Pipelined Mini-MIPS

Kanu Kanu (kjk225) Benard Kipkoech (bk376) April 7, 2016

1 Introduction

This serves as design documentation for a subset of the MIPS32 architecture that we have implemented in Logisim. In this document, we will provide an overview of how we implemented our pipelined processor and outline and nuances that our architectural design choices may present.

2 Overview

In this project we implemented a five-stage MIPS pipeline, which is the most common organization for MIPS. This includes:

1. Fetch

- All MIPS32 instructions are 32 bits in Length which made it very easy to fetch instructions from instruction memory. By manipulating the program counter, we were able to make it so that the program counter increments by four after each 32 bit instruction is fetched.

2. Decode

- MIPS only has a few instruction formats with source register fields being located in the same place in each instruction. We decoded the instruction by recognizing the structural patterns between each instruction and choosing between instruction types based on the operation code.

3. Execute

- Given the operands we used the ALU and and comparators to execute the desired operations.

4. Memory

- Take the ALU result and use as memory address for loading or storing.

5. WriteBack

- The write back value is determined in the memory stage. If the instruction is a Load, then the data to be written back come from the memory else it is the ALU result.

In this project we also dealt with
hazards: Structural Hazards,

- To avoid Structural Hazards, we set all registers set to takes input at a rising edge while only the clock to the register file at a falling edge. This helped us ensure that values being written in to the register file would be written first rather than read first.

Data Hazards

- To avoid data hazards we implemented forwarding logic such that updated register values that needed to be operated on during execute were determined during decode and the forwarded during the next stage.

Control Hazards

- We implement a sub circuit to detect if an instruction is a jump or a branch. The logic sets the write enable control to 0 for the instruction already in PC and therefore allowing a delay slot.

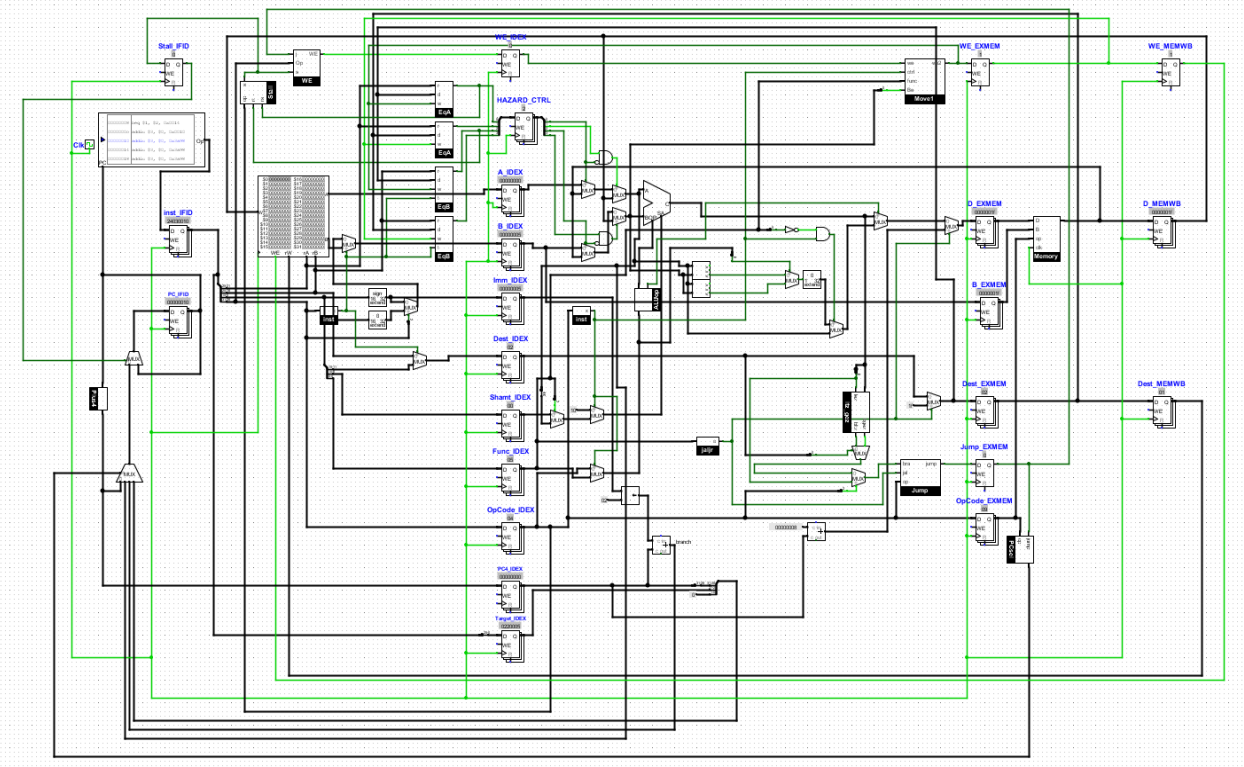
Load-Use Hazards

- To avoid hazards that due to load use, we made a stall logic at decode stage which checks if the instruction is a LW and followed by an instruction that uses the data from the destination register updated by LW. Stall outputs 1 if the condition for load use hazard is met which is in turn used to change all the controls to 0 in that pipeline.

3 Circuit Diagram

MIPS

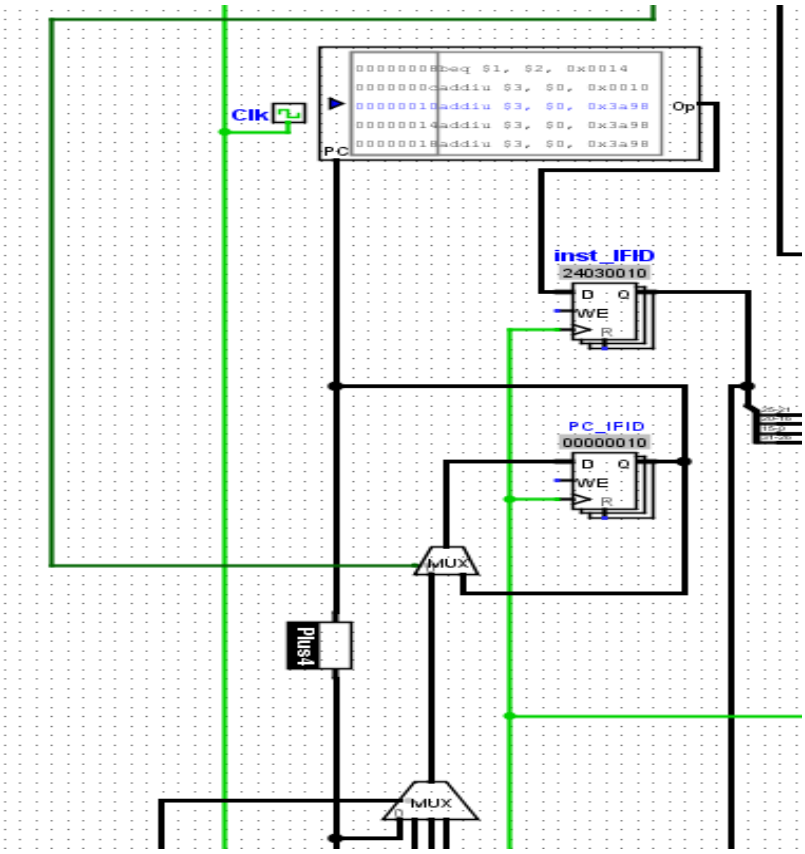
This is the overview of our complete MIPS processor:



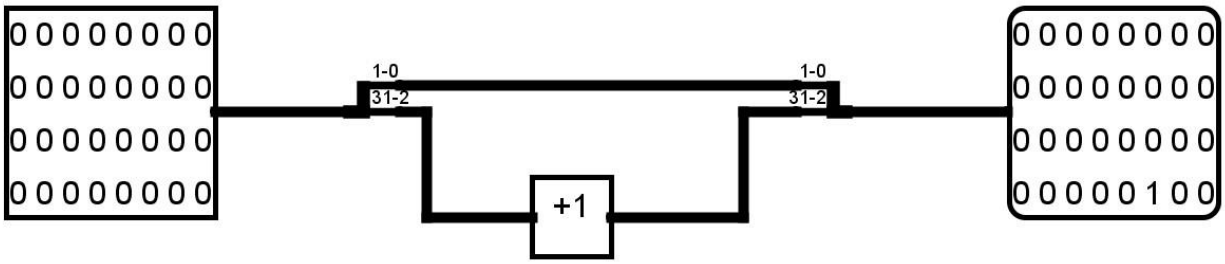
The processor executes all Table A and Table B instructions depending on the instruction code from the program Memory

4 The Fetch Stage

4.1 Circuit Diagram



4.1.1 Plus 4



Counting to 4 in binary occurs in the following pattern.

0000
0100
1000
0000

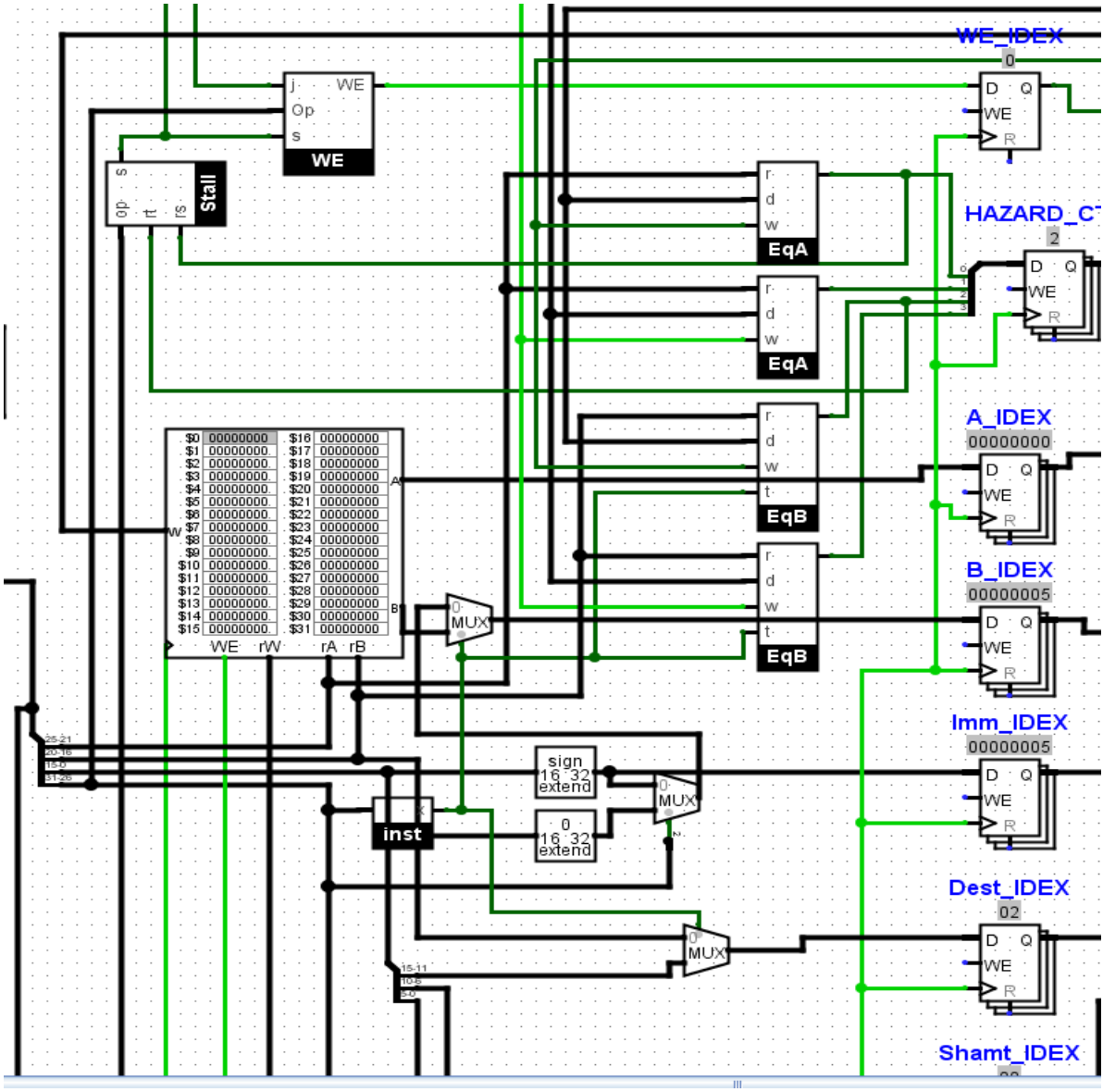
To increment the PC counter by 4, we add 4 to the first 30 bits and leave the last two unchanged

4.2 Correctness Constraints

This function of fetch is to use PC to index Program Memory and increment PC by 4 after each instruction, only when the clock is high.

5. Decode

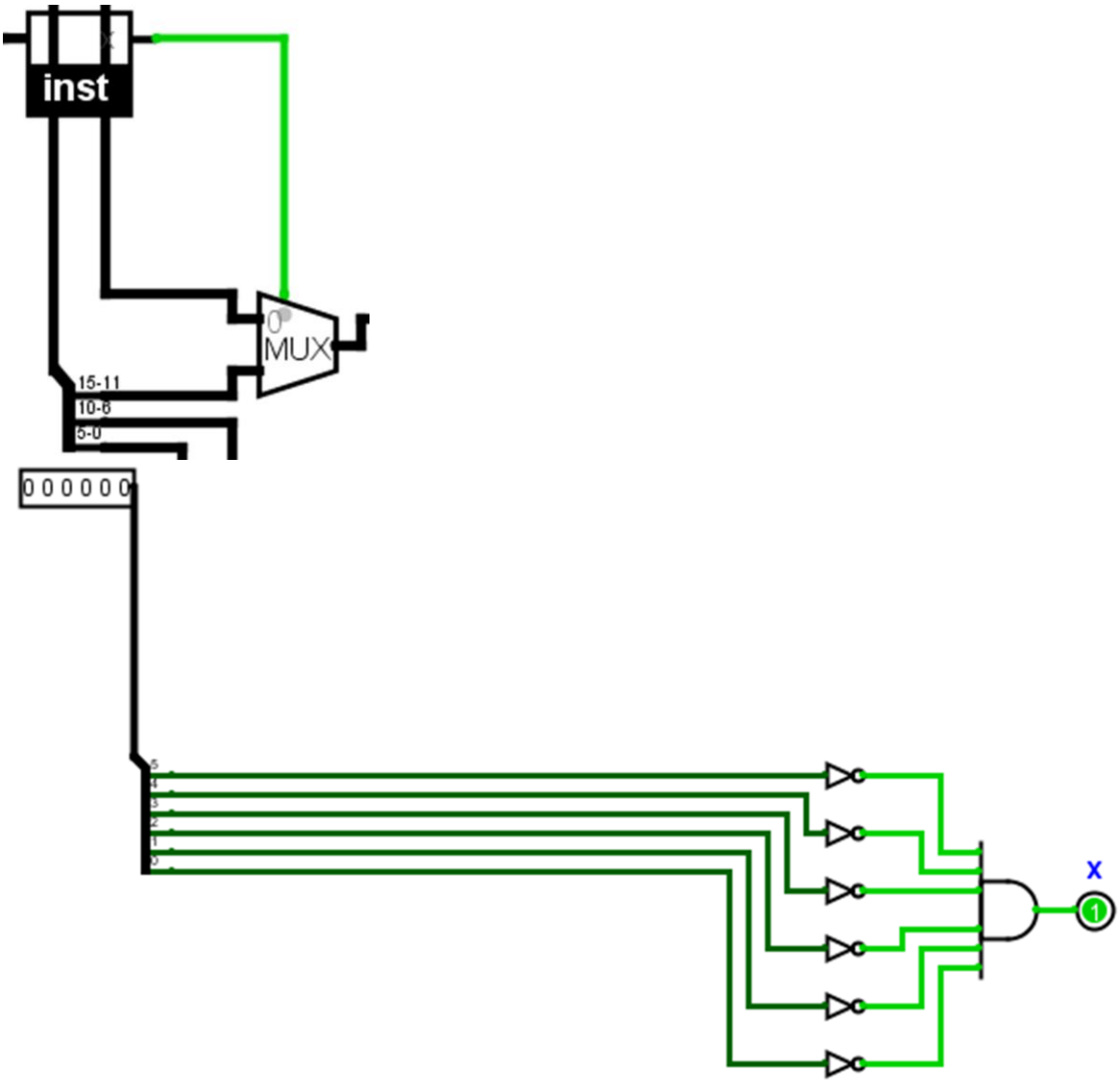
5.1 Circuit Diagram



The decode stage takes the values from the instruction memory and decodes each based on the instruction type. We also detect forwarding and stalling here (which will be explained in the hazard control part of the document.)

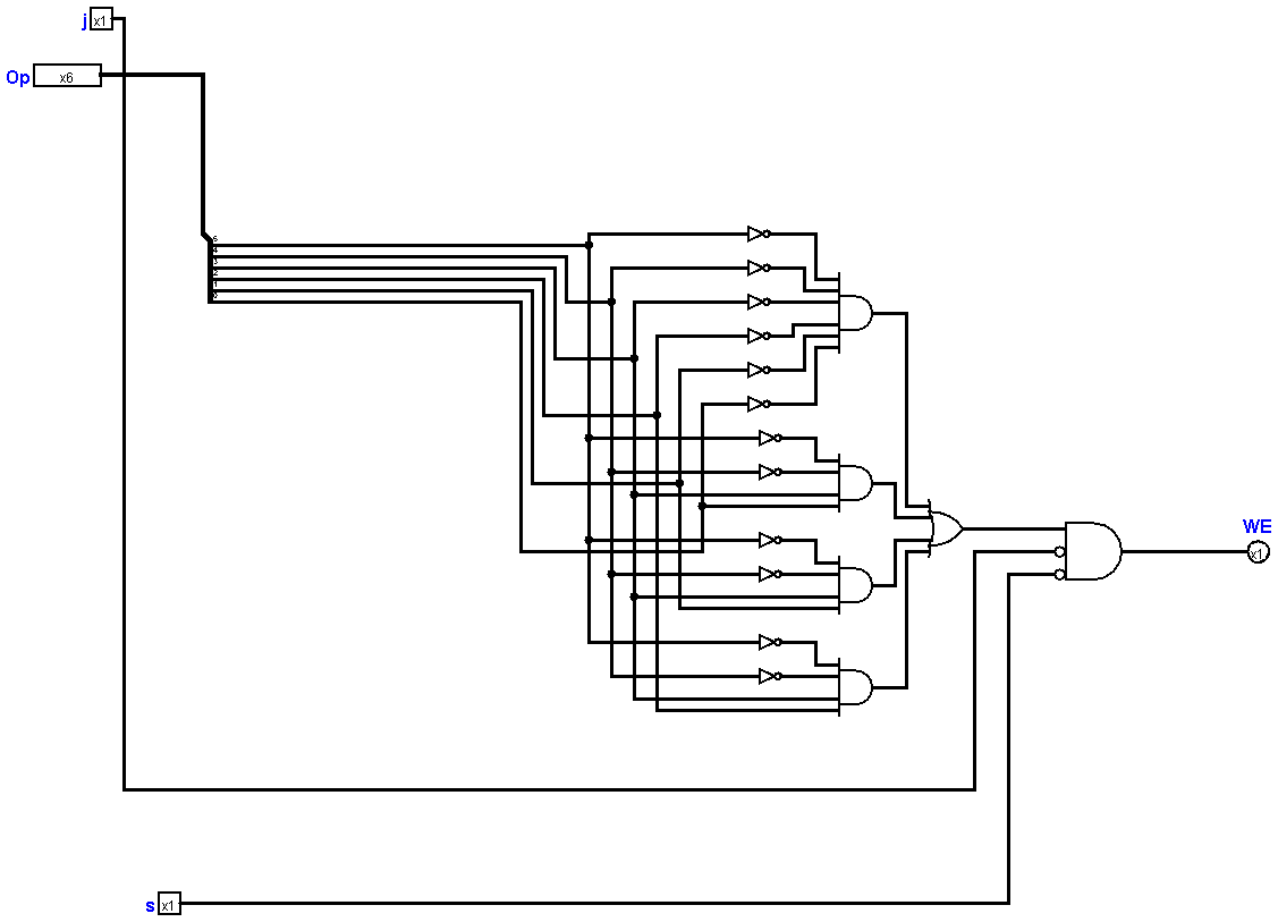
Here we split the data bits for every instruction such that we get the rs, rt, rd, the immediate (which is then sign extended), the shift amount, the function code, and the opcode, even if the instruction does not require such information.

- Destination Control



The inst control bit is used to determine if the given instruction is an R-Type or not. We classify all R-Types as having an opcode of 000000. If this is the case then the destination from instruction is in the bits 15 -11 of the instructions. Otherwise it must be an I-type, which is in bits 20-16 of the instruction.

- Write Back



The WE determines if the write enable of the register file should be high or low. If WE is 1, then data will be written into the register file. If its 0, the opposite will occur.

- **Correctness Constraints**

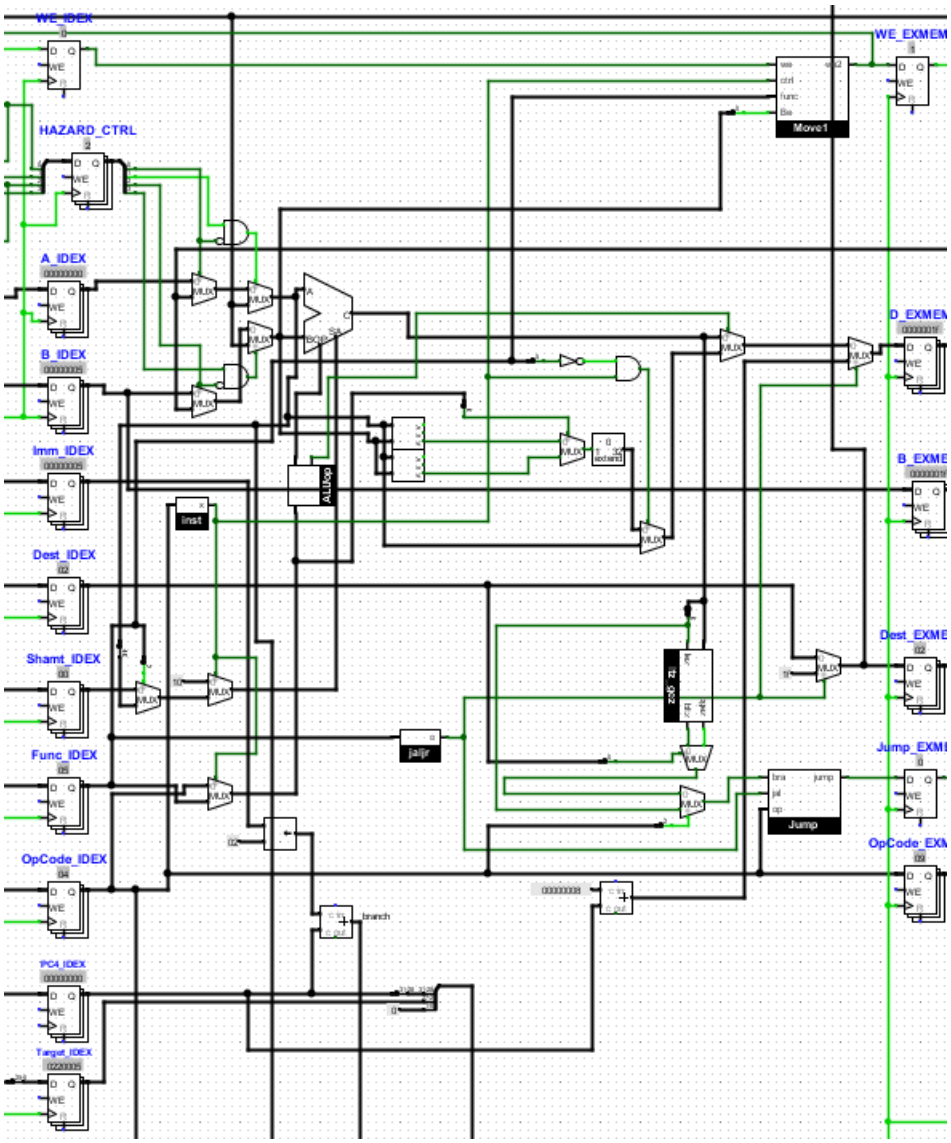
Decode should decode instructions, generate control signals, read register files.

- **Testing**

To test we used probes to ensure that each of the instructions were separated and distributed to each latch.

6. Execute

6.1 Circuit Diagram



The execute stage uses the information given from the decode stage, which was sent through the IDEX latch, and operates on the values. Almost all of the instructions use the ALU, so we use it to operate on the A and B

8

values to give us the 32 bit output of the decoded operation. If the result does not come from the ALU it comes from one of the two comparators (the top one for signed comparisons, the lower one for unsigned), or it is just A itself (move). For move

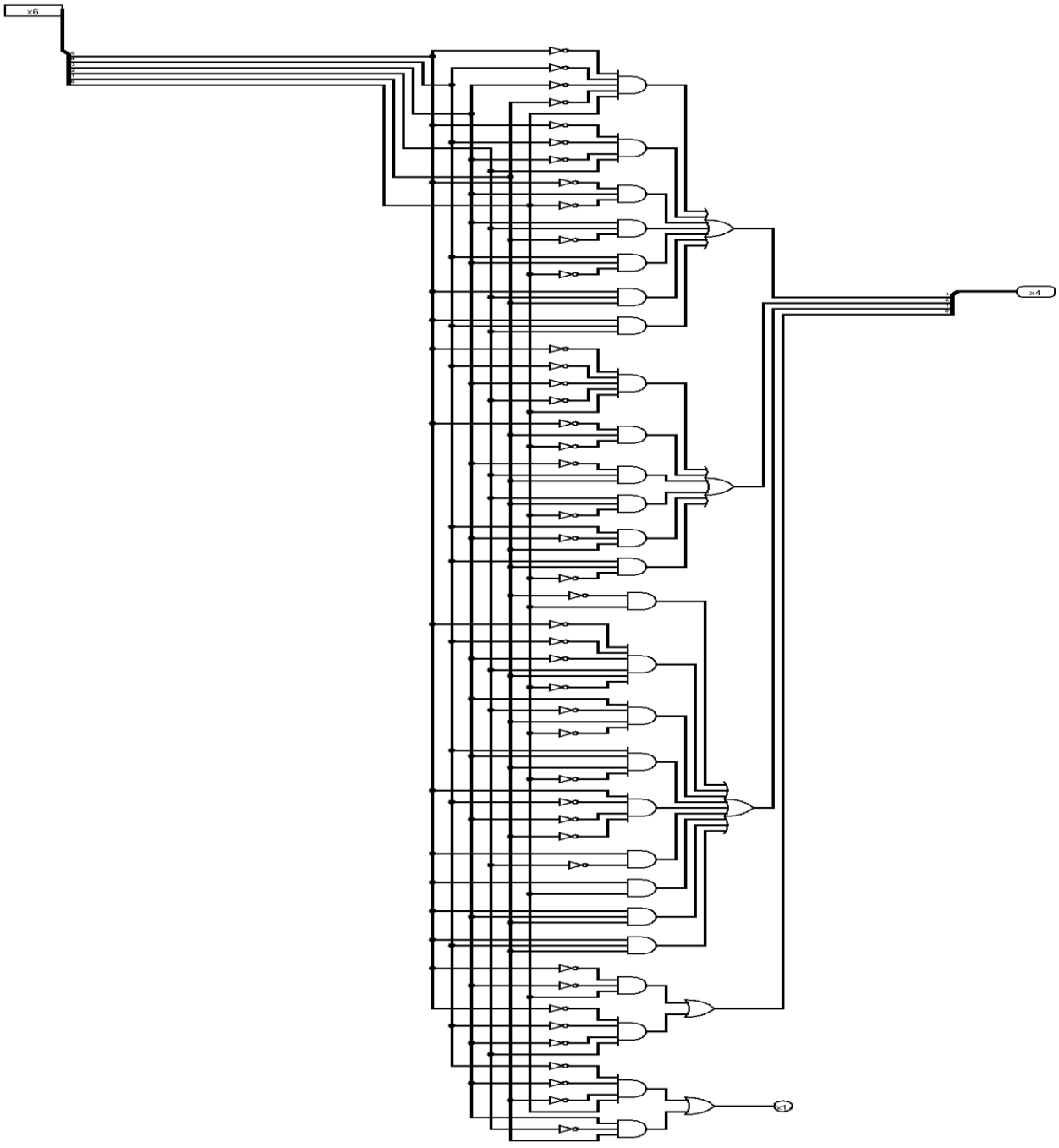
One of the main things determined here is whether the instruction will used the opcode or the ALU as the source of control. The inst control (described earlier) decides this.

To determine the shift amount (for shift operations only) we use the function code to choose between the shift amount decoded from the register file value, the lower 4 bits of A if shifting by a variable, or a constant of 16 if shifting by 16 bits.

To determine the A Value we choose between the values from the register, the value just executed from the ALU, the value in WriteBack. (Explained more in forwarding section)

The same goes for B, with the added addition of having to choose between B and the immediate.

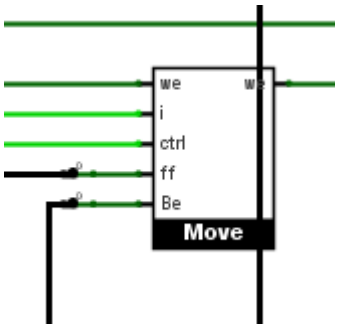
6.1.1 ALU Control

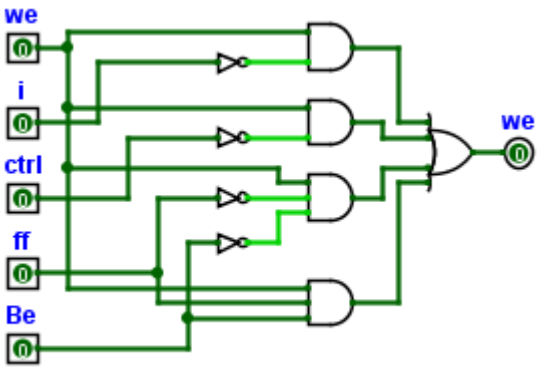


This uses either the op code or the function code outputs the 4 digit ALU OpCode that corresponds with the given instruction.

The second output determines(v) determines whether the output should come from the ALU(v=0) or not(v=1).

6.1.2 Move





Move takes the write enable bit, i (which determines whether or not it's a move instruction), ctrl (which comes from the inst control described above), the lsb of the function code, and the lsb of the B value (we assume that the value of B will either be 0 or 1 since it is the output of a set less than operation) and outputs the value of the write enable bit.

We input the following truth table in logisim:

we	i	ctrl	ff	Be	we2
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	0	1	0
0	1	1	1	0	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	0	1	1
1	0	0	1	0	1
1	0	0	1	1	1
1	0	1	0	0	1
1	0	1	0	1	1
1	0	1	1	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	0	1	1
1	1	0	1	0	1
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	1

If the WE is low, output 0.

If the WE is high, as well as I and ctrl, then output 0 only if ff=0 and Be=1 because that means the MOVZ condition has failed or if ff=1 and Be=0 because that means the MOVEN condition has failed and the rs value should not be moved.

Otherwise, output 1 if WE is high.

2. Correctness Constraints

Execute should perform the desired operation given an instruction

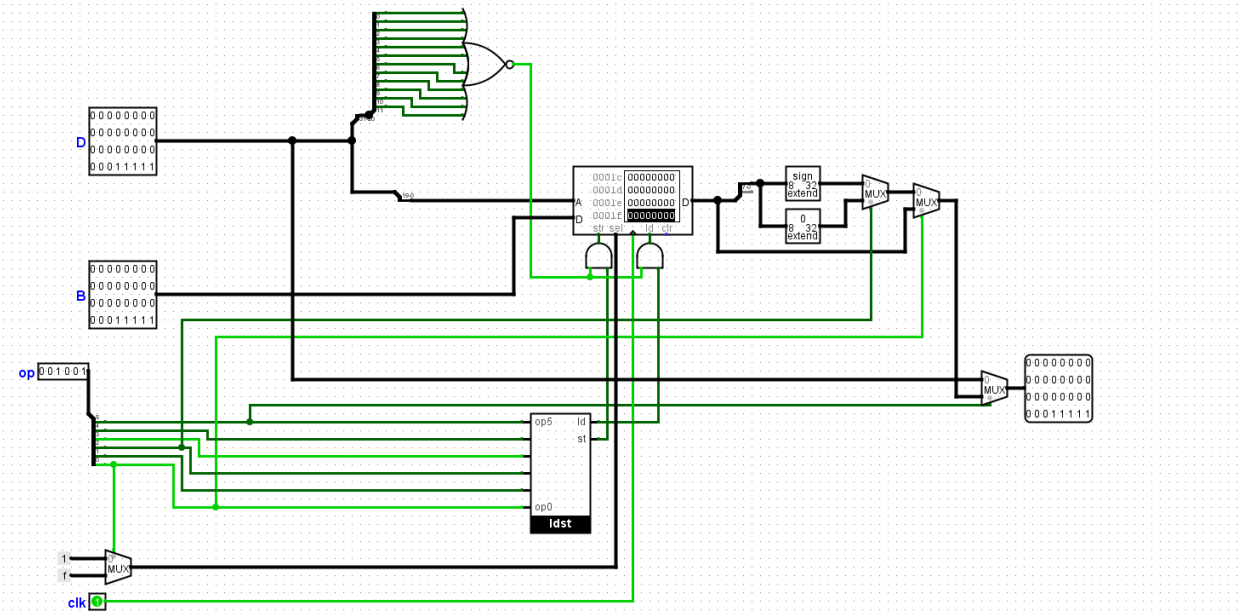
3. Testing

To test, handwrote test codes to see if the ALU was executing the correct operations or not at the correct time.. (See test file)

7. Memory and Write Back

7.1 Circuit Diagram

Memory
The memory stage takes in value from the ALU, memory address, the opcode and the controls. The sub circuit for the whole memory operation looks like this:



In the sub circuit, we made two constants; OX1 and OXf. These two determine if the operation is Word or a Byte. From the Opcode, the difference between word addressed and byte address instructions is the least significant bit and therefore this determines the value to be selected by the multiplexor. If the instruction is a LB or LBU, the multiplexor after the memory output decides if it sign extends or zero extend if it is LB and LBU respectively.
Then, the final multiplexer determines between the data from and the Memory and the unchanged data from the ALU. If the Opcode is LB, LBU or LW, then the selected instruction is from the memory, else it the data from the ALU

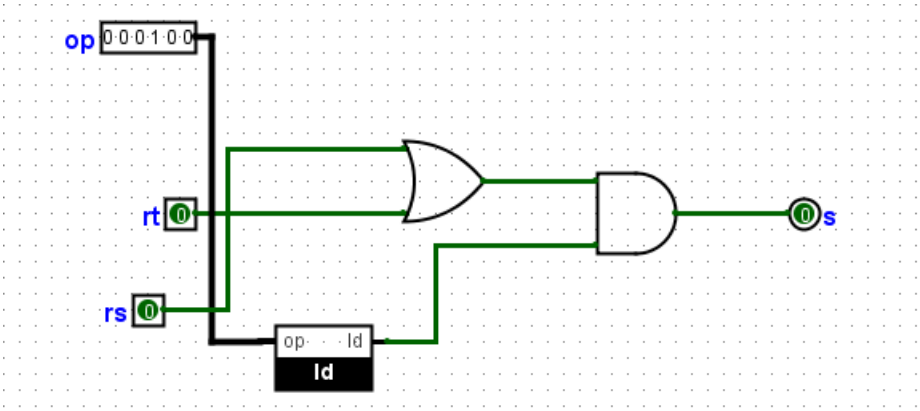
Hazard Detection

(i) Control hazards

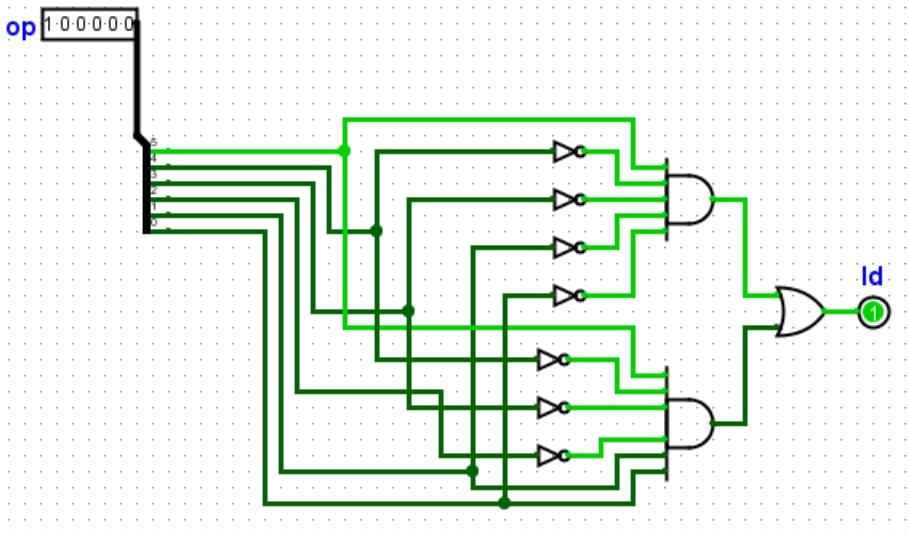
The control hazards are taken care of by the WE sub circuit which disables register write control in decode stage if the instruction is branch or jump.

(ii) Load-use hazards are detected in decode stage by the **stall** sub circuit.

If the stall output is 1, then the PC selects the previous PC value instead of PC+4 which allows the instruction to be executed again. The stall output also makes the write enable signal 0 and therefore not allowing the instruction after LW to update the register.



ld sub circuit in **stall** checks if the opcode is LB, LBU or LW and therefore acting as one major condition for detecting if load-use occurred. The rt and rs input are the outs of forwarding unit which checks the destination register used in the next instruction has been changed but the value has not been updated in the register and that the destination register is not 0.



The control hazards are taken care of by the WE sub circuit which disables register write control in decode stage if the instruction is branch or jump.

We do not use program memory in this project so did not include it. The In WB, the registers will write back to the register file if WE is enable, using the destination register, and the D value.

8. Forwarding

Forwarding logic occurs throughout most of the circuit.

We implemented forwarding to prevent data hazards in the

execution stage. To prevent data hazards in the execute stage, we

forward:

- only if the forwarding instruction will write to a register
  EX/MEM.Regwrite, MEM/WB.Regwrite
- AND only if Rd for that instruction is not $zero
  EX/MEM.RegisterRd != 0, MEM/WB.RegisterRd != 0

To prevent data hazards from the mem/wb stage, we forward:

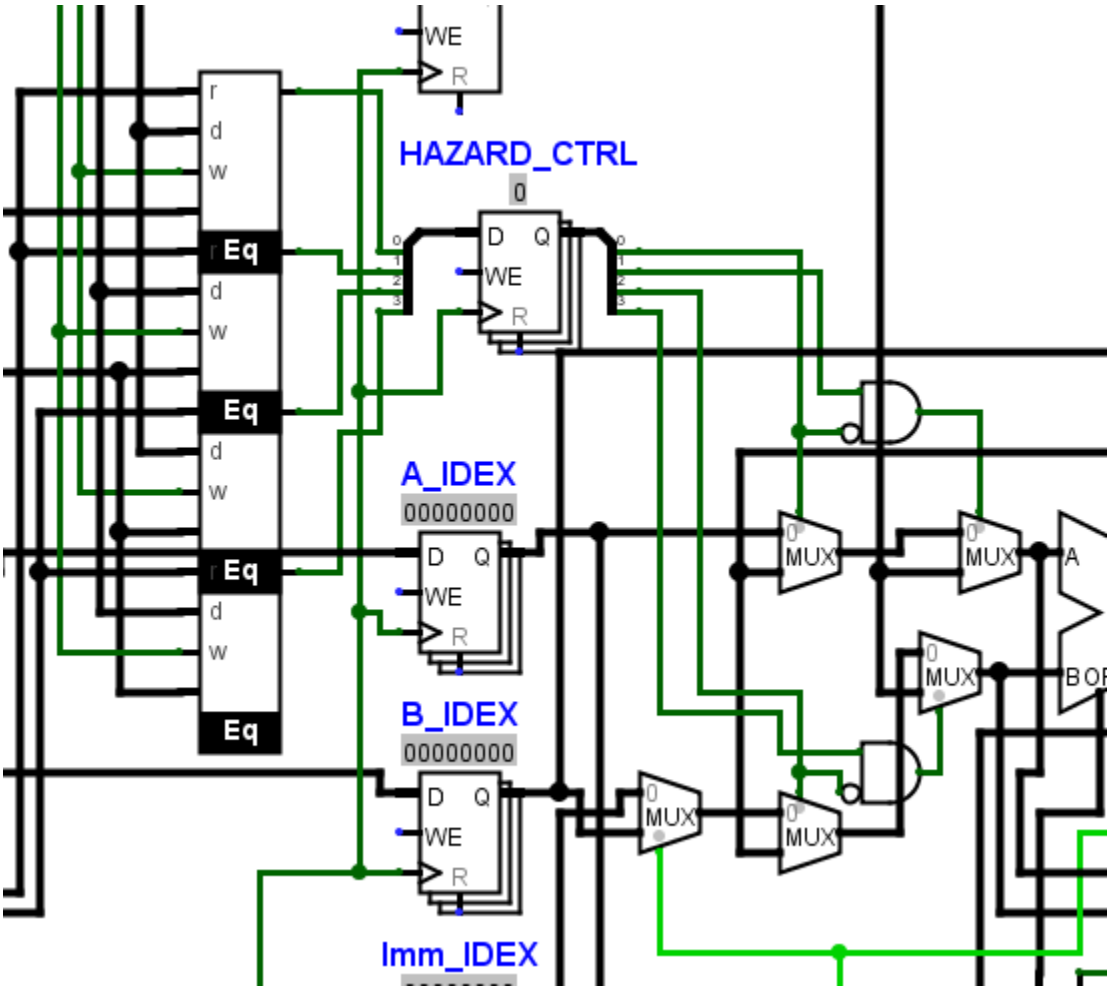
- only if the forwarding instruction will write to a register
  EX/MEM.Regwrite, MEM/WB.Regwrite
- AND only if Rd for that instruction is not $zero
  EX/MEM.RegisterRd != 0, MEM/WB.RegisterRd != 0

```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
```



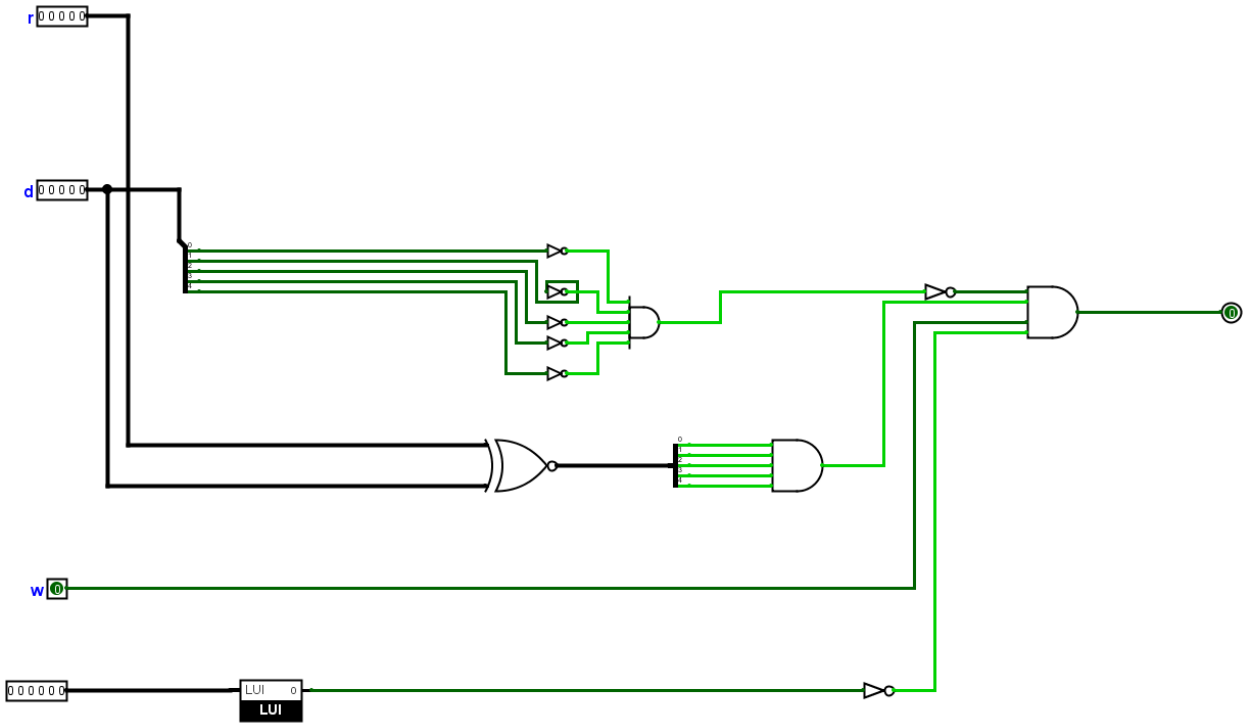
```
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
  and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
    ForwardA = 01
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
  and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
    ForwardB = 01
```

To ensure this we use the EQ sub circuit:



The EQ sub circuits determine whether or not to use the forwarded value in the execute stage. The first EQ (top) represents forwarding A as it's in Memory. The one below it represents forwarding A as it's in Write Back. If both signals are low (00) A is the value read from the register file. If the first is high and the second is low
(10) then that was just executed and is now in mem stage is forwarded. If the first one is low and then the second one is high (01) then the value from Write back is forwarded ad the A input. If they are both high, then we will take the input that's in the Mem stage because it represents the most recent value of A.

The same happens for B except that the value of B must first be decided if it's an immediate or the value from the register.

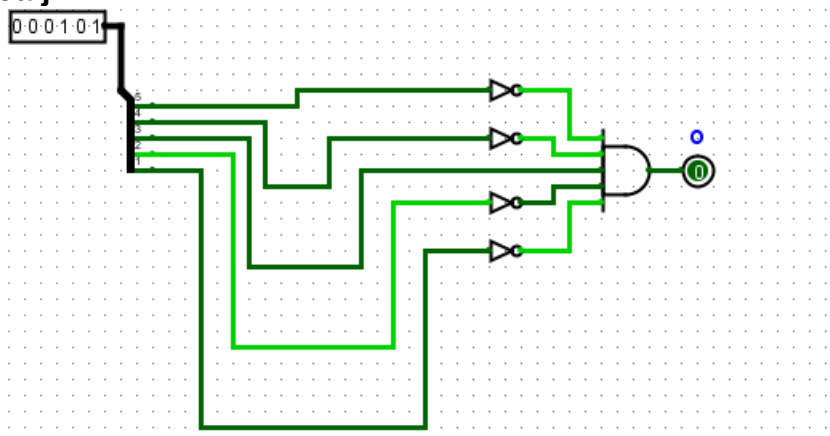


EQ checks more than if the two register locations are equal. It also checks if the destination register is 0 which should not be forwarded. It also checks if both operations are LUI operations because we don't want to forward LUI operations to each other if they are writing to the same register.

Jump Instructions and Branch

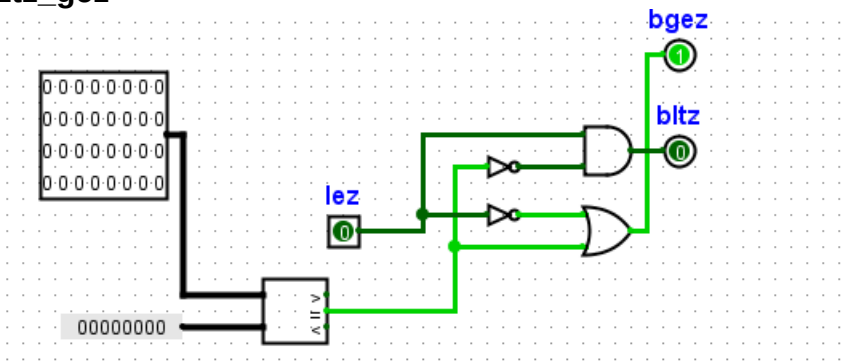
Branches and Jumps are detected in execute stage. We use the ALU to calculate the correct memory offset and to determine whether we are stalling or disabling the controls.
The following sub circuits aids is branch and jump instruction

Jaljr



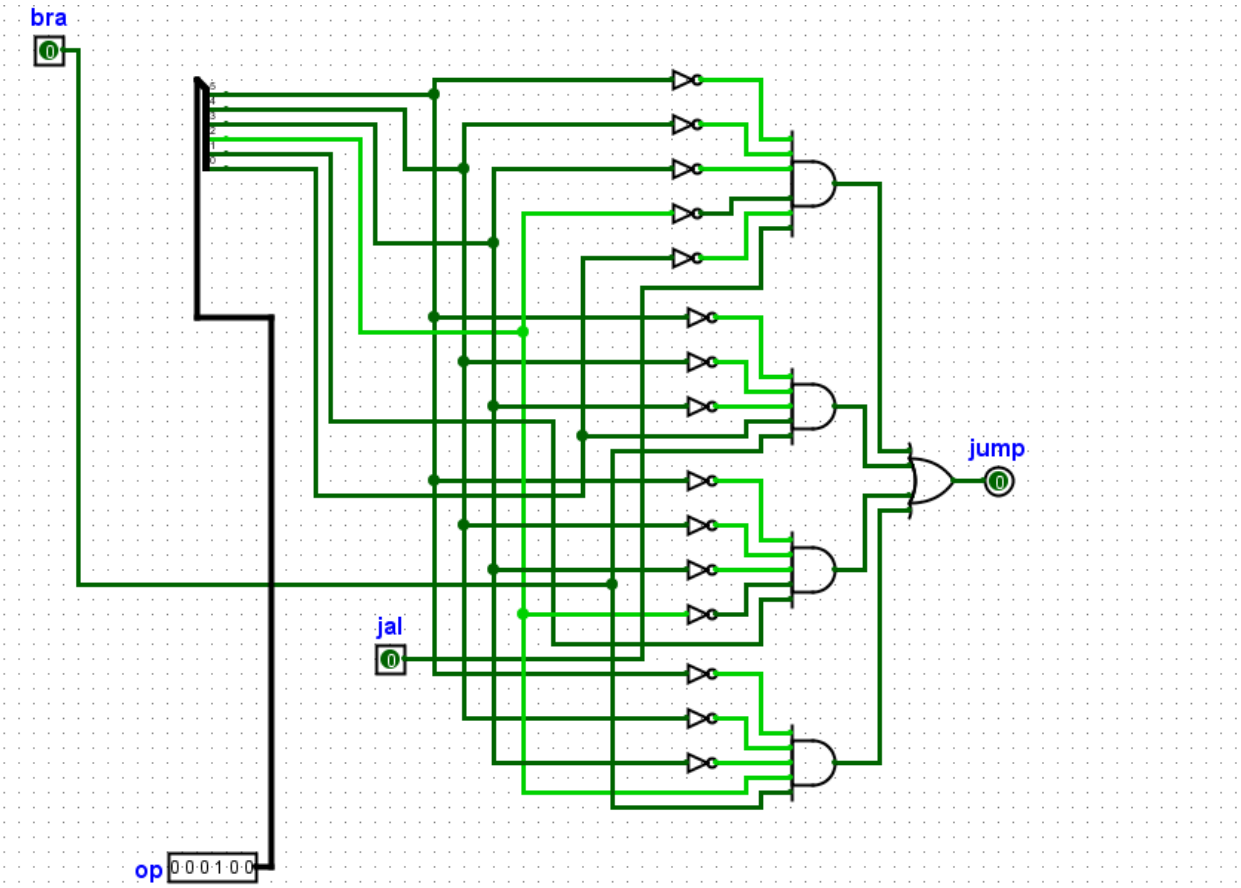
This sub circuit determines if an instruction is a JR or JAL for proper linking. If it is either, it outputs 1 and 0 otherwise.

Ltz_gez



Takes a 32 bit input from the ALU which is the value from register rs and the comparator checks if the input is greater or equal to zero. The outputs are one bit indicators; bgez and bltz which determines if branch happens or no.

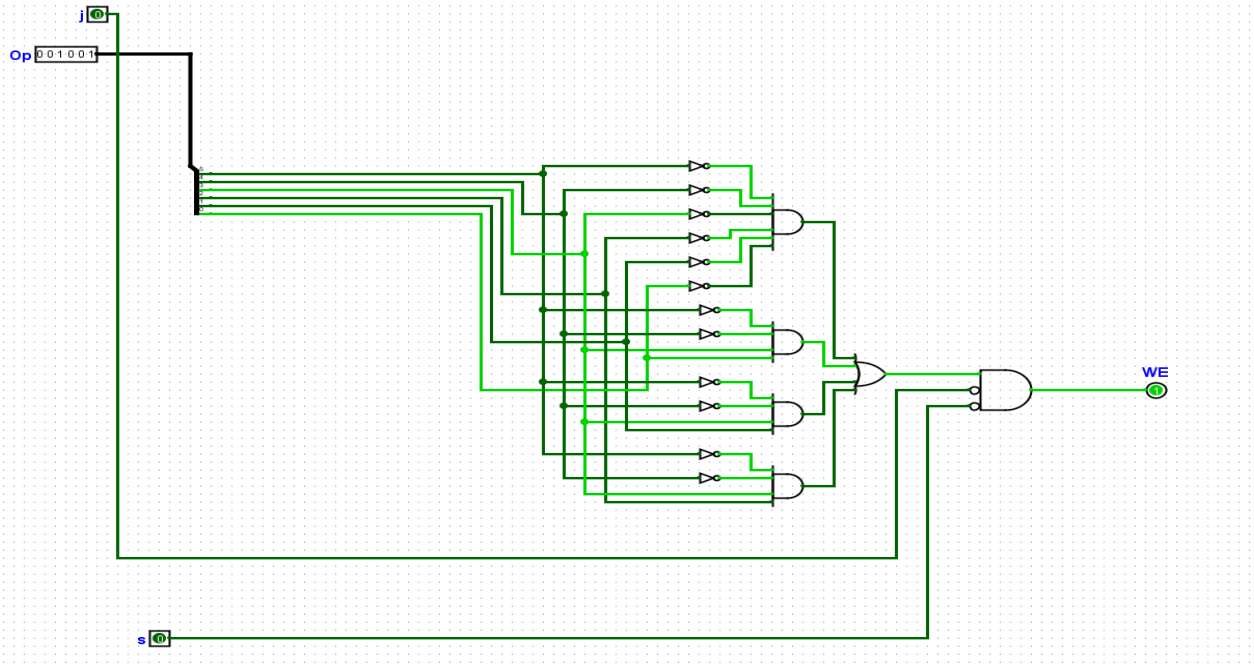
Jump



It takes as inpute the Opcode and the results from the **jaljr** and **ltz_gez.** The opcode will determine if the instruction is jump and then the one bit output will be used to decide if to or not to set the instruction in Program ROM to nop.

DELAY SLOT.

The delay slot is applicable if an instruction is a jump or branch and also when stalling is enabled. We use the result from **jump** sub circuit as input which covers both branch and jump cases. The other input is the stall signal and the opcode. The sub circuit **WE** determines if the dela should happen:



The opcode enabls writing t the register. The other two inputs acts as parameters to ensure that the instruction is not stalled (**s)** or branch or jump (**j)**

TESTING

We tested our processor in variety of ways. First, we wrote the assembly code test cases which cover all the instructions in Table B that we didn't cover in the project one. We made sure to test all the possible cases. For example, in branches and jumps, our MIPS code test cases cover the situation when a branch condition is met and the situation where it not met for each branch instruction. In addition, we made test cases to cover the memory instructions and the associated hazards. First, we cover the general memory instructions to ensure that the memory stores and loads as expected. Then we made instructions specific for testing if SB saves as little endian and LB, LBU loads the little endian.

The test cases also checks the likely load-use and control hazards. The way we ensured this instructions work is we load the MIPS code into the program ROM and following the data changes in each cycle in the pipeline to the data updated in the register to ensure that the circuit works correctly.

Besides the test cases, we use the hailstone program to help test the circuit and checking the register changes with our expected values.