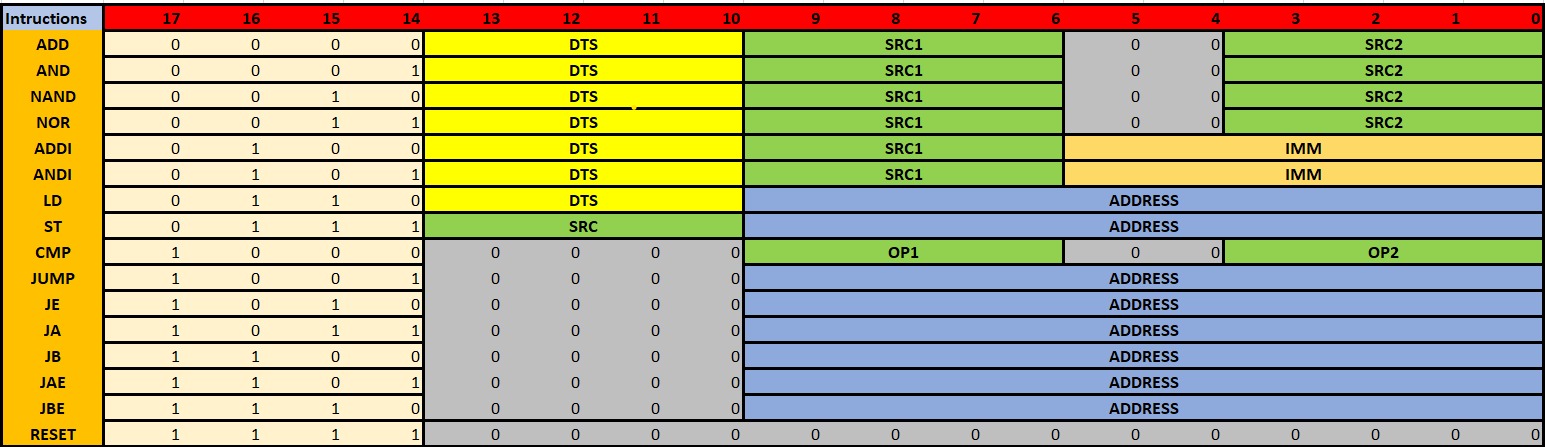
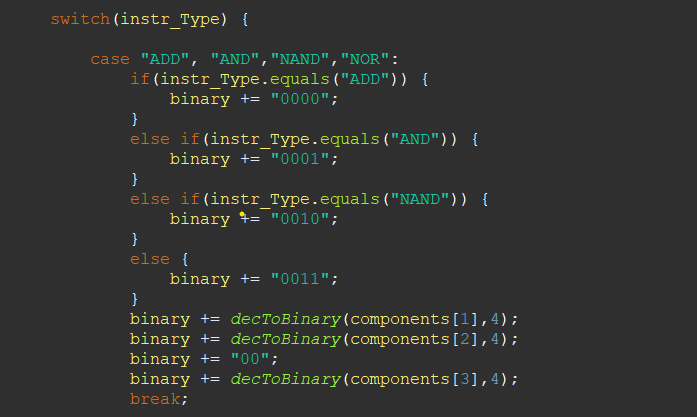
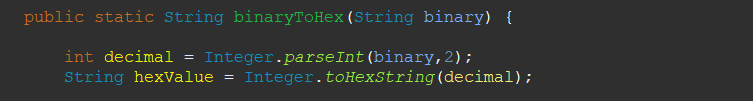
**PROJECT REPORT**

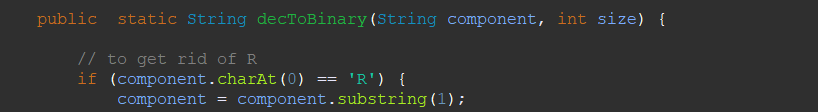
FIRST PART: INSTRUCTION SET ARCHITECTURE

The first stage is the stage where we decide what structure the incoming instructions will be, that is, the part where we design the instruction set architecture (ISA). As stated in the project file, the instructions are 18 bit. We set the opcode between bits [17:14]. The next 4 bits [14:11] were used as register bits where necessary. We tried to align the register bits (src/dst, src1/op1, src2/op2). We added an extra instruction and its name is reset. Because when the system first started, we wrote the assembler accordingly and it constantly starts with a reset. It puts the reset instruction first. 

SECOND PART: ASSEMBLER

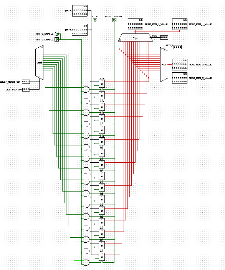
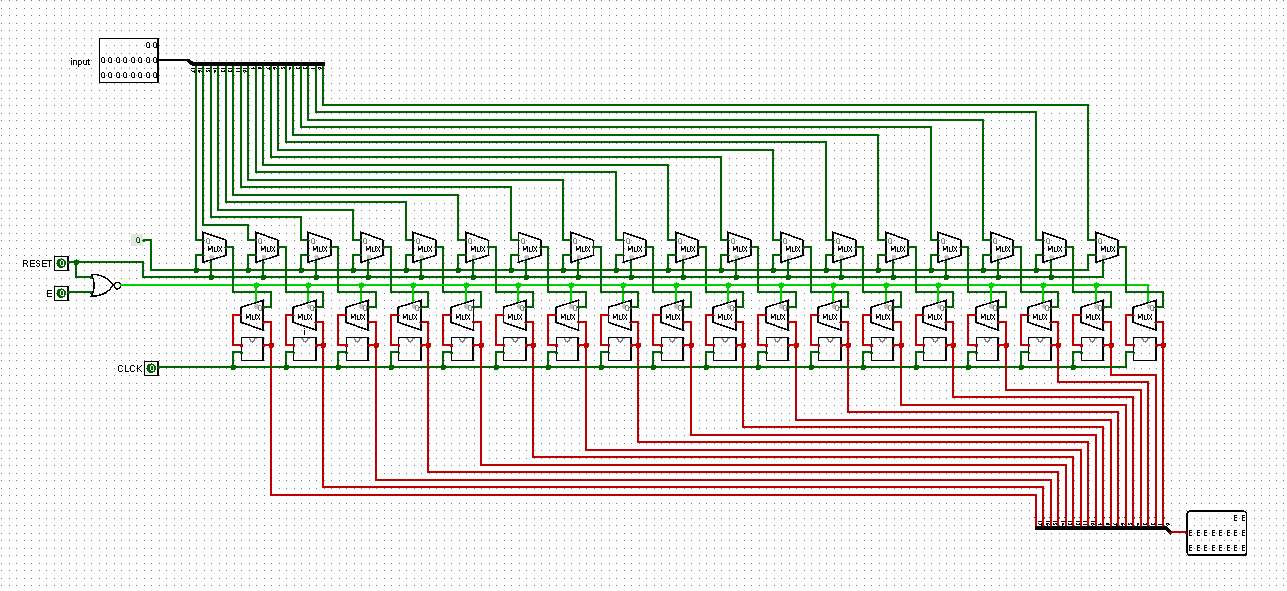
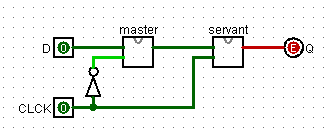
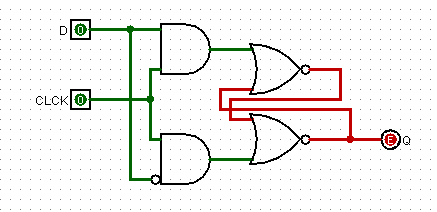
In Part 2, we wrote an assembler code and stated that the instructions coming to us would be understood by the logism. We used Java as the language. For opcodes, we added them to our output in binary form with a switch case. For the other parts, we converted the decimal value we received to binary and then converted the binary value to hex and wrote it to the output file. The written address and immediate values were written in accordance with 2s complement.





THIRD PART: LOGISIM COMPONENT DESIGN

Register File

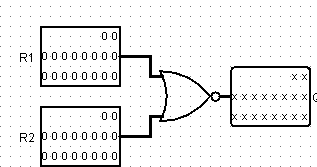
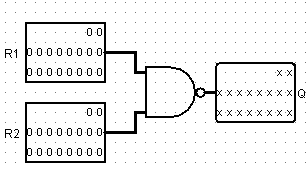
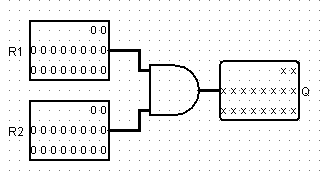


In this part, we first created the registers needed to hold our values from the foundation (starting from the dlatch). Dlatch--> Dflipflop-->18bitsRegisters-->RegisterFile

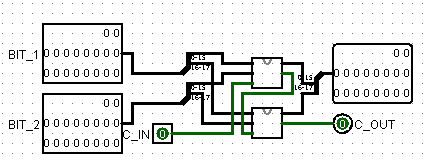
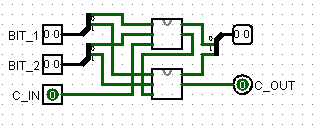
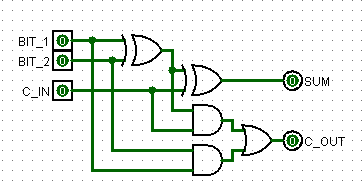
We have a total of 16 registers in the RegisterFile (as requested in the project). Therefore, the value of each register is represented by 4 bits. We have 3 register values in this circuit. 2 of them are for reading. The other is for writing. In addition to the register for writing, we also have an 18-bit data input. We have clock and enable pins. We also have a reset input if you want to reset our registers. This circuit is designed to read 2 values simultaneously. A circuit that can read 2 values and send the data to the control unit if there is an ALU operation. We decide which registers we will write to with the decoder, and which registers we will read from with the mux.

ALU

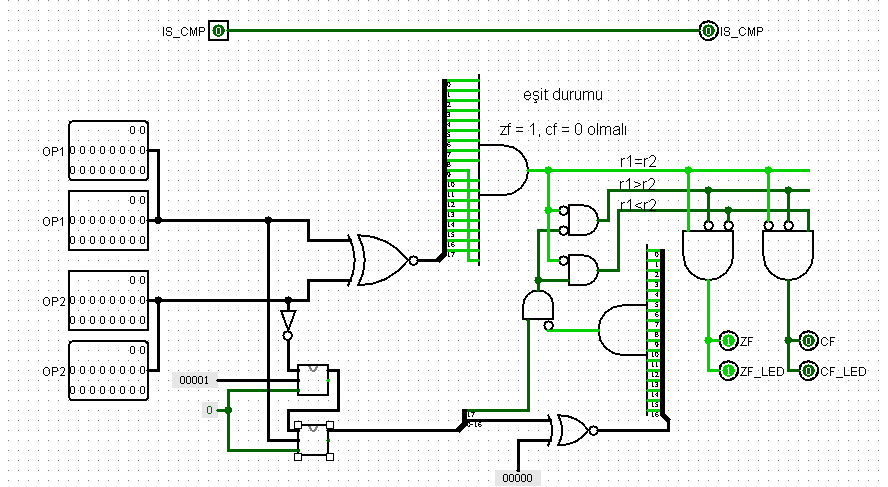
First of all, I will show the content of our alu processes.

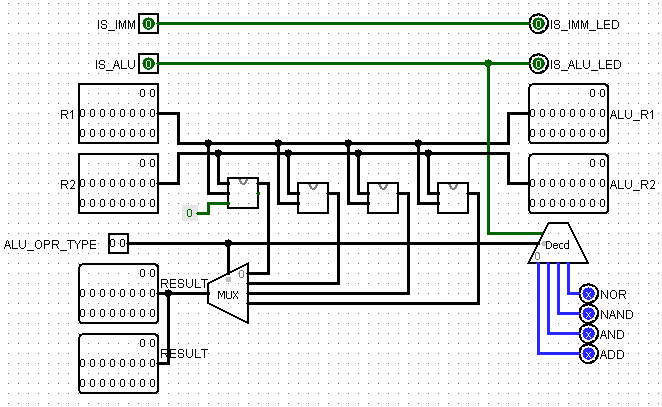


These three circuits are the simplest and for nand and nor. Circuits that take 18-bit data and output it with 1 gate.

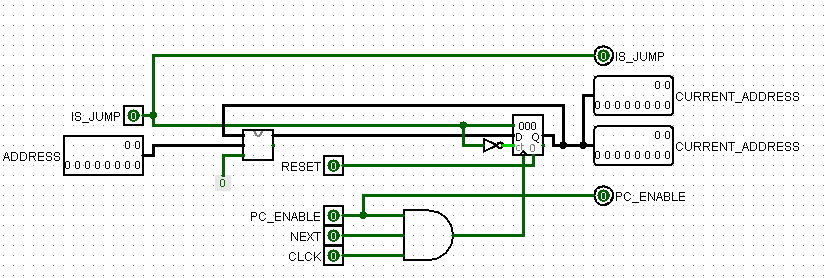


The above circuits are the circuits created starting from the most basic (one bit) and used inside each other as a closed circuit and increased to 18 bits.

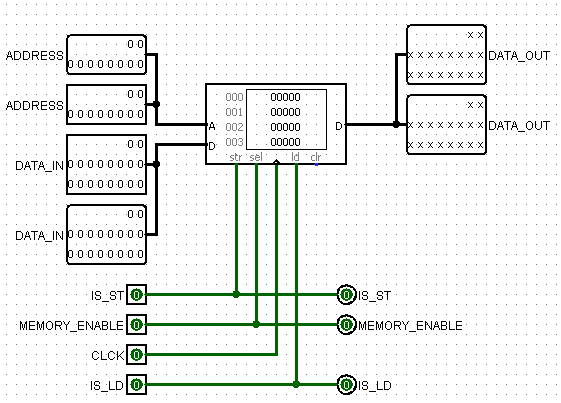
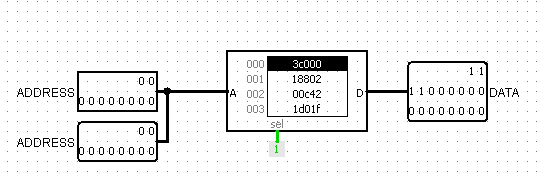
The circuit on the side enables the comparison of two incoming 18-bit data and finally sets the zf cf flags. First, we get the negative value by reversing the 2nd value and adding the value 1. Then we add it with the first value and actually apply a subtract operation. Afterwards, we burn the necessary doors with door operations.



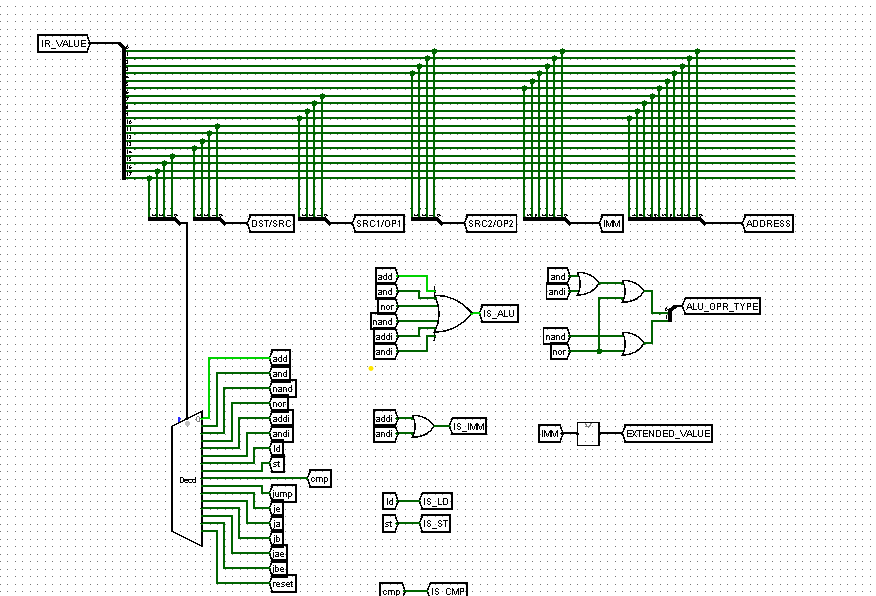
Our ALU closed circuit ensures that the value according to the instruction is thrown to the output after the incoming 18-bit data goes through all the operations. It also receives information about which value this value will be from the control unit.

Program Counter

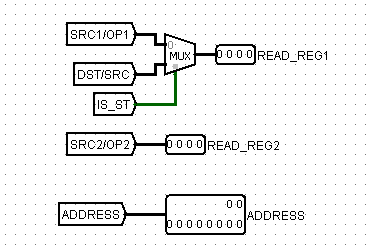
The program counter is the circuit where the next address assignment is made according to the jump information of the 10-bit instruction address we receive. It allows moving to the next address according to the next and pc\_enable bits coming from the Finite State Machine outputs. It allows going to the next value by adding 1 with the incoming value or to the jump address by adding it with the jump value. However, the buildin counter was used because the counter we wrote worked normally but caused problems when we connected it to the circuit.

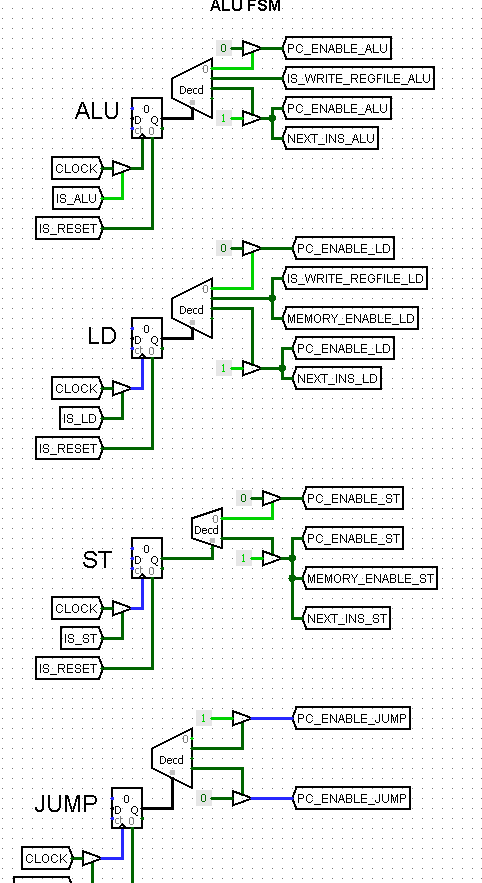
RAM and ROM

Our ROM circuits, which store our instructions, and our RAM circuits, which store our data, choose which action to take, thanks to some inputs from the control unit (load enable or store enable etc.).

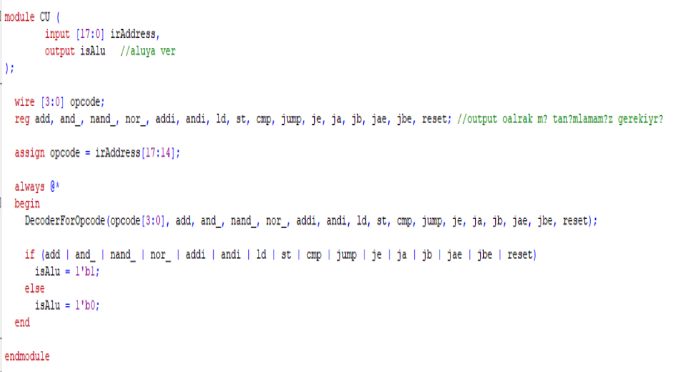
FOURTH PART: CONTROL UNIT

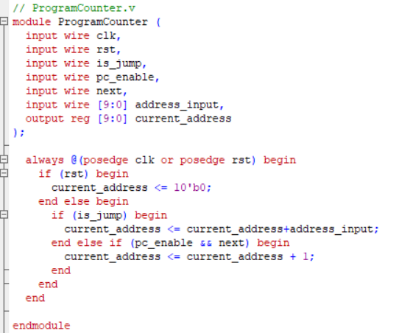
In the fourth step, there is the control unit. The instructions that come as a result of the coding we do in the assemblies are kept in the instruction register (ROM) and are decoded by the control unit as their turn comes. The decoding process is like choosing which opcode of the binaries we separated with a splitter using a multiplexer. When an instruction is decoded, it signals everything that needs to be done. For example, when an Ld instruction arrives, we connect the tunnel we get from the opcode to the is\_ld signal and indicate to the system that this is a load operation.

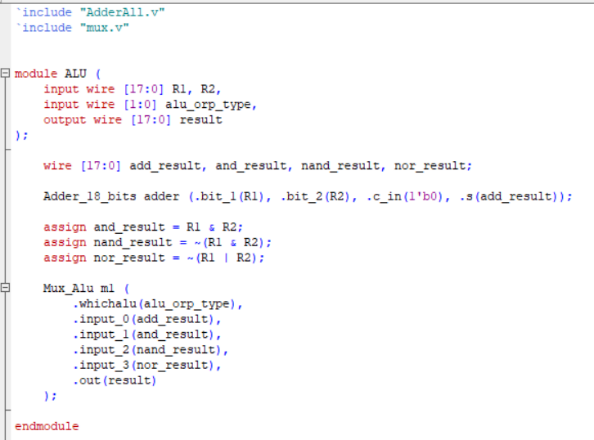
Matching the register bits we made in the ISA section while establishing the register addresses that need to go is useful here. The bits we select while decoding can also have more than one address. We choose among these according to the signals triggered by the incoming transaction.For example, by checking whether there is a store operation on the side, it is determined which address will be sent to the register file.

After decoding all instruction information, we set up a finite state machine in synchronization with the clock. We had to create separate fsm for each process because each process has separate states. For example, in an alu operation, there is a state that we need to write to the register, while in a store operation, there is a state that we need to access to the ram. As a result, separate finite state machines were installed for ALU operations, load/store operations and jump, and the counter we made here could not be used because it gave an error even though it worked normally. After adjusting the maximum values of the build counters to the number of operations of each instruction (for example, 3 steps in alu but 2 in jump), the necessary signals (next and pc\_enable used in the program counter) were set in the required steps. For example, after decoding and processing in an ALU operation, when the step of writing it to the register comes, the counter increases by 1 and lights the is\_write\_register\_alu signal. In the last step, it turns on the pc\_enable\_alu signal and moves on to the next step. The reason why pc\_enable and other signals are separated according to each instruction is to avoid complexity. For example, if every pc\_enable was used jointly, pc\_enable would be active in other fsms except for the active instructions.

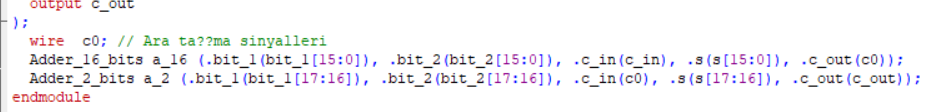
FIFTH PART : VERILOG PART

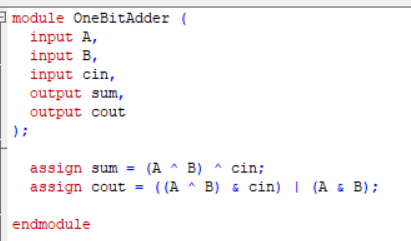


In the Verilog section, we tried to write a code that does the same operations without being too faithful to our own logisim circuit. The control unit is again the decoding side. In the program counter, output is always assigned with the input and signals coming at the time of the posedge clock.



Assignment is made to the output according to the operation selected on the Alu. After being decoded in an instruction control unit, they are processed in their small codes. Thanks to the posedge clock, transactions continue at every clock posedge moment.



As in Logisim, the operations performed here start from 1 bit code and build up to 18 bits.

Eren ÇETİN 150120008

Zeynep YILMAZ 150120066

Semih BAĞ 150120070