Project 2 Design Document

Jason Zhou (jz629) and Olusola Adegunloye (oaa35)

Introduction

The purpose of this document is to outline our implementation of our MINI MIPS 32 bit microprocessor. The scope of this document covers the implementation of a CPU by providing an overview of all the main processor components that are required to run through a single instruction cycle. The intended audience is for the TA's to understand our logic when implementing the CPU and possibly for us for future reference. The textbook and lecture notes were used for reference for this design documentation.

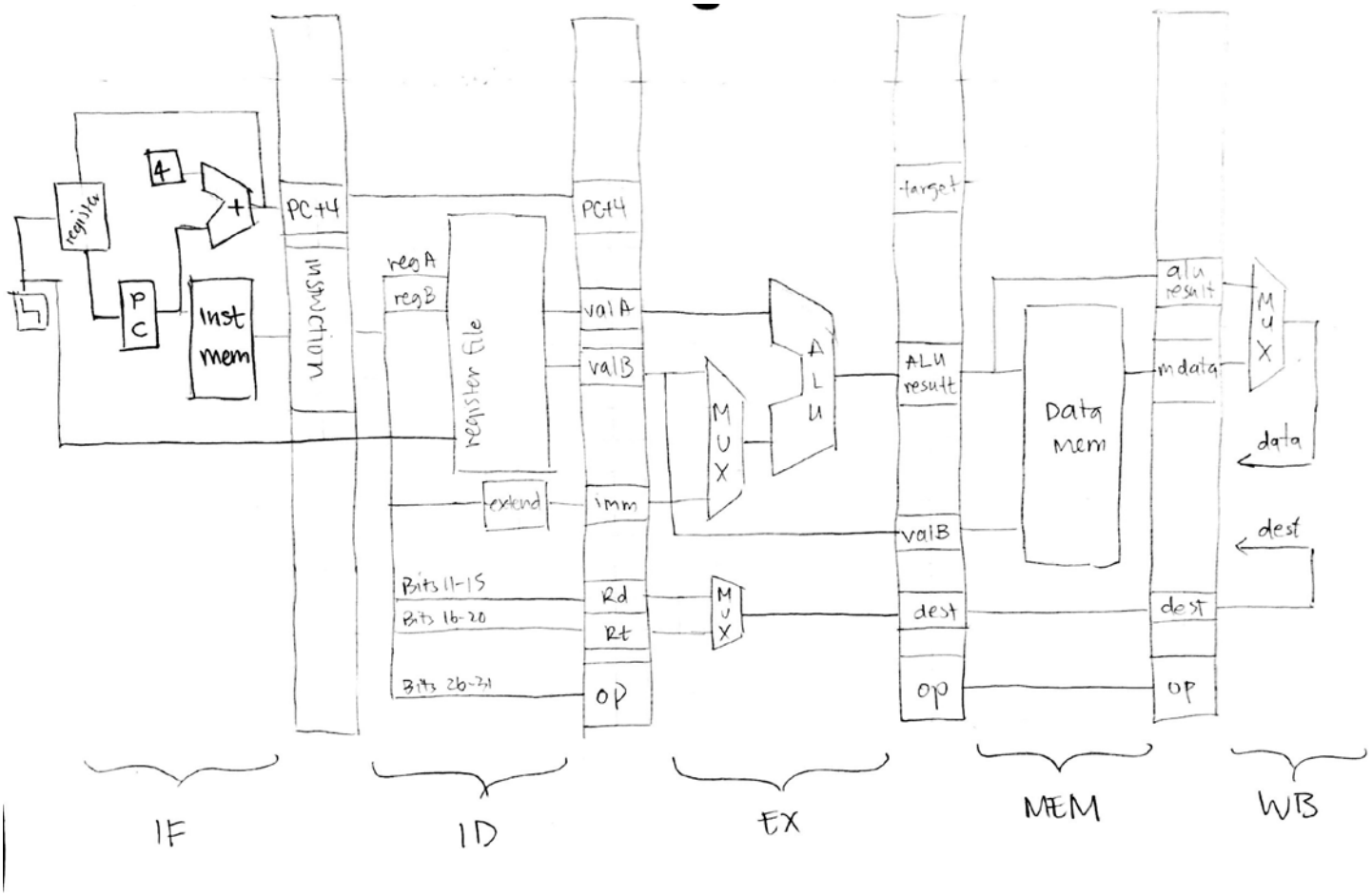
Important abbreviations:

- IF = Instruction Fetch
- ID = instruction decode
- EX = execution
- MEM = memory
- WB = writeback

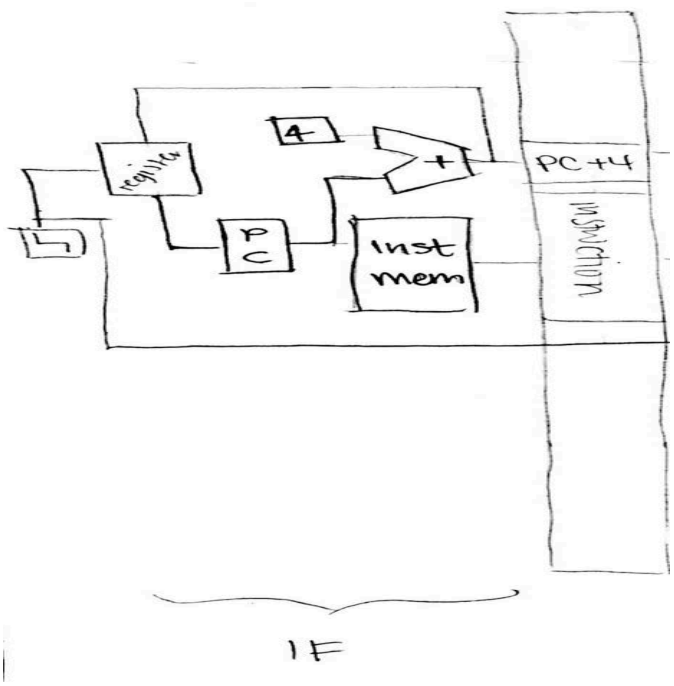
This document first goes over the overarching structure of the CPU then goes in depth into the IF, ID, and EX stages, providing descriptions as well as diagrams, constraints, and approaches for testing.

Overview

The diagram below shows our initial, top level design of our processor. There are 5 stages: IF, ID, EX, MEM, and WB, which each correspond to a stage of instruction execution. Instructions and data move generally from left to right through the five stages as they complete execution. There is no forwarding unit in place yet.


4
+
register
PC
Inst mem
PC+4
instruction
regA
regB
register file
extend
Bits 11-15
Bits 16-20
Bits 26-31
PC+4
valA
valB
imm
Rd
Rt
op
MUX
ALU
MUX
target
ALU result
valB
dest
op
Data mem
alu result
mdata
dest
op
MUX
data
dest
IF
ID
EX
MEM
WB

3 The Fetch Stage

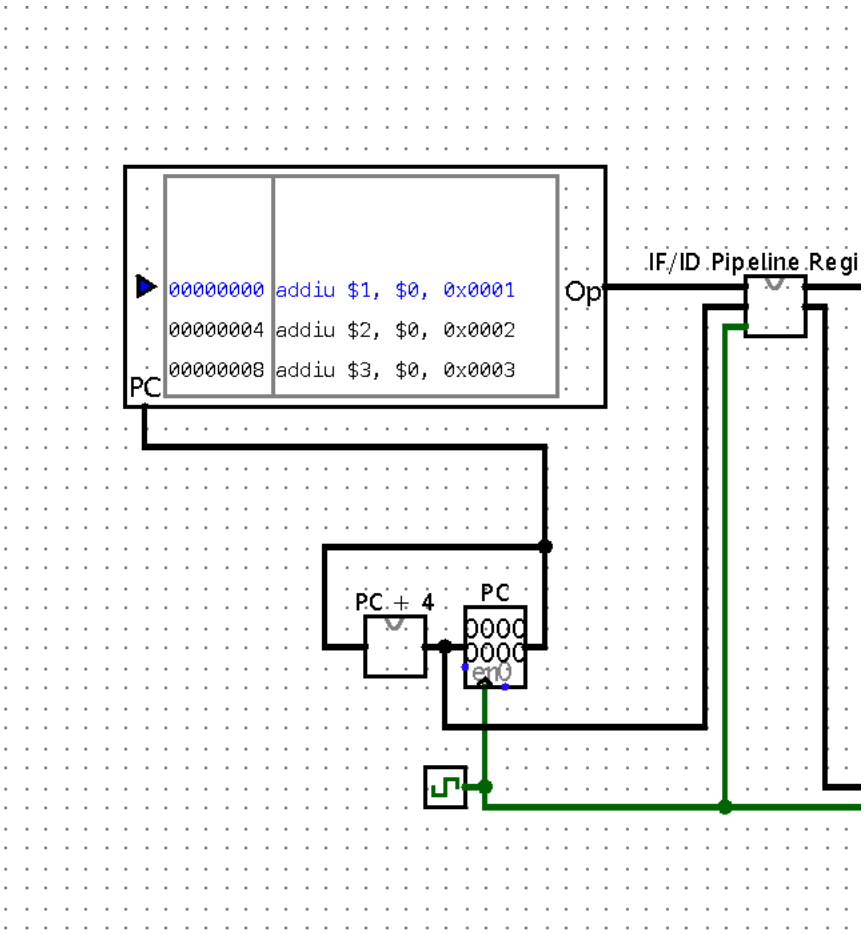


The diagram above has 5 components in the fetch stage:

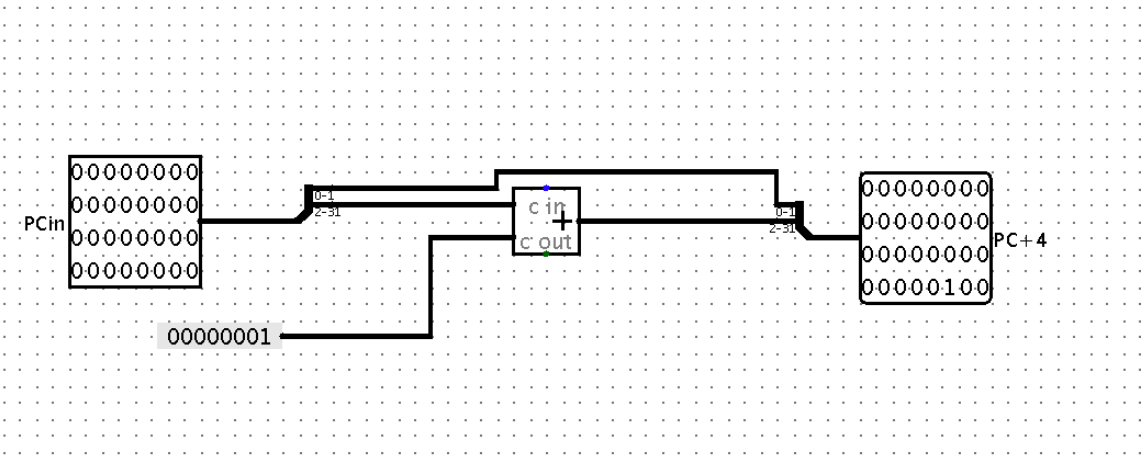
- PC (program counter)
- Register
- Clock
- 1-bit Adder
- Instruction memory

The instruction is read from memory using the address in the PC, then placed in the IF/ID pipeline register. The PC address is then incremented by 4, written back into the PC, and the incremented address is saved in the IF/ID pipeline register.

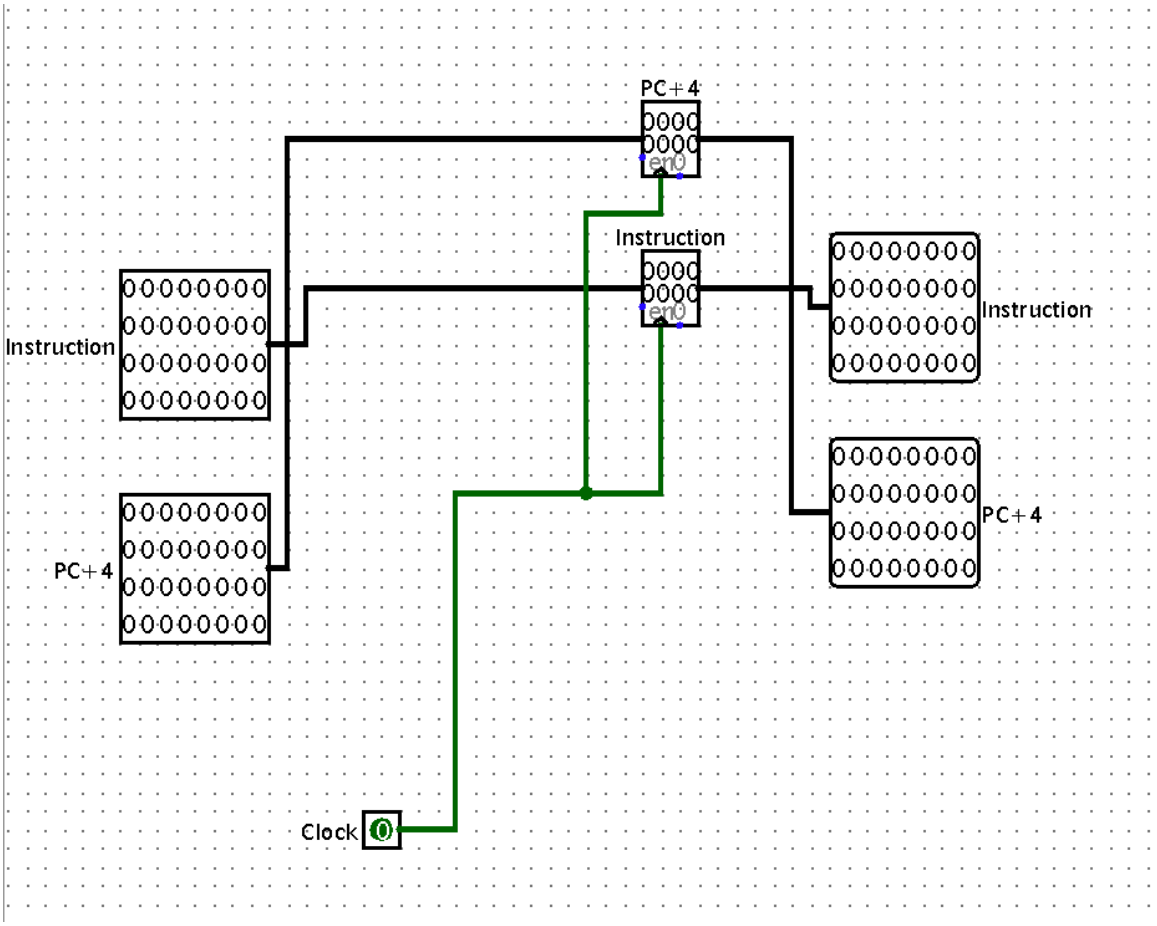
Here's what our implementation of the IF stage looks like:



32-bit instructions are fetched from the program memory unit and stored into the IF/IF pipeline register. The PC begins at 0 and increments by 4 every rising edge. The PC+4 value (not PC) is placed into the pipeline register for later use. Lets take a closer look inside our PC+4 unit implementation.

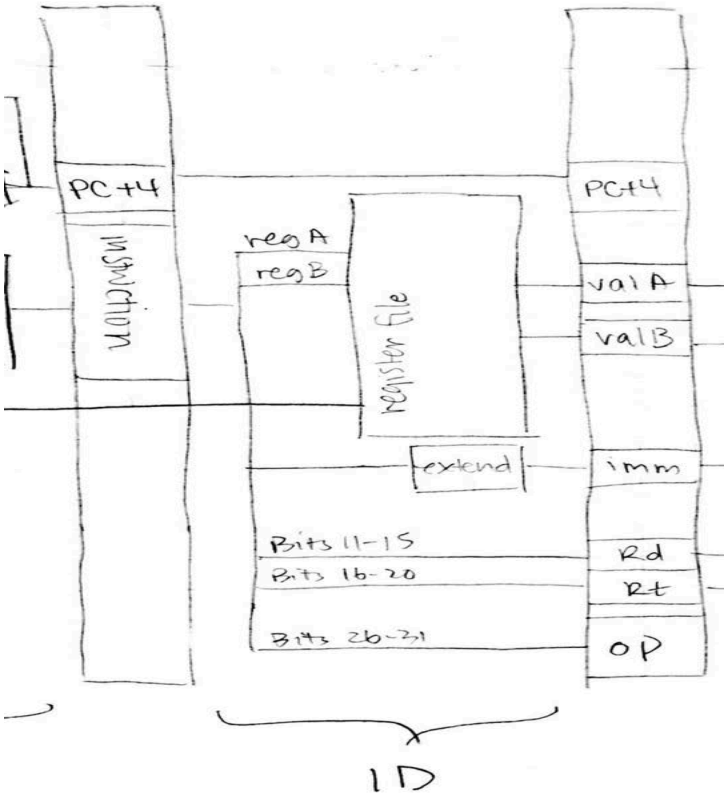


Our PC+4 was given the constraint that we could only use a 1 bit adder to achieve its functionality, not a full adder. To do this, we split the least significant 2 bits and added 1 to the remaining 30-bit number, and recombined them, as this is equivalent to adding 4.



Our pipeline register merely contains the instruction and the PC+4 value. We implemented our pipeline register in a sub circuit using two registers in parallel and connecting them to the same clock. This is the same structure for subsequent pipeline registers.

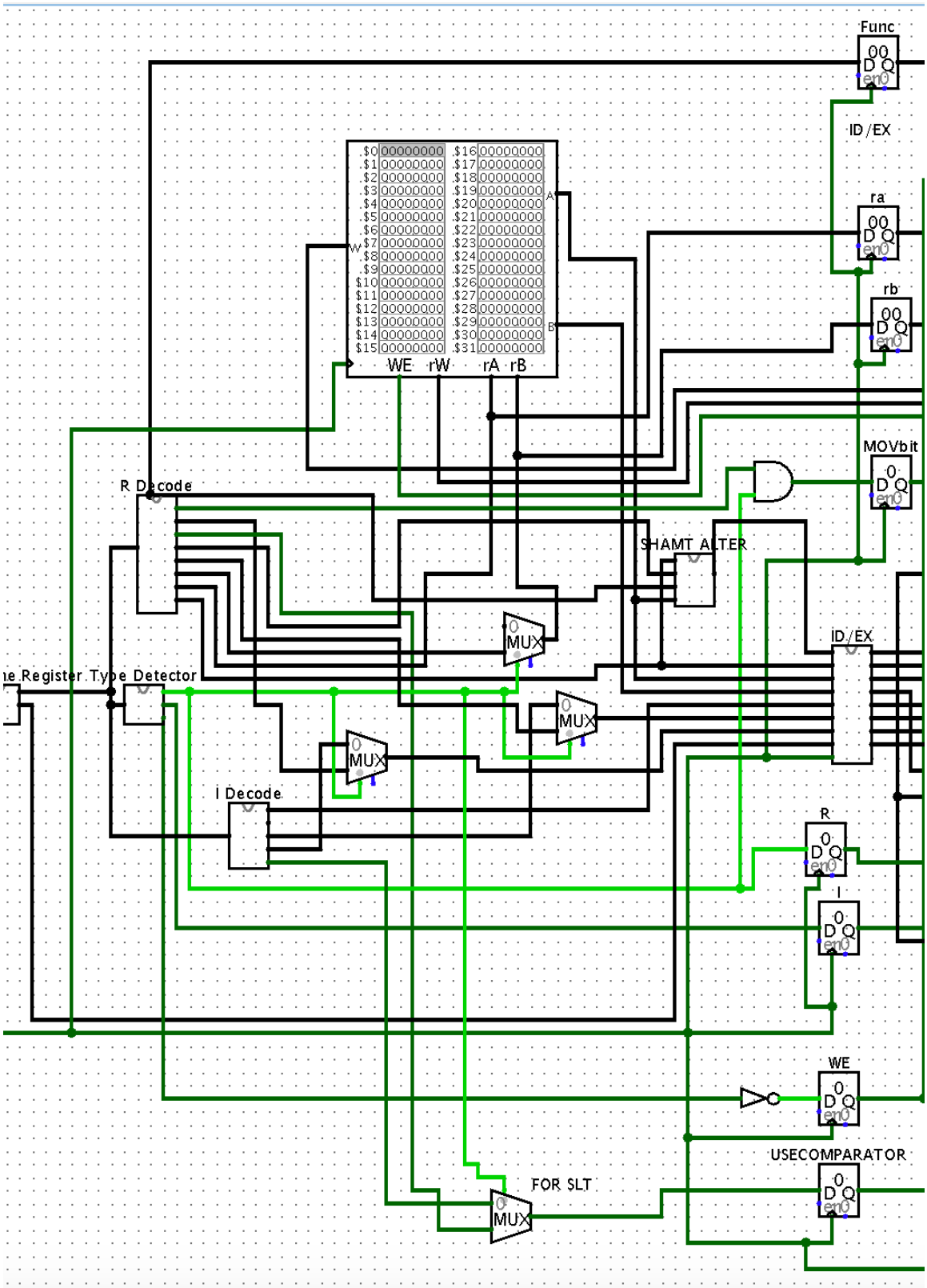
4 The Instruction Decode Stage



This initial diagram shows our ID stage without any hazard detection or control/decode logic, hence why it is so clean. But at the highest level, this is essentially how our ID stage still behaves. The 32-bit instruction is split, and the Opcodes and values encoded in the instruction will run through the stage and extract the correct values from the register file and store them into the ID/EX pipeline register. This would be easy if there was only 1 type of instruction that we had to decode, but our MIPS processor must handle all three types: R, I, and J type instructions.

Our decode stage was organized by splitting its decode logic into separate components, one for R type instructions, and one for I type instructions. J type instructions are dealt with by disabling the write enable on the register file. Multiplexers were then used, with the corresponding control bits supplied by a type detector sub circuit, to control which values are eventually placed into the pipeline register.

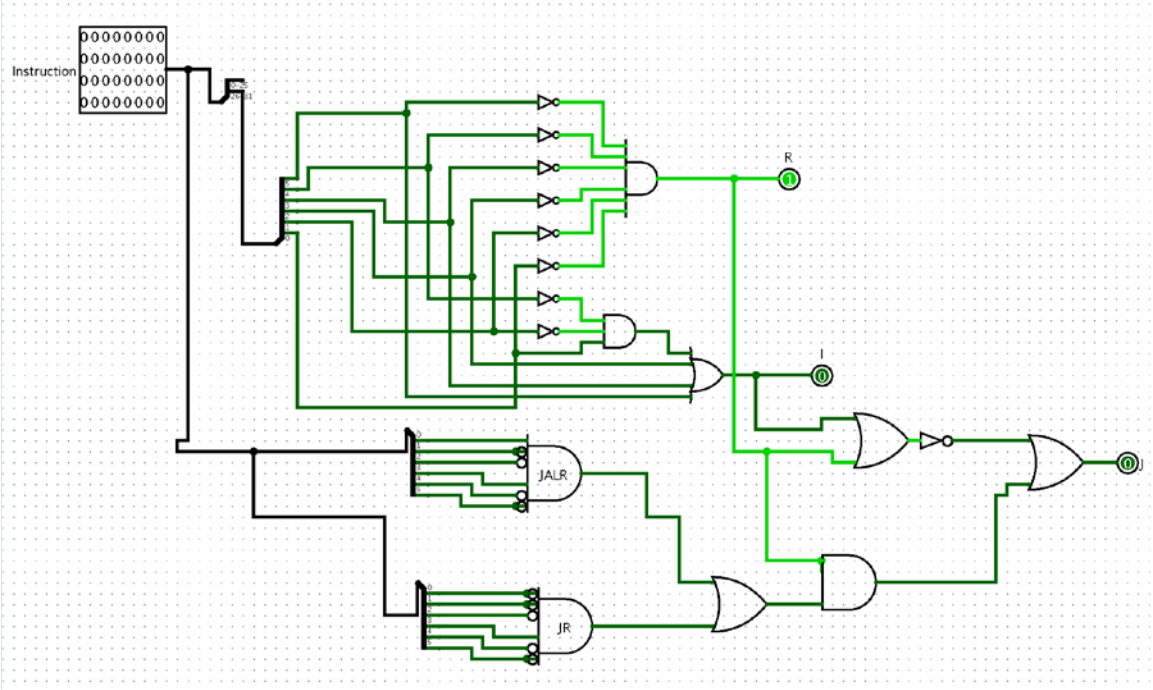
Our implementation is shown below.



Instruction Decode Stage

One thing we realized after we began making our processor was that we would need more control bits than we had anticipated. But we had already created our ID/EX pipeline with registers for what we had initially imagined in our preliminary design doc. Hence why in our diagram, there are several registers outside the central ID/EX pipeline register block. Most of these new registers are extracted control bits that help select what will happen in the execution stage. We decided not to condense these individual control bits into a larger sized control signal, because that would require even more crazy wires running over our diagram than there currently are.

Glancing at our ID stage, you will notice that the full 32 bit instruction is run through 3 sub circuits immediately; an R decoder, an I decoder, and Instruction Type detector. Lets start by looking into our Instruction type detector:

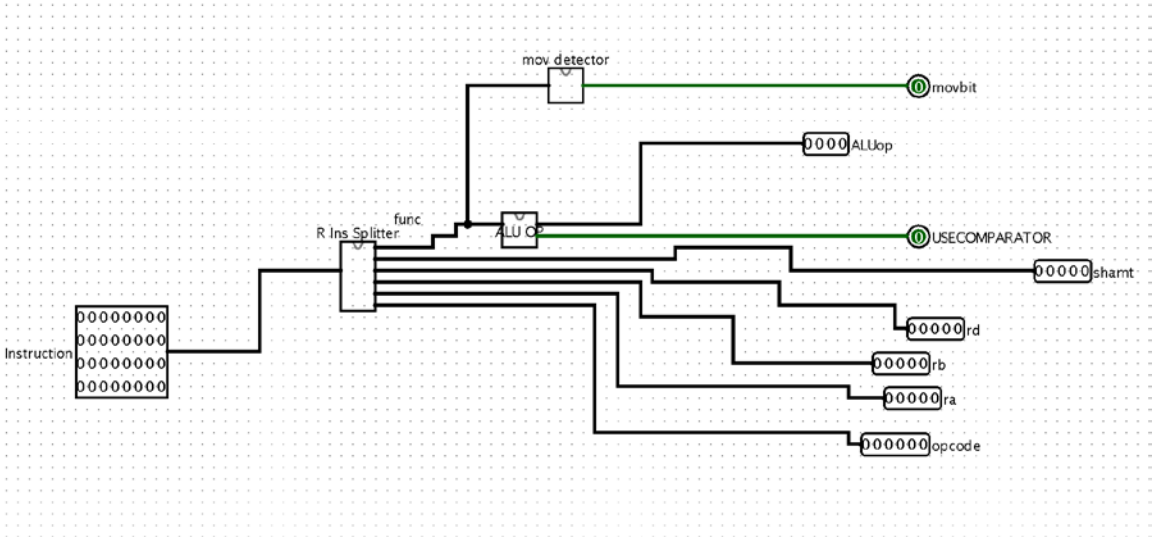


Instruction Type Detector

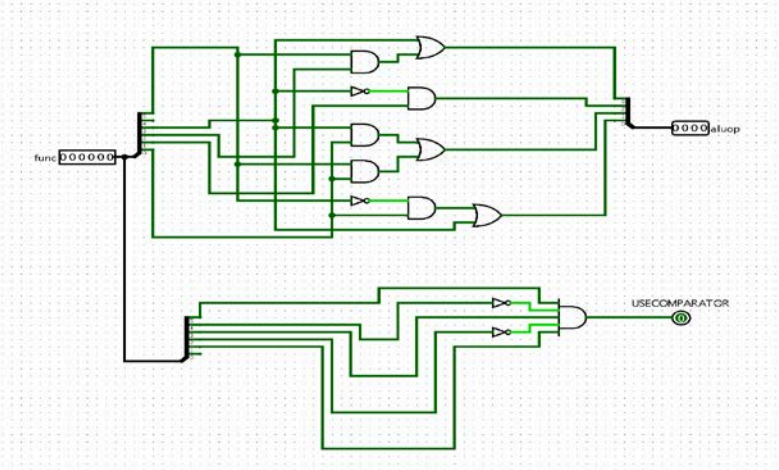
This sub circuit was built using two key pieces of information provided on the PowerPoint slides for this project. It takes in the full instruction as the input and outputs 3 control bits that tells the type of the instruction. The first piece is that for all R type instructions, the opcode is 000000. So our circuit outputs a 1 for R if this is the case. The opcodes of I-type instructions can be anything other than 000000, 00001x, 0100xx, so we took the negation of this and ran it through combinatorial analysis to create the decoding logic. Also, if both R and I are 0, then it must be a J type. In addition, there are two R type instructions (JALR and JR) that are classified as R types but are technically jumps, and since we do not have to deal with them in this project, we hardcoded the detection for the opcodes of either instruction to also

output J =1, since this J bit determines whether our instruction will be write enabled or not. The R and I bits are stored in the pipeline register.

R DECODE:



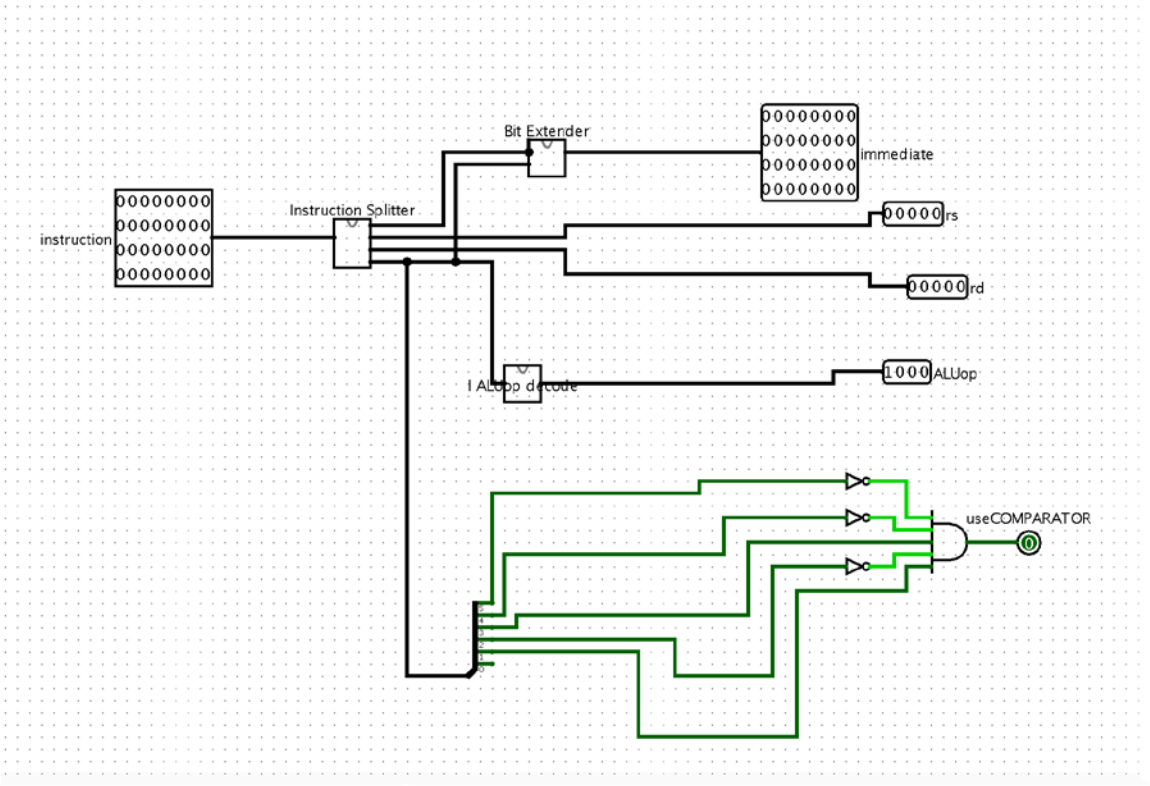
This is all the logic for our R-type instruction decoder. The R-ins Splitter sub circuit simply splits the instruction into the 6 relevant pieces (op, rs, rt, rd, shamt, func) and thus I will not show a diagram of it in this document. The shamt, rd, rb, ra, and op outputs are all extracted directly from the instruction. The mov detector sub circuit runs the func code of the R-type and checks if it matches the MOVZ or MOVN function. If it does, it sets the value of the movbit control bit to 1. Likewise, the ALU OP sub circuit is a 6 to 4 bit decoder that takes the func code of the R-type instruction and outputs the 4 digit opcode that will be put in the ALU during execution. It also detects for the func codes for SLT and SLTU, and outputs a control bit that will determine whether to use a comparator rather than the ALU in the execution stage. Our 6-4 decoder is below:



Function to ALU Op Decoder

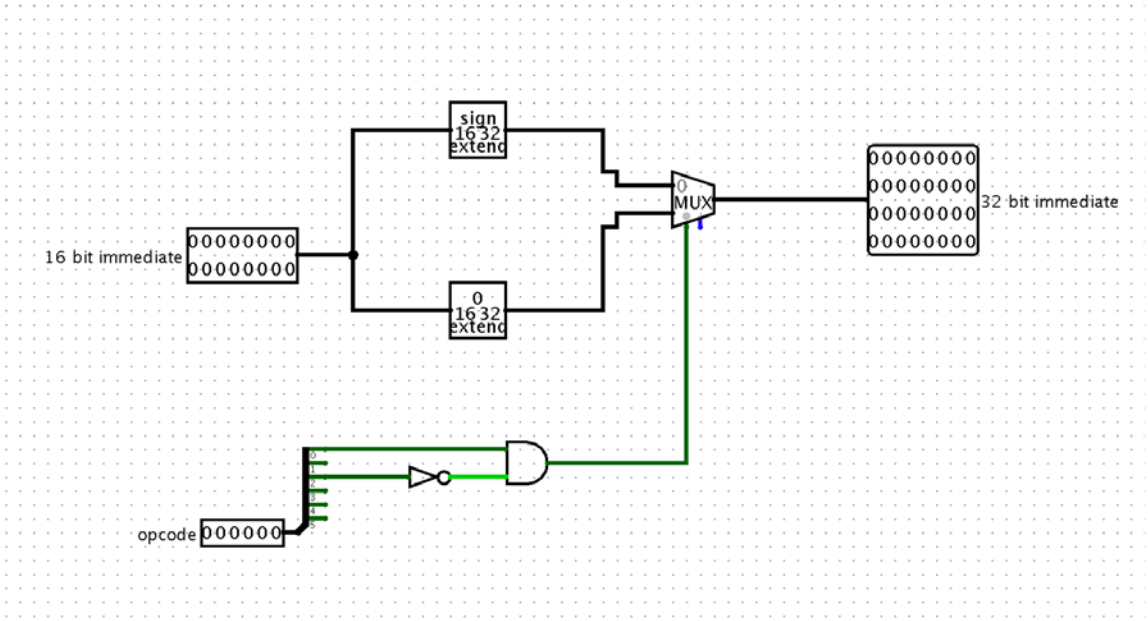
I DECODE:

Next up as we traverse through our instruction decode stage, is our I-type instruction decoder. It takes in the instruction from the IF/ID pipeline register and processes it through the circuit below. It is similar to the R decoder previously explained. Here it is shown below:



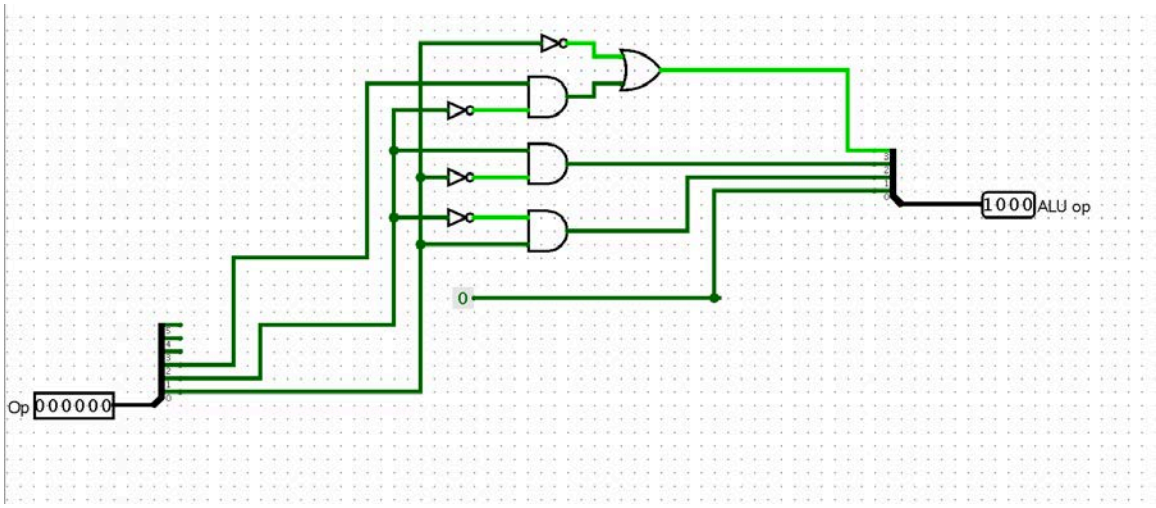
I-Type Decoder

Once again, the splitter simply splits the 32 bit instruction into the relevant op, rd,rs, and immediate values. The rs and rd values do not need any further decoding. For the immediate though, we must determine whether to sign extend or zero extend the 16-bit immediate. For I-types, it is always sign extended unless the instruction is an explicit unsigned operation, such as ADDIU. Our Bit extender sub circuit below implements this:



Bit Extender

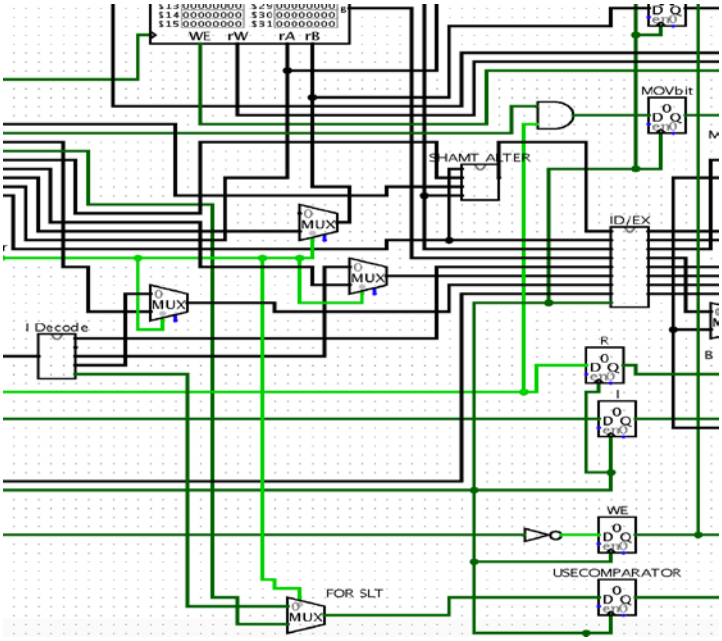
The last and third to last bits determine whether an instruction will be signed or unsigned. Moving on, we also need another 6-4 decoder, this time for I type instructions. Since I types have different op codes, we decode not the function field as we did in R types, but the opcode itself. We took the relevant instructions and input them into combinatorial analysis. The decoder is shown below:



6-4 I-Type Opcode to ALU Op Decoder

Finally, the last part of our I-Decoder is an AND gate that detects for the opcodes for SLTI and SLTIU instructions. This will enable the USECOMPARATOR bit, which tells a mux in the EX stage to ignore the ALU and instead take the value from the comparator that we made.

MORE ID DECODE:

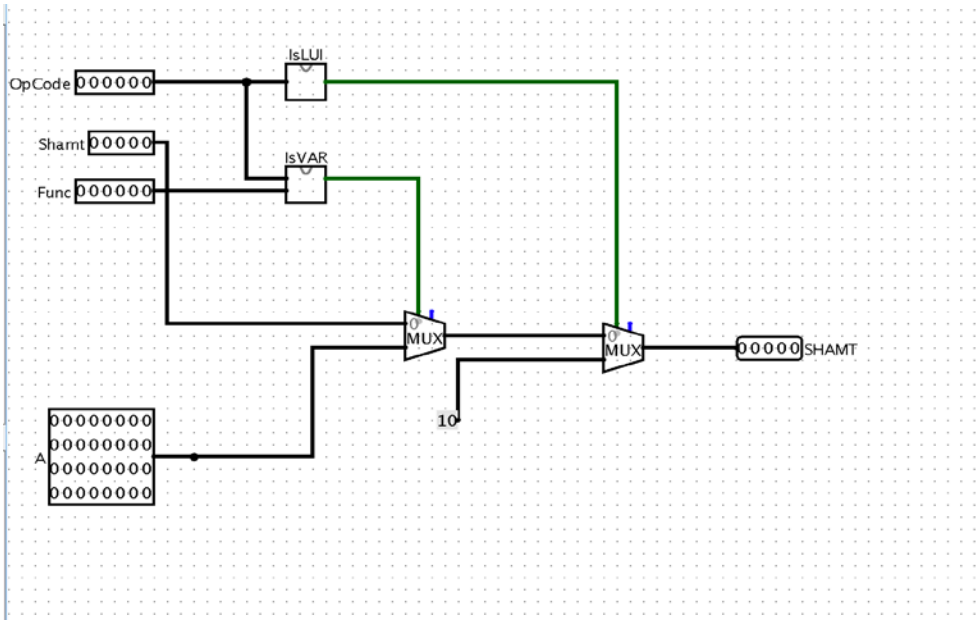


Progressing through our ID stage, we now know the instruction type, and we have decoded all the bits for both possible instructions. Now, we just select which of the two values we will put into the ID/EX pipeline register. This is what a majority of the multiplexers here are for. The green wire is the R control bit, and is used as the control bit for these muxes. If R=1 then the mux will send the R type bits through, and vice versa. (Since if R=1, I must equal 0, cant have an instruction that is both R and I.) One important mux however, is the one for the rB value. For immediate instructions, rB does not have any input, so a mux with an empty 0-input handles this. The R-control bit is still the mux control bit.

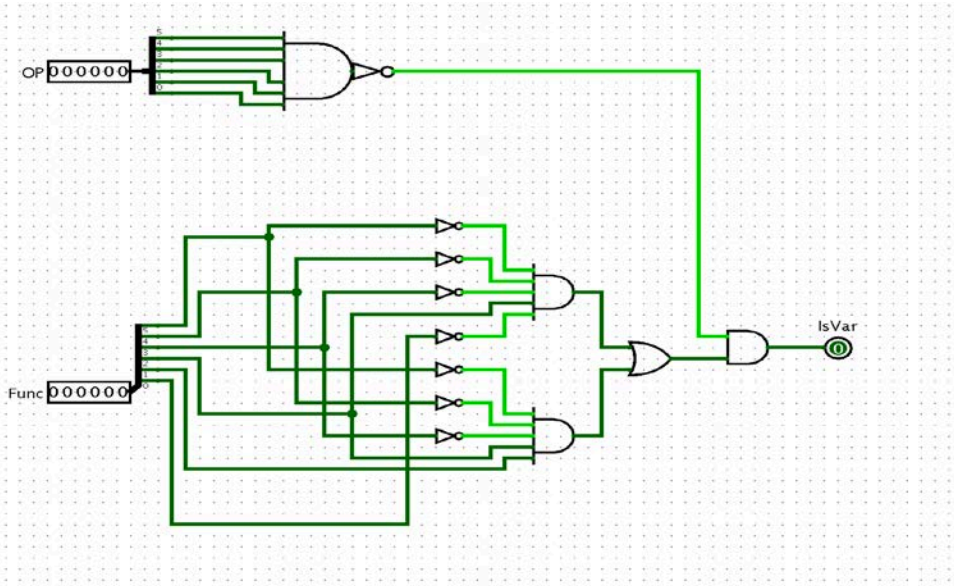
One final feature of our ID stage is the SHAMT decoder. It takes in 4 inputs:

1.Shamt from R-Decode,
2.Opcode
3. Function code from R-Decode
4.Value from register A

Its job is to handle what to use as the shift amount for the shift instructions, including the variable shifts (SLLV, SRAV, SRLV), and LUI. It is shown below:

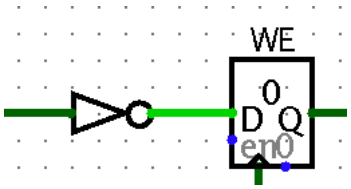


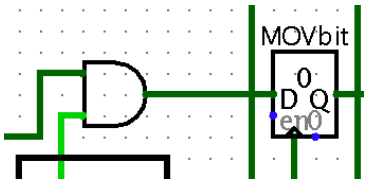
Two sub circuits detect for LUI instructions and variable shift instructions. If it is a variable shift instruction, the SHAMT becomes the last 5 bits of the value in rA. If it is a LUI instruction, SHAMT must be 16. The LUI circuit simply look at the Op code and runs it through an AND gate that matches the Op code for LUI. It is a bit more complicated for our IsVAR circuit. It is shown below:



For IsVAR, we first checked if it is an R type, and then checked if the function code matched any of the variable shifts. This is run through an AND gate to determine the value of the IsVar control bit.

Two final parts of our Decode that you may notice are shown here:





The negation on the left is for the J- control bit introduced earlier. If it is a J type, we simply turned off write enable. On the right side, we want MOVbit to be 1 only if the decoder from R says the MOVbit is 1. But it could be 1 even if it wasn't 1; this happens if the instruction is not R type. This AND gate simply confirms that the instruction type is indeed R before storing the MOVbit.

Truth Table for R Type func-ALUop Decoder:

	b5	b4	b3	b2	b1	b0	OP
SLL	0	0	0	0	0	0	000x
SRL	0	0	0	0	1	0	0100
SRA	0	0	0	0	1	1	0101
SLLV	0	0	0	1	0	0	000x
SRLV	0	0	0	1	1	0	0100
SRAV	0	0	0	1	1	1	0101
ADDU	1	0	0	0	0	1	001x
SUBU	1	0	0	0	1	1	011x
AND	1	0	0	1	0	0	1000
OR	1	0	0	1	0	1	1010
XOR	1	0	0	1	1	0	1100
NOR	1	0	0	1	1	1	1110
SLT	1	0	1	0	1	0	xxxx
SLTU	1	0	1	0	1	1	xxxx
MOVZ	0	0	1	0	1	0	1001
MOVN	0	0	1	0	1	1	1011

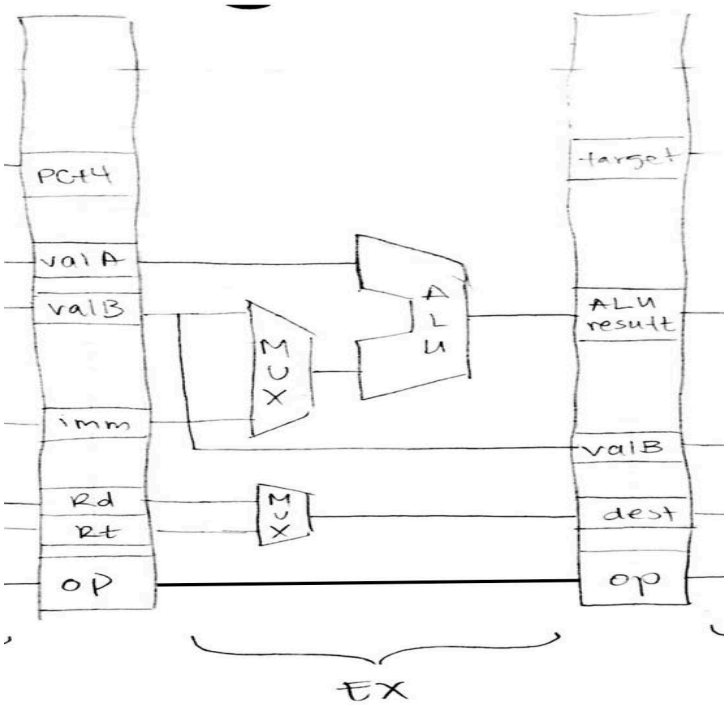
b5-b0 refer to the last 6 bits of the instruction, which is the function field.

Truth Table for I Type Op-ALUop Decoder:

INS	b31	b30	b29	b28	b27	b26	ALU OP
ADDIU	0	0	1	0	0	1	001x
SLTI	0	0	1	0	1	0	xxxx
SLTIU	0	0	1	0	1	1	xxxx
ANDI	0	0	1	1	0	0	1000
ORI	0	0	1	1	0	1	1010
XORI	0	0	1	1	1	0	1100
LUI	0	0	1	1	1	1	000x

b31-26 are the first 6 bits of the instruction, which is the Op field. Note how SLT instructions do not matter since we run them through the comparator instead.

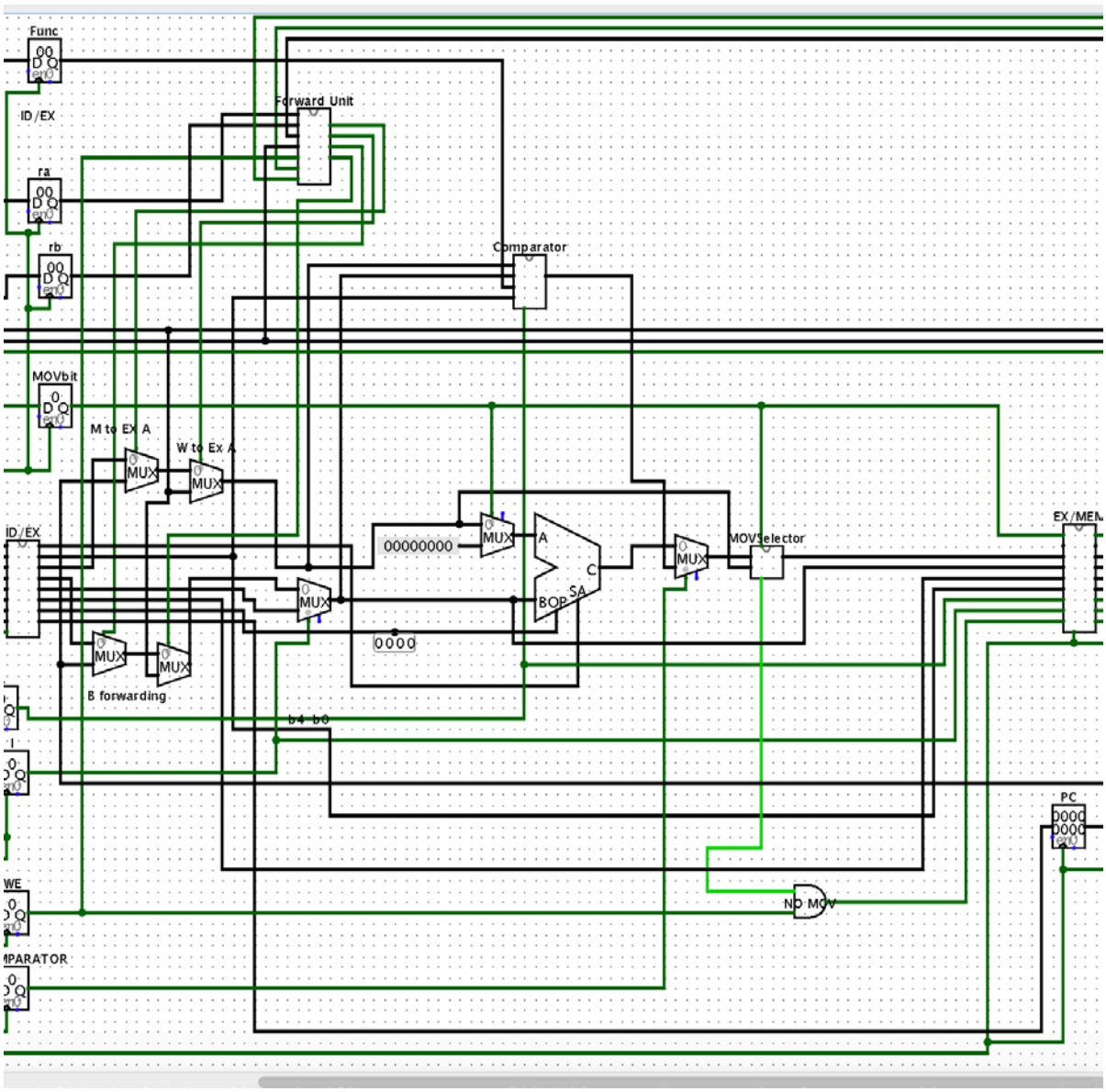
5 Execution Stage



The diagram has 2 componenets:

- ALU (arithmetic logic unit)
- Mux's

The load instruction reads the contents of Val A, Val B, and the immediate value, using a multiplexor to decide whether to use the immediate value or Val B in the instruction. The values are then fed through the ALU, and the result is put in the EX/MEM pipeline register. Muxes are also used to find the destination and also stored in the EX/MEM register. This preliminary EX stage diagram does not include a data forwarding unit, and some other important multiplexers that guide the execution of the instruction. Our full implementation of the EX stage is shown on the next page:

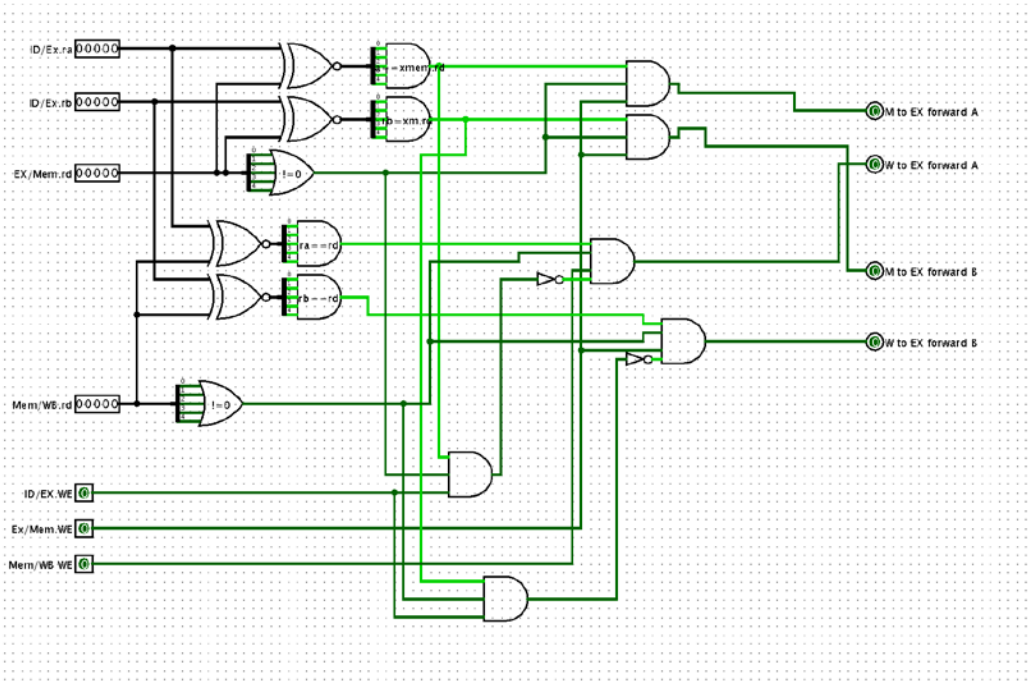


EX Stage Implementation

At a glance we can see there is a data forwarding unit at the top of the diagram that takes in inputs from the pipeline registers of ID/EX, EX/M, and M/WB. The logic for this forwarding unit comes straight from the lecture slide:

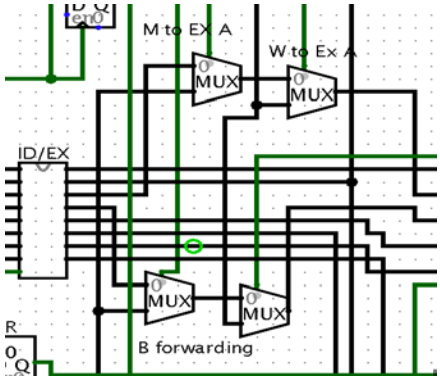
```
forward = (M/WB.WE && M/WB.Rd != 0 &&
                ID/Ex.Ra == M/WB.Rd &&
          not (ID/Ex.WE && Ex/M.Rd != 0 &&
                ID/Ex.Ra == Ex/M.Rd)
     || (same for Rb)
```

Check pg. 369

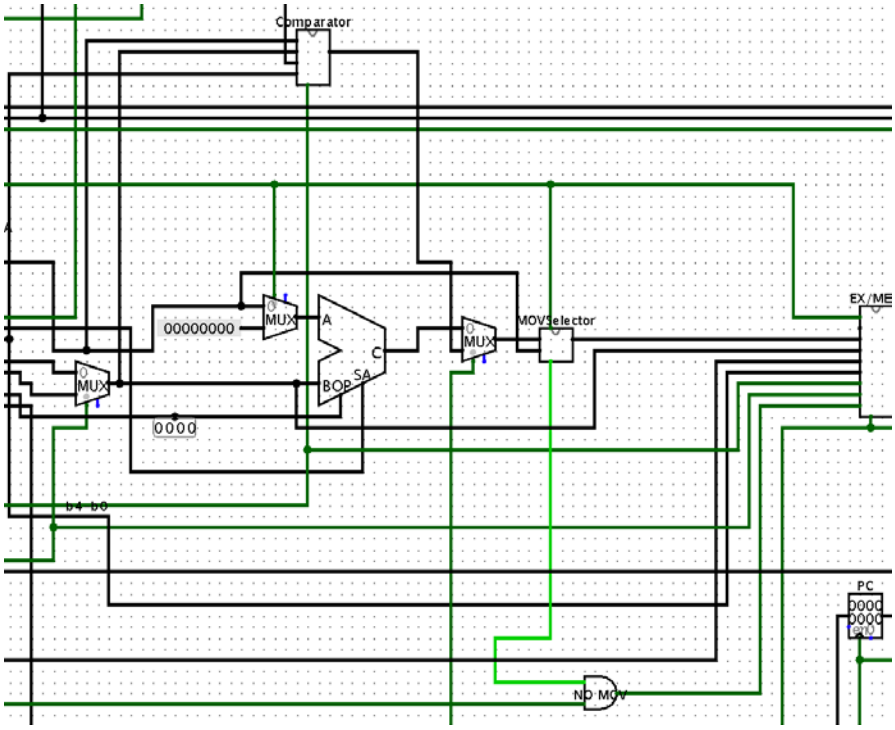


Forwarding Unit

These 4 control bits will be 0 unless a hazard is detected. When it is detected, these bits will change the value of either A or B to the forwarded, correct value from a later register. This is shown with the 4 muxes in the beginning of the EX stage:



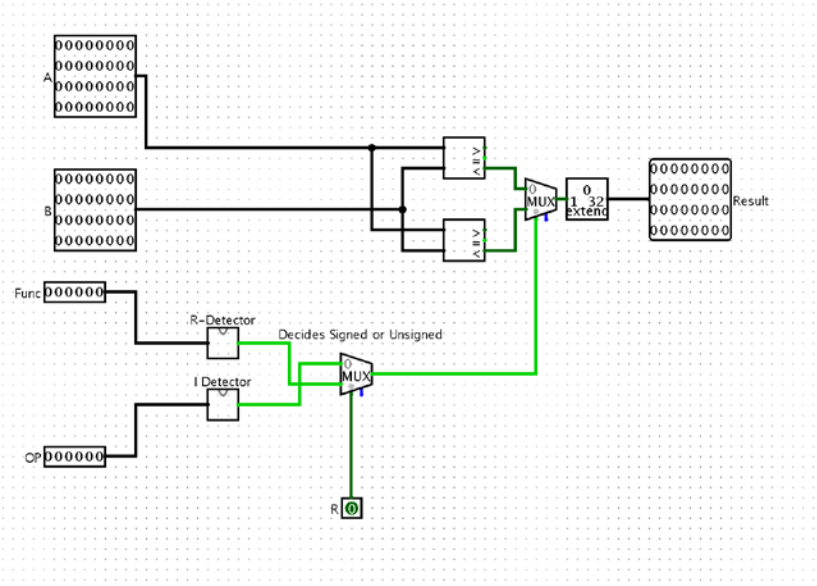
Data Forwarding Muxes. Note that if the value of the control bits are 0, the A/ B value proceeds through as normal.



Second half of EX Stage

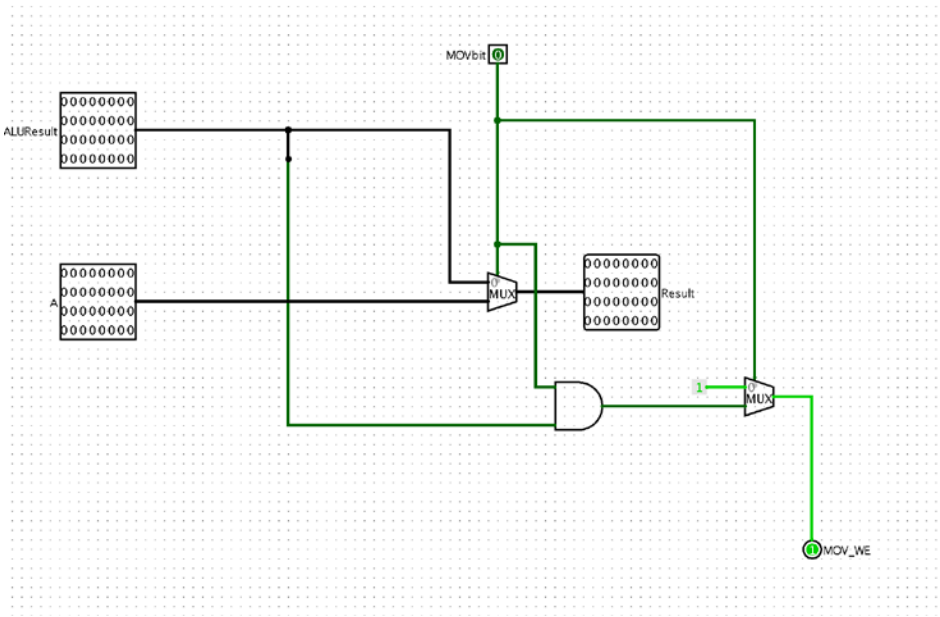
Our EX stage is reprinted above for convenience. The mux before the A input to the ALU is for MOV instructions. Since we decided to implement MOV using the ALU, rather than the comparator, we need to compare the value in B to 0. So if MOVbit is 1, A becomes 0. The mux after the ALU is controlled by the USECOMPARATOR bit and decides whether the instruction requires the result from the comparator, or the ALU. So this would be the SLT instructions.

Comparator Unit:



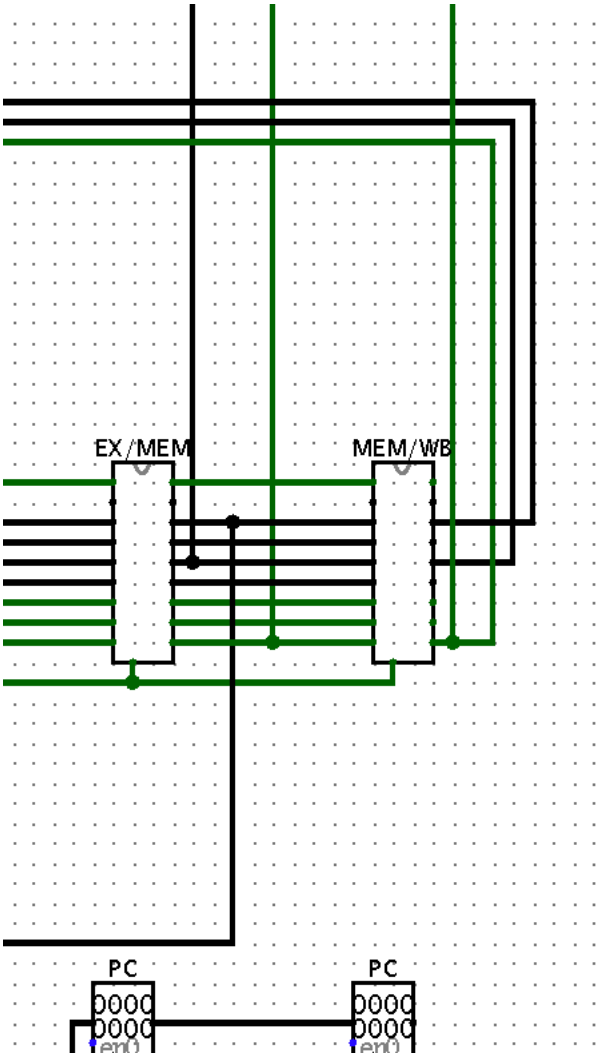
Our comparator unit takes in A, B, R, Func, and Op from the ID/EX pipeline register. It handles SLT, SLTI, SLTU, and SLTIU instructions. One comparator is set to handle Two's Complement comparisons, and another is set to handle unsigned ones. Two decoders that merely look at the op/function codes will determine whether the shift function is signed or unsigned. The R control bit selects which decoder to use. This value will then become the select bit for which comparison to use. This is then zero extended to 32 bits.

MOV Selector:

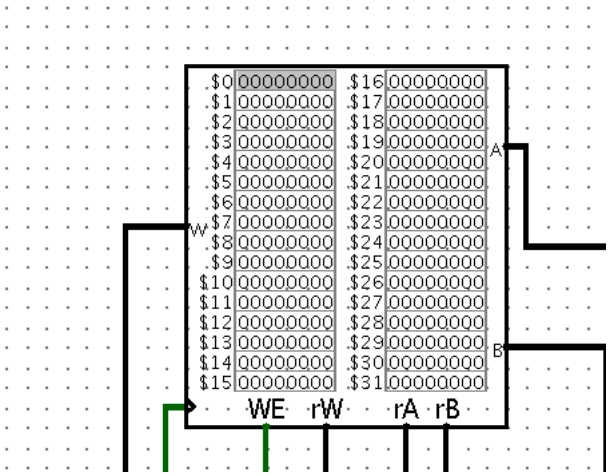


Our MOV selector unit takes in the ALU result, and the A value. Recall that the MOV functions move the value in A to register rd if our ALU comparison is true (the ALU runs either *eq* or *ne* with 0 and B). If the instruction is indeed a MOV, and the ALU result is 1, then the A will be written into rd. But if the MOV condition is not satisfied, then Result is still the ALU result, and WE is disabled. When it is not a MOV instruction, WE is always enabled. But when it IS, we must check that the ALU result is 1 before enabling. There is a final AND gate near the bottom that handles this new condition for WE.

MEM & WB



Our MEM stage is simply a write back. The A and B values are wired back to the EX stage and become inputs into the data forwarding muxes for A and B. The WE bit is needed for the forwarding unit. The clock is connected below. For our next project we will add the memory unit and this stage will become more important.



Register File Writeback

Finally, we can write back into our register file! Our rd output in the MEM/WB pipeline register gets sent into rW, and WE goes into WE. Our computed value result gets sent into W to be written. Note that things can only be written if WE is 1. Also, our register file is sent to falling edge so that the first half of the cycle can be used to calculate. This makes our processor capable of handling both read and write in the same cycle without introducing new hazards.

Testing

For testing, we tested each individual instruction with around 4-10 manually generated test cases. We stored values 1-10 in registers 1-10 first by using ADDIU. We then repeatedly referenced these registers (and avoided rewriting into them) to make it easier to write and evaluate our test cases. We tested our forwarding unit by intentionally writing instructions that we know would require data that has not been rewritten in the register yet.

```
ANDI $11, $0, 1 #0
ANDI $12, $1, 0 #0
ANDI $13, $0, 0 #0
ANDI $14, $1, 1 #1

ORI $15, $0, 1 #1
ORI $16, $1, 0 #1
ORI $17, $0, 0 #0
ORI $18, $1, 1 #1

XORI $19, $0, 1 #1
XORI $20, $1, 0 #1
XORI $21, $0, 0 #0
XORI $22, $1, 1 #0
```

Above is a sample of our test cases. We commented our expected values next to the instruction and manually checked the register file display to confirm the correctness of our MINI MIPS processor.

P1 Errors

Grading Comments & Requests (hide)

Grading Comment: Submitted by bhw45 on October 20, 2015 10:30AM
[2 errors] Tests that ADDIU sign extends its immediate operand.
Error in register 1. Expected 0xffffffff, but got 0x0000ffff.
Error in register 4. Expected 0xffff8000, but got 0x00008000.
[3 errors] Tests that ANDI, ORI, XORI zero extend their immediate operands.
Error in register 4. Expected 0x00008888, but got 0xcccc8888.
Error in register 6. Expected 0xcccceeee, but got 0xffffeeee.
Error in register 8. Expected 0xcccc6666, but got 0x33336666.
[0 errors] Tests the LUI instruction.
[0 errors] Tests hazards for LUI instructions.
[0 errors] Tests the MOVN and MOVZ instructions.
[0 errors] Tests hazards for MOV instructions where the operand is not moved.
[0 errors] Tests hazards for MOV instructions where the operand is moved.
[0 errors] Tests the R-type arithmetic and logic instructions.
[0 errors] Tests constant shifts.
[0 errors] Tests variable shifts with shift amounts < 32.
[0 errors] Tests variable shifts with shift amounts >= 32.
[2 errors] Tests forwarding of variable shift amount.
Error in register 5. Expected 0x05800000, but got 0x00000016.
Error in register 7. Expected 0x18000000, but got 0x00000018.
[0 errors] Tests the SLT instruction.
[0 errors] Tests the SLTI instruction.
[1 error] Tests the SLTIU instruction.
Error in register 19. Expected 0x00000001, but got 0x00000000.
[0 errors] Tests the SLTU instruction.
[0 errors] Ensures that I-type instructions work even if last six bits of immediate field match a function code of an R-type instruction.
[0 errors] Tests hazard detection: 1-cycle hazards.
[0 errors] Tests hazard detection: 2-cycle hazards.
[0 errors] Tests hazard detection: 3-cycle hazards.
[0 errors] Tests that the correct forwarding path is prioritized for simultaneous hazards.
[0 errors] Tests that hazards are resolved by bypassing, not by stalling.
[0 errors] Tests hazards: An I-type instruction writes to the same register as a previous R-type instruction. Immediate value must not be forwarded.
[0 errors] Tests that hazards involving $0 are handled correctly.
[4 errors] Runs all table B instructions, which should all be NOPs.
Error in register 1. Expected 0x00000000, but got 0x80000019.
Error in register 3. Expected 0x00000000, but got 0xfffffffc.
Error in register 4. Expected 0x00000000, but got 0xfffffffc.
Error in register 5. Expected 0x00000000, but got 0xfffffffc.
TOTAL: 12 errors
Tests with no errors: 20/25

We dealt with debugging these main errors in the previous Project 1 circuitry. We previously had misinterpreted the extension of the immediate values for ADDIU operations and for ANDI, ORI, XORI. By asserting the proper bit extender after the execution of the arithmetic; the problem was solved.

For SLTIU, we found a simple error in terms of the conversion of the immediate value, whereby it was sending an incorrect ALU code through the mem stage under strange circumstances. For that we just needed to go back into our Decoding logic and hardcode a condition for SLTIU.

Lastly, “Runs All Table B” should have come up as NOPs. The way we dealt with this error was we went back to Table B and manually reconstructed the logic to disable WE on any TABLE B brand instruction.

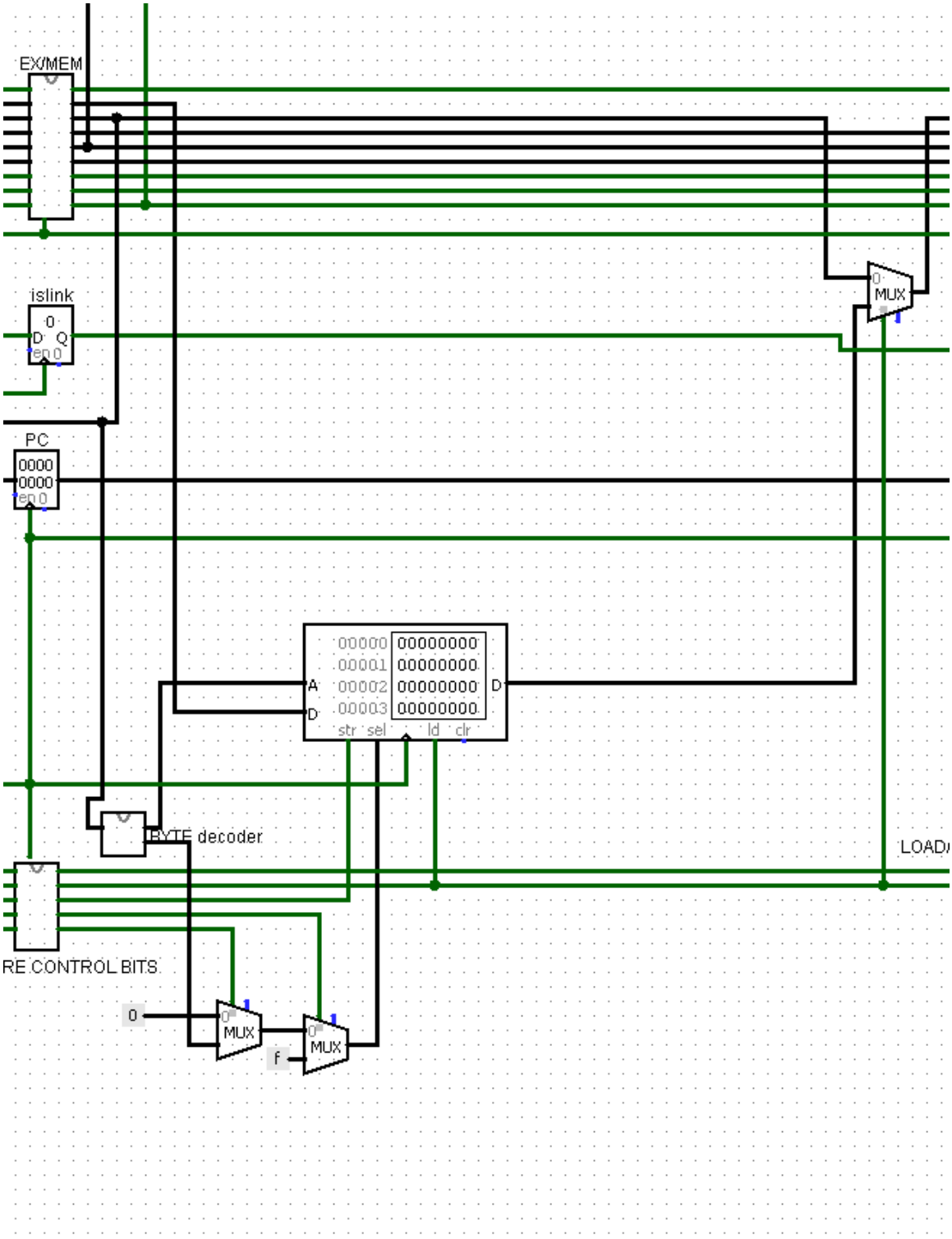
The Fruits of our labor left us at:

Grading Comments & Requests (hide)

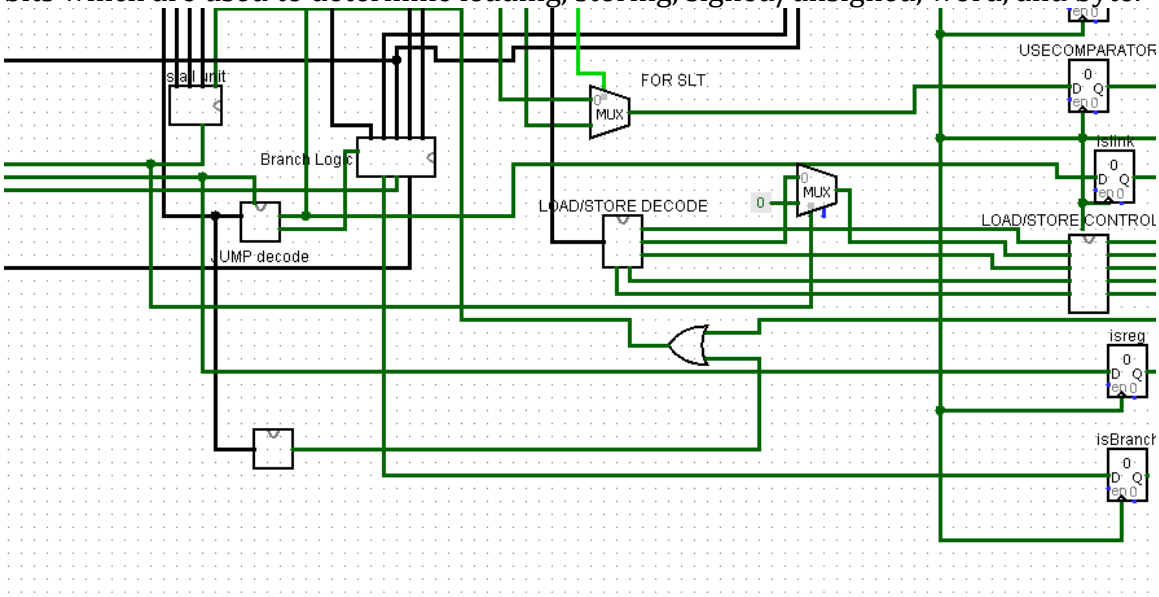
Grading Comment: Submitted by bhw45 on November 1, 2015 03:31PM

[0 errors] Tests that ADDIU sign extends its immediate operand.
[0 errors] Tests that ANDI, ORI, XORI zero extend their immediate operands.
[0 errors] Tests the LUI instruction.
[0 errors] Tests hazards for LUI instructions.
[0 errors] Tests the MOVN and MOVZ instructions.
[0 errors] Tests hazards for MOV instructions where the operand is not moved.
[0 errors] Tests hazards for MOV instructions where the operand is moved.
[0 errors] Tests the R-type arithmetic and logic instructions.
[0 errors] Tests constant shifts.
[0 errors] Tests variable shifts with shift amounts < 32.
[0 errors] Tests variable shifts with shift amounts >= 32.
[2 errors] Tests forwarding of variable shift amount.
Error in register 5. Expected 0x05800000, but got 0x00000016.
Error in register 7. Expected 0x18000000, but got 0x00000018.
[0 errors] Tests the SLT instruction.
[0 errors] Tests the SLTI instruction.
[0 errors] Tests the SLTIU instruction.
[0 errors] Tests the SLTU instruction.
[0 errors] Ensures that I-type instructions work even if last six bits of immediate field match a function code of an R-type instruction.
[0 errors] Tests hazard detection: 1-cycle hazards.
[0 errors] Tests hazard detection: 2-cycle hazards.
[0 errors] Tests hazard detection: 3-cycle hazards.
[0 errors] Tests that the correct forwarding path is prioritized for simultaneous hazards.
[0 errors] Tests that hazards are resolved by bypassing, not by stalling.
[0 errors] Tests hazards: An I-type instruction writes to the same register as a previous R-type instruction. Immediate value must not be forwarded.
[0 errors] Tests that hazards involving $0 are handled correctly.
[2 errors] The old P1 test program.
Error in register 27. Expected 0x05800000, but got 0x0000b000.
Error in register 31. Expected 0x4eed1001, but got 0x4b6da001.
TOTAL: 4 errors
Tests with no errors: 23/25

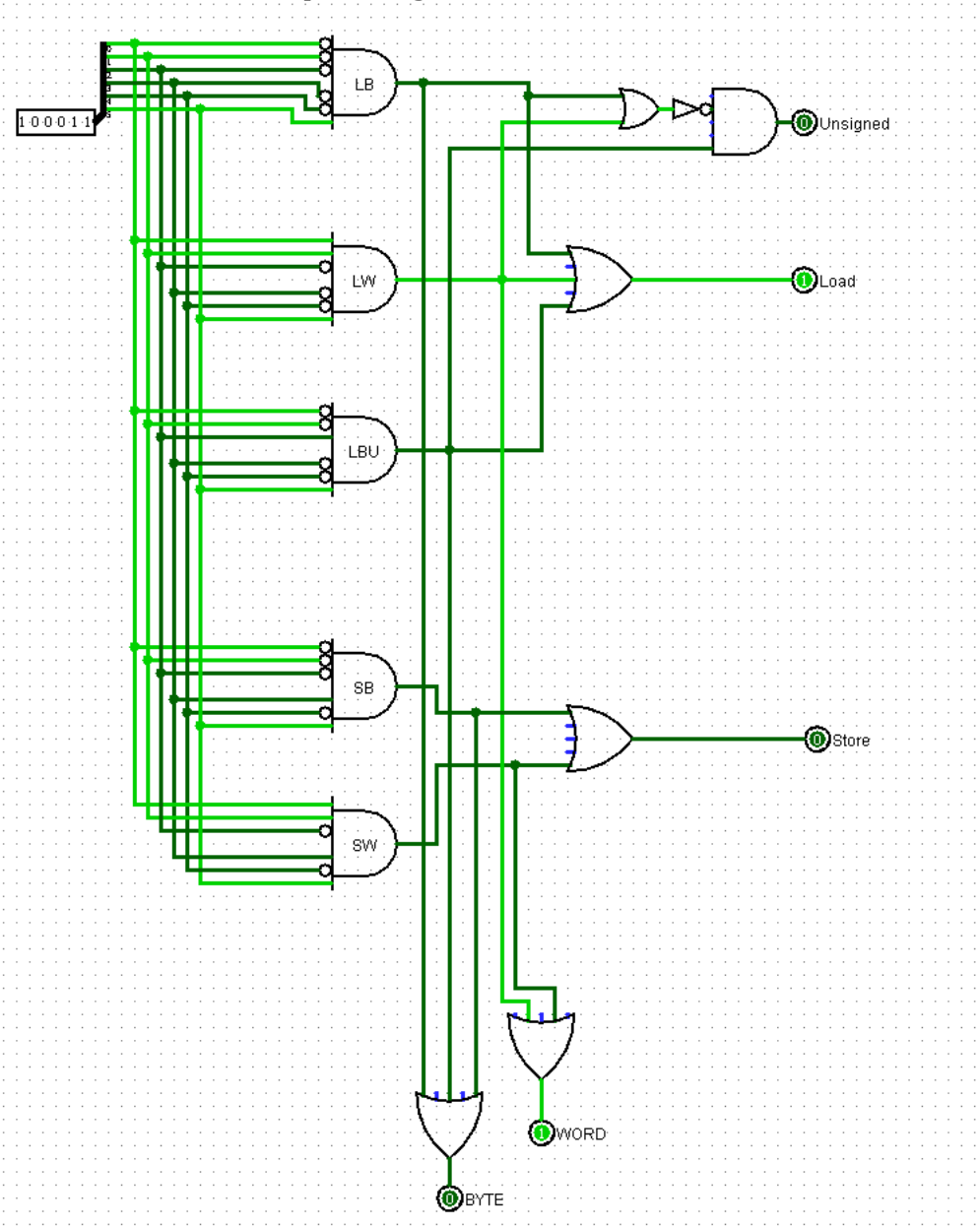
MEM STAGE


EX/MEM
islink
0
D Q
en 0
PC
0000
0000
en 0
00000 00000000
00001 00000000
A 00002 00000000 D
D 00003 00000000
str sel ld clr
BYTE decoder
RE CONTROL BITS
0
MUX
MUX
f
MUX
LOAD

This is our MEM stage. Quite simple looking, really. Its main function is to check whether or not a word or byte needs to be loaded or stored. In order to make this possible, there needed to be extra decoding logic within the Decode stage.
This new decoding logic, termed “Load/Store Decode” is shown below. It outputs 5 bits which are used to determine loading, storing, signed/unsigned, word, and byte.

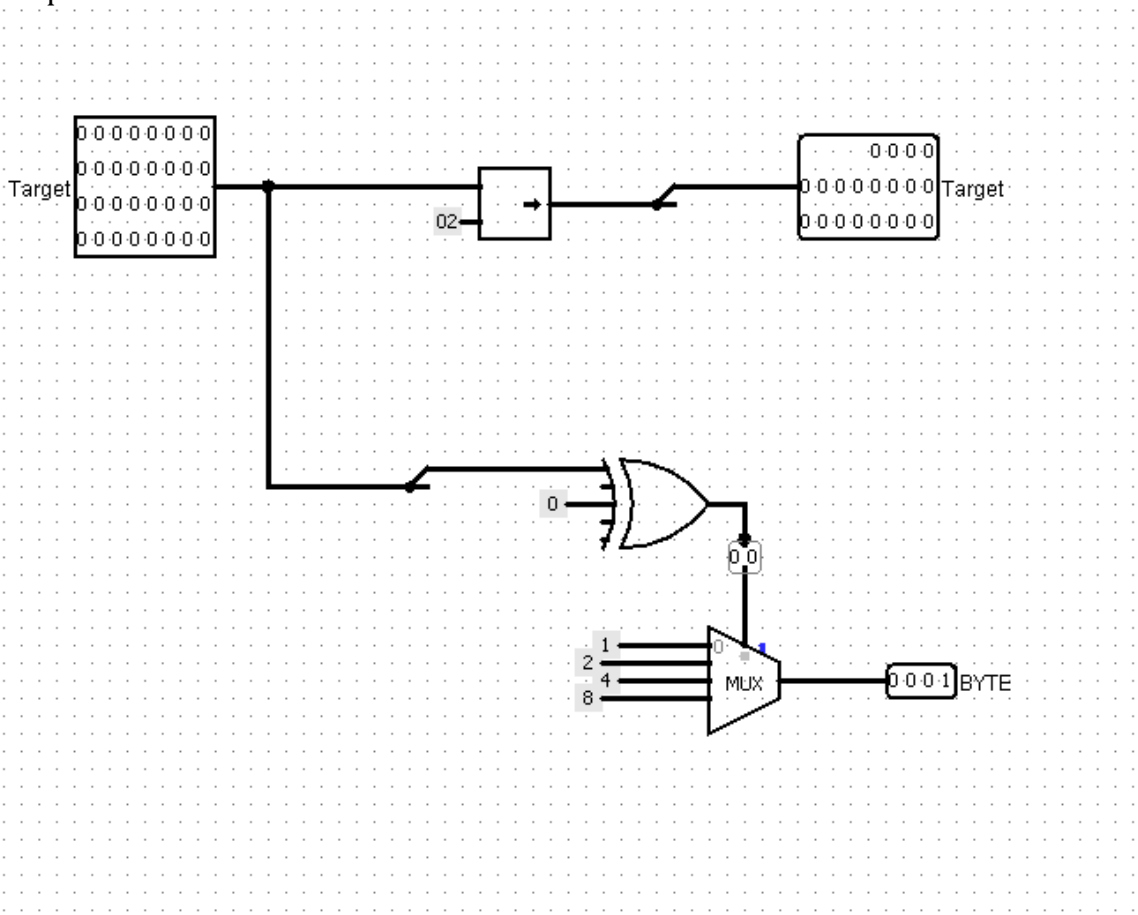


This information passes through the pipeline as well as the stalling unit (which will be mentioned later) in order to provide with efficient MIPS RAM access. The Load/Store subcircuit determines whether the OP code is running one of the 5 storage types we account for in this project, LB, LW, LBU, SB, SW. It's a very absolute subcircuit; not adhering to one of the 5 types will end up outputting a consecutive 0 bits in each of the 5 output categories.

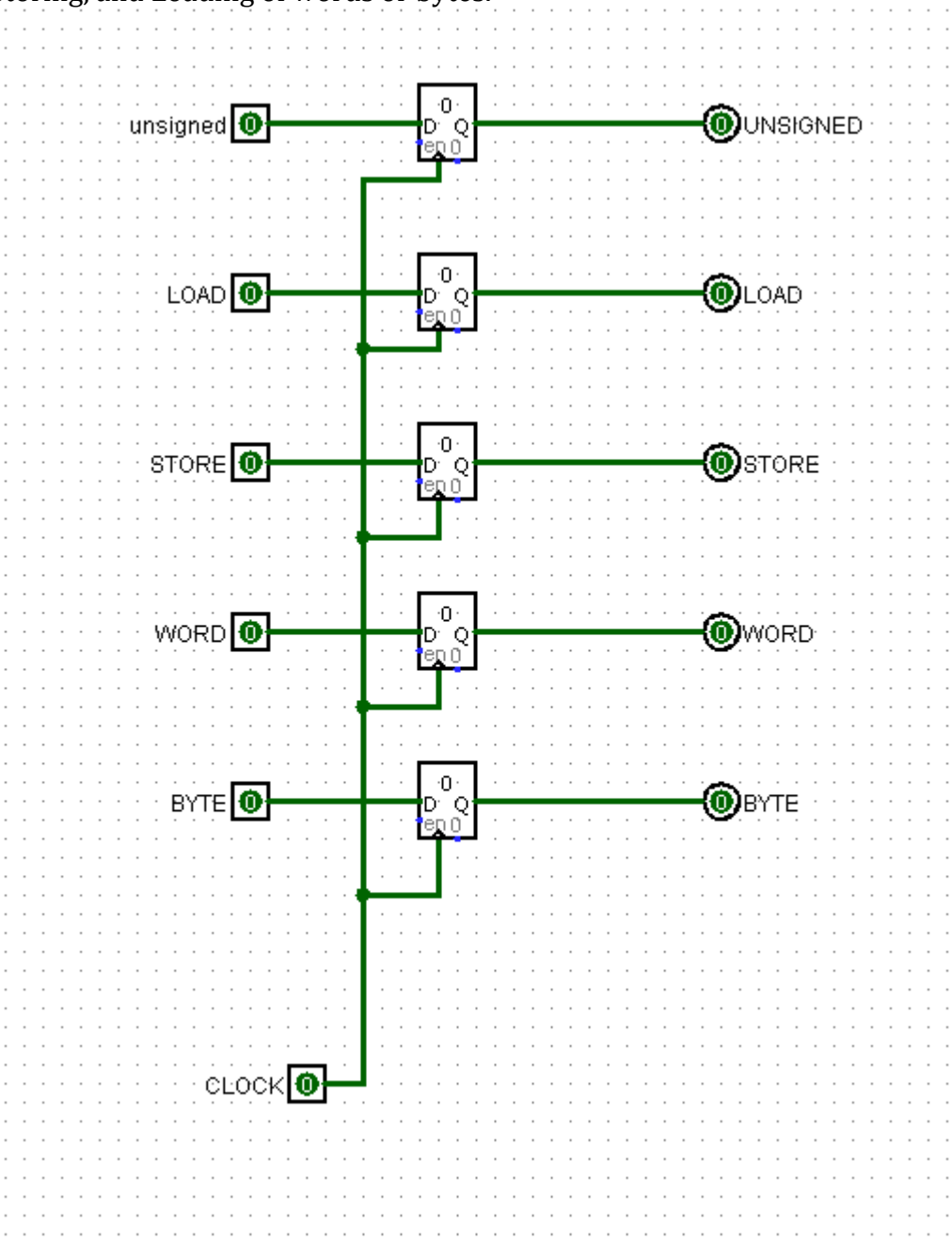


Next up is our BYTE Decoder. This is present within the MEM stage of the decoding process. This subcircuit is responsible for converting the MIPS user instruction into a referenceable address in the MIPS RAM. Our MIPS RAM operations on a 20bit input maximum meaning it can reference up to address 0xfffff.

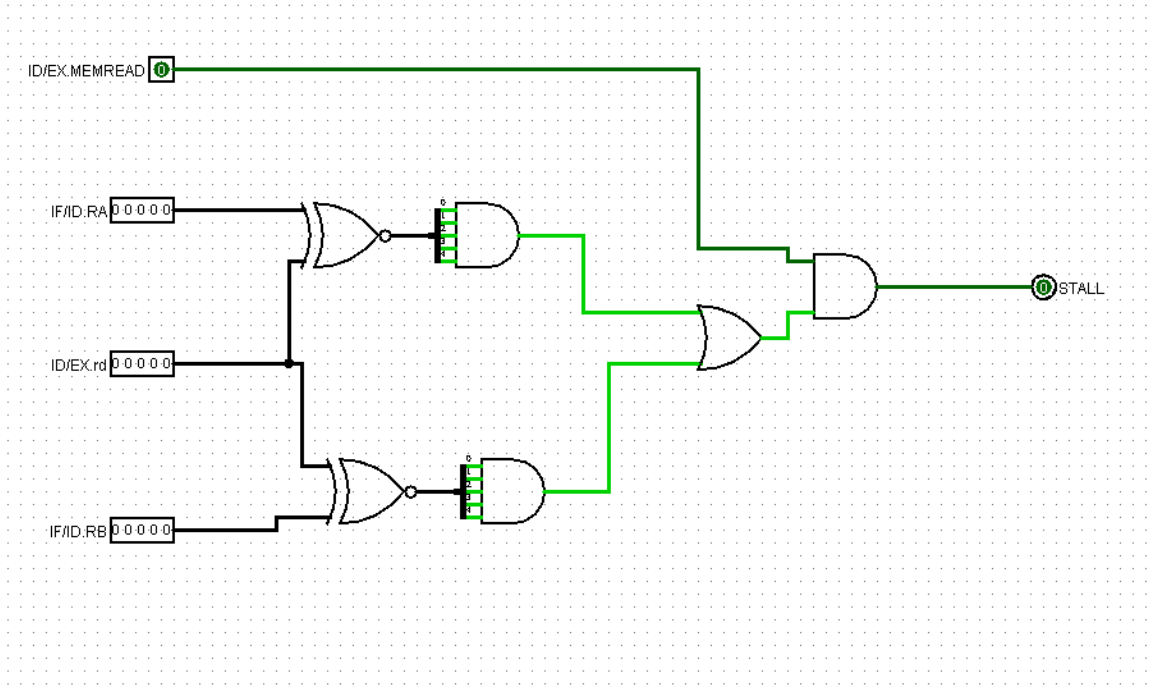
The BYTE decoder works by taking in the address calculated on offset by the ALU during the EX stage ("Target" - this value can also been seen as Result outside of the subcircuit) After going through a Shift Right Logical of 2 the value gets divided by 4 and can now address word values. However, in the case that Target is attempting to access a byte, we have the Target look at the last 2 bits of the Target input. We XOR the value with 0 in order to see what byte address the Target must be referencing. Afterward we use this XOR value in a MUX and then use constants to then reassign it to its representative 4 bit code that's set to be input in the MIPS RAM "Sel" component.



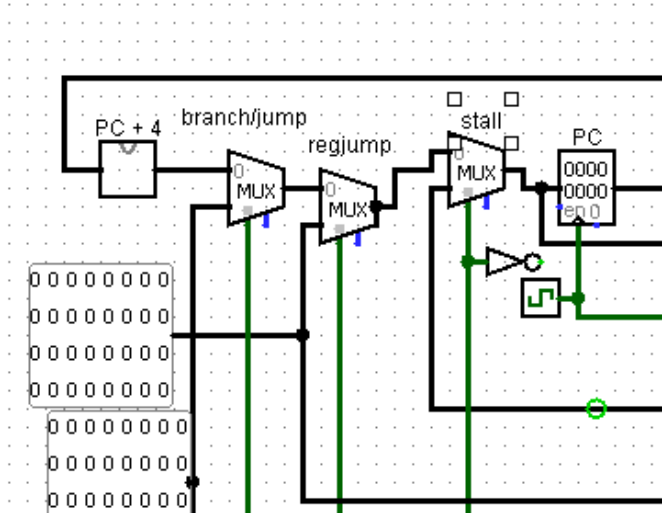
Next is the LOAD/STORE CONTROL BITS subcircuit. This circuit is primarily responsible for transferring the bits from the decode stage over the MEM stage and ensuring proper pipeline protocol is followed. The bits UNSIGNED/LOAD/STORE/WORD/BYTE all potentially have some influence on MIPS RAM either directly, or indirectly. All are needed in the grand scheme of the Stalling, Storing, and Loading of words or bytes.



Which brings us to another key portion of the MEM STAGE. The Stalling Unit. This is perhaps the crux in ensuring we avoid any possible data hazards that we could face relating to loading words immediately after an instruction stores a word into the similar register. Once it finds there's a data hazard that stems from the association of the register conflict arising from the register values in the EX versus the Register values in the ID stage, it then loads a bit—[STALL].

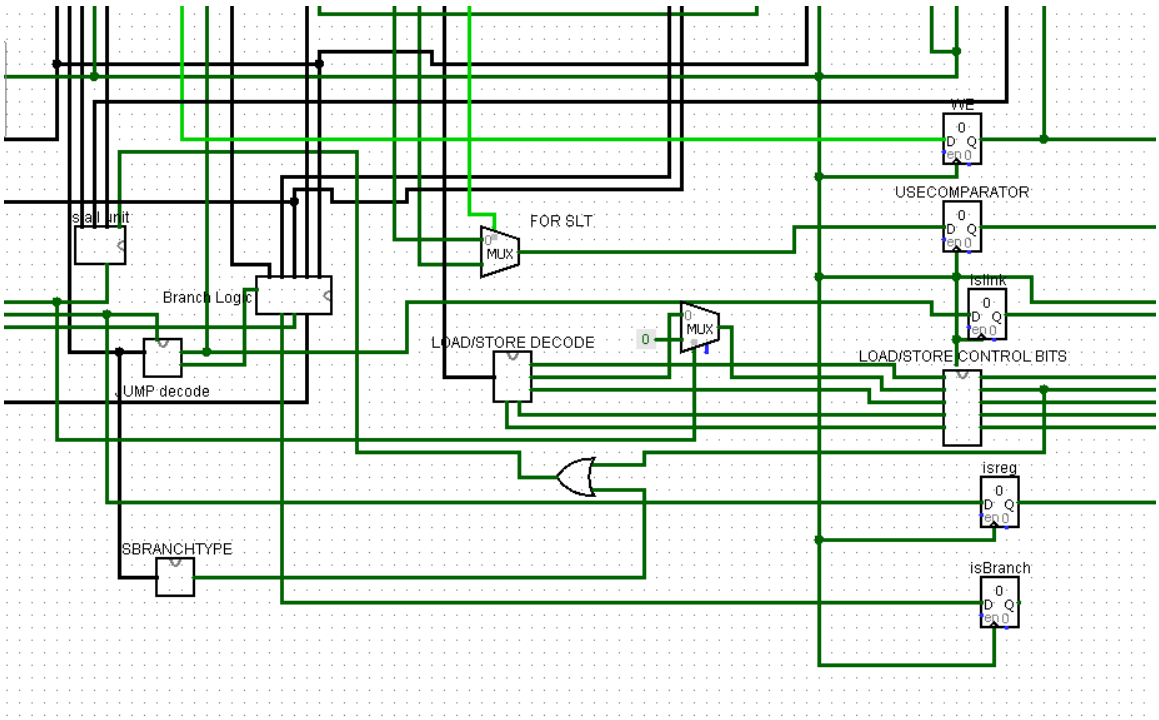


After calling on the STALL bit, it uses that value to MUX between the expected outcome without a stall and the PC+4 Value that stems from the IF pipeline. By doing this it stays to the current instruction, freezes the PC, IF/ID pipeline, and ID/EX pipeline and lets the other instructions run through until the writeback occurs successfully. Once that happens, the stalling has done its job and the instructions continue as normal.

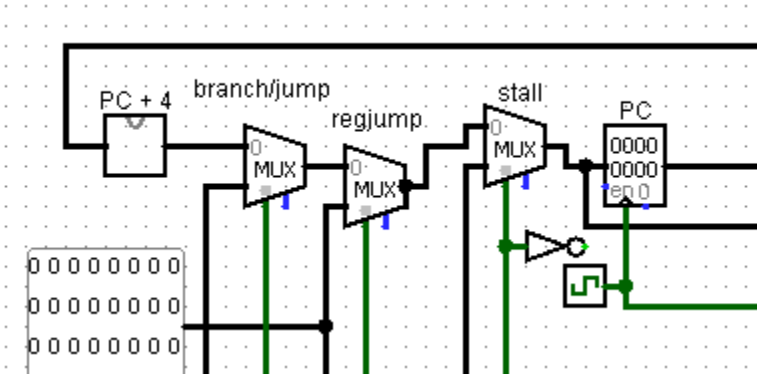


JUMP/BRANCH

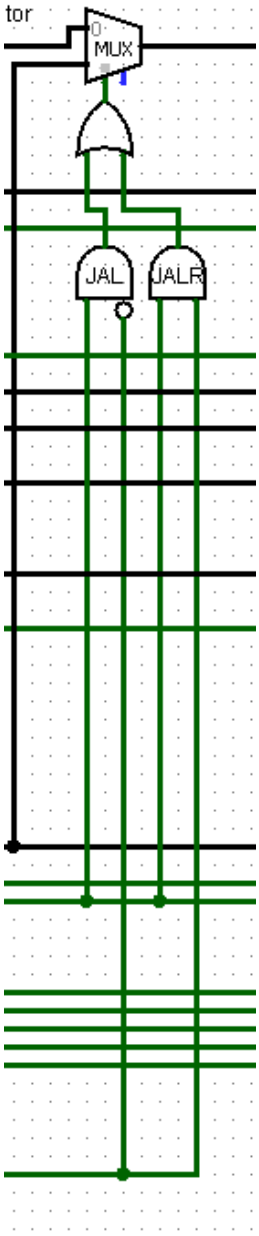
Overview



In a nutshell, most of the Branch and Jump logic happens within this space of the Circuit. On the far left we see all the circuitry that determines how we jump, when we jump and where we jump. This logic on how, where, and what amount we jump is then followed up by some muxing logic.



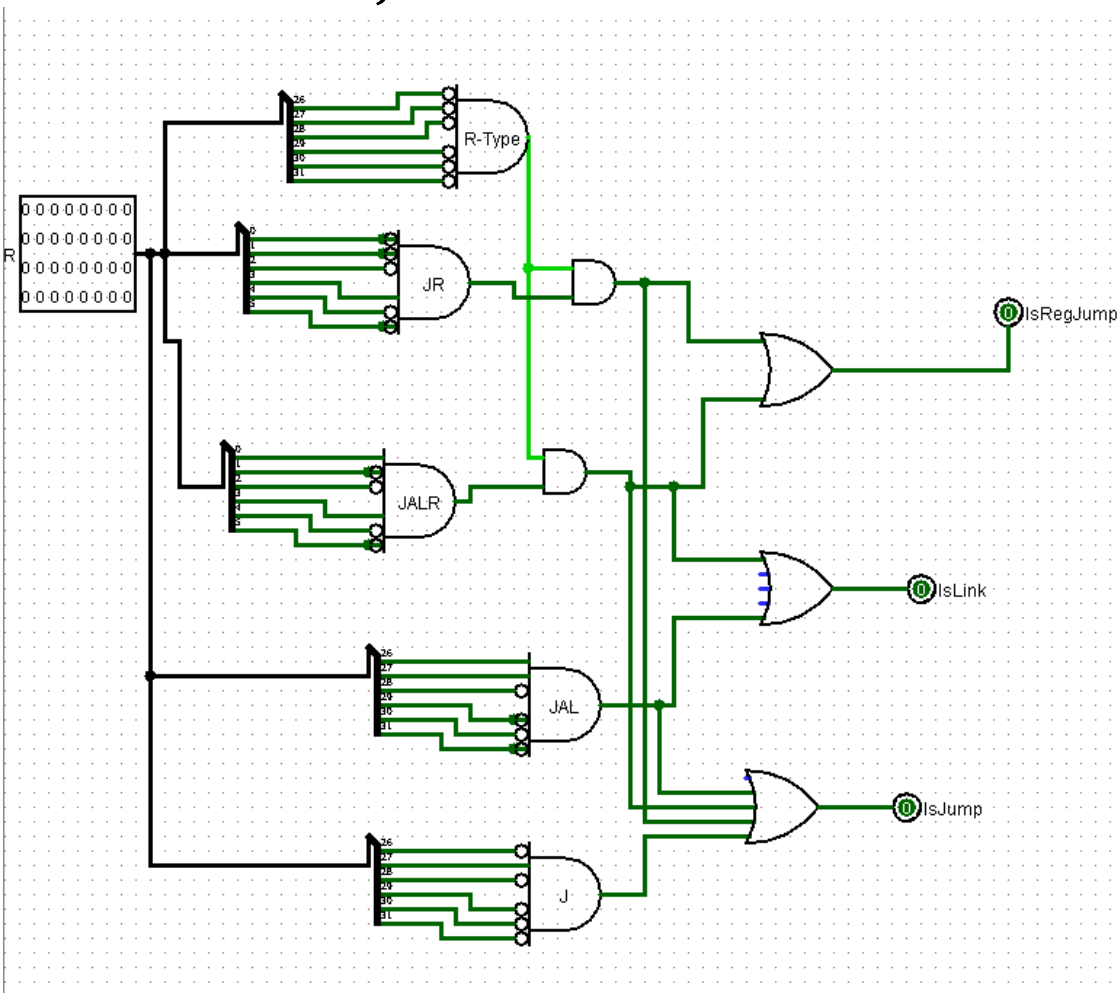
This MUXing logic then appends the proper value (beit absolute address, offset, variable amnt offset, stall) depending on whichever bits correlate to the logic in the various Jump/Branch Decoders.



Some instructions get a more specific MUXing later on. Mainly the JAL and JALR instructions due to them having to take the current PC address and storing it in the address of register 31. Because of this deviation we had to tap into an edit a specific region of the Execute stage, but this does little to how we interpret data and deal with Hazards.

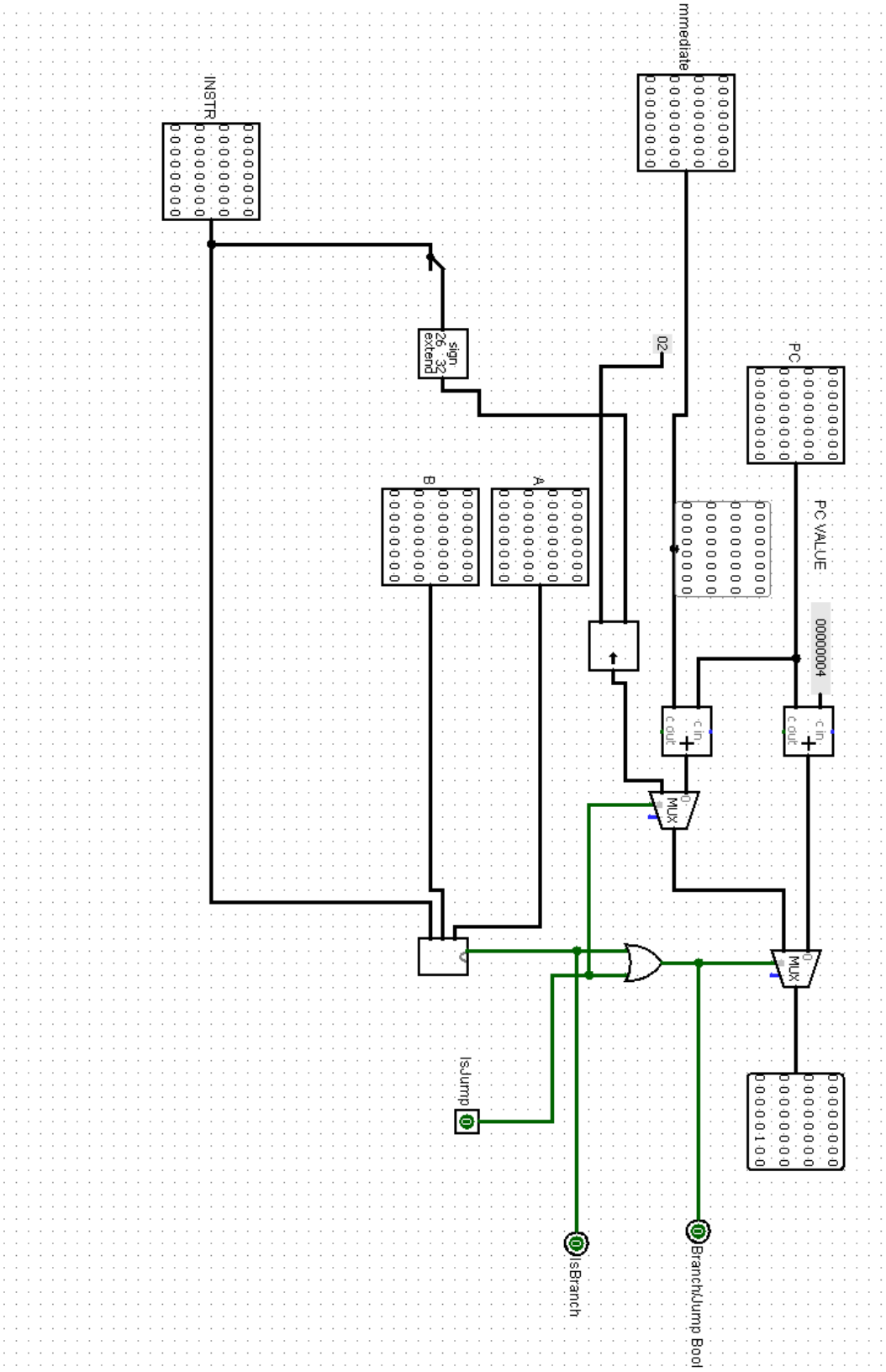
The real big components that come up are now the JUMP and the BRANCH LOGIC decoders.

JUMP

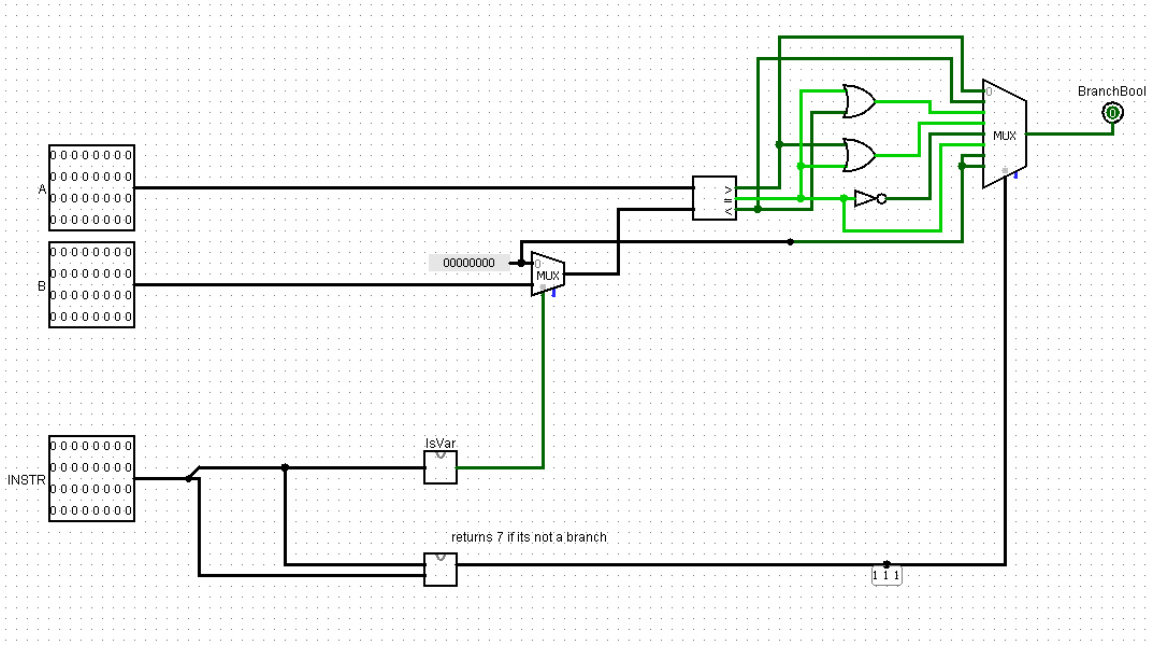


For JUMP Decode (Above) we have a subcircuit that feeds in three bits to other major subcircuits. This yields better deciphering in terms of MUXing and also allows us to differentiate between what is a JUMP and what is a BRANCH. This information is good for when trying to determine what we need to do for STALLing. The point is, the JUMP decode tells us whether or not it's a jump and what type of jump category it fits in. a Register type Jump, or a Link type jump, or both. From these bits we can have more specialized logic on the outside.

BRANCH


immediate
INSTR
sign
26 32
extend
02
PC
PC VALUE
B
A
0000004
c_in
c_out
MUX
IsJump
IsBranch
BranchJump Bool

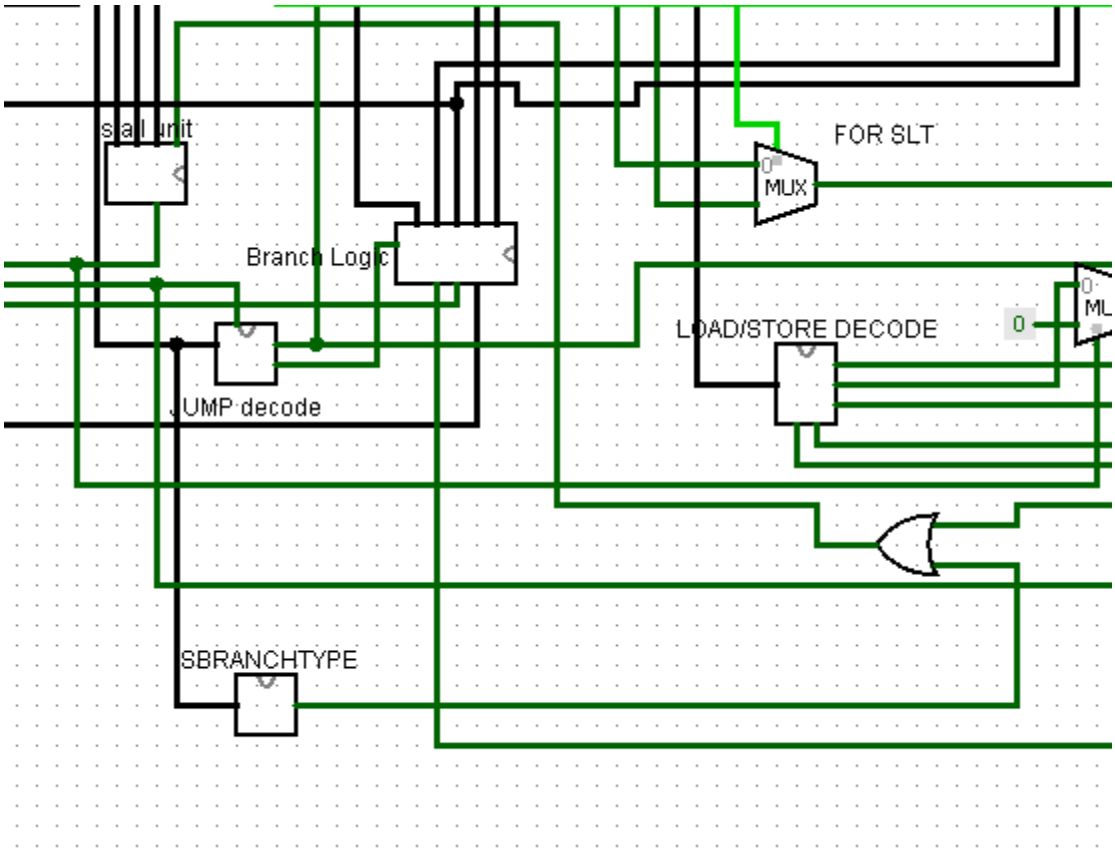
Above is the BRANCH LOGIC. This is the terminus that all jump or branch type instructions have to flow through in order for any sort of muxing or data to get jumped to. By deciding between whether or not it's a Jump or a Branch it exchanges the PC value with either an offset or an Absolute address (provided by a MUX). Branch Logic is also where the immediate value for the JUMP or Branch gets extracted and turned into something that can be used by the PC. It also takes in Values of A and B to further analyze and decide whether the Branch logic it may have to run is true. That subcircuit, located next to the "IsJump" input bit is known as BRANCH HARDWARE and processes Branch specific operations.



BRANCH HARDWARE as showcased above, is primarily used to determine whether the decoded Branch logic is true. If it's true it outputs a value "BRANCHBOOL" which then runs through the BRANCH LOGIC (larger) subcircuit. It is from there that we get whether or not we can operate on the Branch Jump argument. If the BRANCHBOOL returns 0 on a BRANCH instruction type, then nothing happens for it won't MUX or interpret a value. That being said, there is no way for the BRANCH to interpret a non-branch type instruction to make the BRANCHBOOL output true. The two subcircuits IsVar and the SUBCIRCUIT below that "BRANCHTYPE" will ensure that it follows the exact instruction type and formation.

After MUXing through the 5 types of BRANCH we know about, it then sees whether or not the comparison outputs TRUE. If it is, this value then waits to get MUX'd to alter the PC.

BRANCH & STALL



For BRANCH type instructions, it makes sense that we may run into data hazards that occur due to dependencies (a situation much like LOAD/STORE) and from there we deal with it in the same light. Notice the OR gate running through to the STALL UNIT subcircuit. This means that if the circuit detects that the OP code running through is of BRANCHTYPE (regardless of whether or not the BRANCH statement is true) and it faces a dependency, it will stall the Pipeline until the dependency is no longer an issue.

For that, we take a simple look at the ISBRANCHTYPE subcircuit. This logic determines if the INSTRUCTION would process like a BRANCH and if so, it will then pass through and go through the proper stalling procedure. (For information on proper Stalling procedure, reference the LOAD/STORE information section)