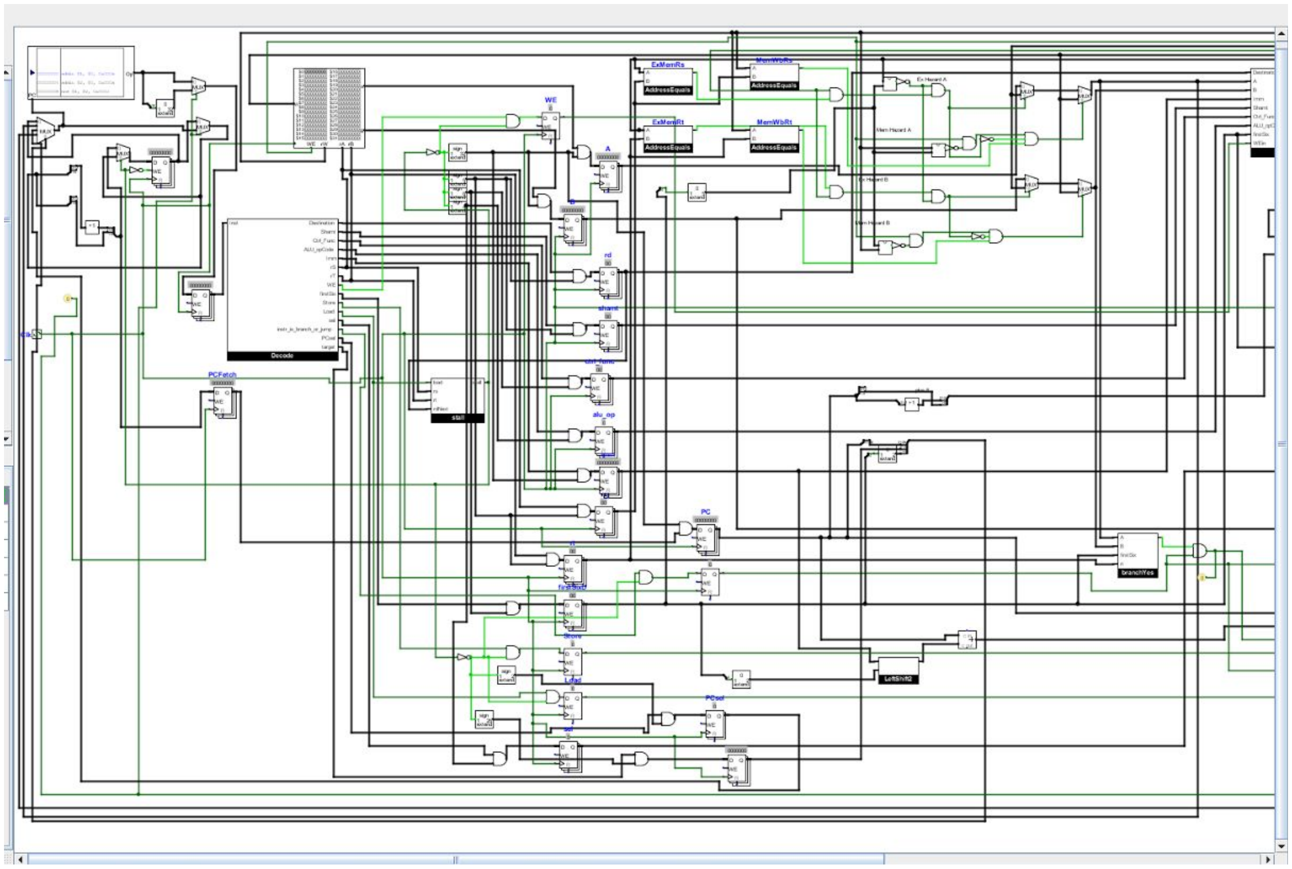
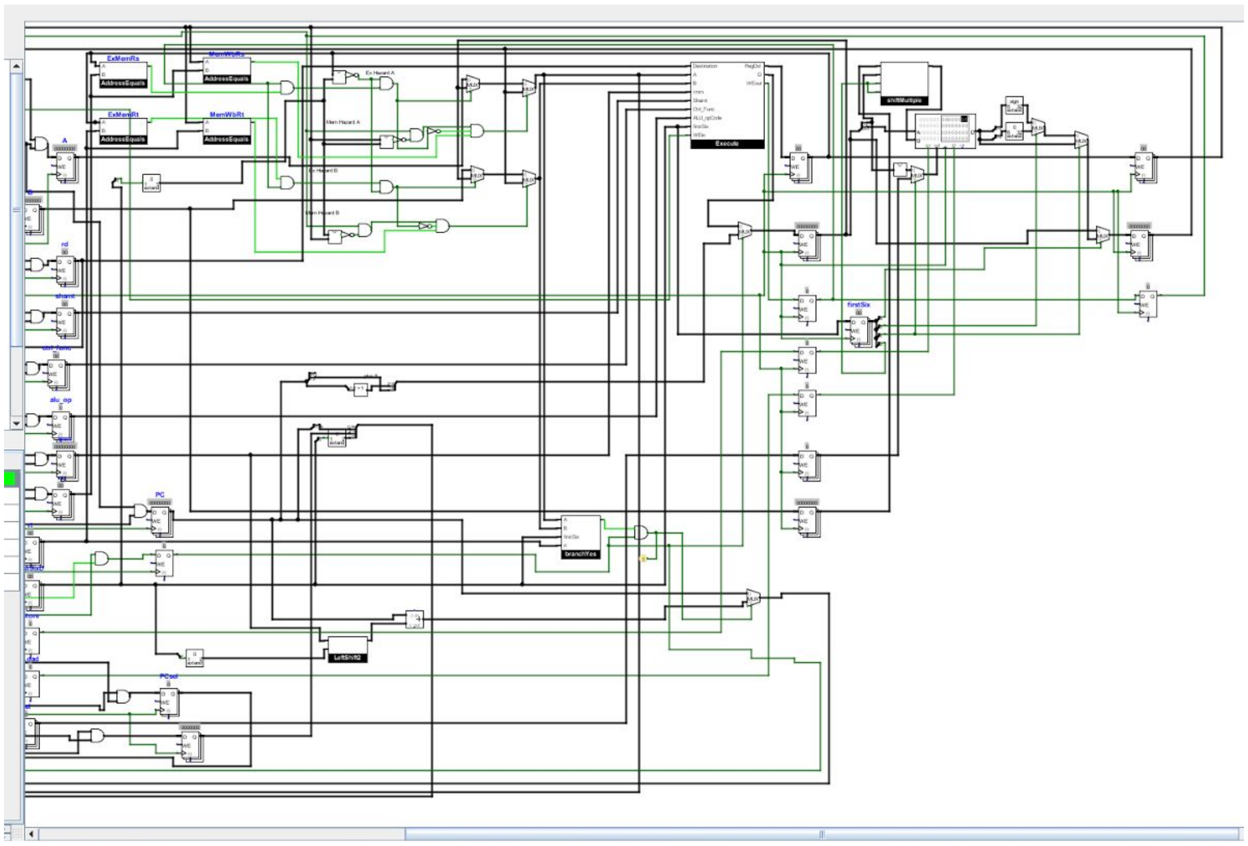
Justin Tran
Kevin Li
Project 4 Design Doc

1 Introduction

This is our design document for the Pipelined Mini-MIPS circuit in Logisim. We will go into each of the stages, talking about some of the wiring and providing descriptions of the control and instruction decoding logic.

2 Overview





Our MIPS processor has five stages. These stages will be described in section 3 in-depth. The first thing that we have to do is fetch the instruction from the general purpose register file; the instruction is then sent onwards and the PC (program counter) is then incremented by 4 with an adder (taken out of the ALU) for branches and jumps. We then decode the instruction, which involves splitting the bits in order to send part to the ALU and parts to the register file. These particular specifications will be discussed later.

After the decoding stage, the data passes through the ALU. Before it does this, there are MUXs which decide whether the ALU gets data from the pipeline or from the forwarded result of the previous instructions' result from the ALU, or from the writeback forwarding. Once the MUXs decide what data to take, the ALU executes the function and the data passes into the next register to wait for the clock cycle.

In the memory stage, sometimes the data just passes through, but other times, it must work with the data memory in the case of a memory-access instruction (load and store instructions). Again, this data goes into the pipeline register after finishing the stage.

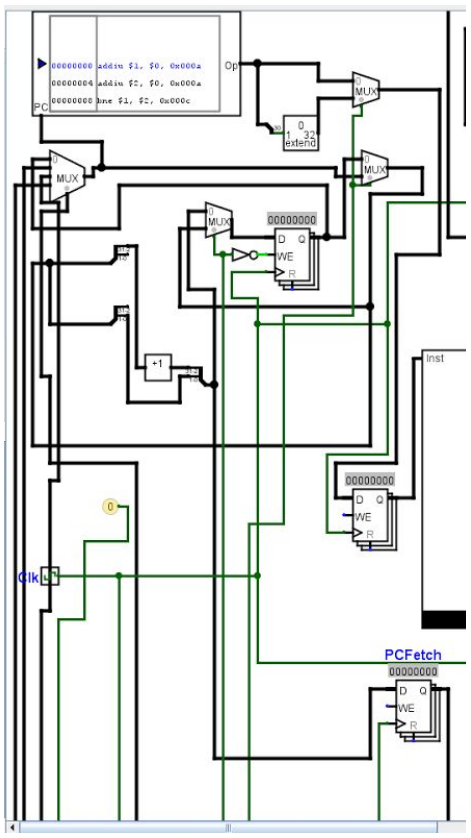
Finally, in the writeback stage, data goes from the last stage into the register file into the intended destination as defined by the initial instruction. Some data may also be forwarded.

In addition to what we had in project 3, we implemented our full memory stage, as well as all table B instructions. The ways in which we did this will be discussed later.

3 Stages

This section describes the different stages in the pipelined processor.

3.1 Fetch

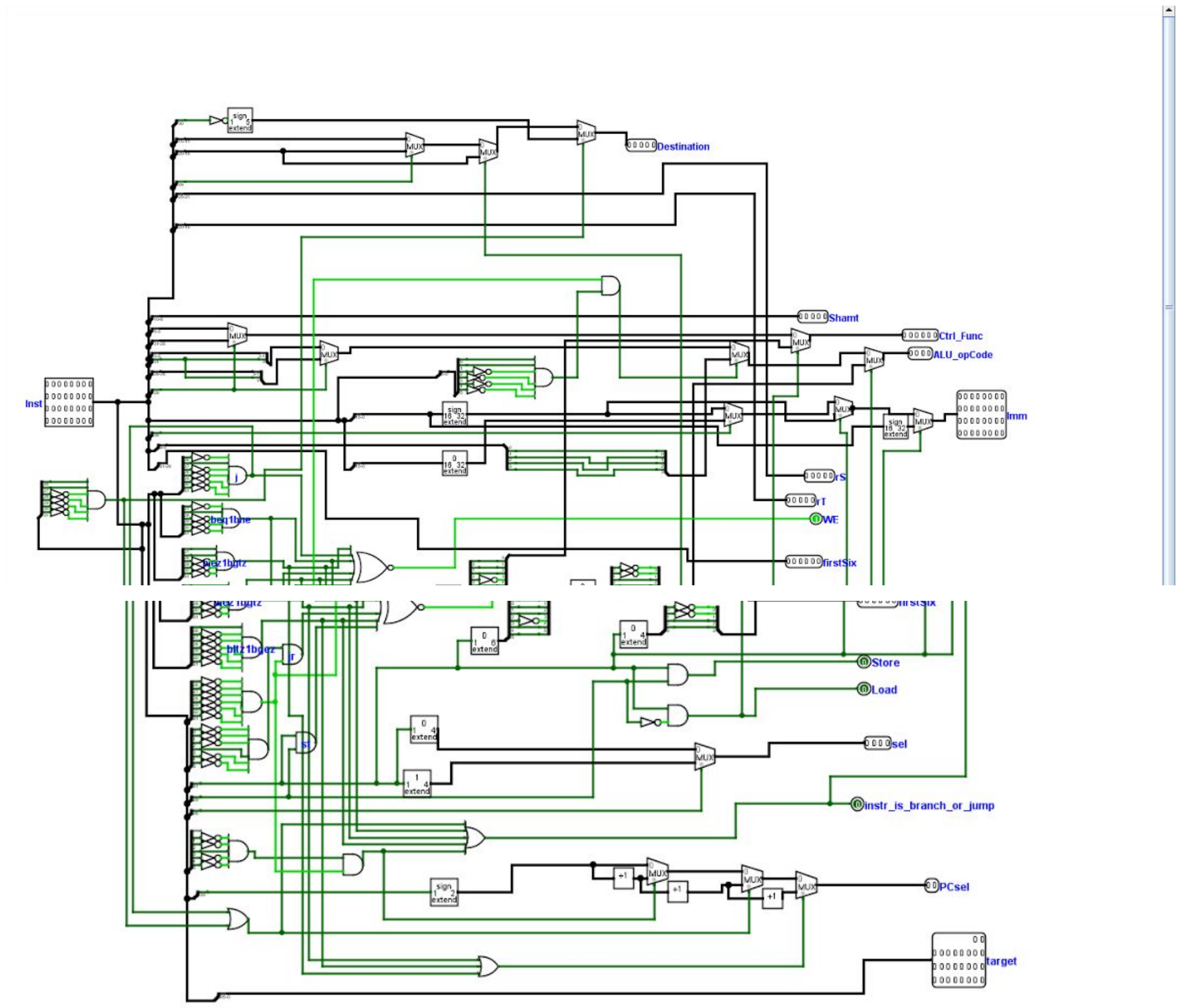


```
00000000 addiu $1, $0, 0x000a
00000004 addiu $2, $0, 0x000a
00000008 bne $1, $2, 0x000c
```

Fetch changed a lot from project three. The main changes had to do with the PC. The 4-input MUX decides what to do if there was a jump or branch calculated - as in the slides, there is the "regular" incremented PC, the PCReg (if the PC is data from a register), PCAbs (for the target), and PCRel (for branches). The logic for the control signal will be discussed in the Decode section. Additionally, for stalls, we used more logic from decode, which is also negated and put into the PC register to prevent incrementing if we need to halt the processes. The highest MUX is controlled by a bit that figured out if the instruction was a jump or branch (using logic from project 3). If it is, we NOP-ed the instruction. The MUX directly below that one makes sure that if there was a jump or branch calculated, the PC does not pick up from where it left off - also using the jump or branch detection.

3.2 Decode

3.2.1 Decode subcircuit



The Decode stage is the "brains" of our circuit. In this stage, we have to determine what type of general instruction we have (R-type, I-type, Jump/Branch, or load/store) and then look at specific parts of the 32 bits in order to send it onto the next pipeline register.

There are 9 outputs to our subcircuit, and one input. The input is simply the instruction that we got from the MIPS ROM, which we are passing to the subcircuit. From this, we interpret the 32 bits to figure out where to go next. The first output, destination, is simply a choice of two different "instruction slots" - one is bits 16 through 20 (for I-types) and the other is bits 11 through 15 (for R-types). The 29th bit is responsible for choosing between those two, as it is 1 in all I-type instructions and 0 in all R-type instructions.

However, in project 4, we added another possibility - 11111, for register 31 in the case of a JAL instruction. We made this by taking bit 30 (which is always 0), negating it, and extended it by five. Using other logic from project 4 (determining if the instruction is JAL), we MUX it with the other destination wire to get the correct destination register.

Next is the shift amount. This is not used in all instructions, but it is consistently in bits 6 through 10 for all shift instructions. If it is not used, it will simply do nothing in execute, but to cut down on MUX numbers, we included it in Decode.

The control function is either the first six bits or the last six, depending on what type of instruction it is. For R-type we pass the first six bits, while for I-type we pass the last six. This is, again, decided by bit 29. We also pass the first six bits through regardless of what type of instruction it is, to help make execute easier.

The ALU_opCode is used in the execute stage. For *most* instructions, we can take the last three bits of the control bits and tack a 0 onto the end. However, for right shift arithmetic, this would be 0110, while the opcode is 0101. Thus, we need a specific check for that control signal and another MUX to figure out if we need this specialized, specific opcode.

The Imm field is either sign-extended or bit extended, and is chosen through MUXs with bit 28, as shown in the MIPS cheat sheet.

rS and rT are simply split off from the main 32 bits and passed down, since they are always in the same place. Sometimes they are not used, but it is easier to just pass them and not use them later rather than use MUXs to decide.

WE (the write enable signal) is only set to 1 if the instruction is not a branch or jump, but it is still one for JALR and JAL - the AND gates figure out if the instruction is one of those types, and if it is any of them (or if the 31st bit is 1), WE is set to 0 so that those instructions do not affect our main register.

The immediate has an additional control signal to figure out if we need a sign-extended offset for our branches; the control signal comes from the detection if the instruction is a jump or branch.

Store and load bits are simple; it comes from the 31st bit (which is always a 1 for a load or store) and an AND with the 29th bit, which is 0 for loads and 1 for stores.

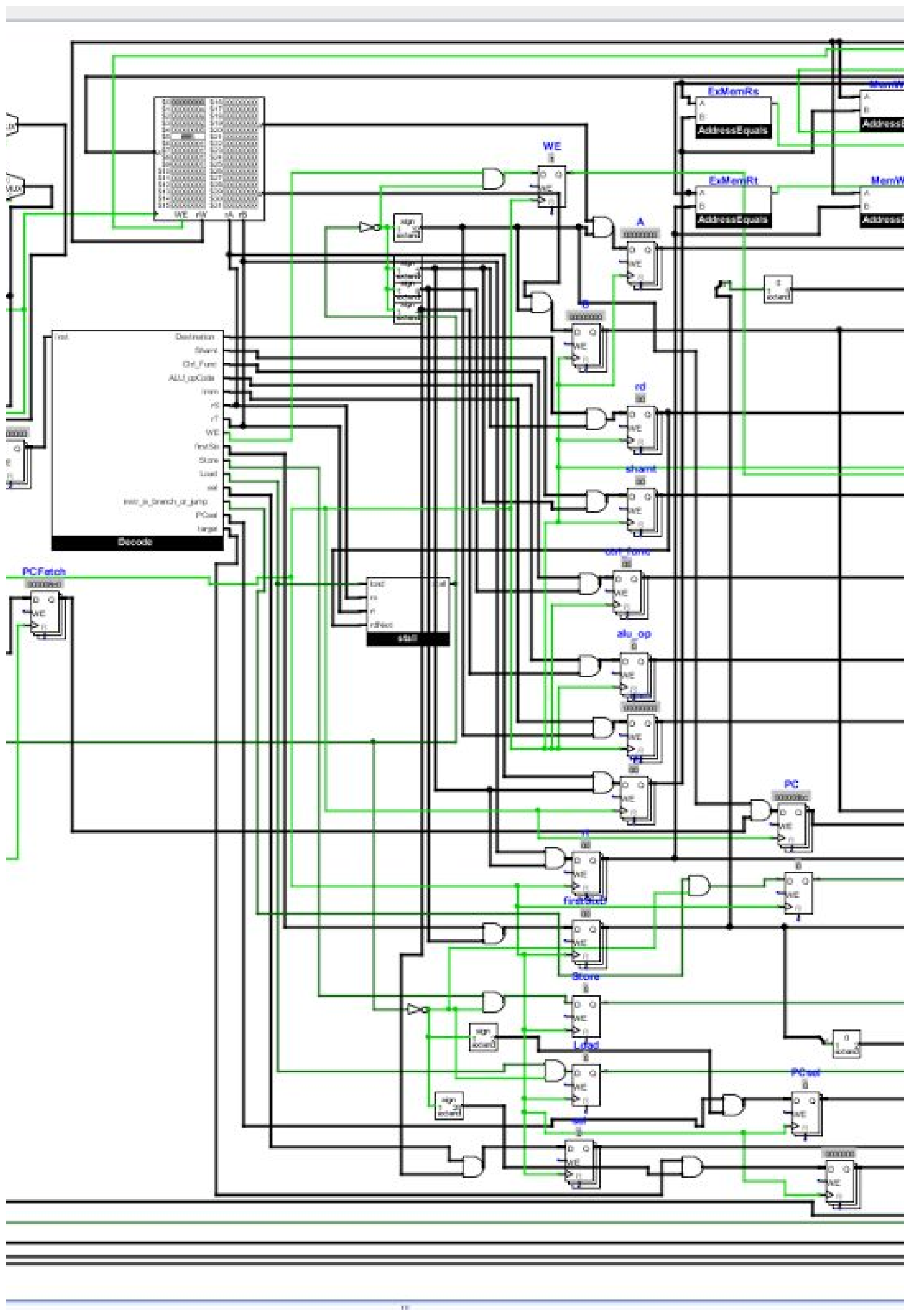
The sel field is all 1s for a word instruction, and is simply passed through to be calculated later for a byte instruction (specifically, it will be calculated in memory).

Instr_is_branch_or_jump is fairly obvious. It is the logic that disabled WE from project three (here, since some jumps need WE, we had to changed that logic).

PCsel is the control signal for the 4-input MUX in Fetch. As there are four possibilities (00, 01, 10, 11) we used incrementers and MUXs. For example, if PCsel needs to be 01 for a PCreg branch, we use the signal that originally determined if an instruction was JR or JALR and used that as the control. This was done for all four possibilities (including the possibility that nothing happens and the PC should increment normally).

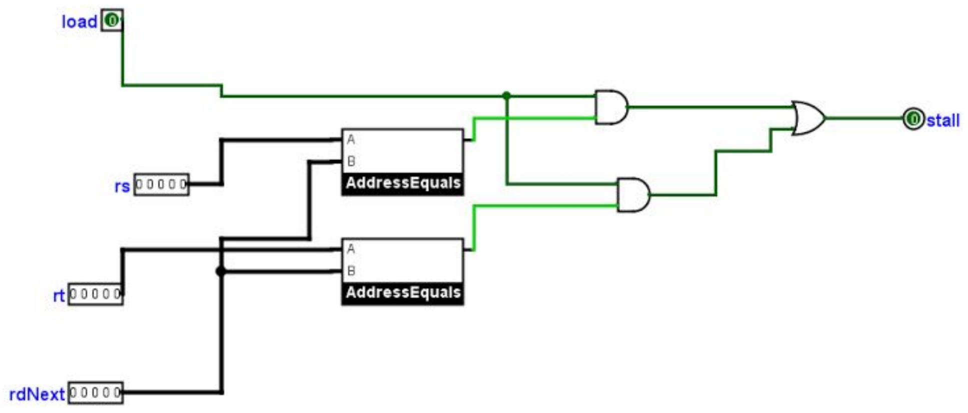
Target is simply the lower 26 bits, for J and JAL.

3.2.2 Outer circuit



This part of the circuit looks messy, but is almost identical to the project 3 circuit with the exception of the stall subcircuit. The added outputs are put into new registers. The stall circuit (shown later) figures out if there is a hazard from loads and stores and outputs a 1 if so. This signal is then negated, bit-extended and bitwise AND-ed with each signal so that if there *is* a stall, the instruction will not pass through. That way there is a bubble in the pipeline (the signal is also passed back to fetch, as shown earlier).

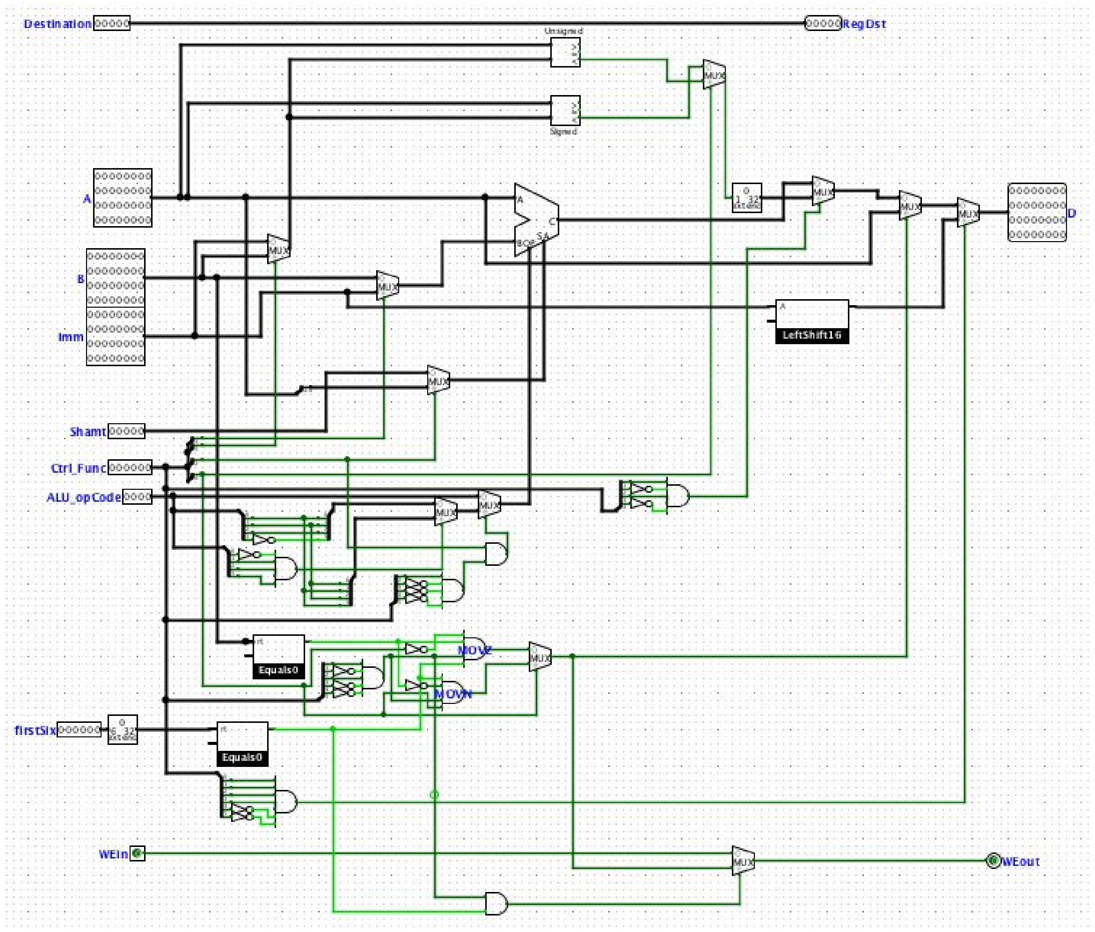
3.2.3 Stall



This subcircuit simply uses the load signal from the Decode subcircuit, the rs and rt outputs, and a forwarded value from execute (the destination register). We used the AddressEquals subcircuit from execute. If rs equals that destination or rt equals the destination, we must stall.

3.3 Execute

3.3.1 Execute subcircuit



The inputs from this circuit all come from the previous pipeline register. This subcircuit handles all of the execute commands (the forwarding is done outside of this subcircuit, and will be described later). Thus, we can assume that the inputs are all "correct."

The inputs to the ALU are A, and then either B or Imm depending on bit 3 of the control signal (which signifies an immediate arithmetic/logic command. The shift amount is either Shamt or the lower five bits of A for a variable shift, determined by bit 2 of the control.

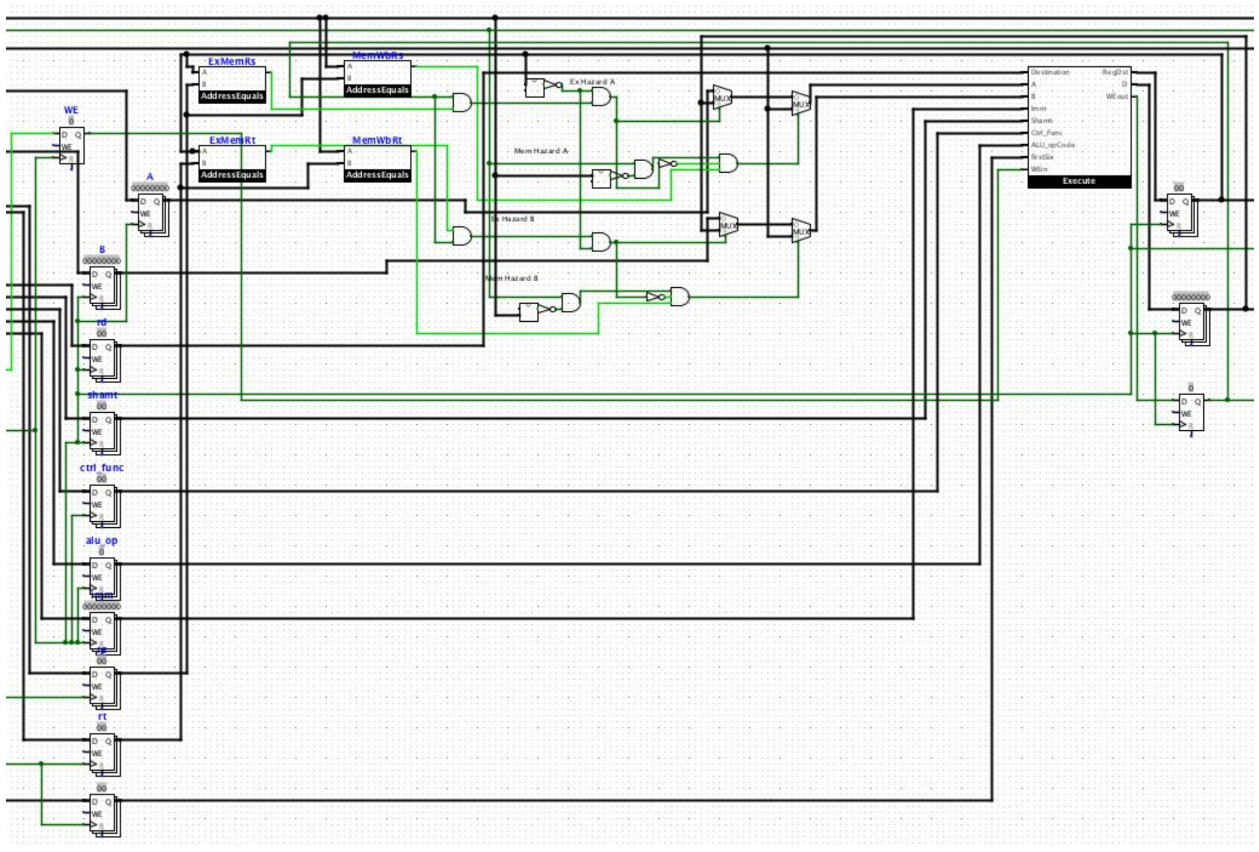
The two comparators at the top have to do with slt, slti and other "set if less than" commands. These are simply compared using MUXs (bit 0 to pick between signed and unsigned). This is compared with the ALU output, using an AND from the control signal - if the middle four bits are for one of the "set" commands, then we make the choice between the comparator output and the ALU output (there is no one bit to do this, so we have to use four. If it was just R-types, we could use bit 3, but bit 3 is 1 for ALL I-types so it would not work for the overall processor).

The next MUX deals with MOVN and MOVZ. These are done lower in the circuit, and we use our "equals0" subcircuit. This is essentially the equals subcircuit from the previous project, but with only one input, so the other defaults to 0 and we can compare the input to 0. If it is 0, and the general MOV code is applied, and the first six bits equal 0 (if not, it may be an I-type instead of an R-type MOV), then we can use the MUX. If the command ends up such that nothing should be done to the registers, we set write enable to 0.

LUI is done by shifting the immediate value up by 16 bits and checking the LUI control with the other main output in a MUX. All of this ensures the correct output (as proved in our MIPS commands text), assuming the correct inputs from forwarding.

This subcircuit did not change from project 3 to 4.

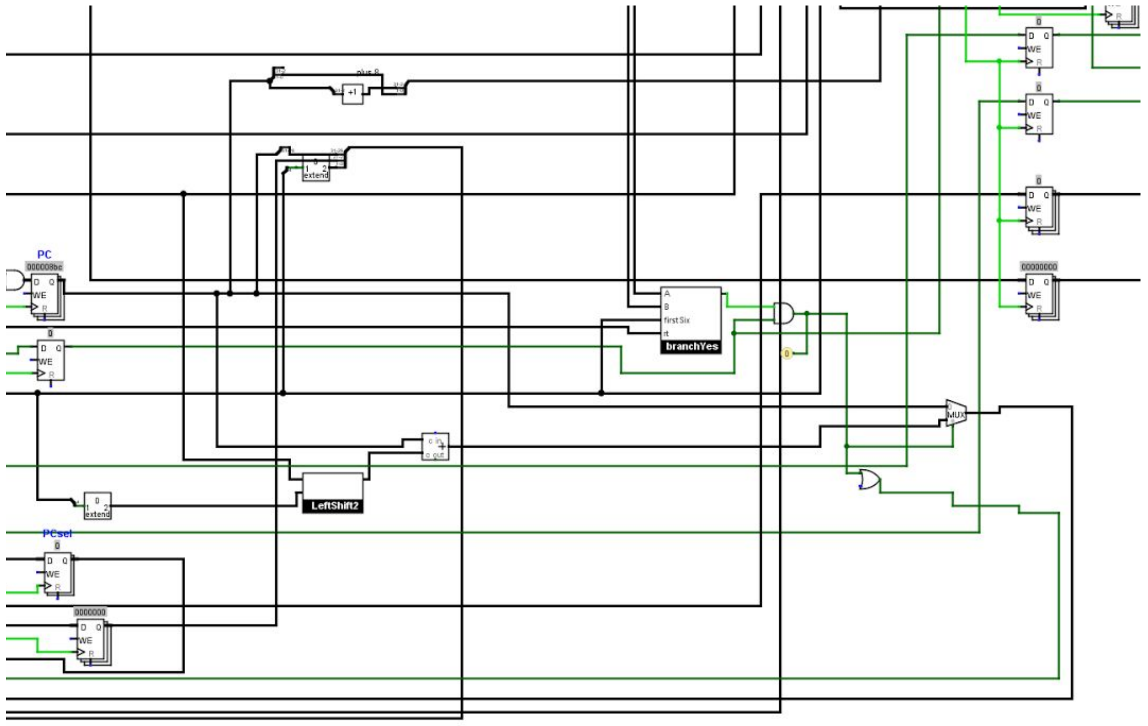
3.3.2 Execute main (forwarding logic)



Forwarding is done using the table from the project specs. We must compare registers rS and rT to the rD from memory and from writeback. If any are the same, there will be a data hazard. In this section, we will discuss the general forwarding case, as the forwarding for A and B are identical except that A uses rS and B uses rT.

If the EX/MEM rD matches rS, we also see if WE from EX/MEM is 1. If it is not 1, there is no need to forward as the register will not be written to and there will be no effect on our current execute command. These comparisons are done in identical equals subcircuits, which are 5-bit versions of an equals subcircuit. If rD is NOT equal to 0, then all of these wires lead to forwarding and we must use EX/MEM's "D" wire instead of our current "A" wire. However, there can also be a MEM/WB hazard; in this case, there are similar comparisons. If rD matches rS, we must forward, but only if rD is not equal to 0, WE from MEM/WB is 1, and there is no EX/MEM hazard (as shown in the project specs). A 1 here results in the "D" wire from MEM/WB passing through to A instead of the original register from Decode.

3.3.3 Execute main (jumps and branches)



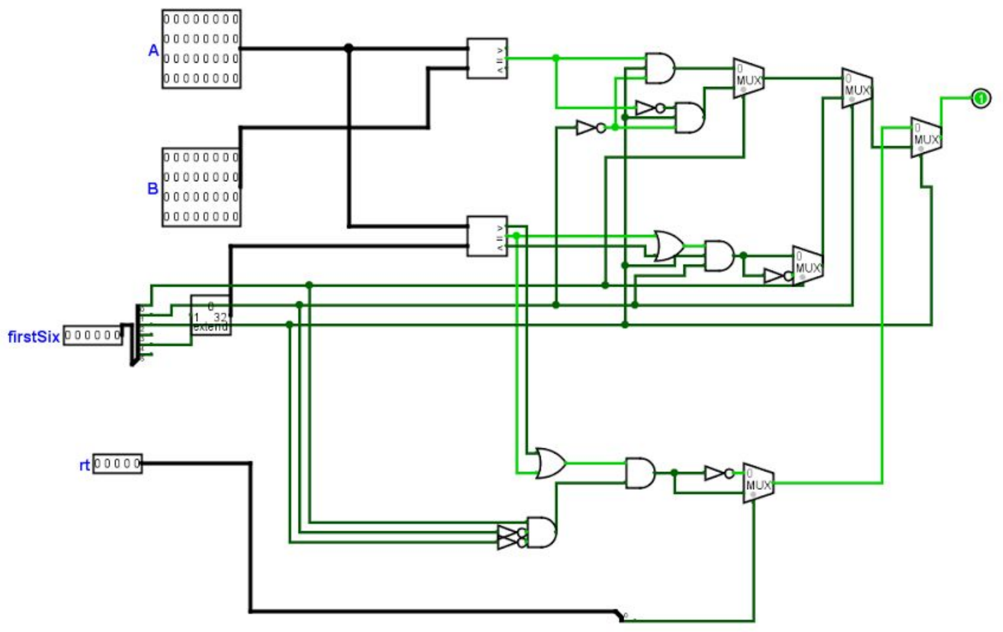
There are a few new parts here. Starting from the top:

The PC is incremented by 8 and then MUX-ed with the data from the ALU. In some jumps, we need to store the PC + 8 in register 31 or a register determined by the user, so if the command is one of those, we pass that value into the pipeline register instead of the data from the ALU.

Next, we calculate the PCabs using the target, the upper bits of the PC, and 00. This is passed back to fetch (and is fairly simple).

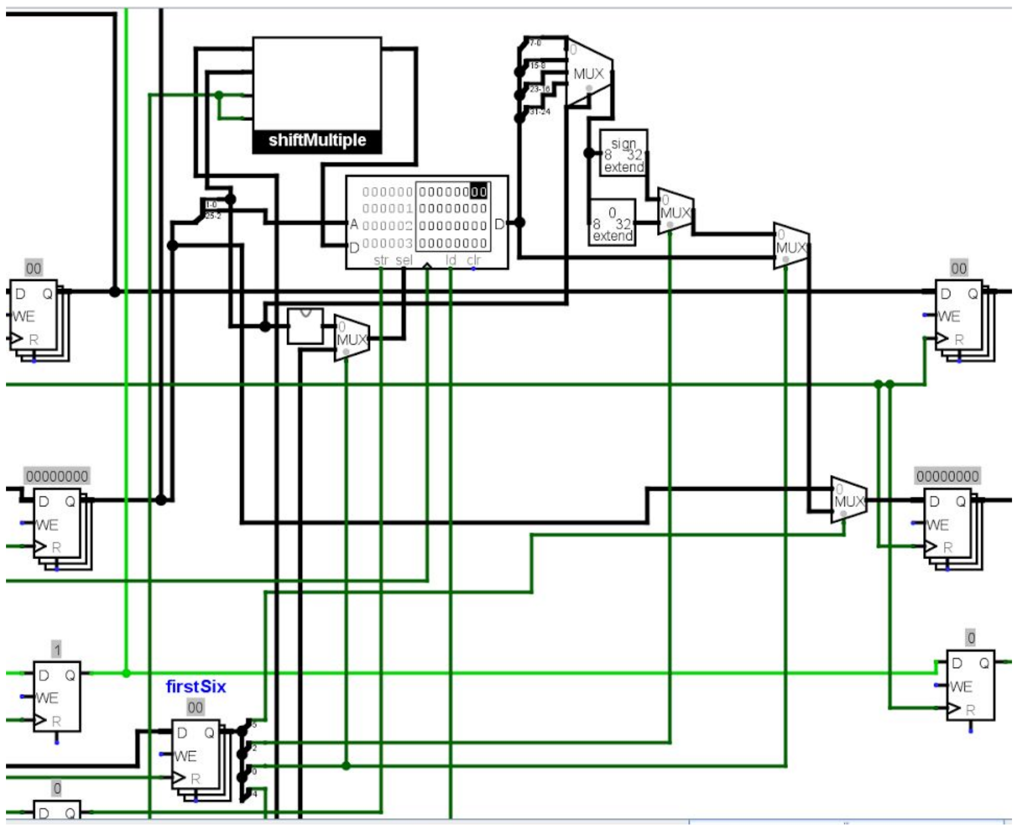
LeftShift2 is used from the old ALU project, as the immediate must be shifted by two and added to the PC for branches (PCrel). This is MUX-ed with the original PC with a signal that figured out if the branch was even taken, which was AND-ed with the signal that told us if the instruction was actually a jump or a branch. All of these are passed back to fetch to be MUX-ed with the control signal PCsel from Decode (which was passed into execute to keep it synchronized).

3.3.4 branchYes subcircuit



This takes two inputs (after figuring out if there was forwarding from above in Execute main), the upper six bits of the original instruction, and rt (for BLTZ and BGEZ). It uses two comparators, for the equal instructions (which are AND-ed with signals that prove that they are branch if equal/not equal), and for the less than or greater than 0 branches. The top comparator is negated for one of the AND gates, since we sometimes "want" is to be equal, or not equal depending on the signal (the 0th bit of the firstSix input determines if it is BEQ or BNE). Similar logic is used for the lower comparator with the remaining 4 branch instructions. In the end, if the branch is taken, the subcircuit outputs 1.

3.4 Memory

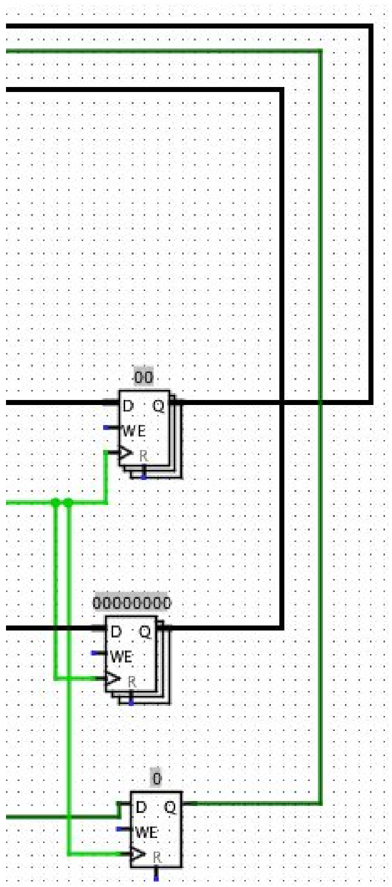


This circuit now does something for project 4. Parts of D - the calculated address - are sent into the A field. The lower two bits are split off, however, and decoded into a one-hot signal which goes into sel. The other signal is the original select field from Decode, which is 1111 if the command was for a word (bit 1 of firstSix is the control here).

shiftMultiple is an exact copy of leftShift from the original ALU project. However, it is only ever shifted by a multiple of 8 - if the instruction wants to write to byte 1 of a word, for example, in word addressed memory, we must shift is by 8 in order to write to the correct bit. Bit 4 of firstSix is sent in to effectively be a constant 0 (since it is always 0). str and ld are from decode, passed along registers to this stage.

Depending on where the output comes from - first byte, second bytes, third, etc - we must move it into the lower part of the output using splitters and the un-decoded control signal. Then, it is either sign extended or 0-extended. We then figure out if we wanted the full word or just a byte, and then pass that into the final D register.

3.5 Writeback



The write back stage passes the data from the Memory stage (or in this case execute) all the way back to the general purpose register. You also pass down the rW value, which tells the register where to store the data, and the write enable (WE) signal. Some of these signals are also forwarded to the execute stage (more details about the forwarding process is described in the execute section).

3.6 Pipeline Registers

The pipeline registers that are located in between each stage serve as ways to separate each clock cycle to perform only one part of the MIPS operation. Every time the clock is turned on after being off, the values are then passed onto the next stage. There is a single register for every output/input of each stage.

4 Testing

We wrote test vectors for Decode in order to make sure that we were splitting the 32-bits correctly. The testing for our main circuit came from the MIPS commands that we wrote. These commands were run through Danny Qiu's MIPS interpreter to ensure that they were correct (aside from jumps and branches, which are not supposed to do anything).

The MIPS commands test every command, including ones that are supposed to do nothing. They also test for the three types of hazards that we need to account for (EX/MEM, MEM/WB, and reading before writing in the register). All of the register values match those in the interpreter. Thus, our circuit should be correct in terms of its output values into the registers. For certain instructions, like slti and sltiu, we used binary immediates which would change the outcome if they were unsigned or signed so that we could ensure that the commands worked (for example, -6 in binary is a large number unsigned but a small number when signed, so it is larger than 5 unsigned but smaller than 5 signed).