Part 1 – Assembly Language

First of all, we all sit together thought about it how can we design our assembly language so that we can easily implement the circuit part. We decided that setting op code for 4 bits is the best because we have 15 different opcodes. Because of that 2^4 = 16 we can set 16 different instructions for our ISA. After that we created a template ISA for our processor. We used 4 bits representing the registers because we are going to have 16 registers. Also, we can have 6 bits for representing immediate values for ADDI and ANDI. Because immediate values can be negative too the values for immediate are in [-32,31]. For LD and ST, we are using every bit of 18 bits. 4-bit opcode, 4-bit register and 10 for address. For JUMP, JE, JA, JB, JAE, JBE we are using 11-bit address. Because the given addresses can be negative and our address width 10 bit. If we think that we are that end of the program counter, which is 1023, and we want to go all the back down we need negative -1023 and this can be happen by subtracting 1023. Therefore, we need 11 bits to represent -1023 without any hobble. And we decided to represent all the meaningless bits as 0’s. This is our ISA template.

A screenshot of a document

Description automatically generated

Instructions

A table with numbers and letters

Description automatically generated

After creating our ISA template, we started to writing our assembler which will take an list of instruction, then convert them into hexadecimal instruction shape. We decide use STRATEGY pattern for this because some of the instruction same argument just the opcode was different. Then we created 6 strategies in total. Our program reads line by line the instructions in the given txt file which is needed renamed as “input.txt”. After writing our assembler we tested that our assembler works without any bug so we can move to part 2 of the project.



Instructions in hexadecimal

Part 2 – Logisim Component Design

**Register File**

First, we created a D Latch to create a D Flip-Flop. After we created D Flip-Flop using 2 D Latch which are master and slave. Then we created a 18-bit register using 18 D Flip Flop, splitters and a clock signal. After that we created a Register File which contains 16 registers in total. Each register has 18 bits to store. Also, there are 7 different inputs in our register file. Two of them are read register input which are 4 bits to read two registers from register file. One of them is specifying the register that the data written inside of it. One of them is represents the data will be written inside of specified register. There is a write enable signal for enabling writing on register. There are also one clock signal and one reset signal.

**ALU**

To create our Arithmetic-Logic Unite we needed to create a 18 bit adder. To create a 18 bit adder we think that creating a 1 bit adder is compulsory and creating 2 bit adder will help to implement 18 bit adder. So we created a 1 bit adder and after create a 2 bit adder using 2 1-bit adders. We will implement 18 bit using 9 2-bit adders.

In alu we have 4 different operations. To select one of them we need one control input which is size of 2 bits. Our fours operations implemented with one 18-bit adder, one and gate, one nand gate and one nor gate.

We also created a subtractor and comparer to use in following parts.

Part 3 – Logisim Design with Control Unit

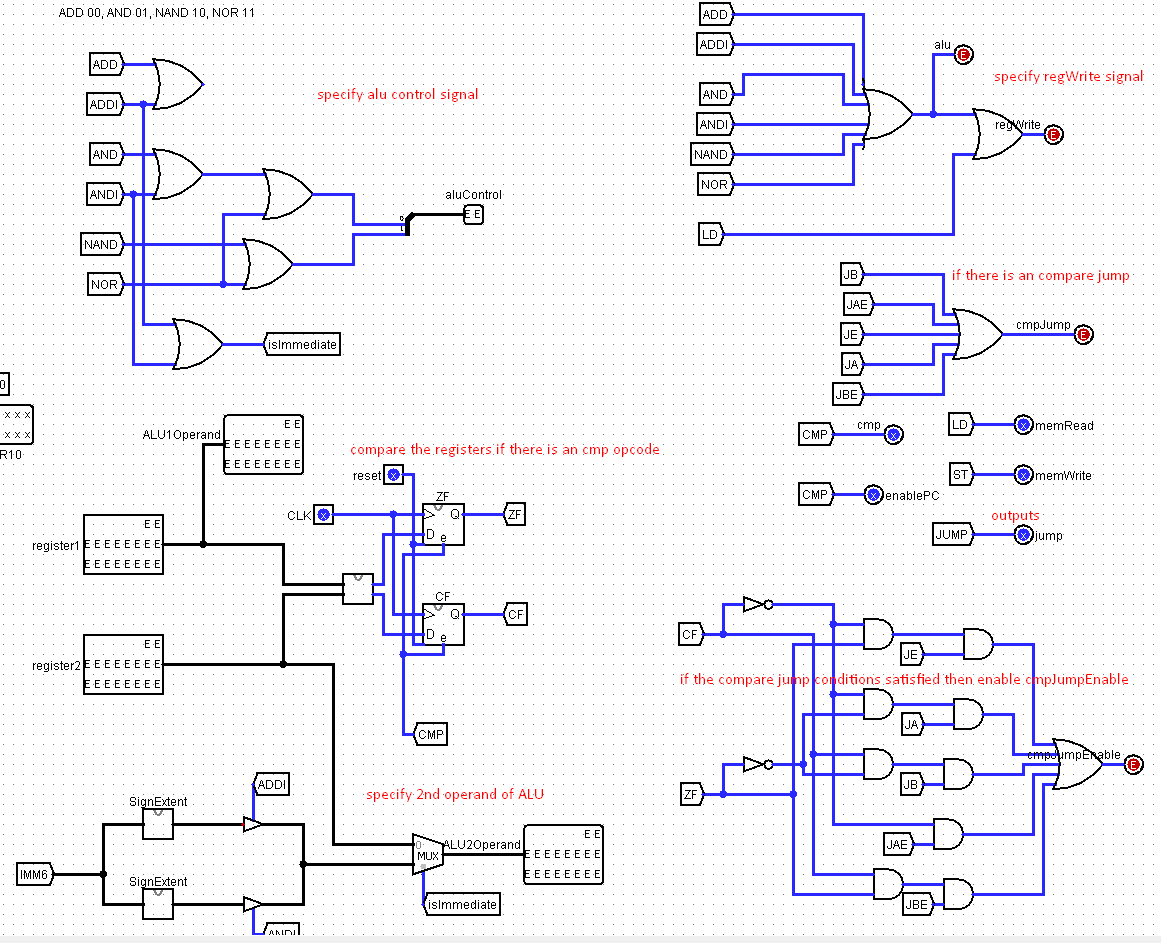
**Control Unit**

First we created a control unit to split the current instruction on ROM. And the specific flags will be turn on and the for specific opcodes the processor will enter a specific state. This is an example of our Control Unit.

A diagram of a machine

Description automatically generated

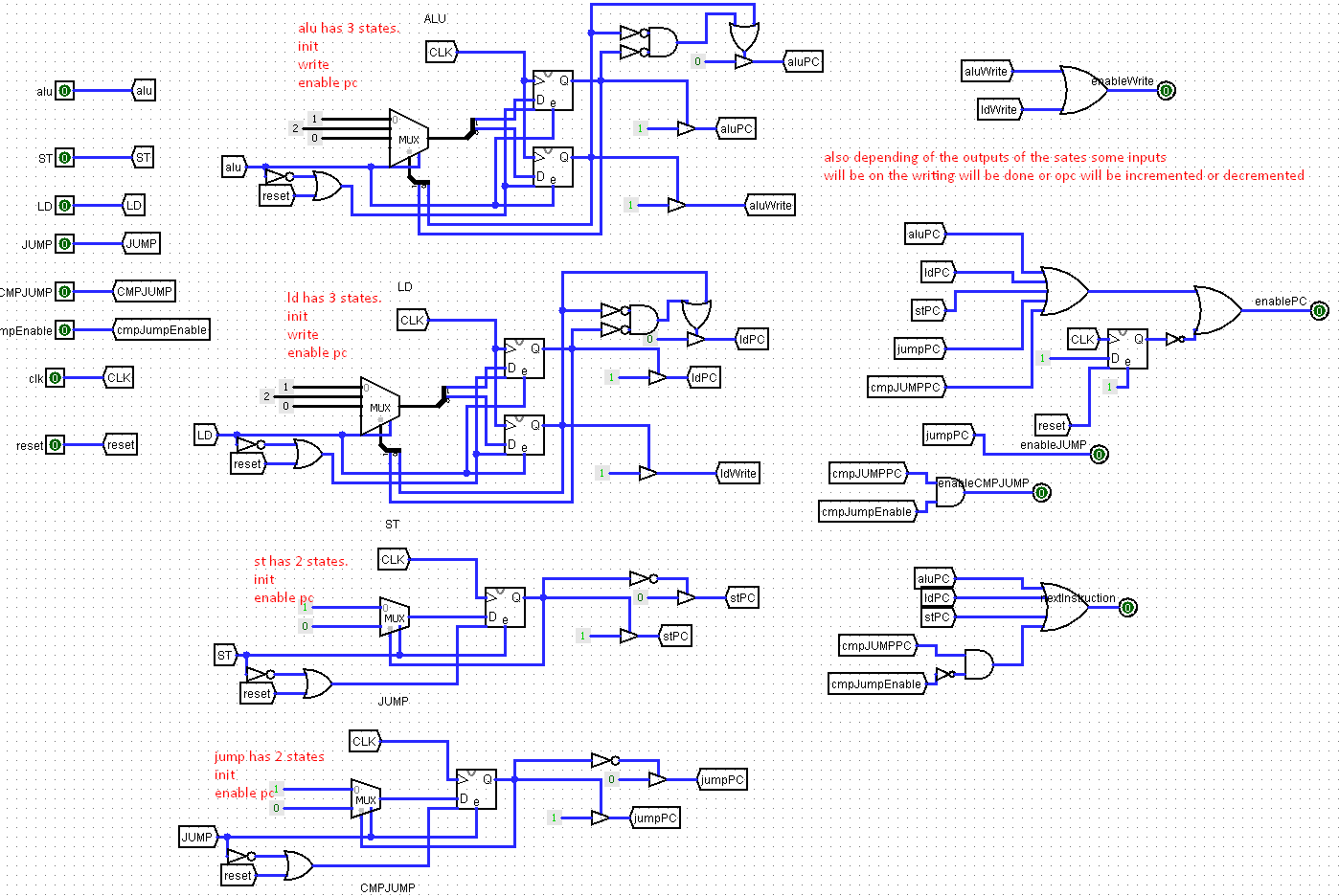
Specifying the arguments and opcode.



The rest of control unit.

**FSM**

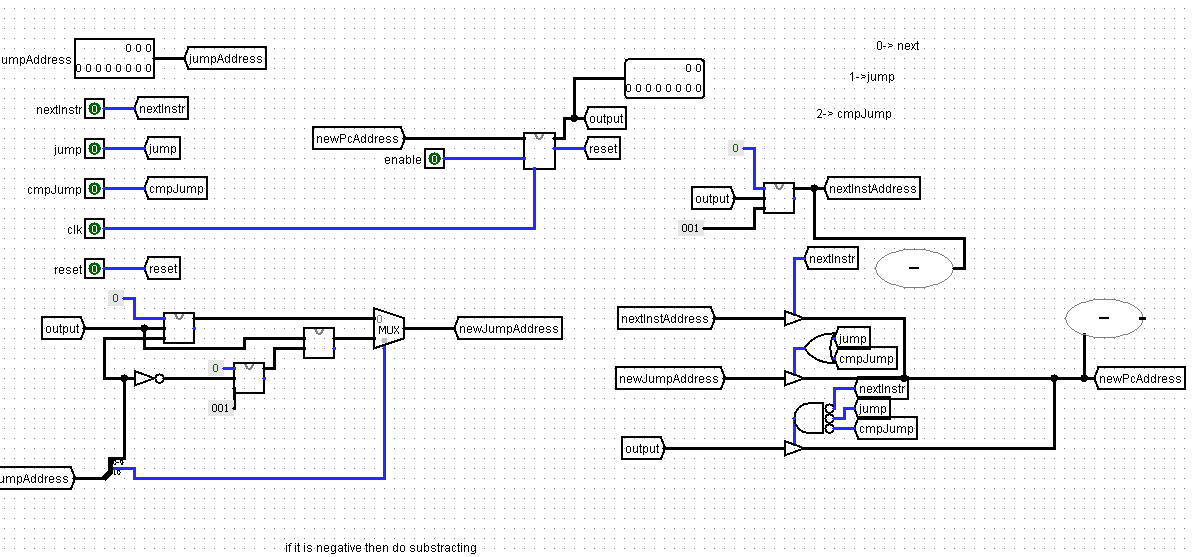
We created a finite state machine to handle synchronous processes. Which are calculating ALU then writing into register, or loading data from memory etc.



We implemented FSM using 1 or 2 1-bit register. One of the states will be entered depending on the signal coming from control unit.

**PC**

We created a PC which will be incremented depending on the signals coming from control unit and fsm. The signal can be enablePC, nextInstruction, jump, cmpjump. If enablePC is on then the pc value will be set. The new value of the PC will be decided to the inputs jump, cmpjump and nextInstruction. If nextInstruction is on then PC will be incremented by 1. If one of the jump and cmpjump is on then the PC value will addition of currentPCValue and the PCOffset coming from control unit.



**Datapath**

**A diagram of a computer

Description automatically generated**

This is our datapath.