BIRZEIT UNIVERSITY

Faculty of Engineering & Technology
Electrical & Computer Engineering Department

COMPUTER ORGANIZATION AND MICROPROCESSOR
ENCS2380

Single Cycle Processor Design

Prepared by

Saeed Rasras 1230757

Majd Jehad 1231360

Ibrahim Isbaitan 1230992

Date 14-6-2

Table of Contents

Table of figures

I. Design and Implementation

I.I Arithmetic and Logic Unit

The Arithmetic and Logic Unit (ALU) is a component in the Central Processing Unit (CPU), with the objective of executing arithmetical and logical operations. The construction of the ALU unit consists of a 16-1 Multiplexer that receives its inputs from two Data Buses, which are received from the Register file. The multiplexer's inputs are connected to two 32-bit data buses, 'A' and 'B'.

These buses originate from the register file, which read register contents based on control signals. The ALU is assigned to perform 12 logical and arithmetic operations, those being: AND, CAND, OR, XOR, ADD, NADD, SEQ, SLT, SLL, SRL, SRA, ROR. Therefore, explaining the need for a 16-1 multiplexer to be enough for the 1100 operations needing at least 4-bits for the selection line of the multiplexer.

After the input of an operation on the selection line for the multiplexer, which consists of a 4-bit binary input, and receiving the inputs a & b from the register file, the ALU performs the selected operation by using logic gates and rotation logical and arithmetical blocks, then it gives a 32-bit output, which is used later for the needed operations.

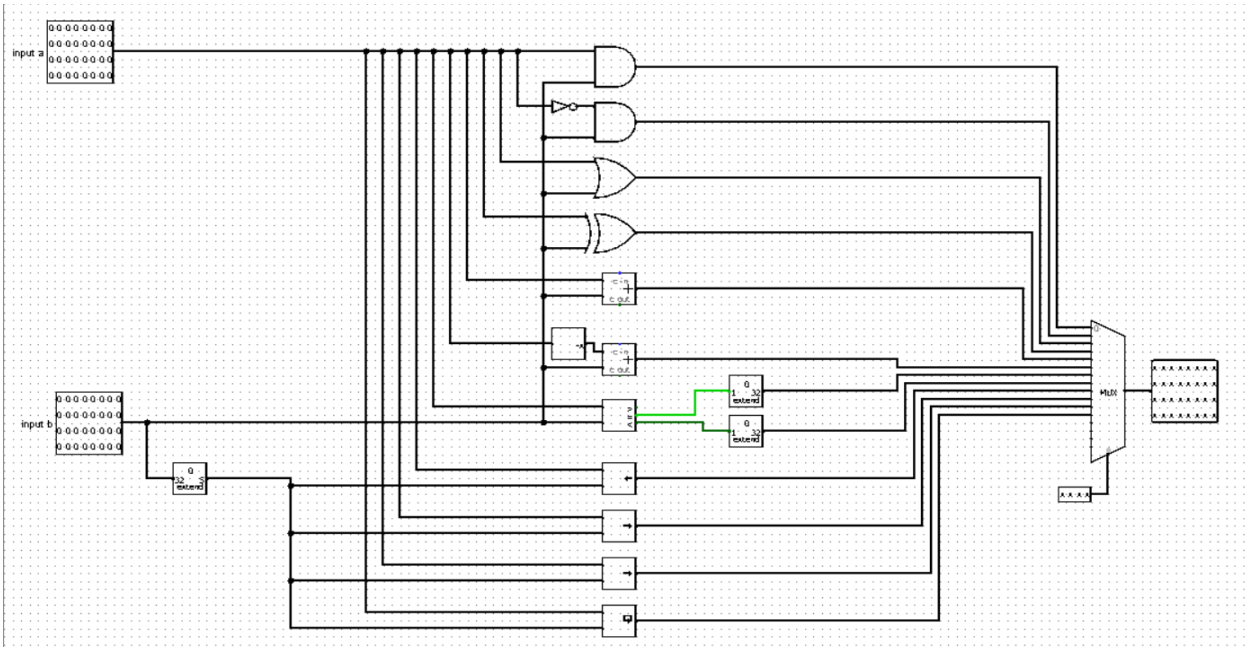


Figure 1: The Arithmetic and Logic Unit (ALU).

I.I.I AND

The ALU's first operation is the bitwise AND. An AND gate is connected to the zero input (index 0) of the main MUX, which corresponds to selection inputs '0000'. For instance, with data buses A and B holding '00000000000000000000000000000101' and '00000000000000000000000000000011' respectively, setting the selection lines to '0000' directs the bitwise AND result, '00000000000000000000000000000001', to the ALU output. it works by comparing bits one by one as follows: 0 AND 0,0 AND 1, 1 AND 0. all correspond to zero output while 1 AND 1 outputs 1.

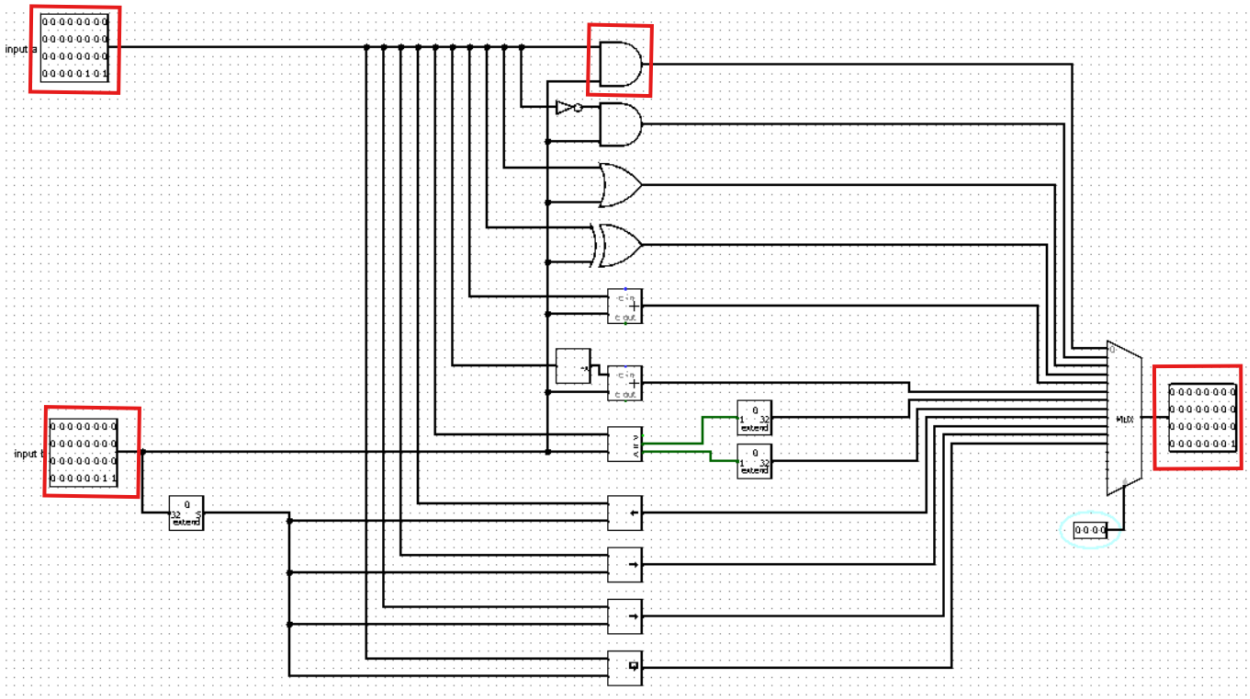


Figure 2: AND operation performed in the ALU.

I.I.II CAND

The CAND operation is the second operation of the ALU, it computes a bitwise result of ~ A&B. This means each bit of data bus A is first inverted, and then the inverted bit is ANDed with the corresponding bit from data bus B. This functional unit is placed at the main MUX's second input (index 1), corresponding to selection inputs '0001'. For example, if data buses A and B hold '00000000000000000000000000000101' and '00000000000000000000000000000011' respectively, setting the selection lines to '0001' would direct the bitwise CAND result, '00000000000000000000000000000010', to the ALU output. This operation can be summarized as follows for each bit pair (A, B): 0 CAND 0 outputs 0, 0 CAND 1 outputs 1, 1 CAND 0 outputs 0, and 1 CAND 1 outputs 0.

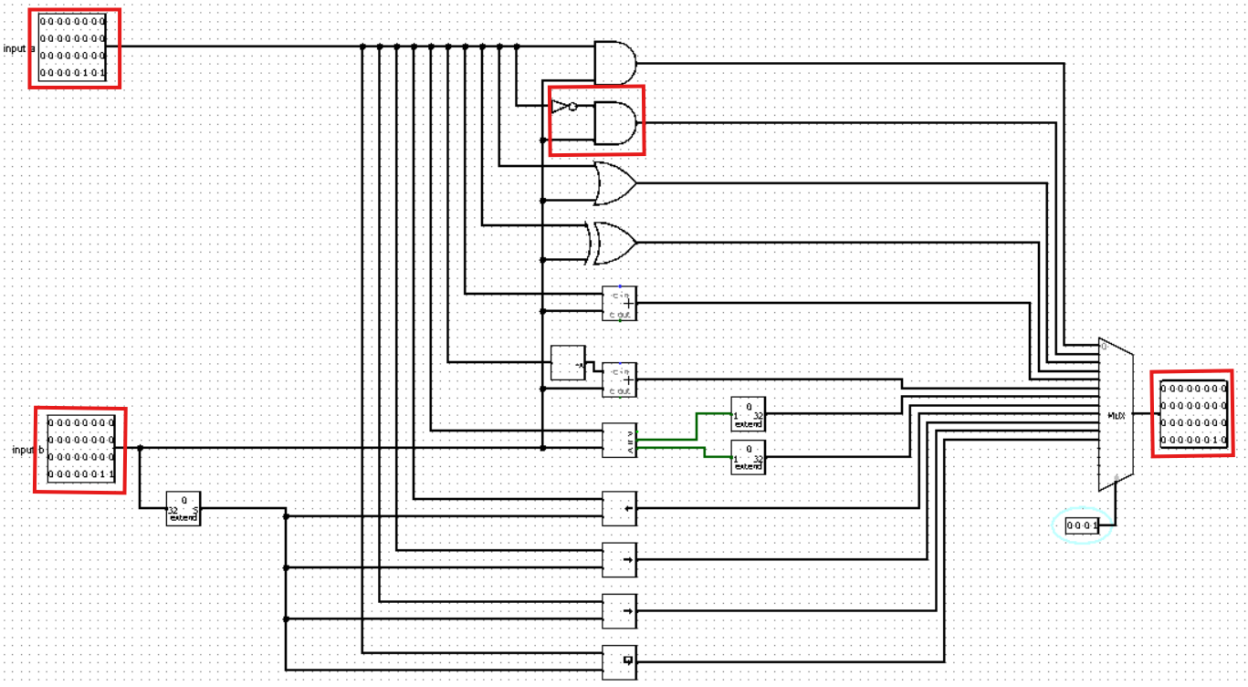


Figure 3: CAND operation performed in the ALU.

I.I.III OR

The OR operation performs a bitwise logical OR between the two input data buses. An OR gate is connected to the main MUX's third input (index 2), corresponding to selection inputs '0010'. This operation compares each bit of data bus A with the corresponding bit of data bus B. The output bit is '1' if at least one of the input bits is '1'; otherwise, the output is '0'. For example, if data buses A and B hold '00000000000000000000000000000101' and '00000000000000000000000000000011' respectively, setting the selection lines to '0010' directs the bitwise OR result, '00000000000000000000000000000111', to the ALU output. This operation can be summarized for each bit pair (A, B): 0 OR 0 outputs 0, 0 OR 1 outputs 1, 1 OR 0 outputs 1, and 1 OR 1 outputs 1.

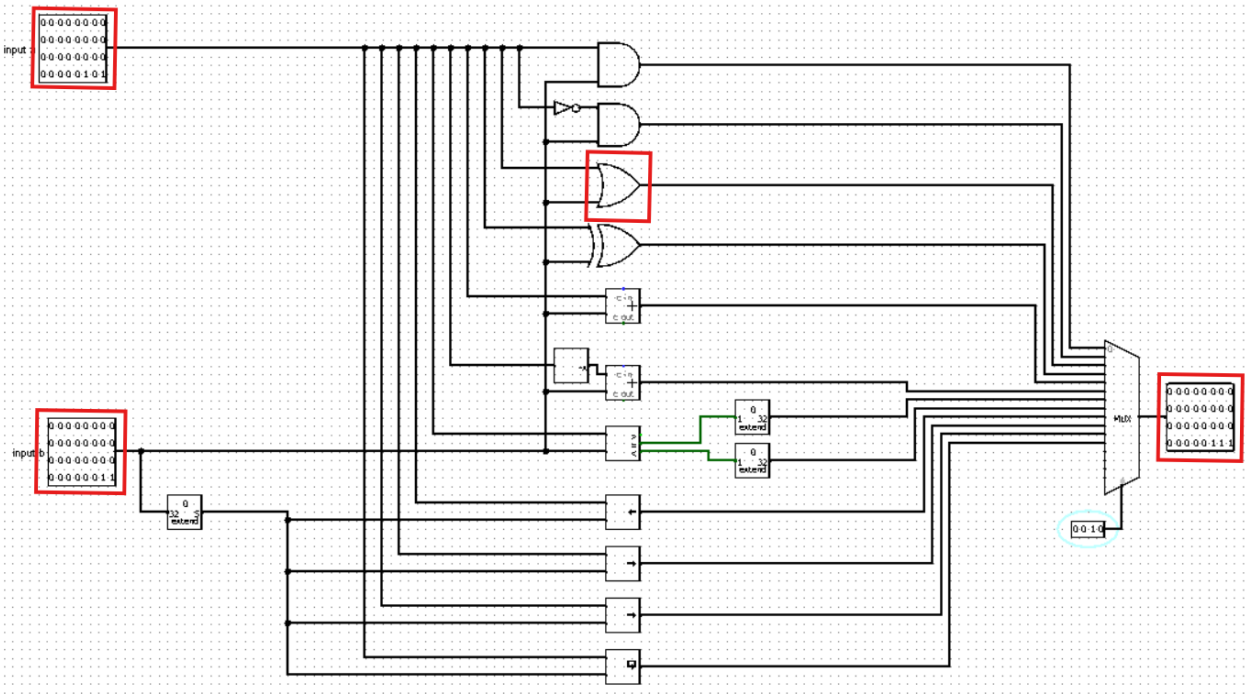


Figure 4: OR operation performed in the ALU.

I.I.IV XOR

The XOR is the fourth operation of the ALU, it performs a bitwise logical exclusive OR between the two 32-bit input data buses. An XOR gate is connected to the main MUX's fourth input (index 3), corresponding to selection inputs '0011'. This operation compares each bit of data bus A with the corresponding bit of data bus B. The output bit is '1' if the input bits are different, and '0' if they are the same. For example, if 32-bit data buses A and B hold '00000000000000000000000000001111' and '00000000000000000000000000110011' respectively, setting the selection lines to '0011' directs the bitwise XOR result, '00000000000000000000000000111100', to the ALU output. This operation can be summarized for each bit pair (A, B): 0 XOR 0 outputs 0, 0 XOR 1 outputs 1, 1 XOR 0 outputs 1, and 1 XOR 1 outputs 0.

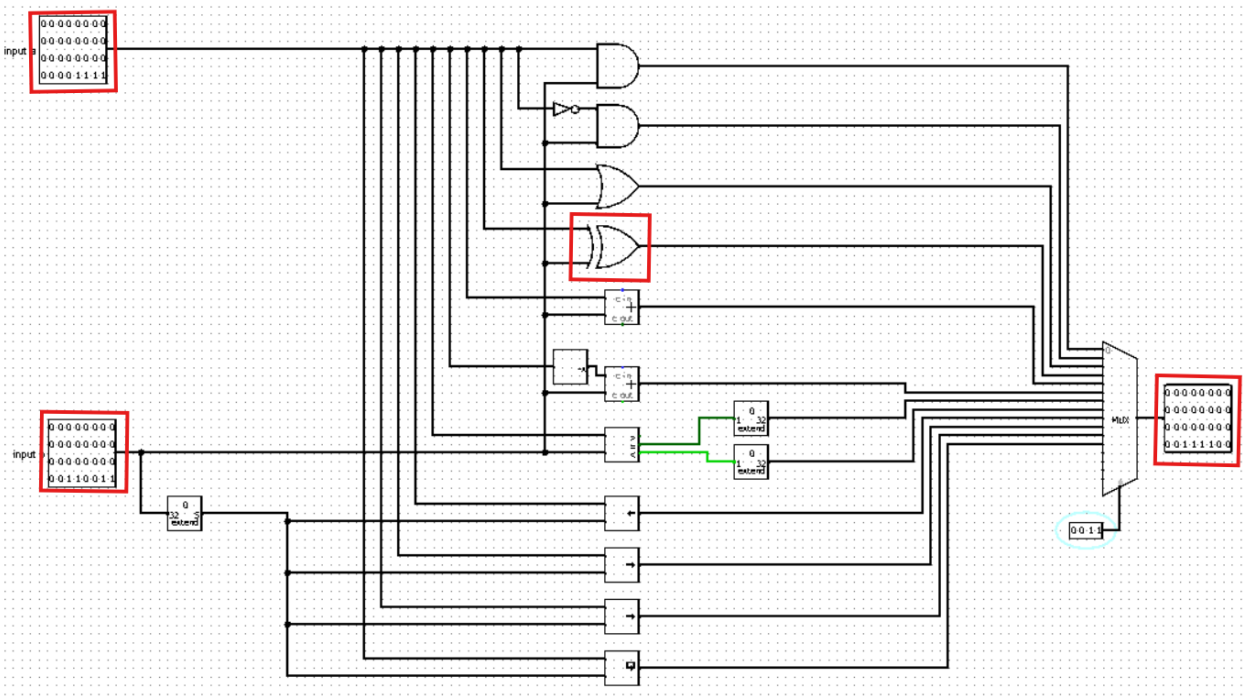


Figure 5: XOR operation performed in the ALU.

I.I.V ADD

The ADD operation is the fifth operation in the ALU, it is implemented using a 32bit adder, which takes inputs from both data buses. During an ADD operation, each bit from one data bus is added to the corresponding bit of the other, with carries propagated to the next bit pair. To output the calculated sum from the main 16-to-1 MUX, its selection lines must be set to '0100'. The adder's inputs are connected to 32-bit data buses 'A' and 'B'. For example, if data bus 'A' contains four and data bus 'B' contains three (in binary), selecting '0100' on the MUX lines directs their sum, 7 (in binary), to the ALU result, as shown in the figure bellow.

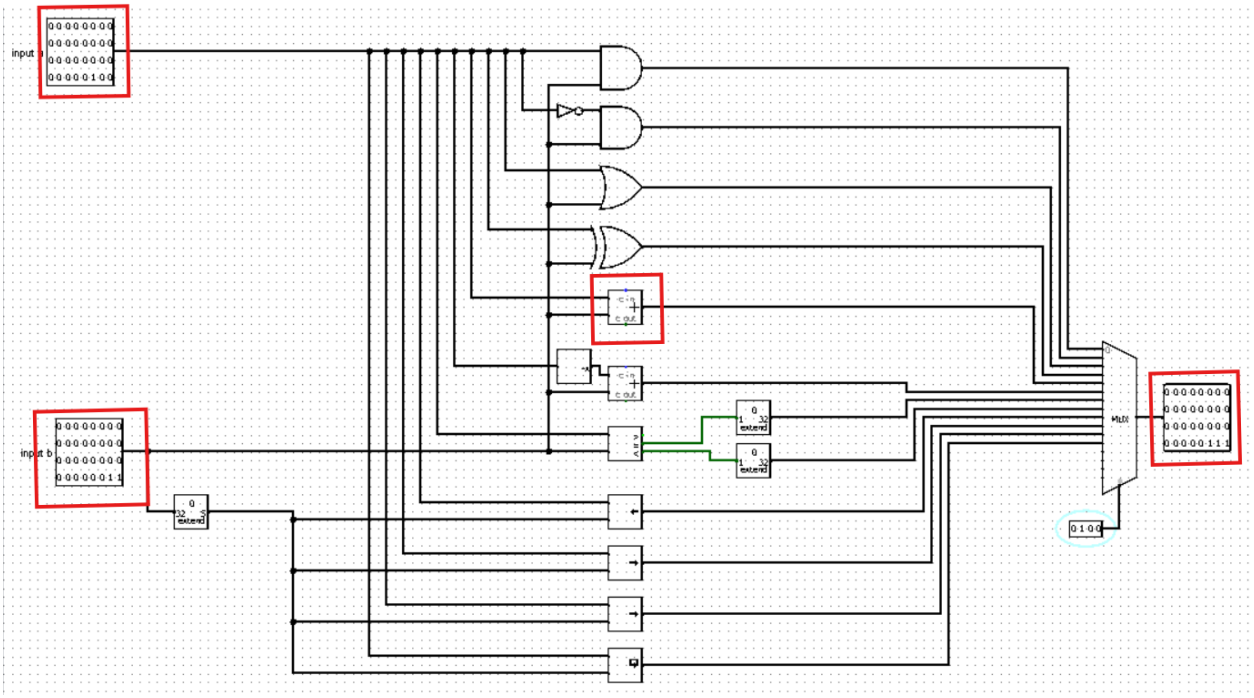


Figure 6: ADD operation performed in the ALU.

I.I.VI NADD

The NADD operation is the sixth operation of the ALU, it performs an arithmetic subtraction, specifically calculating (-A) + B, which is equivalent to B - A. This involves computing the two's complement of the 32-bit value from data bus A and then adding it to the 32-bit value from data bus B. This functional unit is connected to the main MUX's sixth input (index 5), corresponding to selection inputs'0101'. For example, if 32-bit data bus A holds 00000000000000000000000000000101 (decimal 5) and data bus B holds 00000000000000000000000000001000 (decimal 8), setting the selection lines to '0101' directs the result, 00000000000000000000000000000011 (decimal 3), to the ALU output

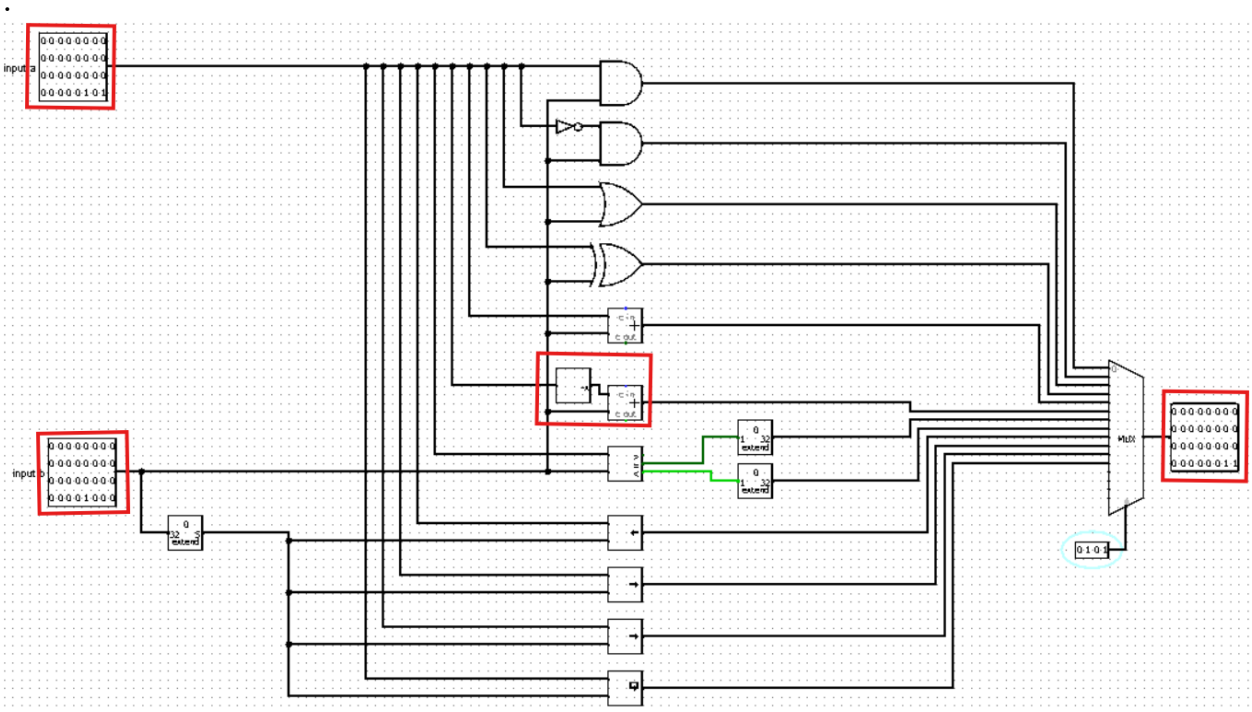


Figure 7: NADD operation performed in the ALU.

I.I.VII SEQ

The SEQ (Set if Equal) operation is the seventh operation in the ALU, it uses a comparator to check for equality between the two 32-bit input data buses. This functional unit is connected to the main MUX's seventh input (index 6), corresponding to selection inputs '0110'. A dedicated comparator circuit determines if the value on data bus A is identical to the value on data bus B. If the two input values are equal, the ALU output is typically set to `0x00000001` (all bits '0' except for the LSB representing a logical true). If the values are not equal, the ALU output is `0x00000000`. For example, if both data buses A and B hold `00000000000000000000000000000101`, setting the selection lines to '0110' directs the result, `00000000000000000000000000000001`, to the ALU output.

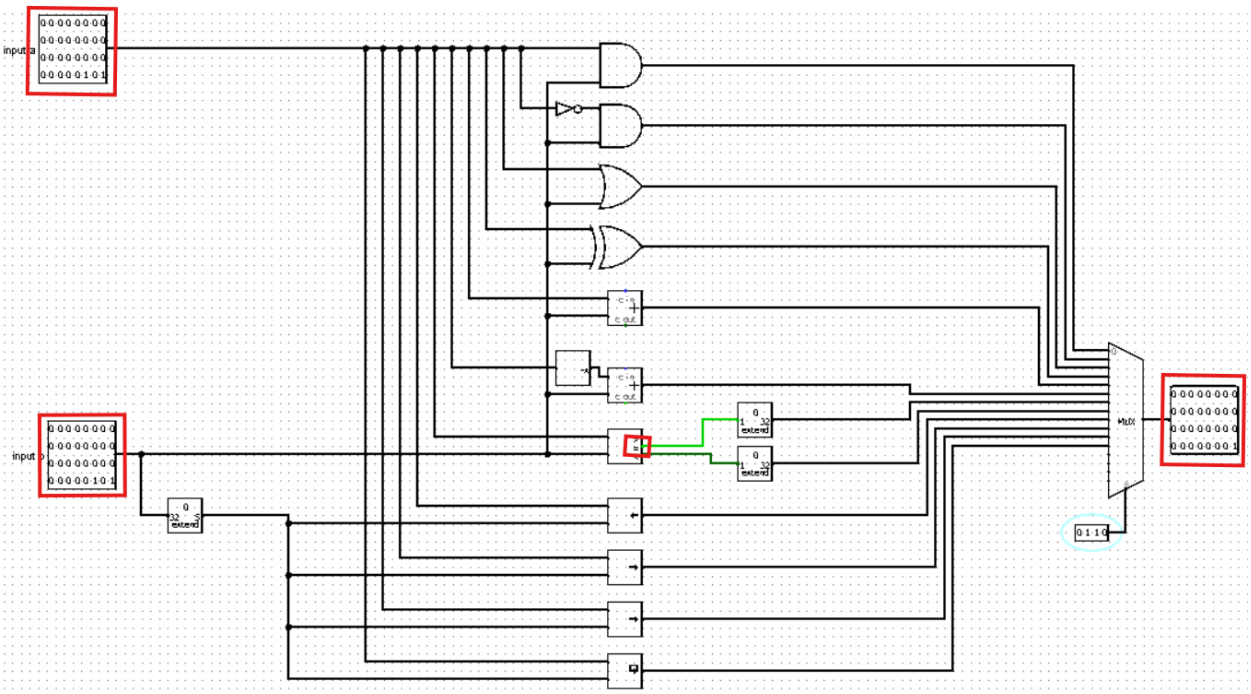


Figure 8: SEQ operation performed in the ALU.

I.I.VIII SLT

The SLT (Set Less Than) operation is the eighth operation in the ALU, it performs a comparison between the two 32-bit signed input data buses. This functional unit is connected to the main MUX's eighth input (index 7), corresponding to selection inputs '0111'. A dedicated comparison circuit determines if the value on data bus A is strictly less than the value on data bus B. If A is less than B, the 32-bit ALU output is set to `0x00000001`. Otherwise, the ALU output is `0x00000000`. For example, if data bus A holds `00000000000000000000000000000101` (decimal 5) and data bus B holds `00000000000000000000000000001000` (decimal 8), setting the selection lines to '0111' directs the result, `00000000000000000000000000000001`, to the ALU output.

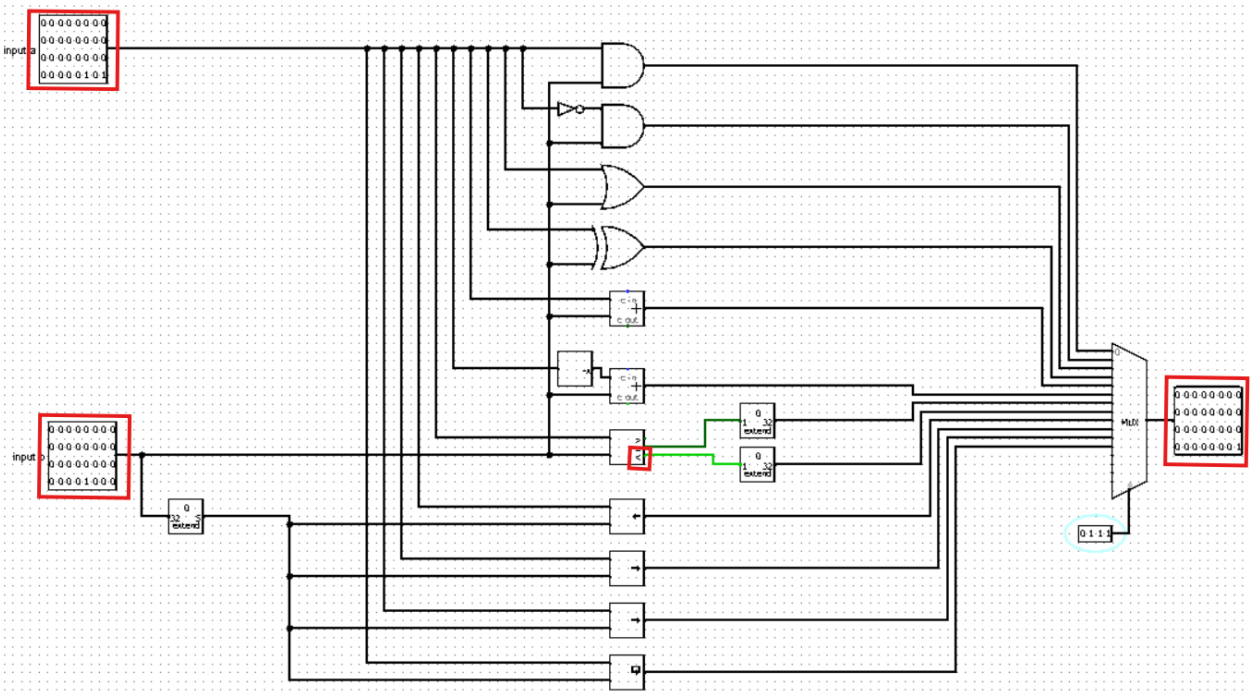


Figure 9: SLT operation performed in the ALU.

I.I.VIIII SLL

The SLL (Shift Left Logical) is the nineth operation of the ALU, it performs a bitwise logical left shift on the 32-bit value from data bus A. The shift amount is determined by an 8-bit immediate value (`imm8`), which is typically derived from the instruction's operand field or a portion of data bus B. During the shift, bits are moved to the left, and zeroes are inserted into the vacated least significant bit positions. This functional unit is connected to the main MUX's ninth input (index 8), corresponding to selection inputs '1000'. For example, if 32-bit data bus A holds `00000000000000000000000000000101` (decimal 5) and the `imm8` value is 2, setting the selection lines to '1000' directs the shifted result, `00000000000000000000000000010100` (decimal 20), to the ALU output.

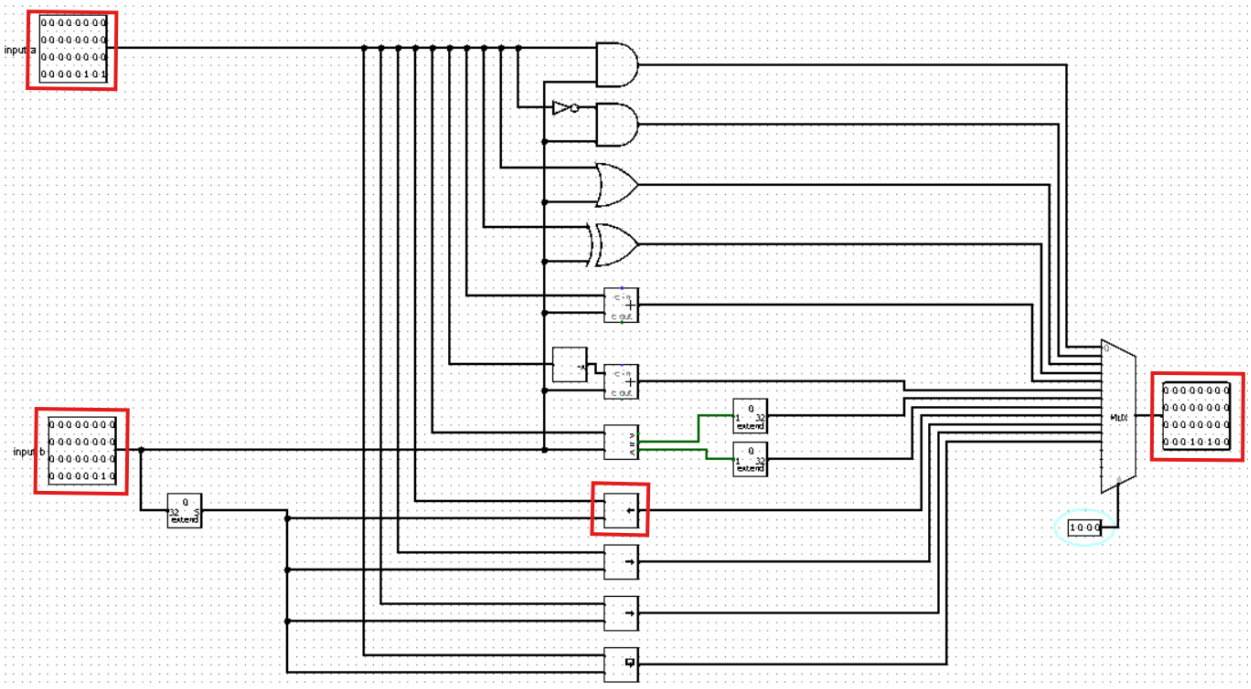


Figure 10: SLL operation performed in the ALU.

I.I.X SRL

The SRL (Shift Right Logical) is the tenth operation of the ALU, it performs a bitwise logical right shift on the 32-bit value from data bus A. The shift amount is determined by an 8-bit immediate value (`imm8`), typically derived from the instruction's operand field or a portion of data bus B. During the shift, bits are moved to the right, and zeroes are inserted into the vacated most significant bit positions. This functional unit is connected to the main MUX's tenth input (index 9), corresponding to selection inputs '1001'. For example, if 32-bit data bus A holds `00000000000000000000000000010100` (decimal 20) and the `imm8` value is 2, setting the selection lines to '1001' directs the shifted result, `00000000000000000000000000000101` (decimal 5), to the ALU output.

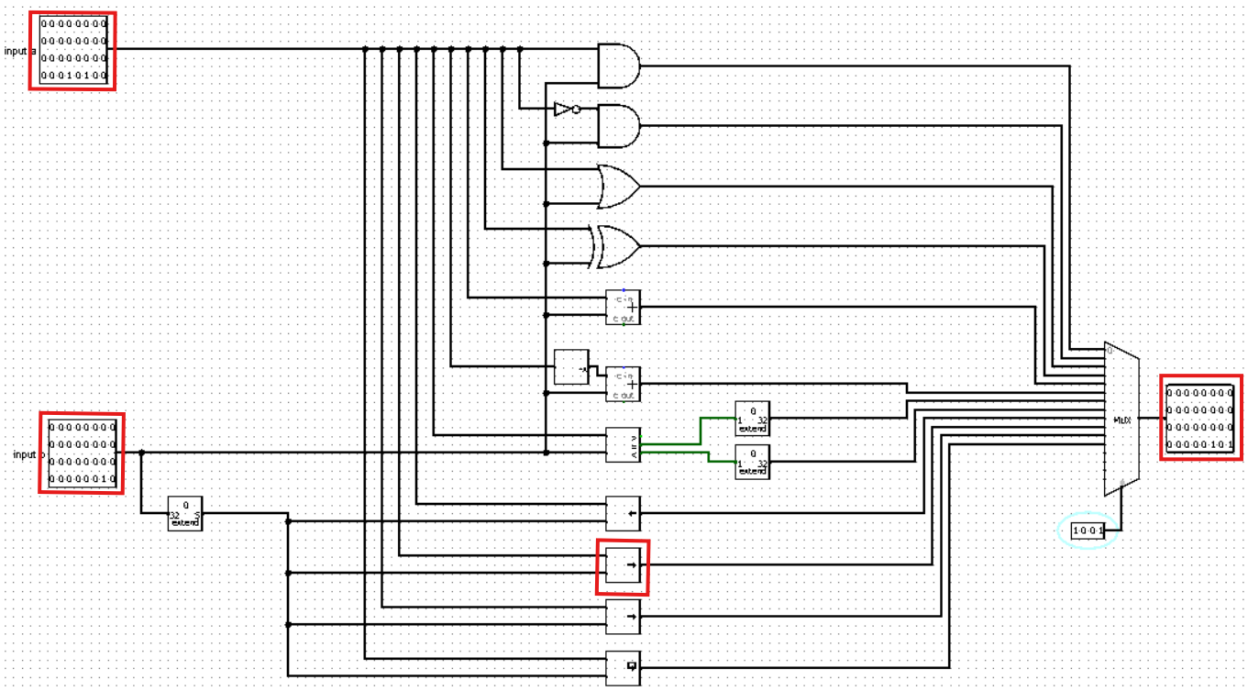


Figure 11:SRL operation performed in the ALU.

I.I.XI SRA

The SRA (Shift Right Arithmetically) is the eleventh operation of the ALU, it performs a bitwise arithmetic right shift on the 32-bit signed value from data bus A. The shift amount is determined by an 8-bit immediate value (`imm8`), typically derived from the instruction's operand field or a portion of data bus B. During the shift, bits are moved to the right, and the most significant bit (sign bit) is duplicated and inserted into the vacated most significant bit positions. This preserves the sign of the original number. This functional unit is connected to the main MUX's eleventh input (index 10), corresponding to selection inputs '1010'. For example, if 32-bit data bus A holds `11111111111111111111111111111000` (decimal -8) and the `imm8` value is 2, setting the selection lines to '1010' directs the shifted result, `11111111111111111111111111111110` (decimal -2).

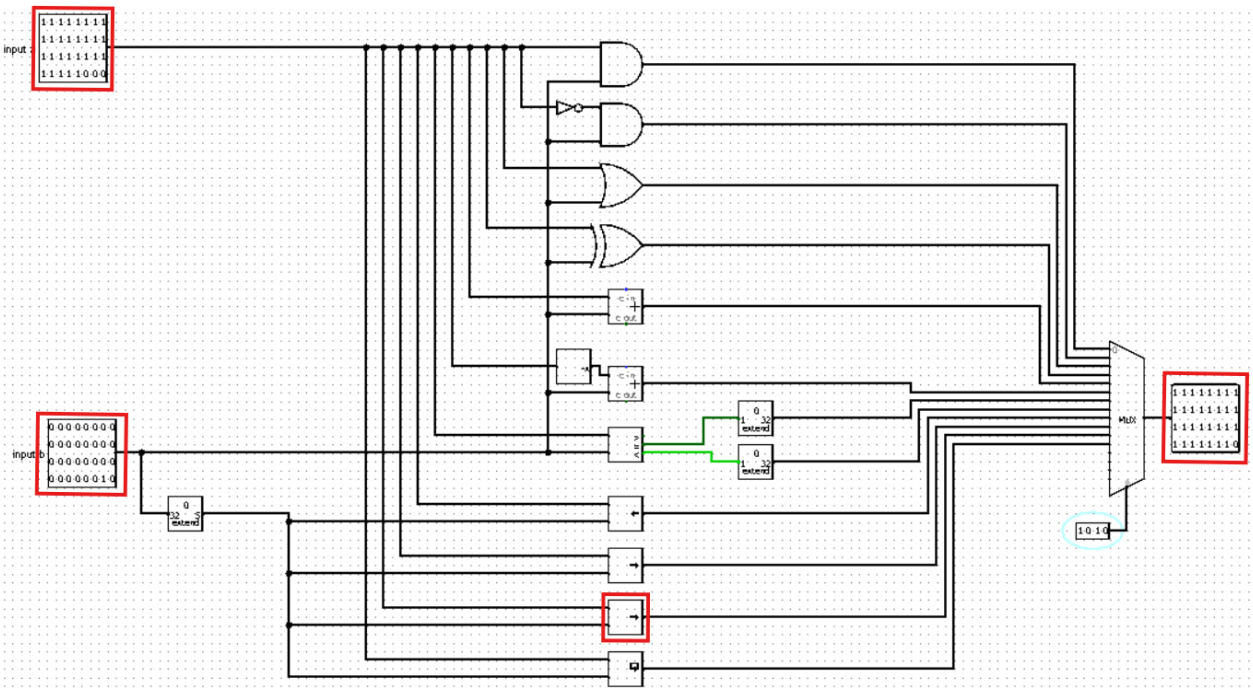


Figure 12: SRA operation performed in the ALU.

I.I.XII ROR

The ROR (Rotate Right) is the twelfth and last operation of the ALU, it performs a bitwise right rotation on the 32-bit value from data bus A. The rotation amount is determined by an 8-bit immediate value (`imm8`), typically derived from the instruction's operand field or a portion of data bus B. During a right rotation, bits shifted out from the least significant bit positions are re-inserted into the most significant bit positions, effectively preserving all original bits within the word. This functional unit is connected to the main MUX's twelfth input (index 11), corresponding to selection inputs '1011'. For example, if 32-bit data bus A holds `00010010001101000101011001111000` (representing `0x12345678`) and the `imm8` value is 4, setting the selection lines to '1011' directs the rotated result, `10000001001000110100010101100111`, to the ALU output.

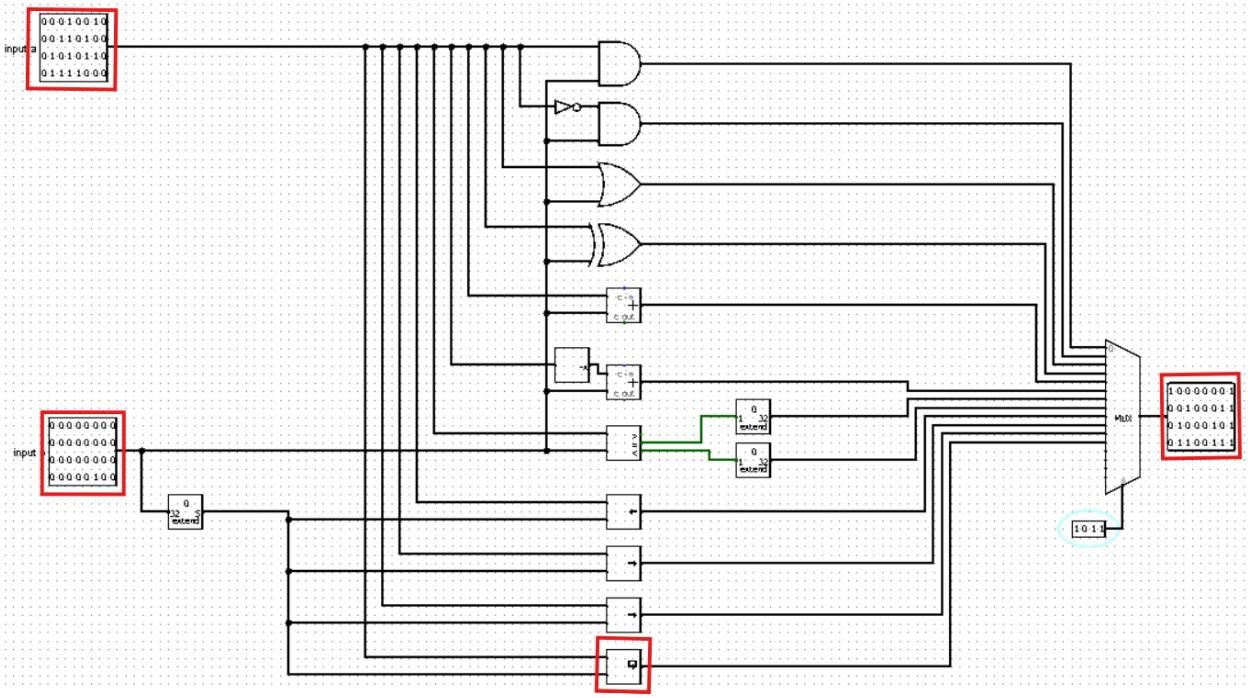


Figure 13: ROR operation performed in the ALU.

I.II Register File

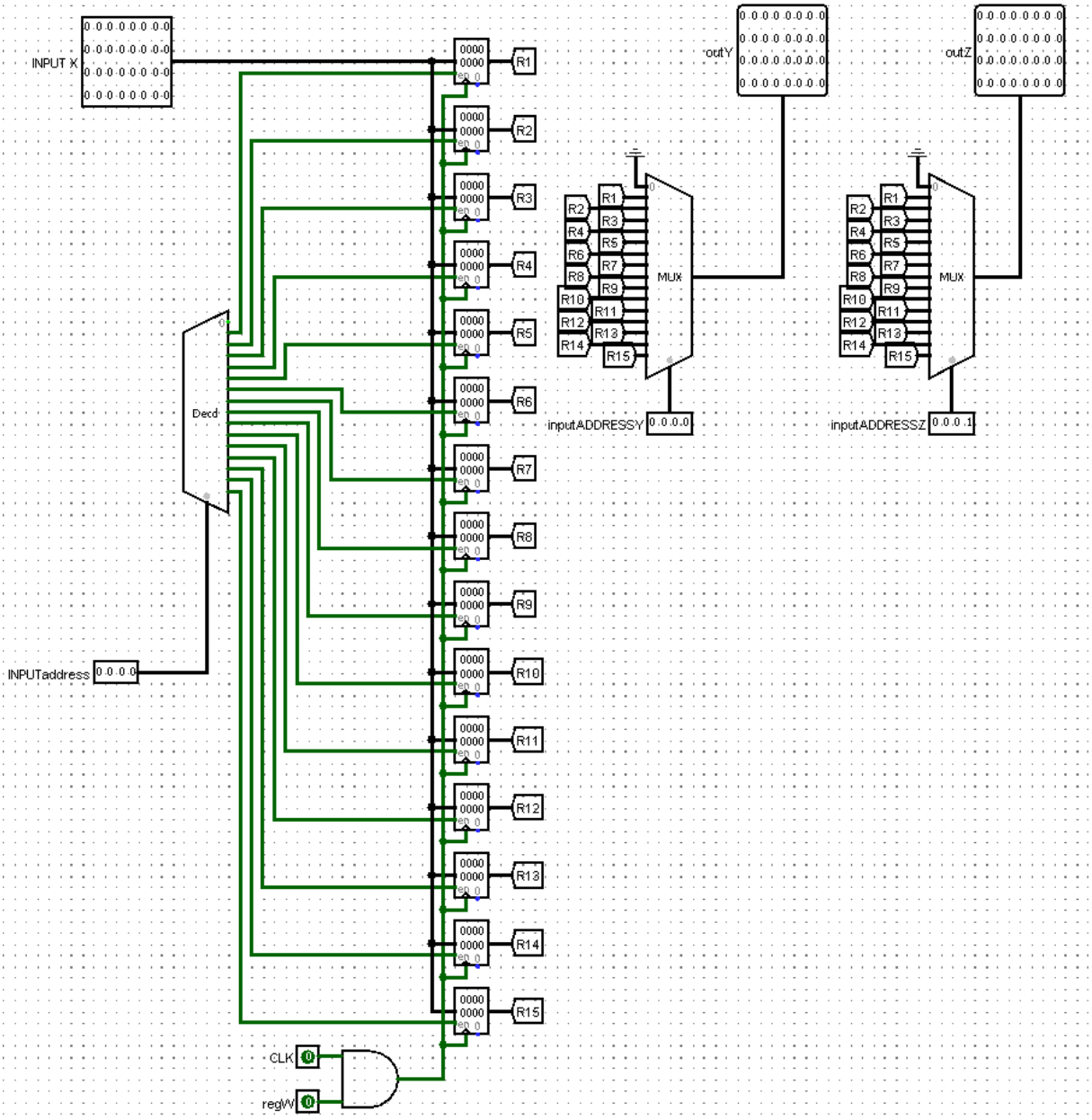


Figure 14: The Register File.

The implementation of the single-cycle processor begins with the architectural design and development of the **register file**. A register file provides a small, simple, high-speed storage device within the processor, used to maintain temporary data related to the current computation.

I.II.I Register File Design: Write Operation

The design of the register file commences with the construction of the registers designated for writing. Our register file contains sixteen registers, numbered 0 through 15 (R0 to R15), R0 is hard wired to zero input, each capable of storing 32 bits of data.

To select which register to write to, a 4-to-16 decoder (labeled "Decd") is used, taking a 4-bit selection input labeled Input Address. This Input Address input determines the destination register for a write operation. The Write input serves as the write enable signal for the entire register file. The outputs of the decoder are connected directly to the individual enable inputs of each corresponding register (R0 through R15). This means that when Write is asserted high and Input Address selects a specific register, data will be written to that register.

Data is brought to the registers via a 32-bit data bus labeled Input X. All registers are connected to a main clock, labeled CLK, which controls the writing operation within the register file.

The writing operation can be tested as follows: to write a value, the Write input must be enabled (set to 1). For example, to write the number 7 to R1, we would enter 0001 on the InputAddress selection input and 000....111 on Input X. After applying a clock pulse, the value 7 would be stored in register one.

I.II.II Register File Design: Read Operation

For read operations, the outputs of the registers are connected to the inputs of two separate 16-to-1 multiplexers (labeled "MUX"). Each multiplexer serves as a read port and receives a 4-bit selection input (labeled inputADDRESSY and inputADDRESSZ respectively), which designates which register's contents will be read.

When a specific inputADDRESSY or inputADDRESSZ value is applied, the corresponding multiplexer selects the data from the chosen register and drives it onto its respective 32-bit output data bus (labeled outY and outZ). This allows for simultaneous reading of two different registers. The read operation is combinational through the multiplexers; the data is available on the outY and outZ buses as soon as the inputADDRESSY and inputADDRESSZ lines are stable.

For example, to examine the read operation: if the inputADDRESSY selection input is set to 0001, the multiplexer will output the contents of R1 onto the outY bus. If R1 contains the binary value 00....111, then outY will display this value. Similarly, inputADDRESSZ can be set to 0001 to output R1's contents onto outZ.

I.II.III Control and Impact

Ultimately, the entire process of reading and writing in the register file, including the write enable signal, the InputAddress for writing, and the inputADDRESSY and inputADDRESSZ for reading, are all managed by control signals sent from the control unit of the processor. The construction of this register file significantly enhances the processor's ability to achieve high-speed data processing by enabling the CPU to rapidly access data required for arithmetic, logic, and control operations.

I.III Control Unit

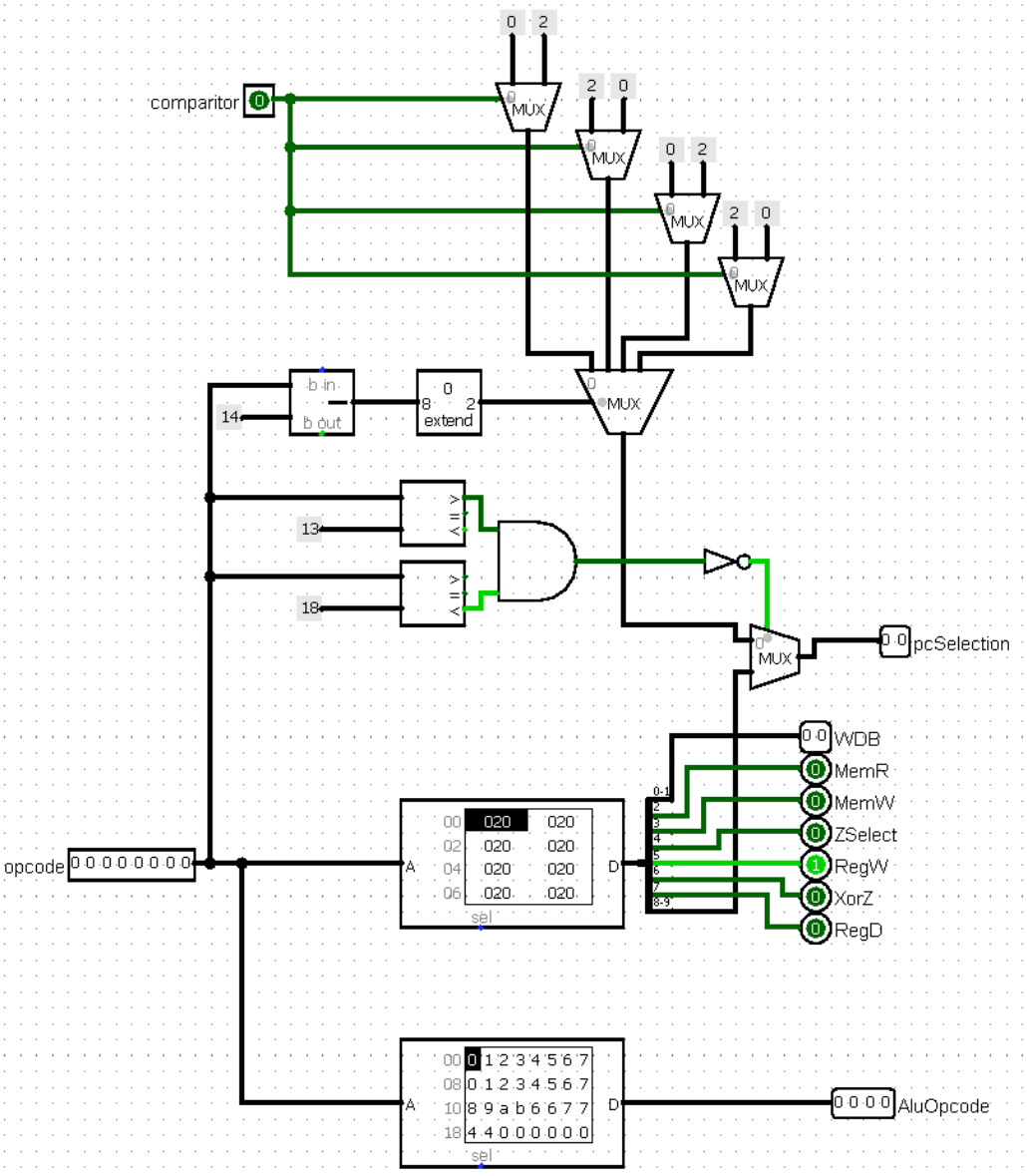


Figure 15: The Control Unit.

The shown figure above illustrates a control unit that acts as the central manager for executing all instructions from the Single Cycle Processor's instruction memory. Designed as a single block “shown in figure bellow”, it governs the entire system's operations.

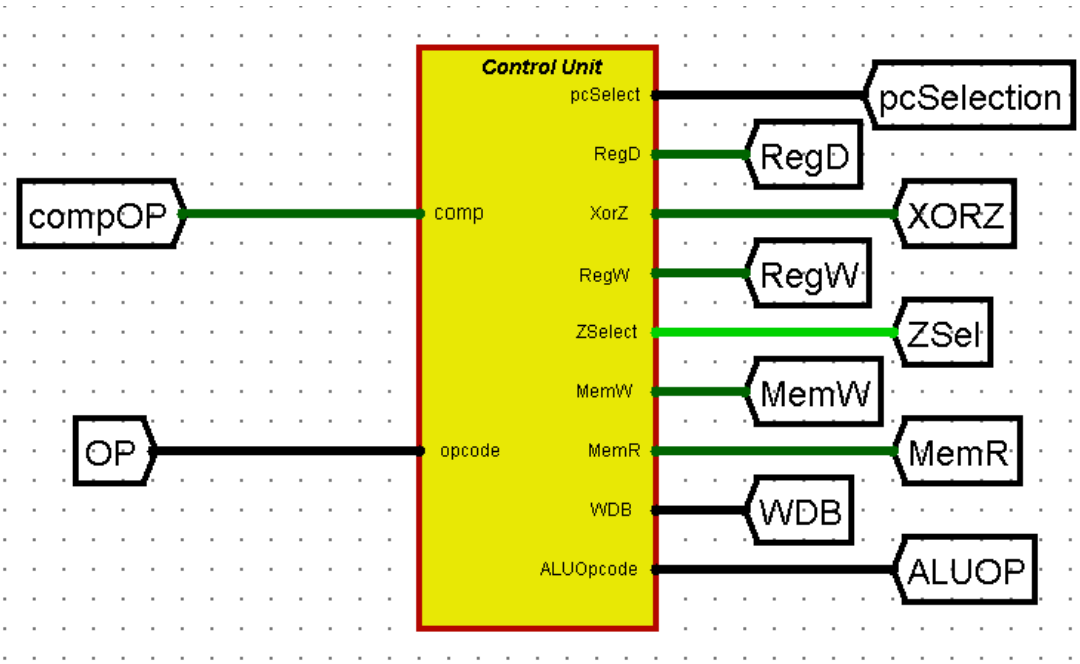


Figure 16: The Control Unit block.

This control unit is designed to interpret a 9-bit operation code (8-bit for Instruction opcode and 1 bit for comparison result of ALU) and generate the necessary control signals to manage the data-path of the processor. It consists of three primary sections: a main control logic section, an ALU control section and a Branch PC selection Logic handler.

I.III.I Main Control Logic

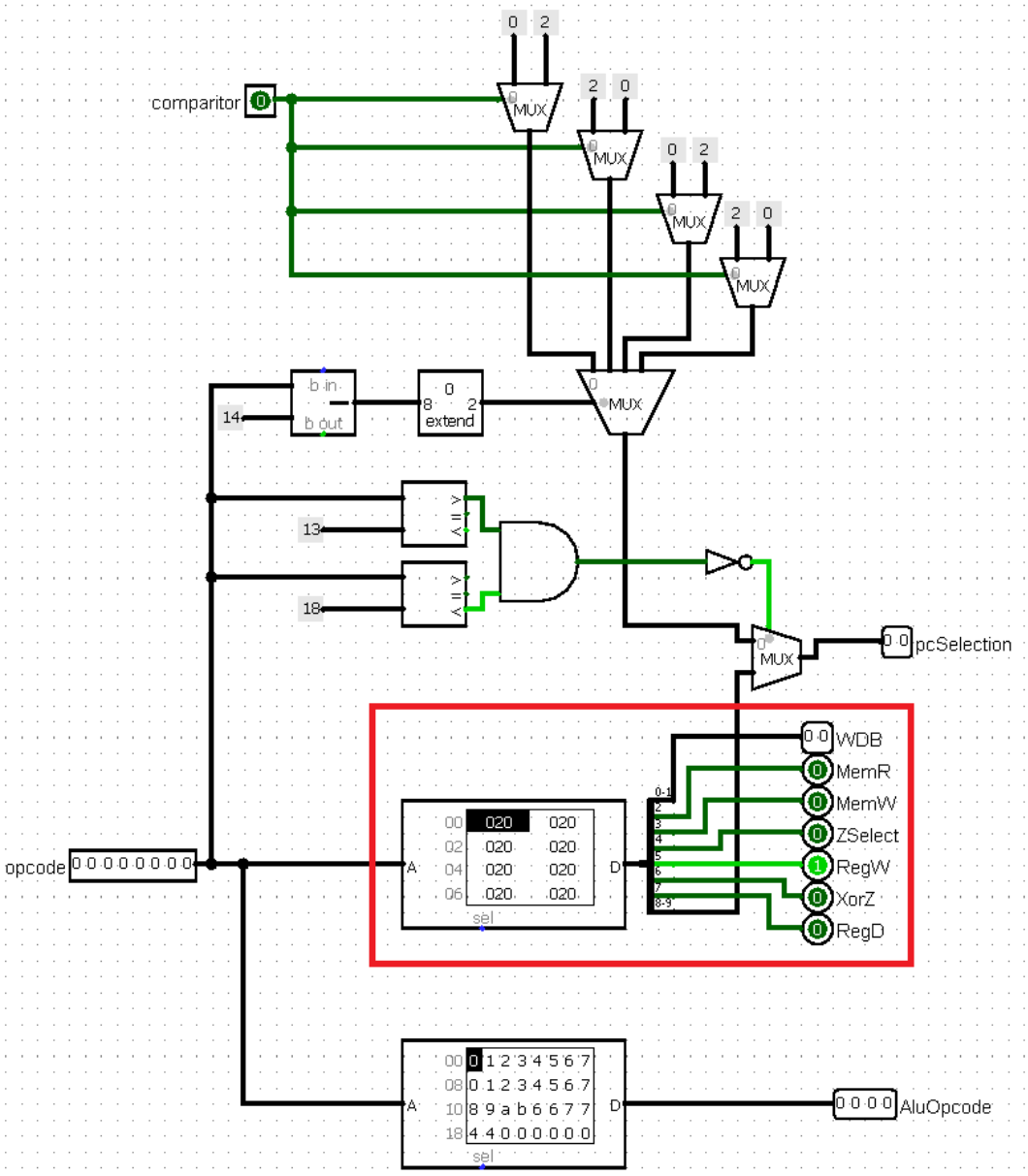


Figure 17: Main Control Logic.

The main control section is responsible for generating most of the control signals that direct the operation of the data-path, excluding the specific ALU operation. It takes the opcode as input to a Read-Only Memory (ROM) module. This ROM is programmed to output a specific set of control signals based on the instruction being executed. The outputs from this section manage data selection for the register file, memory operations, and program counter modifications in some cases, explaining the need for the multiplexer when doing branch handling.

- **Inputs:**
  Opcode + comparison: A 9-bit input that specifies the Control Signal to be outputted for current instruction being executed.

- **Outputs:**

  pcSelection: 2-bits, determines how the next instruction's PC address will be calculated.

pcSelection	Selection
00	Pc = Pc + 1
01	Pc = Pc + (imm16 << 1)
10	Pc = Pc + (imm8 << 1)

  RegD: 1-bit, determines whether we write into RX, or R15 (used in JAL instruction).

RegD	Selection
0	Rx
1	R15

  XORZ: 1-bit, determines whether BusZ will be Showing the values in RX or RZ.

XORZ	Selection
0	Rz
1	Rx

  RegW: 1 bit, Enables or disables writing to the register file.

RegW	Selection
0	Disable write
1	Enable write

ZSel: 1 bit, determines whether the second operand will be coming from BusZ or imm8.

ZSel	Selection
0	BusZ
1	Imm8

MemW:1 bit, Enables or disables writing to data memory. '1' would signal a memory write.

MemW	Selection
0	Disable write
1	Enable write

MemR: 1 bit, Enables or disables reading from data memory. '1' would signal a memory read.

MemR	Selection
0	Disable read
1	Enable read

WDB: 2 bits, determines which input is assigned to the input register file to be written at RX if RegW is 1.

WDB	Selection
00	ALU Res
01	Memory
10	Pc + 3

Below is the control signal for each instruction along with its corresponding hexadecimal encoding

Instruction	opcode	pcSelection	RegD	XORZ	RegW	ZSel	MemW	MemR	WDB	hex
AND	0	00	0	0	1	0	0	0	0	020
CAND	1	00	0	0	1	0	0	0	0	020
OR	2	00	0	0	1	0	0	0	0	020
XOR	3	00	0	0	1	0	0	0	0	020
ADD	4	00	0	0	1	0	0	0	0	020
NADD	5	00	0	0	1	0	0	0	0	020
SEQ	6	00	0	0	1	0	0	0	0	020
SLT	7	00	0	0	1	0	0	0	0	020
ANDI	8	00	0	0	1	1	0	0	00	030
CANDI	9	00	0	0	1	1	0	0	00	030
ORI	10	00	0	0	1	1	0	0	00	030
XORI	11	00	0	0	1	1	0	0	00	030
ADDI	12	00	0	0	1	1	0	0	00	030
NADDI	13	00	0	0	1	1	0	0	00	030
SEQI	14	00	0	0	1	1	0	0	00	030
SLTI	15	00	0	0	1	1	0	0	00	030
SLL	16	00	0	0	1	1	0	0	00	030
SRL	17	00	0	0	1	1	0	0	00	030
SRA	18	00	0	0	1	1	0	0	00	030
ROR	19	00	0	0	1	1	0	0	00	030
BEQ	20	00	0	1	0	0	0	0	00	040
BNE	21	00	0	1	0	0	0	0	00	040
BLT	22	00	0	1	0	0	0	0	00	040
BGE	23	00	0	1	0	0	0	0	00	040
LW	24	00	0	0	1	1	0	1	01	035
SW	25	00	0	1	0	1	1	0	00	058
J	26	01	0	0	0	0	0	0	00	100
JAL	27	01	1	0	1	0	0	0	10	1A2

I.III.II ALU Control

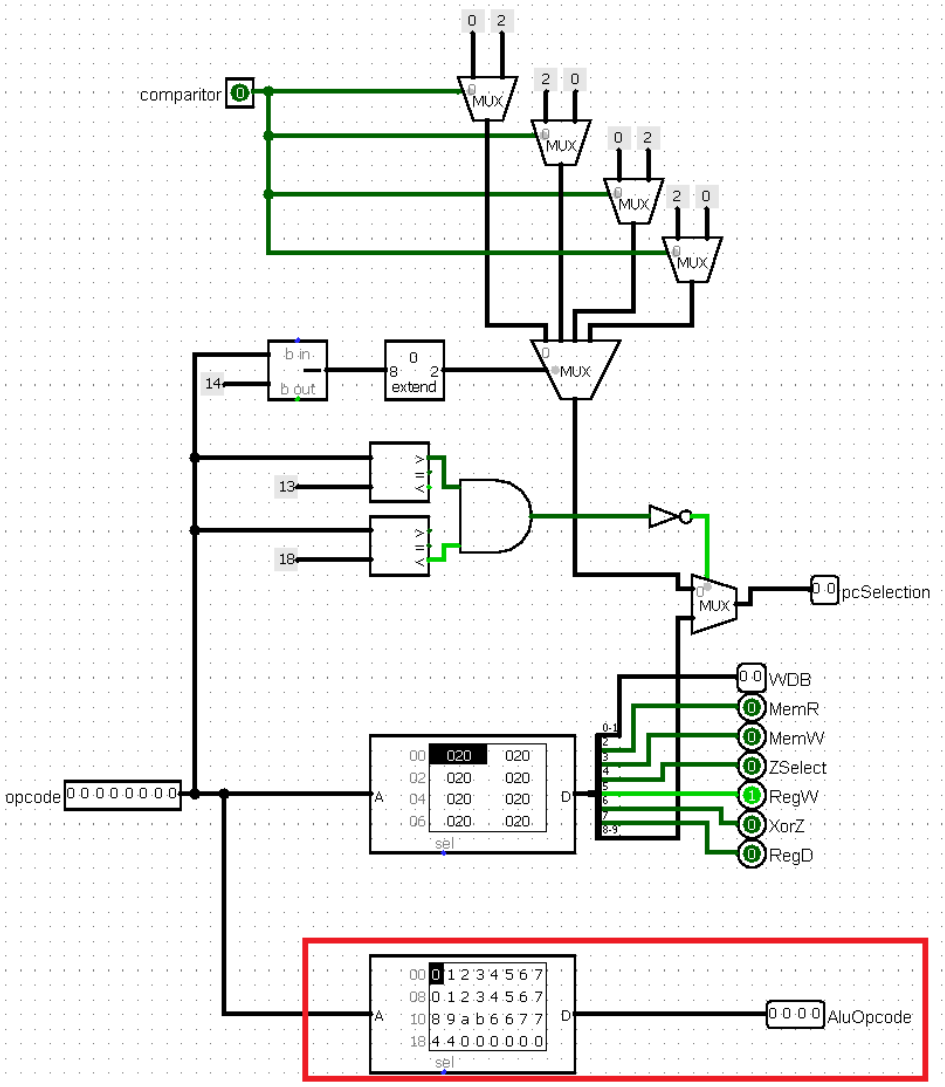


Figure 18: ALU Control.

This lower section is specifically for controlling the Arithmetic Logic Unit (ALU). It generates a 4-bit code that instructs the ALU on which operation to perform.

- **Inputs:**
  opcode: The instruction's opcode is used as an address into a dedicated ROM for ALU operations.
- **Outputs:**

  AluOpcode: A 4-bit signal that is sent to the ALU to select the specific arithmetic or logical function to run.

Instruction	opcode	ALUOP	Hexadecimal
AND	0	0000	0
CAND	1	0001	1
OR	2	0010	2
XOR	3	0011	3
ADD	4	0100	4
NADD	5	0101	5
SEQ	6	0110	6
SLT	7	0111	7
ANDI	8	0000	0
CANDI	9	0001	1
ORI	10	0010	2
XORI	11	0011	3
ADDI	12	0100	4
NADDI	13	0101	5
SEQI	14	0110	6
SLTI	15	0111	7
SLL	16	1000	8
SRL	17	1001	9
SRA	18	1010	A
ROR	19	1011	B
BEQ	20	0110	6
BNE	21	0110	6
BLT	22	0111	7
BGE	23	0111	7
LW	24	0100	4
SW	25	0100	4
J	26	0000	0
JAL	27	0000	0

I.III.III Branch and Jump Logic

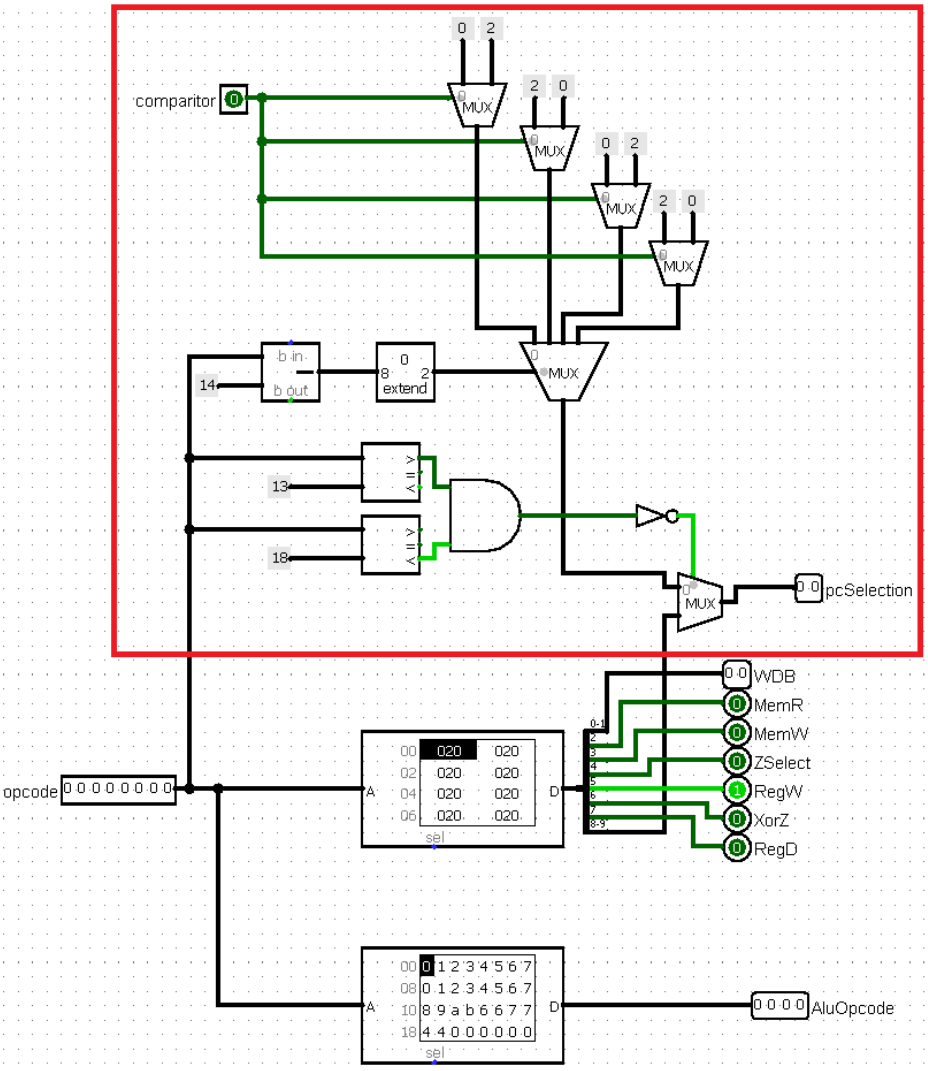


Figure 19: Branch and Jump Logic.

A dedicated portion of the control unit handles the logic for altering the program PC selection in branches Instructions specifically.

- **Inputs:**

  opcode: The instruction's opcode is used to determine if a branch or jump should even be considered.
  Comparator: result of ALU comparison Operation used in finding whether branching comparator condition is true or not such as to be able to decide the PC selection value.

constants :0x14 = 20 decimal, 0x13 = 19 decimal, 0x14 = 24 decimal. Used to subtract from the opcode so it fits into MUX 0-3 Selection range.

Outputs:

PcSelection: 2 bits, this signal selects the next value for the Program Counter. For example, it might select between `PC+1` and a calculated branch/jump address. The logic, which includes a series of multiplexers controlled by instruction bits and an `AND` gate, are the necessary conditions to make the selection.

I.IV Single Cycle Processor

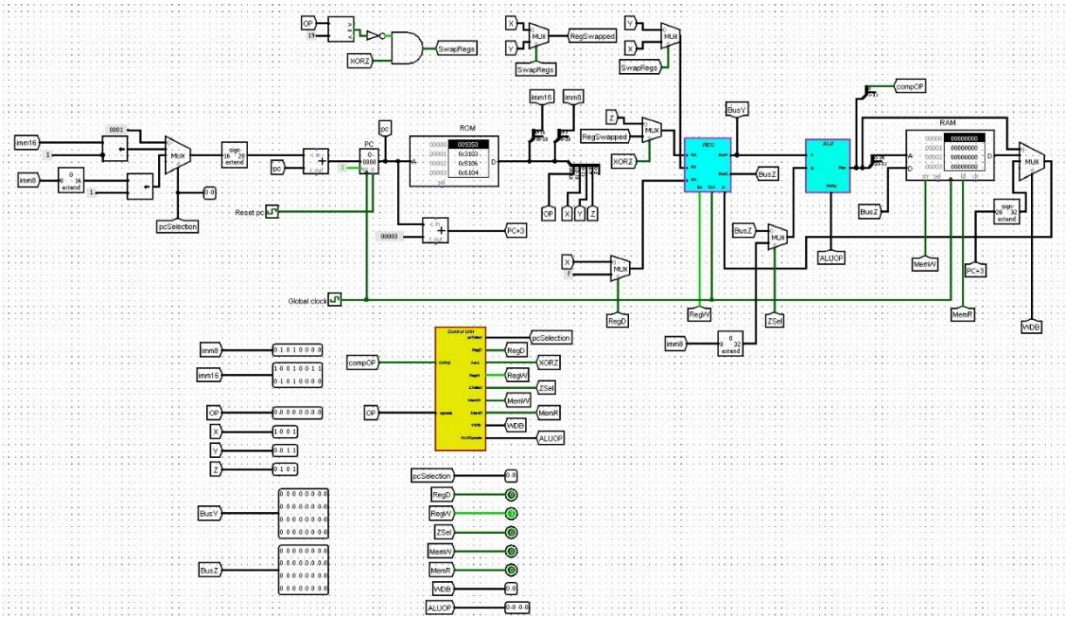


Figure 20: The data path.

The diagram above illustrates the architecture of the 32-bit single cycle processor. As shown, each instruction is executed in one single clock cycle. It's a complete cycle, in which every operation follows a logical data path through the main units of functional components, going from data fetching to generating instruction signals to executing the instructions based on the signals generated, finally to memory access and fetching instructions for the next cycle.

I.IV.I Data Fetching

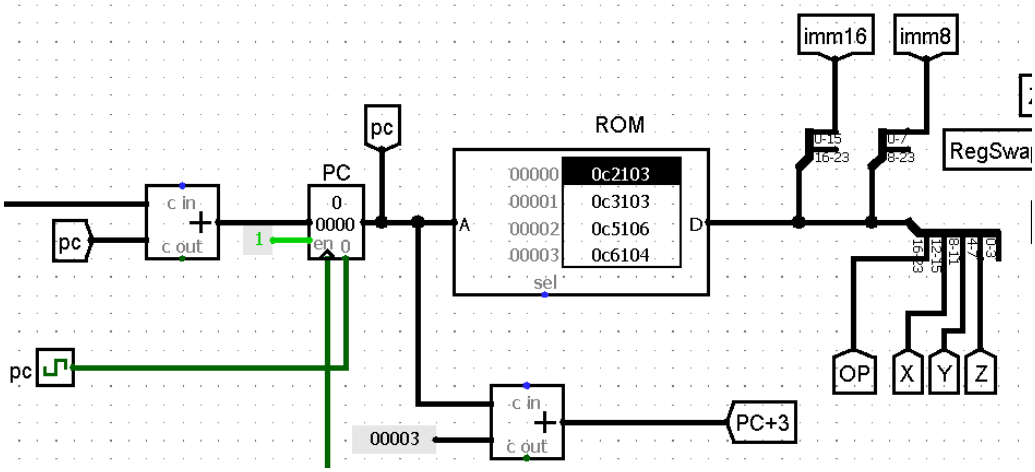


Figure 21: The ROM.

The execution cycle begins when the Program Counter (PC) provides a 20-bit address to the ROM. Which is already loaded with the program's instructions, which fetches the 24-bit instruction at the given address. Then it divides the Instruction into OP, X, Y, Z, IMM8, IMM16 to be used by the program later in the execution phase.

I.IV.II Generating Signals

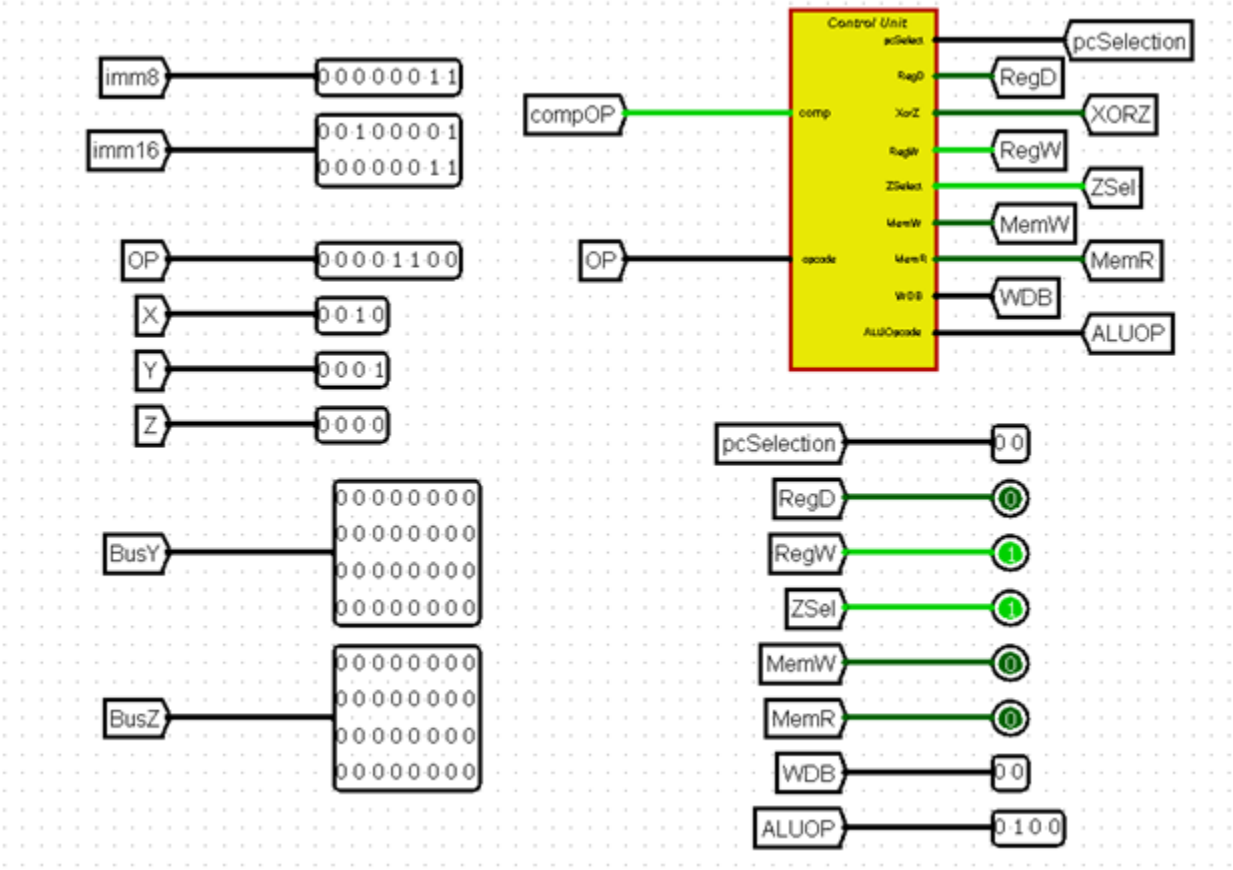


Figure 22: Control signals.

The fetched instruction opcode is sent into the CU to generate the appropriate Control Signal for each opcode. Where the signal depends directly on the opcode of the instruction except for the branching instructions, where the pcSelection also depends on the ALU comparison result in case of successfully branching.

I.IV.III Executing

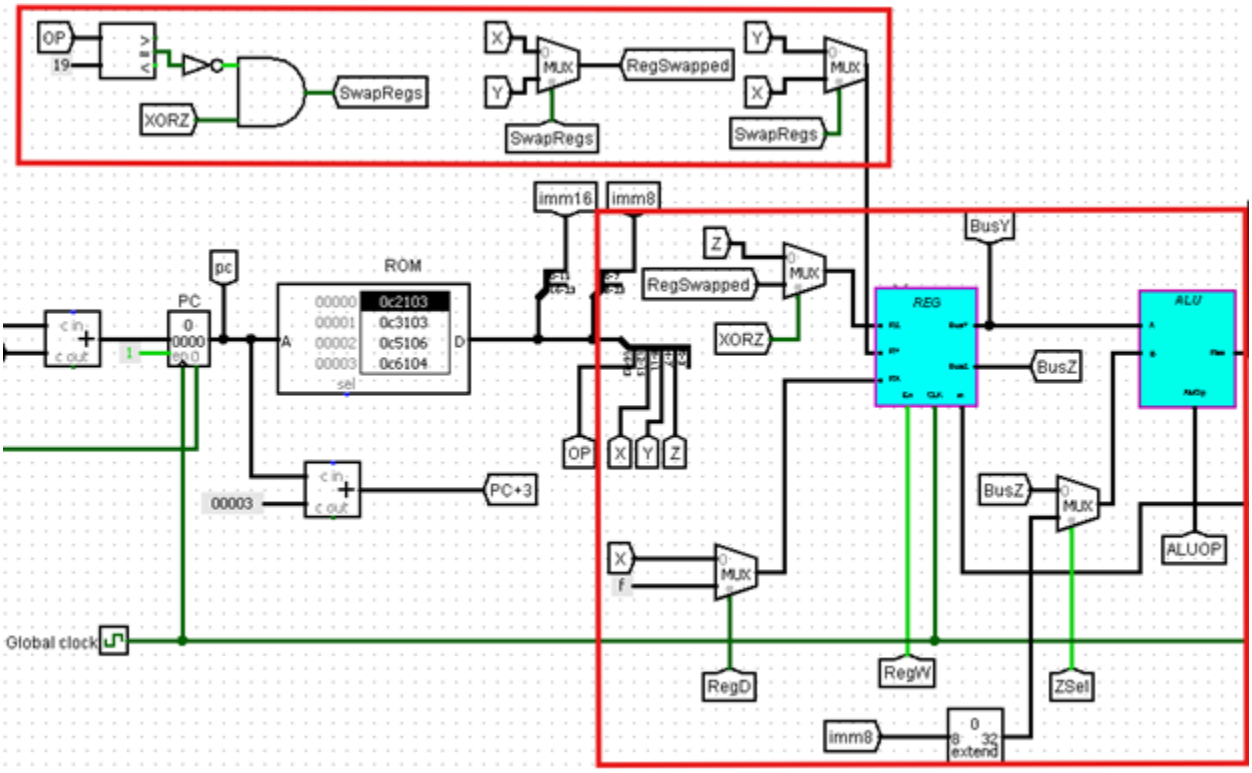


Figure 23: Executing of ALU and Reg file.

Here, the main computation occurs in the ALU, where usually 2 operands are fetched either from the Register files or from register and Immediate value determined by the Control Signals and the Instruction operands. Where they enter the ALU and the result is then either stored into RAM in the future or returned into a register depending on the Instruction.

To determine what operands are inserted and from what register, or if they are Immediate value from Instruction, Control signals from the CU are used. Where the ALUOP Signal determines the ALU operation to be executed.

Also, the Zsel, XorZ Signals are very helpful here where Zsel helps in determining whether the second operand is Imm8 or Rz. While the XorZ determines whether to redirect Rx value into the BusZ which is very helpful in branch operations.

I.IV.IV Memory Access and Write Back

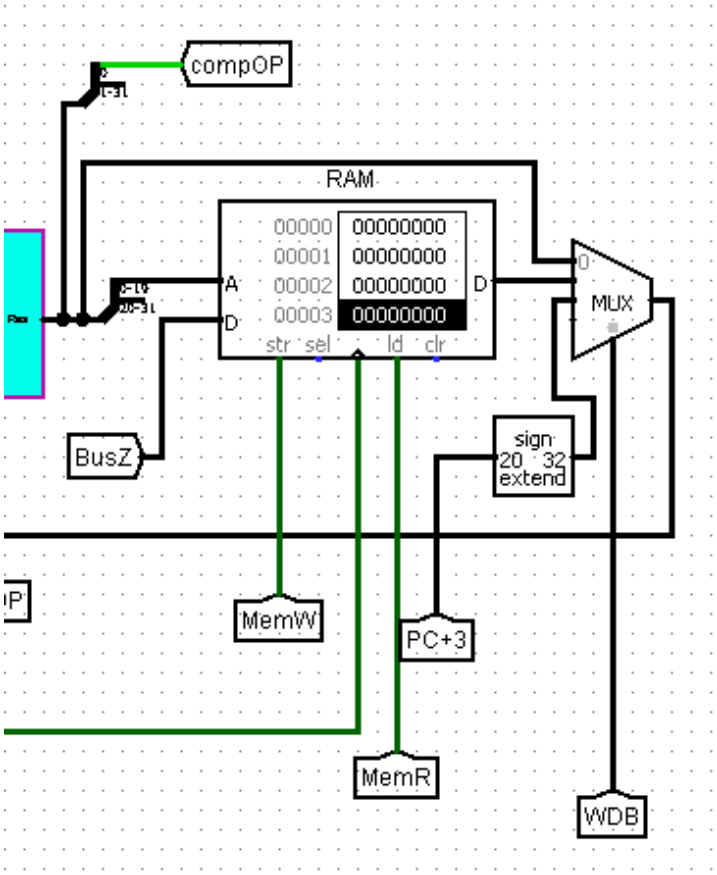


Figure 24: The RAM and write back processing.

The SCP can support writing into the RAM through MemW control signal for writing value in Rz into the RAM through the SW Instruction at position dictated by the result of ALU operation of Ry+IMM8.

the SCP can also support reading value stored in memory address dictated by ALU results of Ry+IMM8, and store that value into Rx through LW Instruction controlled by the MemR and WDB control signal.

The PC+3 value can also be stored in R15 here which can be useful in large programs when branching is used and there is need to return to a PC value and instruction in the future.

I.IV.V Fetching next Instruction

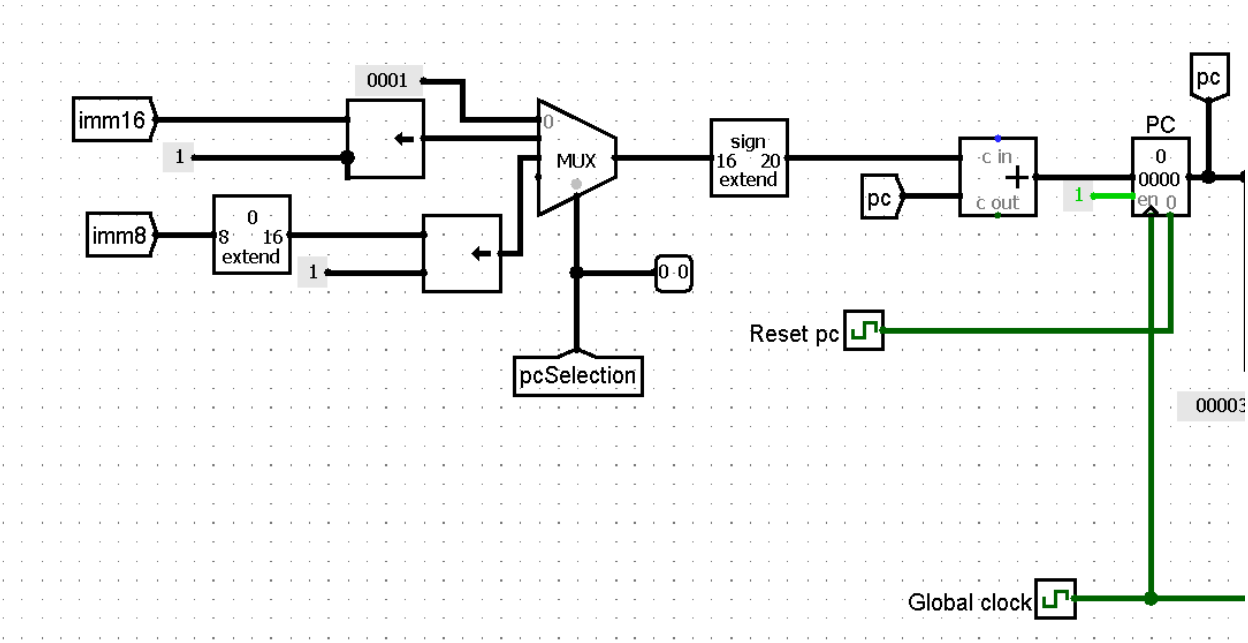


Figure 25: Pc selection processing.

The next Instruction to be executed depends on the PC value so after an instruction is done there needs to be an increment in the PC value by +1 in case of a normal instruction.

In case of a branching or jumping instruction, the next PC value is determined by incrementing 1 or, (Imm8<<1), (Imm16<<1).

What determines which of these values to increment the PC value in branching or jumping is decided by a multiplexer controlled by the pcSelection signal coming from the CU.

II Simulation and Testing

We conducted a series of tests, with each test containing a specific set of instruction examples designed to verify the functionality of selected instructions. These tests are structured to ensure that, collectively, all instructions are covered and thoroughly examined. By the end of the testing process, every instruction in the instruction set will have been tested at least once. Below, we present each instruction along with its corresponding example, and we provide debugging evidence through images to visually demonstrate and validate the correctness of the program.

II. I AND – SLT test set

The following ROM image shows the instructions contents

```
00000 0c310a 0c5103 009350 018930 026580 032650 041260 05d530 06a280 07fd80 000000
00010 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00020 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00030 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00040 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00050 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00060 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00070 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00080 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00090 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
000a0 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
```

Figure 26: Instructions in ROM for first test.

- 0x0C310A (ADDI)

  ADDI R5, R1, 3

opcode	RX	RY	Imm8
00001100	0101	0001	00000011

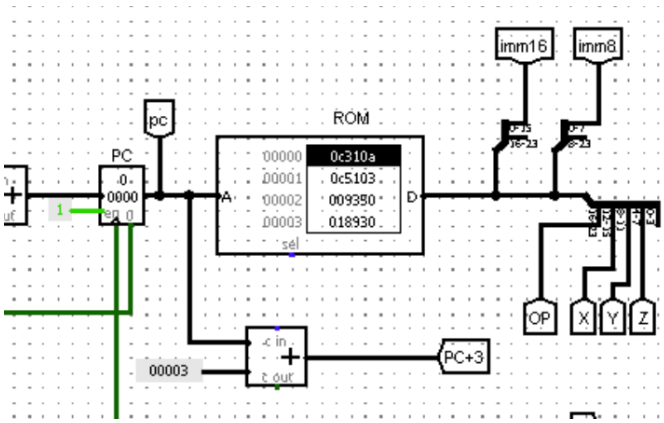


Figure 27: ADDI instruction in ROM (hex).

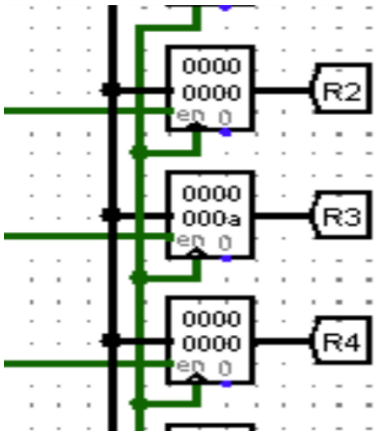


Figure 27: ADDI Reg file.

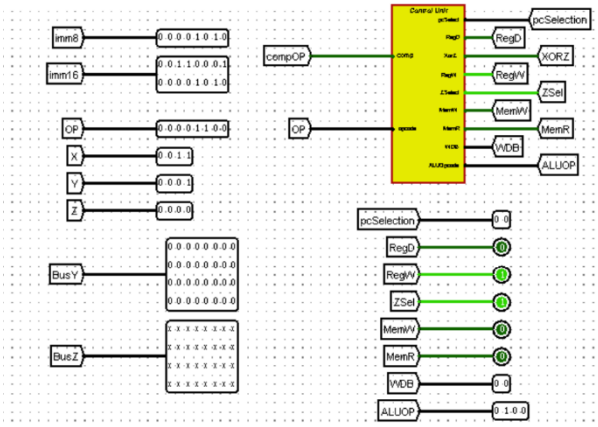


Figure 28: ADDI control signal.

- 0x0C5103 (ADDI)

  ADDI R3, R1, 10

opcode	RX	RY	Imm8
00001100	0011	0001	00001010

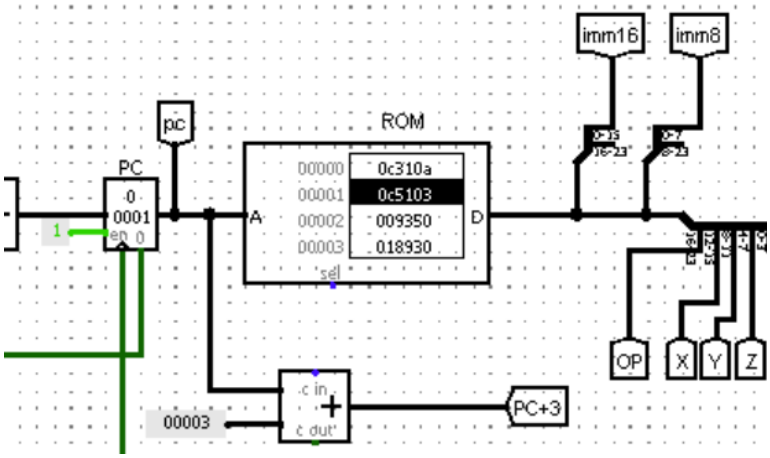


Figure 29: ADDI instruction in ROM (hex).

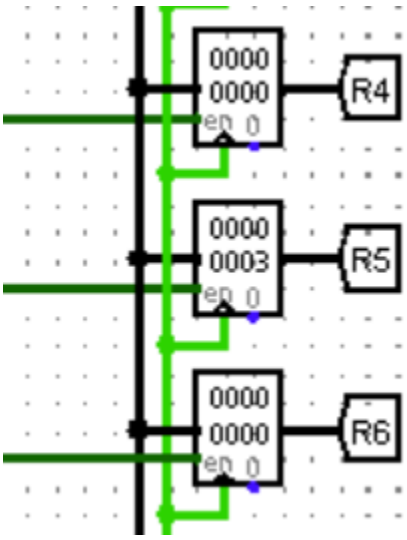


Figure 31: ADDI Reg file.

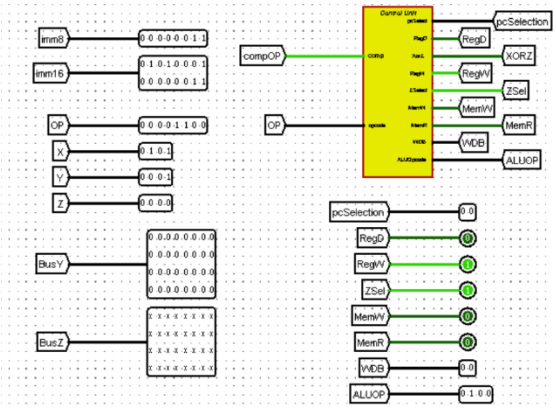


Figure 32: ADDI control signal.

- 0x009350 (AND)

  AND R9, R3, R5

opcode	RX	RY	RZ	Not used
00000000	1001	0011	0101	0000

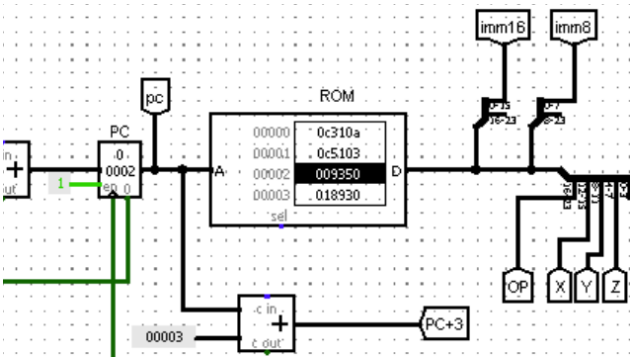


Figure 32: AND instruction in ROM (hex).

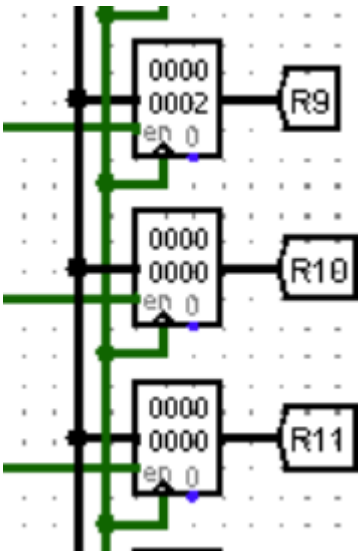


Figure 34: AND Reg file.

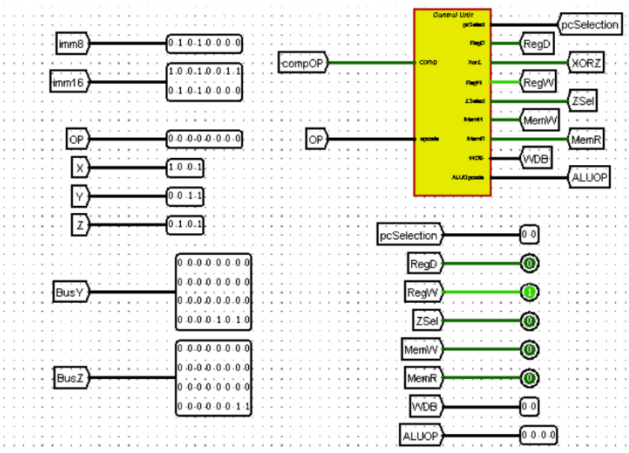


Figure 35: AND control signal.

- 0x018930 (CAND)

  CAND R8, R9, R3

opcode	RX	RY	RZ	Not used
00000001	1000	1001	0011	0000

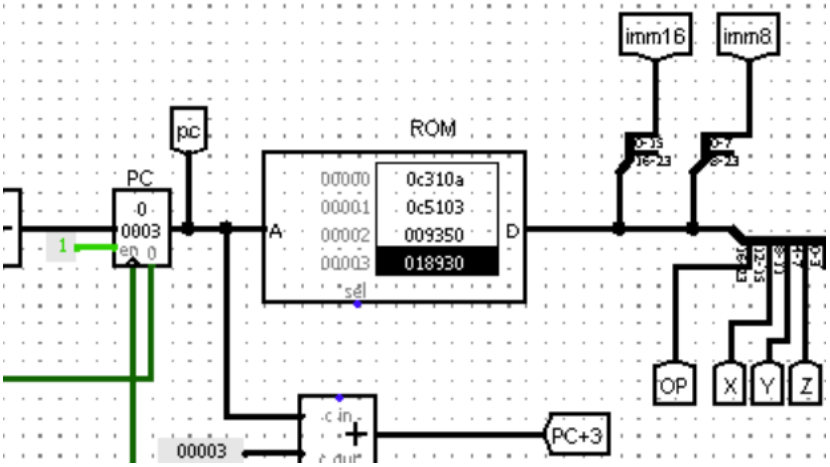


Figure 33: CAND instruction in ROM (hex).

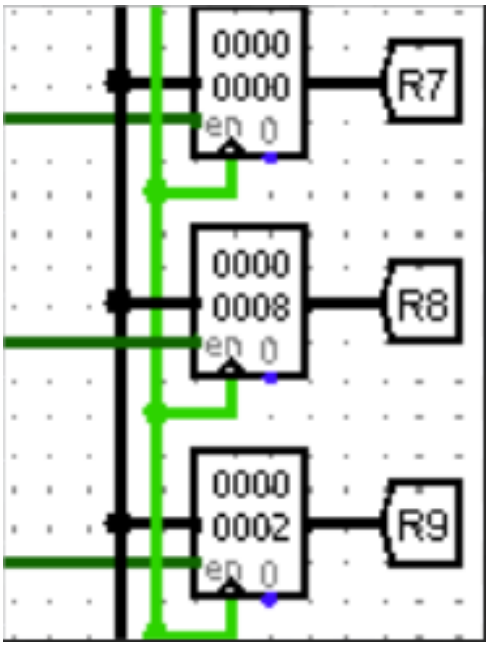


Figure 34: CAND Reg file.

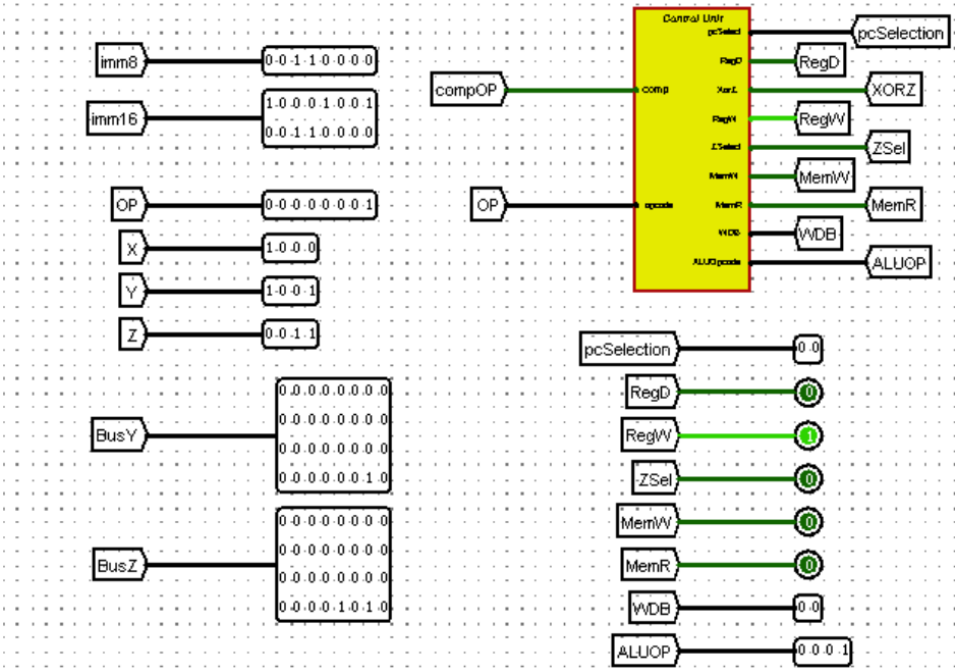


Figure 35: CAND control signal.

- 0x026580 (OR)

  OR R6, R5, R8

opcode	RX	RY	RZ	Not used
00000010	0110	0101	1000	0000

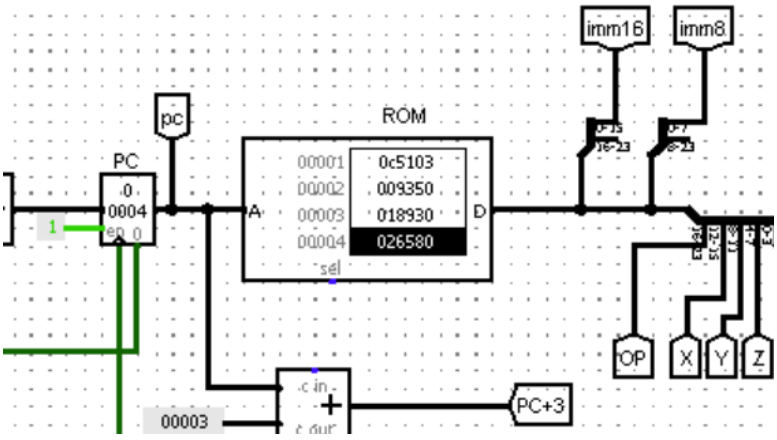


Figure 36: OR instruction in ROM (hex).

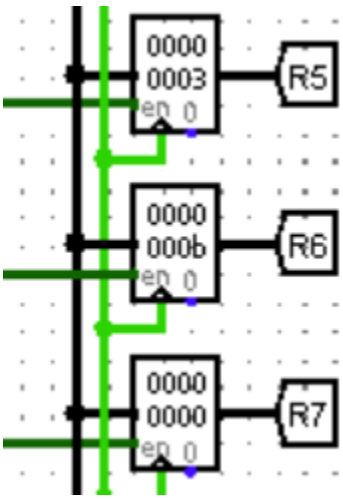


Figure 37: OR Reg file.

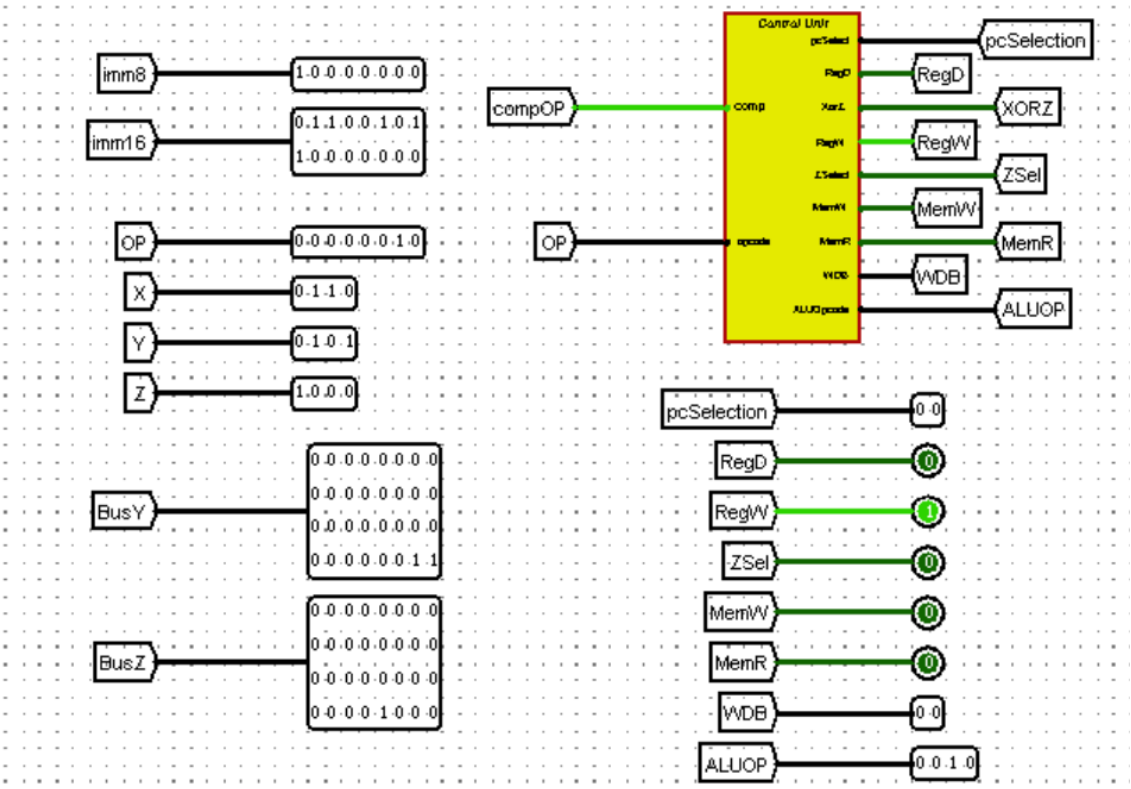


Figure 38: OR control signal.

- 0x032650 (XOR)

  OR R2, R6, R5

opcode	RX	RY	RZ	Not used
00000011	0010	0110	0101	0000

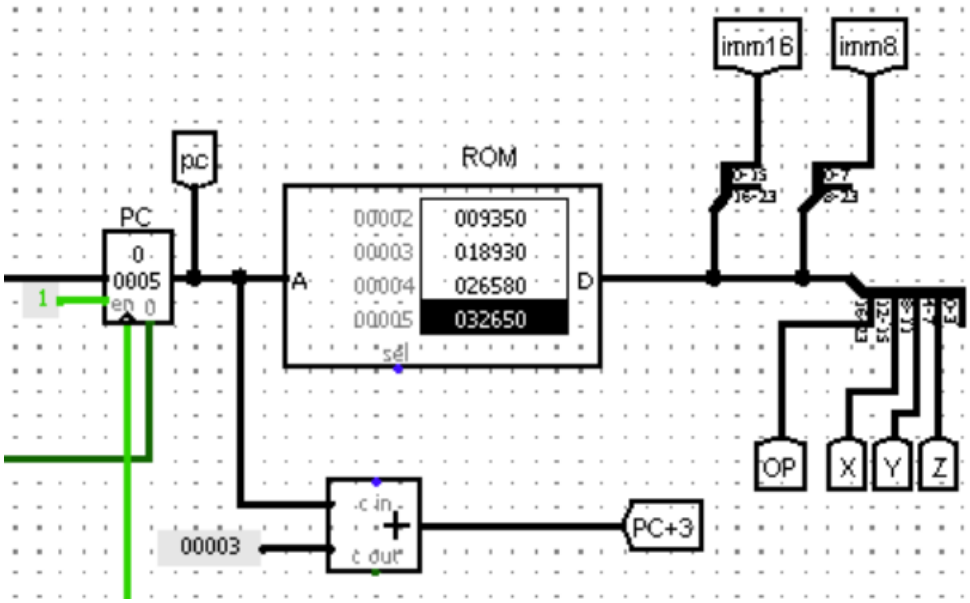


Figure 39: XOR instruction in ROM (hex).

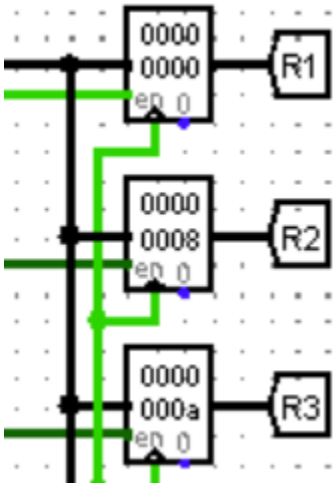


Figure 40: XOR Reg file.

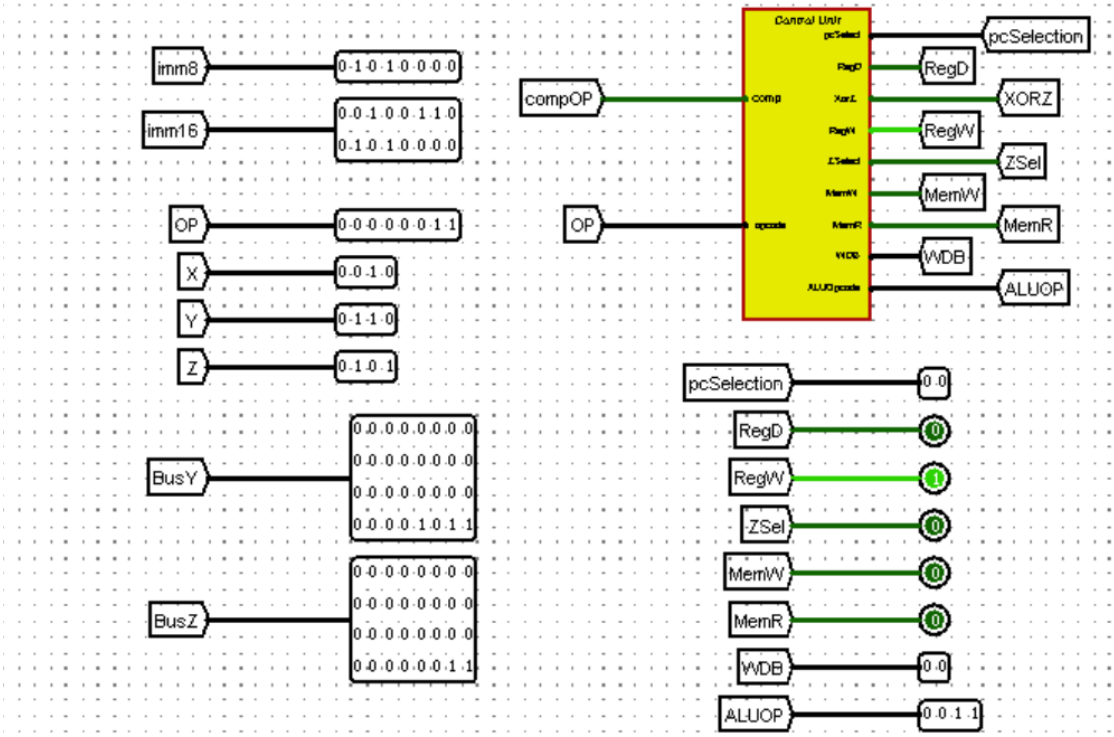


Figure 41: XOR control signal.

- 0x041260 (ADD)

  ADD R1, R2, R6

opcode	RX	RY	RZ	Not used
00000100	0001	0010	0110	0000

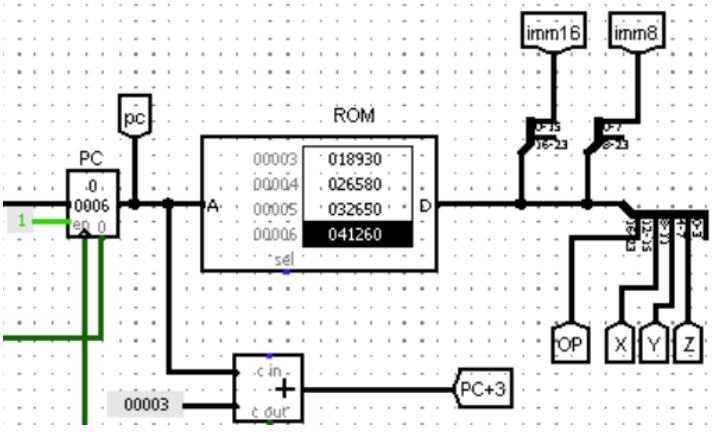


Figure 42: ADD instruction in ROM (hex).

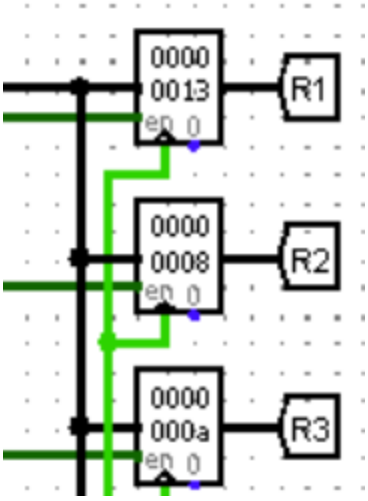


Figure 43: ADD Reg file.

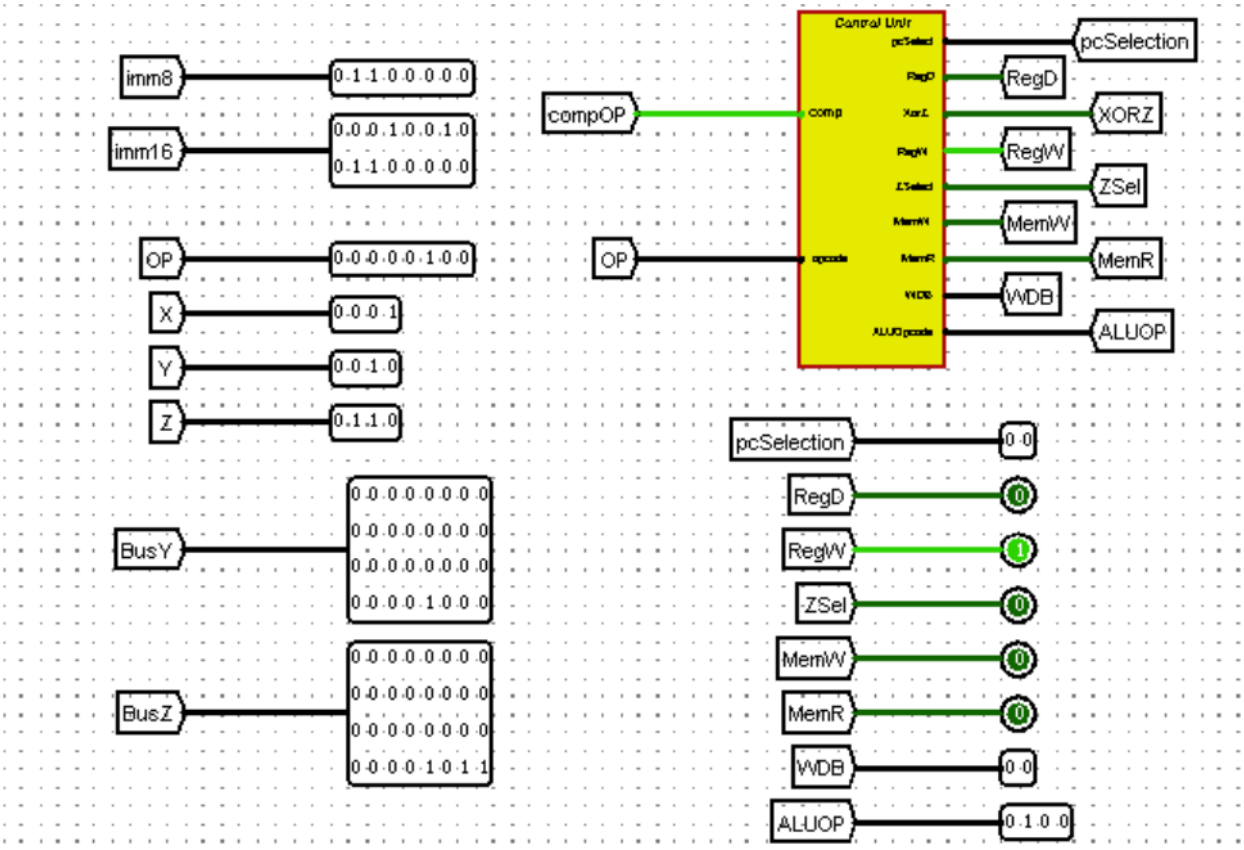


Figure 44: ADD control signal.

- 0x05D530 (NADD)

  NADD R13, R5, R3

opcode	RX	RY	RZ	Not used
00000101	1101	0101	0011	0000

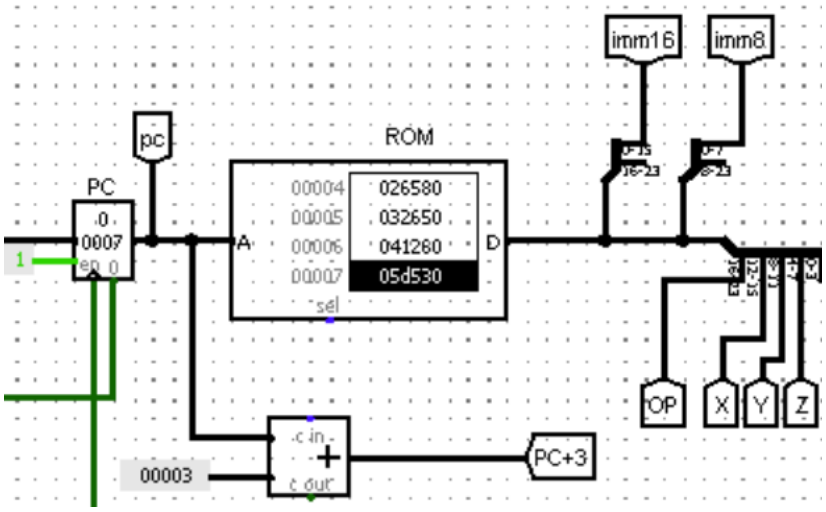


Figure 45: NADD instruction in ROM (hex).

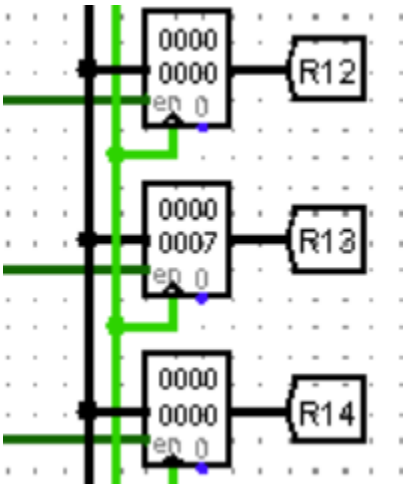


Figure 46: ADD Reg file.

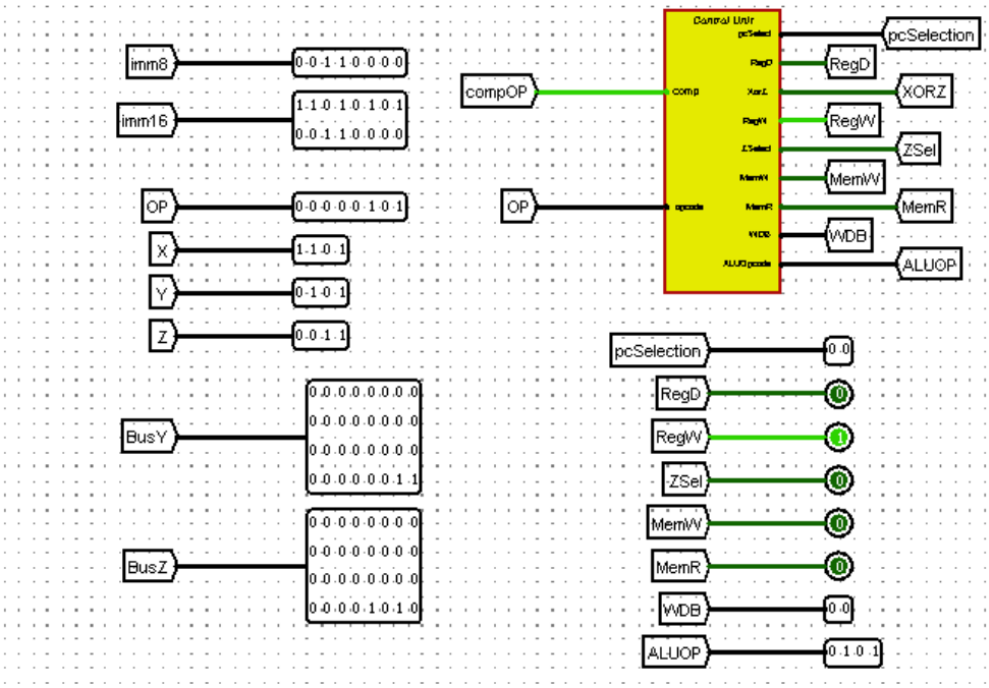


Figure 47: ADD control signal.

- 0x06A280 (SEQ)

  SEQ R10, R2, R8

opcode	RX	RY	RZ	Not used
00000110	1010	0010	1000	0000

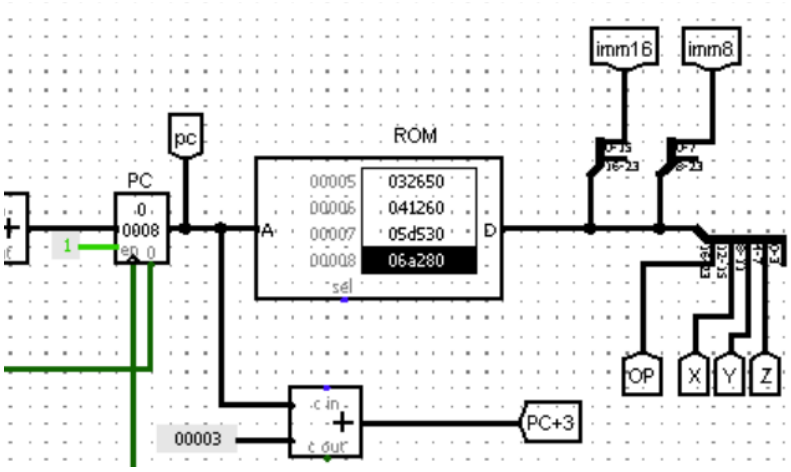


Figure 48: SEQ instruction in ROM (hex).

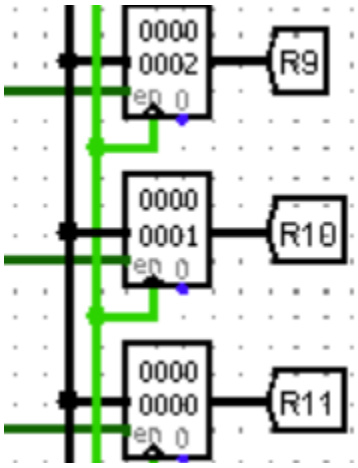


Figure 49: SEQ Reg file.

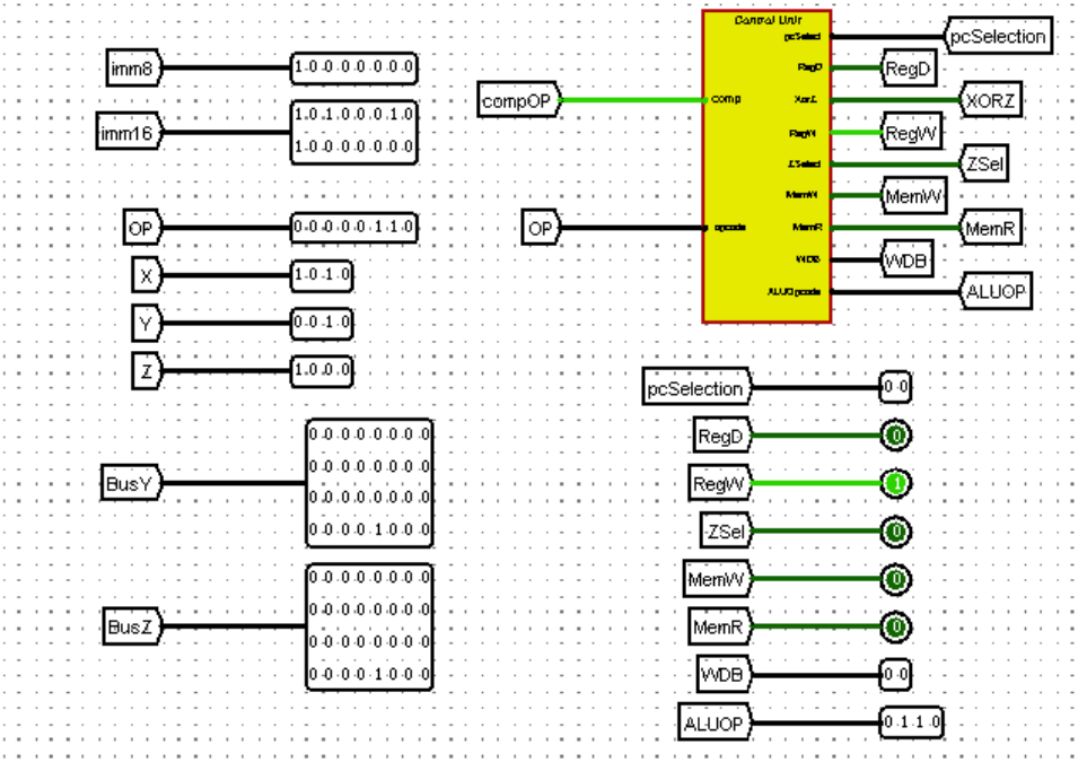


Figure 50: SEQ control signal.

- 0x07FD80 (SLT)

SLT R15, R13, R8

opcode	RX	RY	RZ	Not used
00000111	1111	1101	1000	0000

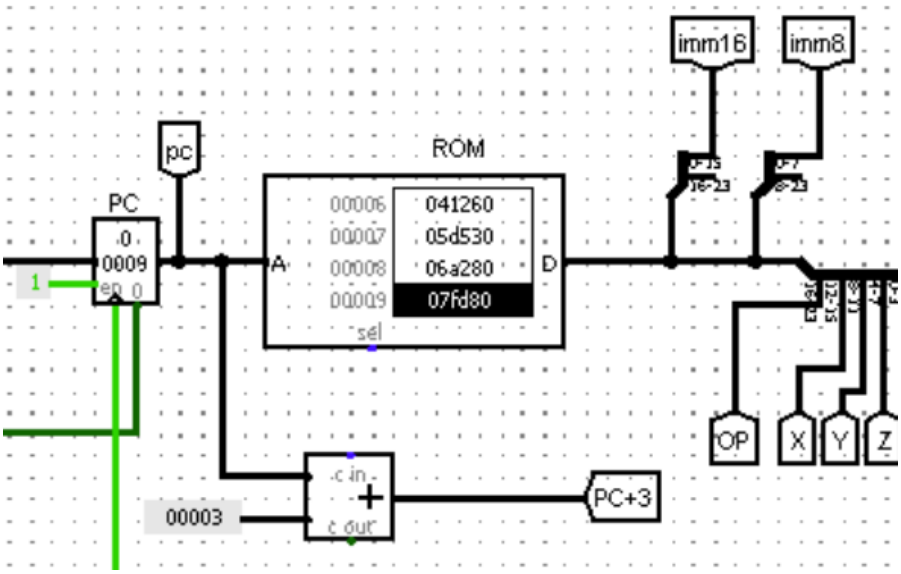


Figure 51: SLT instruction in ROM (hex).

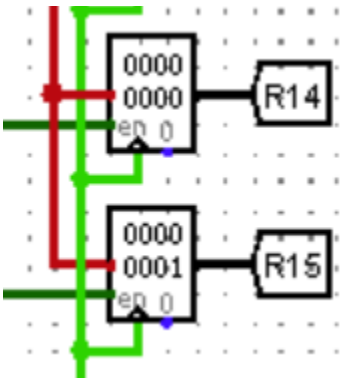


Figure 52: SLT Reg file.

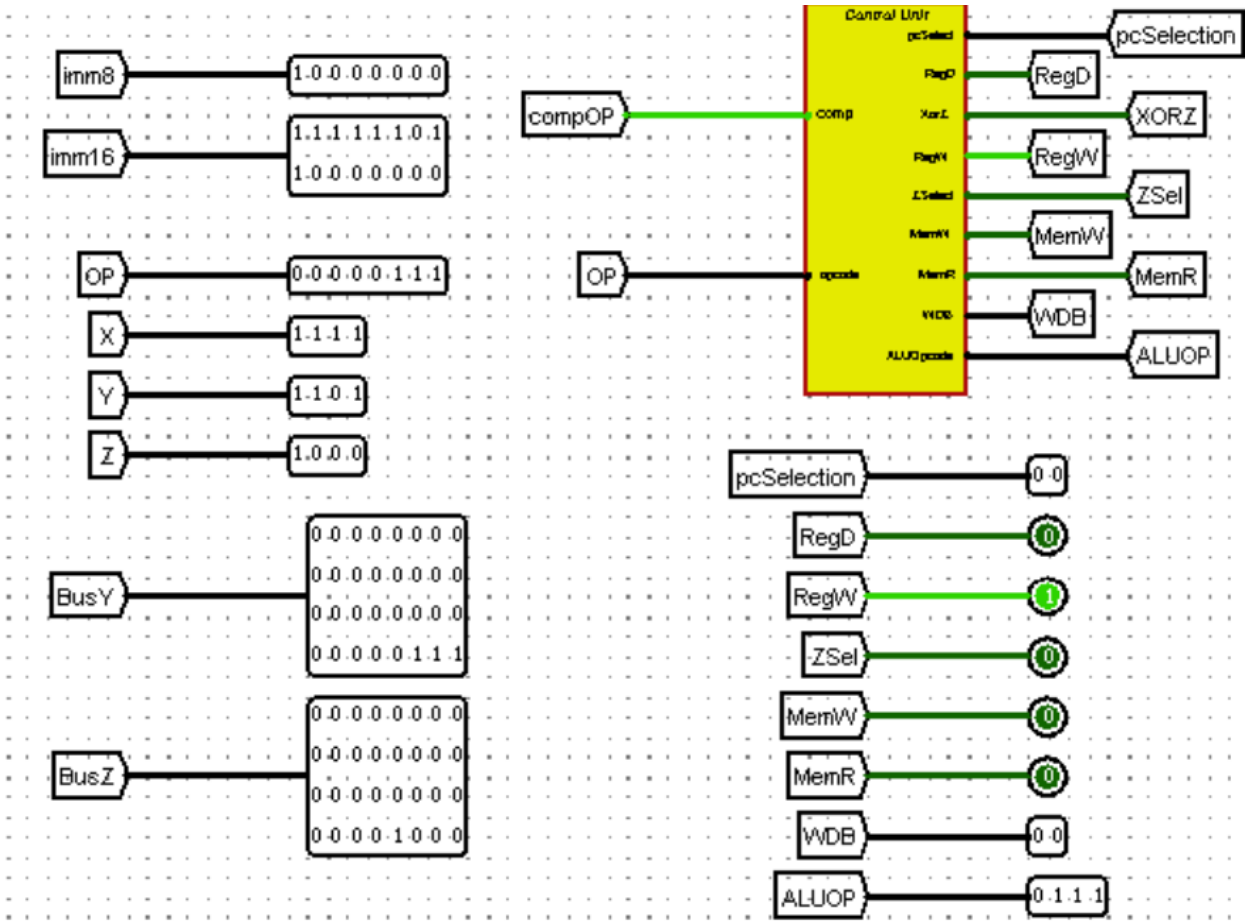


Figure 53: SLT control signal.

II.II ANDI - ROR test set

The following ROM image shows the instructions contents

```
00000 0c2107 08320d 094210 0a5401 0b6210 0d7624 0e870d 0f961f 10a501 11b402 12cb02 13db01 (
00010 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 (
00020 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 (
00030 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 (
00040 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 (
00050 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 (
```

Figure 54: instructions in ROM for second test.

- 0x0C2107 (ADDI)

  ADDI R2, R1, 7

opcode	RX	RY	Imm8
00001100	0010	0001	00000111

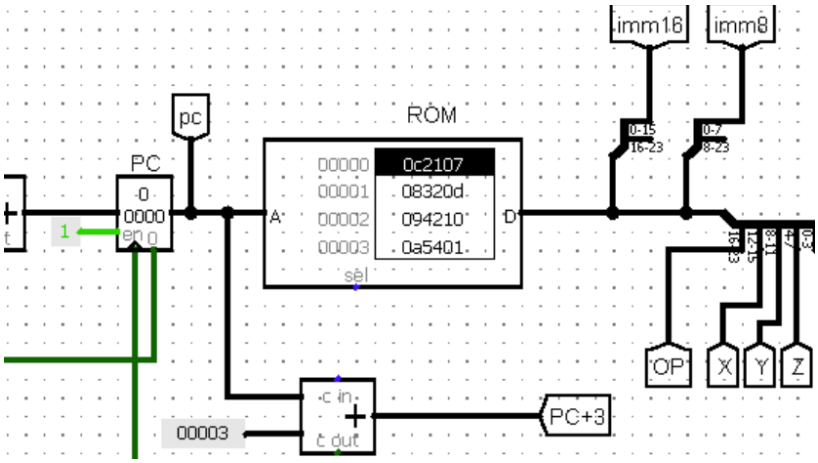
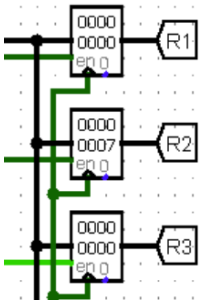


Figure 55: ADDI instruction in ROM (hex).



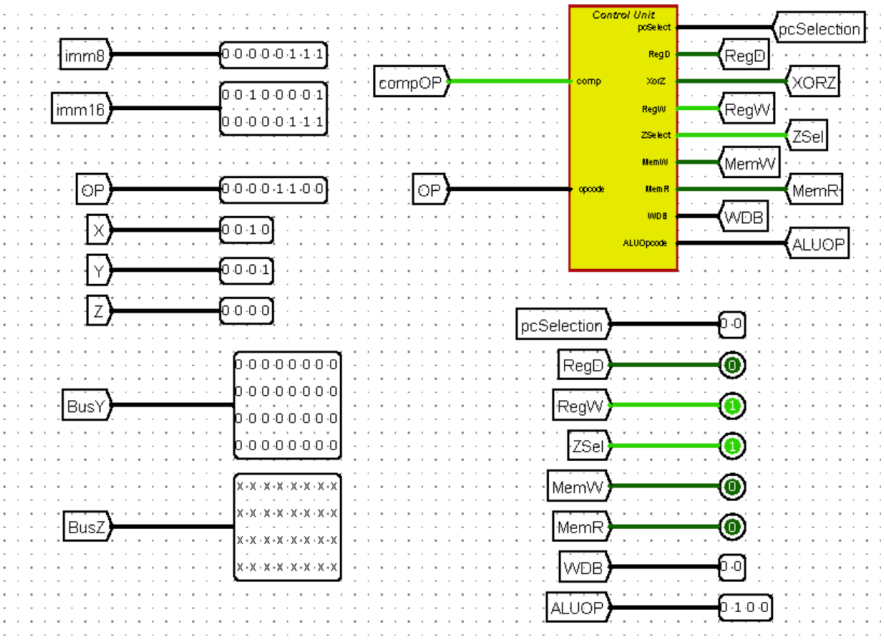


Figure 56: ADDI control signal

- 0x08320D (ANDI)

  ANDI R3, R2, 13

opcode	RX	RY	Imm8
00001000	0011	0010	00001101

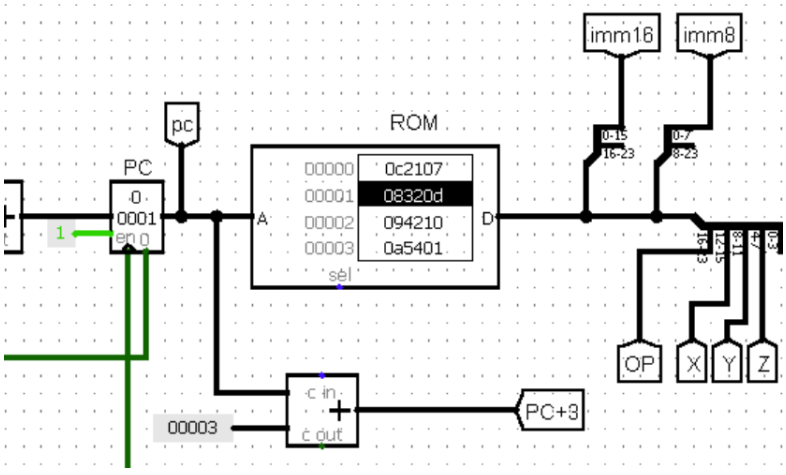


Figure 57: ANDI instruction in ROM (hex).

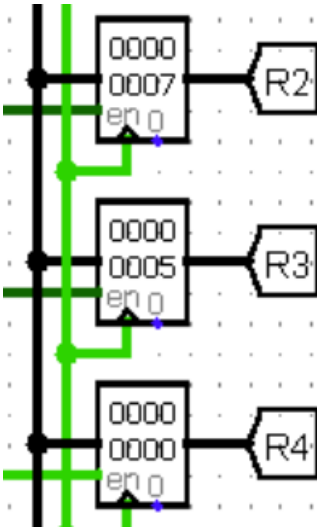


Figure 58: ANDI Reg file.

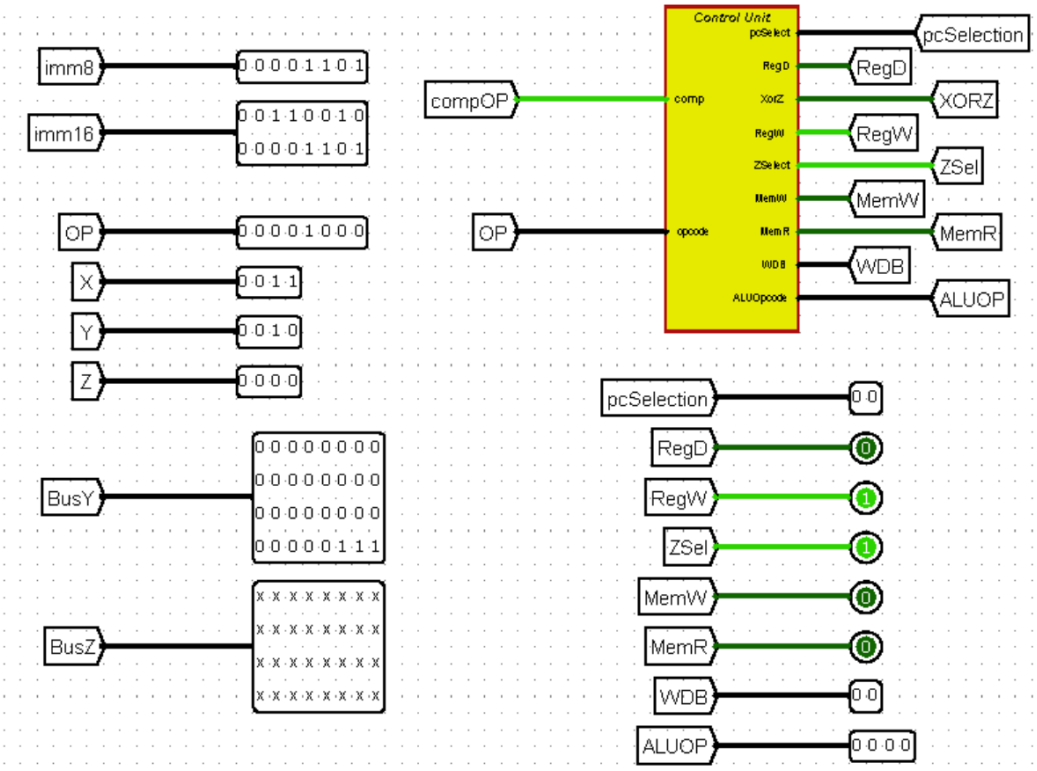


Figure 59: ANDI control signal.

- 0x094210 (CANDI)

  CANDI R4, R2, 16

opcode	RX	RY	Imm8
00001001	0100	0010	00010000

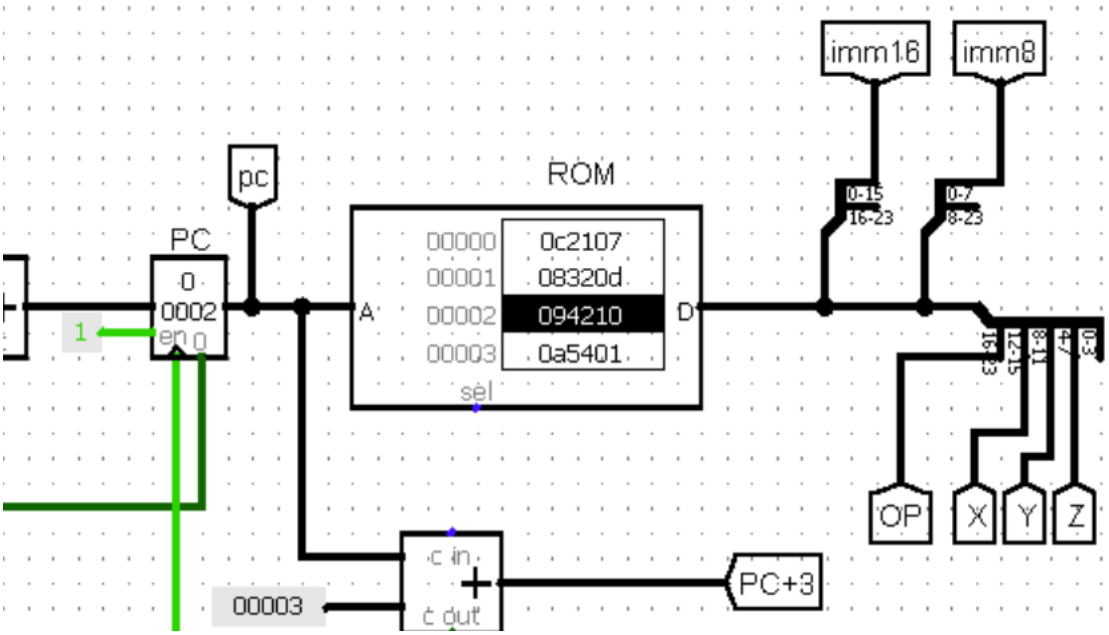


Figure 60: CANDI instruction in ROM (hex).

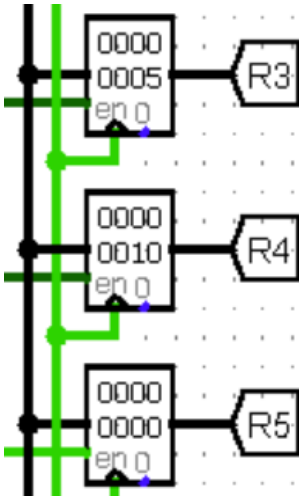


Figure 61: CANDI Reg file.

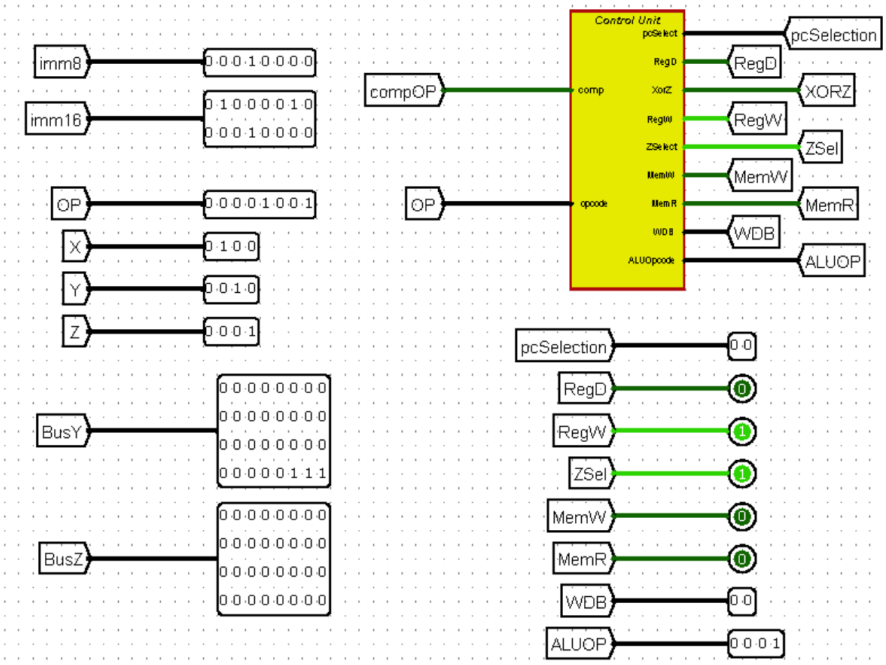


Figure 62: CANDI control signal.

- 0x0A5401 (ORI)

  ORI R5, R4, 1

opcode	RX	RY	Imm8
00001010	0101	0100	00000001

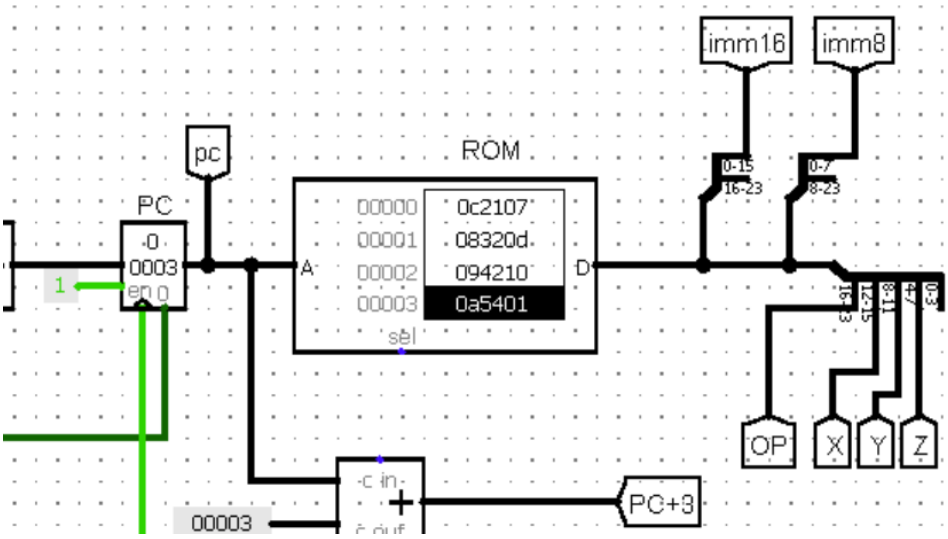


Figure 63: ORI instruction in ROM (hex).

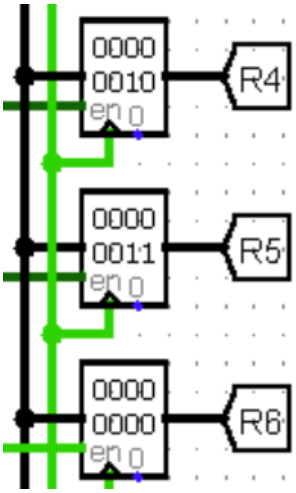


Figure 64: ORI Reg file.

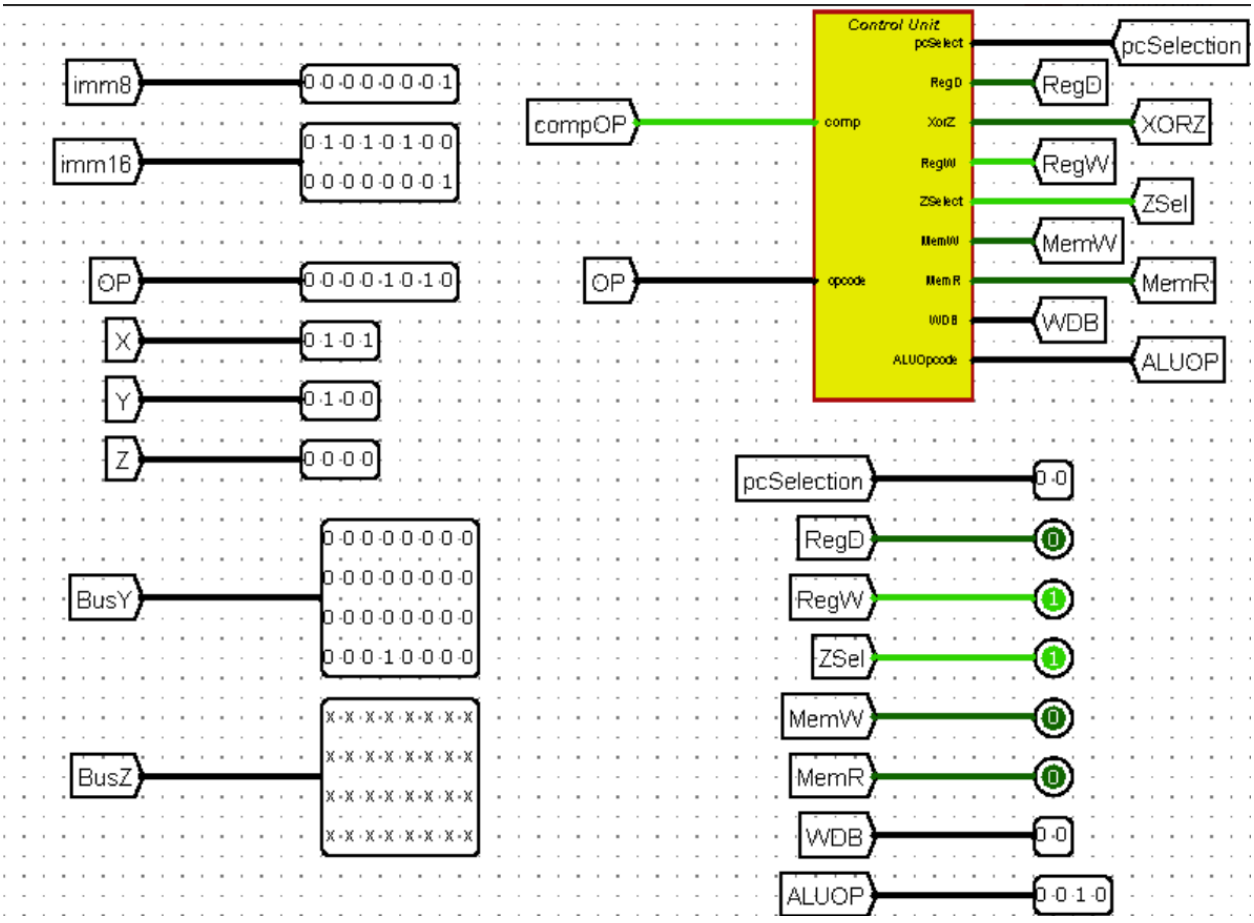


Figure 65: ORI control signal.

- 0x0B6210 (XORI)

  XORI R6, R2, 16

opcode	RX	RY	Imm8
00001011	0110	0010	00010000

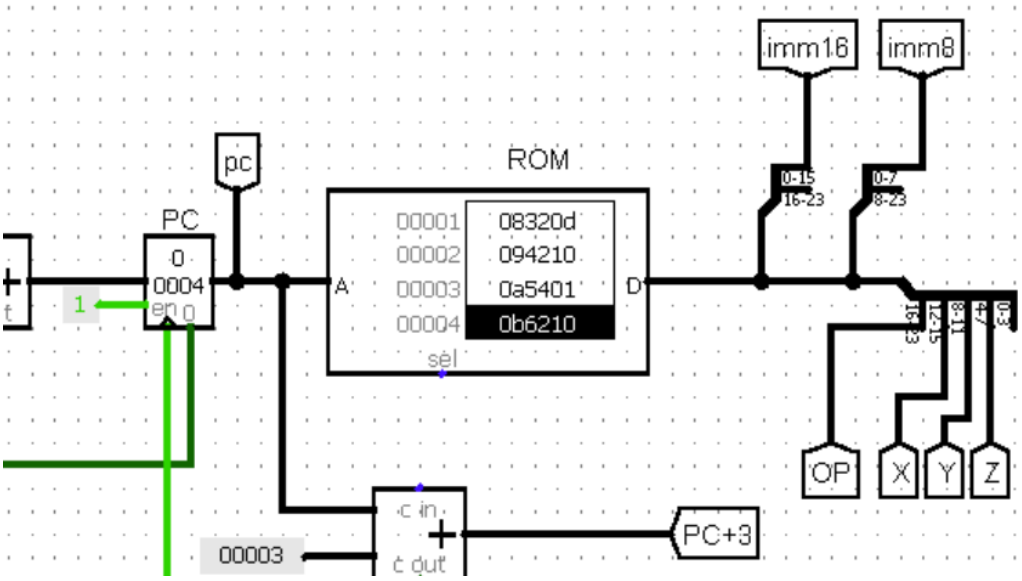


Figure 66: XORI instruction in ROM (hex).

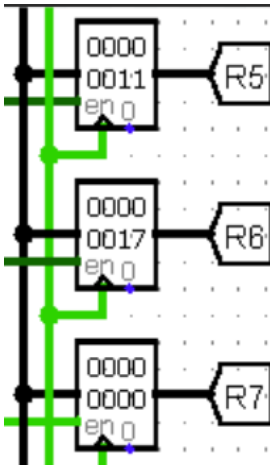


Figure 67: XORI Reg file.

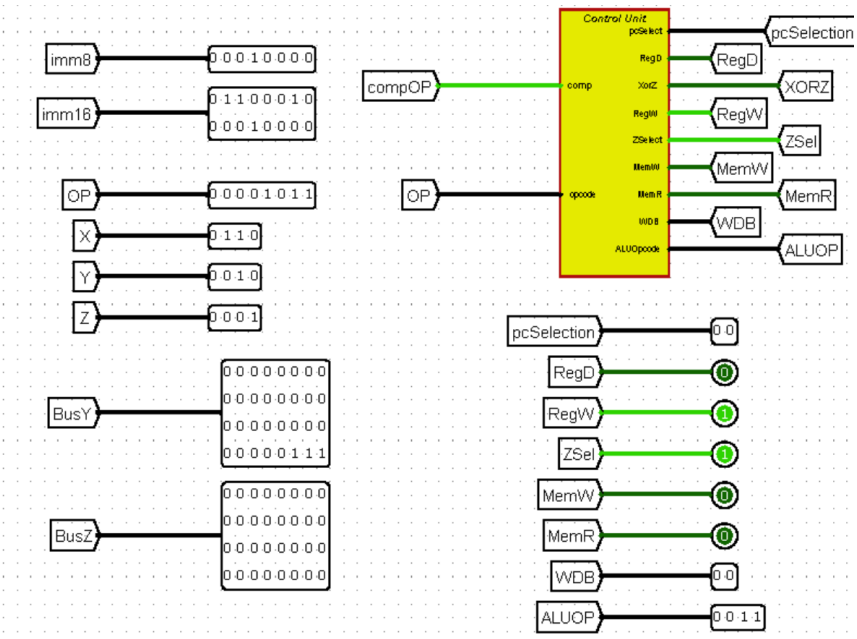


Figure 68: XORI control signal.

- 0x0D7624 (NADDI)

  NADDI R7, R6, 36

opcode	RX	RY	Imm8
00001101	0111	0110	00100100

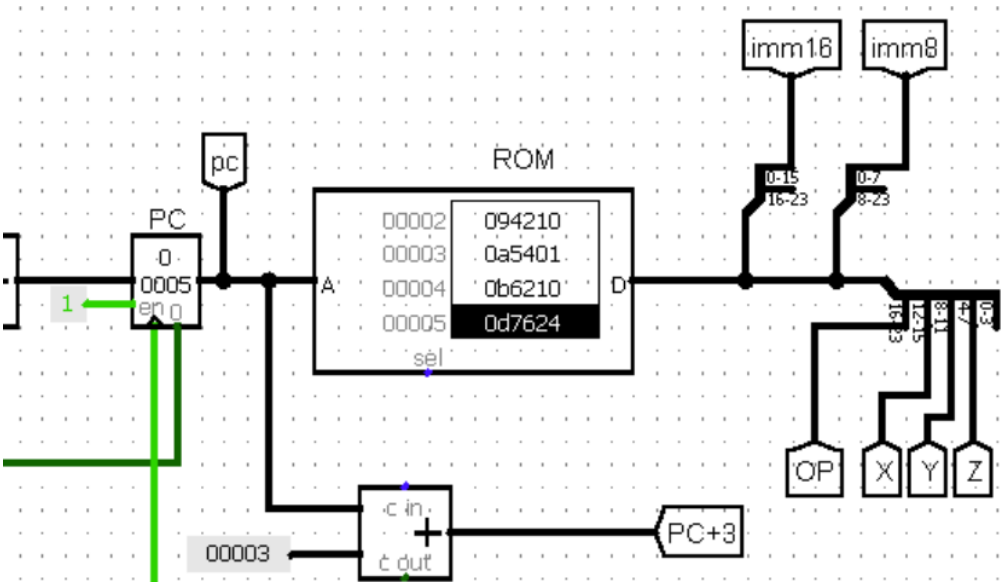


Figure 69: NADDI instruction in ROM (hex).

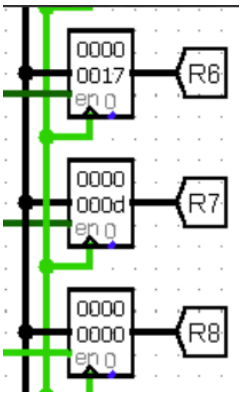


Figure 70: NADDI Reg file.

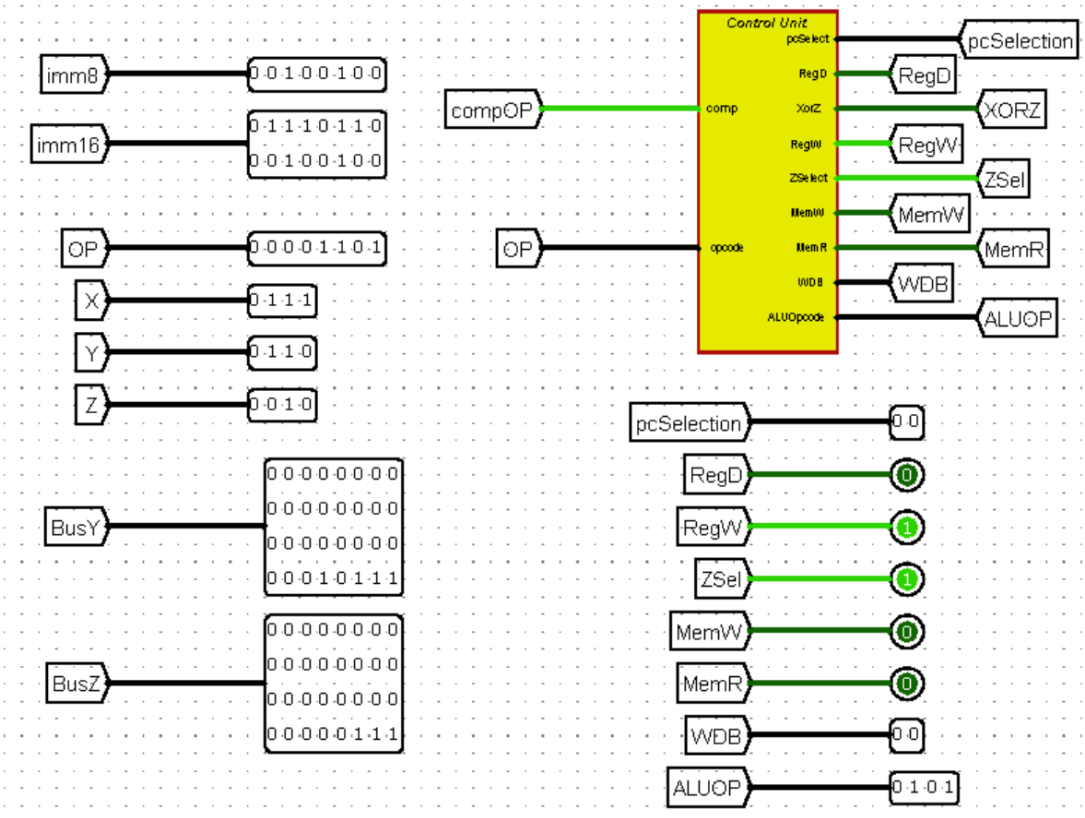


Figure 71: NADDI control signal.

- 0x0E870D (SEQI)

  SEQI R8, R7, 13

opcode	RX	RY	Imm8
00001110	1000	0111	00001101

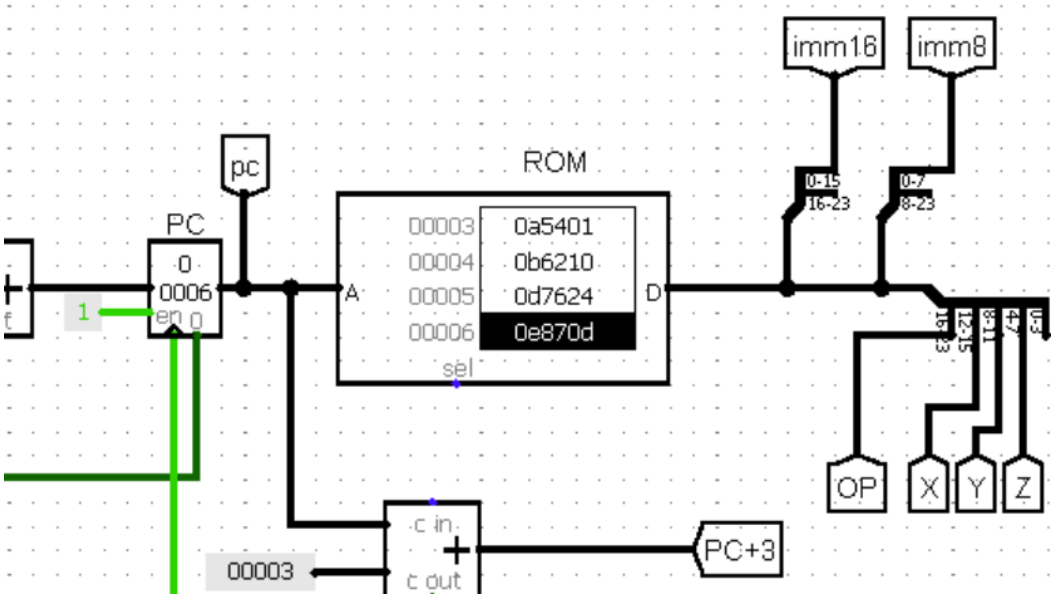


Figure 72: SEQI instruction in ROM (hex).

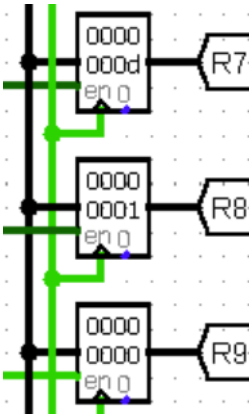


Figure 73: SEQI Reg file.

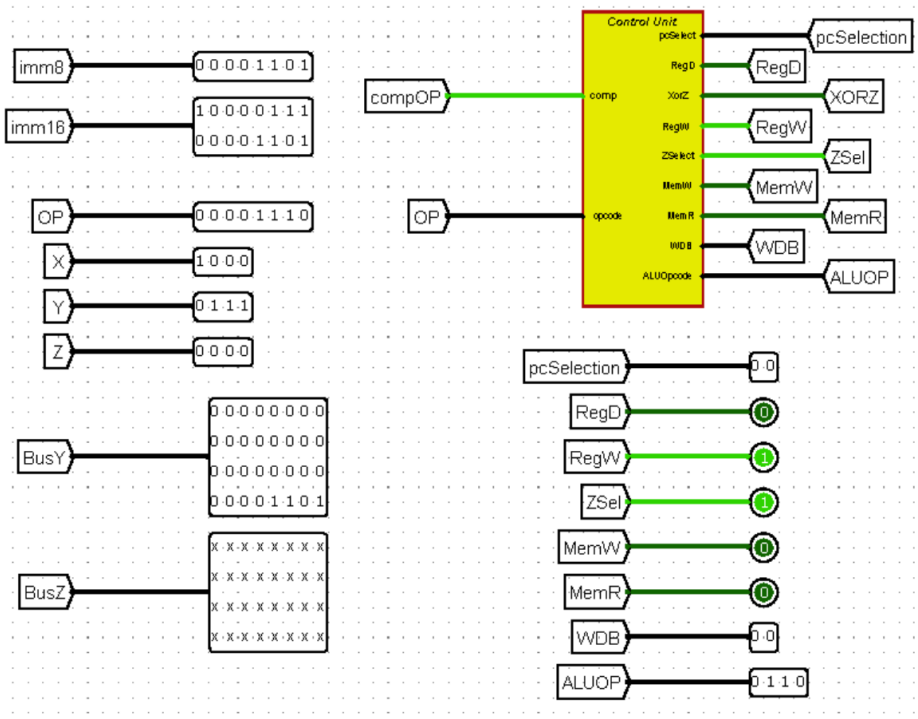


Figure 74: SEQI control signal.

- 0x0F961F (SLTI)

  SLTI R9, R6, 31

opcode	RX	RY	Imm8
00001111	1001	0110	00011111

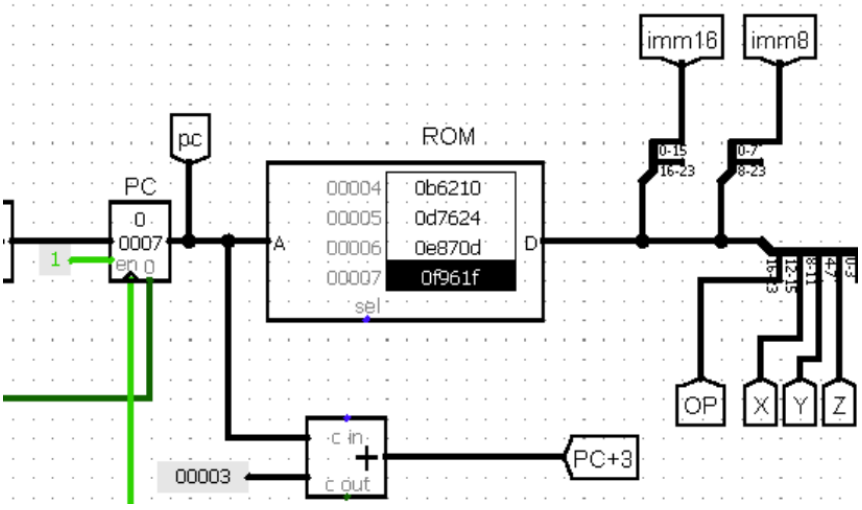


Figure 75: SLTI instruction in ROM (hex).

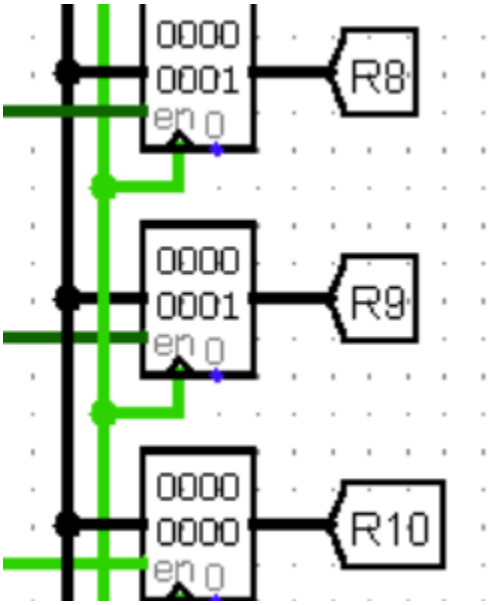


Figure 76: SLTI Reg file.

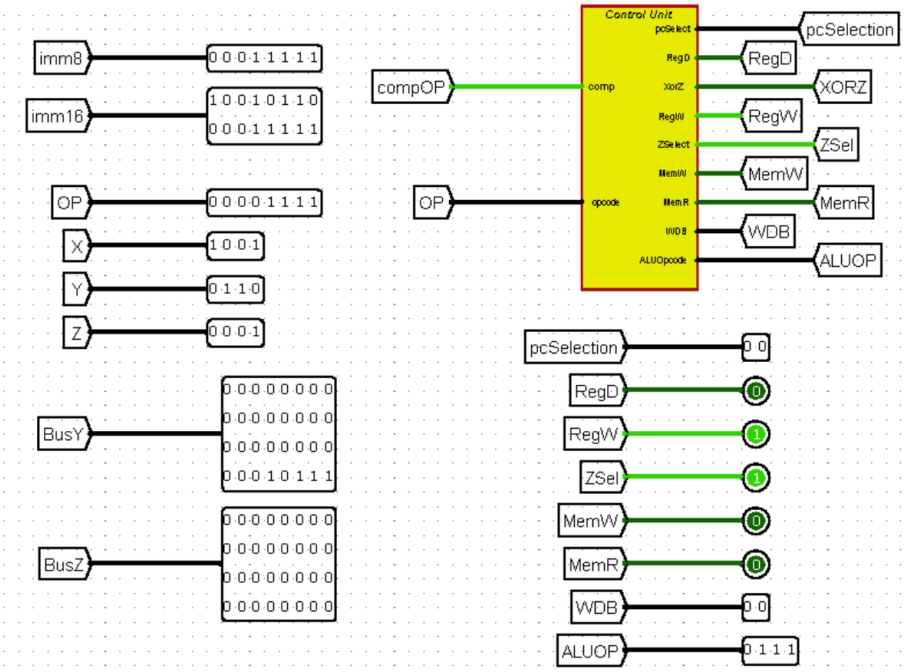


Figure 77: SLTI control signal.

- 0x10A501 (SLL)

  SLL R10, R5, 1

opcode	RX	RY	Imm8
00010000	1010	0101	00000001

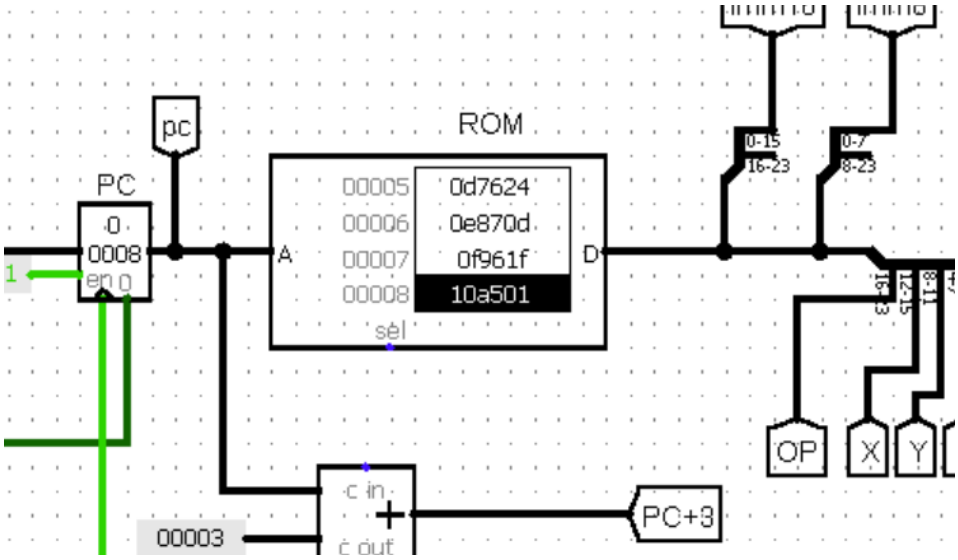


Figure 82: SLL instruction in ROM (hex).

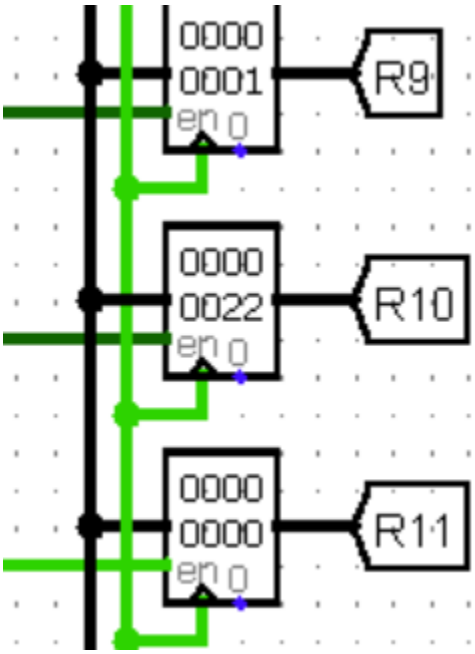


Figure 83: SLL Reg file.

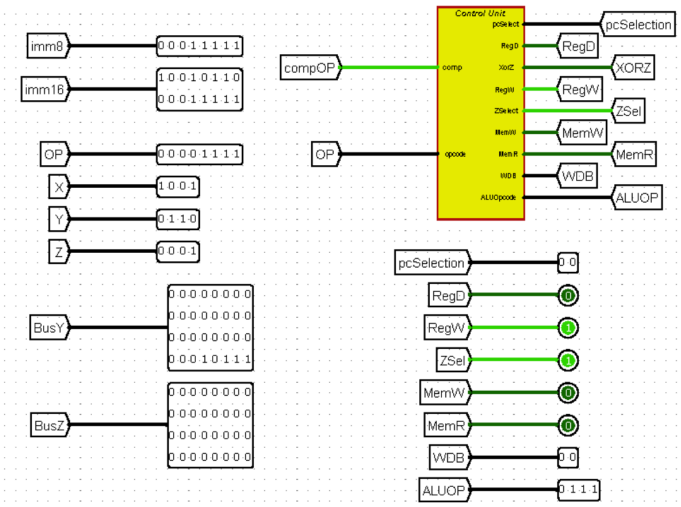


Figure 78: SLTI control signal.

- 0x11B402 (SRL)

  SRL R11, R4, 2

opcode	RX	RY	Imm8
00010001	1011	0100	00000010

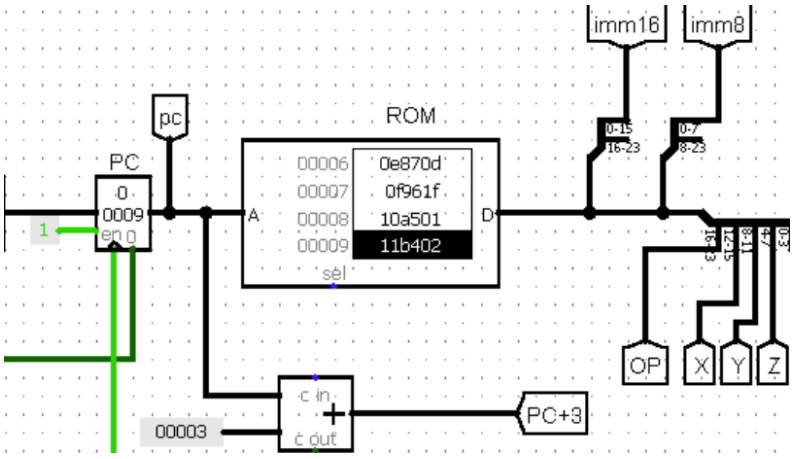


Figure 79: SRL instruction in ROM (hex).

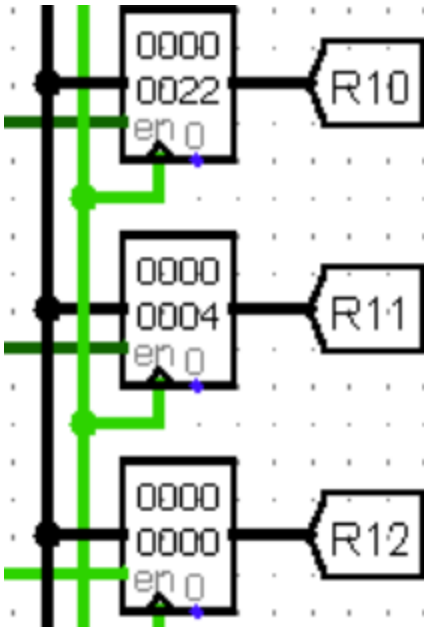


Figure 80: SRL Reg file.

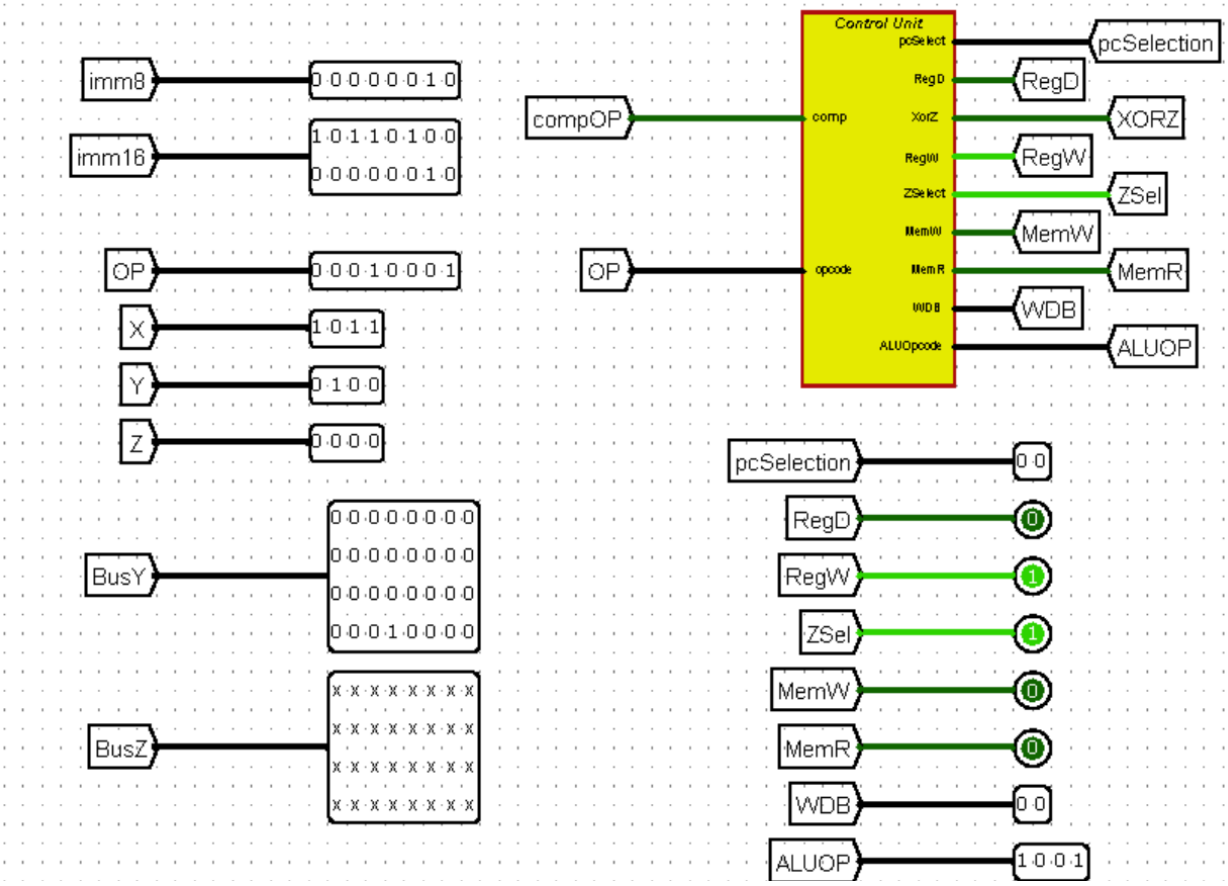


Figure 87: SRL control signal.

- 0x12CB02 (SRA)

  SRA R12, R11, 2

opcode	RX	RY	Imm8
00010010	1100	1011	00000010

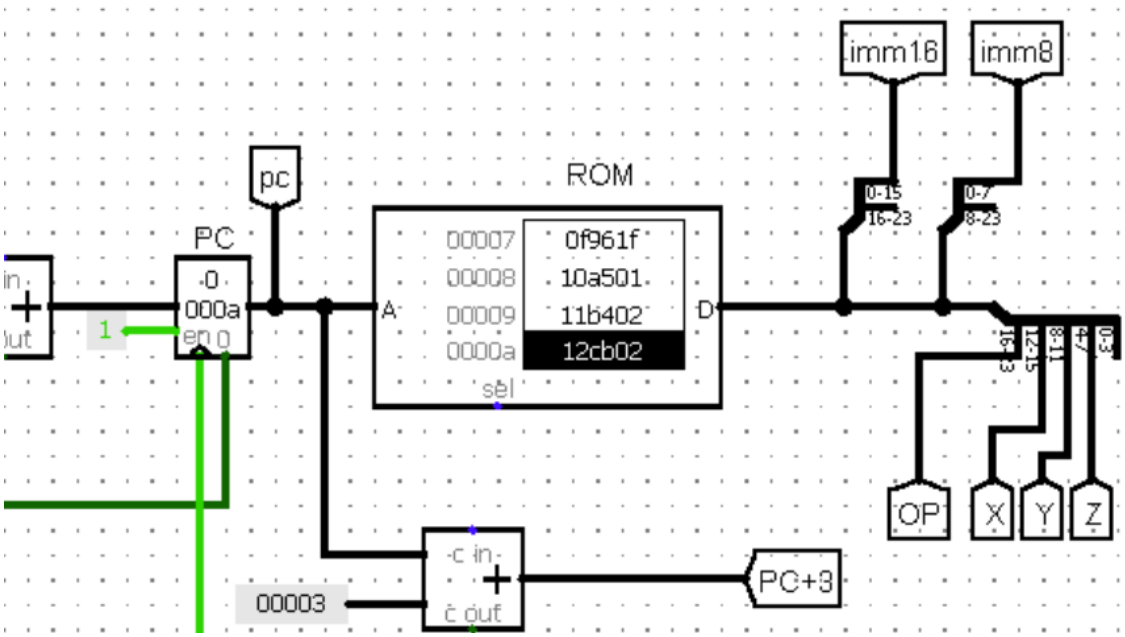


Figure 81: SRA instruction in ROM (hex).

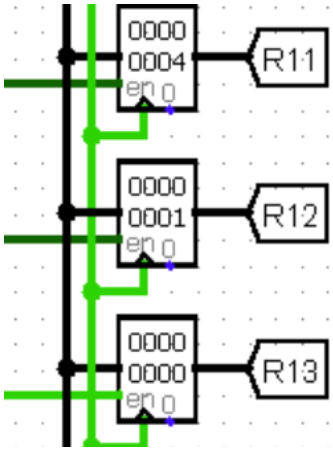


Figure 82: SRA Reg file.

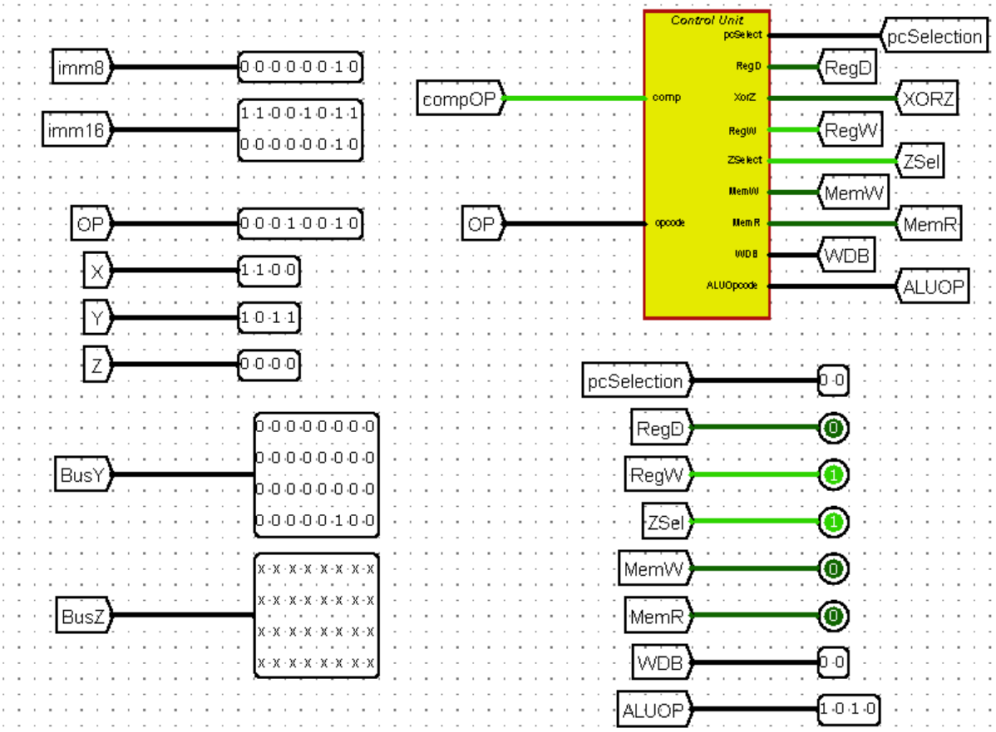


Figure 83: SRA control signal.

- 0x13DB01 (ROR)

  ROR R13, R11, 1

opcode	RX	RY	Imm8
00010011	1101	1011	00000001

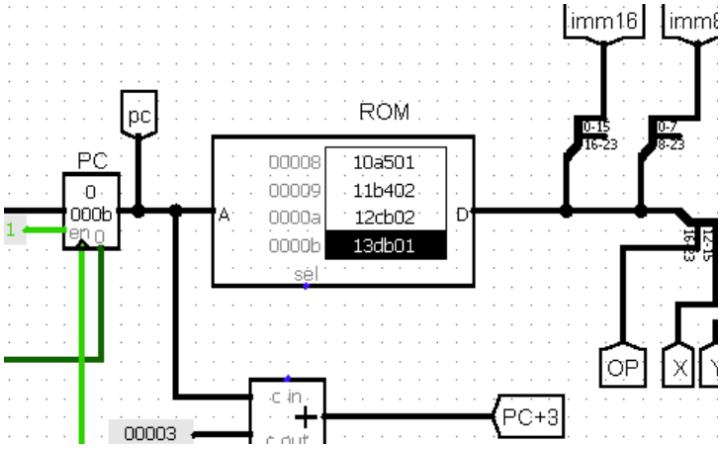


Figure 84: ROR instruction in ROM (hex).

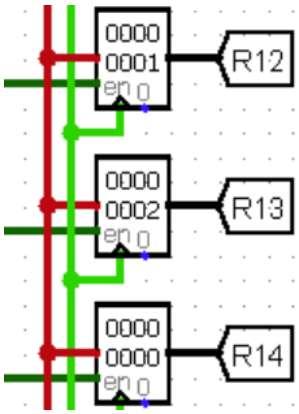


Figure 85: ROR Reg file.

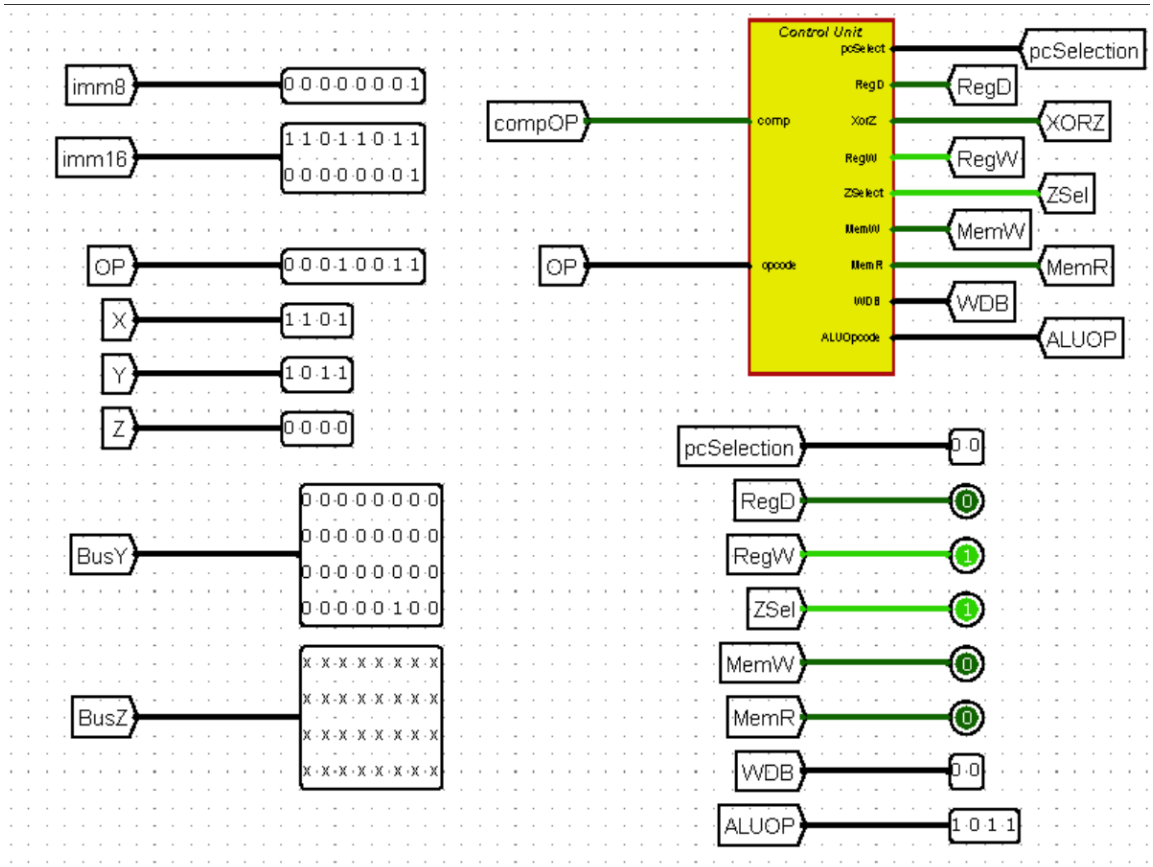


Figure 86: ROR control signal.

II.III BEQ - BGE test set

The following ROM image shows the instructions contents

```
00000 0c2103 0c3103 0c5106 0c6104 142303 000000 000000 000000 000000 000000 143502 153504 000000 000000 000000 000000
00010 000000 000000 000000 153205 166501 000000 165303 000000 000000 000000 000000 000000 000000 000000 000000 000000
00020 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00030 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00040 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00050 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
```

Figure 94: Instruction in ROM for third test set.

The first four instructions are ADDI instructions, which we have already tested earlier. We are now starting from instruction 5, with the following initial register values: R2 = 3, R3 = 3, R5 = 6, and R6 = 4.

- 0x142303 (BEQ) // Branch condition true

  BEQ R12, R3, 3

opcode	RX	RY	Imm8
00010100	0010	0011	00000011

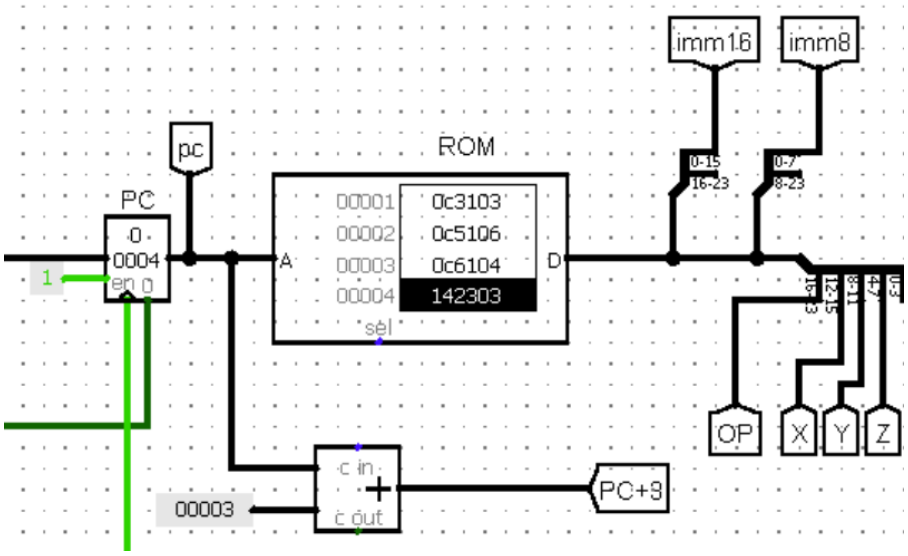


Figure 87: BEQI instruction in ROM (hex).

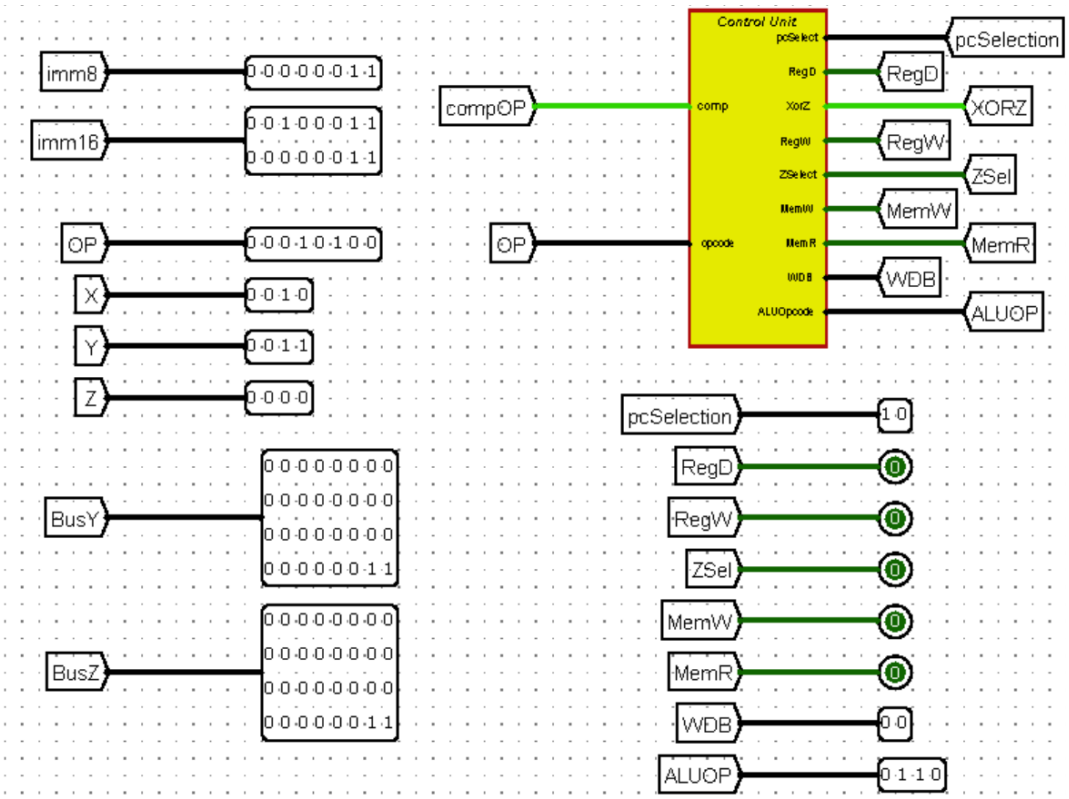


Figure 88: BEQI control signal.

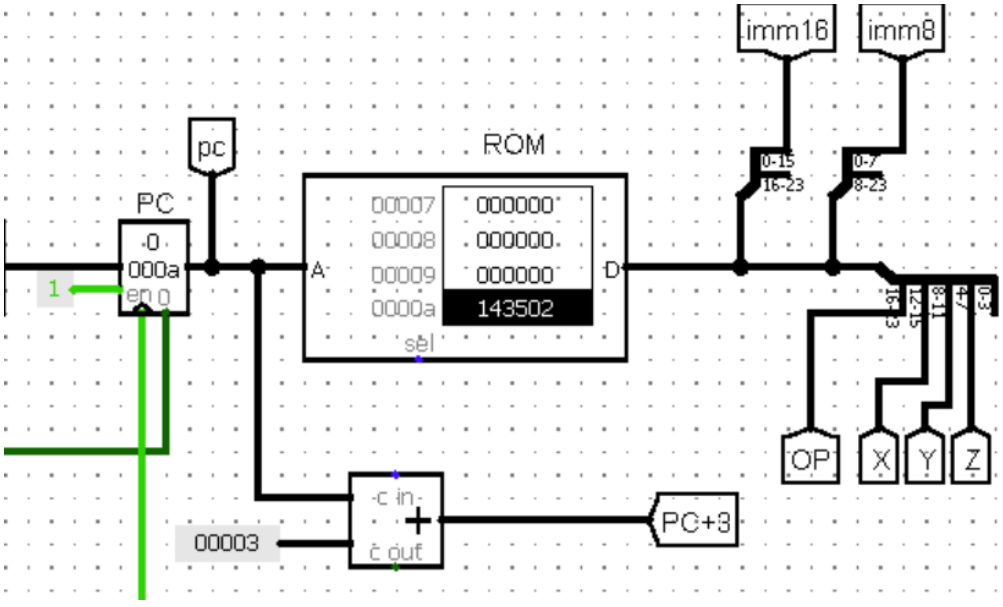


Figure 89: Pc position after executing the instruction -now at BEQ-.

- 0x143502 (BEQ) // Branch condition false

  BEQ R3, R5, 2

opcode	RX	RY	Imm8
00010100	0011	0101	00000010

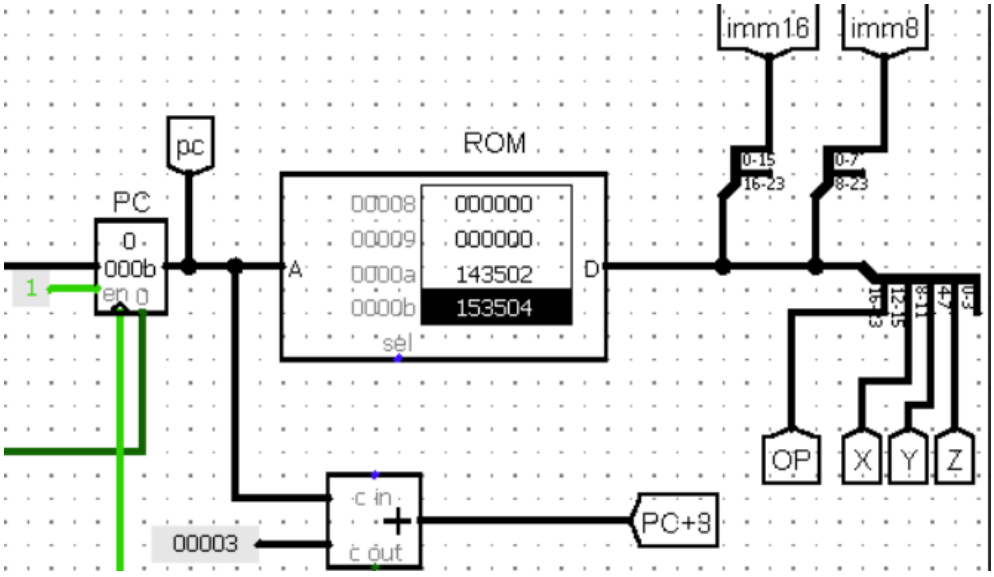


Figure 90: Pc position after executing the instruction -now at BNE-.

- 0x153504 (BNE) // Branch condition true

  BNE R3, R5, 4

opcode	RX	RY	Imm8
00010101	0011	0101	00000100

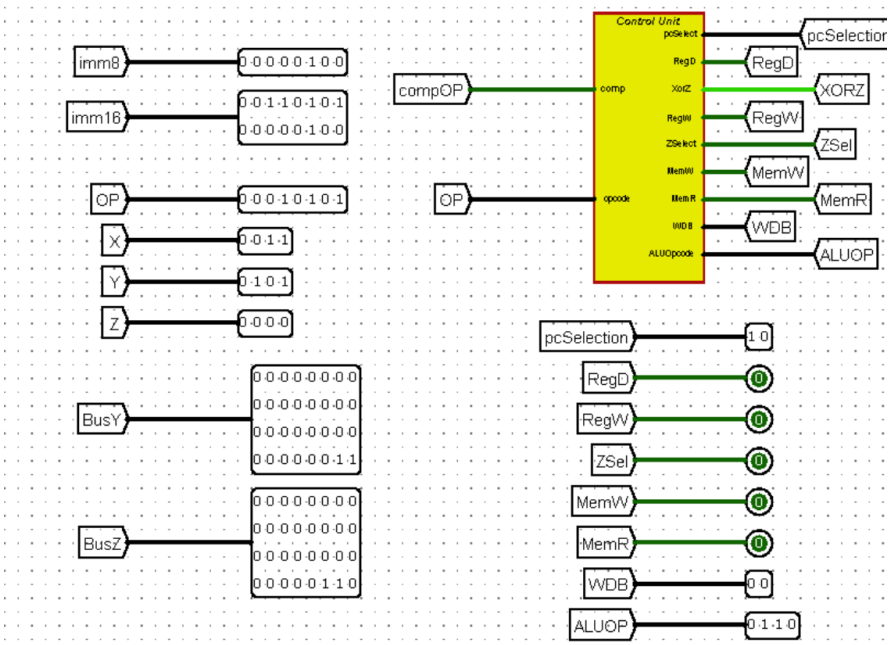


Figure 91: BNE control signal.

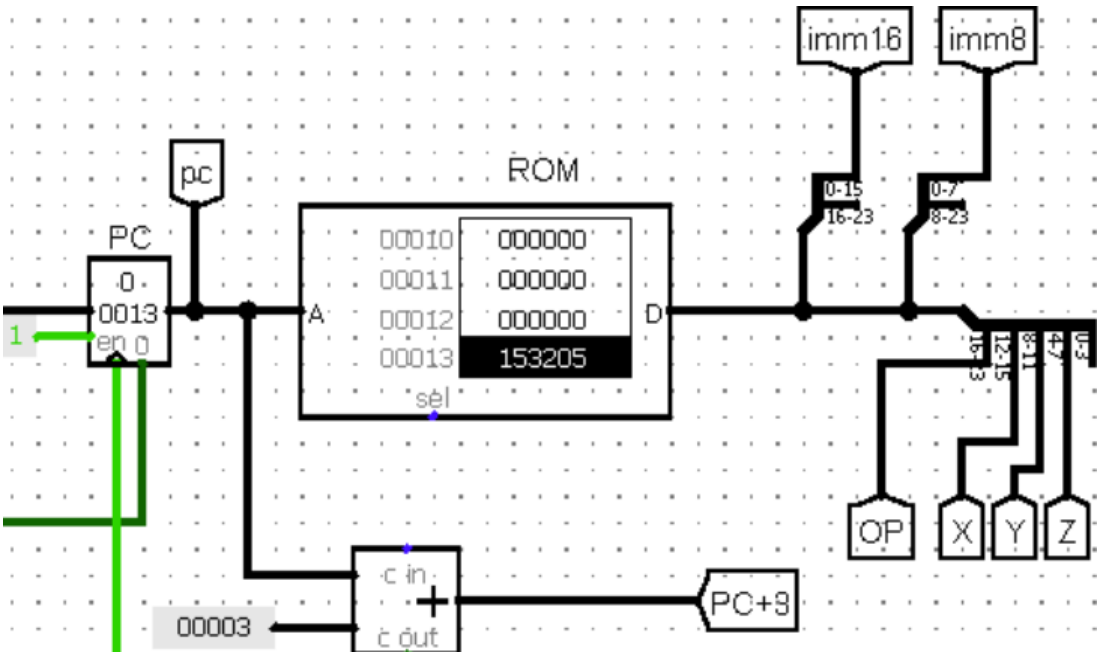


Figure 92: Pc after executing the instruction -now at BNE-.

- 0x153205 (BNE) // Branch condition false

  BNE R3, R5, 4

opcode	RX	RY	Imm8
00010101	0011	0010	00000101

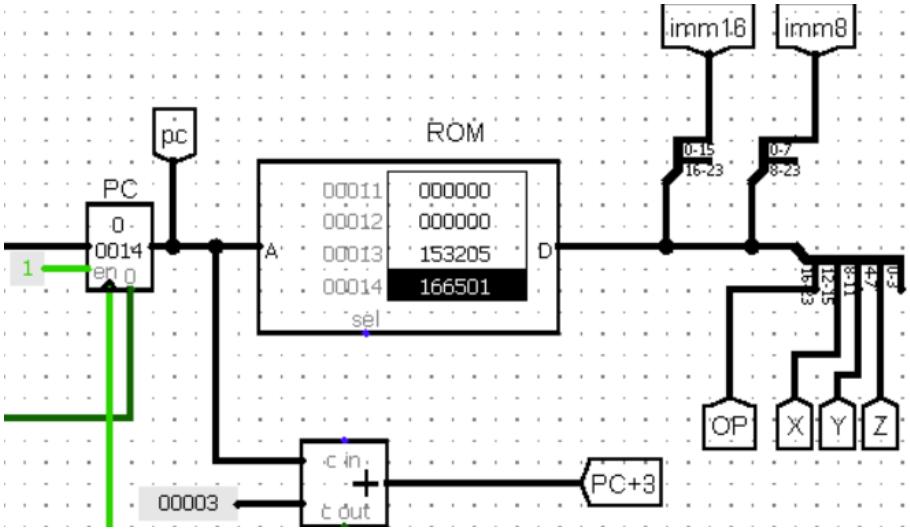


Figure 93: Pc after executing the instruction -now at BLT-.

- 0x166501 (BLT) // Branch condition true

  BLT R6, R5, 1

opcode	RX	RY	Imm8
00010110	0110	0101	00000001

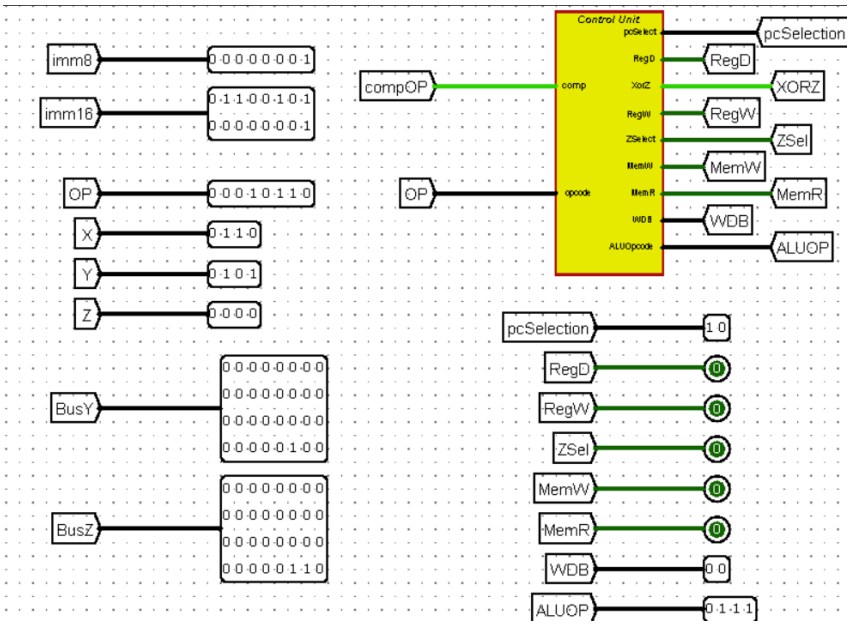


Figure 94: BLT control signal.

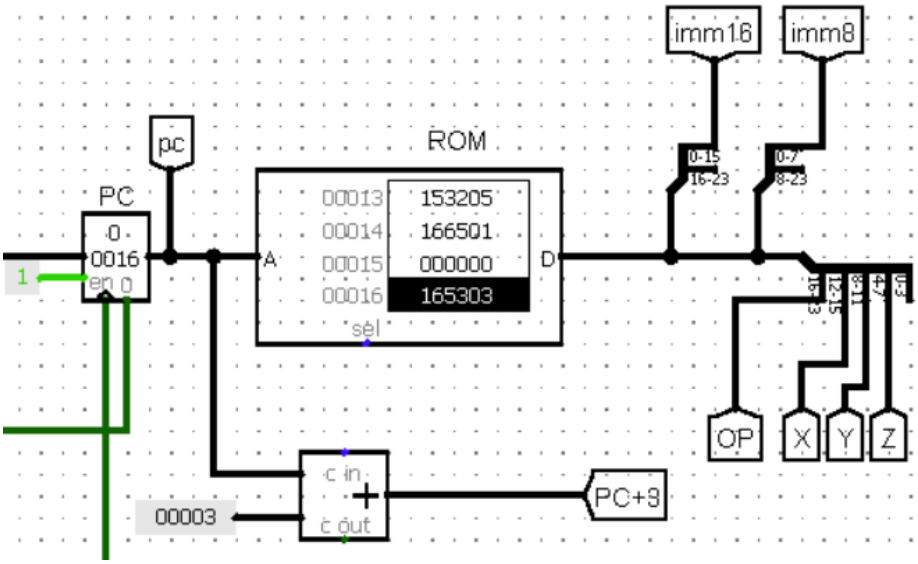


Figure 95: Pc after executing the instruction -now at BLT-.

- 0x166501 (BLT) // Branch condition false

  BLT R5, R3, 3

opcode	RX	RY	Imm8
00010110	0101	0011	00000011

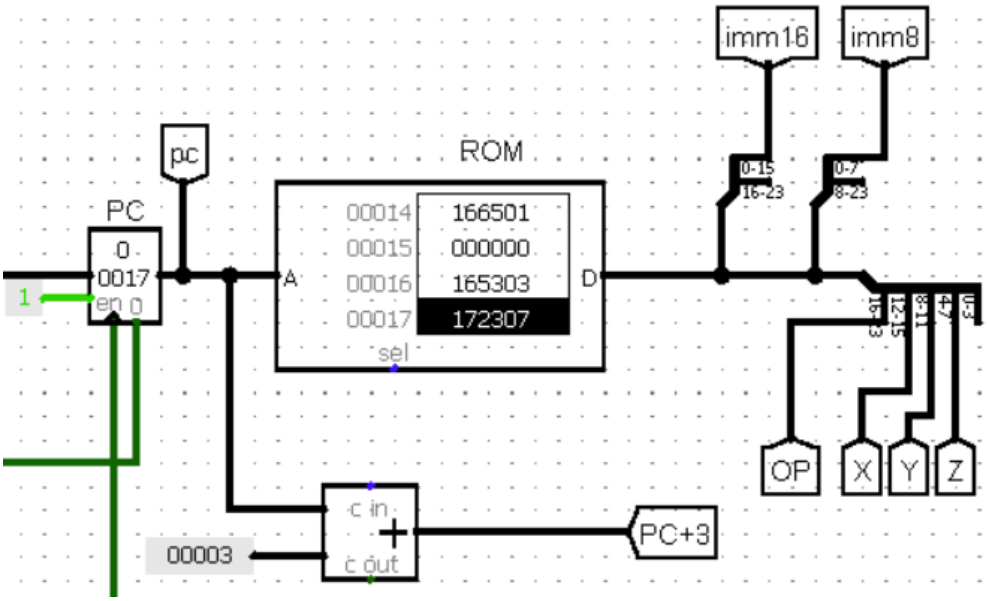


Figure 96: Pc after executing the instruction -now at BGE-.

- 0x172307 (BGE) // Branch condition true

  BGE R2, R3, 7

opcode	RX	RY	Imm8
00010111	0010	0011	00000111

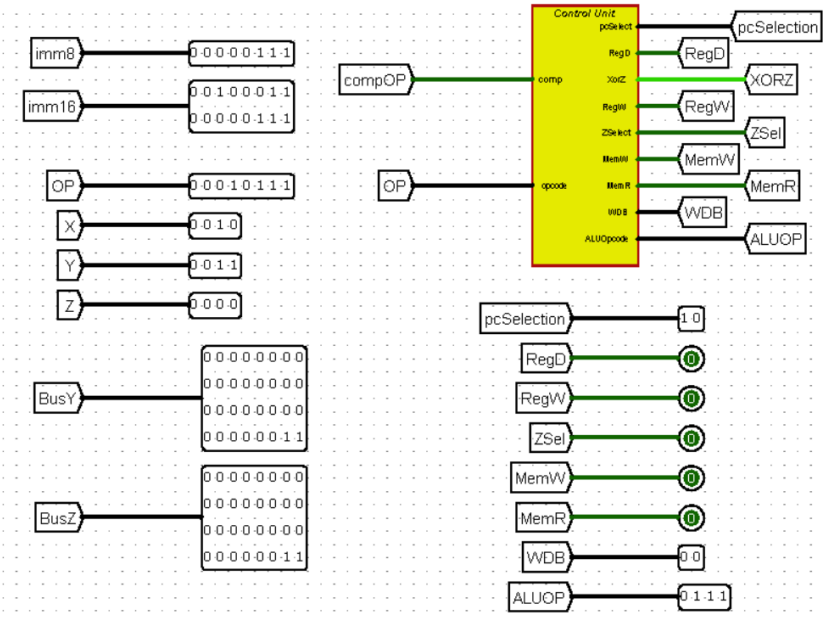


Figure 97: BGE control signal.

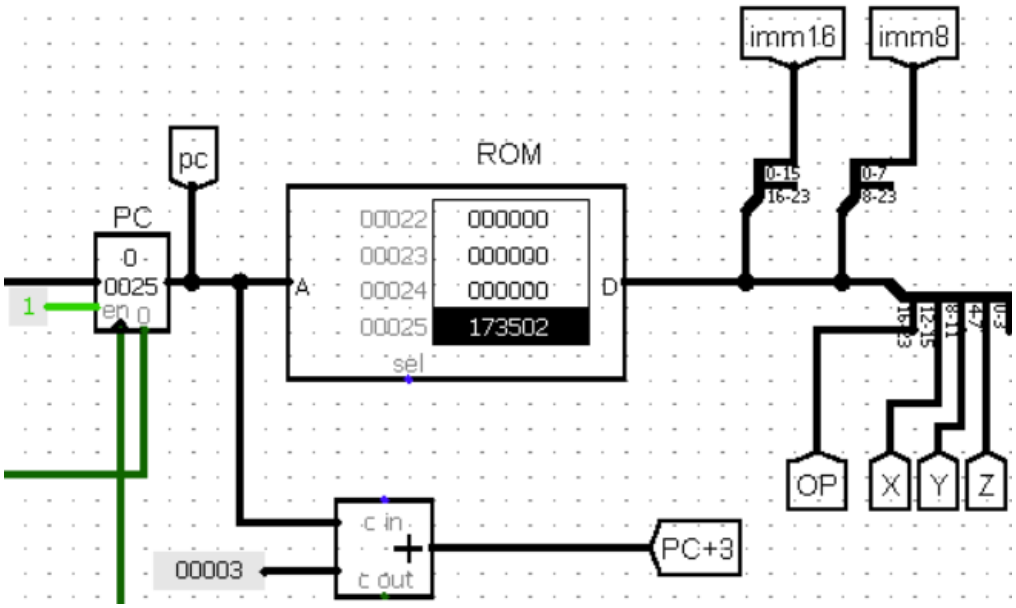


Figure 98: Pc after executing the instruction -now at BGE-.

- 0x173502 (BGE) // Branch condition false

  BGE R3, R5, 2

opcode	RX	RY	Imm8
00010111	0011	0101	00000010

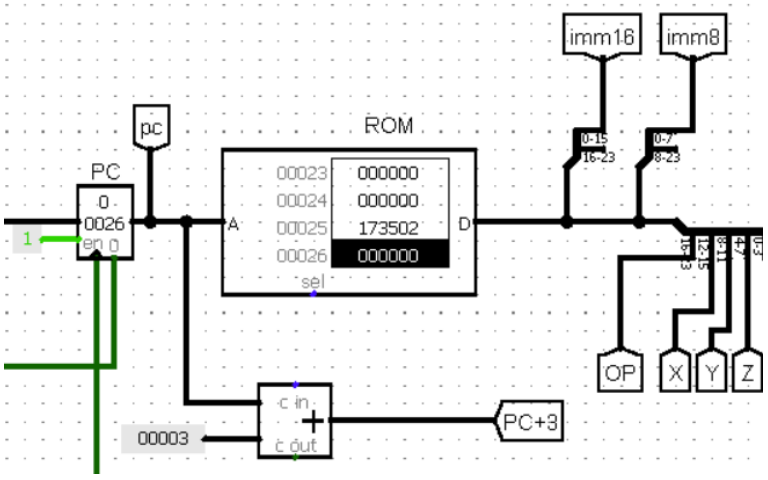


Figure 99: Pc after executing the instruction.

II.IIII Loading instructions test set

The following ROM image shows the instructions contents

```
00000  182106 192107 000000 000
00010  000000 000000 000000 000
00020  000000 000000 000000 000
00030  000000 000000 000000 000
```

Figure 100: Instructions in ROM (hex).

```
00000  00000000 00000000 00000000 00000000  00000000 00000000 000a56a3 00000000  0000000
00010  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  0000000
00020  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  0000000
00030  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  0000000
00040  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  0000000
00050  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  0000000
```

Figure 101: : RAM before executing instructions.

- 0x182106 (LW)

  LW R2, R1, 6

opcode	RX	RY	Imm8
00011000	0010	0001	00000110

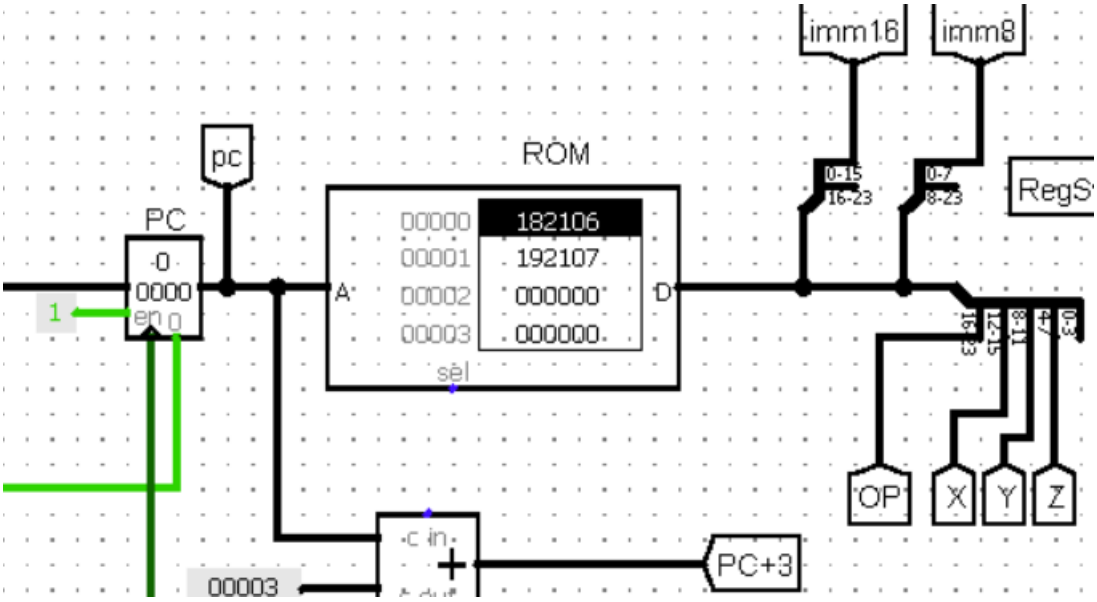


Figure 110: LW instruction in ROM (hex).

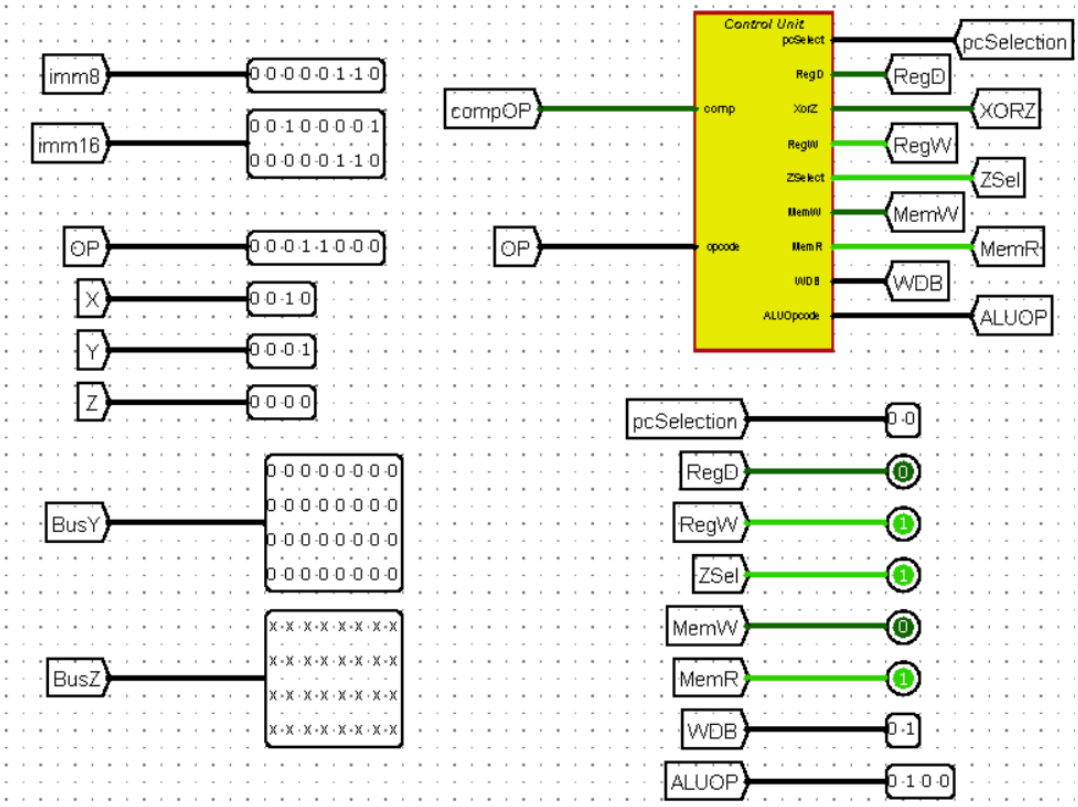


Figure 102: LW control signal.

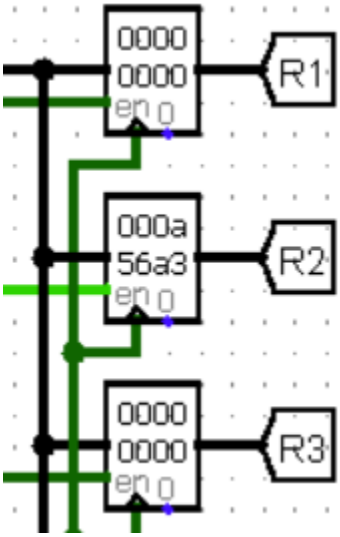


Figure 103:: LW Register file.

- 0x192107 (SW)

  SW R2, R1,7

opcode	RX	RY	Imm8
00011001	0010	0001	00000111

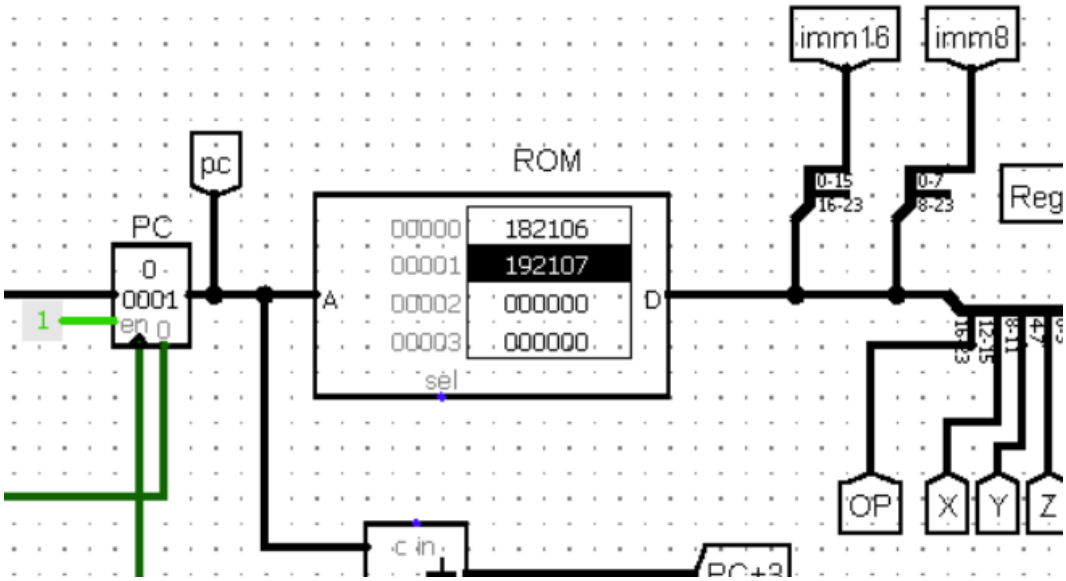


Figure 104: SW instruction in ROM (hex).

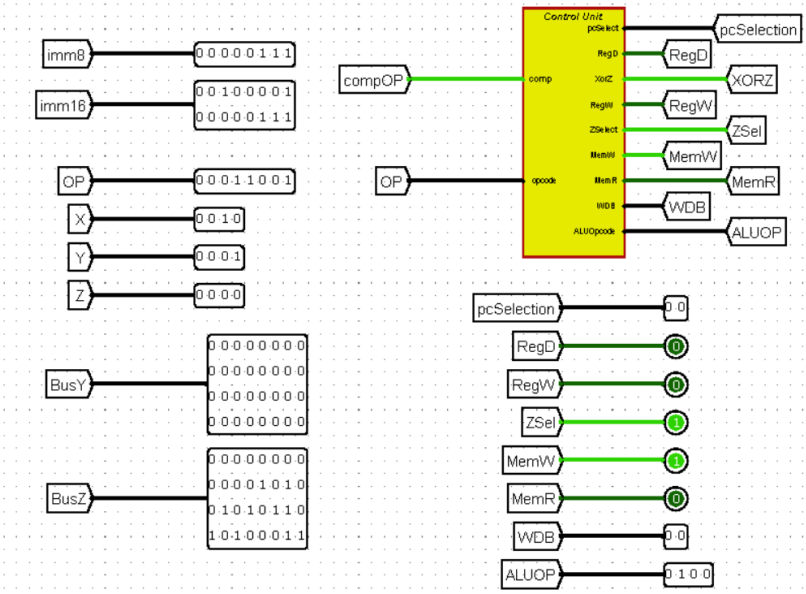


Figure 105: SW control signal.

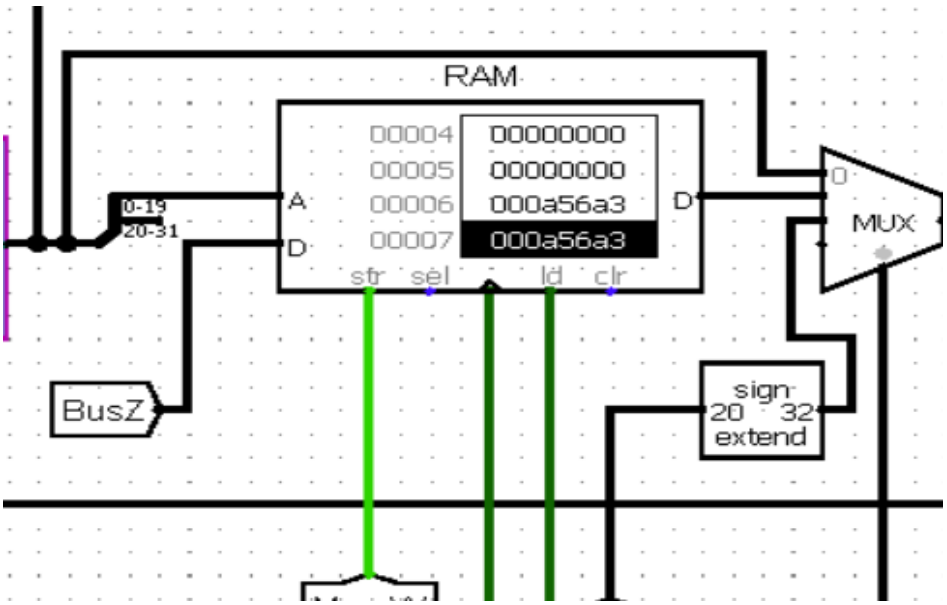


Figure 115: RAM after executing the SW instruction.

II.IV J type test set

The following ROM image shows the instructions contents

```
00000 1a001e 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00010 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00020 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00030 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 1a0005 000000 000000
00040 000000 000000 000000 000000 000000 000000 1b000f 000000 000000 000000 000000 000000 000000 000000 000000
00050 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00060 000000 000000 000000 000000 1b0001 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
00070 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
```

Figure 116: instructions in ROM (hex).

- 0x1A001E (J)

  J 30

opcode	Imm16
00011010	0000000000011110

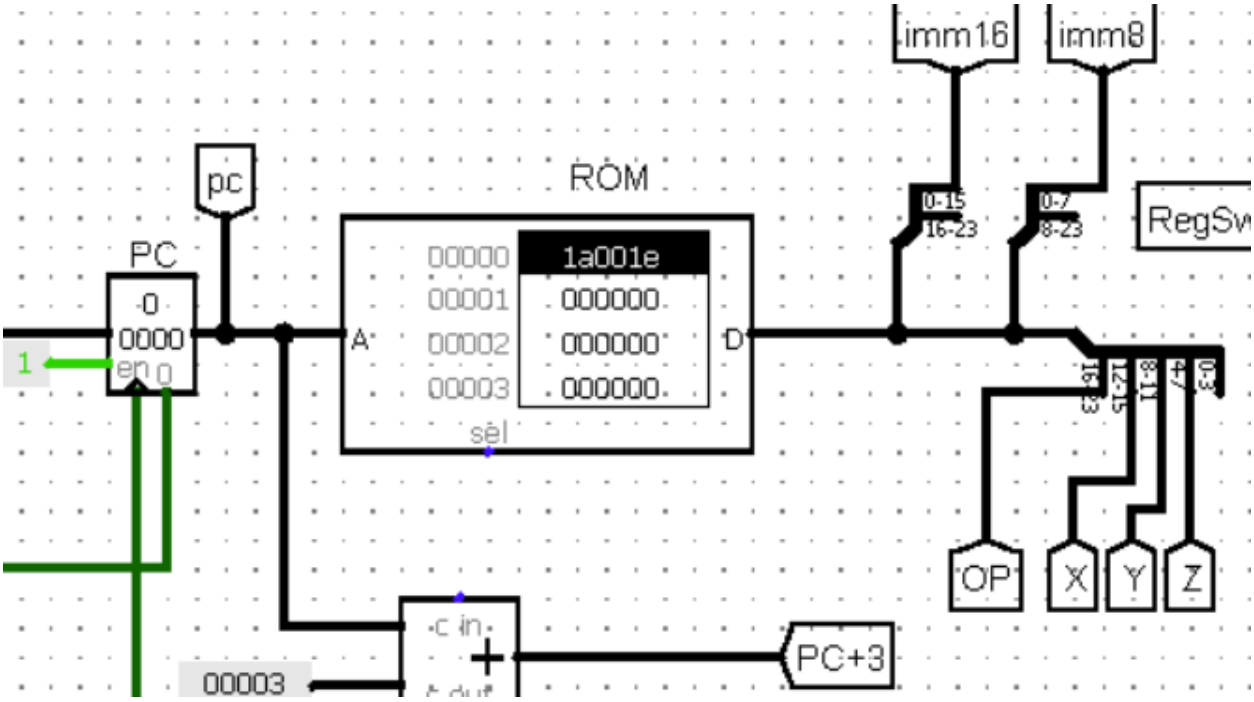


Figure 117: J instruction in ROM (hex).

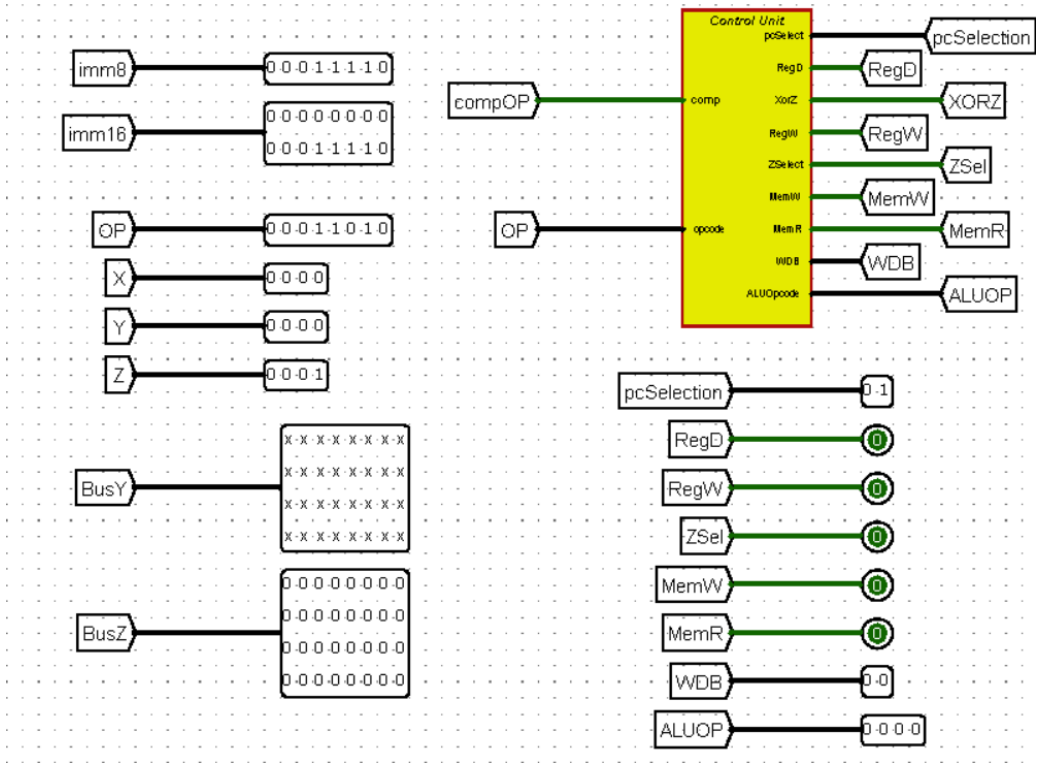


Figure 106: J control signal.

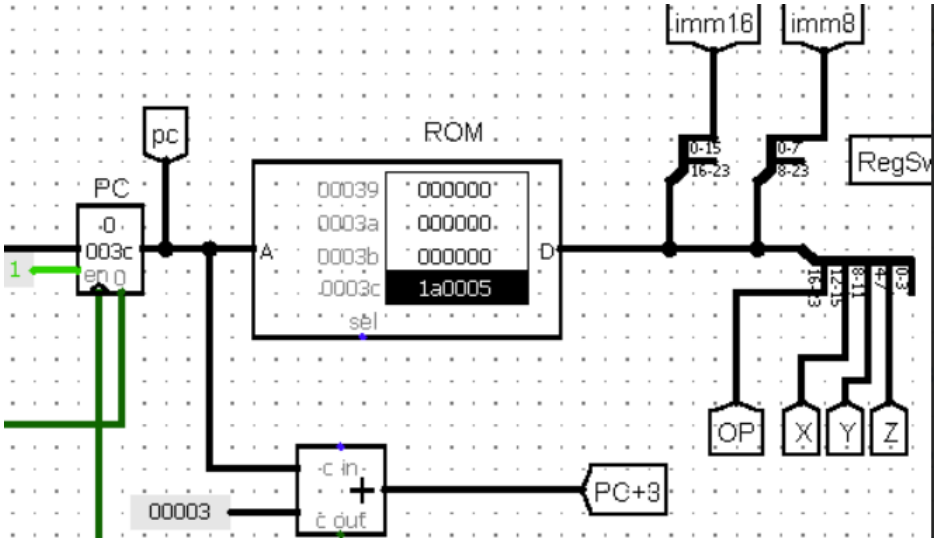


Figure 107:Pc after executing the instruction -now at J-.

- 0x1A0005 (J)

J 5

opcode	Imm16
00011010	0000000000000101

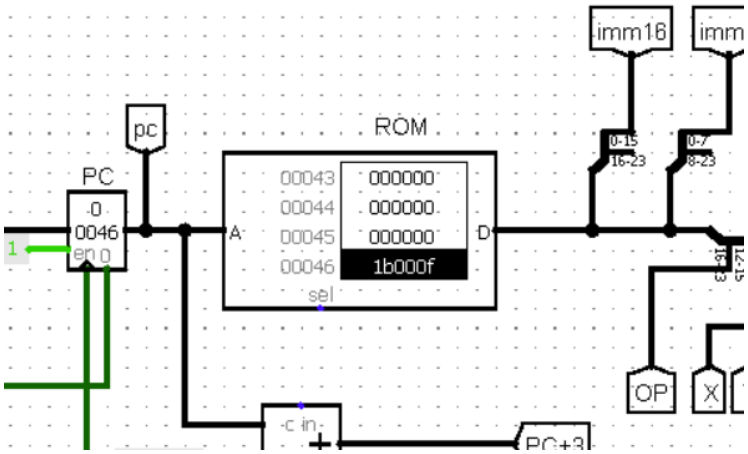


Figure 108: Pc after executing the instruction -now at JAL-.

- 0x1B000F (JAL)

JAL 7

opcode	Imm16
00011011	0000000000000111

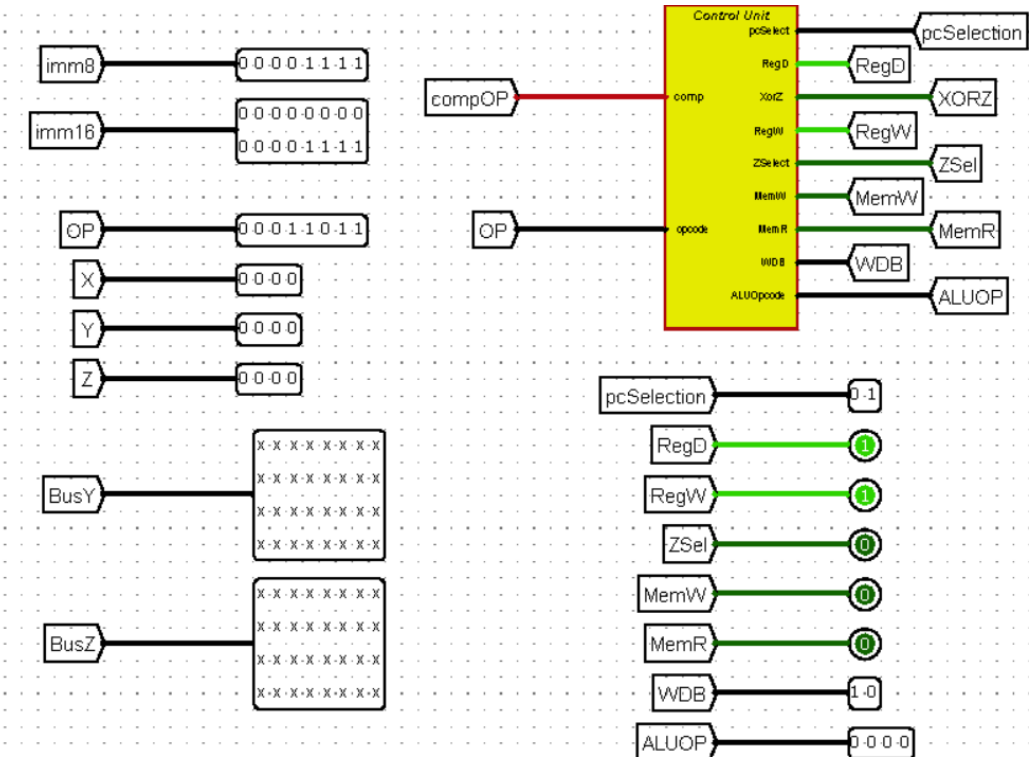


Figure 121: JAL control signal.

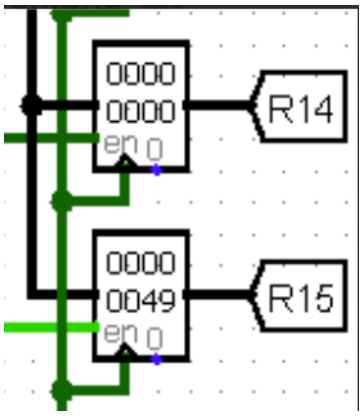


Figure 122: Register file after executing JAL.

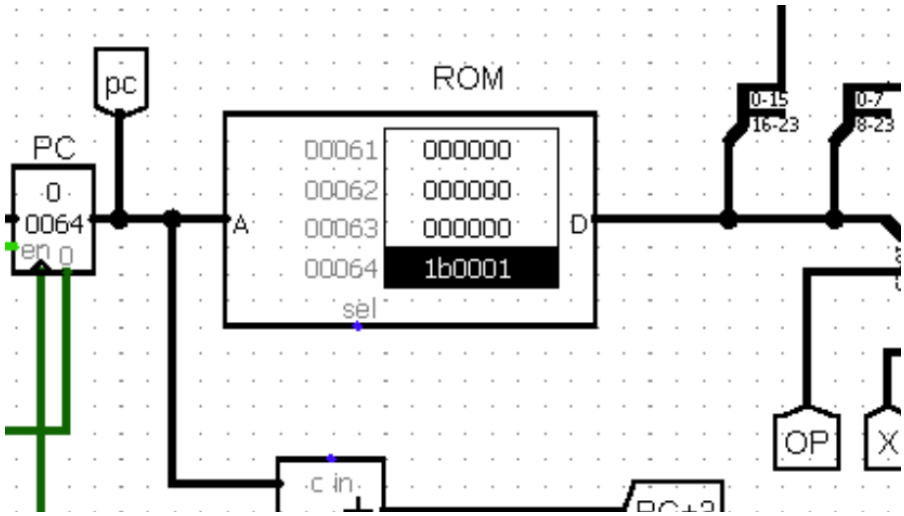


Figure 123: Pc after executing the instruction -now at JAL-.

- 0x1B0001 (JAL)

JAL 1

opcode	Imm16
00011011	0000000000000001

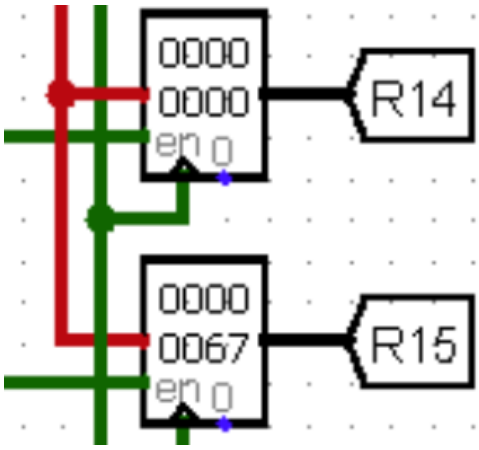


Figure 124: Register file after executing instruction JAL.

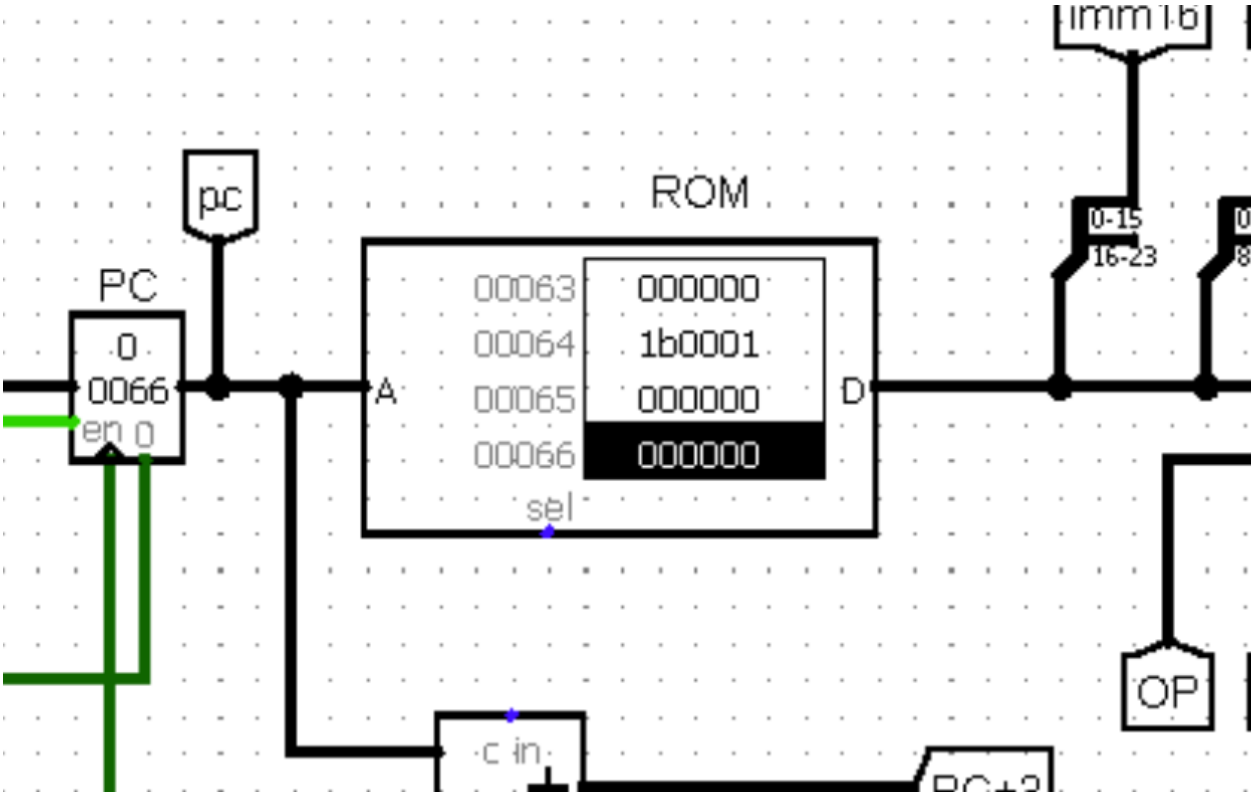


Figure 125: Pc after executing JAL instruction.

Here is simple C++ Code for testing

```
void solve(){
    int R1=0;
    int R2=0;
    int R3=0;

    R1 = R2+5;
    R2 = R1+6;

    cout<<"R1: "<<R1<<endl;
    cout<<"R2: "<<R2<<endl;
}
```

Figure 109: C++ Code for testing.

output.txt

```
1   R1: 5
2   R2: 11
3
```

Figure 110: Expected output.

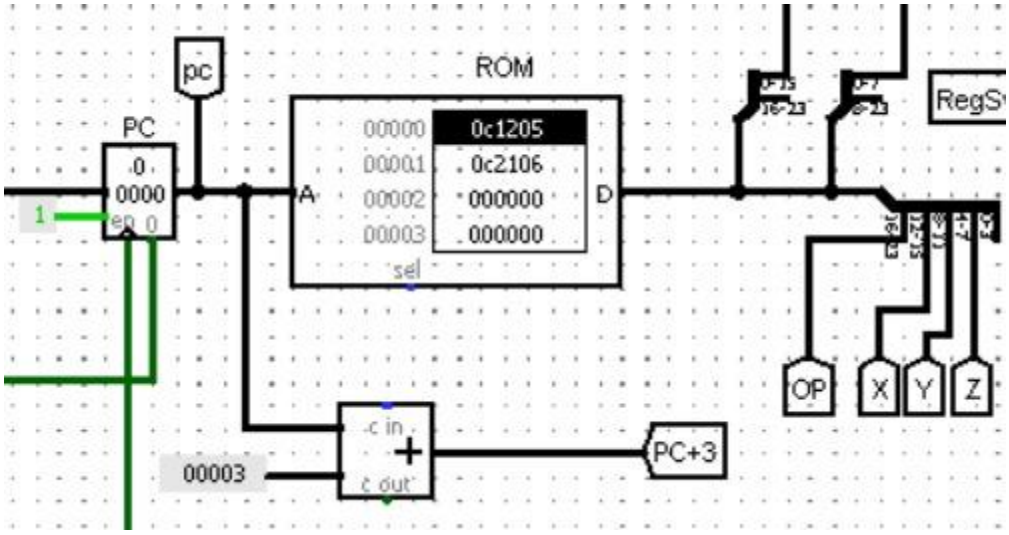


Figure 128: Instructions in ROM.

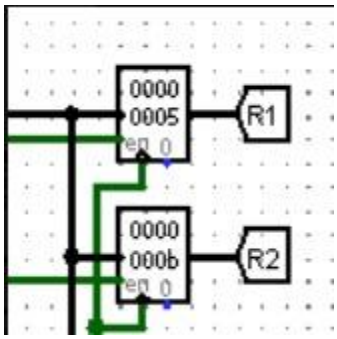


Figure 129: Registers output.

The Assembly Code

1. ADDI 1 2 5
2. ADDI 2 1 6

III. Design Alternatives, Issues and Limitations Part

To execute the Branching Instructions, we needed to be able to execute operations on the Register X which was not simple to do since Register X was always chosen for writing not reading. To fix this we needed to use a mux and redirect X into Z and deal with X as if it was Z in the logic. But this also had a problem due to limitations in the ALU comparison operations we could only support less than and equal to operations. So we needed to work around the limitations and swap the values of Y and Z as into be able to logically support the operations.

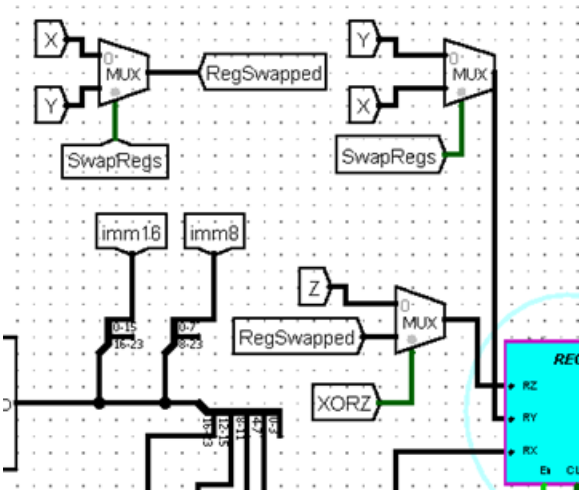


Figure 111: Process for solving first problem.

We also faced a problem in the loading into RAM SW Instruction, since we needed to also access Register X and the fix for the branching instructions of the swapping messed up with the ALU addition logic since we always add Reg z into immediate so we needed to also implement logic to determine whether we are in loading instruction or not based on the opcode and whether to swap the Y and Z or not.

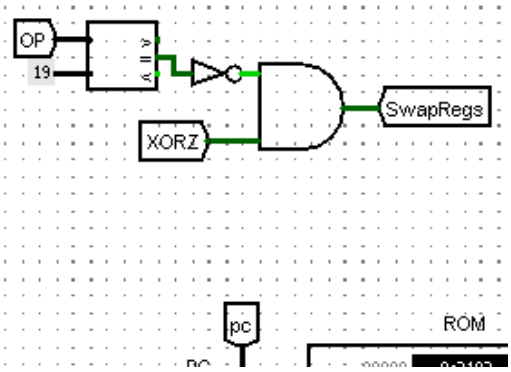


Figure 112: Process for solving second problem..

We also faced problem in generating the Signals for the branching Instructions since the signal depended on the result of the ALU comparison so we needed to put an extra input of the result of comparison into the Control Unit and implement logic inside of the CU to determine whether we are in a branching operation and generate different pcSelection Signal depending on the result of the ALU comparison.

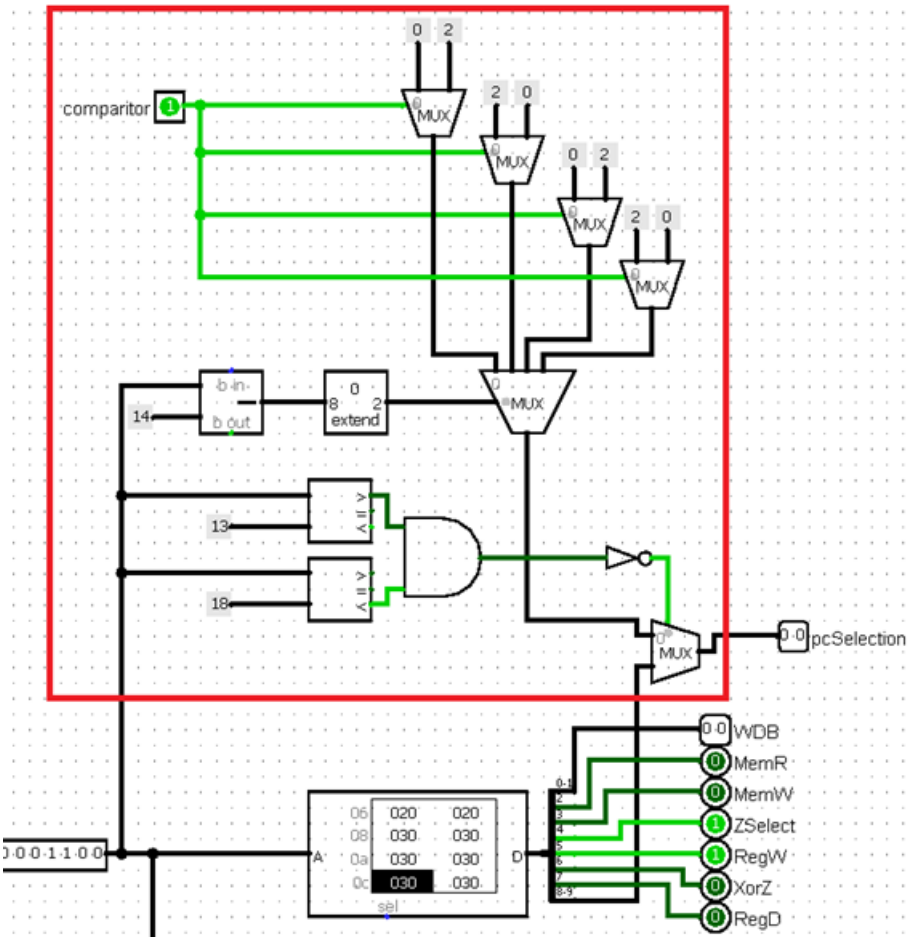


Figure 113: Process for solving third problem.

We also faced many problems with the Signal due to first having the output bits in reverse order and sometimes not having the correct Signal for certain opcodes which took a lot of time to debug. To catch these mistakes and other logical errors we had to output the entire signal and all operands and ALU result to more efficiently find out from where the problem was coming from for the Instruction and fix it.

IV. Teamwork

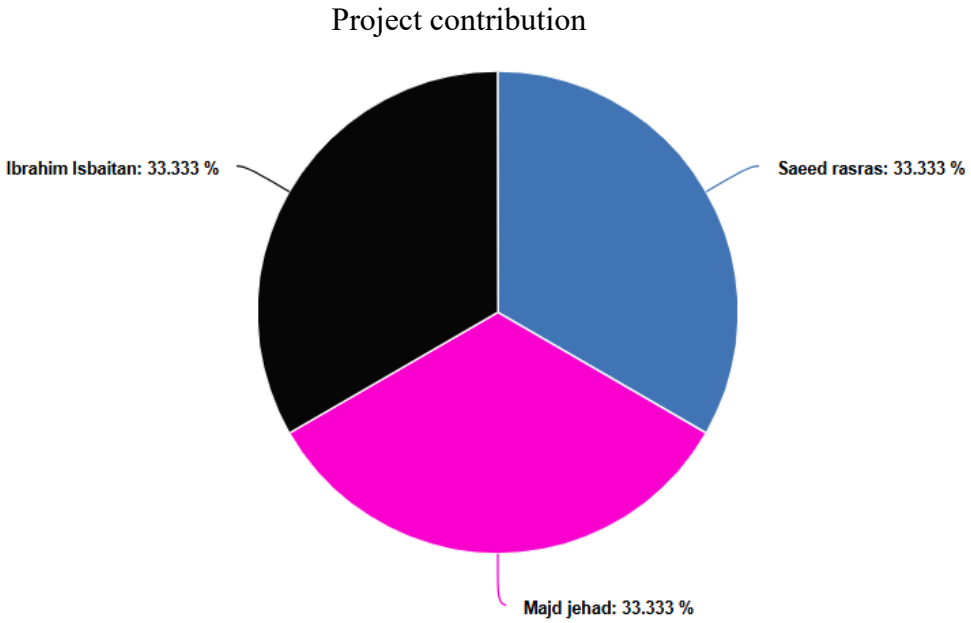


Figure 114: Project contribution