CS 342000 | CS343000
Instructor: Professor Izidor Gertner
Spring 2022
Azwad Shameem, 4/24/2022
Laboratory Project: Lab BNE, BEQ, J

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 this lab is to utilize the knowledge from the previous lab to implement MIPS instructions BEQ, BNE and j. Implementation of BEQ, BNE and j MIPS instructions requires the usage of multiplexers, comparators and the adder components in order to properly test the usage of BEQ, BNE and j MIPS instructions. In addition to implementing the MIPS instructions we will have to create a Next Address Logic Unit to output the next address of the instruction whether or not a jump or branching occurs, which will be shown in the PC or otherwise known as the Program Counter.

Description of Functionality & Simulation:

Functionality:

Components

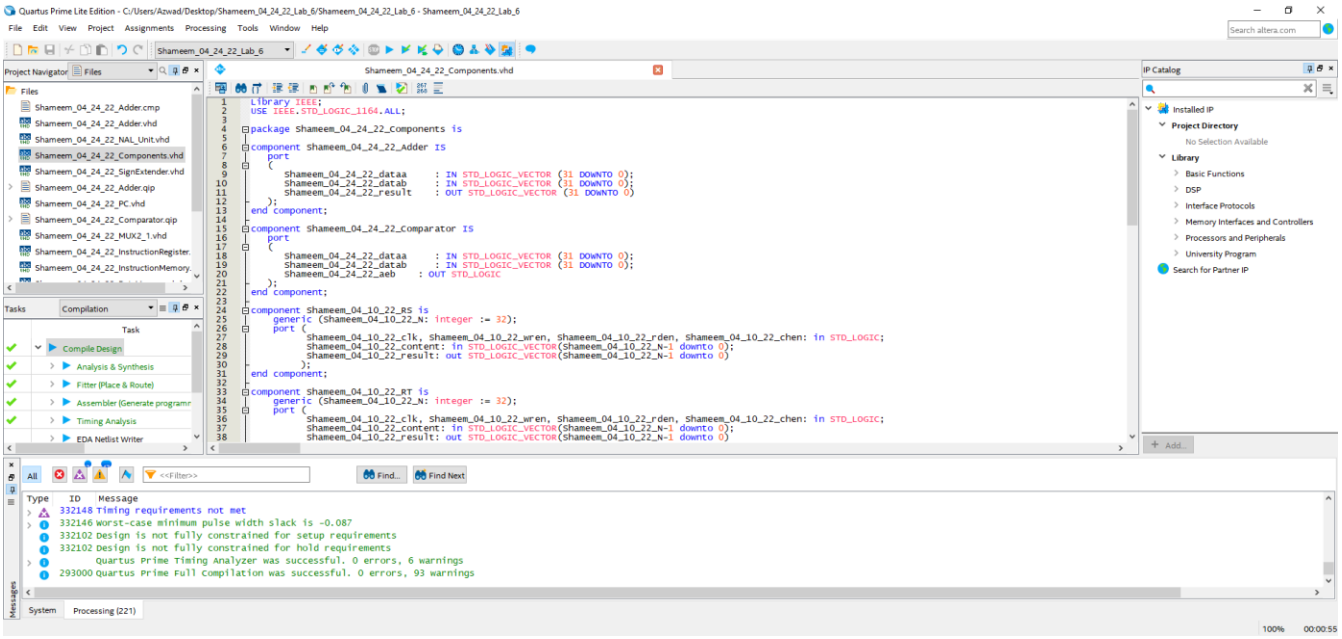


Figure 1: Components.vhd file which contains all of the components compiled successfully on Quartus.

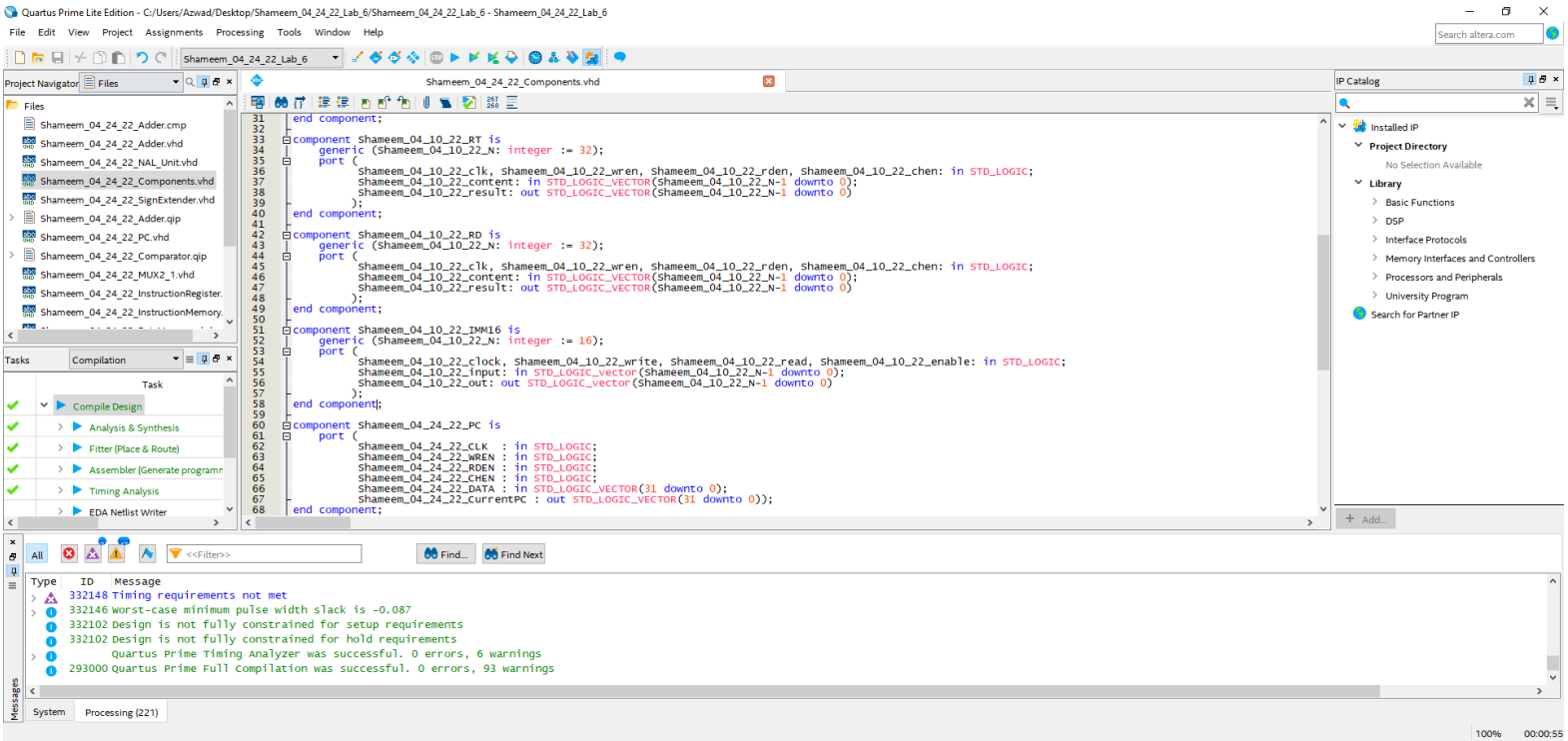


Figure 2: Components.vhd code continued.

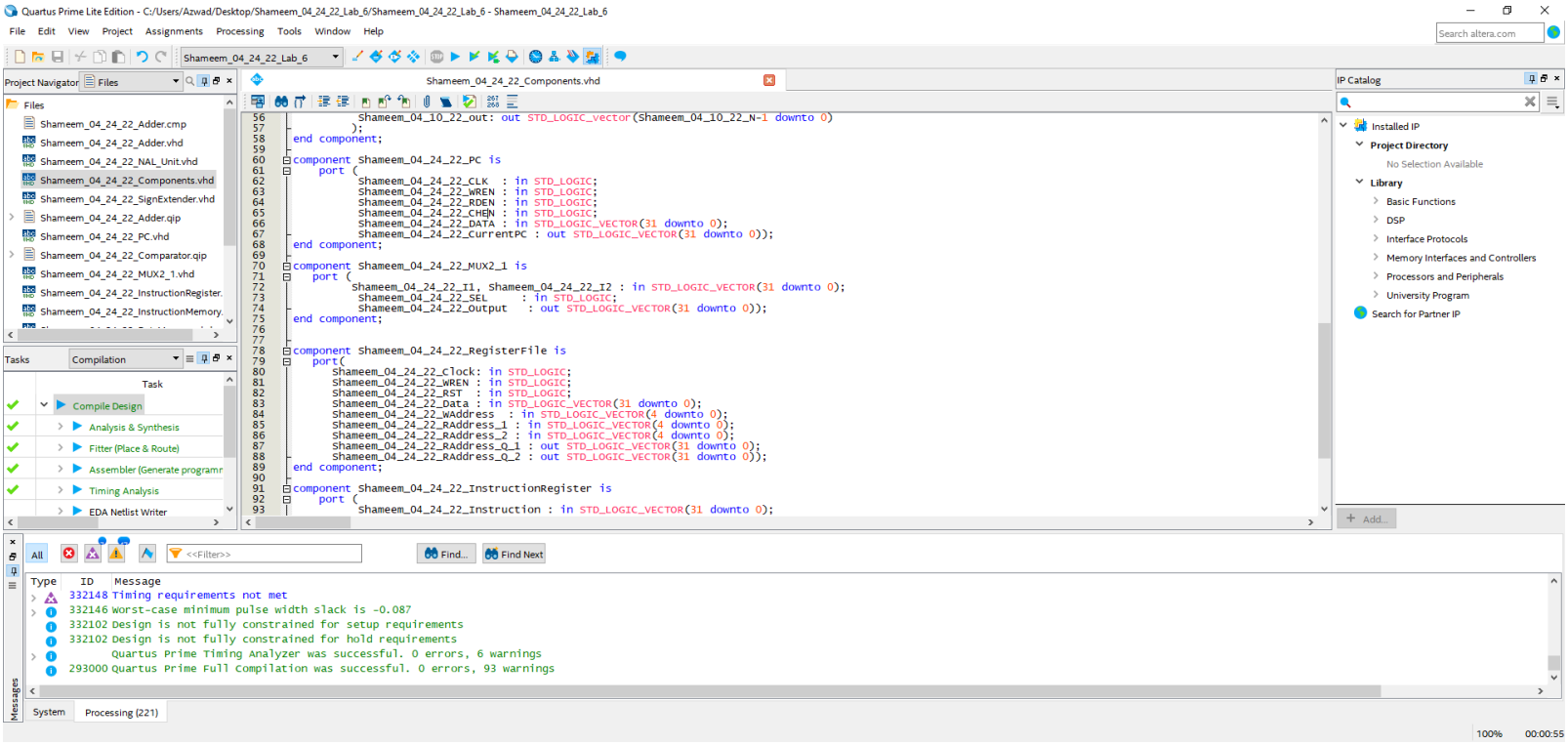


Figure 3: Components.vhd code continued

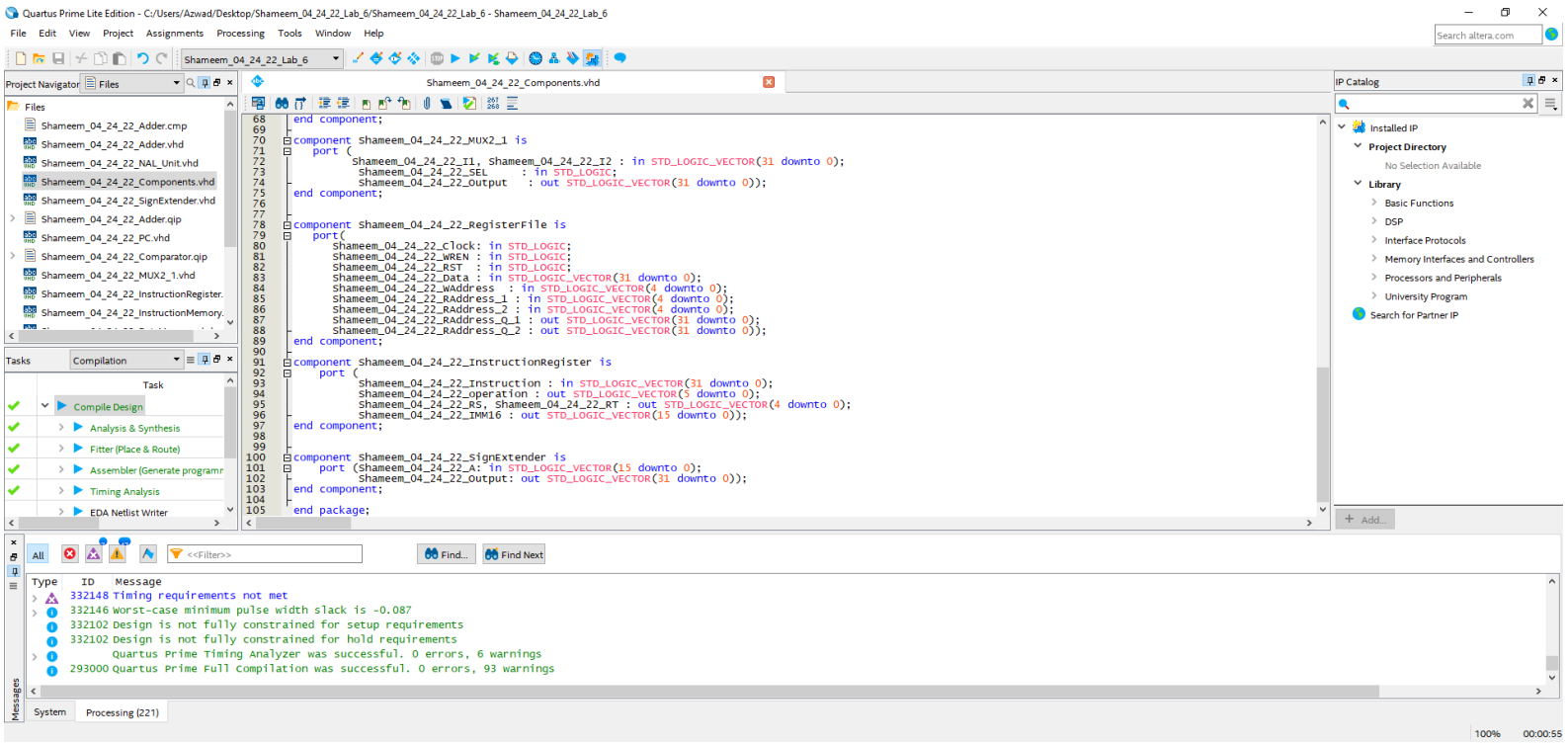


Figure 4: Components.vhd code continued.

The file in figures 1-4 contains all of the components utilized in the lab and is used to reduce redundancy of rewriting the components in each file.

RS

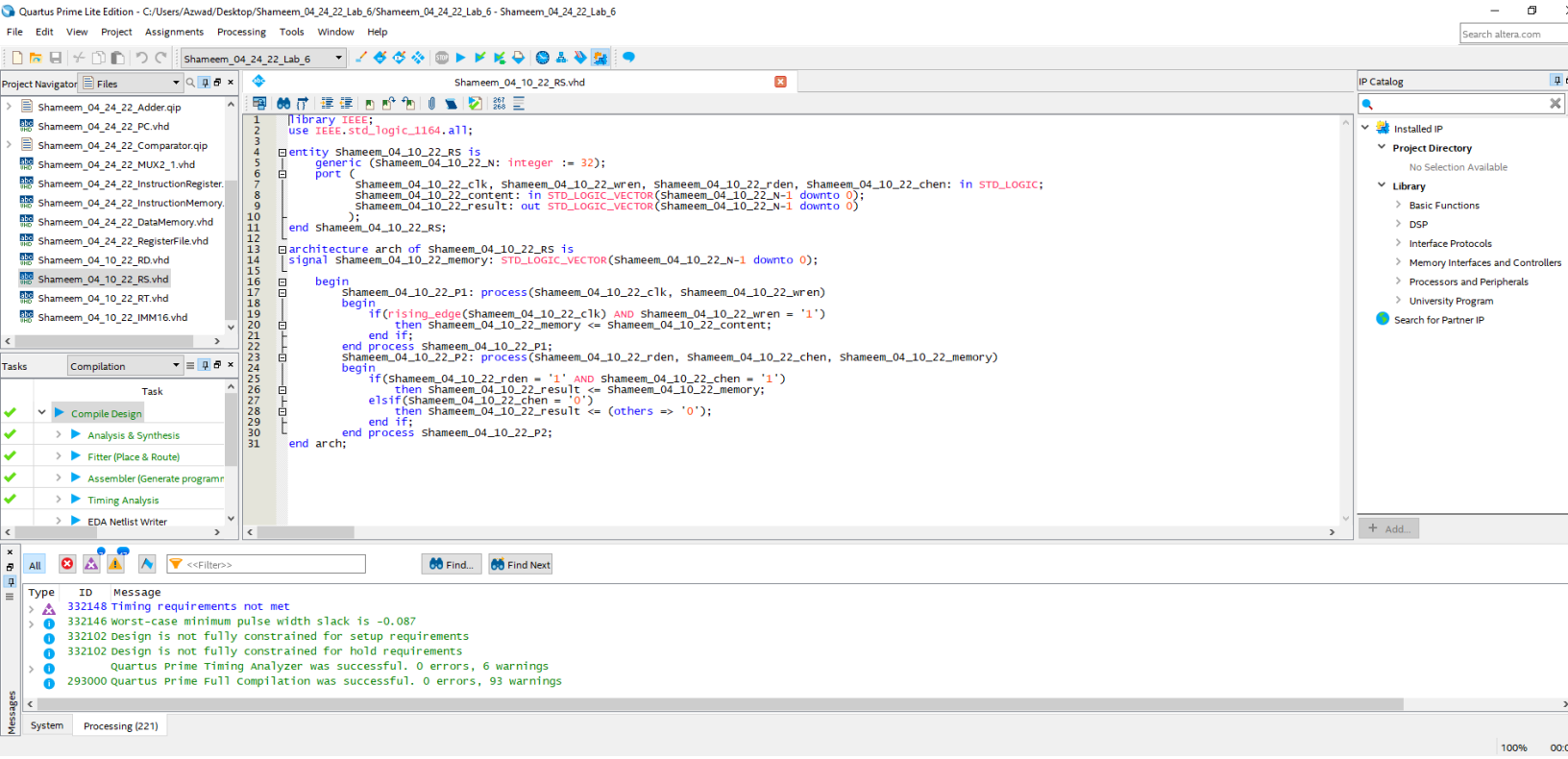


Figure 4: RS Register VHDL code.

This is the VHDL code for the RS register, this component is utilized in circuit and register files so that it the circuit can perform BNE, BEQ and j MIPS instructions. This is the RS Register that was utilized in the previous ALU lab.

RD

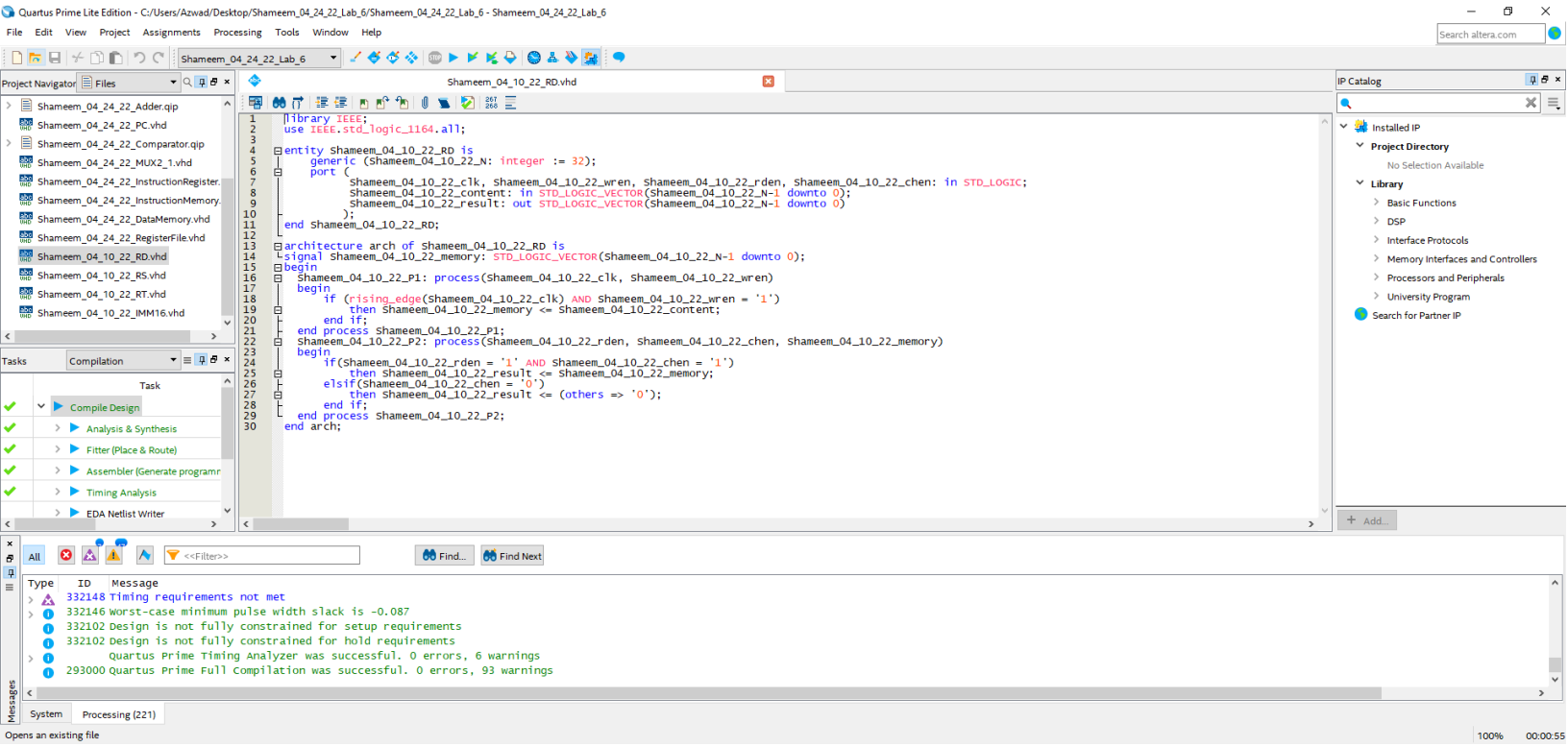


Figure 5: RD Register VHDL code.

This is the VHDL code for the RS register, this component is utilized in circuit and register files so that it the circuit can perform BNE, BEQ and j MIPS instructions. This is the RD Register that was utilized in the previous ALU lab.

RT

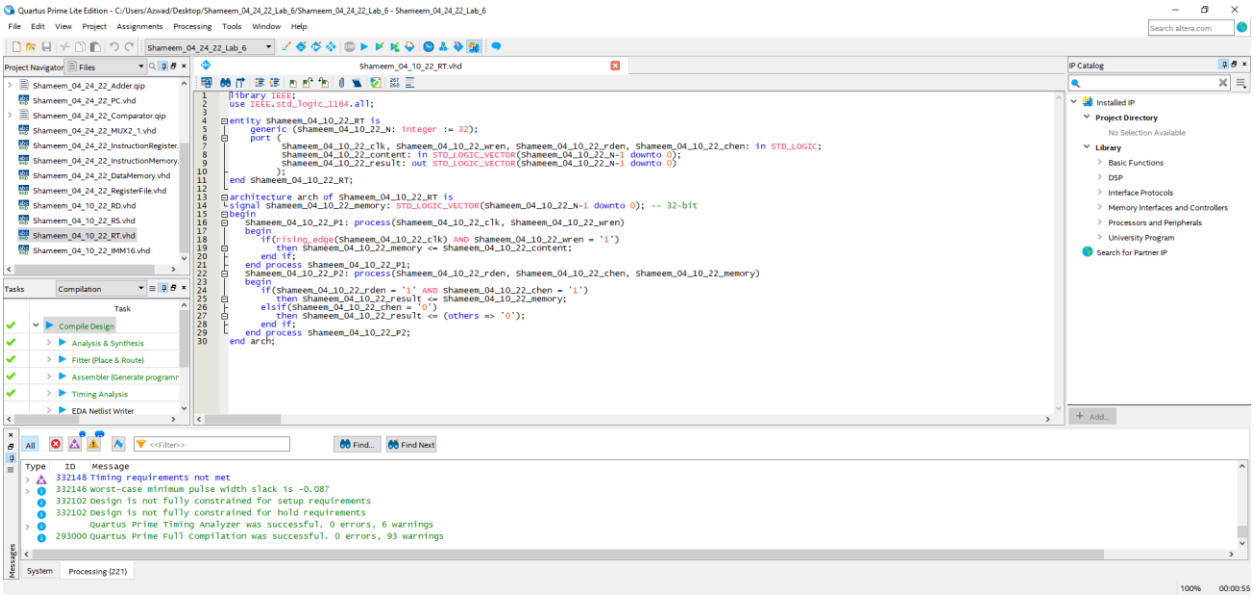


Figure 6: RT Register VHDL code.

This is the VHDL code for the RT register, this component is utilized in circuit and register files so that it the circuit can perform BNE, BEQ and j MIPS instructions. This is the RT Register that was utilized in the previous ALU lab.

IMM16 (Immediate 16-bit Register)

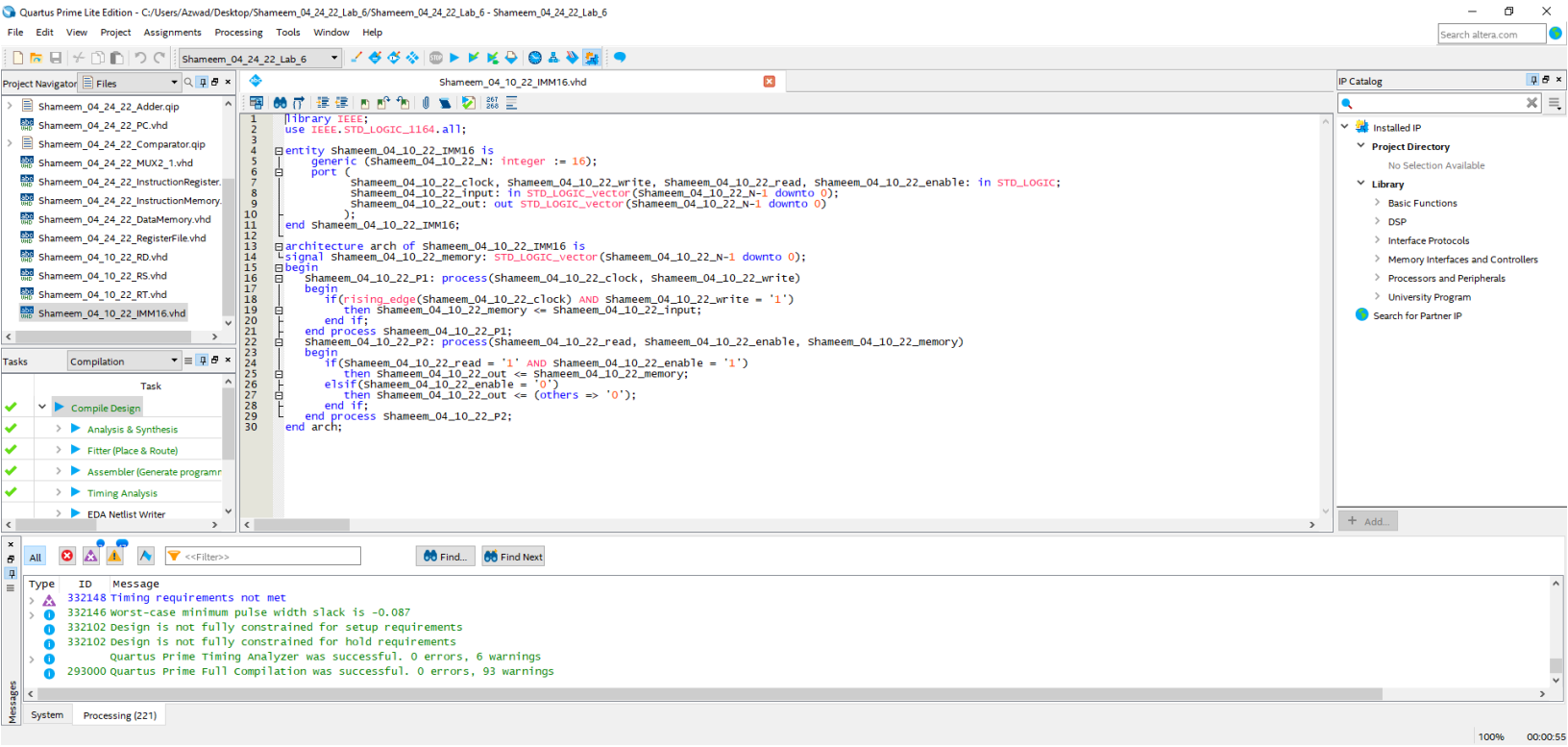


Figure 7: IMM16 Register VHDL code.

This is the VHDL code for the IMM16 register, this component is utilized in circuit and register files so that it the circuit can perform BNE, BEQ and j MIPS instructions.

This is the IMM16 Register that was utilized in the previous ALU lab.

Sign Extender

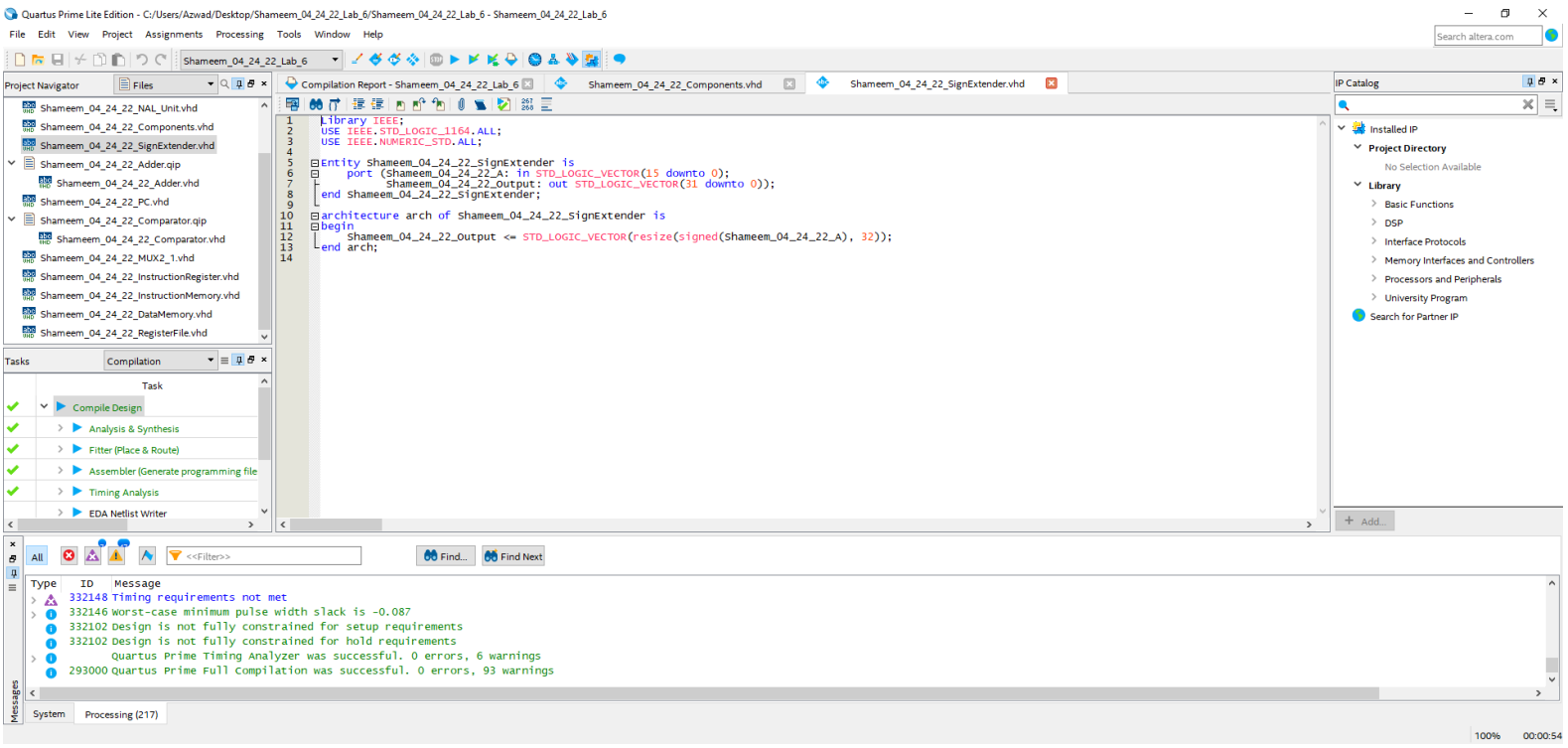


Figure 8: Sign Extender VHDL code compiled successfully in Quartus.

The Sign Extender component is used to extend the Immediate 16-bit Register so that it can be used in conjunction with the other registers. The other registers are 32-bits which is why the Sign Extender component is essential. Without the sign extender component, we would not be able to add or compare the 16 bit register because it would give an error when trying to compare two inputs of different number of bits.

PC

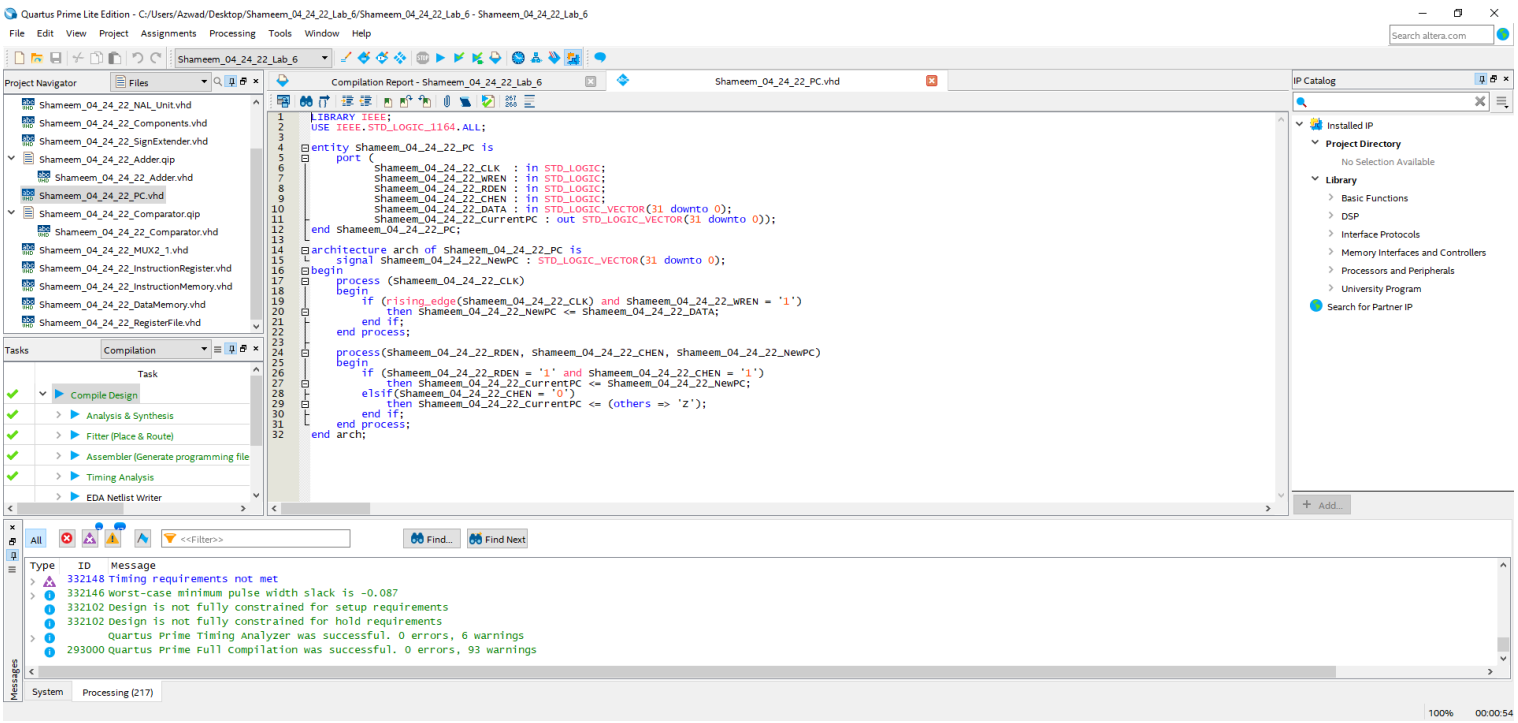


Figure 9: PC (Program Counter) VHDL code.

This is the VHDL code for the Program Counter, this component is utilized in circuit and register files so that it the circuit can perform BNE, BEQ and j MIPS instructions. The program counter differentiates itself from the other components is because it gets the next program counter which is the data which is used to replace the current program counter for every instruction.

Adder

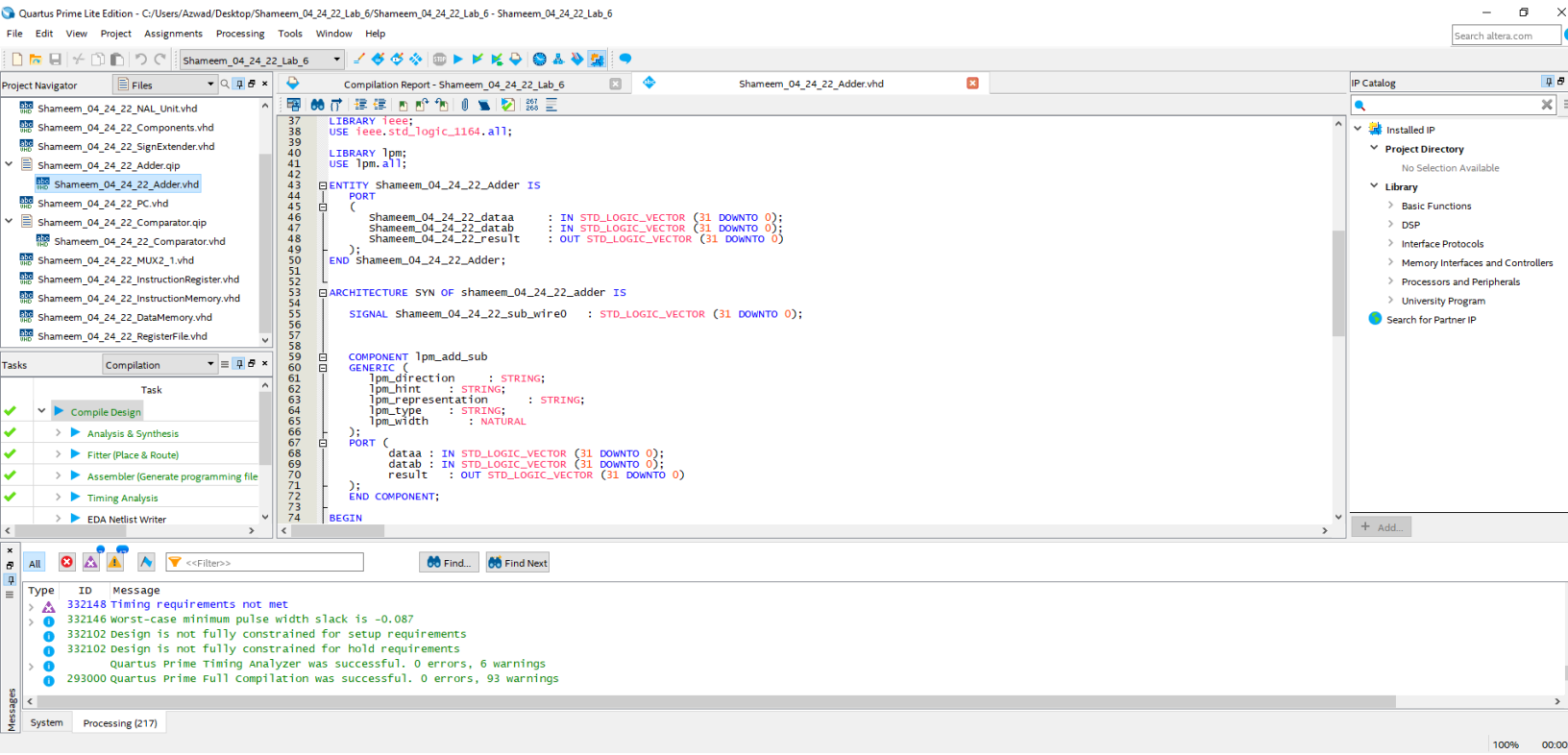


Figure 10: Adder VHDL code compiled successfully on Quartus

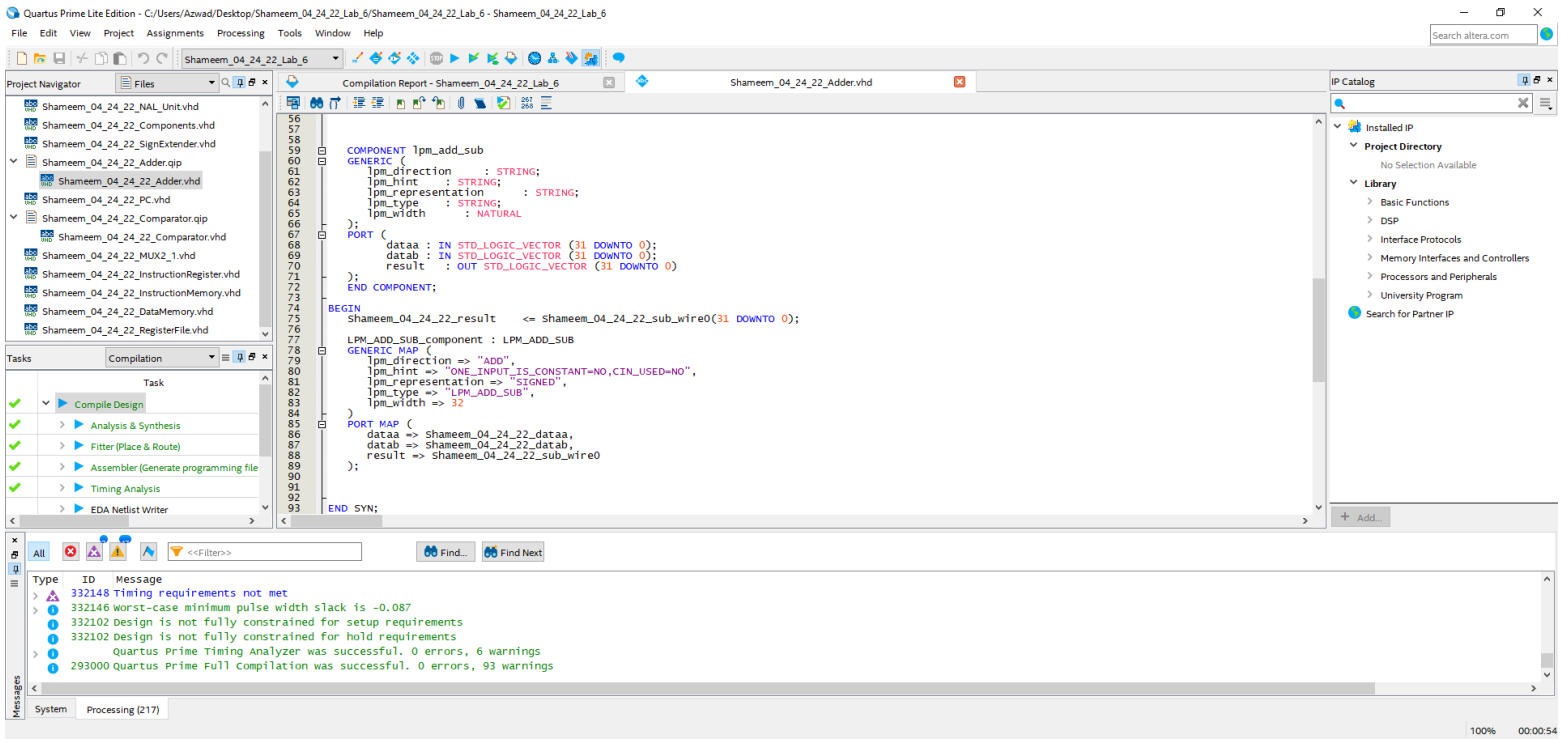


Figure 11: Adder VHDL code continued.

The idea of the Adder component is simple, but it is an essential part of the circuit. The adder component takes two 32-bit inputs dataa and datab. The two inputs are then added together and then the component then outputs the result of the addition when adding the two inputs dataa and datab. This Adder component is essential to the overall circuit basically it is very important because the instructions BNE, BEQ and J might require addition. Take for example when the BNE instruction is used and we want to branch to an instruction labeled L1, in which we have to take the program counter and add four. Clearly, in order to update the program counter, we will first need to compute the addition of the program counter and the number four which is why the Adder component is a must.

Comparator

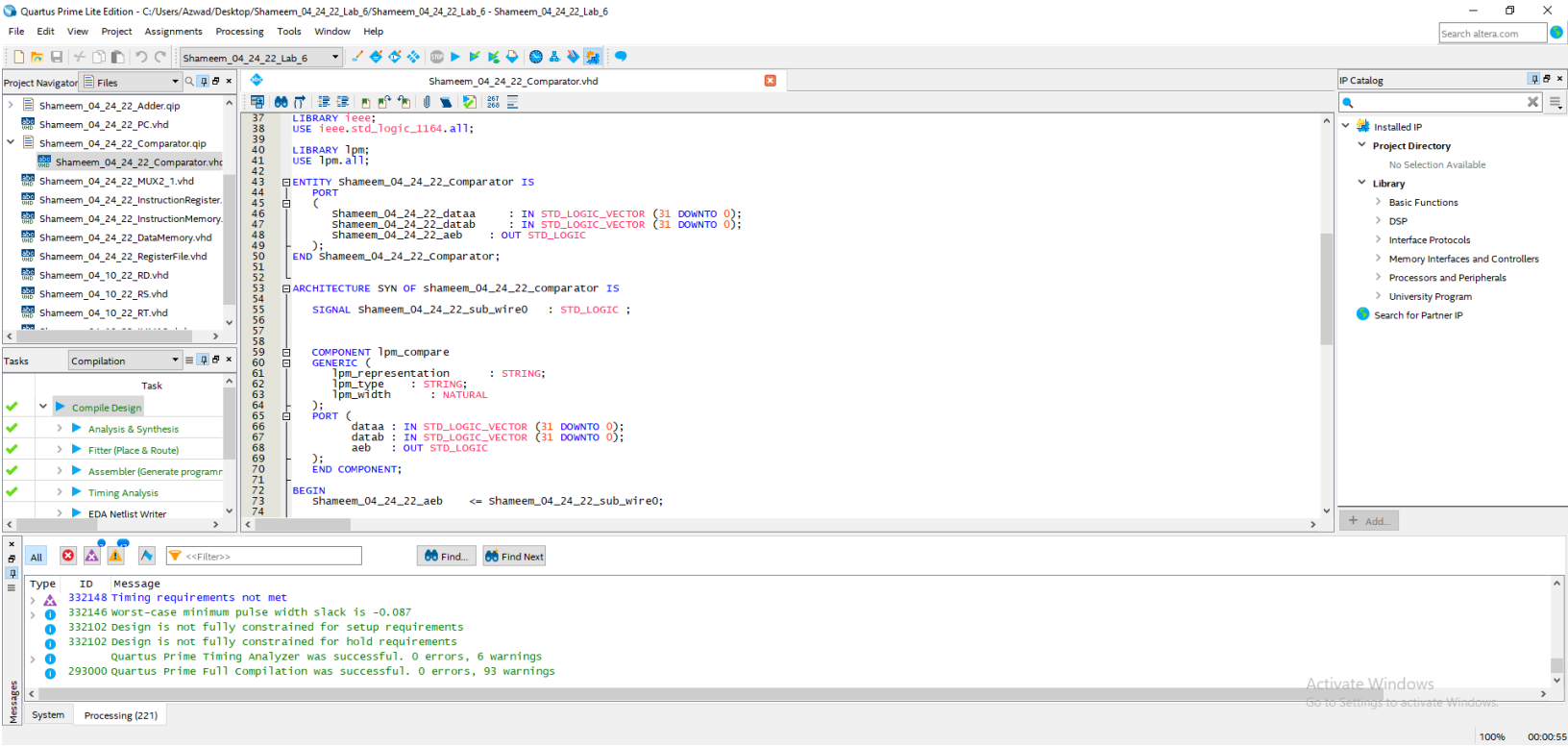


Figure 12: Comparator VHDL code compiled successfully in Quartus.

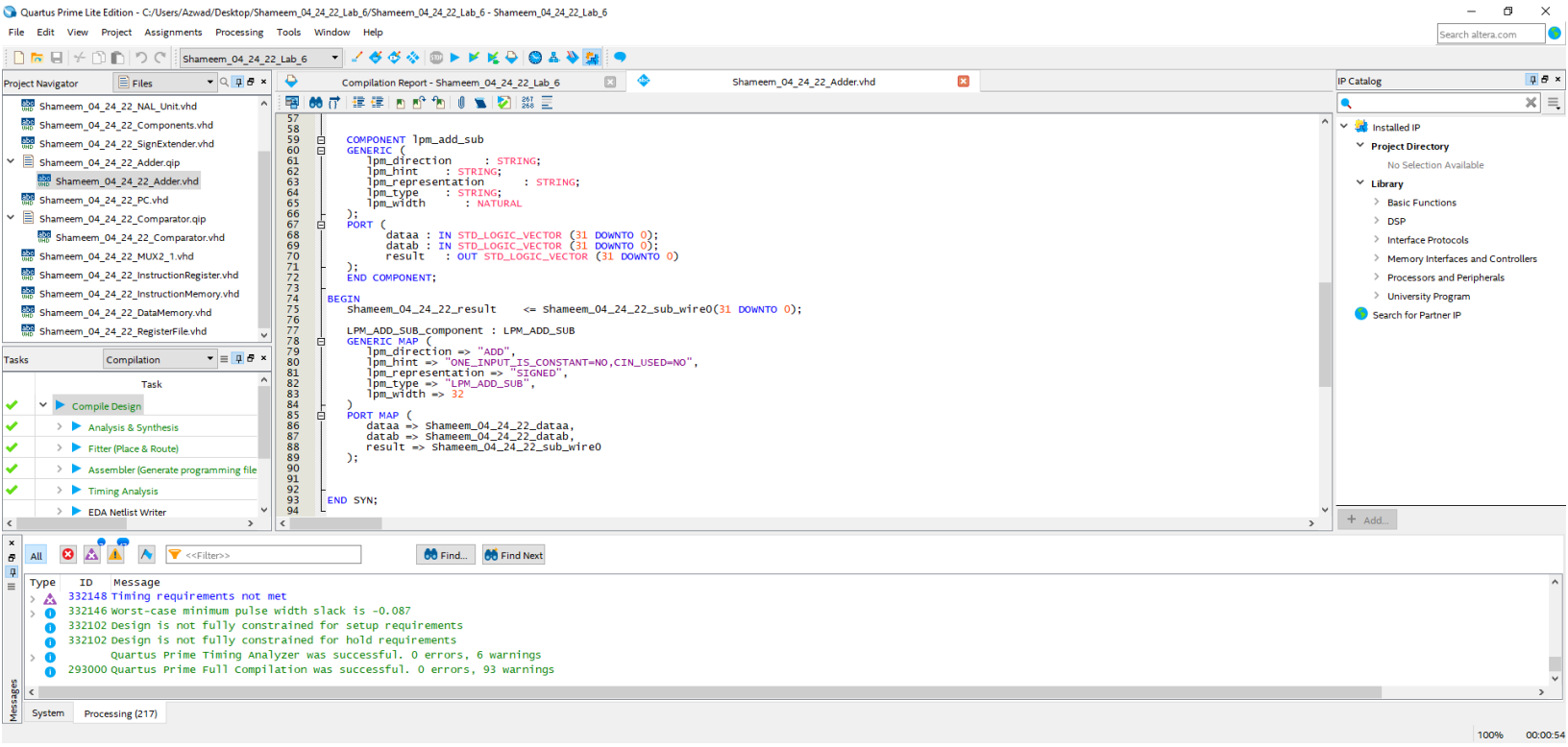


Figure 13: Comparator VHDL code continued.

The Comparator component is another essential component of the circuit, just like the Adder component, because it is essential in order to perform the MIPS instructions BNE, BEQ and J. The Comparator component is simple, it takes two inputs dataa and datab and then compares then outputs the comparison of the two integers. This comparison is very important because if in MIPS a BEQ or BNE instruction is called then you would have to do a comparison if the two integers are either equal or not equal and then branch to the instruction depending on the comparison. Therefore the Comparator component is an essential part of the circuit because it helps the circuit perform comparisons.

Mux2:1

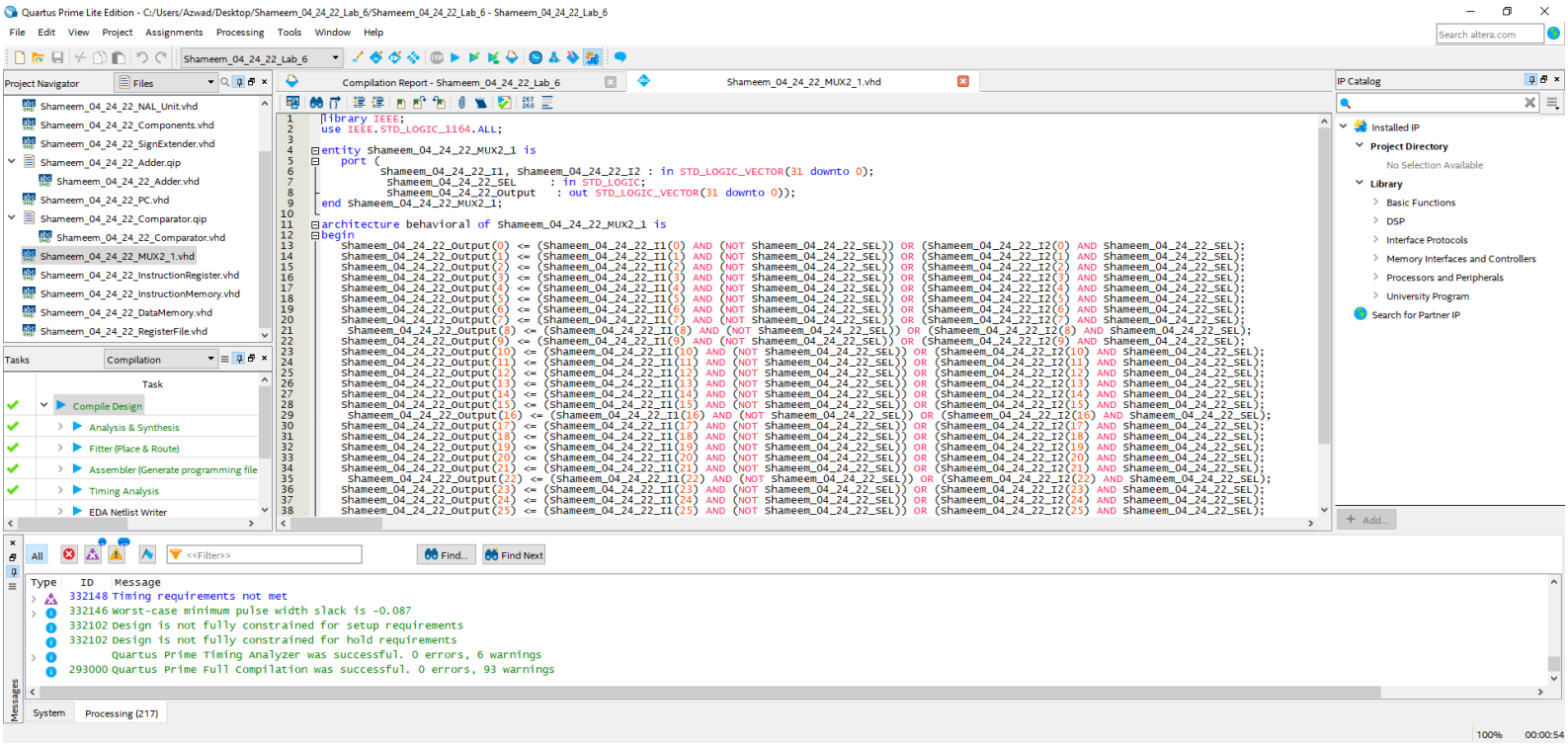


Figure 14: Mux 2:1 VHDL code compiled successfully in Quartus.

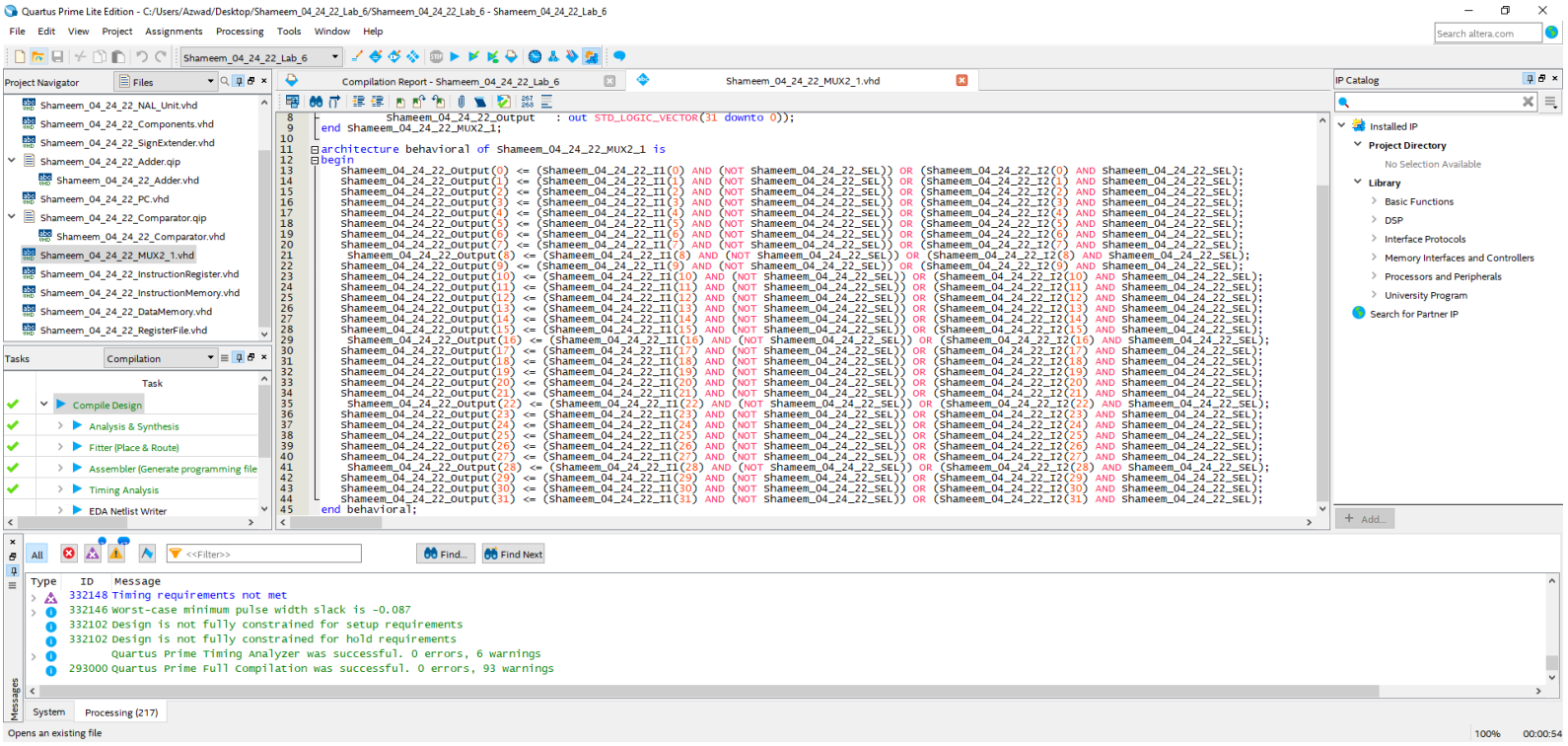


Figure 15: Mux 2:1 VHDL code continued.

Data Memory

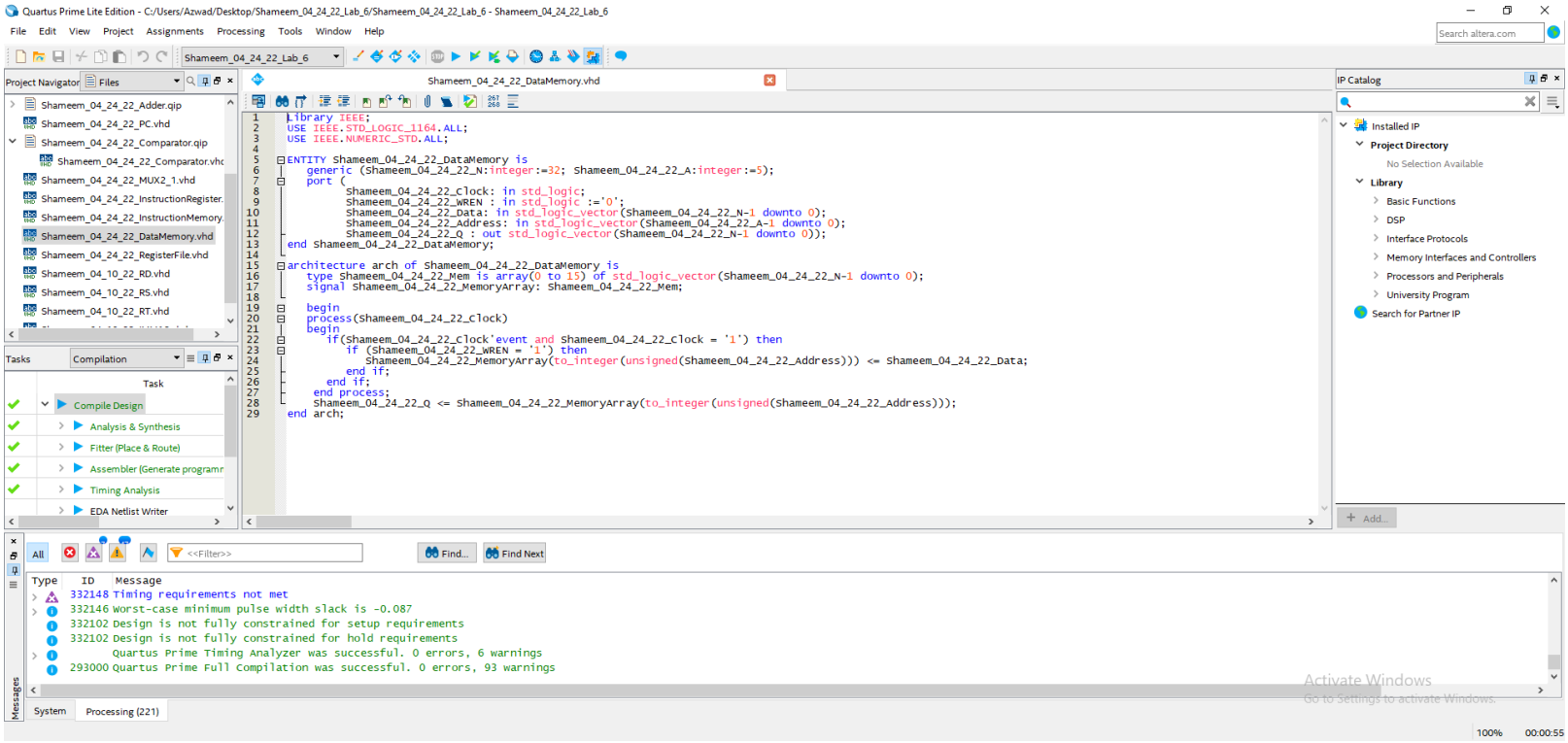


Figure 16: Data Memory VHDL code compiled successfully in Quartus.

Register File

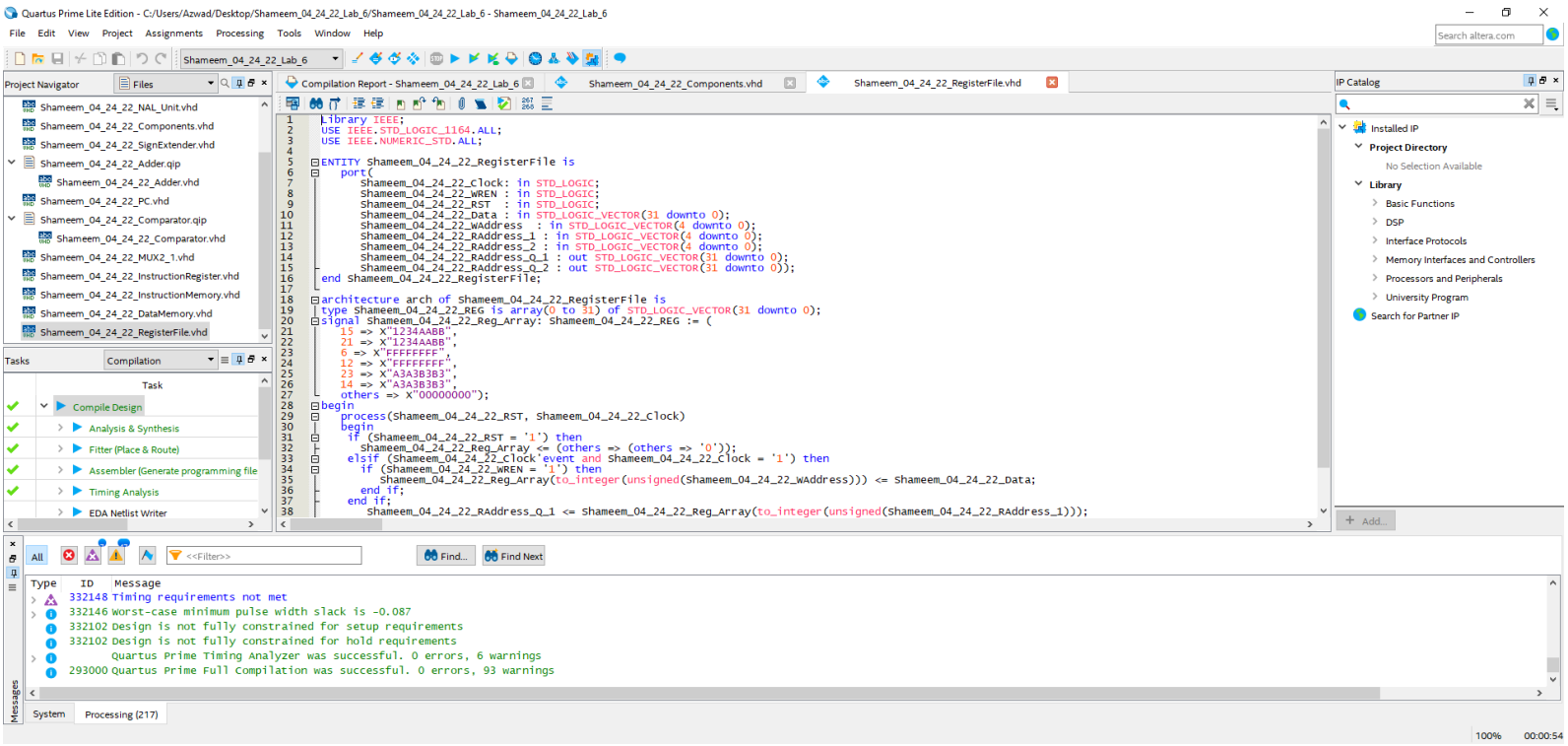


Figure 17: Register File VHDL code compiled successfully in Quartus.

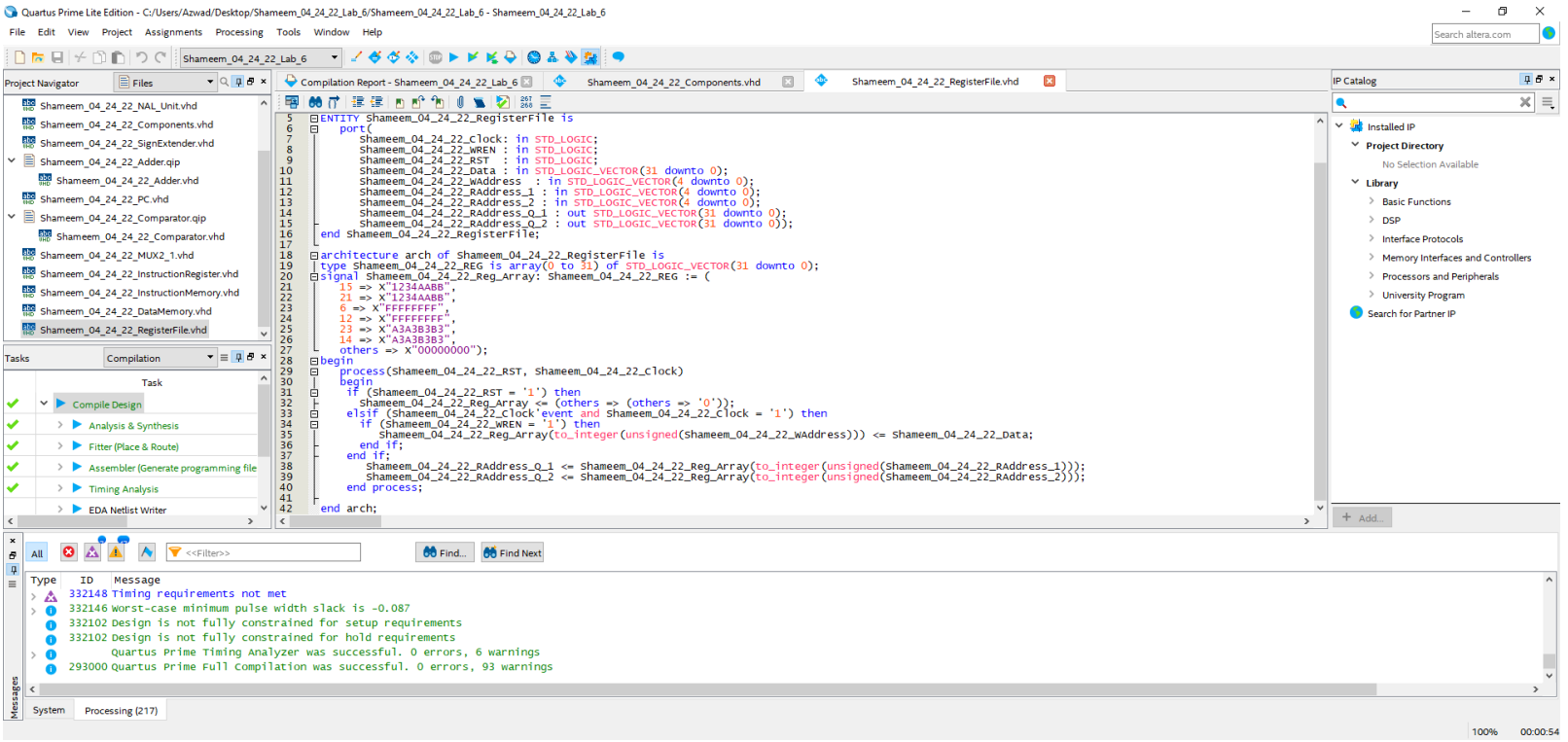


Figure 18: Register File VHDL code continued.

Instruction Register

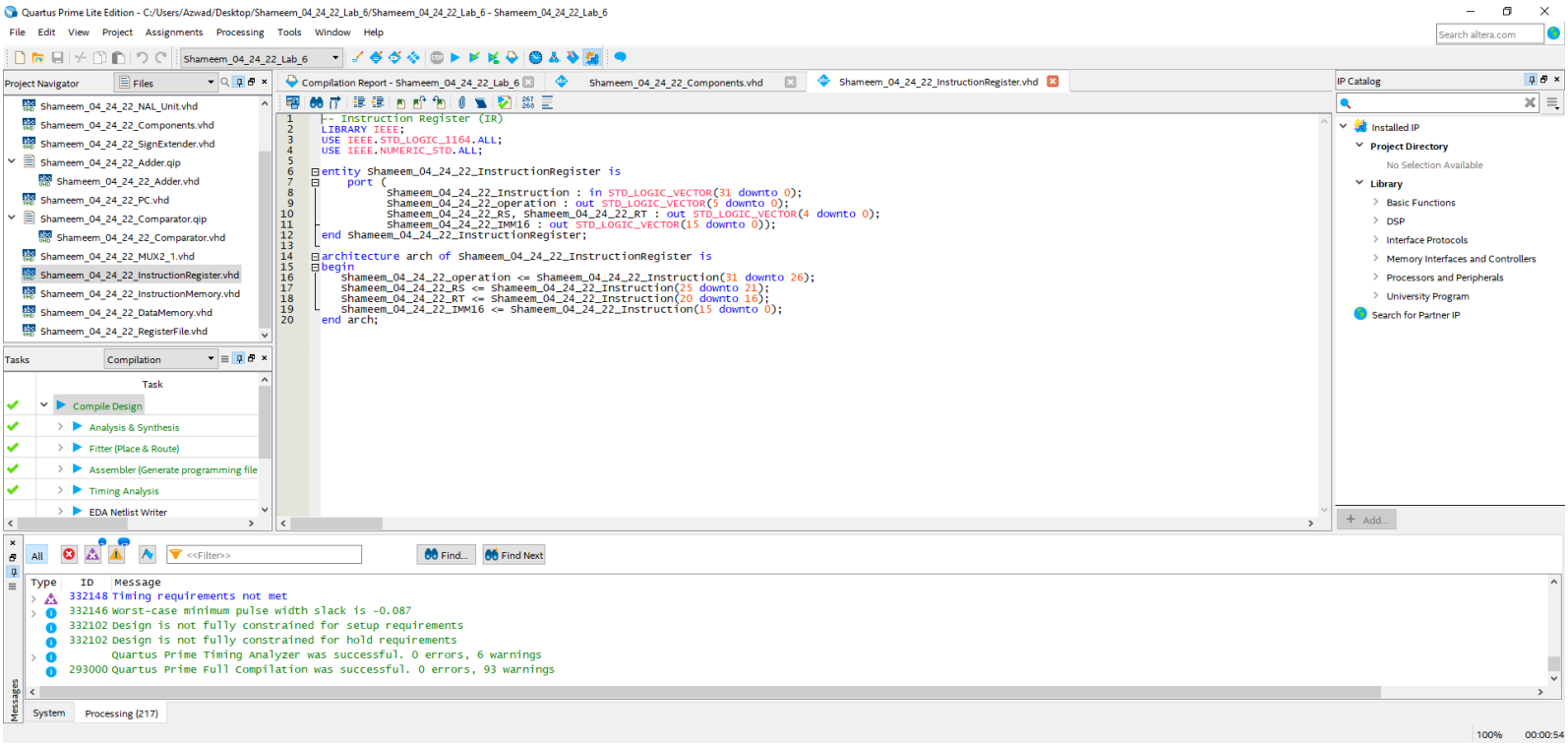
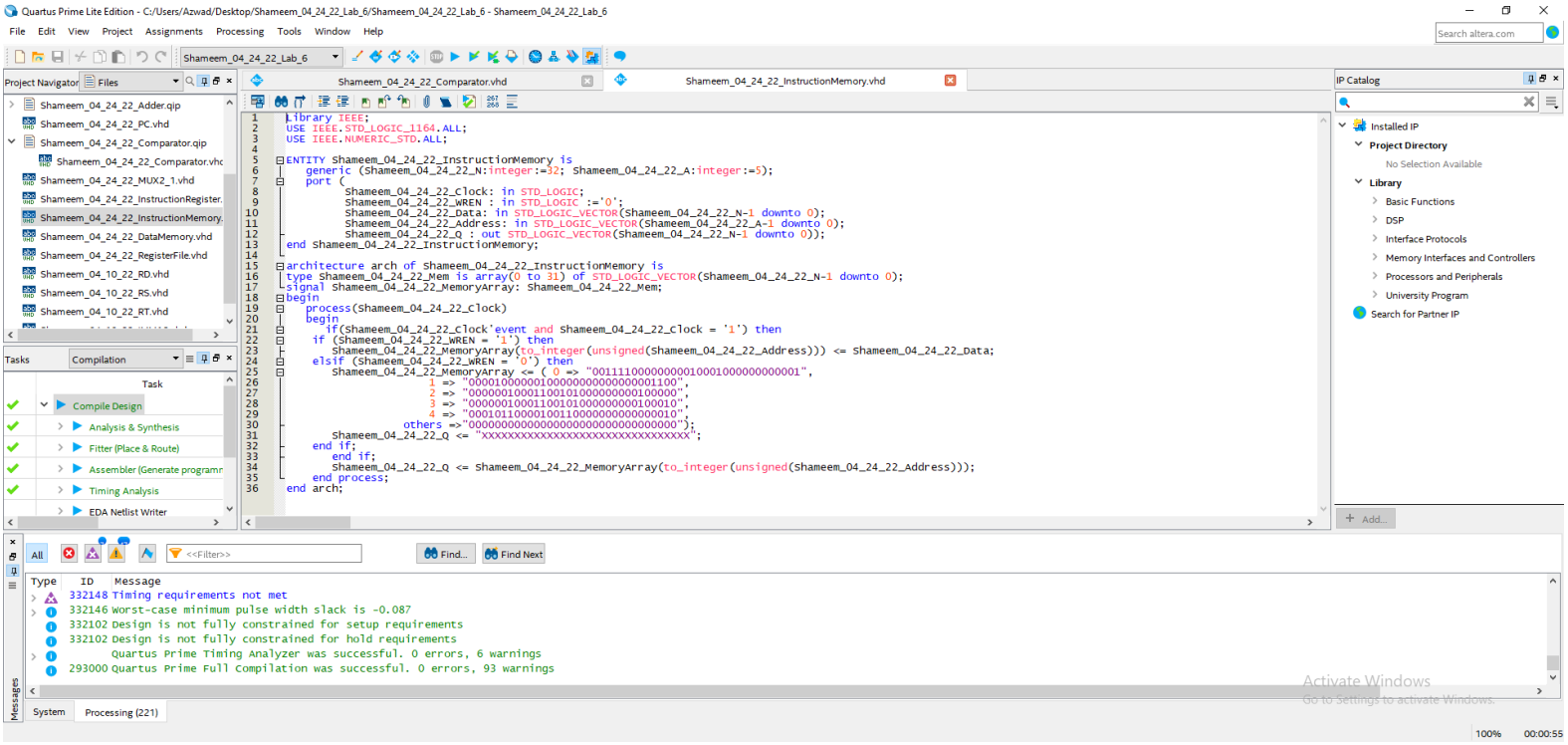


Figure 19: Instruction Register VHDL code compiled successfully in Quartus.

Instruction Memory




```
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Shameem_04_24_22_InstructionMemory is
    generic (Shameem_04_24_22_N:integer:=32; Shameem_04_24_22_A:integer:=5);
    port (
          Shameem_04_24_22_Clock: in STD_LOGIC;
          Shameem_04_24_22_WREN : in STD_LOGIC :='0';
          Shameem_04_24_22_Data: in STD_LOGIC_VECTOR(Shameem_04_24_22_N-1 downto 0);
          Shameem_04_24_22_Address: in STD_LOGIC_VECTOR(Shameem_04_24_22_A-1 downto 0);
          Shameem_04_24_22_Q : out STD_LOGIC_VECTOR(Shameem_04_24_22_N-1 downto 0));
end Shameem_04_24_22_InstructionMemory;

architecture arch of Shameem_04_24_22_InstructionMemory is
type Shameem_04_24_22_Mem is array(0 to 31) of STD_LOGIC_VECTOR(Shameem_04_24_22_N-1 downto 0);
signal Shameem_04_24_22_MemoryArray: Shameem_04_24_22_Mem;
begin
    process(Shameem_04_24_22_Clock)
    begin
      if(Shameem_04_24_22_Clock'event and Shameem_04_24_22_clock = '1') then
     if (Shameem_04_24_22_WREN = '1') then
        Shameem_04_24_22_MemoryArray(to_integer(unsigned(Shameem_04_24_22_Address))) <= Shameem_04_24_22_Data;
     elsif (Shameem_04_24_22_WREN = '0') then
        Shameem_04_24_22_MemoryArray <= ( 0 => "00111100000000010001000000000001",
                          1 => "00001000000100000000000000001100",
                          2 => "00000010001100101000000000100000",
                          3 => "00000010001100101000000000100010",
                          4 => "00010110000100110000000000000010",
                      others =>"00000000000000000000000000000000");
        Shameem_04_24_22_Q <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
     end if;
        end if;
        Shameem_04_24_22_Q <= Shameem_04_24_22_MemoryArray(to_integer(unsigned(Shameem_04_24_22_Address)));
    end process;
end arch;
```

Figure 20: Instruction Memory VHDL code compiled successfully in Quartus.

NAL Unit

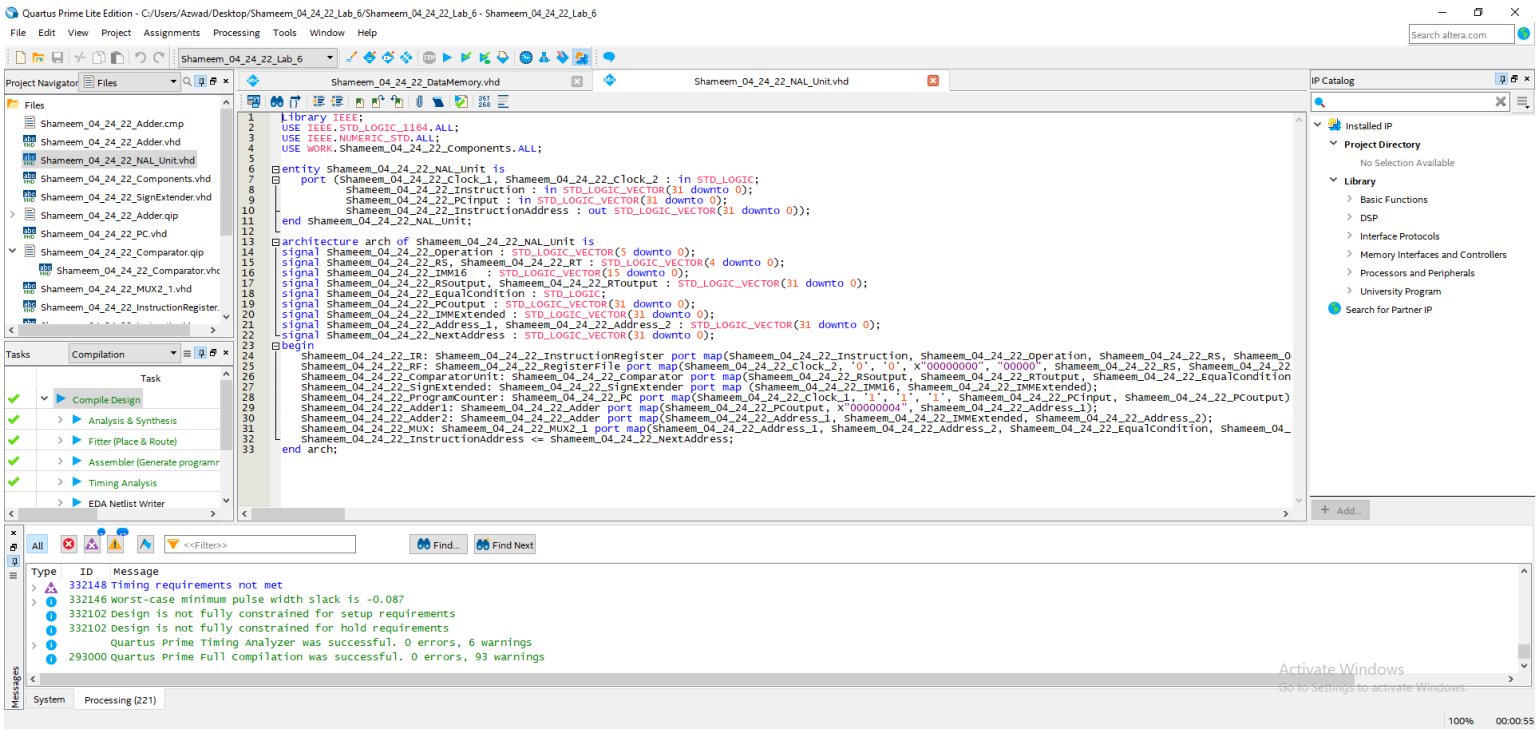


Figure 21: NAL Unit VHDL code compiled successfully in Quartus.

Simulation:

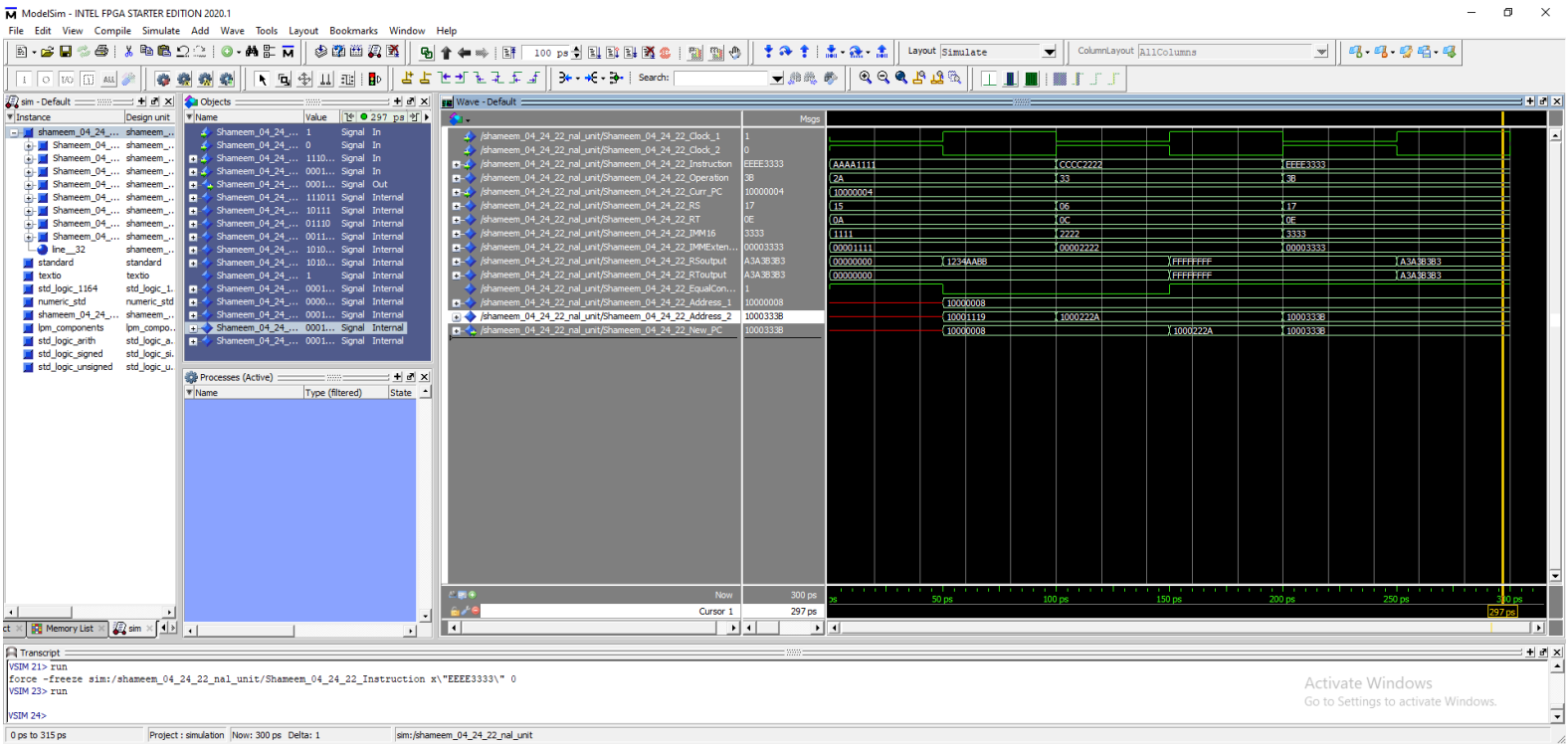


Figure 22: BEQ Simulation Waveforms (Radix: Hexadecimal)

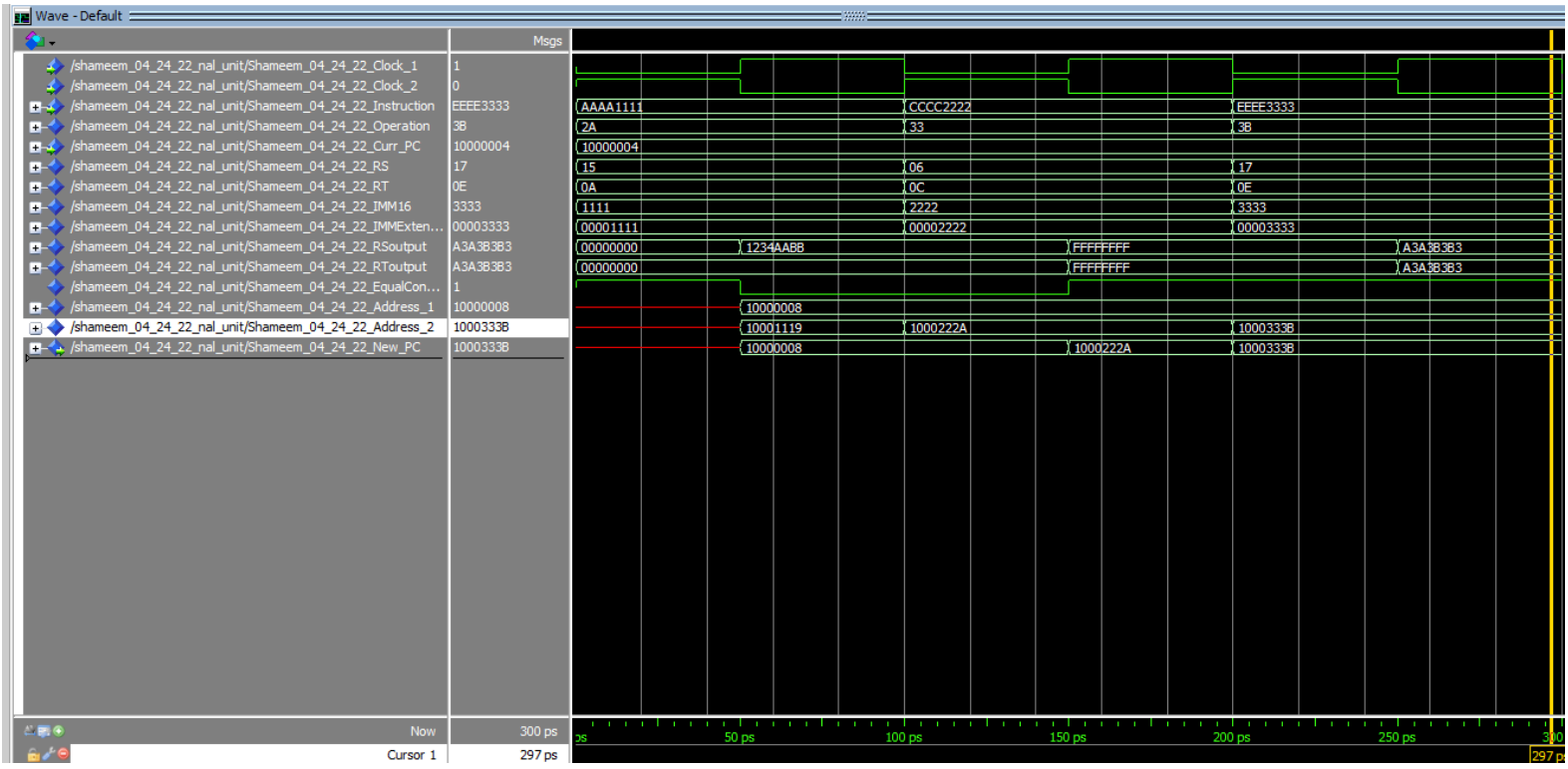


Figure 23: BEQ Simulation Waveforms Zoomed In (Radix: Hexadecimal)

The BEQ instruction is a branching instruction that jumps or branches to the specified label instruction, if the contents of the two stored numbers are equal. In this simulation BEQ will jump or branch if the RS and RT register values are equal. Furthermore, the address of the next instruction will be concluded at that time with the following logic in the image below.

```
Equal <= (R[rs] == R[rt])      Calculate the branch condition
if (Equal)                     Calculate the next instruction's address
        PC  <=  PC + 4 + { SignExt(imm16) , 2b00 }
Else  PC  <=  PC + 4
```

Figure 25: Logic shown in programming code

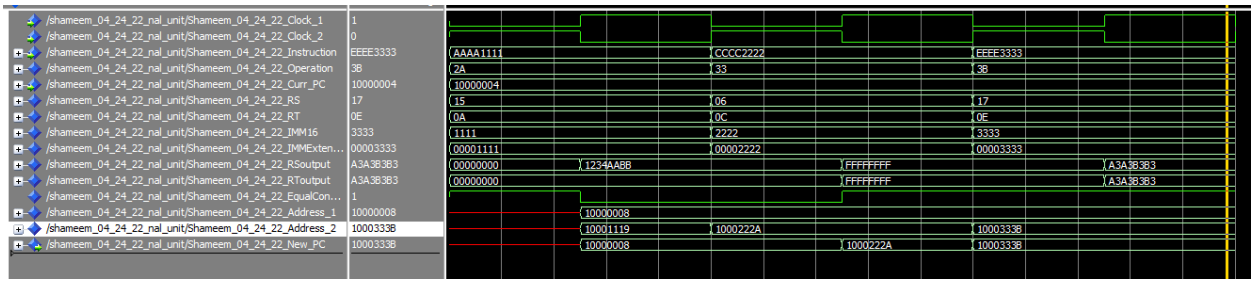


Figure 26: BEQ Simulation Waveforms Zoomed (Radix: Hexadecimal)

To verify the correctness of the circuit, we will use three different instructions in which there will be different RS, RT, IMM16 addresses and values. We can see the value of RS and RT in RSoutput signal and RToutput signal and the value of the IMM16 extended in the IMMExtended signal. Furthermore, in this simulation we are using BEQ so if RS and RT are equal the EqualCond is 1, otherwise if they are not equal it is 0. Lastly, it notable that the new address of the operation is at the output New_PC.

In 0 to 100 ps, the 32-bit instruction AAAA1111 is inputted into the circuit. This 32-bit instruction goes into the InstructionRegister component where the instruction gets split and put into registers, RS, RT, IMM16 which then gives back the value of the registers at the instruction. In 0 to 100 ps we see that RSoutput and RToutput are not equal, so by following the logic shown in figure 25, we set the New_PC to be equal to the curr_pc + 4. Therefore, in 50 to 100 ps we see the new_PC value be Curr_PC + 4 which is 10000004 + 4 which equals 10000008, which is correctly shown in the simulation

In 100 ps to 200 ps, the 32-bit instruction CCCC2222 is inputted into the circuit. This 32-bit instruction goes into the InstructionRegister component where the instruction gets split and put into registers, RS, RT, IMM16 which then gives back the value of the registers at the instruction. At 100 ps to 200 ps we see that RSoutput and RToutput are both equal and have the value FFFFFFFF. Therefore, since RS and RT are equal we follow the logic shown in figure 25, that the New_PC is set to Curr_PC + SignExt(Imm16) + 4. As a result, at 150 ps to 200 ps, the New_PC is set to 1000222A, which is the result of adding 10000004 + 00002222 + 4.

In 200 ps to 300 ps, the 32-bit Instruction EEEE3333 is inputted into the circuit. This 32-bit instruction goes into the InstructionRegister component where the instruction gets split and put into registers, RS, RT, IMM16 which then gives back the value of the registers at the instruction. At 200 to 300 ps, RS and RT are equal with the value of A3A3B3B3 and IMM16 is 3333. Therefore, since RS and RT are equal we follow the logic shown in figure 25, that the New_PC is set to Curr_PC + SignExt(Imm16) + 4. As a result, at 250 ps to 300 ps, the New_PC is set to 1000333B because when we follow the logic of the equation, we get 10000004 + 00003333 + 4 = 1000333B.

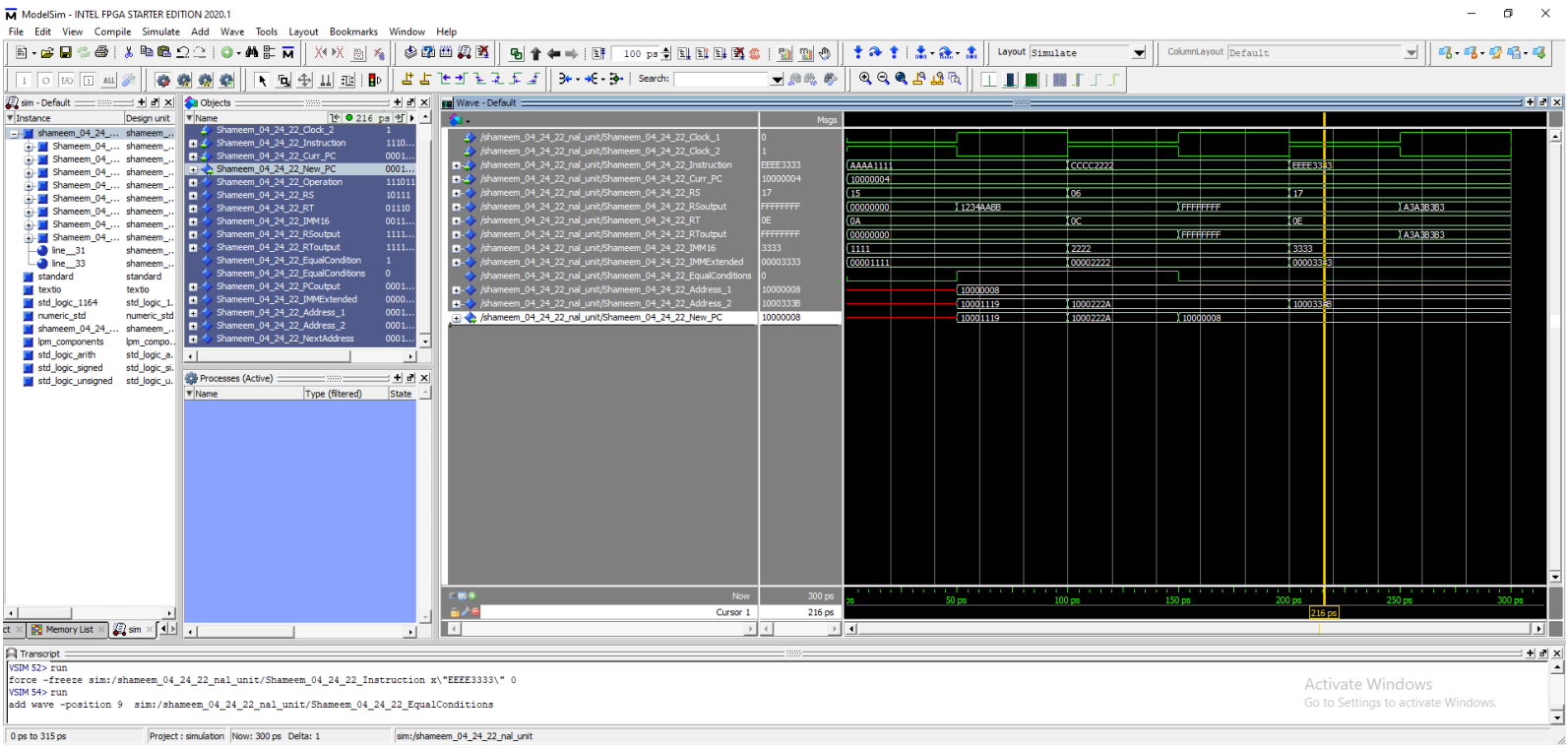


Figure 27: BNE Simulation Waveforms (Radix: Hexadecimal)

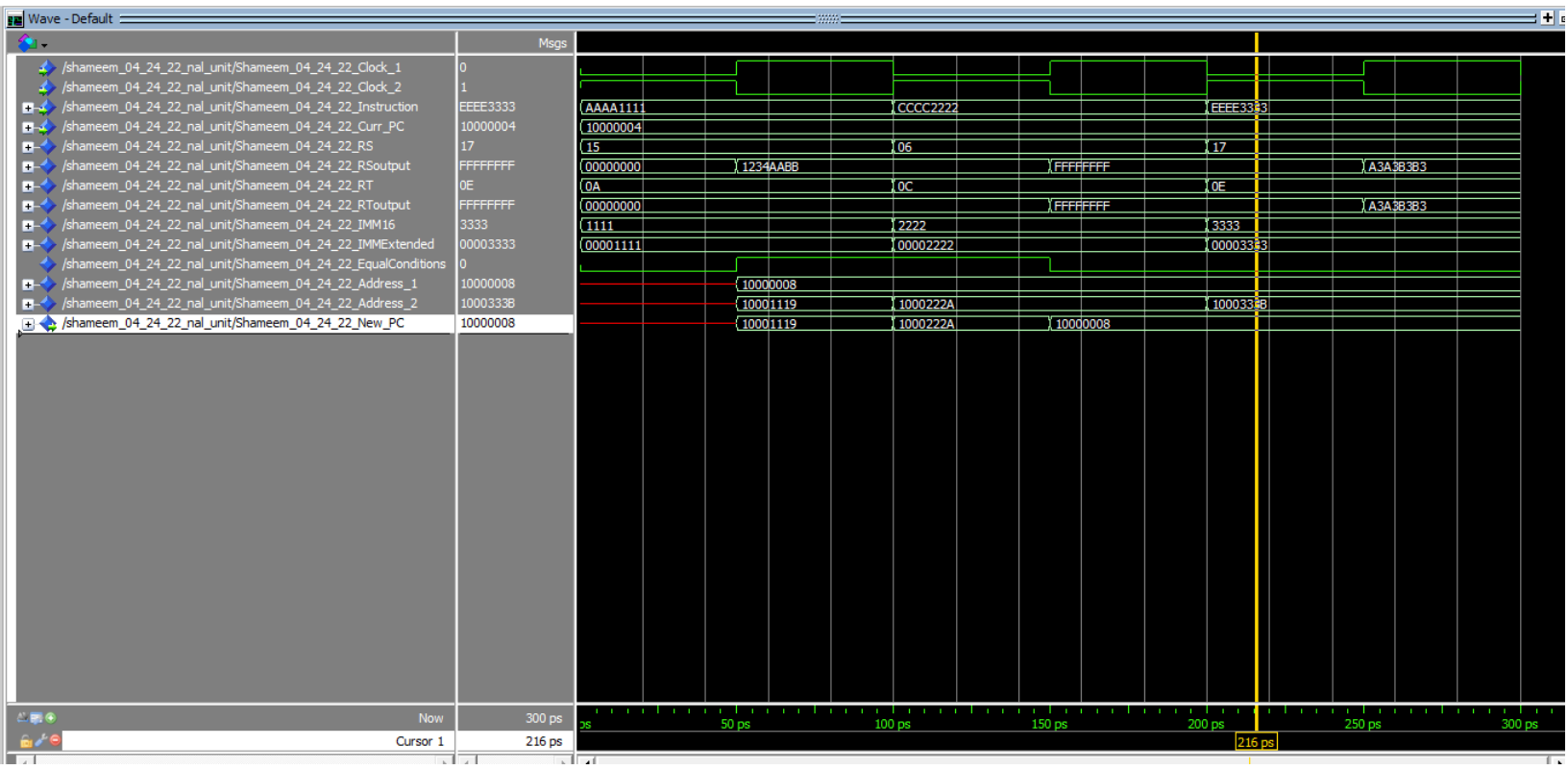


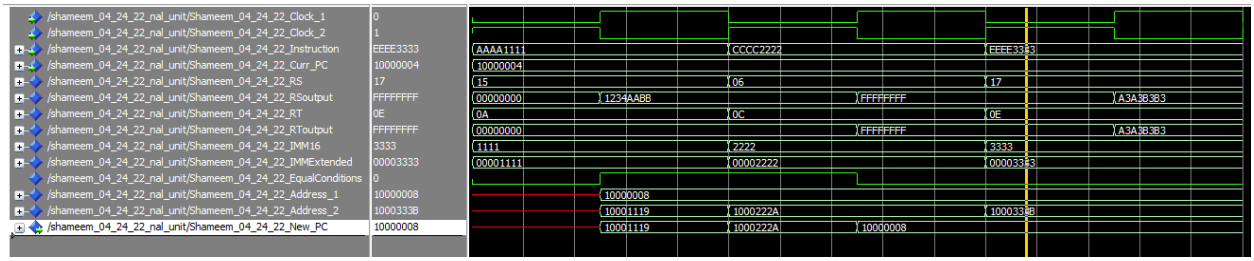
Figure 28: BNE Simulation Waveforms (Radix: Hexadecimal

The BNE instruction is a branching instruction that jumps or branches to the specified label instruction, if the contents of the two stored numbers are not equal. In this simulation BNE will jump or branch if the RS and RT register values are not equal. Furthermore, the address of the next instruction will be concluded at that time with the following logic for BEQ.

```
Equal <= (R[rs] == R[rt])      Calculate the branch condition
if (Equal)                     Calculate the next instruction's address
        PC  <=  PC + 4 + { SignExt(imm16) , 2b00 }
Else  PC  <=  PC + 4
```

Figure 29: BEQ Logic

Fortunately, BNE logic is pretty much the same as the BEQ logic with the exception of where it says if (Equal), the logic would say if (Not Equal).



Figer 30: BNE Simulation Waveforms Zoomed in

To verify the correctness of the circuit, we will use three different instructions in which there will be different RS, RT, IMM16 addresses and values. We can see the value of RS and RT in RSoutput signal and RToutput signal and the value of the IMM16 extended in the IMMExtended signal. Furthermore, in this simulation we are using BEQ so if RS and RT are **not** equal the EqualCond is 1, otherwise if they are equal, it is 0. Lastly, it notable that the new address of the operation is at the output New_PC.

In 0 to 100 ps, the 32-bit instruction AAAA1111 is inputted into the circuit. This 32-bit instruction goes into the InstructionRegister component where the instruction gets split and put into registers, RS, RT, IMM16 which then gives back the value of the registers at the instruction. In 0 to 100 ps we see that RSoutput and RToutput are not equal, so by following the logic shown in figure 25, we set the New_PC to be equal to the curr_pc + SignExt(Imm16) + 4. As a result, at 0 ps to 100 ps, the New_PC is set to 10001119, which is the result of adding 10000004 + 00001111 + 4, which is 10000008 + 00001111 = 10001119.

In 100 ps to 200 ps, the 32-bit instruction CCCC2222 is inputted into the circuit. This 32-bit instruction goes into the InstructionRegister component where the instruction gets split and put into registers, RS, RT, IMM16 which then gives back the value of the registers at the instruction. At 100 ps to 200 ps we see that RSoutput and RToutput are both equal and have the value FFFFFFFF. Therefore, since RS and RT are equal, we follow the logic shown in figure 25, that the New_PC is set to Curr_PC + 4. As a result, at 150 ps to 200 ps, the New_PC is set to 10000008 because we have the equation when equal for BNE which is Curr_PC + 4, which equals 10000004 + 4 = 10000008.

In 200 ps to 300 ps, the 32-bit Instruction EEEE3333 is inputted into the circuit. This 32-bit instruction goes into the InstructionRegister component where the instruction gets split and put into registers, RS, RT, IMM16 which then gives back the value of the registers at the instruction. At 200 to 300 ps, RS and RT are equal with the value of A3A3B3B3 and IMM16 is 3333. Therefore, since RS and RT are equal, we follow the logic shown in figure 25, that the New_PC is set to Curr_PC + 4. As a result, at 150 ps to 200 ps, the New_PC is set to 10000008 because we have the equation when equal for BNE which is Curr_PC + 4, which equals 10000004 + 4 = 10000008. This is the same PC as the 100 ps to 200 ps because both of the scenarios have equal RS and RT values.

Conclusions

In conclusion this lab is to utilize the knowledge from the previous lab to implement MIPS instructions BEQ, BNE and j. Therefore, by implementing the MIPS instructions BEQ, BNE and j, we learn the usage of multiplexers, comparators and the adder components in order to properly test the usage of BEQ, BNE and j MIPS instructions. In addition to learning how to implement the MIPS instructions, we learn how to create a Next Address Logic Unit to output the next address of the instruction whether or not a jump or branching occurs, which will be shown in the PC or otherwise known as the Program Counter. This lab is very beneficial for understanding how MIPS instructions work and how to implement them in VHDL about their intricacies.