Andrew Pimental

Dillon Dragomir

**Project 4: Pipelined Control Unit**

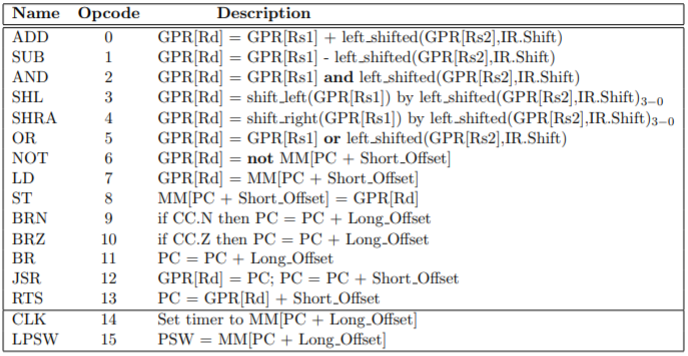
**Introduction:**

In this lab, students are tasked to create a basic pipelined machine. For the design described in this paper, a three stage pipelined machine was created. The three stages are for Fetching the Instruction, Decoding the Operands, and Executing the Instruction. Each stage will operate simultaneously and will allow for a fast system. Also, each stage will only need one clock cycle to move the instruction to the next stage. Various hazards have been handled to allow for seamless running and the Traps are handled outside of the pipeline.

**Requirements:**

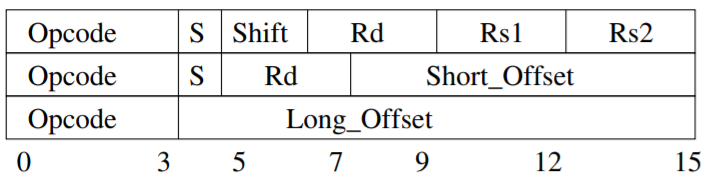
For this project, the following requirements have been given:

* Word Size = 16-bits
* 16 Instructions as followed:



**Figure 1: Instruction Set**

* 8 16-bit General Purpose Registers
* Two read and three write ports on GPR
* Main memory can be read and written simultaneously
* 16-bit Program Counter in GPR([R7])
* 2’s Complement Number Representation
* Instruction Formats:



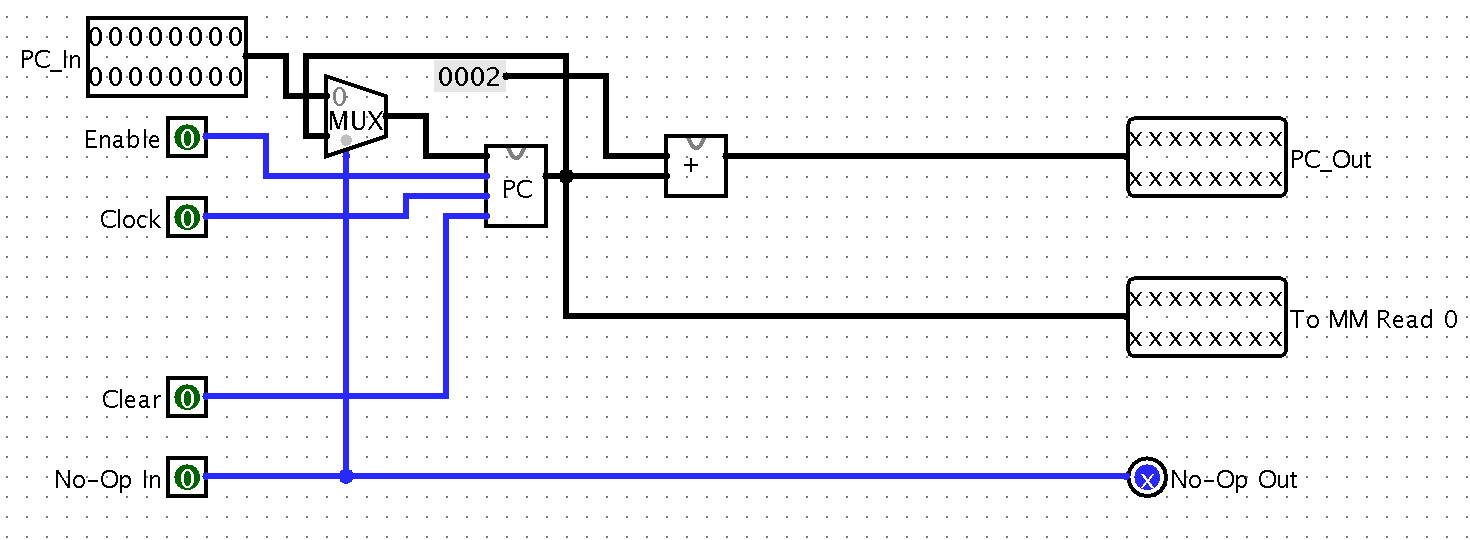
**Figure 2: Instruction Formats**

**Fetch (Stage 1):**

The fetch stage consists a register that will hold the current PC value, logic to add 2 for incrementing the program counter for the next stage, and a line to pass the PC out of the Fetch stage. The memory read should be assumed to start at the beginning of each clock cycle and the value will be available at the end of the clock cycle. When the PC is acquired, there are two main possible values that enter into the stage, the incremented PC or a branch PC. If later in the pipeline, it is found to be taken, the instruction that is passing from Fetch to Decode Operands needs to be assigned No-Op and the PC for the next Fetch needs to be the branch value. The reasons why branch prediction is still necessary is the value cannot be determined branching or not and passed back to stage one in time. Below shows a summary of what the Fetch stage does for its instruction.

* Load PC from GPR[R7]
* Place PC into Main Memory Read
* Initialize Main Memory Read
* Increment PC
* Pass PC out to next stage

The following circuit is the fetch circuit and also incrementes the program counter by two on every clock cycle. If a branch is taken the clear is enabled to flush the value in the PC and then the PC is set to the branch location.



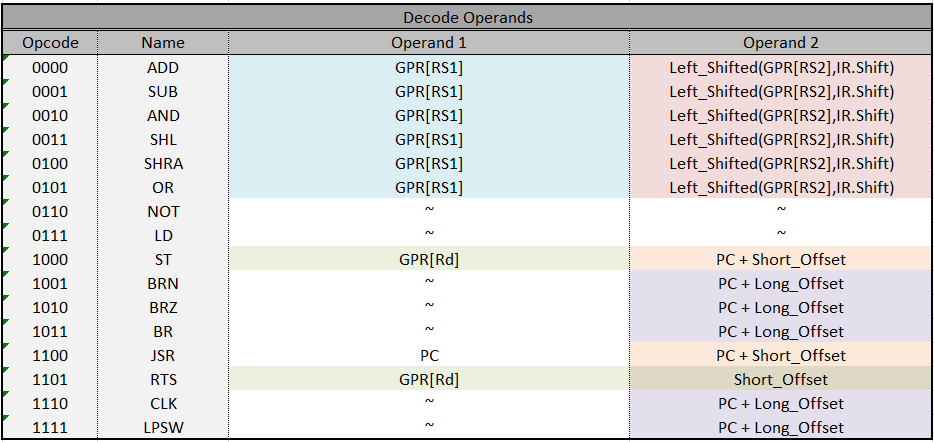
**Figure 3: Fetch Circuit**

**Decode (Stage 2):**

The decode stage is responsible for finding the needed operands and starting the memory reads for the execute step. This stage takes in the following values and stores them in registries:

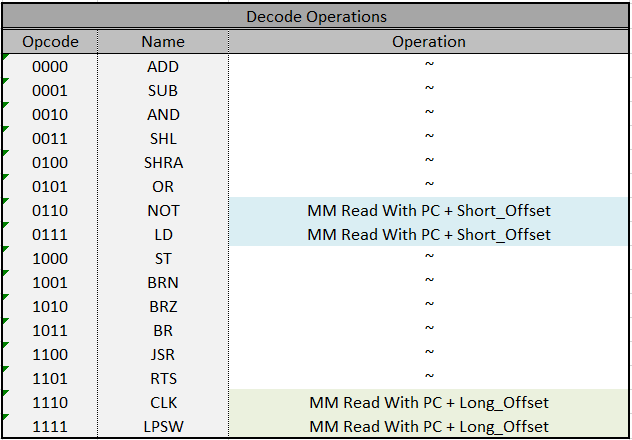
* PC
* IR (From Previous MM Read)
* GPR Read Port 0
* GPR Read Port 1
* Any No-Op Commands

This means that at the end of the previous clock cycle, these values will be loaded into this stage. When the clock cycle then starts, the IR will be taken from the registry and decoded so the parts can be seen independently. The next step of this stage is to find both Operand 1 and Operand 2 using the opcode previously found. Figure 4 shows which opcodes need which Operand 1 and 2

****

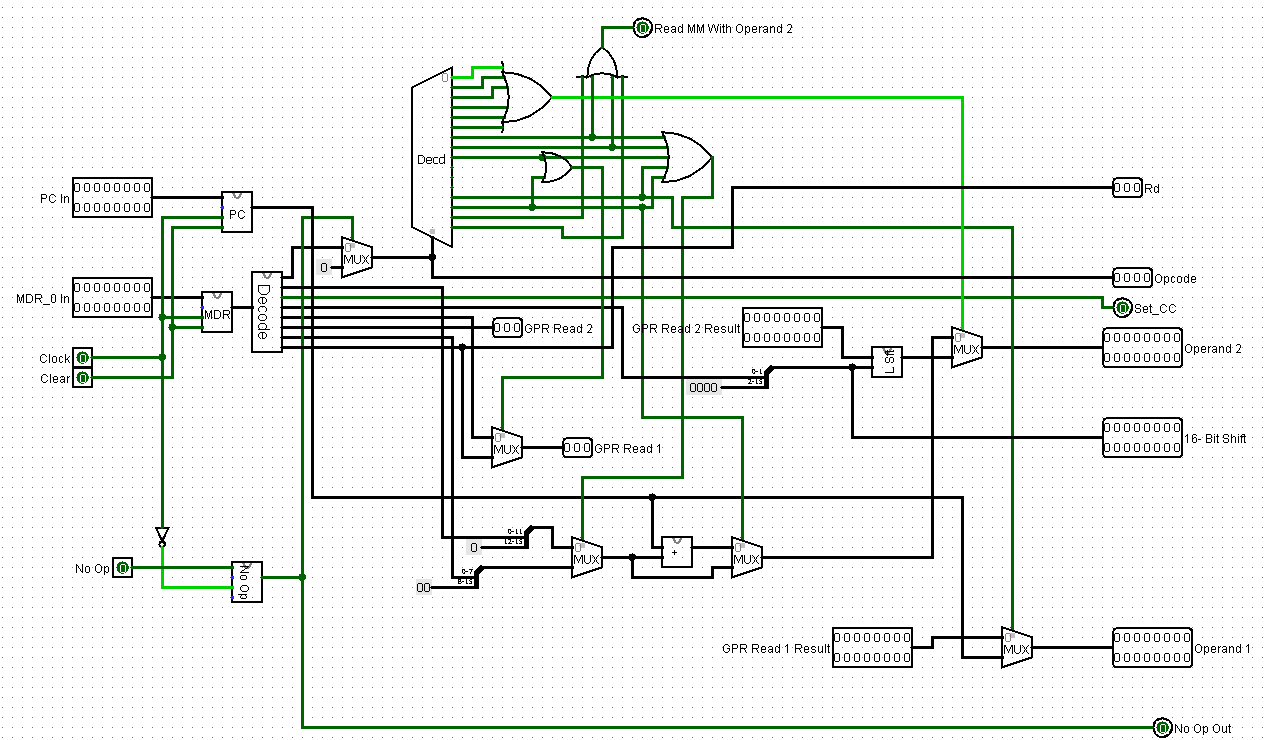
**Figure 4: Decode Operands**

As mentioned before these operands will be acquired depending on the found opcode of the current instruction. All operands will have their values passed into the components needed to calculate them and a set of MUXes will determine which operand value to pass to the next stage. Another thing to note is that there are some read operations that are needed to be started in this stage the be able to use them in the Execute stage. Figure 5 shows the operations needed to be done.



**Figure 5: Decode Operations**

Another point to make is that all actions in this stage in the Fetch stage are not affecting the memory space of the main machine. This means if a branch occurs or an invalid access timeout occurs, the only action of these two stages that needs to occur is to insert no-ops for the incorrect branch instructions and continue down the correct PC value at the fetch.



**Figure 6: Decode Circuit**

The above circuit shows an example of these operations. On the left are the inputs described above and on the right are the values passed to the next stage or that are used for between stages steps.

The following values are passed from the decode stage to the execute stage:

* Rd (Which register is Rd)
* Opcode
* Set\_CC (Set Conditional Codes)
* Operand 1
* Operand 2
* 16-Bit IR.Shift
* No\_Op

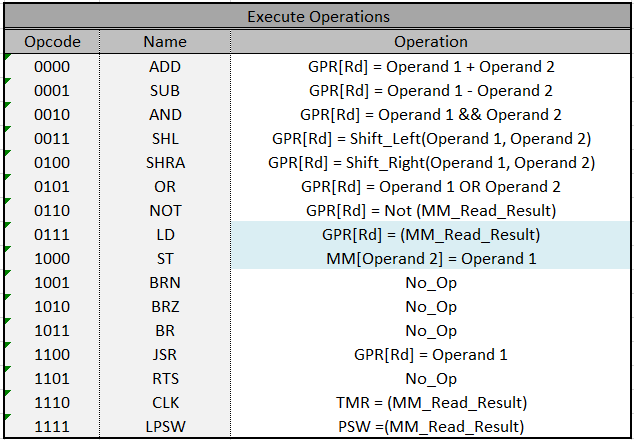
It is to note that to handle the No-Op in this stage, the opcode is set to 0000. This is to prevent any read operations to or from memory and to allow easy handling in the execute step.

**Execute (Stage 3):**

The execute stage is where all the logic happens and where the values are written back to the main memory after an ALU operation. This stage takes in the following values in addition to the clock and clear signals and stores them in registries:

* PSW
* Set\_CC
* MDR 1 (From Previous MM Read)
* Operand 1
* Operand 2
* Rd
* Opcode
* No\_Op

After acquiring the signals above on the off cycle of the clock, the following operations shown in Figure 7 need to happen in the Execute stage.



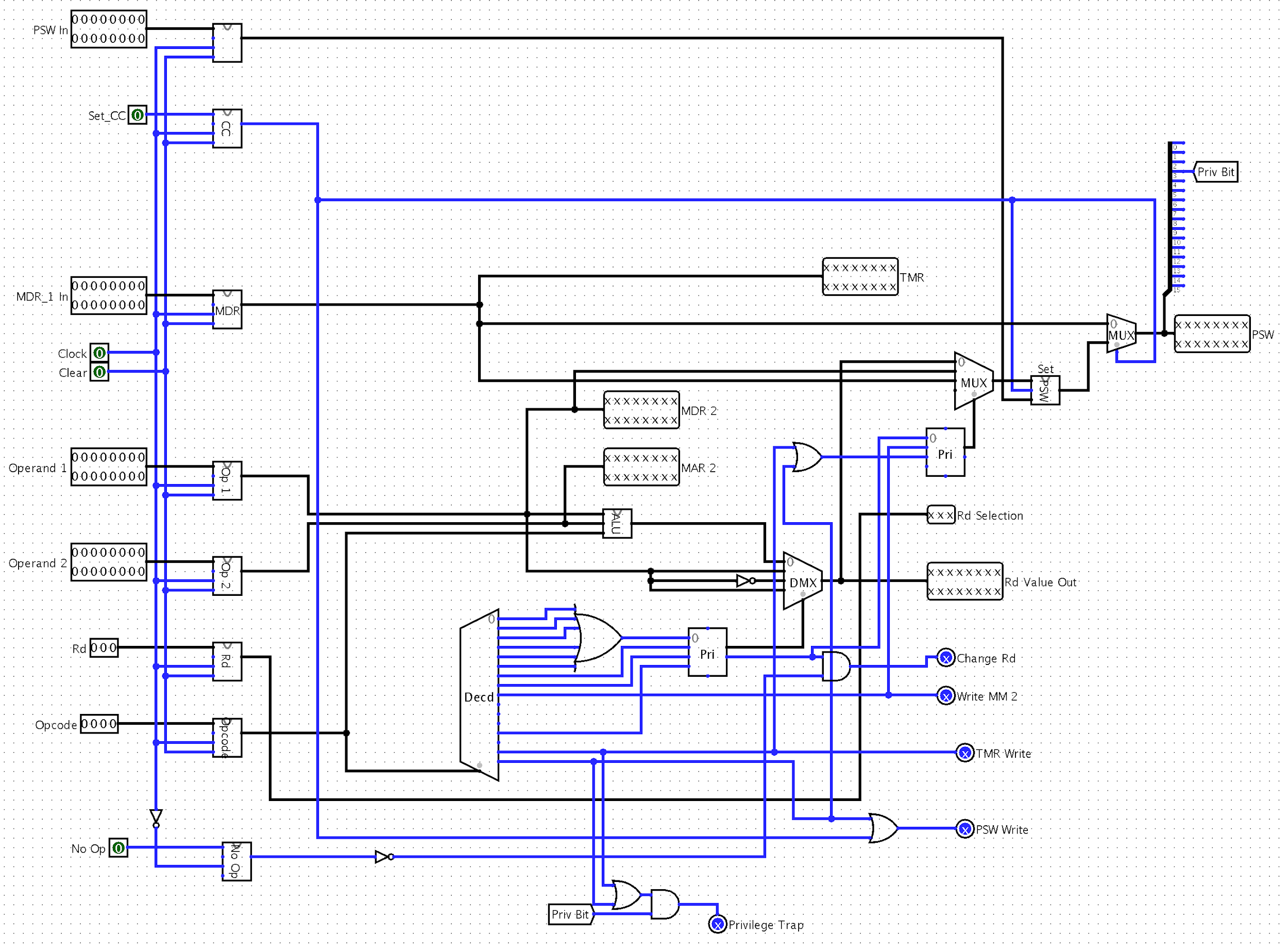
**Figure 7: Execute Operations**

One other operation that needs to happen within this stage is writing to the PSW when the Set\_CC flag is true. When this occurs, the value that was just used, is then used to compare to see if it is zero or negative. If the value is zero, the PSW.Z bit will be 1 else it is 0. The same is for the PSW.N bit with negative. If the value is negative, the PSW.N bit will be set to 1 else it is set to 0. Now with this implementation, the rest of the PSW is copied back into the output PSW. This new PSW value is then passed out of the stage to be placed into the PSW.

Another part of the circuit that needs to be mentioned is the No\_Op signal. When No\_Op is true, the opcode is forced to be 0000 and the resultant value is not allowed to be written to the register. This allows the circuit to have no effect of the overall machine and not be overly complicated by preventing all writable values.

The final part to note about this circuit is the output values of the execute stage. These values will be sent to direct connections to other part of the system to either write values or handle hazards. These are the Execute Stage’s outputs:

* Write to PSW
* Write to TMR
* TMR Write Bool
* Rd Selector
* Rd Value
* Change Rd Bool
* MDR out
* MAR out
* Write to MM



**Figure 8: Execute Circuit**

**Trap:**

When trying to optimize a system, Traps are considered less important than the rest of the system. As such, in this design, Traps will not be handled within the pipelined system. Instead an external system will handle Traps. The way each will be handled will be seperated into the two types of traps within this problem; Program Check Violations and Timeouts.

**Timeout Traps:**

A time trap will occur when the timer reaches 0. This allows programs to have a set amount time with the system and if they are not finished in the allotted time, will be set aside for the time being. In this system, it is defined that when a timeout trap occurs, the pipeline will finish all instructions that are partially completed and then will execute the trap. This process will occur with the following process.

1. With a full pipeline, the system receives a binary signal to Timeout Trap. This leads to starting a counter iterating from 3 to 2.
2. Each time a clock cycle passes, the counter will continue to iterate down, and a No\_Op is passed into the front of the pipeline.
3. When the counter reaches 0, the clock is prevented from entering the pipelined system.
4. The system will then save the following values as followed using direct connections
   1. MM[8] = PSW
   2. MM[10] = PC
   3. PSW = MM[12]
   4. PC = MM[14]
5. Finally the flag for Timeout trap is set back to 0 and the counter is returned to 3 allowing the clock to once again enter the pipelined system.

After these steps finish the system will be pointing at the PC to continue on with the pipelined system.

**Program Check Violations Traps:**

If a privileged instruction is processed and the privileged bit is not enabled a trap will occur. When this trap occurs the pipeline will be flushed and the following steps will occur:

1. The clock is prevented from entering into the pipelined system upon seeing a Program Check Violation by setting a One bit register to 1.
2. The system will save the following values as followed using direct connections
   1. MM[0] = PSW
   2. MM[2] = PC
   3. PSW = MM[4]
   4. PC = MM[6]
3. When this is complete, the system will set the Program Check Violation back to 0 and the clock will resume.

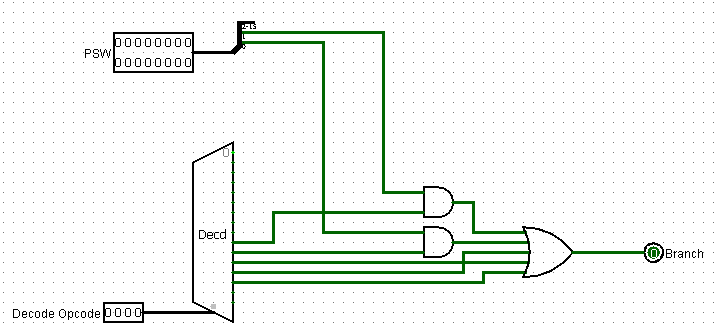
After these steps occur the system will be pointing at the correct PC to continue on with the pipelined system.

**Hazard Handling:**

**Control Hazards:**

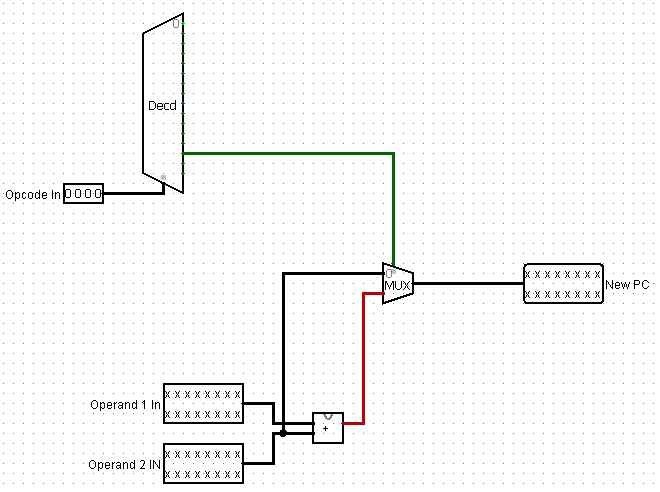
This pipeline does static branch prediction and always predicts branch not taken. If the prediction is correct the pipeline continues functioning. If the branch is found to be taken, the pipeline is flushed, the branch is taken, and the pipeline continues.

The following circuit shows how the opcode and PSW is analyzed to see if the branch needs to be taken. It is to note, that the PSW is passed through the execute stage first. So if the execute stage changes the PSW, this is taken into account. Also the branch is resolved after the decode step. This mean that the value for the PSW will be accurate.



**Figure 9: Branch Check**

If the branch check shows a positive for branching, then the value of the PC will be written with the value found from the below circuit. The circuit will always pass through the 2nd operand unless the opcode is 13. For opcode 13, the new branch PC is GPR[Rd] + Short Offset. This means that the GPR[Rd] needs to run through the check for data hazards before being added to the Short Offset. This step is describe in the data hazards section.



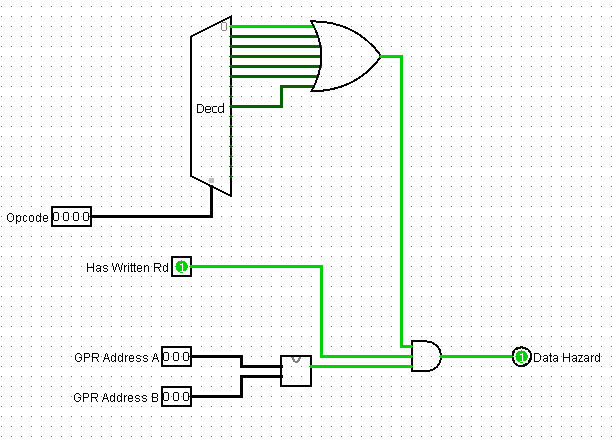
**Figure 10: Branch Calculation**

The final step of handling this hazard, is passing a No\_Op signal to the decode step in the next stage. This will prevent the incorrectly loaded instruction from executing. The correct instruction will be loaded in the Fetch stage and will make it into the Decode stage one cycle later.

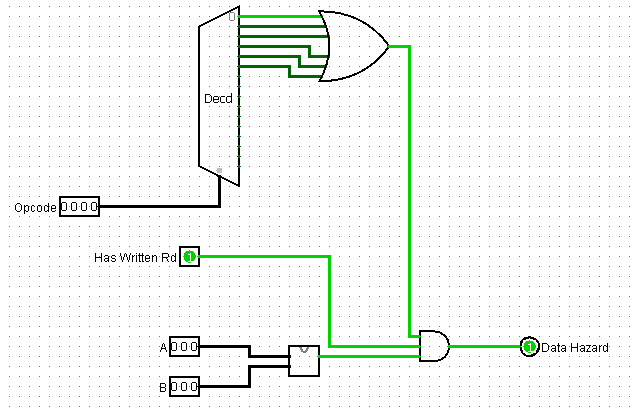
**Data Hazards:**

This pipeline does not stall when a data hazard happens. The data hazards can be handled with data forwarding. In a five stage pipeline there would be three different types of data hazards, but keeping the design to three stages drastically reduces the amount of hardware required to handle data hazards.

For this design the data hazard is check in between the decode and execute stages. The decode stage will pass out the opcode of the instruction attempting to move from decode to execute and the location of the register for that operand. These values are used together with the operands provided by the execute stage which are, if the registers have been written to and to which register. The circuit below shows the comparison of the two register locations and the check to see if the decode instruction will need a register value and if the execute stage has written to the GPR. If so, the circuit will provide a true as a data hazard has occurred. This will cause the circuit to use a MUX to pass through the value just saved by the execute step instead of the original operand. It is to note that since operand two is left\_shifted when a collision is possible, the resulting value will correspondly be left\_shifted with the value provided by the MUX using a single module for left\_shifting and the shift value passed from the decode stage.



**FIgure 11: Operand 1 Data Hazard Check**

****

**FIgure 12: Operand 2 Data Hazard Check**

**Structural Hazards:**

Since the ALU does not have floating point operations, structural hazards do not have to be handled. Additionally, since this machine has a split I and D cache a structural hazard cannot occur when the execute stage is writing back to the main memory. Also, this project assumes that registers can read and write simultaneously.

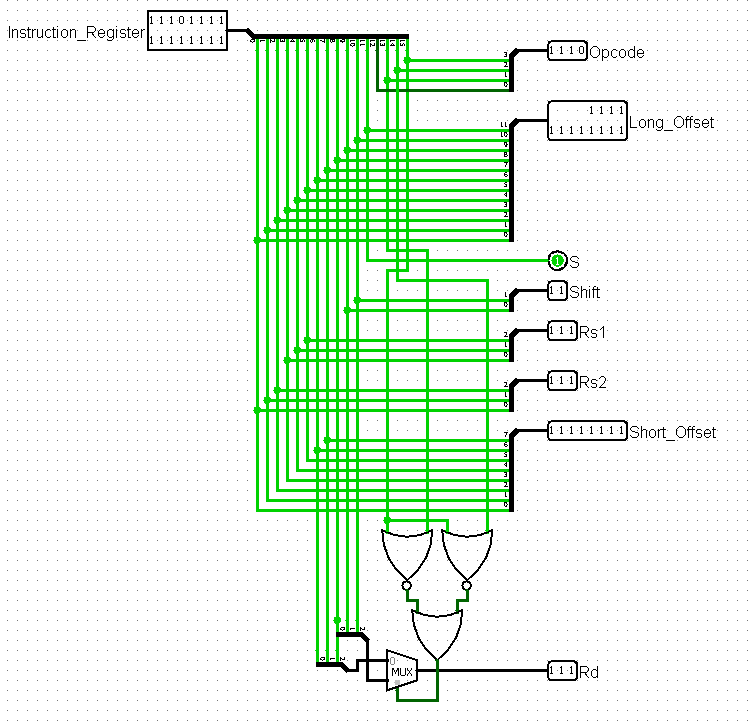
**Hardware Examples:**

**GPR:**

The GPR will store the current stage program counter in register 7. Though this is probably redundant, if the pipeline was extended and more hazards occur there could be some benefit to having the program counter stored. It is also assumed that the GPR has two write and three read ports. This enables the machine design to be simplified and handle less hazards because more data can be simultaneously written back from the execution stage and read in the decode stage and not cause a structural hazard. A well optimized compiler could also help with trying to read and write the same register simultaneously.

**IR Decoder:**

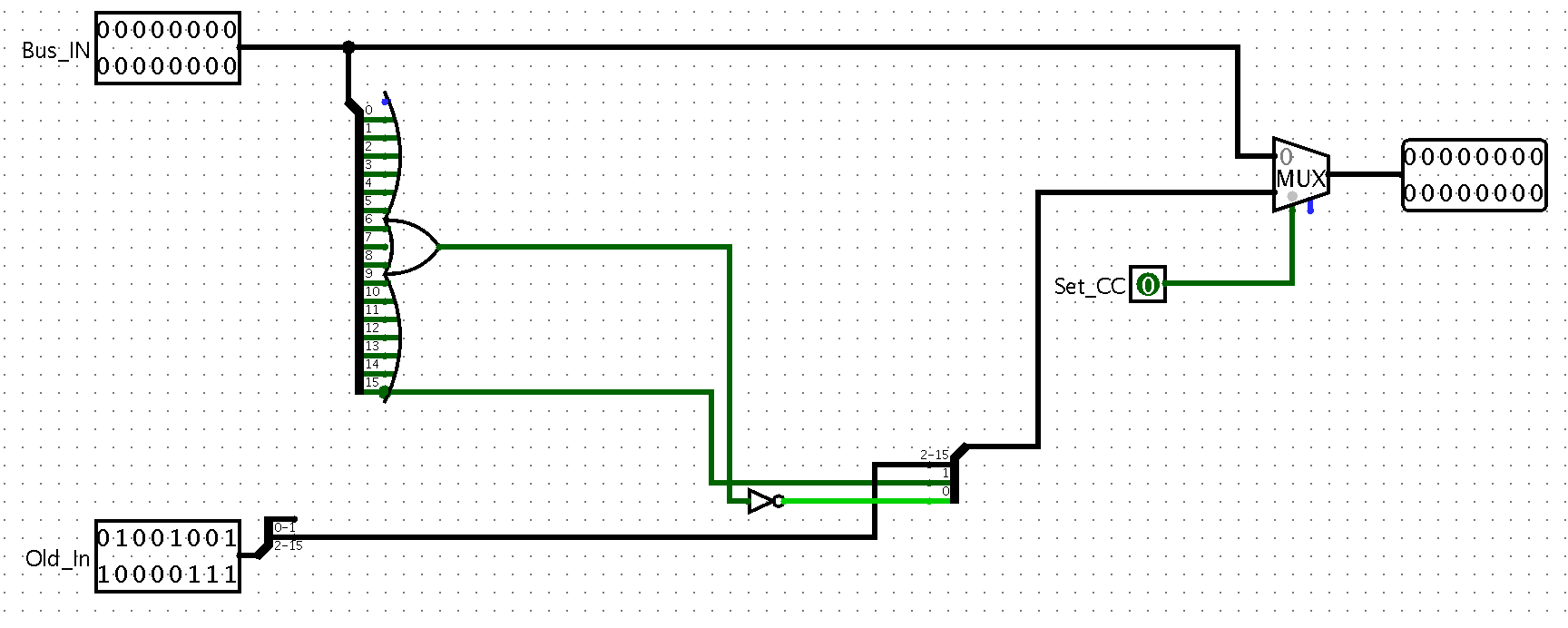
The instruction register decoder takes the information from the IR and decodes the instruction format into separate codes to be accessed elsewhere. The IR decoder assumes that the machine only accesses the outputs that are desired for a given opcode. Finally, since Rd is not always in the same location, 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 13: IR Decoder**

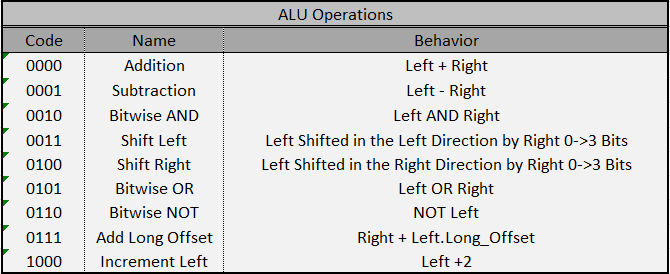
**Set Conditional Codes PSW:**

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.

****

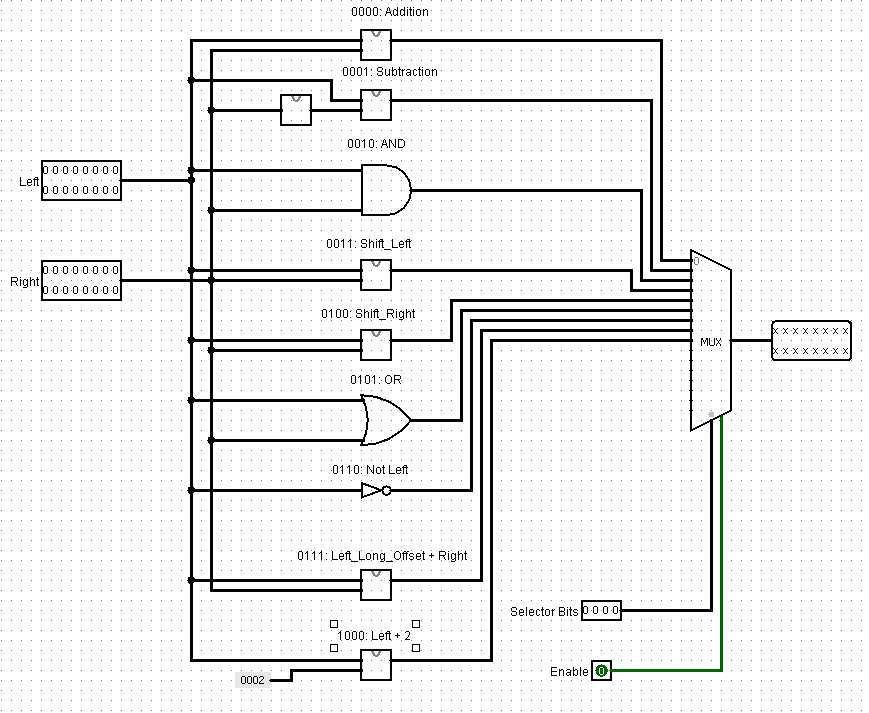
**Figure 14: Set\_CC Circuit**

**ALU Design:**

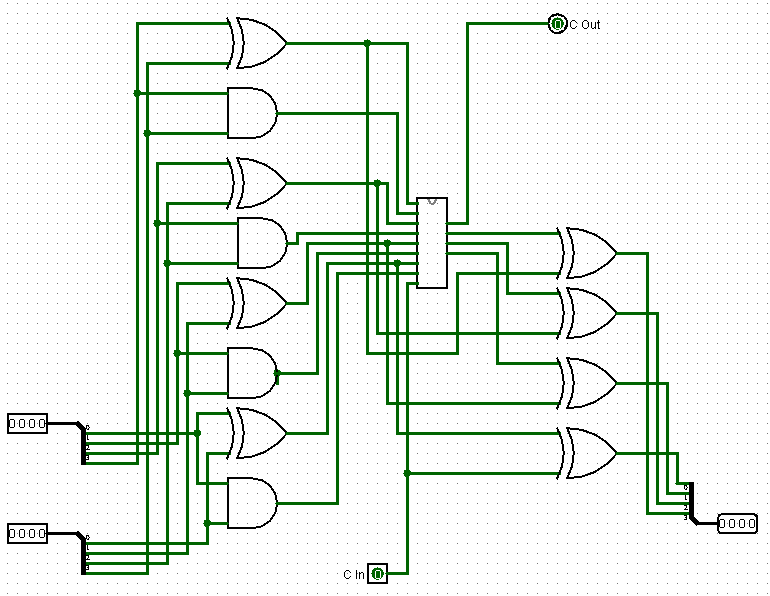
****

**FIgure 15: ALU Operations**

In this project, the ALU needs to match the following operations shown in Figure 15. Each operations is needed in a particular step within the above reduced signals. The overall circuit is shown in Figure 16. 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.

****

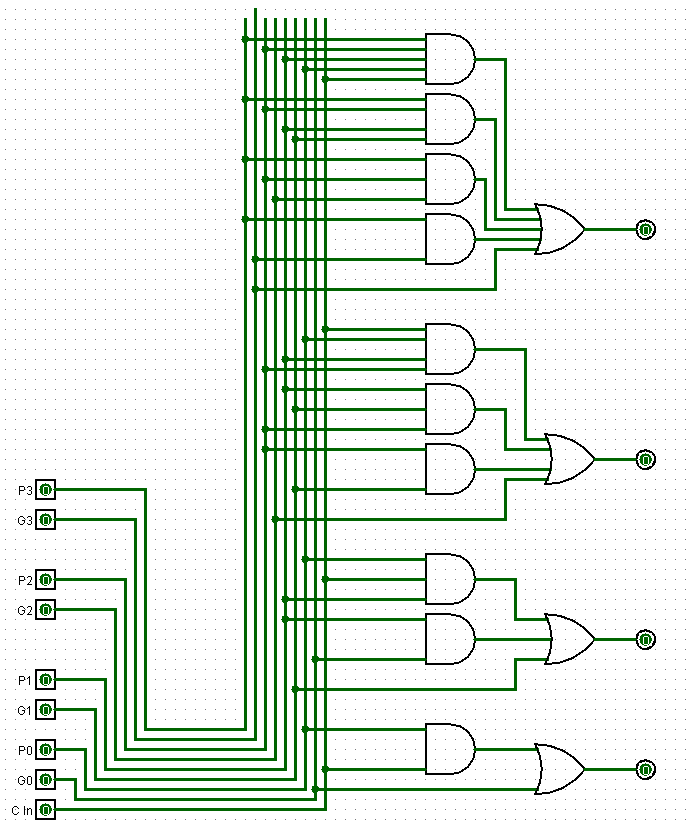
**Figure 16: ALU Overall Circuit**



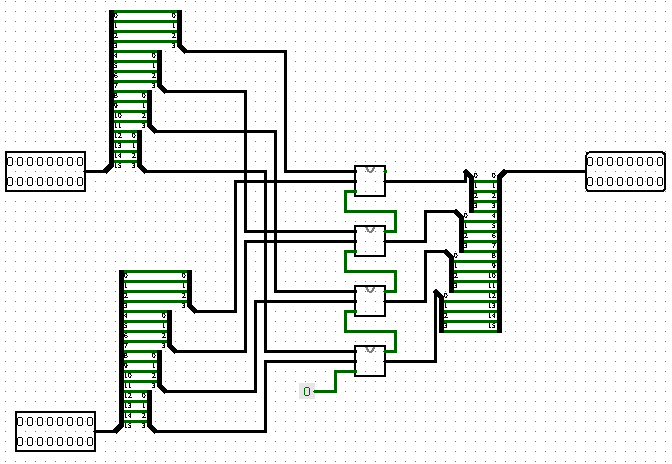
**Figure 17: 4-Bit Look-Ahead Adder Circuit**

**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 17, 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 18 to use.



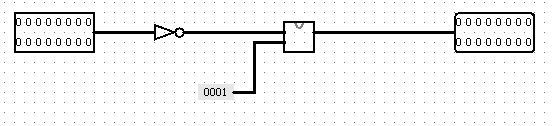
**Figure 18: 4-Bit Adder Resolution Module**



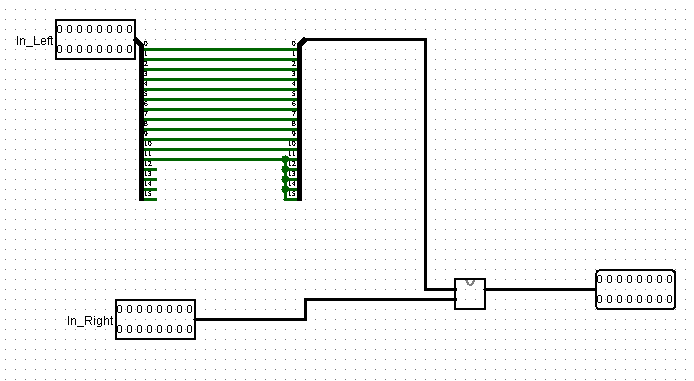
**Figure 19: 16-Bit Ripple Adder**

The final portion of the adder is shown in Figure 19. 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.

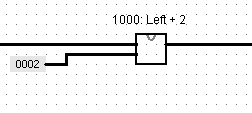
The 16-bit adder circuit is then used for the addition circuit, the subtraction circuit, by flipping the sign of the bits as shown in Figure 20, the Long\_Offset addition circuit as shown in Figure 21, and the Increment by 2 circuit as shown in Figure 22. It is to be noted that the modules shown in Figures 20 to Figure 22 are the 16-Bit Adder Circuits.



**Figure 20: Flip Sign Circuit**



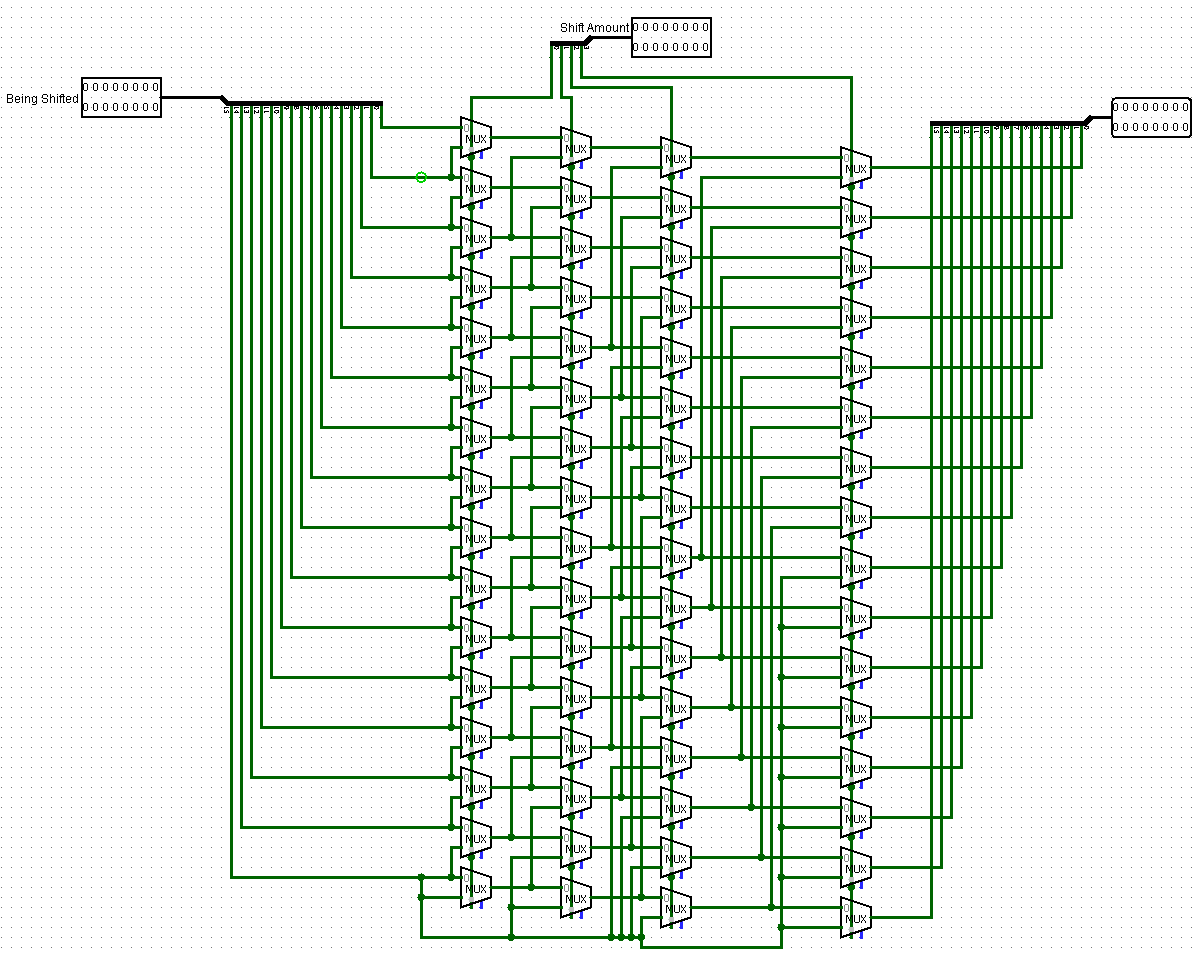
**Figure 21: Add Long Offset**

****

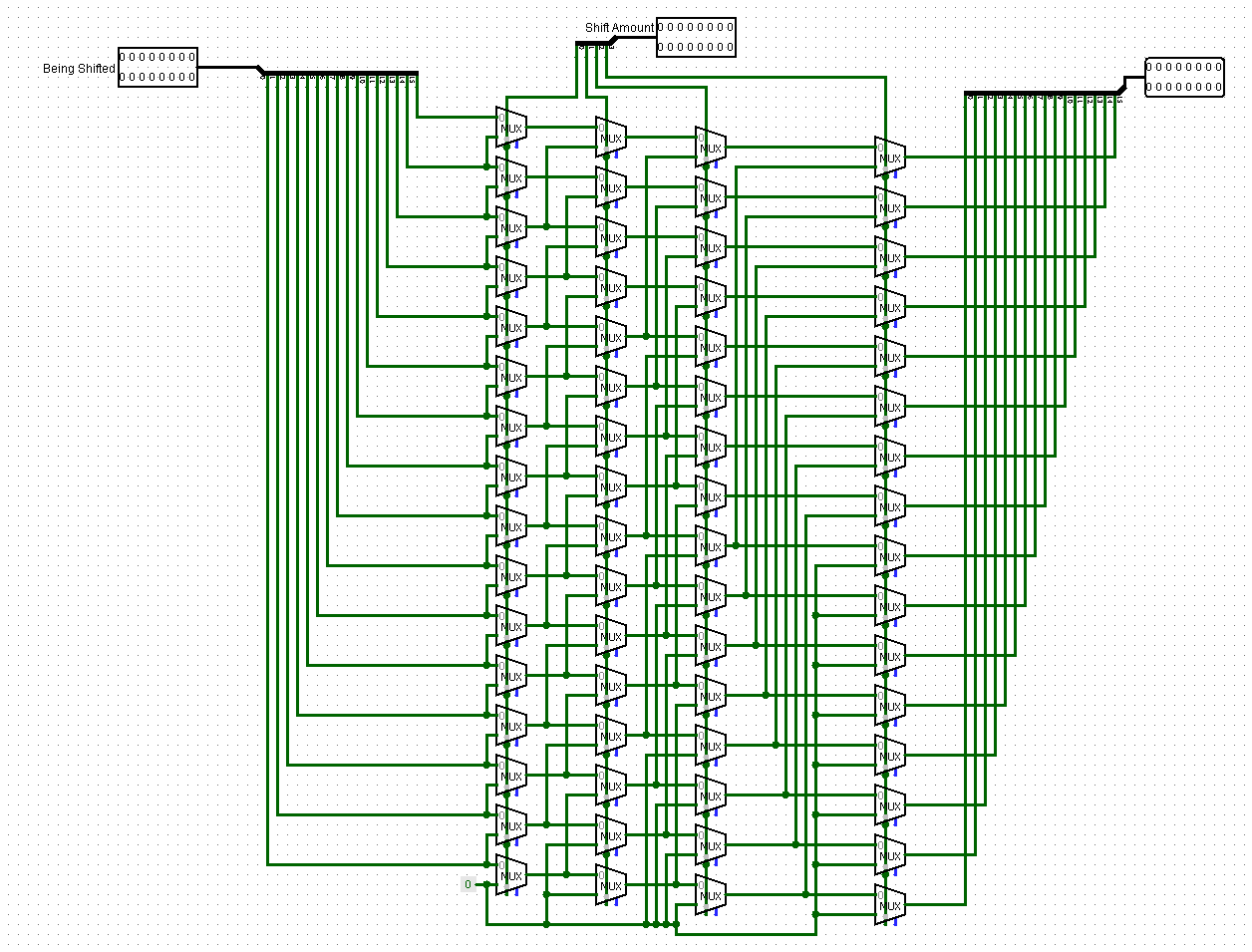
**Figure 22: Increment by Two Circuit**

**Barrel Shifters:**

In Figure 23 and Figure 24, 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.



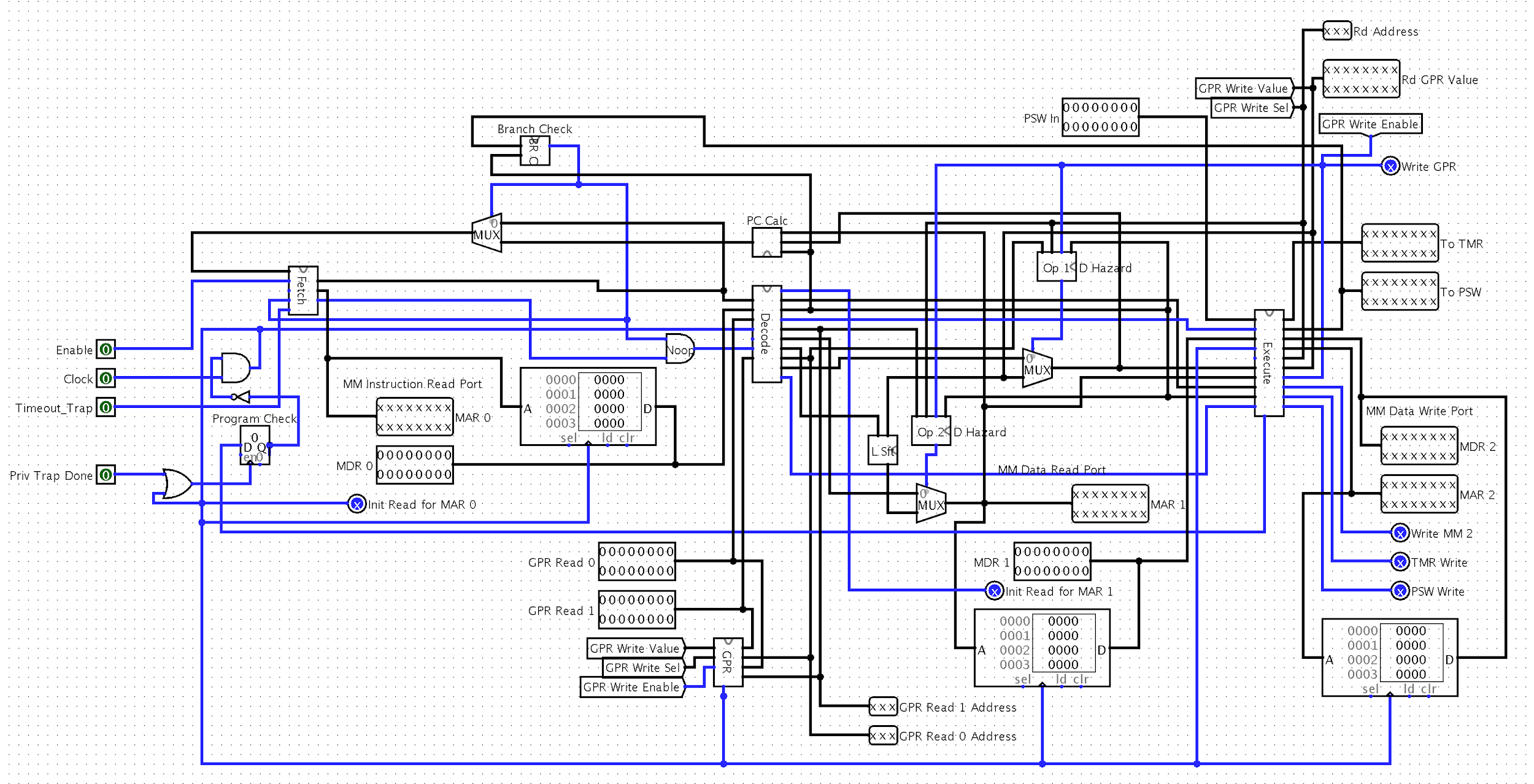
**Figure 23: Barrel Shifter Right**

****

**Figure 24: Barrel Shifter Left**

**Pipeline Circuit:**

The following circuit shows the three stages in the pipeline. In this block level implementation a few assumptions were made that did not fit within the constraints of logisim and will be described in the following sentences. The main memory was assumed to be able to read and write simultaneously. Since this is not a feature that is implemented in logisim, it was chosen to represent this assumption with multiple memory units. Since the memory unit cannot be implemented properly, the read and write back that occurs during traps specifically is not implemented but the trap check and noop insertion for traps occur. The countdown timer is not implemented, so an input bit is utilized to trigger the timeout trap, but would later be triggered by a countdown timer. Additionally, an external circuit that handles the privileged instruction trap would be implemented that would start the pipeline back up when the privileged trap is done. Also, making these assumptions and our limited instruction set makes it possible to avoid all control hazards. Also, since the GPR has multiple read and write ports the control hazards that would occur where data is written back to the GPR is avoided. The instances where data hazards occur, forwarding is shown in this circuit and the data is forwarded from the operands to the ALU. Finally, this pipeline has a static branch predictor implemented at the top of the circuit that controls branching in the fetch stage. If a branch happens, no-ops will be inserted in the fetch stage where the previous instruction was fetched and the branch location is fetched.

****

**Figure 24: Pipeline Circuit with Hazards Handled**

**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.

**Design Optimizations:**

With this system all stages will finish their operation in a single clock cycle. This means that each instruction will only need 3 cycles to complete and an instruction will be completed each cycle. This comes with the caveat that no branches occured. When the branch is taken, there is only a loss of a single clock cycle.