TRISC Implementation

Using Quartus II and Altera's DE-1 Board

By: William Wallace

CSE 2441-FALL 2017

12/5/17

Table of Contents

Figures

2. Introduction

2.1 Project Overview

This project was assigned on November 9th and was demoed on the 4th of December 2017. The goal of this project was to design, test, and realize a Tiny Reduced Instruction Set Computer or TRISC. The reasoning behind the name TRISC is based on the CPU holding a limited or reduced set of instructions compared to typical CPU instruction set. The CPU was built to read and execute instructions from a random-access memory module, RAM, using various other components like a counter to execute the decoded instructions. For this project, a step by step approach was implemented to complete the project in a timely manner. The project can be divided up into several smaller components which were designed in previous labs or could be designed using previously realized components. These components were designed in Quartus II, software produced by Altera used to design programmable logic devices. Altera's DE-1 was also used to test and program the TRISC onto the DE-1's field-programmable gate array chip (FPGA).

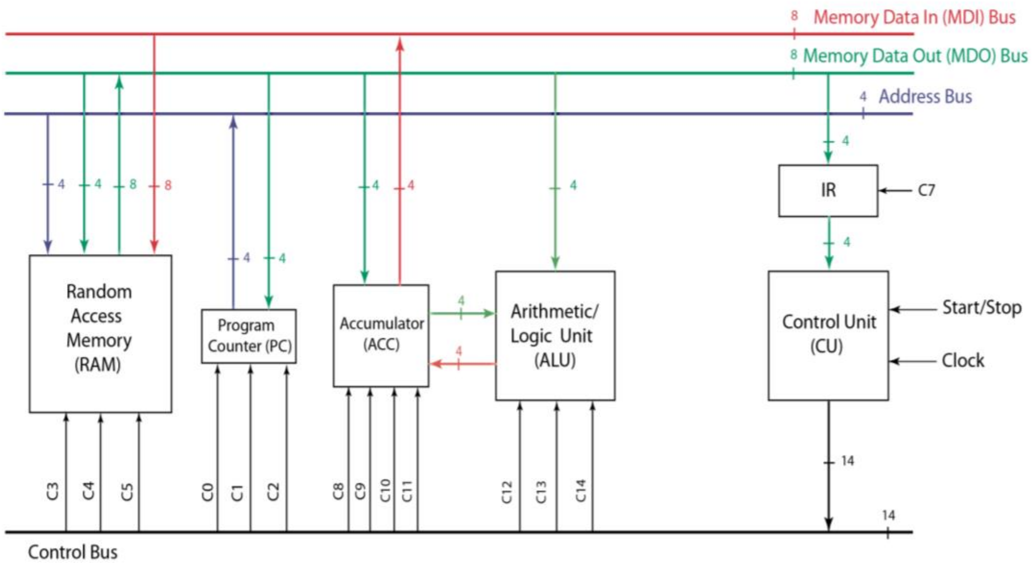

8 Memory Data In (MDI) Bus
8 Memory Data Out (MDO) Bus
4 Address Bus
Random Access Memory (RAM)
Program Counter (PC)
Accumulator (ACC)
Arithmetic/ Logic Unit (ALU)
IR
C7
Control Unit (CU)
Start/Stop
Clock
C3 C4 C5 C0 C1 C2 C8 C9 C10 C11 C12 C13 C14
14
Control Bus

Figure 1 – Part C Taken from Dr. Bill Carroll's in class slides

The requirements for this project were divided into three distinct parts. For the first part, a controller was designed to implement a clear and increment instruction from the given RAM by connecting the following components together: RAM, counter, register, accumulator, controller, instruction decoder, and binary to seven hex components to check the outputs of the various buses of the TRISC design. Testing was of course done on these components before implementation into the TRISC design to allow for fewer design errors in the future and not compromise the entire design due to faulty components. This first part, A, the instruction decoder should read from RAM, send the instruction to the controller and execute that instruction. Even though many of the parts were previously designed in previous labs, the challenge was the integration of these components and how to design the controller to handle the various states of the system of the TRISC as a whole.

The second part, B, had the same components as part A, but had additional instructions that had to be integrated into the controller. The controller, designed as a finite state machine using Verilog, had to be changed to allow for the additional states the load and store instructions required to function properly. Verilog is a hardware description language used to model electronic systems, and was used for some of the components like the controller and instruction decoder (the Verilog for the controller will be expanded upon in the fourth section of this report). The store instruction copies data from the accumulator to a specific address in memory, and the load instruction transfers data from memory at a specific address to the accumulator. Control signals were also added to the control bus of the design to accommodate the additional states that were added to the controller.

The final task for this project was implementing an additional two instructions, add and jump, which required an ALU (arithmetic logic unit) and register in tandem with the ALU. The controller was manipulated to add the additional set of instructions and more signals were added to the control bus to reflect the controller changes. For all three parts of this project, a pre-build RAM module was given. Each RAM module had a set of instructions used to test the functionality of each part.

2.2 Project Status

As of writing this report, all project requirements were met and correctly implemented. The extra instructions: SUB, XOR, JPZ, JPN, and HLT were not put into this project. The Op codes of the instructions supported by this TRISC design are shown below in figure 2 as well as the instructions that were not implemented in this design. The ones implemented in the design are highlighted in yellow.

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

Figure 2

2.3 Report Overview

This report will provide an overview of the entire project. The system design, and controller design will be described in detail in their own respective sections as well as a discussion of what alternate design choices that could have been made. Figures will also be provided to provide a visual aid as well as clarify what is currently being discussed and to clarify concepts. The report will also detail the testing process and the strategy used to implement all the components needed for the TRISC design. Results of the simulations will also be provided in this report, including a resolution of the design and the issues encountered throughout the process of producing the TRISC.

3. System Design

3.1 System-level description and diagrams

Shown below in figure 3, is the schematic of the finished TRISC system (Part C). The controller can be seen on the lower right side of the figure and is labeled as “IDcontrol”. The controller will be discussed in greater detail in a later section. The IDcontrol block diagram hooks up to the parallel in parallel out register located just above it and is itself connected to the memory data out bus, labeled as “q[7..0]”. Looking back towards the leftmost side of the figure is where the RAM module is located. The RAM is connected to a multiplexor (74157-block file supplied by Quartus) through its address bus and splits into eight signals. Four of the signals go to another address bus labeled “addr[3..0]” and the other four go to a bus dedicated to the ALU labeled as “alu[3..0]”. Next, the counter is located to the right of the RAM module and is connected to the memory data out bus (“q[3..0]”) and the secondary address bus (“addr[3..0]”). This acts as the system counter, this counter should display an incrementing number when the clock is triggered. To the right of the counter is the accumulator. The accumulator is made up of another counter and a multiplexor (74157). The multiplexor gets input from both the memory data out (“q[3..0]”) and the alu (“alu[3..0]”) bus. The output is hooked up to the counter and is finally outputted to the memory data in bus (“data[7..0]”). Finally the last component to look at for the system level analysis is the ALU and its register. The ALU connects to the memory data out and memory data in buses, while its output is sent to a register. The register outputs its signals to the alu bus where it is again sent to the accumulator’s multiplexor. Cut off on the top of the figure are the four binary to hex components, these are used to output on the hex display on the DE-1. Most if not all of the components at the system level has an input that originates from the control bus (which is the output of the IDcontroller).

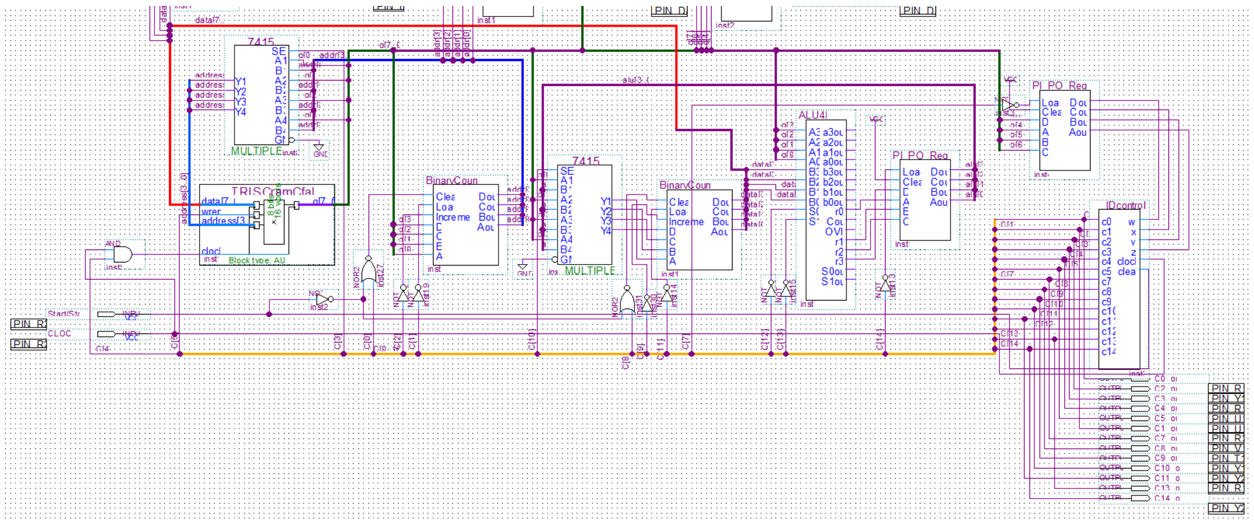


Figure 3

3.2 Subsystem descriptions and diagrams

In figure 4 shown below is the accumulator used in the system. The role of the accumulator in this system is to hold the data that has been fetched from memory, with an additional increment function which increments the currently stored value. Just as was previously stated in the system level analysis, the accumulator encompasses the multiplexor and the counter. The multiplexor hooks up the memory data out bus and the alu bus and the selector input is used to choose between the two. The counter, in addition to having output that goes to the memory data in bus, has four control signals as input. The four control signals are used to execute a command interpreted from the RAM by the instruction decoder/controller and sent to the control bus. As an example, to execute the CLR instruction a signal would be sent to the clear input of the counter from the C[8] control signal. This clear instruction clears the accumulator, which essentially resets back to the original value of 0.

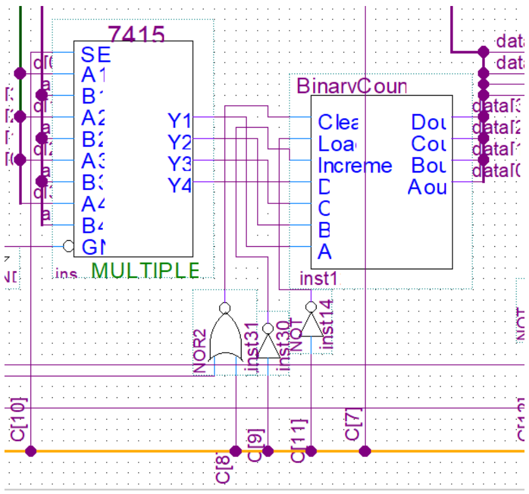


Figure 4

Another component used for the TRISC was the counter. The schematic for the counter used is shown below in figure 5. The design for the counter compared to the other components is relatively simple. The 74193 chip was used to realize the counter and has an active low load as an input. The up and clear inputs have not gates due to a previous lab assignment and had to be considered for the gates surrounding the counter at the system level and to some extent the controller itself. Since only the up counter is used on the chip the “DN” input was set to logical 1 or Vcc. The output of the counter provides the address of the next instruction on the memory. The counter’s role in the system is to increment the address pointer, which is essentially an incremental read from the RAM. When a JMP instruction is encountered the counter loads the address to jump to rather than increment a stored value.

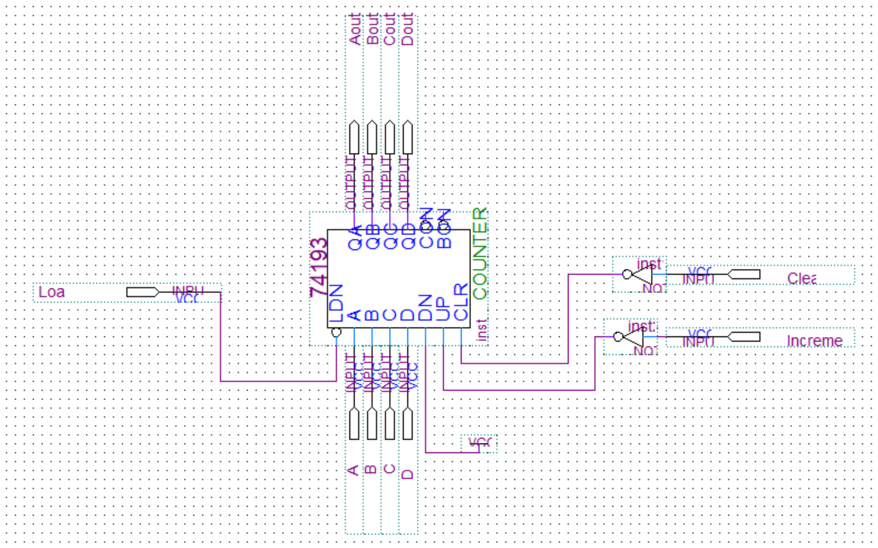


Figure 5

The parallel in parallel out register was another component used in the system. The register, displayed in figure 6, has an active low clear which is dealt with at the system level with Vcc as input to keep it inactive. A not gate is also used for the clock for this design. This is again due to having reused assets from previous lab assignments. Another not gate was used at the system level to negate the not gate used at this level of the system. Two of these registers were used at the final stage of this project, one for the ALU and one for the instruction decoder.

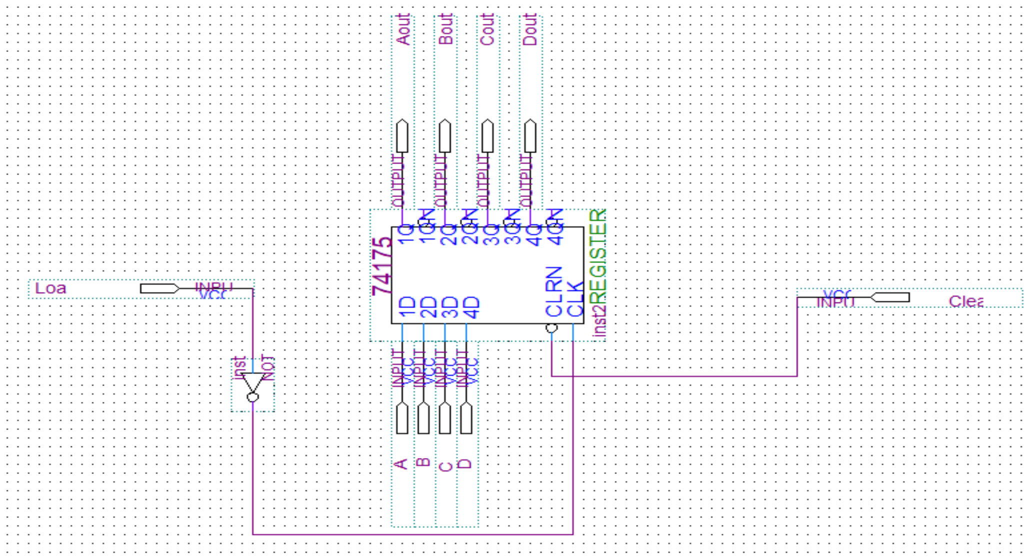


Figure 6

The last major component used in this TRISC realization was the ALU. The ALU was required in part C of this project to execute the ADD instruction. The ALU, pictured below in figure 7, is a four bit ALU. This makes it possible to add, subtract, AND, and XOR four bit numbers. Although this ALU can AND or XOR numbers, the controller was not designed to execute the instructions needed to run those functions. Towards the center of the figure is the component with the name “RippleCarryAdder”. This subcomponent of the ALU is used to perform the addition and subtraction of four bit numbers (only the ADD instruction was enabled for this project). Finally, a multiplexor was used to select between the ANDXOR output and the “RippleCarryAdder”. Because the only the adder was used the selector for the multiplexor, both selectors of the ALU were given not gates at the system level. Figure 8, shows the design of the adder/subtractor block diagram in the ALU. The schematic for the “RippleCarryAdder” can be seen in figure 8, and the schematic for the full adders used in the adder is shown in figure 9.

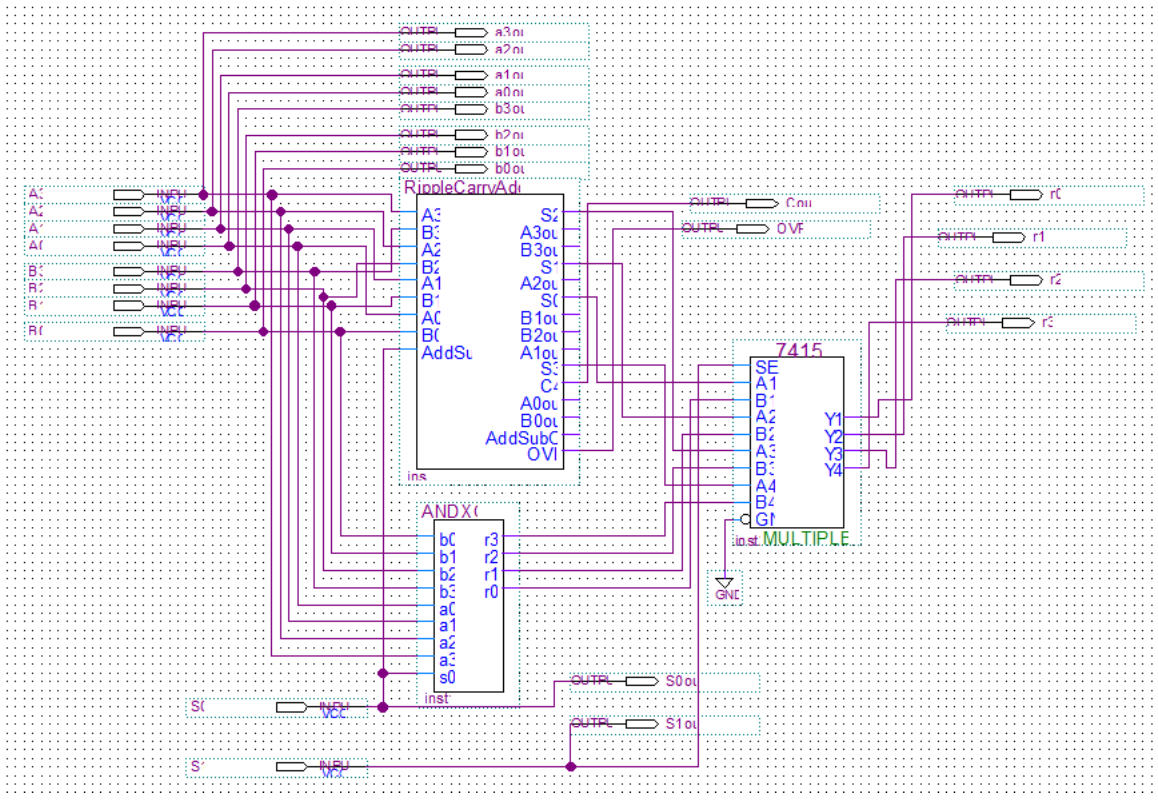


Figure 7

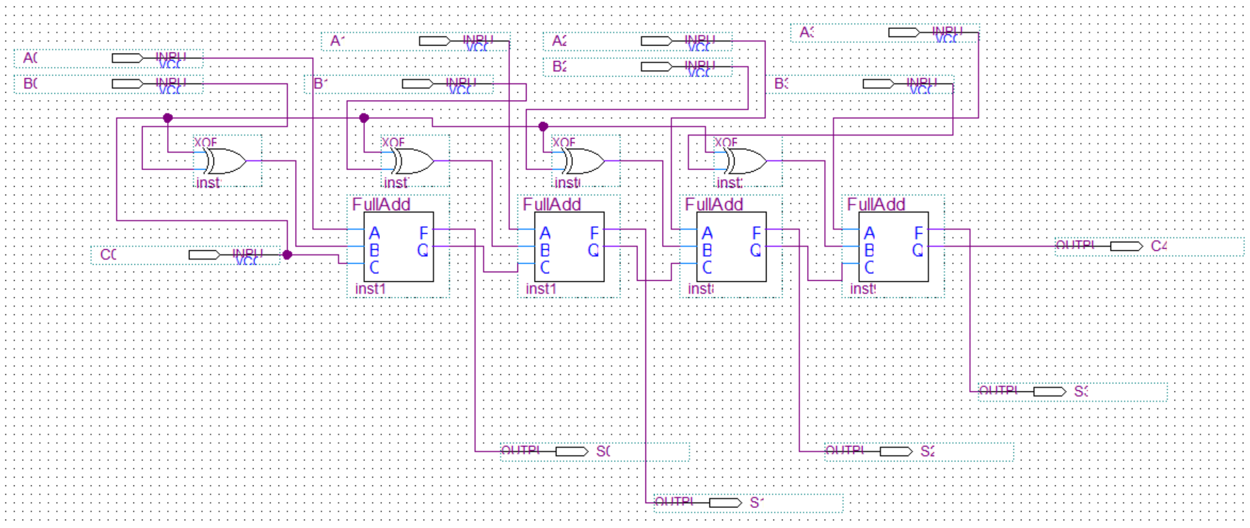


Figure 8

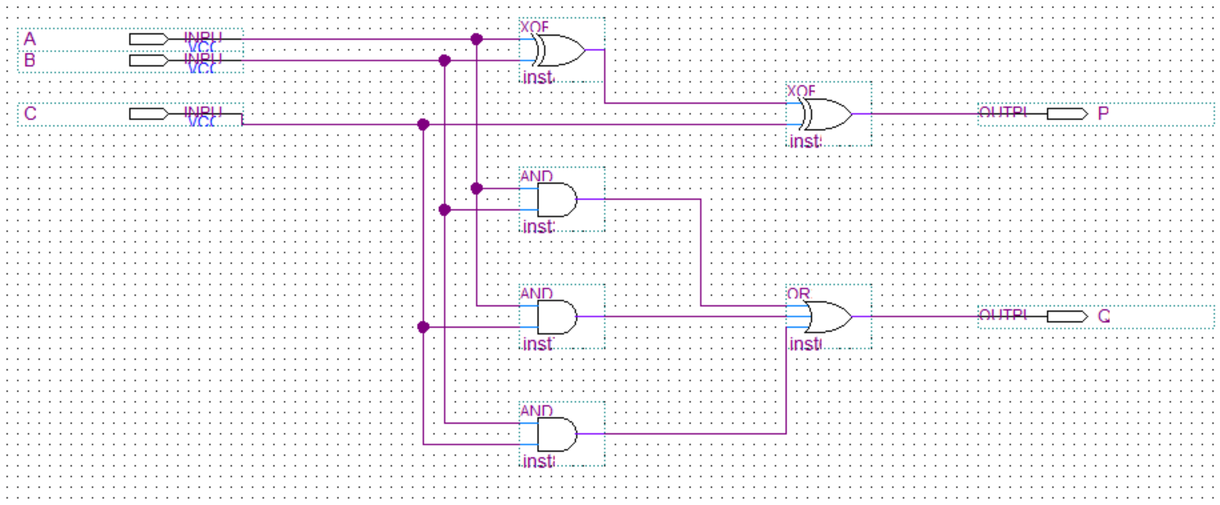
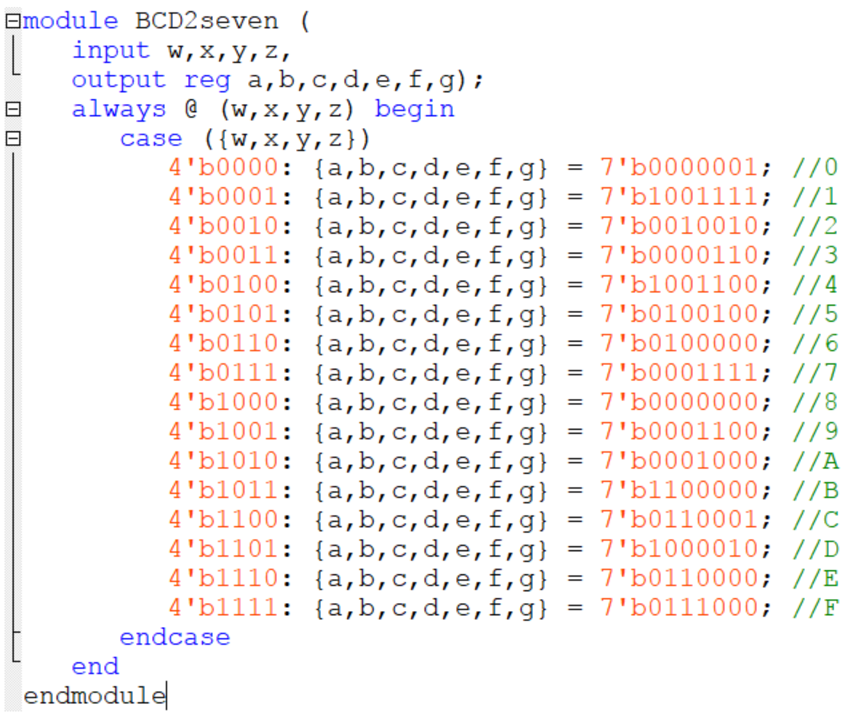


Figure 9

Finally, four binary to seven hex Verilog components were used to display data outputted from the three buses: memory data in, memory data out, and address bus. This was essential for understanding if the correct addresses were read, the instructions were executed correctly, and if the TRISC design was implemented correctly. The Verilog used to create this component is shown below in figure 10.



```
module BCD2seven (
    input w,x,y,z,
    output reg a,b,c,d,e,f,g);
    always @ (w,x,y,z) begin
        case ({w,x,y,z})
            4'b0000: {a,b,c,d,e,f,g} = 7'b0000001; //0
            4'b0001: {a,b,c,d,e,f,g} = 7'b1001111; //1
            4'b0010: {a,b,c,d,e,f,g} = 7'b0010010; //2
            4'b0011: {a,b,c,d,e,f,g} = 7'b0000110; //3
            4'b0100: {a,b,c,d,e,f,g} = 7'b1001100; //4
            4'b0101: {a,b,c,d,e,f,g} = 7'b0100100; //5
            4'b0110: {a,b,c,d,e,f,g} = 7'b0100000; //6
            4'b0111: {a,b,c,d,e,f,g} = 7'b0001111; //7
            4'b1000: {a,b,c,d,e,f,g} = 7'b0000000; //8
            4'b1001: {a,b,c,d,e,f,g} = 7'b0001100; //9
            4'b1010: {a,b,c,d,e,f,g} = 7'b0001000; //A
            4'b1011: {a,b,c,d,e,f,g} = 7'b1100000; //B
            4'b1100: {a,b,c,d,e,f,g} = 7'b0110001; //C
            4'b1101: {a,b,c,d,e,f,g} = 7'b1000010; //D
            4'b1110: {a,b,c,d,e,f,g} = 7'b0110000; //E
            4'b1111: {a,b,c,d,e,f,g} = 7'b0111000; //F
        endcase
    end
endmodule
```

Figure 10

3.3 Hierarchical design structure

The hierarchical design of the system was planned with the previously build components in mind. The counter, registers, binary to hex, and ALU all have their own respective symbol files from previous assignments. The multiplexors were pre-built by Quartus II and the RAM files were provided. Only the accumulator, instruction decoder, and controller were designed from scratch during this project. The accumulator was designed at the system level and doesn't have its own symbol file. The instruction decoder was designed using Verilog along with the controller. Both the instruction decoder and controller were connected within its own symbol file named "IDcontroller" (This symbol file was placed at the system level).

3.4 Operating procedure

Altera's DE-1 was used to operate the TRISC system designed in this project. Two push buttons were used and assigned for the clock and clear inputs. A user would repeated press the clock push button to run the system. As the user pushes the clock button the RAM's instructions would be read, decoded, and executed by the controller. The left most seven-bit hex display was assigned to the program counter, the center two were assigned to the memory address, and the right most hex display was assigned the accumulator. The user could look at the four displays to ensure that the system is running without errors and see the instructions being carried out using the instructions loaded onto the RAM.

4. Controller design details

4.1 Functional description and diagram showing I/O

The controller for this project was designed using states. The controller gets its input from the instruction decoder. This input has one instruction on logical 1, and is sent through a series of states. All the instructions for this project had the same first five states A, B, C, D, and E. State E is where the instructions branch off depending on the input of the controller (the state diagrams will be discussed next section). A total of 19 states were needed which required 5 bits for the parameters to accommodate that many states. Figure 11 on the right shows the block diagram of the controller. The right side of the diagram shows all the inputs while the left shows the control signal outputs. These outputs will be hooked up to the control bus located at the system level.

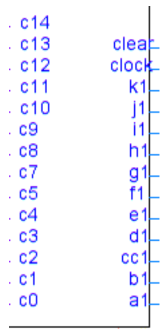


Figure 11

The Verilog for both the instruction decoder and controller as well as the connections between both components are in section 4.3 of the report and are figures 12, 13, and 14.

4.2 State Diagrams

State diagrams were made for each implemented instruction. Each state would output some combination of control signals out to the control bus which would then feed into the various components in the system. Here are the various state diagrams produced for this project along with the control signals that relate to each state:

CLR: A/C0 -> B/C3 -> C/C4 -> D/C4 -> E/C2, C7 -> G/C8 -> return to B

INC: A/C0 -> B/C3 -> C/C4 -> D/C4 -> E/C2, C7 -> F/C9 -> return to B

STA: A/C0 -> B/C3 -> C/C4 -> D/C4 -> E/C2, C7 -> H/0 -> L/C5 -> M/C4, C5 -> N/C4 -> return to B

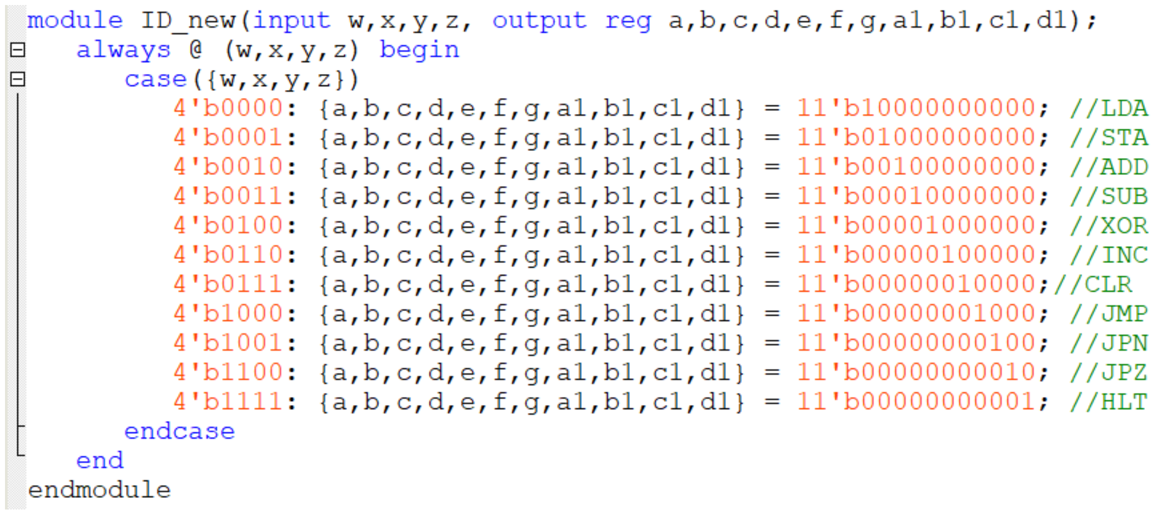
LDA: A/C0 -> B/C3 -> C/C4 -> D/C4 -> E/C2, C7 -> H/C3* -> I/C4 -> J/C4 -> K/C11 -> return to B

ADD: A/C0 -> B/C3 -> C/C4 -> D/C4 -> E/C2, C7 -> H/0 -> J/C4 -> P/C10 -> Q/C10, C12, C13 -> R/C10, C14 -> S/C10, C11 -> return to B

JMP: A/C0 -> B/C3 -> C/C4 -> D/C4 -> E/C2, C7 -> O/C1 -> return to B

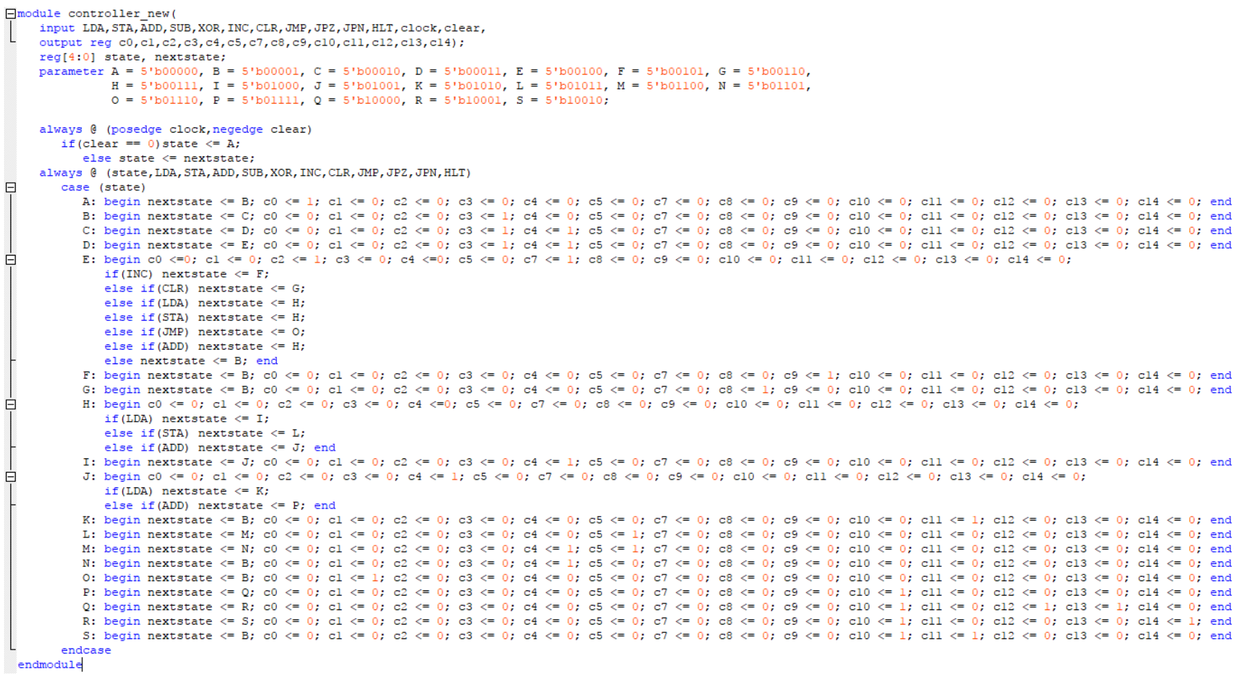
4.3 Verilog code

The following figures below show the Verilog of the instruction decoder (figure 12), the controller (figure 13), and the last figure shows the connections between them (figure 14).



```
module ID_new(input w,x,y,z, output reg a,b,c,d,e,f,g,a1,b1,c1,d1);
    always @ (w,x,y,z) begin
        case({w,x,y,z})
            4'b0000: {a,b,c,d,e,f,g,a1,b1,c1,d1} = 11'b10000000000; //LDA
            4'b0001: {a,b,c,d,e,f,g,a1,b1,c1,d1} = 11'b01000000000; //STA
            4'b0010: {a,b,c,d,e,f,g,a1,b1,c1,d1} = 11'b00100000000; //ADD
            4'b0011: {a,b,c,d,e,f,g,a1,b1,c1,d1} = 11'b00010000000; //SUB
            4'b0100: {a,b,c,d,e,f,g,a1,b1,c1,d1} = 11'b00001000000; //XOR
            4'b0110: {a,b,c,d,e,f,g,a1,b1,c1,d1} = 11'b00000100000; //INC
            4'b0111: {a,b,c,d,e,f,g,a1,b1,c1,d1} = 11'b00000010000;//CLR
            4'b1000: {a,b,c,d,e,f,g,a1,b1,c1,d1} = 11'b00000001000; //JMP
            4'b1001: {a,b,c,d,e,f,g,a1,b1,c1,d1} = 11'b00000000100; //JPN
            4'b1100: {a,b,c,d,e,f,g,a1,b1,c1,d1} = 11'b00000000010; //JPZ
            4'b1111: {a,b,c,d,e,f,g,a1,b1,c1,d1} = 11'b00000000001; //HLT
        endcase
    end
endmodule
```

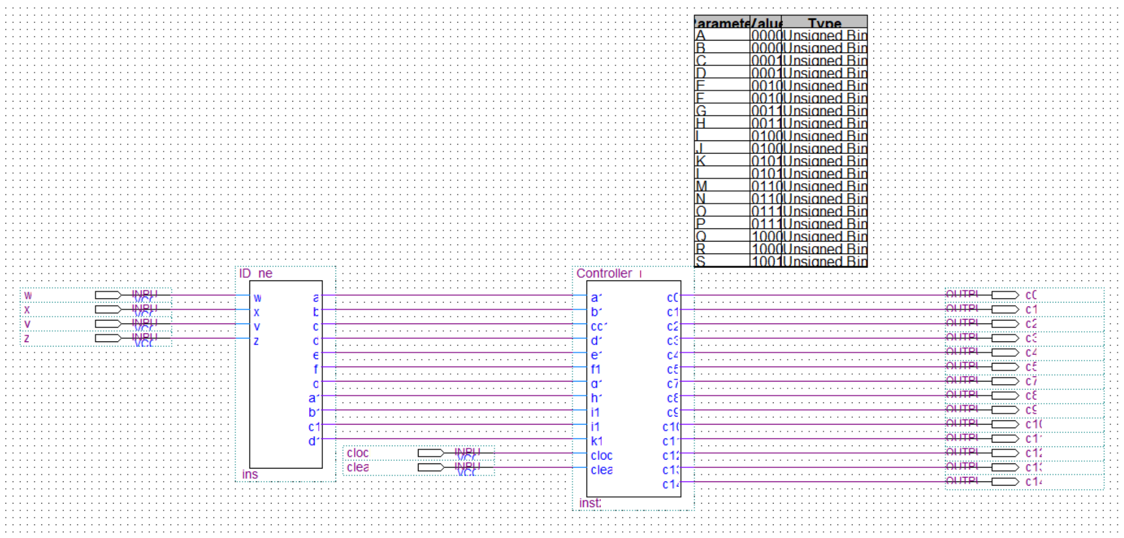
Figure 12



```
module controller_new(
   input LDA,STA,ADD,SUB,XOR,INC,CLR,JMP,JPZ,JPN,HLT,clock,clear,
   output reg c0,c1,c2,c3,c4,c5,c7,c8,c9,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;

   always @ (posedge clock,negedge clear)
      if(clear == 0)state <= A;
         else state <= nextstate;
   always @ (state,LDA,STA,ADD,SUB,XOR,INC,CLR,JMP,JPZ,JPN,HLT)
      case (state)
         A: begin nextstate <= B; c0 <= 1; c1 <= 0; c2 <= 0; c3 <= 0; c4 <= 0; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0; end
         B: begin nextstate <= C; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 1; c4 <= 0; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0; end
         C: begin nextstate <= D; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 1; c4 <= 1; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0; end
         D: begin nextstate <= E; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 1; c4 <= 1; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0; end
         E: begin c0 <=0; c1 <= 0; c2 <= 1; c3 <= 0; c4 <=0; c5 <= 0; c7 <= 1; c8 <= 0; c9 <= 0; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0;
            if(INC) nextstate <= F;
            else if(CLR) nextstate <= G;
            else if(LDA) nextstate <= H;
            else if(STA) nextstate <= H;
            else if(JMP) nextstate <= O;
            else if(ADD) nextstate <= H;
            else nextstate <= B; end
         F: begin nextstate <= B; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0; c4 <= 0; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 1; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0; end
         G: begin nextstate <= B; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0; c4 <= 0; c5 <= 0; c7 <= 0; c8 <= 1; c9 <= 0; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0; end
         H: begin c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0; c4 <=0; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0;
            if(LDA) nextstate <= I;
            else if(STA) nextstate <= L;
            else if(ADD) nextstate <= J; end
         I: begin nextstate <= J; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0; c4 <= 1; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0; end
         J: begin c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0; c4 <= 1; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0;
            if(LDA) nextstate <= K;
            else if(ADD) nextstate <= P; end
         K: begin nextstate <= B; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0; c4 <= 0; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 0; c11 <= 1; c12 <= 0; c13 <= 0; c14 <= 0; end
         L: begin nextstate <= M; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0; c4 <= 0; c5 <= 1; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0; end
         M: begin nextstate <= N; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0; c4 <= 1; c5 <= 1; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0; end
         N: begin nextstate <= B; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0; c4 <= 1; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0; end
         O: begin nextstate <= B; c0 <= 0; c1 <= 1; c2 <= 0; c3 <= 0; c4 <= 0; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 0; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0; end
         P: begin nextstate <= Q; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0; c4 <= 0; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 1; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 0; end
         Q: begin nextstate <= R; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0; c4 <= 0; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 1; c11 <= 0; c12 <= 1; c13 <= 1; c14 <= 0; end
         R: begin nextstate <= S; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0; c4 <= 0; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 1; c11 <= 0; c12 <= 0; c13 <= 0; c14 <= 1; end
         S: begin nextstate <= B; c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0; c4 <= 0; c5 <= 0; c7 <= 0; c8 <= 0; c9 <= 0; c10 <= 1; c11 <= 1; c12 <= 0; c13 <= 0; c14 <= 0; end
      endcase
endmodule
```

Figure 13

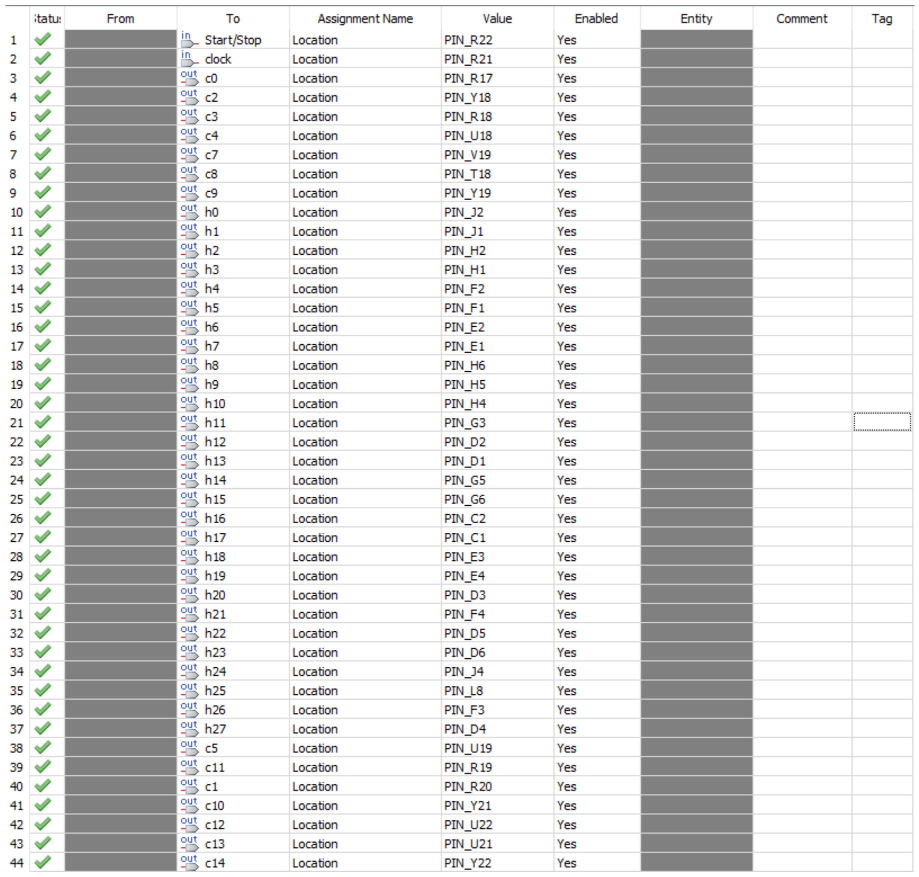


aramete	/alue	Type
A	0000	Unsigned Bin
B	0000	Unsigned Bin
C	0001	Unsigned Bin
D	0001	Unsigned Bin
E	0010	Unsigned Bin
F	0010	Unsigned Bin
G	0011	Unsigned Bin
H	0011	Unsigned Bin
I	0100	Unsigned Bin
J	0100	Unsigned Bin
K	0101	Unsigned Bin
L	0101	Unsigned Bin
M	0110	Unsigned Bin
N	0110	Unsigned Bin
O	0111	Unsigned Bin
P	0111	Unsigned Bin
Q	1000	Unsigned Bin
R	1000	Unsigned Bin
S	1001	Unsigned Bin

Figure 14

4.4 DE1 pin assignments

Here are the pin assignments for the TRISC on the DE-1. In figure 15, it shows Part C of the project. Some additional lights were added for testing and to verify if ALU was adding properly.



	Statu:	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
1	✓		in Start/Stop	Location	PIN_R22	Yes			
2	✓		in clock	Location	PIN_R21	Yes			
3	✓		out c0	Location	PIN_R17	Yes			
4	✓		out c2	Location	PIN_Y18	Yes			
5	✓		out c3	Location	PIN_R18	Yes			
6	✓		out c4	Location	PIN_U18	Yes			
7	✓		out c7	Location	PIN_V19	Yes			
8	✓		out c8	Location	PIN_T18	Yes			
9	✓		out c9	Location	PIN_Y19	Yes			
10	✓		out h0	Location	PIN_J2	Yes			
11	✓		out h1	Location	PIN_J1	Yes			
12	✓		out h2	Location	PIN_H2	Yes			
13	✓		out h3	Location	PIN_H1	Yes			
14	✓		out h4	Location	PIN_F2	Yes			
15	✓		out h5	Location	PIN_F1	Yes			
16	✓		out h6	Location	PIN_E2	Yes			
17	✓		out h7	Location	PIN_E1	Yes			
18	✓		out h8	Location	PIN_H6	Yes			
19	✓		out h9	Location	PIN_H5	Yes			
20	✓		out h10	Location	PIN_H4	Yes			
21	✓		out h11	Location	PIN_G3	Yes			
22	✓		out h12	Location	PIN_D2	Yes			
23	✓		out h13	Location	PIN_D1	Yes			
24	✓		out h14	Location	PIN_G5	Yes			
25	✓		out h15	Location	PIN_G6	Yes			
26	✓		out h16	Location	PIN_C2	Yes			
27	✓		out h17	Location	PIN_C1	Yes			
28	✓		out h18	Location	PIN_E3	Yes			
29	✓		out h19	Location	PIN_E4	Yes			
30	✓		out h20	Location	PIN_D3	Yes			
31	✓		out h21	Location	PIN_F4	Yes			
32	✓		out h22	Location	PIN_D5	Yes			
33	✓		out h23	Location	PIN_D6	Yes			
34	✓		out h24	Location	PIN_J4	Yes			
35	✓		out h25	Location	PIN_L8	Yes			
36	✓		out h26	Location	PIN_F3	Yes			
37	✓		out h27	Location	PIN_D4	Yes			
38	✓		out c5	Location	PIN_U19	Yes			
39	✓		out c11	Location	PIN_R19	Yes			
40	✓		out c1	Location	PIN_R20	Yes			
41	✓		out c10	Location	PIN_Y21	Yes			
42	✓		out c12	Location	PIN_U22	Yes			
43	✓		out c13	Location	PIN_U21	Yes			
44	✓		out c14	Location	PIN_Y22	Yes			

Figure 15

5. Alternative design considerations

5.1 Alternatives considered

A schematic approach (no Verilog) was considered for the instruction decoder. This approach would likely add a greater possibility of more errors. This is due to the possibility of a wire connecting where it should not, flipping the most significant bit in the schematic, or just any error that could occur when designing with gates and wires. The controller could also be designed using gates and flip flops, but the complexity of the wiring would allow for even more errors to occur. While Verilog could have been used to design most, if not all, of the components in the system, it was only used for the instruction decoder and controller for this project. A Verilog approach could have been taken to design, for instance, the accumulator.

5.2 Reasons for selection of final design

The reasoning behind choosing Verilog for the instruction decoder is due to a lesser chance for errors as well as being a more straightforward and relatively less complex endeavor. The same holds true for the controller, where the number of states needed would require at least three flip flops and increasingly complex wiring and gate usage. Verilog can easily be written to handle system states which was perfect for this project.

6. Integration and Test Plan

6.1 Integration strategy

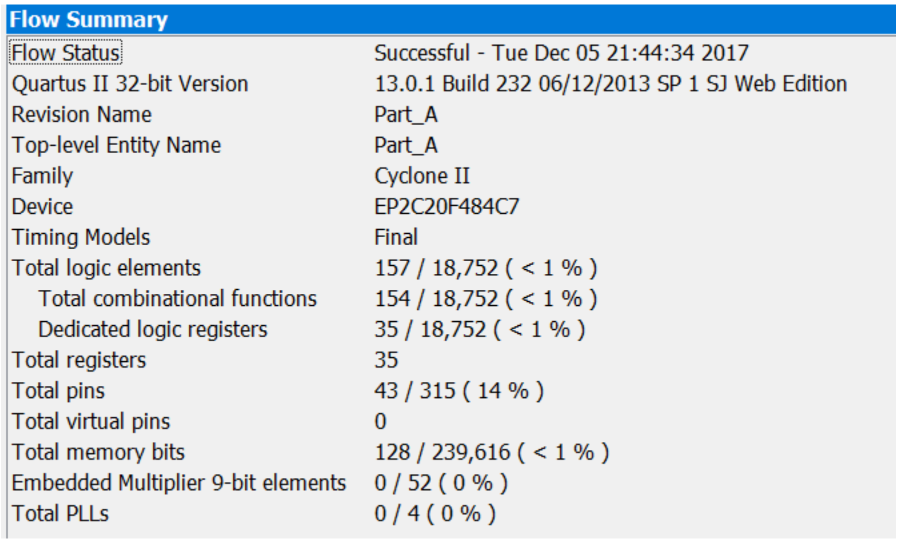
The integration of the components was a major and vital part of the project. A clear understanding of the purpose of each component as well as how they all interact with each other was essential. The strategy taken in integrating all the components was incremental in nature. A couple components were connected at a time, with testing done as frequently as possible. If the accumulator was not functioning properly, thorough testing was done on its connections as well as the accumulator itself. The design for the project was given for parts A and B which outlined the general connections between the components. This guideline was very helpful as a starting point in integration.

6.2 Test strategy

Testing was a very prominent and time consuming portion of this project. Testing was done at every step to ensure that no errors or faulty outputs occur. Testing was first done on each individual component that was going to be used in the final design. From this initial testing, we could conclude that most errors would relate to the connections rather than the components themselves (there are of course exceptions to this conclusion). The hex displays on the DE-1 played a vital role in the testing of the system. When clocking through the program, just by looking at the values output by the counter, address, and accumulator, one could isolate the problem to a specific component or connection.

6.3 Simulation results from Quartus II

Shown below in figure 16 is the simulation results from Quartus II. (Listed as Part A because the project was copied for Part B and Part C. So, this is the compilation report for Part C).



Flow Summary	
Flow Status	Successful - Tue Dec 05 21:44:34 2017
Quartus II 32-bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	Part_A
Top-level Entity Name	Part_A
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	157 / 18,752 (< 1 %)
Total combinational functions	154 / 18,752 (< 1 %)
Dedicated logic registers	35 / 18,752 (< 1 %)
Total registers	35
Total pins	43 / 315 (14 %)
Total virtual pins	0
Total memory bits	128 / 239,616 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 16

6.4 Test results from DE1 implementation

After a series of many tests and many troubleshooting sessions, the correct sequence of instruction addresses/accumulator values were displayed on the DE-1. The RAM was read successfully for all three parts and executed each instruction as well with no errors during each demo. The CLR, INC, STR, LDA, ADD, and JMP instructions were executed properly with the LEDs on the DE-1 also corresponding to the correct states for each instruction as well. Although the instruction decoder supports the other instructions i.e. SUB, XOR, JPZ, JPN, and HLT, the controller and system could be modified to allow for such functionality.

7. Conclusion

7.1 Resolution of design and implementation issues

Many implementation issues were encountered throughout the duration of this project. One major issue was the implementation of the accumulator. The accumulator had its own symbol file in the early stages of the project. This symbol file would not output the correct values for the accumulator and for some cases, just displayed a zero no matter what instruction was executed. This implementation issue was fixed by just designing the accumulator at the system level with no symbol file. One other implementation issue was the implementation of the previously built components, like the counter, into the system. The not gates within the symbol files for these previously built components needed more gates at the system level to either negate them or utilize them. This was a trial and error process that thankfully worked towards the end. The design was completed and demoed on time with all parts executing their respective instructions.

7.2 Lessons learned

One lesson I have learned while working on this project was that starting earlier would have saved me from all the stress and overnight work sessions that I experienced. Another lesson I learned is that testing early and frequently can save a lot of time and can lead to better design choices. This project gave me a better understanding of some digital logic concepts that previously confused me, like the design of a state controller using Verilog. I plan to continue to pursue more projects like this one and to continue expanding my knowledge for the future.