Project 1: Design Documentation

Yogisha Dixit (yad4) Nicholas De Tullio (nrd24)

March 11, 2014

1 Introduction

This document is intended to describe the planned implementation of a MIPS pipeline processor with a simplified instruction set. There is information on design, testing, and the correctness constraints of each subcomponent and the overall circuit included. The intended audience is anyone familiar with the assignment and who has a level of understand comparable to the information learned in CS3410 at Cornell University. References include *Computer Organization and Design* by David Patterson and John Hennessy, *MIPS32 Architecture for Programmers*, Volumes 1,2, and 3 by MIPS Technologies, and slides and lectures from CS3410 at Cornell University, Spring 2013. This document contains information on the implementation of a mini MIPS pipeline processor, including diagrams, control logic, correctness constraints, and information on testing the processor.

Common abbreviations:
Mux: Multiplexer
PC: Program Counter
ALU: Arithmetic Logic Unit

2 Overview

We implement the pipeline processor by splitting it into 5 stages: Fetch, Decode, Execute, Memory, and Write Back. The Memory stage and Jump/Branch instructions are not going to be implemented during this stage of the project. The Fetch stage will get the instruction and update PC as necessary. The Decode stage will determine what action we will perform with the instruction, split it into different input values as appropriate, and pass relevent information onto the next stage. The execute stage will perform any non-Memory operations and pass the information on. The memory stage will, in the future, allow us to use instructions that require access to memory. For now, it is a dummy stage where information is just passed through unaltered. The Write Back stage will determine what we need to update in the register, if anything, and send the relevent register and output information back. Here is the overall circuit diagram. We provide more detail in each stage.

3 Transition Register Stages

There are four different substages that store the information between stages: ID/DE, DE/EX, EX/MEM, MEM/WB. They store the values in registers or in a d flip-flop if it is only 1 bit. Both kinds of memory storage devices use the rising edge as their trigger value. We include the diagram for only the EX/MEM stage in Figure 1, as the others are identical except for the values they store.

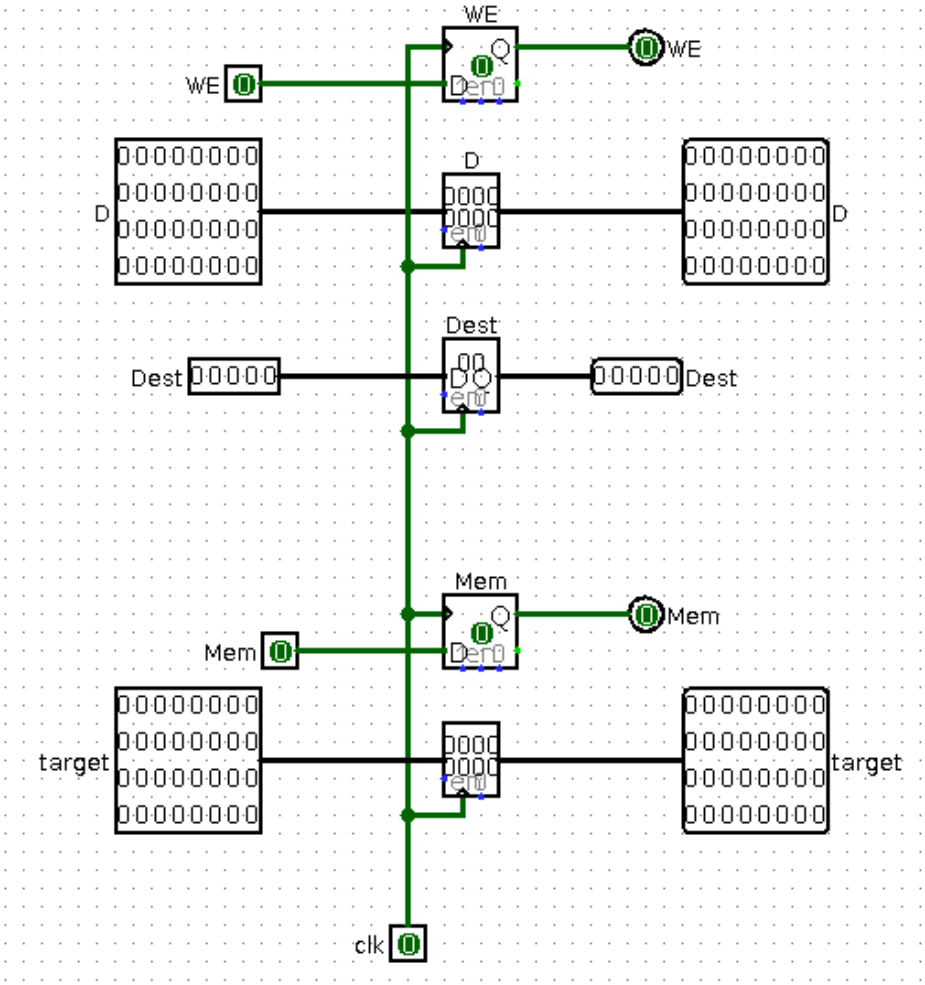

Figure 1: Ex/Mem Transition Register Stage

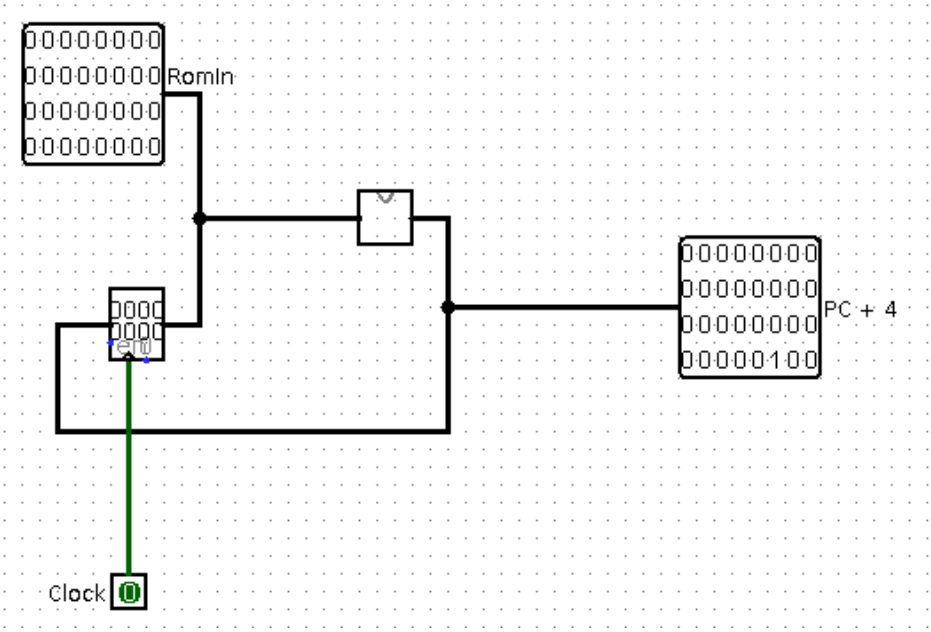

Figure 2: Fetch Stage

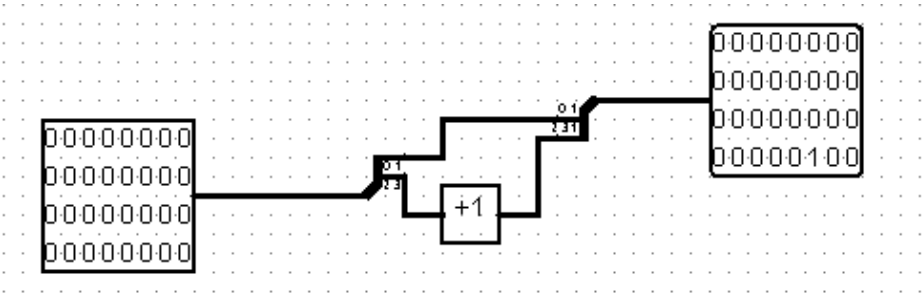

Figure 3: +4 Incrementer

4 Circuit Diagram

5 The Fetch Stage

Outputs: Instr[32],PC+4[32]
The Fetch stage use the current value of PC to retrieve the instruction from the program ROM and increments PC as necessary. PC+4 is the incremented value of PC and instr is the instruction at the address of the current PC in the program memory. It is shown in Figure 2.

5.1 Submodule: PC+4

This submodule increments the program counter by 4. It uses the incrementer provided in the CS3410 jar file. Since the provided incrementer only increments the value it is provided by 1, we split the 32-bit input into the 30 most significant bits and the 2 least significant bits. We feed the 30 bits into the incrementer and combine the result with the 2 bits in the same order as before. The circuit is shown in Figure 3.

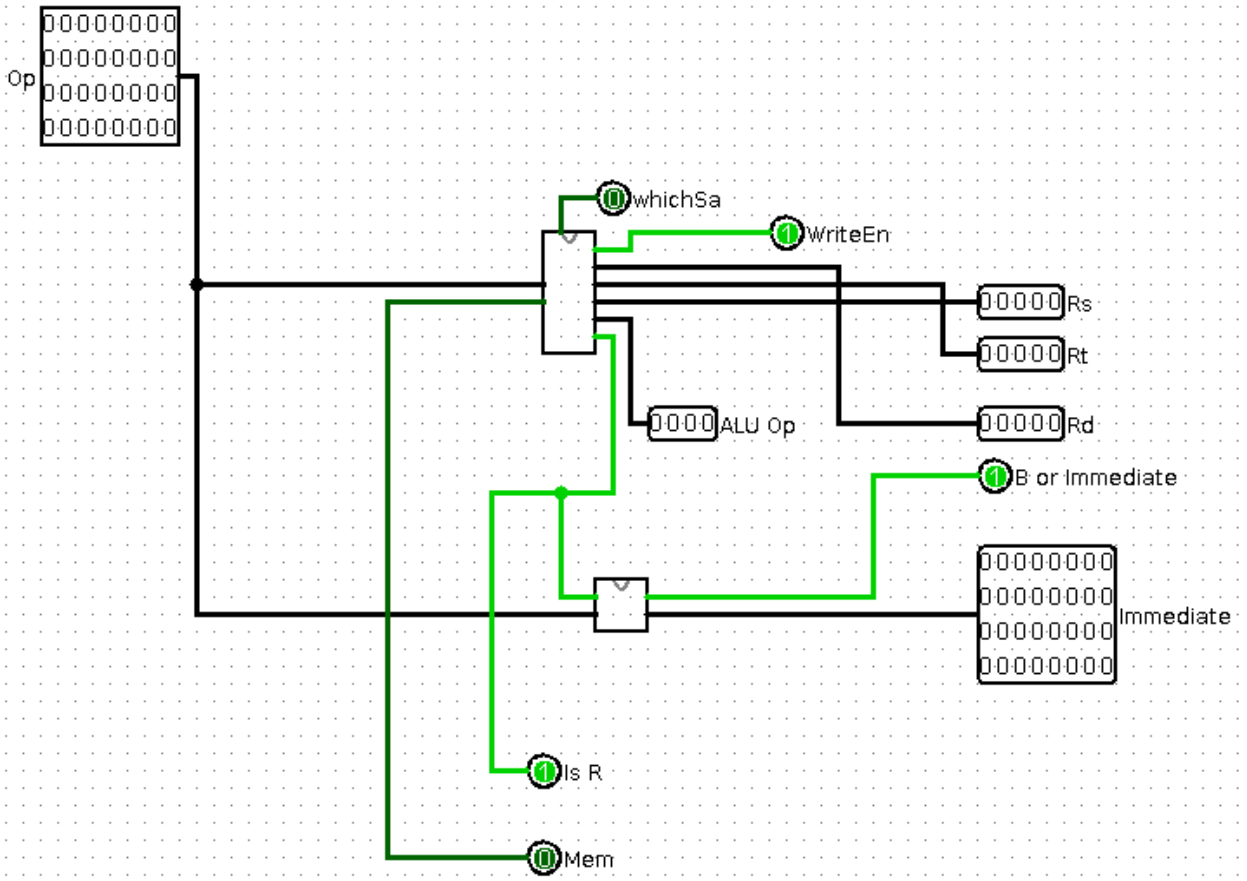

Figure 4: Top level decode stage

5.2 Correctness Constraints

The Fetch Stage must correctly get the instruction and update PC. Since we are not yet implementing Jumps and Branches, PC always increments by 4 each clock cycle. It should retrieve the instruction at the index of PC in the ROM.

6 The Decode Stage

Inputs: Op[32] Outputs: Mem[1], Is R[1], whichSa[1], WriteEn[1], B or Immediate[1], Rs[5], Rt[5], Rd[5], Immediate[32] The decode stage determines which register addresses we wish to access as well as if we wish to write to the register. This is where much of the control logic necessary for the execute stage is determined as well. It also extends the immediate value of I instructions here as necessary. Op is the instruction retrieved from the program ROM at the previous stage, Is R determines if the instruction is an R-type, WriteEn determines if we should enable writing to the register, B or Immediate determines if we should use the value at Rt in the register or the immediate part of the instruction, Rs, Rt, Rd are all register addresses, and Immediate is the immediate value of an instruction (its 16 least significant bits). All of the actual calculations are done in its two submodules. The top-level decode circuit is shown in Figure 4.

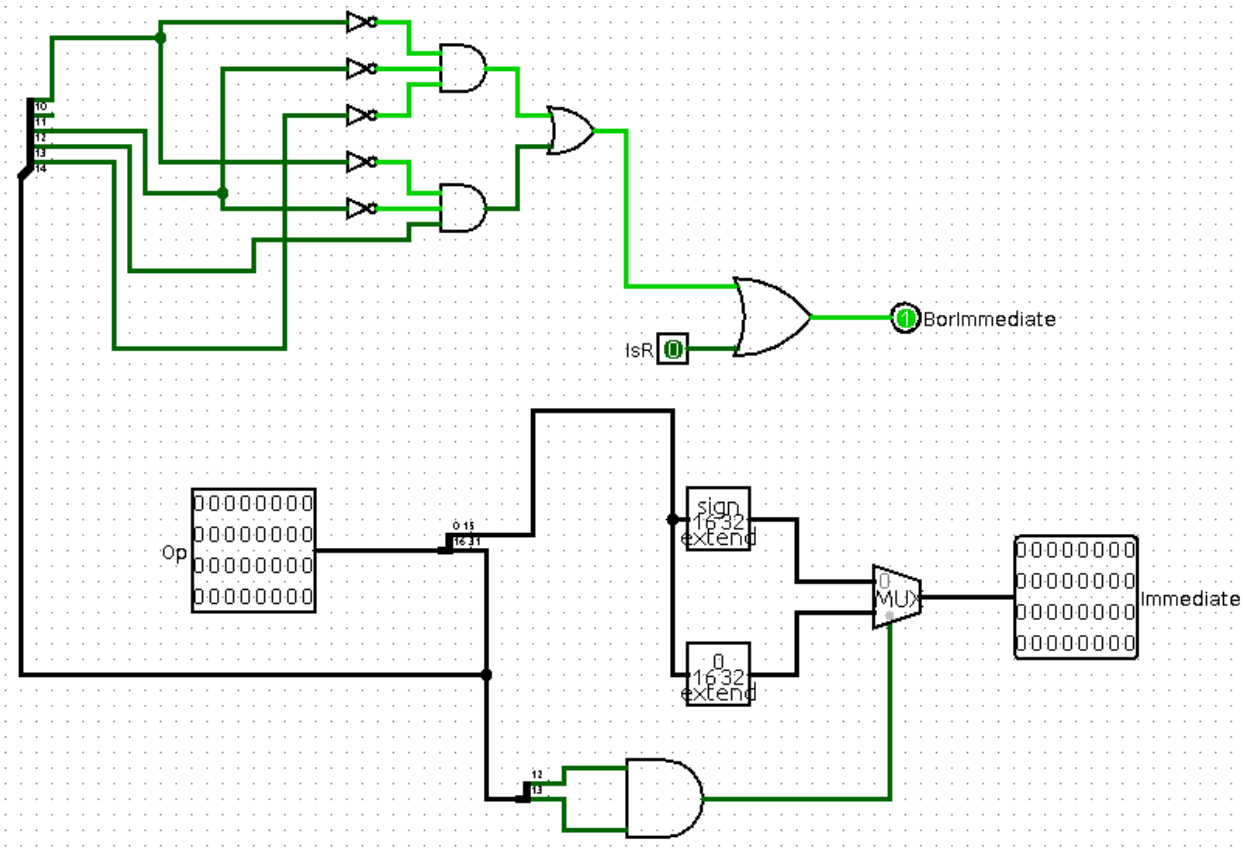

Figure 5: Immediate decode stage

6.1 Submodule: Immediate Decode

Inputs: Op[32]
Outputs: Immediate[32], B or Immediate[1]
This submodule determines how we should extend the immediate value as well as if we should use the B or Immediate value in the execute stage. The circuit is shown in Figure 5.

6.1.1 Extend Mux Logic

We only care about how we extend Immediate when we have I type instructions, as they are the only ones to use this value. The value that we choose to extend by depends on the specification of the instruction. We present a table with the significant values below, all other values are treated as don't cares. The Mux chooses zero extend if 1 and sign extend if 0. Each instruction is an I-type presented by its Op value (it's first 5 most significant bits).

Op5	Op4	Op3	Op2	Op1	Op0	Mux
0	0	1	0	0	1	0
0	0	1	1	0	0	1
0	0	1	1	0	1	1
0	0	1	0	1	0	0
0	0	1	0	1	1	0
0	0	1	1	1	0	1
0	0	1	1	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	0	0
0	0	0	0	0	1	0
0	0	0	1	1	1	0
0	0	0	0	0	1	0
0	0	0	0	0	1	0
1	0	0	0	1	1	0
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	1	0
1	0	1	0	0	0	0

So the logic equation is $mux = Op_4 Op_4$.

6.1.2 B or Immediate logic

B or Immediate is simply a marker for if we should use the B value of the Rt register or the immediate value of the instruction, as specified by the instruction itself. We encode B or Immediate as follows: B is 1 and Immediate is 0.

The following truth table indicates what this value should be set to based on the instruction. Note that all R-types use the B value, while most, but not all, I-types use the immediate value. As a result, the truth table only contains op codes for immediate instructions.

Op5	Op4	Op3	Op2	Op1	Op0	B or Immediate
0	0	1	0	0	1	0
0	0	1	1	0	0	0
0	0	1	1	0	1	0
0	0	1	0	1	0	0
0	0	1	0	1	1	0
0	0	1	1	1	0	0
0	0	1	1	1	1	0
0	0	0	1	0	0	1
0	0	0	1	0	1	1
0	0	0	1	1	0	1
0	0	0	0	0	1	1
0	0	0	1	1	1	1
0	0	0	0	0	1	1
0	0	0	0	0	1	1
1	0	0	0	1	1	0
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	1	0
1	0	1	0	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	0	1
0	0	0	0	1	1	0

Since we have th input value isR to tell us if an instruction is R-type or not (this is calculated in the non immediate decode stage that will be explained below), we can just look at the I-type instructions that use B instead of immediate to finish the logic. Then the logic equation is then $BorImmediate = IsR + \bar{(}Op_5 Op_3)$.

6.2 Submodule: Non Immediate Decode

Non Immediate Decode deals with the logic that isn't related to the immediate value. It parses the register locations from the instruction, which is done with a splitters. The circuit diagram is shown in Figure 6.

6.2.1 Control for ALU

It also determines the control value for the ALU. This is done with two ROMS. The upper one deals with R-type instructions and the lower one deals with I-type instructions. The R-instruction ROM is addressed by $Func_5, Func_3, Func_2, Func_1, Func_0$, where Func is the function code of the R-type instructions, as this determines them. We noticed $Func_4$ is always 0 in our instruction set, and so we don't need it. The I-type ROM is the same, except it uses the Op part of the instruction.

The mux to control that chooses between these ROMs is based on whether or not the instruction is an R-type. Note that isR=$Op5 Op4 Op3 \bar{Op}2 Op1 Op0$ since all of our R type instructions start with all 0's in their Op code. As a result, Control = isR.

6.2.2 Jump Instructions

Even though we ignore jump instructions in this project, we still need to decode them. Based on whether they are I-type instructions or R-type instructions, we utilize the op code or the function code in the 32 bit instruction. As a result, the instruction is a register jump if $Func5 \bar{Func}4 Func3 Func2 \bar{Func}1$. On the other hand, it is an immediate jump if $Op5 Op4 \bar{Op}3 Op2 Op1$.

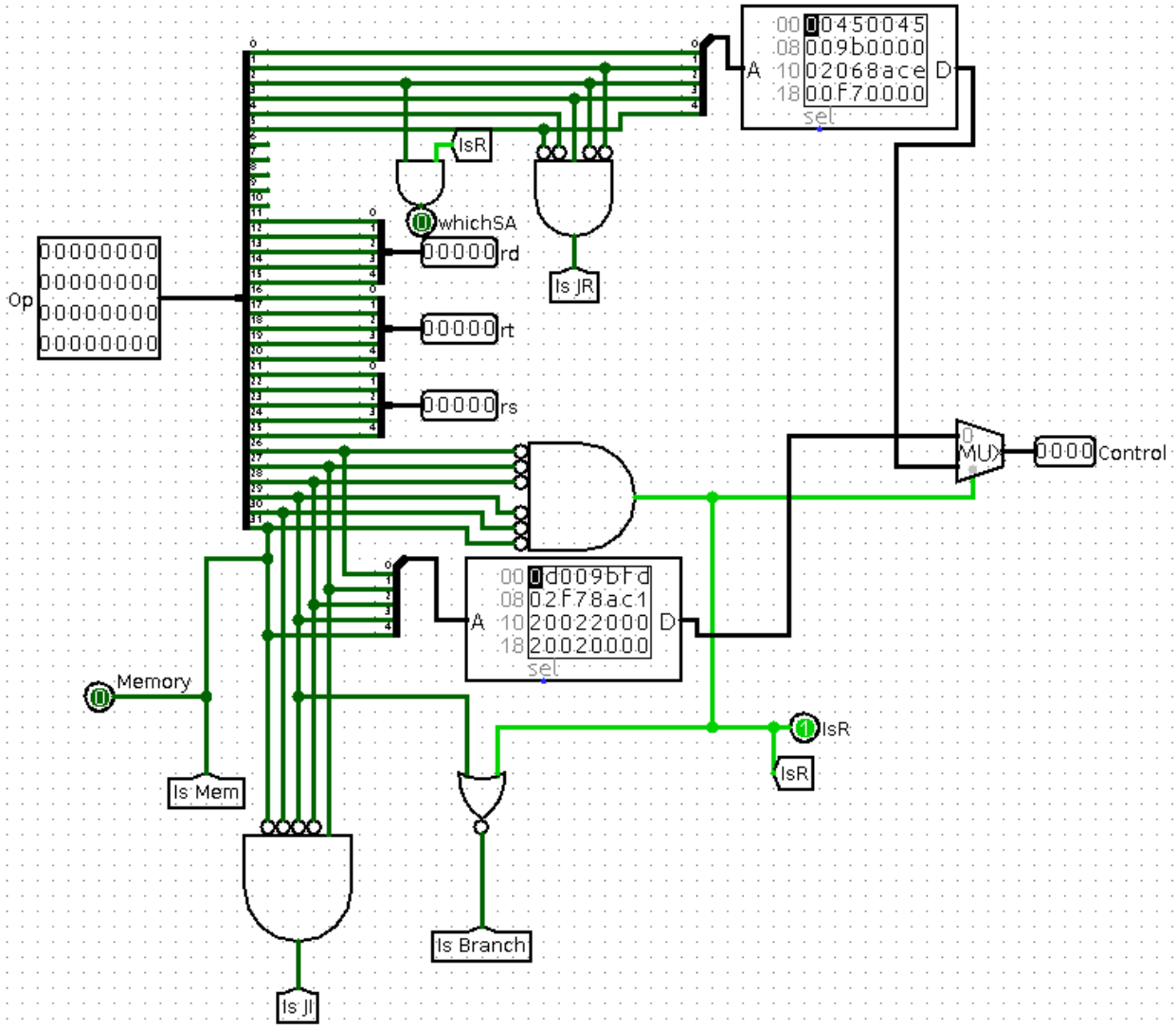

Figure 6: Non Immediate Decode Stage

6.2.3 Memory Instructions

This value just determines if the instruction needs to access memory. It is defined by the instruction's Op value. We set Is Mem to 1 when we get a memory instruction and 0 otherwise.

Op5	Op4	Op3	Op2	Op1	Op0	Memory
0	0	1	0	0	1	0
0	0	1	1	0	0	
0	0	1	1	0	1	0
0	0	1	0	1	0	0
0	0	1	0	1	1	0
0	0	1	1	1	0	0
0	0	1	1	1	1	0
0	0	0	1	0	0	1
0	0	0	1	0	1	1
0	0	0	1	1	0	1
0	0	0	0	0	1	1
0	0	0	1	1	1	0
0	0	0	0	0	1	0
0	0	0	0	0	1	0
1	0	0	0	1	1	1
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	1	1
1	0	1	0	0	0	1

So Memory=$Op5$.

6.2.4 WhichSa

WhichSa is based on the fact that sometimes we need the value in a register to use as the shift amount for certain commands. The truth table for these commands is presented below, based on their function code, as only R-types are shift instructions. We set WhichSa to 0 when we use the immediate Sa value and 1 if we have to grab it from a register.

Func5	Func4	Func3	Func2	Func1	Func0	WhichSa
0	0	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	1
0	0	0	1	1	0	1
0	0	0	1	1	1	1

So then whichSa=$func2$.

6.2.5 Write Enable

This value determines if Write Enable is activated or not. Writing to the register file is disabled for jump, branch, and memory commands, effectively turning them into nops for this project. As a result, we set Enable to 0 when we get a jump, branch, or memory instruction and 0 otherwise. A circuit diagram is given in Figure 7.

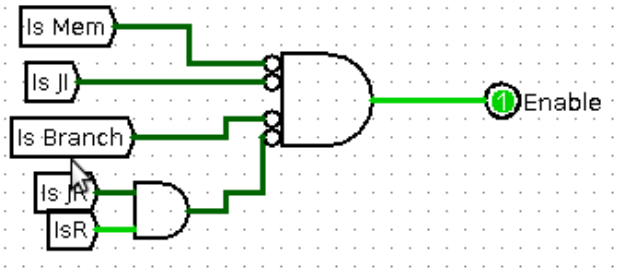

Figure 7: Write Enable Circuit

Op5	Op4	Op3	Op2	Op1	Op0		func5	func4	func3	func2	func1	func0	Mux
0	0	0	0	0	0	...	1	0	0	0	0	1	1
0	0	0	0	0	0	...	1	0	0	0	1	1	1
0	0	0	0	0	0	...	1	0	0	1	0	0	1
0	0	0	0	0	0	...	1	0	0	1	0	1	1
0	0	0	0	0	0	...	1	0	0	1	1	0	1
0	0	0	0	0	0	...	1	0	0	1	1	1	1
0	0	0	0	0	0	...	1	0	1	0	1	0	1
0	0	0	0	0	0	...	1	0	1	0	1	1	1
0	0	0	0	0	0	...	0	0	1	0	1	1	1
0	0	0	0	0	0	...	0	0	1	0	1	0	1
0	0	0	0	0	0	...	0	0	0	0	0	0	1
0	0	0	0	0	0	...	0	0	0	0	1	0	1
0	0	0	0	0	0	...	0	0	0	0	1	1	1
0	0	0	0	0	0	...	0	0	0	1	0	0	1
0	0	0	0	0	0	...	0	0	0	1	1	0	1
0	0	0	0	0	0	...	0	0	0	1	1	1	1
0	0	0	0	0	0	...	0	0	1	0	0	0	0
0	0	0	0	0	0	...	0	0	1	0	0	1	0
0	0	1	0	0	1	...	x	x	x	x	x	x	1
0	0	1	1	0	0	...	x	x	x	x	x	x	1
0	0	1	1	0	1	...	x	x	x	x	x	x	1
0	0	1	0	1	0	...	x	x	x	x	x	x	1
0	0	1	0	1	1	...	x	x	x	x	x	x	1
0	0	1	1	1	0	...	x	x	x	x	x	x	1
0	0	1	1	1	1	...	x	x	x	x	x	x	1
0	0	0	1	0	0	...	x	x	x	x	x	x	0
0	0	0	1	0	1	...	x	x	x	x	x	x	0
0	0	0	1	1	0	...	x	x	x	x	x	x	0
0	0	0	0	0	1	...	x	x	x	x	x	x	0
1	0	0	0	1	1	...	x	x	x	x	x	x	0
1	0	0	0	0	0	...	x	x	x	x	x	x	0
1	0	0	1	0	0	...	x	x	x	x	x	x	0
1	0	1	0	1	1	...	x	x	x	x	x	x	0
1	0	1	0	0	0	...	x	x	x	x	x	x	0
0	0	0	0	1	0	...	x	x	x	x	x	x	0
0	0	0	0	1	1	...	x	x	x	x	x	x	0

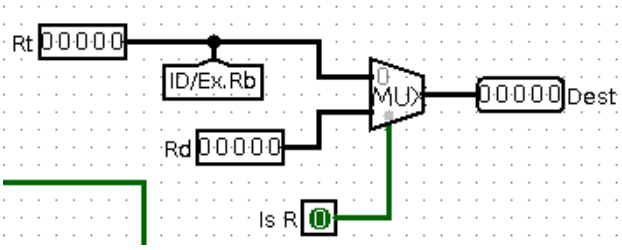

Figure 8: Destination Register Selection Circuit

6.3 Correctness Constraints

The Decode stage must correctly interpret the instruction into its relevent values and operations. This includes finding the correct vales for Rb, Ra, Rb and write enable in the register and correctly extending the immediate value of an I type instrution. It should also correctly determine important decode variables used in the execute stage, such as whether the instruction is an R-type and what we should use for the shift amount.

7 The Execute Stage

Inputs: A[32], Immediate[32], B[32], B or Immediate[1], Mem in[1], PC+4[32], WE[1], WhichSa[1]
Outputs:D[32], WE2[1], target[32], Dest[5] The execute stage performs any necessary computation for the commands that do not access memory. It contains an ALU to do this, as well as some comparators. A and B are the values at the register locations given by Rs and Rt, respectively, Immediate is 16 least significant bits of the instruction(its immediate field if I-type), B or Immediate determines whether we use B or Immediate in our calculation, Mem in describes whether or not the instruction uses memory, WE defines whether write enable should be turned off from information we parsed in the last stage, WhichSa defines what we use for the shamt in the ALU, and PC+4 is the possible address of the next instruction. D is then the output of the calculation, WE2 is whether or not we should allow writes to the register after doing our calculations in this stage, and Dest is the address of the register we wish to write to. Target is the location of a branch instruction, but it is simply PC+4 in this case since we are not implementing branch instructions.

7.1 Circuit Diagram

7.2 Control Logic

7.2.1 Rd Mux

This determines what value of Rd we send back to the register. It is the rt section if the command is an I-type, in which case the mux chooses register encoded by the the 21-17 bits, and the rd section of the instruction if it is an r-type, in which case the mux chooses the register enoded by 16-12 bits. Thus we use the previously calcuated value is R, which is 1 if an instruction is R-type to determine this. The circuit diagram is shown in Figure 8.

7.2.2 Immediate or B to ALU Mux

This determines if we send value B from the register or the Immediate value we computed from the instruction to the ALU. It depends on the Opcode of the instruction, in the sense that only I types have an immediate value and for the branch I-type instructions we do not send the immediate value to the ALU. The select bit for the mux is the B or Immediate value we calculated in the decode stage. A circuit diagram is shown in Figure 9.

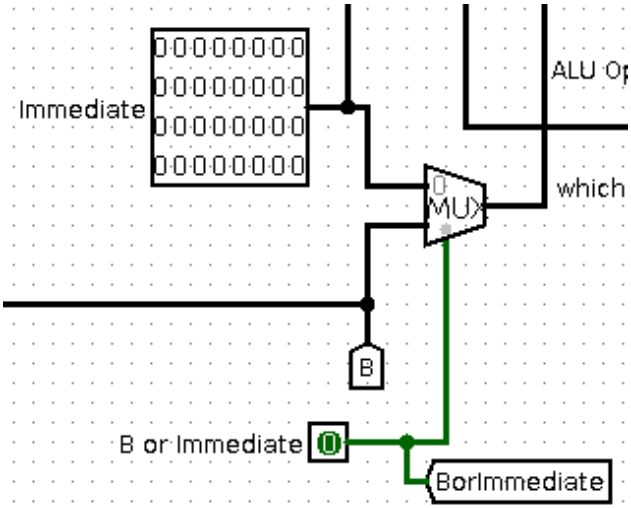

Figure 9: B or Immediate Selection Circuit

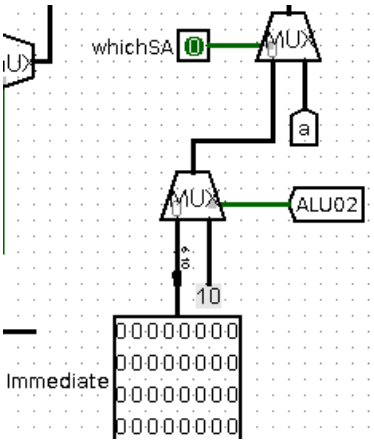

Figure 10: Shift Amount Selection Circuit

7.2.3 Shift Ammount Muxes

This mux chooses what value we use for the shift amount. For SLL, SRL, and SRA use the shamt that is encoded in the R-type instructions, while SLLV, SRLV, and SRAV use the 5 least significant bits of the value in the register at address Rs and for some Immediate instructions, we shift the bit extended value by 16. We first choose between the 6-10 bits of the immediate value and 16 based on $\bar{Op}2Op0$ of the ALU op code (the ALU02 tunnel in the circuit diagram). Then, we choose between the value generated by the previous mux, and the 0-4 bits of A, based on WhichSa. We determined this value in the previous stage. The circuit diagram is shown in Figure 10

7.2.4 ALU Control Logic

This value is determined soley by the ALU Op code we decoded from the previous stage.

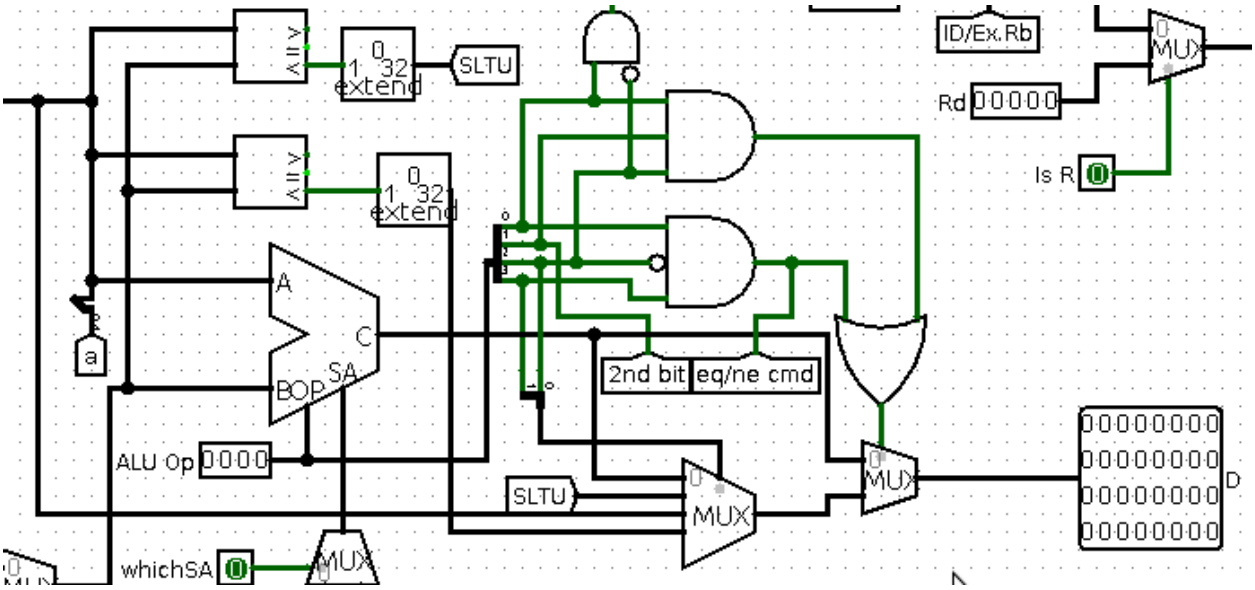

Figure 11: Comparator Logic Circuit

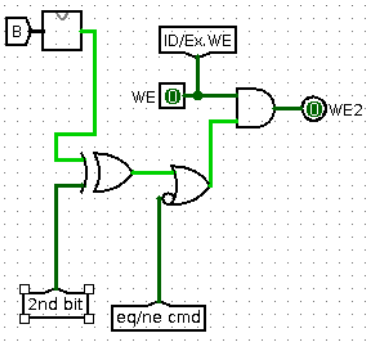

Figure 12: WE2 Control Logic Circuit

7.2.5 Comparators

For the SLTxx commands, we need two comparators – one for unsigned comparison and one for two's complement. We also need logic to choose between these two, the ALU output, and A (for load commands). We use some logic on the ALU Op code to choose between these values. The circuit diagram is shown in Figure 11.

7.2.6 WE2 Control Logic

We partially determined whether nor not we should permit writing to the register in the previous stage. We now account for the cases that result from MOVN and MOVZ, as we only want to write to the register for those commands if B!=0 and B==0, respectively. As a result, we look at whether it is a equal/not equal command, and whether B equals zero. The circuit diagram is shown in Figure 12.

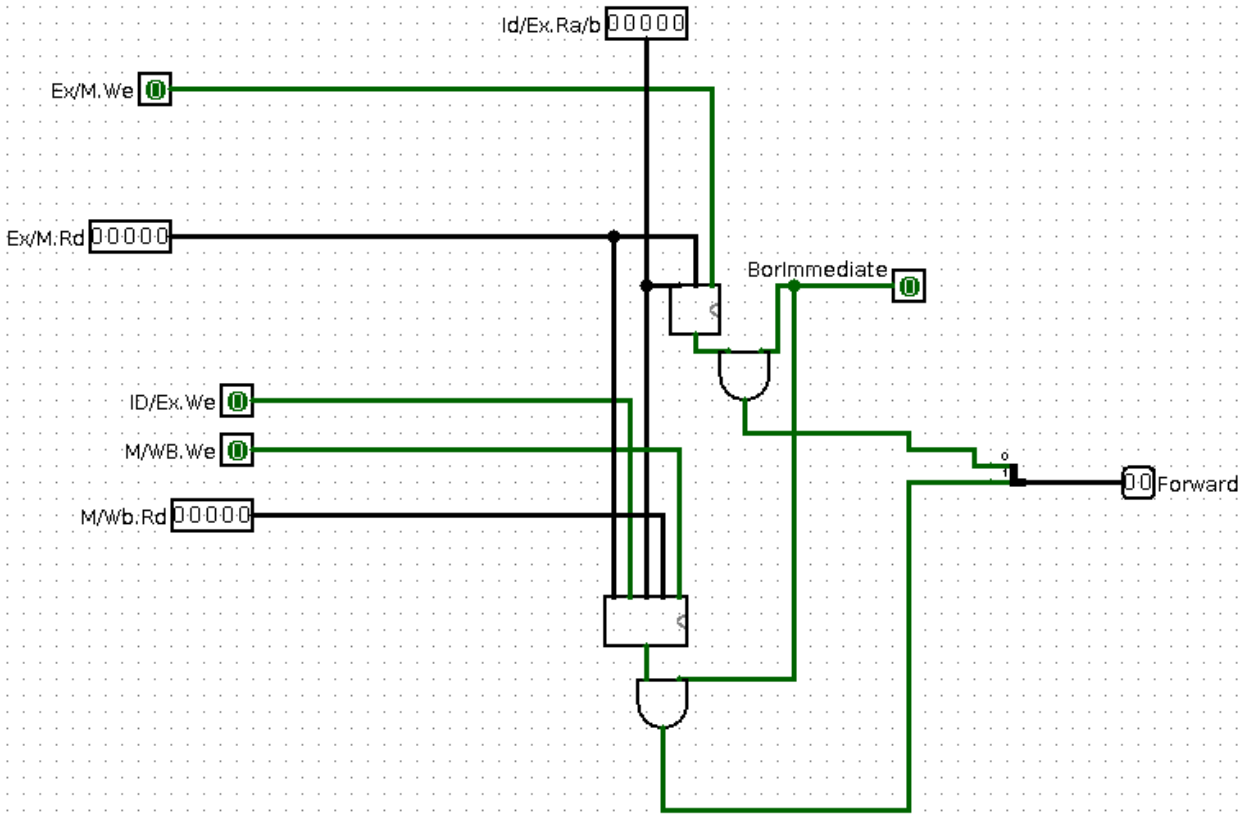

Figure 13: Forwarding Top Level Circuit

7.3 Submodule: Forwarding Unit

In this submodule, we determine if any sort of forwarding is needed to bypass a data hazard. We check if an Ex/Mem bypass or an Mem/WB bypass is necessary using the following subcircuits and also check to ensure that the immediate value is not needed instead. A circuit diagram is shown in Figure 13.

7.3.1 Ex/Mem Bypass

It checks if the value from the previous instruction currently entering the memory stage needs to be forwarded to the execute stage of the next instruction. The logic is as follows:
forward = (Ex/M.WE && EX/M.Rd != 0 && ID/Ex.Ra/b == Ex/M.Rd)
The circuit diagram is shown in Figure 14.

7.3.2 Mem/Wb Bypass

It checks if the value from two previous instructions ago currently entering the writeback stage needs to be forwarded to the execute stage of the next instruction. The logic is as follows:
forward = (M/WB.WE && M/WB.Rd != 0 && ID/Ex.Ra/b == M/WB.Rd) && not Ex/Mem Bypass
The circuit diagram is shown in Figure 15.

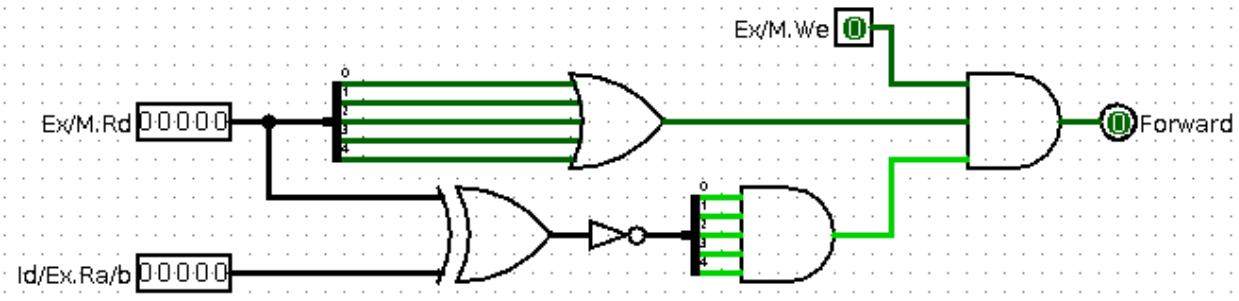

Figure 14: Ex/Mem Bypass Circuit

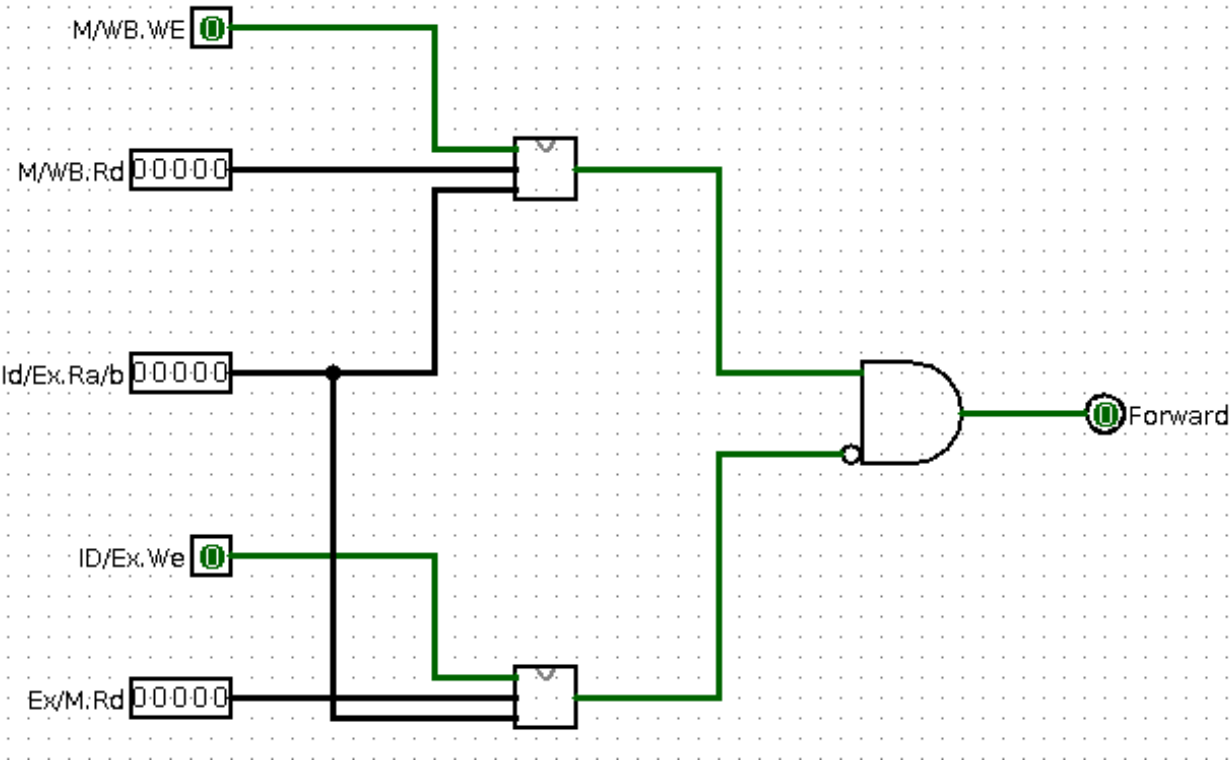

Figure 15: Mem/WB Bypass Circuit

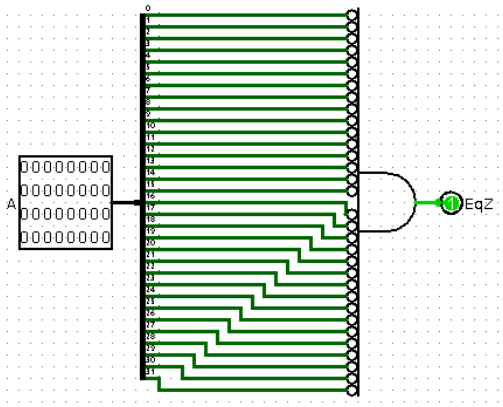

Figure 16: Equals Zero Circuit

7.4 Submodule: EqZero

This submodule determines if a 32 bit input is equal to 0. We have made it a submodule primarily for space constraints in the execute stage. The circuit diagram is shown in Figure 16.

7.5 Correctness Constraints

This subsection should be able to int erpret the control logic to select the correct operation in the ALU, as well as determing which values we want to input into the ALU. This all depends on the specifications of each instruciton.

8 The Memory Stage

We are not implementing this stage at this time, but in the future it will allow us to implement instructions that access memory. For now it is a place holder that simply passes the relevent information on. The circuit diagram is shown in Figure 17.

9 The Writeback Stage

The write back stage sends the final computed values back to the register. The circuit diagram is shown in Figure 18.

9.0.1 M or D to D Mux

This multiplexer determines if we send the output value from memory stage or the output value from the execute back to the execute stage as the D value. The Mux selects the 0 value if we want to use the current D and 1 if we want to use Memory output. This is based on the value of Mem, which we have determined earlier (it is 1 when the instruction accesses memory). The mux is then simply the value of Mem. WE, whether or not write is enabled on the register, and Dest, the address of the register to potentially write a value to have also been calculated previously and are just passed through here.

9.0.2 Correctness Constraints

The Writeback stage should write back the correct value to the correct register if it is an instruction that modifies a value in a register. It should return a value from memory if it is a memory instruction and the value we computed in the execute stage otherwise.

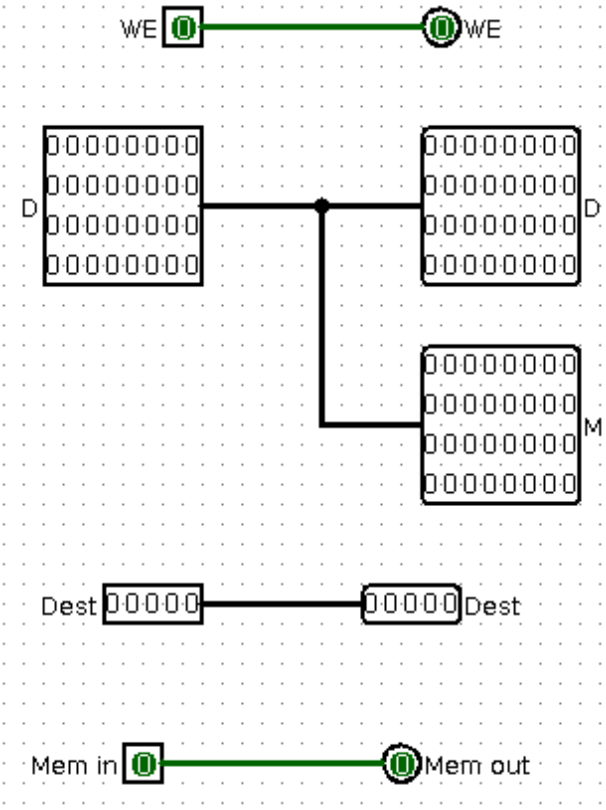

Figure 17: Memory Stage

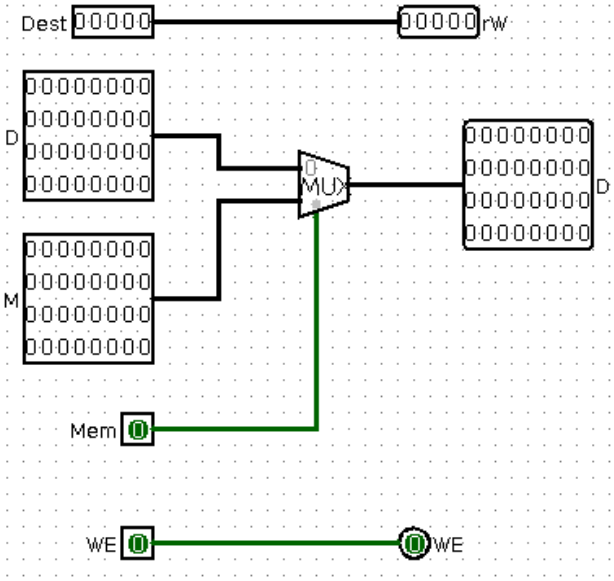

Figure 18: Writeback Stage