CS 342000 | CS343000 - Spring 2022
Instructor: Professor Izidor Gertner
Spring 2022 – 5/22/22
Single Cycle CPU - Azwad Shameem

Table of Contents

Objective

The objective of the Single Cycle CPU lab was to write a program to compute the sum of five integers. Furthermore, we had to prove the correctness of the simulation using MIPS instructions in MARS Simulator. In order to simulate MIPS instructions in VHDL we needed to create data memory, instruction memory, 3-ported register file, IR-Instruction Register, Program Counter, AdderSub, Extender, and multiplexers components. Not to mention it was also imperative to implement a control unit that generated the control signals and regulated what action should be taken. Therefore, by completing the previous steps it became possible to simulate the cycle that an instruction takes when executing in the CPU such as reading the instruction from memory, breaking it down to specific parts and then executing the instruction and also determining the address of PC.

Components

Package

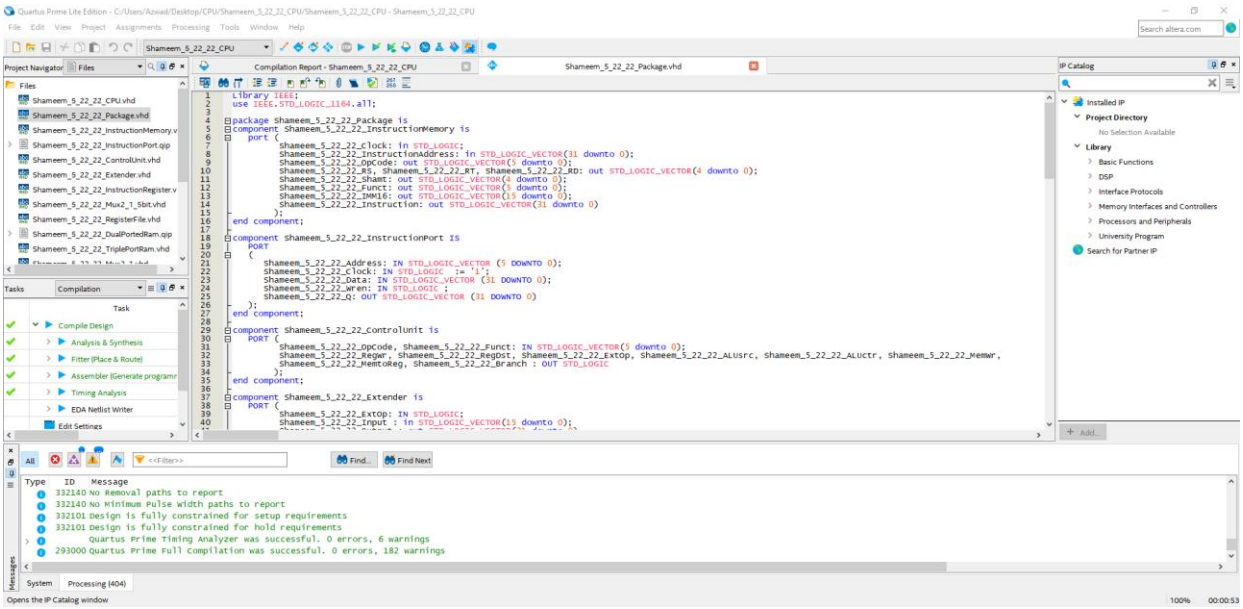


Figure 1: Package VHDL code. This package contains all of the components used.

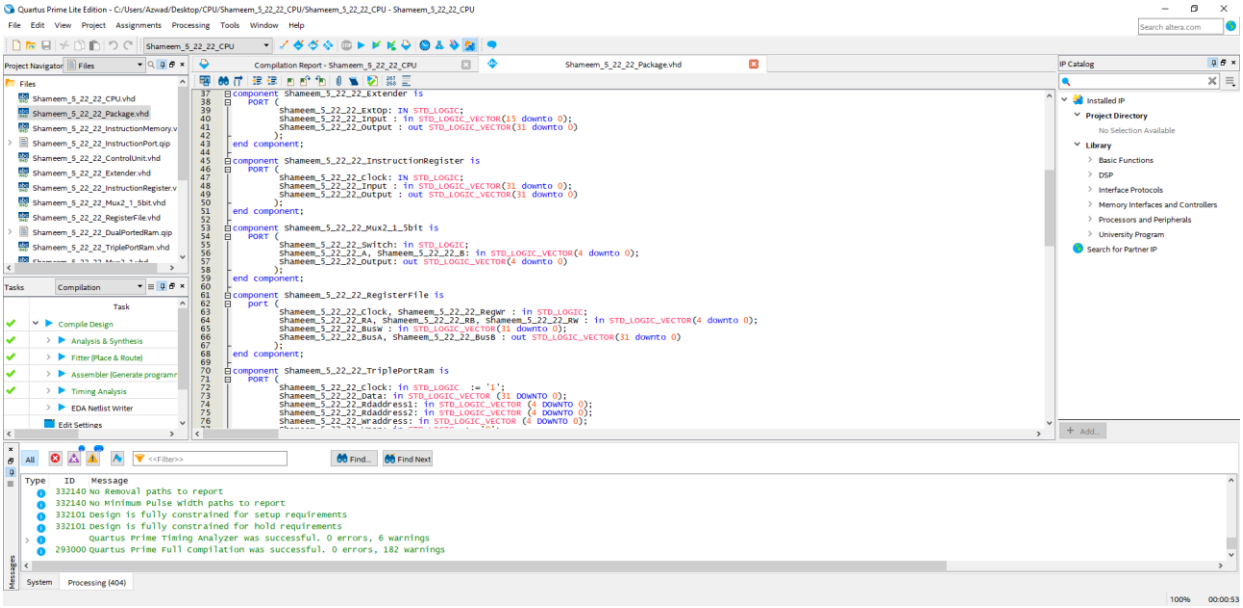


Figure 2: Package VHDL code continued.

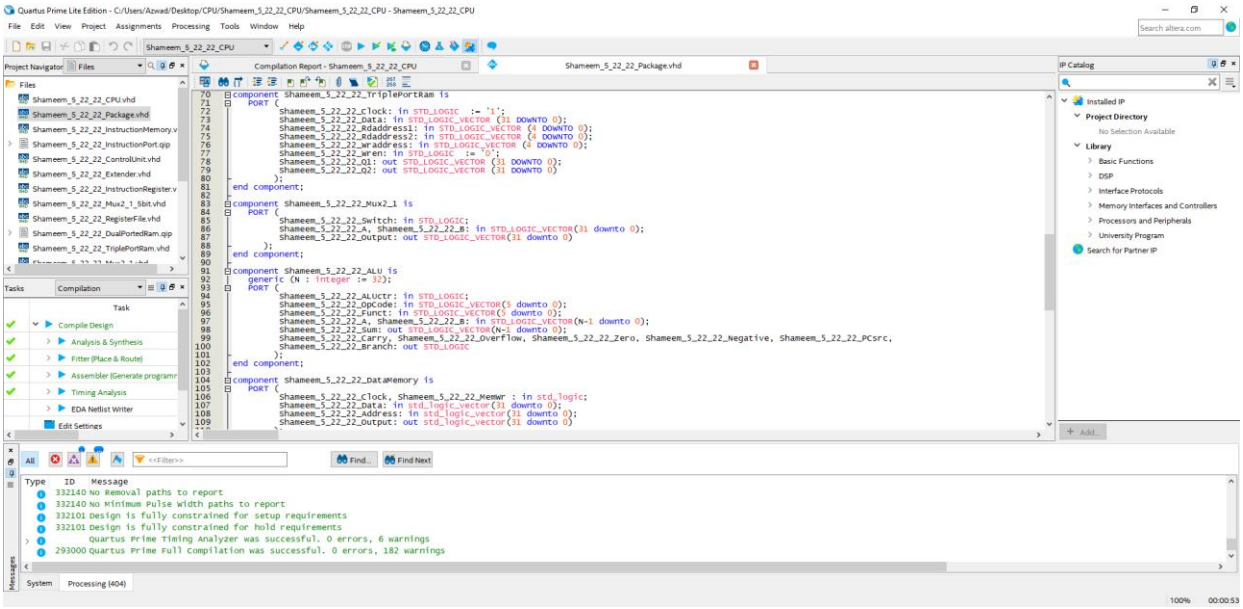


Figure 3: Package VHDL code continued.

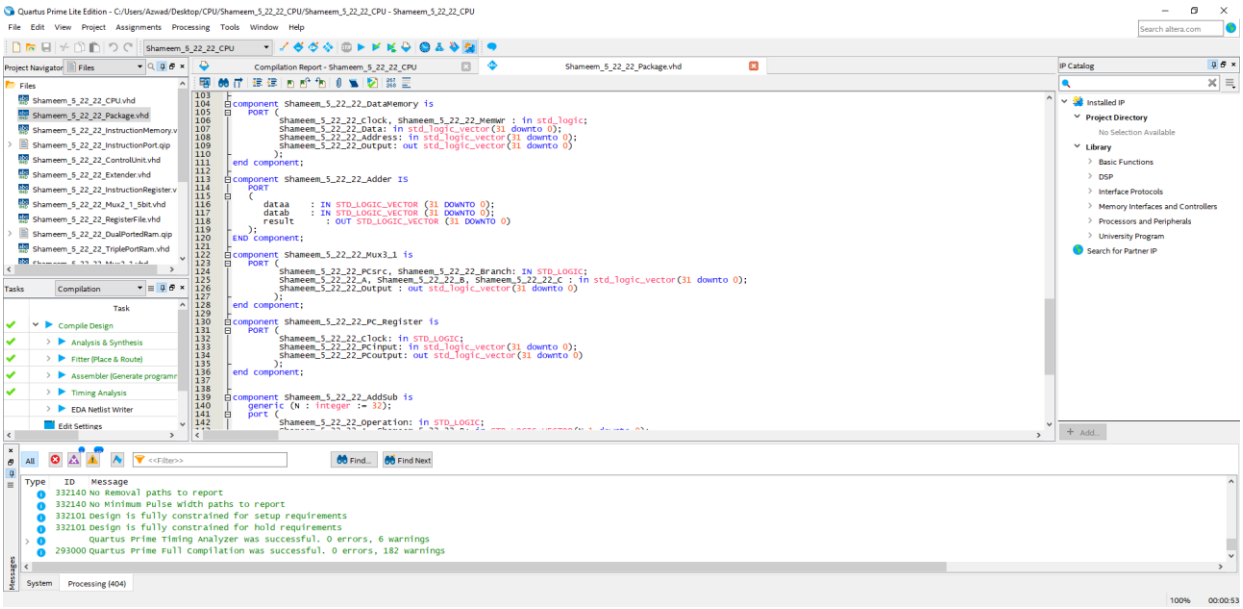


Figure 4: Package VHDL code continued.

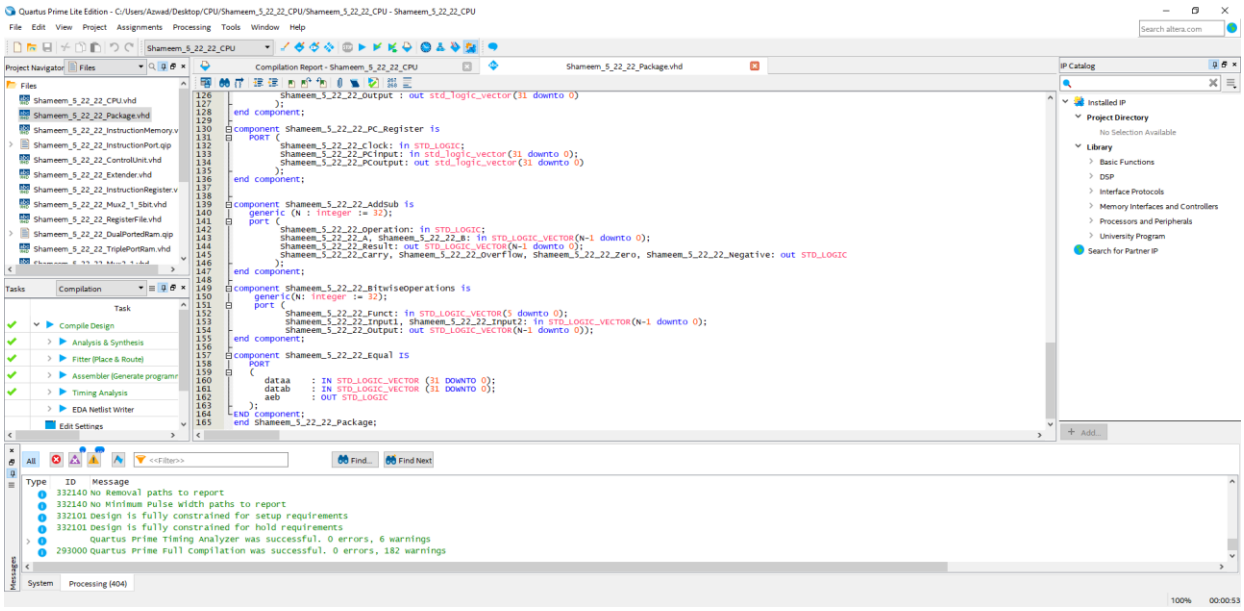


Figure 5: Package VHDL code continued.

Instruction Memory

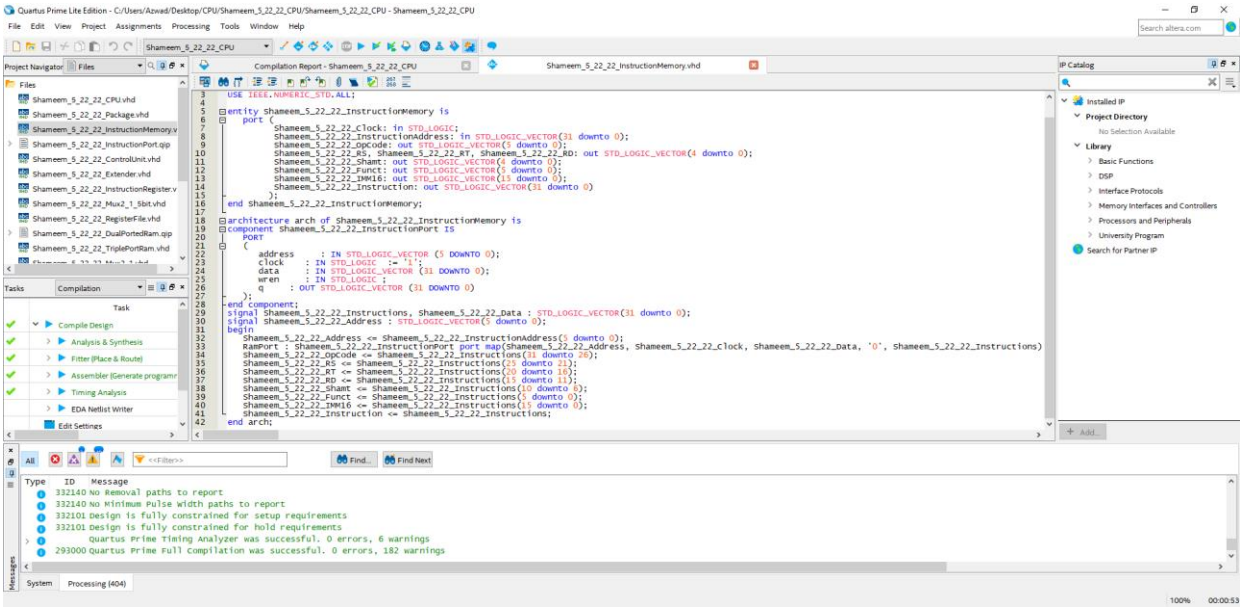


Figure 6: Instruction Memory VHDL code.

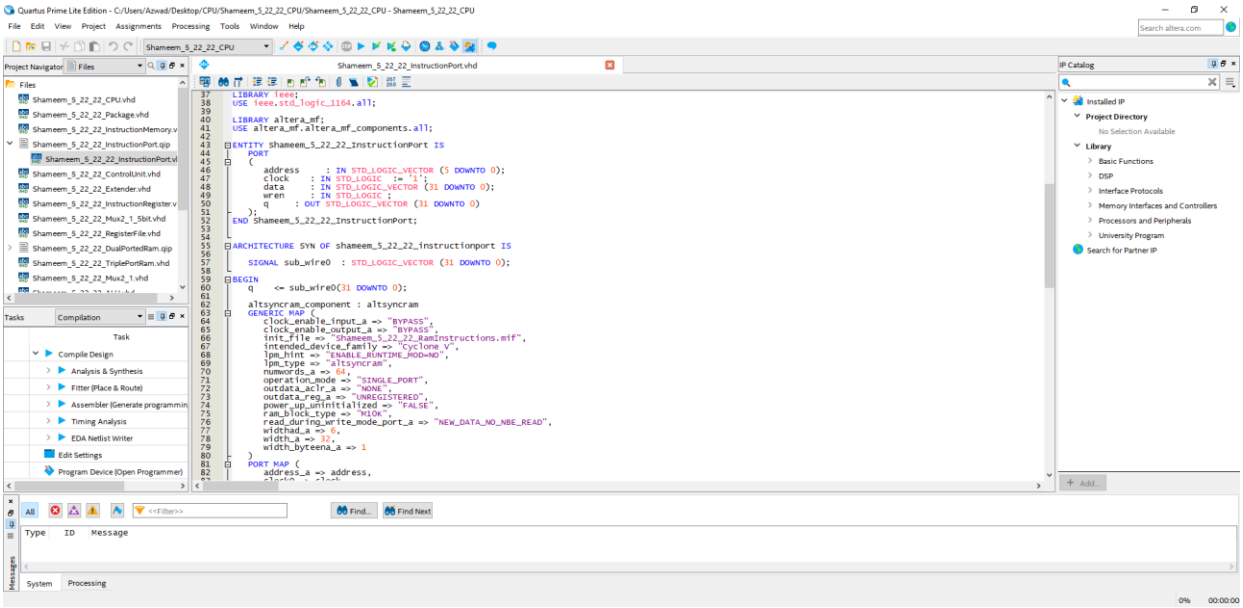
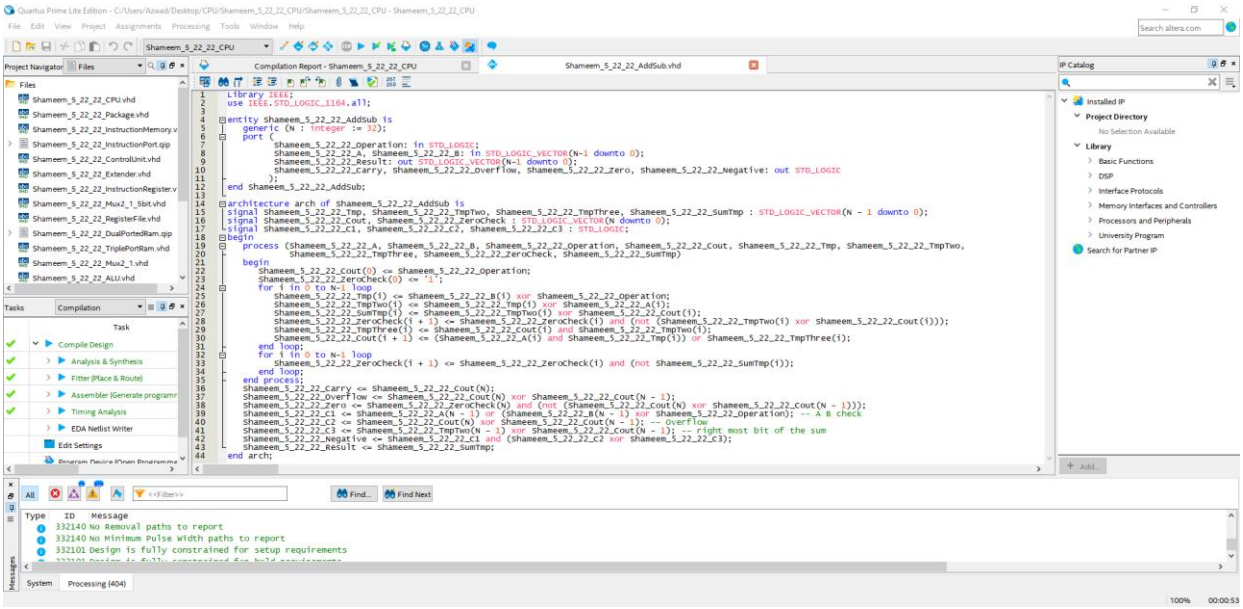


Figure 6: Instruction Memory Port VHDL code.

AdderSub



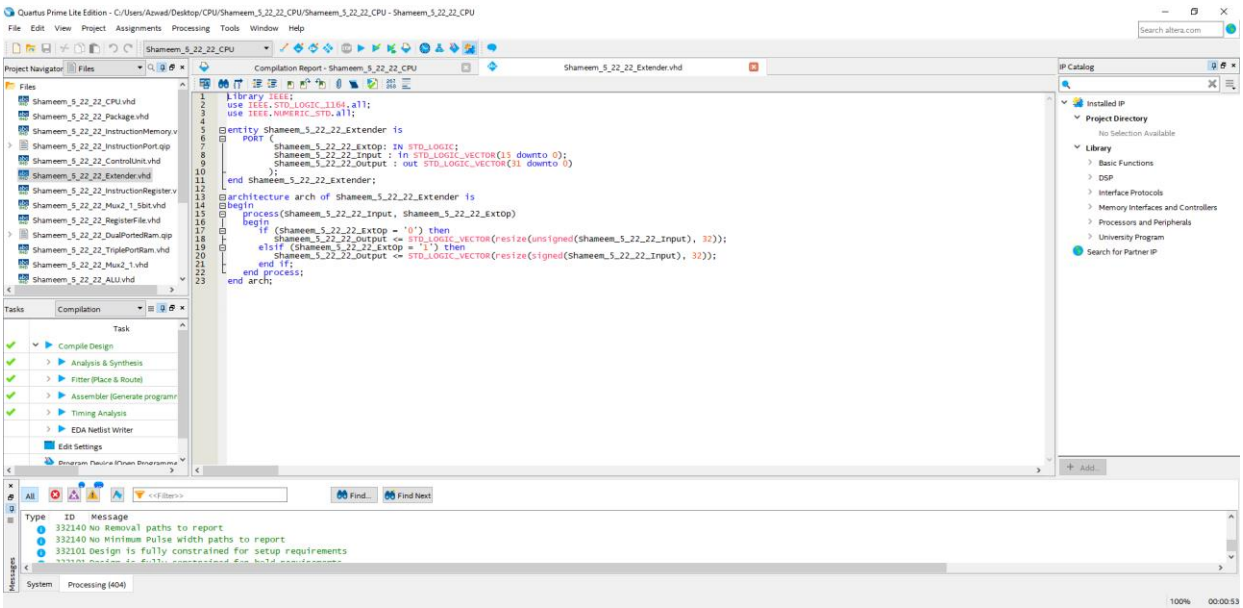
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Shameem_5_22_22_AddSub is
    generic (N : integer := 32);
    port (
          Shameem_5_22_22_operation: in STD_LOGIC;
          Shameem_5_22_22_A, Shameem_5_22_22_B: in STD_LOGIC_VECTOR(N-1 downto 0);
          Shameem_5_22_22_Result: out STD_LOGIC_VECTOR(N-1 downto 0);
          Shameem_5_22_22_Carry, Shameem_5_22_22_Overflow, Shameem_5_22_22_Zero, Shameem_5_22_22_Negative: out STD_LOGIC
         );
end Shameem_5_22_22_AddSub;

architecture arch of Shameem_5_22_22_AddSub is
 signal Shameem_5_22_22_Tmp, Shameem_5_22_22_TmpTwo, Shameem_5_22_22_TmpThree, Shameem_5_22_22_SumTmp : STD_LOGIC_VECTOR(N - 1 downto 0);
 signal Shameem_5_22_22_Cout, Shameem_5_22_22_ZeroCheck : STD_LOGIC_VECTOR(N downto 0);
 signal Shameem_5_22_22_C1, Shameem_5_22_22_C2, Shameem_5_22_22_C3 : STD_LOGIC;
begin
    process (Shameem_5_22_22_A, Shameem_5_22_22_B, Shameem_5_22_22_Operation, Shameem_5_22_22_Cout, Shameem_5_22_22_Tmp, Shameem_5_22_22_TmpTwo,
             Shameem_5_22_22_TmpThree, Shameem_5_22_22_ZeroCheck, Shameem_5_22_22_SumTmp)
    begin
        Shameem_5_22_22_Cout(0) <= Shameem_5_22_22_Operation;
        Shameem_5_22_22_ZeroCheck(0) <= '1';
        for i in 0 to N-1 loop
            Shameem_5_22_22_Tmp(i) <= Shameem_5_22_22_B(i) xor Shameem_5_22_22_Operation;
            Shameem_5_22_22_TmpTwo(i) <= Shameem_5_22_22_Tmp(i) xor Shameem_5_22_22_A(i);
            Shameem_5_22_22_SumTmp(i) <= Shameem_5_22_22_TmpTwo(i) xor Shameem_5_22_22_Cout(i);
            Shameem_5_22_22_ZeroCheck(i + 1) <= Shameem_5_22_22_ZeroCheck(i) and (not (Shameem_5_22_22_TmpTwo(i) xor Shameem_5_22_22_Cout(i)));
            Shameem_5_22_22_TmpThree(i) <= Shameem_5_22_22_Cout(i) and Shameem_5_22_22_TmpTwo(i);
            Shameem_5_22_22_Cout(i + 1) <= (Shameem_5_22_22_A(i) and Shameem_5_22_22_Tmp(i)) or Shameem_5_22_22_TmpThree(i);
        end loop;
        for i in 0 to N-1 loop
            Shameem_5_22_22_ZeroCheck(i + 1) <= Shameem_5_22_22_ZeroCheck(i) and (not Shameem_5_22_22_SumTmp(i));
        end loop;
    end process;
    Shameem_5_22_22_Carry <= Shameem_5_22_22_Cout(N);
    Shameem_5_22_22_Overflow <= Shameem_5_22_22_Cout(N) xor Shameem_5_22_22_Cout(N - 1);
    Shameem_5_22_22_Zero <= Shameem_5_22_22_ZeroCheck(N) and (not (Shameem_5_22_22_Cout(N) xor Shameem_5_22_22_Cout(N - 1)));
    Shameem_5_22_22_C1 <= Shameem_5_22_22_A(N - 1) or (Shameem_5_22_22_B(N - 1) xor Shameem_5_22_22_Operation); -- A B check
    Shameem_5_22_22_C2 <= Shameem_5_22_22_Cout(N) xor Shameem_5_22_22_Cout(N - 1); -- Overflow
    Shameem_5_22_22_C3 <= Shameem_5_22_22_TmpTwo(N - 1) xor Shameem_5_22_22_Cout(N - 1); -- right most bit of the sum
    Shameem_5_22_22_Negative <= Shameem_5_22_22_C1 and (Shameem_5_22_22_C2 xor Shameem_5_22_22_C3);
    Shameem_5_22_22_Result <= Shameem_5_22_22_SumTmp;
end arch;
```

Figure 7: AdderSub VHDL code. Used to compute instructions Add, Addi, Addu, Addiu, Sub, Subu.

Extender



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

entity Shameem_5_22_22_Extender is
    PORT (
          Shameem_5_22_22_ExtOp: IN STD_LOGIC;
          Shameem_5_22_22_Input : in STD_LOGIC_VECTOR(15 downto 0);
          Shameem_5_22_22_Output : out STD_LOGIC_VECTOR(31 downto 0)
         );
end Shameem_5_22_22_Extender;

architecture arch of Shameem_5_22_22_Extender is
begin
    process(Shameem_5_22_22_Input, Shameem_5_22_22_ExtOp)
    begin
        if (Shameem_5_22_22_ExtOp = '0') then
            Shameem_5_22_22_Output <= STD_LOGIC_VECTOR(resize(unsigned(Shameem_5_22_22_Input), 32));
        elsif (Shameem_5_22_22_ExtOp = '1') then
            Shameem_5_22_22_Output <= STD_LOGIC_VECTOR(resize(signed(Shameem_5_22_22_Input), 32));
        end if;
    end process;
end arch;
```

Figure 7: Extender VHDL code.

Mux

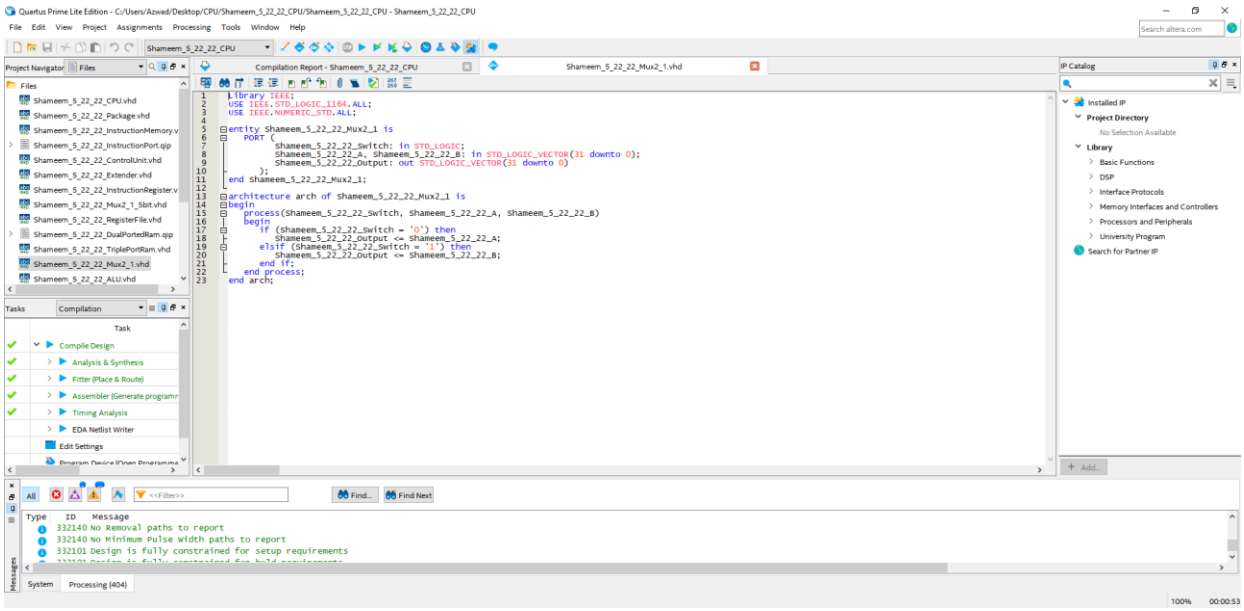


Figure 8: Mux 2:1 VHDL code.

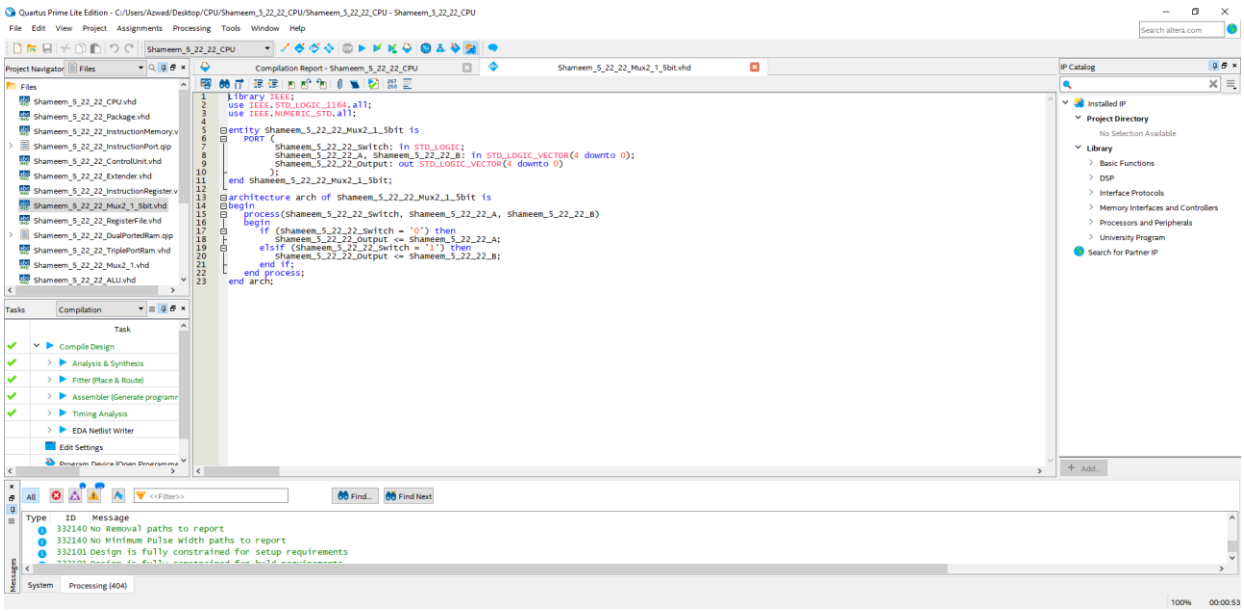


Figure 9: Mux 2:1 5-bit VHDL code. Used for input decide RW from RD or RT.

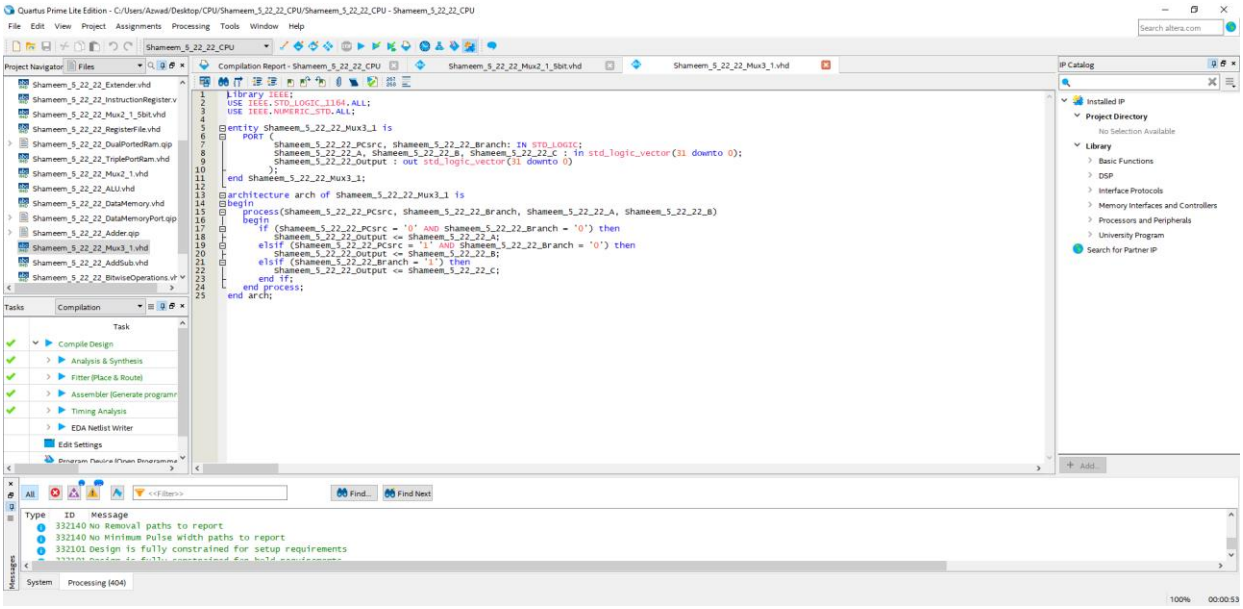


Figure 10: Mux 3:1 VHDL code. Used for input decide address for regular instructions, BNE/BEQ and jump.

Bitwise Operations

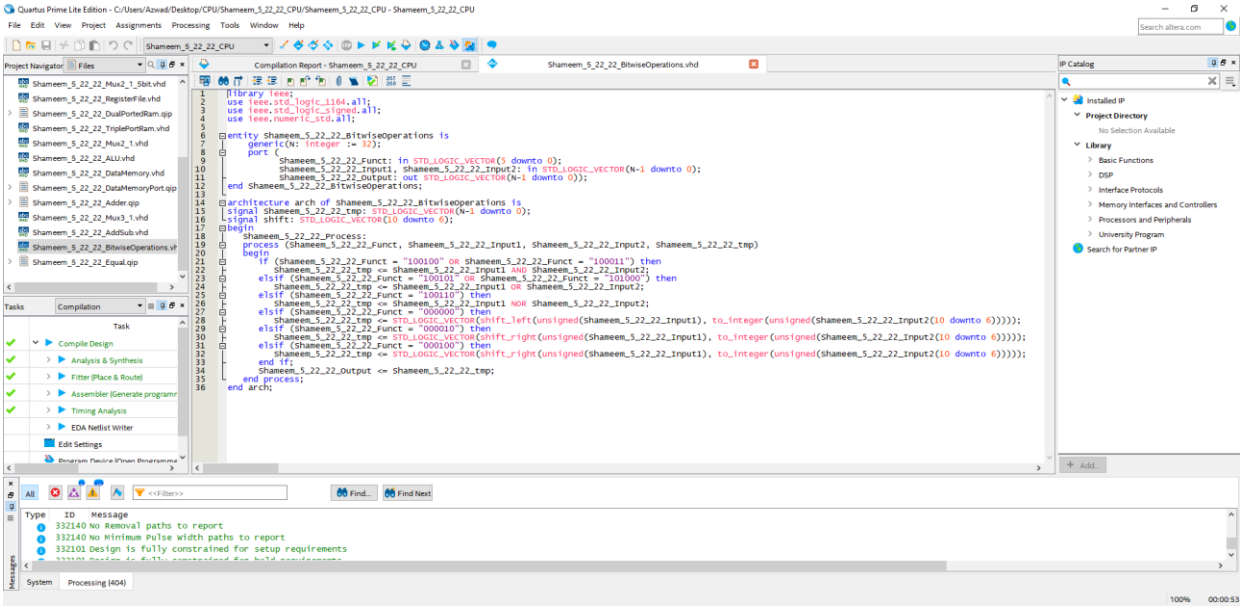


Figure 11: Bitwise Operations VHDL code. Used for instructions such as AND, OR, NOR, ORI, ANDI, SRL, SRL, SLL.

Adder

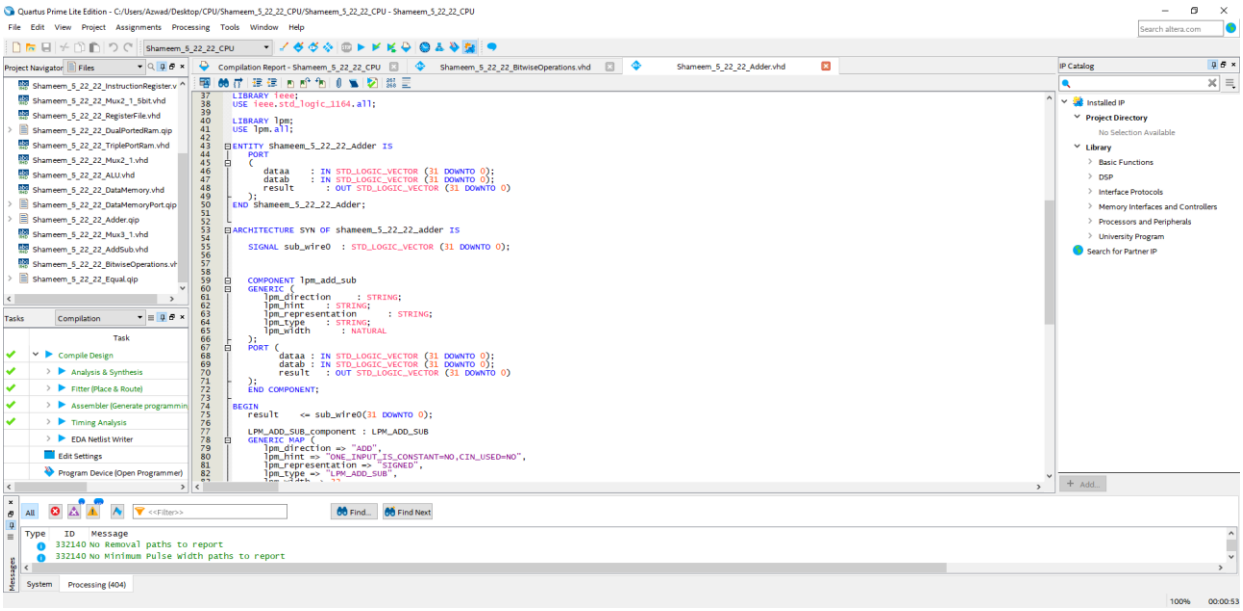


Figure 12: Adder VHDL code. Used for address change per instruction.

Equal

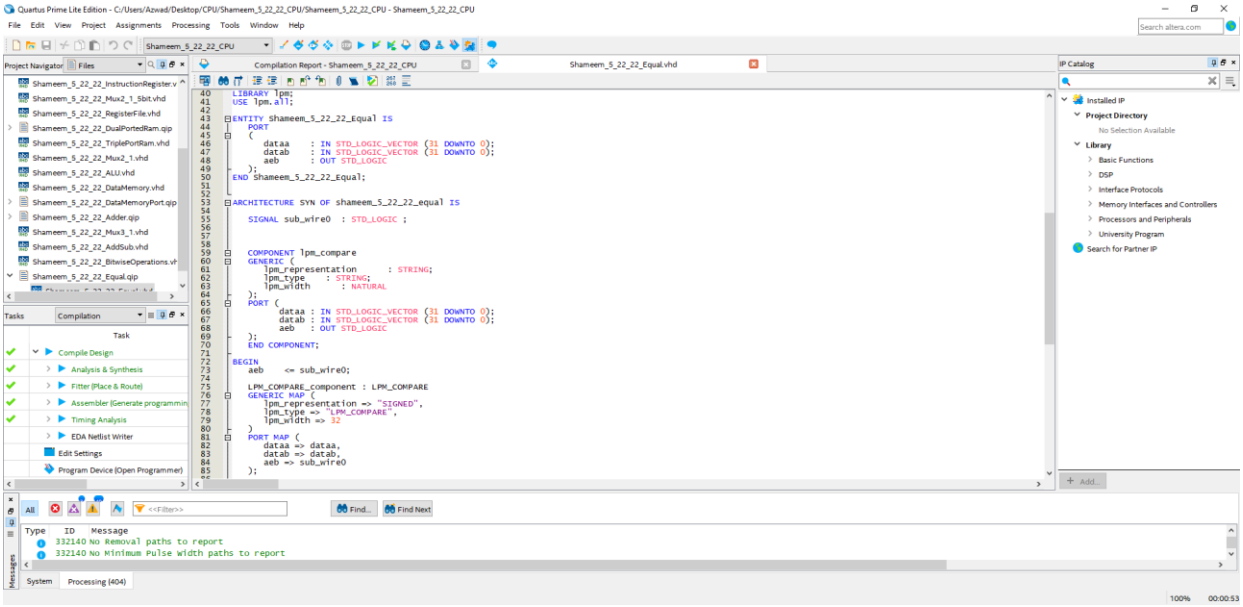


Figure 12: Equal VHDL code. Used BNE and BEQ to check condition to branch.

Data Memory

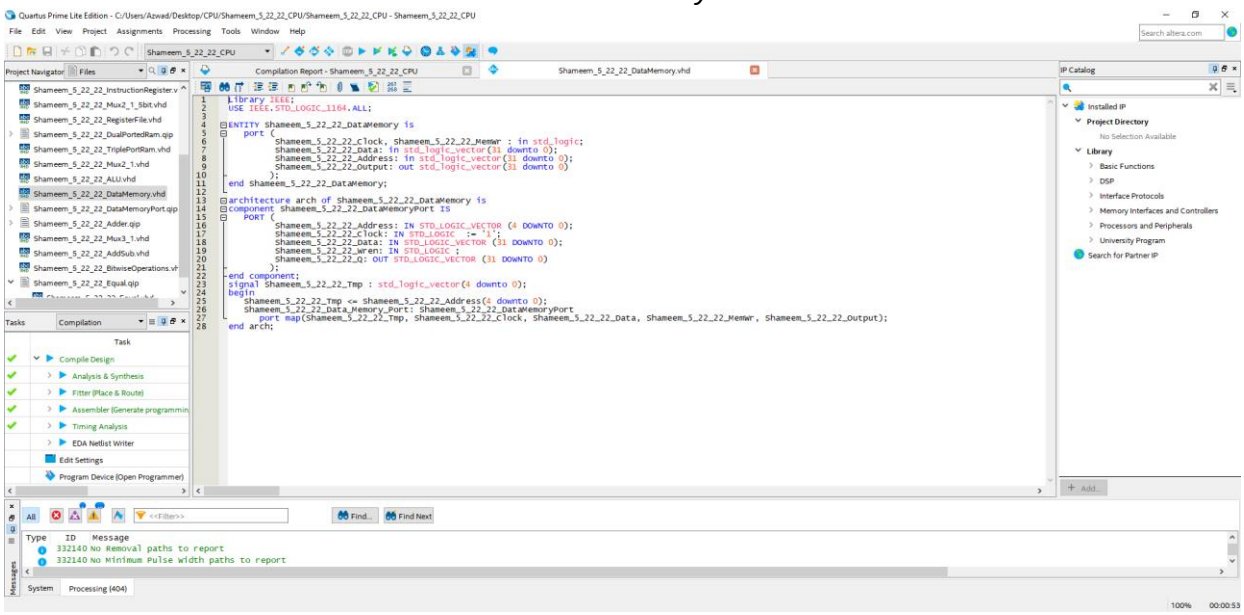


Figure 13: Data Memory VHDL code.

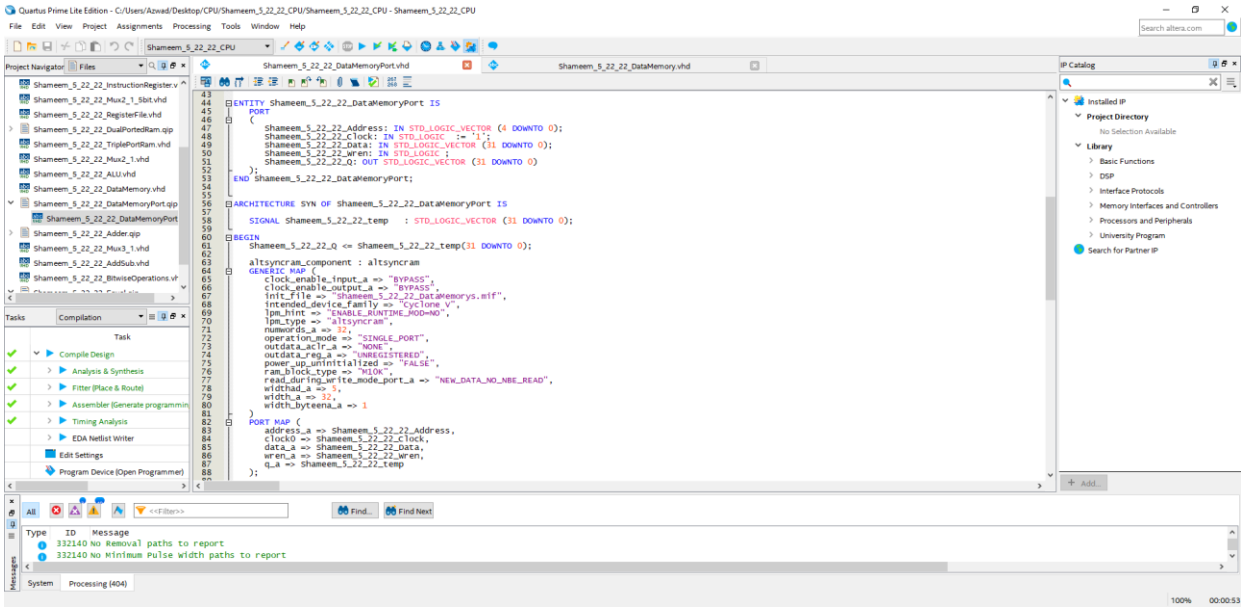


Figure 14: Data Memory Port VHDL code. This is used in Data Memory.

Register File

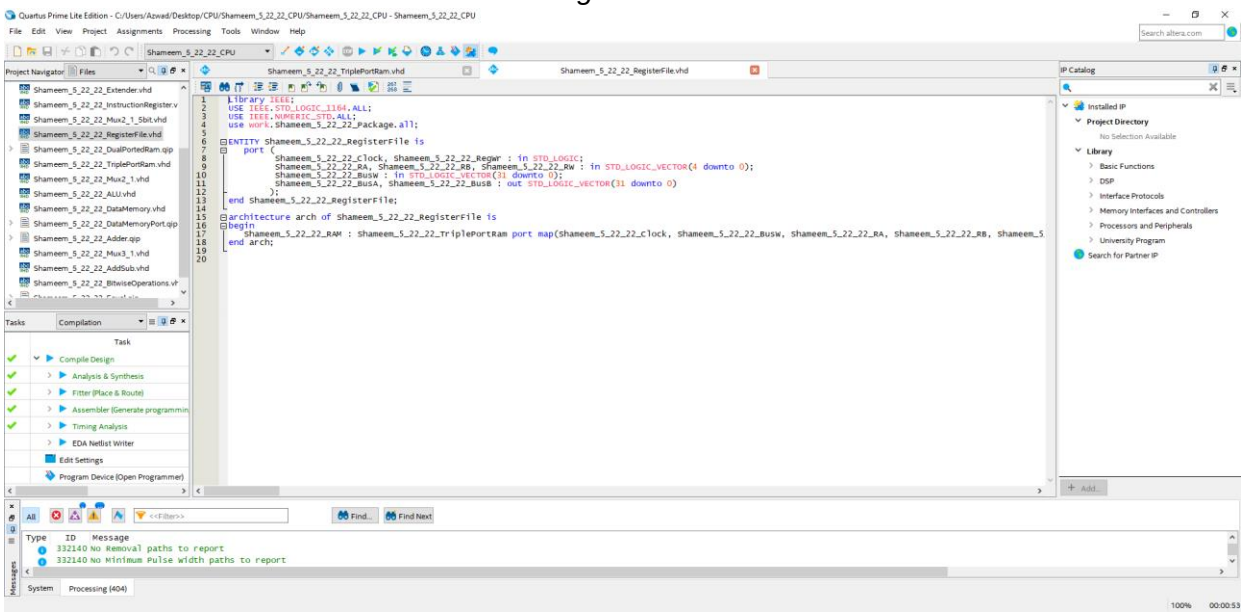


Figure 15: Register File VHDL code.

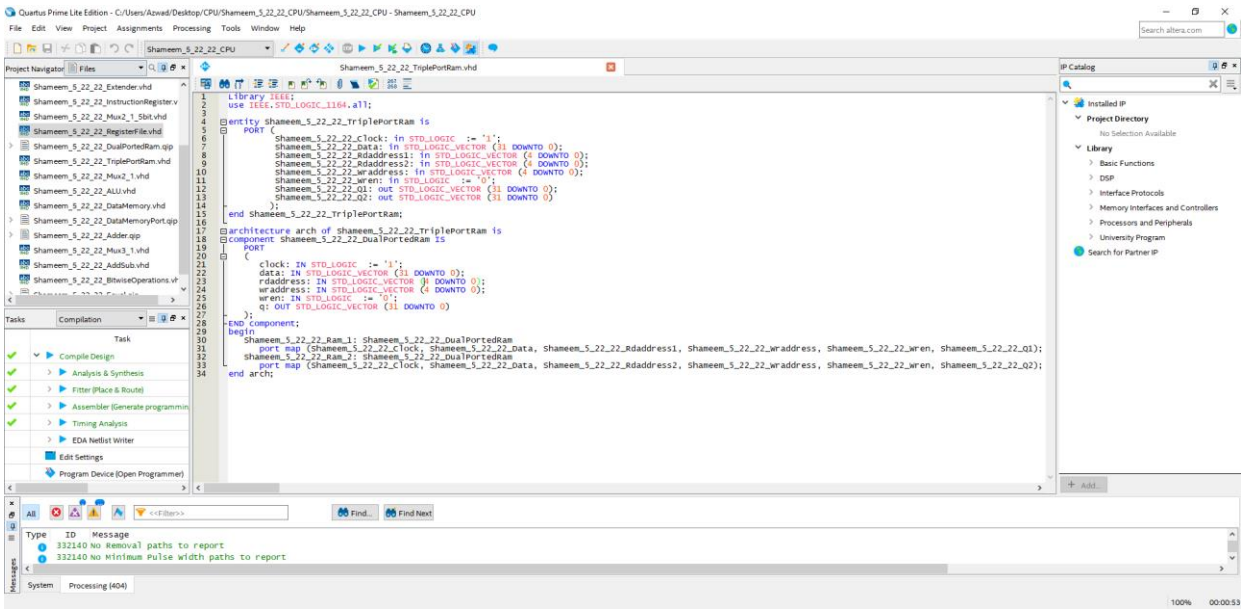


Figure 16: Triple Port Ram VHDL code.

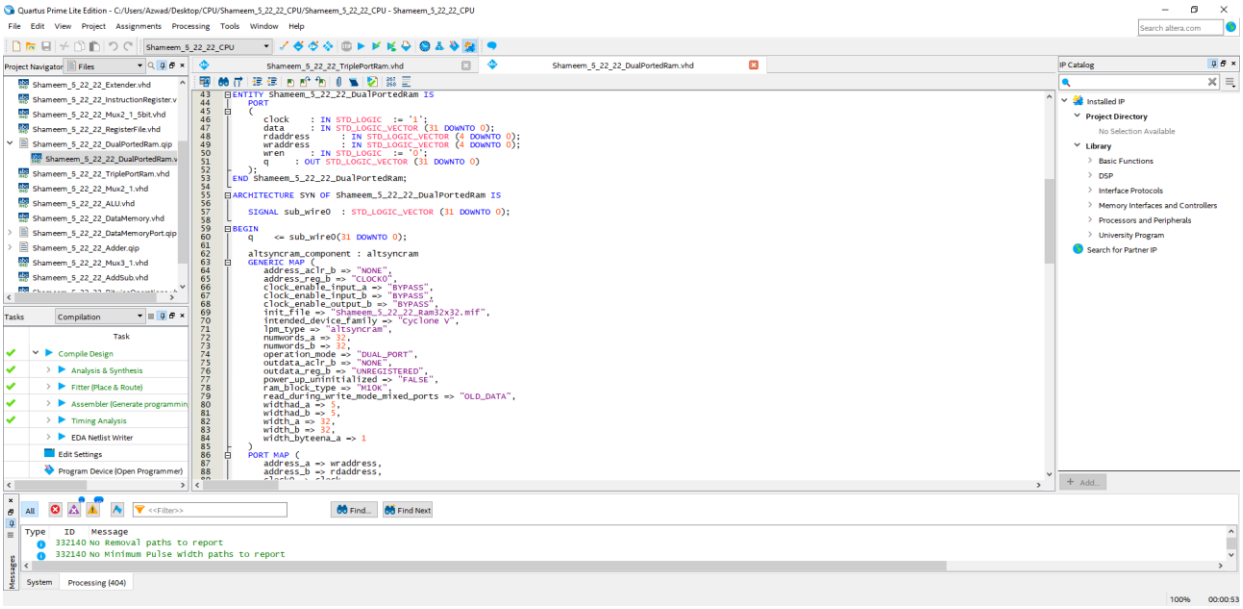


Figure 16: Dual Port Ram VHDL code. Used in Triple Ported Ram.

Instruction Register

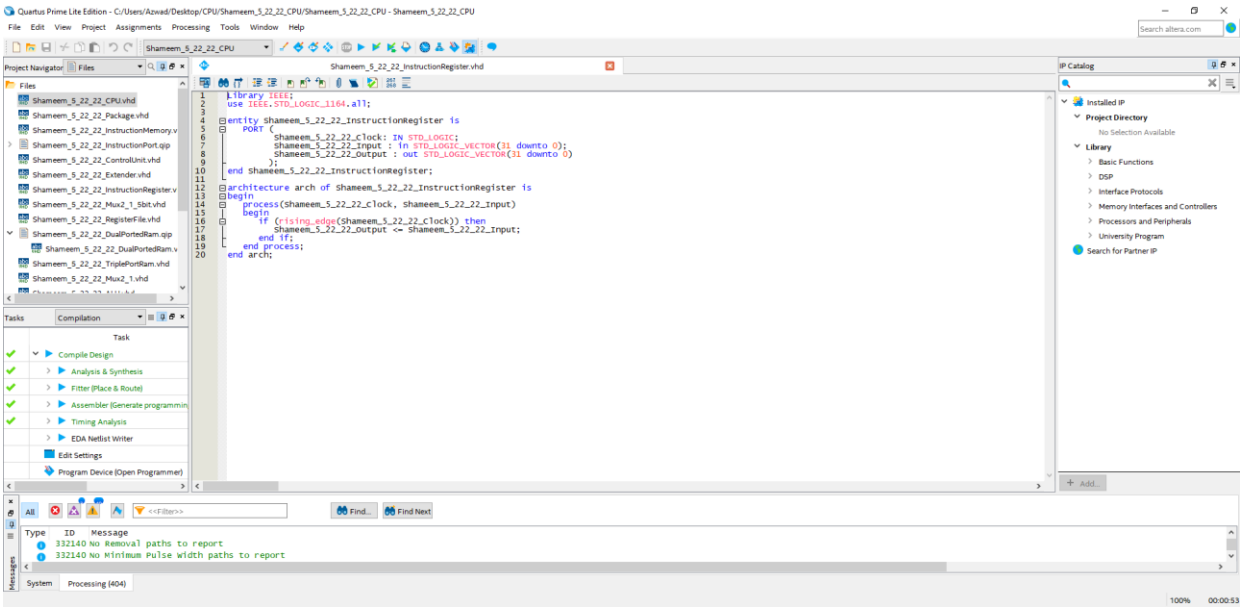


Figure 17: Instruction Register VHDL code.

PC Register

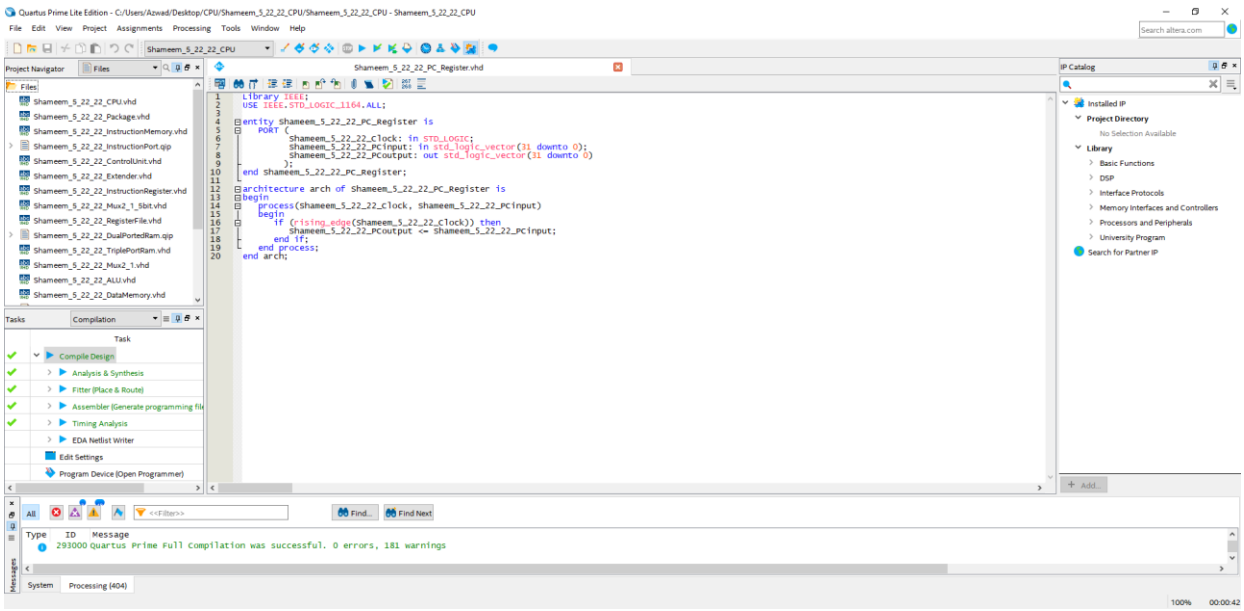


Figure 18: PC Register VHDL code.

Control Unit

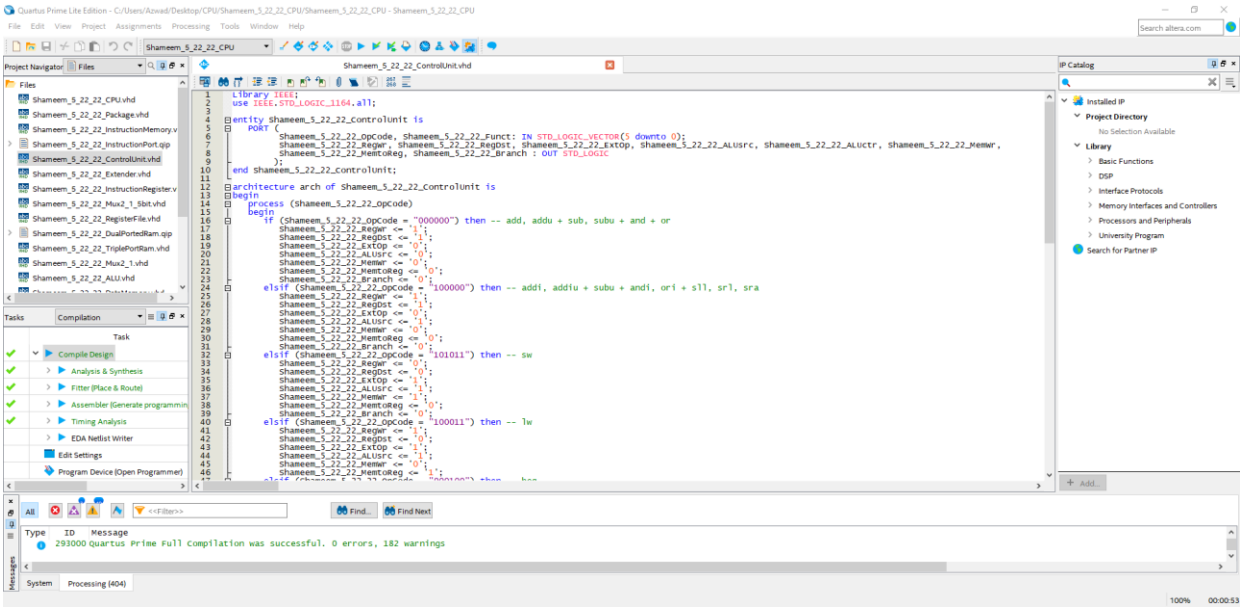


Figure 19: Control Unit VHDL code.

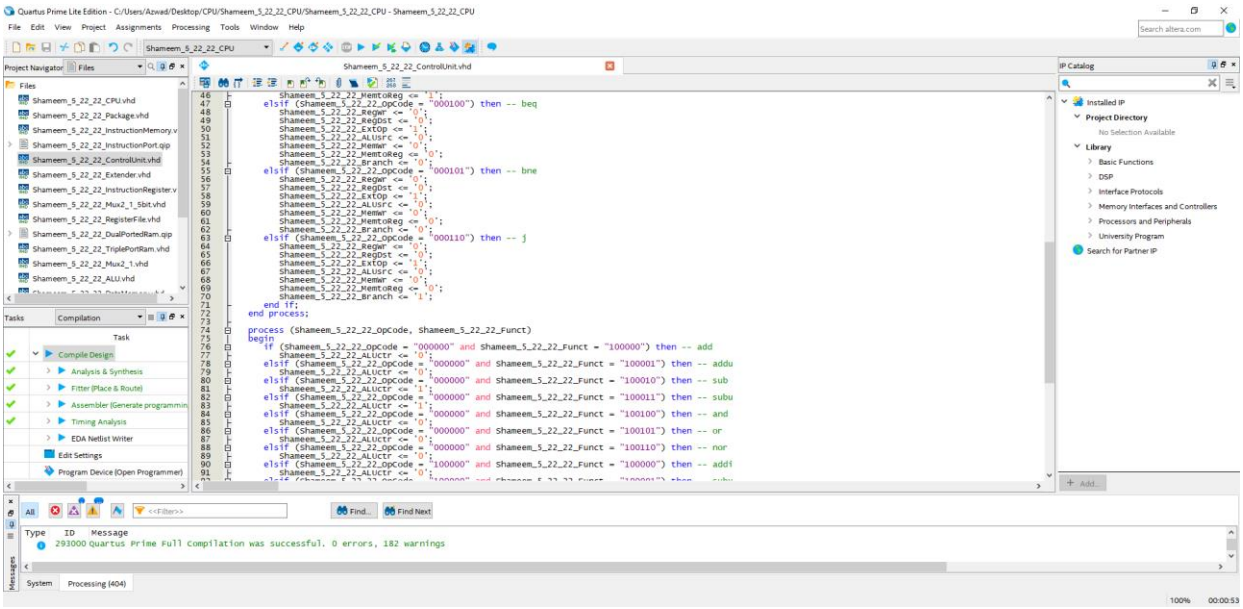


Figure 20: Control Unit VHDL code continued.

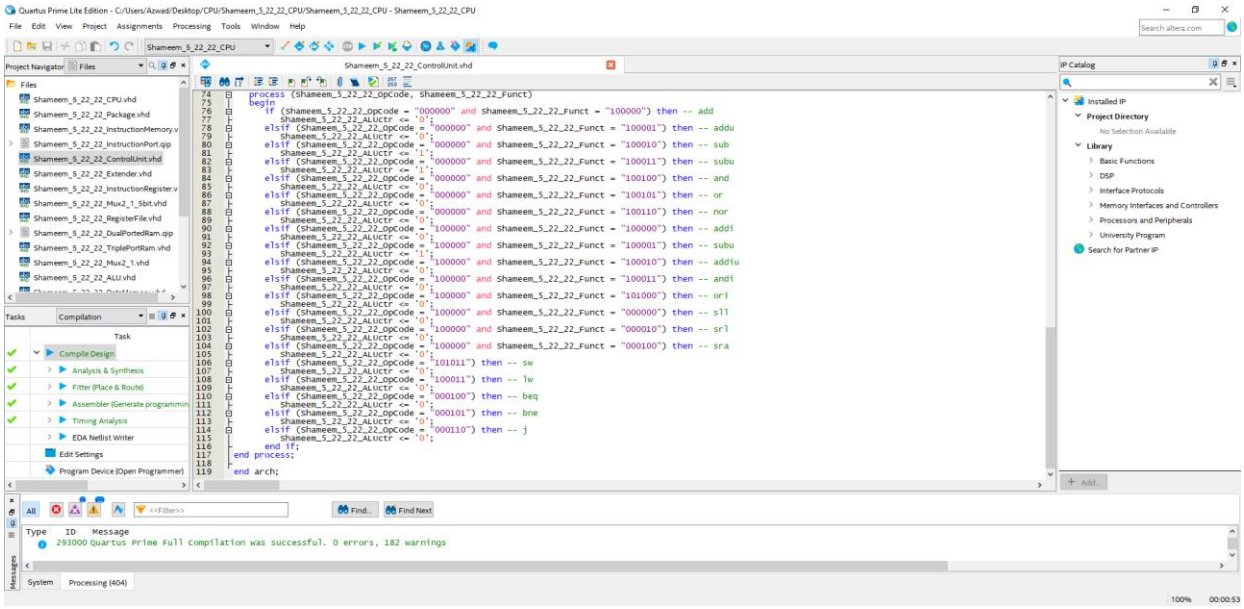


Figure 21: Control Unit VHDL code continued.

ALU

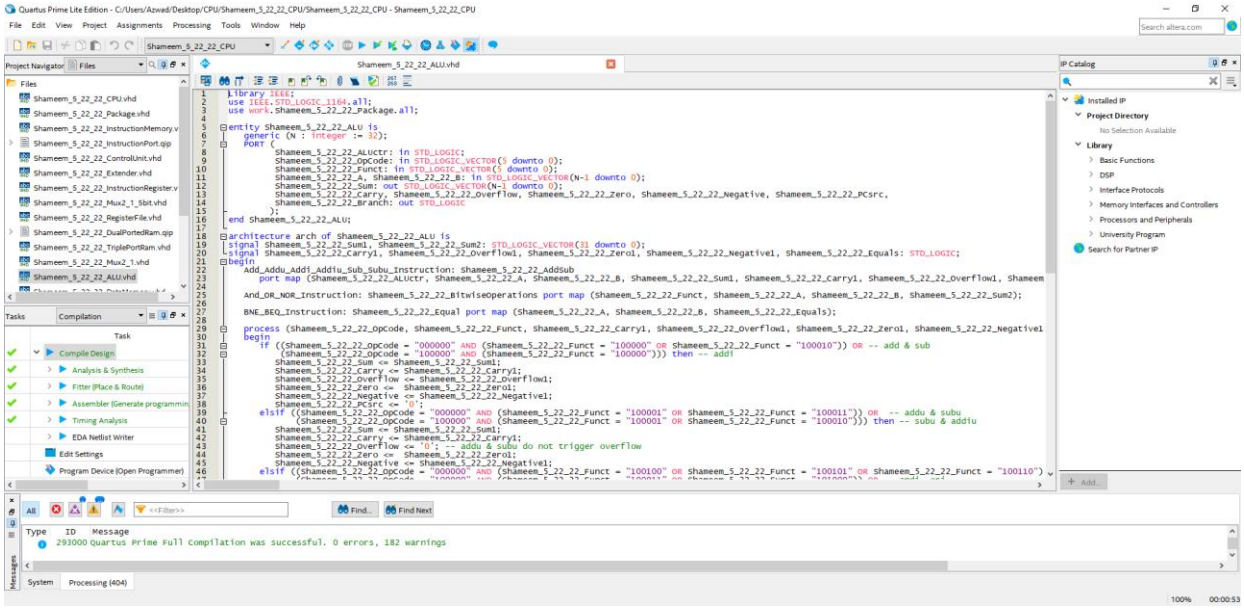


Figure 22: ALU VHDL code.

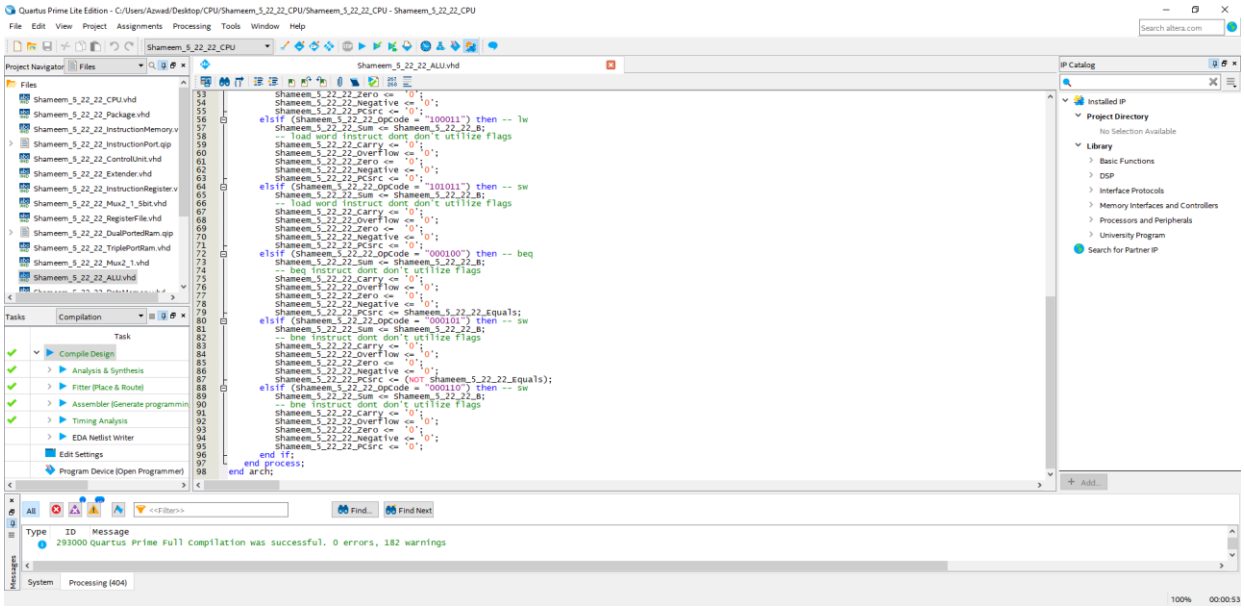
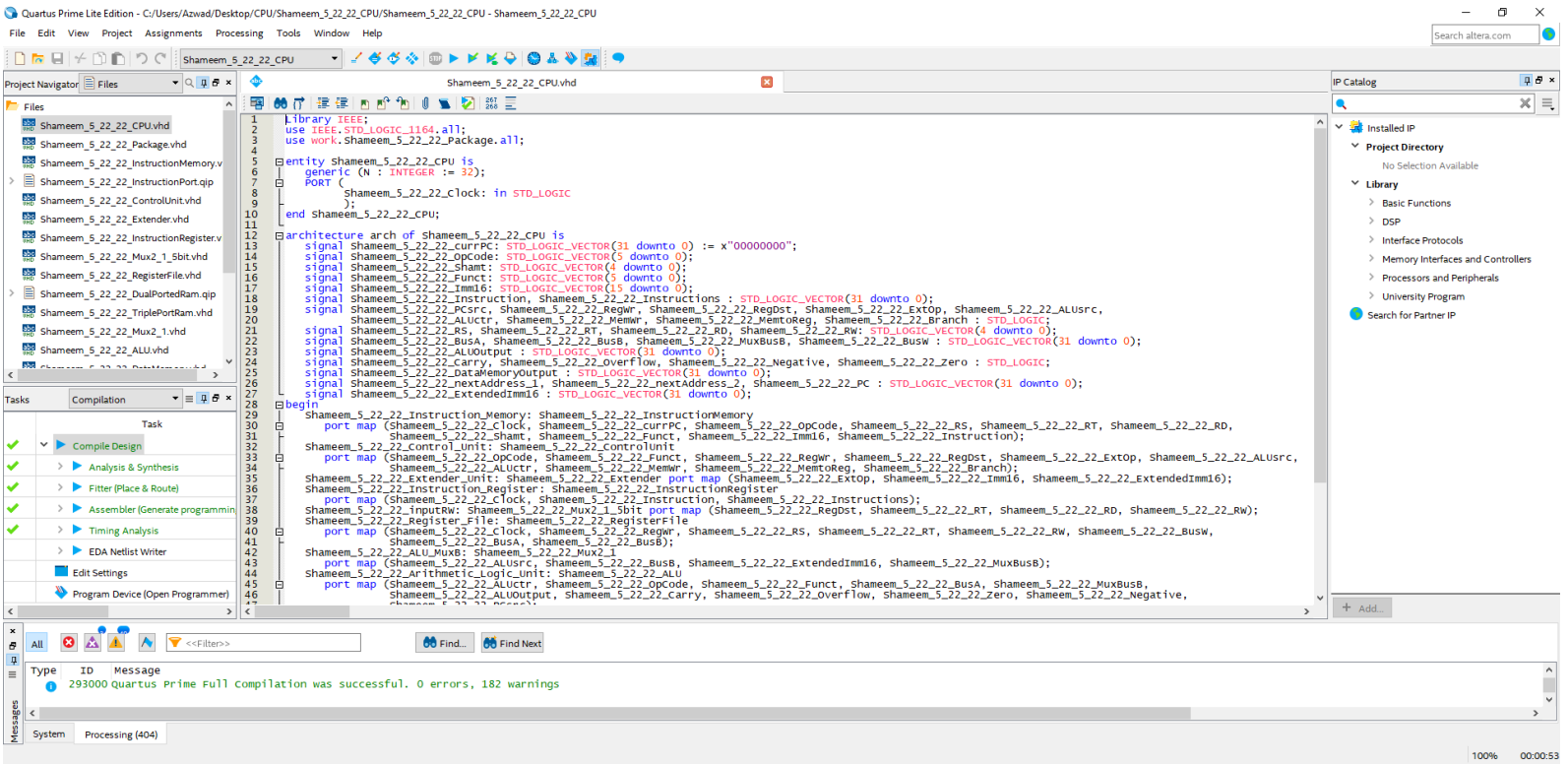


Figure 23: ALU VHDL code continued.

Single Cycle CPU




```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.Shameem_5_22_22_Package.all;

entity Shameem_5_22_22_CPU is
    generic (N : INTEGER := 32);
    PORT (
          Shameem_5_22_22_Clock: in STD_LOGIC
          );
end Shameem_5_22_22_CPU;

architecture arch of Shameem_5_22_22_CPU is
    signal Shameem_5_22_22_currPC: STD_LOGIC_VECTOR(31 downto 0) := x"00000000";
    signal Shameem_5_22_22_OpCode: STD_LOGIC_VECTOR(5 downto 0);
    signal Shameem_5_22_22_Shamt: STD_LOGIC_VECTOR(4 downto 0);
    signal Shameem_5_22_22_Funct: STD_LOGIC_VECTOR(5 downto 0);
    signal Shameem_5_22_22_Imm16: STD_LOGIC_VECTOR(15 downto 0);
    signal Shameem_5_22_22_Instruction, Shameem_5_22_22_Instructions : STD_LOGIC_VECTOR(31 downto 0);
    signal Shameem_5_22_22_PCsrc, Shameem_5_22_22_Regwr, Shameem_5_22_22_RegDst, Shameem_5_22_22_ExtOp, Shameem_5_22_22_ALUsrc,
           Shameem_5_22_22_ALUctr, Shameem_5_22_22_Memwr, Shameem_5_22_22_MemtoReg, Shameem_5_22_22_Branch : STD_LOGIC;
    signal Shameem_5_22_22_RS, Shameem_5_22_22_RT, Shameem_5_22_22_RD, Shameem_5_22_22_RW: STD_LOGIC_VECTOR(4 downto 0);
    signal Shameem_5_22_22_BusA, Shameem_5_22_22_BusB, Shameem_5_22_22_MuxBusB, Shameem_5_22_22_BusW : STD_LOGIC_VECTOR(31 downto 0);
    signal Shameem_5_22_22_ALUOutput : STD_LOGIC_VECTOR(31 downto 0);
    signal Shameem_5_22_22_Carry, Shameem_5_22_22_Overflow, Shameem_5_22_22_Negative, Shameem_5_22_22_Zero : STD_LOGIC;
    signal Shameem_5_22_22_DataMemoryOutput : STD_LOGIC_VECTOR(31 downto 0);
    signal Shameem_5_22_22_nextAddress_1, Shameem_5_22_22_nextAddress_2, Shameem_5_22_22_PC : STD_LOGIC_VECTOR(31 downto 0);
    signal Shameem_5_22_22_ExtendedImm16 : STD_LOGIC_VECTOR(31 downto 0);
begin
    Shameem_5_22_22_Instruction_Memory: Shameem_5_22_22_InstructionMemory
        port map (Shameem_5_22_22_Clock, Shameem_5_22_22_currPC, Shameem_5_22_22_OpCode, Shameem_5_22_22_RS, Shameem_5_22_22_RT, Shameem_5_22_22_RD,
                  Shameem_5_22_22_Shamt, Shameem_5_22_22_Funct, Shameem_5_22_22_Imm16, Shameem_5_22_22_Instruction);
    Shameem_5_22_22_Control_Unit: Shameem_5_22_22_ControlUnit
        port map (Shameem_5_22_22_OpCode, Shameem_5_22_22_Funct, Shameem_5_22_22_Regwr, Shameem_5_22_22_RegDst, Shameem_5_22_22_ExtOp, Shameem_5_22_22_ALUsrc,
                  Shameem_5_22_22_ALUctr, Shameem_5_22_22_Memwr, Shameem_5_22_22_MemtoReg, Shameem_5_22_22_Branch);
    Shameem_5_22_22_Extender_Unit: Shameem_5_22_22_Extender port map (Shameem_5_22_22_ExtOp, Shameem_5_22_22_Imm16, Shameem_5_22_22_ExtendedImm16);
    Shameem_5_22_22_Instruction_Register: Shameem_5_22_22_InstructionRegister
        port map (Shameem_5_22_22_Clock, Shameem_5_22_22_Instruction, Shameem_5_22_22_Instructions);
    Shameem_5_22_22_inputRW: Shameem_5_22_22_Mux2_1_5bit port map (Shameem_5_22_22_RegDst, Shameem_5_22_22_RT, Shameem_5_22_22_RD, Shameem_5_22_22_RW);
    Shameem_5_22_22_Register_File: Shameem_5_22_22_RegisterFile
        port map (Shameem_5_22_22_Clock, Shameem_5_22_22_Regwr, Shameem_5_22_22_RS, Shameem_5_22_22_RT, Shameem_5_22_22_RW, Shameem_5_22_22_BusW,
                  Shameem_5_22_22_BusA, Shameem_5_22_22_BusB);
    Shameem_5_22_22_ALU_MuxB: Shameem_5_22_22_Mux2_1
        port map (Shameem_5_22_22_ALUsrc, Shameem_5_22_22_BusB, Shameem_5_22_22_ExtendedImm16, Shameem_5_22_22_MuxBusB);
    Shameem_5_22_22_Arithmetic_Logic_Unit: Shameem_5_22_22_ALU
        port map (Shameem_5_22_22_ALUctr, Shameem_5_22_22_OpCode, Shameem_5_22_22_Funct, Shameem_5_22_22_BusA, Shameem_5_22_22_MuxBusB,
                  Shameem_5_22_22_ALUOutput, Shameem_5_22_22_Carry, Shameem_5_22_22_Overflow, Shameem_5_22_22_Zero, Shameem_5_22_22_Negative,
```

Figure 24: Single Cycle CPU VHDL code.

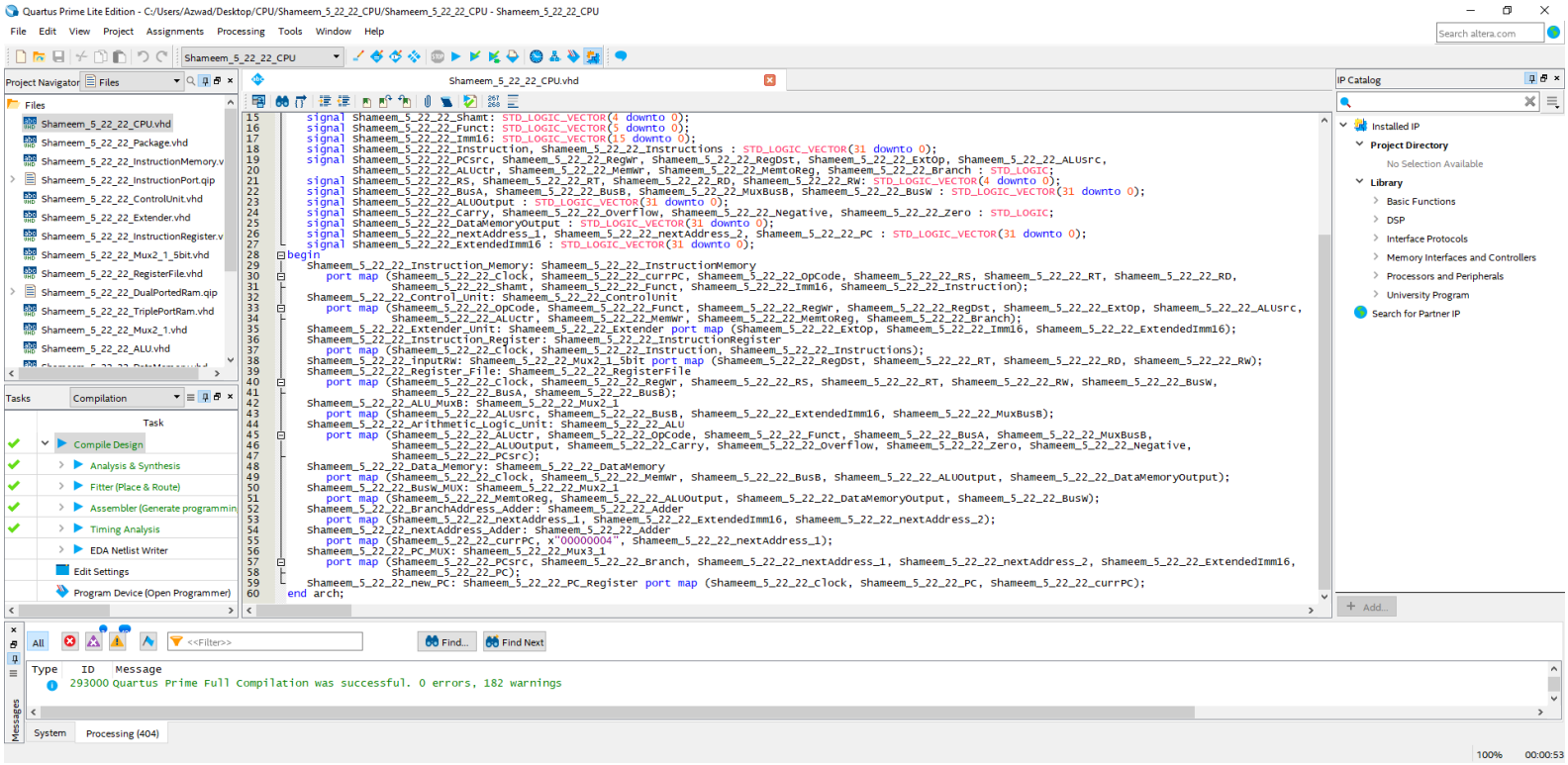


Figure 25: Single Cycle CPU VHDL code continued.

1. **Perform the ultimate test of your design by inputting integers x_1, x_2, x_3, x_4, x_5, into DATA Memory, and computing the following expression**

$$Z = \sum_{k=1}^{5} X_k$$

Simulation

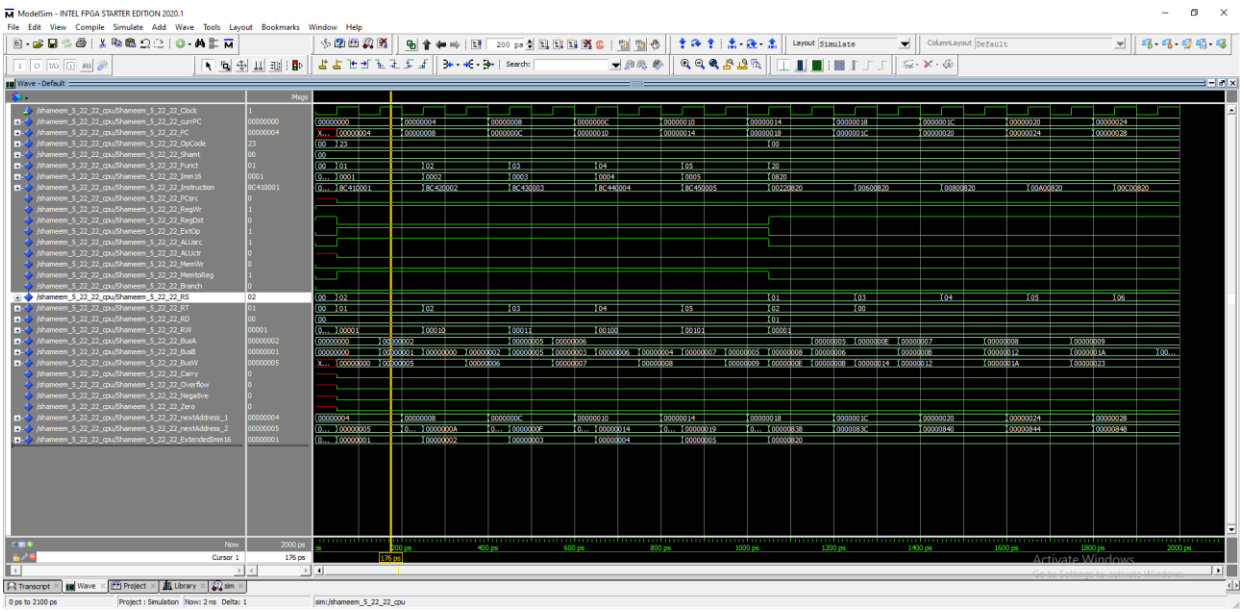


Figure 26: Simulation Waveform

This is the simulation of using 5 load word instructions and then 5 add instructions and finally one save word instruction. We will be verifying every instruction one by one in a zoomed in snipit.

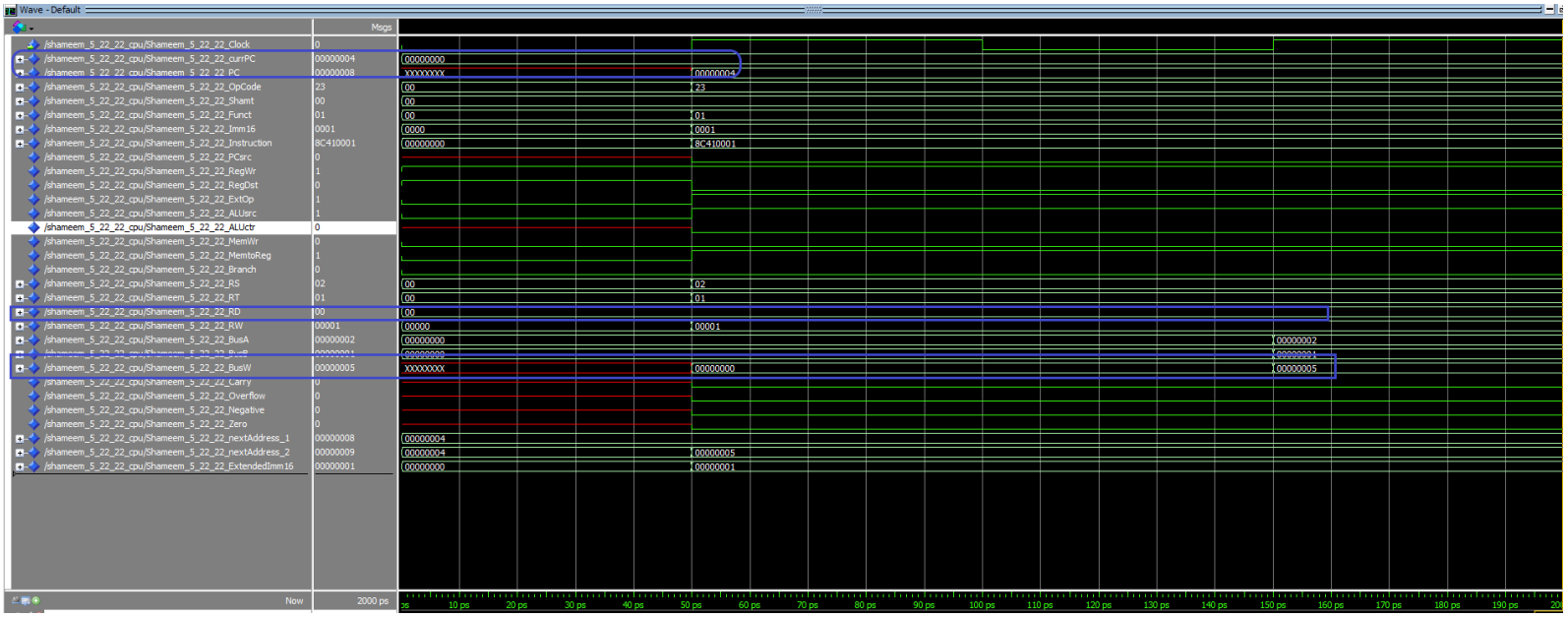


Figure 27: Simulation Waveform Zoomed in 100 – 200 ps - LW

First of all let's verify the program counter. On the top it is easily noticeable that the PC goes from 0x00000000 to 0x00000004 at 50 ps this is because the BNE, BEQ or J instructions are not called so we only add 4 to the current program counter. Next we see that at RW at 50 ps the register is the register at 00001. Then if you look at BusW it is noticeable that at 150 ps the number 0x00000005 is loaded into RW which is the register at 00001.

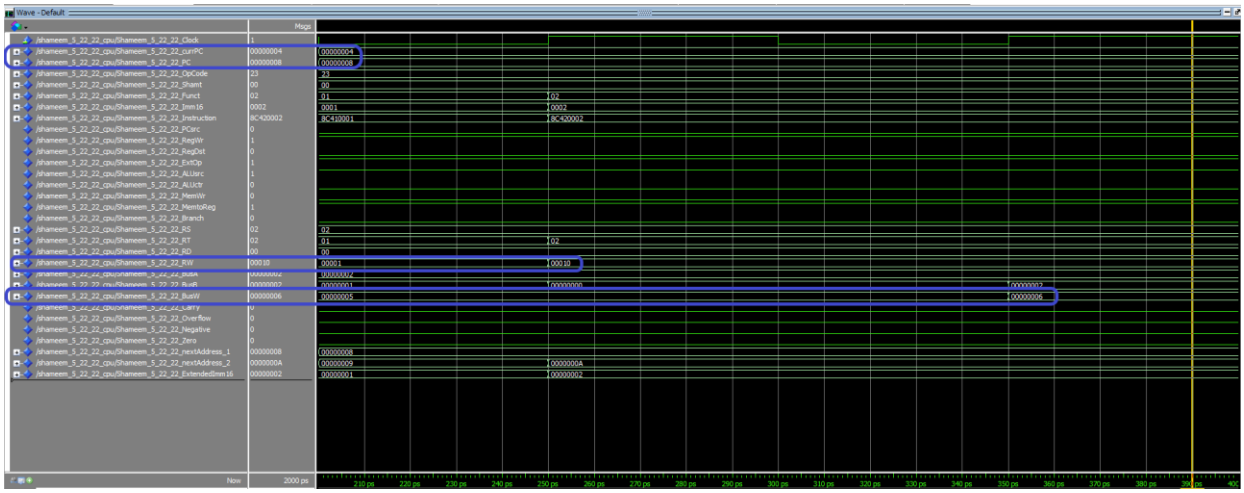


Figure 28: Simulation Waveform Zoomed in 200 - 400ps - LW

First of all, let's verify the program counter. On the top it is easily noticeable that the PC goes from 0x00000004 to 0x00000008 at 200 ps this is because the BNE, BEQ or J instructions are not called so we only add 4 to the current program counter. Next, we see that at RW at 250 ps the register is the register at 00010. Then if you look at BusW it is noticeable that at 150 ps the number 0x00000006 is loaded into RW which is the register at 00010.

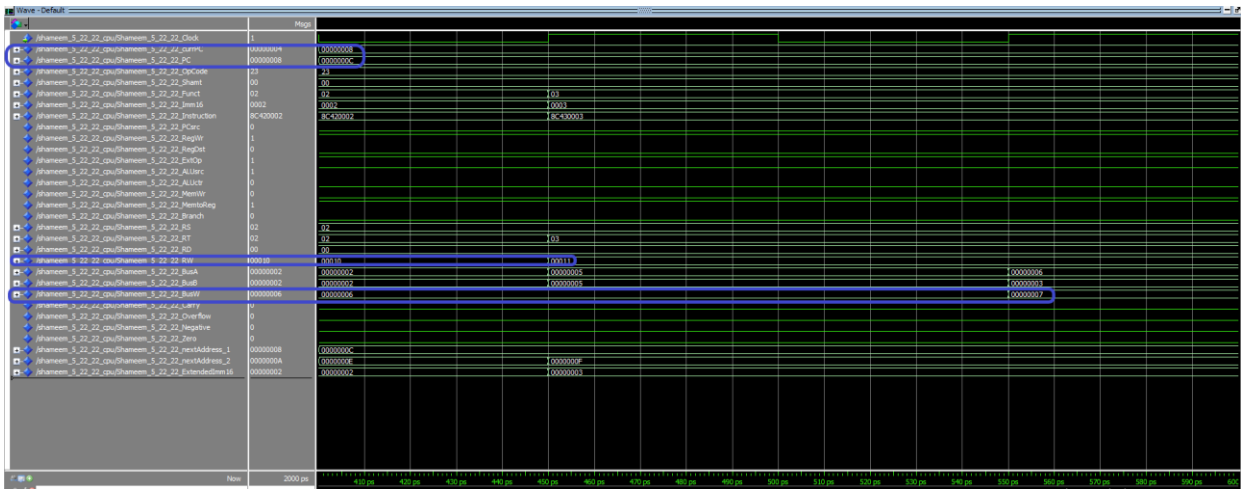


Figure 29: Simulation Waveform Zoomed in 400 - 600 ps - LW

First of all let's verify the program counter. On the top it is easily noticeable that the PC goes from 0x00000008 to 0x0000000C at 400 ps this is because the BNE, BEQ or J instructions are called so we only add 4 to the current program counter. Next we see that at RW at 450 ps the register is the register at 00011. Then if you look at BusW it is noticeable that at 550 ps the number 0x00000007 is loaded into RW which is the register at 00011.

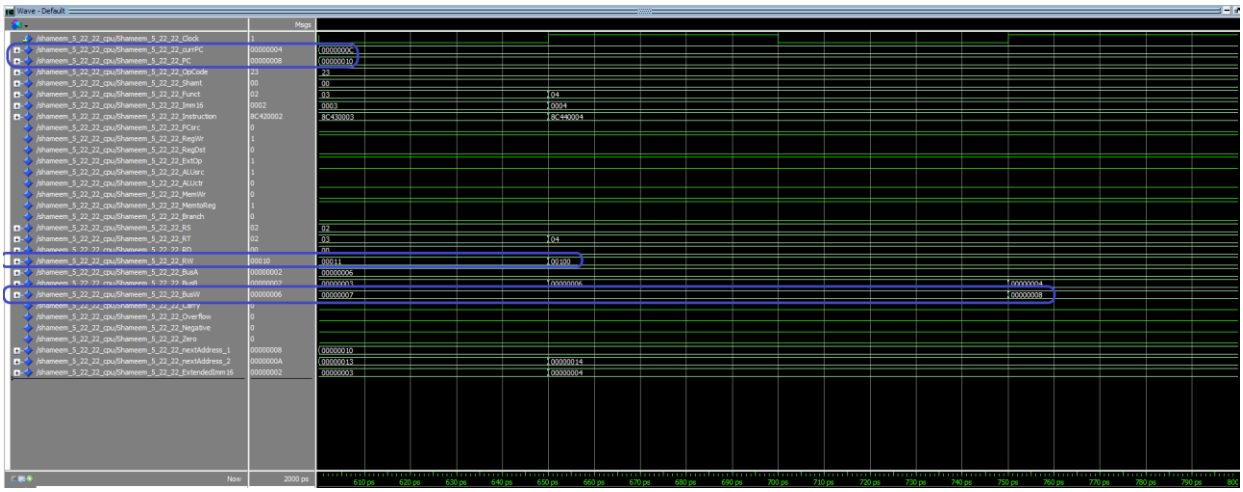


Figure 30: Simulation Waveform Zoomed in 600 - 800ps - LW

First of all let's verify the program counter. On the top it is easily noticeable that the PC goes from 0x0000000C to 0x00000010 at 600 ps this is because the BNE, BEQ or J instructions are called so we only add 4 to the current program counter. Next we see that at RW at 650 ps the register is the register at 00100. Then if you look at BusW it is noticeable that at 750 ps the number 0x00000008 is loaded into RW which is the register at 00100.

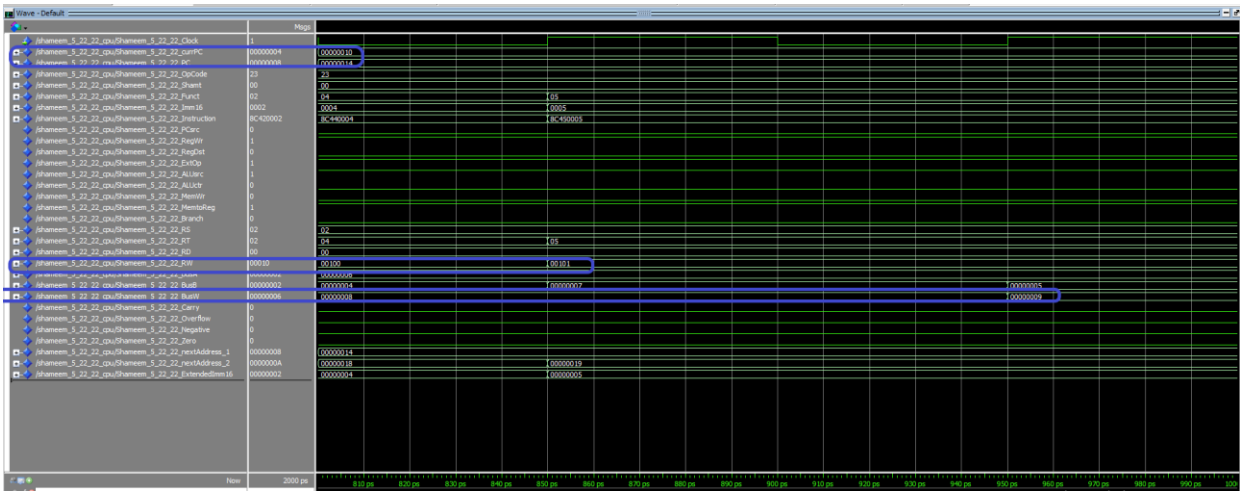


Figure 31: Simulation Waveform Zoomed in 800-1000ps - LW

First of all, let's verify the program counter. On the top it is easily noticeable that the PC goes from 0x00000010 to 0x00000014 at 800 ps this is because the BNE, BEQ or J instructions are called so we only add 4 to the current program counter. Next, we see that at RW at 850 ps the register is the register at 00101. Then if you look at BusW it is noticeable that at 950 ps the number 0x00000009 is loaded into RW which is the register at 00101. Therefore, at this point 5 instructions have been completed and 5 integers are stored in the registers 00001, 00010, 00011, 00100, 00101

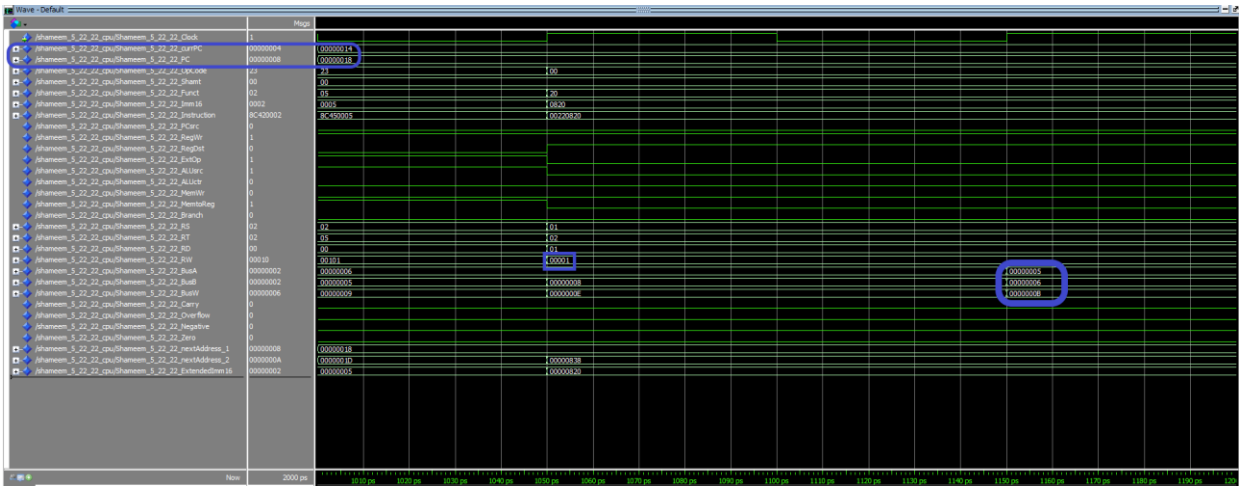


Figure 32: Simulation Waveform Zoomed in 1000 - 1200ps - Add

First of all, let's verify the program counter. On the top it is easily noticeable that the PC goes from 0x00000014 to 0x00000018 at 800 ps this is because the BNE, BEQ or J instructions are called so we only add 4 to the current program counter. Next, we see that RS is 1 which is register 00001, and RT is 2 which is register 00010 at 1050 ps. Then at 1150 ps it is noticeable that the numbers 0x00000005 and 0x00000006 we have loaded into the registers are adding and we get the result 0x0000000B in BusW. Furthermore, RW is 00001 from 1050 ps and after so the sum of addition will be saved in the register 00001.

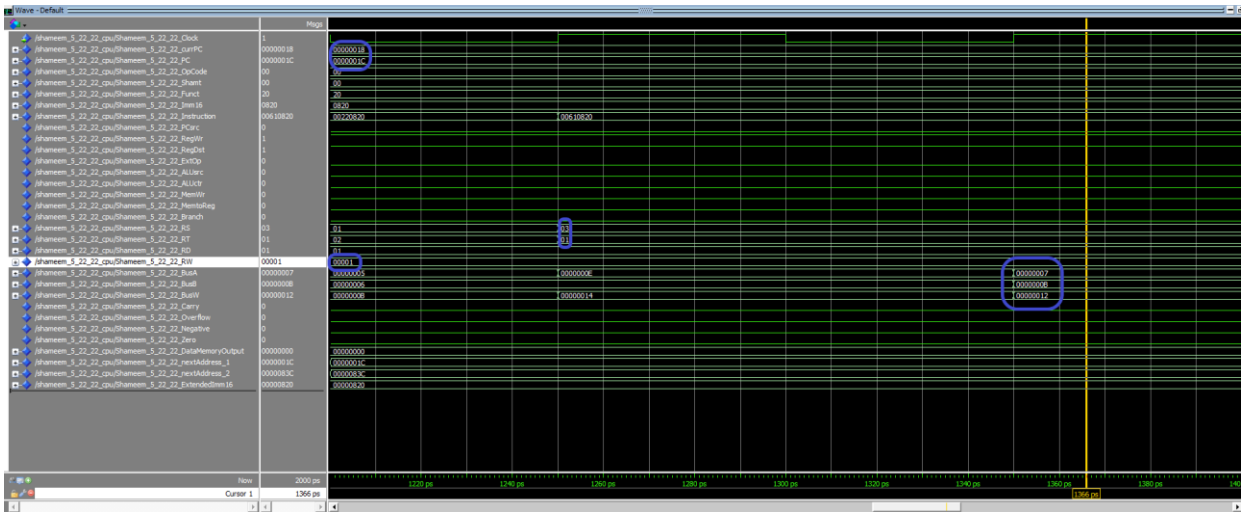


Figure 33: Simulation Waveform Zoomed in 1200 - 1400ps - Add

First of all, let's verify the program counter. On the top it is easily noticeable that the PC goes from 0x00000018 to 0x0000001C at 1200 ps this is because the BNE, BEQ or J instructions are called so we only add 4 to the current program counter. Next, we see that RS is 3 which is register 00011, and RT is 1 which is register 00001 at 1050 ps. Then at 1350 ps it is noticeable that the number 0x00000007 in BusA we have loaded is there and the number 0x0000000B in BusB, which is the sum we added for previously is now being added and gives us the value of 0x00000012 in BusW. Furthermore, RW is 00001 from 1200 ps and after so the sum of addition will be saved in the register 00001.

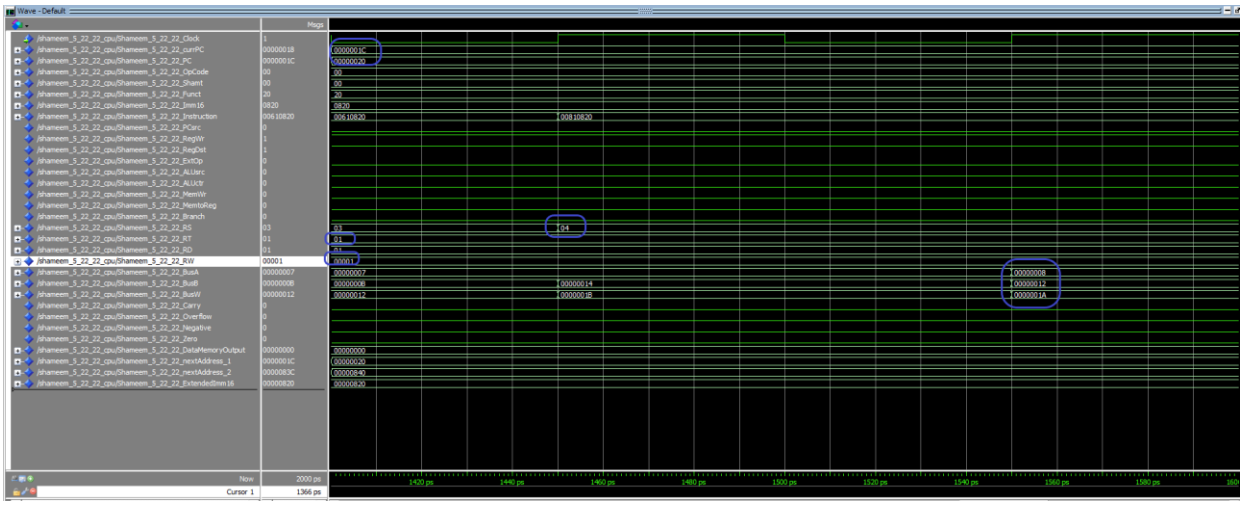


Figure 34: Simulation Waveform Zoomed in 1400 - 1600ps - Add

First of all, let's verify the program counter. On the top it is easily noticeable that the PC goes from 0x0000001C to 0x00000020 at 1400 ps this is because the BNE, BEQ or J instructions are called so we only add 4 to the current program counter. Next, we see that RS is 4 which is register 00100, and RT is 1 which is register 00001 at 1450 ps. Then at 1550 ps it is noticeable that the number 0x00000008 in BusA we have loaded is there and the number 0x00000012 in BusB, which is the sum we added for previously is now being added and gives us the value of 0x0000001A in BusW. Furthermore, RW is 00001 from 1400 ps and after so the sum of addition will be saved in the register 00001.

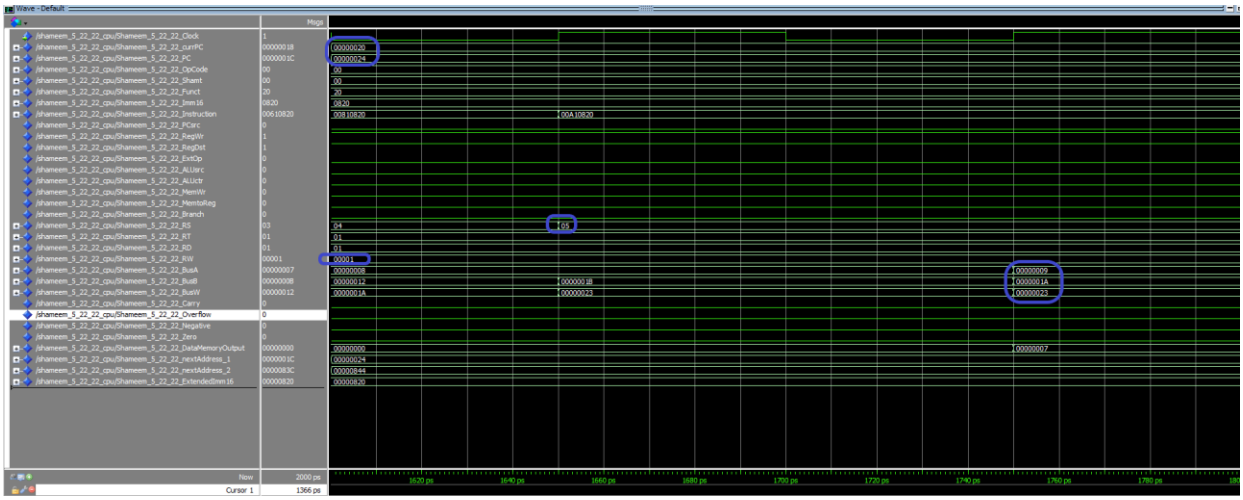


Figure 34: Simulation Waveform Zoomed in 1600 - 1800ps - Add

First of all, let's verify the program counter. On the top it is easily noticeable that the PC goes from 0x00000020 to 0x00000024 at 1600 ps this is because the BNE, BEQ or J instructions are called so we only add 4 to the current program counter. Next, we see that RS is 5 which is register 00101, and RT is 1 which is register 00001 at 1650 ps. Then in 1750 ps it is noticeable that the number 0x00000009 in BusA we have loaded is there and the number 0x0000001A in BusB, which is the sum we added for previously is now being added and gives us the value of 0x00000023 in BusW. Furthermore, RW is 00001 from 1600 ps and after so the sum of addition will be saved in the register 00001.

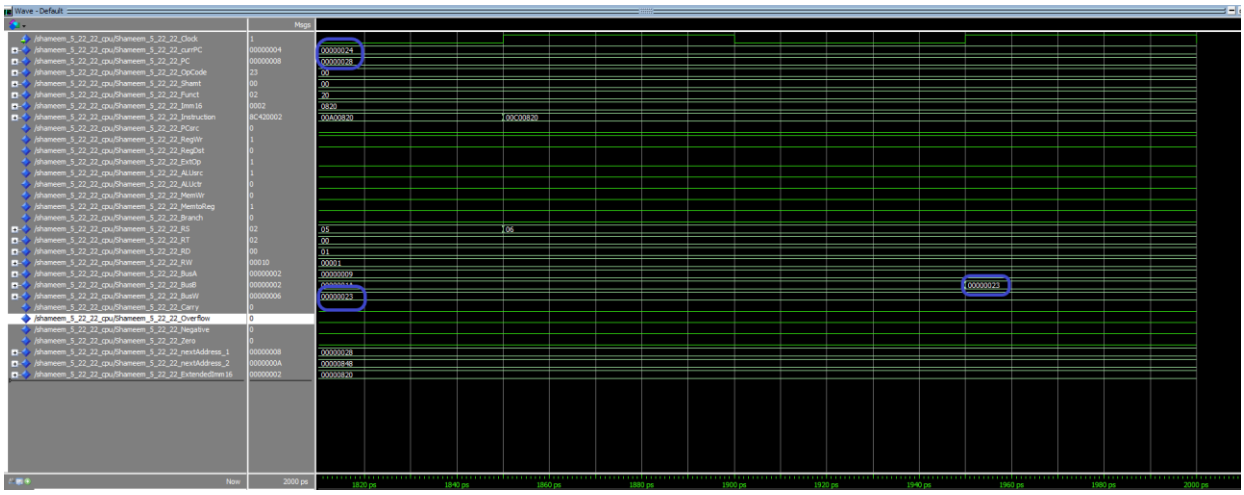


Figure 35: Simulation Waveform Zoomed in 1800 – 2000 ps - SW

First of all, let's verify the program counter. On the top it is easily noticeable that the PC goes from 0x00000024 to 0x00000028 at 1800 ps this is because the BNE, BEQ or J instructions are called so we only add 4 to the current program counter. We see at 1800 ps RW is is 00001 and busW is 0x00000023. Furthermore in 1950 ps we see that BusB shows 0x00000023, which shows that the number has saved properly.

MIPS in Mars Simulator

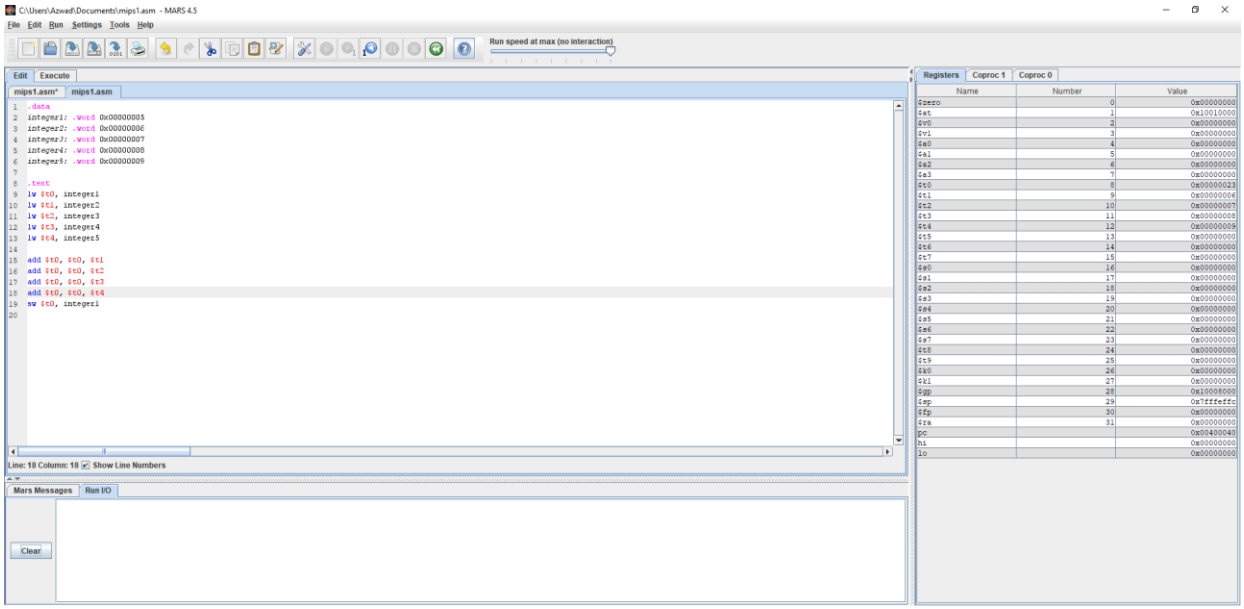


Figure 36: MIPS Code in MARS Simulator.

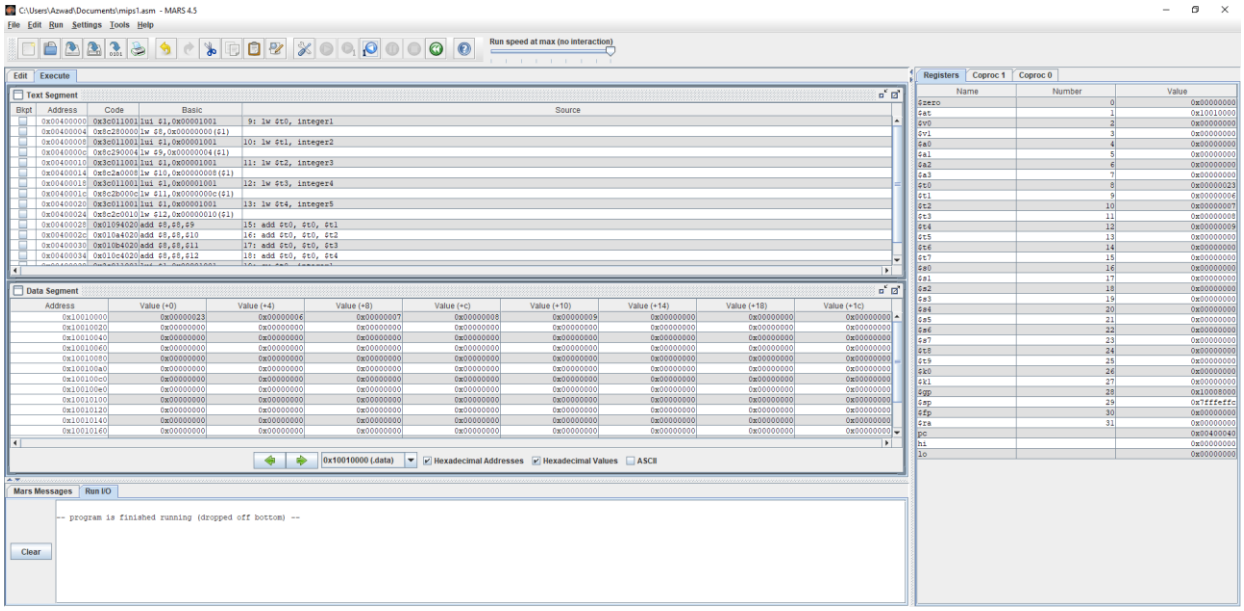


Figure 37: Executed in MARS Simulator.

At the Data Segment at 0x10010000, MARS Simulator also shows that 0x00000023 is the sum of the addition of the 5 numbers, just like the simulation waveforms.

Explanation

Let’s recap we got a total of 0x23. The To obtain this total we did 0x5 + 0x6 = 0xB then 0xB + 0x7 = 0x12 then 0x8 + 0x12 = 0x1A then 0x9 + 0x1A = 0x23. To verify if this is correct, we check the decimal version can add 5 + 6 + 7 + 8 + 9 = 35. 35 is in decimal, but in hexadecimal 35(decimal) is 0x23. So far by checking decimal version the correctness stacks up to the hexadecimal version. This is furthered by the same numbers showing up in MIPS in MARS Simulator. Therefore, we can say that the simulation correctly computes the number in VHDL.

Conclusion

The Single Cyle CPU lab was a great assignment to test our VHDL skills to simulate MIPS instructions in VHDL. Specifically, the assignment was to write a program to compute the sum of five integers. Furthermore, we had to prove the correctness of the simulation using MIPS instructions in MARS Simulator. Therefore, by using VHDL and creating components to simulate the MIPS instruction to get the sum of five integers we learn a lot about how the CPU works and how we can use VHDL to simulate things that may be happening in the CPU of our computers.