Fall 2014 Project – Tiny Reduced Instruction Set Computer (TRISC)

Prepared for: Dr. Bill Carroll professor CSE 2441

Prepared by: Scott Nidell student CSE 2441

Date: 12/3/2014

Table of Contents

Introduction. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

System Design. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

Controller Design. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

Alternative Design Considerations. . . . . . . . . . . . . . . . . . . . . . . . 33

Integration and Test Plan. . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

Conclusion. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

Appendices. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

**Introduction (1)**

The Accreditation Board of Engineering and Technology (ABET) is an organization that provides accreditation to University Engineering programs. The Tiny Reduced Instruction Set Computer (TRISC) is a project that helps support that ABET accreditation. The TRISC is a small adaptation of every day computer at its most basic level. This TRISC will essentially interface an assembly programmer to the digital logic level of the computer architecture. Given an operational code the TRISC will execute various commands detailed later in this report.

**-Project Overview (1.1)**

The TRISC is to be a control unit that interfaces with an assembly programmer. However, instead of using a terminal program instructions are read and written to memory provided by the professor. The TRISC will execute various commands (Table 1) that are detailed in the table below:

**Table 1:** *Table of Instructions for TRISC project*

|  |  |  |  |
| --- | --- | --- | --- |
| *Instruction* | *Function* | *Register Transfer* | *Op Code* |
| *LDA* | *Load Accumulator* | *ACC<-(MDR)* | *0000* |
| *STA* | *Store Accumulator* | *MDR<-(ACC)* | *0001* |
| *ADD* | *Add Accumulator* | *ACC<-(ACC)+(MDR)* | *0010* |
| *INC* | *Increment Acc.* | *ACC<-(ACC)+1* | *0110* |
| *CLR* | *Clear Accumulator* | *ACC<-0* | *0111* |

The Accumulator, MDR and others are parts of the TRISC subsystem that will be detailed in (2.2). An Operational Code or Op Code is given to the TRISC and executes this function.

Load: This function will load the contents of the accumulator into temporary memory

Store: This function will store data in the accumulator at a given memory address.

Add: This function will add a given binary value to the value currently store in the accumulator

Increment: This function will increment the accumulator by 1.

Clear: This function will clear whatever contents in the accumulator to zero.

**-Project Status (1.2)**

Currently the project is completed up to Section A the table below (Table 2) provides dates of completion.

**Table 2:** *Current status of project parts.*

|  |  |  |
| --- | --- | --- |
| Section | Completed? | Outstanding |
| Part A | Completed | Completed Nov. 24th |
| Part B | In process | Addition function |
| Part C | In queue | Expected December 2nd |

**-Report Overview (1.3)**

The project report is a central location of all project data. This includes but is not limited to Design, Implementation, Explanation, Test, Results and Outcomes, and Current Project Status. This report will also act as an official correspondence to the professor of project completion and any other reader of interest to the on goings of the final project of the CSE 2441 at the University of Texas at Arlington.

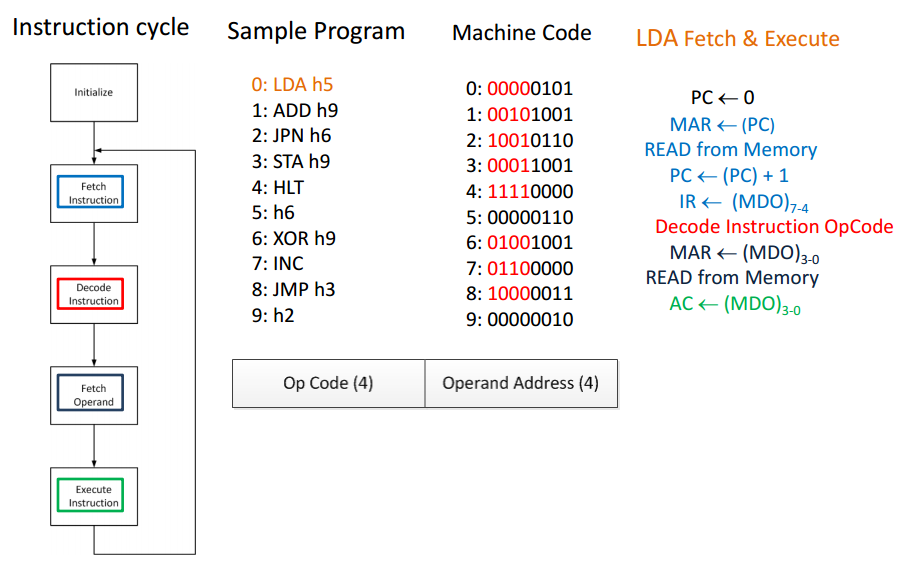
**System Design (2)**

With the direction of the professor and teaching assistants the TRISC was design over the course of the semester by supplement of labs and project time allotment. The system is designed using multiple modules and learning activities throughout the semester these include: Counters, Registers, Decoders, Adders, Flip Flops and basic logic gates. These modules will be described in section 2.2

**-System Level Description and Diagrams (2.1)**

The TRISC is designed to take op codes in form of a binary combination. The program counter points to the first instruction in memory. This instruction is then read from the memory and loaded into the Instruction Register. After the Instruction Register outputs its Op code to the Op Code Decoder which will send a control signal to the TRISC controller. The controller will then use this control signal to make decisions on how to execute the instruction. A flow diagram (Figure 1) is provided below to show the control lines between modules

**Figure 1:** *Flow diagram for the TRISC system*

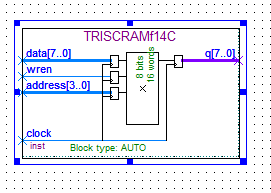
**

As shown above each instruction goes through a Fetch and execute cycle. Each instruction is first fetched from the memory and then executed given the instruction parameters. The Machine code represents 8 bits the first four are the Op code and the second the operand. Some codes like Add, use both the op code and operand.

**-Subsystem Descriptions and Diagrams (2.2)**

TRISC is built with multiple subsystems and modules. These modules are described here to clarify future and previous explanations of the system. The system uses multiple **buses**. These buses carry multiple control wires in a bundle and are denoted by two brackets: data[7..0]. This syntax displays that the bus named data holds 8 wires in the bus bundle. To access a single wire on the bus an explicit name is required to describe what wire is being uses: data[7]. This syntax will use the seventh wire of the eight wire data bus.

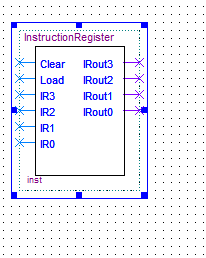
**Figure 2:** *TRISC RAM (Random Access Memory)*



The *TRISC RAM* is the central storage area for the Op codes and operands through all operations of the computer. Op codes are read from the memory from the Q bus. The first four wires of the Q bus are for the op code the remaining four are used for operands out to the rest of the TRISC. The data bus is used to read operands into memory and display what is currently in the accumulator. The WREN input is a write enable pin. This prevents writing to the memory inadvertently. The address bus is used to select what memory address an operation takes place on.

The *Instruction Register* (Figure 3) holds the op code to provide this code to the controller throughout each instructions execution.

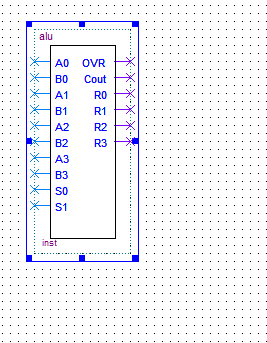
**Figure 3:** *Instruction Register*



The instruction register has multiple pins. Clear will set the output on all pins to zero. Load will load the data currently being held on IR0-IR3 to update on IRout0-IRout3. These are the inputs and outputs of the register.

The *Arithmetic Logic Unit (ALU Figure 4)* is the unit of all math expressions available to the TRISC. These include ADD, Subtract, XOR, and AND. However for this project add will be the only math operation used.

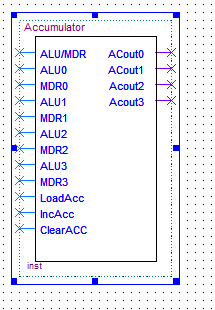
**Figure 4:** *Arithmetic Logic Unit*

**

The ALU has multiple pins. S1 and S0 control which operation will be selected A0-A3 and B0-B3 are the two operands being considered the OVR pin and Cout are the over flow and carryout that follow the add and subtract operations. R0-R3 are the pins that the result of the arithmetic are displayed.

The *Accumulator* (Figure 5) is a special register that hold information on current operations. It helps read and write from memory along with incrementing and clear data being written into memory.

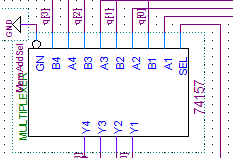
**Figure 5:** *Accumulator*



There are multiple pins on the accumulator. ALU/MDR is a selection switch to choose to read from the ALU or the outgoing memory on the TRISCRAM. ALU0-ALU3 and MDR0-MDR3 are the pins being selected. LoadAcc loads the selected pins to the output pins ACout0-Acout3. IncAcc will increment the current value on the output pins by one. ClearAcc will set all output pins to zero.

The *Memory Address Register (MAR)* (Figure 6) is a basic multiplexor that selects which address to read given a control signal between the program counter and the memory address output.

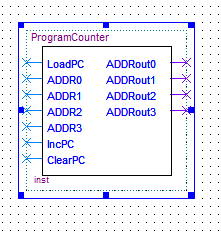
**Figure 6:** *Memory Address Register*

**

The few pins on the MAR will provide a selection channel to redirect which address will be selected. SEL will select either A0-A3 or B0-B3 to the outputs Y0-Y3.

The *Program Counter* (Figure 7) is a simple binary counter and provides the location of the current address in memory being considered. It also helps with the jump operation.

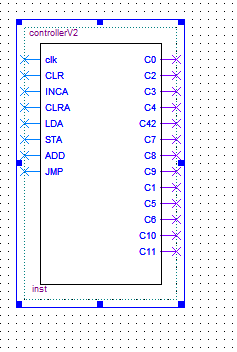
**Figure 7:** *Program Counter*

**

The program counter has multiple pins. LoadPC will load the current address ADDR0-3 to the output ADDRout0-3. The IncPC will increase the output by one and the ClearPC pin will set the current output of the Program Counter to zero.

The *Controller (Figure 8)* is the center of the TRISC. All control logic is derived from this unit which consists of a finite state machine.

**Figure 8:** *TRISC Controller*

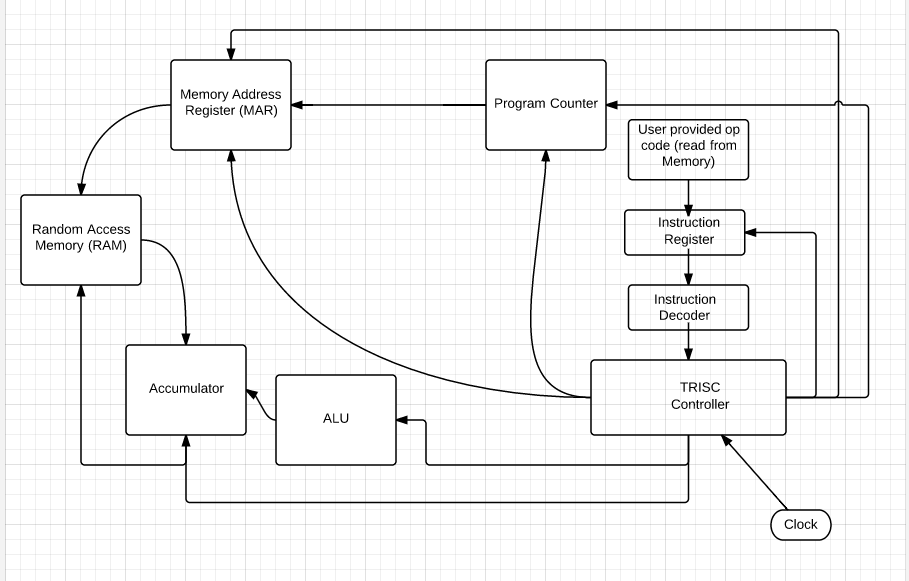
**

The controller has the most amount of pins as it ties all the modules in the TRISC together. The Clk is the system clock this would general come from the processor in most computers. This clock helps move the finite state machine progress through each state. CLR clears the controller and sets the controller to state zero. INCA, CLRA, LDA, STA, ADD, and JMP are all control signals that are taken from the Op code decoder. This will help the controller with branching decisions in the states progression. C0, C2, C3, C4 C42, C7, C8, C9, C1, C5, C6, C10, and C11 are all integrated into each sub system throughout the design. One or more of these control signals are turned on depending on what state the controller is currently in.

**-Hierarchical Design Structure (2.3)**

The TRISC project is centered on the control unit (Figure 9). Each subsystem described above is connected to it in some form. The control unit provides timed controls to each subsystem given an op code parameter from the instructions register.

**Figure 9:** *Hierarchical Design Structure of the TRISC project*

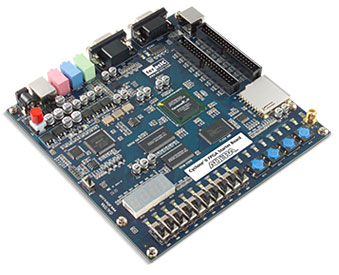


The design is almost completely encapsulated besides the clock. This is provided by the user or a system clock. This clock partially determines how fast the instructions are carried out.

**-Operating Procedure (2.4)**

The operation of the TRISC from a user standpoint is relatively simple. The user has three buttons on the Cyclone II (Figure 10) to use. Key 0 the clear button which resets the controller to the ‘A’ state (described in section 3.2) and Key 1 the clock signal and also the power switch to turn on the unit. This key will step through the states and branch. Each state is determined on the clock and the input signal provided by the op code. Each state will produce one or more outputs either being active low or active high in nature. These outputs are sent to the Green LEDs on the DE1 unit. Along with the control signals 3 seven-segment displays are used to provide the current address being analyzed, the current Op code being executed, and the value in the accumulator.

**Figure 10:** *Cyclone II DE1 layout*



Power Button

Seven-Segment Displays

Key0, Key1

Control Lights

**Controller Design Details (3)**

The following section will describe the controllers for both part ‘A’ and part ‘B’. The controller for part ‘A’ was a derivation of Lab 9. This controller had two main functions. The first, to clear the accumulator, and second increment the accumulator. The second controller for part B added multiple functions including Jump, Store, Load and add.

**-Functional Description and Diagram showing I/0 (3.1)**

The controller for part ‘A’ develops a basic fetch and execute code instruction. This will fetch an op code from the memory, load it into the Instruction Register and execute that operation. The controller will make branching decisions based on the op code input. For part ‘A’ these decisions are either increment the accumulator or clear the accumulator. The fetch and decode operations take up the majority of the control signals in part ‘A’ (Figure 11).

-Control signal C0 will clear the output pins to zero and is attached to the clear pin for the Program Counter

-Control signal C3 will set the Memory Address register to read from the Program Counter

-Control signal C4 will send the first pulse to the TRISC memory for reading

-Control signal C42 will send the second pulse to the TRISC memory to transfer data from memory to the data output bus of the memory.

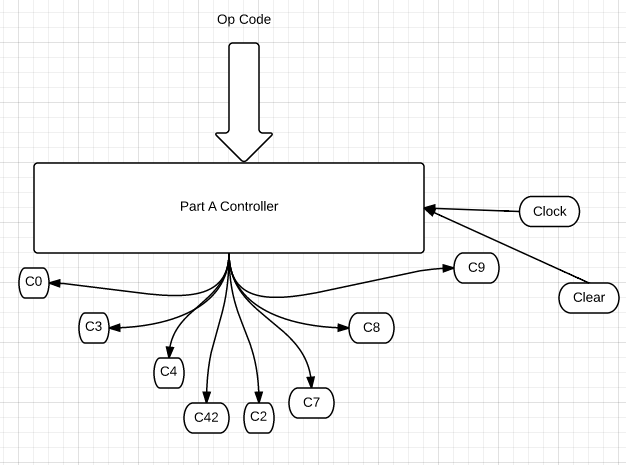
-Control signal C2 will increment the program counter to point to the next address in memory

-Control signal C7 Loads the Op code that was previously transfer to the Instruction Register.

-Control Signal C8 is connected to the accumulator clear pin. This will clear the output values of the accumulator to zero

-Control Signal C9 is connected to the increment pin of the accumulator this will add a one to the output pins of the accumulator.

**Figure 11:** *Diagram I/O for section ‘A’*

**

The controller for part ‘B’ (Figure 12) adds functions for Jump, Load, Store and add. This also effectively changes the I/O by adding multiple control lines to the controller.

-Control Signal C1 attaches to the Program Counter’s load pin. This aids the jump operation by sending the current address in memory to the program counter which is then loaded into the Memory address register.

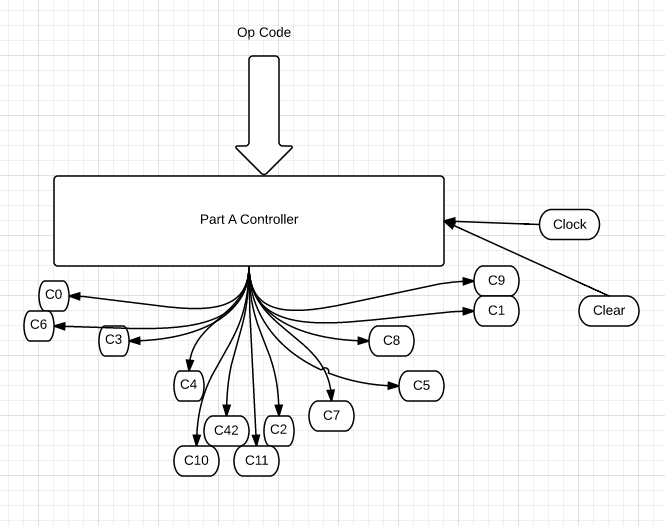
-Control Signal C5 is the write enable control line. This control line is essential in being able to write to the memory along with the memory read function (C4&C42) to write to the current memory address.

-Control Signal C6 clears the instruction register to output pins to zero.

-Control Signal C10 is attached to the accumulator select pin. This allows the accumulator to select between the address currently in operation and the ALU. This aids in the add operation of the TRISC

-Control Signal C11 is connected to the accumulator load pin. This helps with the indirect load or store function and load function. This will set the output pins to whatever selected input pins are selected on the accumulator.

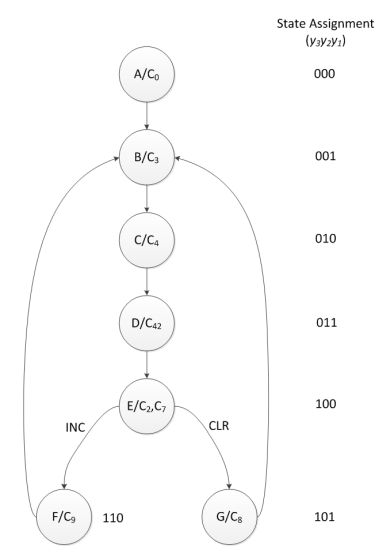
**Figure 12:** *Diagram I/0 for section ‘B’ controller*

**

**-State Diagrams (3.2)**

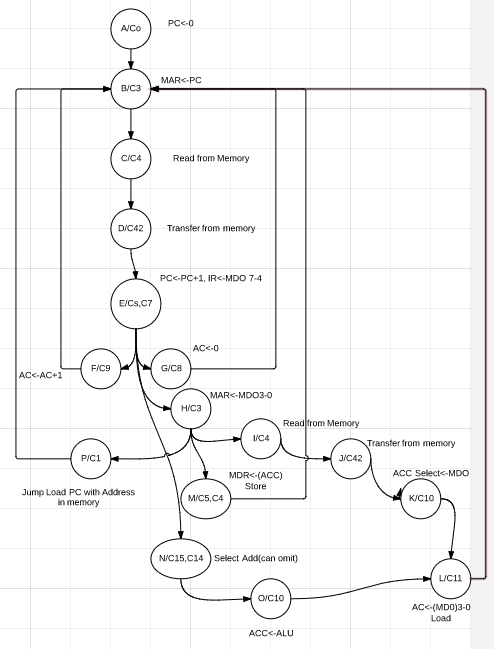
The state diagram for the part ‘A’ controller (Figure 13) was provided to us from lecture instructions and the Lab 9 write up. It shows the fetch and decode cycles along with the increment and clear accumulator states.

**Figure 13:** *State diagram for part ‘A’ controller*

**

The state diagram for part ‘B’ controller (Figure 14) had to be designed by the student. This diagram includes the added functions of Jump, Store, Load and Add.

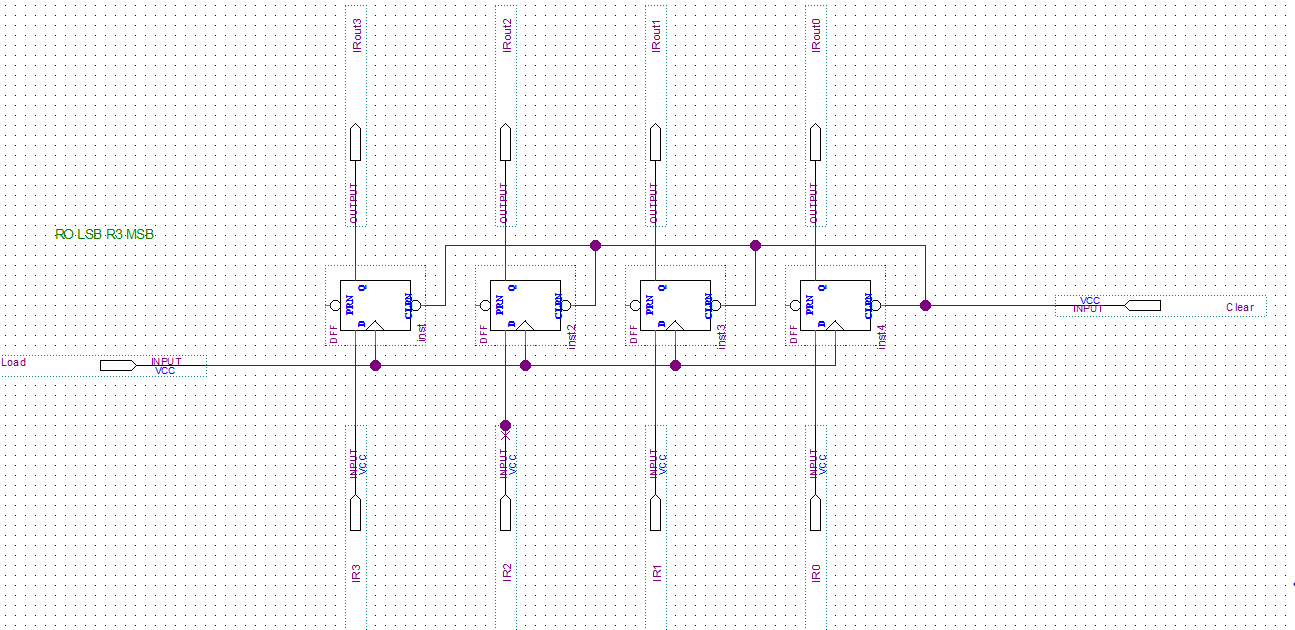
**Figure 14:** *State Diagram for part ‘B’ controller*

**

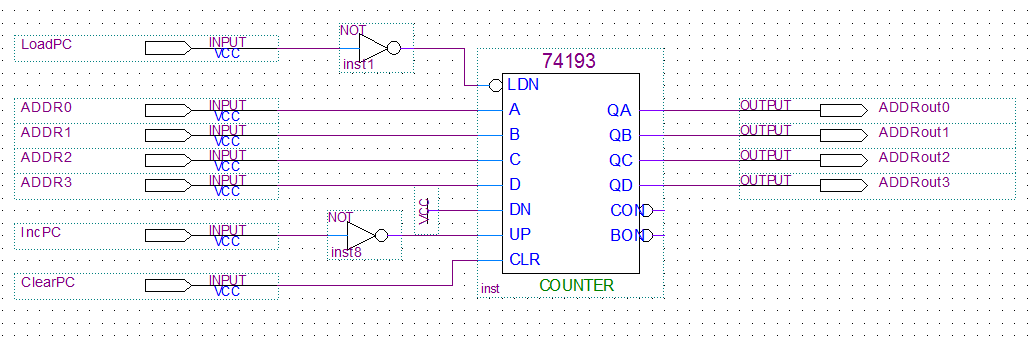
**-Schematic Diagrams and/or Verilog Code (3.2)**

This section details the electrical schematics and Verilog code for all subsystems

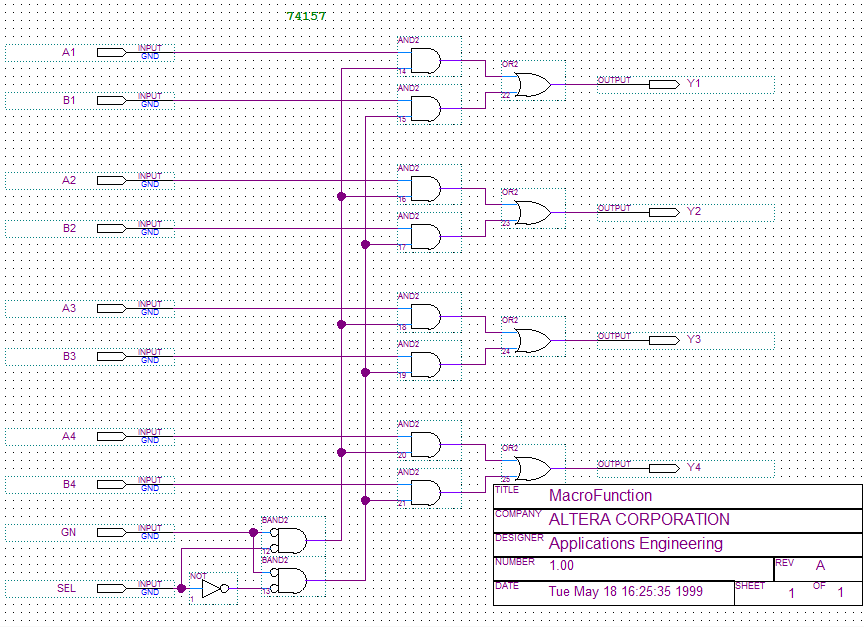
**Figure 15:***Schematic for the Instruction Register*



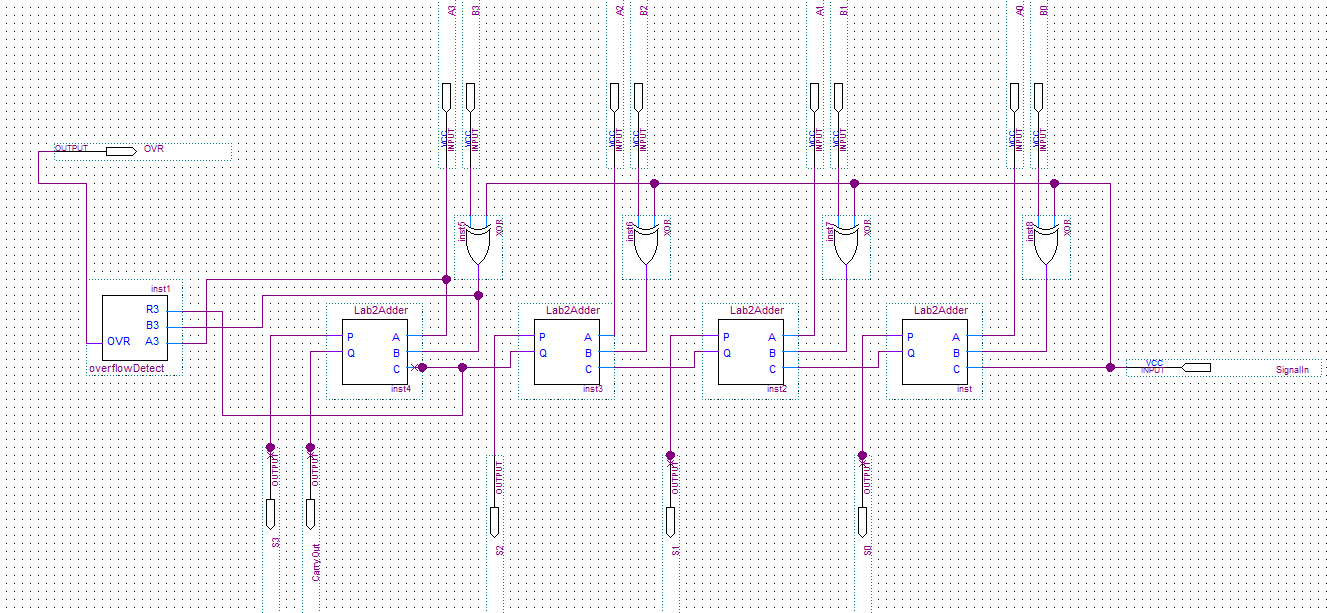
**Figure 16:** *Schematic for the Program Counter*

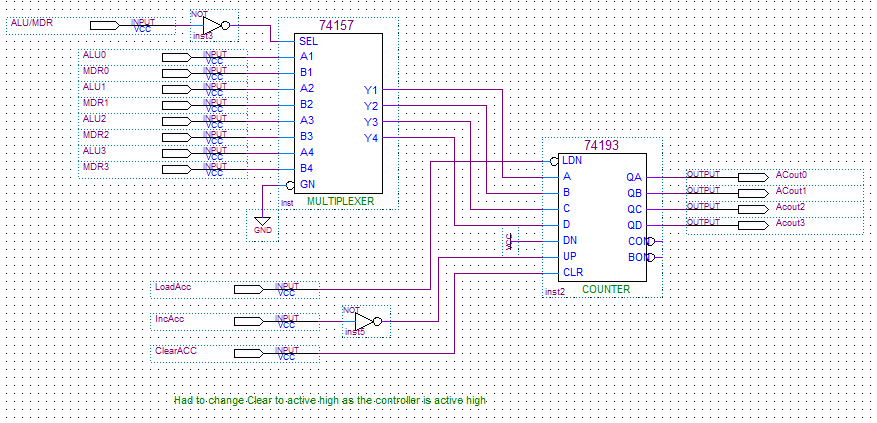
**

**Figure 17:** *Schematic for Memory Address Register*

**

**Figure 18:** *Schematic for**ALU*

******

**Figure 19:** *Schematic for Accumulator*

**Figure 20:** *Verilog for seven segment display decoders*

//Four bit binary to seven-segment decoder. Active Low output

**module** fourtosev**(**

**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**;**

4'b0001**:** **{**a**,**b**,**c**,**d**,**e**,**f**,**g**}=** 7'b1001111**;**

4'b0010**:** **{**a**,**b**,**c**,**d**,**e**,**f**,**g**}=** 7'b0010010**;**

4'b0011**:** **{**a**,**b**,**c**,**d**,**e**,**f**,**g**}=** 7'b0000110**;**

4'b0100**:** **{**a**,**b**,**c**,**d**,**e**,**f**,**g**}=** 7'b1001100**;**

4'b0101**:** **{**a**,**b**,**c**,**d**,**e**,**f**,**g**}=** 7'b0100100**;**

4'b0110**:** **{**a**,**b**,**c**,**d**,**e**,**f**,**g**}=** 7'b0100000**;**

4'b0111**:** **{**a**,**b**,**c**,**d**,**e**,**f**,**g**}=** 7'b0001111**;**//7

4'b1000**:** **{**a**,**b**,**c**,**d**,**e**,**f**,**g**}=** 7'b0000000**;**//8 all on

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 21:***Verilog Code for controller part ‘A’*

**module** accController**(** clk**,** CLR**,**INCA**,**CLRA**,**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**);**

**input** clk**,** CLR**,** INCA**,** CLRA**;**

**output** C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**;**

**reg** **[**2**:**0**]** state**;**

**reg** C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**;**

**always** **@(** **posedge** clk**,** **posedge** CLR **)**

**begin**

**if(** CLR **)**

state **<=** 3'b000**;**

**else**

**begin**

**case(** state **)**

3'b000**:**//State A

**begin**

**if(**1**)** state **<=** 3'b001**;**

**else** state **<=** 3'b001**;**

**end**

3'b001**:**//State B

**begin**

**if(**1**)**

state **<=** 3'b010**;**

**else** state **<=** 3'b010**;**

**end**

3'b010**:**//State C

**begin**

**if(**1**)** state **<=** 3'b011**;**

**else** state **<=** 3'b011**;**

**end**

3'b011**:**//State D

**begin**

**if(**1**)** state **<=** 3'b100**;**

**else** state **<=** 3'b100**;**

**end**

3'b100**:**//State E

**begin**

**if(**INCA**)** state **<=** 3'b110**;** //If Increment Accumulator is active, go to state F

**else** **if(**CLRA**)** state **<=** 3'b101**;**// If Clear Accumulator is active, go to State G

**else** state**<=**3'b001**;**

**end**

3'b110**:**//State F

**begin**

**if(**1**)** state **<=** 3'b001**;** //go back to B not A

**else** state **<=** 3'b001**;**

**end**

3'b101**:**//State G

**begin**

**if(**1**)** state **<=** 3'b001**;** //go back to B not A

**else** state **<=** 3'b001**;**

**end**

**endcase**

**end**

**end**

**always** **@(**state**)begin**

**case(**state**)**

3'b000**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**}=**8'b10000000**;**//State A activate C0

3'b001**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**}=**8'b00100000**;**//State B activate C3

3'b010**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**}=**8'b00010000**;**//State C activate C4

3'b011**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**}=**8'b00001000**;**//State D activate C42

3'b100**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**}=**8'b01000100**;**//S activate C2 and C7

3'b110**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**}=**8'b00000001**;**//State F activate C9

3'b101**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**}=**8'b00000010**;**//State G activate C8

3'b111**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**}=**8'bzzzzzzzz**;**//State H keep all outputs off

**endcase**

**end**

**endmodule**

**Figure 22:***Verilog Code for Controller part B*

**module** controllerV2**(** clk**,** CLR**,**INCA**,**CLRA**,**LDA**,**STA**,**ADD**,**JMP**,**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**);**

**input** clk**,** CLR**,** INCA**,** CLRA**,** LDA**,**STA**,**ADD**,** JMP**;**

**output** C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**;**

**reg** **[**3**:**0**]** state**;**

**reg** C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**;**

**always** **@(** **posedge** clk**,** **posedge** CLR **)**

**begin**

**if(** CLR **)**

state **<=** 4'b0000**;**

**else**

**begin**

**case(** state **)**

4'b0000**:**//State A

**begin**

**if(**1**)** state **<=** 4'b0001**;**

**else** state **<=** 4'b0001**;**

**end**

4'b0001**:**//State B

**begin**

**if(**1**)**state **<=** 4'b0010**;**

**else** state **<=** 4'b0010**;**

**end**

4'b0010**:**//State C

**begin**

state **<=** 4'b0011**;**

**end**

4'b0011**:**//State D

**begin**

**if(**1**)** state **<=** 4'b0100**;**

**else** state **<=** 4'b0100**;**

**end**

4'b0100**:**//State E

**begin**

**if(**INCA**)begin**

state **<=** 4'b0110**;** //If Increment Accumulator is active, go to state F

**end** **else** **if(**CLRA**)** **begin**

state **<=** 4'b0101**;**// If Clear Accumulator is active, go to State G

**end** **else** **if(**LDA**)** **begin**

state **<=** 4'b0111**;**//If Load go to state H branch later

**end** **else** **if(**STA**)** **begin**

state **<=** 4'b0111**;**//If store go to state H branch later

**end** **else** **if** **(**ADD**)begin**

state **<=**4'b1110**;**//Go to State O

**end** **else** **if** **(**JMP**)** **begin**

state **<=**4'b0111**;**//If store go to state H branch later

**end** **else** **begin**

state **<=**4'b0001**;**//if nothing else go to state B

**end**

**end**

4'b0110**:**//State F

**begin**

**if(**1**)** state **<=** 4'b0001**;** //go back to B not A

**else** state **<=** 4'b0001**;**

**end**

4'b0101**:**//State G

**begin**

**if(**1**)** state **<=** 4'b0001**;** //go back to B not A

**else** state **<=** 4'b0001**;**

**end**

4'b0111**:**//State H

**begin**

**if(**LDA**)** **begin** //If load go to statego to state I

state **<=**4'b1000**;**

**end** **else** **if(**STA**)** **begin** //If storign go to state M

state **<=** 4'b1100**;**

**end** **else** **if(**JMP**)** **begin** //If Jump go back to state C

state **<=** 4'b1111**;**

**end** **else** **begin** //If neither go back to State B fail-safe for troubleshooting

state **<=** 4'b0001**;**

**end**

**end**

4'b1000**:**//State I

**begin**

state **<=** 4'b1001**;**//Go to state J

//else state <= 4'b0001;

**end**

4'b1001**:**//State J

**begin**

**if(**LDA**)** state **<=** 4'b1010**;**//Go to state K

**else** **if** **(**JMP**)** state **<=** 4'b1111**;** //Go to state P

**else** state **<=** 4'b0001**;**

**end**

4'b1010**:**//State K

**begin**

**if(**LDA**)** state **<=** 4'b1011**;**//Go to state L

**else** **if(**JMP**)**

state**<=** 4'b0001**;**

**else** state **<=** 4'b0001**;**

**end**

4'b1011**:**//State L

state **<=** 4'b0001**;**//Go back to state B for next Instruction

4'b1100**:**//State M

**begin**

**if(**1**)** state **<=** 4'b0001**;**//Go back to state B for next Instruction

**else** state **<=** 4'b0001**;**

**end**

4'b1101**:**//State N--Not used

**begin**

**if(**1**)** state **<=** 4'b1110**;**//Go to State O

**else** state **<=** 4'b0001**;**

**end**

4'b1110**:**//State O

**begin**

**if(**1**)** state **<=** 4'b1011**;**//Go to state L

**else** state **<=** 4'b0001**;**

**end**

4'b1111**:**//State P

**begin**

**if(**1**)** state **<=** 4'b0010**;**//Go to state B

**else** state **<=** 4'b0010**;**

**end**

**endcase**

**end**

**end**

**always** **@(**state**)begin**

**case(**state**)**

4'b0000**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b1000000000001**;**//State A activate C0

4'b0001**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0000000000001**;**//State B activate C3--Off at this point (select PC)

4'b0010**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0001000000001**;**//State C activate C4 Read Memory

4'b0011**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0000100000001**;**//State D activate C42 Transfer memory

4'b0100**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0100010000001**;**//S activate C2 and C7

4'b0110**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0000000100001**;**//State F activate C9

4'b0101**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0000001000001**;**//State G activate C8

//added for part B

4'b0111**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0010000000001**;**//State H activate C3--C3 on at this point (select MDO) (Jump)

4'b1000**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0011000000001**;**//State I activate C4 Read Memory

4'b1001**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0010100000001**;**//State J activate C42 Transfer memory

4'b1010**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0000000000001**;**//State K activate C10=0 ACC Select<-MDO

4'b1011**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0000000000010**;**//State L activate C11 MDI<-ACC (Load) load active low now

4'b1100**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0011000001001**;**//State M activate C5 C4 Write to memory (Store)

4'b1101**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0000000000001**;**//State N activate C15 and C14 Add=00,

4'b1110**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0000000000011**;**//State 0 activate C10=1 Select ALU for ACC

4'b1111**:** **{**C0**,**C2**,**C3**,**C4**,**C42**,**C7**,**C8**,**C9**,**C1**,**C5**,**C6**,**C10**,**C11**}=**13'b0000000010001**;**//State P activate C1 load PC current MDI

**endcase**

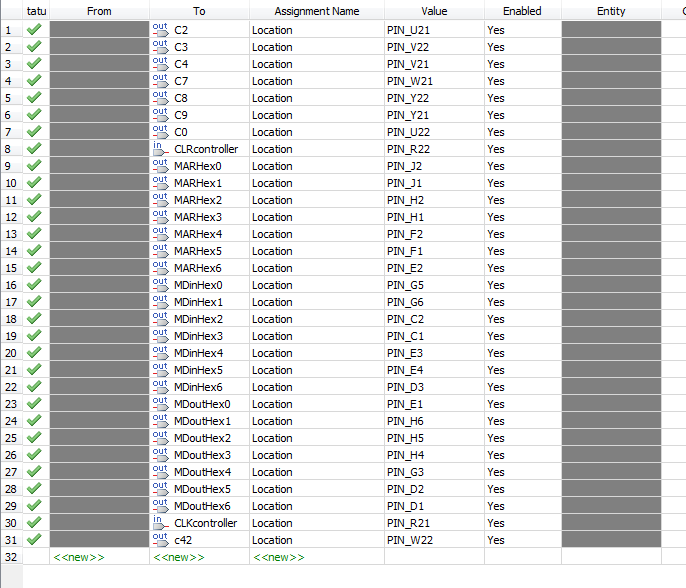
**end**

**endmodule**

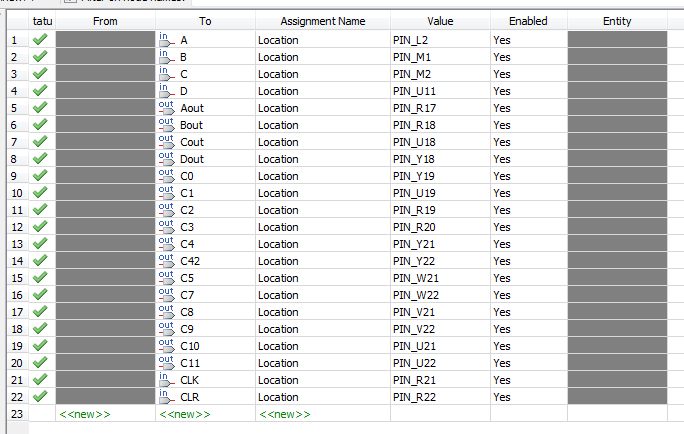
**-DE1 Pin Assignments (3.3)**

The pin assignments for part ‘A’ (Figure 23) are as follows:

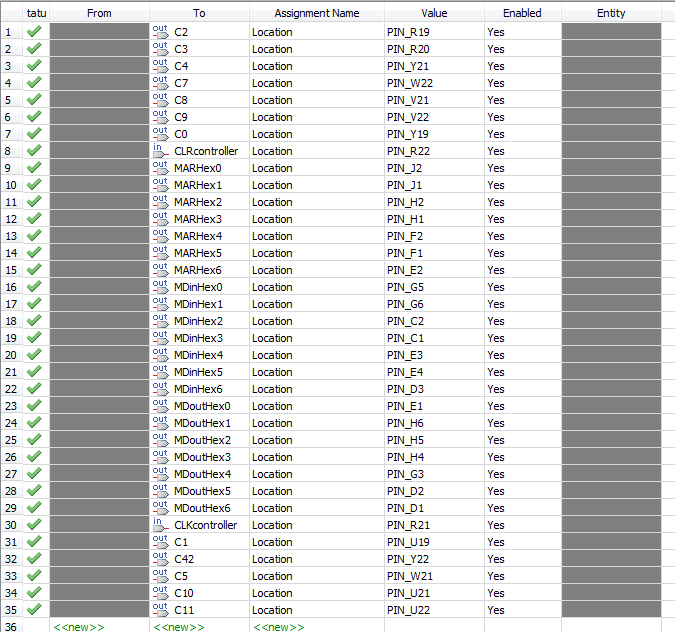
**Figure 23:***Pin assignments for part ‘A’ controller*

**

**Figure 24:** *Pin assignments for part ‘B’*

**

**Figure 25:** *Pin assignments for part ‘C’*

**

**Alternative Design Considerations (4)**

Multiple designs were purposed for the final project. Three major alternatives were purposed. The first was a strictly hardware based controller using Karnaugh maps it is a standalone controller strictly based on hardware. The second was a phase controller design using a twisted ring counter. The last controller was a Verilog based controller that used a behavioral syntax structure.

**-Alternatives Considered (4.1)**

*First alternative:* Basic hardware finite state machine. The first alternative uses a standard development of a finite state machine with state assignments that start at zero and count up. This design provides a quick solution however the logic equations for the D-Flip-Flops are large and require an increased amount for logic gates. This could also reduce the execution efficiency. It also provides a more difficult adaptation for future designs.

*Second alternative:* Phase based controller. This option provides a twisted ring counter along with the standard finite state machine logic. Even though there is more hardware it provides a cleaner state transistion resulting in a smaller logic equations for the D-Flip-Flops and also less logic gates therefore making it a more hardware efficient design.

*Third alternative:* Verilog based controller. This option is the longer of all three approaches as Verilog is still a relatively new technology to most students in the class. Among the learning curve of the syntax this is the longest alternative to implement. However, this option also allows for quick change implementation. If a bug is found in the overall design it may be easier to switch by changing a line or two of code instead of completely rerunning analysis on the Karnaugh maps.

**-Reasons for Selection of Final Design (4.2)**

Design three was chosen (Verilog Code) as it is the most flexible. As this is the first big digital design project I have done this consideration weighted heavily in the selection. Another reason to consider is personal skill set improvement. Verilog is more in demand as a skill than creating a finite state machine from a Karnaugh map (kmaps). The last consideration is scalability. Verilog is easily scalable to five six or seven bit state sequences. However, to do this with just kmaps would take large amount of calculation. Each new feature would take increasingly longer with the other two designs than with Verilog.

**Integration and Test Plan (5)**

This section will provide information on how integrating the controller and other subsystems together along with how each subsystem and overall system was tested.

**-Integration Strategy (5.1)**

The strategy in integration took place was based on the fetch and decode/execute operations provided in the class slides. The first operation in the instruction set is to clear the program counter. The controller is first implemented in the top level design and then only connected to the program counter.

These modules by themselves are tested using a waveform and input/output pins. This method is then carried throughout the integration. Second the Memory Address Register is installed and tested. Thirdly the TRISC memory module. After the memory was integrated the Instruction Register and then the Accumulator were integrated. This method takes longer however it reduces the amount of errors encountered at each step and eases troubleshooting.

**-Test Strategy (5.2)**

Testing was broken into three major phases. The first phase is unit testing. Each subsystem or unit was tested by itself with no external hardware hooked up. A waveform was then created with multiple test bench I/O’s. Unit testing reduces the errors when the whole systems is connected. The next phase is integration testing. As described in the previous section as each subsystem is connected to one another it was tested to ensure accuracy. This again was verified by waveforms and test bench I/O’s. The last step is system testing. All subsystem modules are connected and tested as one unit. Again waveforms and test bench I/O’s were used. After the system waveforms were completed pins were assigned and the project was programmed to the DE1. All waveforms that were executed are in the appendix for reference.

**-Simulation Results from DE1 implementation (5.3)**

For each part of the project it was programmed into the DE1 and demonstrated to the lab instructor. The three seven segment displays indicated the Memory Address being read (HEX0) the instruction currently being executed (HEX1) and the data that is being stored in the accumulator (HEX3). Along with the seven segment displays the green LEDs are used to provide indication of where the controller was in the state sequence.

The part ‘A’ demonstration showed the Memory Addresses being increased by one and the operational code bouncing between code six (increment ACC) and code seven (clear ACC). As the controller executed its states the accumulator data increased by one or cleared to zero.

The part ‘B’ demonstration is the second controller with no subsystems connected. Using the slides switches as the op codes and Key 1 and Key 0 as the clock and clear buttons the controller stepped through the state sequence by lighting up each representing green LED as described in the state diagram in section 3.2.

The part ‘C’ demonstration takes the part ‘B’ controller and integrates it with the rest of the TRISC controller. Similar to the part ‘A’ demonstration the seven-segment displays, Key 1 and Key 0 along with the green LED’s are again used for control inputs and outputs. However, part ‘C’ the outputs in the seven segment display varied in this section. The address would increase by one but would also jump around as the Jump commands were executed. Also the op code display also switched between zero, one, two, six, seven, and eight. This relate to Load, Store, Add, Increment, Clear, and Jump commands. The third seven segment display that provides information on the data in the accumulator incremented and cleared along with the load and store commands which would change depending on the operands in the current memory address.

**Conclusion (6)**

This section provides final insight and project wrap up information.

**-Resolution of Design and/or Implementation Issues (6.1)**

The first issues that were encountered was the integration of each subsystem. As our controllers are active high some of the subsystem clears and load signals were active low. This essentially has each sub module either clearing or loading incorrect data to outputs pins. This was resolved by adding an additional not gate to these control lines.

The second issue was with part ‘C’ as the ALU and the Accumulator were connected and tested a vector mismatch error was created during the waveform simulation. It was suggested by the professor to program and test directly from the DE1. This worked however HEX2 had mixed results afterward.

The last issue was the effect of random lights turning on the DE1 when demonstrating, the seven segment display on HEX 4 and multiple red LED’s would turn on randomly even though outputs were not assigned to these pins. I discussed this issue with the lab instructor and other fellow students and it appears to be a common problem. However, this issue did not affect the functionality of the project.

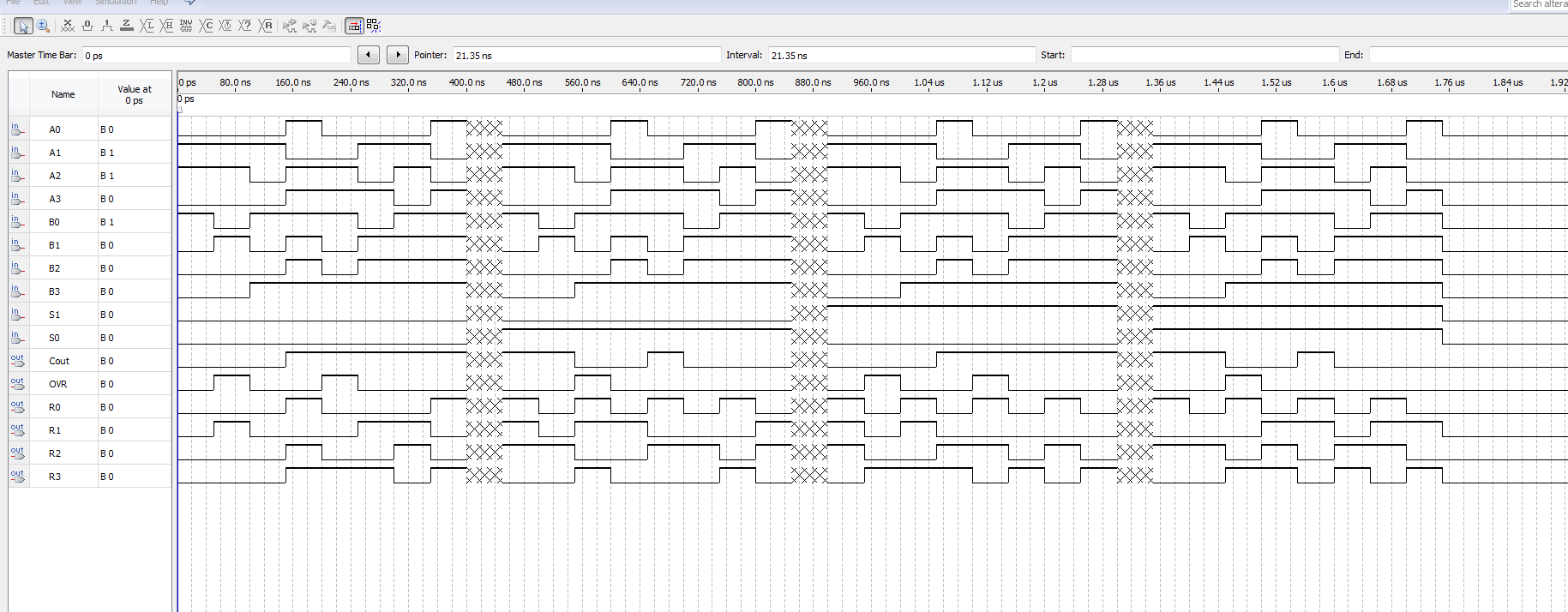
**-Lessons Learn (6.2)**

The first lesson learned from this project is the scheduling time and time management. As with all projects starting early are truly understanding the time scope of the project is crucial in student success and outcomes and also being a good employee in the labor force. Secondly, when encountering a road block that is not resolved in an acceptable amount of time reach out for help sooner and move on to the next priority on the project list. This was encountered with part ‘C’ of the project with the “vector mismatch error” after 6 hours of troubleshooting and not getting anywhere I decided with the time constraint to put that particular problem aside and move onto the Project report and other functional implementations. This allowed the project to continue to grow closer to completion instead of stagnating in one task.

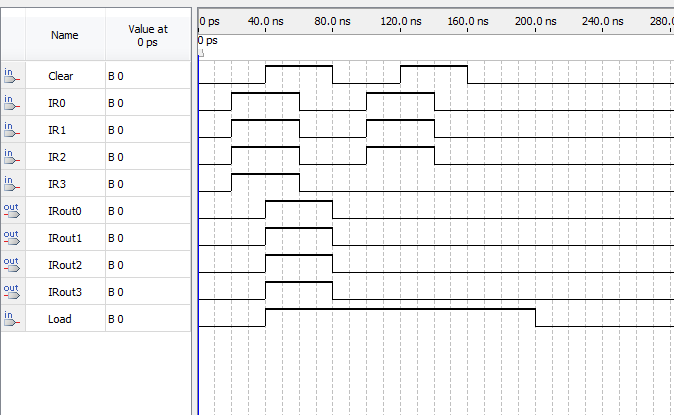
**Appendices (7)**

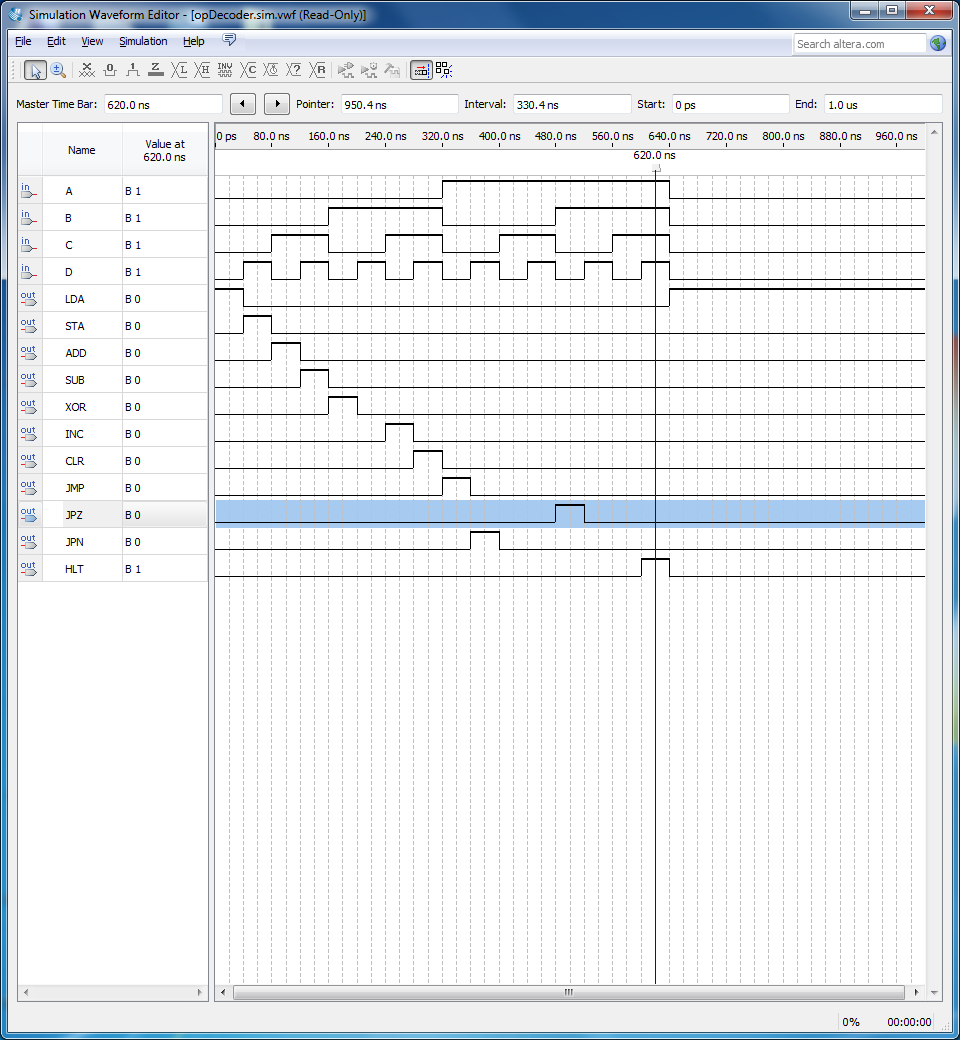
This appendices provides testing results for multiple subsystems and top level systems in form of waveform test results.

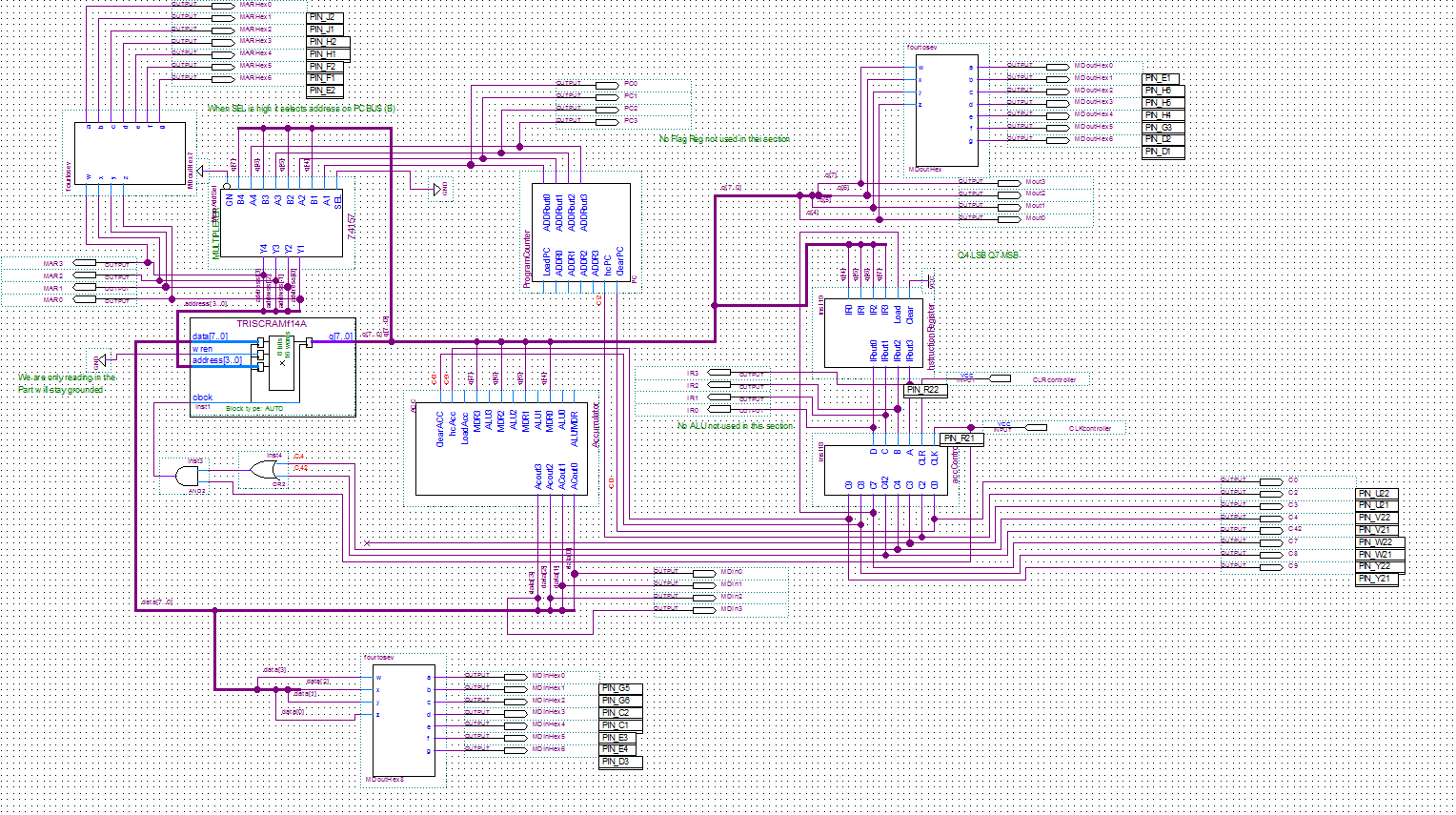
**APX1:** *ALU Waveform*

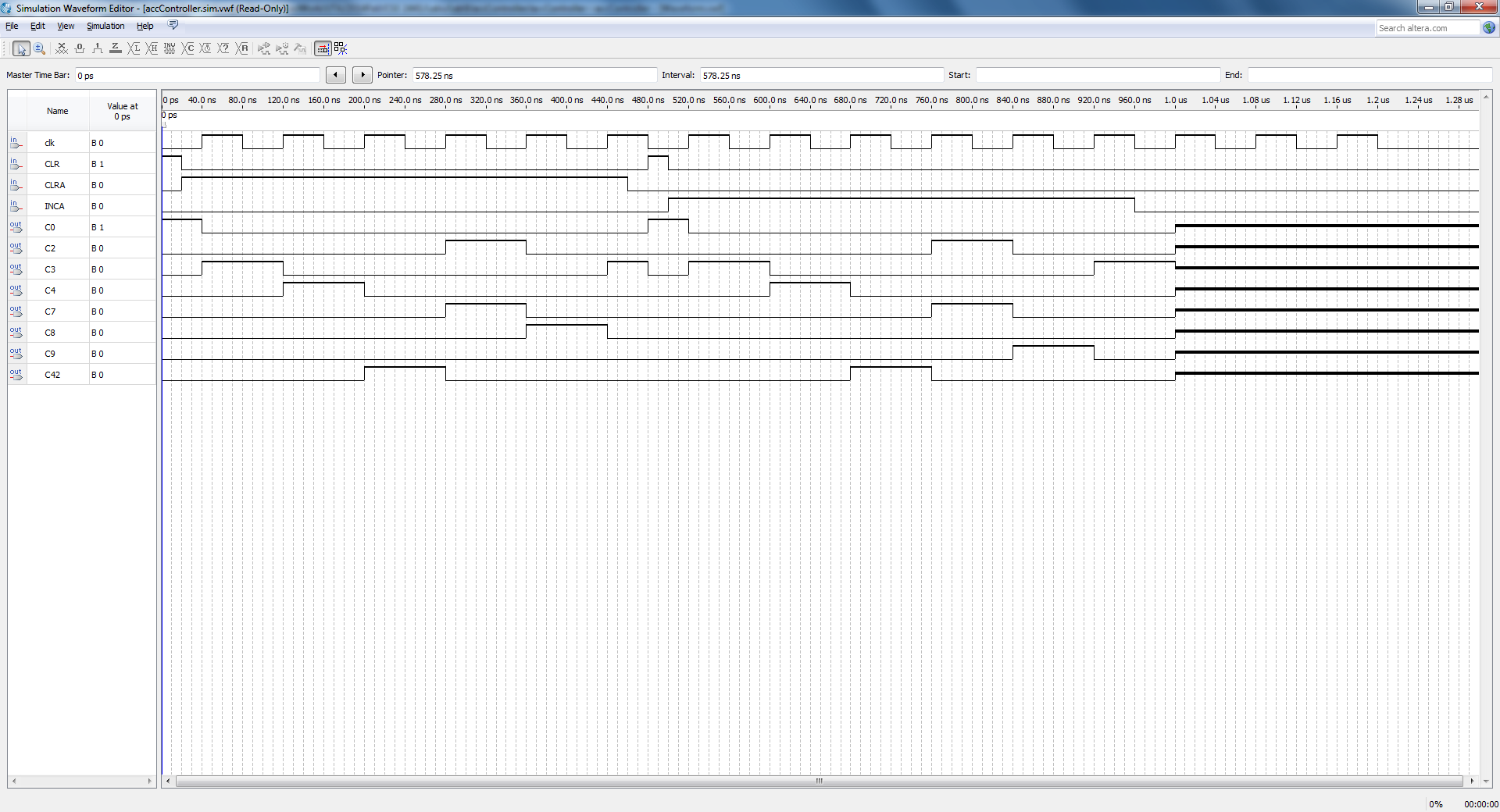


**APX2:** *Instruction Register waveform*

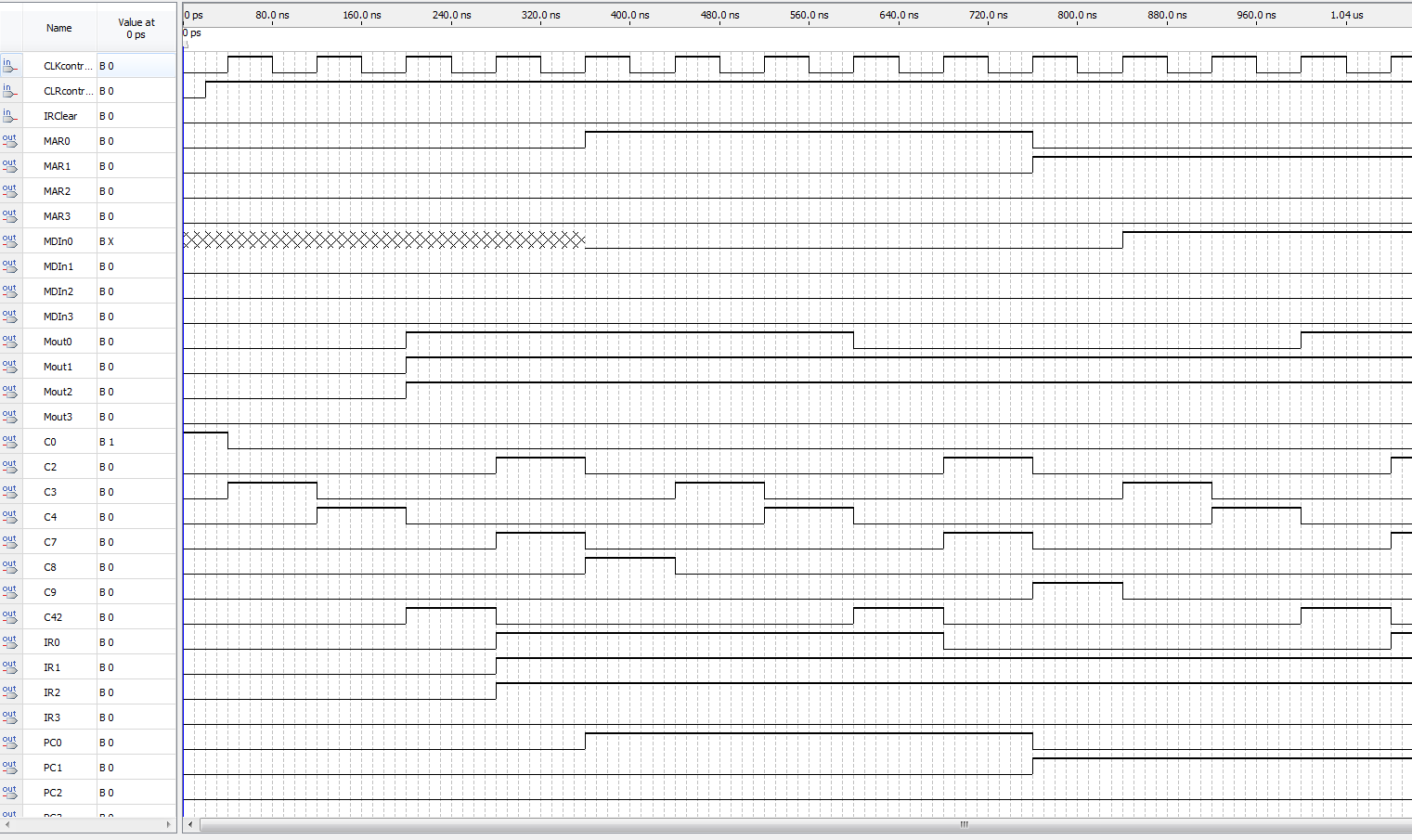
**

***APX3:***Op Code Decoder Waveform**

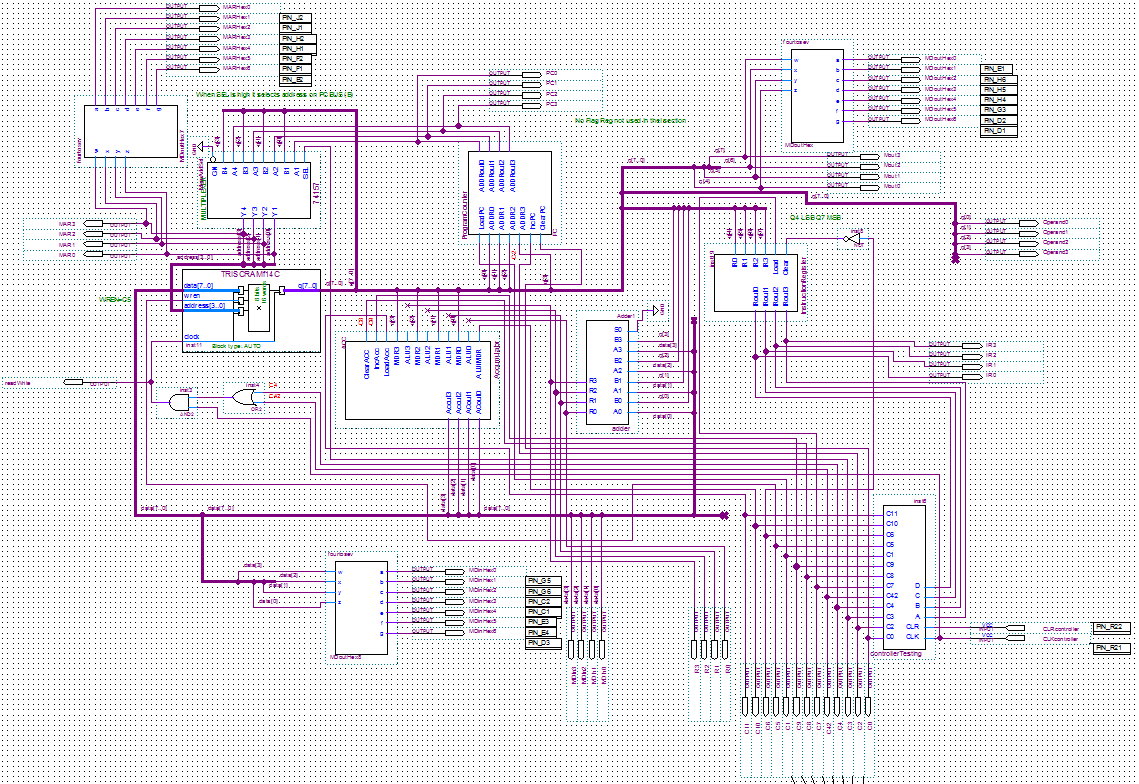
**APX4:** Part A system schematic******

**APX5:** *Part A controller Waveform*

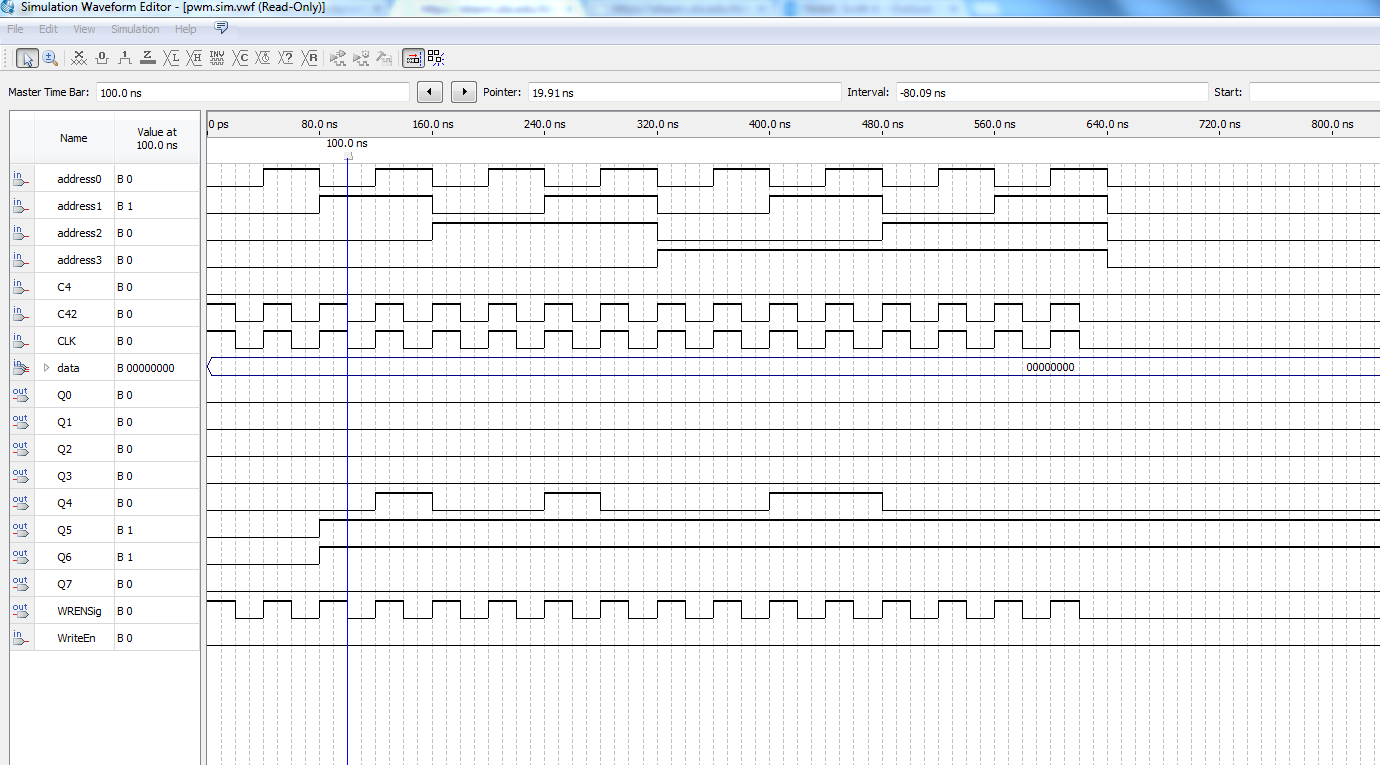
**APX6:** *Part A integration*

**

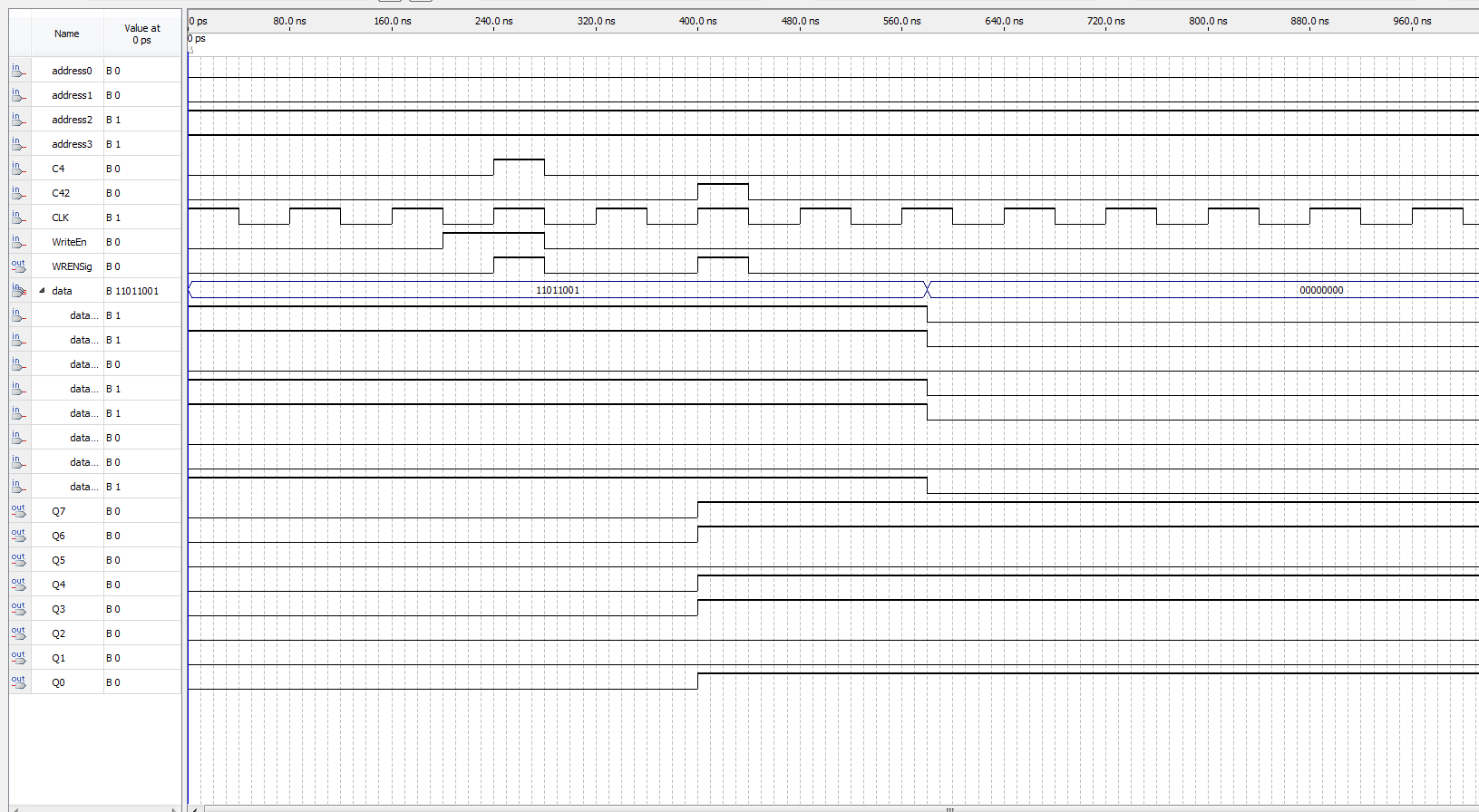
**APX7:** *Part ‘C’ schematic*

****

**APX7:** Reading standalone memory

**

**APX8:** *Writing to memory and then reading from it.*

**