CS 342000 | CS343000
Instructor: Professor Izidor Gertner
Spring 2022
Azwad Shameem, 4/10/2022
Laboratory Project: Design Arithmetic Logic Unit

Please hand write and sign statements affirming that you will not cheat here and in your submission:

"I will neither give nor receive unauthorized assistance on this LAB. I will use only one computing device to perform this LAB".

I will neither give nor recieve unathorized assistance on this exam. I will only use one computing device to perform this test

Azwad Shameem

Table of Contents

Objective

The objective of the Arithmetic Logic Unit lab was to implement MIPS instructions in VHDL. MIPS instructions such as ADD, ADDU, SUB, SUBU, AND, NOR, OR, SLL, SRA, ADDI, ADDIU, ANDI, ORI, LW, SW was all intended to be implements by VHDL programming. The VHDL program was then to be tested by utilizing ModelSim simulations to create waveforms for select cases that would show the results of the MIPS instructions that were implemented in VHDL. Furthermore, in order to ascertain correctness of the VHDL implementation of MIPS instructions we had to compare the MIPS instructions that ran on the MARS environment with the ModelSim simulation waveforms to confirm the correctness of the VHDL implementation of the MIPS instructions.

Description of Functionality and Simulation

Functionality:

RS Register

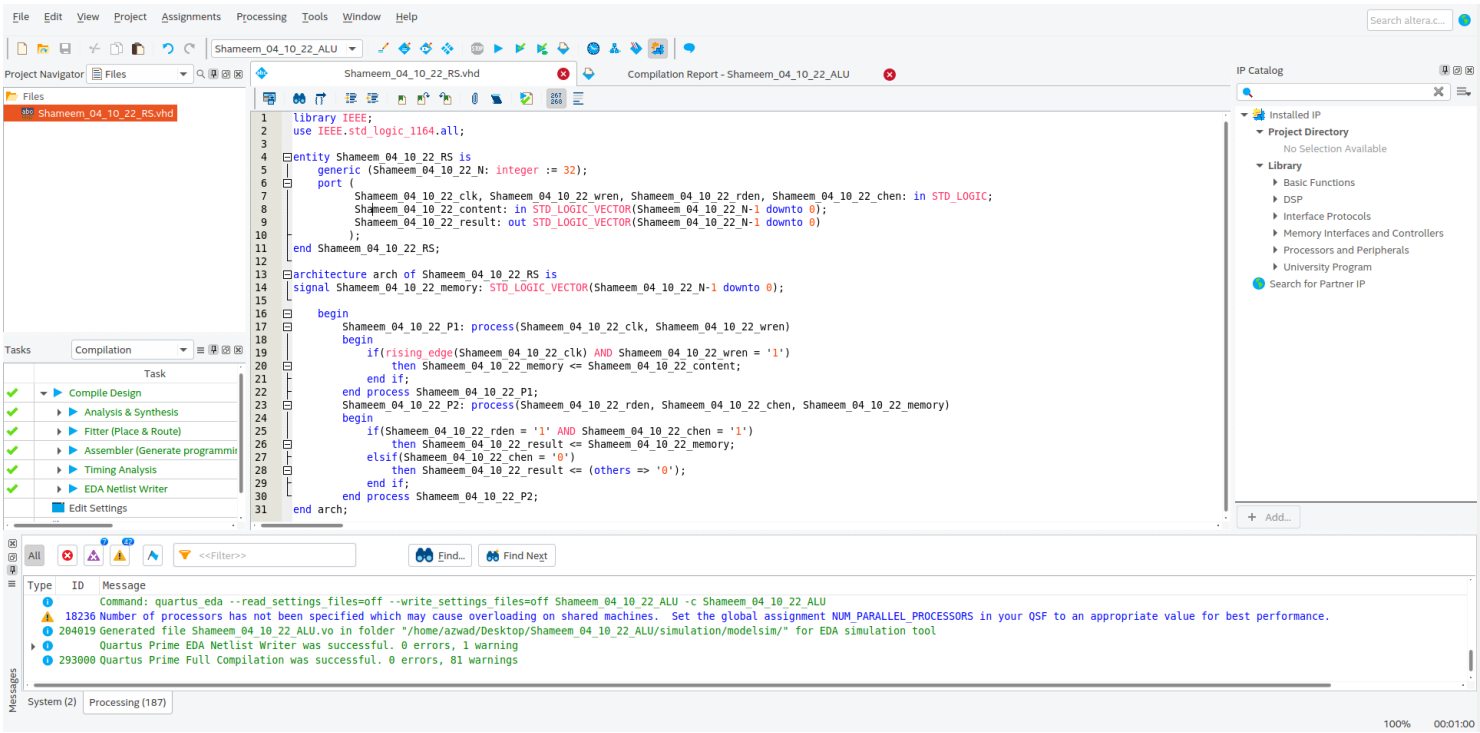


Figure 1: RS Register VHDL code compiled successfully in Quartus.

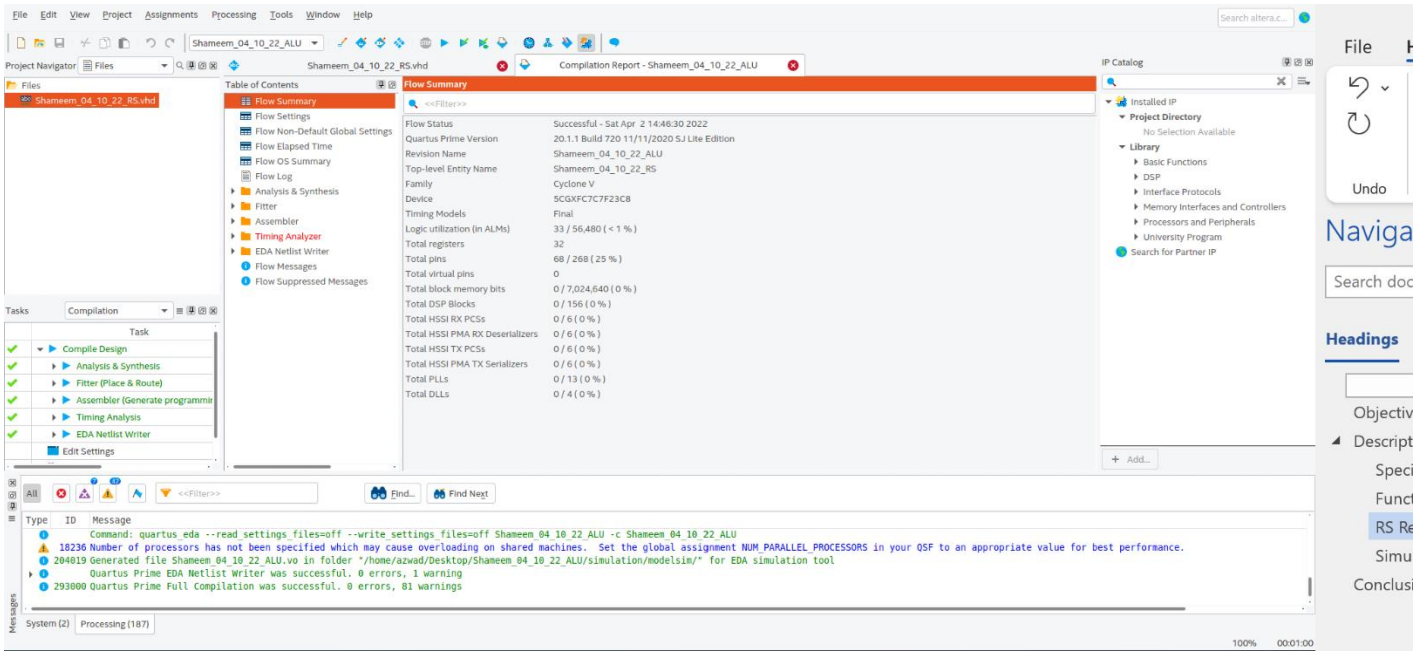


Figure 2: RS Register compilation report in Quartus.

RT Register

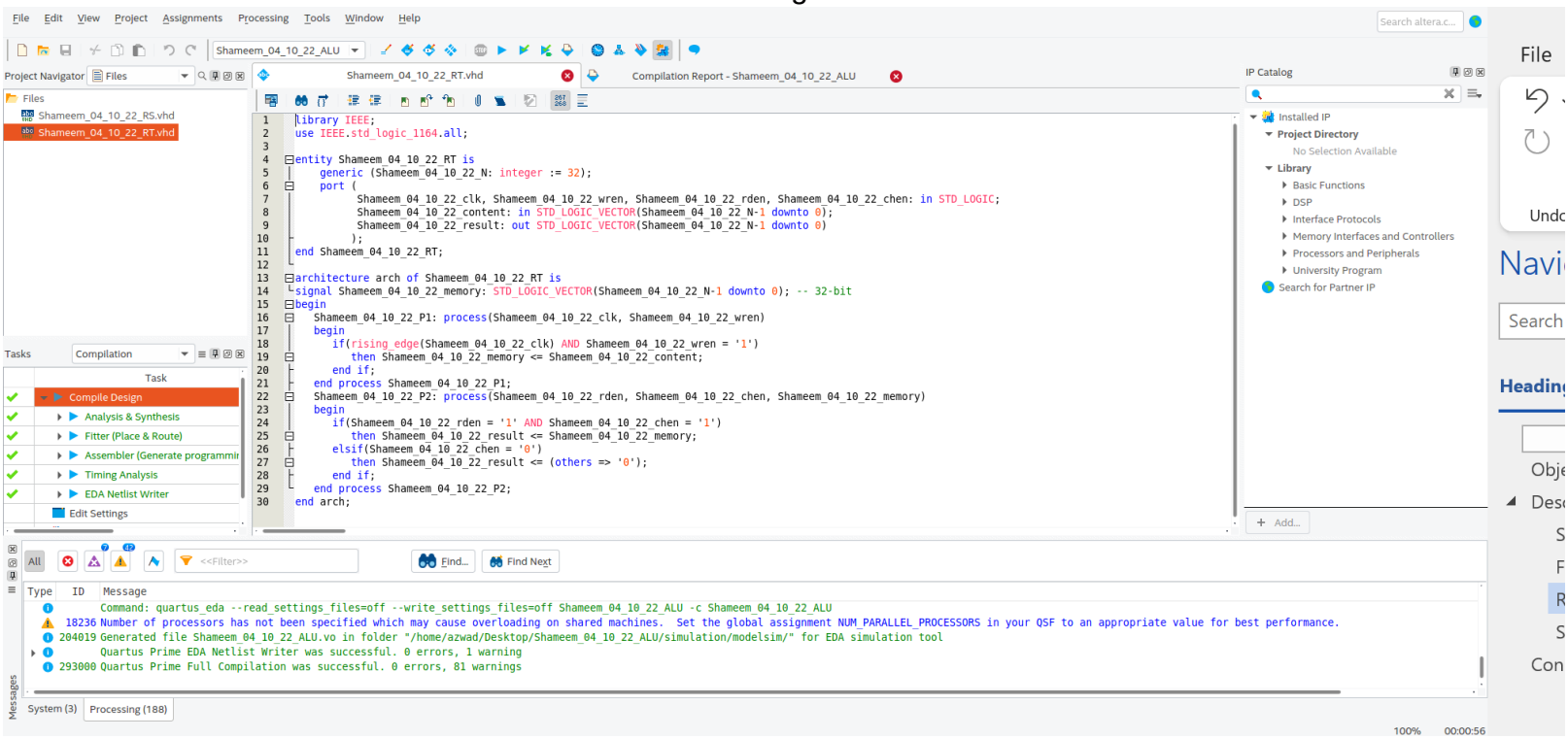


Figure 3: RT Register VHDL code compiled successfully in Quartus.

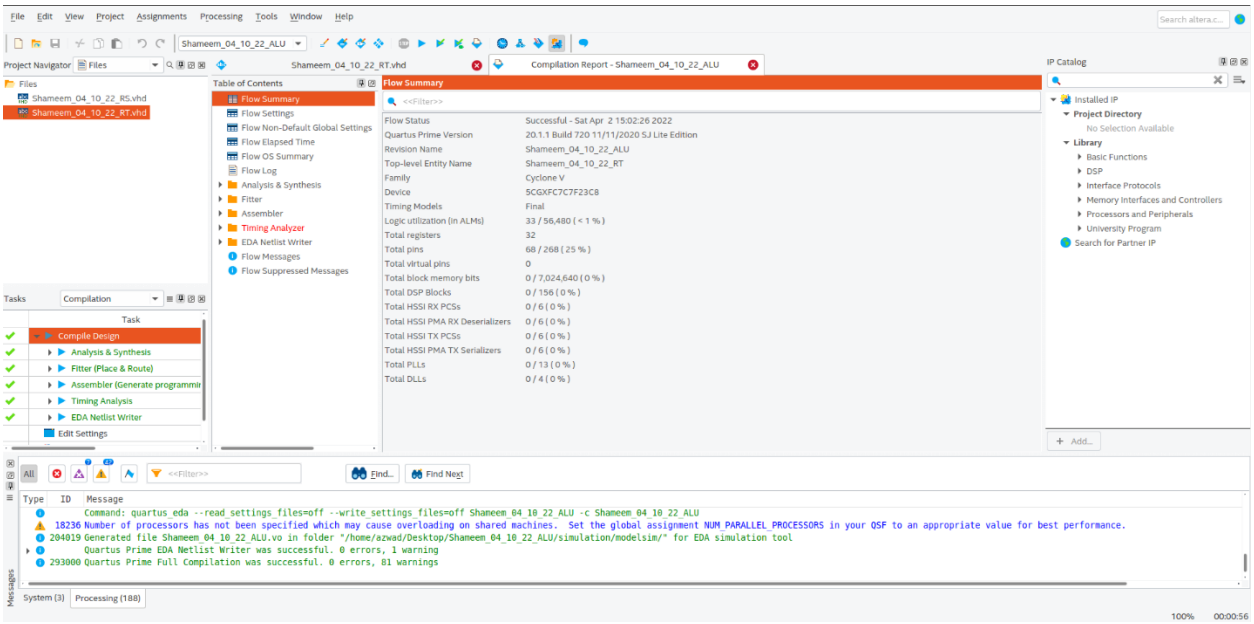


Figure 4: RT Register compilation report in Quartus.

.

RD Register

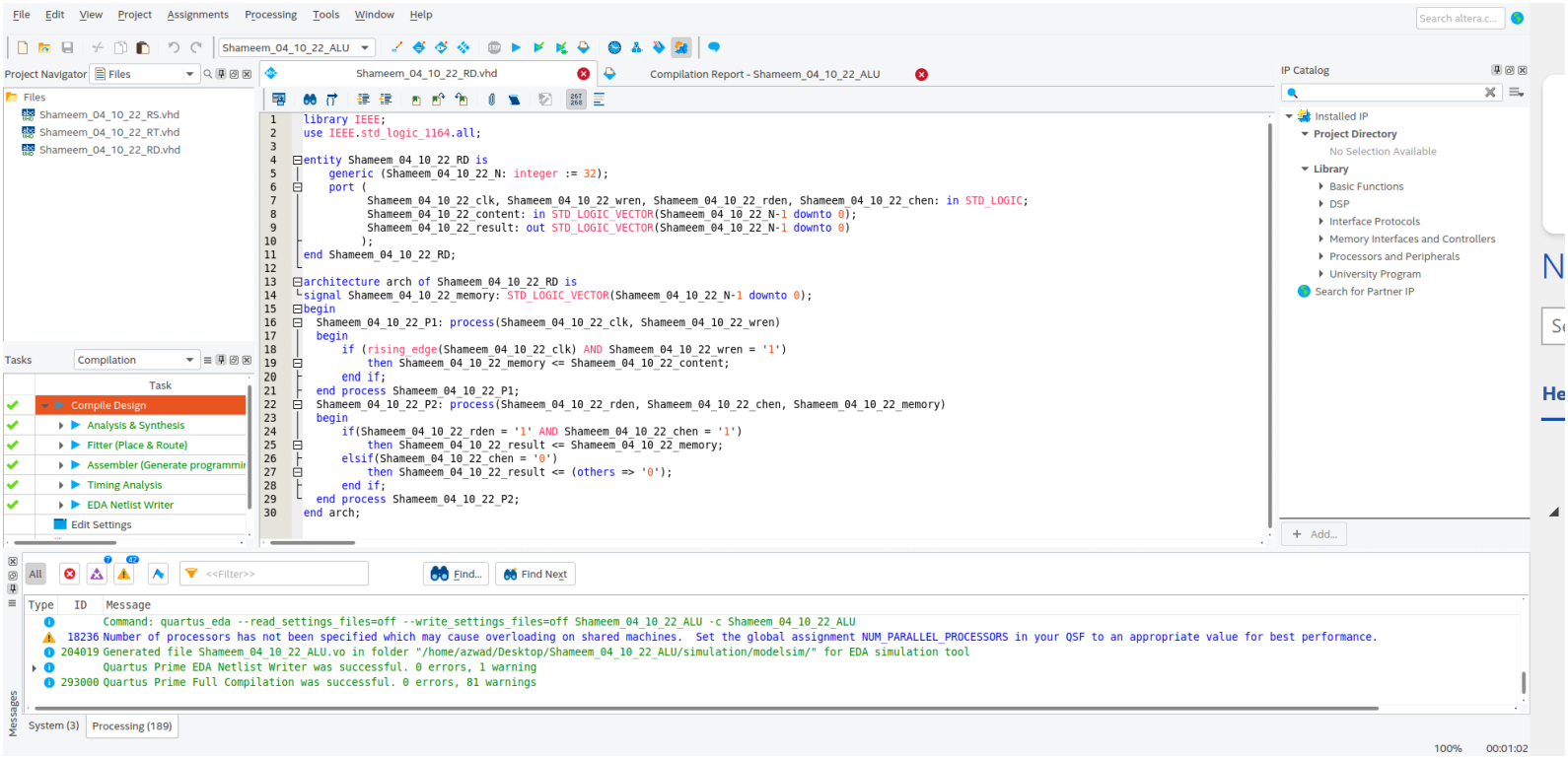


Figure 5: RD Register VHDL code compiled successfully in Quartus.

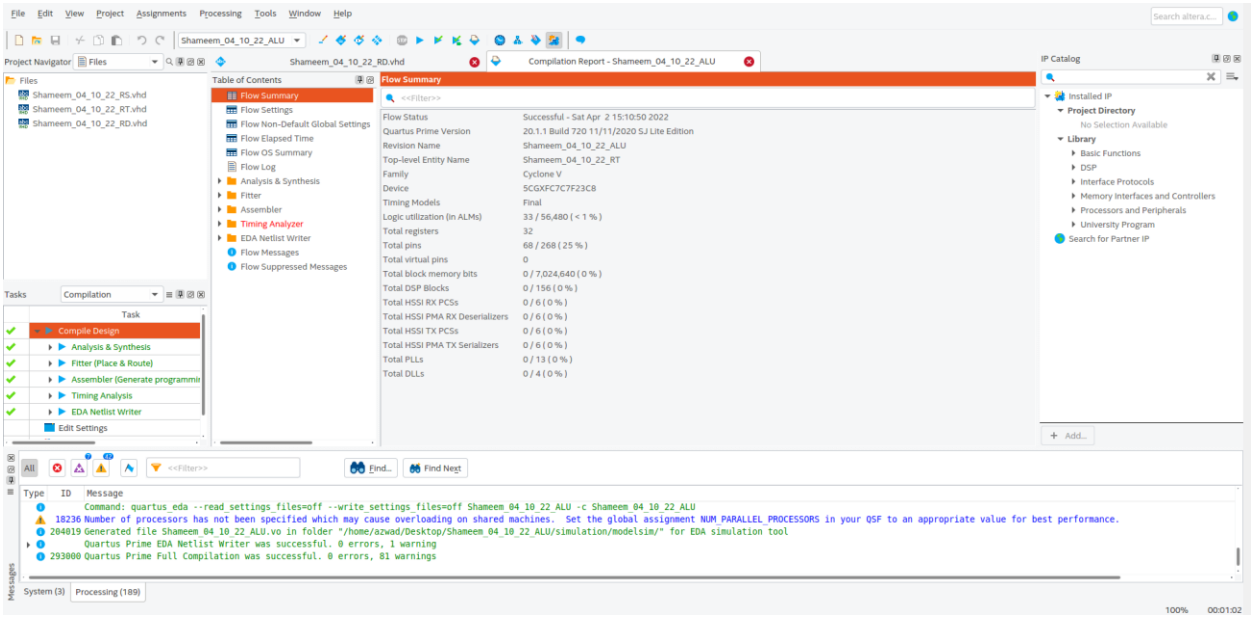


Figure 6: RT Register compilation report in Quartus.

N-Bit AddSub with Flags Unit

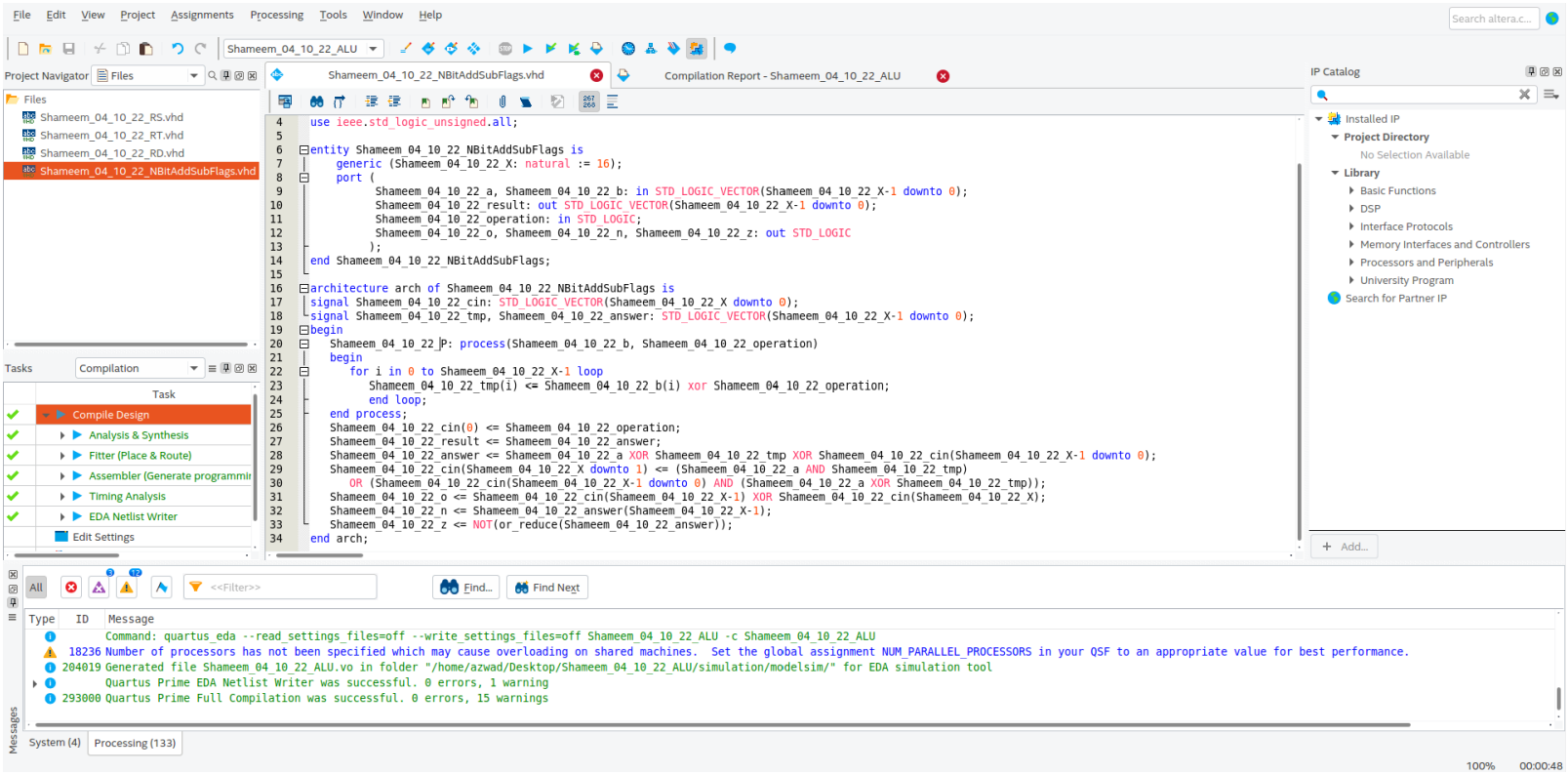


Figure 7: N-Bit AddSub with Flags VHDL code compiled successfully in Quartus.

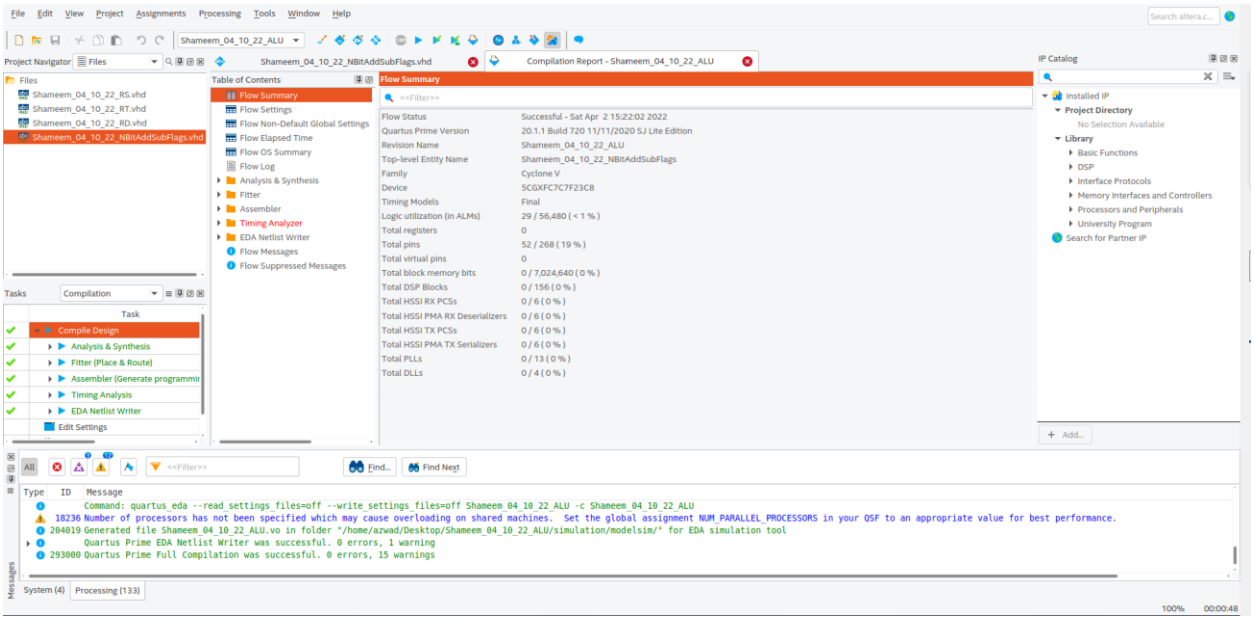
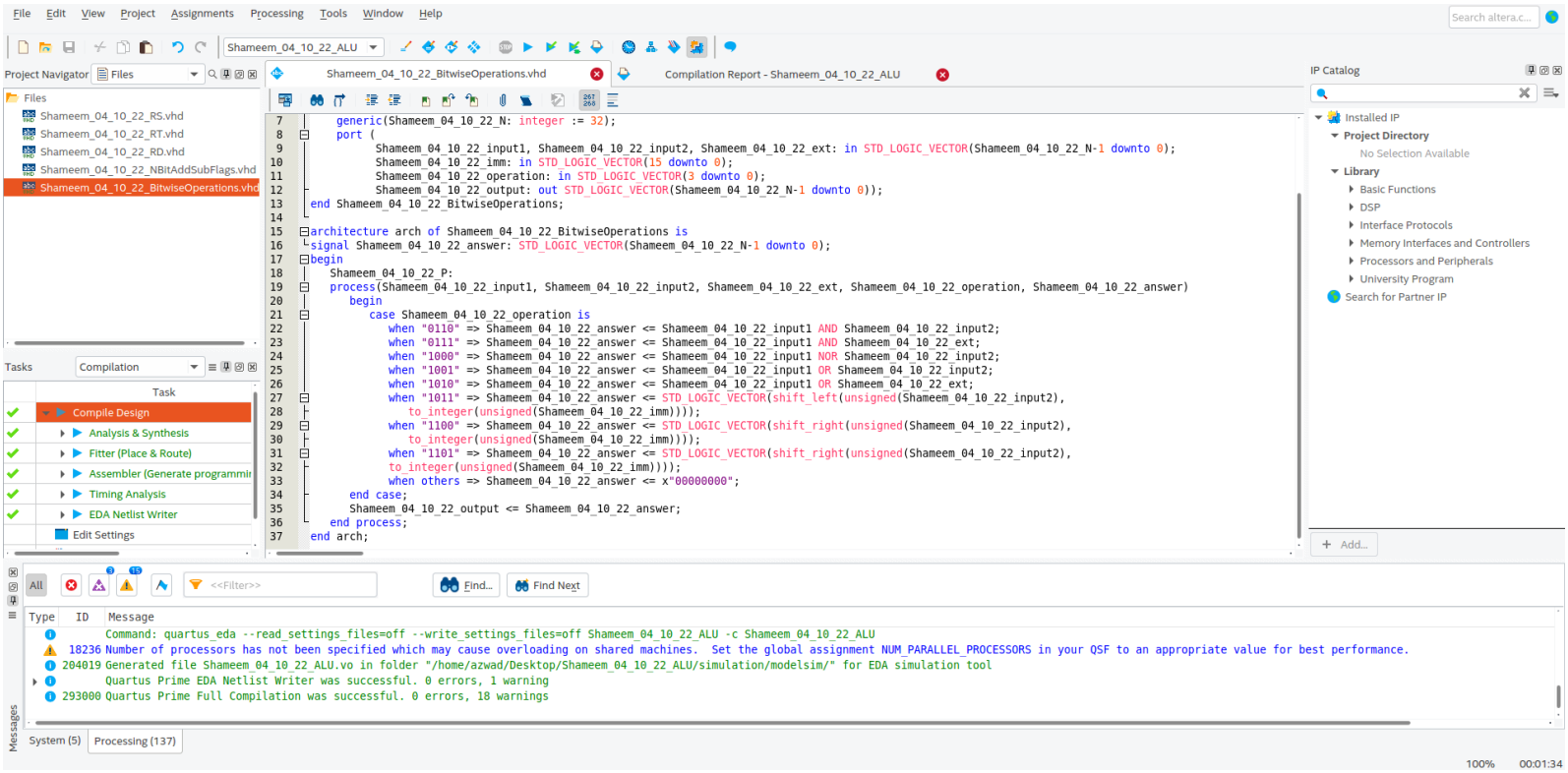


Figure 8: N-Bit AddSub with Flags compilation report in Quartus.

Bitwise Operations Unit




```
7    generic(Shameem_04_10_22_N: integer := 32);
8    port (
9          Shameem_04_10_22_input1, Shameem_04_10_22_input2, Shameem_04_10_22_ext: in STD_LOGIC_VECTOR(Shameem_04_10_22_N-1 downto 0);
10         Shameem_04_10_22_imm: in STD_LOGIC_VECTOR(15 downto 0);
11         Shameem_04_10_22_operation: in STD_LOGIC_VECTOR(3 downto 0);
12         Shameem_04_10_22_output: out STD_LOGIC_VECTOR(Shameem_04_10_22_N-1 downto 0));
13 end Shameem_04_10_22_BitwiseOperations;
14
15 architecture arch of Shameem_04_10_22_BitwiseOperations is
16 signal Shameem_04_10_22_answer: STD_LOGIC_VECTOR(Shameem_04_10_22_N-1 downto 0);
17 begin
18    Shameem_04_10_22_P:
19    process(Shameem_04_10_22_input1, Shameem_04_10_22_input2, Shameem_04_10_22_ext, Shameem_04_10_22_operation, Shameem_04_10_22_answer)
20       begin
21          case Shameem_04_10_22_operation is
22             when "0110" => Shameem_04_10_22_answer <= Shameem_04_10_22_input1 AND Shameem_04_10_22_input2;
23             when "0111" => Shameem_04_10_22_answer <= Shameem_04_10_22_input1 AND Shameem_04_10_22_ext;
24             when "1000" => Shameem_04_10_22_answer <= Shameem_04_10_22_input1 NOR Shameem_04_10_22_input2;
25             when "1001" => Shameem_04_10_22_answer <= Shameem_04_10_22_input1 OR Shameem_04_10_22_input2;
26             when "1010" => Shameem_04_10_22_answer <= Shameem_04_10_22_input1 OR Shameem_04_10_22_ext;
27             when "1011" => Shameem_04_10_22_answer <= STD_LOGIC_VECTOR(shift_left(unsigned(Shameem_04_10_22_input2),
28                to_integer(unsigned(Shameem_04_10_22_imm))));
29             when "1100" => Shameem_04_10_22_answer <= STD_LOGIC_VECTOR(shift_right(unsigned(Shameem_04_10_22_input2),
30                to_integer(unsigned(Shameem_04_10_22_imm))));
31             when "1101" => Shameem_04_10_22_answer <= STD_LOGIC_VECTOR(shift_right(unsigned(Shameem_04_10_22_input2),
32             to_integer(unsigned(Shameem_04_10_22_imm))));
33             when others => Shameem_04_10_22_answer <= x"00000000";
34       end case;
35       Shameem_04_10_22_output <= Shameem_04_10_22_answer;
36    end process;
37 end arch;
```

Figure 9: Bitwise Operations VHDL code compiled successfully in Quartus.

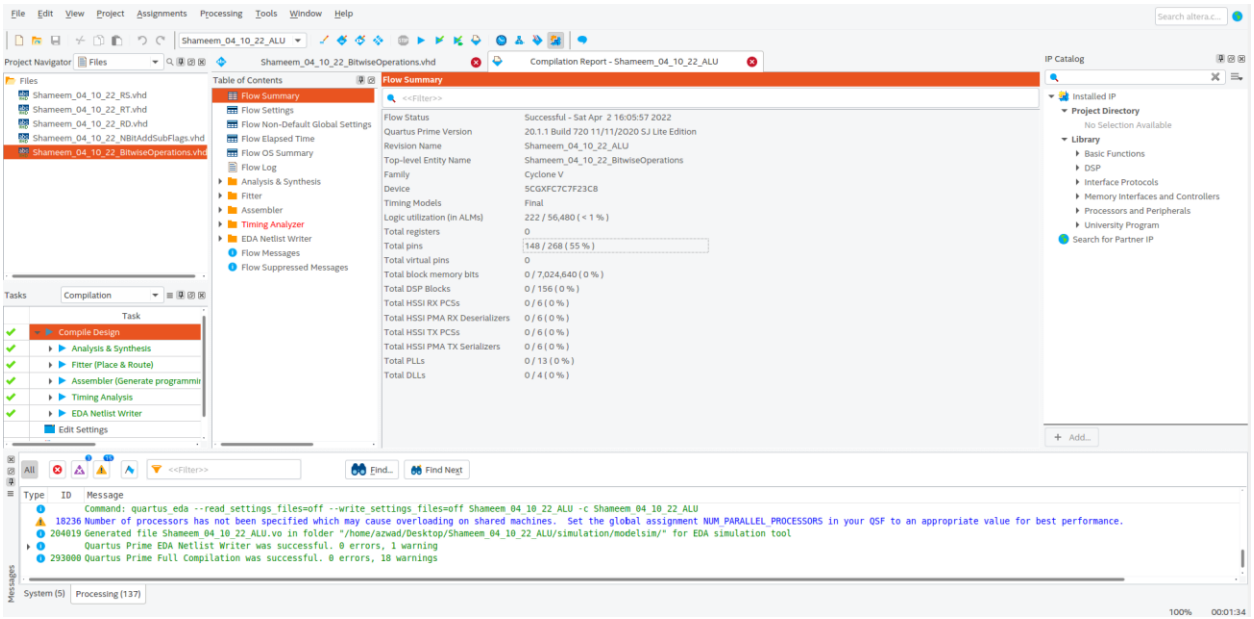
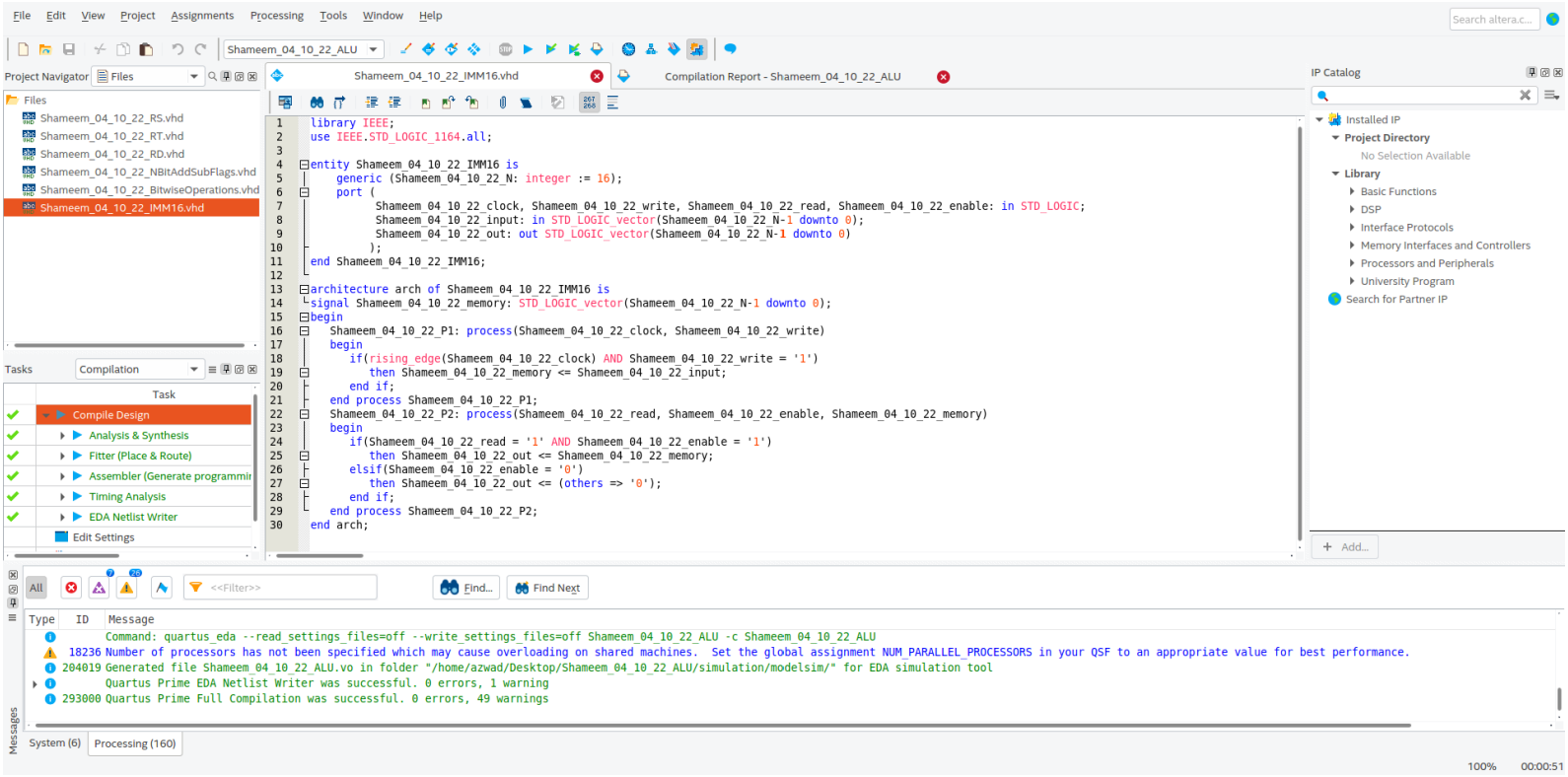


Figure 10: Bitwise Operations compilation report in Quartus.




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

entity Shameem_04_10_22_IMM16 is
    generic (Shameem_04_10_22_N: integer := 16);
    port (
          Shameem_04_10_22_clock, Shameem_04_10_22_write, Shameem_04_10_22_read, Shameem_04_10_22_enable: in STD_LOGIC;
          Shameem_04_10_22_input: in STD_LOGIC_vector(Shameem_04_10_22_N-1 downto 0);
          Shameem_04_10_22_out: out STD_LOGIC_vector(Shameem_04_10_22_N-1 downto 0)
          );
end Shameem_04_10_22_IMM16;

architecture arch of Shameem_04_10_22_IMM16 is
signal Shameem_04_10_22_memory: STD_LOGIC_vector(Shameem_04_10_22_N-1 downto 0);
begin
   Shameem_04_10_22_P1: process(Shameem_04_10_22_clock, Shameem_04_10_22_write)
   begin
      if(rising_edge(Shameem_04_10_22_clock) AND Shameem_04_10_22_write = '1')
         then Shameem_04_10_22_memory <= Shameem_04_10_22_input;
      end if;
   end process Shameem_04_10_22_P1;
   Shameem_04_10_22_P2: process(Shameem_04_10_22_read, Shameem_04_10_22_enable, Shameem_04_10_22_memory)
   begin
      if(Shameem_04_10_22_read = '1' AND Shameem_04_10_22_enable = '1')
         then Shameem_04_10_22_out <= Shameem_04_10_22_memory;
      elsif(Shameem_04_10_22_enable = '0')
         then Shameem_04_10_22_out <= (others => '0');
      end if;
   end process Shameem_04_10_22_P2;
end arch;
```

IMM16 Register

Figure 11: IMM16 Register VHDL code compiled successfully in Quartus.

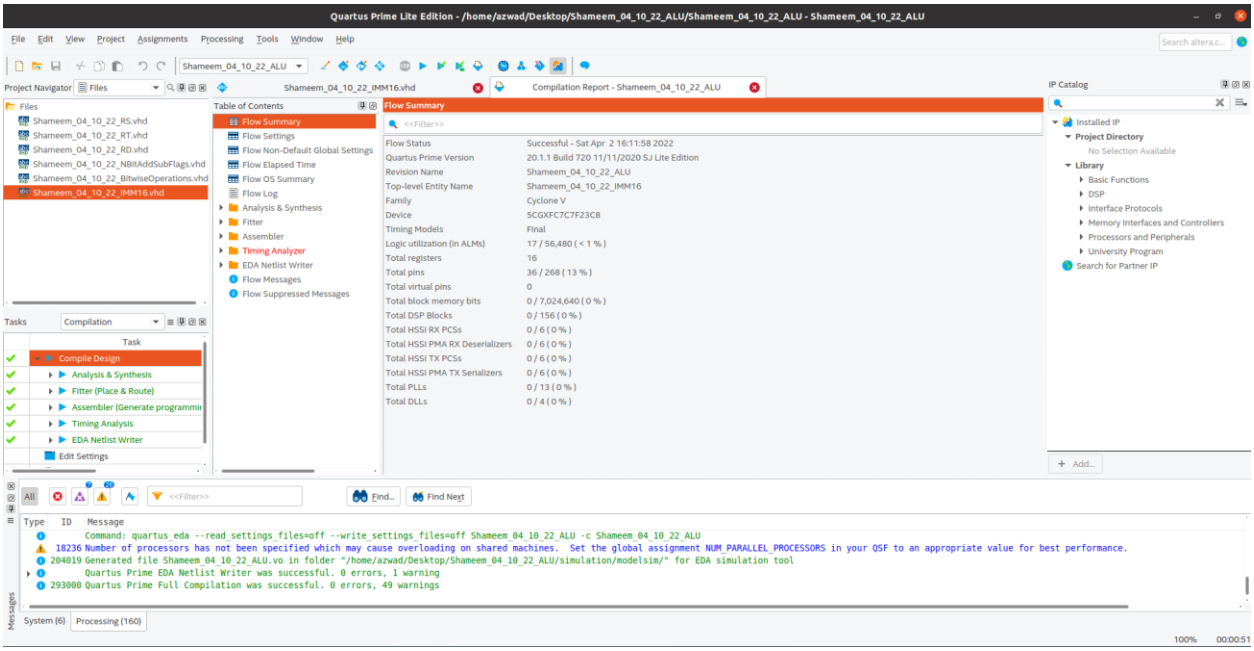


Figure 12: IMM16 Register compilation report in Quartus.

Sign Extender Unit

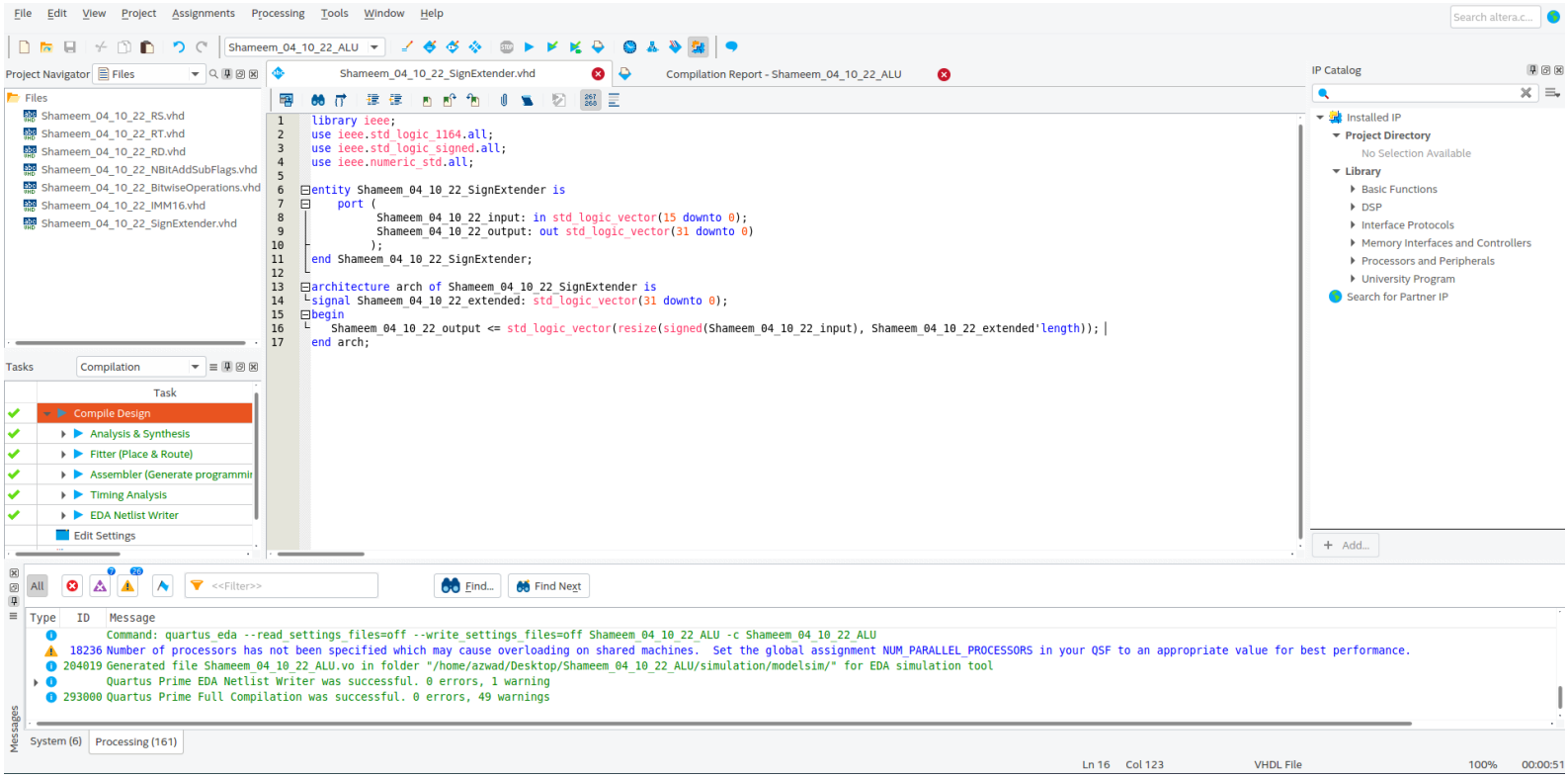


Figure 13: Sign Extender VHDL code compiled successfully in Quartus.

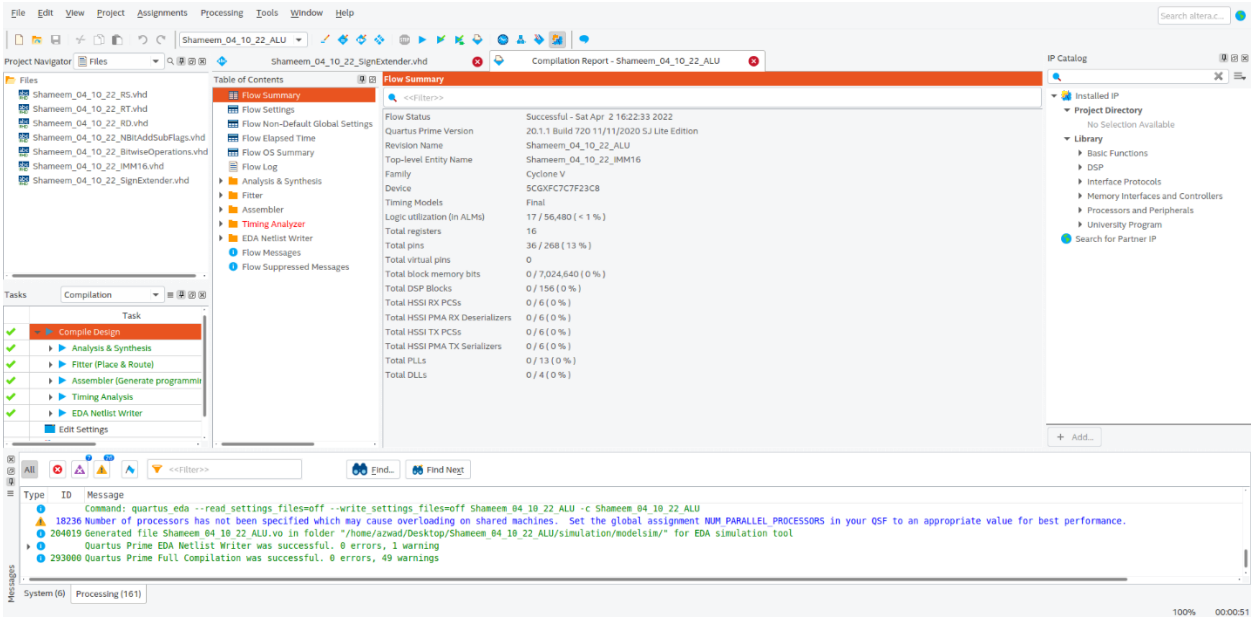


Figure 14: Sign Extender compilation report in Quartus.

MAR – Memory Address Register

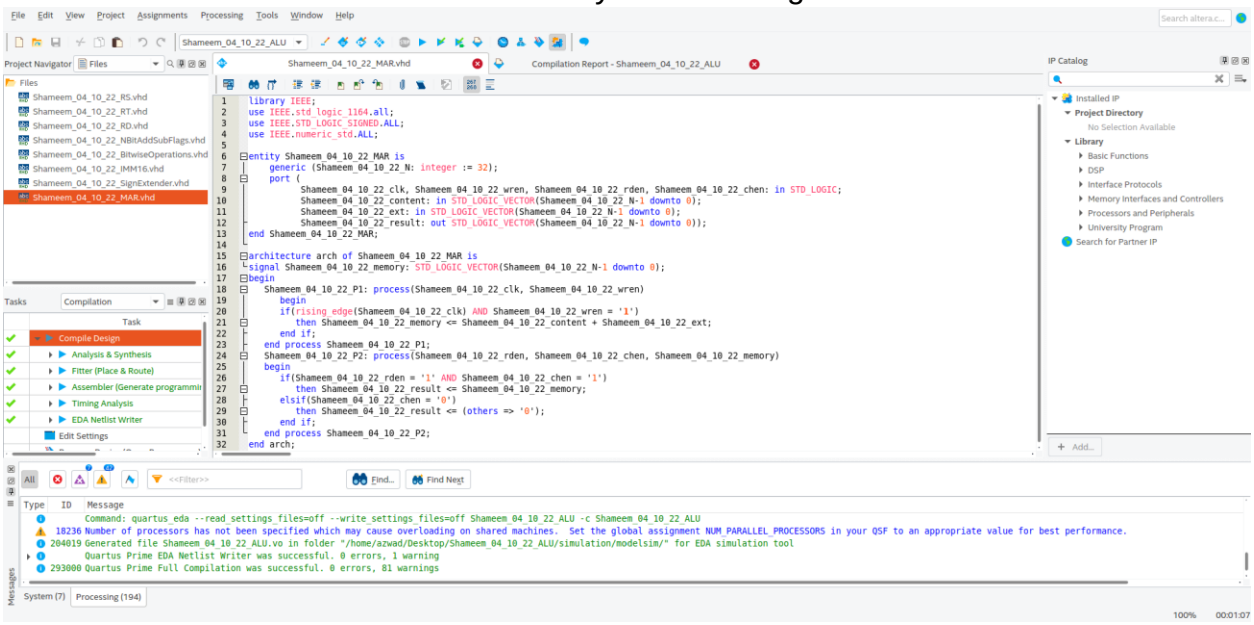


Figure 15: Memory Address Register VHDL code compiled successfully in Quartus.

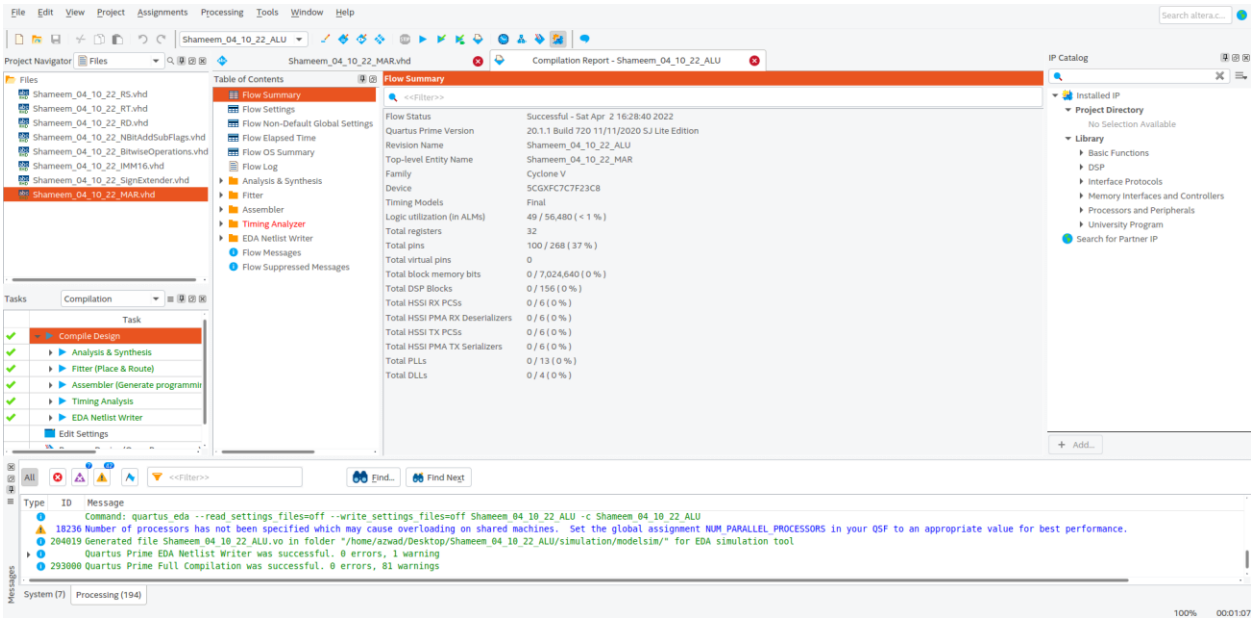


Figure 16: Memory Address Register compilation report in Quartus.

MDR – Memory Data Register

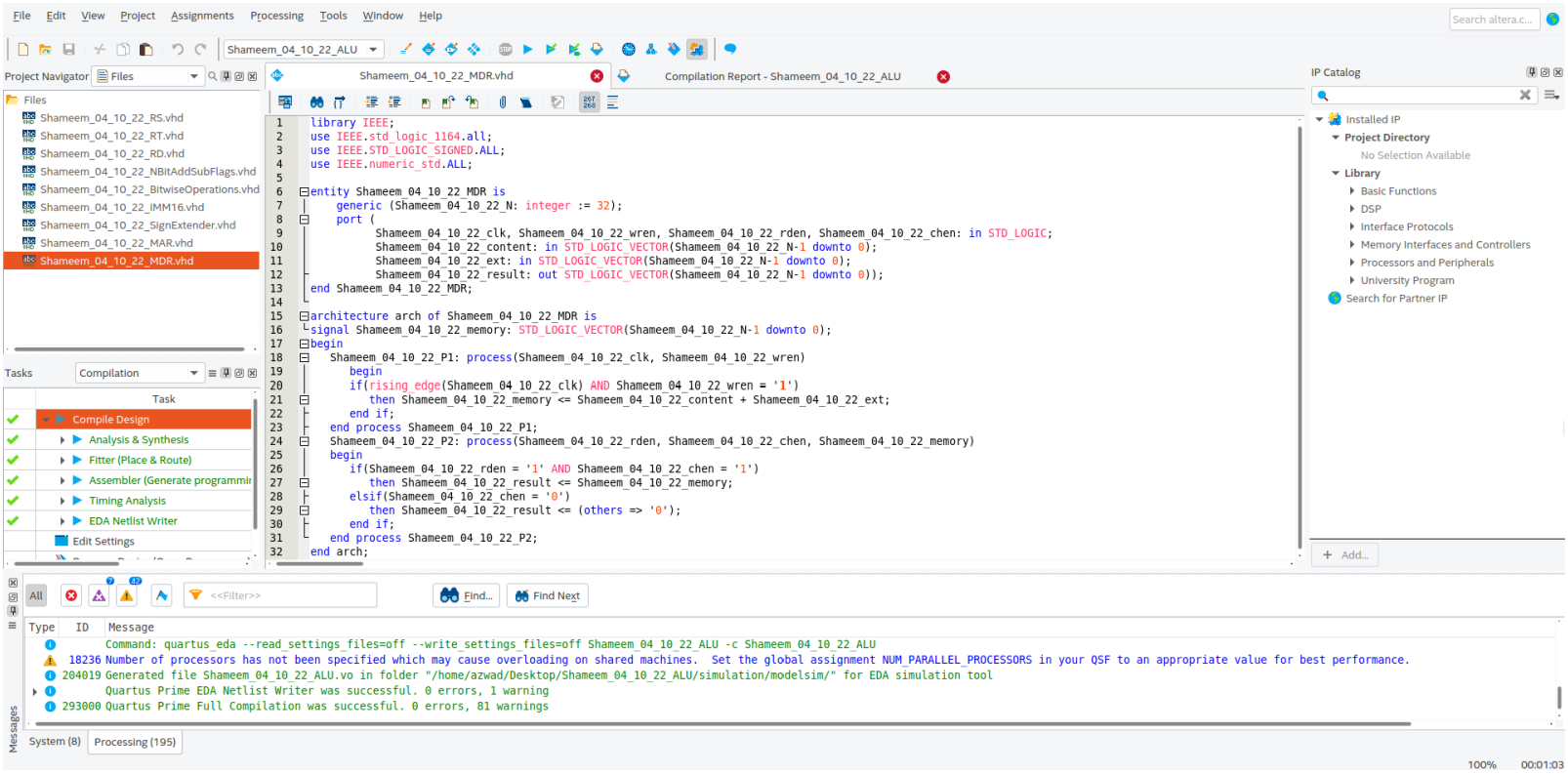


Figure 17: Memory Data Register VHDL code compiled successfully in Quartus.

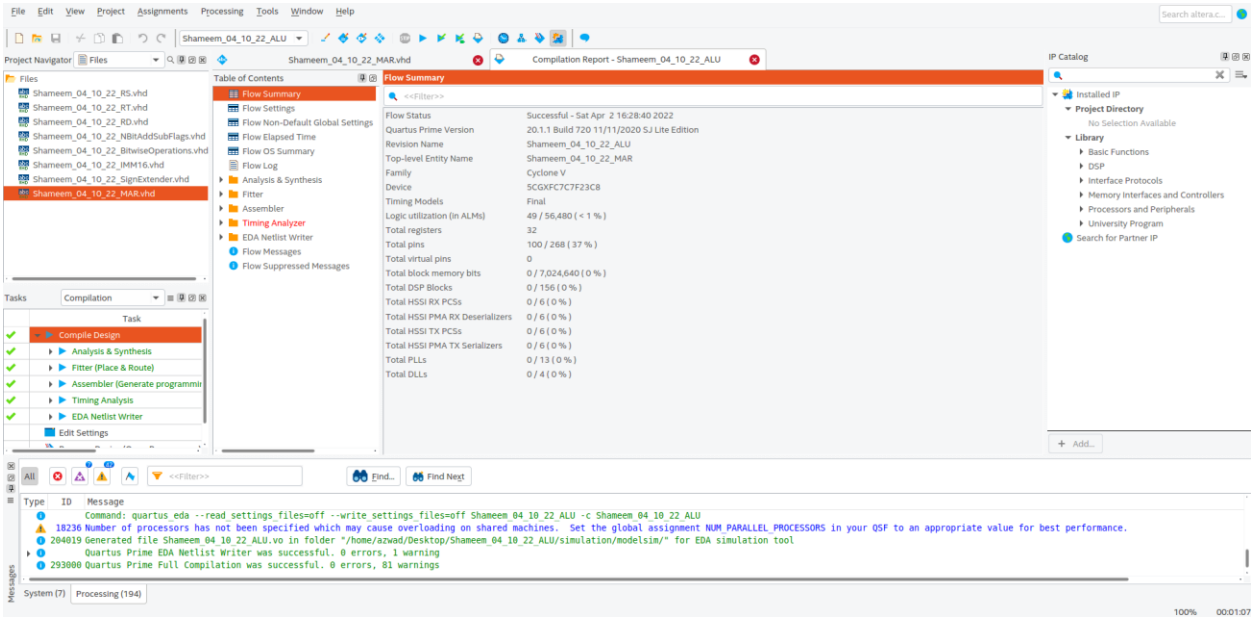


Figure 18: Memory Data Register compilation report in Quartus.

ALU – Arithmetic Logic Unit

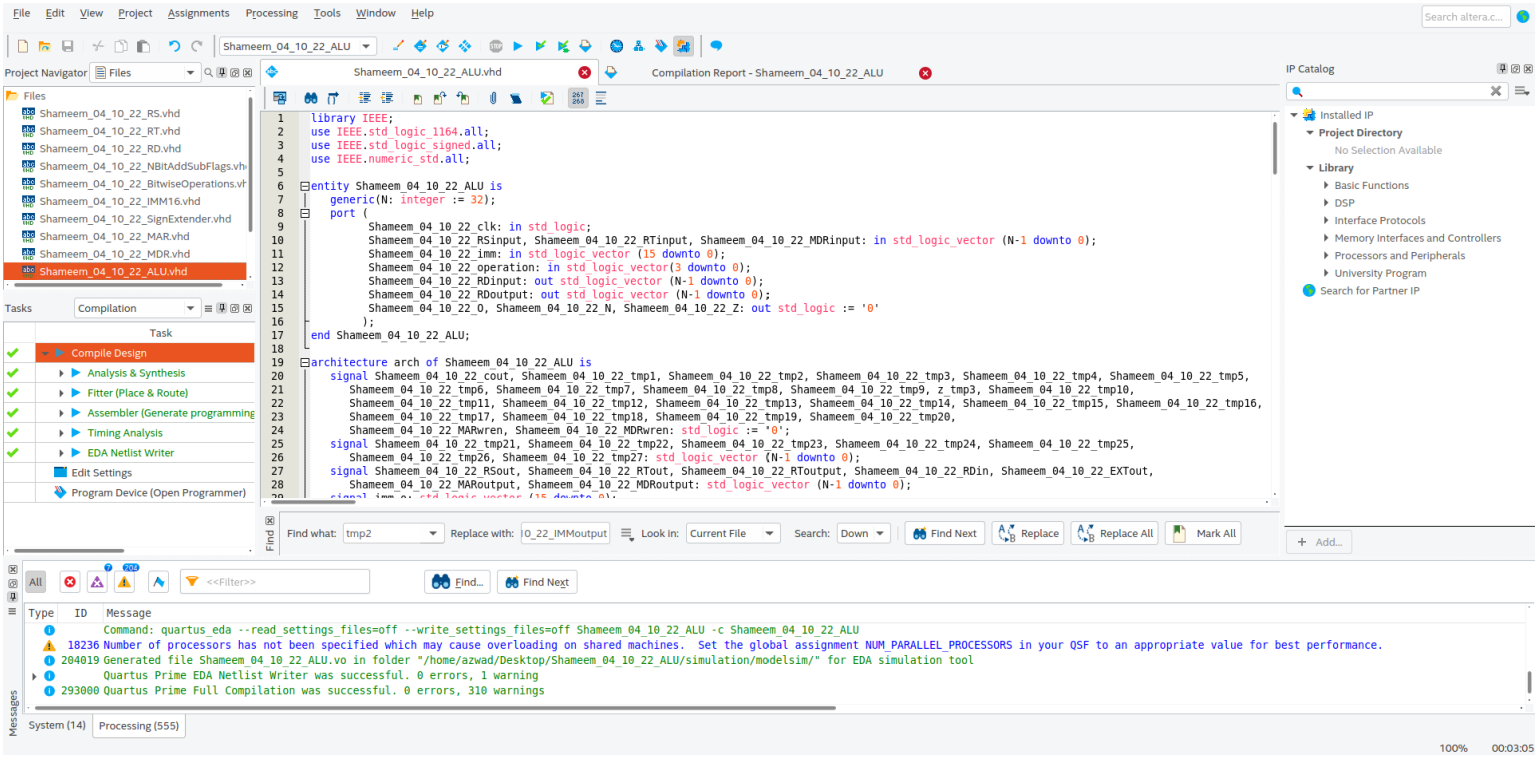


Figure 19: Arithmetic Logic Unit VHDL code compiled successfully in Quartus.

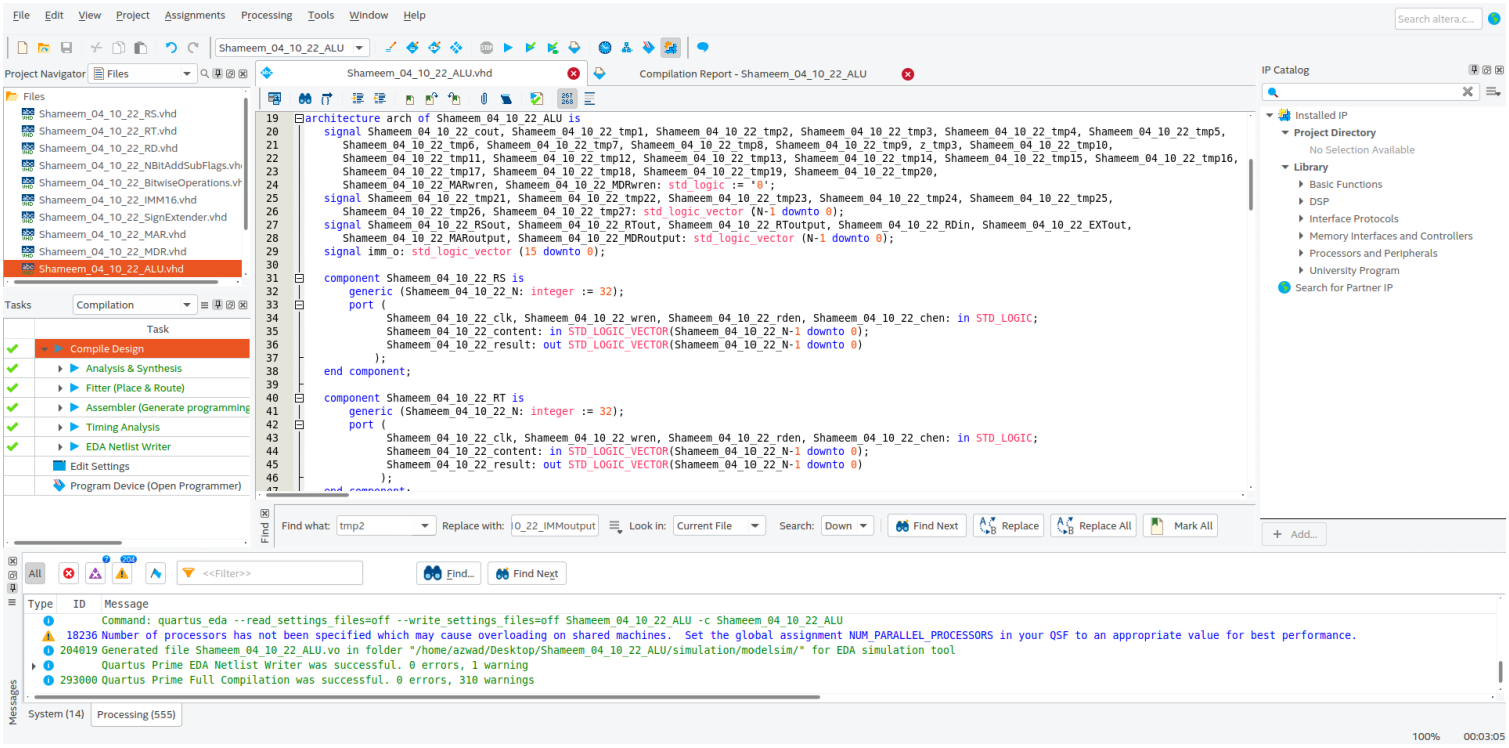


Figure 20: Arithmetic Logic Unit VHDL code continued.

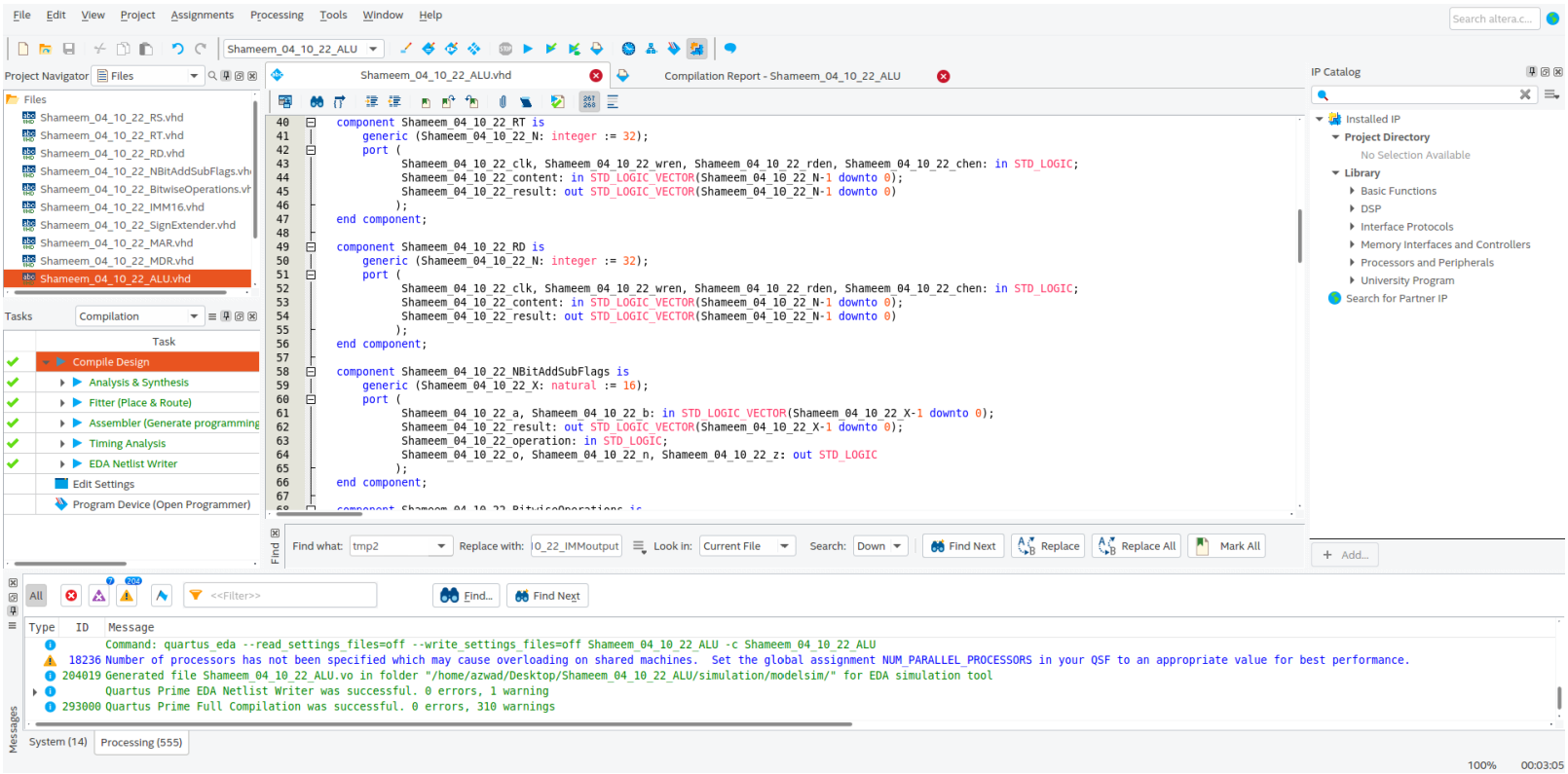


Figure 21: Arithmetic Logic Unit VHDL code continued.

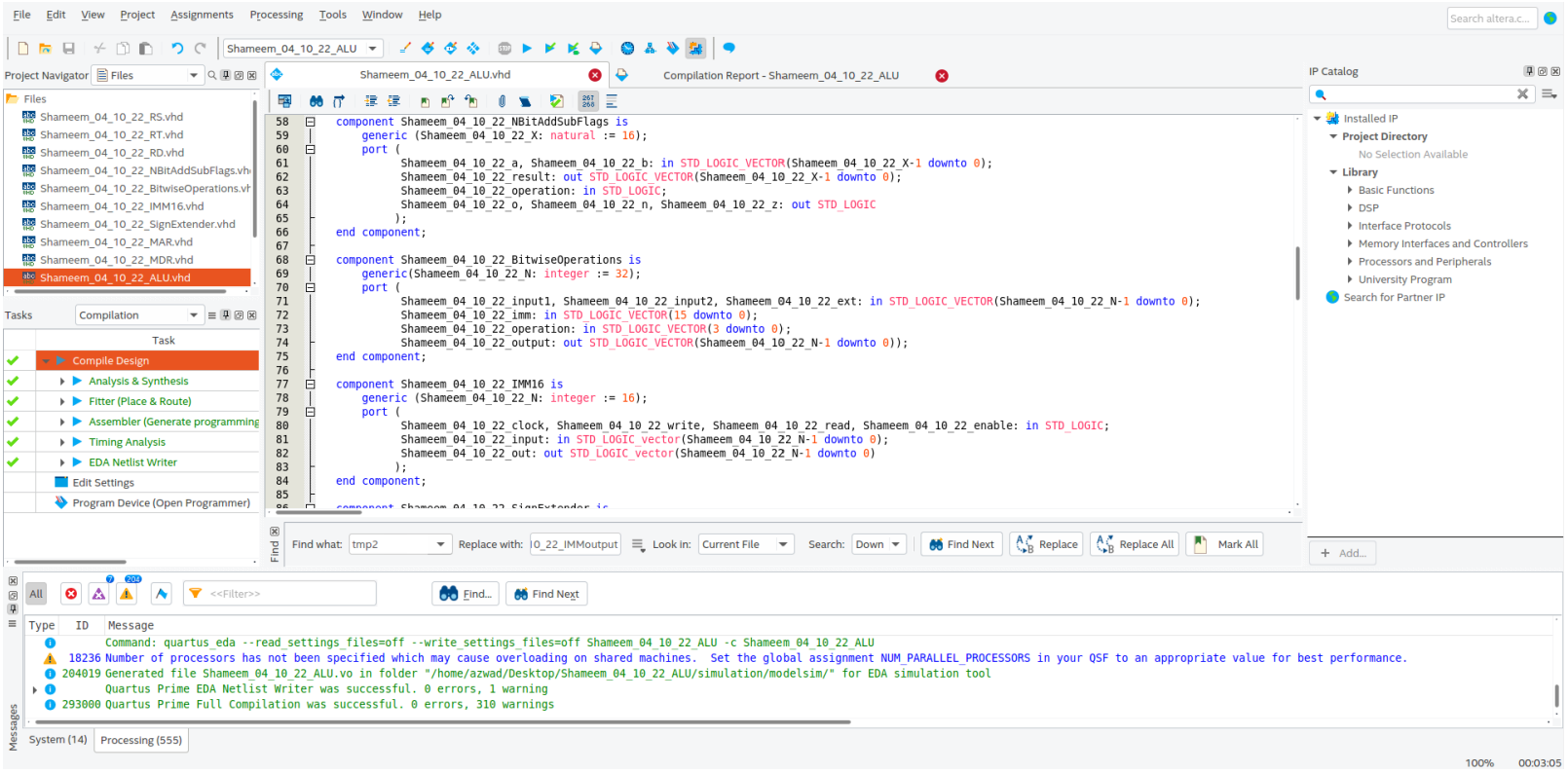


Figure 22: Arithmetic Logic Unit VHDL code continued

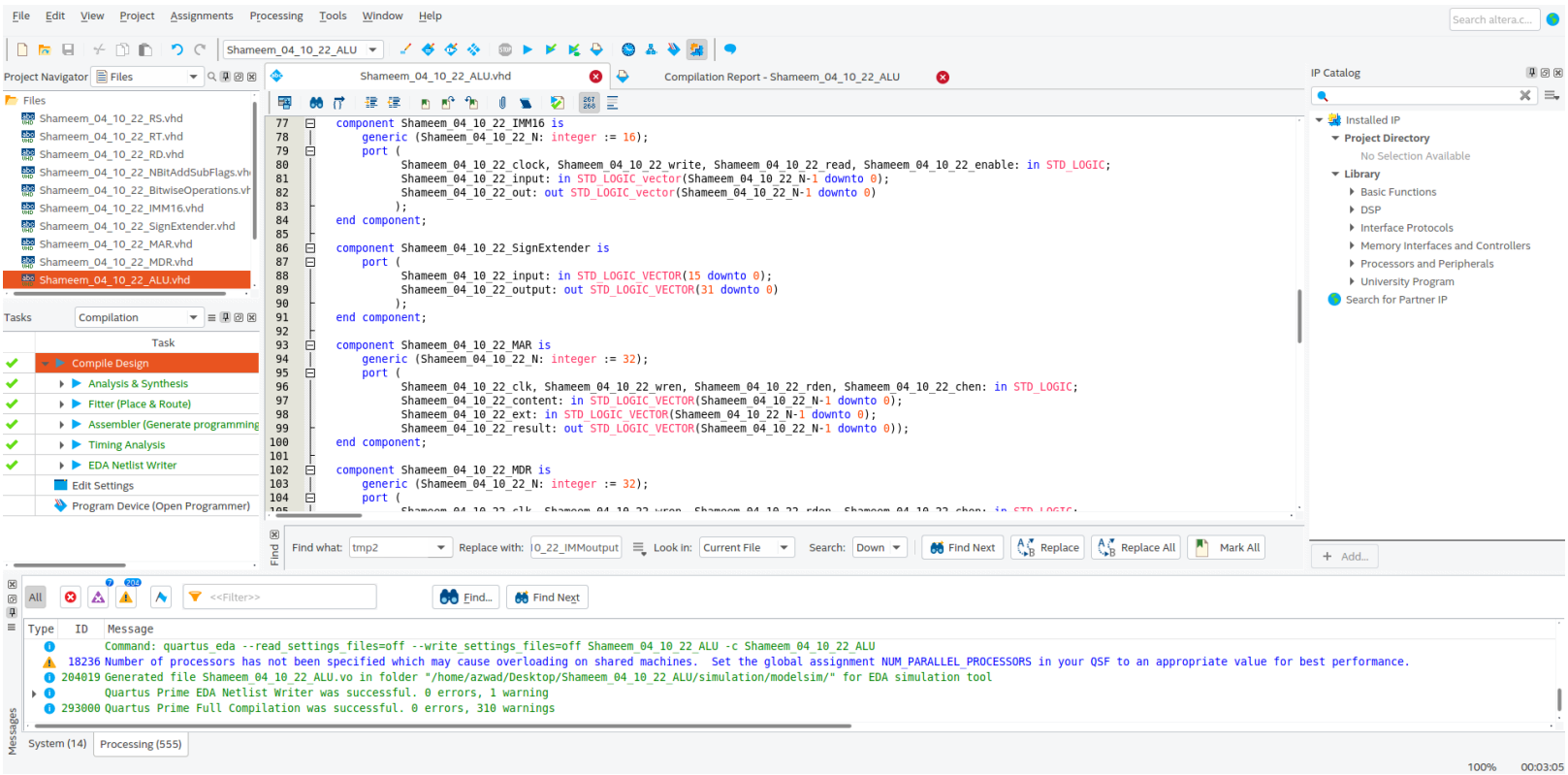


Figure 23: Arithmetic Logic Unit VHDL code continued.

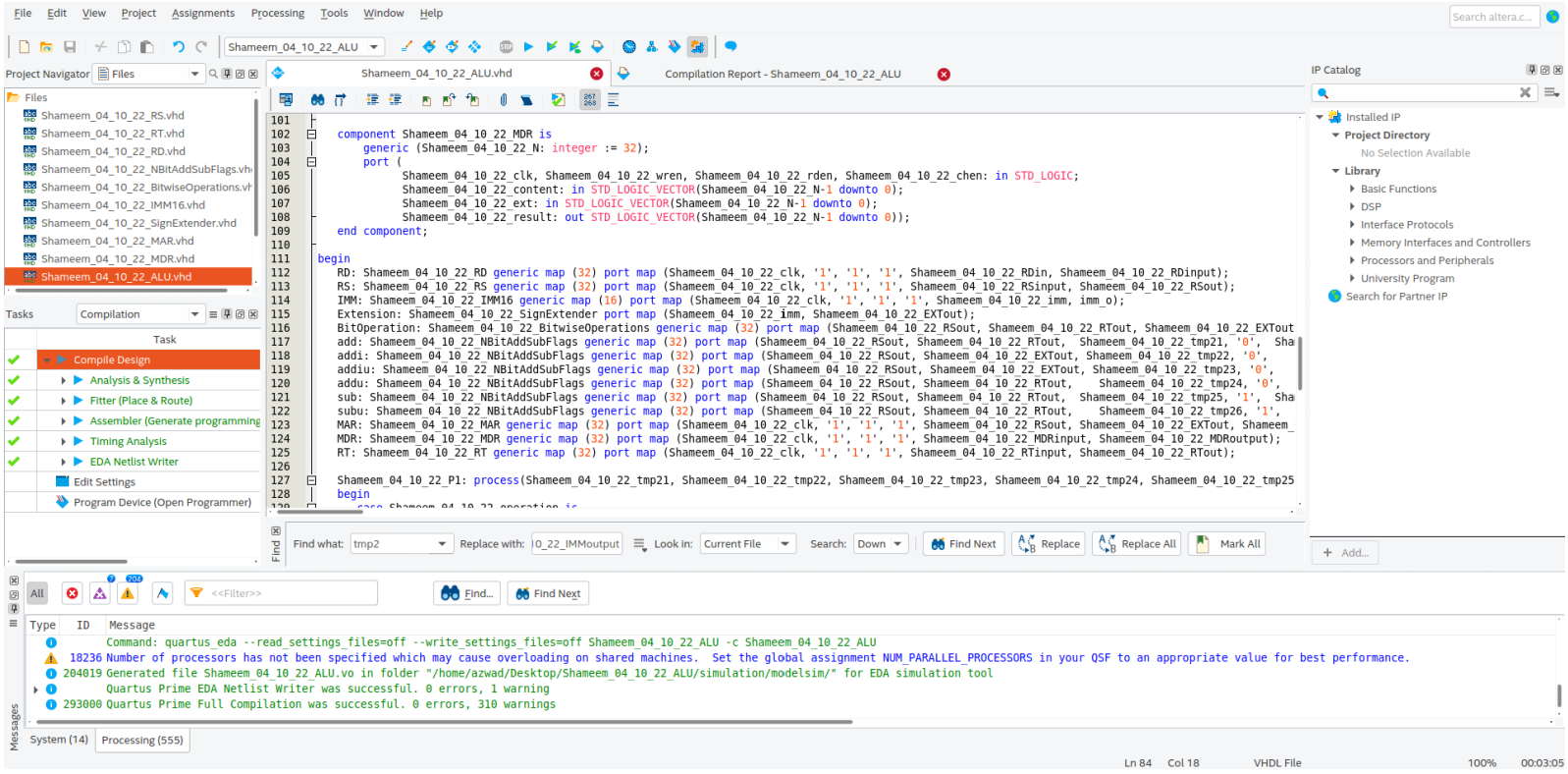


Figure 24: Arithmetic Logic Unit VHDL code continued.

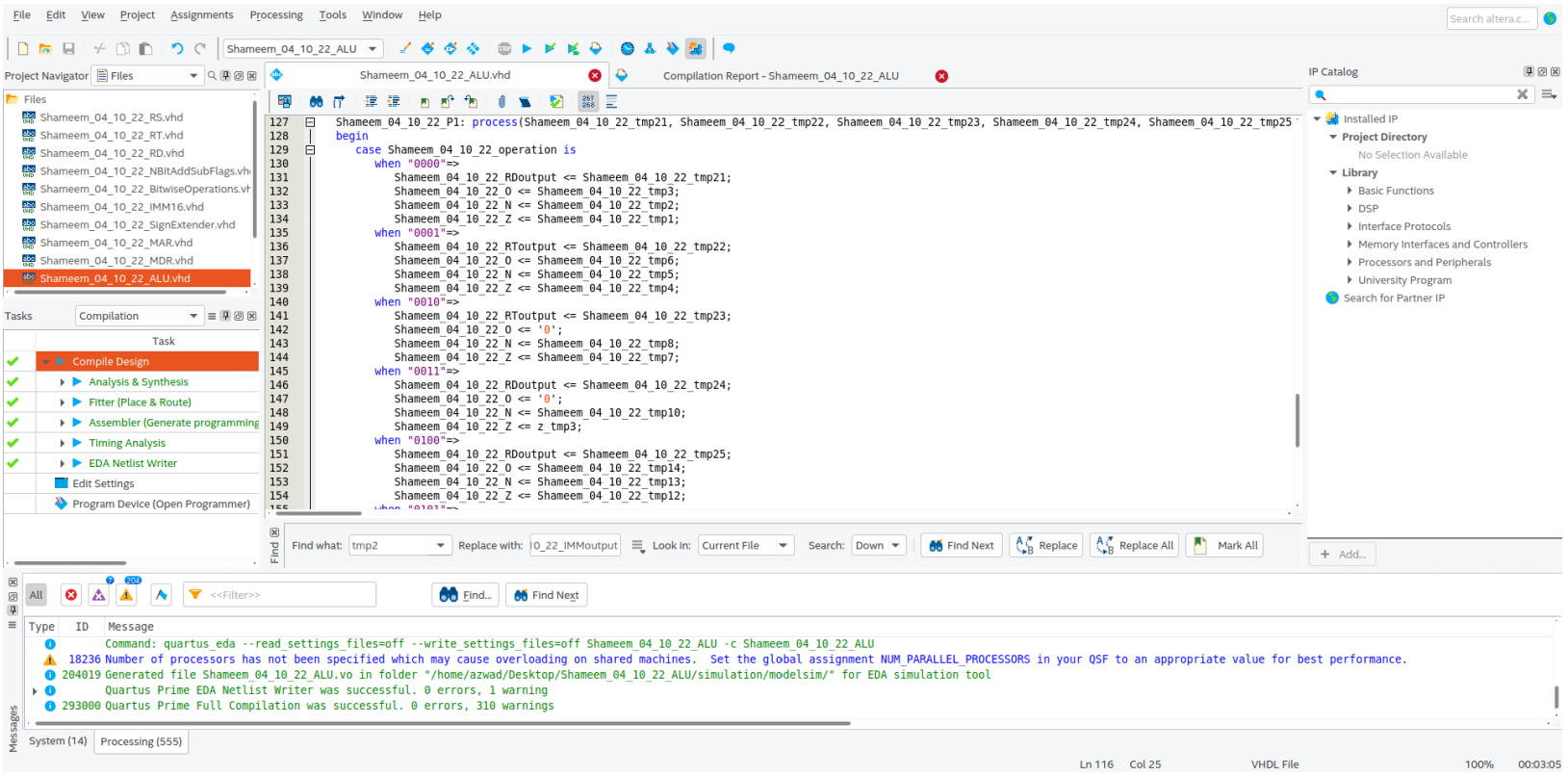


Figure 25: Arithmetic Logic Unit VHDL code continued.

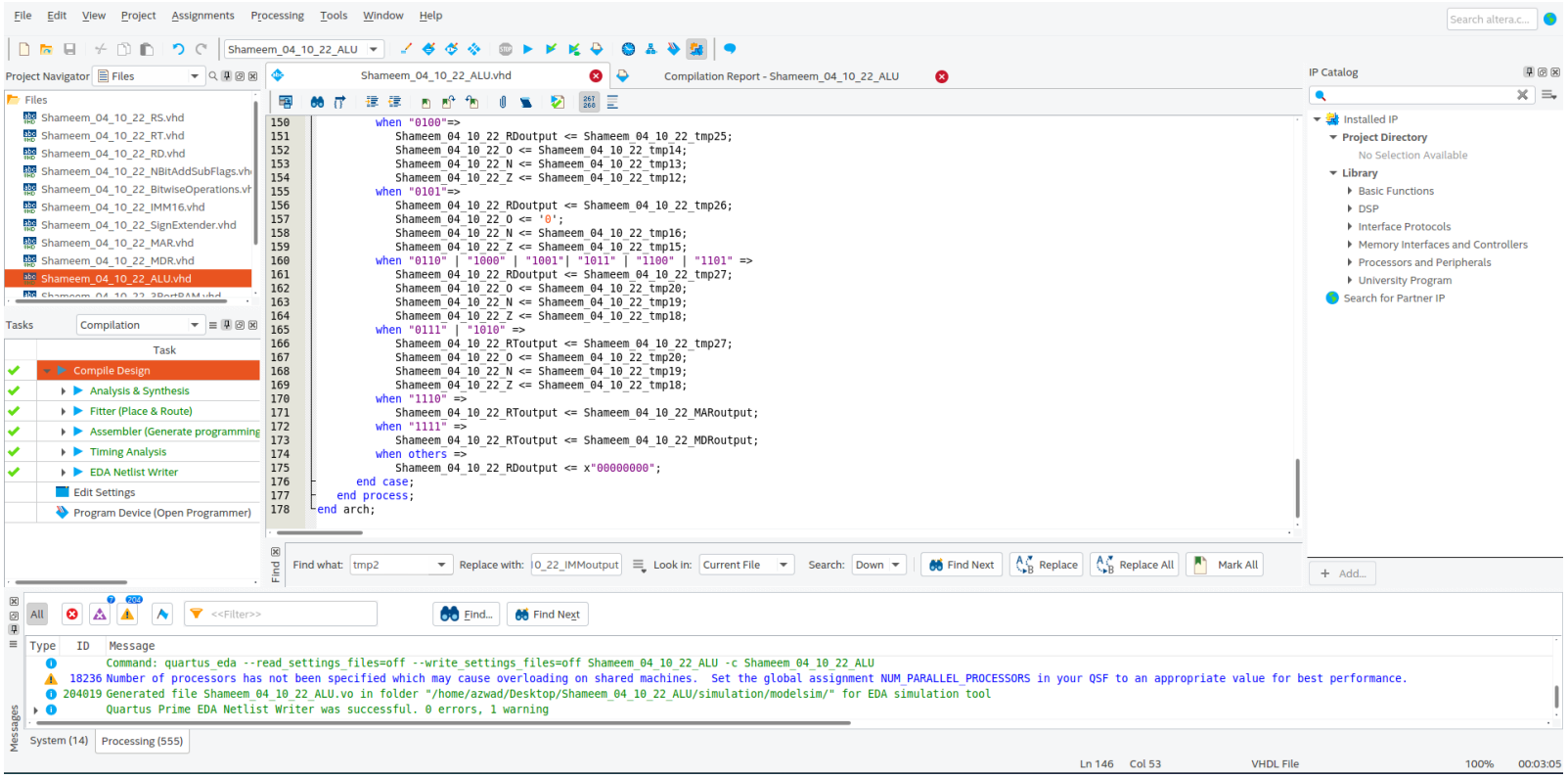


Figure 26: Arithmetic Logic Unit VHDL code continued.

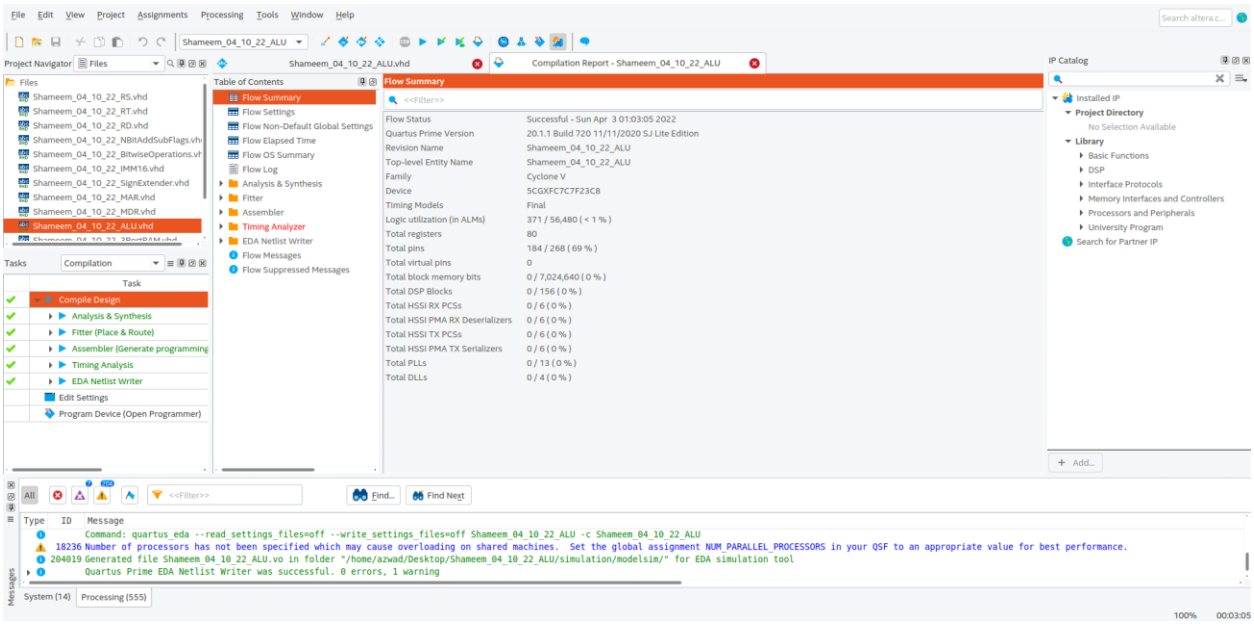
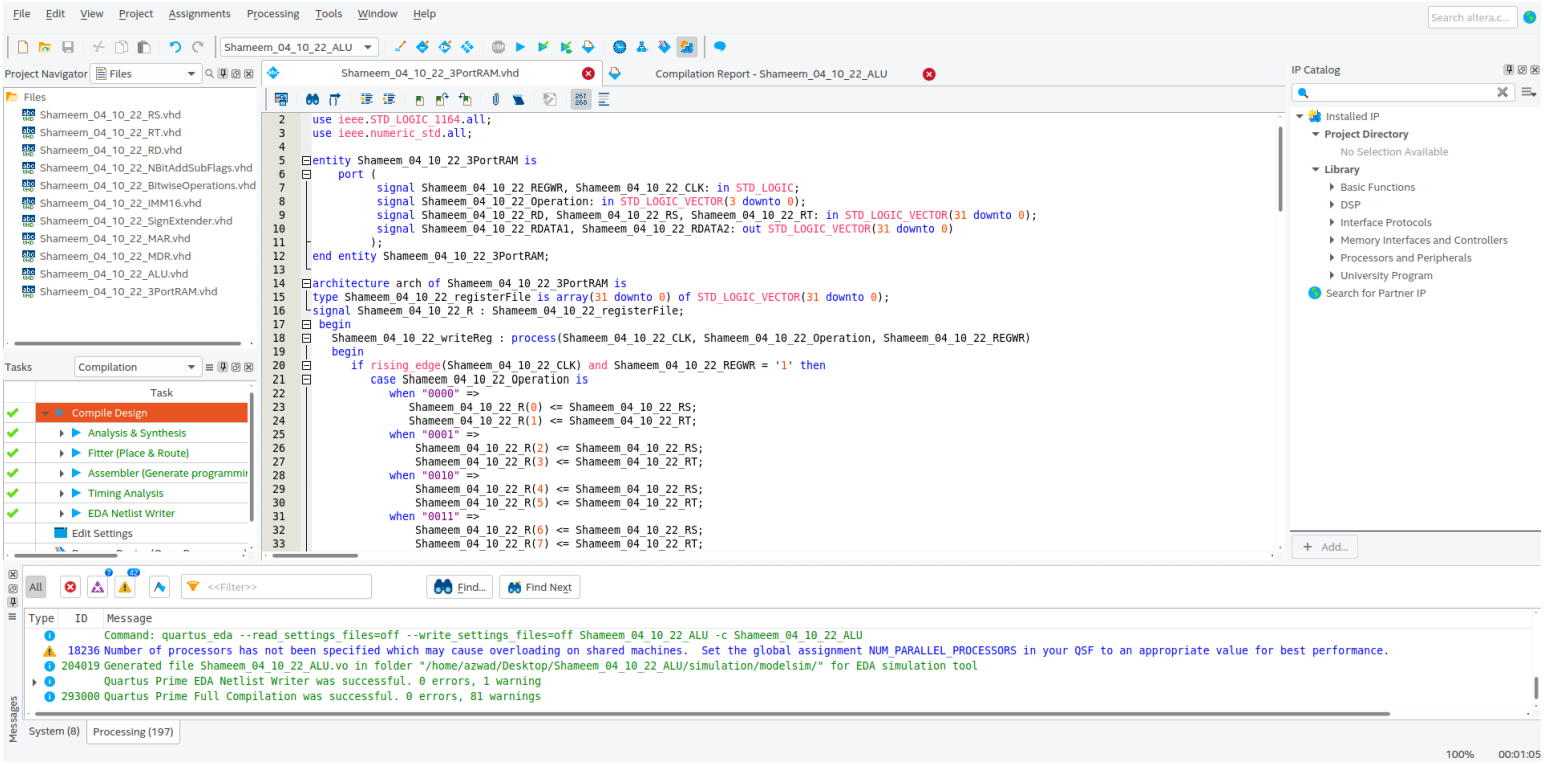


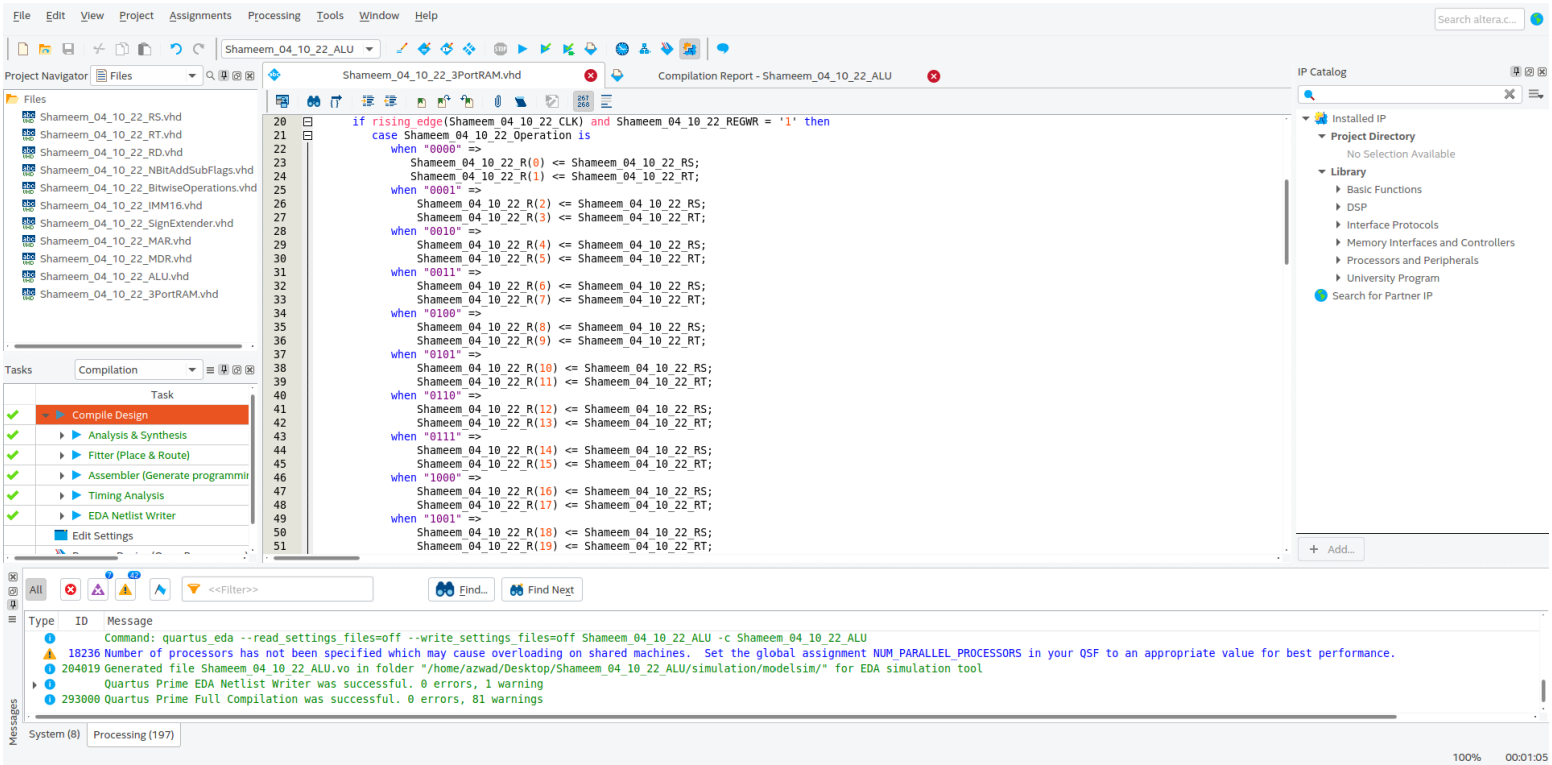
Figure 27: Arithmetic Logic Unit compilation report.

3 Ported RAM



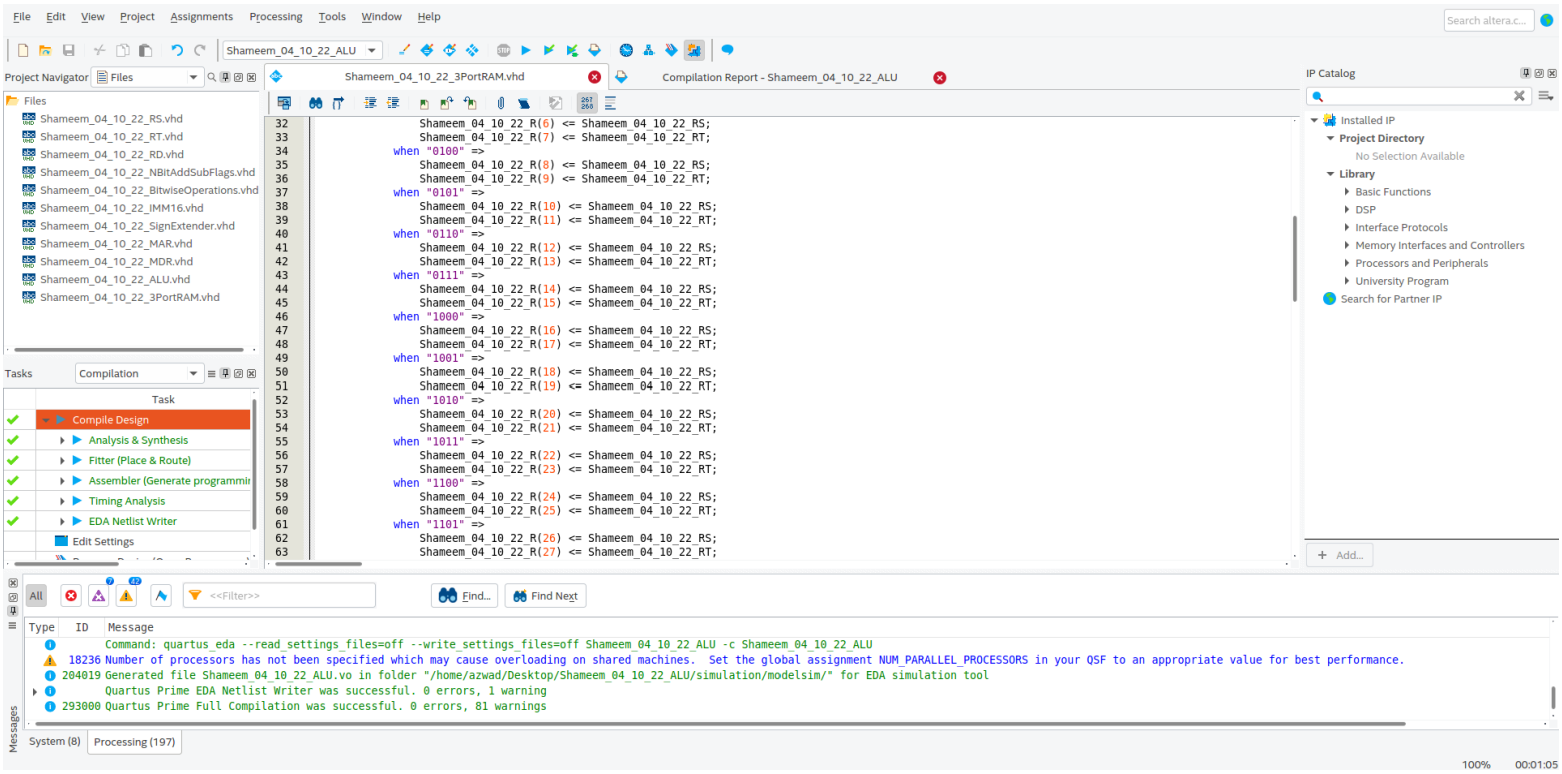

```
use ieee.STD_LOGIC_1164.all;
use ieee.numeric_std.all;

entity Shameem_04_10_22_3PortRAM is
    port (
          signal Shameem_04_10_22_REGWR, Shameem_04_10_22_CLK: in STD_LOGIC;
          signal Shameem_04_10_22_Operation: in STD_LOGIC_VECTOR(3 downto 0);
          signal Shameem_04_10_22_RD, Shameem_04_10_22_RS, Shameem_04_10_22_RT: in STD_LOGIC_VECTOR(31 downto 0);
          signal Shameem_04_10_22_RDATA1, Shameem_04_10_22_RDATA2: out STD_LOGIC_VECTOR(31 downto 0)
         );
end entity Shameem_04_10_22_3PortRAM;

architecture arch of Shameem_04_10_22_3PortRAM is
type Shameem_04_10_22_registerFile is array(31 downto 0) of STD_LOGIC_VECTOR(31 downto 0);
signal Shameem_04_10_22_R : Shameem_04_10_22_registerFile;
begin
  Shameem_04_10_22_writeReg : process(Shameem_04_10_22_CLK, Shameem_04_10_22_Operation, Shameem_04_10_22_REGWR)
  begin
    if rising_edge(Shameem_04_10_22_CLK) and Shameem_04_10_22_REGWR = '1' then
       case Shameem_04_10_22_Operation is
          when "0000" =>
             Shameem_04_10_22_R(0) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(1) <= Shameem_04_10_22_RT;
          when "0001" =>
             Shameem_04_10_22_R(2) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(3) <= Shameem_04_10_22_RT;
          when "0010" =>
             Shameem_04_10_22_R(4) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(5) <= Shameem_04_10_22_RT;
          when "0011" =>
             Shameem_04_10_22_R(6) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(7) <= Shameem_04_10_22_RT;
```

Figure 28: 3 Ported RAM VHDL code compiled successfully on Quartus.



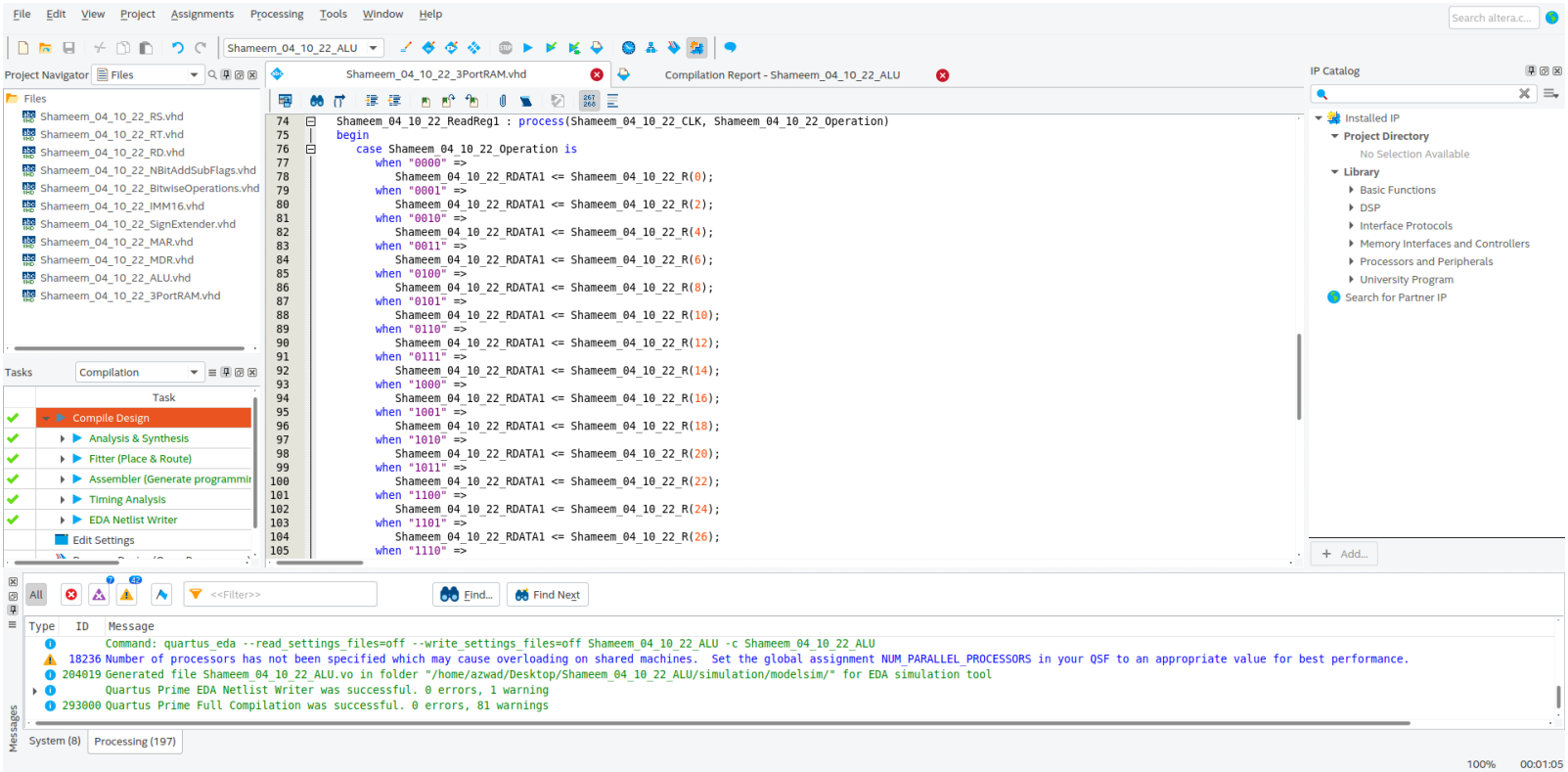

```
    if rising_edge(Shameem_04_10_22_CLK) and Shameem_04_10_22_REGWR = '1' then
       case Shameem_04_10_22_Operation is
          when "0000" =>
             Shameem_04_10_22_R(0) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(1) <= Shameem_04_10_22_RT;
          when "0001" =>
             Shameem_04_10_22_R(2) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(3) <= Shameem_04_10_22_RT;
          when "0010" =>
             Shameem_04_10_22_R(4) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(5) <= Shameem_04_10_22_RT;
          when "0011" =>
             Shameem_04_10_22_R(6) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(7) <= Shameem_04_10_22_RT;
          when "0100" =>
             Shameem_04_10_22_R(8) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(9) <= Shameem_04_10_22_RT;
          when "0101" =>
             Shameem_04_10_22_R(10) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(11) <= Shameem_04_10_22_RT;
          when "0110" =>
             Shameem_04_10_22_R(12) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(13) <= Shameem_04_10_22_RT;
          when "0111" =>
             Shameem_04_10_22_R(14) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(15) <= Shameem_04_10_22_RT;
          when "1000" =>
             Shameem_04_10_22_R(16) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(17) <= Shameem_04_10_22_RT;
          when "1001" =>
             Shameem_04_10_22_R(18) <= Shameem_04_10_22_RS;
             Shameem_04_10_22_R(19) <= Shameem_04_10_22_RT;
```

Figure 29: 3 Ported RAM VHDL code continued.




```
32                  Shameem_04_10_22_R(6) <= Shameem_04_10_22_RS;
33                  Shameem_04_10_22_R(7) <= Shameem_04_10_22_RT;
34              when "0100" =>
35                  Shameem_04_10_22_R(8) <= Shameem_04_10_22_RS;
36                  Shameem_04_10_22_R(9) <= Shameem_04_10_22_RT;
37              when "0101" =>
38                  Shameem_04_10_22_R(10) <= Shameem_04_10_22_RS;
39                  Shameem_04_10_22_R(11) <= Shameem_04_10_22_RT;
40              when "0110" =>
41                  Shameem_04_10_22_R(12) <= Shameem_04_10_22_RS;
42                  Shameem_04_10_22_R(13) <= Shameem_04_10_22_RT;
43              when "0111" =>
44                  Shameem_04_10_22_R(14) <= Shameem_04_10_22_RS;
45                  Shameem_04_10_22_R(15) <= Shameem_04_10_22_RT;
46              when "1000" =>
47                  Shameem_04_10_22_R(16) <= Shameem_04_10_22_RS;
48                  Shameem_04_10_22_R(17) <= Shameem_04_10_22_RT;
49              when "1001" =>
50                  Shameem_04_10_22_R(18) <= Shameem_04_10_22_RS;
51                  Shameem_04_10_22_R(19) <= Shameem_04_10_22_RT;
52              when "1010" =>
53                  Shameem_04_10_22_R(20) <= Shameem_04_10_22_RS;
54                  Shameem_04_10_22_R(21) <= Shameem_04_10_22_RT;
55              when "1011" =>
56                  Shameem_04_10_22_R(22) <= Shameem_04_10_22_RS;
57                  Shameem_04_10_22_R(23) <= Shameem_04_10_22_RT;
58              when "1100" =>
59                  Shameem_04_10_22_R(24) <= Shameem_04_10_22_RS;
60                  Shameem_04_10_22_R(25) <= Shameem_04_10_22_RT;
61              when "1101" =>
62                  Shameem_04_10_22_R(26) <= Shameem_04_10_22_RS;
63                  Shameem_04_10_22_R(27) <= Shameem_04_10_22_RT;
```

Figure 30: 3 Ported RAM VHDL code continued.




```
74  Shameem_04_10_22_ReadReg1 : process(Shameem_04_10_22_CLK, Shameem_04_10_22_Operation)
75  begin
76      case Shameem_04_10_22_Operation is
77          when "0000" =>
78              Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(0);
79          when "0001" =>
80              Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(2);
81          when "0010" =>
82              Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(4);
83          when "0011" =>
84              Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(6);
85          when "0100" =>
86              Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(8);
87          when "0101" =>
88              Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(10);
89          when "0110" =>
90              Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(12);
91          when "0111" =>
92              Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(14);
93          when "1000" =>
94              Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(16);
95          when "1001" =>
96              Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(18);
97          when "1010" =>
98              Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(20);
99          when "1011" =>
100             Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(22);
101         when "1100" =>
102             Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(24);
103         when "1101" =>
104             Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(26);
105         when "1110" =>
```

Figure 31: 3 Ported RAM VHDL code continued.

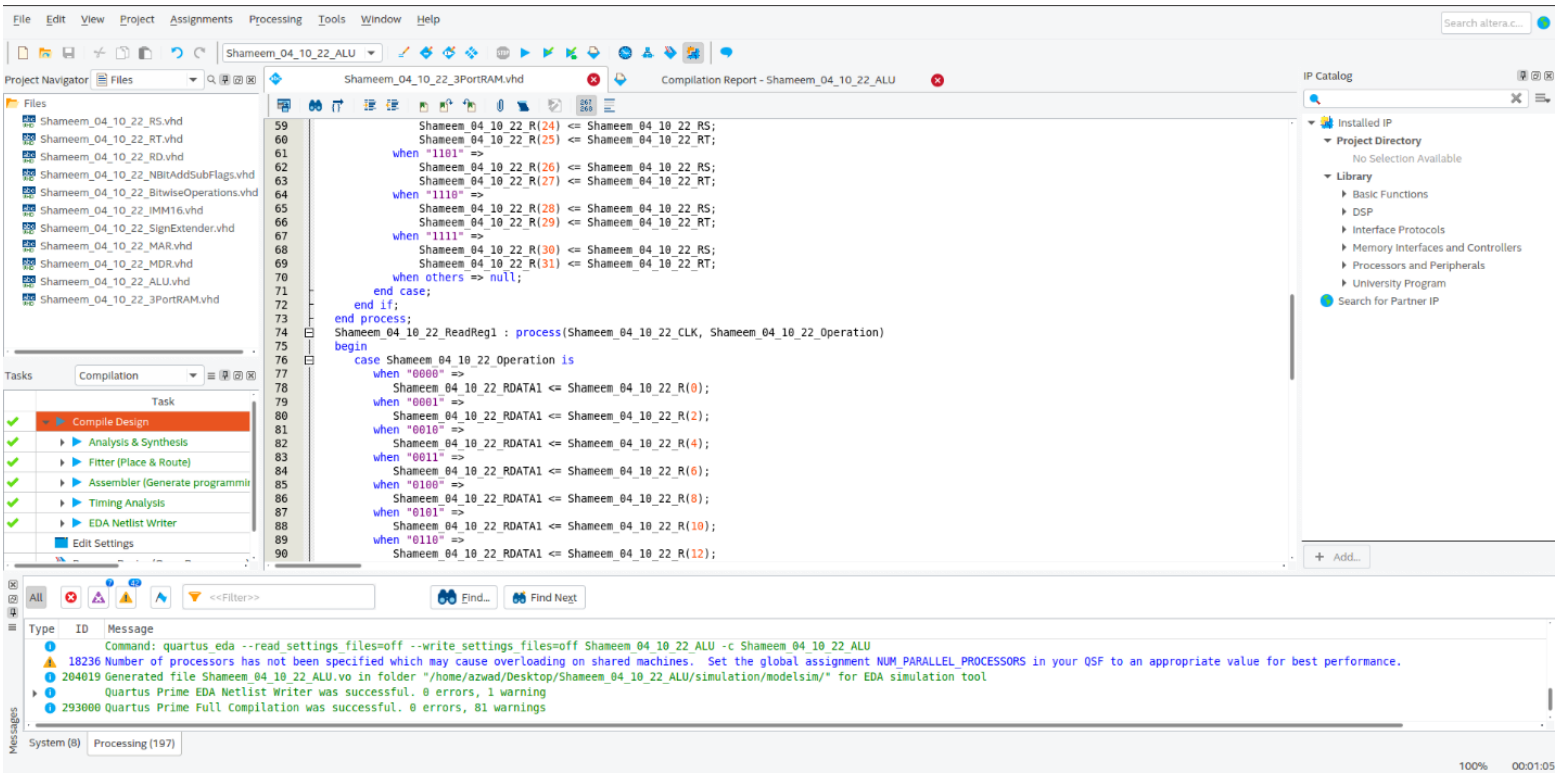


Figure 32: 3 Ported RAM VHDL code continued.

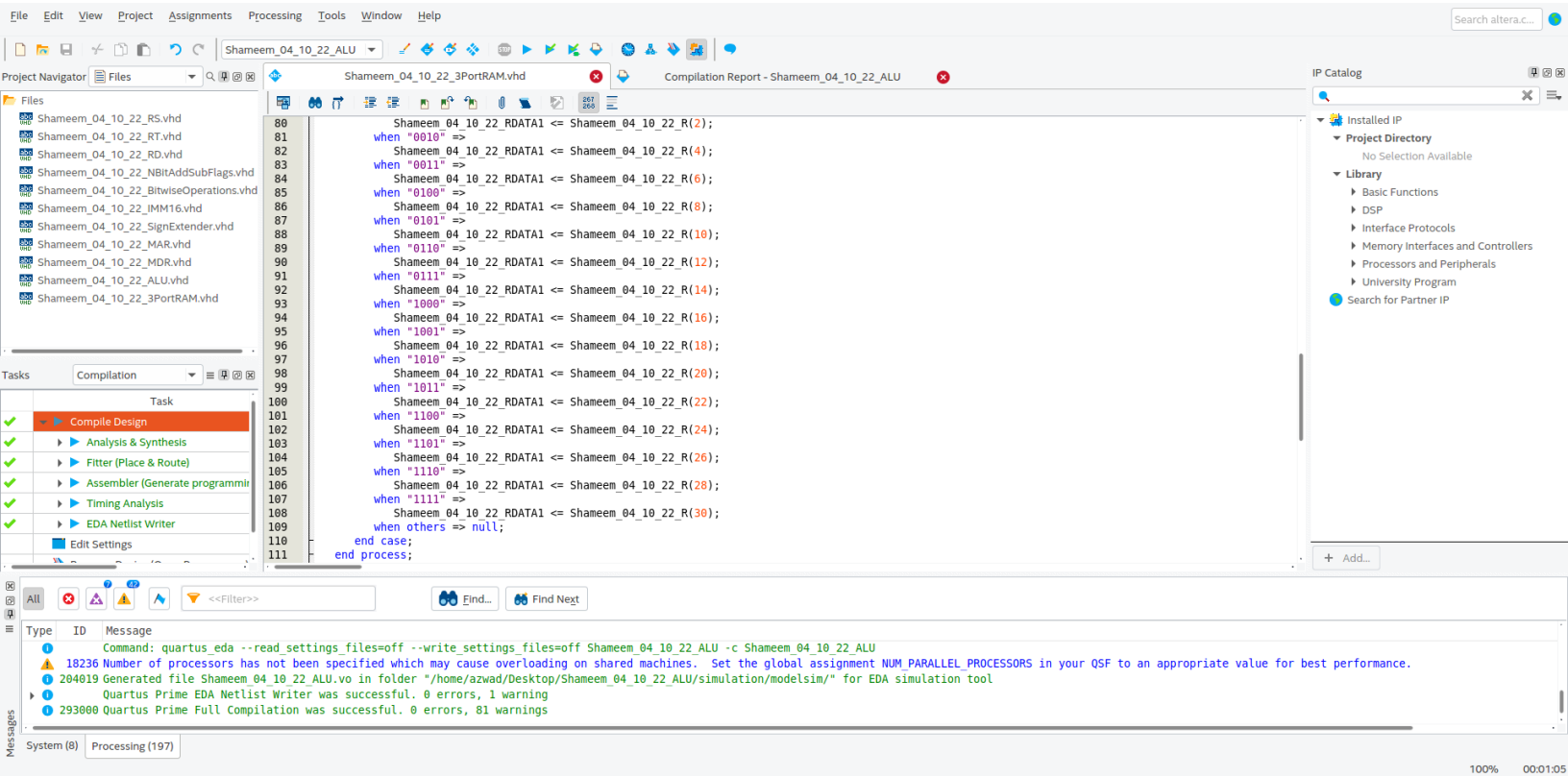
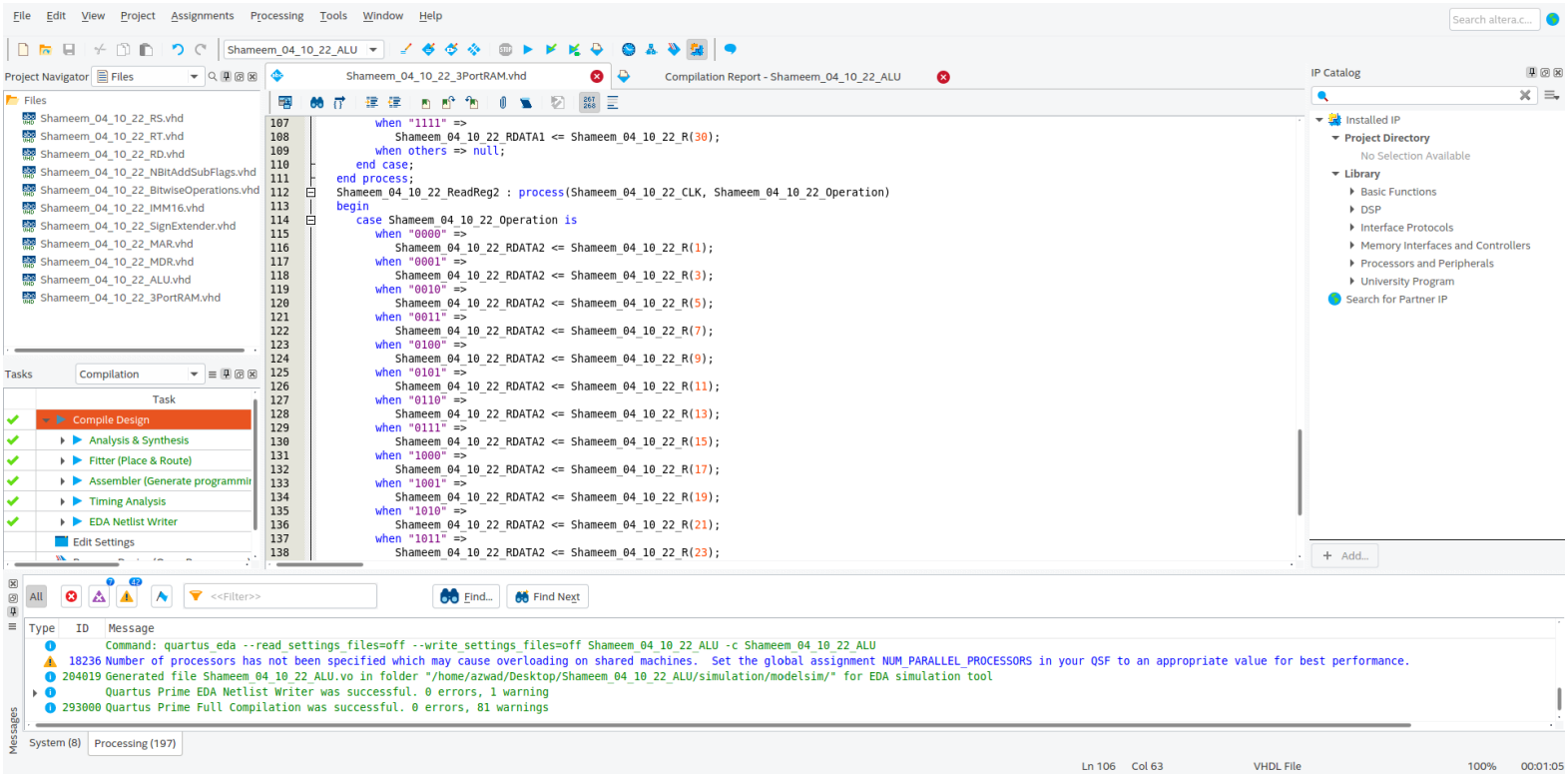
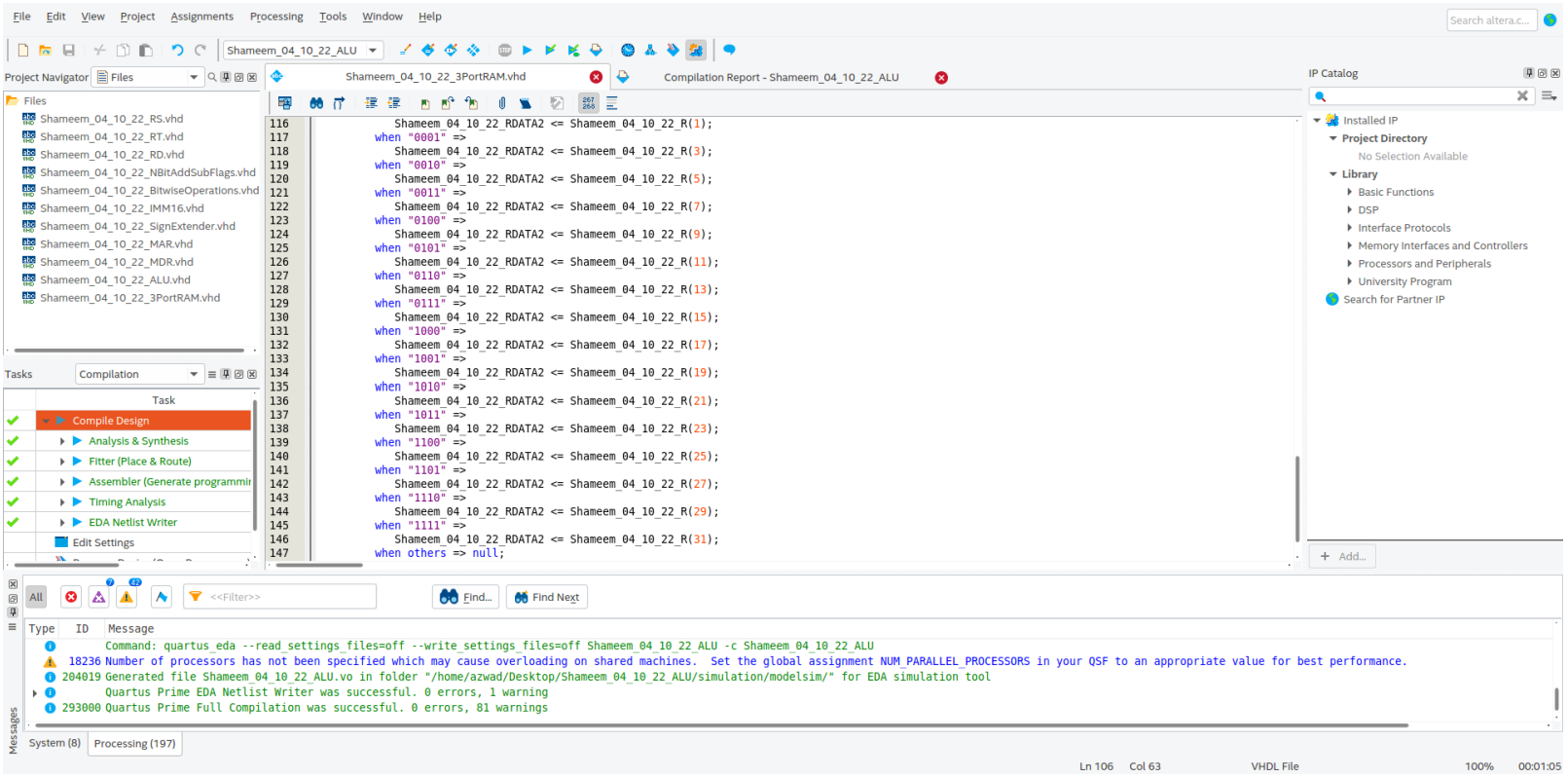


Figure 33: 3 Ported RAM VHDL code continued.




```
107             when "1111" =>
108                 Shameem_04_10_22_RDATA1 <= Shameem_04_10_22_R(30);
109             when others => null;
110         end case;
111     end process;
112     Shameem_04_10_22_ReadReg2 : process(Shameem_04_10_22_CLK, Shameem_04_10_22_Operation)
113     begin
114         case Shameem_04_10_22_Operation is
115             when "0000" =>
116                 Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(1);
117             when "0001" =>
118                 Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(3);
119             when "0010" =>
120                 Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(5);
121             when "0011" =>
122                 Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(7);
123             when "0100" =>
124                 Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(9);
125             when "0101" =>
126                 Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(11);
127             when "0110" =>
128                 Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(13);
129             when "0111" =>
130                 Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(15);
131             when "1000" =>
132                 Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(17);
133             when "1001" =>
134                 Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(19);
135             when "1010" =>
136                 Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(21);
137             when "1011" =>
138                 Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(23);
```

Figure 34: 3 Ported RAM VHDL code continued.




```
116             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(1);
117         when "0001" =>
118             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(3);
119         when "0010" =>
120             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(5);
121         when "0011" =>
122             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(7);
123         when "0100" =>
124             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(9);
125         when "0101" =>
126             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(11);
127         when "0110" =>
128             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(13);
129         when "0111" =>
130             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(15);
131         when "1000" =>
132             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(17);
133         when "1001" =>
134             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(19);
135         when "1010" =>
136             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(21);
137         when "1011" =>
138             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(23);
139         when "1100" =>
140             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(25);
141         when "1101" =>
142             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(27);
143         when "1110" =>
144             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(29);
145         when "1111" =>
146             Shameem_04_10_22_RDATA2 <= Shameem_04_10_22_R(31);
147         when others => null;
```

Figure 35: 3 Ported RAM VHDL code continued.

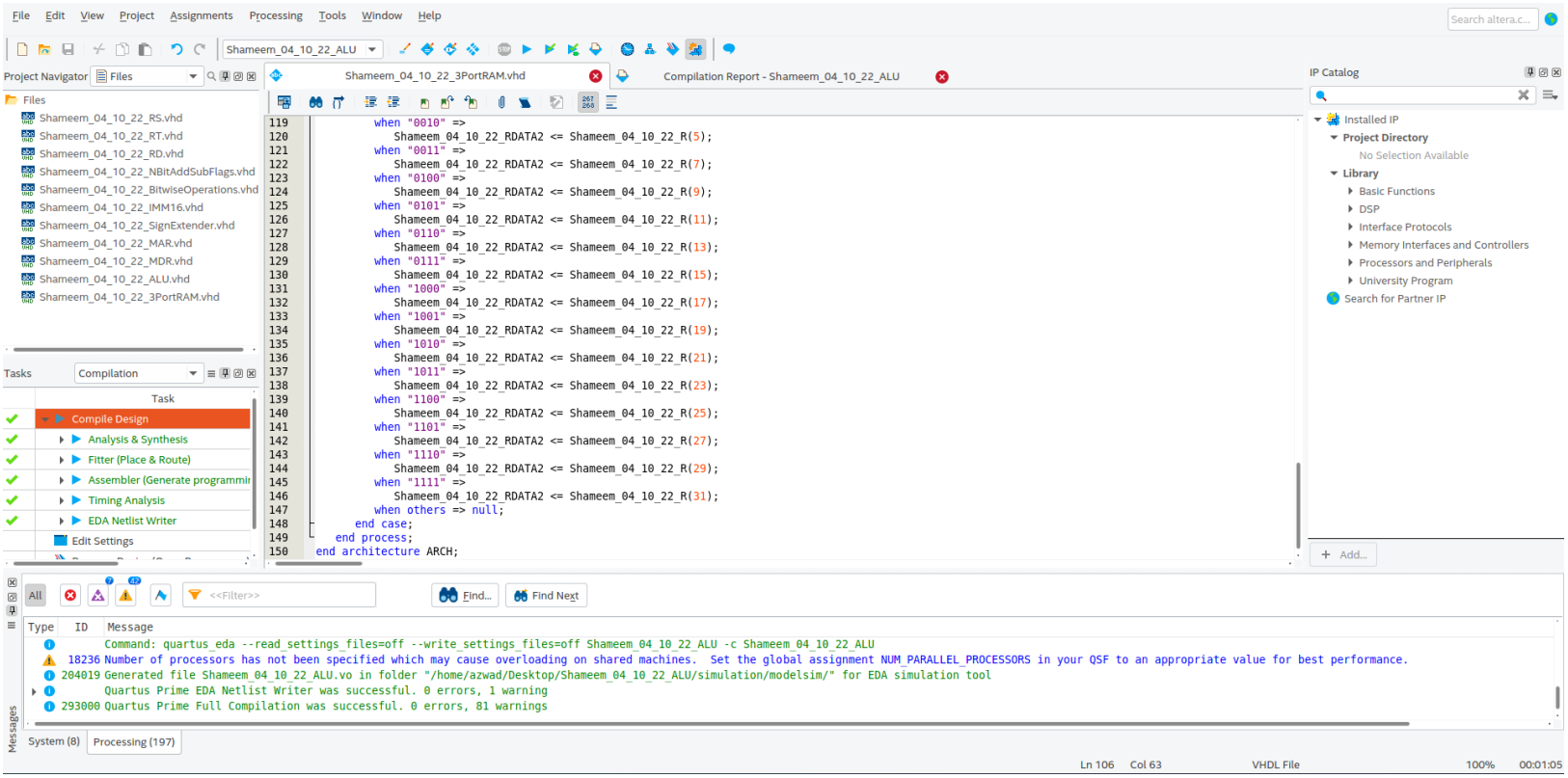


Figure 36: 3 Ported RAM VHDL code continued.

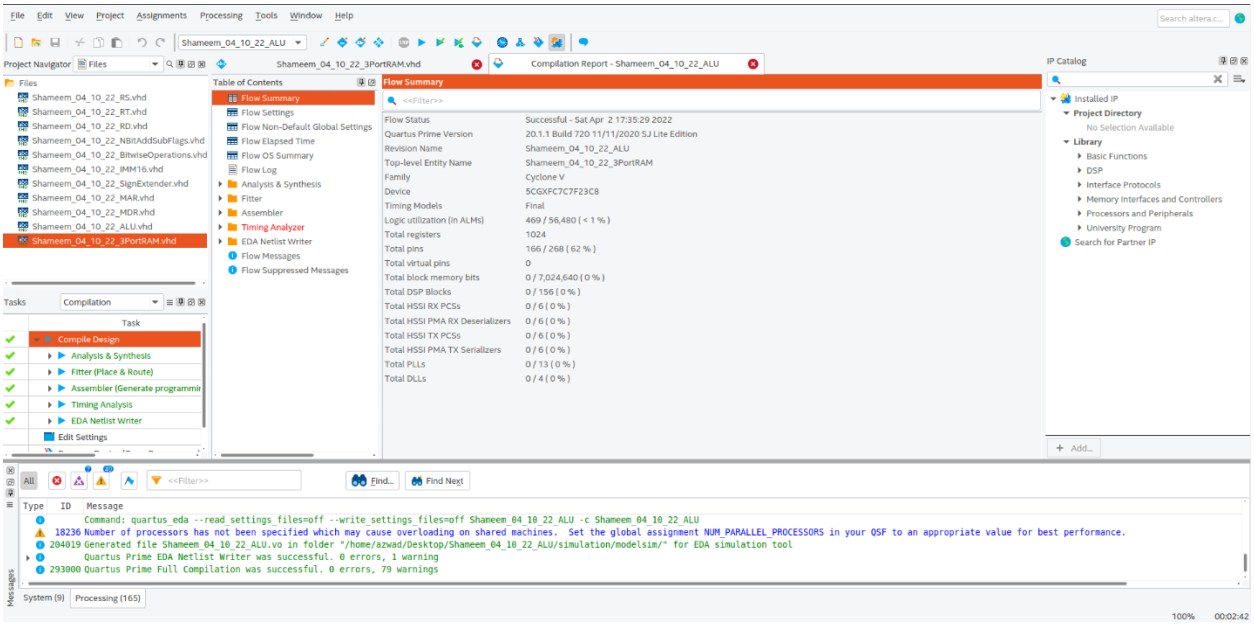
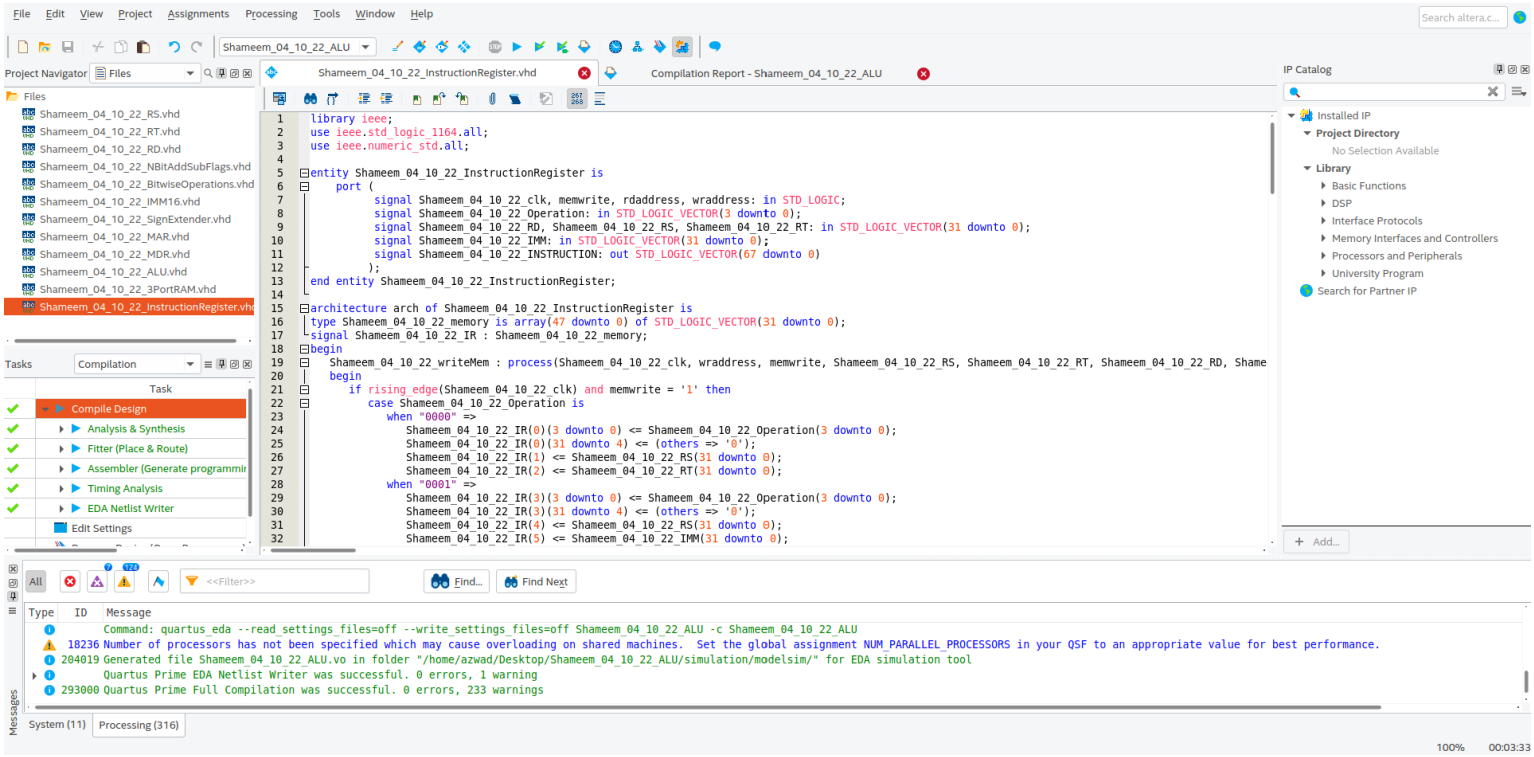


Figure 37: 3 Ported RAM compilation report

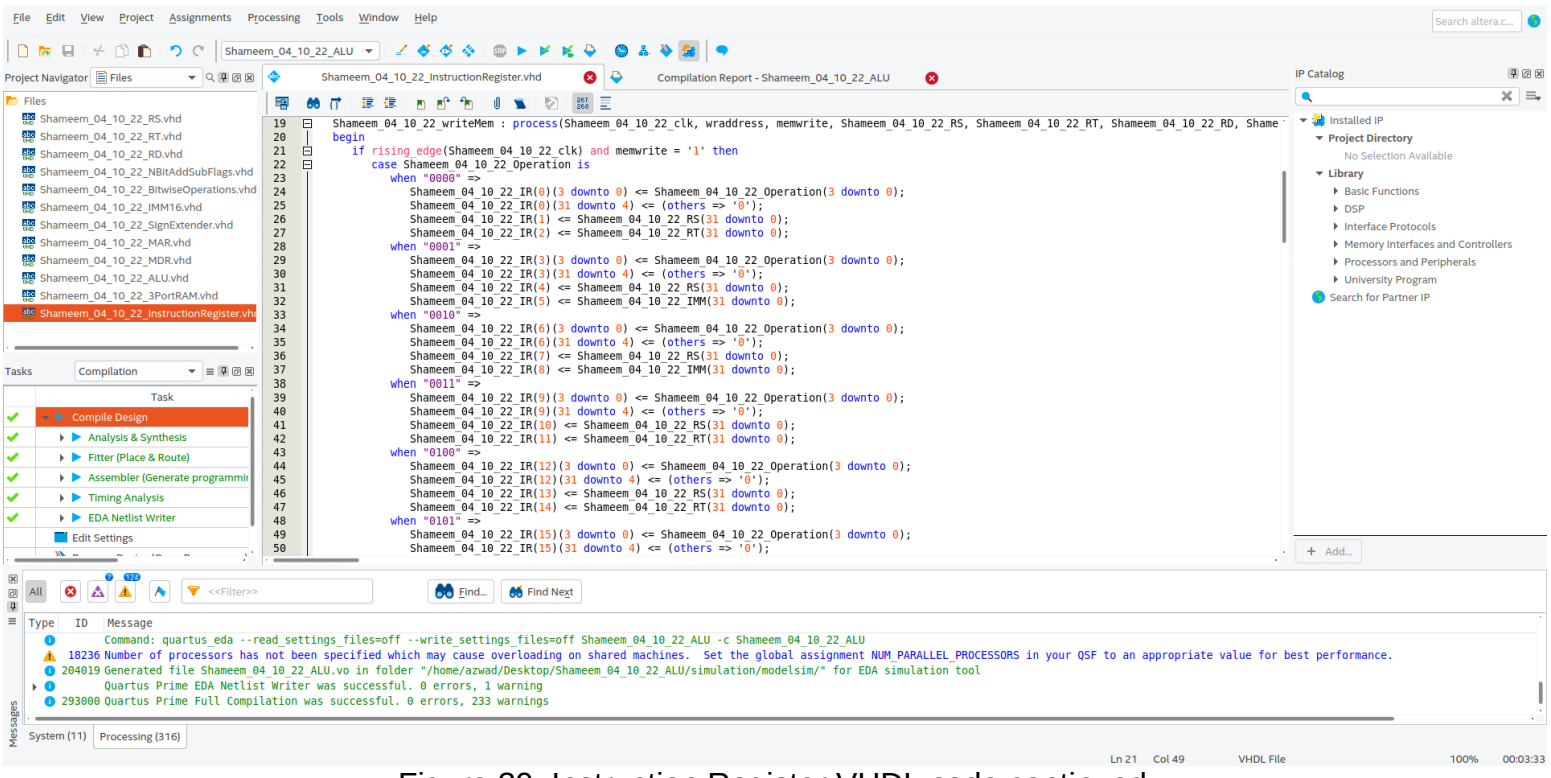
Instruction Register



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

entity Shameem_04_10_22_InstructionRegister is
    port (
          signal Shameem_04_10_22_clk, memwrite, rdaddress, wraddress: in STD_LOGIC;
          signal Shameem_04_10_22_Operation: in STD_LOGIC_VECTOR(3 downto 0);
          signal Shameem_04_10_22_RD, Shameem_04_10_22_RS, Shameem_04_10_22_RT: in STD_LOGIC_VECTOR(31 downto 0);
          signal Shameem_04_10_22_IMM: in STD_LOGIC_VECTOR(31 downto 0);
          signal Shameem_04_10_22_INSTRUCTION: out STD_LOGIC_VECTOR(67 downto 0)
         );
end entity Shameem_04_10_22_InstructionRegister;

architecture arch of Shameem_04_10_22_InstructionRegister is
type Shameem_04_10_22_memory is array(47 downto 0) of STD_LOGIC_VECTOR(31 downto 0);
signal Shameem_04_10_22_IR : Shameem_04_10_22_memory;
begin
   Shameem_04_10_22_writeMem : process(Shameem_04_10_22_clk, wraddress, memwrite, Shameem_04_10_22_RS, Shameem_04_10_22_RT, Shameem_04_10_22_RD, Shame
   begin
      if rising_edge(Shameem_04_10_22_clk) and memwrite = '1' then
         case Shameem_04_10_22_Operation is
            when "0000" =>
               Shameem_04_10_22_IR(0)(3 downto 0) <= Shameem_04_10_22_Operation(3 downto 0);
               Shameem_04_10_22_IR(0)(31 downto 4) <= (others => '0');
               Shameem_04_10_22_IR(1) <= Shameem_04_10_22_RS(31 downto 0);
               Shameem_04_10_22_IR(2) <= Shameem_04_10_22_RT(31 downto 0);
            when "0001" =>
               Shameem_04_10_22_IR(3)(3 downto 0) <= Shameem_04_10_22_Operation(3 downto 0);
               Shameem_04_10_22_IR(3)(31 downto 4) <= (others => '0');
               Shameem_04_10_22_IR(4) <= Shameem_04_10_22_RS(31 downto 0);
               Shameem_04_10_22_IR(5) <= Shameem_04_10_22_IMM(31 downto 0);
```

```
Type  ID   Message
      Command: quartus_eda --read_settings_files=off --write_settings_files=off Shameem_04_10_22_ALU -c Shameem_04_10_22_ALU
 18236 Number of processors has not been specified which may cause overloading on shared machines.  Set the global assignment NUM_PARALLEL_PROCESSORS in your QSF to an appropriate value for best performance.
204019 Generated file Shameem_04_10_22_ALU.vo in folder "/home/azwad/Desktop/Shameem_04_10_22_ALU/simulation/modelsim/" for EDA simulation tool
      Quartus Prime EDA Netlist Writer was successful. 0 errors, 1 warning
293000 Quartus Prime Full Compilation was successful. 0 errors, 233 warnings
```

Figure 38: Instruction Register VHDL code compiled successfully in Quartus.



```
   Shameem_04_10_22_writeMem : process(Shameem_04_10_22_clk, wraddress, memwrite, Shameem_04_10_22_RS, Shameem_04_10_22_RT, Shameem_04_10_22_RD, Shame
   begin
      if rising_edge(Shameem_04_10_22_clk) and memwrite = '1' then
         case Shameem_04_10_22_Operation is
            when "0000" =>
               Shameem_04_10_22_IR(0)(3 downto 0) <= Shameem_04_10_22_Operation(3 downto 0);
               Shameem_04_10_22_IR(0)(31 downto 4) <= (others => '0');
               Shameem_04_10_22_IR(1) <= Shameem_04_10_22_RS(31 downto 0);
               Shameem_04_10_22_IR(2) <= Shameem_04_10_22_RT(31 downto 0);
            when "0001" =>
               Shameem_04_10_22_IR(3)(3 downto 0) <= Shameem_04_10_22_Operation(3 downto 0);
               Shameem_04_10_22_IR(3)(31 downto 4) <= (others => '0');
               Shameem_04_10_22_IR(4) <= Shameem_04_10_22_RS(31 downto 0);
               Shameem_04_10_22_IR(5) <= Shameem_04_10_22_IMM(31 downto 0);
            when "0010" =>
               Shameem_04_10_22_IR(6)(3 downto 0) <= Shameem_04_10_22_Operation(3 downto 0);
               Shameem_04_10_22_IR(6)(31 downto 4) <= (others => '0');
               Shameem_04_10_22_IR(7) <= Shameem_04_10_22_RS(31 downto 0);
               Shameem_04_10_22_IR(8) <= Shameem_04_10_22_IMM(31 downto 0);
            when "0011" =>
               Shameem_04_10_22_IR(9)(3 downto 0) <= Shameem_04_10_22_Operation(3 downto 0);
               Shameem_04_10_22_IR(9)(31 downto 4) <= (others => '0');
               Shameem_04_10_22_IR(10) <= Shameem_04_10_22_RS(31 downto 0);
               Shameem_04_10_22_IR(11) <= Shameem_04_10_22_RT(31 downto 0);
            when "0100" =>
               Shameem_04_10_22_IR(12)(3 downto 0) <= Shameem_04_10_22_Operation(3 downto 0);
               Shameem_04_10_22_IR(12)(31 downto 4) <= (others => '0');
               Shameem_04_10_22_IR(13) <= Shameem_04_10_22_RS(31 downto 0);
               Shameem_04_10_22_IR(14) <= Shameem_04_10_22_RT(31 downto 0);
            when "0101" =>
               Shameem_04_10_22_IR(15)(3 downto 0) <= Shameem_04_10_22_Operation(3 downto 0);
               Shameem_04_10_22_IR(15)(31 downto 4) <= (others => '0');
```

```
Type  ID   Message
      Command: quartus_eda --read_settings_files=off --write_settings_files=off Shameem_04_10_22_ALU -c Shameem_04_10_22_ALU
 18236 Number of processors has not been specified which may cause overloading on shared machines.  Set the global assignment NUM_PARALLEL_PROCESSORS in your QSF to an appropriate value for best performance.
204019 Generated file Shameem_04_10_22_ALU.vo in folder "/home/azwad/Desktop/Shameem_04_10_22_ALU/simulation/modelsim/" for EDA simulation tool
      Quartus Prime EDA Netlist Writer was successful. 0 errors, 1 warning
293000 Quartus Prime Full Compilation was successful. 0 errors, 233 warnings
```

Figure 39: Instruction Register VHDL code continued.

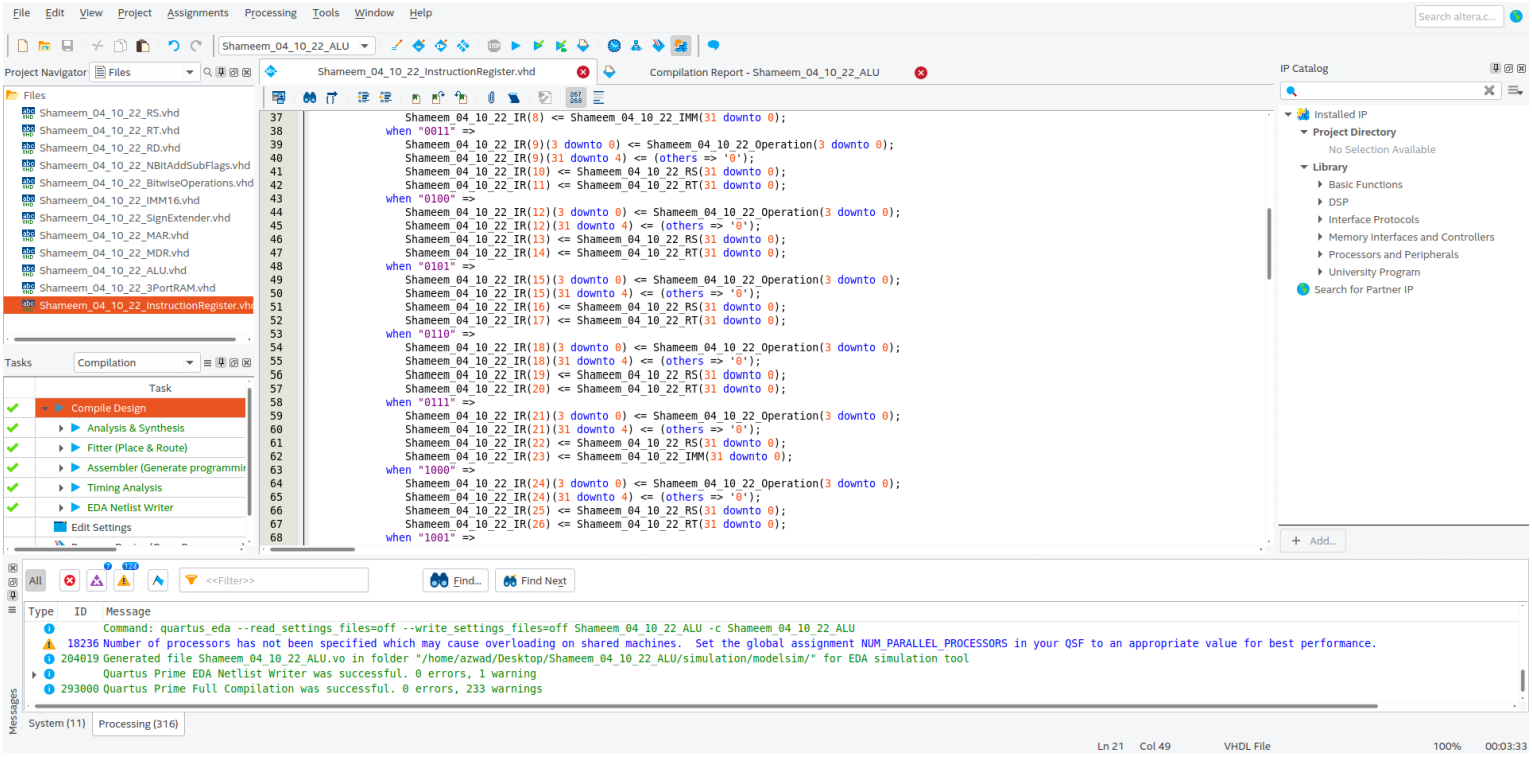


Figure 40: Instruction Register VHDL code continued.

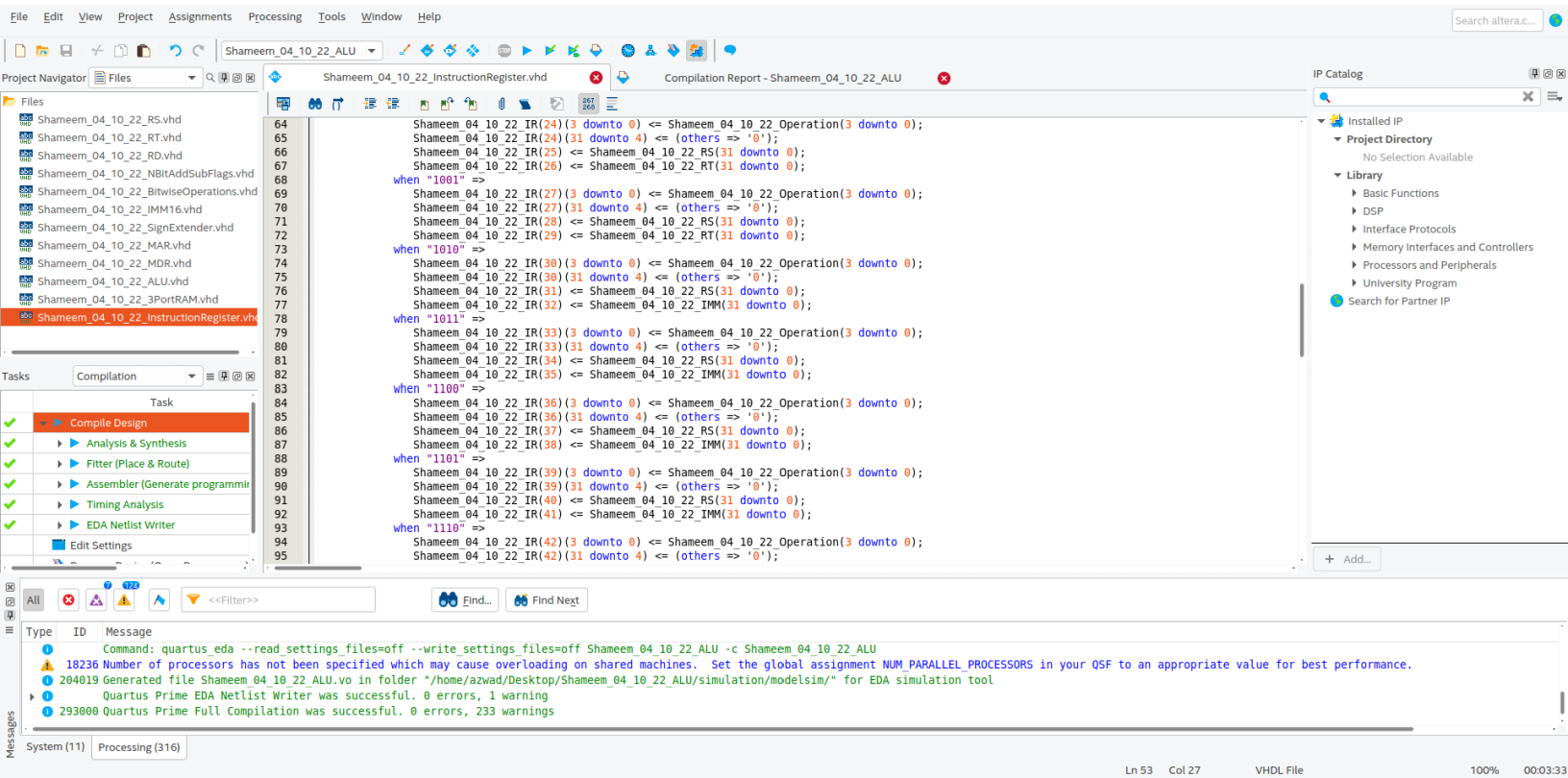
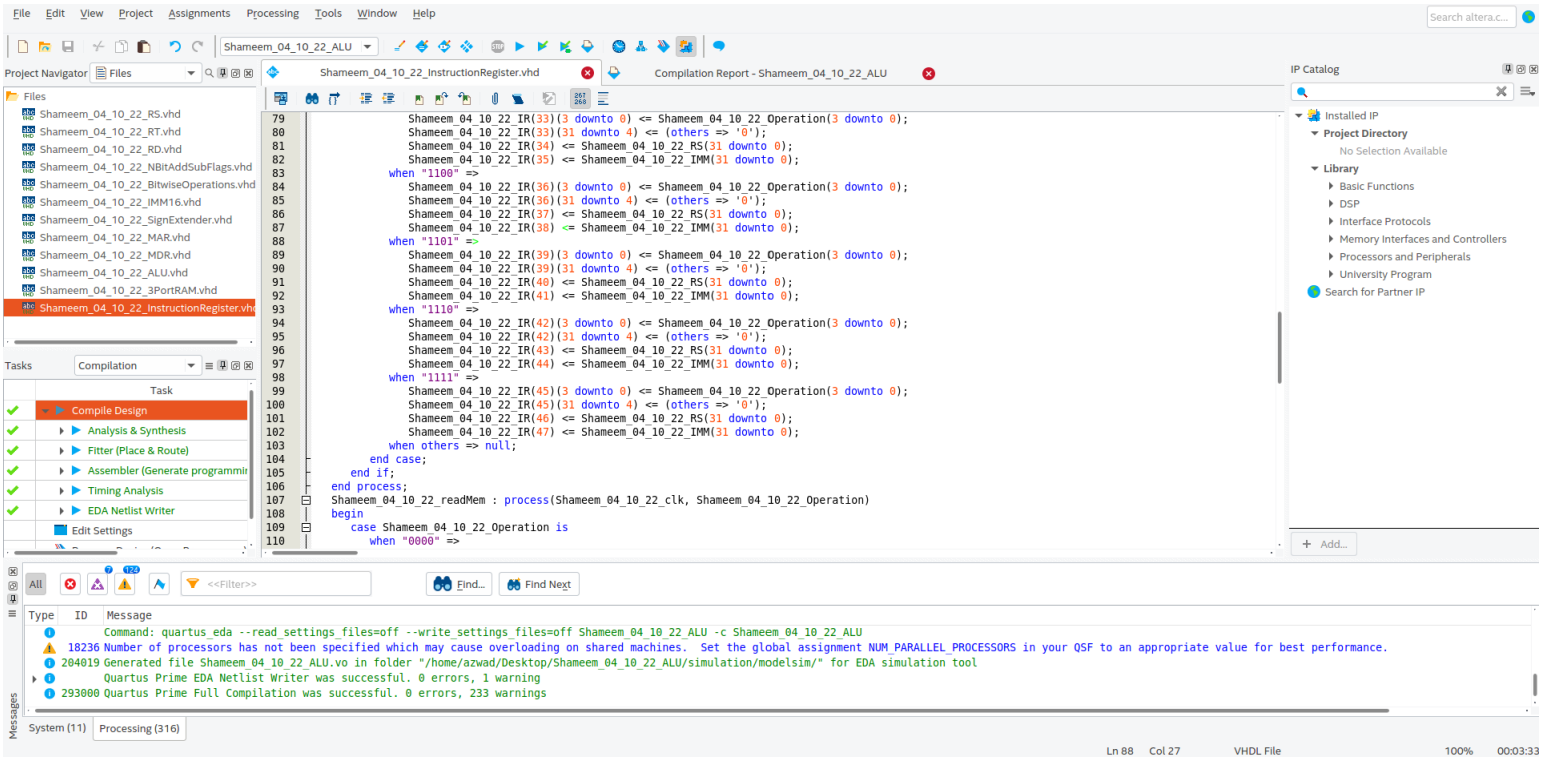
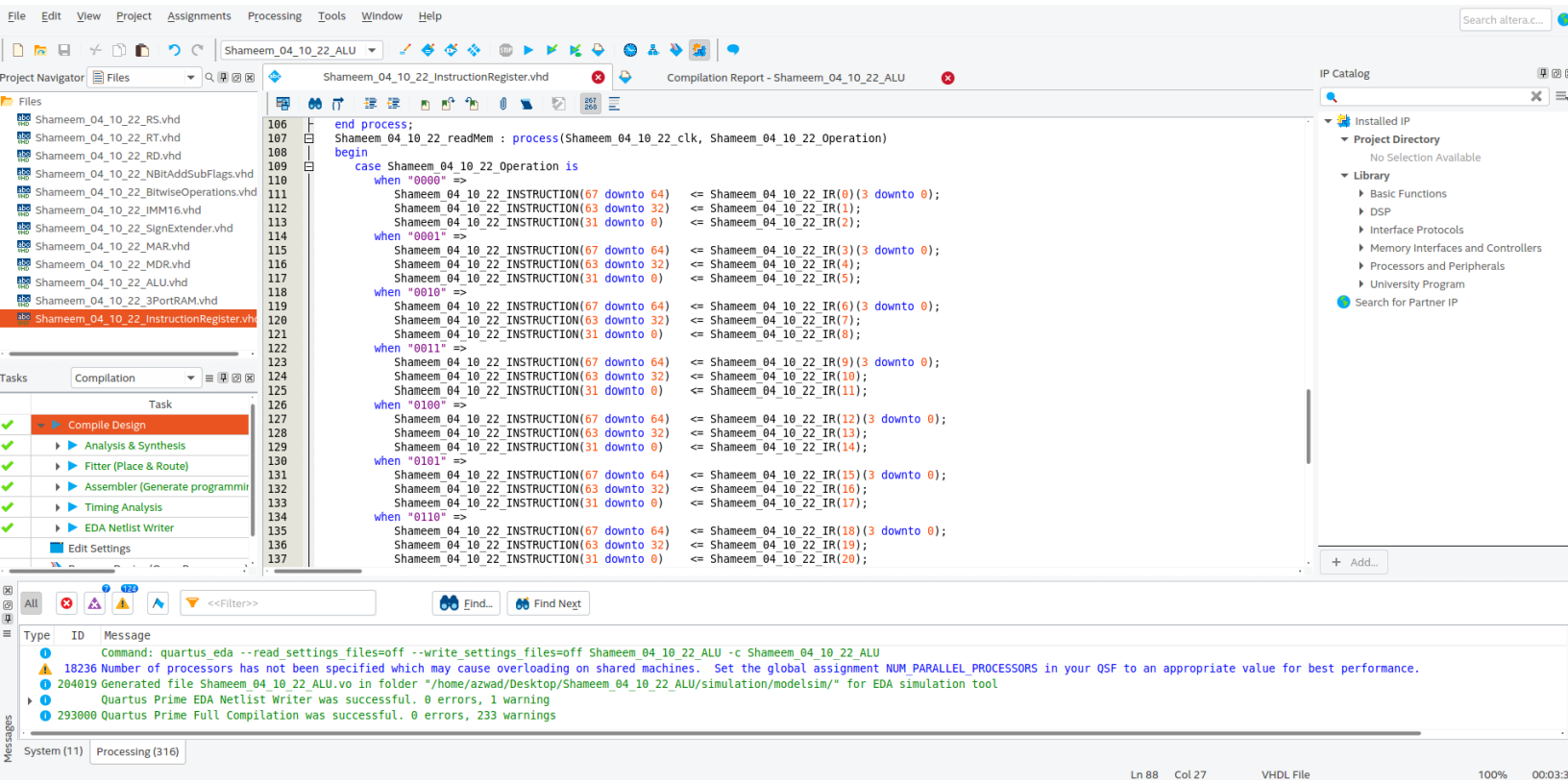


Figure 41: Instruction Register VHDL code continued.




```
                Shameem_04_10_22_IR(33)(3 downto 0) <= Shameem_04_10_22_Operation(3 downto 0);
                Shameem_04_10_22_IR(33)(31 downto 4) <= (others => '0');
                Shameem_04_10_22_IR(34) <= Shameem_04_10_22_RS(31 downto 0);
                Shameem_04_10_22_IR(35) <= Shameem_04_10_22_IMM(31 downto 0);
             when "1100" =>
                Shameem_04_10_22_IR(36)(3 downto 0) <= Shameem_04_10_22_Operation(3 downto 0);
                Shameem_04_10_22_IR(36)(31 downto 4) <= (others => '0');
                Shameem_04_10_22_IR(37) <= Shameem_04_10_22_RS(31 downto 0);
                Shameem_04_10_22_IR(38) <= Shameem_04_10_22_IMM(31 downto 0);
             when "1101" =>
                Shameem_04_10_22_IR(39)(3 downto 0) <= Shameem_04_10_22_Operation(3 downto 0);
                Shameem_04_10_22_IR(39)(31 downto 4) <= (others => '0');
                Shameem_04_10_22_IR(40) <= Shameem_04_10_22_RS(31 downto 0);
                Shameem_04_10_22_IR(41) <= Shameem_04_10_22_IMM(31 downto 0);
             when "1110" =>
                Shameem_04_10_22_IR(42)(3 downto 0) <= Shameem_04_10_22_Operation(3 downto 0);
                Shameem_04_10_22_IR(42)(31 downto 4) <= (others => '0');
                Shameem_04_10_22_IR(43) <= Shameem_04_10_22_RS(31 downto 0);
                Shameem_04_10_22_IR(44) <= Shameem_04_10_22_IMM(31 downto 0);
             when "1111" =>
                Shameem_04_10_22_IR(45)(3 downto 0) <= Shameem_04_10_22_Operation(3 downto 0);
                Shameem_04_10_22_IR(45)(31 downto 4) <= (others => '0');
                Shameem_04_10_22_IR(46) <= Shameem_04_10_22_RS(31 downto 0);
                Shameem_04_10_22_IR(47) <= Shameem_04_10_22_IMM(31 downto 0);
             when others => null;
          end case;
       end if;
    end process;
    Shameem_04_10_22_readMem : process(Shameem_04_10_22_clk, Shameem_04_10_22_Operation)
    begin
       case Shameem_04_10_22_Operation is
          when "0000" =>
```

Figure 42: Instruction Register VHDL code continued.




```
    end process;
    Shameem_04_10_22_readMem : process(Shameem_04_10_22_clk, Shameem_04_10_22_Operation)
    begin
       case Shameem_04_10_22_Operation is
          when "0000" =>
             Shameem_04_10_22_INSTRUCTION(67 downto 64)   <= Shameem_04_10_22_IR(0)(3 downto 0);
             Shameem_04_10_22_INSTRUCTION(63 downto 32)   <= Shameem_04_10_22_IR(1);
             Shameem_04_10_22_INSTRUCTION(31 downto 0)    <= Shameem_04_10_22_IR(2);
          when "0001" =>
             Shameem_04_10_22_INSTRUCTION(67 downto 64)   <= Shameem_04_10_22_IR(3)(3 downto 0);
             Shameem_04_10_22_INSTRUCTION(63 downto 32)   <= Shameem_04_10_22_IR(4);
             Shameem_04_10_22_INSTRUCTION(31 downto 0)    <= Shameem_04_10_22_IR(5);
          when "0010" =>
             Shameem_04_10_22_INSTRUCTION(67 downto 64)   <= Shameem_04_10_22_IR(6)(3 downto 0);
             Shameem_04_10_22_INSTRUCTION(63 downto 32)   <= Shameem_04_10_22_IR(7);
             Shameem_04_10_22_INSTRUCTION(31 downto 0)    <= Shameem_04_10_22_IR(8);
          when "0011" =>
             Shameem_04_10_22_INSTRUCTION(67 downto 64)   <= Shameem_04_10_22_IR(9)(3 downto 0);
             Shameem_04_10_22_INSTRUCTION(63 downto 32)   <= Shameem_04_10_22_IR(10);
             Shameem_04_10_22_INSTRUCTION(31 downto 0)    <= Shameem_04_10_22_IR(11);
          when "0100" =>
             Shameem_04_10_22_INSTRUCTION(67 downto 64)   <= Shameem_04_10_22_IR(12)(3 downto 0);
             Shameem_04_10_22_INSTRUCTION(63 downto 32)   <= Shameem_04_10_22_IR(13);
             Shameem_04_10_22_INSTRUCTION(31 downto 0)    <= Shameem_04_10_22_IR(14);
          when "0101" =>
             Shameem_04_10_22_INSTRUCTION(67 downto 64)   <= Shameem_04_10_22_IR(15)(3 downto 0);
             Shameem_04_10_22_INSTRUCTION(63 downto 32)   <= Shameem_04_10_22_IR(16);
             Shameem_04_10_22_INSTRUCTION(31 downto 0)    <= Shameem_04_10_22_IR(17);
          when "0110" =>
             Shameem_04_10_22_INSTRUCTION(67 downto 64)   <= Shameem_04_10_22_IR(18)(3 downto 0);
             Shameem_04_10_22_INSTRUCTION(63 downto 32)   <= Shameem_04_10_22_IR(19);
             Shameem_04_10_22_INSTRUCTION(31 downto 0)    <= Shameem_04_10_22_IR(20);
```

Figure 43: Instruction Register VHDL code continued.

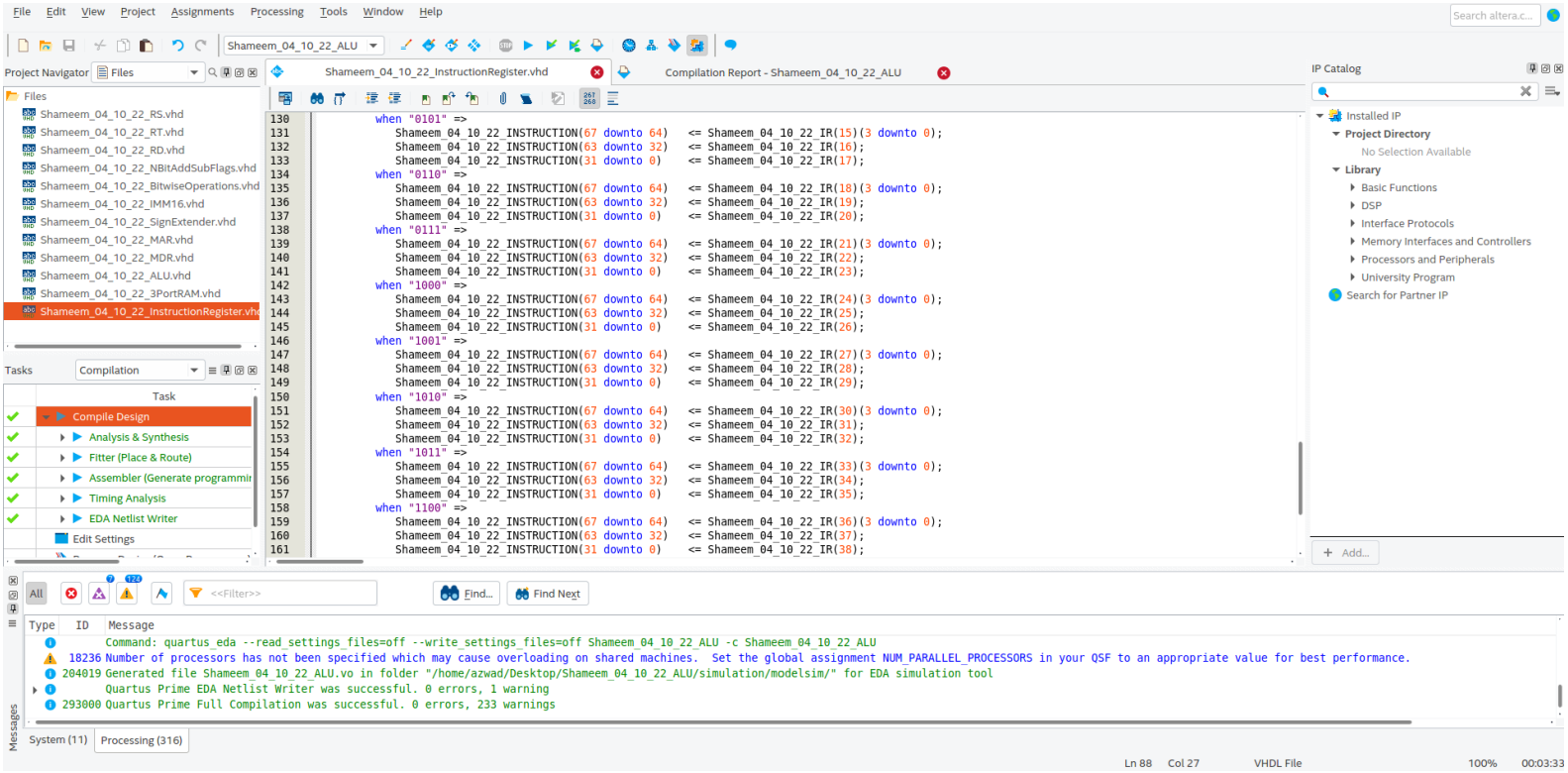


Figure 44: Instruction Register VHDL code continued.

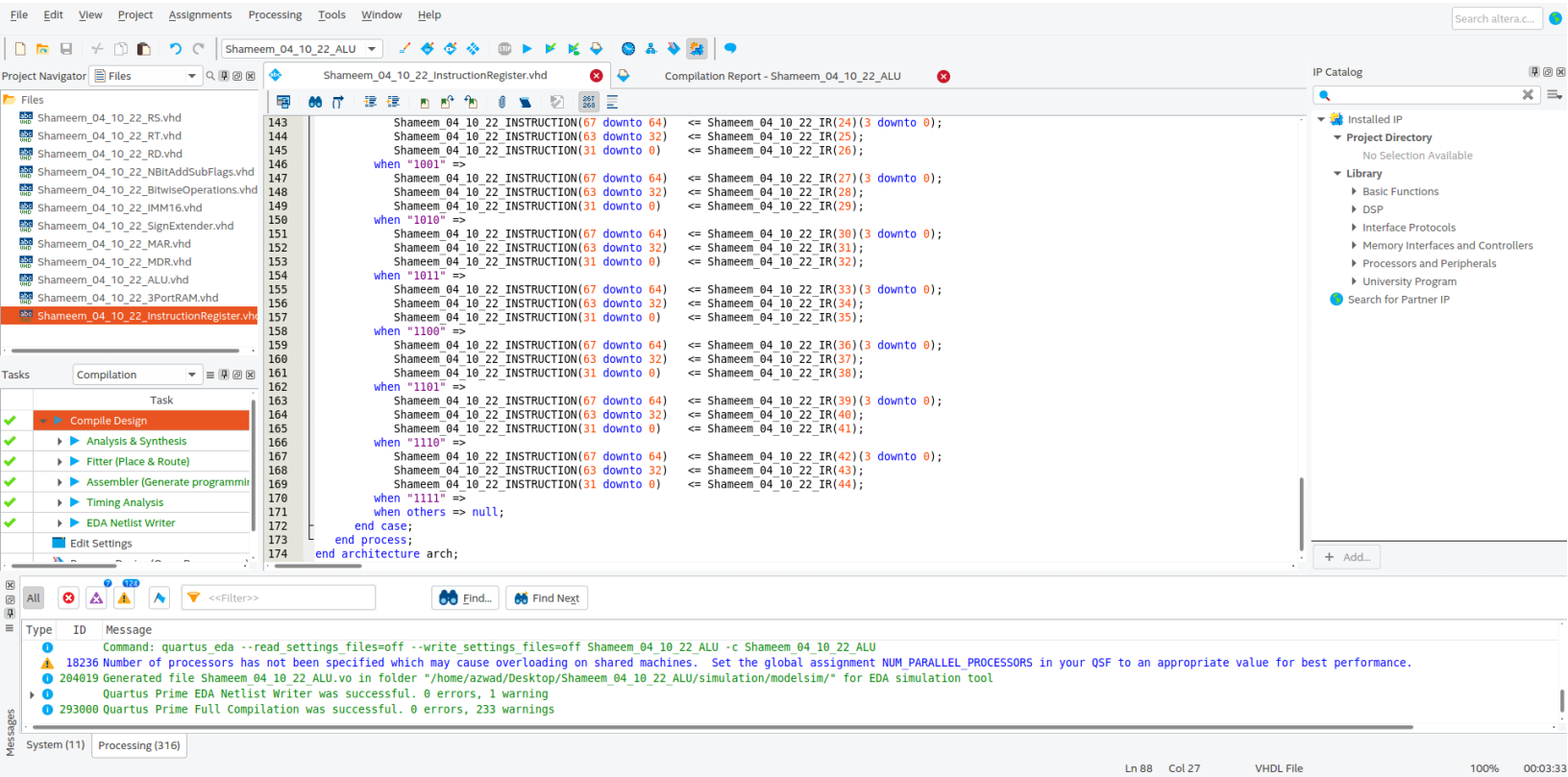


Figure 45: Instruction Register VHDL code continued.

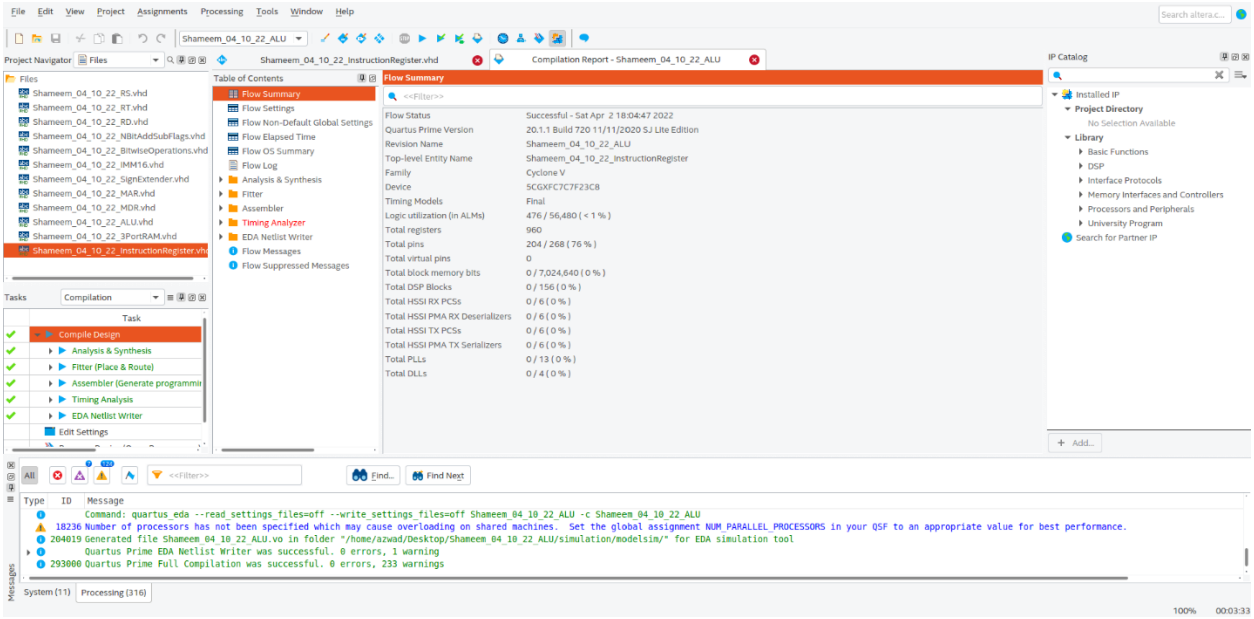


Figure 46: Instruction Register compilation report.

Data Memory

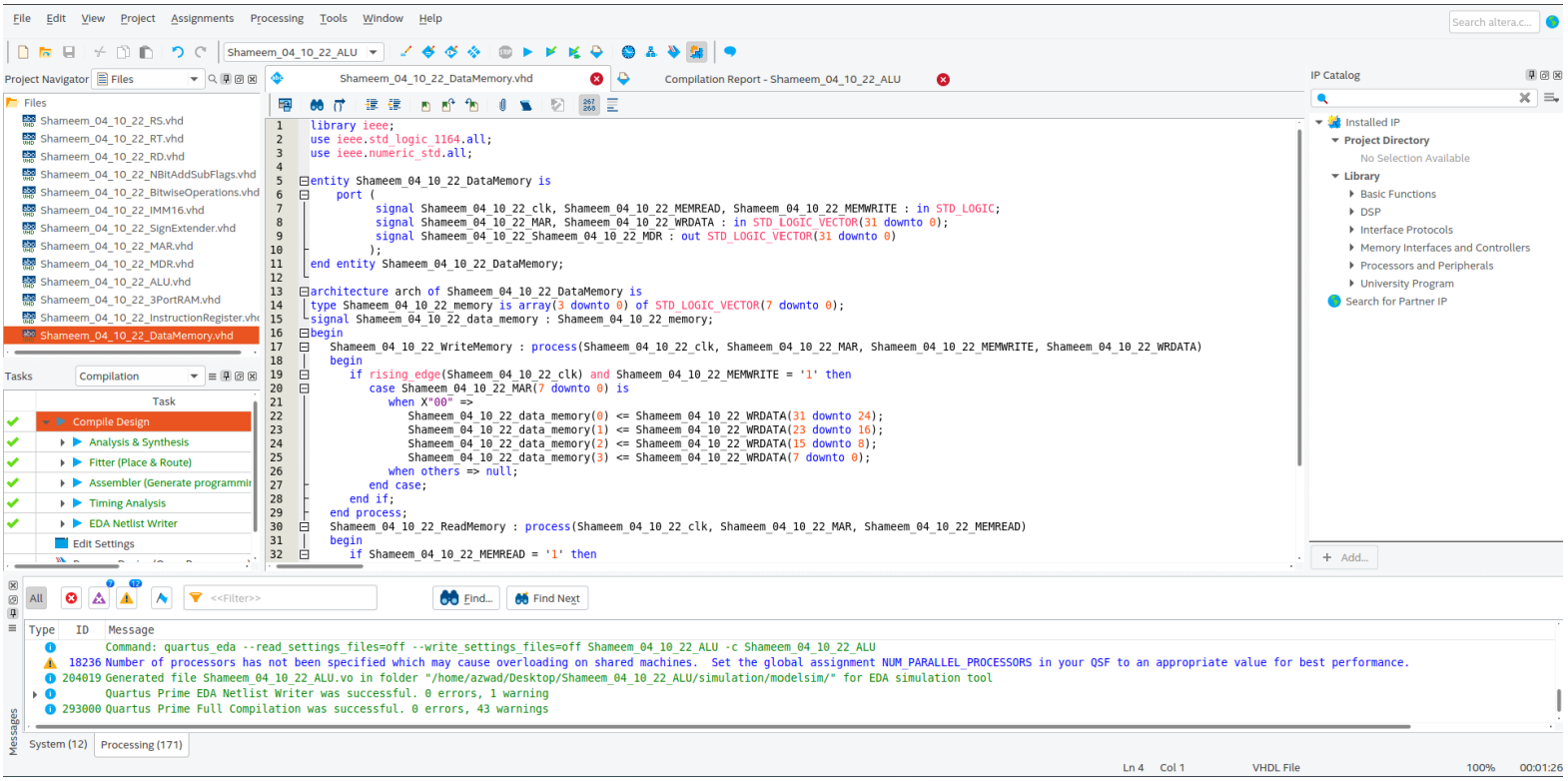


Figure 47: Data Memory VHDL code compiled successfully in Quartus

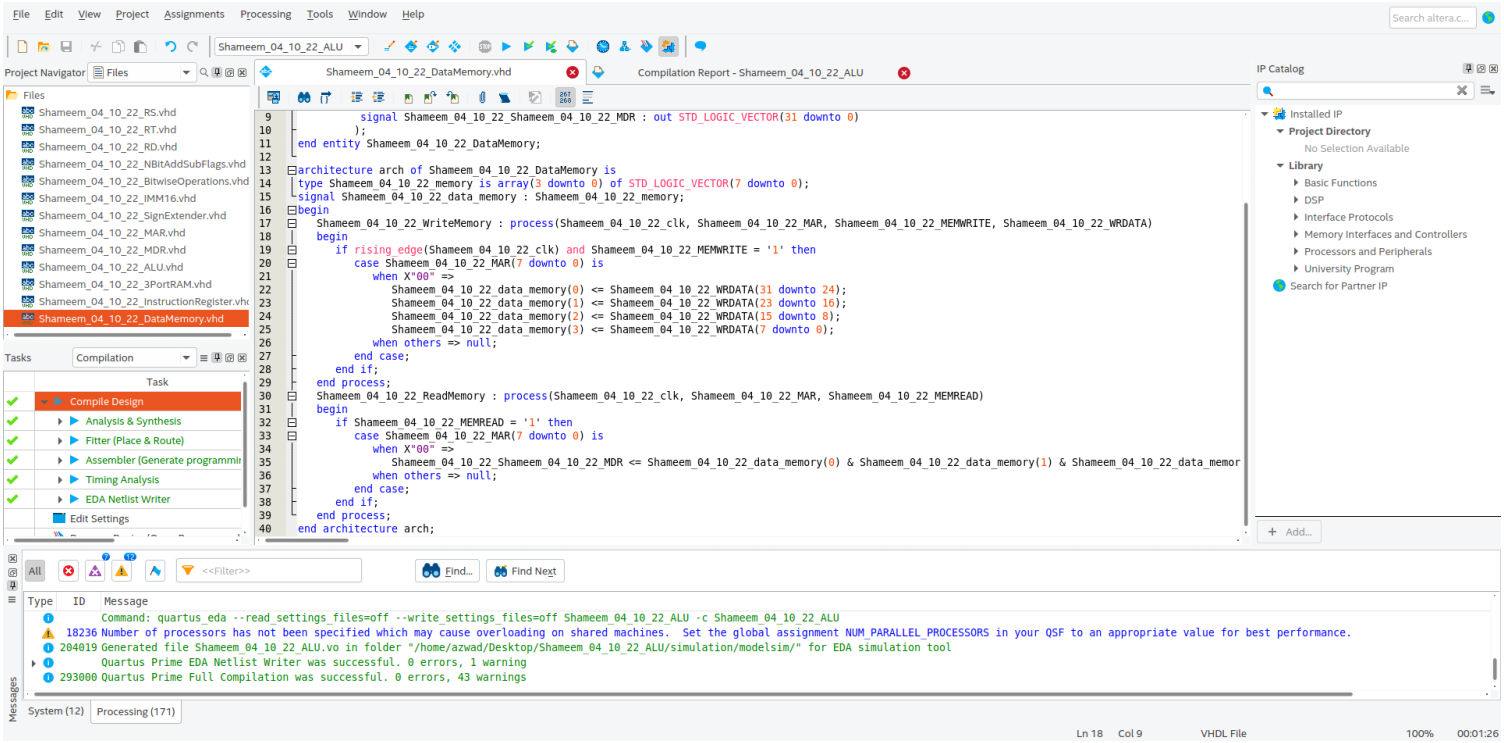


Figure 48: Data Memory VHDL code continued.

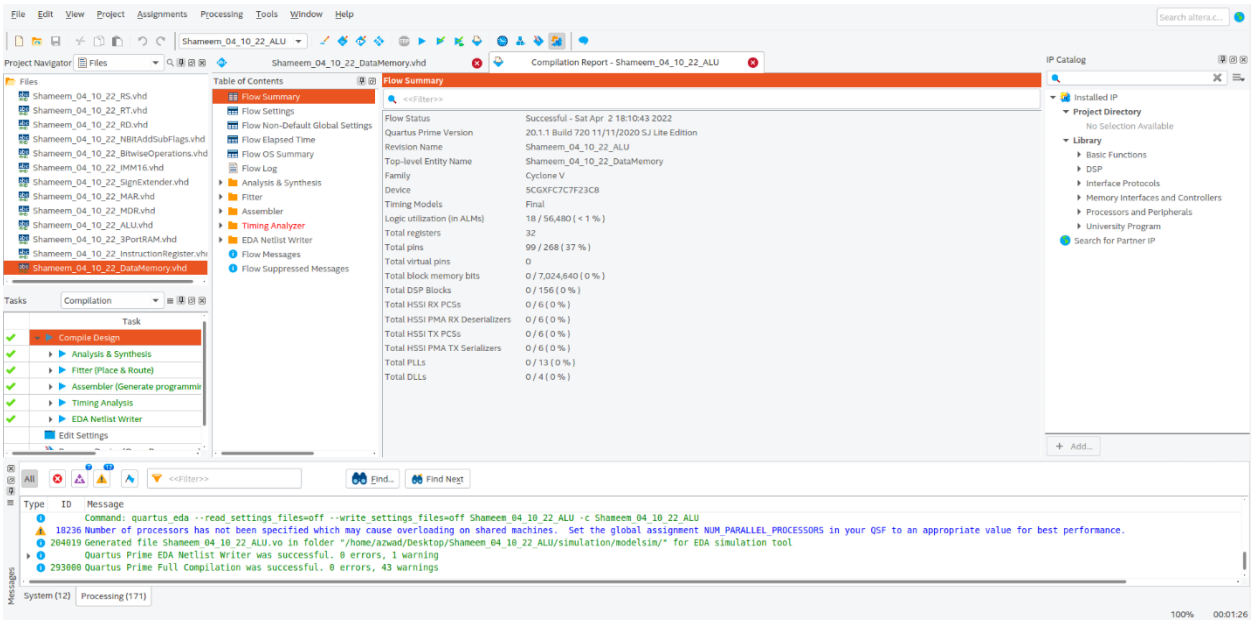


Figure 49: Data Memory compilation report.

Simulation:
Part I.A
Add

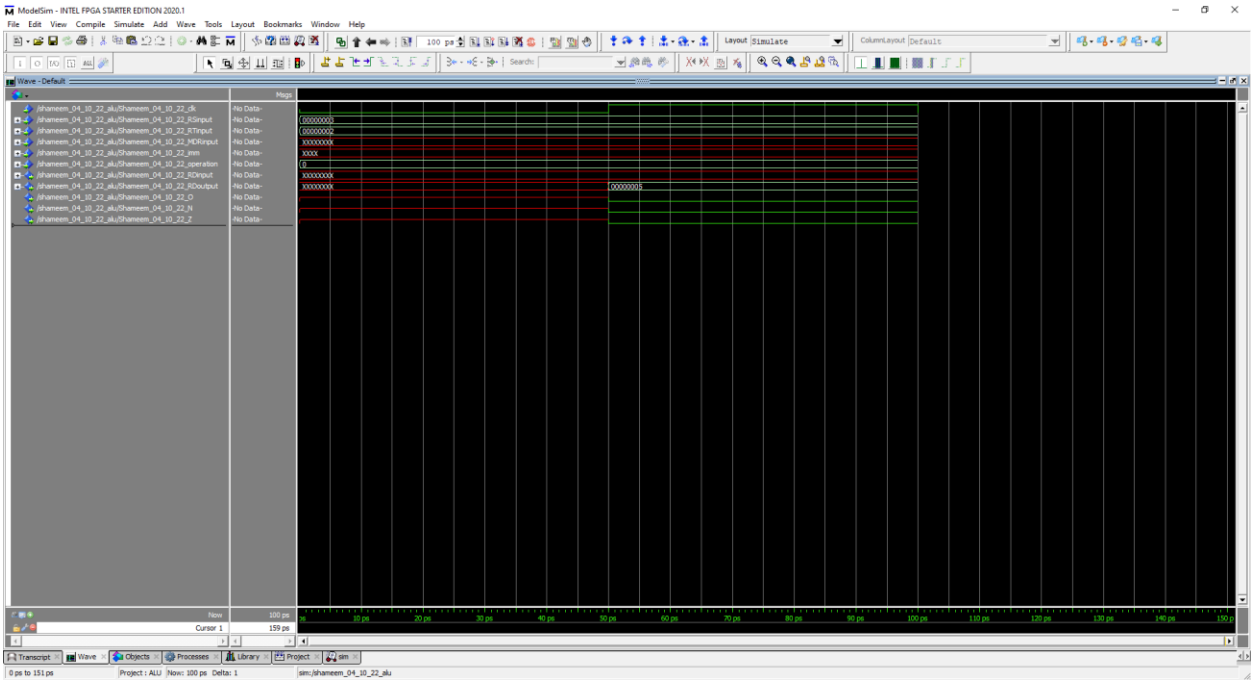


Figure 50: Simulation to represent ADD (radix is hexadecimal for everything)

In this simulation the operation is 0000 which stands for add so it takes RS and RT and adds them together and outputs them in the RD. Furthermore, 0x00000003 plus 0x00000002 equals 0x00000005, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because the output does not trigger the rules of overflow since a positive plus a positive can equal a positive without any overflow.

R [rd] (0x00000005) = R[rs] (0x00000003) + R[rt] (0x00000002)

Shameem_04_10_22_Add.asm

```
1   .data
2   rd: .word 0x00000000
3   rs: .word 0x00000003
4   rt: .word 0x00000002
5
6   .text
7   lw $s0, rd
8   lw $s1, rs
9   lw $s2, rt
10
11  # rd = rs + rt
12  add $s0, $s1, $s2
13  sw $s0 rd
14
```

Figure 51: MIPS code for Add

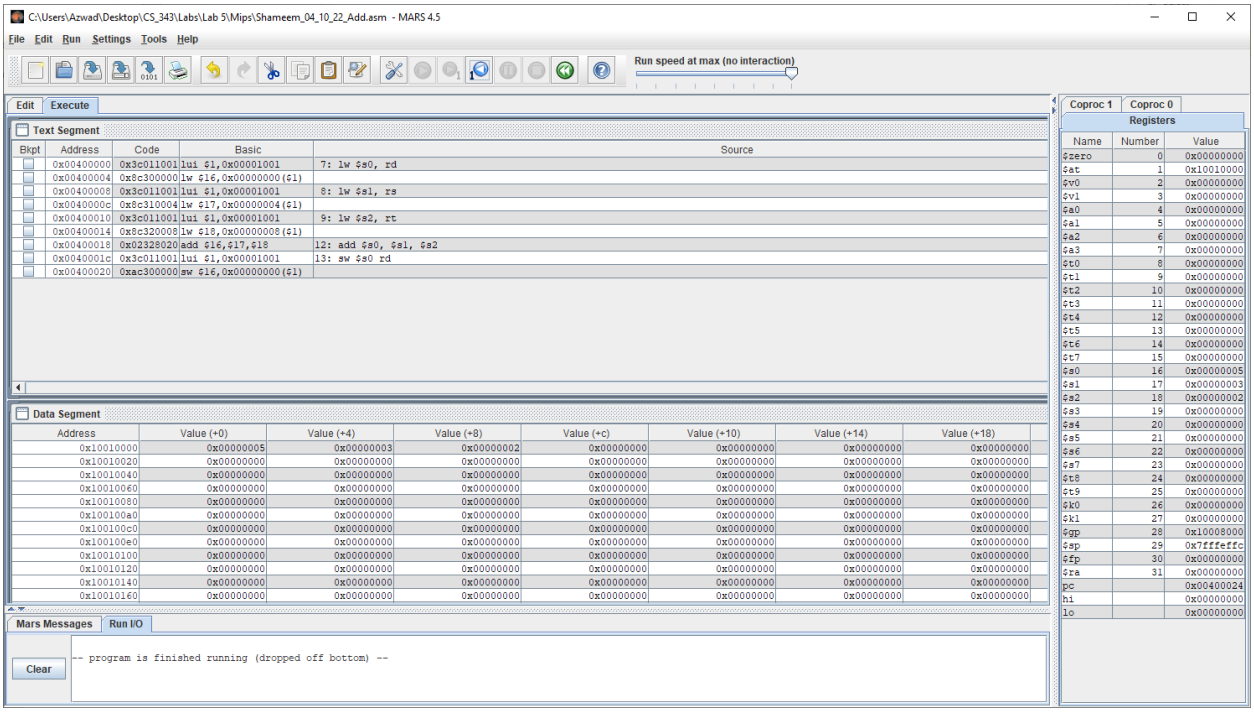


Figure 52: Executed Successfully.

Values RD, RS, RT shown clearly in data segment in the order of 0x00000005, 0x00000003, 0x00000002. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 50. Therefore, add is the same on MARS and ModelSim waveform, so the VHDL code for add is working correctly.

Addu

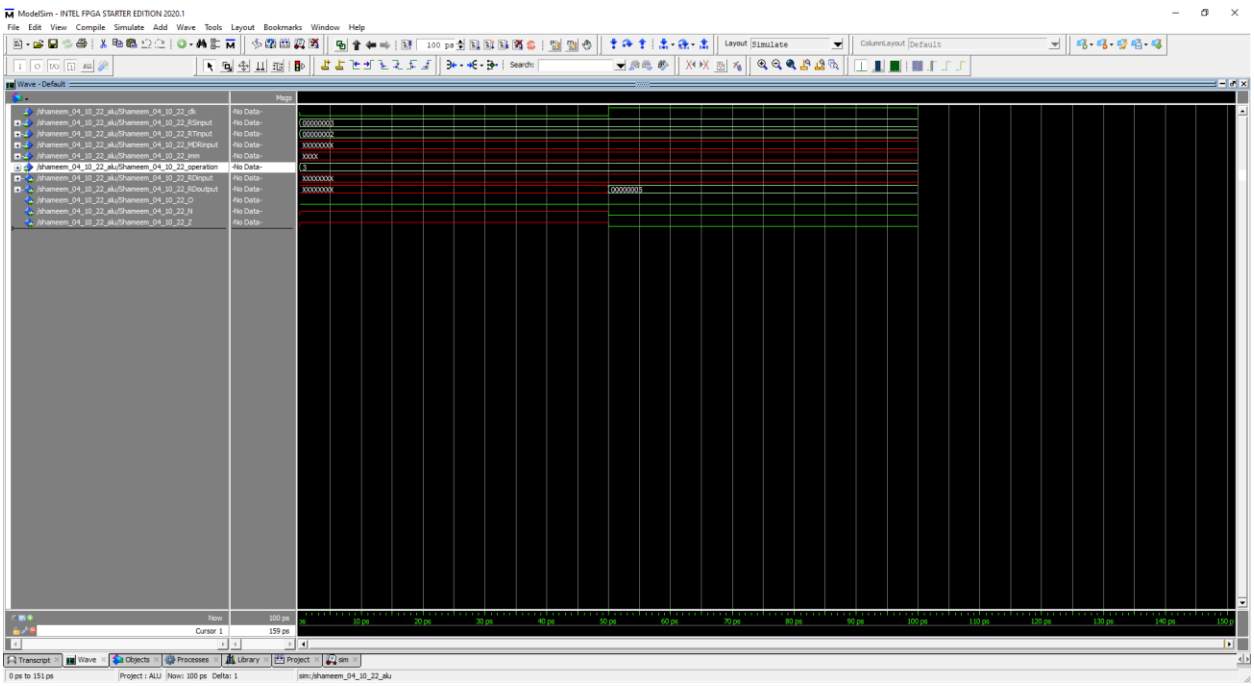


Figure 53: Simulation to represent Addu (radix is hexadecimal for everything)

In this simulation the operation is 0011, which stands for addu, so it takes RS and RT and adds them together and outputs them in the RD. Furthermore, since the operation is doing addu the overflow flag is set to ‘0’ and will not be triggered.

Edit | Execute

Shameem_04_10_22_Addu.asm

```
1   .data
2   rd: .word 0x00000000
3   rs: .word 0x00000003
4   rt: .word 0x00000002
5
6   .text
7   lw $s0, rd
8   lw $s1, rs
9   lw $s2, rt
10
11  # rd = rs + rt
12  addu $s0, $s1, $s2
13  sw $s0, rd
14
```

Figure 54: MIPS Addu

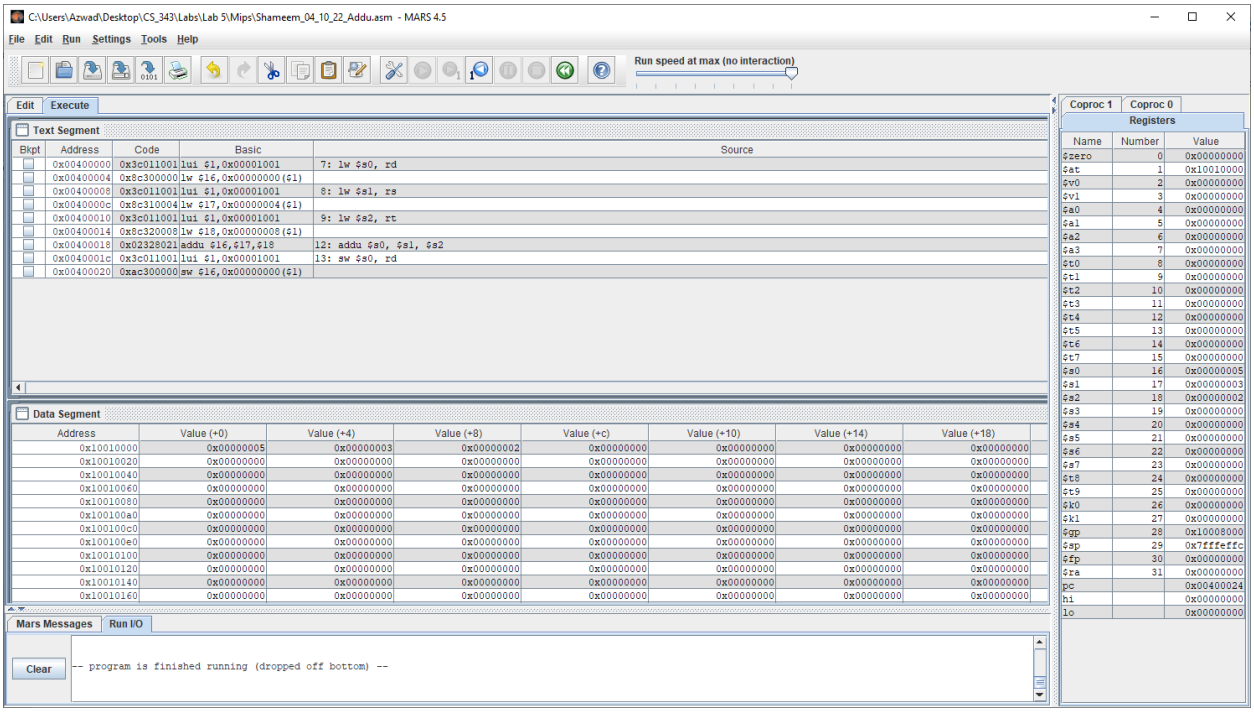


Figure 55: Executed Successfully in MARs.

Values RD, RS, RT from the equation RD = RS + RT shown clearly in data segment in the order of 0x00000005, 0x00000003, 0x00000002. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 53. Therefore, addu is the same on MARS and ModelSim waveform, so the VHDL code for addu is working correctly.

Sub

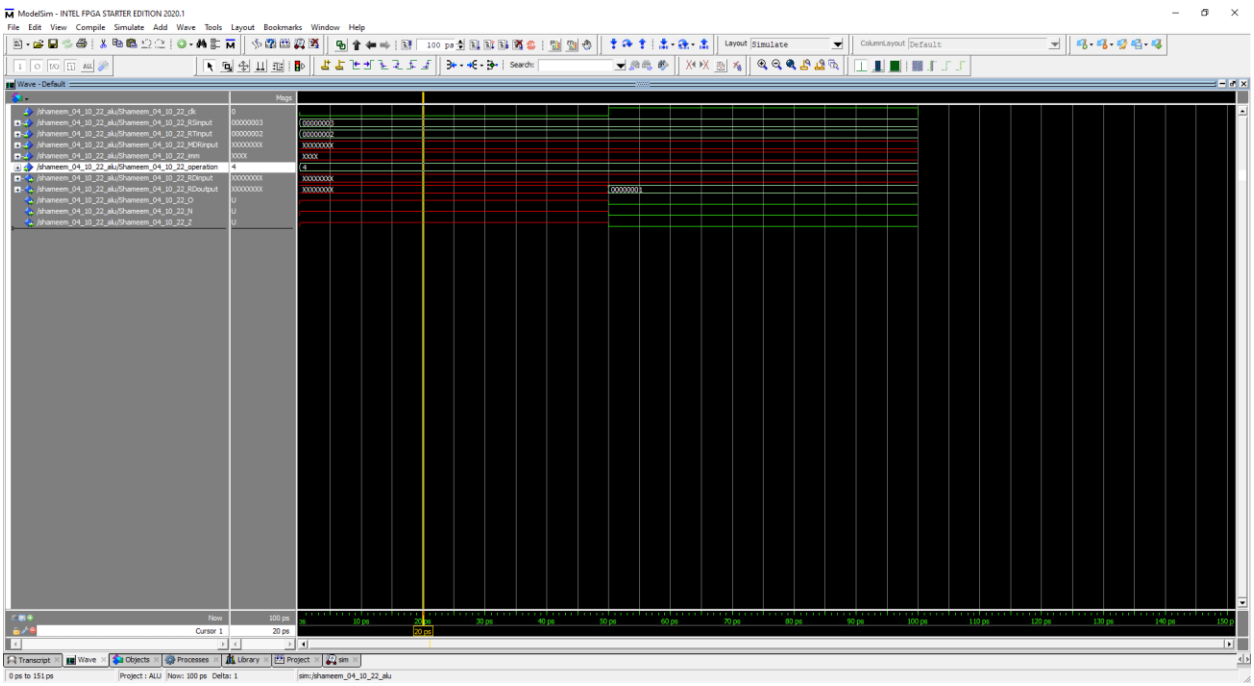


Figure 56: Simulation to represent Sub (radix is hexadecimal for everything)

In this simulation the operation is 0100 which stands for sub so it takes RS and RT and subtracts them and outputs them in the RD. Furthermore, 0x00000003 minus 0x00000002 equals 0x00000001, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because the output does not trigger the rules of overflow since a positive minus a positive can equal a positive without any overflow.

R [rd] (0x00000001) = R[rs] (0x00000003) - R[rt] (0x00000002)

Shameem_04_10_22_Sub.asm

```
1  .data
2  rd: .word 0x00000000
3  rs: .word 0x00000003
4  rt: .word 0x00000002
5
6  .text
7  lw $s0, rd
8  lw $s1, rs
9  lw $s2, rt
10
11 # rd = rs - rt
12 sub $s0, $s1, $s2
13 sw $s0, rd
14
```

Figure 57: MIPS sub

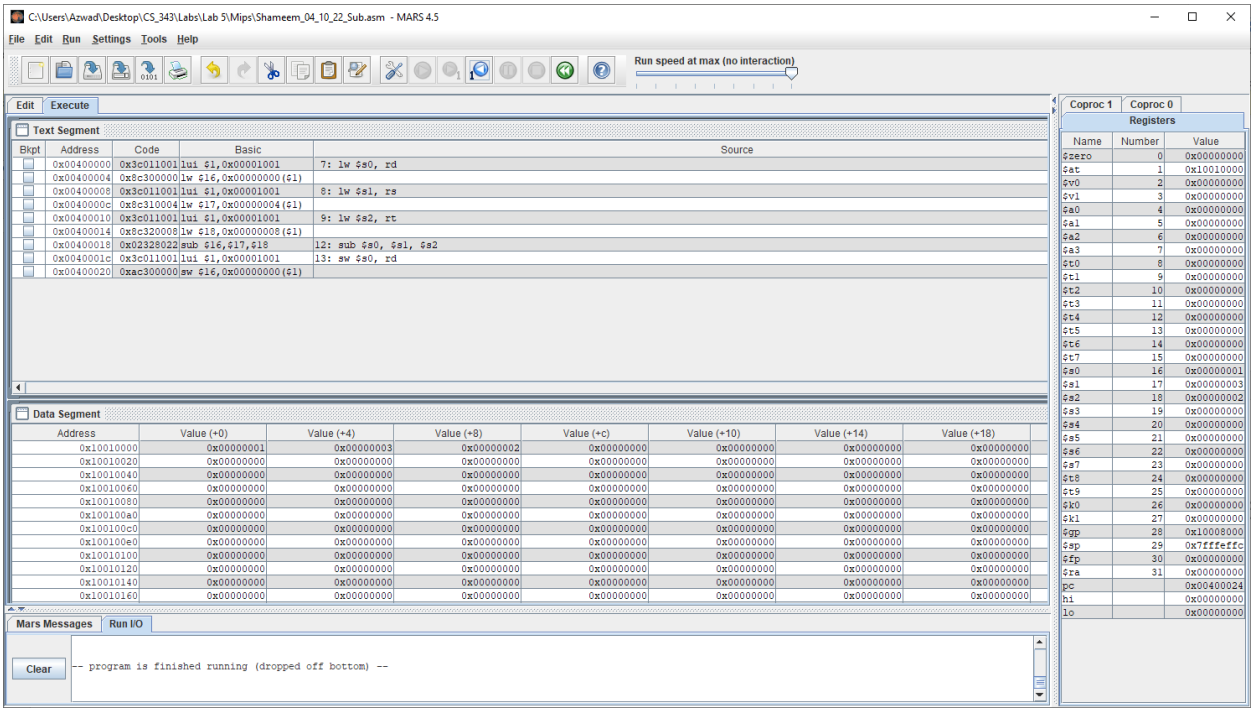


Figure 58: Executed Successfully on Mars.

Values RD, RS, RT shown clearly in data segment in the order of 0x00000001, 0x00000003, 0x00000002. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 50. Therefore, sub is the same on MARS and ModelSim waveform, so the VHDL code for sub is working correctly.

Subu

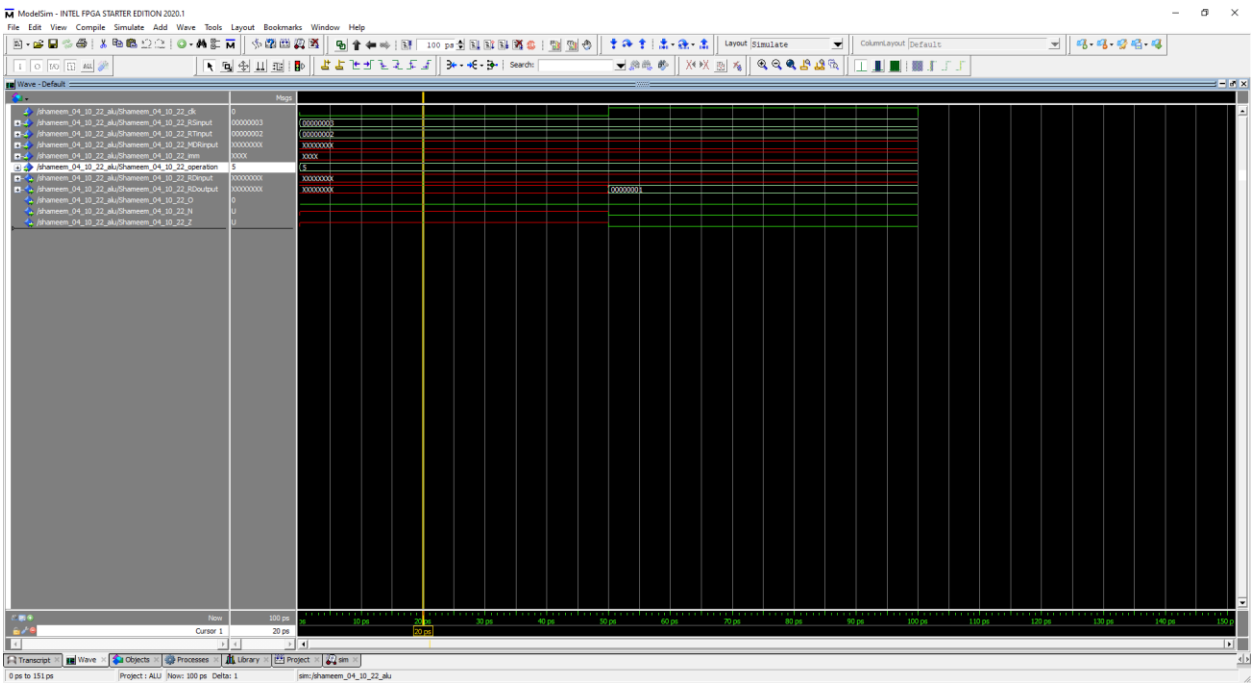


Figure 59: Simulation to represent Sub (radix is hexadecimal for everything)

In this simulation the operation is 0101 which stands for subu, so it takes RS and RT and subtracts them and outputs them in the RD. Furthermore, 0x00000003 minus 0x00000002 equals 0x00000001, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because the output does not trigger the rules of overflow since a positive minus a positive can equal a positive without any overflow.

R [rd] (0x00000001) = R[rs] (0x00000003) - R[rt] (0x00000002)

Shameem_04_10_22_Subu.asm

```
1   .data
2   rd: .word 0x00000000
3   rs: .word 0x00000003
4   rt: .word 0x00000002
5
6   .text
7   lw $s0, rd
8   lw $s1, rs
9   lw $s2, rt
10
11  # rd = rs - rt
12  subu $s0, $s1, $s2
13  sw $s0, rd
14
```

Figure 60: MIPS subu

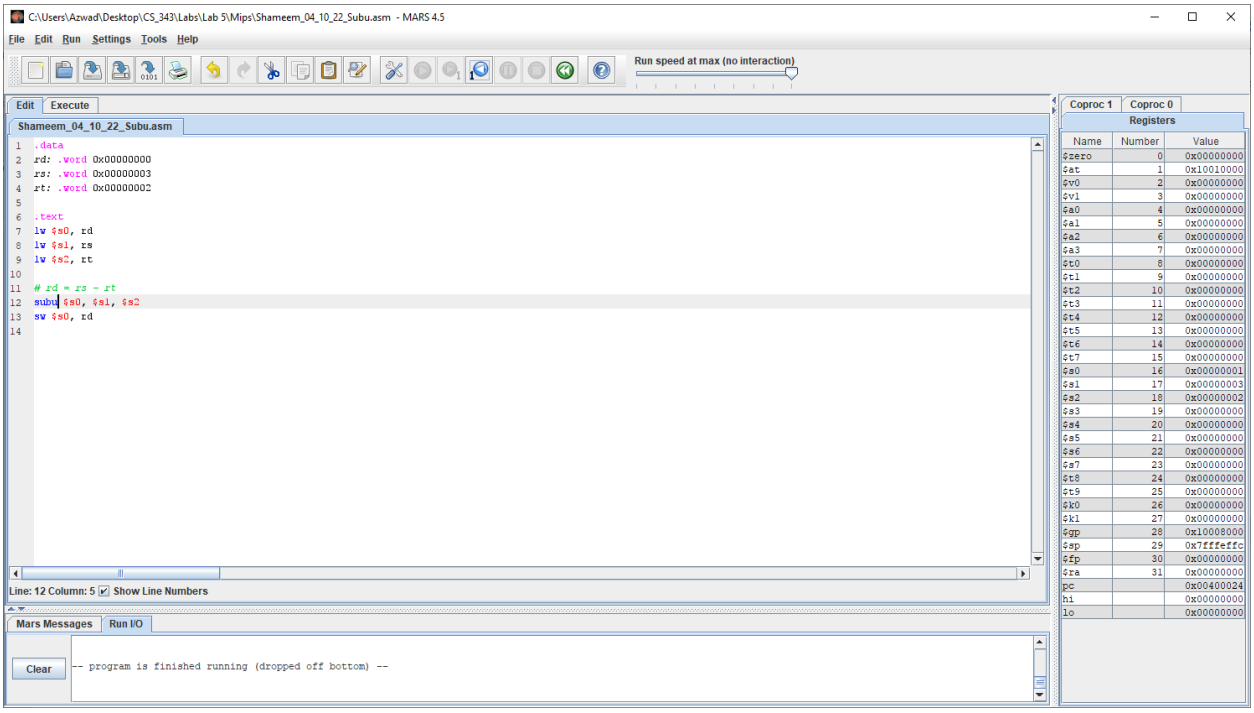


Figure 61: Executed Successfully on Mars.

Values RD, RS, RT shown clearly in data segment in the order of 0x00000001, 0x00000003, 0x00000002. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 59. Therefore, subu is the same on MARS and ModelSim waveform, so the VHDL code for subu is working correctly.

And

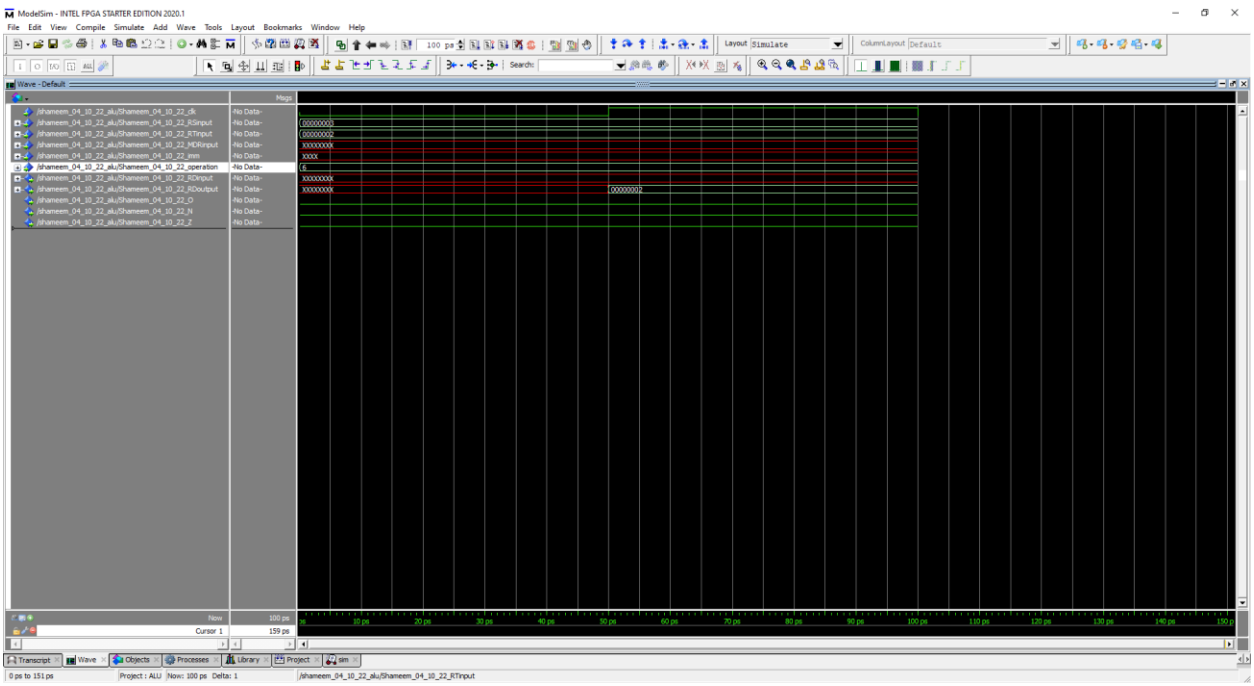


Figure 62: Simulation to represent AND (radix is hexadecimal for everything)

In this simulation the operation is 0110 which stands for add so it takes RS and RT and ands them, which outputs them in the RD. Furthermore, 0x00000003 and 0x00000002 equals 0x00000002, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because and does not trigger an overflow.

R [rd] (0x00000002) = R[rs] (0x00000003) and R[rt] (0x00000002)

Shameem_04_10_22_And.asm

```
1  .data
2  rd: .word 0x00000000
3  rs: .word 0x00000003
4  rt: .word 0x00000002
5
6  .text
7  lw $s0, rd
8  lw $s1, rs
9  lw $s2, rt
10
11 # rd = rs & rt
12 and $s0, $s1, $s2
13 sw $s0, rd
14
```

Figure 63: MIPS and

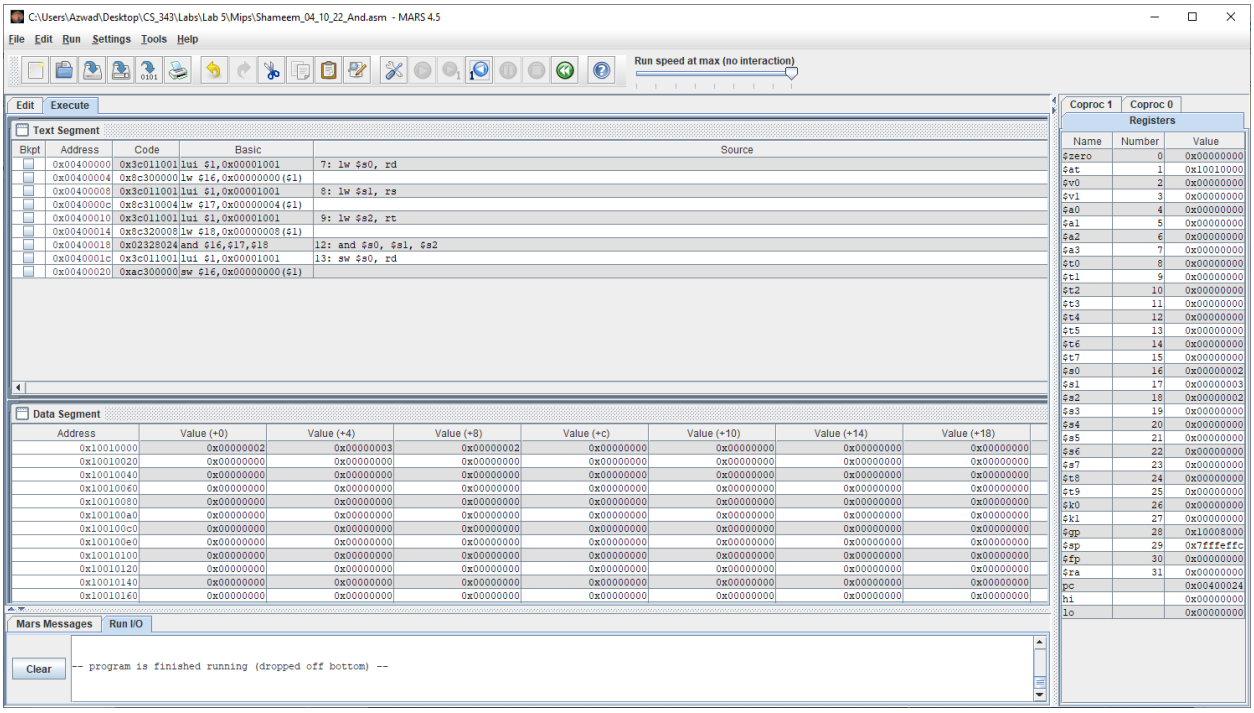


Figure 64: Successfully executed in MARs.

Values RD, RS, RT shown clearly in data segment in the order of 0x00000001, 0x00000003, 0x00000002. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 62. Therefore, and is the same on MARS and ModelSim waveform, so the VHDL code for and is working correctly.

Nor

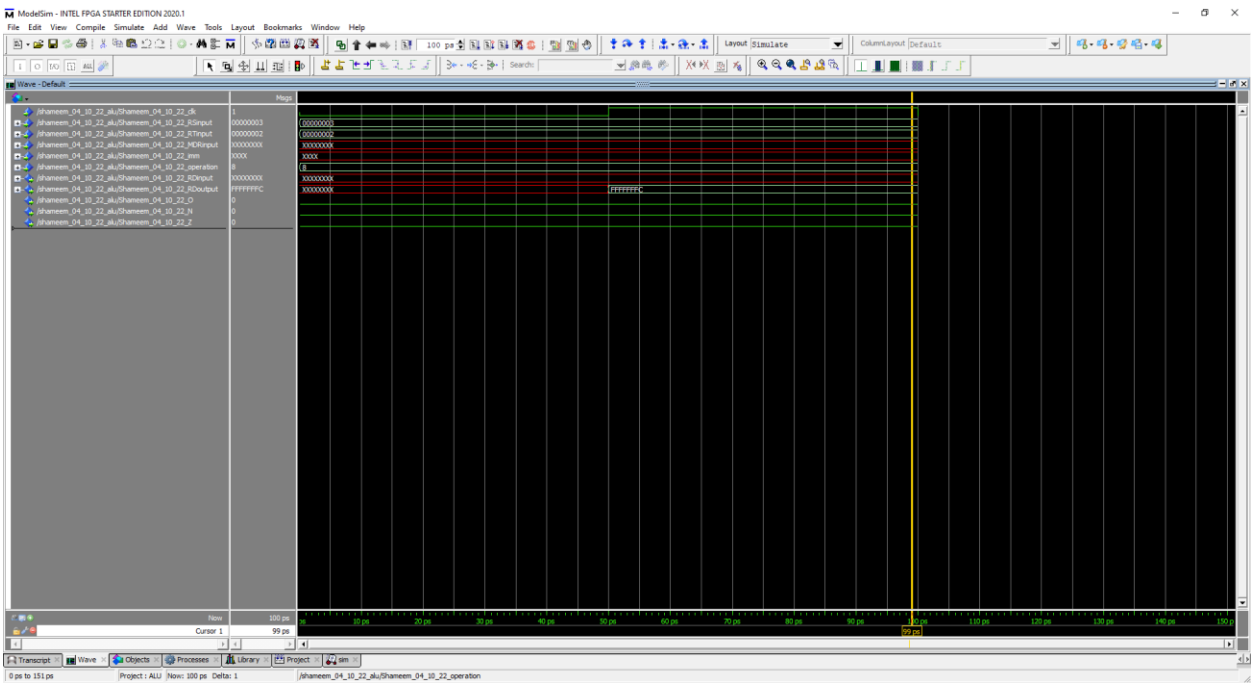


Figure 65: Simulation to represent AND (radix is hexadecimal for everything)

In this simulation the operation is 1000 which stands for add so it takes RS and RT and ands them, which outputs them in the RD. Furthermore, 0x00000003 and 0x00000002 equals 0xFFFFFFFC, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because nor does not trigger an overflow.

R [rd] (0xFFFFFFFC) = R[rs] (0x00000003) nor R[rt] (0x00000002)

Edit | Execute

Shameem_04_10_22_Nor.asm

```
1   .data
2   rd: .word 0x00000000
3   rs: .word 0x00000003
4   rt: .word 0x00000002
5
6
7   .text
8   lw $s0, rd
9   lw $s1, rs
10  lw $s2, rt
11
12  # rd = ~ (rs | rt)
13  nor $s0, $s1, $s2
14  sw $s0, rd
15
```

Figure 66: MIPS nor

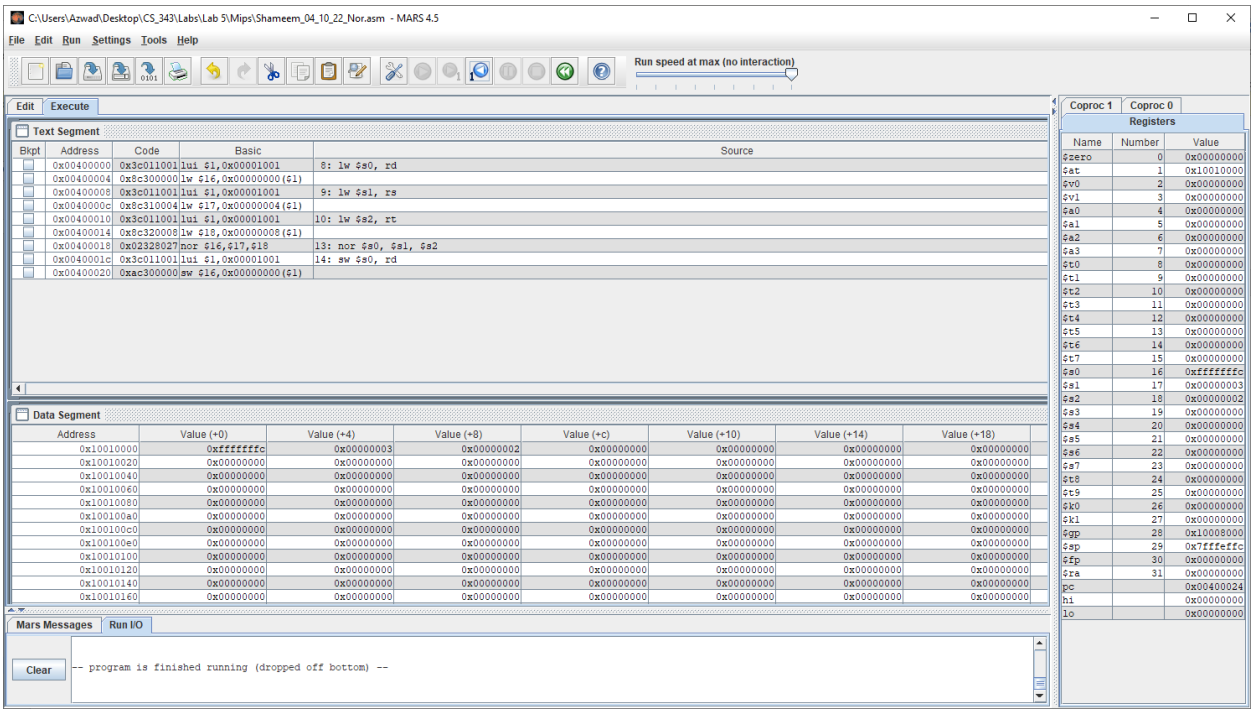


Figure 67: Successfully executed in MARs.

Values RD, RS, RT shown clearly in data segment in the order of 0xFFFFFFFC, 0x00000003, 0x00000002. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 65. Therefore, nor is the same on MARS and ModelSim waveform, so the VHDL code for nor is working correctly.

OR

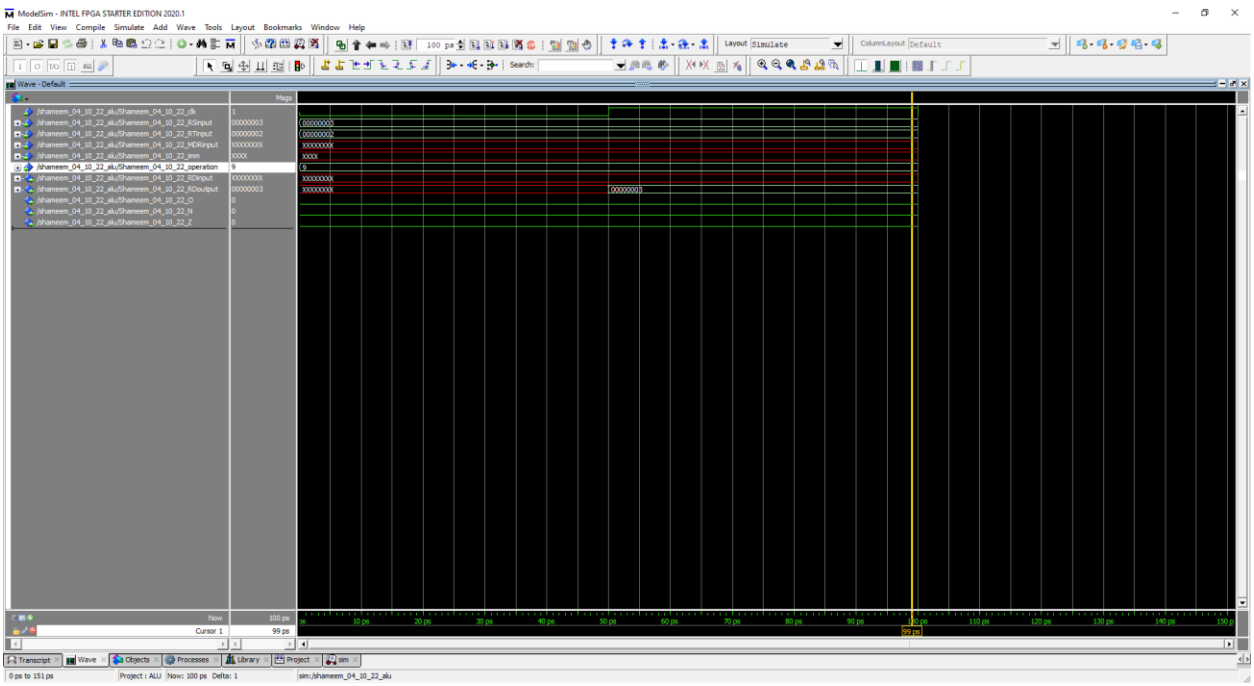


Figure 68: Simulation to represent AND (radix is hexadecimal for everything)

In this simulation the operation is 1001 which stands for add so it takes RS and RT and ands them, which outputs them in the RD. Furthermore, 0x00000003 and 0x00000002 equals 0x00000003, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because or does not trigger an overflow.

R [rd] (0x00000003) = R[rs] (0x00000003) nor R[rt] (0x00000002)

Shameem_04_10_22_Or.asm

```
1   .data
2   rd: .word 0x00000000
3   rs: .word 0x00000003
4   rt: .word 0x00000002
5
6   .text
7   lw $s0, rd
8   lw $s1, rs
9   lw $s2, rt
10
11  # rd = (rs | rt)
12  or $s0, $s1, $s2
13  sw $s0, rd
14
```

Figure 69: MIPS or

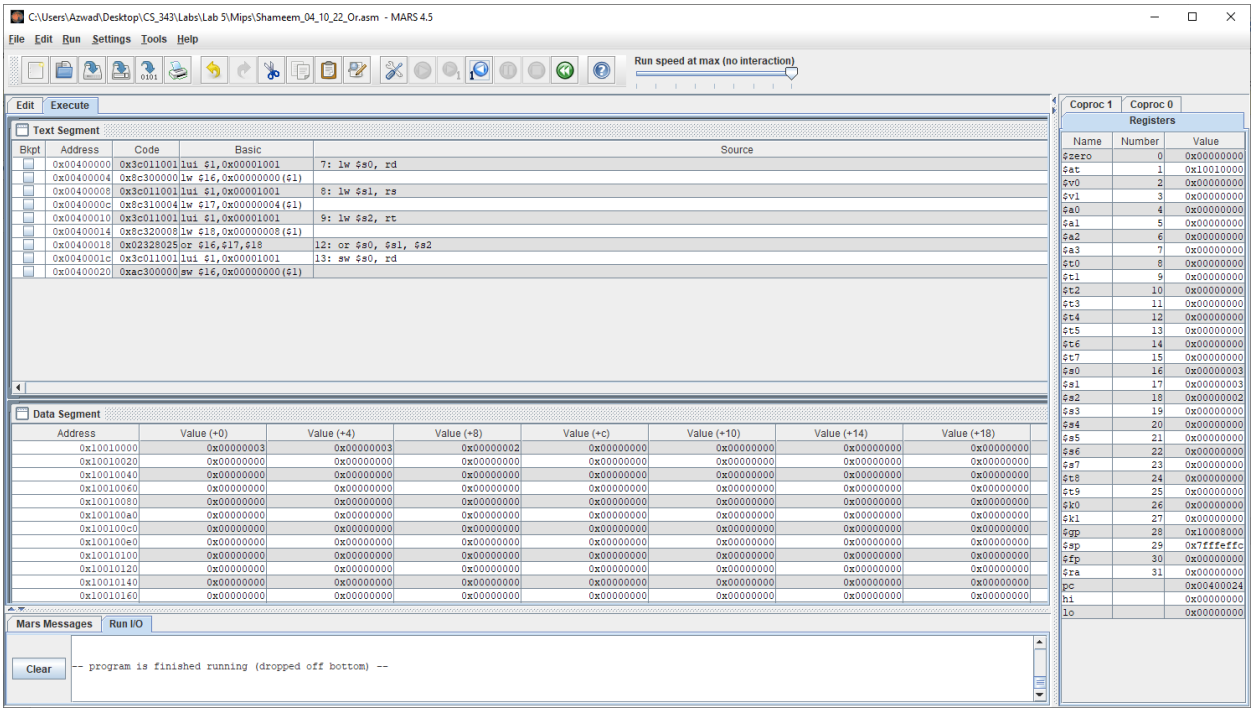


Figure 70: Executed successfully on MARs.

Values RD, RS, RT shown clearly in data segment in the order of 0x00000003, 0x00000003, 0x00000002. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 62. Therefore, OR is the same on MARS and ModelSim waveform, so the VHDL code for OR is working correctly.

Part I.B
Shift Left Logical – SLL

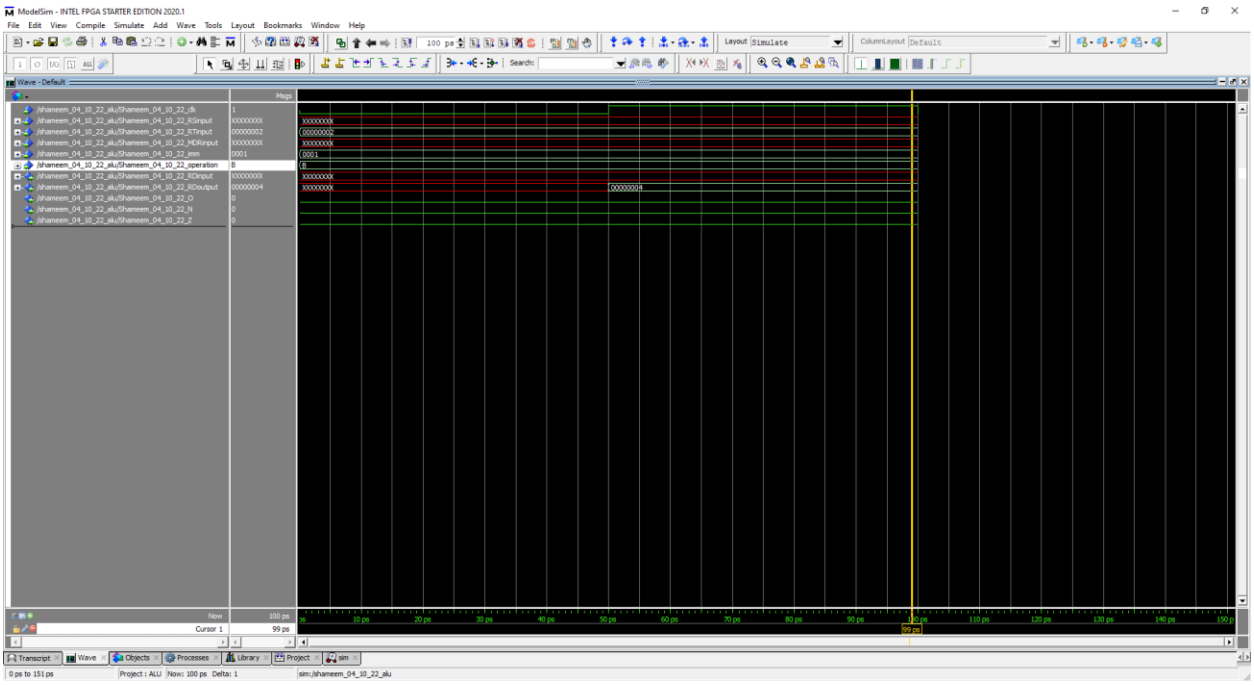


Figure 71: Simulation to represent SLL (radix is hexadecimal for everything)

R [rd] (0x00000004) = R[rt] (0x00000002) << shamt

In this simulation the operation is 1011 which stands for add so it takes RT and shifts it left by shamt, which then outputs in the RD. Furthermore, 0x00000002 shifted by shamt equals 0x00000004, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because shifting does not trigger an overflow.

Edit | Execute

Shameem_04_10_22_SLL.asm

```
1   .data
2   rd: .word 0x00000002
3   rt: .word 0x00000002
4
5   .text
6   lw $s0, rd
7   lw $s1, rt
8
9   # rd = rt << shamt
10  sll $s0, $s1, 1
11  sw $s0, rd
12
```

Figure 72: MIPS sll

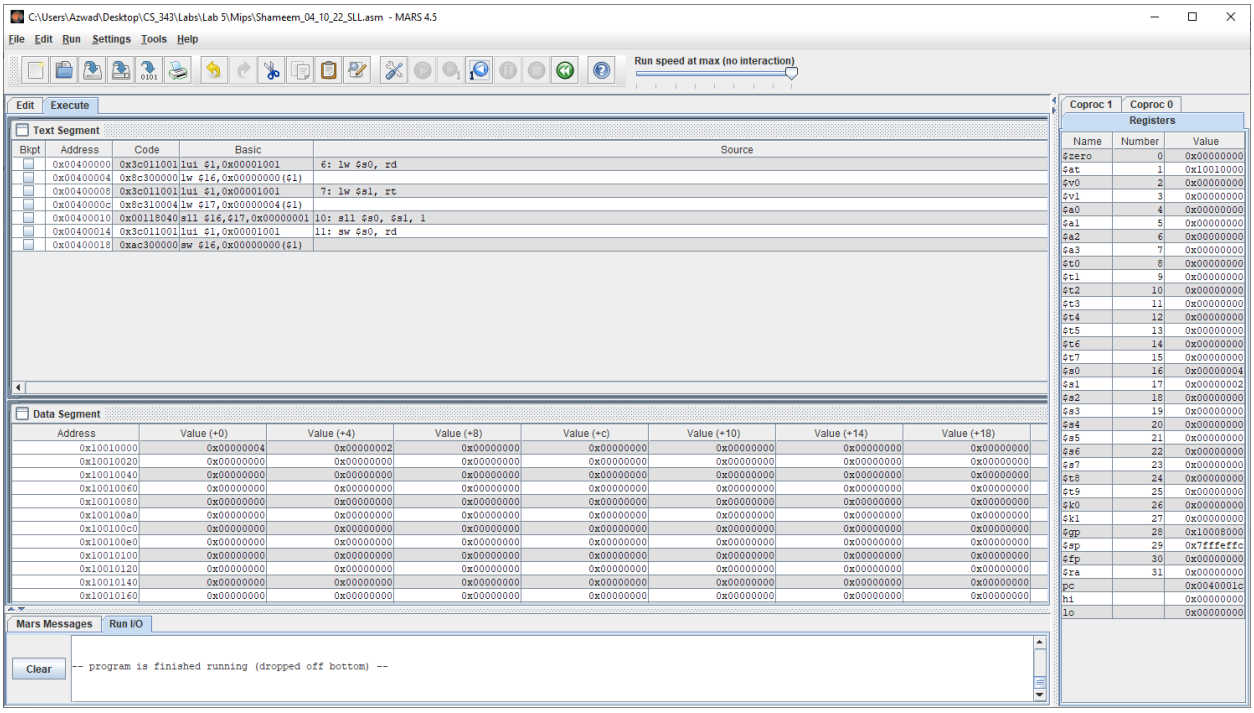


Figure 73: Executed successfully in MARs.

R [rd] (0x00000004) = R[rt] (0x00000002) << shamt

Values RD, RS shown clearly in data segment in the order of 0x00000004, 0x00000002. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 71. Therefore, SLL is the same on MARS and ModelSim waveform, so the VHDL code for SLL is working correctly.

Shift Right Logical - SRL

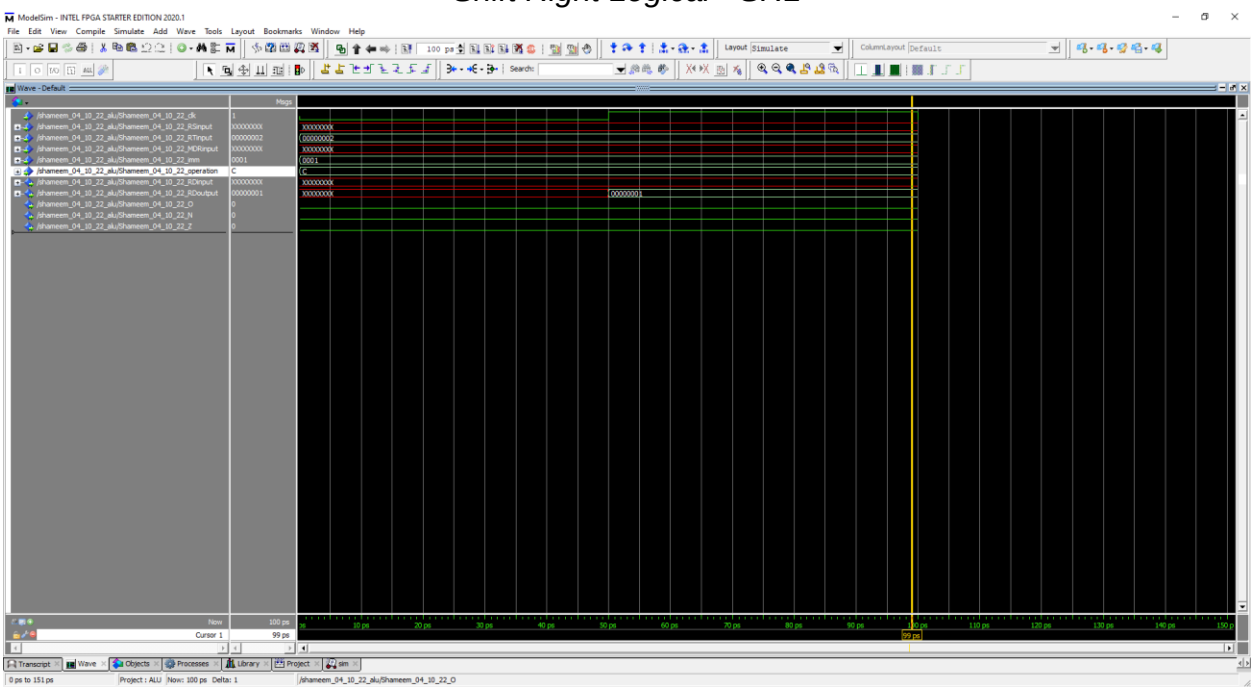


Figure 74: Simulation to represent SRL (radix is hexadecimal for everything)

R [rd] (0x00000001) = R[rt] (0x00000002) >> shamt

In this simulation the operation is 1100 which stands for add so it takes RT and shifts it left by shamt, which then outputs in the RD. Furthermore, 0x00000002 shifted by shamt equals 0x00000001, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because shifting does not trigger an overflow.

Shameem_04_10_22_SRL.asm

```
1   .data
2   rd: .word 0x00000000
3   rt: .word 0x00000002
4
5   .text
6   lw $s0, rd
7   lw $s1, rt
8
9   # rd = rt << shamt
10  srl $s0, $s1, 1
11  sw $s0, rd
12
```

Figure 75: MIPS srl

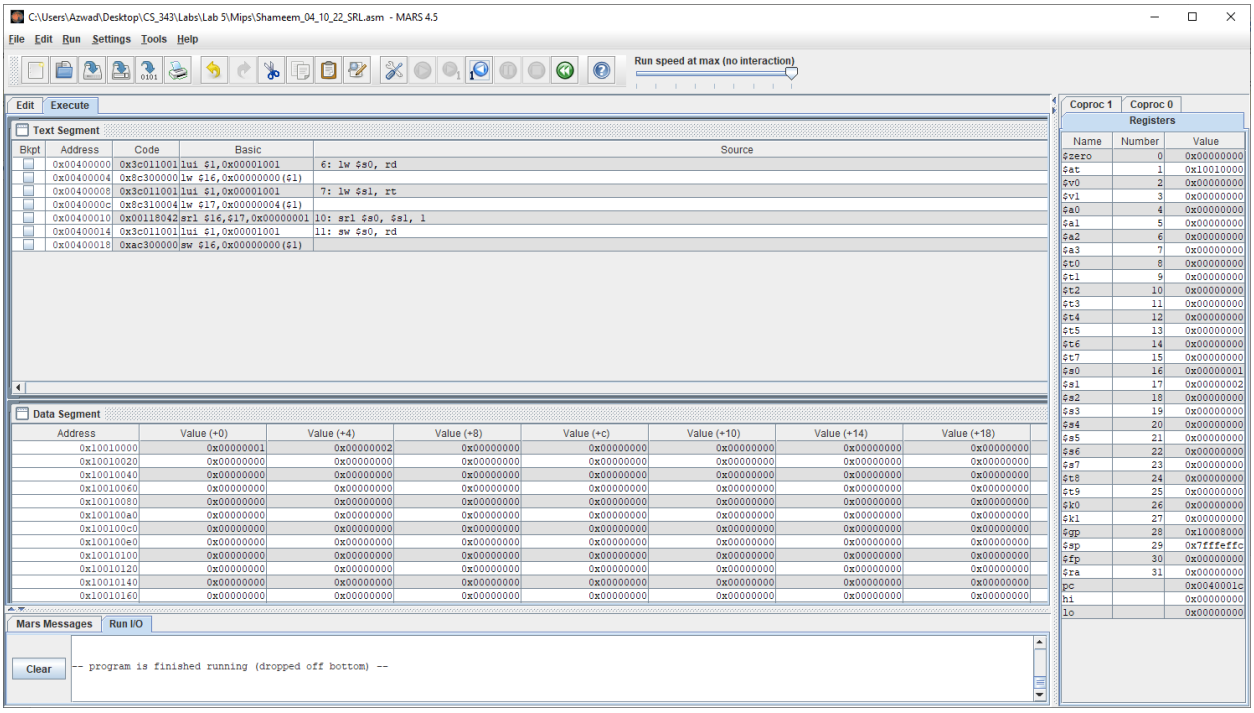


Figure 76: Executed successfully on MARs.

R [rd] (0x00000001) = R[rt] (0x00000002) >> shamt

Values RD, RS shown clearly in data segment in the order of 0x00000001, 0x00000002. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 74. Therefore, SRL is the same on MARS and ModelSim waveform, so the VHDL code for SRL is working correctly.

Shift Left Arithmetic - SRA

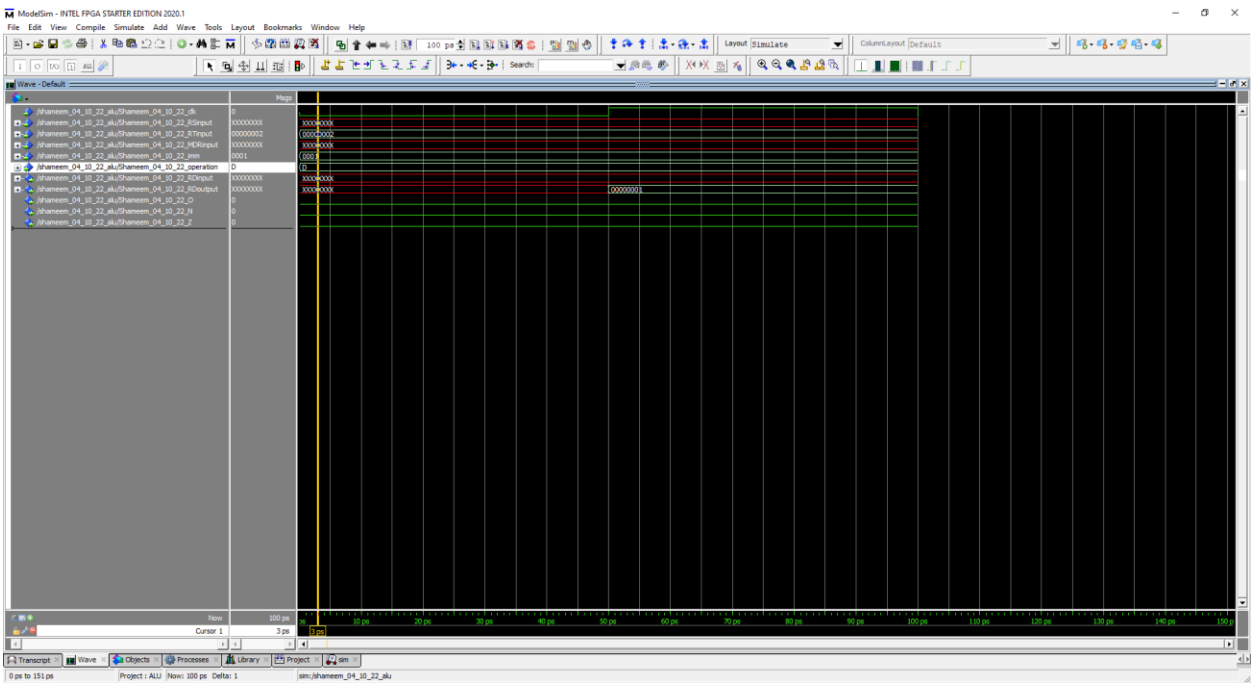


Figure 77: Simulation to represent SLL (radix is hexadecimal for everything)

R [rd] (0x00000001) = R[rt] (0x00000002) >>> shamt

In this simulation the operation is 1101 which stands for add so it takes RT and shifts it left by shamt, which then outputs in the RD. Furthermore, 0x00000002 shifted by shamt equals 0x00000001, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because shifting does not trigger an overflow.

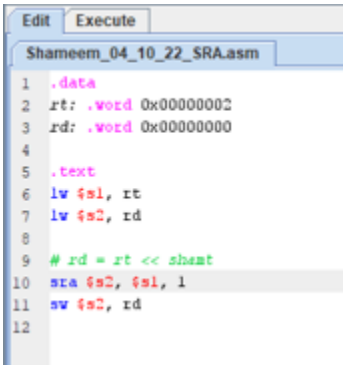


Figure 78: MIPS sra

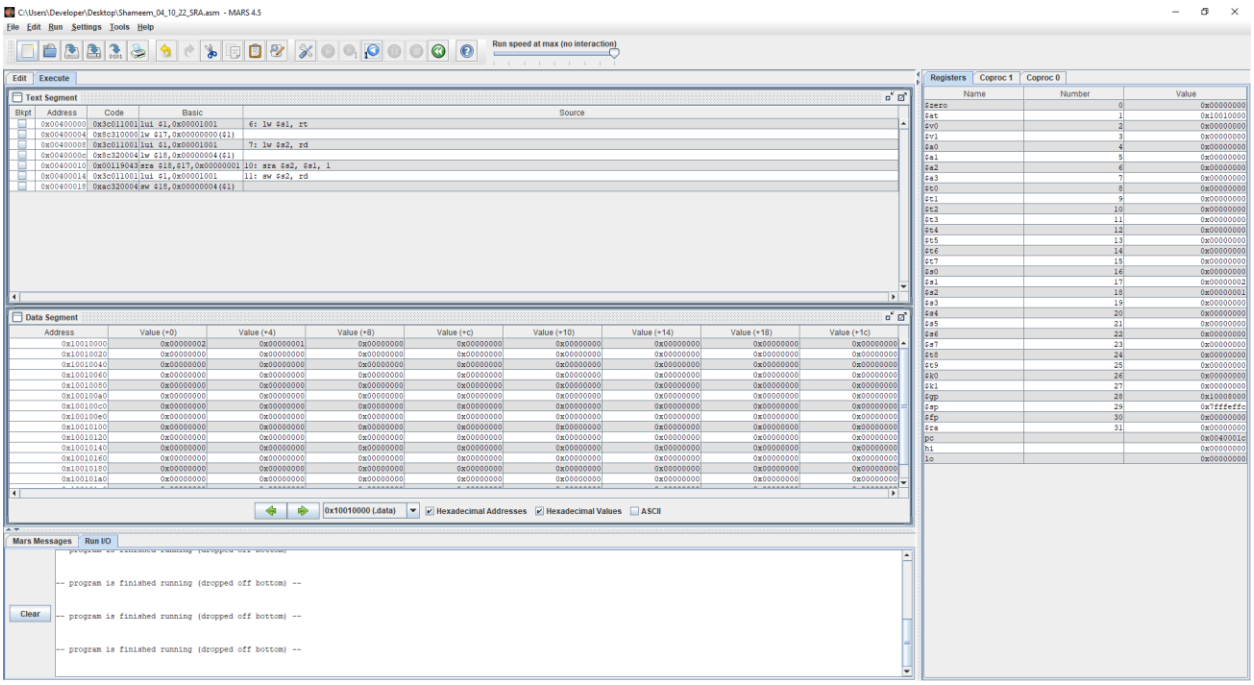


Figure 78: Executed successfully on MARs.

R [rd] (0x00000001) = R[rt] (0x00000002) >>> shamt

Values RS, RD shown clearly in data segment in the order of 0x00000002, 0x00000001. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 76. Therefore, SRA is the same on MARS and ModelSim waveform, so the VHDL code for SRA is working correctly.

Part II
Addi

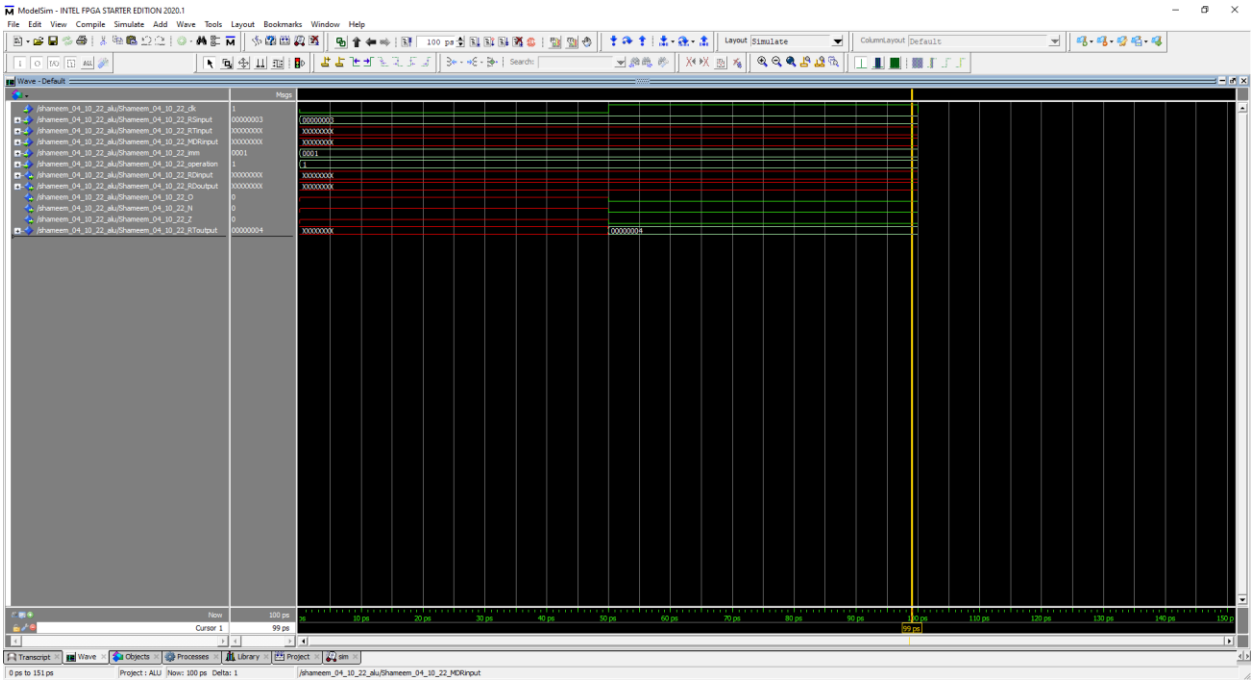


Figure 79: Simulation to represent Addi (radix is hexadecimal for everything)

R [rt] (0x00000004) = R[rs] (0x00000003) + SignExtImm

In this simulation the operation is 0001 which stands for addi so it takes RS and ZeroExtImm and adds them together and outputs them in the RD. Furthermore, 0x00000003 plus 0x00000001 equals 0x00000004, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because positive plus positive equals a positive does not trigger the overflow.

Edit | Execute

Shameem_04_10_22_Addi.asm

```
1   .data
2   rt: .word 0x00000000
3   rs: .word 0x00000003
4
5   .text
6   lw $s0, rt
7   lw $s1, rs
8
9   # rd = rt + signExtImm
10  addi $s0, $s1, 1
11  sw $s0, rt
12
```

Figure 80: MIPS addi

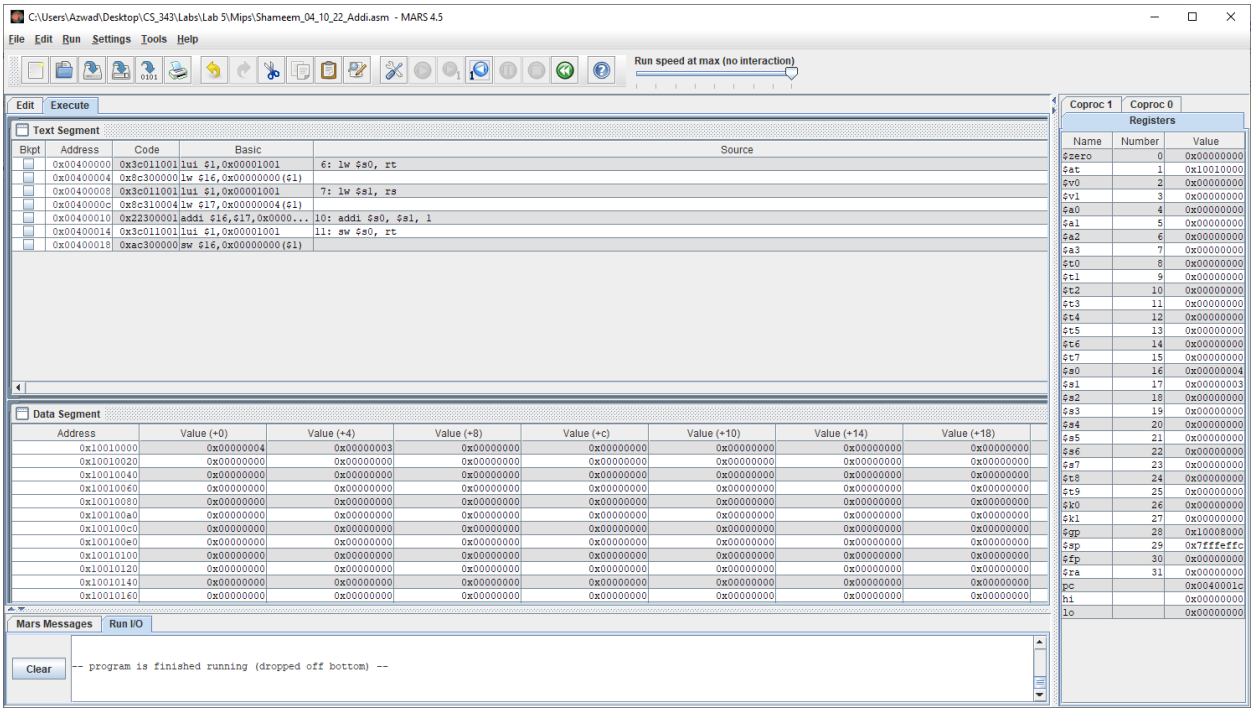


Figure 81: Executed successfully on MARs.

R [rt] (0x00000004) = R[rs] (0x00000003) + SignExtImm

Values RT, Rs shown clearly in data segment in the order of 0x00000004, 0x00000003. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 79. Therefore, Addi is the same on MARS and ModelSim waveform, so the VHDL code for Addi is working correctly.

Addiu

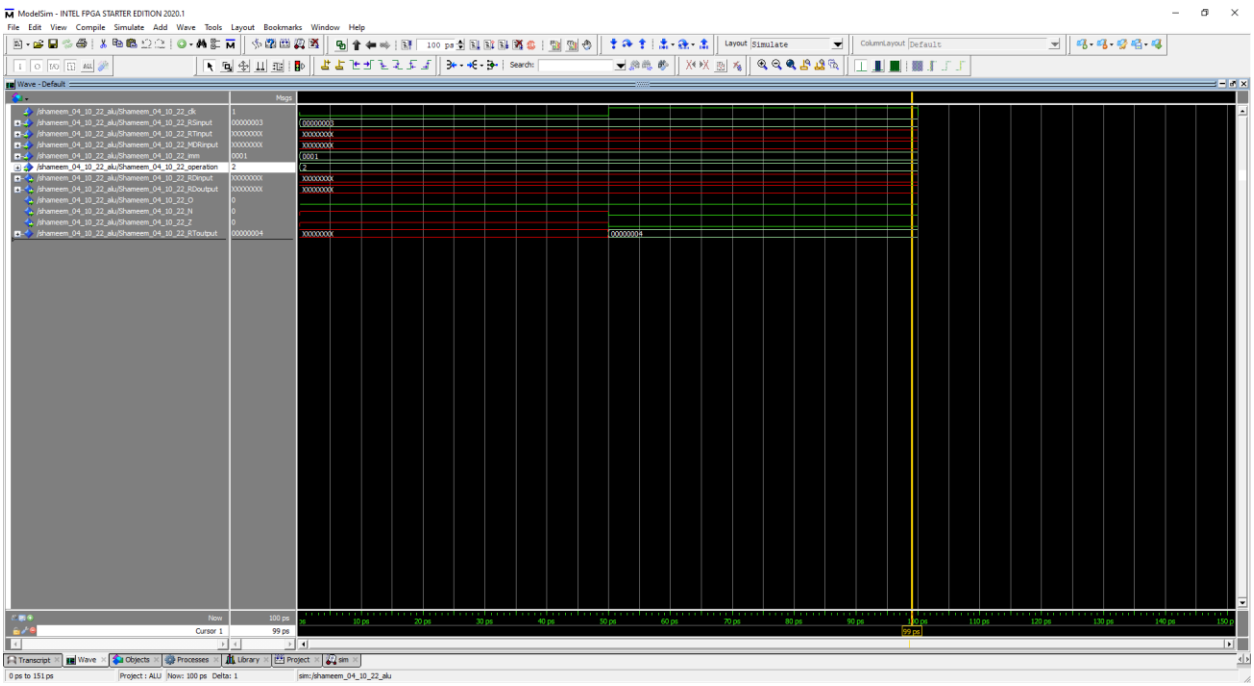


Figure 83: Simulation to represent Addiu (radix is hexadecimal for everything)

R [rt] (0x00000004) = R[rs] (0x00000003) + SignExtImm

In this simulation the operation is 0010 which stands for addiu, so it takes RS and ZeroExtImm and adds them together and outputs them in the RD. Furthermore, 0x00000003 plus 0x00000001 equals 0x00000004, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because addiu does not detect overflows.

Edit | Execute

Shameem_04_10_22_Addiu.asm

```
1   .data
2   rt: .word 0x00000000
3   rs: .word 0x00000003
4
5   .text
6   lw $s0, rt
7   lw $s1, rs
8
9   # rd = rt + signExtImm
10  addiu $s0, $s1, 1
11  sw $s0, rt
12
```

Figure 83: MIPS addiu

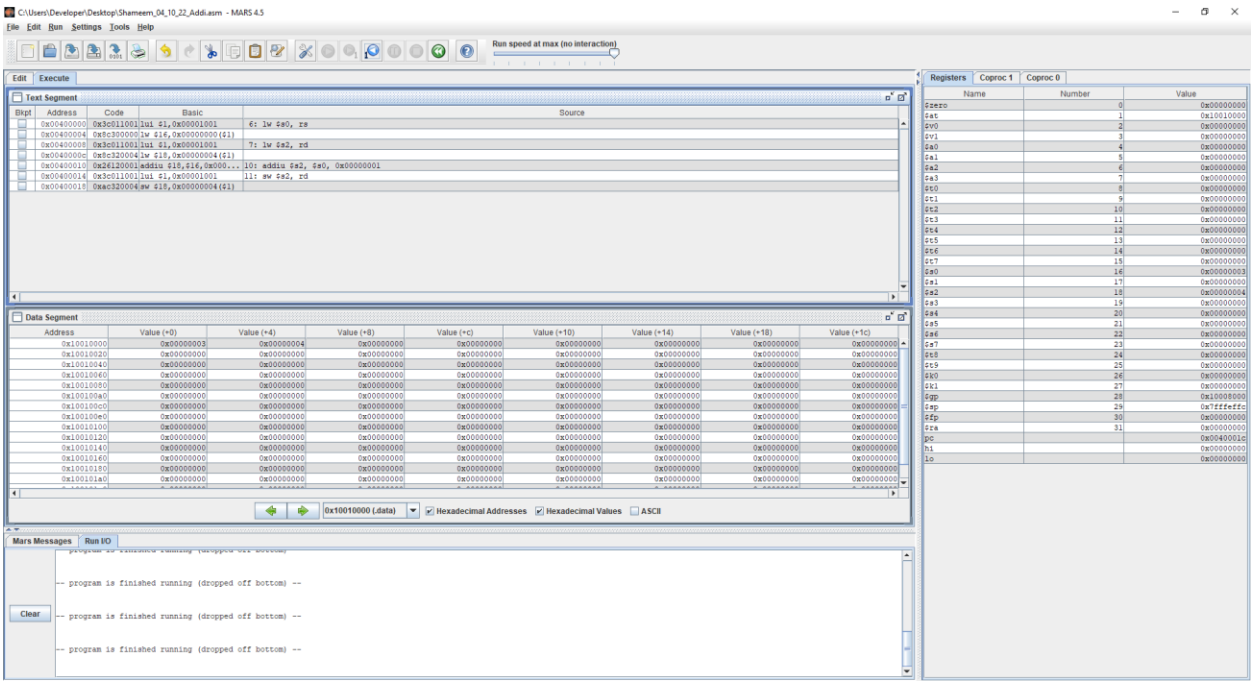


Figure 84: Executed successfully on MARs.

R [rt] (0x00000004) = R[rs] (0x00000003) + SignExtImm

Values RT, Rs shown clearly in data segment in the order of 0x00000004, 0x00000003. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 79. Therefore, Addiu is the same on MARS and ModelSim waveform, so the VHDL code for Addiu is working correctly.

Andi

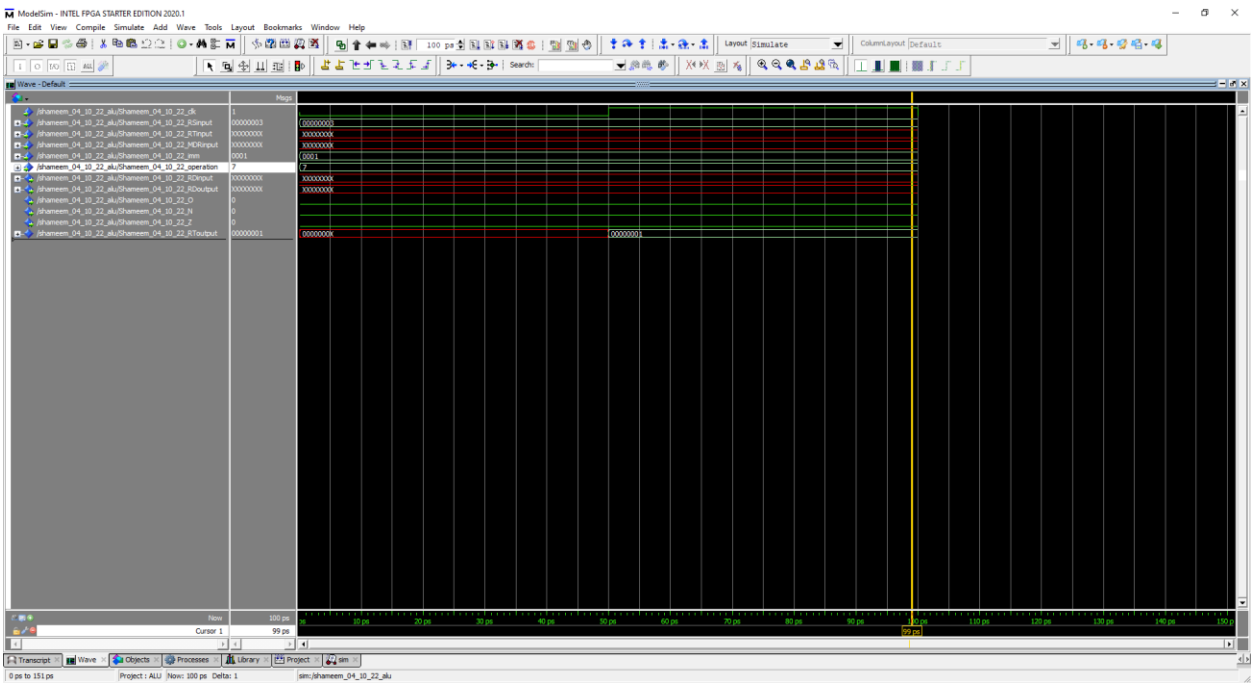


Figure 85: Simulation to represent Andi (radix is hexadecimal for everything)

R [rt] (0x00000001) = R[rs] (0x00000003) & ZeroExtImm

In this simulation the operation is 0111 which stands for andi, so it takes RS and RT and adds them together and outputs them in the RD. Furthermore, 0x00000003 & 0x00000001 equals 0x00000001, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because andi does not trigger overflows.

Edit | Execute

Shameem_04_10_22_Andi.asm

```
1   .data
2   rt: .word 0x00000000
3   rs: .word 0x00000003
4
5   .text
6   lw $s0, rt
7   lw $s1, rs
8
9   # rd = rt & zeroExtImm
10  andi $s0, $s1, 0x00000001
11  sw $s0, rt
12
```

Figure 86: MIPS andi

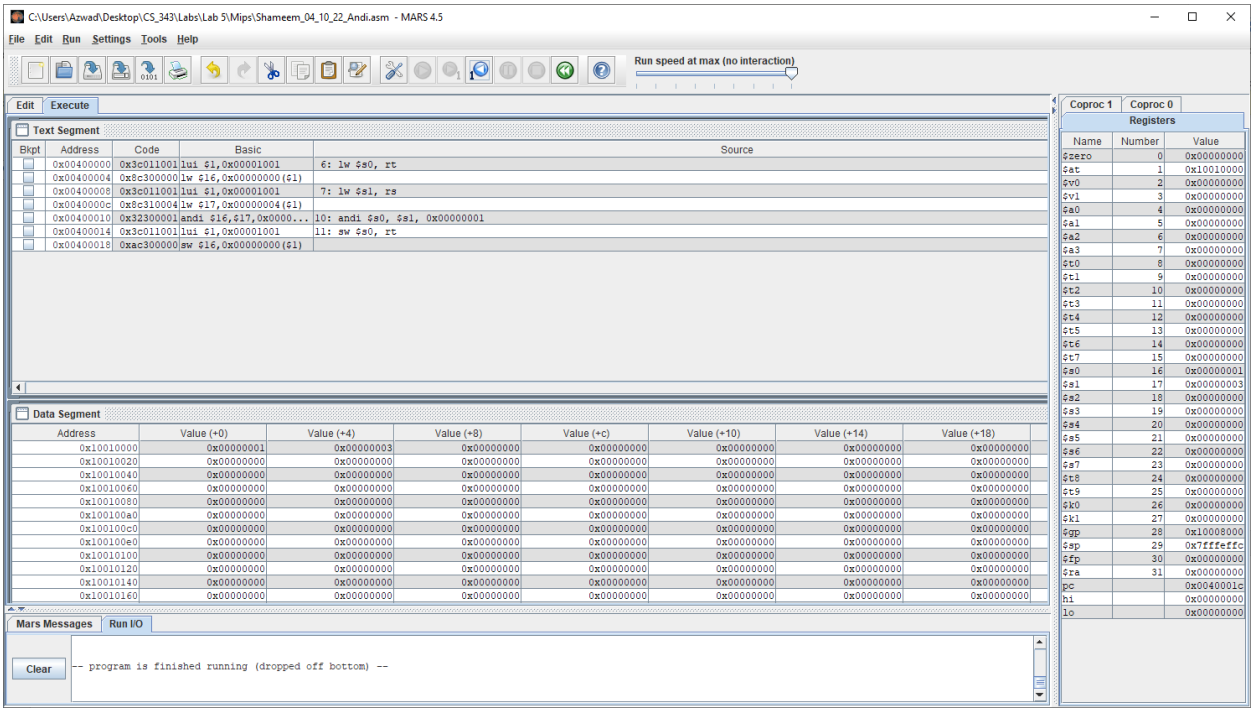


Figure 87: Executed successfully on MARs.

R [rt] (0x00000001) = R[rs] (0x00000003) & ZeroExtImm

Values RT, RS shown clearly in data segment in the order of 0x00000001, 0x00000003. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 82. Therefore, Andi is the same on MARS and ModelSim waveform, so the VHDL code for Andi is working correctly.

Ori

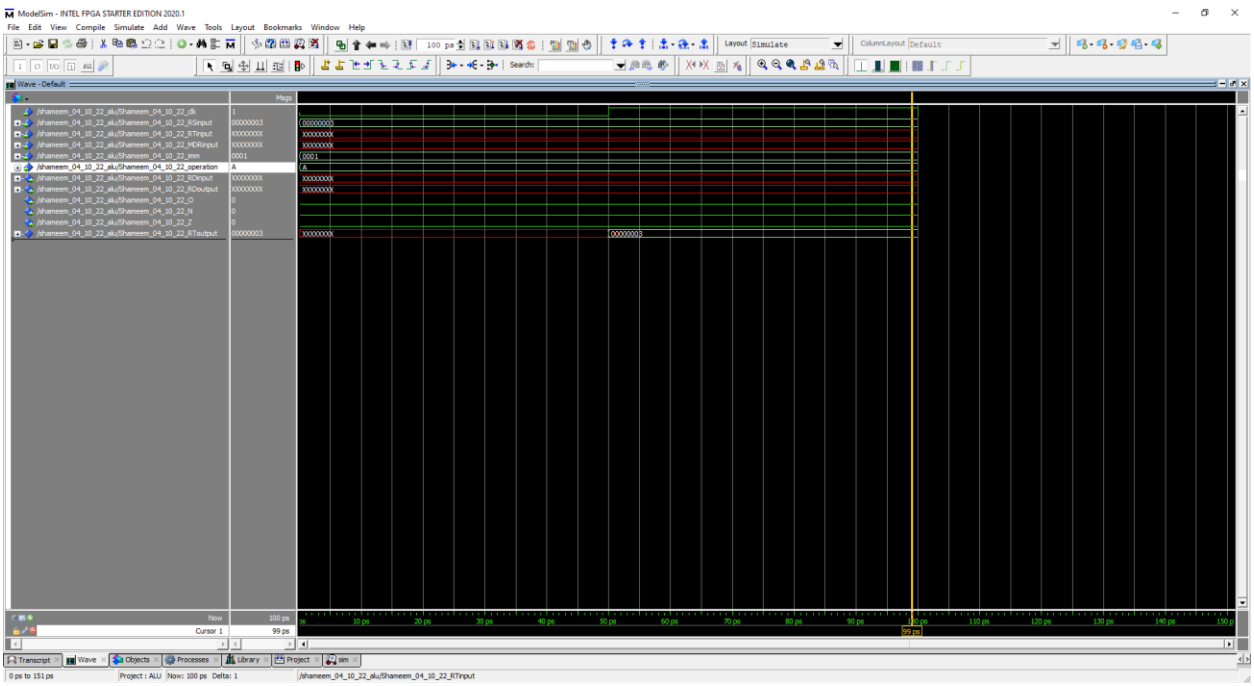


Figure 88: Simulation to represent Andi (radix is hexadecimal for everything)

R [rt] (0x00000003) = R[rs] (0x00000003) | ZeroExtImm

In this simulation the operation is 1010 which stands for Ori, so it takes RS and RT and adds them together and outputs them in the RD. Furthermore, 0x00000003 or 0x00000001 equals 0x00000003, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because Ori does not trigger overflows.

Edit | Execute

Shameem_04_10_22_Ori.asm

```
1   .data
2   rt: .word 0x00000000
3   rs: .word 0x00000003
4
5   .text
6   lw $s0, rt
7   lw $s1, rs
8
9   # rt = rs | ZeroExtImm
10  ori $s0, $s1, 1
11  sw $s0, rt
12
13
```

Figure 89: MIPS ori

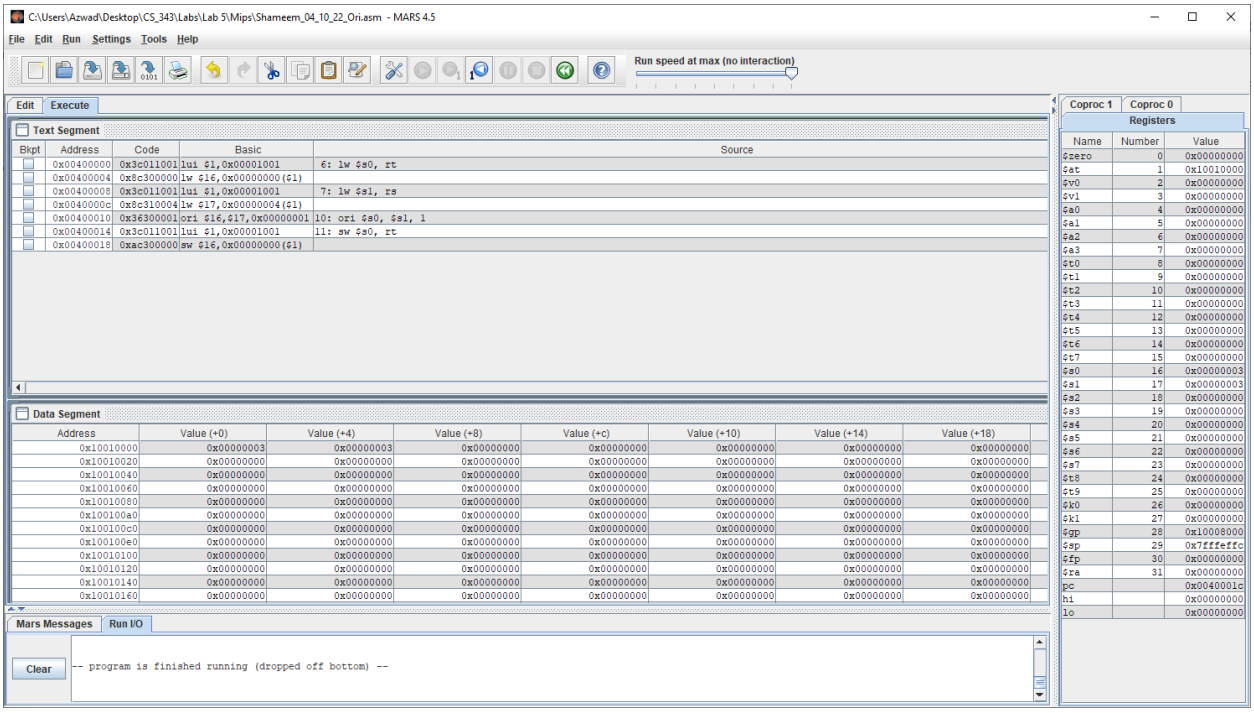


Figure 90: Executed successfully on MARs.

R [rt] (0x00000003) = R[rs] (0x00000003) | ZeroExtImm

Values RT, RS shown clearly in data segment in the order of 0x00000001, 0x00000003. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 87. Therefore, Ori is the same on MARS and ModelSim waveform, so the VHDL code for Ori is working correctly.

Part III

LW

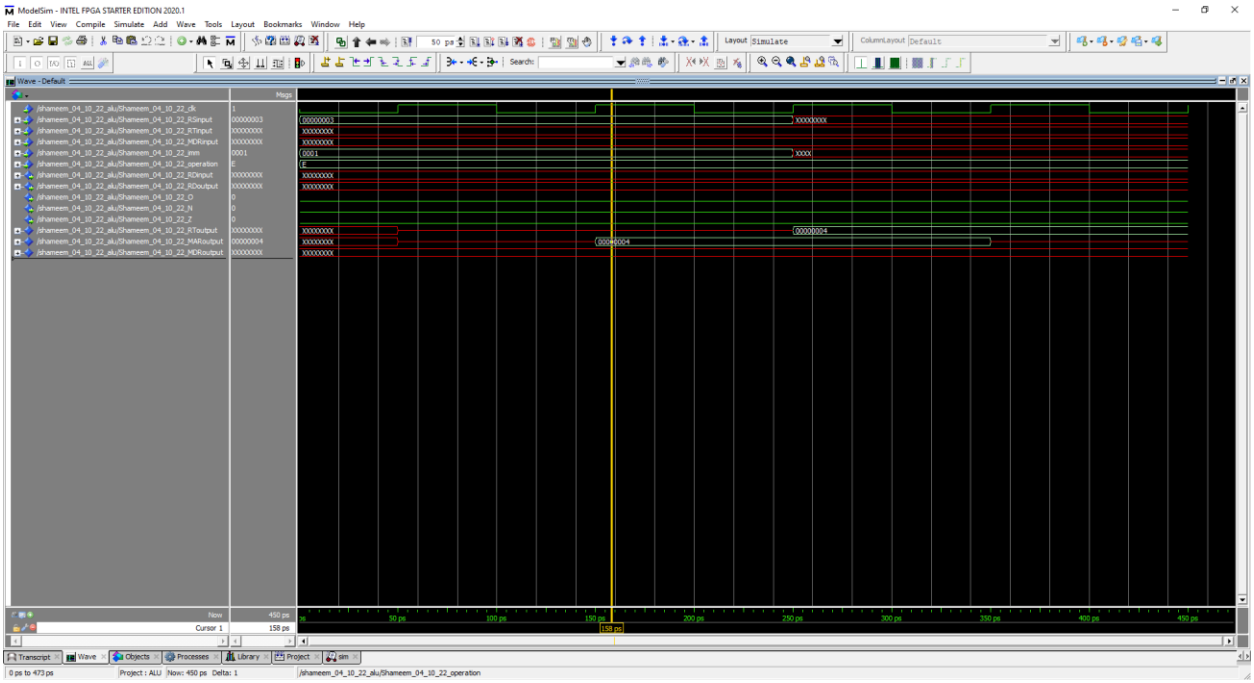


Figure 91: Simulation to represent LW (radix is hexadecimal for everything)

R [rt] (0x00000004) = M[R[rs] (0x00000003) + SignExtImm]

In this simulation the operation is 1111 which stands for LW, so it takes RS and SignExtImm and adds them together and outputs them into MAR which then goes into RT. Furthermore, 0x00000003 plus 0x00000001 equals 0x00000004, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because the output does not trigger the rules of overflow since a positive plus a positive can equal a positive without any overflow.

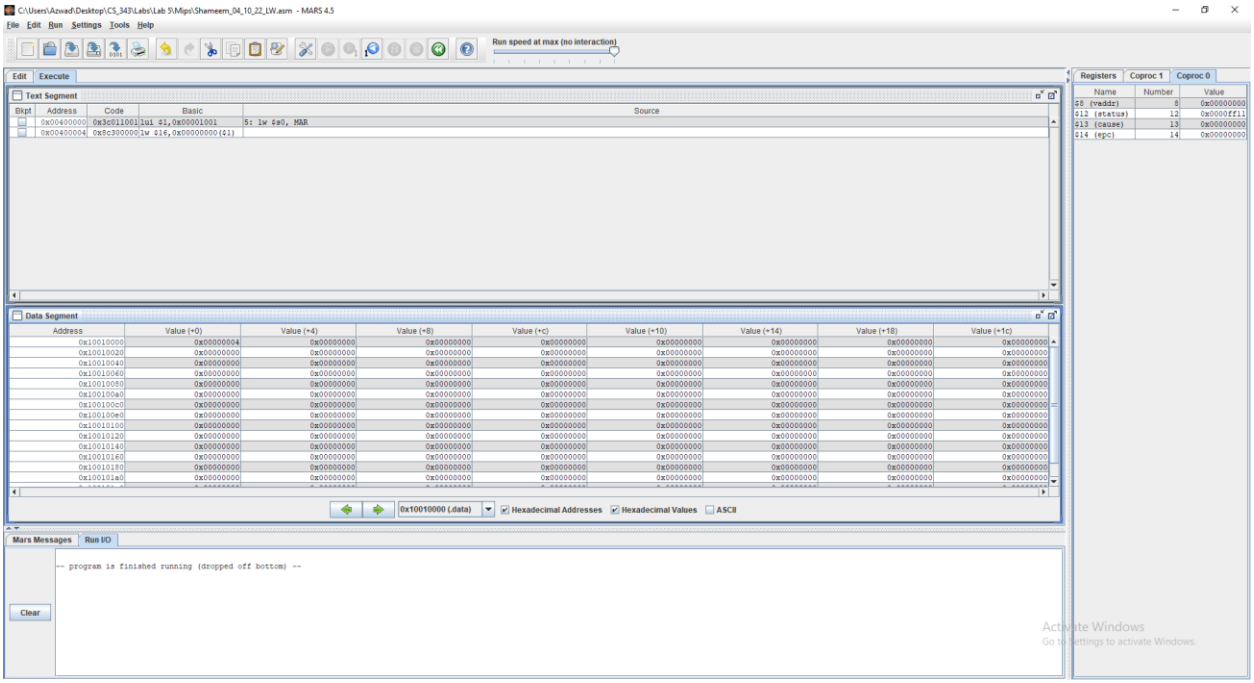


Figure 92: Executed successfully on MARs.

R [rt] (0x00000004) = M[R[rs] (0x00000003) + SignExtImm]

Values MAR shown clearly in data segment, which is 0x00000004. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 91. Therefore, lw is the same on MARS and ModelSim waveform, so the VHDL code for lw is working correctly.

Part IV
SW

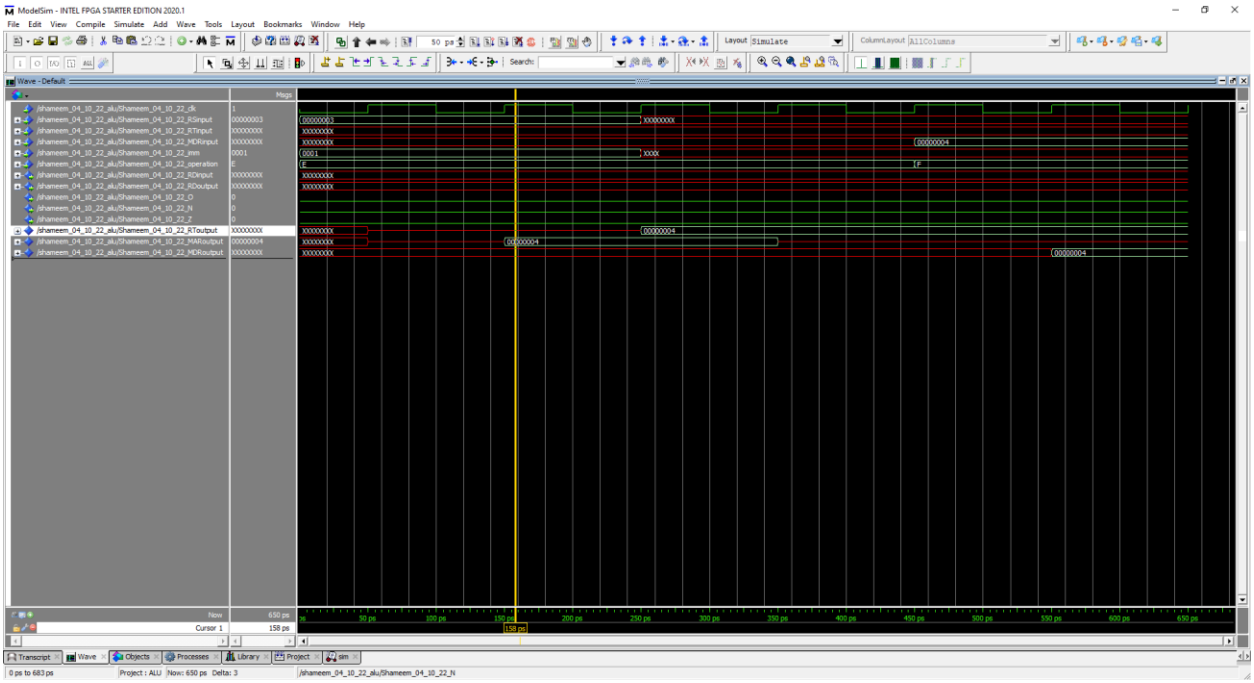


Figure 93: Simulation to represent SW (radix is hexadecimal for everything)

R [rt] (0x00000004) = M[R[rs] (0x00000003) + SignExtImm]

In this simulation the operation is 1110 at 450 ps which stands for SW, so it takes RS and SignExtImm and adds them together and outputs them into MDR which then goes into RT. Furthermore, 0x00000003 plus 0x00000001 equals 0x00000004, which means the flags zero and negative should be 0 as shown in the waveform because the output is clearly not negative or zero. The overflow flag is also 0 because the output does not trigger the rules of overflow since a positive plus a positive can equal a positive without any overflow.

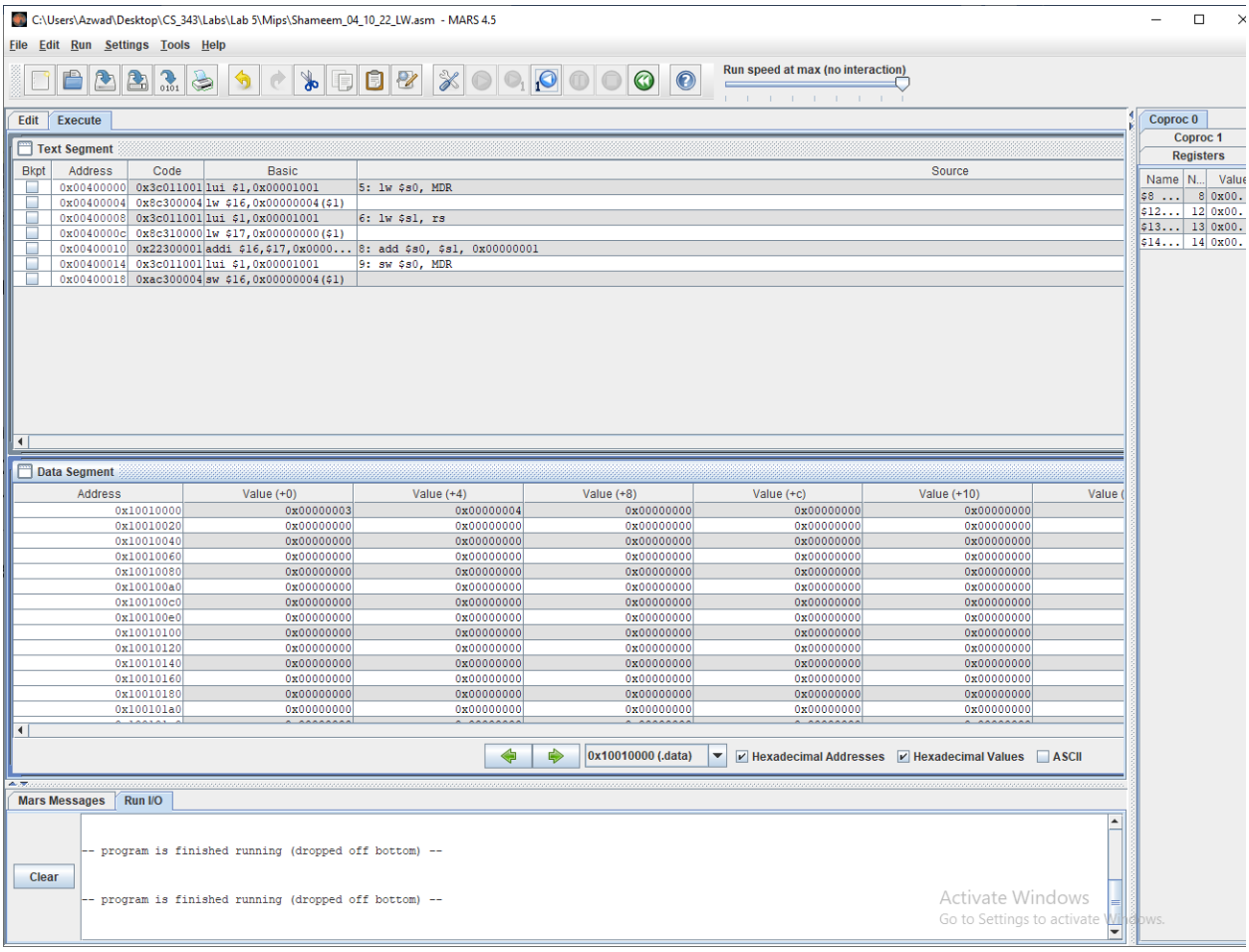


Figure 94: Executed successfully on MARs.

R [rt] (0x00000004) = M[R[rs] (0x00000003) + SignExtImm]

Values RS, MDR shown clearly in data segment, which is 0x00000003 and 0x00000004. The results of the MIPS code on MARS resulted in the same result as on the waveform simulation in figure 91. Therefore, SW is the same on MARS and ModelSim waveform, so the VHDL code for SW is working correctly.

Conclusion:

The Arithmetic Logic Unit lab was an important step because it allowed us to implement MIPS instructions in VHDL. MIPS instructions such as ADD, ADDU, SUB, SUBU, AND, NOR, OR, SLL, SRA, ADDI, ADDIU, ANDI, ORI, LW, SW was to be implemented by VHDL programming. Furthermore, the VHDL program that implemented MIPS instructions was then to be tested by utilizing ModelSim simulations and MIPS instructions on MARS in order to verify correctness. After completing the Arithmetic Logic Unit lab we have taken a step forward closer to making a CPU because we have proved that we can create instructions that a CPU may have taken through VHDL programming.