**18 Bit Processor**

*Authors: Mahmut Salman*

**Introduction**

In this project, we designed a processor with 18 bit address width, 18 bit data width, 16 registers and it can 16 distinct instructions. They will be shown in the ISA part in the report.

Our processor supports the instruction set architecture with instructions: AND, OR, ADD, LD, ST, ANDI, ORI, ADDI, XOR, XORI, JUMP, BEQ, BGT, BLT, BGE, BL.

A line of instruction should be in the following formats

ADD R1,R2,R3

ADDI R1,R2,1

AND R1,R2,R3

ANDI R1,R2,1

OR R1,R2,R3

ORI R1,R2,1

XOR R1,R2,R3

XORI R1,R2,1

ST R1,0

LD R1,0

JUMP 0

BEQ R1,R2,0

BGT R1,R2,0

BLT R1,R2,0

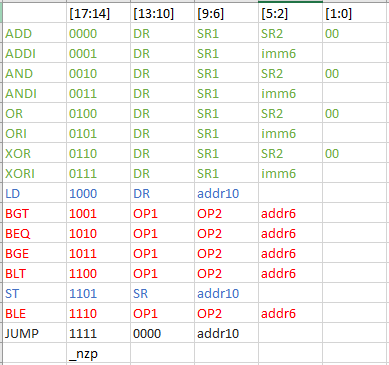
BGE R1,R2,0

BL R1,R2,0

(Registers and the values were shown arbitrarily as an example.)

**Part 1: Instruction Set Architecture (ISA)**

We designed an ISA in order our processor to understand our human language and interpret it in a low level language. Our assembler code converts the instructions in the ISA into a low level representation that our processor can understand.



This ISA reserves the first 4 bits for the instruction type, bits [13:10], [9:6], [5:2] for destination and source registers (mostly).

For ALU operations, not immediate instructions has two non-used bits in the end as ‘00’. Immediate operations has immediate value with 6 data bits in the right most bits.

LOAD and STORE operations use the [13:10] bits as the register and the [9:0] bits as the address to be loaded or stored.

JUMP instruction jumps ADDR times in the instruction memory. ADDR will be offset and in PC relative mode.

BRANCH instructions compares two registers, and if the conditions are met, jumps to the specified address in the instruction memory. Comparison is made by the n,z,p flags which are embedded in the instruction opcode as the last 3 bits.

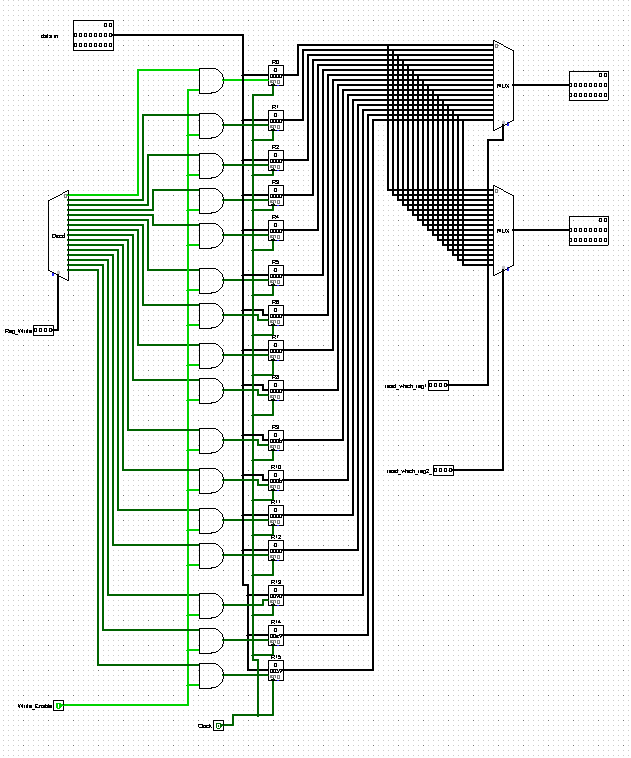
We programmed a code in order to convert high level instructions into 18 bit binary representations outputed as hexadecimal numbers according to our ISA.

Example conversion:

 : 

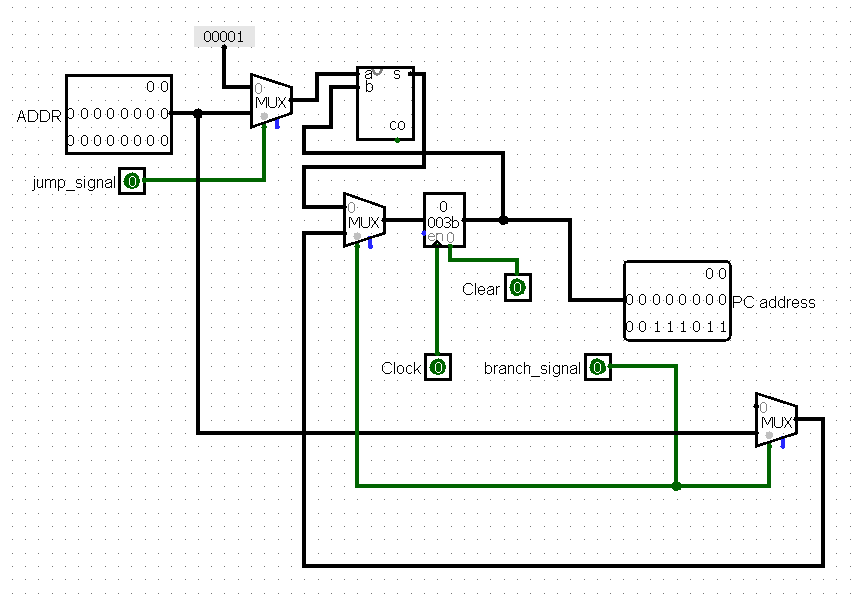
**Part 2: Component Design**

**Register File**



Register file holds all 16 registers, can write 18 bit data to selected registers, can read data from 2 selected registers. It has write enable input for load operation, shared clock input.

**Program Counter**



Program counter determines the instruction address in the instruction memory. Ideally starts from 0, and increments in each clock.

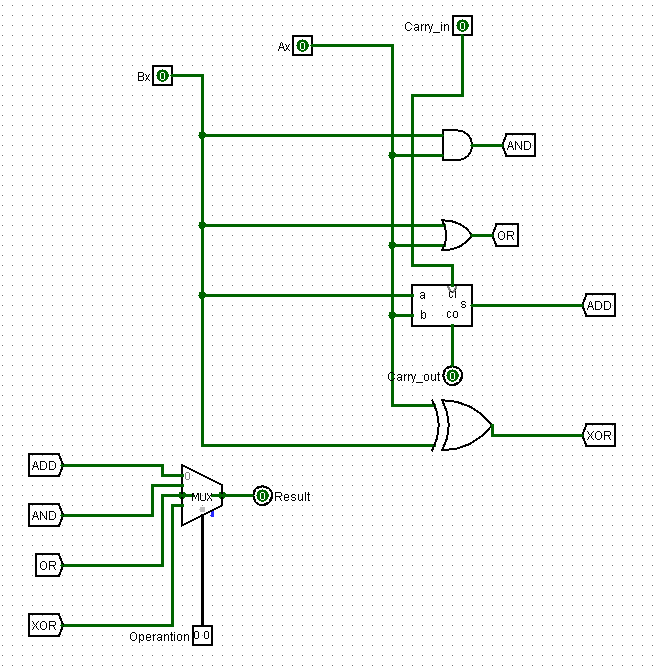
It has 3 mods. Increment, jump and branch.

Increment: No signals are received, the data in the register is incremented by 1 and PC address will be updated.

Jump: Jump signal is received. Since the jump address is an offset, the input ADDR is added to the PC address with the adder.

Branch: Branch signal is received. Input ADDR is directly written to the register and sent to the output PC address.

**1 Bit ALU (Arithmetic Logic Unit)**

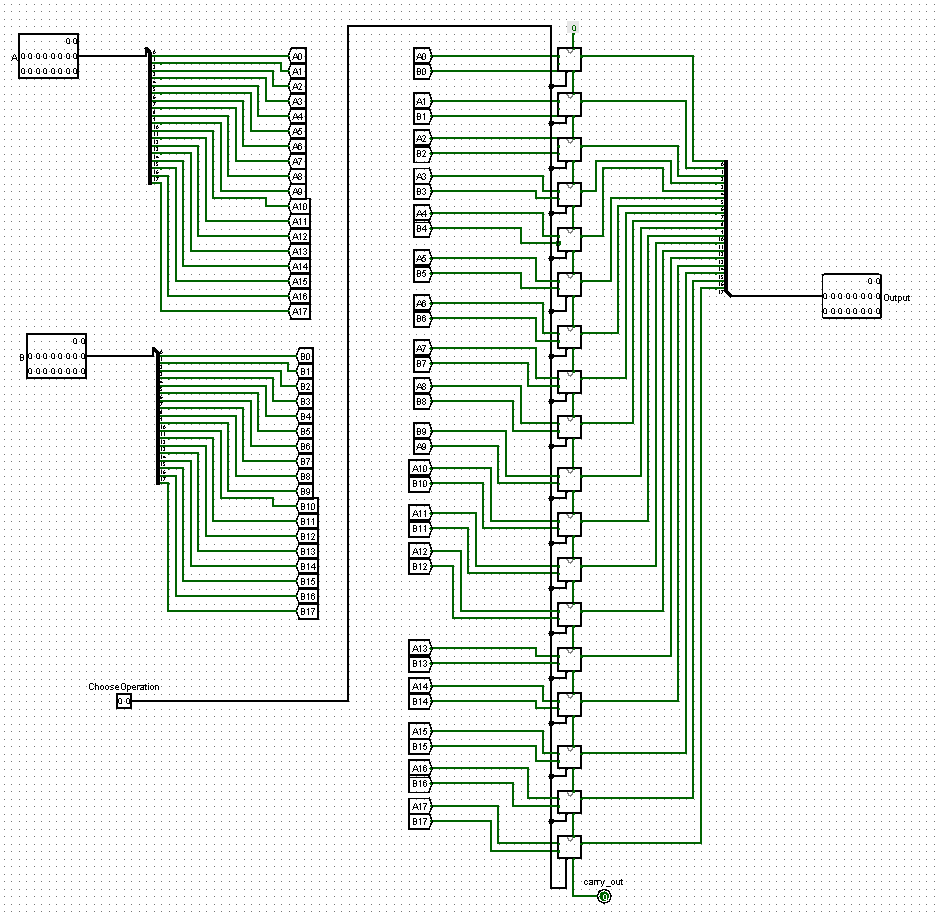


Our ALU calculates 4 operations: AND, OR, ADD and XOR. Takes 3 inputs: operand1, operand2 and operation code.

Operation code is derived from the instruction set architecture. We designed our ISA such that the middle 2 bits of our 4 bits opcode determines the operation type. For instance first 4 bits of our instruction is 0110. The middle two bits are 11 which corresponds to XOR operation. Other operation codes are: 00:ADD, 01:AND, 10:OR, 11:XOR.

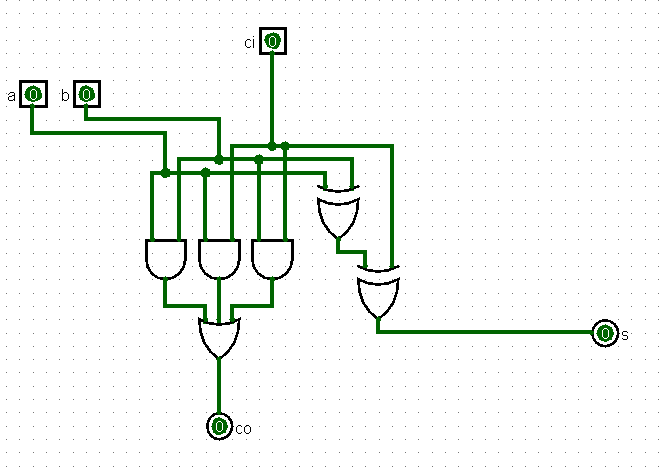
ALU calculates all possible operations and then picks one with MUX according to the operation code.

**18 Bit ALU (Arithmetic Logic Unit)**



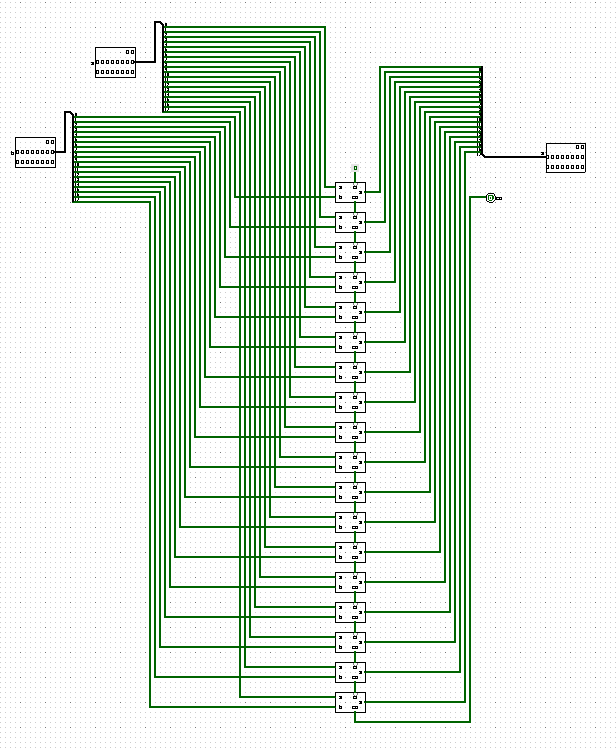
It is basically the combination of 18 1-bit ALU’s. Operates two operands’ bits one by one, considers carry bit if needed and derives an output.

**1 Bit Adder**



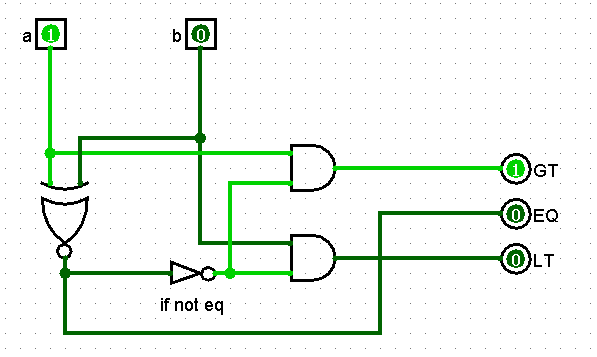
Our 1 bit adder is simple as it is generic. Takes 2 operands and a carry in, produces a sum and a carry out.

**18 Bit Adder**



Our 18 bit adder is a combination of 18 1-bit adders. Operates each operands’ bits one by one and produces a sum and a carry out.

**1 Bit Comparator**

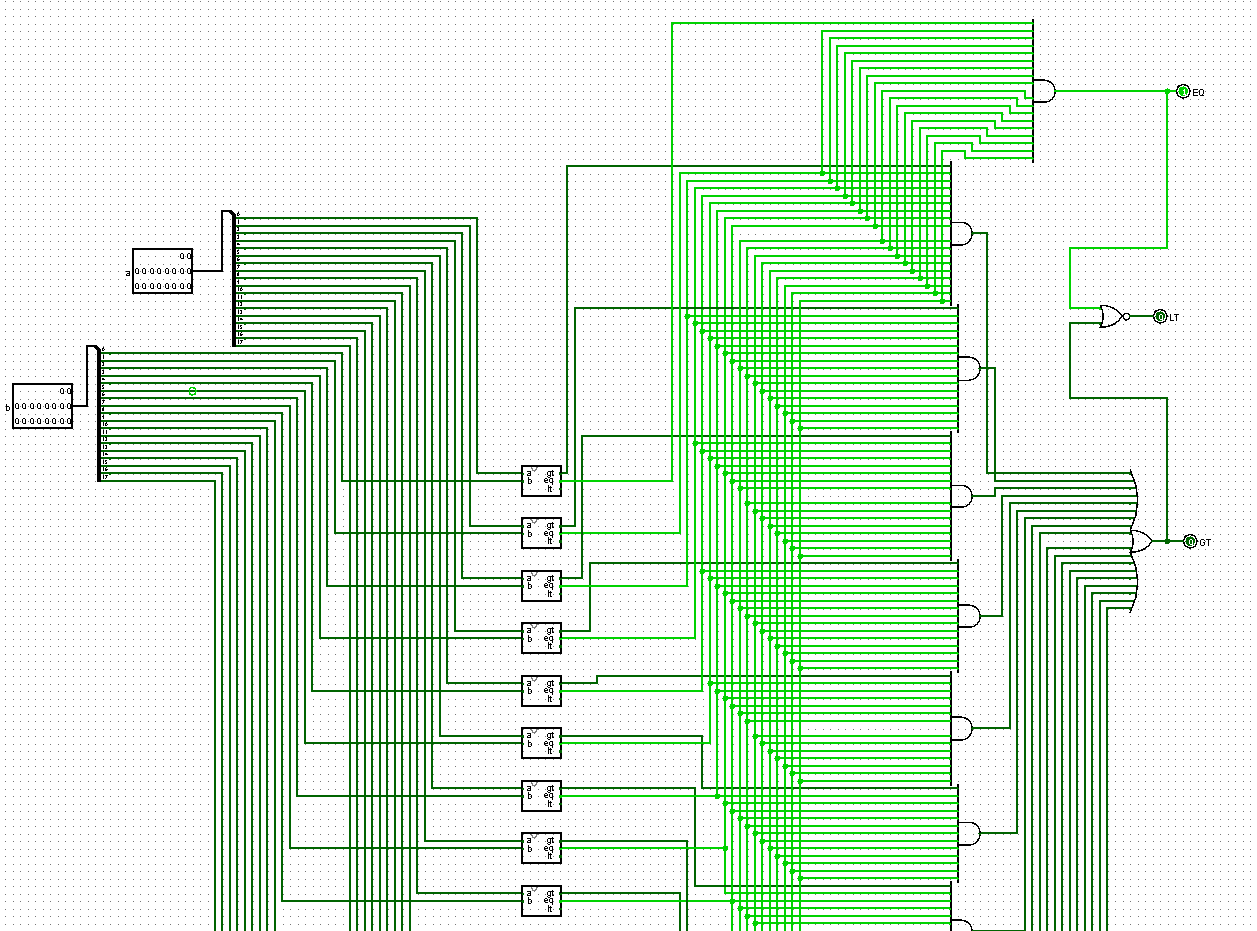


Our comparator is simple and generic. Takes 2 inputs as operands, checks equality with an XNOR gate.

If not equal cable is connected with both operands seperately with and gates, only one of them will give 1 since they are not the same.

It produces 3 inputs, greater, equal and less than signals.

**18 Bit Comparator**



The whole component didn’t fit in the screenshot, please see the full component in our logisim file.

It is basically a combination of 18 1 bit comparators. The mechanism works as follows:

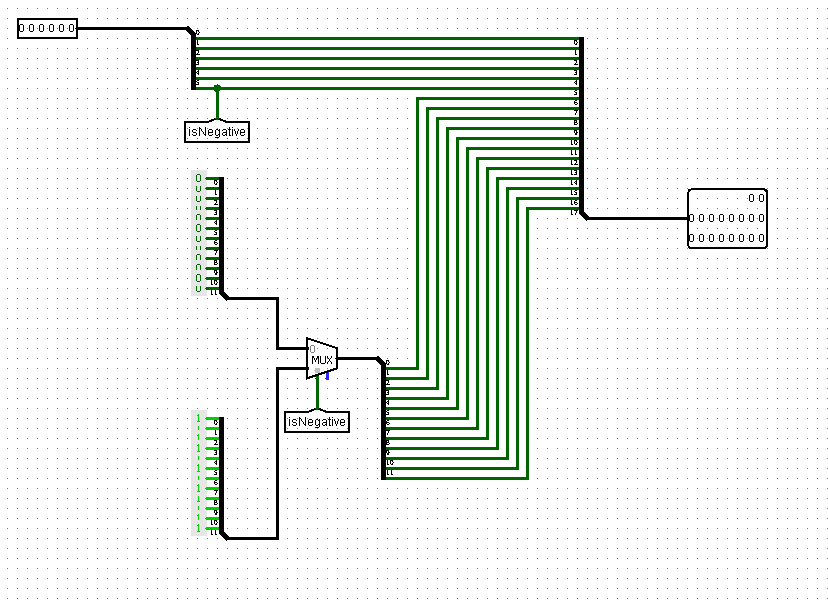
-Start from the most significant bit

-Check if they are not equal, if so then send a GT or LT signal

-If equal, then don’t send signal yet and proceed.

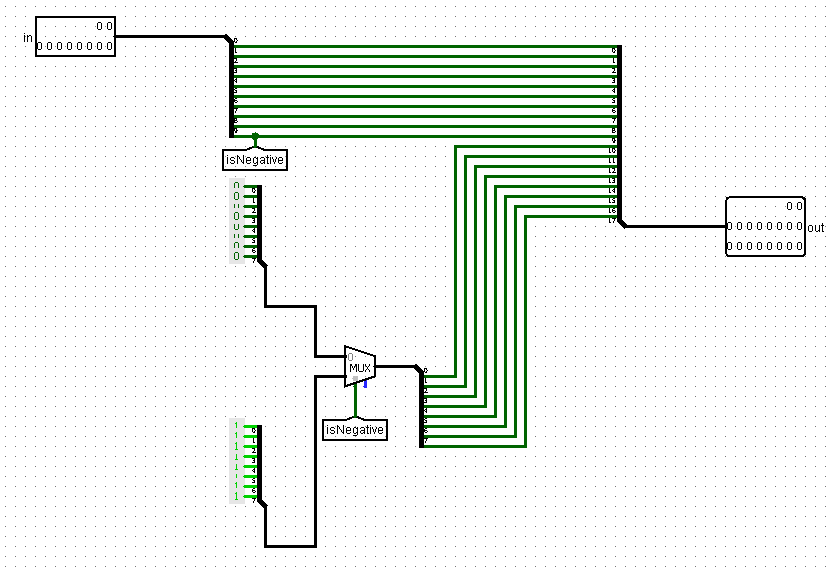
We implemented this mechanism by AND gating each bit with all previous compared bits’ equality signals.

**Extender 6-18 bits**



It is simple and generic. Checks if the input is negative or not and adds 12 1’s or 0’s accordingly.

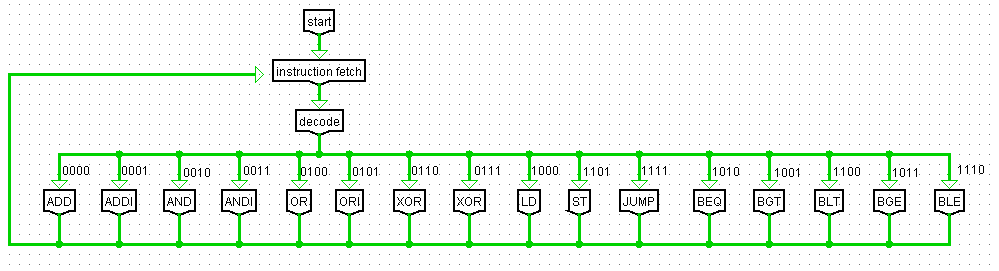
**Extender 10-18 bits**



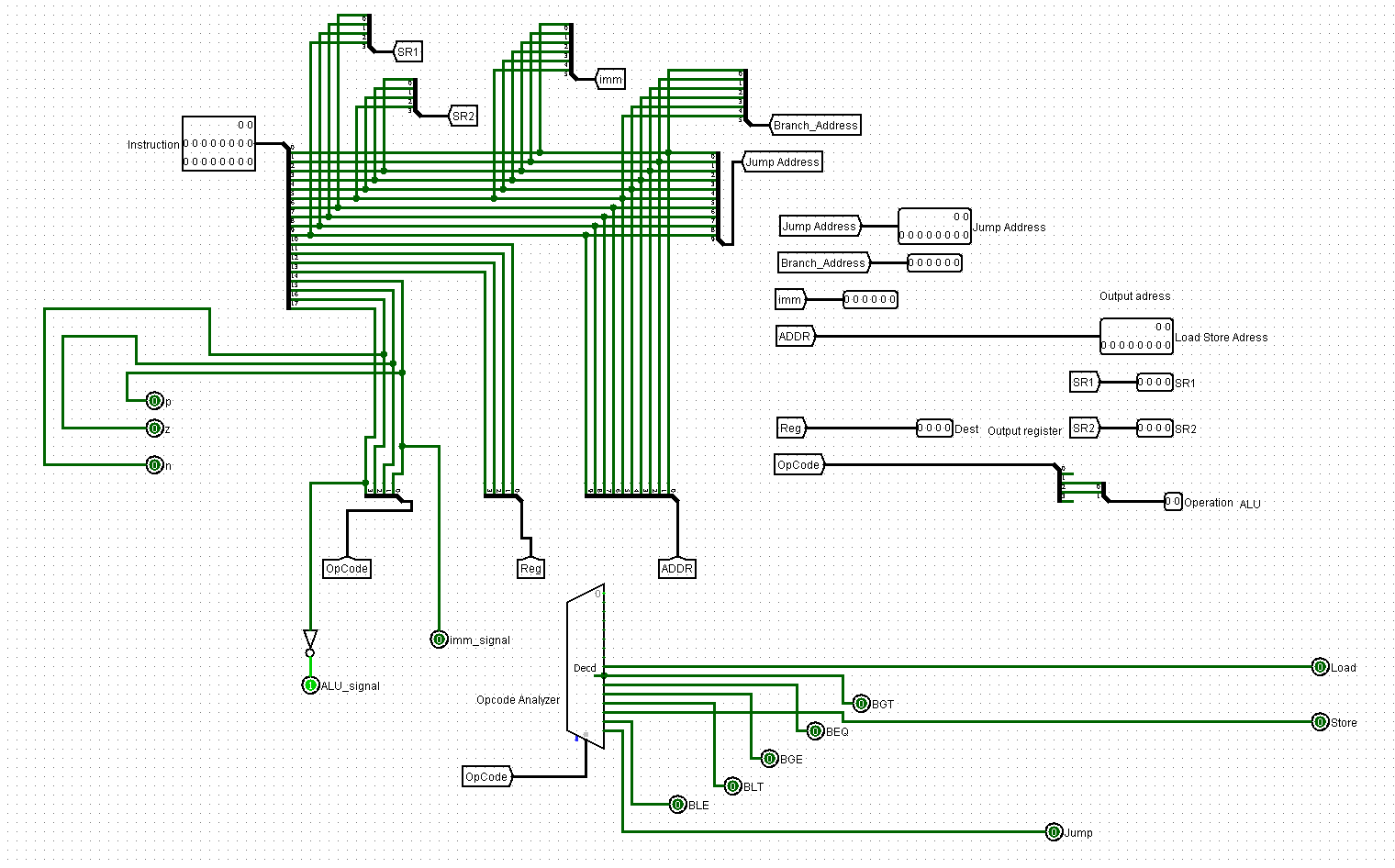
It is very similar to our 6-18 extender.

**Part 3: Control Unit**

**Finite State Machine of Control Unit**

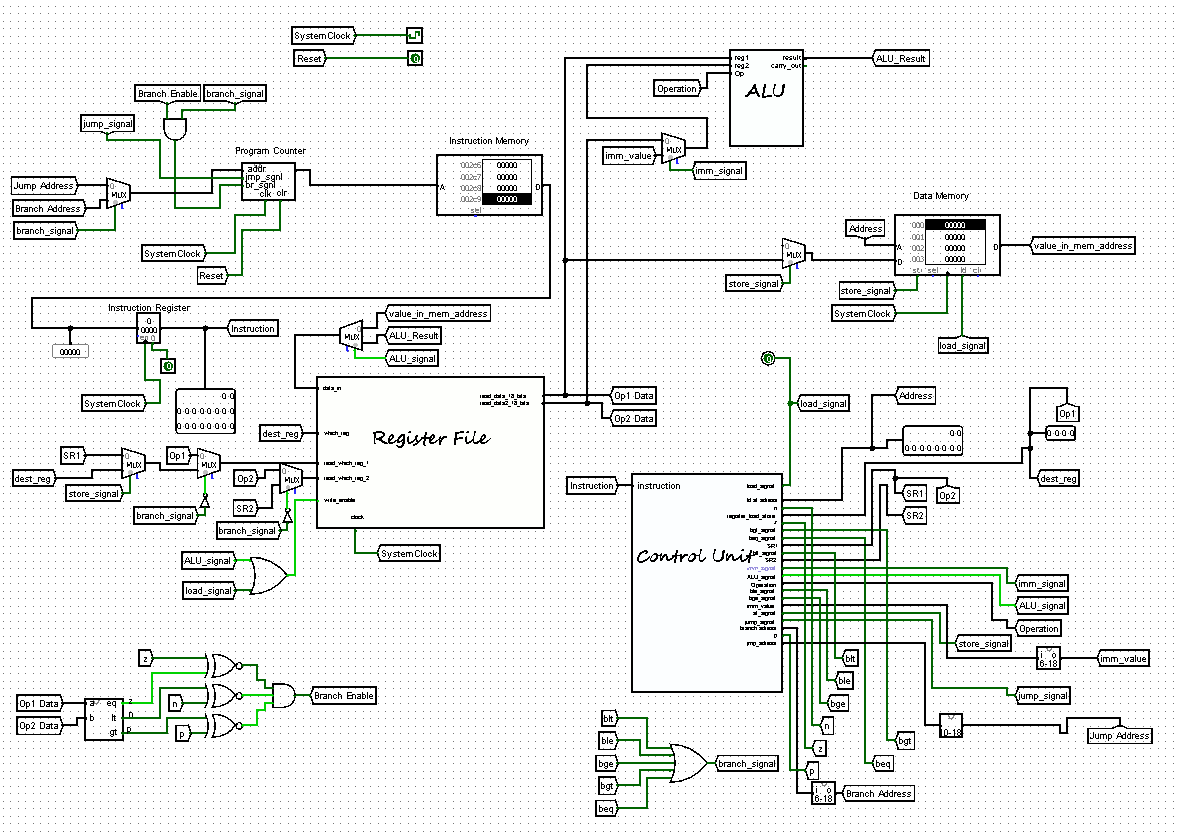


**Control Unit**



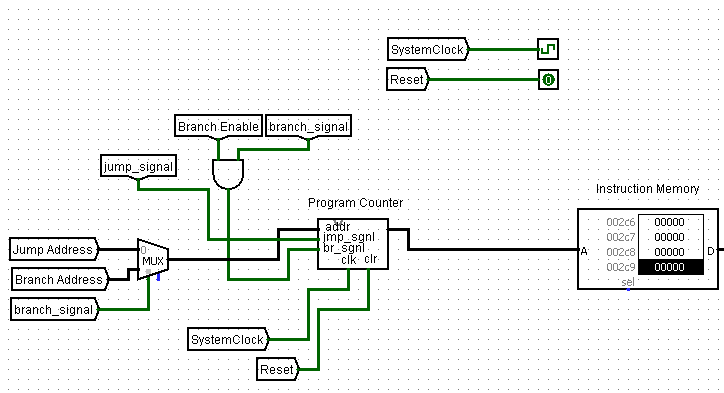
Control unit takes the 18 bit instruction as input and produces proper signals. Determines the opcode by the first bits, produces nzp signals from last 3 bits in opcode, takes the registers from according bit intervals, takes addresses, determines the ALU operation and calculates all the needed infos and gives their output.

**Processor (main unit)**

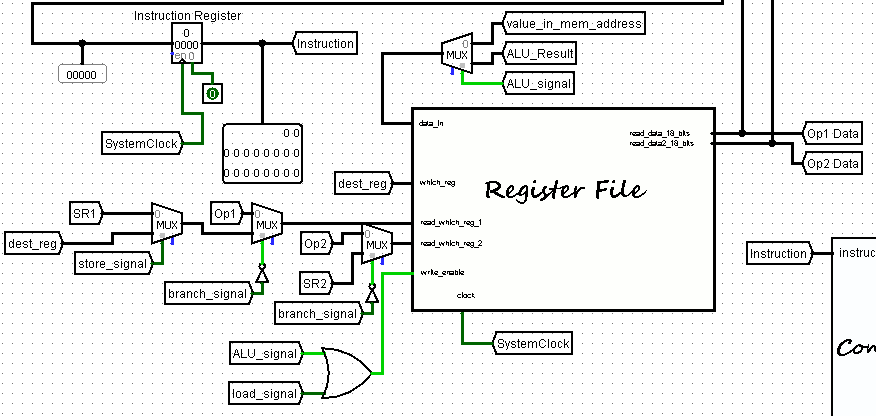


Ideally program starts with fetching the instruction from the 0th address of the instruction memory. Than the processor proceeds to process the instruction in control unit, generate the proper signals.

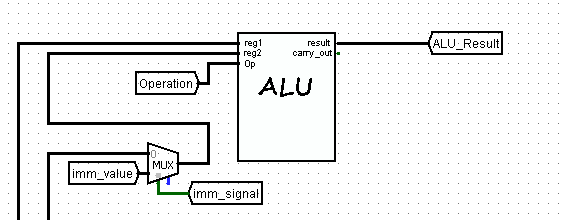
With the produced signals, our processor can determine what to do next.



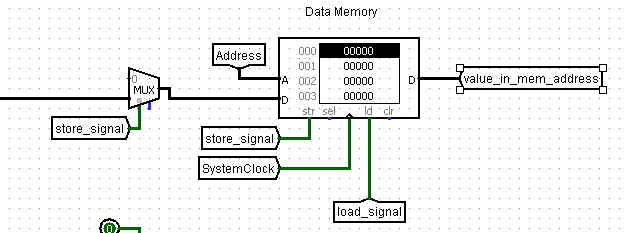
In this part, counter mod is determined. If default, increment 1. If jump, use the address as offset. If branch, jump to the address.



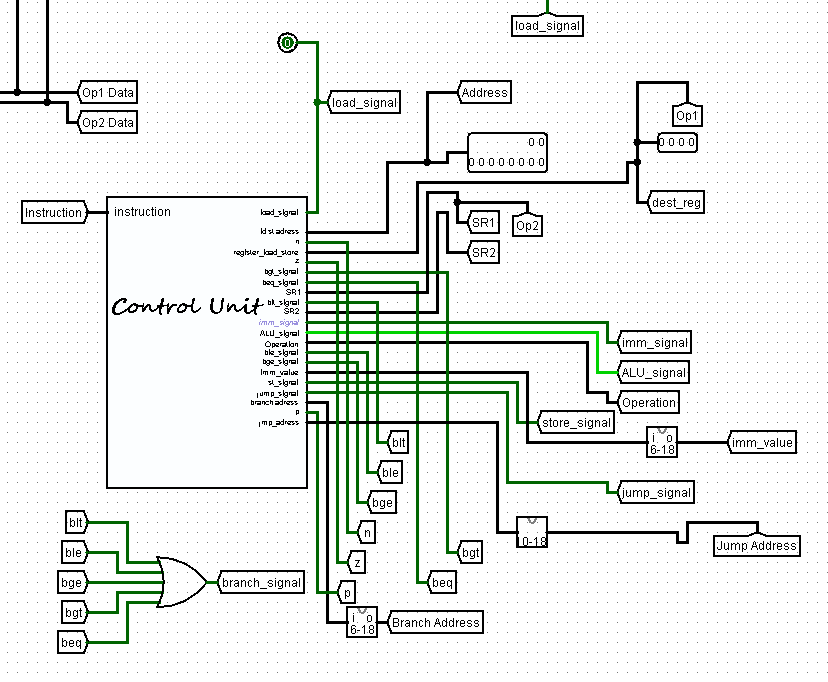
In the register part, write enable is on when ALU or LOAD signals are on. Determines the data in from the ALU signal, otherwise retrieves data from the RAM (value\_in\_mem\_address). Register file takes 2 input registers, chosen with specific conditions according to our design of control unit. All of the MUX distinctions and selections of registers are related to our instruction structure design.



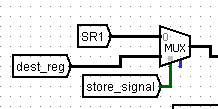
In this part, data from registers are sent to ALU, if immediate signal is on, retrieve the immediate value instead as operand2.



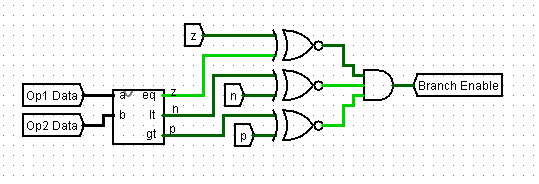
Here is the RAM, where Address tunnel is the load address, the cable goes into the D holds the data in reg1 output of the register file.



This is the control unit, producing proper signals for other components. As seen in the bottom left, branch signal is derived from or gating the branch signals individually. Store register is named as dest\_reg here, not named with a different tag. This is handled in the register file part as this:



Picks dest\_reg if store signal is on.



There are 2 branch related signals in the processor. One is ‘branch\_signal’ and the other one is ‘Branch Enable’.

Branch signal checks if the instruction starts with BEQ, BGE, BLE, BGT or BLT.

Branch enable checks if the branching conditions are met after comparing given 2 registers.