

FINAL Lab Project: Single Cycle CPU-LITE

MdShahid Bin Emdad

May 22, 2022

CSC 34200/34300

Table of Contents

Objective:.....	2
Description of Specifications, and Functionality:.....	2
Components:.....	2
R-type Instructions:.....	3
I-type Instructions:	3
Design Implementation:	4
Single Cycle CPU lab:.....	5
Multiplexer:.....	7
Register File:	8
Sign Extender:	9
Program Counter:.....	10
Data Memory:	11
Instruction Register:	12
Instruction Memory:.....	13
Arithmetic Logic Unit (ALU):	14
Control Unit:	15
MIPS:	16
Block Diagrams:	19
MIPS:	21
Simulation:.....	22
Conclusion:.....	24

Objective:

The objective of this assignment is to learn and develop a single cycle CPU. We were instructed to design and implement in VHDL CPU controller that generates control signals to determine the data path for each instruction. We needed to write a program to compute the sum of integers using instructions and single cycle CPU. Overall, this lab is to design single cycle CPU based on the MIPS instruction set architecture

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 (ns unit).

We needed to implement MIPS instructions listed below:

add	addi	addiu	addu	sub	subu	and	andi	nor	ori	sll	srl	sra	sw	lw	beq	bne	j
---------------------	----------------------	-----------------------	----------------------	---------------------	----------------------	---------------------	----------------------	---------------------	---------------------	---------------------	---------------------	---------------------	--------------------	--------------------	---------------------	---------------------	-------------------

Components:

Half Adder Ports: I used two inputs one for the first operand of the adder and other one is the second operand of the adder. I also used carry which is the bit that is carried in from the next less significant stage.

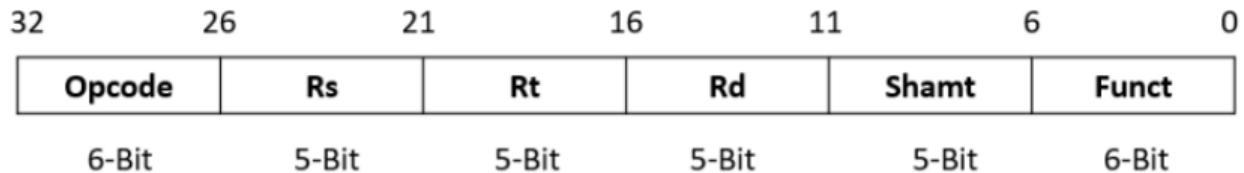
Full Adder Ports: I used two inputs one for the first operand of the adder and other one is the second operand of the adder. I also used cin which is the bit that is carried in from the next less significant stage. I used cout for the output which is bit that is carried over from an addition when necessary and sum is the last bit of the result.

Instruction Memory: I used 2 cins, (cin1 and cin2) which are the first and second operand of the adder for the 16 bits. I also had op which is the bit that carried in from the less significant stage. For the outputs, I had neg which is the carry output that carried in from addition, sum (sum of the two inputs, cin1 and cin2), overflow which checks for overflow errors and finally zero for two input addition.

Data Memory: I used two operands. The first operand of the adder and other one is the second operand of the adder. Then, I used obit which is the bit that carried from the less significant stage. I had carried for the bit output from addition, sum for the inputs addition, overflow which checks for overflow errors and finally zero for two input addition.

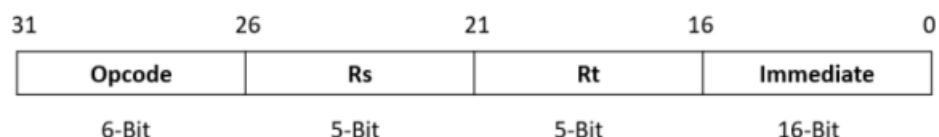
ALU: The inputs are 32-bit registers (RS, RT) to compute operations, 16-bit immediate to compute immediate operations, 32-bit MDR, clock and operation. The outputs are 32-bit register RD and 3 flags: zero, negative, overflow. I also used the other codes as a component to get the waveform.

Extender: 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.

R-type Instructions:

The R-type instruction is divided into 6 parts according to the above structure. First, the opcode is 0b000000. For all R-type commands. Second, the Rs, Rt, and Rd fields are all two source and destination indexes. Each register in addition, the Shamt field specifies the number of bits to shift the contents of a. Registers-Applies only to shift instructions sll, srl, and sra.

R-type Instructions					
	Add	Sub	OR	Mult	Div
RegDst	1	1	1	1	1
AluSRC	0	0	0	0	0
MemToReg	0	0	0	0	0
RegWrite	1	1	1	1	1
MemWrite	0	0	0	0	0
PCSrc	0	0	0	0	0
ExtOp	X	X	0	X	X
ALUOp	0000	0001	0010	0100	0101

I-type Instructions:

Unlike the RType statement, the IType statement is split into four fields according to the structure above. First, the opcode distinguishes between IType statements. Next, Rs and Rt are the source indexes / addresses of the two source registers. Finally, the last 16 bits or immediately after are zero-extended or sign-extended to a 32-bit value, depending on the operation.

I-type Instructions					
	AddI	ORI	LW	SW	Beq
RegDst	0	0	0	X	X
AluSRC	1	1	1	1	1
MemToReg	0	0	1	X	X
RegWrite	1	1	1	0	0
MemWrite	0	0	0	1	0
PCSrc	0	0	0	0	1
ExtOp	1	0	1	1	X
ALUOp	0000	0010	0000	0000	0000

Design Implementation:

Now we have the breakdowns of the instructions and components, we need go for design implementation. The instructions follow different types of formats, not all instructions will handle the same way. Therefore, we need some mechanism to organize and control the data paths for each instruction. We can see in the figure 1 the CPU controller operation.

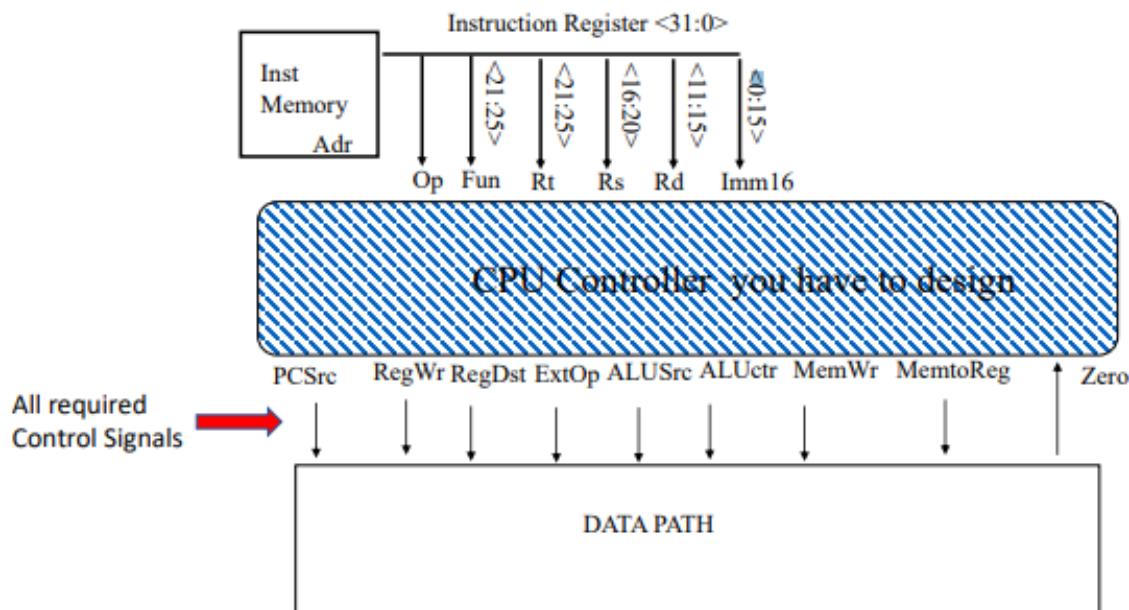


Figure 1: Visualization of CPU Controller Operation

Single Cycle CPU lab:

Signal	Value = 0	Value = 1
Branch	$PC \leq PC + 4$	$PC \leq PC + 4 + \{SignExt(Imm), \ll 2\}$
ALUsrc	Operand 2 \leq BusB	Operand 2 \leq SignExt(Imm)
ExtOp	Zero Extend	Sign Extend
RegWrite	No write	The register on the BusW input is written with the value on the Write data input
RegDst	ALUResult < RT	ALUResult < RD
MemWrite	No write	Write to memory
MemRd	No read	Read from memory
MemtoReg	The value fed to the register BusW input comes from the ALU	The value fed to the register BusW input comes from the Data Memory
Jump	No jump	Jump to target address
ALUCtrl	4-Bit vector that serves as an input to the Arithmetic Logic Unit (ALU) to perform the necessary arithmetic operation.	4-Bit vector that serves as an input to the Arithmetic Logic Unit (ALU) to perform the necessary arithmetic operation.

In figure 2, we can see the “A Single Cycle CPU” block diagram. This is the last step and the intended design to create. We will be using Quartus to build it from the VHDL code.

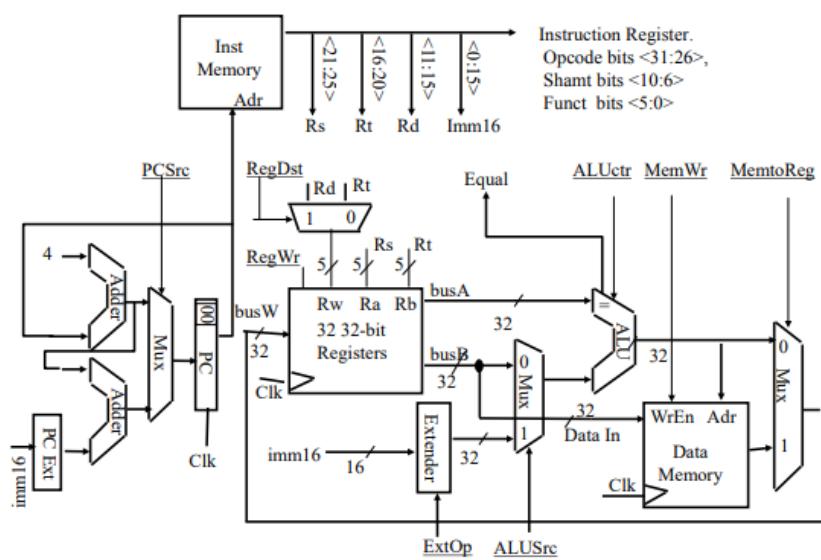


Figure 2: A Single Cycle CPU

In figure 3, we can see the project directory for the CPU lab.

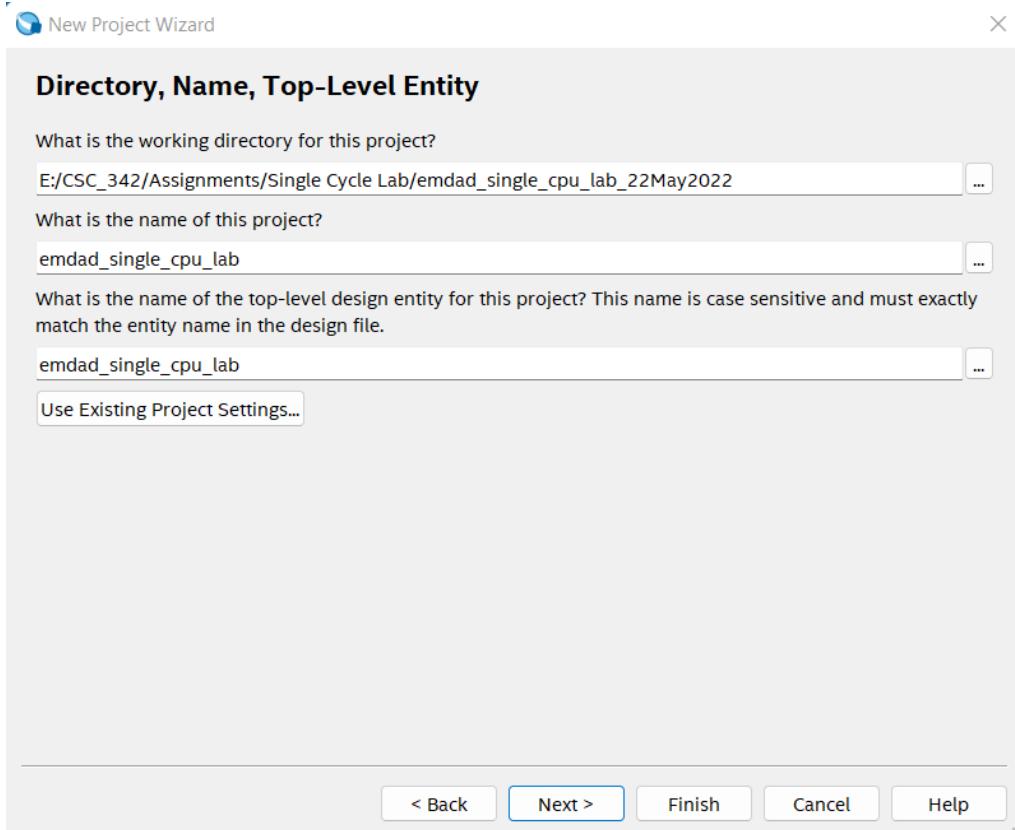


Figure 3: project directory for the CPU lab.

In figure 4, we can see the project summary for the CPU lab project.

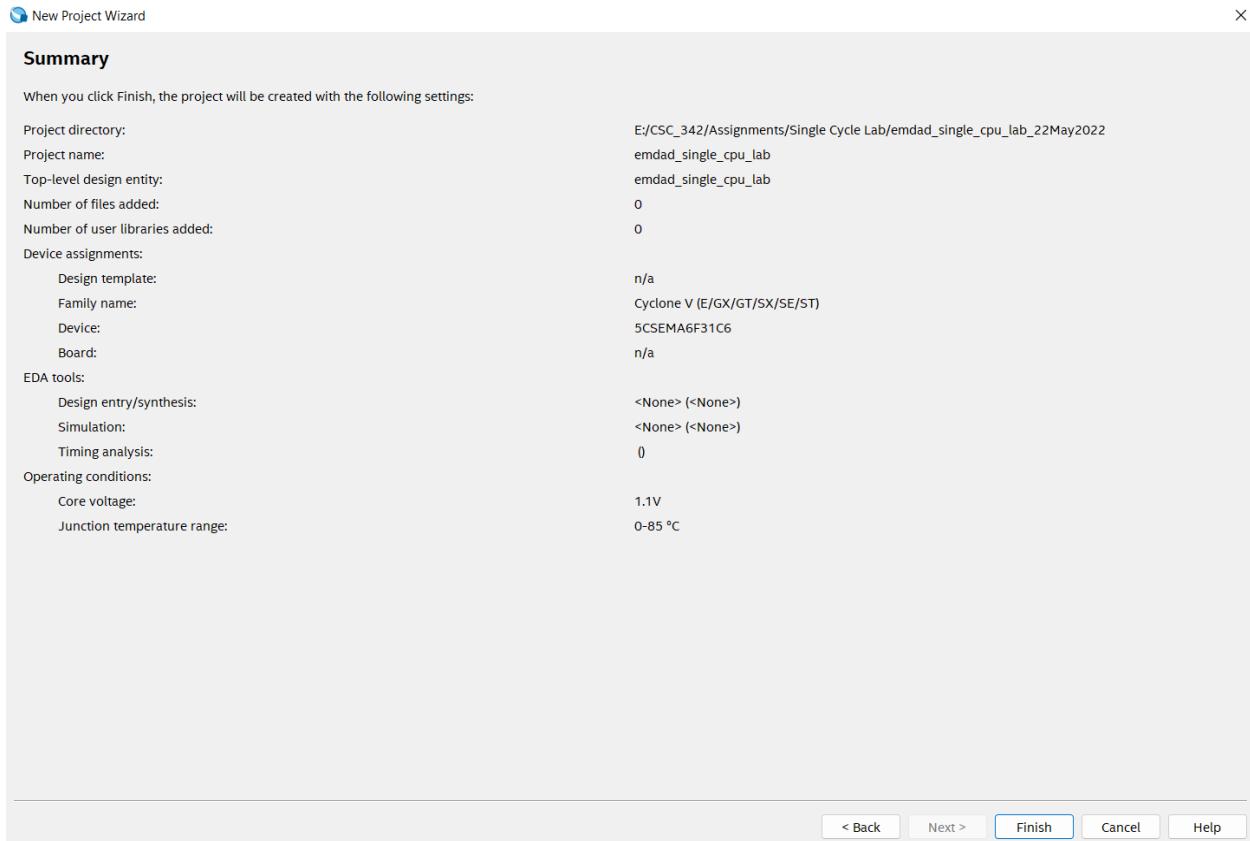


Figure 4: project summary for the CPU lab project.

Multiplexer:

In figure 5, we can see the VHDL code for multiplexer. I have 3 inputs and 1 output for this code.

```

Quartus Prime Lite Edition - E:/CSC_342/Assignments/Single Cycle Lab/emdad_single_cpu_lab_22May2022/emdad_single_cpu_lab - emdad_single_cpu_lab
File Edit View Project Assignments Processing Tools Window Help
Project Navigator Files emdad_single_cpu_lab emdad_mux_22ndmay2022.vhd Compilation Report - emdad_single_cpu_lab
emdad_mux_22ndmay2022.vhd 267 268
1 Library ieee;
2 use ieee.std_logic_1164.all;
3
4 Entity emdad_mux_22ndmay2022 is
5 port (
6     emdad_22ndmay2022_input1 : in std_logic_vector(31 downto 0);
7     emdad_22ndmay2022_input2 : in std_logic_vector(31 downto 0);
8     emdad_22ndmay2022_sel : in std_logic;
9     emdad_22ndmay2022_output : out std_logic_vector(31 downto 0)
10 );
11 end emdad_mux_22ndmay2022;
12
13 Architecture struct of emdad_mux_22ndmay2022 is
14
15 begin
16     begin
17         with emdad_22ndmay2022_sel select
18             emdad_22ndmay2022_output <= emdad_22ndmay2022_input2 when '1',
19             emdad_22ndmay2022_input1 when others;
20     end struct;

```

Tasks Compilation

Task	Time
43% ▾ > Compile Design	00:01:02
✓ > Analysis & Synthesis	00:00:11
73% ▾ > Fitter (Place & Route)	00:00:51
0% > Assembler (Generate p...)	00:00:00
0% > Timing Analysis	00:00:00
> EDA Netlist Writer	
Edit Settings	
Program Device (Open Pr...)	

Figure 5: VHDL code for multiplexer

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

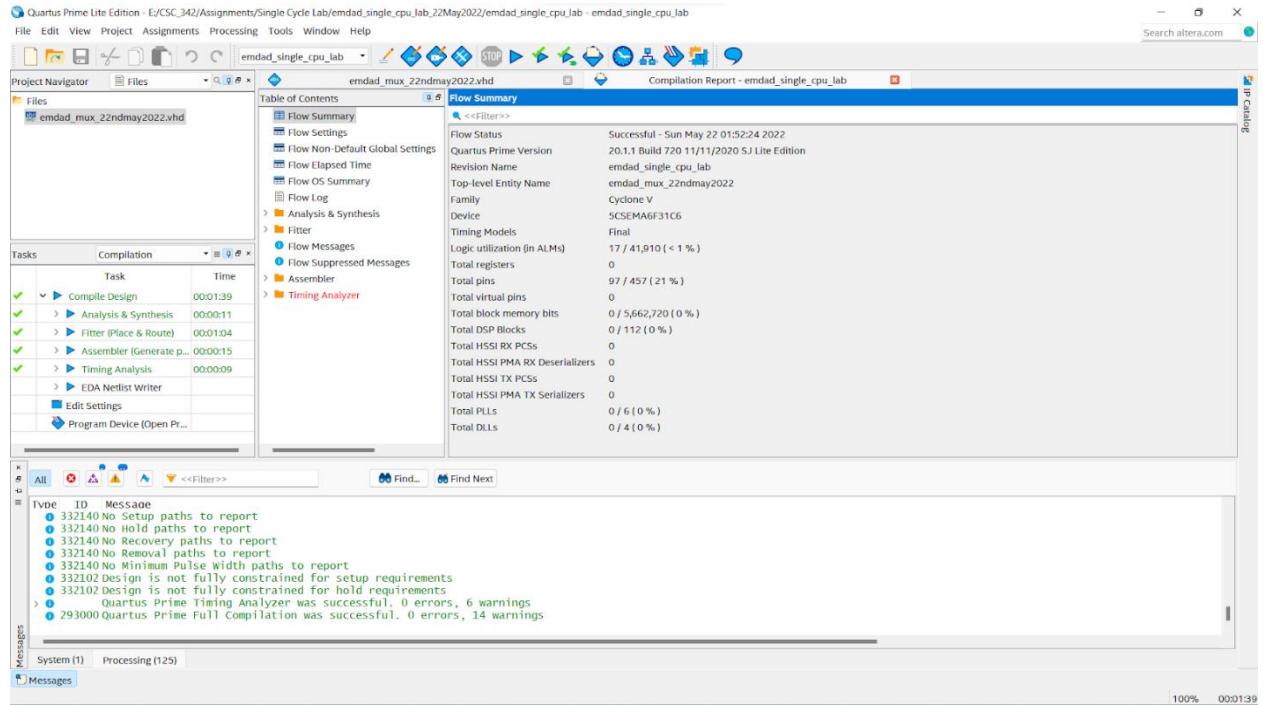


Figure 6: compiled successfully

Register File:

In figure 7, we can see the VHDL code the register file. I have 6 inputs and 2 outputs for the registers read and output.

```

1  library IEEE;
2  use IEEE.STD.TEXTIO.all;
3  use IEEE.NUMERIC_STD.all;
4
5  entity emdad_Register_file_22ndmay2022 is
6    port
7      (
8        emdad_22ndmay2022_clk : in STD_LOGIC;
9        emdad_22ndmay2022_reqwrite : in STD_LOGIC;
10       emdad_22ndmay2022_writedata : in STD_LOGIC_VECTOR(4 DOWNTO 0);
11       emdad_22ndmay2022_readdata : out STD_LOGIC_VECTOR(31 DOWNTO 0);
12       emdad_22ndmay2022_ReadRegister1 : in STD_LOGIC_VECTOR(4 DOWNTO 0);
13       emdad_22ndmay2022_ReadRegister2 : in STD_LOGIC_VECTOR(4 DOWNTO 0);
14       emdad_22ndmay2022_ReadData1 : out STD_LOGIC_VECTOR(31 DOWNTO 0);
15       emdad_22ndmay2022_ReadData2 : out STD_LOGIC_VECTOR(31 DOWNTO 0)
16      );
17  end emdad_Register_file_22ndmay2022;
18
19 architecture struct of emdad_Register_file_22ndmay2022 is
20
21   type emdad_22ndmay2022_Reg is array(0 TO 31) of STD_LOGIC_VECTOR(31 DOWNTO 0);
22   signal emdad_22ndmay2022_mem : emdad_22ndmay2022_Reg := (others => "0");
23
24 begin
25   process (emdad_22ndmay2022_clk)
26   begin
27     if (rising_edge(emdad_22ndmay2022_clk)) then
28       if (emdad_22ndmay2022_reqwrite = '1') then
29         emdad_22ndmay2022_mem(to_integer(unsigned(emdad_22ndmay2022_writeRegister))) <= emdad_22ndmay2022_writedata;
30       end if;
31     end if;
32   end process;
33   emdad_22ndmay2022_ReadData1 <= (others => '0') when emdad_22ndmay2022_ReadRegister1 = "00000" else emdad_22ndmay2022_mem(to_integer(unsigned(emdad_22ndmay2022_ReadRegister1)));
34   emdad_22ndmay2022_ReadData2 <= (others => '0') when emdad_22ndmay2022_ReadRegister2 = "00000" else emdad_22ndmay2022_mem(to_integer(unsigned(emdad_22ndmay2022_ReadRegister2)));
35
36 end struct;

```

Figure 7: VHDL code the register file

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

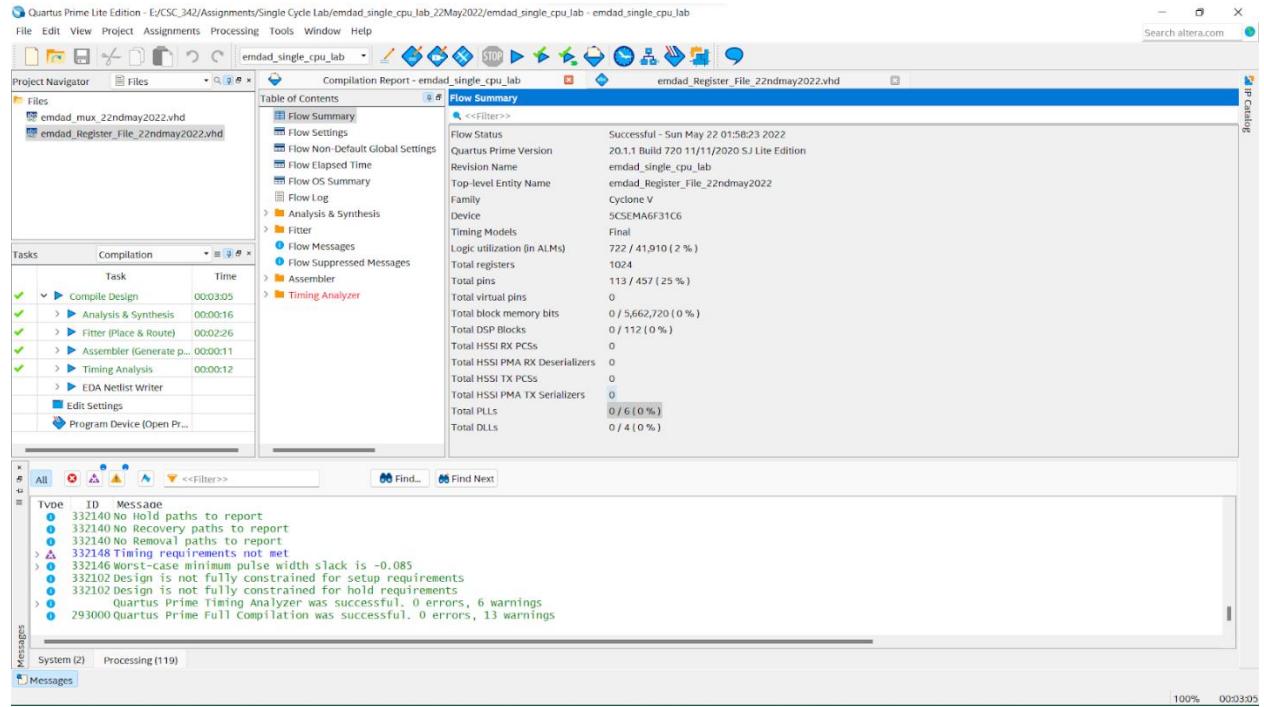


Figure 8: compiled successfully.

Sign Extender:

In figure 9, we can see the VHDL code the sign extender. We will be using to extend the bit from 16bits to 32bits. I have 1 input and 1 output.

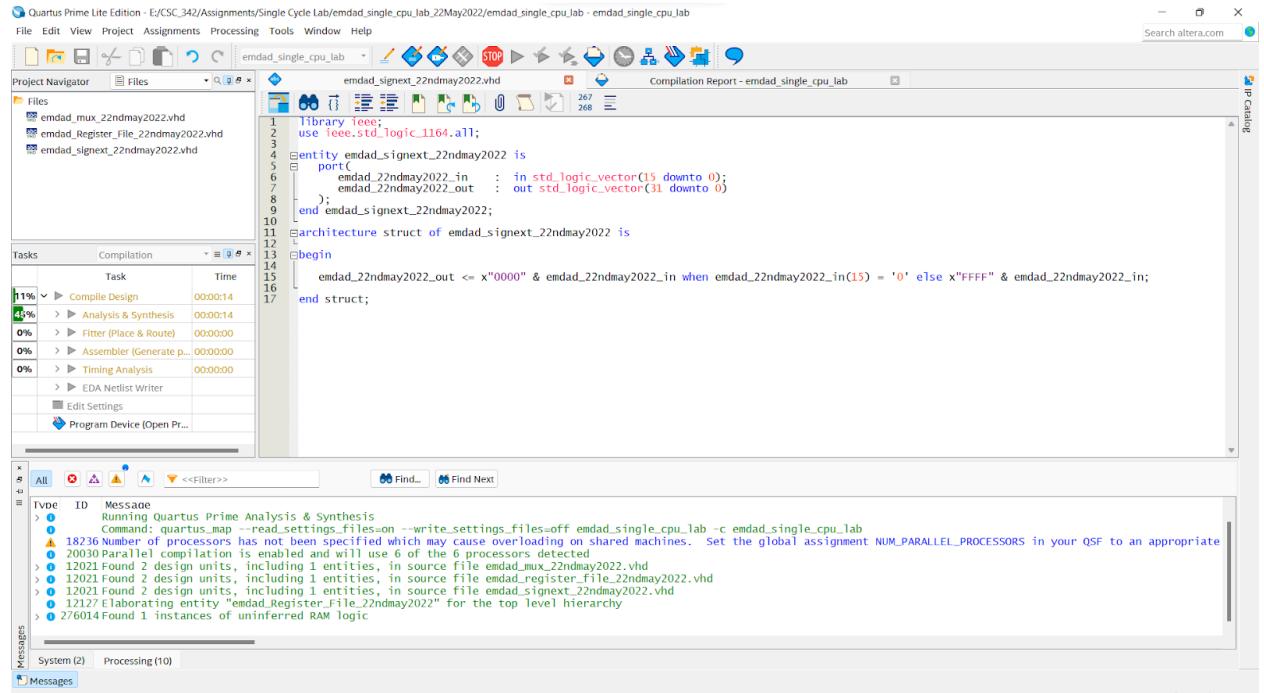


Figure 9: VHDL code the sign extender

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

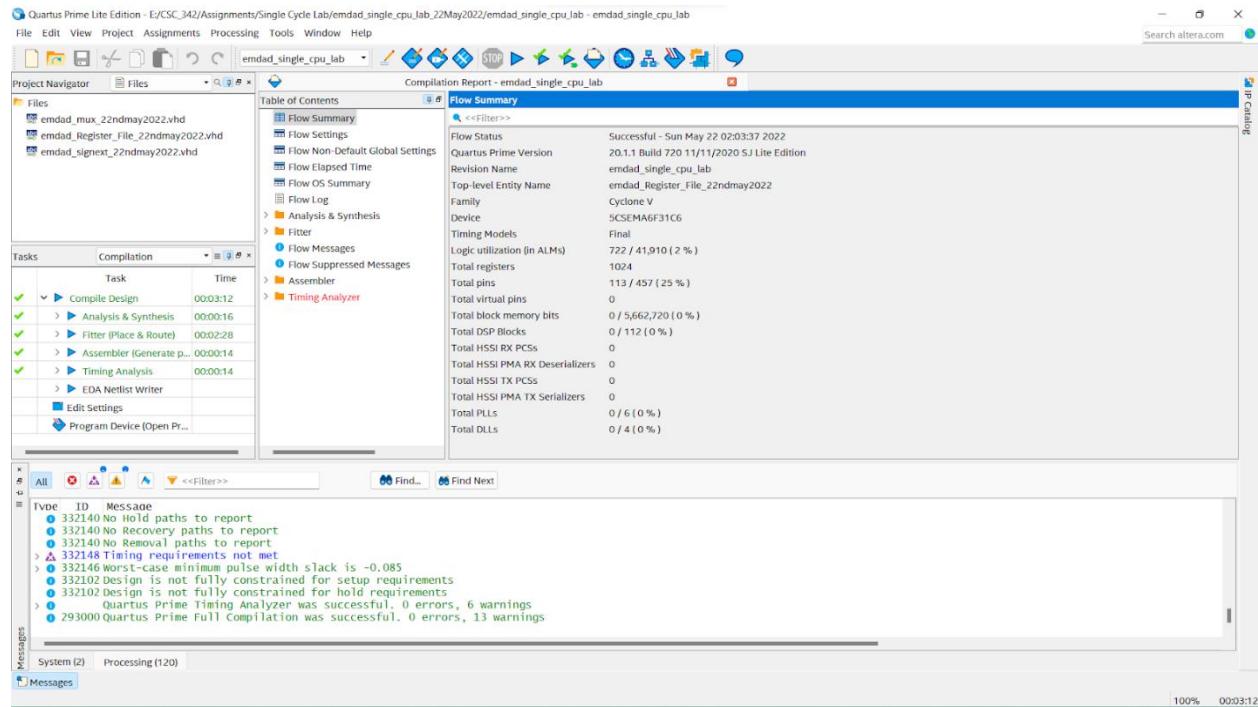


Figure 10: compiled successfully.

Program Counter:

In figure 11, we can see the VHDL code the program counter. I have 3 inputs and 1 output.

```

1 library IEEE;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_signed.all;
4
5 entity emdad_program_counter_22ndmay2022 is
6   port(
7     emdad_22ndmay2022_clk, emdad_22ndmay2022_reset : in std_logic;
8     emdad_22ndmay2022_jump, emdad_22ndmay2022_branch : in std_logic;
9     emdad_22ndmay2022_instruction, emdad_22ndmay2022_imm_Ext : in std_logic_vector(31 downto 0);
10    emdad_22ndmay2022_pc : out std_logic_vector(31 downto 0)
11  );
12 end emdad_program_counter_22ndmay2022;
13
14 architecture struct of emdad_program_counter_22ndmay2022 is
15
16   signal emdad_22ndmay2022_current_pc: std_logic_vector(31 downto 0);
17
18 begin
19
20   process (emdad_22ndmay2022_clk, emdad_22ndmay2022_reset)
21   begin
22     if (emdad_22ndmay2022_reset = '1') then
23       emdad_22ndmay2022_current_pc <= (others => '0');
24     elsif (rising_edge(emdad_22ndmay2022_clk)) then
25       if (emdad_22ndmay2022_branch = '1')then
26         emdad_22ndmay2022_current_pc <= emdad_22ndmay2022_current_pc + x"00000004" + emdad_22ndmay2022_imm_Ext(29 downto 0) & "00";
27       elsif (emdad_22ndmay2022_jump = '1') then
28         emdad_22ndmay2022_current_pc <= (emdad_22ndmay2022_current_pc and x"F0000000") or ("0000" & emdad_22ndmay2022_instruction(25 downto 0) & "00");
29       else
30         emdad_22ndmay2022_current_pc <= emdad_22ndmay2022_current_pc + x"00000004";
31       end if;
32     end if;
33   end process;
34
35   emdad_22ndmay2022_pc <= emdad_22ndmay2022_current_pc;
36 end struct;

```

Figure 11: VHDL code the program counter

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

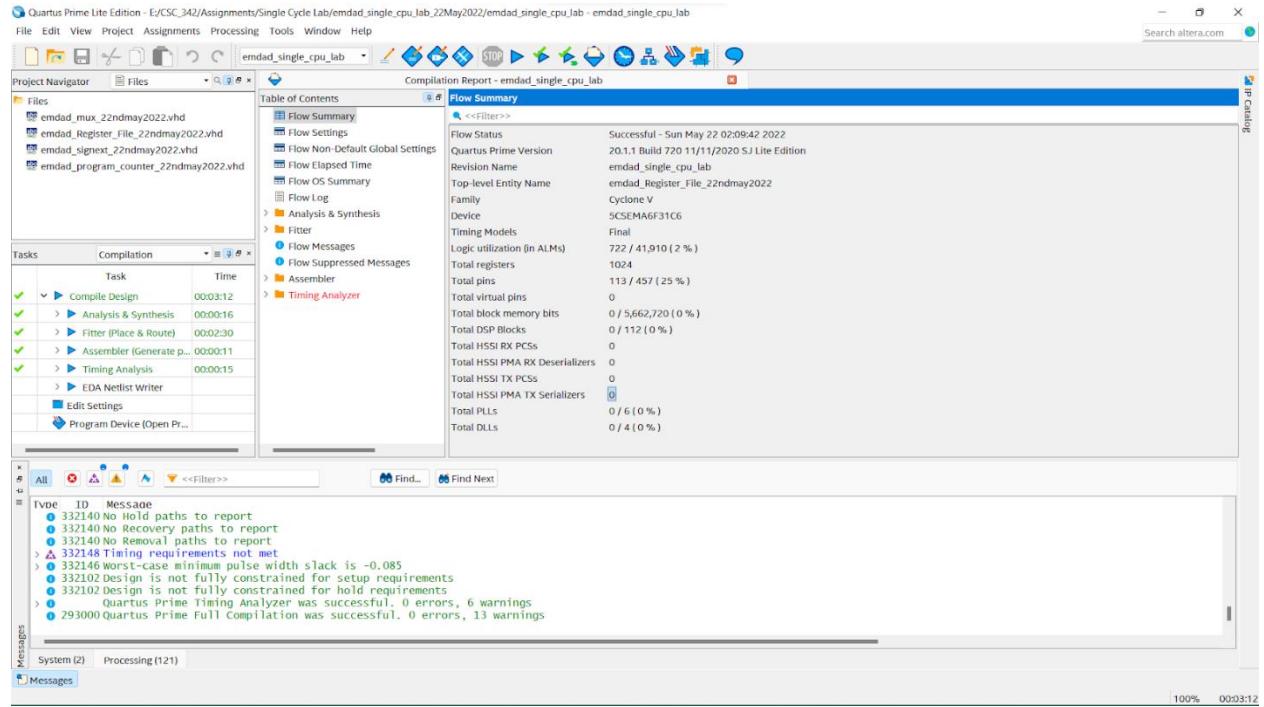


Figure 12: compiled successfully

Data Memory:

In figure 13, we can see the VHDL code the data memory. We will be using to extend the bit from 16bits to 32bits. I have 4 inputs and 1 output.

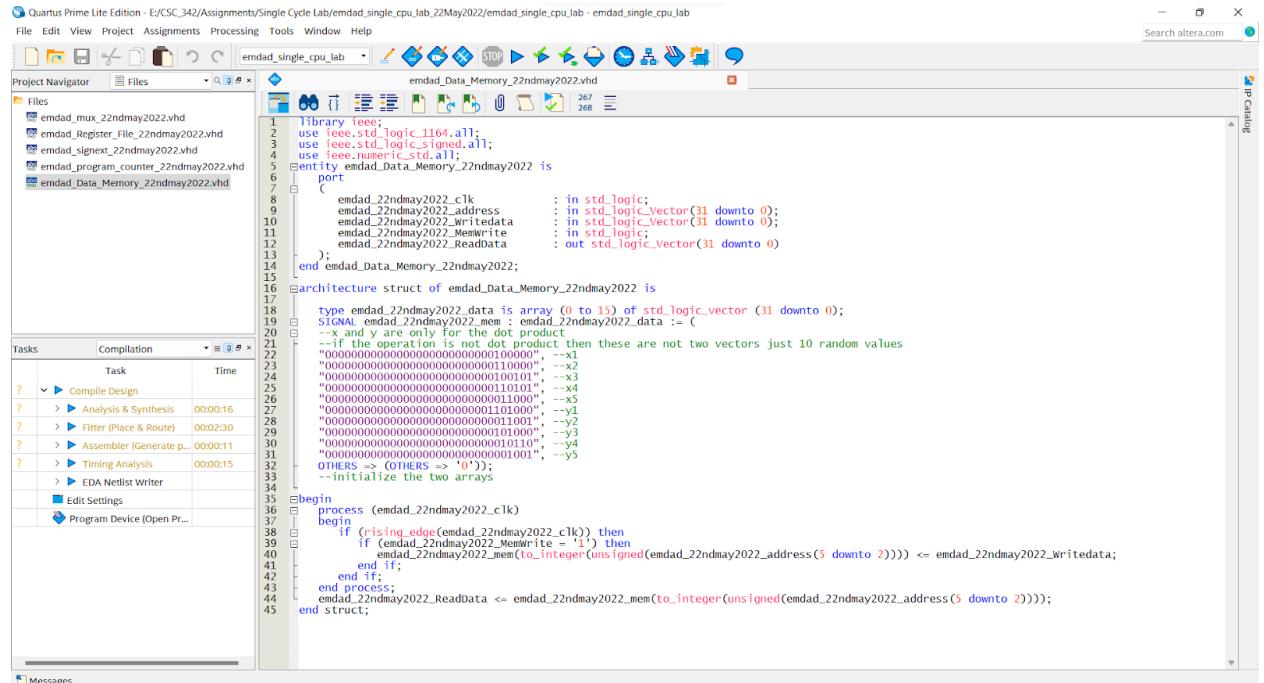


Figure 13: VHDL code the data memory

In figure 14, we can see that the code compiled successfully.

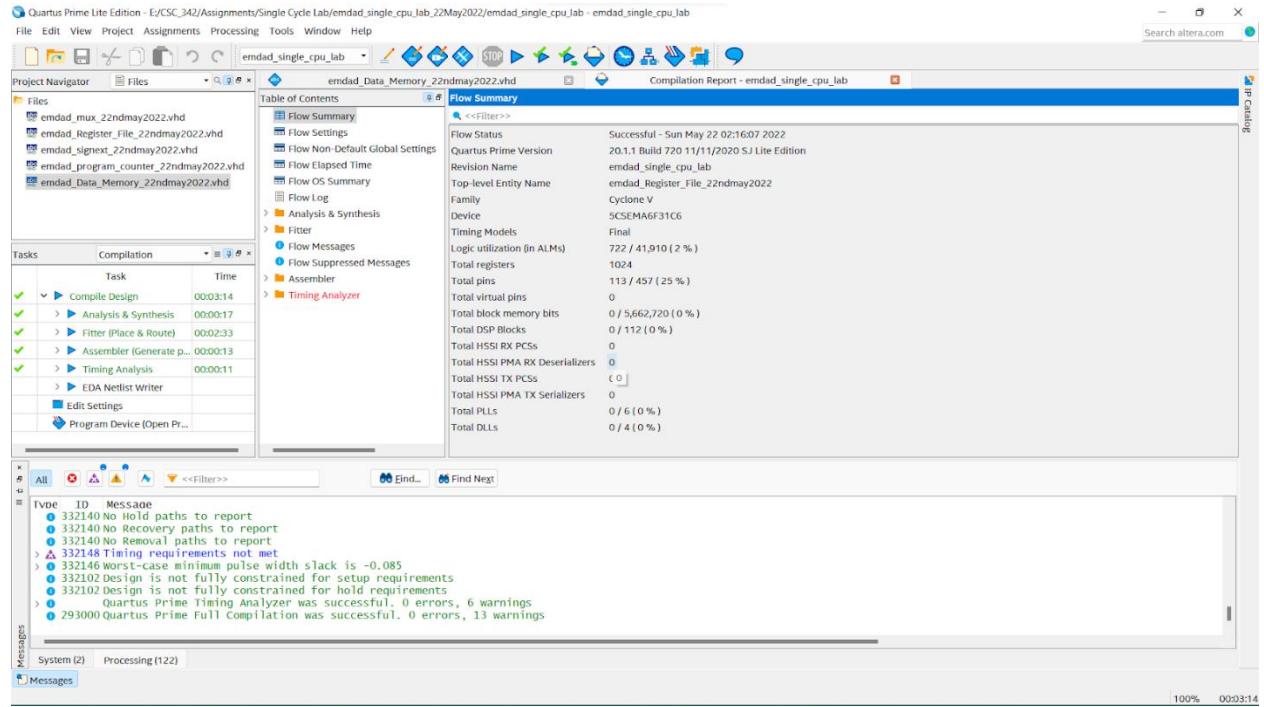


Figure 14: compiled successfully

Instruction Register:

In figure 15, we can see the VHDL code the instruction Register. I have 1 input and 4 outputs.

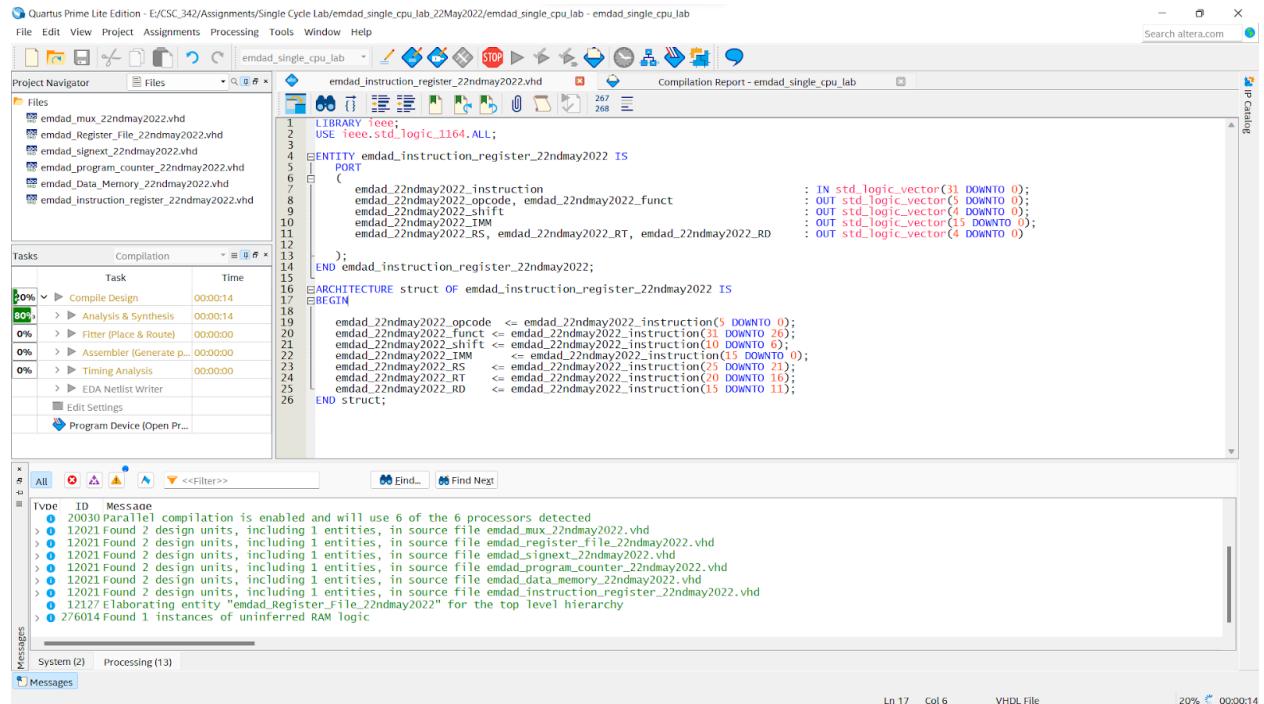


Figure 15: VHDL code the instruction Register

In figure 16,we can see that the code compiled successfully.

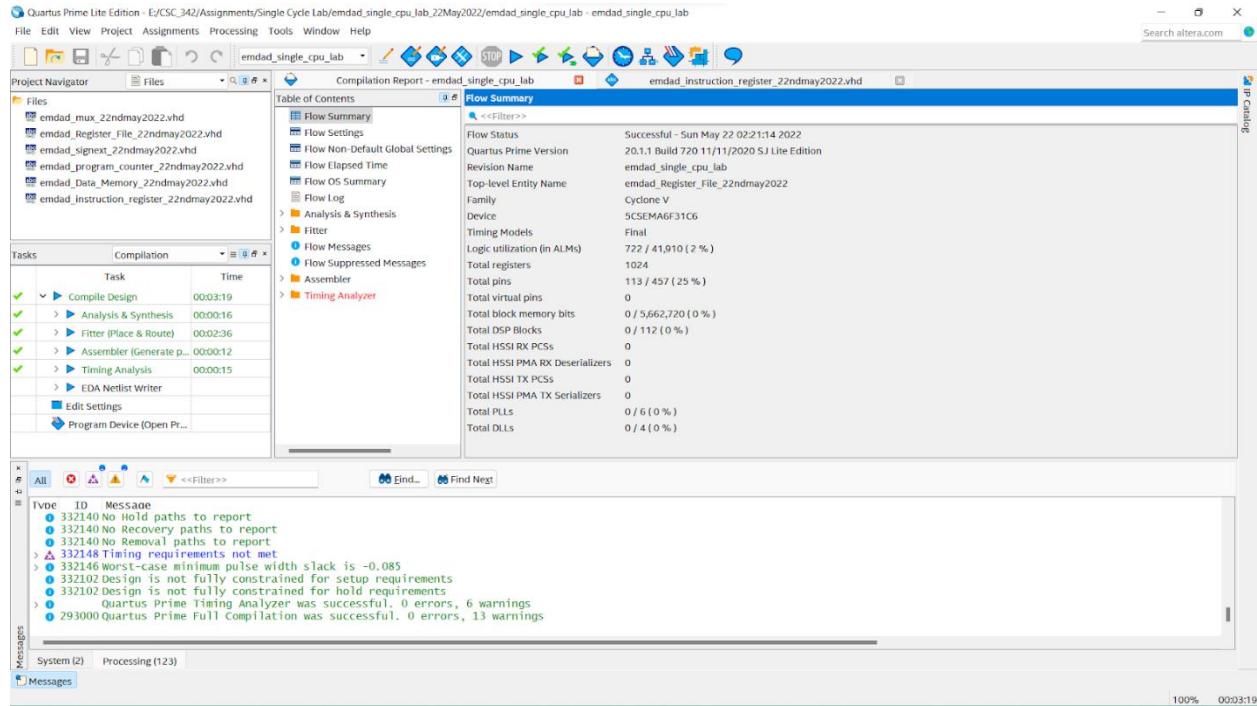


Figure 16: compiled successfully

Instruction Memory:

In figure 17, we can see the VHDL code the instruction memory. I have 1 input and 1 output.

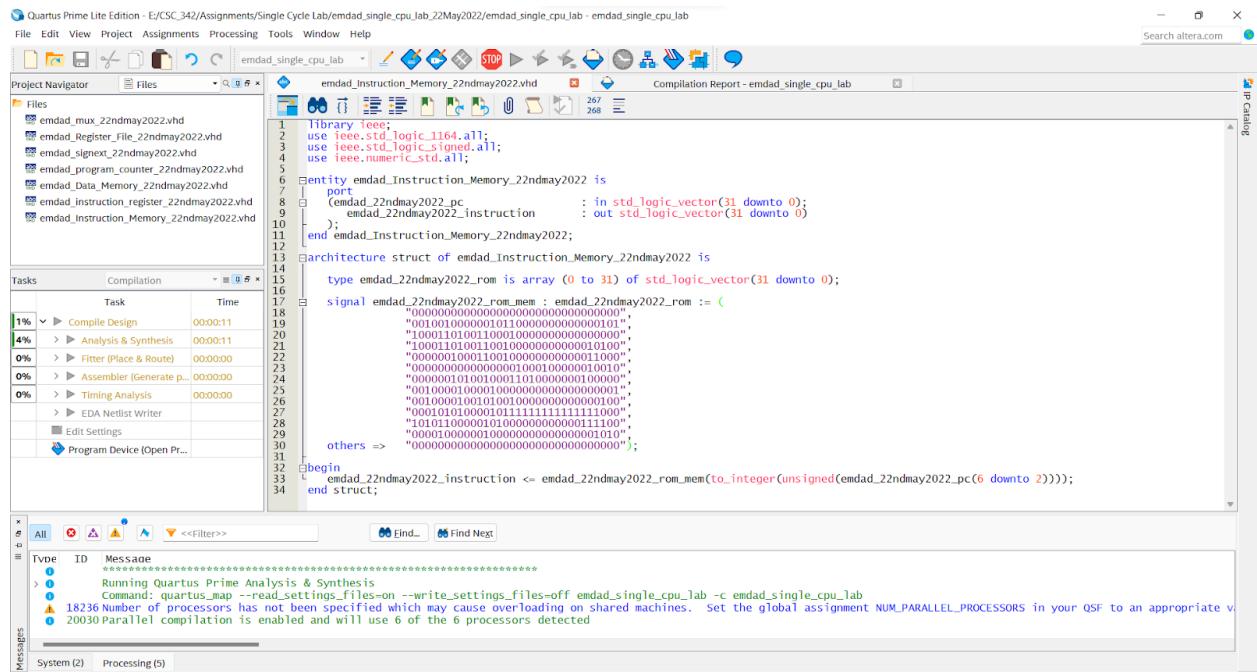


Figure 17: VHDL code the instruction memory

In figure 18, we can see that the code compiled successfully.

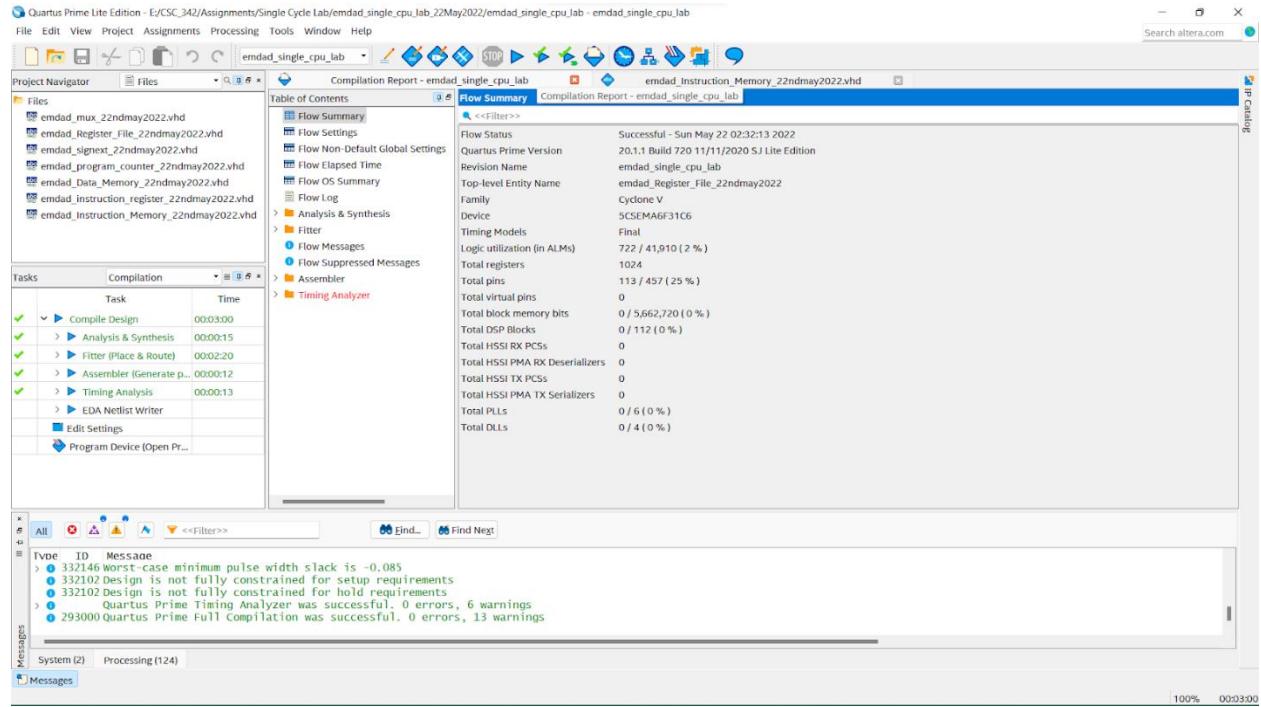


Figure 18: compiled successfully.

Arithmetic Logic Unit (ALU):

In figure 19, we can see the VHDL code the ALU. I have 4 inputs and 3 outputs.

```

1 Library IEEE;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 Entity emdad_alu_22ndmay2022 is
6   port (
7     emdad_22ndmay2022_input1 : in std_logic_vector(31 downto 0);
8     emdad_22ndmay2022_input2 : in std_logic_vector(31 downto 0);
9     emdad_22ndmay2022_op : in std_logic_vector(3 downto 0);
10    emdad_22ndmay2022_shift : in std_logic_vector(4 downto 0);
11    emdad_22ndmay2022_result : out std_logic_vector(31 downto 0);
12    emdad_22ndmay2022_HI_10_result : out std_logic_vector(63 downto 0);
13    emdad_22ndmay2022_zero : out std_logic;
14  );
15 end emdad_alu_22ndmay2022;
16
17
18 Architecture struct of emdad_alu_22ndmay2022 is
19
20   signal emdad_22ndmay2022_result : std_logic_vector(31 downto 0);
21   signal emdad_22ndmay2022_remainder, emdad_22ndmay2022_quotient : std_logic_vector(31 downto 0);
22
23 begin
24
25   emdad_22ndmay2022_result <= std_logic_vector(signed(emdad_22ndmay2022_input1) & signed(emdad_22ndmay2022_input2)) when (emdad_22ndmay2022_op = "0010") else -- SUB
26   std_logic_vector(unsigned(emdad_22ndmay2022_input1) - unsigned(emdad_22ndmay2022_input2)) when (emdad_22ndmay2022_op = "0000") else -- ADD
27   emdad_22ndmay2022_input1 AND emdad_22ndmay2022_input2 when (emdad_22ndmay2022_op = "0100") else -- AND
28   emdad_22ndmay2022_input1 NOR emdad_22ndmay2022_input2 when (emdad_22ndmay2022_op = "0101") else -- NOR
29   emdad_22ndmay2022_input1 OR emdad_22ndmay2022_input2 when (emdad_22ndmay2022_op = "0110") else -- OR
30   std_logic_vector(unsigned(emdad_22ndmay2022_input1) + unsigned(emdad_22ndmay2022_input2)) when (emdad_22ndmay2022_op = "0001") else -- ADDU
31   std_logic_vector(unsigned(emdad_22ndmay2022_input1) - unsigned(emdad_22ndmay2022_input2)) when (emdad_22ndmay2022_op = "0011") else -- SUBU
32   std_logic_vector(shift_right(unsigned(emdad_22ndmay2022_input2), to_integer(unsigned(emdad_22ndmay2022_shift)))) when (emdad_22ndmay2022_op = "1000") else -- SRL
33   std_logic_vector(shift_left(unsigned(emdad_22ndmay2022_input2), to_integer(unsigned(emdad_22ndmay2022_shift)))) when (emdad_22ndmay2022_op = "0111") else -- SLL
34   std_logic_vector(shift_right(signed(emdad_22ndmay2022_input2), to_integer(unsigned(emdad_22ndmay2022_shift)))) when (emdad_22ndmay2022_op = "1001") else -- SRA
35   (others => '0');
36
37
38 process (emdad_22ndmay2022_op, emdad_22ndmay2022_input2)
39 begin
40   if (emdad_22ndmay2022_op = "1011") then
41     emdad_22ndmay2022_quotient <= std_logic_vector(unsigned(emdad_22ndmay2022_input1) / unsigned(emdad_22ndmay2022_input2));
42     emdad_22ndmay2022_remainder <= std_logic_vector(unsigned(emdad_22ndmay2022_input1) rem unsigned(emdad_22ndmay2022_input2));
43     emdad_22ndmay2022_HI_10_result <= emdad_22ndmay2022_remainder & emdad_22ndmay2022_quotient;
44   elsif(emdad_22ndmay2022_op = "1010") then
45     emdad_22ndmay2022_HI_10_Result <= std_logic_vector(unsigned(emdad_22ndmay2022_input1) * unsigned(emdad_22ndmay2022_input2));
46   end if;
47 end process;
48
49 emdad_22ndmay2022_zero <= '1' when emdad_22ndmay2022_result = x"00000000" else '0';
50 emdad_22ndmay2022_alu_result <= emdad_22ndmay2022_result;
51 end struct;

```

Figure 19: VHDL code the ALU

In figure 20, we can see that the code compiled successfully.

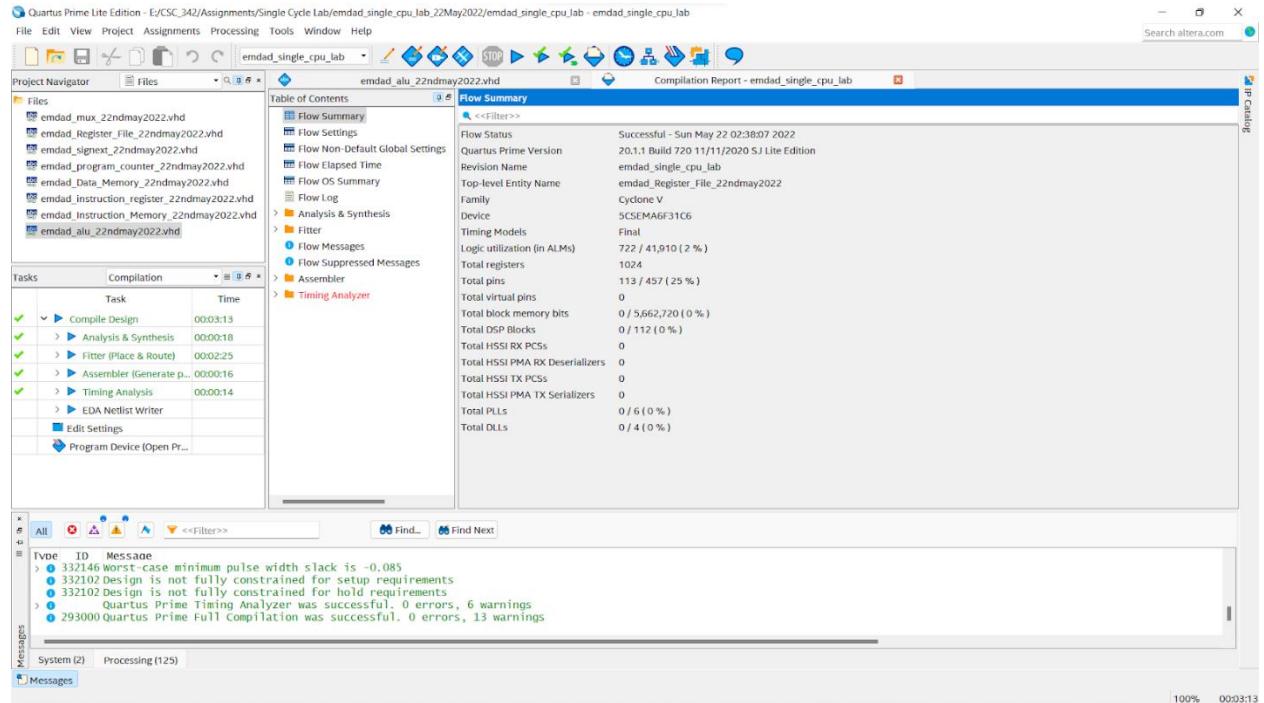


Figure 20: compiled successfully

Control Unit:

In figure 21, we can see the VHDL code the control unit. I have 2 inputs and 11 outputs.

```

1 Library IEEE;
2 use IEEE.STD.TEXT.all;
3 use IEEE.STD_LOGIC_UNSIGNED.all;
4 use IEEE.NUMERIC_STD.all;
5
6 ENTITY emdad_control_unit_22ndmay2022 IS
7 PORT
8 (
9   emdad_22ndmay2022_func, emdad_22ndmay2022_opcode: IN STD.LOGIC_VECTOR(5 DOWNTO 0);
10  emdad_22ndmay2022_ShiftAmount: IN STD.LOGIC_VECTOR(4 DOWNTO 0);
11  emdad_22ndmay2022_RegOut : OUT STD.LOGIC;
12  emdad_22ndmay2022_Regst : OUT STD.LOGIC;
13  emdad_22ndmay2022_Branch_EQ : OUT STD.LOGIC;
14  emdad_22ndmay2022_Branch_NE : OUT STD.LOGIC;
15  emdad_22ndmay2022_MemReg : OUT STD.LOGIC;
16  emdad_22ndmay2022_ALUOp : OUT STD.LOGIC;
17  emdad_22ndmay2022_ALUSign : OUT STD.LOGIC;
18  emdad_22ndmay2022_RegWrite : OUT STD.LOGIC;
19  emdad_22ndmay2022_MemWrite : OUT STD.LOGIC;
20  emdad_22ndmay2022_PCsrc : OUT STD.LOGIC;
21  emdad_22ndmay2022_ALUOp : OUT STD.LOGIC_VECTOR(3 DOWNTO 0);
22  emdad_22ndmay2022_Shift : OUT STD.LOGIC_VECTOR(4 DOWNTO 0)
23 );
24
25 END emdad_control_unit_22ndmay2022;

```

Figure 21: VHDL code the control unit (Part 1)

```

26 L
27  ARCHITECTURE struct_OF emdad_control_unit_22ndmay2022 IS
28
29 BEGIN
30   emdad_22ndmay2022_shift    <= emdad_22ndmay2022_shiftAmount WHEN (emdad_22ndmay2022_opcode = "000000") AND
31   (emdad_22ndmay2022_func = "000000" OR emdad_22ndmay2022_func = "000010" OR emdad_22ndmay2022_func = "000011")) ELSE "0000";
32   emdad_22ndmay2022_RegSt  <= '0' WHEN (emdad_22ndmay2022_opcode = "000000") ELSE '1';
33   emdad_22ndmay2022_RegUrc <= '0' WHEN (emdad_22ndmay2022_opcode = "000000" OR
34   (emdad_22ndmay2022_funcode = "000100") OR (emdad_22ndmay2022_funcode = "000110")) ELSE '1';
35   emdad_22ndmay2022_MemWrite <= '1' WHEN (emdad_22ndmay2022_opcode = "101011") ELSE '0';
36   emdad_22ndmay2022_RegWrite <= '0' WHEN (emdad_22ndmay2022_opcode = "101011") ELSE '1';
37   emdad_22ndmay2022_OpcReg <= '1' WHEN (emdad_22ndmay2022_opcode = "000101" OR emdad_22ndmay2022_opcode = "000010") ELSE '0';
38   emdad_22ndmay2022_jump   <= '1' WHEN (emdad_22ndmay2022_opcode = "000101") ELSE '0';
39   emdad_22ndmay2022_Branch_EQ <= '1' WHEN (emdad_22ndmay2022_opcode = "000101") ELSE '0';
40   emdad_22ndmay2022_Branch_NE <= '1' WHEN (emdad_22ndmay2022_opcode = "000101") ELSE '0';
41   emdad_22ndmay2022_PCSR <= '1' WHEN (emdad_22ndmay2022_opcode = "101011") ELSE '0';
42   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "101011") ELSE "addi--"
43   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "100000" AND emdad_22ndmay2022_opcode = "000000") ELSE ---add
44   emdad_22ndmay2022_ALUOp <= '000001' WHEN (emdad_22ndmay2022_opcode = "001001") ELSE ---addiu
45   emdad_22ndmay2022_ALUOp <= '000010' WHEN (emdad_22ndmay2022_opcode = "101011") ELSE ---sw
46   emdad_22ndmay2022_ALUOp <= '000011' WHEN (emdad_22ndmay2022_opcode = "010101") ELSE ---lw
47   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "100000" AND emdad_22ndmay2022_opcode = "100000") ELSE ---sub
48   emdad_22ndmay2022_ALUOp <= '000001' WHEN (emdad_22ndmay2022_opcode = "000100") ELSE ---and
49   emdad_22ndmay2022_ALUOp <= '000002' WHEN (emdad_22ndmay2022_opcode = "100002" AND emdad_22ndmay2022_opcode = "000000") ELSE ---addu
50   emdad_22ndmay2022_ALUOp <= '000010' WHEN (emdad_22ndmay2022_opcode = "000101") ELSE ---bne
51   emdad_22ndmay2022_ALUOp <= '000011' WHEN (emdad_22ndmay2022_opcode = "001001") ELSE ---bne
52   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "000000" AND emdad_22ndmay2022_opcode = "000000") ELSE ---subu
53   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "000100") ELSE ---andi
54   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "100100" AND emdad_22ndmay2022_opcode = "000000") ELSE ---and
55   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "100111" AND emdad_22ndmay2022_opcode = "000000") ELSE ---nor
56   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "010101") ELSE ---or
57   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "100101" AND emdad_22ndmay2022_opcode = "000000") ELSE ---or
58   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "000000" AND emdad_22ndmay2022_opcode = "000000") ELSE ---s1
59   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "000010" AND emdad_22ndmay2022_opcode = "000000") ELSE ---sr1
60   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "000001" AND emdad_22ndmay2022_opcode = "000000") ELSE ---sra
61   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "011000" AND emdad_22ndmay2022_opcode = "000000") ELSE ---MUL
62   emdad_22ndmay2022_ALUOp <= '000000' WHEN (emdad_22ndmay2022_opcode = "010110" AND emdad_22ndmay2022_opcode = "000000") ELSE ---DIV
63   (OTHERS >> '0');
64 END struct;

```

Figure 22: VHDL code the control unit (Part 2)

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

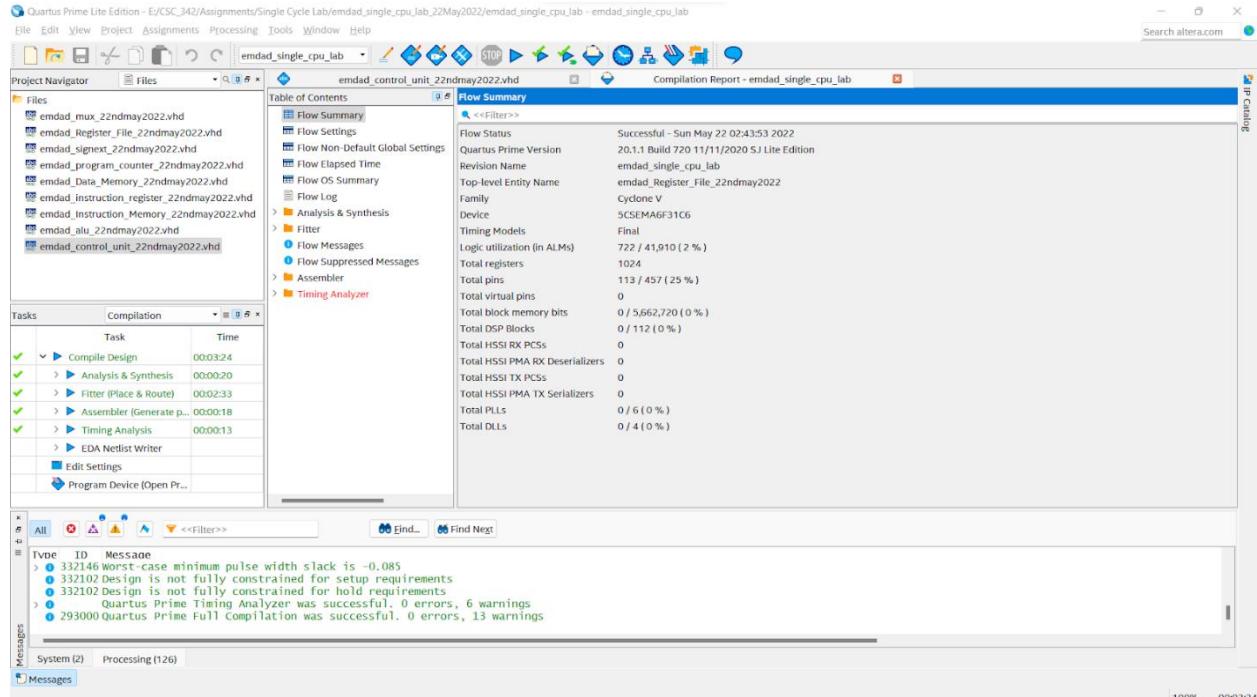
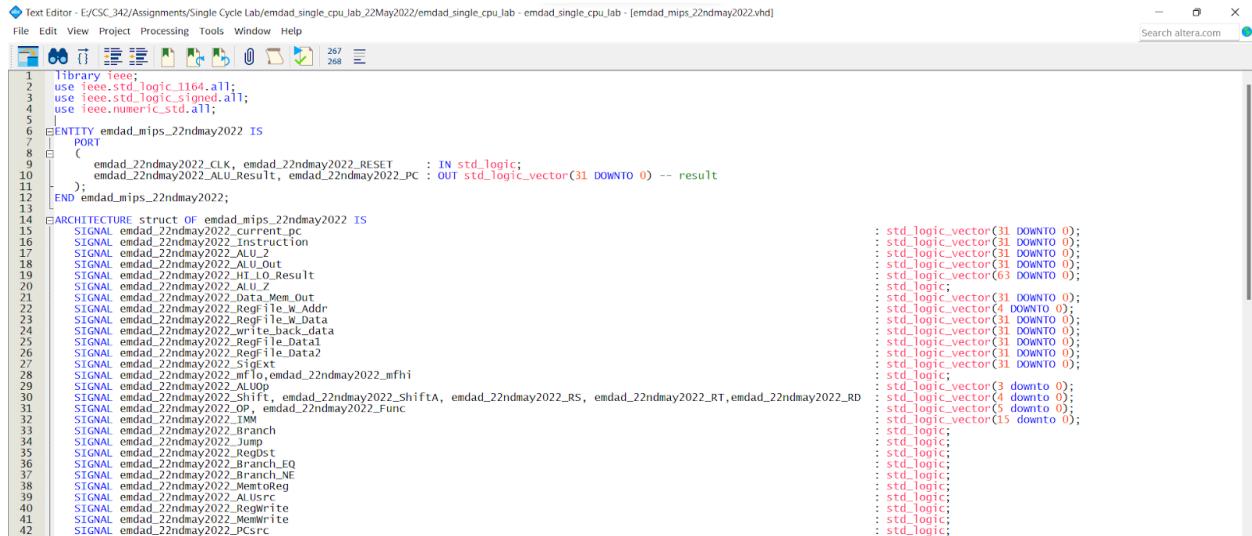


Figure 23: compiled successfully

MIPS:

In figure 24, we can see the VHDL code the CPU datapath. I have 1 input and 1 output.

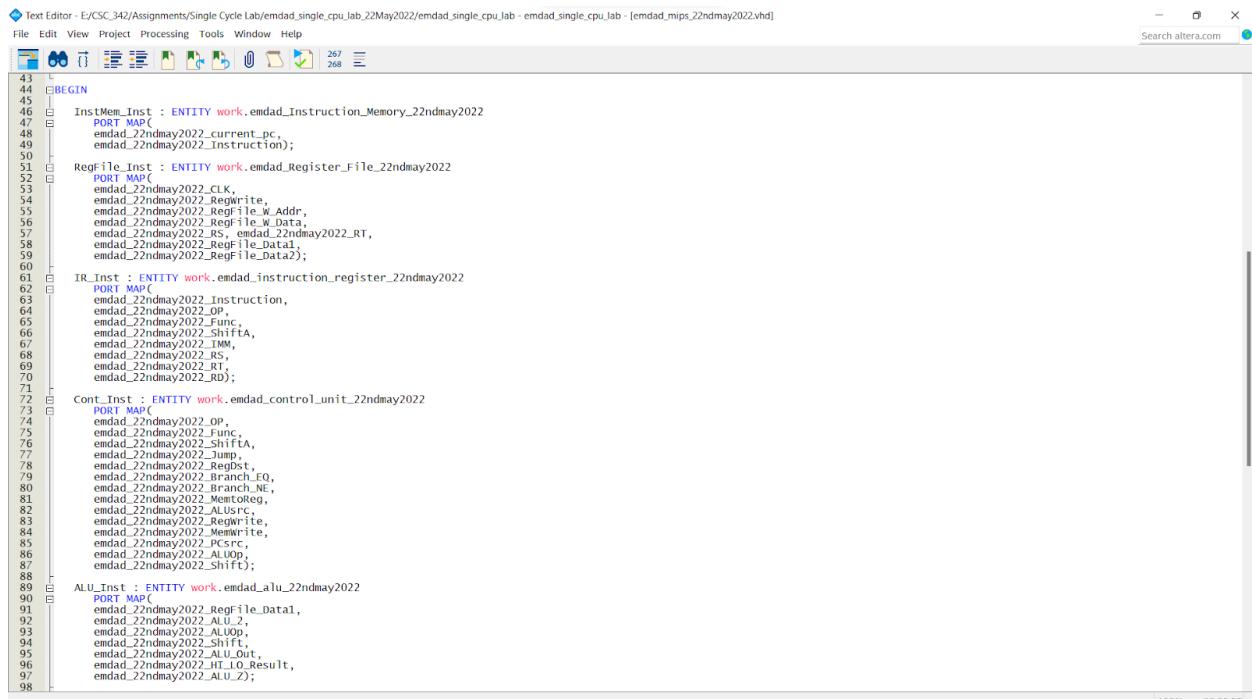


```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_signed.all;
4  use ieee.numeric_std.all;
5
6  ENTITY emdad_mips_22ndmay2022 IS
7    PORT
8      (
9        emdad_22ndmay2022_CLK, emdad_22ndmay2022_RESET : IN std_logic;
10       emdad_22ndmay2022_ALU_Result, emdad_22ndmay2022_PC : OUT std_logic_vector(31 DOWNTO 0) -- result
11      );
12  END emdad_mips_22ndmay2022;
13
14  ARCHITECTURE struct OF emdad_mips_22ndmay2022 IS
15    SIGNAL emdad_22ndmay2022_current_pc : std_logic_vector(31 DOWNTO 0);
16    SIGNAL emdad_22ndmay2022_Instruction : std_logic_vector(31 DOWNTO 0);
17    SIGNAL emdad_22ndmay2022_ALU_Out : std_logic_vector(31 DOWNTO 0);
18    SIGNAL emdad_22ndmay2022_HL_I_O_Result : std_logic_vector(31 DOWNTO 0);
19    SIGNAL emdad_22ndmay2022_ALU_I_O : std_logic_vector(31 DOWNTO 0);
20    SIGNAL emdad_22ndmay2022_Mem_Out : std_logic_vector(31 DOWNTO 0);
21    SIGNAL emdad_22ndmay2022_RegFile_W_Addr : std_logic_vector(4 DOWNTO 0);
22    SIGNAL emdad_22ndmay2022_RegFile_W_Data : std_logic_vector(31 DOWNTO 0);
23    SIGNAL emdad_22ndmay2022_Write_back_data : std_logic_vector(31 DOWNTO 0);
24    SIGNAL emdad_22ndmay2022_RegFile_Data1 : std_logic_vector(31 DOWNTO 0);
25    SIGNAL emdad_22ndmay2022_RegFile_Data2 : std_logic_vector(31 DOWNTO 0);
26    SIGNAL emdad_22ndmay2022_SignExt : std_logic;
27    SIGNAL emdad_22ndmay2022_mfTo : std_logic;
28    SIGNAL emdad_22ndmay2022_mfhI : std_logic;
29    SIGNAL emdad_22ndmay2022_ALUOp : std_logic;
30    SIGNAL emdad_22ndmay2022_ShiftA : std_logic;
31    SIGNAL emdad_22ndmay2022_RS : std_logic;
32    SIGNAL emdad_22ndmay2022_IMM : std_logic;
33    SIGNAL emdad_22ndmay2022_Branch : std_logic;
34    SIGNAL emdad_22ndmay2022_Jump : std_logic;
35    SIGNAL emdad_22ndmay2022_MemReq : std_logic;
36    SIGNAL emdad_22ndmay2022_Branch_EQ : std_logic;
37    SIGNAL emdad_22ndmay2022_Branch_NE : std_logic;
38    SIGNAL emdad_22ndmay2022_MemOrReg : std_logic;
39    SIGNAL emdad_22ndmay2022_ALUsrc : std_logic;
40    SIGNAL emdad_22ndmay2022_RegWrite : std_logic;
41    SIGNAL emdad_22ndmay2022_PcWrite : std_logic;
42    SIGNAL emdad_22ndmay2022_PcReset : std_logic;

```

Figure 24: VHDL code the CPU Datapath (part 1)



```

43  BEGIN
44
45    InstMem_Inst : ENTITY work.emdad_Instruction_Memory_22ndmay2022
46      PORT MAP
47        (
48          emdad_22ndmay2022_current_pc,
49          emdad_22ndmay2022_Instruction);
50
51    Regfile_Inst : ENTITY work.emdad_Register_File_22ndmay2022
52      PORT MAP(
53        emdad_22ndmay2022_CLK,
54        emdad_22ndmay2022_RegWrite,
55        emdad_22ndmay2022_RegFile_W_Addr,
56        emdad_22ndmay2022_RegFile_W_Data,
57        emdad_22ndmay2022_RS, emdad_22ndmay2022_RT,
58        emdad_22ndmay2022_RegFile_Data1,
59        emdad_22ndmay2022_RegFile_Data2);
60
61    IR_Inst : ENTITY work.emdad_instruction_register_22ndmay2022
62      PORT MAP(
63        emdad_22ndmay2022_Instruction,
64        emdad_22ndmay2022_OP,
65        emdad_22ndmay2022_ShiftC,
66        emdad_22ndmay2022_ShiftA,
67        emdad_22ndmay2022_IMM,
68        emdad_22ndmay2022_RS,
69        emdad_22ndmay2022_RT,
70        emdad_22ndmay2022_RD);
71
72    Cont_Inst : ENTITY work.emdad_control_unit_22ndmay2022
73      PORT MAP(
74        emdad_22ndmay2022_OP,
75        emdad_22ndmay2022_Func,
76        emdad_22ndmay2022_ShiftA,
77        emdad_22ndmay2022_Jump,
78        emdad_22ndmay2022_RegDst,
79        emdad_22ndmay2022_RegRt,
80        emdad_22ndmay2022_Branch_EQ,
81        emdad_22ndmay2022_Branch_NE,
82        emdad_22ndmay2022_MemOrReg,
83        emdad_22ndmay2022_RegWrite,
84        emdad_22ndmay2022_RegWrite,
85        emdad_22ndmay2022_PCSrc,
86        emdad_22ndmay2022_ALUOp,
87        emdad_22ndmay2022_Shift);
88
89    ALU_Inst : ENTITY work.emdad_alu_22ndmay2022
90      PORT MAP(
91        emdad_22ndmay2022_Regfile_Data1,
92        emdad_22ndmay2022_ALU_1,
93        emdad_22ndmay2022_ALU_2,
94        emdad_22ndmay2022_Shift,
95        emdad_22ndmay2022_ALU_Out,
96        emdad_22ndmay2022_HL_I_O_Result,
97        emdad_22ndmay2022_ALU_Z);
98

```

Figure 25: VHDL code the CPU Datapath (part 2)

```

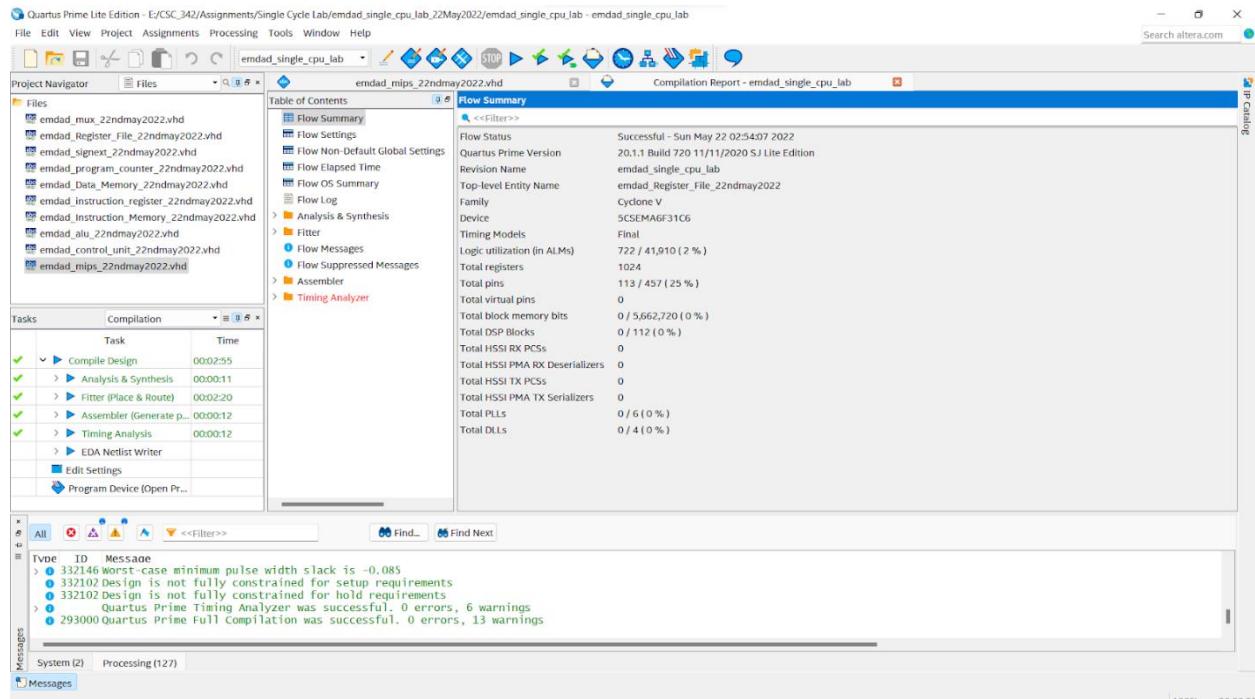
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152

```

100% 00:02:55

Figure 26: VHDL code the CPU Datapath (part 3)

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



Block Diagrams:

In figure 28, we can see the symbol diagram generated from the VHDL code of mux file.

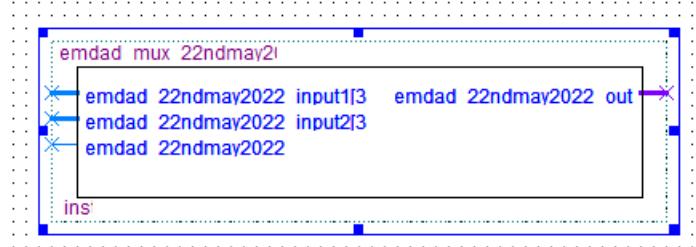


Figure 28: symbol diagram generated from the VHDL code of mux file

In figure 29, we can see the symbol diagram generated from the VHDL code of register file.

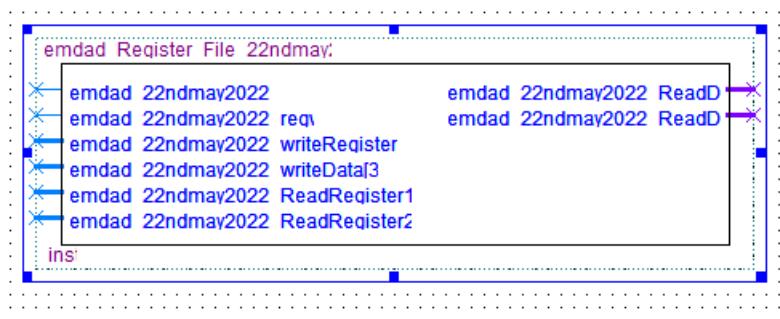


Figure 29: symbol diagram generated from the VHDL code of register file.

In figure 30, we can see the symbol diagram generated from the VHDL code of sign extender file.

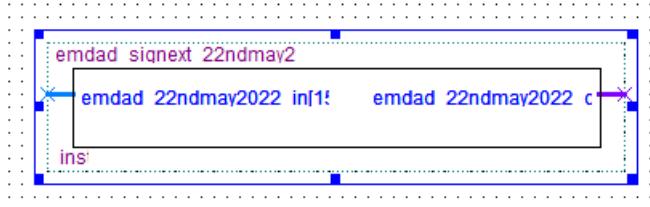


Figure 30: symbol diagram generated from VHDL code of sign extender file

In figure 31, we can see the symbol diagram generated from the VHDL code of program counter file.

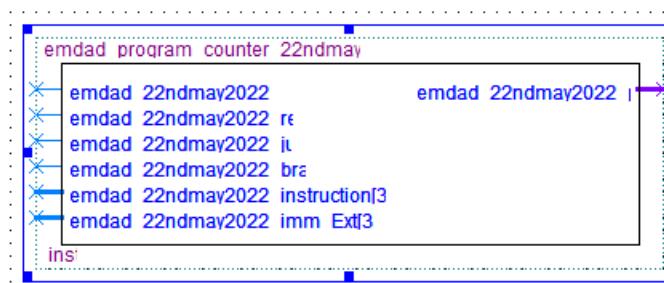


Figure 31: symbol diagram generated from VHDL code of program counter file

In figure 32, we can see the symbol diagram generated from the VHDL code of data memory file.

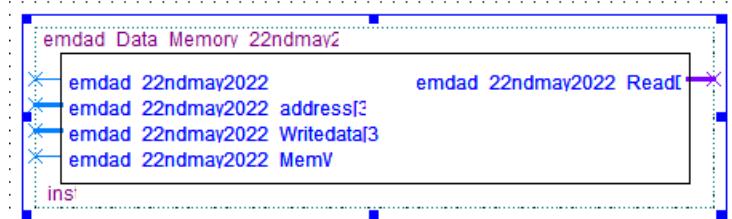


Figure 32: symbol diagram generated from VHDL code of data memory file

In figure 33, we can see the symbol diagram generated from the VHDL code of instruction register file.

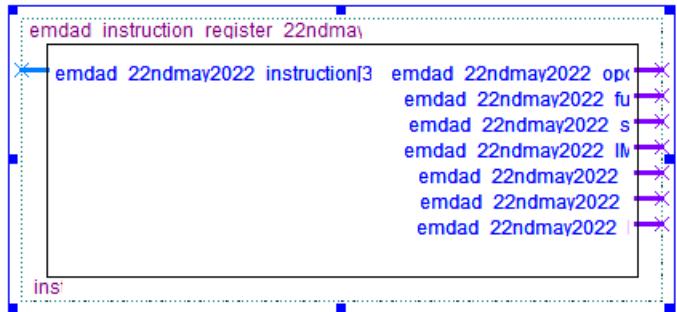


Figure 33: symbol diagram generated from VHDL code of instruction register file

In figure 34, we can see the symbol diagram generated from the VHDL code of instruction memory file.

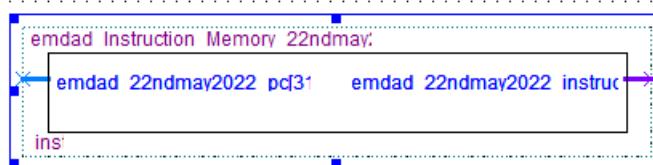


Figure 34: symbol diagram generated from VHDL code of instruction memory file

In figure 35, we can see the symbol diagram generated from the VHDL code of arithmetic logic unit (ALU) file.

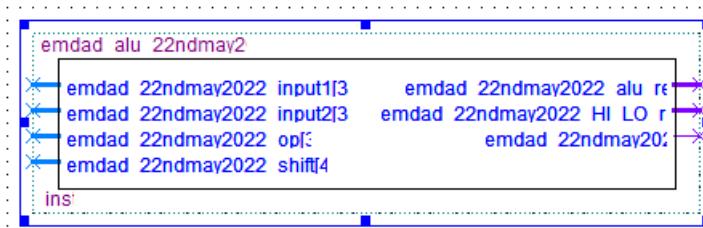


Figure 35: symbol diagram generated from VHDL code of alu file

In figure 36, we can see the symbol diagram generated from the VHDL code of control unit file.

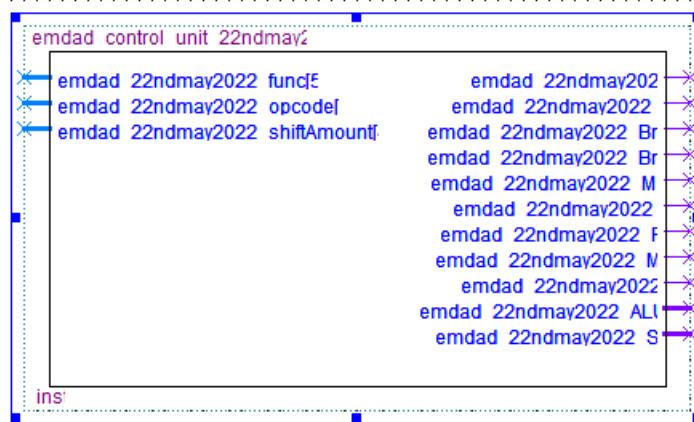


Figure 36: symbol diagram generated from VHDL code of control unit file

In figure 36, we can see the symbol diagram generated from the VHDL code of mips file.



Figure 37: symbol diagram generated from VHDL code of mips file

MIPS:

In figure 38, we can see the MIPS code which add integers and return the sum of it.

The screenshot shows the MARS 4.5 assembly editor interface. The title bar reads "E:\CSC_342\Assignments\Single Cycle Lab\emdad_CPU_test_program.asm - MARS 4.5". The menu bar includes File, Edit, Run, Settings, Tools, and Help. The toolbar contains various icons for file operations and simulation controls. The main window has tabs for "Edit" and "Execute". The code area displays the following MIPS assembly code:

```

1 .text
2
3 addi    $s0, $zero, 1
4 addi    $s1, $zero, 5
5 addu   $s2, $s1, $s0
6 subu   $s3, $s0, $s1
7 beq    $s0, $s3, Else
8 mult   $s1, $s0
9 j L1
10
11 Else:
12 div    $s1, $s0
13
14 L1:
15 addu   $s0, $s2, $s0

```

Figure 38: MIPS code

In figure 39, we can see it assembles successfully.

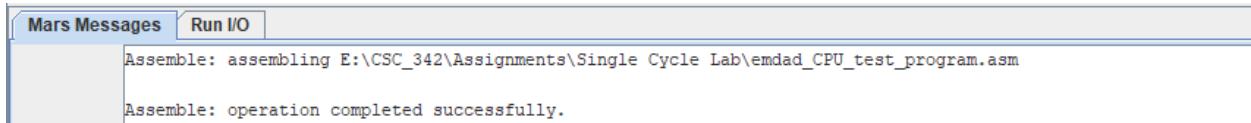


Figure 39: assembles successfully

Simulation:

In figure 40, we can see MIPS code executed, text segment, data segment, registers.

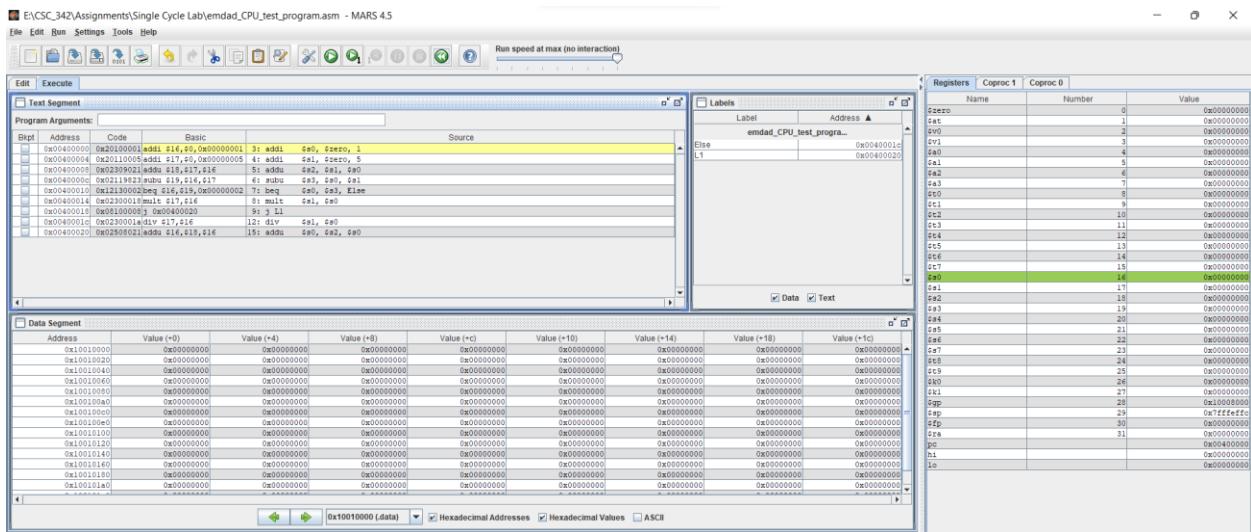


Figure 40: text segment, data segment, registers

We can see `$s0` is 1 and `$s1` is 5 and then the program does the sum. The below code did the sum

0x02119823 subu \$19,\$16,\$17 6: subu \$s3, \$s0, \$s1

After that, we can see \$s2 is 6 and it jumped it to **lo**.

\$s0		16	0x00000001
\$s1		17	0x00000005
\$s2		18	0x00000006
\$s3		19	0xfffffff0
\$s4		20	0x00000000
\$s5		21	0x00000000
\$s6		22	0x00000000
\$s7		23	0x00000000
\$t8		24	0x00000000
\$t9		25	0x00000000
\$k0		26	0x00000000
\$k1		27	0x00000000
\$gp		28	0x10008000
\$sp		29	0x7ffffefffc
\$fp		30	0x00000000
\$ra		31	0x00000000
pc			0x00400018
hi			0x00000000
lo			0x00000005

In figure 41, we can see the directory for the CPU project for ModelSim. All the files compiled without any errors.

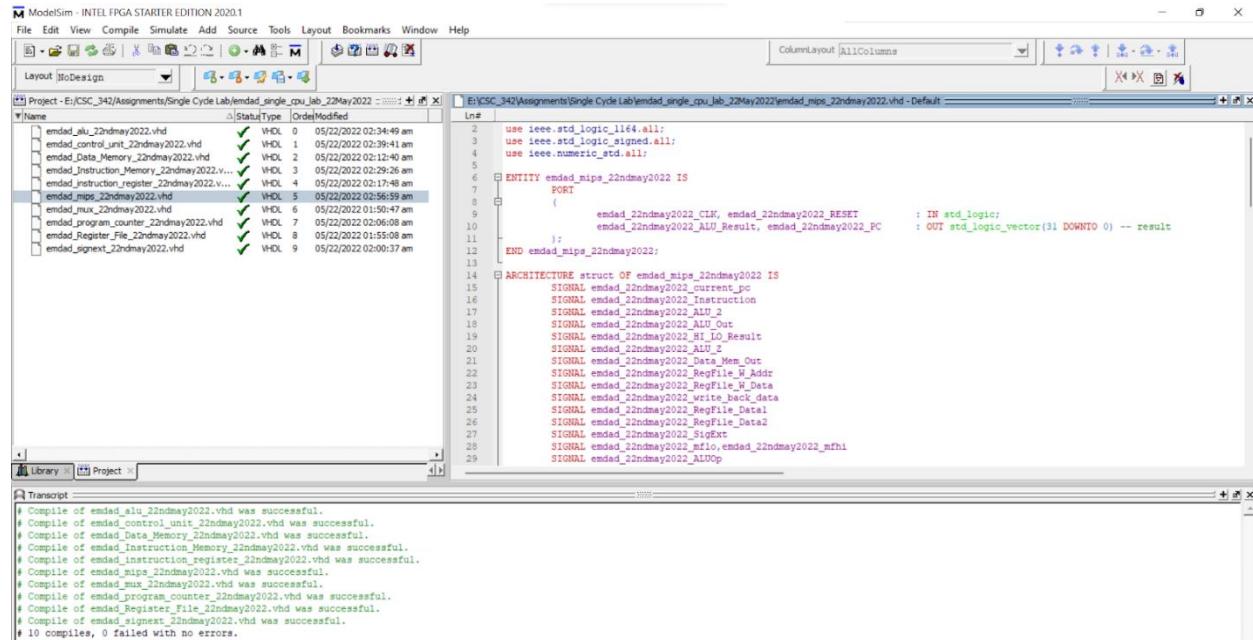


Figure 41: directory for the CPU project for ModelSim

In figure 42, we can see the simulation for the addition operation. I used the addition operation where $R[rd] = R[rs] + R[rt]$.

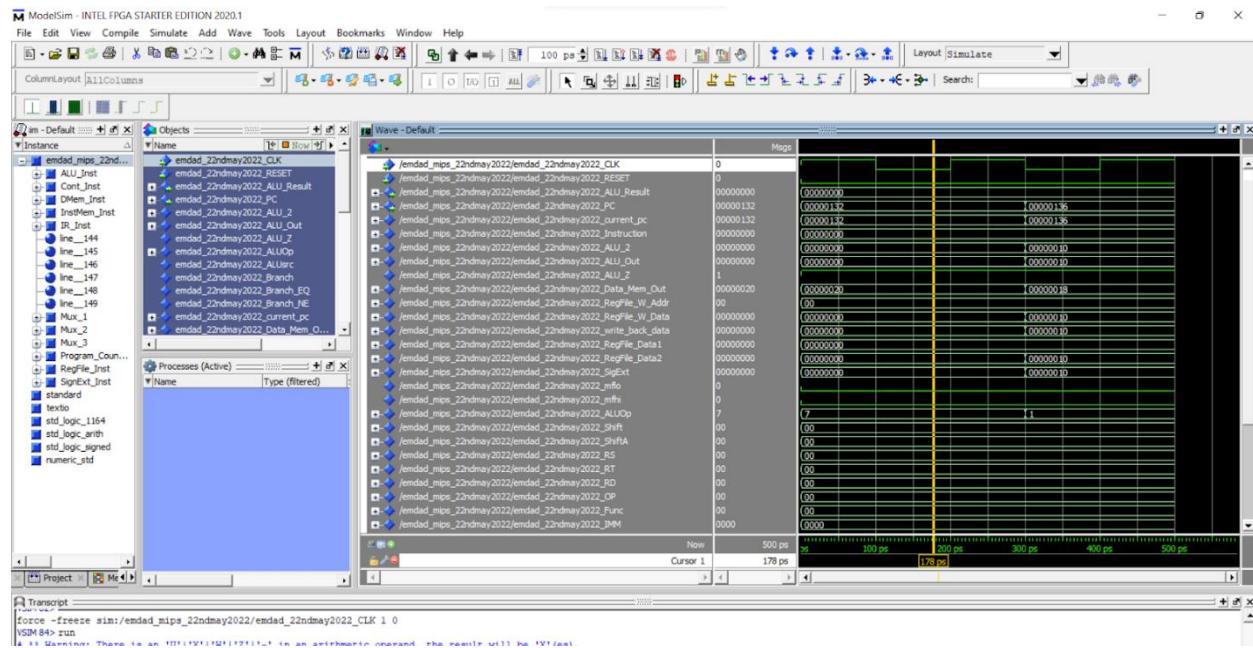


Figure 41: add operation

In figure 42, we can see the simulation for subtraction of MIPS instructions followed the equation, $R[rd] = R[rs] - R[rt]$. 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.

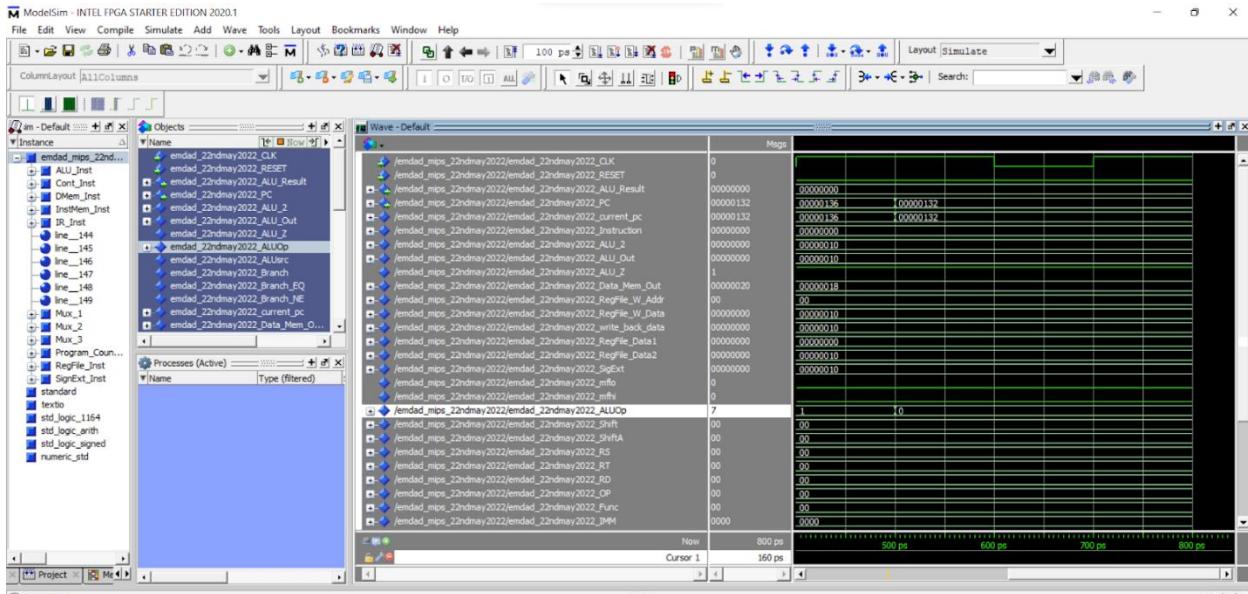


Figure 42: subtraction operation

In figure 43, we can see the simulation for multiplication of MIPS instructions.

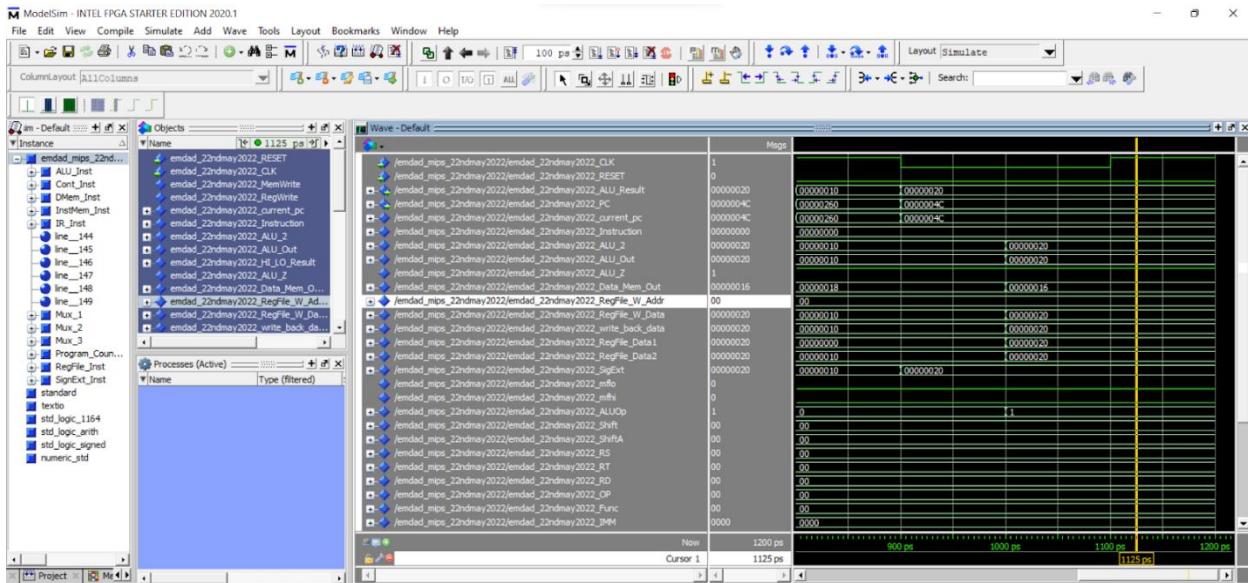


Figure 43: multiplication operation

Conclusion:

In this assignment, I learned how to learn and develop a single cycle CPU using Quartus and Modelsim. It was a good practice on Modelsim for circuit simulation. The experiment taught me how to design and implement in VHDL CPU controller that generates control signals to determine the data path for each instruction. The operations logics were helpful to learn easily. I relearned and reviewed the concept again and which will help me in the course further.