Names: Kollin Labowski and Matthew Winston

Computer Engineering 272-002

Final Project Report CPE 272

Date Completed: 4/27/2020

Introduction

The goal of this project was to program a simple, fully-functioning CPU. This was accomplished by using several different code modules to simulate the different components of the CPU. While much of the required code for the project was provided, functionality for four specific functions was to be implemented to complete the CPU. The first of these instructions was FETCH, which would read in an instruction from memory which would be decoded to determine the next step in the process. The next instruction to be implemented was LOADA, which would read in a value from a specified memory location and store it in an accumulator register for later use. The third instruction to be implemented was ADDA, which would add a value stored at a specified memory location to the value stored in the accumulator register. The last instruction to be implemented was STOREA, which would take a value from the accumulator register and store it into a specified memory location. Implementing these instructions was accomplished by completing a diagram of all components in the CPU and their connections. This diagram was then analyzed and used to develop a state diagram of the four different processes. This state diagram was used as a reference when filling in the incomplete portions of code to allow the CPU to perform the four processes. The entire project was completed using the Quartus II software, and all code was created in VHDL files. The project was completed based on guidance provided in the final project materials.

Software Description

The Quartus II software, released June of 2013, is a simulation environment that allows the user to model circuit gates, program GAL chips, and code in VHDL, among many other functions. The VHDL code can be connected to an Altera board to have physical switch inputs and LED outputs to visualize and test the code. The inputs and outputs have to be manually mapped to their respective switches and LEDs before the Altera board can be used properly.

If an Altera board is unavailable, Quartus II has a Waveform Simulation that can show outputs and the “signal” data type over time, with inputs being set by the user. In the case of a clock as an input, the Waveform Simulation has an Overwrite Clock option that allows the user to customize a clock.

The coding language used, VHDL, stands for “VHSIC Hardware Description Language”. The acronym “VHSIC” stands for “Very High Speed Integrated Circuits”. The VHDL code in one file can be brought into another file for quick use of repeated functionality. This process is known as “importing”, and is a common practice. Importing allows code to be created and understood efficiently due to the code being organized.

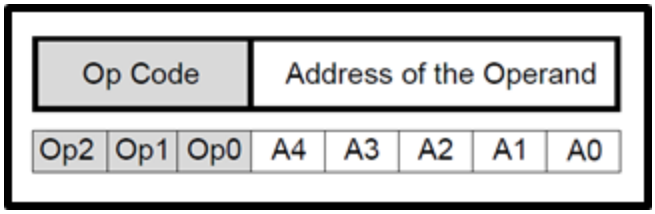
Description of VHDL Files

Much of the code for the VHDL files was provided with the final project materials. Each component of the CPU was simulated using a specific VHDL file. The final CPU used a combination of seven different VHDL files. The first VHDL file was titled “alu”, and it contained code to simulate an arithmetic logic unit (ALU). This code can be seen in **Figure 1**. It used three inputs, two of which were values to be operated on, and the third of which was an opcode used to determine the operation to perform on them. The determination of the operation to perform was completed by using conditional if statements. The operations it could perform included addition, subtraction, logical and, and logical or, and it could also return either of the two inputs unchanged.

Another VHDL file created using the code template was called “memory_8_by_32”, and it was used to simulate random access memory (RAM). The function of the code was to store data at specific memory locations. It had four inputs, one of which was a clock. The input “Write_Enable” was used to determine whether data should be stored in or read from memory. When “Write_Enable” was equal to 1, it would write the input to memory, and it would simply read from memory when this input was equal to 0. The input “Read_Addr” was a logic vector containing the memory location to access. “Data_in” was the value to store into a specific memory location. “Data_out” was an output which contained the value read from the location based on “Read_Addr”. The functionality of the memory was completed using a signal and conditional statements. The code for “memory_8_by_32” can be seen in **Figure 2**.

Another VHDL file created entirely with provided VHDL code was called “reg”. This file was to function as a register, such that it would store any value that needs to be stored. It included 3 inputs, one of which was a clock. The input labeled “input” was whatever value was to be stored into the register. The register value would only update in the case that the other input, “load”, was equal to 1 and the clock switched from low to high. This was accomplished using a single conditional if statement. The full code for “reg” can be seen in **Figure 3**. This code was used five times to represent five different components in the CPU that had the same basic behavior. These components were the instruction register (IR), accumulator (A), memory address register (MAR), memory data register input (MDRI), and memory data register output (MDRO). The IR was a register used in the design of the CPU which held the instruction that was currently being executed. The format of the instruction included an opcode and a memory address, and this format can be seen in **Table 1**. The A was a register that held a value that could be operated on by the ALU. This register was used as the basis for the LOADA, ADDA, and STOREA instructions. The MAR was a register that contained the memory address to be accessed next by either the MDRI or the MDRO. The MDRI was a register that contained a value that was read from memory. Alternatively, the MDRO was a register that contained the next value to be written to memory. Despite their differing uses in the design of the CPU, their basic function was the same.

Table 1: The Format of an Instruction in the CPU



The last VHDL file to be created entirely based on code from the template was labeled "TwoToOneMux". This code was intended to function as a multiplexer (MUX) with two inputs and an output. The actual code had three inputs. Inputs "A" and "B" represented the two different inputs the multiplexer could toggle between. The third input "address" was a single bit input that determined which input to connect to the output. The output "output" contained the value of the input that the multiplexer was wired to. Conditional if statements were used to connect the inputs to the output based on the value of the address. The full code can be seen in **Figure 4**. This code was used in the CPU to allow the outputs of both the program counter and the instruction register to be input to the memory address register.

Another VHDL file used in the project was called "ProgramCounter". The function of this part of the program was to hold the location of the next location in memory to read an instruction. This file had two inputs, one of which was a clock. The other input was labeled "increment", and would cause the counter to count up to the next location of a memory instruction. This program was incomplete when taken from the template, and three sections needed code filled in in order to fulfill the required behavior. Incrementing the memory location was done by an addition of 1 which was performed by importing *ieee.std_logic_arith.all* and *ieee.std_logic_unsigned.all*. The final VHDL file can be seen in **Figure 5**, and portions added in after the template can be seen in bold. In the design of the CPU, this file was used to simulate the program counter (PC).

The final VHDL file based on a particular component of the CPU was called "ControlUnit", and it represented the control unit (CU) component of the CPU design. The function of this component was to control each of the other components by altering the "load" inputs of various registers. These inputs were altered by the CU to complete the four different instructions, FETCH, LOADA, ADDA, and STOREA. The VHDL file only had two inputs, one of which was a clock. The other input, "OpCode" was a 3 bit code from the first 3 bits of the instruction register to tell the CU which function to perform. **Table 2** shows the different opcode combinations that the CU would read in and their corresponding functions. The outputs in the VHDL code connect to the "load" inputs of each of the five registers, as well as the "increment" input in the program counter, the "address" input of the multiplexer, the "Write_Enable" input of the RAM, and the opcode of the ALU. One portion of the CU that had to be filled in was the portion which defined the order of the state transitions based on the opcode input. This was accomplished by consulting the

created state diagram to determine the correct state transition order. The names of the states used in the state diagram came from the enumerated type that was declared in the template for this file. The next code portion to fill in in this file included defining each of the outputs at each state. In order to ensure each of the instructions worked as intended, the components were set to active only when the state the CU was in required it. Determining the functions to perform at each step was completed by reviewing the state diagram and the English translation for each of the instructions. The completed code for the CU can be seen in **Figure 6**.

Table 2: Functions of Opcode Combinations in Control Unit

Op 2	Op 1	Op 0	Instruction
0	0	0	LOADA
0	0	1	ADDA
0	1	0	STOREA

The final VHDL file used to create the CPU was labeled “SimpleCPU_Template”, and it served as the top level file in the project. This file had a single input, which was a clock, and 7 outputs to represent the values of the five registers, the program counter, and the increment variable from the program counter. The code from all of the other VHDL files was brought into the code using component statements. The portion to be filled in in this file included the creation of port map statements for each of the different components, including five instances of the “reg” code. The inputs and outputs of each of the components were set to signals so that the components would be connected as intended. The correct port mappings were made by reviewing the created CPU diagram. Finally, all of the outputs were set to their respective signals so that the program could be tested using the Waveform Simulator. The complete code for this VHDL file can be seen in **Figure 7**.

The code was tested using Waveform Simulator, and the completed waveforms can be seen in **Figure 8**.

Encountered Problems and Solutions

There were multiple problems that were encountered during the completion of this project, although most of them were solved in a timely manner. The most significant difficulties encountered in the project occurred during the coding of the project. After writing out all of the code, a test on the Waveform Simulator was attempted. Unfortunately, after trying several times to simulate the code, a timing simulation would not run at all. We had originally thought that we were using the simulator incorrectly, but this was not the case. Instead, the project was named “FinalProjectCPE272”, and there was a VHDL file called “FinalProjectCPE272” which we did not use. When we were running the Waveform Simulator, we assumed it was running from the

“SimpleCPU_Template” file, and completely forgot to consider that there was another, blank file that was being run instead. This problem was solved by creating a new project titled “SimpleCPU_Template”, and then transferring all of the VHDL files to this new project. Another significant issue encountered while programming the project caused the Waveform Simulator output to appear offset for the MAR output. This offset in the beginning of the Waveform Simulation caused almost all of the values later in the simulation to be incorrect. The issue was made more difficult to track by the fact that tracing each component seemed to show a correct behavior other than the initial offset. Eventually, after reviewing the code several times, it was found that while the multiplexer was programmed with the correct inputs from the control unit, in the SimpleCPU_Template the port map for the multiplexer was created in the reverse order of how it should have been. Instead of the program counter being set to input A and the instruction register set to B, the two inputs were unintentionally swapped, causing the hard-to-find error.

Complete Code

```
--Arithmetic Logic Unit Code
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
-- 8 bit operands and output
ENTITY alu IS
PORT(
A : in std_logic_vector          (7 downto 0);
B : in std_logic_vector          (7 downto 0);
AluOp : in std_logic_vector      (2 downto 0);
output : out std_logic_vector    (7 downto 0)
);

end alu;

-- decode op code, perform operation,
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;
```

Figure 1: The contents of VHDL file "alu"

```
-- 8 By 32 Memory Array
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity memory_8_by_32 is

Port(
        clk:            in std_logic;
        Write_Enable: in std_logic;
        Read_Addr:      in std_logic_vector     (4 downto 0);
        Data_in:        in std_logic_vector     (7 downto 0);
        Data_out:       out std_logic_vector(7 downto 0));
        end memory_8_by_32;

        architecture behavior of memory_8_by_32 is
        type ram_type is array(0 to 31) of std_logic_vector(7 downto 0);
        --instructions / data go into memory here
        signal Z: ram_type:=("00000101","00100110","01000111","00000111","00101000",
        "00001010","00010100","01010101","00000001","10110100","10001010","10101010",
        "10101001","00000000","10100101","01010101","10101110","10110100","10001010",
        "10101010","10101001","00000000","10100101","01010101","10101110","10110100",
        "10001010","10101010","10101001","00000000","10100101","01010101");
        Begin
        Process(clk,Read_Addr, Data_in, Write_Enable)
        Begin
        --Read from memory
        if(clk'event and clk='1' and Write_Enable='0') then
        Data_out<=Z(conv_integer(Read_Addr));
        --Write to Memory
        elsif(clk'event and clk='1' and Write_Enable='1') then
        Z(conv_integer(Read_Addr))<=Data_in;
        end if;
        end process;
        End;
```

Figure 2: The contents of the VHDL file “memory_8_by_32”

```
--Register component for CPU
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity reg is
port (
        input : in std_logic_vector                 (7 downto 0);
        output : out std_logic_vector     (7 downto 0);
        clk : in std_logic;
        load : in std_logic
);
end;

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;
```

Figure 3: The contents of the VHDL file “reg”

```
--Mux used to create a shared connection between PC and IR to the MAR
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity TwoToOneMux is
port (
        A : in std_logic_vector                 (7 downto 0);
        B : in std_logic_vector                 (7 downto 0);
        address : in std_logic;
        output : out std_logic_vector     (7 downto 0)
);
end;

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;
```

Figure 4: The contents of the VHDL file “TwoToOneMux”

```
--Program Counter Code
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
--Increments the program counter by 1 if there is a positive edge clock and increment =1
entity ProgramCounter is
port (
        output : out std_logic_vector(7 downto 0);
        clk : in std_logic;
        increment : in std_logic
);
end;

architecture behavior of ProgramCounter is
begin

process(clk,increment)
--Define a counter variable as an integer and initialize it to 0 (use variable counter: integer:=) and
fill in the value
--SEE INSERTED CODE BELOW
variable counter: integer:= 0;

begin
        --Create an if statement to check for the condition of a positive edge clock and increment
=1
        if (clk'event and clk = '1' and increment = '1') then
                --Increment counter variable by 1
                --SEE INSERTED CODE BELOW
                counter:= counter + 1;



                --Output the counter variable as a std logic vector of 8 bits,
                --Use function conv_std_logic_vector(counter,8)
                --SEE INSERTED CODE BELOW
                output <= conv_std_logic_vector(counter,8);
        end if;
end process;
end behavior;
```

Figure 5: The contents of the VHDL file "ProgramCounter"

```
-- Control Unit Code
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity ControlUnit is

port (
        --Op code used for instructions (NOT the ALU Op)
        OpCode : in std_logic_vector(2 downto 0);
        --Clock Signal
        clk : in std_logic;
        --Load bits to basically turn components on and off at a given state
        ToALoad : out std_logic;
        ToMarLoad : out std_logic;
        ToIrLoad : out std_logic;
        ToMdriLoad : out std_logic;
        ToMdroLoad : out std_logic;
        ToPcIncrement : out std_logic := '0';
        ToMarMux : out std_logic;
        ToRamWriteEnable : out std_logic;
        --This is the ALU op code, look inside the ALU code to set this
        ToAluOp : out std_logic_vector (2 downto 0)
);

end;

architecture behavior of ControlUnit is
--Custom Data Type to Define Each State
type cu_state_type is (load_mar, read_mem, load_mdri, load_ir, decode,
                                                    ldaa_load_mar, ldaa_read_mem,
ldaa_load_mdri, ldaa_load_a,
                                                    adaa_load_mar, adaa_read_mem,
adaa_load_mdri, adaa_store_load_a,
                                                    staa_load_mdro, staa_write_mem,
                                                    increment_pc);

--Signal to hold current state
signal current_state : cu_state_type;

begin
--Defines the transitions in our state machine
```

```
process(clk)
begin

if (clk'event and clk = '1') then
        case current_state is
                --Increment the pc and fetch the instruction, then load the IR with the fetched
instruction
                --Decode the instruction, use the diagram in the handout to determine the next
states
                when increment_pc =>
                        current_state <= load_mar;
                when load_mar =>
                --SEE INSERTED CODE BELOW
                        current_state <= read_mem;
                when read_mem =>
                        current_state <= load_mdri;
                when load_mdri =>
                        current_state <= load_ir;
                when load_ir =>
                        current_state <= decode;



                --Decode Opcode to determine Instruction
                --Assign current state based on the opCode
                when decode =>
                --SEE INSERTED CODE BELOW
                        if(OpCode = "000") then
                                current_state <= ldaa_load_mar;
                        elsif(OpCode = "001") then
                                current_state <= adaa_load_mar;
                        else
                                current_state <= staa_load_mdro;
                        end if;

                --Instructions, need to determine the next state to implement each instruction
                --Follow the path to perform each instruction as described in the handout, and
determine
                --Where the state machine needs to go to implement the instruction
                ---Load instruction
                when ldaa_load_mar =>
                        current_state <= ldaa_read_mem;
                        --SEE INSERTED CODE BELOW
```

```
                when ldaa_read_mem =>
                        current_state <= ldaa_load_mdri;
                when ldaa_load_mdri =>
                        current_state <= ldaa_load_a;
                when ldaa_load_a =>
                        current_state <= increment_pc;

                --Add Instruction
                when adaa_load_mar =>
                        current_state <= adaa_read_mem;
                        --SEE INSERTED CODE BELOW
                when adaa_read_mem =>
                        current_state <= adaa_load_mdri;
                when adaa_load_mdri =>
                        current_state <= adaa_store_load_a;
                when adaa_store_load_a =>
                        current_state <= increment_pc;

                --Store Instruction
                when staa_load_mdro =>
                        --SEE INSERTED CODE BELOW
                        current_state <= staa_write_mem;
                when staa_write_mem =>
                        current_state <= increment_pc;

        end case;
end if;
end process;
-- Defines what happens at each state, set to '1' if we want that component to be on
-- Set Op Code accordingly based on ALU, different from the instruction op code, look at the
actual ALU code
-- Keep in mind when ToMarMux = 0 , MAR is loaded from PC address, when ToMarMux = 1, MAR
is loaded with IR address

process(current_state)
begin

        ToALoad <= '0';
        ToMdroLoad <= '0';
        ToAluOp <= "000";

        case current_state is
                --Turns on the increment pc bit
```

```
when increment_pc =>
        ToALoad <= '0';
        ToPcIncrement <= '1';
        ToMarMux <= '0';
        ToMarLoad <= '0';
        ToRamWriteEnable <= '0';
        ToMdriLoad <= '0';
        ToIrLoad <= '0';
        ToMdroLoad <= '0';
        ToAluOp <= "000";
--Loads MAR with address from program counter
when load_mar =>
--SEE INSERTED CODE BELOW
        ToALoad <= '0';
        ToPcIncrement <= '0';
        ToMarMux <= '0';
        ToMarLoad <= '1';
        ToRamWriteEnable <= '0';
        ToMdriLoad <= '0';
        ToIrLoad <= '0';
        ToMdroLoad <= '0';
        --ToAluOp <= "100";

--Reads Address located in MAR
when read_mem =>
--SEE INSERTED CODE BELOW
        ToALoad <= '0';
        ToPcIncrement <= '0';
        ToMarMux <= '0';
        ToMarLoad <= '0';
        ToRamWriteEnable <= '0';
        ToMdriLoad <= '0';
        ToIrLoad <= '0';
        ToMdroLoad <= '0';
        --ToAluOp <= "100";

--Load Memory Data Register Input
when load_mdri =>
--SEE INSERTED CODE BELOW
        ToALoad <= '0';
        ToPcIncrement <= '0';
        ToMarMux <= '0';
        ToMarLoad <= '0';
```

```
            ToRamWriteEnable <= '0';
            ToMdriLoad <= '1';
            ToIrLoad <= '0';
            ToMdroLoad <= '0';
            --ToAluOp <= "100";

    --Loads the Instruction Register with instruction fetched from Memory
    when load_ir =>
    --SEE INSERTED CODE BELOW
            ToALoad <= '0';
            ToPcIncrement <= '0';
            ToMarMux <= '0';
            ToMarLoad <= '0';
            ToRamWriteEnable <= '0';
            ToMdriLoad <= '0';
            ToIrLoad <= '1';
            ToMdroLoad <= '0';
            --ToAluOp <= "100";

    --Decodes The current instruction (everything should be off for this)
    when decode =>
    --SEE INSERTED CODE BELOW
            ToALoad <= '0';
            ToPcIncrement <= '0';
            ToMarMux <= '0';
            ToMarLoad <= '0';
            ToRamWriteEnable <= '0';
            ToMdriLoad <= '0';
            ToIrLoad <= '0';
            ToMdroLoad <= '0';
            --ToAluOp <= "100";

    --Loads the MAR with address stored in IR
    when ldaa_load_mar =>
    --SEE INSERTED CODE BELOW
            ToALoad <= '0';
            ToPcIncrement <= '0';
            ToMarMux <= '1';
            ToMarLoad <= '1';
            ToRamWriteEnable <= '0';
            ToMdriLoad <= '0';
            ToIrLoad <= '0';
            ToMdroLoad <= '0';
```

```
        --ToAluOp <= "100";

--Reads Data in memory retrieved from Address in MAR
when ldaa_read_mem =>
--SEE INSERTED CODE BELOW
        ToALoad <= '0';
        ToPcIncrement <= '0';
        ToMarMux <= '0';
        ToMarLoad <= '0';
        ToRamWriteEnable <= '0';
        ToMdriLoad <= '0';
        ToIrLoad <= '0';
        ToMdroLoad <= '0';
        --ToAluOp <= "100";

--Loads the Memory data Register Input with data read from memory
when ldaa_load_mdri =>
--SEE INSERTED CODE BELOW
        ToALoad <= '0';
        ToPcIncrement <= '0';
        ToMarMux <= '0';
        ToMarLoad <= '0';
        ToRamWriteEnable <= '0';
        ToMdriLoad <= '1';
        ToIrLoad <= '0';
        ToMdroLoad <= '0';
        --ToAluOp <= "101";

--Loads the accumulator with data held in MDRI
when ldaa_load_a =>
--SEE INSERTED CODE BELOW
        ToALoad <= '1';
        ToPcIncrement <= '0';
        ToMarMux <= '0';
        ToMarLoad <= '0';
        ToRamWriteEnable <= '0';
        ToMdriLoad <= '0';
        ToIrLoad <= '0';
        ToMdroLoad <= '0';
        ToAluOp <= "101";

--Loads the MAR with address held in IR
when adaa_load_mar =>
```

```
--SEE INSERTED CODE BELOW
        ToALoad <= '0';
        ToPcIncrement <= '0';
        ToMarMux <= '1';
        ToMarLoad <= '1';
        ToRamWriteEnable <= '0';
        ToMdriLoad <= '0';
        ToIrLoad <= '0';
        ToMdroLoad <= '0';
        --ToAluOp <= "100";

--Reads Memory based on address in MAR
when adaa_read_mem =>
--SEE INSERTED CODE BELOW
        ToALoad <= '0';
        ToPcIncrement <= '0';
        ToMarMux <= '0';
        ToMarLoad <= '0';
        ToRamWriteEnable <= '0';
        ToMdriLoad <= '0';
        ToIrLoad <= '0';
        ToMdroLoad <= '0';
        --ToAluOp <= "100";

--Loads MDRI with data just read from memory
when adaa_load_mdri =>
--SEE INSERTED CODE BELOW
        ToALoad <= '0';
        ToPcIncrement <= '0';
        ToMarMux <= '0';
        ToMarLoad <= '0';
        ToRamWriteEnable <= '0';
        ToMdriLoad <= '1';
        ToIrLoad <= '0';
        ToMdroLoad <= '0';
        --ToAluOp <= "100";

--Loads accumulator with data in MDRI
when adaa_store_load_a =>
--SEE INSERTED CODE BELOW
        ToALoad <= '1';
        ToPcIncrement <= '0';
        ToMarMux <= '0';
```

```
                ToMarLoad <= '0';
                ToRamWriteEnable <= '0';
                ToMdriLoad <= '0';
                ToIrLoad <= '0';
                ToMdroLoad <= '0';
                ToAluOp <= "000";

        --Loads MDRO with data to be written to memory (this data comes from the
accumulator)
        when staa_load_mdro =>
        --SEE INSERTED CODE BELOW
                ToALoad <= '0';
                ToPcIncrement <= '0';
                ToMarMux <= '1';
                ToMarLoad <= '1';
                ToRamWriteEnable <= '0';
                ToMdriLoad <= '0';
                ToIrLoad <= '0';
                ToMdroLoad <= '1';
                ToAluOp <= "100";

        --Writes to memory the data stored in MDRO
        when staa_write_mem =>
        --SEE INSERTED CODE BELOW
                ToALoad <= '0';
                ToPcIncrement <= '0';
                ToMarMux <= '0';
                ToMarLoad <= '0';
                ToRamWriteEnable <= '1';
                ToMdriLoad <= '0';
                ToIrLoad <= '0';
                ToMdroLoad <= '0';
                --ToAluOp <= "100";

    end case;
end process;
end behavior;
```

Figure 6: The contents of the VHDL file “ControlUnit”

```
--Simple CPU template, This is the top level entity in your project
library ieee;
use ieee.std_logic_1164.all;

entity SimpleCPU_Template is
--These are the Outputs that can be displayed on the FPGA, More port statements may be necessary,
--Depending on how you want to display each signal to the FPGA
port (
        clk : in std_logic;
        pcOut : out std_logic_vector(7 downto 0);
        marOut : out std_logic_vector (7 downto 0);
        irOutput : out std_logic_vector (7 downto 0);
        mdriOutput : out std_logic_vector (7 downto 0);
        mdroOutput : out std_logic_vector (7 downto 0);
        aOut : out std_logic_vector (7 downto 0);
        incrementOut : out std_logic
);

end;

architecture behavior of SimpleCPU_Template is
--Initialize our memory component
component memory_8_by_32
port(   clk:            in std_logic;
        Write_Enable: in std_logic;
        Read_Addr:      in std_logic_vector     (4 downto 0);
        Data_in:        in std_logic_vector     (7 downto 0);
        Data_out:       out std_logic_vector(7 downto 0)
);
end component;
--initialize the alu
component alu
port (
        A : in std_logic_vector                 (7 downto 0);
        B : in std_logic_vector                 (7 downto 0);
        AluOp : in std_logic_vector             (2 downto 0);
        output : out std_logic_vector   (7 downto 0)
);
end component;
--initialize the registers
component reg
port (
        input : in std_logic_vector             (7 downto 0);
```

```
        output : out std_logic_vector      (7 downto 0);
        clk : in std_logic;
        load : in std_logic
);
end component;
--initialize the program counter
component ProgramCounter
port (
        increment : in std_logic;
        clk : in std_logic;
        output : out std_logic_vector      (7 downto 0)
);
end component;
--initialize the mux
component TwoToOneMux
port (
        A : in std_logic_vector                    (7 downto 0);
        B : in std_logic_vector                    (7 downto 0);
        address : in std_logic;
        output : out std_logic_vector      (7 downto 0)
);
end component;
--initialize the seven segment decoder
component sevenseg
port(
        i : in std_logic_vector(3 downto 0);
        o : out std_logic_vector(0 to 7)
);
end component;

-- initialize control unit
component ControlUnit
port (
        OpCode : in std_logic_vector(2 downto 0);
        clk : in std_logic;
        ToALoad : out std_logic;
        ToMarLoad : out std_logic;
        ToIrLoad : out std_logic;
        ToMdriLoad : out std_logic;
        ToMdroLoad : out std_logic;
        ToPcIncrement : out std_logic;
        ToMarMux : out std_logic;
        ToRamWriteEnable : out std_logic;
```

```
        ToAluOp : out std_logic_vector (2 downto 0)
);
end component;

--The following signals will be used in your port map statements, don't use the port variables in your port maps

-- Connections : Need to be sorted
signal ramDataOutToMdri : std_logic_vector (7 downto 0);

-- MAR Multiplexer connections
signal pcToMarMux : std_logic_vector(7 downto 0);
signal muxToMar : std_logic_vector      (7 downto 0);

-- RAM connections
signal marToRamReadAddr : std_logic_vector    (4 downto 0);
signal mdroToRamDataIn : std_logic_vector (7 downto 0);

-- MDRI connections
signal mdriOut : std_logic_vector          (7 downto 0);

-- IR connection
signal irOut : std_logic_vector            (7 downto 0);

-- ALU / Accumulator connections
signal aluOut: std_logic_vector  (7 downto 0);
signal aToAluB : std_logic_vector          (7 downto 0);

-- Control Unit connections
signal cuToALoad : std_logic;
signal cuToMarLoad : std_logic;
signal cuToIrLoad : std_logic;
signal cuToMdriLoad : std_logic;
signal cuToMdroLoad : std_logic;
signal cuToPcIncrement : std_logic;
signal cuToMarMux : std_logic;
signal cuToRamWriteEnable : std_logic;
signal cuToAluOp : std_logic_vector (2 downto 0);

begin

--PORT MAP STATEMENTS GO HERE
```

```
-- Create port map statements for each component in the CPU and map them to the appropriate signal defined above
-- RAM
--SEE INSERTED CODE BELOW
ramA : memory_8_by_32 port map(
        clk => clk,
        Write_Enable => cuToRamWriteEnable,
        Read_Addr => marToRamReadAddr,
        Data_in => mdroToRamDataIn,
        Data_out => ramDataOutToMdri
);

-- Accumulator
--SEE INSERTED CODE BELOW
accumA : reg port map(
        input => aluOut,
        output => aToAluB,
        clk => clk,
        load => cuToALoad
);

-- ALU
--SEE INSERTED CODE BELOW
aluA : alu port map(
        A => mdriOut,
        B => aToAluB,
        AluOp => cuToAluOp,
        output => aluOut
);

-- Program Counter
--SEE INSERTED CODE BELOW
ProgramCounterA : ProgramCounter port map(
        increment => cuToPcIncrement,
        clk => clk,
        output => pcToMarMux
);

-- Instruction Register
--SEE INSERTED CODE BELOW
InstructionRegisterA : reg port map(
        input => mdriOut,
        output => irOut,
```

```
        clk => clk,
        load => cuToIrLoad
);

-- MAR mux
--SEE INSERTED CODE BELOW
MuxA : TwoToOneMux port map(
        A => pcToMarMux,
        B => irOut,
        address => cuToMarMux,
        output => MuxToMar
);

-- Memory Access Register
--SEE INSERTED CODE BELOW
MarA : reg port map(
        input => MuxToMar,
        output(4 downto 0) => MarToRamReadAddr,
        clk => clk,
        load => cuToMarLoad
);

-- Memory Data Register Input
--SEE INSERTED CODE BELOW
MdriA : reg port map(
        input => RamDataOutToMdri,
        output => mdriOut,
        clk => clk,
        load => cuToMdriLoad
);

-- Memory Data Register Output
--SEE INSERTED CODE BELOW
MdroA : reg port map(
        input => aluOut,
        output => mdroToRamDataIn,
        clk => clk,
        load => cuToMdroLoad
);

-- Control Unit
--SEE INSERTED CODE BELOW
ControlUnitA : ControlUnit port map(
```

```
        OpCode => irOut(7 downto 5), --May have to change this
        clk => clk,
        ToALoad => cuToALoad,
        ToMarLoad => cuToMarLoad,
        ToIrLoad => cuToIrLoad,
        ToMdriLoad => cuToMdriLoad,
        ToMdroLoad => cuToMdroLoad,
        ToPcIncrement => cuToPcIncrement,
        ToMarMux => cuToMarMux,
        ToRamWriteEnable => cuToRamWriteEnable,
        ToAluOp => cuToAluOp
);

--REMAINING CODE GOES HERE
--Here is where you connect the port statement to the matching signal to display it on the FPGA
--If you want to display the signal on LED's, just set it to the port statement port<=signal;
--If you want to send the signal to the seven segment display, initialize an instance of the sevenseg
--Then map i=>signal, o=>port , keep in mind i needs to be 4 bits and o 8 bits
--pcOut <= pcToMarMux;

pcOut <= pcToMarMux;
marOut <= "000"&MarToRamReadAddr;
irOutput <= irOut;
mdriOutput <= mdriOut;
mdroOutput <= mdroToRamDataIn;
aOut <= aToAluB;
incrementOut <= cuToPcIncrement;

end behavior;
```

Figure 7: The contents of the VHDL file "SimpleCPU_Template"

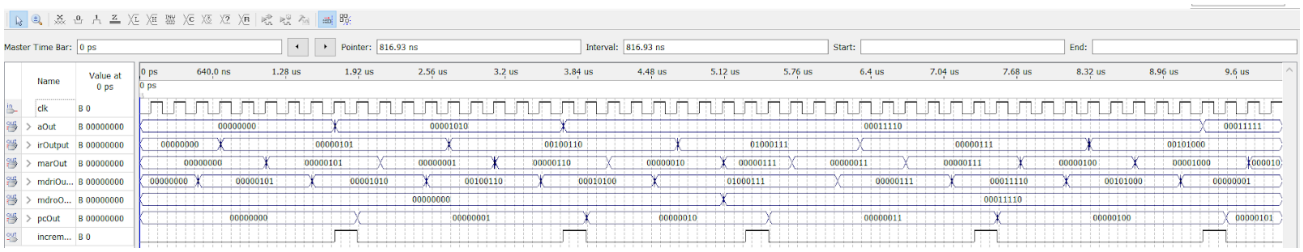


Figure 8: Waveforms Generated Using the Above Code

Finite State Machine Diagram

To better understand the functions of the CPU, a Finite State Machine (**Figure 9**) was created, which included four functions: FETCH, LOADA, ADDA, and STOREA. The state names in the Finite STate Machine came from the CU code in **Figure 6.**

The FETCH instruction took an input from the increment counter, which began by having the memory address loaded into the MAR. The memory was then read at whatever location was inputted into the MAR. The value in memory was loaded into the ALU, and a determined operation was performed (see **Figure 1**). The value is then sent to the IR, which includes the 3-bit op code needed to decide what to do with the data and the 5-bit data itself in 8 total bits. This output was read through a decoder, which decided which of the other three other instructions to send the data to.

The LOADA instruction took a 5-bit input from the decoder. This was loaded into the MAR and had the address of the data read. The data was sent to the accumulator, and then the system was incremented. The ADDA instruction took a 5-bit input from the decoder. This was loaded into the MAR and had the address of the data read. The data was sent to the accumulator, where the data and the contents of the accumulator were added. The system was then incremented. The STOREA instruction took a 5-bit input from the decoder. This was loaded into the MDRO to access the inputted address in memory. The data was then written into the memory address. Then, the system was incremented. All of the instructions fed their increments into the FETCH instruction, which continued the cycle.

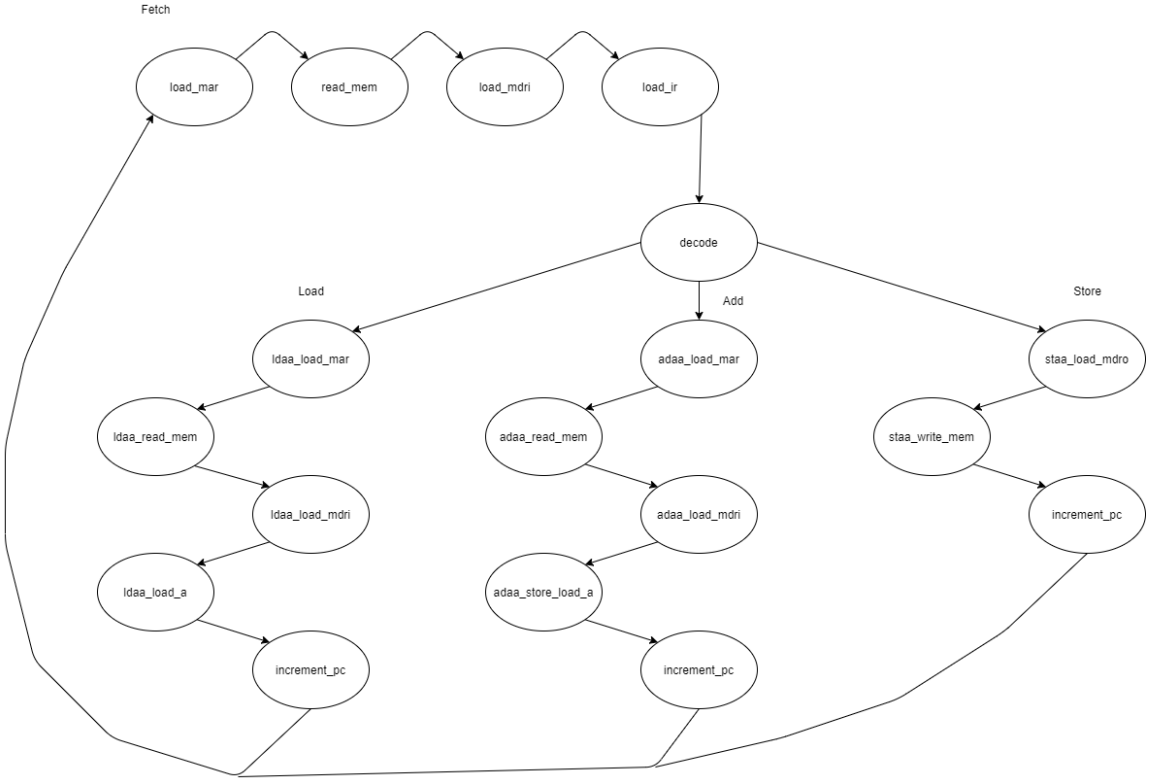


Figure 9: The Finite State Machine of the CPU

Block Diagram Components

The main portion of the diagram in **Figure 10** was the Control Unit (CU) because it fed into every other component. The CU had nine outputs to the nine components, and took one input from the IR. The ALU took inputs from the Accumulator, MDRI, and CU. It had one output that fed back into the Accumulator. The MAR took inputs from the MUX and the CU. It fed into memory, acting as the address to be read. The Accumulator took inputs from the ALU and the CU. It fed back into the ALU. The MUX took inputs from the IR, PC, and CU. It fed into the MAR. The MDRI took inputs from memory and the CU. It fed into the IR and the ALU. The MDRO took inputs from the ALU and the CU. It fed into memory, carrying the data to be written in. The PC took one input from the CU. It had one output to the MUX. The IR took an input from the MDRI and from the CU. It fed into the MUX and back into the CU. The memory took inputs from the MAR, MDRO, and CU. The MAR input acted as the wanted address in memory. The MDRO acted as the data to be written in memory. The CU acted as the switch that allowed memory to be written in the location. It had one output to the MDRI. This was used to transmit whatever data was at the specified address. Every component in the diagram was connected to a clock, but the connection lines were omitted for simplification.

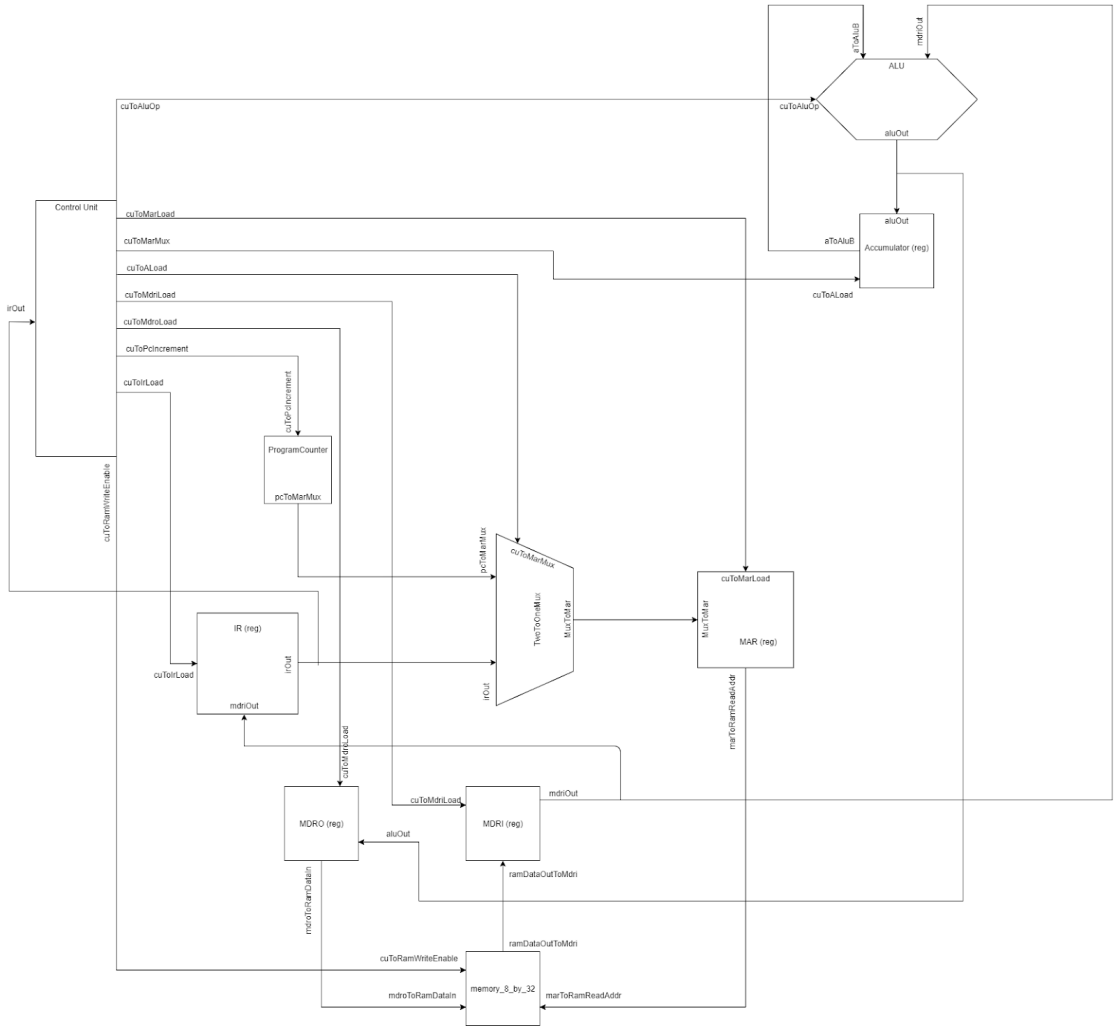


Figure 10: The Block Diagram of the CPU

Conclusion

Building the CPU was not as bad as originally thought. The most difficult part was to choose where to start, which was resolved by reading the task outline given. Once the Finite State Machine (**Figure 9**) and the Block Diagram (**Figure 10**) were created, the code seemed to fall into place on its own. The majority of the code that was left out for us to write was import statements, so the diagrams helped visualize what data was going between the different components. Because of COVID-19 and WVU closing, the project had to be completed without the Altera board and entirely tested on the Waveform Simulator. The project would have been easier if we could have been in the same room, coding on the same CPU, but the main issues occured because of small misnamings, so being in the same room might not have helped catch the errors as they happened anyways. The Waveform Simulator was confusing to test with because we did not know if it was the Simulator itself that was messing up or if it was our code. Once we found and fixed our errors, however, the Waveform output followed the example of the 100% working output. This project was really interesting because it seemed so daunting the entire semester, but ended up being moderately simple once the understanding was there (thanks to the notes given). The project helped us understand cyclical logic better due to each component needing an input from another component. Building this CPU will serve as excellent experience for future projects.