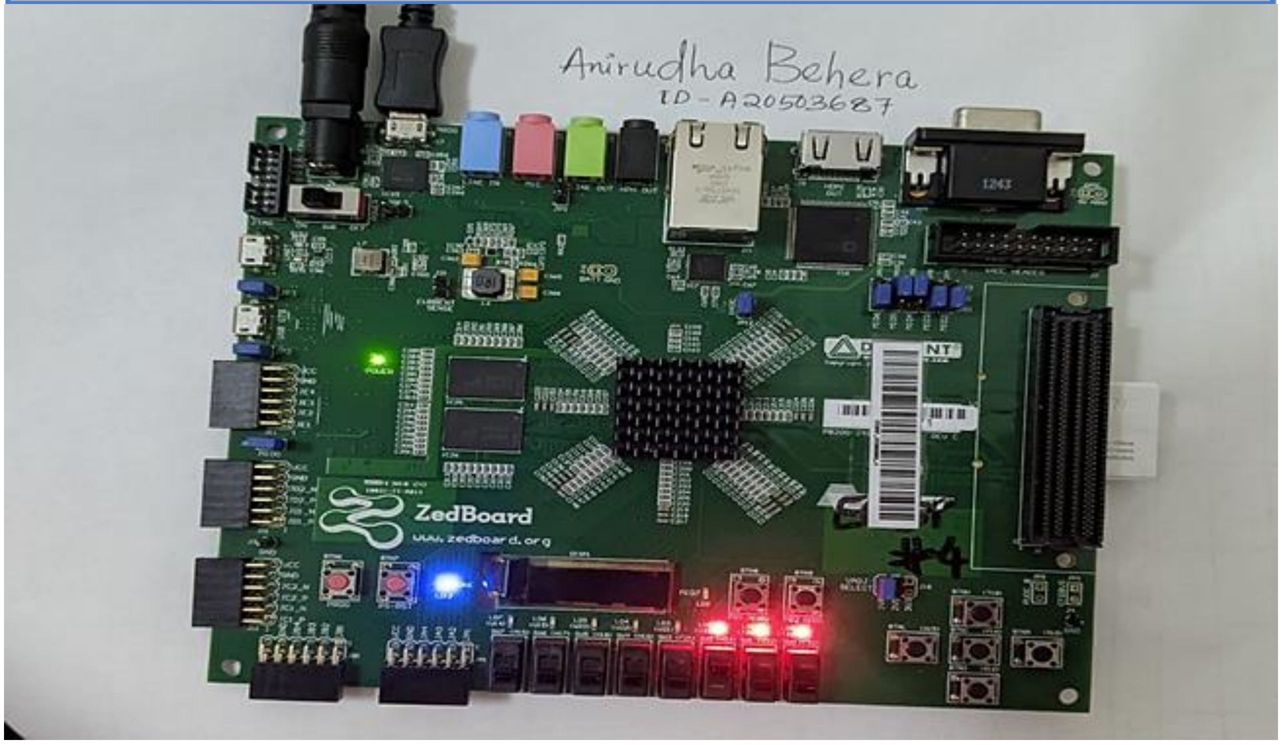
- All respective outputs w.r.t the inputs can be verified from above BCD to Ecxess-3 conversion table.

BCD input to the testing circuit: **0100**

Excess_3 output from the testing Circuit: **0111**



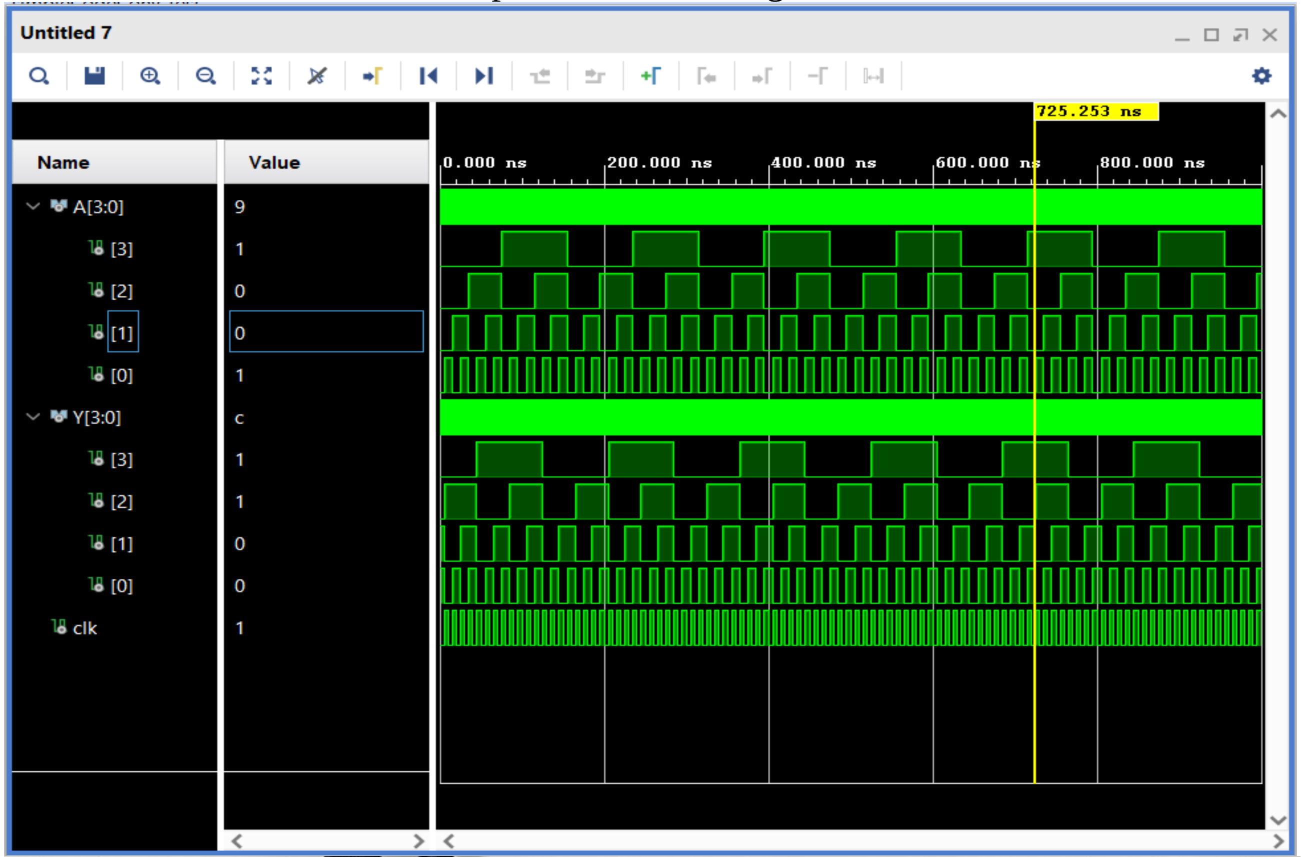
Name	Value
a[3:0]	4
[3]	0
[2]	1
[1]	0
[0]	0
y[3:0]	7
[3]	0
[2]	1
[1]	1
[0]	1
clk	0



2nd Code Converter:

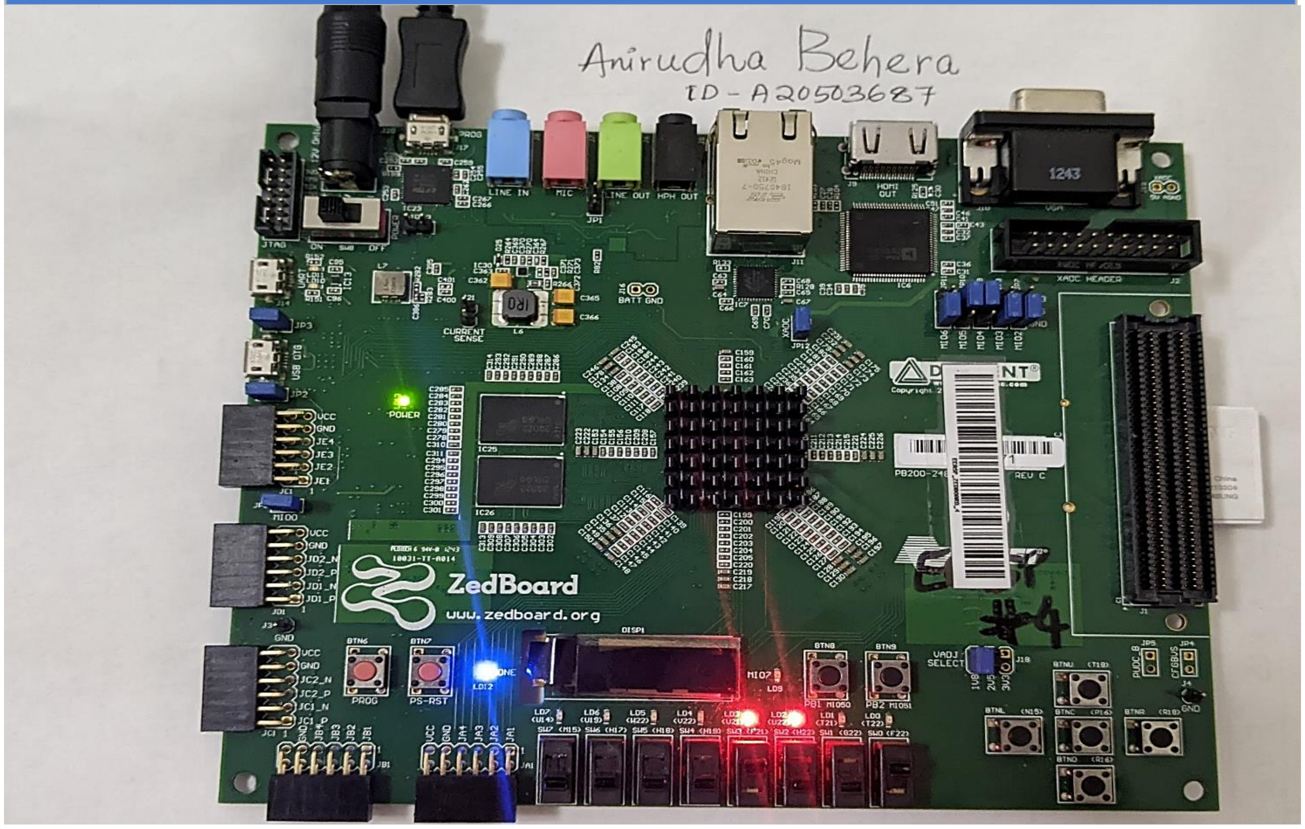
BCD input to the testing circuit: **1001**

Excess_3 output from the testing Circuit: **1100**



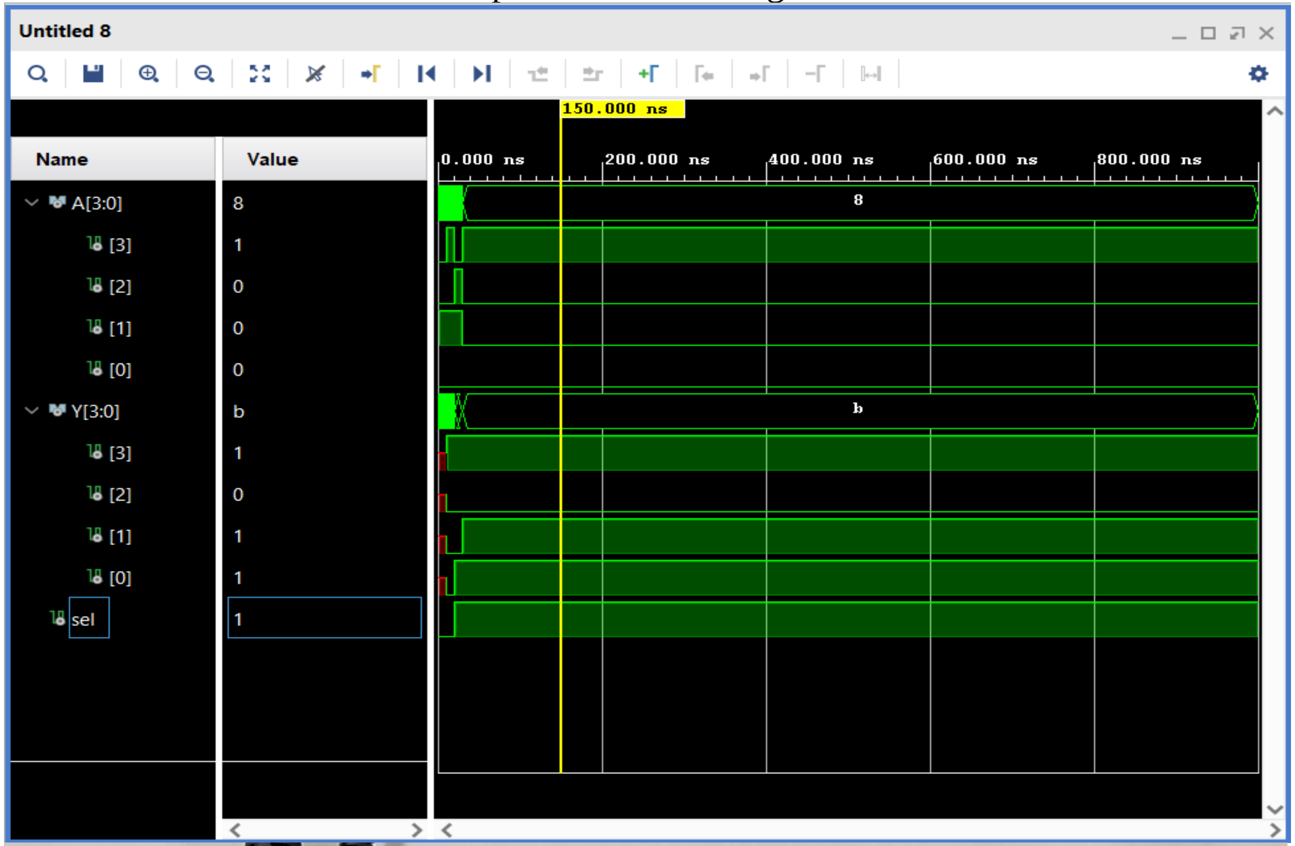
Untitled 7

Name	Value
A[3:0]	9
[3]	1
[2]	0
[1]	0
[0]	1
Y[3:0]	c
[3]	1
[2]	1
[1]	0
[0]	0
clk	1

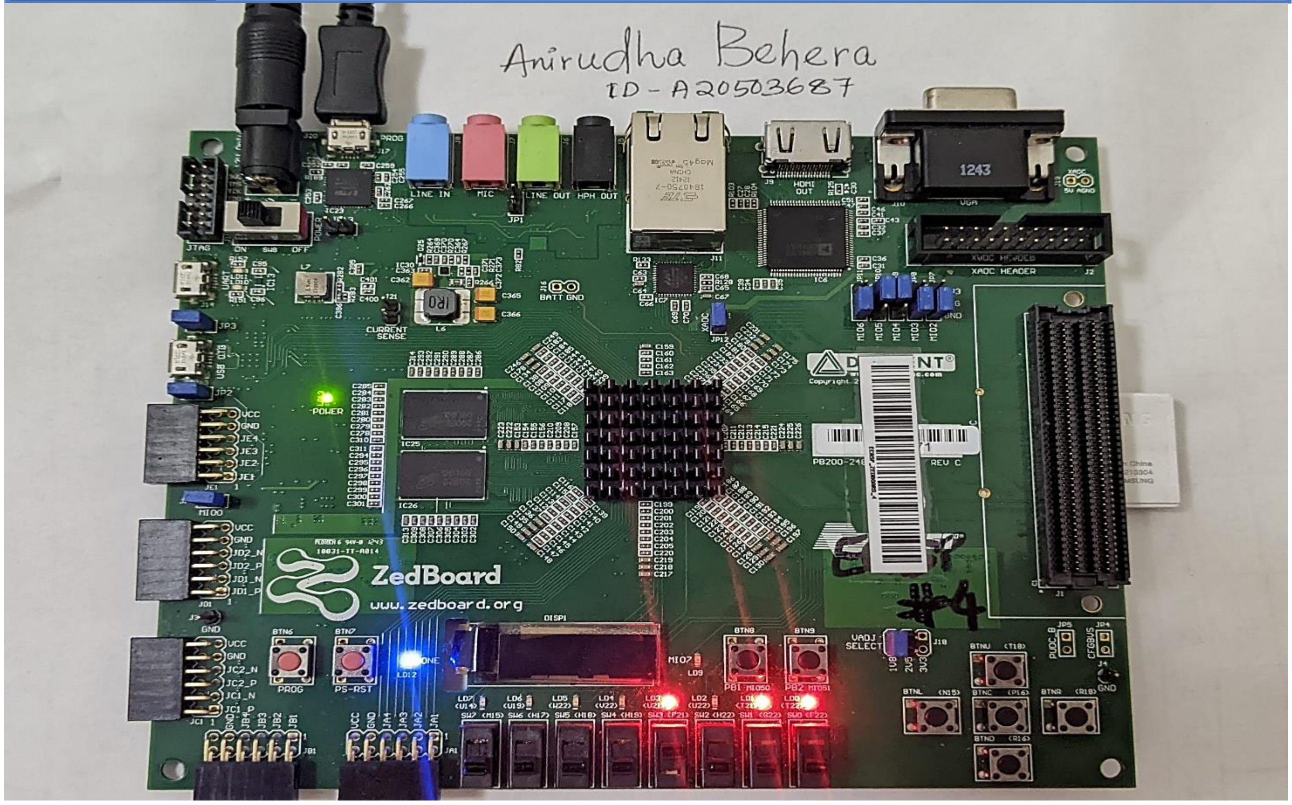


Universal Code Converter:

BCD input to the testing circuit: **1000** (BCD)
Excess_3 output from the testing Circuit: **1011**



Name	Value
A[3:0]	8
[3]	1
[2]	0
[1]	0
[0]	0
Y[3:0]	b
[3]	1
[2]	0
[1]	1
[0]	1
sel	1



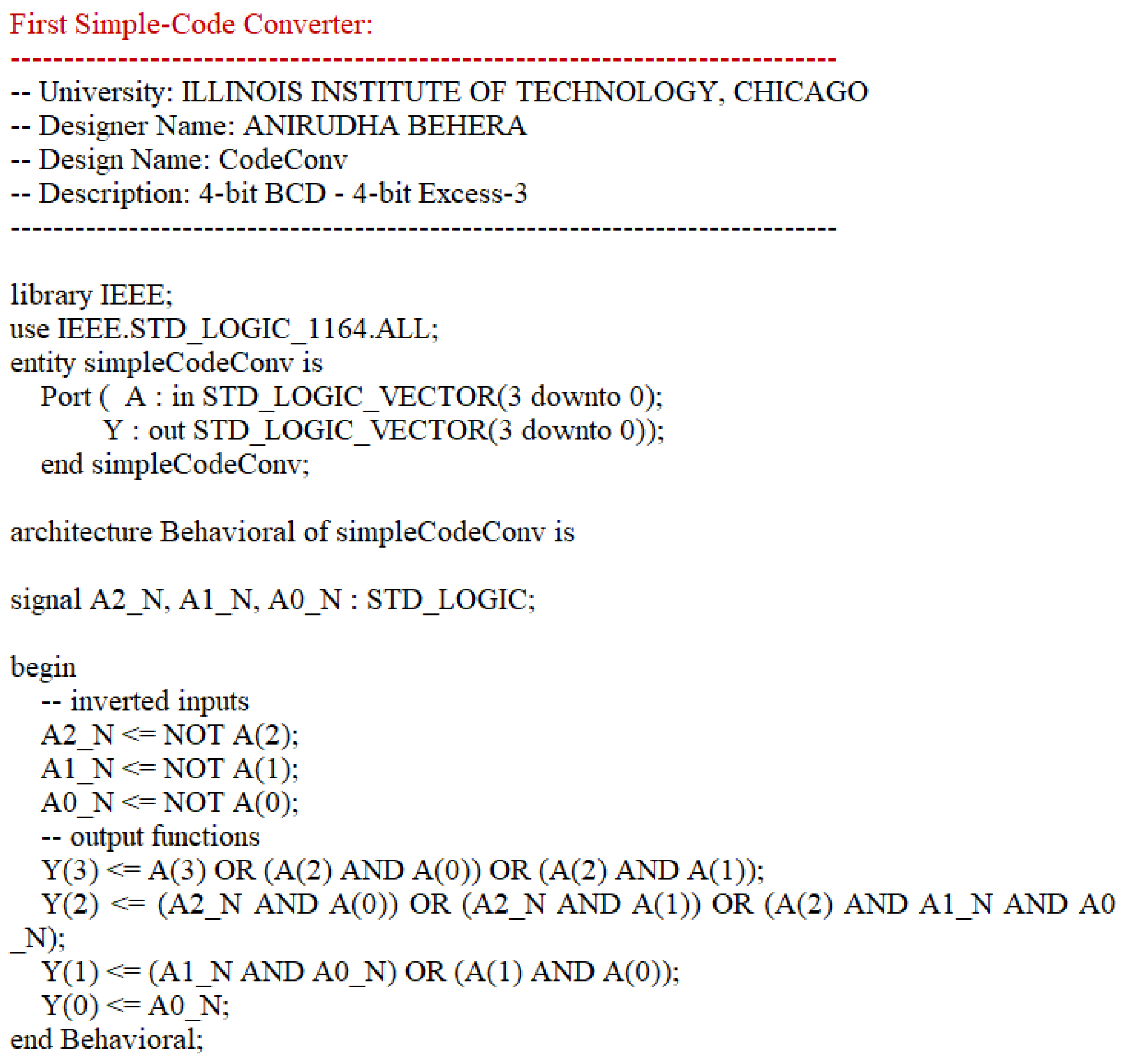
5. CONCLUSION:

We have got brief information almost the Vivado and handle of how to run blend, usage and produce bitstream for the board. We moreover executed the BCD to Excess-3 converter. And after that changed over it to an all-inclusive code converter sing a select bit. We learnt the nuts and bolts of the VHDL programming such as how to write substance, design, and the method.

6. REFERENCE:

- Mno Moris, Digital Design. New Jersey: Prentice Hall. 2.
- Wakerley, Jon, Digital-Design Principles and Practices, 3rd Edition. New Jersey: Prentice Hall.

7. APPENDIX:



First Simple-Code Converter:

```
--------------------------------------------------------------------------
-- University: ILLINOIS INSTITUTE OF TECHNOLOGY, CHICAGO
-- Designer Name: ANIRUDHA BEHERA
-- Design Name: CodeConv
-- Description: 4-bit BCD - 4-bit Excess-3
--------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity simpleCodeConv is
   Port (  A : in STD_LOGIC_VECTOR(3 downto 0);
          Y : out STD_LOGIC_VECTOR(3 downto 0));
   end simpleCodeConv;

architecture Behavioral of simpleCodeConv is

signal A2_N, A1_N, A0_N : STD_LOGIC;

begin
   -- inverted inputs
   A2_N <= NOT A(2);
   A1_N <= NOT A(1);
   A0_N <= NOT A(0);
   -- output functions
   Y(3) <= A(3) OR (A(2) AND A(0)) OR (A(2) AND A(1));
   Y(2) <= (A2_N AND A(0)) OR (A2_N AND A(1)) OR (A(2) AND A1_N AND A0
_N);
   Y(1) <= (A1_N AND A0_N) OR (A(1) AND A(0));
   Y(0) <= A0_N;
end Behavioral;
```

```
------------------------TESTBENCH---------------------------------
-- Designer Name: ANIRUDHA BEHERA
-- Design Name: Simple Code Converter Testbench
-- Description: Converts 4-bit Natural BCD to 4-bit Excess-3 BCD
------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity simpleCodeConv_test is
-- Port ( );
end simpleCodeConv_test;

architecture sim of simpleCodeConv_test is
component simpleCodeConv2
   port (  A : in std_logic_vector(3 downto 0);
           Y : out std_logic_vector(3 downto 0));
end component;

signal A : std_logic_vector(3 downto 0) := "0000";
signal Y : std_logic_vector(3 downto 0);
signal clk : std_logic := '1';

begin
   uut : simpleCodeConv2
   port map (  A => A,
               Y => Y);

   clock : process is
   begin
   clk <= '0'; wait for 5ns;
   clk <= '1'; wait for 5ns;
   -- clock period is 10ns
    end process clock;

    process(clk)
    variable count : integer := 0;
    begin
       if clk = '1' and clk'event then
          count := count + 1;
          A <= std_logic_vector(to_unsigned(count,4));
       end if;
    end process;
 end sim;
```

Second Simple-Code Converter:

```
--------------------------------------------------------------------------
-- Designer Name: ANIRUDHA BEHERA
-- Design Name: Simple Code Converter-2
-- Description:4-bit BCD-4-bit Excess-3
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity simpleCodeConv2 is
   Port (  A : in STD_LOGIC_VECTOR(3 downto 0);
           Y : out STD_LOGIC_VECTOR(3 downto 0));
   end simpleCodeConv2;

architecture Behavioral of simpleCodeConv2 is

begin
   Y <= std_logic_vector(to_unsigned(to_integer(unsigned( A )) + 3, 4));

end Behavioral;
```

```
----------------------TESTBENCH-----------------------------------------
-- Designer Name: ANIRUDHA BEHERA
-- Design Name: Simple Code Converter-2 Testbench
-- Description: Converts 4-bit Natural BCD to 4-bit Excess-3 BCD
------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity simpleCodeConv_test is
-- Port ( );
end simpleCodeConv_test;

architecture sim of simpleCodeConv_test is

component simpleCodeConv2
   port (  A : in std_logic_vector(3 downto 0);
           Y : out std_logic_vector(3 downto 0));

end component;

signal A : std_logic_vector(3 downto 0) := "0000";
signal Y : std_logic_vector(3 downto 0);
signal clk : std_logic := '1';

begin
   uut : simpleCodeConv2
   port map (  A => A,
               Y => Y);
```

```
    clock : process is
    begin
    clk <= '0'; wait for 5ns;
    clk <= '1'; wait for 5ns;
    -- clock period is 10ns
    end process clock;

    process(clk)
    variable count : integer := 0;
    begin
       if clk = '1' and clk'event then
          count := count + 1;
          A <= std_logic_vector(to_unsigned(count,4));
       end if;
    end process;
end sim;
```

Universal Code Converter:

```
--------------------------------------------------------------------------------
-- University: ILLINOIS INSTITUTE OF TECHNOLOGY, CHICAGO
-- Designer Name: ANIRUDHA BEHERA
-- Design Name: Universal Code Converter
-- Description:4-bit BCD-4-bit Excess-3 & Vice Versa
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity UniCodeConv is
   Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
          Y : out  STD_LOGIC_VECTOR (3 downto 0);
        sel : in  STD_LOGIC);
end UniCodeConv;

architecture Behavioral of UniCodeConv is
begin
   -- Excess-3 to BCD conversion-----------
   process(sel, A)
   begin
      if sel = '0' then
         case A is
            when "0011" => Y <= "0001";
            when "0100" => Y <= "0010";
            when "0101" => Y <= "0011";
            when "0110" => Y <= "0100";
            when "0111" => Y <= "0101";
            when "1000" => Y <= "0110";
            when "1001" => Y <= "0111";
            when "1010" => Y <= "1000";
            when "1011" => Y <= "1001";
            when "1100" => Y <= "0000";
            when "1101" => Y <= "0001";
            when "1110" => Y <= "0010";
            when "1111" => Y <= "0011";
            when others => Y <= "XXXX";
         end case;
      end if;

      -- BCD to excess-3 conversion------------
      if sel = '1' then
         case A is
            when "0000" => Y <= "0011";
            when "0001" => Y <= "0100";
            when "0010" => Y <= "0101";
            when "0011" => Y <= "0110";
            when "0100" => Y <= "0111";
            when "0101" => Y <= "1000";
            when "0110" => Y <= "1001";
            when "0111" => Y <= "1010";
            when "1000" => Y <= "1011";
            when "1001" => Y <= "1100";
            when "1010" => Y <= "1101";
            when "1011" => Y <= "1110";
            when "1100" => Y <= "1111";
            when others => Y <= "XXXX";
         end case;
      end if;
   end process;
 end Behavioral;

----------------------------TESTBENCH-------------------------------------
-- University: ILLINOIS INSTITUTE OF TECHNOLOGY, CHICAGO
-- Designer Name: ANIRUDHA BEHERA
-- Design Name: Universal Code Converter Testbench
-- Description: 4-bit BCD-4-bit Excess-3 & Vice Versa
---------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity UniCodeConv_tb is
end UniCodeConv_tb;

architecture Behavioral_tb of UniCodeConv_tb is
   component UniCodeConv is
      Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
             Y : out  STD_LOGIC_VECTOR (3 downto 0);
           sel : in  STD_LOGIC);
   end component;

   signal   A : std_logic_vector(3 downto 0) := "0000";
   signal   Y : std_logic_vector(3 downto 0);
   signal sel : std_logic := '0';
```

```
begin

   -- Device under test instantiation
    uut: UniCodeConv
        port map (
              A => A,
              Y => Y,
            sel => sel
        );

   -- Stimulus process
    stimulus: process
    begin
        -- Test 1: excess-3 to BCD conversion with sel = 0
        sel <= '0';
          A <= "0010";
        wait for 10 ns;
        assert Y = "0001"
    report "Test 1 failed: expected output = 0001, excess-3 to BCD conversion failed for
input 0010"
            severity error;


        -- Test 2: excess-3 to BCD conversion with sel = 0
        sel <= '0';
          A <= "1010";
        wait for 10 ns;
        assert Y = "1000"
    report "Test 2 failed: expected output = 1000, excess-3 to BCD conversion failed for
input 1010"
            severity error;
         -- Test 3: BCD to excess-3 conversion with sel = 1
         sel <= '1';
           A <= "0110";
         wait for 10 ns;
         assert Y = "1101"
     report "Test 3 failed: expected output = 1101, BCD to excess-3 conversion failed for
 input 0110"
             severity error;

         -- Test 4: BCD to excess-3 conversion with sel = 1
         sel <= '1';
           A <= "1000";
         wait for 10 ns;
         assert Y = "0011"
     report "Test 4 failed: expected output = 0011, BCD to excess-3 conversion failed for
 input 1000"
             severity error;

         wait;
     end process stimulus;
 end Behavioral;
```

PROJECT-2

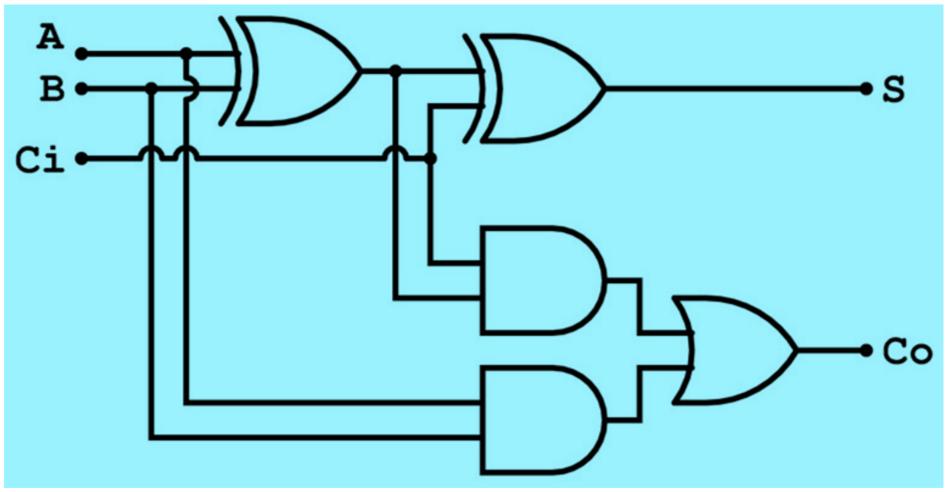
1. PURPOSE:

In this project I am going to design and implement a 4-bit RCA circuit using ZedBoard Zynq7000 development board and Xilinx Vivado as digital circuit development tool.

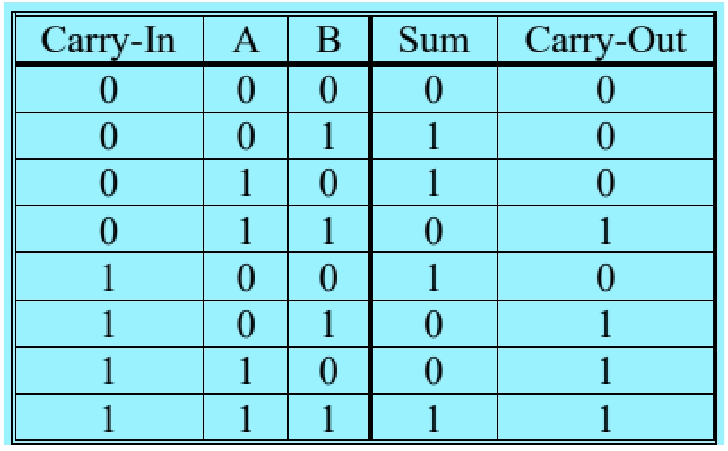
2. BACKGROUND:

Traditionally Adders are used in ALU and other computational applications like calculator etc. We have many kind of adders like, CLA, RCA, CSeA and CSA. Ripple-Carry Adder is one of the most popular adder among these adders because of its first computational function. We basically use them for addition or subtraction in any circuit or individually.

Below I have shown the logic function, circuit diagram and truth table for RCA. Here I will design one 1-bit full adder circuit first using VHDL programming.



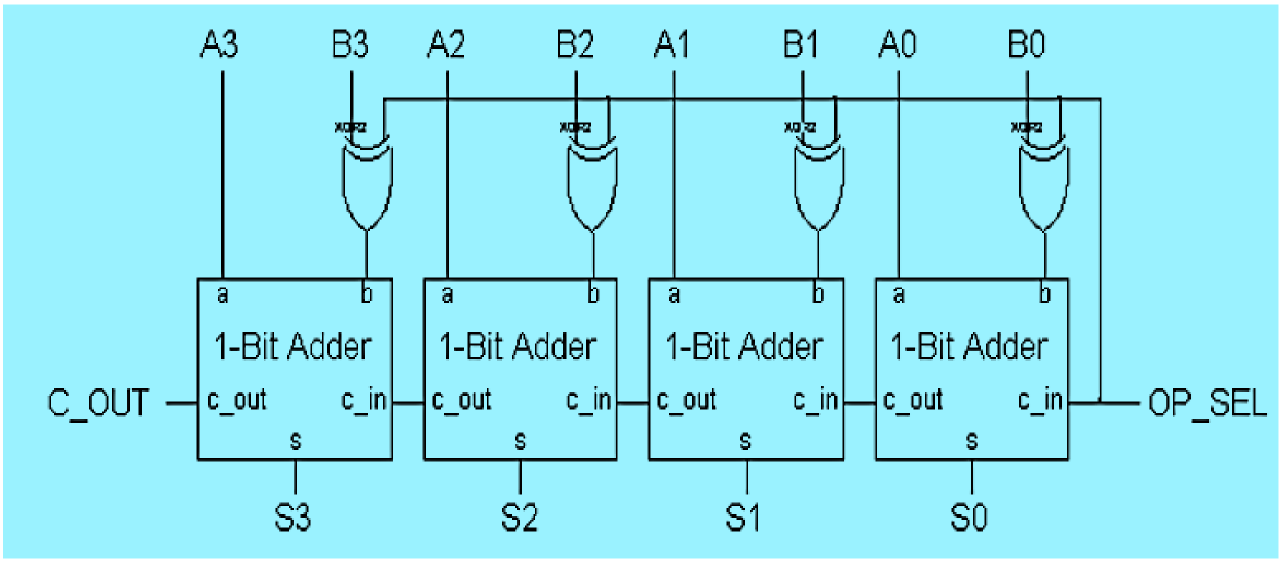
1-bit FA circuit



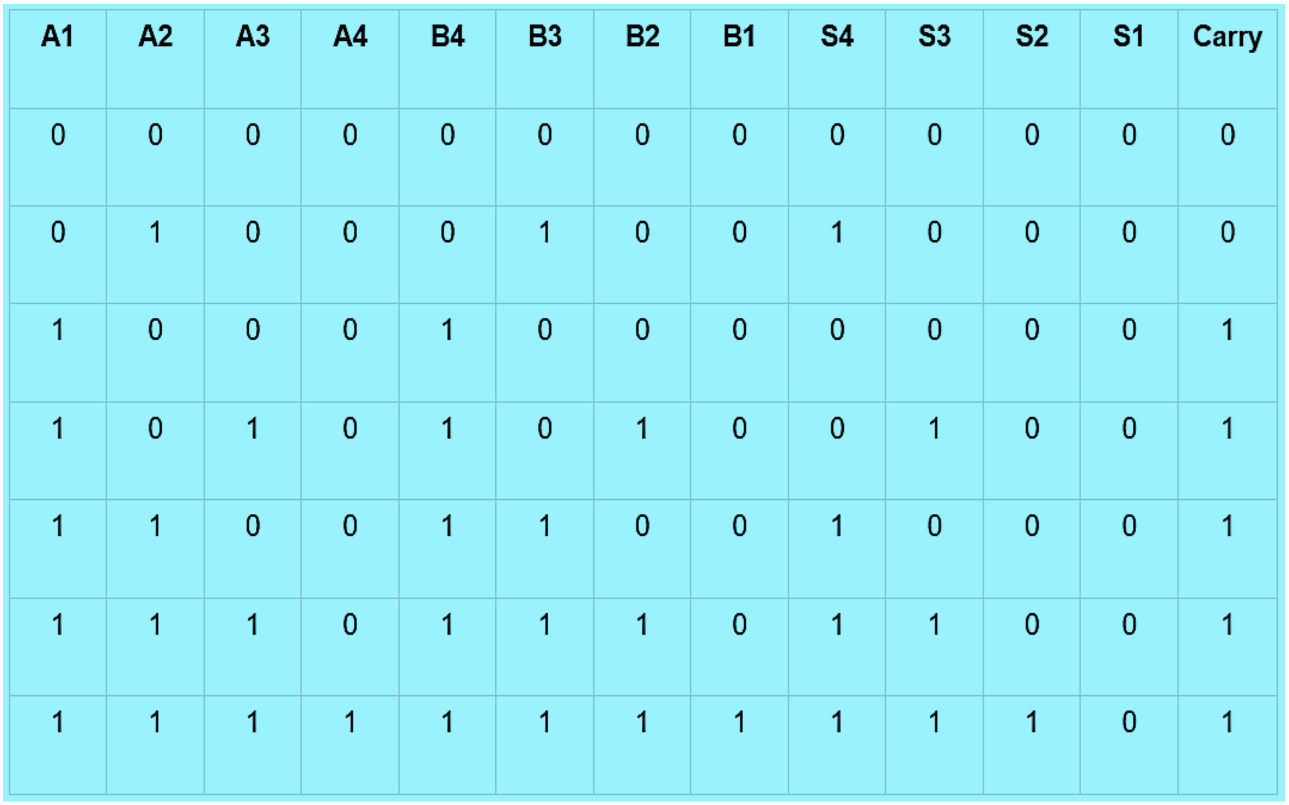
Carry-In	A	B	Sum	Carry-Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Truth-Table for 1-bit FA

After successfully designing 1-bit full adder circuit logic I will design 4-bit Ripple-Carry Adder by instantiating the 1-bit FA.



4-bit RCA Circuit



A1	A2	A3	A4	B4	B3	B2	B1	S4	S3	S2	S1	Carry
0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	0	1	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0	1
1	0	1	0	1	0	1	0	0	1	0	0	1
1	1	0	0	1	1	0	0	1	0	0	0	1
1	1	1	0	1	1	1	0	1	1	0	0	1
1	1	1	1	1	1	1	1	1	1	1	0	1

4-bit RCA Truth-Table

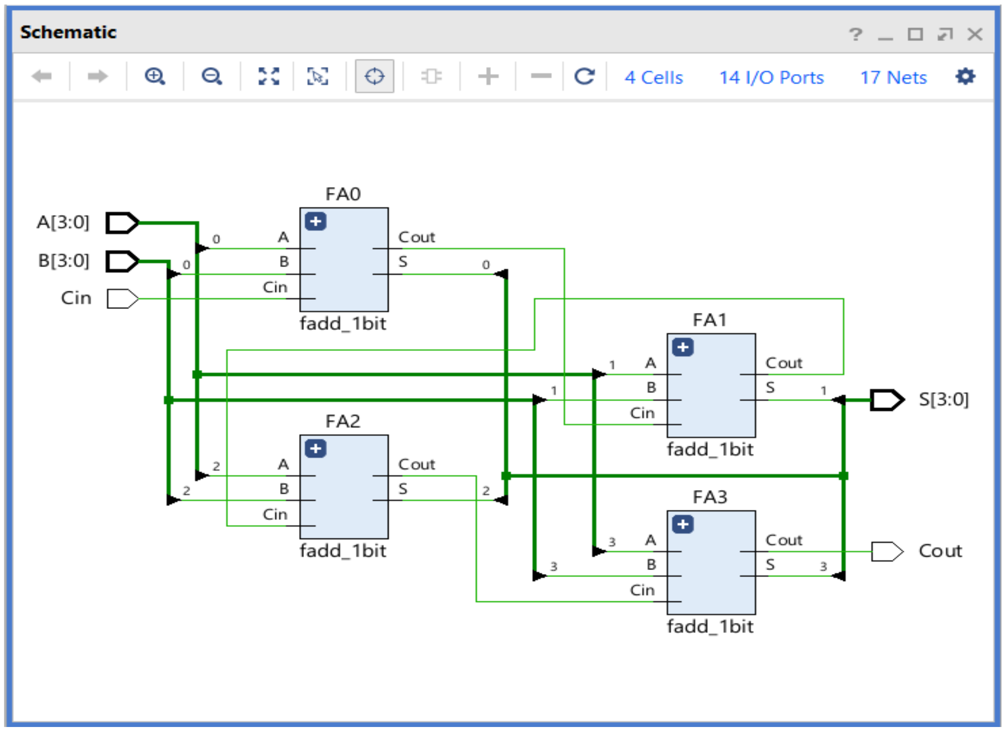
3. PROCEDURE:

- Till the third step where we will add the ZedBoard in Vivado suite will be the same as lab-1 but after that we will start our design based on this model.
- Now we will start writing a 1-bit Full Adder VHDL code logic with required pins.
- After successfully writing an error free 1-bit FA code we will create one more source file for 4-bit Ripple Carry Adder and then we will declare our pins according to 4-bit and after that we will instantiate our 1-bit FA model to this 4-bit RCA circuit.
- Exactly we will instantiate the 1-bit adder here total four times to declare the relation between pins.
- After writing .vhd code I designed testbench for the circuit where again I instantiated the 4-bit RCA component to it/s testbench.

4. RESULTS:

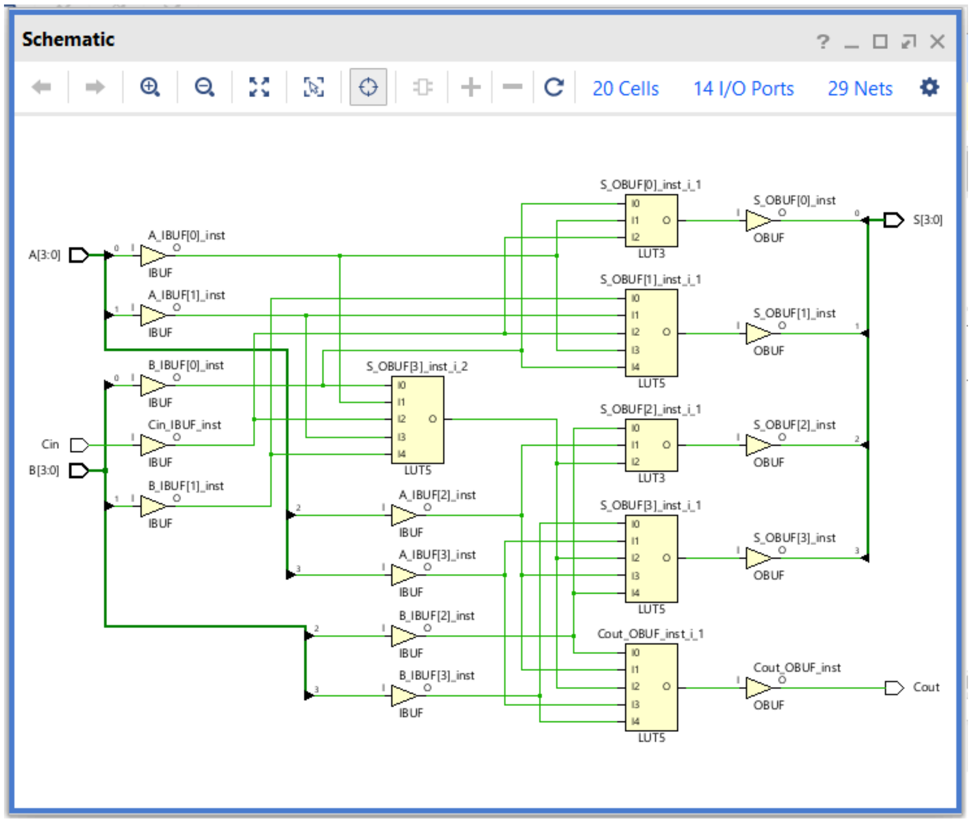
- Now to run the testbench and connect ZedBoard hardware to the Xilinx Vivado, we will follow the same steps as described in lab 1.
- After successfully following the above similar steps, we will achieve our intended result as shown below. For this lab I only executed in Vivado suite.

RTL Schematic:

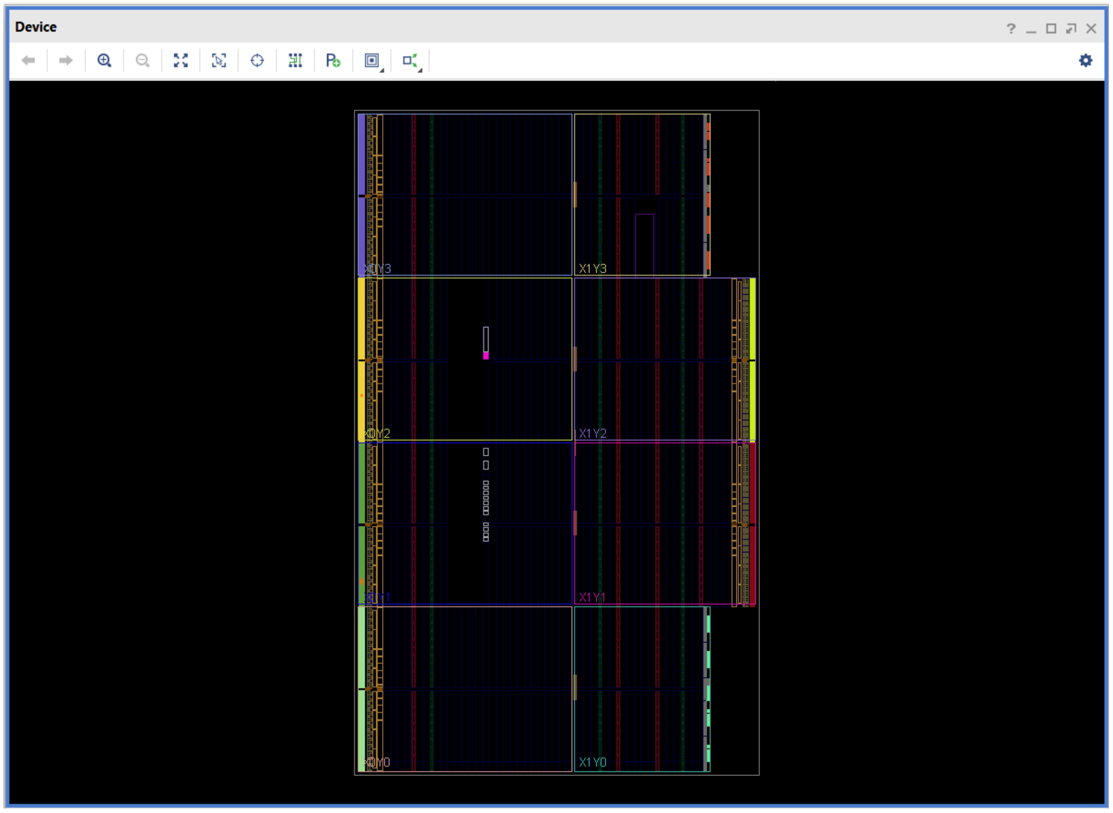


4-bit RCA Schematic

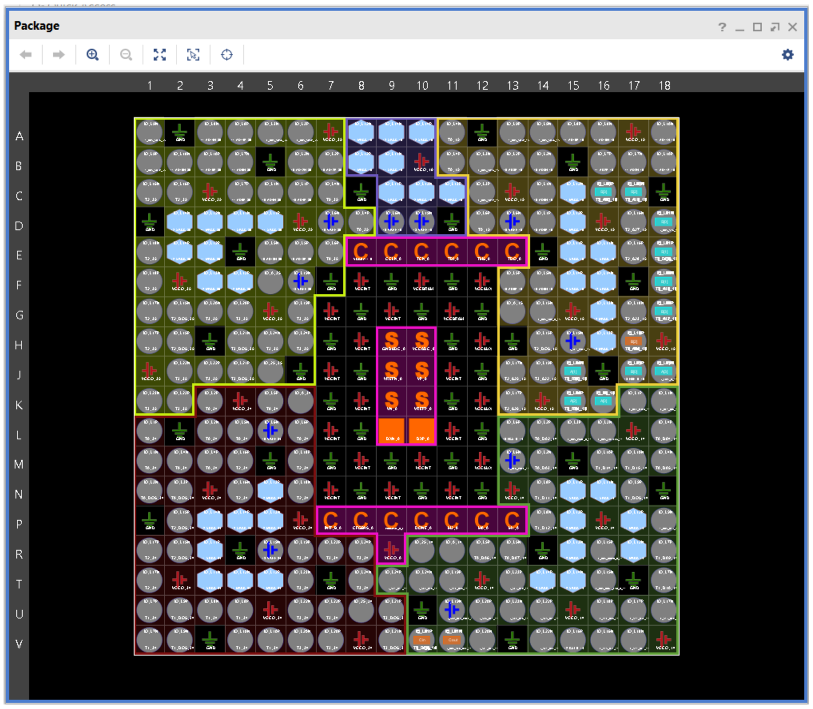
Implemented Schematic:



Floorplan:

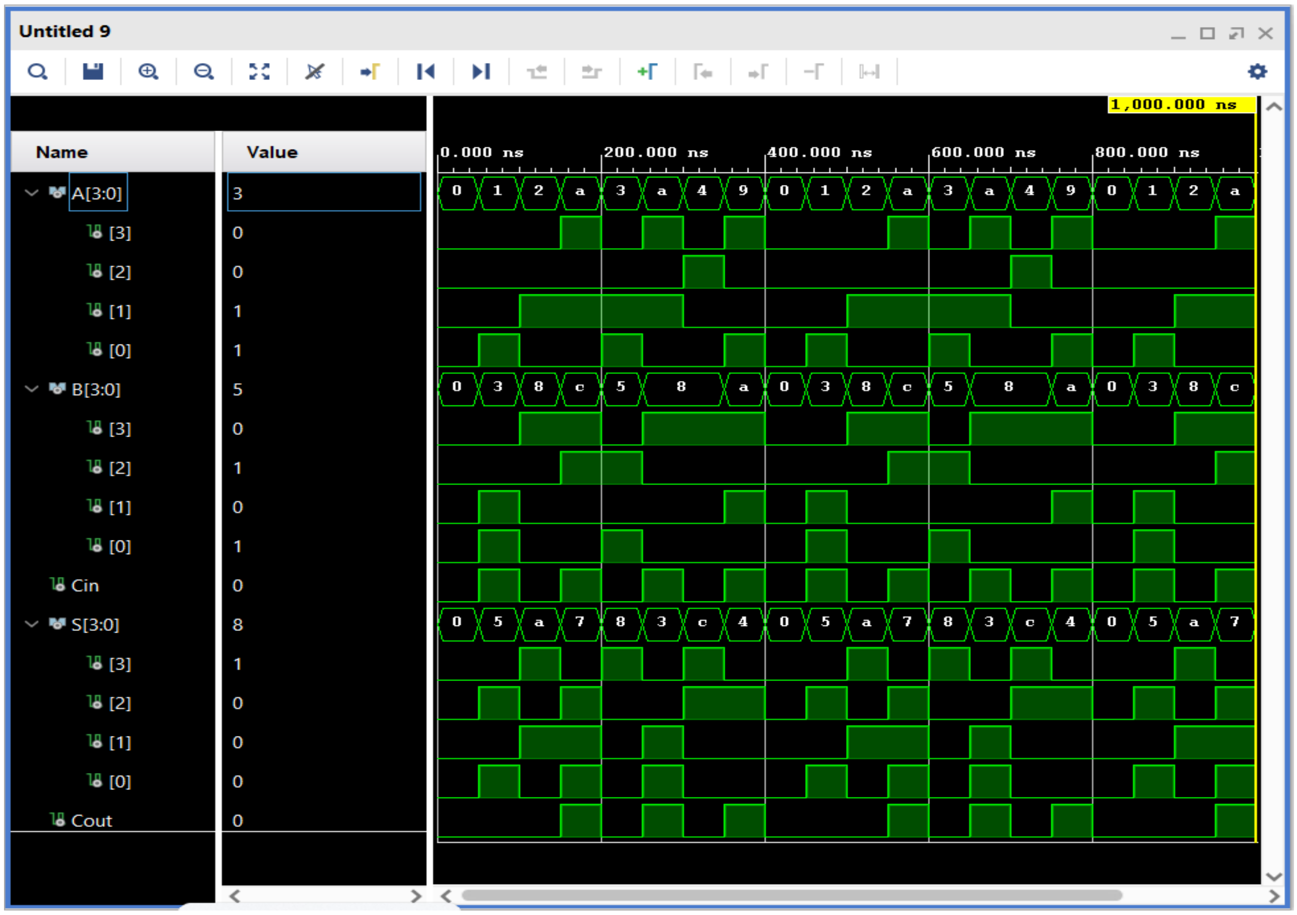


I/O planning:



Simulation:

A = 0011 (3), B = 0101 (5), A+B =1000 (8), Cin = 0, Cout = 0



5. CONCLUSION:

We examined the secluded plan procedure utilizing the case of the 4-bit swell carry viper and utilizing the total snake component in it. We moreover got the usage prerequisites for the component and how it imports everything from the initial substance of the complete snake and within the conclusion, we included the operation determination bit which triggers the XOR gates, and we get the one's complement which is used for performing the subtraction operation.

6. REFERENCE:

- Mno Moris, Digital Design. New Jersey: Prentice Hall. 2.
- Wakerley, Jon, Digital-Design Principles and Practices, 3rd Edition. New Jersey: Prentice Hall.

7. APPENDIX:

1-bit Full Adder

```
---------------1BIT FULL ADDER STRUCTURAL MODEL---------------
-- University: ILLINOIS INSTITUTE OF TECHNOLOGY, CHICAGO
-- Engineer: ANIRUDHA BEHERA
-- Design Name: 1bit_full_adder_structural
-- Module Name: fadd_1bit - Structural
-- Project Name: LAB_2_4bit_structural--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity fadd_1bit is
 Port ( A : in STD_LOGIC;
        B : in STD_LOGIC;
        Cin : in STD_LOGIC;
        S : out STD_LOGIC;
        Cout : out STD_LOGIC);
end fadd_1bit;

architecture Structural of fadd_1bit is


begin
 S <= A XOR B XOR Cin ;
 Cout <= (A AND B) OR (Cin AND A) OR (Cin AND B) ;

end Structural;
------------------4BIT FULL ADDER STRUCTURAL MODEL------------
-- University: ILLINOIS INSTITUTE OF TECHNOLOGY, CHICAGO
-- Engineer: ANIRUDHA BEHERA
-- Design Name: 4bit_fadd_struct
-- Module Name: fadd_4bit_struct - Structural
-- Project Name: LAB_2_4bit
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity RCA_4bit_struct is
Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
       B : in STD_LOGIC_VECTOR (3 downto 0);
       Cin : in STD_LOGIC;
       S : out STD_LOGIC_VECTOR (3 downto 0);
       Cout : out STD_LOGIC);
end RCA_4bit_struct; -----|input and output variables are declaired in 4bit manner
```

```
architecture Structural of RCA_4bit_struct is

component fadd_1bit is -----|1bit- full adder components declared here inorder to
implement 4bit using 1bit full adder|
port (A:in STD_LOGIC;
     B:in STD_LOGIC;
     Cin:in STD_LOGIC;
     S:out STD_LOGIC;
     Cout:out STD_LOGIC);
     end component;

     signal C1,C2,C3: STD_LOGIC; -----ripple carry
begin

FA0:fadd_1bit port map(A(0), B(0), Cin,S(0),C1);

FA1:fadd_1bit port map(A(1), B(1), C1,S(1),C2);

FA2:fadd_1bit port map(A(2), B(2), C2,S(2),C3);

FA3:fadd_1bit port map(A(3), B(3), C3,S(3),Cout);

----|above 4lines of 1bit logic construct a 4bit full adder|

end Structural;






-------------------4BIT FULL ADDER TESTBENCH ---------------------
--University: ILLINOIS INSTITUTE OF TECHNOLOGY, CHICAGO
-- Engineer: ANIRUDHA BEHERA
-- Create Date: 10/28/2022 05:04:04 AM
-- Design Name: 4bit_fadd_struct_tb
-- Module Name: fadd_4bit_struct - Structural
-- Project Name: LAB_2_4bit
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity fadd_4bit_struct_tb is
--  Port ( );
end fadd_4bit_struct_tb;

architecture Structural of fadd_4bit_struct_tb is
```

```
component RCA_4bit_struct is   ---|components of 4bit fadd, declared here|
port(A:in STD_LOGIC_VECTOR(3 downto 0);
    B:in STD_LOGIC_VECTOR(3 downto 0);
    Cin:in STD_LOGIC;
    S:out STD_LOGIC_VECTOR(3 downto 0);
    Cout:out STD_LOGIC);
    end component;

signal A:STD_LOGIC_VECTOR(3 downto 0):="0000";
signal B:STD_LOGIC_VECTOR(3 downto 0):="0000";
signal Cin:STD_LOGIC:='0';
signal S:STD_LOGIC_VECTOR(3 downto 0):="0000";
signal Cout:STD_LOGIC:='0';
    -------------|each signals port width is increased to four bit testbench|
begin

UUT: RCA_4bit_struct ----|new variables declared to existing variables of 4bit strucral
model|
port map(A => A,
B => B,
Cin => Cin,
S => S,
Cout => Cout);
process begin -----input values are declared below to perform testbench|

A<= "0000";
B<="0000";
Cin<='0';
wait for 50 ns;

A<= "0001";
B<="0011";
Cin<='1';
wait for 50 ns;
```

```
A<= "0010";
B<="1000";
Cin<='0';
wait for 50 ns;

A<= "1010";
B<="1100";
Cin<='1';
wait for 50 ns;

A<= "0011";
B<="0101";
Cin<='0';
wait for 50 ns;

A<= "1010";
B<="1000";
Cin<='1';
wait for 50 ns;

A<= "0100";
B<="1000";
Cin<='0';
wait for 50 ns;

A<="1001";
B<="1010";
Cin<='1';
wait for 50 ns;

    end process;
end Structural;
```

PROJECT-3

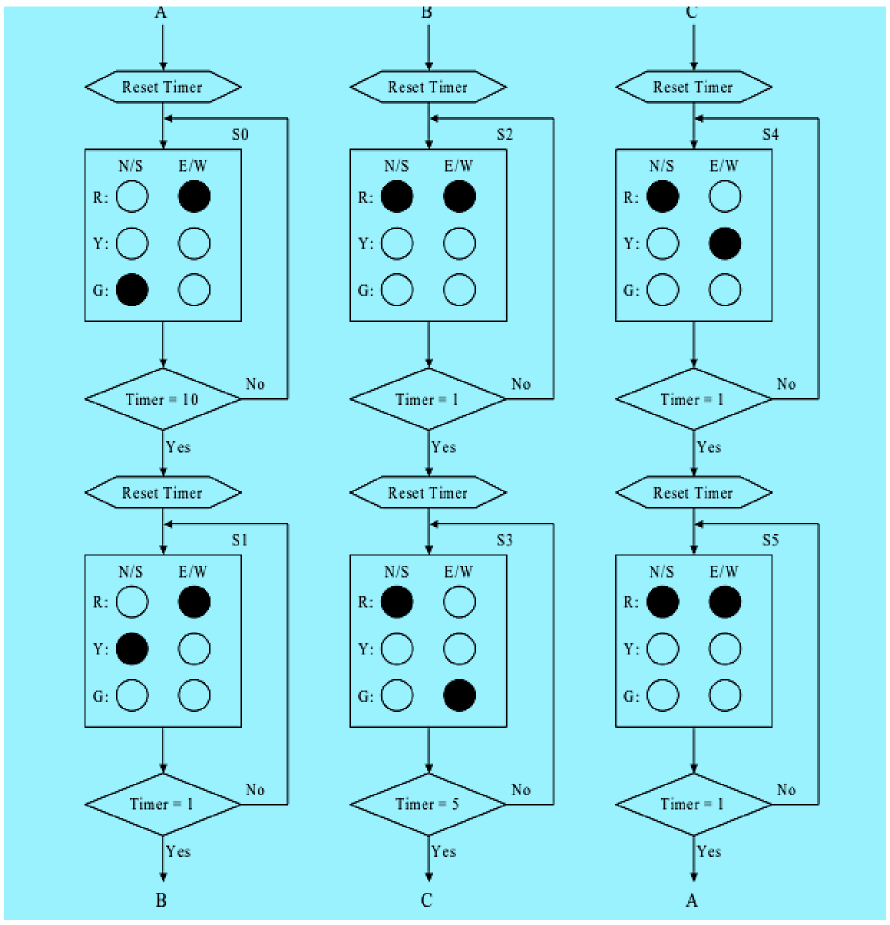
1. PURPOSE:

In this project I am going to design and implement a Traffic Light Controller circuit and its testbench using ZedBoard Zynq7000 development board and Xilinx Vivado as digital circuit development tool.

2. BACKGROUND:

Traffic light Controller is a very popular and commercially available Finite State machine model. It has huge impact in our day-to-day life. For us the main challenge for design and implement this FSM model is going to generate or regulate the timing of this lights accurately on-time.

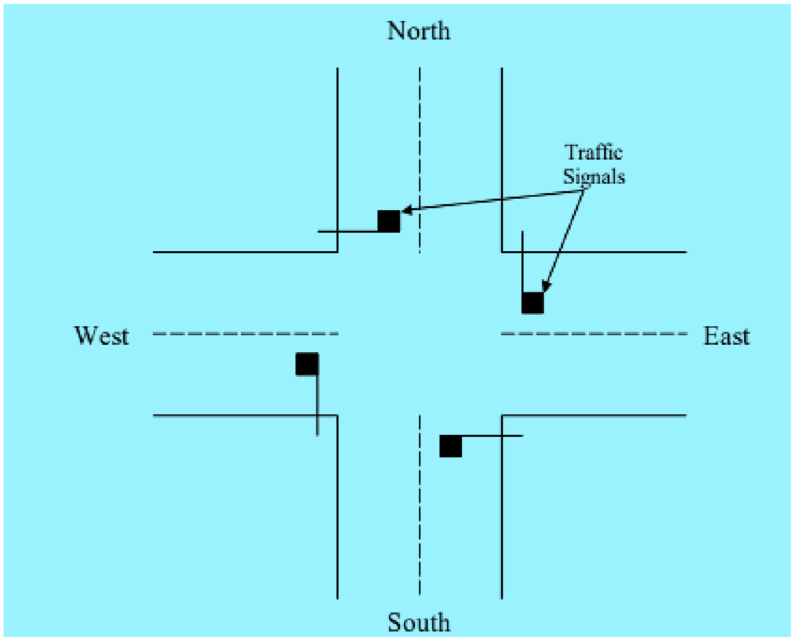
Here in this proposed model I am going to design a sensor based traffic light controller which will be capable of generating timing constrains accurately. Below I have shown a flow chart of traffic light controller to get an overall idea how actually it works.



Simple Traffic Light Controller flow chart

From the above shown simple traffic light controller flow chart, Clearly, we can see that towards the North/South the green light stays for 10 seconds and then it turns to yellow for 1 seconds and then again turns to red for 8 seconds and then at the end it turns to the green light again For the East/West it works exact opposite manner. But in the sensor-based traffic light controller the sensor used on each side on the intersection or traffic. If any vehicle is sensed on the road by the sensor, then the traffic lights will get activated and change into which ever color we have implement.

Below you can see I have shown an illustrative image of how a simple traffic light system works.



Illustrative image of simple traffic light system

3. PROCEDURE:

Two-Way Intersection with No Sensors

- Till the third step where we will add the ZedBoard in Vivado suite will be the same as lab-1 but after that we will start our design based on proposed traffic light model.
- First, I focus on simulating design so that my light should obey the specifications of the flow chart.
- I will use same VHDL programming again for designing and similarly I will start my design from declaring Port names and their functions.
- After defining ports, I will move towards declaring signals and then I will start writing logic for function of the traffic light according to our specifications.

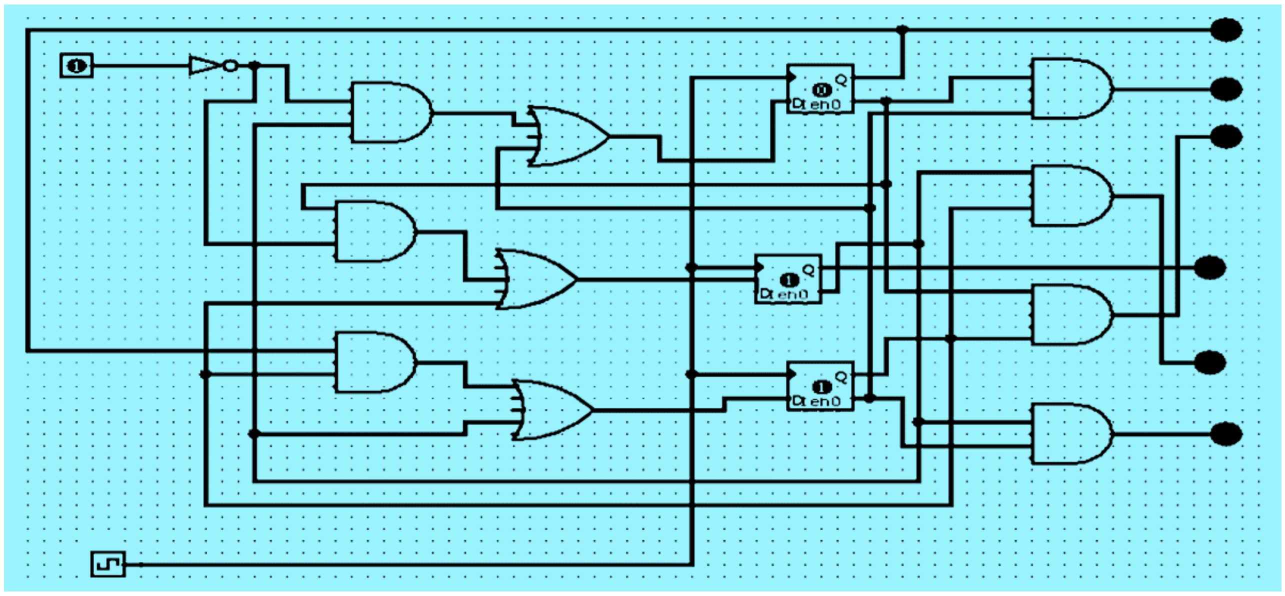
- Incorporate one of the 100-MHz clock divider modules from Reference section B of this research facility manual in your plan and interface its yield to the clock flag of the activity light controller. Either clock divider circuit may be utilized with this plan in any case, the selectable recurrence module may make physical testing of our circuit less time devouring.
- We have to program the ZedBoard board together with our designed model and test the physical circuit. Once you have confirmed that your plan capacities legitimately, appear the lab teachers its behavior some time recently moving on to another step in this research facility.

Two-Way Intersection with Sensors

- Enter your plan for the moment two-way activity controller into the Xilinx Vivado improvement environment.
- Mimic your plan to guarantee the timing of the lights is fitting and fits the determinations of the flowchart in Foundation Area 2.2 of this research facility explore. Be beyond any doubt to reenact all combinations of sensor inputs. In the event that either sensor or both are activated, the activity light timing ought to be started. The as it were way for the timing prepares to be ended is for both sensors to be turned off. On the off chance that your plan does not reenact appropriately, rectify all plan issues some time recently proceeding to another method in this research facility.
- Incorporate one of the I 00-MHz clock divider modules from Reference section B of this research facility manual in your plan and interface its yield to the clock flag of your activity light controller.
- Program the ZedBoard board together with your design and physically test your circuit. Once you have confirmed that your plan capacities appropriately, appear the lab teachers its behavior some time recently completing this research facility.

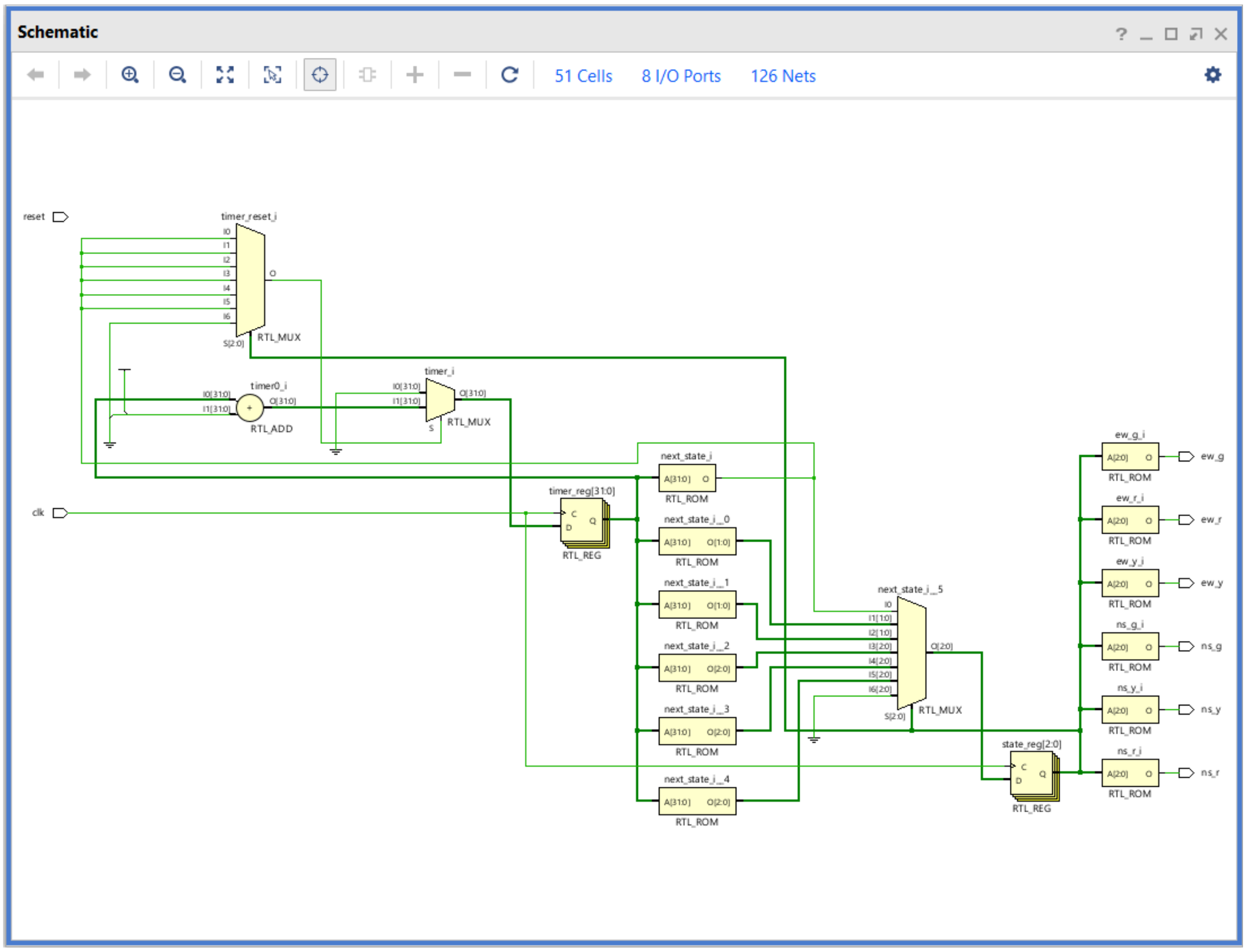
4. RESULTS:

- Now to run the testbench and interface ZedBoard equipment to the Xilinx Vivado, we'll take after the same steps as depicted in lab.
- After successfully following the over comparable steps, we are going accomplish our expecting result as appeared underneath. For this lab I as it were executed in Vivado suite, so I only executed using it and attached all required output results and schematic images.
- Below I have shown a typical Traffic Light Controller logic gate schematic.



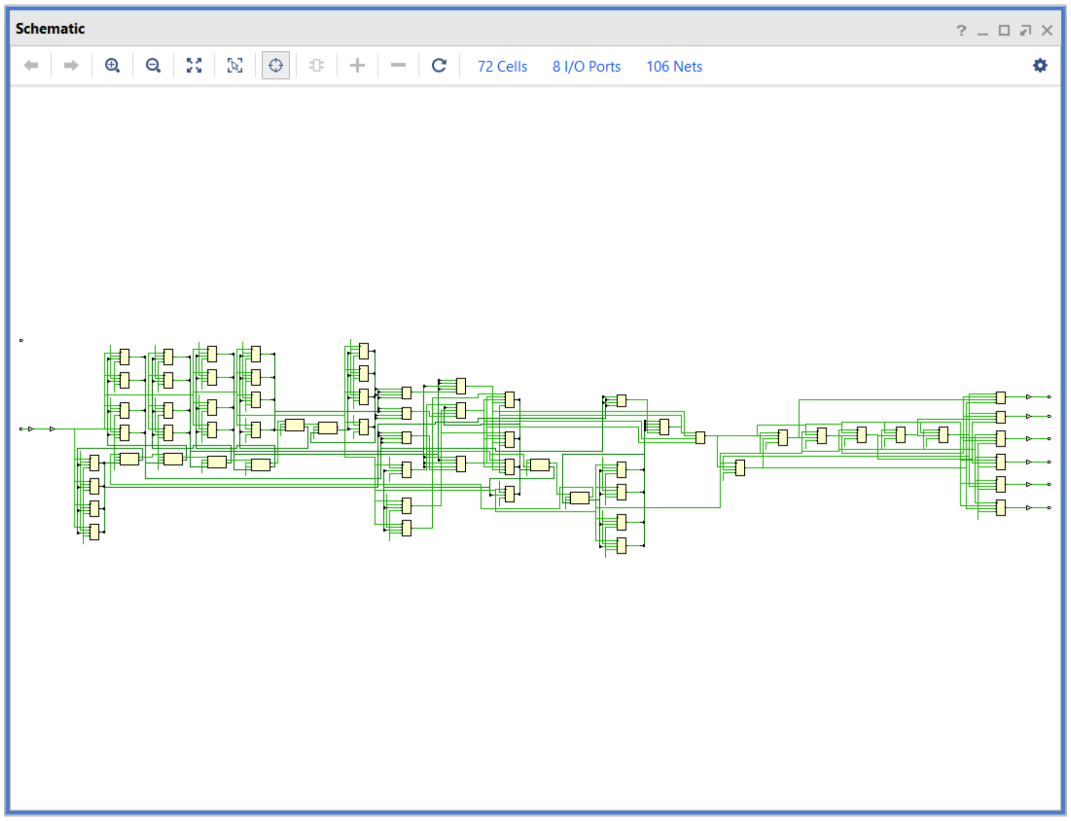
Logic gate diagram of traffic control system

RTL Schematic:

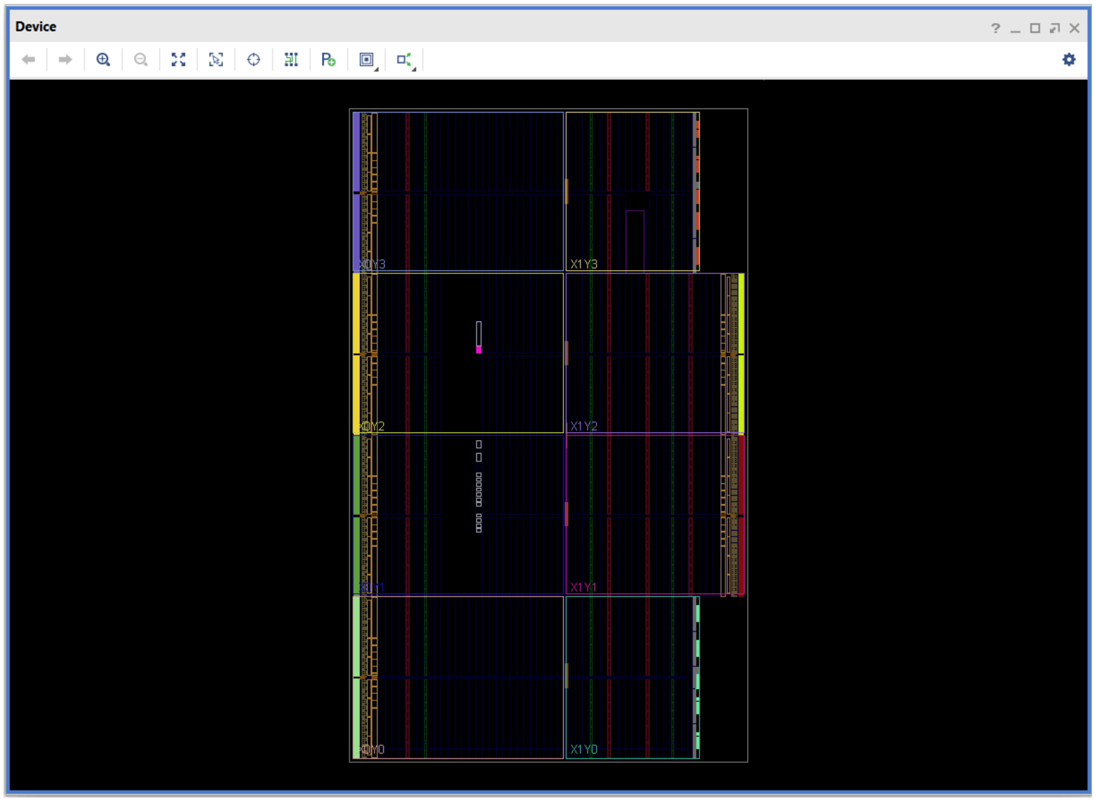


Traffic-Light Controller Schematic

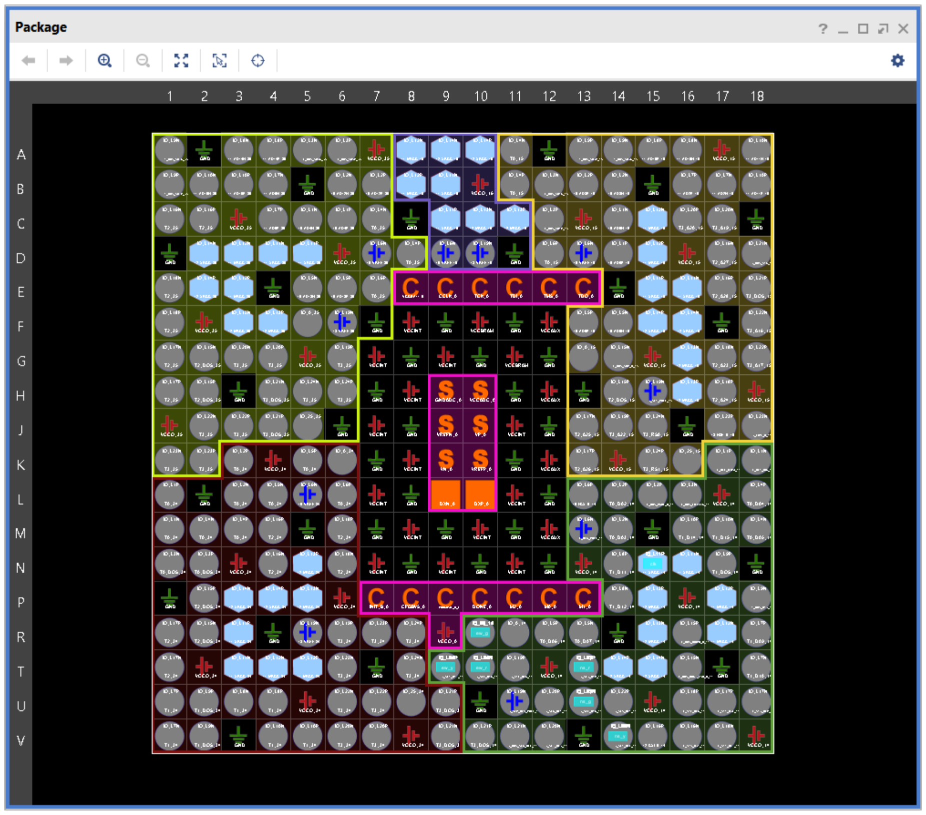
Implemented Schematic:



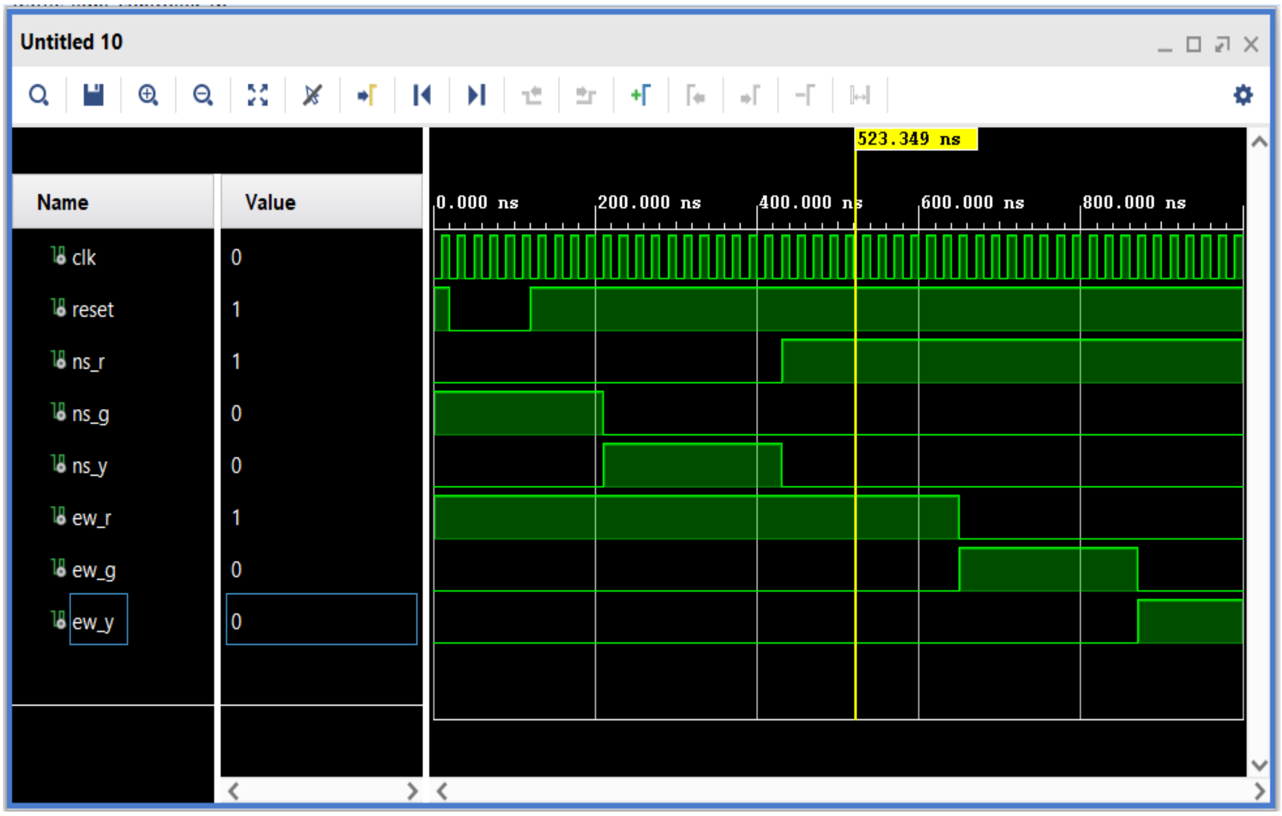
Floorplan:



I/O plan:



Simulation:



Name	Value
clk	0
reset	1
ns_r	1
ns_g	0
ns_y	0
ew_r	1
ew_g	0
ew_y	0

5. CONCLUSION:

We effectively actualize the activity light controller utilizing VHDL programming. We learnt how to utilize then when and case articulation within the VHDL programming, this investigated us to more to the exchanging conditions and how to utilize them in our code. We too learnt how to utilize the clocks rising edge for changing the state within the program and doing so we parallelly alter the signals at each closes of the crossing point. This tackles numerous issues on the crossing point of any street.

6. REFERENCE:

- Mno Moris, Digital Design. New Jersey: Prentice Hall. 2.
- Wakerley, Jon, Digital-Design Principles and Practices, 3rd Edition. New Jersey: Prentice Hall.

7. APPENDIX:

Traffic Light Controller

```
---------------TRAFFIC LIGHT CONTROLLER-----------------------------
-- University: ILLINOIS INSTITUTE OF TECHNOLOGY, CHICAGO
-- Engineer: ANIRUDHA BEHERA
-- Design Name: traffic_light_controller
-- Module Name: Traffic_light_controller - Behavioral
-- Project Name: LAB_8
---------------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity Traffic_Light_Controller is
   port (  clk : in std_logic;
           reset : std_logic;
           ns_r, ns_y, ns_g,
           ew_r, ew_y, ew_g : out std_logic);
end Traffic_Light_Controller;

architecture behavioral of Traffic_Light_Controller is
-- traffic light controlled by FSM with timer
-- 10 clock cycles for green

-- 1 clock cycle for yellow
-- 1 clock cycle for both red
type state_type is (s0, s1, s2, s3, s4, s5);
signal state, next_state : state_type;
signal timer : integer := 0;
signal timer_reset : std_logic := '0';
```

```
begin
   sync_process : process (clk) is
   -- synchronize state transitions and timer with clock
   begin
      if (rising_edge(clk)) then
         state <= next_state;
         if (timer_reset = '1') then
            timer <= 0; -- reset timer
         else
            timer <= timer + 1;
         end if;
      end if;
   end process;

   output_decode : process (state) is
   begin
      case (state) is
         when s0 =>
            -- north/south gets green
            ns_r <= '0';
            ns_y <= '0';
            ns_g <= '1';
            -- east/west gets red
            ew_r <= '1';
            ew_y <= '0';
            ew_g <= '0';
            -- hold state for 10 seconds
         when s1 =>
            -- north/south gets yellow
            ns_r <= '0';
            ns_y <= '1';
            ns_g <= '0';
            -- east/west gets red
            ew_r <= '1';
            ew_y <= '0';
            ew_g <= '0';

         when s2 =>
            -- north/south gets red
            ns_r <= '1';
            ns_y <= '0';
            ns_g <= '0';
            -- east/west gets red
            ew_r <= '1';
            ew_y <= '0';
            ew_g <= '0';

         when s3 =>
            -- north/south gets red
            ns_r <= '1';
            ns_y <= '0';
            ns_g <= '0';
            -- east/west gets green
            ew_r <= '0';
            ew_y <= '0';
            ew_g <= '1';
          when s4 =>
             -- north/south gets red
             ns_r <= '1';
             ns_y <= '0';
             ns_g <= '0';
            -- east/west gets yellow
            ew_r <= '0';
            ew_y <= '1';
            ew_g <= '0';

        when s5 =>
            -- north/south gets red
            ns_r <= '1';
            ns_y <= '0';
            ns_g <= '0';
            -- east/west gets red
            ew_r <= '1';
            ew_y <= '0';
            ew_g <= '0';

        when others =>
            -- north/south gets unknown
            ns_r <= '0';
            ns_y <= '0';
            ns_g <= '0';
            -- east/west gets unknown
            ew_r <= '0';
            ew_y <= '0';
            ew_g <= '0';
    end case;
end process;

next_state_decode : process (state, timer) is
begin
    next_state <= s0; -- initial state
    timer_reset <= '0'; -- negate timer reset
    case (state) is
when s0 =>
    if (timer = 10) then
    next_state <= s1;
    timer_reset <= '1';
    else
    next_state <= s0;
    end if;
when s1 =>
    if (timer = 10) then
    next_state <= s2;
    timer_reset <= '1';
    else
    next_state <= s1;
    end if;
when s2 =>
    if (timer = 10) then
    next_state <= s3;
    timer_reset <= '1';
    else
    next_state <= s2;
    end if;
```

```
          when s3 =>
             if (timer = 10) then
             next_state <= s4;
             timer_reset <= '1';
             else
             next_state <= s3;
             end if;
           when s4 =>
              if (timer = 10) then
              next_state <= s5;
              timer_reset <= '1';
              else
              next_state <= s4;
              end if;
           when s5 =>
              if (timer = 10) then
              next_state <= s0;
              timer_reset <= '1';
              else
              next_state <= s5;
              end if;
           when others =>
              next_state <= s0;
        end case;
     end process;
end behavioral;

 -------------TRAFFIC LIGHT CONTROLLER TESTBENCH--------------
 -- University: ILLINOIS INSTITUTE OF TECHNOLOGY, CHICAGO
 -- Engineer: ANIRUDHA BEHERA
 -- Design Name: traffic_light_controller_tb
 -- Module Name: Traffic_light_controller - Behavioral
 -- Project Name: LAB_8
 -- Revision 0.01 - File Created
 -- Additional Comments:
 --------------------------------------------------------------------------------

 library ieee;
 use ieee.std_logic_1164.all;

 entity traffic_light_controller_tb is
 end traffic_light_controller_tb;

 architecture testbench of traffic_light_controller_tb is
    -- import the entity to be tested
    component Traffic_Light_Controller is
    port (  clk : in std_logic;
           reset : std_logic;
           ns_r, ns_y, ns_g,
           ew_r, ew_y, ew_g : out std_logic);
    end component;
    signal ew_y : std_logic;

begin
    -- instantiate the entity to be tested
    uut : Traffic_Light_Controller
     -- declare signals for input and output ports
     signal clk : std_logic := '0';
     signal reset : std_logic := '0';
     signal ns_r : std_logic;
     signal ns_g : std_logic;
     signal ns_y : std_logic;
     signal ew_r : std_logic;
     signal ew_g : std_logic;
      port map(
          clk => clk,
          reset => reset,
          ns_r => ns_r,
          ns_g => ns_g,
          ns_y => ns_y,
          ew_r=> ew_r,
          ew_g=> ew_g,
          ew_y=> ew_y );

  -- clock process with 50 ns period
  process
  begin
      while true loop
          clk <= '0';
          wait for 10 ns;
          clk <= '1';
          wait for 10 ns;
      end loop;
  end process;

  -- reset process
  process
  begin
      reset <= '1';
      wait for 20 ns;
      reset <= '0';
      wait for 100 ns;
      reset <= '1';
      wait;
    end process;
end testbench;
```

PROJECT-4

1. PURPOSE:

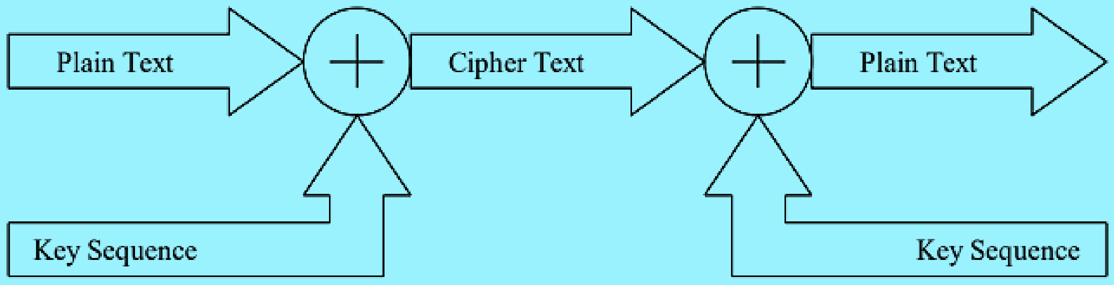
In this project I am going to design and implement a Data Encryption/Decryption using LFSRs circuit with ZedBoard Zynq7000 development board and Xilinx Vivado as digital circuit development tool.

2. BACKGROUND:

In this project we will consider concepts with respect to the cryptography and encryption on the ZedBoard board. We got a bunch of encrypted ASCII content and need to unscramble them utilizing the required strategy. In this project we just encrypt the 8-bit piece of information utilizing 8-bit key sequence, which is simple for usage other than the complex cryptography calculations. The most rationale behind the cryptography is throughout the XOR-gate behavior. The most working of the encryption can be clarified utilizing the square graph underneath.

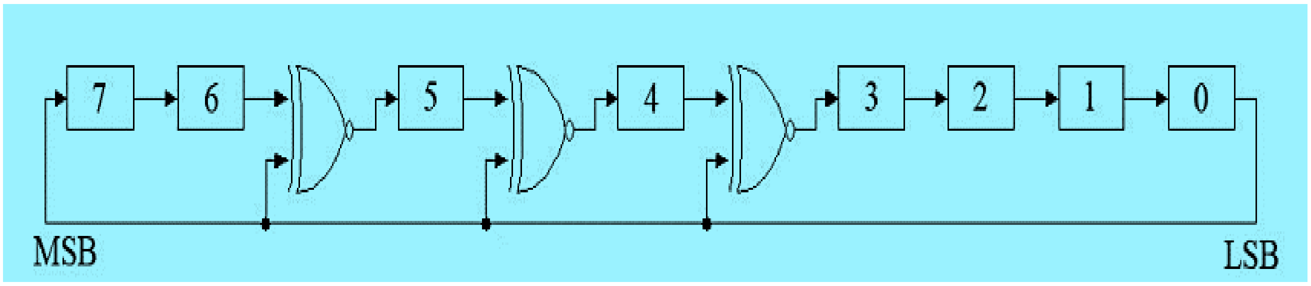
$$P \oplus K \oplus K = P \oplus 0 = P$$

The entire encryption/decryption process is shown in the following diagram:



Basic Encryption/Decryption methodology

This appear simple but in case we utilize the key which is known anybody can split it at any time. So, for dodging this we ought to alter different perspectives of the key powerfully amid the encryption and unscrambling at both the closes sender and collectors. So that it'll get more stable and can't be broken by anybody. For the era of an irregular key, we utilize the LFSR with XOR entryways put at arbitrary places. This will make beyond any doubt that the key created isn't speculated by any one and cannot violets the security. The straightforward plan for this LFSR is appeared underneath.



An LFSR design used to generate key values.

For guaranteeing that both the getting and sending side employments the same key we must ought to indicate somethings such as number of shifts in inspected key and the beginning esteem of LFSR bits. This will let both the client to use the same key for the encryption and decoding conjointly gives extra security instead of utilizing straightforward increase.

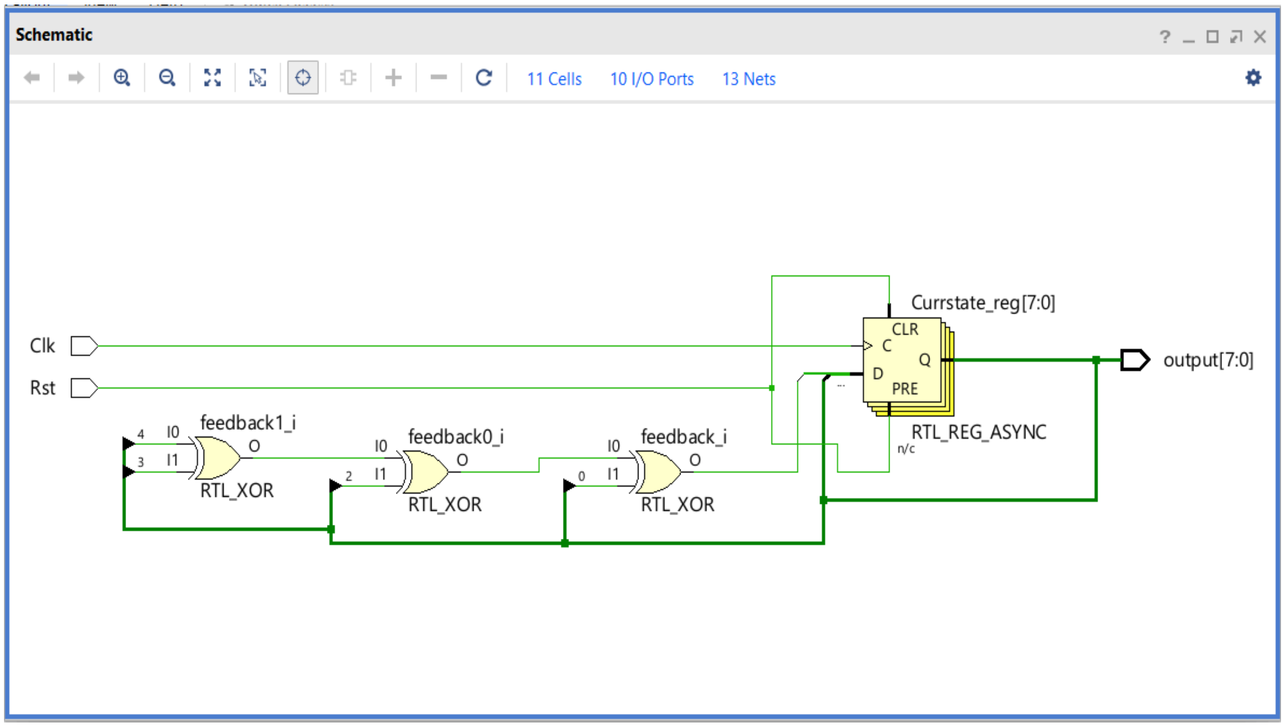
3. PROCEDURE:

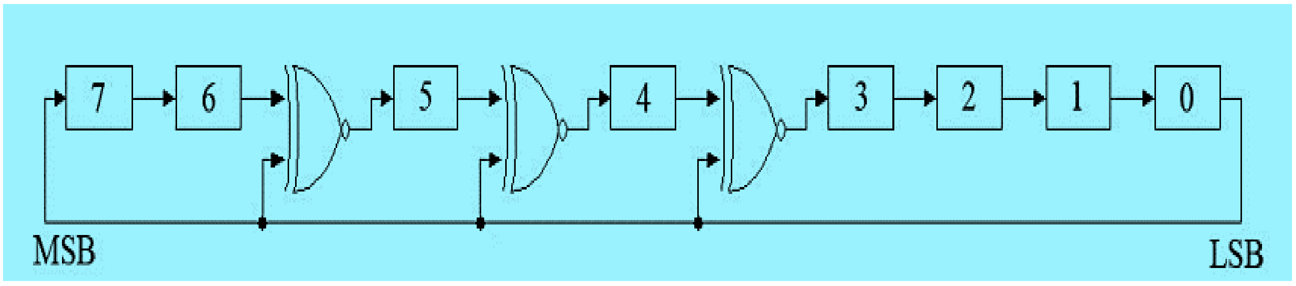
- We open Xilinx Vivado suite in arrange to plan our proposed code converter demonstrate.
- In the primary page we'll select to “Create Modern Project” alternative and after that I gave name to my plan records and included Zedboard oblige record in another page.
- Then within the third page I chosen my Zedboard from sheets for testing purposes.
- All the programs are composed in VHDL programming which is known as a really well-known equipment portrayal dialect.
- After indicating the pins to the show utilizing VHDL dialect we begun characterizing the logic of basic encryptor or descriptor model based on LFSRs.
- After composing the .vhd code for code converter I begun composing testbench for the proposed plan demonstrate which can be utilized for testing purposes.
- In that testbench we are going instantiate the already written encryptor module and its components.
- Then we'll include signals to allot factors and type in the test code for execution testing.
- To understand the given inputs and suggexted exicuition methords we can follow the given LAB-9, In the Lab paragraph from Manual.

5. RESULTS:

I performed the simulation according to given parameters of LFSRs logic and then all the schematic images and simulation results were captured which are shown below.

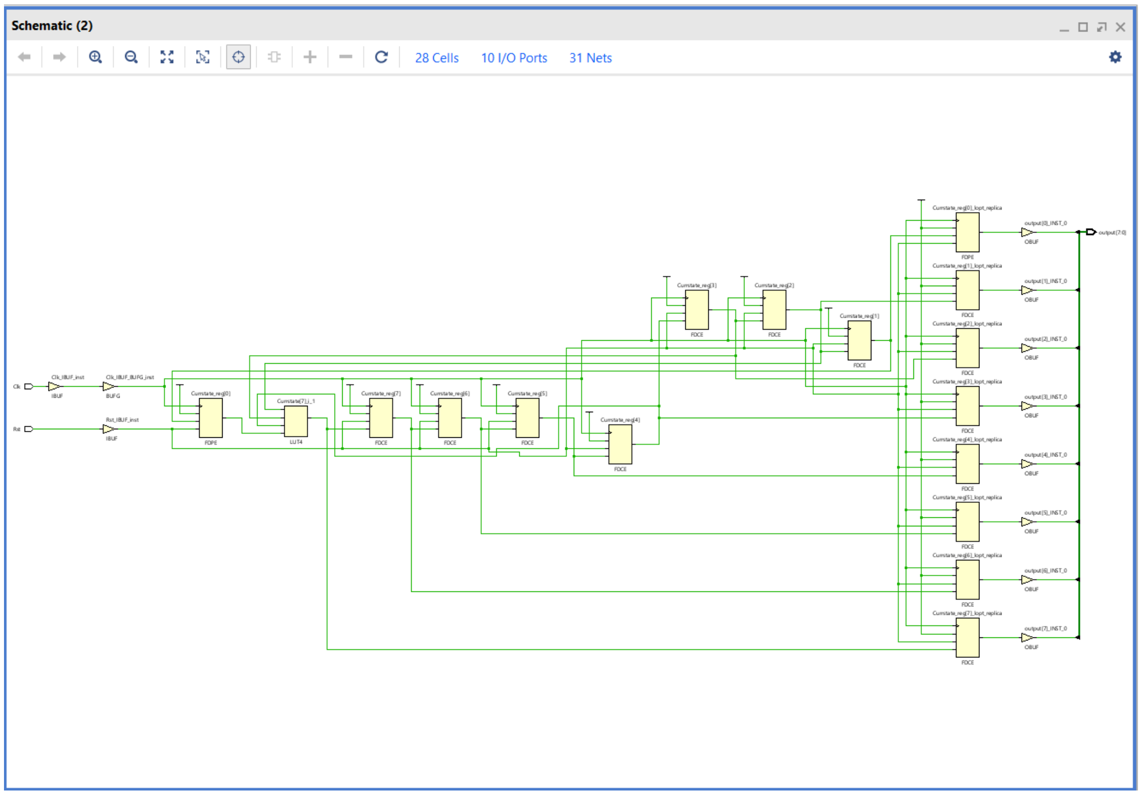
RTL Schematic:



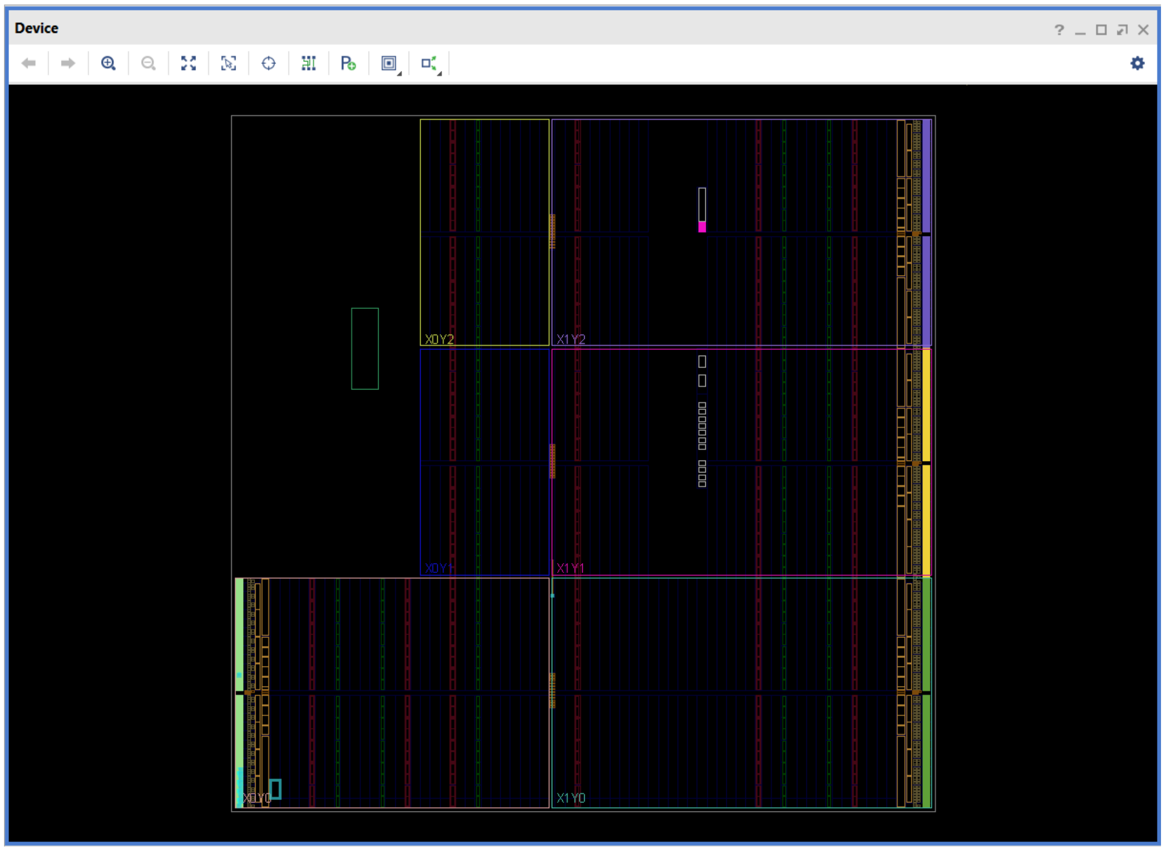


We can see above two logic simulation are very much similar and same. First one is given for our reference and the second one I got from RTL synthesis of my designed VHDL program. Seven registers and three XOR gates in both simulation and XOR gates are connected to feedback path as required or described.

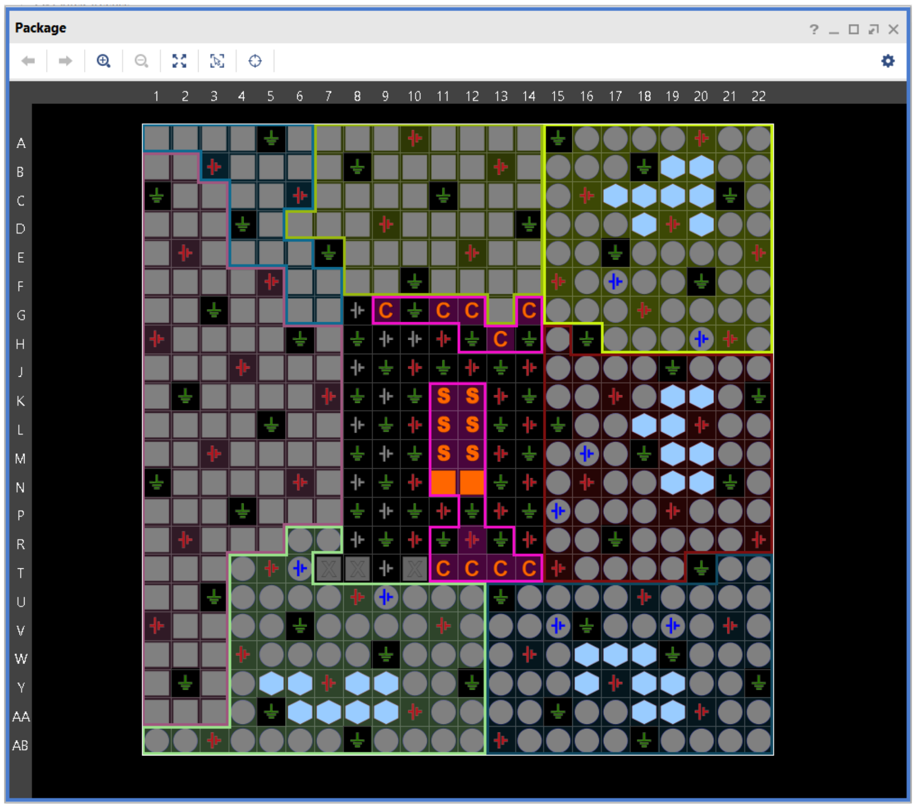
Implemented Schematic:



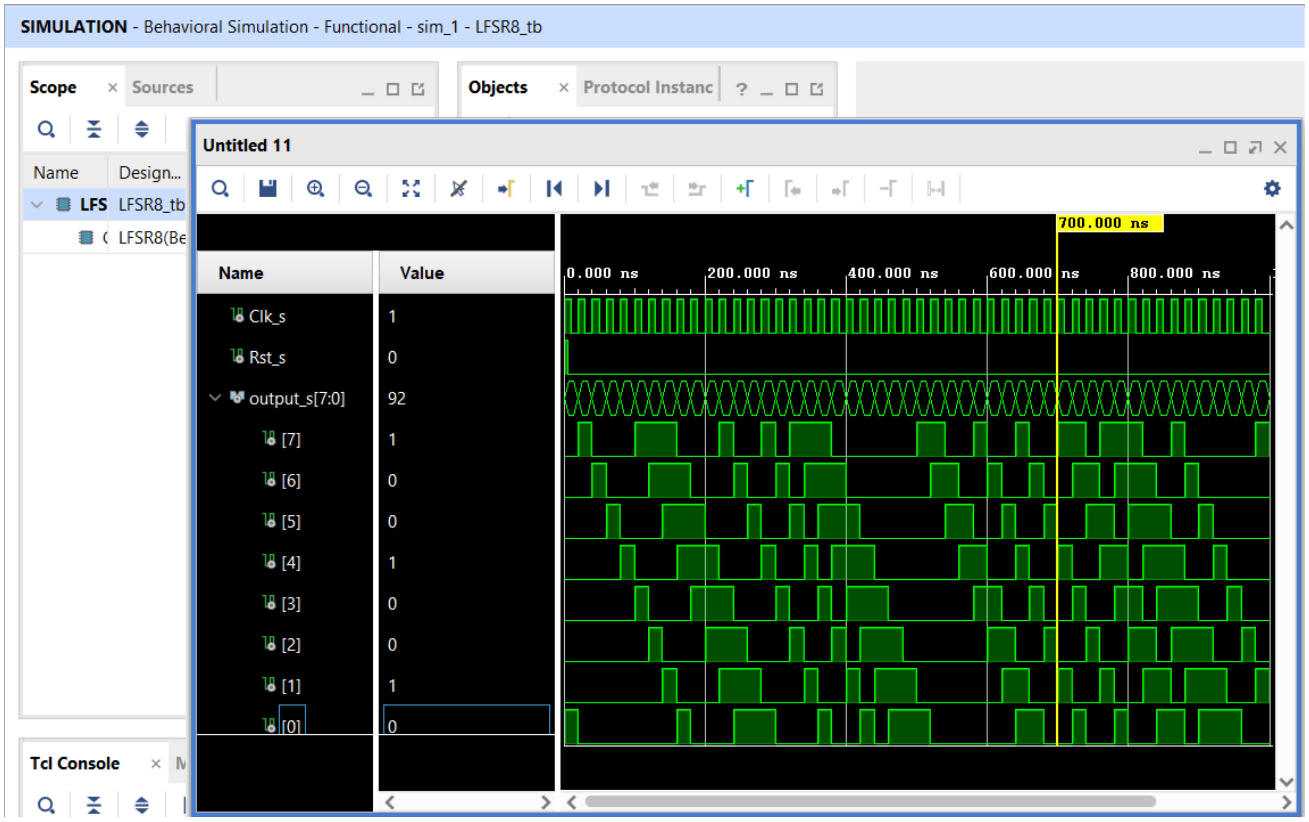
Floorplan:



I/O Plan:



Simulation:



Name	Value
Clk_s	1
Rst_s	0
output_s[7:0]	92
[7]	1
[6]	0
[5]	0
[4]	1
[3]	0
[2]	0
[1]	1
[0]	0

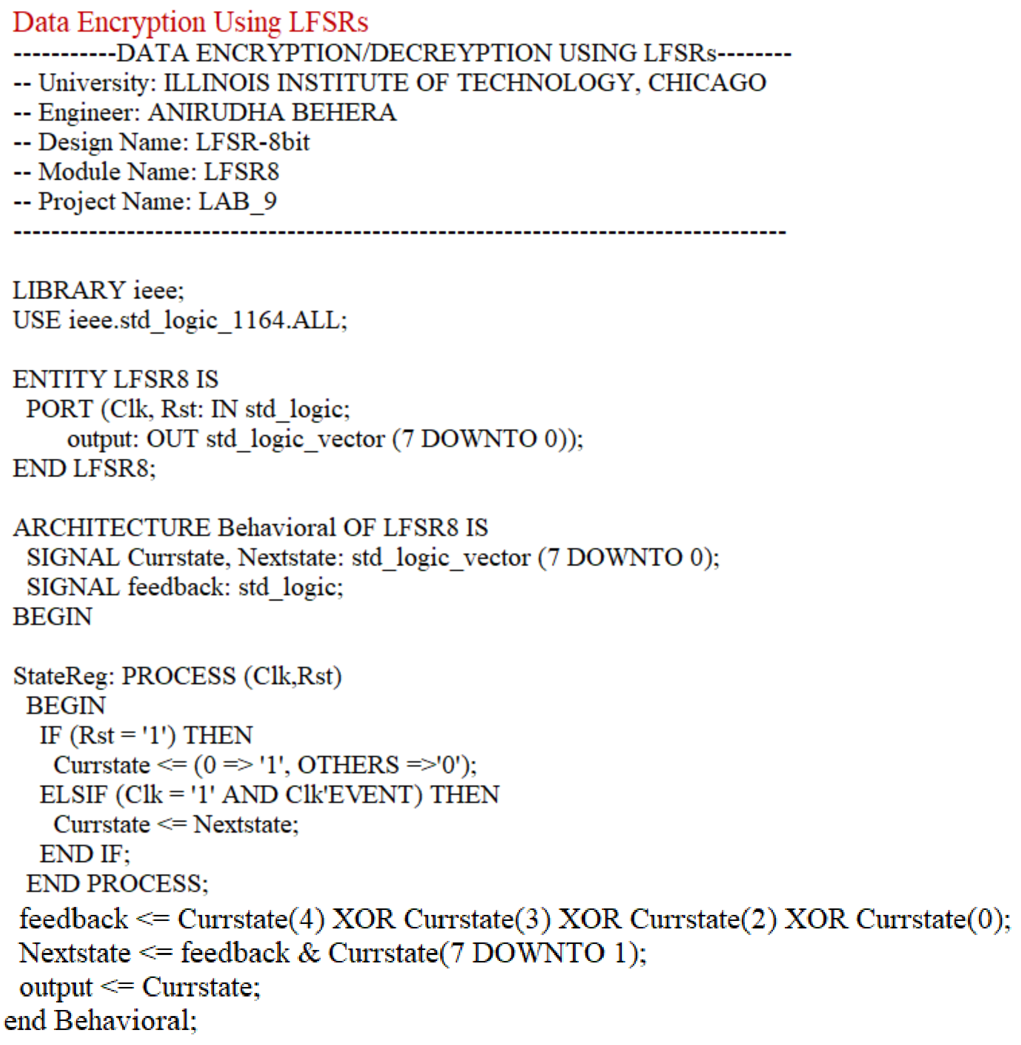
5. CONCLUSION:

In this 9th lab we contemplate how to do encryption and decryption information employ the LFSRs. We moreover learnt numerous viewpoints with respect to the information security such as we cannot utilize same key continuously for the encryption as anybody can hack into it, same key must be utilized at both the conclusion of sending and accepting. For LFSRs we have to be specify the beginning point and the number of shifts within the key esteem. Considering all this security issues we actualized a secure information encryption calculation.

6. REFERENCE:

- Mno Moris, Digital Design. New Jersey: Prentice Hall. 2.
- Wakerley, Jon, Digital-Design Principles and Practices, 3rd Edition. New Jersey: Prentice Hall.

7. APPENDIX:



Data Encryption Using LFSRs

```
-----------DATA ENCRYPTION/DECREYPTION USING LFSRs--------
-- University: ILLINOIS INSTITUTE OF TECHNOLOGY, CHICAGO
-- Engineer: ANIRUDHA BEHERA
-- Design Name: LFSR-8bit
-- Module Name: LFSR8
-- Project Name: LAB_9
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY LFSR8 IS
  PORT (Clk, Rst: IN std_logic;
        output: OUT std_logic_vector (7 DOWNTO 0));
END LFSR8;

ARCHITECTURE Behavioral OF LFSR8 IS
  SIGNAL Currstate, Nextstate: std_logic_vector (7 DOWNTO 0);
  SIGNAL feedback: std_logic;
BEGIN

StateReg: PROCESS (Clk,Rst)
  BEGIN
    IF (Rst = '1') THEN
      Currstate <= (0 => '1', OTHERS =>'0');
    ELSIF (Clk = '1' AND Clk'EVENT) THEN
      Currstate <= Nextstate;
    END IF;
  END PROCESS;
 feedback <= Currstate(4) XOR Currstate(3) XOR Currstate(2) XOR Currstate(0);
 Nextstate <= feedback & Currstate(7 DOWNTO 1);
 output <= Currstate;
end Behavioral;
```

```
-----------DATA ENCRYPT/DECREYPTUSING LFSRs TESTBENCH-------------
-- University: ILLINOIS INSTITUTE OF TECHNOLOGY, CHICAGO
-- Engineer: ANIRUDHA BEHERA
-- Design Name: LFSR-8bit
-- Module Name: LFSR8_tb
-- Project Name: LAB_9
-------------------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY LFSR8_tb IS
END LFSR8_tb;

ARCHITECTURE Testbench OF LFSR8_tb IS
  COMPONENT LFSR8 IS
    PORT (Clk, Rst: IN std_logic;
          output: OUT std_logic_vector (7 DOWNTO 0));
  END COMPONENT;

  SIGNAL Clk_s, Rst_s: std_logic;
  SIGNAL output_s: std_logic_vector(7 DOWNTO 0);

BEGIN
  CompToTest: LFSR8 PORT MAP (Clk_s, Rst_s, output_s);
  Clk_proc: PROCESS
  BEGIN
    Clk_s <= '1';
    WAIT FOR 10 ns;
    Clk_s <= '0';
    WAIT FOR 10 ns;
  END PROCESS clk_proc;

  Vector_proc: PROCESS
  BEGIN
    Rst_s <= '1';
    WAIT FOR 5 NS;
    Rst_s <= '0';
    FOR index IN 0 To 4 LOOP
      WAIT UNTIL Clk_s='1' AND Clk_s'EVENT;
    END LOOP;
    WAIT FOR 5 NS;
    ASSERT output_s = X"88" REPORT "Failed output=88";
    WAIT;
  END PROCESS Vector_proc;

END Testbench;
```

THANK YOU