**Term Project**

*Dr. Carroll*

*Freddy Aguinaga*

*11/29/2018*

**Table of Contents**

|  |  |
| --- | --- |
|  |  |
| Introduction …………………………………………………………………………… | Pg. 3 |
| System Level …………………………………………………………………………… | Pg. 3 |
| Subsystem Level ……………………………………………………………………….. | Pg. 4 |
| Hierarchical Design …………………………………………………………………….. | Pg. 8 |
| Operation Procedure ……………………………………………………………………. | Pg. 9 |
| Controller Design Details ………………………………………………………………. | Pg. 9 |
| Alternatives Design Considerations ……………………………………………………. | Pg. 10 |
| Integration and Test Plan ………………………………………………………………. | Pg. 10 |
| Conclusion ……………………………………………………………………………... | Pg. 14 |

**List of Figures and Tables**

|  |  |
| --- | --- |
| Figure 1.1: Altera DE1 Board | Pg. 3 |
| Figure 3.1.1: Register | Pg. 4 |
| Figure 3.2.1: Binary Counter | Pg. 4 |
| Figure 3.3.1: Accumulator | Pg. 5 |
| Figure 3.4.1: ALU | Pg. 5 |
| Figure 3.5.1: RAM | Pg. 6 |
| Figure 3.6.1: Control Unit Block Diagram | Pg. 6 |
| Figure 3.6.2: Instruction Decoder Verilog | Pg. 7 |
| Figure 3.6.3: Controller Verilog | Pg. 7 |
| Figure 4.1: Hierarchy of TRISC | Pg. 8 |
| Figure 5.1: Instruction Cycle | Pg, 9 |
| Figure 6.1: State Diagram | Pg. 10 |
| Figure 8.1.1: TRISC organization comparison | Pg. 11 |
| Figure 8.3.1: ALU waveform | Pg. 12 |
| Figure 8.3.2: Adder\_Substractor waveform | Pg. 12 |
| Figure 8.3.3: Ripple Carry Adder | Pg. 13 |
| Figure 8.3.4: Full Adder | Pg. 13 |
| Figure 8.4.1: Altera DE1 Executing instruct. | Pg. 14 |

**1 Introduction**

For our term project, we were tasked with constructing and testing a simple computer processor capable of storing and executing instructions from the provided TRISC instruction set. My configuration can successfully execute the 6 required instructions such as Jump, Increment, Clear, Load, Store, and Add. Addresses and data should be displayed on the four 7-segment displays integrated in the Altera DE-1 circuit board provided, as well as the control signals that are being outputted with LEDs.

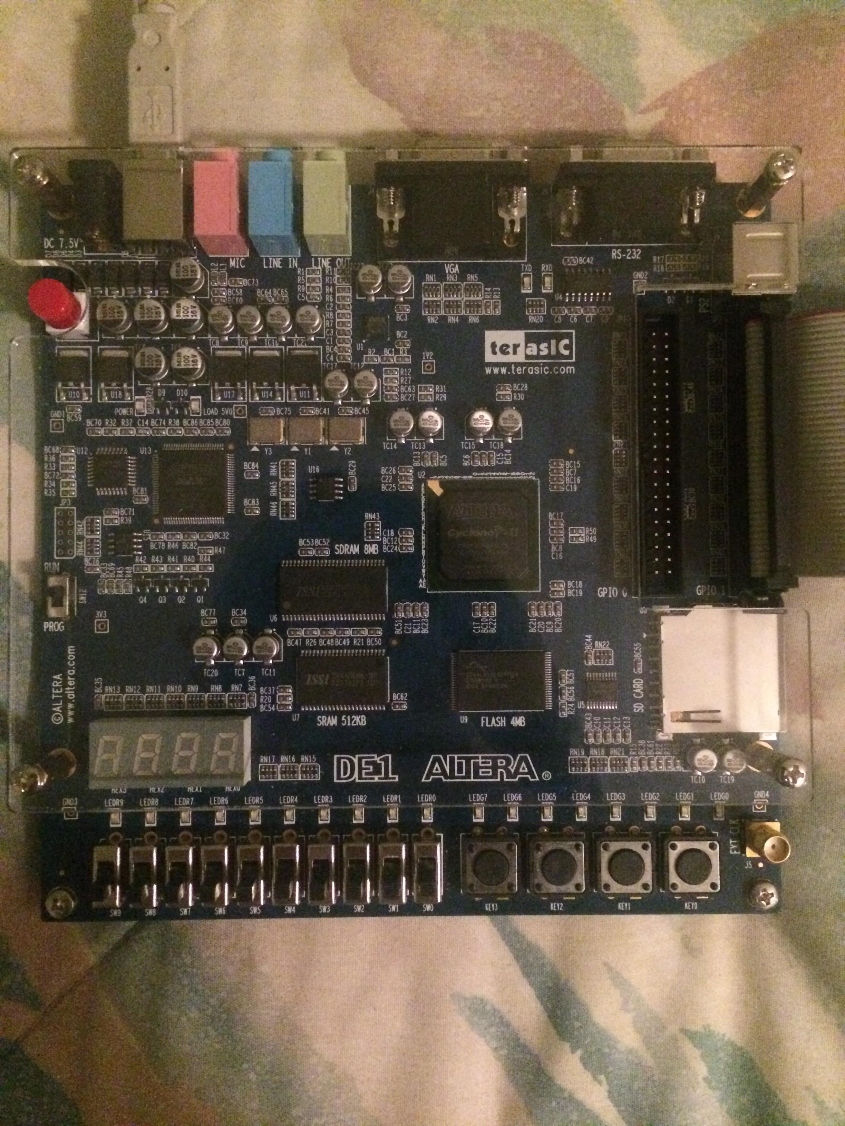


Figure 1.1: Altera DE1 Board

To be considered complete, the processor would need to be able to run a program that was saved in the RAM provided by the instructor.

**2 System Level**

Circuits are realized using Quartus II due to the unavailability of resources and complexity of implementing the processor design with physical chips. Circuits were designed used either Verilog or block diagrams depending on complexity. Block diagrams is more visual by constructing the circuits like a schematic while Verilog is a hardware description language which allows for flexibility in creating the behavior of a device. It becomes more apparent when devices are updated because Verilog can be adapted with just a few changes in code, but the block diagram might require you to reevaluate how many and what type of gates would be needed.

**3 Subsystem level**

The types of components can be divided into two types: combinational and sequential. Combinational circuits cannot contain any data previously inputted through, so they are used for determining paths and calculations. Sequential circuits are used for buffering and saving the results of certain operations.

**3.1 Register:** The register is an essential component as it is a storing device that can hold data for later use. It is used as the instruction register which holds the operation data until the controller is ready to accept a new instruction. The registers we used are 4-bit registers that only override data if the load input changes from logic 1 to 0 due to the NOT gate making it into negative edge activated. It can also be reset by setting the clear input to logic 0 because it is also contains a NOT gate indicated by the circle at the port. Devices like the accumulator contain a register in order to hold the data passed onto it by the ALU or MDO (memory data out) bus.

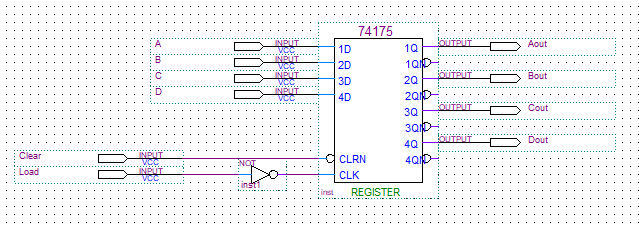


Figure 3.1.1: 4-bit Register

**3.2 Program Counter:** It is a device that is very similar to a register with the load and clear functionality as well as being able to count up and down. It is essentially a binary counter shown in figure 2.2.1. We used it to increment through the addresses of the RAM to retrieve instructions one by one.

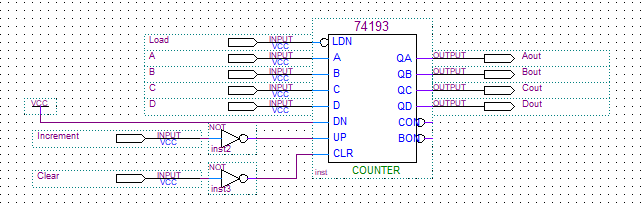


Figure 3.2.1: Binary Counter

**3.3 Accumulator:** The accumulator is in essence a program counter with the ability to store data from either the ALU or the data retrieved from the RAM. Data from both are set on the inputs of the multiplexer and the input ALU/MDR will decide what is transferred to the binary counter (figure 2.3.1). We used it to simply store the result of the instructions executed with value current stored within the accumulator. It is always outputting what is stored in its binary counter to the ALU and the last 4 bits of the MDI (Memory Data In) bus.

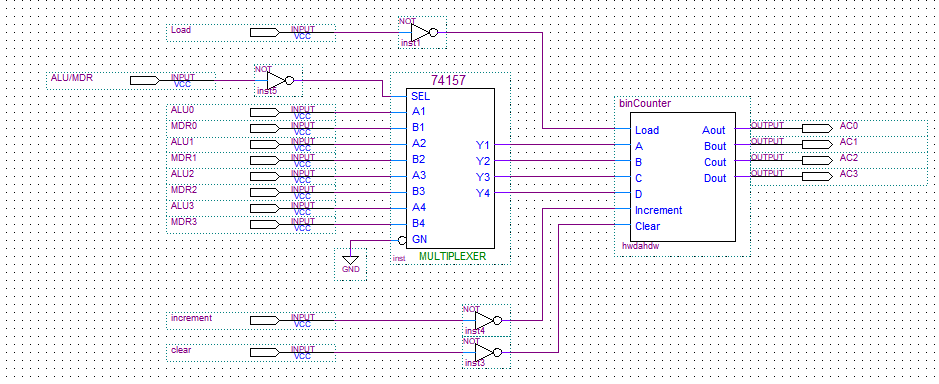


Figure 3.3.1: Accumulator

**3.4 Arithmetic Logic Unit (ALU):** The ALU is able to do various operations like addition, subtraction, AND and XOR. Like Accumulator, it is able to select between these operations with 2 multiplexers and various AND and OR gates. It is also capable of detecting overflow which is when calculations go beyond what is representable with 4 bits. My design contains two overflow devices (figure 2.4.1) due to a flaw in wiring which wouldn’t detect overflow correctly when subtracting. However, in this project we used the ALU only for the add instruction which would add the ACC’s current data and last 4 bits (operand) provided by the RAM.

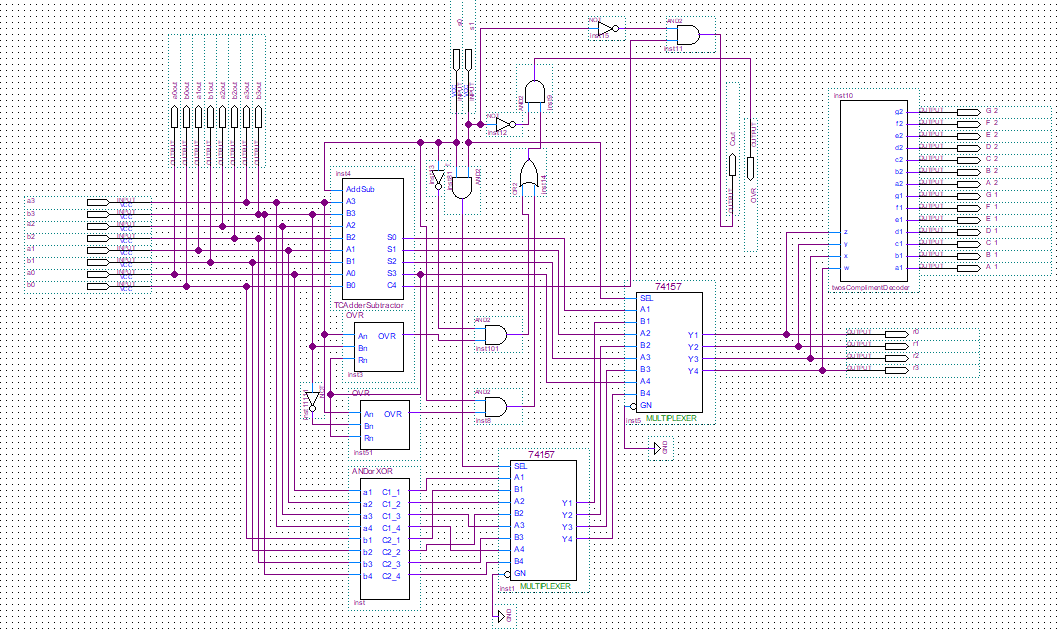


Figure 3.4.1: ALU

**3.5 RAM (Rapid Access Memory):** is like a storage box with multiple registers that can be accessed individually by entering 4-bits into the address input bus. Like the Program Counter, current must pass through the clock input to store the new address in the Memory Address Register (MAR). However, there needs to be a second 0-1 transition at the clock input for the register within the given address to begin outputting its contents. The Data bus and the wren input ports shown in Figure 3.5.1 are for writing into the registers. To write, wren would have to be set to logic 1, and the data bus should have the values that are going to be saved. At this time the clock would have to go through 2 cycles (two logic 0 to logic 1 transitions) to go to the address in the address bus and override the data in the register. The RAM device that we are using contains 16 different registers that can hold up to 8 bits of data. The left-most 4 bits were used to determine the operation to be executed and the last 4 bits are the operand address. Figure 3.5.1 is not a block diagram, but rather a symbol that simplifies the Verilog programming that was used to create the RAM.

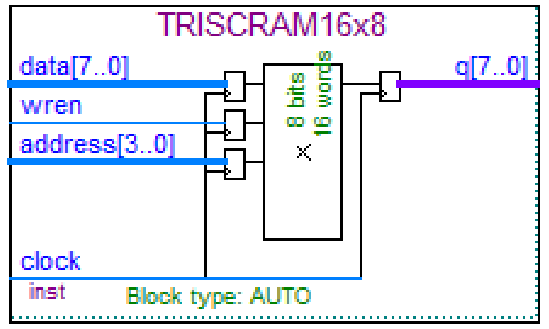


Figure 3.5.1: RAM

**3.6 Control Unit:** The Control Unit is like the manager of all the devices because it is what decides what will be needed depending on the opcode that it is given by the Instruction Register. The Control Unit consists of an instruction decoder and a controller as shown in the Figure 3.6.1.

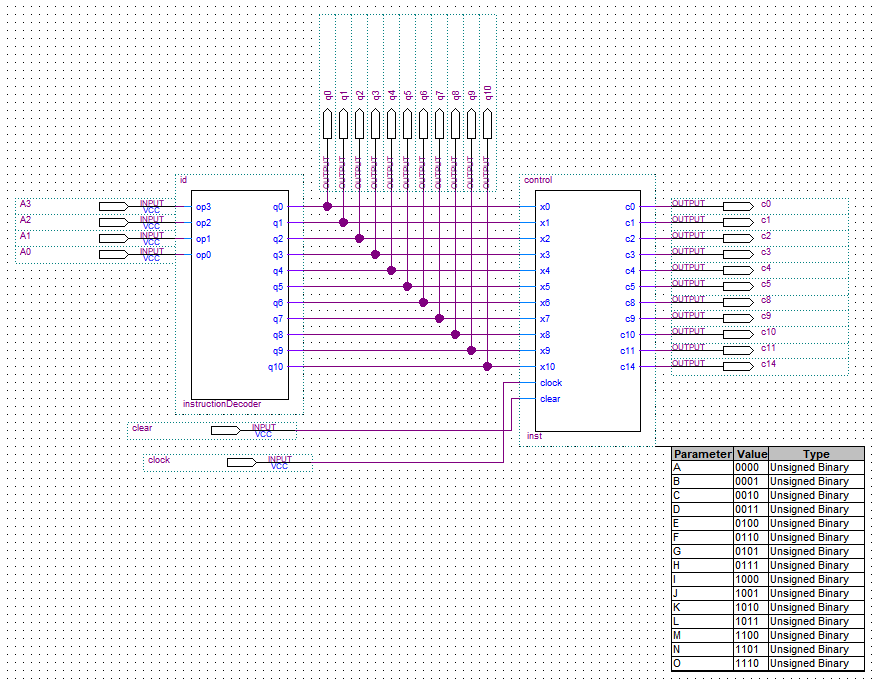


Figure 3.6.1: Control Unit block diagram

Both devices were realized using Verilog due to its ability to easily accept alterations in behavior which would prove its significance during testing.

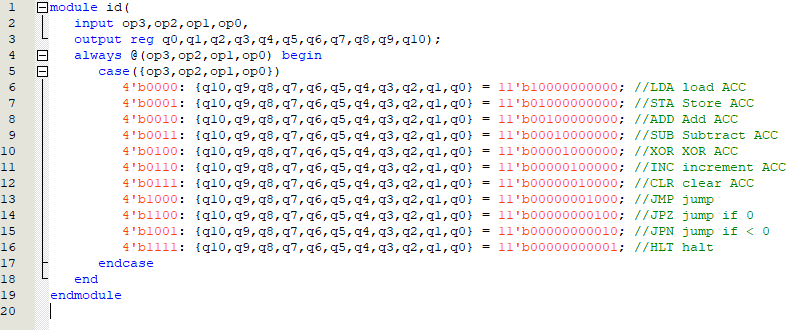
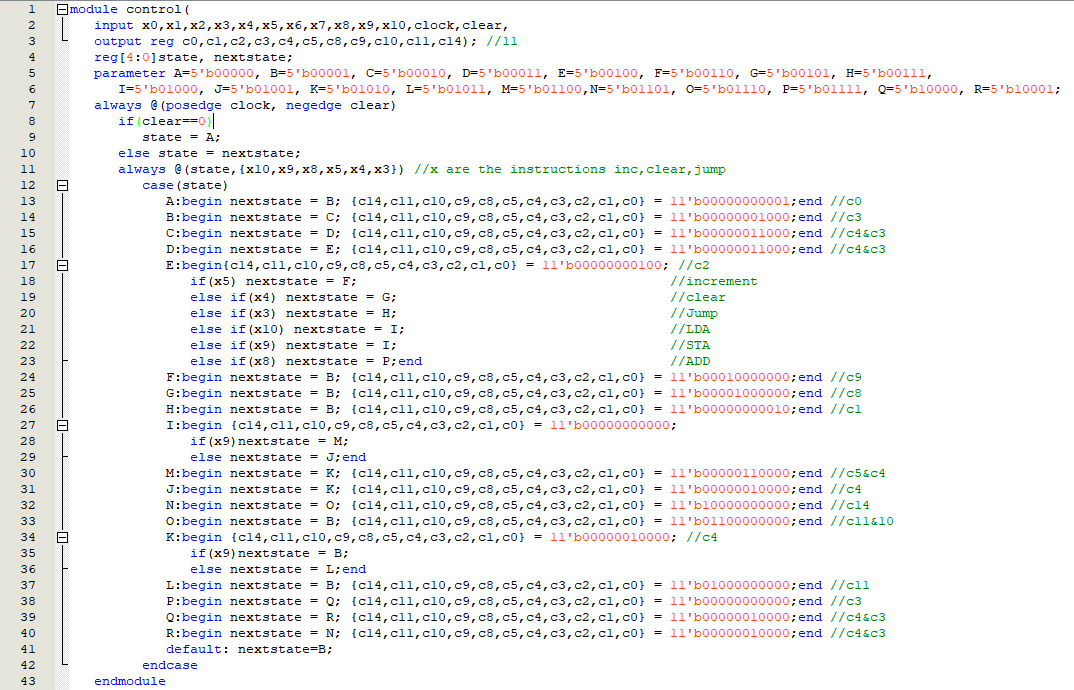


Figure 3.6.2: Instruction Decoder Verilog

The instruction decoder (ID) accepts an opcode to determine what out of the 11 outputs to send to the controller. Each output of the ID corresponds to each of instructions within the TRISC instruction set provided (Figure 3.6.4). After the controller accepts the output of the ID, it would then output a series of control signals that are connected to the each of devices within the processor. The x inputs in Figure 3.6.3 are the possible inputs it can receive from the ID.

  
Figure 3.6.3 Controller Verilog

The controller is a combination of sequential and combinational circuits that together form a device that cycles through certain stages that output their respective control signals according to the instruction that is given.

**3.7 address-To-Hex Decoders:** These are directly connected to the 7-segment displays in order to convert 4-bits into a hex number (0-F). These were realized using Verilog (Figure 3.7.1) for convenience because decoders generally require several gates and much more effort to get the outputs for all different combinations of the 4 inputs. No Verilog could be shown due to a human error.

**4 Hierarchical Design Structure**

Many of the devices mentioned before are comprised of other smaller devices/ components that help achieve a much more complex function. An example of this can be seen in the ALU which has the 4 full adders, an overflow detector, multiplexers, etc. These full adders complete a simple task of adding two bytes and outputting the sum and carry. These devices then are used in the design of an ALU which has the task of being able to choose between 4 different operations. Just like the ALU the processor made of up of all the individual devices made in previous labs. This is shown in Figure 4.1. The labs in yellow were directly used in the project while the green are building blocks either the labs in yellow or the project. The project directly depends on lab 8 and 9, being the RAM tester and the Control Unit. It also uses lab 4, 5, and 6 which are the decoders for various number representations, ALU, and lastly the memory holding Binary counters and registers. The ALU couldn’t be made without first completing labs 1 and 2 because those build upon each other. Lab 1 is a simple ripple carry adder that is capable of adding two 4-bit numbers. Lab 2 is a continuation of lab 1 to create an Adder-Subtractor which was capable of both addition and subtraction. From there, the ALU used the Adder-Subtractor and other components to be able to execute the 4 operations.

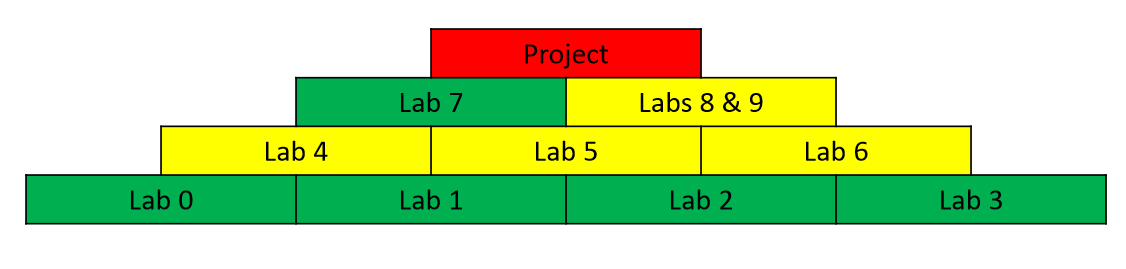


Figure 4.1: Hierarchy of TRISC

|  |  |
| --- | --- |
| **5 Operation Procedure:**  The processor is operated through only 2 keys which are the Control Unit’s Clear and Clock. Every key press that activates the clock on the Control Unit, allows it to transition to the next state and can be reset at any time by simply pressing the Clear key. When the Altera De1 board is first programmed using Quartus II’s programming tool, it sets the controller to state A which outputs the control signal C0 to clear the Program Counter. From there, the controller would then go to state B which outputs control signal C3 that connects to an address selector. This then allows the program counters value to pass through into the address bus on the RAM. Now, the next two states C and D will output control signals C3 and C4. Signal C4 is the clock signal for the RAM and allows for data in the address bus to be saved in the MAR and then outputs the contents at that address to the MDO bus. Next, the processor changes to state E which increments the Program Counter and saves the opcode from the MDO in the Instruction Register. The states B through E are executed every instruction cycle. However, from point E depending on what is stored on the instruction register the controller could go through any of the branches that execute a certain instruction by going through certain states. One example would be where the instruction register holds the 4-bit number 0110. The ID within the controller would identify this opcode as the increment instruction, outputting q5. The controller would recognize the q5 input and go to state H to increment the Accumulator by outputting the control signal C9. The processor continues to go through this cycle (Figure 5.1) until it executed the halt instruction which was never implemented in the final project. | Figure 5.1: Instruction Cycle |

**6 Controller Design Details**

As described before, the controller manages when and what is done in all the devices within the processor. The controller has a total of 10 control signals, each with a different function. It also includes the outputs of the instruction decoder which is not used in my configuration. The controller has 18 states that enable the processor to execute the 6 implemented instructions: Load, Store, Add, Increment, Jump, and Clear. In the diagram below, it shows the different paths that the controller can branch out from State E. This point in time the instruction is decoded and moves into the execute instruction phase. It can be noted that the store and the load instructions split off after state I instead of having their own separate state paths. I decided to do this in order to reduce the amount of states. In doing this, the diagram and Verilog became slightly less readable. Another thing that can be noticed is that there are some states that do not have any control signals. A better way of looking at this is that the states are setting all the control signals to 0. The reason for doing this is because some multiplexers needed the control signal to be 0 to allow a certain set of inputs to go through. For example, at state I, the controller needs to set C3 to 0 to allow the address selector to pass the first 4 bits of the MDO to the input address bus of the RAM.

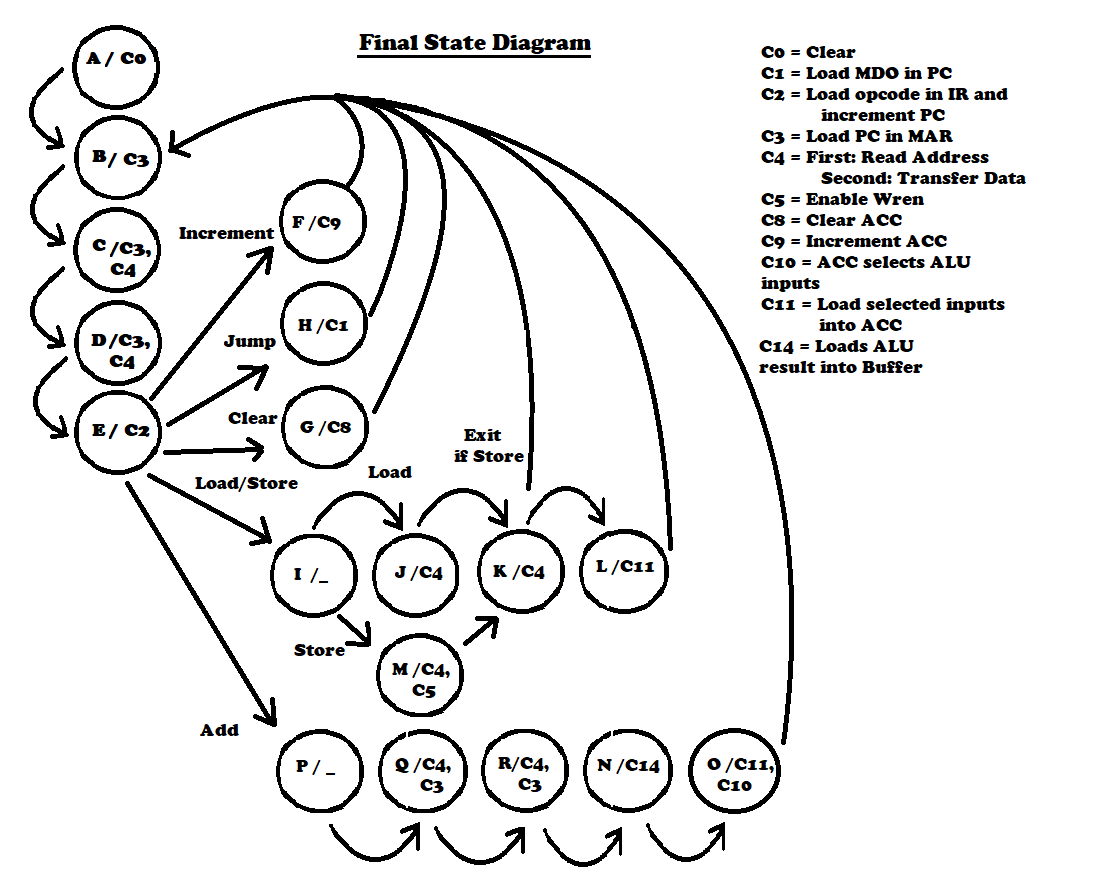


Figure 6.1: State Diagram

**7 Alternative Design considerations**

After completing part A of the project, I noticed that the controller was going to need many states with the same control signals in the same order as states B, C, and D. I considered having states reduced by two by having state P return to state B and have an if statement at state D to check if the add instruction was being executed. However, this would destroy the flow of the state diagram shown above. It would make the diagram harder to read and complicate the process.

**8 Integration and Test Plan**

**8.1 Integration Strategy:** The strategy that I decided to have is to follow the circuit diagram provided shown in Figure 8.1.1 as closely as possible to prevent confusion. I divided the devices in the beginning in the case of having to add gates like the NOT gates as shown below. Also, all the components were tested and checked with the TA to verify that they functioned correctly before being implemented in the TRISC. Not all the components were inserted all at once. In the beginning, I only inserted what was necessary to execute the Clear, Increment and Jump instructions. This would exclude the ALU, and a buffer from the final schematic.

|  |
| --- |
|  |
|  |
| Figure 8.1.1: TRISC organization comparison |

**8.2 Test Strategy:** Before adding any component to the processor, I would test them through the DE1 or using the University Program VWF. I would simulate it with a series of inputs to see the behavior of the device. Once everything seemed to function correctly, it would then be inserted in the processor schematic. Once everything needed to run the instructions were implemented, I would run through the program in the RAM to see if any bugs occur. If any bugs were present, I would check control signals that were being output by the controller, the values that were displayed on the 7-segment displays. When the displays showed a different value than expected, the LED’s that occur during that time will indicate where the bug occurs. For instance, at some point my DE1 showed that the processor was returning to state A which only occurs when the board is first programmed. From this I knew that the issue lied with the controller where I failed to adjust the amount of bits that the states needed to represent all 18 states to 5 bits.

**8.3 Simulation Results:** These are the simulations that I ran on the ALU and the components within it.

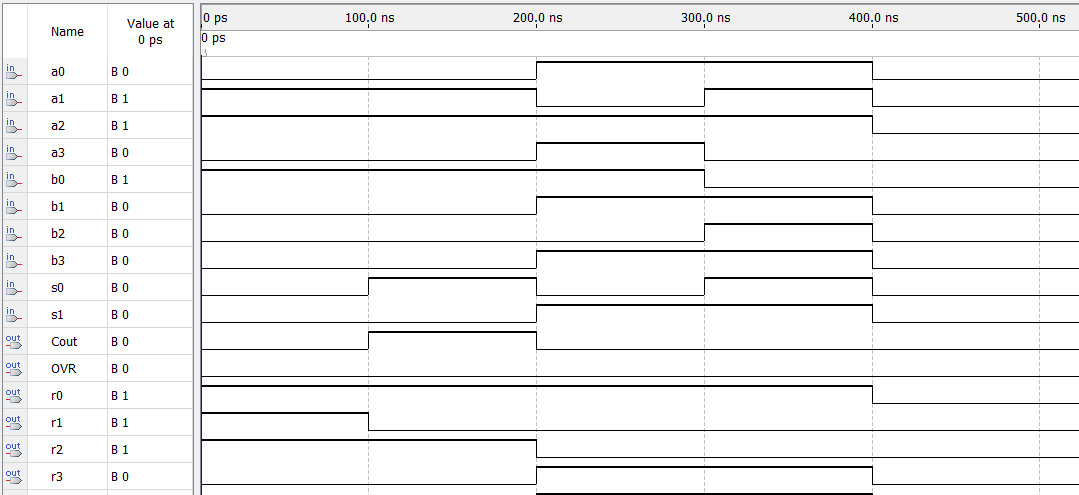


Figure 8.3.1: ALU Waveform

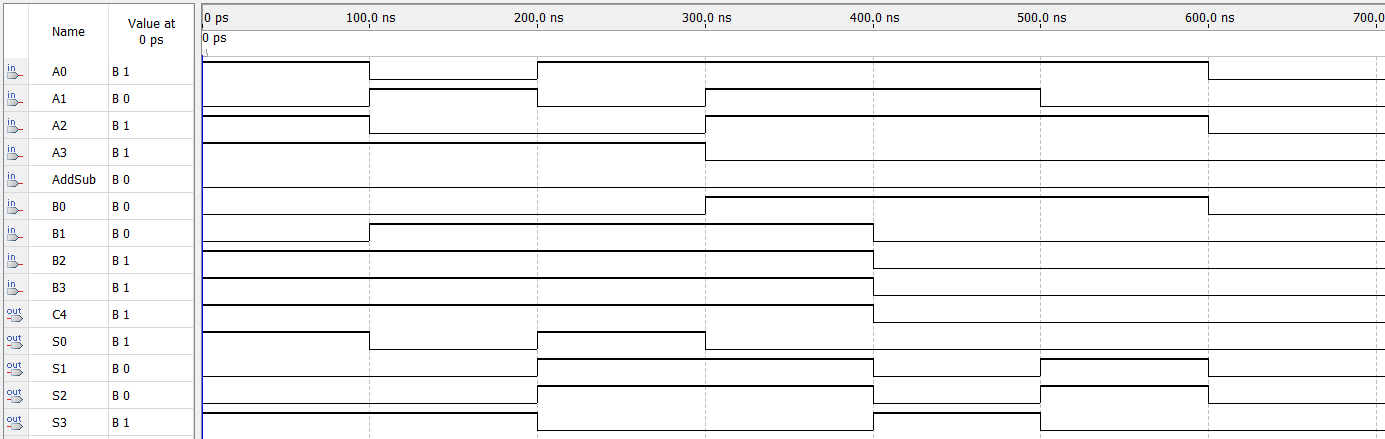


Figure 8.3.2: Adder\_Subtrator

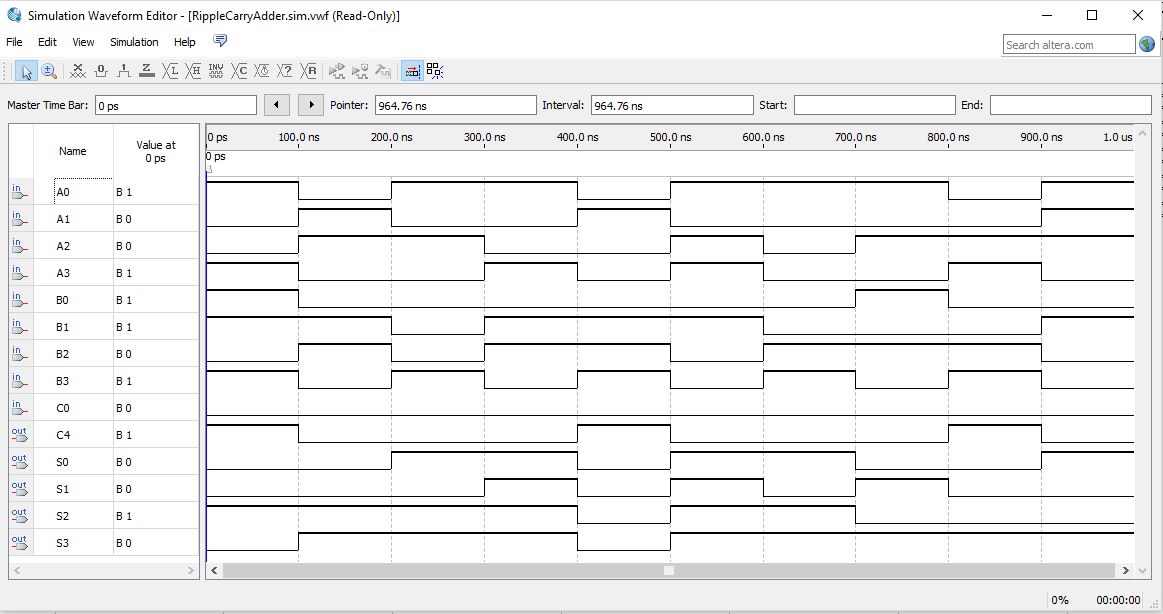


Figure 8.3.3 Ripple Carry Adder

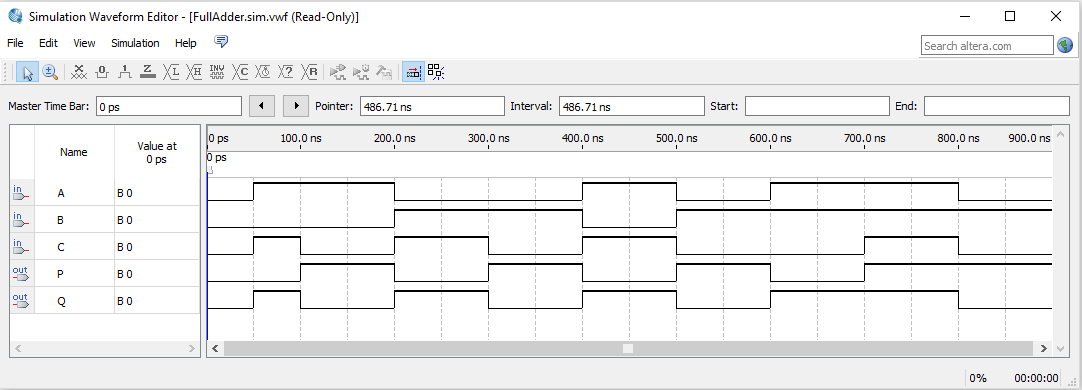


Figure 8.3.4: Full Adder

**8.4 Test Results:** The DE1 achieves stores the desired values in the Accumulator by doing various loads and increments and a Jump (Figure 8.4.1). The ACC correctly goes through the sequence: C, D, E, F, F, 0, F, 0, 6, 2, 3, 3. And then cycles back to the beginning again due to the fact that the halt instruction was not implemented.

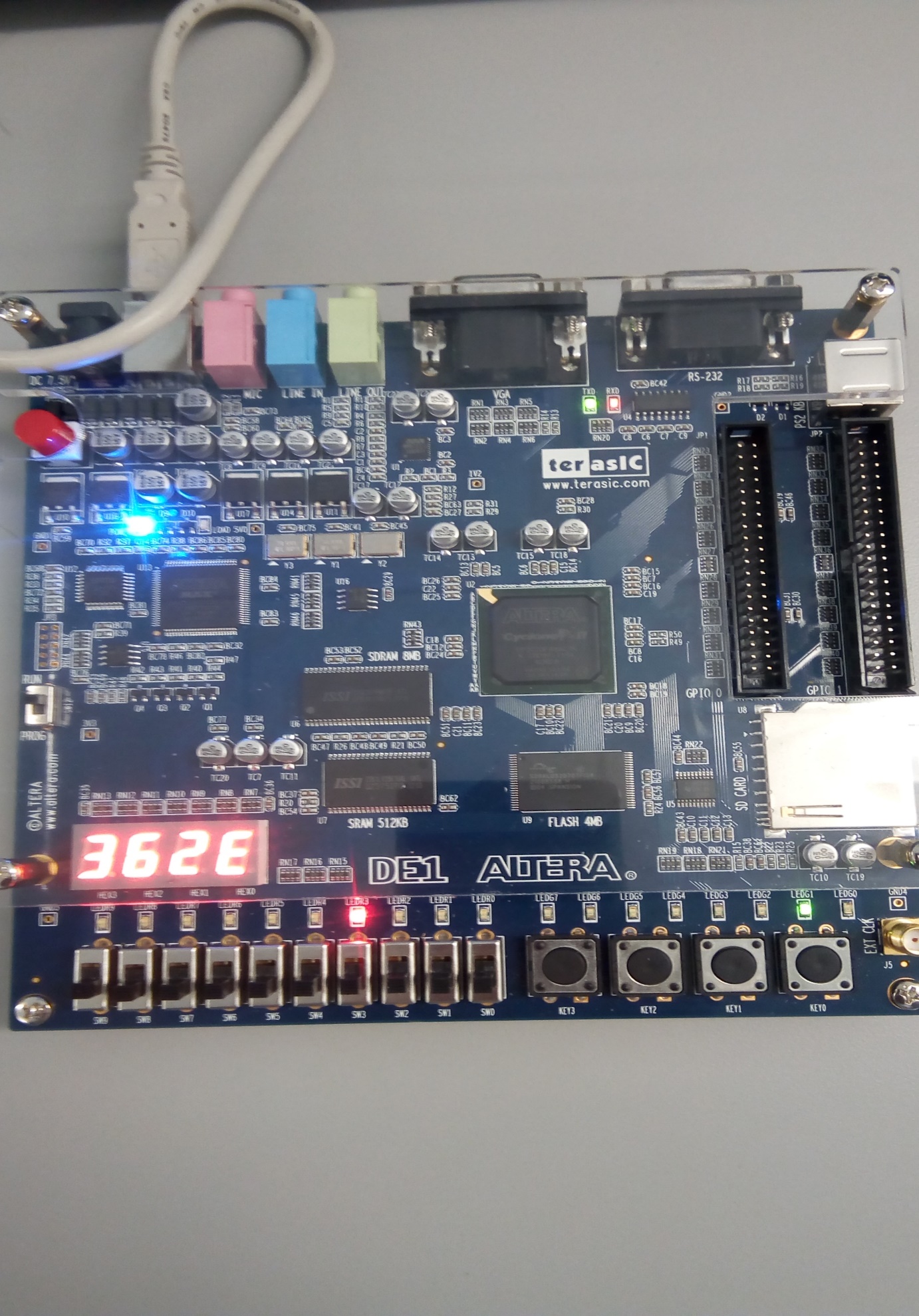


Figure 8.4.1: Altera DE1 executing instructions

**9 Conclusion**

**9.1 Resolutions:** There were some instances where a problem was solved by simply adding outputs to debug and the circuit would suddenly start working as needed. This phenomenon has occurred to many students in the lab and all cannot explain how outputs could change the whole behavior of the circuit. I am still unsure if my configuration would function correctly if constructed physically.

**9.2 Lessons Learned:** I learned that having some type of feedback like the LED’s for debugging is important since more than likely there is an issue somewhere and the LEDs will help find bugs with less hassle. Another good thing I noticed is that it’s better to prepare things to be updated to save time. I changed the Verilog of the controller every time I was working on a new instruction. This made me unnecessarily update the code for the controller as well as replacing the old controller symbols with the new updated ones several times than I needed to. What I should have done was draw out the complete diagram like in Figure 6.1 to set everything in the code and wire it when I am ready.