CSE3015 Digital Logic Design Term Project Report

Students

Emir Büçkün

İrem Kubalas

Baran Özyurt

Enes Karaduman

Instructor

Asst. Prof. Betül Boz

Date

10 January 2023

Introduction

In this project, we are expected to design and implement a processor which supports these instruction sets:

SUB, ADD, AND, OR, XOR, ADDI, SUBI, ANDI, ORI, XORI, LD, ST, JUMP, PUSH, POP, BE, BNE

Processor will have below components and properties:

- Register File
  - Contains 20 bit x 16 registers
- Instruction Memory (read-only memory) 10 bits address width
  - 10 bits address width
  - 20 bits data width
- Data Memory (read-write memory)
  - 10 bits address width
  - 20 bits data width
- Control Unit
  - Produces proper signals to all data path components
- Arithmetic Logic Unit (ALU)
  - Add, Sub, And, Or, Xor operations

Assembler

In order to manage the instructions, we first design a 20 bits of instruction set architecture (ISA) which shows our way of representing the instructions. To use 10 bits for addresses in the ISA, we used two bits opcode for four group of instructions. These are:

1. ADD, SUB, AND, OR, XOR, LD, ST, JUMP -> 00
2. ADDI, SUBI, ANDI, SUBI, ANDI, ORI, XORI, PUSH, POP -> 01

3. BE -> 10
4. BNE -> 11

After separating BE and BNE instructions, we used another three bits opcode for the remaining instructions. In this way, each instruction had a unique opcode code sequence.

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	0	0	0	0	0	DST				SRC1				x	x	x	SRC2			
SUB	0	0	0	0	1	DST				SRC1				x	x	x	SRC2			
AND	0	0	0	1	0	DST				SRC1				x	x	x	SRC2			
OR	0	0	0	1	1	DST				SRC1				x	x	x	SRC2			
XOR	0	0	1	0	0	DST				SRC1				x	x	x	SRC2			
LD	0	0	1	0	1	DST				x	ADDR									
ST	0	0	1	1	0	SRC1				x	ADDR									
JUMP	0	0	1	1	1	x	x	x	x	x	ADDR									
ADDI	0	1	0	0	0	DST				SRC1				IMM						
SUBI	0	1	0	0	1	DST				SRC1				IMM						
ANDI	0	1	0	1	0	DST				SRC1				IMM						
ORI	0	1	0	1	1	DST				SRC1				IMM						
XORI	0	1	1	0	0	DST				SRC1				IMM						
PUSH	0	1	1	0	1	SRC1				x	x	x	x	x	x	x	x	x	x	x
POP	0	1	1	1	0	SRC1				x	x	x	x	x	x	x	x	x	x	x
BE	1	0	REG1				REG2				ADDR									
BNE	1	1	REG1				REG2				ADDR									

ISA

We used Java to create assembly codes from given input set. Assembler's purpose is to create an input file for Logisim.

In our Java program, we have the instruction set as a string array, 7 bits for immediate values, 10 bits for address values, our input file name is “input.txt” and output file name is “output.txt”.

```
private final String[] instructionSet = {
    "ADD", "SUB", "AND", "OR", "XOR", "LD", "ST", "JUMP",
    "ADDI", "SUBI", "ANDI", "ORI", "XORI", "PUSH", "POP",
    "BE", "BNE"
};
private final int immBit = 7;
private final int addressBit = 10;

private final static String inputFile = "input.txt";
private final String outputFile = "output.txt";
```

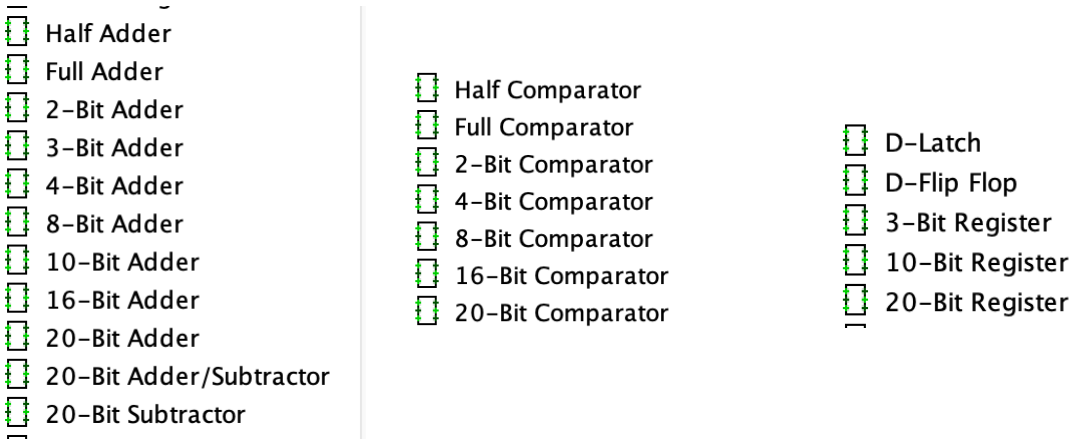
Constant Values In The Program

```
switch (opcode) {
    case "ADD", "SUB", "AND", "OR", "XOR" -> {
        sb.append(registerToBinary(parseData[1])); // DST
        sb.append(registerToBinary(parseData[2])); // SRC1
        sb.append(str: "000");
        sb.append(registerToBinary(parseData[3])); // SRC2
    }
    case "ADDI", "SUBI", "ANDI", "ORI", "XORI" -> {
        sb.append(registerToBinary(parseData[1])); // DST
        sb.append(registerToBinary(parseData[2])); // SRC1
        sb.append(immOrAddrToBinary(parseData[3], isImm: true)); // IMM
    }
    case "LD", "ST" -> {
        sb.append(registerToBinary(parseData[1])); // DST/SRC1
        sb.append(str: "0");
        sb.append(immOrAddrToBinary(parseData[2], isImm: false)); // ADDR
    }
    case "JUMP" -> {
        sb.append(str: "00000");
        sb.append(immOrAddrToBinary(parseData[1], isImm: false)); // ADDR
    }
    case "PUSH", "POP" -> {
        sb.append(registerToBinary(parseData[1])); // SRC1
        sb.append(str: "00000000000");
    }
    case "BE", "BNE" -> {
        sb.append(registerToBinary(parseData[1])); // DST
        sb.append(registerToBinary(parseData[2])); // SRC1
        sb.append(immOrAddrToBinary(parseData[3], isImm: false)); // ADDR
    }
}
```

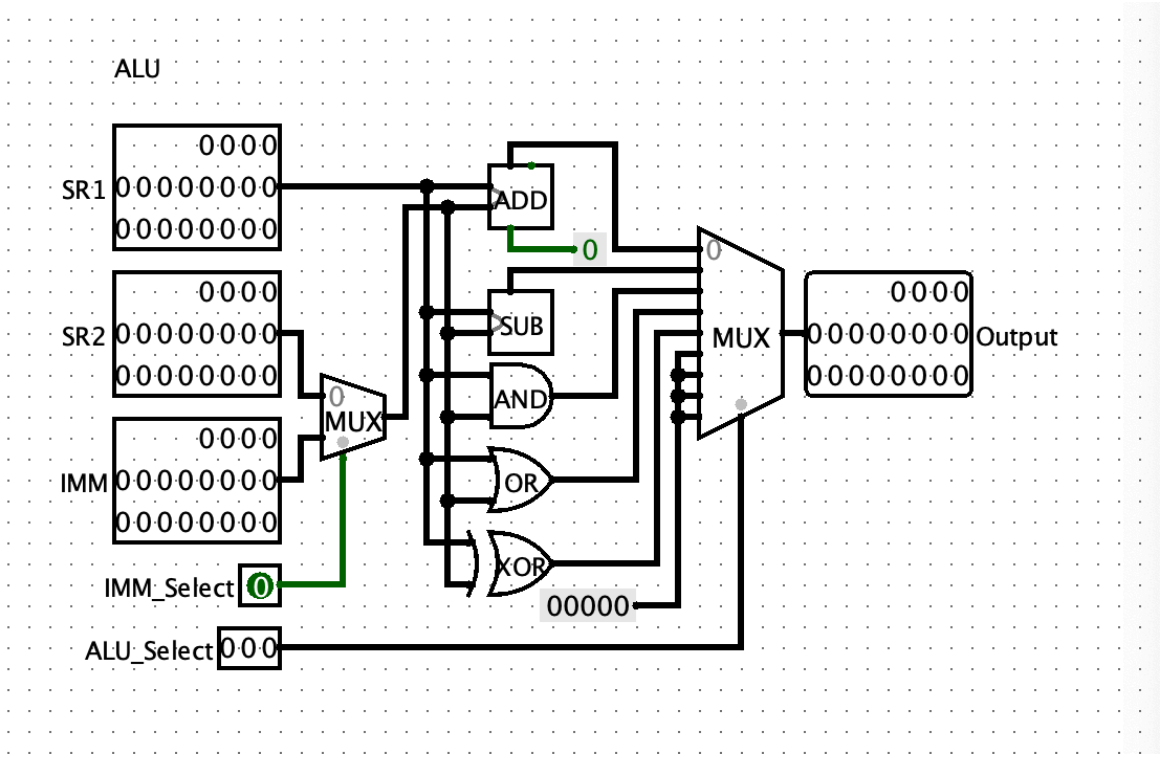
Determining Assembly Codes Using Switch-Case Statement

Logisim Design

We always used small components to design bigger and more complex components. To have a 20 bits adder, we used 2, 4, 8 and 16 bits adders. We tried to begin from the smallest part and then join them to produce bigger parts. We also used adders to create a subtractor and adder/subtractor.



ALU

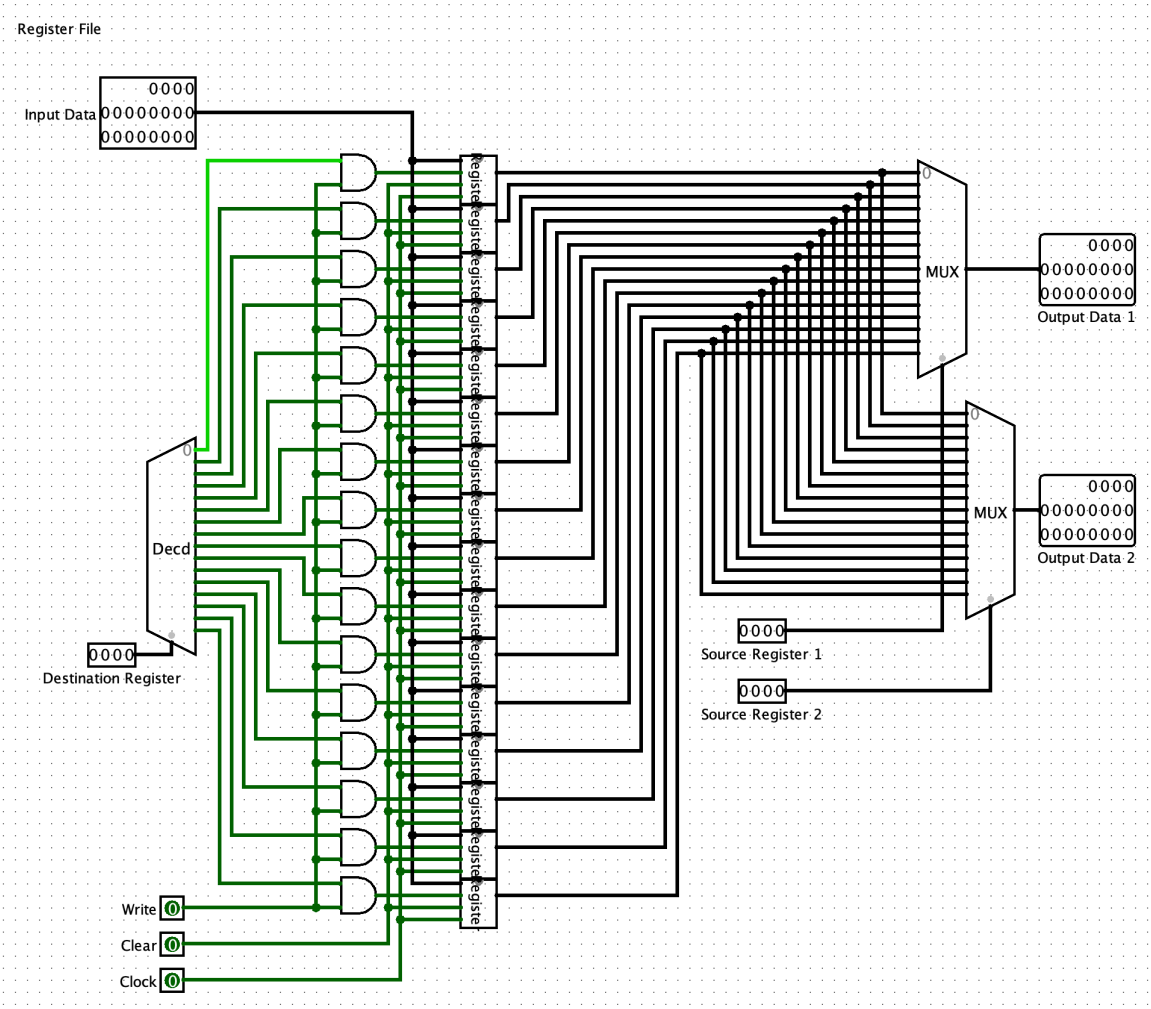


ALU Internal Design

We have ADD, SUB, AND, OR and XOR operations in the ALU. We select the input by using the IMM_Select signal and the operation that will be performed by using the ALU_Select signal. Both signals are coming from the Control Unit. Then we give the output after the operation performed.

Register File

We have 16 20-bit registers in our Register File. We used 4 bits to represent each register from zero to fifteen. We use Source Register signals to read a register. If we want to write to a register, we need to use Destination Register and Write signal together and then we can write the Input Data to the selected register. Moreover, we have a option to clear all the registers.

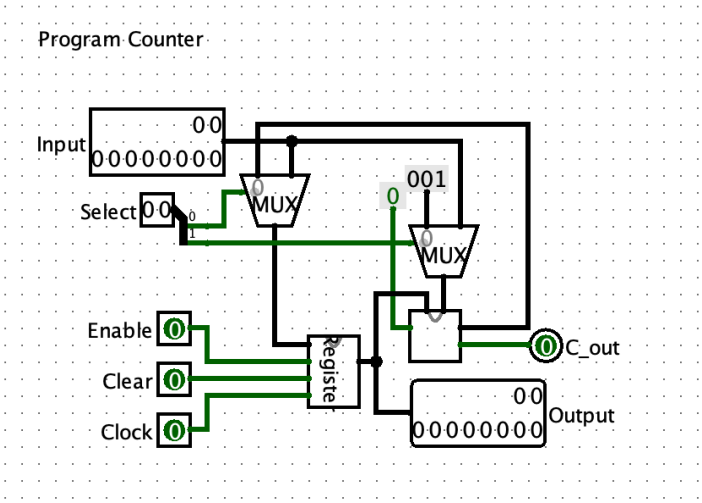


Register File Internal Design

Program Counter

In our program counter, we have a Select signal to decide which instruction will be next, a Enable signal to write to the register, a Clear signal to clear the register and a input data to jump directly or increment the PC.

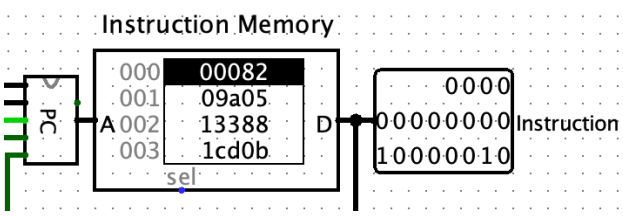
- Select = 00 -> Increment the PC value by one
- Select = 01 OR 11 -> Jump to the input data
- Select = 10 -> Increment the PC value by given input value

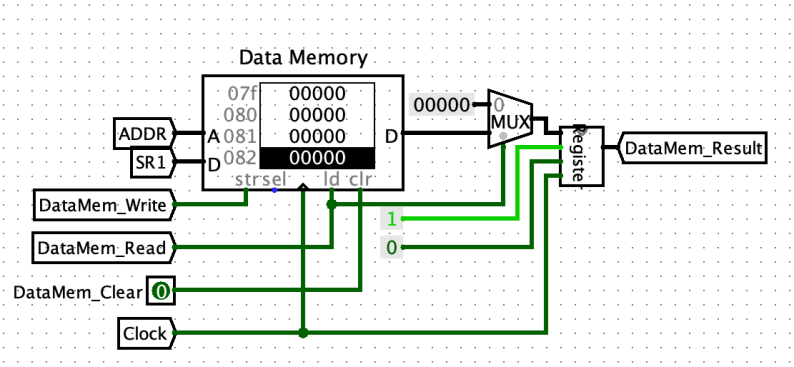


Program Counter Internal Design

Data Memory & Instruction Memory

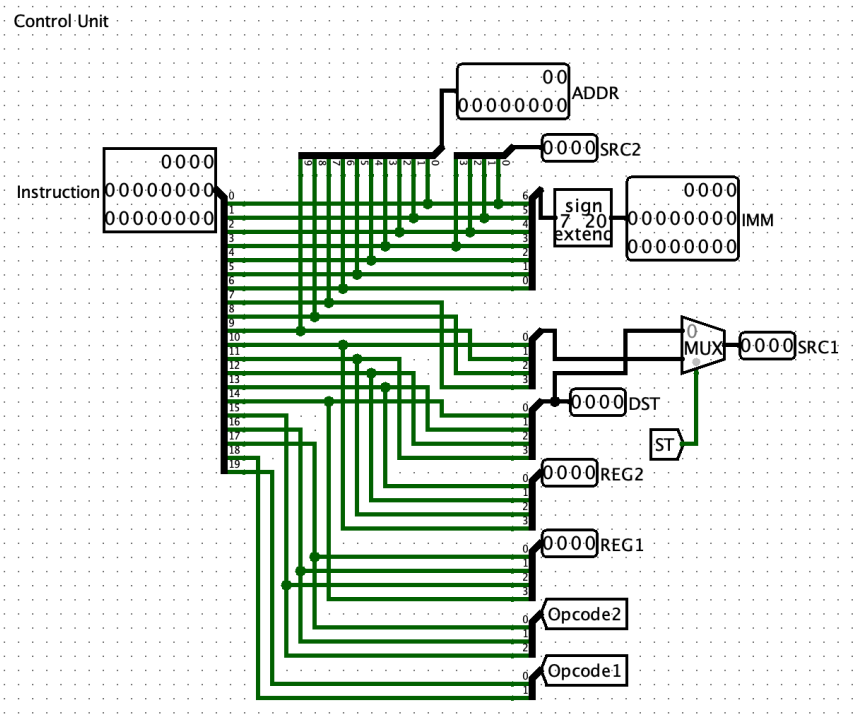
We used RAM and ROM components with 10 address bits width and 20 data bits width for Data Memory and Instruction Memory. Instruction Memory address input is Program Counter's output and its output is input to the Control Unit. For the Data Memory, we have two input that are address and source register 1 coming from Control Unit, four signals that are Write, Read, Clear and Clock.





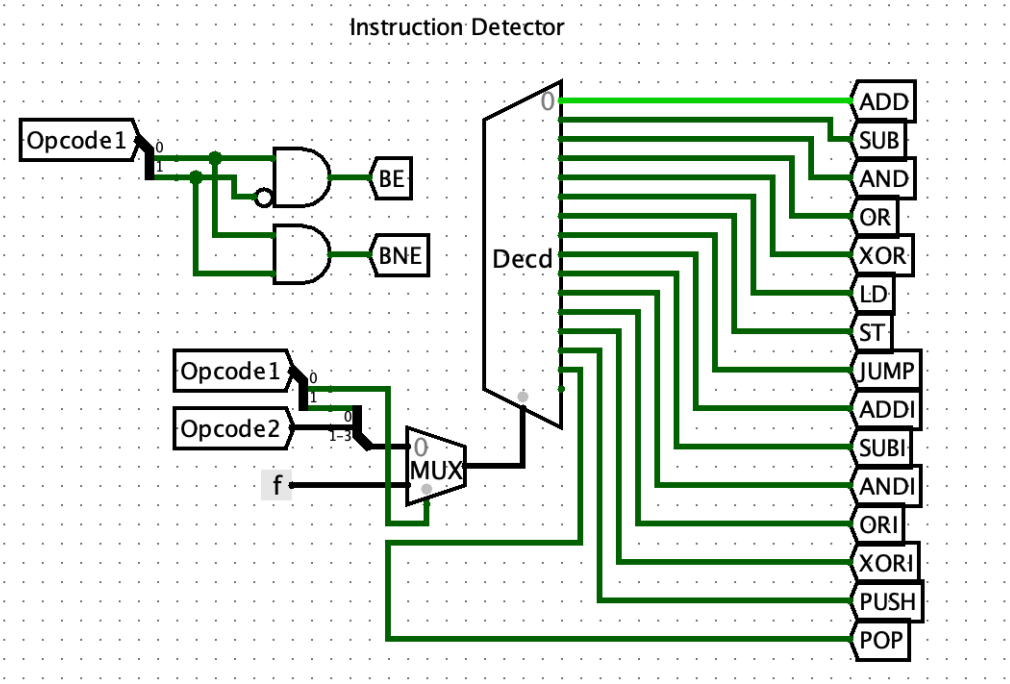
Control Unit

We have three input in our Control Unit design. These are the current instruction, clear signal and clock signal. Using the current instruction, we use our ISA design to decide address, immediate, registers and opcodes values.



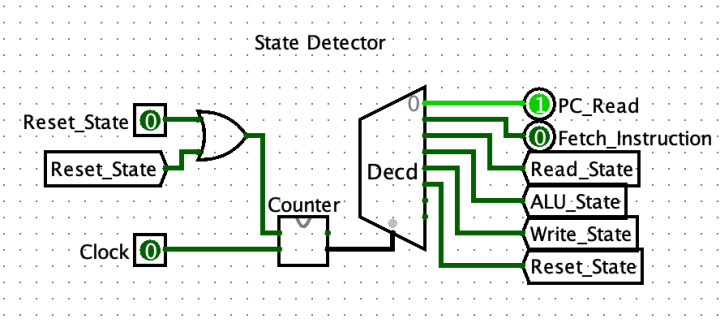
Deciding Parts of The Instruction In Control Unit

After getting the opcodes, we choose the given instruction with Instruction Detector. Opcode1 signal is especially for BE and BNE instructions. Then Opcode1 and Opcode2 signals used together to select from the remaining instructions.



Instruction Decision Mechanism

On the other hand, we should manage the states of the instructions. We have a State Detector for that purpose. In the detector, there is a counter that counts in every clock cycle and walks through the next state. There six states that are PC_Read, Fetch_Instruction, Read_State, ALU_State, Write_State and at the end Reset_State that resets the state cycle. It can be reseted by using the Reset_State input manually.

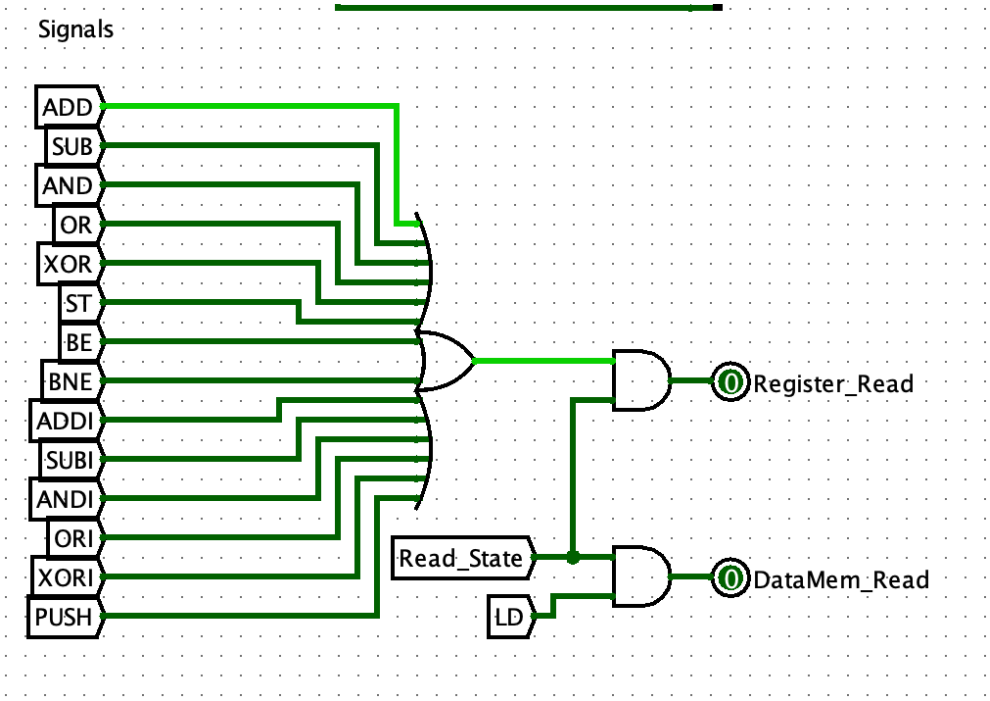


State Decision Mechanism

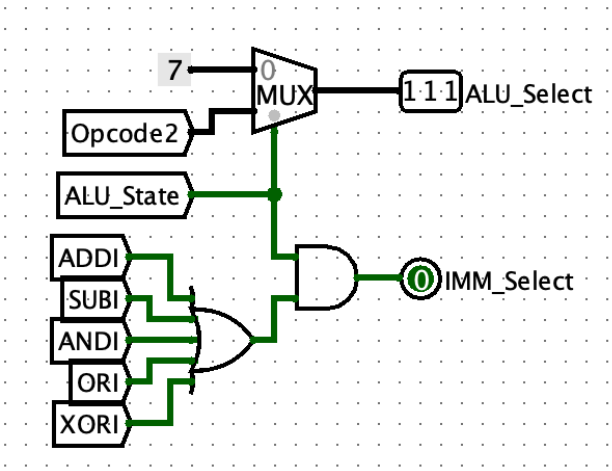
Our signals are as follows:

Register_Read, DataMem_Read, ALU_Select, IMM_Select, Register_Write, Register_Load, DataMem_Write

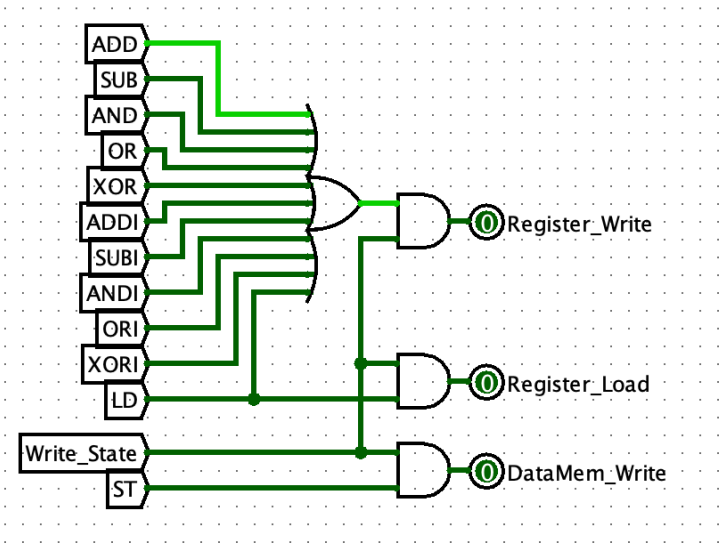
We decided which instructions will take which operations. Using that information and the logic gates, we prepared the signals.



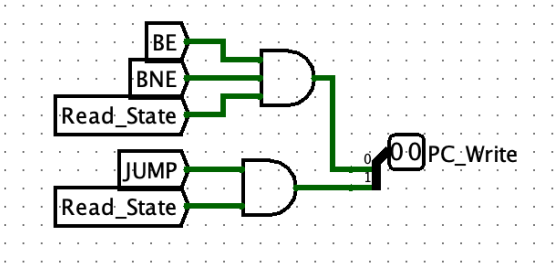
Read Signals Decision



ALU Signals Decision

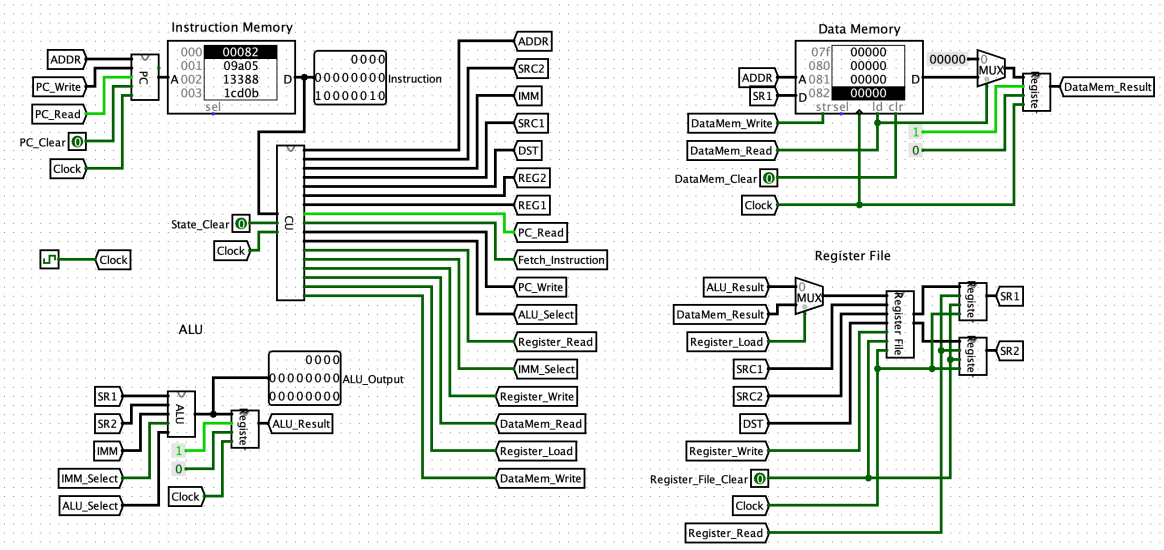


Write Signals Decision



PC_Write Signal Decision

At the end, we combined these signals and the data path components.



Main Design