Name: Temitayo Aderounmu

UTA ID: 1001568524

CSE 2441-001 Spring 2022 Term Project

# Project Report

Instructor: Dr. Bill Carroll

TA: Khaled Ahmed

Due Date: 05/02/2022

"I __Temitayo Aderounmu__ did not give or receive any assistance on this project, and the report submitted is wholly my own."

X [illegible]

## Contents

CSE 2441 Term Project: Introduction

## INTRODUCTION

Project Overview:

The term project involves the design and implementation task that is based on the knowledge we have gained throughout the semester in the lecture, lab sessions and homework. The main task is for us to incorporates different and various parts into one to make the TRISC processor. The different components have been performed in the previous labs. The different components include the Random-Access Memory [RAM], the program counter [PC], the Accumulator [ACC], the Arithmetic Logic Unit [ALU], the Instruction Register [IR], and the Control Unit [CU]. This many parts incorporated into one, with carefully specified inputs and outputs made up the whole project.

Project Requirements:

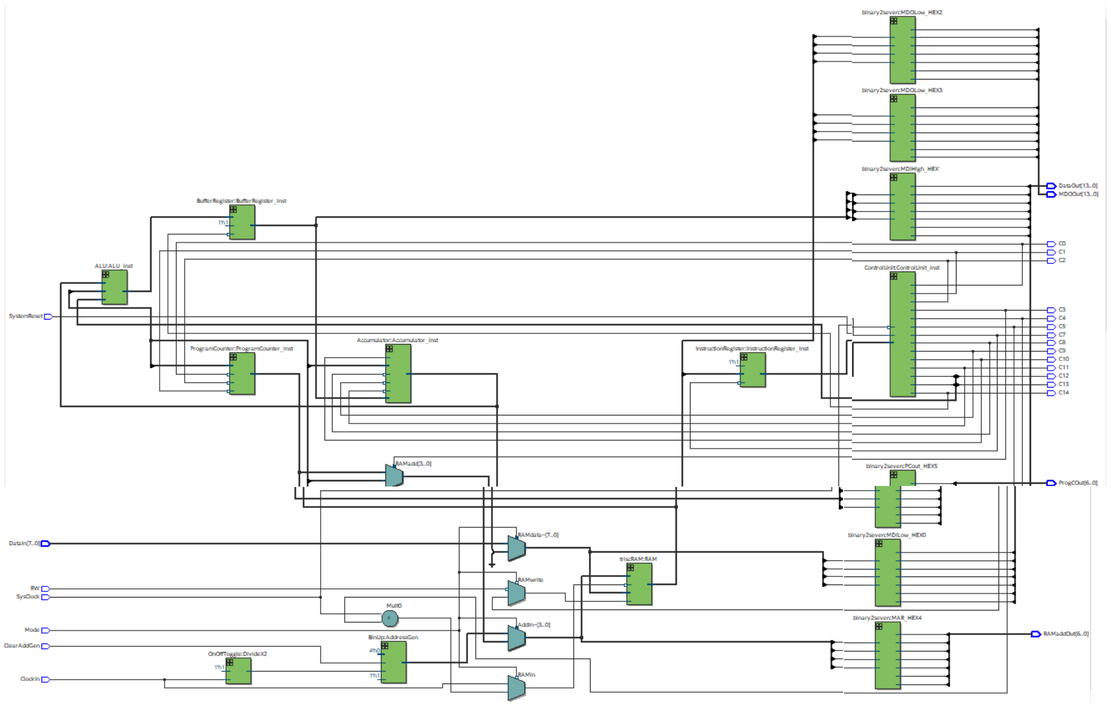
For the project, we have to have a completed design, that is simulated, implemented and tested. We also have to make a project report, which has to include a cover sheet, Table of contents, Introduction, System design, Controller design details, test results and conclusion. For the project, before testing the TRISC design and realization, we first have to load a given program into the RAM, which is then incorporated into the TRISC Organization.

Project Status:

The Project was completed on Friday, the 29th of April 2022, and was signed off by Khaled as being completed. The design met all specifications, and my code executed properly and correctly.

## SYSTEM DESIGN DETAILS:

System Level description and diagrams showing input and output.



This figure shows the RTL diagram of the whole TRISC project obtained from the Netlist Viewer option on the Tool tab of Quartus prime.

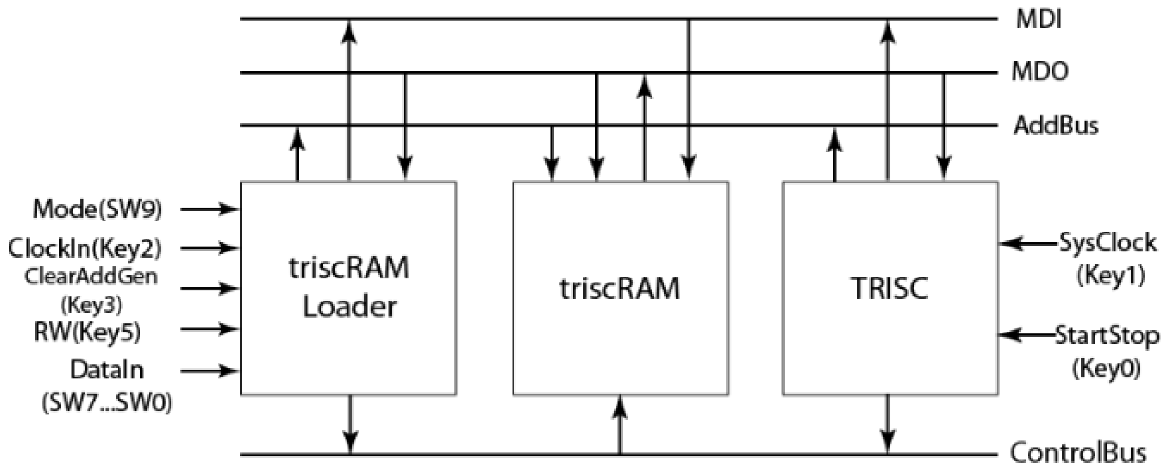


Figure 1 – triscRAM Loader (TRISC mode: SW9=0, Loader mode: SW9=1)

This figure shows the incorporation of the triscRAM loader into the TRISC organization processor. Before testing the TRISC design and realization, we first have to load the triscRAM with the given inputs:

*0*: 0F, *1*: 61, *2*: 62, *3*: 1E, *4*: 74, *5*: 0E, *6*: 66, *7*: 89, *8*: 88, *9*: 69, *A*: 2E, *B*: 7B, *C*: 6C, *D*: 88, *E*: EE, *F*: FF

The loaded data is then sent into the RAM associated with the main TRISC organization unit, and the information is used to fetch instructions and perform them.

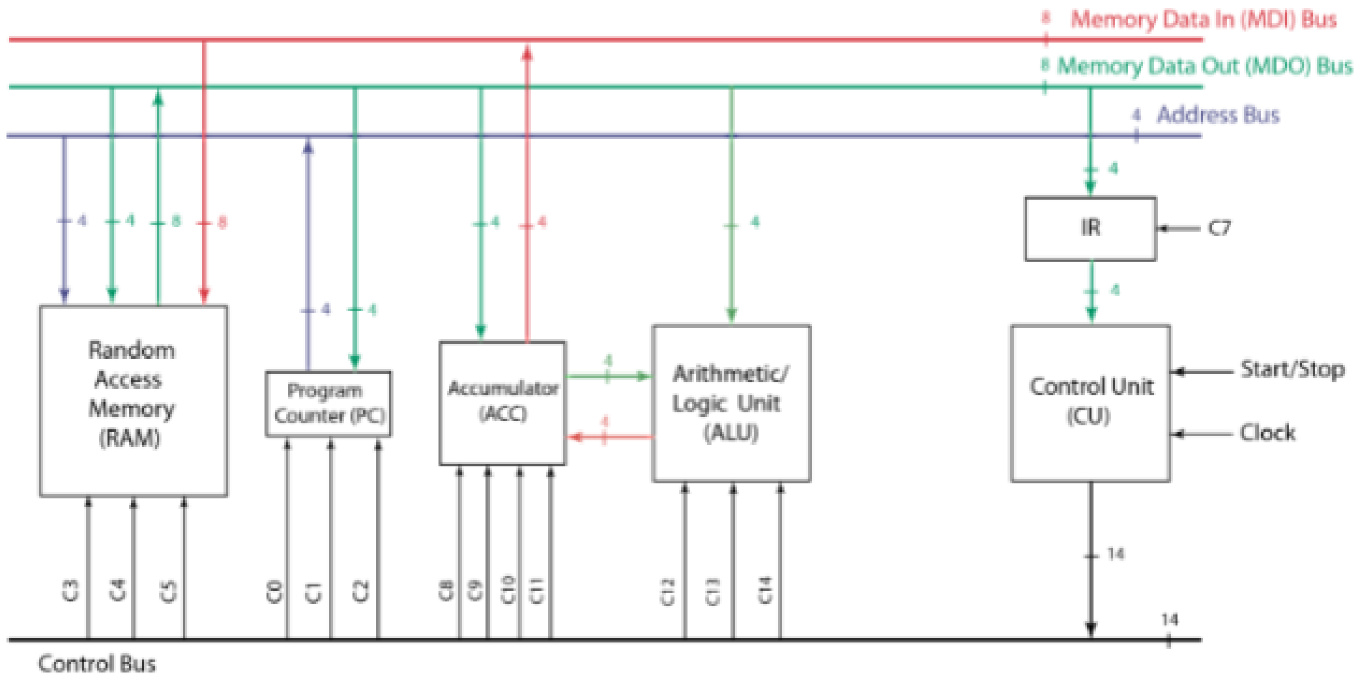
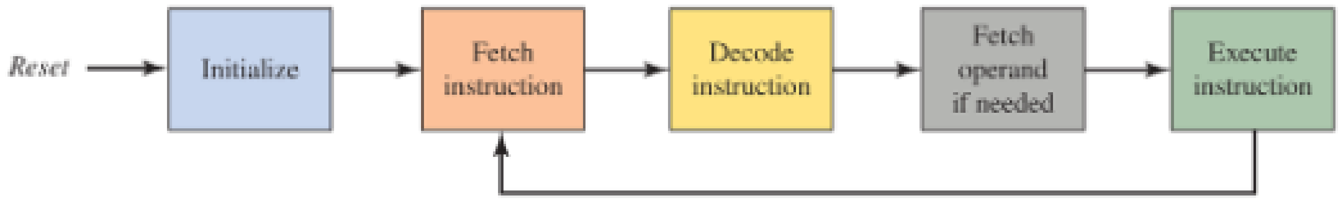


Figure 2 – TRISC for INC. CLR. LDA. STA. ADD. and JMP.

The TRISC organization is a machine that encodes integers parsed into it. The arithmetic logic unit incorporated into the unit, can perform several operations including, Add, Subtract, And, and XOR The registers also have an instruction set including it being cleared or incremented. The program counter holds the address of memory where the instructions that we want to fetch is.

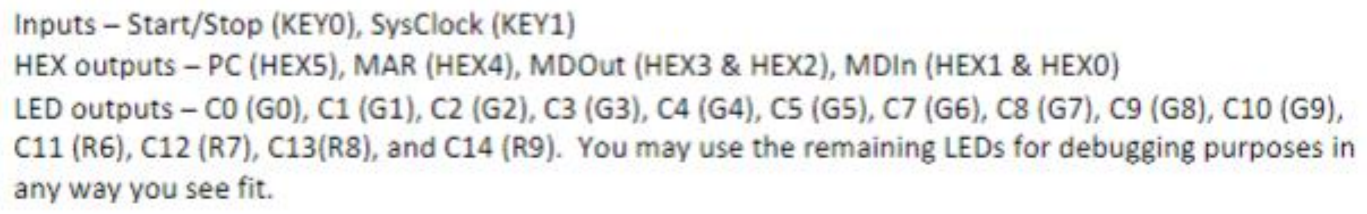
Figure 8.7 TRISC4 instruction cycle



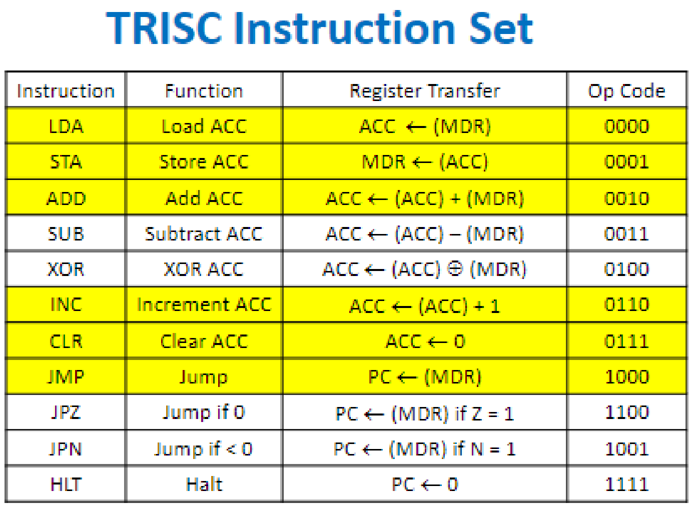
This instruction cycle shows the necessary steps needed for the TRISC organization machine to fetch from memory and execute any given instruction.

Fetch means to read instruction from memory. And once we read it from the RAM, it comes out of the memory and goes onto the Memory Data Out Bus [MDO], and the operation code[opcode], which is in the first 4-bit of the MDO Bus, goes into the Instruction register as an input. Then the control unit reads the instruction register and determines what instructions needs to be executed. The control unit then generates the signal needed to implement that instruction to the functional units, which then outputs and executes the instruction.

The image below shows the input and output used to integrate the project on the DE10-Lite board.


Inputs – Start/Stop (KEY0), SysClock (KEY1)
HEX outputs – PC (HEX5), MAR (HEX4), MDOut (HEX3 & HEX2), MDIn (HEX1 & HEX0)
LED outputs – C0 (G0), C1 (G1), C2 (G2), C3 (G3), C4 (G4), C5 (G5), C7 (G6), C8 (G7), C9 (G8), C10 (G9), C11 (R6), C12 (R7), C13(R8), and C14 (R9). You may use the remaining LEDs for debugging purposes in any way you see fit.

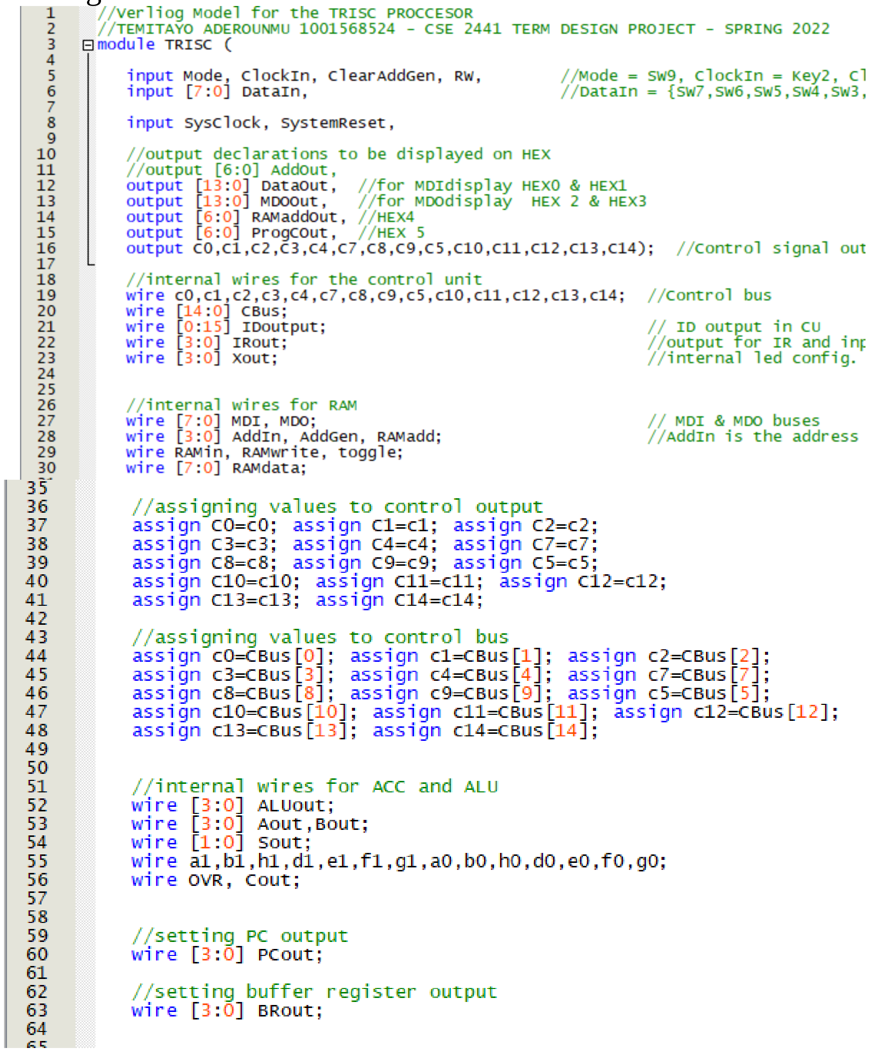
The table below shows the instruction set for the TRISC processor, and the highlighted instruction, shows the instruction that were actually executed for the project.



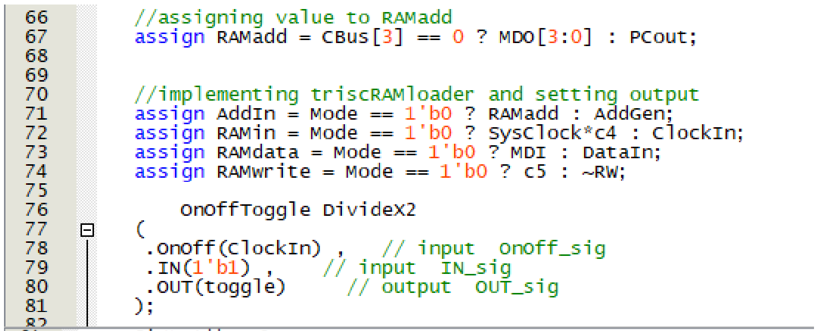
## TRISC Instruction Set

| Instruction | Function | Register Transfer | Op Code |
|---|---|---|---|
| LDA | Load ACC | ACC ← (MDR) | 0000 |
| STA | Store ACC | MDR ← (ACC) | 0001 |
| ADD | Add ACC | ACC ← (ACC) + (MDR) | 0010 |
| SUB | Subtract ACC | ACC ← (ACC) – (MDR) | 0011 |
| XOR | XOR ACC | ACC ← (ACC) ⊕ (MDR) | 0100 |
| INC | Increment ACC | ACC ← (ACC) + 1 | 0110 |
| CLR | Clear ACC | ACC ← 0 | 0111 |
| JMP | Jump | PC ← (MDR) | 1000 |
| JPZ | Jump if 0 | PC ← (MDR) if Z = 1 | 1100 |
| JPN | Jump if < 0 | PC ← (MDR) if N = 1 | 1001 |
| HLT | Halt | PC ← 0 | 1111 |

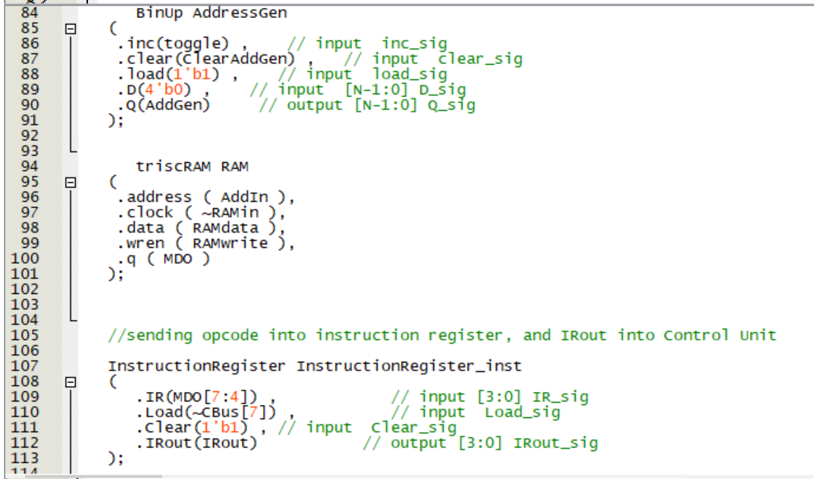
Verilog code:



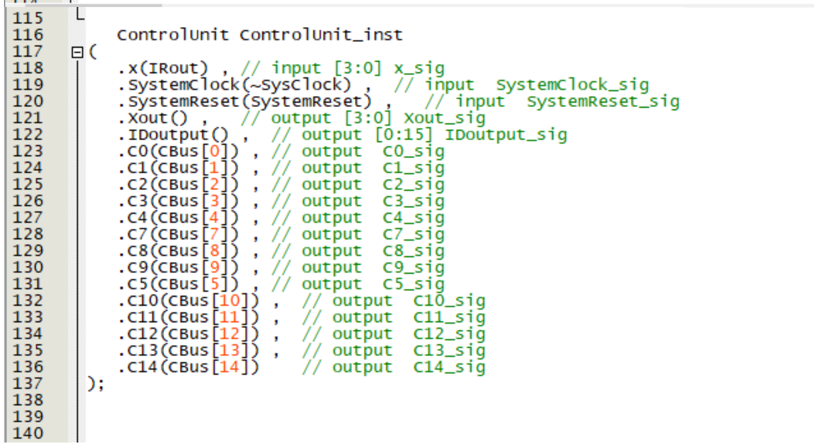
```
//Verliog Model for the TRISC PROCCESOR
//TEMITAYO ADEROUNMU 1001568524 - CSE 2441 TERM DESIGN PROJECT - SPRING 2022
module TRISC (

    input Mode, ClockIn, ClearAddGen, RW,        //Mode = SW9, ClockIn = Key2, Cl
    input [7:0] DataIn,                          //DataIn = {SW7,SW6,SW5,SW4,SW3,

    input SysClock, SystemReset,

    //output declarations to be displayed on HEX
    //output [6:0] AddOut,
    output [13:0] DataOut,  //for MDIdisplay HEX0 & HEX1
    output [13:0] MDOOut,   //for MDOdisplay  HEX 2 & HEX3
    output [6:0] RAMaddOut, //HEX4
    output [6:0] ProgCOut,  //HEX 5
    output C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14);  //Control signal out

    //internal wires for the control unit
    wire c0,c1,c2,c3,c4,c7,c8,c9,c5,c10,c11,c12,c13,c14;  //Control bus
    wire [14:0] CBus;
    wire [0:15] IDoutput;                                 // ID output in CU
    wire [3:0] IRout;                                     //output for IR and inp
    wire [3:0] Xout;                                      //internal led config.


    //internal wires for RAM
    wire [7:0] MDI, MDO;                                  // MDI & MDO buses
    wire [3:0] AddIn, AddGen, RAMadd;                     //AddIn is the address
    wire RAMin, RAMwrite, toggle;
    wire [7:0] RAMdata;

    //assigning values to control output
    assign C0=c0; assign C1=c1; assign C2=c2;
    assign C3=c3; assign C4=c4; assign C7=c7;
    assign C8=c8; assign C9=c9; assign C5=c5;
    assign C10=c10; assign C11=c11; assign C12=c12;
    assign C13=c13; assign C14=c14;

    //assigning values to control bus
    assign c0=CBus[0]; assign c1=CBus[1]; assign c2=CBus[2];
    assign c3=CBus[3]; assign c4=CBus[4]; assign c7=CBus[7];
    assign c8=CBus[8]; assign c9=CBus[9]; assign c5=CBus[5];
    assign c10=CBus[10]; assign c11=CBus[11]; assign c12=CBus[12];
    assign c13=CBus[13]; assign c14=CBus[14];


    //internal wires for ACC and ALU
    wire [3:0] ALUout;
    wire [3:0] Aout,Bout;
    wire [1:0] Sout;
    wire a1,b1,h1,d1,e1,f1,g1,a0,b0,h0,d0,e0,f0,g0;
    wire OVR, Cout;


    //setting PC output
    wire [3:0] PCout;

    //setting buffer register output
    wire [3:0] BRout;
```



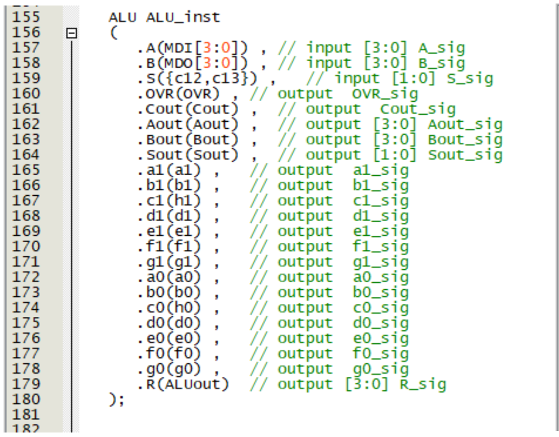
```
    //assigning value to RAMadd
    assign RAMadd = CBus[3] == 0 ? MDO[3:0] : PCout;


    //implementing triscRAMloader and setting output
    assign AddIn = Mode == 1'b0 ? RAMadd : AddGen;
    assign RAMin = Mode == 1'b0 ? SysClock*c4 : ClockIn;
    assign RAMdata = Mode == 1'b0 ? MDI : DataIn;
    assign RAMwrite = Mode == 1'b0 ? c5 : ~RW;

        OnOffToggle Dividex2
    (
     .onoff(ClockIn) ,    // input  onoff_sig
     .IN(1'b1) ,     // input  IN_sig
     .OUT(toggle)      // output  OUT_sig
    );
```



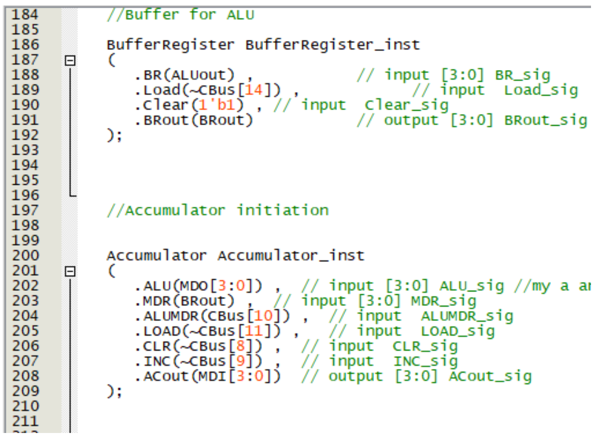
```
        BinUp AddressGen
    (
     .inc(toggle) ,     // input  inc_sig
     .clear(ClearAddGen) ,   // input  clear_sig
     .load(1'b1) ,     // input  load_sig
     .D(4'b0) ,      // input  [N-1:0] D_sig
     .Q(AddGen)      // output [N-1:0] Q_sig
    );


        triscRAM RAM
    (
     .address ( AddIn ),
     .clock ( ~RAMin ),
     .data ( RAMdata ),
     .wren ( RAMwrite ),
     .q ( MDO )
    );



    //sending opcode into instruction register, and IRout into Control Unit

    InstructionRegister InstructionRegister_inst
    (
       .IR(MDO[7:4]) ,             // input [3:0] IR_sig
       .Load(~CBus[7]) ,           // input  Load_sig
       .Clear(1'b1) , // input  Clear_sig
       .IRout(IRout)             // output [3:0] IRout_sig
    );
```



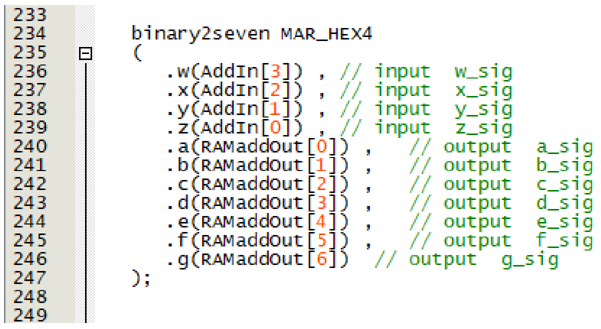
```

     ControlUnit ControlUnit_inst
    (
     .x(IRout) , // input [3:0] x_sig
     .SystemClock(~SysClock) ,  // input  SystemClock_sig
     .SystemReset(SystemReset) ,   // input  SystemReset_sig
     .Xout() ,  // output [3:0] Xout_sig
     .IDoutput() ,  // output [0:15] IDoutput_sig
     .C0(CBus[0]) , // output  C0_sig
     .C1(CBus[1]) , // output  C1_sig
     .C2(CBus[2]) , // output  C2_sig
     .C3(CBus[3]) , // output  C3_sig
     .C4(CBus[4]) , // output  C4_sig
     .C7(CBus[7]) , // output  C7_sig
     .C8(CBus[8]) , // output  C8_sig
     .C9(CBus[9]) , // output  C9_sig
     .C5(CBus[5]) , // output  C5_sig
     .C10(CBus[10]) ,  // output  C10_sig
     .C11(CBus[11]) ,  // output  C11_sig
     .C12(CBus[12]) ,  // output  C12_sig
     .C13(CBus[13]) ,  // output  C13_sig
     .C14(CBus[14])    // output  C14_sig
    );
```



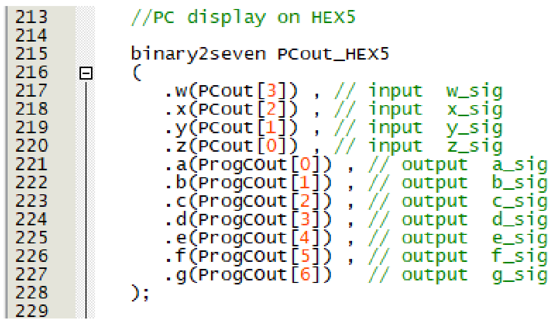
```
    ALU ALU_inst
    (
       .A(MDI[3:0]) , // input [3:0] A_sig
       .B(MDO[3:0]) , // input [3:0] B_sig
       .S({c12,c13}) ,   // input [1:0] S_sig
       .OVR(OVR) , // output  OVR_sig
       .Cout(Cout) ,  // output  Cout_sig
       .Aout(Aout) ,  // output [3:0] Aout_sig
       .Bout(Bout) ,  // output [3:0] Bout_sig
       .Sout(Sout) ,  // output [1:0] Sout_sig
       .a1(a1) ,   // output  a1_sig
       .b1(b1) ,   // output  b1_sig
       .c1(h1) ,   // output  c1_sig
       .d1(d1) ,   // output  d1_sig
       .e1(e1) ,   // output  e1_sig
       .f1(f1) ,   // output  f1_sig
       .g1(g1) ,   // output  g1_sig
       .a0(a0) ,   // output  a0_sig
       .b0(b0) ,   // output  b0_sig
       .c0(h0) ,   // output  c0_sig
       .d0(d0) ,   // output  d0_sig
       .e0(e0) ,   // output  e0_sig
       .f0(f0) ,   // output  f0_sig
       .g0(g0) ,   // output  g0_sig
       .R(ALUout)  // output [3:0] R_sig
    );
```



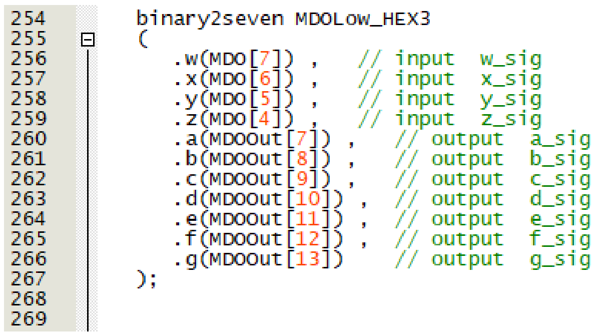
```
    //Buffer for ALU

    BufferRegister BufferRegister_inst
    (
       .BR(ALUout) ,          // input [3:0] BR_sig
       .Load(~CBus[14]) ,            // input  Load_sig
       .Clear(1'b1) , // input  Clear_sig
       .BRout(BRout)          // output [3:0] BRout_sig
    );




    //Accumulator initiation


    Accumulator Accumulator_inst
    (
       .ALU(MDO[3:0]) ,  // input [3:0] ALU_sig //my a ar
       .MDR(BRout) ,  // input [3:0] MDR_sig
       .ALUMDR(CBus[10]) ,  // input  ALUMDR_sig
       .LOAD(~CBus[11]) ,   // input  LOAD_sig
       .CLR(~CBus[8]) ,  // input  CLR_sig
       .INC(~CBus[9]) ,  // input  INC_sig
       .ACout(MDI[3:0])  // output [3:0] ACout_sig
    );
```



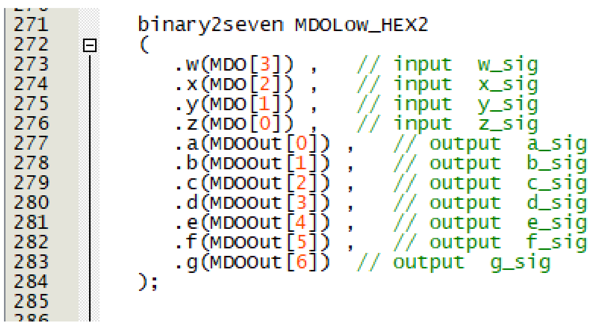
```
binary2seven MAR_HEX4
(
    .w(AddIn[3]) , // input  w_sig
    .x(AddIn[2]) , // input  x_sig
    .y(AddIn[1]) , // input  y_sig
    .z(AddIn[0]) , // input  z_sig
    .a(RAMaddOut[0]) ,   // output  a_sig
    .b(RAMaddOut[1]) ,   // output  b_sig
    .c(RAMaddOut[2]) ,   // output  c_sig
    .d(RAMaddOut[3]) ,   // output  d_sig
    .e(RAMaddOut[4]) ,   // output  e_sig
    .f(RAMaddOut[5]) ,   // output  f_sig
    .g(RAMaddOut[6])  // output  g_sig
);
```



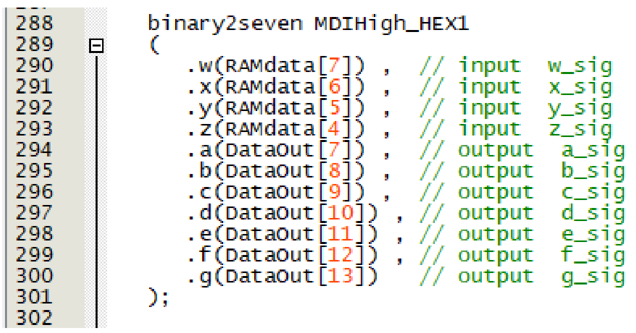
```
//PC display on HEX5

binary2seven PCout_HEX5
(
    .w(PCout[3]) , // input  w_sig
    .x(PCout[2]) , // input  x_sig
    .y(PCout[1]) , // input  y_sig
    .z(PCout[0]) , // input  z_sig
    .a(ProgCout[0]) , // output  a_sig
    .b(ProgCout[1]) , // output  b_sig
    .c(ProgCout[2]) , // output  c_sig
    .d(ProgCout[3]) , // output  d_sig
    .e(ProgCout[4]) , // output  e_sig
    .f(ProgCout[5]) , // output  f_sig
    .g(ProgCout[6])   // output  g_sig
);
```



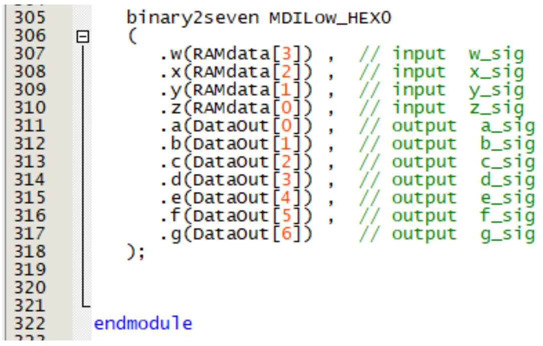
```
binary2seven MDOLow_HEX3
(
    .w(MDO[7]) ,   // input  w_sig
    .x(MDO[6]) ,   // input  x_sig
    .y(MDO[5]) ,   // input  y_sig
    .z(MDO[4]) ,   // input  z_sig
    .a(MDOout[7]) ,   // output  a_sig
    .b(MDOout[8]) ,   // output  b_sig
    .c(MDOout[9]) ,   // output  c_sig
    .d(MDOout[10]) ,  // output  d_sig
    .e(MDOout[11]) ,  // output  e_sig
    .f(MDOout[12]) ,  // output  f_sig
    .g(MDOout[13])    // output  g_sig
);
```



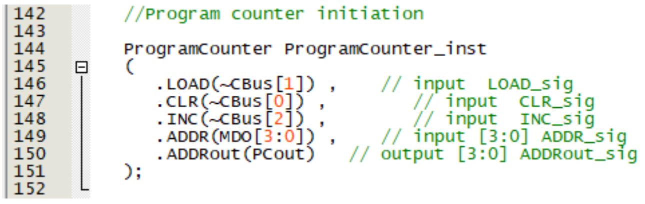
```
binary2seven MDOLow_HEX2
(
    .w(MDO[3]) ,   // input  w_sig
    .x(MDO[2]) ,   // input  x_sig
    .y(MDO[1]) ,   // input  y_sig
    .z(MDO[0]) ,   // input  z_sig
    .a(MDOout[0]) ,   // output  a_sig
    .b(MDOout[1]) ,   // output  b_sig
    .c(MDOout[2]) ,   // output  c_sig
    .d(MDOout[3]) ,   // output  d_sig
    .e(MDOout[4]) ,   // output  e_sig
    .f(MDOout[5]) ,   // output  f_sig
    .g(MDOout[6])  // output  g_sig
);
```



```
binary2seven MDIHigh_HEX1
(
    .w(RAMdata[7]) ,  // input  w_sig
    .x(RAMdata[6]) ,  // input  x_sig
    .y(RAMdata[5]) ,  // input  y_sig
    .z(RAMdata[4]) ,  // input  z_sig
    .a(DataOut[7]) ,  // output  a_sig
    .b(DataOut[8]) ,  // output  b_sig
    .c(DataOut[9]) ,  // output  c_sig
    .d(DataOut[10]) , // output  d_sig
    .e(DataOut[11]) , // output  e_sig
    .f(DataOut[12]) , // output  f_sig
    .g(DataOut[13])   // output  g_sig
);
```

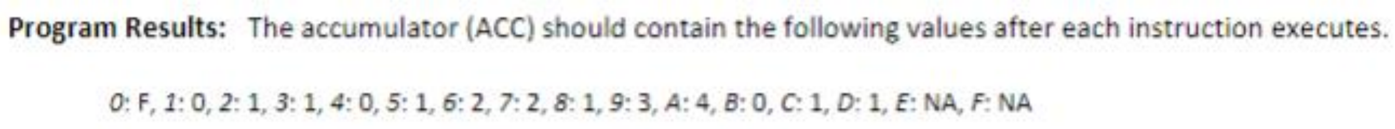


```
binary2seven MDILow_HEX0
(
    .w(RAMdata[3]) ,  // input  w_sig
    .x(RAMdata[2]) ,  // input  x_sig
    .y(RAMdata[1]) ,  // input  y_sig
    .z(RAMdata[0]) ,  // input  z_sig
    .a(DataOut[0]) ,  // output  a_sig
    .b(DataOut[1]) ,  // output  b_sig
    .c(DataOut[2]) ,  // output  c_sig
    .d(DataOut[3]) ,  // output  d_sig
    .e(DataOut[4]) ,  // output  e_sig
    .f(DataOut[5]) ,  // output  f_sig
    .g(DataOut[6])    // output  g_sig
);

endmodule
```



```
//Program counter initiation

ProgramCounter ProgramCounter_inst
(
    .LOAD(~CBus[1]) ,     // input  LOAD_sig
    .CLR(~CBus[0]) ,         // input  CLR_sig
    .INC(~CBus[2]) ,         // input  INC_sig
    .ADDR(MDO[3:0]) ,     // input [3:0] ADDR_sig
    .ADDRout(PCout)    // output [3:0] ADDRout_sig
);
```

Test Results:

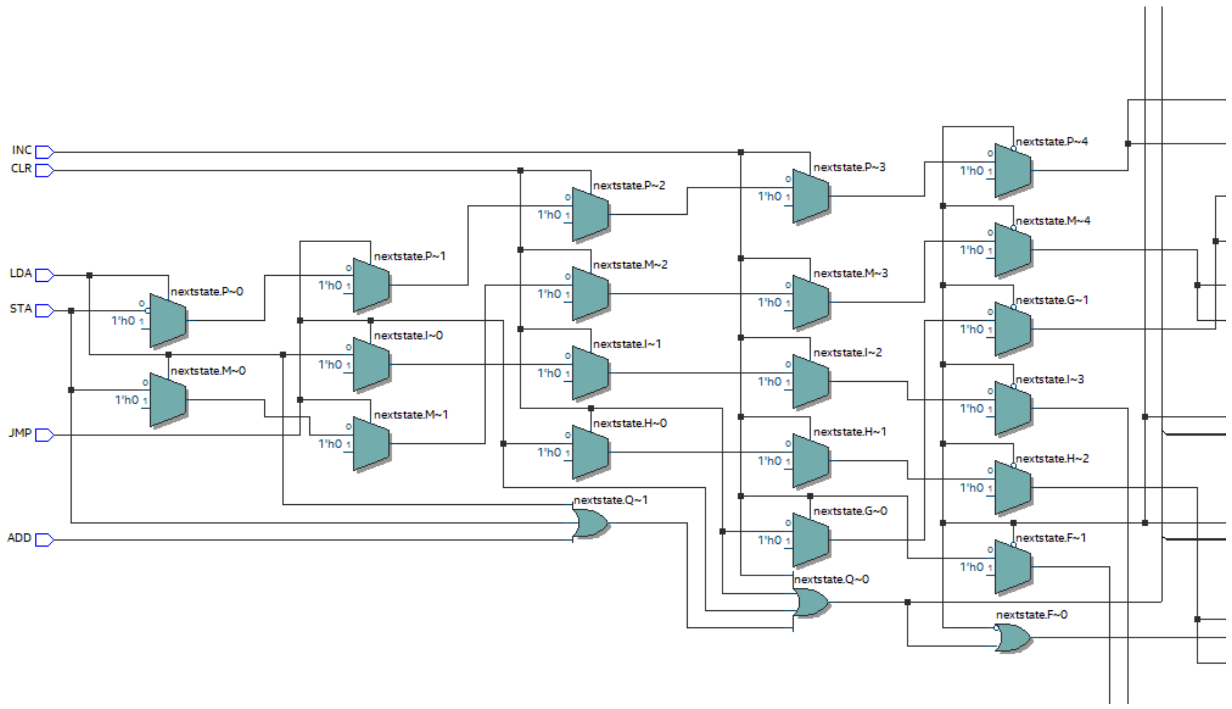


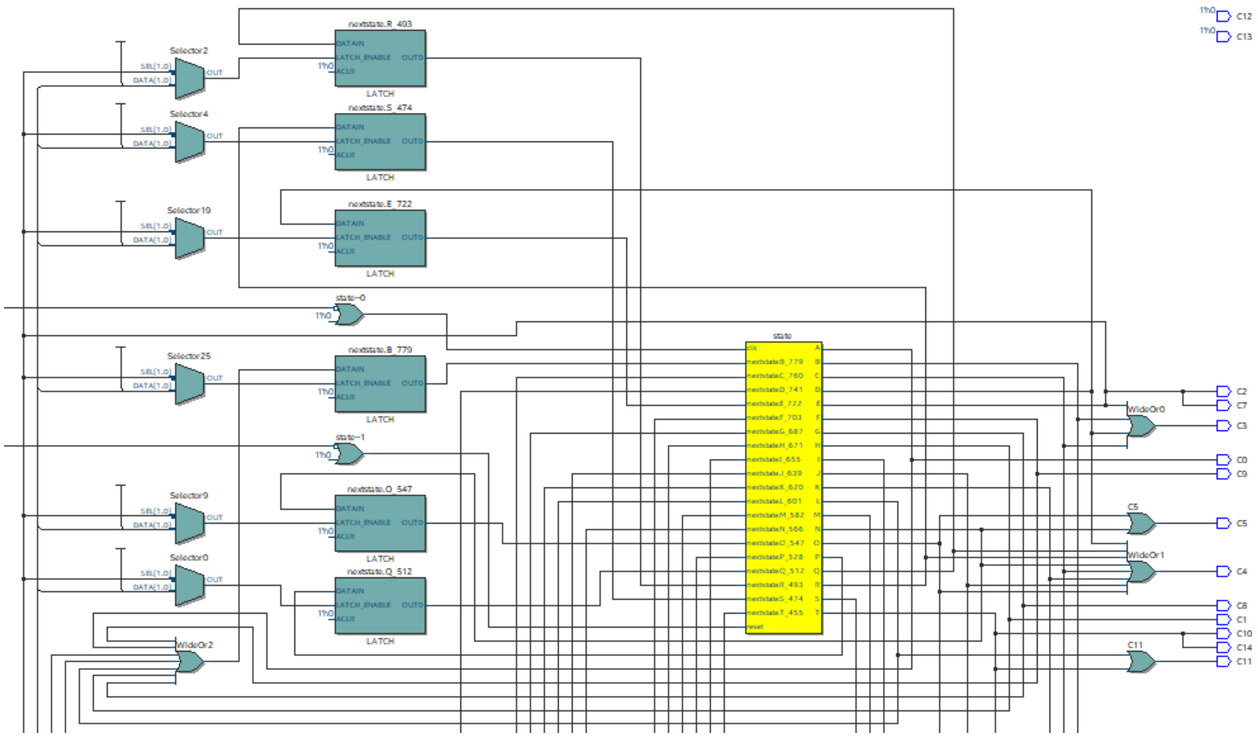
**Program Results:** The accumulator (ACC) should contain the following values after each instruction executes.

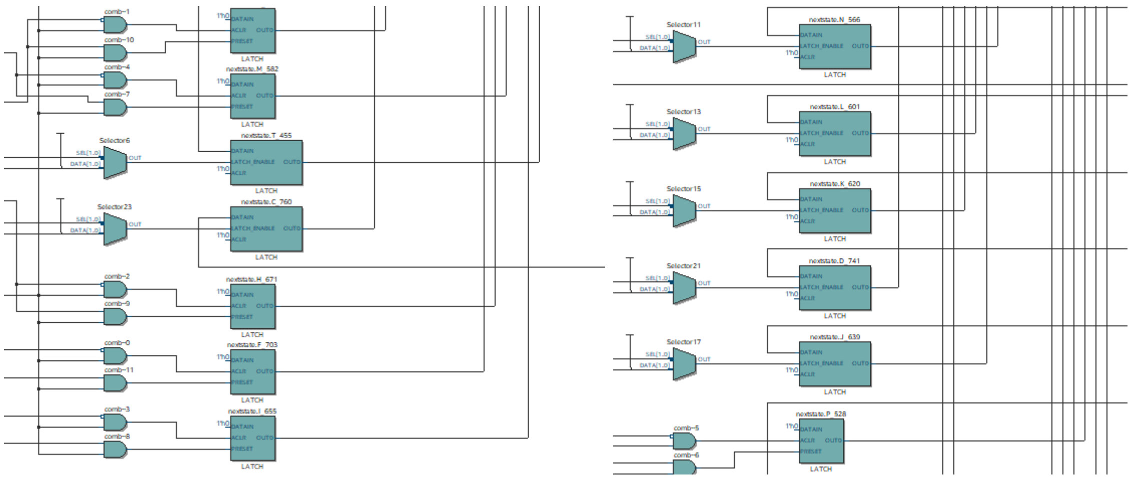
*0*: F, *1*: 0, *2*: 1, *3*: 1, *4*: 0, *5*: 1, *6*: 2, *7*: 2, *8*: 1, *9*: 3, *A*: 4, *B*: 0, *C*: 1, *D*: 1, *E*: NA, *F*: NA

## CONTROLLER DESIGN DETAILS:

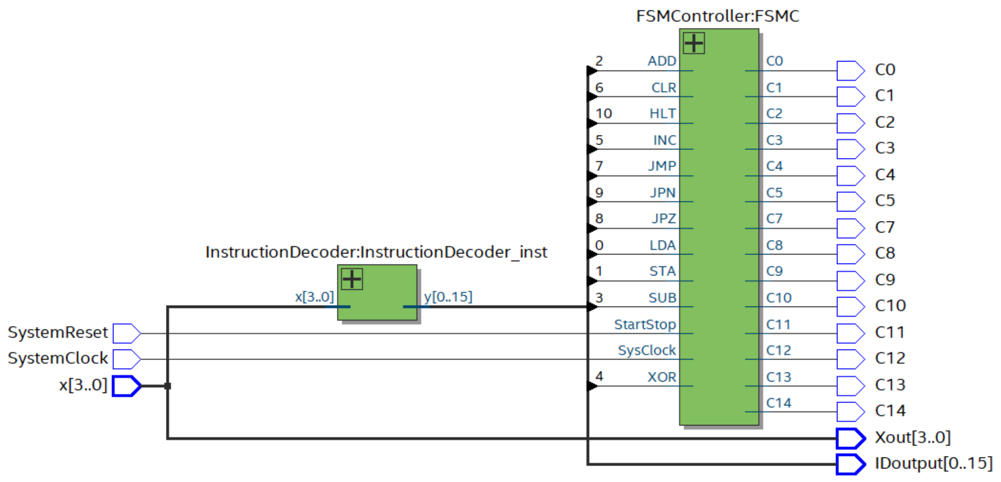
Functional description and diagrams showing input and output.



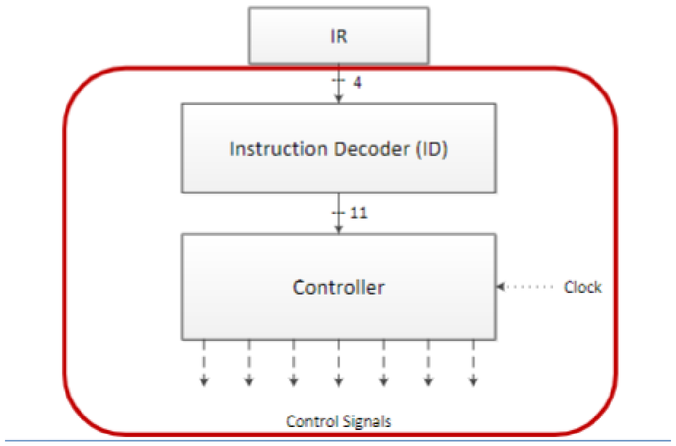




This is the RTL viewer of the controller

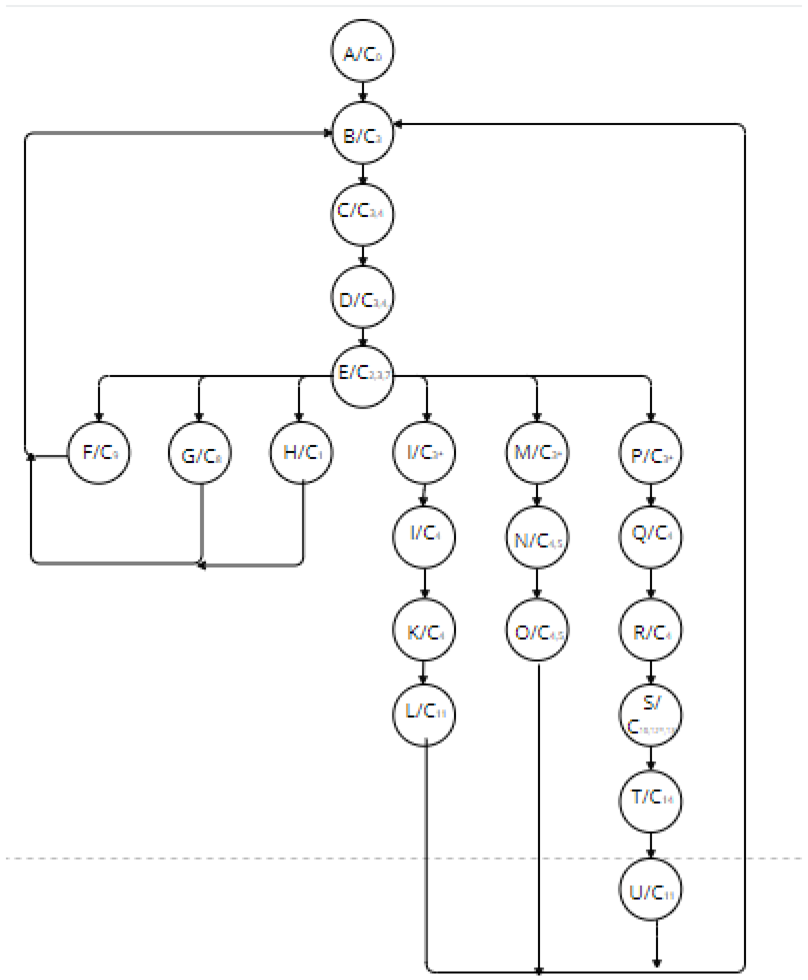


The RTL viewer of the control unit that implemented the controller and the Instruction decoder.



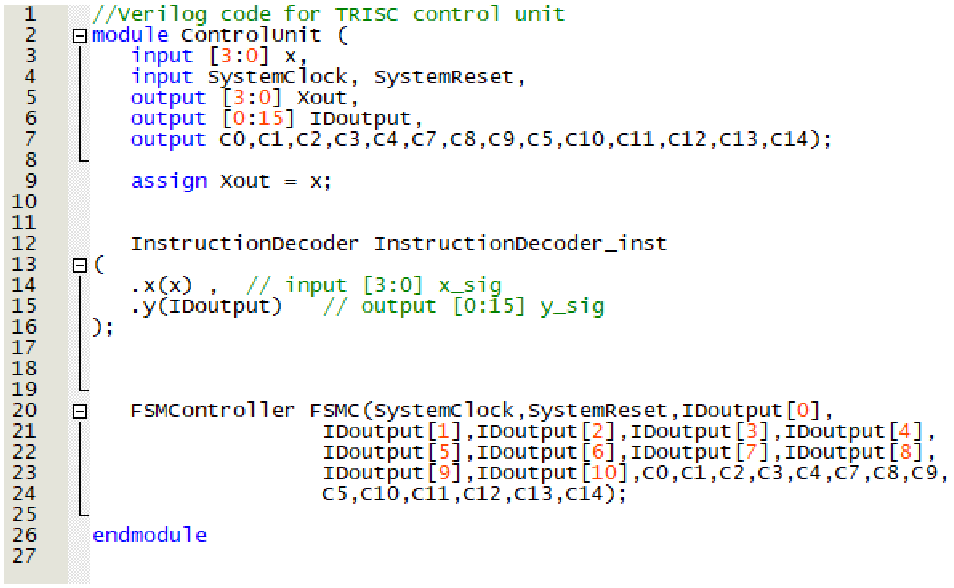
The control unit is an integration of the instruction decoder, and the controller. The input from the instruction register [IR] is sent into the instruction decode [ID], which converts the 4-bit integer into a 16-bit, which we will be needing 11 out of that. The controller then goes through a process, of first initializing, then fetching the instructions, and decoding it before sending out various output for the specific instruction to be executed.

State diagram:



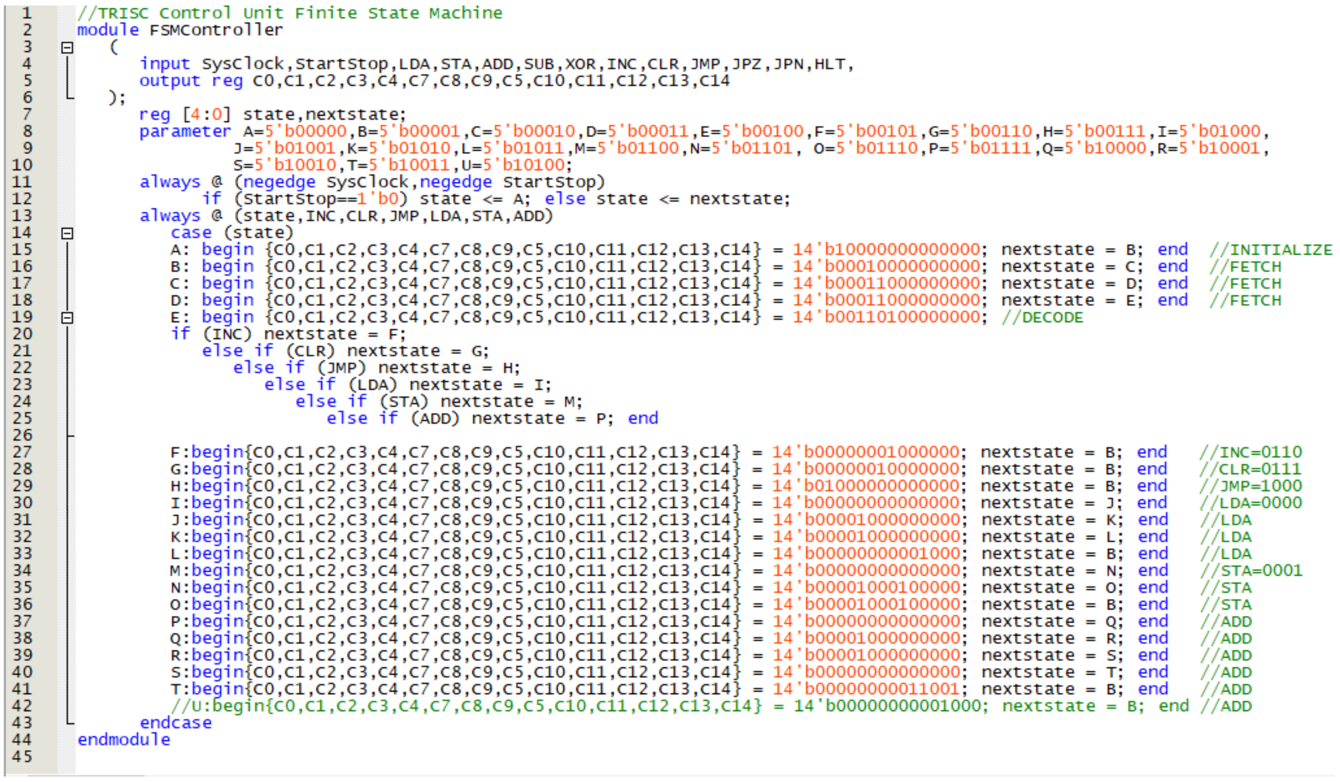
The A state is the initializing stage, the B state to the D state is Fetch stage, and the E state is the decoding state. Going from the left to the right, the instruction is increment [INC], clear [CLR], jump [JMP], load [LDA], store [STA], and ADD.

Verilog code:



```verilog
//Verilog code for TRISC control unit
module ControlUnit (
    input [3:0] x,
    input SystemClock, SystemReset,
    output [3:0] Xout,
    output [0:15] IDoutput,
    output C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14);

    assign Xout = x;


    InstructionDecoder InstructionDecoder_inst
(
    .x(x) ,  // input [3:0] x_sig
    .y(IDoutput)    // output [0:15] y_sig
);



    FSMController FSMC(SystemClock,SystemReset,IDoutput[0],
                       IDoutput[1],IDoutput[2],IDoutput[3],IDoutput[4],
                       IDoutput[5],IDoutput[6],IDoutput[7],IDoutput[8],
                       IDoutput[9],IDoutput[10],C0,C1,C2,C3,C4,C7,C8,C9,
                       C5,C10,C11,C12,C13,C14);

endmodule
```

The control unit Verilog code.



```verilog
//TRISC Control Unit Finite State Machine
module FSMController
   (
      input SysClock,StartStop,LDA,STA,ADD,SUB,XOR,INC,CLR,JMP,JPZ,JPN,HLT,
      output reg C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14
   );
      reg [4:0] state,nextstate;
      parameter A=5'b00000,B=5'b00001,C=5'b00010,D=5'b00011,E=5'b00100,F=5'b00101,G=5'b00110,H=5'b00111,I=5'b01000,
               J=5'b01001,K=5'b01010,L=5'b01011,M=5'b01100,N=5'b01101, O=5'b01110,P=5'b01111,Q=5'b10000,R=5'b10001,
               S=5'b10010,T=5'b10011,U=5'b10100;
      always @ (negedge SysClock,negedge StartStop)
          if (StartStop==1'b0) state <= A; else state <= nextstate;
      always @ (state,INC,CLR,JMP,LDA,STA,ADD)
         case (state)
         A: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b10000000000000; nextstate = B; end  //INITIALIZE
         B: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00010000000000; nextstate = C; end  //FETCH
         C: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00011000000000; nextstate = D; end  //FETCH
         D: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00011000000000; nextstate = E; end  //FETCH
         E: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00110100000000; //DECODE
         if (INC) nextstate = F;
            else if (CLR) nextstate = G;
               else if (JMP) nextstate = H;
                  else if (LDA) nextstate = I;
                     else if (STA) nextstate = M;
                        else if (ADD) nextstate = P; end

         F:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000001000000; nextstate = B; end   //INC=0110
         G:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000010000000; nextstate = B; end   //CLR=0111
         H:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b01000000000000; nextstate = B; end   //JMP=1000
         I:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000000000; nextstate = J; end   //LDA=0000
         J:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00001000000000; nextstate = K; end   //LDA
         K:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00001000000000; nextstate = L; end   //LDA
         L:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000001000; nextstate = B; end   //LDA
         M:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000000000; nextstate = N; end   //STA=0001
         N:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00001000100000; nextstate = O; end   //STA
         O:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00001000100000; nextstate = B; end   //STA
         P:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000000000; nextstate = Q; end   //ADD
         Q:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00001000000000; nextstate = R; end   //ADD
         R:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00001000000000; nextstate = S; end   //ADD
         S:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000000000; nextstate = T; end   //ADD
         T:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000011001; nextstate = B; end   //ADD
         //U:begin{C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000001000; nextstate = B; end //ADD
      endcase
endmodule
```

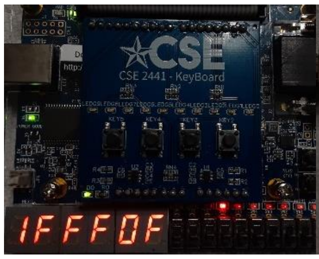
Controller Verilog Code

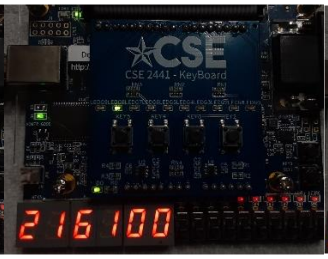
## TEST RESULTS

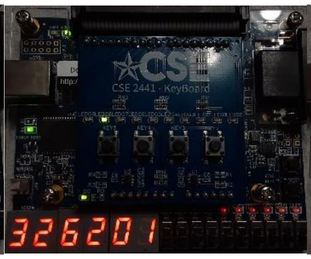
Summary:

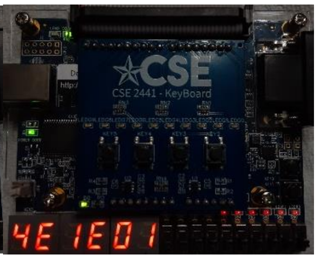
With the program loaded into the TRISC RAM loader, the various instructions were executed, and different output were shown on the DE10 lite board. The first value outputted was F, and the corresponding program loaded was 0F, this shows that the opcode for this was 0000, and the instruction for that would be load, which means that F is being loaded. The next loaded program was 61, this shows that the opcode for this was 0110, and the instruction for that would be increment, which means 0 would be outputted. And for the next program, which is 62, it also implements the increment instruction and 1 would now be outputted. This occurs for the rest of the program loaded into the RAM at the beginning of the testing process. All of the output I obtained while testing matched the expected results for each instruction.

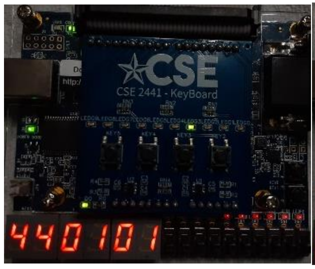
Photos and videos of execution:

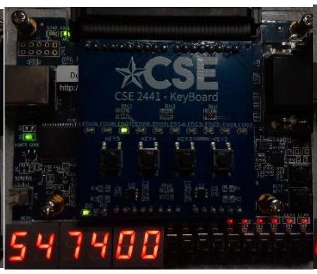


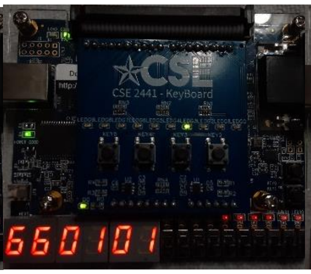


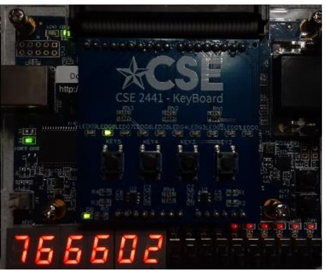


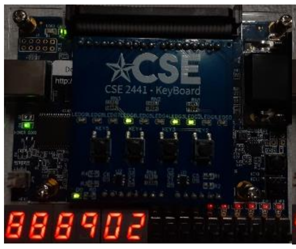


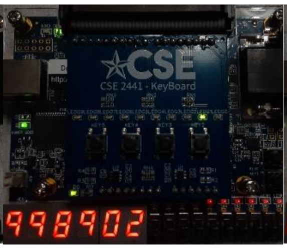


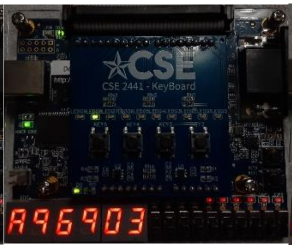


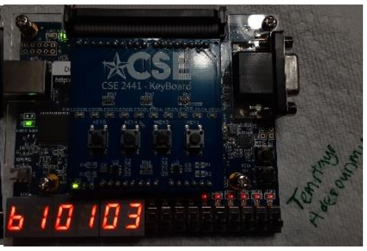


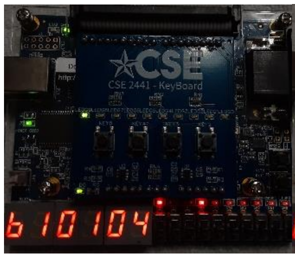


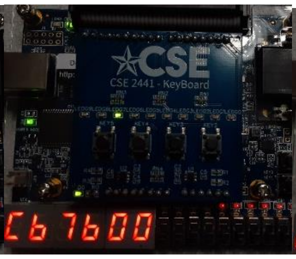


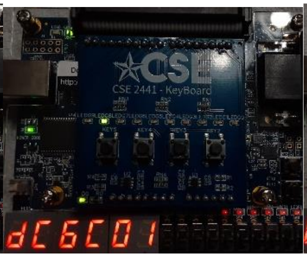


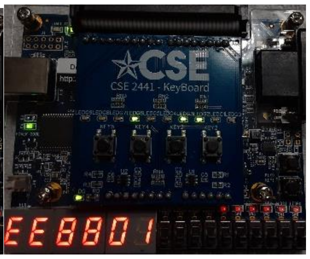


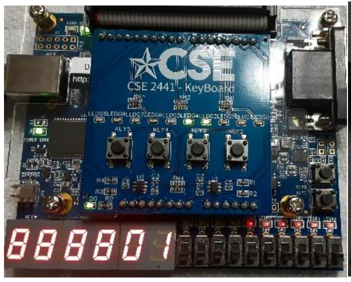


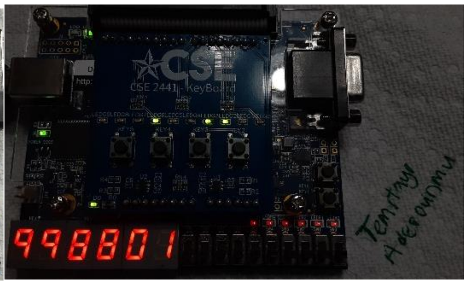












## CONCLUSION

Resolution of Design:

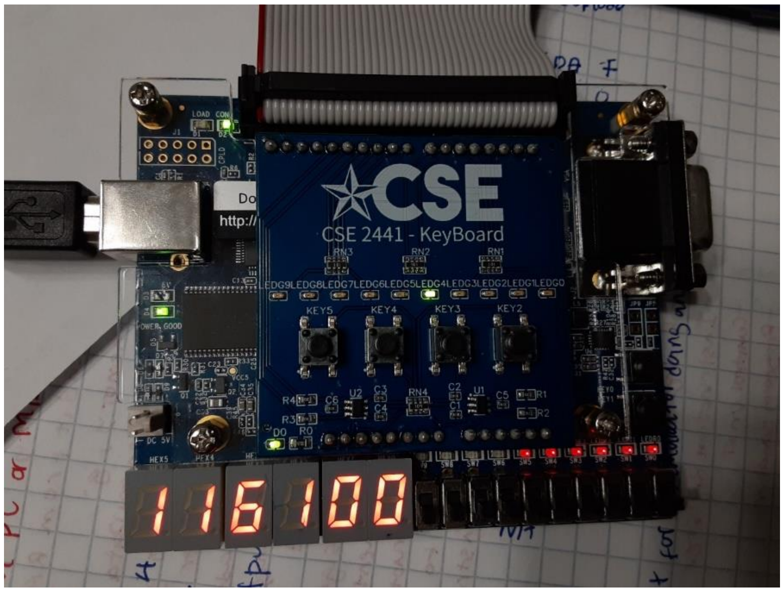
For this project, after working on it for about a week and a half, I was able to resolve all the initial issues I had by the 29th of April. After many deliberations, it was discovered that the main problem that prevented my code from executing was an error found in my finite state controller code, which was fixed thereafter, and everything started working perfectly.

Implementation Difficulties:

April 28th, 2022, testing difficulties documentation:

I am still running into some issues. I am not sure where the problem is, but I do not really understand how to store the output of my accumulator. Using the buffer register, I sent in the ALU program output as input for the buffer register, and the output that I get from the buffer register is an input for the accumulator. And looking at the TRISC diagram, it illustrates that the accumulator outputs to the MDI bus, and also to the ALU, I am not sure if I correctly set the condition, for where the accumulator output should be.

While testing the processor on the DE 10 lite board, though the program was loaded correctly, as I read through it by continuously clicking on key 2, to iterate through the memory, and made sure it was loaded correctly. Then I turned off the mode [SW 9], and test if the processor works correctly, however only some of it were correct. And after several rounds of testing, I noticed that after the program fetched it first increment instruction, it stayed at that instruction (the MDO display on the board did not change) and kept incrementing the value shown on the MDI in on HEX 0 and on HEX 1. The picture below shows where it worked Before it started the incrementing instruction continuously.



On the twenty ninth of April, after many debugging, I noticed that when assigning signals that was from the control unit, I forgot to assign C3, which is a signal being sent into the TRISC RAM to determine whether to select the MDO or the program counter output as an input for the TRISC RAM. The worries I had before about whether I was using the buffer register correctly was needless, as it was implemented correctly from the start. Another mistake I noticed was that the values of the controls being sent was offset by one, so I went back to edit the FSM controller Verilog code, which solved the problem I was experiencing.

Lesson:

With this project, we have gained a lot of knowledge about instantiating different components into a main Verilog HDL file. In this project, we integrated the TRISC components on the DE10-Lite board, that have been previously completed in various exercises and homework to implement a function processor that can execute programs consisting of INC, CLR, JMP, LDA, STA, and ADD instructions. Integrating all the different components that were already implemented, allowed us to get the full experience of designing the processor.

References:

Nelson, Victor P., et al *Digital Logic Circuit and Design.* Pearson Education, Inc., 2019.