Project 1 Design Doc

Yiming Li, yl564 Pulkit Kashyap, pk374

October 18, 2015

1 Overview

This document outlines the major stages of a basic 32-bit, pipelined Microprocessor without Interlocked Pipeline Stages (MIPS) with data hazard detection and forwarding, and the components of these stages. The processor is comprised of 5 stages: fetch (IF), decode (ID), execute (EX), memory (MEM), and writeback (WB). We are implementing pipelining by having the output of each stage saved to a series of registers referred to as the pipe (each pipe is named by the stages it receives input from and outputs to.) The subsequent sections will run through the top level design of the processor, each major stage, all subcircuits, testing methodology and decoding logic. For reference, we used the lecture notes for the Fall 2015 semester of CS 3410.

Since the processor we are building for this project does not contain data memory, execute memory instructions and Jump-Type Instructions, handle control hazards and delay slots, and include coprocessor instructions and traps, this document will not discuss these features. The design doc is written for instructors and students who have taken Systems Programming or any equivalent computer engineering course.

2 The Top Level Pipeline (MIPS32)

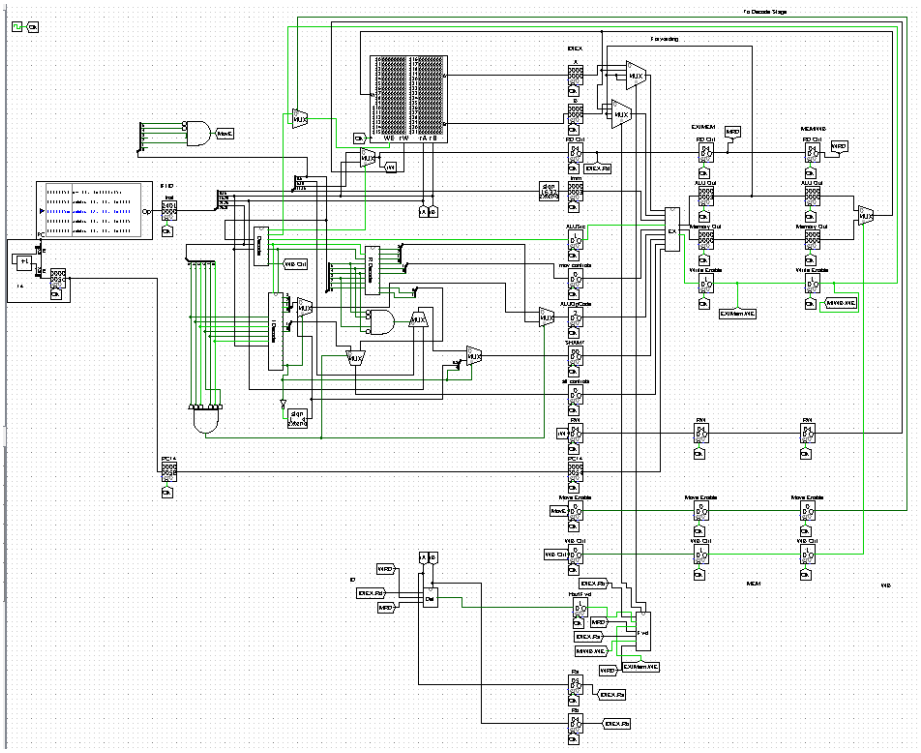


Figure 1: Top View Diagram

Fig. 1 goes into detail what the processor should look like from a top level perspective. To elaborate, this processor will have 5 stages: Fetch, Decode, Execute, Memory, and Writeback. Following with our in-class discussion, the Memory, Writeback, and Execute stage will all take a part in data forwarding in order to make sure data hazards do not cause not optimal situations for our processor.

2.1 Timing

All registers in the pipeline, the register file in ID, and the program counter (PC) operate on a single clock (Clk). The register file updates on the falling edge whereas all other registers update on the rising edge to account for the register file bypass data hazard.

3 Fetch Stage (IF)

Incoming Latches: N/A
Outgoing Latches: Instruction, PC+4

Figure 2 shows the fetch stage subcircuit. The Inst register holds the Op code for the decode stage and the PC+4 holds address for the next instruction.

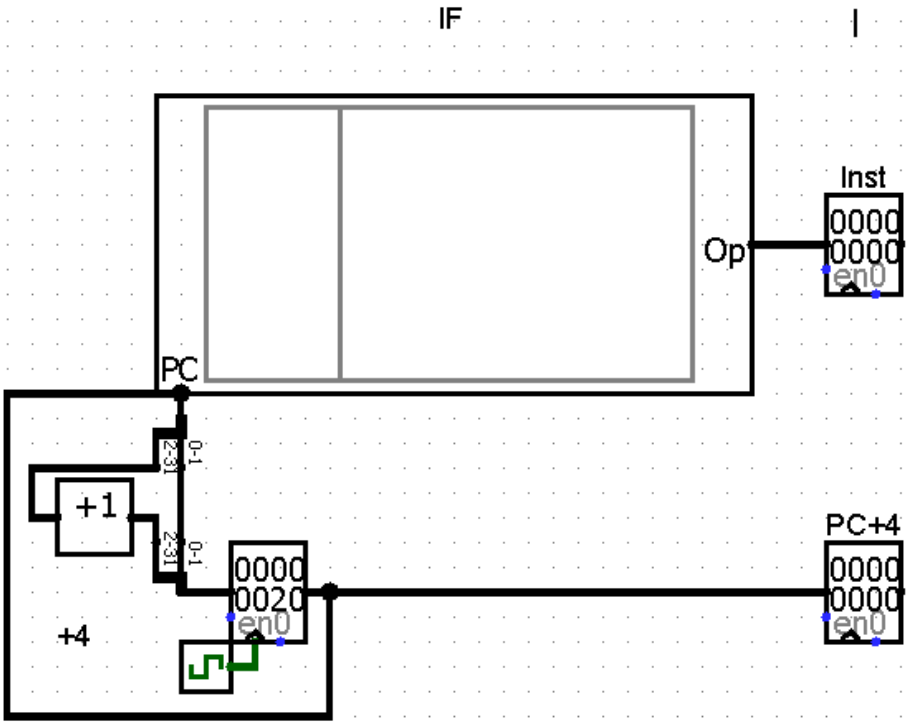


Figure 2: Fetch portion + IF/ID latch.

3.1 PC+4

With every rising edge of Clk, this module increments the program counter by 4 bits, which is where the next instruction is located (each op code is 4 bits wide)

4 Decode Stage

Incoming Latches: Inst
Outgoing Latches: A, B, RD Ctrl, Imm, ALUSrc, mov controls, ALUOpCode, SHAMT, slt controls

See Fig. 3. Data from the Inst latch is fed into a decode function that sends its output to the register file. Instructions are also decoded and sent to the

appropriate latches. If the instruction requires an immediate, then the immediate is sent to a bit extender before being passed to the ID/EX latch. Finally, inputs for the ID/EX latch are sent from the register file. The decode stage is also where our hazard detection will be happening. If a hazard is detected this information will need to be passed to later stages so that data forwarding can be used to retrieve the necessary values in the case of a data hazard.

The register file regurgitates the contents of the 'READ' registers to latches A and B. The value from the writeback stage is fed into the register file and the WB selector decides whether or not to write this value to the 'WRITE' register.

We decided to not create a subcircuit for the decode stage as that is known to cause errors with the register file.

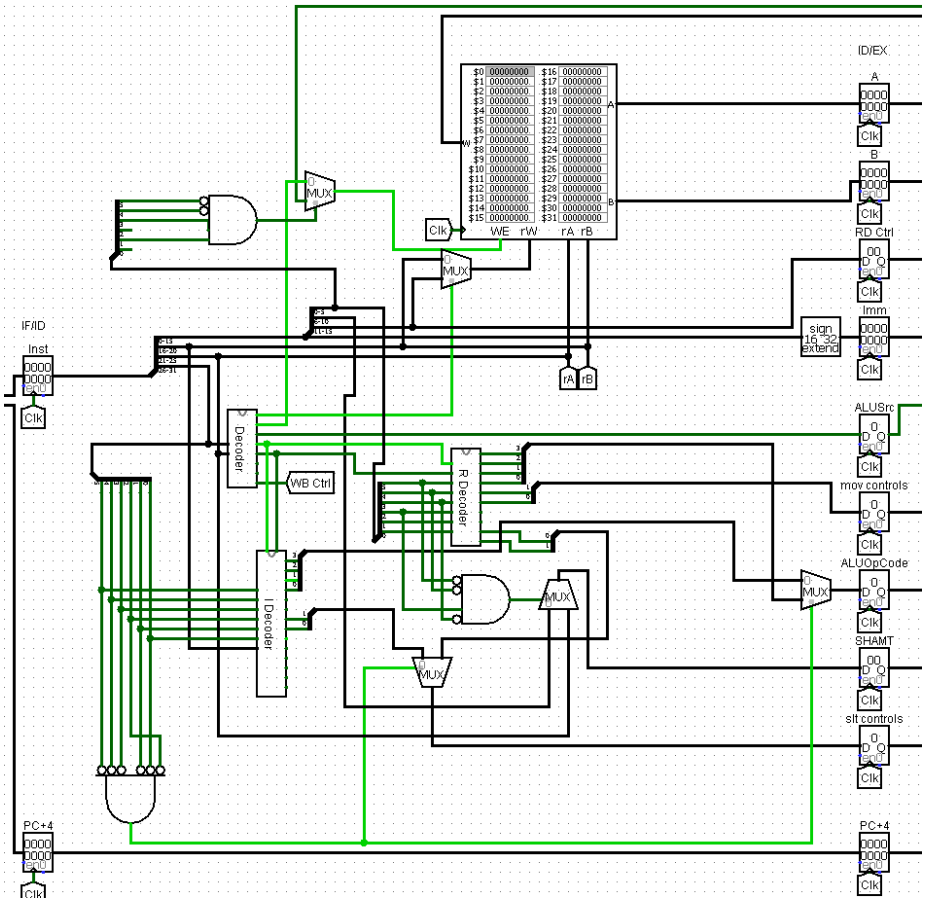


Figure 3: Decode Stage in MIPS32 main circuit with IF/ID and ID/EX pipelines.

4.1 Control Decoder (Control Decoder)

Input: Op, rt
Output: RegDst, RegWr, ALUSrc, ALUOp1, ALUOp0, MemRead, MemWrite, MemToReg

This module determines what type of instruction data is coming in from the Inst latch. This module also decodes miscellaneous instructions such as determining if the ALU is using the and immediate, where and whether to write to the register file, and where and whether to write to memory.

If the instruction is from table B (such as jumping and branching), then the the write enable signal is disabled and no changes will be made to the register or forwarding unit.

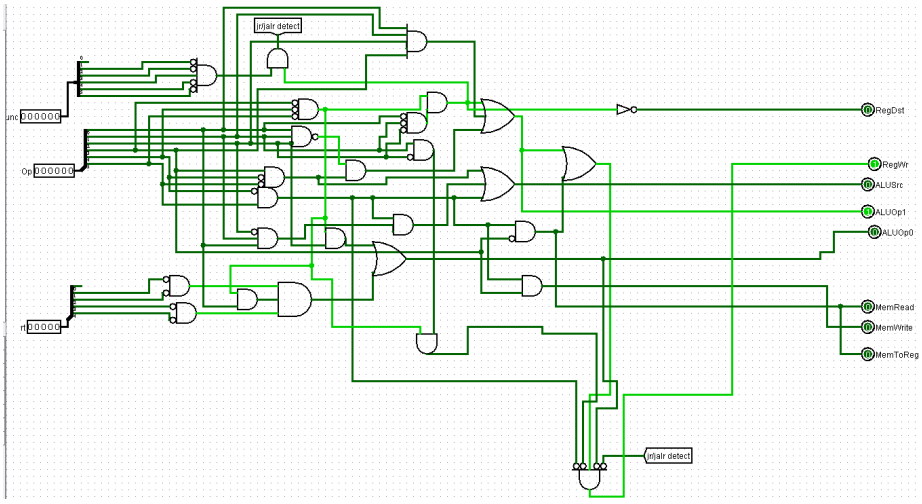


Figure 4: The Control Decoder subcircuit

4.2 R-Type Instruction Decoder (ALUOpCoderRTypeInstruction)

Input: Bits 0-5 of the Op code
Output: Flags for R-Type Instructions and J-Type Instructions located in bits 0-5 of Op code.

This circuit contains the decoding logic if the Op code is either an R-Type Instruction or a J-Type Instruction that is decoded in bits 0-5 of the Op code. This will map every function to its respective ALU Function from the given table. The only special value(s) occur with any variation of the SLT and Movz which will decode to their respective control signals and also the add-ALU OPcode (001x)

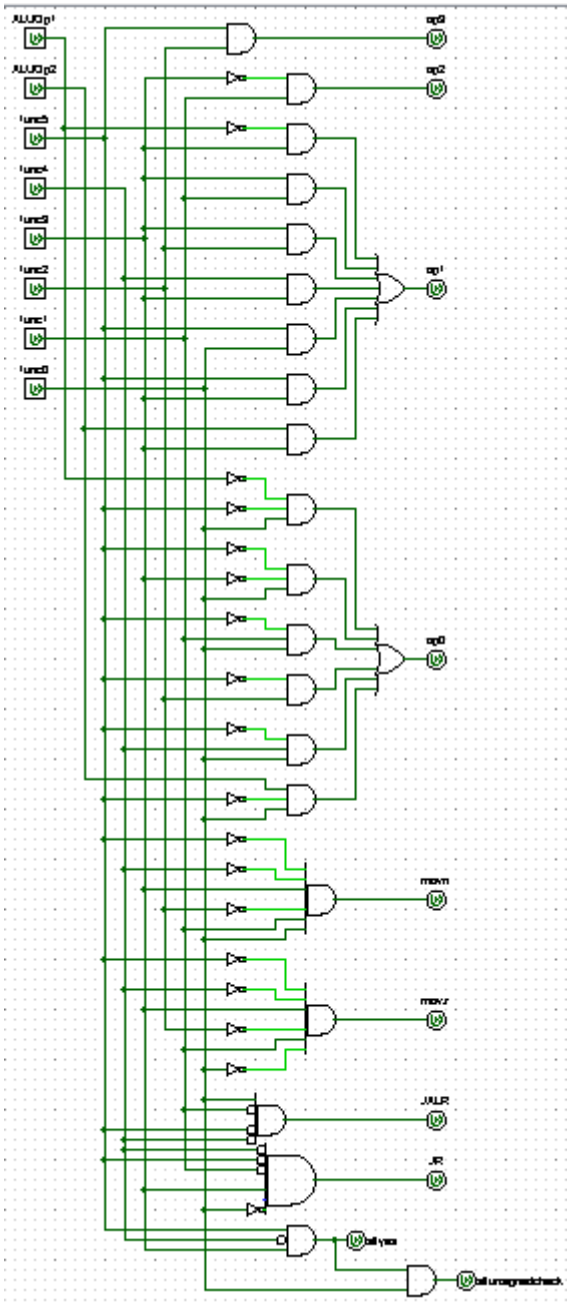


Figure 5: Decoding logic for R-Type Instructions

4.3 I-Type Instruction Decoder(ALUImmediateOPCircuit)

Input: Bits 26-31 of the Op code
Output: Flags for I-Type Instructions and J-Type Instructions located in bits 26-31 of Op code.

This circuit contains the decoding logic if the Op code is either an I-Type Instruction or a J-Type Instruction that is decoded in bits 26-31 of the Op code.

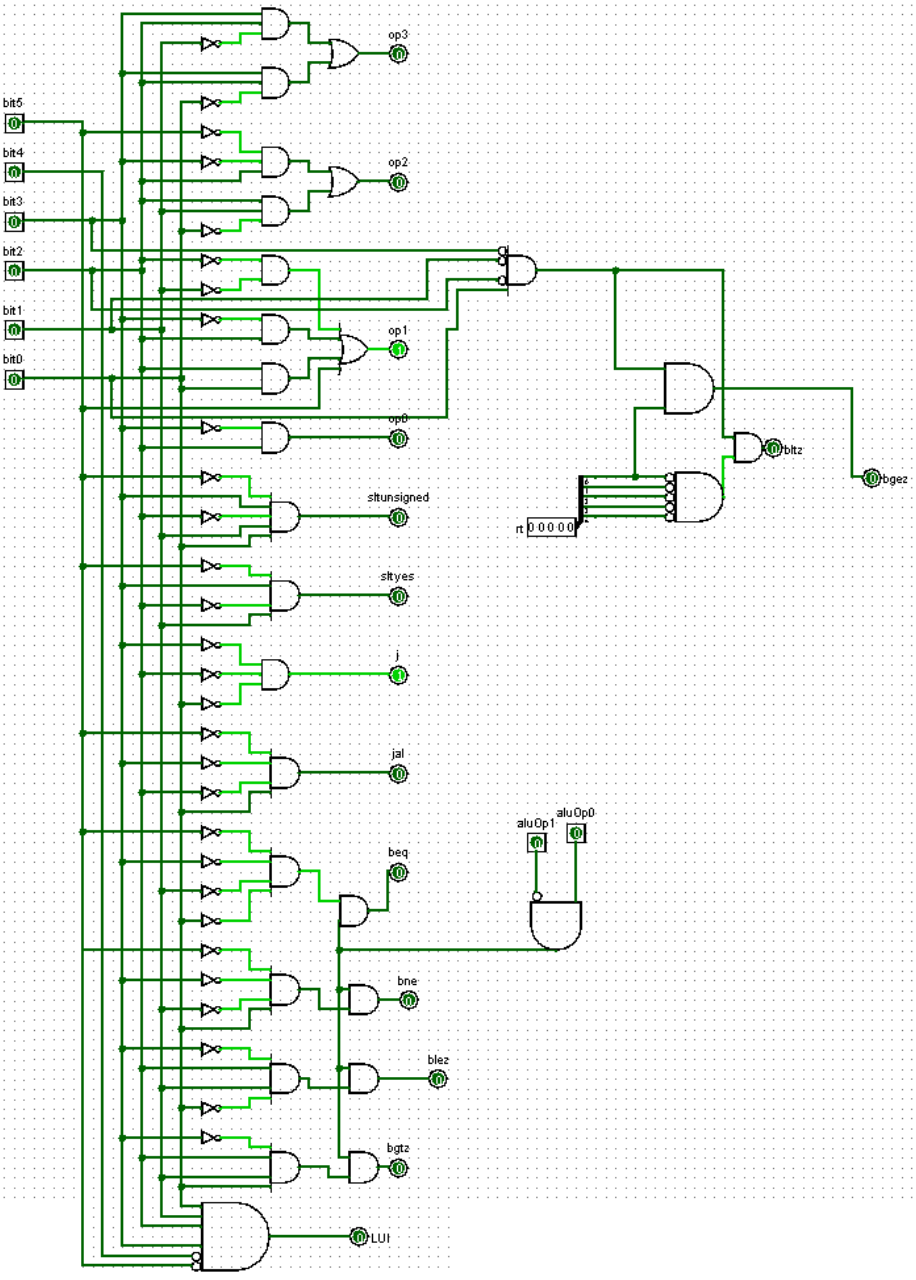


Figure 6: Decoding logic for I-Type Instructions

5 Execute Stage

Incoming Latches: A, B, Imm, ALUSrc, mov controls, ALUOpCode, SHAMT, slt controls, PC+4
Outgoing Latches: ALU Out, Memory Out, Write Enable

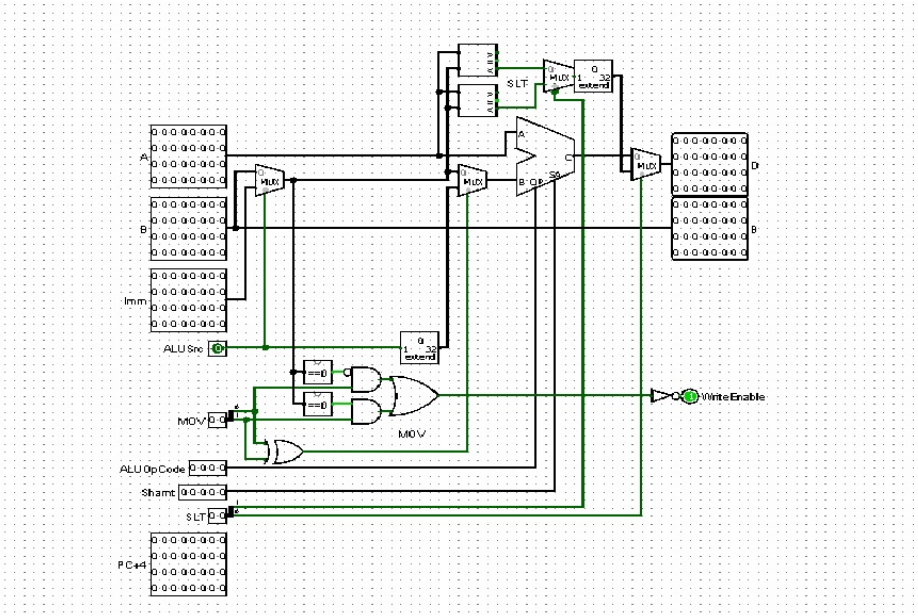


Figure 7: Execute Stage subcircuit

5.1 Execute Stage Description

On every cycle this execute stage reads the ID/EX Latch Registers to get values and control bits in order to preform ALU operations. It will also pass values of interest to the EX/MEM latch. Values of interest can include values that need to be stored in memory or control values among others. This stage will also support data forwarding from EX/MEM to EX.The first picture shows what is going on within the actual execute circuit. For example, this subcircuit takes in the MOV, SHAMT, SLT, PC+4, IMM, B, and A values. If slt for example is 01 then this indicates we are doing an SLT, if it is 11 then that means we are doing an slt-unsigned operation. If Mov is 01 then that indicates we are doing movz, if it is mov10 then that means we are doing MovN. The rest of the values are standard in function with the shamt being either the original shamt from our instruction or another if we are doing a logical value instruction. Finally,

the write enable signal we see at the end is going to be sent down the circuit until it comes back to the ID-Stage where it will be used to determine if the movz/movn command should actually be doing anything.

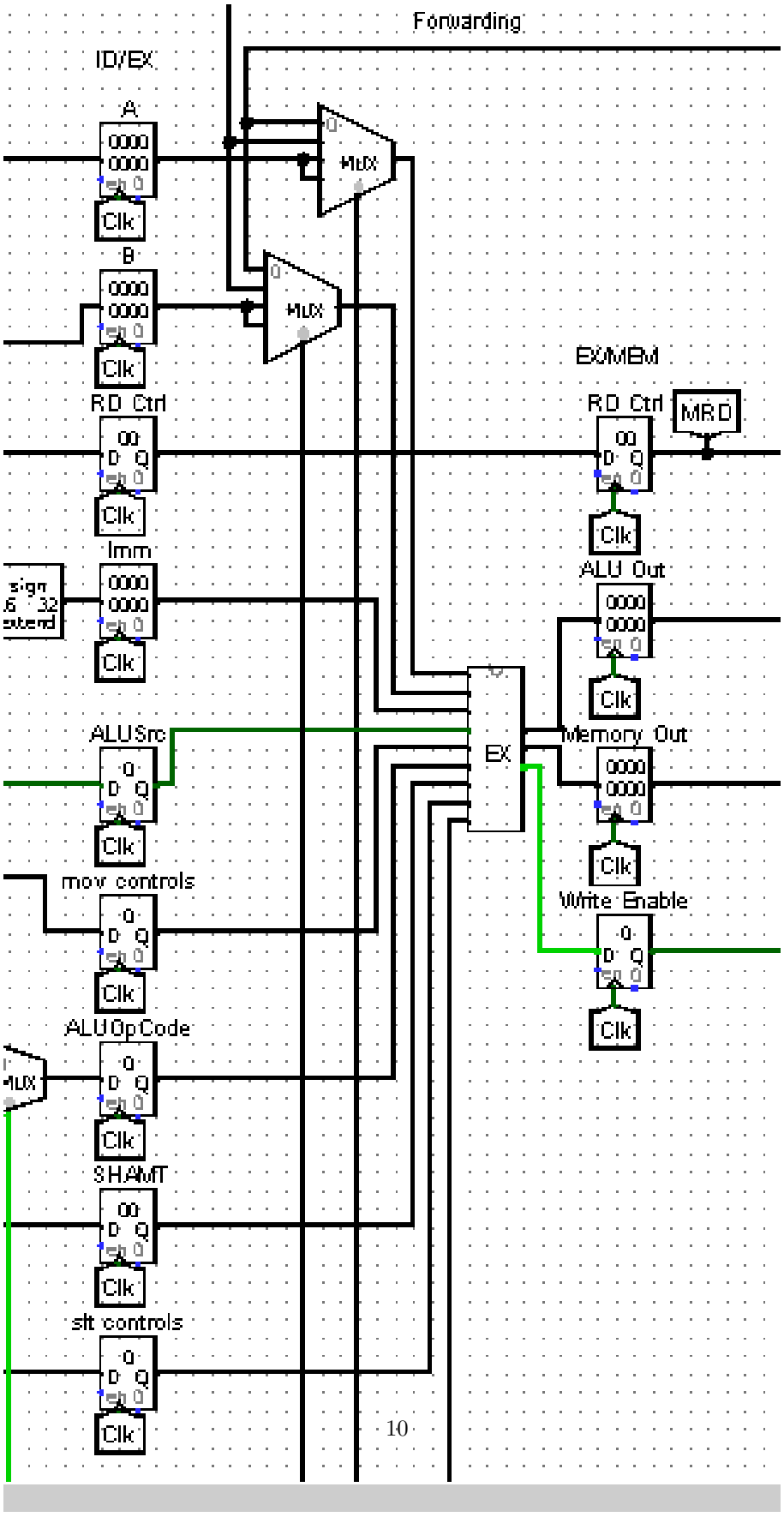


Figure 8: Execute Latch Stage (taken from main circuit)

5.2 Execute Latches

To clarify, this picture is showing the execute stage along with all the latches coming in and out for the stage (namely ID/EX EX/MEM). You can see that the forwarding logic is also implemented here.

6 Memory

For this project, the memory section will simply pass the information from the EX/MEM pipeline to the MEM/WB pipeline. However, this stage will still support data forwarding from the adjacent pipelines to the execute stage.

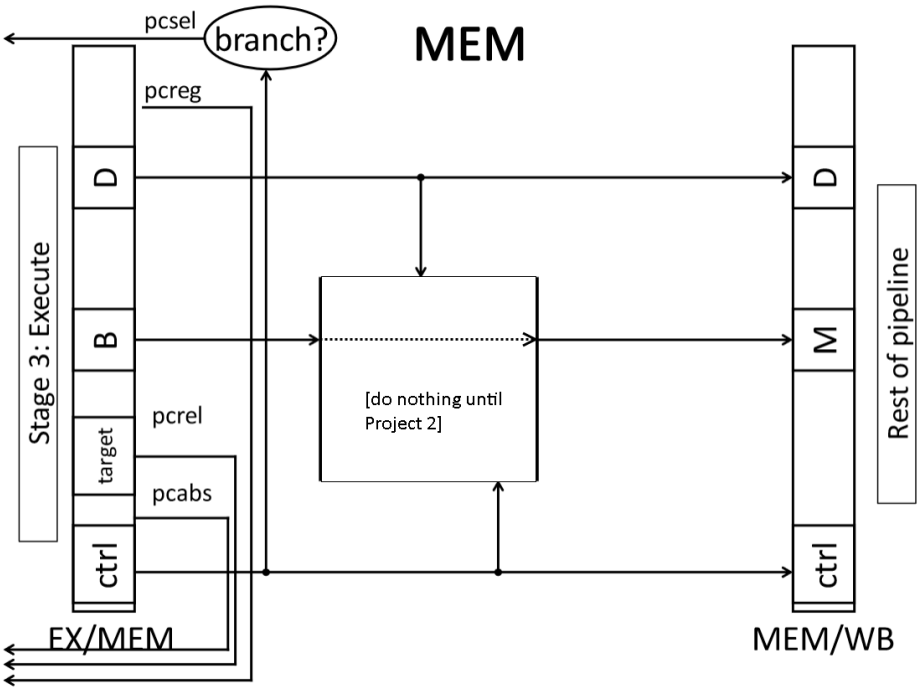


Figure 9: The abstract memory cycle with the EX/MEM pipeline(left) and the MEM/WB pipeline(right). This was taken from lecture.

7 Write-Back

Incoming Latches: ALU Out, Memory Out, Write Enable, RW, WB Ctrl, Move Enable
Outgoing Latches: N/A

This stage decides whether to store the ALU or the MEM output into the register file in the Decode stage. Since this stage utilizes only one MUX and does not utilize any logic gates, this stage has no modules nor is it placed in its

own sub-circuit. This stage also decides where and whether or not to write back to the register file. The WriteEnable Control tells us if we need to be writing to memory or not

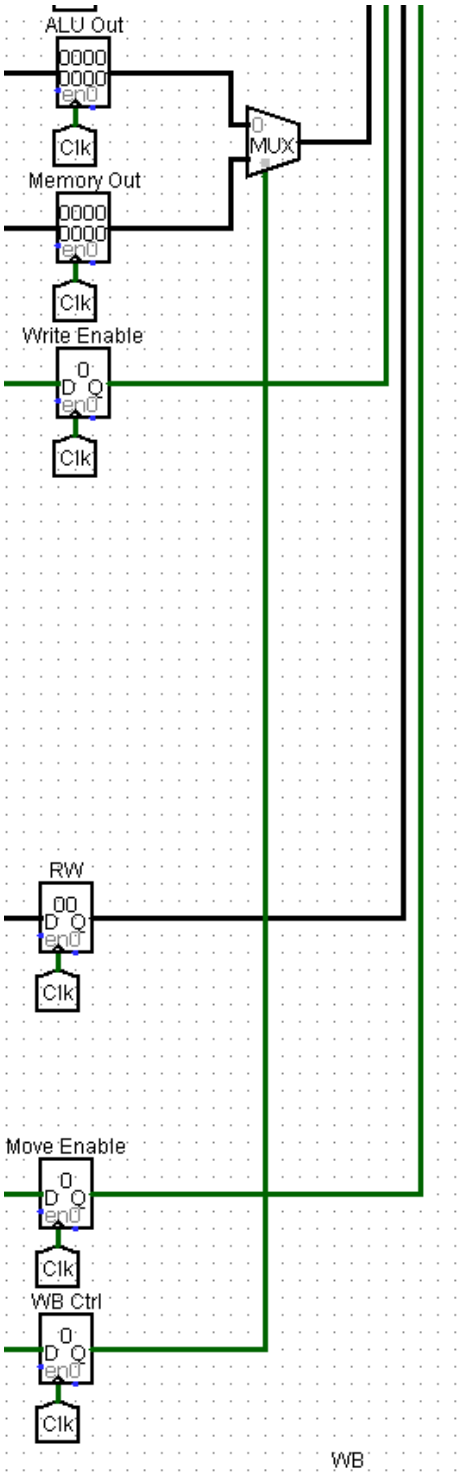


Figure 10: The writeback cycle with the MEM/WB pipe.

8 Data Hazards

Data Hazard detection and forwarding account for three types of data hazards, EX—MEM-¿EX, MEM—WB-¿EX, and Register File Bypass. These units also account for false data hazards ($0 is writing while being read) and simultaneous data hazards (in which case the most up to date value of the register will be read).

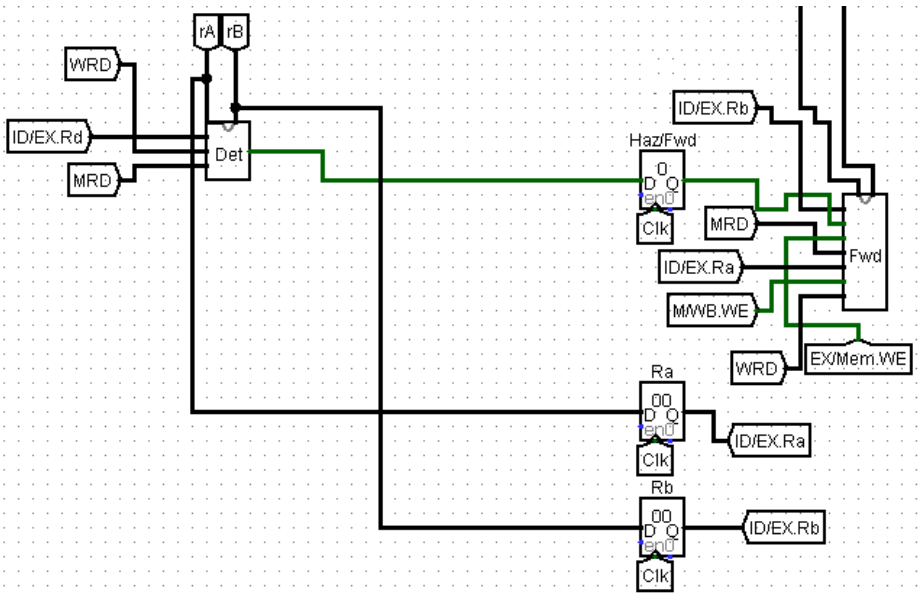


Figure 11: Closeup of the data hazard and fowarding logic unit from the main circuit.

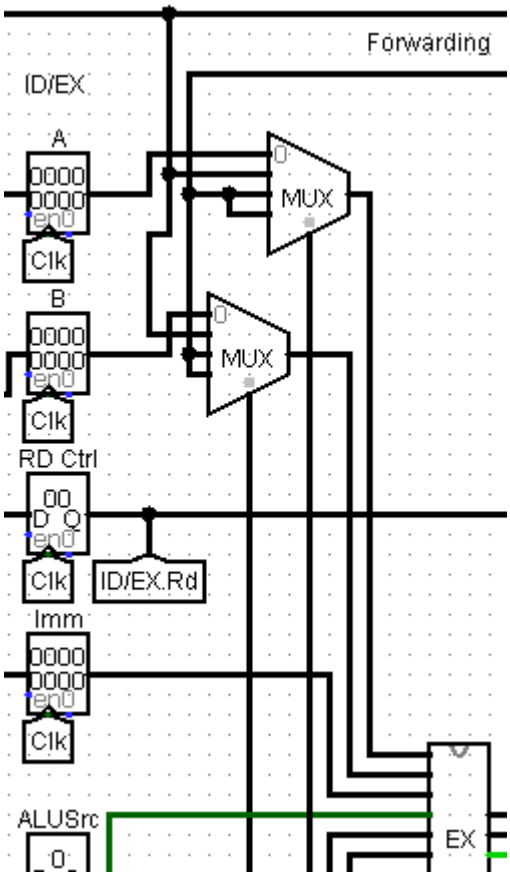


Figure 12: Closeup of the forwarding unit from the main circuit.

8.1 Hazard Detection (Detect Hazard)

Incoming Latches: N/A
Outgoing Latches: Haz/Fwd

This subcircuit outputs a one bit flag indicating whether or not there is a data hazard and feeds it to the forwarding logic in the next stage via latch. Our implementation of hazard detection lies in the ID stage The type of hazard is not determined here. Figure 13 shows the combinational logic used to detect data hazards. Note that the 5-input OR gate checks for false hazards if the two read addresses are 0. Since the $0 register always holds 0, there should never be a data hazard at that register.

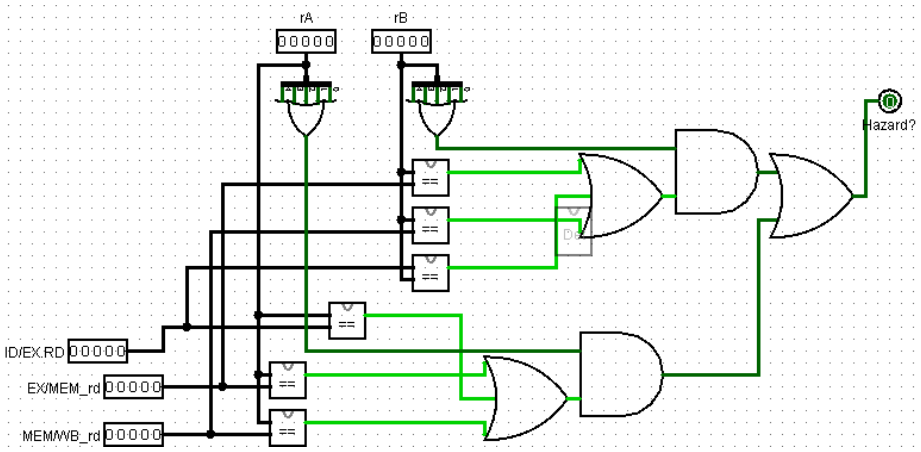


Figure 13: Detect Hazard subcircuit

8.2 Forwarding Logic (Forwarding)

Incoming Latches: Haz/Fwd, rA, rB
Outgoing Latches: N/A

This component determines what sort of data hazard (i.e. MEM-¿EX, WB-¿EX, Register Bypass) is present (if any) and outputs two 2-bit selectors. Figure 14 shows the combinational logic used to determine the type of data hazard. For each 2-bit signal, the 1st bit determines whether or not there is a MEM-¿EX hazard, the 0th bit determines whether or not there is a WB-¿EX hazard. The B output selects for data hazards in the B register, while the A output selects for data hazards in the A register. This subcircuit is located in the EX stage.

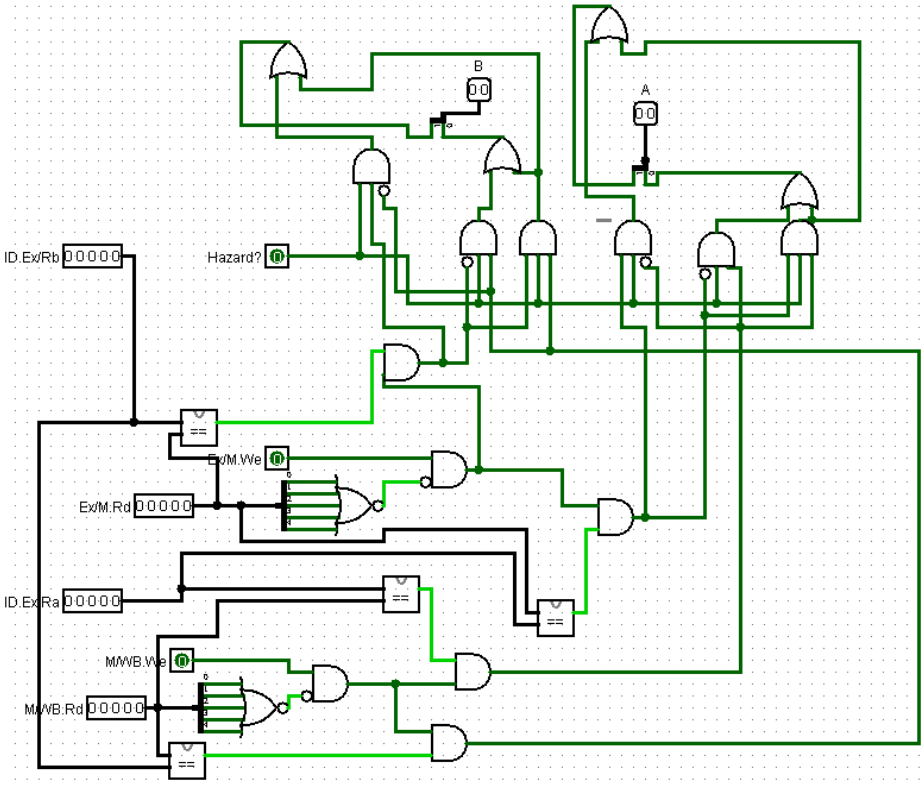


Figure 14: Forwarding subcircuit

8.3 Forwarding Unit

Refer to Figure 12. The forwarding unit provides the most recent changes to a register with data dependence along with the values at rB and rA from the ID stage. It selects for what to shove into the execute stage depending on the type of data hazard–or lack of–was found by the forwarding logic. In the case that both MEM-¿EX and WB-¿EX hazards are present, this unit will feed the execute stage the most recent (i.e. MEM-¿EX errors have priority over WB-¿EX errors).

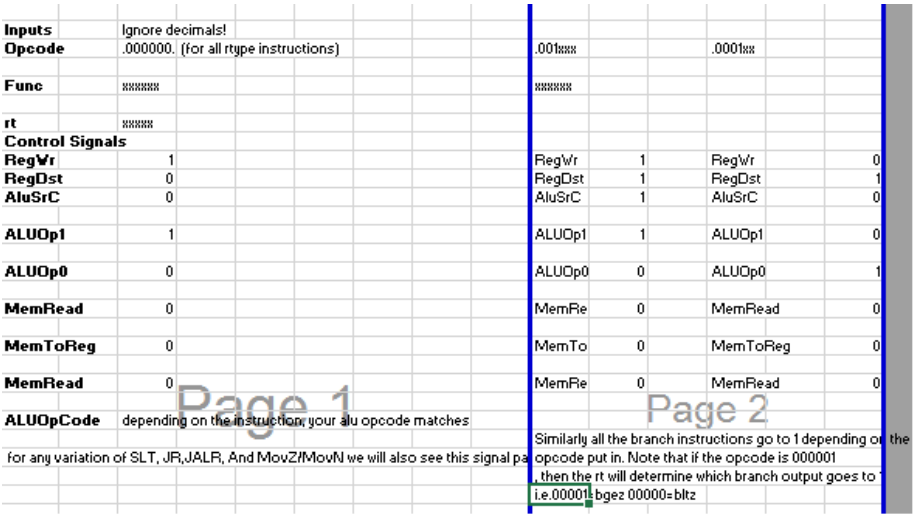
9 Testing

9.1 Testing Methodology

Our testing methodology utilizes test vectors to complete the bulk of the work. These test vectors test all instructions listed in Table A of the project description. We consider data hazards and false hazards as corner cases and account for them by testing sets of instructions that would incur single or multiple data

hazards, then by testing data hazards that involve the zero register (i.e. false hazard).

10 Decoding Logic

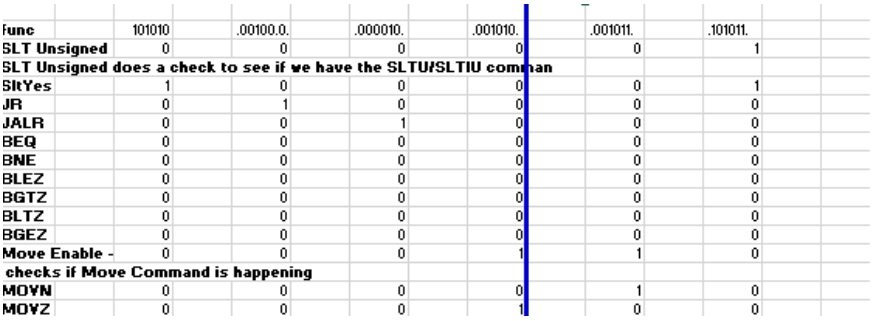


Inputs	Ignore decimals!							
Opcode	.000000.	(for all rtype instructions)		.001xxx			.0001xx	
Func	xxxxxx			xxxxxx				
rt	xxxxx							
Control Signals								
RegWr	1			RegWr	1		RegWr	0
RegDst	0			RegDst	1		RegDst	1
AluSrC	0			AluSrC	1		AluSrC	0
ALUOp1	1			ALUOp1	1		ALUOp1	0
ALUOp0	0			ALUOp0	0		ALUOp0	1
MemRead	0			MemRe	0		MemRead	0
MemToReg	0			MemTo	0		MemToReg	0
MemRead	0			MemRe	0		MemRead	0
ALUOpCode	depending on the instruction, your alu opcode matches							
for any variation of SLT, JR,JALR, And MovZ/MovN we will also see this signal pa				Similarly all the branch instructions go to 1 depending o the opcode put in. Note that if the opcode is 000001 , then the rt will determine which branch output goes to i.e.00001 bgez 00000=bltz				

Page 1

Page 2

Figure 15: Decoding Logic OpCodes



Func	101010	.00100.0.	.000010.	.001010.	.001011.	.101011.
SLT Unsigned	0	0	0	0	0	1
SLT Unsigned does a check to see if we have the SLTU/SLTIU comman						
SltYes	1	0	0	0	0	1
JR	0	1	0	0	0	0
JALR	0	0	1	0	0	0
BEQ	0	0	0	0	0	0
BNE	0	0	0	0	0	0
BLEZ	0	0	0	0	0	0
BGTZ	0	0	0	0	0	0
BLTZ	0	0	0	0	0	0
BGEZ	0	0	0	0	0	0
Move Enable -	0	0	0	1	1	0
checks if Move Command is happening						
MOVN	0	0	0	0	1	0
MOVZ	0	0	0	1	0	0

Figure 16: Decoding Logic Func

10.1 Important Signals

We have 7 important control signals, namely: RegDst, ALUSrc, MemtoReg, Reg-Write, Mem-Read, Mem-Write, Branch, ALUOp1, ALUOp0. We will be using the opcode to determine what the value of these control signals will be. For example, if we are dealing with a R-Format instruction, we would for example set ALUOp to 10 (aka ALUOp1 = 1, ALUOp0 =0) to indicate we will have to decode the func portion of our instruction for what our ALU should be doing. Moreover, in this example our Mem-Read, Mem-Write would both be set to

0 since an R-type instruction does not write or read from memory. ALUOp1, ALUOp0 are both important for the execution stage. Mem-Read, Mem-Write are similarly both important for the Memory Stage. Reg-Write is important for the Writeback stage as it writes to a register, and RegDst/ALUSrc are useful in the ID stage. RegDst is used to determine whether or not we use rt or rd for our destination value. This is determined based off whether we have an immediate or not. Similarly, ALUSrc is 1 if we have an immediate. The table attached shows what values these signals maps to depending on if we have a r-type instruction, i-type, or other.