**CPE 272**

**Digital Logic Design Laboratory**

**Simple CPU Design**

**Fall 2020**

Trey McAtee

12/4/20

**Introduction**

**Hardware -** (DE10-Lite)

The hardware used in throughout the labs was the DE10-Lite device. The board provided students with a tangible way to test the code they had written in the Quartus Prime software. The purpose of the board was to execute whatever program had been compiled for it, allowing for user input via physical switches on the board while also being able to show the results of the current program configuration on the board via LEDs just above the switches. The LEDs were used throughout nearly every lab project in the course, showing the accuracy of code produced.

**Software -** (Quartus Prime)

To create the code for my project, I used the Quartus Prime Program. This program allowed me to create the necessary project files as well as compile VHDL code to be then used by the board. The program has error troubleshooting functionality as well for code. The program also has a built in pin planner for the board used throughout the semester, which allowed students to program the specific switches and LEDs they would want to use for their representation of the code on the board. Finally, Quartus Prime has a built in waveform simulator that can virtually simulate the results of code with certain inputs. I will use the waveform simulator to test my project as well, as it is needed because I no longer have the board.

**VHDL Files**

Cpu - The cpu file is the highest hierarchy file. It imports all other files into the respective components and handles communication between the components.

ALU - The ALU (arithmetic logic unit) can add, subtract, and perform logical operations on given inputs.

Control Unit - The control unit basically coordinates every CPU operation. The control unit enables any piece of the CPU that needs to be active at that moment. It also performs all state transitions.

Memory Array – This is just an 8 x 32 array that has addresses containing memory values.

Mux - This creates a shared connection between the PC and the IR to the MAR. Depending on the inputs, it will also send the outputs A or B.

Program Counter – Will increment the program by one upon receiving a positive edge trigger from the clock input.

Register – This simulates many different components within the CPU. It will assign the output to the input on a positive clock when a load is applied. The register stores 8-bit code.

**Problems and Solutions**

I encountered my first error upon compiling my program for the first time. I honestly have no idea if I simply forgot to change it from the default or misclicked somehow, but I was set to use the wrong family. I simply switched the device to use back to the MAX 10M50DAF484C7G.

I encountered multiple small syntax errors throughout my code as well and used the built-in compiler from Quartus to guide me in fixing them.

Finally, I had an issue with a few of my variables, I was setting them to incorrect values. I think I was trying to go to quickly on the section, because it was an easy fix. The corrected version is thus:

“ToALoad => cuToALoad,

ToMarLoad => cuToMarLoad,

ToIrLoad => cuToIrLoad,

“

And my original mistake was simply setting all three of those to cuToALoad.

**Completed 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;

**-- 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 =>

**--INSERT CODE HERE**

**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 =>**

**--INSERT CODE HERE**

**if OpCode = "000" then**

**current\_state <= ldaa\_load\_mar;**

**elsif OpCode = "001" then**

**current\_state <= adaa\_load\_mar;**

**elsif OpCode = "010" then**

**current\_state <= staa\_load\_mdro;**

**else**

**current\_state <= increment\_pc;**

**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;**

**--INSERT CODE HERE**

**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;**

**--INSERT CODE HERE**

**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 =>**

**current\_state <= staa\_write\_mem;**

**when staa\_write\_mem =>**

**current\_state <= increment\_pc;**

--INSERT CODE HERE

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 =>**

**ToALoad <= '0';**

**ToPcIncrement <= '0';**

**ToMarMux <= '0';**

**ToMarLoad <= '1';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '0';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '0';**

**ToAluOp <= "000";**

**--INSERT CODE HERE**

**--Reads Address located in MAR**

**when read\_mem =>**

**ToALoad <= '0';**

**ToPcIncrement <= '0';**

**ToMarMux <= '0';**

**ToMarLoad <= '0';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '0';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '0';**

**ToAluOp <= "000";**

**--INSERT CODE HERE**

**--Load Memory Data Register Input**

**when load\_mdri =>**

**ToALoad <= '0';**

**ToPcIncrement <= '0';**

**ToMarMux <= '0';**

**ToMarLoad <= '0';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '1';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '0';**

**ToAluOp <= "000";**

**--INSERT CODE HERE**

**--Loads the Instruction Register with instruction fetched from Memory**

**when load\_ir =>**

**ToALoad <= '0';**

**ToPcIncrement <= '0';**

**ToMarMux <= '0';**

**ToMarLoad <= '0';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '0';**

**ToIrLoad <= '1';**

**ToMdroLoad <= '0';**

**ToAluOp <= "000";**

**--INSERT CODE HERE**

**--Decodes The current instruction (everything should be off for this)**

**when decode =>**

**ToALoad <= '0';**

**ToPcIncrement <= '0';**

**ToMarMux <= '0';**

**ToMarLoad <= '0';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '0';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '0';**

**ToAluOp <= "000";**

**--INSERT CODE HERE**

**--Loads the MAR with address stored in IR**

**when ldaa\_load\_mar =>**

**ToALoad <= '0';**

**ToPcIncrement <= '0';**

**ToMarMux <= '1';**

**ToMarLoad <= '1';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '0';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '0';**

**ToAluOp <= "101";**

**--INSERT CODE HERE**

**--Reads Data in memory retrieved from Address in MAR**

**when ldaa\_read\_mem =>**

**ToALoad <= '0';**

**ToPcIncrement <= '0';**

**ToMarMux <= '0';**

**ToMarLoad <= '0';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '0';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '0';**

**ToAluOp <= "101";**

**--INSERT CODE HERE**

**--Loads the Memory data Register Input with data read from memory**

**when ldaa\_load\_mdri =>**

**ToALoad <= '0';**

**ToPcIncrement <= '0';**

**ToMarMux <= '0';**

**ToMarLoad <= '0';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '1';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '0';**

**ToAluOp <= "101";**

**--INSERT CODE HERE**

**--Loads the accumulator with data held in MDRI**

**when ldaa\_load\_a =>**

**ToALoad <= '1';**

**ToPcIncrement <= '0';**

**ToMarMux <= '0';**

**ToMarLoad <= '0';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '0';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '0';**

**ToAluOp <= "101";**

**--INSERT CODE HERE**

**--Loads the MAR with address held in IR**

**when adaa\_load\_mar =>**

**ToALoad <= '0';**

**ToPcIncrement <= '0';**

**ToMarMux <= '1';**

**ToMarLoad <= '1';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '0';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '0';**

**ToAluOp <= "000";**

**--INSERT CODE HERE**

**--Reads Memory based on address in MAR**

**when adaa\_read\_mem =>**

**ToALoad <= '0';**

**ToPcIncrement <= '0';**

**ToMarMux <= '0';**

**ToMarLoad <= '0';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '0';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '0';**

**ToAluOp <= "000";**

**--INSERT CODE HERE**

**--Loads MDRI with data just read from memory**

**when adaa\_load\_mdri =>**

**ToALoad <= '0';**

**ToPcIncrement <= '0';**

**ToMarMux <= '0';**

**ToMarLoad <= '0';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '1';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '0';**

**ToAluOp <= "000";**

**--INSERT CODE HERE**

**--Loads accumulator with data in MDRI**

**when adaa\_store\_load\_a =>**

**ToALoad <= '1';**

**ToPcIncrement <= '0';**

**ToMarMux <= '0';**

**ToMarLoad <= '0';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '0';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '0';**

**ToAluOp <= "000";**

**--INSERT CODE HERE**

**--Loads MDRO with data to be written to memory (this data comes from the accumulator)**

**when staa\_load\_mdro =>**

**ToALoad <= '0';**

**ToPcIncrement <= '0';**

**ToMarMux <= '1';**

**ToMarLoad <= '1';**

**ToRamWriteEnable <= '0';**

**ToMdriLoad <= '0';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '1';**

**ToAluOp <= "100";**

**--INSERT CODE HERE**

**--Writes to memory the data stored in MDRO**

**when staa\_write\_mem =>**

**ToALoad <= '0';**

**ToPcIncrement <= '0';**

**ToMarMux <= '0';**

**ToMarLoad <= '0';**

**ToRamWriteEnable <= '1';**

**ToMdriLoad <= '0';**

**ToIrLoad <= '0';**

**ToMdroLoad <= '0';**

**ToAluOp <= "100";**

**--INSERT CODE HERE**

**end case;**

end process;

end behavior;

**--CPU**

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;

--inialize 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**

**--INSERT CODE HERE**

**RAM: memory\_8\_by\_32 port map (**

**clk => clk,**

**Write\_Enable => cuToRamWriteEnable,**

**Read\_Addr => marToRamReadAddr,**

**Data\_in => mdroToRamDataIn,**

**Data\_out => ramDataOutToMdri**

**);**

**-- Accumulator**

**--INSERT CODE HERE**

**Acc: reg port map (**

**input => aluOut,**

**output => aToAluB,**

**clk => clk,**

**load => cuToALoad**

**);**

**-- ALUs**

**--INSERT CODE HERE**

**aluPort: alu port map (**

**A => mdriOut,**

**B => aToAluB,**

**AluOp => cuToAluOp,**

**output => aluOut**

**);**

**-- Program Counter**

**--INSERT CODE HERE**

**ProgCount: ProgramCounter port map (**

**increment => cuToPcIncrement,**

**clk => clk,**

**output => pcToMarMux**

**);**

**-- Instruction Register**

**--INSERT CODE HERE**

**IR: reg port map (**

**input => mdriOut,**

**output => irOut ,**

**clk => clk,**

**load => cuToIrLoad**

**);**

**-- MAR mux**

**--INSERT CODE HERE**

**Mux: TwoToOneMux port map (**

**A => pcToMarMux,**

**B => irOut,**

**address => cuToMarMux,**

**output => muxToMar**

**);**

**-- Memory Access Register**

**--INSERT CODE HERE**

**mar: reg port map (**

**input => muxToMar,**

**output => marToRamReadAddr,**

**clk => clk,**

**load => cuToMarLoad**

**);**

**-- Memory Data Register Input**

**--INSERT CODE HERE**

**MDRI: reg port map (**

**input => ramDataOutToMdri,**

**output => mdriOut,**

**clk => clk,**

**load => cuToMdriLoad**

**);**

**-- Memory Data Register Output**

**--INSERT CODE HERE**

**mdro: reg port map (**

**input => aluOut,**

**output => mdroToRamDataIn,**

**clk => clk,**

**load => cuToMdroLoad**

**);**

**-- Control Unit**

**--INSERT CODE HERE**

**CU: ControlUnit port map (**

**OpCode => irout(7 downto 5), --to only get the Op portion**

**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;**

**aOut <= aToAluB;**

**irOutput <= irout;**

end behavior;

**-- 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;

**--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)

--INSERT CODE HERE

**Output <= conv\_std\_logic\_vector(counter,8);**

end if;

end process;

end behavior;

**--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;

-**-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

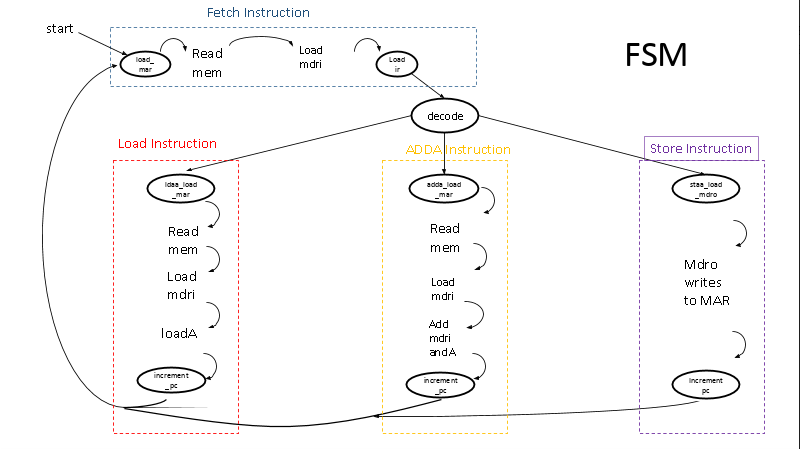
output <= B;

end if;

end process;

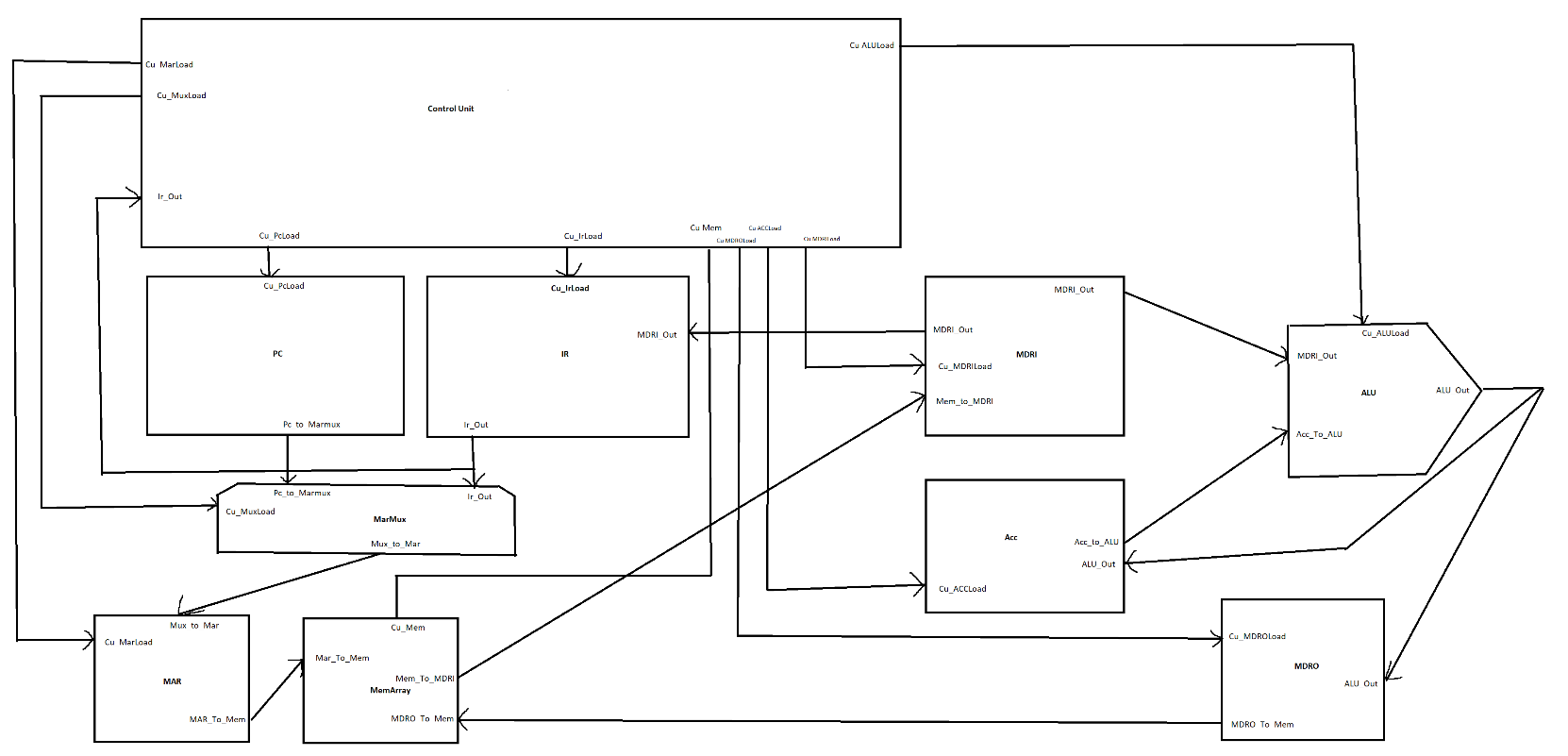
end behavior;

**State Machine Diagram**

****

(Figure 1: FSM Diagram)

**Component Block Diagram**



(Figure 2: Component Block Diagram)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Memory Data | | | | | |
| Ram | Binary | OpCode | Address | Value | Accumulator |
| 0 | 00000101 | 000-loadA | 00101-5 | 10 | 10 |
| 1 | 00100110 | 001-addA | 00110-6 | 20 | 30 |
| 2 | 01000111 | 010-storeA | 00111-7 | 30 | - |
| 3 | 00000111 | 000-loadA | 00111-7 | 30 | 30 |
| 4 | 00101000 | 001-addA | 01000-8 | 1 | 31 |
| 5 | 00001010 | 000-loadA | 01010-10 | - | - |
| 6 | 00010100 | 000-loadA | 10100-20 | - | - |
| 7 | 00011110 | 000-loadA | 11110-30 | - | - |
| 8 | 00000001 | 000-loadA | 00001-1 | 5 | 5 |

**Conclusion**

I completed all tasks for this project successfully in the end and used the given outline for this report to create it. I went off track on one thing, which was including the table. I was admittedly surprised that this report did not want to have the waveform simulator results posted in, however. My finished code was able to perform the add, load, and store functions on values given in the 8 by 32 memory array based upon the operation code. This project introduced me to a higher level of problem solving by having me learn about the functions of a CPU, and the way each component works together to achieve the overall goal.