CPE 272-Section 004

West Virginia University at Lane Department of Computer Science and Electrical Engineering

Final Laboratory Experiment Report

Design of a Simple CPU

Author: Ian Jackson

Partner: Natalie French

Date: November 21, 2021

Table of Contents

List of Figures

1. Introduction

The final project for the lab is to design a simple Central Processing Unit (CPU). CPUs are crucial parts of modern computers as they fetch information from memory and execute instructions. The CPU that is designed in this project follows the Von Nuemann model, the program instructions and data are stored in the same memory space. Thus, the operation code (Op Code), 3 most significant bits, and the memory address, 5 remaining bits, are stored in the same string of 8-bits.

When executing an instruction, the CPU begins by reading the first 8-bit data entry and decodes it. Based on what the Op Code is in the data entry, the CPU can do one of three things: LOADA, ADDA, or STOREA. LOADA is the operation to load the value at the specified memory location to the accumulator. ADDA is the operation that adds the value at the specified memory location to the value in the accumulator. Lastly, the STOREA instruction stores the value of the accumulator to a specified memory location. More detail about the steps the CPU goes through to complete these instructions is in Section 5.

The CPU has nine components that work together to execute instructions. The accumulator register (A) holds the 8-bit output of the ALU and as an input to the ALU, it also is one of the outputs displayed on the DE-10 Lite board. The instruction register (IR) holds the 8-bit instruction that is currently being executed. The control unit (CU) directs all the components of the CPU. The CU can be modeled as a finite state machine, which is described in Section 5. The program counter register (PC) is a register that holds the next instruction to be executed. The arithmetic & logic unit (ALU) preforms different operations on two 8-bit numbers. The memory address register (MAR) is a register that holds the value of the memory location. The MAR can receive data from either the PC or the IR, so a two-to-one multiplexer is placed in from of the

MAR to control what register has access to it. The multiplexer is controlled by the CU. The memory data register input (MDRI) and the memory data register (MDRO) are the registers that reads, MDRI, or writes, MDRO, from/to the memory of the CPU. Lastly, the memory structure (RAM) is the memory component of the CPU. It can store 32 different 8-bit values, with a memory location of 5-bits.

2. Hardware Description

The hardware used for this project is the DE-10 Lite board by Intel. This FPGA device has been used for almost every lab. The main components that were used was the ten switches, two buttons, ten LEDs and six seven-segment displays. A picture of the DE-10 Lite board can be seen in Figure 1.

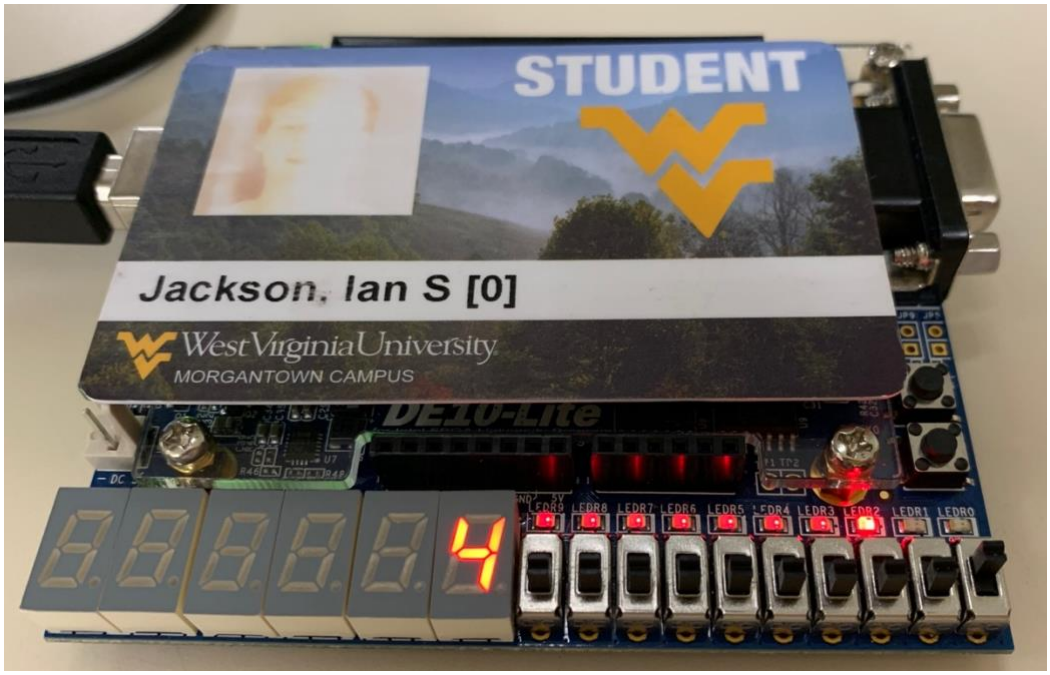

Figure 1. DE10-Lite Board

This board is versatile and can be programmed to do a plethora; from demonstrating function of logic gates to demonstrating finite state machines using sequential logic. The

possibilities are endless and there are other components on the DE10-Lite board that have not been used in lab but opens the doors to more possibilities.

3. Software Description

To program the DE10-Lite board, the software Quartus Prime is used. Quartus Prime is an IDE from Intel designed for the many FPGA devices from Intel. This software can program the board using logic schematics, VHDL, and more. Most of the labs were coded in VHDL. Quartus also has a waveform simulation feature, which can run a program without a DE-10 Lite board present.

4. Code Description

Most of the code was provided, but critical parts of the code needed to be added. The code that was added can be seen in the appendix. The entire code for the project can found here at this GitHub repository. The lines of code in the report refer to the lines of code in the GitHub repository.

4.1. CPU

The CPU file is the top-level file that houses each of the lower-level files. Most of the CPU file is importing each of the lower-level files via component statements (lines 23-96). The connections between each of the different components, named busses, are declared as signal variables with varying width depending on what bus it is (lines 99-116). The final portion of the CPU file is to port map the different inputs and outputs of the lower-level files to the busses (lines 119-195). The port mapping was based off the block diagram which is described in Section 6.

4.2. ControlUnit

The ControlUnit file contains the code that lays out the operation of the CU. The first part of the architecture block (lines 24-28) defines each of the different possible states the CU can be in. As mentioned above, the CU can be modeled as a finite state machine, so enumerations are used for this VHDL code. The second part of the architecture block puts the finite state machine diagram in Figure 2 into code. The code goes through each state, depending on the current state of the machine. The last part of the architecture block (lines 89-246) define what components of the CPU are enabled during each state. Each component of the CPU has an enable input, which is controlled by the CU. The block diagram for the CU can be seen in Figure 2.

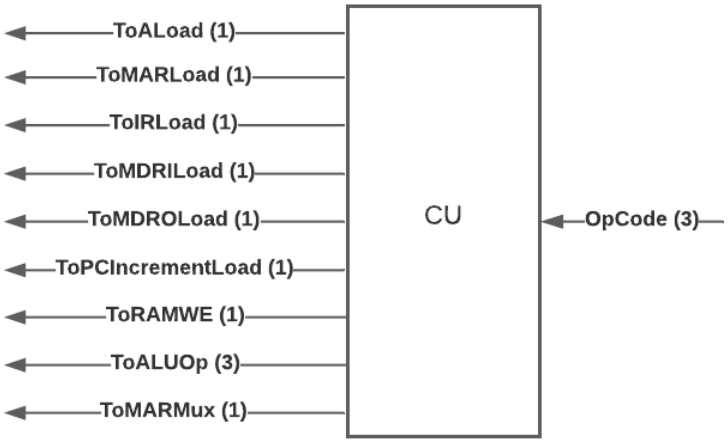

Figure 2. Control Unit Block Diagram

4.3. Memory

The memory file defines the RAM of the CPU. The RAM for this CPU can store up to 32 different 8-bit binary numbers. The RAM is pre-loaded with nine instructions (line 20) and the remaining storage of the RAM is filled with zeros. The RAM has a write enable input, which tells the RAM if it is reading from or writing to (lines 25-32). The block diagram for the RAM can be seen in Figure 3.

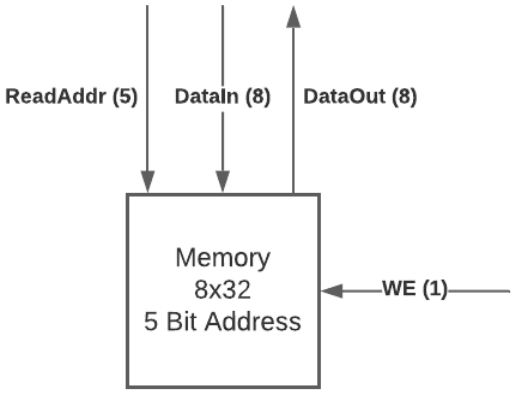

Figure 3. Memory Block Diagram

4.4. Reg

The reg file is the code for each of the register components. The accumulator, IR, MAR, MDRI, and MDRO all use the register code for their port maps. The register is a D Flip-Flop, so when the clock is at its rising edge, the input gets copied to the output. The only difference is that the register has an enable input, which is what the CU controls. The accumulator block diagram, called a register, can be seen in Figure 4.

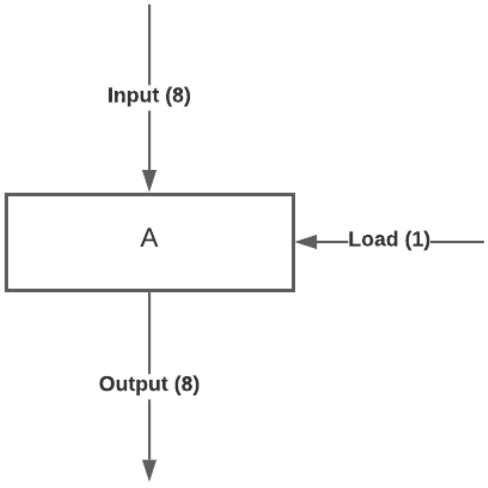

Figure 4. Register Block Diagram

4.5. ProgramCounter

The programCounter code defines the function of the PC. The initial value of the counter is zero. On the rising edge of the clock and if the counter receives input from the CU, the counter

increases its value by one. Since the output of the PC is an 8-bit binary number, the value of the counter is converted to an 8-bit binary number (lines 17-22). The PC block diagram can be seen in Figure 5.

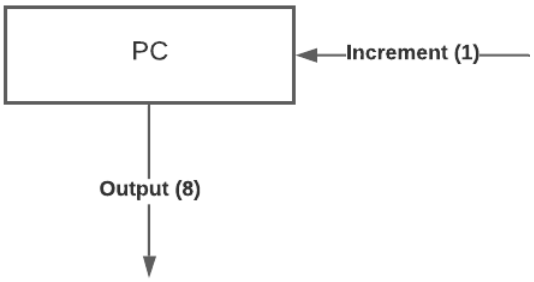

Figure 5. Program Counter Block Diagram

4.6. ALU

The ALU code defines what operations the ALU can perform. The ALU is coded to do addition, subtraction, bitwise AND, bitwise OR, output the B input, and output the A input. Each of these operations have their own Op Code, which is an input controlled by the CU. It is important to note that the Op Code from the data in the RAM and the ALU Op Code are two different things. The block diagram for the ALU can be seen in Figure 6.

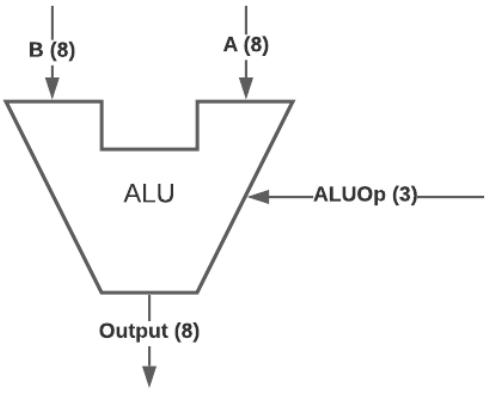

Figure 6. ALU Block Diagram

4.7. TwoToOneMux

The twoToOneMux is the code for the multiplexer that is placed in front of the MAR. The output of the multiplexer is controlled by the CU. The two inputs are from the IR or the PC. The block diagram for the multiplexer can be seen in Figure 7.

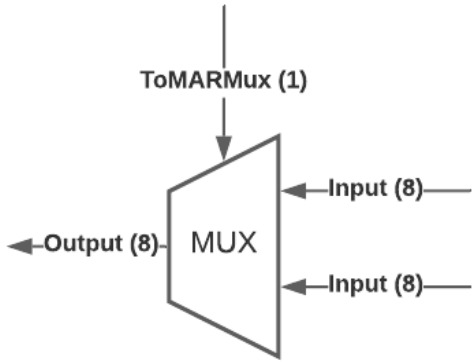

Figure 7. Multiplexer Block Diagram

4.8. SevenSeg

The sevenSeg code is the code that converts the binary outputs to outputs for the seven-segment display. All the Boolean expressions are imported from Lab 2. The accumulator and the PC are displayed on the seven-segment display.

5. Finite State Machine Diagram

As mentioned above, the CU can be modeled as a finite state machine. Each state is an instruction run by the CU. When the CU reaches a state, based on said state, it sends a signal to one or more of the elements using its outputs. For example, if the instruction is to load the MDRI, the CU sends the “toMDRILoad” signal to the load input of the MDRI. The state diagram for the CU can be seen in Figure 8. The FSM starts with a fetch instruction: load the MAR, read from the memory, load the MDRI, load the IR, and decode what is in the IR. Once the CU is in the decode state, it chooses what instruction was received based on the three most significant bits of the retrieved.

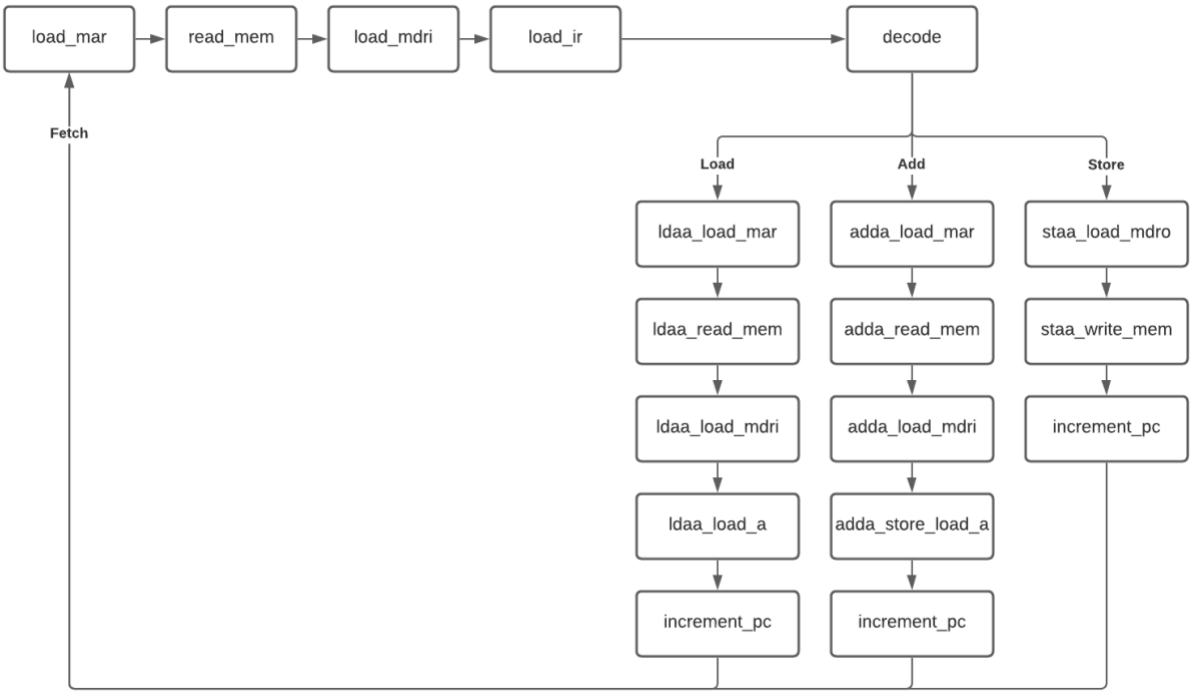

Figure 8. Control Unit Finite State Machine

6. Block Diagrams

A simple block diagram of the CPU was given in the background, but it was lacking detail. A more detailed block diagram can be seen in Figure 9. This clock diagram shows the individual bus connections and their width. The names of the busses correspond to the names given in the code.

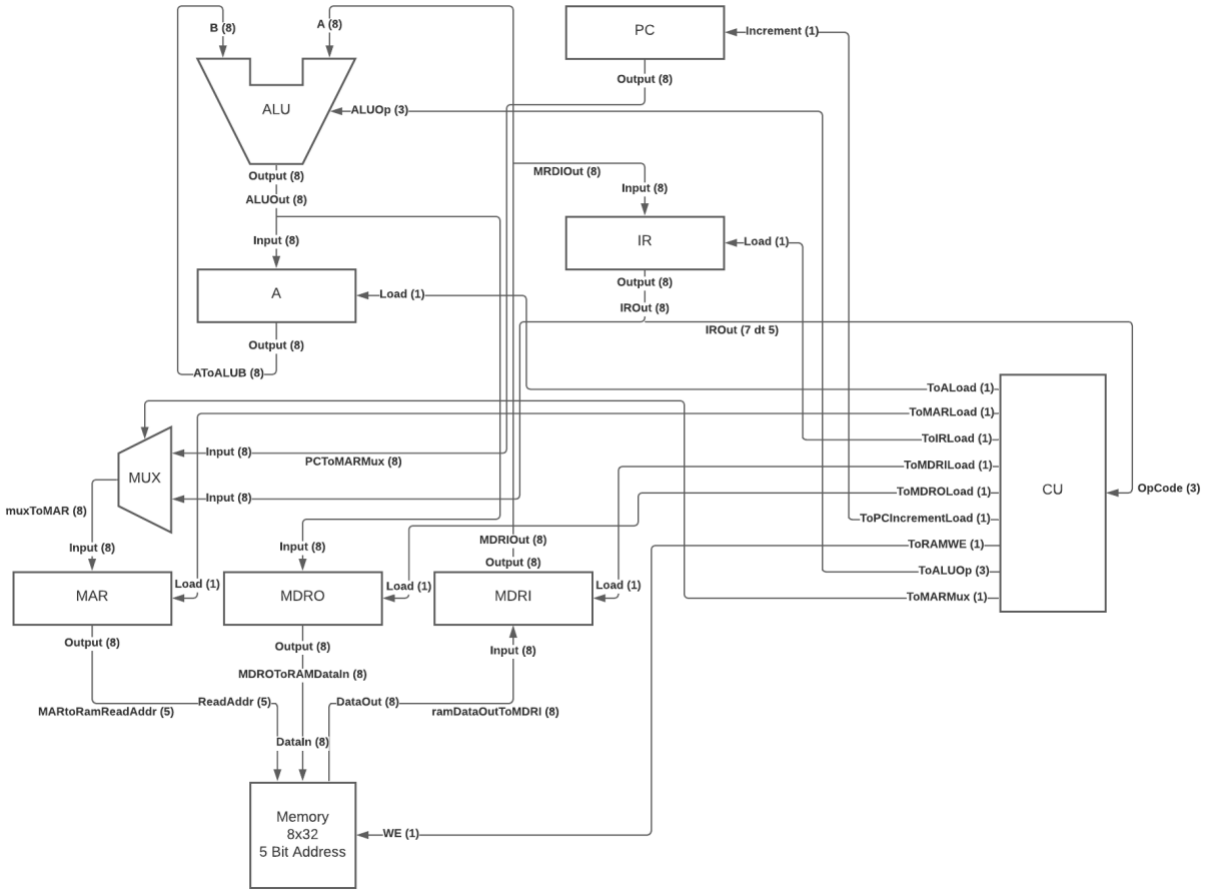

Figure 9. CPU Block Diagram

7. Results

The CPU program is run through a waveform simulation, so all the outputs could be seen at once. The waveforms are seen in Figure 10, Figure 11, and Figure 12. The accumulator (top waveform) and the PC (bottom waveform) has the decimal equivalent shown in the waveform.

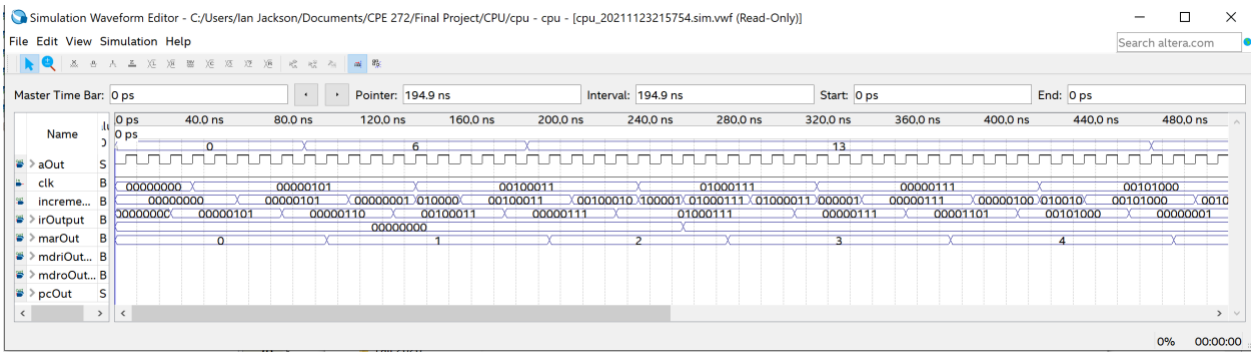

Figure 10. Waveform Simulation (0 ns to 500 ns)

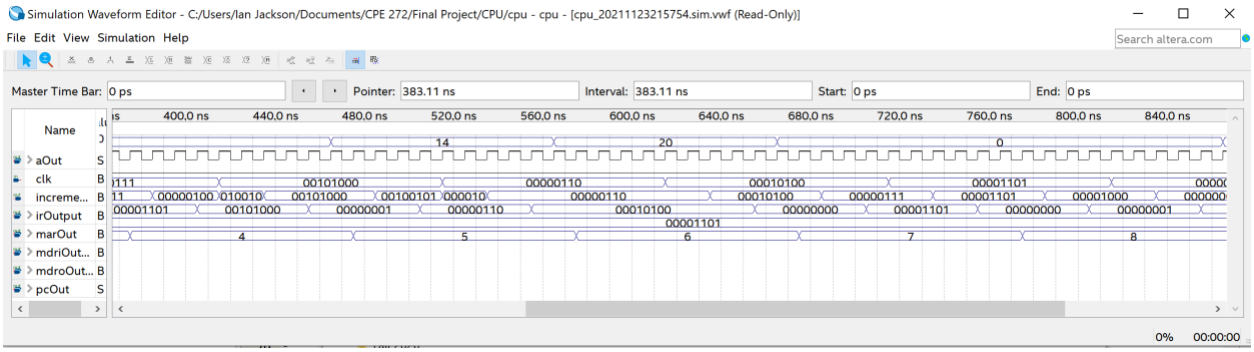

Figure 11. Waveform Simulation (370 ns to 860 ns)

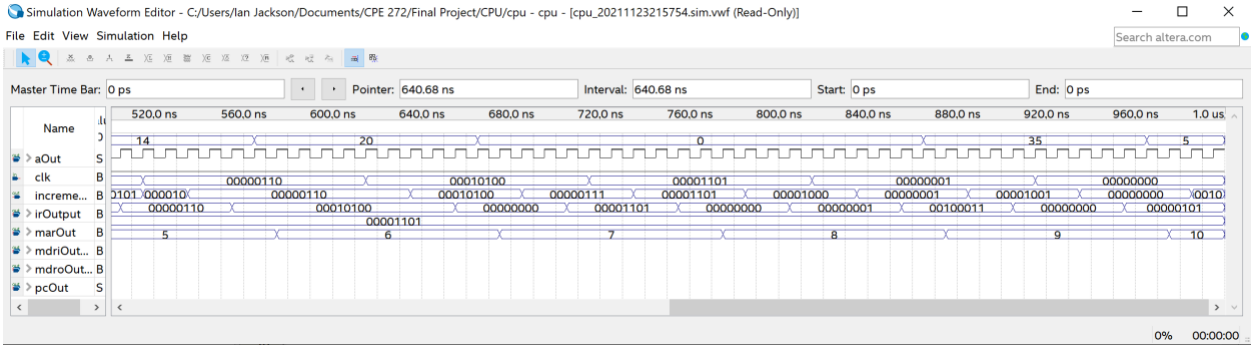

Figure 12. Waveform Simulation (500 ns to 1000 ns)

The waveform simulation is hard to read unless you walk through each clock cycle and decode what is going on. The table below in Figure 13. The first column shows the location of the memory where the data is being pulled from. The second column shows what the binary value of the data point is. The opcode column decodes what operation code is being called from the binary number pulled. The address column decodes what address in the ram to pull the data from. The value column shows the decimal equivalent of the value at the address given from the binary input. The last column, accumulator, shows the value of the accumulator after the operation is complete.

Memory Data					
RAM	**Binary**	**OpCode**	**Address**	**Value**	**Accumulator**
0	00000101	000 – loadA	00101 – 5	6	6
1	00100011	001 – addA	00011 – 3	7	13
2	01000111	010 – storeA	00111 – 7	13	13
3	00000111	000 – loadA	00111 – 7	13	13
4	00101000	001 – addA	01000 – 8	1	14
5	00000110	000 - storeA	00110 – 6	20	20
6	00010100	000 – storeA	10100 – 20	0	0
7	00001101	000 – storeA	01101 – 13	0	0
8	00000001	000 - storeA	00001 – 35	35	35

Figure 13. CPU Operation Table

8. Problems Occurred

The one problem was deciding what connections are made to each element. The two connections that were the most problematic were what was the output of the internal register is and the input to the CU. After reading the project background carefully, it made sense that the three most significant bits of the IR output goes to the input of the CU. The full output of the IR goes to the MAR but goes through the multiplexer first.

9. Conclusions

The design of a simple CPU is a great end of the semester project. It takes almost all the elements that were learned this semester and integrate it into one project. Even though this is called a “simple” CPU, the steps and process to create this was anything but simple. This project could be expanded, by adding more memory, another ALU and accumulator, or allowing the ALU to do more operations.

10. Appendix

A. CPU Code

```
-- accumulator
mapAccumulator: reg port map(clk => clk, load => CUToALoad,
     input => ALUOut, output => AToALUB);

-- ALU
mapALU: ALU port map(A => MDRIOut, B => AToALUB, ALUOp =>
     CUToALUOp, output => ALUOut);

-- program counter
mapPC: programCounter port map(clk => clk, increment =>
     CUToPCIncrement, output => pcToMARMux);

-- instruction register
mapIR: reg port map(clk => clk, load => CUToIRLoad, input =>
     MDRIOut, output => IROut);

-- MAR Mux
mapMARMux: twoToOneMux port map(A => pcToMARMux, B => IROut,
     address => CUToMARMUX, output => muxToMAR);

-- memory access register
mapMAR: reg port map (clk => clk, input => muxToMAR, output =>
     MARToRAMReadAddr, load => CUToMARLoad);

-- memory data register input
mapMDRI: reg port map(clk => clk, input => RAMDataOutToMDRI,
     output => MDRIOut, load => CUToMDRILoad);

-- memory data register output
mapMDRO: reg port map(clk => clk, input => ALUOut, output =>
     MDROToRamDataIn, load => CUToMDROLoad);

-- control unit
mapCU: controlUnit port map(clk => clk, opCode => IROut(7
     downto 5), toALoad => CUToALoad, toMARLoad => CUToMARLoad,
     toIRLoad => CUToIRLoad, toMDRILoad => CUtoMDRILoad,
     toMDROLoad => CUToMDROLoad, toPCIncrement =>
     CUtoPCIncrement, toMARMux => CUToMARMUX, toRAMWriteEnable
     => CUtoRAMWriteEnable, toALUOp => CUToALUOp);

-- ssd
aOutHex0: sevenSeg port map(i(3) => AToALUB(3), i(2) =>
     AToALUB(2), i(1) => AToALUB(1), i(0) => AToALUB(0), o(6)
     => ssd1(6), o(5) => ssd1(5), o(4) => ssd1(4), o(3) =>
```

```
     ssd1(3), o(2) => ssd1(2), o(1) => ssd1(1), o(0) =>
     ssd1(0));

aOutHex1: sevenSeg port map(i(3) => '0', i(2) => '0', i(1) =>
     '0', i(0) => AToALUB(4), o(6) => ssd2(6), o(5) => ssd2(5),
     o(4) => ssd2(4), o(3) => ssd2(3), o(2) => ssd2(2), o(1) =>
     ssd2(1), o(0) => ssd2(0));

pcOutHex5: sevenSeg port map(i(3) => PCToMARMux(3), i(2) =>
     PCToMARMux(2), i(1) => PCToMARMux(1), i(0) =>
     PCToMARMux(0), o(6) => ssd3(6), o(5) => ssd3(5), o(4) =>
     ssd3(4), o(3) => ssd3(3), o(2) => ssd3(2), o(1) =>
     ssd3(1), o(0) => ssd3(0));
```

B. ControlUnit Code

```
-- add instruction
when adaa_load_mar =>
     currentState <= adaa_read_mem;
when adaa_read_mem =>
     currentState <= adaa_load_mdri;
when adaa_load_mdri =>
     currentState <= adaa_store_load_a;
when adaa_store_load_a =>
     currentState <= increment_pc;

-- store instruction
when staa_load_mdro =>
     currentState <= staa_write_mem;
when staa_write_mem =>
     currentState <= increment_pc;

                                ⋮

when load_mdri =>
     toALoad <= '0';
     toPCIncrement <= '0';
     toMARMux <= '0';
     toMARLoad <= '0';
     toRAMWriteEnable <= '0';
     toMDRILoad <= '1';
     toIRLoad <= '0';
     toMDROLoad <= '0';
when load_ir =>
     toALoad <= '0';
     toPCIncrement <= '0';
```

```
      toMARMux <= '0';
      toMARLoad <= '0';
      toRAMWriteEnable <= '0';
      toMDRILoad <= '0';
      toIRLoad <= '1';
      toMDROLoad <= '0';
when  decode =>
      toALoad <= '0';
      toPCIncrement <= '0';
      toMARMux <= '0';
      toMARLoad <= '0';
      toRAMWriteEnable <= '0';
      toMDRILoad <= '0';
      toIRLoad <= '0';
      toMDROLoad <= '0';

                                    ⋮

when  ldaa_read_mem =>
      toALoad <= '0';
      toPCIncrement <= '0';
      toMARMux <= '0';
      toMARLoad <= '0';
      toRAMWriteEnable <= '0';
      toMDRILoad <= '0';
      toIRLoad <= '0';
      toMDROLoad <= '0';

                                    ⋮

when  ldaa_load_a =>
      toALoad <= '1';
      toPCIncrement <= '0';
      toMARMux <= '0';
      toMARLoad <= '0';
      toRAMWriteEnable <= '0';
      toMDRILoad <= '0';
      toIRLoad <= '0';
      toMDROLoad <= '0';
      toALUOp <= "101";

                                    ⋮

when  adaa_read_mem =
      toALoad <= '0';
      toPCIncrement <= '0';
      toMARMux <= '0';
```

```
     toMARLoad <= '0';
     toRAMWriteEnable <= '0';
     toMDRILoad <= '0';
     toIRLoad <= '0';
     toMDROLoad <= '0';
when adaa_load_mdri =>
     toALoad <= '0';
     toPCIncrement <= '0';
     toMARMux <= '0';
     toMARLoad <= '0';
     toRAMWriteEnable <= '0';
     toMDRILoad <= '1';
     toIRLoad <= '0';
     toMDROLoad <= '0';

                                ⋮


when staa_write_mem =>
     toALoad <= '0';
     toPCIncrement <= '0';
     toMARMux <= '0';
     toMARLoad <= '0';
     toRAMWriteEnable <= '1';
     toMDRILoad <= '0';
     toIRLoad <= '0';
     toMDROLoad <= '0';
```

C. Memory Code

```
architecture behavior of memory is

type ram_type is array(0 to 31) of std_logic_vector(7 downto
     0);
signal mem: ram_type := ("00000101", "00100011", "01000111",
     "00000111", "00101000","00000110", "00010100", "00001101",
     "00000001", others=>(others=>'0'));

begin
process(clk, we)
begin
     if clk'event and clk='0' then
          if we = '0' then
               dataOut <= mem(conv_integer(readAddr));
          elsif we = '1' then
               mem(conv_integer(readAddr)) <= dataIn;
```

```
          end if;
     end if;
end process;
end behavior;
```

D. Reg Code

```
architecture behavior of reg is
begin
process(clk,load)
begin
     if(clk'event and clk='1' and load='1') then
          output <= input;
     end if;
end process;
end behavior;
```

E. ProgramCounter Code

```
process(clk,increment)
     variable counter: integer := 0;
     begin
          if (clk'event and clk='1' and increment = '1') then
               counter := counter + 1;
               output <= conv_std_logic_vector(counter,8);
          end if;
end process;
```

F. ALU Code

```
architecture behavior of alu is
begin
process(A,B,ALUOp)
begin
     if(ALUOp = "000") then output <= (A + B);
     elsif(ALUOp = "001") then output <= (A - B);
     elsif(ALUOp = "010") then output <= (A and B);
     elsif(ALUOp = "011") then output <= (A or B);
     elsif(ALUOp = "100") then output <= B;
     elsif(ALUOp = "101") then output <= A;
     end if;
end process;
end behavior;
```

G. TwoToOneMux Code

```
architecture behavior of twoToOneMux is
begin
process(A,B,address)
begin
     if(address = '0') then
          output <= A;
     elsif(address = '1') then
          output <= B;
     end if;
end process;
end behavior;
```

H. SevenSeg Code

```
architecture behavior of sevenSeg is

component binary2hex
     port(
          --inputs
          W: in std_logic;
          X: in std_logic;
          Y: in std_logic;
          Z: in std_logic;

          --outputs
          a: out std_logic;
          b: out std_logic;
          c: out std_logic;
          d: out std_logic;
          e: out std_logic;
          f: out std_logic;
          g: out std_logic
     );
end component;

begin
sgd: binary2hex port map(W => i(3), X => i(2), Y => i(1), Z =>
     i(0), a => o(6), b => o(5), c => o(4), d => o(3), e =>
     o(2), f => o(1), g => o(0));
end behavior;
```