**DIGITAL LOGIC DESIGN TERM PROJECT REPORT**

**Kadir Hızarcı – 150116004**

**Ömer Faruk Çakı – 150117821**

1. **Introduction**

As the first part of our project, we needed to create an assembler. Before coding it, we had to assign our opcodes and registers valid values. Since there exist 13 instructions, we needed at least 4 bits dedicated to the instructions.

Considering MSF as the 1st bit, we created the truth table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instructions** | **1st** | **2nd** | **3rd** | **4th** |
| ADD | 0 | 0 | 0 | 0 |
| ADDI | 0 | 0 | 0 | 1 |
| AND | 0 | 0 | 1 | 0 |
| ANDI | 0 | 0 | 1 | 1 |
| CMP | 0 | 1 | 0 | 0 |
| JE | 0 | 1 | 0 | 1 |
| JA | 0 | 1 | 1 | 0 |
| JB | 0 | 1 | 1 | 1 |
| JBE | 1 | 0 | 0 | 0 |
| JAE | 1 | 0 | 0 | 1 |
| LD | 1 | 0 | 1 | 0 |
| ST | 1 | 0 | 1 | 1 |
| JMP | 1 | 1 | 0 | 0 |

Next up, we needed to do the same for the registers. Since we had to create 16 registers, we devoted 4 more bits for their notation. We created the truth table for them down below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Registers** | **1st** | **2nd** | **3rd** | **4th** |
| R0 | 0 | 0 | 0 | 0 |
| R1 | 0 | 0 | 0 | 1 |
| R2 | 0 | 0 | 1 | 0 |
| R3 | 0 | 0 | 1 | 1 |
| R4 | 0 | 1 | 0 | 0 |
| R5 | 0 | 1 | 0 | 1 |
| R6 | 0 | 1 | 1 | 0 |
| R7 | 0 | 1 | 1 | 1 |
| R8 | 1 | 0 | 0 | 0 |
| R9 | 1 | 0 | 0 | 1 |
| R10 | 1 | 0 | 1 | 0 |
| R11 | 1 | 0 | 1 | 1 |
| R12 | 1 | 1 | 0 | 0 |
| R13 | 1 | 1 | 0 | 1 |
| R14 | 1 | 1 | 1 | 0 |
| R15 | 1 | 1 | 1 | 1 |

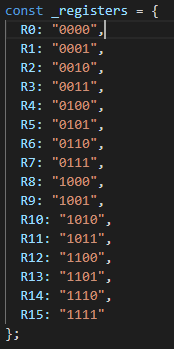
Since we finished our instruction set architecture, we moved onto creating an assembler program.

1. **Assembler**

We picked **JavaScript** as our language of choice and **Node.js** in order to code our assembler.

We included two 3rd party libraries *csv-parse* and *s-binary.* One for reading and parsing given input file and the other for helping the conversion between binary and hex-digit.

We defined our register values and opcodes inside the code as objects, shown below:



In the code snipped above, we used the array **asmData** to hold parsed values which we read from the **“instructions.txt”** in two pieces, as opcode and the rest of the instructions respectively.

Following for loop in looping all instructions one by one, there exist a few more variables inside the loop; opcode(holds binary value of given opcode), binaryValue which we will be defined in the next steps and the registers array (holding all current instructions as an array).

Then we used if-else statements to determine which steps will be applied to produce the correct output for the given instructions.



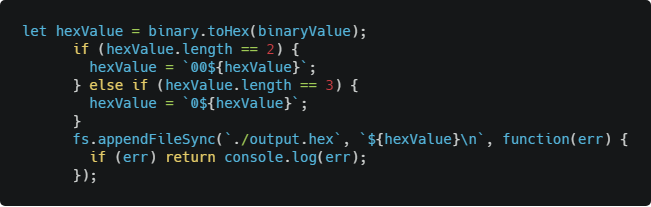
For example, the code above is checking whether our opcode is **ADDI** or **ANDI**, therefore we can conduct necessery steps.

**binayValue** variable holds the whole 16 bits the end-result for the instruction.

It is calculated as opcode(4 bits) + binary val. of 1st val. (4 bits) + binary val. of 2nd val. (4 bits) + result variable.

Results variable is defined as above. Since last value of the instruction is signed and can be either positive or negative, we are checking the value, if it is positive, we are converting to its binary value. And if it’s a negative value, we will first we assume it as a positive then convert it to a signed binary number. After last value is calculated we added it to the final binary value variable and we end up with 16 bits instruction value. Program converts that binary number to hexadecimal while saving it to the .hex file.

For the other opcodes, process is quite similar with a few unsignificant changes.



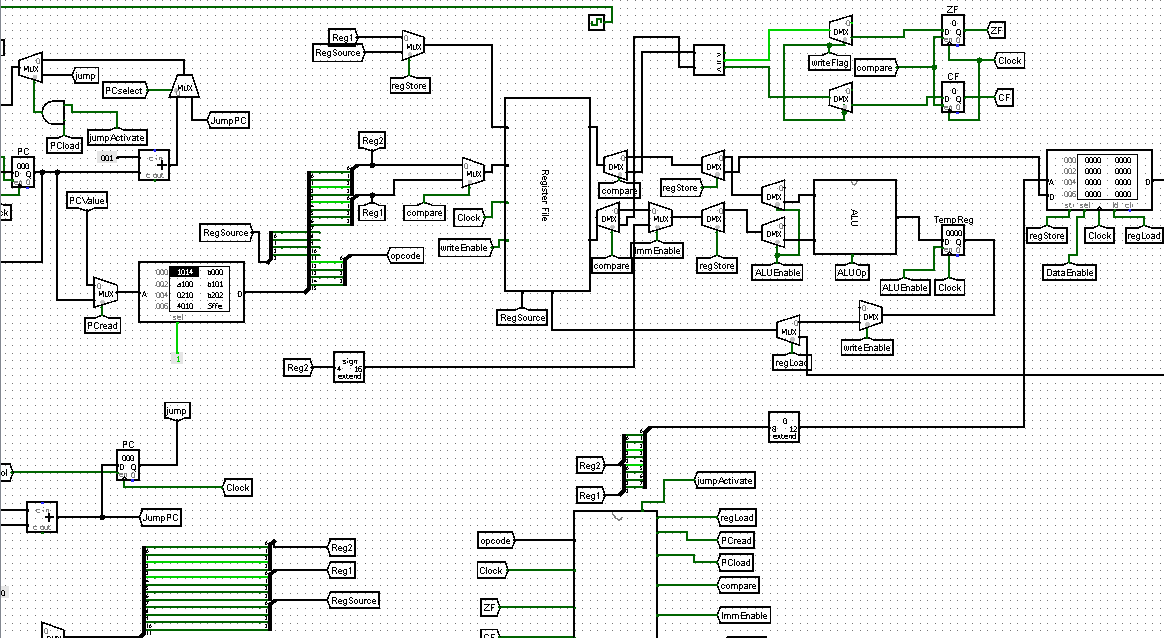
Lastly,we convert the 16 bits binary number we calculated in the previous step into hexadecimal representation, using the help of a 3rd party library. Since it is not including zeros in its output value ( for example “12” vs “0012”) we check the length of the output hexadecimal value and added zeros to the start.

Finally we wrote the results into **“AssemblerOutput.hex”** file.

1. **Logisim**

Our Logisim design has 4 circuits. Main circuit containing the clock, Program Counter, Instruction Memory and Data Memory. Control Unit containing our finite state machine implementations for each instruction and numerous output signals. Register File, containing 16 registers with both the ability to read from and write to them, depending on the signal given. Lastly, ALU, in charge of conducting ADD, ADDI, AND and ANDI instructions.

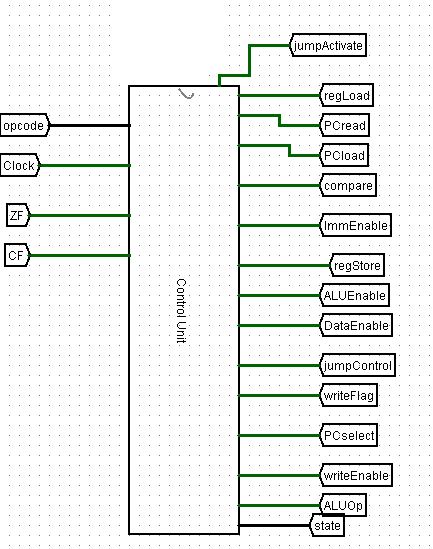
1. **Main Circuit**



**Components Used:** Registers, tunnels, AND gates, splitters, MUXes, DEMUXes, RAM, ROM, adders, bit extenders, Register File, Control Unit, ALU and clock.

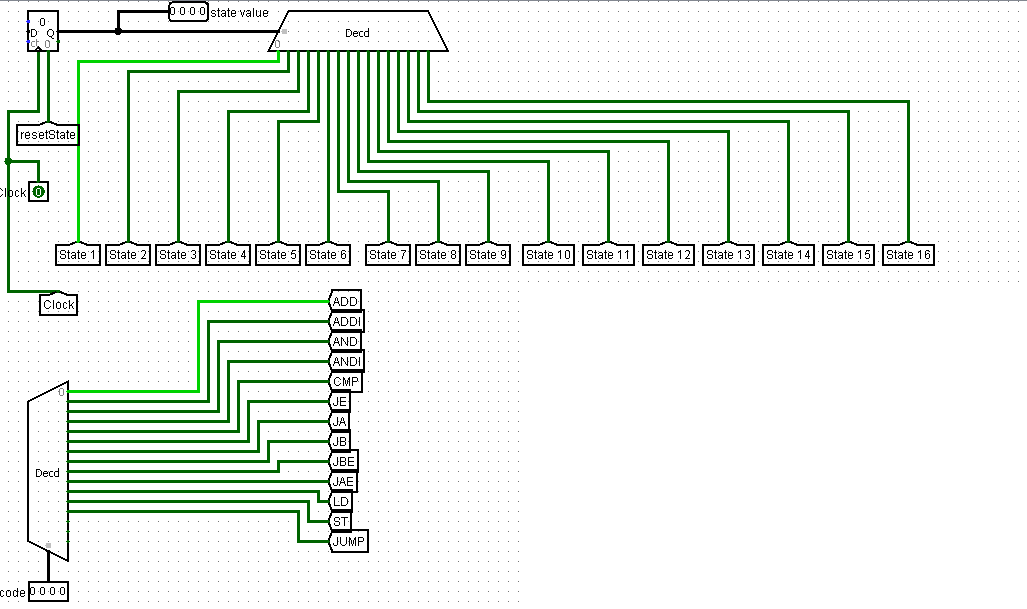
**Overview:** Program counter pointing the instruction address on ROM, instruction’s 16 bits gets split into pieces and depending on the signals, particular ones gets put into register. Extended usage of MUXes and DEMUXes stimulates the separation of multi purpose components such as Register File’s both write and read capabilities whilst some temporary registers help with preserving data between the states of the same instruction.

1. **Control Unit**



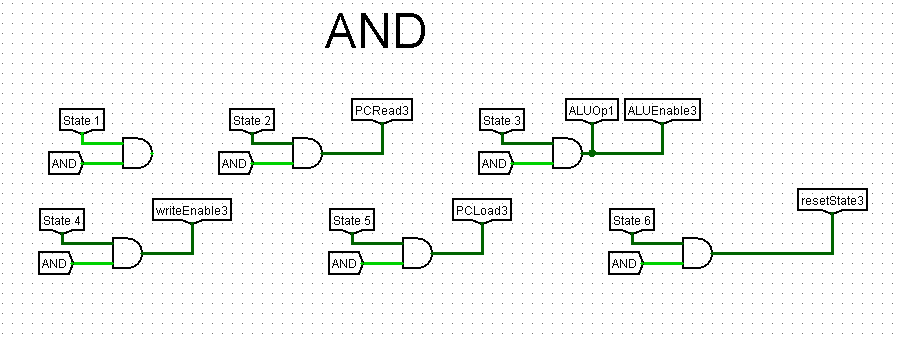
**Components Used:** Decoders, registers, clock, inputs, outputs, tunnels, OR gates, NOR gates, AND gates, NOT gates...

**Overview:** Takes 4 inputs, opcode representing the last 4 bits of the instruction, clock as in main circuit’s clock, ZF and CF for judging whether conditional jump operations are eligible depending on their requirements. Gives 15 outputs, in which 14 are signals for main circuit and the other one is a failsafe system to keep track of the state register.

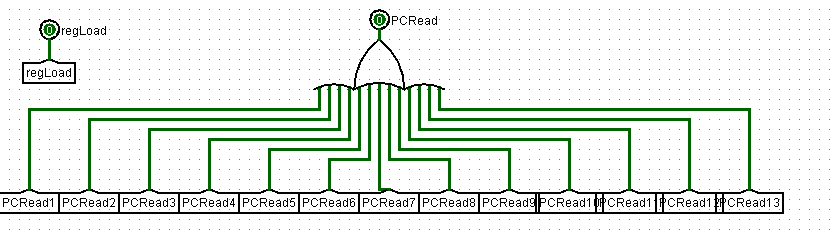


Register on top left keeps track of the state which the current clock cycle is at. Each time a clock rising edge is reached, register gets incremented and forces the instruction to move to the next state.

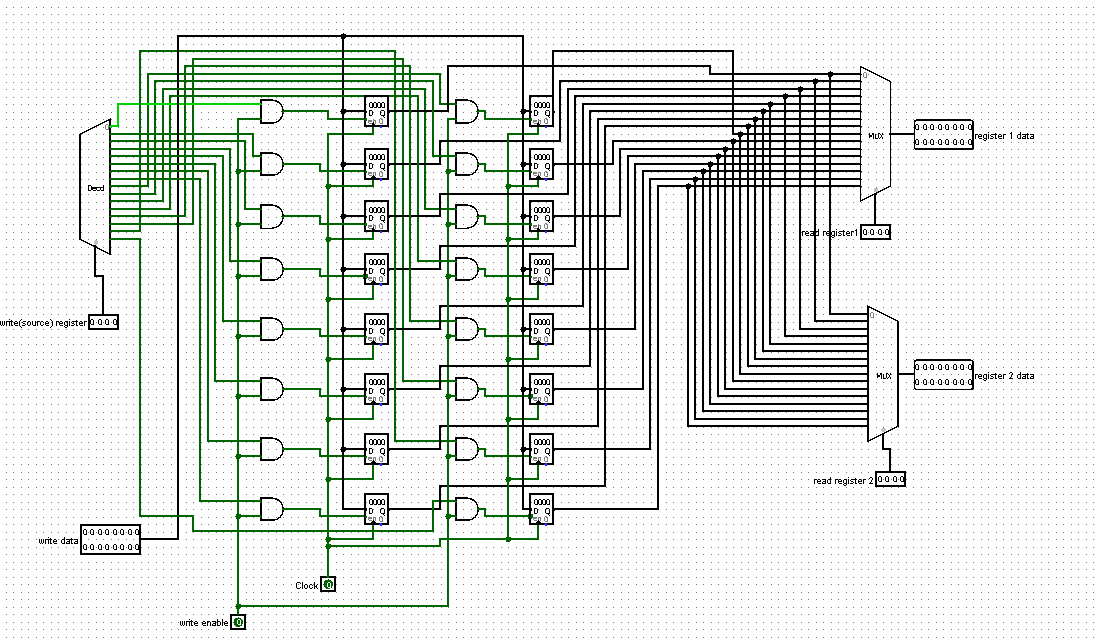
Opcode input on the bottom keeps track of the instruction currently in progress.



By using an AND gate, we managed to combine both instruction decoder and state decoder to produce right signals at the right time. State 1 is blank for any register updates to be conducted (especially needed after jump instructions). Special cases, signals and much are put into thought for creating the Control Unit as a finite state machine. As a result we have 14 signals spread into 13 uniquely implemented instructions. At the last state, resetState signal clears state register back to state 1 for the next instruction to take place.

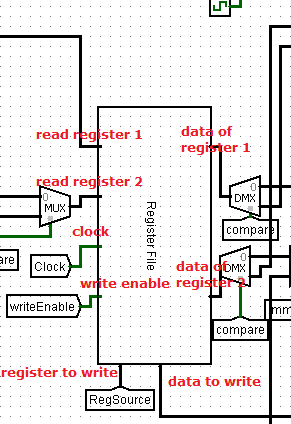


Final output signals are created by OR gates, as the same signal can be 1 and 0 at the same time coming from different instructions, therefore we numbered the tunnels uniquely for each instruction to create working signals.

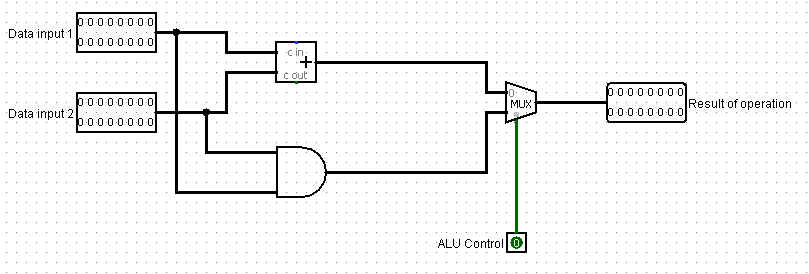
**c) Register File**

**Components Used**: Decoder, MUXes, inputs, outputs, registers, AND gates...

**Overview:** Has two tasks. Either reading data from registers (when write enable = 0) or write data to registers (write enable = 1). For reading, takes 2 registers addresses as inputs and puts their data to outputs. When write enable = 1, writes write data input to specified source register. Inputs and outputs of Register File are given in below.



1. **ALU**



**Components Used:** Inputs, outputs, adder, AND gate and MUX.

**Overview:** Takes two data as inputs and depending on the ALUop signal, either executes AND or Adder path to output the result.

**4) Verilog**

I (Ömer Faruk Çakı), was responsible from assembler converter and **Verilog**. I couldn't able to make anything regarding the Verilog. So, unfortunately this part is going to be empty due to lack of Verilog. However, my partner did everything which he was responsible for.