FINAL Lab Project: Single Cycle CPU-LITE

MdShahid Bin Emdad

May 22, 2022

CSC 34200/34300

Table of Contents

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.

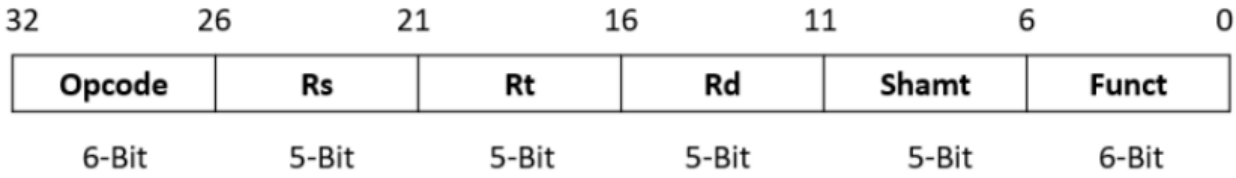
Instruction Memory: I used 2 cins, (cin1 and cin2) which are the first and second operand of the adder for the 16 bits. I also had op which is the bit that carried in from the less significant stage. For the outputs, I had neg which is the carry output that carried in from addition, sum (sum of the two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

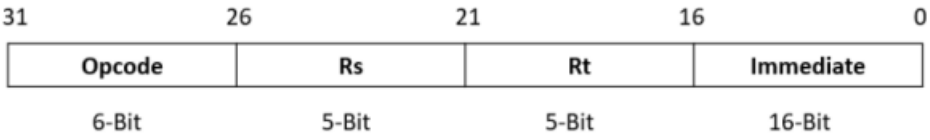
R-type Instructions:



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 Rt 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.

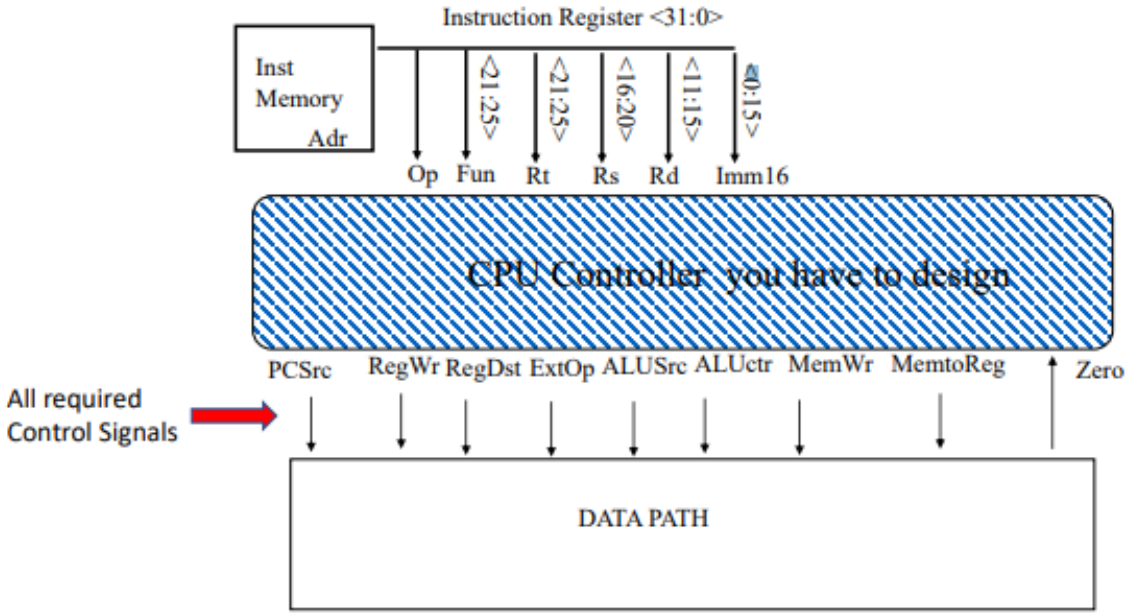


Figure 1: Visualization of CPU Controller Operation

Single Cycle CPU lab:

Signal	Value = 0	Value = 1
Branch	PC < = PC +4	PC < = PC +4 + {SignExt(Imm), <<2}
ALUsrc	Operand 2 <= BusB	Operand 2 <= 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.

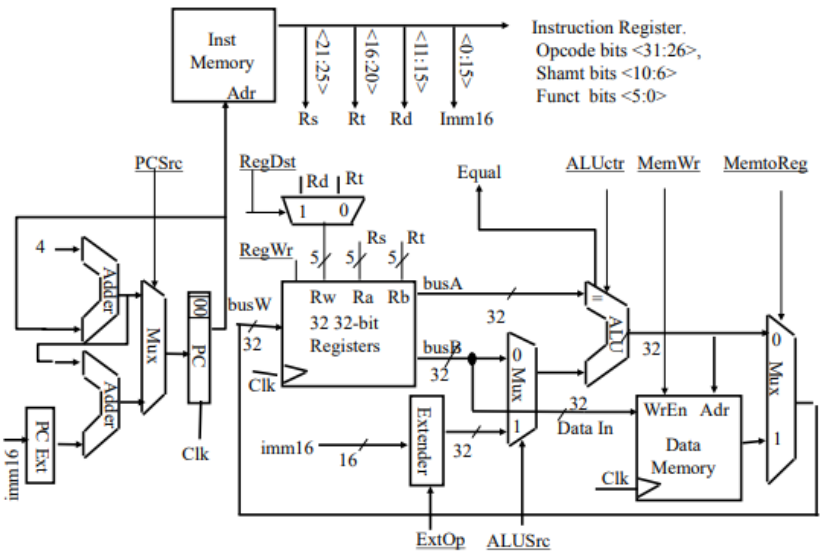


Figure 2: A Single Cycle CPU

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

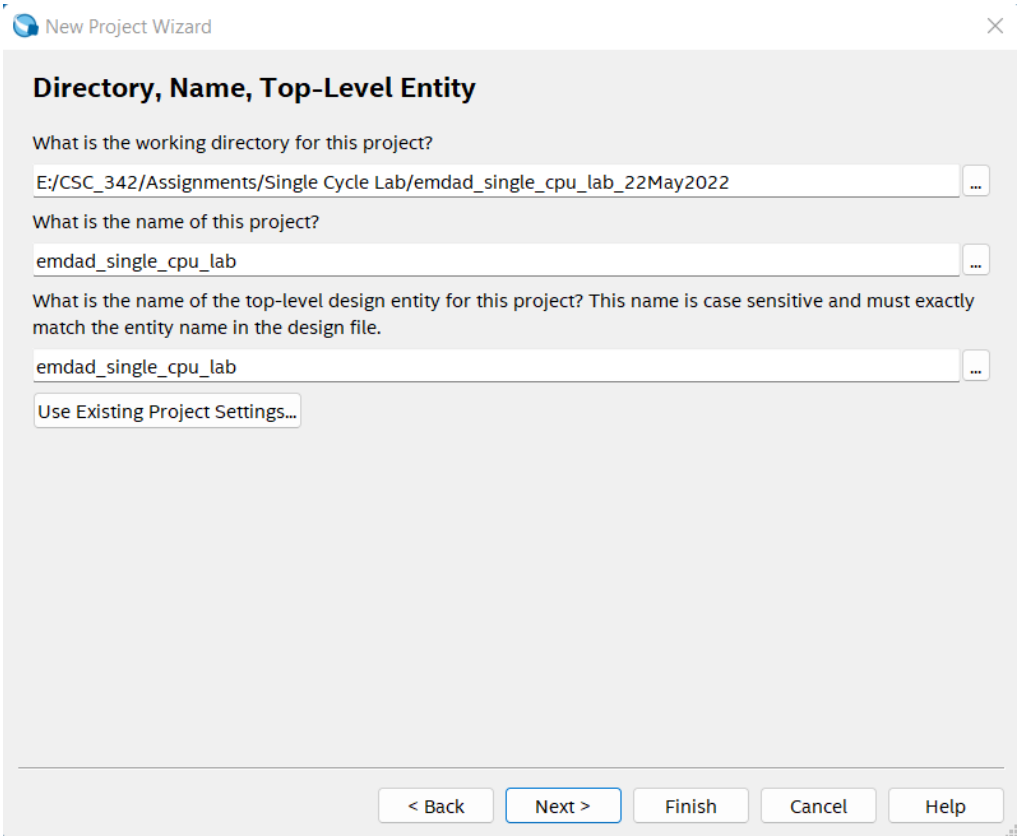


Figure 3: project directory for the CPU lab.

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

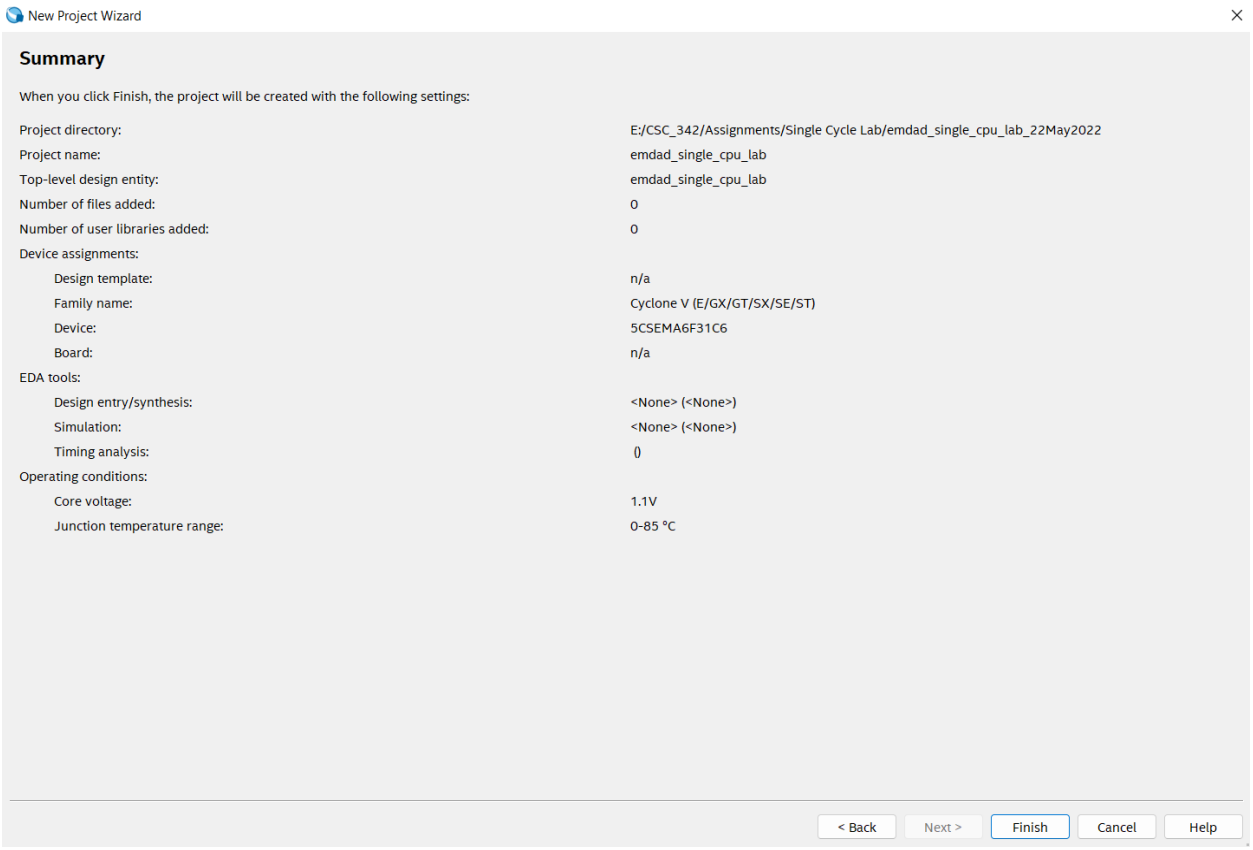


Figure 4: project summary for the CPU lab project.

Multiplexer:

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

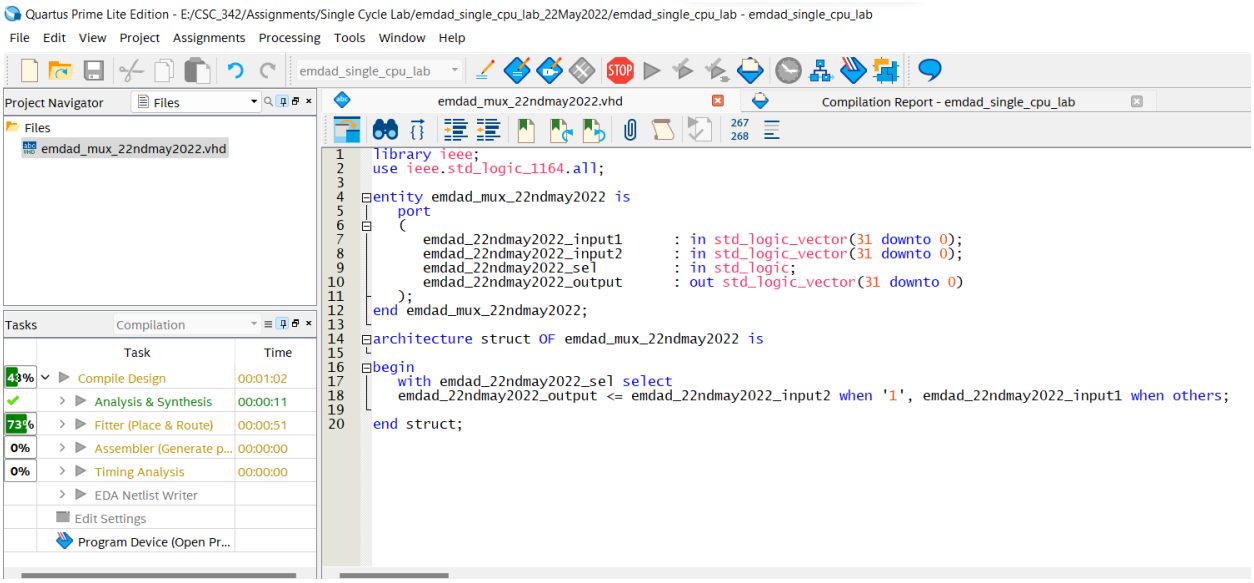


Figure 5: VHDL code for multiplexer

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

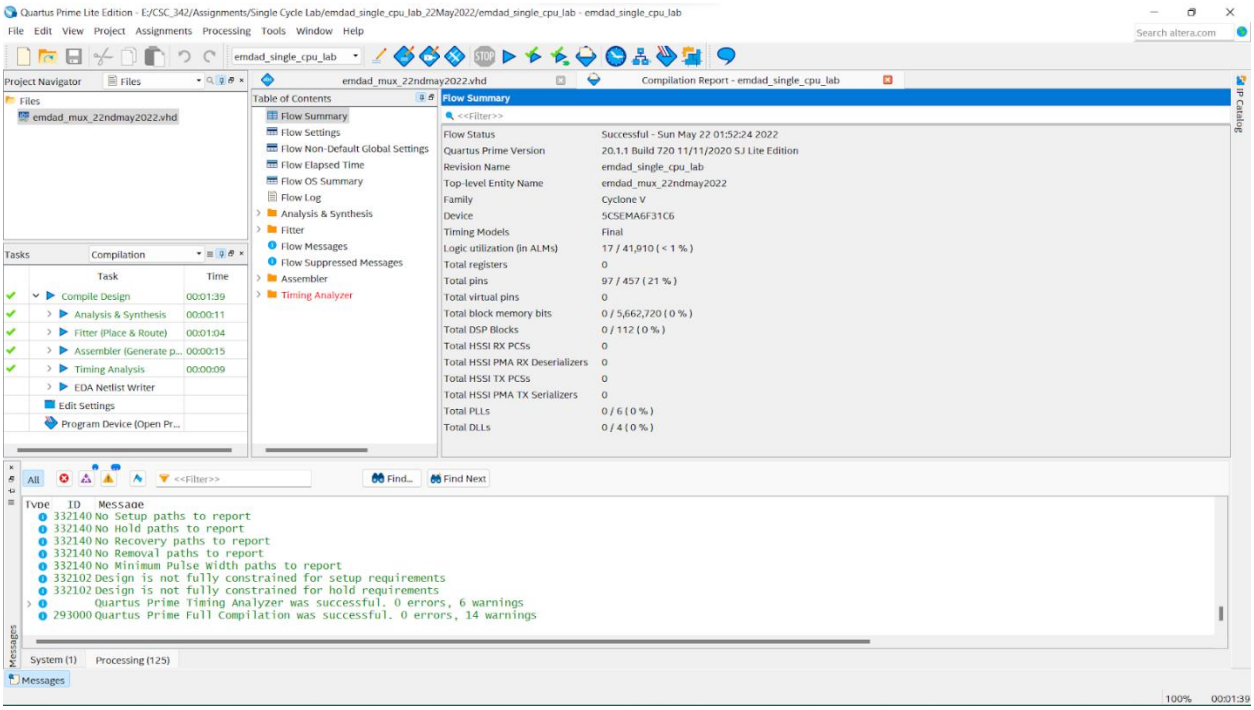


Figure 6: compiled successfully

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.

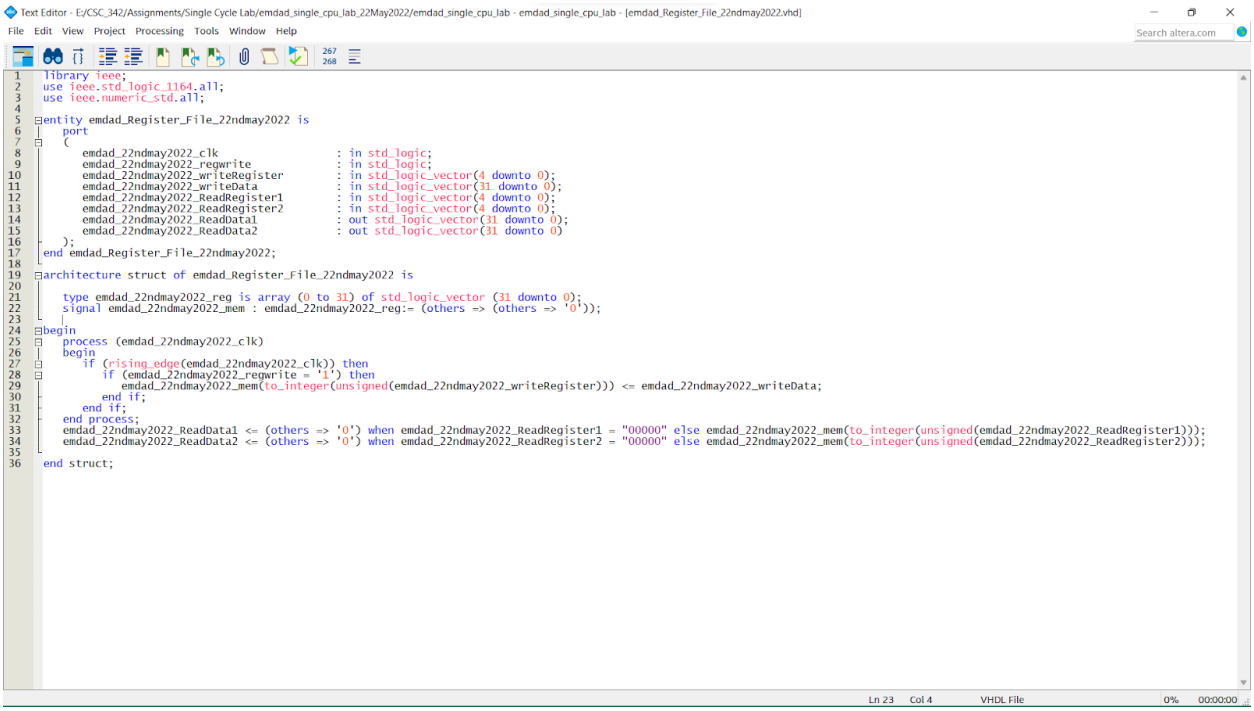


Figure 7: VHDL code the register file

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

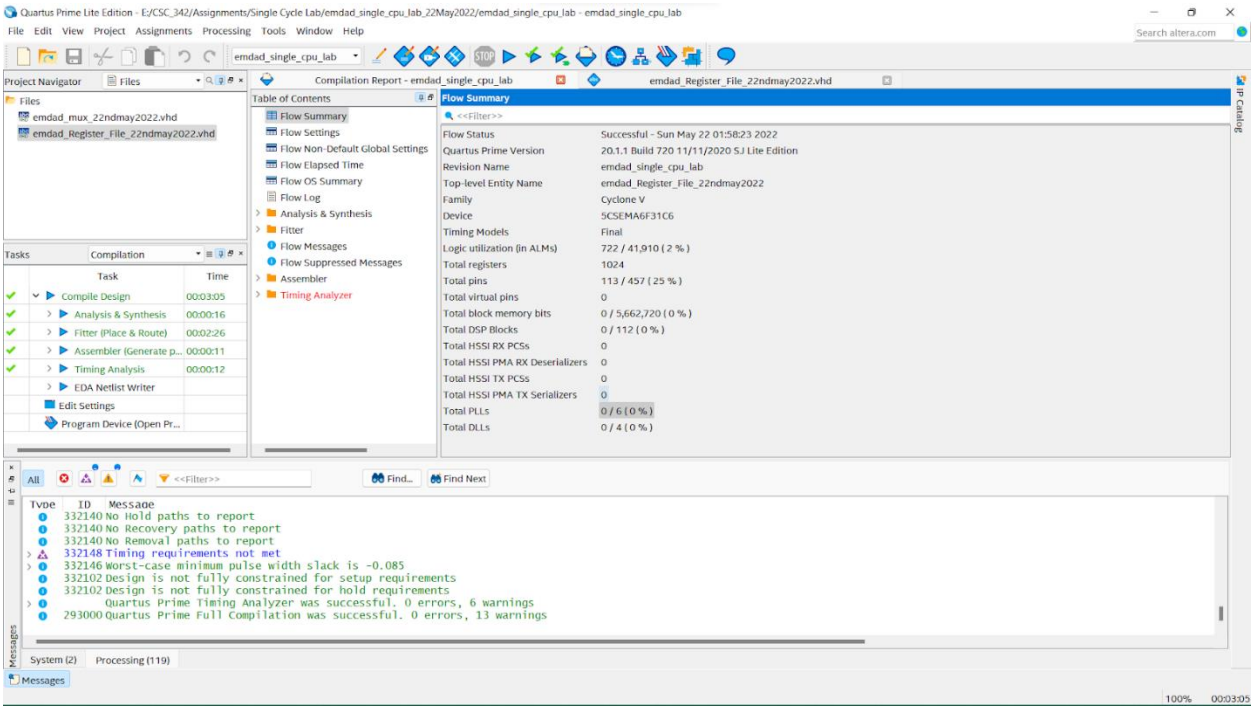


Figure 8: compiled successfully.

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.

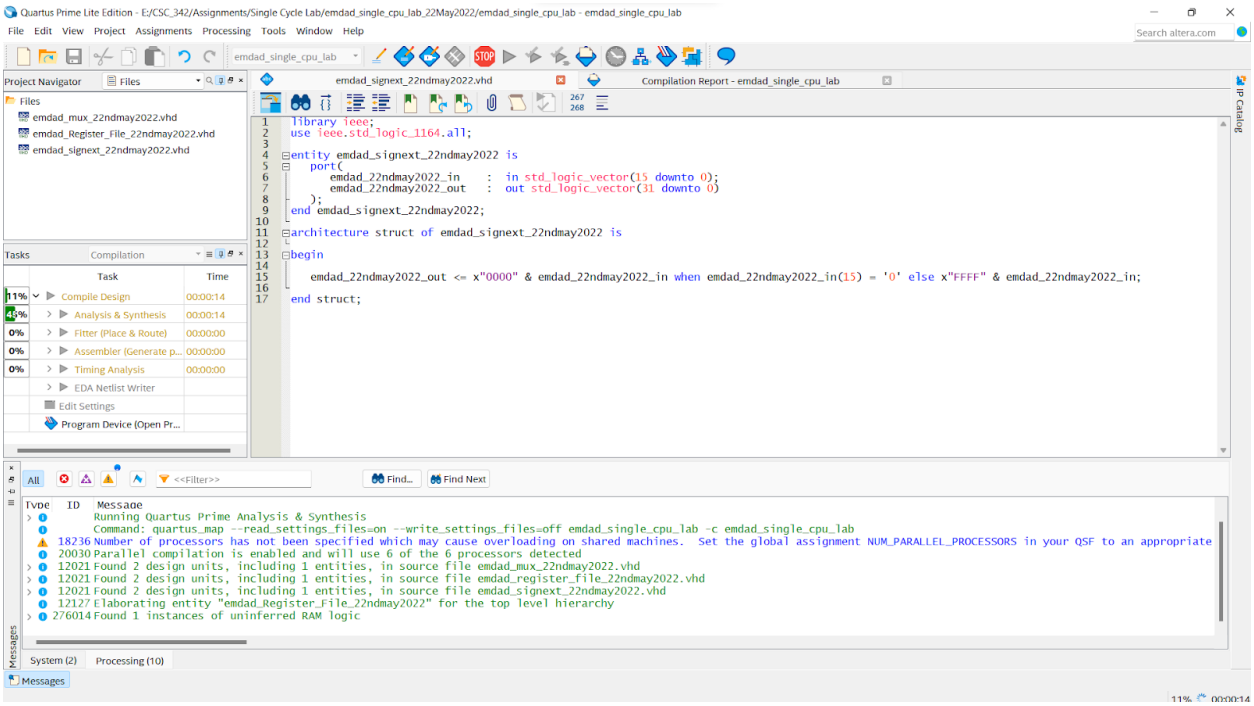


Figure 9: VHDL code the sign extender

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

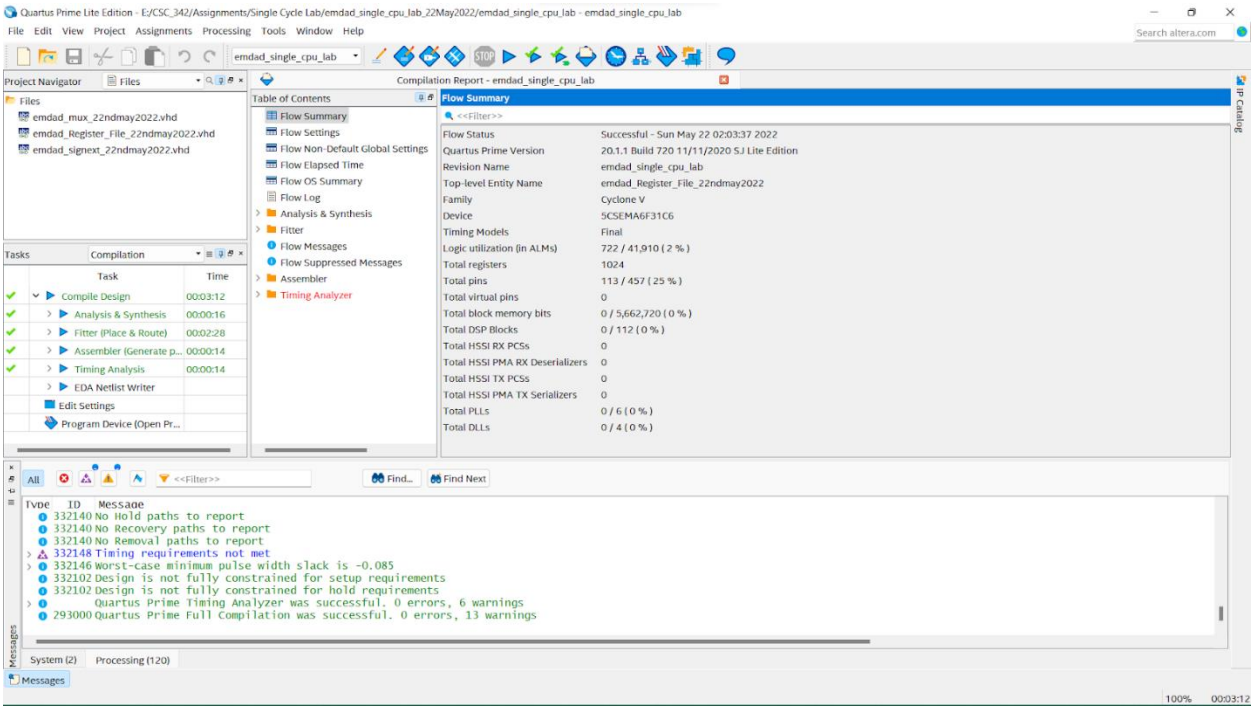


Figure 10: compiled successfully.

Program Counter:

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

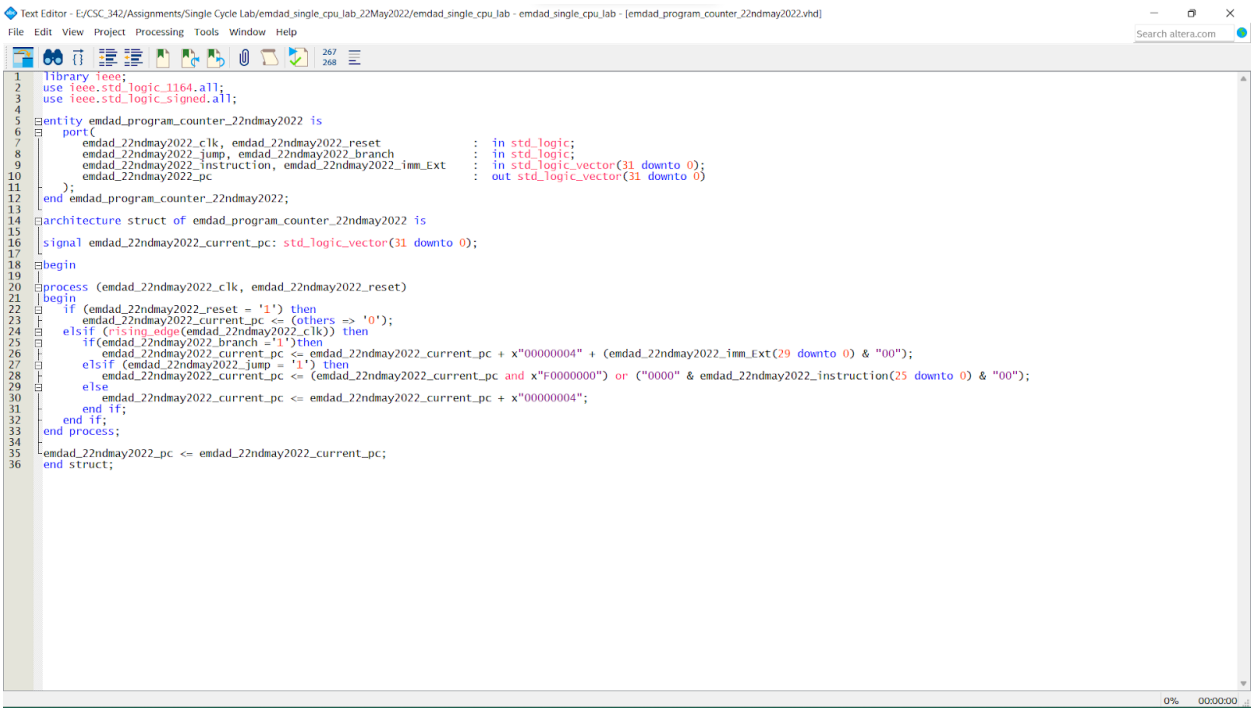


Figure 11: VHDL code the program counter

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

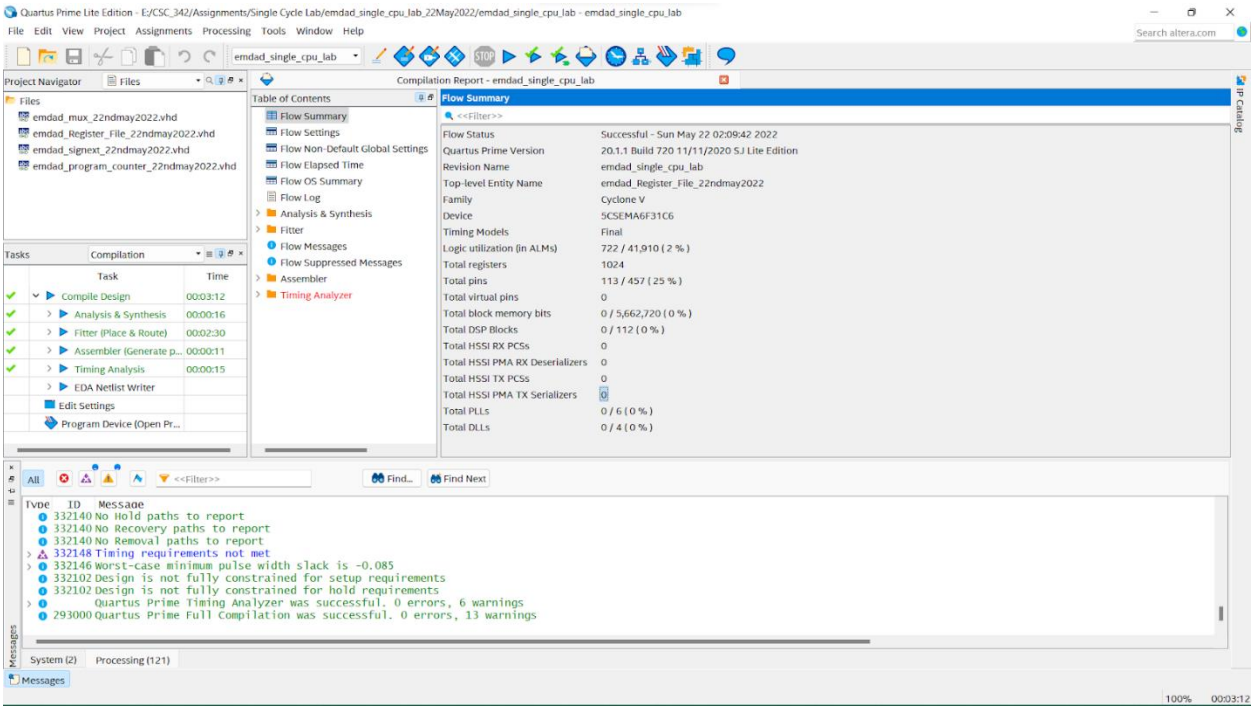


Figure 12: compiled successfully

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.

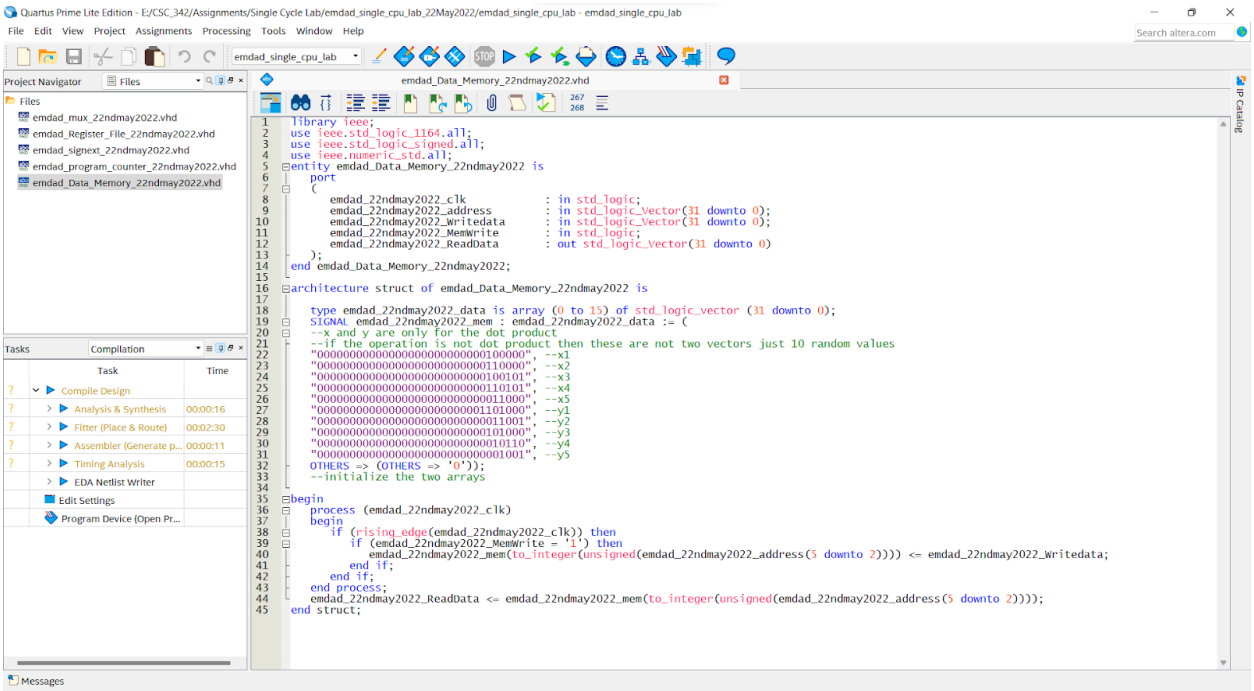


Figure 13: VHDL code the data memory

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

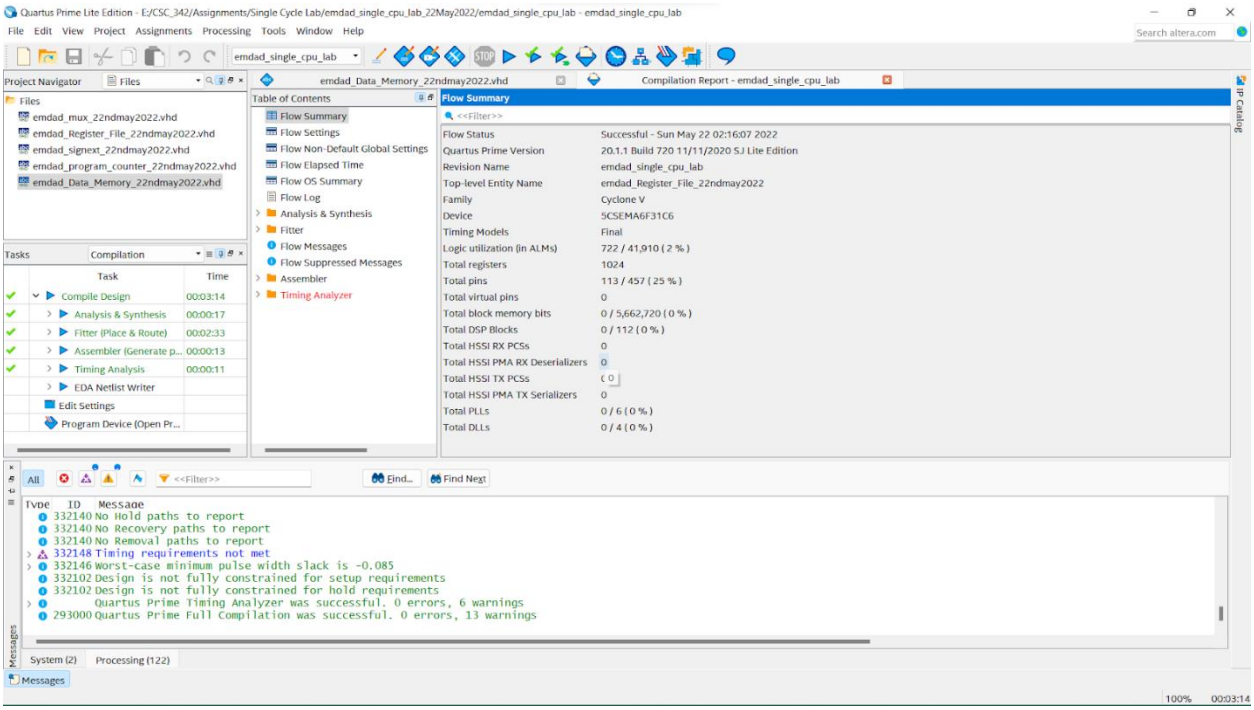


Figure 14: compiled successfully

Instruction Register:

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

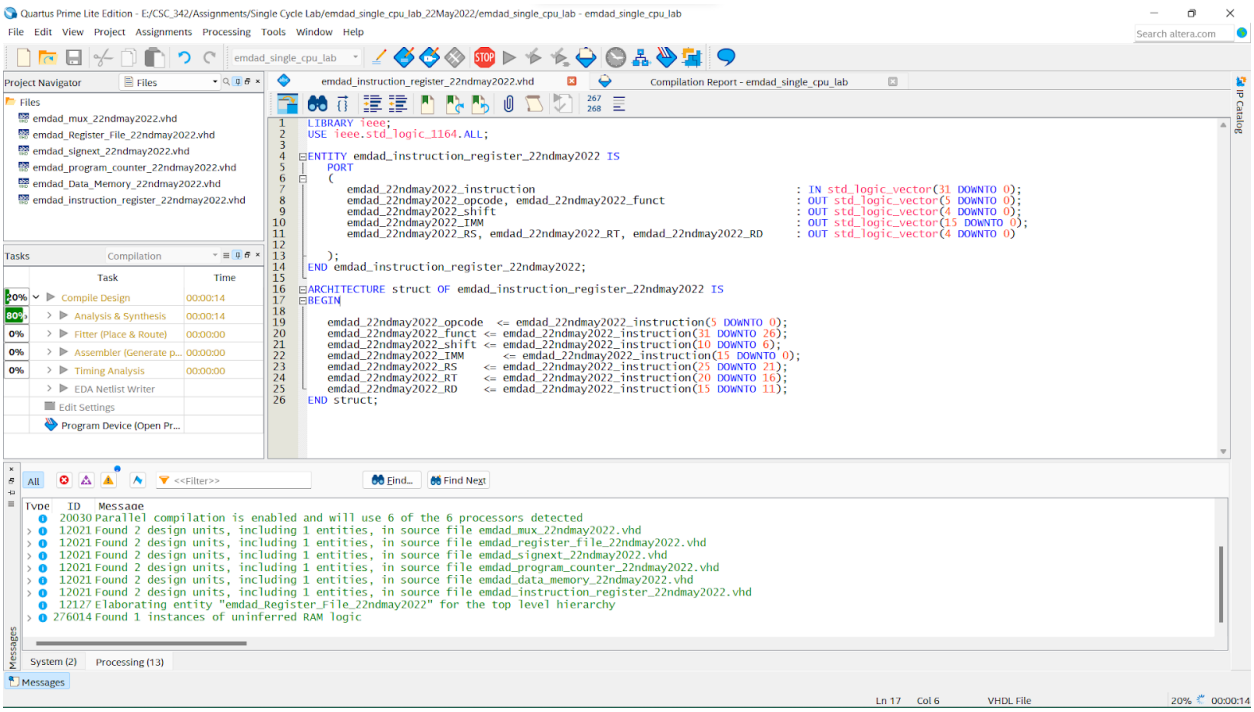


Figure 15: VHDL code the instruction Register

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

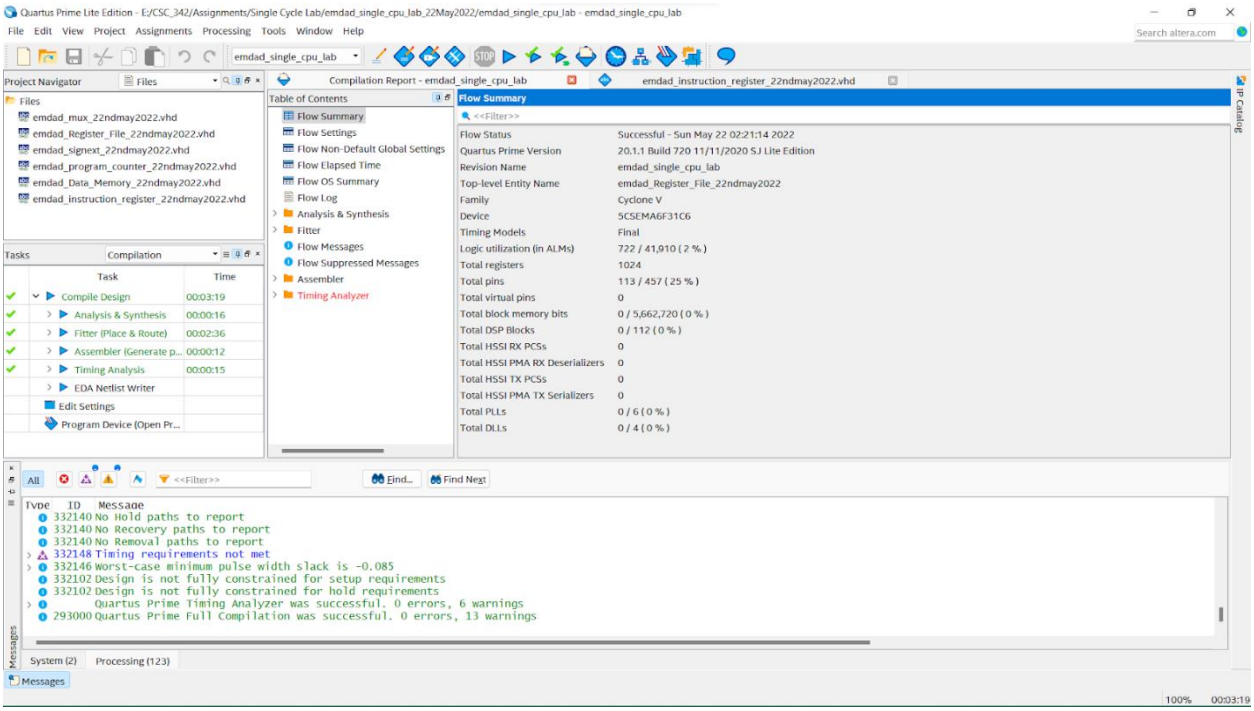


Figure 16: compiled successfully

Instruction Memory:

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

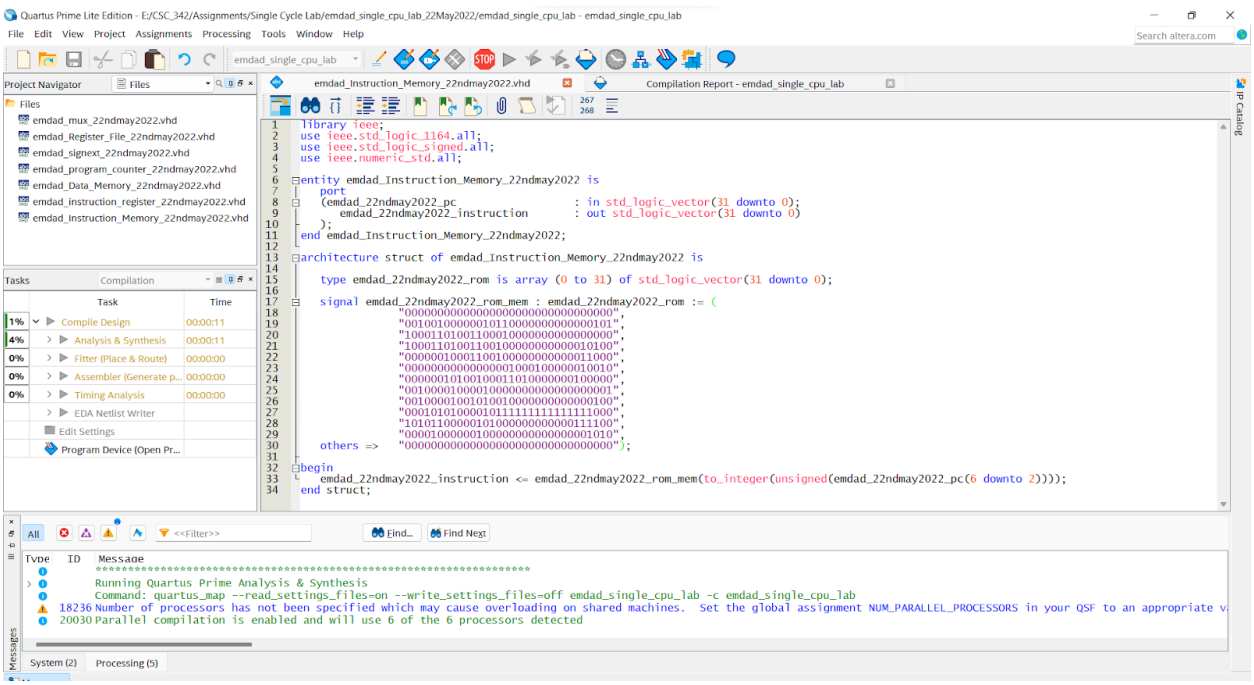


Figure 17: VHDL code the instruction memory

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

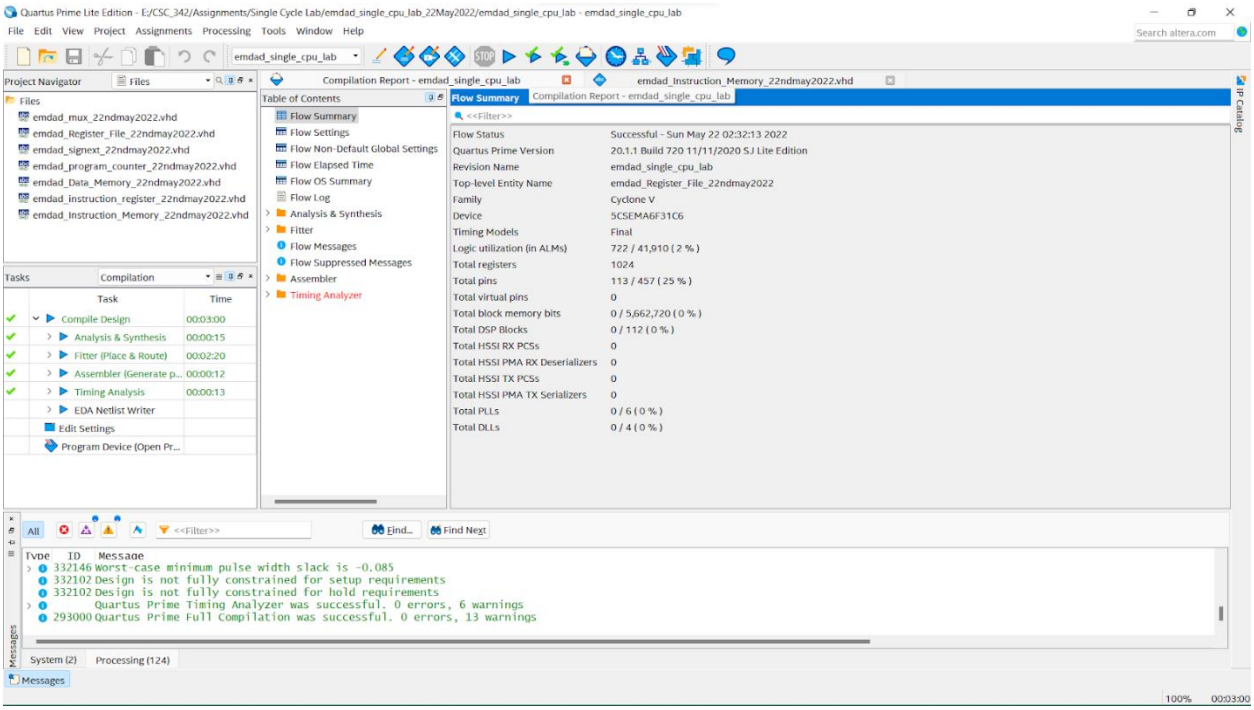


Figure 18: compiled successfully.

Arithmetic Logic Unit (ALU):

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

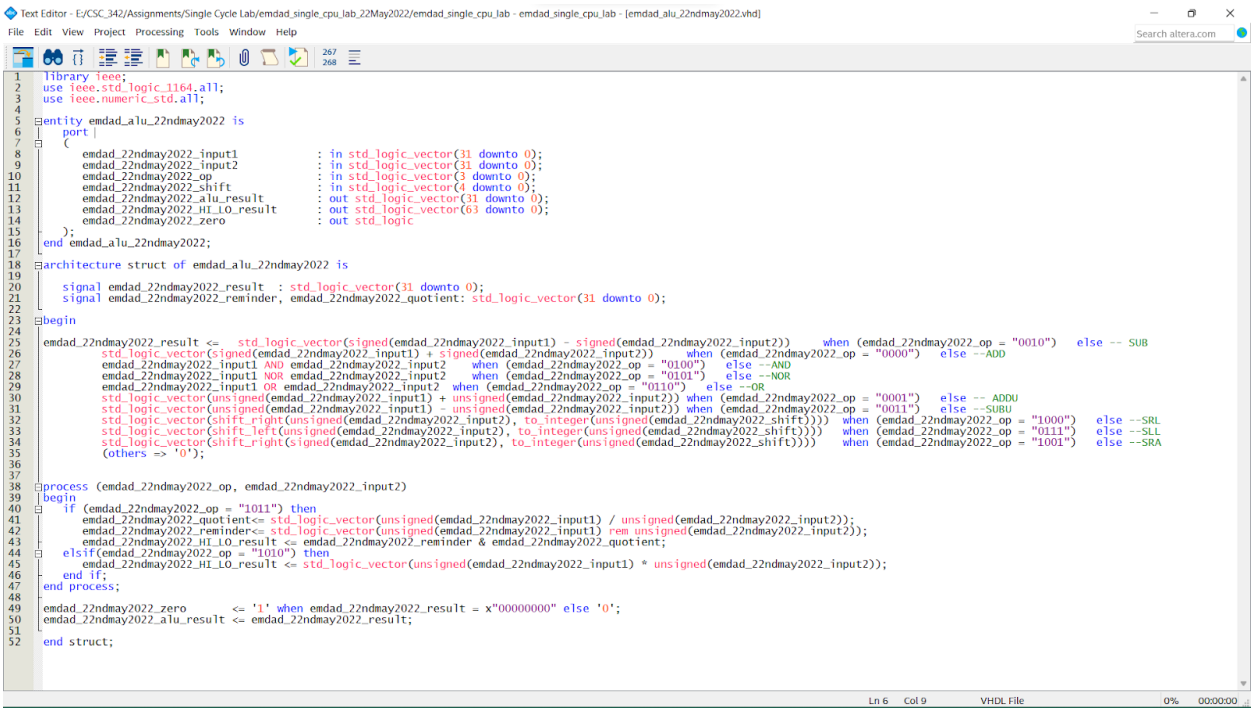


Figure 19: VHDL code the ALU

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

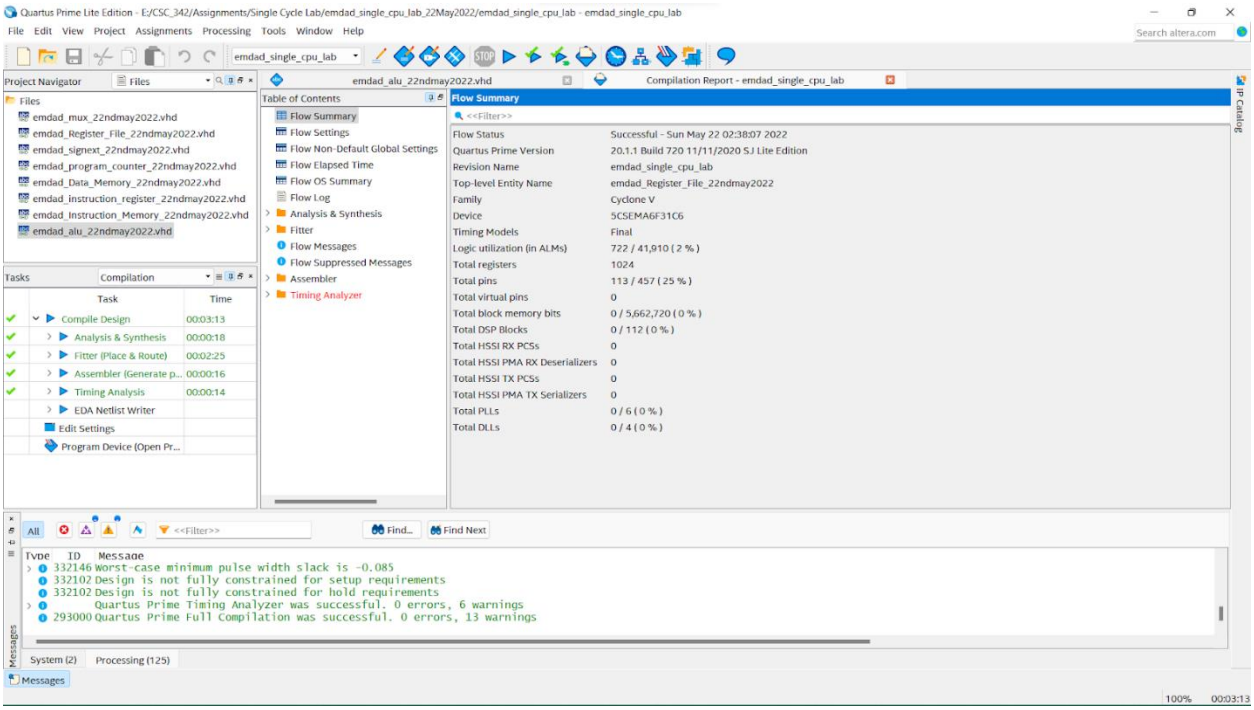
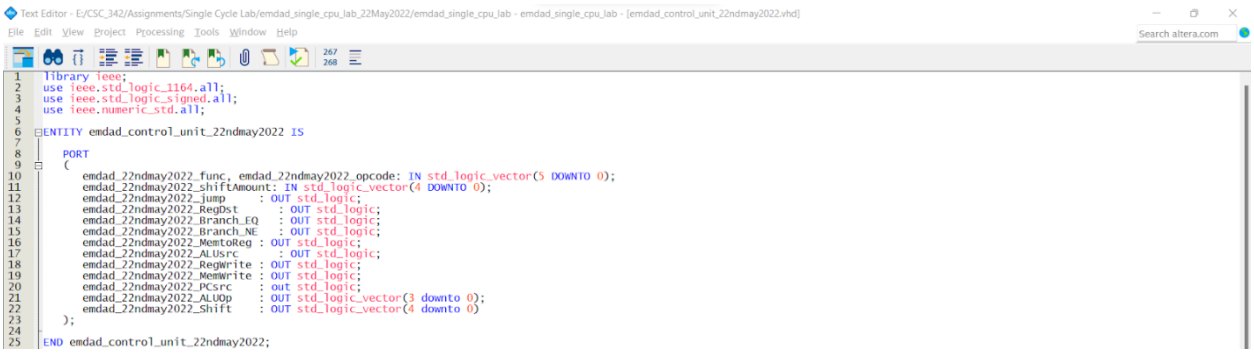


Figure 20: compiled successfully

Control Unit:

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



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

ENTITY emdad_control_unit_22ndmay2022 IS

    PORT
    (
        emdad_22ndmay2022_func, emdad_22ndmay2022_opcode: IN std_logic_vector(5 DOWNTO 0);
        emdad_22ndmay2022_shiftAmount: IN std_logic_vector(4 DOWNTO 0);
        emdad_22ndmay2022_jump     : OUT std_logic;
        emdad_22ndmay2022_RegDst      : OUT std_logic;
        emdad_22ndmay2022_Branch_EQ   : OUT std_logic;
        emdad_22ndmay2022_Branch_NE   : OUT std_logic;
        emdad_22ndmay2022_MemtoReg : OUT std_logic;
        emdad_22ndmay2022_ALUSrc      : OUT std_logic;
        emdad_22ndmay2022_RegWrite : OUT std_logic;
        emdad_22ndmay2022_MemWrite : OUT std_logic;
        emdad_22ndmay2022_PCSrc    : out std_logic;
        emdad_22ndmay2022_ALUOp    : OUT std_logic_vector(3 downto 0);
        emdad_22ndmay2022_Shift    : OUT std_logic_vector(4 downto 0)
    );

END emdad_control_unit_22ndmay2022;
```

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

```
ARCHITECTURE struct OF emdad_control_unit_22ndmay2022 IS

BEGIN
    emdad_22ndmay2022_Shift    <= emdad_22ndmay2022_shiftAmount WHEN ((emdad_22ndmay2022_opcode = "000000") AND
    (emdad_22ndmay2022_func = "000000" OR emdad_22ndmay2022_func = "000010" OR emdad_22ndmay2022_func = "000011")) ELSE "00000";
    emdad_22ndmay2022_RegDst      <= '0' WHEN (emdad_22ndmay2022_opcode = "000000") ELSE '1';
    emdad_22ndmay2022_ALUsrc      <= '0' WHEN (emdad_22ndmay2022_opcode = "000000" or
    (emdad_22ndmay2022_opcode = "000100") or (emdad_22ndmay2022_opcode = "000101")) ELSE '1';
    emdad_22ndmay2022_MemWrite <= '1' WHEN (emdad_22ndmay2022_opcode = "101011") ELSE '0';
    emdad_22ndmay2022_RegWrite <= '0' WHEN (emdad_22ndmay2022_opcode = "101011"
    or emdad_22ndmay2022_opcode = "000100" or emdad_22ndmay2022_opcode = "000101" or emdad_22ndmay2022_opcode = "000010") ELSE '1';
    emdad_22ndmay2022_MemtoReg <= '1' WHEN (emdad_22ndmay2022_opcode = "100011") ELSE '0';
    emdad_22ndmay2022_jump     <= '1' WHEN (emdad_22ndmay2022_opcode = "000010") ELSE '0';
    emdad_22ndmay2022_Branch_EQ   <= '1' WHEN (emdad_22ndmay2022_opcode = "000100") ELSE '0';
    emdad_22ndmay2022_Branch_NE   <= '1' WHEN (emdad_22ndmay2022_opcode = "000101") ELSE '0';
    emdad_22ndmay2022_PCsrc <= '1' WHEN(emdad_22ndmay2022_opcode = "101011") ELSE '0';
    emdad_22ndmay2022_ALUOp  <= "0000" WHEN (emdad_22ndmay2022_opcode = "001000") ELSE ---addi---
                          "0000" WHEN (emdad_22ndmay2022_func = "100000" AND emdad_22ndmay2022_opcode = "000000") ELSE ---add
                          "0001" WHEN (emdad_22ndmay2022_opcode = "001001") ELSE ---addiu
                          "0001" WHEN (emdad_22ndmay2022_opcode = "101011") ELSE ---sw
                          "0001" WHEN (emdad_22ndmay2022_opcode = "100011") ELSE ---lw
                          "0001" WHEN (emdad_22ndmay2022_func = "100001" AND emdad_22ndmay2022_opcode = "000000") ELSE ---addu
                          "0010" WHEN (emdad_22ndmay2022_opcode = "000100") ELSE ---beq
                          "0010" WHEN (emdad_22ndmay2022_opcode = "000101") ELSE ---bne
                          "0010" WHEN (emdad_22ndmay2022_func = "100010" AND emdad_22ndmay2022_opcode = "000000") ELSE ---sub
                          "0011" WHEN (emdad_22ndmay2022_func = "100011" AND emdad_22ndmay2022_opcode = "000000") ELSE ---subu
                          "0100" WHEN (emdad_22ndmay2022_opcode = "001100") ELSE ---andi
                          "0100" WHEN (emdad_22ndmay2022_func = "100100" AND emdad_22ndmay2022_opcode = "000000") ELSE ---and
                          "0101" WHEN (emdad_22ndmay2022_func = "100111" AND emdad_22ndmay2022_opcode = "000000") ELSE ---nor
                          "0110" WHEN (emdad_22ndmay2022_opcode = "001101") ELSE ---ori
                          "0110" WHEN (emdad_22ndmay2022_func = "100101" AND emdad_22ndmay2022_opcode = "000000") ELSE ---or
                          "0111" WHEN (emdad_22ndmay2022_func = "000000" AND emdad_22ndmay2022_opcode = "000000") ELSE ---sll
                          "1000" WHEN (emdad_22ndmay2022_func = "000010" AND emdad_22ndmay2022_opcode = "000000") ELSE ---srl
                          "1001" WHEN (emdad_22ndmay2022_func = "000011" AND emdad_22ndmay2022_opcode = "000000") ELSE ---sra
                          "1010" WHEN (emdad_22ndmay2022_func = "011000" AND emdad_22ndmay2022_opcode = "000000") ELSE ---MUL
                          "1011" WHEN (emdad_22ndmay2022_func = "011010" AND emdad_22ndmay2022_opcode = "000000") ELSE ---DIV
                      (OTHERS => '0');
END struct;
```

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

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

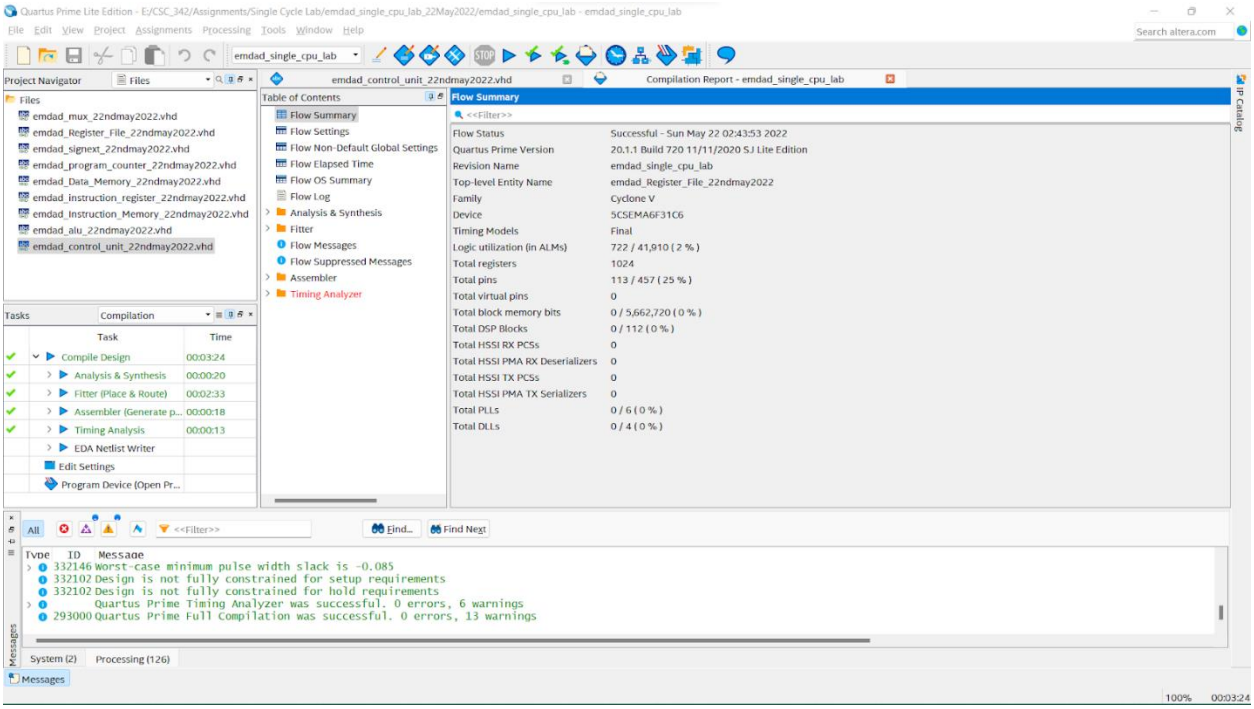


Figure 23: compiled successfully

MIPS:

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

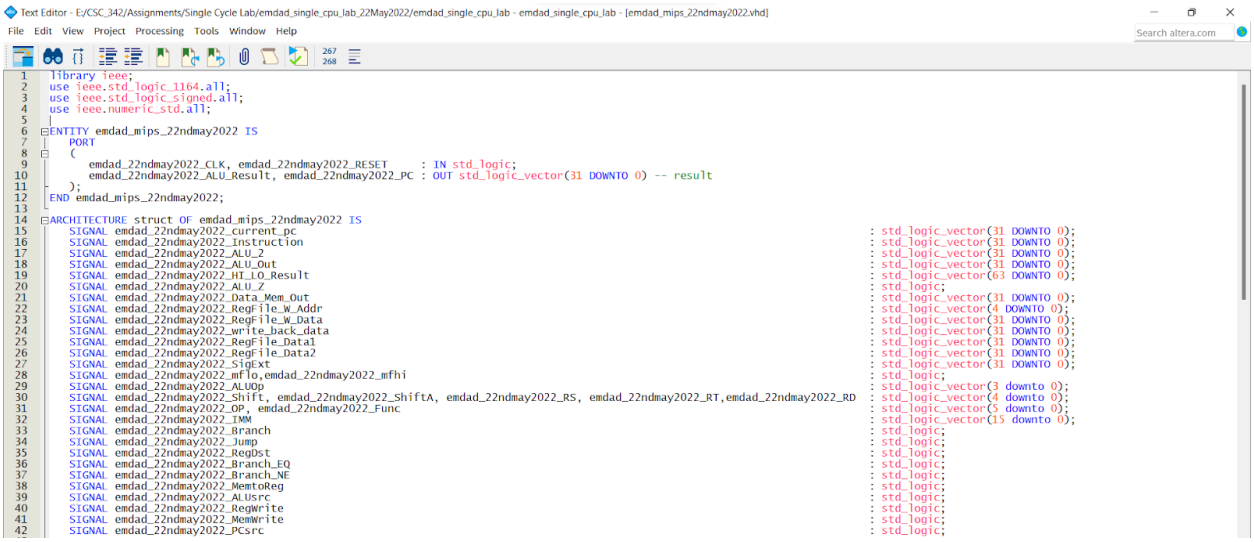


Figure 24: VHDL code the CPU Datapath (part 1)

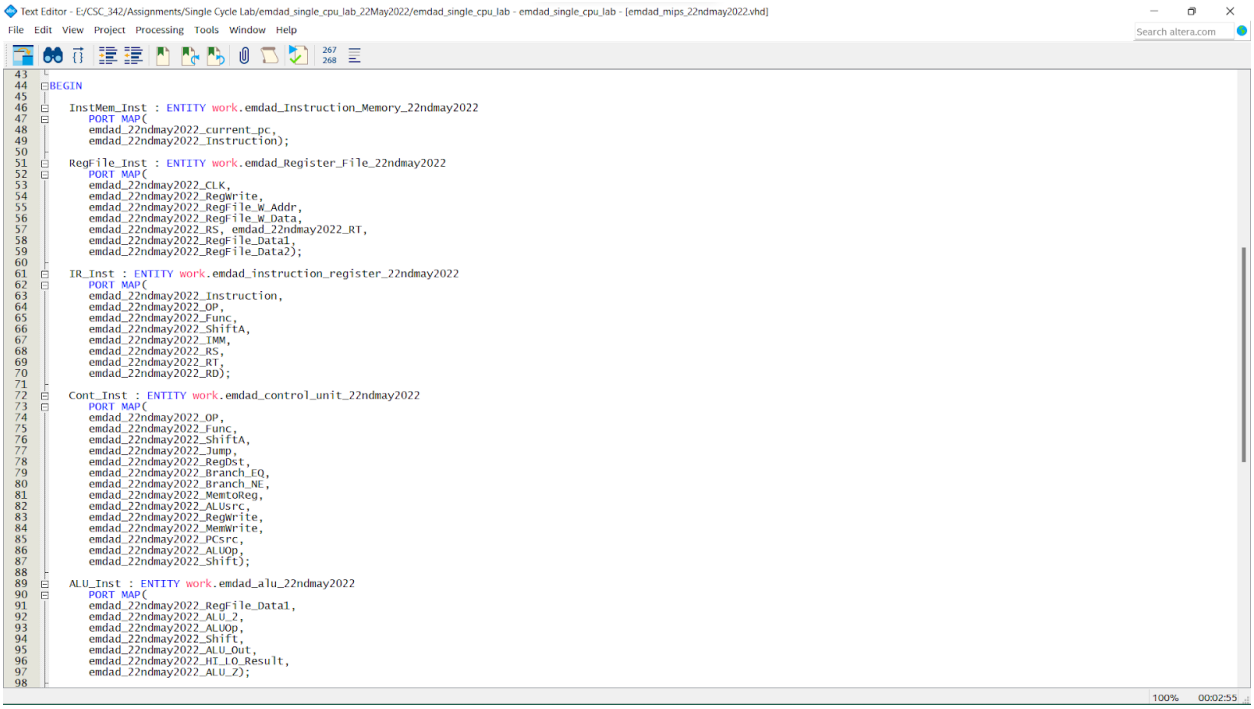
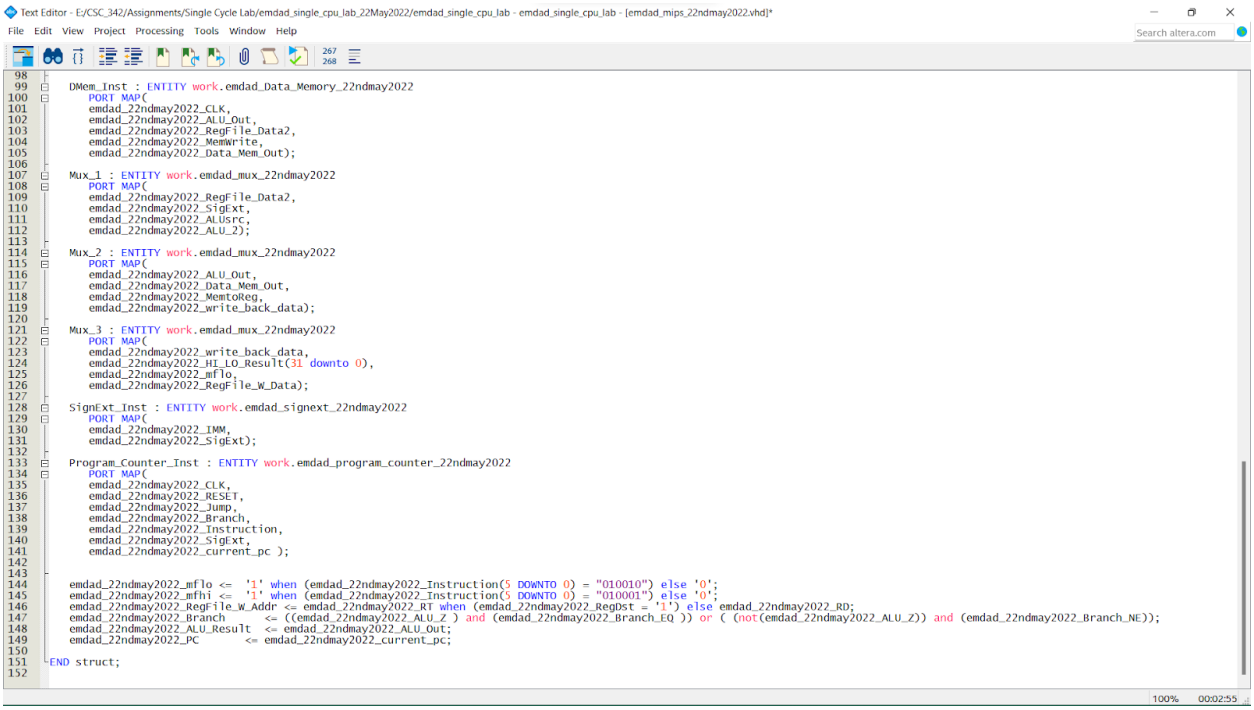


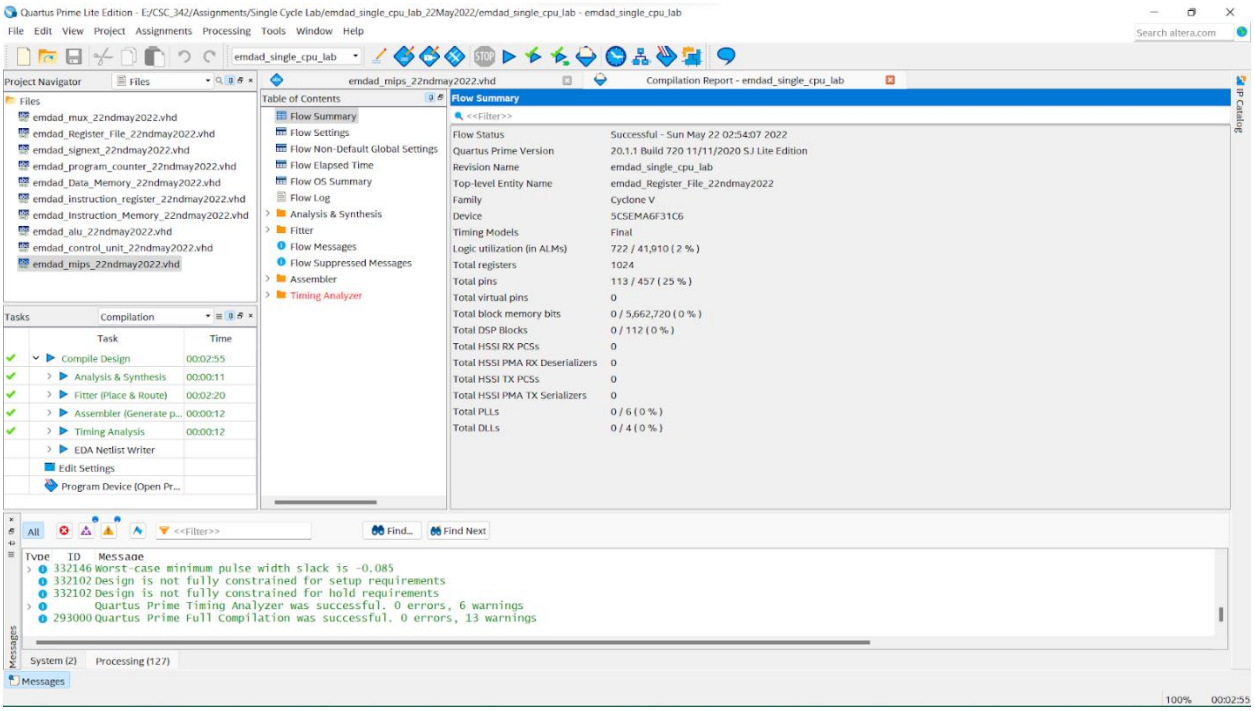
Figure 25: VHDL code the CPU Datapath (part 2)



```
98
99    DMem_Inst : ENTITY work.emdad_Data_Memory_22ndmay2022
100      PORT MAP(
101      emdad_22ndmay2022_CLK,
102      emdad_22ndmay2022_ALU_Out,
103      emdad_22ndmay2022_RegFile_Data2,
104      emdad_22ndmay2022_MemWrite,
105      emdad_22ndmay2022_Data_Mem_Out);
106
107   Mux_1 : ENTITY work.emdad_mux_22ndmay2022
108      PORT MAP(
109      emdad_22ndmay2022_RegFile_Data2,
110      emdad_22ndmay2022_SigExt,
111      emdad_22ndmay2022_ALUsrc,
112      emdad_22ndmay2022_ALU_2);
113
114   Mux_2 : ENTITY work.emdad_mux_22ndmay2022
115      PORT MAP(
116      emdad_22ndmay2022_ALU_Out,
117      emdad_22ndmay2022_Data_Mem_Out,
118      emdad_22ndmay2022_MemtoReg,
119      emdad_22ndmay2022_write_back_data);
120
121   Mux_3 : ENTITY work.emdad_mux_22ndmay2022
122      PORT MAP(
123      emdad_22ndmay2022_write_back_data,
124      emdad_22ndmay2022_HI_LO_Result(31 downto 0),
125      emdad_22ndmay2022_mflo,
126      emdad_22ndmay2022_RegFile_W_Data);
127
128   SignExt_Inst : ENTITY work.emdad_signext_22ndmay2022
129      PORT MAP(
130      emdad_22ndmay2022_IMM,
131      emdad_22ndmay2022_SigExt);
132
133   Program_Counter_Inst : ENTITY work.emdad_program_counter_22ndmay2022
134      PORT MAP(
135      emdad_22ndmay2022_CLK,
136      emdad_22ndmay2022_RESET,
137      emdad_22ndmay2022_Jump,
138      emdad_22ndmay2022_Branch,
139      emdad_22ndmay2022_Instruction,
140      emdad_22ndmay2022_SigExt,
141      emdad_22ndmay2022_current_pc );
142
143
144   emdad_22ndmay2022_mflo <=  '1' when (emdad_22ndmay2022_Instruction(5 DOWNTO 0) = "010010") else '0';
145   emdad_22ndmay2022_mfhi <=  '1' when (emdad_22ndmay2022_Instruction(5 DOWNTO 0) = "010001") else '0';
146   emdad_22ndmay2022_RegFile_W_Addr <= emdad_22ndmay2022_RT when (emdad_22ndmay2022_RegDst = '1') else emdad_22ndmay2022_RD;
147   emdad_22ndmay2022_Branch      <= ((emdad_22ndmay2022_ALU_Z ) and (emdad_22ndmay2022_Branch_EQ )) or ( (not(emdad_22ndmay2022_ALU_Z)) and (emdad_22ndmay2022_Branch_NE));
148   emdad_22ndmay2022_ALU_Result  <= emdad_22ndmay2022_ALU_Out;
149   emdad_22ndmay2022_PC       <= emdad_22ndmay2022_current_pc;
150
151 END struct;
152
```

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

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



```
Flow Summary
Flow Status                        Successful - Sun May 22 02:54:07 2022
Quartus Prime Version              20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name                      emdad_single_cpu_lab
Top-level Entity Name              emdad_Register_File_22ndmay2022
Family                             Cyclone V
Device                             5CSEMA6F31C6
Timing Models                      Final
Logic utilization (in ALMs)        722 / 41,910 ( 2 % )
Total registers                    1024
Total pins                         113 / 457 ( 25 % )
Total virtual pins                 0
Total block memory bits            0 / 5,662,720 ( 0 % )
Total DSP Blocks                   0 / 112 ( 0 % )
Total HSSI RX PCSs                 0
Total HSSI PMA RX Deserializers    0
Total HSSI TX PCSs                 0
Total HSSI PMA TX Serializers      0
Total PLLs                         0 / 6 ( 0 % )
Total DLLs                         0 / 4 ( 0 % )

Tasks
Task                    Time
Compile Design          00:02:55
Analysis & Synthesis    00:00:11
Fitter (Place & Route)  00:02:20
Assembler (Generate p...  00:00:12
Timing Analysis         00:00:12
EDA Netlist Writer
Edit Settings
Program Device (Open Pr...

Type  ID      Message
332146 Worst-case minimum pulse width slack is -0.085
332102 Design is not fully constrained for setup requirements
332102 Design is not fully constrained for hold requirements
       Quartus Prime Timing Analyzer was successful. 0 errors, 6 warnings
293000 Quartus Prime Full Compilation was successful. 0 errors, 13 warnings
```

Figure 27: compiled successfully

Block Diagrams:

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

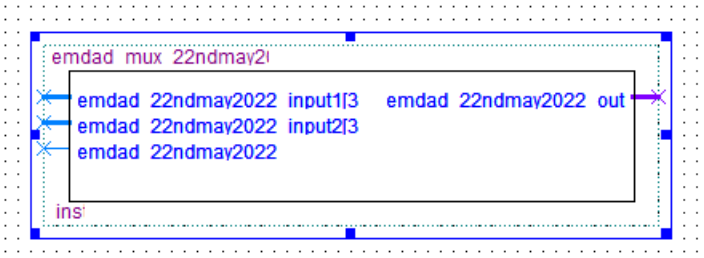


Figure 28: symbol diagram generated from the VHDL code of mux file

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

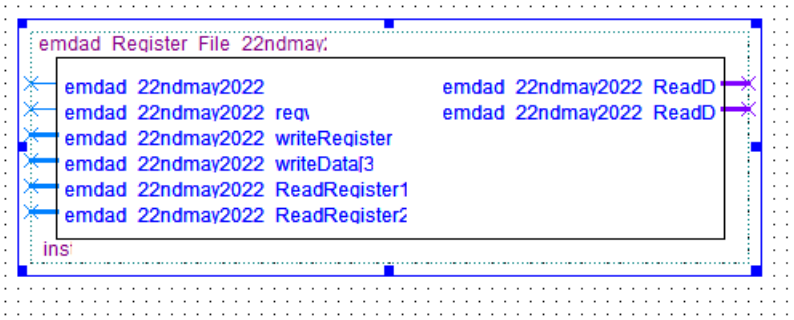


Figure 29: symbol diagram generated from the VHDL code of register file.

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

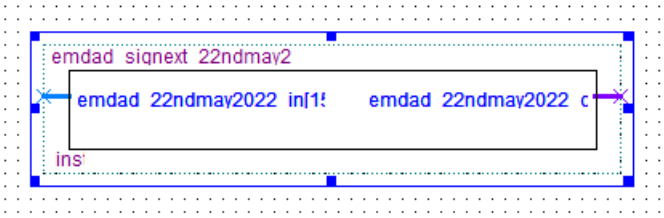


Figure 30: symbol diagram generated from VHDL code of sign extender file

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

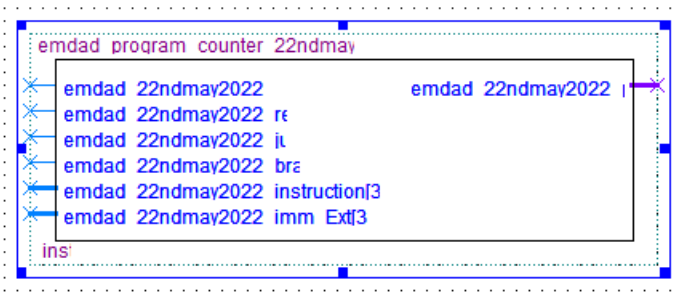


Figure 31: symbol diagram generated from VHDL code of program counter file

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

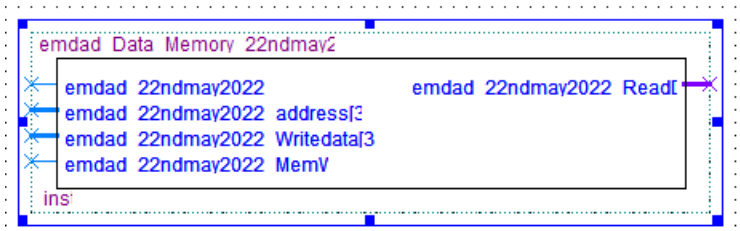


Figure 32: symbol diagram generated from VHDL code of data memory file

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

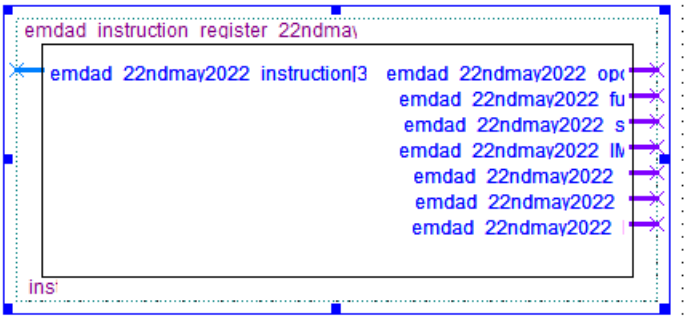


Figure 33: symbol diagram generated from VHDL code of instruction register file

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

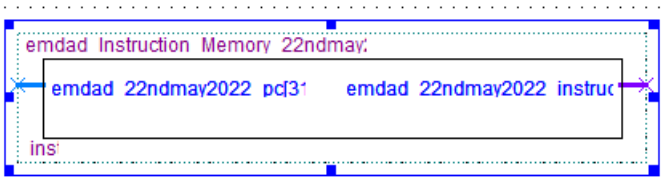


Figure 34: symbol diagram generated from VHDL code of instruction memory file

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

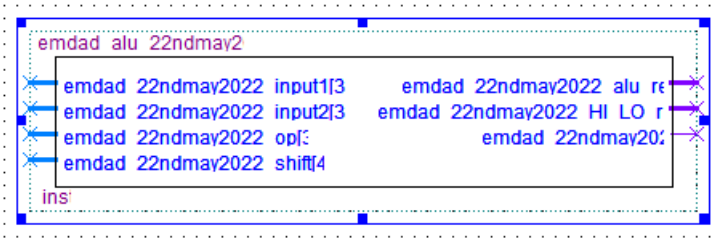


Figure 35: symbol diagram generated from VHDL code of alu file

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

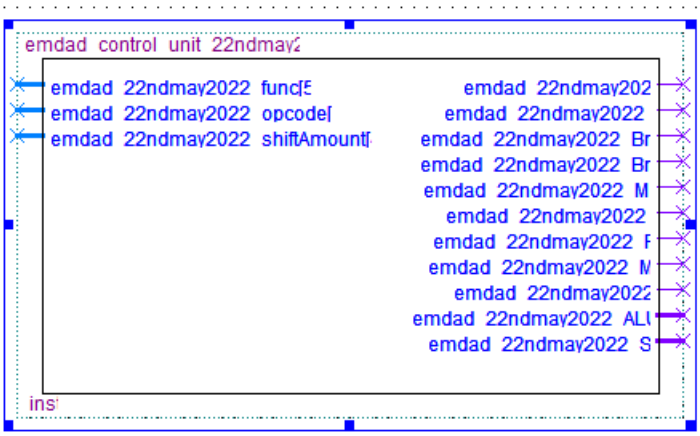


Figure 36: symbol diagram generated from VHDL code of control unit file

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

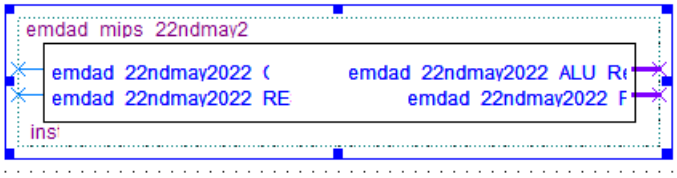


Figure 37: symbol diagram generated from VHDL code of mips file

MIPS:

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

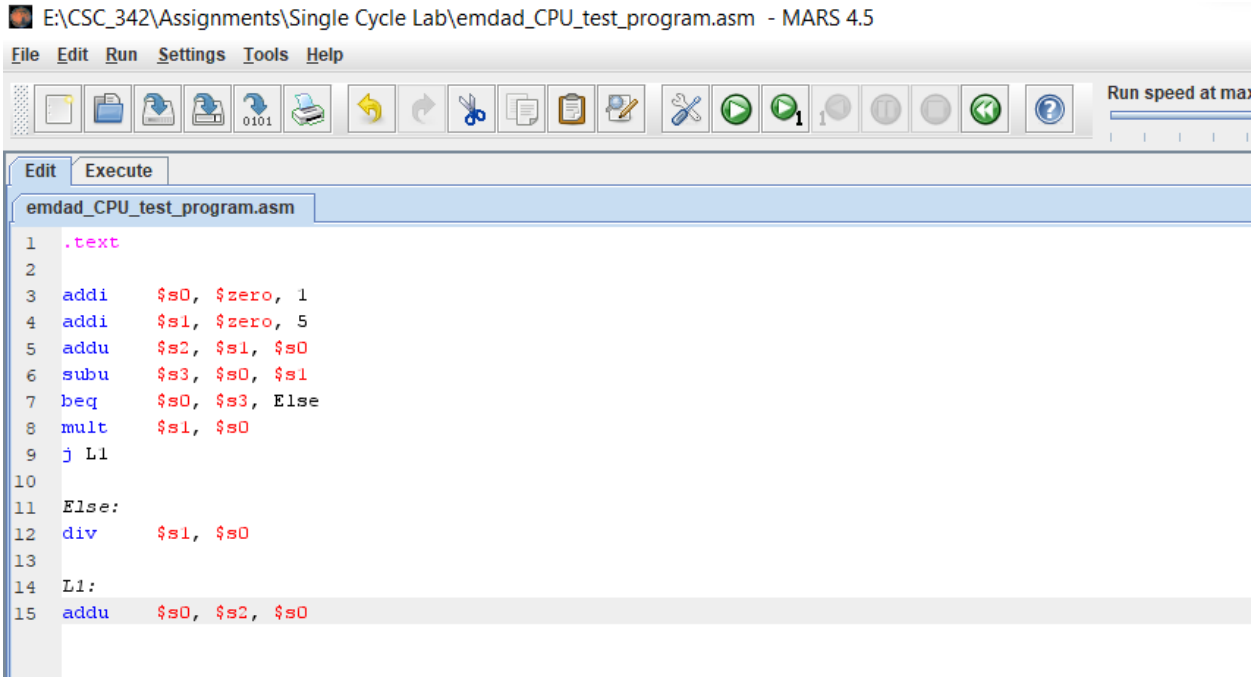


Figure 38: MIPS code

In figure 39, we can see it assembles successfully.

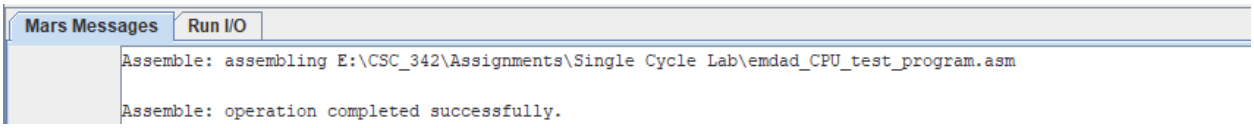


Figure 39: assembles successfully

Simulation:

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

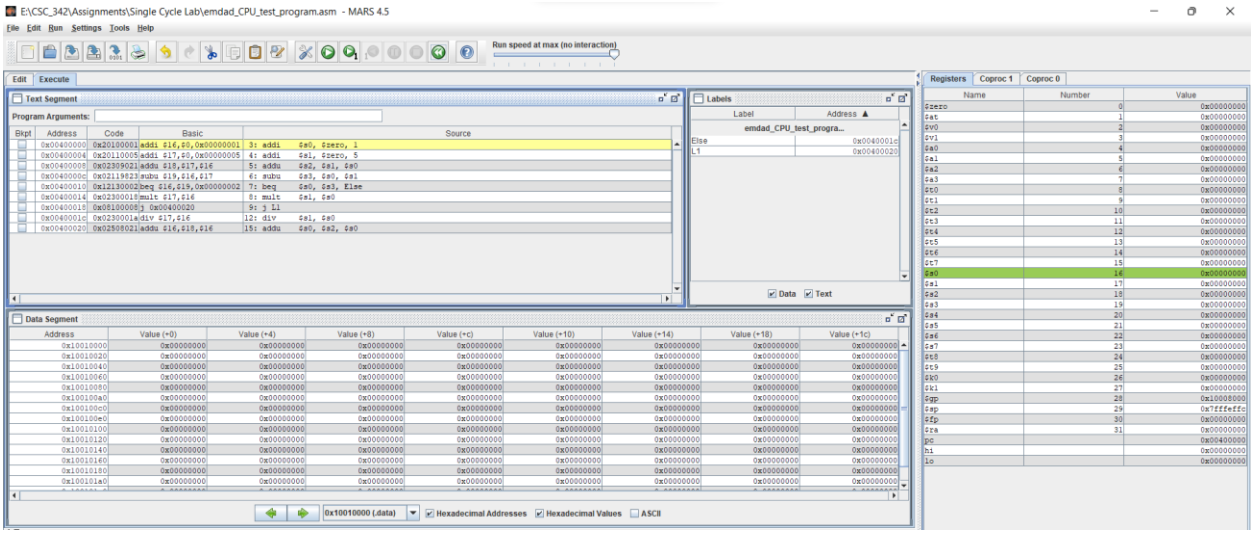


Figure 40: text segment, data segment, registers

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 $s2is 6 and it jumped it to **lo**.

$s0	16	0x00000001
$s1	17	0x00000005
$s2	18	0x00000006
$s3	19	0xfffffffc
$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	0x7fffeffc
$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.

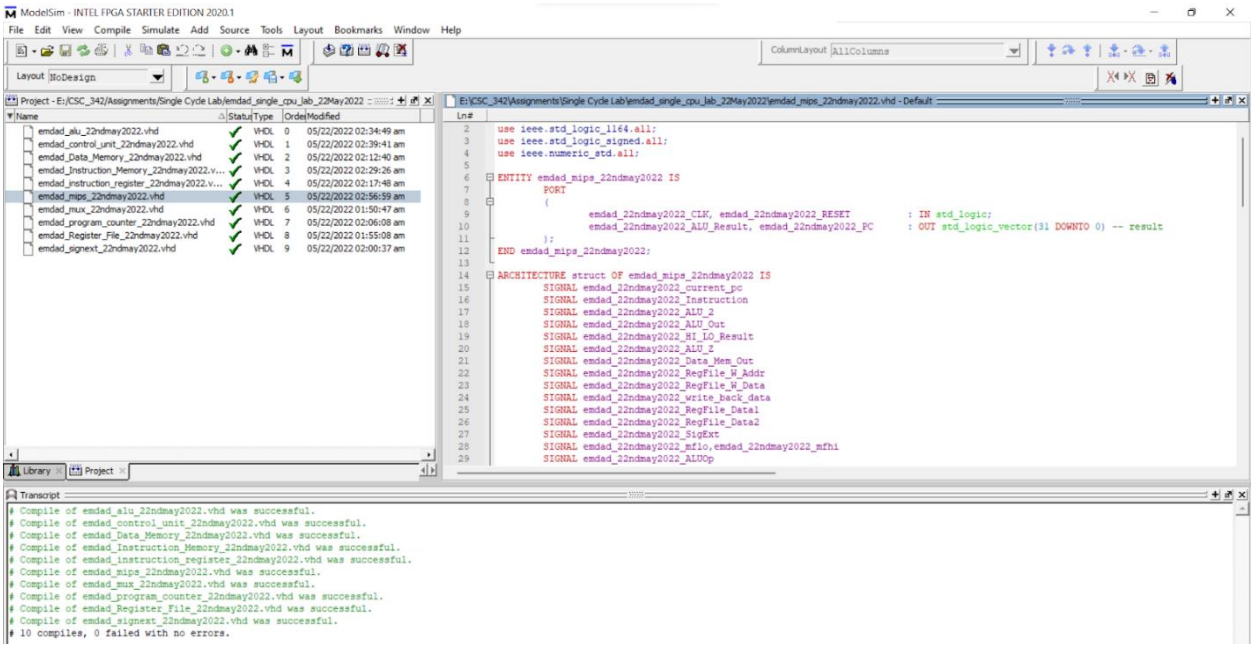


Figure 41: directory for the CPU project for ModelSim

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

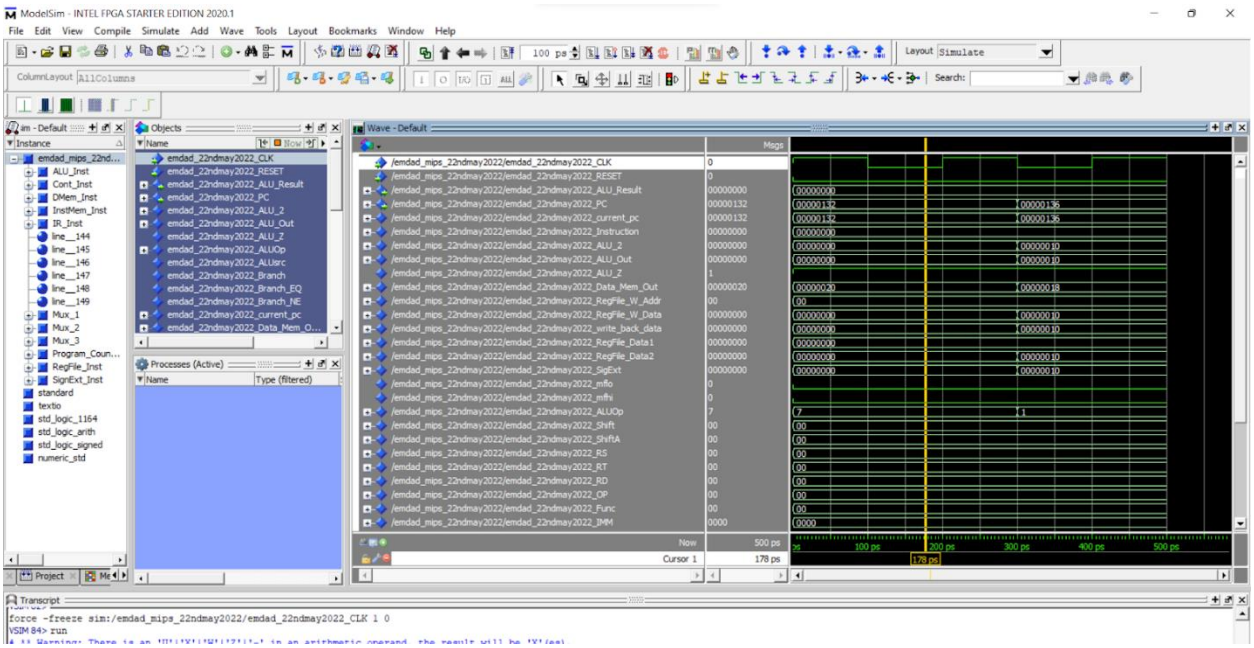


Figure 41: add operation

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.

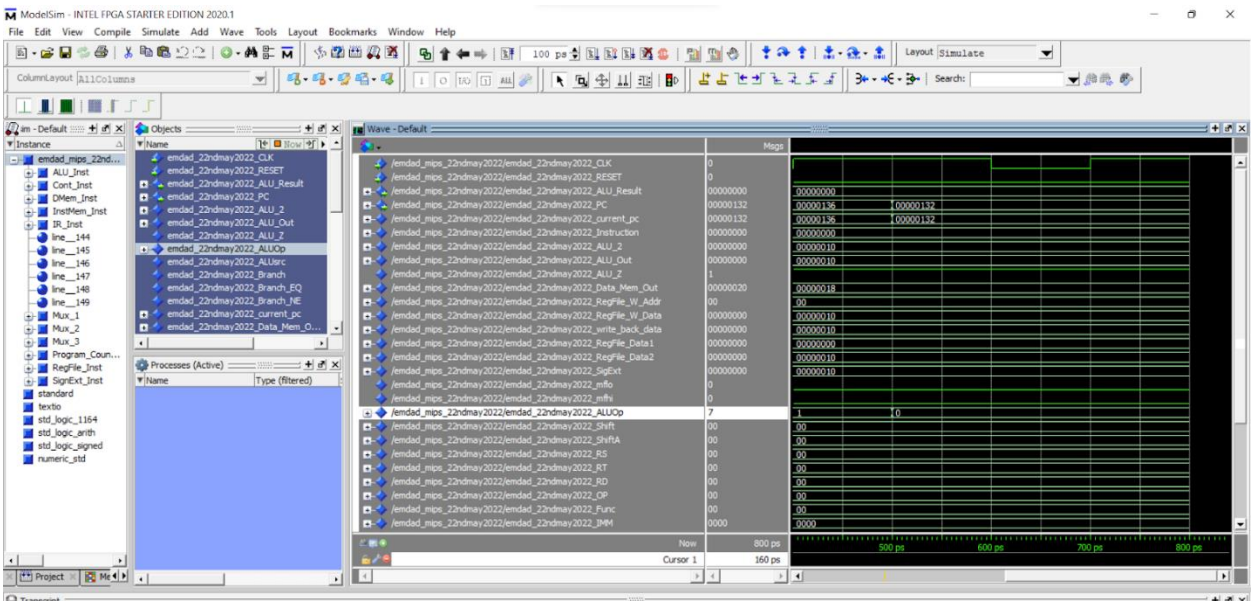


Figure 42: subtraction operation

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

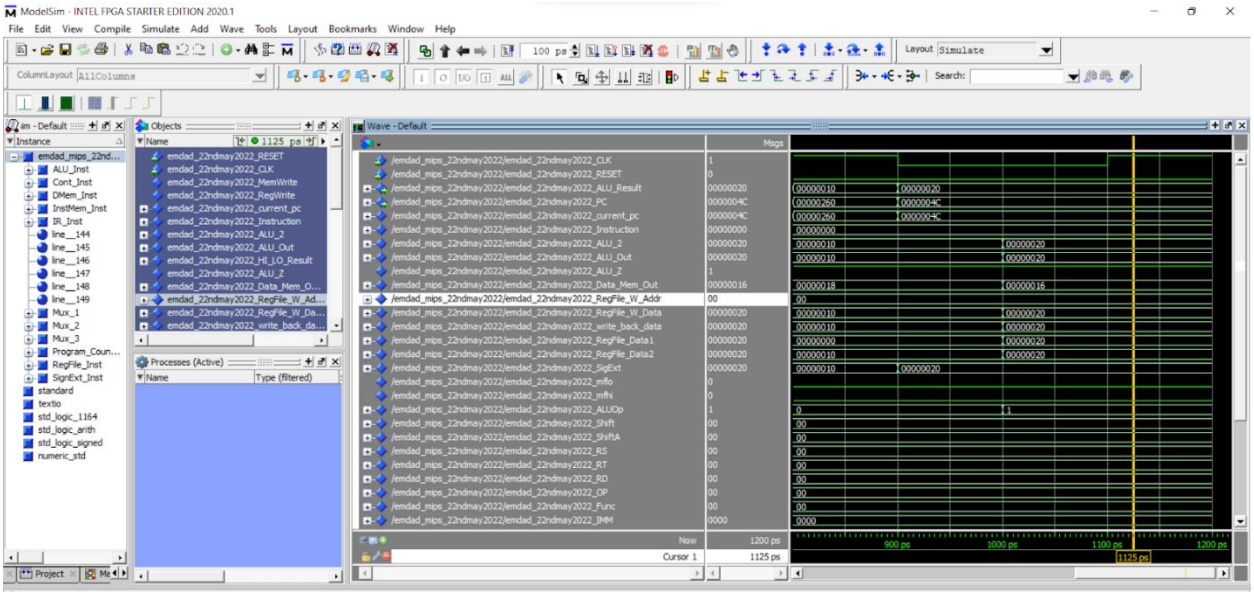


Figure 43: multiplication operation

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.