Lane Department of Computer Science and Electrical Engineering

CpE 271L: Digital Logic Laboratory
Final Project: Design of a Simple CPU

Spring 2025

Report Due By: 5/2/2025

Table of Contents

Introduction

This document outlines the theory, procedure, and results for Nathaniel Muhs and Daria Panova's final project. For this project we were asked to create a simple CPU using the information and knowledge that we have acquired throughout this semester. This project uses the DE 10-lite development board and the Quartus prime software to model the simple CPU. This project was a valuable experience to apply the knowledge that we learned this semester.

Hardware Description

This lab and project utilizes the DE10-lite prototype development board to represent the circuit and hardware. The DE10-lite board is a Field-programmable gate array chip installed in this board to model its versatility and ease of use. FPGA boards are a valuable tool in electronics prototyping due to their ability to be easily troubleshooted and reprogrammed. This ability to be easily reprogrammed means that they are perfect for this type of project. When we had a bug or mistake in our code it was easy to fix the problem and reupload to the board for efficient testing.

Software Description

Along with the hardware we needed a language to program the DE10-lite board in. For this project we chose to go with the Quartus Prime software to program the board in VHSIC Hardware Description Language. Quartus Prime was an easy software for us to choose considering its compatibility with the FPGA board. Quartus Prime has a fantastic user interface, Options, and Pin Planner that were vital for the iterative phase of this project. The only downfall of this software surfaced when our knowledge of the software was put to the test. Our training of the Quartus Prime software is extremely limited, we were never explicitly taught how to use the software and were thus expected to use resources elsewhere. I watched youtube videos to grow my knowledge of the software and its features.

Description of VHDL files

Program Counter

The program counter code is a register that holds the memory address of the next instruction to be performed. The CPE performs the instruction that is held within the program counter. Once the instruction is performed that PC is incremented and stores the next instruction to be performed. This cycle continues with the different instructions held within memory.

Reg

The register code is fundamental to this project and is utilized in many aspects of its operation. The register code is utilized by the Accumulator, Instruction Register, and Memory Address Register. All of these registers reuse the code in different applications and features. The fundamental requirements for the register to output a new value is, a positive edge trigger and load bit 1.

SevenSeg

The SevenSegment display code is utilized to display the values that the CPU computes. For the binary numbers to be displayed on the hexadecimal display they had to be converted. This code is utilized for that translation from binary to hexadecimal. Specific segments of the seven segment display were illuminated to display the hexadecimal number on the display. Or instance segments b and c were illuminated to display the number 1.

ALU

The arithmetic logic unit (ALU) was a fundamental part of this project that can not be overlooked. The ALU performs an arithmetic operation on two inputs then outputs the answer to the specific instruction. There were 4 main operations that could be performed by the ALU, addition, subtraction, bitwise OR, and Bitwise AND. The ALU operation input would specify the operation to be performed on two binary number inputs.

TwoToOneMux

This is a two to one Multiplexer that is used to determine which of two inputs is sent to the output based on the control signal. The two input values are from the Program Counter and Instruction Register. The output goes to the memory address register (MAR). The control signal is sent by the control unit of the CPU.

Memory_8_by_32

This code was written in lab number 9 of this course. This code creates a 8 by 32 Random access memory to be utilised. The RAM can either be read to write depending on if the Write_Enabel bit is higher or low. If the WE pin is high that means that the data is to be written to the RAM and stored. if the WE pin is low that means that the data at that location should be sent to the output of the Read_addr.

Control Unit

If the CPU is considered the brain of a computer the Control Unit would be the brain of the CPU. The control unit is best described as a state machine that executes a specific set of instructions given a specific flow. The three instructions that can be performed by the control unit are loadA, addA, storeA. These three steps are performed in this order to insure the reliability of the CPU.

CPU

The CPU file is where all of this code comes together. All of the other files demonstrate small functions of the overall CPU, the CPU file brings all of that together in a cohesive file. The first bit of our code are component statements of the smaller parts like the pc and alu. The next big chunk are the signal initializations. signals were declared as they can be used as both input and output later on. The following code that is highlighted in green is the code that we completed. This section of code is the port mapping statements. The final section of our CPU code is mapping the outputs to potential display segments.

Problems Occurred & Solutions

During development of the Simple CPU, we ran into several problems in the VHDL and control‑FSM. Below are the two most impactful issues and how we resolved them.

Firstly, RAM never saw the correct address. Every fetch returned the same data (or “X” in simulation), regardless of PC value. We had a full 8-bit marRegOut register but never routed its low five bits into the RAM’s Read_Addr port. The signal marToRamReadAddr stayed at its default “00000.”

As a solution, we inserted marRegOut <= irOut(7 downto 5) &marToRamReadAddr to ensure all bits are properly used and are not overflowing at any instance. Once we placed it, marOut matched the PC or IR address as expected, and memory fetches began returning the correct words.

The other problem was that FSM would not start in the fetch state. In simulation the control unit sat in an undefined state and never asserted ToPcIncrement. We forgot to initialize current_state so our state machine began in the first enumerated type (which happened to be load_mar) instead of increment_pc.

Solution was to set signal current_state : cu_state_type := increment_pc; in ControlUnit. This ensures the very first micro-cycle pulses ToPcIncrement and kicks off the fetch sequence.

Competed Code

ALU 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;
```

memory_8_by_32 code

```
-- 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","00100011","01000111","00000111","00101000","000
00110","00010100","00001101","00000001","10110100","10001010","1010101
0","10101001","00000000","10100101","01010101","10101110","10110100","
10001010","10101010","10101001","00000000","10100101","01010101","1010
1110","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;
```

ProgramCounter code

```
--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
--INSERT CODE HERE
        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
        counter := counter + 1;

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

reg code

```
--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;
```

sevenseg code

```
--Seven Segment Display, keep in mind the output here is 8 bits to
match
--The CPU component outputs, when connecting the pins, ignore the MSB
of o or o(7)
library ieee;
use ieee.std_logic_1164.all;

entity sevenseg is
port(
     i : in std_logic_vector(3 downto 0);
     o : out std_logic_vector(7 downto 0)
);
end sevenseg;

architecture logic of sevenseg is
begin
o <= "00000001" when i="0000" else
     "01001111" when i="0001" else
     "00010010" when i="0010" else
     "00000110" when i="0011" else
     "01001100" when i="0100" else
     "00100100" when i="0101" else
     "00100000" when i="0110" else
     "00001111" when i="0111" else
     "00000000" when i="1000" else
     "00000100" when i="1001" else
     "00001000" when i="1010" else
     "01100000" when i="1011" else
     "00110001" when i="1100" else
     "01000010" when i="1101" else
     "00110000" when i="1110" else
     "00111000" when i="1111";
end;
```

TwoToOneMux code

```
--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;
```

ControlUnit code

```
-- 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 := increment_pc;

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
            when increment_pc        => current_state <= load_mar;

            -- FETCH sequence
            when load_mar            => 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 next instruction
            when decode =>
                if    OpCode = "000" then current_state <= ldaa_load_mar;   -- LOADA
                elsif OpCode = "001" then current_state <= adaa_load_mar;   -- ADDA
                elsif OpCode = "010" then current_state <= staa_load_mdro;  -- STOREA
                else                      current_state <= increment_pc;    -- NOP
                end if;

            -- LOADA micro-steps
            when ldaa_load_mar       => current_state <= ldaa_read_mem;
            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;

            -- ADDA micro-steps
            when adaa_load_mar       => current_state <= adaa_read_mem;
            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;

            -- STOREA micro-steps
            when staa_load_mdro      => current_state <= staa_write_mem;
            when staa_write_mem      => current_state <= increment_pc;

            when others              => current_state <= increment_pc;
        end case;
    end if;
end process;

-- Defines what happens at each state, set to '1' if we want that component on
process(current_state)
```

```
begin
    -- defaults
    ToALoad          <= '0';
    ToMarLoad         <= '0';
    ToIrLoad         <= '0';
    ToMdriLoad        <= '0';
    ToMdroLoad        <= '0';
    ToPcIncrement      <= '0';
    ToMarMux          <= '0';
    ToRamWriteEnable   <= '0';
    ToAluOp           <= "000";

    case current_state is
        -- increment PC
        when increment_pc =>
            ToPcIncrement <= '1';

        -- MAR ← PC
        when load_mar =>
            ToMarLoad     <= '1';

        -- memory read (no control outputs)
        when read_mem =>
            null;

        -- MDRI ← memory
        when load_mdri =>
            ToMdriLoad    <= '1';

        -- IR ← MDRI
        when load_ir =>
            ToIrLoad      <= '1';

        -- decode (all off)
        when decode =>
            null;

        -- LOADA: MAR ← IR
        when ldaa_load_mar =>
            ToMarMux      <= '1';
            ToMarLoad     <= '1';

        -- LOADA: memory read
        when ldaa_read_mem =>
        null;

      -- LOADA: MDRI ← memory
      when ldaa_load_mdri =>
        ToMdriLoad    <= '1';

      -- LOADA: A ← MDRI
      when ldaa_load_a =>
        ToALoad       <= '1';
        ToAluOp       <= "101";  -- pass-through A

      -- ADDA: MAR ← IR
      when adaa_load_mar =>
        ToMarMux      <= '1';
        ToMarLoad     <= '1';

      -- ADDA: memory read
      when adaa_read_mem =>
        null;

      -- ADDA: MDRI ← memory
      when adaa_load_mdri =>
        ToMdriLoad    <= '1';

      -- ADDA: A ← A + MDRI
      when adaa_store_load_a =>
        ToALoad       <= '1';
        ToAluOp       <= "000";  -- ADD

      -- STOREA: MDRO ← A, MAR ← IR
      when staa_load_mdro =>
        ToMarMux          <= '1';
        ToMarLoad         <= '1';
        ToMdroLoad        <= '1';
        ToAluOp           <= "100"; -- pass-through B

      -- STOREA: write memory
      when staa_write_mem =>
        ToRamWriteEnable <= '1';

    when others =>
        null;
    end case;
  end process;
```

```
end behavior;
```

CPU code

```
--Simple CPU template – this is the top-level entity
library ieee;
use ieee.std_logic_1164.all;

entity SimpleCPU_Template is
   -- These are the outputs you might tie to LEDs / 7-seg on the DE10-Lite
   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
   ----------------------------------------------------------------
   --  component declarations (memory, alu, reg, pc, mux, sevenseg,
   --  control-unit) – these match the individual .vhd files you'll add
   ----------------------------------------------------------------
   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;

   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;

   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;

component ProgramCounter
   port ( increment : in  std_logic;
        clk       : in  std_logic;
        output    : out std_logic_vector(7 downto 0) );
end component;

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;

component sevenseg
   port ( i : in  std_logic_vector(3 downto 0);
        o : out std_logic_vector(7 downto 0) );
end component;

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;

------------------------------------------------------------------
--  internal signals (wires between components)
------------------------------------------------------------------
-- memory
signal ramDataOutToMdri : std_logic_vector(7 downto 0);
```

```
    -- MAR multiplexer
    signal pcToMarMux : std_logic_vector(7 downto 0);
    signal muxToMar   : std_logic_vector(7 downto 0);

    -- RAM
    signal marToRamReadAddr : std_logic_vector(4 downto 0);
    signal mdroToRamDataIn  : std_logic_vector(7 downto 0);
    -- full 8-bit MAR register output
    signal marRegOut        : std_logic_vector(7 downto 0);

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

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

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

    -- Control-unit control lines
    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 <<<
      ------------------------------------------------------------------


    -- RAM
    RAM0 : memory_8_by_32
        port map (
            clk           => clk,
            Write_Enable  => cuToRamWriteEnable,
            Read_Addr     => marToRamReadAddr,
            Data_in       => mdroToRamDataIn,
            Data_out      => ramDataOutToMdri
```

```
	);

    -- Accumulator register
    REG_ACC : reg
	port map (
	    input  => aluOut,
	    output => aToAluB,
	    clk    => clk,
	    load   => cuToALoad
	);

    -- ALU
    ALU0 : alu
	port map (
	    A      => mdriOut,
	    B      => aToAluB,
	    AluOp  => cuToAluOp,
	    output => aluOut
	);

    -- Program Counter
    PC0 : ProgramCounter
	port map (
	    increment => cuToPcIncrement,
	    clk       => clk,
	    output    => pcToMarMux
	);

    -- Instruction Register
    REG_IR : reg
	port map (
	    input  => mdriOut,
	    output => irOut,
	    clk    => clk,
	    load   => cuToIrLoad
	);

    -- MAR multiplexer
    MUX_MAR : TwoToOneMux
	port map (
	    A       => pcToMarMux,
	    B       => irOut,
	    address => cuToMarMux,
	    output  => muxToMar
    );

  -- MAR register
  REG_MAR : reg
    port map (
        input  => muxToMar,
        output(4 downto 0) => marToRamReadAddr,
        clk    => clk,
        load   => cuToMarLoad
    );

  -- MDRI register
  REG_MDRI : reg
    port map (
        input  => ramDataOutToMdri,
        output => mdriOut,
        clk    => clk,
        load   => cuToMdriLoad
    );

  -- MDRO register
  REG_MDRO : reg
    port map (
        input  => aToAluB,
        output => mdroToRamDataIn,
        clk    => clk,
        load   => cuToMdroLoad
    );

  -- Control Unit
  CU0 : ControlUnit
    port map (
        OpCode          => irOut(7 downto 5),
        clk             => clk,
        ToALoad         => cuToALoad,
        ToMarLoad       => cuToMarLoad,
        ToIrLoad        => cuToIrLoad,
        ToMdriLoad      => cuToMdriLoad,
        ToMdroLoad      => cuToMdroLoad,
        ToPcIncrement   => cuToPcIncrement,
        ToMarMux        => cuToMarMux,
        ToRamWriteEnable => cuToRamWriteEnable,
        ToAluOp         => cuToAluOp
    );
    ------------------------------------------------------------------
    --  Any optional display connections (LEDs / seven-seg) can be
    --  added below.  Example lines are commented out:
    ------------------------------------------------------------------
    pcOut        <= pcToMarMux;
    irOutput     <= irOut;
    aOut         <= aToAluB;
    marOut       <= marRegOut;
    mdriOutput   <= mdriOut;
    mdroOutput   <= mdroToRamDataIn;
    incrementOut <= cuToPcIncrement;

    marRegOut <= irOut(7 downto 5) &marToRamReadAddr;

end behavior;
```

Finite State Machine

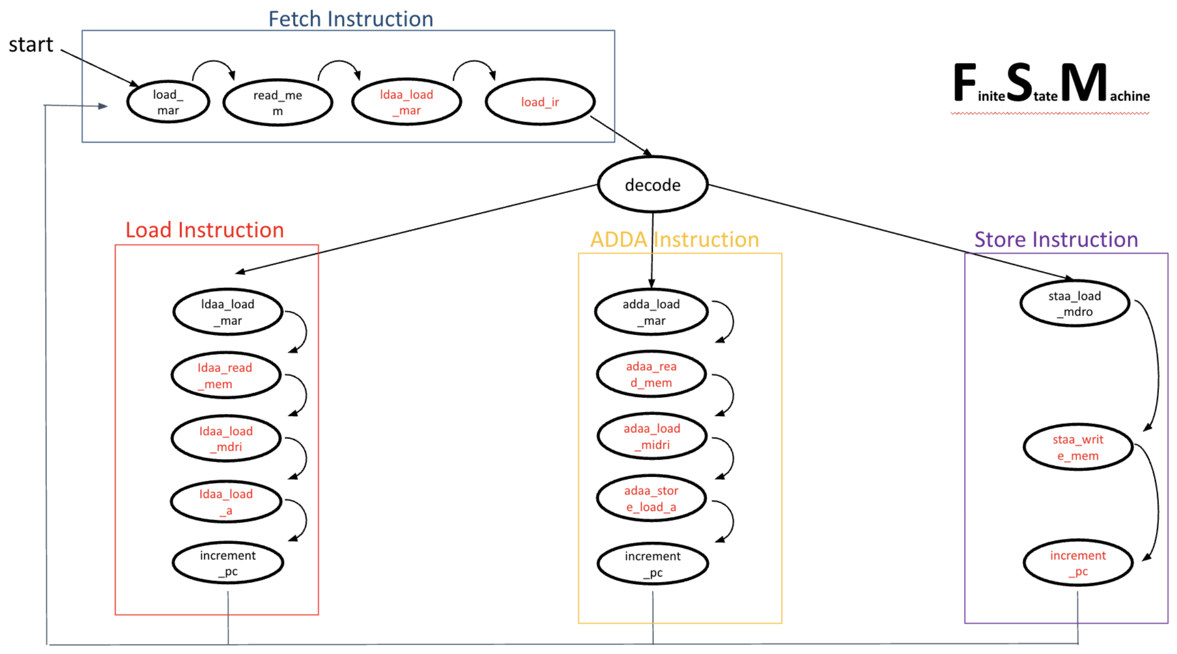


Figure #1: Finite State Machine

Block Diagram

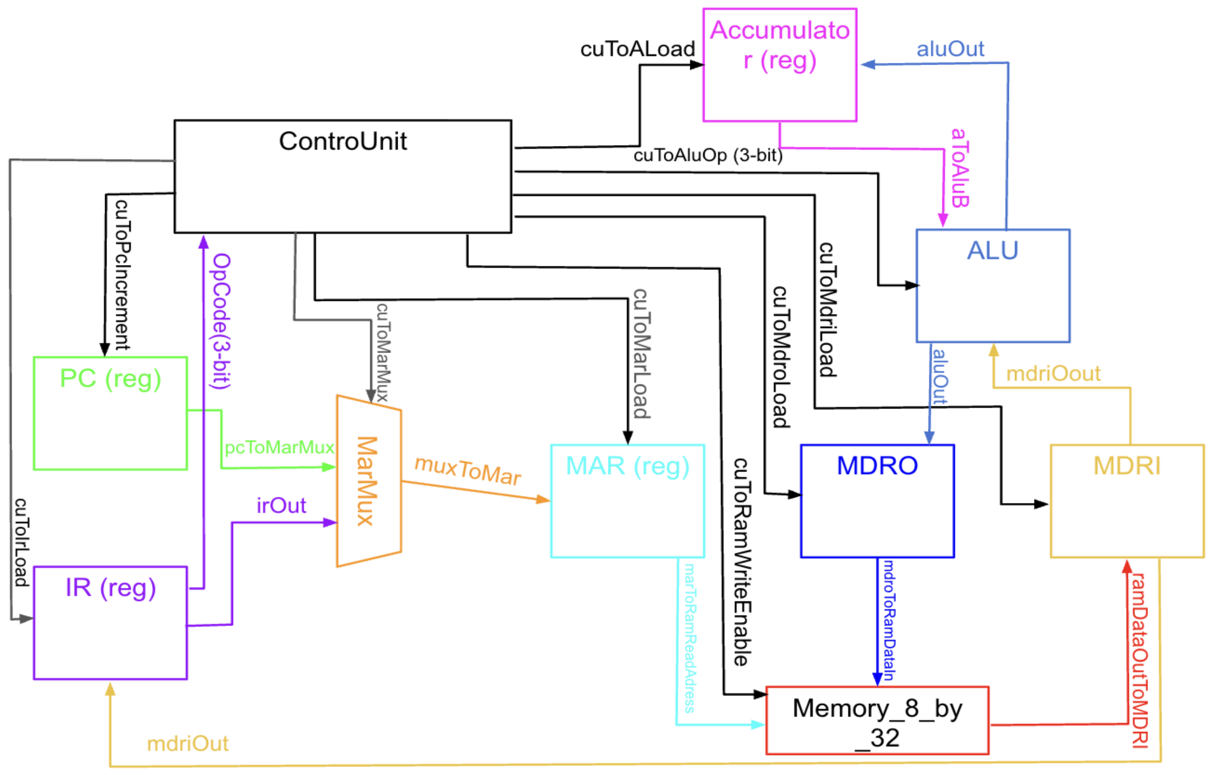
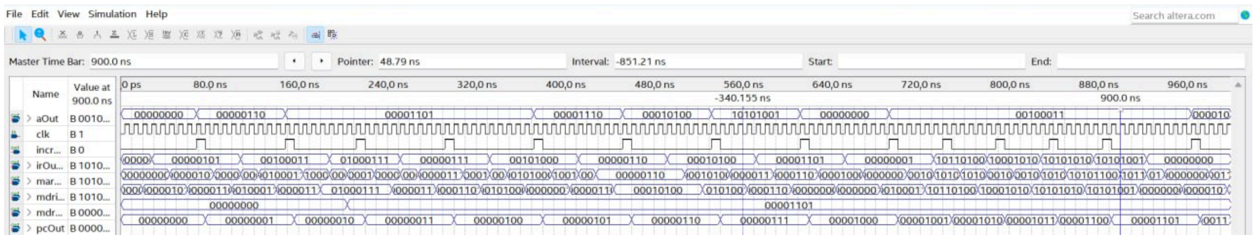


Figure #2: Block Diagram

Waveform snip



Master Time Bar: 900.0 ns

Pointer: 48.79 ns

Interval: -851.21 ns

Start:

End:

Name	Value at 900.0 ns
aOut	B 0010...
clk	B 1
incr...	B 0
irOu...	B 1010...
mar...	B 1010...
mdri...	B 1010...
mdr...	B 0000...
pcOut	B 0000...

Figure #3: Waveform snip

Waveform Result Discussion

In the 10 ns-clock simulation, the CPU cleanly steps through LOADA 5, ADDA 3, and STOREA 7 without any spurious signals. Each instruction begins with five fetch cycles—PC increments, MAR←PC, MDRI←MEM, IR←MDRI, decode—then the execute micro-steps: LOADA reads address 5 and loads A=6; ADDA reads address 3 and updates A→13; STOREA drives MDRO=13, pulses the write enable, and returns to fetch. Throughout, pcOut, marOut, mdriOutput, irOutput, and aOut update exactly as expected, and incrementOut pulses once per instruction, confirming correct implementation of our FSM and datapath.

Conclusion

This project was an excellent conclusion to the material that we have been learning in the lab this semester. In past labs we followed step by step instructions provided by the lab TA but this was a much more hands off experience. It was reassuring to see that my partner and I could implement an entire daunting CPU by ourselves. I feel like this project is a valuable experience that I can share with an employer. I feel like it would have been valuable to learn more about how to use the VHDL software prior to this lab project. We were never explicitly taught how to use the software by anything other than the labs. I learned material in class and that was something we were never explicitly taught. Overall I thoroughly enjoyed the material and concepts presented in this class and feel that it has taught me a specific career path that I may like to follow.