ECGR 4146/5146: Intro to VHDL

Project 1: Microcontroller Design

By: Logan Mayhew, Danny Murray, Roy Helms, John Humphries, Sreng Heng

Objective:

The objective of Project 1 is to design and implement a microcontroller in VHDL that follows the explanation and process described in the project description. The microcontroller was to utilize the design of an embedded 8-bit RISC microcontroller core. The microcontroller would use the theory of Programmable State Machines (PSMs), which allows for a set of instructions defined by the user to be loaded onto the microcontroller, held in memory, and encoded into machine code for the process desired to take place.

Design Specifications:

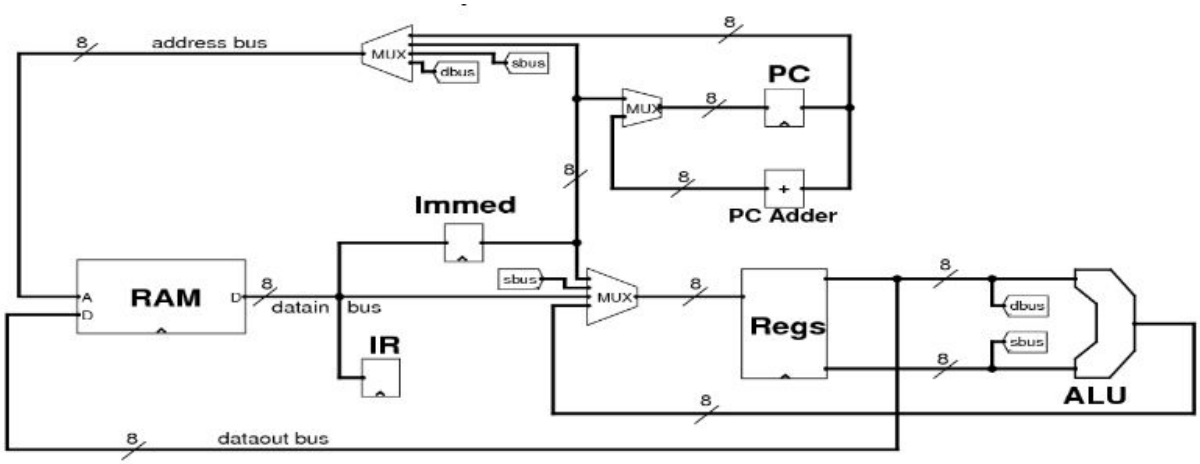

Figure 1: Microcontroller Block Diagram [1]

1. The CPU has an 8-bit data bus and an 8-bit address bus, so it can support 256 bytes of memory to hold both instructions and data.
2. Internally, there are four 8-bit registers, R0 to R3, an Instruction Register, IR, the Program Counter, PC,and an 8-bit immediate register that holds immediate values.
3. The ALU performs the four operations AND, OR, ADD, SUB, on two 8 - bit values, and supports signed ADDs and SUBs.
4. The CPU is a load/store architecture: data must be brought into registers for manipulation, as the ALU only reads from and writes back to the registers.

Instruction:

op1	op2	Mnemonic	Purpose
00	00	AND Rd, Rs	Rd = Rd AND Rs
00	01	OR Rd, Rs	Rd = Rd OR Rs
00	10	ADD Rd, Rs	Rd = Rd + Rs
00	11	SUB Rd, Rs	Rd = Rd - Rs
01	00	LW Rd, (Rs)	Rd = Mem[Rs]
01	01	SW Rd, (Rs)	Mem[Rs] = Rd
01	10	MOV Rd, Rs	Rd = Rs
01	11	NOP	Do nothing
10	00	JEQ Rd, immed	PC = immed if Rd == 0
10	01	JNE Rd, immed	PC = immed if Rd != 0
10	10	JGT Rd, immed	PC = immed if Rd > 0
10	11	JLT Rd, immed	PC = immed if Rd < 0
11	00	LW Rd, immed	Rd = Mem[immed]
11	01	SW Rd, immed	Mem[immed] = Rd

Op1	Op2	Rd	Rs
2	2	2	2

Op1	Op2	Rd	Rs	Immediate
2	2	2	2	8

Figure 2: Instruction Set Architecture (ISA) [1]

The first piece of the microcontroller is the definition of an Instruction Set Architecture (ISA). This is the main set of instructions for the microcontroller. Depending on the input instruction, the opcode of that specific instruction explains the process that the microcontroller needs to complete. Starting from the most significant bit (MSB), the first 4 bits of each instruction are the opcode, op1 and op2, which specifies the necessary steps the microcontroller will be taking during the following clock cycles. The second 4 bits are the destination register and the source register respectively. The final 8 bits of each instruction describe the value that needs to be loaded into the immediate register.

As a design choice the team decided to use the basic functionality of accepting instructions and converting them to machine code, which would be conducted by a microprocessor. In doing so, the team noticed that, based off of the main truth table explaining the instructions, the immediate register was only needed when the MSB of the instruction was high. Therefore, to create the specified RISC architecture, which would mean having only one set length of each instruction, when an instruction is loaded into the microcontroller with this condition, then the next incoming instruction

would be the value of that needs to be loaded into the immediate register instead of a new instruction the microcontroller should compute. This allowed the team to generate code that would mimic the test program specified in the project description.

Program Counter:

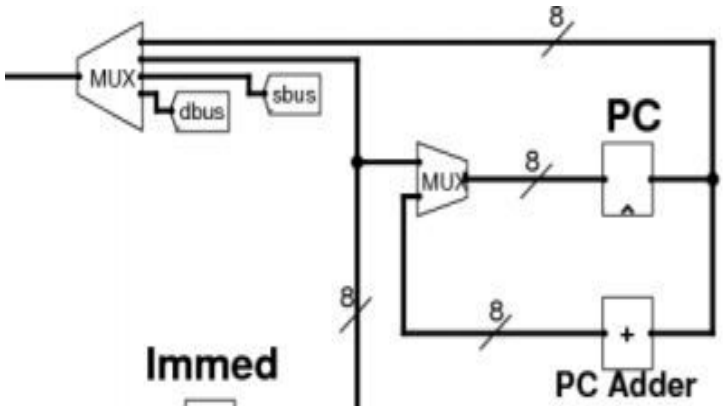

Figure 3: Program Counter Component [1]

The program counter (PC) is the driving force behind the Microcontroller as it specifies which instruction are to be loaded out of memory next. It consists of four components, a PC register, and a PC Adder. A 2-to-1 multiplexer (mux) is used to either selected the immediate register or PC + one to be loaded into the the PC register. The final component is a 4-to-1 mux that decides which signal is going to be passed along the address bus. The code for the multiplexers and the register files are shown later in the components breakdowns of the report. The below code shows the installed components that are used in the PC, shown in figure 4, and followed by the code of the PC component, shown in figure 5.

```
component PC_reg
    Port ( clk : in STD_LOGIC;
           enable : in STD_LOGIC;
           PC_in : in STD_LOGIC_VECTOR (7 downto 0);
           PC_out : out STD_LOGIC_VECTOR (7 downto 0):="00000000");
end component;

component mux_2_1
    Port ( in_0 : in STD_LOGIC_VECTOR (7 downto 0);
           in_1 : in std_logic_vector (7 downto 0);
           out_mux : out STD_LOGIC_vector (7 downto 0);
           op : in STD_LOGIC);
end component;

component mux_4_1
    Port ( in_0 : in STD_LOGIC_VECTOR (7 downto 0);
           in_1 : in STD_LOGIC_VECTOR (7 downto 0);
           in_2 : in STD_LOGIC_VECTOR (7 downto 0);
           in_3 : in STD_LOGIC_VECTOR (7 downto 0);
           op : in std_logic_vector (1 downto 0);
           out_mux : out STD_LOGIC_VECTOR (7 downto 0));
end component;
```

Figure 4: Components used in the PC component

```
 signal mux_PC,temp_PC_out,PC_plus,temp_address : std_logic_vector (7 downto 0);

begin
PC_mux : mux_2_1
    port map (in_0=>Immed_in,in_1=>PC_plus,out_mux=>mux_PC,op=>pcsel);
REGPC : PC_reg
    port map (clk=>clk,enable=>pcload,PC_in=>mux_PC,PC_out=>temp_PC_out);
AddrMUX : mux_4_1
    port map (in_0=>temp_PC_out,in_1=>Immed_in,in_2=>sbus,in_3=>dbus,
              op=>addrsel,out_mux=>temp_address);

PC_plus <= std_logic_vector(unsigned(temp_PC_out) + "00000001");
address <= temp_address;
```

Figure 5: PC Component Code

The following image in figure 6 is the test bench code used to test the overall PC component. It first starts by declaring the signals for the inputs that are not changed in PC. It then selects address “0x00”, which is the PC components value, and loads the PC plus one into the register for two cycles with pcsel set to ‘1’. It then switches the pcsel to ‘0’ and loads the immediate. It then changes the address select to “01’, “10”, and “11” to show the immediate, sbus, and dbus being passed. Following the test bench code is the waveform simulation of the code shown in figure 7.

```
process
    begin
        Immed_in <= "00001000";
        sbus <= "00000100";
        dbus <= "00000010";
        addrsel <= "00";
        pcsel <= '1';
        pcload <='1';
        wait for 100ns;
        wait for 100ns;
        pcsel <= '0';
        wait for 100ns;
        wait for 100ns;
        pcload <='0';
        addrsel <= "01";
        wait for 100ns;
        addrsel <= "10";
        wait for 100ns;
        addrsel <= "11";
        wait for 100ns;
end process;
```

Figure 6: PC component test bench

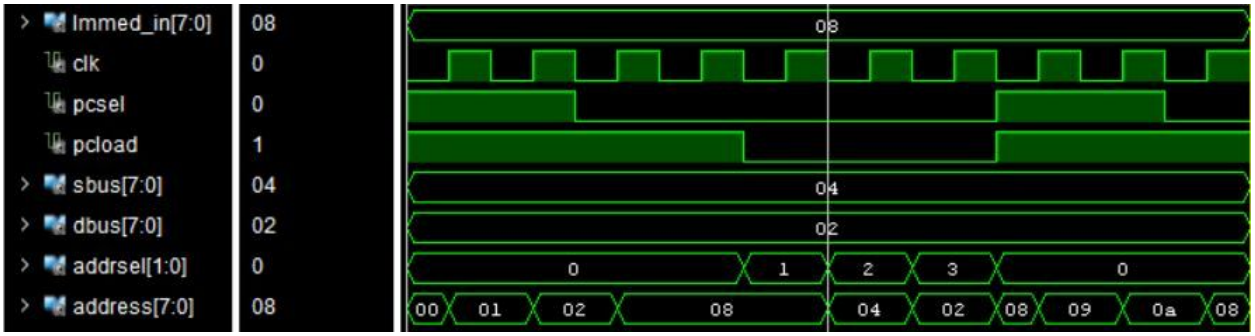

Figure 7: PC component simulation

Memory:

Figure 8: 256 bytes of Random Access Memory (RAM) [1]

The team chose to make memory an array of std_logic_vectors of size eight. The memory is a synchronous device that has a toggle bit for the desired functionality. There is a readwrite bit that must be high in order to read from the memory. If the readwrite bit is low, then the memory is writable. To specify the index of the array, the controlling part must specify an address between “0x00” and “0xFF”. Finally, there is a reset bit that clears the dataout buffer.

```
18      SHARED VARIABLE ADDR : INTEGER RANGE 0 TO 255;
19      BEGIN
20          PROCESS(ADDRESS, DATAIN, readwrite, clk, rst)
21          BEGIN
22              IF(rising_edge(clk)) THEN
23                  IF(rst='1') THEN
24                      dataout <= "00000000";
25                  ELSE
26                      ADDR:=CONV_INTEGER(ADDRESS);
27                      IF(readwrite='1')THEN
28                          MEMORY(ADDR)<=datain;
29                      ELSE
30                          dataout<=MEMORY(ADDR);
31                      END IF;
32                  END IF;
33              END IF;
34          END PROCESS;
35  END BEV;
```

Figure 9: The Architecture of RAM.

The core functionality of memory was tested by loading the test program and then reading the contents of that were wrote to the memory. The instruction memory can be offset by modifying the instruction pointer. For the sake of the simulation, address “0x00” was selected.

```
process
begin

    --begin load ISA into memory, takes 2250 ns
    clk <= '0';
    wait for T/2;
    clk <= '1';
    readwrite <= '0';

    instr_loop : for i in 0 to NUM_INSTRUCTIONS - 1 loop
        datain <= ISA(i);
        clk <= '0';
        address <= STD_LOGIC_VECTOR(to_unsigned((CONV_INTEGER(INSTRUCTION_POINTER) + i), address'length));
        wait for T/2;
        clk <= '1';
        wait for T/2;
    end loop instr_loop;
    --end load ISA into memory

    clk <= '0';
    readwrite <= '1';

    --begin ISA load verif
    verif_loop : for j in 0 to NUM_INSTRUCTIONS - 1 loop
        address <= STD_LOGIC_VECTOR(to_unsigned((CONV_INTEGER(INSTRUCTION_POINTER) + j), address'length));
        wait for T/2;
        clk <= '1';
        wait for T/2;
        clk <= '0';
    end loop verif_loop;

    wait;
    --end ISA load verif

end process;
```

Figure 10: The Process Block Used To Populate And Read The Memory.

The waveform produced by program.vhd matched the expected output exactly. After breaking the instructions down into machine code where the immediate values are following the associated instruction, there were 17 instructions. This is shown in figure 11.

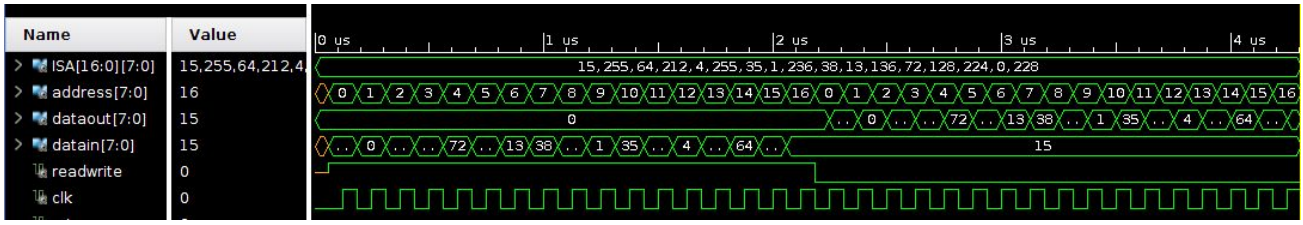

Figure 11: The Waveform Produced By program.vhd

Stage Counter:

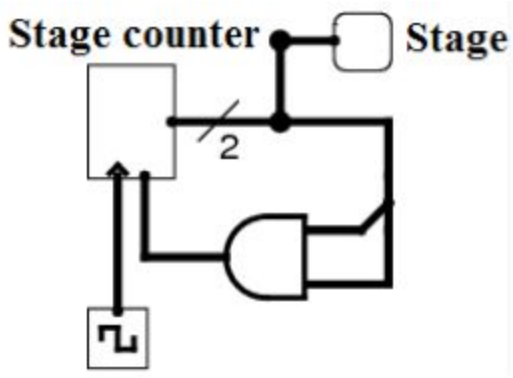

Figure 12: Stage Counter Logic [1]

This logical counter shown in figure 12 is a simple combination of a register file, an AND gate, and a clock signal. The output of the stage counter is a 2-bit bus that allows the Programmable State Machine (PSM) to step through each stage one at a time from integer value ‘0’ through ‘2’ with each clock cycle. Once the stage counter reaches to the integer value ‘2’ the stage counter will restart at the integer value 0. Also a reset function, rst, was added to the stage counter to be used if needed in the final program.

A screenshot of the VHDL program written to design the stage counter can be seen below in figure 13. Sequential logic, which requires a process block, was used to create the functionality of the stage counter as seen below.

```
architecture Behavioral of Stage_Count is
signal stage_next: std_logic_vector(1 downto 0) := "00";
begin

process(clk,rst)
    begin
        if (rst = '1') then
            stage <= "00";
            stage_next <= "00";
        end if;
        if (stage_next = "11") then
            stage_next <= "00";
        elsif (clk'event and clk = '1') then -- could also use "if rising_edge(clk) then ..."
            stage <= stage_next;
            stage_next <= std_logic_vector(unsigned(stage_next) + 1);
        end if;
    end process;

--stage_next <= std_logic_vector(unsigned(stage_next) + 1);

end Behavioral;
```

Figure 13: Stage Counter Program [1]

The stage counter was tested using a generated clocks signal, which used a process block to toggle the clock input high and low. The process started with an initial stage of “00”. The output of the test bench can be seen in the simulation screenshot in figure 14.

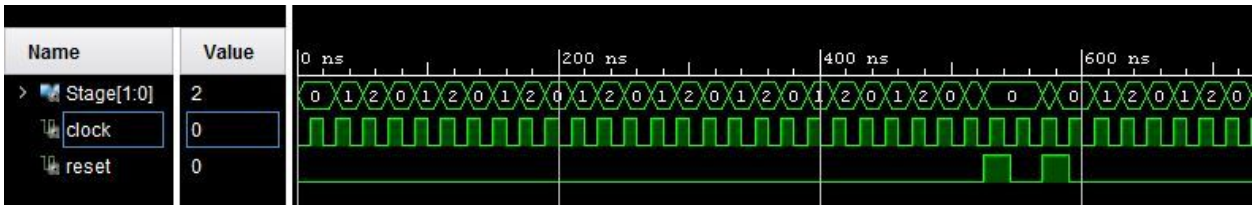

Figure 14: Stage Counter Simulation [1]

Decode Logic:

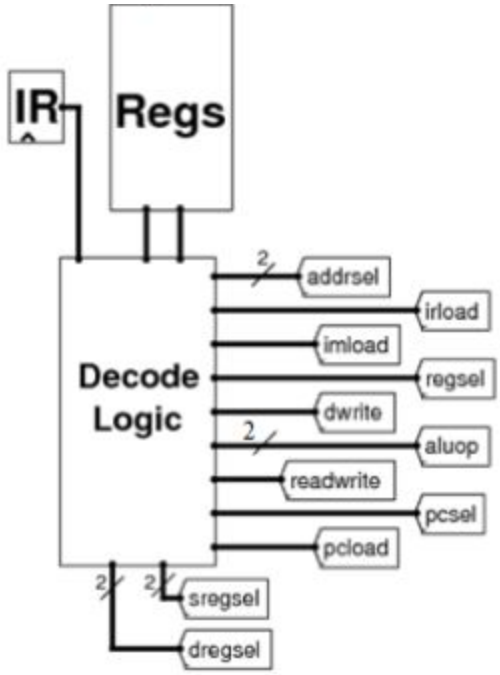

Figure 15: Decode Logic [1]

The decode logic block receives four inputs: the stage from the stage counter, the instruction from the instruction register, and the zero and negative lines from the register file. The block is capable of sending eleven control line outputs. Output logic was implemented in several different ways. The addrsel, regsel, dwrite, and pcload control lines were each instantiated within the decode logic block. The irload, imload, readwrite, and pcsel control lines were handled within a process block. The aluop, sregsel, and dregsel control lines were hard-wired to various inputs to the decode logic block.

```
136     --instantiate Address_Select, Dest_Reg_Write, Register_Select, and pcload
137     Addr_Sel : Address_Select port map(op1=>op1,op2=>op2,stage=>stage,PC=>PC,Rs=>Datain,Rd=>Rd,Immediate=>Immediate,addrsel=>addrsel);
138     Dest_Write : Dest_Reg_Write port map(op1=>op1,op2=>op2,stage=>stage,dwrite=>dwrite);
139     Reg_Sel : Register_Select port map(ALU_out=>ALU_out,Datain=>Datain,Rs=>Rs,Immediate=>PC,op1=>op1,op2=>op2,stage=>stage,regsel=>regsel);
140     PC_Load : PCL port map(zero=>zero,negative=>negative,irbit=>instruction,op2=>op2,stage=>stage,pcload=>pcload);
```

Figure 16: Control Line Component Instantiation

Address selection: The addrsel control line, which is a component inside the decode logic. It receives the operation codes and several constants as input values, as well as the stage count. These are shown in figure 17.

```
48  --Address Select Decode Logic Block
49  component Address_Select
50      port(
51          op1 : in std_logic_vector(1 downto 0);
52          op2 : in std_logic_vector(1 downto 0);
53          stage : in std_logic_vector(1 downto 0);
54          PC : in std_logic_vector(1 downto 0);
55          Rs : in std_logic_vector(1 downto 0);
56          Rd : in std_logic_vector(1 downto 0);
57          Immediate : in std_logic_vector(1 downto 0);
58          addrsel : out std_logic_vector(1 downto 0)
59        );
60  end component;
```

Figure 17: Address Select Component Declaration

Inside the component, a process block implements a mux that selects which constant to pass through. The stage count and operation codes are used in conditional statements that determine the operation of the mux.

```
51  begin
52  operator <= op1&op2;
53    process(operator, stage)
54      begin
55        if (stage = "10") then
56          case operator is
57                when "0100" =>
58                  addrsel <= Rs; -- load the signal between the MUXs with Source Register value
59                when "0101" =>
60                  addrsel <= Rd; -- load the signal between the MUXs with Destination Register value
61                when "1100"|"1101" =>
62                  addrsel <= Immediate; -- load the signal between the MUXs with Immediate value
63                when others =>
64                  addrsel <= PC;-- load the signal between the MUXs with PC value
65              end case;
66        else
67          addrsel <= PC;
68        end if;
69      end process;
```

Figure 18: Address Select Process Block

Register Select: The regsel control line is a component inside the decode logic. It receives the operation codes and several constants as input values, as well as the stage count.

```
72  --Register Select Logic Block
73  component Register_Select
74      port(
75          ALU_out : in std_logic_vector(1 downto 0);
76          Datain : in std_logic_vector(1 downto 0);
77          Rs : in std_logic_vector(1 downto 0);
78          Immediate : in std_logic_vector(1 downto 0);
79          op1 : in std_logic_vector(1 downto 0);
80          op2 : in std_logic_vector(1 downto 0);
81          stage : in std_logic_vector(1 downto 0);
82          regsel : out std_logic_vector(1 downto 0)
83      );
84  end component;
```

Figure 19: Register Select Component Declaration

Inside the component, a process block implements a mux that selects which constant to pass through. The stage count and operation codes are used in conditional statements that determine the operation of the mux.

```
54  begin
55  operator <= op1&op2;
56    process(operator, stage)
57      begin
58        if (stage = "10") then
59          case operator is
60            when "0000"|"0001"|"0010"|"0011" =>
61              regsel <= ALU_out; -- load the signal between the MUXs with ALU output value
62            when "0110" =>
63              regsel <= Rs; -- load the signal between the MUXs with Source Register value
64            when "0100"|"1100" =>
65              regsel <= Datain; -- load the signal between the MUXs with Datain value
66            when others =>
67              regsel <= Immediate; -- load the signal between the MUXs with Immediate value
68            end case;
69        else
70          regsel <= Immediate;
71        end if;
72      end process;
```

Figure 20: Register Select Process Block

Destination Register Select: The dwrite control line is a component inside the decode logic. It receives the operation codes and several constants as input values, as well as the stage count.

```
62  --Destination Register Write Logic Block
63  component Dest_Reg_Write
64      port(
65          op1 : in std_logic_vector(1 downto 0);
66          op2 : in std_logic_vector(1 downto 0);
67          stage : in std_logic_vector(1 downto 0);
68          dwrite : out std_logic
69      );
70  end component;
```

Figure 21: Destination Register Write Component

Inside the component, a process block implements a mux that selects which constant to pass through. The stage count and operation codes are used in conditional statements that determine the operation of the mux.

```
47  begin
48  operator <= op1&op2;
49    process(operator, stage)
50      begin
51        if (stage = "10") then
52          case operator is
53            when "0000"|"0001"|"0010"|"0011"|"0100"|"0110"|"1100"|"1110" => -- values from table
54              dwrite <= '1'; -- set signal between MUXs to HIGH
55            when others =>
56              dwrite <= '0'; -- set signal between MUXs to HIGH
57          end case;
58        else
59          dwrite <= '0';
60        end if;
61      end process;
```

Figure 22: Destination Register Write Process Block

PC Load:

op1op2	Test	Zero	Negative
1000 (JEQ)	Rd == 0	1	x
1001 (JNE)	Rd != 0	0	x
1010 (JGT)	Rd > 0	0	0
1011 (JLT)	Rd < 0	x	1

Figure 23: PC Load Truth Table [1]

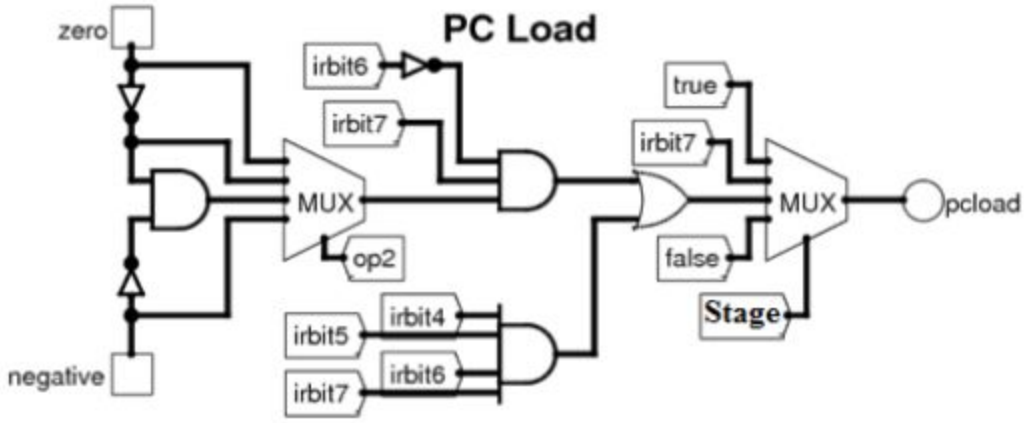

Figure 24: Program Counter Load Logic [1]

PC Load enables PC to be loaded with new values, or disable the loadability. The decision is dependent on the 2-bit stage input and opcode, which is the four most significant bits of irbit. The right multiplexer behaves as the truth-table generated below:

Table 1: Truth-table of the rightmost mux of the PC Load Logic

stage	**pcload**
00 (zero)	true
01 (one)	irbit 7
10 (two)	jump logic
11 (three)	false

The jump logic signal received from and OR logic gate outputs a ‘1’ when there is a successful jump test or a jump always instruction. The jump always instruction, which is

when op1op2 (irbit 4, 5, 6, 7) is equal to “1111”, is implemented using a 4-input AND logic gate. To elaborate jump test logic, first, algebraic expressions were derived.

Jump = irbit 6 AND irbit 7 AND (left MUX output)

Table 2: Truth-table of the leftmost mux of the PC Load Logic

op2	**Left MUX out**
00 (zero)	zero
01 (one)	NOT(zero)
10 (two)	NOT(zero) AND NOT(negative)
11 (three)	negative

The VHDL programming of pcload follows the logic diagram strictly, assuming zero, negative, op1, op2, and stage are input signals. A file creating a 4-to-1 mux is create so that the component can be called in the PC_Load file. The code and simulation for the PC load file is shown in figure 25 and 26 respectively.

```
begin
    m0: mux port map ( a => zero,
                       b => m, --not zero
                       c => n, --not zero(m) and not negative(s)
                       d => negative,
                      s1 => op2(1),
                      s0 => op2(0),
                       y => o); --m0
    m1: mux port map ( a => '1',
                       b => irbit(7),
                       c => p, --q or r
                       d => '0',
                      s1 => stage(1),
                      s0 => stage(0),
                       y => pcload);
    m <= not zero;
    n <= m and p;
    p <= q or r;
    q <= o and irbit(7) and (not irbit(6));
    r <= irbit(4) and irbit(5) and irbit(6) and irbit(7);
    s <= not negative;
end dataflow;
```

Figure 25: PC Load Code

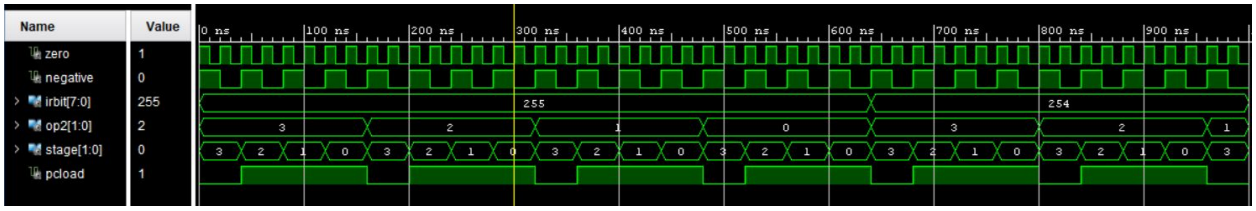

Figure 26: PC Load Simulation

Process Block programming: A process block was used to set the irload, imload, readwrite, and pcsel control lines. Inside the process block, an if statement checks the value of state. Depending on the current value of state, more conditional statements are checked. Once conditions are met, a shared variable corresponding to each control line is set to ‘0’ or ‘1’. These shared variables are then routed to the corresponding output signal. These signals are shown in figure 27.

```
142     process(stage,instruction)
143     begin
144         irbit4_temp := irbit4;
145         irbit5_temp := irbit5;
146         irbit6_temp := irbit6;
147         irbit7_temp := irbit7;
148         --Handle simple control lines based on stage
149         if stage = "00" then
150             --stage 0 decoding
151             pcsel_temp := '1';
152             irload_temp := '1';
153             imload_temp := '0';
154             readwrite_temp := '0';
155         elsif stage = "01" then
156             --stage 1 decoding
157             if instruction(7) = '1' then
158                 pcsel_temp := '1';
159                 irload_temp := '0';
160                 imload_temp := '1';
161                 readwrite_temp := '0';
162             elsif instruction(7) = '0' then
163                 pcsel_temp := '0';
164                 irload_temp := '0';
165                 imload_temp := '0';
166                 readwrite_temp := '0';
167             end if;
168             --stage 2 decoding
169         elsif stage = "10" then
170             pcsel_temp := '0';
171             irload_temp := '0';
172             imload_temp := '0';
173             if instruction(6) = '1' and instruction(5) = '0' and instruction(4) = '1' then
174                 readwrite_temp := '1';
175             else
176                 readwrite_temp := '0';
177             end if;
178         end if;
179         pcsel <= pcsel_temp;
180         irload <= irload_temp;
181         imload <= imload_temp;
182         readwrite <= readwrite_temp;
```

Figure 27: Decode Logic Process Block

Direct Wire: The aluop, sregsel, and dregsel control lines were “hard-wired” to various inputs of the decode logic block. The aluop line is tied to the op2 line from the instruction register input. The sregsel and dregsel lines are tied to bits 3 and 2, and bits 1 and 0 from the instruction register input, respectively.

```
131     --direct wire, not stage based
132     dregsel <= instruction(3 downto 2);
133     sregsel <= instruction(1 downto 0);
134     aluop <= op2;
```

Figure 28: Direct Wire Control Lines

Two simulations were ran on the decode logic. This first simulation tests multiple different values at each stage before incrementing the stage. This is not how the CPU will function, but it allowed for better error checking.

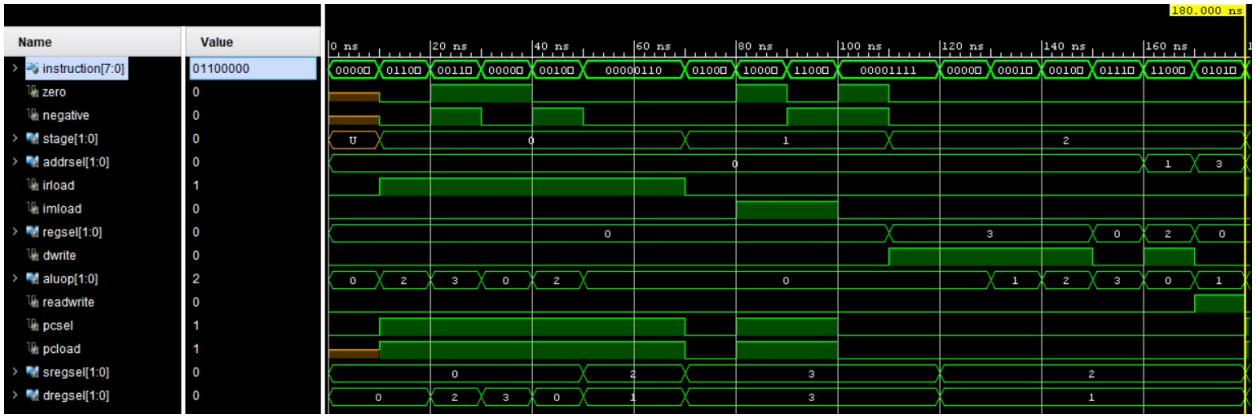

Figure 29: Simulation 1 of Decode Logic

This simulation indicates that the pcsel, pcload, irload, and imload control lines are functioning properly at stages 0 and 1. The next simulation sets an instruction code and then steps through each of the stages. This simulation shows how the code will react when it is implemented in the CPU file.

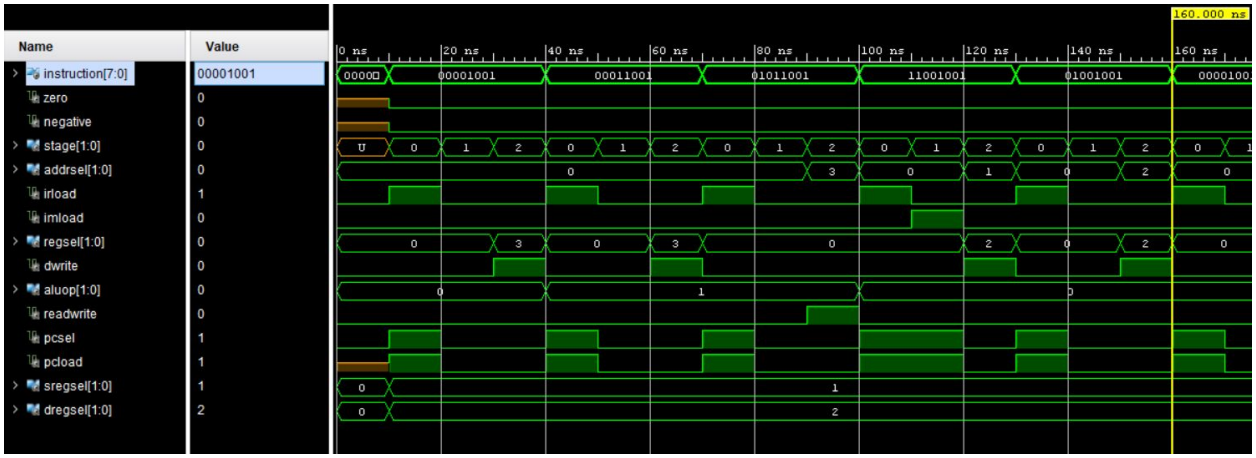

Figure 30: Simulation 2 of Decode Logic

Registers File:

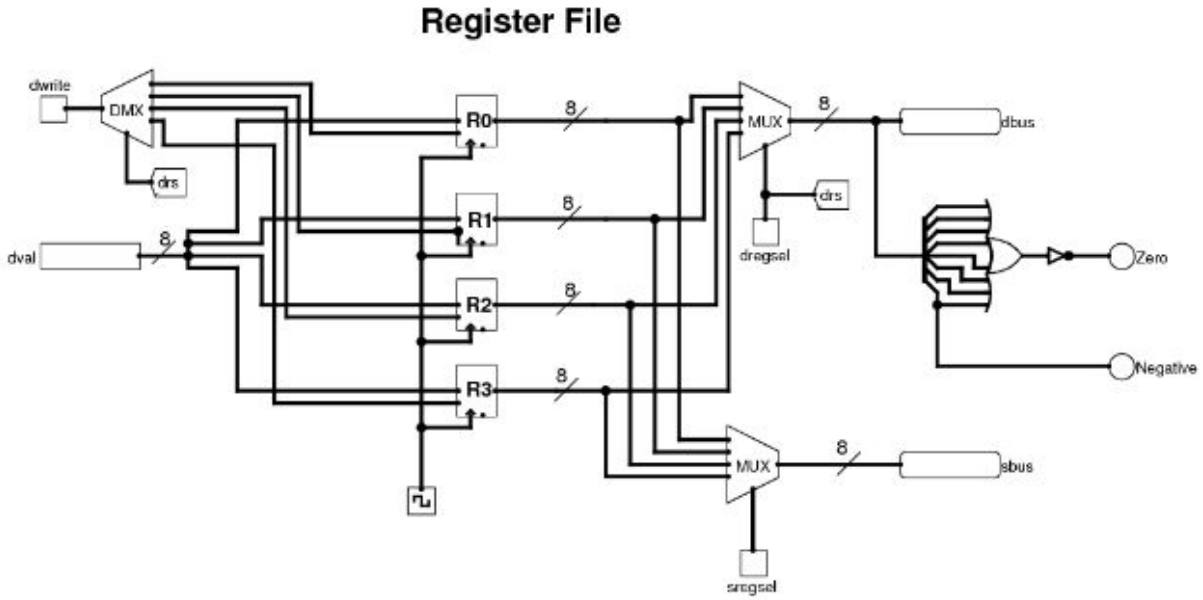

Figure 31: Register File Logic [1]

The register file is a component used to store values that are need during the operation of the microcontroller. Each of the registers are used to hold a specific value that will be used by the microcontroller. The first register R0 is used to hold the pointer for the next number to add. The second register R1 is used to hold the running sum. The third register R2 is used to hold the next number to add to the running sum. The last register R3 is used as a temporary register. The register file is controlled by the decode logic with dwrite, sregsel and dregsel. When dwrite is high dregsel will select the register which will load the value of dval. Dregsel also controls which register value is passed to

dbus. Sregsel controls which register value is passed to sbus. The value passed to dbus will also check the flags for zero and negative used by the decode logic. The components that make up the register File, as seen in figure 29, is a 1-to-4 mux, two 4-to-1 muxes, and four registers. The following code in figures 32-34 shows the components and code used to make the register file block.

```
component bit_reg_8
    Port ( clk : in std_logic;
           reset : in std_logic;
           enable : in std_logic;
           Reg_in : in STD_LOGIC_VECTOR (7 downto 0);
           Reg_out : out STD_LOGIC_VECTOR (7 downto 0));
end component;

component Mux_4_1
    Port ( in_0 : in STD_LOGIC_VECTOR (7 downto 0);
           in_1 : in STD_LOGIC_VECTOR (7 downto 0);
           in_2 : in STD_LOGIC_VECTOR (7 downto 0);
           in_3 : in STD_LOGIC_VECTOR (7 downto 0);
           op : in std_logic_vector (1 downto 0);
           out_mux : out STD_LOGIC_VECTOR (7 downto 0);
           clk : in std_logic);
end component;

component Mux_1_4
    Port ( Mux_in : in STD_LOGIC;
           clk : in std_logic;
           op : in std_logic_vector(1 downto 0);
           out_0 : out STD_LOGIC;
           out_1 : out STD_LOGIC;
           out_2 : out STD_LOGIC;
           out_3 : out STD_LOGIC);
end component;
```

Figure 32: Register File components code

```
signal R0_e,R1_e,R2_e,R3_e : std_logic;
signal R0_out,R1_out,R2_out,R3_out : std_logic_vector (7 downto 0);
signal temp_dbus, temp_sbus : std_logic_vector (7 downto 0);
signal temp_dval : std_logic_vector (7 downto 0);
signal temp_dregsel : std_logic_vector ( 7 downto 0);
signal temp_sregsel : std_logic_vector ( 7 downto 0);
```

Figure 33: Signal used for Register File code

```
temp_dval <= dval;
DMX1 : Mux_1_4
    port map (Mux_in=>dwrite,clk=>clk,op=>dregsel,out_0=>R0_e,out_1=>R1_e,out_2=>R2_e,out_3=>R3_e);

REG0 : bit_reg_8
    port map (clk=>clk,reset=>reset,enable=>R0_e,Reg_in=>temp_dval,Reg_out=>R0_out);
REG1 : bit_reg_8
    port map (clk=>clk,reset=>reset,enable=>R1_e,Reg_in=>temp_dval,Reg_out=>R1_out);
REG2 : bit_reg_8
    port map (clk=>clk,reset=>reset,enable=>R2_e,Reg_in=>temp_dval,Reg_out=>R2_out);
REG3 : bit_reg_8
    port map (clk=>clk,reset=>reset,enable=>R3_e,Reg_in=>temp_dval,Reg_out=>R3_out);

DbusMux : Mux_4_1
    port map (in_0=>R0_out,in_1=>R1_out,in_2=>R2_out,in_3=>R3_out,
            op=>dregsel,out_mux=>temp_dbus,clk=>clk);
SbusMux : Mux_4_1
    port map (in_0=>R0_out,in_1=>R1_out,in_2=>R2_out,in_3=>R3_out,
            op=>sregsel,out_mux=>temp_sbus,clk=>clk);

Zero <= NOT(temp_dbus(0) OR temp_dbus(1) OR temp_dbus(2) OR temp_dbus(3) OR temp_dbus(4)
        OR temp_dbus(5) OR temp_dbus(6) OR temp_dbus(7));

Negative <= temp_dbus(7);

dbus <= temp_dbus;
sbus <= temp_sbus;
```

Figure 34: Register File Code

The following code is the test bench, which is used to check the register files functionality. The test bench first starts by loading values into each register. It then disables dwrite and uses sregsel by going through each register making sure that the value was loaded. It then stays at R3 and puts a new value on dval. Then it switches dwrite to ‘1’ to make sure the mux updates on the value change. Following the test bench in figure 35 is the simulation waveform of the testbench show in figure 36.

```
process
    begin
    dwrite <= '1';
    dval <= "00000001";
    dregsel <= "00";
    sregsel <= "01";
    wait for 100ns;
    dval <= "00000010";
    dregsel <= "01";
    wait for 100ns;
    dval <= "00000011";
    dregsel <= "10";
    sregsel <= "01";
    wait for 100ns;
    dval <= "00000100";
    dregsel <= "11";
    wait for 100ns;
    dwrite <= '0';
    dregsel <= "00";
    dval <= "00000101";
    sregsel <= "00";
    wait for 100ns;
    sregsel <= "01";
    wait for 100ns;
    sregsel <= "10";
    wait for 100ns;
    sregsel <= "11";
    wait for 100ns;
    dwrite <= '1';
    wait for 100ns;
end process;
```

Figure 35: Register File test bench

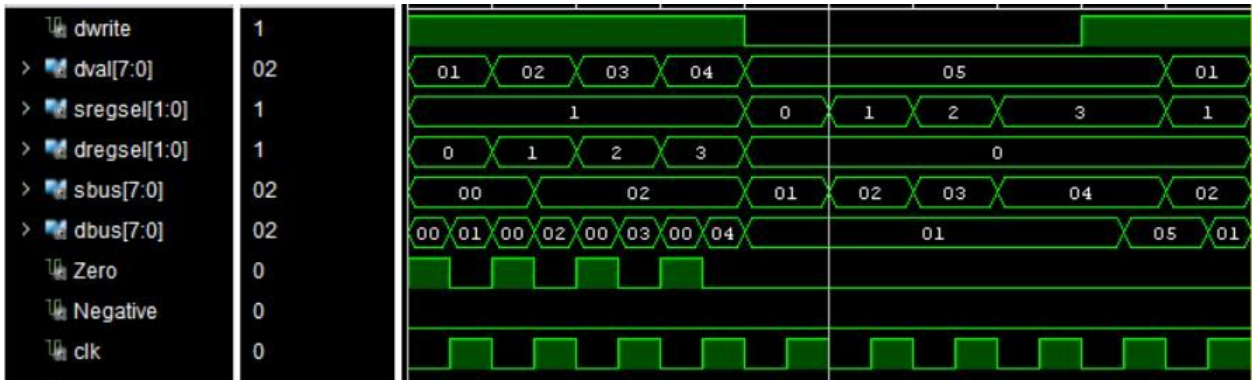

Figure 36: Register File simulation waveform

Registers:

Register are a common part of the microcontroller and are used to hold values that will be used by the microcontroller when they are needed. The following code in figure 37 shows how the registers are implemented throughout the microcontroller.

```
process(clk,enable)
    begin
        if (clk'event and clk = '1' and enable ='1') then
            PC_out <= PC_in;
        end if;
end process;
```

Figure 37: Register code

To show the loading process of the register the following test bench in figure 38 was used. This file loads in a value on the first enable. Then it turns enable off and does not load the next value then turns in on again. During this time it loads the last value. Following test bench is the waveform simulation is shown in figure 39.

```
process
    begin
        Reg_in <= "01000100";
        enable <='1';
        wait for 100ns;
        Reg_in <= "11101010";
        enable <='0';
        wait for 100ns;
        Reg_in <= "01010000";
        enable <= '1';
        wait for 100ns;
end process;
```

Figure 38: Register test bench

Figure 39: Register Simulation Waveform

Multiplexers:

Multiplexers are an important part of the project and are used in various locations. There are 3 different types of muxes that are used in the design of this microcontroller. The three types of muxes are a 4-to-1 mux, a 1-to-4 mux, and 2-to-1 mux. The following code in figures 40-48 shows the implementation of each mux, it's testbench, and simulation. The code for all the muxes is pretty simple. The files includes case statements that change what the output is based on the op that is passed to it.

4 to 1 Mux:

```
process(op,in_0,in_1,in_2,in_3)
    begin
        case op is
            when "00" =>
                out_mux <= in_0;
            when "01" =>
                out_mux <= in_1;
            when "10" =>
                out_mux <= in_2;
            when "11" =>
                out_mux <= in_3;
            when others =>
                out_mux <= "00000000";
        end case;
end process;
```

Figure 40: Mux_4_1 Code

```
process
    begin
        in_0 <= "00101100";
        in_1 <= "01010101";
        in_2 <= "10001000";
        in_3 <= "00010101";

        op <= "00";
        wait for 50ns;
        op <= "01";
        wait for 50ns;
        op <= "10";
        wait for 50ns;
        op <= "11";
        wait for 50ns;
        in_3 <= "00001000";
        wait for 50ns;
end process;
```

Figure 41: Mux_4_1 Test Bench:

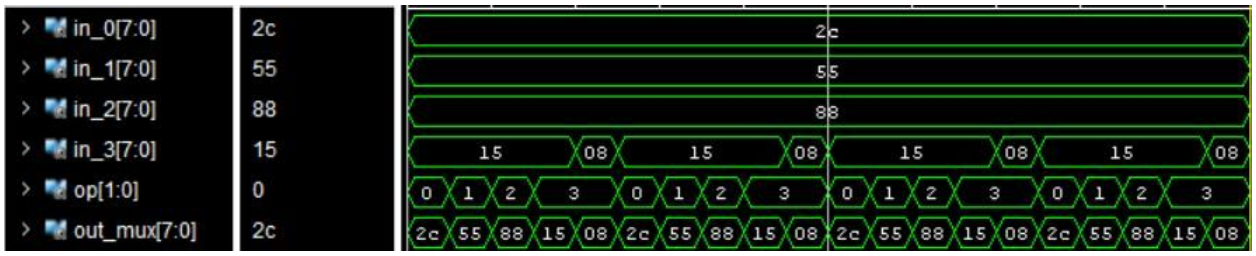

Figure 42: Mux_4_1 Simulation Waveform

Mux 1 to 4:

```
process(Mux_in,op)
    begin
        case op is
            when "00" =>
                out_0 <= Mux_in;
                out_1 <= '0';
                out_2 <= '0';
                out_3 <= '0';
            when "01" =>
                out_0 <= '0';
                out_1 <= Mux_in;
                out_2 <= '0';
                out_3 <= '0';
            when "10" =>
                out_0 <= '0';
                out_1 <= '0';
                out_2 <= Mux_in;
                out_3 <= '0';
            when "11" =>
                out_0 <= '0';
                out_1 <= '0';
                out_2 <= '0';
                out_3 <= Mux_in;
            when others =>
                out_0 <= '0';
                out_1 <= '0';
                out_2 <= '0';
                out_3 <= '0';
        end case;
end process;
```

Figure 43: Mux_1_4 Code

```
process
    begin
        Mux_in <='1';
        op <= "00";
        wait for 50ns;
        op <= "01";
        wait for 50ns;
        op <= "10";
        wait for 50ns;
        op <= "11";
        wait for 50ns;
        Mux_in <='0';
        wait for 50ns;
end process;
```

Figure 44: Mux_1_4 Test Bench

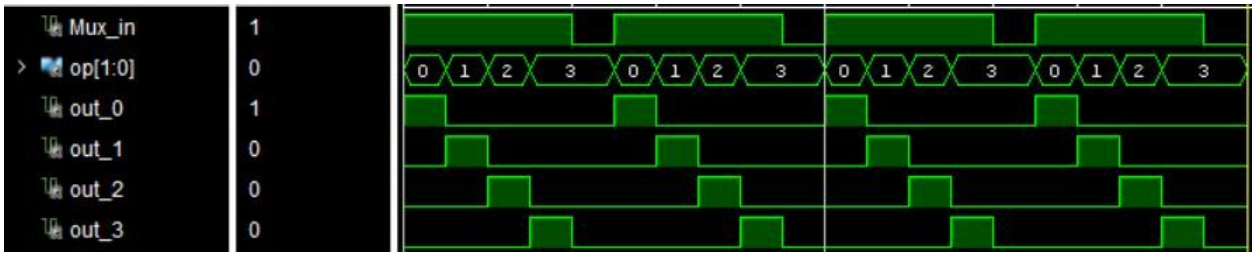

Figure 45: Mux_1_4 Simulation Waveform

Mux 2 to 1:

```
process(op,in_0,in_1)
begin
    case op is
        when '0' =>
            out_mux <= in_0;
        when '1' =>
            out_mux <= in_1;
        when others =>
            out_mux <= "00000000";
    end case;
end process;
```

Figure 46: Mux_2_1 Code

```
process
    begin
        in_0 <= "00000001";
        in_1 <= "00000010";

        op <= '0';
        wait for 100ns;
        op <= '1';
        wait for 100ns;
        in_1 <= "00000011";
        wait for 100ns;
        op <= '0';
        wait for 100ns;
        in_0 <= "00001000";
        wait for 100ns;
end process;
```

Figure 47: Mux_2_1 Test Bench

Figure 48: Mux_2_1 Simulation Waveform

ALU:

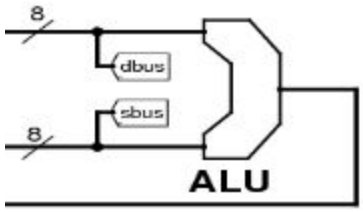

Figure 49: ALU Block Diagram [1]

The arithmetic logic unit (ALU) was built using the knowledge gained from previous labs. The instructions supported include bitwise AND, bitwise OR, arithmetic add, and arithmetic subtraction. When an improper instruction is selected, the result of the ALU is zero.

```
50 begin
51
52     process(A, B, aluop)
53     begin
54         case aluop is
55             when OP_AND =>
56                 result <= a and b;
57             when OP_OR =>
58                 result <= a or b;
59             when OP_ADD =>
60                 result <= STD_LOGIC_VECTOR(SIGNED(A) + SIGNED(B));
61             when OP_SUB =>
62                 result <= STD_LOGIC_VECTOR(SIGNED(A)-SIGNED(B));
63             when OTHERS =>
64                 result <= ZERO;
65         end case;
66     end process;
67 end Arch;
```

Figure 50: Arithmetic Logic Unit Architecture

Microcontroller Implementation:

Once all of the components necessary for the microcontroller to work were built and tested successfully, then the components were mapped together and included in the microcontroller entity. A test bench was made following the instructions included in the specified final program, but the microcontroller did not work properly on the first simulation. The team had to make come up with many ideas to correct this error. The problem in the microcontroller was that the memory file did not contain the instructions needed to run the program already loaded onto it. There was a miscommunication between team members to where two team members expected the memory to be written at the beginning of the program, and the other team members expected the instructions to already be loaded onto the memory. This was fixed and the instructions were preloaded onto the memory to follow the instructions on the project description. The microcontroller file is show in figure 51, and the first problem is shown in figure 52.

```
entity Microcontroller is
    Port(
        clk : in std_logic;
        sbus : out std_logic_vector(7 downto 0);
        dbus : out std_logic_vector(7 downto 0);
        aluout : out std_logic_vector(7 downto 0);
        immed : out std_logic_vector(7 downto 0);
        aluop : out std_logic_vector(1 downto 0);
        negative : out std_logic;
        zero : out std_logic;
        pcsel : out std_logic;
        stage : out std_logic_vector(1 downto 0);
        pcload : out std_logic;
        addressout : out std_logic_vector(7 downto 0);
        irlineout : out std_logic_vector(7 downto 0)
    );
end entity;
```

Figure 51: Microcontroller Entity

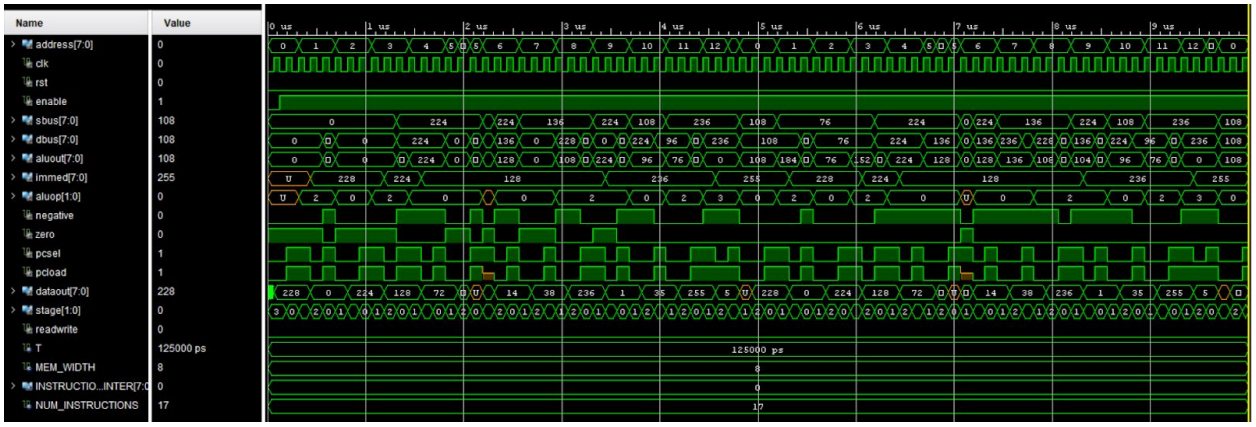

Figure 52: Microcontroller Simulation Problem 1

Another problem that occurred, once the first problem was fixed, was a timing issue. The team made the decision that when the MSB of the current instruction is set to ‘1’, then the following instruction would not be an instruction that the microcontroller needed to compute, but the next input of the immediate register. This created the timing use that when the other logic did not reflect this decision, what actually occurred was that the current value of the instruction was loaded into the immediate register. This problem was fixed by delaying the immedload signal a clock stage cycle so that the following instruction would be loaded into the immediate register. Although this fixed the first

implementation of the problem, it created the same problem further in the program. This problem can be seen in the simulation included in figure 53

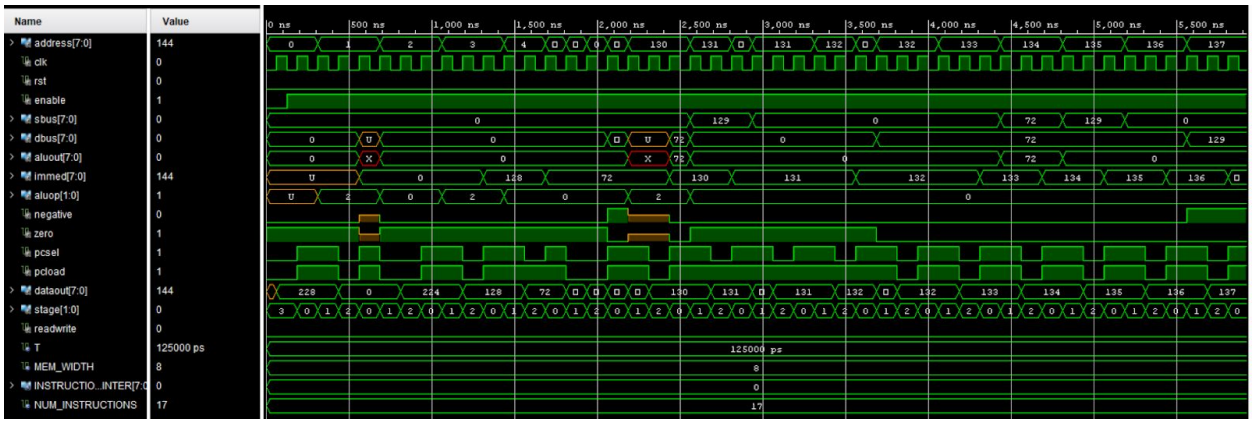

Figure 53: Microcontroller Simulation Problem 2

Additional Contributions Made By The Team:

An assembler was built to convert the assembly instruction set architecture to executable machine code. The assembler was built by parsing an assembly file using a python script. The machine code is then wrote into the top level simulation file where the instructions are then loaded into memory. Other tweaks occur to the file such as instruction length. The usage of the assembler is as follows:

```
python assembler.py > Source/Memory.vhd
```

The top level file would then be loaded into vivado and ran.

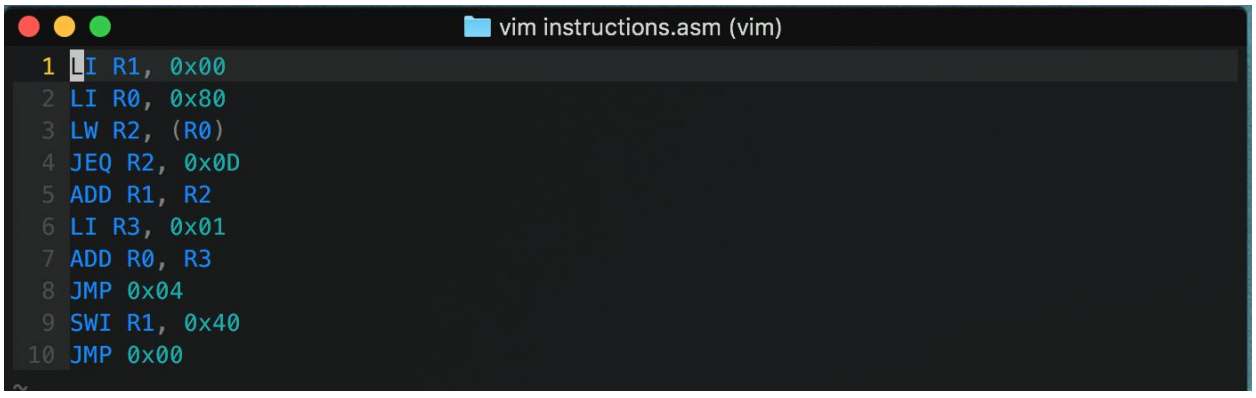

Figure 54: Assembly File - input to assembler

References:

1. Saqib, Fareena. “ECE 4146/5146 Introduction to VHDL Microcontroller design.” August 25, 2018. https://uncc.instructure.com/courses/73966/files/folder/Project?preview=3366702