**CSE 315 DIGITAL LOGIC DESIGN TERM PROJECT**

**DESIGNING AND IMPLEMENTING A PROCESSOR**

***150116046 – Fatmanur Özdemir***

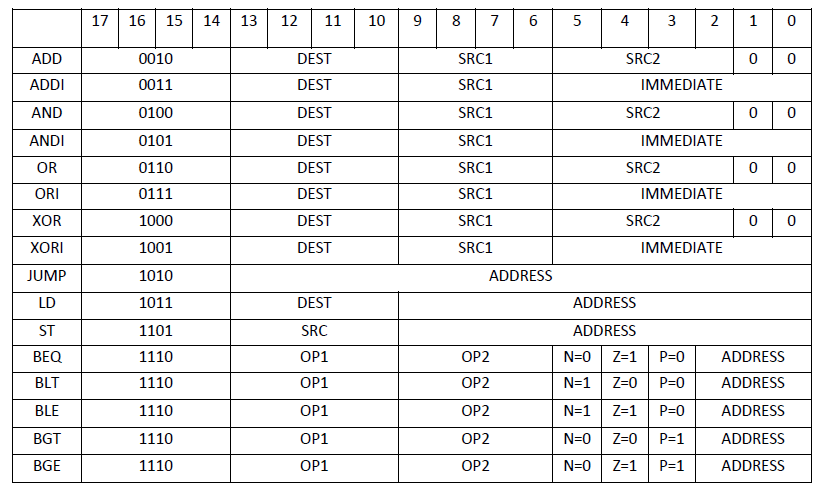
***150116048 – Kevser İldeş***

***150117011 – Buse Batman***

* In this project, our aim was designing and implementing a processor which supports these instruction sets:
* ADD, ADDI, AND, ANDI, OR, ORI, XOR, XORI, LD, ST, JUMP, BEQ, BGT, BLT, BGE, BLE
* Processor has 18 bits address width and 18 bits data width. And it has those 5 parts:

***Register File*** - ***Instruction Memory*** - ***Data Memory*** : (10 bits address width, 18 bits data width.) - ***Control Unit***  - ***Arithmetic Logic Unit*** (***ALU***)

**INSTRUCTION SET ARCHITECTURE – JAVA IMPLEMENTATION:**

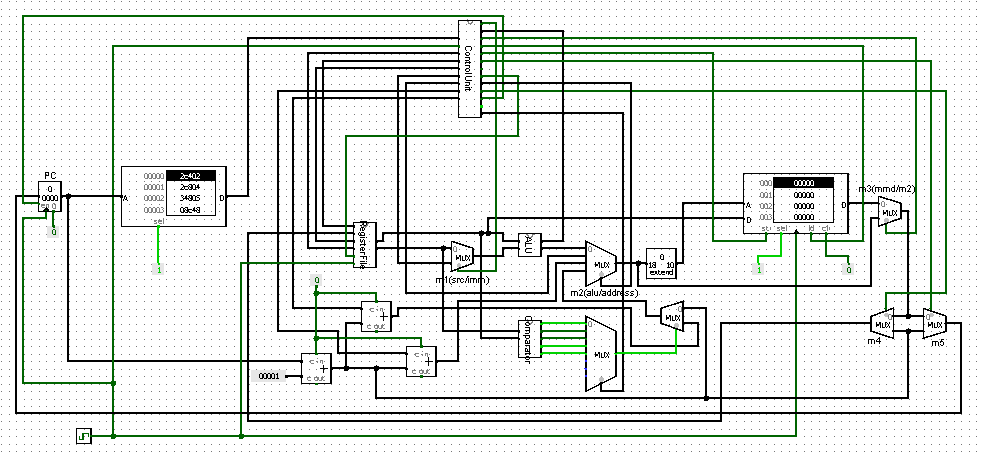
* In the ISA part, we gave unique 4 bit opcodes for the instructions. We started from 0010, because we gave 0000 to fetch1 and 0001 to fetch2 in the logisim design.
* Because we have only 16 registers, we used

4 bit for the registers.

* For ADD, AND, OR and XOR operations we used two 0 bits at the beginning of the instructions. Because instructions are 18 bits range and we used only 16 bits of them. So, to complete them to 18, two bits are unused 0’s.
* For the branch operations, we used nzp bits. Greater than makes p 1, less than makes n 1 and equal makes z 1.
* After designing the instruction set architecture, we implemented a java code to take the instructions and values from the user and converted them into a hexadecimal value. It also adds “v2.0 raw” string to beginning of the output.txt file.

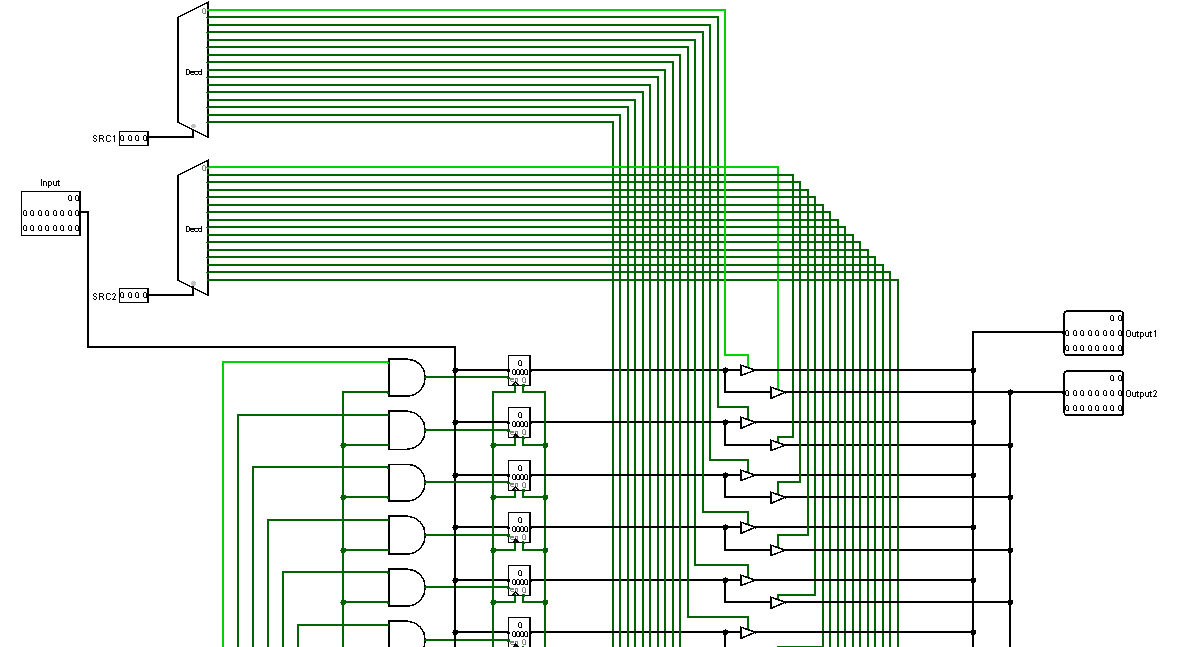
**PHYSICAL IMPLEMENTATION - LOGISIM DESIGN:**

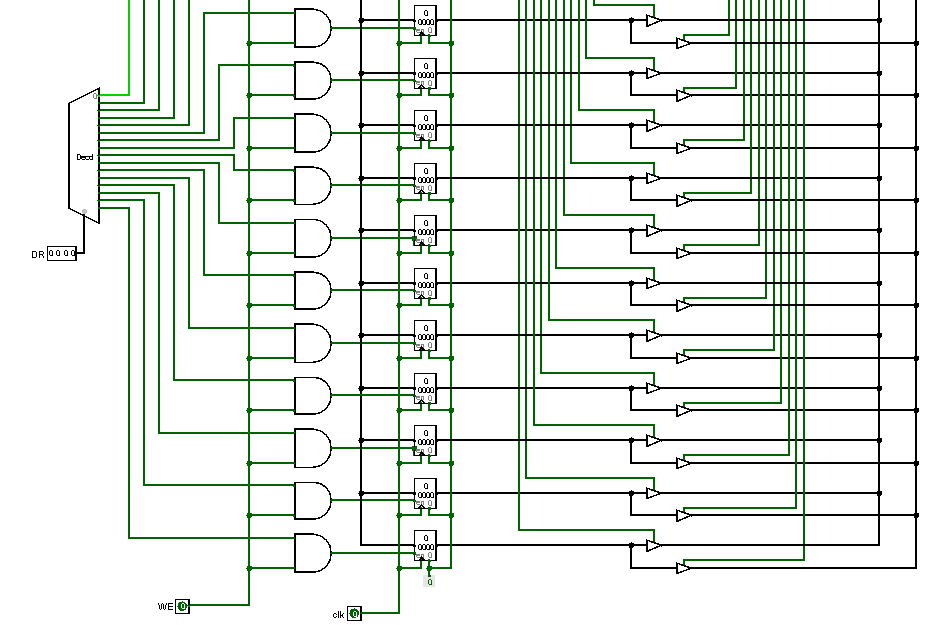
**Main:**

****

Main circuit consists a program counter, instruction memory, data memory, registers, control unit, arithmetical logic unit, multiplexers, adders, comparator, bit extender and clock. With the clock signal it reads the instructions from the instruction memory, sends it to the control unit. After splitting the instruction and setting the signal values, control unit sends this output values to the arithmetical logical unit, program counter or to the memory depending the need of the state.

**Regfile:**

****

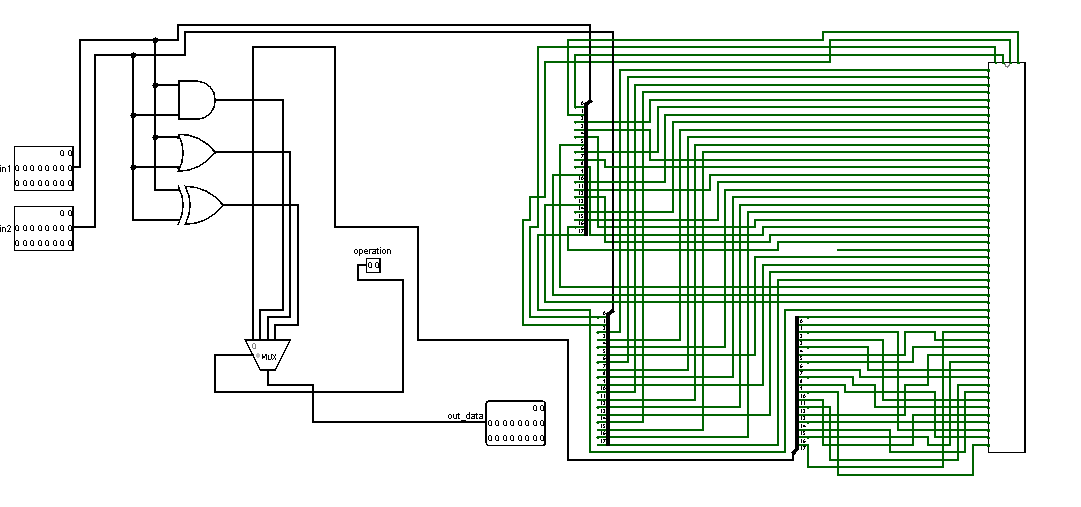
****

Register file has two tasks: When the we(write enable) bit is 0, it reads data from registers, otherwise it writes data to the registers.

For the read operation, it takes 2 register addresses as inputs (src1 and src2) and puts the data to the output.

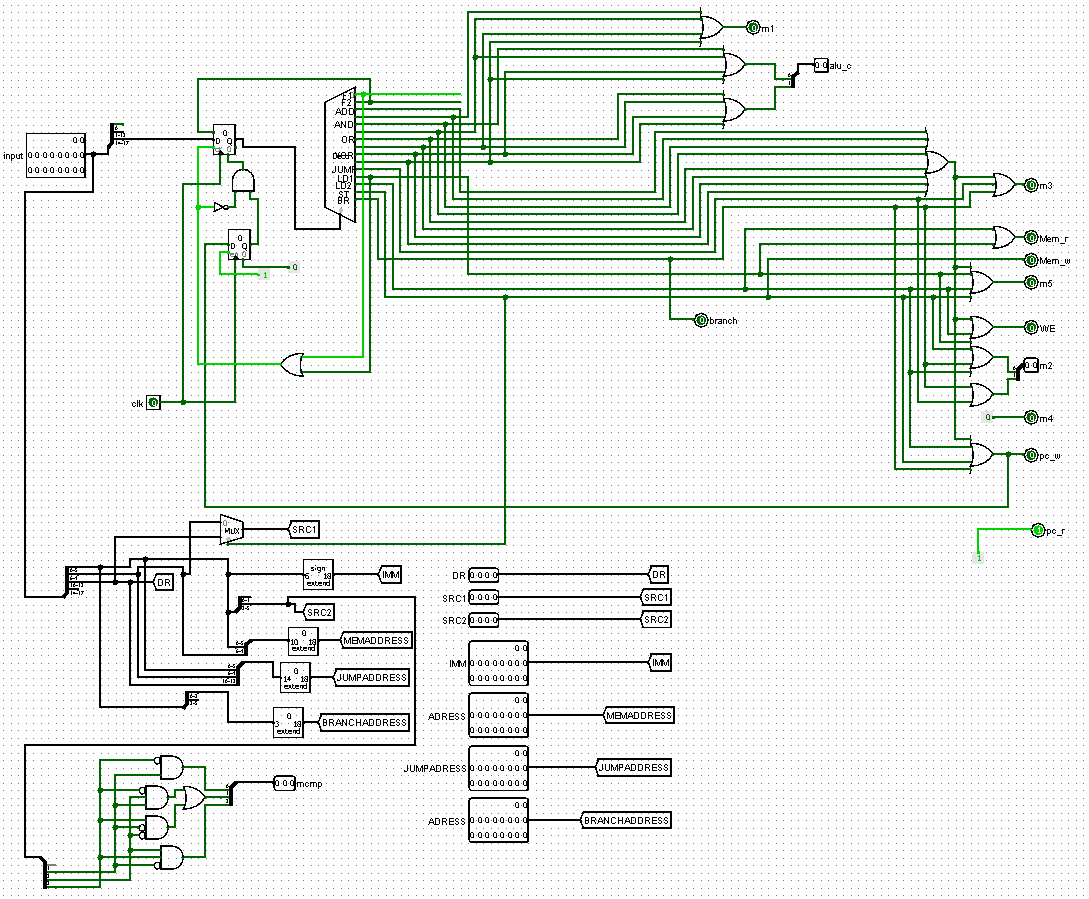
And for the write operation, it writes data input to source register.

**Arithmetical Logical Unit (ALU):**

****

Arithmetical logical unit takes two data as inputs and depending on the operation signal, it executes add, and, or, xor operations and outputs the result.

**Control Unit:**

****

Control unit takes the 18 bit input and splits these input to understand the opcode, register values, immediate values and address values. For every operation, it splits to all even not needed. But it only uses the necessary ones for that instruction.

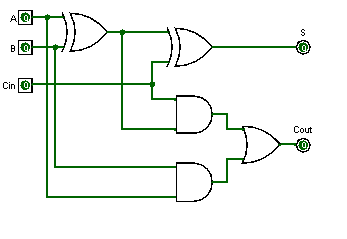
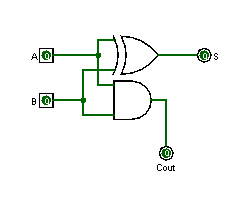
With this split operation we found the instructions and with using tunnels we had a more readable design.

Control unit is designed with finite state machine model. Every instruction (except the load instruction) has 3 states (Fetch 1, Fetch 2 and the instruction –opcode-) (load has 4 states (Fetch 1, Fetch 2, Load 1 and Load 2).

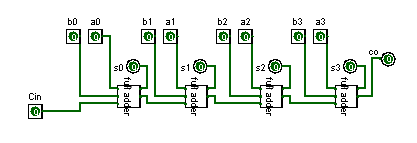
Right after we found the instructions, we setted the signal values which the instructions should have at that state. And we output those signals. Arithmetical and logical control bits (2 bit) is setted if that state needs add,and,or,xor operations; memory write bit is setted at the end of the store instructions; program counter write bit is setted at the final state of the jump and branch instructions.

**Half Adder – Full Adder:**

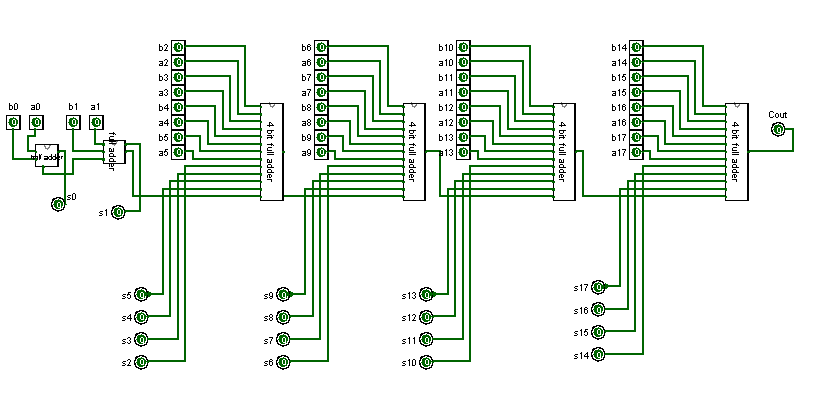
In this project, we had to design our own adders. So we designed a half adder, full adder and using those adders we designed 4 bit and then 18 bits adder (1 half adder - 1 full adder – 4 \* 4 bit full adder).

****

**4 bit adder:**

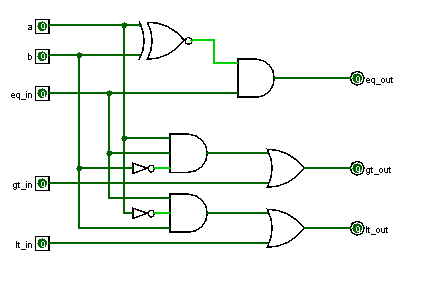
****

**18 bit adder:**

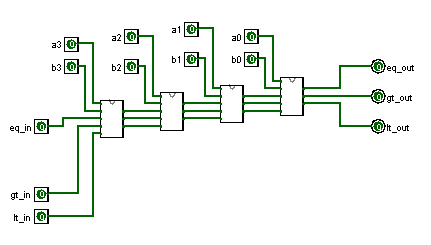
****

**1 bit comparator:**

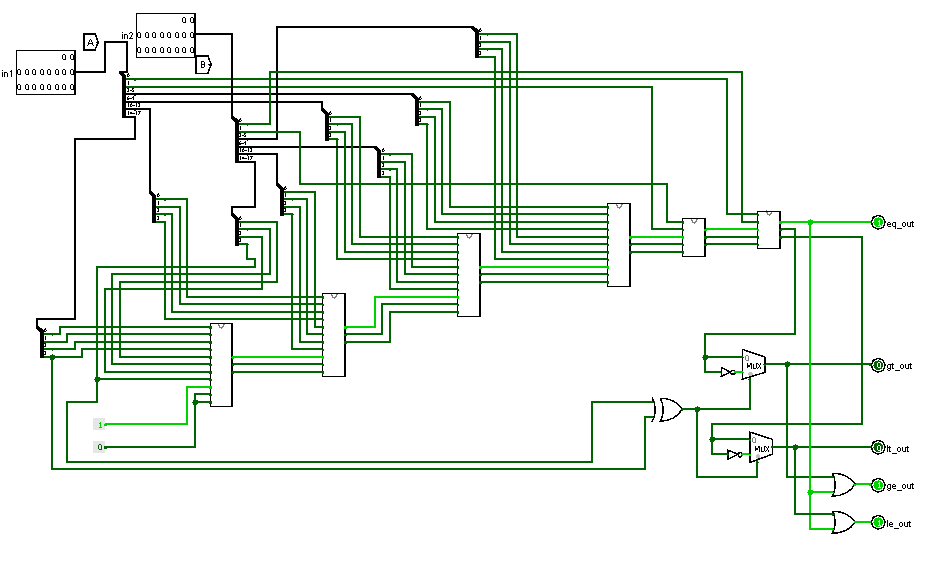
We also had to design comparator starting from the gates. Beginning with 1 bit comparator, we designed 4 bit comparator and using these, we designed 18 bit comparator (4 \* 4 bit comparator + 2\* 1 bit comparator).

****

**4bit comp:**

****

**18 bit comparator:**

****