DIGITAL LOGIC DESIGN REPORT

ERTUĞRUL SADIÇ 150116061

GÖKSEL TOKUR 150116049

MUSTAFA FURKAN ARAS 150116019

About:

In this project, we are supposed to design a basic CPU which has the 14 bit address size, 18 bit data size, 16 registers and supports instructions AND, ANDI, ADD, ADDI, OR, ORI, XOR, XORI, LD, ST, JUMP, BEQ, BGT, BLT, BGE, BLE.

* ADD instruction will add two register, and store result into another register.Structure of instruction is:

**ADD DST,SRC1,SRC2**

where SRC1 and SRC2 are source registers, and DST is destination register for the operation.

* ADDI instruction will add a register value and immediate value, and store the result into another register.Form of instruction is:

**ADDI DST,SRC1,IMM**

where SRC1 is a register, DST is destination register and IMM is immediate value. IMM size will be max available size on your processors design.

* JUMP instruction will set the Program Counter(PC will hold current instruction’s address) to the given value in the instruction.

**JUMP ADDR**

where ADDR will be in PC relative mode. ADDR will be offset and it can be negative.

* LD instruction will load a value from Data Memory to any register.

**LD DST,ADDR**

where DST is a register to load and ADDR is a address in max available bit size. Upper bits of ADDR will be zero extended.

* ST instruction will store value from a register to Data Memory.

**ST SRC,ADDR**

where SRC is a register to fetch data and ADDR is a Data Memory address to store content of the register. Upper bits of ADDR will be zero extended.

* Branch instruction will compare two operands, then will jump to the address according to this comparison. Your instruction set architecture must reserve 3 bit for branch instructions named as n,z,p which are negative,zero and positive.

**BEQ OP1,OP2,ADDR**

will compare registers OP1 and OP2 if they are equal, PC will be set to ADDR(PCrelative). Instructions n,z,p binary values will be 0,1,0.

**BLT OP1,OP2,ADDR**

will compare registers OP1 and OP2, if OP1 is less than OP2, PC will be set to ADDR(PC-relative). Instructions n,z,p binary values will be 1,0,0.

**BGT OP1,OP2,ADDR**

will compare registers OP1 and OP2, if OP1 is greater than OP2, PC will be set to ADDR(PC-relative). Instructions n,z,p binary values will be 0,0,1.

**BLE OP1,OP2,ADDR**

will compare registers OP1 and OP2, if OP1 is less than or equal to OP2, PC will be set to ADDR(PC-relative). Instructions n,z,p binary values will be 1,1,0.

**BGE OP1,OP2,ADDR**

will compare registers OP1 and OP2, if OP1 is greater than or equal to OP2, PC will be set to ADDR(PC-relative). Instructions n,z,p binary values will be 0,1,1.

Assembler:

we know that we have to design an instruction set architecture for all the instructions in order to make computer understand our instructions. First we designed an ISA, then we wrote an assembler according to ISA for converting these instructions to the machine code that our CPU will understand. We used the java programming language for its easy usage.

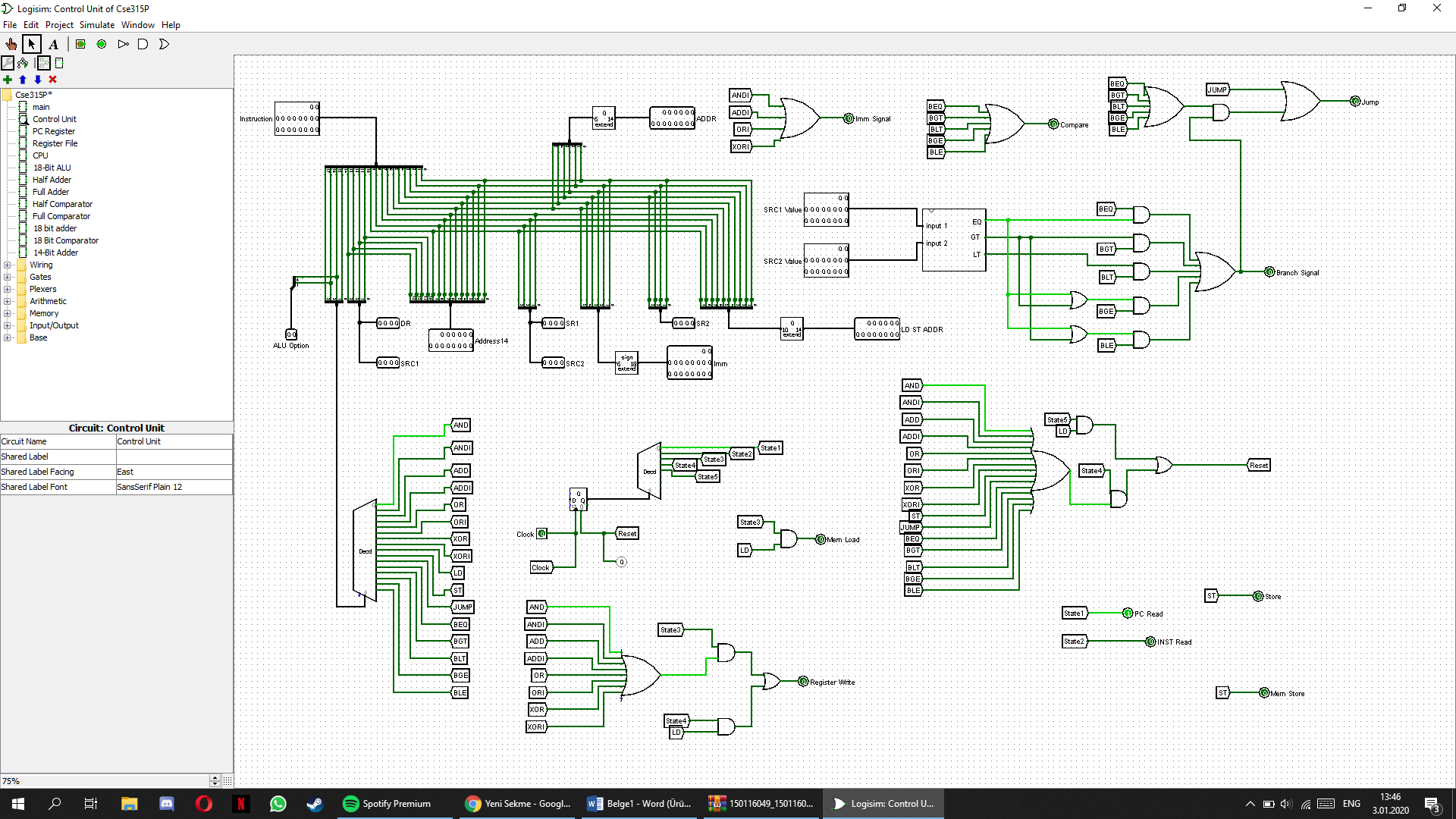
ISA:

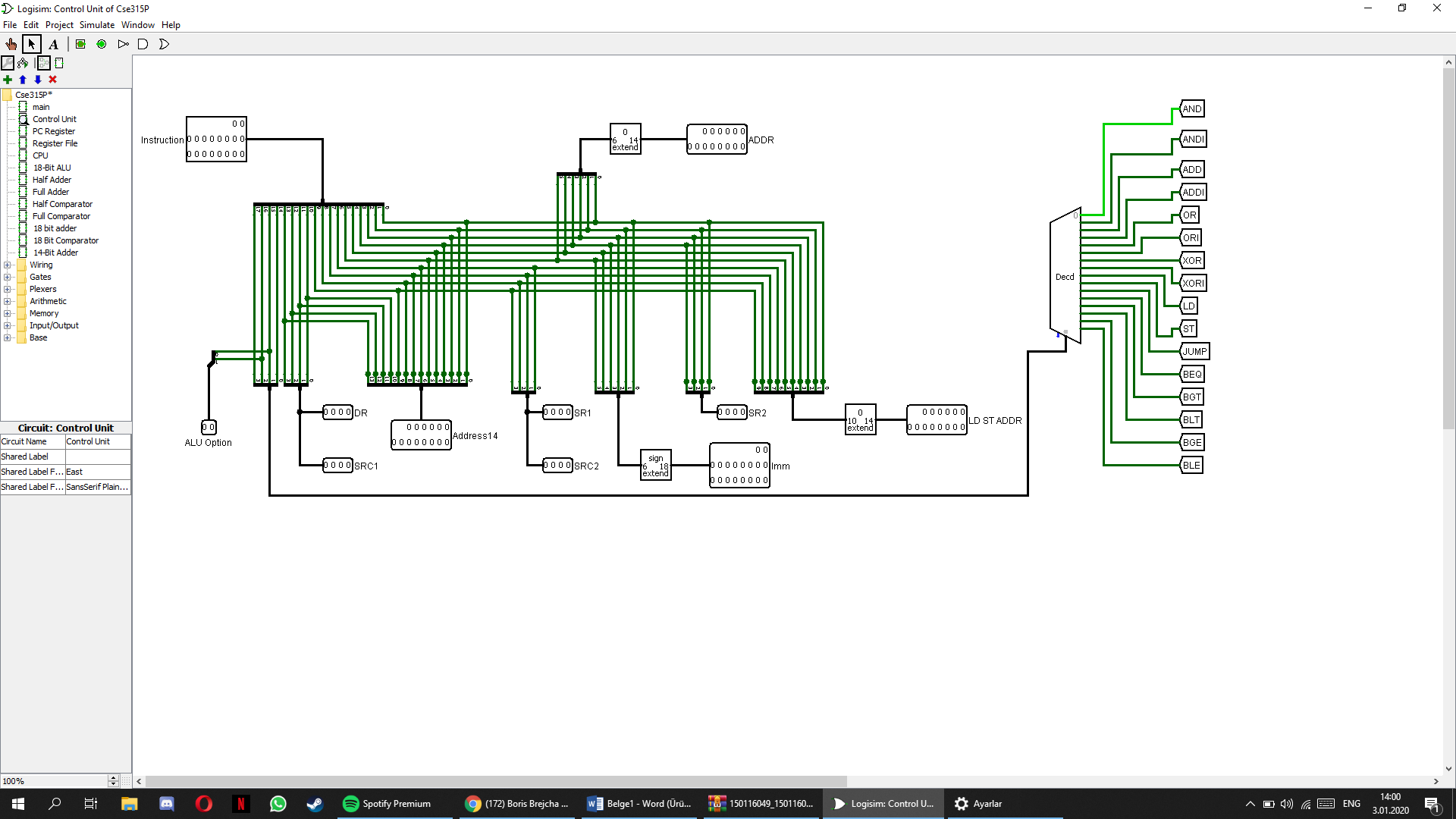
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Opcode[17:14] | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| AND | 0000 | DR | | | | SR1 | | | | 0 | 0 | SR2 | | | |
| ANDI | 0001 | DR | | | | SR1 | | | | 1 | imm5 | | | | |
| ADD | 0010 | DR | | | | SR1 | | | | 0 | 0 | SR2 | | | |
| ADDI | 0011 | DR | | | | SR1 | | | | 1 | imm5 | | | | |
| OR | 0100 | DR | | | | SR1 | | | | 0 | 0 | SR2 | | | |
| ORI | 0101 | DR | | | | SR1 | | | | 1 | imm5 | | | | |
| XOR | 0110 | DR | | | | SR1 | | | | 0 | 0 | SR2 | | | |
| XORI | 0111 | DR | | | | SR1 | | | | 0 | imm5 | | | | |
| LD | 1000 | DR | | | | Address10 | | | | | | | | | |
| ST | 1001 | SR | | | | Address10 | | | | | | | | | |
| JUMP | 1010 | Address14 | | | | | | | | | | | | | |
| BEQ | 1011 | SR1 | | | | SR2 | | | | ADDR | | | n | z | p |
| BGT | 1100 | SR1 | | | | SR2 | | | | ADDR | | | n | z | p |
| BLT | 1101 | SR1 | | | | SR2 | | | | ADDR | | | n | z | p |
| BGE | 1110 | SR1 | | | | SR2 | | | | ADDR | | | n | z | p |
| BLE | 1111 | SR1 | | | | SR2 | | | | ADDR | | | n | z | p |

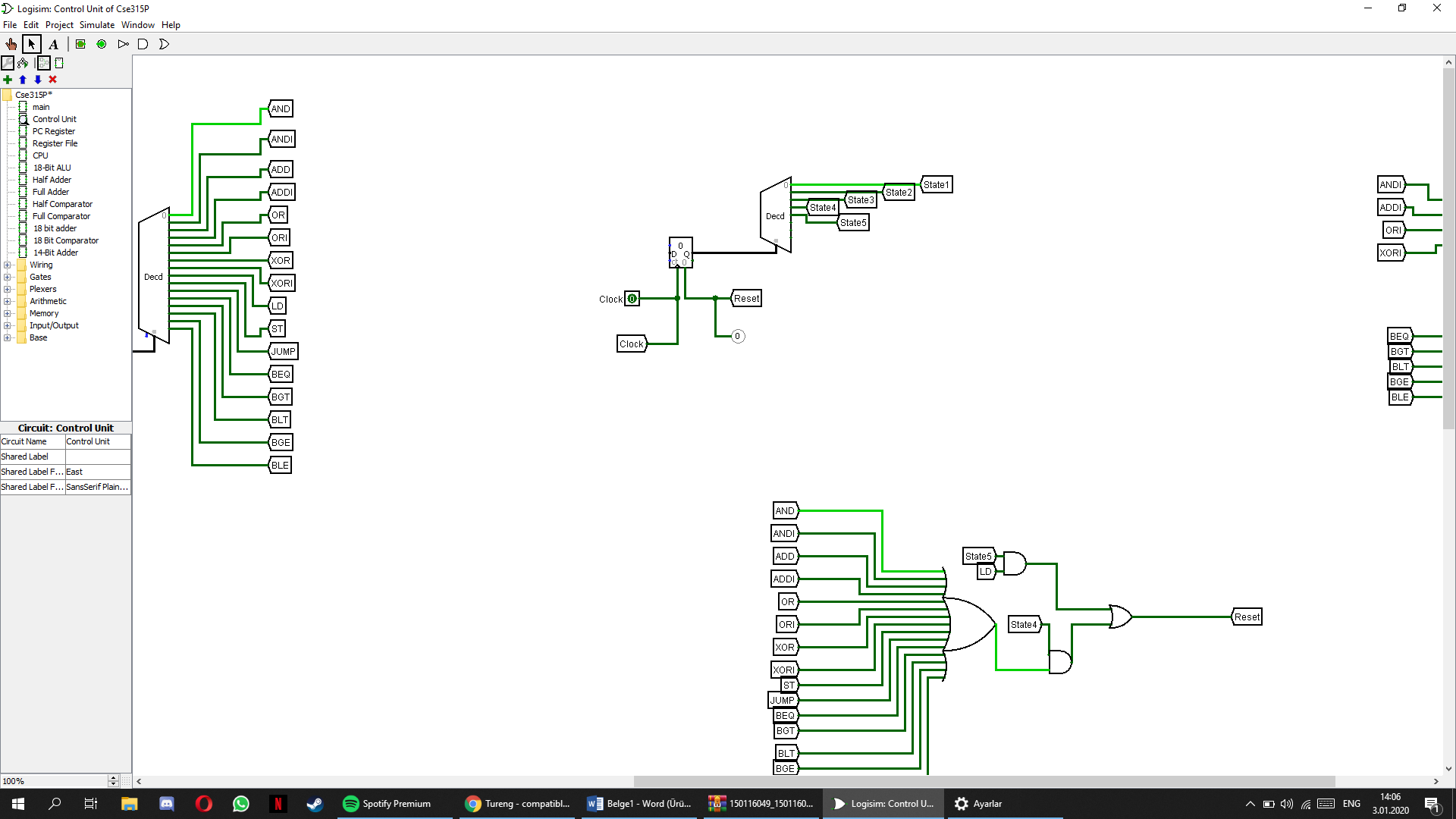
we have 16 instructions, 4 bit allocation is enough to indicate each instruction. As we can see, we have some limitations. IMM value will be represented as two’s complement and we allocated 5 bits for it. So, IMM value will be in the range [-16, 31]. Because of our data being 18 bits, we need to sign extend IMM bits. Also for LD and ST instructions, 10 bits are allocated for Address10.

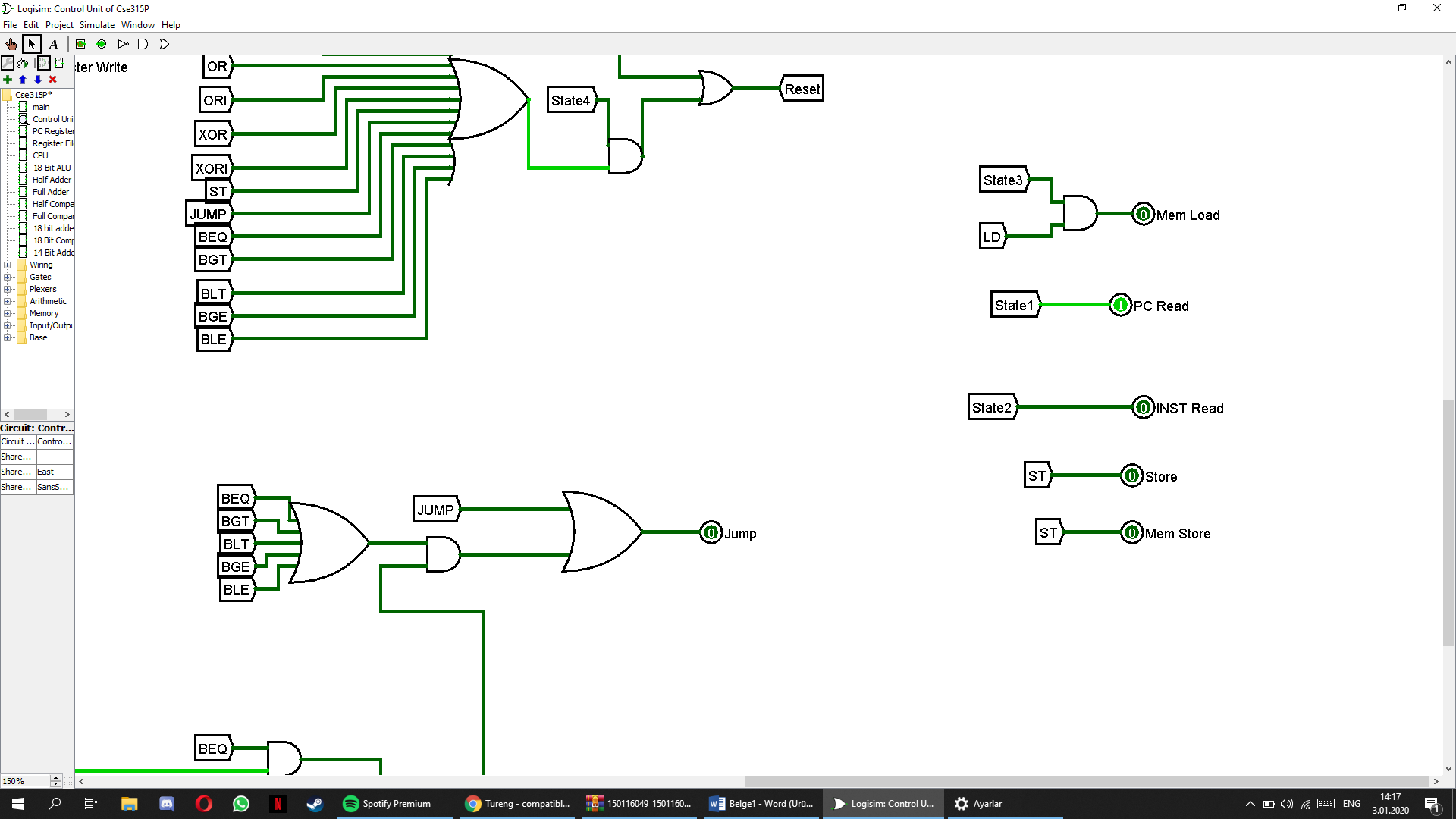
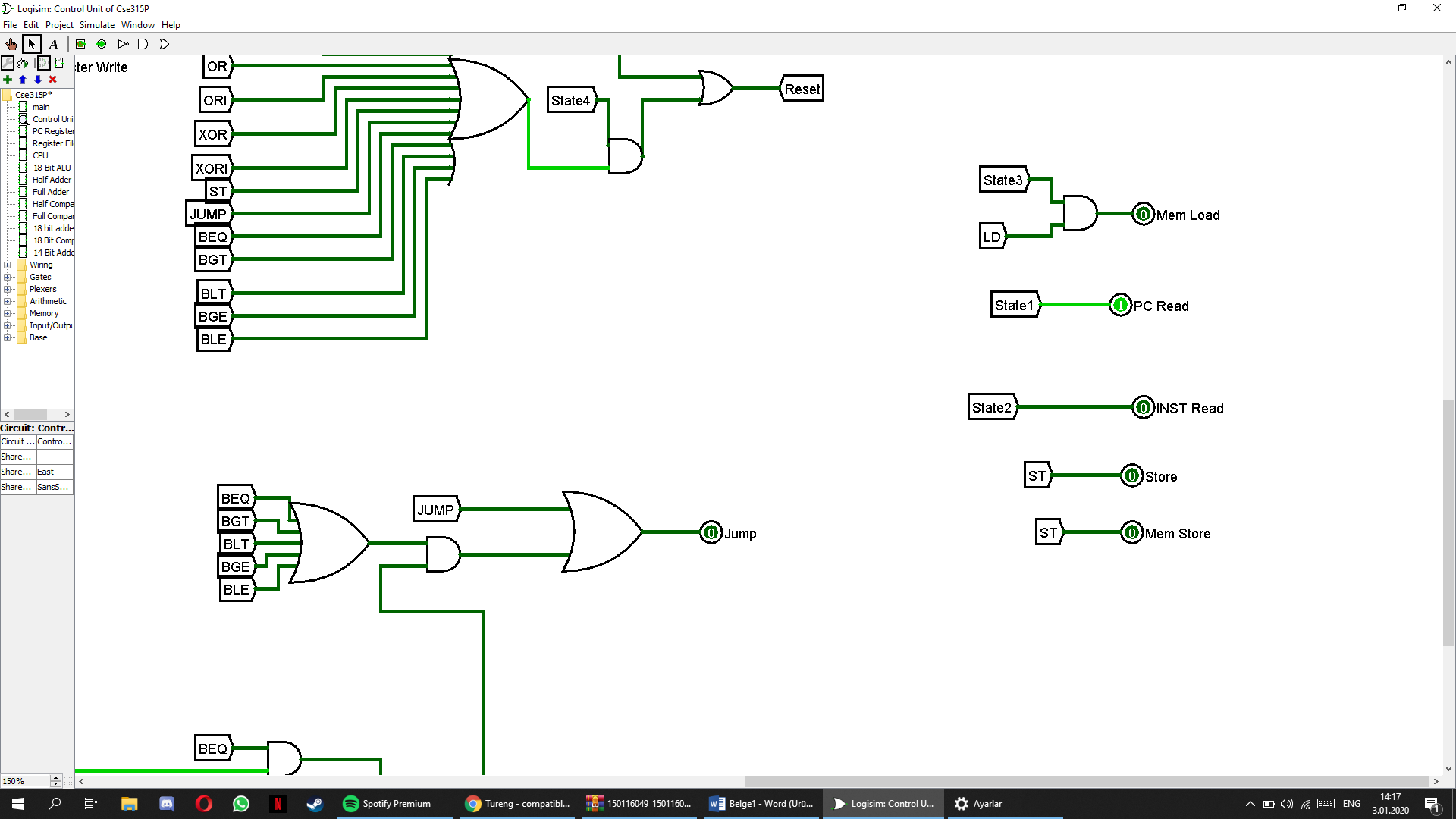
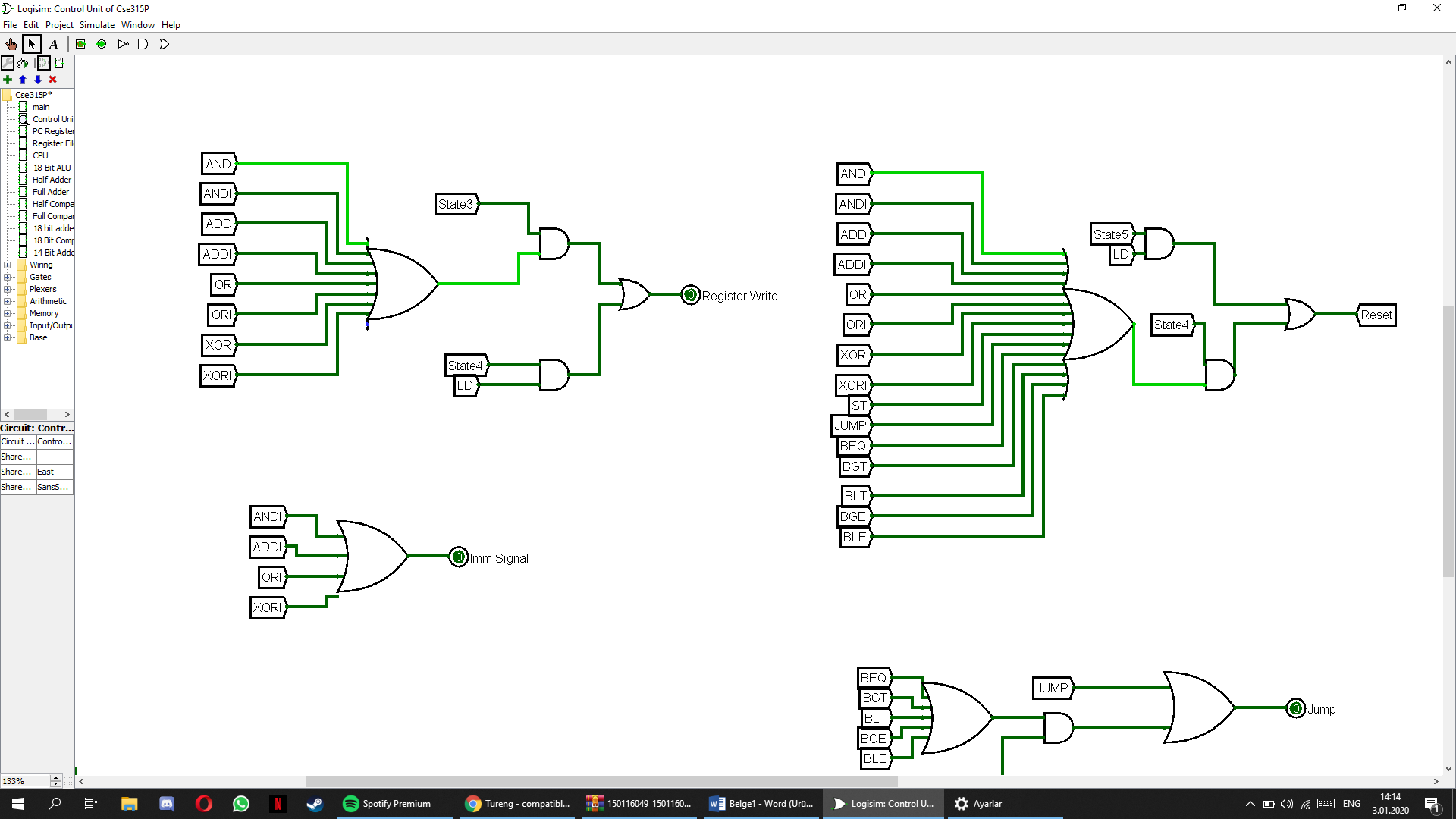
logism part:

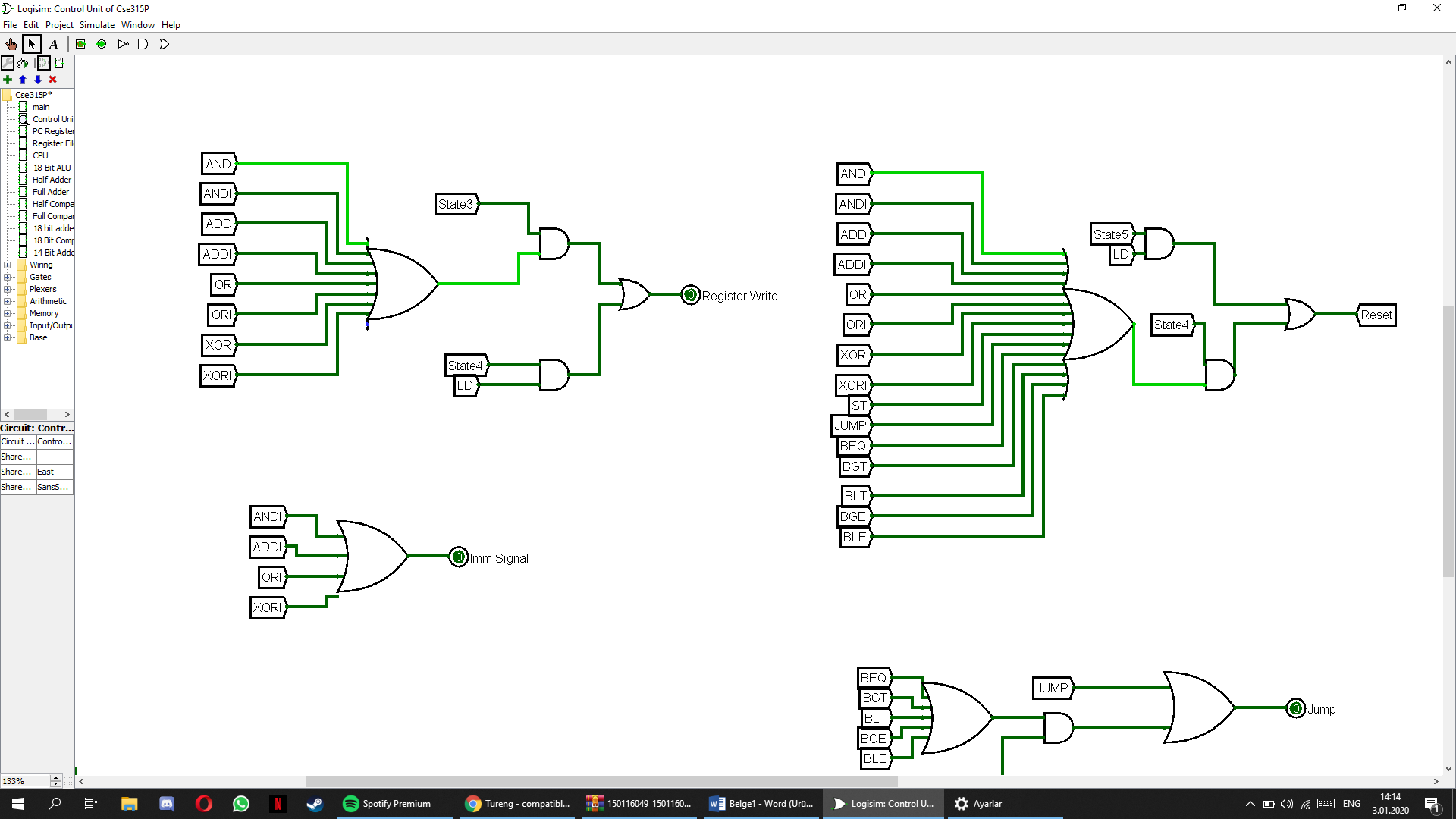
**Control Unit**

 Control unit is the unit that we handle all the signals that other components need.

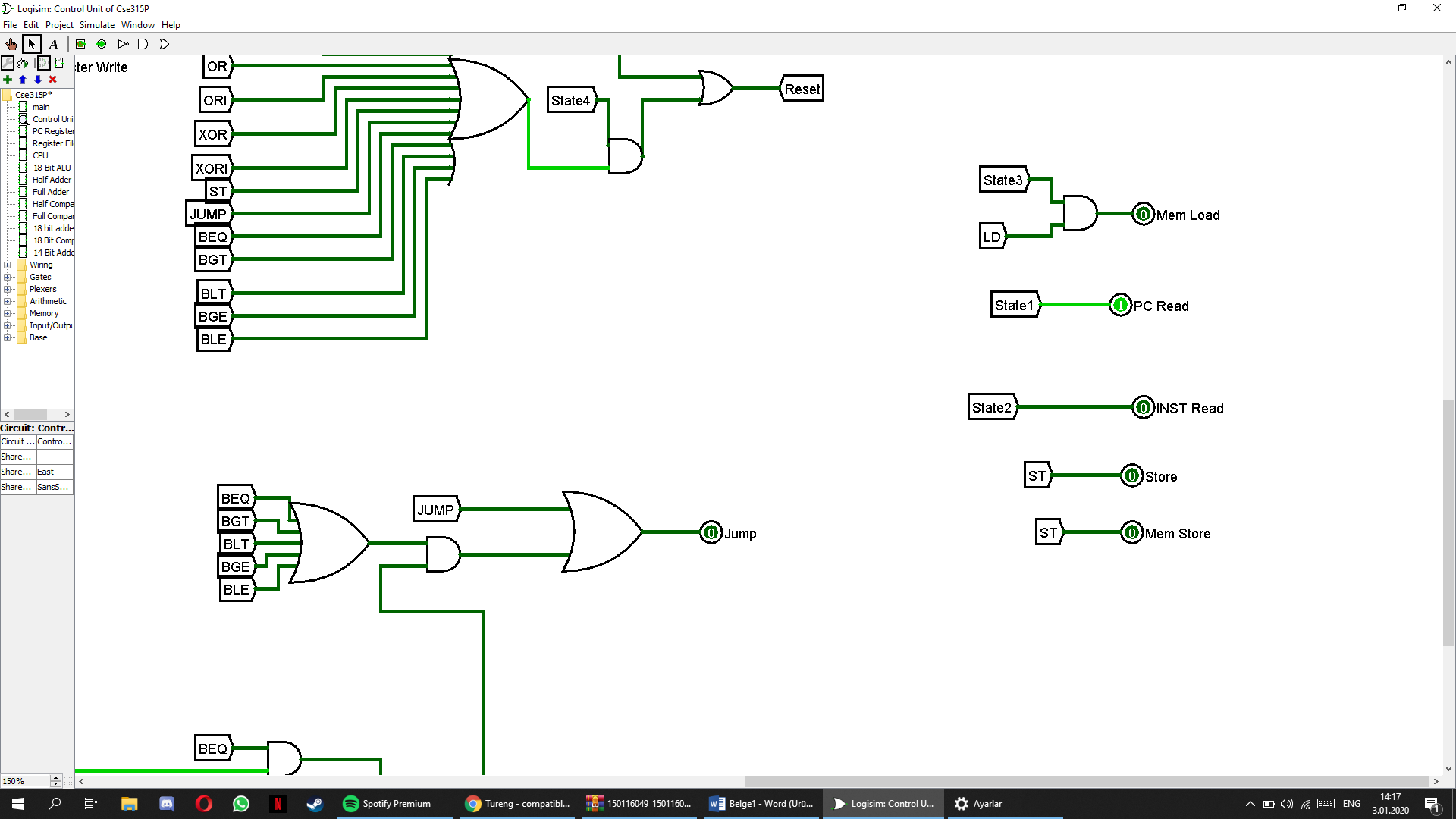
Instruction is sperated bit by bit compatible with instruction set archtitecture.

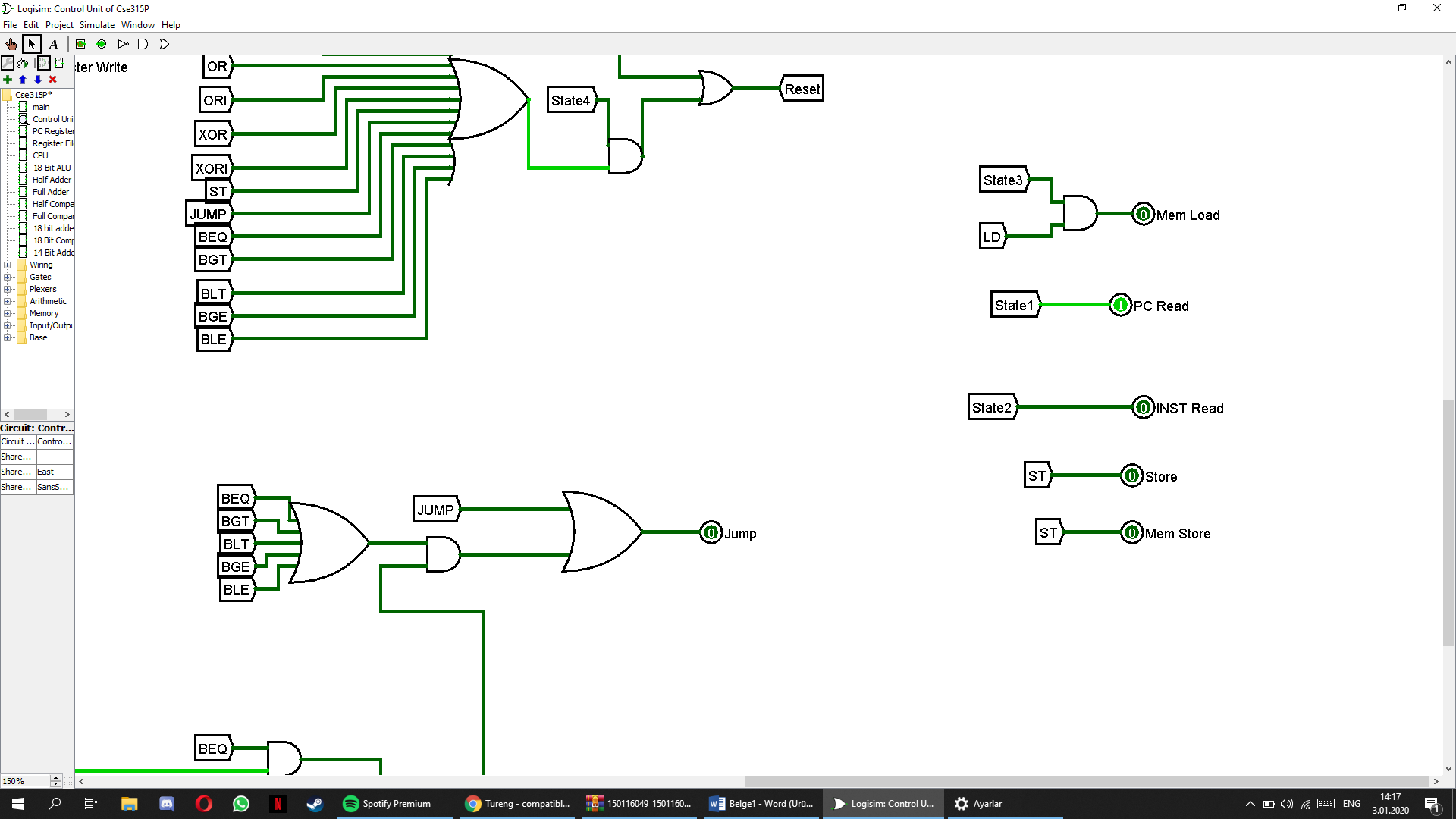
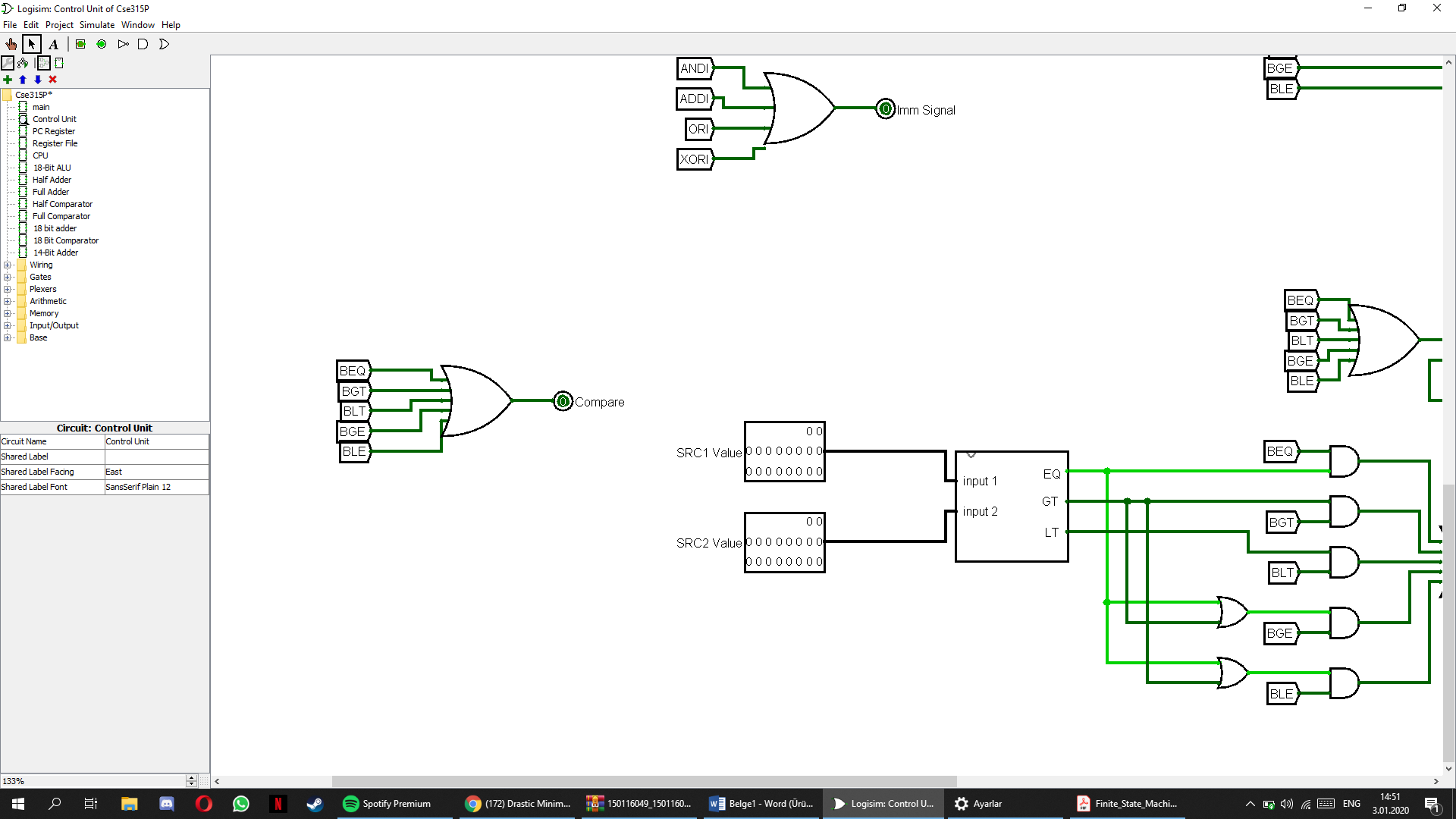
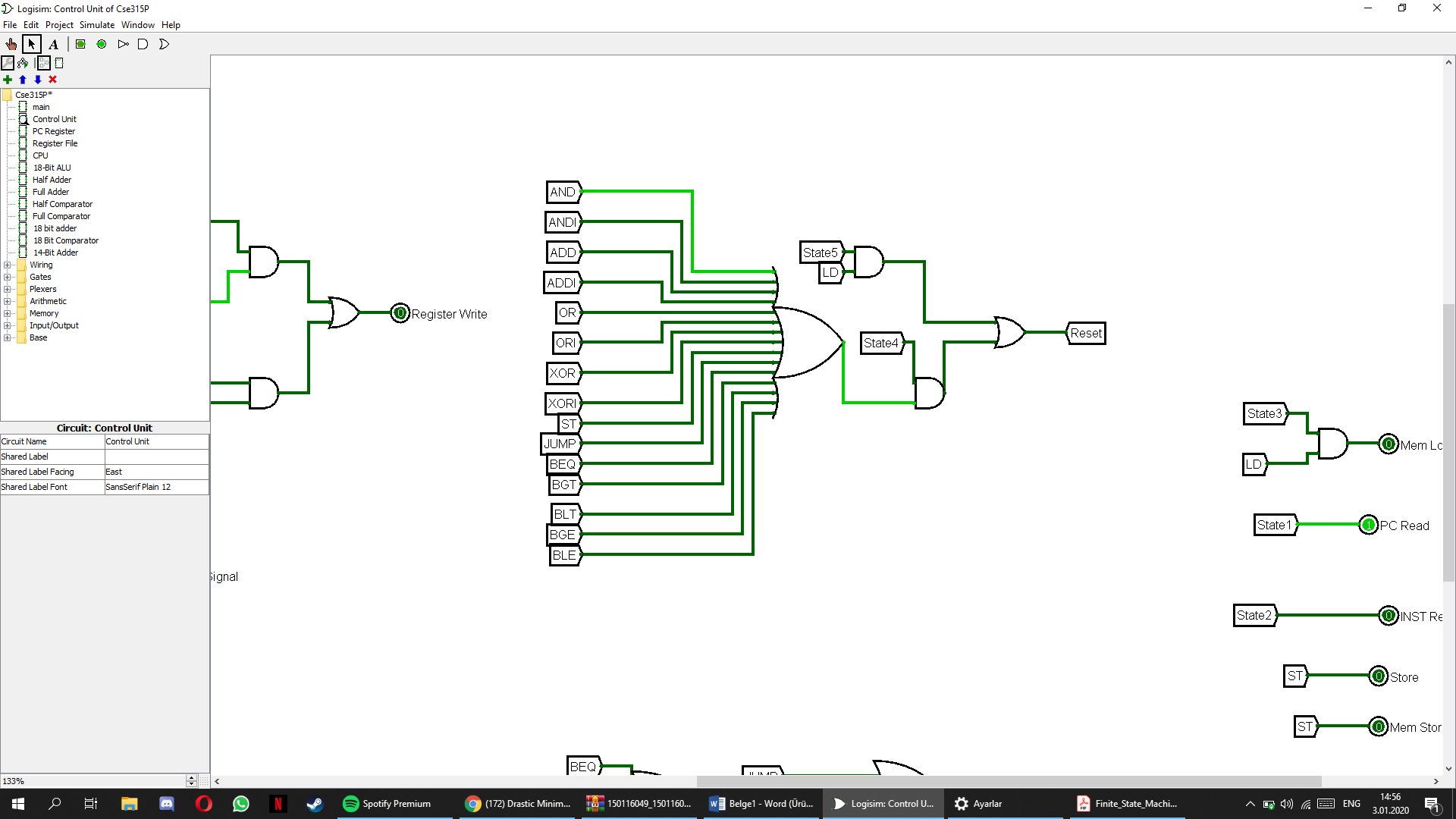
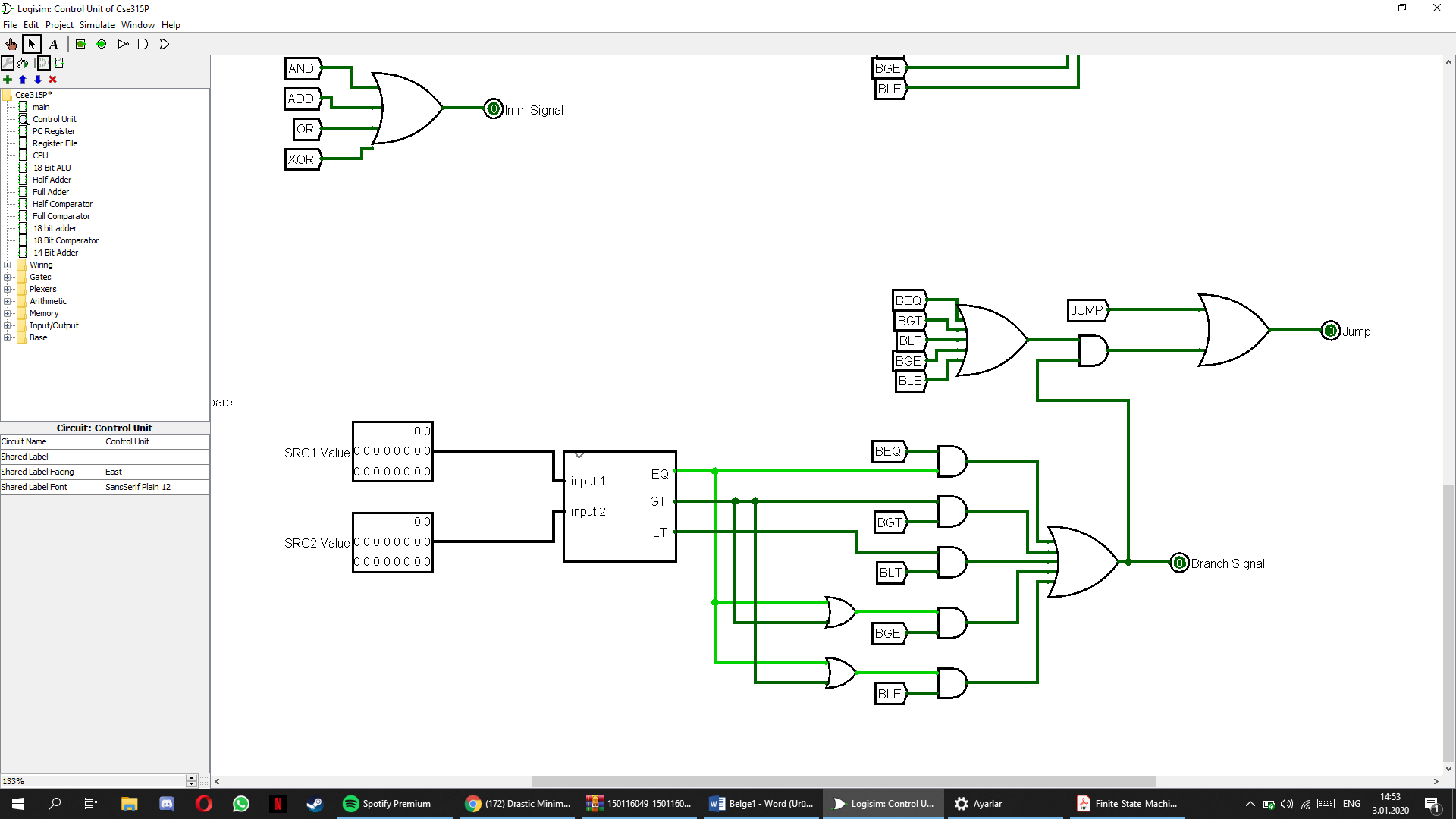
Finite state machine is developed by a counter. For every clock cycle, counter’s value increase and states change.

* For state 1 we generate PC Read signal to enable PC Register.
* For state 2 we generate INST Read signal to enable instruction register.
* For state 3 there are different types of signals that is generated for each instruction set.
  + For AND, ANDI, ADD, ADDI, OR, ORI, XOR, XORI Regiter Write signal is generated.
  + Also, for ANDI, ADDI, ORI, XORI operations imm signal and is generated.

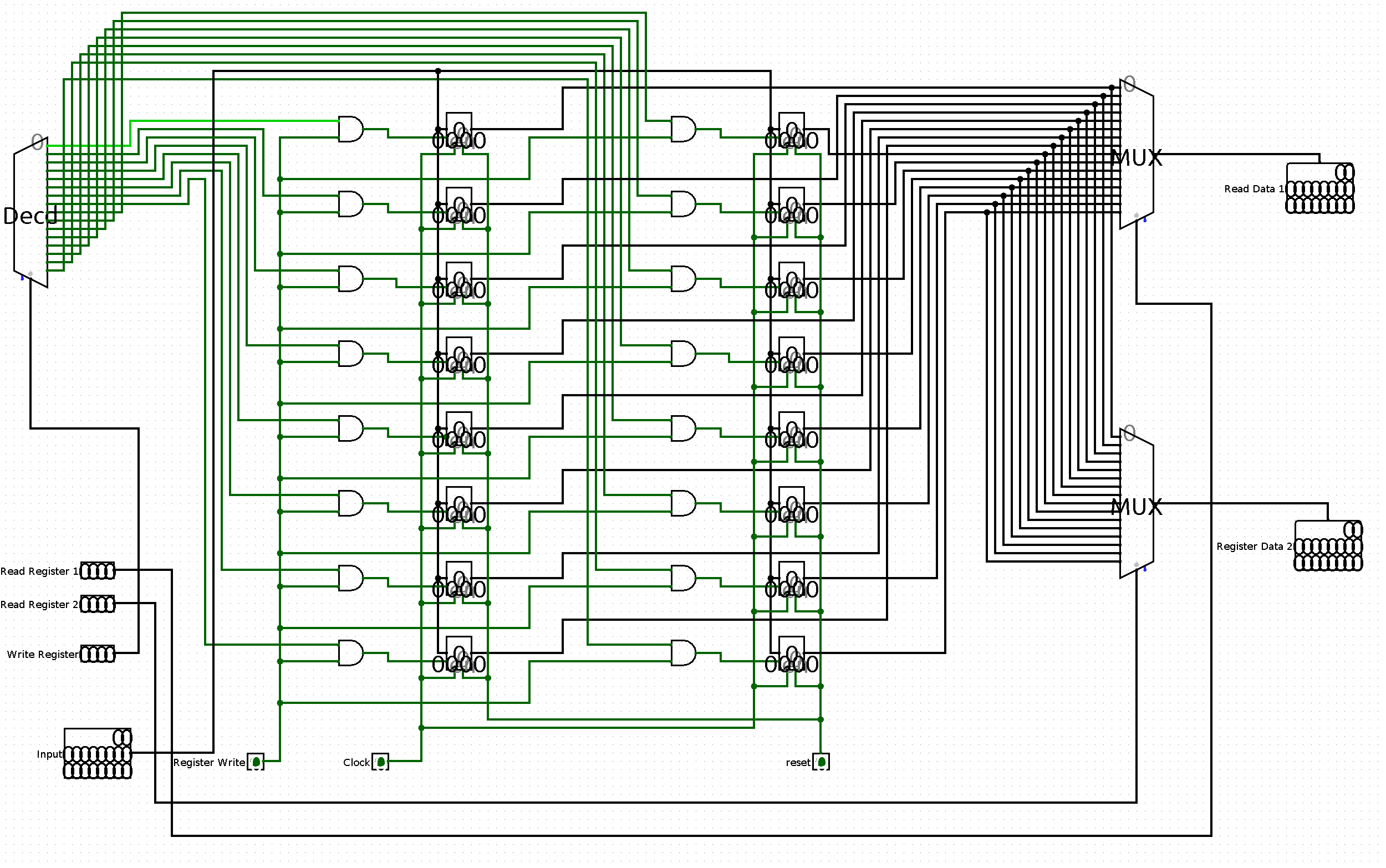


* + For LD intruction Mem Load signal generated.



* + For ST Mem Store signal generated.
  + For BEQ, BGT, BLT, BGE, BLE Compare signal is generated.
  + Compare signal takes values from register and it compares the values. From those values branch signal is generated. Also for JUMP, instruction, jump signal is generated.
  + For AND, ANDI, ADD, ADDI, OR, ORI, XOR, XORI, ST, JUMP, BEQ, BGT, BLT, BGE on state 4 resets the counter. For LD, on state 5 resets the counter.

**Register File**

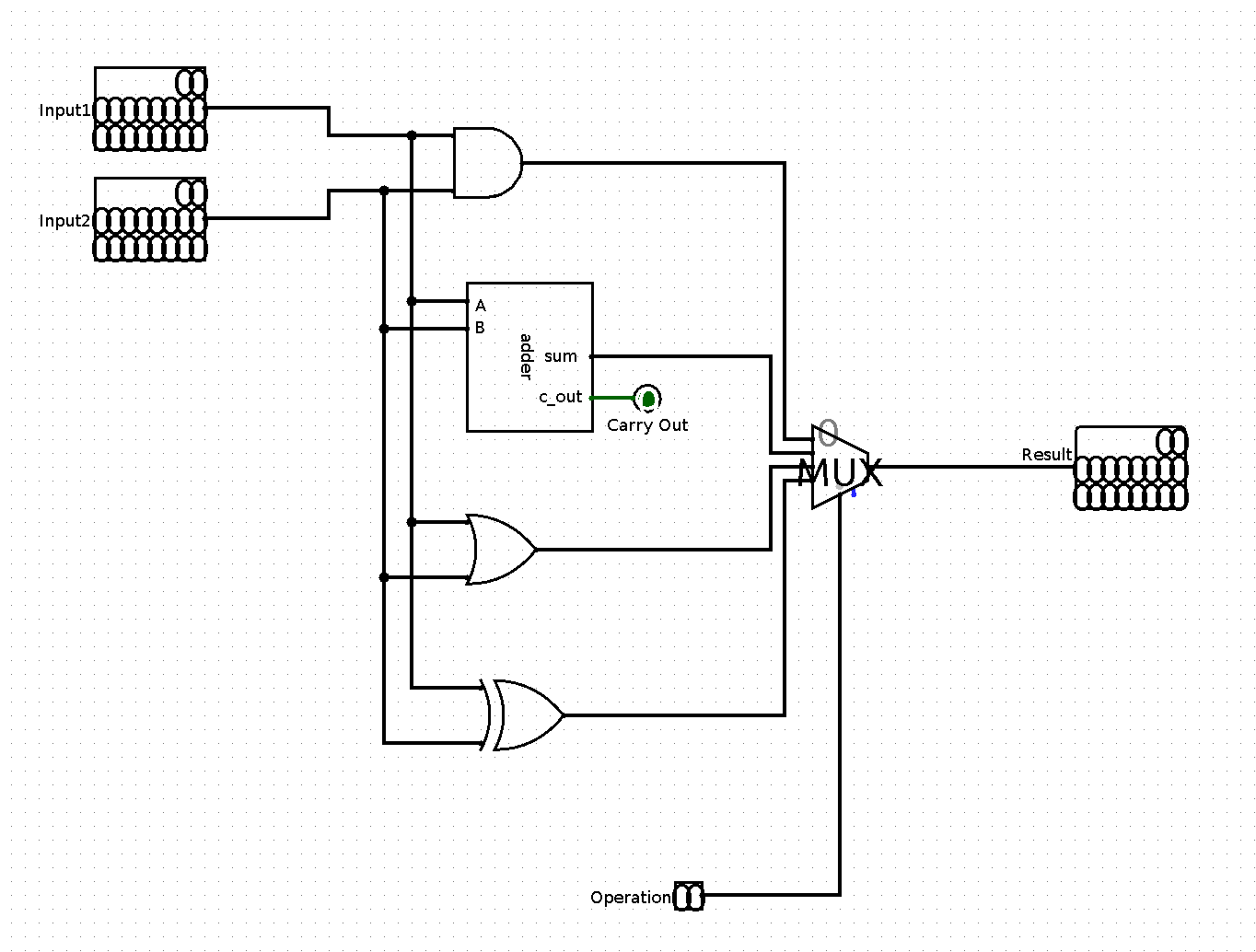


The register file contains sixteen registers, referred to by number as R0 through R15. 18 bit data are stored in each register. All of the registers are general purpose in that they may be freely used by any of the instructions that can write to the register file. Decoder is used to select register which the data will be stored and multiplexers to read values inside the registers.

Register Write signal to enable registers, if it is 1. Data can be written to the destination register. If it is 0 register file just read the data bits of the register.

If clear signal is enabled all registers’ data bits will be cleared.

**18-Bit Arithmetic Logic Unit**



Arithmetic logic unit handles arithmetic operations, in our case ADD, ADDI, AND, ANDI, OR, ORI, XOR and XORI.

ALU has 2 inputs input 1 and input 2 which is directed by the values read from Register File or immediate part of the instruction. Operation signal decides the operation to be done in ALU.

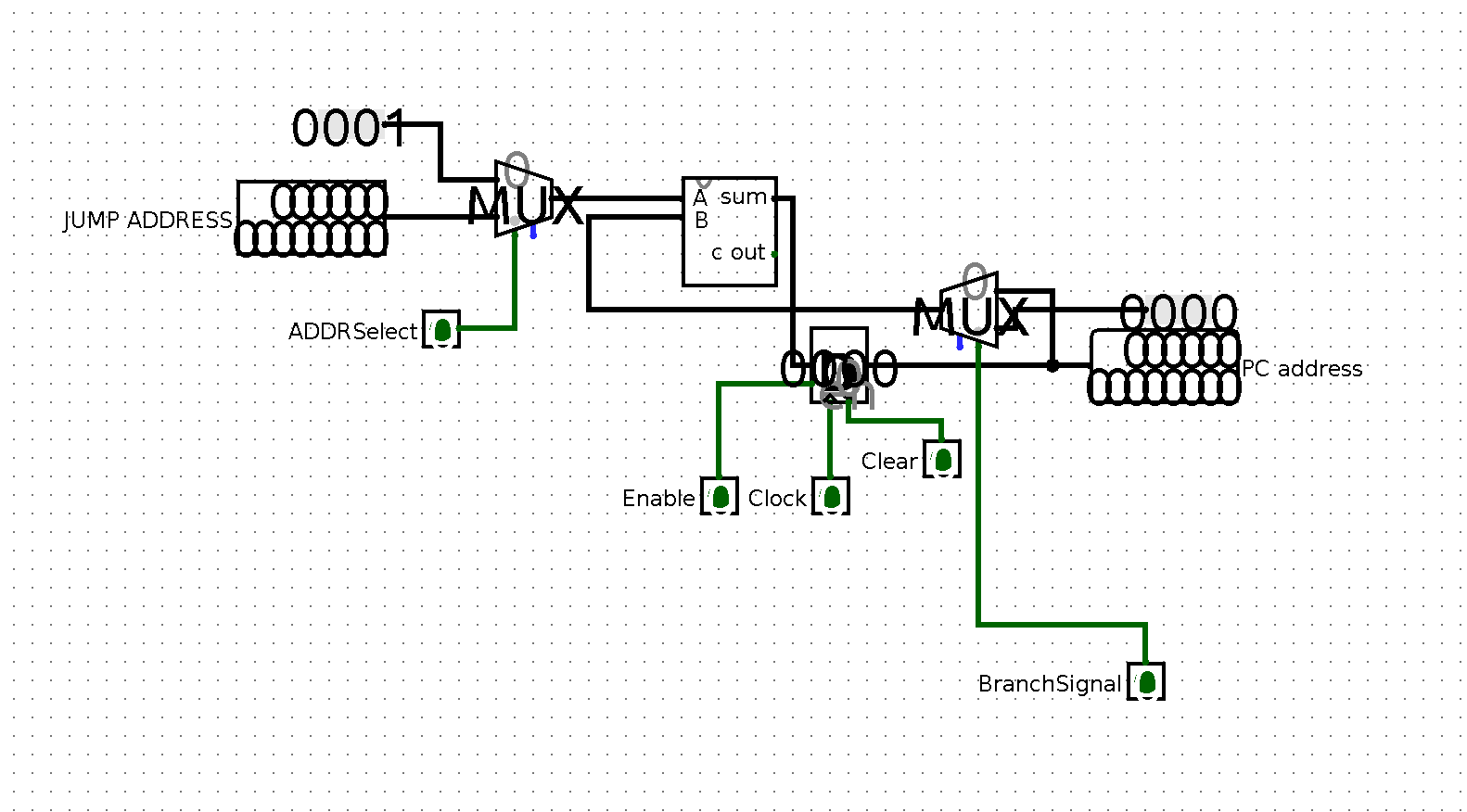
If Operation signal is 00 operation is AND,

If Operation signal is 01 operation is ADD,

If Operation signal is 10 operation is OR,

If Operation signal is 11 operation is XOR.

**PC Register**

Program Counter Register is designed to store PC and to handle operations like increase and JUMP to the destination address.

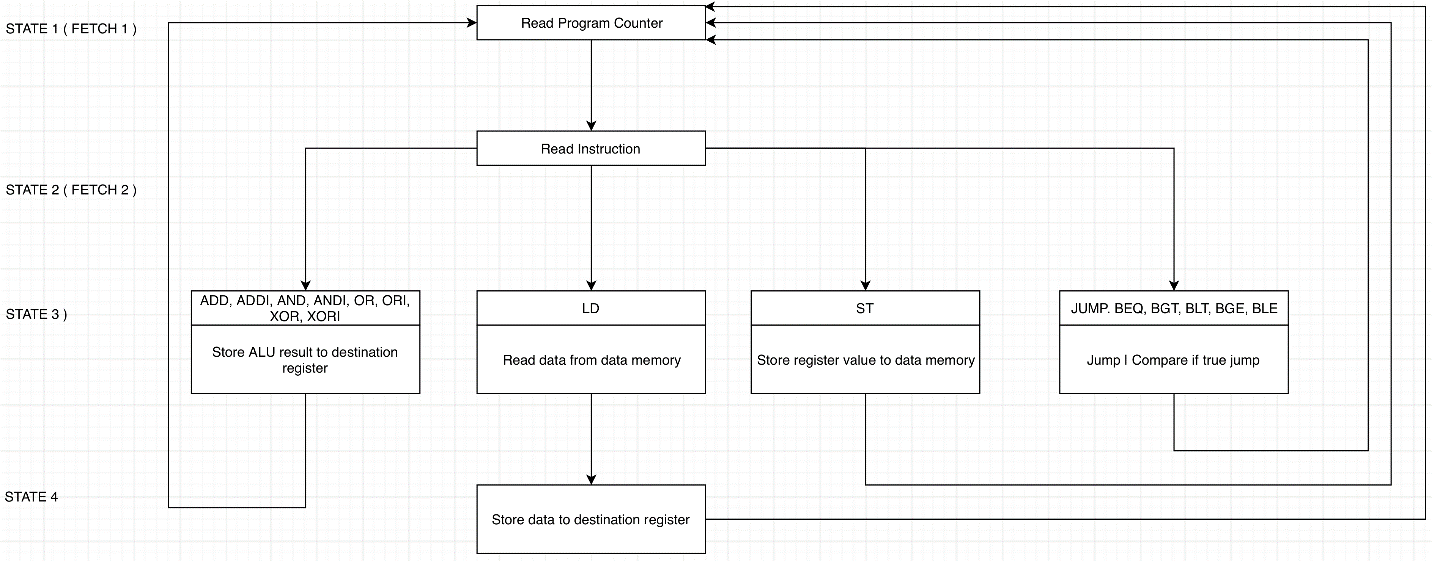
If ADDR select signal is 1, JUMP ADDRESS will be added to the PC value, if it is 0 PC will be increased by 1.

If BranchSignal is 1, PC address will be set to JUMP ADDRESS that comes from instruction address space. If BranchSignal is 0, current PC address will be sent to the adder.

Clear signal is to clear PC register’s stored value.

Enable signal is to activate PC register.

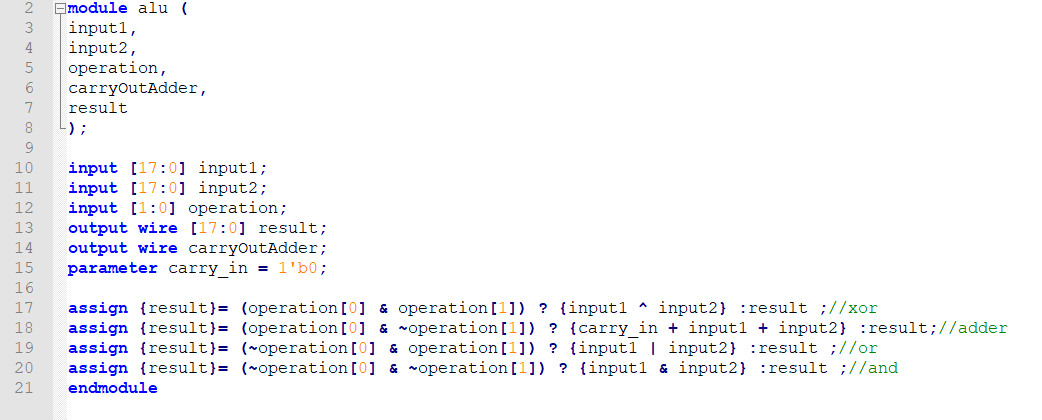
**Finite State Machine**



verilog design:

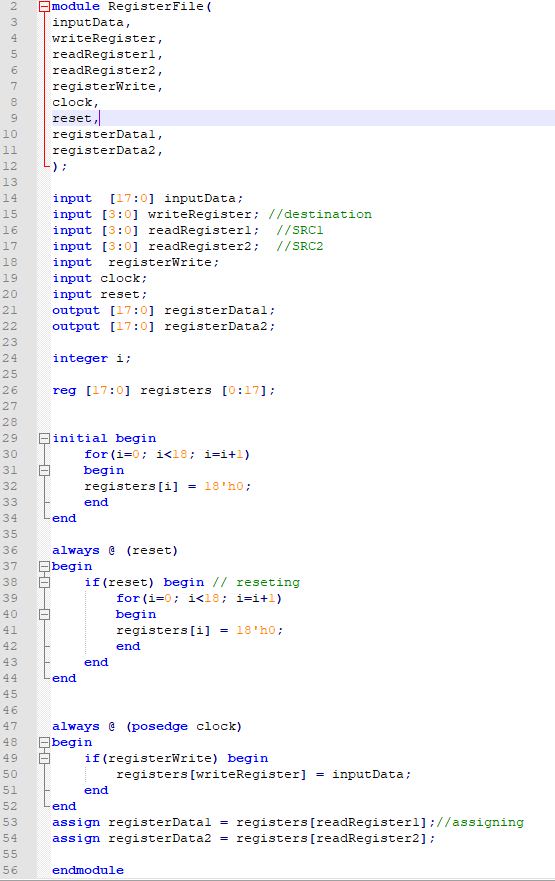
In this part, we need to implement our design with Verilog language, based on our Logisim design. We used ModelSim simulation software while compiling. Our implementation contains 5 modules:

**1-Arithmetic logic unit:**



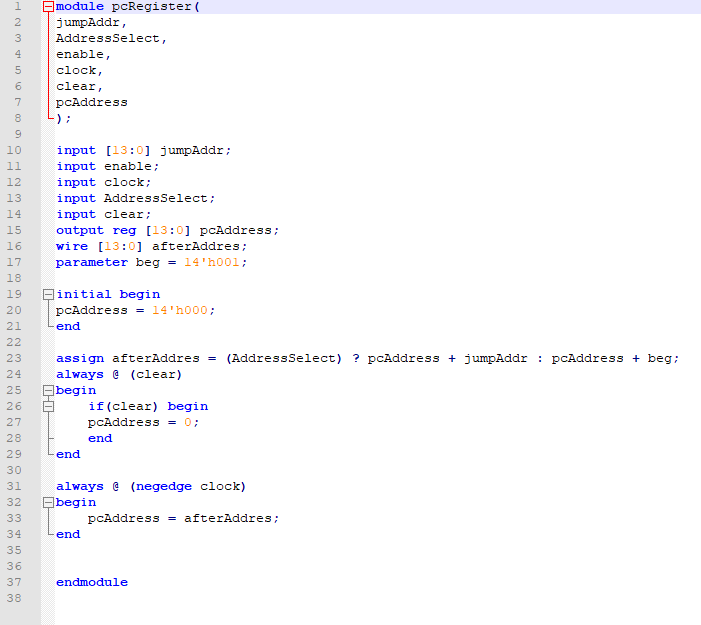
There are 2 inputs which have 18 bits lengths. We are adding them or anding them according to operation selection signal.

**2-Register File:**

****

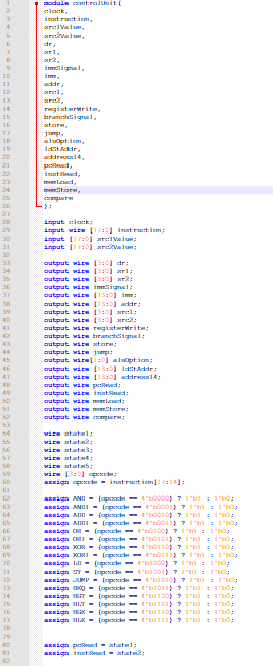
In this module, we have 16 registers which have 18 bits data sizes. This module has input ports for reading values inside two registers; destination register for selecting which register to store input data; input data to store in a register; enable pin; reset pin and clock pin. Register file outputs 2 values stored inside selected 2 registers.

**3-PC Register:**

****

PC register calculates the next PC value according to JUMPSignal and PC relative jump address, updates and outputs the new PC value when clock is on falling edge. If AddressSelect is 1, that means we are able to do jump operation, jumpAddr will be added to the current PC. Else, 1 will be added to the current PC value.

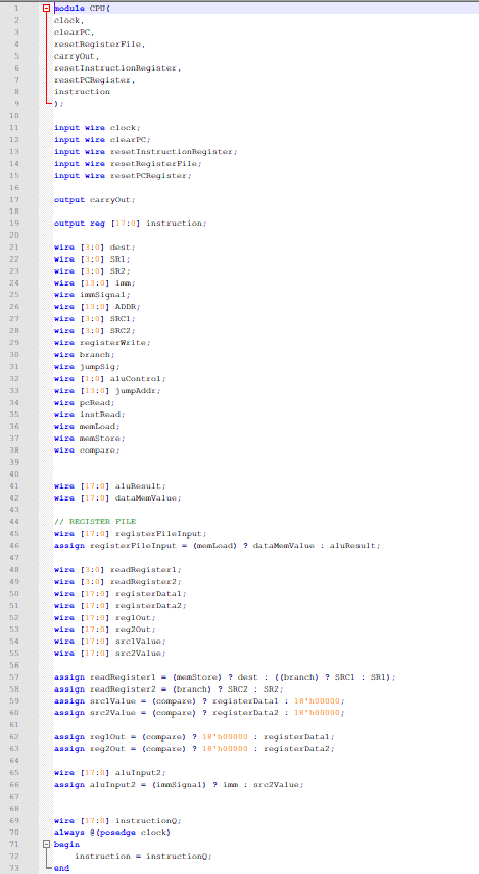
**4-Control Unit:**

****

In this component, what we are basically doing is parsing incoming instruction and setting signals according to this instruction.

We could not do tis part completely.

**5-CPU**



We could not do this part completely.