ENGG*3380

Dr. Abou El Nasr

April 7th, 2023

ENGG*3380 Project

Group 6

Yasir Al-Obaidi (1145544)

Pratham Patel (1140832)

Contents

Problem Statement

The challenge of this project was to create a working CPU and test the various instructions and components to ensure that the CPU is working properly. We had to complete some unfinished code and write new files in VHDL to complete this CPU.

Assumptions and Constraints

We were constrained to use the files provided by the lab instructor as a foundation of the VHDL code and build the new parts and connections from there. We were also instructed to use Vivado 2019 to build and test the CPU on.

System Overview

For the project we were provided the following files:

- ALU_16Bit.vhd
- ALU.vhd,

  and_gate.vhd
- Control.vhd
- CPU_3380_Test.vhd
- full_adder.vhd
- mux2_1.vhd
- mux3_1.vhd

  MUX21_4Bit.vhd
- MUX21_16Bit.vhd
- MUX31.vhd

- or_gate.vhd
- PC_REG.vhd
- Registers.vhd

  Memory.vhd
- Signextend.vhd.

Part 1

These files were for the most part completed and ready to be used for the project. For Part 1 of the project, we had to finish implementing the Data Memory Block, a 3-to-1 Multiplexor, Program Counter and Instruction Memory components into the top-level CPU module so that they can all be called properly with the correct outputs and inputs. We also had to test the design using the provided test bench for Part 1. The Top Level of the CPU is provided in Appendix A. All the code necessary for the project is provided in the appendix area as well, labeled clearly. Some of it will be referenced in the report (Ones that were changed or added by the group), while the rest is for reference.

The main portion of Part 1 was to finish implementing the component calls and port maps for the files provided by the instructor.

We had to implement the memory and program counter components, whose code is shown below:

Figure 1: Implementation of Components - Part 1

```
 component Memory
generic (
    INPUT : string := "in.txt";
    OUTPUT : string := "out.txt"
);
 port (
 -- TODO 1: Finish implementing the memory component
     clk : in std_logic;
     read_en : in std_logic;
     write_en : in std_logic;
     addr : in std_logic_vector(15 downto 0);
     data_in : in std_logic_vector(15 downto 0);
     data_out : out std_logic_vector(15 downto 0);
     mem_dump : in std_logic
 );
 end component;

 -- TODO 2: Finish implementing the program counter component
 component PC_REG
 port(
     clk : in std_logic;
     reset : in std_logic;
     Input : in std_logic_vector(15 downto 0);
     Output : out std_logic_vector(15 downto 0)
 );
```

Next, we had to finish the port maps of the components so that each part of the CPU is getting the correct inputs it needs to perform its function and the outputs from each part are also directed to its desired location. The components that we port mapped were Program Counter, Instruction Memory, Control Unit, Data Memory, and 3-to-1 Multiplexor. All these port maps can be seen in Appendix A, but I will include some here for reference.

Figure 2: Implementation of Port Maps – Part 1

```
-- TODO 4: Finish implementing the instruction memory
  CPU_Instr_MEM:          Memory generic map(INPUT => "Instr2.txt") port map(
      clk          =>      clk,
      read_en      =>      '1',
      write_en         =>      '0',
      addr             =>      pc_reg_output,
      data_in      =>      x"0000",
      data_out         =>      instruction,
      mem_dump     =>      '0'
  );

  op      <=  instruction(15 downto 12);
  rd      <= instruction(11 downto  8);
  rs      <= instruction(7  downto  4);
  rt      <= instruction(3  downto  0);
  ------------------------------------------------------------------------
  -- Instruction Decode
  ------------------------------------------------------------------------

-- TODO 5: Finish implementing the control port map
  CPU_Control_0:          Control port map(
      op               =>      op,
      alu_op       =>      ctrl_alu_op,
      alu_src      =>      ctrl_alu_src,
      reg_dest         =>      ctrl_reg_dest,
      reg_load         =>      ctrl_reg_load,
      reg_src      =>      ctrl_reg_src,
      mem_read         =>      ctrl_mem_read,
      mem_write    =>      ctrl_mem_write,
      jump         =>    jump_taken,
      branch       => branch_taken
  );
```

Another part of the project was to write the code for the PC register. This allowed for the CPU to move from one instruction to another as it would take an input and generate the next instruction location at the next rising edge of the clock. The code for that can be seen completely in Appendix Q, but it can also be shown below.

Figure 3: PC Register Code – Part 1

```
entity PC_REG is
    port(
        clk : in std_logic;
        reset : in std_logic;
        Input : in std_logic_vector(15 downto 0);
        Output : out std_logic_vector(15 downto 0)
    );
end PC_REG;

architecture Behavioral of PC_REG is
begin
    process(reset, clk)
    begin

--To do: write the code; if reset is zero, the output will be zero. Otherwise, at the rising edge of the clock, the input will be transfered to the output.

       if reset = '0' then
            output <= x"0000";
       elsif (rising_edge(clk) and reset='1') then
            output <= input;
       end if;
    end process;
end Behavioral;
```

For the purposes of part 1, the input of the PC register would always be the output + 2. Later in part 2, it gets a bit more complicated as we had to introduce 2 new instructions that impacted the PC register input directly. We will discuss that later.

The final part of Part 1 was to finish writing the code for the Control unit. This part would figure out which parts of the CPU would be required based on the input, which was the instructions or OP code of the CPU. Essentially it is responsible for fetching instructions, decoding those instructions, generating the correct signals to the various components that would be required and executing the instruction. The code is referenced in Appendix C and is partly shown below.

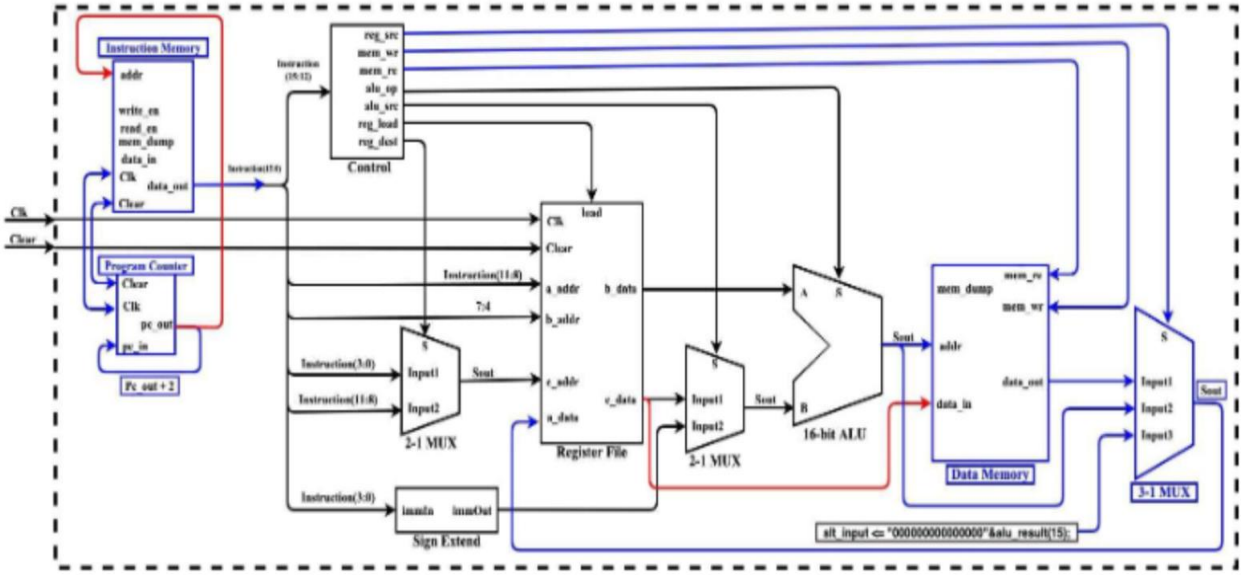
Figure 4: Control Unit Code – Part 1

```
-- op=4, ADDi
when x"4" =>
    alu_op      <=  "00";
    alu_src     <=  '1';
    reg_dest        <=  '0';
    reg_load        <=  '1';
    reg_src     <=  "01";
    mem_read        <=  '0';
    mem_write   <=  '0';
    jump        <= '0';
    branch      <= '0';

-- op=5, SUBi
when x"5" =>
    alu_op      <=  "01";
    alu_src     <=  '1';
    reg_dest        <=  '0';
    reg_load        <=  '1';
    reg_src     <=  "01";
    mem_read        <=  '0';
    mem_write   <=  '0';
    jump        <= '0';
    branch      <= '0';

-- op=8, LW
when x"8" =>
    alu_op      <=  "00";
    alu_src     <=  '1';
    reg_dest        <=  '0';
    reg_load        <=  '1';
    reg_src     <=  "00";
    mem_read        <=  '1';
    mem_write   <=  '0';
    jump        <= '0';
    branch      <= '0';

-- op=C, SW
when x"C" =>
    alu_op      <=  "00";
    alu_src     <=  '1';
    reg_dest        <=  '1';
    reg_load        <=  '0';
    reg_src     <=  "01";
    mem_read        <=  '0';
    mem_write   <=  '1';
    jump        <= '0';
    branch      <= '0';
```

As you can see, depending on the OP code, it will generate different signals for the various components and send those out as its outputs.

In the end, after the correct porting of inputs and outputs, Part 1 of the project looked like the figure below.

Figure 5: Top Module Design CPU – Part 1



Part 2

For part 2 of the project, we had to implement 2 new instructions for the CPU. These instructions were the JUMP and Branch if Not Equal (BNE) instructions. These are types of control flow instructions that would change the program counter (PC) to be something different than the usual PC+2 that we assumed in Part 1. To implement these new instructions, I needed to implement the following components and a few other signals in the top-level CPU module.

- Zero Detector
- 2 2-to-1 Multiplexors
- AND Gate

The BNE instruction required the ALU output as it uses it to determine if the two inputs are equal or not. The ALU output being 0 would mean that the two are equal hence the

instruction should not branch. To figure out this output, the group implemented a Zero Detector from scratch. It was not complicated; however, it essentially was an if statement that compared the output to 0 in hexadecimal. The code for the detector is available in full in Appendix P and partially in the figure below.

Figure 6: Zero Detector Code – Part 2

```
architecture Behavioral of Zero_Detector is
begin
    process (input_data)
    begin
        if input_data = "0000000000000000" then
            zero <= '1';
        else
            zero <= '0';
        end if;
    end process;
end architecture Behavioral;
```

Figure 7: Zero Detector Port Map Implementation

```
ZERO_DETECT:        Zero_Detector port map(
    input_data => alu_src_mux_out,
    zero => zero_flag
);
```

The Zero signal went on to go into a NOT gate as the zero detector outputs a 1 if the output of the ALU is 0 and we want to implement a branch if NOT equal and not a branch if equal. This inverted signal along with the branch signal output from the control unit then went into an AND gate that would ensure both are 1 for the instruction to continue. That AND gate was called PCSrc in the code. It is used as the selection signal for the first 2-to-1 multiplexor.

The BNE instruction also required the normal PC register output to be summed with the sign extend output before it can be used in a multiplexor with the PCSrc. Since those two signals

were already calculated, it was very easy to calculate a new signal as the sum of those in the top-level module and only required 1 line.

The signal referred to above and the original PC register output without the addition of the sign extend are the two inputs of the first multiplexor. This multiplexor, whose code is below in Appendix E, chooses either the regular PC+2 output, for non-control flow instructions or the BNE output if the branch signal is turned on. The top-level code for this is shown below.

Figure 8: Top Level Implementation of BNE Instruction

```
sum_PC_SE <= pc_plus_2 + sign_ex_out;

pc_plus_2 <= pc_reg_output + 2;
PCSrc <= (not zero_flag) and branch_taken;

PCSrc_MUX:          mux2_1 port map(
    Input1      =>      pc_plus_2,
    Input2      =>      sum_PC_SE,
    S               =>      PCSrc,
    Sout            =>      PCSrc_mux_out
);
```

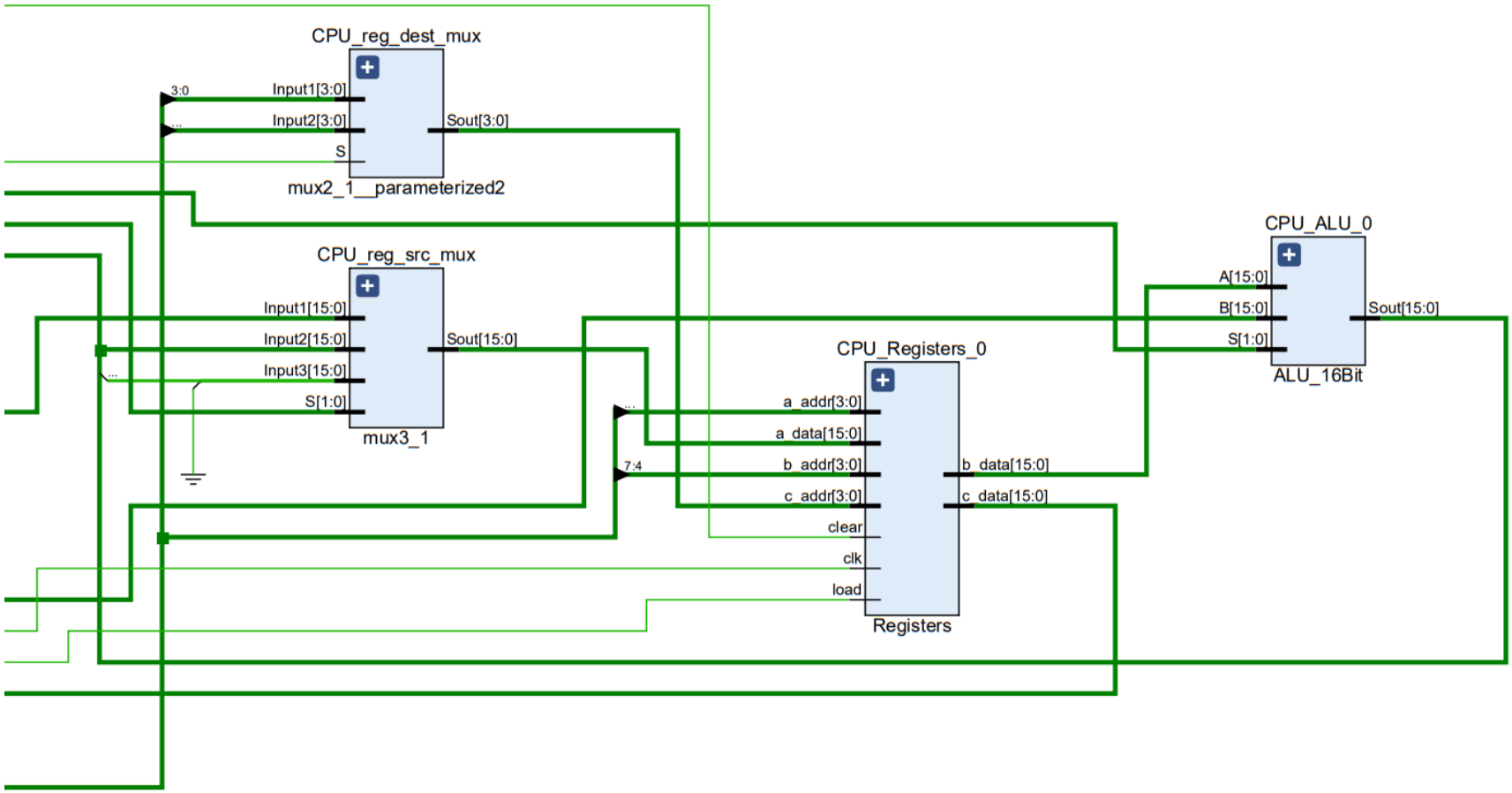
Next came implementing the JUMP instruction. It behaved similarly to the BNE instruction as it was another control flow instruction and just required another multiplexor to implement. This new multiplexor took in the output from the BNE multiplexor, which would be the regular PC+2 output anyways, and a new signal for the jump location, which is calculated by taking the top 4 bits of the PC+2 instruction and adding the last 12 bits of the instruction. This is done by simply ANDing the two parts together and storing the result in a signal called “jumpin” in the top-level module. Those two inputs, along with a selection signal, which was the jump signal from the control unit, were the 3 inputs for the last multiplexor. The output is then taken into the PC register to determine the next instruction location. Below is the code in the top-level module for the jump instruction.

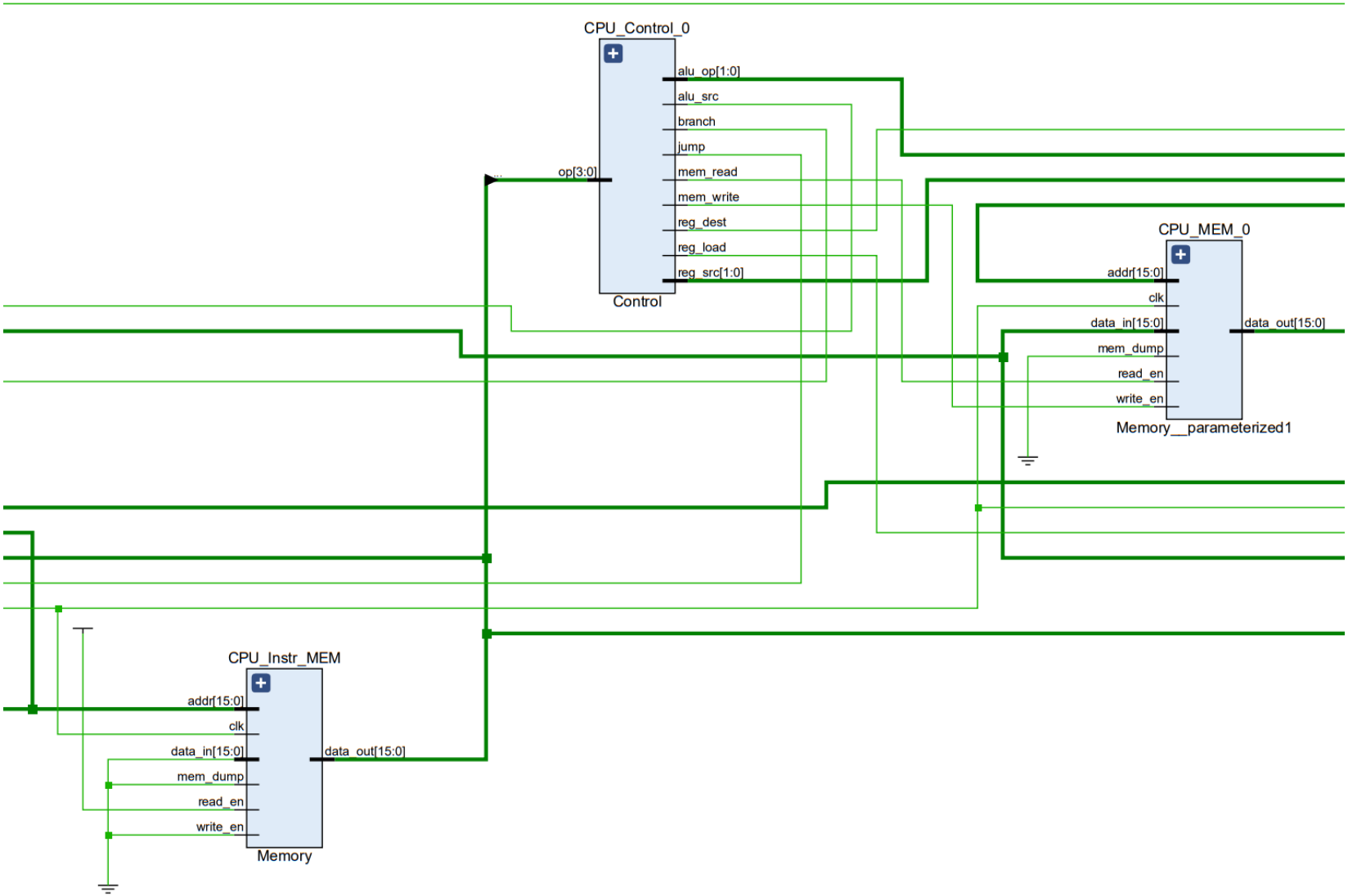
Figure 9: Top Level Implementation of JUMP Instruction

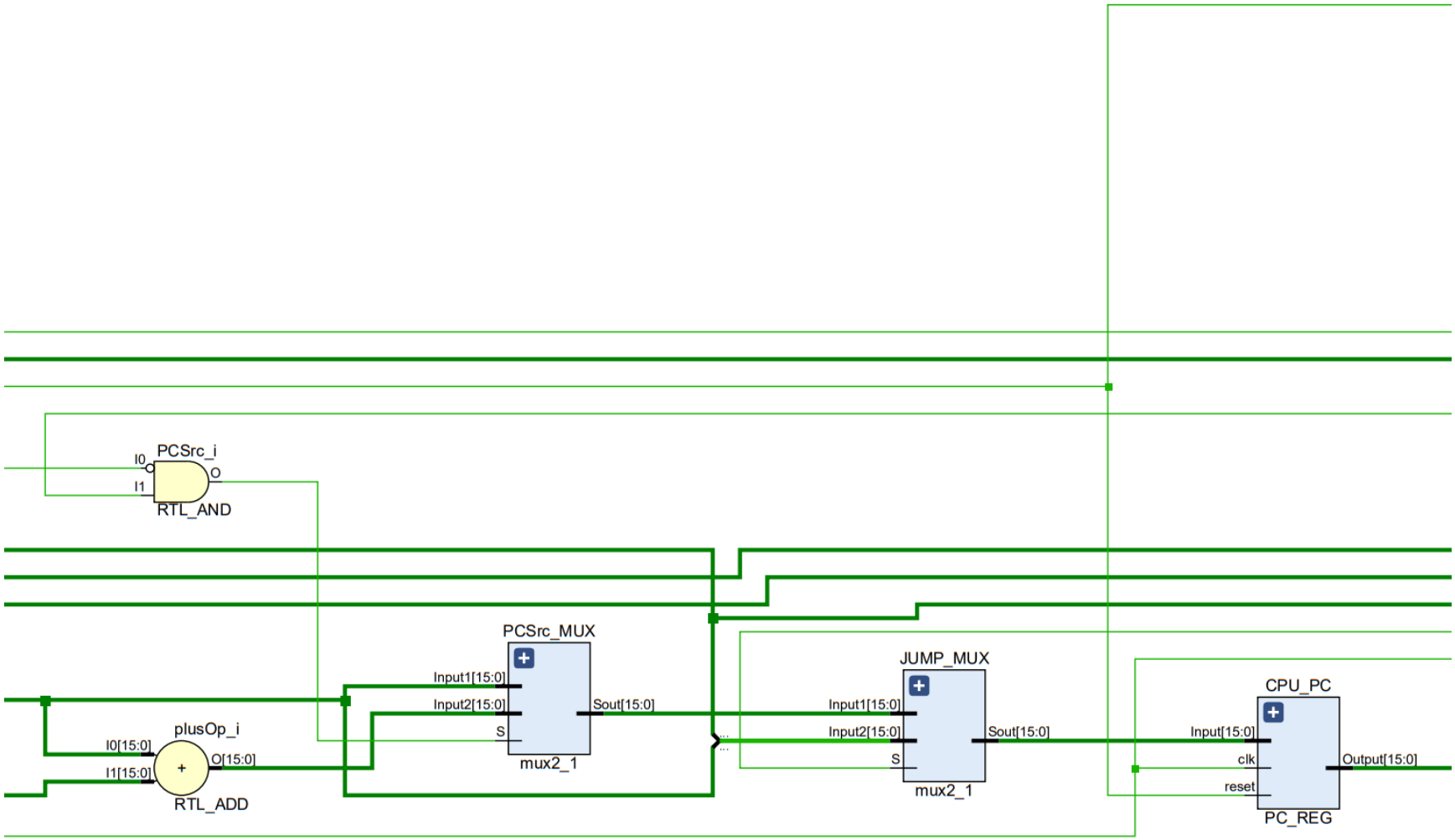
```
jumpin<= (pc_plus_2(15 downto 12) & instruction(11 downto 0));
JUMP_MUX:          mux2_1 port map(
    Input1      =>      PCSrc_mux_out,
    Input2      =>      jumpin,
    S               =>  jump_taken,
    Sout            =>  jumpout
);
```

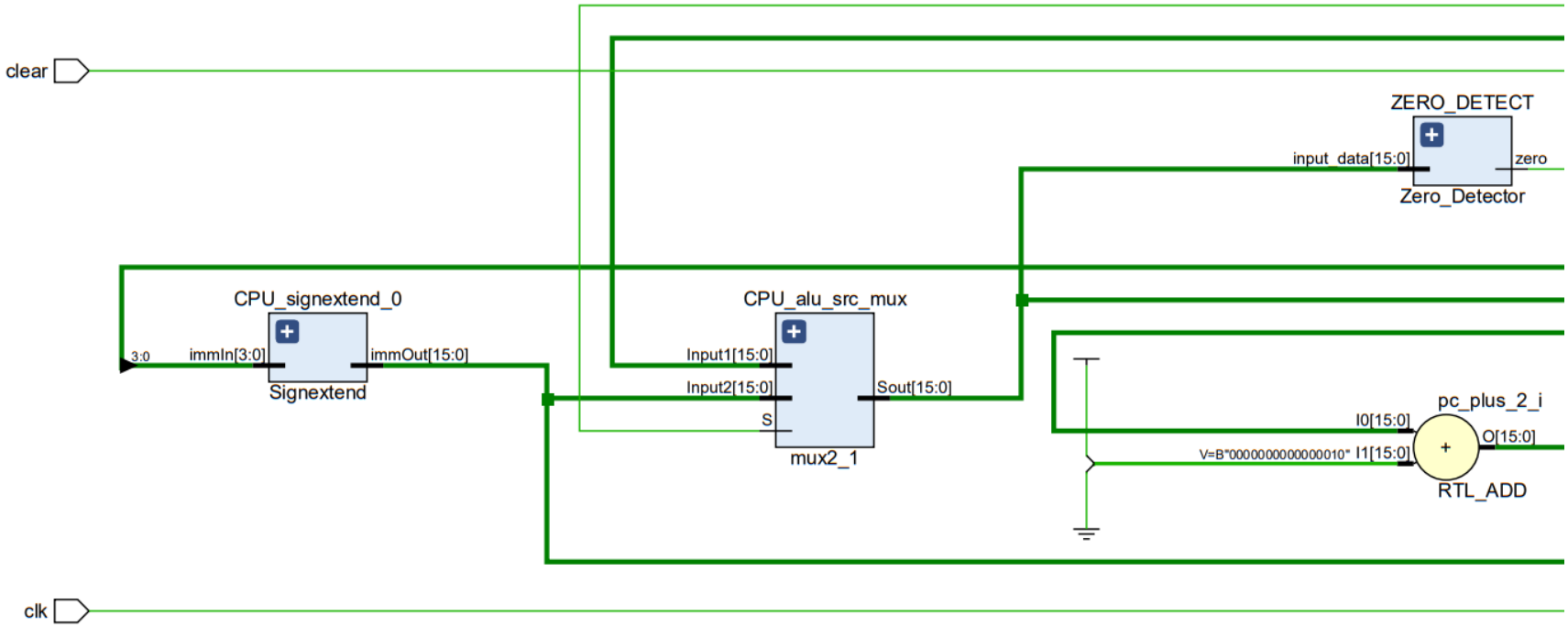
The final diagram of the project looks like the pictures below. Apologies for the separate pictures, I could not manage to fit it all in 1 image and still make it legible. The entire picture, zoomed out, is available in Appendix R.

Figure 10: Block Diagram of Entire Project


CPU_reg_dest_mux
3:0
Input1[3:0]
Input2[3:0]
S
Sout[3:0]
mux2_1__parameterized2
CPU_reg_src_mux
Input1[15:0]
Input2[15:0]
Input3[15:0]
S[1:0]
Sout[15:0]
mux3_1
CPU_Registers_0
7:4
a_addr[3:0]
a_data[15:0]
b_addr[3:0]
c_addr[3:0]
clear
clk
load
b_data[15:0]
c_data[15:0]
Registers
CPU_ALU_0
A[15:0]
B[15:0]
S[1:0]
Sout[15:0]
ALU_16Bit


CPU_Control_0
alu_op[1:0]
alu_src
branch
jump
op[3:0]
mem_read
mem_write
reg_dest
reg_load
reg_src[1:0]
Control
CPU_MEM_0
addr[15:0]
clk
data_in[15:0]
data_out[15:0]
mem_dump
read_en
write_en
Memory__parameterized1
CPU_Instr_MEM
addr[15:0]
clk
data_in[15:0]
data_out[15:0]
mem_dump
read_en
write_en
Memory


PCSrc_i
I0
I1
O
RTL_AND
plusOp_i
I0[15:0]
I1[15:0]
O[15:0]
+
RTL_ADD
PCSrc_MUX
Input1[15:0]
Input2[15:0]
S
Sout[15:0]
mux2_1
JUMP_MUX
Input1[15:0]
Input2[15:0]
S
Sout[15:0]
mux2_1
CPU_PC
Input[15:0]
clk
reset
Output[15:0]
PC_REG


clear
ZERO_DETECT
input_data[15:0]
zero
Zero_Detector
CPU_signextend_0
3:0
immIn[3:0]
immOut[15:0]
Signextend
CPU_alu_src_mux
Input1[15:0]
Input2[15:0]
S
Sout[15:0]
mux2_1
pc_plus_2_i
I0[15:0]
O[15:0]
V=B"0000000000000010"
I1[15:0]
RTL_ADD
clk

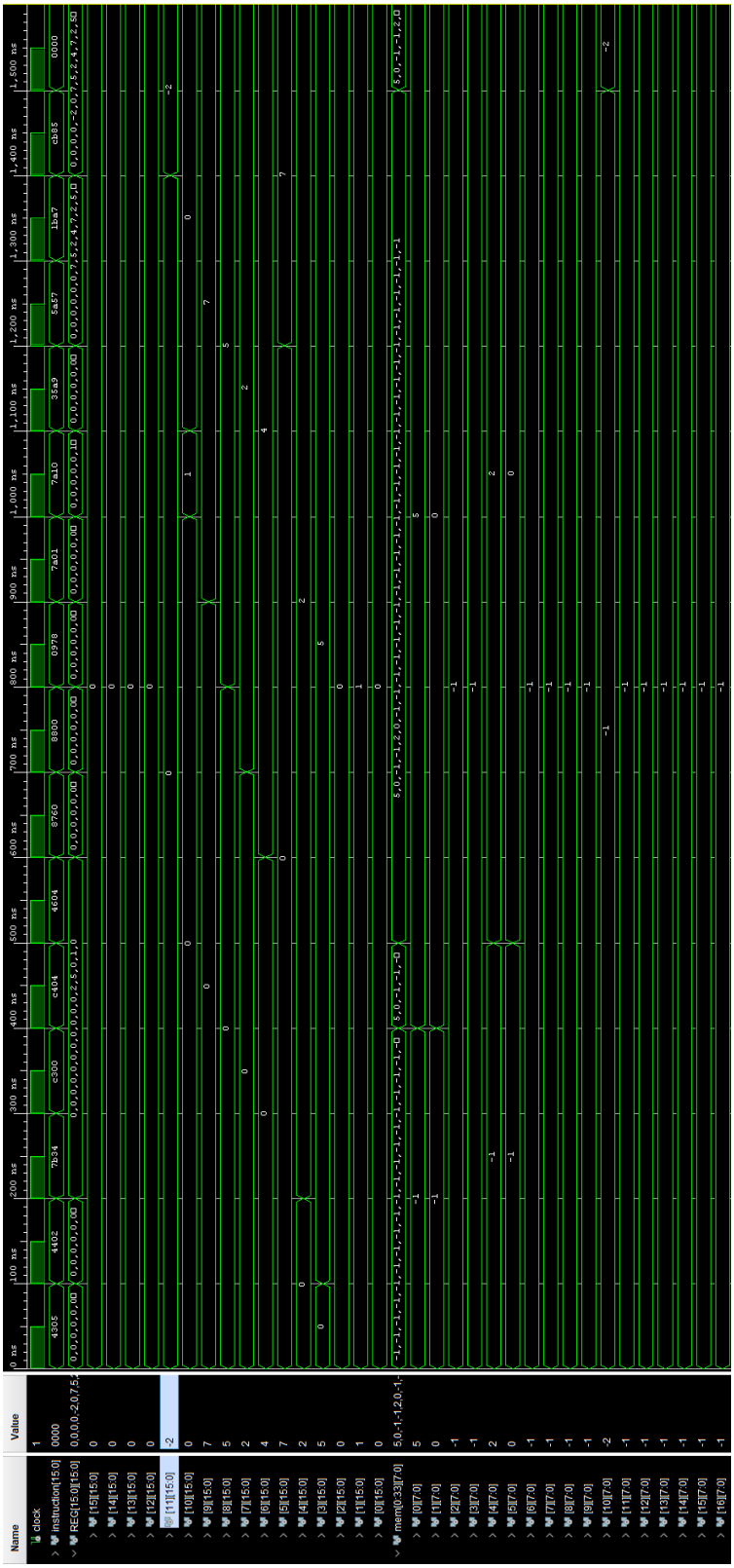
Verification

Part 1

For both parts, we used the provided test bench provided by the instructor, which is available in Appendix S. The input files were also provided by the instructor and part 1 and part 2 had different input instructions so changing the test bench was not required, just using the new instructions input file was enough.

For part 1, which tested the ADD, ADDI, SUB, SUBI, AND, OR, LW, SW and SLT instructions produced a simulation report that is shown below.

Figure 11: Part 1 Simulation

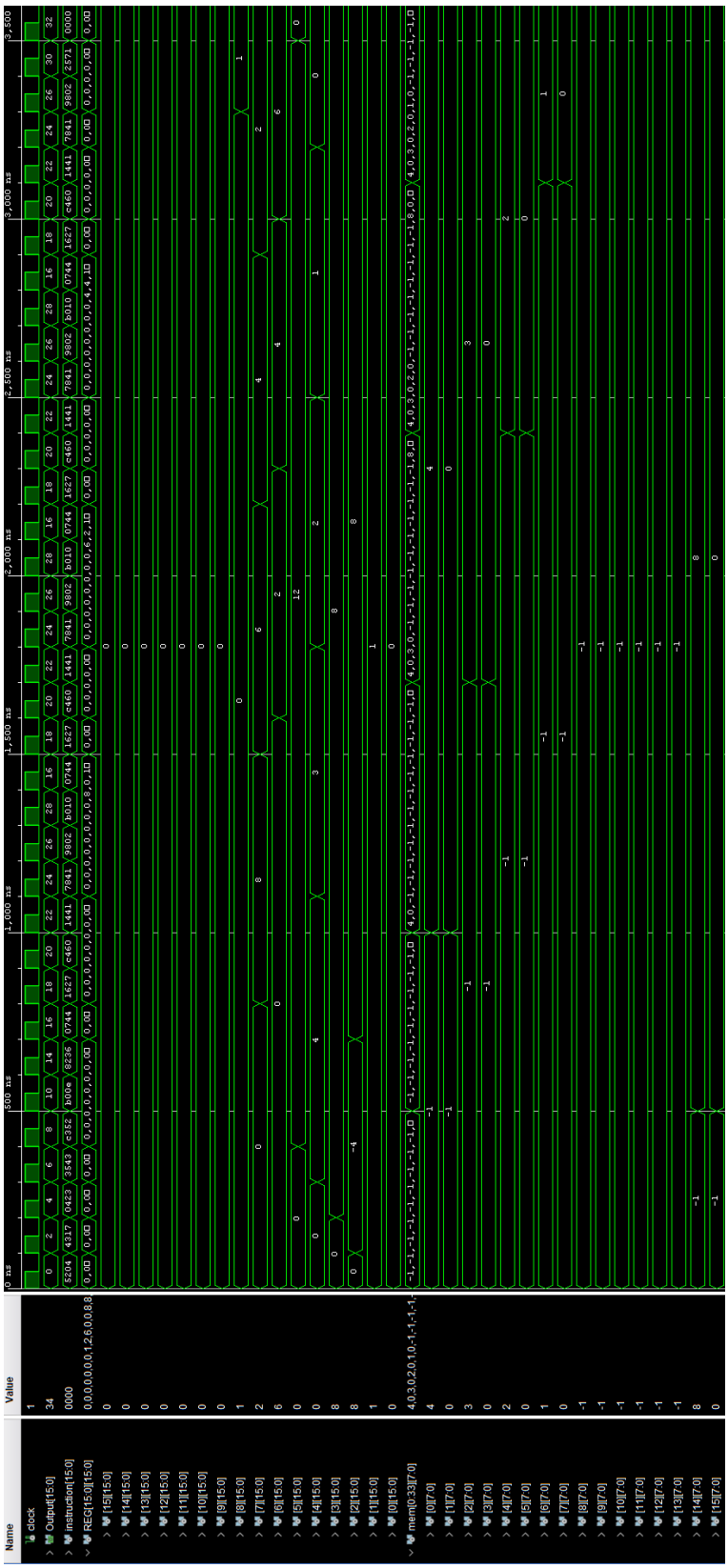


It is a bit difficult to read in this report, but the actual PNG file is included in the project files for reference. Anyways, depending on the input instructions, the values in the registers and memory addresses change on the next rising clock signal.

Part 2

For the second part's simulation, it was very similar to the previous one with the inclusion of the two new instructions, JUMP and BNE. To see the instructions working, the simulation required the PC register output to also be shown. The simulation for this part of the project is below.

Figure 12: Part 2 Simulation



Likewise to the Part 1 picture, this is also included in the project files for closer inspection. Here the second line, below the clock signal, the PC register is shown to show the impact of the BNE and JUMP instructions.

Errors and Issues

There were two main errors that the group experienced while working on the project. In the first part, there were no major errors that had us stumped as the instructions were laid out clearly but in part 2, where there was less guidance, is where we found the most trouble. The first major issue we had was figuring out the control outputs for the BNE instructions. We overlooked the fact that the BNE instruction required a comparison between the two inputs to determine the branch condition and had the ALU_OP signal as “00’ rather than “01”. This error was frustrating to fix as it was easily overlooked and seemed correct, but we eventually found it after reading a detailed explanation of what happens with branch instructions. The final major error was related to the Zero Detector output. The group forgot to invert the signal for the zero flag and as a result the BNE acted like a BEQ. We noticed this error when we tried to run the simulation and it ended quicker than it was supposed to as if the BNE instruction worked properly, it was supposed to loop back through some of the instructions again as it would have been equal the first time, however the test bench continued. We fixed this after that realization and everything else seemed to be working fine with no big issues.

Summary

Overall, this was a very fun project where I learned a lot about the inner workings of a CPU and was happy with the final output. The errors that we experienced were frustrating but, in

the end, only increased my knowledge of the CPU and its functionality as I had to understand what the code was doing differently compared to what I wanted it to do to be able to debug and fix the errors.

Appendix A – CPU Top Level

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use ieee.std_logic_arith.all;

use ieee.std_logic_unsigned.all;

--use IEEE.NUMERIC_STD.ALL;



entity CPU_3380 is
	port(
		clk	: in std_logic;
		clear : in std_logic;
		mem_dump : in std_logic := '0'
	);
end CPU_3380;
```

```
architecture Behavioral of CPU_3380 is

	COMPONENT ALU_16Bit

		port(

			A		:	in		std_logic_vector(15 downto
0);

			B		:	in		std_logic_vector(15 downto
0);

			S		:	in		std_logic_vector(1 downto
0);

			Sout	:	out	std_logic_vector(15 downto 0);

			Cout	:	out	std_logic

		);

	END COMPONENT;


	COMPONENT Registers

		port(

			clk		:	in	std_logic;

			clear	:	in  std_logic;
```

```
        a_addr      :   in      std_logic_vector( 3 downto 0);
        a_data      :   in      std_logic_vector(15 downto 0);
        load        :   in      std_logic;

        b_addr      :   in      std_logic_vector( 3 downto 0);
        c_addr      :   in      std_logic_vector( 3 downto 0);

        b_data      :   out std_logic_vector(15 downto 0);
        c_data      :   out std_logic_vector(15 downto 0)
    );
END COMPONENT;

COMPONENT Control
    port(
        op          :   in      std_logic_vector( 3 downto 0);
        alu_op      :   out     std_logic_vector( 1 downto 0);
        alu_src     :   out     std_logic;
        reg_dest    :   out     std_logic;
```

```
            reg_load        :       out     std_logic;
            reg_src         :       out     std_logic_vector( 1 downto 0);
            mem_read        :       out     std_logic;
            mem_write       :       out     std_logic;
            jump        : out std_logic;
        branch      : out std_logic
    );
end component;

component Signextend
    port(
            immIn           :       in      std_logic_vector( 3 downto 0);
            immOut              :       out     std_logic_vector(15 downto 0)
    );
end component;

component full_adder
  port(
```

```
        A   :       in      std_logic;
        B   :       in      std_logic;
        Cin :       in      std_logic;
        Sout:       out std_logic;
        Cout:       out     std_logic
    );
 end component;

    component mux3_1
generic (WIDTH : positive:=16);
    port(
        Input1      :   in      std_logic_vector(WIDTH-1  downto 0);
        Input2      :   in      std_logic_vector(WIDTH-1  downto 0);
        Input3      :   in      std_logic_vector(WIDTH-1  downto 0);
        S           :   in      std_logic_vector(1
    downto 0);
        Sout        :   out     std_logic_vector(WIDTH-1  downto 0));
    end component;
```

```
        component mux2_1
generic (WIDTH : positive:=16);
        port(
                Input1          :       in              std_logic_vector(WIDTH-1   downto 0);
                Input2          :       in              std_logic_vector(WIDTH-1   downto 0);
                S                       :       in              std_logic;
                Sout                    :       out     std_logic_vector(WIDTH-1   downto 0));
        end component;



        component Memory
generic (
    INPUT : string := "in.txt";
    OUTPUT : string := "out.txt"
);
 port (
        -- TODO 1: Finish implementing the memory component
     clk : in std_logic;
```

```
    read_en : in std_logic;
    write_en : in std_logic;
    addr : in std_logic_vector(15 downto 0);
    data_in : in std_logic_vector(15 downto 0);
    data_out : out std_logic_vector(15 downto 0);
    mem_dump : in std_logic
);
        end component;

        -- TODO 2: Finish implementing the program counter component
        component PC_REG
        port(
                clk : in std_logic;
                reset : in std_logic;
                Input : in std_logic_vector(15 downto 0);
                Output : out std_logic_vector(15 downto 0)
        );
        end component;
```

```
        component Zero_Detector
        Port (
       input_data : in STD_LOGIC_VECTOR (15 downto 0);
       zero : out STD_LOGIC
    );
    end component;


        -- Signals
        signal  instruction                 :       std_logic_vector(15 downto 0);
        signal   op                                 :       std_logic_vector( 3 downto
0);
        signal  rd                                  :       std_logic_vector( 3 downto
0);
        signal  rs                                  :       std_logic_vector( 3 downto
0);
        signal  rt                                  :       std_logic_vector( 3 downto
0);
```

```
signal  alu_result              :       std_logic_vector(15 downto 0);
signal  alu_src_mux_out     :       std_logic_vector(15 downto 0);
signal  sign_ex_out             :       std_logic_vector(15 downto 0);
signal  rs_data                 :       std_logic_vector(15 downto 0);
signal  rt_data                 :       std_logic_vector(15 downto 0);
signal  cout                                    :       std_logic
        ;
signal  reg_dest_mux_out    :       std_logic_vector( 3 downto 0);
signal  reg_src_mux_out     :       std_logic_vector(15 downto 0);
signal  mem_dataout             :       std_logic_vector(15 downto 0);


signal  ctrl_alu_src            :       std_logic;
signal  ctrl_alu_op             :       std_logic_vector( 1 downto 0);
signal  ctrl_reg_dest           :       std_logic;
signal  ctrl_reg_src            :       std_logic_vector( 1 downto 0);
signal  ctrl_reg_load           :       std_logic;
signal  ctrl_mem_read           :       std_logic;
signal  ctrl_mem_write          :       std_logic;
```

```
signal   slt_input                    :        std_logic_vector(15 downto 0);


--signal          reg_src_mux_input3:  std_logic_vector(15 downto 0);
signal   pc_plus_2                    :        std_logic_vector(15 downto 0);
signal   pc_reg_output          :        std_logic_vector(15 downto 0);


signal branch_taken        : std_logic;
signal jump_taken          : std_logic;




signal sum_PC_SE       : std_logic_vector(15 downto 0);
signal zero_flag       : std_logic;
signal PCSrc           : std_logic;
signal PCSrc_mux_out   : std_logic_vector(15 downto 0);
signal jumpin : std_logic_vector(15 downto 0);
signal jumpout : std_logic_vector(15 downto 0);
begin
```

```
        ----------------------------------------------------------------------

        -- Instruction Fetch

        ----------------------------------------------------------------------


  ZERO_DETECT:       Zero_Detector port map(

     input_data => alu_src_mux_out,

     zero => zero_flag

  );

-- TODO 3: Finish implementing the program counter port map

        CPU_PC:                                 PC_REG port map(

                clk                     =>              clk,

                reset                   =>              clear,

                input                   =>              jumpout,

                output          =>    pc_reg_output

        );


        sum_PC_SE <= pc_plus_2 + sign_ex_out;
```

```
pc_plus_2 <= pc_reg_output + 2;

PCSrc <= (not zero_flag) and branch_taken;



PCSrc_MUX:        mux2_1 port map(

   Input1                 =>                pc_plus_2,

         Input2           =>                sum_PC_SE,

         S                                  =>                PCSrc,

         Sout                        =>                PCSrc_mux_out

);

jumpin<= (pc_plus_2(15 downto 12) & instruction(11 downto 0));

JUMP_MUX:         mux2_1 port map(

   Input1                 =>                PCSrc_mux_out,

         Input2           =>                jumpin,

         S                                  =>       jump_taken,

         Sout                        =>       jumpout

);
```

```
 -- TODO 4: Finish implementing the instruction memory

        CPU_Instr_MEM:                  Memory generic map(INPUT => "Instr2.txt") port
map(

                clk                     =>              clk,

                read_en                 =>              '1',

                write_en                =>              '0',

                addr                    =>              pc_reg_output,

                data_in         =>              x"0000",

                data_out                =>              instruction,

                mem_dump        =>              '0'

        );



        op              <=      instruction(15 downto 12);

        rd              <= instruction(11 downto  8);

        rs              <= instruction(7  downto  4);

        rt              <= instruction(3  downto  0);

        ---------------------------------------------------------------------

        -- Instruction Decode

        ---------------------------------------------------------------------
```

```
-- TODO 5: Finish implementing the control port map

    CPU_Control_0:                  Control port map(

        op                          =>          op,

        alu_op          =>          ctrl_alu_op,

        alu_src         =>          ctrl_alu_src,

        reg_dest            =>          ctrl_reg_dest,

        reg_load            =>          ctrl_reg_load,

        reg_src         =>          ctrl_reg_src,

        mem_read            =>          ctrl_mem_read,

        mem_write   =>  ctrl_mem_write,

        jump    =>  jump_taken,

        branch   => branch_taken

    );


    CPU_Registers_0:            Registers port map(

        clk                 =>          clk,

        clear               =>          clear,
```

```
        a_addr          =>          rd,
        a_data          =>          reg_src_mux_out,
        load                =>          ctrl_reg_load,
        b_addr          =>          rs,
        c_addr          =>          reg_dest_mux_out,
        b_data          =>          rs_data,
        c_data          =>          rt_data
);

CPU_signextend_0:       Signextend port map(
        immIn               =>          rt,
        immOut              =>          sign_ex_out
);

CPU_reg_dest_mux:       mux2_1 generic map(4) port map(
        Input1          =>          rt,
        Input2          =>          rd,
        S                           =>          ctrl_reg_dest,
```

```
        Sout            =>          reg_dest_mux_out
);

--------------------------------------------------------------------
-- Execute
--------------------------------------------------------------------
CPU_alu_src_mux:        mux2_1 generic map(16) port map(
        Input1      =>          rt_data,
        Input2      =>          sign_ex_out,
        S                       =>          ctrl_alu_src,
        Sout            =>          alu_src_mux_out
);

CPU_ALU_0:                  ALU_16Bit port map(
        A                       =>          rs_data,
        B                       =>          alu_src_mux_out,
        S                       =>          ctrl_alu_op,
        Sout            =>          alu_result,
```

```
        Cout                    =>              cout

);


-------------------------------------------------------------------------

-- Memory

-------------------------------------------------------------------------


-- TODO 6: Finish implementing the data memory

CPU_MEM_0:Memory port map(

        clk                     =>              clk,

        read_en                 =>              ctrl_mem_read,

        write_en                =>              ctrl_mem_write,

        addr                    =>              alu_result,

        data_in         =>              rt_data,

        data_out                =>              mem_dataout,

        mem_dump    =>              '0'

);
```

```
    ----------------------------------------------------------------------

    -- Write Back

    ----------------------------------------------------------------------


    -- TODO 7: Finish implementing the 3-1 MUX

    slt_input       <=      "000000000000000"&alu_result(15);

    CPU_reg_src_mux:            mux3_1 generic map(16) port map(

        Input1      =>          mem_dataout,

        Input2      =>          alu_result,

        Input3      =>          slt_input,

        S                       =>          ctrl_reg_src,

        Sout                =>  reg_src_mux_out

    );


end Behavioral;
```

Appendix B – Full Adder

```
Library ieee;

Use ieee.std_logic_1164.all;
```

```
Use ieee.std_logic_unsigned.all;


Entity full_adder is

port(
                A       :       in      std_logic;

                B       :       in      std_logic;

                Cin     :       in      std_logic;

                Sout    :       out std_logic;

                Cout:   out     std_logic

);

End;


Architecture behavior of full_adder is


Begin

-- Cin A  B  S  C

--  0  0  0  0  0

--  0  0  1  1  0

--  0  1  0  1  0

--  0  1  1  0  1
```

```
-- 1 0 0 1 0

-- 1 0 1 0 1

-- 1 1 0 0 1

-- 1 1 1 1 1


-- Full adder Logic find the expression in previous ppt


        Sout <= A xor B xor Cin;

        Cout <= (A and B) or ((A Xor B) and Cin);



End;
```

Appendix C – Control

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_unsigned.all;
```

```
Entity Control is
        port(
                op              :       in      std_logic_vector( 3 downto 0);
                alu_op          :       out     std_logic_vector( 1 downto 0);
                alu_src         :       out     std_logic;
                reg_dest                :       out     std_logic;
                reg_load                :       out     std_logic;
                reg_src         :       out     std_logic_vector(1 downto 0);
                mem_read                :       out     std_logic;
                mem_write       :       out     std_logic;
                jump        : out std_logic;
                branch      : out std_logic
                );
End Control;


architecture syn of Control is


begin
        process (op) is
        begin
```

```
case op is

        -- op=0, ADD
        when x"0" =>
                alu_op          <=      "00";
                alu_src         <=      '0';
                reg_dest                <=      '0';
                reg_load                <=      '1';
                reg_src         <=      "01";
                mem_read                <=      '0';
                mem_write       <=      '0';
                jump       <= '0';
                branch     <= '0';


        -- op=1, SUB
        when x"1" =>
                alu_op          <=      "01";
                alu_src         <=      '0';
                reg_dest                <=      '0';
                reg_load                <=      '1';
```

```
                reg_src         <=      "01";
                mem_read        <=      '0';
                mem_write       <=      '0';
                jump    <= '0';
                branch  <= '0';

        -- op=2, AND
        when x"2" =>
                alu_op          <=      "10";
                alu_src         <=      '0';
                reg_dest        <=      '0';
                reg_load        <=      '1';
                reg_src         <=      "01";
                mem_read        <=      '0';
                mem_write       <=      '0';
jump    <= '0';
                branch  <= '0';

        -- op=3, OR
        when x"3" =>
```

```
                alu_op          <=      "11";
                alu_src         <=      '0';
                reg_dest        <=      '0';
                reg_load        <=      '1';
                reg_src         <=      "01";
                mem_read        <=      '0';
                mem_write       <=      '0';
jump    <= '0';
                branch      <= '0';

            -- op=4, ADDi
            when x"4" =>
                alu_op          <=      "00";
                alu_src         <=      '1';
                reg_dest        <=      '0';
                reg_load        <=      '1';
                reg_src         <=      "01";
                mem_read        <=      '0';
                mem_write       <=      '0';
jump    <= '0';
```

```
                branch      <= '0';


            -- op=5, SUBi
            when x"5" =>
                alu_op          <=      "01";
                alu_src         <=      '1';
                reg_dest                <=      '0';
                reg_load                <=      '1';
                reg_src         <=      "01";
                mem_read                <=      '0';
                mem_write       <=      '0';
jump        <= '0';
                branch      <= '0';


            -- op=8, LW
            when x"8" =>
                alu_op          <=      "00";
                alu_src         <=      '1';
                reg_dest                <=      '0';
                reg_load                <=      '1';
```

```
                reg_src         <=      "00";
                mem_read        <=      '1';
                mem_write       <=      '0';
        jump    <= '0';
                branch  <= '0';

            -- op=C, SW
            when x"C" =>
                alu_op          <=      "00";
                alu_src         <=      '1';
                reg_dest        <=      '1';
                reg_load        <=      '0';
                reg_src         <=      "01";
                mem_read        <=      '0';
                mem_write       <=      '1';
        jump    <= '0';
                branch  <= '0';

            -- op=7, SLT
            when x"7" =>
```

```
            alu_op          <=      "01";
            alu_src         <=      '0';
            reg_dest                <=      '0';
            reg_load                <=      '1';
            reg_src         <=      "10";
            mem_read                <=      '0';
            mem_write       <=      '0';
        jump        <= '0';
            branch      <= '0';

-- op=9, BNE
when x"9" =>
            alu_op          <=      "01";
            alu_src         <=      '0';
            reg_dest                <=      '1';
            reg_load                <=      '0';
            reg_src         <=      "01";
            mem_read                <=      '0';
            mem_write       <=      '0';
        jump        <= '0';
```

```
                branch      <= '1';


            -- op=B, JMP
            when x"B" =>
                alu_op          <=      "00";
                alu_src         <=      '0';
                reg_dest                <=      '0';
                reg_load                <=      '0';
                reg_src         <=      "00";
                mem_read                <=      '0';
                mem_write       <=      '0';
jump        <= '1';
                branch      <= '0';


            when others =>
                alu_op          <=      "00";
                alu_src         <=      '0';
                reg_dest                <=      '0';
                reg_load                <=      '0';
                reg_src         <=      "01";
```

```
                    mem_read        <=      '0';
                    mem_write       <=      '0';
                    jump <= '0';
                    branch <= '0';

                end case;
        end process;
end syn;
```

Appendix D – Memory

```
-- This file contains the description of a simple, single-port memory.
--
-- * The data_out port updates asynchronously when read_en is high.
-- * The memory is updated synchronously (on a rising clock edge) from
--   data_in when write_en is high.
--
-- This memory can be initialized from an input file, and can dump its
-- contents to an output file.  To use this, you must assign the INPUT and
-- OUTPUT generics when instantiating a memory component, e.g.:
```

```
--
--     data_mem : entity work.memory
--         generic map (
--             INPUT => "in_data.txt",
--             OUTPUT => "out_data.txt")
--         port map ( ... );
--
-- The input file must have 33 lines, 8 bits per line.  The output file is
-- written in the same format, whenever the mem_dump input goes high.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;



entity Memory is
  generic (
      INPUT : string := "in.txt";
      OUTPUT : string := "out.txt"
```

```
  );

   port (

      clk : in std_logic;

      read_en : in std_logic;

      write_en : in std_logic;

      addr : in std_logic_vector(15 downto 0);

      data_in : in std_logic_vector(15 downto 0);

      data_out : out std_logic_vector(15 downto 0);

      mem_dump : in std_logic := '0'

   );

end entity Memory;



architecture beh of Memory is

   type mem_type is array (0  to  33) of std_logic_vector(7  downto 0);


   impure function init_mem return mem_type is

      file in_file : text;

      variable in_line : line;

      variable byte : bit_vector(7  downto 0);
```

```
        variable mem : mem_type;
    begin
        if INPUT'length = 0 then
            return mem;
        end if;

        file_open(in_file, INPUT, READ_MODE);
        for row in mem_type'range loop
            readline(in_file, in_line);
            read(in_line, byte);
            mem(row) := to_stdlogicvector(byte);
        end loop;

        return mem;
    end function init_mem;

    signal mem : mem_type := init_mem;

--    procedure dump_mem is
--        file out_file : text;
```

```
--      variable out_line : line;
--      variable byte : bit_vector(7  downto 0);
--
--  begin
--      if OUTPUT'length = 0 then
--          report "The mem_dump port was triggered, but no file name was";
--          report "given.  Specify a file name for the generic OUTPUT in";
--          report "the entity declaration to enable memory dump.";
--          return;
--      end if;
--
--      file_open(out_file, OUTPUT, WRITE_MODE);
--      for row in mem_type'range loop
--          byte := to_bitvector(mem(row));
--          write(out_line, byte);
--          writeline(out_file, out_line);
--      end loop;
--  end procedure dump_mem;
```

```
begin

    read_p : process (read_en, mem, addr) is

    begin

        if read_en = '1' then

            data_out(7  downto 0) <= mem(conv_integer(addr(7 downto 0)));

            data_out(15 downto 8) <= mem(conv_integer(addr(7 downto 0))+1);

        end if;

    end process read_p;



    write_p : process (clk) is

    begin

        if rising_edge(clk) and write_en = '1' then

            mem(conv_integer(addr(7 downto 0))) <= data_in(7  downto 0);

            mem(conv_integer(addr(7 downto 0))+1) <= data_in(15 downto 8);

        end if;

    end process write_p;



--    mem_dump_p : process(mem_dump) is

--    begin

--        if rising_edge(mem_dump) then
```

```
--            dump_mem;

--        end if;

--    end process mem_dump_p;

end architecture beh;
```

Appendix E – 2-to-1 Multiplexor

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;




entity mux2_1 is

  generic (WIDTH : positive:=16);

          port(

                  Input1          :       in              std_logic_vector(WIDTH-1        downto 0);

                  Input2          :       in              std_logic_vector(WIDTH-1        downto 0);

                  S                               :       in                      std_logic;

                  Sout                    :       out     std_logic_vector(WIDTH-1        downto 0));

end mux2_1;
```

```
architecture Behavioral of mux2_1 is



begin

        Sout <=Input1 when S='0' else

                                Input2 when S='1';

end Behavioral;
```

Appendix F – 3-to-1 Multiplexor

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;




entity mux3_1 is

  generic (WIDTH : positive:=16);

        port(

                Input1          :       in              std_logic_vector(WIDTH-1        downto 0);

                Input2          :       in              std_logic_vector(WIDTH-1        downto 0);

                Input3          :       in              std_logic_vector(WIDTH-1        downto 0);
```

```
        S                   :       in      std_logic_vector(1
    downto 0);
        Sout                :       out     std_logic_vector(WIDTH-1    downto 0));
end mux3_1;

architecture Behavioral of mux3_1 is

begin
    Sout <=Input1 when S="00" else
                    Input2 when S="01" else
                    Input3 when S="10" else
                    Input3 when S="11";
end Behavioral;
```

Appendix G – 4-Bit 2-to-1 Multiplexor

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.std_logic_unsigned.all;
```

```
entity MUX21_4Bit is

port(   Input1          :       in      std_logic_vector(3 downto 0);

        Input2          :       in      std_logic_vector(3 downto 0);

        S               :       in      std_logic;

        Sout            :       out     std_logic_vector(3 downto 0));

end MUX21_4Bit;


architecture Behavioral of MUX21_4Bit is


Begin


    Sout <=Input1 when S='0' else

                Input2 when S='1';


end Behavioral;
```

Appendix H – 16-Bit 2-to-1 Multiplexor

```
Library ieee;

Use ieee.std_logic_1164.all;
```

```
Use ieee.std_logic_unsigned.all;


entity MUX21_16Bit is
port(   Input1       :       in      std_logic_vector(15 downto 0);
        Input2       :       in      std_logic_vector(15 downto 0);
        S            :       in      std_logic;
        Sout         :       out     std_logic_vector(15 downto 0));
end MUX21_16Bit;


architecture Behavioral of MUX21_16Bit is

Begin

    Sout <=Input1 when S='0' else
                Input2 when S='1';

end Behavioral;
```

Appendix I – 2-Bit 3-to-1 Multiplexor

```
Library ieee;
```

```
Use ieee.std_logic_1164.all;

Use ieee.std_logic_unsigned.all;



entity MUX31 is

port(   Input1          :       in      std_logic;

                Input2          :       in      std_logic;

                Input3          :       in      std_logic;

                S               :       in      std_logic_vector(1 downto 0);

                Sout            :       out     std_logic);

end MUX31;



architecture Behavioral of MUX31 is



Begin

--MUX31 Logic (using When - ElSE structure)

        Sout <=Input1 when S="00" else

                                Input1 when S="01" else

                                Input2 when S="10" else

                                Input3 when S="11";
```

```
end Behavioral;
```

Appendix J – OR Gate

```
Library ieee;

Use ieee.std_logic_1164.all;

Use ieee.std_logic_unsigned.all;


Entity or_gate is

port(
        In1         :       in      std_logic;
        In2         :       in      std_logic;
        Sout    :       out     std_logic
);

End;


Architecture behavior of or_gate is


Begin
```

```
-- In1  In2  Sout

--  0    0    0

--  0    1    1

--  1    0    1

--  1    1    1


-- or gate logic

        Sout <= In1 or In2;

End;
```

Appendix K – Registers

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_unsigned.all;


Entity Registers is

        port(

                clk     :       in      std_logic;

                clear   :       in  std_logic;
```

```
        a_addr: in          std_logic_vector( 3 downto 0);
        a_data: in          std_logic_vector(15 downto 0);
        load    :           in          std_logic;

        b_addr: in          std_logic_vector( 3 downto 0);
        c_addr: in          std_logic_vector( 3 downto 0);

        b_data: out std_logic_vector(15 downto 0);
        c_data: out std_logic_vector(15 downto 0)
        );
End Registers;

architecture syn of Registers is
    type ram_type is array (15 downto 0) of std_logic_vector(15 downto 0);
    signal REG      :       ram_type;
begin
    process(clk, load, clear)
    begin
        if (clear = '0') then
```

```
REG(0)<= x"0000";

REG(1)<= x"0000";

REG(2)<= x"0000";

REG(3)<= x"0000";


REG(4)<= x"0000";

REG(5)<= x"0000";

REG(6)<= x"0000";

REG(7)<= x"0000";


REG(8)<= x"0000";

REG(9)<= x"0000";

REG(10)          <= x"0000";

REG(11)          <= x"0000";


REG(12)          <= x"0000";

REG(13)          <= x"0000";

REG(14)          <= x"0000";

REG(15)          <= x"0000";

elsif (clk'event and clk='1') then
```

```
                if (load = '1') then
                        REG(conv_integer(a_addr)) <= a_data;
                end if;
        end if;
        REG(0)<= x"0000";
        REG(1)<= x"0001";
    end process;
    b_data <= REG(conv_integer(b_addr));
    c_data <= REG(conv_integer(c_addr));
end syn;


--          if (clear = '0') then
--              for i in 15 downto 0 loop
--                  REG(i) <= x"0000";
--              end loop;
--          elsif (rising_edge(clk) and load = '1') then
--              REG(conv_integer(a_addr)) <= a_data;
--          end if;
```

Appendix L – Sign Extend

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_unsigned.all;



Entity Signextend is

        port(

                immIn           :       in      std_logic_vector( 3 downto 0);

                immOut                  :       out     std_logic_vector(15 downto 0)

                );

End Signextend;



architecture syn of Signextend is



begin

        immOut(15)      <=      immIn(3);

        immOut(14)      <=      immIn(3);

        immOut(13)      <=      immIn(3);

        immOut(12)      <=      immIn(3);
```

```
        immOut(11)    <=    immIn(3);
        immOut(10)    <=    immIn(3);
        immOut(9)     <=    immIn(3);
        immOut(8)     <=    immIn(3);
        immOut(7)     <=    immIn(3);
        immOut(6)     <=    immIn(3);
        immOut(5)     <=    immIn(3);
        immOut(4)     <=    immIn(3);
        immOut(3)     <=    immIn(3);
        immOut(2)     <=    immIn(2);
        immOut(1)     <=    immIn(1);
        immOut(0)     <=    immIn(0);

end syn;
```

Appendix M – 16-Bit ALU

```
Library ieee;

Use ieee.std_logic_1164.all;

Use ieee.std_logic_unsigned.all;
```

```
Entity ALU_16Bit is

port(
        A       :   in      std_logic_vector(15 downto 0);
        B       :   in      std_logic_vector(15 downto 0);
        S       :   in      std_logic_vector(1 downto 0);
        Sout    :   out     std_logic_vector(15 downto 0);
        Cout    :   out     std_logic
);
End;


Architecture behavior of ALU_16Bit is
    COMPONENT ALU
    port(
        A       :   in      std_logic;
        B       :   in      std_logic;
        Cin     :   in      std_logic;
        S       :   in      std_logic_vector(1 downto 0);
        Sout    :   out     std_logic;
        Cout    :   out     std_logic
    );
```

```
    END COMPONENT;

    signal Carry        :   std_logic_vector(14 downto 0);

Begin

    alu00 : ALU port map(A(0), B(0), S(0), S, Sout(0), Carry(0));
    alu01 : ALU port map(A(1), B(1), Carry(0), S, Sout(1), Carry(1));
    alu02 : ALU port map(A(2), B(2), Carry(1), S, Sout(2), Carry(2));
    alu03 : ALU port map(A(3), B(3), Carry(2), S, Sout(3), Carry(3));

    alu04 : ALU port map(A(4), B(4), Carry(3), S, Sout(4), Carry(4));
    alu05 : ALU port map(A(5), B(5), Carry(4), S, Sout(5), Carry(5));
    alu06 : ALU port map(A(6), B(6), Carry(5), S, Sout(6), Carry(6));
    alu07 : ALU port map(A(7), B(7), Carry(6), S, Sout(7), Carry(7));
```

```
alu08 : ALU port map(A(8), B(8), Carry(7), S, Sout(8),
Carry(8));
alu09 : ALU port map(A(9), B(9), Carry(8), S, Sout(9),
Carry(9));
alu10 : ALU port map(A(10), B(10), Carry(9), S, Sout(10),
Carry(10));
alu11 : ALU port map(A(11), B(11), Carry(10), S, Sout(11),
Carry(11));

alu12 : ALU port map(A(12), B(12), Carry(11), S, Sout(12),
Carry(12));
alu13 : ALU port map(A(13), B(13), Carry(12), S, Sout(13),
Carry(13));
alu14 : ALU port map(A(14), B(14), Carry(13), S, Sout(14),
Carry(14));
alu15 : ALU port map(A(15), B(15), Carry(14), S, Sout(15),
Cout);
End;
```

Appendix N – 4-Bit ALU

```
Library ieee;

Use ieee.std_logic_1164.all;

Use ieee.std_logic_unsigned.all;



Entity ALU is

port(

                A       :       in      std_logic;

                B       :       in      std_logic;

                Cin     :       in      std_logic;

                S       :       in      std_logic_vector(1 downto 0);

                Sout:   out std_logic;

                Cout:   out     std_logic

);

End;



Architecture behavior of ALU is

   COMPONENT full_adder

        port(
```

```
        A       :       in      std_logic;
        B       :       in      std_logic;
        Cin     :       in      std_logic;
        Sout    :       out std_logic;
        Cout:   out     std_logic
        );
END COMPONENT;

    component MUX31
    port(
        Input1          :       in      std_logic;
        Input2          :       in      std_logic;
        Input3          :       in      std_logic;
        S               :       in      std_logic_vector(1 downto 0);
        Sout            :       out     std_logic
        );
    end component;

    component and_gate
    port(
```

```
        In1             :       in      std_logic;
        In2             :       in      std_logic;
        Sout    :       out     std_logic
        );
    end component;

    component or_gate
    port(
        In1             :       in      std_logic;
        In2             :       in      std_logic;
        Sout    :       out     std_logic
        );
    end component;

--Signals Defination
    signal Sout_full_adder  :       std_logic;
    signal Sout_or_gate             :       std_logic;
    signal Sout_and_gate            :       std_logic;
    signal NB                                       :       std_logic;
Begin
```

```
NB <= B xor S(0);


C1:full_adder    port map(
        A               =>      A,
        B               =>      NB,
        Cin     =>      Cin,
        Sout    =>      Sout_full_adder,
        Cout    =>      Cout
);


C2:and_gate     port map(
        In1     =>      A,
        In2     => B,
        Sout    => Sout_and_gate
);


C3:or_gate              port map(
        In1     =>      A,
        In2     =>      B,
```

```
        Sout    =>      Sout_or_gate
    );


    C4:MUX31            port map(
        Input1  =>      Sout_full_adder,
        Input2  =>      Sout_and_gate,
        Input3  =>      Sout_or_gate,
        S               =>      S,
        Sout            =>      Sout
    );


End;
```

Appendix O – AND Gate

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.std_logic_unsigned.all;


Entity and_gate is
```

```
port(
        In1     :   in  std_logic;
        In2     :   in  std_logic;
        Sout    :   out std_logic
);
End;

Architecture behavior of and_gate is

Begin
-- In1  In2  Sout
--  0    0    0
--  0    1    0
--  1    0    0
--  1    1    1

-- and gate logic
    Sout <= In1 and In2;

End;
```

Appendix P – Zero Detector

```
----------------------------------------------------------------------------------

-- Company:

-- Engineer:

--

-- Create Date: 04/03/2023 12:26:28 PM

-- Design Name:

-- Module Name: Zero_Detector - Behavioral

-- Project Name:

-- Target Devices:

-- Tool Versions:

-- Description:

--

-- Dependencies:

--

-- Revision:

-- Revision 0.01 - File Created

-- Additional Comments:
```

```
--
----------------------------------------------------------------------------------



library IEEE;

use IEEE.STD_LOGIC_1164.ALL;



-- Uncomment the following library declaration if using

-- arithmetic functions with Signed or Unsigned values

--use IEEE.NUMERIC_STD.ALL;



-- Uncomment the following library declaration if instantiating

-- any Xilinx leaf cells in this code.

--library UNISIM;

--use UNISIM.VComponents.all;



library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

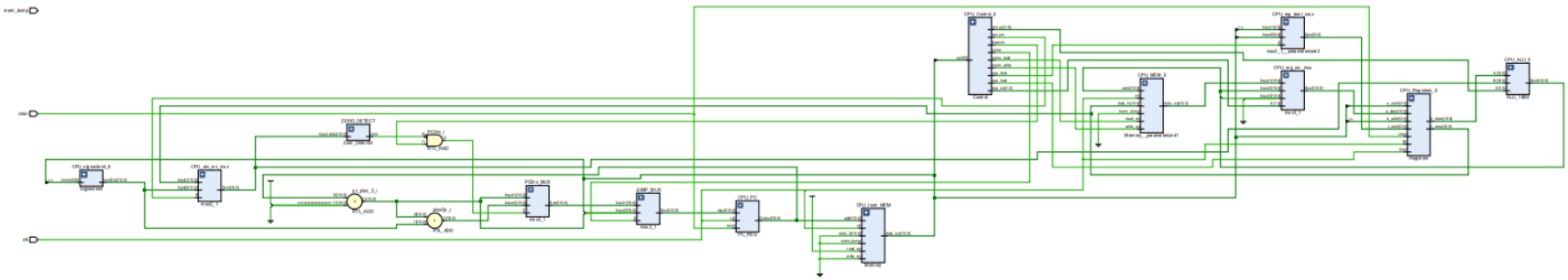
```
entity Zero_Detector is
    Port (
        input_data : in STD_LOGIC_VECTOR (15 downto 0);
        zero : out STD_LOGIC
    );
end entity Zero_Detector;

architecture Behavioral of Zero_Detector is
begin
    process (input_data)
    begin
        if input_data = "0000000000000000" then
            zero <= '1';
        else
            zero <= '0';
        end if;
    end process;
end architecture Behavioral;
```

Appendix Q – PC Register

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_arith.all;

use ieee.std_logic_unsigned.all;

use ieee.numeric_std.all;



entity PC_REG is

	port(

		clk : in std_logic;

		reset : in std_logic;

		Input : in std_logic_vector(15 downto 0);

		Output : out std_logic_vector(15 downto 0)

	);

end PC_REG;



architecture Behavioral of PC_REG is

begin
```

```
        process(reset, clk)
        begin

--To do: write the code; if reset is zero, the output will be zero. Otherwise, at the rising edge of the clock, the input will be transfered to the output.

          if reset = '0' then
        output <= x"0000";
          elsif (rising_edge(clk) and reset='1') then
                  output <= input;
    end if;
        end process;
end Behavioral;
```

Appendix R – Block Diagram



Appendix S – Test Bench

```
library ieee;

use ieee.std_logic_1164.all;




entity CPU_3380_Test is

end entity CPU_3380_Test;




architecture mixed of CPU_3380_Test is

    constant tick : time := 100 ns;

            constant RUN_TIME : integer := 18;

    signal reset, clock : std_logic;

            signal mem_dump : std_logic := '0';

    --signal instruction : std_logic_vector(0 to 15);

begin

    uut : entity work.CPU_3380

        port map(

            clk                     => clock,
```

```
        clear      => reset,
                                mem_dump    => mem_dump
    );

  driver : process is
  begin
    -- reset the system
    reset <= '0';
            wait for 50 ns;
    reset <= '1';

                    for i in  1 to RUN_TIME loop
                            wait for tick;
                    end loop;
--                  addi r3, r0, 5
--                  addi r4, r0, 2
--                  slt r11, r3, r4
--                  sw r3, 0(r0)
--                  sw r4, 4(r0)
--                  addi r6, r0, 4
```

```
--                          lw r7, 0(r6)
--                          lw r8, 0(r0)
--                          add r9, r7, r8
--                          slt r10, r0, r1
--                          slt r10, r1, r0
--                          or r5, r10, r9
--                          subi r10, r5, 7
--                          sub r11, r10, r7
--                          sw r11, 5(r8)
        wait;
    end process driver;


    clock_p : process is
    begin
        for i in 0 to 50 loop
            clock <= '1'; wait for tick/2;
            clock <= '0'; wait for tick/2;
        end loop;
        wait;
    end process clock_p;
```

end architecture mixed;