Andrew Pimental

Dillon Dragomir

**Project 3: Non-Pipelined Control Unit**

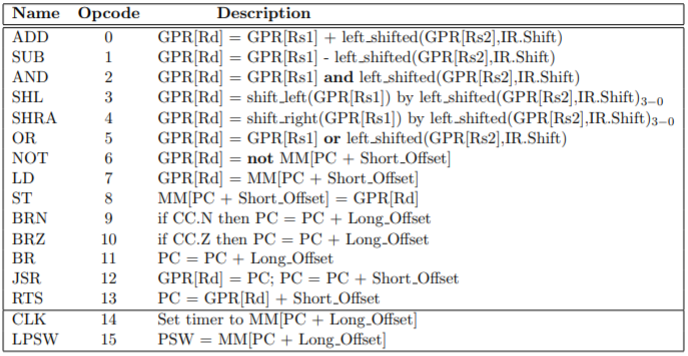
**Introduction:**

In this lab, students are tasked to create a basic control unit within a simple processor. This requires students to implement the concepts of state machine creation, design control signal optimizatimally, and have an understanding of instructions sets. Through the process of defining an instruction set, creating the state machine or state machines, and defining when each input of the instruction set should be applied, this design can be created. Examining each step, helps students to expand on their knowledge of control units and grow in their own design skills.

**Requirements:**

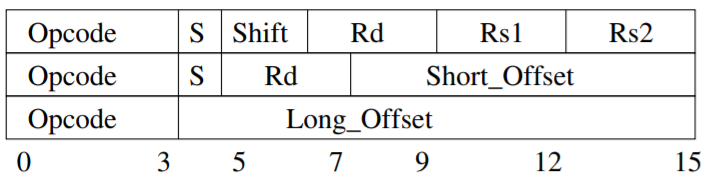
For this project, the following requirements have been given:

* Word Size = 16-bits
* Memory Address and Data Bus Size of 16-bits
* 64 KB Byte Addressable Main Memory
* 16-bit Program Status Word in which the first two bits are Z, N, and Privileged vs User Mode.
* 16 Instructions as followed:



**Figure 1: Instruction Set**

* 8 16-bit General Purpose Registers
* 16-bit Program Counter in GPR([R7])
* 16-bit Count Down Timer
* 2’s Complement Number Representation
* Instruction Formats:



**Figure 2: Instruction Formats**

**Overall State Machine Design:**

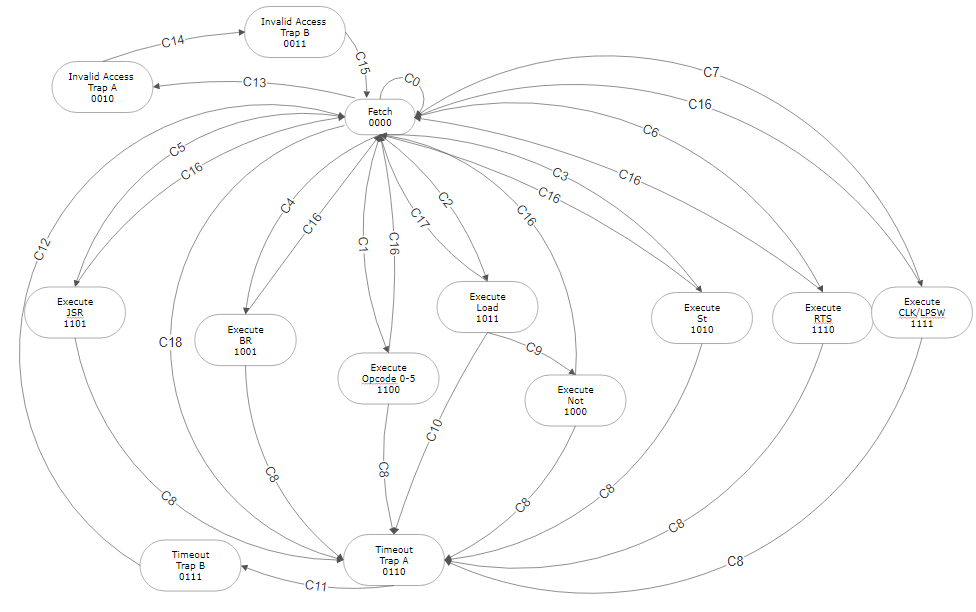
**Overall State Machine:**

With the circuits constructed, the overall design is as follows. The control of the progression of the circuit is contained within two state machines. One state machine controls the overall progression from Fetch, many different Executes, and the various Trap states and the other controls the timing within each states. This machine is shown by Figure 2. With this design, depending on the current state and the conditions presented to the state machine, the state will move in the flow as shown. Each arrow is denoted with a C code which corresponds with the code within Table 1. This state machine will stay in each state until a reset pulse is sent to the second state machine, the numeric state machine. This reset pulse will act as the clock for the overall state machine. This allows the overall state machine to only need to run the number of cycles each state needs. So for example Execute Not 1000 only needs a single cycle to complete its steps. This means after the first cycle, the reset signal will be sent to the numeric state machine and this state machine will iterate to the next state.

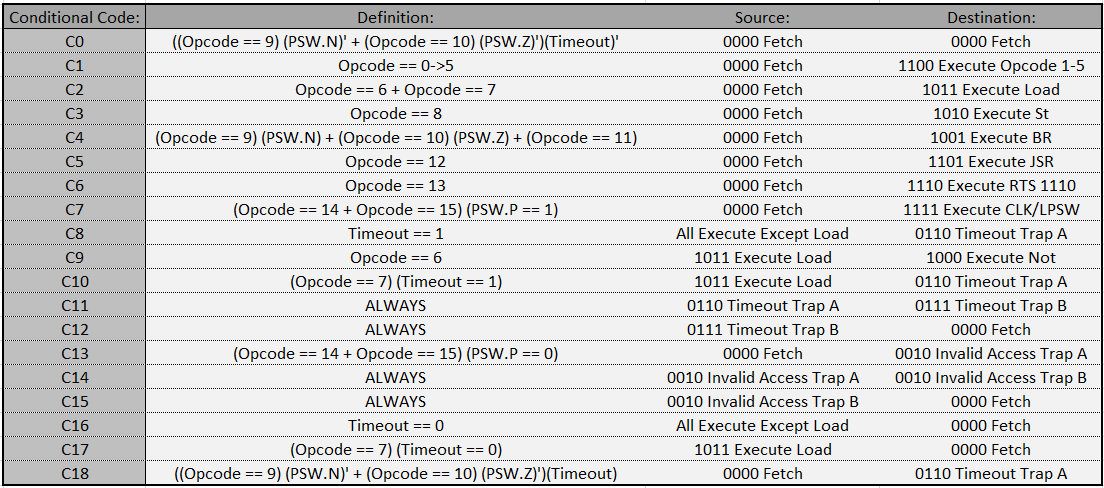
The basic flow of states is as follows. The state machine will start in the Fetch state and depending on the conditions provided to the state machine will enter the corresponding execute state. The exception to this is the branching instructions and the CLK/LPSW. If a branch fails when trying to execute branch on zero and PSW.Z is false or when trying to execute branch on negative and PSW.N is false the state machine will either stay in Fetch or head to the Timeout Trap depending on if Timeout is shown true. The other odd case for fetch is Execute CLK/LPSW. For these cases, if the PSW.P is false and the instruction set wants to run these instructions, the state machine will go to the Invalid Access Trap A.

After each traditional instruction completes their execute step, they will either go into the Timeout Trap A or enter back into Fetch depending on if Timeout is equal to one. This means when Timeout is equal to one, the state machine will enter Timeout Trap A. The exception here is when the opcode is 6 (NOT) and the state machine is in Execute Load. The not instruction runs the Load execution and has one additional cycle needed for its operation, so it needs to complete its instruction before being interrupted by a timeout trap.

The final state transitions to describe are always conditions. For example, when the state machine is at Timeout Trap A or Invalid Access Trap A, it will always enter the B version of each state when the clock ticks. Finally, when the Traps are in the B form of each, the state machine will head back to the Fetch state when the clock ticks.

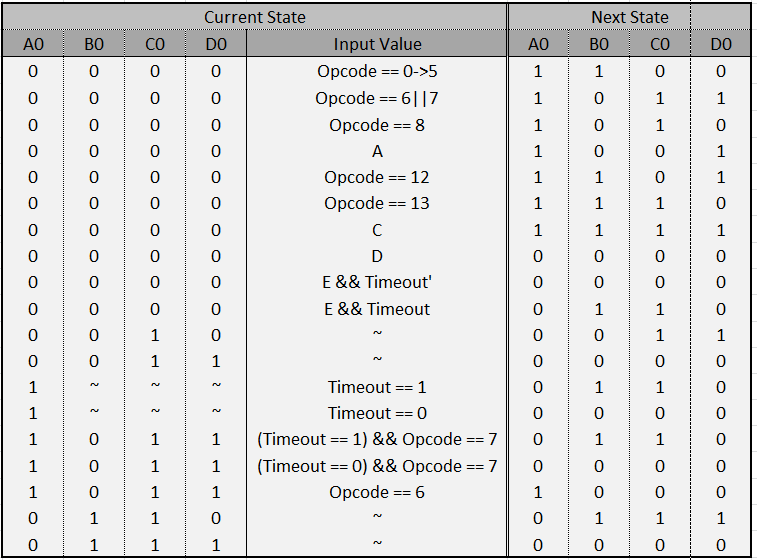


**Figure 2: Overall State Diagram**



**Figure 3: State Machine Conditional Codes**

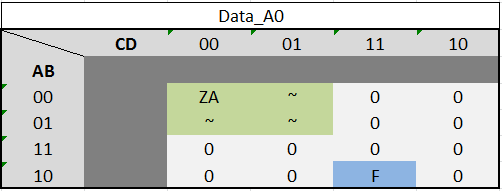
**Finding Conditions for Overall State Machine:**

****

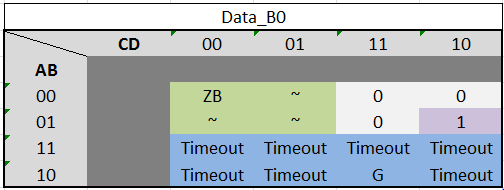
**Figure 4: Overall State Machine State Table**

The next step is to define the conditions of each state and the flow from each state to another. In Figure 4, the various A0, B0, C0, and D0 states are defined and the flow from one set of A0, B0, C0, and D0 state to another. These state variables will then be used to define each latch that is within the Overall State Machine. Also when observing Figure 4, the Input Value column is the conditions that controls branches within each state. For example, if the state machine is in state 0000 and the Opcode is equal to 8, then the next state will be 1010.

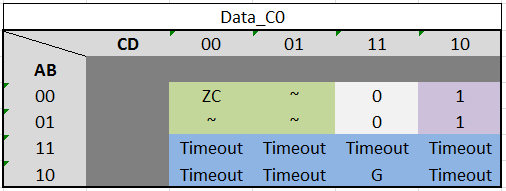
**Overall State Machine K-Maps:**

****

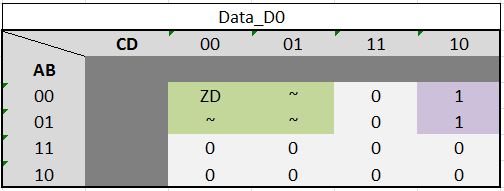
**Figure 5: Data\_A0 K-Map**

****

**Figure 6: Data\_B0 K-Map**

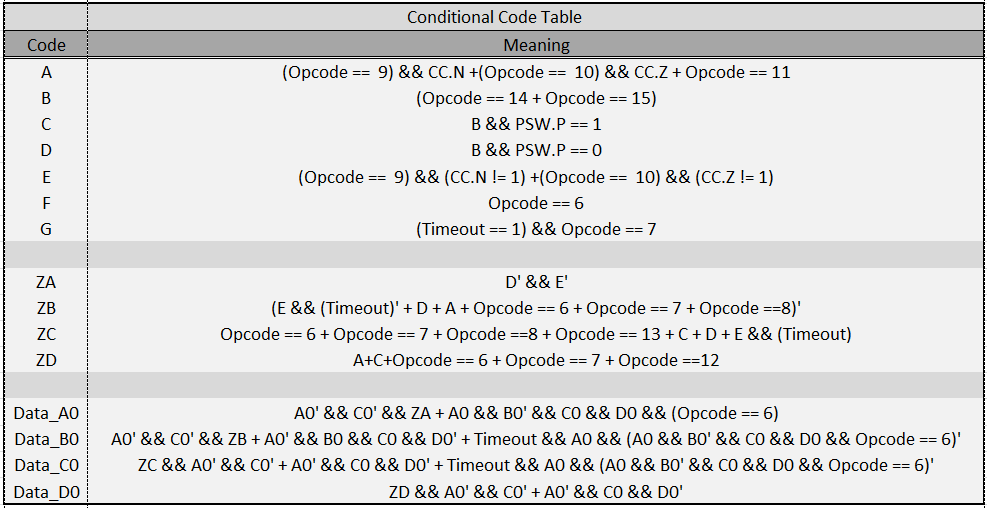
****

**Figure 7: Data\_C0 K-Map**

****

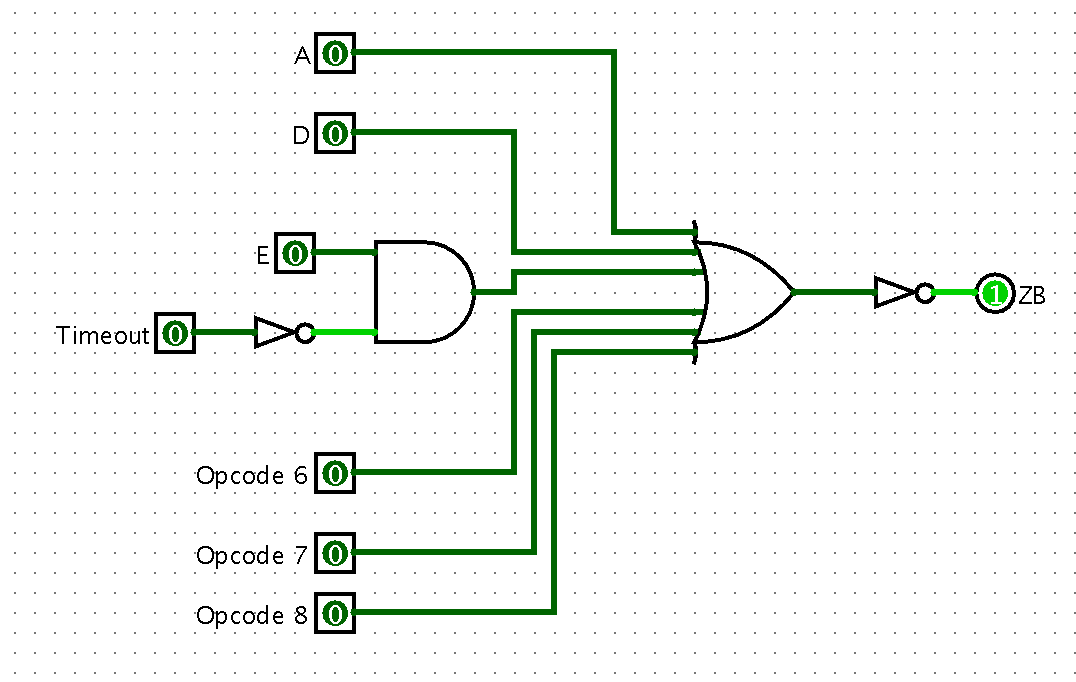
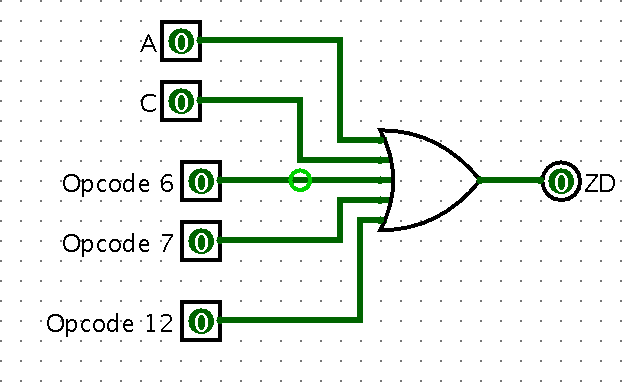
**Figure 8: Data\_D0 K-Map**

From the Overall State Machine State Table in Figure 4, the K-maps in Figures 5-8 can be created. These k-maps will allow the grouping of each conditional set into reduced forms. Now, within the previous figures there were undefined letters such as ZA, ZB, A, B… These conditional variables are shown in Figure 9. Each of the codes can be replaced with their equivalent meaning in the statements the codes are in. The final conditionals for the latches are shown in Data\_A0 ,Data\_B0, Data\_C0, and Data\_D0.

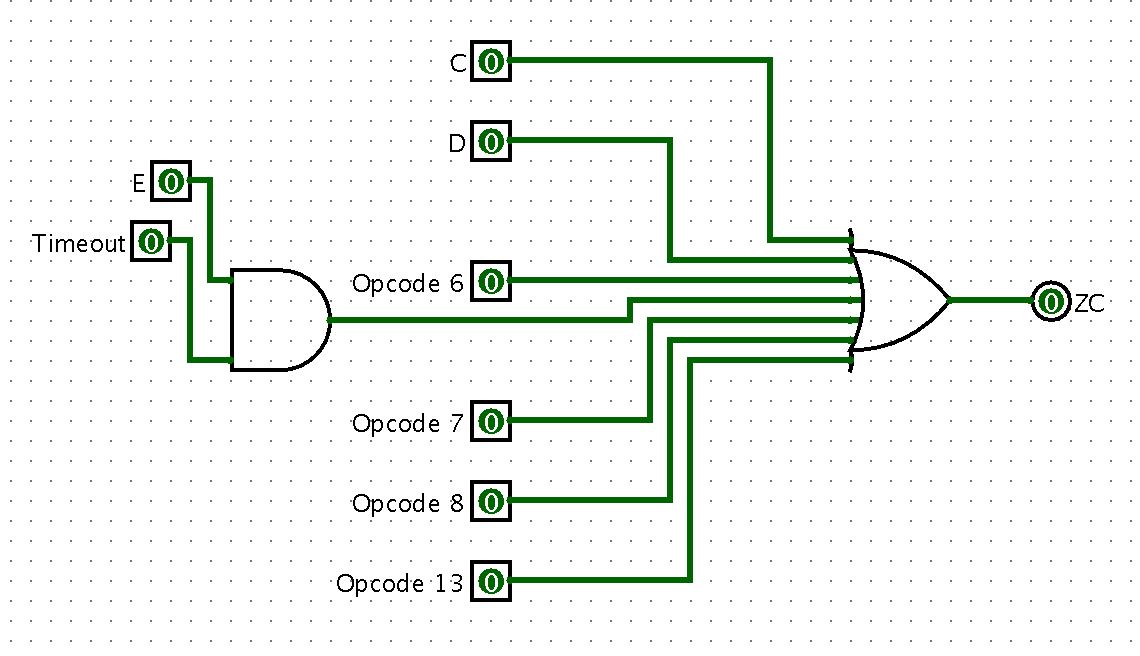
****

**Figure 9: Conditional Code Key**

The following circuits in Figure 10 and 11 are used strictly for defining conditions that are used in the step state machine circuit to decide the next state.

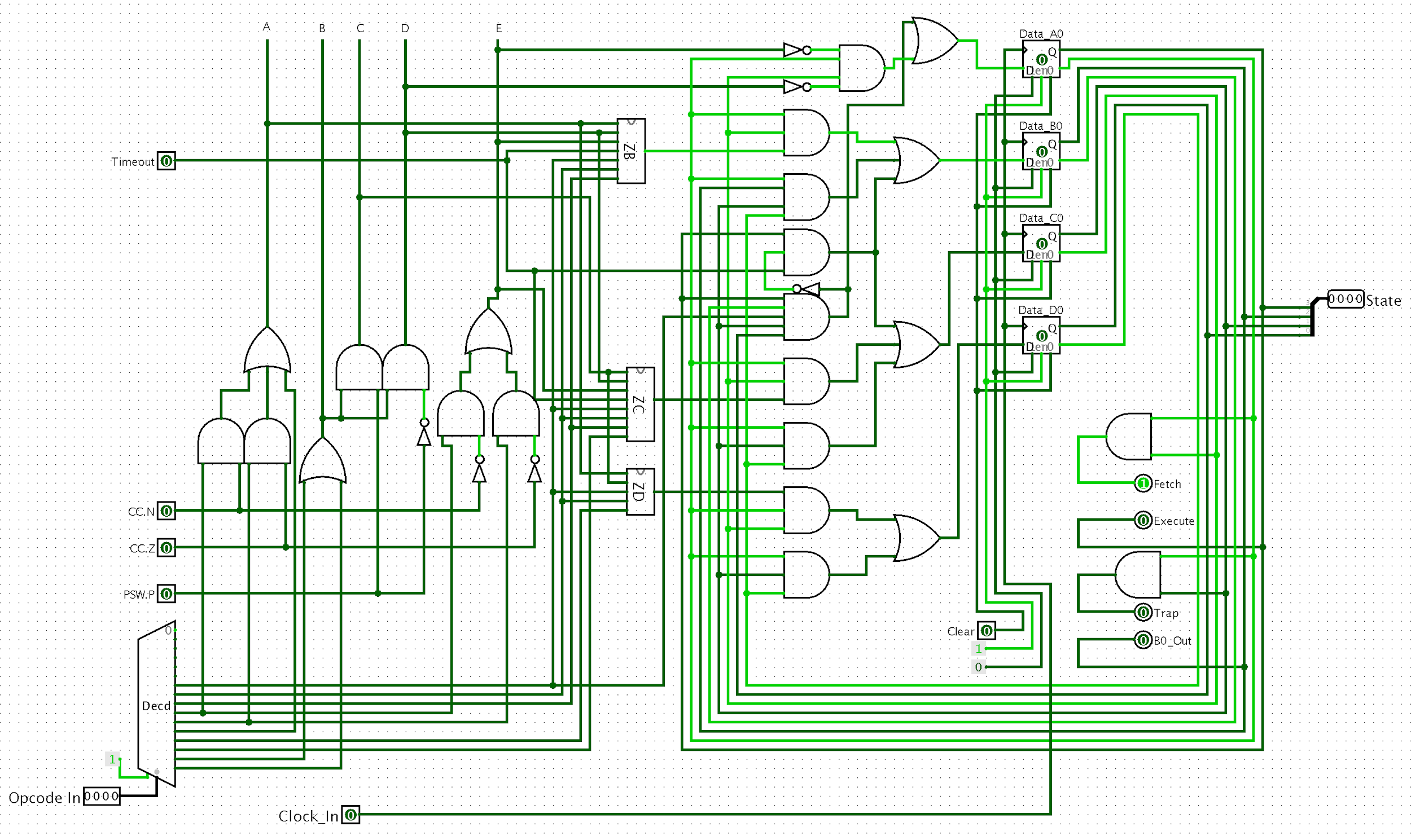
****

**Figure 10: ZD and ZB Step State Machine Conditions**

****

**Figure 11: ZC Step State Machine Conditions**

The following circuit in Figure 12 is the overall state machine circuit implemented. The overall state machine show in the previous figures is implemented at the gate level in this circuit. The state is saved in a set of four d latches and is allowed an option of 16 states. The implementation solution described in this design document takes advantage of thirteen of these states and the remaining states could be leveraged if operations were added to the instruction set.

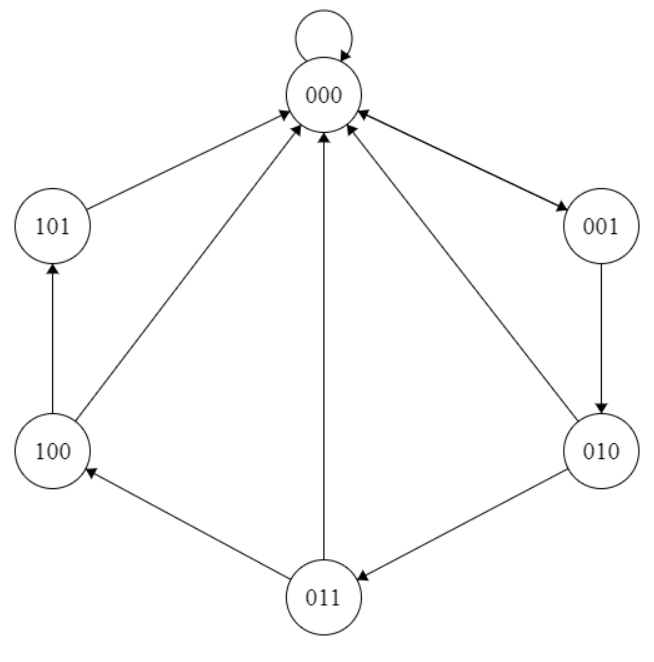
****

**Figure 12: Step State Machine Implemented**

**Numeric State Machine:**

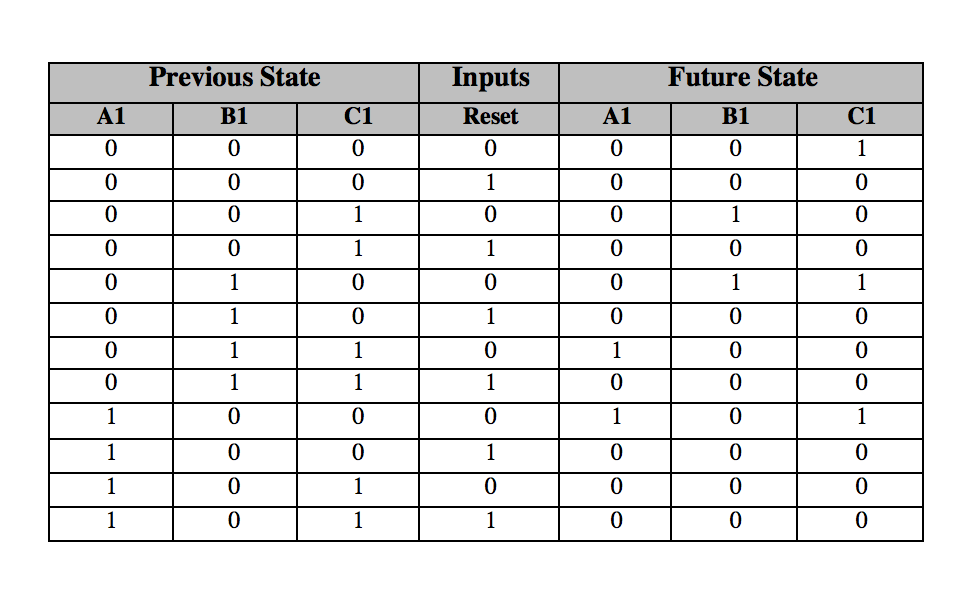
The second state machine controls the steps within each state. The means that this state machine controls each individual step within the states controlled by the Overall State Diagram. So if the state machines describes Fetch step 2, the first state machine will be in 0000, and the second state machine will be in state 010. The state within the Numeric State Diagram is described by the current binary representation of the step number.

When using this Numeric State Machine, if the last step has been hit for this Overall State, the exterior circuit will send a reset signal to both the Numeric State Machine and the Overall State Machine. This will cause the Overall State Machine to iterate as mentioned before and the Numeric State Machine will reset to 000. This is done by resetting all the latches that are contained in the Numeric State Machine.



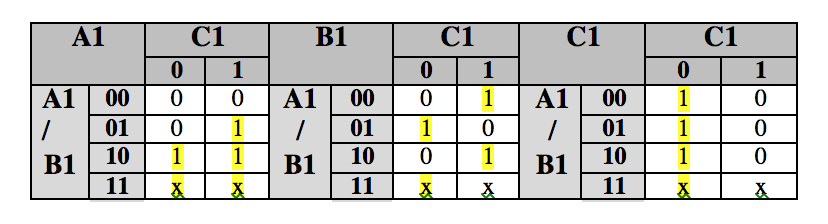
**Figure 13: Numeric State Machine**

The figure shown below represents how the the numeric state machine progresses through until the reset bit is enabled.



**Figure 14: Numeric State Next State Table**

The figure below is the karnaugh map for the numeric state machine and was used to derive the boolean expressions that were implemented in Logisim.



**Figure 15: Numeric State Karnaugh Maps**

**Boolean Expressions**

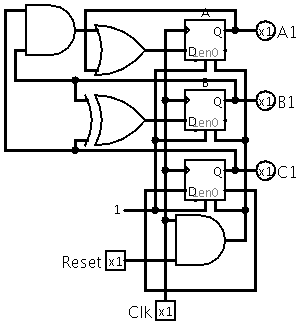
The following boolean expressions were implemented in the numeric state machine to calculate to generate the step signals.

**A1(Next) = A1 + B1 \* C1**

**B1(Next) = B1 XOR C1**

**C1(Next) = C1’**

Figure 16 shows the implementation of the numeric state machine in Logisim. This three bit machine counts up and when the reset bit is enabled the numeric state machine is reset to zero on the clock signal. Having the reset bit allows the each state to be a different number of cycles and helps improve the efficiency.



**Figure 16: Logisim Numeric State Circuit**

**Instruction Set:**

For each state shown in FIgure 2, a set of signals needs to be developed. Each state’s raw signals are shown below:

**Raw Signals:**

1. Fetch
   1. GPR[R7]out, MARin , Read\_mm, ALU\_Inc\_Left, Zin
   2. WMFC, GPR[R7]in
   3. MDRout , IRin , Reset\_State
2. Execute Operand 0->5
   1. IR.Shiftout ,yint
   2. GPR[IR.RS2]out , ALU\_Left\_Shift, Zin
   3. Zout , yin
   4. GRP[IR.RS1]out , (ALU OPERATION), Zin
   5. Zout , GPR[IR.Rd]in , Set\_CC, Reset\_State
3. Execute Not
   1. GPR[R7]out, yin
   2. IR.Short\_Offsetout, ALU\_Add, Zin
   3. Zout, MARin, Read\_mm,
   4. WMFC
   5. MDRout , ALU\_Not\_Left,,Zin
   6. Zout , GPR[IR.Rd]in , Set\_CC, Reset\_State
4. Execute Load
   1. GPR[R7]out, yin
   2. IR.Short\_Offsetout, ALU\_Add, Zin
   3. Zout, MARin, Read\_mm,
   4. WMFC
   5. MDRout , GPR[IR.Rd]in , Set\_CC, Reset\_State
5. Execute Store
   1. GPR[R7]out, yin
   2. IR.Short\_Offsetout, ALU\_Add, Zin
   3. Zout, MARin
   4. GPR[IR.Rd]out , MDRin , Write\_mm
   5. WMFC
6. Execute Branch
   1. GPR[R7]out, yin
   2. IR.Long\_Offsetout, ALU\_Add, Zin
   3. Zout, GPR[R7]in
7. Execute JSR
   1. GPR[R7]out, yin , GPR[IR.Rd]in
   2. IR.Short\_Offsetout, ALU\_Add, Zin
   3. Zout, GPR[R7]in
8. Execute RTS
   1. GPR[IR.Rd]out, yin
   2. IR.Short\_Offsetout, ALU\_Add, Zin
   3. Zout, GPR[R7]in
9. Execute CLK
   1. GPR[R7]out, yin
   2. IR.Long\_Offsetout, ALU\_Add, Zin
   3. MARin, Zout, Read\_mm,
   4. WMFC
   5. TMRin, MDRout
10. Execute LPSW
    1. GPR[R7]out, yin
    2. IR.Long\_Offsetout, ALU\_Add, Zin
    3. MARin, Zout, Read\_mm,
    4. WMFC
    5. PSWIn, MDRout
11. Invalid Access Trap
    1. Constant\_0\_out, MARin
    2. PSWout, MDRin, Write\_mm
    3. WMFC
    4. Constant\_2\_out, MARin
    5. GPR[R7]out, MDRin, Write\_mm
    6. WMFC
    7. Constant\_4\_out, MARin, Read\_mm
    8. PSWint, MDRout
    9. WMFC
    10. Constant\_6\_out, MARin, Read\_mm
    11. GPR[R7]out, MDRout
    12. WMFC
12. Timeout Trap
    1. Constant\_8\_out, MARin
    2. PSWout, MDRin, Write\_mm
    3. WMFC
    4. Constant\_10\_out, MARin
    5. GPR[R7]out, MDRin, Write\_mm
    6. WMFC
    7. Constant\_12\_out, MARin, Read\_mm
    8. WMFC
    9. PSWin, MDRout
    10. Constant\_14\_out, MARin, Read\_mm
    11. WMFC
    12. GPR[R7]in, MDRout

**Reduced Signals:**

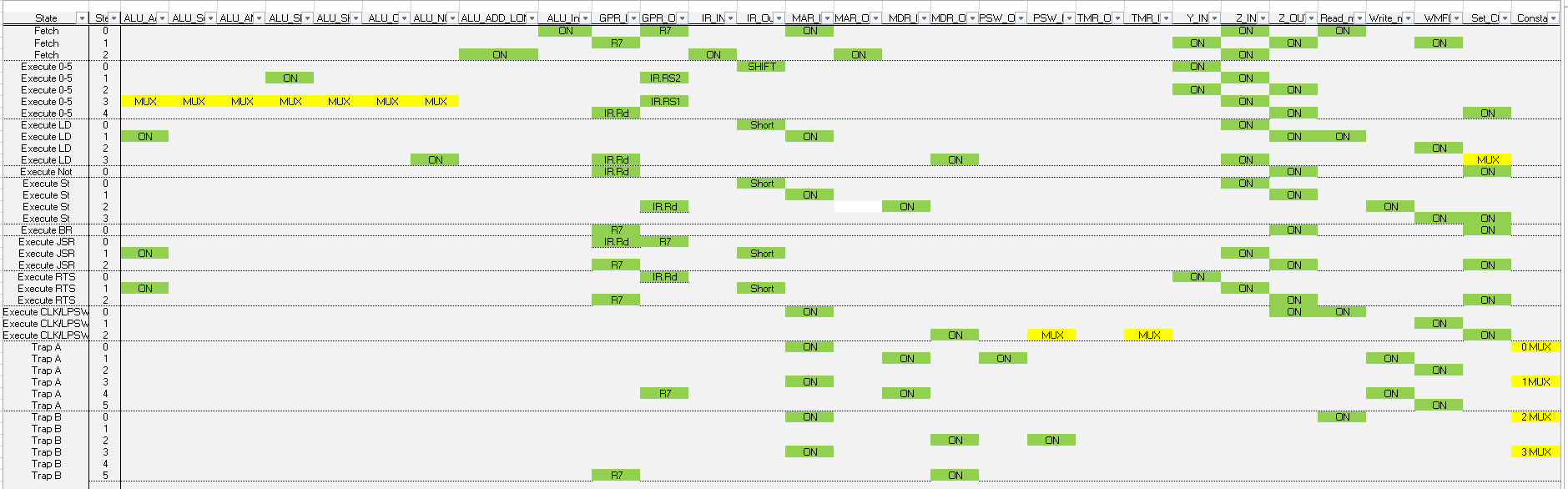
1. Fetch
   1. GPR[R7]out, MARin , Read\_mm, ALU\_Inc\_Left, Zin
   2. WMFC, Zout , yin , GPR[R7]in
   3. MDRout , IRin , Reset\_State, ALU\_Add\_LongOffset, Zin
2. Execute Operand 0->5
   1. IR.Shiftout ,yint
   2. GPR[IR.RS2]out , ALU\_Left\_Shift, Zin
   3. Zout , yin
   4. GRP[IR.RS1]out , (ALU OPERATION), Zin
   5. Zout , GPR[IR.Rd]in , Set\_CC, Reset\_State
3. Execute Not (Execute Load First)
   1. ~~GPR[R7]~~~~out~~~~, y~~~~in~~
   2. ~~IR.Short\_Offset~~~~out~~~~, ALU\_Add, Z~~~~in~~
   3. ~~Z~~~~out~~~~, MAR~~~~in~~~~, Read\_mm,~~
   4. ~~WMFC~~
   5. ~~MDR~~~~out~~ ~~, ALU\_Not\_Left,,Z~~~~in~~
   6. Zout , GPR[IR.Rd]in , Set\_CC, Reset\_State
4. Execute Load
   1. ~~GPR[R7]~~~~out~~~~, y~~~~in~~
   2. IR.Short\_Offsetout, ALU\_Add, Zin
   3. Zout, MARin, Read\_mm,
   4. WMFC
   5. MDRout , GPR[IR.Rd]in , Set\_CC, Reset\_State
5. Execute Store
   1. ~~GPR[R7]~~~~out~~~~, y~~~~in~~
   2. IR.Short\_Offsetout, ALU\_Add, Zin
   3. Zout, MARin
   4. GPR[IR.Rd]out , MDRin , Write\_mm
   5. WMFC
6. Execute Branch
   1. ~~GPR[R7]~~~~out~~~~, y~~~~in~~
   2. ~~IR.Long\_Offset~~~~out~~~~, ALU\_Add, Z~~~~in~~
   3. Zout, GPR[R7]in
7. Execute JSR
   1. GPR[R7]out, ~~y~~~~in~~, GPR[IR.Rd]in
   2. IR.Short\_Offsetout, ALU\_Add, Zin
   3. Zout, GPR[R7]in
8. Execute RTS
   1. GPR[IR.Rd]out, yin
   2. IR.Short\_Offsetout, ALU\_Add, Zin
   3. Zout, GPR[R7]in
9. Execute CLK/LPSW
   1. ~~GPR[R7]~~~~out~~~~, y~~~~in~~
   2. ~~IR.Long\_Offset~~~~out~~~~, ALU\_Add, Z~~~~in~~
   3. MARin, Zout, Read\_mm,
   4. WMFC
   5. PSWIn/TMRin, MDRout

\*For Execute CLK/ Execute LPSW, the location will be determined based off the opcode the IR shows. Other than this, the steps are identical

1. Invalid Access Trap/Timeout Trap
   1. Constant\_Number\_0, MARin
   2. PSWout, MDRin, Write\_mm
   3. WMFC
   4. Constant\_Number\_1, MARin
   5. GPR[R7]out, MDRin, Write\_mm
   6. WMFC
   7. Constant\_Number\_2, MARin, Read\_mm
   8. WMFC
   9. PSWin, MDRout
   10. Constant\_Number\_3, MARin, Read\_mm
   11. WMFC
   12. GPR[R7]out, MDRout

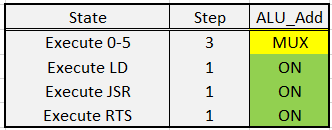
\*For Invalid Access Trap/Timeout Trap, the specific constants will be determined based off which state the state machine is in. Other than this, the steps are identical

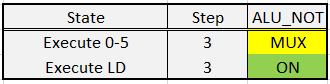
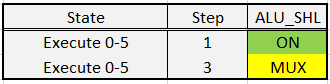
**Conditionals For Signals:**



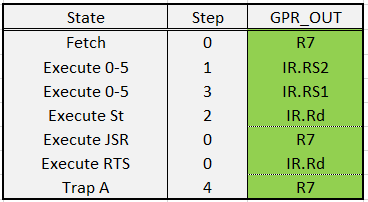
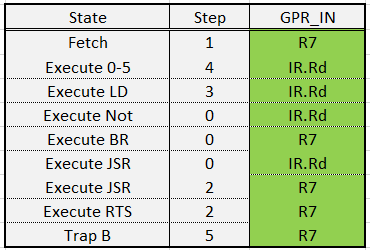
**Figure 17: Chart of Inputs and Outputs**

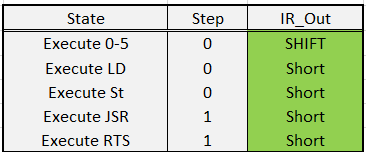
After the steps are configured, each operation can then be simplified to a set of condition statements based off of when each input or output needs to be triggered. Figure 17 shows the chart containing each inputs and outputs signal and when they need to be open. This chart then be used to create the conditions shown below.

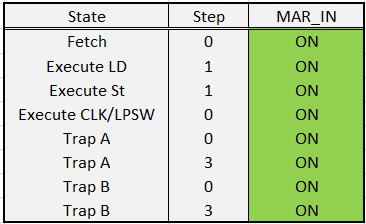


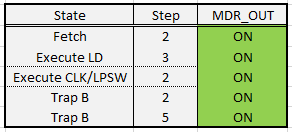
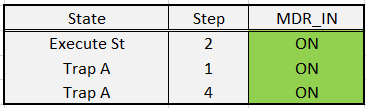


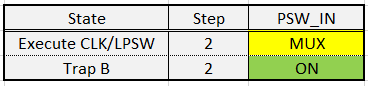


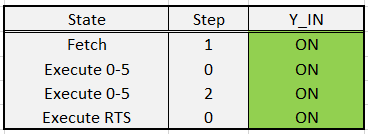


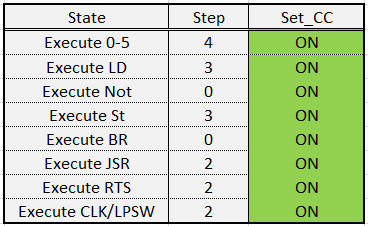
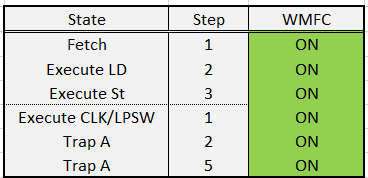
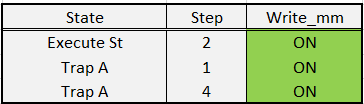
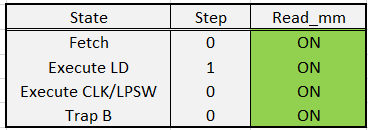
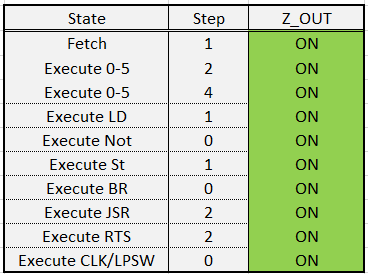
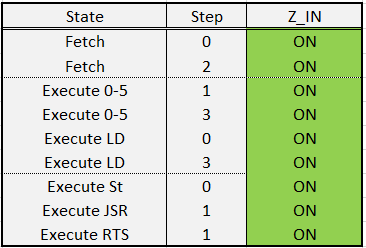


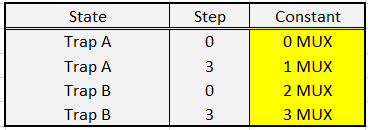












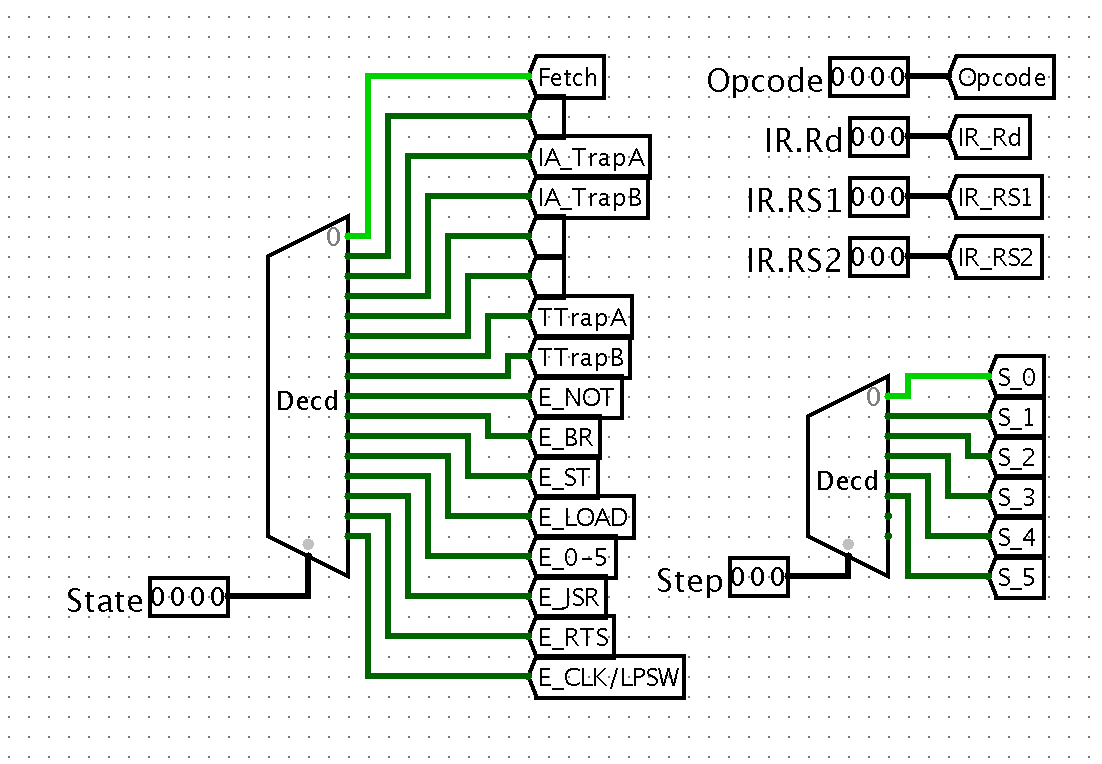
**Figure 18: Set of Conditions For Individual Steps**

These tables of when a signal is on is then used to compare the current state of the Overall State Machine and the Numeric State Machine to determine if a particular signal should be on or off. It should be noted that if a signal is denoted with MUX, this means a MUX is used to determine whether the signal should be a certain value. These are listed below.

* Execute Opcode 0->5 Step 3 MUX: A MUX is used to compare to the opcode to determine whether the signal is adding, subtracting, AND, shift left, shift right, or OR.
* Execute CLK/LPSW Step 2 MUX: The destination of the signal is determined by comparing the opcode.
* Trap MUX: Depending on the type of trap, different constants are passed into the MAR. This is controlled by observing the B0 Latch.

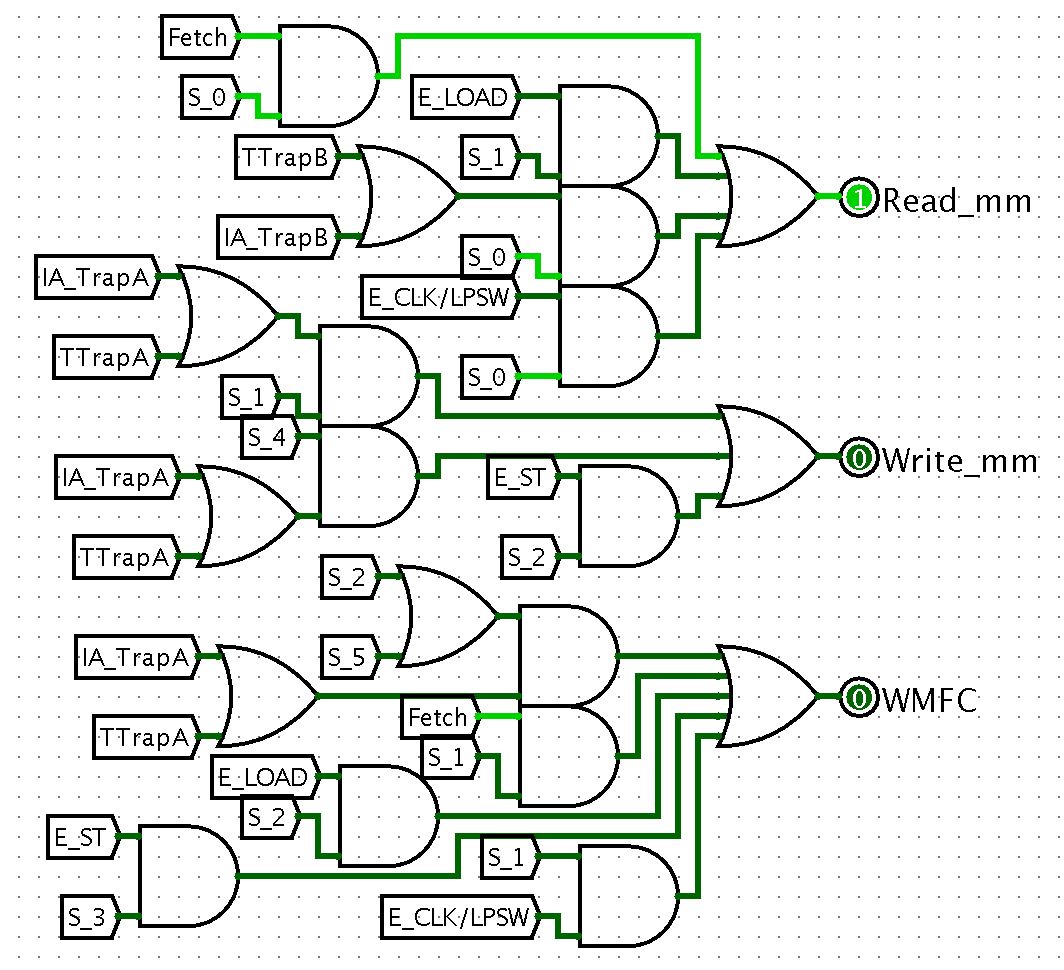
**Signal Generator**

The final step of this process is to compile the tables into a set of AND’s and OR’s that will check that the state and step match at least one condition for the signal to be true. The IR, State and Step machine are leveraged to determine the signals that should be generated and the following circuit implementation is show.



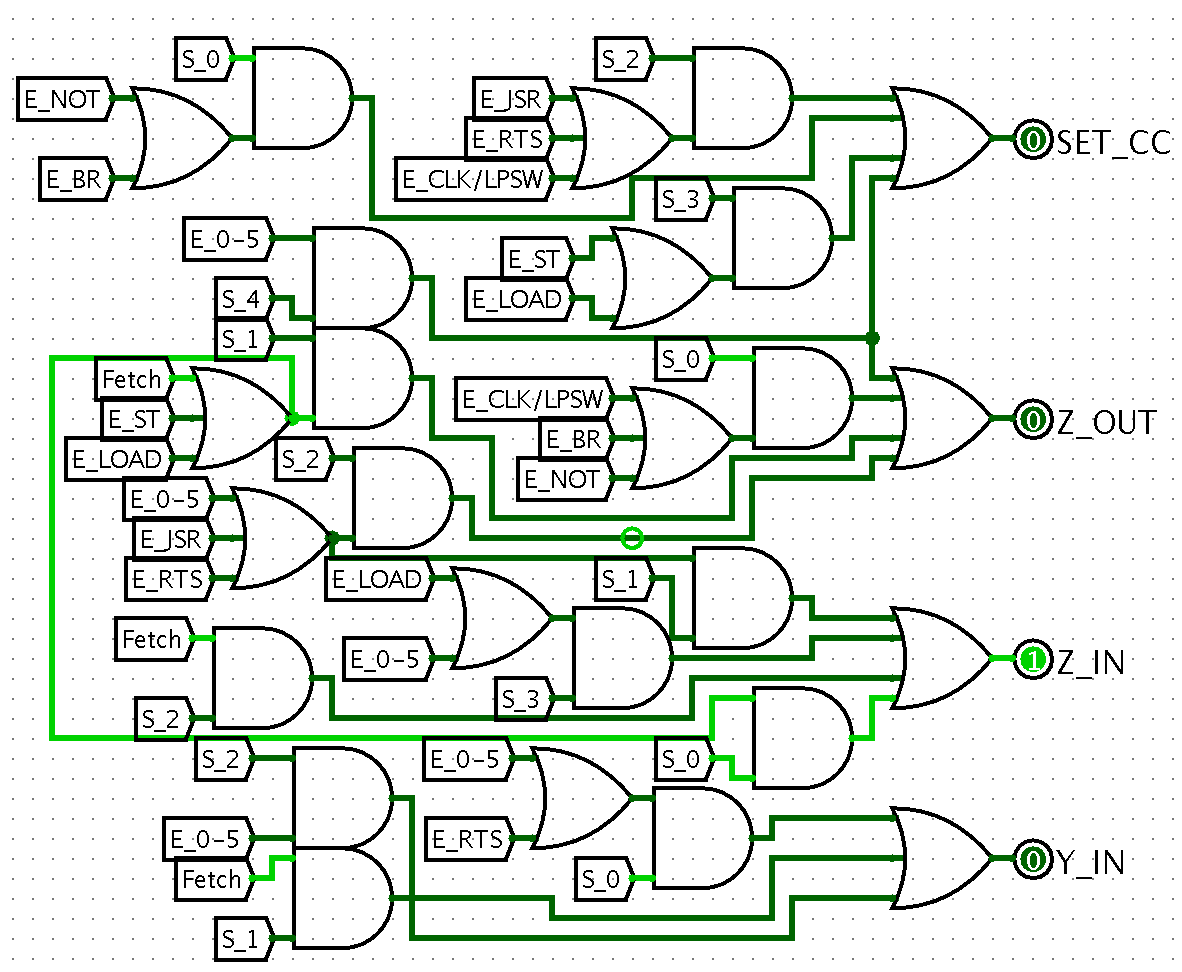
**Figure 19.1: Inputs for Signal Generation**

The following figure shows the related inputs and the memory management related signals that are generated. The tables show in the figures above for the inputs were used when calculating the basic gate level implementation. Since the implementations are fairly straightforward and not complex the boolean expressions will be excluded for brevity purposes.



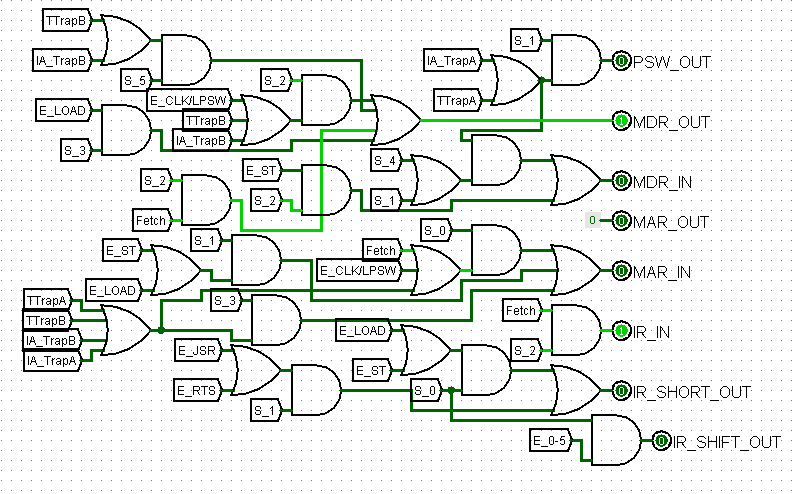
**Figure 19.2: Memory Read and Write Signal Implementation**

The following circuit implementation shows the related signals to load data into the Y register as well as load data into the Z register when an alu operation is performed. Additionally, the Set\_CC bit is used in the PSW implementation to determine when to set the conditional code inside the PSW. The PSW register has a specific implementation that allows for the conditional code to be set inside the register.

****

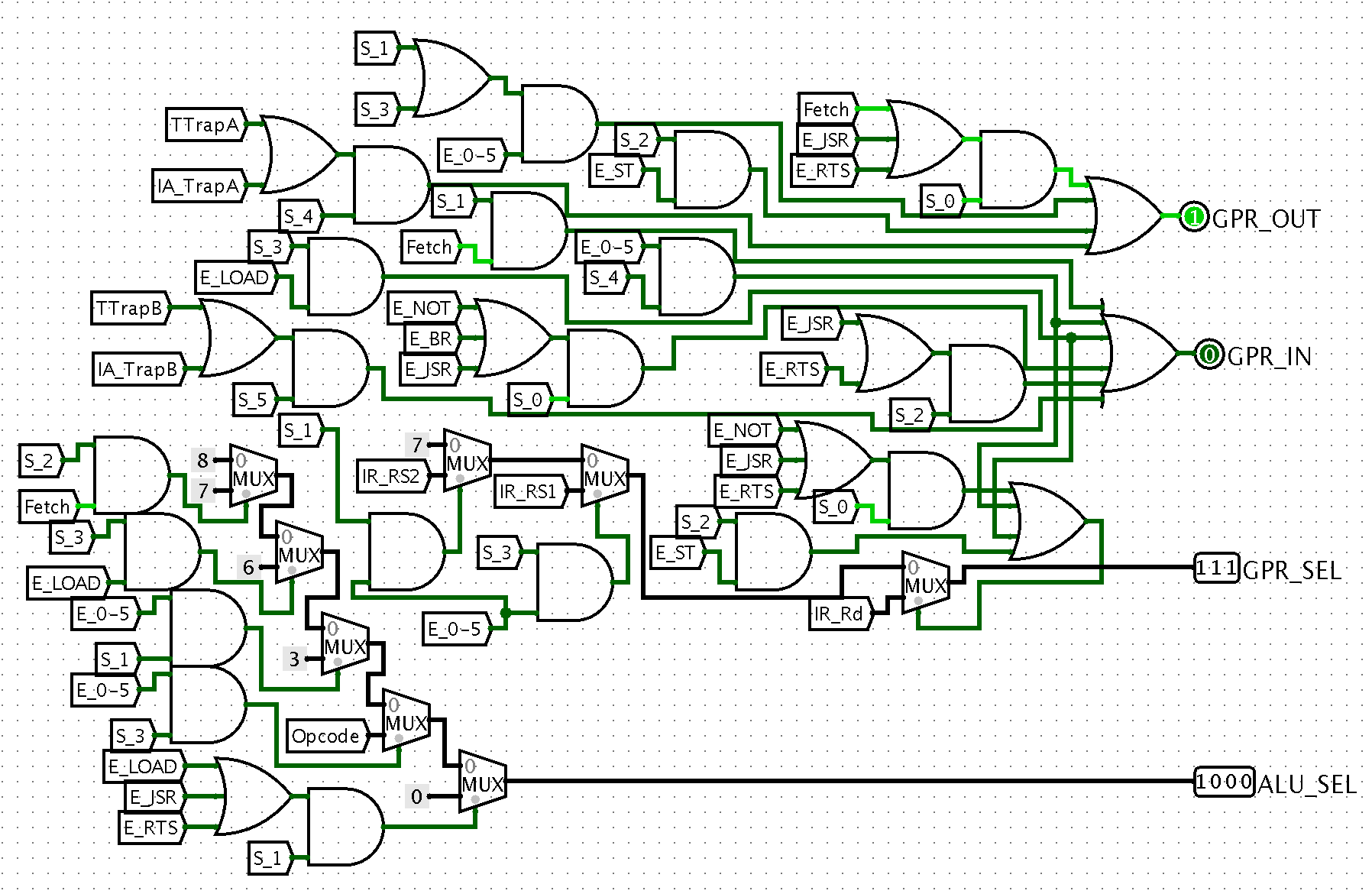
**Figure 19.3: SET\_CC, Y and Z Register Signal Implementation**

The MAR and MDR are memory related signals and are used to load the memory address from the bus and load data from the MM respectively. The PSW\_OUT signal is a case where the PSW is loaded to the bus and the instance when this occurs is represented in the following circuit implementation and happens when Trap A is on its second step. Finally, the instruction register signals load the different decode IR signals onto the bus with extended bits or load the IR with data from the bus.

****

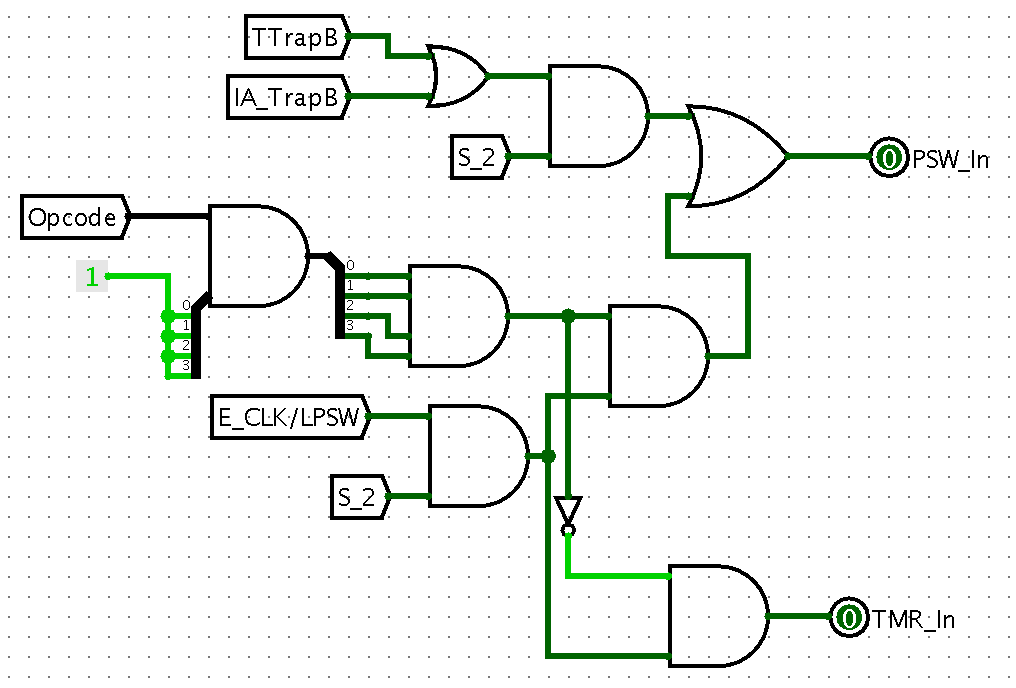
**Figure 19.4: PSW MDR MAR and IR Signal Implementation**

The circuit shown below is used to load the general purpose registers and the selection bit is used to determine what register the data is being loaded into or being fetched from. Finally, the ALU does not have an enable because is is always constantly outputting data, but is only loaded into the Y register when the Y\_IN bit is enabled. The ALU\_SEL goes to a multiplexer that gets the value of the given operation for the ALU that is defined in the following design document.

****

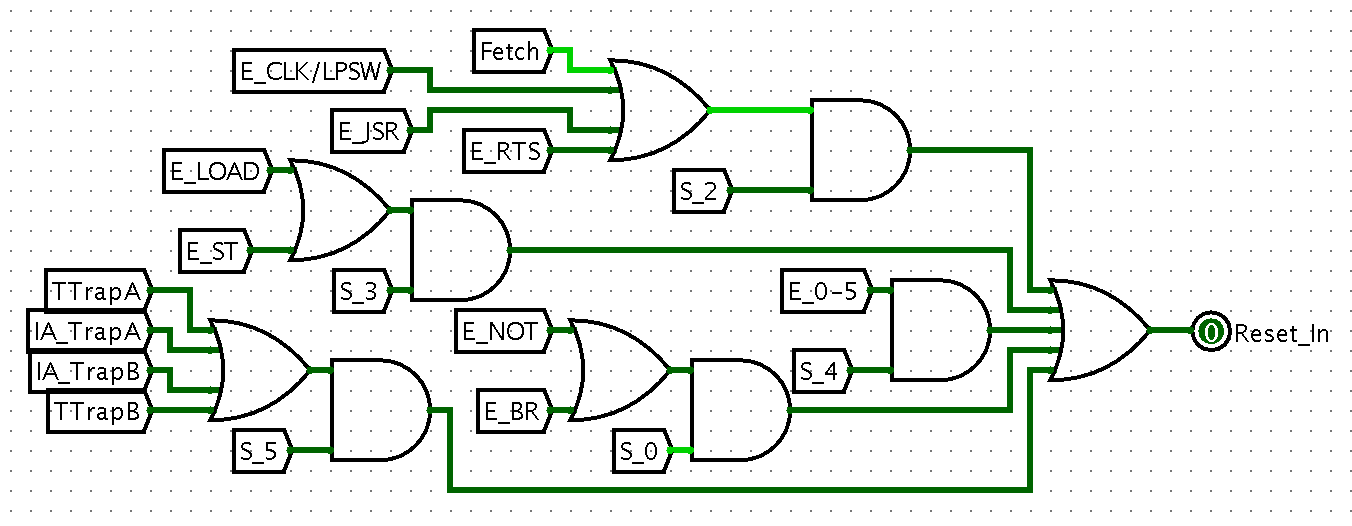
**Figure 19.5: GPR and ALU Signal Implementation**

The circuit shown in the following circuit is used to enable the PSW load and also for the TMR.

****

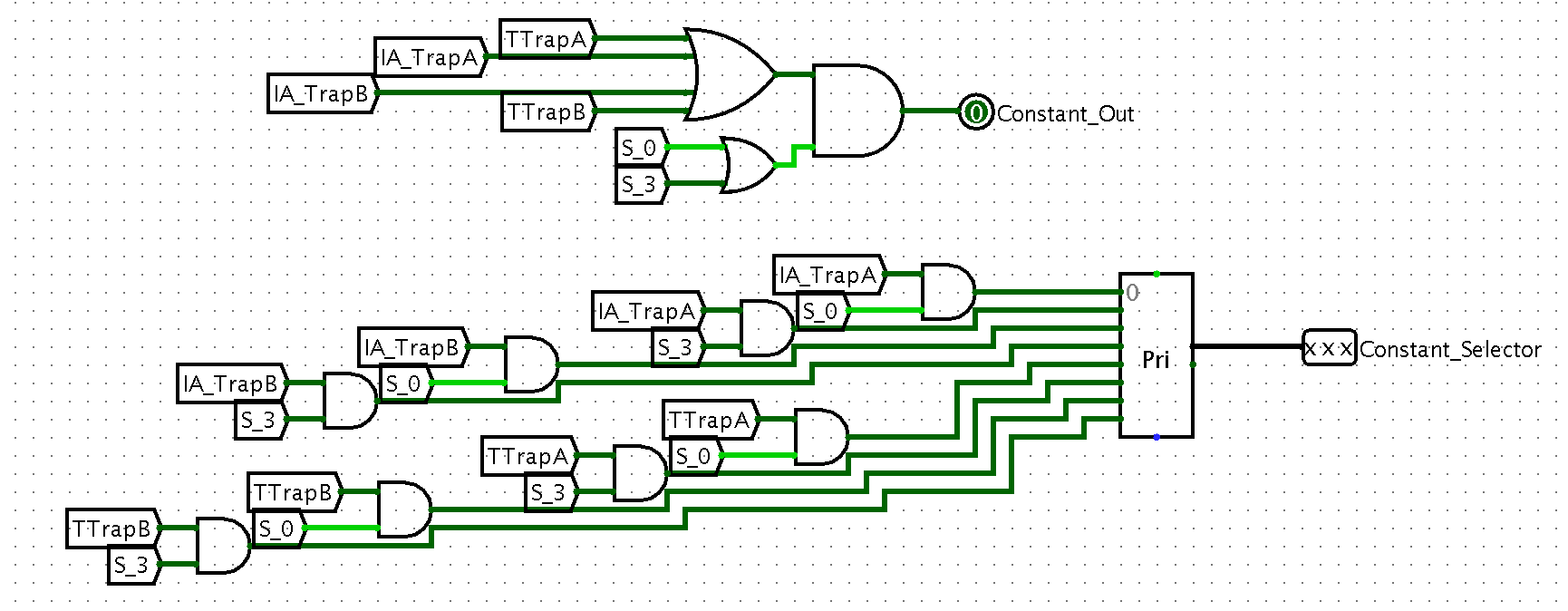
**Figure 19.6: Timer and PSW\_IN Signal Implementation**

The reset bit is one of the most important bits to understand in the following design because it drives much of what happens inside of the state machines. The reset bit is enabled at the end of the state logic and resets the numeric state machine back to zero. The reset bit is also used to trigger the progression of the step state machine to transition to the next state.

****

**Figure 19.7: Reset Signal Implementation**

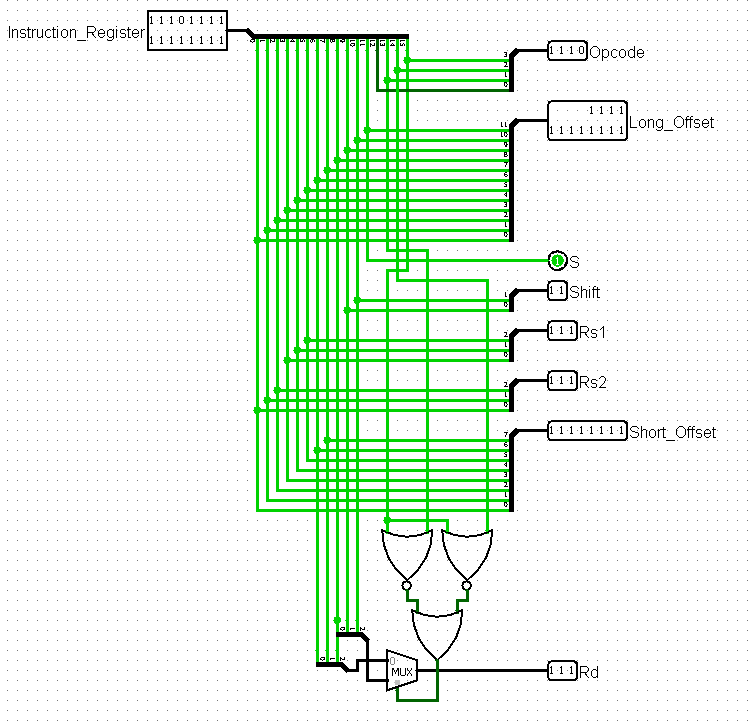
The constant selector is used to choose a constant to put on the bus and the Constant\_Out bit is used to load the given constant chosen by the constant selector onto the bus.

****

**Figure 19.8: Constant Selector Signal Implementation**

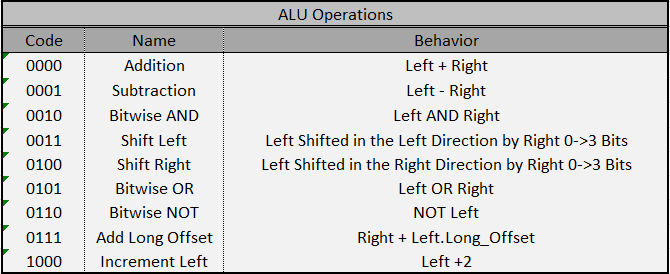
**IR Decoder:**

The instruction register decoder takes the information from the IR and decodes the instruction format into separate codes to be accessed elsewhere. The IR decoder assumes that the machine only accesses the outputs that are desired for a given opcode. Finally, since Rd is not always in the same location, a multiplexer uses the opcode to decide which location to read for Rd. If the bits output is put onto the bus its bits are extended in the main circuit. For the inputs that are used directly in the signal generator the bits are not extended.



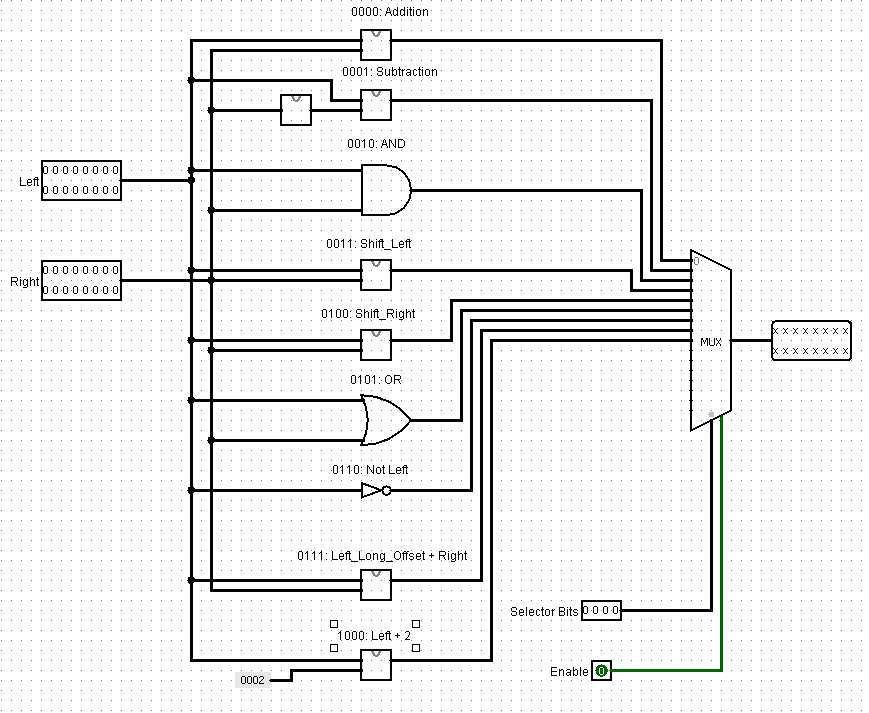
**Figure 20: IR Decoder**

**ALU Design:**

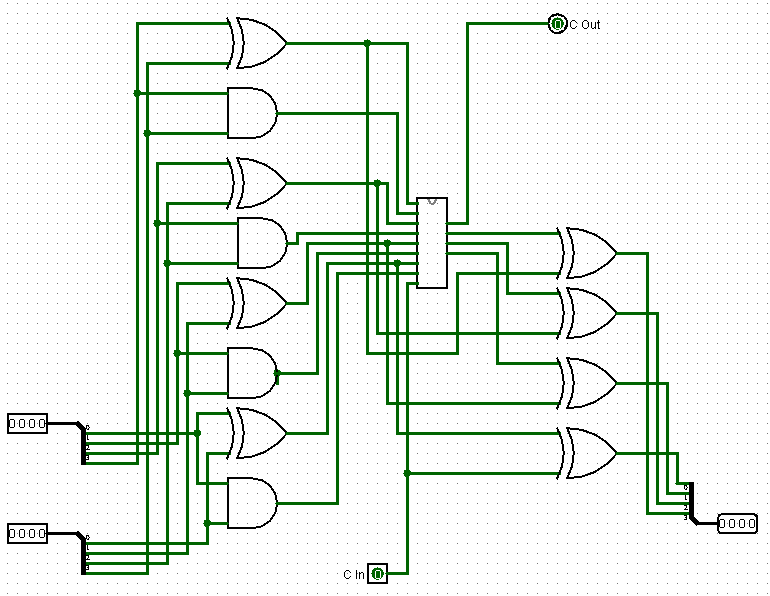
****

**FIgure 21: ALU Operations**

In this project, the ALU needs to match the following operations shown in Figure 21. Each operations is needed in a particular step within the above reduced signals. The overall circuit is shown in Figure 22. You can see each segmented operation is calculated in a different module and a multiplexer decides which operation’s solution to present to the output of the ALU. Also, each module is labeled with the its operation and the associated code. The code is used as the selector bits for the aforementioned multiplexer.

****

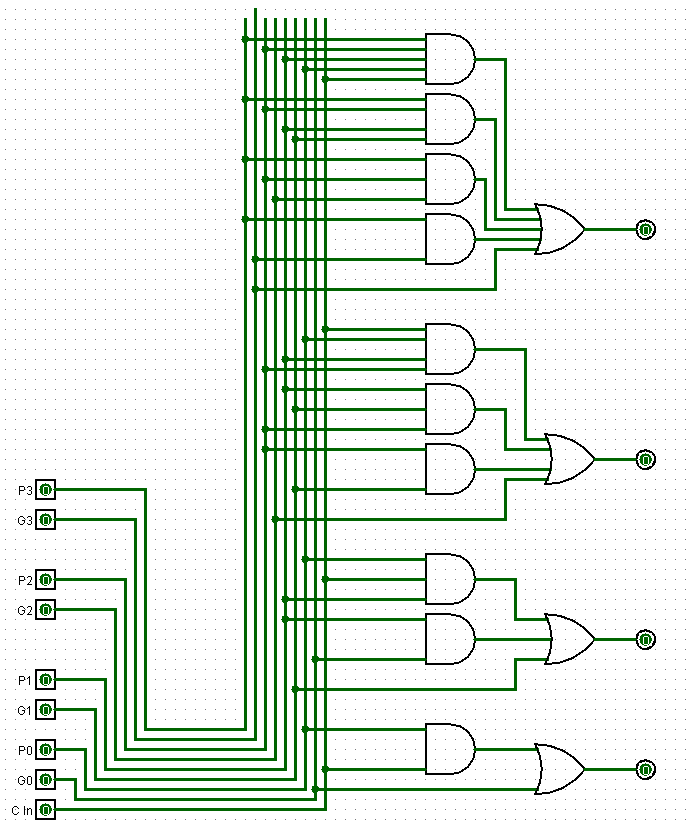
**Figure 22: ALU Overall Circuit**



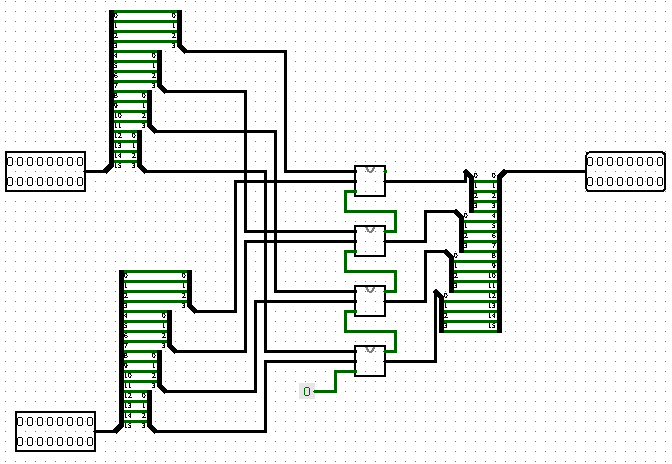
**Figure 23: 4-Bit Look-Ahead Adder Circuit**

**Addition Module:**

For the addition, subtraction, and increment by two modules in the ALU, a simple adder circuit was needed. For this design, a combination of a look-ahead adder and a ripple adder was created to allow for a small gate delay and low gate design. This will allow for a simple, low cost design that also minimized the gate delay. In Figure 23, the core circuit for the 4-bit look-ahead adder is shown. This section of the circuit creates the P and G values for the logic of the circuit in Figure 24 to use.



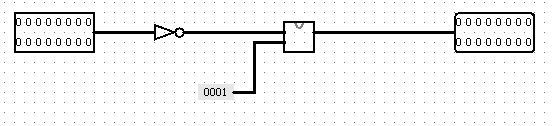
**Figure 24: 4-Bit Adder Resolution Module**



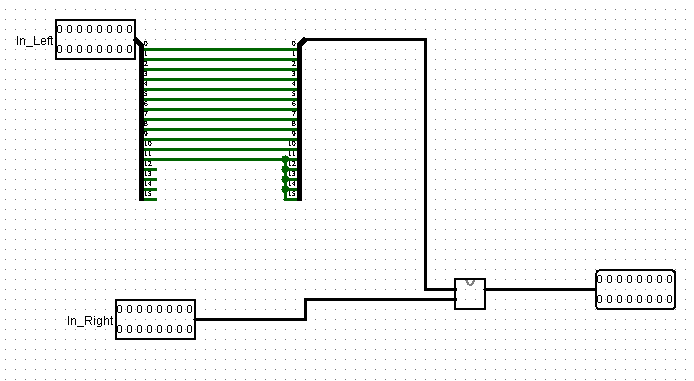
**Figure 25: 16-Bit Ripple Adder**

The final portion of the adder is shown in Figure 25. This figure shows how the look-ahead adders are combined with the carry-out values passing from one 4-bit adder to another. The means that for each 4-Bit adder, the there are 4 bits of the two inputs that are passed in and the carry out value for the previous set of 4-bits are passed into this adder. The first adder gets a 0 for its carry in value. Finally, the solutions of the adders are combined to a final value.

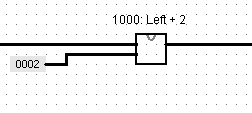
The 16-bit adder circuit is then used for the addition circuit, the subtraction circuit, by flipping the sign of the bits as shown in Figure 26, the Long\_Offset addition circuit as shown in Figure 27, and the Increment by 2 circuit as shown in Figure 28. It is to be noted that the modules shown in Figures 26 to Figure 28 are the 16-Bit Adder Circuits.



**Figure 26: Flip Sign Circuit**



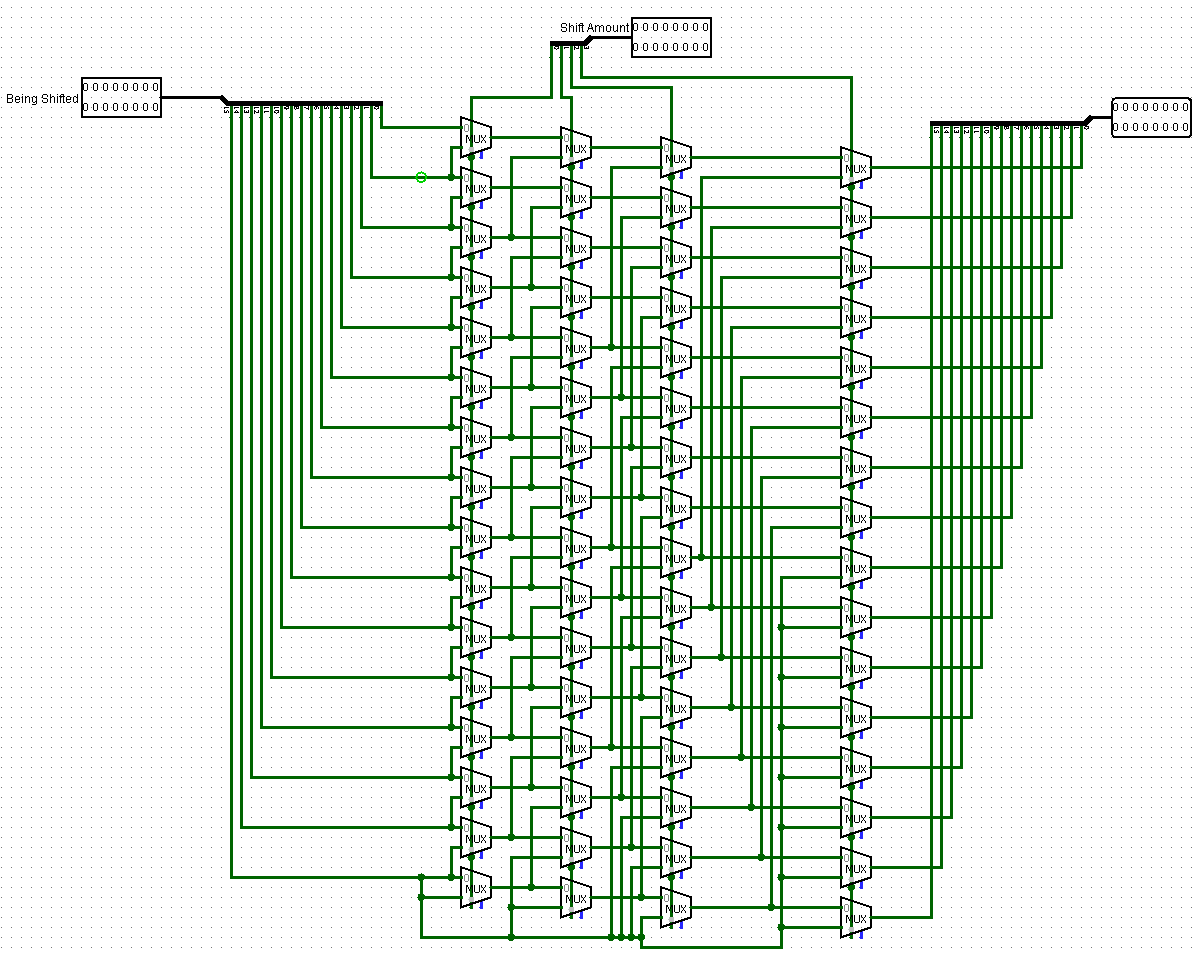
**Figure 27: Add Long Offset**

****

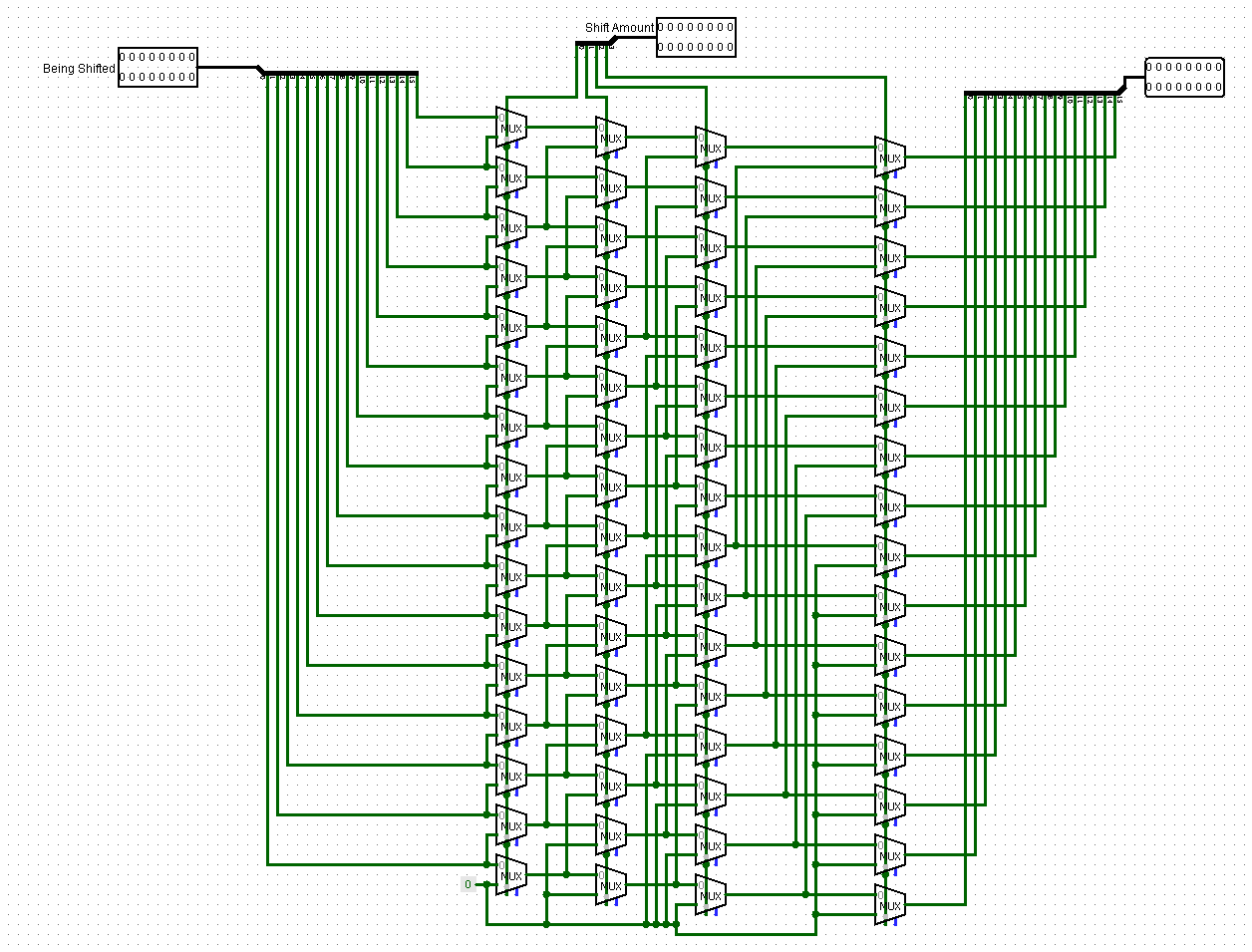
**Figure 28: Increment by Two Circuit**

**Barrel Shifters:**

In Figure 29 and Figure 30, the barrel shifter used for the left and right shift are shown. The barrel shifter takes into advantage the attributes of a binary number. A binary number can be separated to a sum of 2^n values. For example, for the number 0101, the value is equal to 2^0 + 2^2 or 1+4. For the barrel shifter, the 4 digit binary number is separated into 4 different shift amounts of 1, 2, 4, and 8. So, the binary number of 0101 is used to signify a shift of 5 bits, the first column of multiplexers will cause a shift of one bit and the third column of multiplexers will shift the bits by a value of 4 bits. The bit/s that is filled in depends on the direction of the shift. For a left shift, the shift is a logical shift. This means that the bits being filled in will be 0’s. For a right shift, the shift is arithmetic. This means that the bits being filled in will be the sign bit duplicated. The final thing to note is the direction of the wiring of the multiplexers determines the direction of the shift.



**Figure 29: Barrel Shifter Right**

****

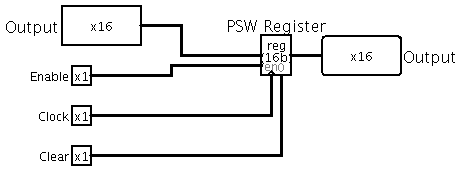
**Figure 30: Barrel Shifter Left**

**MAR / MDR / IR / PSW**

These registers are used to store values utilized throughout the instruction interpretation cycle. Since they are all implemented in the same manner one implementation description is sufficient.

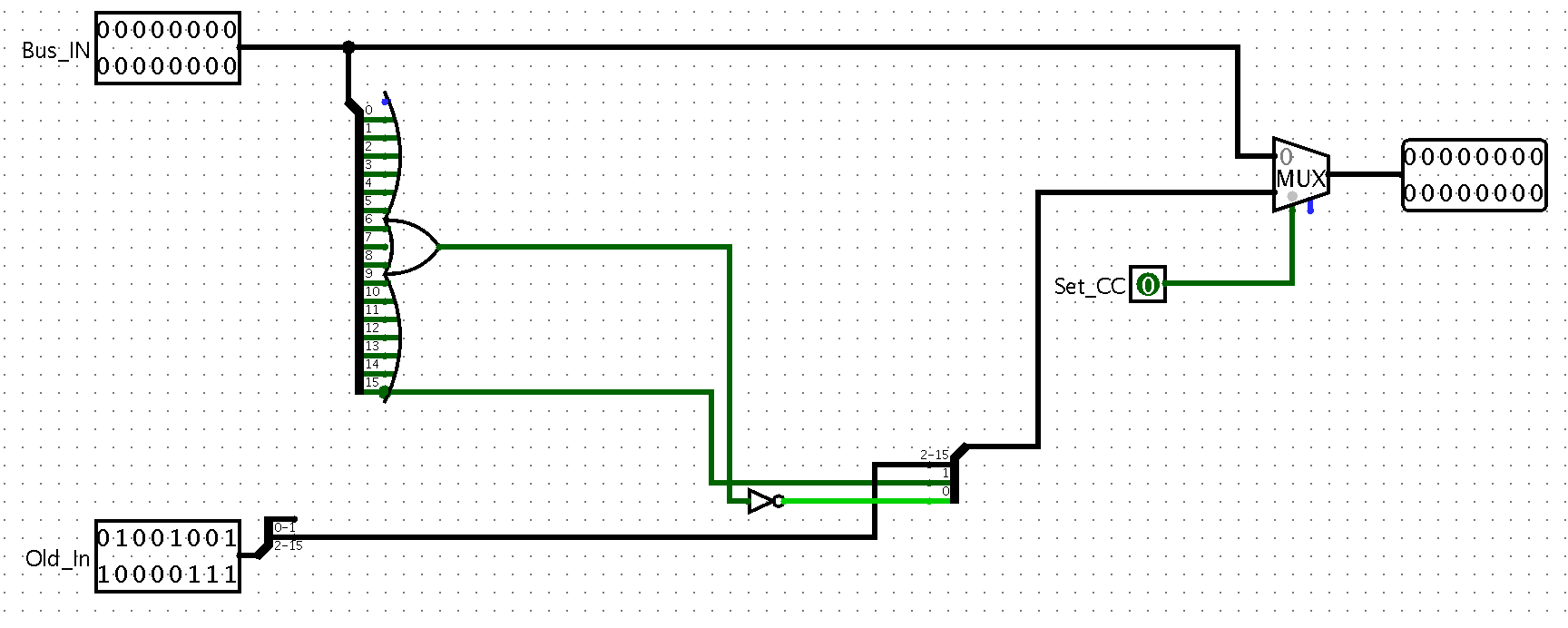
**PSW**

The PSW has a specific bit that allows the conditional codes to be set in the PSW and is one of the only differences the PSW has over the other storage registers.



**Figure 31: Register Implementation**

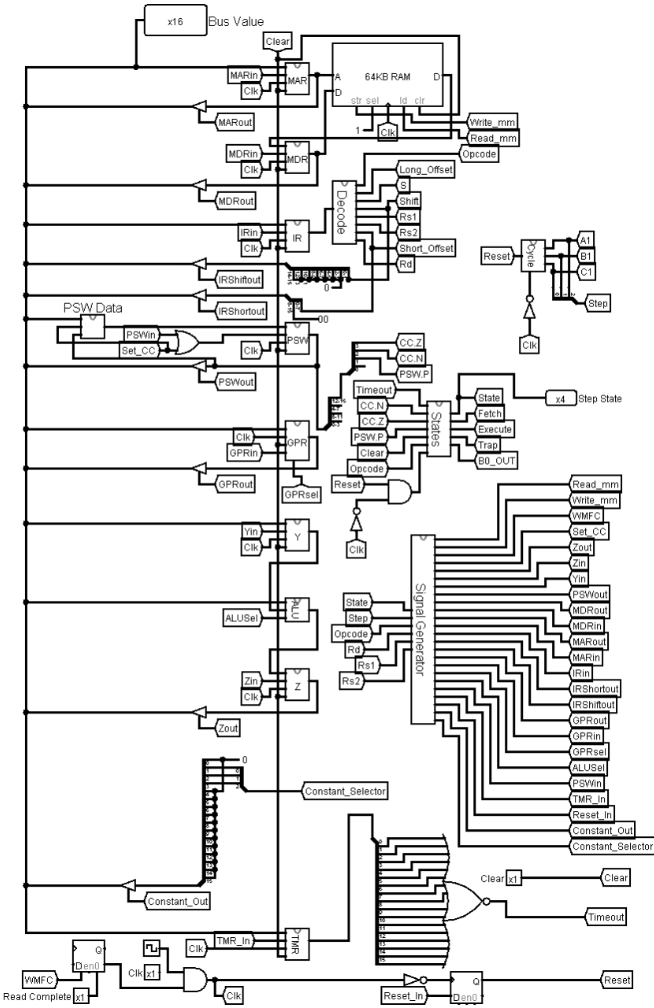
The following implementation was leveraged to set the conditional codes in the PSW when the SET\_CC signal is generated and passes in the conditional code for the given bit into the PSW.

****

**Figure 32: PSW Data Implementation**

The following circuit shows all the sub-circuits being leveraged in a single bus paradigm. Each module is combined from all the previous circuits. The main circuit shown below has 64 KB of RAM, but the circuit cannot be fully tested since there is no data saved in the RAM. All of the subcircuits have been tested and the circuit can be viewed going through the step state progression.

We have the reset on the falling edge of the clock signal because it allowed the reset signal to be included in the S\_5 signal of the numeric state machine. Additionally, the D latch is enabled asynchronously to one when the WMFC signal is activated. Triggering the D-Latch asynchronous means that it does not have to wait till the next clock signal to trigger. When the D-Latch is set to one the clock signal is blocked and the numeric state machine and step state machine is halted. When the read complete signal comes in on the next clock signal the D-Latch is set back to zero allowing the clock to be enabled again.

****

**Figure 33: Main Circuit**

**Optimizations:**

Each optimizational that is listed is sorted into the following sections on what type of optimization they fall into.

**Architecture Optimizations:**

One of the architecture optimizations include using a carry lookahead adder in the ALU. This allows for a ALU that has a low gate delay circuit. This circuit can be seen in Figures 23-25.

Along the same lines of the ALU, the number of gates used for the shift circuits was reduced by using the barrel shifter instead of a linear shifter that has on column of MUXes per desired shift. This means that the number of MUXes was reduced by 192 MUXes.

**Instruction Set Optimizations:**

The optimizations for the instruction set include mapping the first 6 opcodes to the ALU so the opcode can be passed directly to the ALU as the selector bit to perform the given operation. Additionally, by adding a reset capability to the numeric state machine, the numeric state machine can be reset to zero when the signals for a given state are processed. Within the design of the state machines themselves, many cycles were saved in the process of the resetting because some states have only a few execution steps. The reset signal allows the state machine to have six numeric states and not have to have no-ops for the signals that run for less cycles. This means the number of gates has been lessened by using the two state machines and there is not a increase of cycles due to more cycles in numeric state machine.

Another important optimization for the instruction set is shown in the Fetch cycle. In this design, the PC is stored on Y and the value of Y + Long\_Offset is stored in Z. This allows for a reduction in the number of cycles that the branches will have from 3 to 1 cycle and the number of cycles in CLK/LPSW execute from 5 cycles to 3 cycles. Also other operations such as Load, Store, and Not benefited from the PC existing in the Y register and reduced each execute step by a cycle. Other simple optimizations in the instruction set can be seen in the parallelism of the instructions.

One hardware optimization leveraged in this project was the use of previous states to calculate part of an instruction and then the final operations were carried out in succession. An example of this is opcode 7 where the MM is accessed at the PC + Offset and then for opcode 6 the returned value is then inverted. Doing opcode 7 and then inverting the response in a different state is an example of a hardware optimization utilized.