

The TRISC Project

Prepared by: Alejandro Limon Naranjo

Prepared for: Dr. Carroll

ID: 1000818599

Date: December 7th, 2015

Table of Contents

Figures and Tables	3
Introduction	4
Overview.....	4
Status and unresolved issues.....	5
Report Overview	6
System Design	7
System-level descriptions and diagrams	7
Subsystem Diagrams and Descriptions	9
Hierarchical Design Structure	12
Operating Procedure	13
Controller Design Details	14
Functional description showing I/O	14
State Diagrams	15
Controller Verilog Code	17
DE1 Pin Assignments	19
Alternative Design Considerations.....	22
Alternatives Considered	22
Reasons for selection of final design.....	22
Integration and Test Plan	24
Integration Strategy	24
Test Strategy	24
Simulation Results from Quartus	25
Test Results from the DE1	25
Conclusion.....	26
Resolution of Design/Implementation Issues	26
Lessons Learned	26

Figures and Tables

Figure 1. TRISC System Execution.	7
Figure 2. TRISC BSF File.	8
Figure 3. TRISC RAM Part C.	9
Figure 4. Instruction Register.	9
Figure 5. Arithmetic Logic Unit.	10
Figure 6. Address Selector.	10
Figure 7. Accumulator.	11
Figure 8. Full Controller.	11
Figure 9 Hierarchical Structure TRISC.	12
Figure 10. Operating Procedure.	13
Figure 11. Controller I/O.	14
Figure 12. Part A state diagram. <i>Dr. Carroll</i>	15
Figure 13. Part B/C State Diagram.	16
Figure 14. Part A Verilog Code	17
Figure 15. Part A Controller B.	17
Figure 16. Part B/C Verilog Code.	18
Figure 17. DE1 Inputs. <i>Dr. Carroll</i>	19
Table 1. DE1 I/O Pin Assignments. <i>Dr. Carroll</i>	19
Table 2. Part B Controller Pin Assignments.	20
Table 3. Part A Controller Assignments.	20
Table 4. TRISC DE1 Pin Assignments.	21
Figure 17. Full TRISC Design.	23
Figure 18. Simulation Results.	25

Introduction

Overview

The TRISC (Tiny reduced instruction set computer) is a small-scaled computer that can execute 7 different instructions which are: Store from Accumulator (STA), Load from Accumulator (LDA), Jump to Address (JMP), Increment Accumulator (INC), Clear Accumulator (CLR), and Add (ADD). The TRISC architecture is based on 4-bit operations and the components that are required for this design include: Random Access Memory (16 X 8), Instruction Register, Arithmetic Logic Unit, Address Selector/MUX, Accumulator, and a Controller.

The TRISC design is based on Verilog HDL (Hardware Description Language) and Block Schematic Design. All the design was done on the Altera Quartus II application again using the Verilog Editor and Block Schematic Editor.

In order to actually implement and test the TRISC, the designs were programmed into an FPGA (Field Programmable Gate Array). More specifically the Cyclone II (EP2C20F484C7), and the board on which the Cyclone II is mounted on is known as the DE1 (Development and Education Board). To burn the design, the programmer tool in Quartus was used. In order to associate the right switches and LED's on the DE1, the assignment editor was used to instantiate the pin assignments corresponding to the components of DE1 (specific pins are shown later on in the report).

Using the divide and conquer method, this project was split into three different parts to make testing more simplified and efficient. Part A was based on 6 of the 7 components mentioned above. The ALU was not implemented in Part A, and the only true instructions that could be executed at that point were INC and CLR. Based on the instructions specified a controller was created using Verilog. In order to demonstrate and test part A, I ported the clock to one of the KEY buttons on the DE1 board. All the control signals were shown on LED's and the actual values of memory data out, memory data in, and the memory address were shown on seven segment displays (process explained in detail later).

The purpose of part B was to create a completely new controller that could execute all of the instructions mentioned above. This was done by first deriving a state diagram based on the TRISC data cycle (mentioned later). After the derivation of the state diagram, the state diagram's algorithm was implemented and written in the Verilog HDL on Quartus.

Part C was putting everything together. The new controller, the design in part A and an ALU (which was designed earlier). Part C is basically the complete TRISC Computer and the main purpose of this part was to test and fully demonstrate the capability of the instructions mentioned above.

Status and unresolved issues

As writing this report part A had a minor issue regarding the hexadecimal display in regards to the accumulator. This seven segment displays a "C" on the accumulator in part A instead of incrementing the accumulator to the number 4. This issue occurs when the accumulator reaches 3 in OP CODE 0110 which is the INC instruction. Some speculation behind this leads me to believe that the counter in the accumulator has a missing or incorrect connection.

Besides that, part A runs through both INC and CLR instructions correctly. There were, however, other issues that have now been resolved.

One of the major issues dealt with in part A was that the controller and decoder were receiving mixed inputs from the instruction register (IR). The main reason behind the issue was the lack of consideration of how a series of outputs are declared in Verilog. For example, declaring "output reg S [7:0]" is different then declaring "output reg S [0:7]". When assigning bits to this so called array of output, you have to take closely in consideration which one is the most and least significant bit. The reason for this is if you have for example 8'b00000001 and 8'b100000000 the left one is actually considering an active high signal for S [7] and NOT S [0]. The problem with not taking this into consideration is when you put the control signals together to the components, the stages of the data path will not execute in order and the computer is basically useless and it seems as though everything is broken. After countless hours of frustration, I switched the bits accordingly and everything was running, however the instructions that were being executed weren't in the correct sequence.

At first it seemed as though the instructions were being skipped due to flaws in the program counter. This issue was the most complicated one in part A. Every single component had to be retested and there was still ambiguity to what was actually causing the problem. In order to solve this problem, I had to think closely of the concepts and backtrack of what would cause it. I finally got an idea of the problem, and figured that the RAM was reading the wrong address from the program counter. The address that was actually being read backwards because of the how the bus was connected and named. Switching the name instantly fixed the issue, and everything began to run smoothly after that.

For part B, there were not issues that were not resolved, and the controller was running smoothly based on the state diagram shown in the coming pages.

There were, of course, issues that occurred that have been resolved. One of the main issues in putting together this controller was the derivation process and Verilog syntax. The derivation process for a state diagram that could perform LDA, STA, JMP, and ADD became very complicated if the concepts were not clear. After clearing up the concepts a little manipulation was done to the original state diagram that was given and more states were added for obvious

purposes. A small issue in the Verilog syntax was naming the bits by themselves without the “4'b”. This caused the output LED's for the controller to crash all of a sudden. After checking the input to the controller correctly, I added the “4'b” tag and everything in the controller began to run smoothly.

In part C there were various unresolved issues as of writing this report. Putting Part C together didn't necessarily cause any compilation issues, all of the control signal seemed to have been outputting correctly. For some reason when the TRISC was attempting to add two numbers, the hexadecimal display showcased a corrupt value in the display. All the other instructions worked fine.

Report Overview

This report will overview the basic concepts behind the TRISC and its components. It will also display the diagrams used systematically and sub-systematically, along with how it was tested. Shortly in the report there will be some alternatives that were done, but most of the alternatives failed. There will be reasoning behind the design and the thinking behind how everything was implemented, and a conclusion on why that was the right thing to do.

System Design

System-level descriptions and diagrams

The TRISC system flows based on the clock input and executes its instructions based on what is contained in the RAM. These instructions are then decoded and control signals are sent accordingly through the computer. The diagram below is a high level representation of the TRISC data flow.

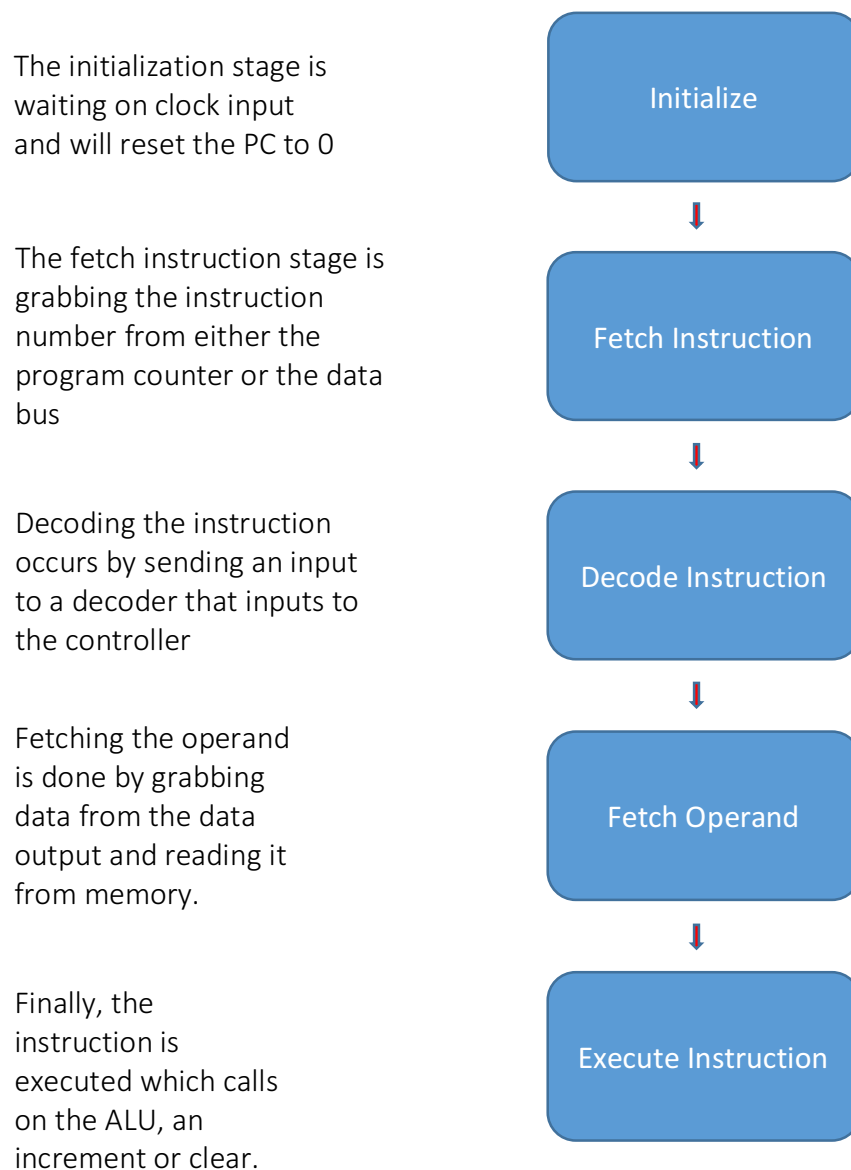
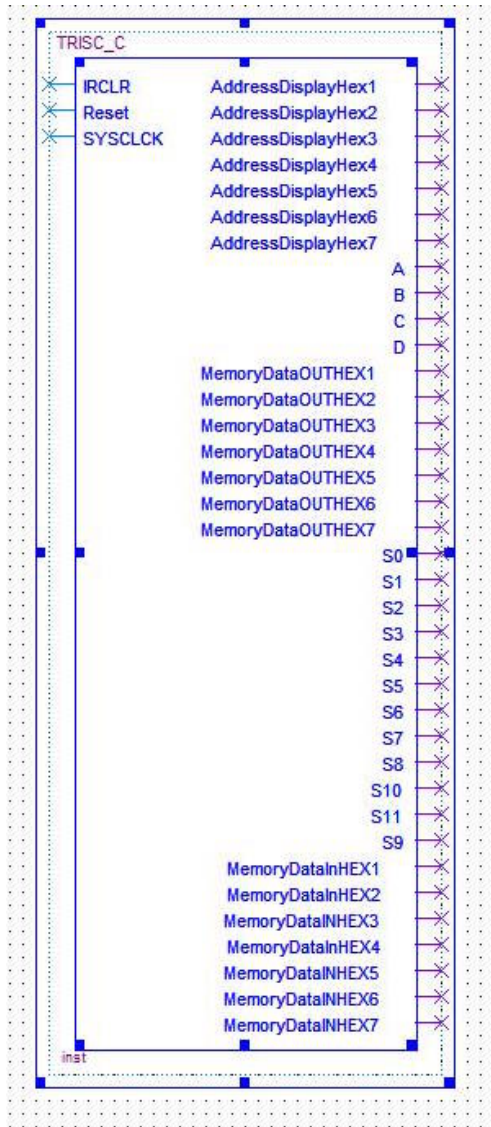


Figure 1. TRISC System Execution.

To better represent the TRISC system as a whole, and clearly see which inputs and outputs are essential to the architecture, below is a block symbol file of what the TRISC looks like.



The outputs of the BDF file are signals to a seven segment display that will display the address being processed, the memory data out and memory data in buses. The clock and reset are what triggers the reading and writing of the RAM during the data stages.

Other noteworthy outputs of the TRISC are the output signals of the controller (explained later). These are not necessary but are used to be ported into LEDs for facilitation of debugging and clarification in the demonstration process.

Figure 2. TRISC BSF File.

Subsystem Diagrams and Descriptions

The main components in the TRISC that were mentioned earlier are: Random Access Memory (16 X 8), Instruction Register, Arithmetic Logic Unit, Address Selector/MUX, Accumulator, and a Controller.

The first component/subsystem that will be analyzed is the RAM. The RAM in this TRISC is capable of 16 instructions, each being 8 bits long. It is capable of receiving a 4-bit address and inputting and outputting 8 bits. It is capable of receiving a write enable signal. The detailed design was provided and I did not derive any of it.

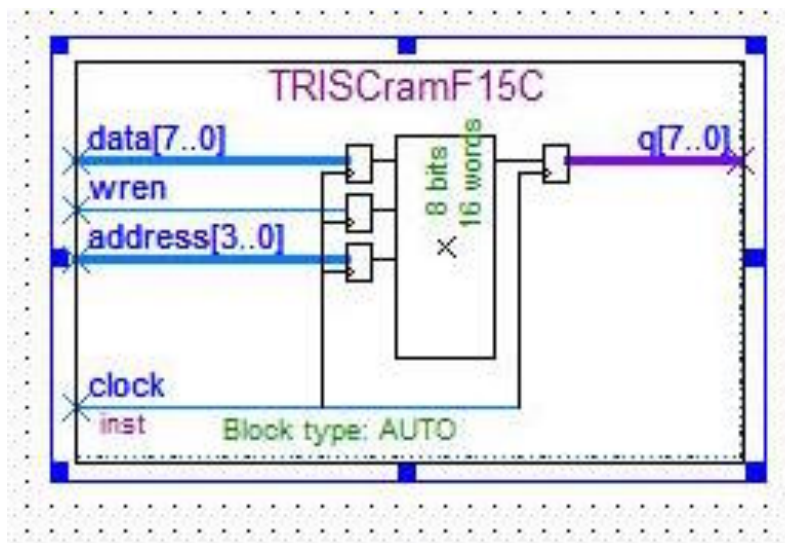


Figure 3. TRISC RAM Part C.

The next component is the instruction register. The design of the instruction register is based on D flip-flops, but is actually premade from an 74175 IC. This specific instruction register stores 4-bits based a positive edge of the clock. The main purpose of the IR is to store the opcode (operation code) obtained from the RAM and then output it on the clock signal into the decoder.

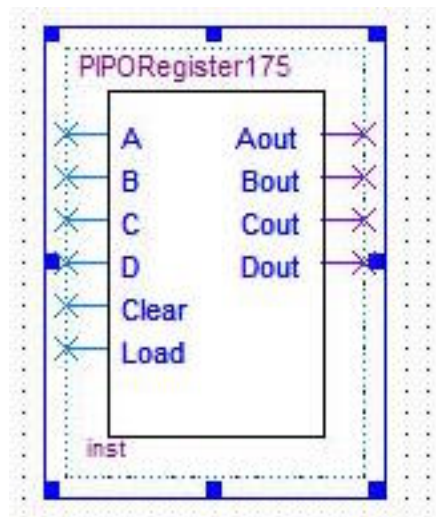


Figure 4. Instruction Register.

The ALU (Arithmetic Logic Unit) is one of the most significant parts of the TRISC; depending on the instruction set that is executed of course. The ALU takes in two 4-bit numbers and produces a 4-bit result with an overflow and carryout which is not hooked up for simplicity purposes of the TRISC, but the component itself has the capability.

The design of the ALU was designed entirely from scratch. A quick synopsis of the design process of the ALU: First a truth table was exhausted from the possibilities of adding two four bit numbers. Once the truth table was created logic Boolean equations were made, reduced using K-Maps and implemented using basic logic gates.

The capabilities of the ALU include being able to ADD, SUBTRACT, XOR and AND. This particular ALU has two selectors and each type of operation is split into each selector. The ADD and SUB is enabled by setting S0 to 1 and the ADD is enabled S1 to 0 and so on. Each arithmetic operation has a certain combination in selectors. For this TRISC system, there were no other instructions besides the ADD.

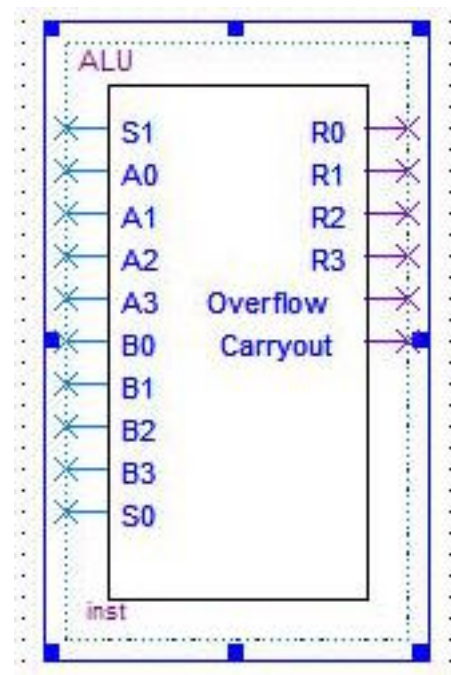


Figure 5. Arithmetic Logic Unit.

The address selector is one of the simplest components to integrate into the TRISC since it is entirely premade from a 74157 IC. It takes 4-bits from either the program counter or the memory data out bus based on a selector.

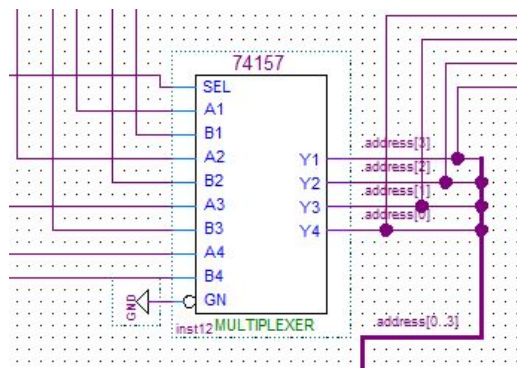


Figure 6. Address Selector.

The Accumulator (ACC) is a storage unit used to keep values from executed instructions or simply just increment or clear random numbers. This is very similar to a register, but not quite the same exact thing. The accumulator is built from a counter and a multiplexer that has the ability to switch between either inputted 4 bit values. The main purpose of having this is to switch between the ALU results or the memory data out bus.

The inputs to the accumulator can be seen to the right, and the LOAD is another important element to the accumulator. This LOAD is made into active high when added into the TRISC. There is also the ability to increment and clear the accumulator. The left and right outputs were originally for a seven segment decoder to facilitate seeing the data.

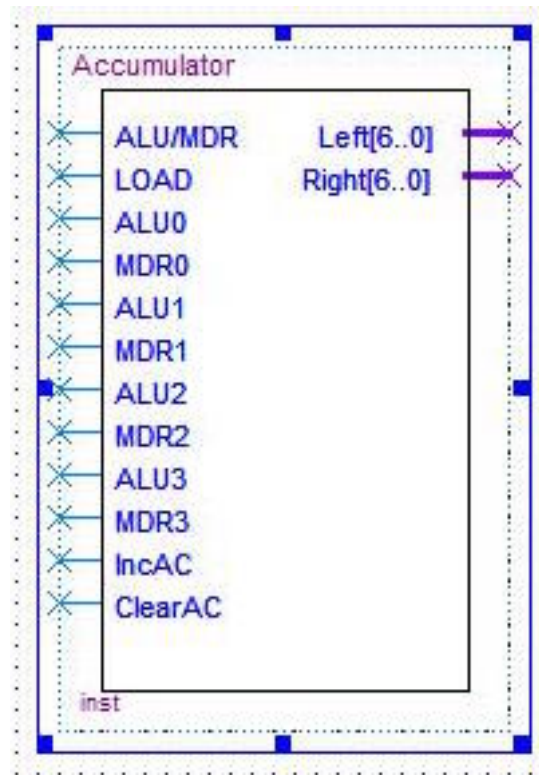


Figure 7. Accumulator.

The very last component that is missing is the controller. Since it is the most important component of the TRISC, it will be mentioned in detail below in the report, including the design and description of how the signals are outputted. Below is just the block symbol file.

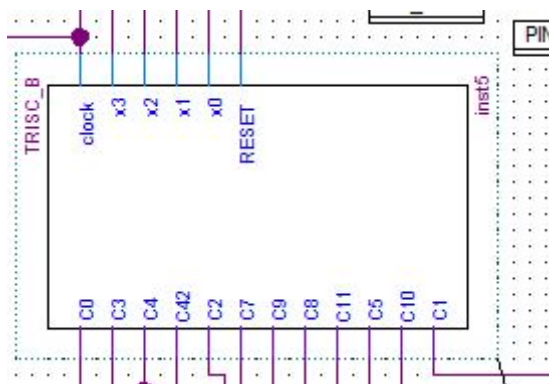


Figure 8. Full Controller.

Hierarchical Design Structure

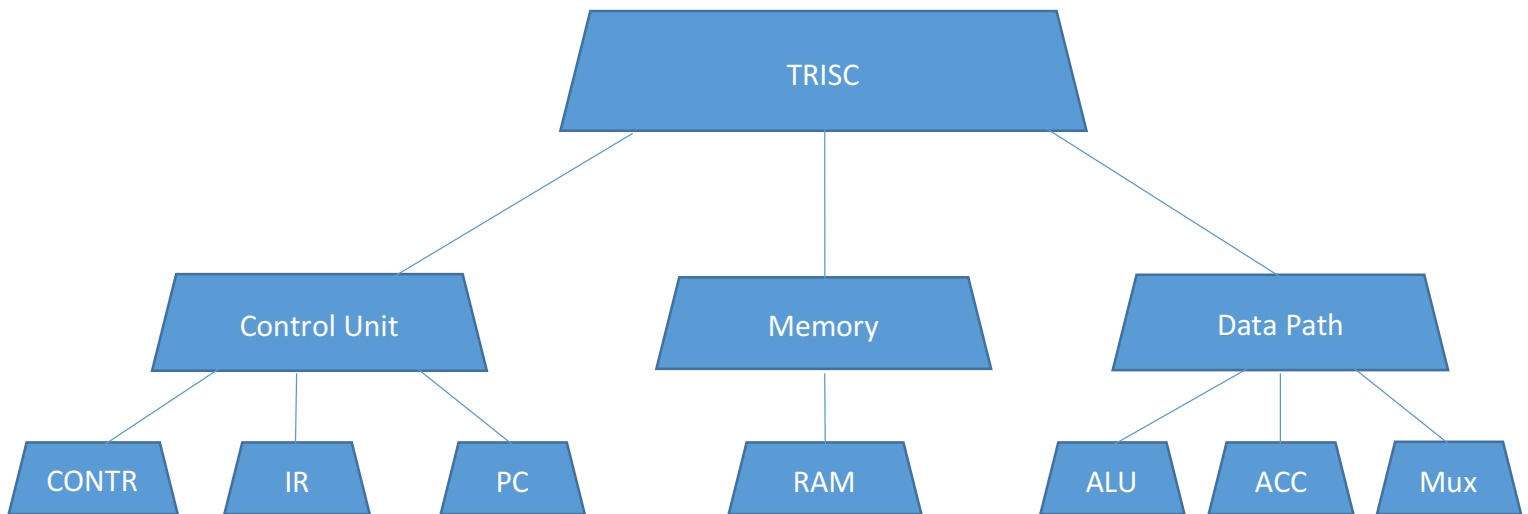


Figure 9 Hierarchical Structure TRISC.

Above is the the hierarchy of the TRISC, and it could trickle down to flip flops and logic gates, but its not necessary to go deep into logic to see how this is laid out.

Operating Procedure

The operating procedure for the TRISC is based on clock and clear inputs shown from Figure 2. The physical operating procedure is based on pin assignments done on the DE1 mentioned in the introduction. To operate the clock, a pulse of a key triggers a state and based on the state it will trigger control signals to read the RAM. To get into the more technical perspective of this works, a basic increment passing through is shown below.

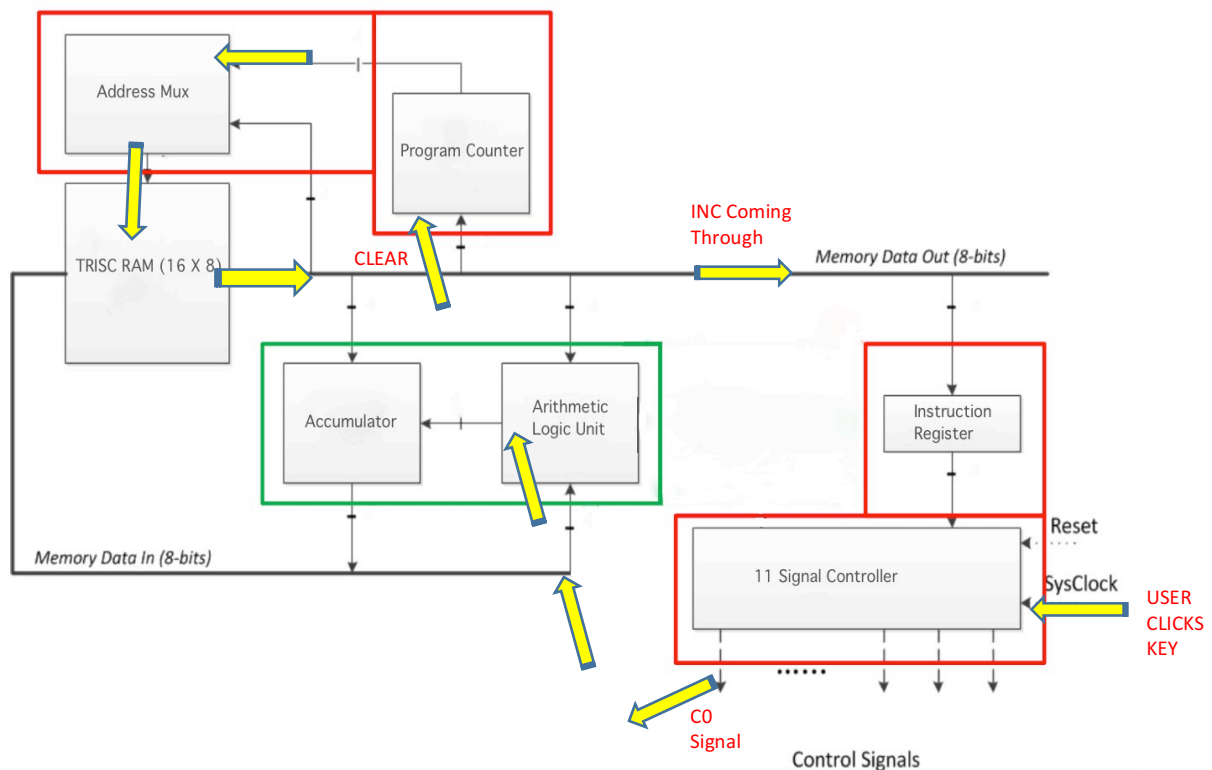


Figure 10. Operating Procedure.

Above you can see the procedure of how when the user clicks the KEY input on the DE1 it triggers all the arrows that are happening. A quick run down of the steps:

1. User clicks KEY input.
2. Controller knows to send a C0 signal which is a wire connecting to the clear of the program counter.
3. Program counter is cleared to 0, and user click KEY again.
4. Address selector is sent a C3 signal to take input from the program counter.
5. User clicks KEY twice to send the address into the RAM and read it.
6. User clicks KEY and the instruction is sent through MDA (memory data out) read by the instruction register.

Controller Design Details

Functional description showing I/O

The way the controller functions is quite simple if you understand binary numbers. It can be seen as a N to 1 multiplexer, where N is the number of control signals except the control signals are not fixed and are dependent on other variables. Inside the controller there is actually a decoder that takes 4 bits and converts them to 11 bits to be sent into signals. How this works more specifically is that the controller takes the 4-bit opcode from 0000-1111. For example, if the controller takes opcode 0110 – (6 in binary), depending on the state of the TRISC (states explained later) it will send a control signal. Control signals are dependent of the clock, input and state its in.

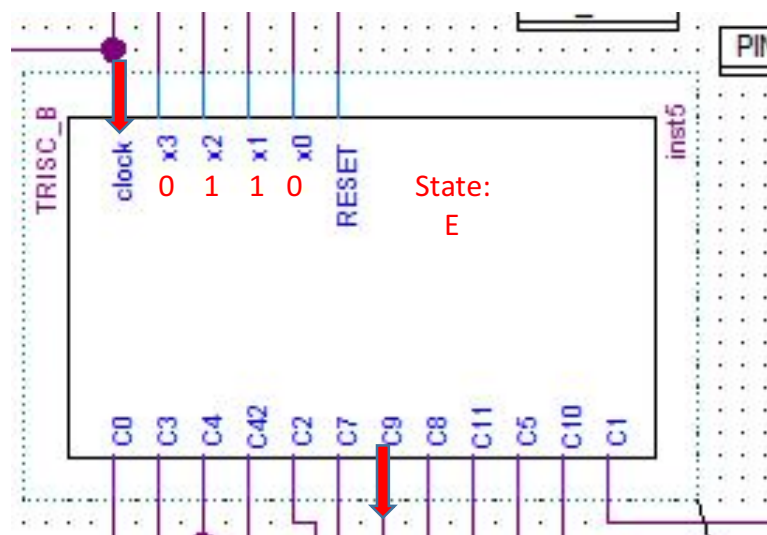


Figure 11. Controller I/O.

If the controller, for example, is in an arbitrary state E and the clock is pulsed (again the clock is user clicking the KEY button). The controller will check input, and go to a different state which will send a corresponding signal. It can be said each state does at least 1 or more control signals. In this case, the TRISC controller will signal C9 to increment the accumulator.

A quick note on another input that has not been mentioned, is the reset button. This reset button will clear the controller's state to A (first state). This is the only time the controller goes to the first state unless it is the true beginning. The first state actually resets the program counter regardless of any garbage or previous input.

State Diagrams

The controller operates on various states as mentioned before. These states progress depending on the clock and input from the instruction register (the opcode).

There are two different state diagrams that are noteworthy. One of the state diagrams is from part A mentioned at the beginning how this project was split into three different parts.

C_0 $PC \leftarrow 0$

C_3 $MAR \leftarrow (PC)$

C_4 READ memory (load address)

C_{42} READ memory (transfer data)

C_2 $PC \leftarrow (PC) + 1$

C_7 $IR \leftarrow (MDO)_{7-4}$

Decode Instruction

C_9 $AC \leftarrow (AC) + 1$

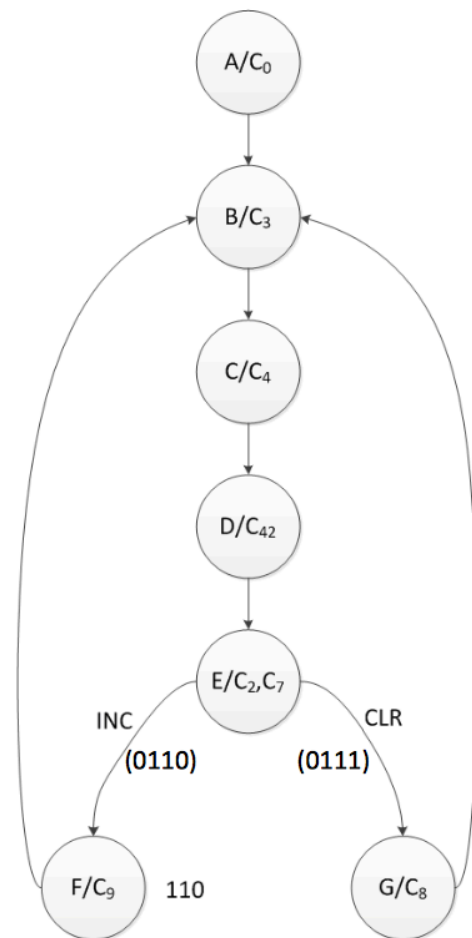


Figure 12. Part A state diagram. Dr. Carroll.

The different states produce control signals, and some states change based inputs which again is the opcode. To the left the control signals are given a small description based on what they actually do in the TRISC computer.

The state diagram for part B and C is a bit more complicated because it goes from 2 simple instructions to 6 more complicated instructions that manipulate other components in the TRISC machine.

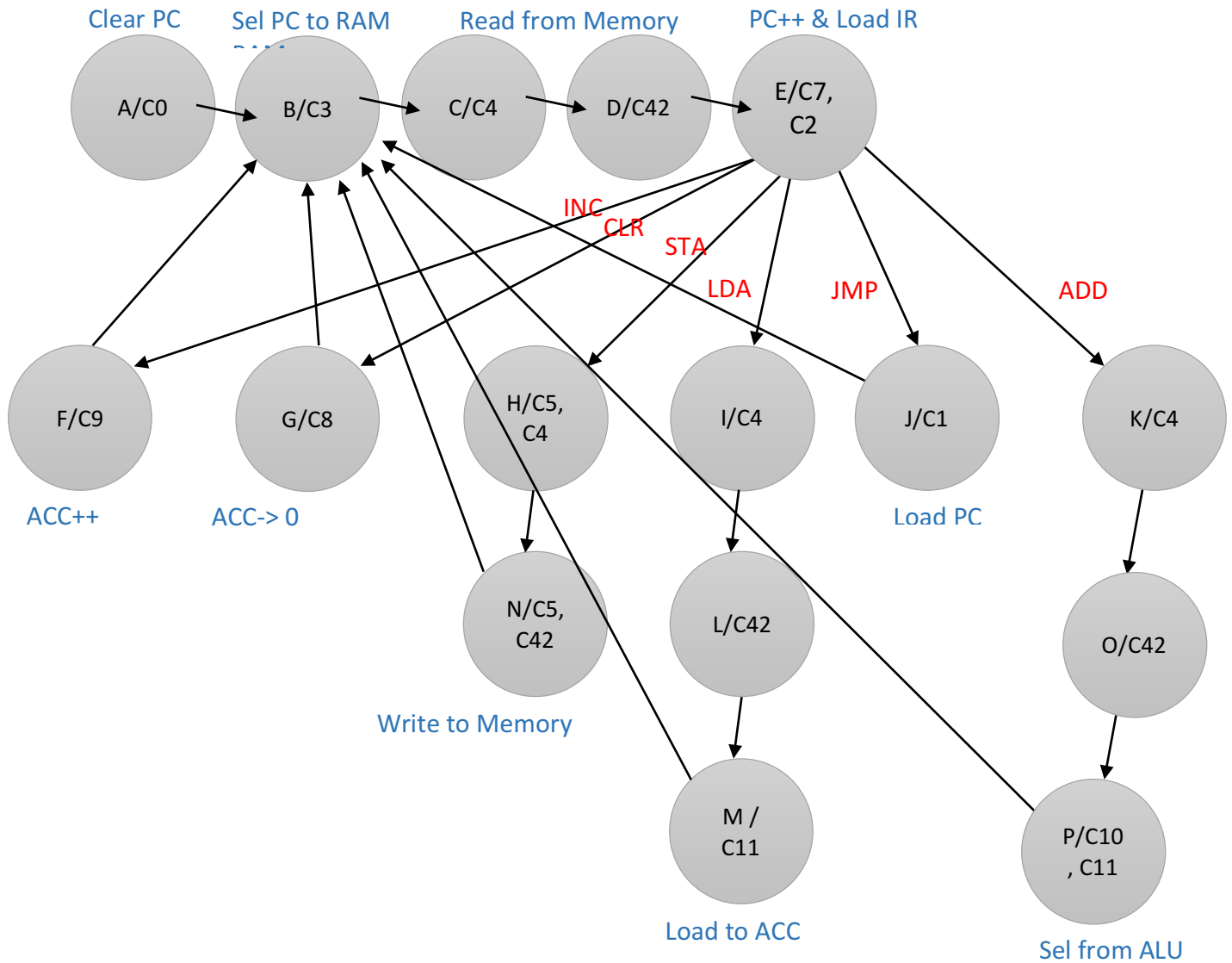


Figure 13. Part B/C State Diagram.

Controller Verilog Code

The actual implementation of the controller was done in Verilog for Part A and Part C.

Below is the Verilog code for the controller that only implements the clear and increment instructions.

```

module Controller(clock,x3,x2,x1,x0,S,RESET);
  input clock,RESET,x3,x2,x1,x0 ;
  output reg [7:0] S;
  reg[2:0]state, nextstate;
  parameter A = 3'b000, B = 3'b001, C = 3'b010, D = 3'b011, E = 3'b100,F = 3'b110, G=3'b101;
  always @(posedge clock)
    if(RESET==0) state <= A; else state <= nextstate;
  always @(state) begin
    case(state)
      A: begin nextstate <=B; S = 8'b00000001;end
      B: begin nextstate <=C; S = 8'b00000010;end
      C: begin nextstate <=D; S = 8'b00000100;end
      D: begin nextstate <=E; S = 8'b00001000;end
      E: begin S = 8'b00110000;
          if({x3,x2,x1,x0}== 4'b0111)
            nextstate <=G;

          else
            nextstate <=F;

        end

      F: begin nextstate <=B; S=8'b01000000;end
      G: begin nextstate <=B; S=8'b10000000;end
    endcase
  end
endmodule

```

Figure 14. Part A Verilog Code

The controller for part A was done inefficiently using an output array S and the bits were assigned based on the progression of each of the states. This was the cause of a lot of confusion when adding the controller to the TRISC part A. The reason for this is because it was not clear which parts of the S array were associated to the control signals. One can see where the confusion would arise when looking at the part A controller BSF shown below.

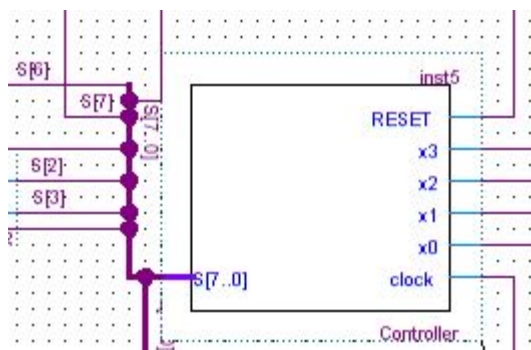


Figure 15. Part A Controller B

The controller for part B was a lot more organized, using case statements to check input in state E. The output was clearly defined by control signals and an array was not used to avoid confusion when connecting it to the components in part C.

```

1 module TRISC_B(clock,x3,x2,x1,x0,RESET,C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1);
2   input clock,RESET,x3,x2,x1,x0;
3   output reg C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1;
4   reg[3:0]state, nextstate;
5   parameter A = 5'b00000, B = 5'b00001,C = 5'b00010, D = 5'b00011, E = 4'b00100,F = 5'b00101, G=5'b00110,H= 5'b00111,I=5'b01000,J=5'b01001,L
6   always @(posedge clock) begin
7     if(RESET==0) state <= A; else state <= nextstate; end
8   always @(state,{x3,x2,x1,x0}) begin
9     case(state)
10      A: begin nextstate <=B;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b100000000000;end
11      B: begin nextstate <=C;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b010000000000;end
12      C: begin nextstate <=D;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b001000000000;end
13      D: begin nextstate <=E;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b000100000000;end
14      E: begin {C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b000011000000;
15        case({x3,x2,x1,x0})
16          4'b0000: nextstate <=I;
17          4'b0001: nextstate <=H;
18          4'b0010: nextstate <=K;
19          4'b0110: nextstate <=F;
20          4'b0111: nextstate <=G;
21          4'b1000: nextstate <=J;
22        endcase
23      F: begin nextstate <=B;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b000000100000;end
24      G: begin nextstate <=B;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b000000010000;end
25      H: begin nextstate <=N;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b001000000100;end
26      I: begin nextstate <=L;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b001000000000;end
27      J: begin nextstate <=B;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b000000000001;end
28      K: begin nextstate <=O;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b001000000000;end
29      L: begin nextstate <=M;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b000100000000;end
30      M: begin nextstate <=B;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b000000001000;end
31      N: begin nextstate <=B;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b000100000100;end
32      O: begin nextstate <=P;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b000100000000;end
33      P: begin nextstate <=B;{C0,C3,C4,C42,C2,C7,C9,C8,C11,C5,C10,C1} = 12'b000000001010;end
34    endcase
35  end

```

States K-P->>

Figure 16. Part B/C Verilog Code.

The controller moves at the positive edge of the clock and always checks for the reset button in which case it goes to state A. At state E it does a case statement to check the opcode and go to the necessary state to output the correct control signals. There were barely enough states to use 4 bits, however I added a 0 just incase I needed to make room for more later on for adjustments to make Part C function.

DE1 Pin Assignments

Through the assignment editor on Quartus, specific pins were assigned to the DE1 to display and debug each one of the controllers that was made.

To understand the pin assignments below is the DE1 with each of the switches, LED's and KEY's that been explained throughout the lab report.

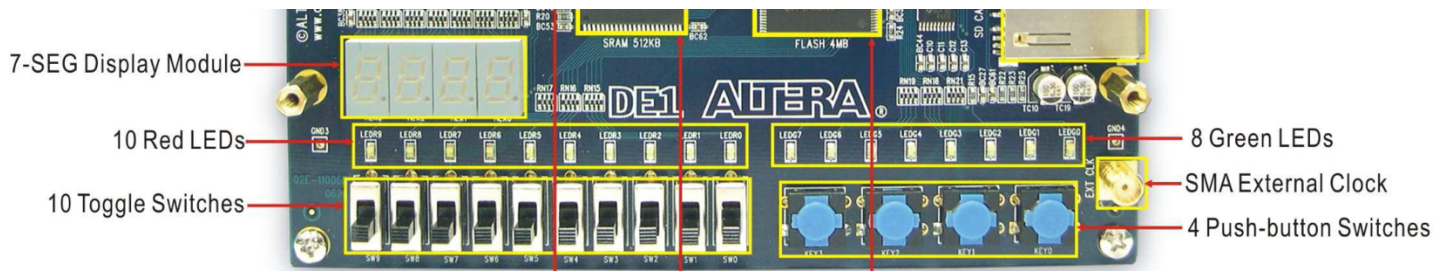


Figure 17. DE1 Inputs. *Dr. Carroll.*

The pin assignments associated with each of the components can be seen below.

Input Switches	LED Outputs	Seven-Segment
SW0 – PIN_L22	LED0 – PIN_R20	HEX0(0) – PIN_J2
SW1 – PIN_L21	LED1 – PIN_R19	HEX0(1) – PIN_J1
SW2 – PIN_M22	LED2 – PIN_U19	HEX0(2) – PIN_H2
SW3 – PIN_V12	LED3 – PIN_Y19	HEX0(3) – PIN_H1
SW4 – PIN_W12	LED4 – PIN_T18	HEX0(4) – PIN_F2
SW5 – PIN_U12	LED5 – PIN_V19	HEX0(5) – PIN_F1
SW6 – PIN_U11	LED6 – PIN_Y18	HEX0(6) – PIN_E2
SW7 – PIN_M2	LED7 – PIN_U18	
SW8 – PIN_M1	LED8 – PIN_R18	HEX1(0) – PIN_E1
SW9 – PIN_L2	LED9 – PIN_R17	HEX1(1) – PIN_H6
		HEX1(2) – PIN_H5
KEY0 – PIN_R22	LEDG0 – PIN_U22	HEX1(3) – PIN_H4
KEY1 – PIN_R21	LEDG1 – PIN_U21	HEX1(4) – PIN_G3
KEY2 – PIN_T22	LEDG2 – PIN_V22	HEX1(5) – PIN_D2
KEY3 – PIN_T21	LEDG3 – PIN_V21	HEX1(6) – PIN_D1
	LEDG4 – PIN_W22	
	LEDG5 – PIN_W21	HEX2(0) – PIN_G5
	LEDG6 – PIN_Y22	HEX2(1) – PIN_G6
	LEDG7 – PIN_Y21	HEX2(2) – PIN_C2
		HEX2(3) – PIN_C1
		HEX2(4) – PIN_E3
		HEX2(5) – PIN_E4
		HEX2(6) – PIN_D3
		HEX3(0) – PIN_F4
		HEX3(1) – PIN_D5
		HEX3(2) – PIN_D6
		HEX3(3) – PIN_J4
		HEX3(4) – PIN_L8
		HEX3(5) – PIN_F3
		HEX3(6) – PIN_D4

Table 1. DE1 I/O Pin Assignments. *Dr. Carroll.*

More specifically to the TRISC the part B controller was assigned the following outputs. To better see which outputs and inputs go to which part of the DE1, reference Table 1 above.



















To	Assignment Name	Value	Enabled
 C0	Location	PIN_R20	Yes
 C3	Location	PIN_R19	Yes
 C4	Location	PIN_U19	Yes
 C42	Location	PIN_Y19	Yes
 C2	Location	PIN_T18	Yes
 C7	Location	PIN_V19	Yes
 C9	Location	PIN_Y18	Yes
 C8	Location	PIN_U18	Yes
 C5	Location	PIN_R17	Yes
 C10	Location	PIN_U22	Yes
 C1	Location	PIN_U21	Yes
 x0	Location	PIN_L22	Yes
 x1	Location	PIN_L21	Yes
 x2	Location	PIN_M22	Yes
 x3	Location	PIN_V12	Yes
 RESET	Location	PIN_R21	Yes
 clock	Location	PIN_R22	Yes
 C11	Location	PIN_R18	Yes

Table 2. Part B Controller Pin Assignments

The controller for part A was given very similar assignments.














 x0	Location	PIN_V12	Yes
 x1	Location	PIN_M22	Yes
 x2	Location	PIN_L21	Yes
 x3	Location	PIN_L22	Yes
 S[0]	Location	PIN_R20	Yes
 S[1]	Location	PIN_R19	Yes
 S[2]	Location	PIN_U19	Yes
 S[3]	Location	PIN_Y19	Yes
 S[4]	Location	PIN_T18	Yes
 S[5]	Location	PIN_V19	Yes
 S[6]	Location	PIN_Y18	Yes
 S[7]	Location	PIN_U18	Yes
 clock	Location	PIN_R22	Yes

Table 3. Part A Controller Assignments.

The most important pin assignments were for the full TRISC with everything put together. The HEX values are the Memory Data Out, Address in RAM and Memory Data in. Each control signal had an LED assignment.

out	hex11	Location	PIN_J1	Yes
out	hex12	Location	PIN_H2	Yes
out	hex13	Location	PIN_H1	Yes
out	hex14	Location	PIN_F2	Yes
out	hex15	Location	PIN_F1	Yes
out	hex16	Location	PIN_E2	Yes
out	hex20	Location	PIN_E1	Yes
out	hex21	Location	PIN_H6	Yes
out	hex22	Location	PIN_H5	Yes
out	hex23	Location	PIN_H4	Yes
out	hex24	Location	PIN_G3	Yes
out	hex25	Location	PIN_D2	Yes
out	hex26	Location	PIN_D1	Yes
out	hex30	Location	PIN_G5	Yes
out	hex31	Location	PIN_G6	Yes
out	hex32	Location	PIN_C2	Yes
out	hex33	Location	PIN_C1	Yes
out	hex34	Location	PIN_E3	Yes
out	hex35	Location	PIN_E4	Yes
out	hex36	Location	PIN_D3	Yes
out	S0	Location	PIN_R20	Yes
in	IRCLR	Location	PIN_T22	Yes
in	Reset	Location	PIN_R21	Yes
out	hex10	Location	PIN_J2	Yes
out	S1	Location	PIN_R19	Yes
out	S2	Location	PIN_U19	Yes
out	S3	Location	PIN_Y19	Yes
out	S4	Location	PIN_T18	Yes
out	S5	Location	PIN_V19	Yes
out	S6	Location	PIN_Y18	Yes
out	S7	Location	PIN_U18	Yes
out	S8	Location	PIN_R18	Yes
out	S9	Location	PIN_R17	Yes
out	S10	Location	PIN_U22	Yes
in	SYSCLK	Location	PIN_R22	Yes
out	S11	Location	PIN_U21	Yes

Table 4. TRISC DE1 Pin Assignments.

Alternative Design Considerations

Alternatives Considered

For this TRISC project there were very few considerations for an alternative design, except for a few things at the beginning that were used and neglected in the final design. For example, the original layout of all of the components along with the buses was not followed at the beginning. I tried putting the program counter right next to the accumulator and the selector along with it. It was not the right way of going about it because even though some wires were accessed easier, the connected to the address selector and RAM was way too far and possibly made everything more confusing.

Another alternative that was considered was to make the counter in Verilog. The reason this was considered was because in Part A the seven segment display was causing errors in the accumulator when going from 3 to 4 it had gone to C. Due to time constraints, the design of the new counter was never implemented and the time was allocated to get other more important things working.

A smaller thing that was considered was to change everything to consistently be active high, so there wouldn't be any small errors due to logic. Again, due to time constraints the plan was never fully executed.

Reasons for selection of final design

The final design chosen for part C is shown in figure 17. The reason I choose this design is because it had most of the components working only the ADD instruction wasn't working. When I tried other implementations of the design some of the other instructions weren't working. The design at the end was mostly to get everything functioning rather than having efficient logic.

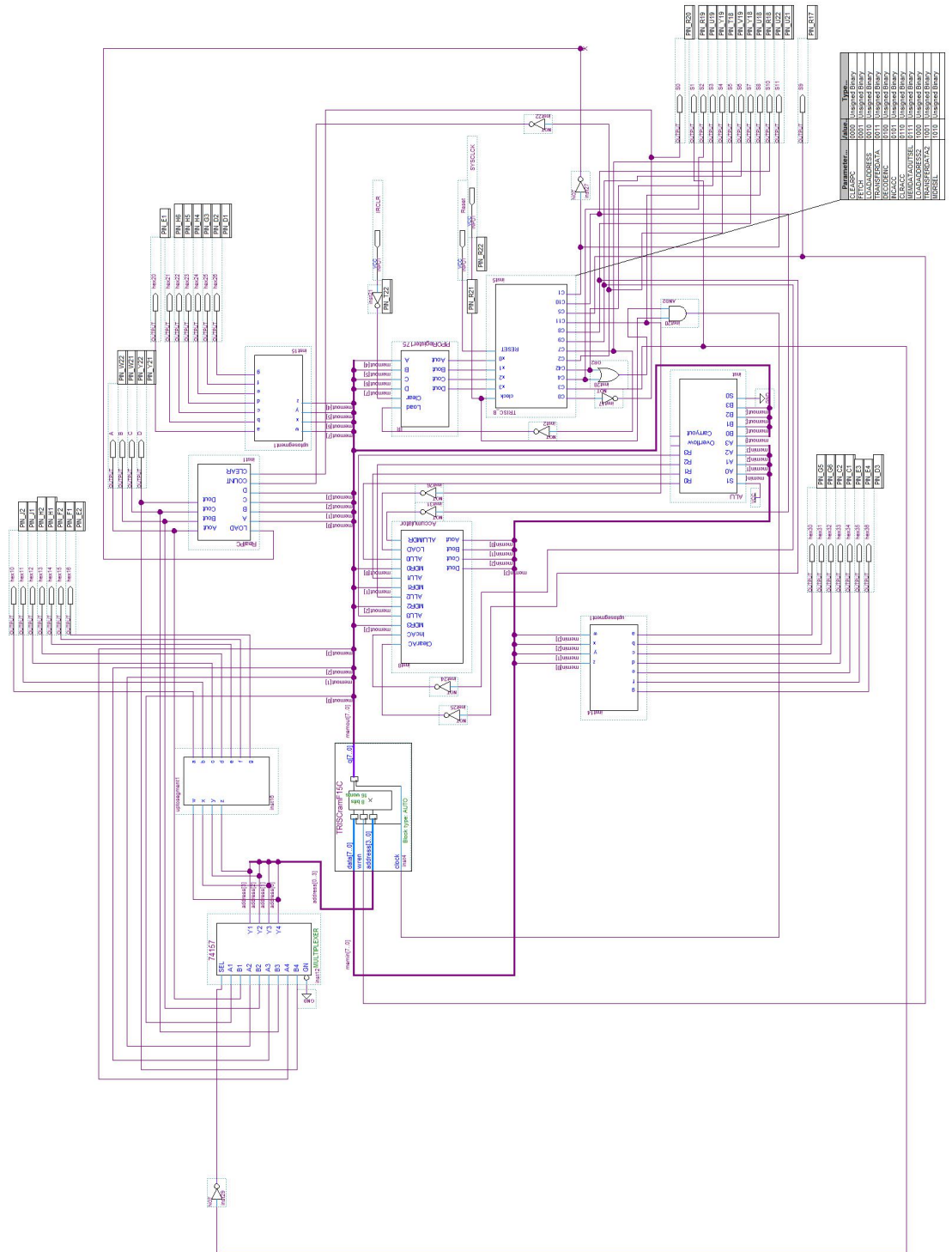


Figure 17. Full TRISC Design.

Integration and Test Plan

Integration Strategy

In order to integrate all the components efficiently to avoid errors, each component was compiled after being added to the bus. The very first component that was added was the RAM that was given. From there the Memory Data Out, Memory Data IN and address buses were extended and then the address selector and so on.

The very last thing that was added to the design was the controller. The reason for doing this was because if you added the controller in one of the middle stages the signals would be more difficult to plan if you don't have an idea of where all the components are.

Another essential part of integrating everything was checking the active low and high inputs. This could mess with the whole functioning system, so checking inside each of the individual block description files was important.

Understanding the data flow and how everything works before integrating it, is obviously, the most important thing in the process. If you attempt to debug something you put together without knowing how it works, it will take a long time.

Something that wasn't taken into consideration in older designs, such as the ALU, was that wires can easily collide if you had two components connected too close to each other, you run out of room to have more wires. Not only does the wiring get clustered and confusing, but the names start to fade in the wiring.

Test Strategy

To test the design everything was obviously burned on cyclone mentioned earlier and the user key input was the main way to see how all the signals are working based on the LED's

The strategy for testing is to look closely at any bug that might happen and observe any patterns. For example, in one of my test cases in Part A it seemed as though everything was working smoothly except for the skipping of one instruction. It wasn't however skipping an instruction, and the reason I saw this was looking at the pattern. The instructions were executing backwards from what the HEX file had in the RAM. After this sequence was recognized, it was a matter of looking at the input of the address in the RAM.

Another strategy of testing is to output everything possible. Whether it be in the seven segment displays or in the LED's. Knowing how everything is flowing as a whole is the first task when looking to find the problem.

Once you have everything working, it is also a good idea to do some exhaustive testing such as going continuously through the sequence for a few minutes, and identify any bugs. This is how I identified the issue of the accumulator having the wrong value of “C” instead of “3”.

Simulation Results from Quartus

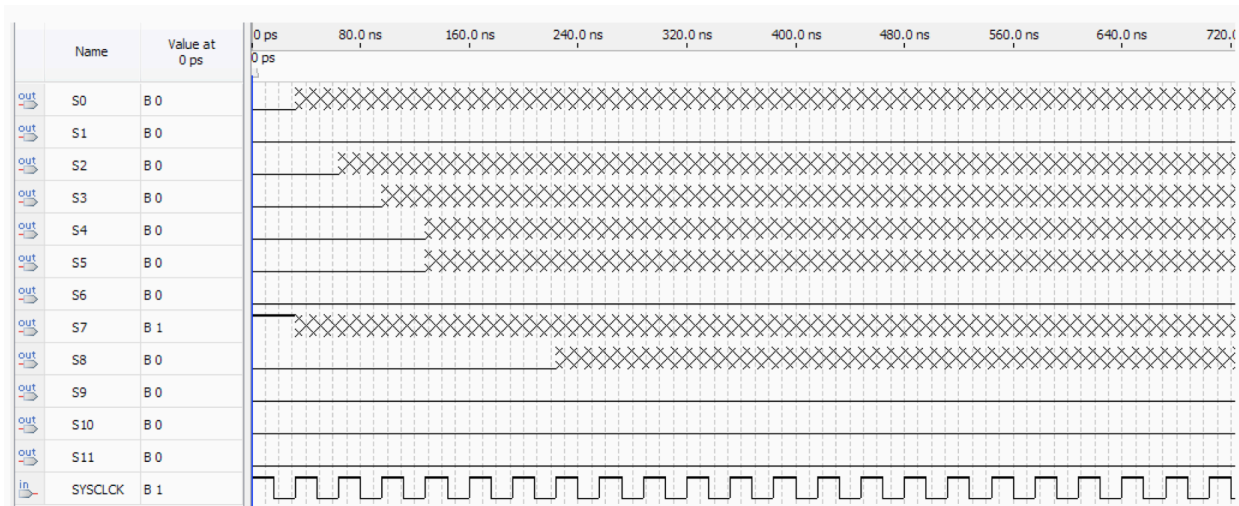


Figure 18. Simulation Results.

The results from Quartus are a bit shaky and the C3 signal was being called along with the other signals all the time. After making this simulations, it can be said that the reason it wasn't adding was due to faulty signals or possibly there are not enough clock signals to fully simulate the full data cycle.

Test Results from the DE1

For part all of the instructions executed perfectly after other testing and design changes. The sequence of instructions was the following:

CLR, INC, CLR, INC, INC, CLR, INC, INC, INC, CLR, CLR, CLR, INC, INC, INC, INC, CLR.

This sequence matched exactly what was on the .HEX file in the RAM.

The part B test results were based on simply seeing the LED's which were the controller signals. That ran smoothly from LEDR0 to some of the used LEDG's.

The main testing errors occurred in part C. The opcodes were being read correctly, and most of the instructions were executing correctly, however the accumulator display output was not working for the ADD and STA.

Conclusion

Resolution of Design/Implementation Issues

The final design can be seen in figure 17, most of the implementation issues have been covered in the previous parts of the report, but to some it up Parts A and B were working fine. However, the full TRISC machine did not execute all the necessary instructions in the sequence. The memory was read fine by the instruction registers, but the control signals were most likely not appropriate to those instructions

Lessons Learned

One of the major lessons learned about the TRISC project was that the divide and conquer method is really efficient when trying to do a huge project. Although it wasn't obvious while we were designing the ALU and other components earlier. Doing one component at a time and testing it to make it work is the best way to design a CPU such as this one.

Another thing that was learned is that even though you may have all the components ready, putting them together is not as obvious as it seems. Consistency is key when building multiple components to a huge system. What is meant by consistency is in the case of active low or active high signals, you must make sure each component will be compatible and it is convenient to make those type of signals consistently one.

The most important lesson and experience to gather from the TRISC project is the design cycle that is done when designing combinational and sequential logic circuits. From putting together, the basic logic gates to using Quartus. Every step in this project had a meaningful purpose that will allow me to not commit the same mistakes, and expand on the successful implementations that did go right with this project.