Lab Project: Arithmetic Logic Unit

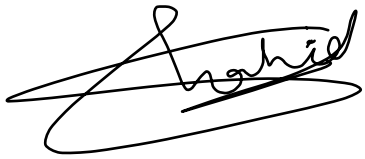
MdShahid Bin Emdad

April 10th, 2022

CSC 34200/34300

Table of Contents

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



Objective:

The objective of this assignment is to learn and understand Arithmetic Logic Unit. We were instructed to implement MIPS arithmetic logic units' instructions by creating and using 3 different 32-bit registers (RD, RS, RT), 16-bit immediate, 32-bit MAR (memory address register), 32-bit MDR (memory data register) and design ALUE with add/sub and bitwise operations. Overall, this lab is to learn how data is written and read from RAM but more complex way.

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).

3 32-bit registers:

In figure 1, we can see the file directory for the ALU project.

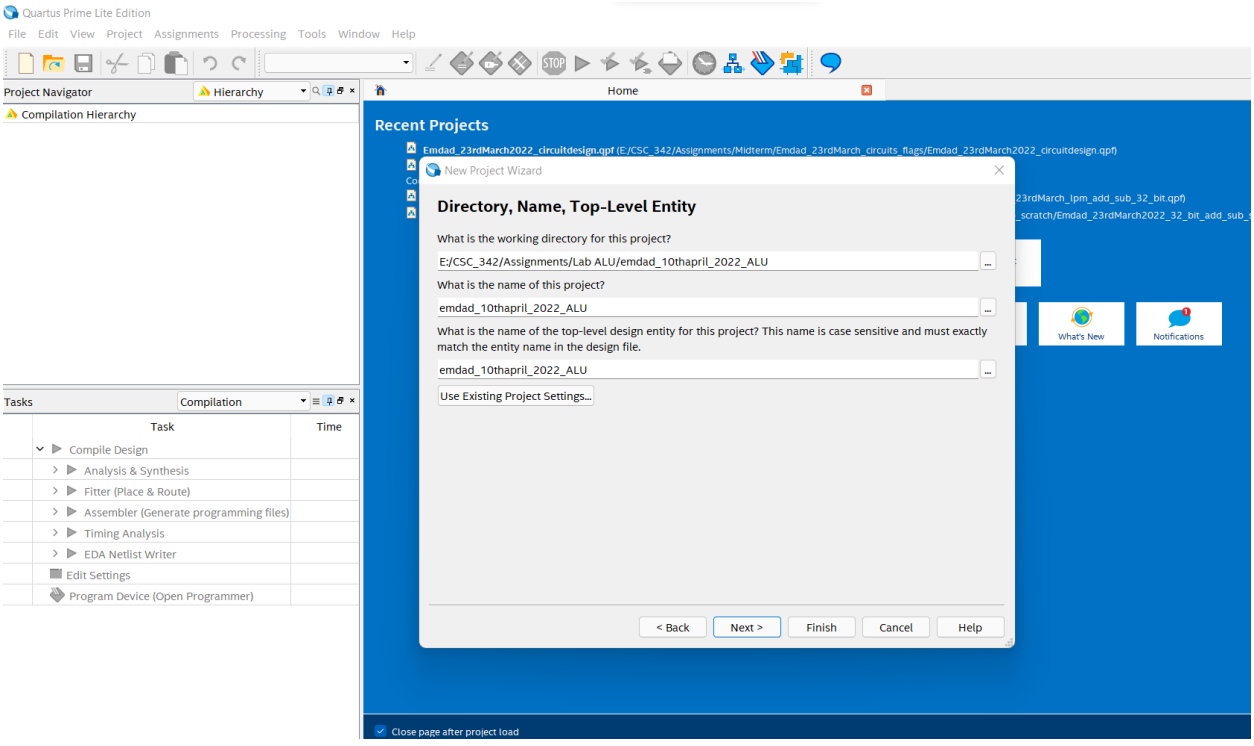


Figure 1: ALU project directory

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

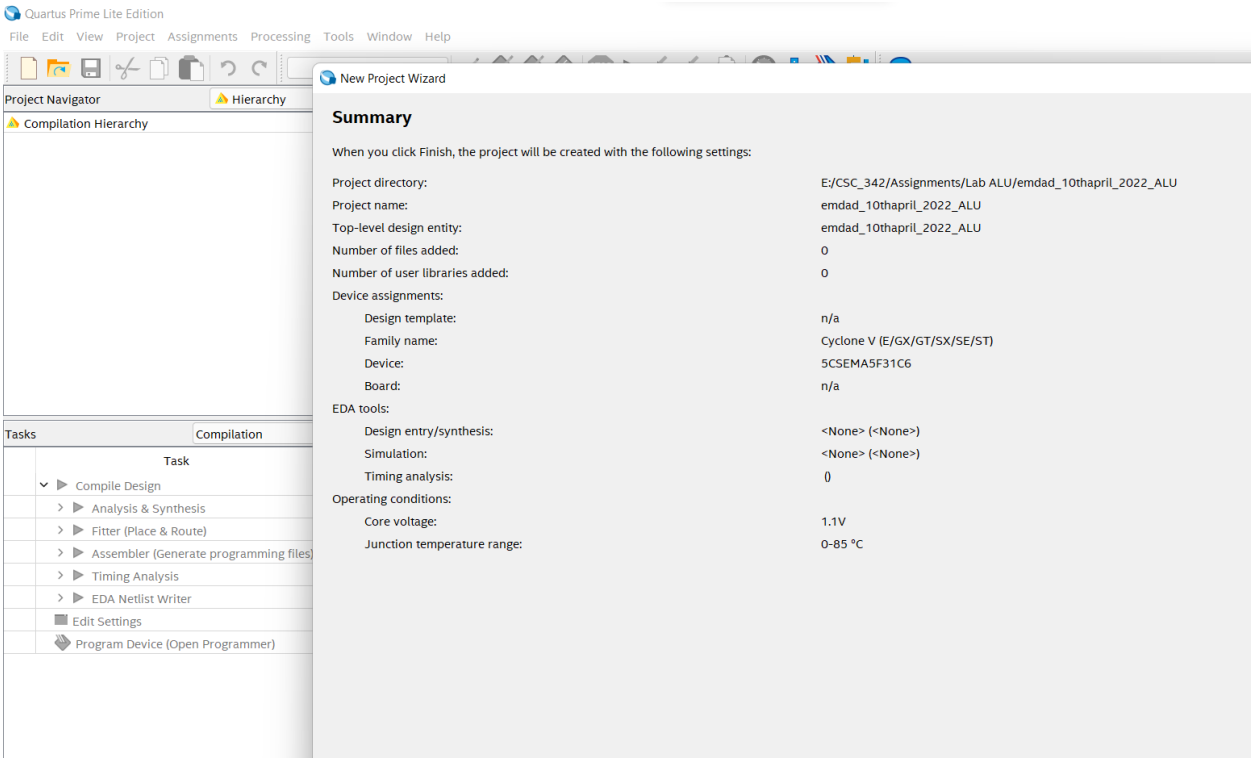


Figure 2: Project summary for ALU

i. 32-bit Register RD:

In figure 3, we can see the VHDL code for 32-bit register RD. It will store data and perform functions to read and write data on a rising clock edge.

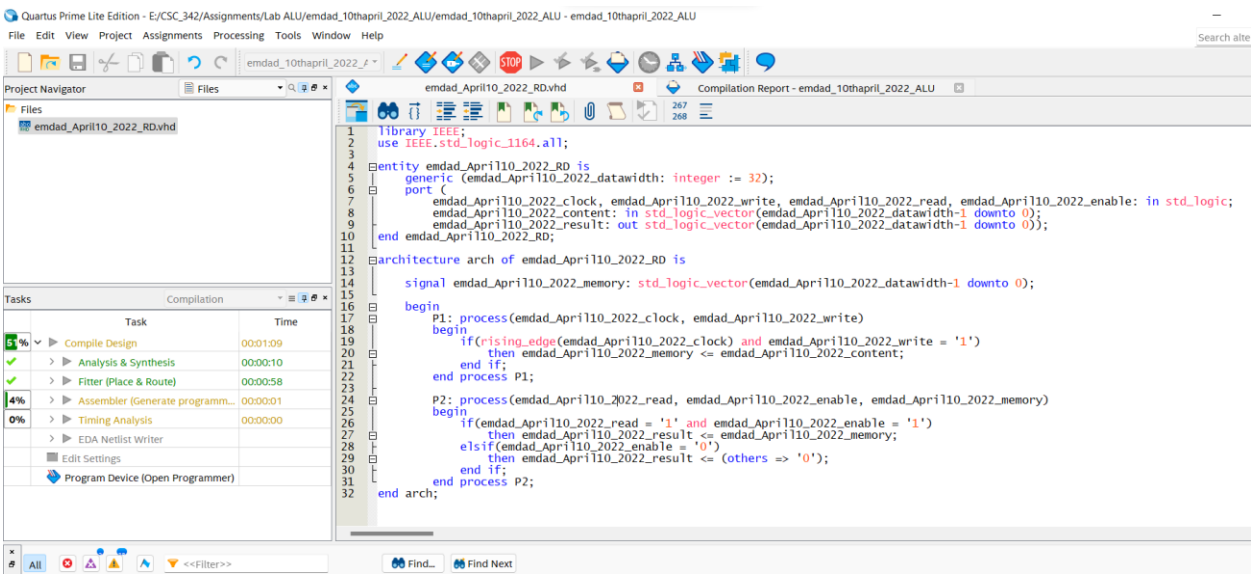


Figure 3: VHDL code for 32-bit RD

In figure 4, we can see it compiled successfully.

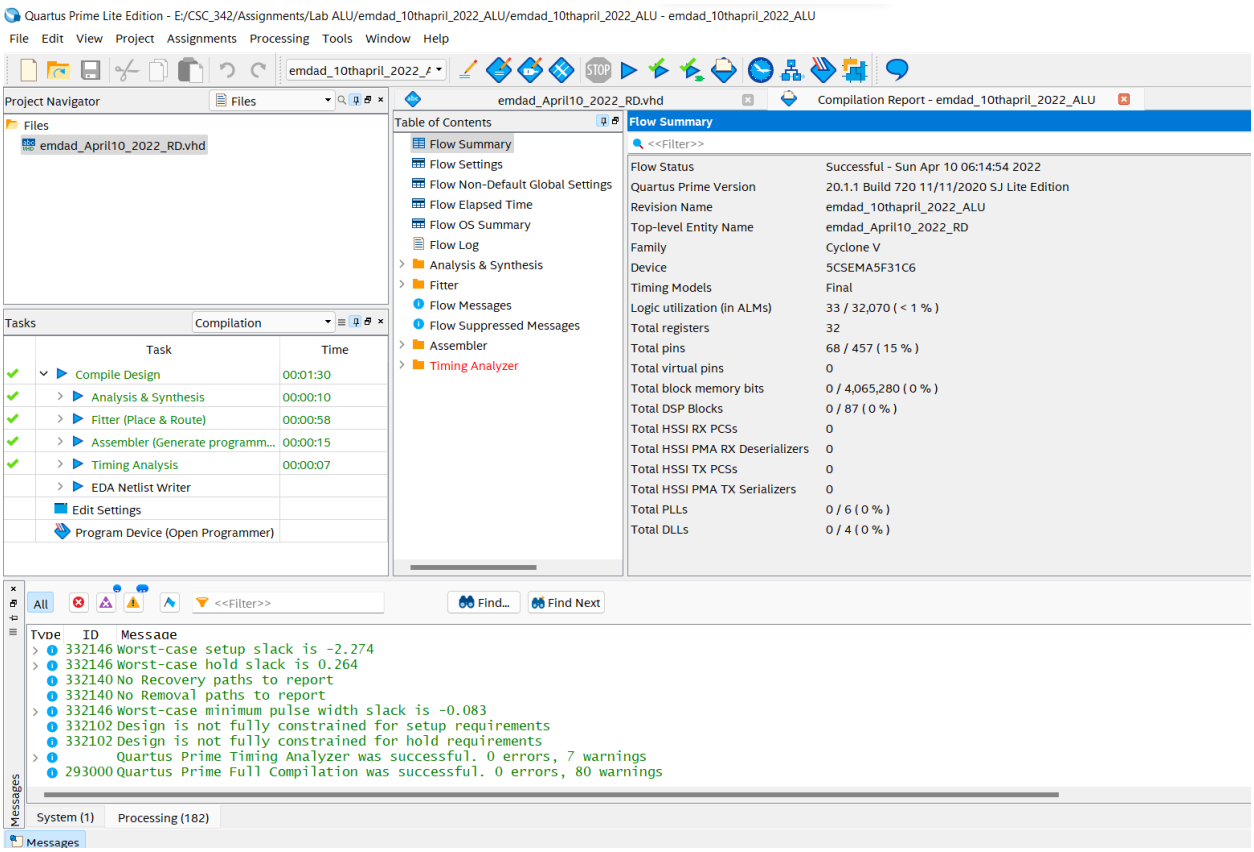


Figure 4: Compilation Report

ii. 32-bit Register RS:

In figure 5, we can see the VHDL code for 32-bit register RS. It will store data and perform functions to read and write data on a rising clock edge.

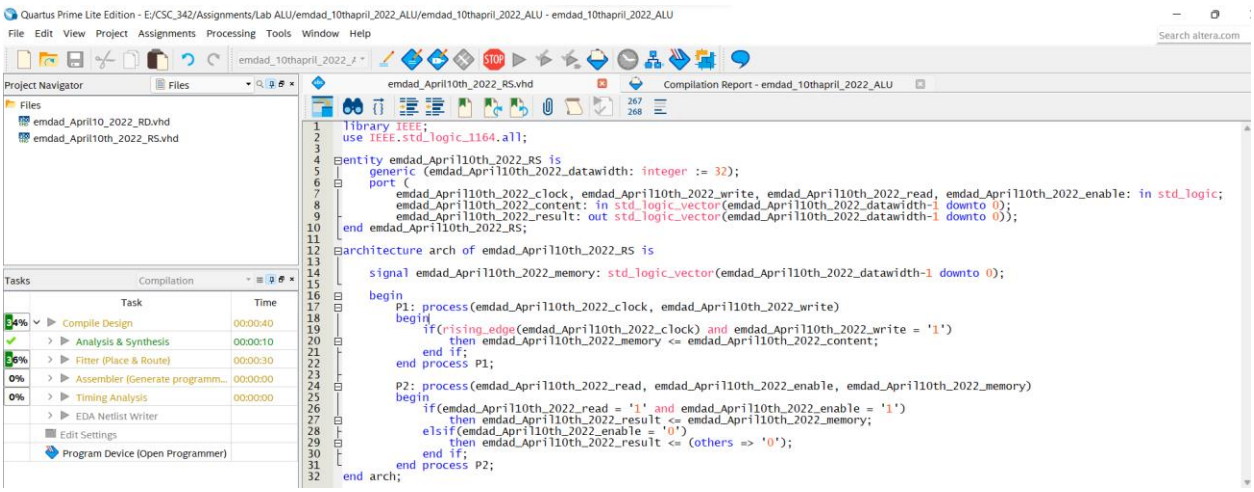


Figure 5: VHDL code for 32-bit RS

In figure 6, we can see it compiled successfully.

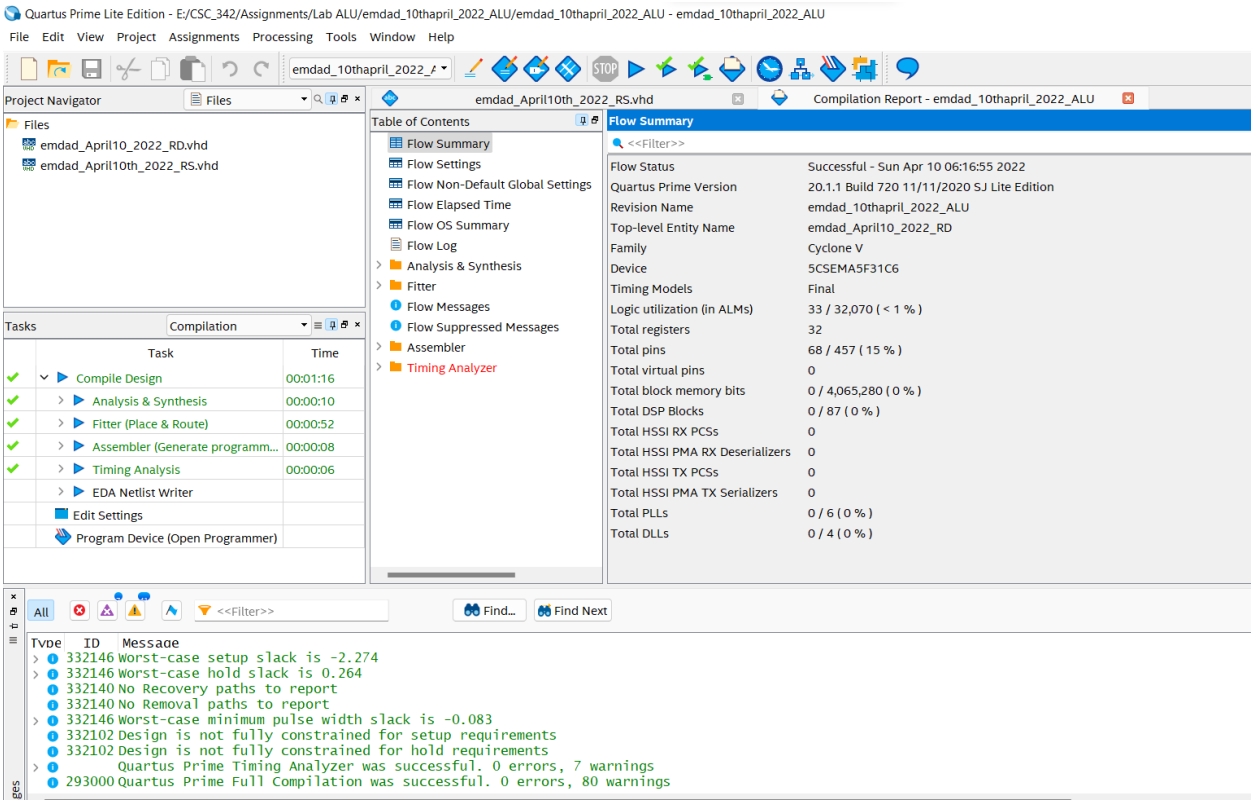


Figure 6: Compilation Report

iii. 32-bit Register RT:

In figure 7, we can see the VHDL code for 32-bit register RT. It will store data and perform functions to read and write data on a rising clock edge.

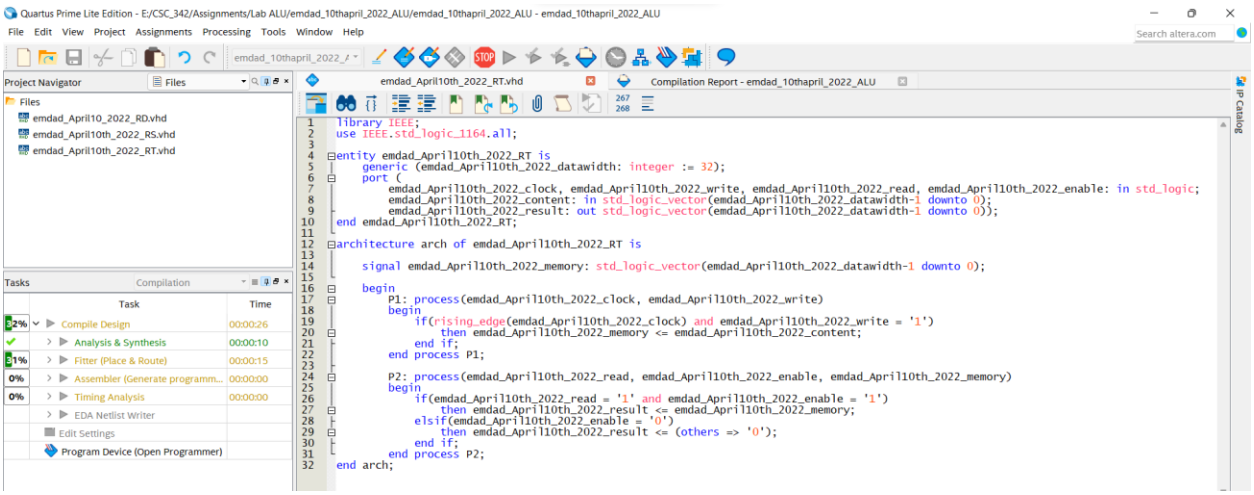


Figure 7: VHDL code for 32-bit RT

In figure 8, we can see it compiled successfully.

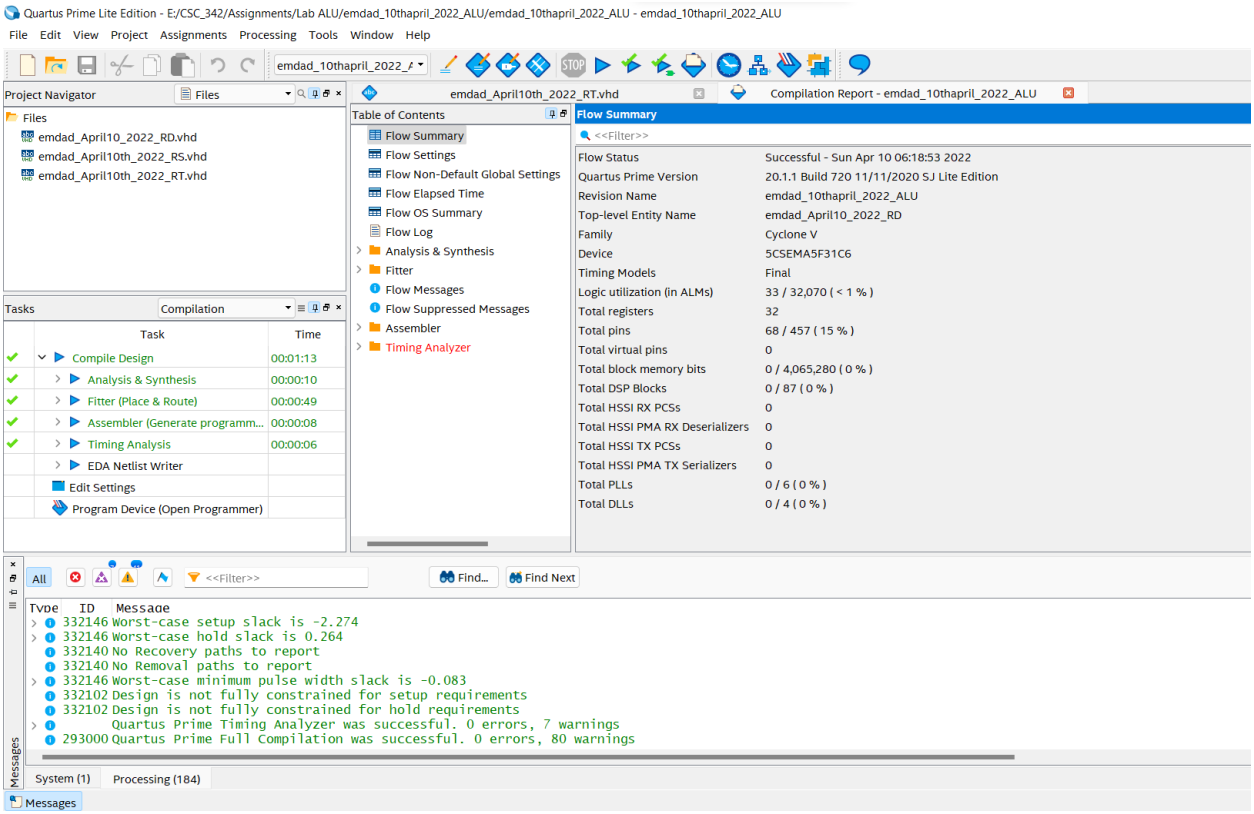


Figure 8: Compilation Report

16-bit immediate register:

In figure 9, we can see code for 16-bit immediate register. It stores data and perform functions to read and write data on a rising clock edge.

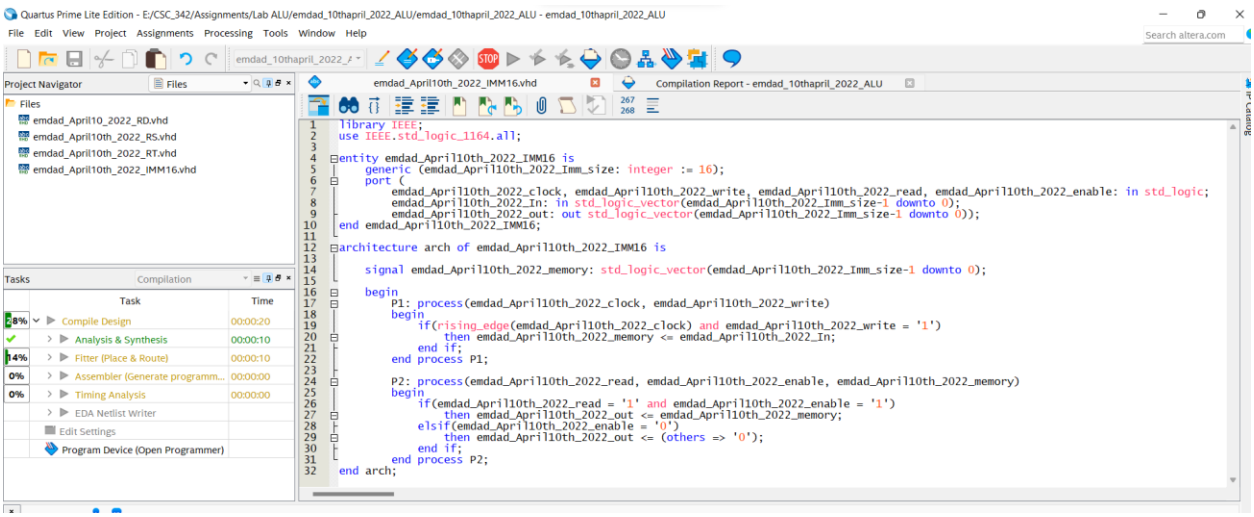


Figure 9: VHDL code 16-bit immediate register

In figure 10, we can see it compiled successfully.

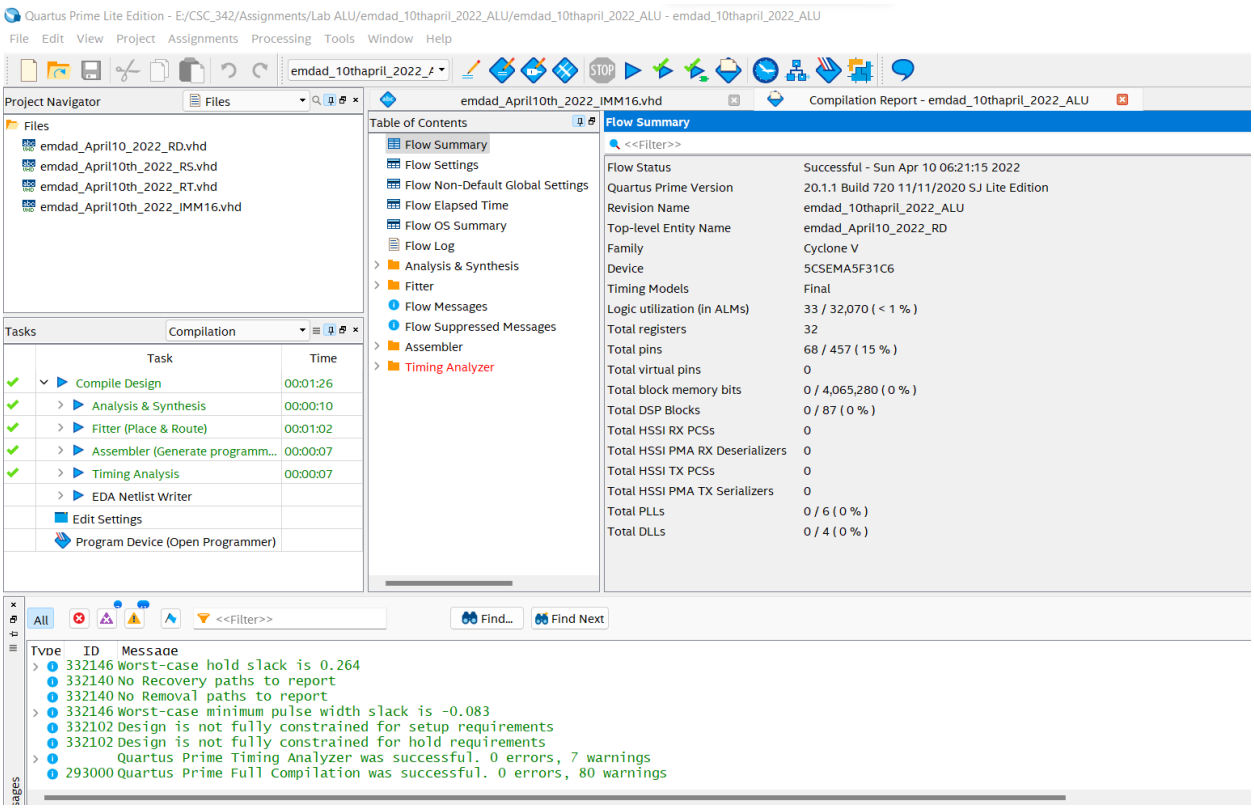


Figure 10: Compilation Report

2:1 32-bit multiplexer:

In figure 11, we can see code for 2:1 32-bit multiplexer. It chooses one of the inputs.

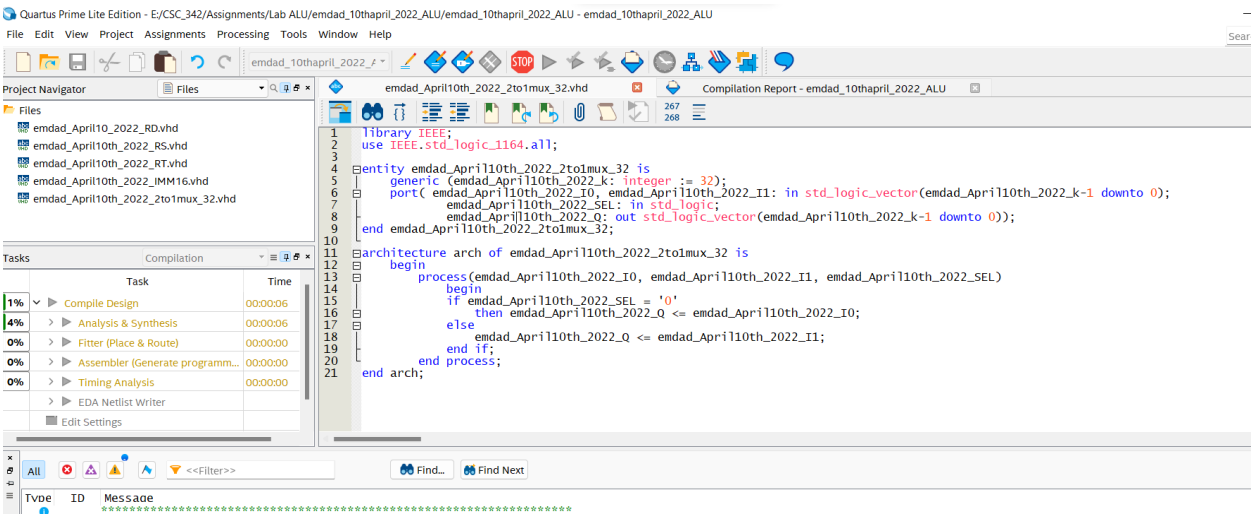


Figure 11: VHDL code for 2:1 32-bit multiplexer

In figure 12, we can see it compiled successfully.

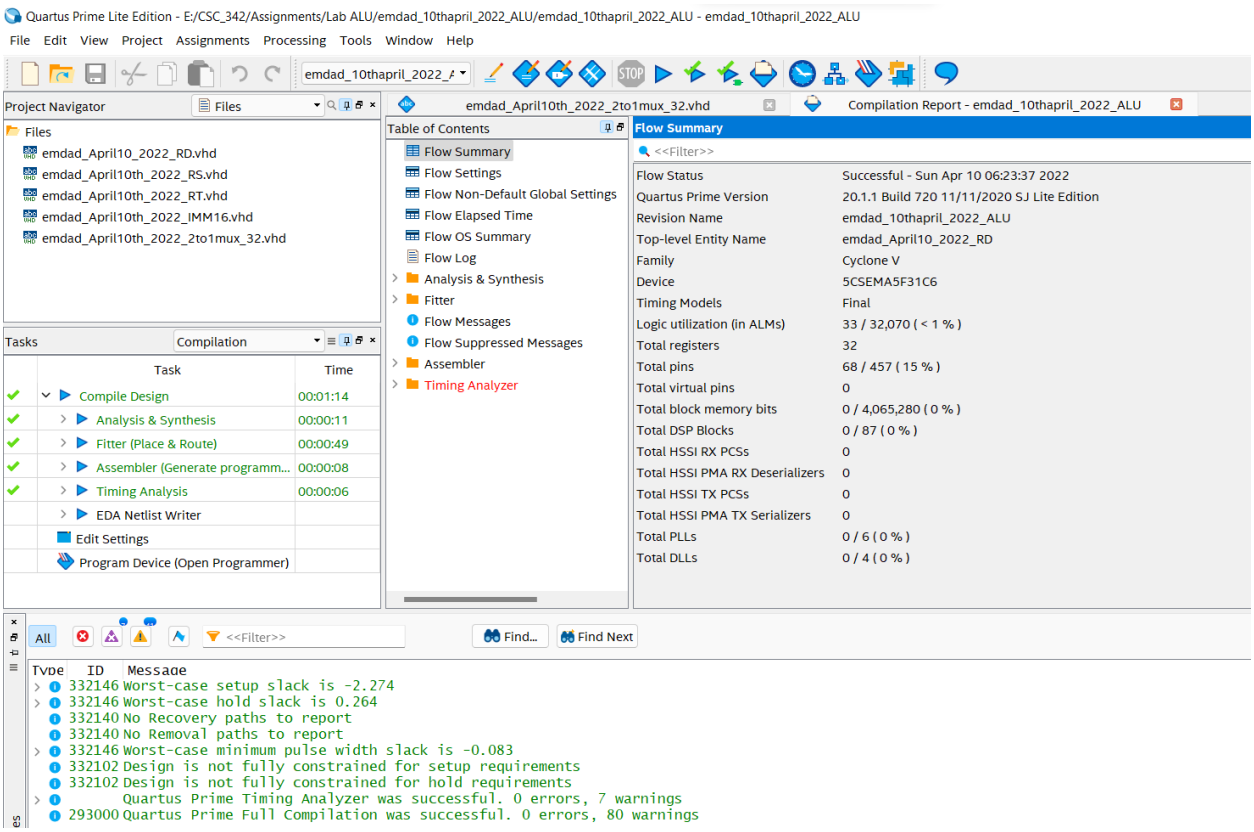
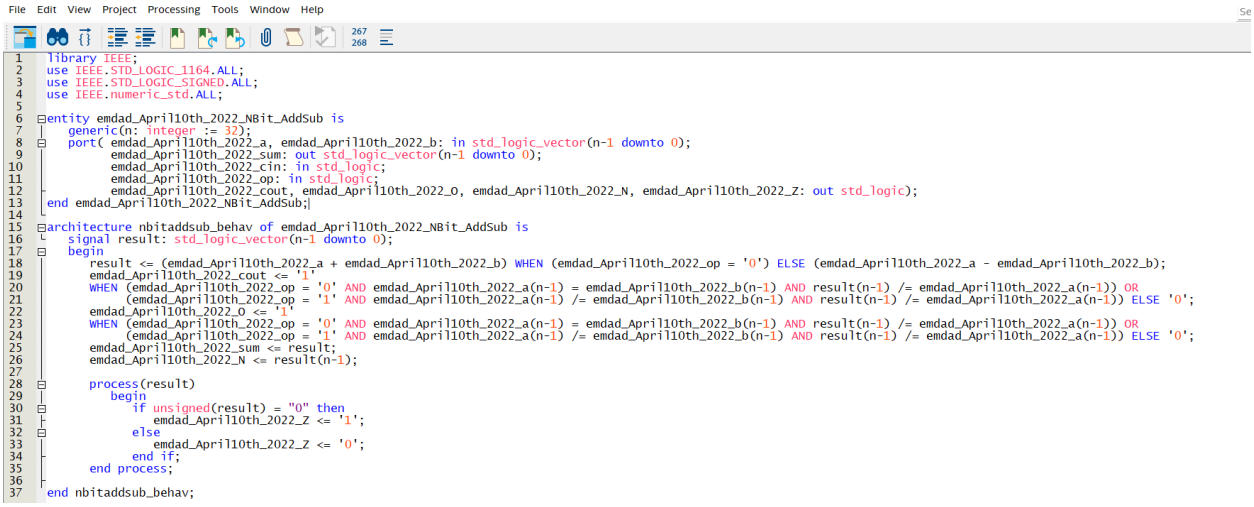


Figure 12: Compilation Report

N-bit add/sub with flags:

In figure 13, we can see the VHDL code 32-bit add/sub with flags. It is to take care of the arithmetic of MIPS instructions.



```
File Edit View Project Processing Tools Window Help
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use IEEE.numeric_std.ALL;

entity emdad_April10th_2022_NBit_AddSub is
    generic(n: integer := 32);
    port( emdad_April10th_2022_a, emdad_April10th_2022_b: in std_logic_vector(n-1 downto 0);
          emdad_April10th_2022_sum: out std_logic_vector(n-1 downto 0);
          emdad_April10th_2022_cin: in std_logic;
          emdad_April10th_2022_op: in std_logic;
          emdad_April10th_2022_cout, emdad_April10th_2022_O, emdad_April10th_2022_N, emdad_April10th_2022_Z: out std_logic);
end emdad_April10th_2022_NBit_AddSub;

architecture nbitaddsub_behav of emdad_April10th_2022_NBit_AddSub is
    signal result: std_logic_vector(n-1 downto 0);
    begin
        result <= (emdad_April10th_2022_a + emdad_April10th_2022_b) WHEN (emdad_April10th_2022_op = '0') ELSE (emdad_April10th_2022_a - emdad_April10th_2022_b);
        emdad_April10th_2022_cout <= '1'
        WHEN (emdad_April10th_2022_op = '0' AND emdad_April10th_2022_a(n-1) = emdad_April10th_2022_b(n-1) AND result(n-1) /= emdad_April10th_2022_a(n-1)) OR
             (emdad_April10th_2022_op = '1' AND emdad_April10th_2022_a(n-1) /= emdad_April10th_2022_b(n-1) AND result(n-1) /= emdad_April10th_2022_a(n-1)) ELSE '0';
        emdad_April10th_2022_O <= '1'
        WHEN (emdad_April10th_2022_op = '0' AND emdad_April10th_2022_a(n-1) = emdad_April10th_2022_b(n-1) AND result(n-1) /= emdad_April10th_2022_a(n-1)) OR
             (emdad_April10th_2022_op = '1' AND emdad_April10th_2022_a(n-1) /= emdad_April10th_2022_b(n-1) AND result(n-1) /= emdad_April10th_2022_a(n-1)) ELSE '0';
        emdad_April10th_2022_sum <= result;
        emdad_April10th_2022_N <= result(n-1);

        process(result)
            begin
                if unsigned(result) = "0" then
                    emdad_April10th_2022_Z <= '1';
                else
                    emdad_April10th_2022_Z <= '0';
                end if;
        end process;

end nbitaddsub_behav;
```

Figure 13: VHDL code 32-bit add/sub with flags

In figure 14, we can see it compiled successfully.

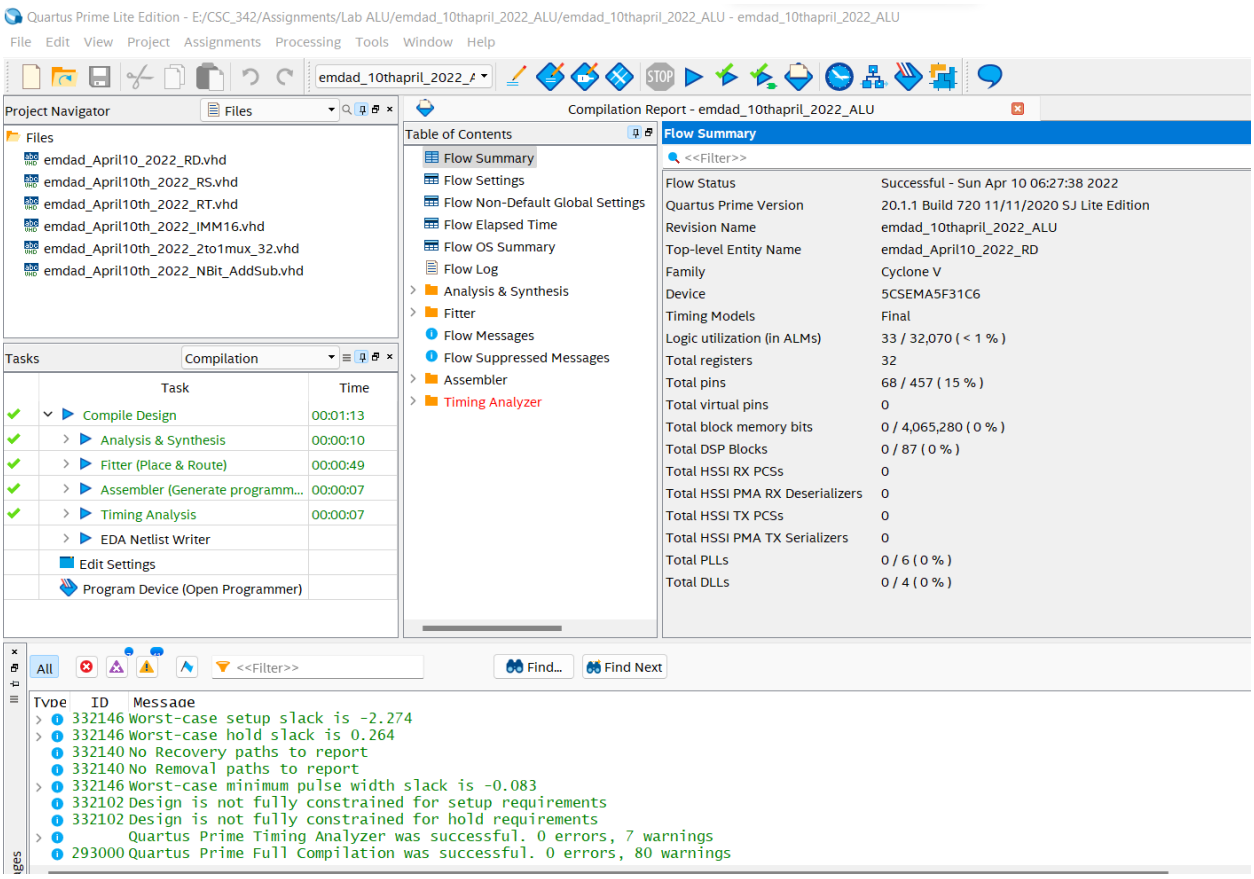


Figure 14: Compilation Report

1:2 32-bit demultiplexer:

In figure 15, we see the code for 1:2 32-bit demultiplexer. it converts the output based on input.

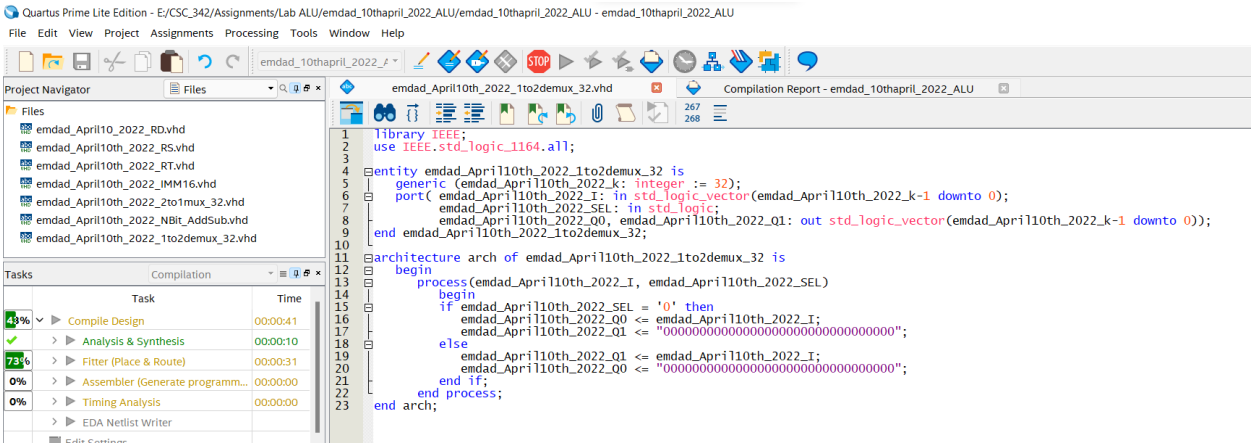


Figure 15: VHDL code for 1:2 32-bit demultiplexer

In figure 16, we can see it compiled successfully.

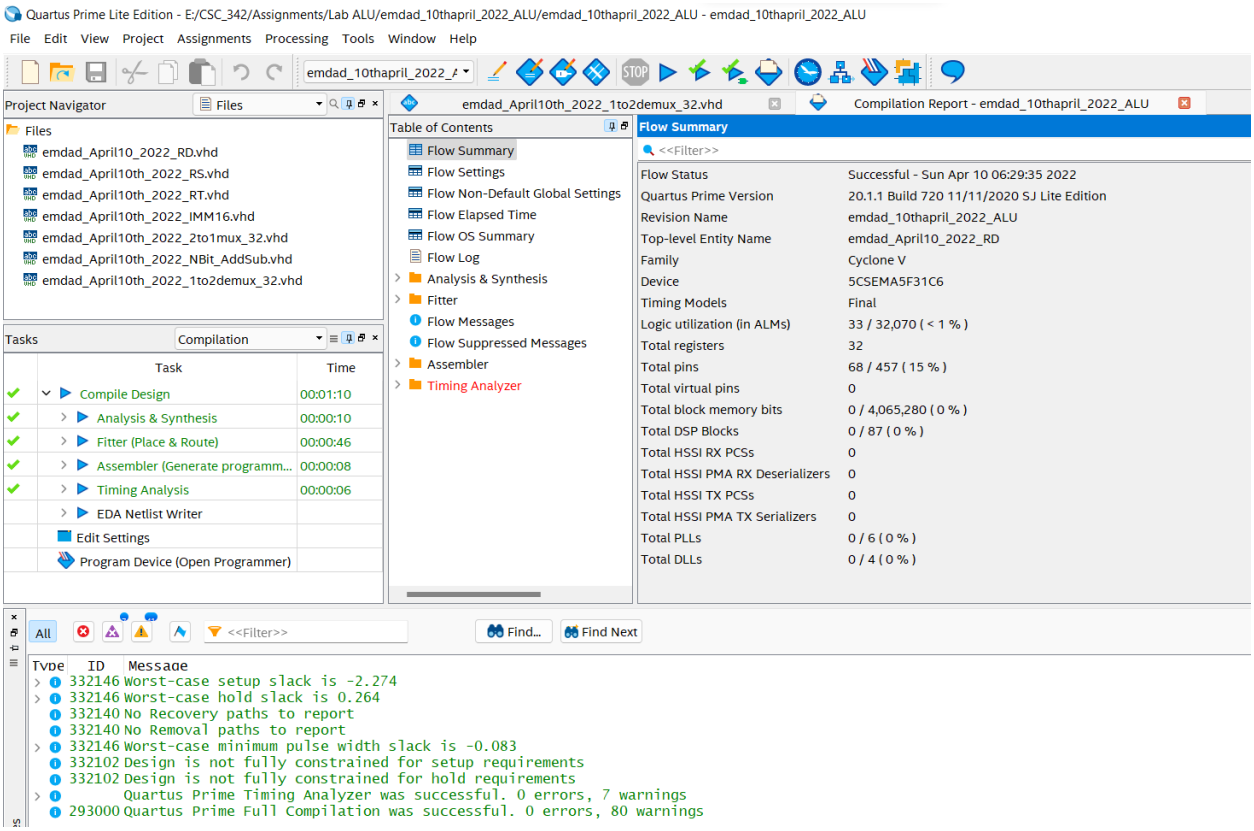


Figure 16: Compilation Report

Bitwise Operations:

In figure 17, we can see the code for the bitwise operation which handles the bitwise for MIPS.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
use IEEE.numeric_std.all;

entity emdad_April10th_2022_bitwise_op is
    generic(N: integer := 32);
    port(
        emdad_April10th_2022_I0, emdad_April10th_2022_I1, emdad_April10th_2022_ext: in std_logic_vector (N-1 downto 0); --32 bit register inputs
        emdad_April10th_2022_Imm: in std_logic_vector (15 downto 0); --16 bit immediate
        emdad_April10th_2022_op: in std_logic_vector(3 downto 0); --Operation Code
        emdad_April10th_2022_Q: out std_logic_vector (N-1 downto 0); --32 bit register output
        emdad_April10th_2022_Z: out std_logic := '0'; --Zero flag
        emdad_April10th_2022_N: out std_logic := '0'; --Negative flag
        emdad_April10th_2022_O: out std_logic := '0'); --Overflow flag
end emdad_April10th_2022_bitwise_op;

architecture arch of emdad_April10th_2022_bitwise_op is
    signal result: std_logic_vector (N-1 downto 0);
    begin
        P1: process(emdad_April10th_2022_I0, emdad_April10th_2022_I1, emdad_April10th_2022_op, result)
            begin
                case emdad_April10th_2022_op is
                    when "0110"=> result <= emdad_April10th_2022_I0 AND emdad_April10th_2022_I1; -- and
                    when "0111"=> result <= emdad_April10th_2022_I0 AND emdad_April10th_2022_ext; -- andi
                    when "1000"=> result <= emdad_April10th_2022_I0 NOR emdad_April10th_2022_I1; -- nor
                    when "1001"=> result <= emdad_April10th_2022_I0 OR emdad_April10th_2022_I1; -- or
                    when "1010"=> result <= emdad_April10th_2022_I0 OR emdad_April10th_2022_ext; --ori
                    when "1011"=> result <= std_logic_vector(shift_left(unsigned(emdad_April10th_2022_I1), to_integer(unsigned(emdad_April10th_2022_Imm)))); -- sll
                    when "1100"=> result <= std_logic_vector(shift_right(unsigned(emdad_April10th_2022_I1), to_integer(unsigned(emdad_April10th_2022_Imm)))); -- srl
                    when "1101"=> result <= std_logic_vector(shift_right(signed(emdad_April10th_2022_I1), to_integer(unsigned(emdad_April10th_2022_Imm)))); -- sra
                    when others=> result <= x"00000000";
                    end case;
        emdad_April10th_2022_Q <= result;
        end process;
end arch;
```

Figure 17: VHDL code for Bitwise Operations

In figure 18, we can see it compiled successfully.

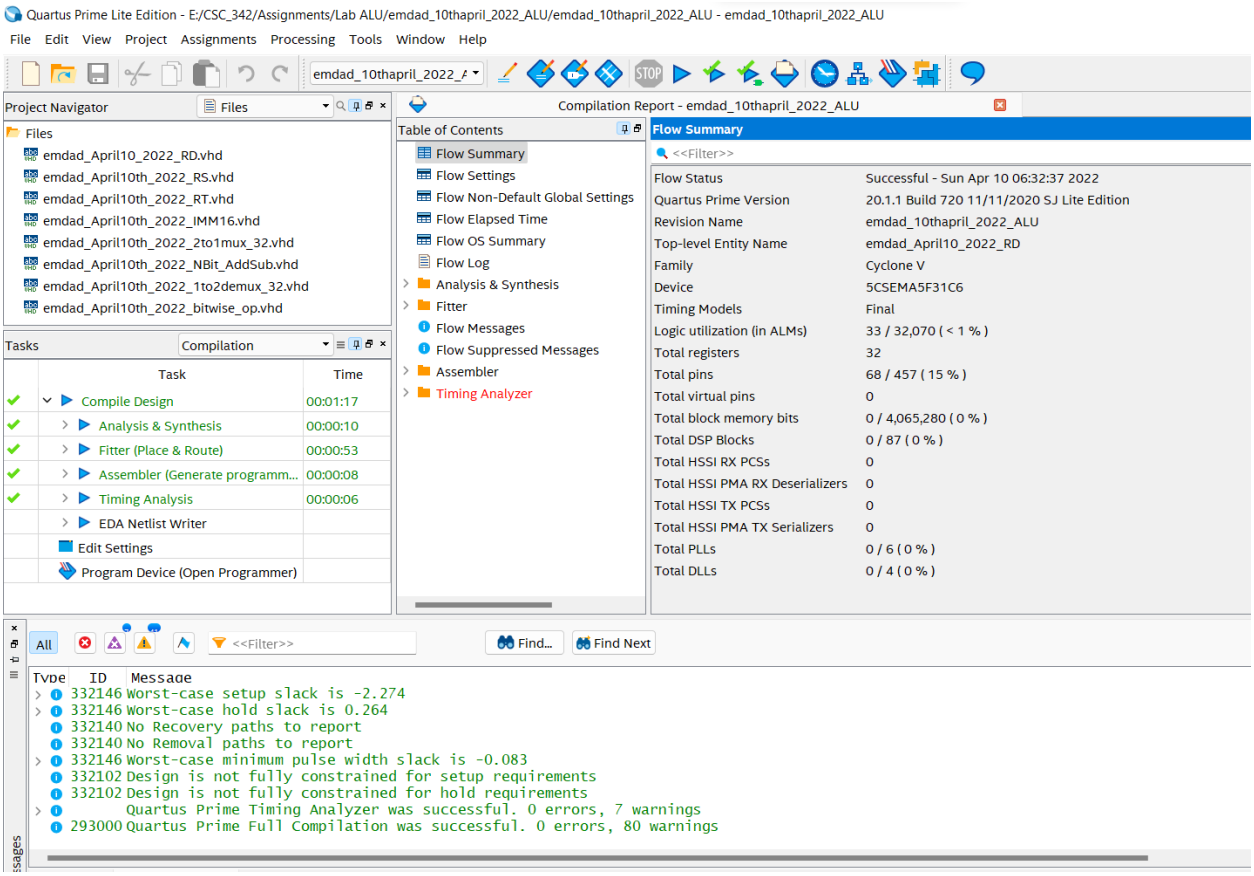


Figure 18: Compilation Report

Sign Extender:

In figure19, we can see the code for sign extender, to perform operation with 32-bit registers.

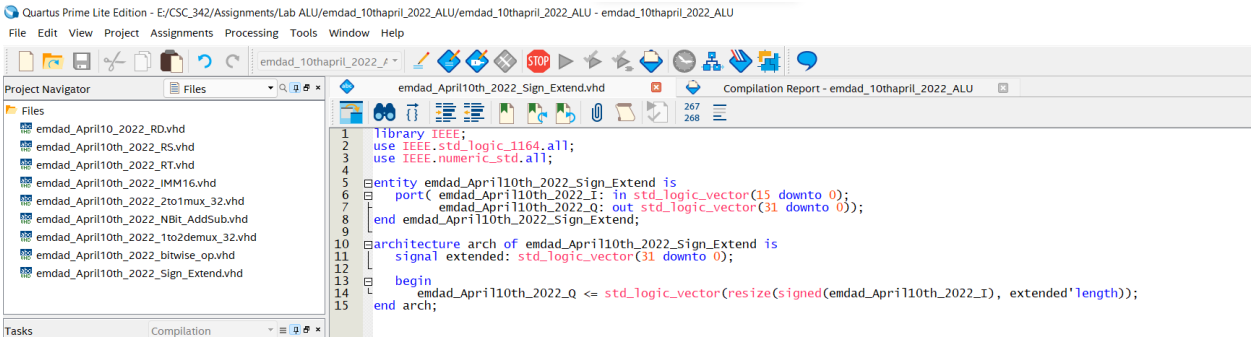


Figure 19: VHDL code for Sign Extender

In figure 20, we can see it compiled successfully.

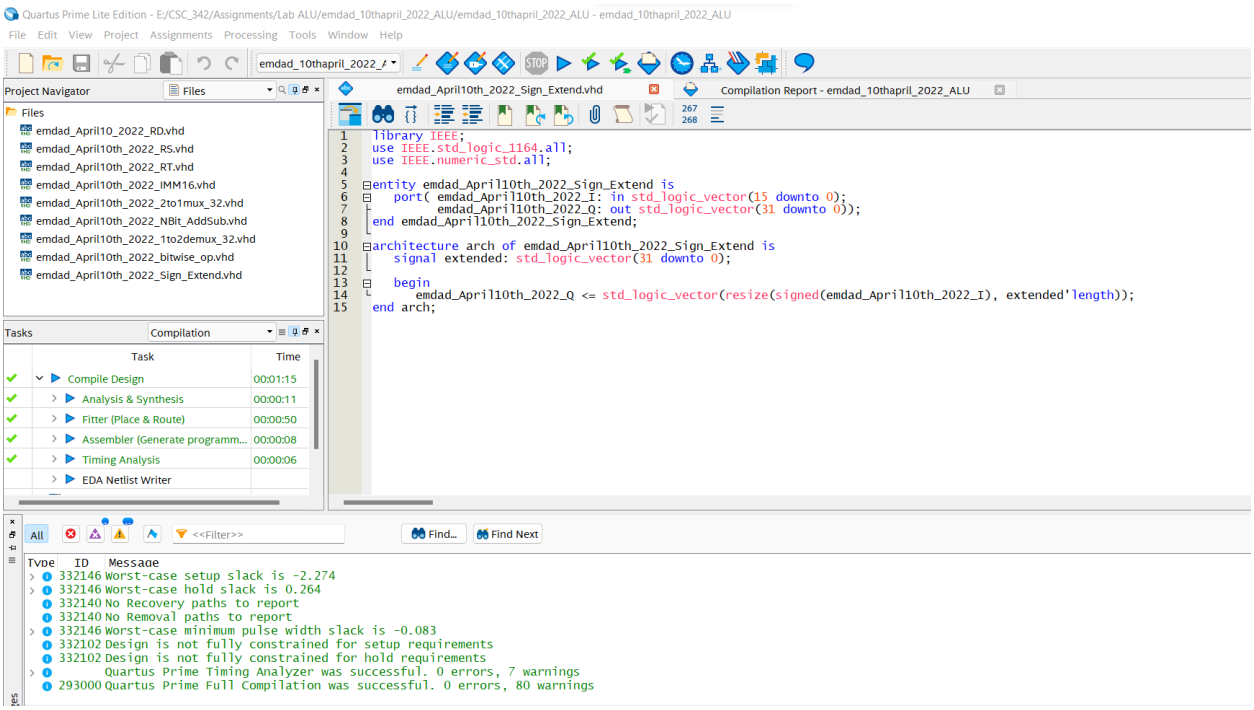


Figure 20: Compilation Report

32-bit MAR:

In figure 21, we can see the VHDL code 32-bit MAR to store and read and write data.

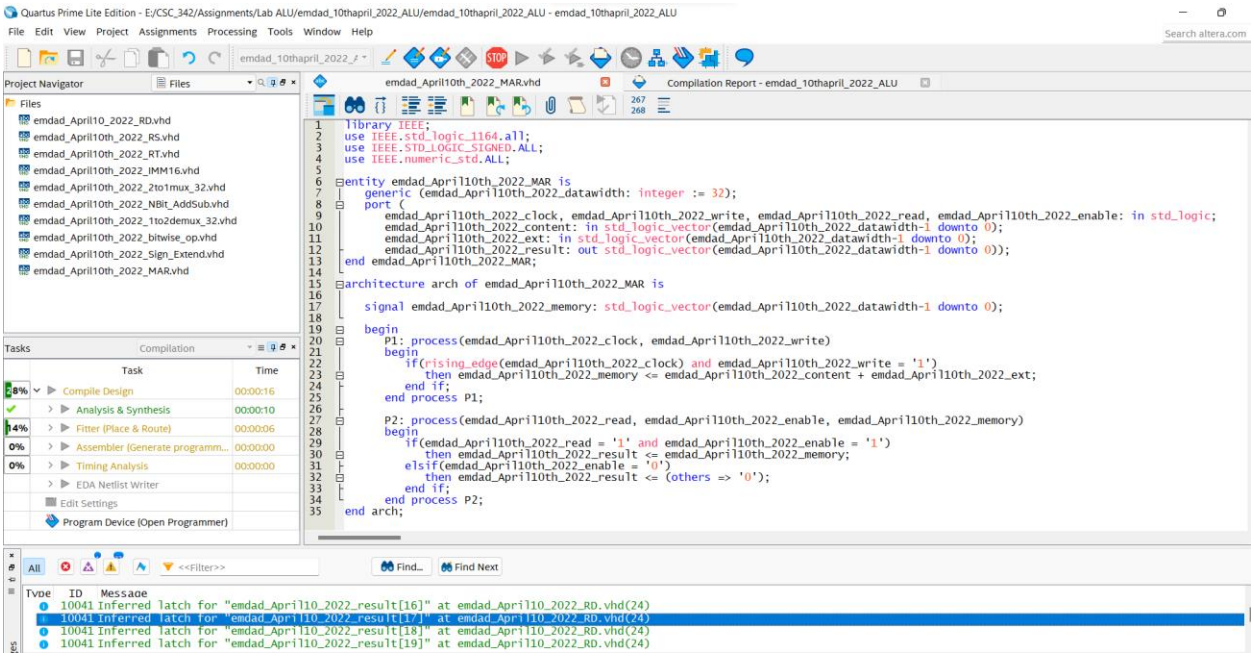


Figure 21: VHDL code 32-bit MAR

In figure 22, we can see it compiled successfully.

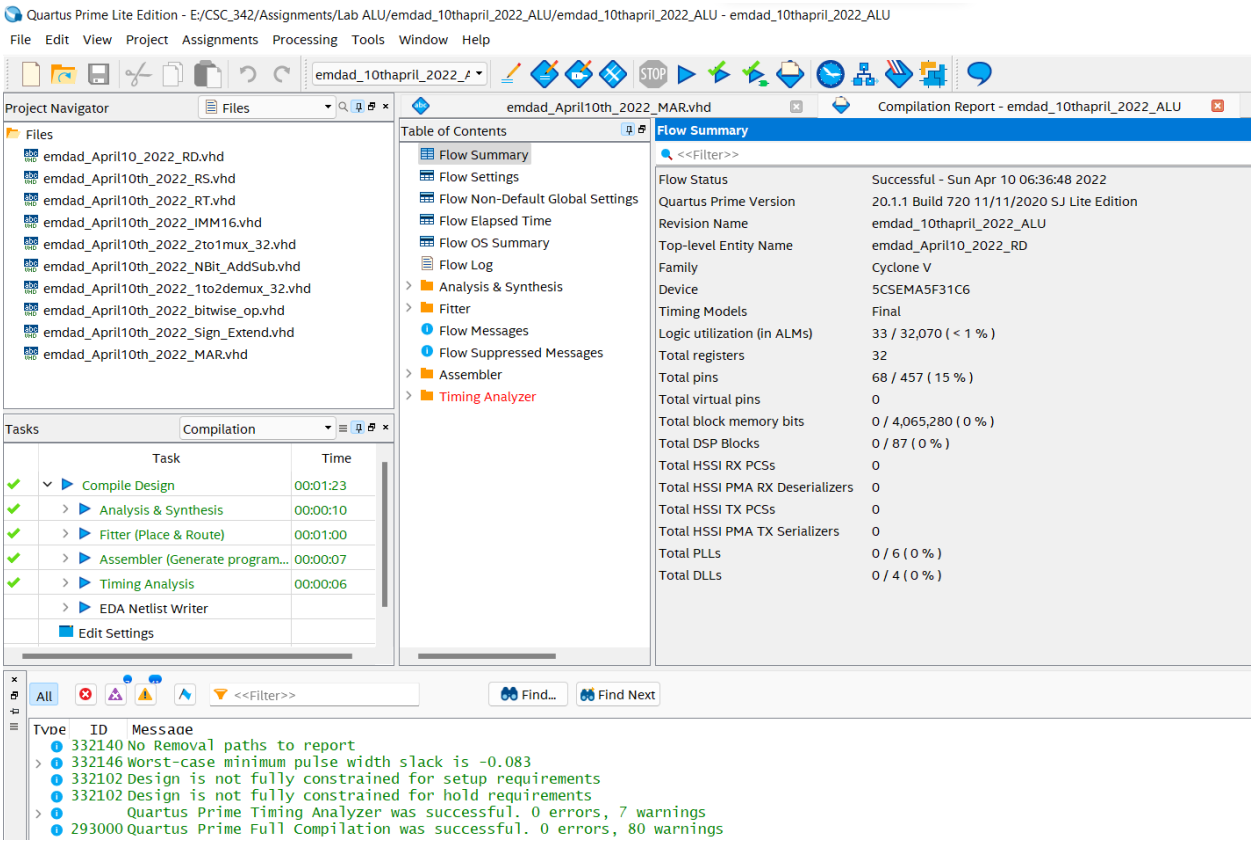


Figure 22: Compilation Report

32-bit MDR:

In figure 23, we can see the VHDL code for 32-bit MDR to store data, read and write.

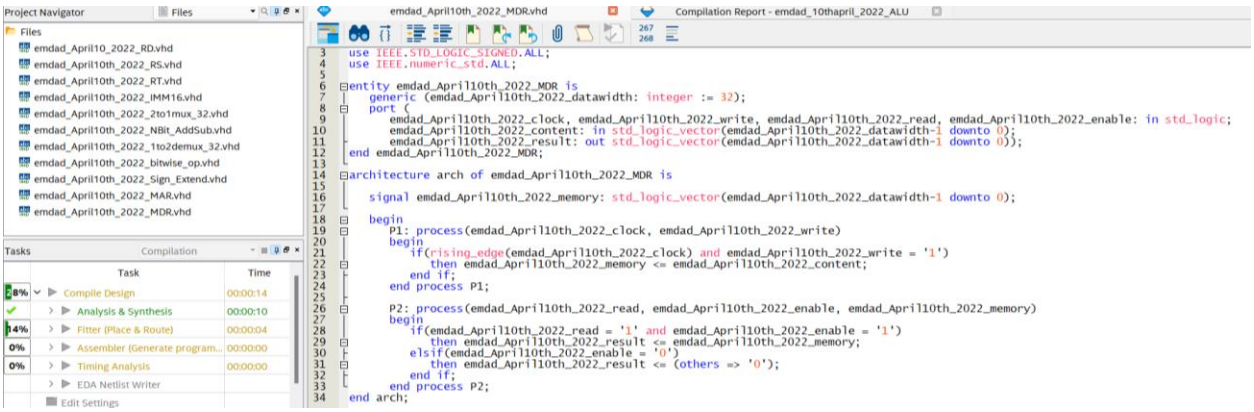


Figure 23: VHDL code for 32-bit MDR

In figure 24, we can see it compiled successfully.

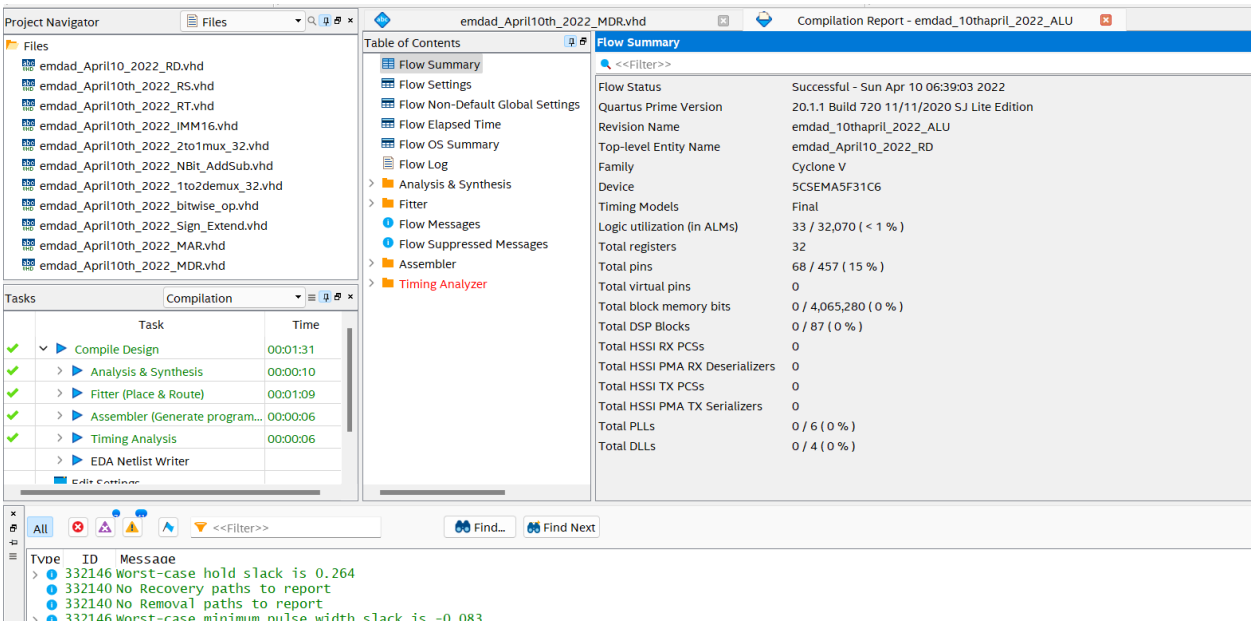


Figure 24: Compilation Report

Arithmetic Logic Unit (ALU):

In figure 25, we can see the VHDL code for the ALU. The inputs are 32-bit registers (RS, RT) to compute operations, 16-bit immediate to compute immediate operations, 32-bit MDR, clock and operation. The outputs are 32-bit register RD and 3 flags: zero, negative, overflow. I also used the other codes as a component to get the waveform.

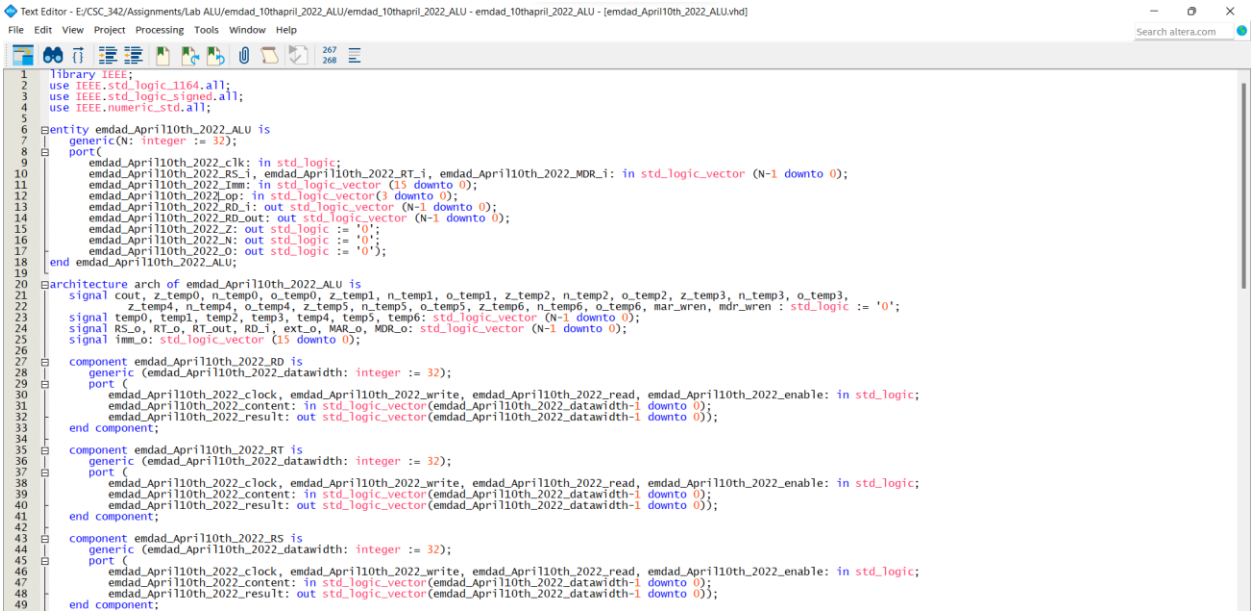


Figure 25: VHDL code for the ALU (Part 1)

```
    component emdad_April10th_2022_IMM16 is
        generic (emdad_April10th_2022_Imm_size: integer := 16);
        port (
            emdad_April10th_2022_clock, emdad_April10th_2022_write, emdad_April10th_2022_read, emdad_April10th_2022_enable: in std_logic;
            emdad_April10th_2022_In: in std_logic_vector(emdad_April10th_2022_Imm_size-1 downto 0);
            emdad_April10th_2022_out: out std_logic_vector(emdad_April10th_2022_Imm_size-1 downto 0));
    end component;

    component emdad_April10th_2022_Sign_Extend is
    port( emdad_April10th_2022_I: in std_logic_vector(15 downto 0);
            emdad_April10th_2022_Q: out std_logic_vector(31 downto 0));
    end component;

    component emdad_April10th_2022_MAR is
        generic (emdad_April10th_2022_datawidth: integer := 32);
        port (
            emdad_April10th_2022_clock, emdad_April10th_2022_write, emdad_April10th_2022_read, emdad_April10th_2022_enable: in std_logic;
            emdad_April10th_2022_content: in std_logic_vector(emdad_April10th_2022_datawidth-1 downto 0);
            emdad_April10th_2022_ext: in std_logic_vector(emdad_April10th_2022_datawidth-1 downto 0);
            emdad_April10th_2022_result: out std_logic_vector(emdad_April10th_2022_datawidth-1 downto 0));
    end component;

    component emdad_April10th_2022_MDR is
        generic (emdad_April10th_2022_datawidth: integer := 32);
        port (
            emdad_April10th_2022_clock, emdad_April10th_2022_write, emdad_April10th_2022_read, emdad_April10th_2022_enable: in std_logic;
            emdad_April10th_2022_content: in std_logic_vector(emdad_April10th_2022_datawidth-1 downto 0);
            emdad_April10th_2022_result: out std_logic_vector(emdad_April10th_2022_datawidth-1 downto 0));
    end component;

    component emdad_April10th_2022_NBit_AddSub is
        generic(n: integer := 32);
        port( emdad_April10th_2022_a, emdad_April10th_2022_b: in std_logic_vector(n-1 downto 0);
              emdad_April10th_2022_sum: out std_logic_vector(n-1 downto 0);
              emdad_April10th_2022_cin: in std_logic;
              emdad_April10th_2022_op: in std_logic;
              emdad_April10th_2022_cout, emdad_April10th_2022_O, emdad_April10th_2022_N, emdad_April10th_2022_Z: out std_logic);
    end component;

    component emdad_April10th_2022_bitwise_op is
        generic(N: integer := 32);
        port(
            emdad_April10th_2022_IO, emdad_April10th_2022_I1, emdad_April10th_2022_ext: in std_logic_vector (N-1 downto 0);
            emdad_April10th_2022_Imm: in std_logic_vector (15 downto 0);
            emdad_April10th_2022_op: in std_logic_vector(3 downto 0);
            emdad_April10th_2022_Q: out std_logic_vector (N-1 downto 0);
            emdad_April10th_2022_Z: out std_logic := '0';
            emdad_April10th_2022_N: out std_logic := '0';
            emdad_April10th_2022_O: out std_logic := '0');
    end component;
```

Figure 26: VHDL code for the ALU (Part 2)

```
    begin

        RD: emdad_April10th_2022_RD generic map (emdad_April10th_2022_datawidth => 32)
        port map (emdad_April10th_2022_clk, '1', '1', '1', RD_i, emdad_April10th_2022_RD_i);
        RS: emdad_April10th_2022_RS generic map (emdad_April10th_2022_datawidth => 32)
        port map (emdad_April10th_2022_clk, '1', '1', '1', emdad_April10th_2022_RS_i, RS_o);
        RT: emdad_April10th_2022_RT generic map (emdad_April10th_2022_datawidth => 32)
        port map (emdad_April10th_2022_clk, '1', '1', '1', emdad_April10th_2022_RT_i, RT_o);
        imm: emdad_April10th_2022_IMM16 generic map (emdad_April10th_2022_Imm_size => 16)
        port map (emdad_April10th_2022_clk, '1', '1', '1', emdad_April10th_2022_Imm, imm_o);
        ext: emdad_April10th_2022_Sign_Extend port map (emdad_April10th_2022_Imm, ext_o);
        MAR: emdad_April10th_2022_MAR generic map (emdad_April10th_2022_datawidth => 32)
        port map (emdad_April10th_2022_clk, '1', '1', '1', RS_o, ext_o, MAR_o);
        MDR: emdad_April10th_2022_MDR generic map (emdad_April10th_2022_datawidth => 32)
        port map (emdad_April10th_2022_clk, '1', '1', '1', emdad_April10th_2022_MDR_i, MDR_o);


        add: emdad_April10th_2022_NBit_AddSub generic map (n => 32) port map (RS_o, RT_o, temp0, '0', '0', cout, o_temp0, n_temp0, z_temp0);
        addi: emdad_April10th_2022_NBit_AddSub generic map (n => 32) port map (RS_o, ext_o, temp1, '0', '0', cout, o_temp1, n_temp1, z_temp1);
        addiu: emdad_April10th_2022_NBit_AddSub generic map (n => 32) port map (RS_o, ext_o, temp2, '0', '0', cout, o_temp2, n_temp2, z_temp2);
        addu: emdad_April10th_2022_NBit_AddSub generic map (n => 32) port map (RS_o, RT_o, temp3, '0', '0', cout, o_temp3, n_temp3, z_temp3);
        sub: emdad_April10th_2022_NBit_AddSub generic map (n => 32) port map (RS_o, RT_o, temp4, '0', '1', cout, o_temp4, n_temp4, z_temp4);
        subu: emdad_April10th_2022_NBit_AddSub generic map (n => 32) port map (RS_o, RT_o, temp5, '0', '1', cout, o_temp5, n_temp5, z_temp5);
        bit_op: emdad_April10th_2022_bitwise_op generic map (N => 32) port map (RS_o, RT_o, ext_o, imm_o, emdad_April10th_2022_op, temp6, z_temp6, n_temp6,  o_temp6);


        P1: process(temp0, temp1, temp2, temp3, temp4, temp5, temp6)
            begin
                case emdad_April10th_2022_op is
                    when "0000"=> emdad_April10th_2022_RD_out <= temp0; emdad_April10th_2022_O <= o_temp0;
                    emdad_April10th_2022_N <= n_temp0; emdad_April10th_2022_Z <= z_temp0;
                    when "0001"=> RT_out <= temp1; emdad_April10th_2022_O <= o_temp1; emdad_April10th_2022_N <= n_temp1; emdad_April10th_2022_Z <= z_temp1;
                    when "0010"=> RT_out <= temp2; emdad_April10th_2022_O <= '0'; emdad_April10th_2022_N <= n_temp2;
                    emdad_April10th_2022_Z <= z_temp2;
                    when "0011"=> emdad_April10th_2022_RD_out <= temp3; emdad_April10th_2022_O <= '0';
                    emdad_April10th_2022_N <= n_temp3; emdad_April10th_2022_Z <= z_temp3;
                    when "0100"=> emdad_April10th_2022_RD_out <= temp4; emdad_April10th_2022_O <= o_temp4;
                    emdad_April10th_2022_N <= n_temp4; emdad_April10th_2022_Z <= z_temp4;
                    when "0101"=> emdad_April10th_2022_RD_out <= temp5; emdad_April10th_2022_O <= '0'; emdad_April10th_2022_N <= n_temp5;
                    emdad_April10th_2022_Z <= z_temp5;
                    when "0110" | "1000" | "1001"| "1011" | "1100" | "1101" => emdad_April10th_2022_RD_out <= temp6;
                    emdad_April10th_2022_O <= o_temp6; emdad_April10th_2022_N <= n_temp6; emdad_April10th_2022_Z <= z_temp6;
                    when "0111" | "1010" => RT_out <= temp6; emdad_April10th_2022_O <= o_temp6;
                    emdad_April10th_2022_N <= n_temp6; emdad_April10th_2022_Z <= z_temp6;
                    when "1110" => RT_out <= MAR_o;
                    when "1111" => RT_out <= MDR_o;
                    when others => emdad_April10th_2022_RD_out <= x"00000000";
                end case;
            end process;
    end arch;
```

Figure 27: VHDL code for the ALU (Part 3)

In figure 28, we can see it compiled successfully.

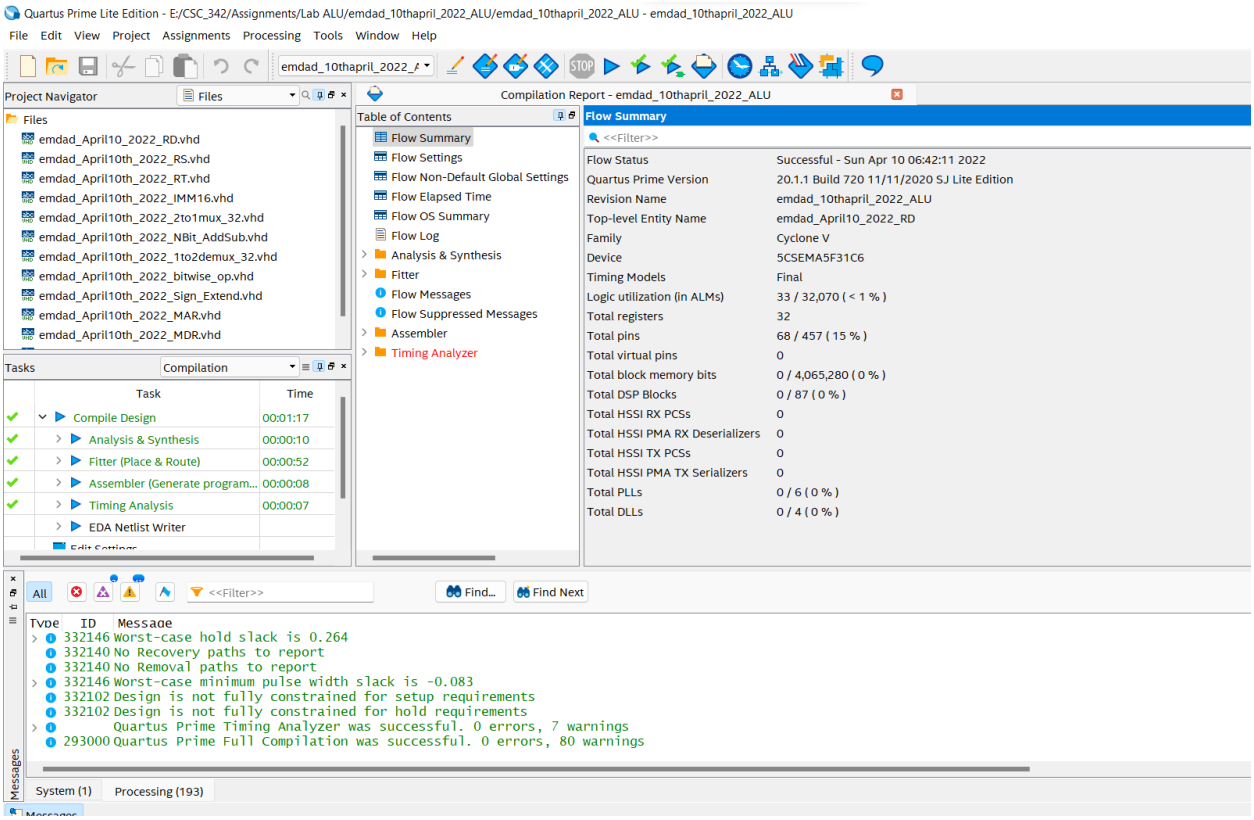


Figure 28: Compilation Report

Simulation:

In figure 29, we can see the directory for the ALU project for ModelSim.

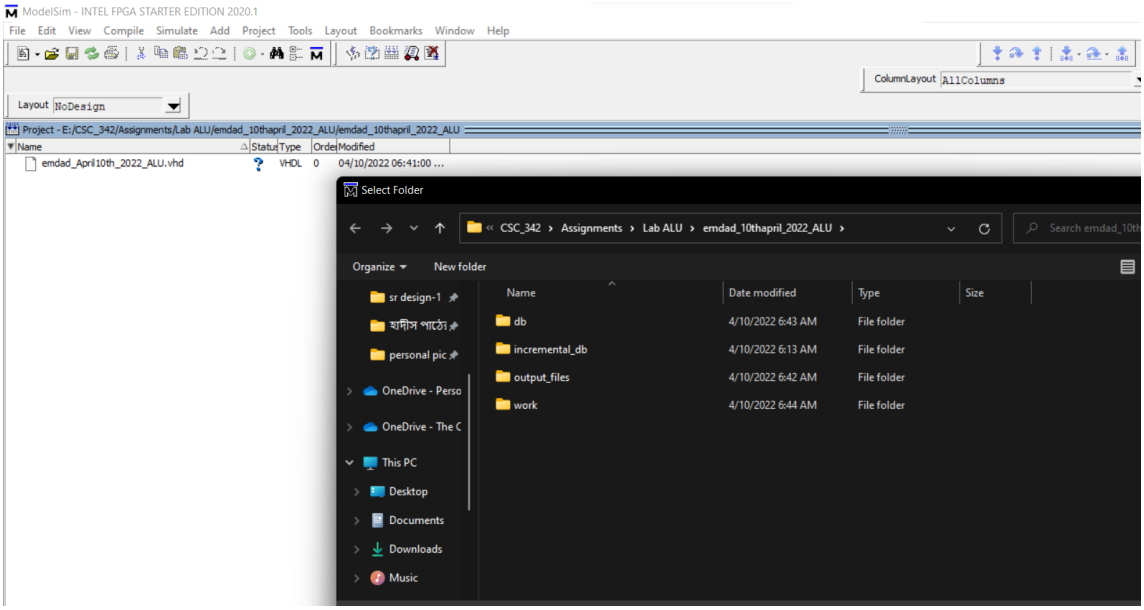


Figure 29: ALU project Modelsim directory

Add:

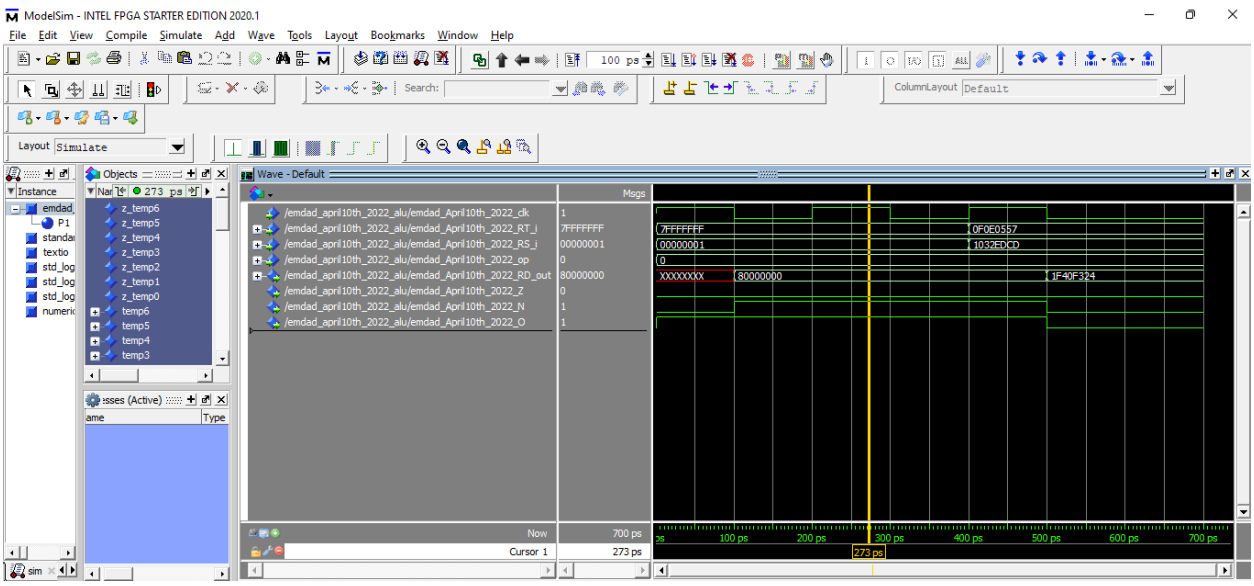


Figure 30: MIPS Instructions ADD simulation

In figure 30, we can see the simulation for the add operation. I used the addition operation where R[rd] = R[rs] + R[rt]. I kept changing the clock signal and forced the value for RS and RT. We can see the negative flag got to 1 because we need two positive numbers.

ADDU:

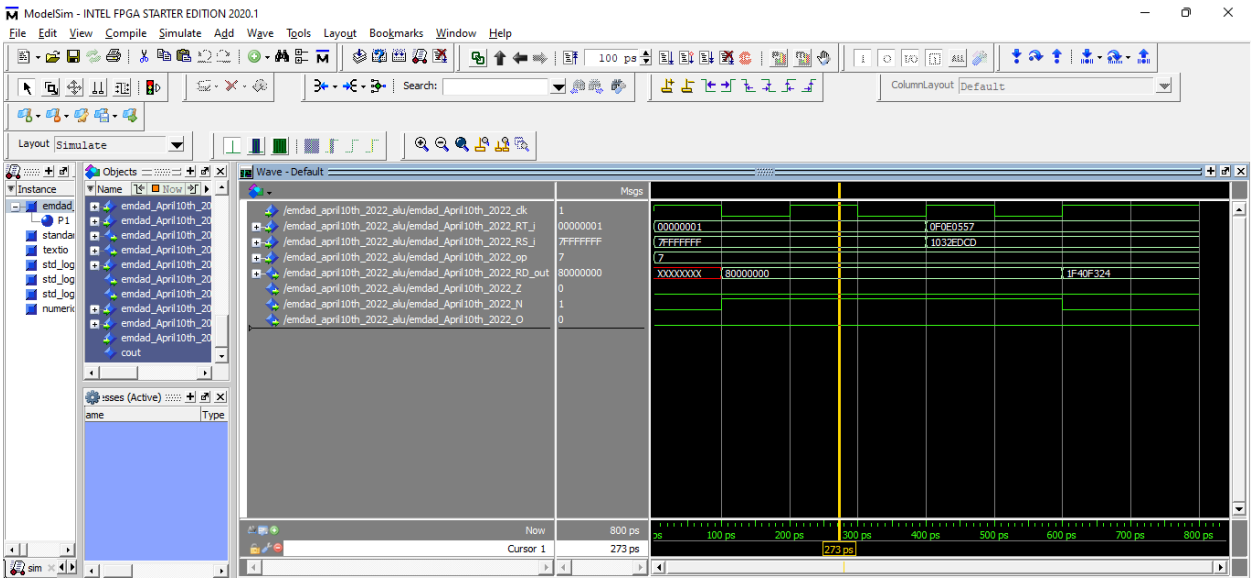


Figure 31: MIPS Instructions ADDU simulation

In figure 31, we can see the simulation for ADDU. Following the equation, R[rd] = R[rs] + R[rt], the VHDL code computes unsigned addition. While changing the clock signal, I forced values into RS and RT. The zero flag and negative flag is 0 because the result is not 0 and positive and there is no overflow occurred as the operation was unsigned.

SUB:

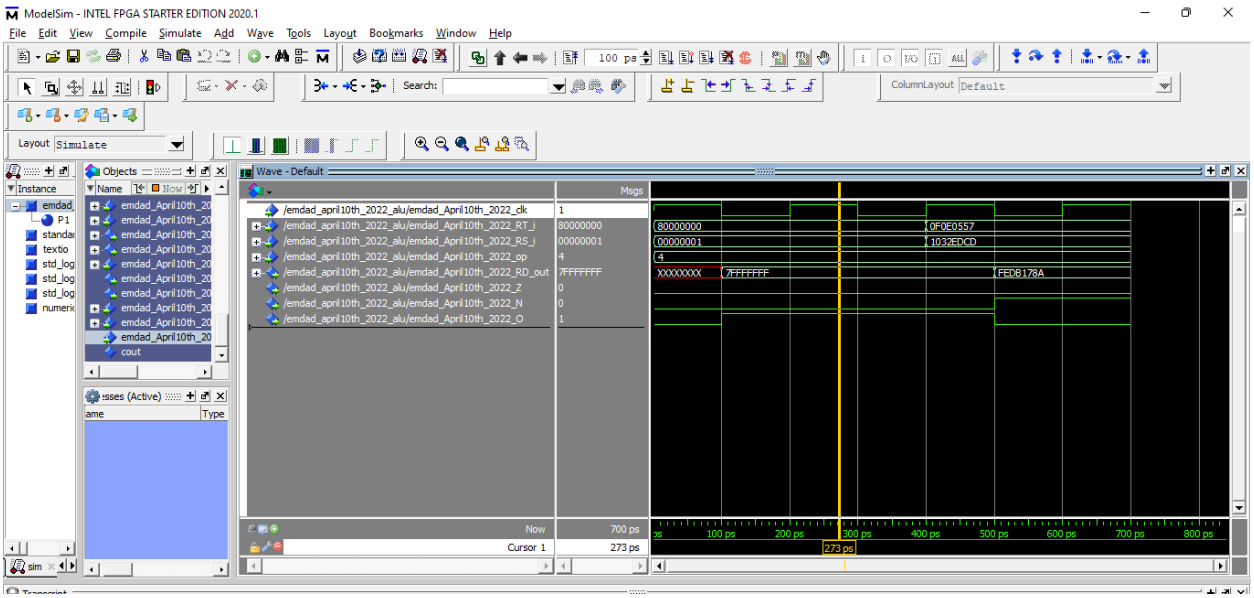


Figure 32: MIPS Instructions Sub simulation

In figure 32, we can see the simulation for SUB of MIPS instructions followed the equation, R[rd] = R[rs] - R[rt]. I forced values into RS and RT while keep changing the clock signal. The

operation code is 4 in hexadecimal (0100 in binary). The zero flag and negative flag is 0 because the result is not 0 and positive and there is no overflow occurred as the operation was unsigned.

SUBU:

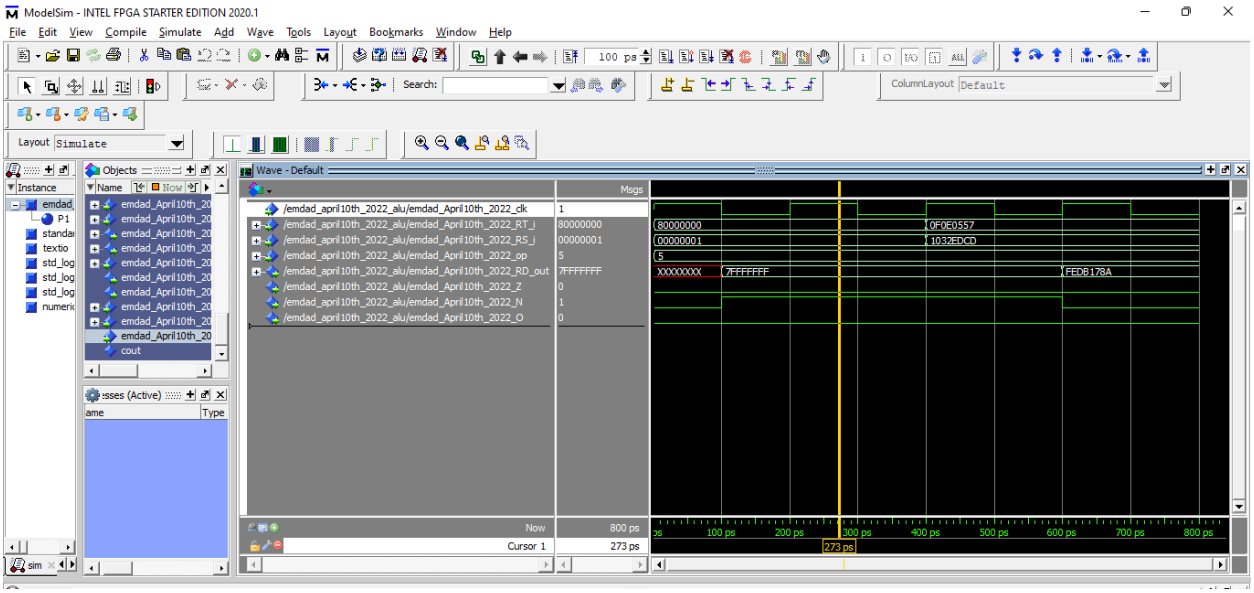


Figure 33: MIPS Instructions SUBU simulation

In figure 33, we can see the simulation for SUBU of MIPS instructions followed the equation, R[rd] = R[rs] - R[rt]. I forced values into RS and RT while keep changing the clock signal. The operation code is 5 in hexadecimal (0101 in binary). The zero flag is not 0 because the result is 0 and there is no overflow occurred as the operation was unsigned. But the negative flag is 1 because the result is negative (F in hexadecimal, 1111 in binary).

AND:

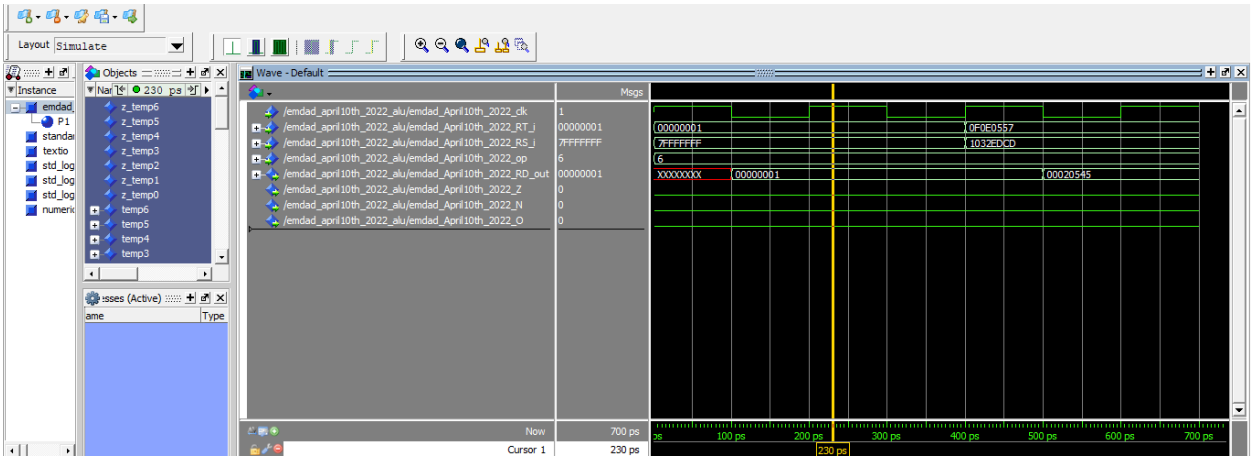


Figure 34: MIPS Instructions AND simulation

In figure 34, we can see the simulation for AND of MIPS instructions followed the equation, R[rd] = R[rs] & R[rt]. I forced values into RS and RT while keep changing the clock signal. As this is a logical operation, all the flags are 0.

NOR:

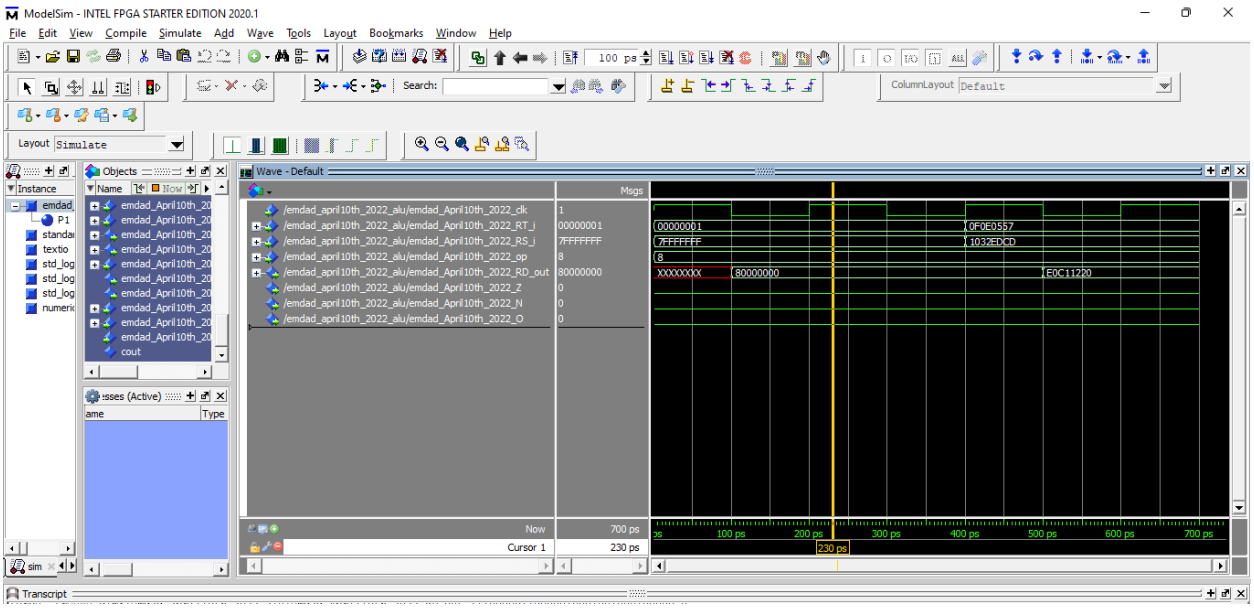


Figure 35: MIPS Instructions NOR simulation

In figure 35, we can see the simulation for NOR of MIPS instructions followed the equation, R[rd] = ~ (R[rs] | R[rt]). I forced values into RS and RT while keep changing the clock signal. As this is a logical operation, all the flags are 0.

OR:

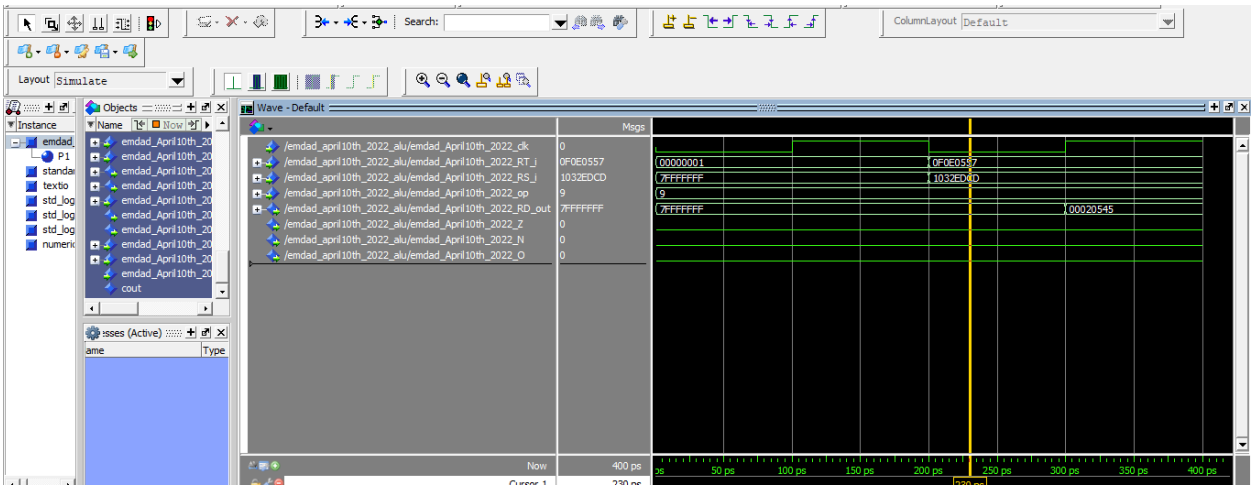


Figure 36: MIPS Instructions OR simulation

In figure 36, we can see the simulation for OR of MIPS instructions followed the equation, R[rd] = R[rs] | R[rt]. I forced values into RS and RT while keep changing the clock signal. As this is a logical operation, all the flags are 0.

SLL:

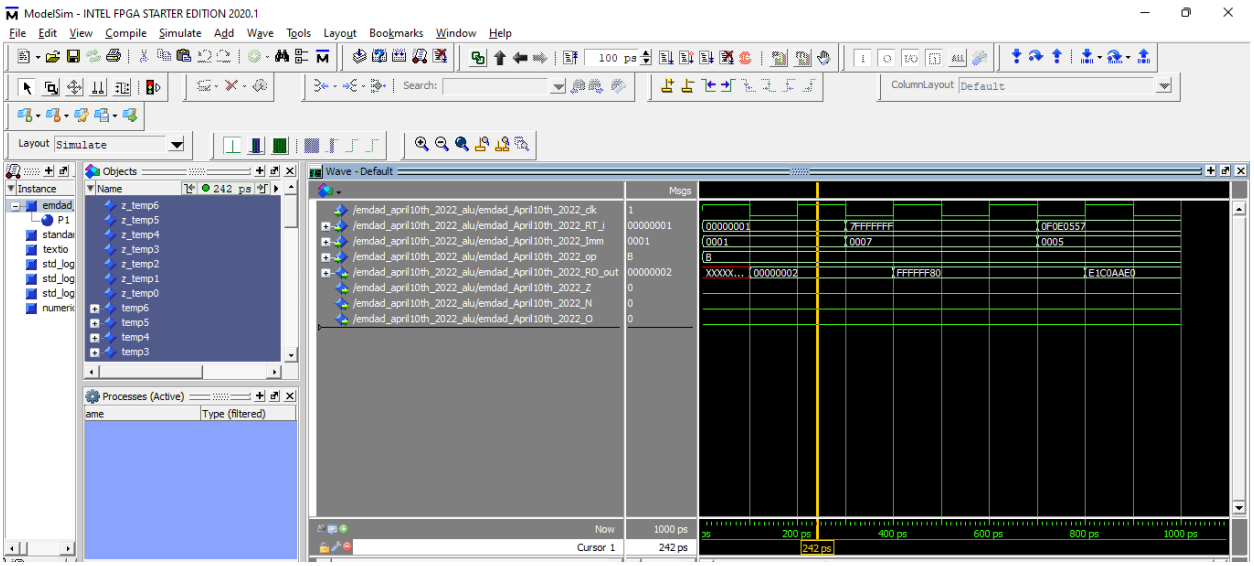


Figure 37: MIPS Instructions SLL simulation

In figure 37, we can see the simulation for SLL of MIPS instructions followed the equation R[rd] = R[rt] << shamt. I forced values into RS and RT while keep changing the clock signal. As this is a logical operation, all the flags are 0.

SRL:

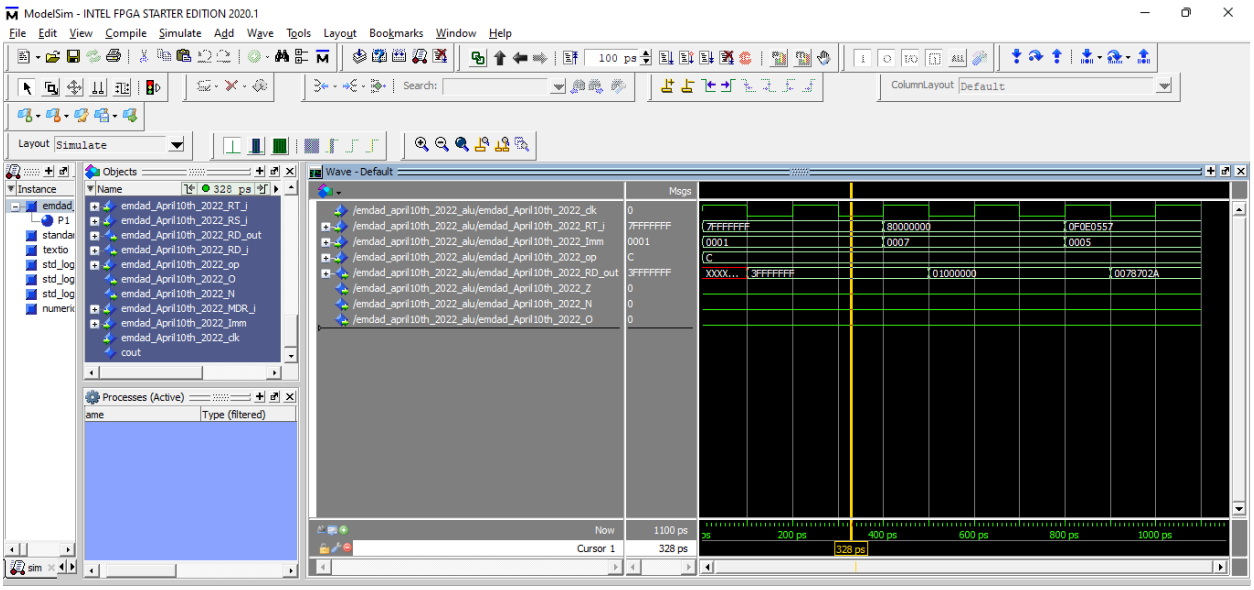


Figure 38: MIPS Instructions SRL simulation

In figure 38, we can see the simulation for SRL of MIPS instructions followed the equation R[rd] = R[rt] >> shamt. I forced values into RS and RT while keep changing the clock signal. As this is a logical operation, all the flags are 0.

SRA:

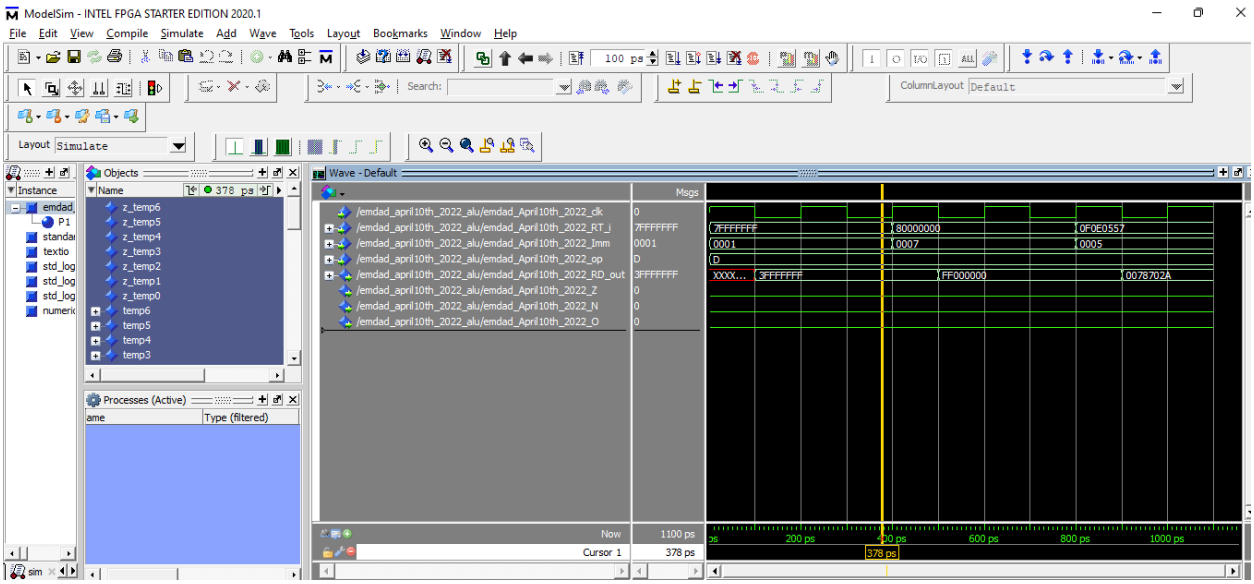


Figure 39: MIPS Instructions SRA simulation

In figure 38, we can see the simulation for SRA of MIPS instructions followed the equation R[rd] = R[rt] >> shamt. I forced values into RS and RT while keep changing the clock signal. As this is a logical operation, all the flags are 0.

I-type Instructions:

i. ADDI:

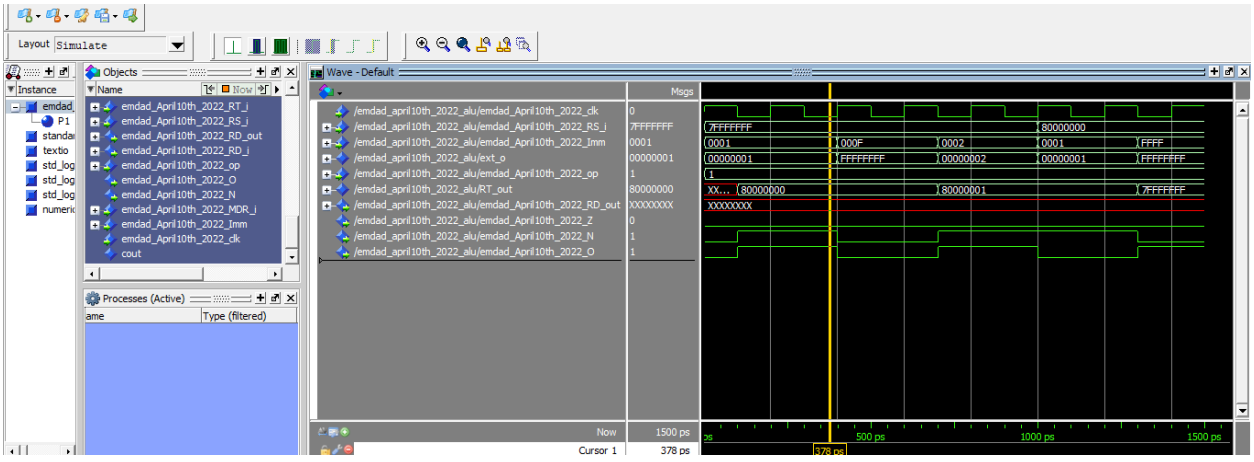


Figure 40: MIPS Instructions ADDI simulation

In figure 40, we can see the simulation for ADDI of MIPS instructions followed the equation R[rt] = R[rs] + SignExtImm. The simulation shows that sign extender is working correctly as it extended 16-bit register into 32-bit. The zero flag is 0 because the result is not 0. The negative and overflow is 1 because the result is negative.

ii. ADDIU:

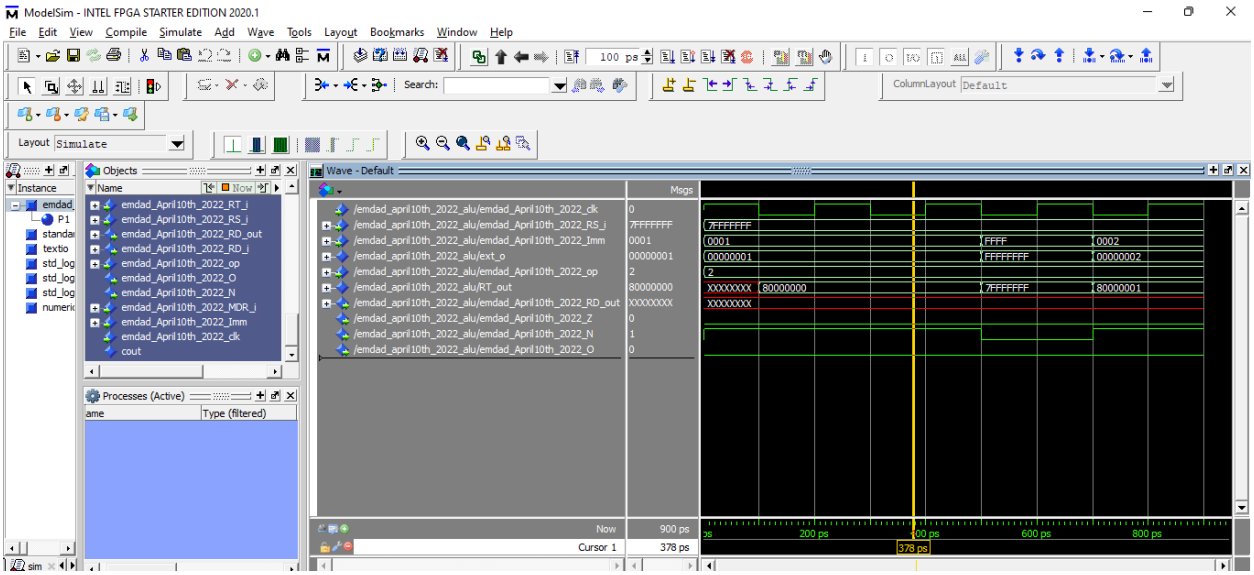


Figure 41: MIPS Instructions ADDIU simulation

In figure 41, we can see the simulation for ADDIU of MIPS instructions. The simulation shows that sign extender is working correctly as it extended 16-bit register into 32-bit. The zero flag is 0 because the result is not 0. The negative is 1 because the result is negative, and the overflow flag is 0.

iii. ANDI:

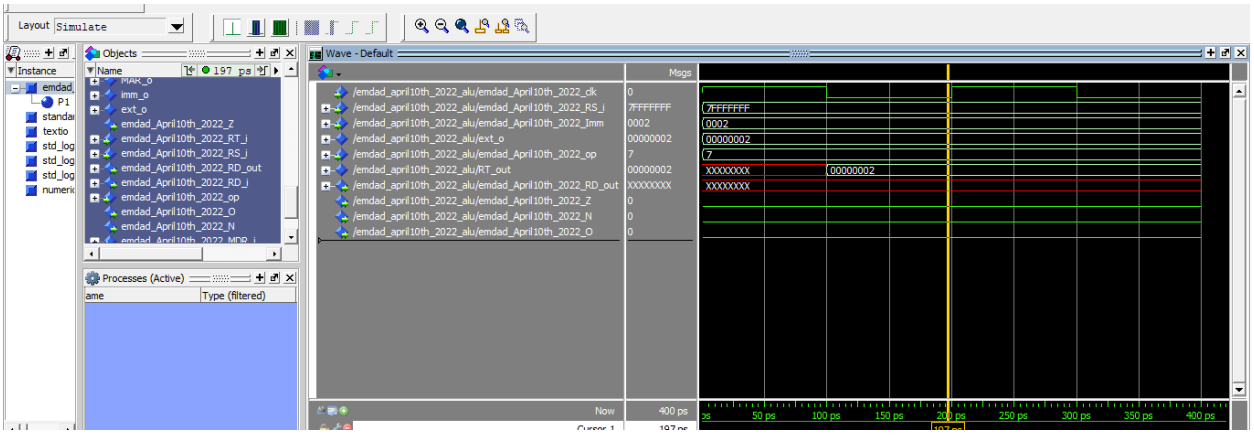


Figure 42: MIPS Instructions ANDI simulation

In figure 42, we can see the simulation for ANDI of MIPS instructions following by the equation R[rt] = R[rs] & ZeroExtImm. The simulation shows that sign extender is working correctly as it extended 16-bit register into 32-bit. As this is a logical operation, all the flags are 0.

iv. ORI:

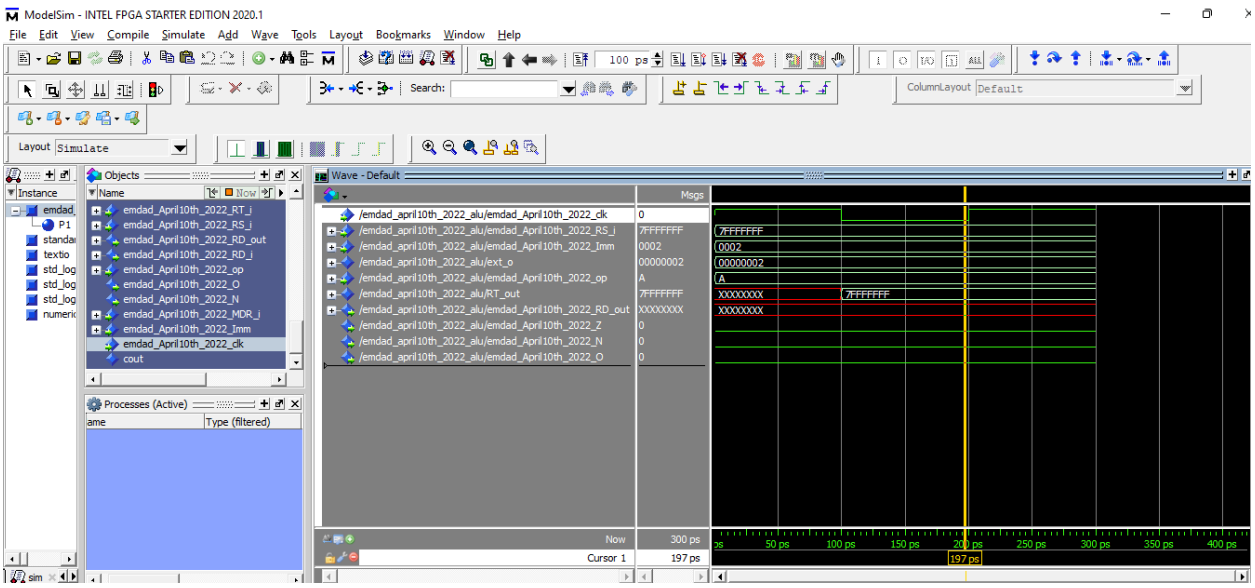


Figure 43: MIPS Instructions ORI simulation

In figure 42, we can see the simulation for ORI of MIPS instructions following by the equation R[rt] = R[rs] | ZeroExtImm. The simulation shows that sign extender is working correctly as it extended 16-bit register into 32-bit. As this is a logical operation, all the flags are 0.

Memory Access Instructions Operations:

i. LW Load Word:

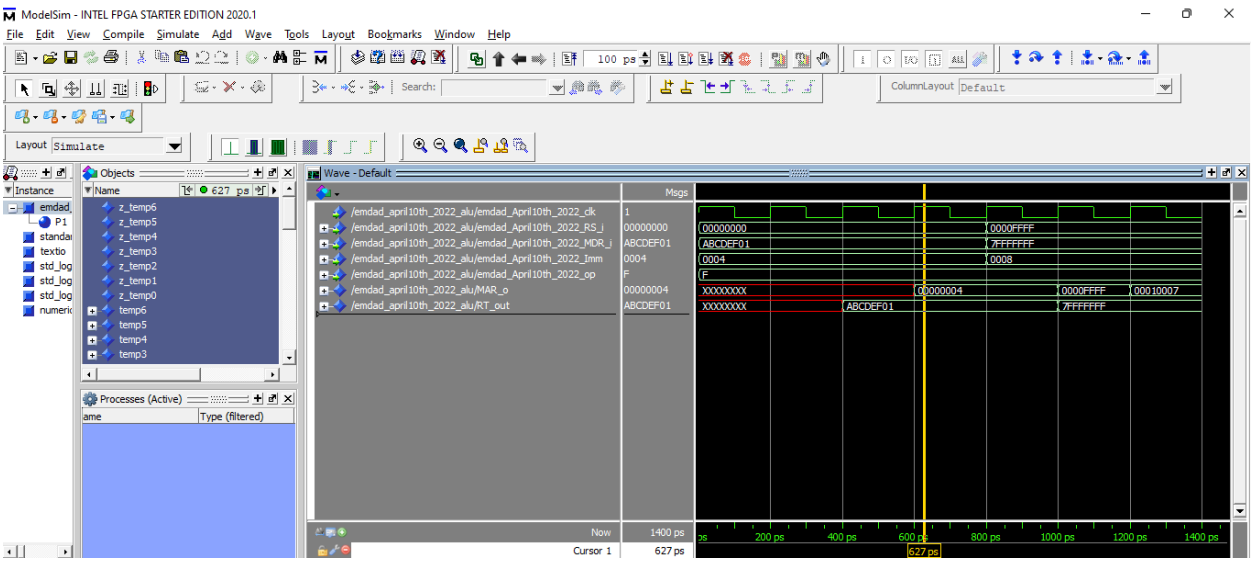


Figure 44: MIPS Instructions LW Load Word Simulation

In figure 44, we can see the simulation for LW Load Word of MIPS instructions following by the equation R[rt] = M[R[rs] + SignExtImm]. While changing the clock signal, I forced values into RS, MDR and IMM.

ii. SW Store Word:

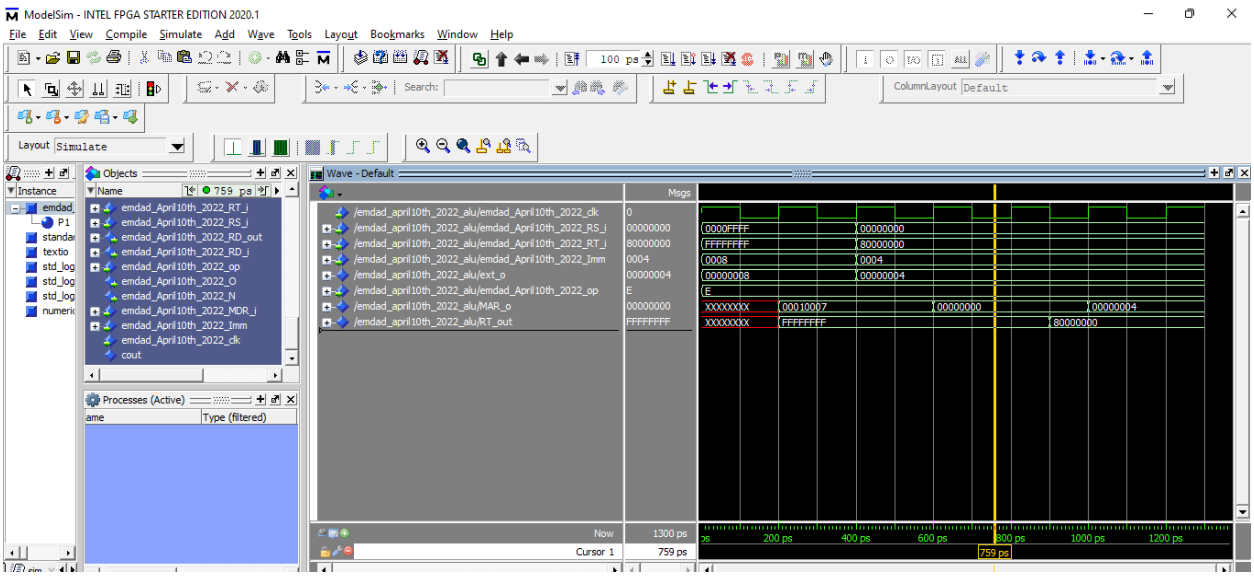


Figure 45: MIPS Instructions SW Store Word Simulation

In figure 45, we can see the simulation for SW Store Word of MIPS instructions following by the equation M[R[rs] + SignExtImm] = R[rt]. While changing the clock signal, I forced values into RS, MDR and IMM.

Conclusion:

In this assignment, I learned how to design and understand the specifications of VHDL extender using Quartus and Modelsim. It was a good practice on Modelsim for circuit simulation. The experiment taught me how to implement MIPS instructions in VHDL and different parts of ALU. The operations logics were helpful to learn easily. I relearned and reviewed the concept again and which will help me in the course further.