Faculty of Engineering & Technology Electrical & Computer Engineering Department
ENCS2380: Computer Organization and Microprocessor
Single Cycle Processor Design Project
Report

Prepared By:

Saifeddin Saleh 1231474

Omar Shobaki 1230329

Ibrahim Irshaid 1231870

Instructor: Abualseoud Hanani

Date: 15/6/2025

Table of Contents

Table of Figures

List of Tables

Register file

The register file contains 16 32-bit registers, R0 is connected to ground meaning that reading from it returns zero value and writing on it has no effect. However, from R1 to R15 are required for the machine's operation.

This file has one write port (Rx) which refers to the destination register and two read ports (Ry and Rz), which are considered the source registers.

To implement the below design, we used three 4 to 16 Decoders, 15 AND gates, 15 registers, 32 tri-state buffers, one clock, and one write enable input to activate the writing process.

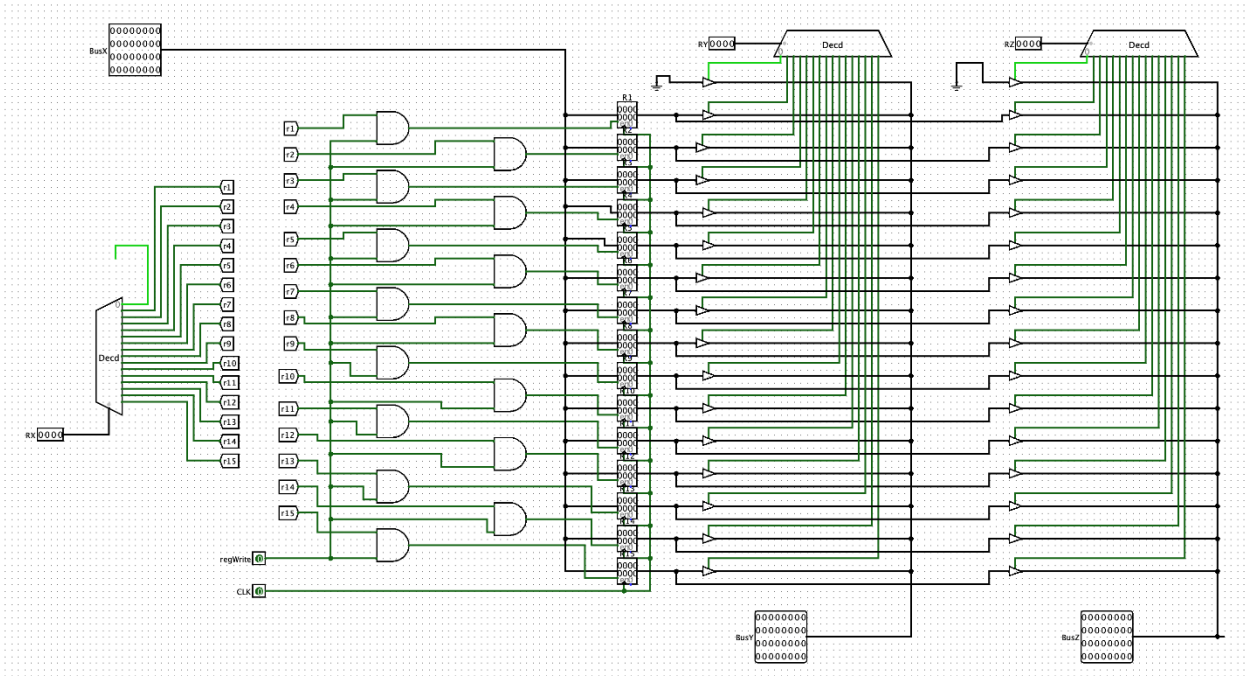


Figure 1: Register File Circuit.

As shown in Figure 1, the main decoder decodes the wanted register to write on specified on the 4 bits input (0001 -> R1, ..., 1111-> R15). Each register takes 2 inputs: one from the clock

and the other is the data to be written from BusX. Furthermore, the output has been chosen based on the 2 decoders which choose the needed register based on the 3 bits each (Ry and Rz).

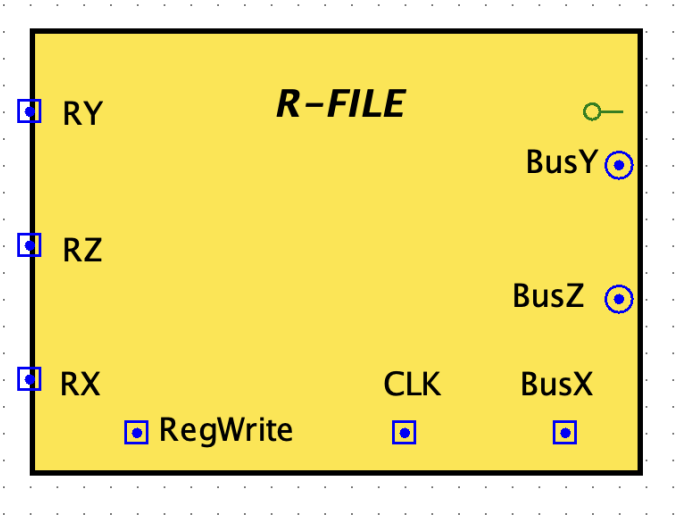


Figure 2: Register File Block.

ALU

This processor has a 32-bit ALU to handle the following operations: XOR, AND, OR, CAND, ADD, SEQ, NADD, SLT, SRA, SRL, SLL, and ROR. As shown in Figures 3, we have used 16x1 MUX, with 4 unneeded inputs.

The Table below describes the functionality of the ALU.

Input No.	Operation	How
0	AND	Basic AND gate.
1	CAND	Invert Ry, then basic AND with Rz.
2	OR	Basic OR gate.
3	XOR	Basic XOR gate.
4	ADD	ADD Ry and Rz.
5	NADD	Invert Ry, then ADD it to Rz.
6	SEQ	Set Rx to 1 if Ry == Rz, else set it to 0.
7	SLT	Set Rx to 1 if Ry<Rz, else set it to 0.
8	SLL	Shift Ry left by immediate value (logical shift).
9	SRL	Shift Ry right by immediate value (logical shift).
10	SRA	Shift Ry right by immediate value (arithmetic shift).
11	ROR	Rotate Ry to the right by immediate value.

Table 1: ALU Operations.

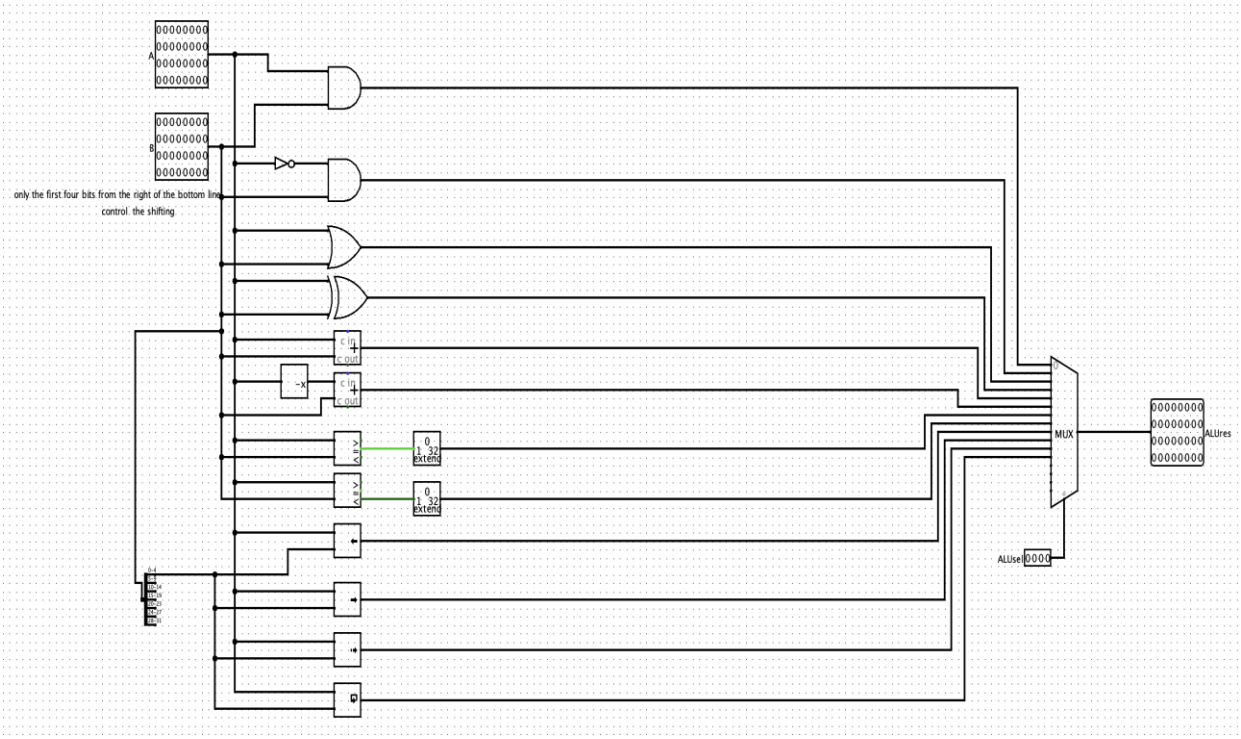


Figure 3: ALU Circuit

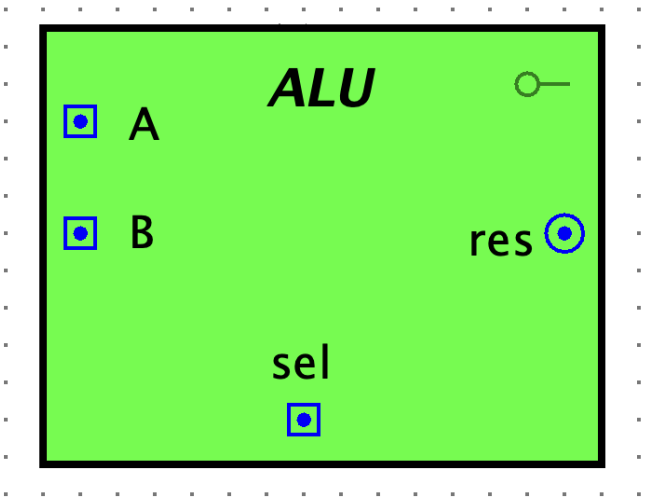


Figure 4: ALU Block.

Control unit

The control unit recieves 1 input, which is the opcode which will be used for all types of instructions and it will the determine the operation to be done. The control unit structure and block is shown the the figure below.

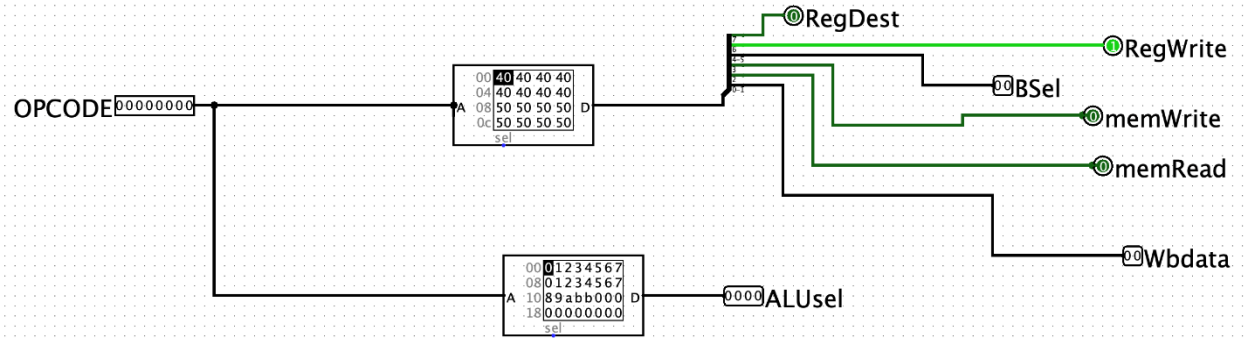


Figure 5: Control Unit Circuit.

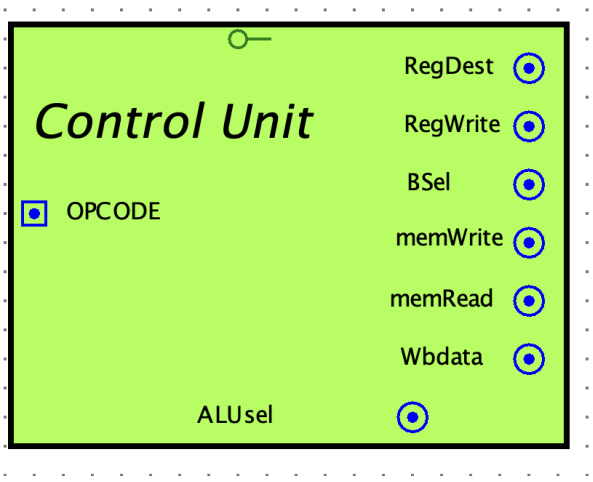


Figure 6: Control Unit.

In the datapath, there are several control signals which will determine how the behavior of the components of the datapath.

1. RegDest(1bit): Register destination chooses the register that will be the destination.

A) Selection 0: Rx will be the destination.

B) Selection 1: R15 will be the destination.

2. RegWrite(1bit): Determines whether to write the data on the destination register or not (writes on register when enabled).

3. Bsel(2bits): This selects the second input for the ALU.

A) Selection 0: BusZ.

B) Selection 1: Imm8(extended to 32bits).

C) Selection 2: Imm16(extended to 32bits).

4. memWrite(1bit): Determines whether to write the data on the memory or not(writes on the memory when enabled).

5. memRead(1bit): Determines whether to read data from the memory or not(read data from the memory when enabled).

6. Wbdata(2bits): Chooses the data to be stored in the destination register.

A) Selection 0: ALU result.

B) Selection 1: RAM output.

C) Selection 2: PC + 3.

D) Selection 3: 0 (gnd).

7. ALUsel: It will retreive a 4-bit control signal that will determine what to choose from the ALU.

Opcode	Control Signals	Operation
0x00	0x40	AND
0x01	0x40	CAND
0x02	0x40	OR
0x03	0x40	XOR
0x04	0x40	ADD
0x05	0x40	NADD
0x06	0x40	SEQ
0x07	0x40	SLT

0x08	0x50	ANDI
0x09	0x50	CANDI
0x0A	0x50	ORI
0x0B	0x50	XORI
0x0C	0x50	ADDI
0x0D	0x50	NADDI
0x0E	0x50	SEQI
0x0F	0x50	SLTI
0x10	0x50	SLL
0x11	0x50	SRL
0x12	0x50	SRA
0x13	0x50	ROR
0x14	0x00	BEQ
0x15	0x00	BNE
0x16	0x00	BLT
0x17	0x00	BGE
0x18	0x55	LW
0x19	0x18	SW
0x1A	0x00	J
0x1B	0xC2	JAL

Table 2: Control Signals.

Figure 7: Control Signals ROM.

Opcode	Location	Operation	ALU operation	ALUsel
0x00	0x00	AND	AND	0x0
0x01	0x01	CAND	CAND	0x1
0x02	0x02	OR	OR	0x2
0x03	0x03	XOR	XOR	0x3
0x04	0x04	ADD	ADD	0x4
0x05	0x05	NADD	NADD	0x5
0x06	0x06	SEQ	SEQ	0x6
0x07	0x07	SLT	SLT	0x7
0x08	0x08	ANDI	AND	0x0
0x09	0x09	CANDI	CAND	0x1
0x0A	0x0A	ORI	OR	0x2
0x0B	0x0B	XORI	XOR	0x3
0x0C	0x0C	ADDI	ADD	0x4
0x0D	0x0D	NADDI	NADD	0x5
0x0E	0x0E	SEQI	SEQ	0x6
0x0F	0x0F	SLTI	SLT	0x7

0x10	0x10	SLL	SLL	0x8
0x11	0x11	SRL	SRL	0x9
0x12	0x12	SRA	SRA	0xA
0x13	0x13	ROR	ROR	0xB
0X24	0x24	SW	ADD	0x4
0X25	0x25	LW	ADD	0x4

Table 3: ALUsel.

Important to note that branch and jump instructions don't use the ALU

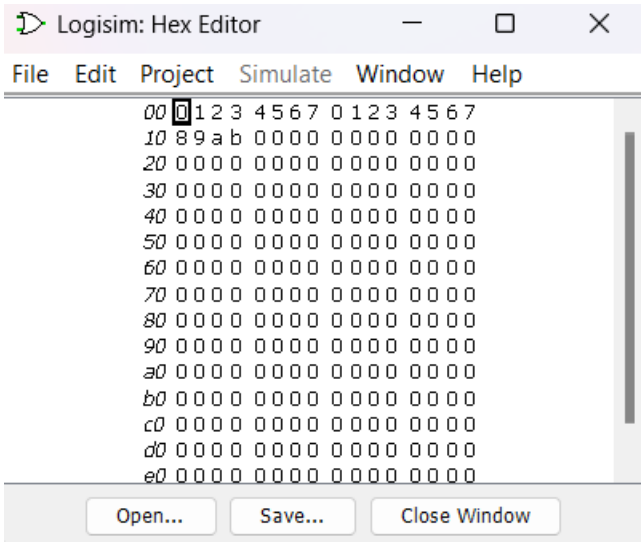


Figure 8: ALUsel ROM.

PC controller:

Used to choose which addressing mode to use or how much to increment the PC by according to the type of instruction we have.

Addressing modes are shown in the table below

Instruction Type	Addressing mode
R-Type	PC+=3
I-Type	PC+=3
Branch instructions (if executed)	PC += Imm8<<1

J-Type (if executed)	PC + signed (Imm16<<1)
Branch/J-Type (if not executed)	PC+=3

Table 4: PC Addressing Modes.

The PC controller has two main components. one that checks if the requirement is met, like BEQ, which checks if they are equal, and the other one is a logic circuit with a priority encoder to decide which addressing mode we need to use.

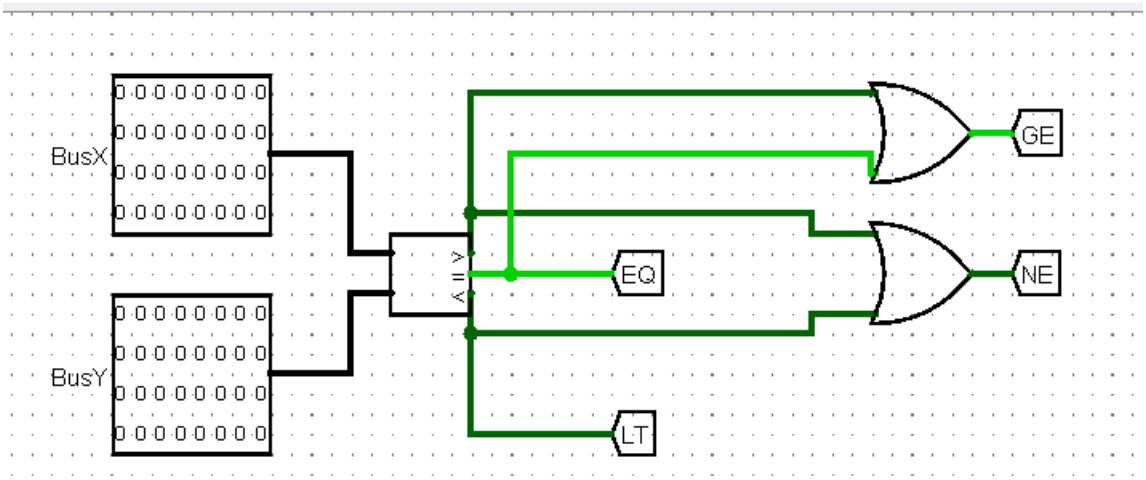


Figure 9: Branch Comparison.

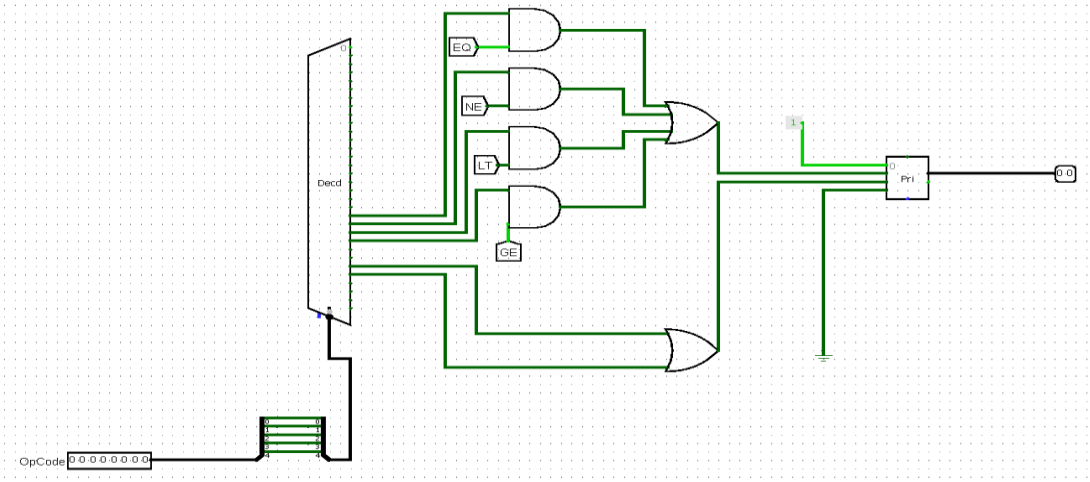


Figure 10: Branch Comparison.

It's important to note, we only use it for instructions that may have a different PC addressing modes as the rest use the default.

Data Path

The data path consists of 7 major parts, PC controller, Register file, ALU, Control unit, RAM, ROM, and the RegSel.

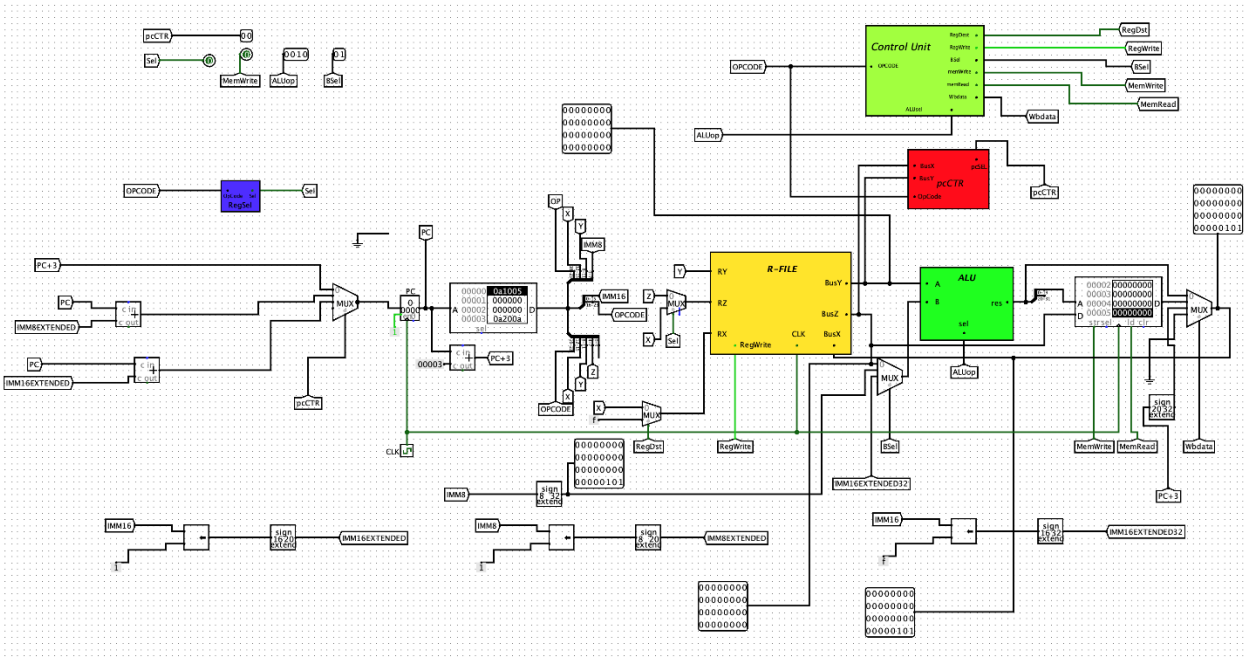


Figure 11: Data Path Circuit.

This datapath represents a simplified processor that follows a typical instruction cycle: fetch, decode, execute, memory access, and write-back. The Program Counter (PC) fetches instructions and is updated through a multiplexer that selects between PC + 3, a branch target, or a jump address. The instruction is decoded into fields like opcode and registers, and the control unit generates signals to manage data flow and operations. The register file reads values from source registers and may write results back depending on the RegWrite signal. The ALU performs computations using register or immediate values, and its result is either stored in memory or written back to the registers. Data memory is accessed if needed, and the write-back stage selects between ALU or memory output. Control and multiplexer logic handle branching and instruction-specific routing throughout the datapath. For the RegSel it is used to fill in Rz for the I-Type instructions and branches handling. More about the RegSel follow appendix(1).

Simulation and Testing:

Every instruction we are going to test is either one of R-Type, I- Type, or J-Type and for each of these ones there is a different instruction structure shown in the table below.

Instruction Type	Instruction Format
R-Type	OpCode(8)-Rx(4)-Ry(4)-Rz(4)-Unused(4)
I-Type	OpCode(8)-Rx(4)-Ry(4)-Imm8(8)
J-Type	OpCode(8)-Imm16(16)

Table 5: Instruction Types.

1) ORI: we used this first to load data into some registers to test before testing other different operations.

   First use: ORI R1, R0, 5, which basically loads R1 with the value 5.

   Machine code: 00001010000010000000000101 = 0x0A1005.

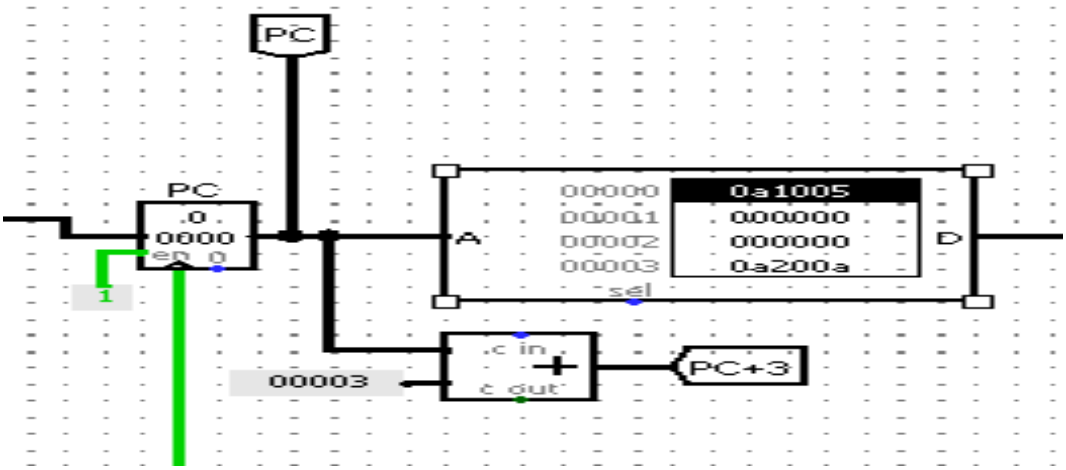


Figure 12: ORI.

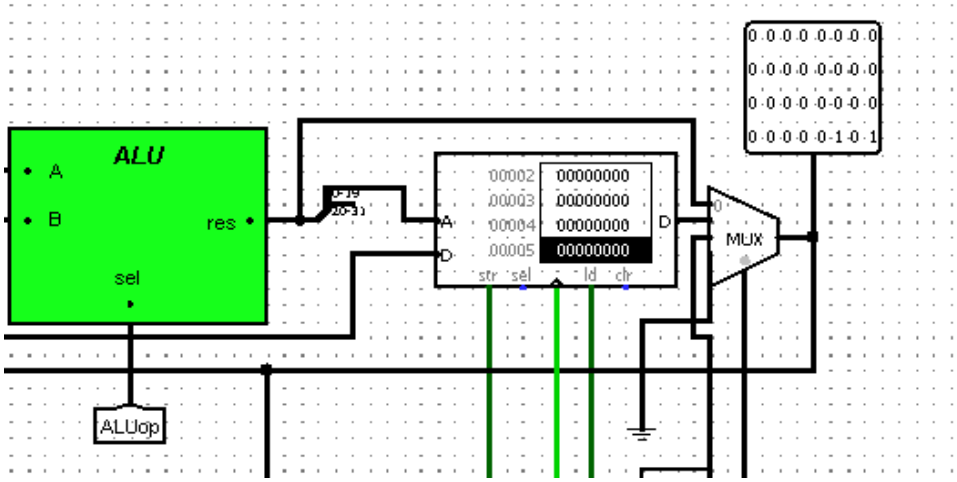


Figure 13: ORI.

Here the result of Rx = R0 (default value is all zeros) or 5 so the result stored in Rx(R1) is 5.

PC is incremented to 3.

Second use: ORI R2,R0, 10 which is just another way of loading R2 with decimal value 10.

Machine code: 00001010001000000000001010= 0x0A200A.

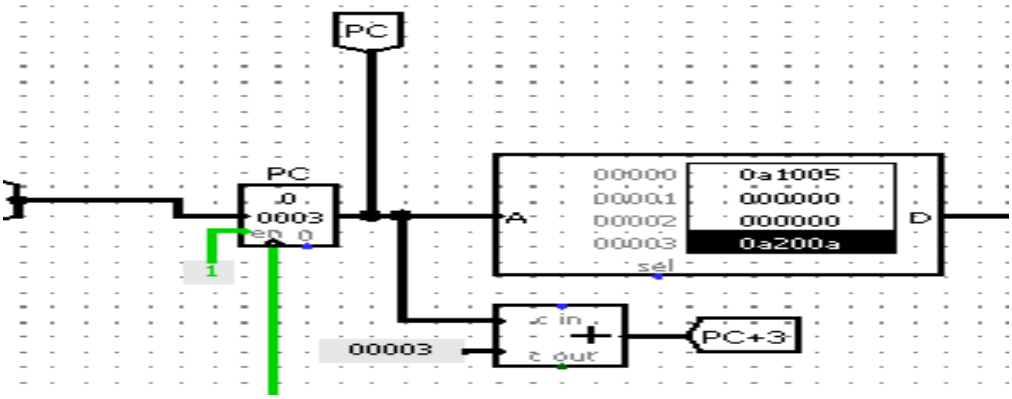


Figure 14: ORI.

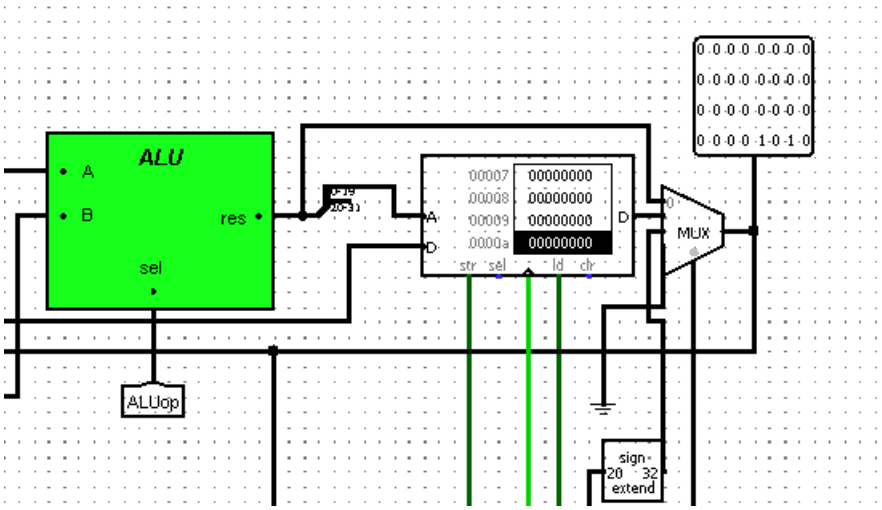


Figure 15: ORI.

Here the result of Rx = R0 (default value is all zeros) or 10 so the result stored in Rx(R2) is 10
PC is incremented by 3 to become 6

2) AND: This is a R-Type instruction that takes the data from two different registers and enters the ALU to perform and AND operation and store the result in a third register. The case we're testing AND R3,R1,R2 which means perform and on R1 and R2 and store the result in R3.
   Machine code: 00000000000110001001000000= 0x003120

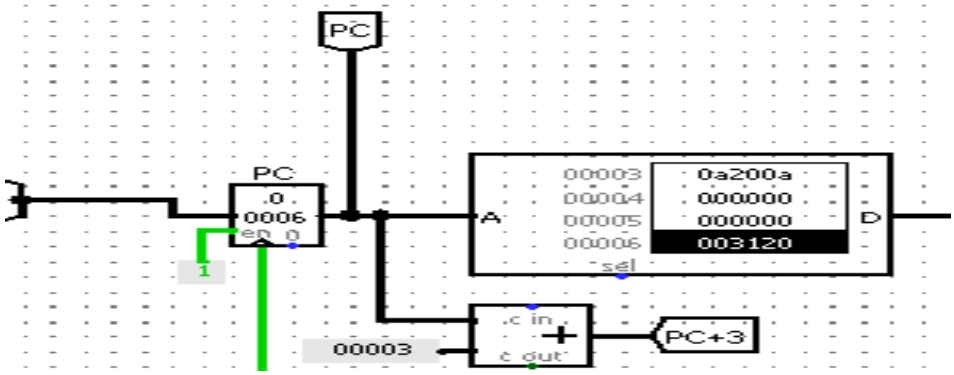


Figure 16: AND.

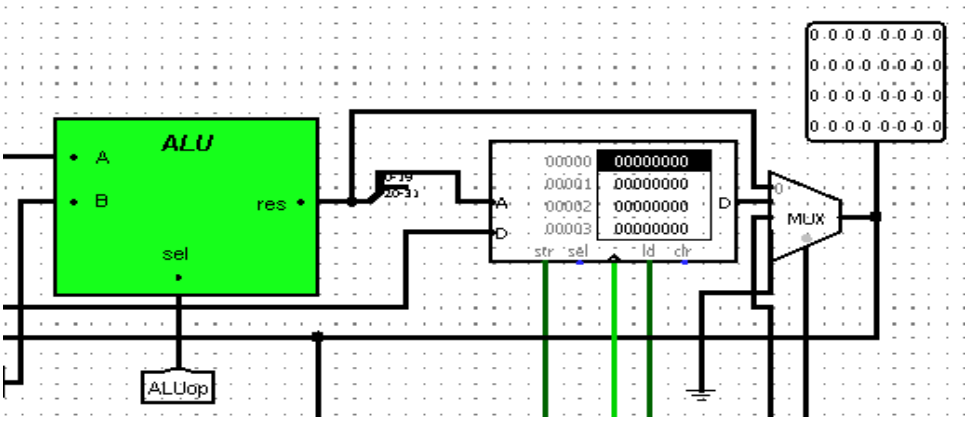


Figure 17: AND.

The Value of R3 (shown in Rx here) is 0 because we performed an and operation on 5(0101) and 10(1010) which means the result of the ALU is zero which is then stored in R3.

3) OR: This is a R-Type instruction that takes the data from two different registers and enters the ALU to perform an OR operation and store the result in a third register

   The case we're testing OR R4,R1,R2 which means perform OR on R1 and R2 and store the result in R4.

   Machine code: 00000010010000001001000000= 0x024120.

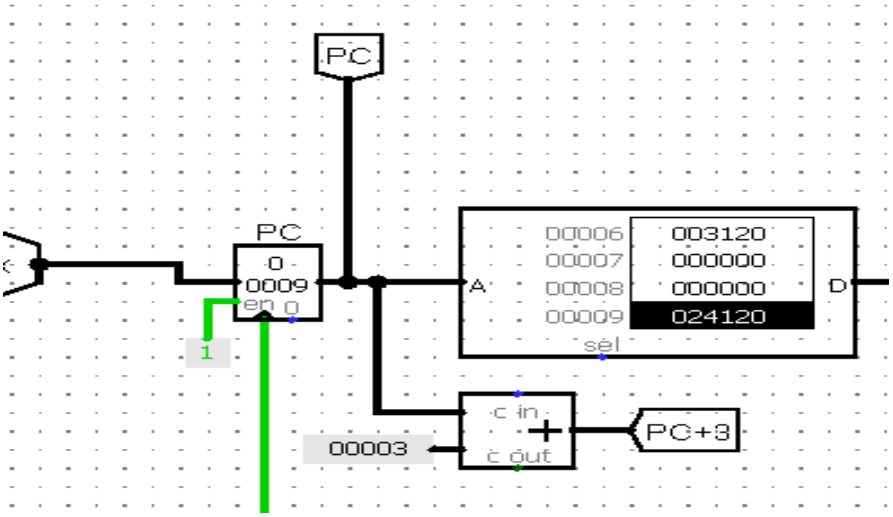


Figure 18: OR.

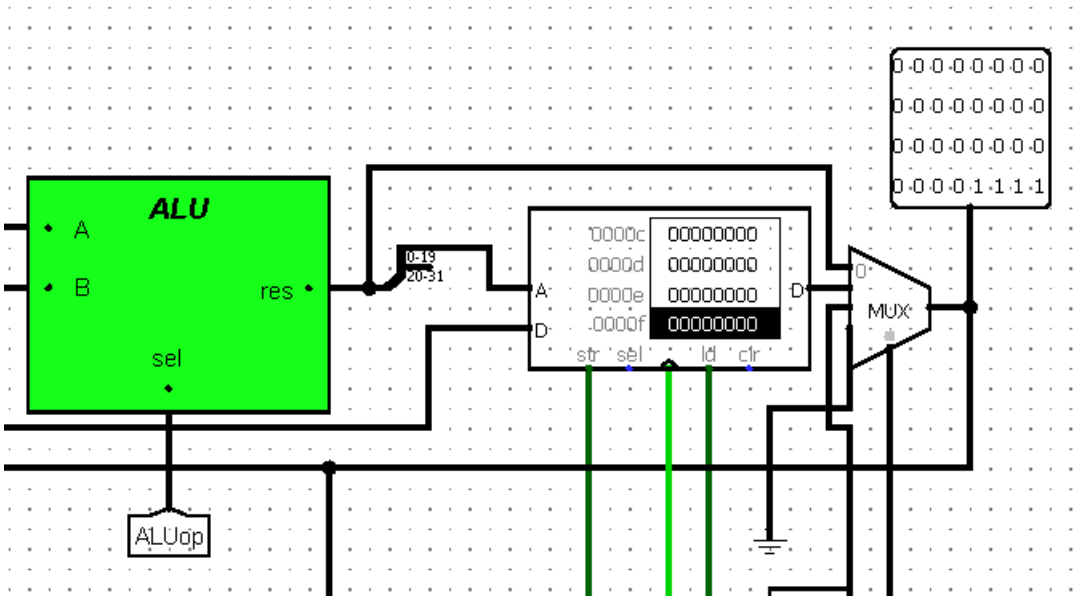


Figure 19: OR.

The value of R4 is 15(1111) which is the result of the OR operation between R1(0101) and R2(1010) which means it works as intended.

4) ADD : Also an R-Type operation that takes the values of two registers and stores the sum of their data into a third register. We are testing the case of ADD R5, R1, R2 which means R1 and R2 enter the ALU and the result of their sum is stored in R5.
   Machine code: 00000100010100010010000= 0x045120.

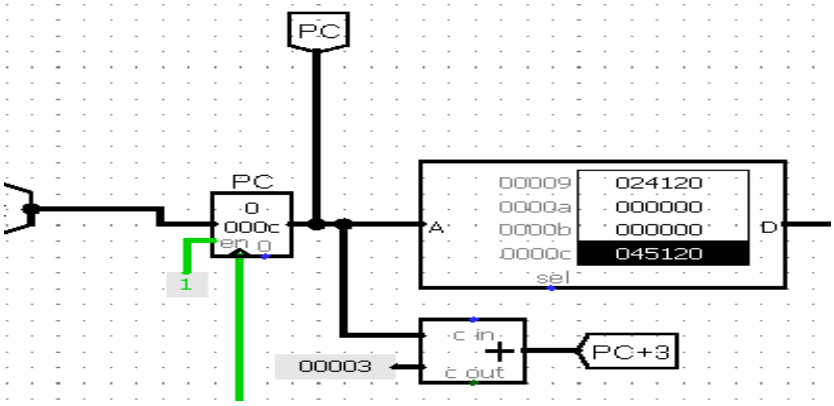


Figure 20: ADD.

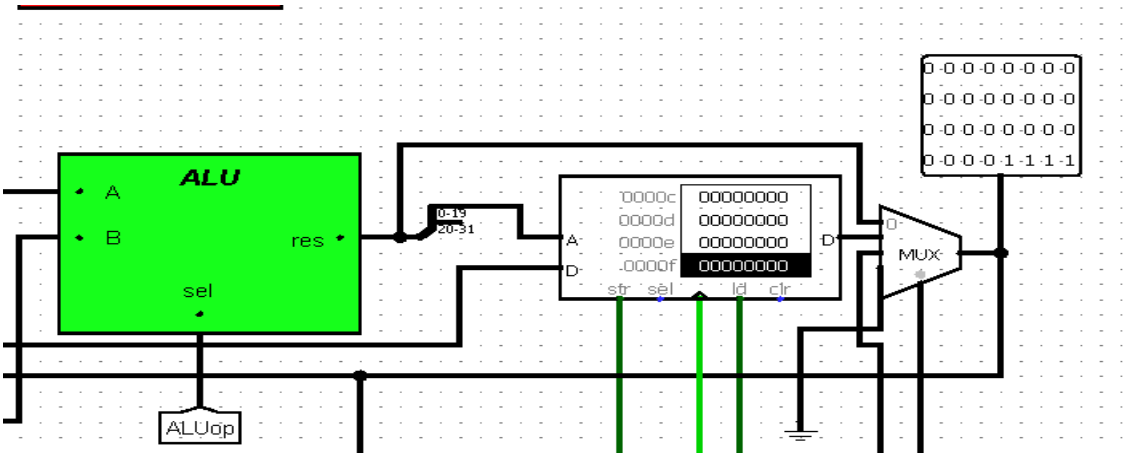


Figure 21: ADD.

Now this shows that the value of R5 is now 15(1111) which is the sum of R1=5(0101) and R2= 10(1010) demonstrating that our result is right.

Values of registers so far

Register	Value
R1	5
R2	10
R3	0
R4	15
R5	15

5) SEQ: A R-Type instruction that compares two registers if their value is equal a value of 1 (they're equal) is stored in a 3rd register, else a value of zero is stored there.

   SEQ R6,R1,R1.

   Machine code= 00000110011000010001000 0= 0x066110.

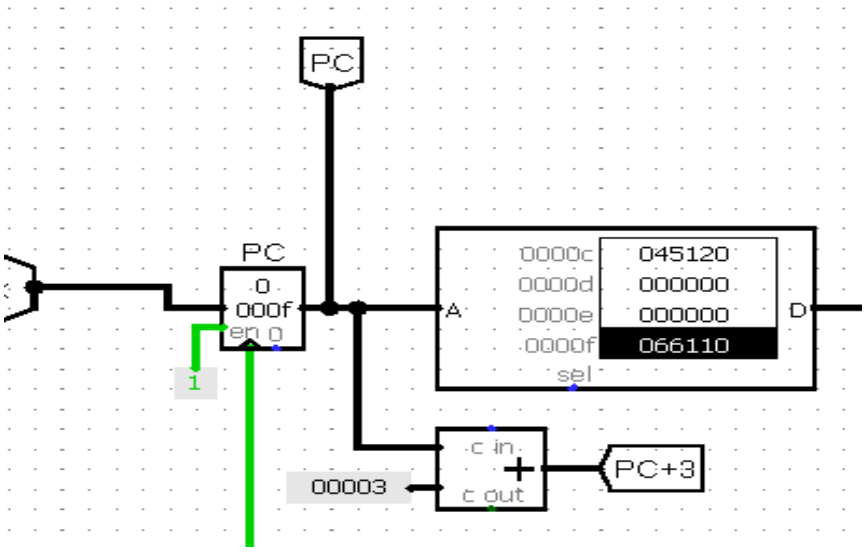


Figure 22: SEQ.

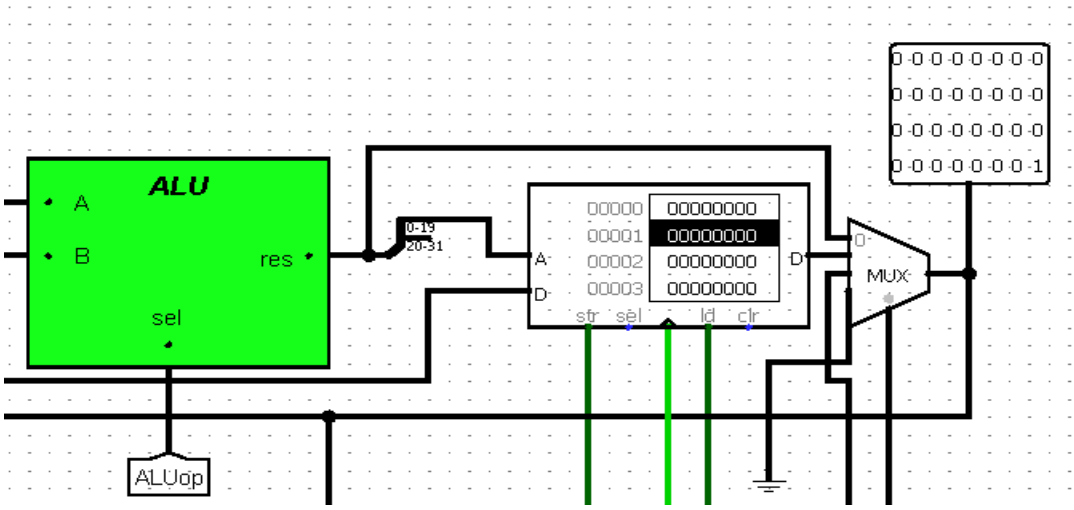


Figure 23: SEQ.

This instruction stores a value of 1 in R6 because the values of R1 and R1 are equal.

6) SLT: R-Type instruction that takes two registers, Ry and Rz, and stores 1 in the third register if Ry is less than Rz; else, it stores zero.

   SLT R7, R1,R2.

   Machine code: 00000111011100010010000 = 0x077120.

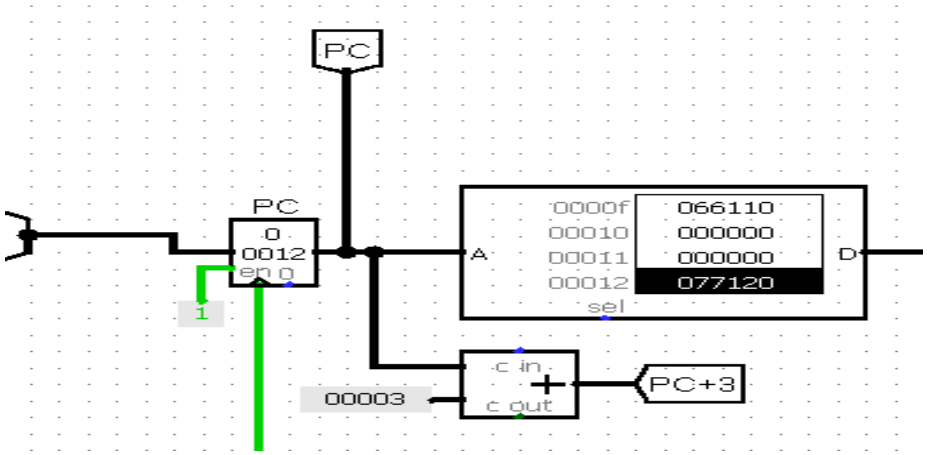


Figure 24: SLT.

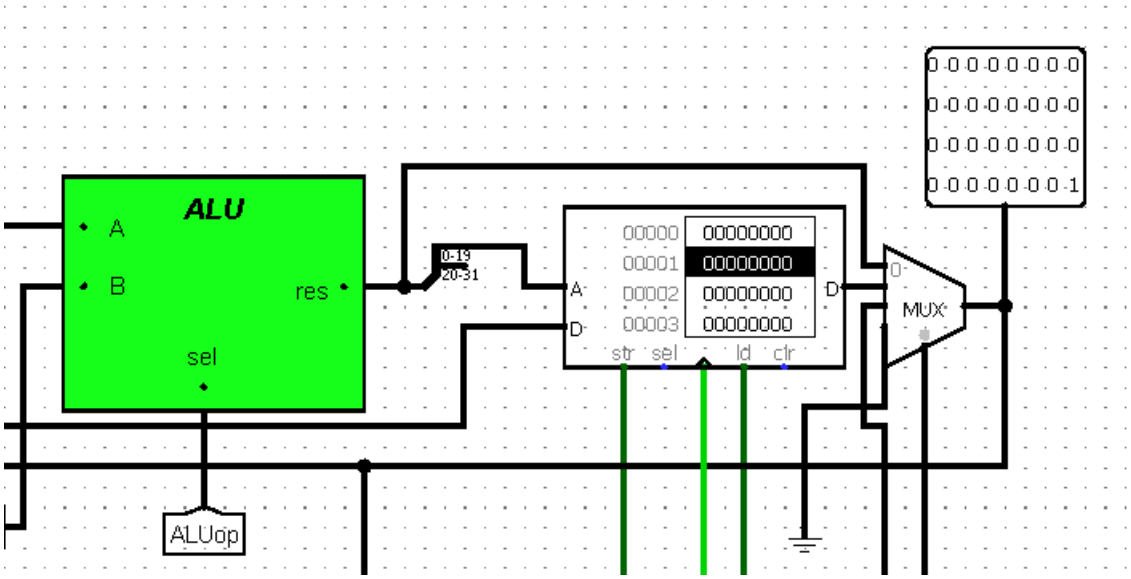


Figure 25: SLT.

The instruction stores a value of 1 into R7 because R1(5) is less than R2 (10).

7) ADDI: I-Type instruction that adds an immediate constant to the data stored another register and stores the data in another register.

   ADDI R8,R1,3.

   Machine code: 000011001000000100000011 = 0x0C8103.

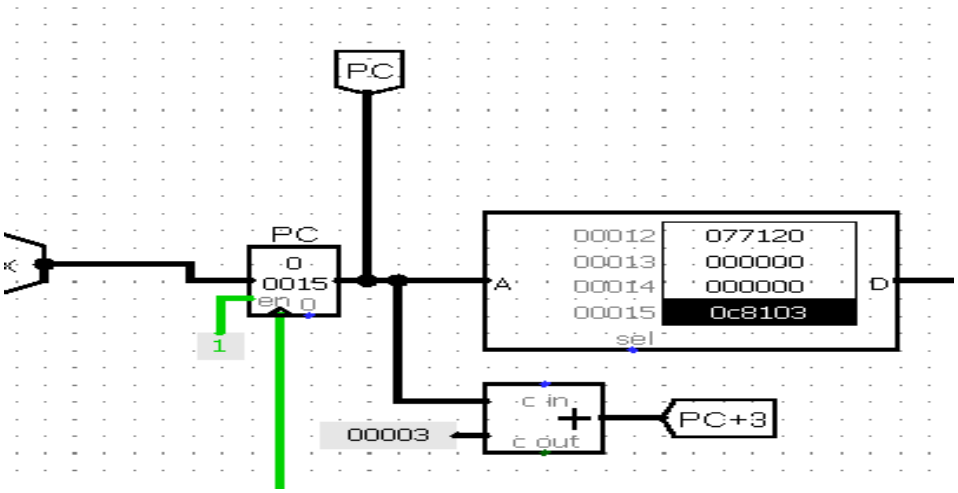


Figure 26: ADDI.

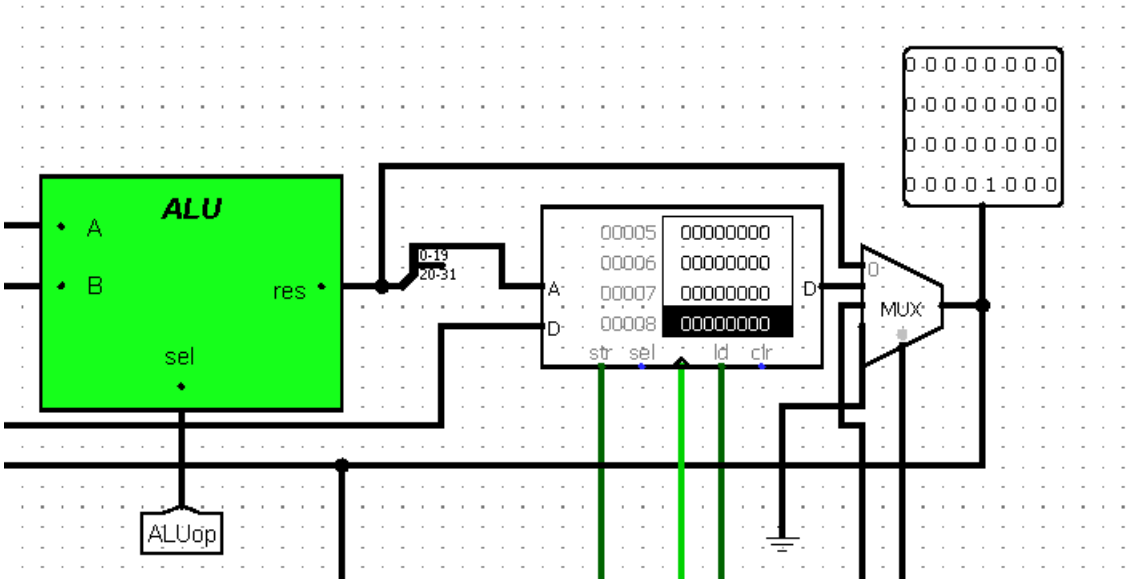


Figure 27: ADDI.

The result of this instruction is storing 8(1000) in R8 which is the sum of the data in R1(5) and the immediate constant 3.

8) SLTI: I-Type instruction similar to SLT difference is instead of comparing the data between two registers, it compares the data between a register and an immediate constant. SLTI R9,R1,7.

   Machine Code: 00001111100100010000011 1= 0x0F9107.

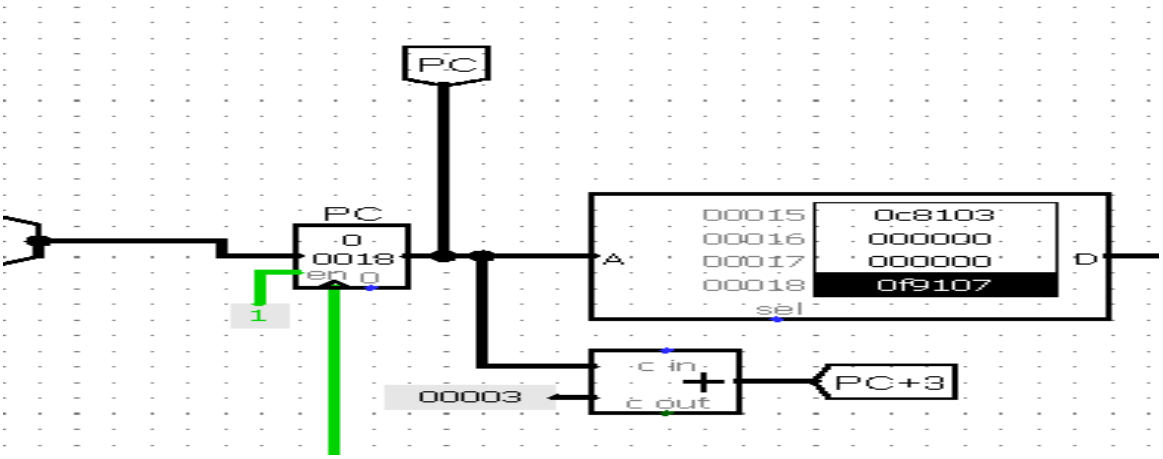


Figure 28: SLTI.

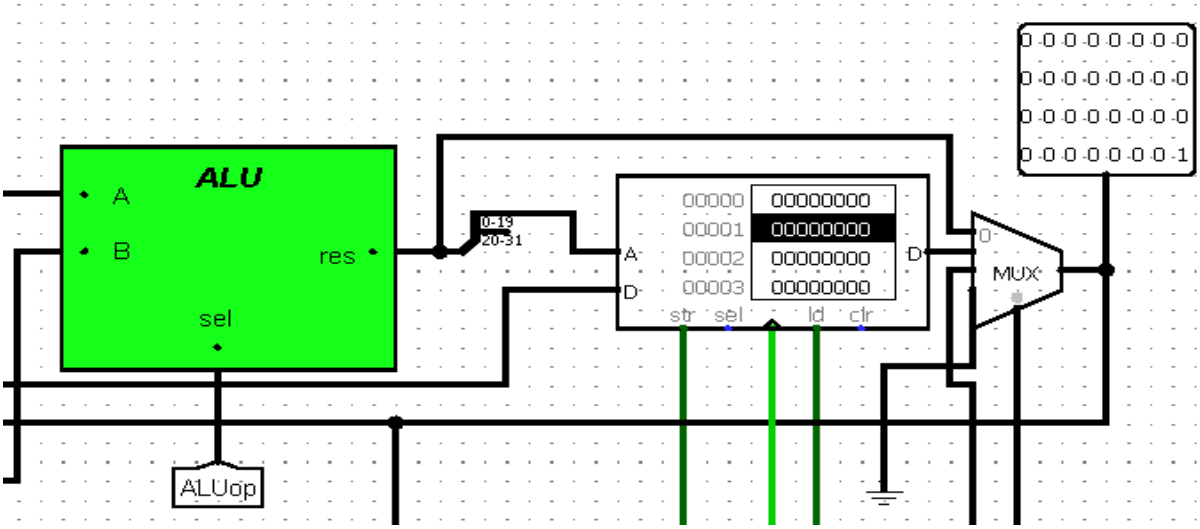


Figure 29: SLTI.

The value of R1 (5) is less than the immediate constant 7 which means a value of 1 is stored in R9.

Updated values of the registers:

Register	Value in decimal
R1	5
R2	10
R3	0
R4	15
R5	15

R6	1
R7	1
R8	8
R9	1

9) SLL: shifts the data in one register an immediate number of bits to the left (or multiply by 2 ^ number of bits shifted) and stores the result in another register.

SLL R10,R1,1.

Machine code: 00010000101000010000000 1 = 0x10a101.

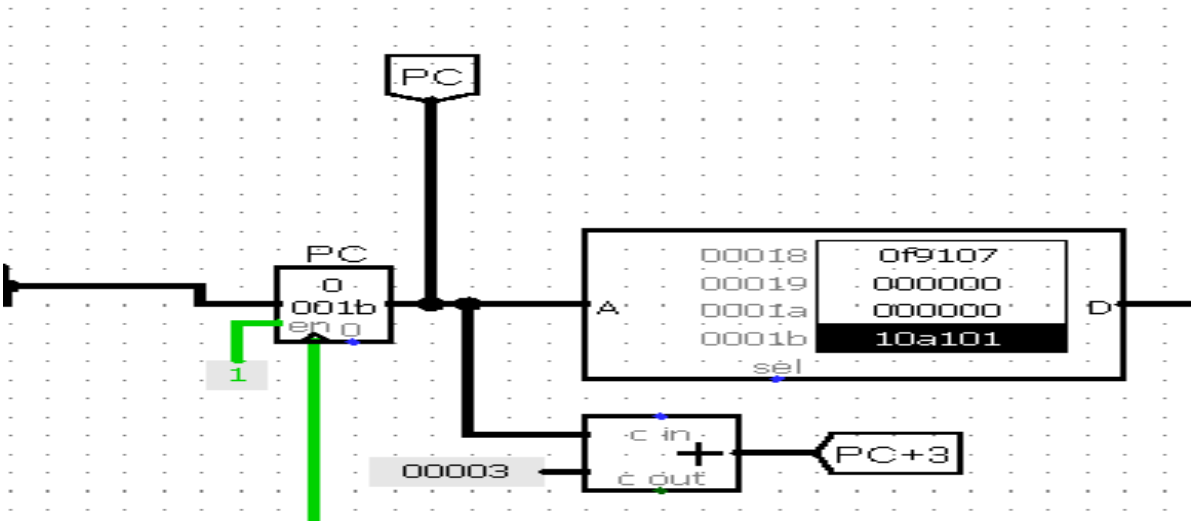


Figure 30: SLL.

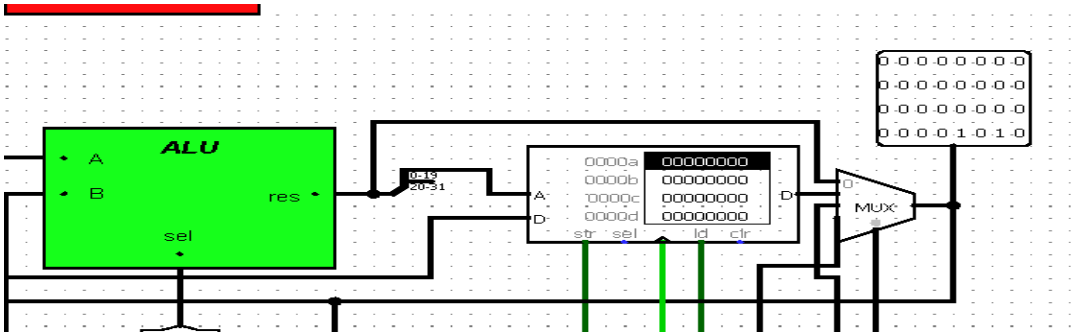


Figure 31: SLL.

This instruction stores a Value of 10(1010) in R10 which is a result of R1= 5 (0101) being shifted to the left 1 bit.

10) ROR: I-Type instruction that is similar to the shifter difference is instead of getting rid of the bits that went out of bounds it places them in the MSB.

ROR R11,R2,2.

Machine code: 00010011101100100000010 = 0x13b202.

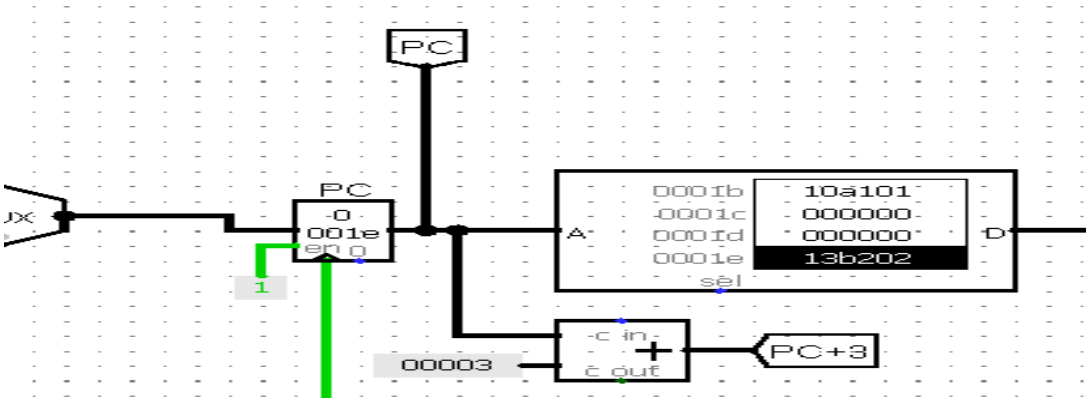


Figure 32: ROR.

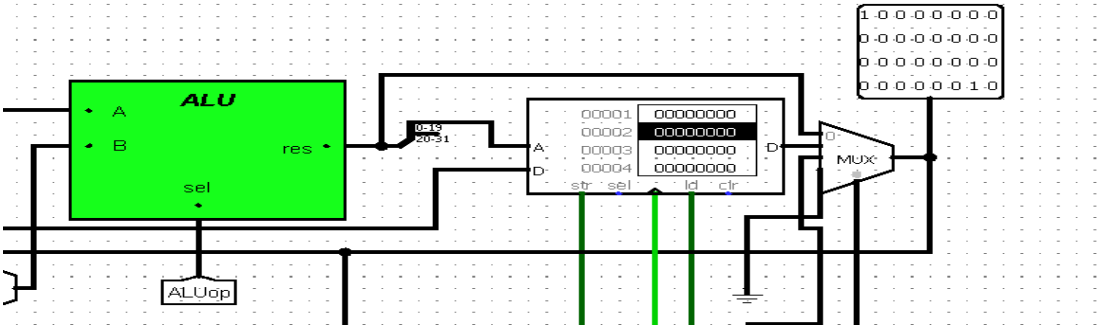


Figure 33: ROR.

We can see that R11 is the result of rotating R2 (1010) two bits as the second 1 goes to the MSB making it an extremely large number.

11) CAND: R-Type instructions that negates the first Register and performs an and operation with the second register while storing the result in the third register.

CAND R6,R2,R3.

Machine code: 00000001011000100011000 =0x016230.

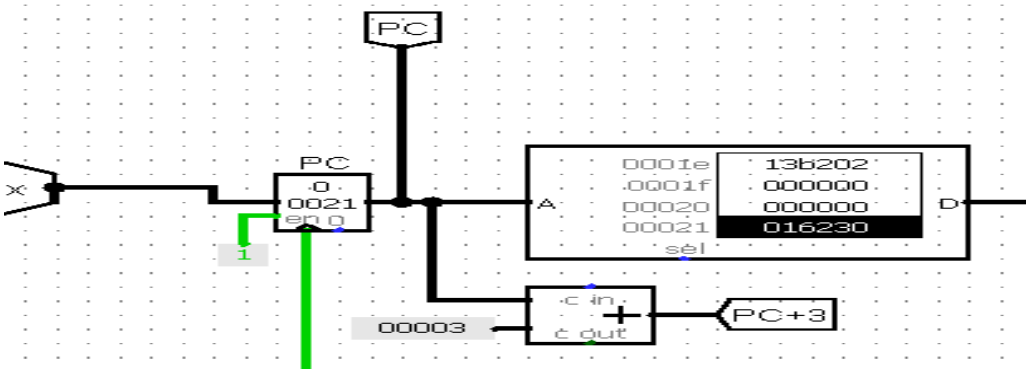


Figure 34: CAND.

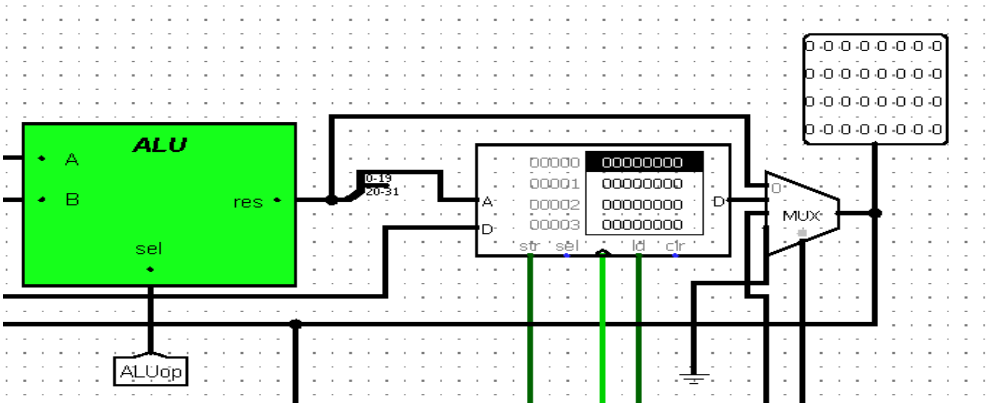


Figure 35: CAND.

This instruction results in zero because R3's Value is 0 so whatever enters the and gate the result will always be zero which is stored in R6.

12) NADD: R-Type instruction which adds the data of the second registers to the 2's compliment of the first register (basically RZ-RY) and stores data in another register.
NADD R8, R3,R2.
Machine code: 00000101100000011001000000 =0x058320.

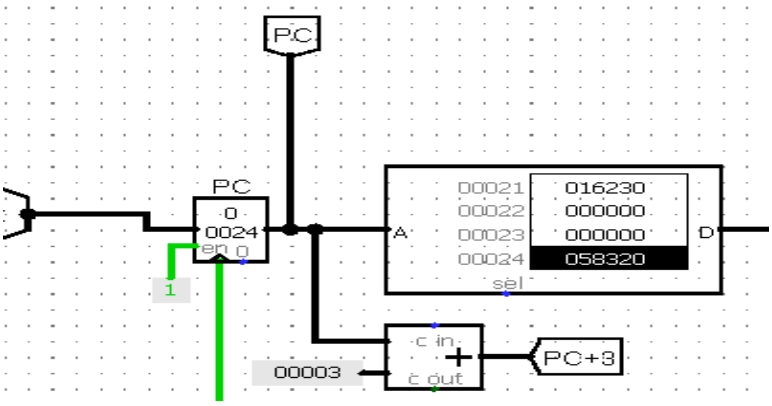


Figure 36: NADD.

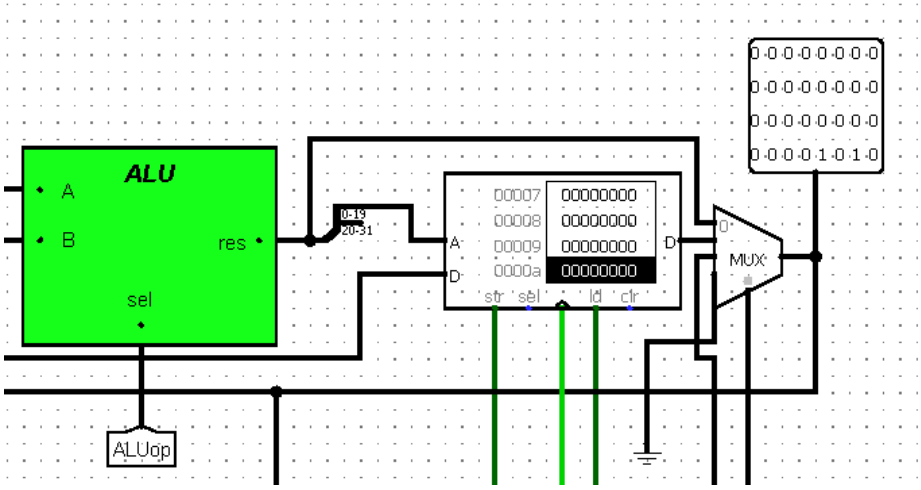


Figure 37: NADD.

The result of 10 which is stored in R8 is a result of -(R3) + R2 but R3 is zero so the result is just R2 which is 10.

13) BEQ: This instruction takes two registers if their data is equal. It increments the PC by the immediate data shifted to the left by one bit, if they are not equal, it continues normally without changing the value of any registers.

BEQ R1,R2,2.

Machine code: 00010100000010010000000010= 0X141202.

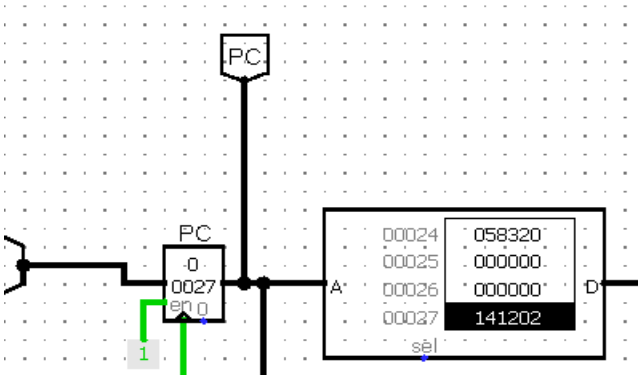


Figure 38: BEQ.

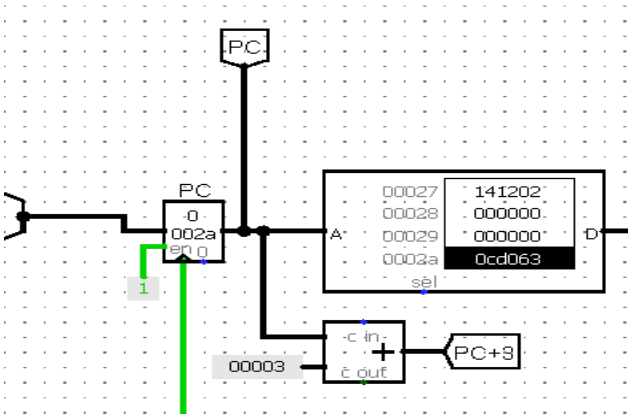


Figure 39: BEQ.

We can see pc only incremented by 3 (default) because R1 and R2 are not equal; therefore, the branch instruction didn't execute.

A case where the branch happens:

This happens a few instructions after

BEQ R1,R1,2.

Machine code: 00010100000100010000010=0x141102.

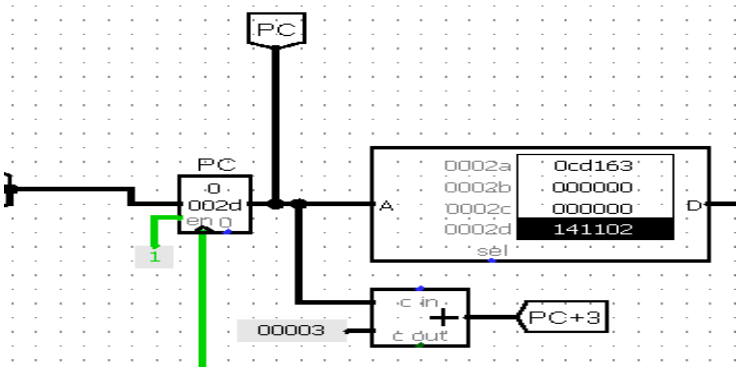


Figure 40: BEQ.

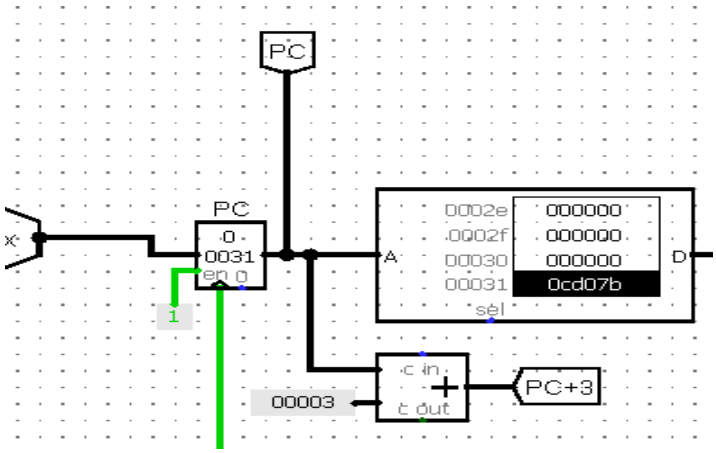


Figure 41: BEQ.

Here we can see as R1 is equal to R1 pc increment by 4 (2*2) and went from 45 to 49.

14) ADDI: I-Type instruction where the data of one register is added to an immediate value and stored in another register.

ADDI R13, R1,99.

Machine code = 00001100110100010110001 1 =0x0cd163.

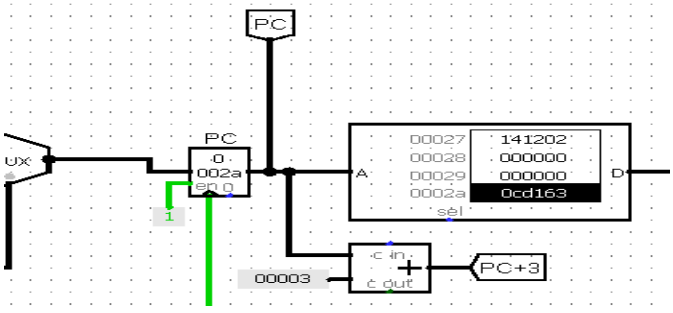


Figure 42: ADDI.

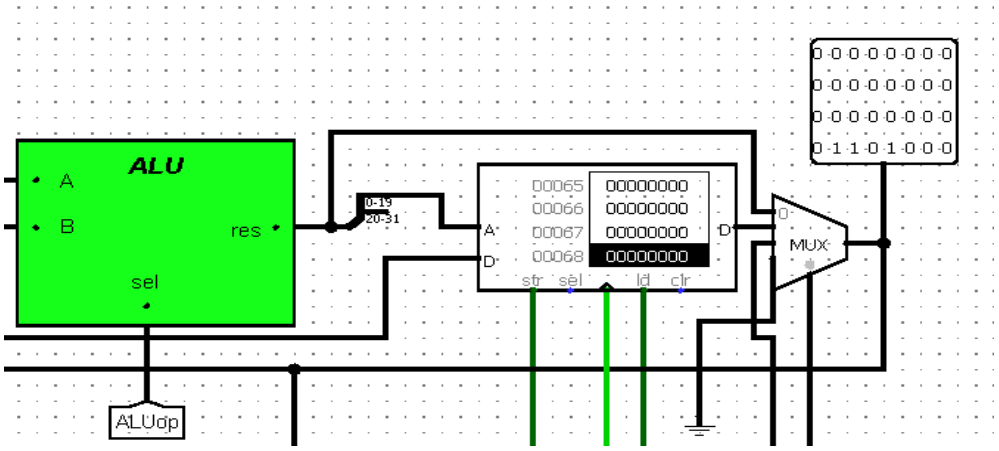


Figure 43: ADDI.

Here R1+ 99 = 5+99 =104 which is the result stored in R13.

15) CANDI: I-type instruction, which does the same as CAND except its a register and an immediate value and the result is stored in another register.

CAND R7,R2,85.

Machine code= 000010010111001001010101= 0x097255.

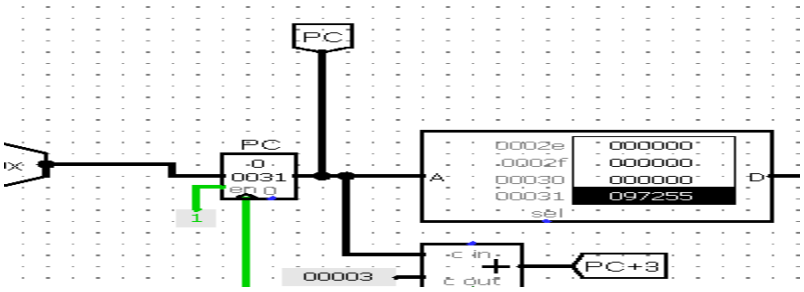


Figure 44: CANDI.

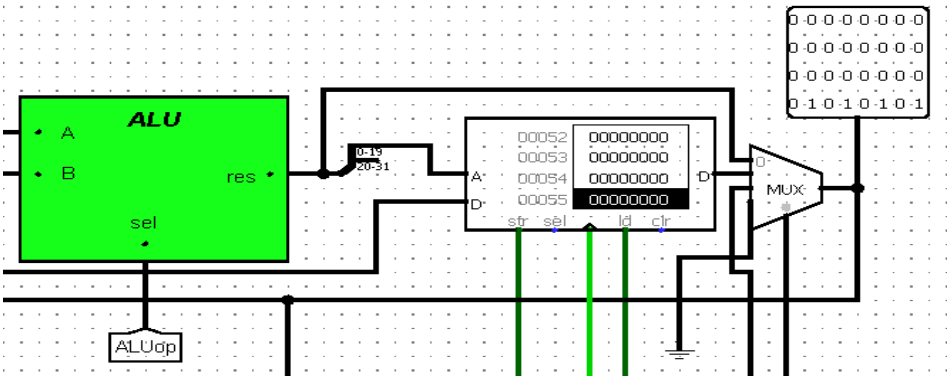


Figure 45: CANDI.

The result shown is the result of the CAND operation between R2(10) and the immediate 85 which also means -(00001010) and (01010101) which is (01010101) which is 85 that is stored in R7.

Update on the values of the registers:

Register	value
R1	5
R2	10
R3	0
R4	15
R5	15
R6	0
R7	85
R8	10
R9	1
R10	10
R11	Very large number (2^31+2)
R12	0
R13	104

16) SEQI: I-Type instruction that compares the value of the register and an immediate constant; if they are equal, it returns 1; else, it returns 0.
SEQI R9,R3,10.
Machine code: 00001110100100110000101O= 0x0E930A.

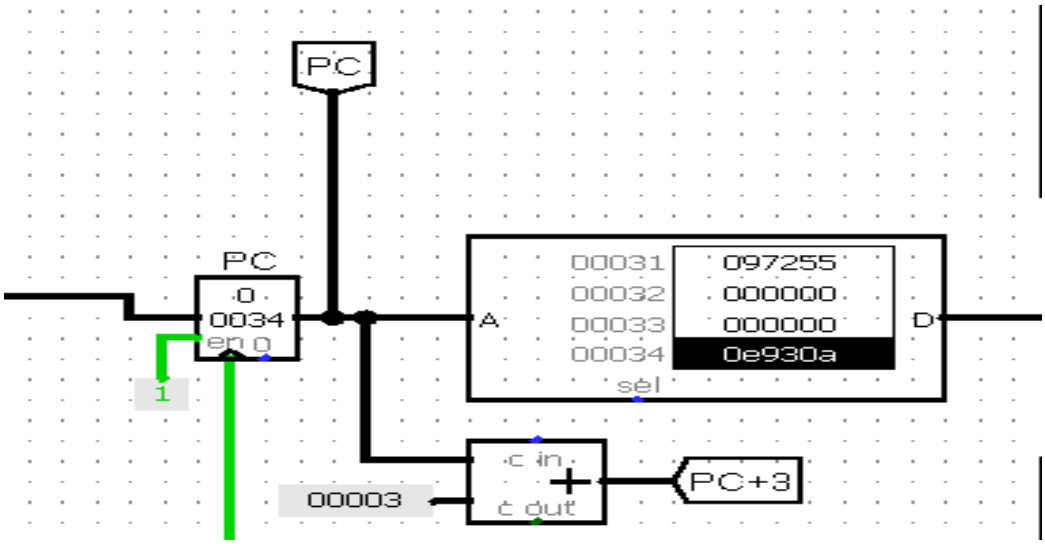


Figure 46: SEQI.

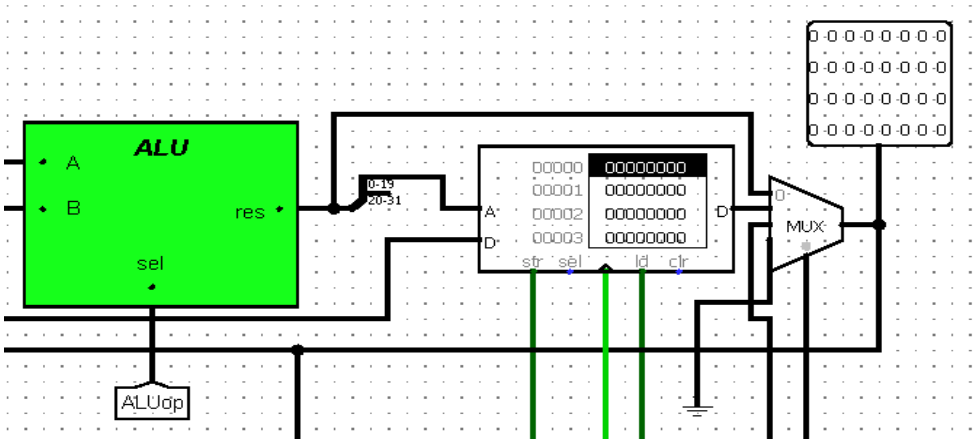


Figure 47: SEQI.

R3 has a value of 0 which doesn't equal the constant value of 10 so a value of 0 is stored in R9.

17) J: short for jump a J-Type instruction that increments the pc by the imm16 value shifted one bit to the left no matter what and it doesn't depend on any other values.

J 2.

Machine code: 00011010000000000000000010=0x1a0002.

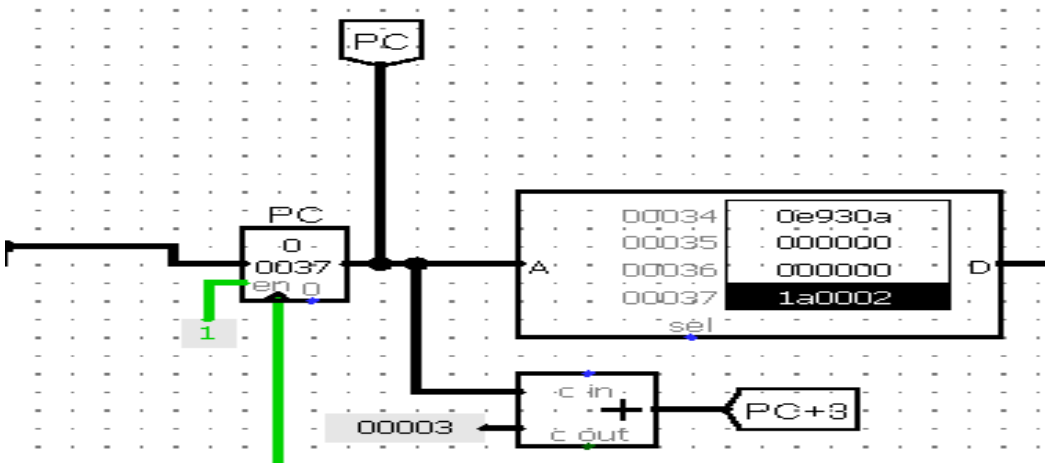


Figure 48: J.

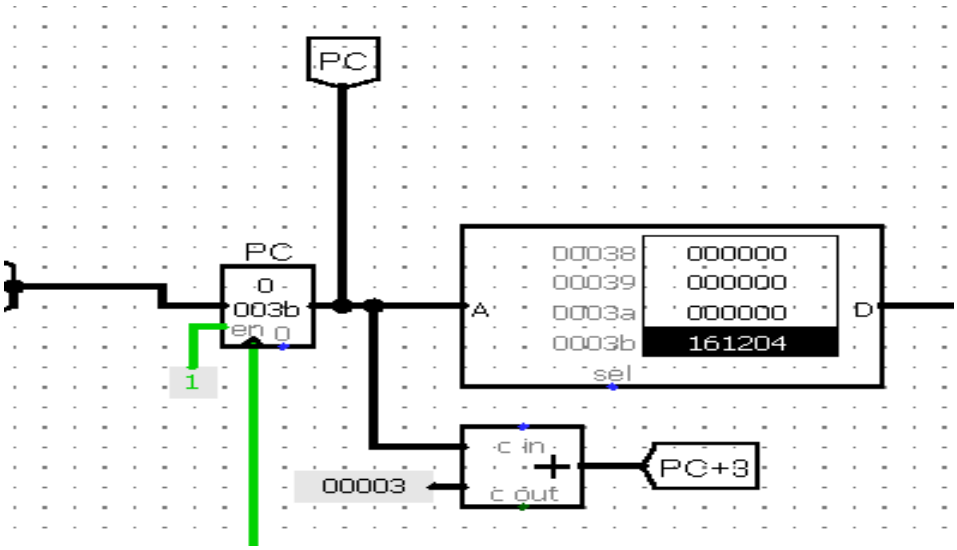


Figure 49: J.

We can see that the address incremented by 4 (2 shifted to the left) going from 0037 to 003b.

18) BLT: branch instruction that increments the program counter by the immediate constant *2 if the first register was less than the second register.

BLT R1,R2,4.

Machine code= 00010110000100100000100=0x161204.

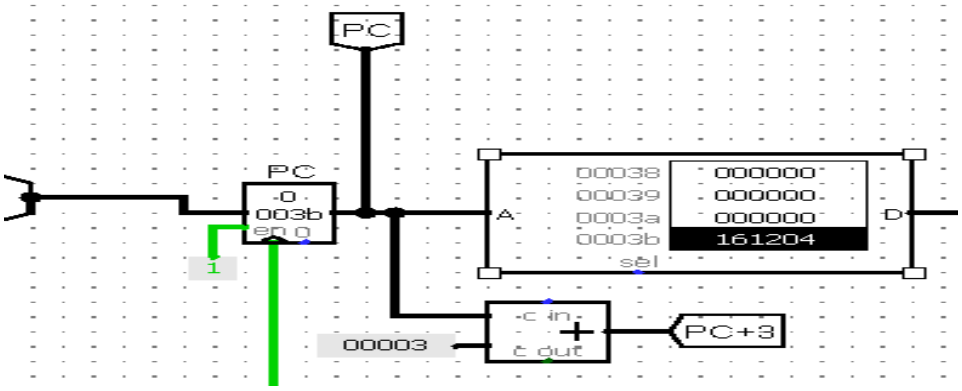


Figure 50: BLT.

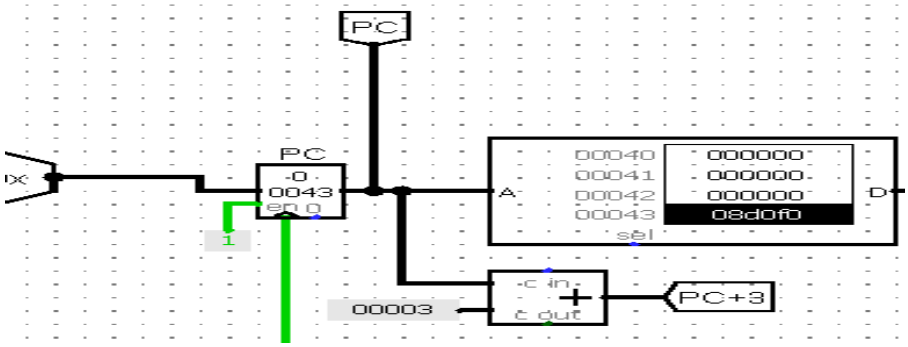


Figure 51: BLT.

Since R1 (5) is smaller than R2 (10) the branch instruction excuted and incremented the pc by 8 (4 shifted to the left).

19) ANDI: I-Type instruction for and, performs an and operation between a register and in immediate constant and stores the result in a different register.

ANDI R13, R0, 240.

Machine code: 00001000110100001111000=0x08D0F0.

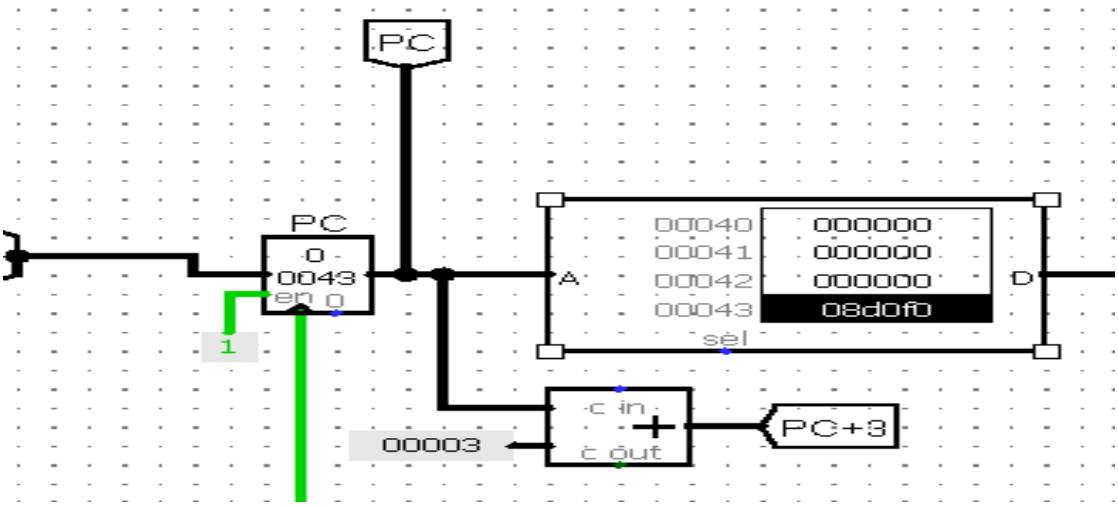


Figure 52: ANDI.

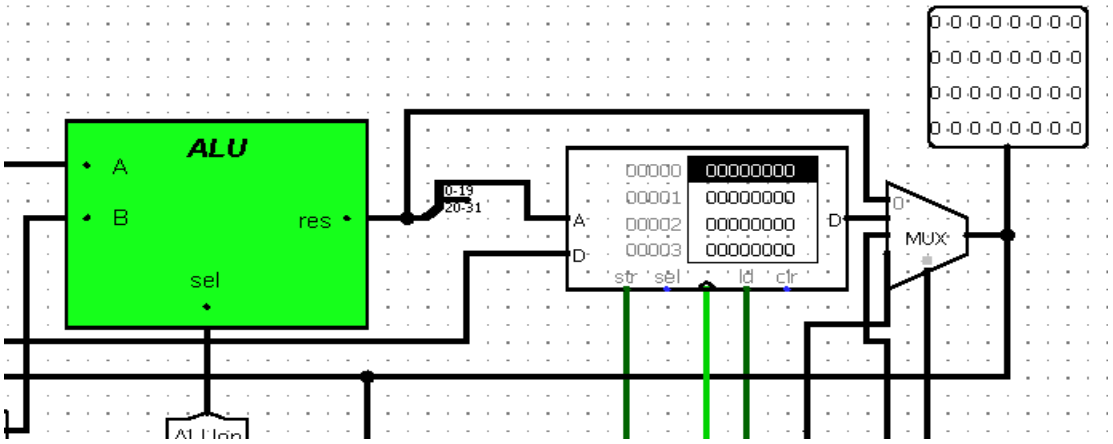


Figure 53: ANDI.

Since r0's value is 0 whatever we put to and with it the result will always be zero which will be stored in R13.

20) JAL: J-Type instruction that stores the value of pc+3 is R15 and then increments the pc by the immediate16 shiftted to the left.

JAL 2.

Machine code= 0001 1011 0000 0000 0000 0010= 0x1B0002.

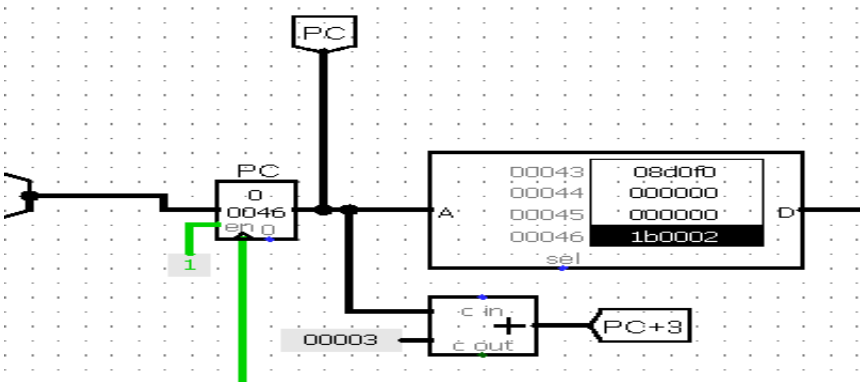


Figure 54: JAL.

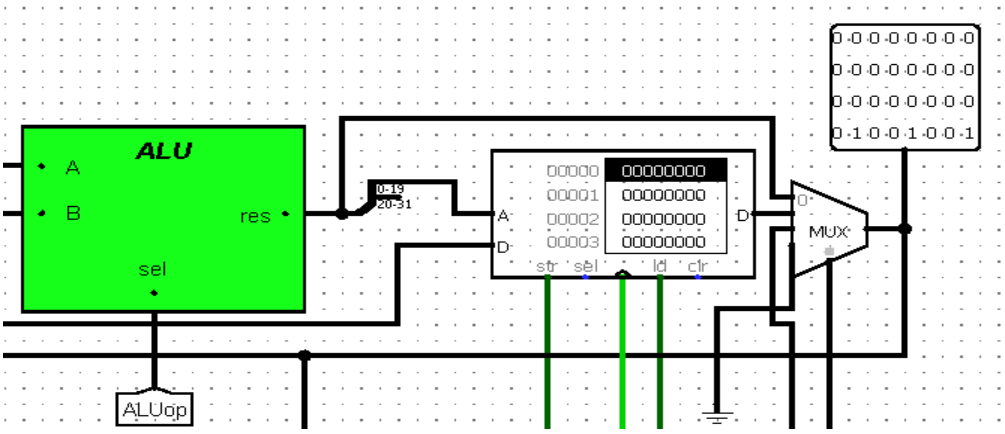


Figure 55: JAL.

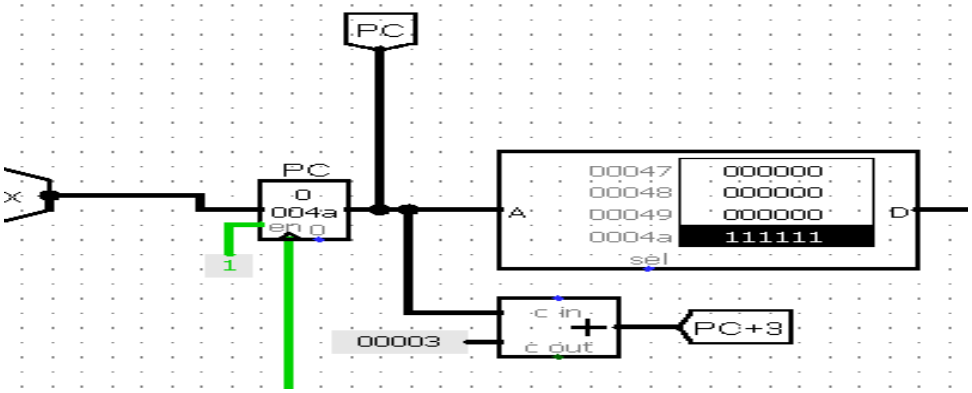


Figure 56: JAL.

The initial pc is 46 (01000110) or 70 and when we add 3 the result is 01001001, or 73 in decimal, which is then stored in R15. Also, the pc increments by 4 (2 shifted to the left).

21) XOR: R-Type instruction that performs the XOR operation between the values in the second and third register and stores the output in the first register.

XOR R5, R1, R2.

Machine Code: 00000011 0101 0001 00100000 = 0x035120.

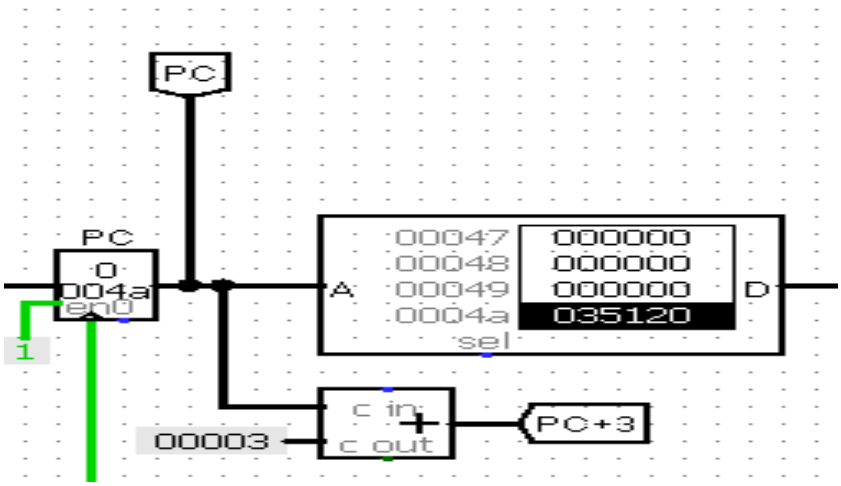


Figure 57: XOR.

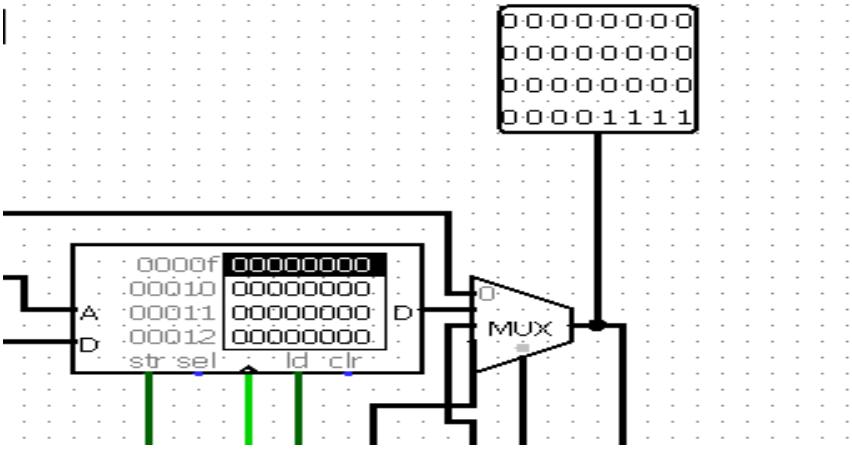


Figure 58: XOR.

Since R1 is 5 in decimal [0101 in 4-bit binary] and R2 is 10 in decimal [1010 in 4-bit binary] the output of the XOR between them is [1111 in 4-bit binary] 15 in decimal which is what the output is showing and what is being stored in R5.

22) SW: [SW Rz, Ry, Imm8] I-Type instruction which stores the value in the register Rz into the memory address [Ry + imm8] .

SW R1, R0, 4.

Machine Code: 00011001 0001 0000 00000100 = 0x191004.

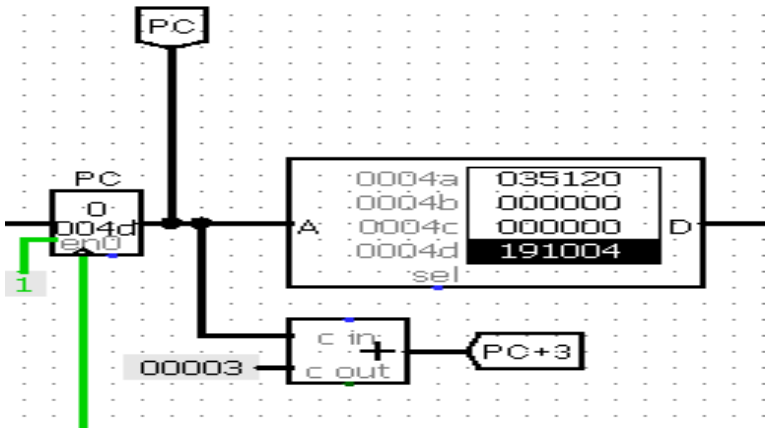


Figure 59: SW.

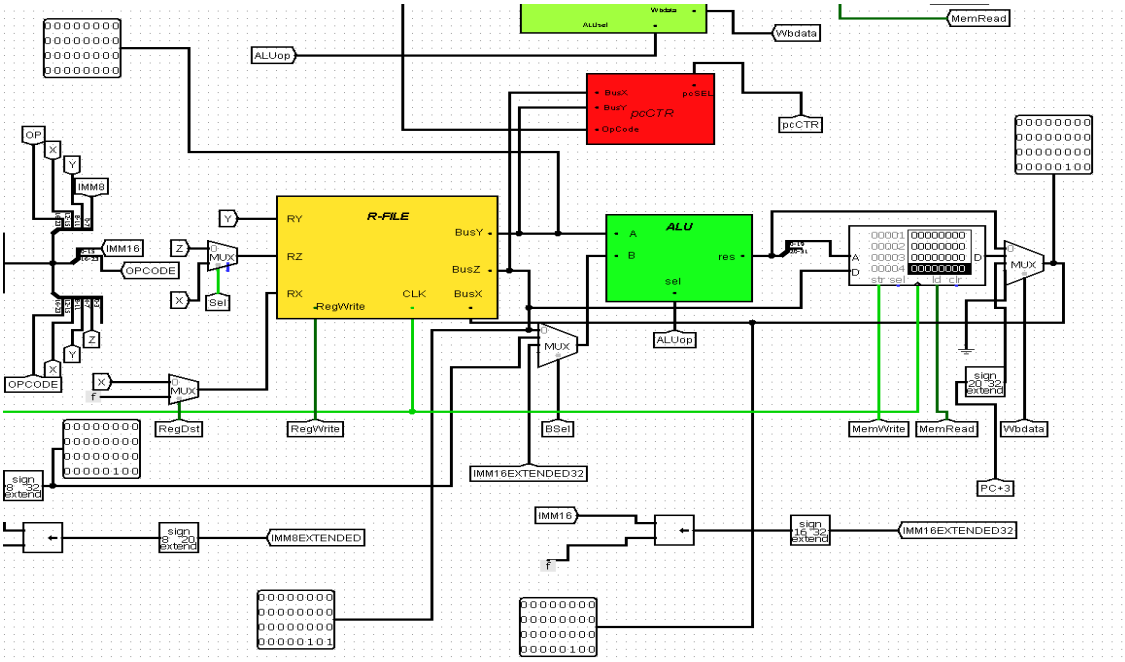


Figure 60: SW.

As we can see from the figure above the Imm8 = 4 and the R0 = 0 so the expected memory address to store at is 4 which is what our memory is pointing at (and will be stored at the next clock cycle as we will show in the next instruction). The value stored in the memory is the value in R1 which is 5 in decimal [0101 in 4-bit binary]. In the next instruction we will load the value at memory address 4 into a register.

23) LW: [LW, Rx, Ry, Imm8] I Type instruction which loades the data stored at memory address [Ry + Imm8] into a register Rx.

LW R6, R0, 4.

Machine Code: 00011000 0110 0000 00000100 = 0x186004.

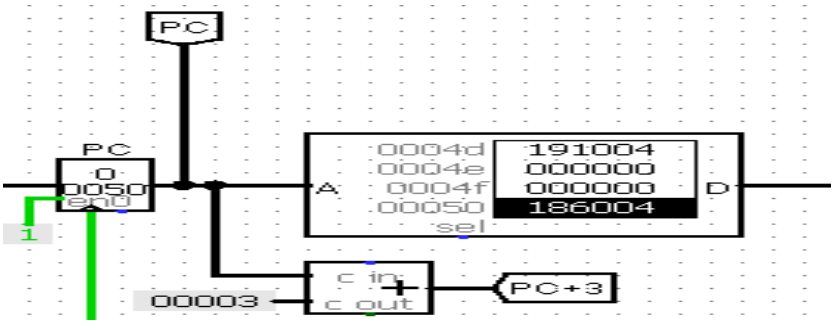


Figure 61: LW.

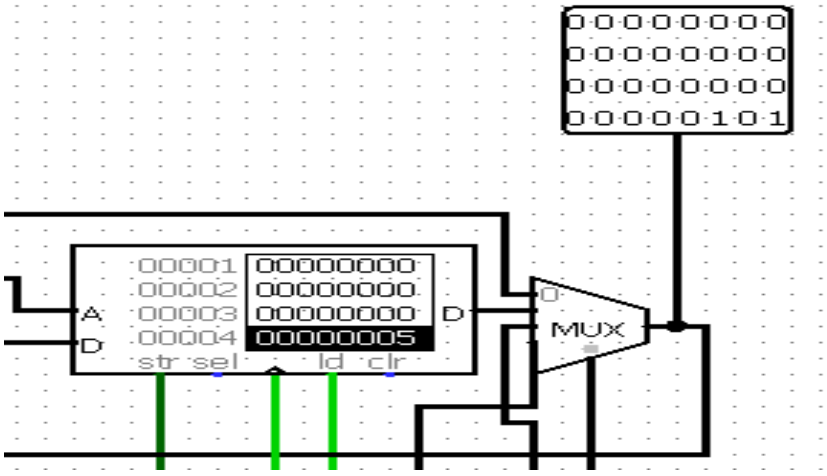


Figure 62: LW.

From the instruction we expect R6 to be loaded from the value in the memory address [R0 + 4] which is 4 (since R0 = 0) which we stored the value 5 in in the previous instruction.

As we can see from the figure above the Memory at address [4] holds the value 5 in decimal which is loaded into the register as expected.

24) BNE: I-Type instruction that takes two registers if their data is NOT equal. It increments the PC by the immediate data shifted to the left by one bit, meanwhile, if they are, it continues normally (PC +3) without changing the value of any registers.

BNE R4, R5, 2

Machine Code: 00010101 0100 0101 00000010 = 0x154502

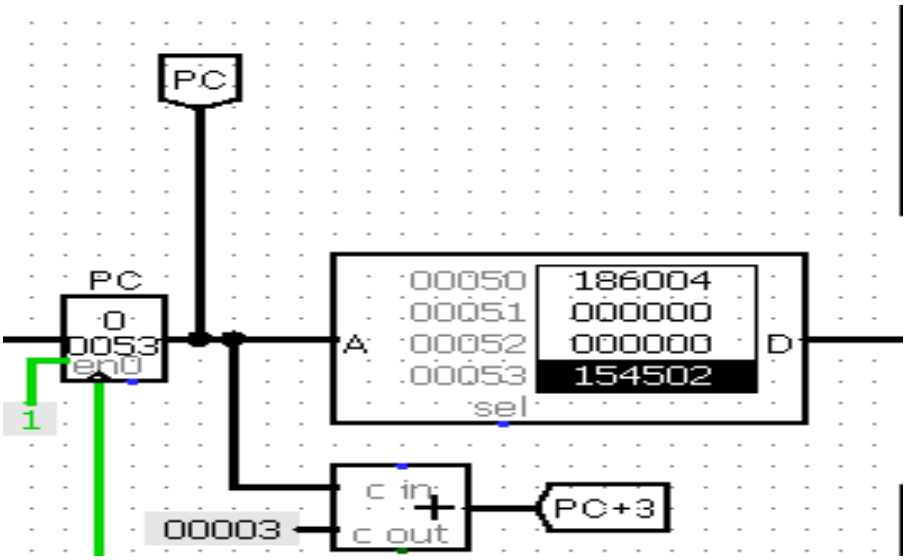


Figure 63: BNE.

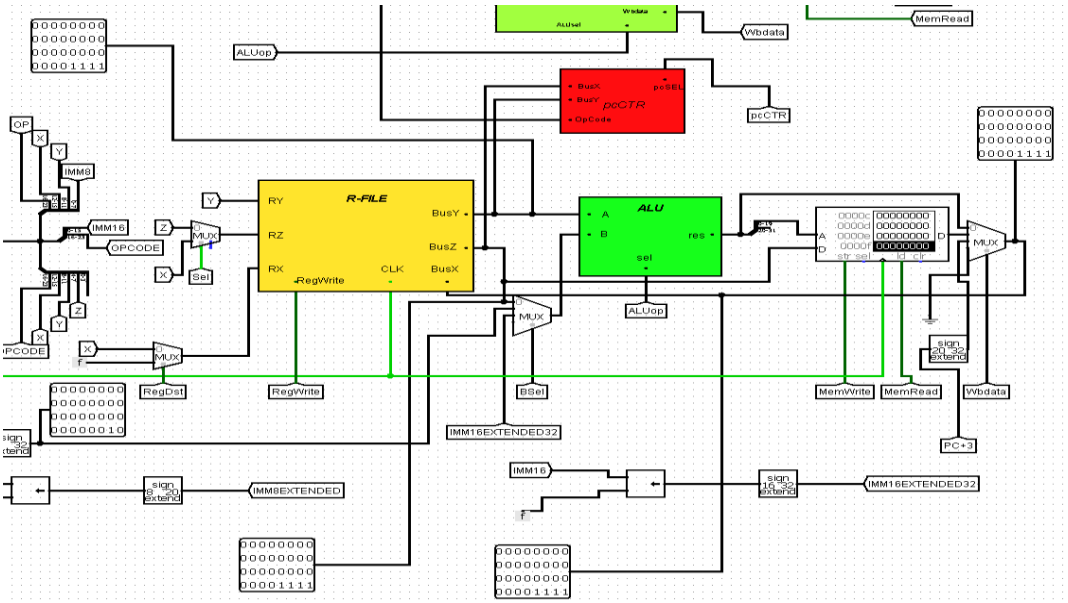


Figure 64: BNE.

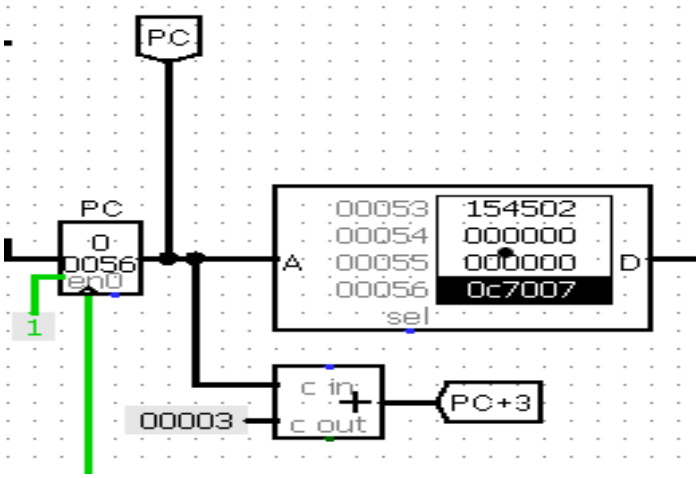


Figure 65: BNE.

As we can see from the figures above when fetching the BNE instruction PC was at 52(hex) and as we can see both the registers hold the value 15 (decimal) [1111 in 4-bit binary] so they are equal therefore PC will not branch and will increment normally [PC + 4 (Imm8 = 2 shifted by 1 => 2 * 2 = 4)]. We can see from the third figure PC is at 56(hex); [PC incremented by 3] which is exactly as expected.

25) ADDI R7, R0, 7

Machine Code: 00001100 0111 0000 00000111 = 0x0C7007.

Loading R7 with 7.

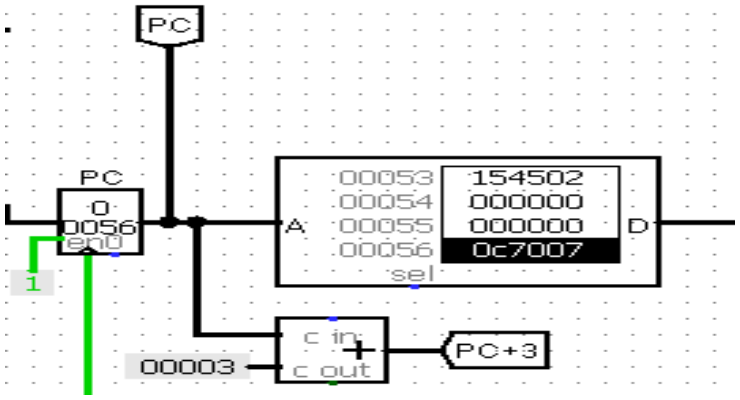


Figure 66: ADDI.

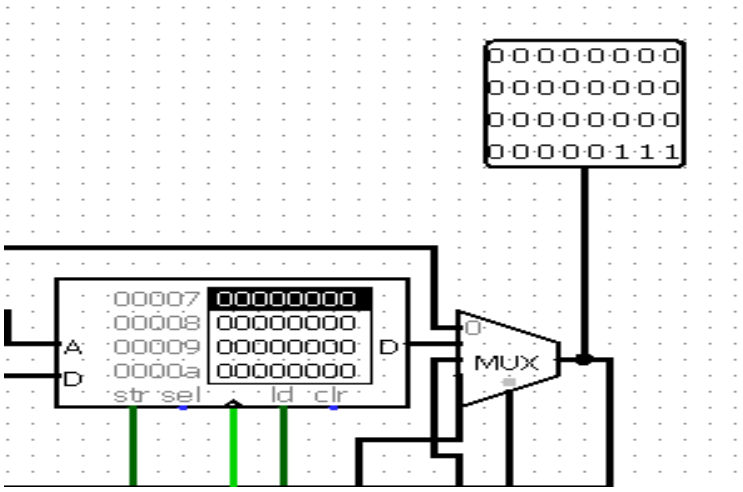


Figure 67: ADDI.

26) BNE: BNE R1, R3, 2

Machine Code: 00010101 0001 0011 00000010 = 0x151302.

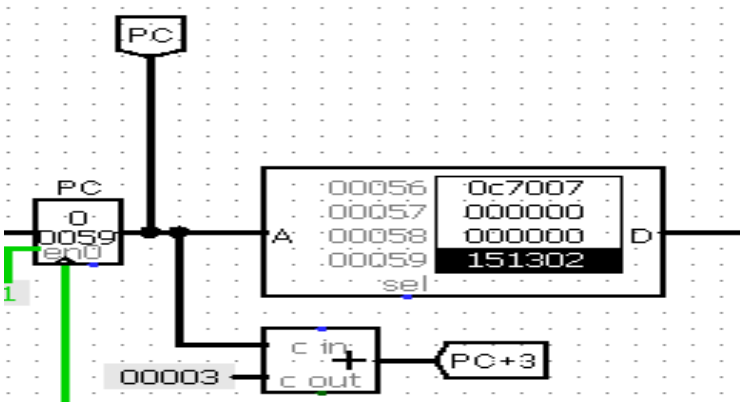


Figure 68: BNE.

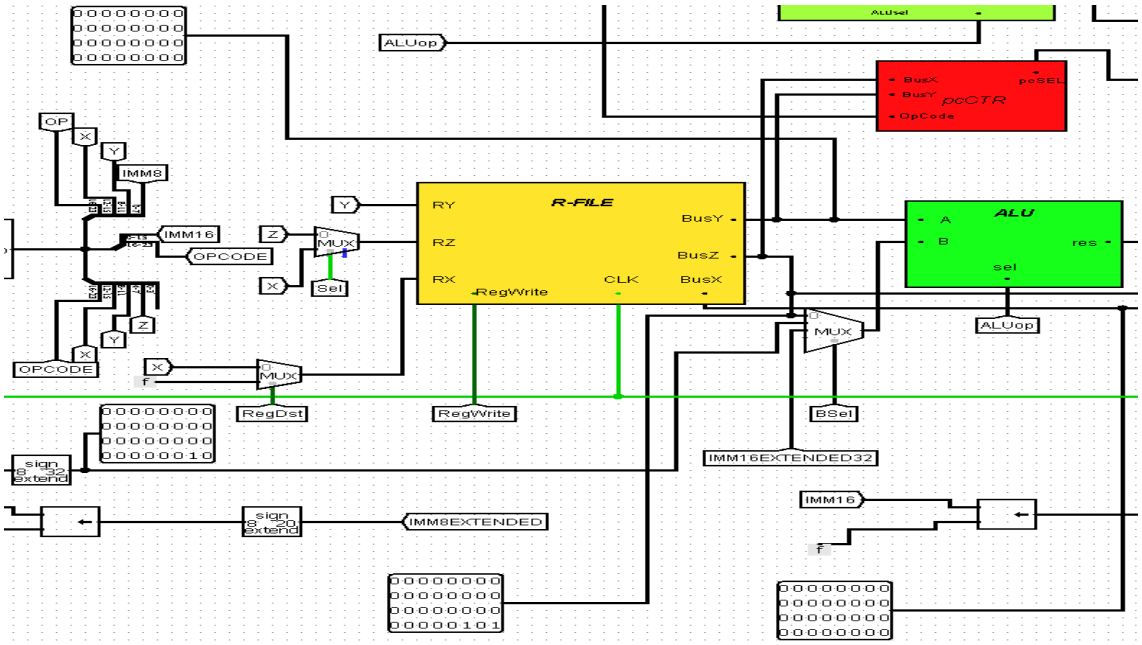


Figure 69: BNE.

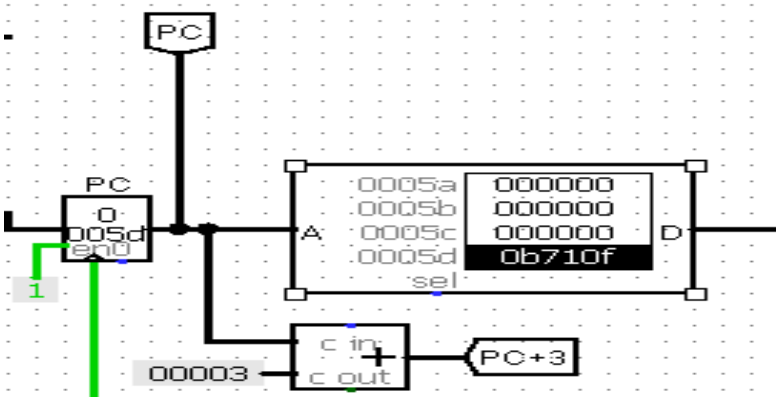


Figure 70: BNE.

From the instruction we expect the PC to increment by 4 [Imm8 = 2 shifted left by 1 => 2*2 = 4 => PC = PC + 4] since R1 and R3 are not equal satisfying the condition. When the BNE instruction was fetched PC was = 59 (hex) and as we can see the registers hold the values 0 and 5 which are not equal and after the instruction PC is 5d (hex d=13) [13 – 9 = 4] which means that the PC branched as expected.

27) XORI: [XORI Rx, Ry, Imm8] I-Type instruction which performs the XOR operation on the value in Ry and Imm8 (extended) and stores the output in Rx. XORI R7, R1, 15.

Machine Code: 00001011 0111 0001 00001111 = 0x0b710f.

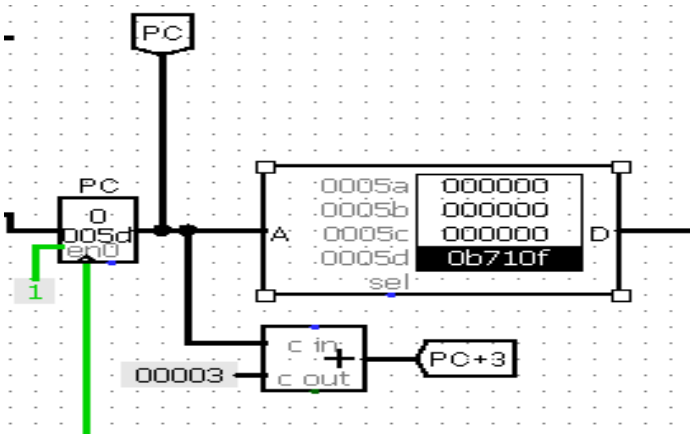


Figure 71: XORI.

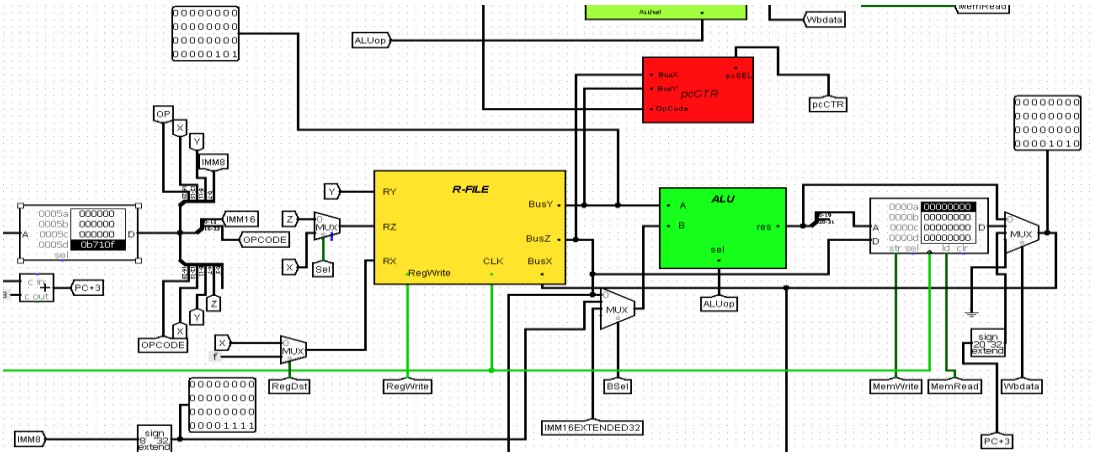


Figure 72: XORI.

As we can see from the figure above the value in the register is 5 [0101 in 4-bit binary] and the Imm8 (extended) = 15 [1111 in 4-bit binary] so the output after performing the XOR is [1010 in 4-bit binary] which the value saved into the R7.

28) BGE: I-Type instruction that takes two registers if their data is GREATER than or EQUAL. It increments the PC by the immediate data shifted to the left by one bit, meanwhile, if they are NOT, it continues normally (PC +3) without changing the value of any registers. BGE R1, R3, 2.
Machine Code: 00010111 0001 0011 00000010 = 0x171302.

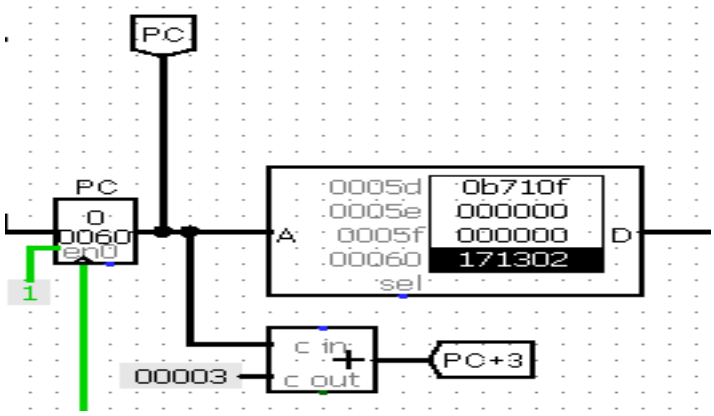


Figure 73: BGE.

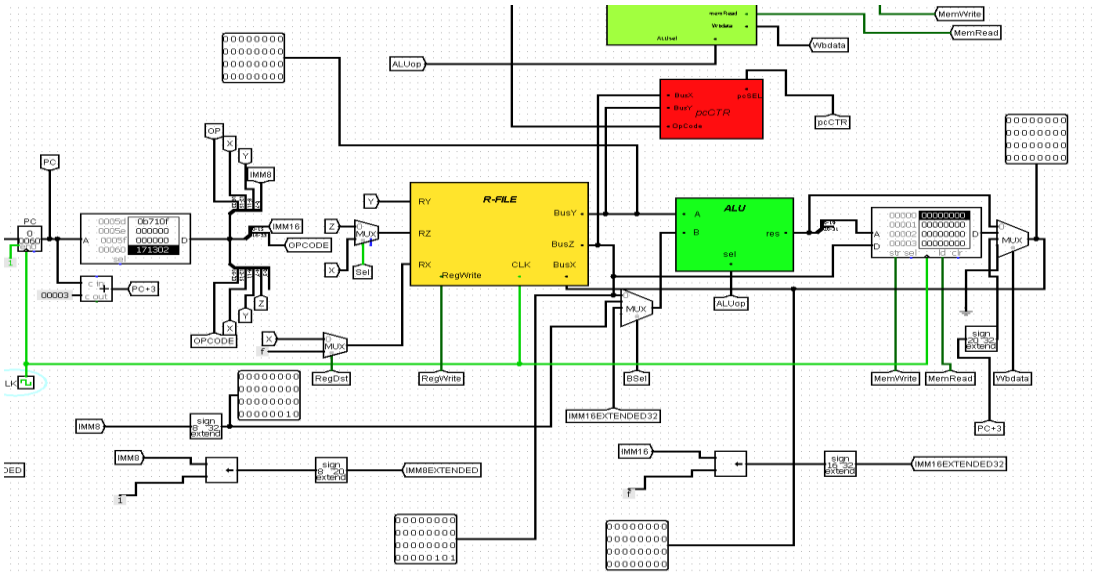


Figure 74: BGE.

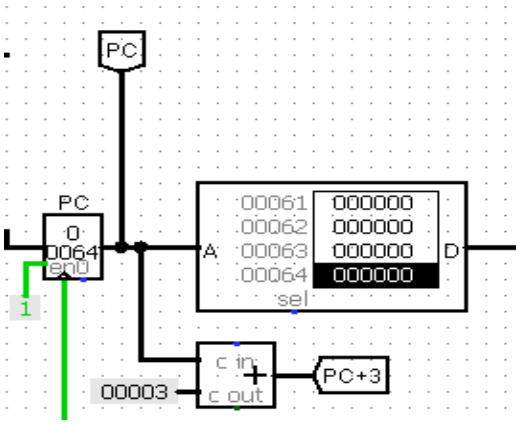


Figure 75: BGE.

As we can see from the figures above when the instruction was fetched PC was = 60 (hex) and the registers hold the values 10 ,0 respectively (10 > 0) so we expect the branching to happened which is exactly what happened as we can see in the third figure so it becomes (60 + Imm8 (2) shifted left by 1 => 4) 64 (hex).

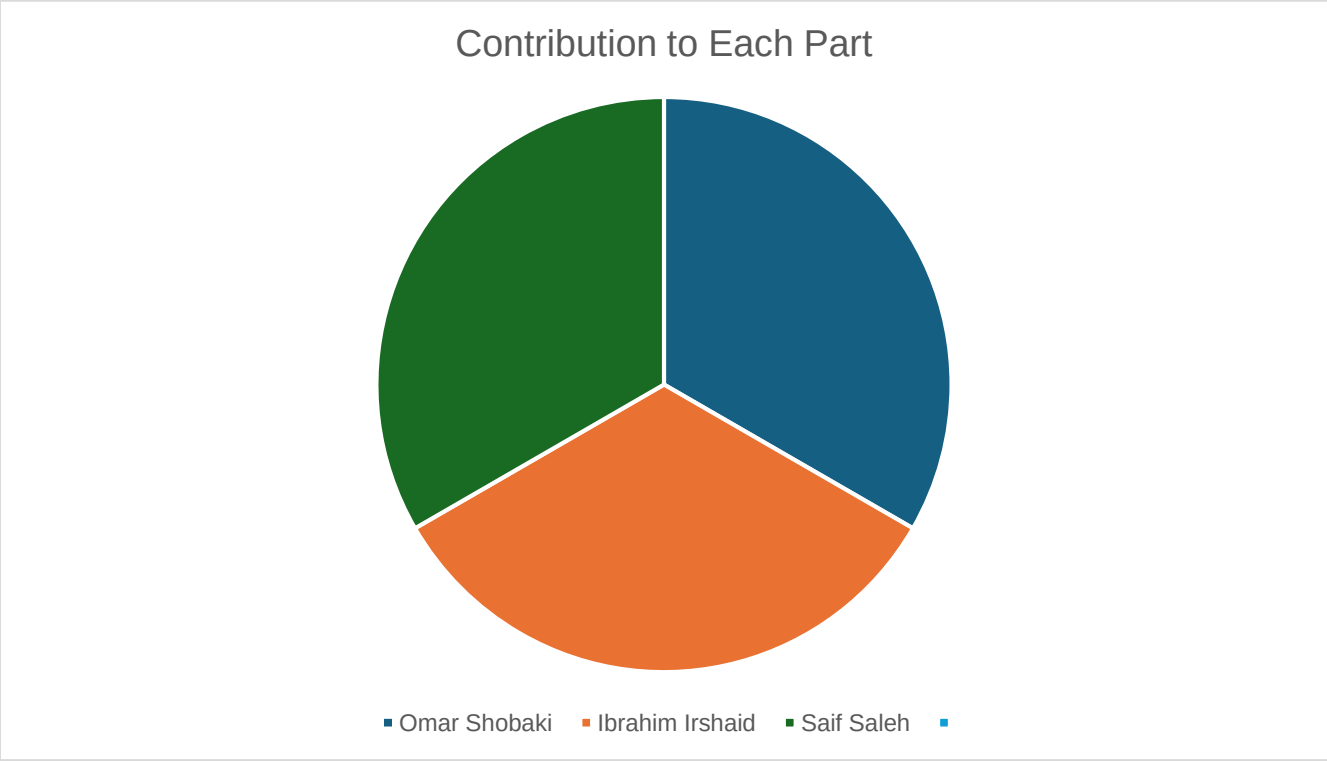
Conclusion

In this project, we successfully designed and implemented a 32-bit single-cycle processor using Logisim based on a 24-bit custom instruction set architecture. We successfully built and integrated essential processor components including the register file, ALU, instruction and data memory, control units, and the overall datapath. The processor could execute a wide range of instructions such as arithmetic, logic, memory access, and control flow operations (branching and jumping), covering all three instruction types: R-type, I-type, and J-type.

We verified the correctness of our design by developing and running multiple test programs that exercised all instruction functionalities. This thorough testing helped us validate not only individual components but also the coordination between the datapath and control units. The results confirmed that our processor correctly handled instruction execution, data movement, branching decisions, and program flow.

In addition to the technical implementation, this project strengthened our understanding of how processors operate at the micro-architectural level, and how control signals and data paths interact to achieve correct instruction behavior. It also emphasized the importance of modular design, debugging skills, and teamwork in successfully managing and completing a hardware-based project. Overall, this project was a valuable learning experience that bridged theoretical knowledge with practical application in computer architecture.

Team Work


Contribution to Each Part
Omar Shobaki
Ibrahim Irshaid
Saif Saleh

Appendices

- Appendix(1)

RegSel: It is used to switch between the input of Rz in the register file so it is the value X [bits 12 – 15] only for the SW instruction and the branch instructions (when 1 use x otherwise use z)

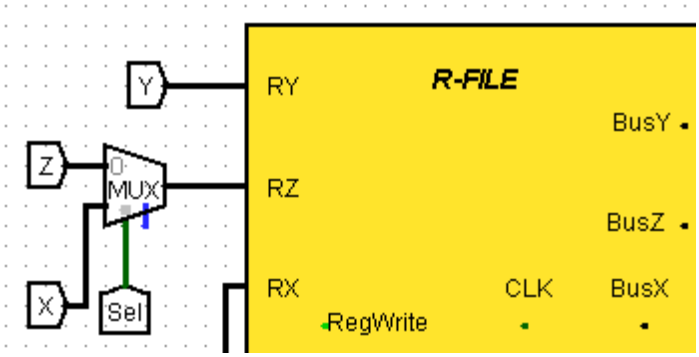


Figure: 76 RegSel

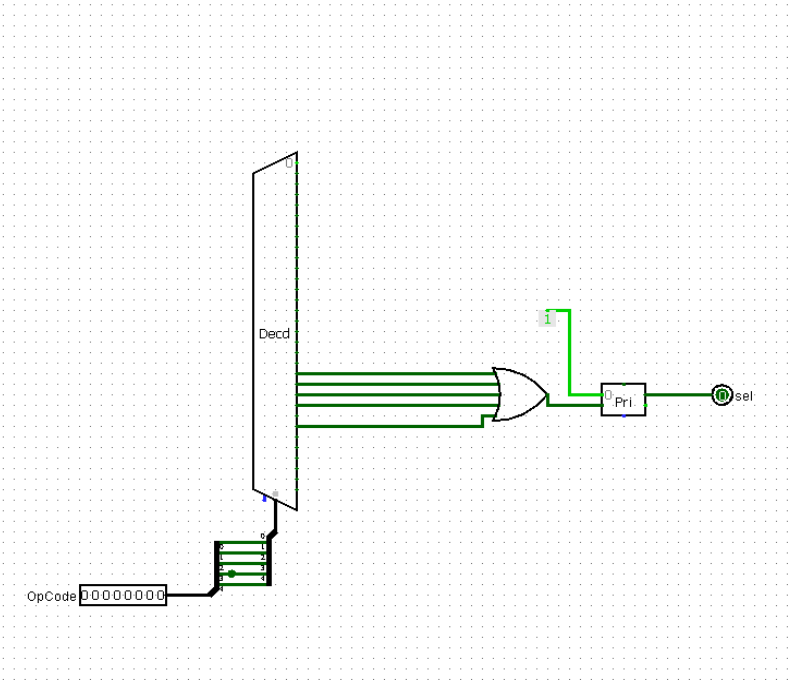


Figure 77: RegSel Implementation

Decoding the OpCode when the op code is 25 SW and 20 – 23 for the branches instruction.

The reason it is used for the branches is to make the Register file take the number of the register from X [bits 12 – 15] of the instruction so BusZ holds the value at register number X then takes into the PC controller along with Ry for comparison later in the PC controller.