CS 342000 | CS343000
Instructor: Professor Izidor Gertner
Spring 2022
Azwad Shameem, 3/2/2022
Laboratory Project 3: Adder Lab

Table of Contents

Objective

The purpose of the adder lab was to design adders and all the intricacies that go along with it. In order to accomplish this task, we designed several adders with different inputs and outputs. For each task, we designed an adder or a testbench for an adder with variations to it.

Description of Specifications and Functionality

Specifications:

1. Shameem_03_06_22_HalfAdder:

Shameem_03_06_22_A: Input A

Shameem_03_06_22_B: Input B

Shameem_03_06_22_Sum: The output from the circuit

Shameem_03_06_22_Carry: The carry output left over by the circuit

2. Shameem_03_06_22_FullAdder:

Shameem_03_06_22_Cin: Input Cin

Shameem_03_06_22_A: Input A

Shameem_03_06_22_B: Input B

Shameem_03_06_22_Sum: The output from the circuit

Shameem_03_06_22_Carry: The carry output left over by the circuit

3. Shameem_03_06_22_FourBitFullAdder

Shameem_03_06_22_Cin: Input Cin

Shameem_03_06_22_A: Input A

Shameem_03_06_22_B: Input B

Shameem_03_06_22_Sum: The output from the circuit

Shameem_03_06_22_Cout: The cout output left over by the circuit

4. Shameem_03_06_22_FourBitFullAddersub

Shameem_03_06_22_Operation: Input operation code

Shameem_03_06_22_A: Input A

Shameem_03_06_22_B: Input B

Shameem_03_06_22_Sum: The output from the circuit

Shameem_03_06_22_Carry: The carry output left over by the circuit

5. Shameem_03_06_22_Package

A collection of all the components in the lab

6. Shameem_03_06_22_NBitFullAdderSub

Shameem_03_06_22_Operation: Input operation code

Shameem_03_06_22_A: Input A

Shameem_03_06_22_B: Input B

Shameem_03_06_22_Sum: The output from the circuit

Shameem_03_06_22_Cout: The cout output left over by the circuit

7. Shameem_03_06_22_NBitFullAdderSubFlags

Shameem_03_06_22_Operation: Input operation code

Shameem_03_06_22_A: Input A

Shameem_03_06_22_B: Input B

Shameem_03_06_22_Sum: The output from the circuit

Shameem_03_06_22_Cout: The cout output left over by the circuit

Shameem_03_06_22_Overflow: If the operation had an overflow

Shameem_03_06_22_Negative: If the operation ended up negative

Shameem_03_06_22_Zero: If the operation ended up as a negative

8. Shameem_03_06_22_NBitLPMAddSub

Shameem_03_06_22_Operation: Input operation code

Shameem_03_06_22_A: Input A

Shameem_03_06_22_B: Input B

Shameem_03_06_22_Sum: The output from the circuit

Shameem_03_06_22_Cout: The cout output left over by the circuit

Functionality:

Task 1: Design in VHDL Half adder using two processes

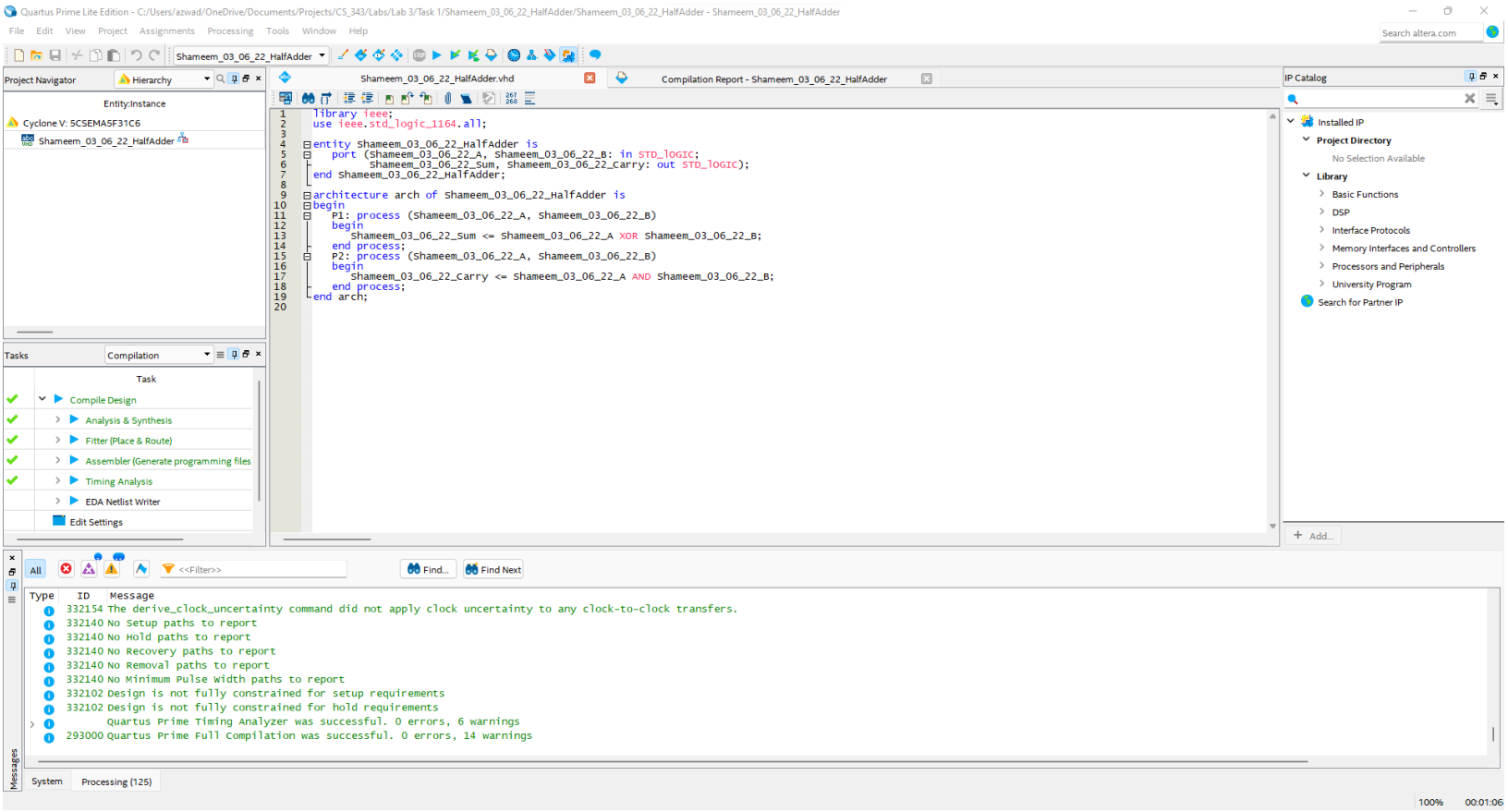


Figure 1: Half Adder VHDL Code

Half Adder utilizes two processes as required in order to compute. The circuit has inputs A and B and outputs the sum as the result and a carry output. The circuit also has compiled successfully in the Quartus application.

Task 3: Design 1-bit Full-adder using Half adder as a component

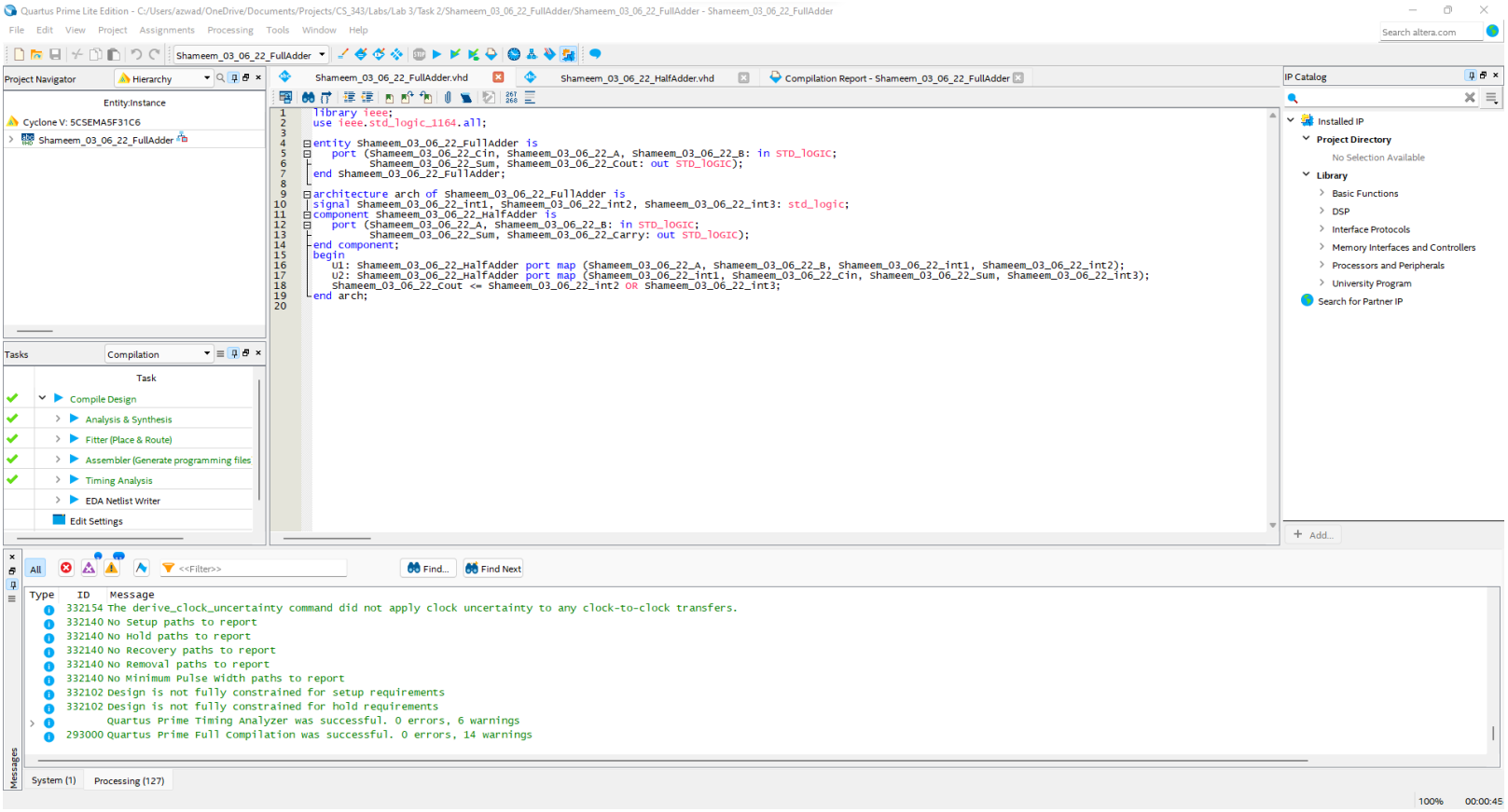


Figure 2: Full Adder VHDL Code

The 1-bit Full Adder circuit utilizes two Half Adders as a component. The Full adder has the same inputs and outputs as the Half adder with the exception of an additional input called Cin over the Half Adder.

Task 3: Design N = 4 bit adder using 1-bit Full adder as a component

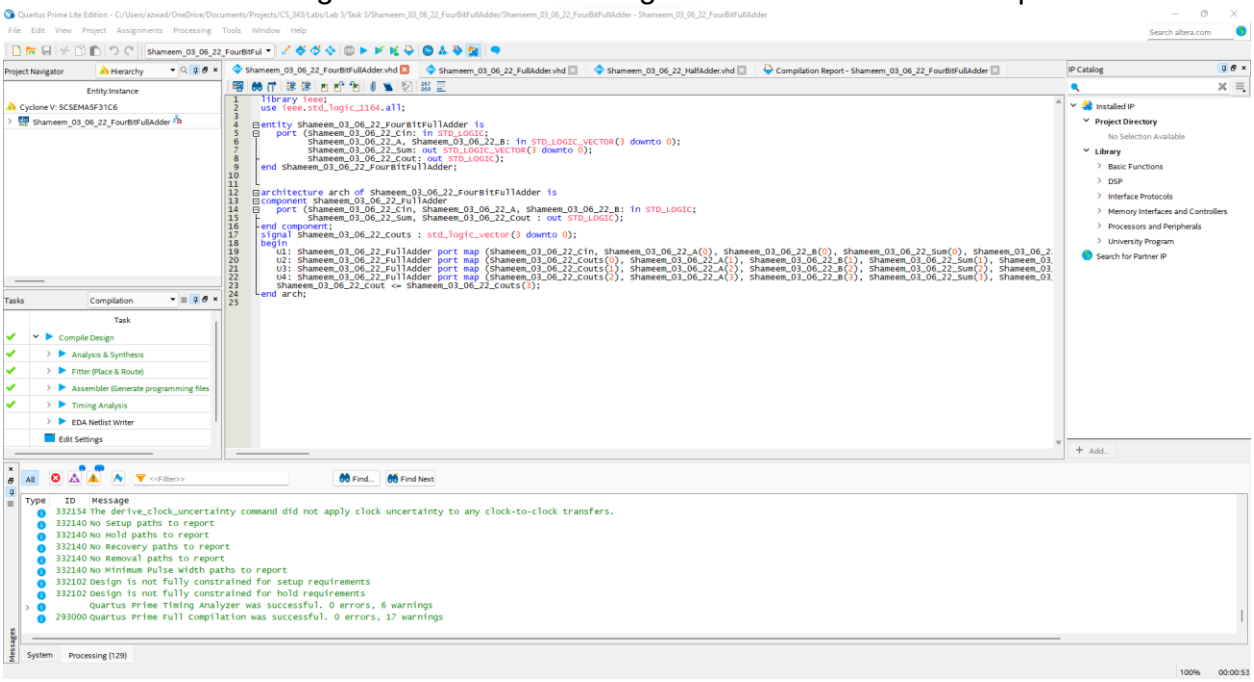


Figure 3: The 4-bit Full Adder VHDL code

The 4-bit Full Adder utilizes four full adders in order to become a four bit adder circuit. The 4-bit Full adder also has the same inputs as the Full Adder with the exception that inputs A and B are now vectors that contain a 4 bit binary instead of a single bit. Also the 4-bit Full Adder and the included components utilized in the 4-bit Full Adder code compiled and ran successfully on Quartus in the image above.

Task 4: Design N = 4 bit a Add/Sub components that performs addition when the operations code = 0, and subtraction when the operations code = 1

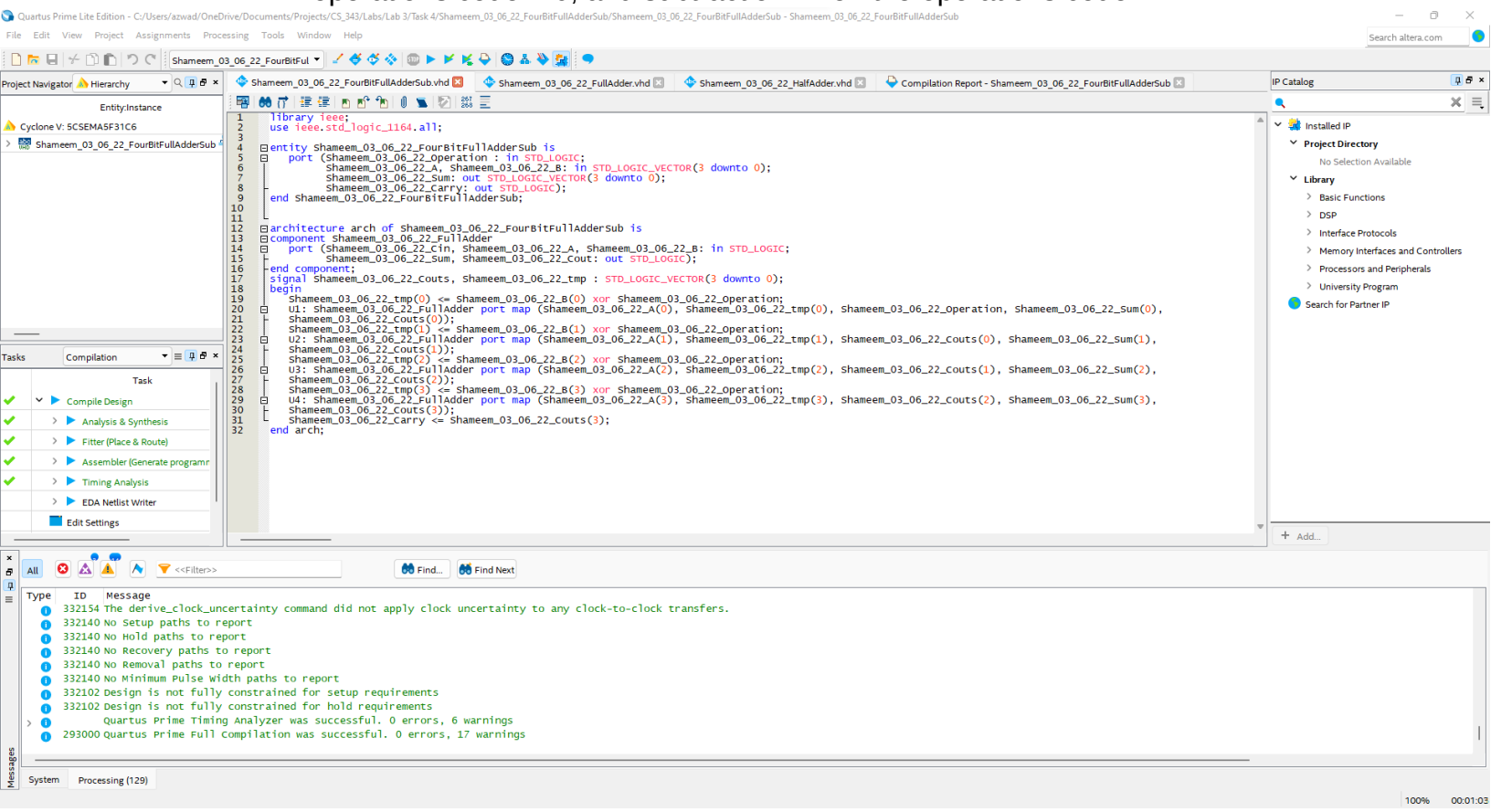


Figure 4: Four bit Full Adder/sub VHDL code

The four bit Full Adder/sub is a circuit that computes addition or subtraction depending on the operation code input. The four bit Full Adder/sub also has a variance to its input compared to the other circuit which is the operation input. If the operations code input is zero, then it adds otherwise if the operations code input is one then it computes subtraction. The four bit full Adder/sub also uses a vector of 4 bits and therefore it utilizes four full Adders as components in order to compute addition or subtraction. The four bit Full Adder/sub compiles and all of the components utilized in the circuit also compiled successfully on the Quartus application.

Task 5: Create a package where you put all components for future use

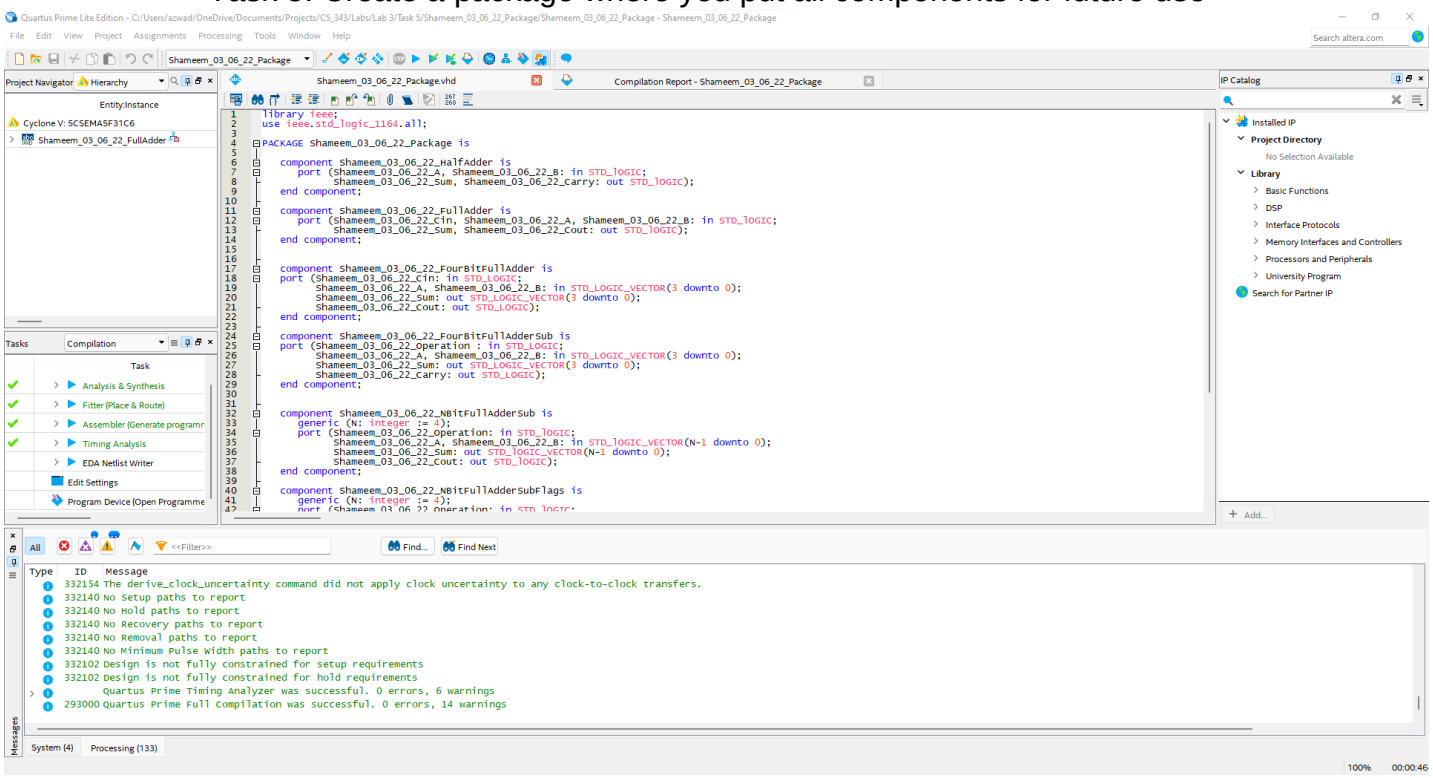


Figure 4: VHDL Code for Package

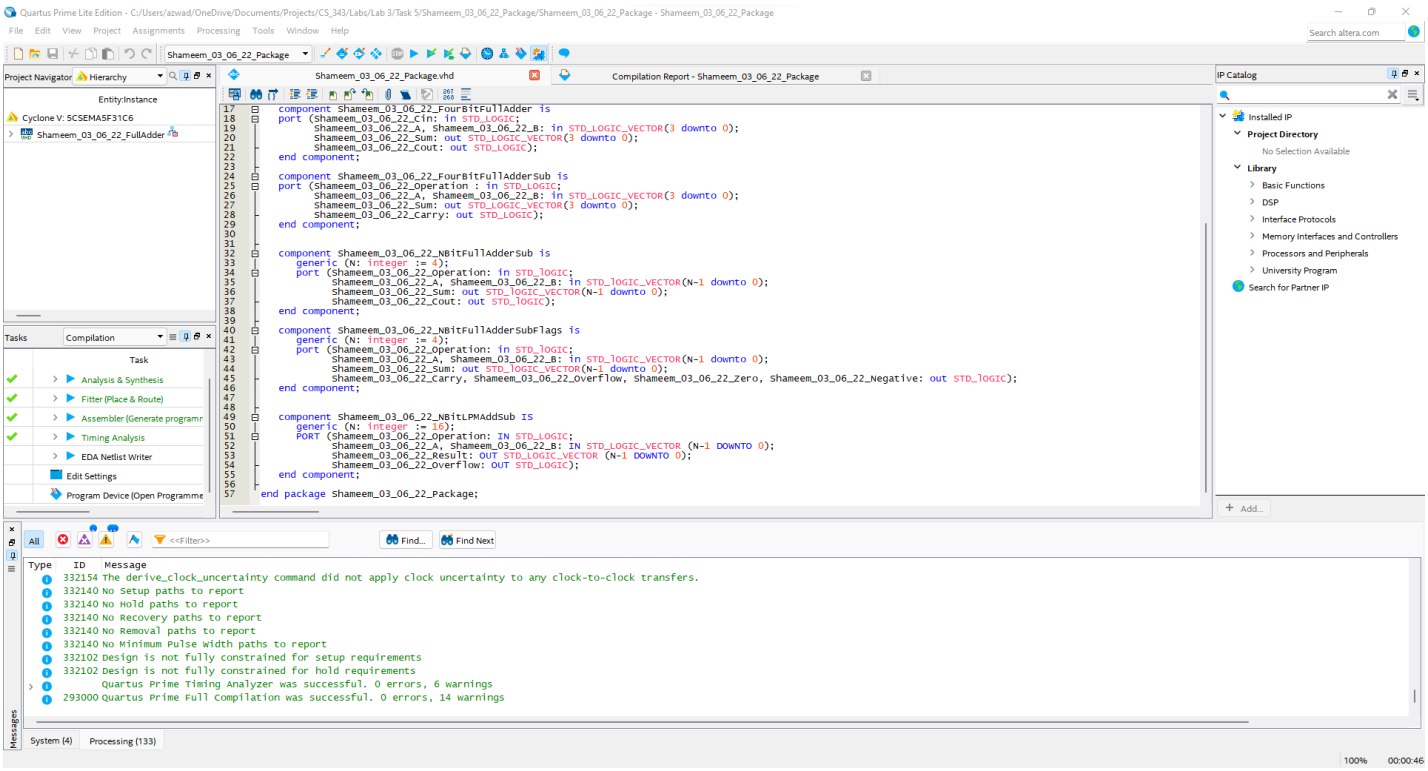


Figure 5: VHDL Code for Package Continued

The Package VHDL code just has all the components used in the lab.

Task 6: Design N – bit add-sub unit using behavioral VHDL model

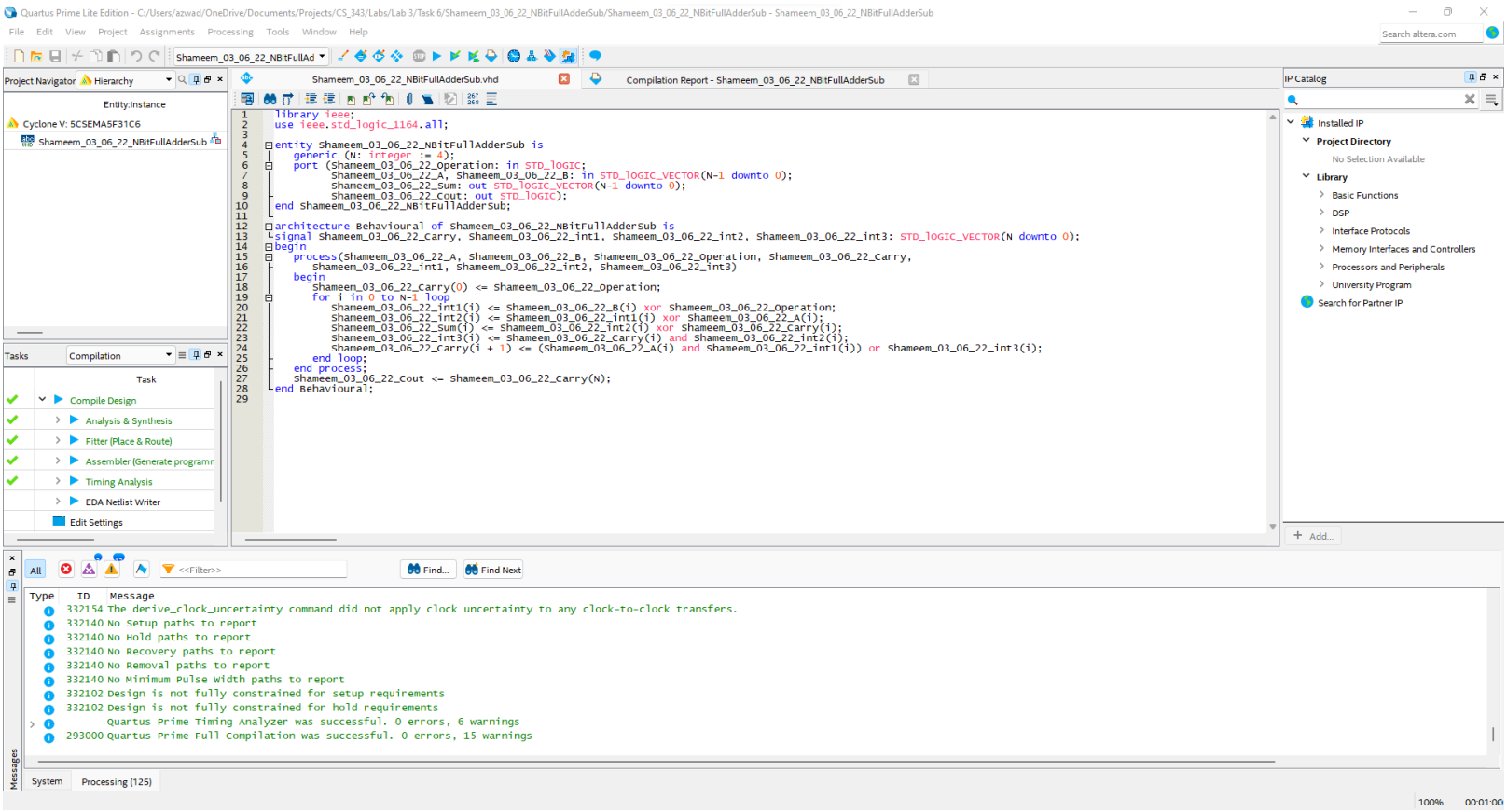


Figure 6: N-Bit Full AdderSub behavioral VHDL code

The N-bit Full AdderSub circuit utilizes N as a integer to make a circuit that is able to compute addition or subtraction for any N bits. The N-bit Full AdderSub circuit utilizes a operation code input and inputs A and B and output Sum to compute addition or subtraction for N bits. The N-bit Full AdderSub circuit utilizes behavioral VHDL programming only by using a for loop to basically compute the logic for a Full Adder for 0 to N-1 times in order to compute addition or subtraction for N bits.

Task 7: You have to design a circuit to output overflow, zero, negative flags for N-bit add/sub.

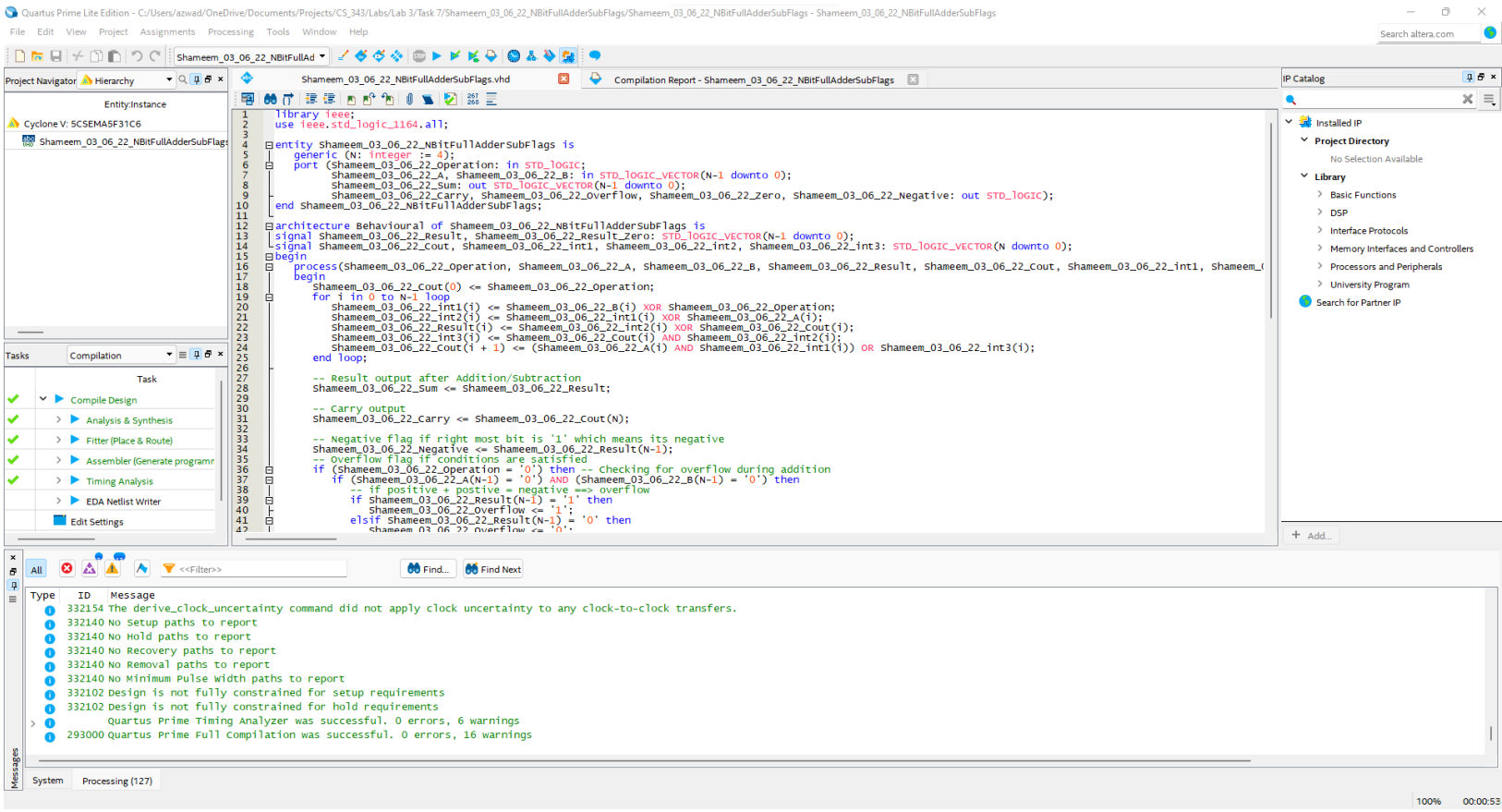


Figure 7: VHDL Code for N-Bit Full Adder/Sub with flags

The N-Bit Full Adder/Sub with flags circuit utilizes the same logic from the N-Bit Full Adder/Sub circuit but adds an additional three outputs called Zero, Negative, Overflow. The logic from the original N-Bit Full Adder/Sub circuit utilizes the behavioral logic for a one bit Full Adder and repeats the logic for 0 to N times in order to compute addition or subtraction for N bits. The logic to check the negative flag is if the output's leftmost bit is 1 it means that the signed binary version of that number is negative. Therefore, if the sum outputs leftmost bit is negative, meaning the leftmost bit is 1, that means we will call the negative flag as 1 because the output is negative.

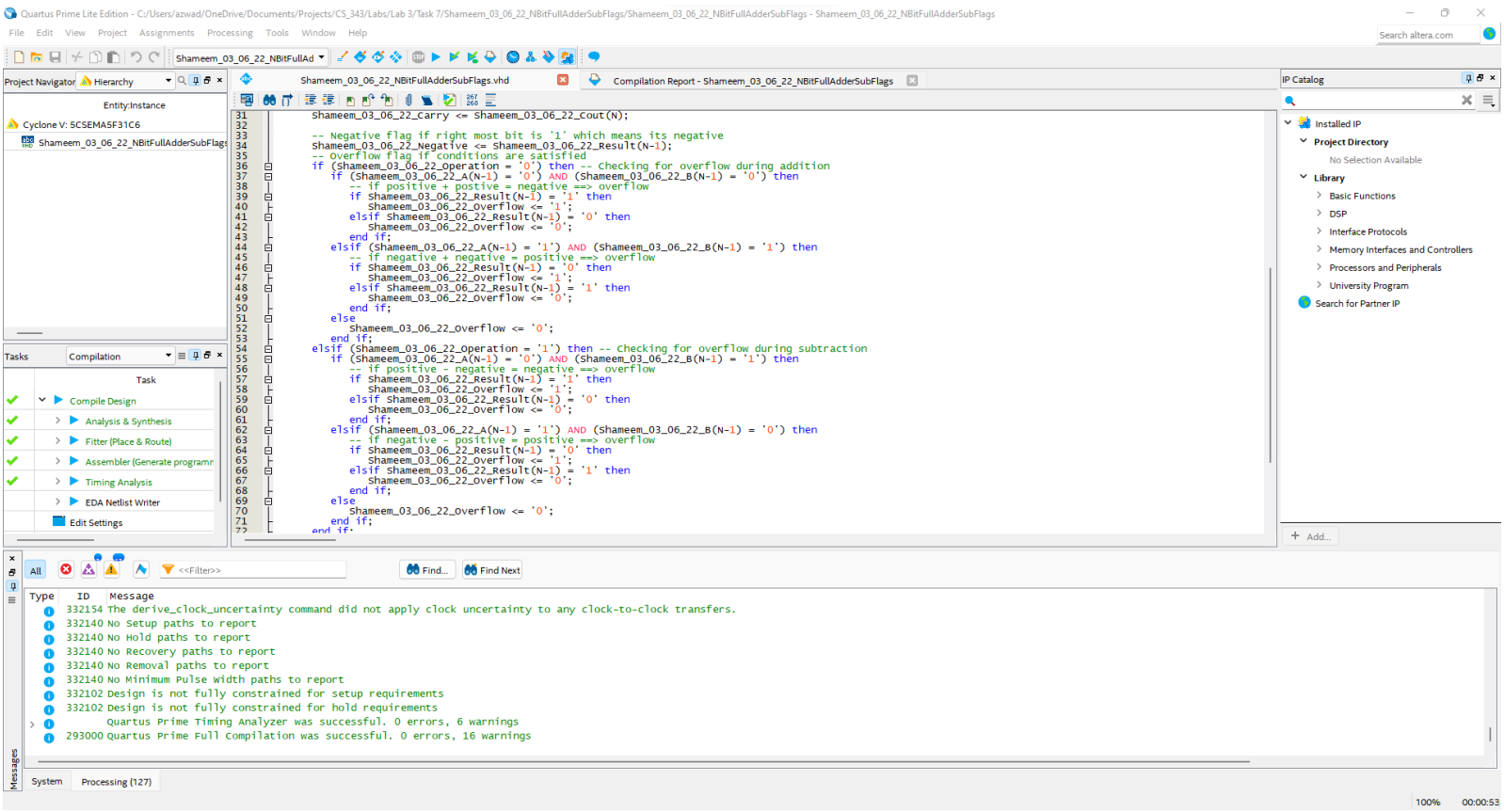
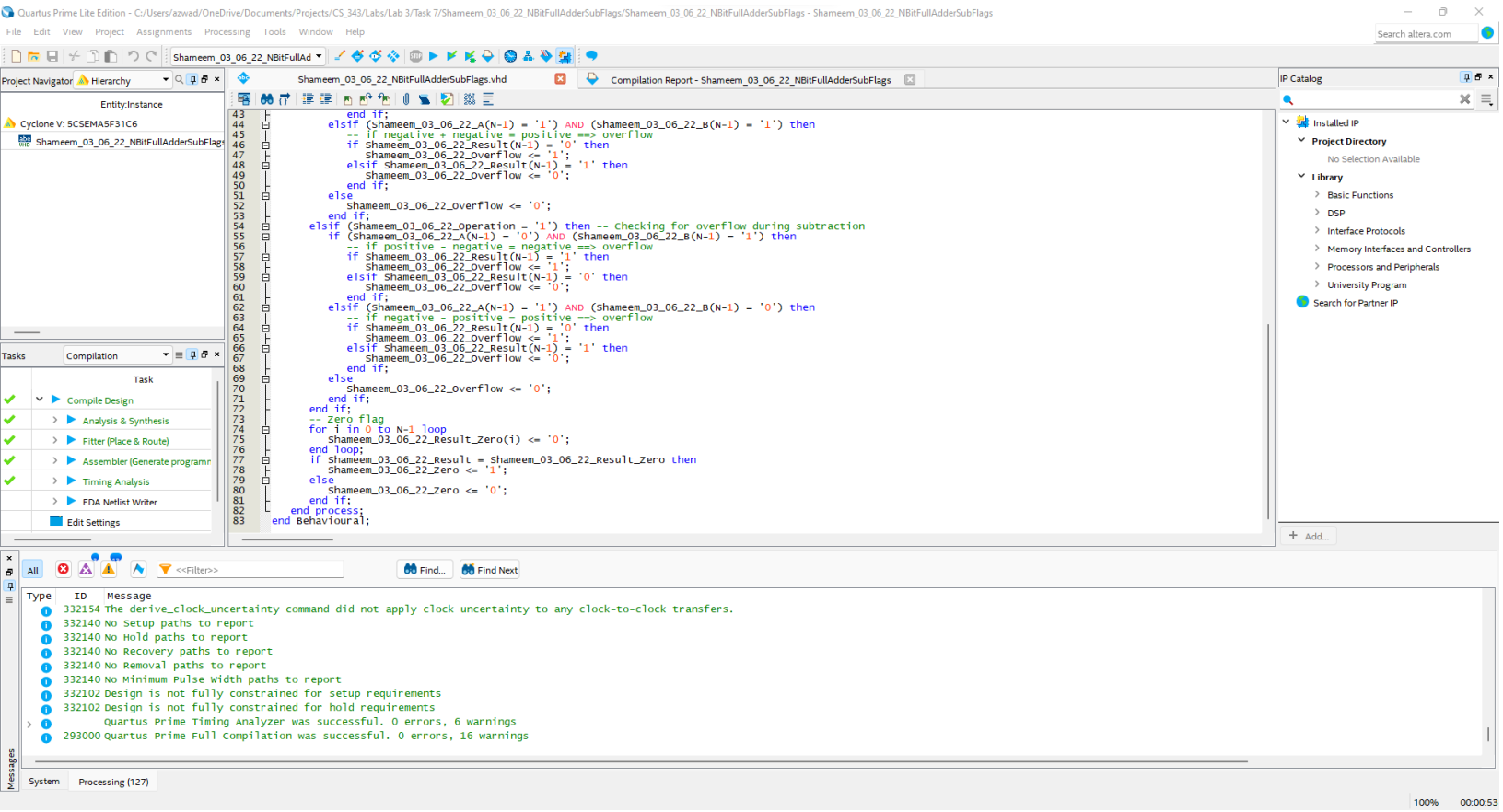


Figure 8: VHDL code for N Bit Full Adder/Sub with Flags Continued

The overflow flag is only 1 when certain conditions are satisfied. The conditions for addition, when the operation code is equal to zero, are the following, when two positives adding equal a negative or when two negatives adding equals a positive. If these cases for addition are satisfied the overflow flag will be trigged to one because these events should not happen unless there has been an overflow error. The conditions for negatives, when the operations code is equal to one, are the following, when one positive minus a negative equal a negative or when a negative minus a positive equal a positive. If these cases for subtraction are satisfied then the overflow flag is trigged and the value of the overflow flag will be shown as one in the simulation.

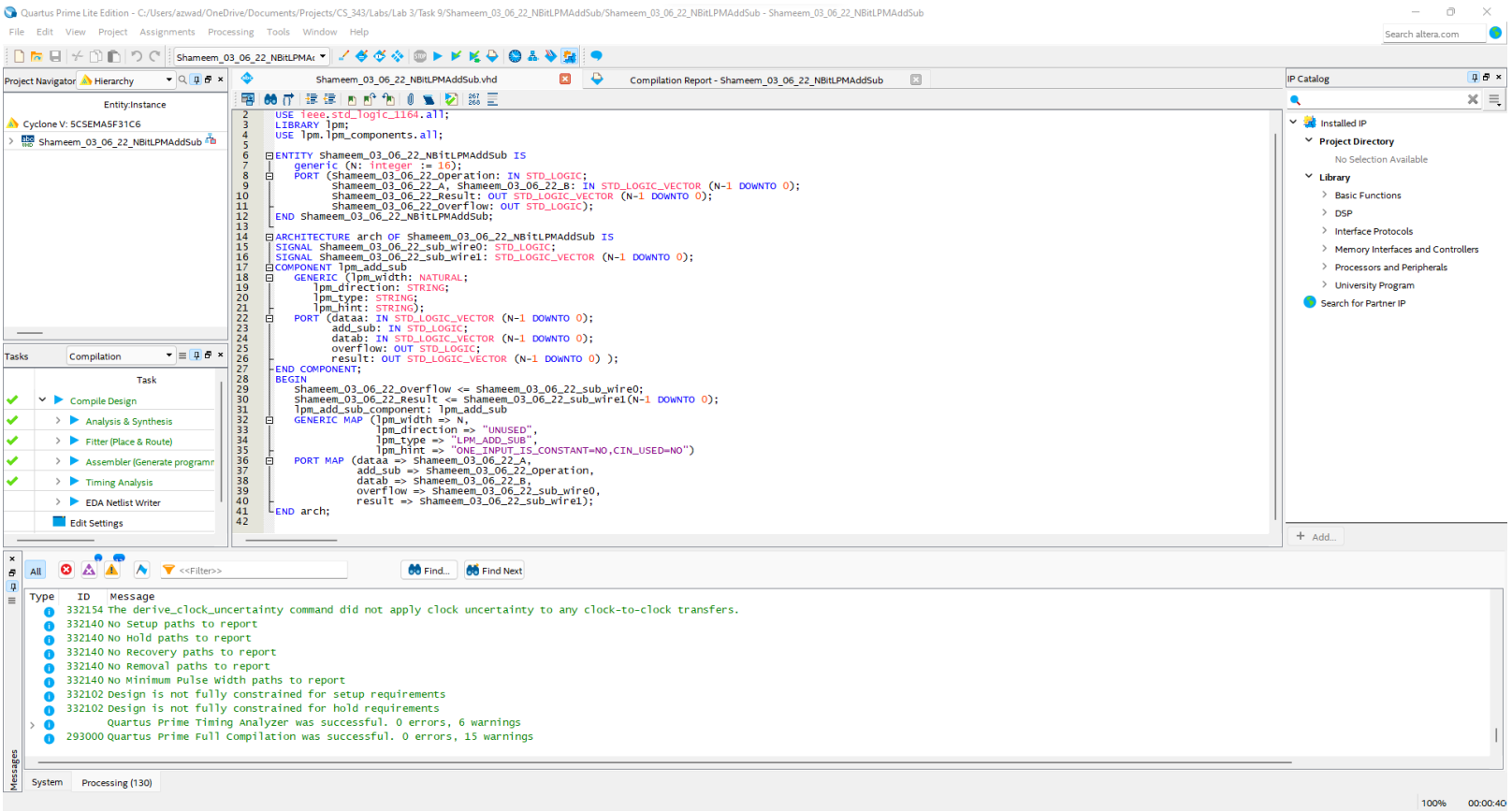



```
43            end if;
44         elsif (Shameem_03_06_22_A(N-1) = '1') AND (Shameem_03_06_22_B(N-1) = '1') then
45            -- if negative + negative = positive ==> overflow
46            if Shameem_03_06_22_Result(N-1) = '0' then
47               Shameem_03_06_22_Overflow <= '1';
48            elsif Shameem_03_06_22_Result(N-1) = '1' then
49               Shameem_03_06_22_Overflow <= '0';
50            end if;
51         else
52            Shameem_03_06_22_Overflow <= '0';
53         end if;
54      elsif (Shameem_03_06_22_Operation = '1') then -- Checking for overflow during subtraction
55         if (Shameem_03_06_22_A(N-1) = '0') AND (Shameem_03_06_22_B(N-1) = '1') then
56            -- if positive - negative = negative ==> overflow
57            if Shameem_03_06_22_Result(N-1) = '1' then
58               Shameem_03_06_22_Overflow <= '1';
59            elsif Shameem_03_06_22_Result(N-1) = '0' then
60               Shameem_03_06_22_Overflow <= '0';
61            end if;
62         elsif (Shameem_03_06_22_A(N-1) = '1') AND (Shameem_03_06_22_B(N-1) = '0') then
63            -- if negative - positive = positive ==> overflow
64            if Shameem_03_06_22_Result(N-1) = '0' then
65               Shameem_03_06_22_Overflow <= '1';
66            elsif Shameem_03_06_22_Result(N-1) = '1' then
67               Shameem_03_06_22_Overflow <= '0';
68            end if;
69         else
70            Shameem_03_06_22_Overflow <= '0';
71         end if;
72      end if;
73      -- Zero flag
74      for i in 0 to N-1 loop
75         Shameem_03_06_22_Result_Zero(i) <= '0';
76      end loop;
77      if Shameem_03_06_22_Result = Shameem_03_06_22_Result_Zero then
78         Shameem_03_06_22_Zero <= '1';
79      else
80         Shameem_03_06_22_Zero <= '0';
81      end if;
82   end process;
83 end Behavioural;
```

Figure 9: VHDL Code for N-Bit Full Adder/Sub with flags Continued

The zero flag for the N-Bit Full Adder/Sub with flags is quite simple, we make a N-1 downto 0 vector that is equal to 0 for the N bits and then we check if the result of the circuit is equal to that vector. If the results of the circuit is equal to the zero vector than pretty much we know that the output is zero and the zero flag shall be triggered as one in the simulation. Otherwise, if it's not equal we do not trigger the zero flag and it will stay as 0.

Task 9: Create N-bit adder/subtractor unit using lpm




```
USE ieee.std_logic_1164.all;
LIBRARY lpm;
USE lpm.lpm_components.all;

ENTITY Shameem_03_06_22_NBitLPMAddSub IS
   generic (N: integer := 16);
   PORT (Shameem_03_06_22_Operation: IN STD_LOGIC;
         Shameem_03_06_22_A, Shameem_03_06_22_B: IN STD_LOGIC_VECTOR (N-1 DOWNTO 0);
         Shameem_03_06_22_Result: OUT STD_LOGIC_VECTOR (N-1 DOWNTO 0);
         Shameem_03_06_22_Overflow: OUT STD_LOGIC);
END Shameem_03_06_22_NBitLPMAddSub;

ARCHITECTURE arch OF Shameem_03_06_22_NBitLPMAddSub IS
SIGNAL Shameem_03_06_22_sub_wire0: STD_LOGIC;
SIGNAL Shameem_03_06_22_sub_wire1: STD_LOGIC_VECTOR (N-1 DOWNTO 0);
COMPONENT lpm_add_sub
   GENERIC (lpm_width: NATURAL;
       lpm_direction: STRING;
       lpm_type: STRING;
       lpm_hint: STRING);
   PORT (dataa: IN STD_LOGIC_VECTOR (N-1 DOWNTO 0);
         add_sub: IN STD_LOGIC;
         datab: IN STD_LOGIC_VECTOR (N-1 DOWNTO 0);
         overflow: OUT STD_LOGIC;
         result: OUT STD_LOGIC_VECTOR (N-1 DOWNTO 0) );
END COMPONENT;
BEGIN
   Shameem_03_06_22_Overflow <= Shameem_03_06_22_sub_wire0;
   Shameem_03_06_22_Result <= Shameem_03_06_22_sub_wire1(N-1 DOWNTO 0);
   lpm_add_sub_component: lpm_add_sub
   GENERIC MAP (lpm_width => N,
                lpm_direction => "UNUSED",
                lpm_type => "LPM_ADD_SUB",
                lpm_hint => "ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO")
   PORT MAP (dataa => Shameem_03_06_22_A,
             add_sub => Shameem_03_06_22_Operation,
             datab => Shameem_03_06_22_B,
             overflow => Shameem_03_06_22_sub_wire0,
             result => Shameem_03_06_22_sub_wire1);
END arch;
```

Figure 10: VHDL code for LPM N-bit Adder/subtractor unit

The requirement for the LPM N-bit Adder/subtractor unit is basically to utilize the LPM library and make a N-bit Adder/subtractor utilizing the library. The LPM library very helpful in making this circuit because it removes most of the need for creating logic and allows us to directly use the addsub circuit from the LPM library in order to make a adder/subtractor unit. Then we can use N as a generic integer in the LPM's width to make the adder/subtractor from LPM work for N bits.

Task 8: Verify your design in simulation using waveforms in ModelSim for N=4, and N=32 bits using Most positive, Most negative integer as a first operand, and integers 1 and/or 2 as a second operand. You have to demonstrate that flags are set correctly in appropriate cases.

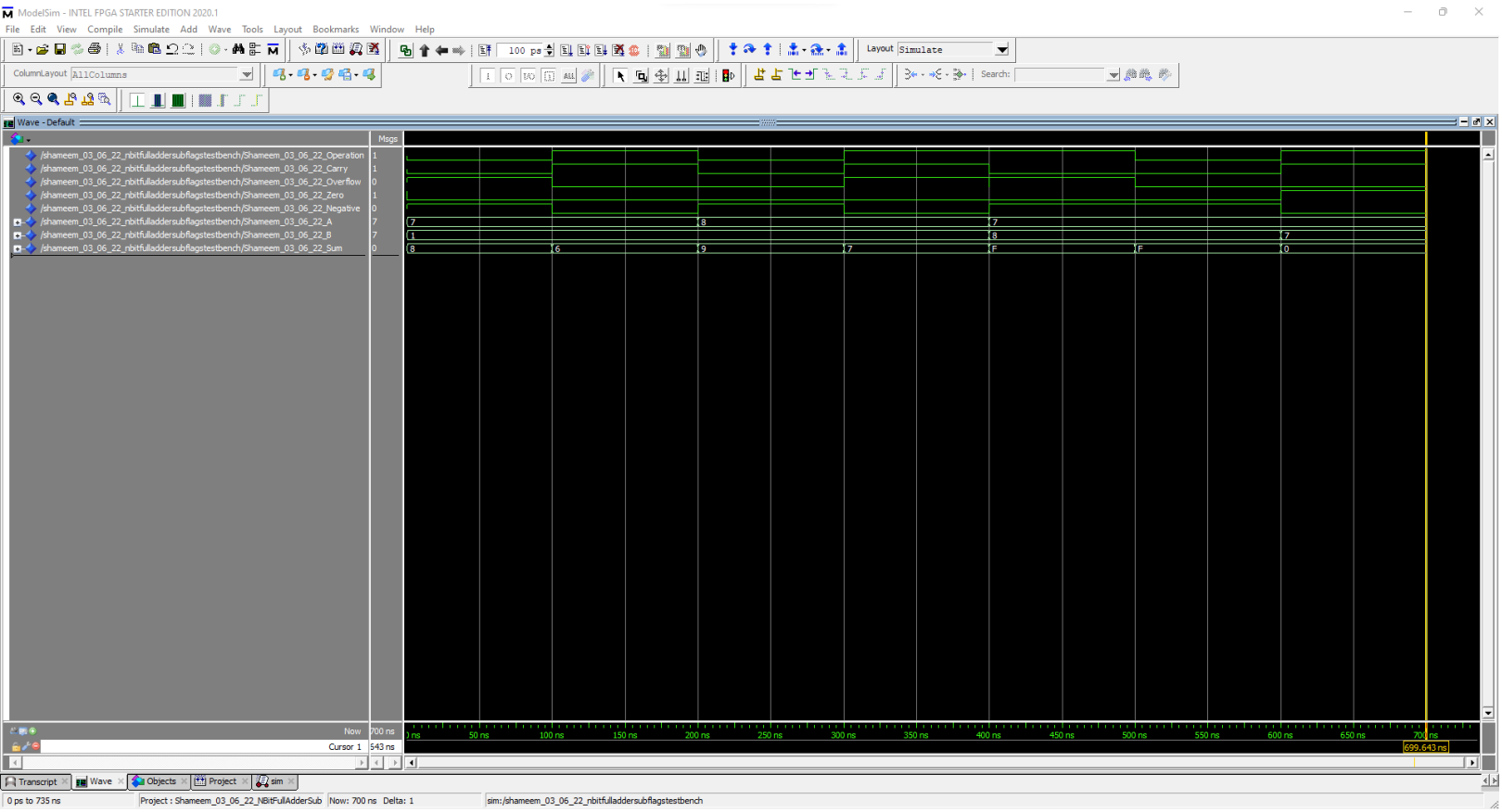


Figure 11: 4-bit Waveform for N-Bit Full Adder/Sub with flags. Waveforms are in hexadecimal.

The waveform will be explained by the cases provided for every 100 ns.

Case A) Most positive 4 bit integer plus one. (Numbers in hexadecimal)

The Sum is correct for case A because it shows 7+1=8 which is correct.

The Zero flag is correct because 8 is not zero, so zero flag is 0.

The Negative Flag is also correct because 8 is actually 1111 in 4 bits binary meaning, that for 4 bits signed integers it's a negative number.

The Overflow flag is correct because in 4 bits binary 1111 is negative which is shows two positives adding to equal negative, hence why the overflow flag is 1

Case B) Most positive 4 bit integer minus one. (Numbers in hexadecimal)

The Sum is correct for case B because it shows 7-1=6 which is correct.

The Zero flag is correct because 8 is not zero, so zero flag is 0.

The Negative Flag is 0 and correct because 6 has leftmost bit as 0.

The Overflow flag is 0 and correct because positive + positive = positive is right.

Case C) Most Negative 4 bit integer plus one. (Numbers in hexadecimal)

The Sum is correct for case C because it shows 8 + 1 = 9 which is correct.

The Zero flag is correct because 9 is not zero, so zero flag is 0.

The Negative Flag is 1 and correct because hexadecimal 9 in binary has leftmost bit as 1.

The Overflow flag is 0 and correct because it did not trigger an overflow error which only occurs when positive + positive = negative or negative + negative = positive.

Case D) Most Negative 4 bit integer minus one. (Numbers in hexadecimal)

The Sum is correct for case D because it shows 8 - 1 = 7 which is correct.

The Zero flag is correct because 8 is not zero, so zero flag is 0.

The Negative Flag is 0 and correct because hexadecimal 7 in binary has leftmost bit as 1.

The Overflow flag is 1 and is correct because for 4 bit binary we have a negative minus a positive equaling a positive

Case E) Most Positive – Most Negative. (4 bits) (Numbers in hexadecimal)

The Sum is correct for case E because it shows 7 – 8 = F which is not correct normally but right in this case since it's for 4 bits.

The Zero flag is correct because F is not zero, so zero flag is 0.

The Negative Flag is 1 and correct because hexadecimal F in binary has leftmost bit as 1.

The Overflow flag is 1 and is correct because for 4 bit binary we get a positive minus a negative equaling a negative.

Case F) Most Positive + Most Negative. (4 bits) (Numbers in hexadecimal)

The Sum is correct for case F because it shows 7 + 8 = F which is not correct normally but right in this case since it's for 4 bits.

The Zero flag is correct because F is not zero, so zero flag is 0.

The Negative Flag is 1 and correct because hexadecimal F in binary has leftmost bit as 1.

The Overflow flag is 0 and is correct because we get a positive adding a negative which may end up as a negative number so no error.

Case G) Most Positive - Most Positive. (4 bits) (Numbers in hexadecimal)

The Sum is correct for case G because it shows 7 - 7 = 0.

The Zero flag is correct because 0 is zero, so zero flag is 1.

The Negative Flag is 0 because 0 is not negative.

The Overflow flag is 0 because no overflow error conditions were satisfied.

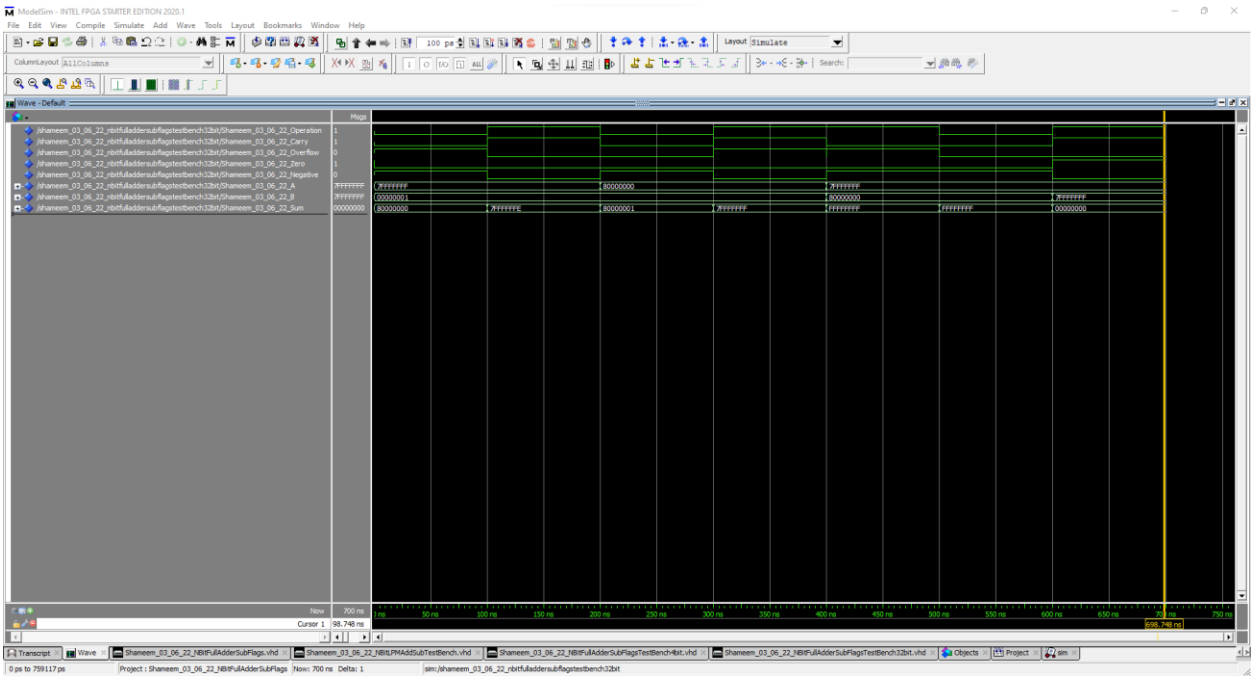


Figure 12: 32-bit Waveform for N-Bit Full Adder/Sub with flags. Waveforms are in hexadecimal.

The waveform will be explained by the cases provided for every 100 ns.

Case A) Most positive 32 bit integer plus one. (Numbers in hexadecimal)

The Sum is correct for case A because it shows correct output.

The Zero flag is correct because the result is not zero, so zero flag is 0.

The Negative Flag is also correct because the result's leftmost bit in 16 bit binary is 1.

The Overflow flag is 1 and is correct because in 32 bit binary we get positive + positive equals a negative.

Case B) Most positive 32 bit integer minus one. (Numbers in hexadecimal)

The Sum is correct for case B shows the correct result.

The Zero flag is correct because the result is not zero, so zero flag is 0.

The Negative Flag is 0 and correct because the result in 16 bit binary has leftmost bit as 0.

The Overflow flag is 0 and correct because positive – positive may be negative.

Case C) Most Negative 32 bit integer plus one. (Numbers in hexadecimal)

The Sum is correct for case C because the correct 16 bit result.

The Zero flag is correct because the result is not zero, so zero flag is 0.

The Negative Flag is 1 and correct because the 32 bit result

The Overflow flag is 0 and correct because it did not trigger an overflow error which only occurs when positive + positive = negative or negative + negative = positive.

Case D) Most Negative 32 bit integer minus one. (Numbers in hexadecimal)

The Sum is correct for case D because the result is correct.

The Zero flag is correct because the result is not zero, so zero flag is 0.

The Negative Flag is 0 because the result's 32 bit binary has leftmost bit as 0.

The Overflow flag is 1 and is correct because for 32 bit binary we have a negative minus a positive equaling a positive.

Case E) Most Positive – Most Negative. (32 bits) (Numbers in hexadecimal)

The Sum is correct for case E the result is correct.

The Zero flag is correct because the result is not zero, so zero flag is 0.

The Negative Flag is 1 and correct because leftmost 32 bit number is 1.

The Overflow flag is 1 and is correct because for 32 bit binary we get a positive minus a negative equals a negative.

Case F) Most Positive + Most Negative. (32 bits) (Numbers in hexadecimal)

The Sum is correct for case F the result is correct.

The Zero flag is correct because the result is not zero, so zero flag is 0.

The Negative Flag is 1 and correct because leftmost 32 bit number is 1.

The Overflow flag is 0 and is correct because a positive + negative may be a negative.

Case G) Most Positive - Most Positive. (32 bits) (Numbers in hexadecimal)

The Sum is correct for case F the result is correct.

The Zero flag is correct because the result is zero, so zero flag is 0.

The Negative Flag is 0 because 0 is not negative.

The Overflow flag is 0 and is correct because positive – positive may be 0.

Task 9: LPM Add/Sub circuit with simulation and comparison

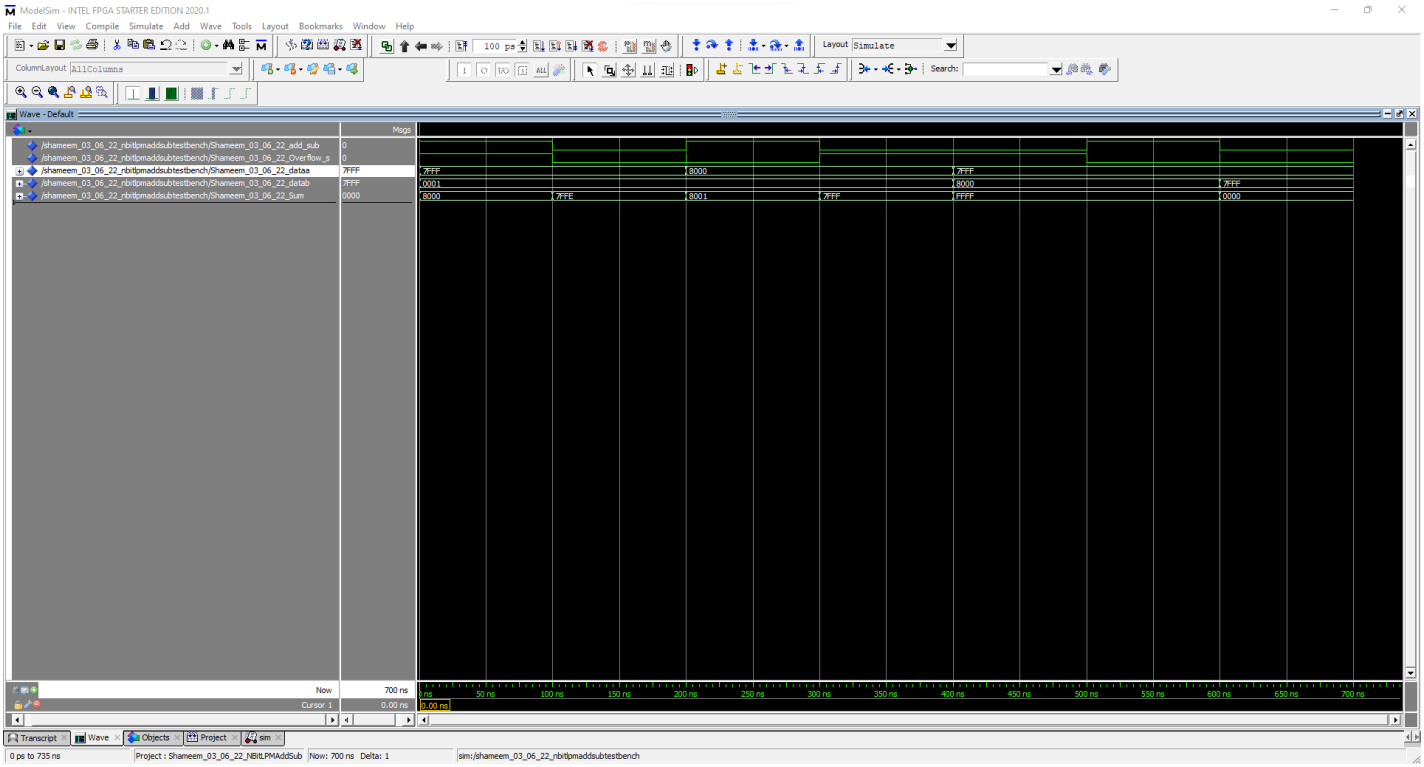


Figure 13: N-Bit LPM Waveform Simulation (16 bit simulation)

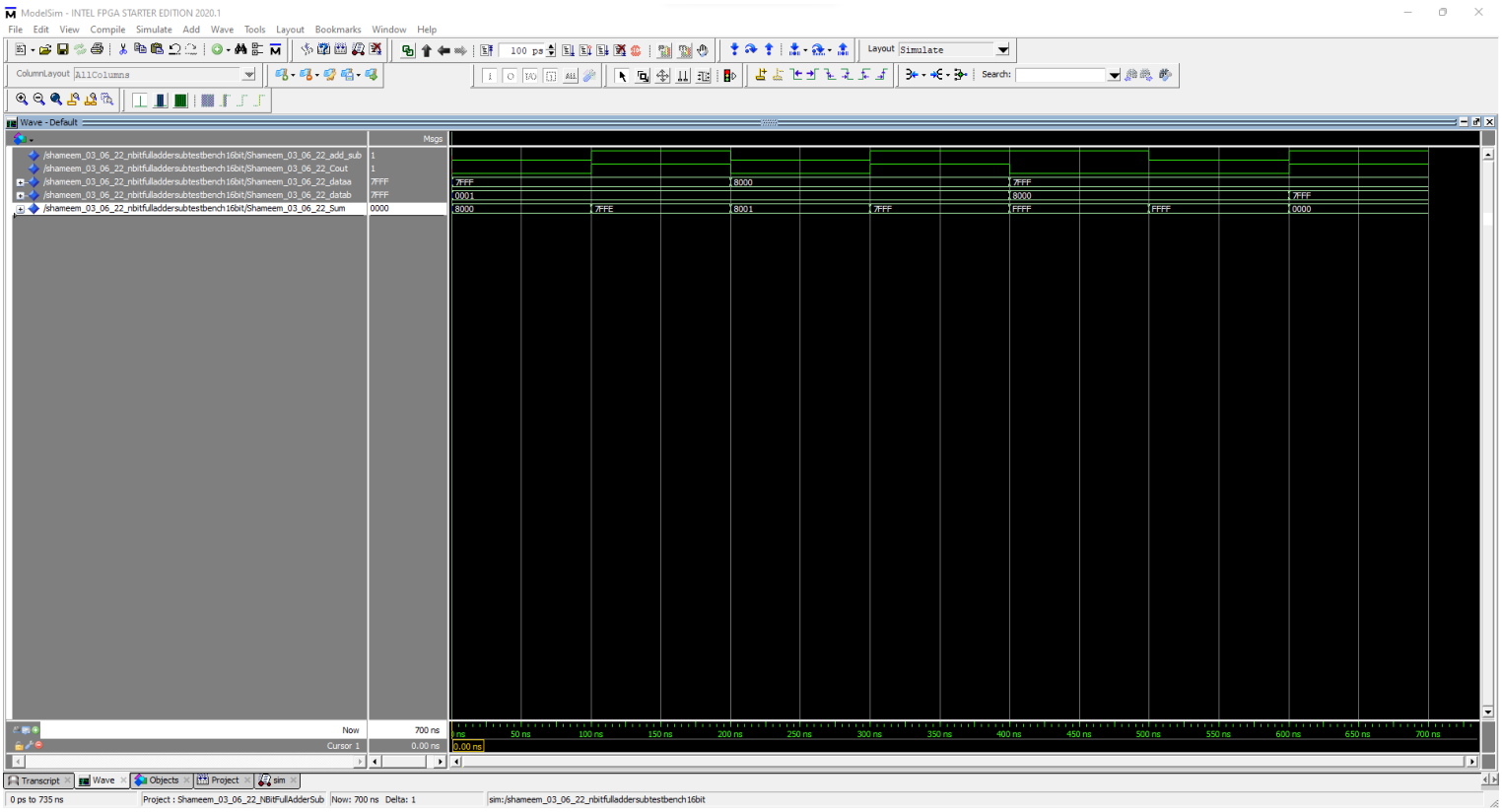
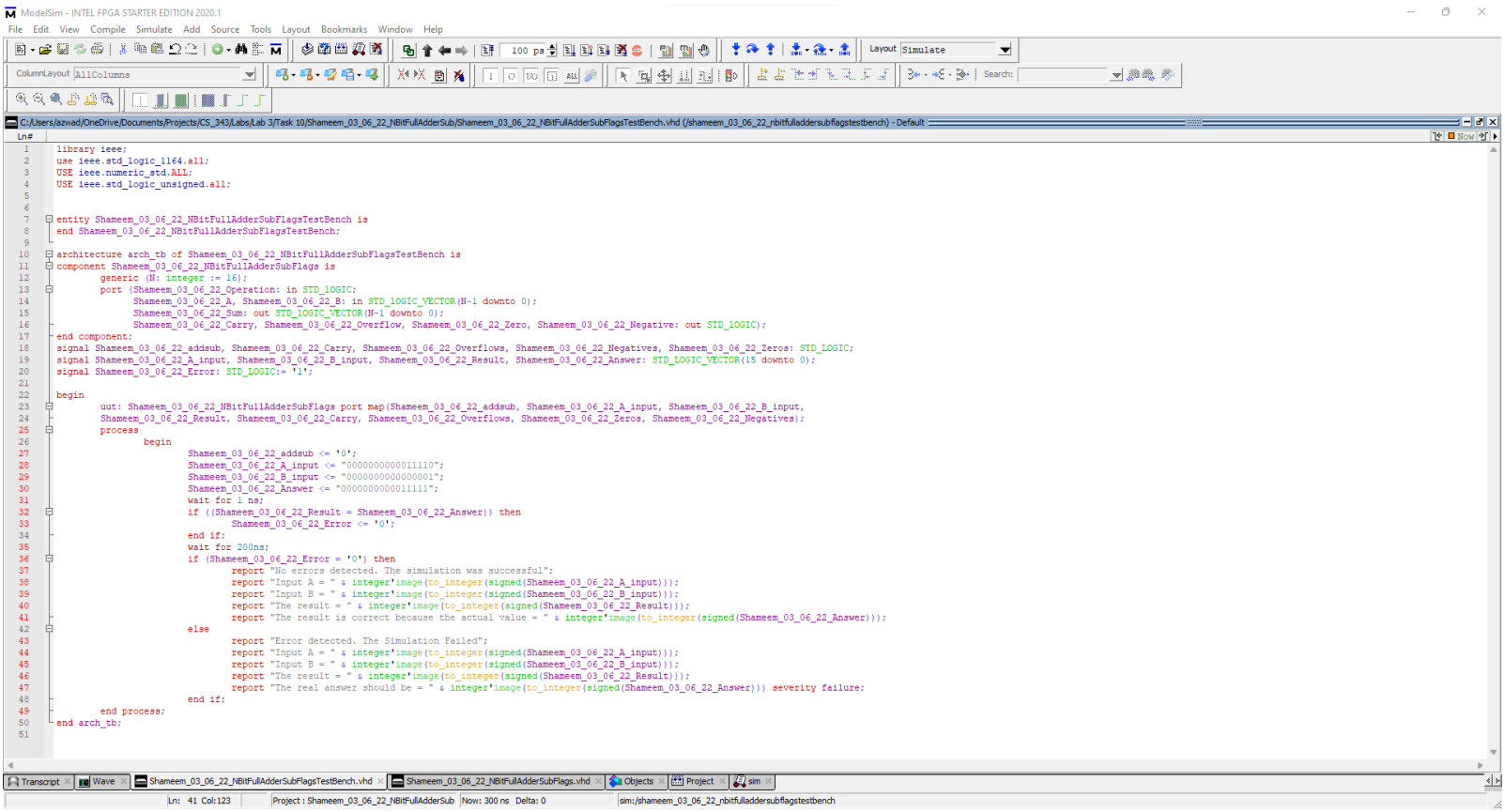


Figure 14: N-Bit AdderSub Waveform Simulation (16 bit simulation)

As you can see in figure 13 and figure 14 the N-bit Addsub unit simulated for 16 bits and the N-bit Addsub using LPM has the same inputs and same result value obtained from the circuits. The only difference is that for LPM operations value 0 is subtraction and operations value 1 is addition. Therefore, while the circuits other outputs may differ slightly the actual output result is the same.

Task 10: Create a Test-Bench file in VHDL to test Add_SUB unit for n=16 bits. Please demonstrate that the test-bench detects an error (intentionally created) in your design and prints out simulation time, expected operand 1 and operand result value, actual result value, and values of operand 1 and operand 2 that caused the error.



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


entity Shameem_03_06_22_NBitFullAdderSubFlagsTestBench is
end Shameem_03_06_22_NBitFullAdderSubFlagsTestBench;

architecture arch_tb of Shameem_03_06_22_NBitFullAdderSubFlagsTestBench is
component Shameem_03_06_22_NBitFullAdderSubFlags is
        generic (N: integer := 16);
        port (Shameem_03_06_22_Operation: in STD_LOGIC;
              Shameem_03_06_22_A, Shameem_03_06_22_B: in STD_LOGIC_VECTOR(N-1 downto 0);
              Shameem_03_06_22_Sum: out STD_LOGIC_VECTOR(N-1 downto 0);
              Shameem_03_06_22_Carry, Shameem_03_06_22_Overflow, Shameem_03_06_22_Zero, Shameem_03_06_22_Negative: out STD_LOGIC);
end component;
signal Shameem_03_06_22_addsub, Shameem_03_06_22_Carry, Shameem_03_06_22_Overflows, Shameem_03_06_22_Negatives, Shameem_03_06_22_Zeros: STD_LOGIC;
signal Shameem_03_06_22_A_input, Shameem_03_06_22_B_input, Shameem_03_06_22_Result, Shameem_03_06_22_Answer: STD_LOGIC_VECTOR(15 downto 0);
signal Shameem_03_06_22_Error: STD_LOGIC:= '1';

begin
        uut: Shameem_03_06_22_NBitFullAdderSubFlags port map(Shameem_03_06_22_addsub, Shameem_03_06_22_A_input, Shameem_03_06_22_B_input,
        Shameem_03_06_22_Result, Shameem_03_06_22_Carry, Shameem_03_06_22_Overflows, Shameem_03_06_22_Zeros, Shameem_03_06_22_Negatives);
        process
                begin
                        Shameem_03_06_22_addsub <= '0';
                        Shameem_03_06_22_A_input <= "0000000000011110";
                        Shameem_03_06_22_B_input <= "0000000000000001";
                        Shameem_03_06_22_Answer <= "0000000000011111";
                        wait for 1 ns;
                        if ((Shameem_03_06_22_Result = Shameem_03_06_22_Answer)) then
                                Shameem_03_06_22_Error <= '0';
                        end if;
                        wait for 200ns;
                        if (Shameem_03_06_22_Error = '0') then
                                report "No errors detected. The simulation was successful";
                                report "Input A = " & integer'image(to_integer(signed(Shameem_03_06_22_A_input)));
                                report "Input B = " & integer'image(to_integer(signed(Shameem_03_06_22_B_input)));
                                report "The result = " & integer'image(to_integer(signed(Shameem_03_06_22_Result)));
                                report "The result is correct because the actual value = " & integer'image(to_integer(signed(Shameem_03_06_22_Answer)));
                        else
                                report "Error detected. The Simulation Failed";
                                report "Input A = " & integer'image(to_integer(signed(Shameem_03_06_22_A_input)));
                                report "Input B = " & integer'image(to_integer(signed(Shameem_03_06_22_B_input)));
                                report "The result = " & integer'image(to_integer(signed(Shameem_03_06_22_Result)));
                                report "The real answer should be = " & integer'image(to_integer(signed(Shameem_03_06_22_Answer))) severity failure;
                        end if;
        end process;
end arch_tb;
```

Figure 15: Test Bench utilized for testing Add_SUB unit for n=16 bits and printouts for simulation time operand 1,2 and result value plus real value

The test bench is quite simple it adds two 16 bit binaries and checks if the result is not the actual result. If the result is not the true answer it will declare an error and trigger the severity failure. However, if the result is the actual result, then it will succeed. In both situations of success or failure the test bench will print out simulation time, operand A and operand B plus the circuit's computed result plus the actual answer.

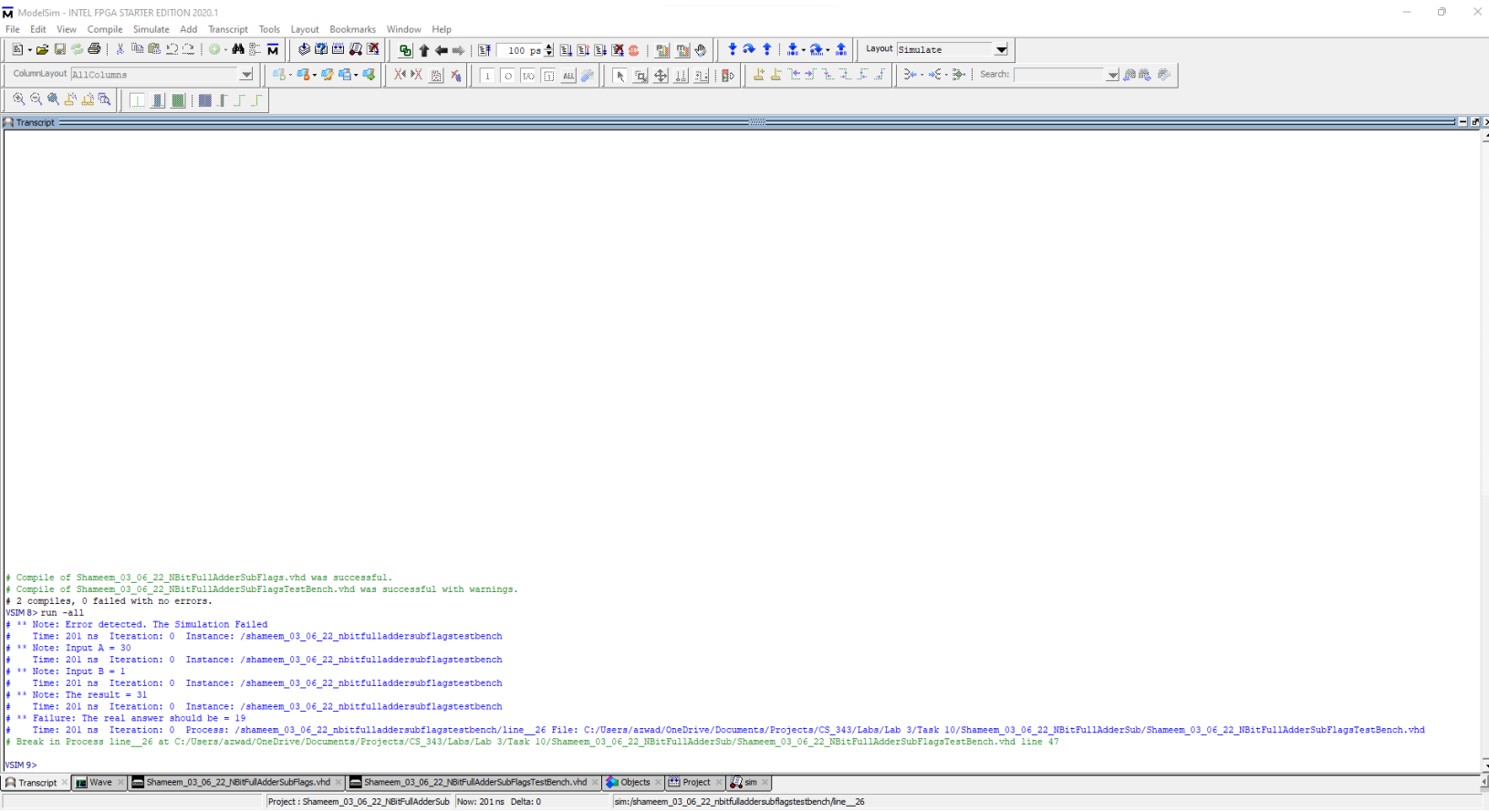


Figure 16: Printout the intentional error made to show that the Test-Bench throws an error if an error occurs

Clearly, an error was detected in the testbench. We purposely constructed this error by making the actual answer wrong so that when it checks the circuits answer with the actual answer it will think it's wrong and therefore trigger an error. In the printouts of the testbench the simulation time is posted and operands A and B plus the results of the circuit. Also, we can see the incorrect actual value that the testbench was given.

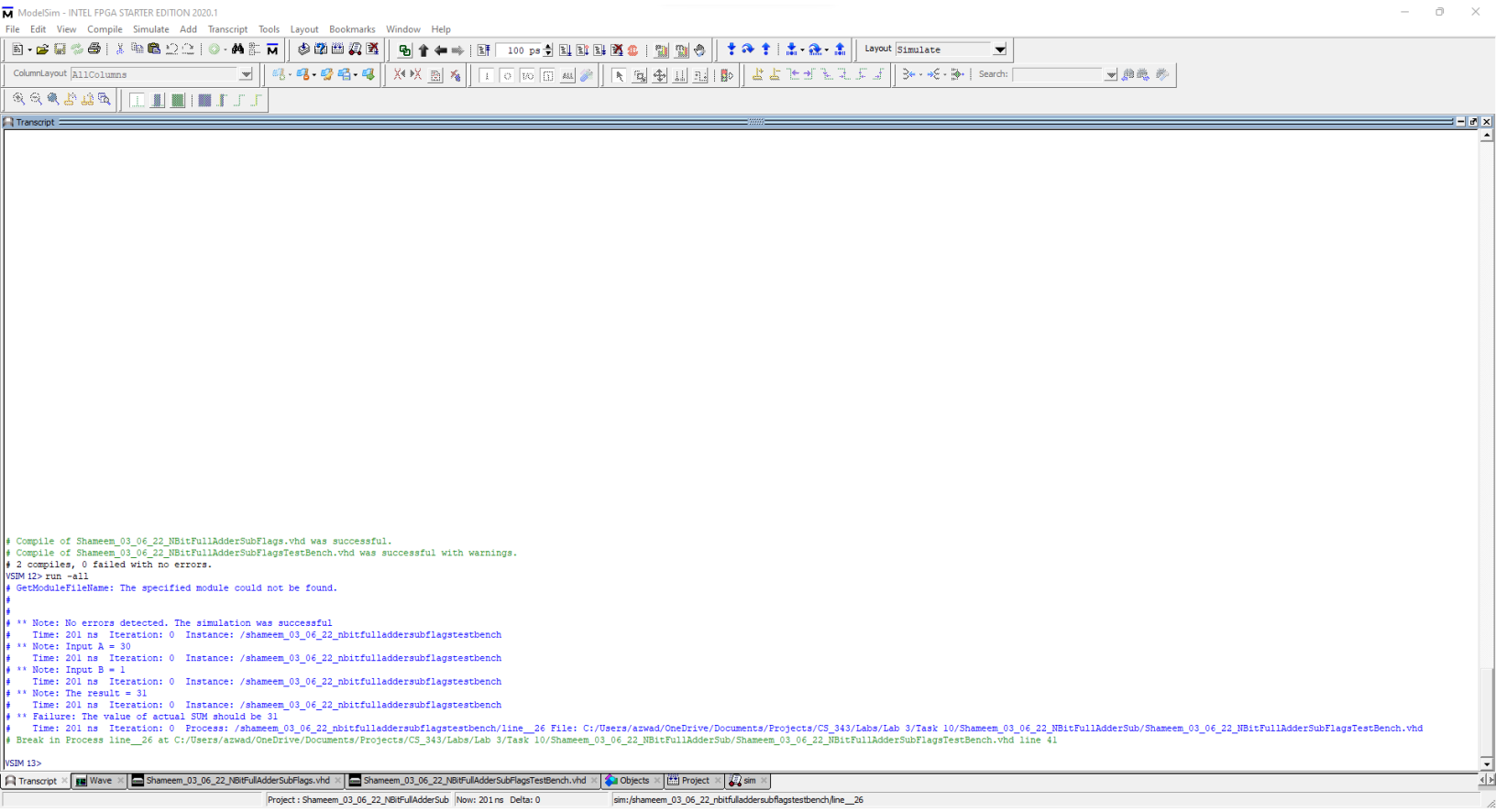


Figure 17: The Test-Bench printouts no error when correct and printouts input values result and the actual values

Clearly, no error was detected in the testbench. This time around we did not input this error so that when it checks the circuits answer with the actual answer it will think it's correct as long as the circuit obtains the correct result. In the printouts of the testbench the simulation time is posted and operands A and B plus the results of the circuit. Also, we can see the actual value that the testbench was given.

Conclusion:

The Adder lab was a great way to experiment with VHDL behavioral and structural code. In addition, the Adder lab gave us an insight about using libraries such as LPM and package creation. Furthermore, we were able to create circuits that would work for any number of N bits instead of creating circuits that are strictly defined for a number of bits. Lastly, this exercise also gave us good practice with ModelSim to simulate the circuits we made and make testbench to printout or show numbers in hexadecimal or any base.