Laboratory Project 3: Adder

MdShahid Bin Emdad

March 6th, 2022

CSC 34200/34300

Table of Contents

Objective:

The objective of this assignment is to design and understand the specifications of adders and create waveforms for them. First, we were instructed to design half adder using two processes and then use the half adder as a component to create full adder 1 bit. Then we created 4-bit adder by using the half and full adder as a component. Then, we extended the work of 4bit adder by using a add/sub component. Then, we save our components in a package for future use. After that, we created N-bit design by using behavioral VHDL model and then use OVERFLOW, ZERO, NEGATIVE flags. Then, I used modelsim to verify all my designs, for example, N = 4 and N= 32. Then, I created N-bit adder/subtractor unit using lpm and compare waveforms with my designs. Finally, I created a testbench VHDL file to test Add_SUB unit for n=16 bits.

Description of Specifications, and Functionality:

The digital system I used in this assignment is Quartus Prime 20.1.1 and ModelSimSetup-20.1.1. There two packages needed are cyclonev and cyclonevi (both versions are 20.1.1). In the VHDL editor, I wrote my VHDL code to get the circuit output and then used Modelsim to simulate and run my circuit over time (ns unit).

Specifications:

1. Half Adder Ports:

   I used two inputs one for the first operand of the adder and other one is the second operand of the adder. I also used carry which is the bit that is carried in from the next less significant stage.

2. Full Adder Ports:

   I used two inputs one for the first operand of the adder and other one is the second operand of the adder. I also used cin which is the bit that is carried in from the next less significant stage. I used cout for the output which is bit that is carried over from an addition when necessary and sum is the last bit of the result.

3. 4-bit Adder Ports:

   I used two inputs one for the first operand of the adder and other one is the second operand of the adder. I also used cin which is the bit that is carried in from the next less significant stage. I used cout for the output which is bit that is carried over from an addition when necessary and sum is the last bit of the result.

4. 16-bit Adder Ports:

   I used 2 cins, (cin1 and cin2) which are the first and second operand of the adder for the 16 bits. I also had op which is the bit that carried in from the less

significant stage. For the outputs, I had neg which is the carry output that carried in from addition, sum (sum of the two inputs, cin1 and cin2), overflow which checks for overflow errors and finally zero for two input addition.

5. 32-bit Adder Ports:

   I used two operands. The first operand of the adder and other one is the second operand of the adder. Then, I used obit which is the bit that carried from the less significant stage. I had carry for the bit output from addition, sum for the inputs addition, overflow which checks for overflow errors and finally zero for two input addition.

Functionality:

1. **Half Adder truth table:**

x	y	Carry c	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 1: Half Adder truth table

2. **Full Adder truth table:**

Inputs			Outputs	
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

Figure 2: Full Adder truth table

3. 4-bit Adder truth table:

	A				B				Sum				Carry
Cin	A3	A2	A1	A0	B3	B2	B1	B0	S3	S2	S1	S0	Cout
0	0	0	0	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	0	1	0	0	0	1	0	0	1	0	0	0
0	0	0	1	1	0	0	1	1	0	1	1	0	0
0	0	1	0	0	0	1	0	0	1	0	0	0	0
0	0	1	0	1	0	1	0	1	1	0	1	0	0
0	0	1	1	0	0	1	1	0	1	1	0	0	0
0	0	1	1	1	0	1	1	1	1	1	1	0	0
0	1	0	0	0	1	0	0	0	0	0	0	0	1
0	1	0	0	1	1	0	0	1	0	0	1	0	1
0	1	0	1	0	1	0	1	0	0	1	0	0	1
0	1	0	1	1	1	0	1	1	0	1	1	0	1
0	1	1	0	0	1	1	0	0	1	0	0	0	1
0	1	1	0	1	1	1	0	1	1	0	1	0	1
0	1	1	1	0	1	1	1	0	1	1	0	0	1
0	1	1	1	1	1	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3: 4-bit Full Adder truth table

4. 4-bit Add/Sub Truth Table:

	B3	B2	B1	B0	M	X3	X2	X1	X0
Addition	0	0	0	0	**0**	0	0	0	0
	0	0	0	1	**0**	0	0	0	1
	0	0	1	0	**0**	0	0	1	0
	0	0	1	1	**0**	0	0	1	1
	0	1	0	0	**0**	0	1	0	0
	0	1	0	1	**0**	0	1	0	1
	0	1	1	0	**0**	0	1	1	0
	0	1	1	1	**0**	0	1	1	1
	1	0	0	0	**0**	1	0	0	0
	1	0	0	1	**0**	1	0	0	1
Subs traction	0	0	0	0	**1**	1	0	0	1
	0	0	0	1	**1**	1	0	0	0
	0	0	1	0	**1**	0	1	1	1
	0	0	1	1	**1**	0	1	1	0
	0	1	0	0	**1**	0	1	0	1
	0	1	0	1	**1**	0	1	0	0
	0	1	1	0	**1**	0	0	1	1
	0	1	1	1	**1**	0	0	1	0
	1	0	0	0	**1**	0	0	0	1
	1	0	0	1	**1**	0	0	0	0

Figure 4: 4-bit Add/Sub Truth Table

5. 16-bit Adder truth table:

By combining four 4-cycle CLAs, a 16-bit adder can be made yet extra rationale is required as an LCU. The LCU acknowledges the gathering propagate() and group generate ( ) from every one of the four CLAs. furthermore, have the accompanying articulations for each CLA adder: The LCU then, at that point, creates the convey input for each CLA.

6. **32-bit Adder truth table:**

To make a 32-bit adder, the necessity is 4 8-bits full adders. It will result in a 32bit sum and create carry outputs.

Code:

1. Task 1:

In figure 5, we can see the project summary for half adder.

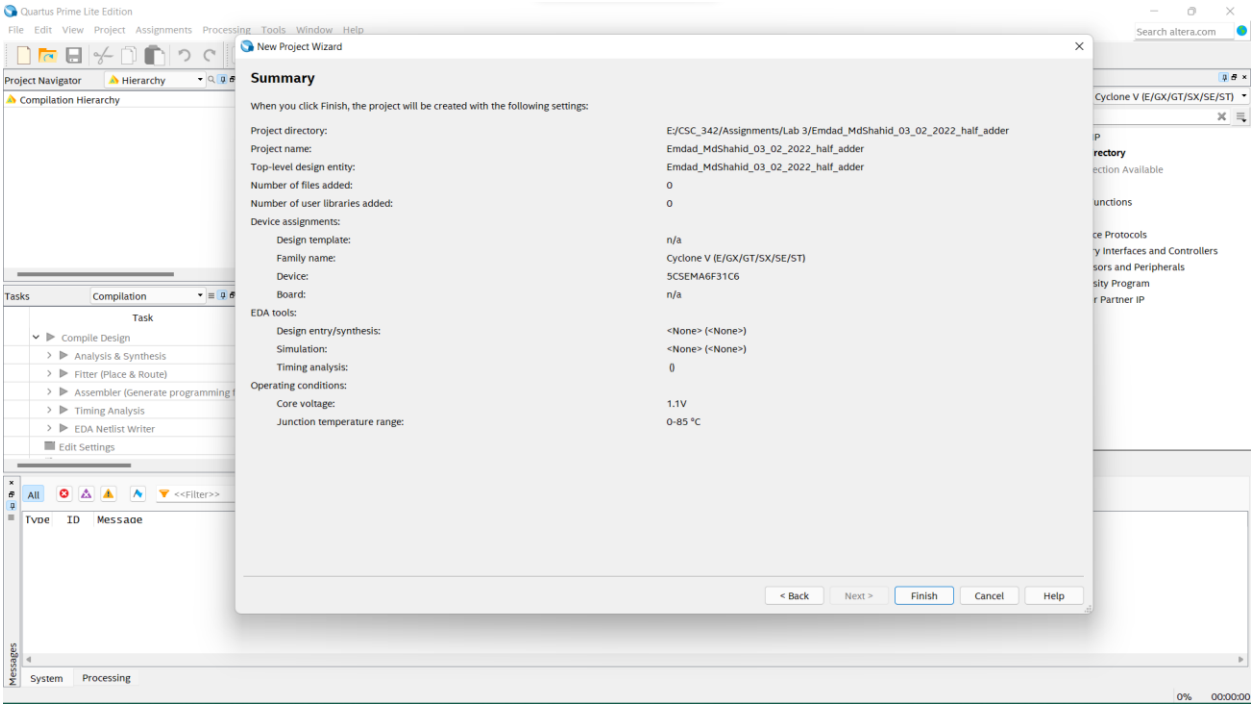


Figure 5: Half Adder Project Summary

In figure 6, we can see the VHDL code for the half adder.

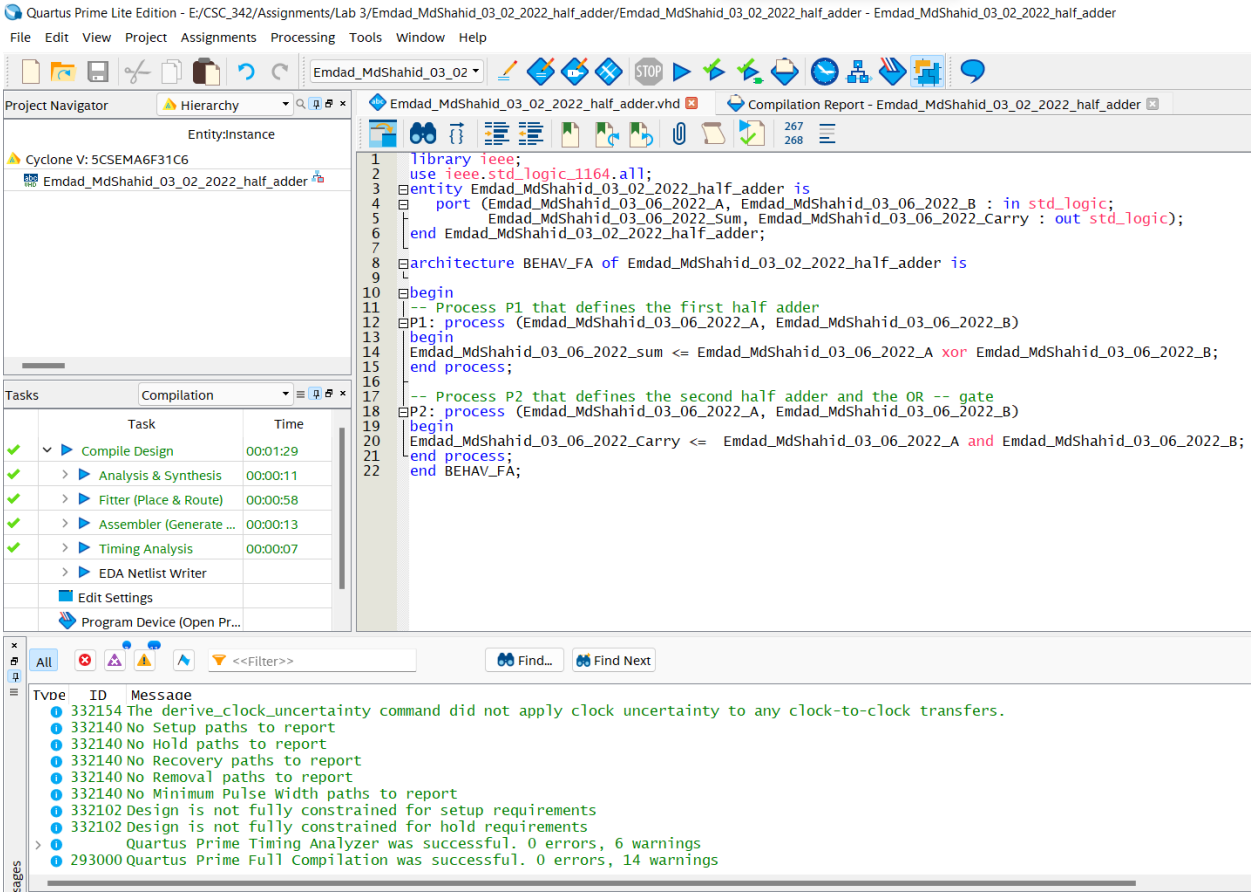


Figure 6: Half Adder VHDL Code

In figure 7, we can see the successful compilation report for the half adder.

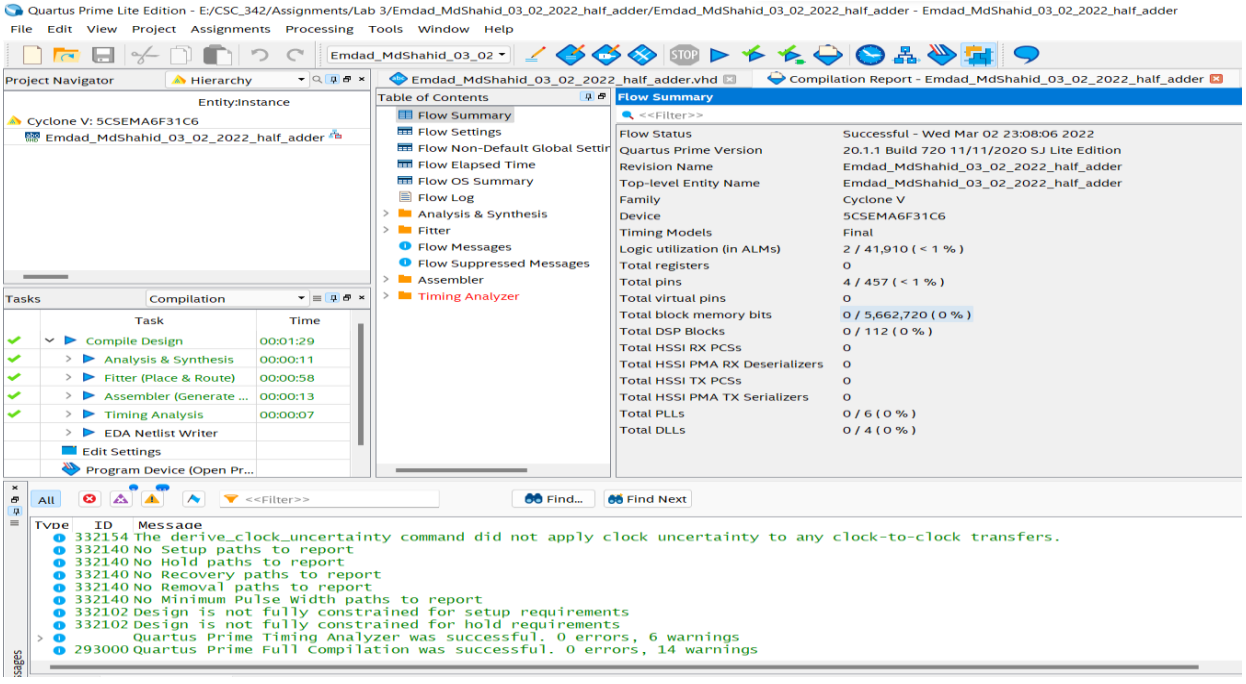


Figure 7: Half Adder successful compilation report

2. TASK 2:

In figure 8, we can see the full adder project summary.

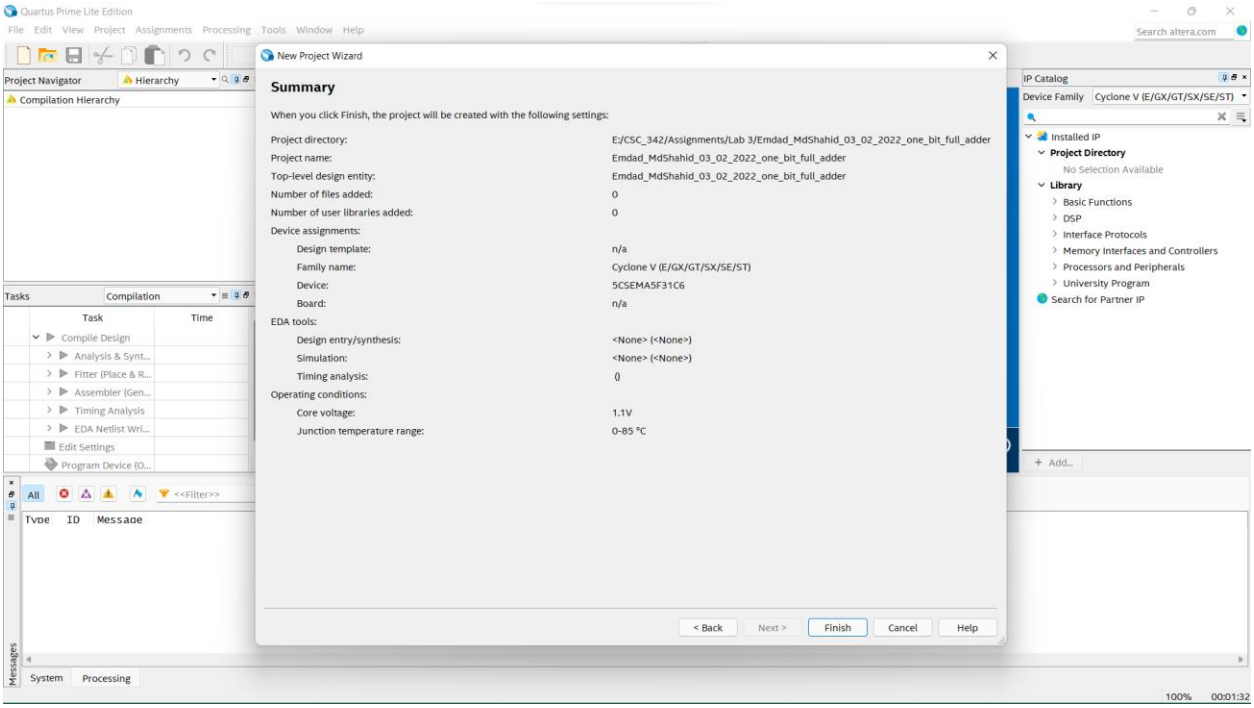


Figure 8: 1-bit Full Adder Project Summary

In figure 9, I used the half adder as a component for the 1-bit full adder.

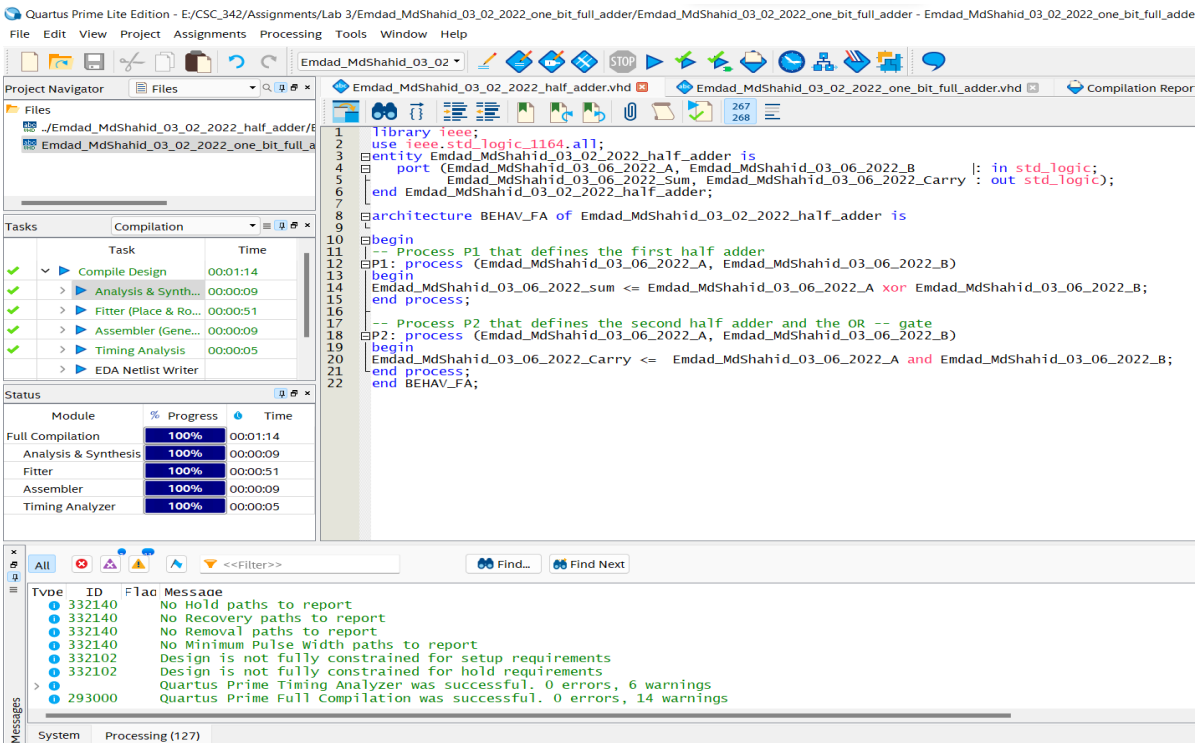


Figure 9: half Adder VHDL Code as a component

In figure 10, we can see the HDL code for the 1-bit full adder where I used half adder as a component.

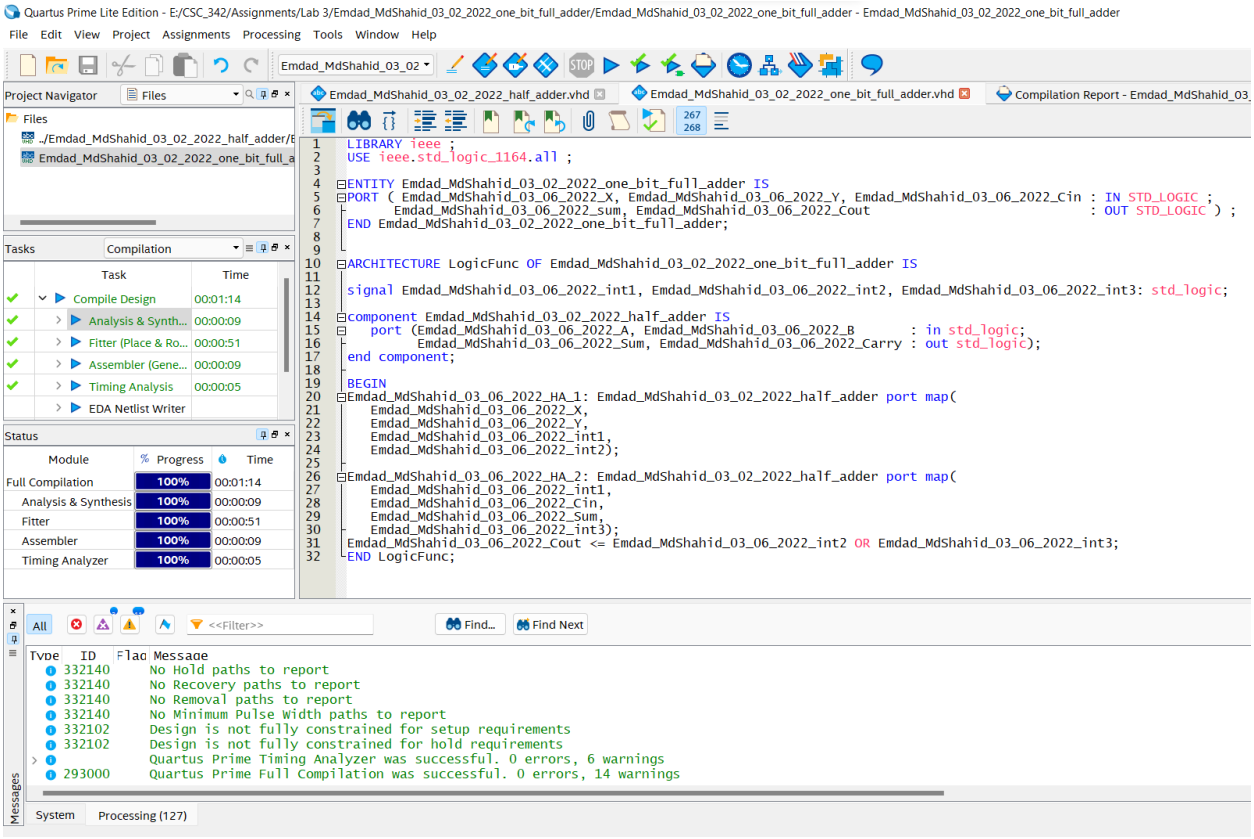


Figure 10: 1-bit full Adder VHDL Code

In figure 11, we can see the successful compilation report for 1-bit full adder.

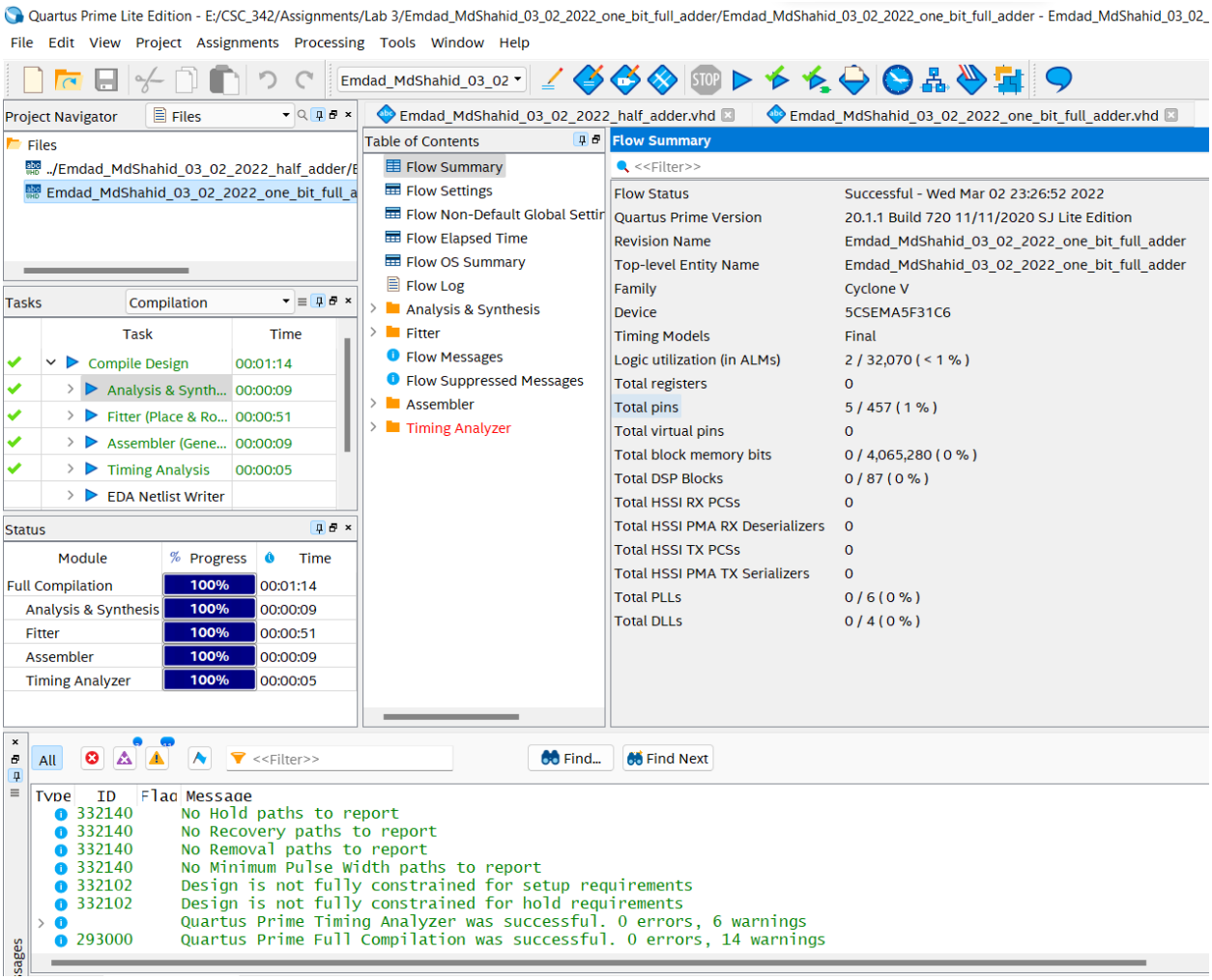


Figure 11: 1-bit full adder successful compilation report

3. TASK 3:

In figure 12, we can see the project summary for 4-bit full adder.

Figure 12: 4-bit full adder Project Summary

In figure 13, we can see I used half adder as a component.

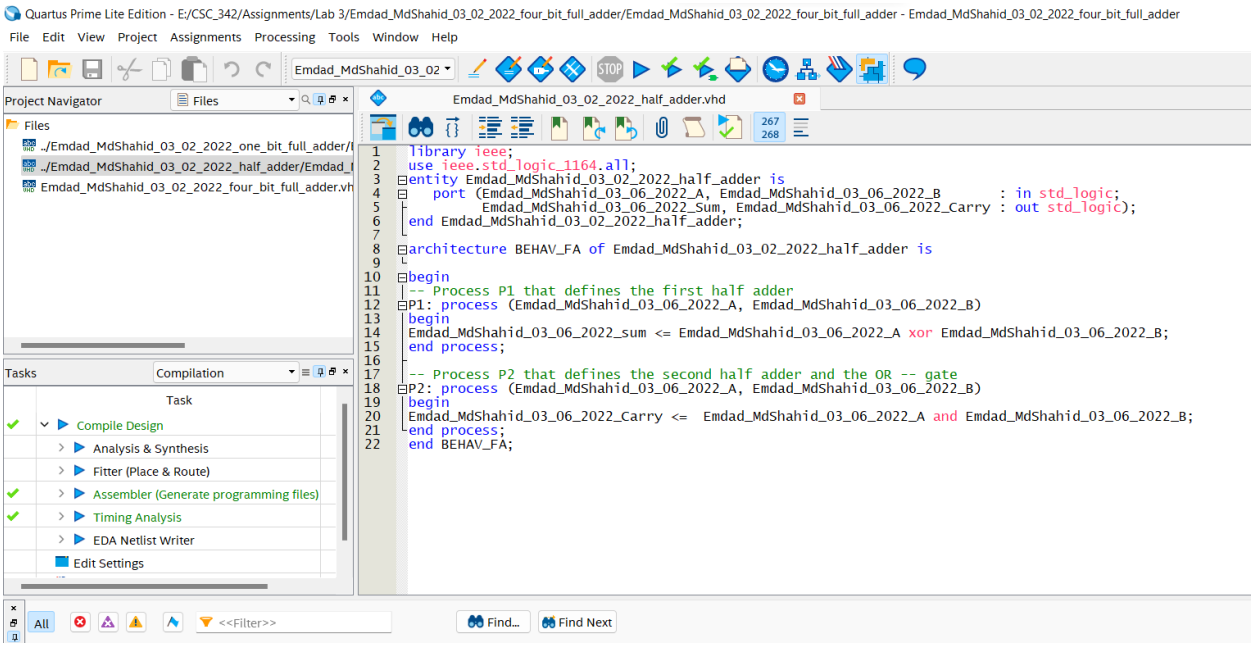


Figure 13: half adder as a component

In figure 13, we can see I used full adder as a component.

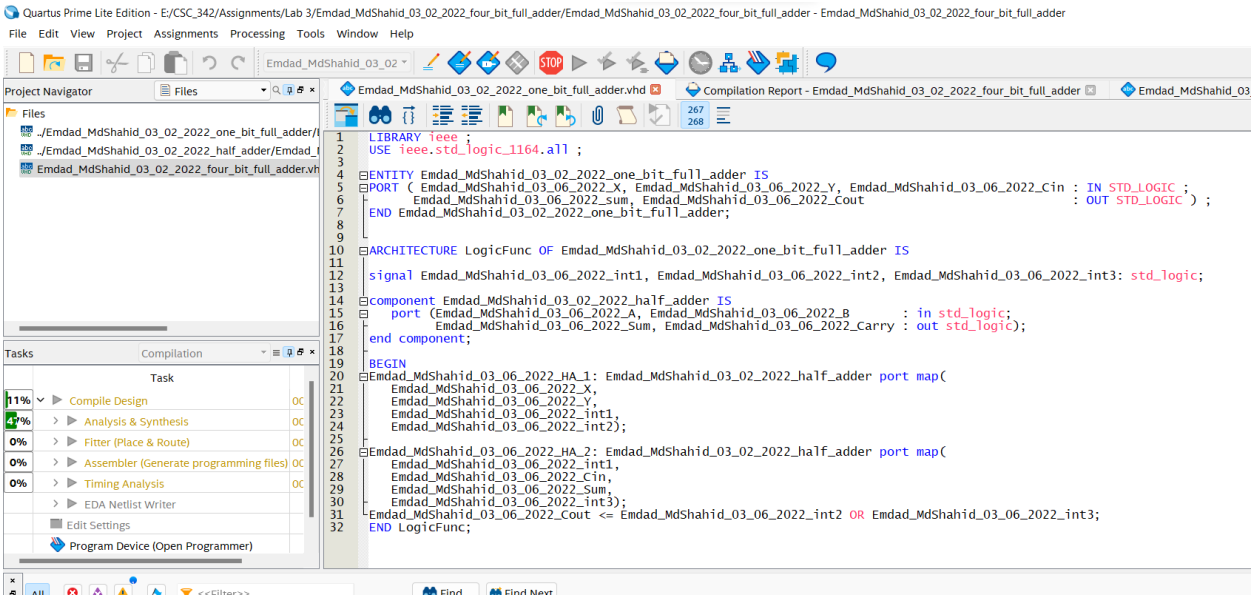


Figure 14: full adder as a component

In figure 15, we can see the 4-bit full adder VHDL code.

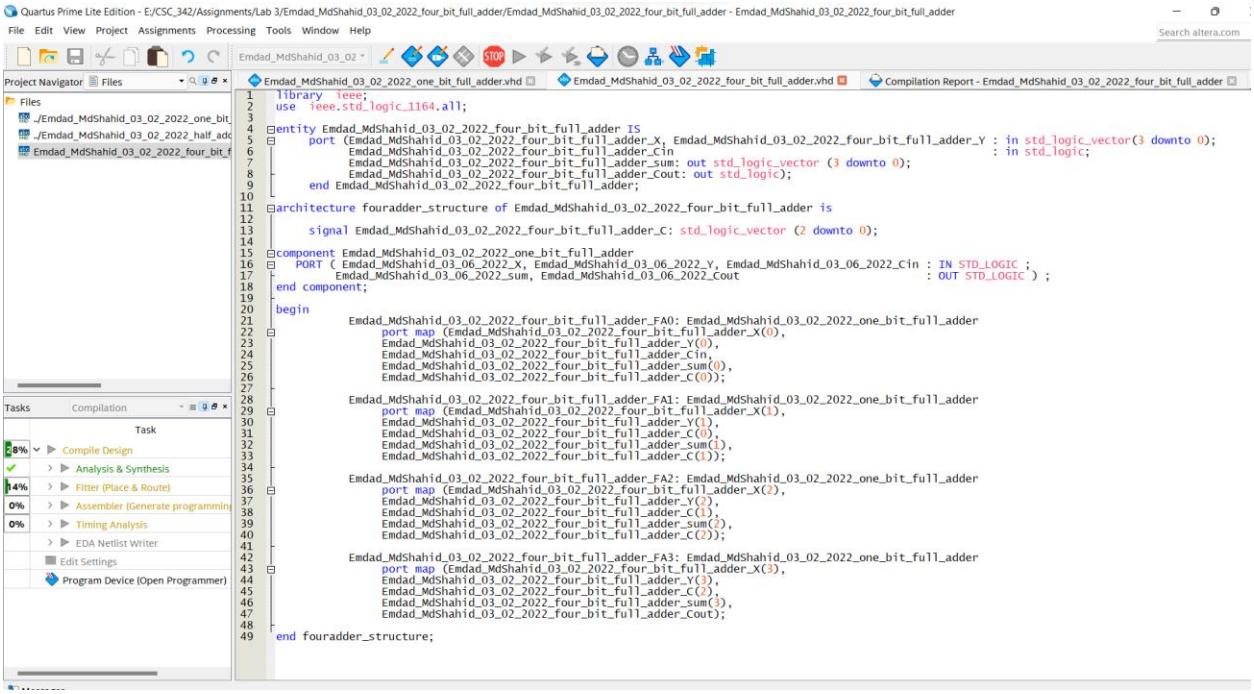
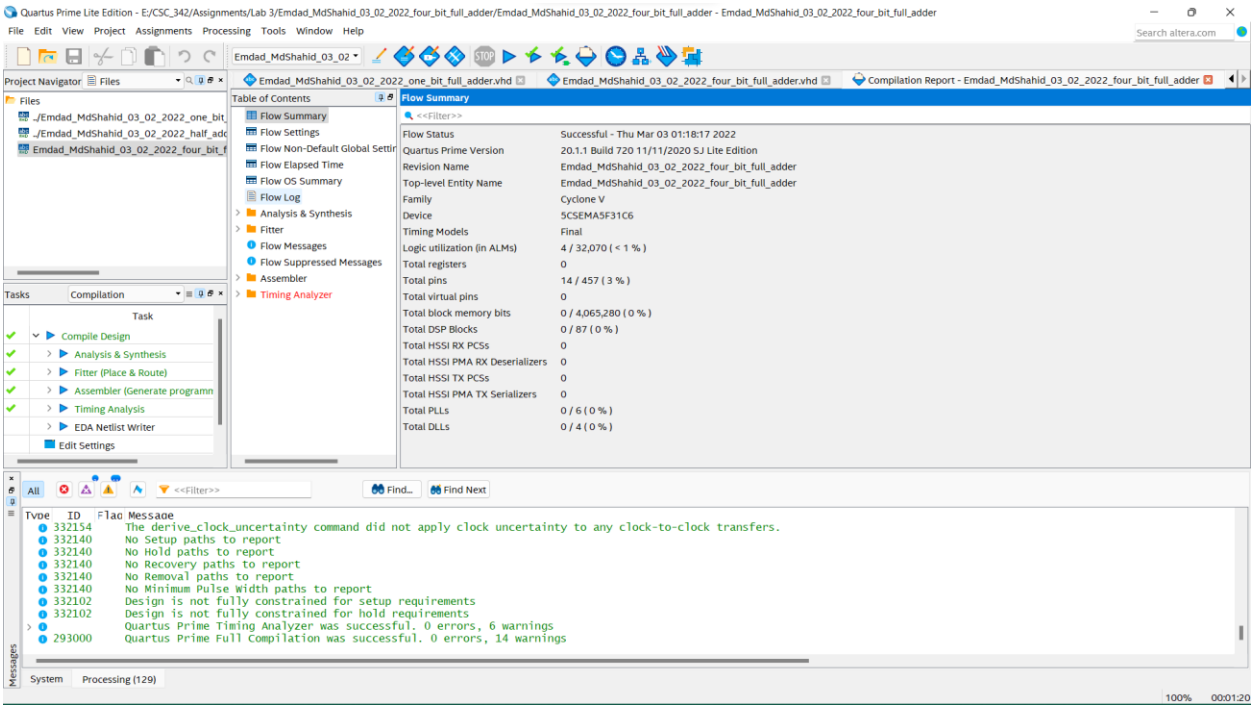


Figure 15: 4-bit full adder VHDL code.

In figure 17, we can see the successful compilation report for 4-bit full adder.



4. TASK 4:

In figure 17, we can see the project summary for 4-bit add/sub.

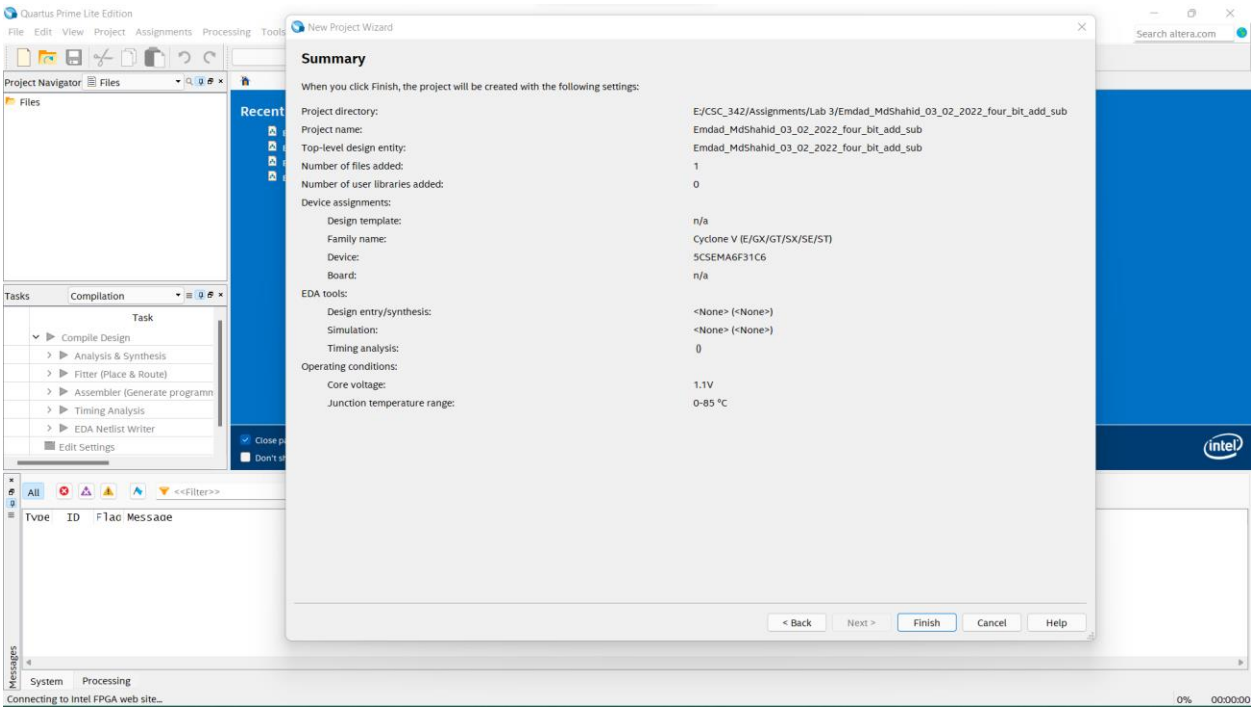


Figure 16: project summary for 4-bit add/sub.

In figure 17, we can see the VHDL code for the 4-bit add/sub.

Text Editor - E:/CSC_342/Assignments/Lab 3/Emdad_MdShahid_03_02_2022_four_bit_add_sub/Emdad_MdShahid_03_02_2022_four_bit_add_sub - Emdad_MdShahid_03_02_2022_four_bit_add_sub - [Emdad_MdShahid_03_02_2022_four_bit_add_sub.vhd]*

File Edit View Project Processing Tools Window Help

```
1  LIBRARY ieee ;
2  USE ieee.std_logic_1164.all;
3
4  ENTITY Emdad_MdShahid_03_02_2022_four_bit_add_sub IS
5     port (Emdad_MdShahid_03_02_2022_X         :in std_logic_vector(3 downto 0);
6           Emdad_MdShahid_03_02_2022_Y         :in std_logic_vector(3 downto 0);
7           Emdad_MdShahid_03_02_2022_Opcode    :in std_logic;
8           Emdad_MdShahid_03_02_2022_sum       :out std_logic_vector (3 downto 0);
9           Emdad_MdShahid_03_02_2022_Cout      :out std_logic;
10          Emdad_MdShahid_03_06_2022_Overflow  :out std_logic);
11 END Emdad_MdShahid_03_02_2022_four_bit_add_sub;
12
13 ARCHITECTURE LogicFunc OF Emdad_MdShahid_03_02_2022_four_bit_add_sub IS
14 component Emdad_MdShahid_03_02_2022_one_bit_full_adder
15    PORT (Emdad_MdShahid_03_06_2022_X, Emdad_MdShahid_03_06_2022_Y, Emdad_MdShahid_03_06_2022_Cin  : IN STD_LOGIC ;
16          Emdad_MdShahid_03_06_2022_sum, Emdad_MdShahid_03_06_2022_Cout                            : OUT STD_LOGIC ) ;
17 end component;
18
19 signal Emdad_MdShahid_03_06_2022_A, Emdad_MdShahid_03_06_2022_B, Emdad_MdShahid_03_06_2022_C, Emdad_MdShahid_03_06_2022_D: std_logic;
20 signal Emdad_MdShahid_03_06_2022_E: std_logic_vector (3 downto 0);
21 BEGIN
22       Emdad_MdShahid_03_06_2022_E <= Emdad_MdShahid_03_02_2022_X XOR Emdad_MdShahid_03_02_2022_Y;
23
24      Emdad_MdShahid_03_02_2022_four_bit_add_sub_FA0: Emdad_MdShahid_03_02_2022_one_bit_full_adder
25           port map(Emdad_MdShahid_03_02_2022_X(0),
26                    Emdad_MdShahid_03_06_2022_E(0),
27                    Emdad_MdShahid_03_02_2022_Opcode,
28                    Emdad_MdShahid_03_02_2022_sum(0),
29                    Emdad_MdShahid_03_06_2022_A);
30
31      Emdad_MdShahid_03_02_2022_four_bit_add_sub_FA1: Emdad_MdShahid_03_02_2022_one_bit_full_adder
32           port map(Emdad_MdShahid_03_02_2022_X(1),
33                    Emdad_MdShahid_03_06_2022_E(1),
34                    Emdad_MdShahid_03_06_2022_A,
35                    Emdad_MdShahid_03_02_2022_sum(1),
36                    Emdad_MdShahid_03_06_2022_B);
37
38      Emdad_MdShahid_03_02_2022_four_bit_add_sub_FA2: Emdad_MdShahid_03_02_2022_one_bit_full_adder
39           port map(Emdad_MdShahid_03_02_2022_X(2),
40                    Emdad_MdShahid_03_06_2022_E(2),
41                    Emdad_MdShahid_03_06_2022_B,
42                    Emdad_MdShahid_03_02_2022_sum(2),
43                    Emdad_MdShahid_03_06_2022_C);
44
45      Emdad_MdShahid_03_02_2022_four_bit_add_sub_FA3: Emdad_MdShahid_03_02_2022_one_bit_full_adder
46           port map(Emdad_MdShahid_03_02_2022_X(3),
47                    Emdad_MdShahid_03_06_2022_E(3),
48                    Emdad_MdShahid_03_06_2022_C,
49                    Emdad_MdShahid_03_02_2022_sum(3),
50                    Emdad_MdShahid_03_06_2022_D);
51
52       Emdad_MdShahid_03_06_2022_Overflow <= Emdad_MdShahid_03_06_2022_C XOR Emdad_MdShahid_03_06_2022_D;
53      Emdad_MdShahid_03_02_2022_Cout <= Emdad_MdShahid_03_06_2022_D;
54 END LogicFunc;
```

Figure 17: 4-bit add/sub VHDL code

In figure 18, we can see the 4-bit add/sub VHDL successfully compiled.

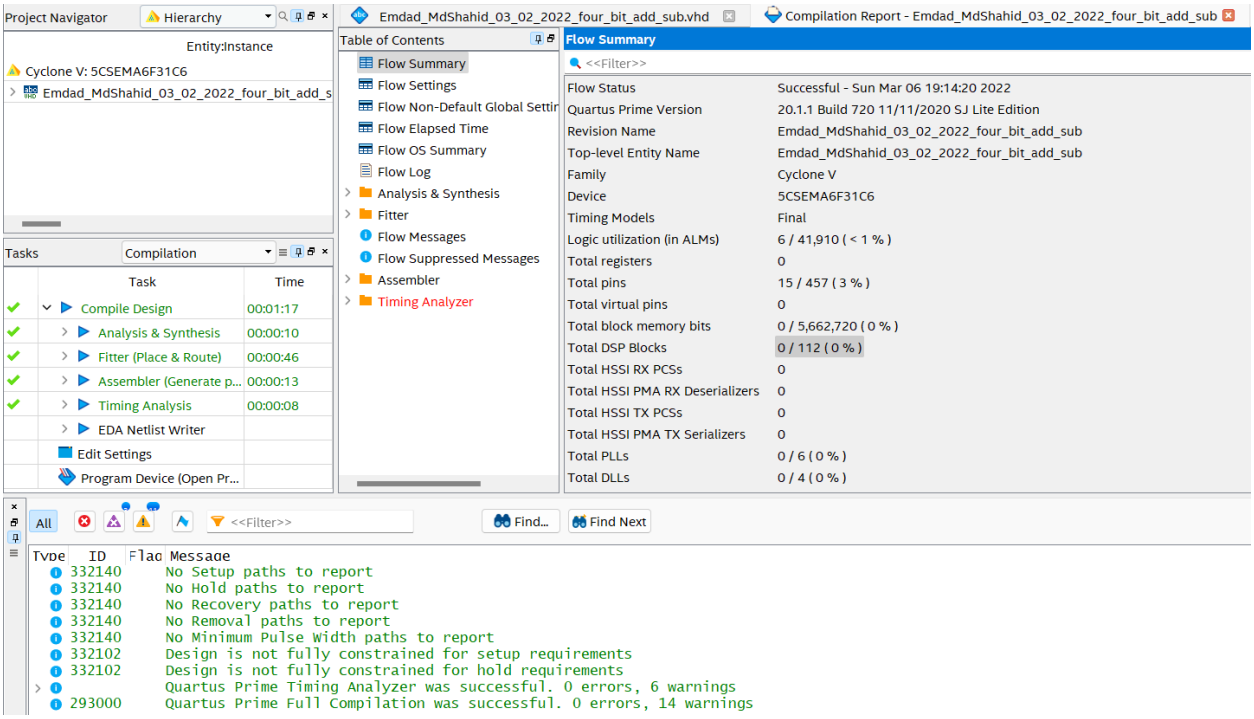


Figure 18: successful compilation report:

5. Task 5:

In figure 19, we can see the package project summary.

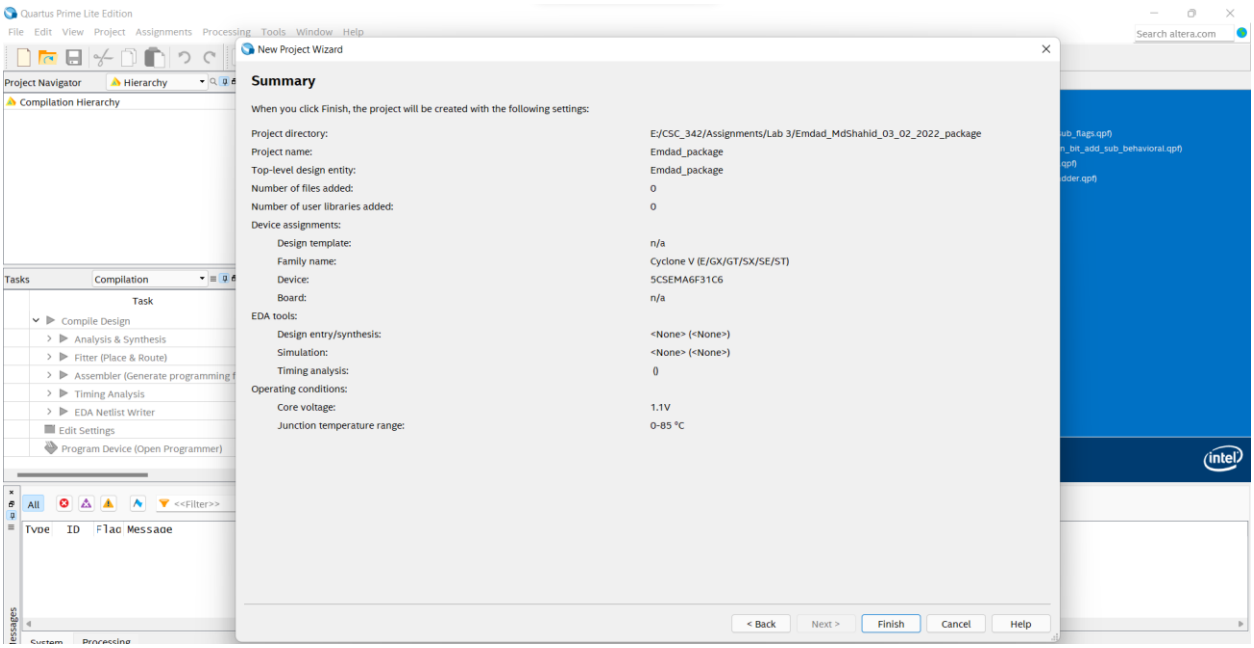


Figure 19: package project summary

In figure 20, I added my components in a package and compiled it and created a package for future use.

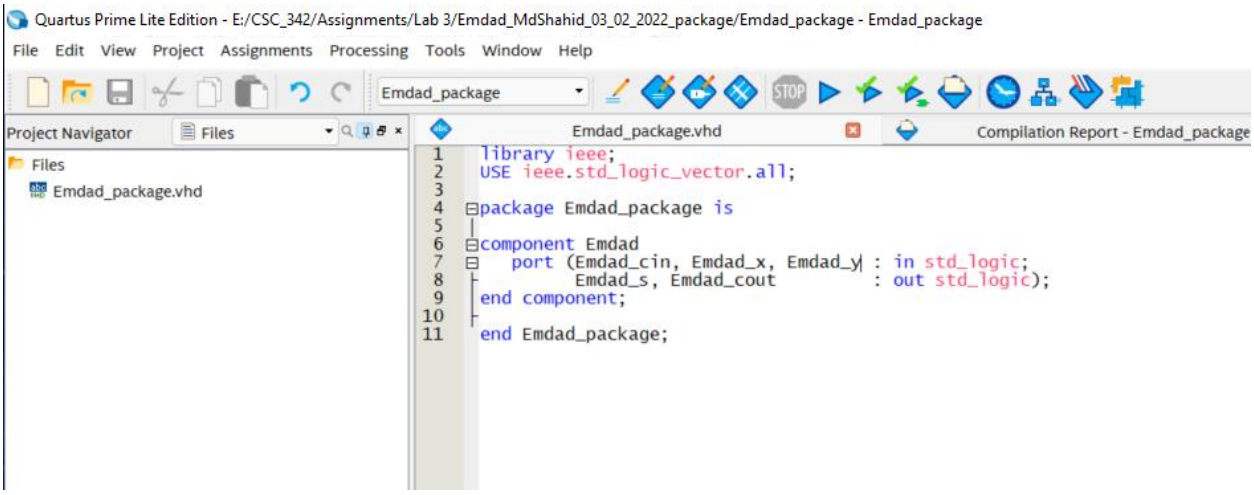


Figure 20: component package

6. TASK 6:

In figure 21, we can see the project summary for n-bit add/sub using behavioral VHDL model.

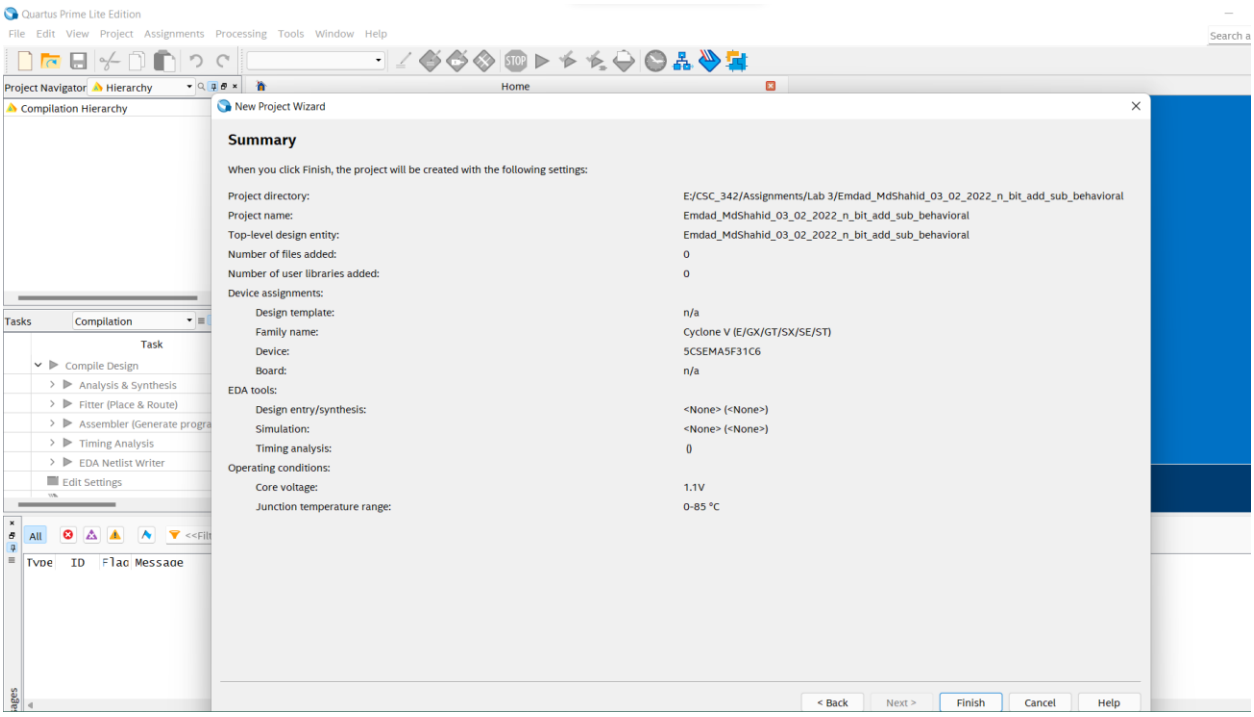


Figure 21: project summary for n-bit add/sub using behavioral VHDL model

In figure 22, we can see the n-bit add/sub using behavioral VHDL code.

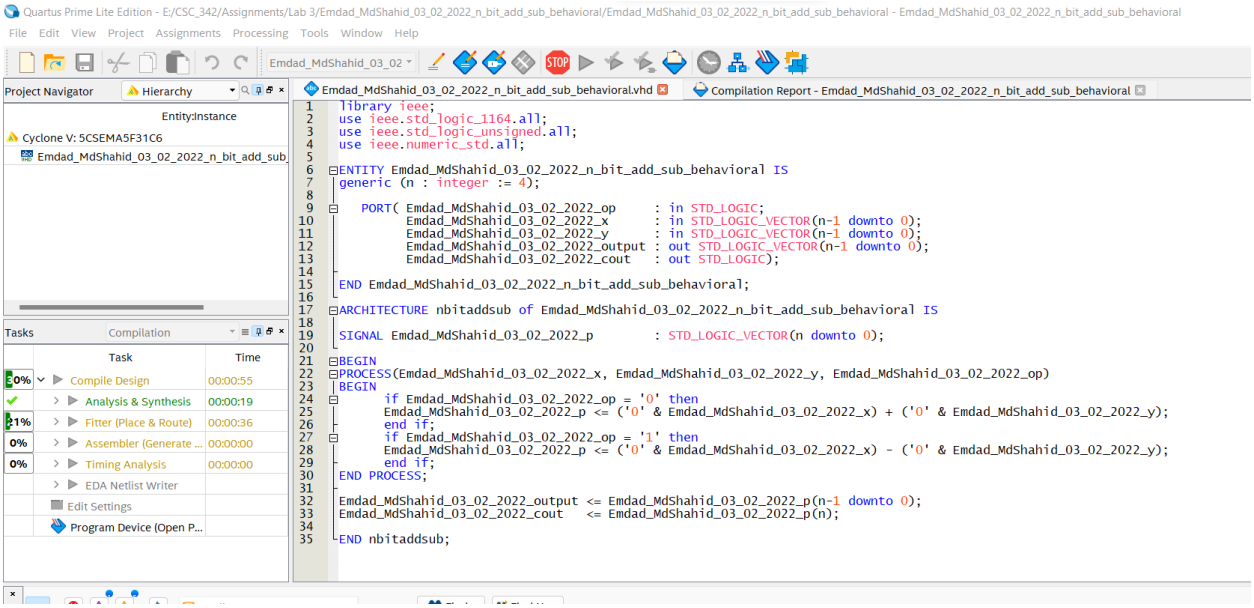


Figure 22: n-bit add/sub using behavioral VHDL code

In figure 23, we can see successful compilation report for the n-bit add/sub using behavioral VHDL model.

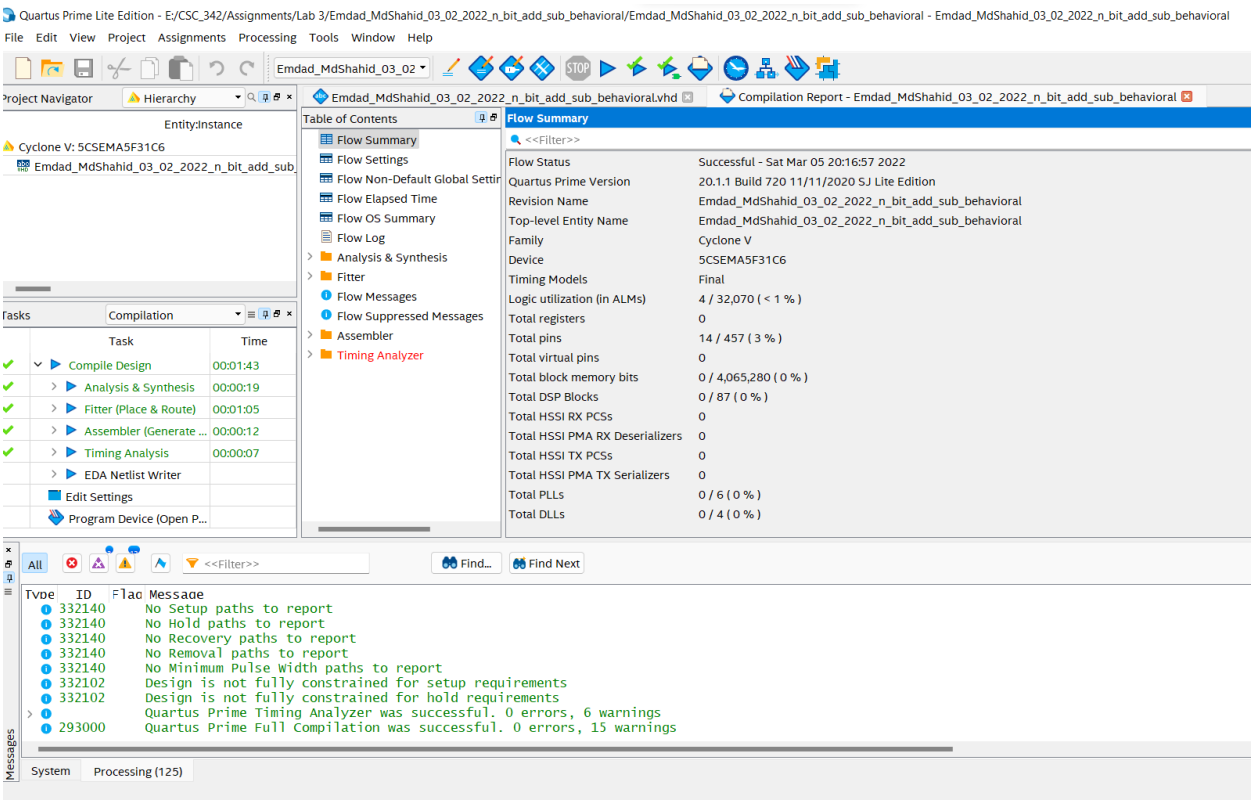


Figure 23: successful compilation report

7. LAB 7:

In figure 24, we can see the project summary for n-bit add/sub flags.

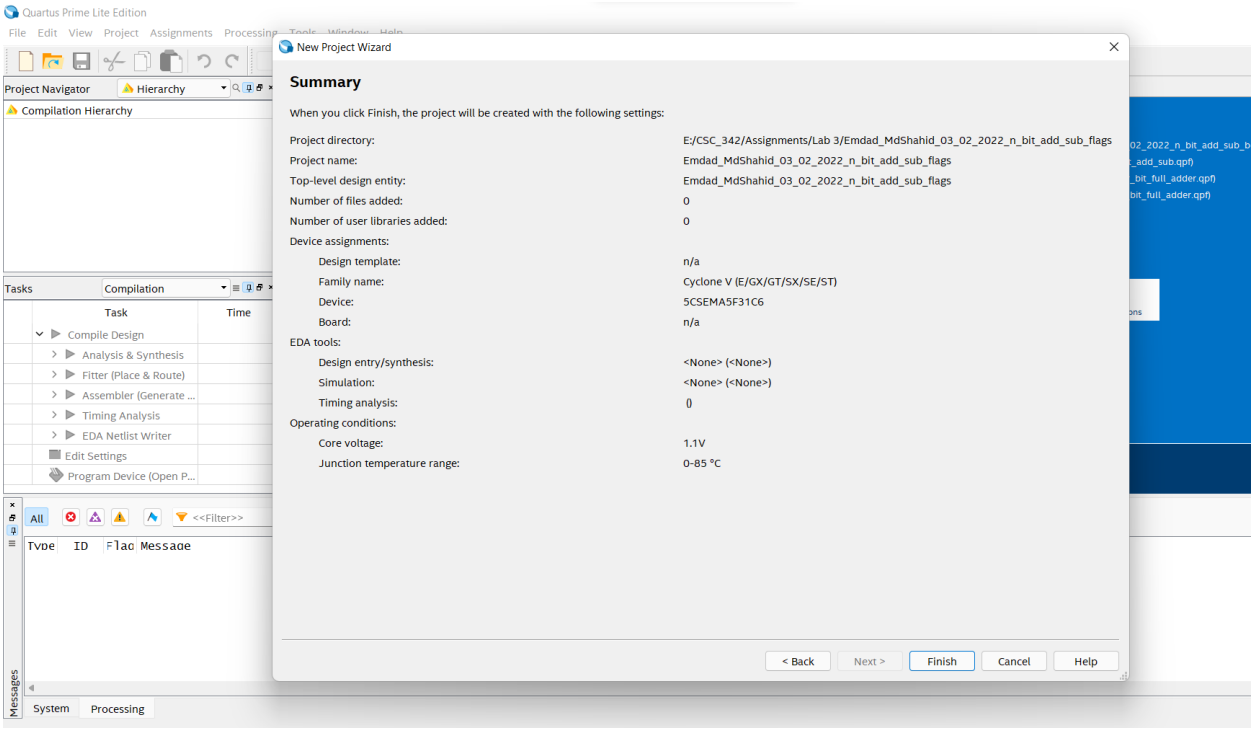


Figure 24: project summary for n-bit add/sub flags

In figure 25, we can see the VHDL code for n-bit add/sub.

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

ENTITY Emdad_MdShahid_03_02_2022_n_bit_add_sub_flags IS
generic (n : integer := 32);

   PORT( Emdad_MdShahid_03_02_2022_op        : in STD_LOGIC;
         Emdad_MdShahid_03_02_2022_x         : in STD_LOGIC_VECTOR(n-1 downto 0);
         Emdad_MdShahid_03_02_2022_y         : in STD_LOGIC_VECTOR(n-1 downto 0);
         Emdad_MdShahid_03_02_2022_output    : out STD_LOGIC_VECTOR(n-1 downto 0);
         Emdad_MdShahid_03_02_2022_Overflow  : out STD_LOGIC;
         Emdad_MdShahid_03_02_2022_zero      : out STD_LOGIC;
         Emdad_MdShahid_03_02_2022_negative  : out STD_LOGIC);

END Emdad_MdShahid_03_02_2022_n_bit_add_sub_flags;

ARCHITECTURE nbitaddsubflags of Emdad_MdShahid_03_02_2022_n_bit_add_sub_flags IS

SIGNAL Emdad_MdShahid_03_02_2022_p        : STD_LOGIC_VECTOR(n downto 0);

BEGIN
PROCESS(Emdad_MdShahid_03_02_2022_x, Emdad_MdShahid_03_02_2022_y, Emdad_MdShahid_03_02_2022_op)
 BEGIN
      if Emdad_MdShahid_03_02_2022_op = '0' then
      Emdad_MdShahid_03_02_2022_p <= ('0' & Emdad_MdShahid_03_02_2022_x) + ('0' & Emdad_MdShahid_03_02_2022_y);
      Emdad_MdShahid_03_02_2022_negative <= '0';
      end if;
      if Emdad_MdShahid_03_02_2022_op = '1' then
         if Emdad_MdShahid_03_02_2022_x < Emdad_MdShahid_03_02_2022_y then
            Emdad_MdShahid_03_02_2022_negative <= '1';
         else
            Emdad_MdShahid_03_02_2022_negative <= '0';
         end if;
            Emdad_MdShahid_03_02_2022_p <= ('0' & Emdad_MdShahid_03_02_2022_x) - ('0' & Emdad_MdShahid_03_02_2022_y);
      end if;
END PROCESS;

Emdad_MdShahid_03_02_2022_output <= Emdad_MdShahid_03_02_2022_p(n-1 downto 0);
Emdad_MdShahid_03_02_2022_Overflow  <= Emdad_MdShahid_03_02_2022_p(n);
Emdad_MdShahid_03_02_2022_zero <= '1' when Emdad_MdShahid_03_02_2022_p(n-1 downto 0) = 0 else '0';

END nbitaddsubflags;
```

Figure 25: VHDL code for n-bit add/sub.

In figure 26, we can see the succesful compilation report.

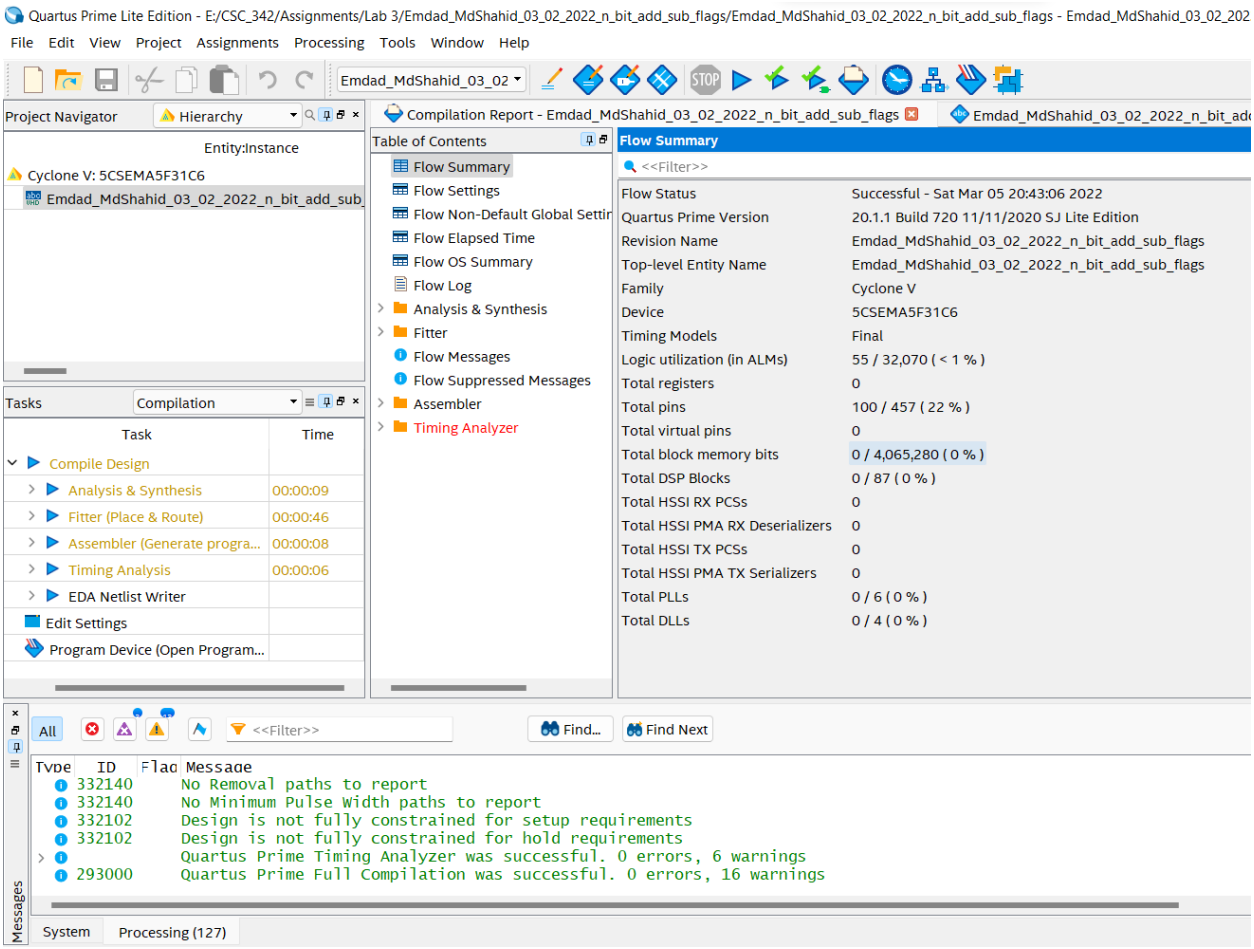


Figure 26: successful compilation report.

8. TASK 8:

Task 8 was divided into 2 parts. First, we worked on the 4-bit design in Modelsim and to verify that I created a testbench to get a waveform out which was provided in the simulation section.

In figure 27, we can see the project summary for N=4 and N =32-bit testbench compilation.

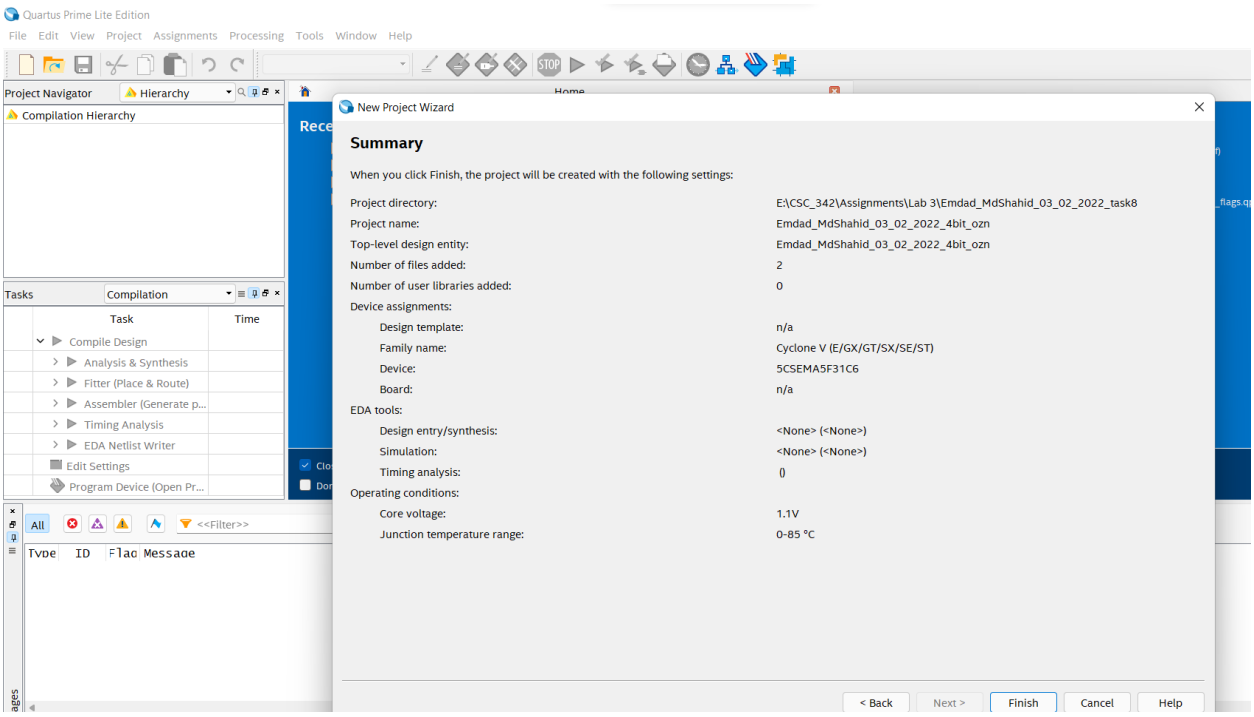


Figure 27: task 8 project summary

In figure 28, we can see the code wirtten for 4-bit. I also added the flags.

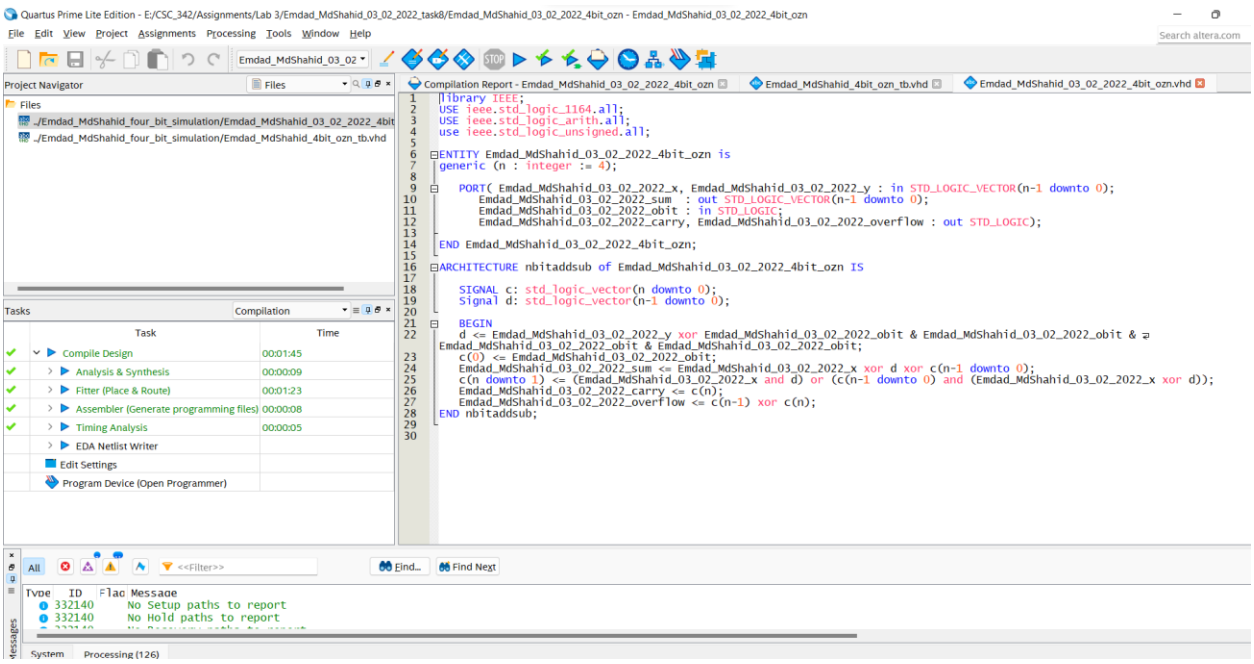


Figure 28: 4bit add/sub with flags

In figure 29, we can see the testbench for 4-bit add/sub flags waveform check.

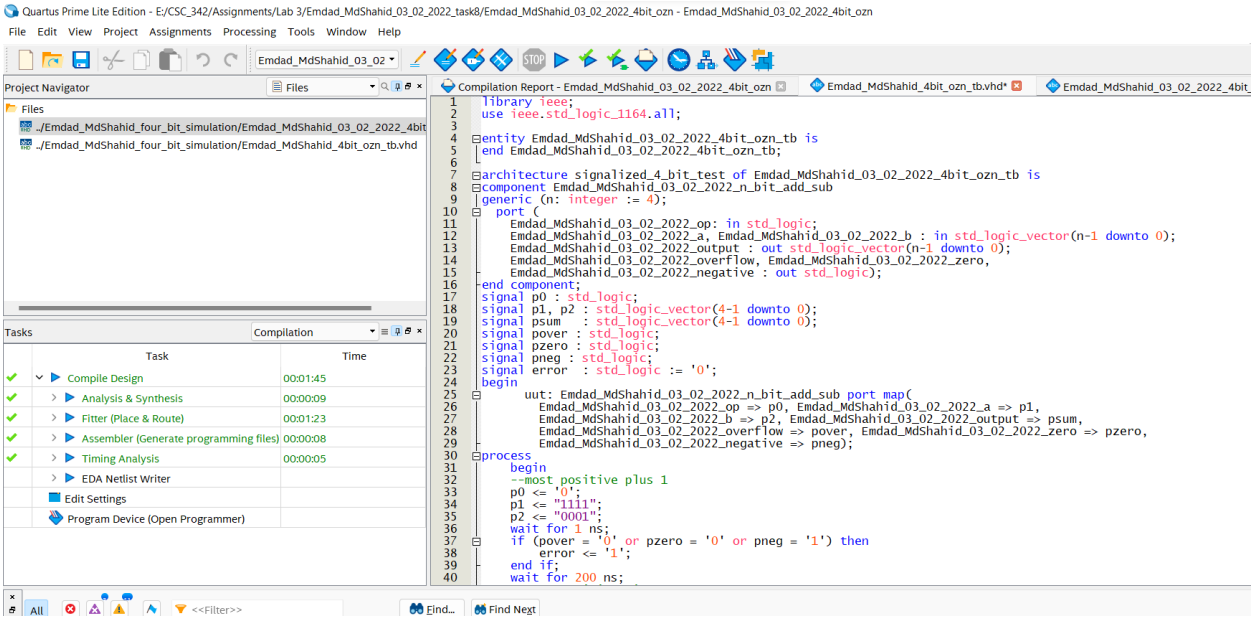


Figure 29: 4-bit testbench (Part 1)

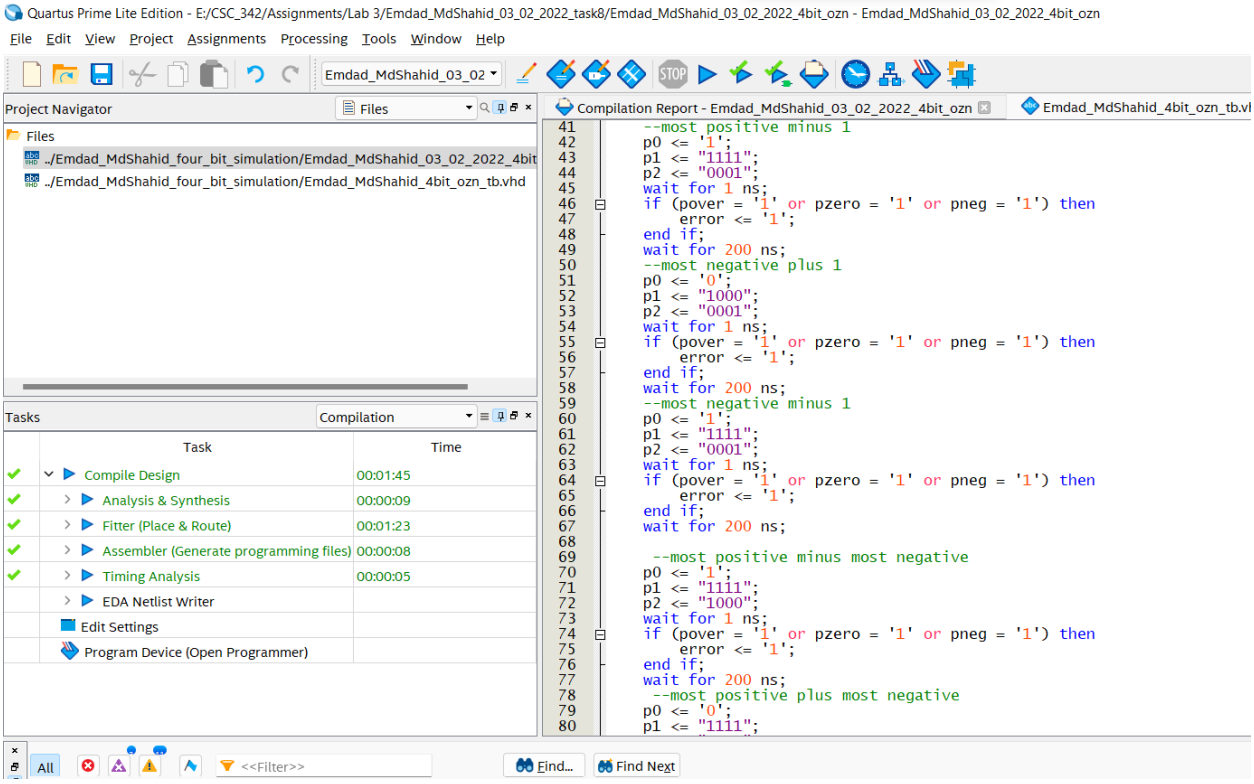


Figure 30: 4-bit testbench (Part 2)

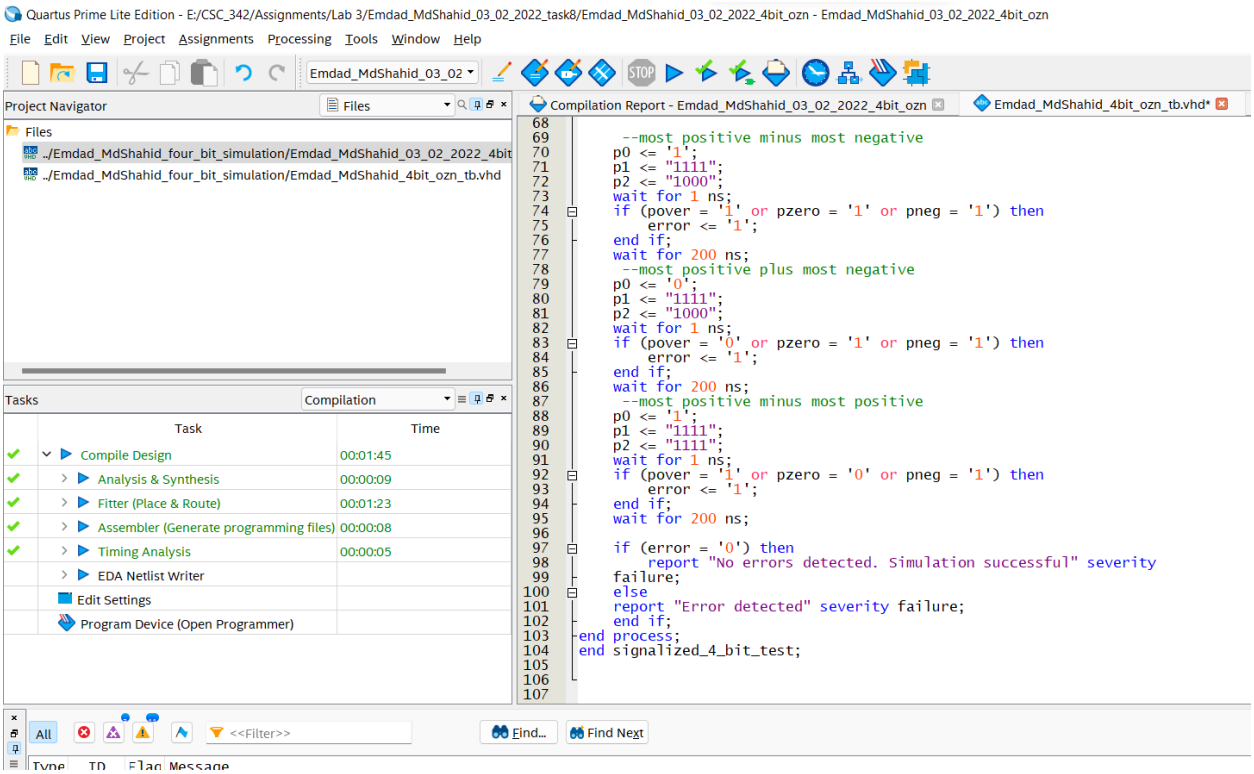


Figure 31:4-bit testbench (Part 3)

In figure 32, we can see the project summary for 32 bits.

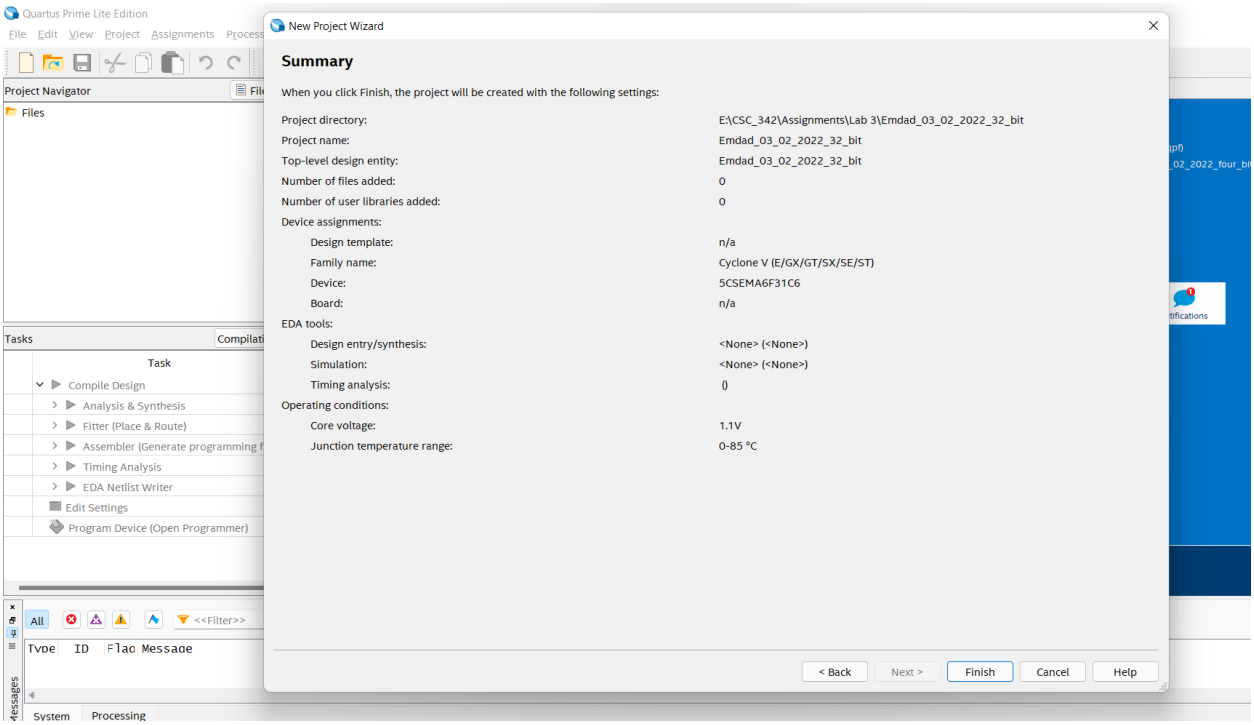


Figure 32: 32 bits project summary

In figure 33, we can see the VHDL code for the 32-bits with flags.

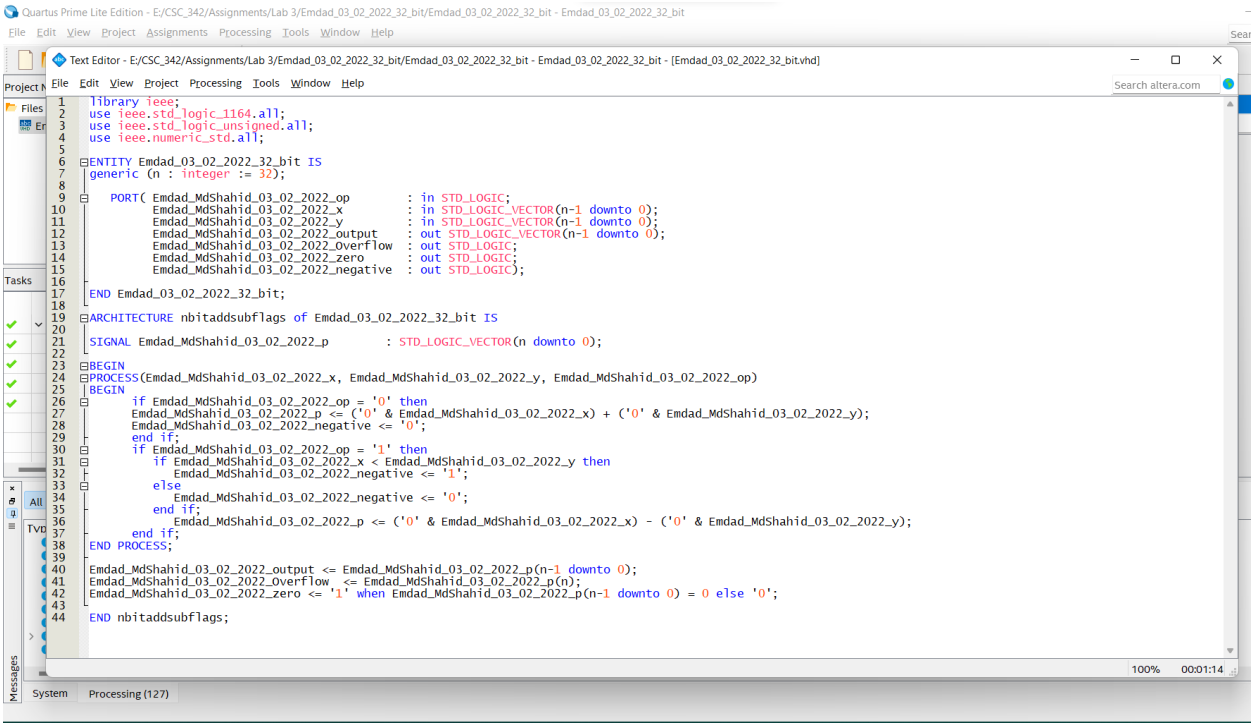


Figure 33: VHDL code for the 32-bits with flags.

In figure 34, we can see it compiled succesfully.

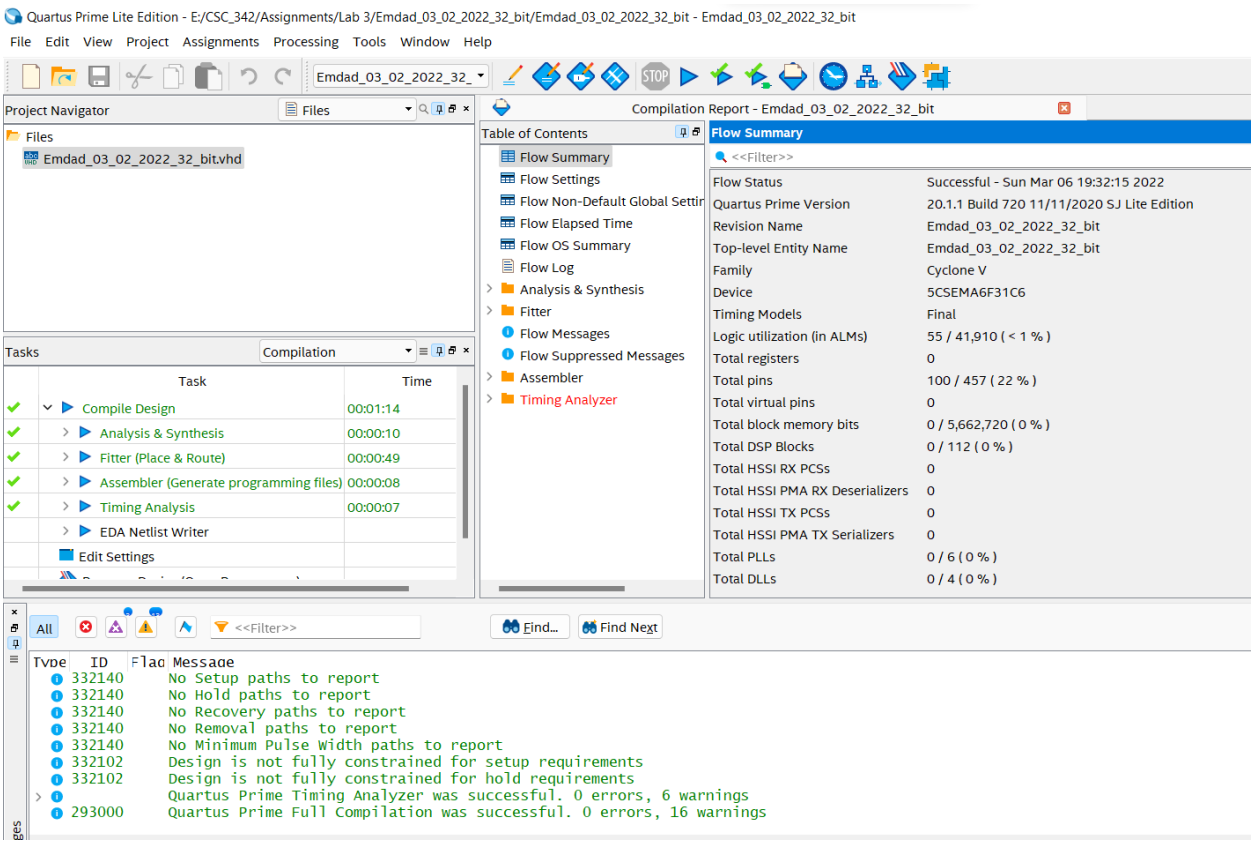


Figure 34: successful compilation report

9. TASK 9:

In figure 35, we can see n-bit LMP add/sub project summary.

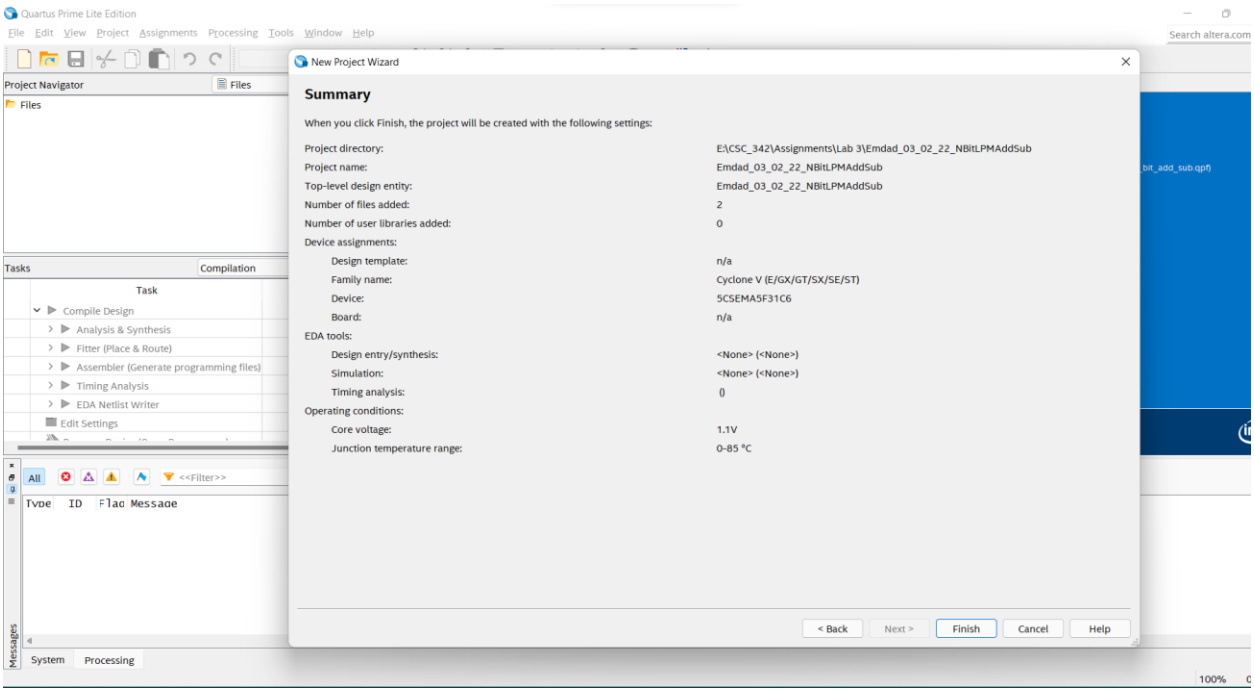


Figure 35:n-bit LMP add/sub

In figure 36, we can see the VHDL code.

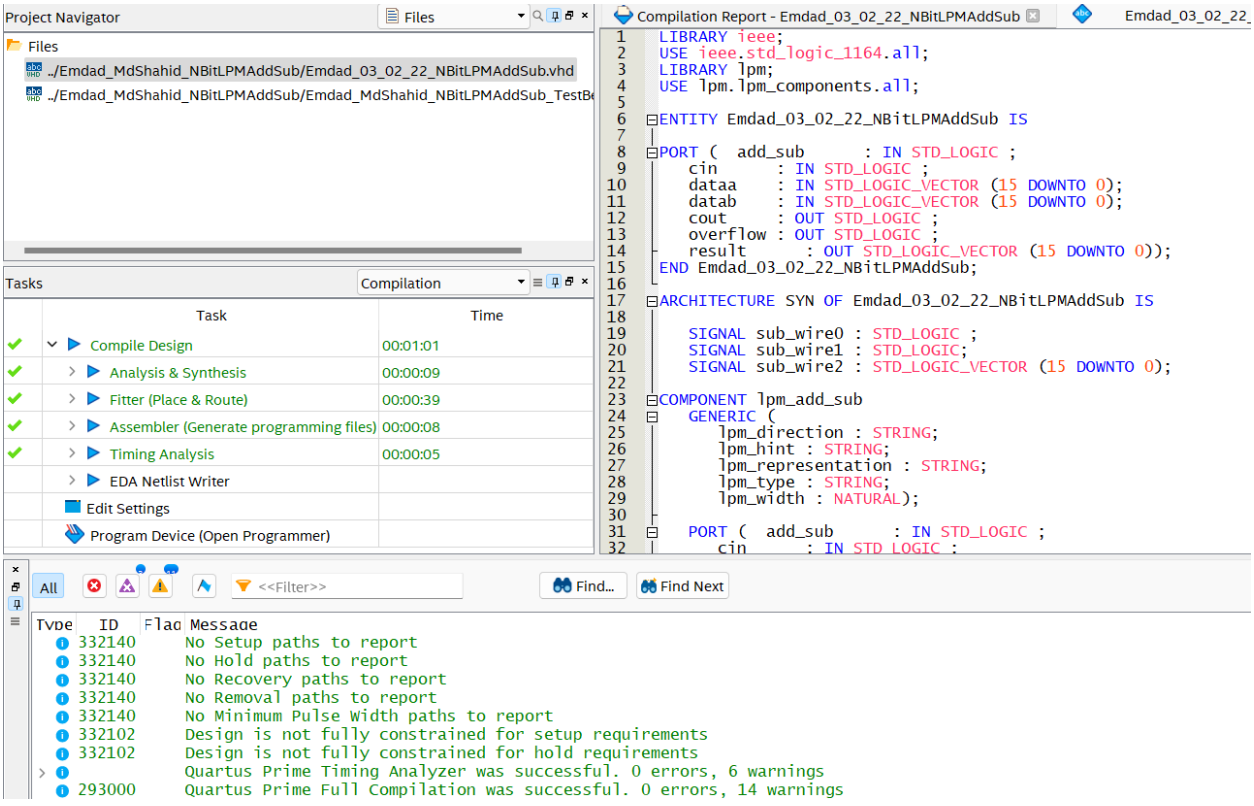


Figure 36: VHDL code for n-bit lmp add/sub(Part 1)

In figure 37, we can see the rest of the code for n-bit lmp add/sub.

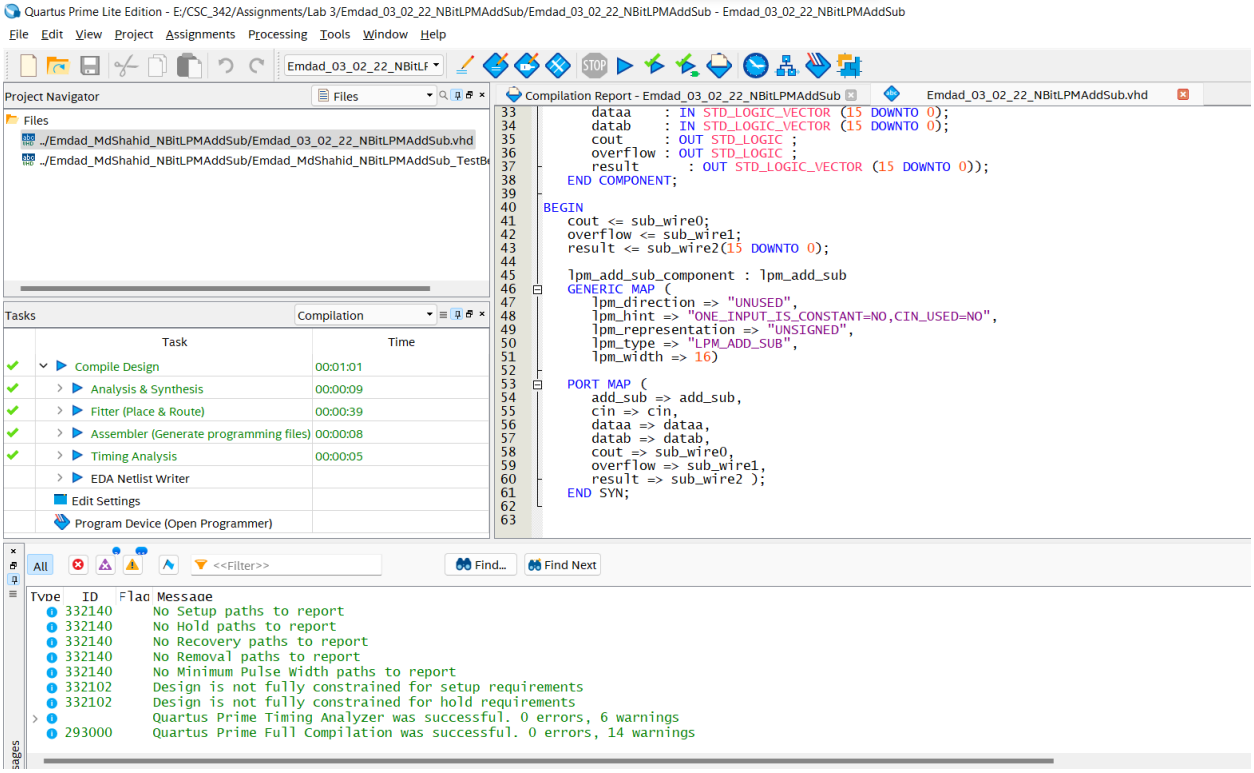


Figure 37: VHDL code for n-bit lmp add/sub (Part 2)

In figure 38, we can see the testbench for the n-bitlmpadd/sub to verify our waveform.

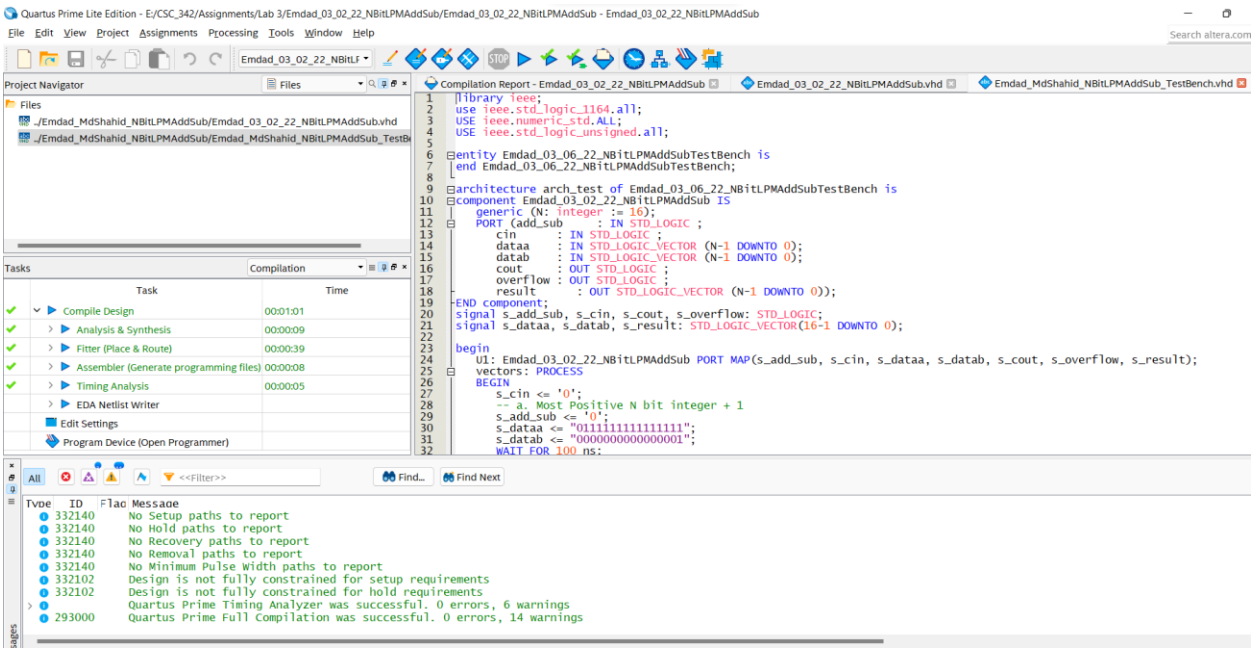


Figure 38: nbitlmpaddsub testbench (Part 1)

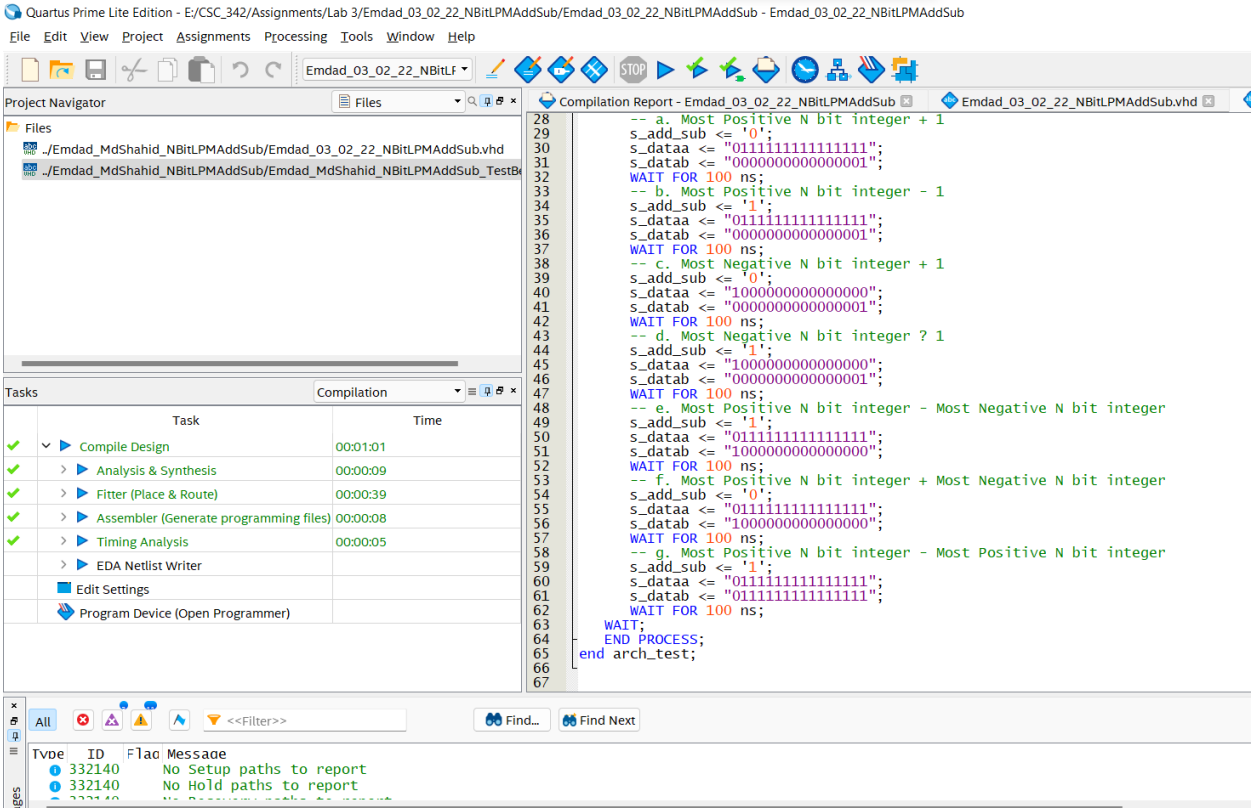


Figure 39: nbitlmpaddsub testbench (Part 2)

10. TASK 10:

In figure 40, we can see the project summary for 16 bit.

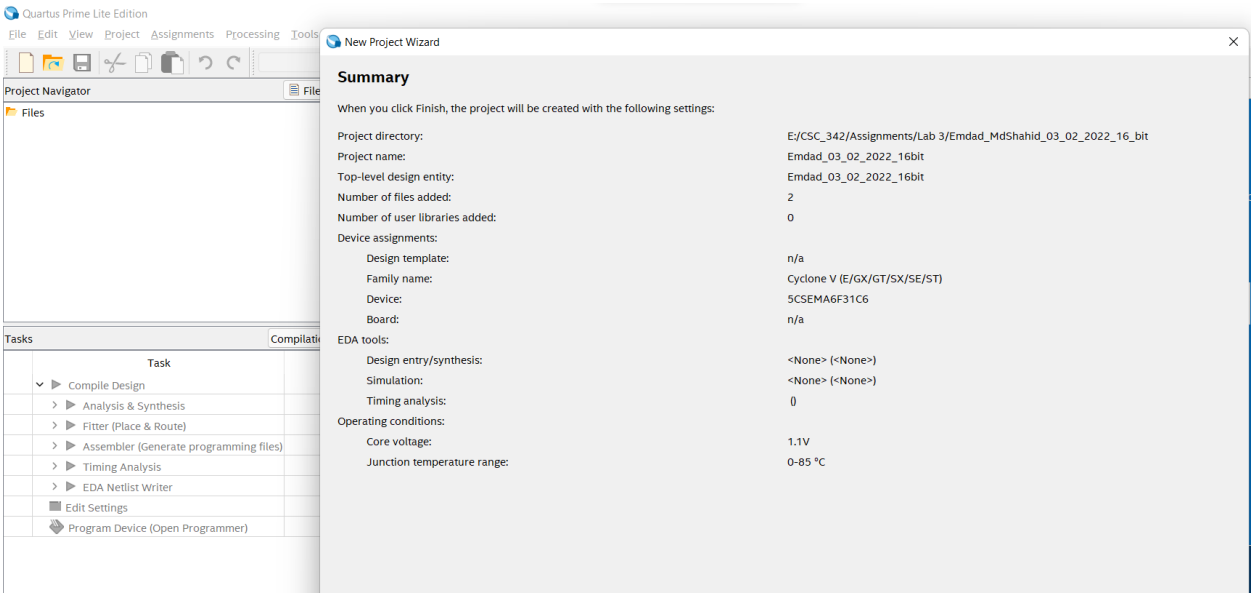


Figure 40: project summary for 16bits

In figure 41, we can see the 16 bits VHDL code.

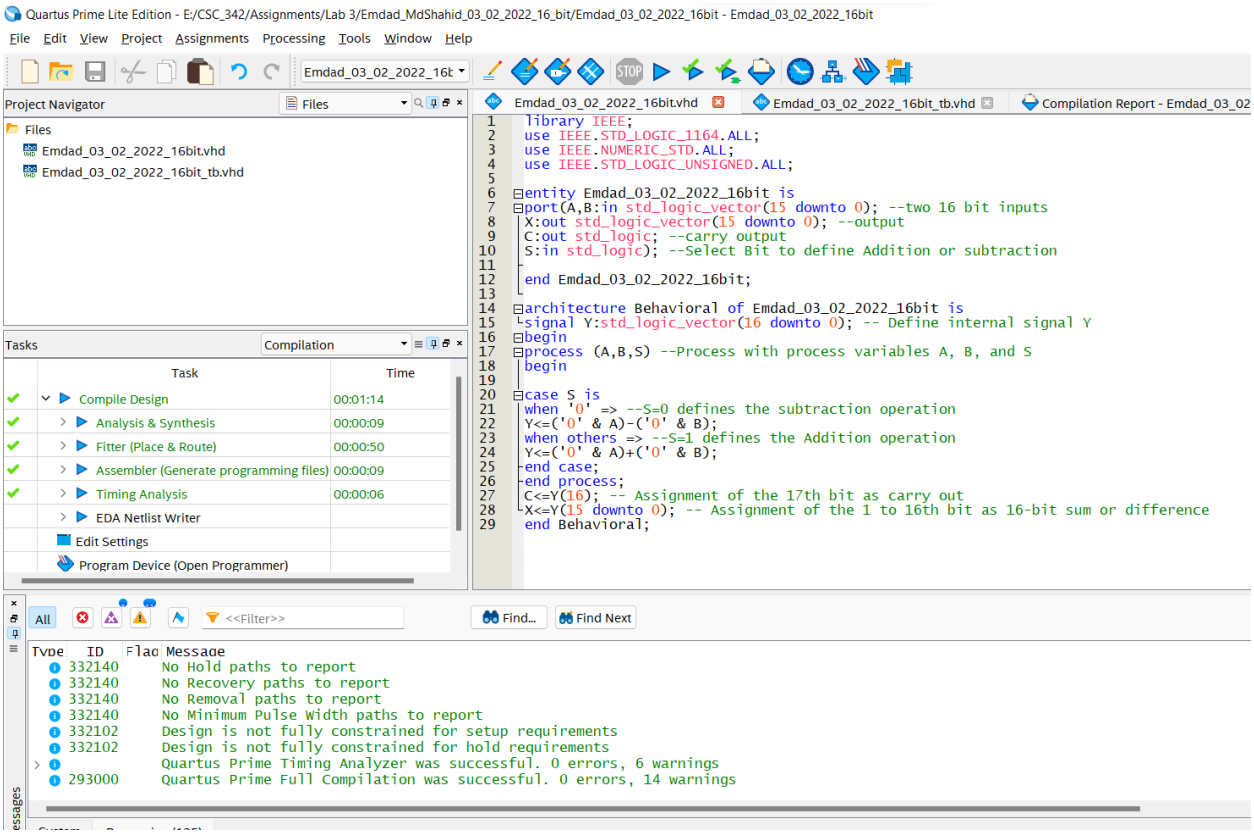


Figure 41: 16-bit VHDL code

In figure 42, we can see the testbench for the 16bit VHDL code to get the waveform.

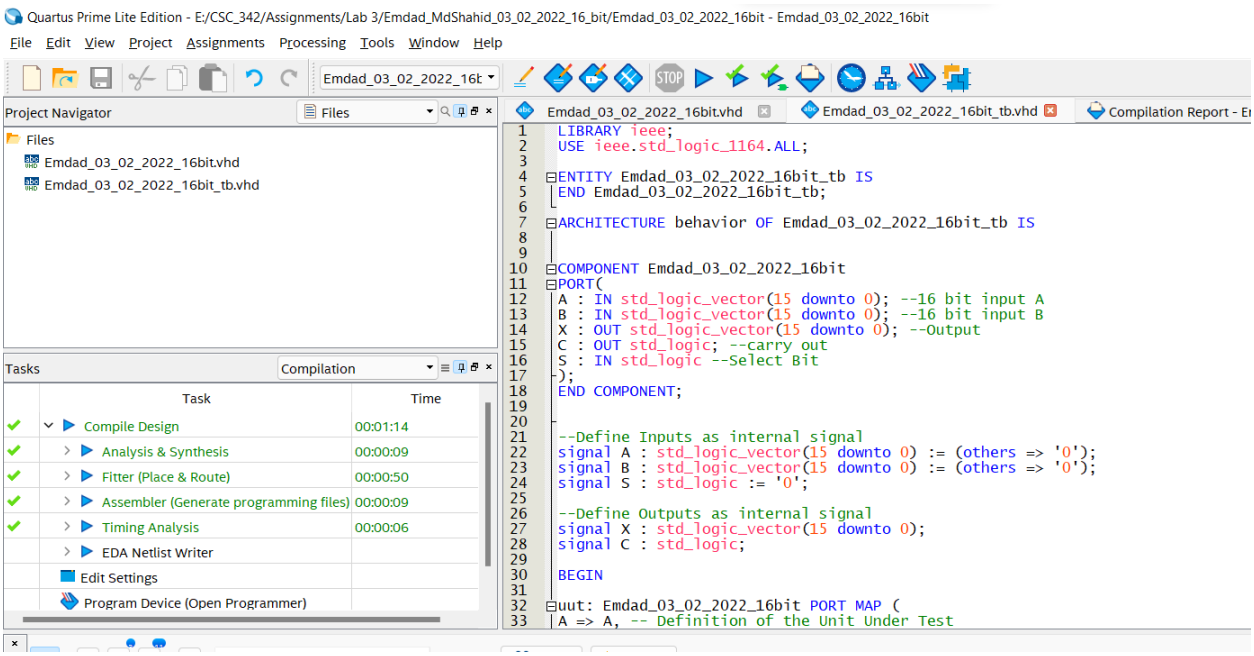


Figure 42: testbench code for 16bit Part 1

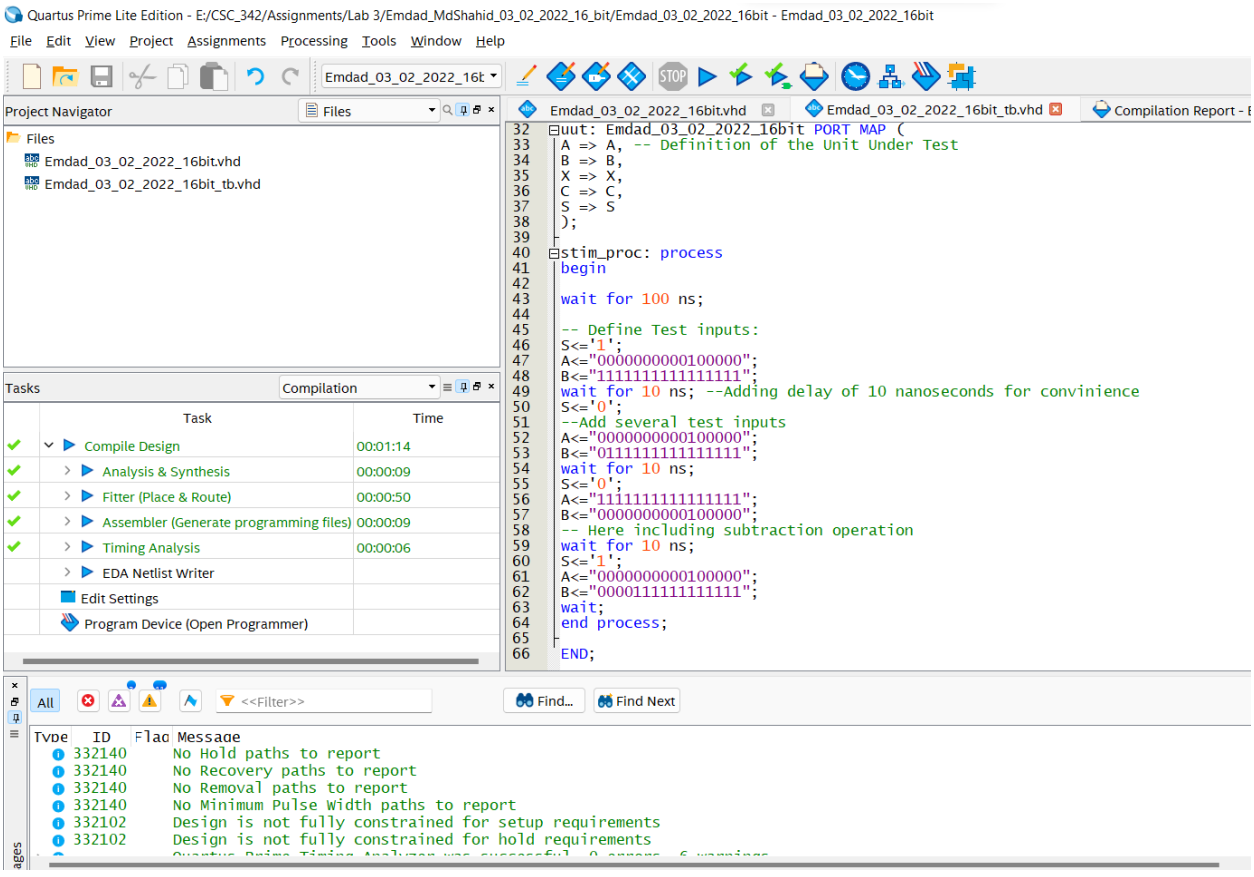


Figure 43: testbench code for 16bit Part 2

Simulation:

4-bit simulation:

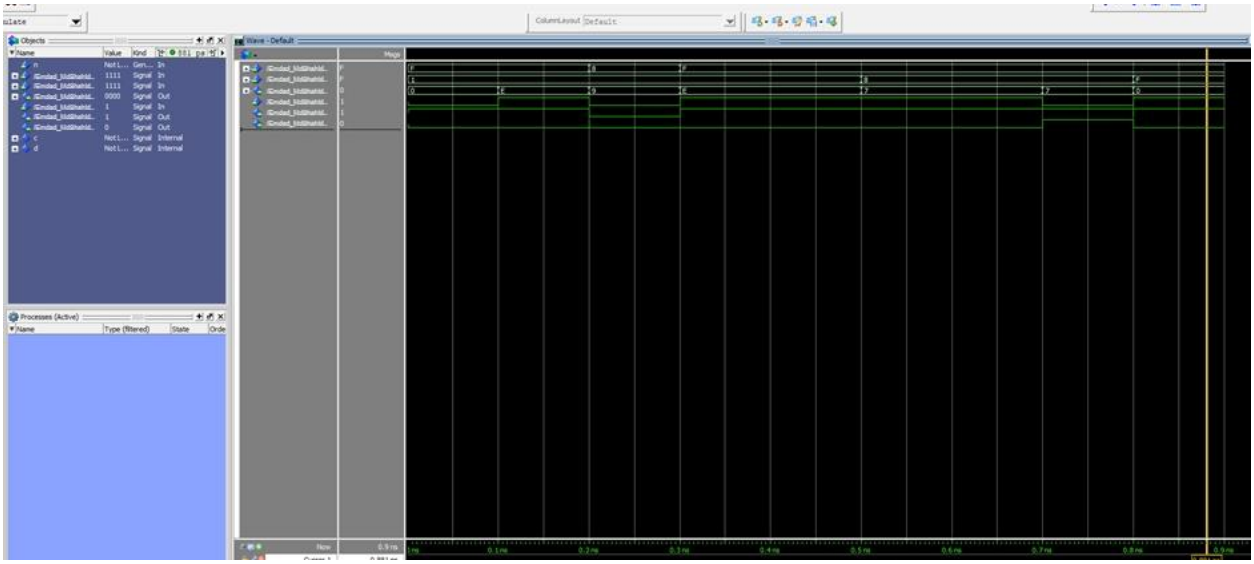


Figure 44: 4-bit simulation

Most negative 4-bit integer: -7 (binary: 0000)

a. Most Positive N bit integer + 1: obit: 0, a = 1111, b = 0001

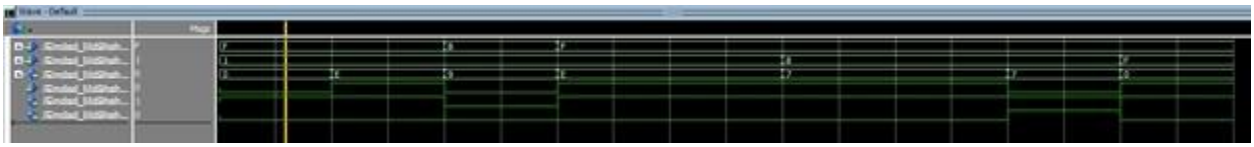


Figure 45: case a

b. Most Positive N bit integer - 1: obit = 1, a = 1111, b = 0001

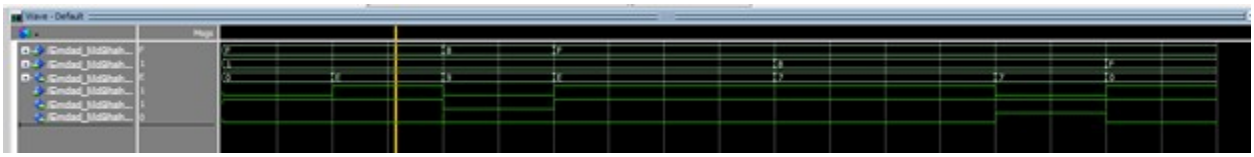


Figure 46: case b

c. Most Negative N bit integer + 1: Obit = 0, a = 1000, b = 0001

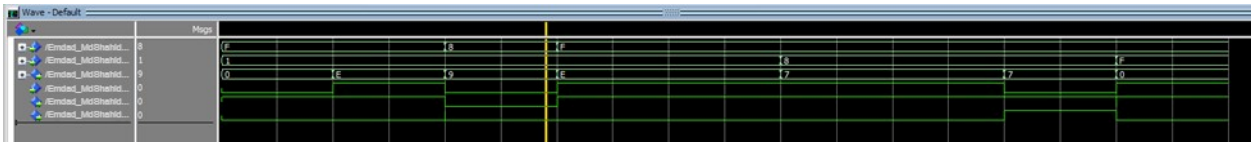


Figure 47: case c

d. Most Negative N bit integer – 1: Obit = 1, a = 1111, b =0001

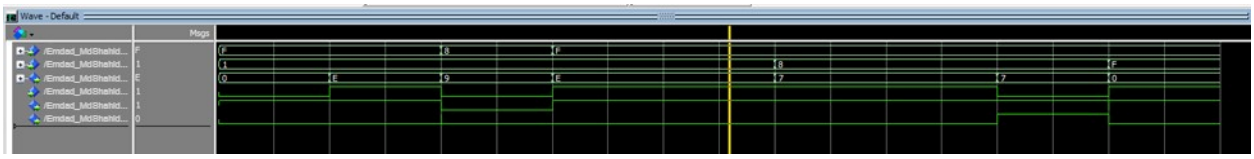


Figure 48: case d

e. Most Positive N bit integer-Most Negative N bit integer: Obit = 1, a = 1111, b =1000

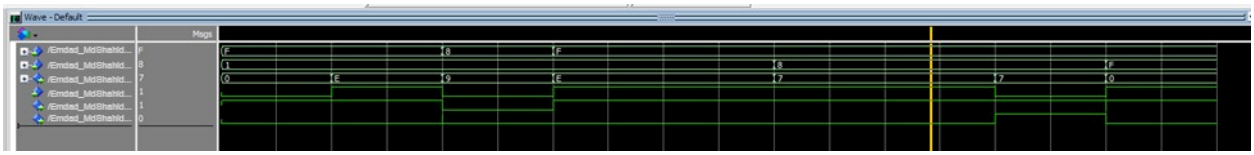


Figure 49: case e

f. Most Positive N bit integer+ Most Negative N bit integer: Obit = 0, a = 1111, b = 1000

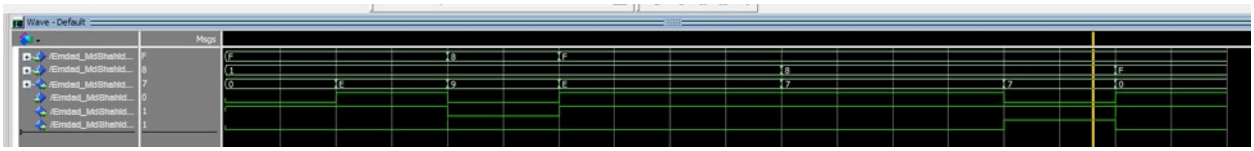


Figure 50: case f

g. Most Positive N bit integer-Most Positive N bit integer: Obit = 1, a =1111, b = 1111

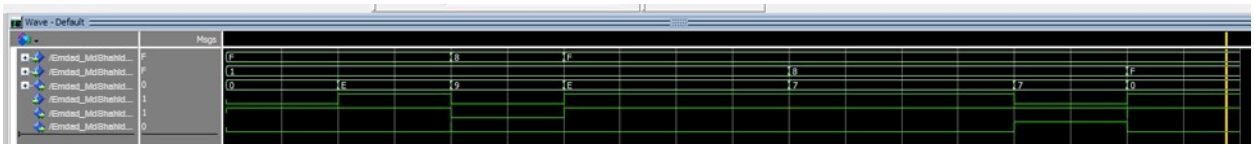


Figure 51: case g

32-bit simulation:

Most 32-bit positive integer: 2,147,483,647 (binary: 01111111111111111111111111111111)
Most 32-bit negative integer: -2,147,483,648 (binary: 10000000000000000000000000000000)

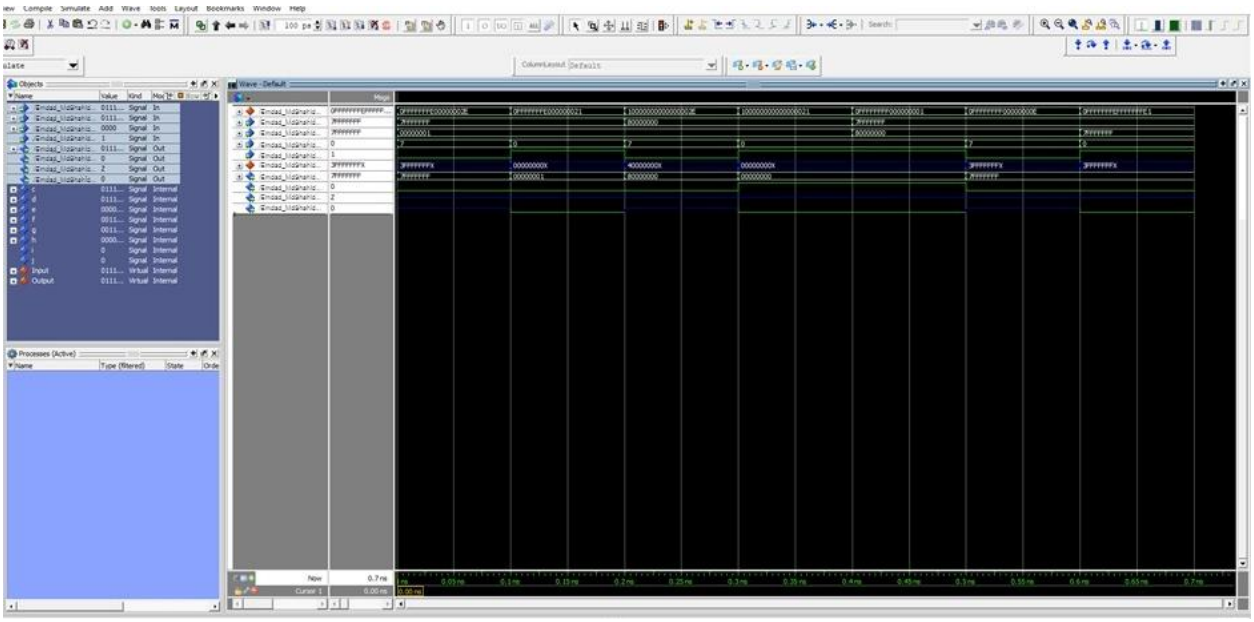


Figure 52: 32-bit simulation

a. Most Positive N bit integer + 1: a = 01111111111111111111111111111111, b = 00000000000000000000000000000001 4bit = 0111, obit = 0

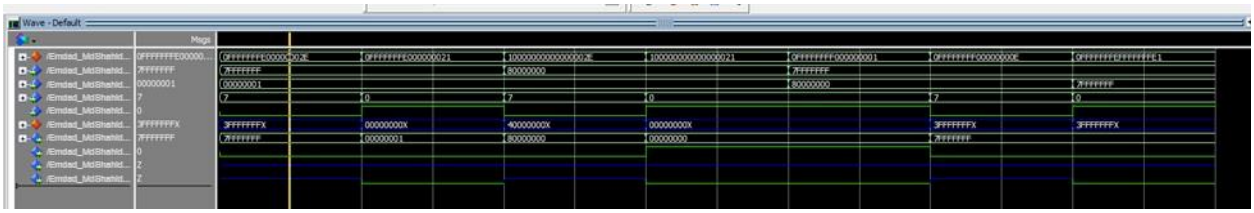


Figure 53: case a

b. Most Positive N bit integer - 1: a = 01111111111111111111111111111111, b = 00000000000000000000000000000001 4bit = 0000, obit = 1

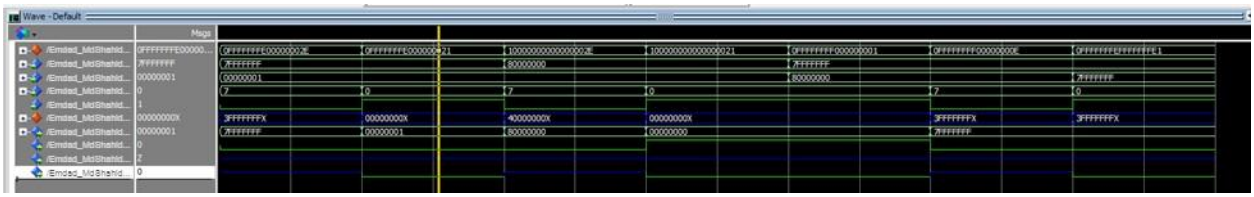


Figure 54: case b

c. Most Negative N bit integer + 1: a = 10000000000000000000000000000000, b = 00000000000000000000000000000001 4bit = 0111, obit = 0

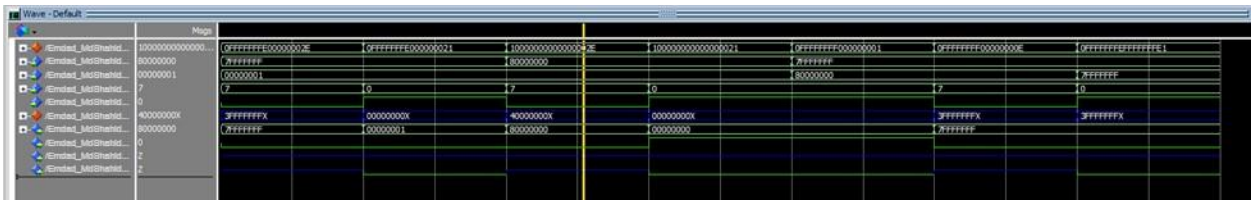


Figure 55: case c

d. Most Negative N bit integer – 1: a = 10000000000000000000000000000000, b = 00000000000000000000000000000001 4bit = 0000, obit = 1

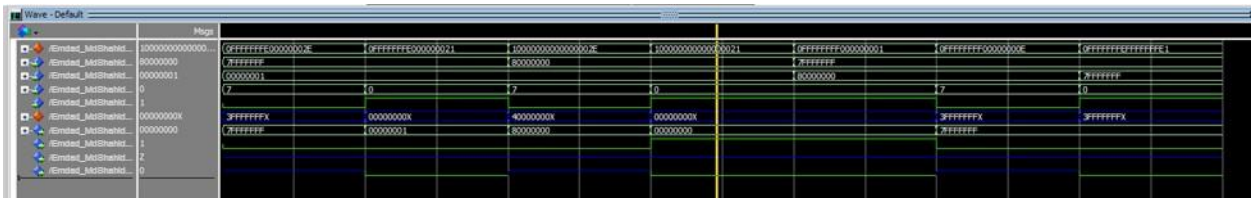


Figure 56: case d

e. Most Positive N bit integer-Most Negative N bit integer: a = 01111111111111111111111111111111, b = 10000000000000000000000000000000 4bit = 0000, obit = 1

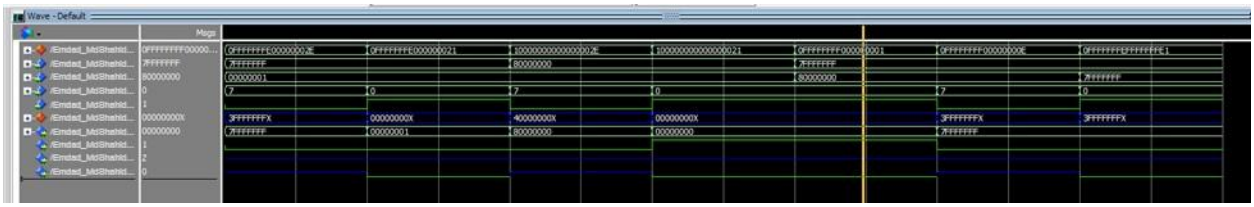


Figure 57: case e

f. Most Positive N bit integer+ Most Negative N bit integer: a = 01111111111111111111111111111111, b = 10000000000000000000000000000000 4bit = 0111, obit = 0

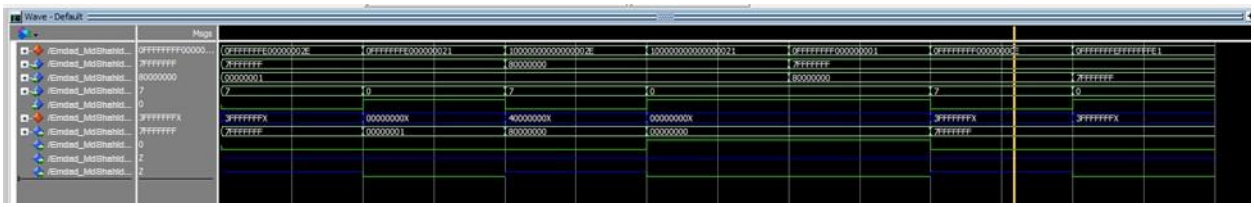


Figure 58: case f

g. Most Positive N bit integer-Most Positive N bit integer: a = 01111111111111111111111111111111, b = 01111111111111111111111111111111 4bit = 0000, obit = 1

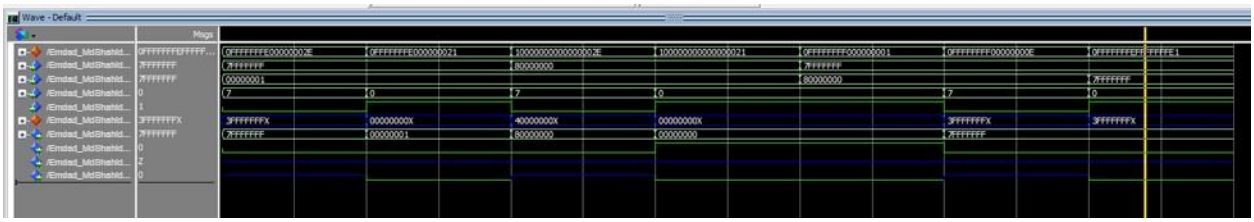


Figure 59: case g

TASK 9:

Simulation with LPM:

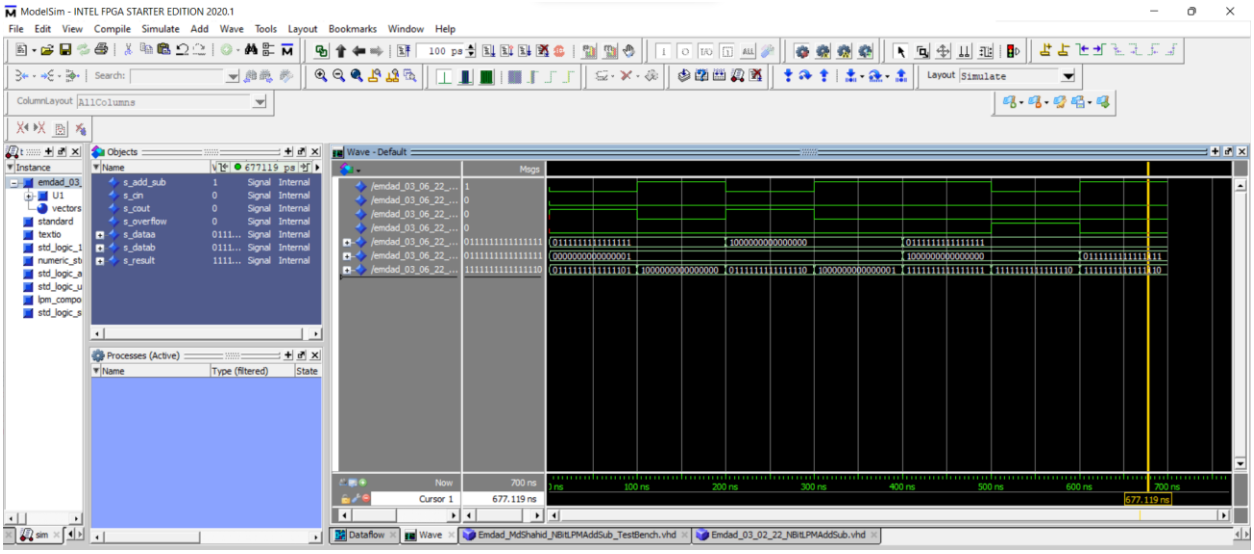


Figure 60: N-bit adder/subtractor unit using lpm simulation

a. Most Positive N bit integer + 1: Obit = 0, a = 1111, b = 0001

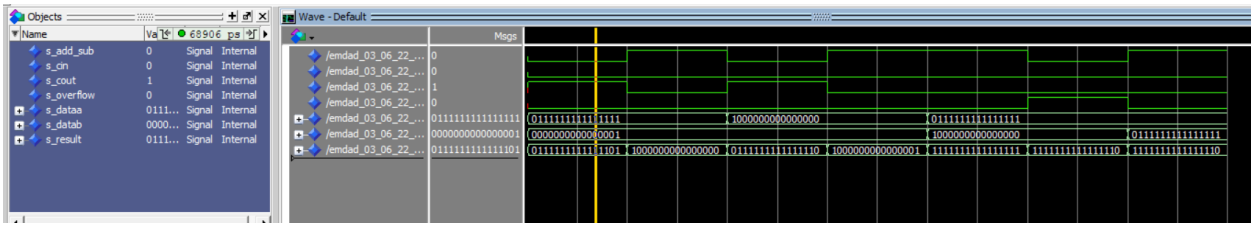


Figure 61: case a

b. Most Positive N bit integer - 1: Obit = 1, a = 1111, b = 0001

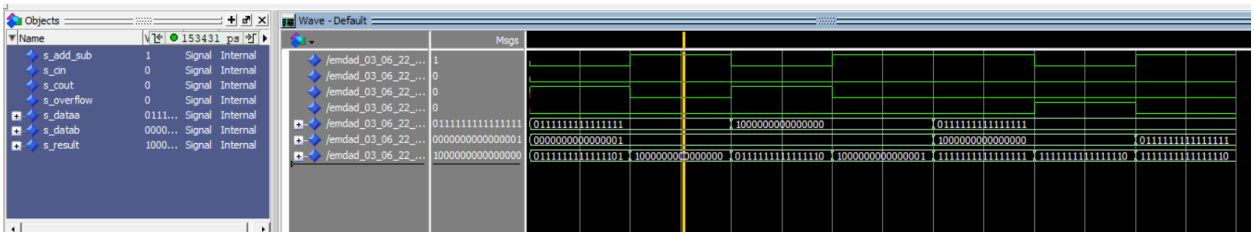


Figure 62: case b

c. Most Negative N bit integer + 1: Obit = 0, a = 1000, b = 0001

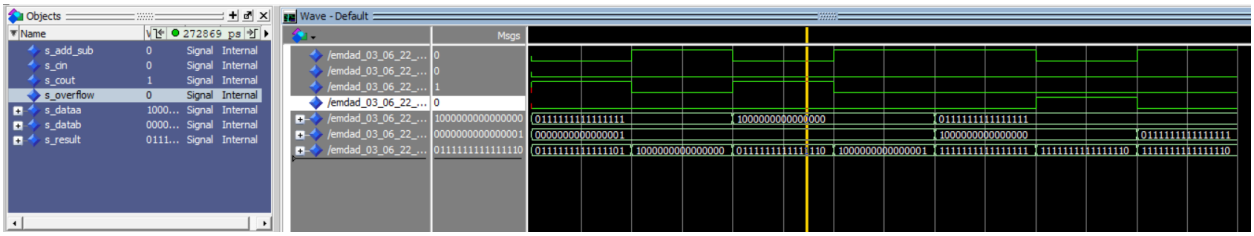


Figure 63: case c

d. Most Negative N bit integer – 1: Obit = 1, a = 1111, b = 0001

Figure 64: case d

e. Most Positive N bit integer-Most Negative N bit integer: Obit = 1, a = 1111, b = 1000

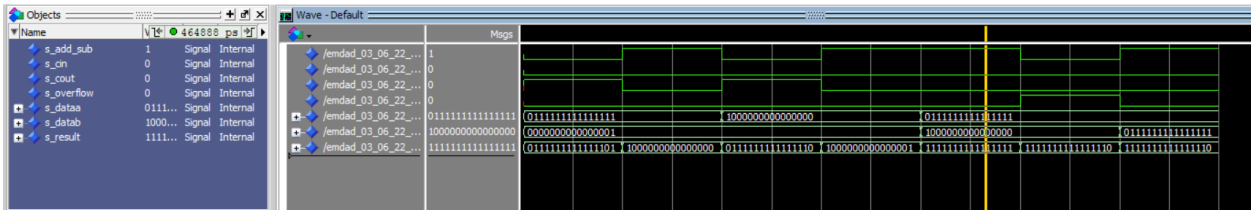


Figure 65: case e

f. Most Positive N bit integer+ Most Negative N bit integer: Obit = 0, a = 1111, b = 1000

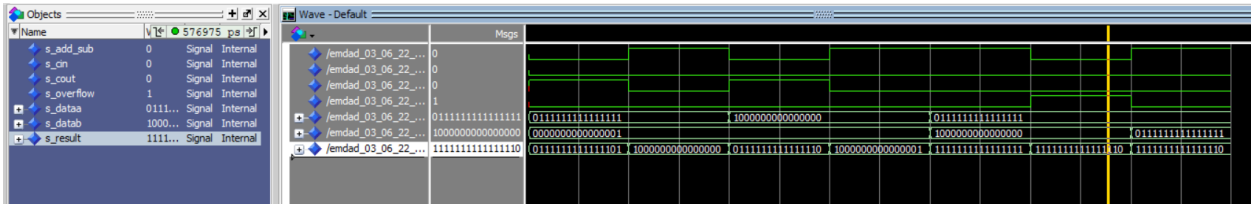


Figure 66: case f

g. Most Positive N bit integer-Most Positive N bit integer: Obit = 1, a = 1111, b = 1111

Figure 67: case g

10. TASK 10:

16 bits simulation

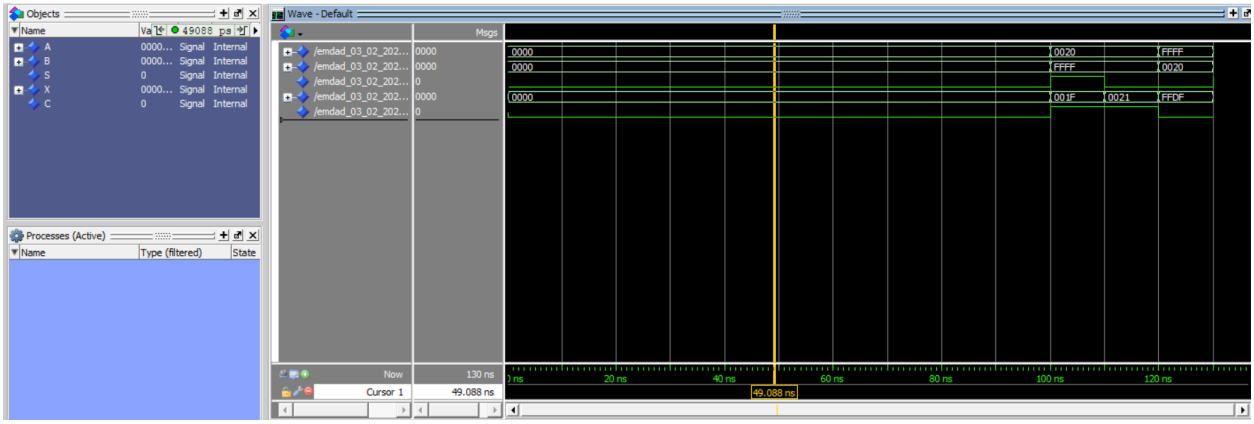


Figure 68: 16bit simulation by testbench without error

In figure 69, the hexadecimal values are different here. Therefore, it shows the error in the simulation.

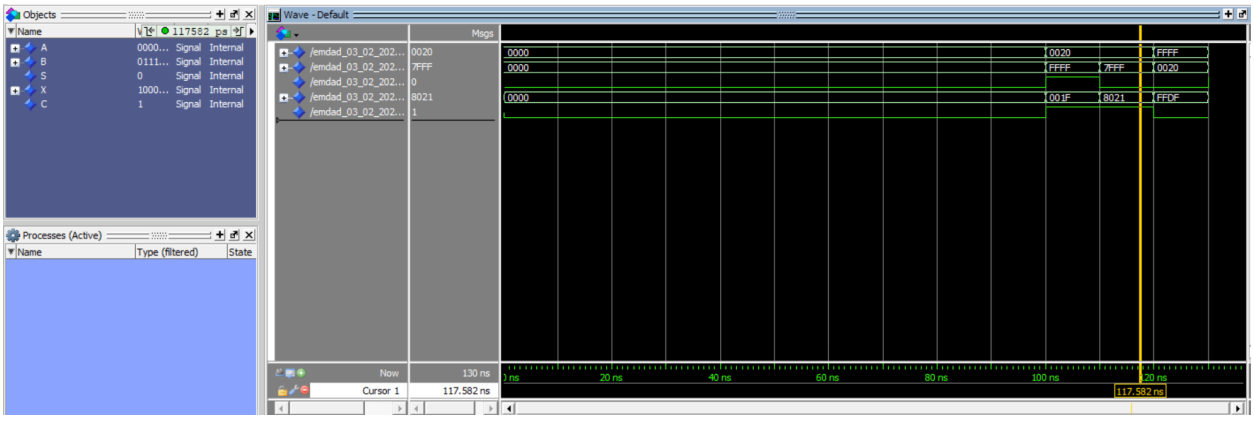


Figure 69: 16bit simulation with error

Conclusion:

In this assignment I learned about the concepts of adders and different features of Quartus and modelsim. I also learned deeply on testbench files, components, port mapping. I also learned to

create my own package. It will help me next time from coding from scratch. This assignment was helpful to learn as it was an introduction to VHDL, design Comparator, and how to test code. It was a good practice on Modelsim for circuit simulation and learn how to simulate comparator in Modelsim. The operations and condition logics were helpful to learn easily. I relearned and reviewed the concept again and which will help me in the course further. Overall, I learned a lot through this lab about adders and VHDL.