# TERM DESIGN PROJECT

Prepared For: Dr. Bill Carroll
Prepared By: Nathan Fusselman
Date: Tuesday, April 30, 2019

# Table of Contents

# Table of Figures

# 1 Introduction

## 1.1 Project Overview

For this Term Project I was assigned to design and implement a TRISC processor which is able to perform basic operation such as Load, Store, Add, Increment, Clear, and Jump. The design must be made in Quartus II and must be tested on the DE1 development board. The design will consist of a provided TRISC RAM module, but all other parts must be constructed by myself.

## 1.2 Project Status

I have completed all required sections of the project and my processor design is capable of running all of the required operations. I plan to add some addition operations such as Subtract, XOR, Jump if 0, Jump > 0, and Halt. These additions would complete the design and would only require a little bit of changes to the design to complete.

## 1.3 Report Overview

This report will consist of six primary section witch can be found in the table of contents. The report will include a full System design, Controller design details, Alternative design considerations, and Integration and Test Plan section among the Introduction and conclusion.

# 2 System Design

## 2.1 System-level description and diagrams

To simplify my design, I have utilized five bus lines to wrap around the entire processor to better assist the design, Figure 1. The bus lines consist of the Control bus (C[16.]) illustrated in Black. This bus contains all of the control signals including the Clock and the System Start signal. The next bus is the Address bus (A[3.]) illustrated in Blue. This bus contains the primary working address for the RAM module. The MDI (Memory data In) bus (MDI[7.]) is used to load information into the provided RAM module. The MDO (Memory data Out) but (MDO[7.]) is the output of the RAM module. I added one other bus called the ALU bus (ALU[3.]) to simplify the design and also provide a good path between the ALU out and the ACC input.

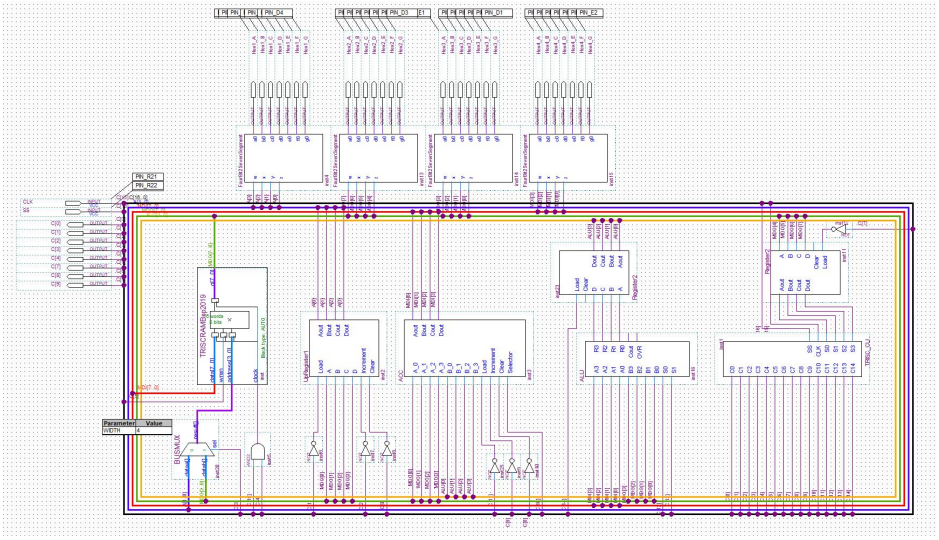

Figure 1 - General Design Diagram

This following section will consist of one subsection for each part of the system design. Please refer to Figure 1 as a visual guide for these parts. (Left to Right)

## 2.2 System-level description and diagrams

### 2.2.1 TRISC-RAM Module

This ram module was provided to hold the program that will be performed and holds all values that are being accessed in memory. Data is stored into a 8 bit section where I later split this up into two 4 bit sections. Figure 2.

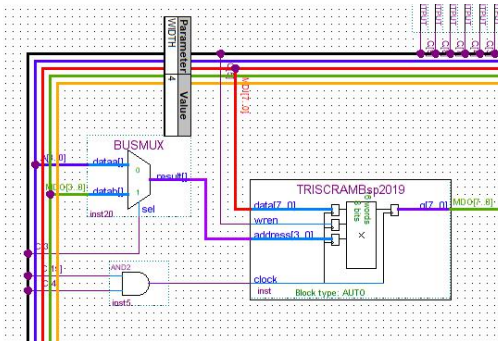


Figure 2 - RAM Module Implementation

The TRISC RAM module needs to have a 2x4 MUX added to the data in lines to allow for the device to be able to change what location of the processor it gets its address from. This could be either the past addressed data out or could be the PC. Figure 2.

## 2.2.2 PC (Program Counter)

The PC performs a very simple task of being an Up-Counter this is needed to keep track of what word in the RAM module should be accessed. The PC can jump to a new address and continue counting from there. Figure 3.

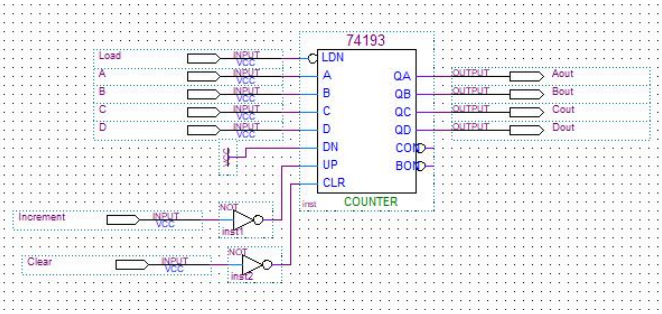


Figure 3 - PC Design

The PC Counter consists of a 74193 IC (4-bit Up Down Counter) where I have connected the UP line to a Not gate to turn into and active low signal. Figure 3.

## 2.2.3 ACC (Accumulator)

The ACC allows for us to select what kind of data is being accessed at that time. It is Up-Counter with an added MUX (Multiplexer) to allow for the selection and incrementing of two Inputs going to one output. Figure 4.

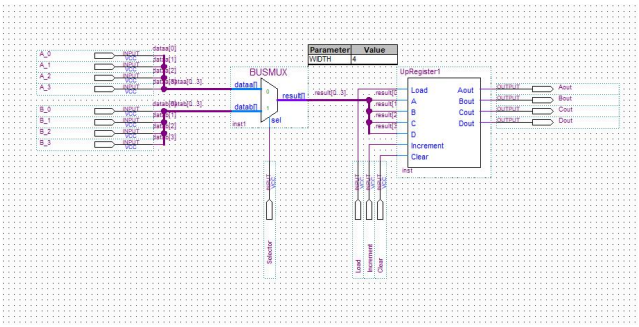


Figure 4 - ACC Design

The ACC consists of the same PC counter circuit with the addition of MUX (Multiplexer) so that you can switch the input of the counter between the BUS lines and the ALU. Figure 4.

## 2.2.4 ALU (Arithmetic Logic Unit)

The ALU is used to perform all math operations such as add, subtract, and XOR. This unit takes in two 4 bit inputs to be used as the operands, and takes in a 2 bit input that is used to determine what operation needs to take place. To stop the output from jumping around when the inputs are still being inputed I have added a buffer to the output of the ALU to only perform the operation when it is requested.

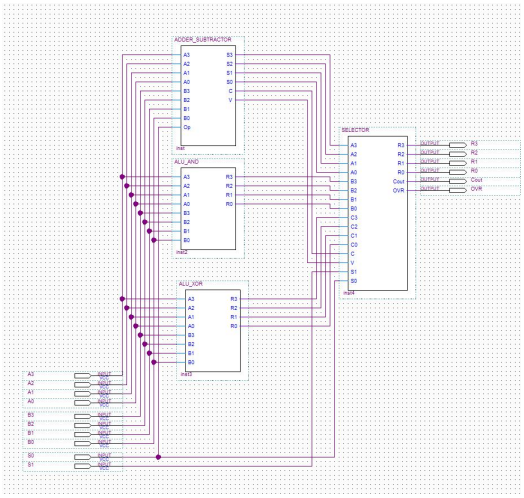


Figure 5 - ALU Top Level Design

The ALU Consists of 4 main subsections. The Adder-Subtractor, AND, XOR, and selector. The Adder-Subtractor, AND , and XOR work in parallel. All possible answers are calculated simultaneously were the selector then takes the proper answer given the request and outputs that. Figure 5.

```
module SELECTOR (A3,A2,A1,A0,B3,B2,B1,B0,C3,C2,C1,C0,R3,R2,R1,R0,C,V,S1,S0,Cout,OVR);
   input A3,A2,A1,A0,B3,B2,B1,B0,C3,C2,C1,C0,S1,S0,C,V;
   output reg R3,R2,R1,R0,Cout,OVR;
      always @ (S1,S0)
         case ({S1,S0})
            2'b10:begin
            R0 <= B0;
            R1 <= B1;
            R2 <= B2;
            R3 <= B3;
            Cout = 0;
            OVR = 0;
            end
            2'b11:begin
            R0 <= C0;
            R1 <= C1;
            R2 <= C2;
            R3 <= C3;
            Cout = 0;
            OVR = 0;
            end
            default:begin
            R0 <= A0;
            R1 <= A1;
            R2 <= A2;
            R3 <= A3;
            Cout <= C;
            OVR <= V;
            end
         endcase
endmodule
```

Figure 6 (Appendix 1) - ALU Selector Code

The Selector is essentially a 3x4 MUX for the 3 subsection outputs, this takes in the 2 operation bits and determines what section should be outputted. It also will output an OVR (Overflow) and a COUT (Carry out) for the Adder-Subtractor if selected. Figure 6.

The ALU Adder-Subtractor is fully written in two verilog modules, It simulates multiple Full-Adders in parallel to allow for a simple design that is compact and simple. The system is able to subtract by using the complement of the number and adding that together. Figure 7.

```
module ADDER_SUBTRACTOR(S3, S2, S1, S0, C, V, A3, A2, A1, A0, B3, B2, B1, B0, Op);
   output S3,S2,S1,S0;   // The 4-bit sum/difference..
   wire Cout;
   output C;
   output   V;   // The 1-bit overflow status.
   input A3,A2,A1,A0;   // The 4-bit augend/minuend.
   input B3,B2,B1,B0;   // The 4-bit addend/subtrahend.
   input    Op;  // The operation: 0 => Add, 1=>Subtract.

   wire  C0; // The carry out bit of fa0, the carry in bit of fa1.
   wire  C1; // The carry out bit of fa1, the carry in bit of fa2.
   wire  C2; // The carry out bit of fa2, the carry in bit of fa3.
   wire  C3; // The carry out bit of fa2, used to generate final carry/borrrow.

   wire  R0; // The xor'd result of B[0] and Op
   wire  R1; // The xor'd result of B[1] and Op
   wire  R2; // The xor'd result of B[2] and Op
   wire  R3; // The xor'd result of B[3] and Op


   xor(R0, B0, Op);
   xor(R1, B1, Op);
   xor(R2, B2, Op);
   xor(R3, B3, Op);
   xor(Cout, C3, Op);     // Carry = C3 for addition, Carry = not(C3) for subtraction.
   xor(V, C3, C2);     // If the two most significant carry output bits differ, then we have an overflow.

   xor(C, Op, Cout);


   full_adder fa0(S0, C0, A0, R0, Op);    // Least significant bit.
   full_adder fa1(S1, C1, A1, R1, C0);
   full_adder fa2(S2, C2, A2, R2, C1);
   full_adder fa3(S3, C3, A3, R3, C2);    // Most significant bit.
endmodule // ripple_carry_adder_subtractor


module full_adder(S, Cout, A, B, Cin);
   output S;
   output Cout;
   input  A;
   input  B;
   input  Cin;

   wire   w1;
   wire   w2;
   wire   w3;
   wire   w4;

   xor(w1, A, B);
   xor(S, Cin, w1);
   and(w2, A, B);
   and(w3, A, Cin);
   and(w4, B, Cin);
   or(Cout, w2, w3, w4);
endmodule
```

Figure 7 (Appendix 2) - ALU Adder-Subtractor Code

The ALU AND section simply runs the AND operand on each input but giving the output to the selector. Figure 8.

```
module ALU_AND(A3,A2,A1,A0,B3,B2,B1,B0,R3,R2,R1,R0);
   input A3,A2,A1,A0,B3,B2,B1,B0;
   output R3,R2,R1,R0;
   and(R0,A0,B0);
   and(R1,A1,B1);
   and(R2,A2,B2);
   and(R3,A3,B3);
endmodule
```

Figure 8 - ALU AND Code

The ALU XOR section is exactly like the AND section with the only difference being the operation it performs. Figure 9.

```
module ALU_XOR (A3,A2,A1,A0,B3,B2,B1,B0,R3,R2,R1,R0);
   input A3,A2,A1,A0,B3,B2,B1,B0;
   output R3,R2,R1,R0;
   xor(R0,A0,B0);
   xor(R1,A1,B1);
   xor(R2,A2,B2);
   xor(R3,A3,B3);
endmodule
```

Figure 9 - ALU XOR Code

### 2.2.5 Controller

The Controller translates the OP Code that is stored in the TRISC RAM module as a single binary input that feeds the main control module. In here, it uses a finite state machine to translate that command into a set of instructions that it will send to each of the component over the Control Bus. As the process for designing this section of the Processor is quite complicated, I have dedicated a full section to this part.

## 2.3 Hierarchical Design Structure

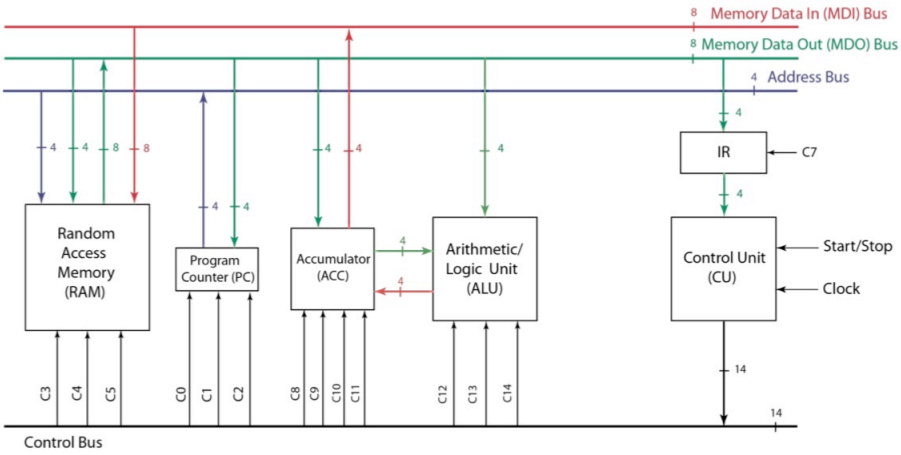


Figure 10 - Hierarchical Design Structure

The overall design of the TRISC Processor is as seen in Figure 10. This allows for a better image of what connects to what with the large number of busses and small NOT gates.

## 2.4 Operating Procedures.

Once you have input a RAM module with the program you would like to execute, you can control the system on the DE1 with KEY0 (System Start), and KEY1 (Clock). To increment the clock and perform another task in the Controller, you need to only press the Clock button. If you would like to restart the program you can press the System Start button and it will Start from the beginning once again. The TRISC Processor outputs its data to the HEX outputs. HEX 0 is the MDI line. HEX 1 is the MDO bus with the address section of it. HEX 2 is the MDO bus with the OPCode section of it. HEX 3 is the PC Counter Address. The LEDs have been configured as debug lights to signal what control signals are being sent to the subsections.

# 3 Controller Design Details

## 3.1 Functional description and diagram showing I/O

The TRISC Controller is like the instruction manual. This unit allows for it to decode the 4-bit OP Codes that are stored in memory. It then is able to trigger controls signals in the correct order based on what OP Code is input so that the device can perform the correct action.

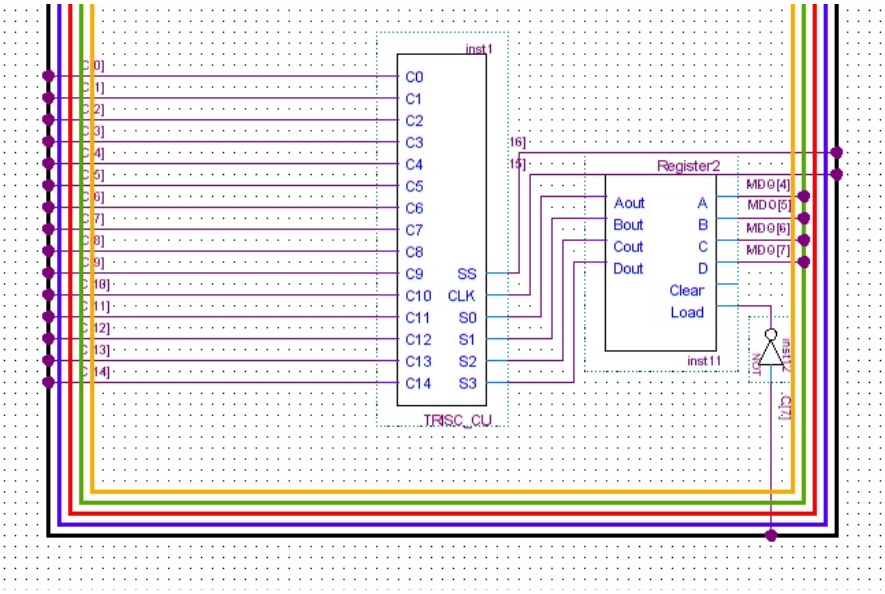


Figure 11 - Controller Integration

The Controller brings in its information from the IR (Instruction Register) where it is stored until the next code is needed. The Controller then splits the 4-bit signal into a single on or off bit in the controller. This allows for the controls to easily determine what Operation is needed. The Controls then outputs the correct signals to the Control Bus. Figure 11.

## 3.2 State Diagrams

An easy way to visualize what the Controller is performing is with a State diagram. The State diagram shows you the path that the commands follow. Each movement is triggered by a single clock pulse. Figure 12. State A Is used only to clear the Program Counter as that restarts the Program back at the beginning. States B through E are used to grab the next OP Code so that the device can read it and perform the Code specific tasks. All of the ends states go back to state B where it restart the OP Code Fetch.

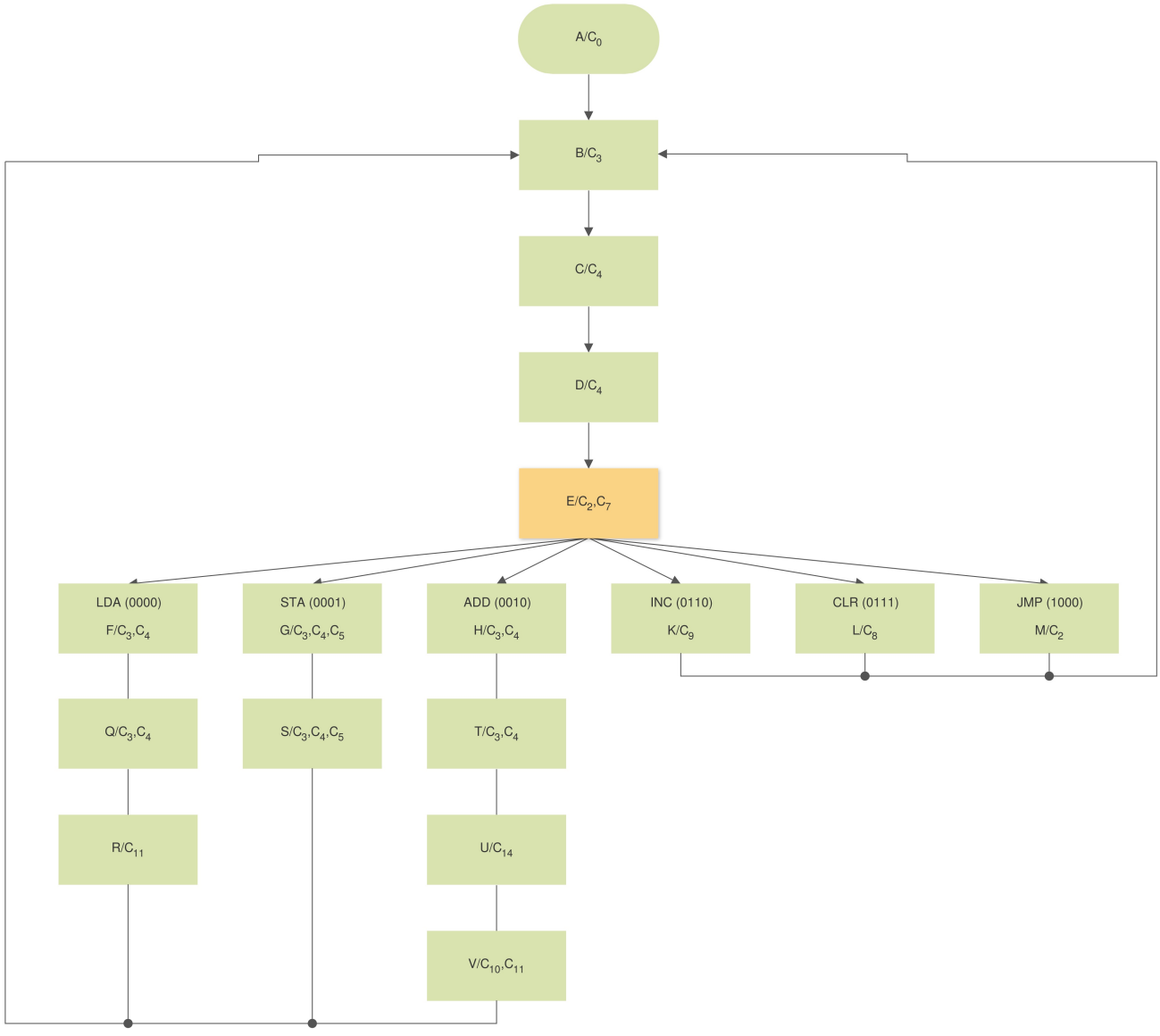


Figure 12 - State Diagram

## 3.3 Schematic Diagrams and Verilog Code

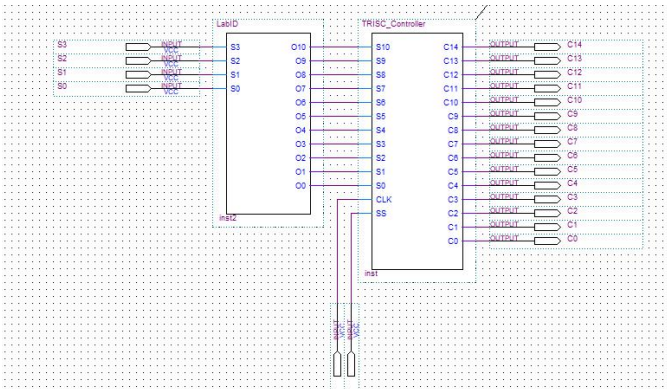


Figure 13 - Controller Top Level Design

There are 3 main sections of the controller. The first is the IR (Instruction Register) Figure 11. This holds the current OP Code that is being worked on. This is just a simple 4-bit register that is able to be triggered by the Control Bus ($C_7$). The Register is built on a 74175 IC that is able to store a 4-Bit value. Figure 14.

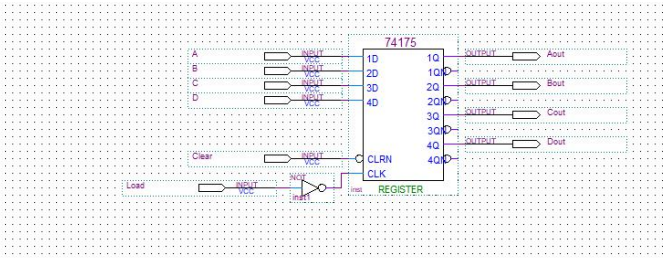


Figure 14 - IR Design

The second subsystems called the Controller ID where the system Decodes the OP Code into a single Bit output into the true Controller. Figure 15.

```
module LabID(S3,S2,S1,S0,O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0);
   input S3,S2,S1,S0;
   output reg O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0;
   always @ (S3,S2,S1,S0) begin
      case ({S3,S2,S1,S0})
         4'b0000: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000000001; //LDA
         4'b0001: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000000010; //STA
         4'b0010: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000000100; //ADD
         4'b0011: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000001000; //SUB
         4'b0100: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000010000; //XOR
         4'b0110: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000100000; //INC
         4'b0111: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00001000000; //CLR
         4'b1000: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00010000000; //JMP
         4'b1100: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00100000000; //JPZ
         4'b1001: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b01000000000; //JPN
         4'b1111: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b10000000000; //HLT
         default: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000000000; //ERR
      endcase
   end
endmodule
```

Figure 15 (Appendix 3) - Controller ID Code

The actual controls was fully made using HDL (Hardware Description Language), Specifically Verilog. (The state names in the Verilog code are the same as the state diagram.) Figure 16.

```
module TRISC_Controller(S10,S9,S8,S7,S6,S5,S4,S3,S2,S1,S0,CLK,SS,C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0);
   input S10,S9,S8,S7,S6,S5,S4,S3,S2,S1,S0,CLK,SS;
   output reg C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0;
   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, V=5'b10101, W=5'b10110, X=5'b10111, Y=5'b11000, Z=5'b11001, AA=5'b11010, AB=5'b11011, AC=5'b11100,
   AD=5'b11101, AE=5'b11110, AF=5'b11111;
   always @ (negedge CLK, negedge SS) begin
      if (SS==0)state<=A;
         else state <= nextstate;
   end
   always @ (state) begin
      case (state)
         A:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000001; nextstate <= B;end
         B:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000000; nextstate <= C;end
         C:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000010000; nextstate <= D;end
         D:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000010000; nextstate <= E;end

         E:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000010000100; if(S0)nextstate<=F;
         else if(S1)nextstate<=G; else if(S2)nextstate<=H; else if(S3)nextstate<=I; else if(S4)nextstate<=J;
         else if(S5)nextstate<=K;else if(S6)nextstate<=L;else if(S7)nextstate<=M;else if(S8)nextstate<=N; else if(S9)nextstate<=O;
         else if(S10)nextstate<=P; else nextstate<=A;end

         F:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000011000; nextstate <= Q;end //LDA:0
         G:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000111000; nextstate <= S;end //STA:0
         H:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000011000; nextstate <= T;end //ADD:0
         I:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000000; nextstate <= B;end //SUB:0
         J:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000000; nextstate <= B;end //XOR:0
         K:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000001000000000; nextstate <= B;end //INC:0
         L:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000100000000; nextstate <= B;end //CLR:0
         M:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000010; nextstate <= B;end //JMP:0
         N:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000000; nextstate <= B;end //JPZ:0
         O:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000000; nextstate <= B;end //JPN:0
         P:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000000; nextstate <= B;end //HLT:0
         Q:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000011000; nextstate <= R;end //LDA:1
         R:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000100000001000; nextstate <= B;end //LDA:2
         S:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000111000; nextstate <= B;end //STA:1
         T:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000011000; nextstate <= U;end //ADD:1
         U:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b010000000000000; nextstate <= V;end //ADD:2
         V:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000110000000000; nextstate <= B;end //ADD:3

         default:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000001; nextstate <= B;end
      endcase
   end
endmodule
```

Figure 16 (Appendix 4) - Controller Code

## 3.4 DE1 Pin Assignments

The pins that I have assigned are simply enough to see the control signals that are being sent, as well as to allow for the KEY inputs and the HEX outputs. Figure 17.

| | | | | |
|---|---|---|---|---|
| Input | CLK | Location | PIN_R21 | Yes |
| Input | SS | Location | PIN_R22 | Yes |
| Output | C2 | Location | PIN_Y18 | Yes |
| Output | C3 | Location | PIN_R18 | Yes |
| Output | C4 | Location | PIN_U18 | Yes |
| Output | C0 | Location | PIN_R17 | Yes |
| Output | C7 | Location | PIN_V19 | Yes |
| Output | C1 | Location | PIN_U19 | Yes |
| Output | C8 | Location | PIN_T18 | Yes |
| Output | C9 | Location | PIN_Y19 | Yes |
| Output | Hex1_B | Location | PIN_D5 | Yes |
| Output | Hex1_C | Location | PIN_D6 | Yes |
| Output | Hex1_D | Location | PIN_J4 | Yes |
| Output | Hex1_E | Location | PIN_L8 | Yes |
| Output | Hex1_F | Location | PIN_F3 | Yes |
| Output | Hex1_G | Location | PIN_D4 | Yes |
| Output | Hex2_A | Location | PIN_G5 | Yes |
| Output | Hex2_B | Location | PIN_G6 | Yes |
| Output | Hex2_C | Location | PIN_C2 | Yes |
| Output | Hex2_D | Location | PIN_C1 | Yes |
| Output | Hex2_E | Location | PIN_E3 | Yes |
| Output | Hex2_F | Location | PIN_E4 | Yes |
| Output | Hex2_G | Location | PIN_D3 | Yes |
| Output | Hex3_A | Location | PIN_E1 | Yes |
| Output | Hex3_B | Location | PIN_H6 | Yes |
| Output | Hex3_C | Location | PIN_H5 | Yes |
| Output | Hex3_D | Location | PIN_H4 | Yes |
| Output | Hex3_E | Location | PIN_G3 | Yes |
| Output | Hex3_F | Location | PIN_D2 | Yes |
| Output | Hex3_G | Location | PIN_D1 | Yes |
| Output | Hex4_A | Location | PIN_J2 | Yes |
| Output | Hex4_B | Location | PIN_J1 | Yes |
| Output | Hex4_C | Location | PIN_H2 | Yes |
| Output | Hex4_D | Location | PIN_H1 | Yes |
| Output | Hex4_E | Location | PIN_F2 | Yes |
| Output | Hex4_F | Location | PIN_F1 | Yes |
| Output | Hex4_G | Location | PIN_E2 | Yes |
| Output | Hex1_A | Location | PIN_F4 | Yes |

Figure 17 (Appendix 5) - PIN Assignments

# 4 Alternate Design Considerations

## 4.1 Alternatives Considered

During the process of creating the TRISC Processor I encountered many decisions that I needed to make. I ended up having to restart my design due to a file management issue and I think this ended up really helping me out because while I lost images of the original it brought up the better design of placing all of the components in the center of all of the Bus lines so that the design would look much neater and be more function to edit. I also had considered designing the ALU as a set of general circuits rather than Verilog HDL because that was a system that at the time I was more familiar with.

## 4.2 Reasons for Selection of Final Design

I ended up deciding to create my design based on the circular Bus design because in my first iteration of the design, I found things to not look very neat and I really wanted the final design to resemble the Hierarchical Design Structure that I was provided so that the overall design would be easier to troubleshoot. This ended up being a great decision for that step.

I also had to make the decision of making the ALU from Verilog and not a general schematic. This decision was very helpful in the end as the design was very easy to understand and gave me a much better understanding of Verilog.

# 5 Integration and Test Plan

## 5.1 Integration Strategy

This project was split into 2 parts which really helped to easily integrate things in an easy manner. I started by just interacting with the RAM module and the controller to make sure that I was able to communicate with it properly. This step required the PC, but did not require the ACC or ALU. I then added the ACC and the few commands associated with it. Once I finished that step I was able to easily integrate the ALU into the design giving the final design I have today. This way of incremental improvements to the design allowed for me to find errors in the design early on before I got to the finished product better allowing me to find issues in the design.

## 5.2 Test Strategy

For testing the design, I found it easiest to just use the DE1, as I found it difficult to create multiple timing diagrams and I prefer to see the large number of outputs on the HEX displays ratter then as binary outputs in the timing diagrams. I did find the Controller to be easier to design without the DE1 and for that part I did opt into using the timing diagrams to verify the design of the Controller.

## 5.3 Simulation Results from Quartus II

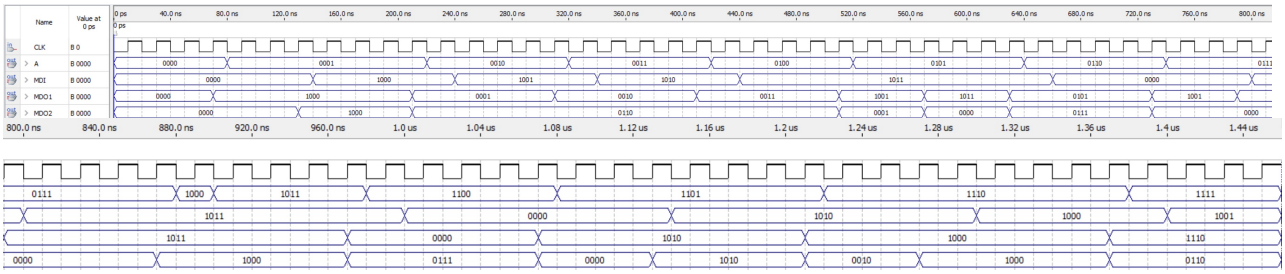

Figure 18 - Timing Diagram

## 5.4 Test Results from DE1 Implementation

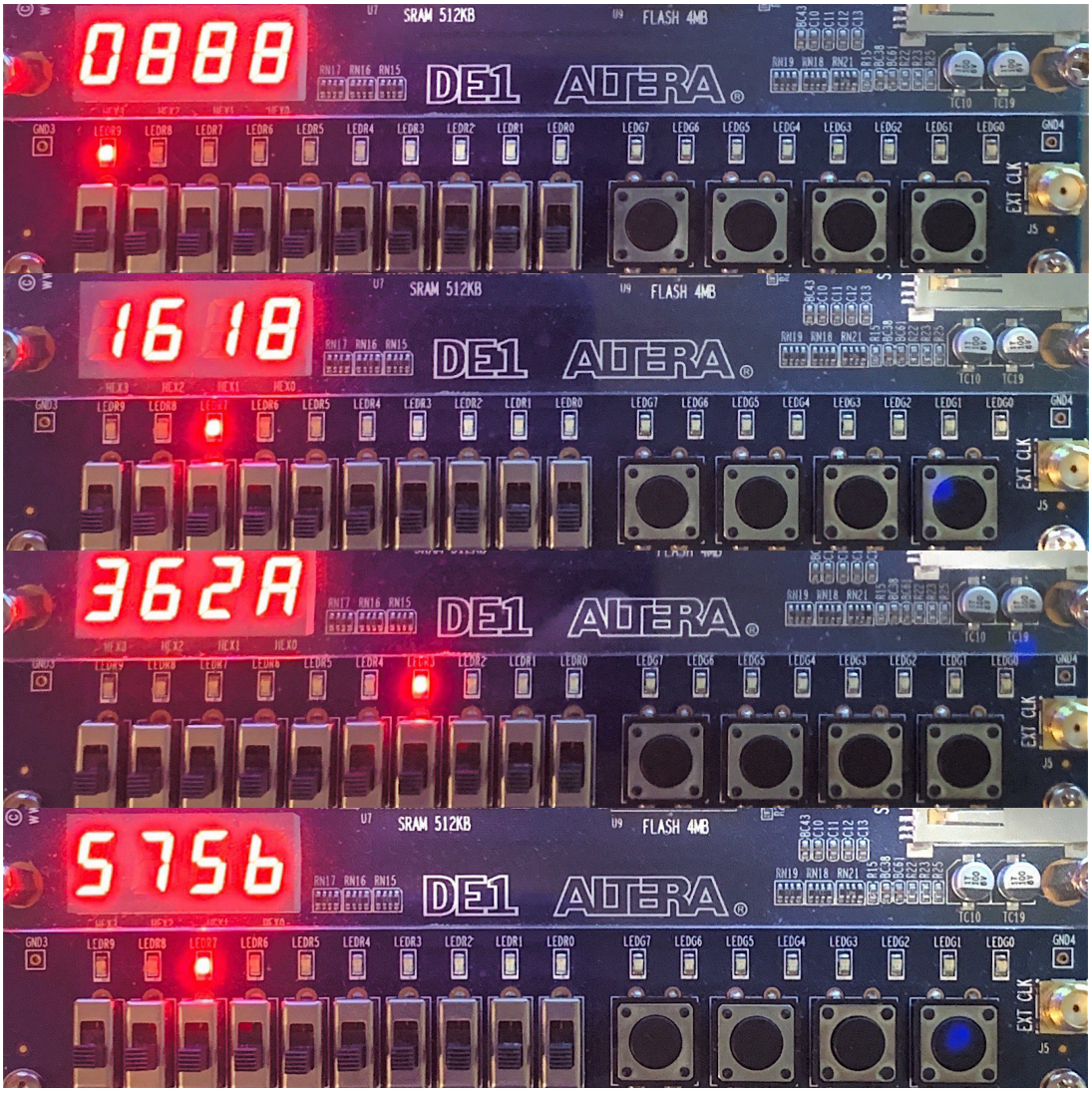


Figure 19 - DE1 Test Run

# 6 Conclusion

## 6.1 Resolution of design

The first time I started to design the TRISC processor I decided to just go for it and get to the final design and while it did end up working out that was it took a log of troubleshooting and that was very time consuming. When I had to rebuild my design. I took a much better approach at splitting the problem up to better solve the sub problems.

A large number of the issues that I encountered where the order of the binary bitts as in I would flip the MSB and the LSB. This could cause some issues in the future that would not be visible in some of the earlier steps.

## 6.2 Lessons Learned

I think the biggest thing that I learned in this whole process was the use of Verilog. I did not feel confident in that design style and think that after designing the ALU and then the controller in that system it allowed for me to get a very good understanding of how that style can have many advantages over standard schematic methods.

I also found that I was able to largely improve my design process. This allowed for me to successfully design this project and allowed me to better split are large problem into smaller sub problems.

# APPENDIX

APPENDIX 1:

```
module SELECTOR (A3,A2,A1,A0,B3,B2,B1,B0,C3,C2,C1,C0,R3,R2,R1,R0,C,V,S1,S0,Cout,OVR);
   input A3,A2,A1,A0,B3,B2,B1,B0,C3,C2,C1,C0,S1,S0,C,V;
   output reg R3,R2,R1,R0,Cout,OVR;
      always @ (S1,S0)
         case ({S1,S0})
            2'b10:begin
            R0 <= B0;
            R1 <= B1;
            R2 <= B2;
            R3 <= B3;
            Cout = 0;
            OVR = 0;
            end
            2'b11:begin
            R0 <= C0;
            R1 <= C1;
            R2 <= C2;
            R3 <= C3;
            Cout = 0;
            OVR = 0;
            end
            default:begin
            R0 <= A0;
            R1 <= A1;
            R2 <= A2;
            R3 <= A3;
            Cout <= C;
            OVR <= V;
            end
         endcase
endmodule
```

APPENDIX 2:

```
module ADDER_SUBTRACTOR(S3, S2, S1, S0, C, V, A3, A2, A1, A0, B3, B2, B1, B0, Op);
   output S3,S2,S1,S0;   // The 4-bit sum/difference..
   wire Cout;
   output C;
   output   V;   // The 1-bit overflow status.
   input A3,A2,A1,A0;   // The 4-bit augend/minuend.
   input B3,B2,B1,B0;   // The 4-bit addend/subtrahend.
   input    Op;  // The operation: 0 => Add, 1=>Subtract.

   wire  C0; // The carry out bit of fa0, the carry in bit of fa1.
   wire  C1; // The carry out bit of fa1, the carry in bit of fa2.
   wire  C2; // The carry out bit of fa2, the carry in bit of fa3.
   wire  C3; // The carry out bit of fa2, used to generate final carry/borrrow.

   wire  R0; // The xor'd result of B[0] and Op
   wire  R1; // The xor'd result of B[1] and Op
   wire  R2; // The xor'd result of B[2] and Op
   wire  R3; // The xor'd result of B[3] and Op



   xor(R0, B0, Op);
   xor(R1, B1, Op);
   xor(R2, B2, Op);
   xor(R3, B3, Op);
   xor(Cout, C3, Op);     // Carry = C3 for addition, Carry = not(C3) for subtraction.
   xor(V, C3, C2);     // If the two most significant carry output bits differ, then we have an overflow.

   xor(C, Op, Cout);


   full_adder fa0(S0, C0, A0, R0, Op);    // Least significant bit.
   full_adder fa1(S1, C1, A1, R1, C0);
   full_adder fa2(S2, C2, A2, R2, C1);
   full_adder fa3(S3, C3, A3, R3, C2);    // Most significant bit.
endmodule // ripple_carry_adder_subtractor


module full_adder(S, Cout, A, B, Cin);
   output S;
   output Cout;
   input  A;
   input  B;
   input  Cin;

   wire   w1;
   wire   w2;
   wire   w3;
   wire   w4;

   xor(w1, A, B);
   xor(S, Cin, w1);
   and(w2, A, B);
   and(w3, A, Cin);
   and(w4, B, Cin);
   or(Cout, w2, w3, w4);
endmodule
```

APPENDIX 3:

```
module LabID(S3,S2,S1,S0,O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0);
    input S3,S2,S1,S0;
    output reg O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0;
    always @ (S3,S2,S1,S0) begin
        case ({S3,S2,S1,S0})
            4'b0000: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000000001; //LDA
            4'b0001: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000000010; //STA
            4'b0010: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000000100; //ADD
            4'b0011: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000001000; //SUB
            4'b0100: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000010000; //XOR
            4'b0110: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000100000; //INC
            4'b0111: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00001000000; //CLR
            4'b1000: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00010000000; //JMP
            4'b1100: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00100000000; //JPZ
            4'b1001: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b01000000000; //JPN
            4'b1111: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b10000000000; //HLT
            default: {O10,O9,O8,O7,O6,O5,O4,O3,O2,O1,O0} = 11'b00000000000; //ERR
        endcase
    end
endmodule
```

APPENDIX 4:

```
module TRISC_Controller(S10,S9,S8,S7,S6,S5,S4,S3,S2,S1,S0,CLK,SS,C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0);
   input S10,S9,S8,S7,S6,S5,S4,S3,S2,S1,S0,CLK,SS;
   output reg C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0;
   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, V=5'b10101, W=5'b10110, X=5'b10111, Y=5'b11000, Z=5'b11001, AA=5'b11010, AB=5'b11011, AC=5'b11100,
   AD=5'b11101, AE=5'b11110, AF=5'b11111;
   always @ (negedge CLK, negedge SS) begin
      if (SS==0)state<=A;
         else state <= nextstate;
   end
   always @ (state) begin
      case (state)
         A:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000001; nextstate <= B;end
         B:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000000; nextstate <= C;end
         C:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000010000; nextstate <= D;end
         D:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000010000; nextstate <= E;end

         E:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000010000100; if(S0)nextstate<=F;
         else if(S1)nextstate<=G; else if(S2)nextstate<=H; else if(S3)nextstate<=I; else if(S4)nextstate<=J;
         else if(S5)nextstate<=K;else if(S6)nextstate<=L;else if(S7)nextstate<=M;else if(S8)nextstate<=N; else if(S9)nextstate<=O;
         else if(S10)nextstate<=P; else nextstate<=A;end

         F:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000011000; nextstate <= Q;end //LDA:0
         G:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000111000; nextstate <= S;end //STA:0
         H:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000011000; nextstate <= T;end //ADD:0
         I:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000000; nextstate <= B;end //SUB:0
         J:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000000; nextstate <= B;end //XOR:0
         K:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000001000000000; nextstate <= B;end //INC:0
         L:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000100000000; nextstate <= B;end //CLR:0
         M:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000010; nextstate <= B;end //JMP:0
         N:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000000; nextstate <= B;end //JPZ:0
         O:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000000; nextstate <= B;end //JPN:0
         P:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000000; nextstate <= B;end //HLT:0
         Q:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000011000; nextstate <= R;end //LDA:1
         R:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000100000001000; nextstate <= B;end //LDA:2
         S:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000111000; nextstate <= B;end //STA:1
         T:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000011000; nextstate <= U;end //ADD:1
         U:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b100000000000000; nextstate <= V;end //ADD:2
         V:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000110000000000; nextstate <= B;end //ADD:3

         default:begin {C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0} = 15'b000000000000001; nextstate <= B;end
      endcase
   end
endmodule
```

APPENDIX 5:

| | | | |
|---|---|---|---|
| in CLK | Location | PIN_R21 | Yes |
| in SS | Location | PIN_R22 | Yes |
| out C2 | Location | PIN_Y18 | Yes |
| out C3 | Location | PIN_R18 | Yes |
| out C4 | Location | PIN_U18 | Yes |
| out C0 | Location | PIN_R17 | Yes |
| out C7 | Location | PIN_V19 | Yes |
| out C1 | Location | PIN_U19 | Yes |
| out C8 | Location | PIN_T18 | Yes |
| out C9 | Location | PIN_Y19 | Yes |
| out Hex1_B | Location | PIN_D5 | Yes |
| out Hex1_C | Location | PIN_D6 | Yes |
| out Hex1_D | Location | PIN_J4 | Yes |
| out Hex1_E | Location | PIN_L8 | Yes |
| out Hex1_F | Location | PIN_F3 | Yes |
| out Hex1_G | Location | PIN_D4 | Yes |
| out Hex2_A | Location | PIN_G5 | Yes |
| out Hex2_B | Location | PIN_G6 | Yes |
| out Hex2_C | Location | PIN_C2 | Yes |
| out Hex2_D | Location | PIN_C1 | Yes |
| out Hex2_E | Location | PIN_E3 | Yes |
| out Hex2_F | Location | PIN_E4 | Yes |
| out Hex2_G | Location | PIN_D3 | Yes |
| out Hex3_A | Location | PIN_E1 | Yes |
| out Hex3_B | Location | PIN_H6 | Yes |
| out Hex3_C | Location | PIN_H5 | Yes |
| out Hex3_D | Location | PIN_H4 | Yes |
| out Hex3_E | Location | PIN_G3 | Yes |
| out Hex3_F | Location | PIN_D2 | Yes |
| out Hex3_G | Location | PIN_D1 | Yes |
| out Hex4_A | Location | PIN_J2 | Yes |
| out Hex4_B | Location | PIN_J1 | Yes |
| out Hex4_C | Location | PIN_H2 | Yes |
| out Hex4_D | Location | PIN_H1 | Yes |
| out Hex4_E | Location | PIN_F2 | Yes |
| out Hex4_F | Location | PIN_F1 | Yes |
| out Hex4_G | Location | PIN_E2 | Yes |
| out Hex1_A | Location | PIN_F4 | Yes |