Digital Circuits and Systems Lab

Laboratory report submitted for the partial fulfillment
of the requirements for the degree of

Bachelor of Technology
in
Electronics and Communication Engineering

by

Mohit Akhouri - 19ucc023

Course Coordinator
Dr. Kusum Lata

Department of Electronics and Communication Engineering
The LNM Institute of Information Technology, Jaipur

August 2020

Contents

List of Figures

List of Tables

Chapter *1*

Experiment - 1

1.1 Name of the Experiment

Design, simulate and implement Half adder, Full adder using dataflow, behavioral and structural modeling in VHDL

1.2 Theory

1.2.1 Half Adder

Half Adder is a combinational arithmetic circuit that adds two single binary digits A and B. The inputs of the half adder are called **AUGEND** and **ADDEND** bits and the outputs are called **SUM** and **CARRY**. Simplest half adder design uses **XOR gate** for sum and **AND gate** for carry out.
The major disadvantage of half adder is that it can only add two bits at a time and if carry is present in the inputs, it does nothing to carry so, the binary addition process is not complete and hence the name half adder is given to this circuit. The truth table of half adder is :

A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 1.1 Half adder

1.2.2 Full Adder

Full Adder is a combinational circuit that adds three single binary digits A,B and C_{in} where C_{in} represents carry-in from the previous less significant stage. The full adder is usually a component in a cascade of adders , which add 8,16 and 32 bit binary numbers. The circuit produces a two bit output C_{out} and **SUM** .

A full adder can be implemented in many different ways such as with a custom transistor-level circuit or composed of other gates. One example implementation i s $SUM = A \oplus B \oplus C_{in}$ and $C_{out} = (A.B) + (C_{in}.(A \oplus B))$, i.e. full adder can be implemented with the use of **AND** , **XOR** and **OR** gates. A full adder can also be constructed from two half added modules by giving two inputs to first half adder and then giving the sum_{out} of the first half adder and C_{in} to the second half adder. The overall sum is the Sum_{out} of the second half adder and C_{out} is the OR of the first carry from the first half adder and second carry from the second half adder.
The truth table of full adder is :

A	B	C_{in}	SUM	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1.2 Full adder

1.3 Coding Techniques used

1. ***Half adder using Dataflow modeling :***

   In this type of modeling , we derived the boolean expression for SUM and CARRY and used the boolean expression to write our code. The boolean expressions are :
   SUM = A $\oplus$ B
   CARRY = A . B

2. ***Half adder using Behavioral modeling :***

   In this type of modeling , we analyse the inputs and outputs of the half adder and make different cases using **IF - ELSE** constructs of VHDL.

3. ***Full adder using Dataflow modeling :***

   In this type of modeling we analysed and derived boolean expressions for SUM and C_{out} and used them to write our code. The boolean expressions are :
   SUM = A $\oplus$ B $\oplus$ C_{in}
   C_{out} = A.B + C_{in}.(A $\oplus$ B)

4. ***Full adder using Behavioral modeling :***

   In this type of modeling , we analyse the inputs and outputs of the full adder and make different cases by using **IF- ELSIF -ELSE** constructs of VHDL.

5. ***Full adder using Structural modeling :***

   In this type of modeling , we model the full adder by using two half adders , in which two of the three inputs are given to first half adder (ha1) and the **sum1** and **carry1** are noted. Now the last input C_{in} and **sum1** is given to the second half adder (ha2) and **sum2** and **carry2** are noted. Now the overall sum and carry are :
   SUM = sum2
   C_{out} = carry1 *OR* carry2

1.4 Simulation and Results

1.4.1 Half Adder using Dataflow modeling

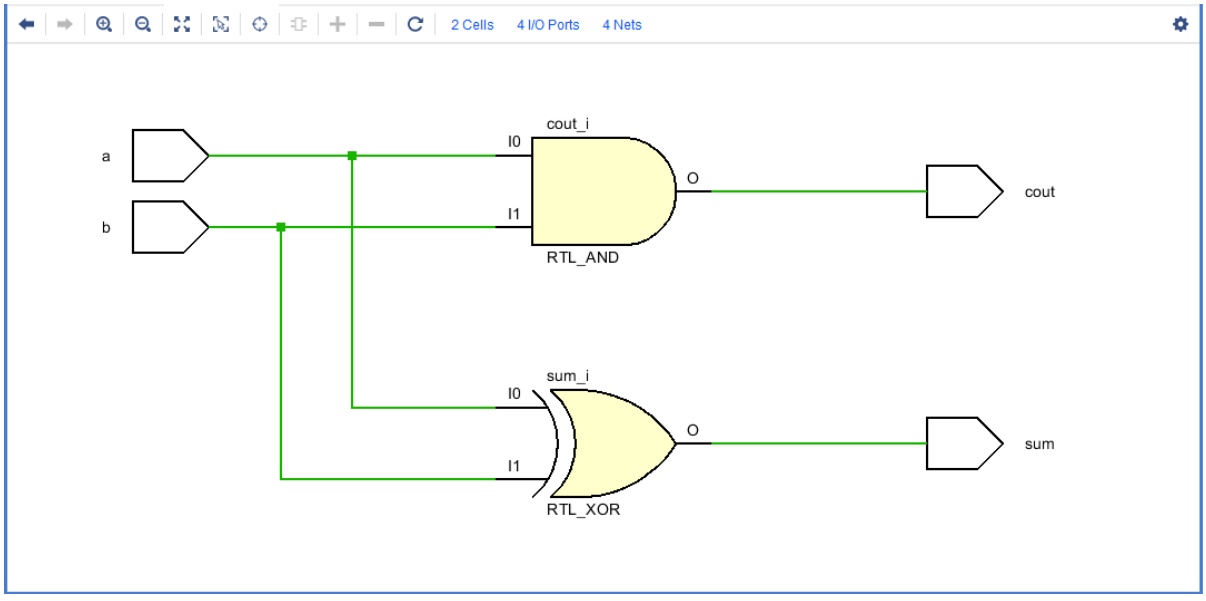

Figure 1.1 Schematic of the Half adder using Dataflow modeling

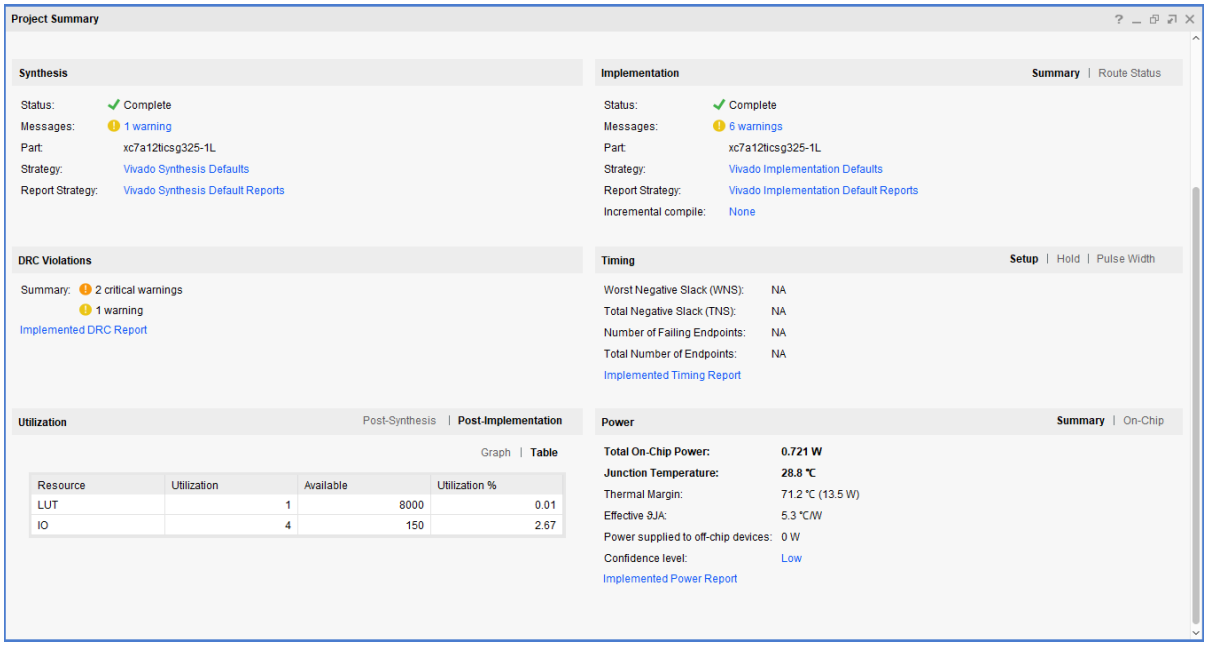

Figure 1.2 Project Summary of the Half adder using Dataflow modeling

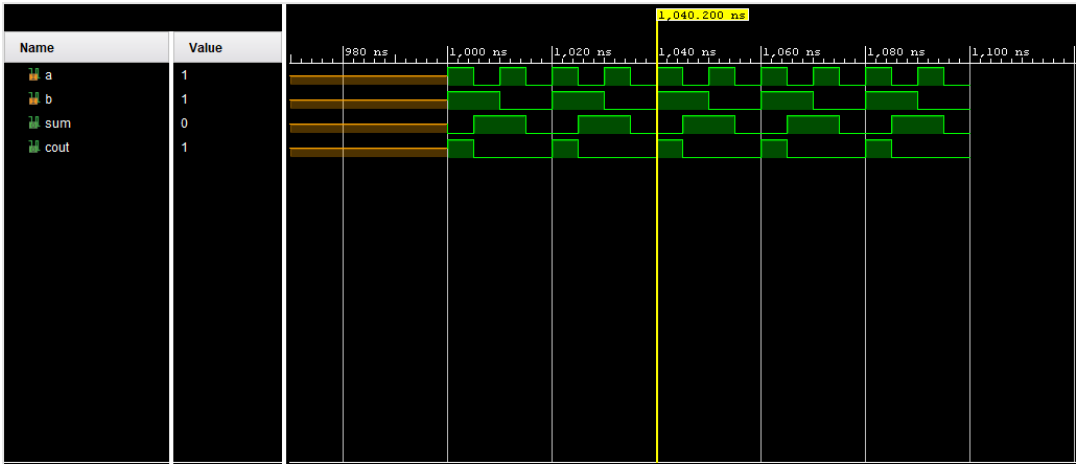

Figure 1.3 Simulation of the Half adder using Dataflow modeling

1.4.2 Half Adder using Behavioral modeling

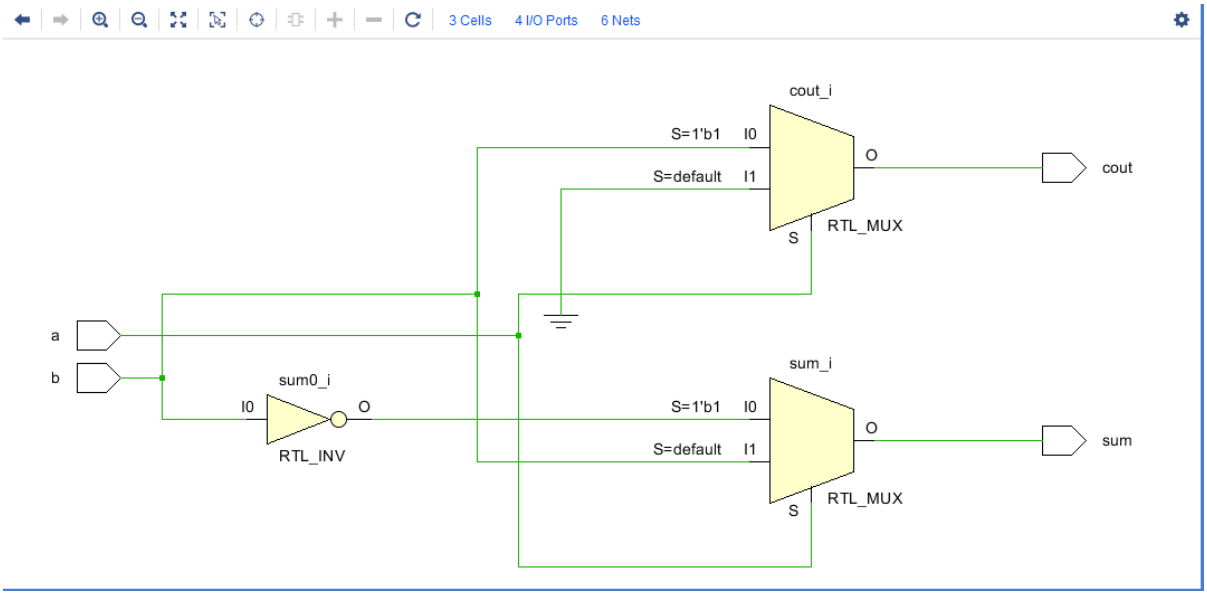

Figure 1.4 Schematic of the Half adder using Behavioral modeling

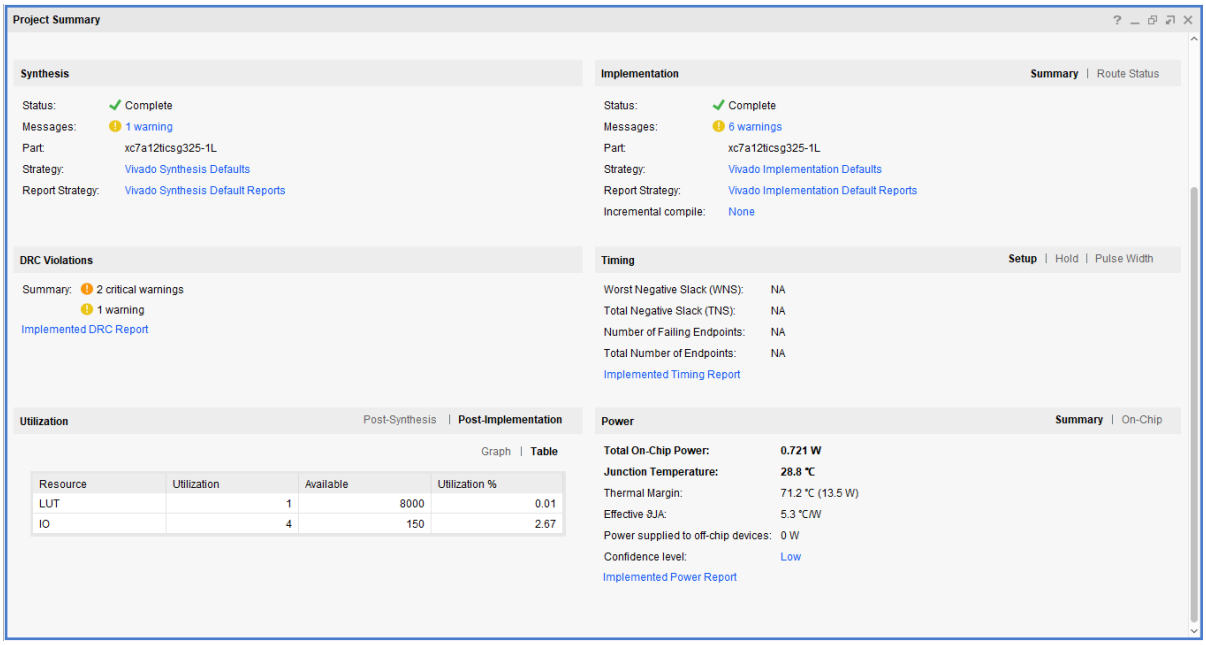

Figure 1.5 Project Summary of the Half adder using Behavioral modeling

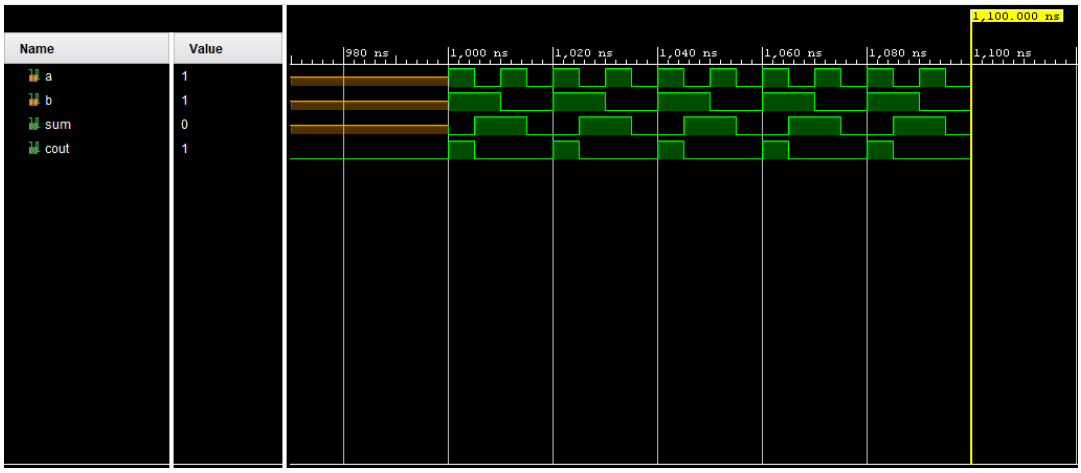

Figure 1.6 Simulation of the Half adder using Behavioral modeling

1.4.3 Full Adder using Dataflow modeling

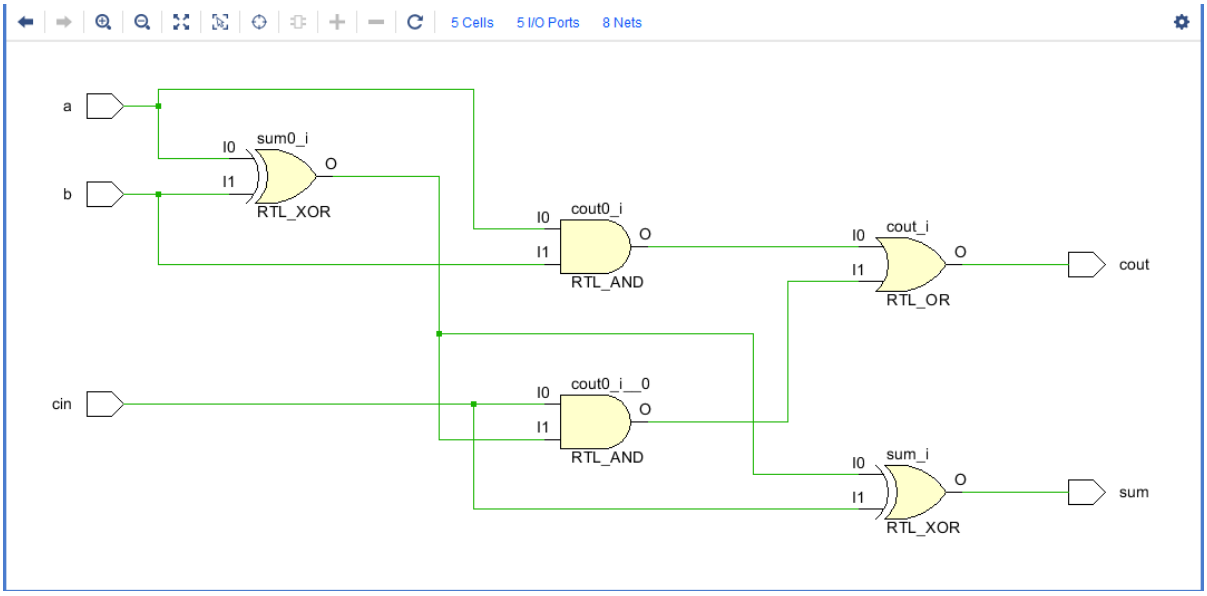

Figure 1.7 Schematic of the Full adder using Dataflow modeling

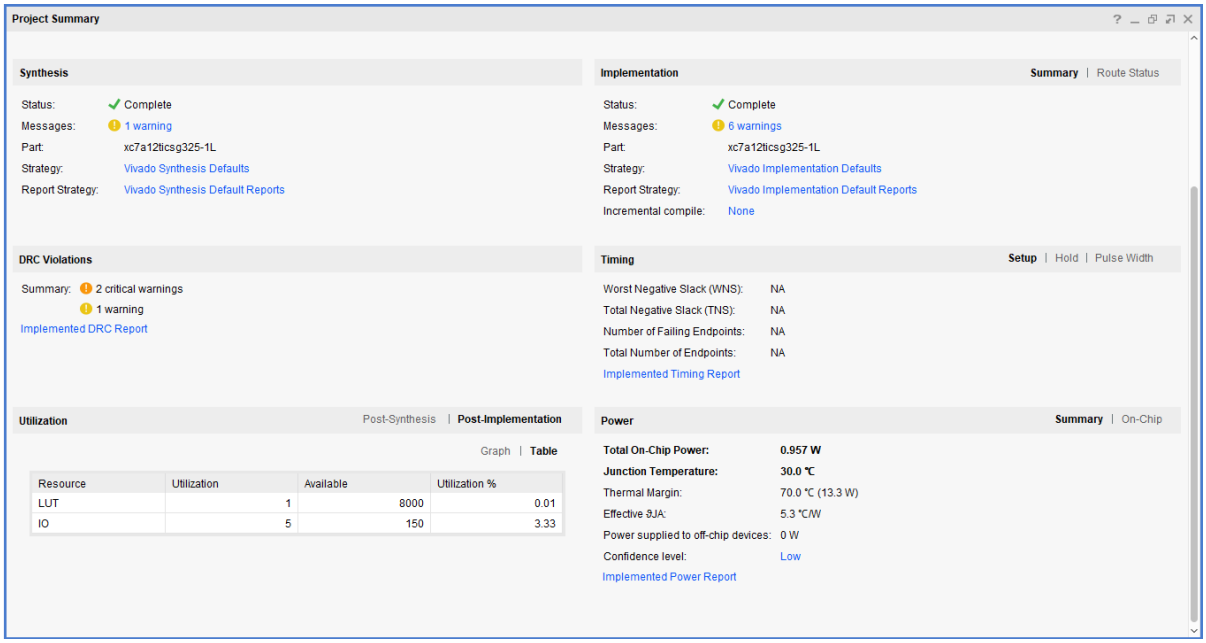

Figure 1.8 Project Summary of the Full adder using Dataflow modeling

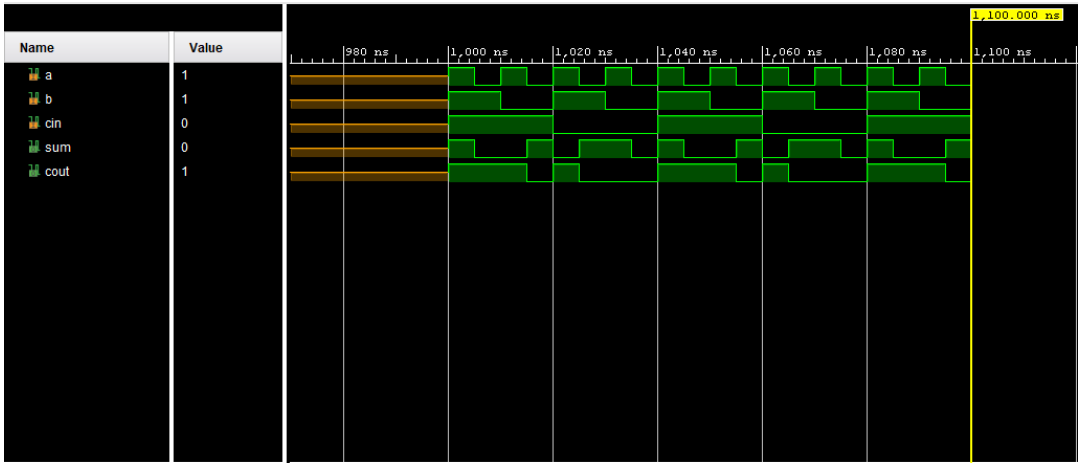

Figure 1.9 Simulation of the Full adder using Dataflow modeling

1.4.4 Full Adder using Behavioral modeling

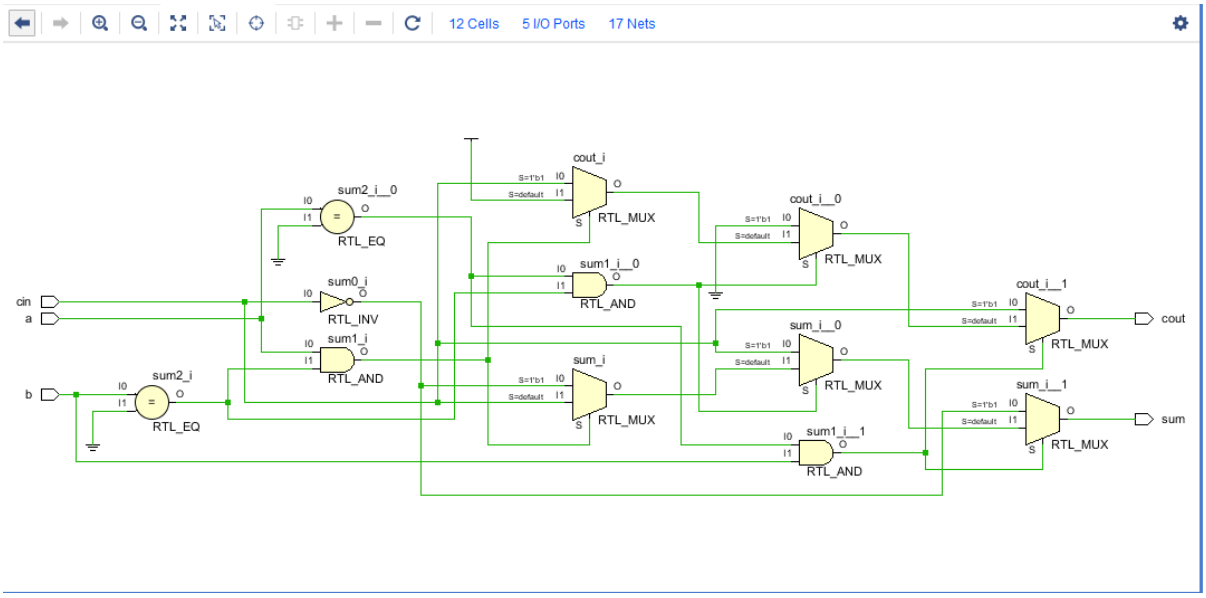

Figure 1.10 Schematic of the Full adder using Behavioral modeling

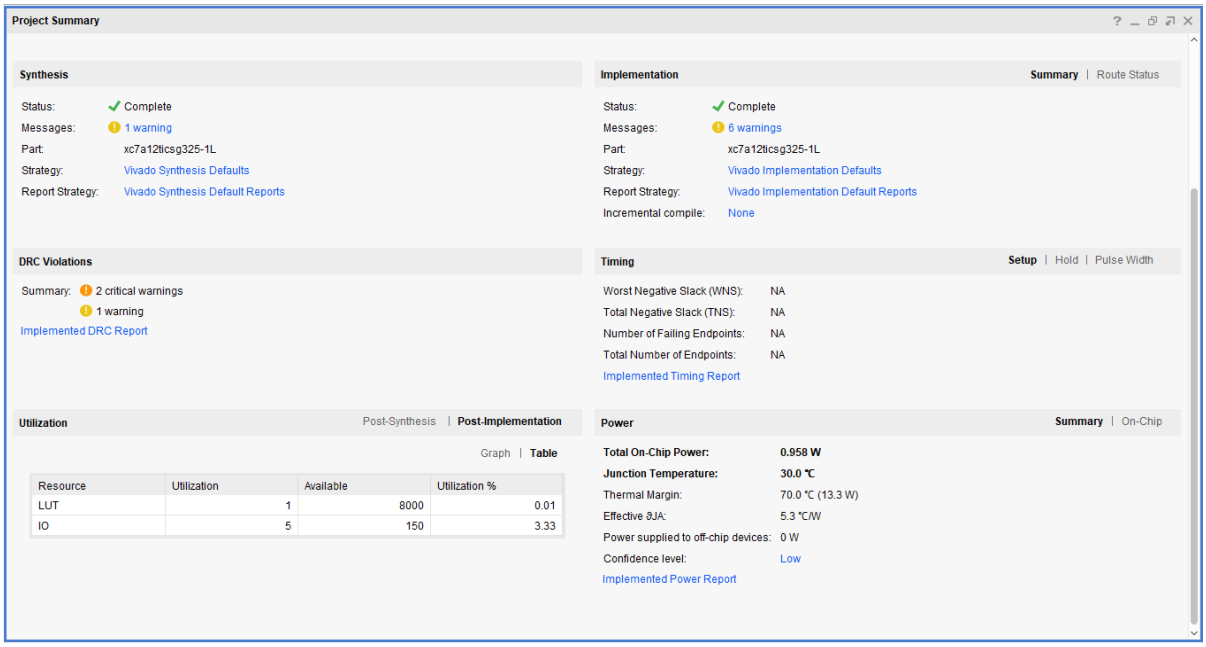

Figure 1.11 Project Summary of the Full adder using Behavioral modeling

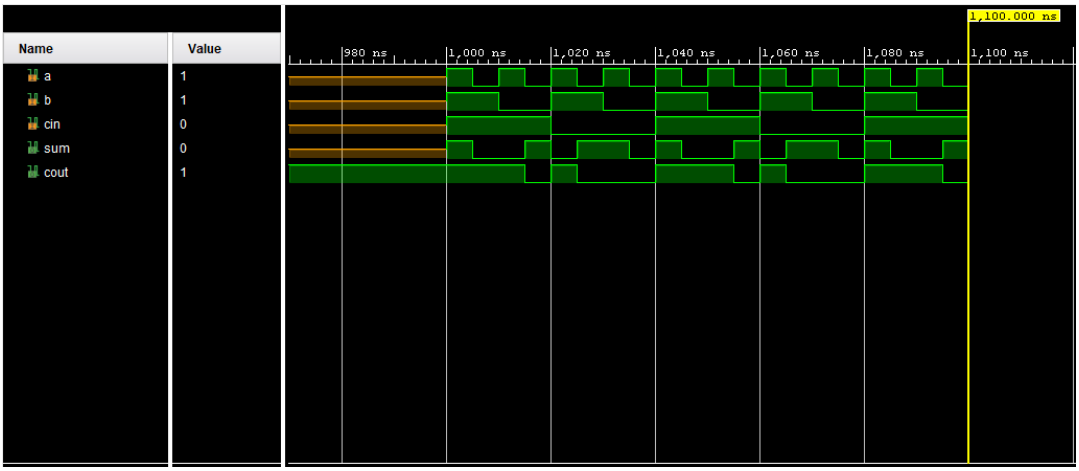

Figure 1.12 Simulation of the Full adder using Behavioral modeling

1.4.5 Full Adder using structural modeling using two half added modules

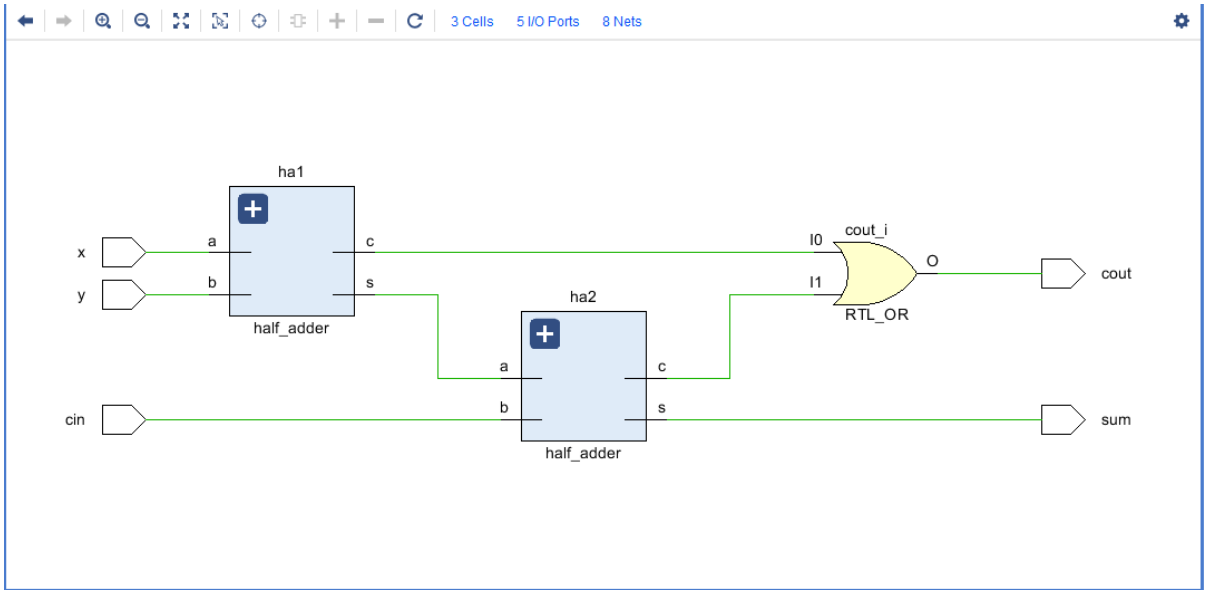

Figure 1.13 Schematic of the Full adder using Structural modeling

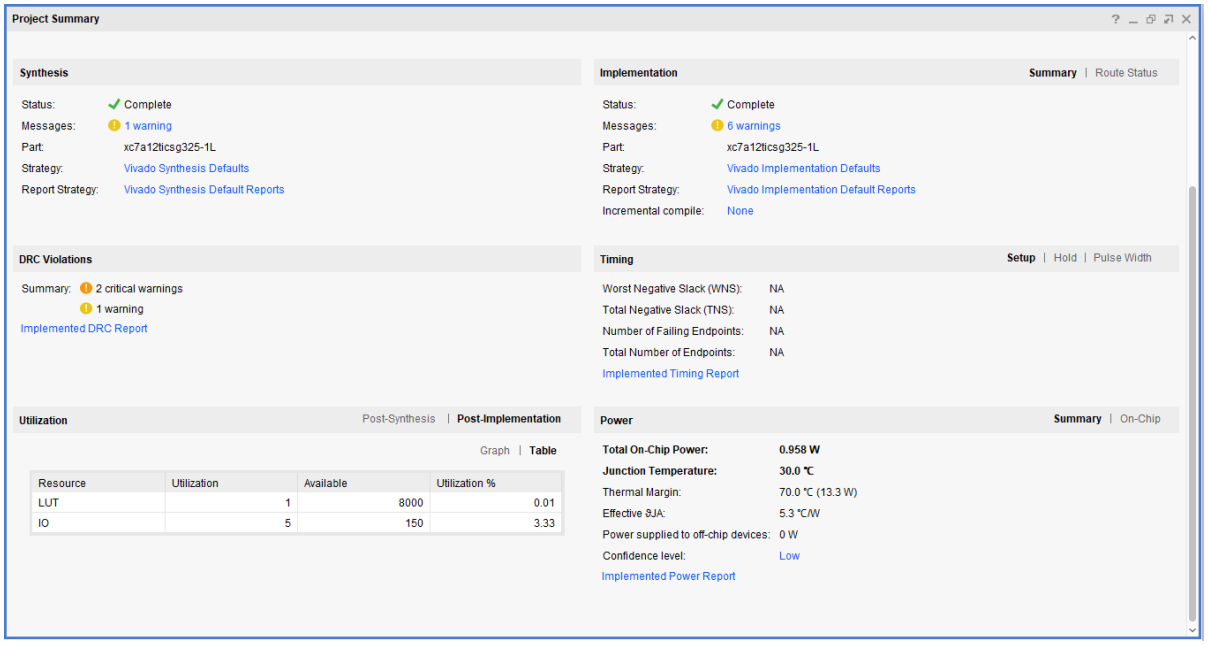

Figure 1.14 Project Summary of the Full adder using Structural modeling

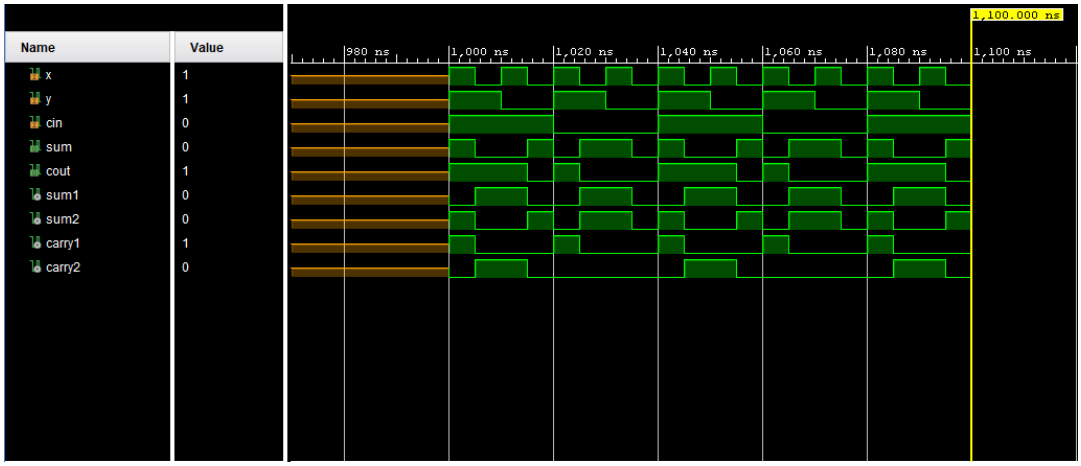

Figure 1.15 Simulation of the Full adder using Structural modeling

1.5 Summary

Tabular comparison of all the codes in terms of area and power usage.

Name of the Entity	No. of LUT used	Total On chip Power
Half Adder using Dataflow	1	0.721W
Half Adder using Behavioural	1	0.721W
Full Adder using Dataflow	1	0.957W
Full Adder using Behavioral	1	0.958W
Full Adder using Structural	1	0.958W

Table 1.3 comparision of Area and power requirements for different kinds of adders.

Chapter *2*

Experiment - 2

2.1 Name of the Experiment

Realize the function, mentioned below in at least four different physical ways :
F(x) = x0 + x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9

2.2 Theory

2.2.1 OR Gate

In this experiment , we have used different forms of **OR gate** like 2 input , 3 input and 4 input . OR gate is a digital logic gate that implements logical disjunction . A high output (**1**) results if one or more of the inputs given are 1 and a low output (**0**) results if all the inputs given are 0 . OR gates are available in **TTL** and **CMOS ICs logic families**. The TTL device is **7432** and the CMOS IC is the **4071**.
The truth table for 2 input OR gate is :

A	B	Result
0	0	0
0	1	1
1	0	1
1	1	1

Table 2.1 2 input OR gate

2.3 Coding Techniques used

1. ***Implement the function F(x) using 8 stages of 2 input OR gates***

   In this part, we have used 2 input OR gates in 8 different stages . In first stage , we have used 2 gates and from the next stage onwards , we have used 1 gate with 1 input as the signal of the previous stage and one input given to the function.

2. ***Implement the function F(x) using three 4 different input OR gates***

   In this part, we have 4 input OR gates . In the first and second OR gate , we have given inputs from x0 to x7 and in the last OR gate , two signals **s1** and **s2** of the previous OR gates and two inputs (x8 and x9), thus getting the function F(x).

3. ***Implement the function F(x) as a three stage network***

   In this part , we have used 4 **3-input OR gates** and 1 **2-input OR gates** to derive the function F(x). The 3-input OR gates are used in stage 1 and stage 2 and 2-input OR gate is used in stage 3, thus making a 3-stage network.

4. ***Implement F(x) as a four stage network***

   In this part , we have used 4 **3-input OR gates** and 1 **2-input OR gates** to derive the function F(x). The 3-input OR gates are used in stage 1,stage 2 and stage 3 and 2-input OR gate is used in stage 4, thus making a 4-stage network.

2.4 Simulation and Results

2.4.1 Implement the function F(x) using 8 stages of 2 input OR gates

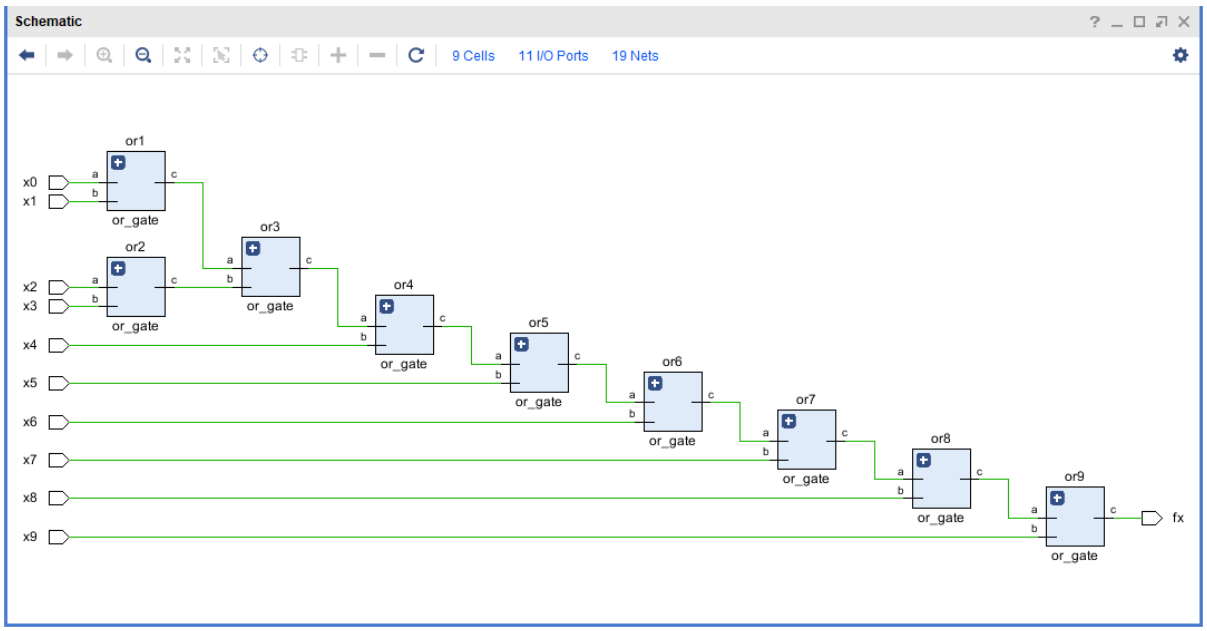

Figure 2.1 Schematic of the F(x) using 8 stages of 2 input OR gates

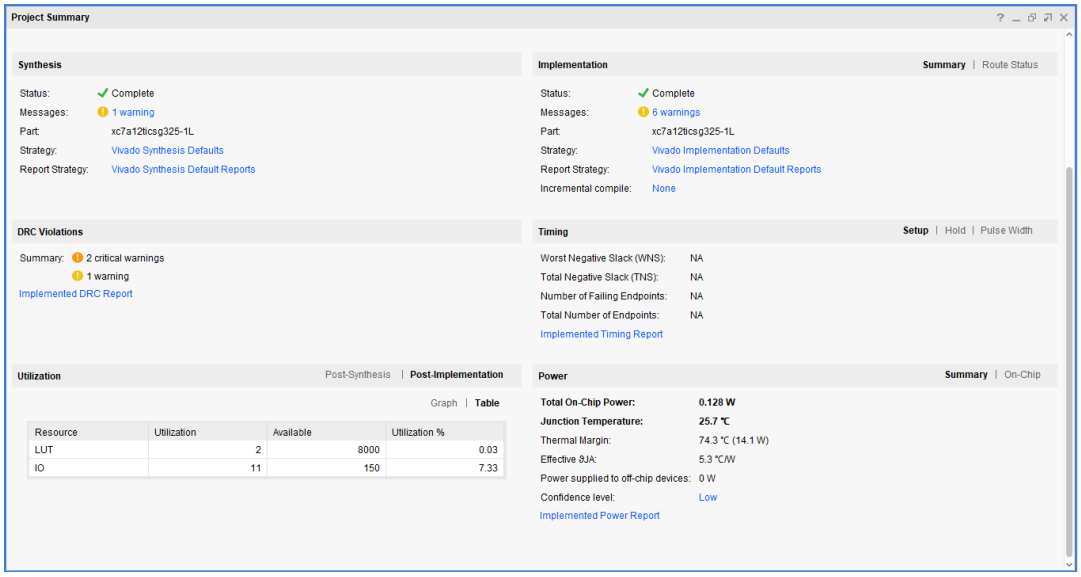

Figure 2.2 Project Summary of the F(x) using 8 stages of 2 input OR gates

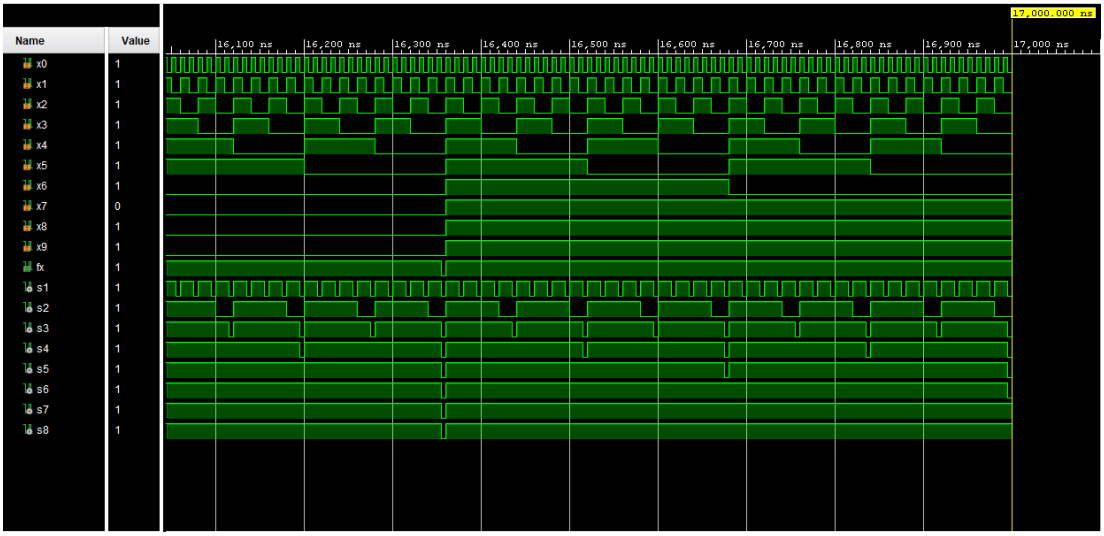

Figure 2.3 Simulation of the F(x) using 8 stages of 2 input OR gates

2.4.2 Implement the function F(x) using three 4 different input OR gates

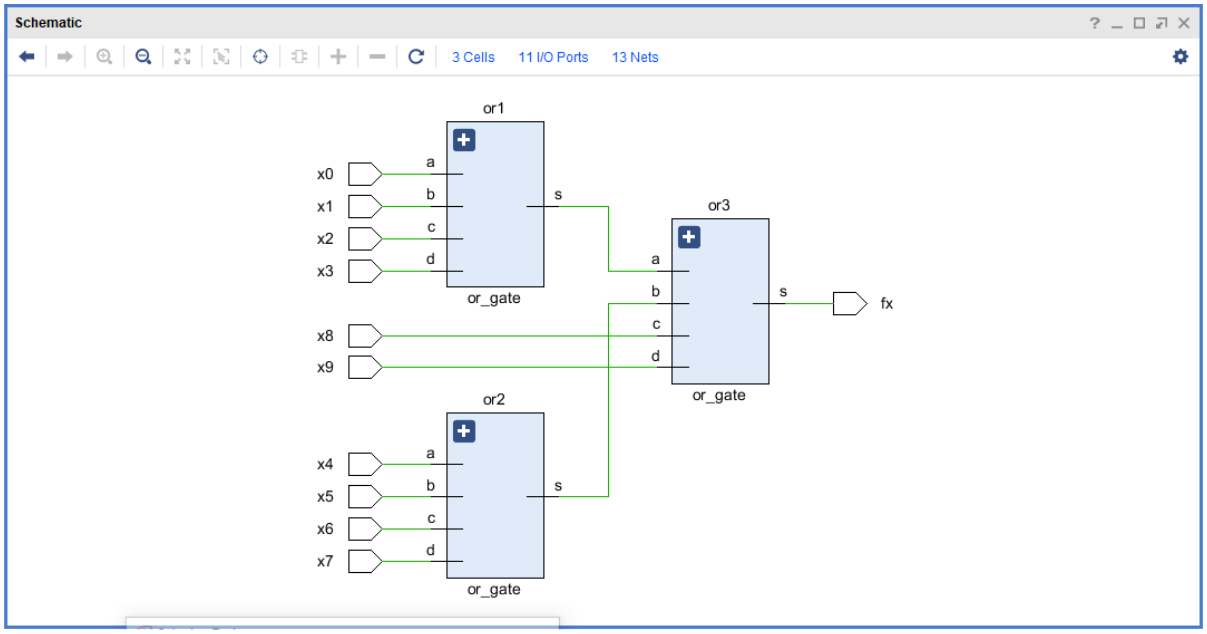

Figure 2.4 Schematic of the F(x) using three 4 different input OR gates

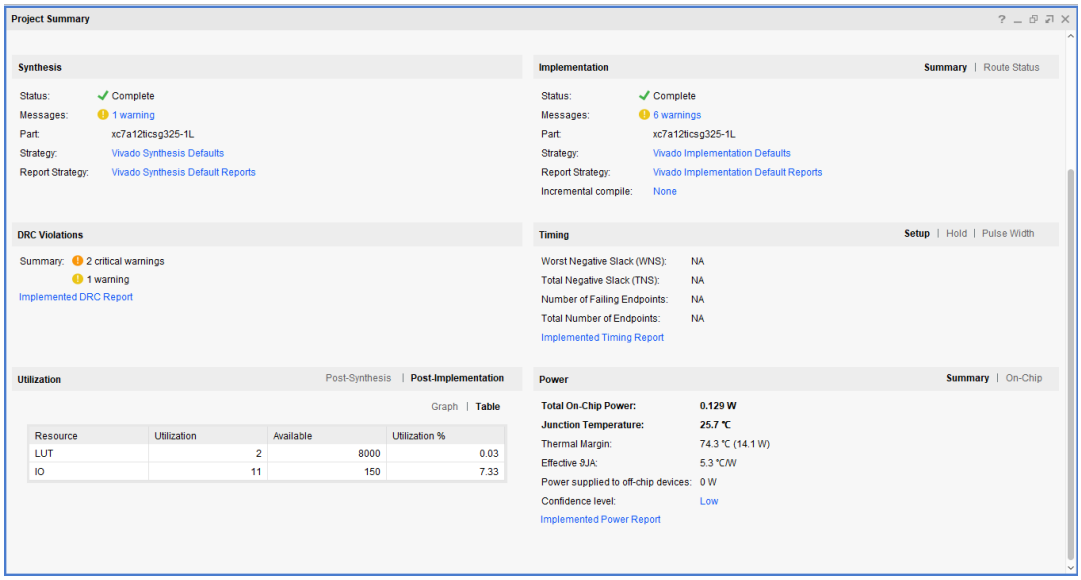

Figure 2.5 Project Summary of the F(x) using three 4 different input OR gates

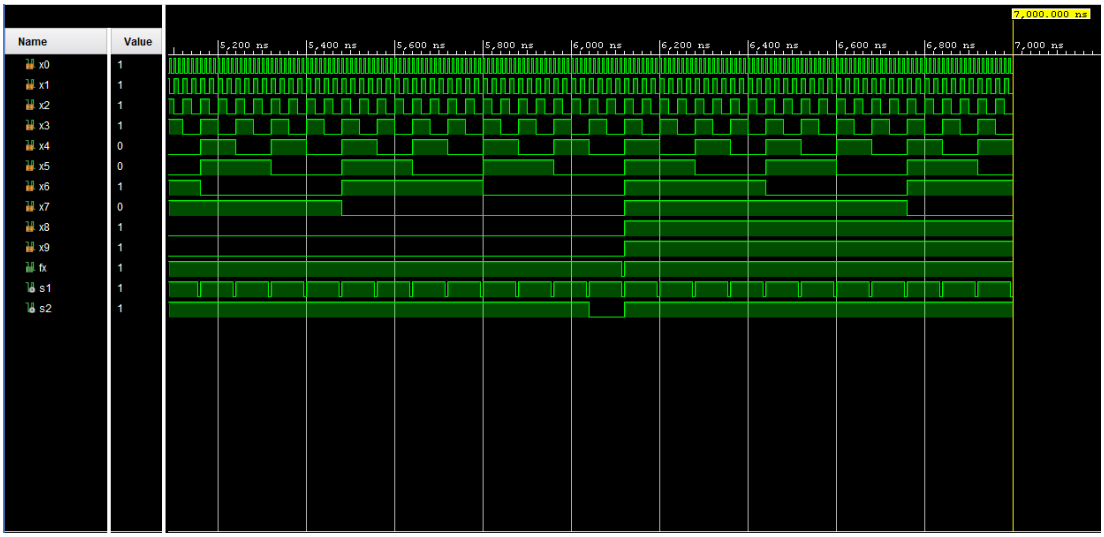

Figure 2.6 Simulation of the F(x) using three 4 different input OR gates

2.4.3 Implement the function F(x) as a three stage network

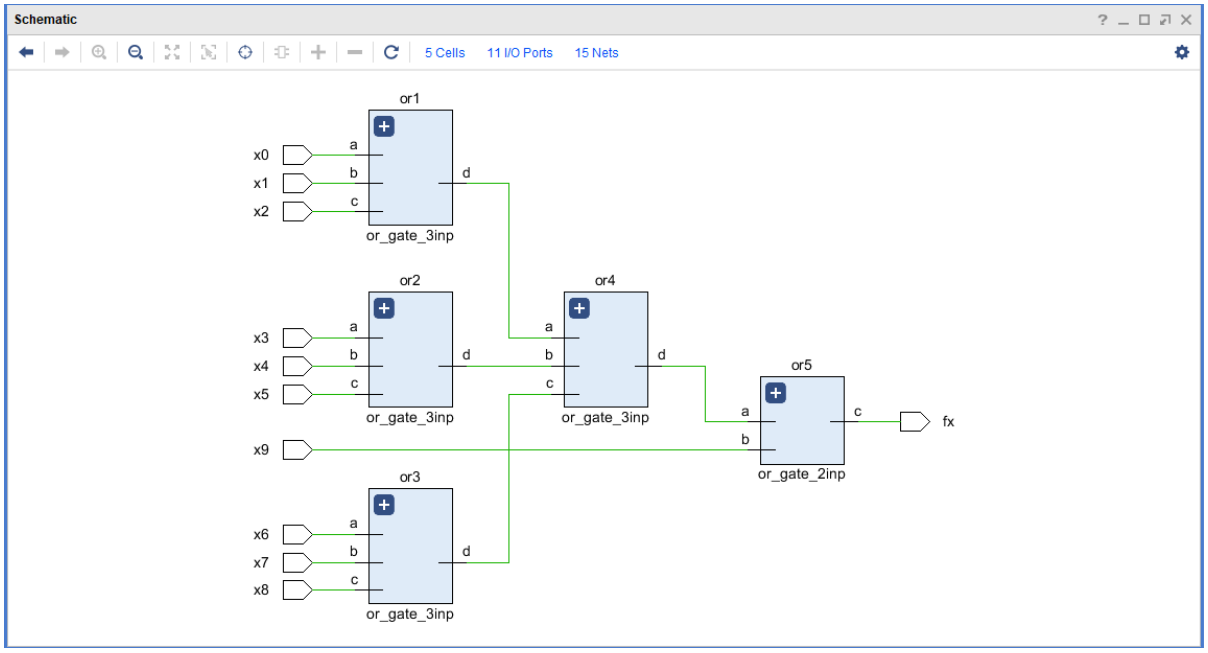

Figure 2.7 Schematic of the F(x) as a three stage network

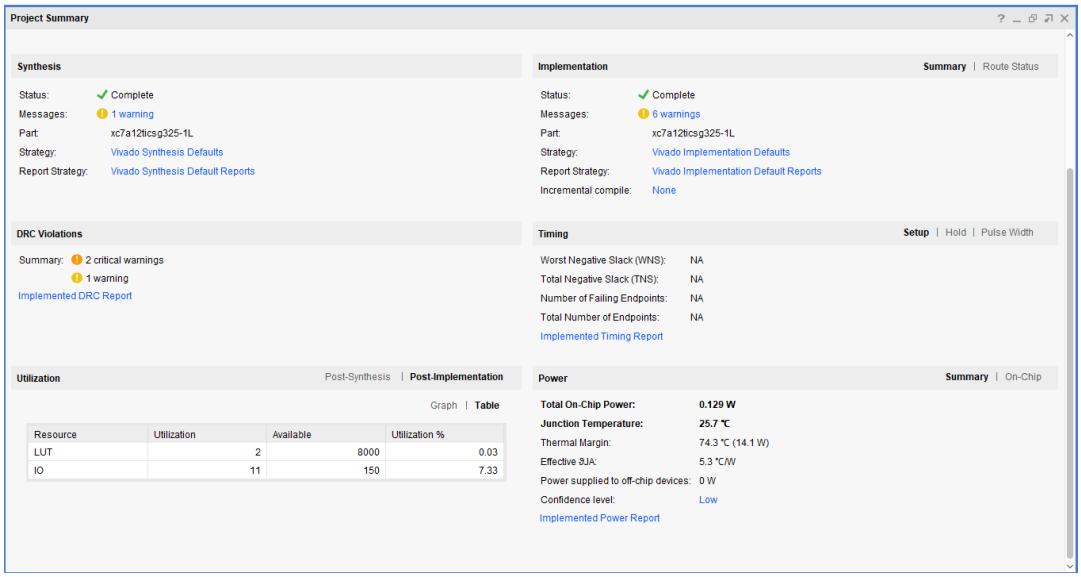

Figure 2.8 Project Summary of the F(x) as a three stage network

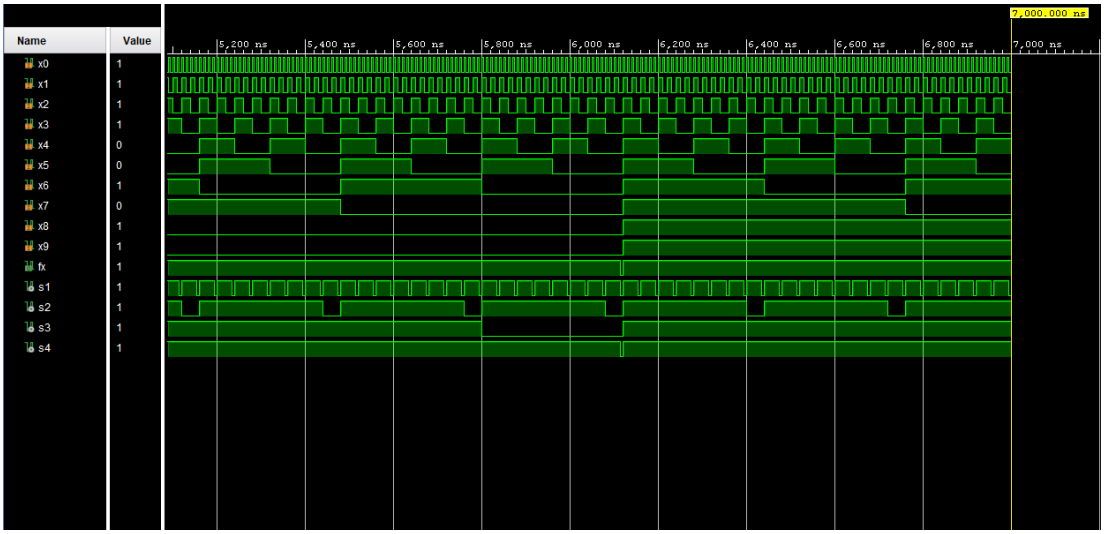

Figure 2.9 Simulation of the F(x) as a three stage network

2.4.4 Implement the function F(x) as a four stage network

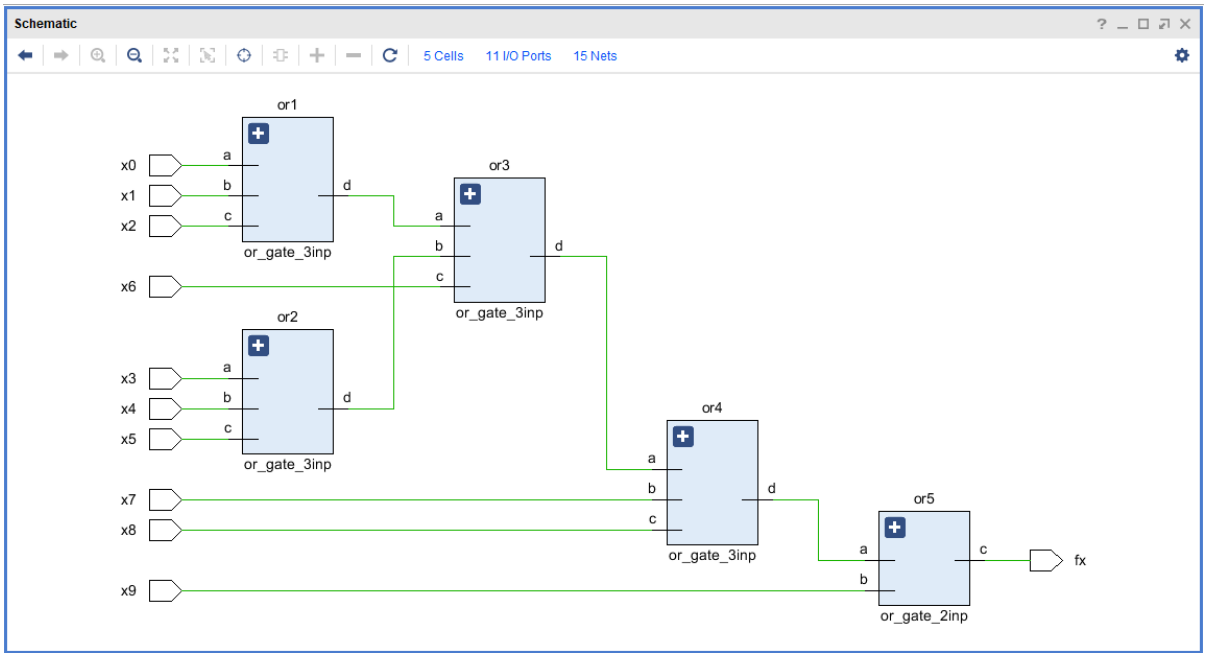

Figure 2.10 Schematic of the F(x) as a four stage network

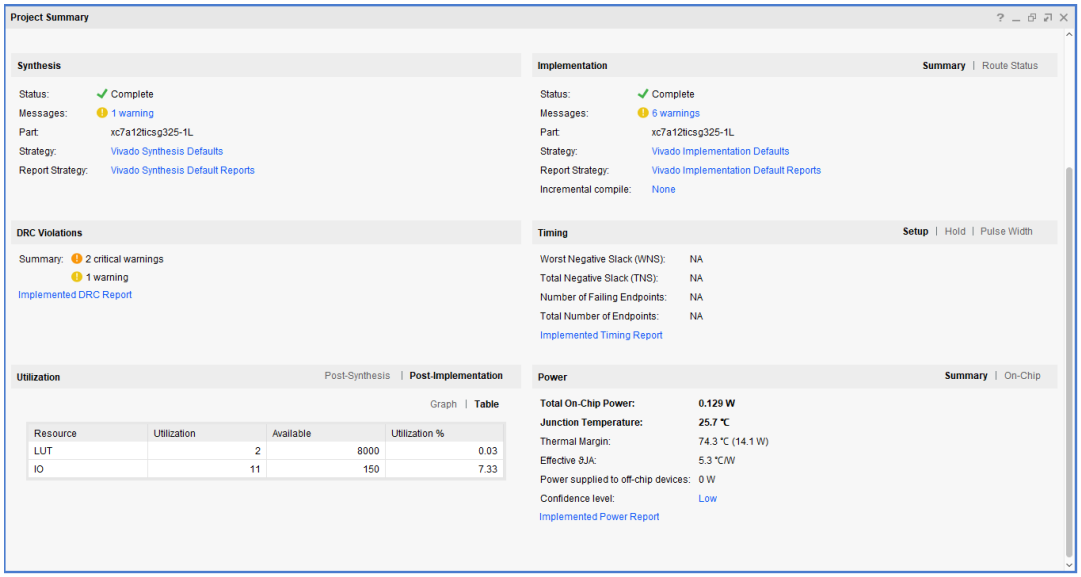

Figure 2.11 Project Summary of the F(x) as a four stage network

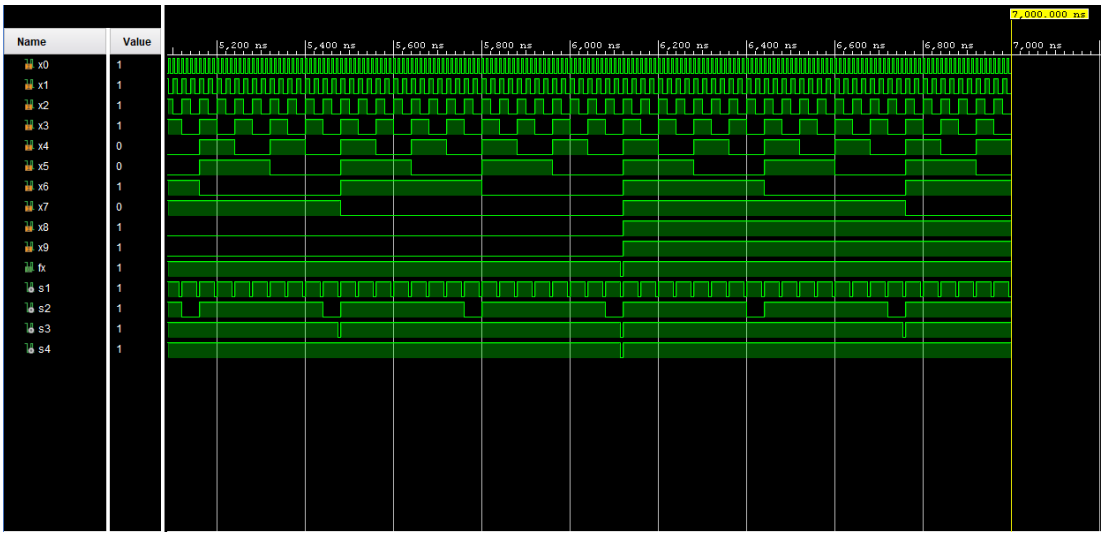

Figure 2.12 Simulation of the F(x) as a four stage network

2.5 Summary

Tabular comparison of all the codes in terms of area and power usage.

Name of the Entity	No. of LUT used	Total On chip Power
Implement the function F(x) using 8 stages of 2 input OR gates	2	0.129W
Implement F(x) using three 4 different input OR gates	2	0.129W
Implement F(x) as a three stage network	2	0.129W
Implement F(x) as a four stage network	2	0.129W

Table 2.2 comparision of Area and power requirements for different kinds of implementation of F(x).

Chapter *3*

Experiment - 3

3.1 Name of the Experiment

Implementation of 2x1, 4x1 and 8x1 multiplexers using dataflow, behavioral and structural modeling in VHDL

3.2 Theory

3.2.1 Multiplexer

In electronics, a multiplexer also called as **MUX** or as **data selector** is a device that selects between several analog and digital signals and forwards it to a single output line. A multiplexer with 2^n inputs has **n** select lines.
Multiplexers are mainly used to increase the amount of data that can be sent over the network within a certain amount of time and bandwidth. They can also be used to implement **Boolean functions** of multiple variables.

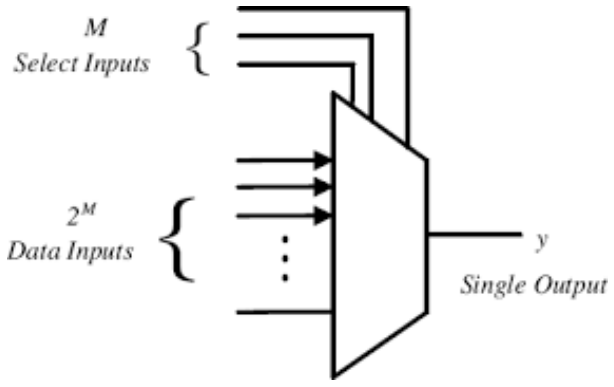

Figure 3.1 Multiplexer Symbol

3.2.2 2x1 Multiplexer :

In 2x1 multiplexer , there are 2 input signals and 1 select signal. Truth table for 2x1 multiplexer is :

S_0	Output
0	I_0
1	I_1

Table 3.1 2x1 Multiplexer

3.2.3 4x1 Multiplexer :

In 4x1 multiplexer, there are 4 input signals and 2 select signals. Truth table for 4x1 multiplexer is :

S_1	S_0	Output
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Table 3.2 4x1 Multiplexer

3.2.4 8x1 Multiplexer :

In 8x1 multiplexer , there are 8 input signals and 3 select signals. Truth table for 8x1 multiplexer is :

S_2	S_1	S_0	Output
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

Table 3.3 8x1 Multiplexer

3.3 Coding Techniques used

1. ***Implement a 2x1 multiplexer using dataflow modeling***

   In this modeling of 2x1 MUX , we derive the boolean expression for the output (**y**) and used it to write the code. The Boolean expression for output (**y**) is :

   $$y = S_0^{'}.I_0 + S_0.I_1$$

   where I_0 and I_1 are input signals and S_0 is the select signal.

2. ***Implement a 2x1 multiplexer using behavioral modeling using case statement***

   In this modeling of 2x1 MUX , we used the **Case** construct of VHDL . We give the condition that when S_0 is 0 then y=I_0 and when S_0 is 1 then y=I_1 . For other cases of S_0 , we gave y the value **don't care** i.e. (hyphen symbol).

3. ***Implement a 4x1 multiplexer using dataflow modeling***

   In this modeling of 4x1 MUX, we derived the boolean expression for the output (**y**) and used it to write our code. The Boolean expression for output (**y**) is :

   $$y = S_1^{'}.S_0^{'}.I_0 + S_1^{'}.S_0.I_1 + S_1.S_0^{'}.I_2 + S_1.S_0.I_3$$

   where I_0 , I_1 , I_2 and I_3 are input signals and S_0 and S_1 are select signals.

4. ***Implement a 4x1 multiplexer using structural modeling and using only 2x1 MUX***

   In this modeling of 4x1 MUX, we use **3** 2x1 multiplexers as structural units and correspondingly write the code. First we write the code of 2x1 MUX, then call it in the main program as a structural unit to implement 4x1 MUX. We give I_0 and I_1 to first MUX , I_2 and I_3 to second MUX and their outputs to third MUX. Select signal S_0 is given to first and second MUX and Select signal S_1 is given to last MUX.

5. ***Implement an 8x1 MUX using structural modeling and using 4x1 and 2x1 MUX***

   In this modeling of 8x1 MUX, we use **2** 4x1 multiplexers and **1** 2x1 multiplexer. First four signals (I_0 to I_3) are given to first 4x1 MUX and next four signals (I_4 to I_7) are given to second 4x1 MUX and the outputs of these MUX are given to the last 2x1 MUX. Select signals (S_0 and S_1) are given to the two 4x1 MUX and Select signal (S_2) is given to the last 2x1 MUX, thus getting a 8x1 MUX.

3.4 Simulation and Results

3.4.1 Implement a 2x1 multiplexer using dataflow modeling

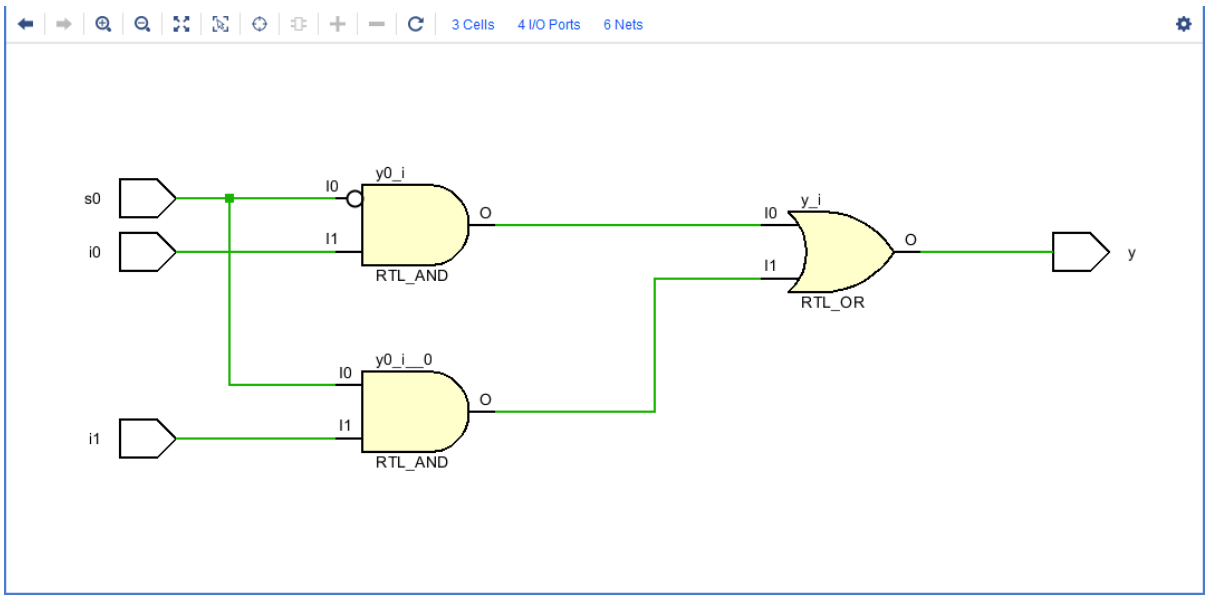

Figure 3.2 Schematic of the 2x1 multiplexer using dataflow modeling

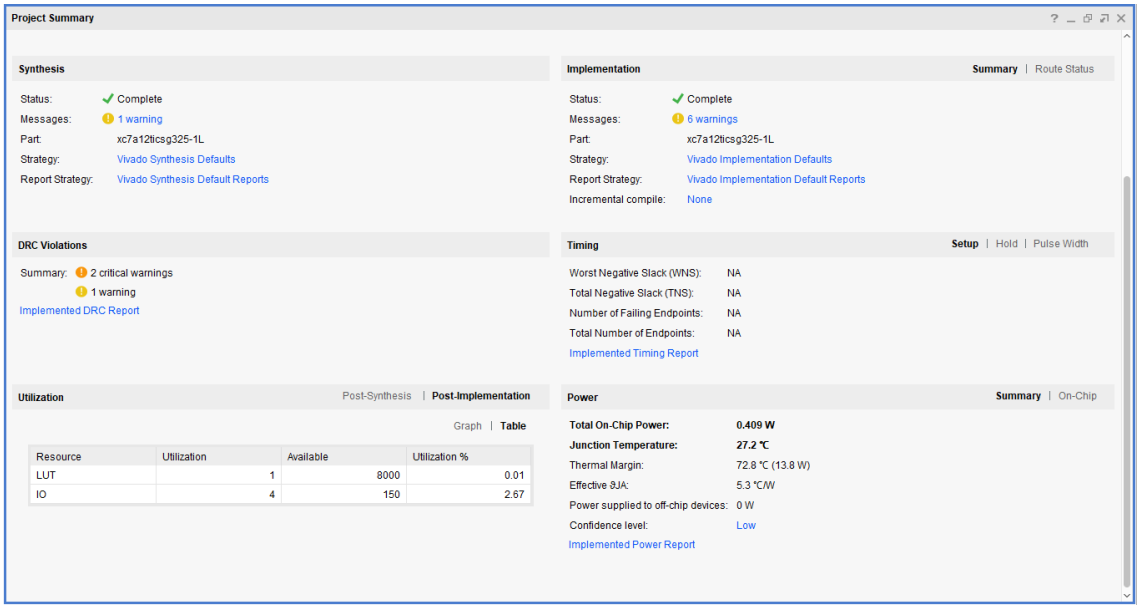

Figure 3.3 Project Summary of the 2x1 multiplexer using dataflow modeling

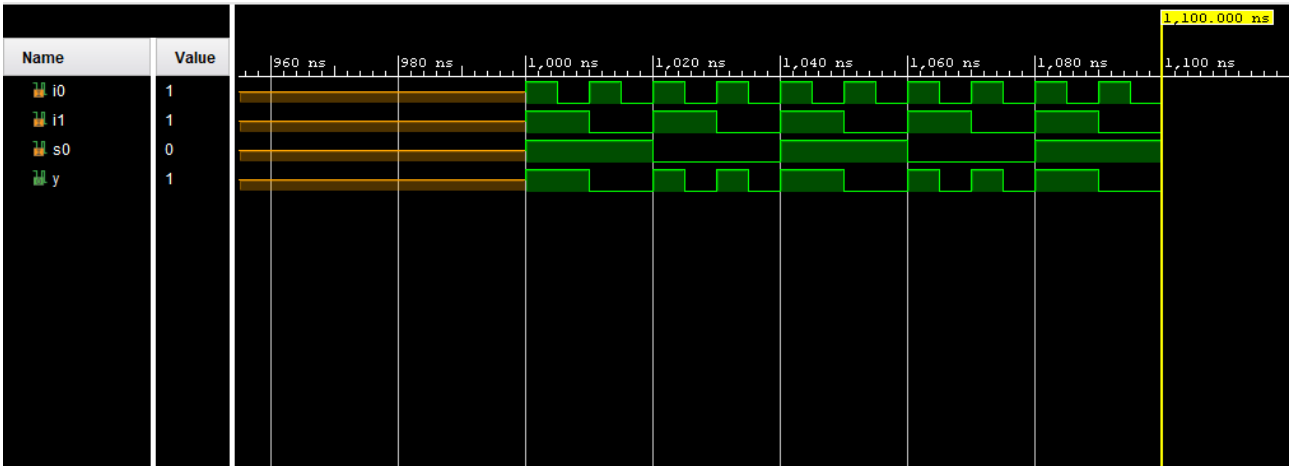

Figure 3.4 Simulation of the 2x1 multiplexer using dataflow modeling

3.4.2 Implement a 2x1 multiplexer using behavioral modeling using case statement

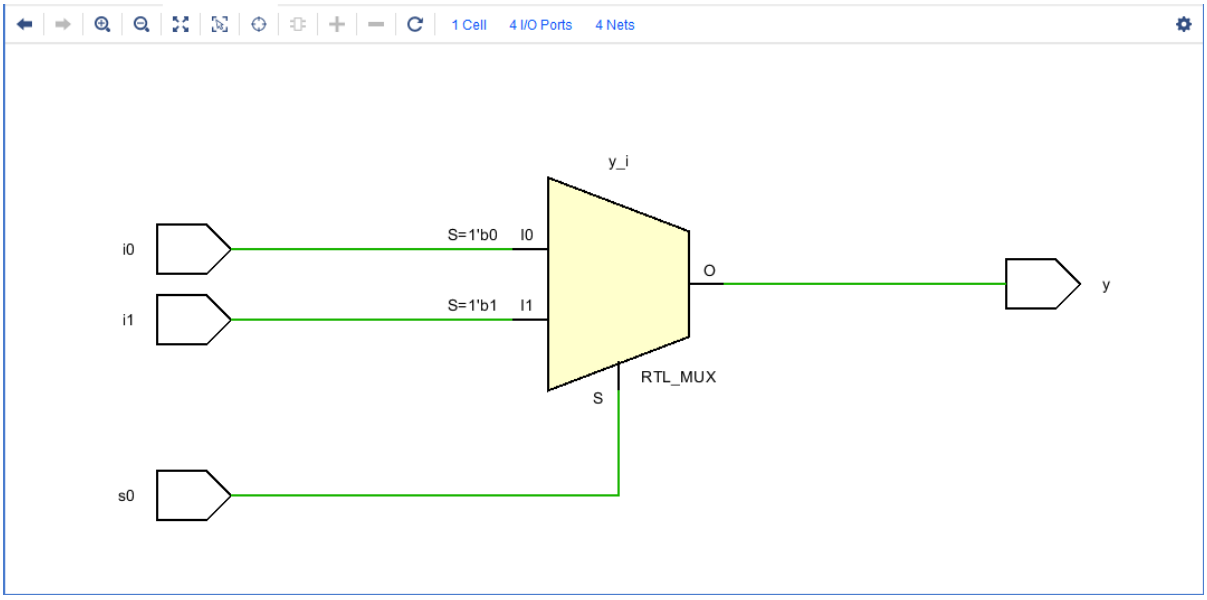

Figure 3.5 Schematic of the 2x1 multiplexer using behavioral modeling using case statement

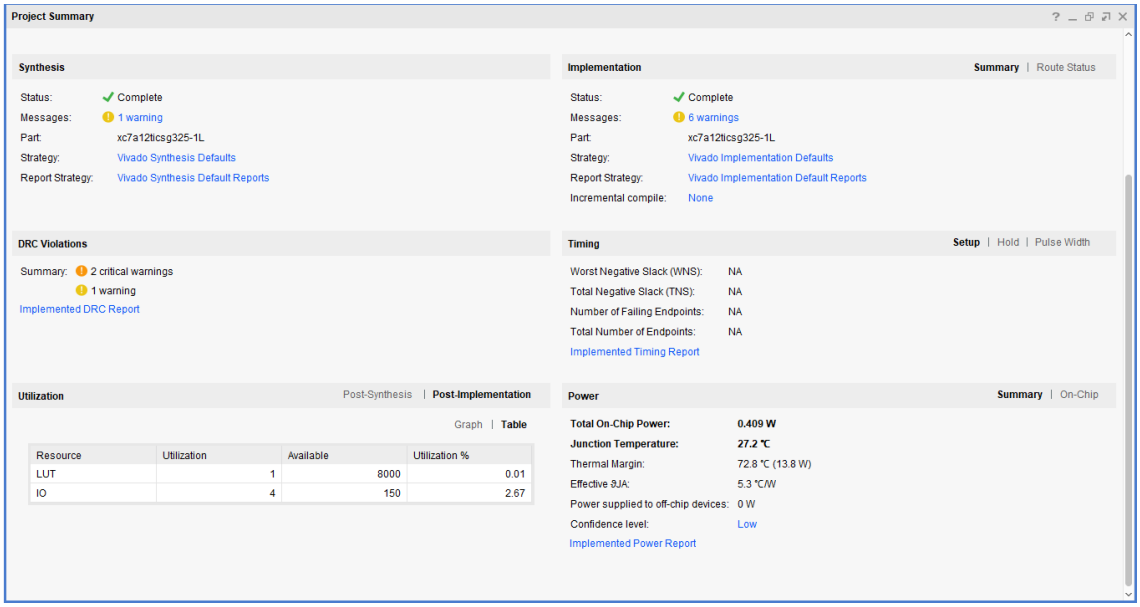

Figure 3.6 Project Summary of the 2x1 multiplexer using behavioral modeling using case statement

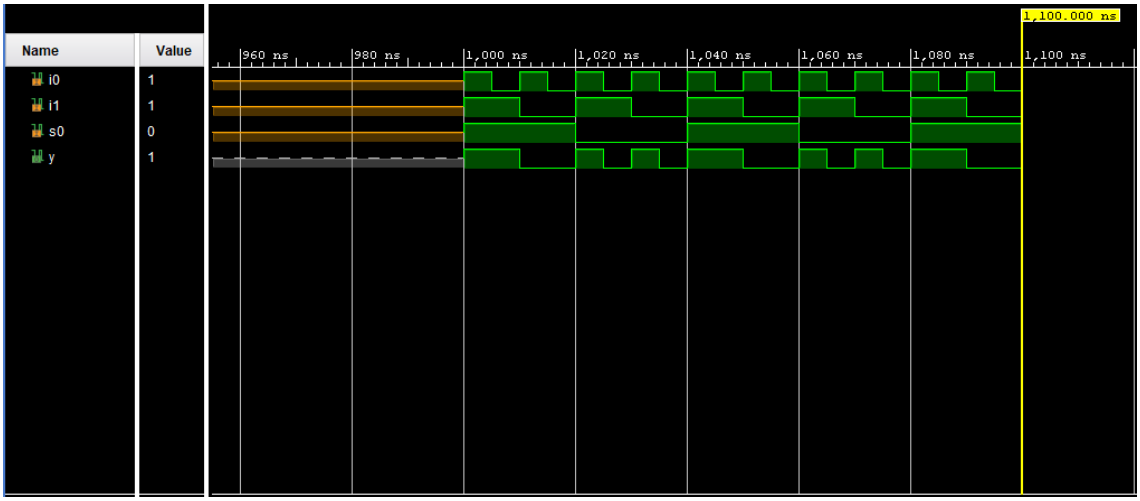

Figure 3.7 Simulation of the 2x1 mutliplexer using behavioral modeling using case statement

3.4.3 Implement a 4x1 multiplexer using dataflow modeling

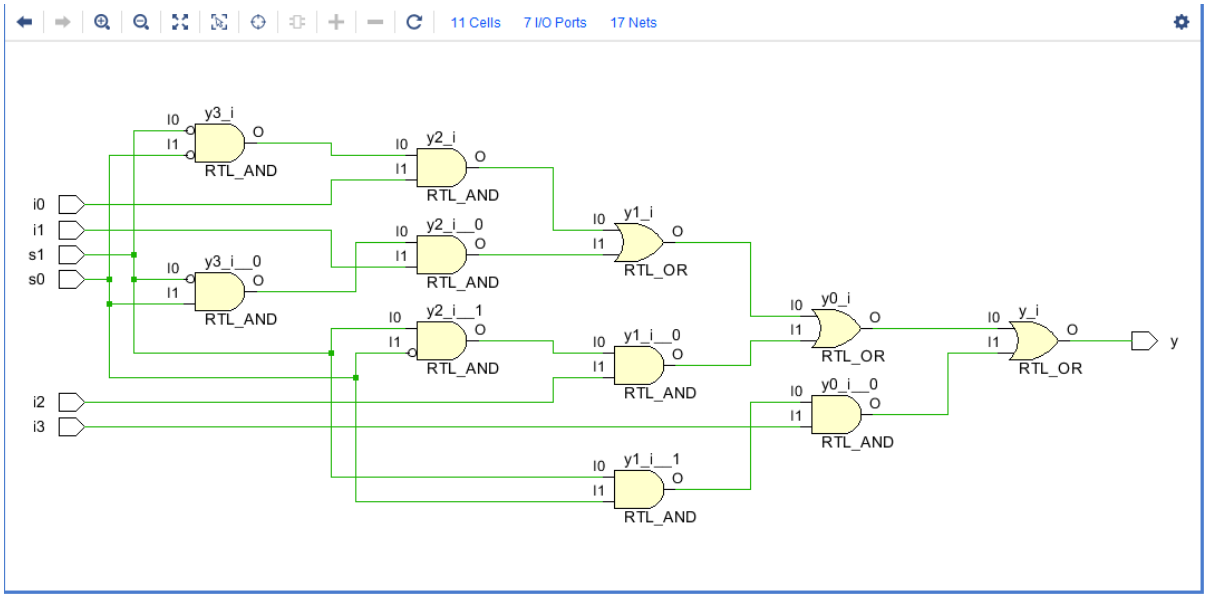

Figure 3.8 Schematic of the 4x1 multiplexer using dataflow modeling

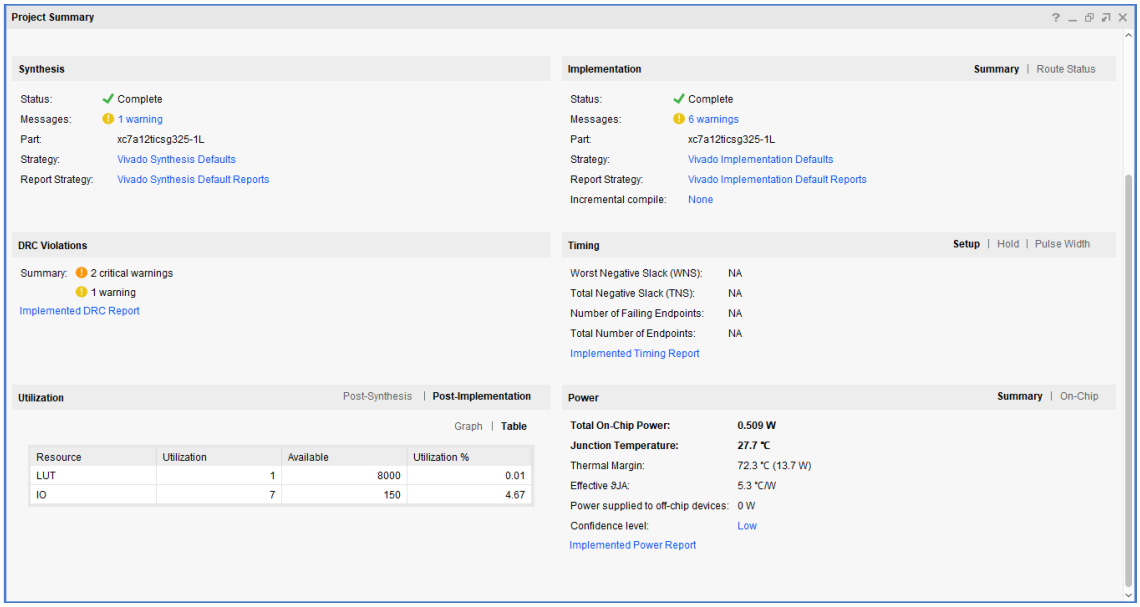

Figure 3.9 Project Summary of the 4x1 multiplexer using dataflow modeling

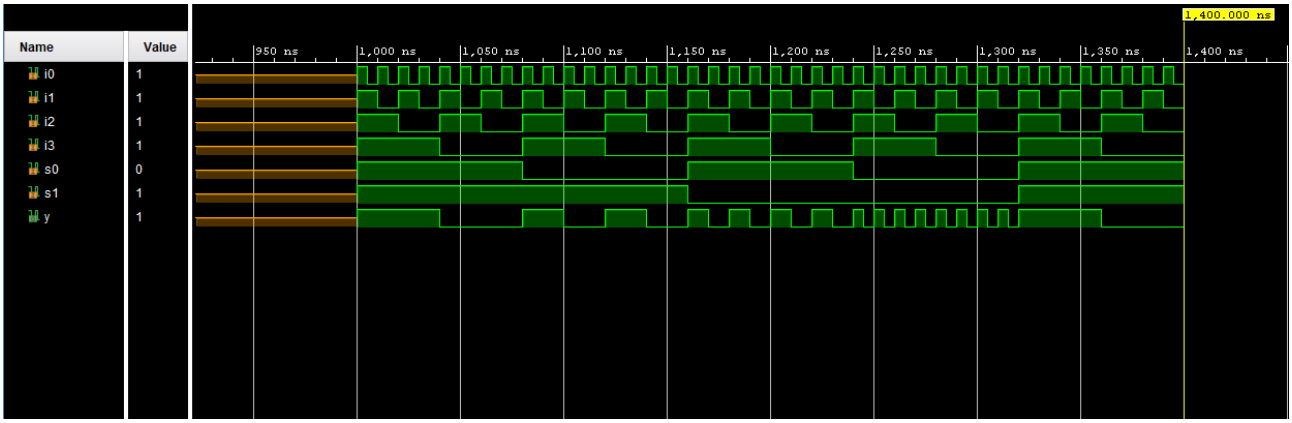

Figure 3.10 Simulation of the 4x1 multiplexer using dataflow modeling

3.4.4 Implement a 4x1 multiplexer using structural modeling and using only 2x1 MUX

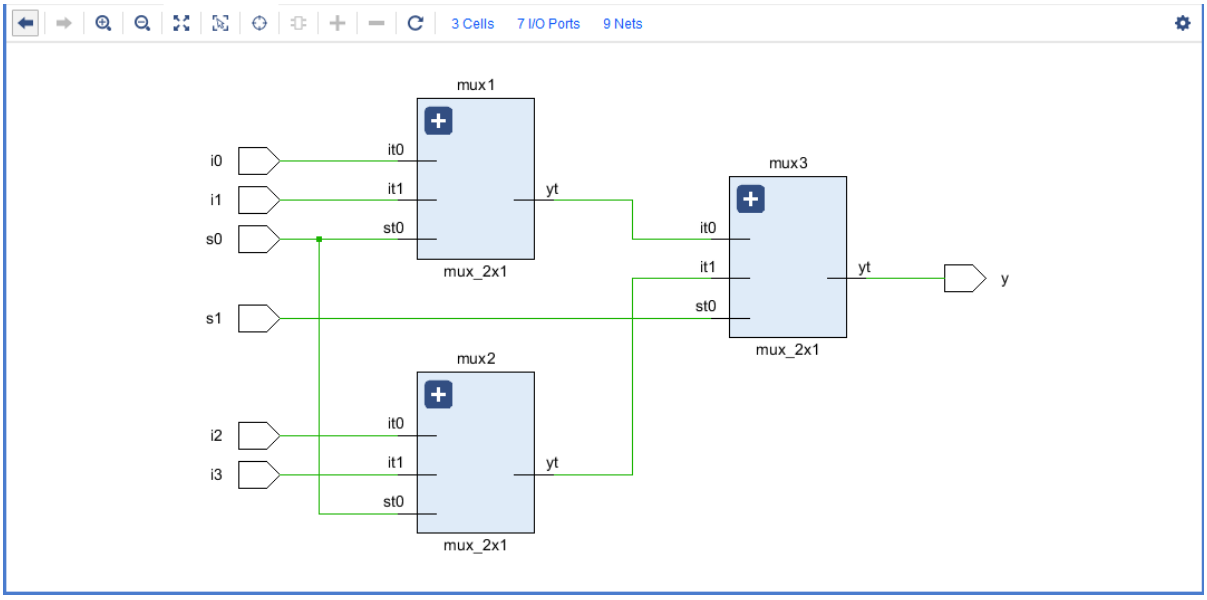

Figure 3.11 Schematic of the 4x1 multiplexer using structural modeling and using only 2x1 MUX

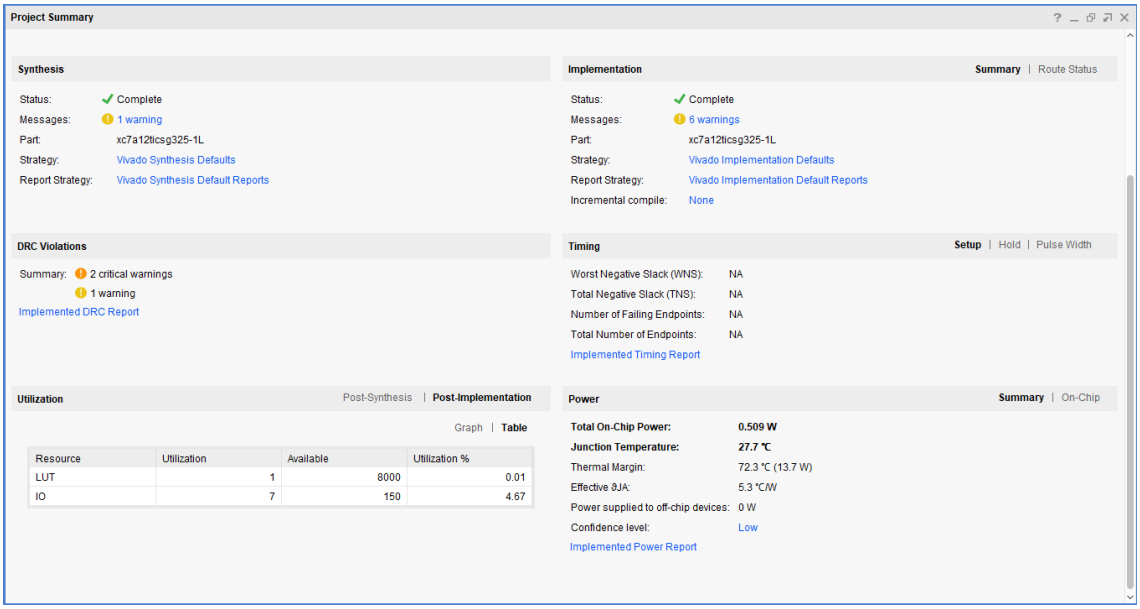

Figure 3.12 Project Summary of the 4x1 multiplexer using structural modeling and using only 2x1 MUX

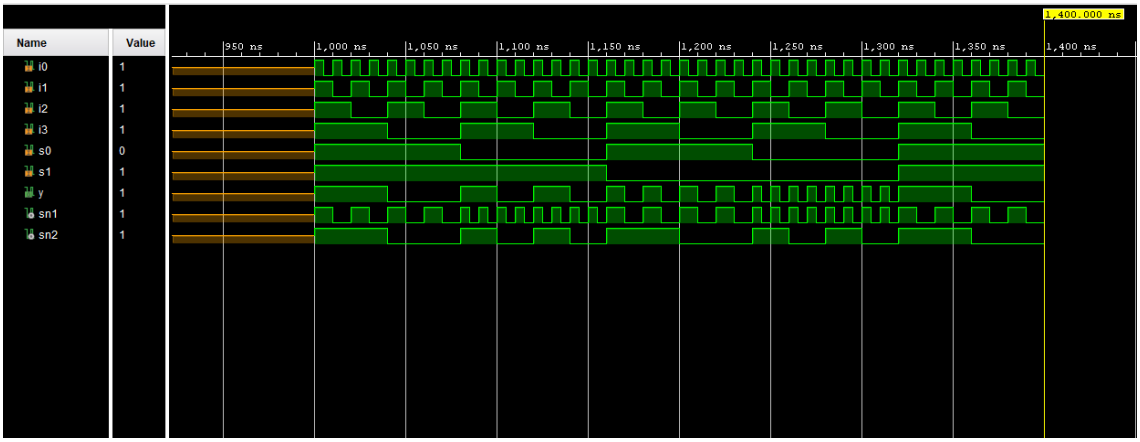

Figure 3.13 Simulation of the 4x1 multiplexer using structural modeling and using only 2x1 MUX

3.4.5 Implement an 8x1 MUX using structural modeling and using 4x1 and 2x1 MUX

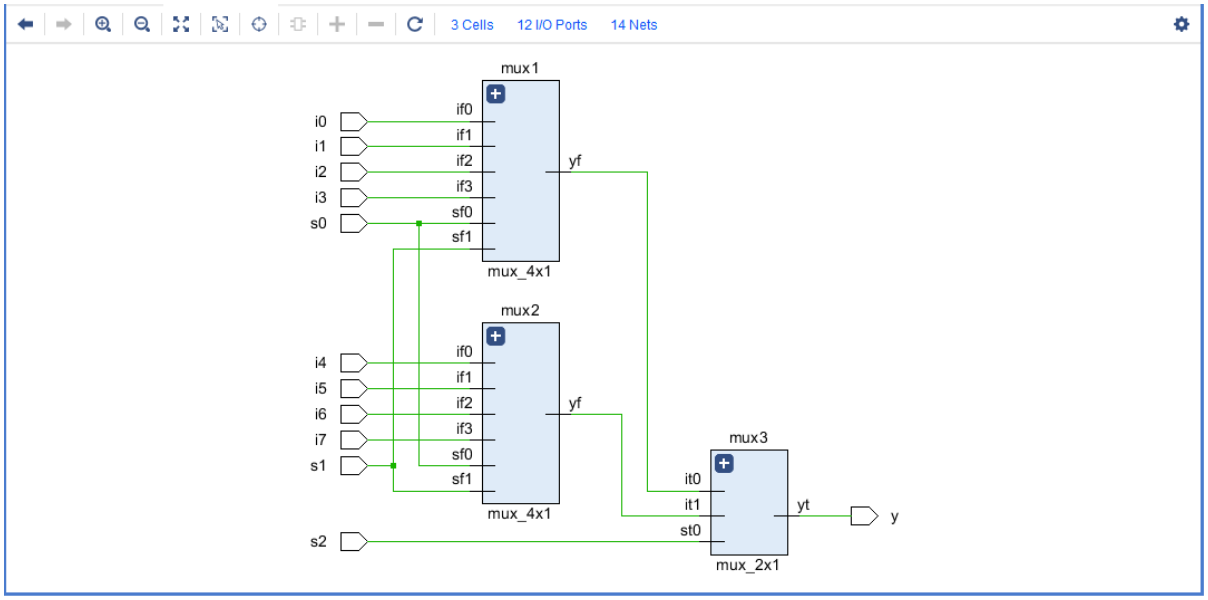

Figure 3.14 Schematic of the 8x1 MUX using structural modeling and using 4x1 and 2x1 MUX

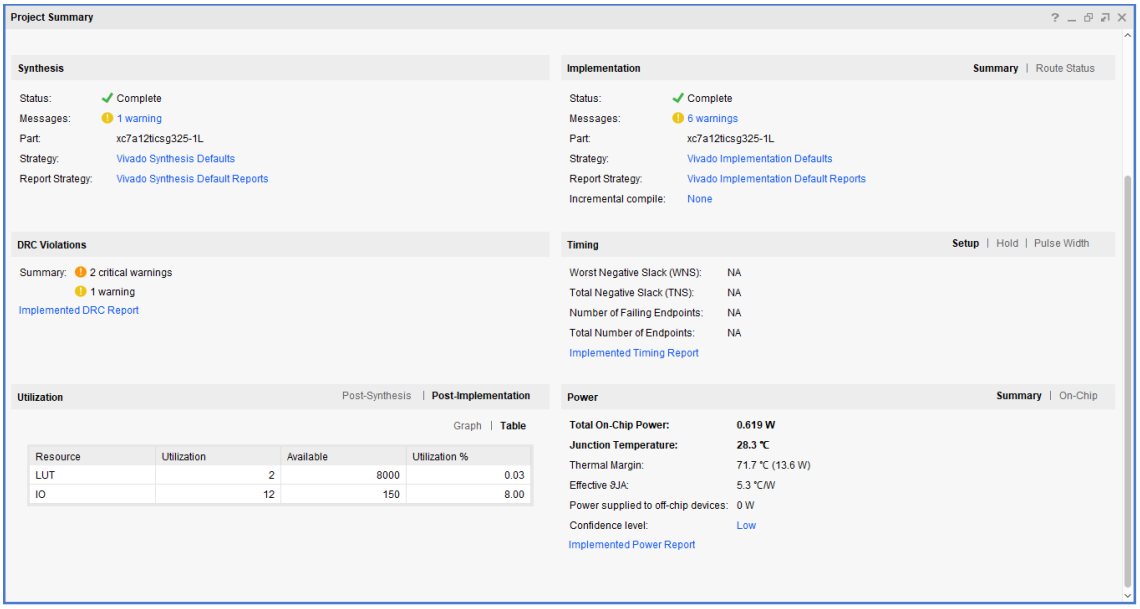

Figure 3.15 Project Summary of the 8x1 MUX using structural modeling and using 4x1 and 2x1 MUX

Figure 3.16 Simulation of the 8x1 MUX using structural modeling and using 4x1 and 2x1 MUX

3.5 Summary

Tabular comparison of all the codes in terms of area and power usage.

Name of the Entity	No. of LUT used	Total On chip Power
2x1 multiplexer using dataflow modeling	1	0.409W
2x1 multiplexer using behavioral modeling using case statement	1	0.409W
4x1 multiplexer using dataflow modeling	1	0.509W
4x1 multiplexer using structural modeling and using only 2x1 MUX	1	0.509W
8x1 MUX using structural modeling and using 4x1 and 2x1 MUX	2	0.619W

Table 3.4 comparision of Area and power requirements for different types of multiplexers (2x1, 4x1 and 8x1).

Chapter *4*

Experiment - 4

4.1 Name of the Experiment

Implement 4 bit ripple carry adder using structural modeling. Implement 4 bit adder/subtractor using structural modeling.

4.2 Theory

4.2.1 Ripple Carry Adder

Ripple carry adder is a combinational logic circuit. It is also known as **n-bit parallel adder** . It is used for the purpose of adding two n-bit binary numbers. It requires **n** full adders in its circuit for adding two n-bit binary numbers. Each full adder in ripple carry adder requires a C_{in} which is the C_{out} of the previous full adder. The layout of a ripple-carry adder is simple, which allows fast design time however the ripple carry adder is relatively slow because of the **gate delay**.

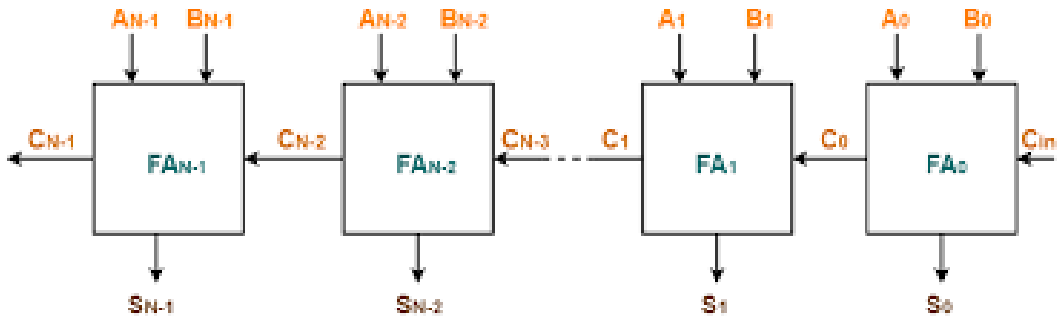

Figure 4.1 N-bit Ripple Carry Adder

4.2.2 4-bit Ripple Carry Adder

In a **4-bit Ripple carry adder** , Inputs given are two 4-bit numbers and C_{in} given to the rightmost first **Full adder** is 0 . Carry out from the last bit is available after **8** gate delays and Sum is availabe after **7** gate delays.

4.2.3 4-bit Adder/Subtractor with Control Input

A **Binary Adder-Subtractor** is one which is capable of both addition and subtraction of binary numbers in one circuit itself. The operation being performed depends upon the binary value the control signal holds. The circuit requires **XOR gates** and **Full adder circuits**.
Let us consider a 4-bit adder subtractor with inputs as A and B In this circuit , control input K is given which is either 0 or 1 and accordingly it decides whether the operation should be **Addition** or **Subtraction**.

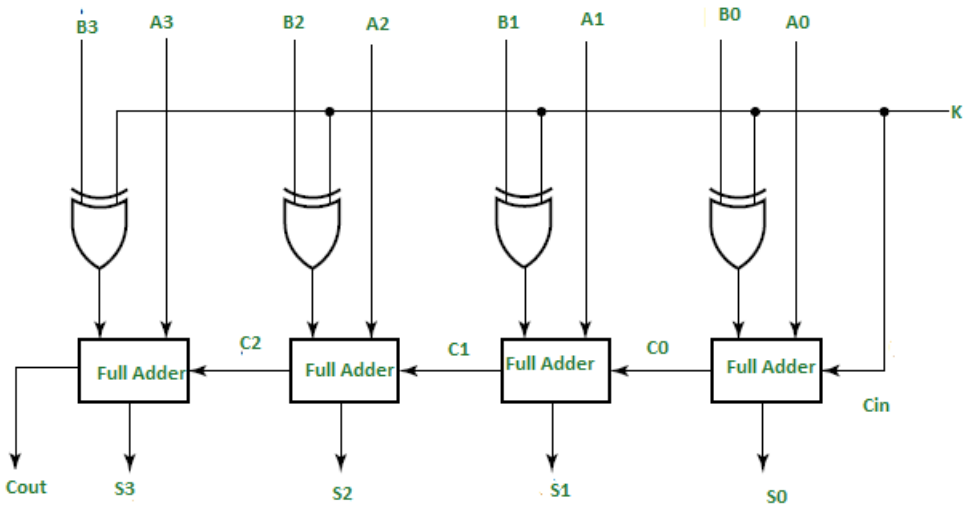

Figure 4.2 4-bit Adder Subtractor with Control Input (K)

The various examples of inputs given and their outputs are (A,B = inputs and M = Control Input):

1. If M=0,A=5 and B=9 Then Sum=e and C_{out}=0
2. If M=1,A=+5 and B=+3 Then Sum=2 and C_{out}=1
3. If M=1,A=+3 and B=+5 Then Sum=e and C_{out}=0

4.3 Coding Techniques used

1. ***Use structural modeling to implement a 4 bit adder to add two 4 bit vectors A and B with initial carry*** C_{in}

   In this modeling of 4 bit adder, we have used 4 Full Adders with C_{in} given to the rightmost first Full adder and value given to C_{in} is 0. Then we used the **Vector** Datatype of VHDL to break the inputs A into A_0,A_1,A_2 and A_3 and B into B_0,B_1,B_2 and B_3. Then we give A_0 and B_0 to the first Full adder along with C_{in} thus producing S_0 and C_{out}. This process is continued for the rest bits of A and B and in each Full adder , the Carry out produced is given as C_{in} for the next Full adder and rest bits of Sum are also produced.
   The Sum and Carry-out are :
   $Sum = S_3S_2S_1S_0$
   $C_{out} =$ Carry out bit from the last Full adder

2. ***Use structural modeling to implement 4 bit adder/subtractor with inputs A and B and control input as M***

   In this modeling of 4 bit adder/subtractor , we used 4 Full Adders with with inputs as A and (XOR of B and M). First input given to each The control input is A and the second input given is XOR of B_i with M where i=0,1,2 and 3. M is given as C_{in} to the first Full adder and for the rest of the Full adders require A and B with C_{in} as the C_{out} from the last Full adder. The control input M decides whether the operation is addition or subtraction according to as M is 0 or 1 . Carry out is the carry out from the last full adder. Sum is the combination of S_0,S_1,S_2 and S_3 .
   The Sum and Carry-out are :
   $Sum = S_3S_2S_1S_0$
   $C_{out} =$ Carry out bit from the last Full adder

4.4 Simulation and Results

4.4.1 Use structural modeling to implement a 4 bit adder with inputs A and B and initial carry C_{in}

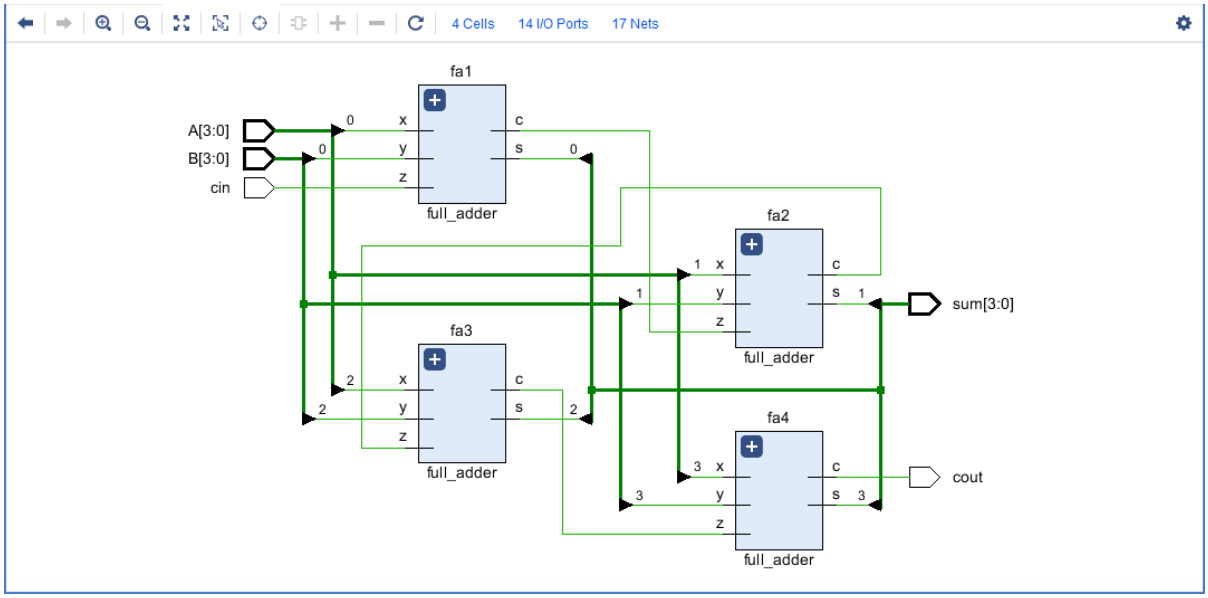

Figure 4.3 Schematic of the 4 bit adder with initial carry C_{in}

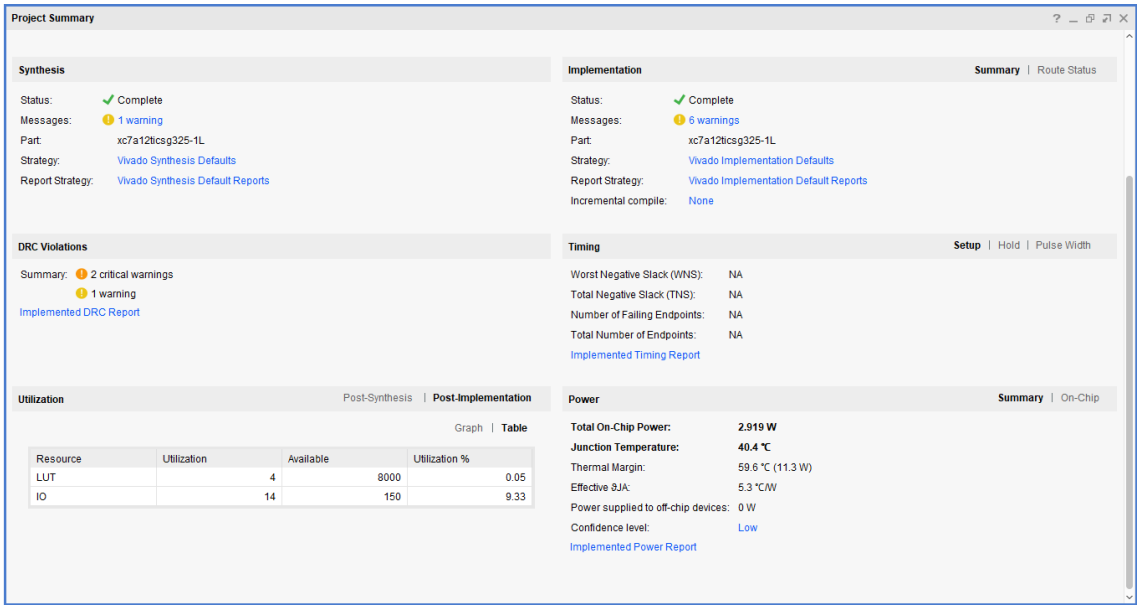

Figure 4.4 Project Summary of the 4 bit adder with initial carry C_{in}

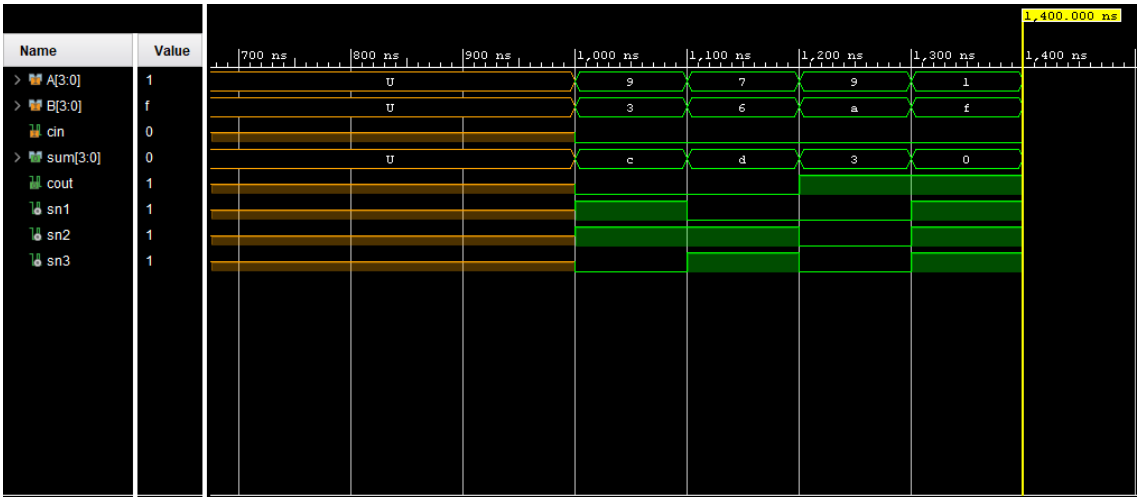

Figure 4.5 Simulation of the 4 bit adder with initial carry C_{in}

4.4.2 Use structural modeling to implement a 4 bit adder/subtractor with inputs as A and B and control input as M

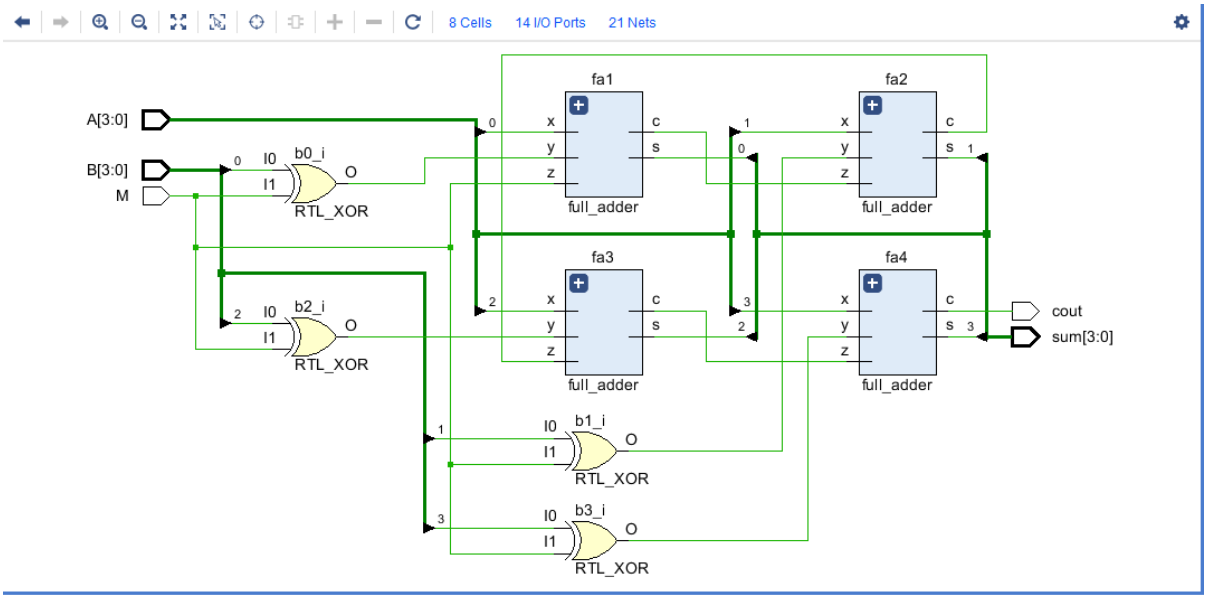

Figure 4.6 Schematic of the 4 bit adder/subtractor with control input as M

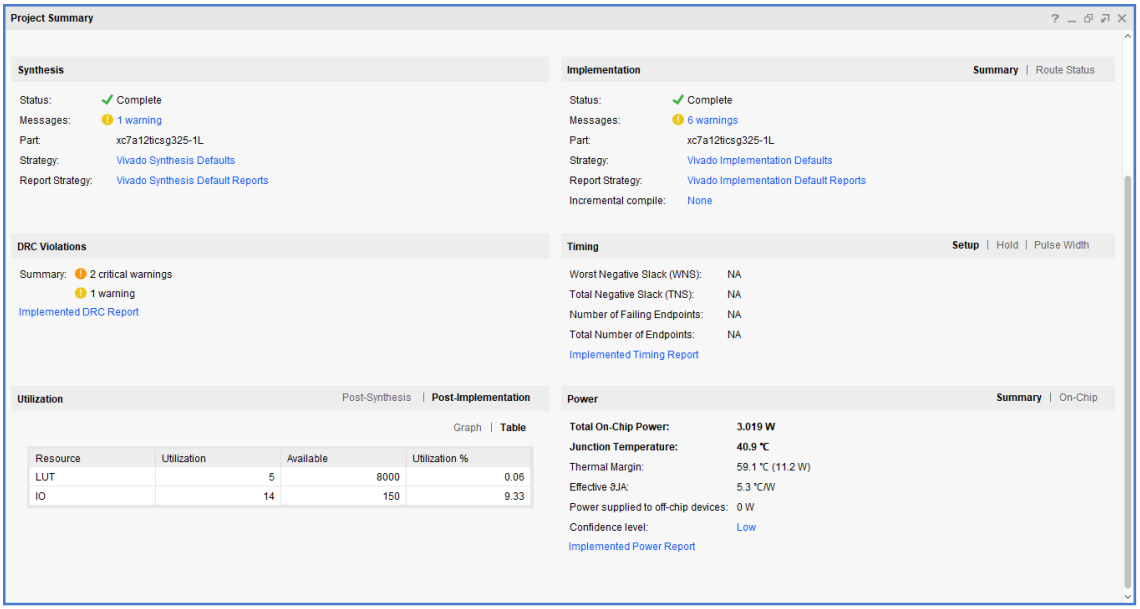

Figure 4.7 Project Summary of the 4 bit adder/subtractor with control input as M

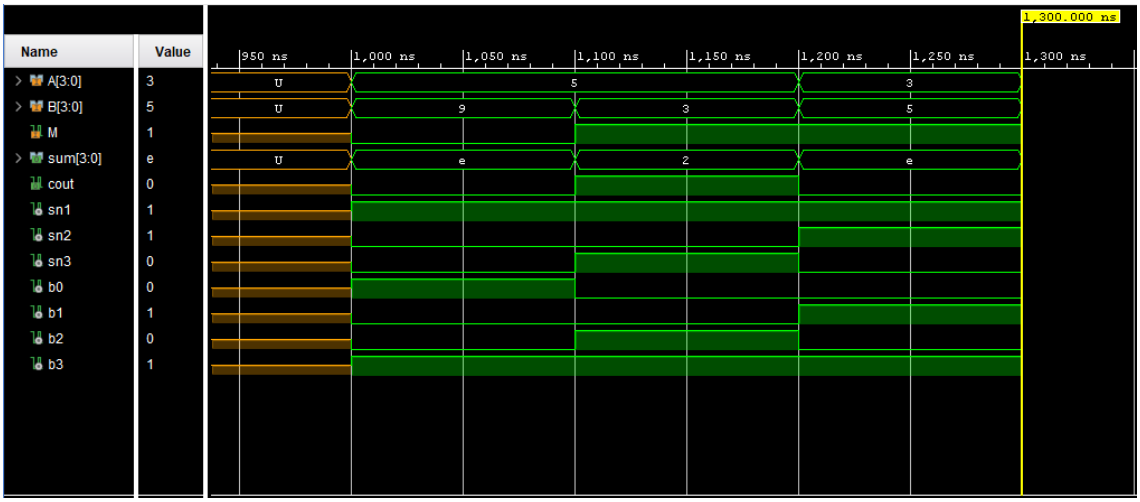

Figure 4.8 Simulation of the 4 bit adder/subtractor with control input as M

4.5 Summary

Tabular comparison of all the codes in terms of area and power usage.

Name of the Entity	No. of LUT used	Total On chip Power
4 bit adder with initial carry C_{in}	4	2.919W
4 bit adder/subtractor with control input as M	5	3.019W

Table 4.1 comparision of Area and power requirements for 4 bit adder and 4 bit adder/subtractor with control input as M.

Chapter *5*

Experiment - 5

5.1 Name of the Experiment

Implement 1 to 2 , 2 to 4 and 3 to 8 line decoder using dataflow, behavioral and mixed modeling in VHDL. Implement Boolean functions using decoders.

5.2 Theory

5.2.1 Decoder

In digital electronics, a **binary decoder** is a combinational logic circuit that converts binary information from the n coded inputs to a maximum of 2^n unique outputs. Decoders are used in a wide variety of applications , including data multiplexing and data demultiplexing, seven segment displays and as address decoders for memory and port mapper I/O.
A binary decoder is usually implemented as either a stand-alone integrated circuit (IC) or as part of a more complex IC. Widely used decoders are often available in the form of standardized ICs.

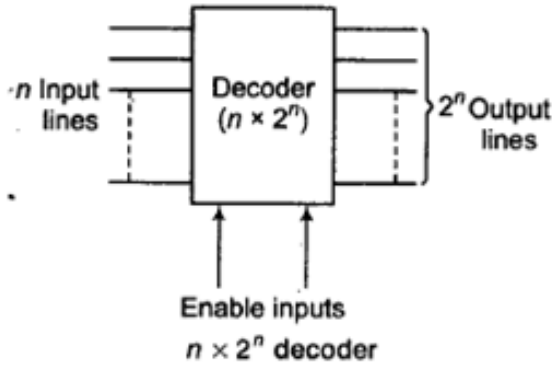

Figure 5.1 n x 2^n decoder

5.2.2 2 to 4 line decoder

In 2 to 4 decoder, There are 2 input lines, 1 **enable** line and 4 output lines. The truth table of 2 to 4 decoder is :

en	A	B	y_3	y_2	y_1	y_0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Table 5.1 2x4 line decoder Truth table

5.2.3 3 to 8 line decoder

In 3 to 8 decoder, There are 3 input lines , 1 **enable** line and 8 output lines. The truth table of 3 to 8 decoder is :

en	A	B	C	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

Table 5.2 3x8 line decoder Truth table

5.2.4 Theory of *Gray Code*

The **reflected binary code (RBC)** or **Gray code** is named after Frank Gray, is an ordering of the two binary numeral system such that two successive values differ in only one bit (binary digit).
For example, the representation of the decimal value ”1” in binary would be ”001” and ”2” would be ”010”. In gray code, these values are represented as ”001” and ”011”. Gray codes are widely used to prevent spurious output from electromechanical switches and to facilitate error correction in digital communications such as digital terrestrial television and some cable TV systems.

5.3 Coding Techniques used

1. ***Using behavioral architecture, implement a 2 to 4 line decoder***

   In this modeling of 2 to 4 line decoder, i have used **vector** data type of input X and output Y and single input for enable input . Then I used **Case constructs** of VHDL to evaluate the results Y. For ex: If X = "00" and en=1 then Y="0001". If en=0 then Y="0000" irrespective of inputs taken. In 2 to 4 decoder, there are 2 input signals and 4 output signals.

2. ***Using dataflow modeling, implement a 3 to 8 line decoder***

   In this modeling of 3 to 8 line decoder, there are 3 input signals and 8 output signals . For the input signal(x) and output signal(y) , I have used the vector data-type of VHDL to represent them. Then I used dataflow modeling to assign values to each component of the output vector. For Example: for y_0 i have assigned the **AND** of complement of x_0 , x_1 and x_2. Same procedures are followed for other components of y.

3. ***Implement function F(A,B,C) = $\sum$(1,2,5,7) using 3 to 8 line decoder using structural architecture.***

   In this modeling of the function , I have used a 3 to 8 line decoder unit as a module . Input are 3 signals and output are 8 signals. Then i have selected the signals 1,2,5 and 7 and passed them to **OR gate** to find the final answer. Therefore modules used are one 3x8 decoder and 1 OR gate.

4. ***Design an entity to encode a 4 bit array of binary number system to corresponding 4 bit array of gray code.***

   I have used Dataflow modeling to design this entity. Input given is a 4 bit vector (x) and output is another 4 bit vector (y). The theory used for coding is that the bit y_3 is same as x_3 . For other bits like y_n where n ranges from 0 to 2 , $y_n = x_{n+1} \oplus x_n$. Logic used is the XOR gate .

5. ***Implement a 4 to 16 decoder using only 2 to 4 decoders using structural modeling.***

   In this modeling of 4 to 16 decoders, I have used **5** 2x4 decoders. In the first decoder I have given the first two input components of the input signal and enable as bit '1'. Then for subsequent decoders, i have given the last two bit components of the input signal and enable signal as one of the **4 outputs** that came out of the first decoder. Therefore , The model acts as a 4 to 16 decoder with 4 inputs and 16 ouputs.

5.4 Simulation and Results

5.4.1 Using behavioral modeling , implement a 2 to 4 line decoder

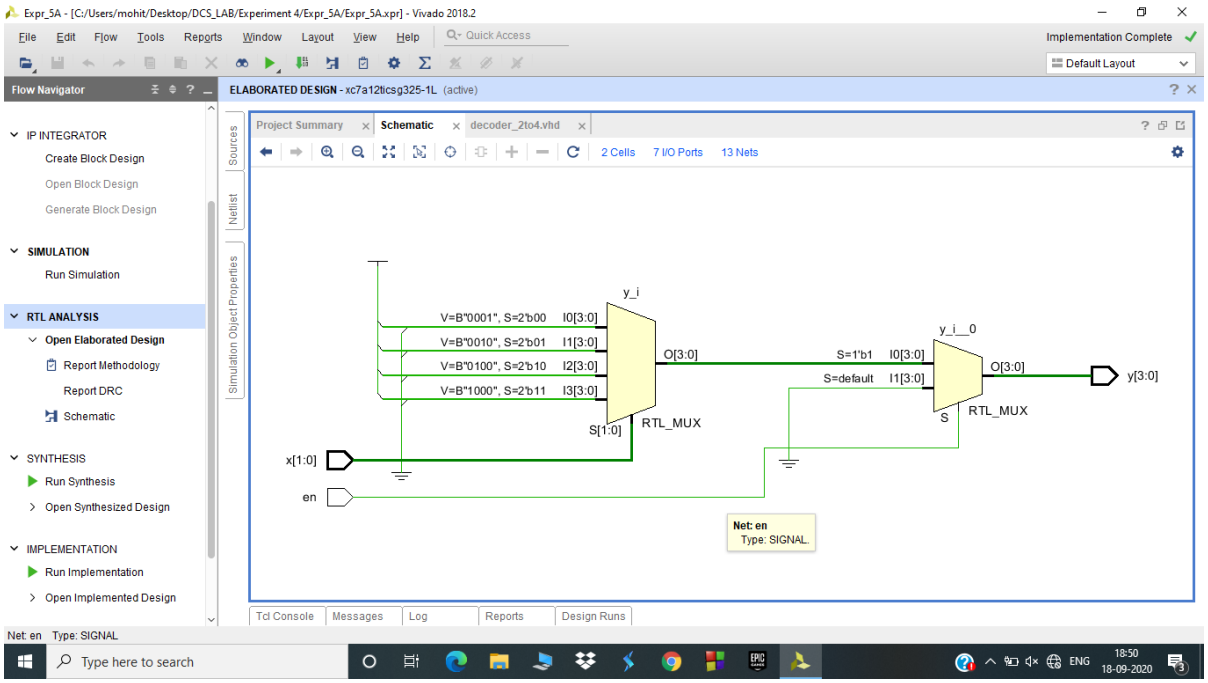

Figure 5.2 Schematic of the 2 to 4 line decoder using behavioral modeling

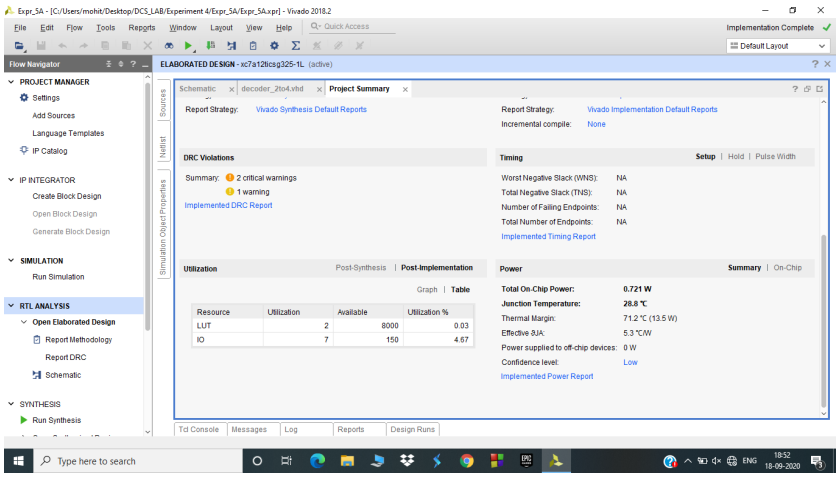

Figure 5.3 Project Summary of the 2 to 4 line decoder using behavioral modeling

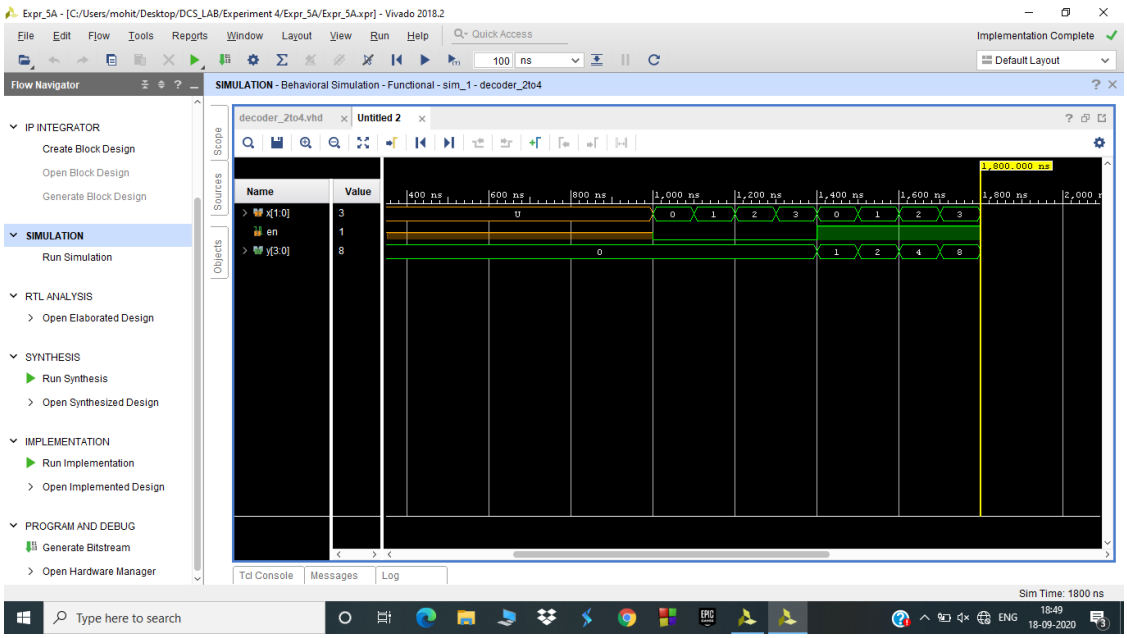

Figure 5.4 Simulation of the 2 to 4 line decoder using behavioral modeling

5.4.2 Using dataflow modeling , implement a 3 to 8 line decoder

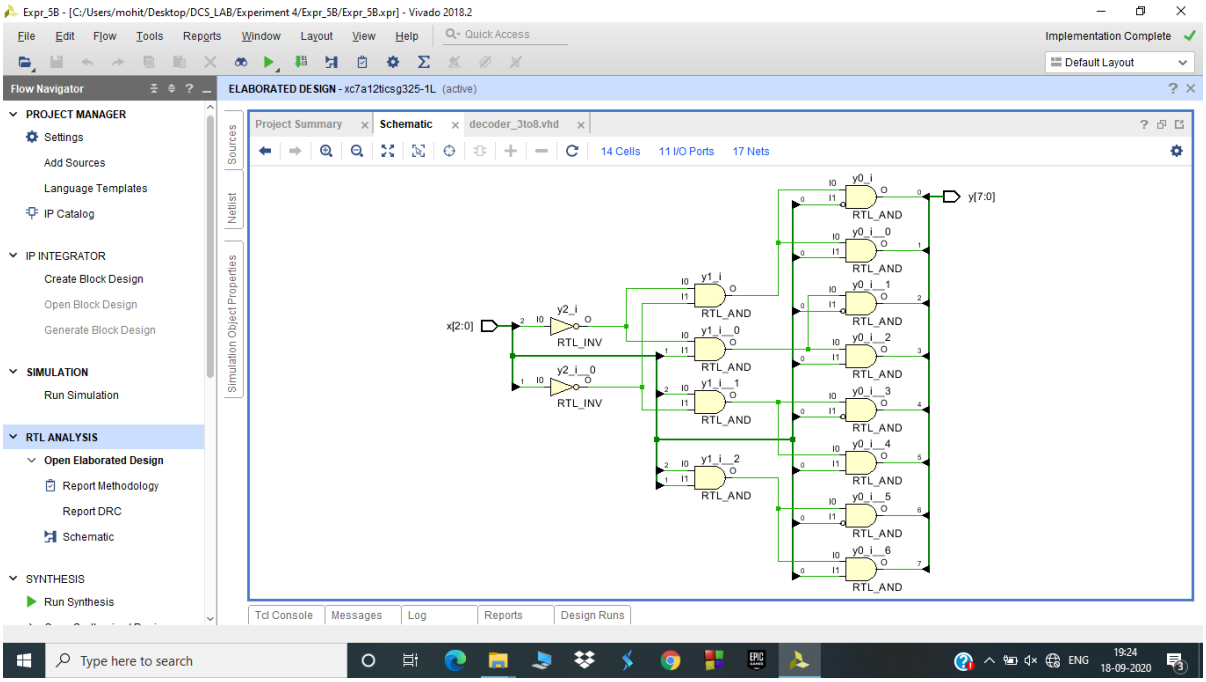

Figure 5.5 Schematic of the 3 to 8 line decoder using dataflow modeling

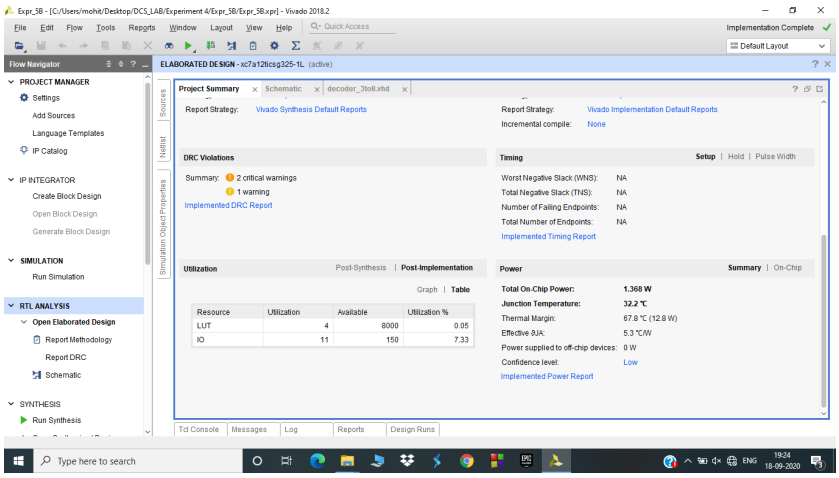

Figure 5.6 Project Summary of the 3 to 8 line decoder using dataflow modeling

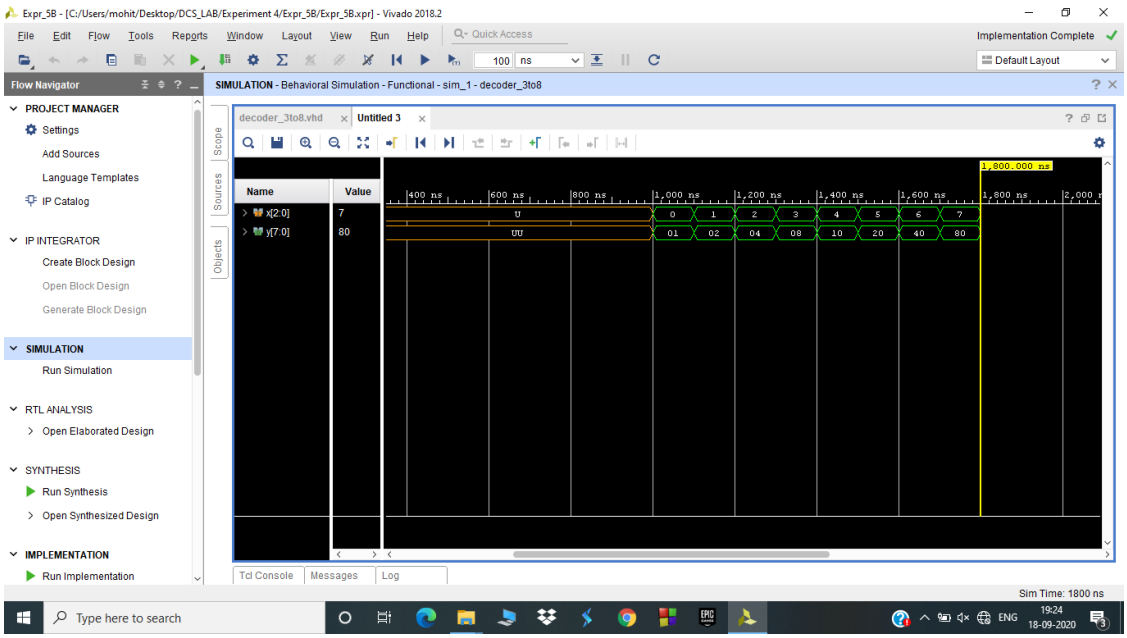

Figure 5.7 Simulation of the 3 to 8 line decoder using dataflow modeling

5.4.3 Implement function F=∑(1,2,5,7) using 3 to 8 line decoder using structural modeling

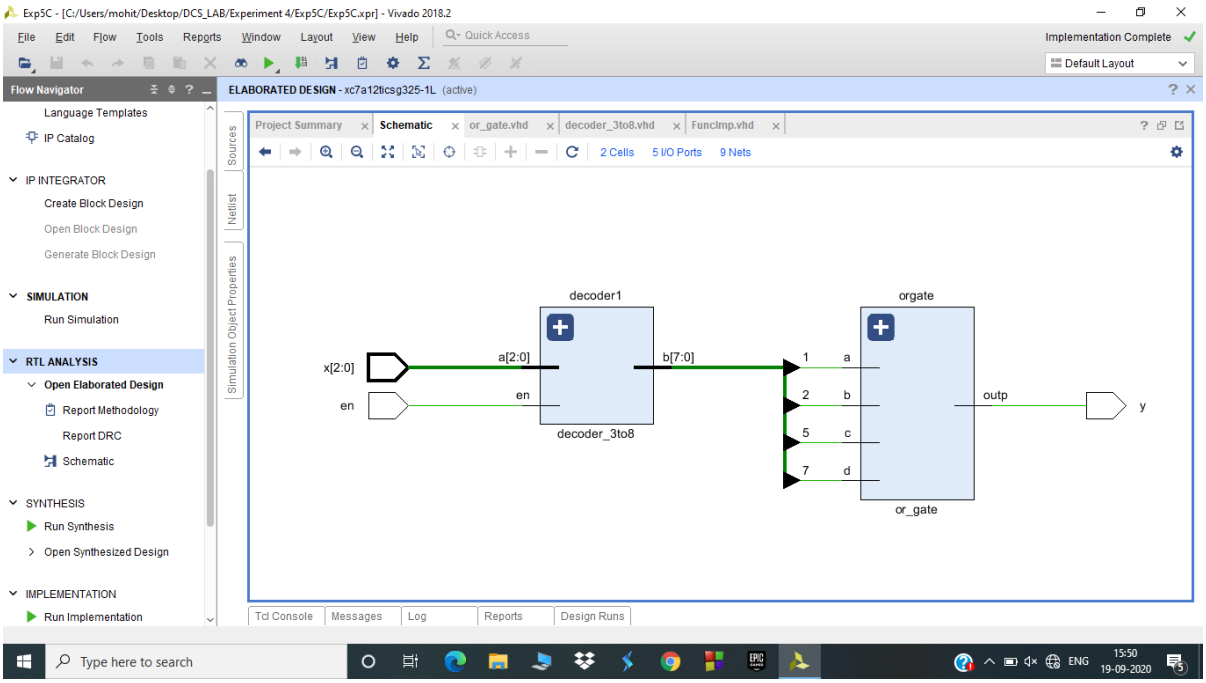

Figure 5.8 Schematic of the given function using 3 to 8 line decoder

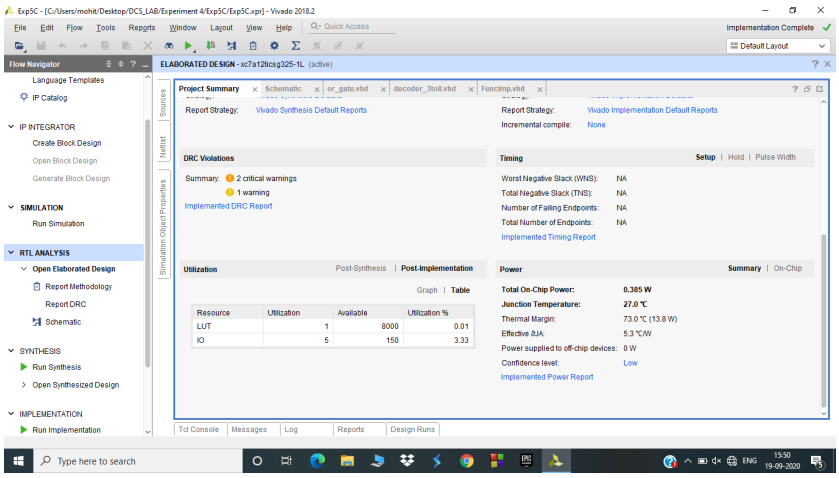

Figure 5.9 Project Summary of the given function using 3 to 8 line decoder

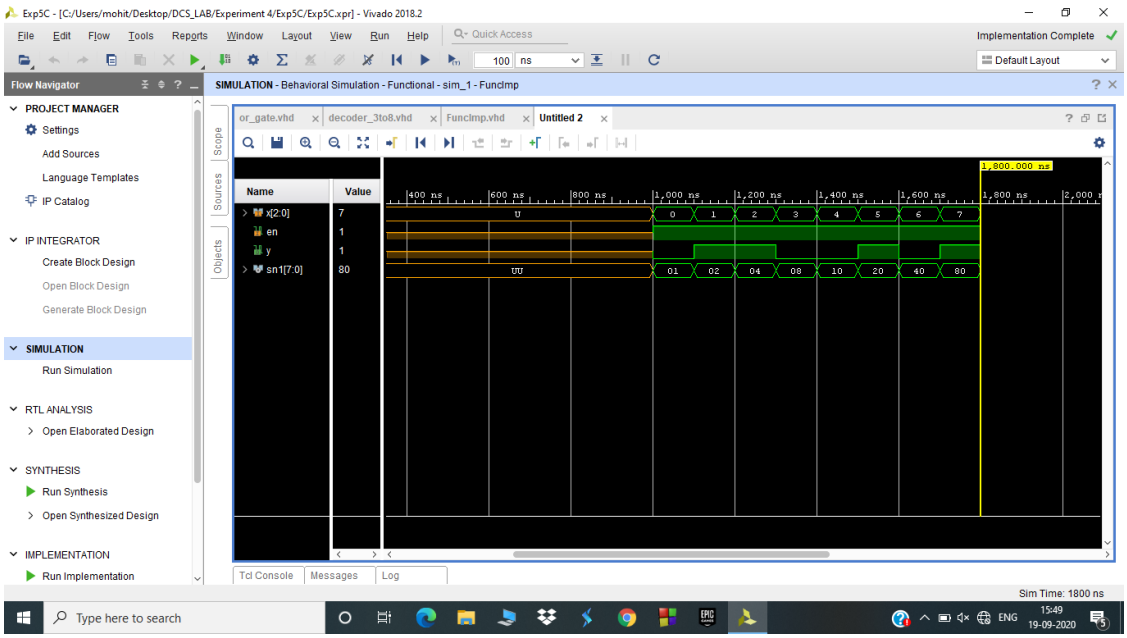

Figure 5.10 Simulation of the given function using 3 to 8 line decoder

5.4.4 Design an entity to encode 4 bit array of binary numbers to 4 bit array of gray code

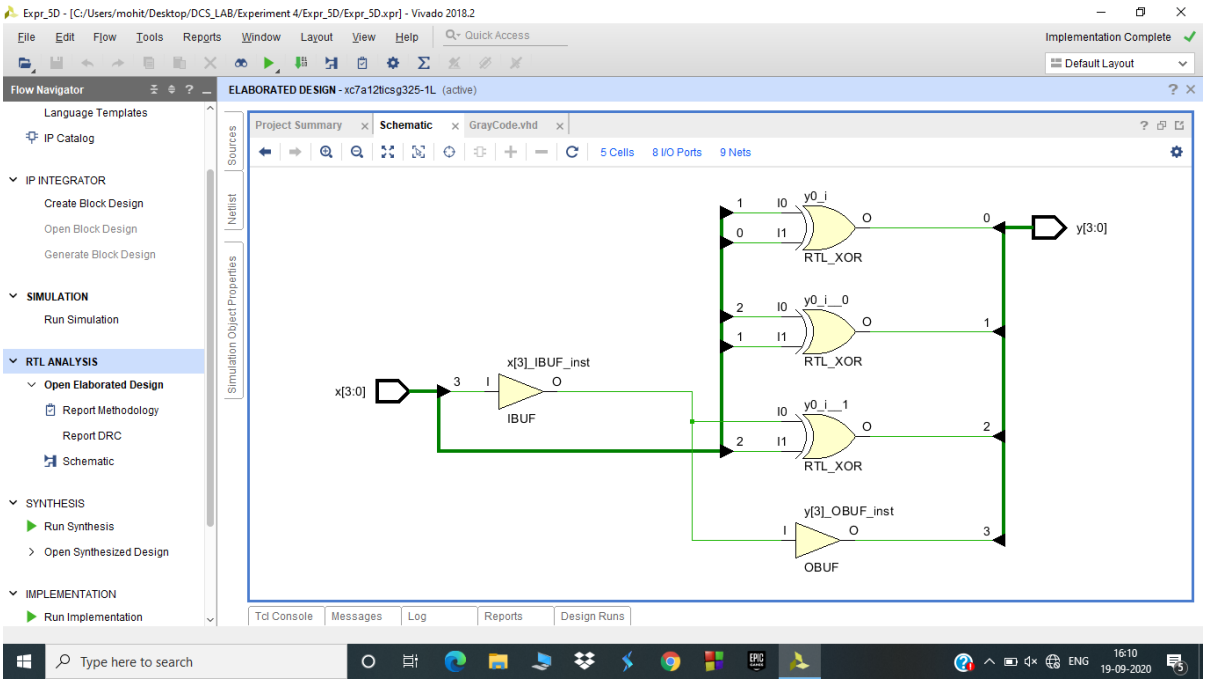

Figure 5.11 Schematic of the gray code entity

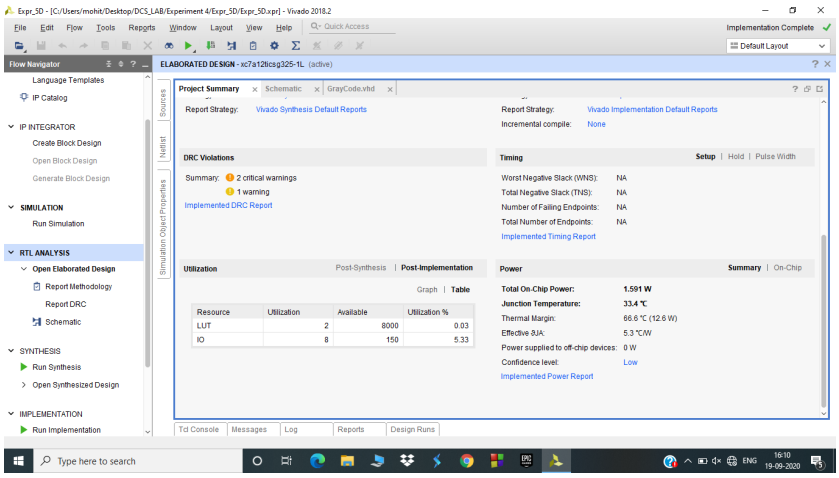

Figure 5.12 Project Summary of the gray code entity

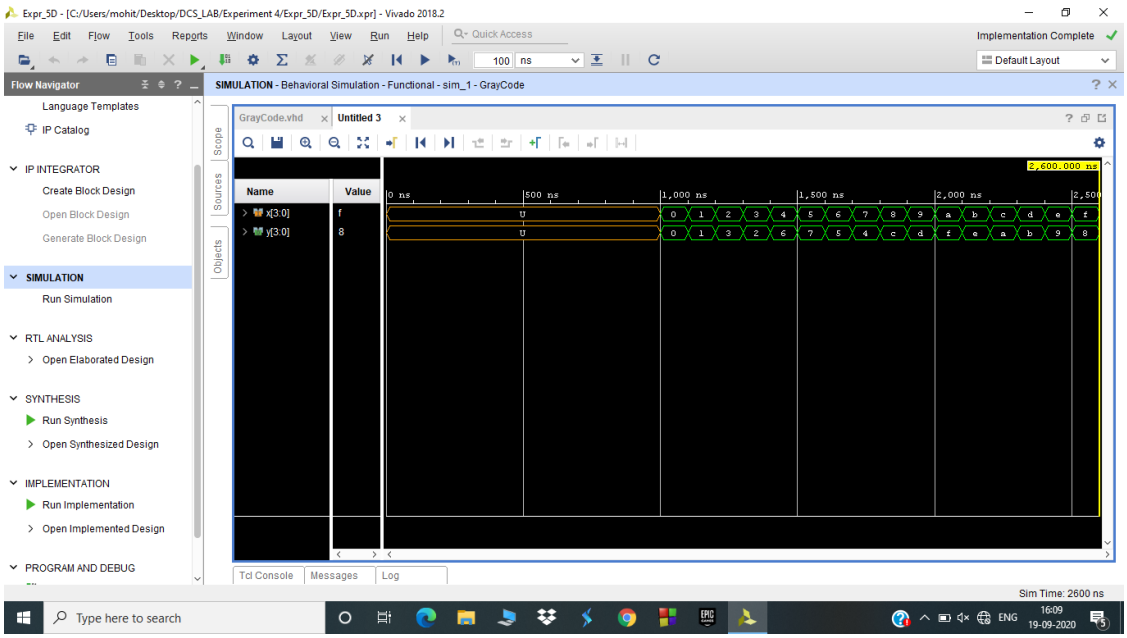

Figure 5.13 Simulation of the gray code entity

5.4.5 Implement a 4 to 16 decoder using only 2 to 4 decoders using structural modeling

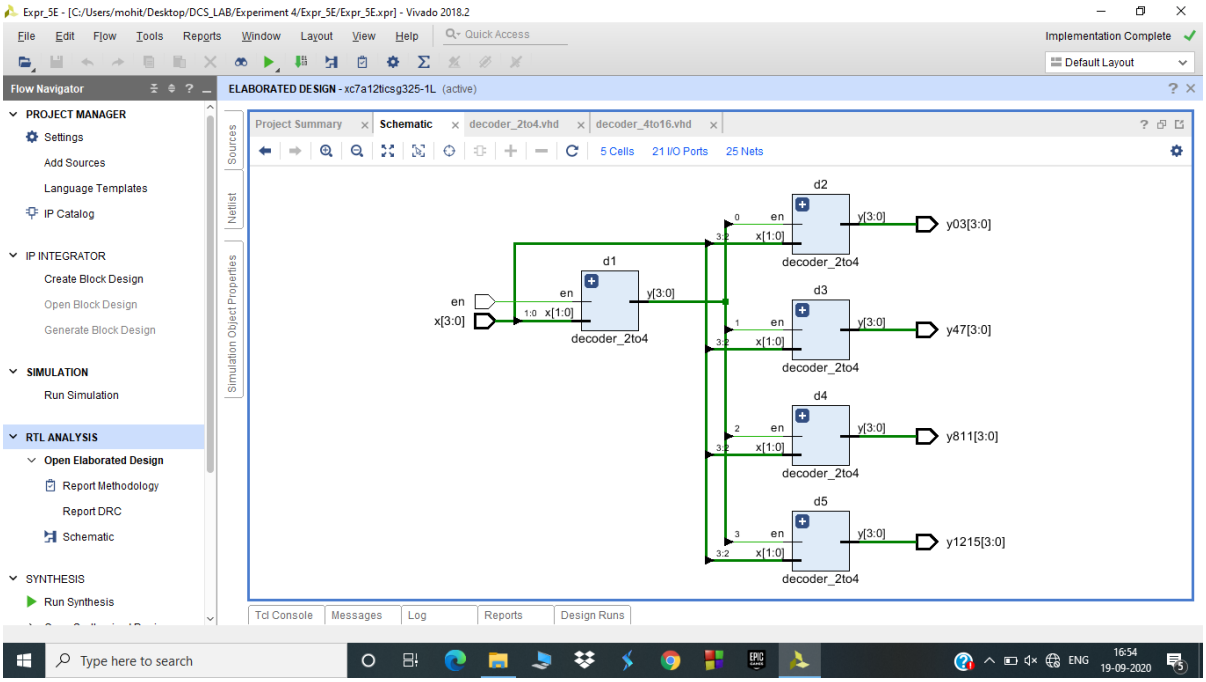

Figure 5.14 Schematic of the 4 to 16 decoder using 2 to 4 decoders as modules

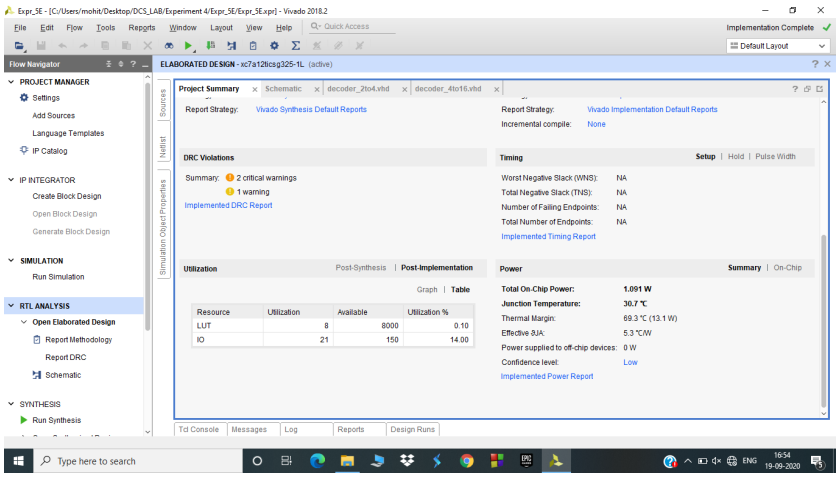

Figure 5.15 Project Summary of the 4 to 16 decoder using 2 to 4 decoders as modules

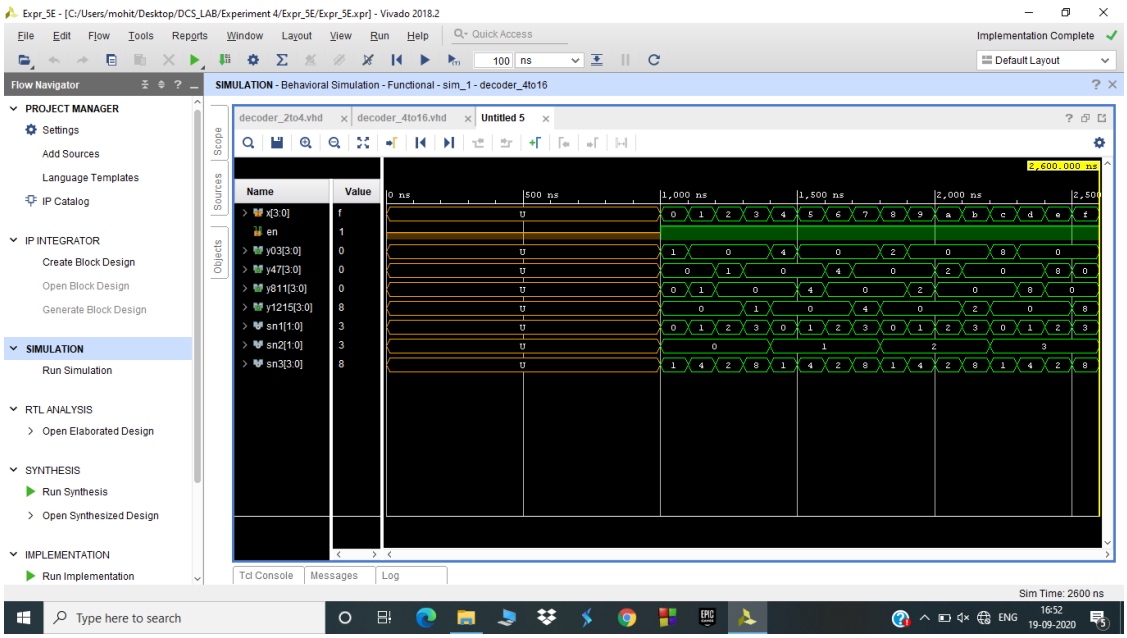

Figure 5.16 Simulation of the 4 to 16 decoder using 2 to 4 decoders as modules

5.5 Summary

Tabular comparison of all the codes in terms of area and power usage.

Name of the Entity	No. of LUT used	Total On chip Power
2 to 4 line decoder using behavioral modeling	2	0.721W
3 to 8 line decoder using dataflow modeling	4	1.368W
Implementing given function using 3 to 8 decoders	1	0.385W
Designing entity to encode 4 bit binary to 4 bit gray code	2	1.591W
4 to 16 decoder using 2 to 4 decoders	8	1.091W

Table 5.3 comparision of Area and power requirements for different types of decoders and gray code entity.

Chapter *6*

Experiment - 6

6.1 Name of the Experiment

Design D latch, JK flip flop, RS flip flop and T flip flop using behavioral and structural modeling

6.2 Theory

6.2.1 Latch

In digital electronics, a Latch is one kind of a logic circuit, and it is also known as a bistable-multivibrator. Because it has two stable states namely active high as well as active low. It works like a storage device by holding the data through a feedback lane. It stores 1-bit of data as long as the apparatus is activated. Once enable is declared then instantly latch can change the stored data. It constantly trials the inputs once enable signal is activated. The latches can be classified into different types:

- SR Latch
- D Latch
- T Latch
- JK Latch

6.2.1.1 D Latch

En	D	Next State
0	X	No change
1	0	Q=0, Reset state
1	1	Q=1, Set state

Table 6.1 D Latch truth table

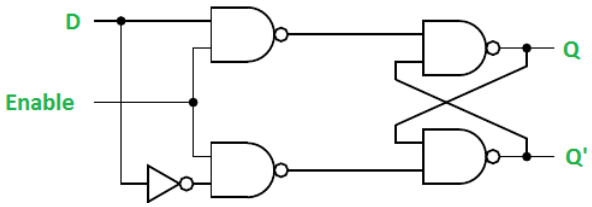

Figure 6.1 D Latch

6.2.2 Flip Flop

Flip Flop is the basic storage element in sequential logic. Flip-flops are building blocks of digital electronics systems used in computers, communications and many other types of systems. Flip flops are generally edge triggered although they can be level triggered also. A basic flip-flop can be constructed using four-NAND or four-NOR gates. Flip flops can be of various types :

- SR Flip Flop
- JK Flip Flop
- D Flip Flop
- T Flip Flop

6.2.2.1 SR Flip Flop

S	R	Present State	Next State
0	0	-	No change
0	1	-	0 (Reset state)
1	0	-	1 (Set state)
1	1	-	X (Undefined)

Table 6.2 SR Flip Flop Truth table (- means don't care , X means undefined)

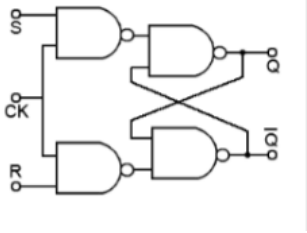

Figure 6.2 SR Flip Flop

6.2.2.2 JK Flip Flop

J	K	Q	Q_{n+1}
0	0	0,1	0,1 (No change)
0	1	-	0
1	0	-	1
1	1	0,1	1,0 (Toggle)

Table 6.3 JK Flip Flop truth table

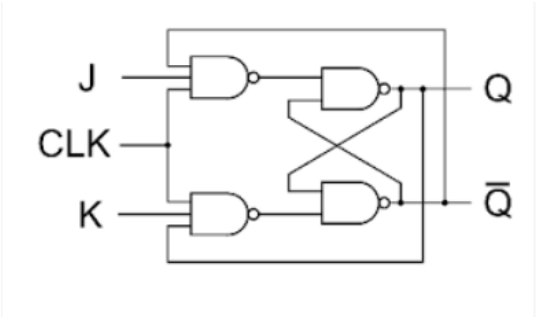

Figure 6.3 JK Flip Flop

6.2.2.3 D Flip Flop

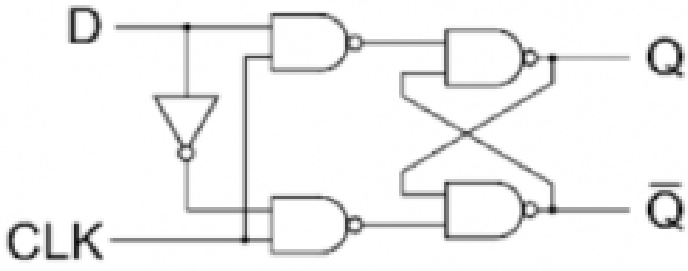

Figure 6.4 D Flip Flop

clk	D	Next state
0	0,1	0,1 (No change)
1	0	0
1	1	1

Table 6.4 D Flip Flop truth table

6.2.2.4 T Flip Flop

clk	T	Present State	Next state
0	-	0,1	0,1 (No change)
1	0	0,1	0,1
1	1	0,1	1,0

Table 6.5 T Flip Flop

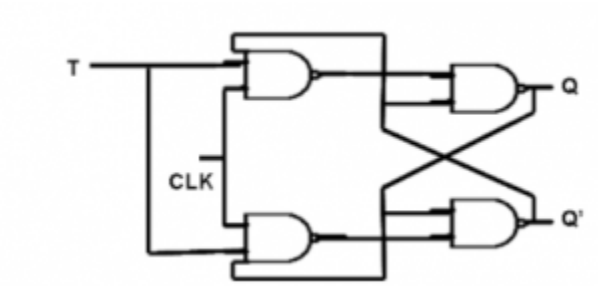

Figure 6.5 T flip flop

6.3 Coding Techniques used

1. ***Synchronous D latch using dataflow modeling***

   In this modeling of D Latch, I have used dataflow method and derived equation for Q and Qbar from the logic diagram of D Latch. The equations for Q and Qbar are as follows :
   Q = s1 nand Qbar where s1 = D *nand* en
   Qbar = s2 *nand* Q where s2 = D' *nand* en
   where **en** is the enable signal and **D** is the input signal.

2. ***Synchronous D Flip Flop using behavioral modeling***

   In this modeling of D Flip Flop , I have used the truth table of D Flip Flop to derive relations between Q , Qbar , D and clk . From the truth table we can easily determine that when clk=0 (falling edge) there is **no change** in the output. When clk=1 then Q = D.

3. ***Synchronous JK Flip Flop using structural modeling***

   In this modeling of JK Flip Flop, I have used D Flip Flop as a structural module and input (D) to D Flip Flop is s3 . Various equations regarding JK Flip Flop are as follows :
   s1 = J *nand* Qbar
   s2 = K' *nand* Q
   s3 = s1 *or* s2
   where J and K are inputs to JK Flip Flop and s1,s2 and s3 are intermediate signals and signal s3 is given to **D Flip Flop**.

4. ***Synchronous RS Flip Flop using behavioral modeling***

   In this modeling of RS Flip Flop , I have used the truth table of RS Flip Flop to derive relationship between inputs (S,R and clk) and ouputs (Q, Qbar). It can be easily seen from the truth table that when (S=0 and R=0) , there is **no change** in the ouput , when (S=0 and R=1) output is equal to 0 irrespective of the input , when (S=1 and R=0) output is equal to 1 irrespective of the input and when (S=1 and R=1) output is **indeterminate**.

5. ***Synchronous T Flip Flop using behavioral modeling***

   In this modeling of T Flip Flop , I have used the truth table of T Flip Flop to derive relationships between inputs (T and clk) to outputs (Q and Qbar) . It can be easily seen from the truth table that when clk=0 , there is **no change** in the output. When clk=1 and T=0 , there is **no change** in the output and when clk=1 and T=1 , the output is **toggled** i.e, 0 becomes 1 and 1 becomes 0.

6.4 Simulation and Results

6.4.1 Synchronous D Latch using dataflow modeling

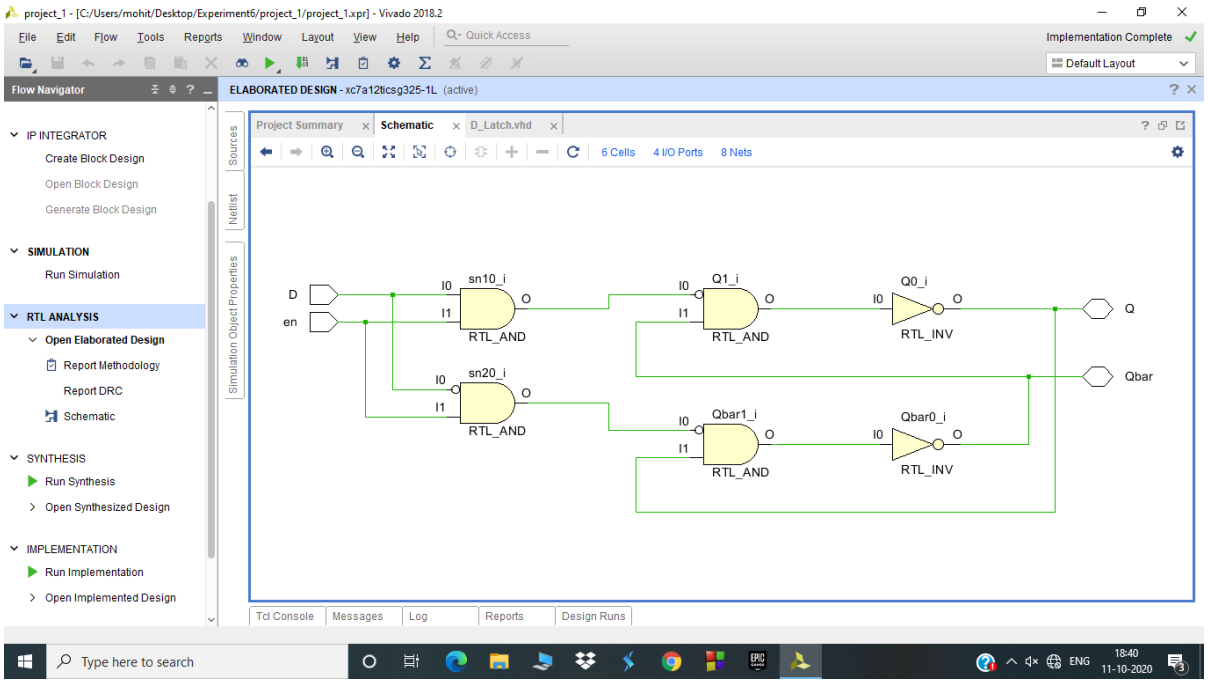

Figure 6.6 Schematic of synchronous D Latch using dataflow modeling

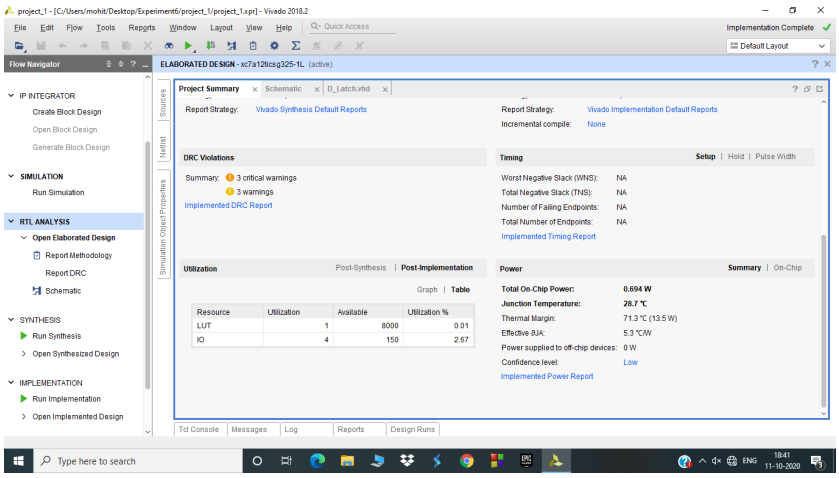

Figure 6.7 Project Summary of the synchronous D Latch using dataflow modeling

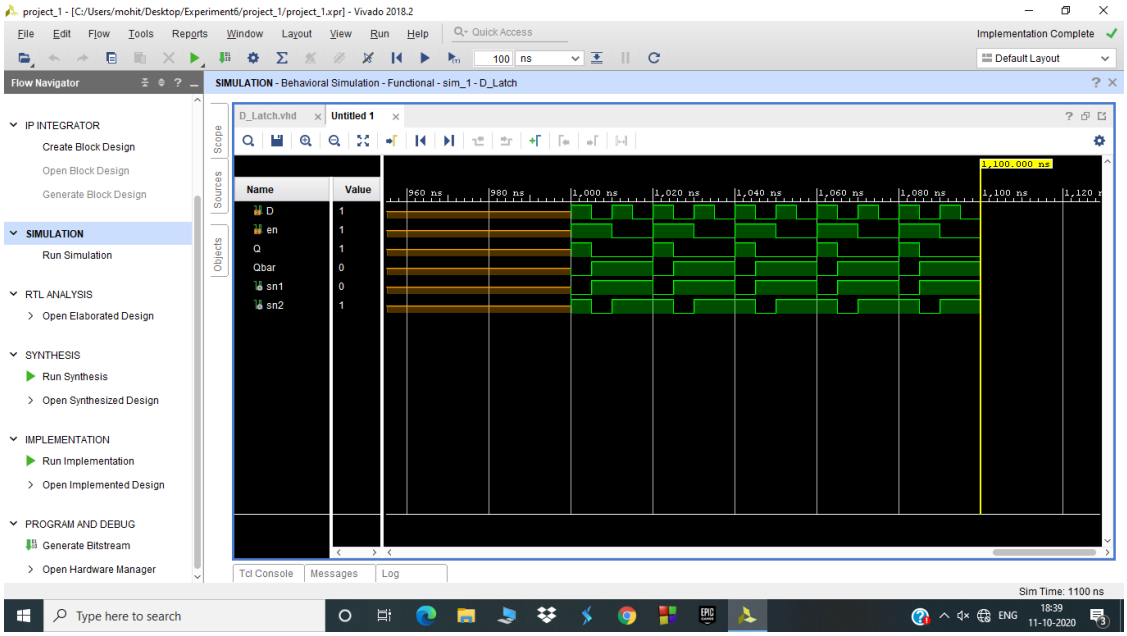

Figure 6.8 Simulation of the synchronous D Latch using dataflow modeling

6.4.2 Synchronous D Flip Flop using behavioral modeling

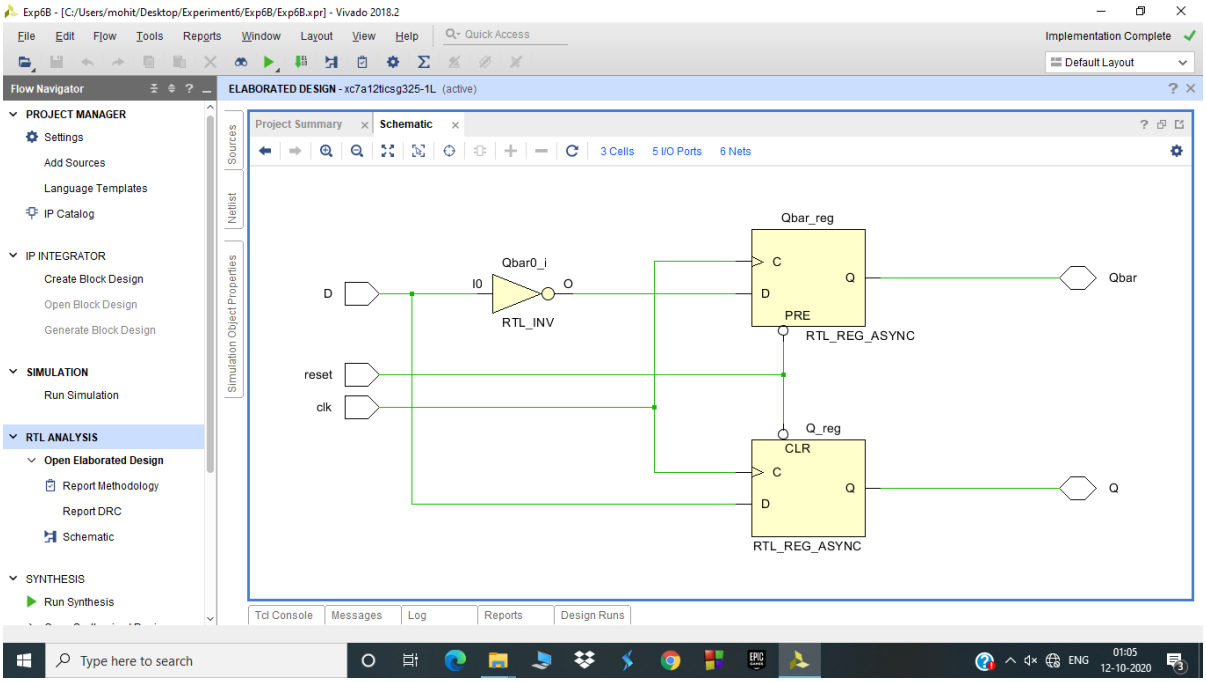

Figure 6.9 Schematic of the synchronous D Flip Flop using behavioral modeling

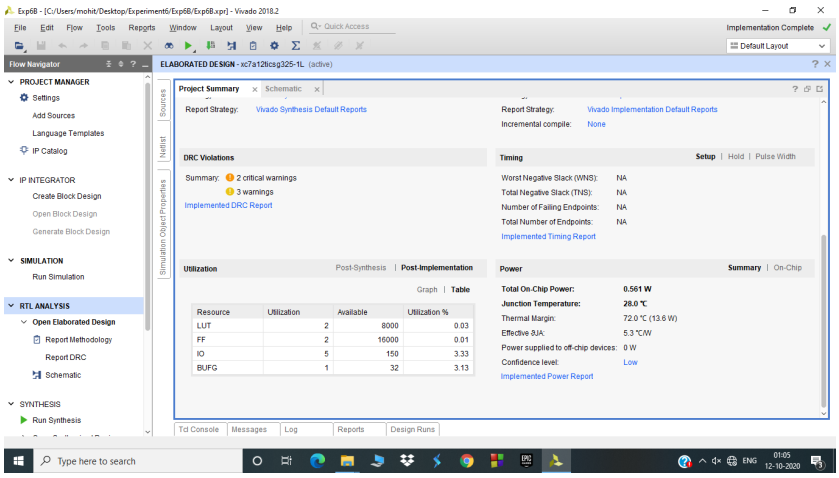

Figure 6.10 Project Summary of the synchronous D Flip Flop using behavioral modeling

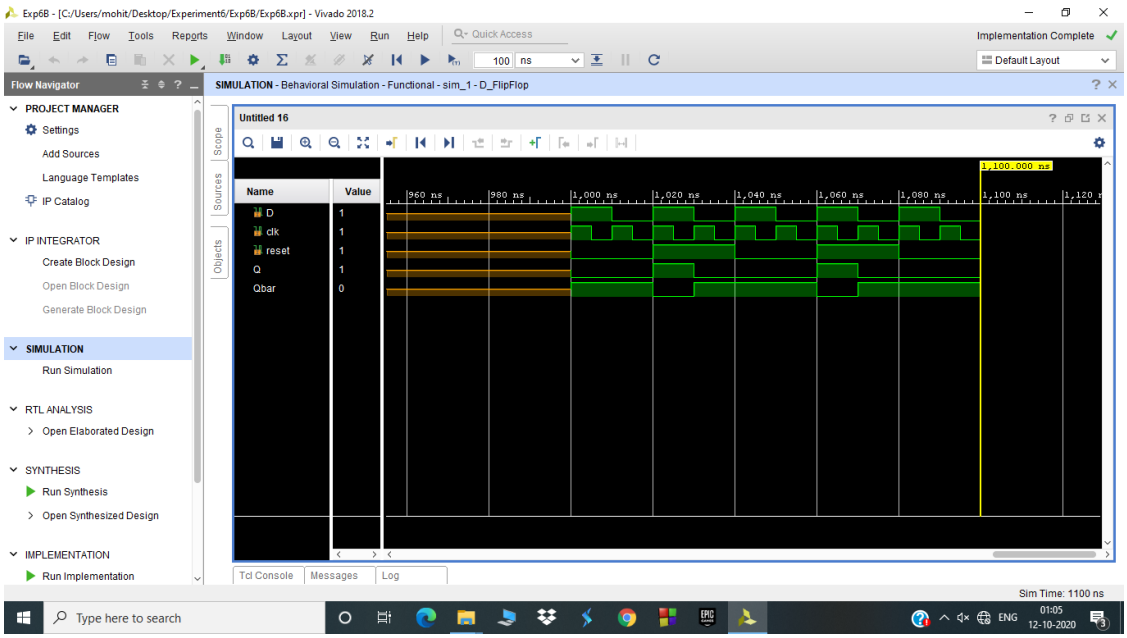

Figure 6.11 Simulation of the synchronous D Flip Flop using behavioral modeling

6.4.3 Synchronous JK Flip Flop using structural modeling

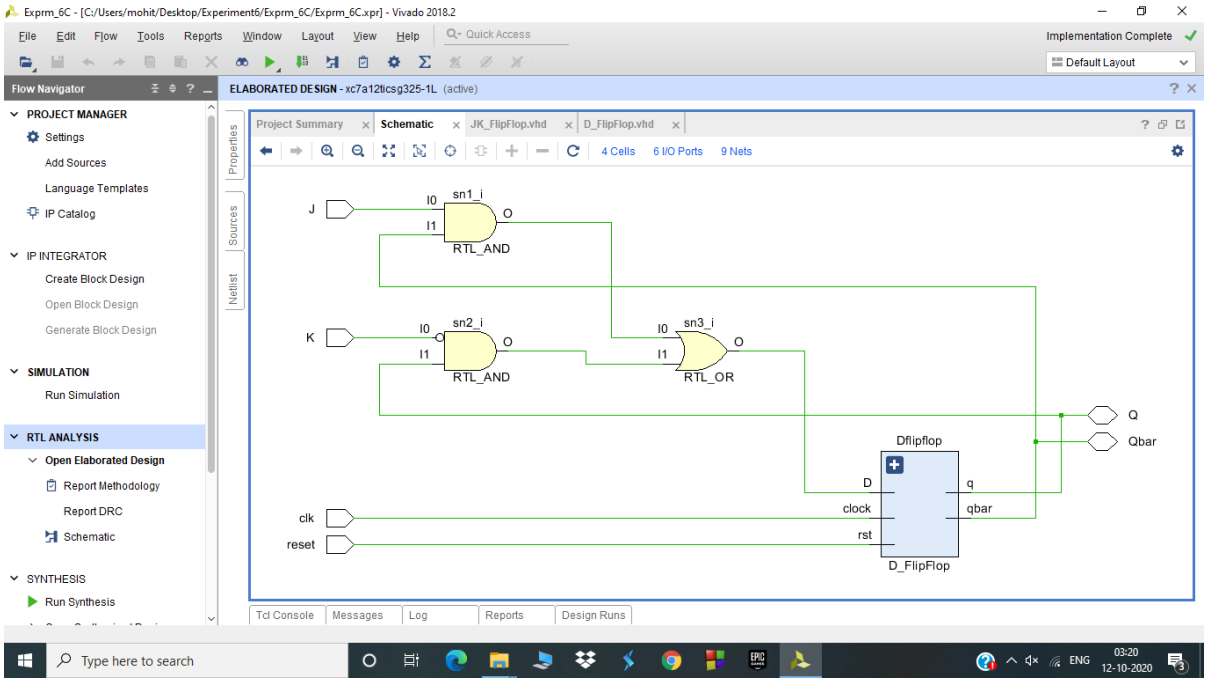

Figure 6.12 Schematic of the synchronous JK Flip Flop using structural modeling

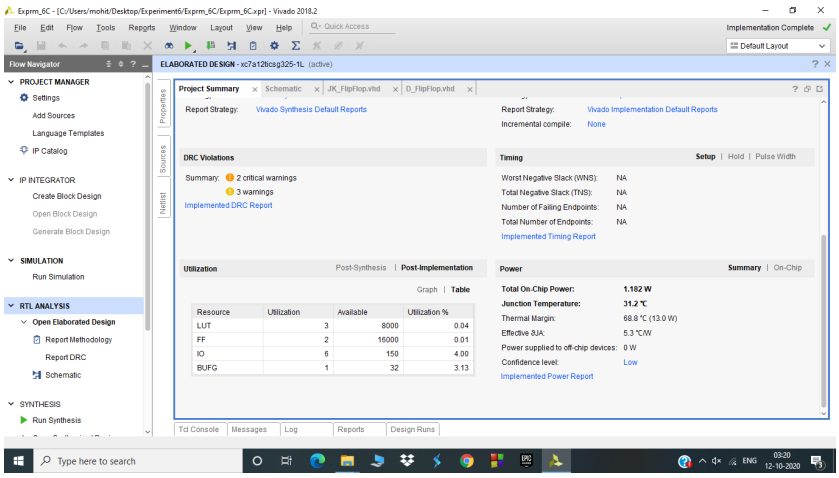

Figure 6.13 Project Summary of the synchronous JK Flip Flop using structural modeling

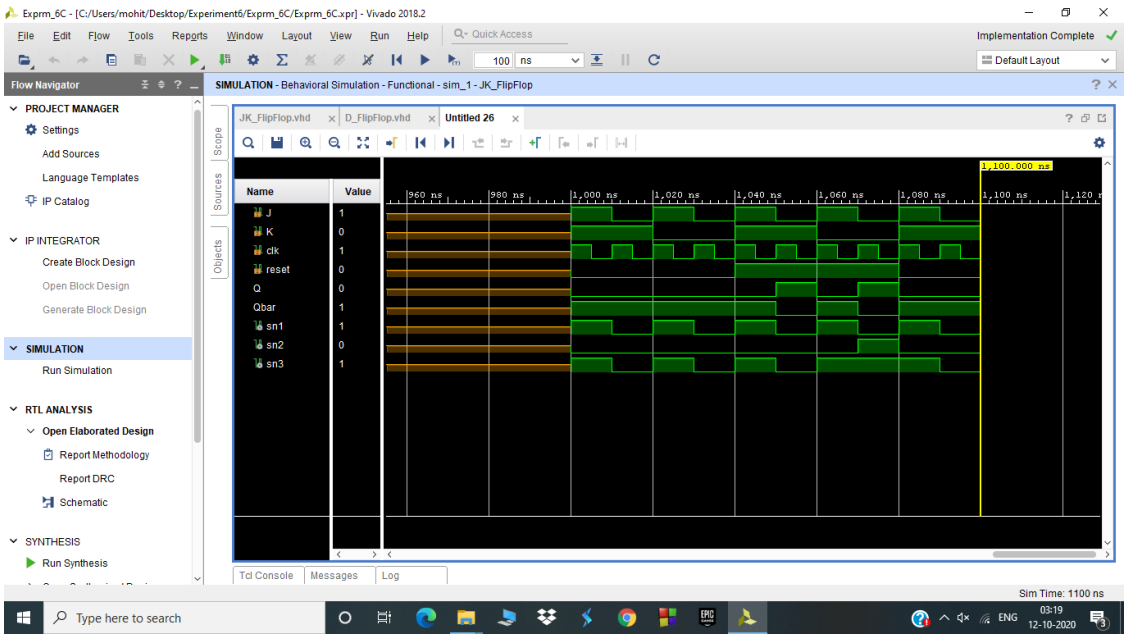

Figure 6.14 Simulation of the synchronous JK Flip Flop using structural modeling

6.4.4 Synchronous RS Flip Flop using behavioral modeling

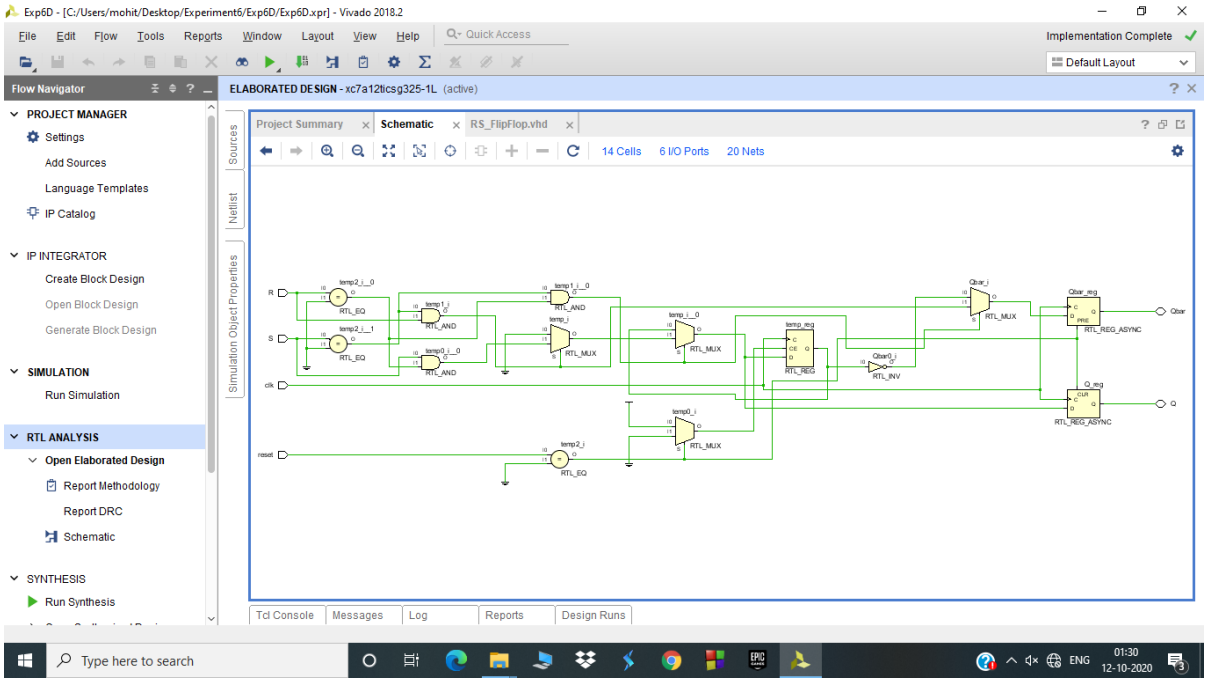

Figure 6.15 Schematic of the synchronous RS Flip Flop using behavioral modeling

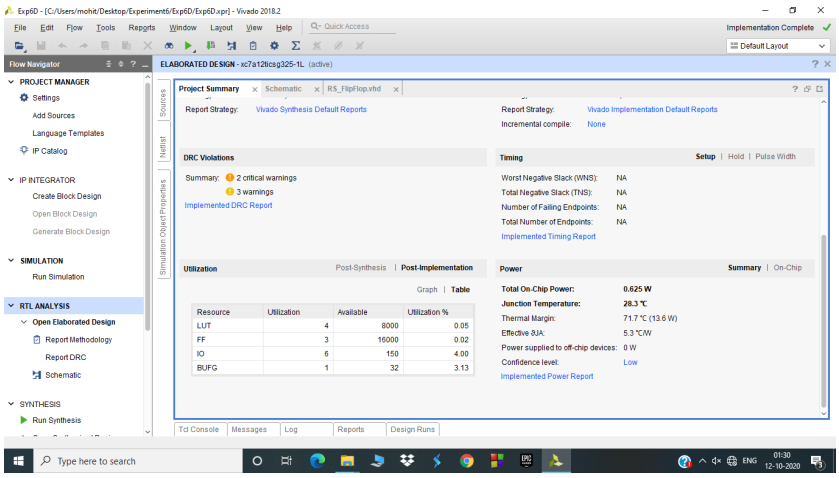

Figure 6.16 Project Summary of the synchronous RS Flip Flop using behavioral modeling

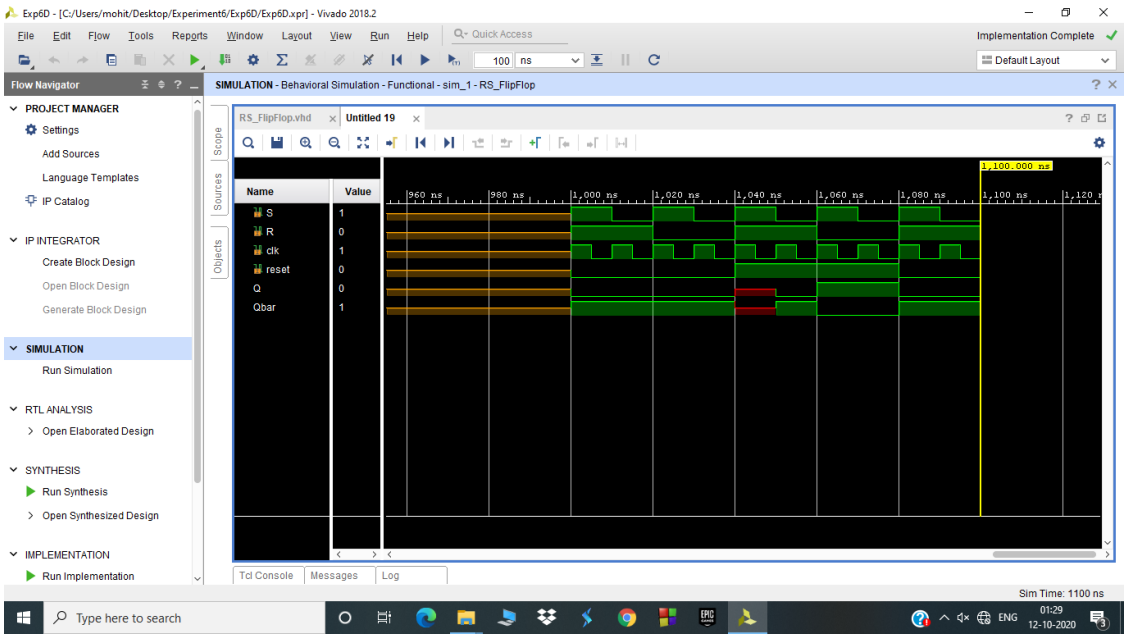

Figure 6.17 Simulation of the synchronous RS Flip Flop using behavioral modeling

6.4.5 Synchronous T Flip Flop using behavioral modeling

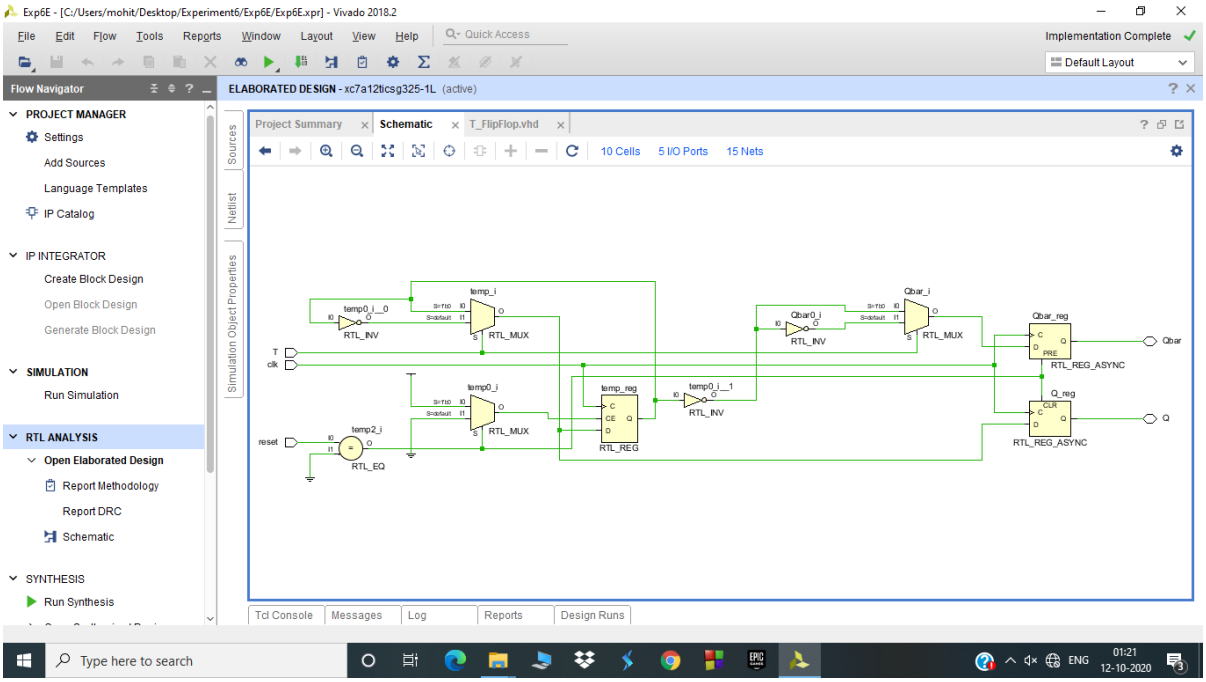

Figure 6.18 Schematic of the synchronous T Flip Flop using behavioral modeling

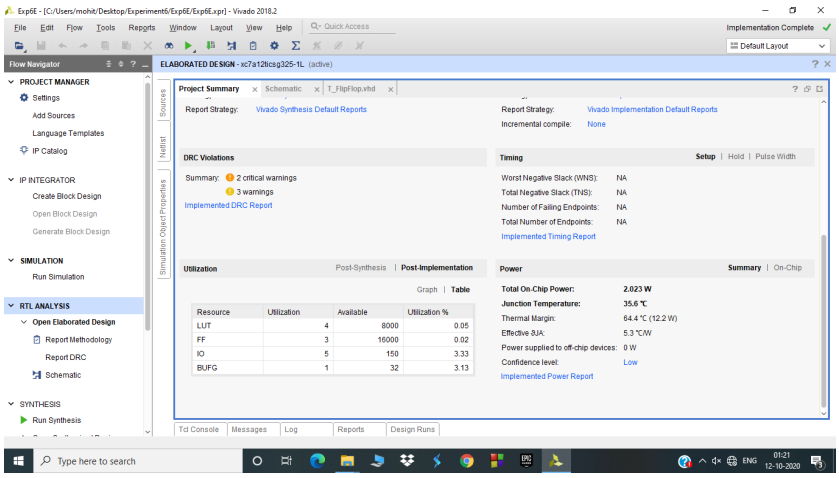

Figure 6.19 Project Summary of the synchronous T Flip Flop using behavioral modeling

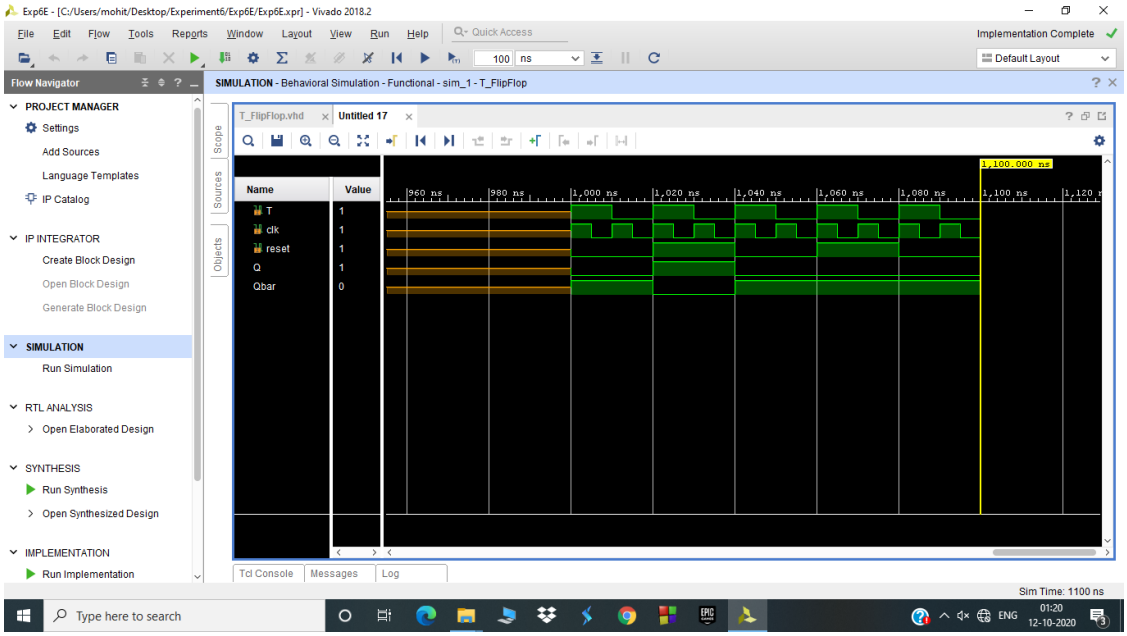

Figure 6.20 Simulation of the synchronous T Flip Flop using behavioral modeling

6.5 Summary

Tabular comparison of all the codes in terms of area and power usage.

Name of the Entity	No. of LUT used	Total On chip Power
synchronous D latch using dataflow modeling	1	0.694W
synchronous D Flip Flop using behavioral modeling	2	0.561W
synchronous JK Flip Flop using structural modeling	3	1.182W
synchronous RS Flip Flop using behavioral modeling	4	0.625W
synchronous T Flip Flop using behavioral modeling	4	2.023W

Table 6.6 comparision of Area and power requirements for different types of sequential circuits (Latches and Flip Flops).

Chapter *7*

Experiment - 7

7.1 Name of the Experiment

Shift Register Design using VHDL

7.2 Theory

7.2.1 Register

In digital electronics, Flip-Flop is a 1 bit memory cell which can be used for storing the digital data. To increase the storage capacity in terms of number of bits, we have to use a group of flip-flops. Such a group of flip-flops is known as **Register**. The **n-bit register** means there are **n** number of flip-flops and it is capable of storing **n-bit word**.

7.2.2 Shift Register

The binary data in a register can be moved within the register from one flip-flop to another. The registers that allow such data transfers are called **Shift registers**. The bits stored in such registers can be made to move within the registers and in/out of the registers by applying clock pulses. The registers which shift the bits to the left are called **Shift left registers** and the registers which shift the bits to the right are called **Shift right registers**. Shift registers are basically of 4 types :

- Serial In Serial Out shift register (SISO)
- Serial In Parallel Out shift register (SIPO)
- Parallel In Serial Out Shift register (PISO)
- Parallel In Parallel Out Shift register (PIPO)

7.2.2.1 SISO Shift register

The shift register which allows serial input (one bit after the other through a single data line) and produces a serial output is called Serial-In Serial-Out Shift register. The circuit consists of 4 **D Flip-Flops** connected in serial manner and all these flip flops are synchronous with each other due to same clock signal being applied.

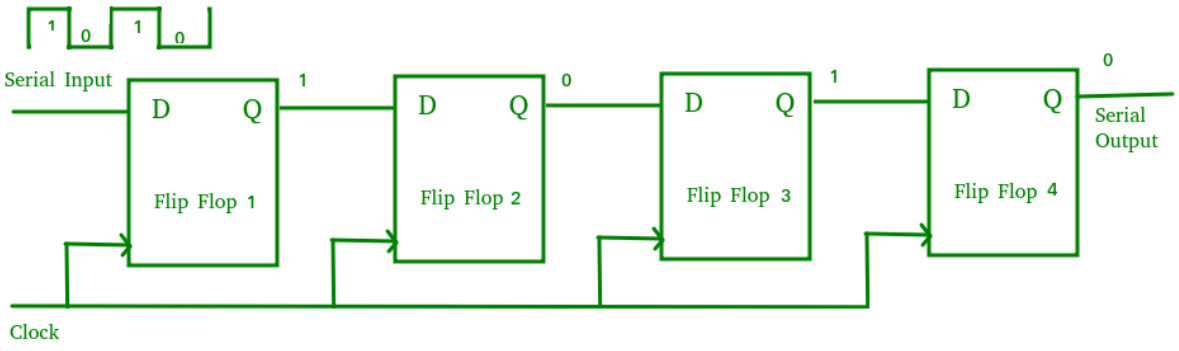

Figure 7.1 Serial-In Serial-Out shift register

7.2.2.2 SIPO Shift register

The shift register which allows serial input and produces a parallel output is known as Serial-In Parallel-Out shift register. The circuit consists of 4 D flip flops and clear (CLR) signal is connected in addition to clock signal to all the 4 flip flops in order to **RESET** them. All the flip flops are synchronous due to same clock signal being applied.

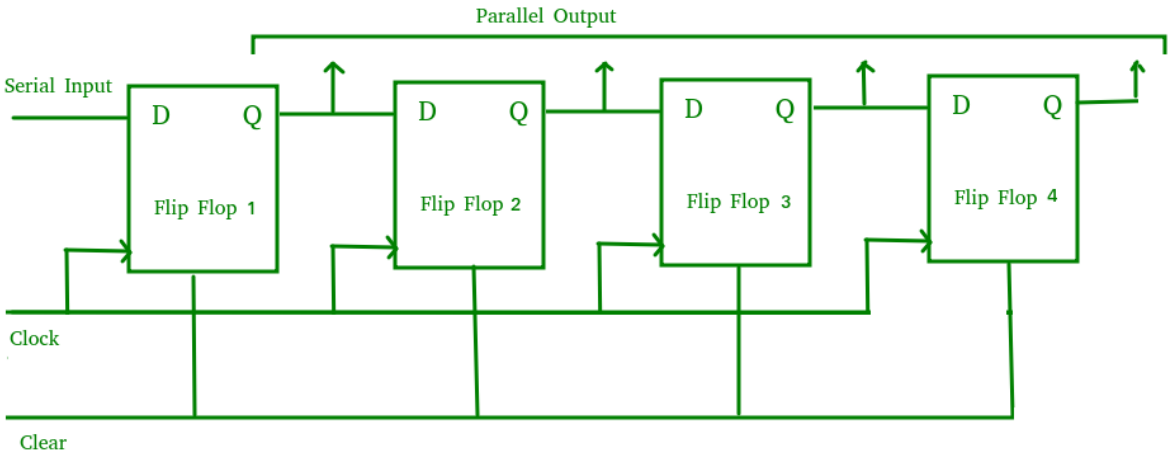

Figure 7.2 Serial-In Parallel-Out shift register

7.2.2.3 PISO Shift register

The shift register which allows parallel input and produces a serial output is called Parallel-In Serial-Out shift register. The circuit consists of 4 D Flip- flops which are connected. The clock input is directly connected to all the flip flops but the input data is connected individually to each flip flop through a multiplexer at the input of every flip flop. All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.

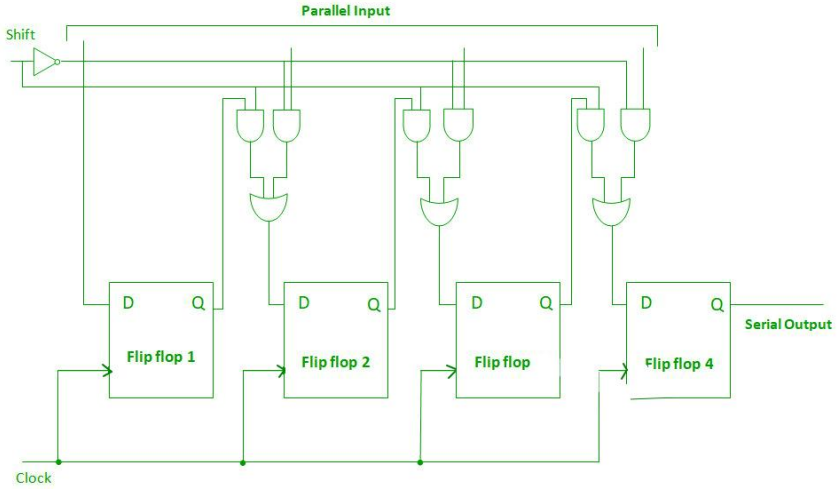

Figure 7.3 Parallel-In Serial-Out shift register

7.2.2.4 PIPO Shift register

The shift register that allows parallel input and produces a parallel output is known as Parallel-in Parallel-out shift register. It consists of 4 D flip-flops connected to each other and a Clear(CLR) signal and all these flip-flops are synchronous since same clock signal is being applied to all 4 flip-flops.

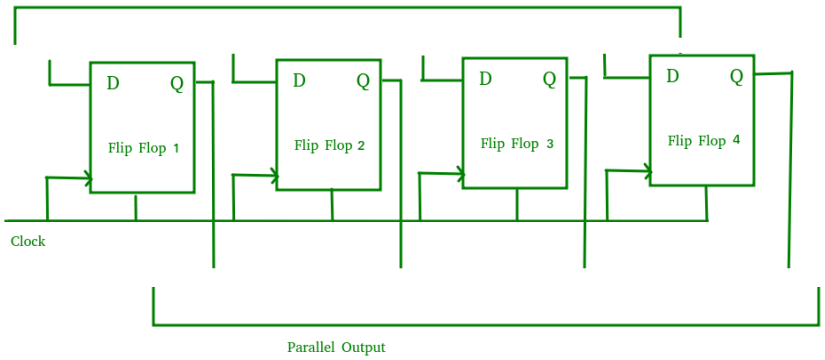

Figure 7.4 Parallel-In Parallel-Out shift register

7.3 Coding Techniques used

1. ***8-bit shift left SISO register with negative edge clock and clock enable***

   In this modeling of SISO register, I have used structural style and used 4 **D Flip-Flops** as modules to construct the 8-bit shift left register and clock signal is being applied with negative edge (falling-edge) . I have given the input serially (Sin) and also taken the output serially (Sout). Same clock signal is being applied to all the flip-flops so that they are synchronous.

2. ***8-bit shift left SISO register with negative edge clock, clock enable and an extra RESET input signal***

   In this modeling of SISO register, I have used structural style and used 4 **D Flip-Flops** as modules to construct the register and overall three input signals are being given - Sin, clk and reset. The function of the **reset** signal is to store 0 in all the flip-flops when reset is 1 and the same function of register continues if reset is 0. I have given the input (Sin) serially and taken the output serially through Sout. Same clock signal is being applied to all the flip-flops so that they all are synchronous.

3. ***8-bit shift left SIPO register with positive edge clock***

   In this modeling of SIPO register, I have used structural style and used 4 **D Flip-Flops** as modules to construct the register. Overall two input signals are being given - Sin and clk. Same clock signal is applied to all the flip -flops so that they are synchronous. Here clock signal is being applied with positive edge (Rising edge). I have given the input serially (Sin) and taken the output Parallel through a **8 bit vector** construct of VHDL.

4. ***16-bit shift left SIPO register with positive edge clock***

   In this modeling of SIPO register, I have used structural style and first constructed a 8-bit shift left register using 4 D flip-flops and then used **2** 8-bit shift left registers to construct the final 16-bit shift left register. Overall two input signals are being applied Sin and clk. same clock signal is being applied to both the 8-bit shift left registers so that they are synchronous . I have given the input serially (Sin) to the first 8-bit shift left register and taken the output as the last 8 bits of the 16-bit output vector, then I have given the Sout(8) component of the vector to the second 8-bit shift left register and taken the output for the first 8 bits of the 16-bit output vector.

7.4 Simulation and Results

7.4.1 8-bit shift left SISO register with negative edge clock and clock enable

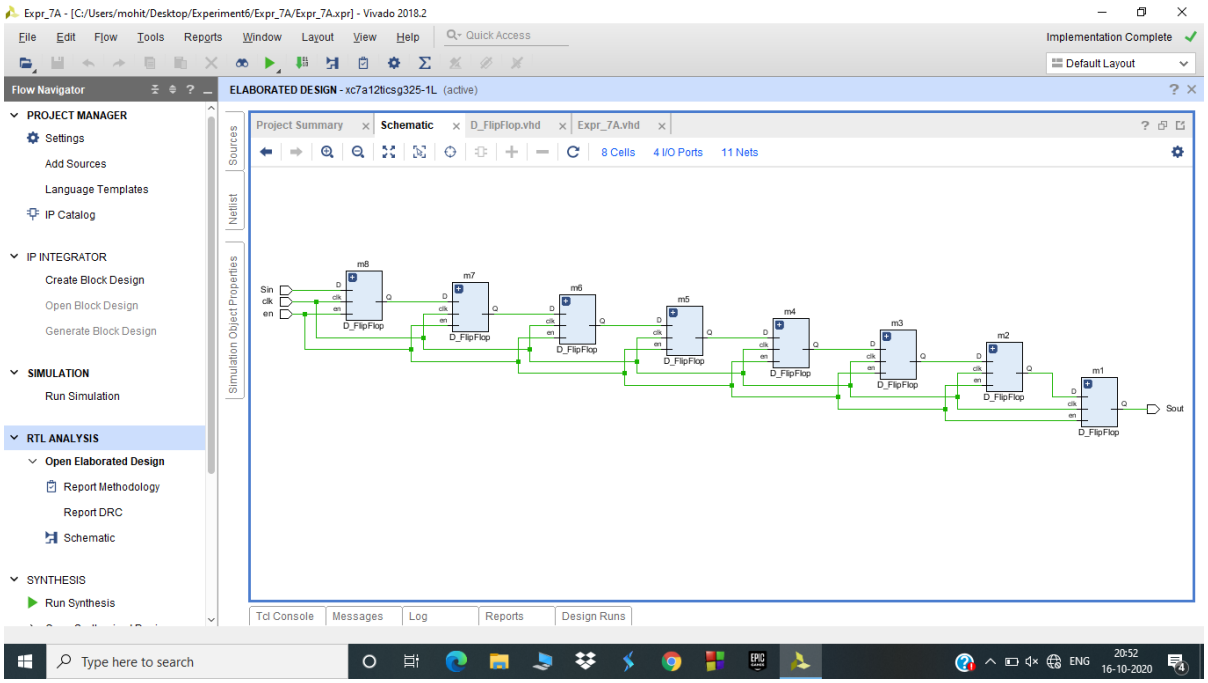

Figure 7.5 Schematic of the 8-bit shift left SISO register with negative edge clock and clock enable

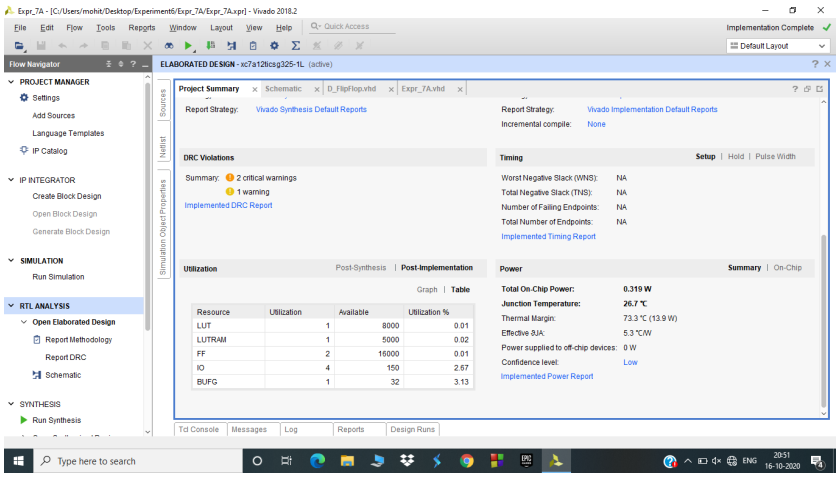

Figure 7.6 Project Summary of the 8-bit shift left SISO register with negative edge clock and clock enable

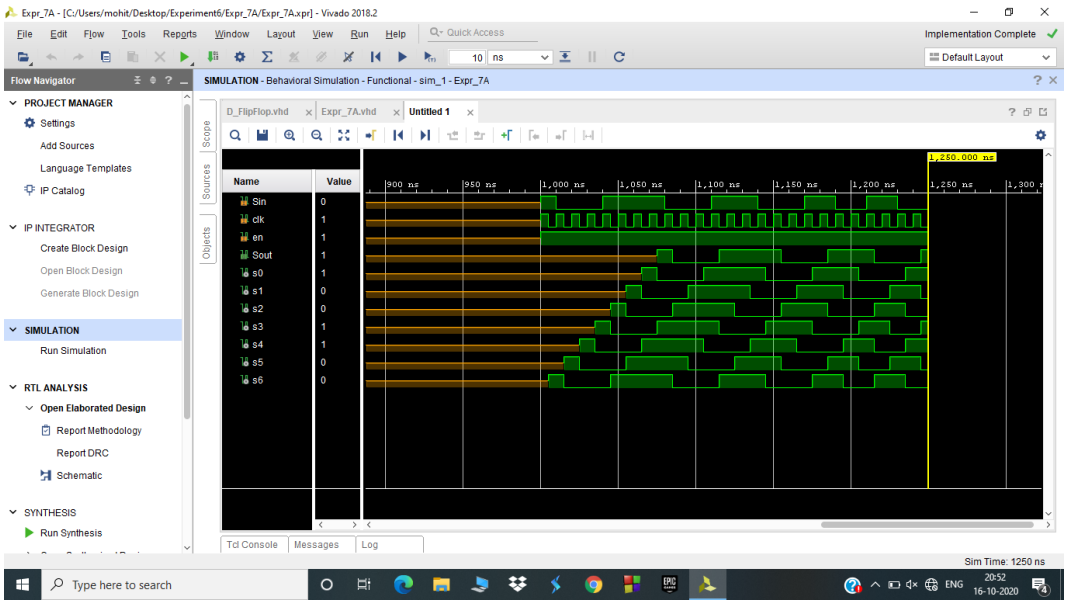

Figure 7.7 Simulation of the 8-bit shift left SISO register with negative edge clock and clock enable

7.4.2 8-bit shift left SISO register with negative edge clock, clock enable and an extra RESET input signal

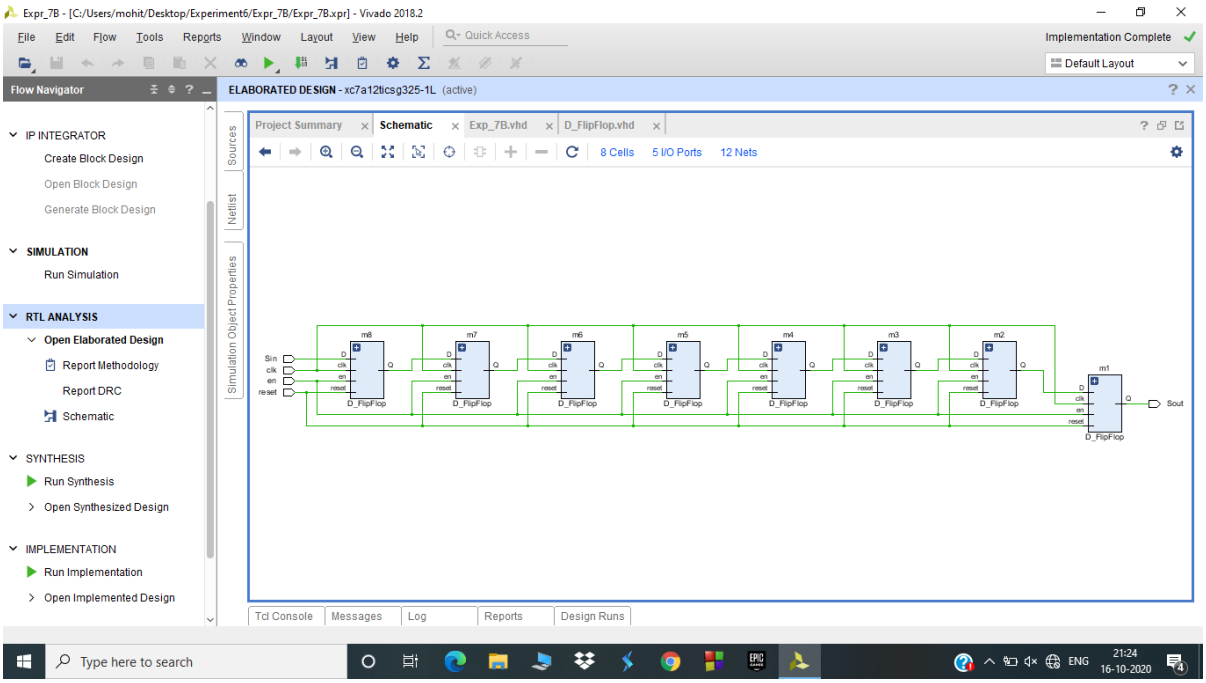

Figure 7.8 Schematic of the 8-bit shift left SISO register with negative edge clock, clock enable and an extra RESET input signal

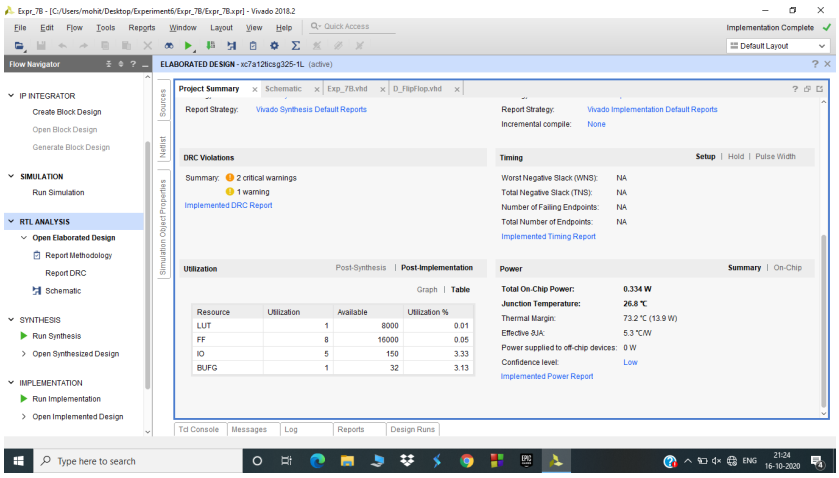

Figure 7.9 Project Summary of the 8-bit shift left SISO register with negative edge clock, clock enable and an extra RESET input signal

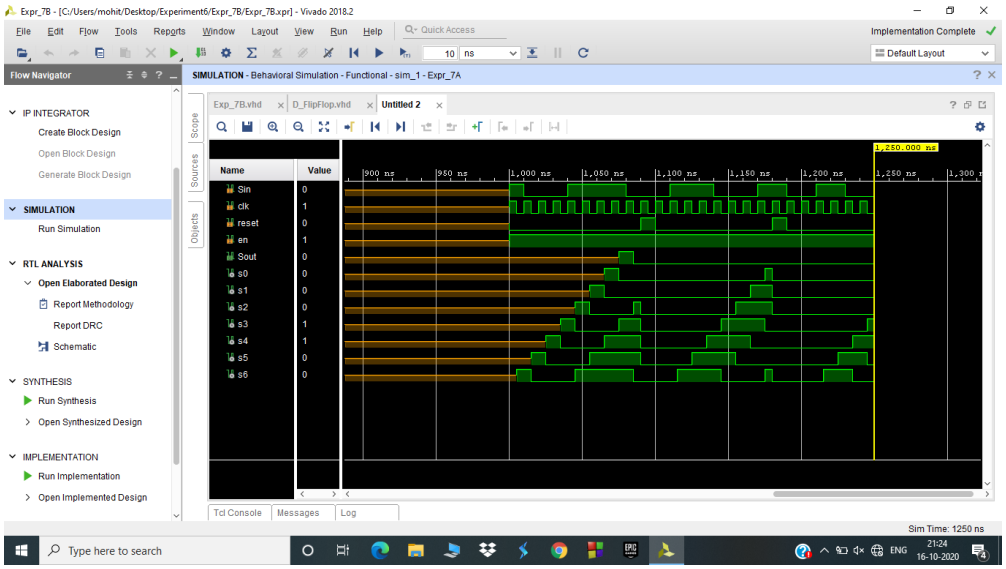

Figure 7.10 Simulation of the 8-bit shift left SISO register with negative edge clock, clock enable and an extra RESET input signal

7.4.3 8-bit shift left SIPO register with positive edge clock

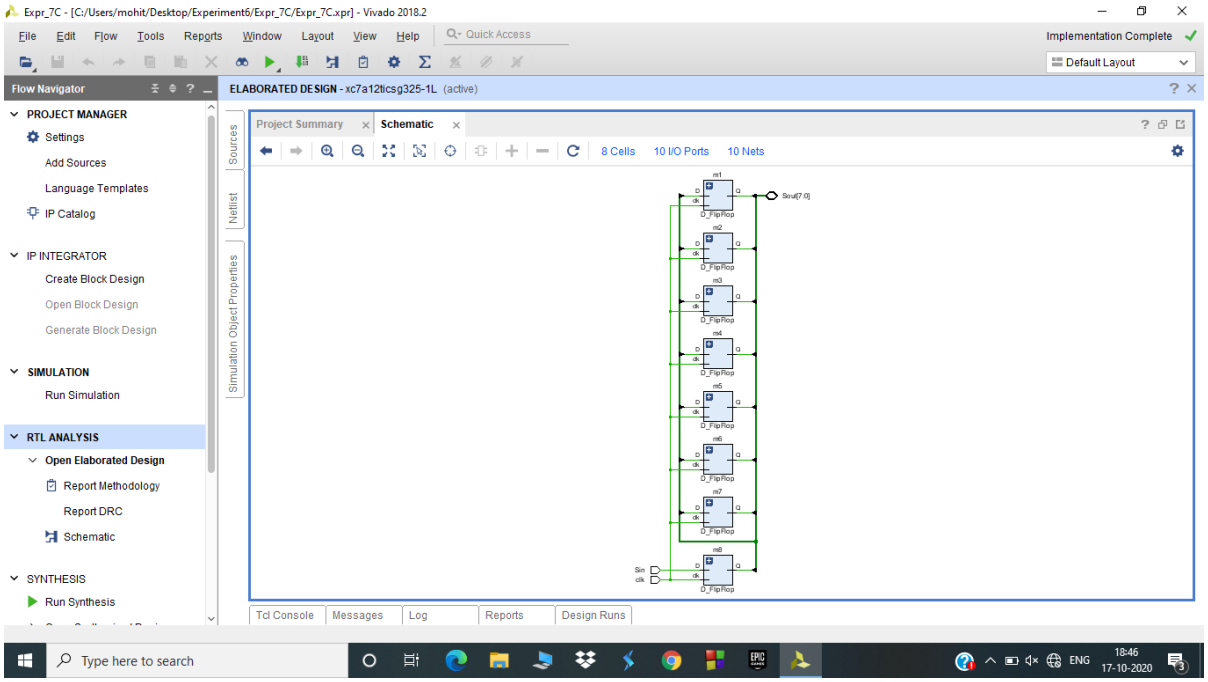

Figure 7.11 Schematic of the 8-bit shift left SIPO register with positive edge clock

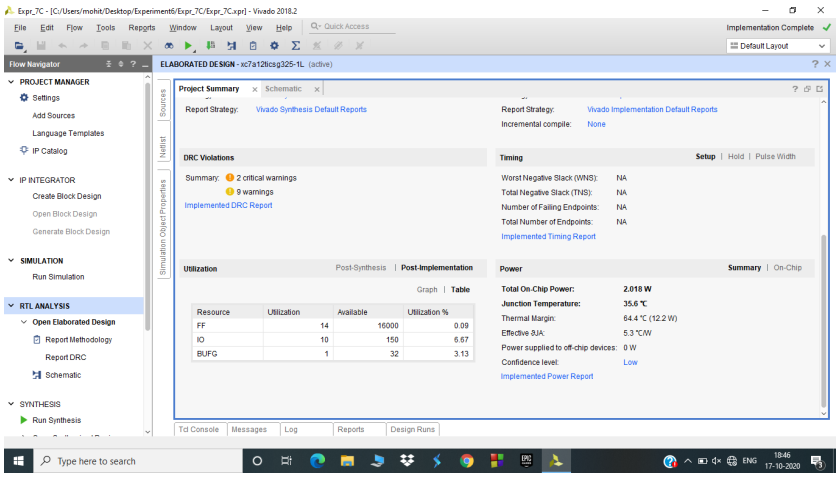

Figure 7.12 Project Summary of the 8-bit shift left SIPO register with positive edge clock

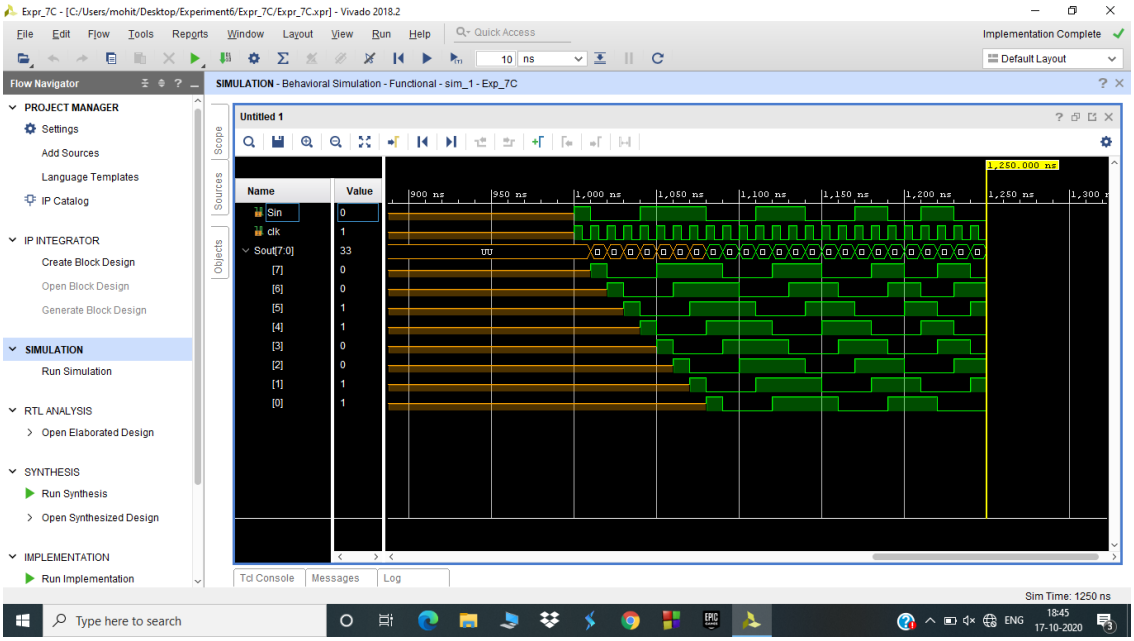

Figure 7.13 Simulation of the 8-bit shift left SIPO register with positive edge clock

7.4.4 16-bit shift left SIPO register with positive edge clock

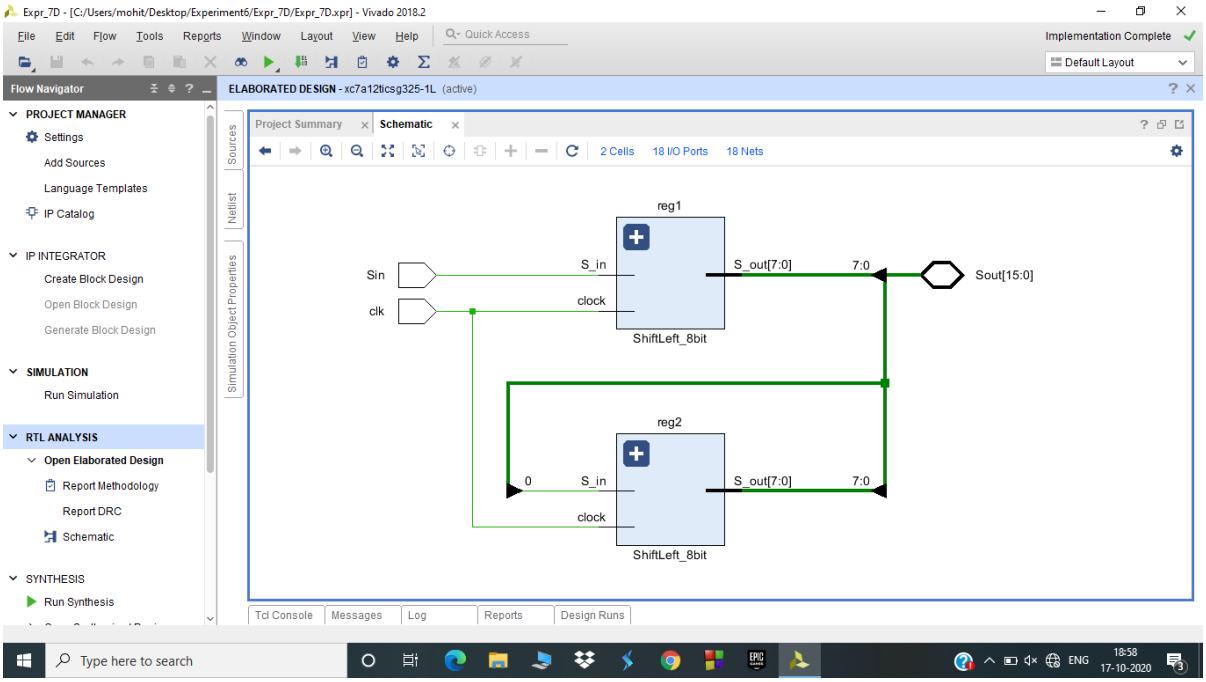

Figure 7.14 Schematic of the 16-bit shift left SIPO register with positive edge clock

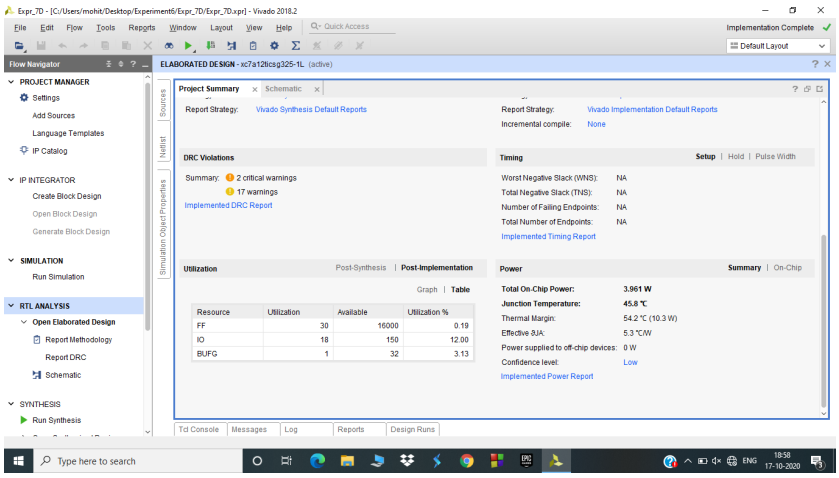

Figure 7.15 Project Summary of the 16-bit shift left SIPO register with positive edge clock

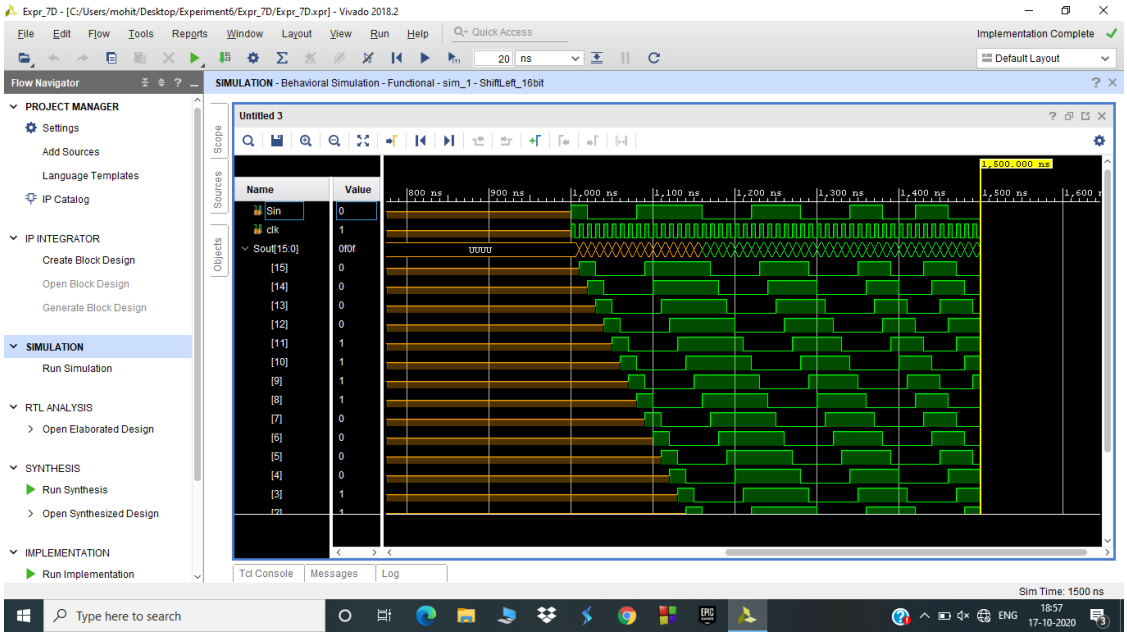

Figure 7.16 Simulation of the 16-bit shift left SIPO register with positive edge clock

7.5 Summary

Tabular comparison of all the codes in terms of area and power usage.

Name of the Entity	No. of LUT used	Total On chip Power
8-bit shift left SISO register with negative edge clock and clock enable	1	0.319W
8-bit shift left SISO register with negative edge clock with RESET signal	1	0.334W
8-bit shift left SIPO register with positive edge clock	-	2.018W
16-bit shift left SIPO register with positive edge clock	-	3.961W

Table 7.1 comparision of Area and power requirements for different types of shift registers.

Chapter *8*

Experiment - 8

8.1 Name of the Experiment

Sequential system design using state Machines

8.2 Theory

8.2.1 Finite State machine (FSM)

A finite state machine (FSM) or finite state automata is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another state in response to inputs and the change from one state to another state is called **Transition**. There are two types of FSM :

- Moore machine
- Mealey machine

8.2.2 Moore machine

A moore machine is a finite state machine whose output values are determined only by its current state. The moore machine is named after **Edward F. Moore**. Using moore machine, a process requires more states than mealey implementation. Moore machines are easy to design.

8.2.3 Mealey machine

a Mealy machine is a finite-state machine whose output values are determined both by its current state and the current inputs.A Mealy machine is a deterministic finite-state transducer: for each state and input, at most one transition is possible. It requires lesser states for implementation of a process using mealey machine than that by moore machine. Mealey machines are difficult to design.

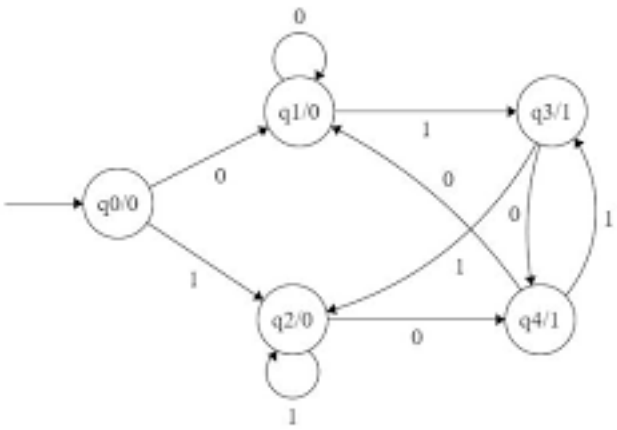

Figure 8.1 Moore machine state diagram

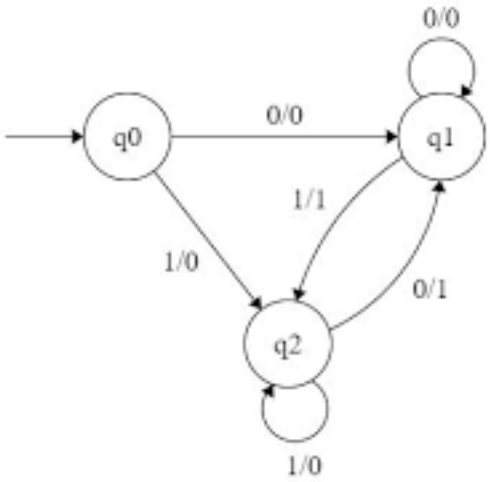

Figure 8.2 Mealey machine state diagram

8.3 Coding Techniques used

1. ***Mod 4 up/down counters using state machines***

   In this modeling of mod 4 up/down counter , I have used moore machine to design this process. In the design , there are 4 states s0,s1,s2 and s3. The output corresponding to various states s0,s1,s2 and s3 are 0,1,2 and 3 using **VHDL vector construct**. In this , there are 3 inputs clk,reset and count and 1 output C which is a vector. clk is given the same settings as before with leading edge 1 and trailing edge 0 , function of reset is to reset the current count to 0. The function of the **count** input signal is that it acts as a **up counter** for count = 0 and as a **down counter** when count = 1. overall behavioral modeling of VHDL is used using case statements and if-else constructs.

2. ***Pattern detector of 1101 using state machines***

   In this modeling of pattern detector, I have used moore machine to design . I have taken the case for **non-overlapping** and designed the machine. In this modeling , there are 5 states s0,s1,s2,s3 and s4. In this machine, first four states (s0,s1,s2 and s3) have output (z)= 0 and last state (s4) has output =1. In this modeling, there are 3 inputs (clk,reset and x) . clk is given the same settings as before with leading edge 1 and trailing edge 0. reset is used to control the machine, that is machine works when reset =0. x is the input signal in which input is given serially , one after the another . The output is equal to 1 if x="1101" and it is equal to 0 otherwise. overall behavioral modeling of VHDL is used using case statements and if-else constructs.

8.4 Simulation and Results

8.4.1 Mod 4 up/down counters using state machines

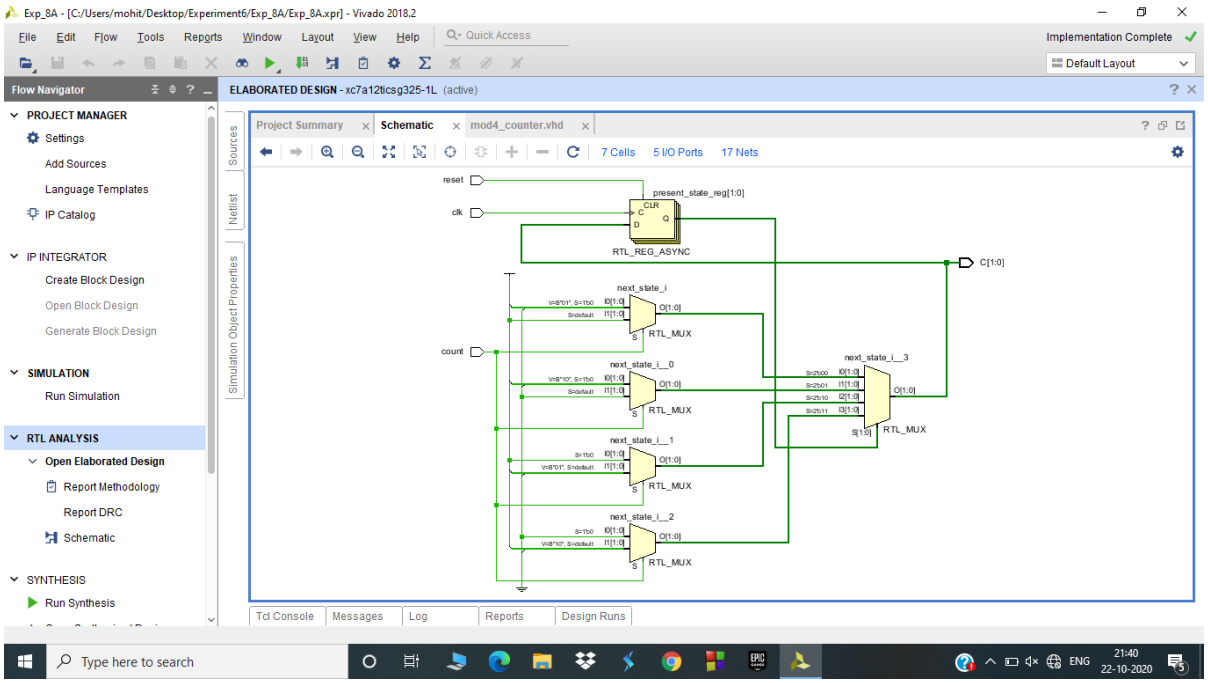

Figure 8.3 Schematic of the Mod 4 up/down counters

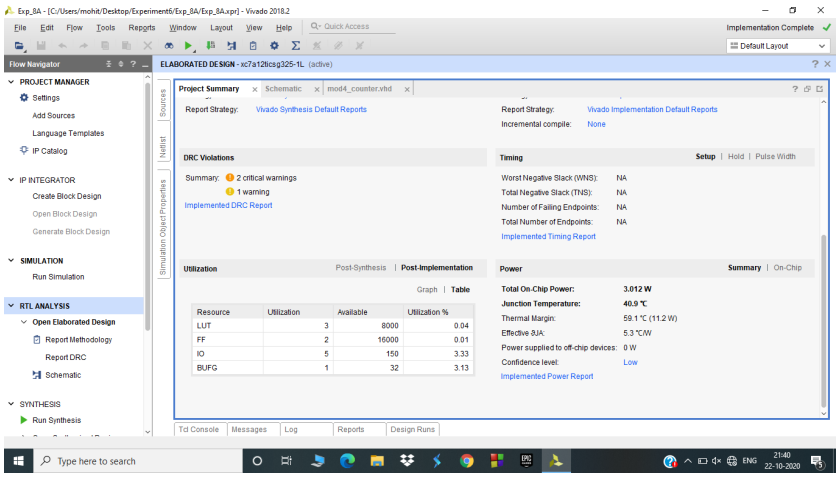

Figure 8.4 Project Summary of the Mod 4 up/down counters

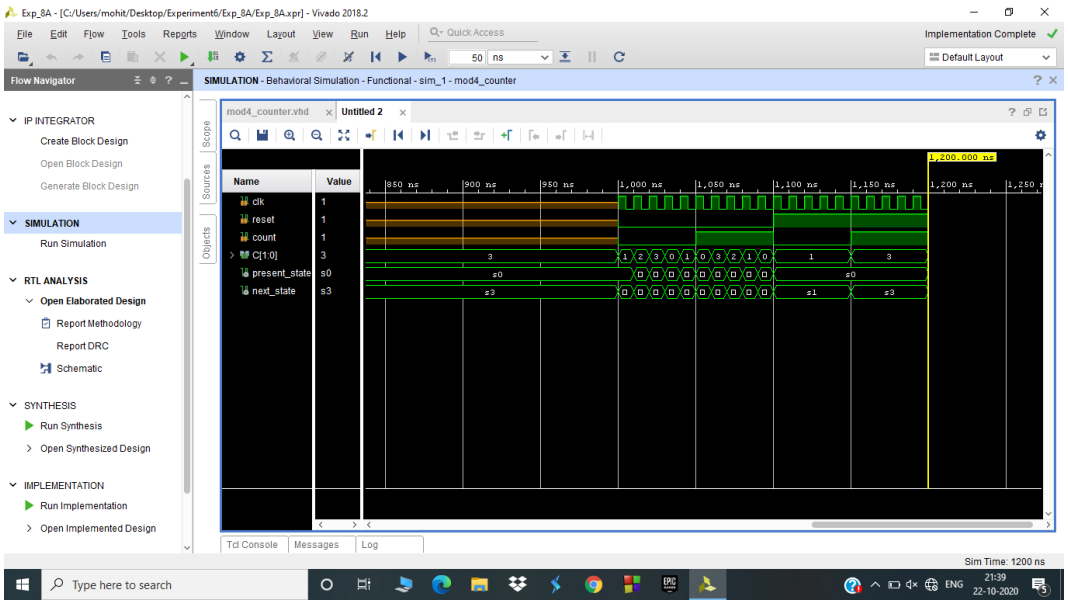

Figure 8.5 Simulation of the Mod 4 up/down counters

8.4.2 Pattern detector of 1101 using state machines

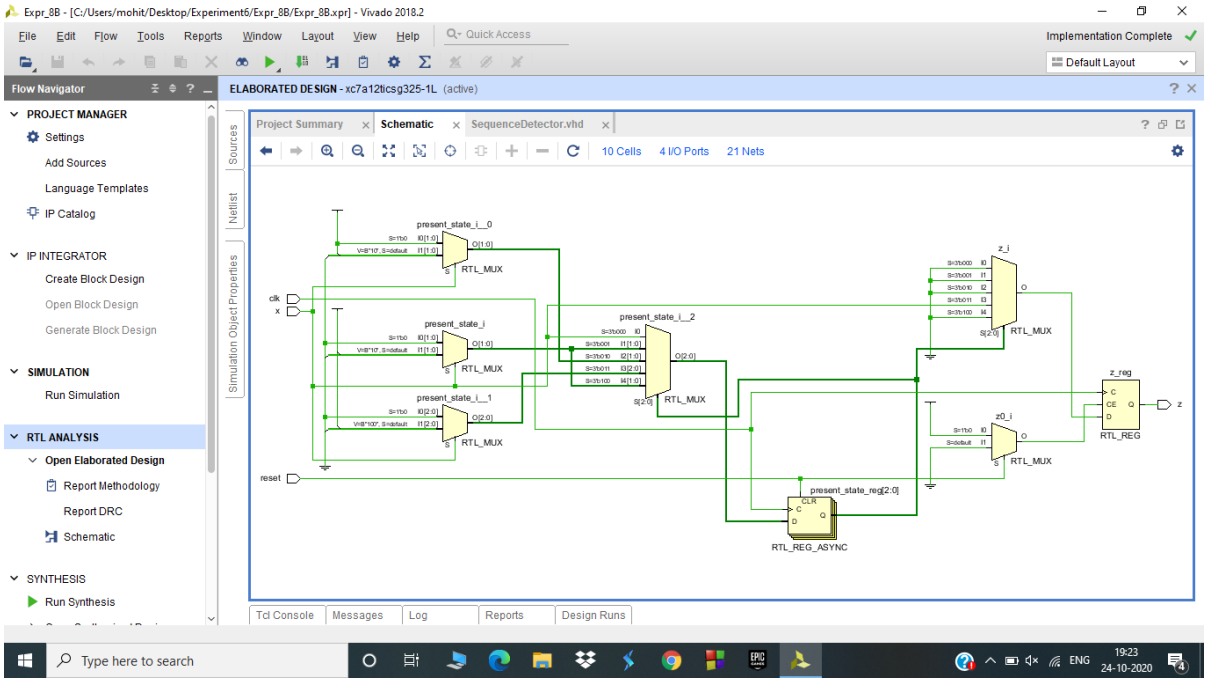

Figure 8.6 Schematic of the Pattern detector of 1101

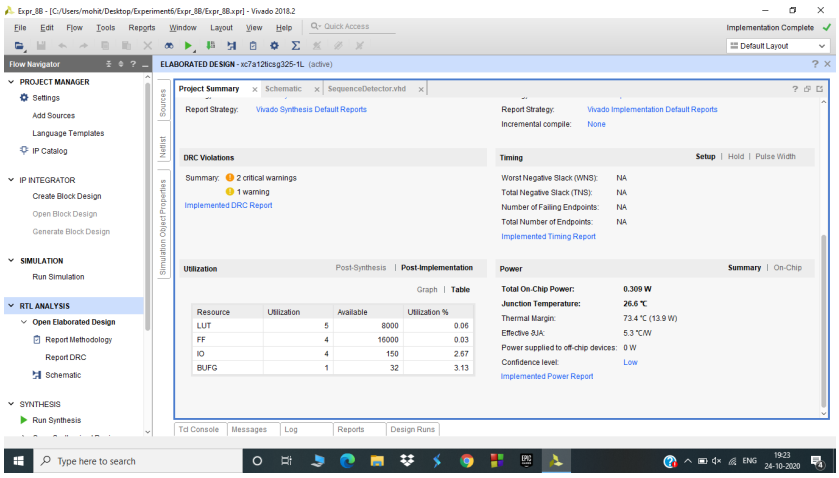

Figure 8.7 Project Summary of the Pattern detector of 1101

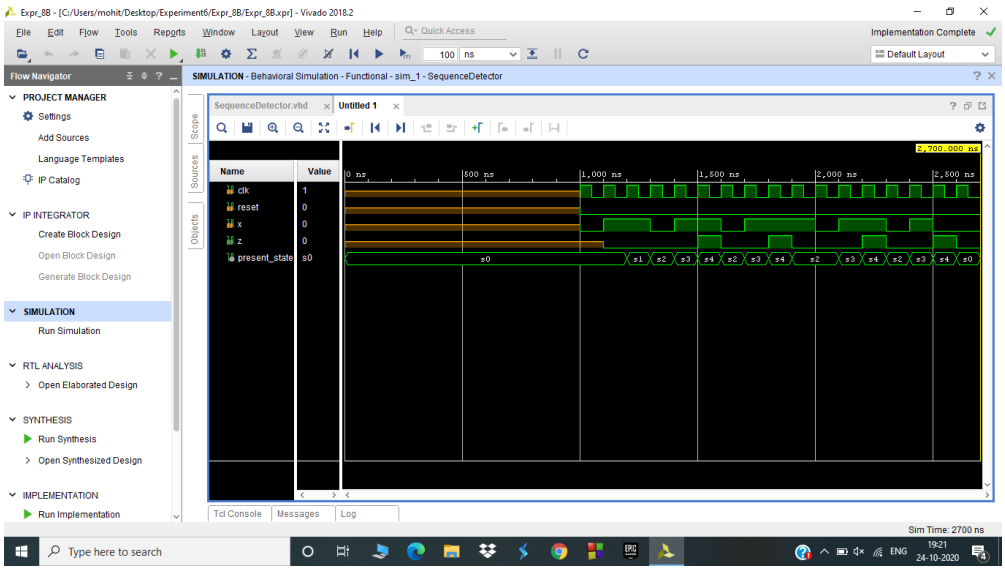

Figure 8.8 Simulation of the Pattern detector of 1101

8.5 Summary

Tabular comparison of all the codes in terms of area and power usage.

Name of the Entity	No. of LUT used	Total On chip Power
Mod 4 up/down counters using state machines	3	3.012W
Pattern detector of 1101 using state machines	5	0.309W

Table 8.1 comparision of Area and power requirements for Mod 4 up/down counters and Pattern detector.

Chapter *9*

Experiment - 9

9.1 Name of the Experiment

Counters and its applications

9.2 Theory

9.2.1 Counters

In digital logic and computing, A **Counter** is a device which stores (and sometimes display) the number of times a particular event or process has occurred, often in relationship to a clock signal. Counters are used in digital electronics from counting purpose, they can count specific event happening in the circuit. For example , in UP counter , a counter increases count for every rising edge of the clock. They can also be designed with the help of flip-flops. Different types of counters are :

9.2.1.1 Asynchronous counters

In asynchronous counter we don't use universal clock, only first flip flop is driven by main clock and the clock input of rest of the following flip flop is driven by output of previous flip flops.

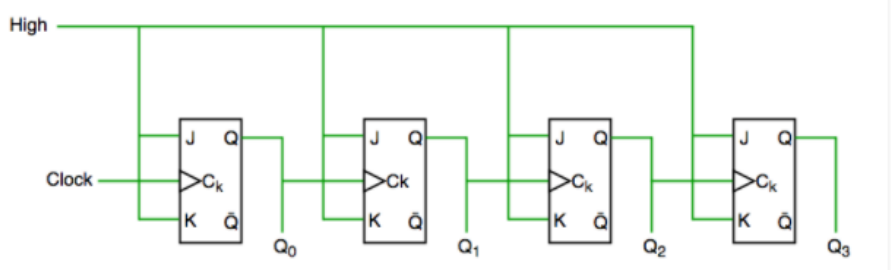

Figure 9.1 Asynchronous Counter

9.2.1.2 Synchronous counters

synchronous counter has one global clock which drives each flip flop so output changes in parallel. The one advantage of synchronous counter over asynchronous counter is, it can operate on higher frequency than asynchronous counter as it does not have cumulative delay because of same clock is given to each flip flop.

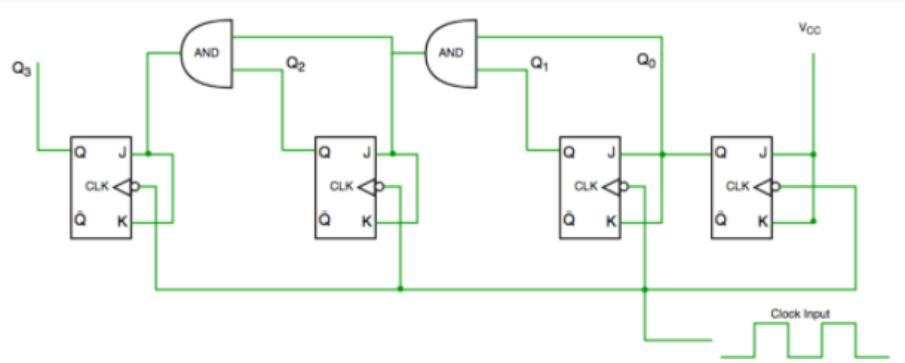

Figure 9.2 Synchronous Counter

9.2.1.3 Modulus Counters

Modulus Counters , or simply MOD counters, are defined based on the number of states that the counter will sequence through before returning back to its original value. For example, a 2-bit counter that counts from $(00)_2$ to $(11)_2$ in binary, that is 0 to 3 in decimal, has a modulus value of 4 (00 - 01 - 10 - 11, and return back to 00) so would therefore be called a modulo-4, or mod-4, counter. Note also that it has taken four clock pulses to get from 00 to 11. A **Mod-N** counter will require **n** number of flip-flops connected together to count a single data bit while providing 2^n different output states.

9.2.2 Application of counters

- Frequency counters
- Digital clock
- Time measurement
- A to D converter
- Frequency divider circuits
- Digital triangular pulse generator

9.3 Coding Techniques used

1. ***Implementing a modulo 16 counter having 4 bit parallel input***

   In this modeling of **mod-16** counter , I have used behavioral modeling style to design it. There are 4 inputs 4-bit vector I, clk, clr, cnt and load (ld) and 2 outputs S and terminal count (tc). I have applied the conditions given in the question for assigning different values to next state **s(t+1)** like for clr=1 , s(t+1) = 0 and for cnt=1,ld=0 s(t+1)=(s(t)+1)mod 16. I have used the **CASE constructs** of VHDL to implement the design. I have given clk the same values as given in the previous experiments with leading edge 1 and trailing edge 0 , and for the postive edge of the clock , the count happens. Overall **Concept of Finite State Machines** have been used in the design of this entity.

2. ***Implementing a 2 to 12 counter using mod-16 counter as a component***

   In this modeling of 2 to 12 counter, I have used the structural style of coding to implement this entity using **mod-16 counter** designed in the previous part as a structural module. I have given the same inputs and outputs declaration in the entity part of the code. I have used three signals new_input , new_load and new_output to implementing the combinational part of this design so as to restrict the count from 2 to 12 only and not going beyond. From the truth table of binary representations of numbers from 1 to 16, we observe that in a 4 bit vector (x) if the **OR operation** of x(3),x(2) and x(1) is 0 and for the **AND operation** of x(3) and x(2) is 1 , we can reset the count to 2 again and start all over again. This logic is used in the design of this entity. I have used process statements with **CASE constructs** of VHDL for designing this entity along with structural module of mod-16 counter. So this is a mixed style modeling. clk is given the same settings as before and for positive edge of clock , count happens. Idea of **Finite State machines** is also used here.

9.4 Simulation and Results

9.4.1 Modulo 16 counter using state machines

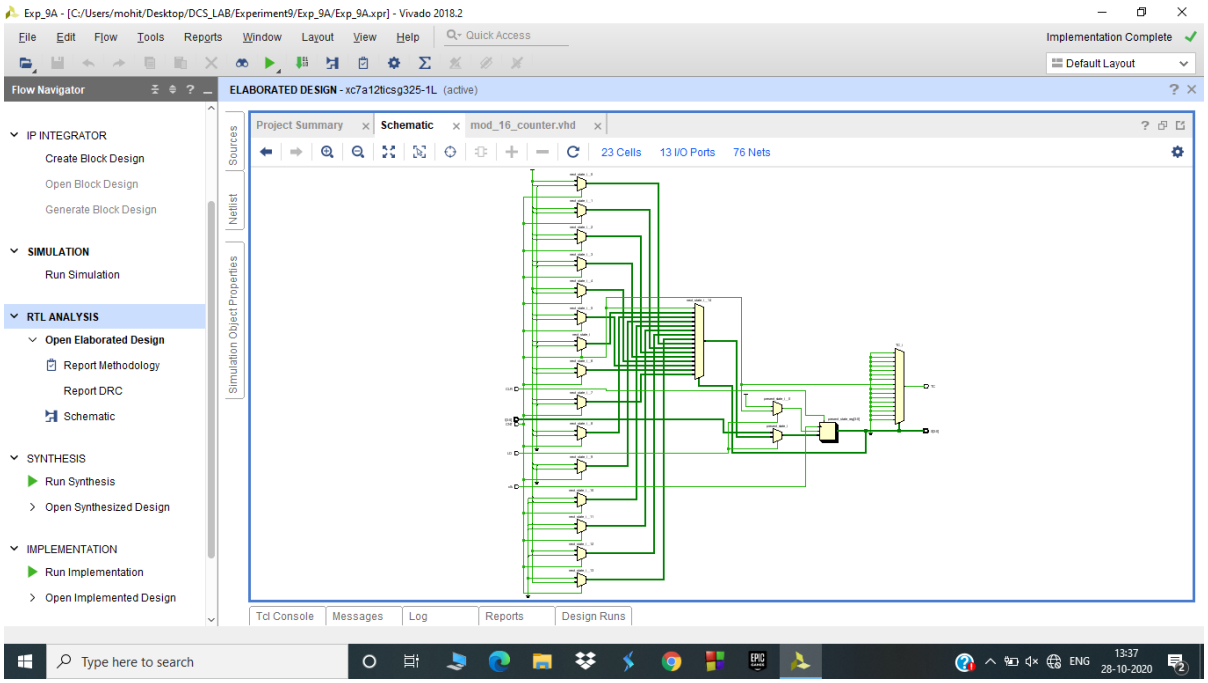

Figure 9.3 Schematic of the Modulo 16 counter

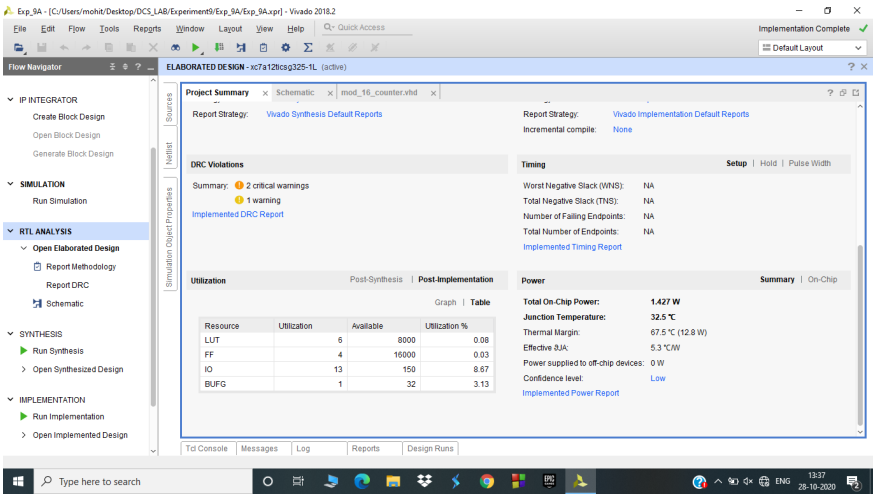

Figure 9.4 Project Summary of the Modulo 16 counter

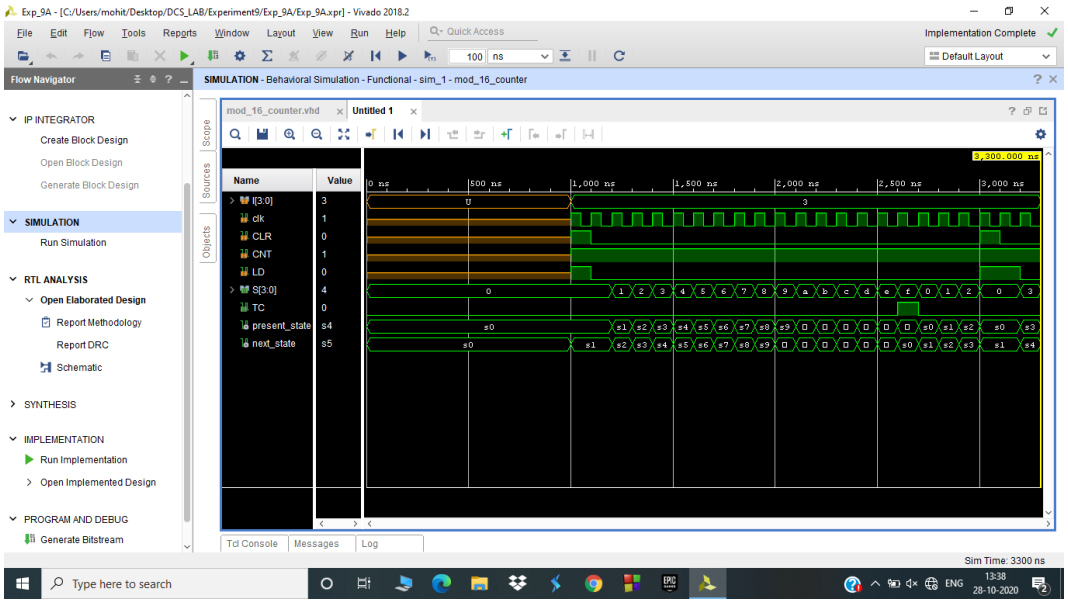

Figure 9.5 Simulation of the Modulo 16 counter

9.4.2 2 to 12 counter using modulo 16 counter as component

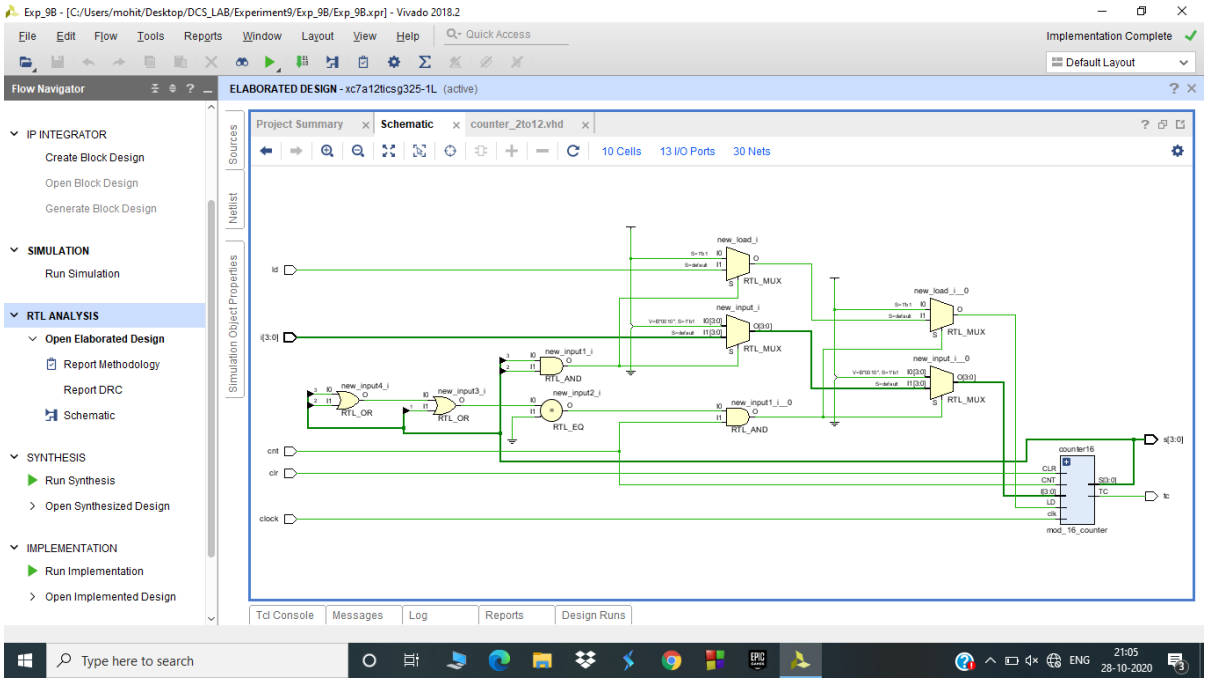

Figure 9.6 Schematic of the 2 to 12 counter

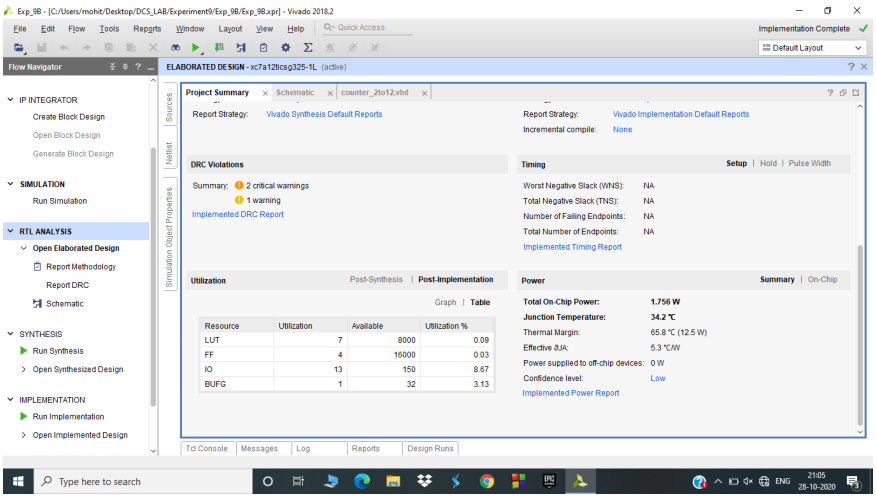

Figure 9.7 Project Summary of the 2 to 12 counter

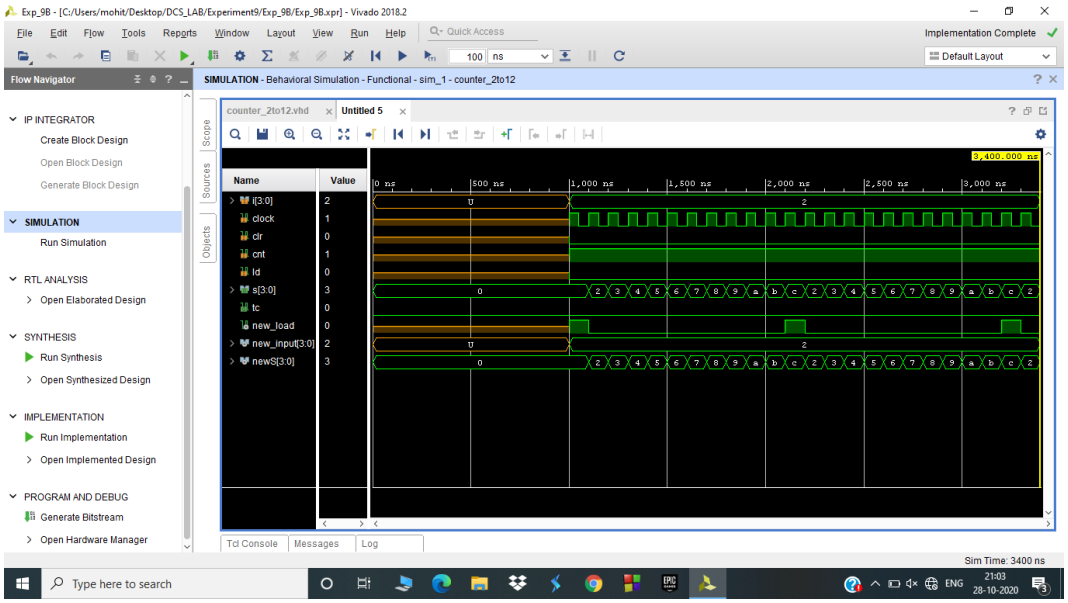

Figure 9.8 Simulation of the 2 to 12 counter

9.5 Summary

Tabular comparison of all the codes in terms of area and power usage.

Name of the Entity	No. of LUT used	Total On chip Power
Modulo 16 counter using state machines	6	1.427W
2 to 12 counter using mod 16 counter as module	7	1.756W

Table 9.1 comparision of Area and power requirements for Counters.

Chapter *10*

Experiment - 10

10.1 Name of the Experiment

Traffic Light Controller using state Machines

10.2 Theory

10.2.1 Applications of State machines

Finite-state machines are significant in many different areas, including electrical engineering, linguistics, computer science, philosophy, biology, mathematics, video game programming, and logic. Finite-state machines are a class of automata studied in automata theory and the theory of computation. They are also used for designing **controllers** and other important circuits.

10.2.2 Traffic Light controller

Traffic light control unit can be designed as a synchronous sequential machine with finite number of states. The machine is modelled with only six states and these states are chosen based on the traffic control algorithm.

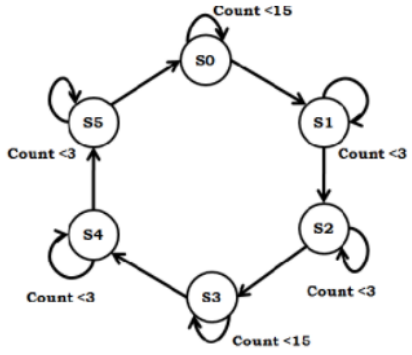

Figure 10.1 Mealey model of Traffic light controller

10.3 Coding Techniques used

1. ***Implementing Traffic Light Controller using State machines***

   In this modeling of Traffic Light controller, I have used the moore model of State machines to design it. I have used in total six states (s0,s1,s2,s3,s4 and s5) to design this entity. In the VHDL code, I have taken two inputs clk and clr. clk stands for clock signals and it is given the same settings as been given since the previous five experiments with leading edge 1 and trailing edge 0. clr stands for **clear signal** and it means that if clr=1 , then the state should initialize to the starting state that is s0. Basically, clr signal acts as a reset signal to reset to first state (s0). I have taken one 6 bit vector output "lights" that is used to store the state of traffic lights with first three bits reserved for the first traffic light and the last three bits for the second traffic light in **RYGRYG** order. I have used two signals st and count to design the controlling part , that is if count ¡ 15 then stay is first state and if greater , go ahead. Similar algorithm is used for other states as well. Overall behavioral modeling by using **CASE constructs** of VHDL along with state machine implementation are used.

10.4 Simulation and Results

10.4.1 Traffic Light controller using state machines

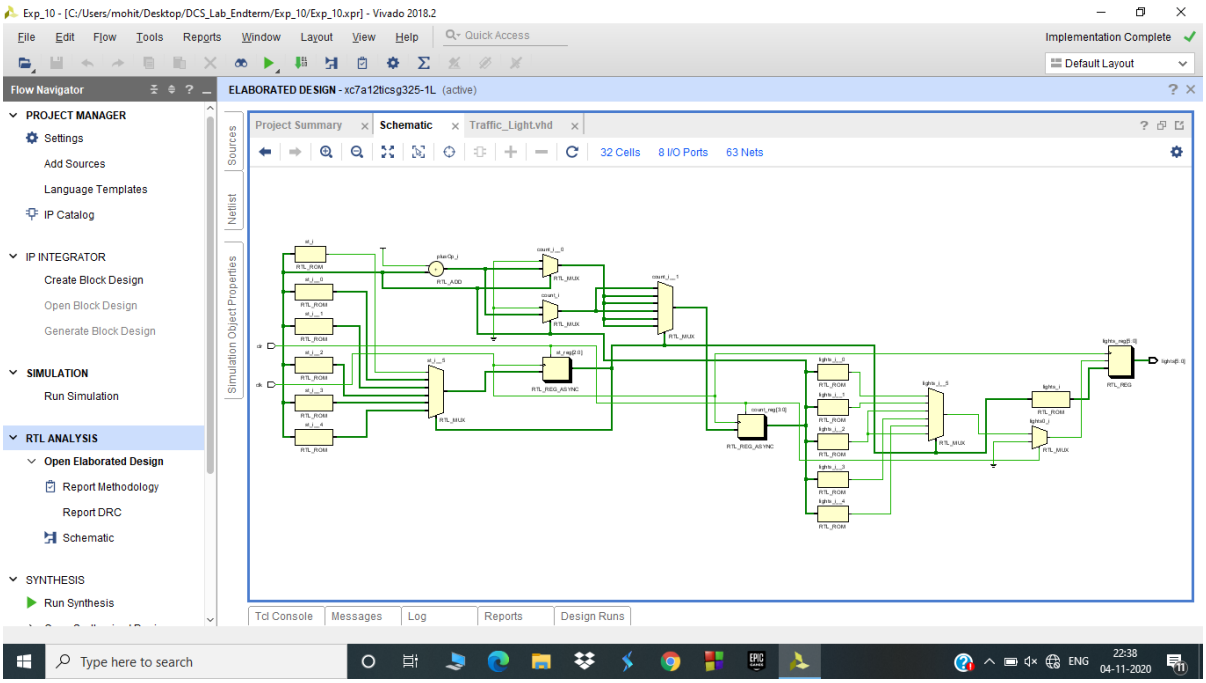

Figure 10.2 Schematic of the Traffic Light controller

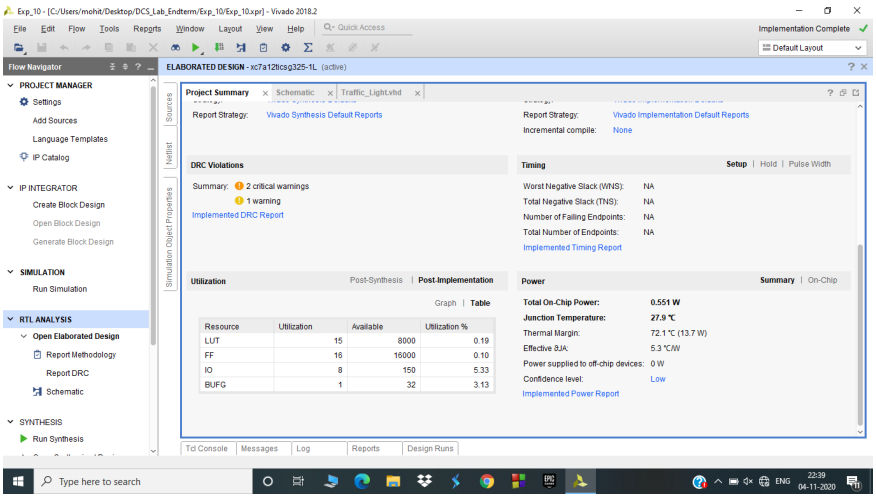

Figure 10.3 Project Summary of the Traffic Light controller

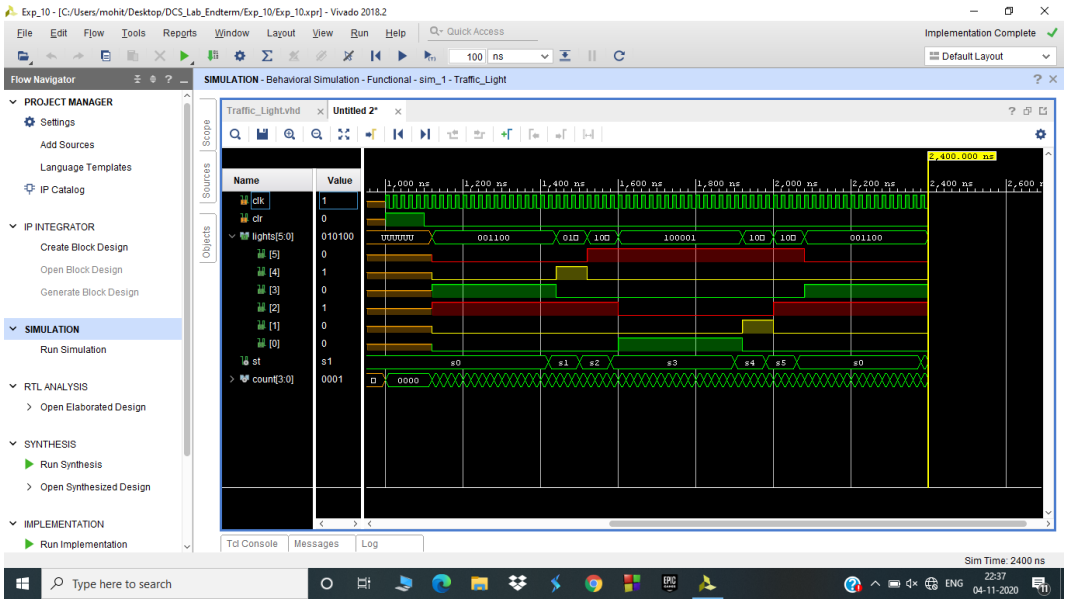

Figure 10.4 Simulation of the Traffic Light controller

10.5 Summary

Tabular comparison of all the codes in terms of area and power usage.

Name of the Entity	No. of LUT used	Total On chip Power
Traffic Light controller	15	0.551W

Table 10.1 comparision of Area and power requirements for the Traffic Light controller.