Lab : BEQ, BNE, J MIPS OPERATIONS

MdShahid Bin Emdad

April 24th, 2022

CSC 34300

TABLE OF CONTENTS

Objective: 2
Description of Specifications, and Functionality: 2
Project Directory: 2
Half adder: 3
Full adder: 4
Adder package: 5
32bit add/sub: 6
32-bit register: 7
Register file 8
MIF file: 10
1-bit comparator: 11
32-bit comparator: 12
Sign extension: 14
2_1_MUX: 15
Jump Calculation: 16
Controller: 17
Main Unit: 18
Testbench: 20
Simulation: 22
Conclusion: 25

Objective:

The objective of this assignment is to learn and understand the specifications of comparator and use it with registers and mif file to compute next address. We were instructed to implement MIPS instructions BEQ, BNE and J.

Description of Specifications, and Functionality:

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

Project Directory:

In figure 1, we can see the project directory for the beq, bne, j mips operations lab.

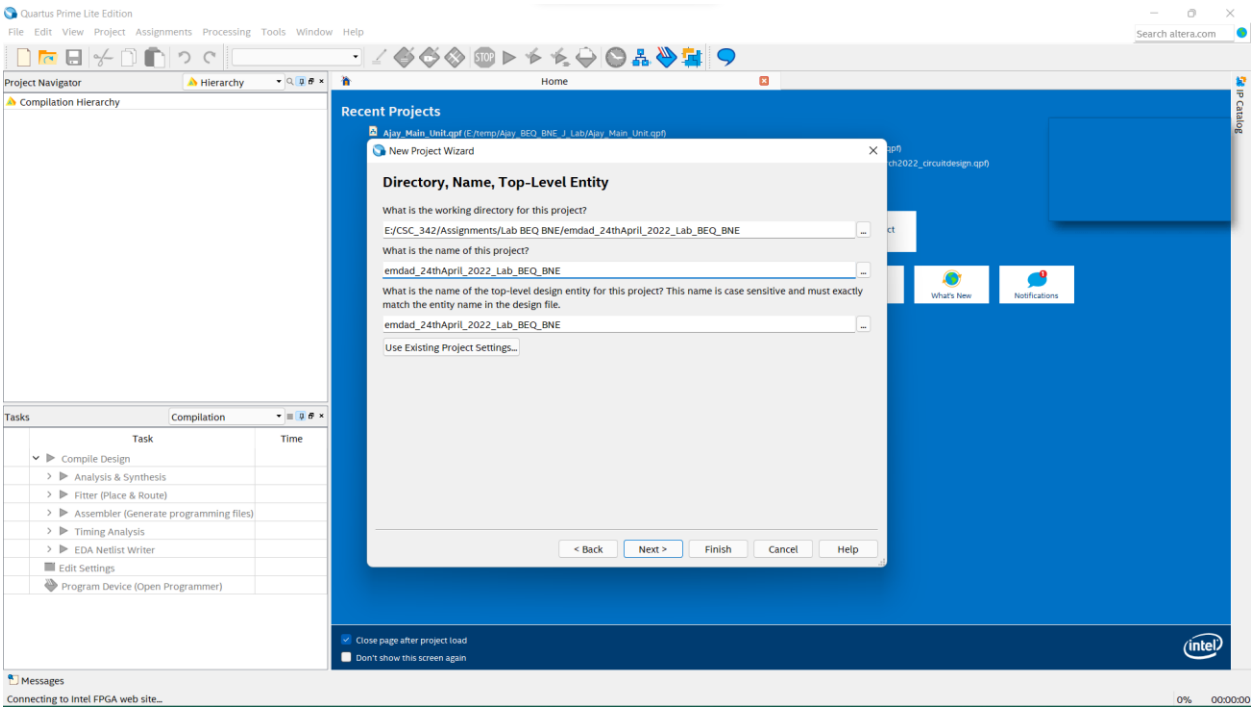


Figure 1: Project directory

In figure 2, we can see the project summary for the lab project.

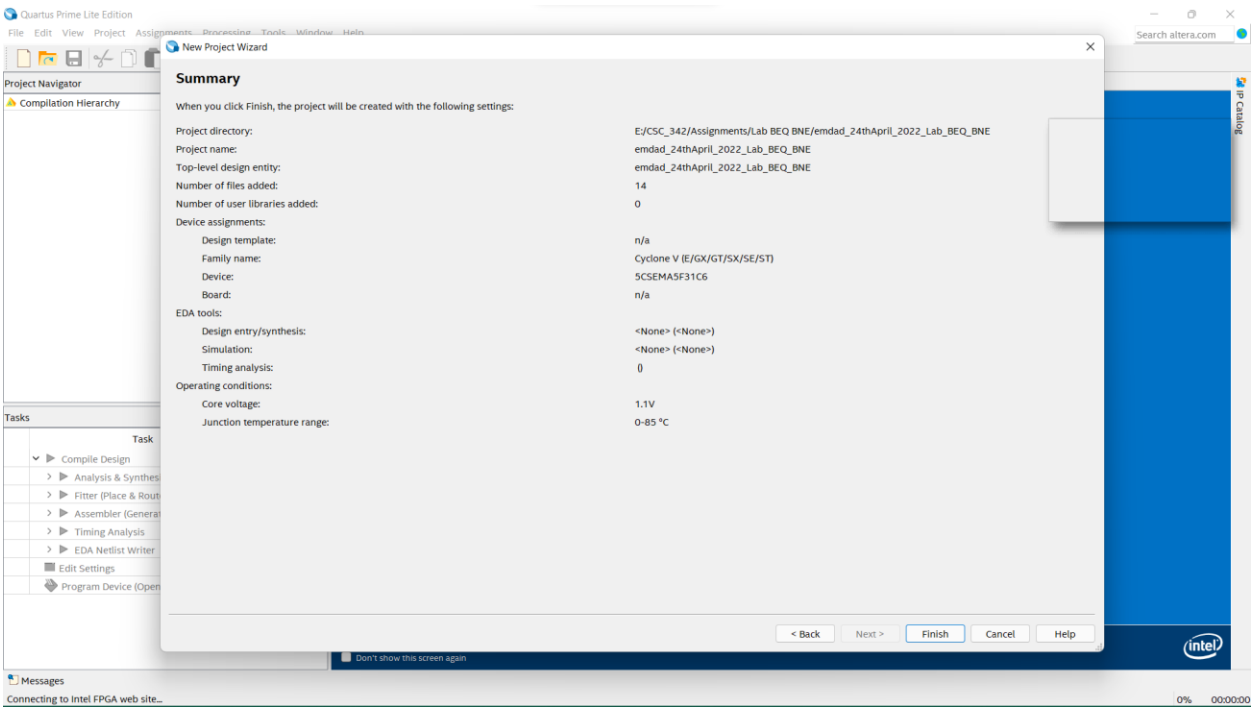


Figure 2: Project Summary

Half adder:

In figure 3, we can see the VHDL code for half adder and will be using this as a component to build full adder.

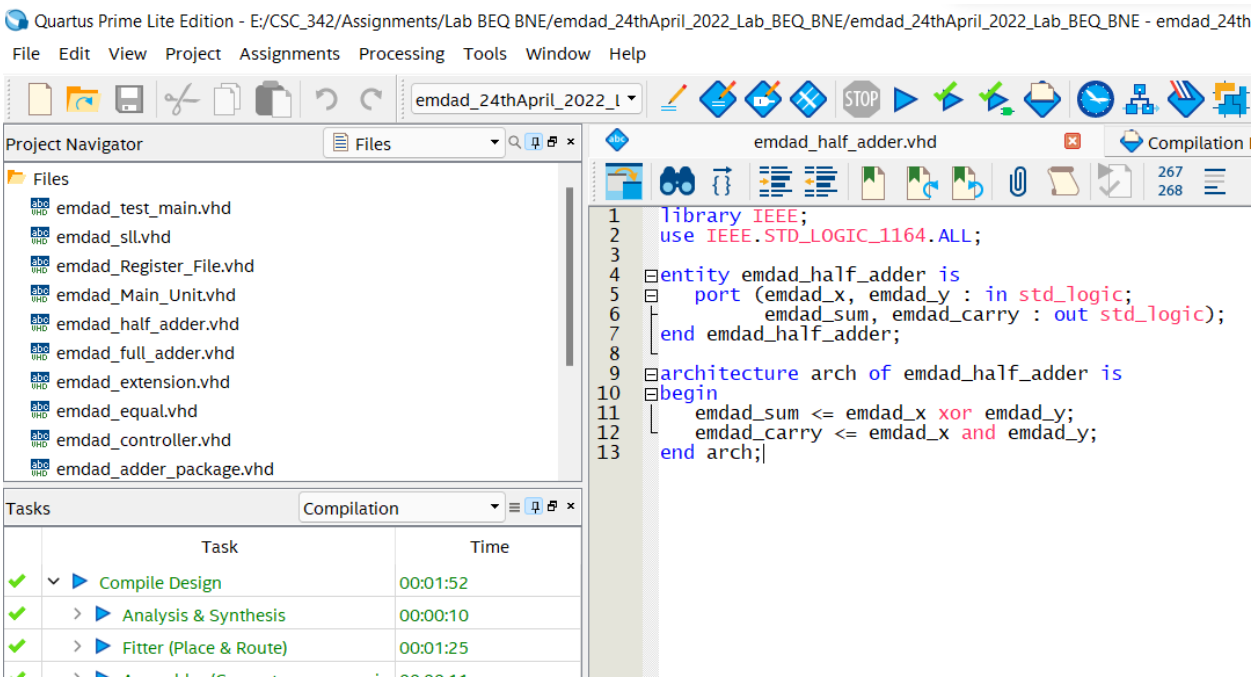


Figure 3: VHDL code for half adder

In figure 4, we can see that the VHDL code compiled successfully.

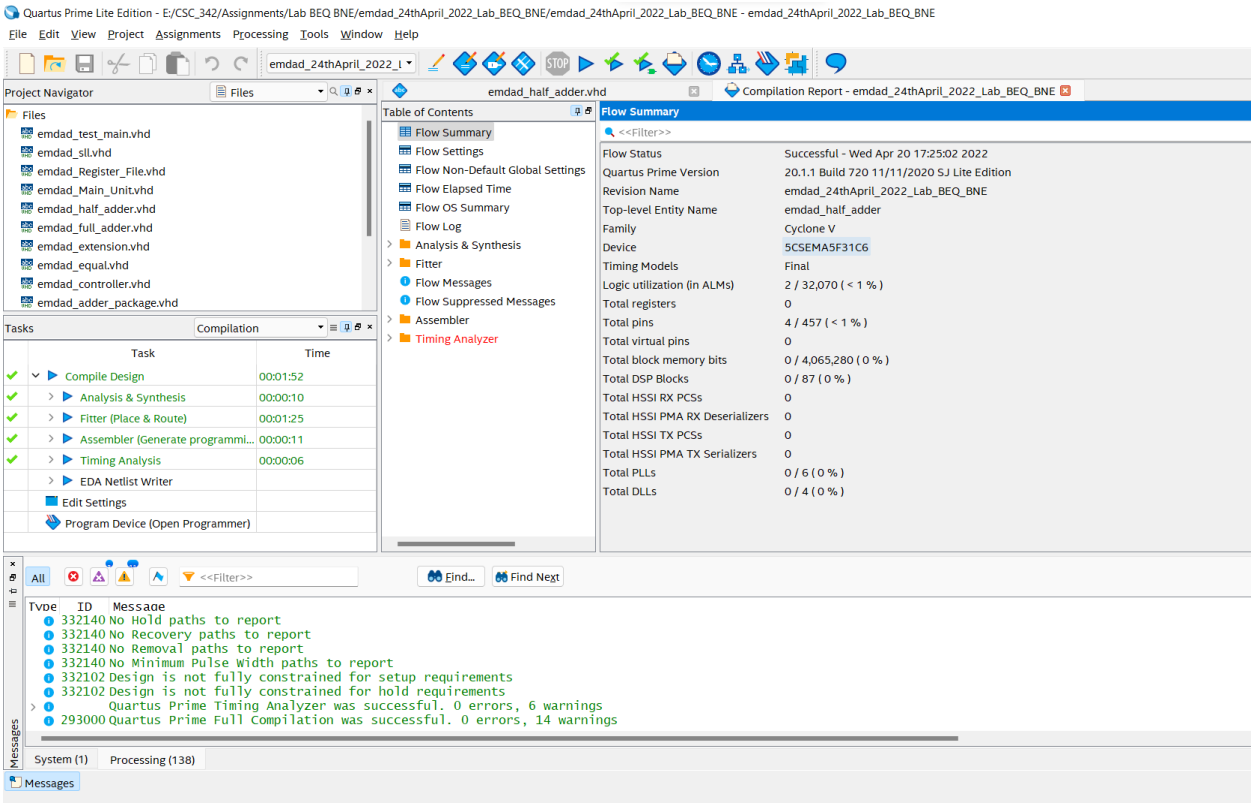


Figure 4: Successful compilation report

Full adder:

In figure 5, we can see the VHDL code for the full adder. I used half adder as a component.

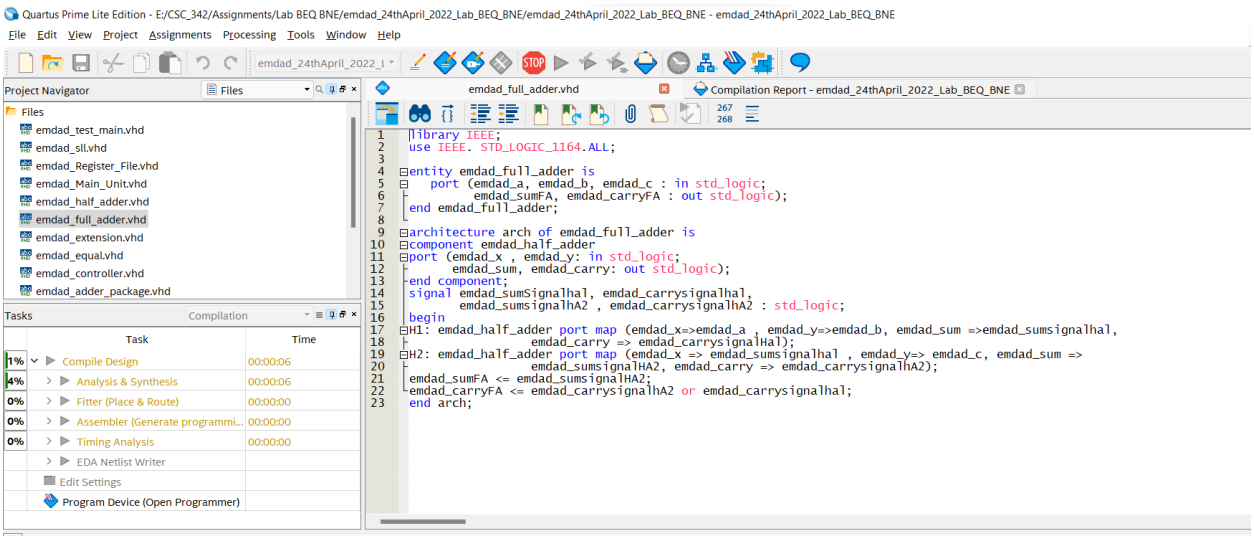


Figure 5: VHDL code for the full adder

In figure 6, we can see that the VHDL code compiled successfully.

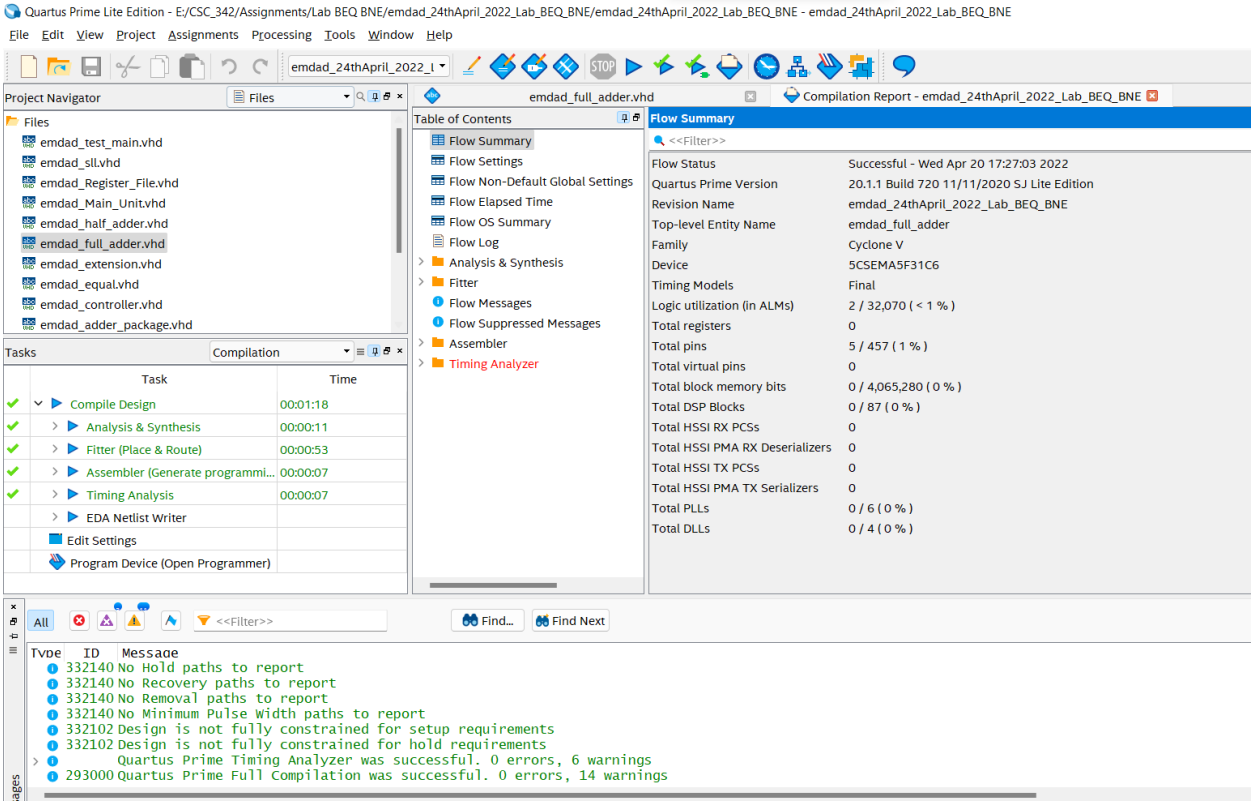


Figure 6: Successful compilation report.

Adder package:

In figure we can see that I am using both half and full adder as a component and creating the adder package to use it later in the project.

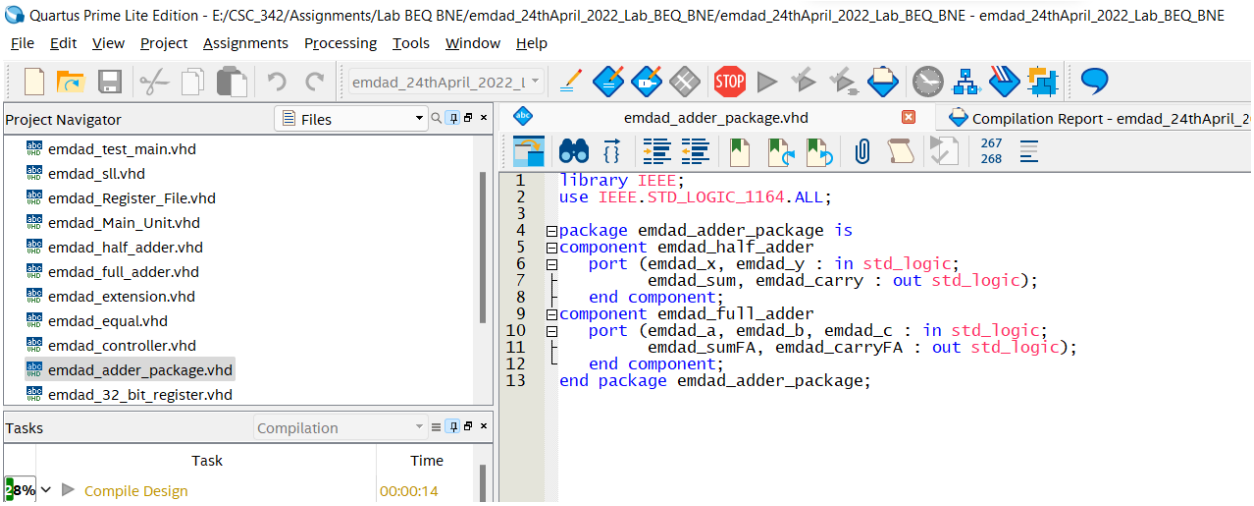


Figure 7: Adder package

In figure 8, we can see that the VHDL code compiled successfully.

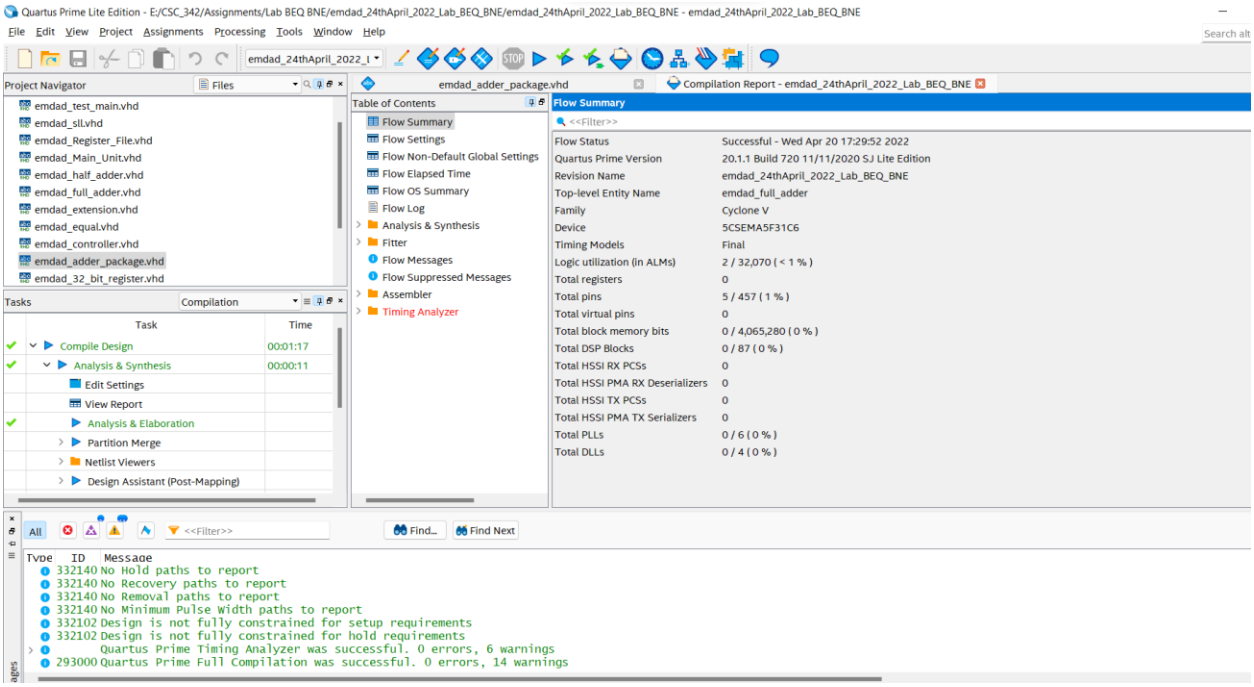


Figure 8: Successful compilation report.

32bit add/sub:

In figure 9, we can see the VHDL code for the 32 bit add/sub.

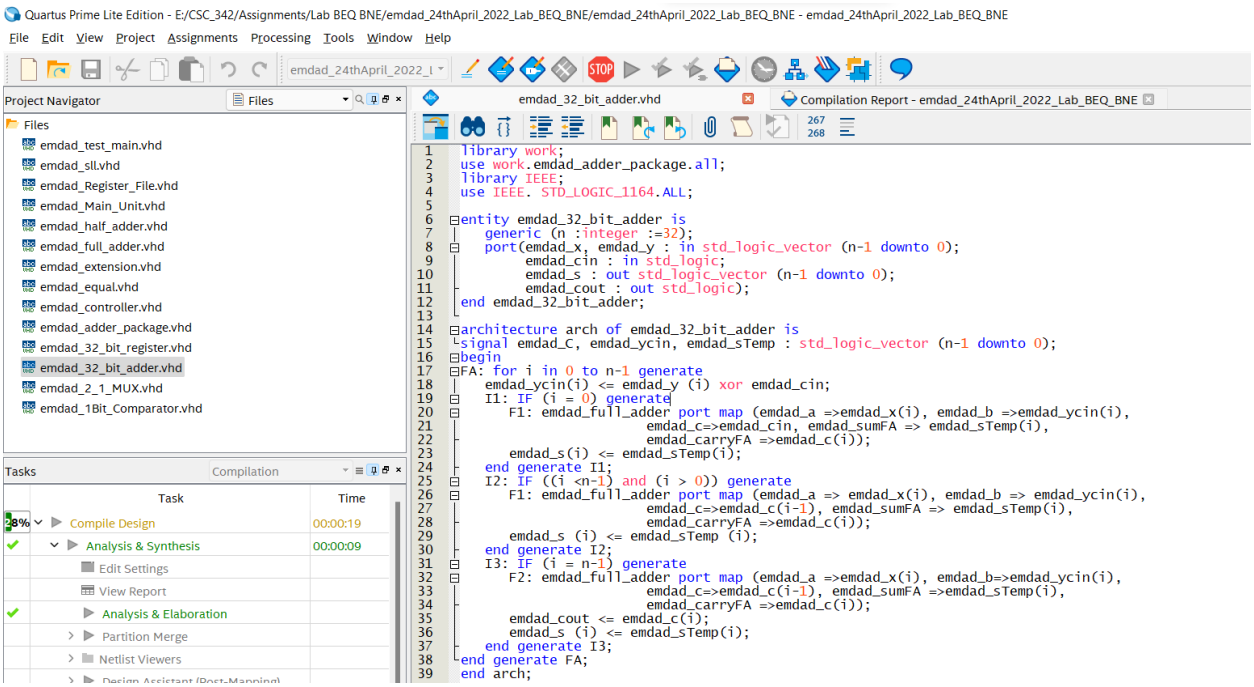


Figure 9: VHDL code for the 32 bit add/sub.

In figure 10, we can see that the VHDL code compiled successfully.

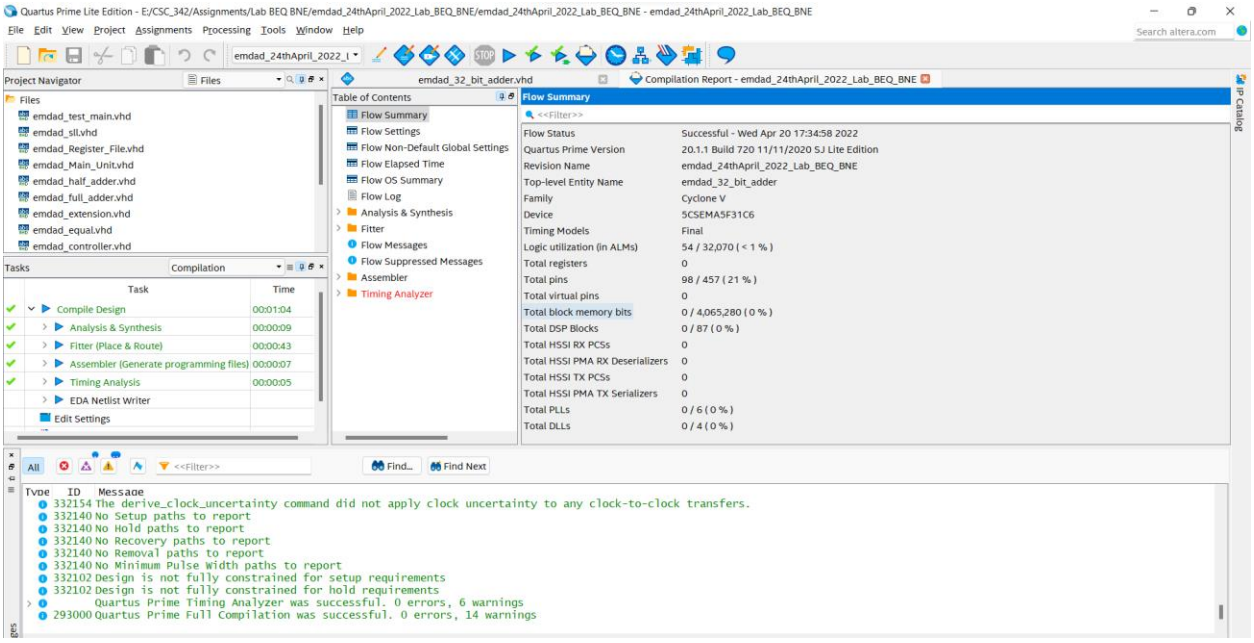


Figure 10: Successful compilation report.

32-bit register:

In figure 11, we can see the VHDL code for the 32-bit register. It will be used for PC register and instruction register.

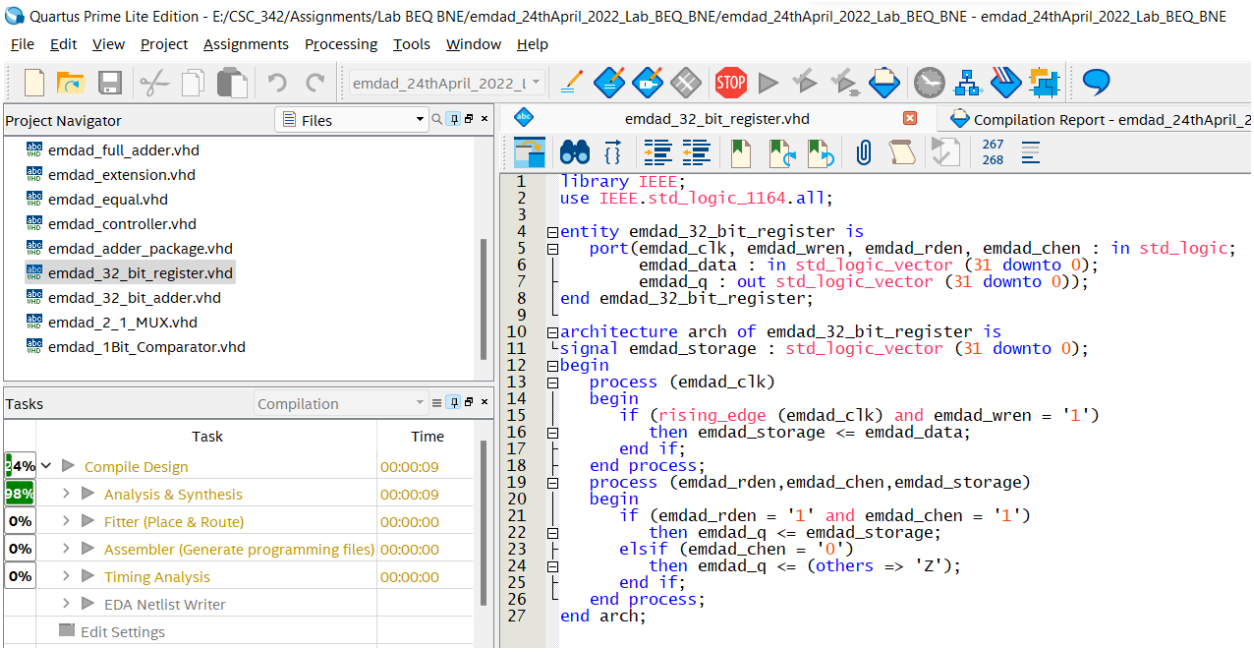


Figure 11: VHDL code for the 32-bit register

In figure 12, we can see that the VHDL code compiled successfully.

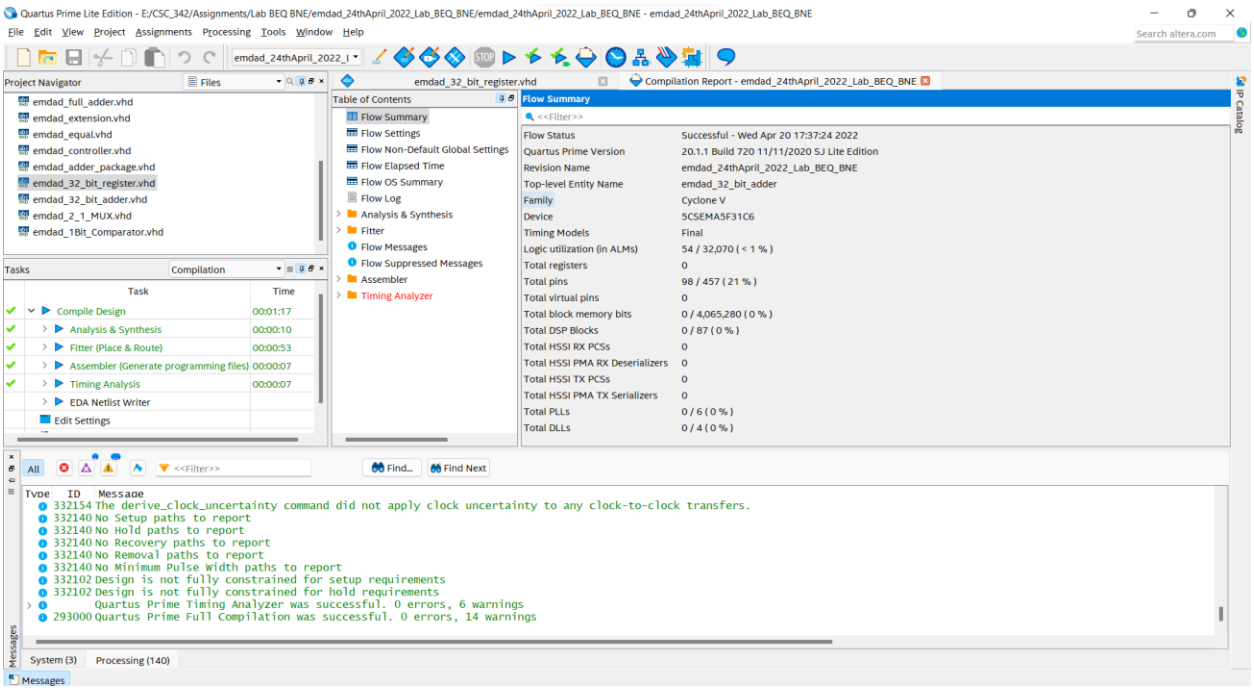


Figure 12: Successful compilation report.

Register file

In figure 13, we can see the VHDL code for register file. It is generated from LPM modules. I removed the extra comments to keep the code clean. I used 32-bit memory size and 32-bit registers. The wraddress is default to 0. We will be using this code to read the MIPS file for the inputs for rdaddress 1 and 2.

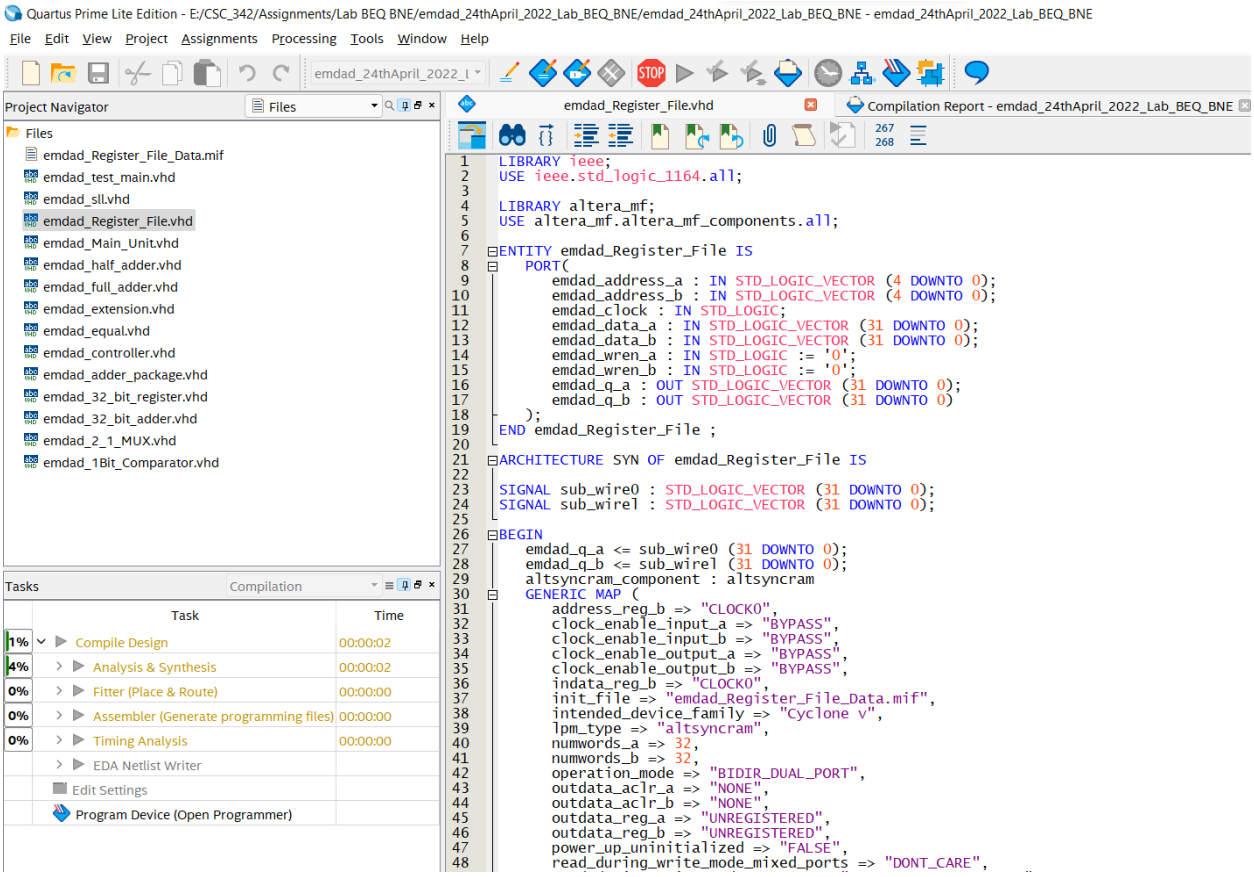


Figure 13: VHDL code for register file (Part 1)

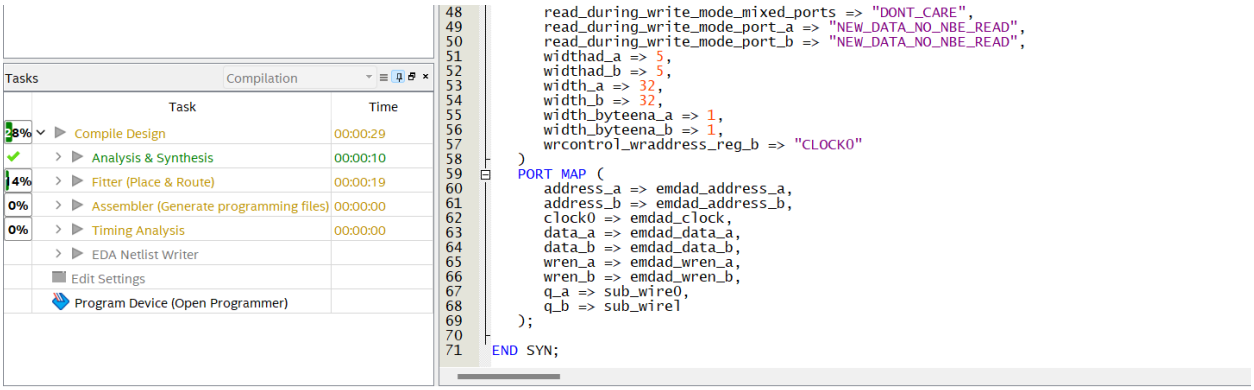


Figure 14: VHDL code for register file (Part 2)

In figure 15, we can see that the VHDL code compiled successfully.

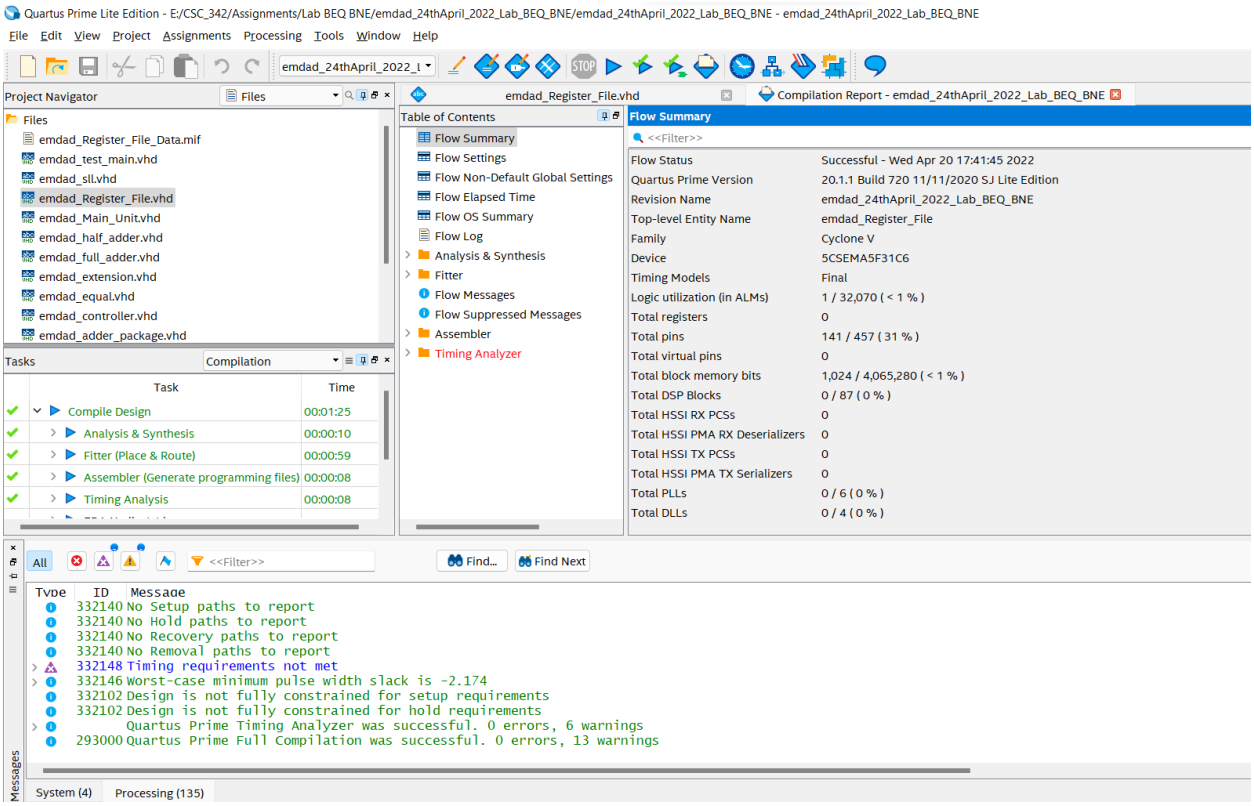


Figure 15: Successful compilation report.

MIF file:

In figure 16, we can see the MIPS file. I randomly entered some hexadecimal inputs for the simulation. I did memory size and register size both 32-bit.

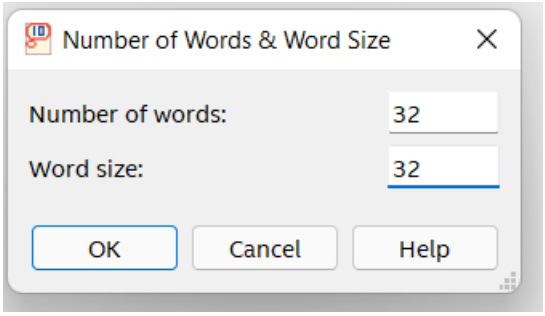


Figure 16: MIFS file specification

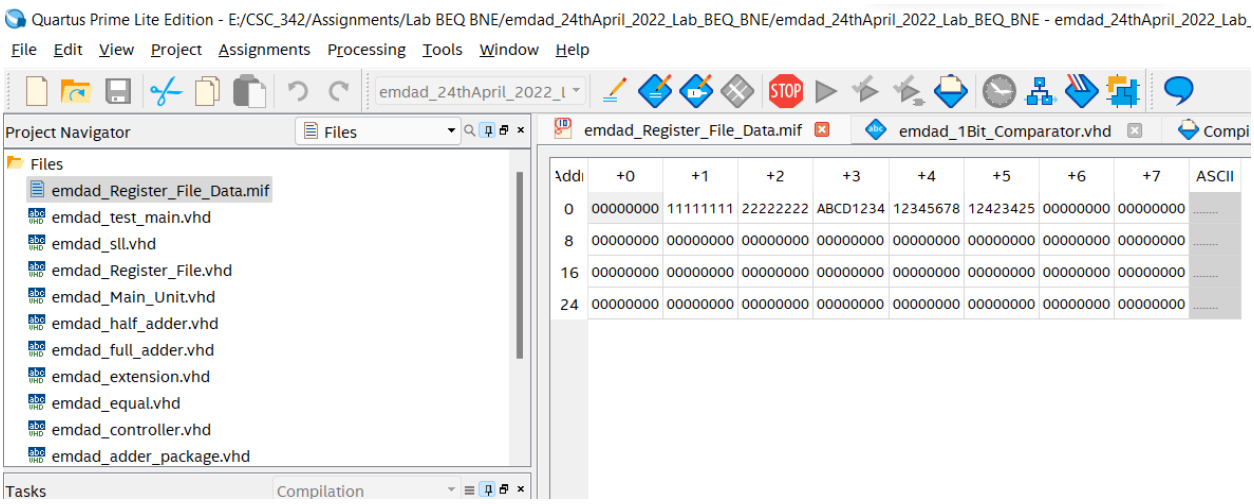


Figure 17: MIPS file for the simulation

1-bit comparator:

In figure 17, we can see the VHDL code for the 1-bit comparator. It will be used to compute the output for BNE and BEQ.

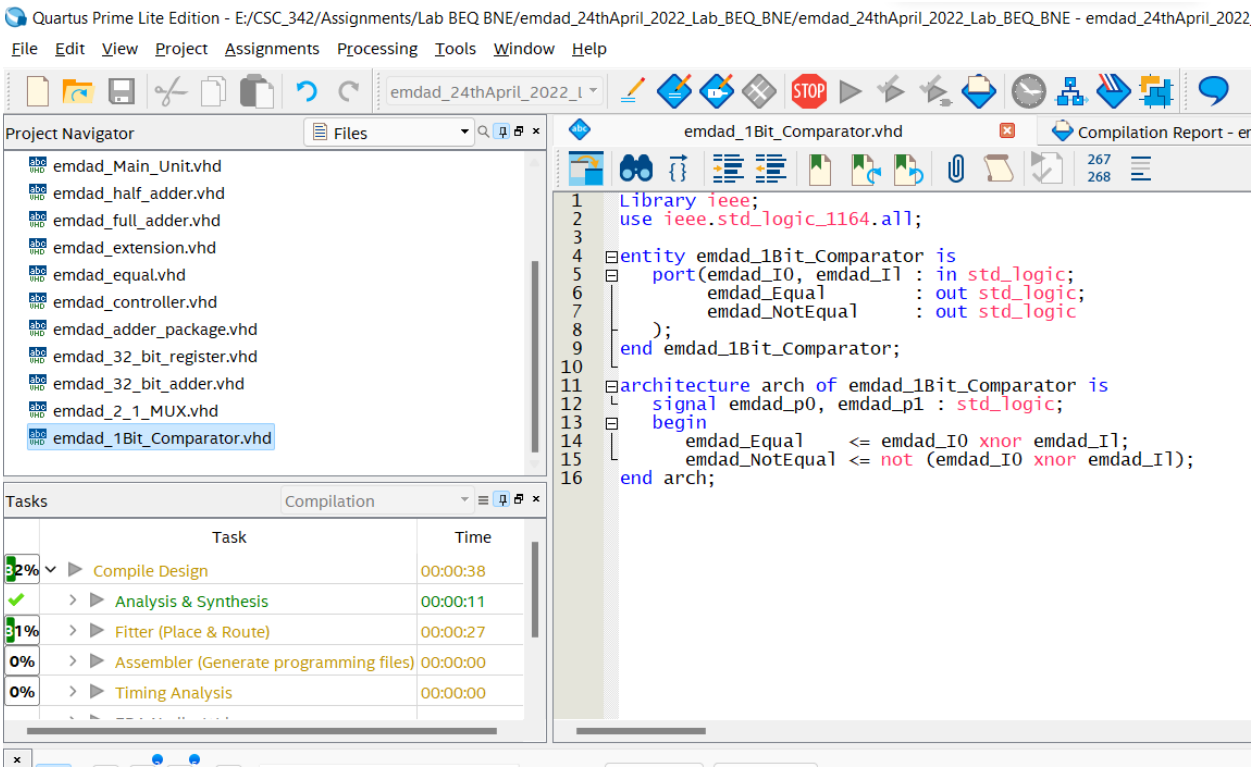


Figure 18: VHDL code for the 1-bit comparator

In figure 19, we can see that the VHDL code compiled successfully.

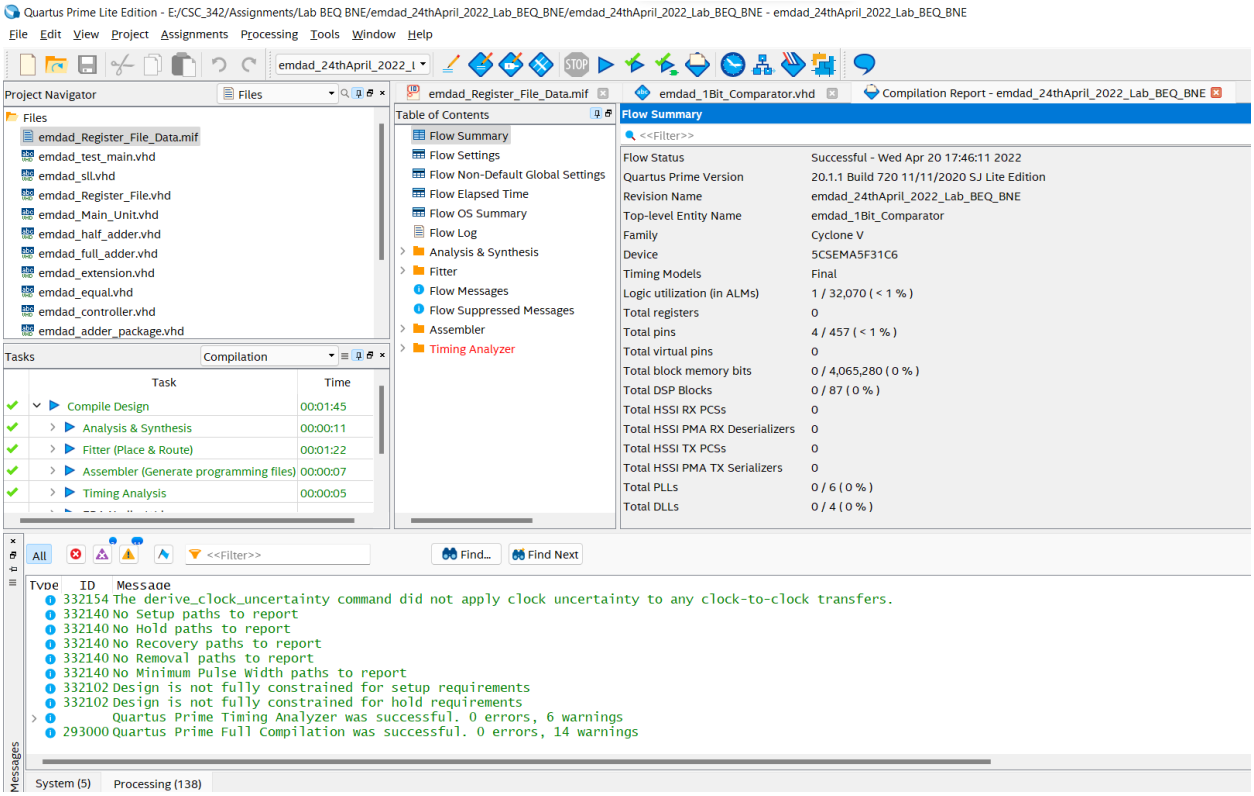


Figure 19: Successful compilation report.

32-bit comparator:

In figure 20, we can see the vhdl code for the 32-bit comparator. The file computes the equal and non-equal for each bit. The when else returns the output on condition for BEQ and BNE.

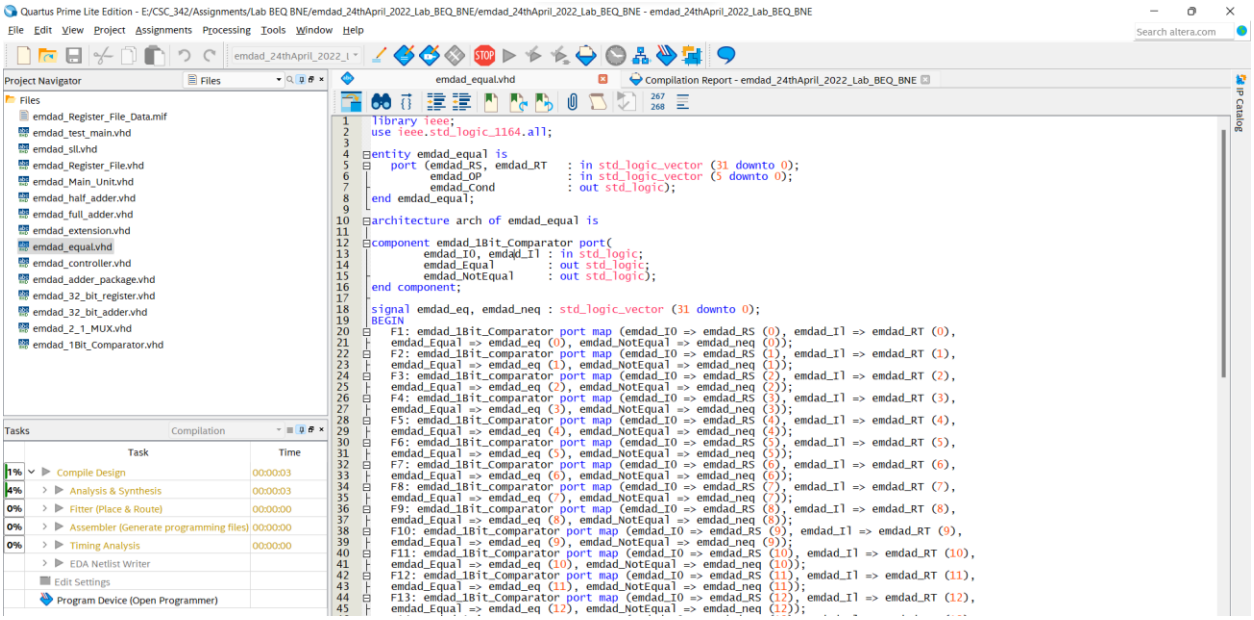


Figure 20: VHDL code for 32-bit comparator (part 1)

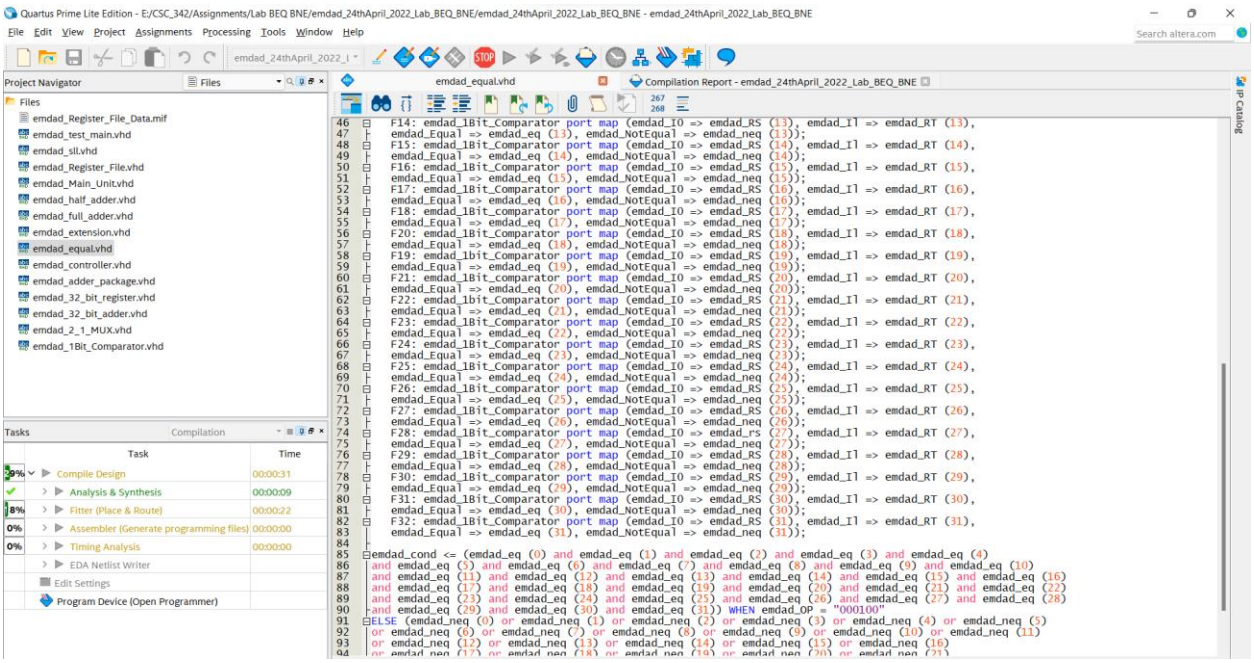


Figure 21: VHDL code for 32-bit comparator (part 2)

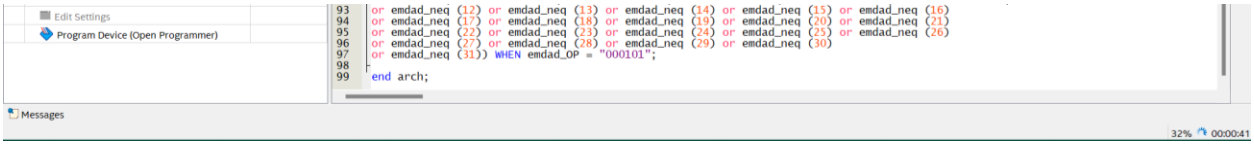


Figure 22: VHDL code for 32-bit comparator (part 3)

In figure 23, we can see that the VHDL code compiled successfully.

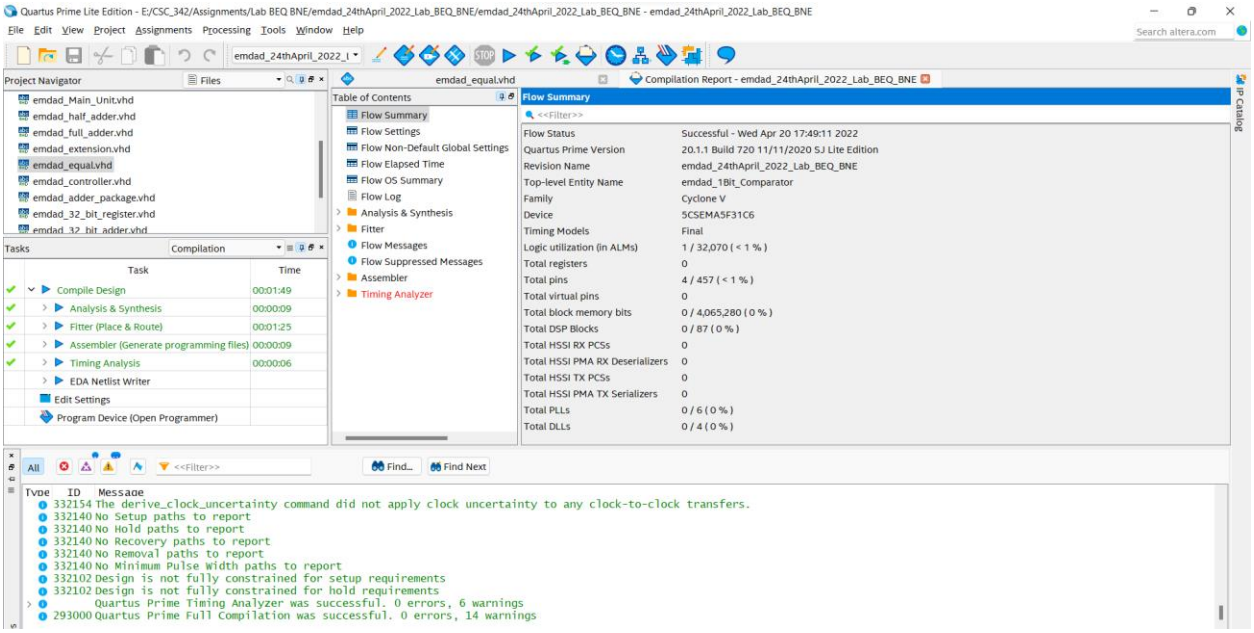


Figure 23: Successful compilation report.

Sign extension:

In figure 24, we vhdl code for sign extension. I am using this to extend the most significant bit for IMM. It is used to extend the 16bit register data to 32bit register so I can use the 32-bit add/sub

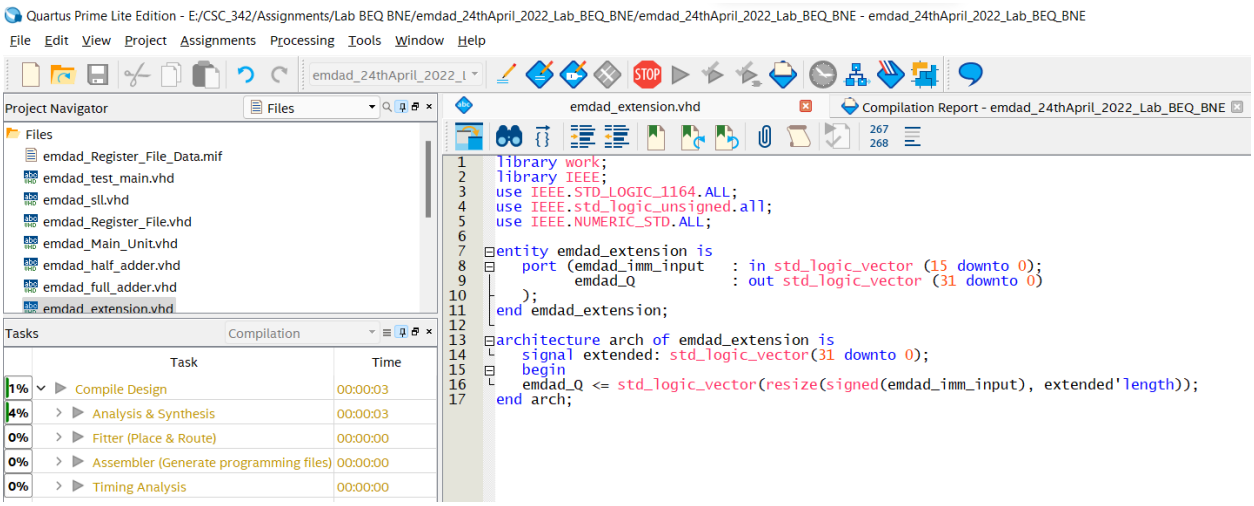


Figure 24: vhdl code for sign extension

In figure 25, we can see that the VHDL code compiled successfully.

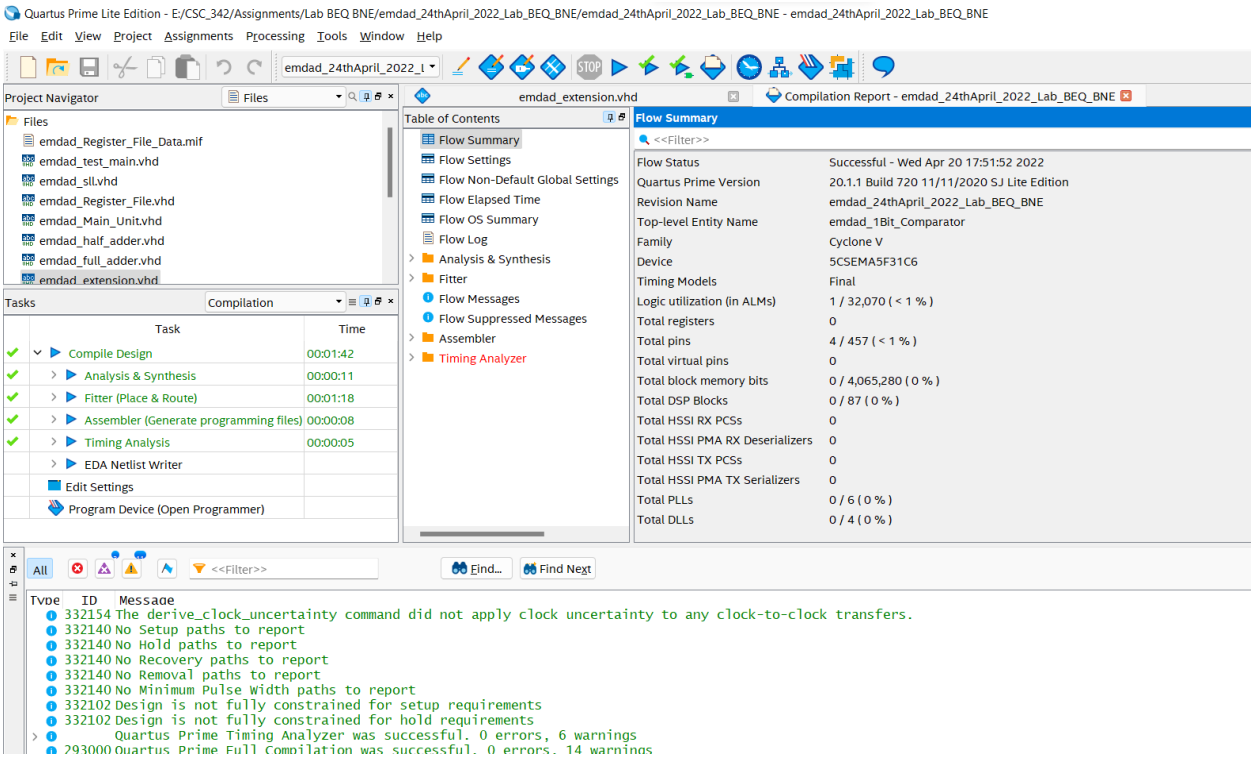


Figure 25: Successful compilation report.

2_1_MUX:

In figure 17, we can see the VHDL code for the 2:1 mux. This code decides for output to return PC + 4 or PC + +SIGN EXTENDEC IMM.

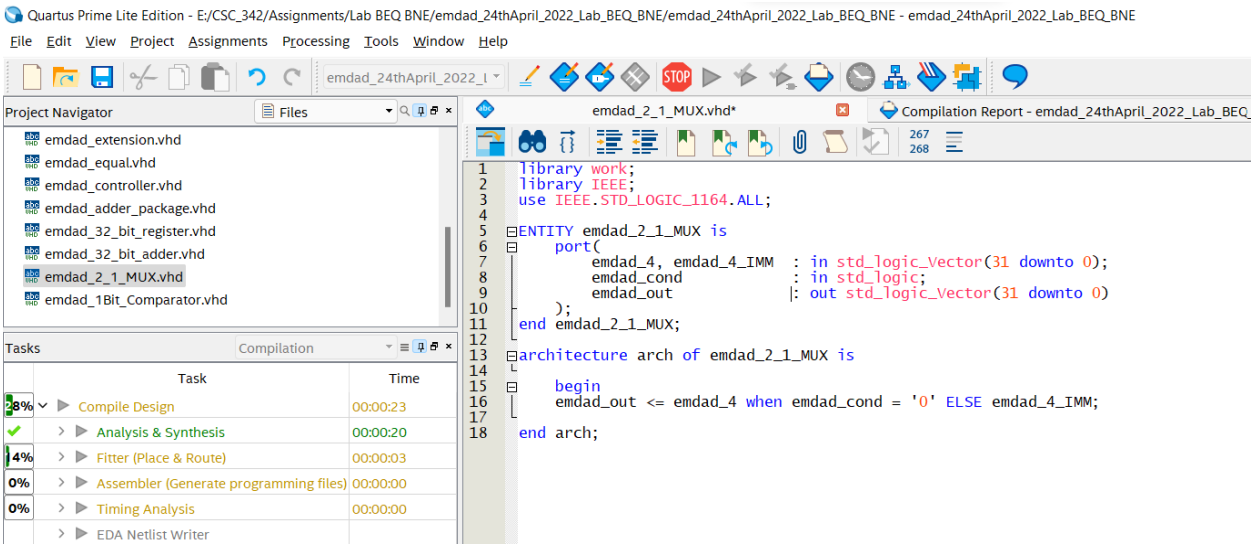


Figure 26: VHDL code for the 2:1 mux

In figure 27, we can see that the VHDL code compiled successfully.

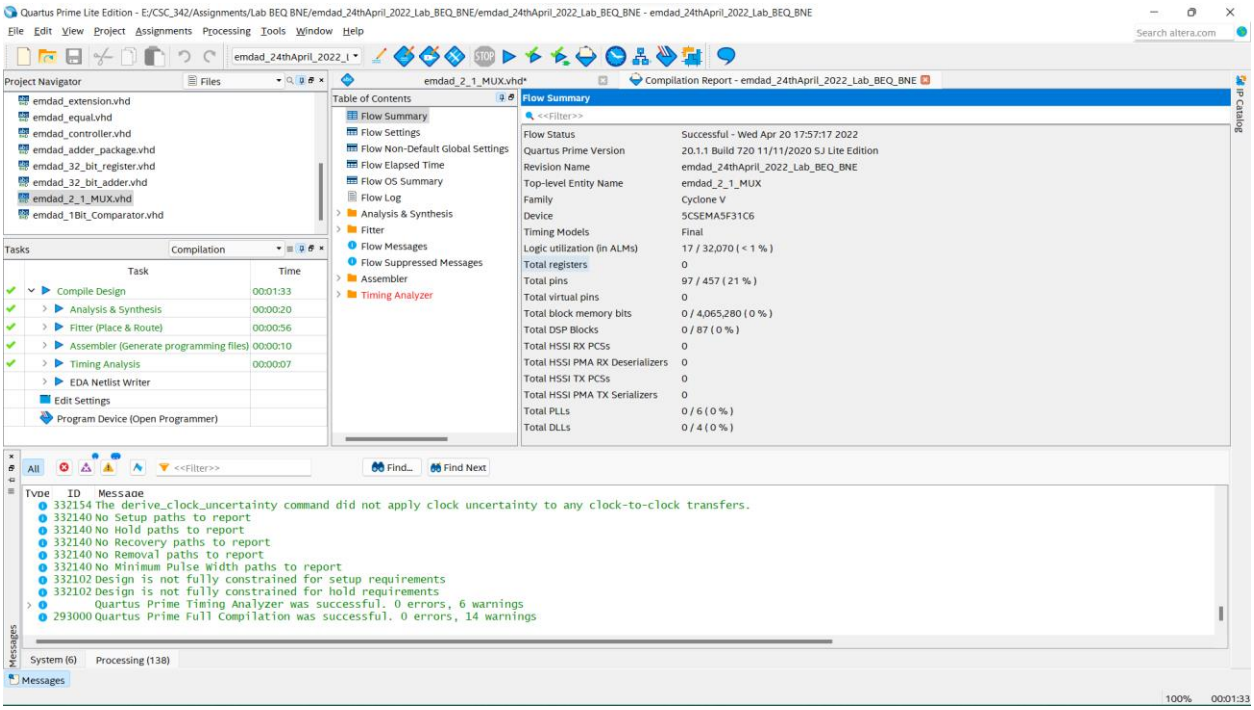


Figure 27: Successful compilation report.

Jump Calculation:

In figure 28, we can see the VHDL code for jump calculation. This code requires the input of the 4 most significant bits of PC and 26 least significant bits.

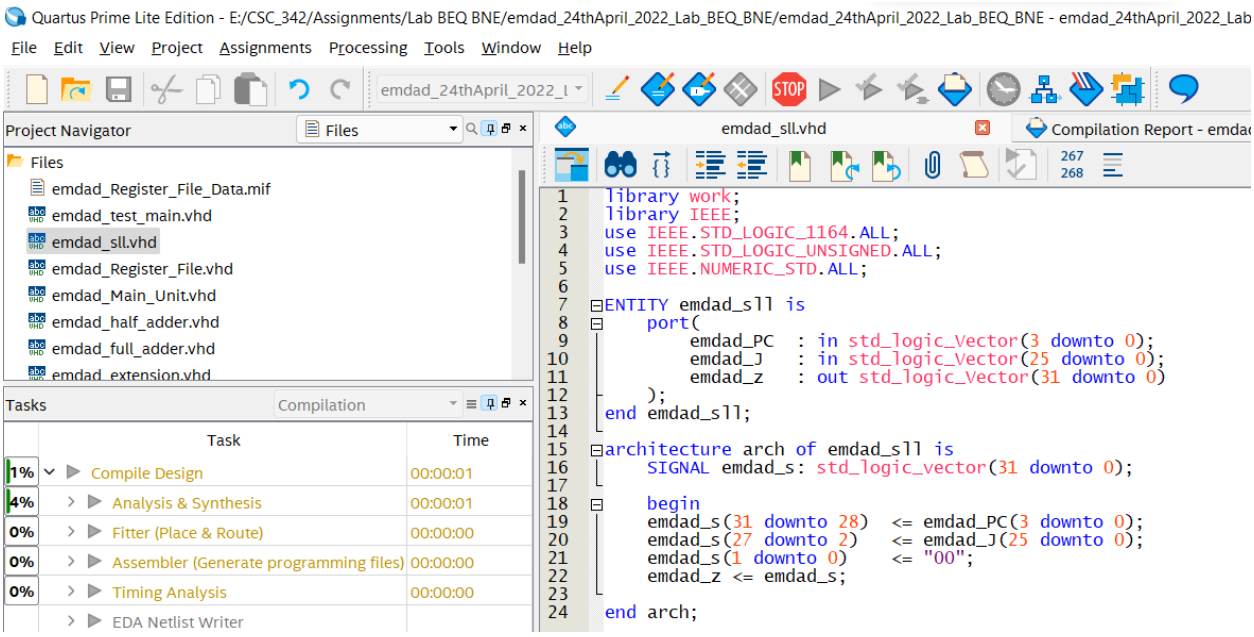


Figure 28: VHDL code for jump calculation

In figure 29, we can see that the VHDL code compiled successfully.

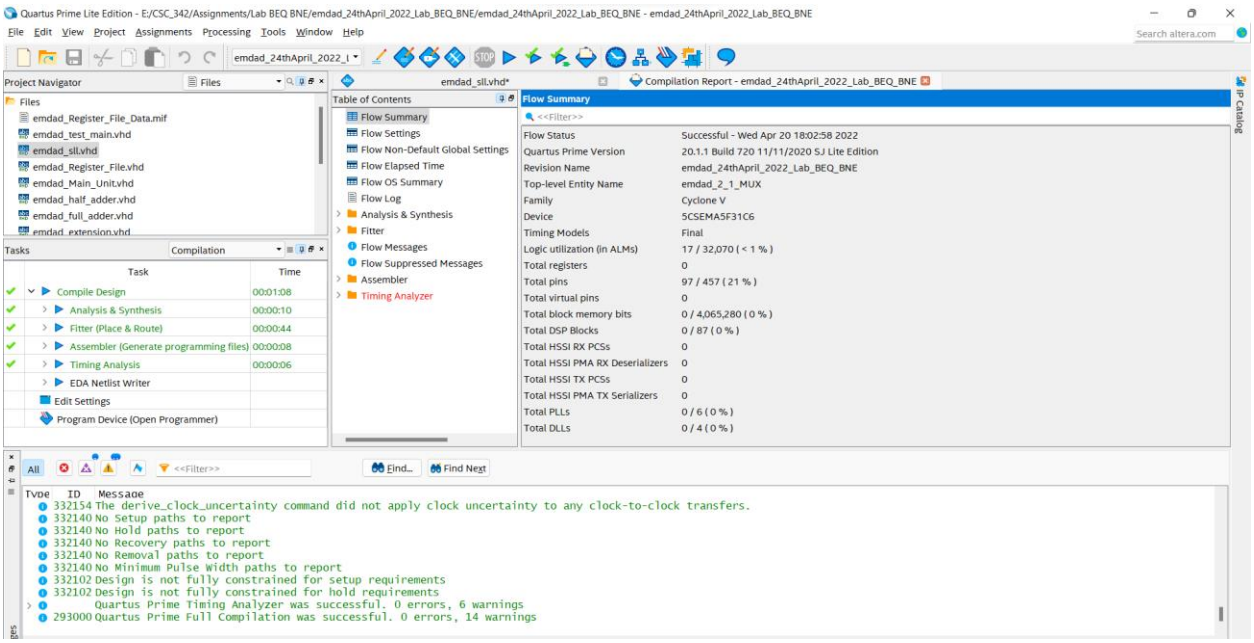


Figure 29: Successful compilation report.

Controller:

In figure 30, we can see the VHDL code for the controller. It designates the input date for the 32bit PC register, it selects the right output based on the OP code from instruction register.

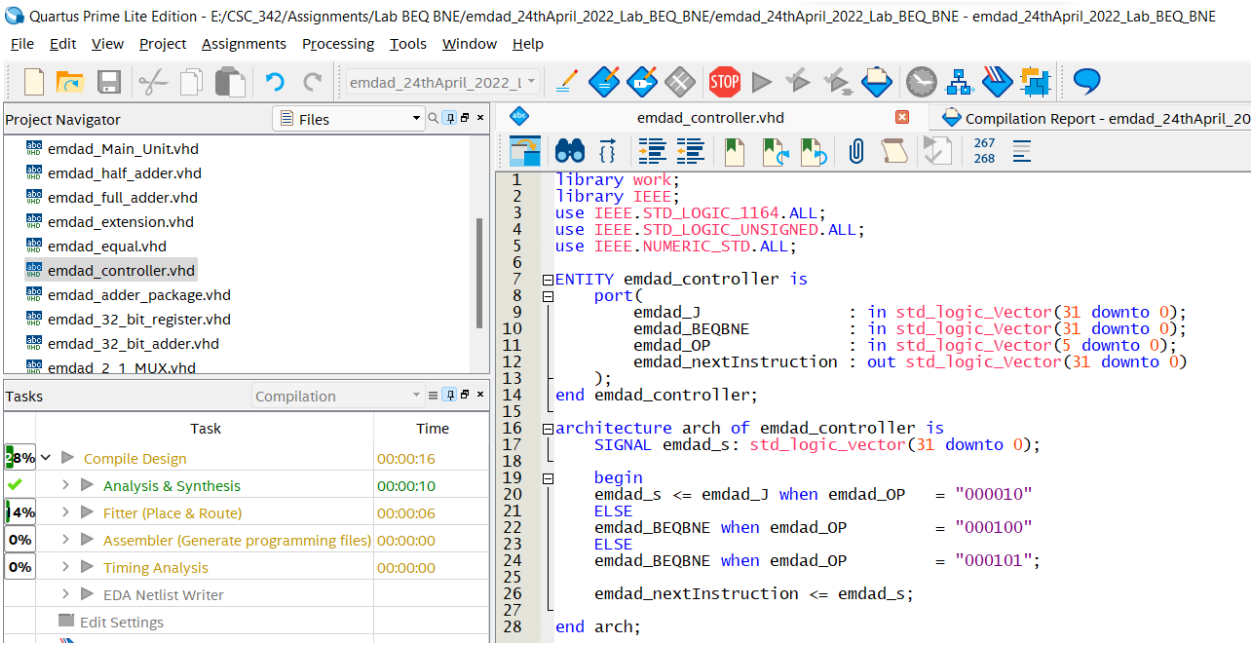


Figure 30: VHDL code for the controller

In figure 31, we can see that the VHDL code compiled successfully.

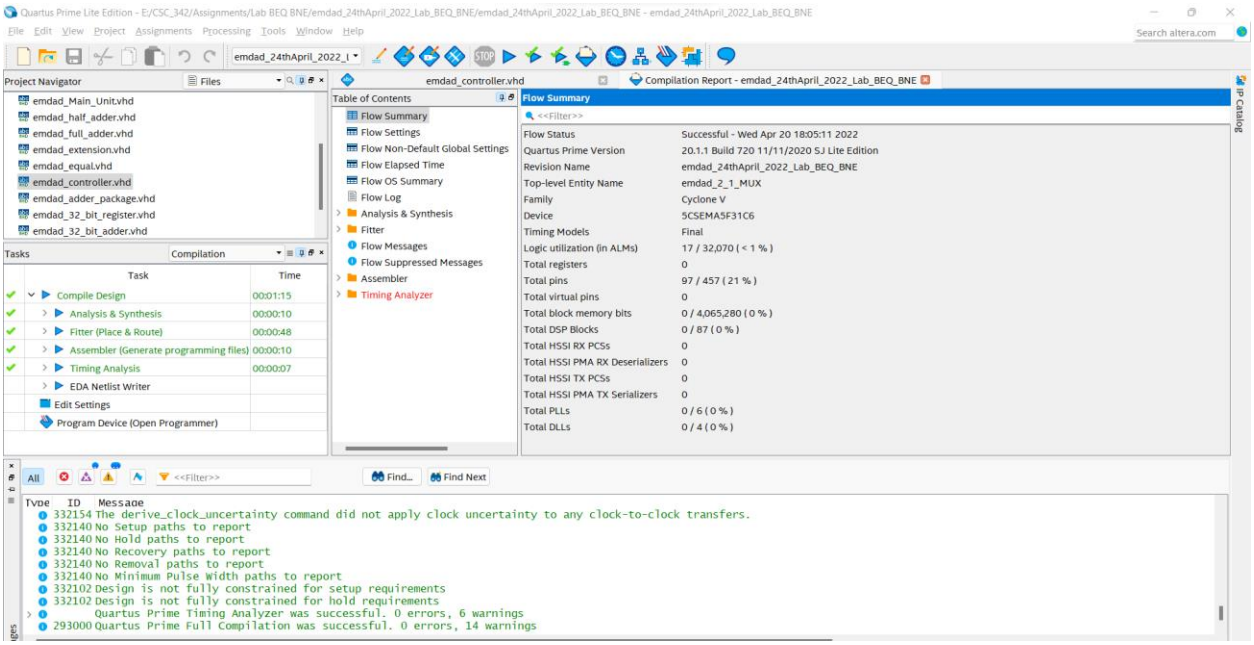
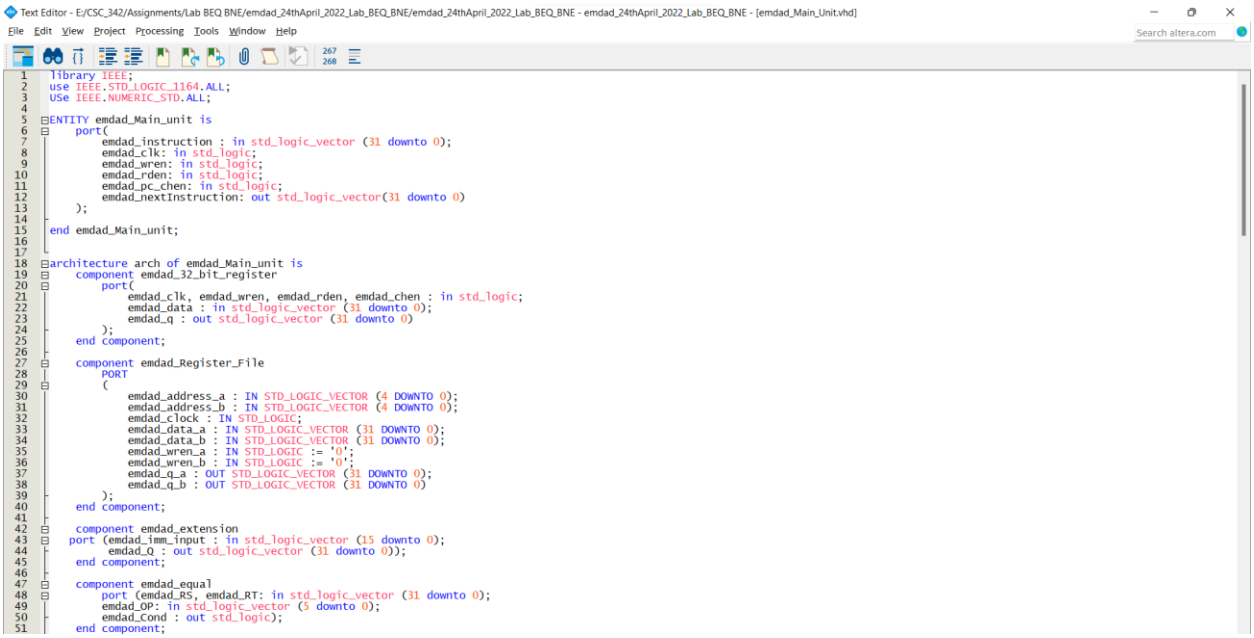


Figure 31: Successful compilation report.

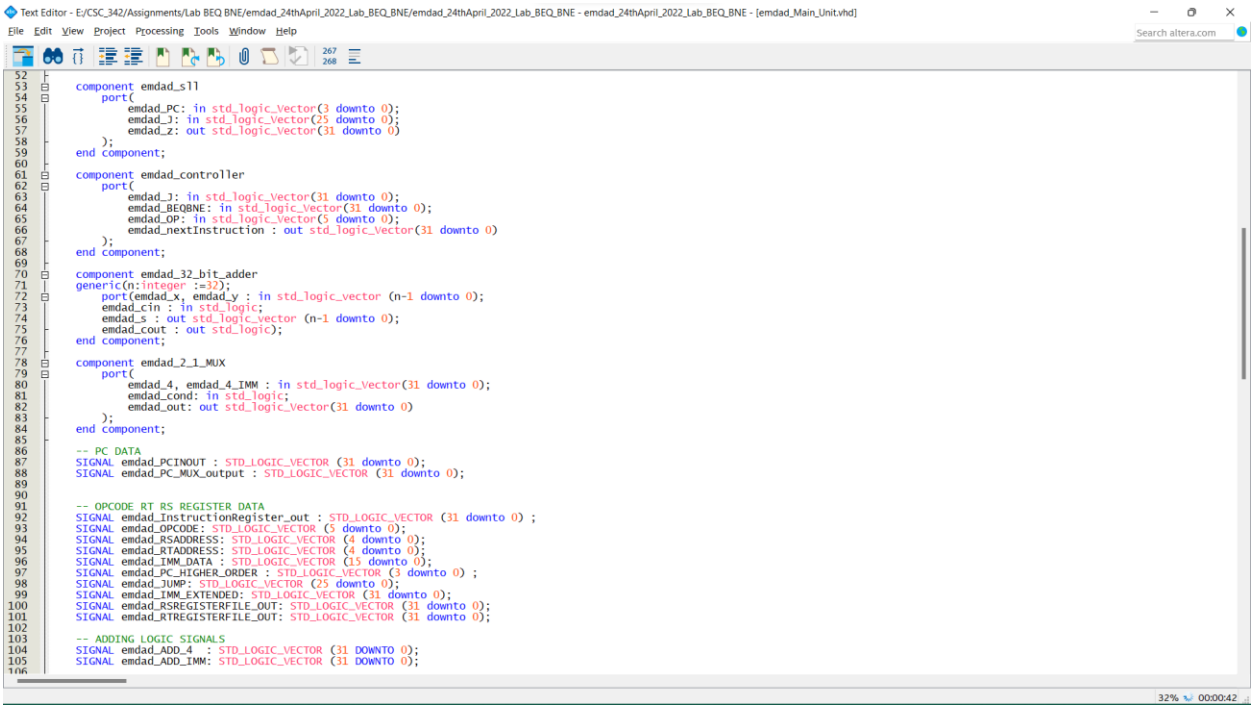
Main Unit:

In figure 32, we can see the VHDL code for main unit, has other VHDL codes as component.



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
Use IEEE.NUMERIC_STD.ALL;

ENTITY emdad_Main_unit is
    port(
        emdad_instruction : in std_logic_vector (31 downto 0);
        emdad_clk: in std_logic;
        emdad_wren: in std_logic;
        emdad_rden: in std_logic;
        emdad_pc_chen: in std_logic;
        emdad_nextInstruction: out std_logic_vector(31 downto 0)
    );

end emdad_Main_unit;

architecture arch of emdad_Main_unit is
    component emdad_32_bit_register
        port(
            emdad_clk, emdad_wren, emdad_rden, emdad_chen : in std_logic;
            emdad_data : in std_logic_vector (31 downto 0);
            emdad_q : out std_logic_vector (31 downto 0)
        );
    end component;

    component emdad_Register_File
        PORT
        (
            emdad_address_a : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
            emdad_address_b : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
            emdad_clock : IN STD_LOGIC;
            emdad_data_a : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            emdad_data_b : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            emdad_wren_a : IN STD_LOGIC := '0';
            emdad_wren_b : IN STD_LOGIC := '0';
            emdad_q_a : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
            emdad_q_b : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
        );
    end component;

    component emdad_extension
   port (emdad_imm_input : in std_logic_vector (15 downto 0);
            emdad_Q : out std_logic_vector (31 downto 0));
    end component;

    component emdad_equal
        port (emdad_RS, emdad_RT: in std_logic_vector (31 downto 0);
        emdad_OP: in std_logic_vector (5 downto 0);
        emdad_Cond : out std_logic);
    end component;
```

Figure 32: VHDL code for main unit (part 1)



```
    component emdad_sll
        port(
            emdad_PC: in std_logic_Vector(3 downto 0);
            emdad_J: in std_logic_Vector(25 downto 0);
            emdad_z: out std_logic_Vector(31 downto 0)
        );
    end component;

    component emdad_controller
        port(
            emdad_J: in std_logic_Vector(31 downto 0);
            emdad_BEQBNE: in std_logic_Vector(31 downto 0);
            emdad_OP: in std_logic_Vector(5 downto 0);
            emdad_nextInstruction : out std_logic_Vector(31 downto 0)
        );
    end component;

    component emdad_32_bit_adder
    generic(n:integer :=32);
        port(emdad_x, emdad_y : in std_logic_vector (n-1 downto 0);
        emdad_cin : in std_logic;
        emdad_s : out std_logic_vector (n-1 downto 0);
        emdad_cout : out std_logic);
    end component;

    component emdad_2_1_MUX
        port(
            emdad_4, emdad_4_IMM : in std_logic_Vector(31 downto 0);
            emdad_cond: in std_logic;
            emdad_out: out std_logic_Vector(31 downto 0)
        );
    end component;

    -- PC DATA
    SIGNAL emdad_PCINOUT : STD_LOGIC_VECTOR (31 downto 0);
    SIGNAL emdad_PC_MUX_output : STD_LOGIC_VECTOR (31 downto 0);


    -- OPCODE RT RS REGISTER DATA
    SIGNAL emdad_InstructionRegister_out : STD_LOGIC_VECTOR (31 downto 0) ;
    SIGNAL emdad_OPCODE: STD_LOGIC_VECTOR (5 downto 0);
    SIGNAL emdad_RSADDRESS: STD_LOGIC_VECTOR (4 downto 0);
    SIGNAL emdad_RTADDRESS: STD_LOGIC_VECTOR (4 downto 0);
    SIGNAL emdad_IMM_DATA : STD_LOGIC_VECTOR (15 downto 0);
    SIGNAL emdad_PC_HIGHER_ORDER : STD_LOGIC_VECTOR (3 downto 0) ;
    SIGNAL emdad_JUMP: STD_LOGIC_VECTOR (25 downto 0);
    SIGNAL emdad_IMM_EXTENDED: STD_LOGIC_VECTOR (31 downto 0);
    SIGNAL emdad_RSREGISTERFILE_OUT: STD_LOGIC_VECTOR (31 downto 0);
    SIGNAL emdad_RTREGISTERFILE_OUT: STD_LOGIC_VECTOR (31 downto 0);

    -- ADDING LOGIC SIGNALS
    SIGNAL emdad_ADD_4  : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL emdad_ADD_IMM: STD_LOGIC_VECTOR (31 DOWNTO 0);
```

Figure 33: VHDL code for main unit (part 2)

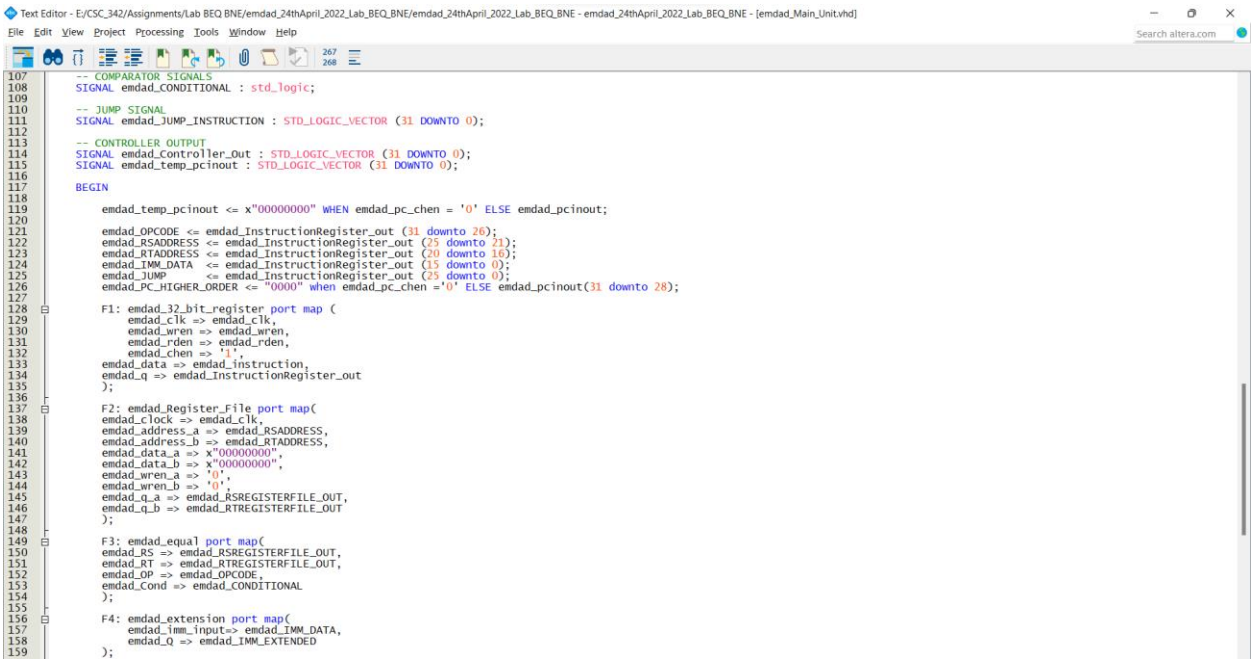


Figure 34: VHDL code for main unit (part 3)

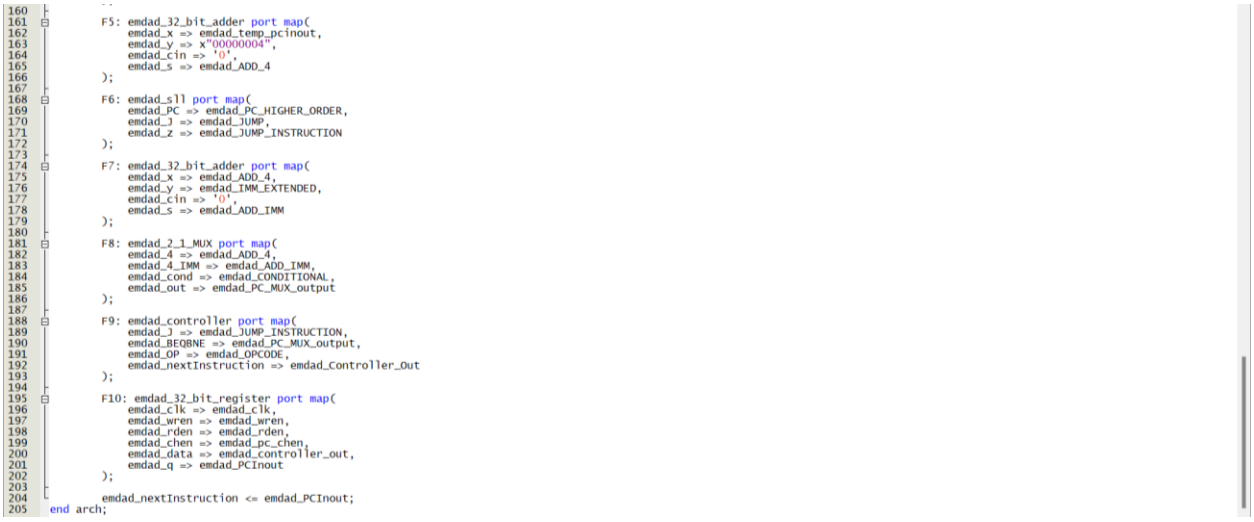


Figure 35: VHDL code for main unit (part 4)

F1 is 32-bit register, stores the instructions from input and output is instructionRegister_Out. F2 is register file component. The RSADDRESS and RTADDRESS receive address date from 32-bit instruction register and output 32bit values. The address data loaded from .mif file. F3 is comparator component output the condition based on the OPCODE BEQ or BNE. F4 is the extension component to compute the 32-bit extension of IMM field to store from the instructions register. F5 is the 32bit add/sub and add the PC register output with 4. F6 is jump calculation and it takes higher order bits and 26 least significant bits. F7 is takes the output from F5 and extend it from F4. F9 is controller to decide which input to select for 32-bit register. F10 is 32-bit register.

In figure 36, we can see that the VHDL code compiled successfully.

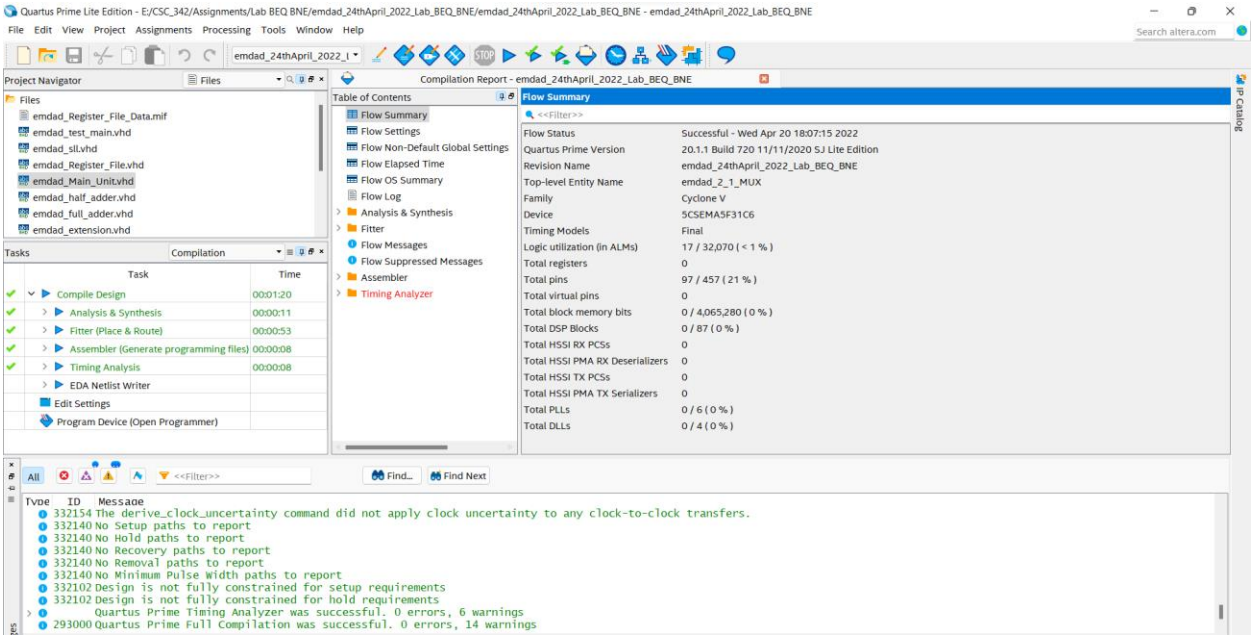


Figure 36: Successful compilation report.

Testbench:

In figure 37, we can see the VHDL code for testbench. It is to get waves on ModelSim and test my code validation.

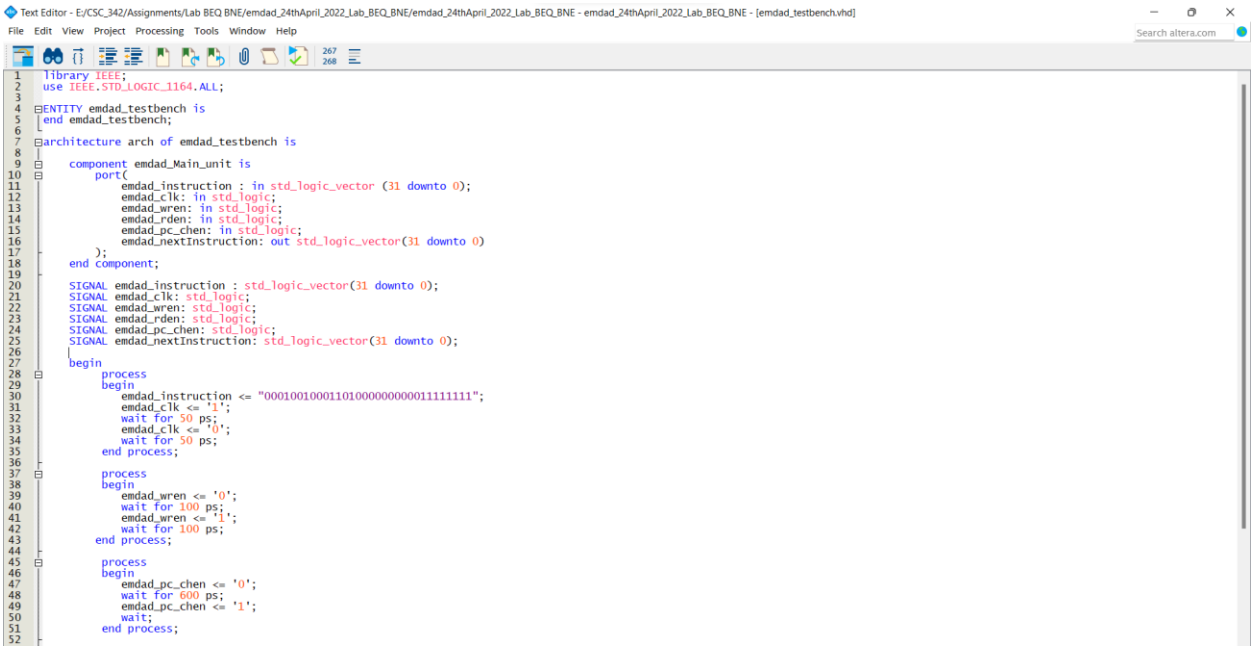


Figure 37: VHDL code for testbench (Part 1)

```
        process
        begin
            emdad_rden <= '0';
            wait for 200 ps;
            emdad_rden <= '1';
            wait for 200 ps;
        end process;

        F1: emdad_Main_unit port map(
            emdad_instruction => emdad_instruction,
            emdad_clk => emdad_clk,
            emdad_wren => emdad_wren,
            emdad_rden => emdad_rden,
            emdad_pc_chen => emdad_pc_chen,
            emdad_nextInstruction => emdad_nextInstruction
        );

end arch;
```

Figure 38: VHDL code for testbench (Part 2)

In figure 39, we can see that the VHDL code compiled successfully.

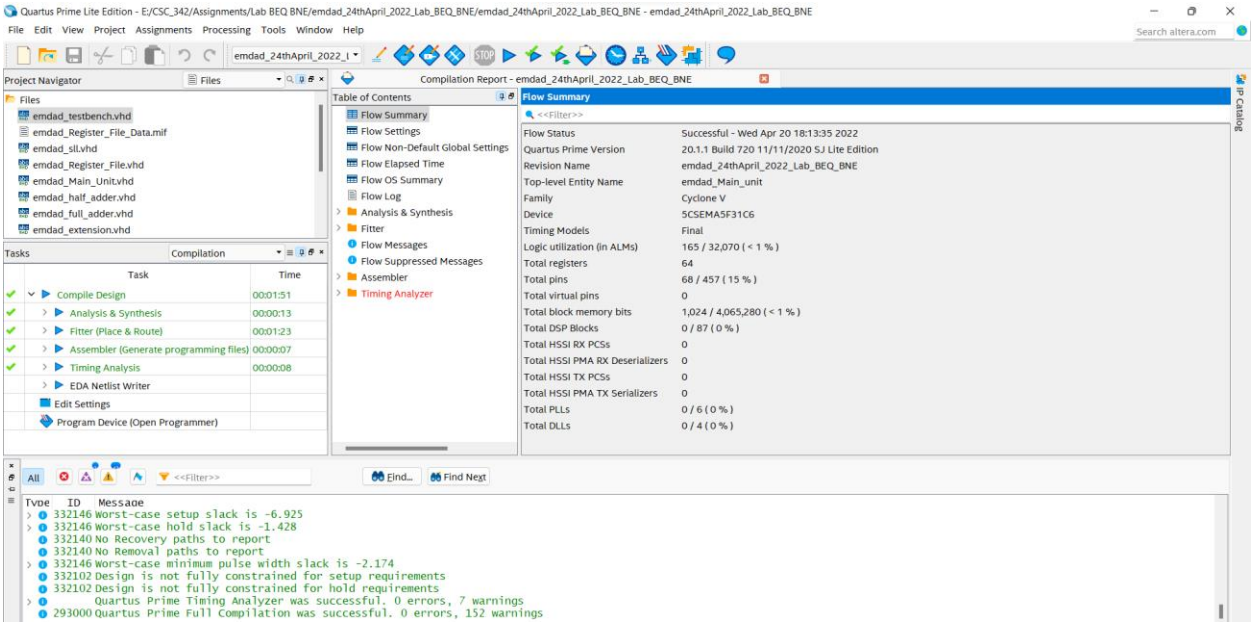


Figure 39: Successful compilation report.

Simulation:

In figure 40, we can see the ModelSim project directory for lab project and VHDL codes compiled successfully.

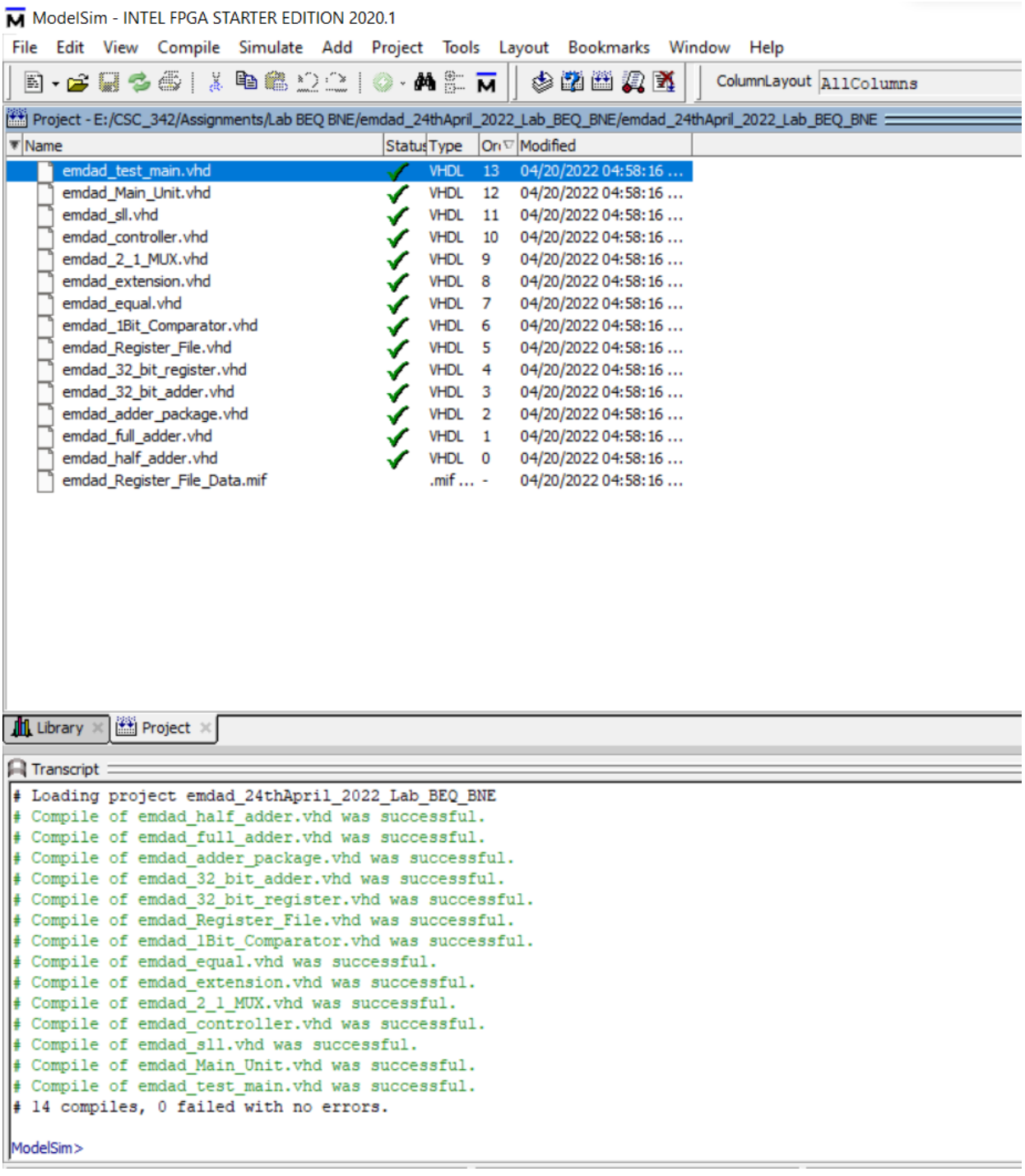


Figure 40: ModelSim project directory

In figure 41, we can see the input entered, instruction is 12340FF in hexadecimal. I did add 4 in the add_4 and kept pc higher order 0. The pcinout is 0.

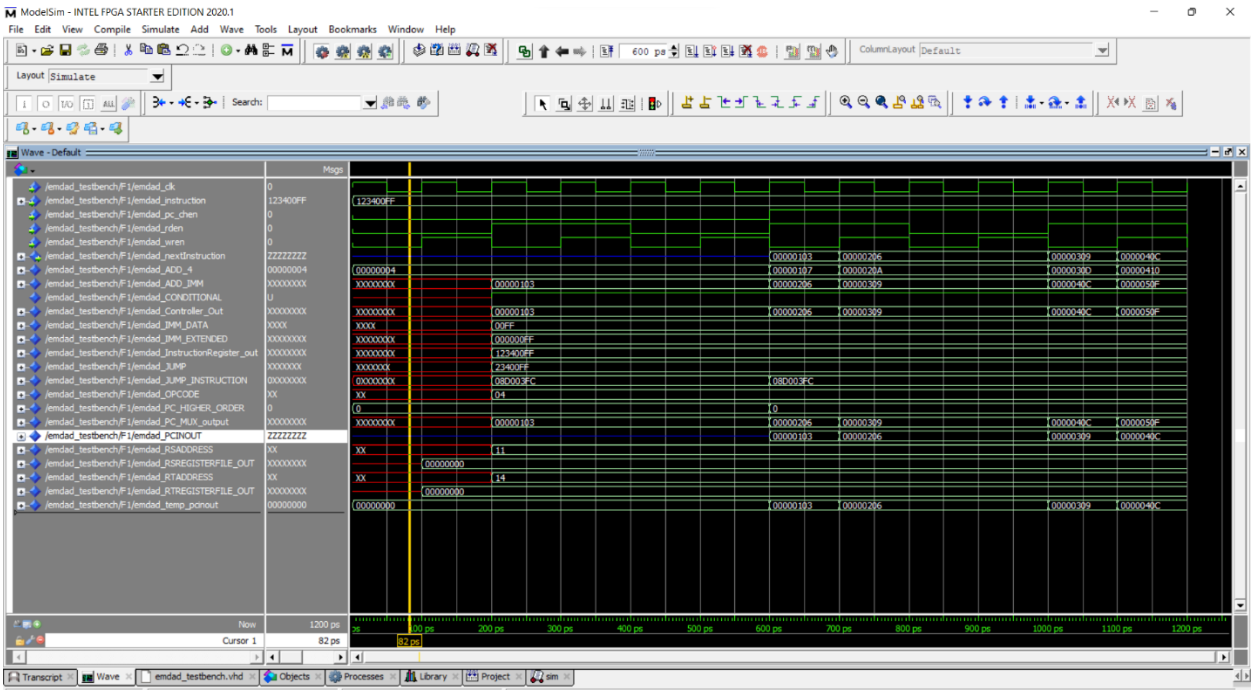


Figure 41: Simulation for inputs

In figure 42, we can see that everything stayed same, but RS and RT got the address input and giving output.

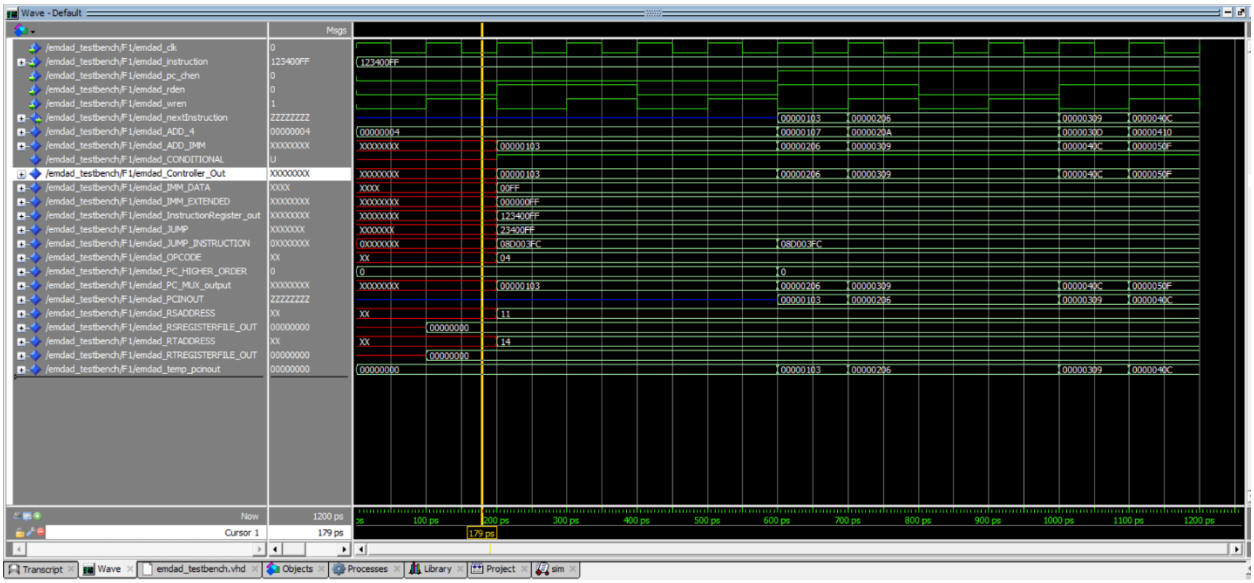


Figure 42: Simulation for RS and RT address output

In figure 43, we can see after the rising clock, both the add_imm address and controller out changes to 103, it is also same for PC_MUX_output. The RSADDRESS and RTADDRESS gave output of 11 and 14. The opcode is 04.

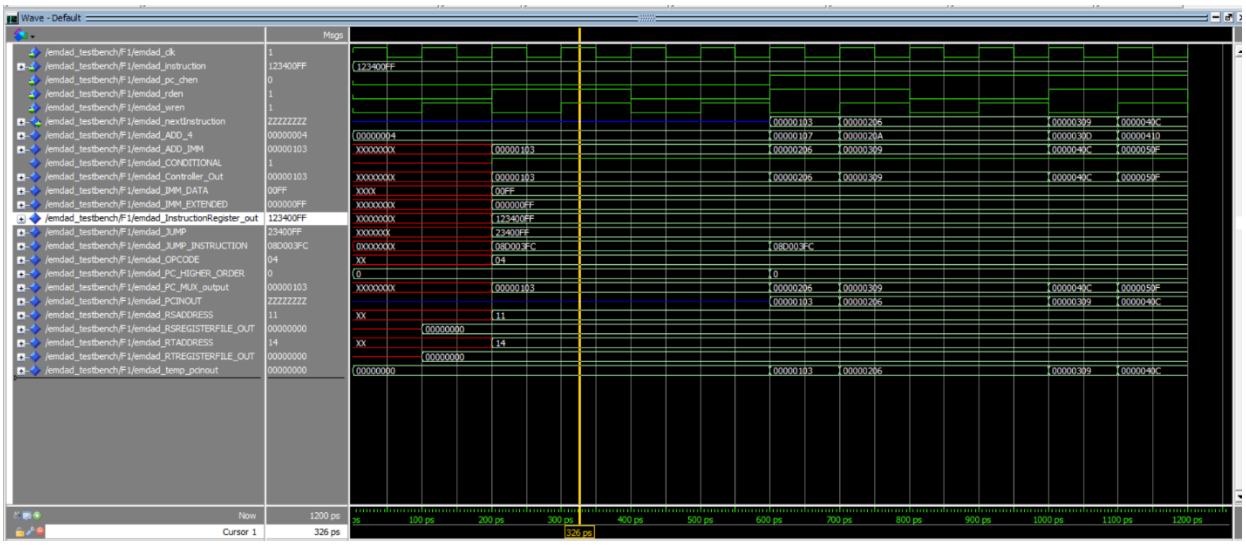


Figure 43: Simulation for RSADDRESS and RTADDRESS output

In figure 44, we can see that after rising clock, we can see the add/sub gave the output to 107 which produced from 103+4=107. The ADD_IMM got double as well by extended.

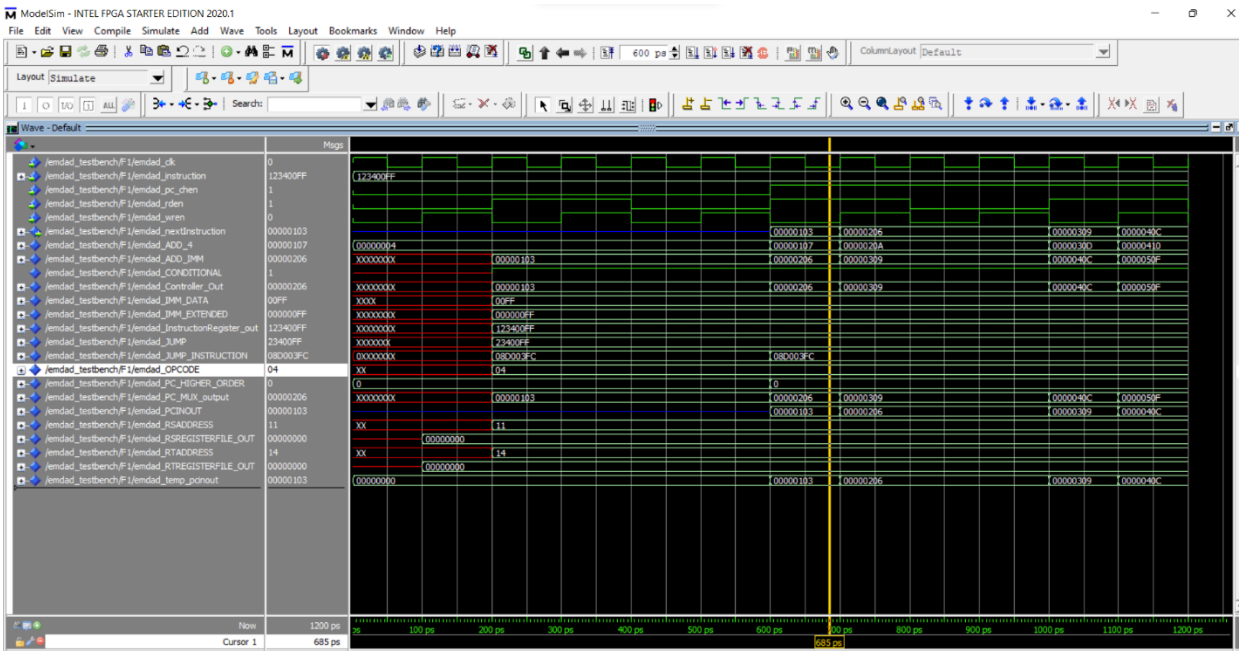


Figure 44: Simulation for add/sub

Conclusion:

In this assignment, I learned how to design and understand the specifications of BNE, BEW and J instruction. I was also able to implement it. It was a code writing and using the components right way was hard. The experiment taught me how to implement MIPS instructions in VHDL further. I relearned and reviewed the concept again and which will help me in the course further.