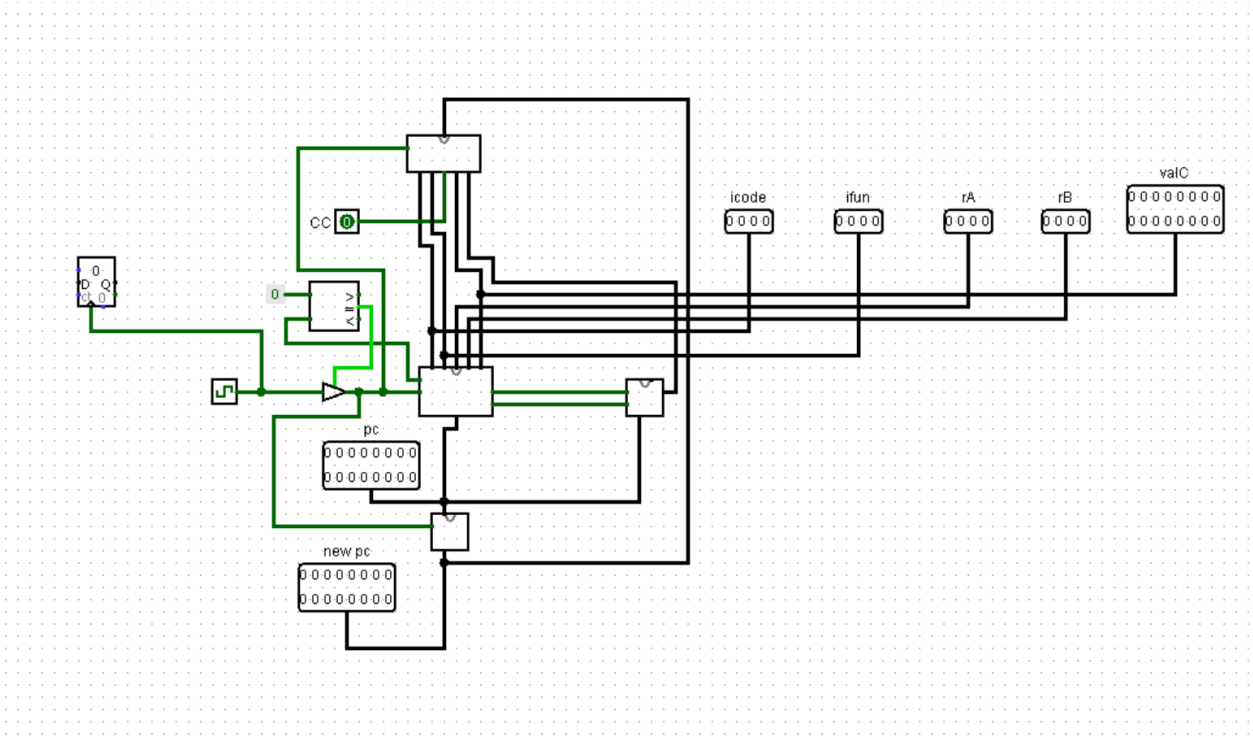
Kyle Diano

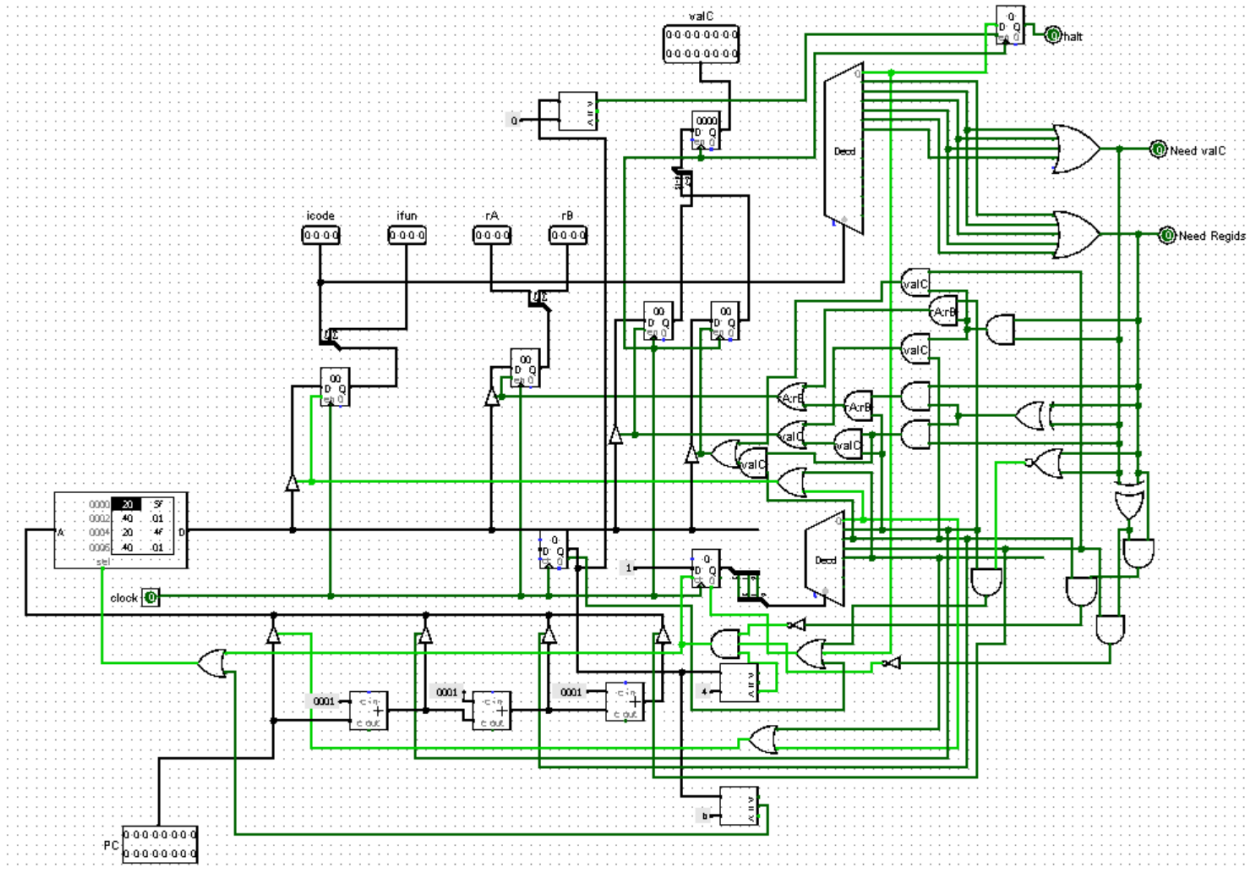
Group Members:
Trisha Narwekar
Patricio Bunt
Muneeb Hashmi
Nick Martinez

8-bit Load Store Processor Individual Report

My part in the project was to build the fetch, PC, PC increment, and to put the final circuit together and ensure that everything works by running tests and making adjustments where it was needed.

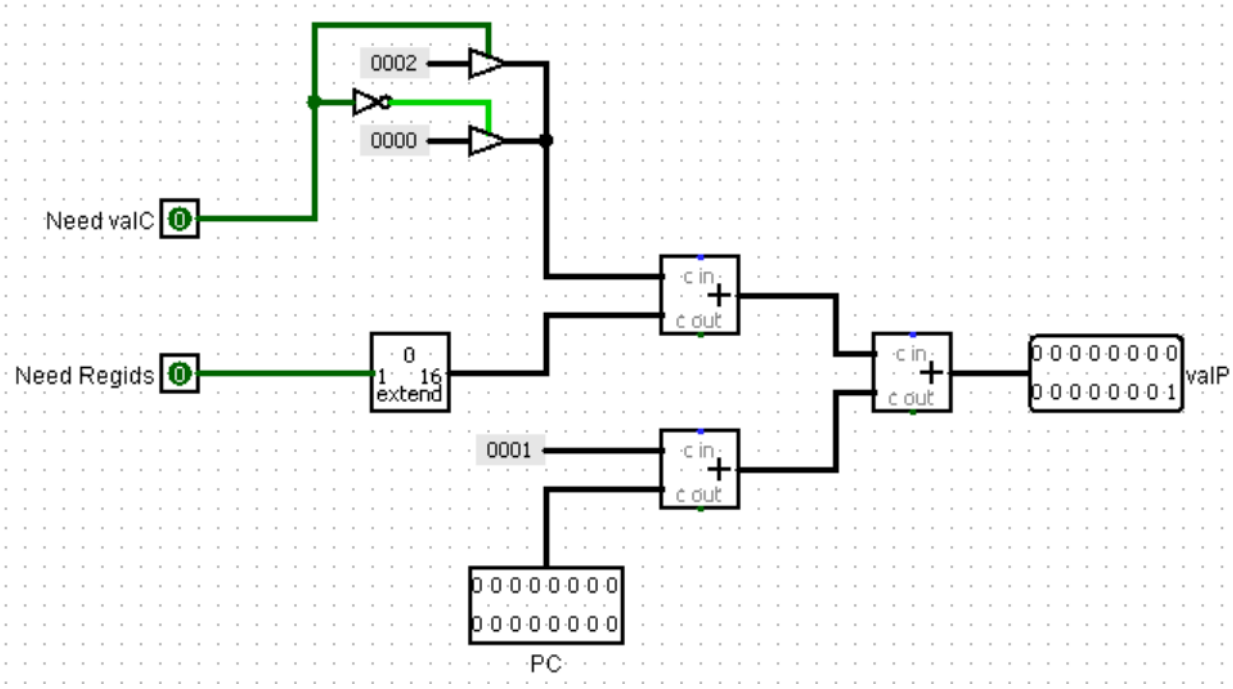


This is a general view of the fetch implemented with PC, PC increment, and New PC. This shows how the clock is going to work for the overall processor. Once halt is reached in the program instructions, the output of the clock for the whole processor is shut off.

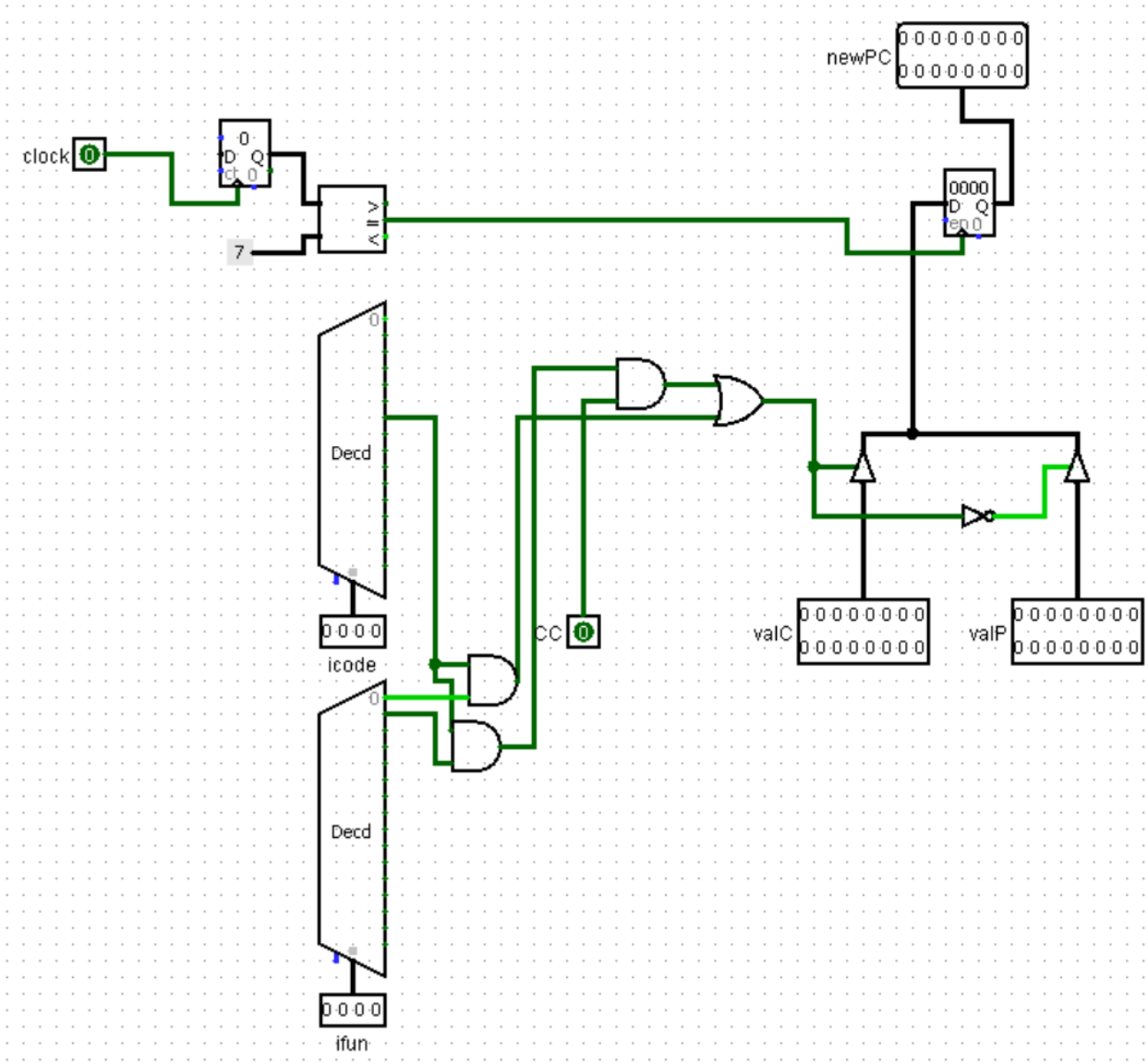


This is the complete circuit of the fetch in our processor. Since fetch is going to always be the first thing to run, it begins on the first clock cycle. The way I decided to implement fetch was to utilize a counter and a decoder to allow me to use specific cycles to activate different things. After the first cycle, icode and ifun are loaded, which allows the processor to determine whether we need registers or valc. This then outputs to PC increment, but also determines how the circuit deals with the clock cycles for that specific instruction. For example, if the instruction is something like “irmov”, the fetch needs to get icode:ifun, ra:rb, and valc, which would take a total of 5 cycles. If it was an instruction that just needed icode:ifun, and ra:rb such as “rrmov”, then it would only grab those two things from the rom in 2 cycles. If it was an instruction that just needed icode:ifun and valc, such as “jmp”, it would only grab those two things from the rom in 4 cycles. If it was an instruction that just needed icode:ifun, such as “halt” it would only get that from the rom. After accounting for these specific cases, the decoder combined with this logic

determines which controlled buffer to activate of the 4 calculated values from the PC. There's the PC value, the PC + 1 value, the PC + 2 value, and the PC + 3 value. It is organized this way since the rom stores all the program instructions 1 byte at a time, and the max amount of bytes that one instruction would take would be 4 bytes. Therefore, using this logic allows the fetch to access the correct indexes for what it needs depending on the instruction. From these indexes, the data for a specific value is being run into separate 8-bit registers. Using the decoder and logic decides what controlled buffer is activated, which would then store it in a register. For icode:ifun and ra:rb, these registers split up from 8-bit into 4-bits since these four values are four-bit. For valC, it actually takes two 8-bit registers that would combine into a 16-bit register, as this is the max value we have for our registers in decode. It is important to note that the bits 0-7 are taken first, followed by 8-15 when loading valC. Finally, halt is implemented by checking after grabbing icode if the instruction is halt, which then outputs out of fetch into the main clock, and cuts off the clock's output to the whole circuit. In addition, at the last cycle of an instruction or cycle 11, the value at the next PC is being loaded from ROM, in order to correctly load it into the icode:ifun register on the next first cycle.

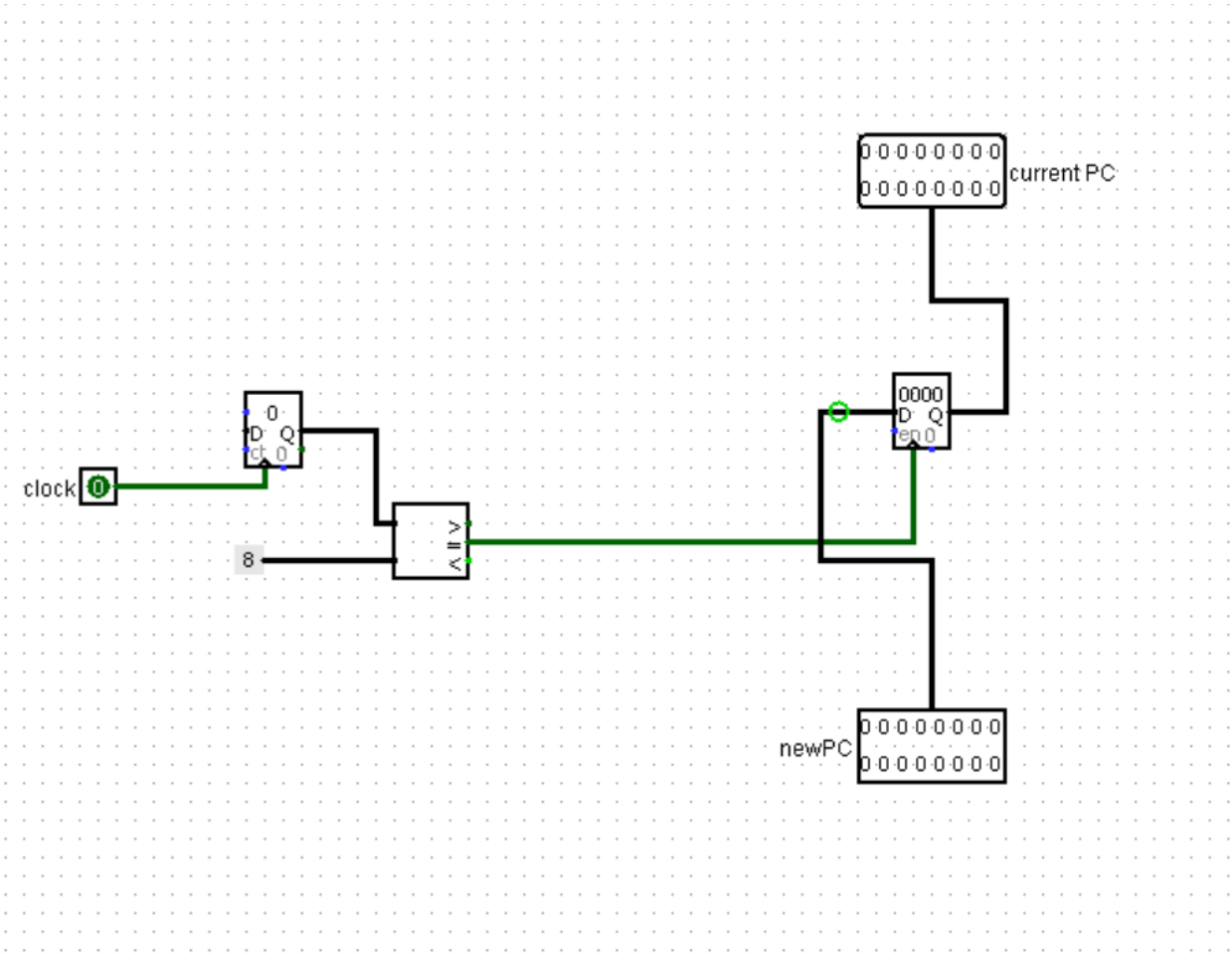


This is the PC increment circuit. It takes in the “need valC” and “need regID flag” from fetch and calculates valP. The equation for our circuit to do this is valP = 1 + r + 2*v, where r is 0 or 1 depending on if we need registers, and v is 0 or 1 depending on if we need valC. Since this comes directly from icode, valP will be calculated on the first cycle.

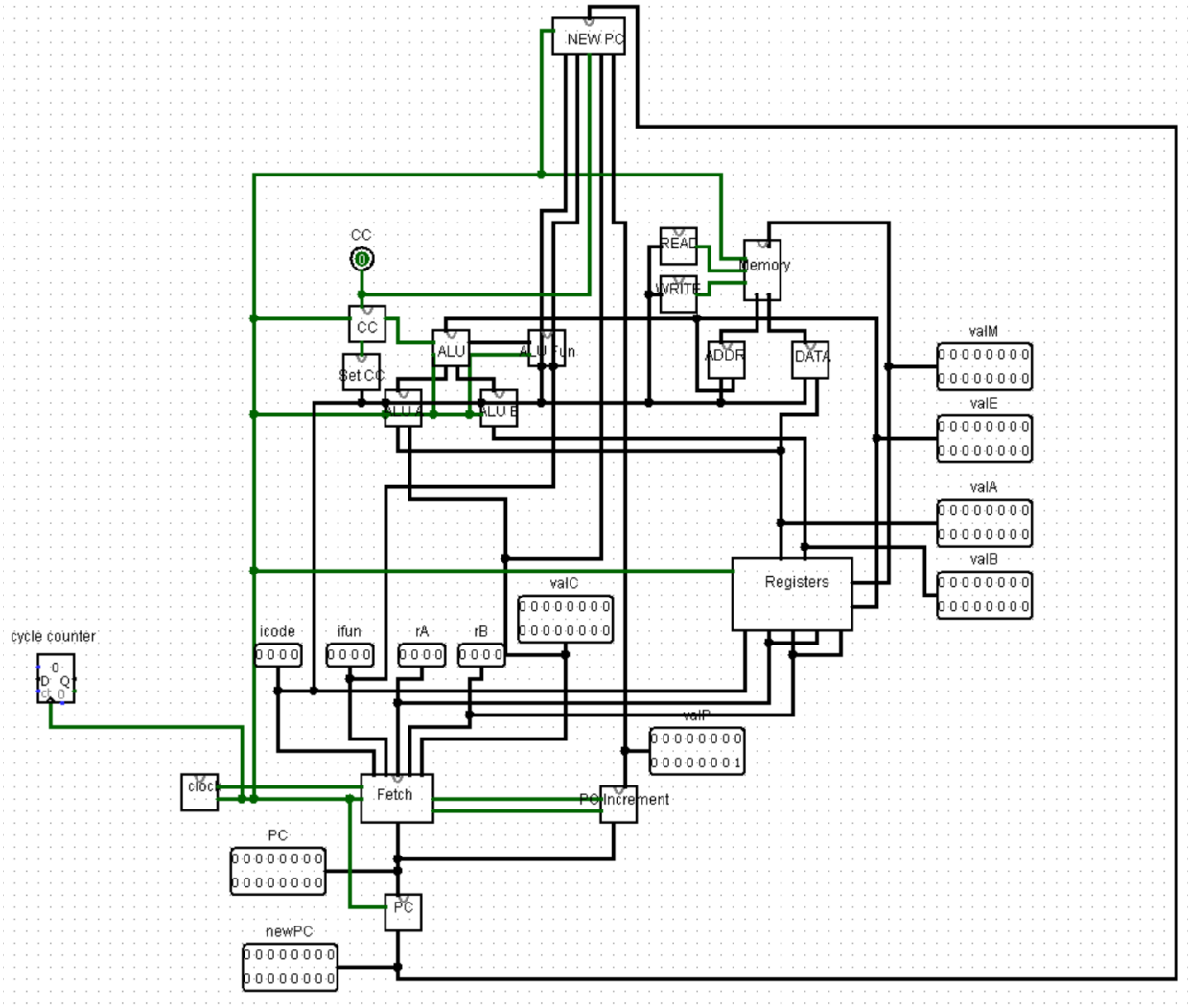


This is the new PC circuit that I built with help from fellow groupmate Nicholas Martinez. It takes in the value of valC, valP, ifun, icode, and the condition flag. The newPC is always going to be valP, unless the instruction is jmp or je. When it is jmp, it allows valC to pass through.

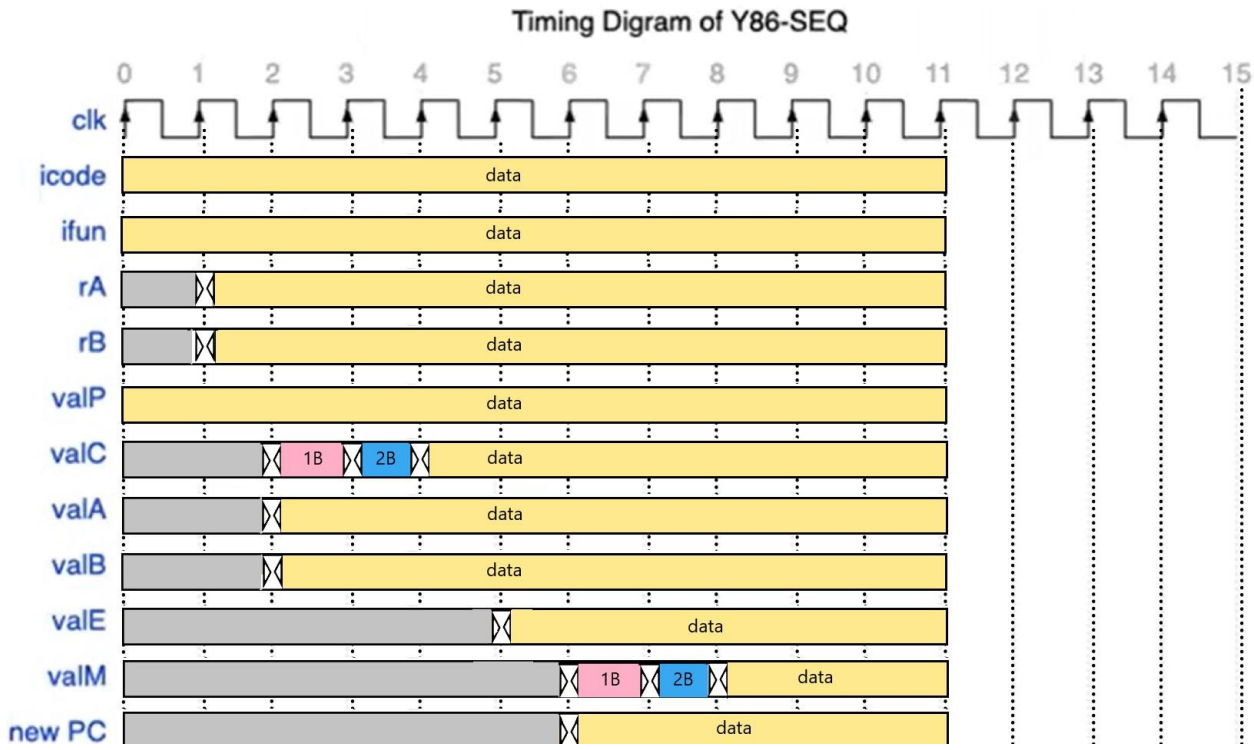
When it is je, it first checks the condition flag, if the condition flag is not on, then it will not jump, so it will use valP. Otherwise, it uses valC.



This is the PC circuit. It takes in the value of newPC from the newPC circuit and it updates the current PC on cycle 8, so that fetch knows where to move next in the program instructions.



This is the complete 8-bit load store processor unit. It includes fetch, decode, execute, memory, PC, PC increment, new PC, and writeback. In order to implement all of these together, we had to make sure that each part was on the same page with clock cycles. Therefore, each part includes a counter that goes up to 11, which is the amount of cycles for each instruction, besides halt, which would stop the entire processor.



This timing diagram that I made shows when each value is outputted, which was key into making sure that the whole system integrated together.

For this assignment, there were several advantages and disadvantages by working with a team. The advantages include the fact that there was less overall work for the individual and that it helps practicing working with a team. The disadvantages include having to coordinate times to work together and to maintain being on the same page about certain things. Specifically for this project, it would be very easy to fall off track with how the whole circuit is supposed to work, which makes it extremely challenging when putting everyone's individual parts together.

In my opinion, I think that Trisha's design was the most superior, as it was the part that took the least adjustments when putting it together with the whole processor. It is also the most clear and structured part out of all the other integrated circuits. I suppose it could be improved by having even cleaner organization.