Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Computer Organization and Microprocessor

ENCS2380

Single Cycle Processor Design

Prepared by:

Name: Shahd Manasra **Number:** 1230308

Name: Sojood Asfour **Number:** 1230298

Name: Shahd Tamimi **Number:** 1231204

Section: (1)

Date: 22/12/2024

Table of Contents

Table of Figures

List of Table

Design and Implementation

Register File

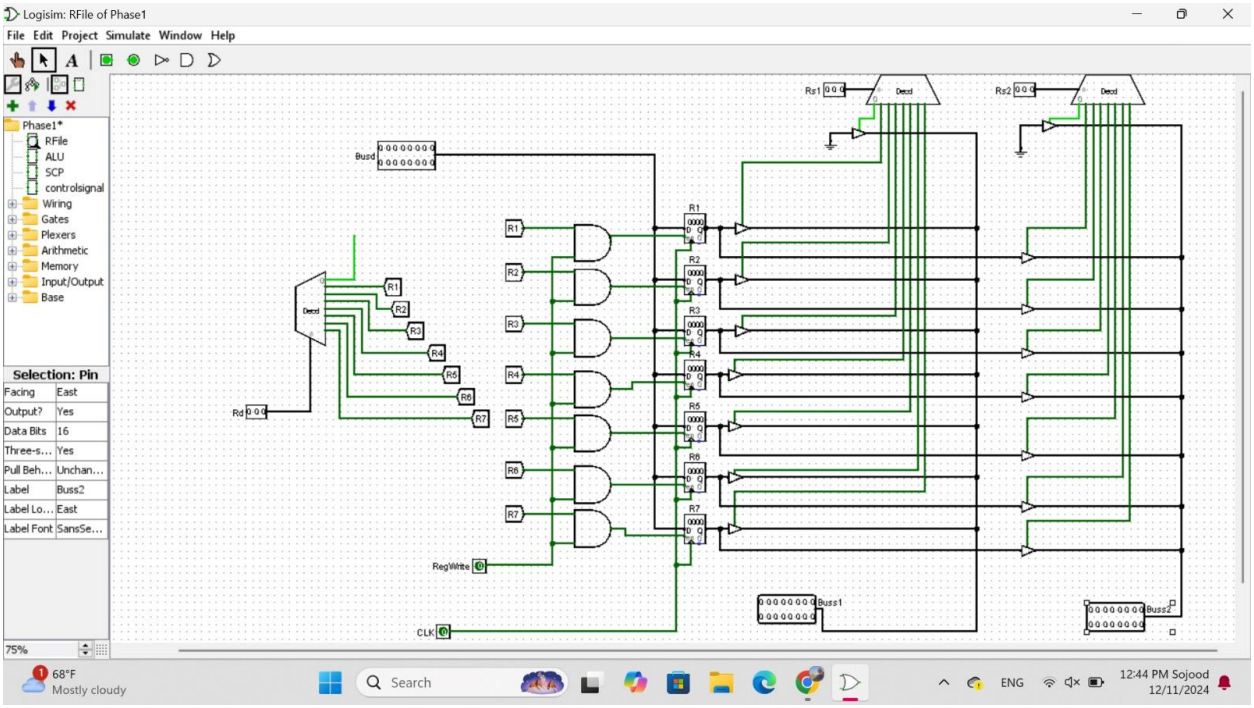


Figure 1: Register File.

The register file that is shown in figure 1 is an important component of a processor for enabling temporary storage and efficient movement of data through various processor components. It contains eight registers of 16 bits each (R0 to R7), R0 is a special register since it is hardwired to zero and can be read, not written. It accommodates two read ports and one write port for parallel data access.

Register File Components:

Here's a more detailed explanation of its functionality:

A decoder is utilized to select one of the available registers (from R1 to R7) to receive the write data as indicated by the Rd input signal when the write operation is taking place. A control signal

called RegWrite will enable or disable the writing. On activation, the data present at input Busd will be written to that particular register on the positive edge of the clock signal (CLK). It should be noted that R0 is hardwired to zero and has no write capabilities.

For reading, each register connects to two distinct read buses, Buss1 and Buss2, through tri-state buffers. The signals Rs1 and Rs2 determine which register drives its stored value on Buss1 and Buss2, respectively. Only a register selected by one of those signals opens its tri-state buffers to let its data flow to the bus, all others stay in high-impedance. Thus, this effectively prevents bus contention and ensures proper operation.

Test Case:

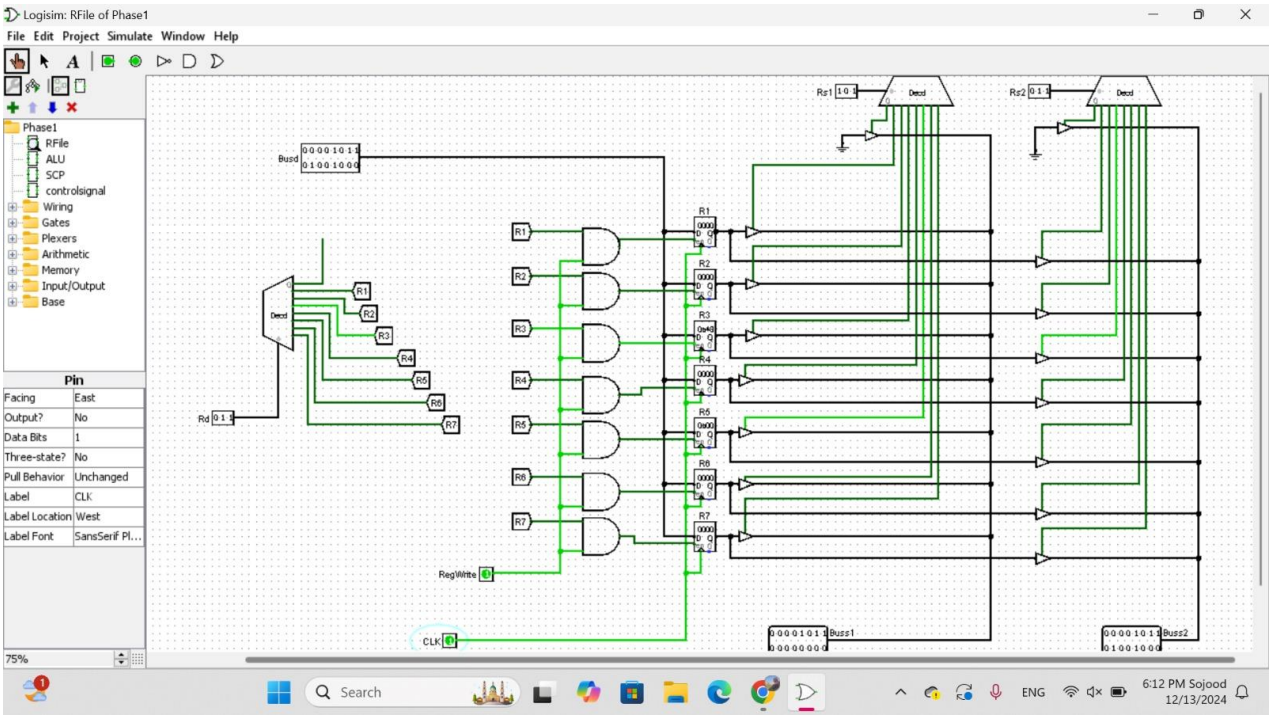


Figure 2: test R-File

In the figure (2) the Rd =101 in binary equal to 5 in decimal means that the fifth register is chosen and Buss1= 0000 1011 0000 0000 in binary equal to 2816 in decimal and when Rd =011 in binary

equal to 3 in decimal means that the third register is chosen and Buss2= 0000 1011 0100 1000 in binary equal to 2888 in decimal.

ALU

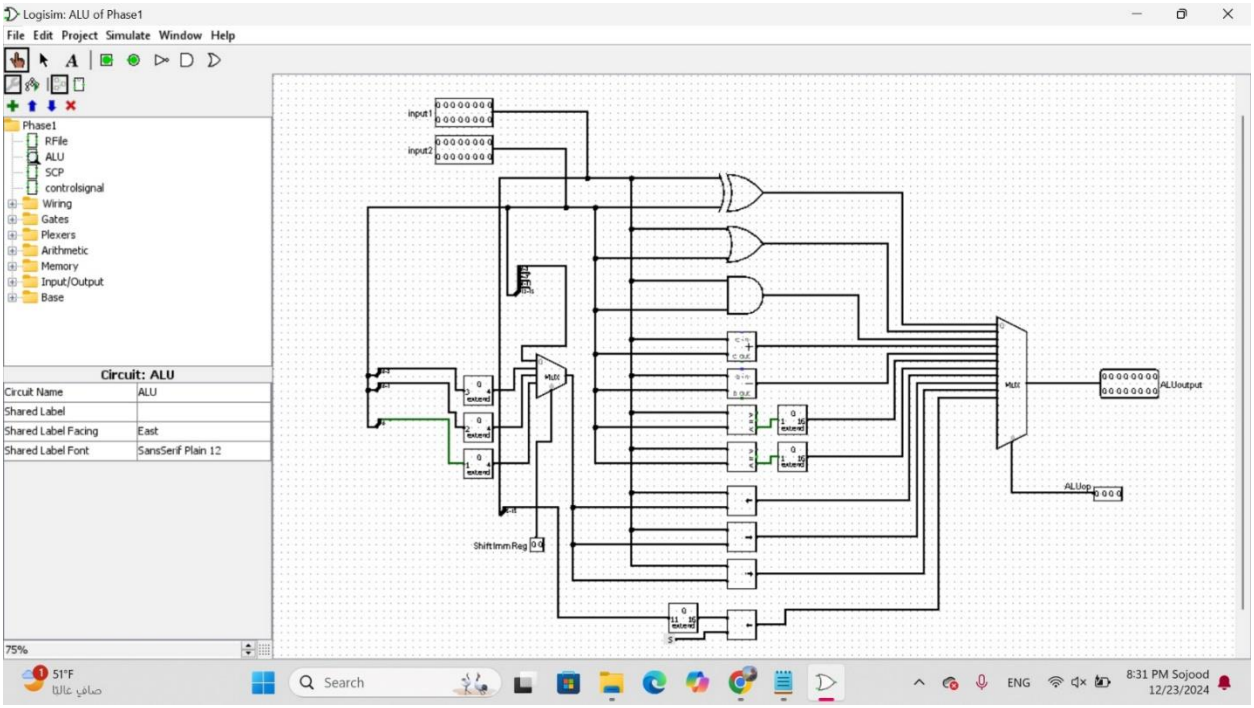


Figure 3: ALU.

This component can implement all the arithmetic and logical operations in this sequence:

XOR, OR, AND, ADD, SUB, SLT, SLTU, SLL, SRL, SRA, LUI.

The MUX selects one of the operation outputs based on the ALUop control signals, which act as the selector, the selected result from the MUX is forwarded to the ALUOutput line, which represents the result of the selected operation.

A binary control signal **(**ALUop**)** is used to specify which operation the ALU should perform. For example:

- 0000, XOR.
- 0001, OR.

- 0010, AND.
- 0011, ADD
- 0100, SUB.
- 0101, SLT (set less than).
- 0110, SLTU (set less than unsigned).
- 0111, SLL (shift left logical).
- 1000, SRL (shift right logical).
- 1001, SRA (shift right arithmetic).
- 1010, LUI (Load Upper Immediate).

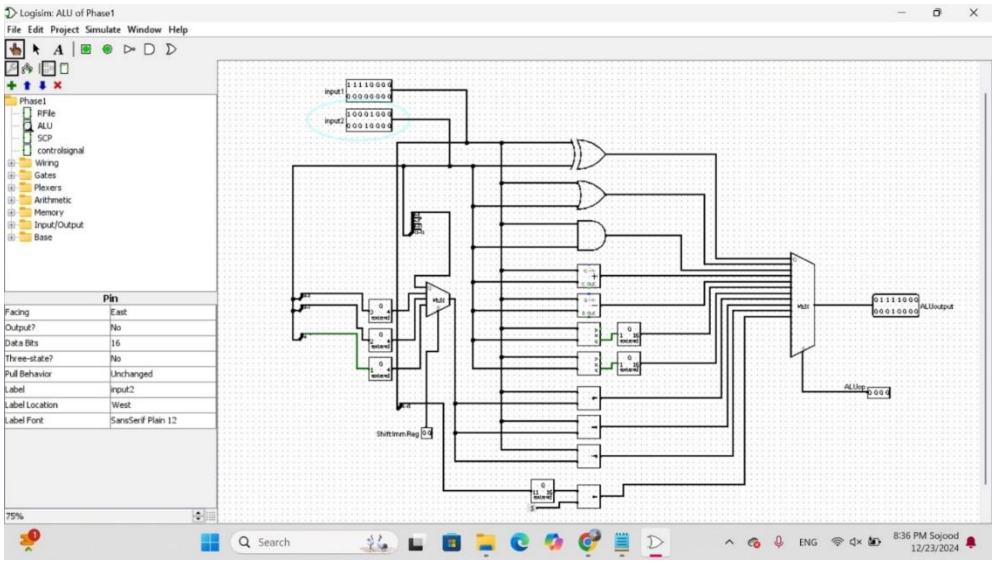


Figure 4: ALU test1.

In figure 4, we selected the operation code 0000 to perform the XOR operation. With the inputs set as (1111 0000 0000 0000) and (1000 1000 0001 0000) respectively, the ALU produced the output (0111 1000 0001 0000).

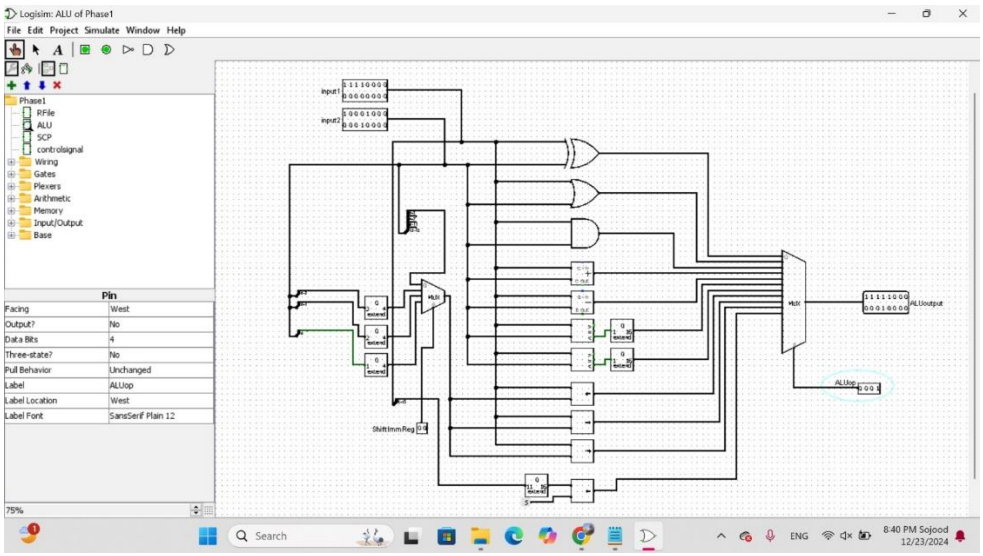


Figure 5 : ALU test 2.

In figure 5, we selected the operation code 0001 to perform the OR operation. With the inputs set as (1111 0000 0000 0000) and (1000 1000 0001 0000) respectively, the ALU produced the output (1111 1000 0001 0000).

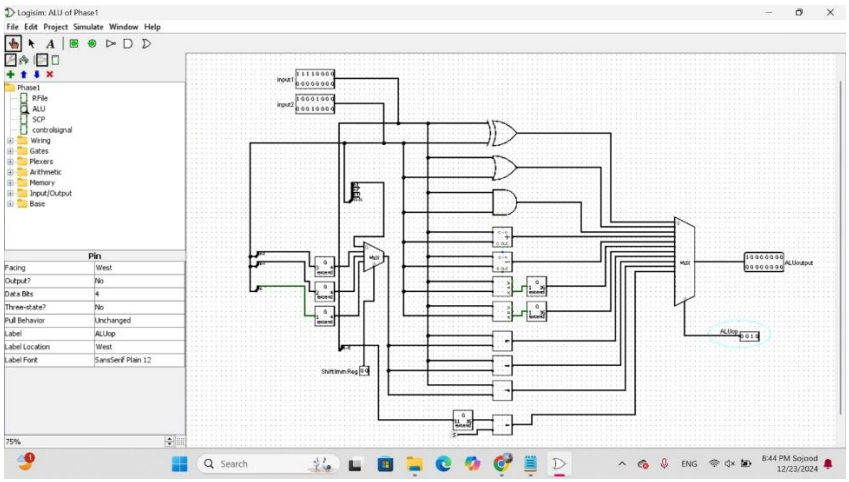


Figure 6 : ALU test 3.

In figure 6, we selected the operation code 0010 to perform the AND operation. With the inputs set as (1111 0000 0000 0000) and (1000 1000 0001 0000) respectively, the ALU produced the output (1000 0000 0000 0000).

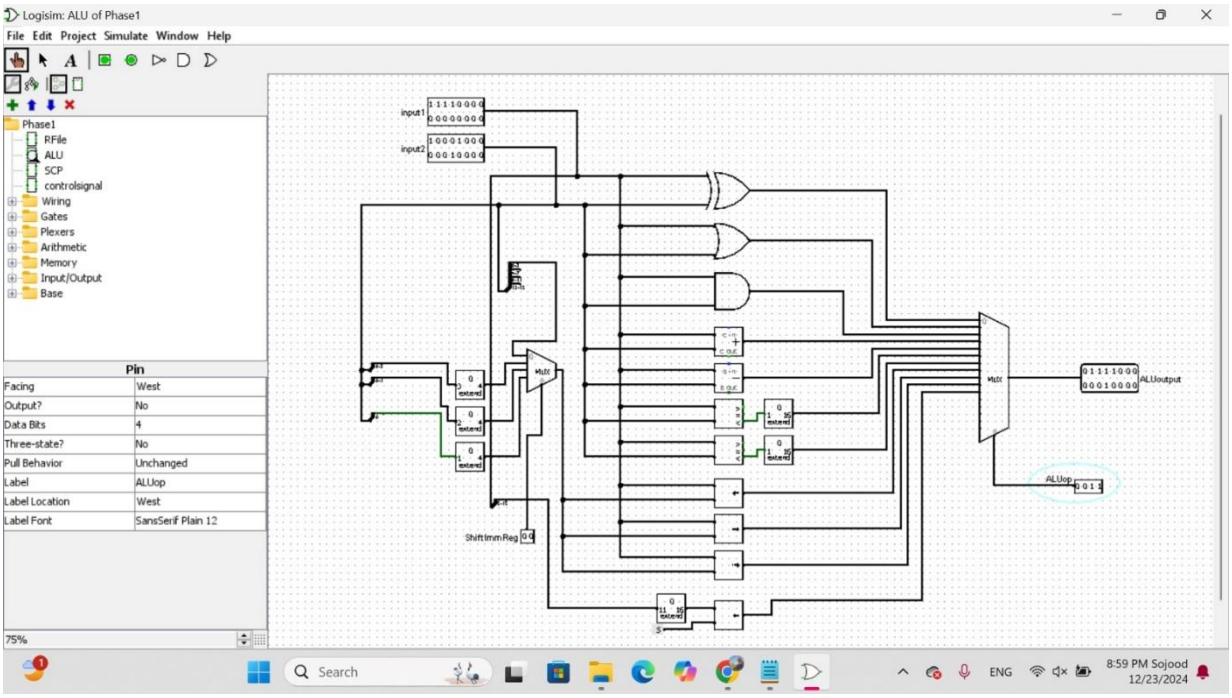


Figure 7: ALU test4.

In figure 7, we selected the operation code 0011 to perform the ADD operation. With the inputs set as (1111 0000 0000 0000) and (1000 1000 0001 0000) respectively, the ALU produced the output (0111 1000 0001 0000).

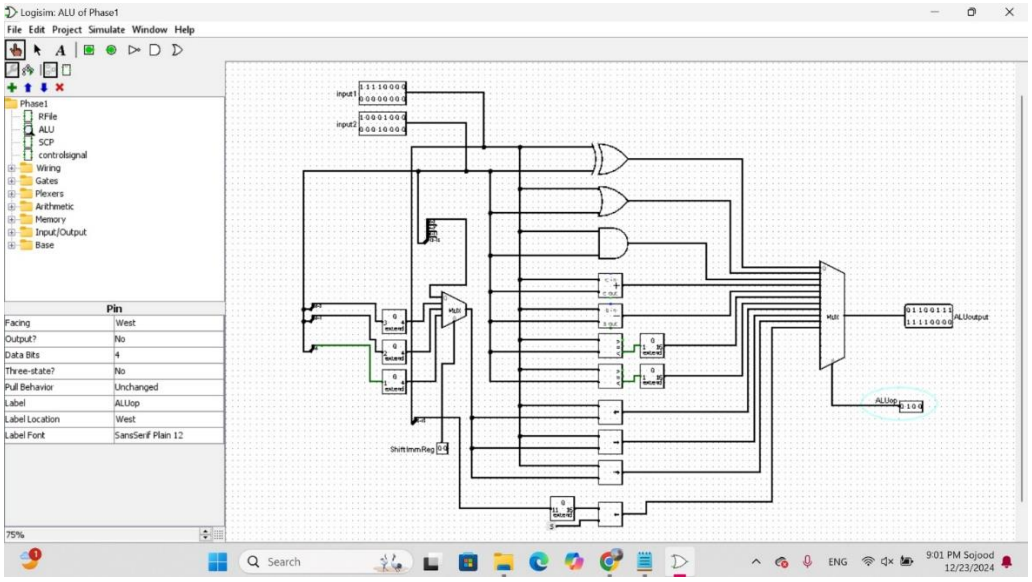


Figure 8: ALU test5.

In figure 8, we selected the operation code 0100 to perform the SUB operation. With the inputs set as (1111 0000 0000 0000) and (1000 1000 0001 0000) respectively, the ALU produced the output (0110 0111 1111 0000).

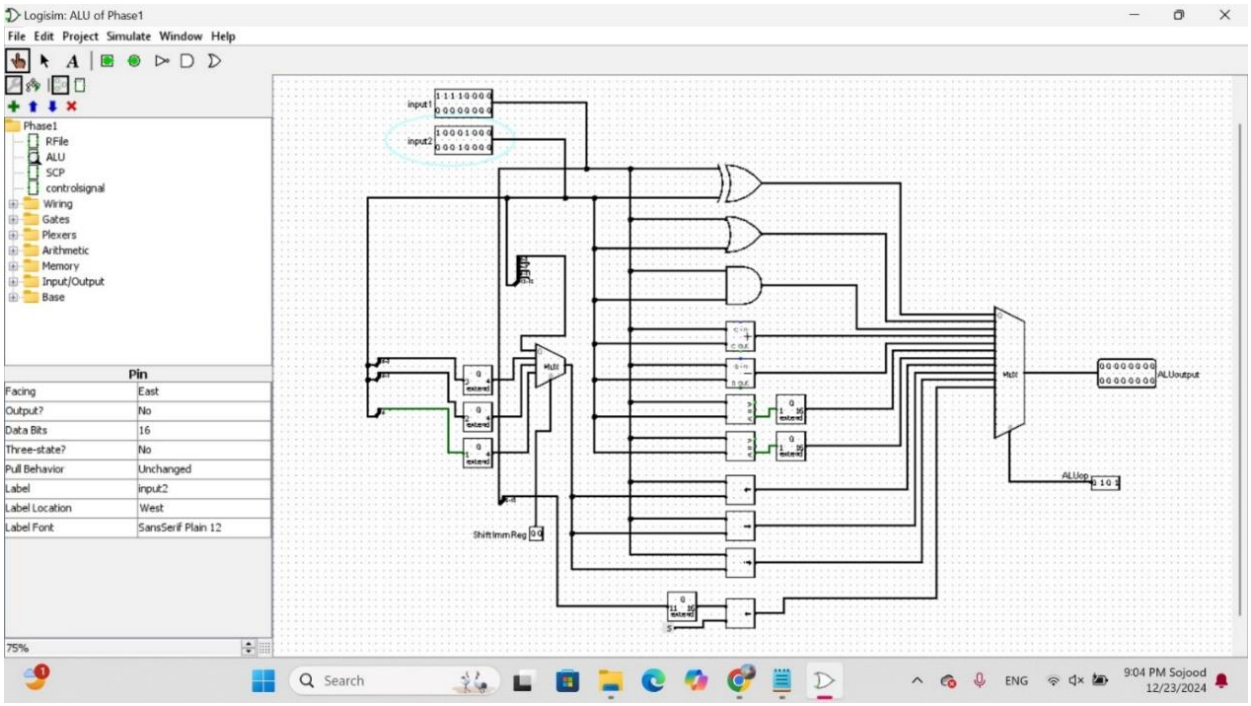


Figure 9: ALU test6.

In figure 9, we selected the operation code 0101 to perform the SLT operation. With the inputs set as (1111 0000 0000 0000) and (1000 1000 0001 0000) respectively, the ALU produced the output (0000 0000 0000 0000).

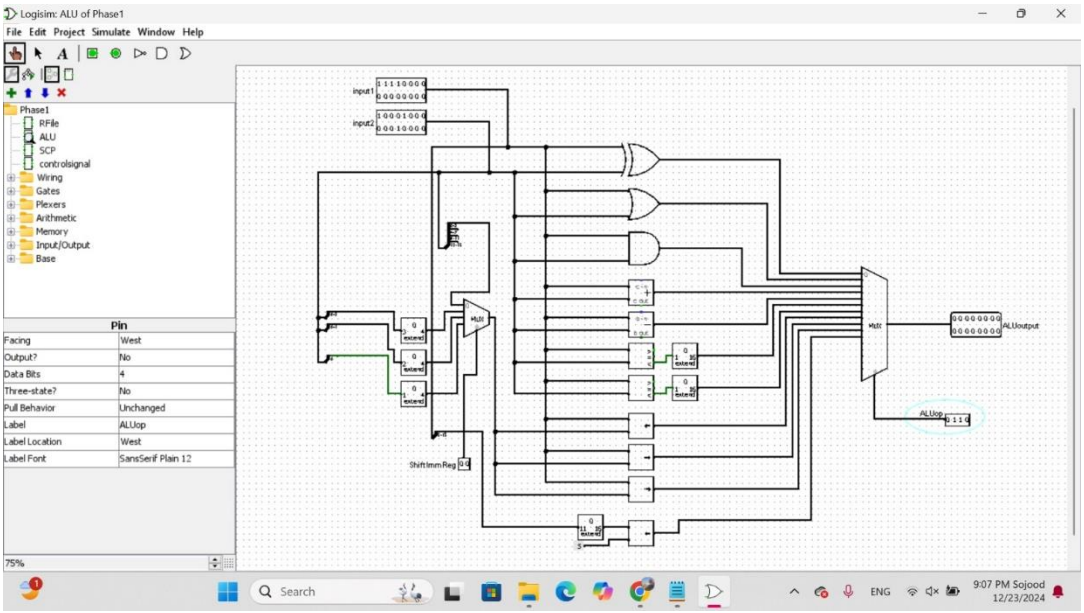


Figure 10: ALU test7.

In figure 10, we selected the operation code 0110 to perform the SLTU operation. With the inputs set as (1111 0000 0000 0000) and (1000 1000 0001 0000) respectively, the ALU produced the output (0000 0000 0000 0000).

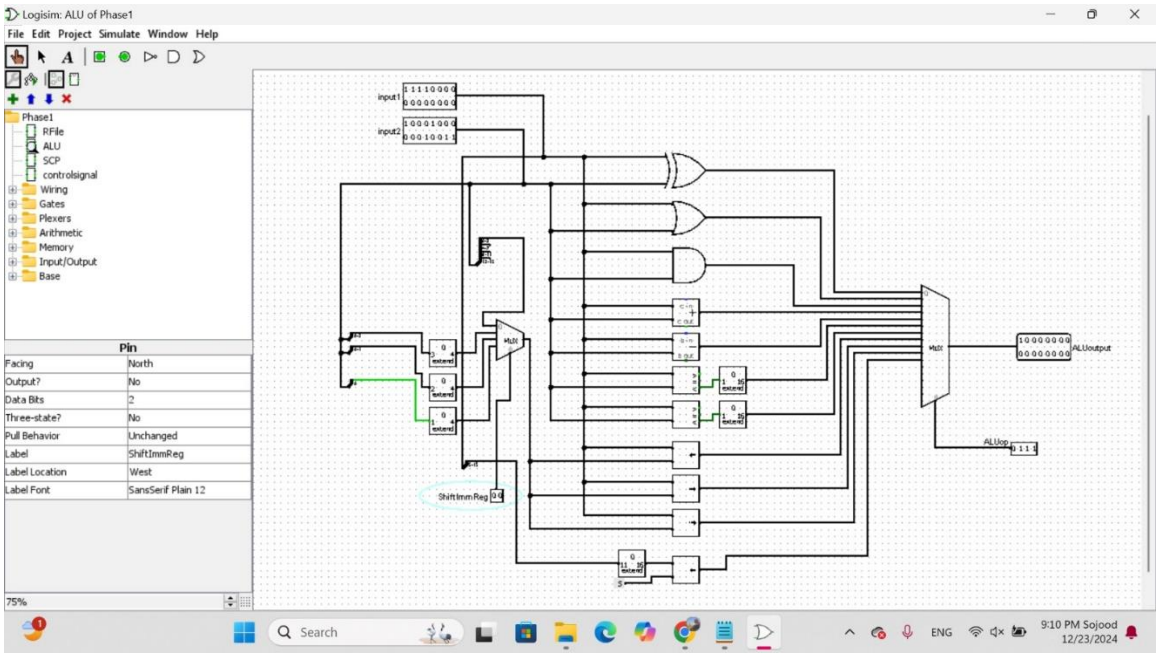


Figure 11: ALU test8.

In Figure 11, the operation code 0111 was selected to perform the SLL (Shift Left Logical) operation. The inputs were set as (1111 0000 0000 0000) and (1000 1000 0001 0011) respectively. The ALU will perform a left shift on the first input by the number of bits specified by the first four bits of the second input (Imm4). In this case, the first input is shifted by 3 bits, and the ALU produces the output (1000 0000 0000 0000). The Imm4 value was chosen because the control signal "shiftImmReg" was set to 00, indicating that the shift amount is taken from the Imm4 field.

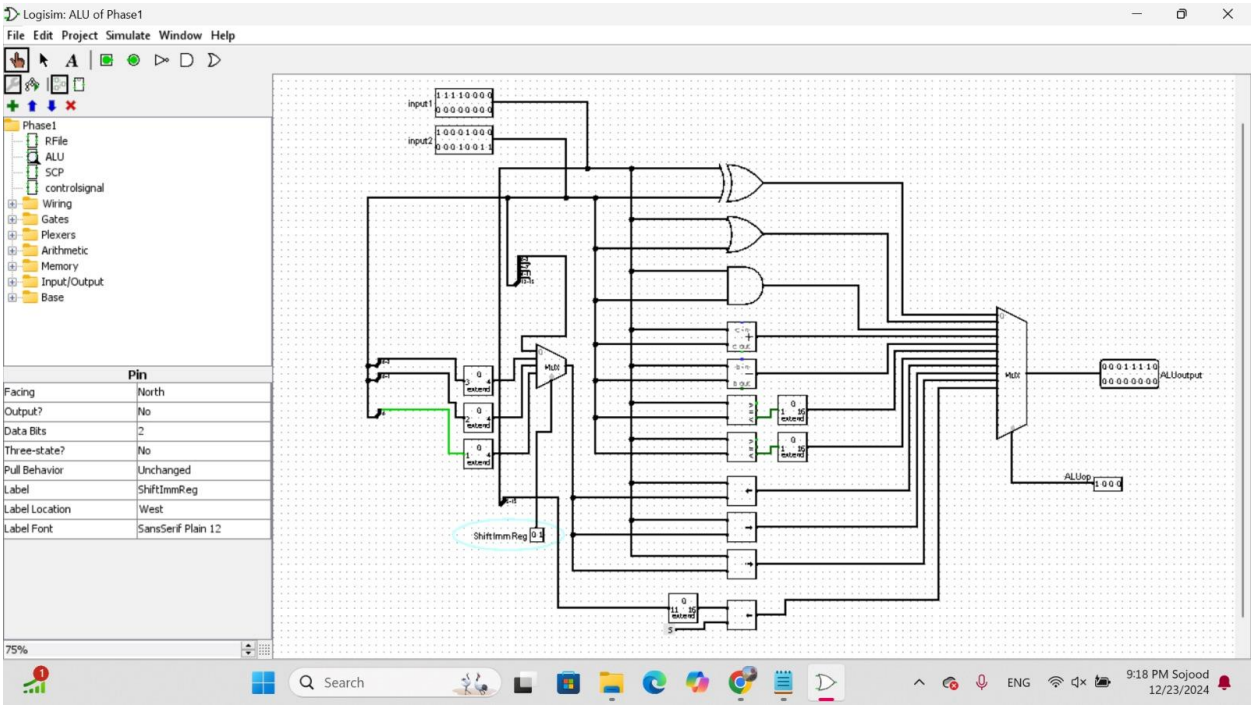


Figure 12: ALU test9.

In Figure 12, the operation code 1000 was selected to perform the SRL (Shift Right Logical) operation. The inputs were set as (1111 0000 0000 0000) and (1000 1000 0001 0011) respectively. The ALU will perform a right shift on the first input by the number of bits specified by the first three bits of the second input (Imm3). In this case, the first input is shifted by 3 bits, and the ALU produces the output (0001 1110 0000 0000). The Imm3 value was chosen because the control signal "shiftImmReg" was set to 01, indicating that the shift amount is taken from the Imm3 field.

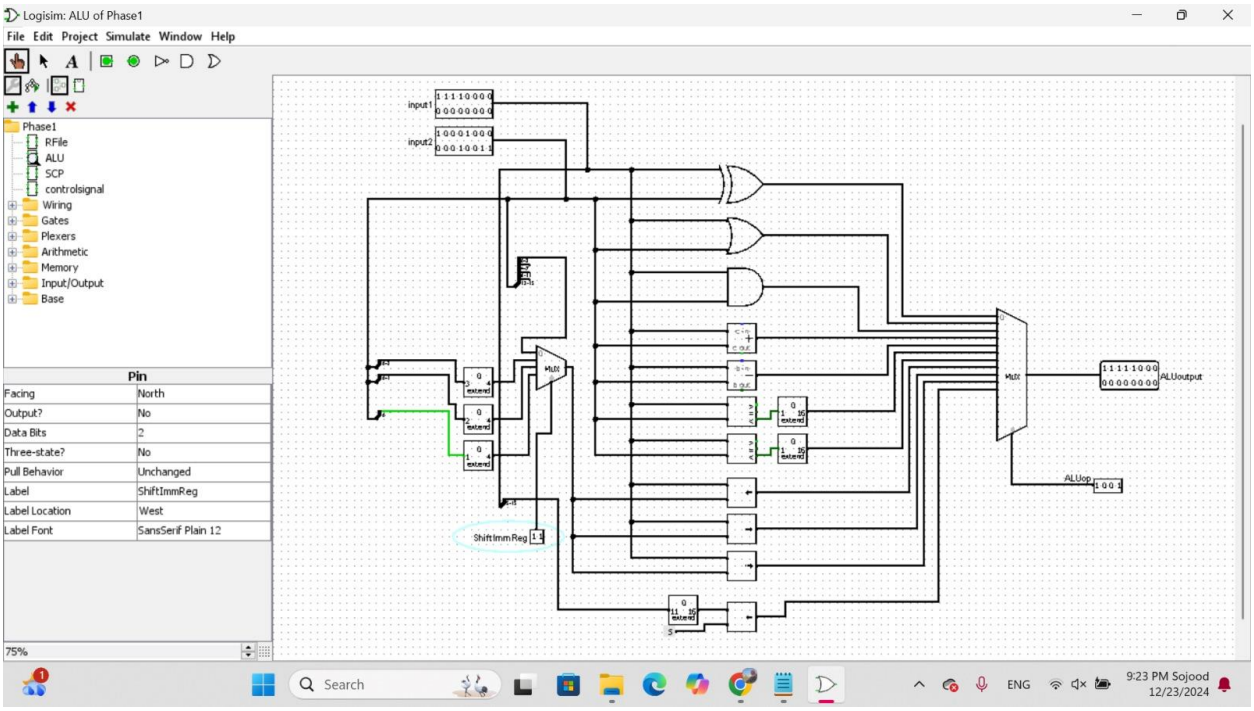


Figure 13: ALU test10.

In Figure 13, the operation code 1001 was selected to perform the SRA (Shift Right Arithmetic) operation. The inputs were set as (1111 0000 0000 0000) and (1000 1000 0001 0011) respectively. The ALU will perform a right shift on the first input by the number of bits specified by the first one bit of the second input (Imm1). In this case, the first input is shifted by 1 bit, and the ALU produces the output (1111 1000 0000 0000). The Imm1 value was chosen because the control signal "ShiftImmReg" was set to 11, indicating that the shift amount is taken from the Imm1 field.

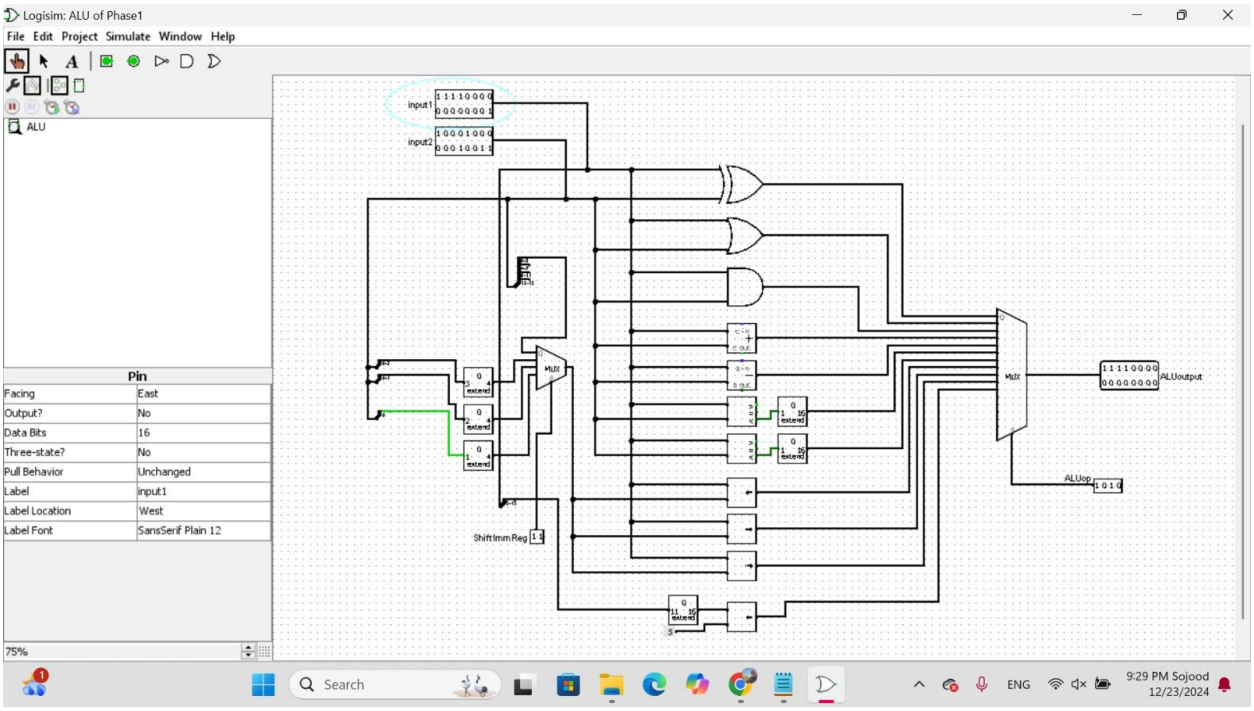


Figure 14: ALU test11.

In figure 14, We selected the operation code 1010 to perform the LUI operation, LUI places the imm11 value in the upper 11 bits of the destination register R1(input 1), filling the lowest 5 bits with zeros. With the inputs set as (1111 0000 0000 0001) and (1000 1000 0001 0000) respectively, the ALU produced the output (1111 0000 0000 0000).

Data Path

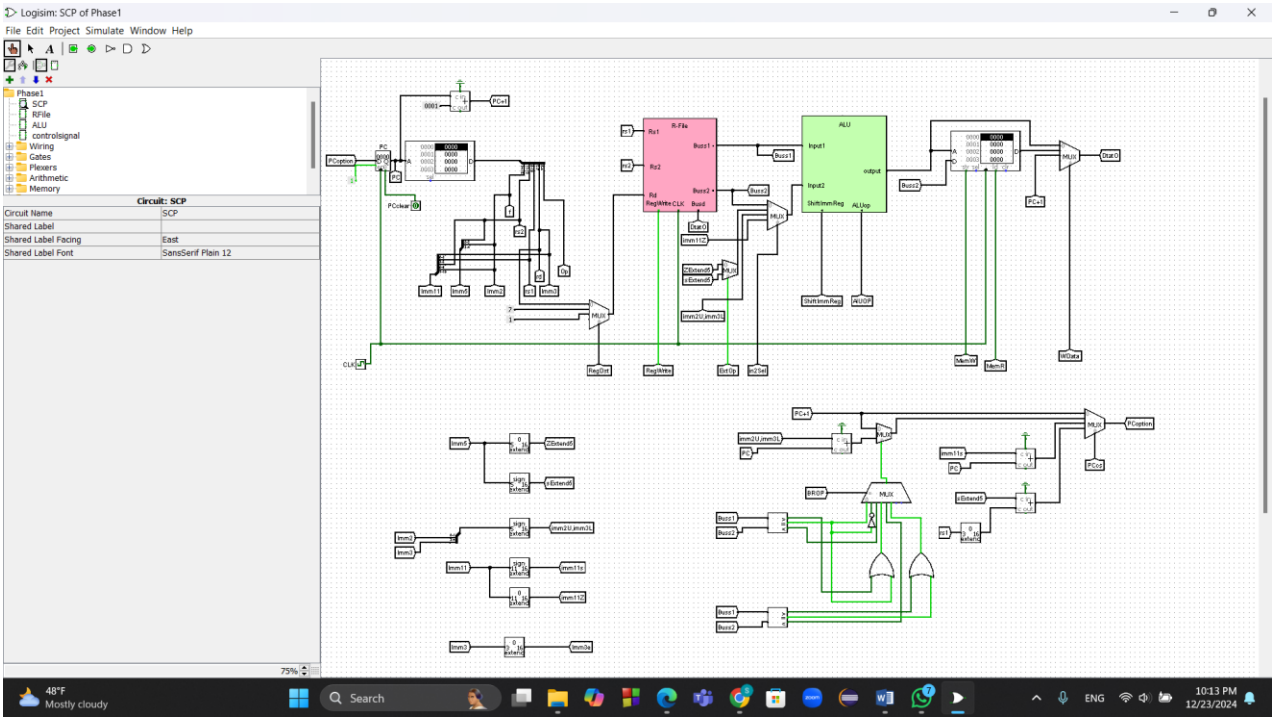


Figure 15: DataPath.

In this part, we connected the components together to construct the single cycle processor, so we connect the instructions memory (which contain the instructions to be executed in the even addresses), the Registers File, ALU, and the Data memory (which contains the data). And obtain a special purpose register which is PC register that contain the address of the next instruction to be executed, PC register is 16-bits register, the instructions memory is 16-bits addresses bus, and each cell is 16-bits because each instruction is 16-bits. The PC registers value increment by 1 at every clock cycle [PC + 1], but when there is a branch (conditional) instruction if the condition is true the PC value changes as PC= PC+sign_extend({imm2U, imm3L}), otherwise the PC increment by 1 (e.g. PC = PC + 1), when there is a jump (J, JAL) instruction the PC changes as PC = PC+sign_extend(imm11), and there is a JALR (Jump-And-Link-Register) instruction the PC changes as PC= Ra+sign_extend(imm5).

DataPath Component

Instruction Memory

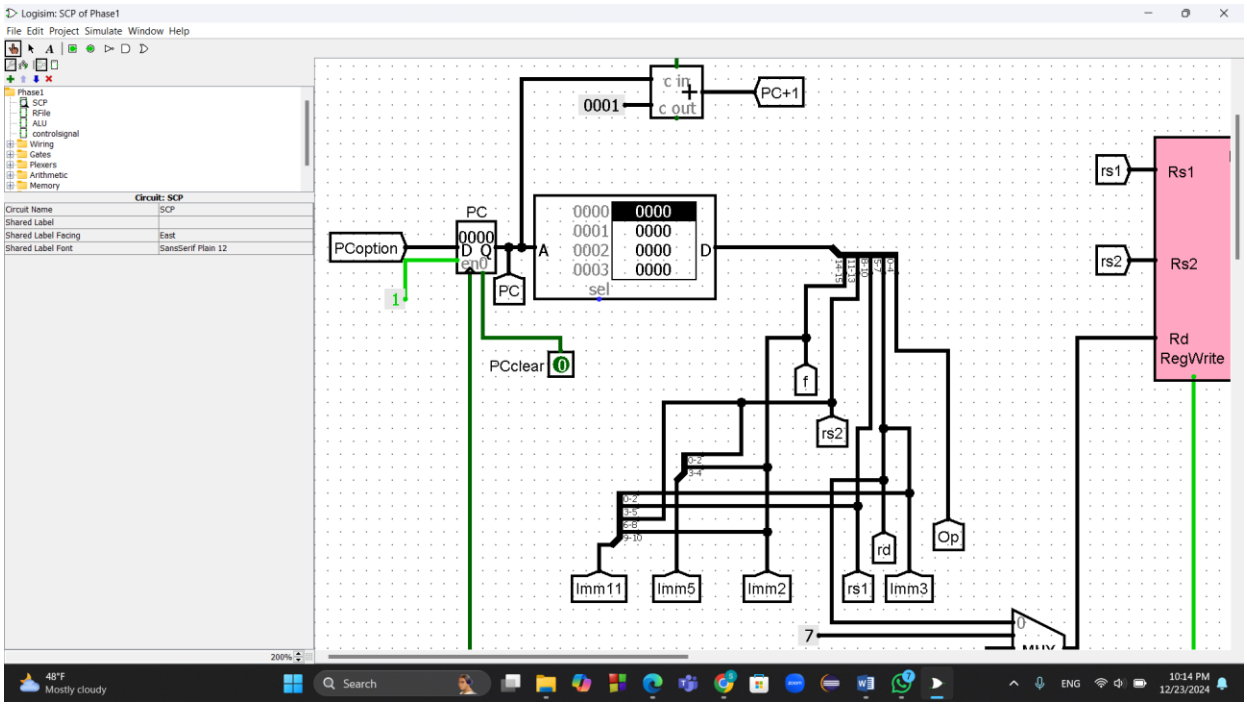


Figure 16: PC and instruction memory.

The RISC processor supports four types of instructions: R-type, I-type, SB-type, and J-type, each 16 bits long and aligned in memory.

R-type:

5-bit opcode (Op), 3-bit register numbers (*rs1*, *rs2*, and *rd*), and 2-bit function field (*f*)

f^2		$rs2^3$			$rs1^3$			rd^3			Op^5				
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 17: R-type.

R-type instructions are used for register-to-register operations, such as arithmetic and logical operations, and include fields for two source registers (rs1 and rs2), a destination register (rd), a 2-bit function field (f), and a 5-bit opcode (Op) that determines the operation.

I-type:

5-bit opcode (Op), 3-bit register numbers (*rs1* and *rd*) and 5-bit Immediate

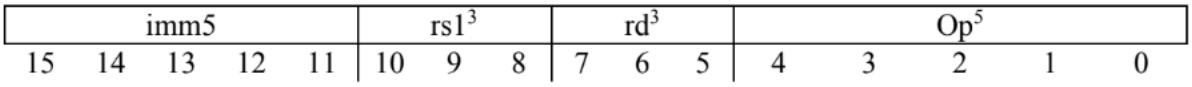


Figure 18: I-type.

I-type instructions perform operations involving an immediate value, such as arithmetic with constants or memory operations, and consist of a 5-bit immediate field (imm5), a source register (rs1), a destination register (rd), and a 5-bit opcode (Op).

SB-type:

5-bit opcode (Op), 3-bit register numbers (*rs1* and *rs2*) and 5-bit Immediate split into (imm2U and imm3L)

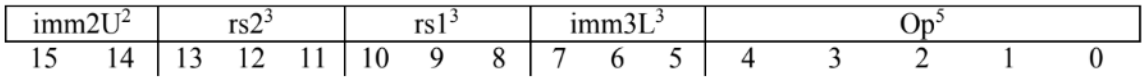


Figure 19: SB-type.

SB-type instructions are designed for conditional branches and include two registers (rs1 and rs2) for comparison, a split immediate field (imm2U and imm3L) to calculate the branch target in the pc, and a 5-bit opcode (Op) to specify the branch condition.

J-type:

5-bit opcode (Op) and 11-bit Immediate

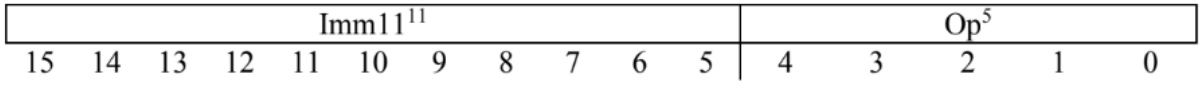


Figure 20: J-type.

J-type instructions support unconditional jumps and consist of an 11-bit immediate field (imm11) for the jump address and a 5-bit opcode (Op).

Register File

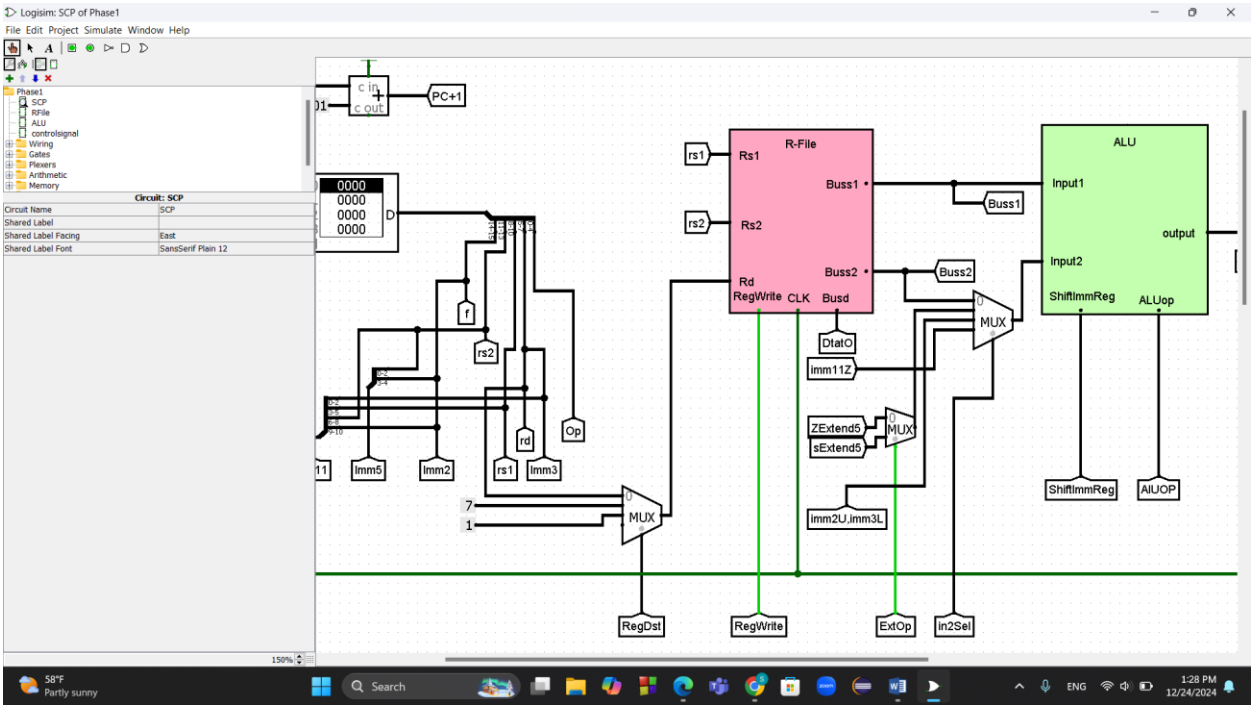


Figure 21: Register File.

The register file is a crucial component in the datapath of the processor, responsible for storing and providing the operands required by instructions. In this design, it contains seven 16-bit general-purpose registers (R1 to R7) along with the special register R0, which is hardwired to zero. The register file has three input ports: two for selecting source registers (Rs1 and Rs2) and one for specifying the destination register (Rd). It also has two output ports (Buss1 and Buss2) that carry the values of the source registers. RegWrite controls the writing operation; hence, data is only written to the destination register when required. The CLK input makes operations synchronized with each other. A multiplexer (MUX), connected to the Rd input, helps determine the information going through the format of the instruction (for example, R-type or I - type). There are many varieties of immediate extensions included in the register file: Zero Extends and sign Extends. Besides, just as the types of instruction differ, so do these extensions for their immediate5 values . To cater the immediate values to the 16-bit data width of the registers, an immediate value comes into operation with the multiplexers' connectivity configuration. The output multiplexers thus select either an immediate value or register data type as and when required.

ALU

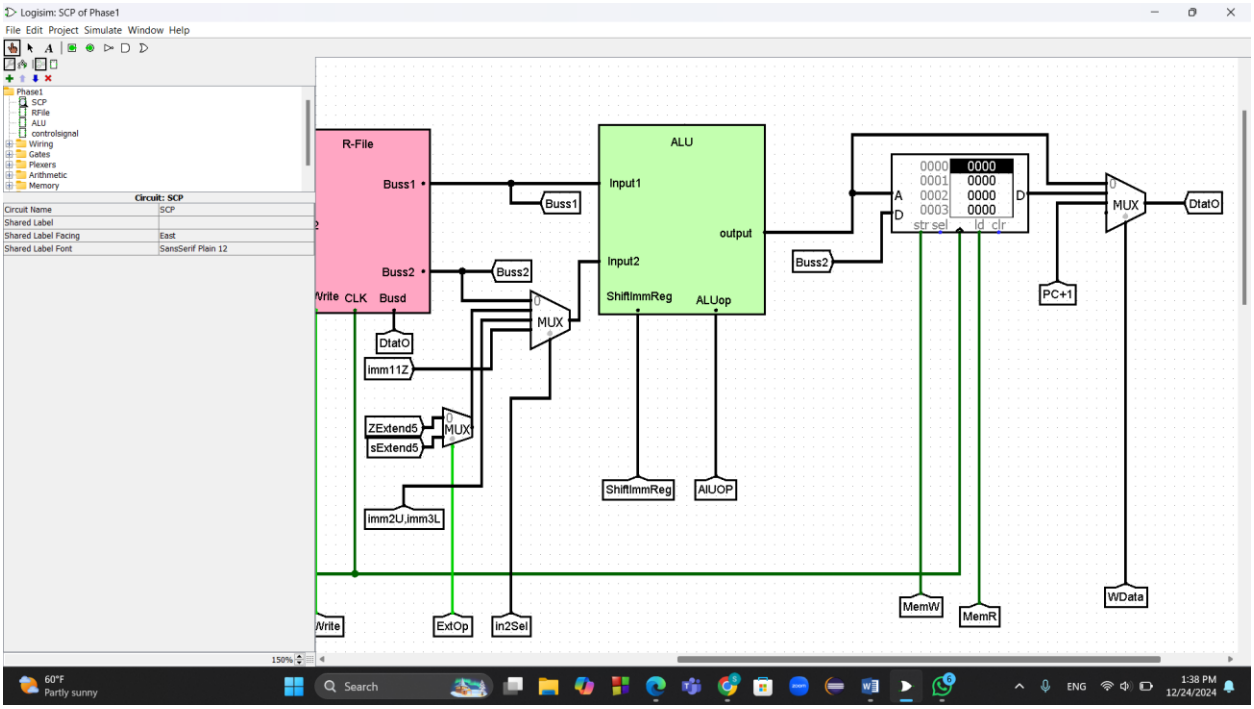


Figure 22: ALU.

The First operand (input) for the ALU is always Rs1 so connected directly with ALU (Buss1), but the second one may be one of four (e.g. Rs2 (Buss2) from Register File, zero or sign extend of Immediate 5, Zero extend of Immediate 11, or sign extend of {Immediate 3, Immediate 2}), so we connected all of these cases to Mux to choose one of them depends on the instruction type. The Result of the ALU is stored on Register or be the address of the data memory to read or write.

Data Memory

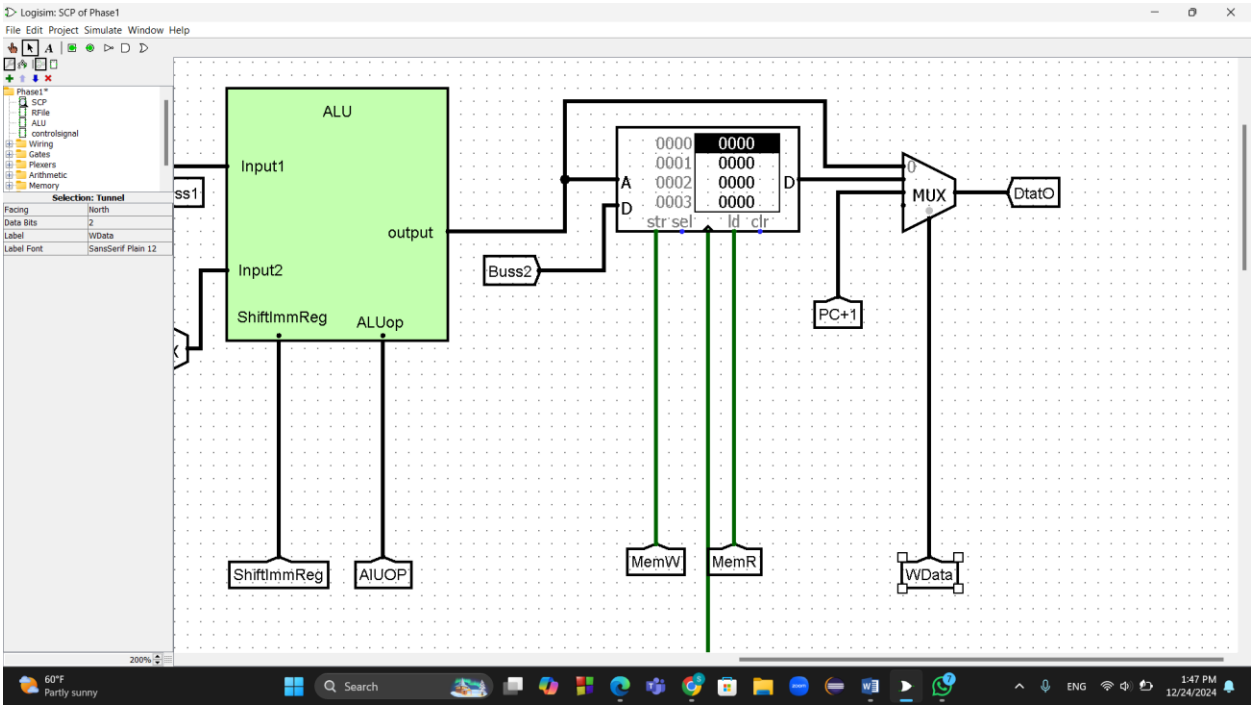


Figure 23: Data Memory.

The Data Memory is a critical component used to read from or write to memory during instruction execution. The Address input specifies the memory location to be accessed and is generated by the ALU, typically as a result of operations like Ra + (Imm5) (e.g., for the LW instruction). The data to be stored in memory comes from the content of register Rb, which is directly connected to the Write Data (WData) input of the memory. The memory's operation is controlled by two signals: MemW (Write Enable) and MemR (Read Enable), both generated by the control unit based on the instruction type. When MemW is active, the value in WData is written to the memory at the address specified by Address. When MemR is active, the memory outputs the data stored at the specified address to Dtato. This design ensures efficient interaction between the ALU, registers, and memory, allowing the processor to perform operations like loading data into registers (LW) or storing data from registers into memory (SW).

PC Controller

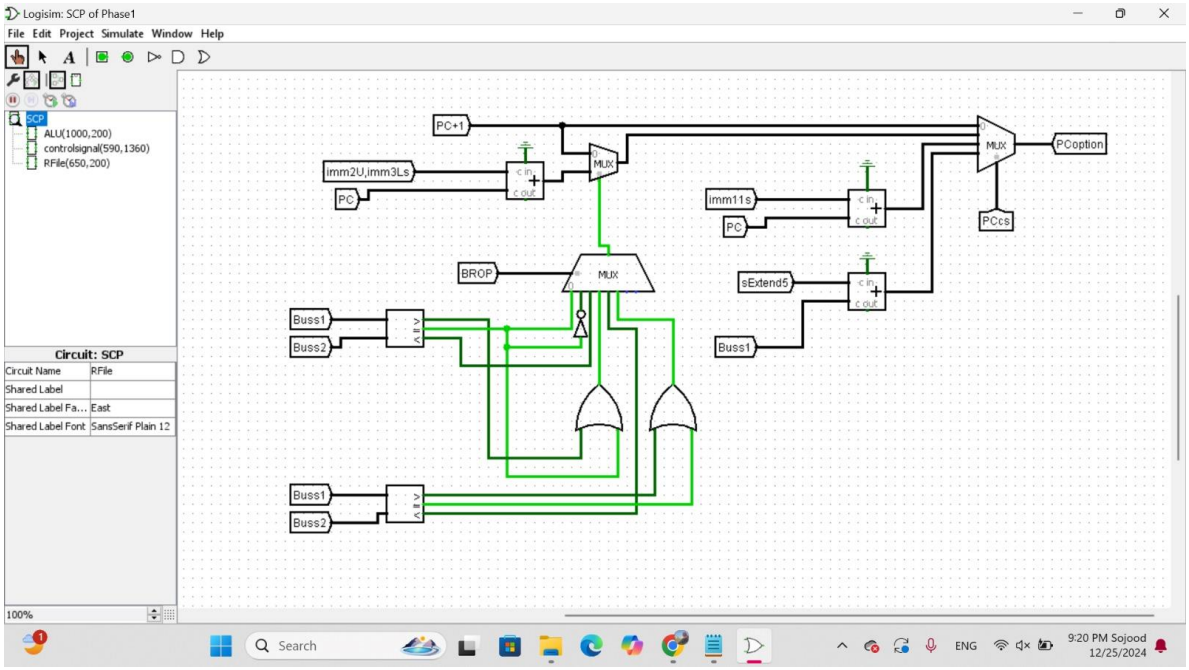


Figure 24: PC Controller.

The circuit diagram is for the control logic of Program Counter (PC) components according to the instruction types whose operations are sequentially executed, branch executed, or jumped-to executed. Multiplexors (MUX) are used to select the source for PC updates according to the condition under which the operation is being performed PC+1 for a sequential operation, PC + sign_extend({imm2U, imm3L}) for branch operation, and PC + sign_extend(imm11) for jump instructions. Furthermore, branch conditions are satisfied by comparing Ra with Rb (BEQ) or by judging Ra is not equal to Rb (BNE). Those conditions will determine whether the branch is taken or not taken. The sign extension or zero extension units, such as sign_extend(imm5) or sign_extend({imm2U, imm3L}), process immediates for effective addressing. A further example of a jump instruction is JALR, which also uses an address in a register to add the constant sign-extend(imm5) but conditionally stores the return address in the linking mechanism. Upon correct command opcode and control signals, the correct update of all PC update paths would then be initiated by the MUX PCoption so as to ensure the intricate control of the execution flow considering R-type, I-type, J-type, and branch instructions.

Control Unit

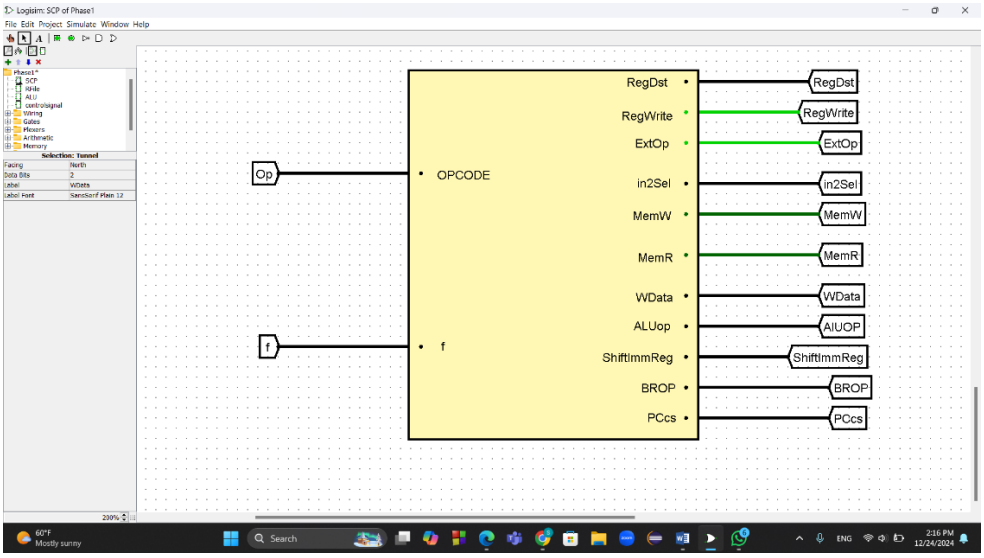


Figure 25: Control Unit Block.

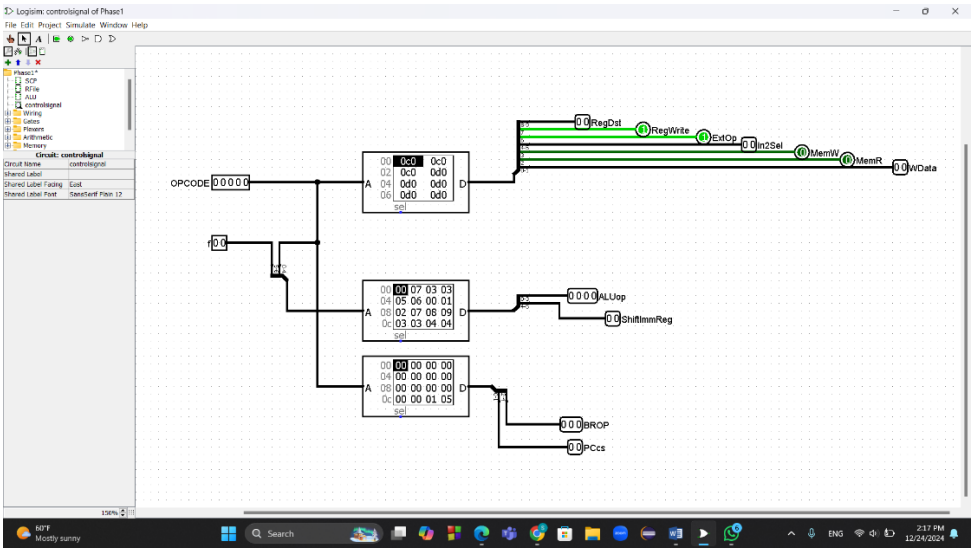


Figure 26: Control Unit.

The main control unit containing three Microprogrammed control units, one to generate general control signals, one to generate the opcode for the ALU, and the last is PC control unit.

Table 1: General Control Signal.

Control Signal	Number of bits	Description	Signals meaning
RegDst	2	Determine which Register to write on it , Rd or R7 or R1	00 >> Rd 01 >> R7 10 >> R1 11 >> xxx
RegWrite	1	Enable write on the Register or disable it	0 >> no write to memory 1 >> write to memory
ExtOp	1	Determine the type of extend to the Imm5, zero or sign extend	0 >> zero extend (Imm5) 1 >> sign extend (Imm5)
In2Sel	2	Determine the source (input) 2 to the ALU	00 >> Buss2 01 >> extend (Imm5) 10 >> (Imm3,Imm2) 11 >> Imm11
MemW	1	Enable write on memory or disable it	0 >> no write 1 >> write
MemR	1	Enable read from memory or disable it	0 >> no read 1 >> read
WData	2	Choose what to write on the Register, from memory, ALU Result, or PC+1	00 >> ALU result 01 >> Data memory 10 >> PC + 1 11 >> xxxxxxxxxxxxxxxx
BROP	3	Determines the type of branch operation.	000 >> BEQ 001 >> BNE 010 >> BLT 011 >> BGE 100 >> BLTU 101 >> BGEU 110 >> X 111 >> X
PCcs	2	determines how the Program Counter (PC) is updated	00 >> PC = PC + 1 01 >> PC=PC+sign_extend({imm2U, imm3L}) 10 >> PC=PC+sign_extend(imm11) 11 >> PC=rs1+sign_extend(imm5)

General Control Unit

We implement this Microprogrammed control unit to generate the general control signals (RegDst, RegWrite, ExtOp, In2Sel, MemW, MemR, WData). The RegDst signal (2 bits) determines the destination register for write back operations, with options including Rd, R7, or R1. The RegWrite signal (1 bit) indicates whether data is written to a register, with 1 enabling a write operation. The ExtOp signal specifies whether the immediate value (Imm5) is zero-extended (0) or sign-extended (1). The in2Sel signal (2 bits) selects the second ALU operand, which can come from Buss2, the extended immediate (Imm5), {Imm3, Imm2}, or Imm11. For memory operations, MemW and MemR (1 bit each) control whether memory is written or read, respectively. The WData signal (2 bits) determines the source of data for writing, such as the ALU result, data memory, or PC + 1. We use a ROM to store this control signals for each different opcode – the address of this memory is 5-bits (opcode).

(Op = 5bits), (RegDst = 2bits), (RegWrite = 1bit), (ExtOp =1bit), (In2Sel = 2bits), (MemW =1bit), (MemR = 1bit), (WData = 2bits).

Table 2: General Control Signal Values.

Opcode (Address) (5 bits)	Control signals (10 bits)	Control signals in Hexadecimal
00000 = 0x00	00_1_1_00_0_0_00	0x0C0
00001 = 0x01	00_1_1_00_0_0_00	0x0C0
00010 = 0x02	00_1_1_00_0_0_00	0x0C0
00011 = 0x03	00_1_1_01_0_0_00	0x0D0
00100 = 0x04	00_1_1_01_0_0_00	0x0D0
00101 = 0x05	00_1_1_01_0_0_00	0x0D0
00110 = 0x06	00_1_1_01_0_0_00	0x0D0
00111 = 0x07	00_1_1_01_0_0_00	0x0D0

01000 = 0x08	00_1_1_01_0_0_00	0x0D0
01001 = 0x09	00_1_1_01_0_0_00	0x0D0
01010 = 0x0A	00_1_1_01_0_0_00	0x0D0
01011 = 0x0B	00_1_1_01_0_0_00	0x0D0
01100 = 0x0C	00_1_0_01_0_1_01	0x095
01101 = 0x0D	00_0_0_10_1_0_01	0x029
01110 = 0x0E	00_0_0_10_0_0_01	0x021
01111 = 0x0F	00_0_0_10_0_0_01	0x021
10000 = 0x10	00_0_0_10_0_0_01	0x021
10001 = 0x11	00_0_0_10_0_0_01	0x021
10010 = 0x12	00_0_0_10_0_0_01	0x021
10011 = 0x13	00_0_0_10_0_0_01	0x021
10100 = 0x14	00_1_0_11_0_0_00	0x2B0
10101 = 0x15	00_0_0_00_0_0_00	0x000
10110 = 0x16	01_1_0_00_0_0_10	0x182
10111 = 0x17	00_1_0_00_0_0_10	0x082

ALU control unit

ALU control signals determine the operation that is performed by the arithmetic logic unit (ALU) as indicated by its function (f) and operation (Op) fields in the instruction. The ALUop field helps to determine that the specific ALU operation to be performed is either one of logical operations, shift operations, arithmetic operations and immediately related ones. For shift operations, the ShiftImmReg signal determines the selection between the source of the shift amount: immediate (SLLI, SRLI, SRAI) or for register (SLL, SRL, SRA). In the case of immediate shifts, ShiftImmReg then indicates the bit-width for immediate: 00 = 4 bits, 01 = 3 bits, 10 = 2 bits, 11 = 1 bit. Each unique combination of f and Op correlates to a specific ALUop value and leads to an efficient utilization of ALU by executing different instructions.

Table 3: ALU operation.

Arithmetic or Logical operation	OpCode of the ALU	Description
XOR	0000	Rd=Ra ^ Rb
OR	0001 = 0x1	Rd=Ra | Rb
AND	0010 = 0x2	Rd=Ra & Rb
ADD	0011 = 0x3	Rd=Ra + Rb
SUB	0100 = 0x4	Rd=Ra – Rb
SLT	0101 = 0x5	Rd=(Ra < Rb) signed
SLTU	0110 = 0x6	Rd=(Ra < Rb) unsigned
SLL	0111 = 0x7	Rd= Shift Left Logical (Ra, Rb[3:0])

SRL	1000 = 0x8	Rd= Shift Right Logical (Ra, Rb[3:0])
SRA	1001 = 0x9	Rd= Shift Right Arith (Ra, Rb[3:0])
LUI	1010 = 0xA	R1=imm11 << 5
XXXX	1011 = 0xB	Don't Care
XXXX	1100 = 0xC	Don't Care
XXXX	1101 = 0xD	Don't Care
XXXX	1110 = 0xE	Don't Care
XXXX	1111 = 0xF	Don't Care

Table 4: ALU Control signal values

{ f, Opcode } (Address) (7 bits)	ShiftImmReg (2 bits)	ALU opcode (4 bits)	Control signals in Hexadecimal
0000000 = 0x 00	00	0000	0x 00
0100000 = 0x20	00	0001	0x 01
1000000 = 0x40	00	0010	0x 02
0000010 = 0x02	00	0011	0x 03
0100010 = 0x22	00	0100	0x 04
1000010 = 0x42	00	0101	0x 05
1100010 = 0x62	00	0110	0x 06
0000001 = 0x01	00	0111	0x 07
0100001 = 0x21	00	1000	0x 08
1000001 = 0x41	00	1001	0x 09

xx00011 0000011 = 0x03 0100011 = 0x23 1000011 = 0x43 1100011 = 0x63	00	0011	0x 03
xx00100 0000100 = 0x04 0100100 = 0x24 1000100 = 0x44 1100100 = 0x64	00	0101	0x 05
xx00101 0000101 = 0x05 0100101 = 0x25 1000101 = 0x45 1100101 = 0x65	00	0110	0x 06
xx00110 0000110 = 0x06 0100110 = 0x26 1000110 = 0x46 1100110 = 0x66	00	0000	0x 00
xx00111 0000111 = 0x07 0100111 = 0x27 1000111 = 0x47 1100111 = 0x67	00	0001	0x 01
xx01000 0001000 = 0x08 0101000 = 0x28 1001000 = 0x48 1101000 = 0x68	00	0010	0x 02

xx01001 0001001 = 0x09 0101001 = 0x29 1001001 = 0x49 1101001 = 0x69	00	0111	0x07
xx01010 0001010 = 0x0A 0101010 = 0x2A 1001010 = 0x4A 1101010 = 0x6A	00	1000	0x08
xx01011 0001011 = 0x0B 0101011 = 0x2B 1001011 = 0x4B 1101011 = 0x6B	00	1001	0x09
xx01100 0001100 = 0x0C 0101100 = 0x2C 1001100 = 0x4C 1101100 = 0x6C	00	0011	0X03
xx01101 0001101 = 0x0D 0101101 = 0x2D 1001101 = 0x4D 1101101 = 0x6D	00	0011	0X03
xx01110 0001110 = 0x0E 0101110 = 0x2E	00	0100	0X04

1001110 = 0x4E 1101110 = 0x6E			
xx01111 0001111 = 0x0F 0101111 = 0x2F 1001111 = 0x4F 1101111 = 0x6F	00	0100	0X04
xx10000 0010000 = 0x10 0110000 = 0x30 1010000 = 0x50 1110000 = 0x70	00	0101	0x05
xx10001 0010001 = 0x11 0110001 = 0x31 1010001 = 0x51 1110001 = 0x71	00	0101	0x05
xx10010 0010010 = 0x12 0110010 = 0x32 1010010 = 0x52 1110010 = 0x72	00	0110	0x06
xx10011 0010011 = 0x13 0110011 = 0x33 1010011 = 0x53 1110011 = 0x73	00	0110	0x06
xx10100 0010100 = 0x14 0110100 = 0x34 1010100 = 0x54	00	1010	0x0A

1110100 = 0x74			

PC Control Unit

It is this Microprogrammed control unit that we have employed to generate the control signal, PCcs, for the Mux which utilizes conditions for the value of the next instruction (for example, auto increment e.g. PC + 1, conditional Branch, or unconditional Jump) (2-bits) and control signal BROP that determines the condition of the Branch to check it (e.g., Rs1 == Rs2). We use another ROM to store this control signals for different opcode; the address for this memory is 5-bits (opcode).

Table 5: PC Control Signal Values.

Opcode (Address) (5 bits)	BROP(3 bits)	PCcs (2 bits)	Control Signals in Hexadecimal
00000 = 0x 00	000 = 0x 0	00 = 0x 0	0x 00
00001 = 0x 01	000 = 0x 0	00 = 0x 0	0x 00
00010 = 0x 02	000 = 0x 0	00 = 0x 0	0x 00
00011 = 0x 03	000 = 0x 0	00 = 0x 0	0x 00
00100 = 0x 04	000 = 0x 0	00 = 0x 0	0x 00
00101 = 0x 05	000 = 0x 0	00 = 0x 0	0x 00
00110 = 0x 06	000 = 0x 0	00 = 0x 0	0x 00
00111 = 0x 07	000 = 0x 0	00 = 0x 0	0x 00
01000 = 0x 08	000 = 0x 0	00 = 0x 0	0x 00
01001 = 0x 09	000 = 0x 0	00 = 0x 0	0x 00
01010 = 0x 0A	000 = 0x 0	00 = 0x 0	0x 00
01011 = 0x 0B	000 = 0x 0	00 = 0x 0	0x 00
01100 = 0x 0C	000 = 0x 0	00 = 0x 0	0x 00
01101 = 0x 0D	000 = 0x 0	00 = 0x 0	0x 00
01110 = 0x 0E	000 = 0x 0	01 = 0x 1	0x 01
01111 = 0x 0F	001 = 0x 1	01 = 0x 1	0x 05
10000 = 0x 10	010 = 0x 2	01 = 0x 1	0x 09
10001 = 0x 11	011 = 0x 3	01 = 0x 1	0x 0D
10010 = 0x 12	100 = 0x 4	01 = 0x 1	0x 11
10011 = 0x 13	101 = 0x 5	01 = 0x 1	0x 15
10100 = 0x 14	000 = 0x 0	00 = 0x 0	0x 00
10101 = 0x 15	000 = 0x 0	10 = 0x 2	0x 02
10110 = 0x 16	000 = 0x 0	10 = 0x 2	0x 02
10111 = 0x 17	000 = 0x 0	11 = 0x 3	0x 03

Simulation and Testing:

Table 6: R-type test.

Address of instruction	Assembly	Binary	Hexadecimal
0x 0000	LW 0,R1,R2	0000 0001 0100 1100	0x014C
0x 0001	LW 1,R1,R3	0000 1001 0110 1100	0x096C
0x 0002	XOR (f=0),R2,R3,R4	0001 0011 1000 0000	0x1380
0x 0003	OR R2,R3,R5	0101 0011 1010 0000	0x53A0
0x 0004	AND R2,R3,R7	1001 0011 1110 0000	0x93E0
0x 0005	SLL R2,R3,R4	0001 0011 1000 0001	0x1381
0x 0006	SRL R2,R3,R5	0101 0011 1010 0001	0x53A1
0x 0007	SRA R2,R3,R5	1001 0011 1010 0001	0x93A1
0x 0008	ADD R2,R3,R5	0001 0011 1010 0010	0x13A2
0x 0009	SUB R2,R3,R5	0101 0011 1010 0010	0x53A2
0x 000A	SLT R2,R3,R5	1001 0011 1010 0010	0x93A2
0x 000B	SLTU R2,R3,R5	1101 0011 1010 0010	0xD3A2

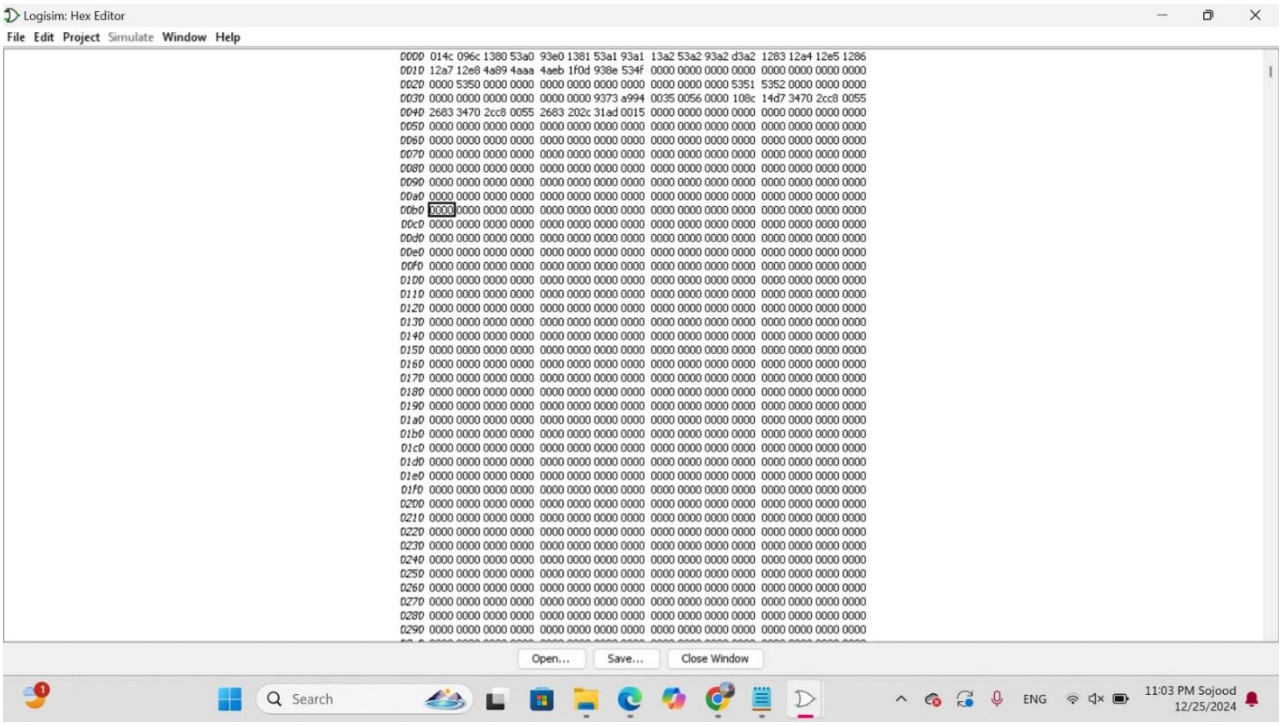


Figure 27: Instruction Memory.

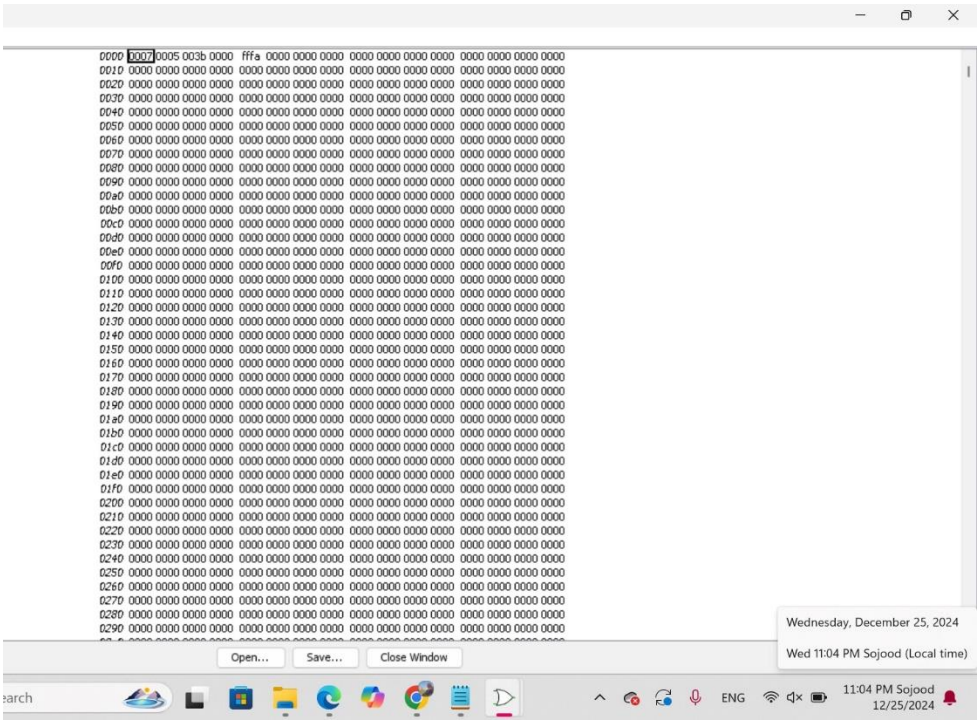


Figure 28: Data Memory.

LW 0, R1, R2 -> Load to R2 the content of the Address R1 + (00000), the content of R1 is 0 so Load the Data at address 0x 00000 to R2, so the content of R2 will be 0x 0007.

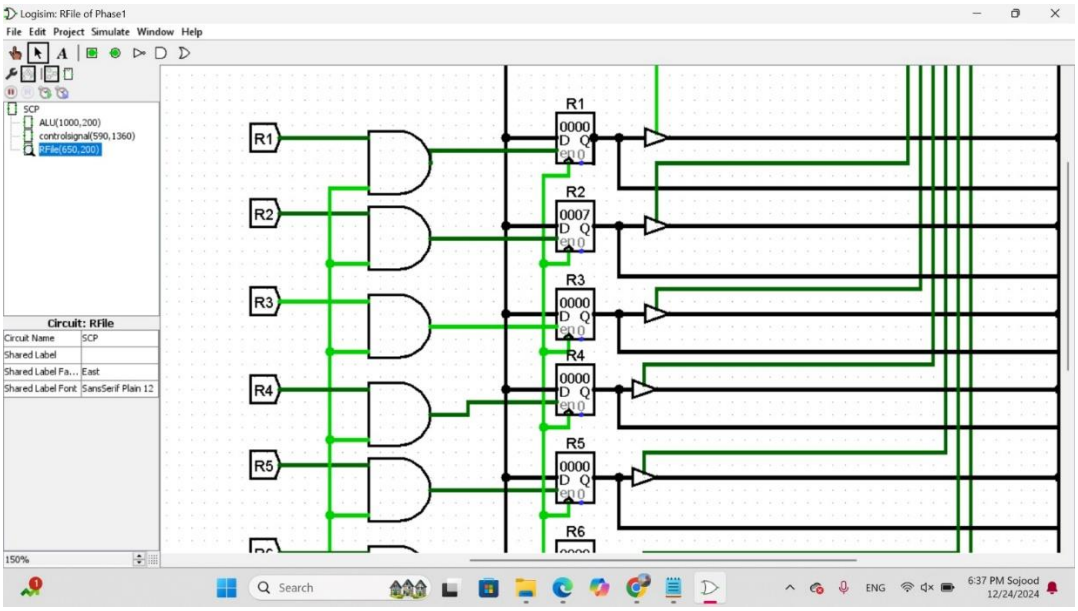


Figure 29: LW1 test.

LW 1, R1, R3 -> Load to R3 the content of the Address R1 + (00001), the content of R1 is 0 so Load the Data at address 0x 00001 to R3, so the content of R3 will be 0x 0005.

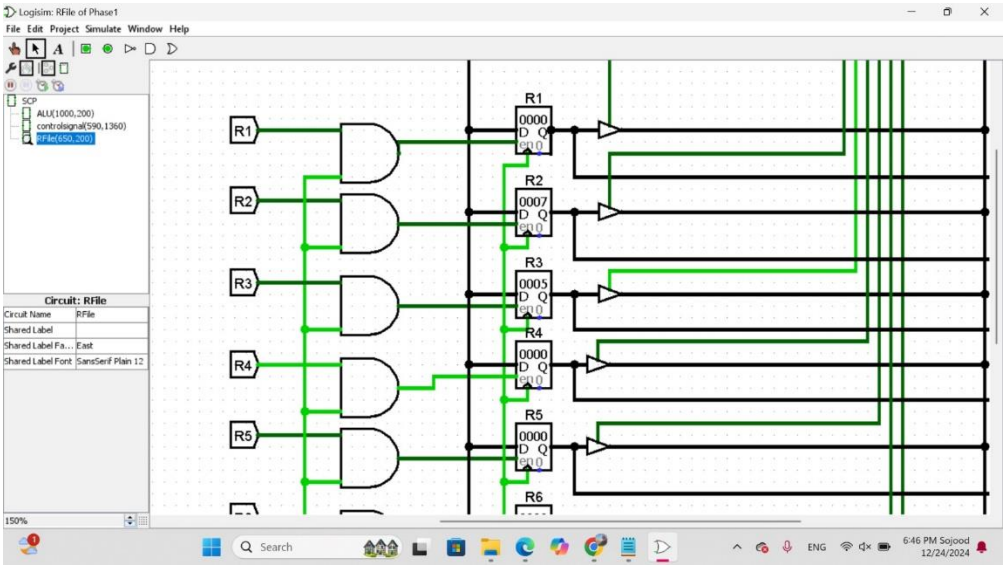


Figure 30: LW2 test.

XOR R2, R3, R4 -> R4 = R2 xor R3 = (0000 0000 0000 0111) xor (0000 0000 0000 0101) = (0000 0000 0000 0010) = 0x0002.

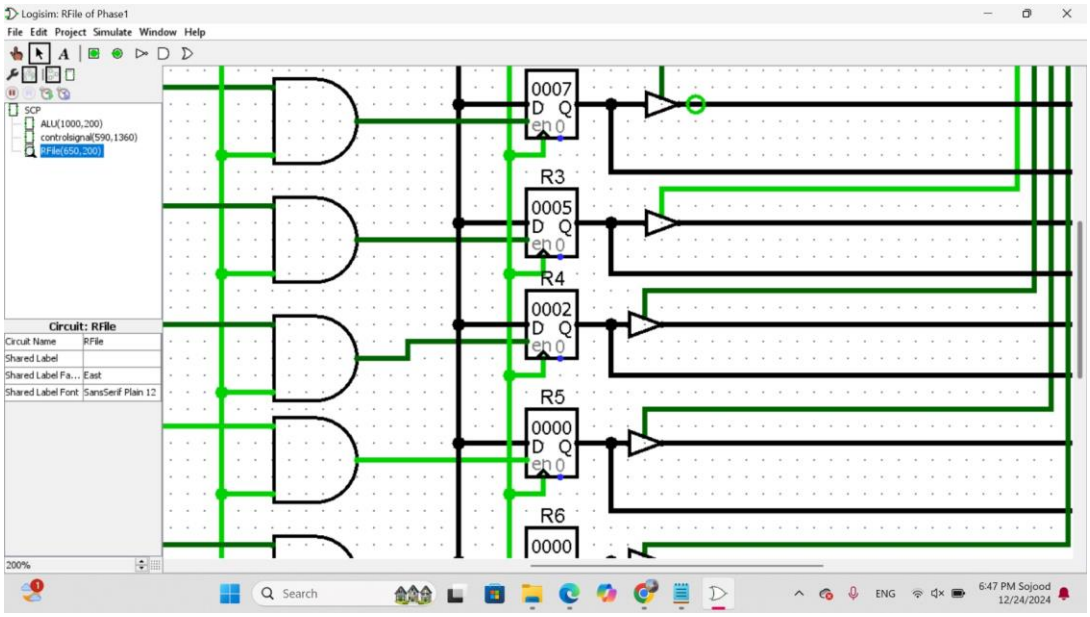


Figure 31: XOR test.

OR R2, R3, R5 -> R5 = R2 or R3 = (0000 0000 0000 0111) or (0000 0000 0000 0101) = (0000 0000 0000 0111) = 0x0007.

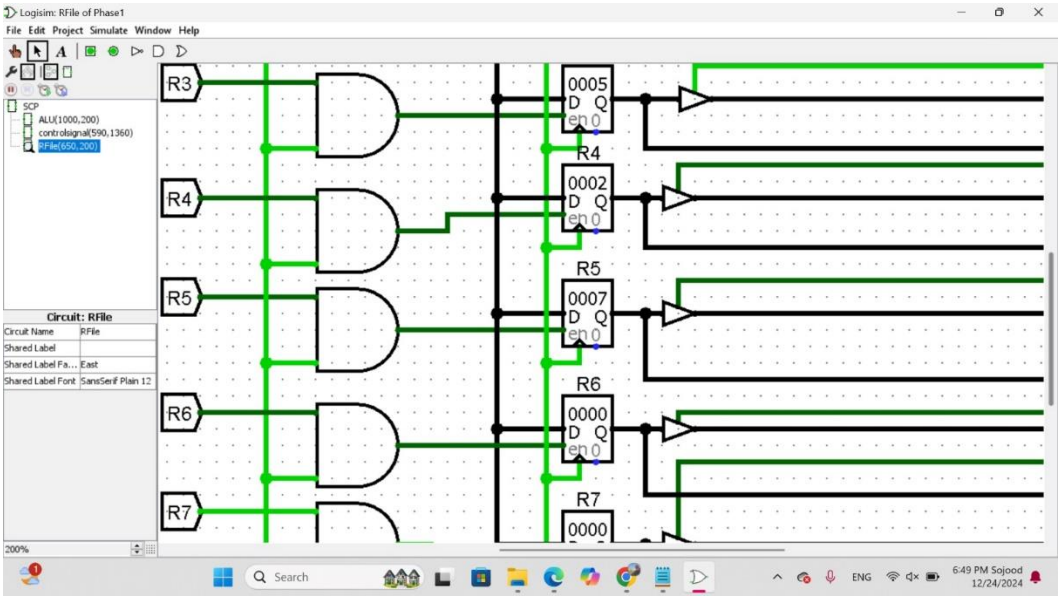


Figure 32: OR test.

AND R2, R3, R7 -> R7 = R2 and R3 = (0000 0000 0000 0111) and (0000 0000 0000 0101) = (0000 0000 0000 0101) = 0x0005

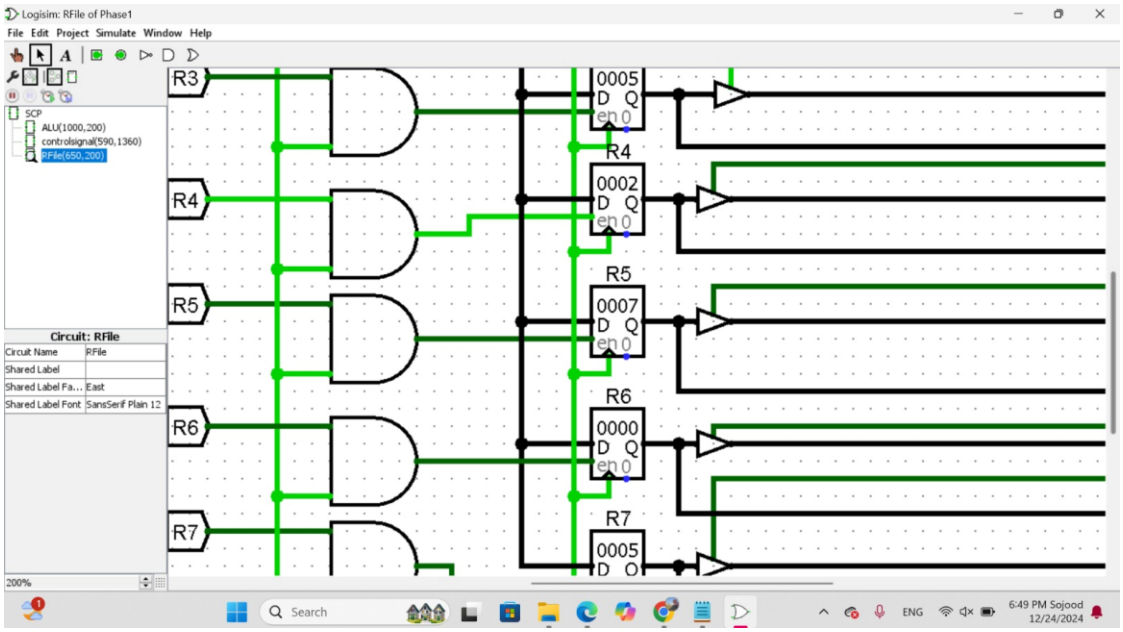


Figure 33: AND test.

SLL R2, R3, R4 -> R4 = shift left logical for R3 by 7 bits -> R4 = (0000 0010 1000 0000) > 0x0280

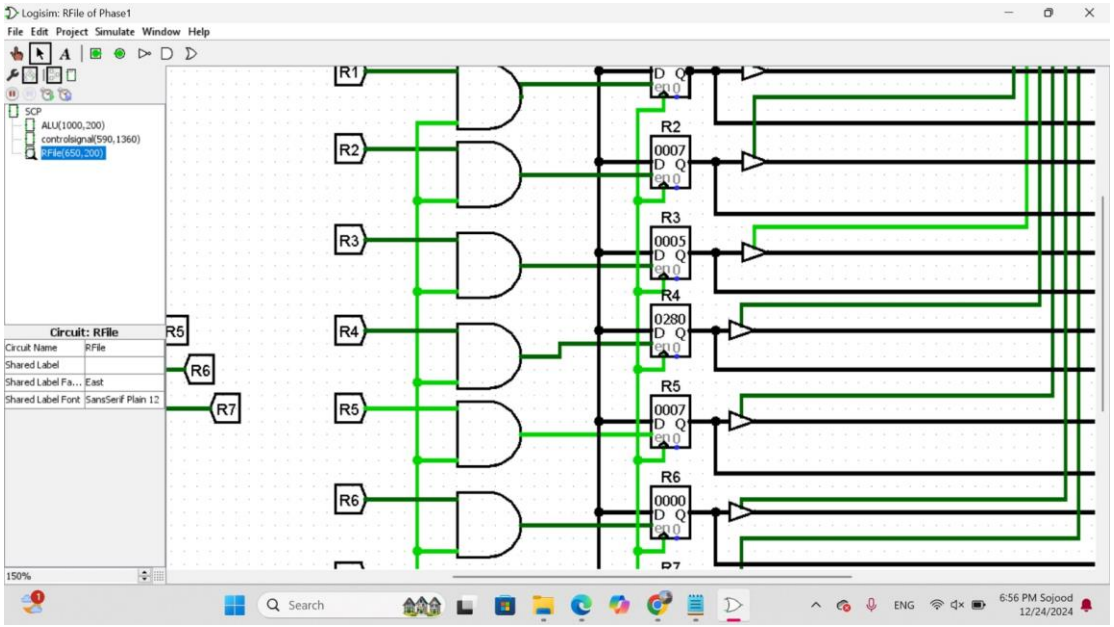


Figure 34: SLL test.

SRL R2, R3, R5 >> R5 = shift Right logical for R3 by 7 bits >> R5 = (0000 0000 0000 0000) >> 0x0000

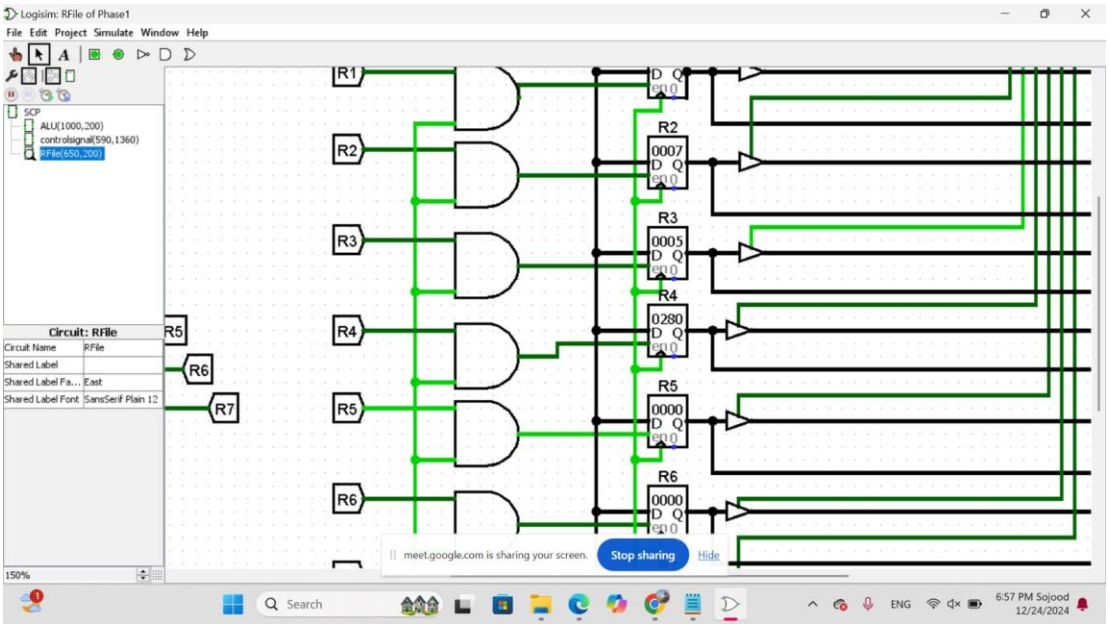


Figure 35: SRL test.

SRA R2, R3, R5 >> R5 = shift Right Arithmetic for R3 by 7 bits >> R5 = (0000 0000 0000 0000) >> 0x0000

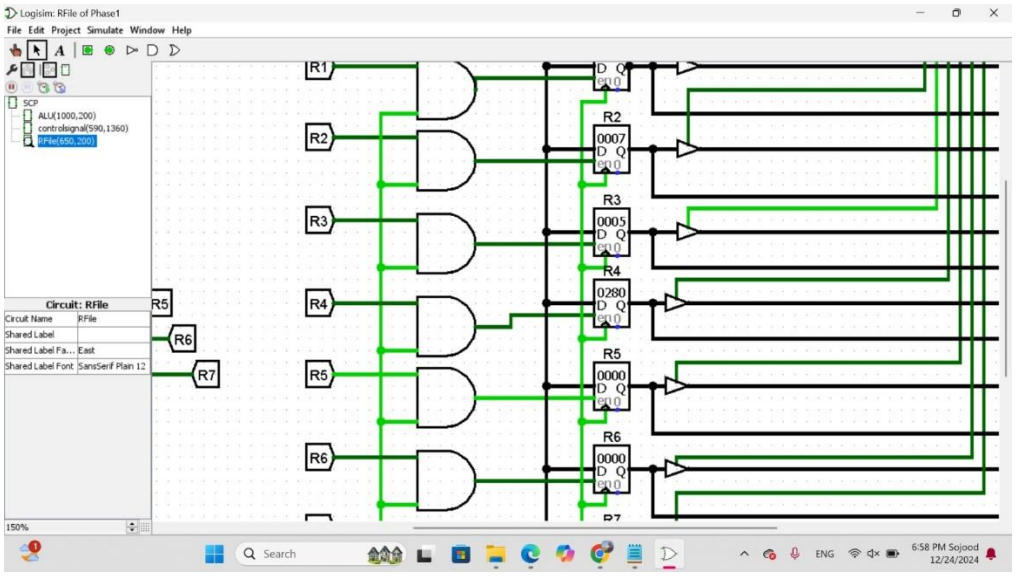


Figure 36: SRA test.

ADD R2, R3, R5 >> R5 = R2 + R3 = (0000 0000 0000 0111) + (0000 0000 0000 0101) = (0000 0000 0000 1100) = 0x000C

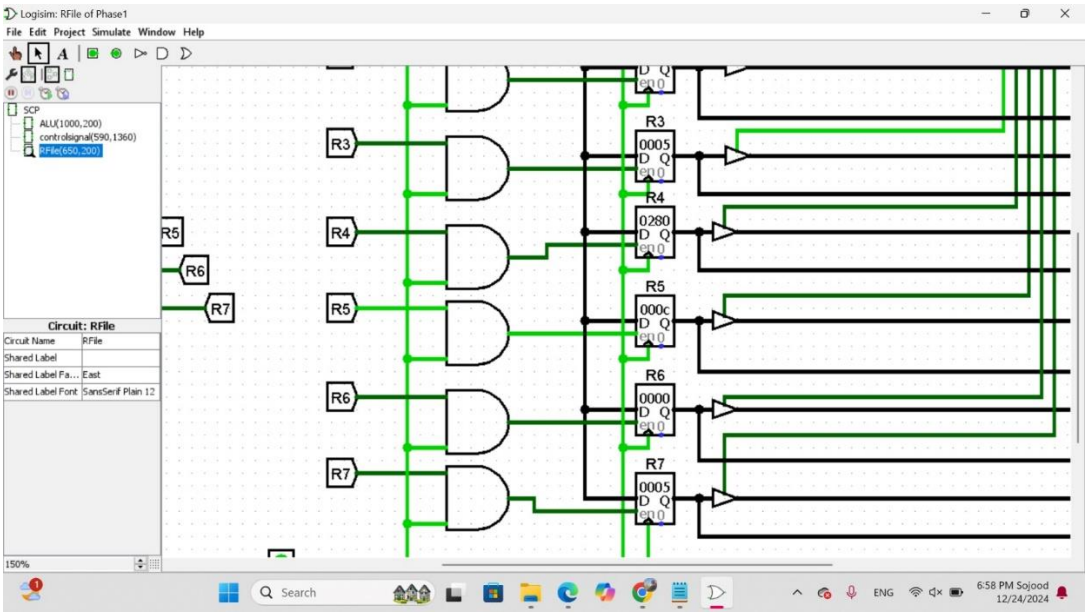


Figure 37: ADD test.

SUB R2, R3, R5 >> R5 = R3 - R2 = (0000 0000 0000 0101) - (0000 0000 0000 0111) = (0000 0000 0000 0101) + (1111 1111 1111 1001) = (1111 1111 1111 1110) = 0XFFFE

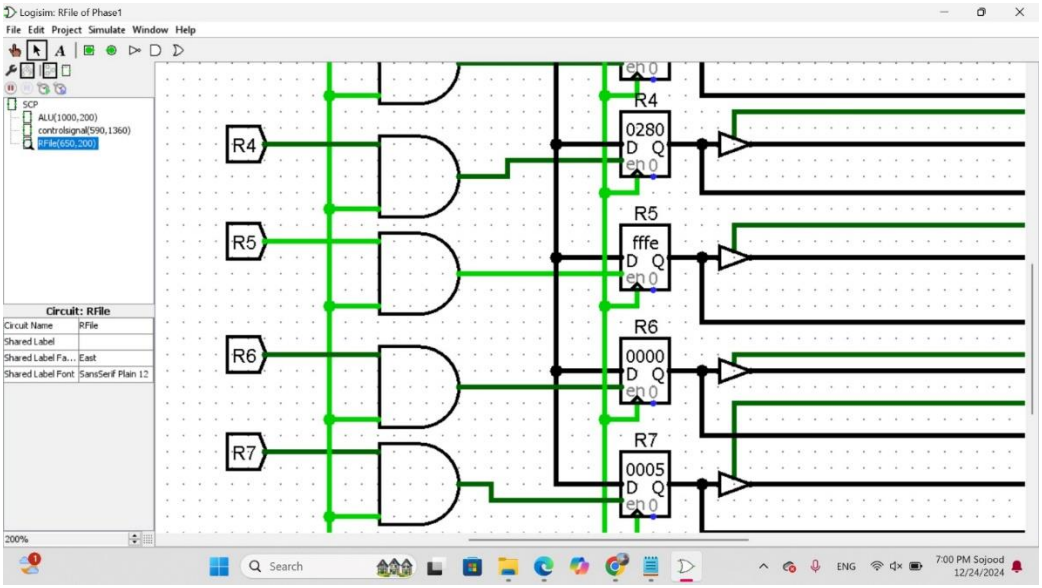


Figure 38: SUB test.

SLT R2, R3, R5 >> R5 = R3 < R2 (signed) >> (0000 0000 0000 0101) < (0000 0000 0000 0111) = (0000 0000 0000 0001)

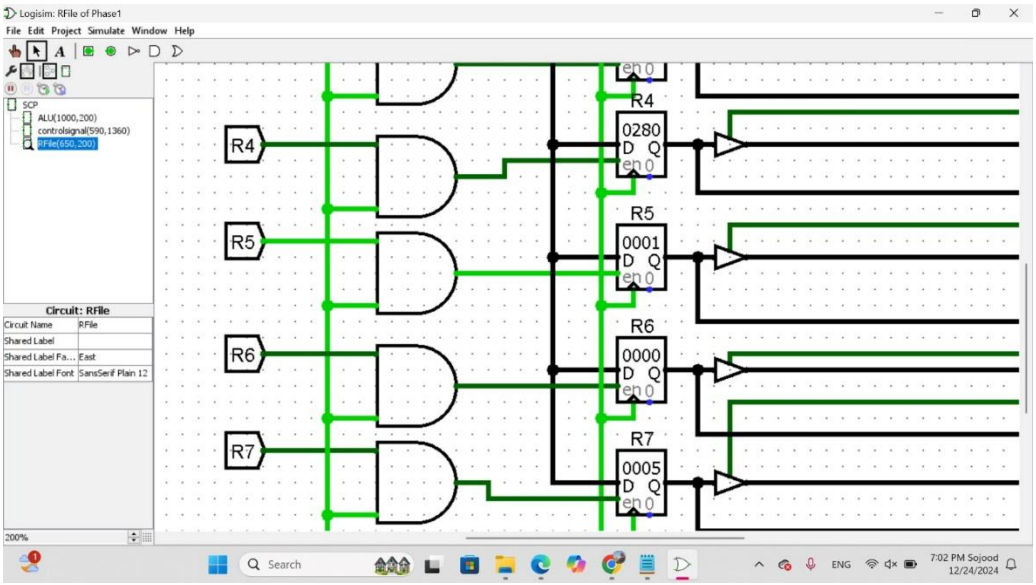


Figure 39: SLT test.

SLTU R2, R3, R5 >> R5 = R3 < R2 (Unsigned) >> (0000 0000 0000 0101) < (0000 0000 0000 0111) = (0000 0000 0000 0001)

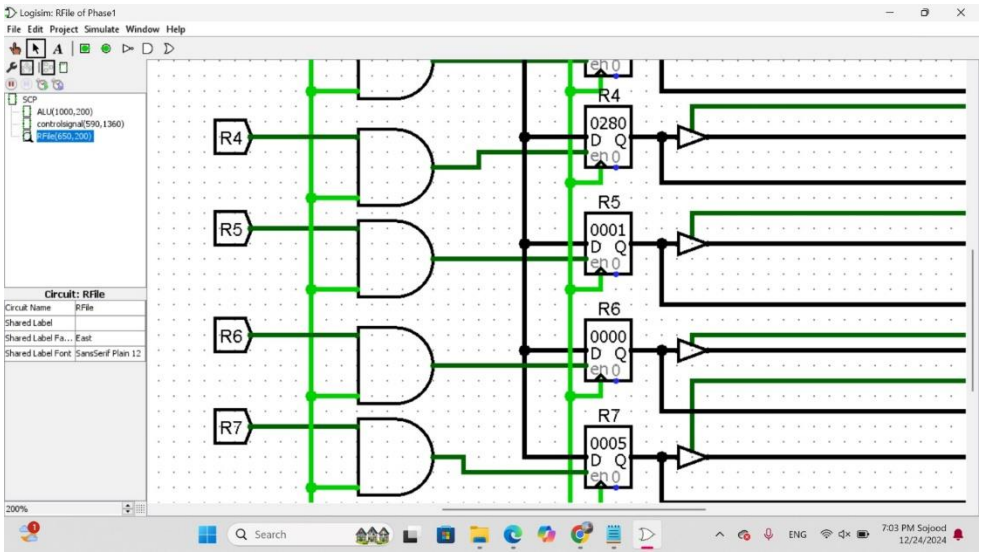


Figure 40: SLTU test.

Table 7: I-type test.

Address of instruction	Assembly	Binary	Hexadecimal
0x 000C	ADDI 2,R2,R4	0001-0010-1000-0011	0x1283
0x 000D	SLTI 2,R2,R5	0001-0010-1010-0100	0x12A4
0x 000E	SLTIU 2,R2,R7	0001-0010-1110-0101	0x12E5
0x 000F	XORI 2,R2,R4	0001-0010-1000-0110	0x1286
0x 0010	ORI 2,R2,R5	0001-0010-1010-0111	0x12A7
0x 0011	ANDI 2,R2,R7	0001-0010-1110-1000	0x12E8
0x 0012	SLLI 0,9,R2,R4	0100-1010-1000-1001	0x4A89
0x 0013	SRLI 0,9,R2,R5	0100-1010-1010-1010	0x4AAA
0x 0014	SRAI 0,9,R2,R7	0100-1010-1110-1011	0x4AEB

ADDI 2, R2, R4 >> R4 = R2 + Imm(2) = (0000 0000 0000 0111) + (0000 0000 0000 0010) = (0000 0000 0000 1001) >> 0x0009

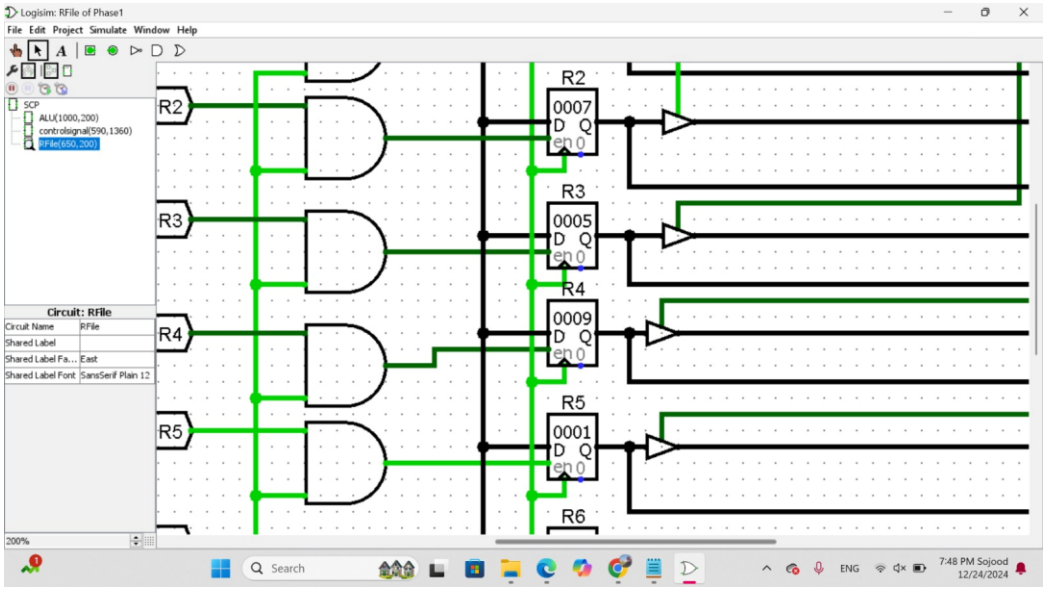


Figure 41: ADDI test.

SLTI 2, R2, R5 >> R5 = (R2 < sign_extend(IMM5)) = ((0000 0000 0000 0111) < (0000 0000 0000 0010)) = 0000 0000 0000 0000 >> 0x0000

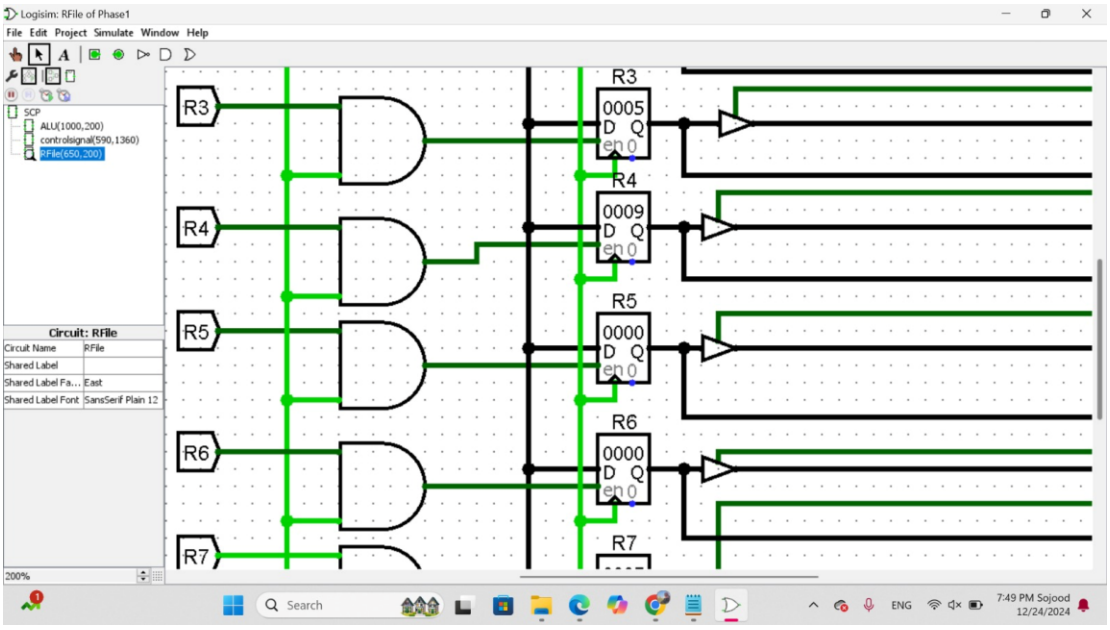


Figure 42: SLTI test.

SLTIU 2, R2, R7 >> R7 = (R2 < sign_extend(IMM5)) = ((0000 0000 0000 0111) < (0000 0000 0000 0010)) = 0000 0000 0000 0000 >> 0x0000

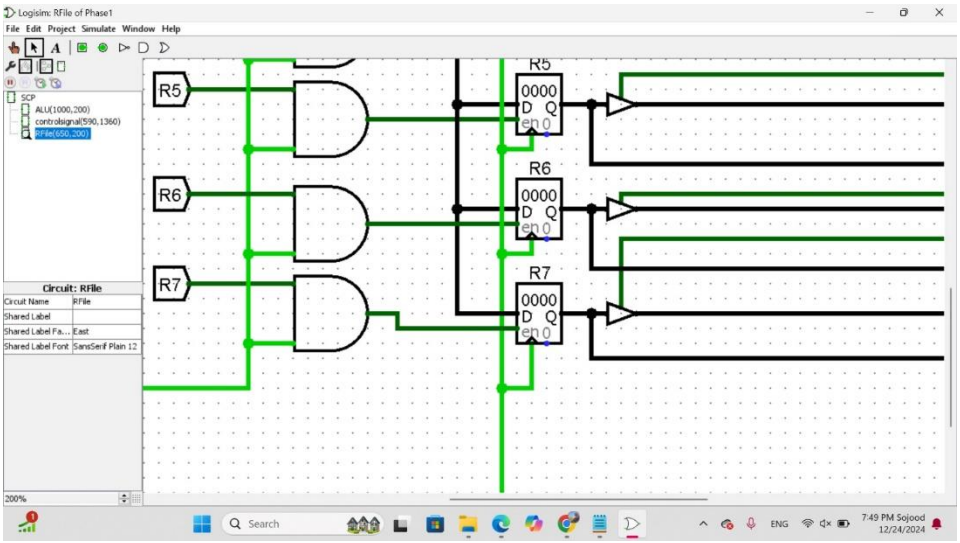


Figure 43: SLTIU test.

XORI 2, R2, R4 >> R4 = R2 xor IMM5 = R2 xor (2) = (0000 0000 0000 0111) xor (0000 0000 0000 0010) = 0000 0000 0000 0101 >> 0x0005

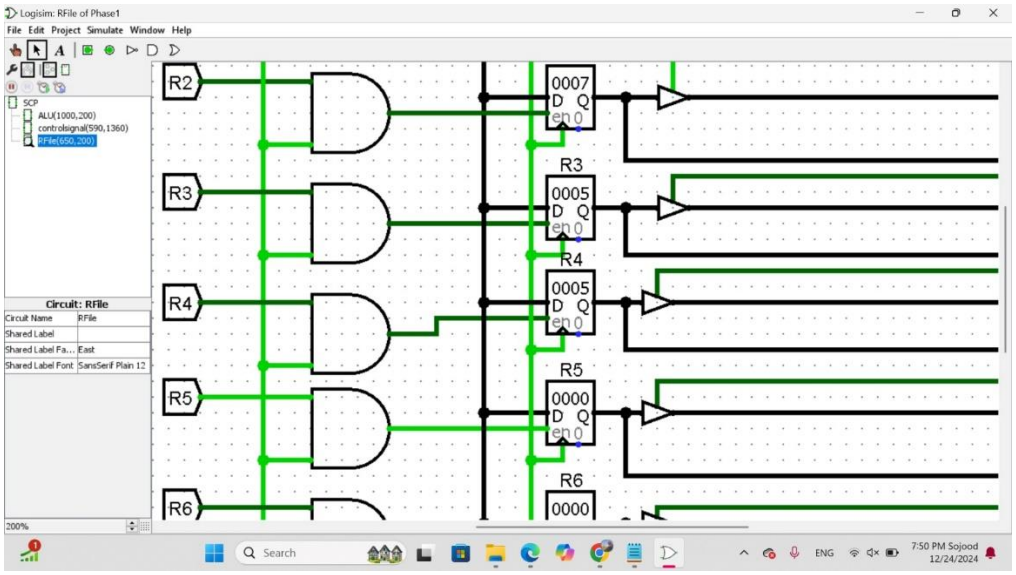


Figure 44: XORI test.

ORI 2, R2, R5 >> R5 = R2 or IMM5 = R2 or (2) = (0000 0000 0000 0111) or (0000 0000 0000 0010) = 0000 0000 0000 0111 >> 0x0007

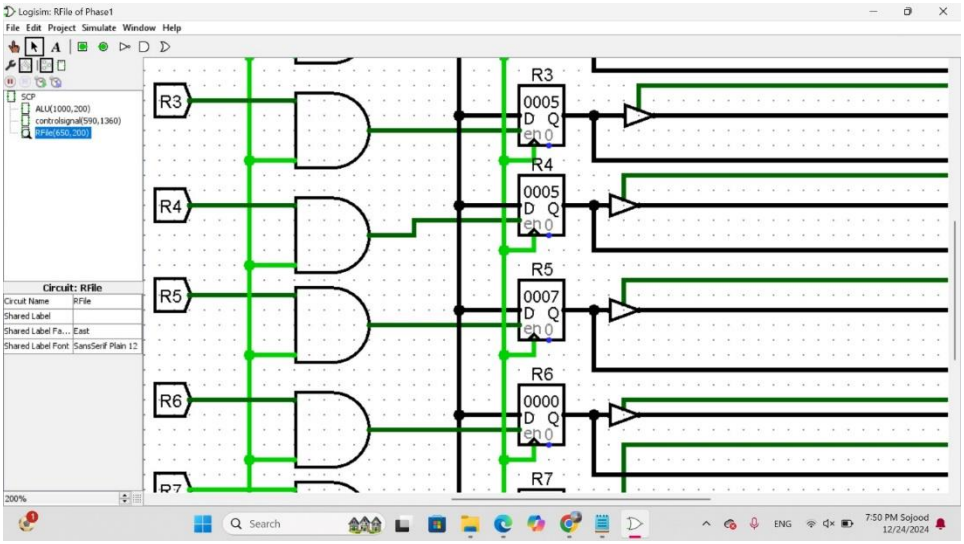


Figure 45: ORI test.

ANDI 2, R2, R7 >> R7 = R2 and IMM5 = R2 and (2) = (0000 0000 0000 0111) and (0000 0000 0000 0010) = 0000 0000 0000 0010 >> 0x0002

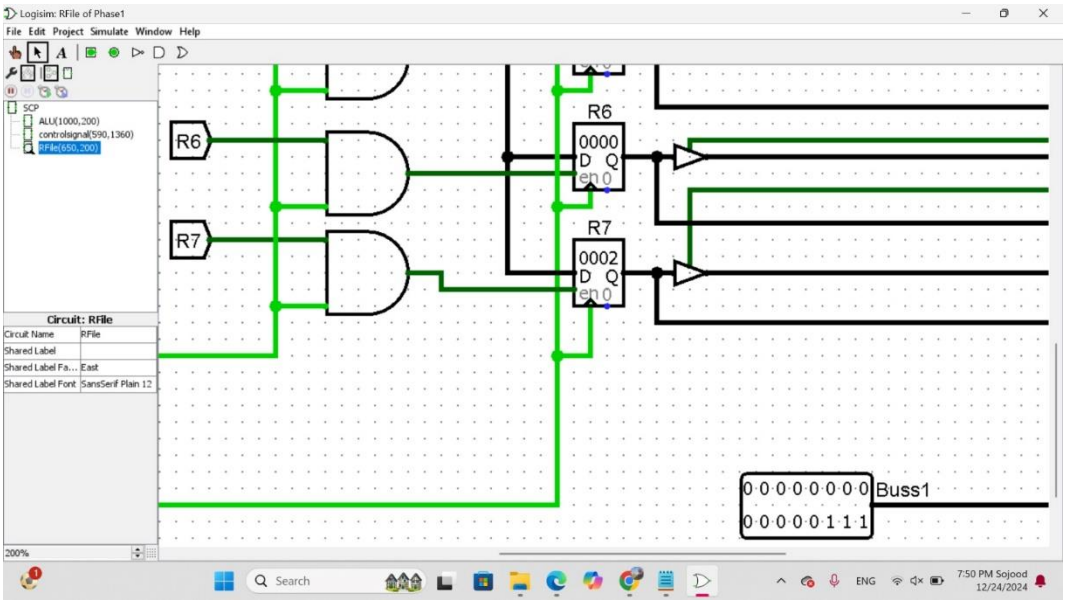


Figure 46: ANDI test.

SLLI 0, 9, R2, R4 >> R4 = shift left logical for R2 by 9 bits -> R4 = (0000 1110 0000 0000) >> 0x0E00

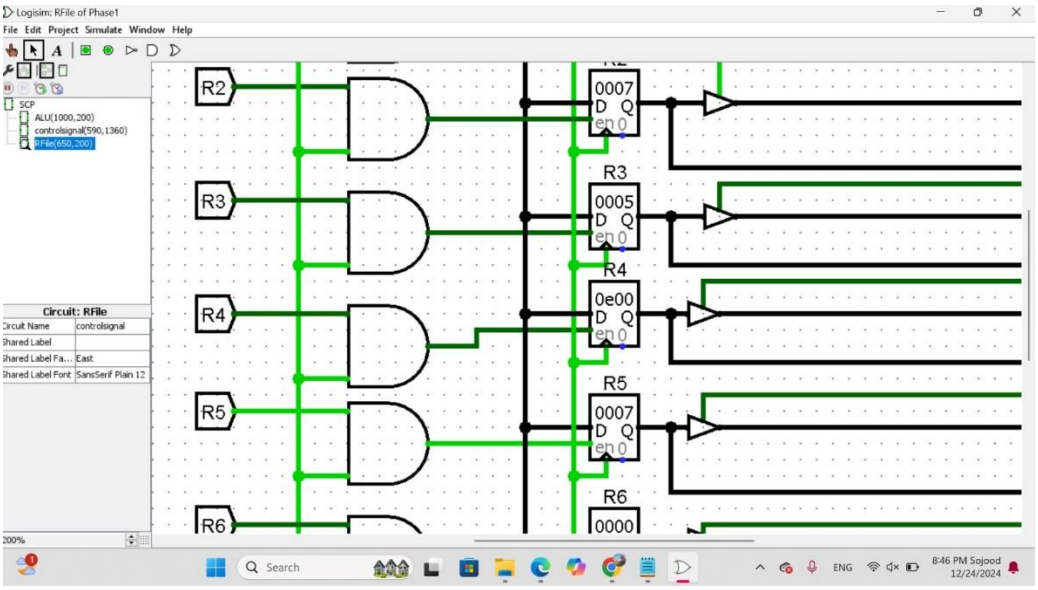


Figure 47: SLLI test.

SRLI 0, 9, R2, R5 -> R5 = shift Rigth logical for R2 by 9 bits >> R5 = (0000 0000 0000 0000) >> 0x0000

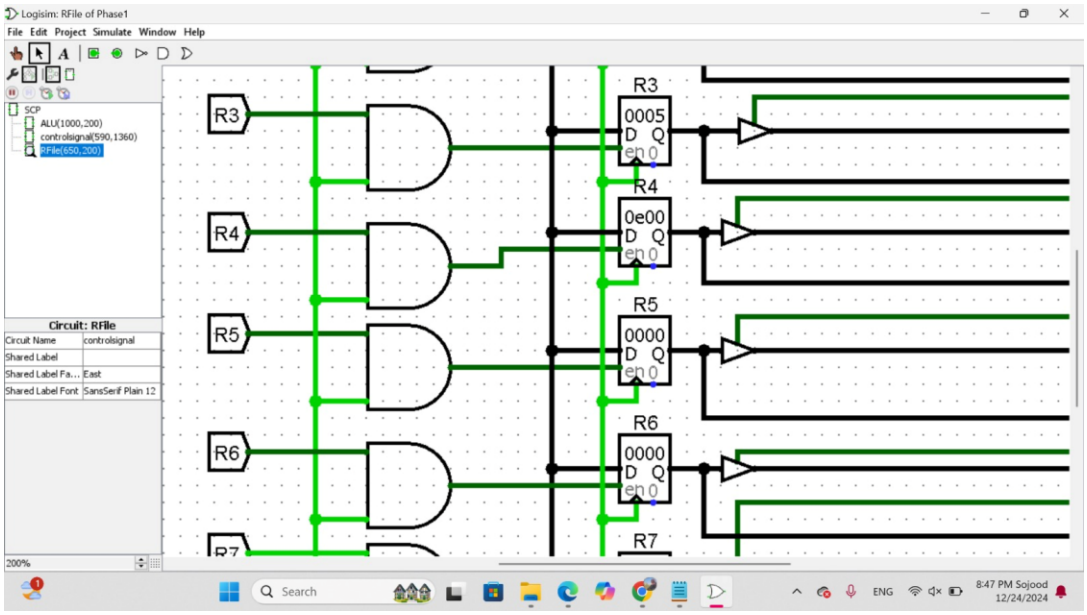


Figure 48: SRLI test.

SRAI 0, 9, R2, R7 >> R7 = shift Right Arithmetic for R2 by 9 bits >> R7 = (0000 0000 0000 0000) >> 0x0000

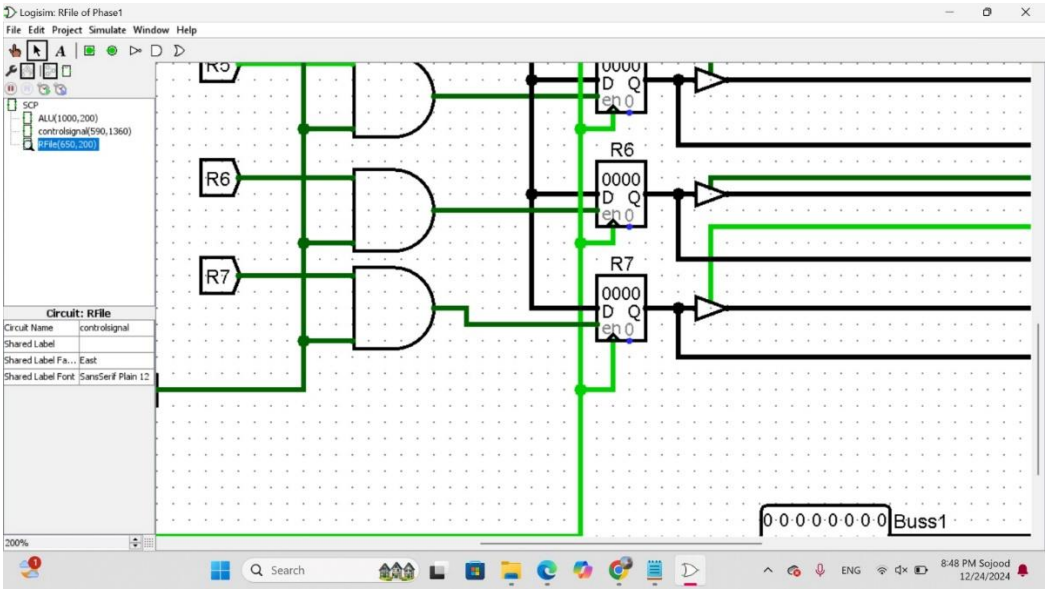


Figure 49: SRAI test.

Table 8: SB-type test.

Address of instruction	Assembly	Binary	Hexadecimal
0x0015	SW 0,R3,R7,0	0001 1111 0000 1101	0x1F0D
0x0016	BEQ 2,R2,R3,4	1001 0011 1000 1110	0x938E
0x0017	BNE 1,R2,R3,2	0101 0011 0100 1111	0x534F
0x0021	BLT 1,R2,R3,2	0101 0011 0101 0000	0x5350
0x002B	BGE 1,R2,R3,2	0101 0011 0101 0001	0x5351
0x002C	BLTU 1,R2,R3,2	0101 0011 0101 0010	0x5352
0x0036	BGEU 2,R2,R3,3	1001 0011 0111 0011	0x9373

Store R2 contents to memory address (R7 + (Imm3, Imm2)) = R7 + 0, and R7 = 0 then Store the content of R2 to the Address 0 of the memory.

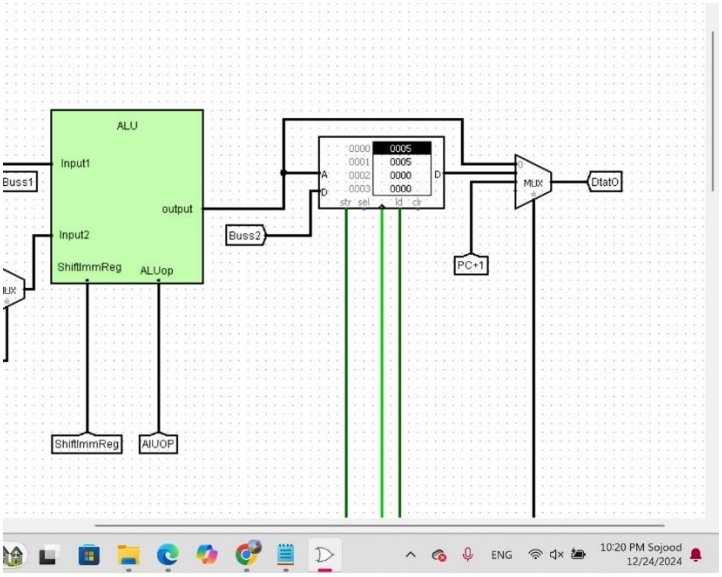


Figure 50: SW test.

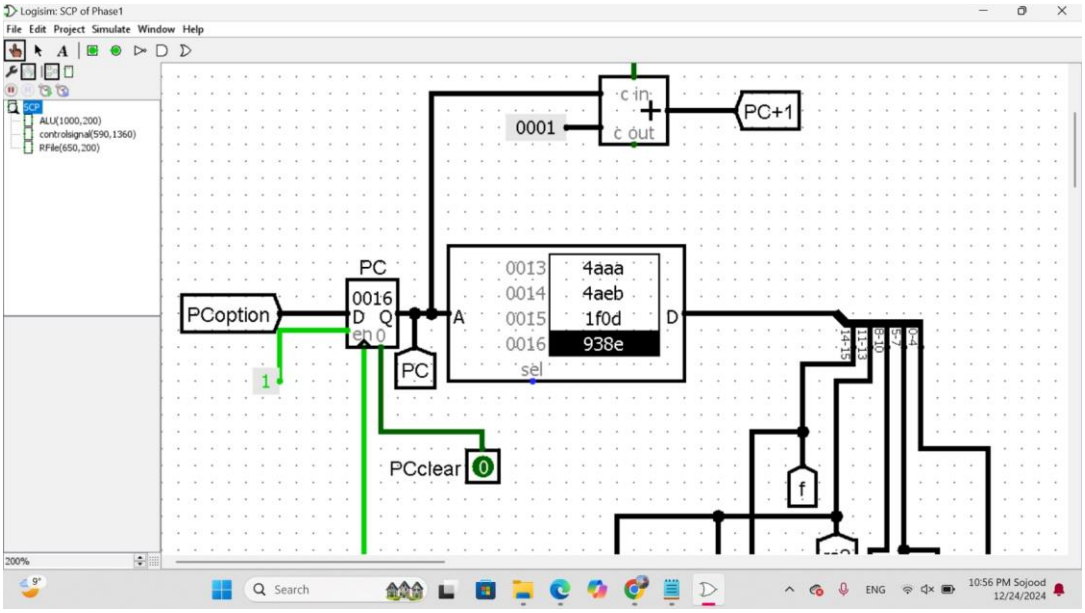


Figure 51: Before Branch.

BEQ 2, R2, R3, 4 if(R3==R2) PC = PC + {IMM2,IMM3} PC = 0000 0000 0001 0110 + 0000 0000 0001 0100 else PC = PC+1 here R3 = 0X0005 , R2 = 0X0007 and not equal so PC = 0016 + 1 = 0X0017.

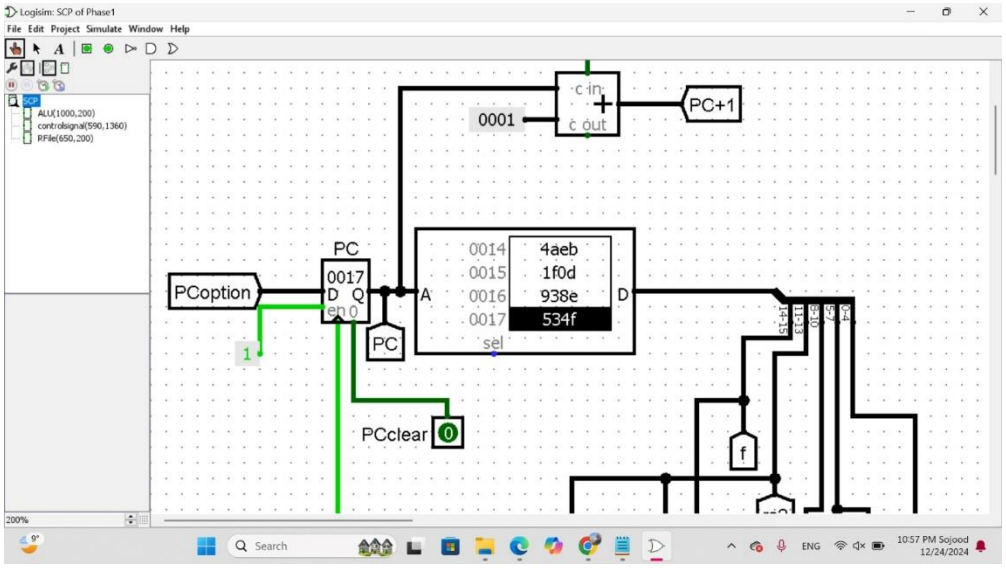


Figure 52: BEQ test.

BNE 1,R2,R3,2 if(R3!=R2) PC = PC + sign_extend{IMM2,IMM3} PC = 0000 0000 0001 0111 + 0000 0000 0000 1010 else PC = PC+1 here R3 = 0X0005 , R2 = 0X0007 and not equal so PC = 0x0016 + 0x000A = 0X0021.

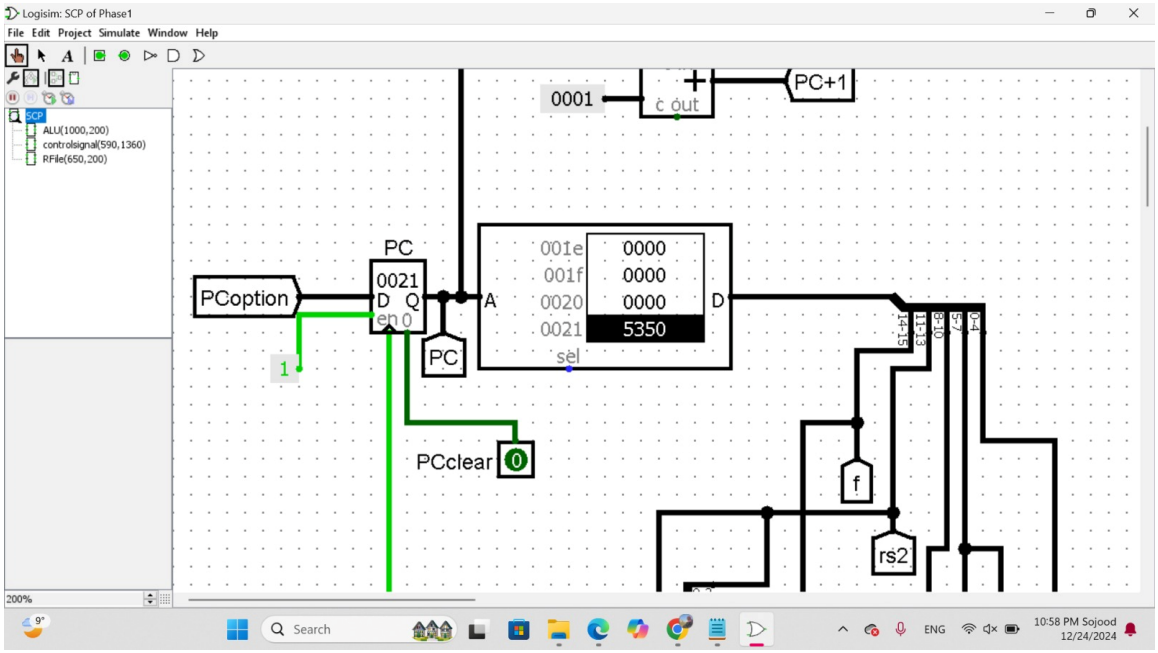


Figure 53: BNE test.

BLT 1, R2, R3, 2 if(R3 < R2) PC = PC + sign_extend{IMM2,IMM3} PC = 0000 0000 0010 0001 + 0000 0000 0000 1010 else PC = PC+1 here R3 = 0X0005 , R2 = 0X0007 and R3 less than R2 so PC = 0x0021 + 0x000A = 0X002B.

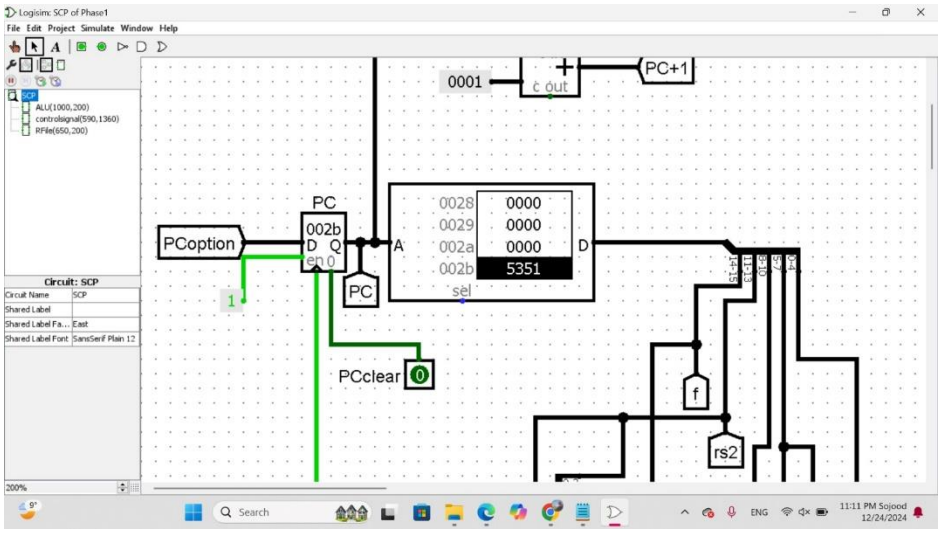


Figure 54: BLT test.

BGE 1,R2,R3,2 if(R3 >= R2) PC = PC + sign_extend{IMM2,IMM3} PC = 0000 0000 0010 1011 + 0000 0000 0000 1010 else PC = PC+1 here R3 = 0X0005 , R2 = 0X0007 and R3 not grater or equal R2 so PC = 0x002B + 0x0001 = 0X002C.

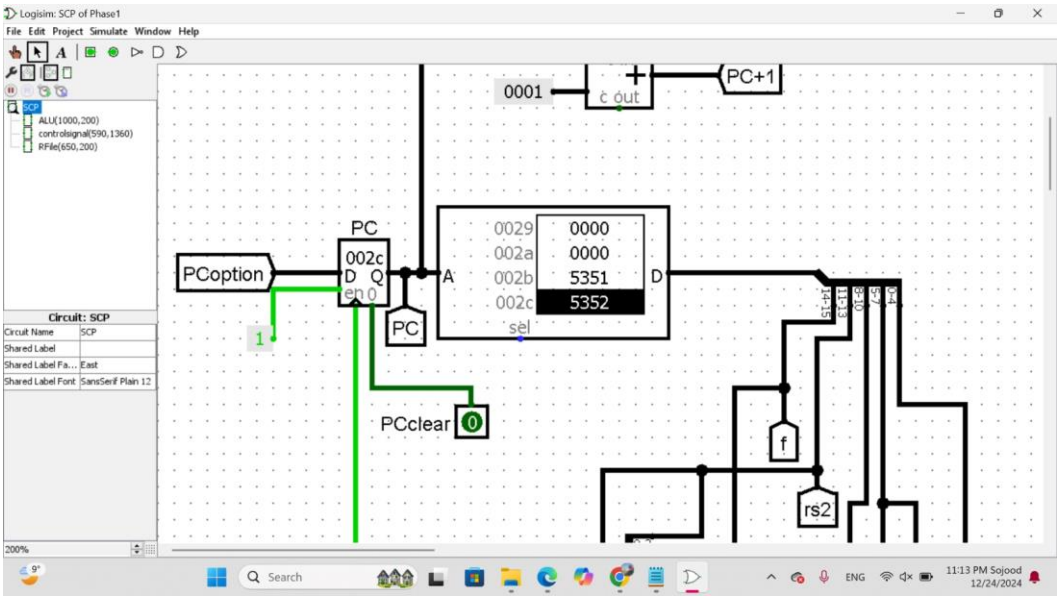


Figure 55: BGE test.

BLTU 1, R2, R3, 2 if (R3 < R2 (signed)) PC = PC + sign_extend{IMM2,IMM3} PC = 0000 0000 0010 1100 + 0000 0000 0000 1010 else PC = PC+1 here R3 = 0x0005 , R2 = 0x0007 and R3 less than R2 so PC = 0x002C + 0x000A = 0x0036.

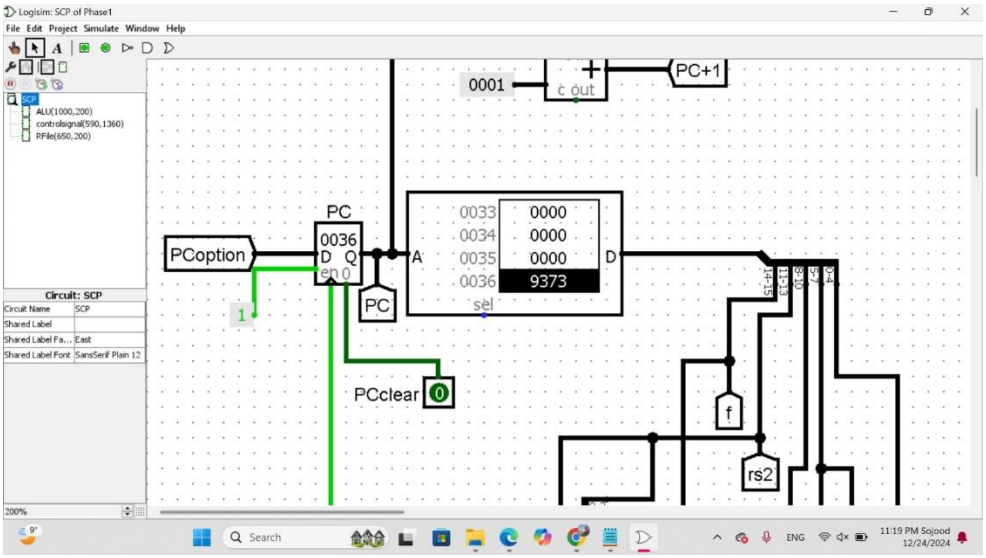


Figure 56: BLTU test.

BGEU 2, R2, R3, 3 if (R3 >= R2 (signed)) PC = PC + sign_extend{IMM2,IMM3} PC = 0000 0000 0011 0110 + 1111 1111 1111 0011 else PC = PC+1 here R3 = 0x 0005 , R2 = 0X0007 and R3 not equal or greater than R2 so PC = 0x0036 + 0x0001 = 0x0037.

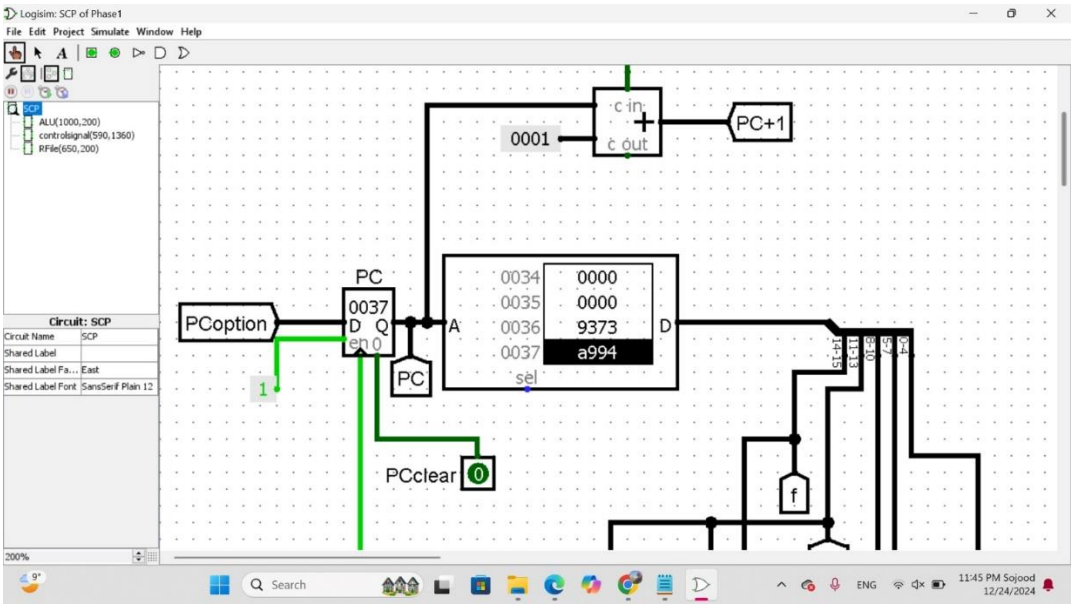


Figure 57: BGEU test.

Table 9: J-type test.

Address of instruction	Assembly	Binary	Hexadecimal
0x 0037	LUI 1356	1010 1001 1001 0100	0x A994
0x 0038	J 1	0000 0000 0011 0101	0x 0035
0x 0039	JAL 2	0000 0000 0101 0110	0x 0056
0x 003C	JALR 2,R4,R6	0001 0100 1101 0111	0x14D7

LUI 1356 loads the value 1356 << 5 into R1, which equals 0000 0101 0100 1100 in binary and then it will take the upper value of it and fill the lowest with zero >> 0000 0101 0100 0000 >> 0540 and link this value to R1.

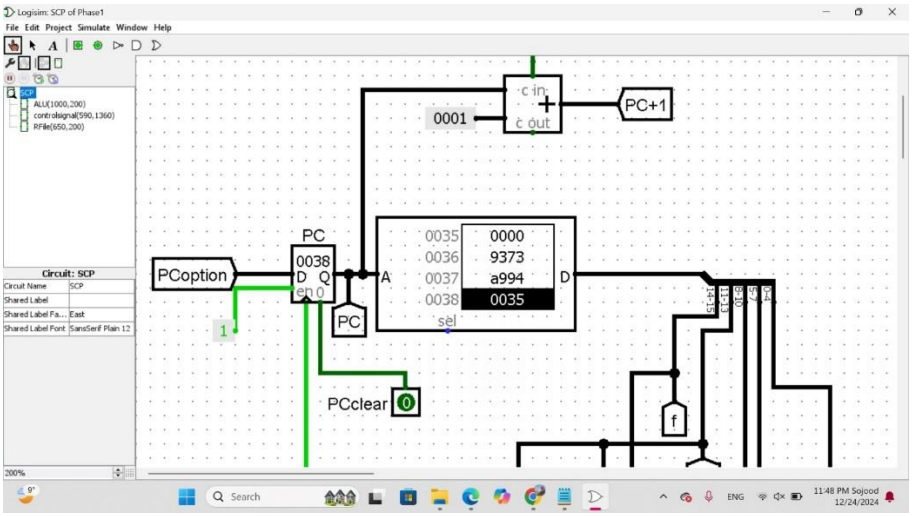


Figure 58: LUI Test 1.

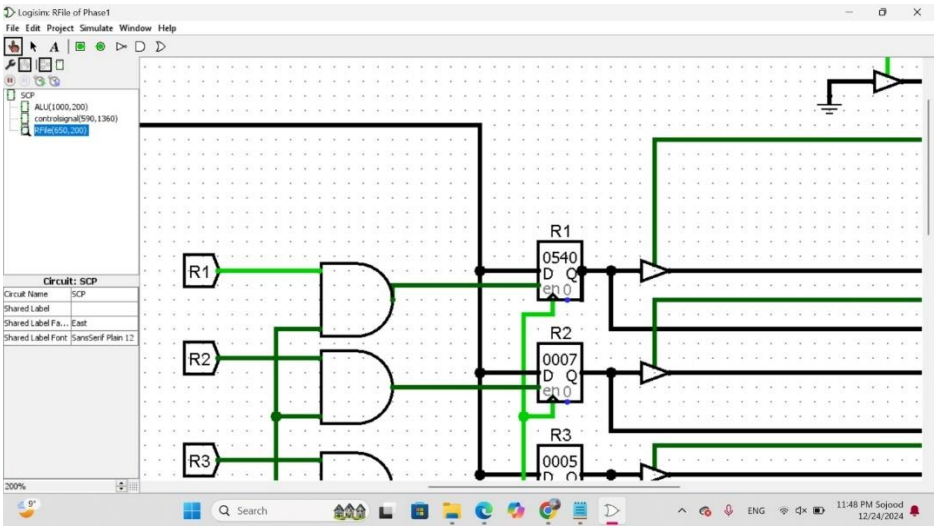


Figure 59: LUI Test 2.

PC = PC + sign_extend(imm11) >> PC = 0x0038 + 0x0001 = 0x0039

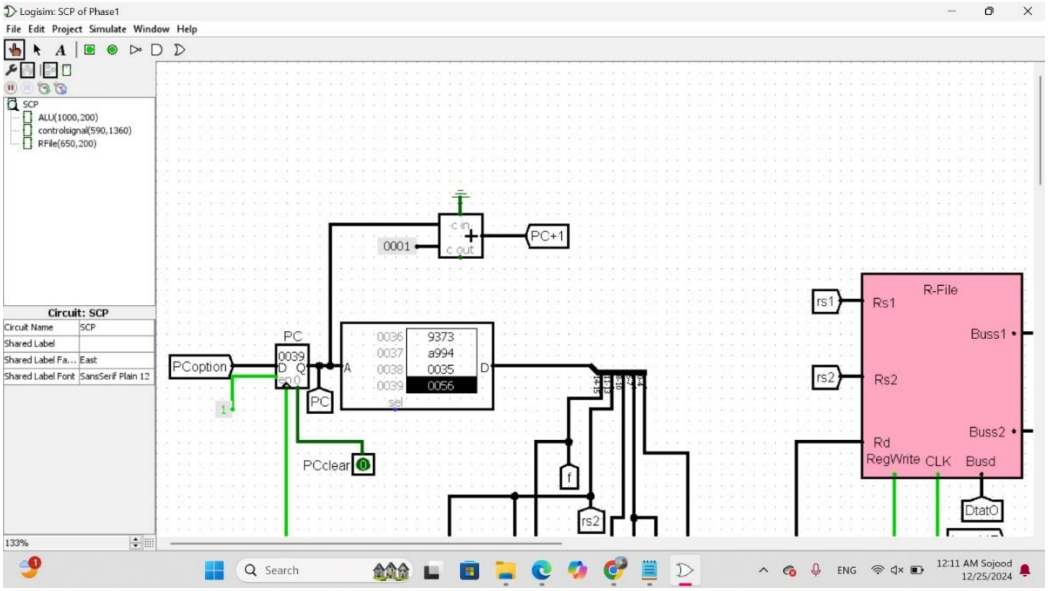


Figure 60: J test.

JAL 2 >> PC= PC + sign_extend (imm11) PC = 0x0039 + 0x0002 = 0x003B, and R7 will be PC+1 = 0X0039 = 0X003A

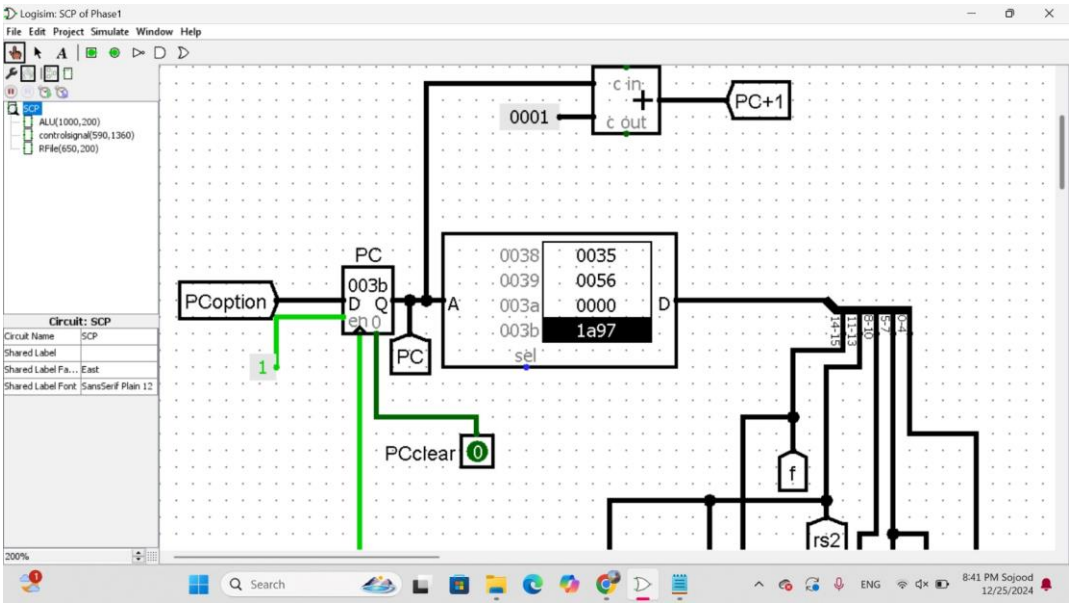


Figure 61: JAL test1.

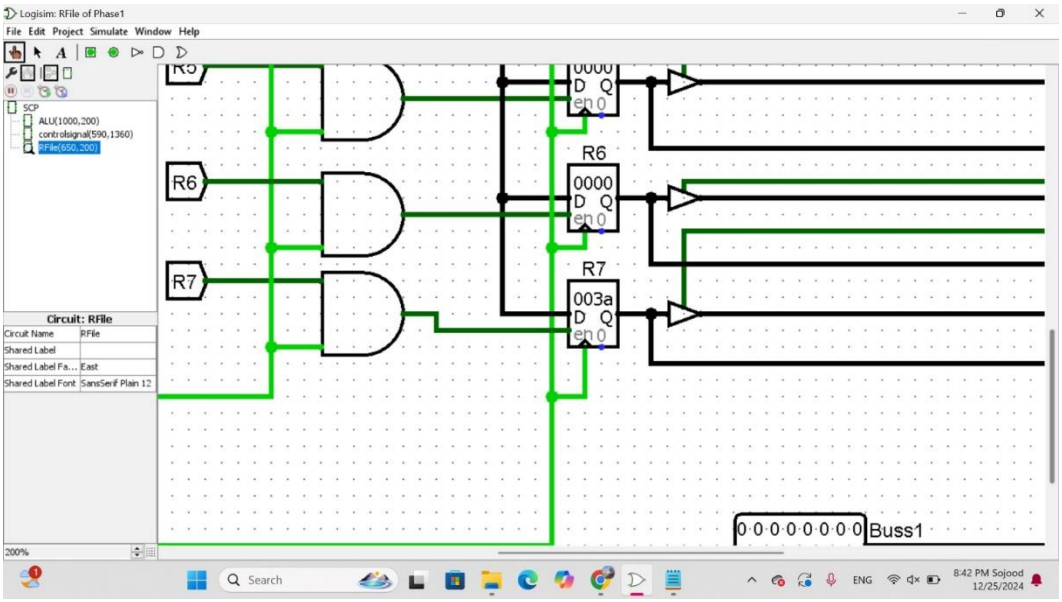


Figure 62: JAL test2.

LW 2,R0,R4 -> Load to R4 the content of the Address R0 + (00010), the content of R0 is 0 so Load the Data at address 0x 00002 to R4, so the content of R4 will be 0x 003B

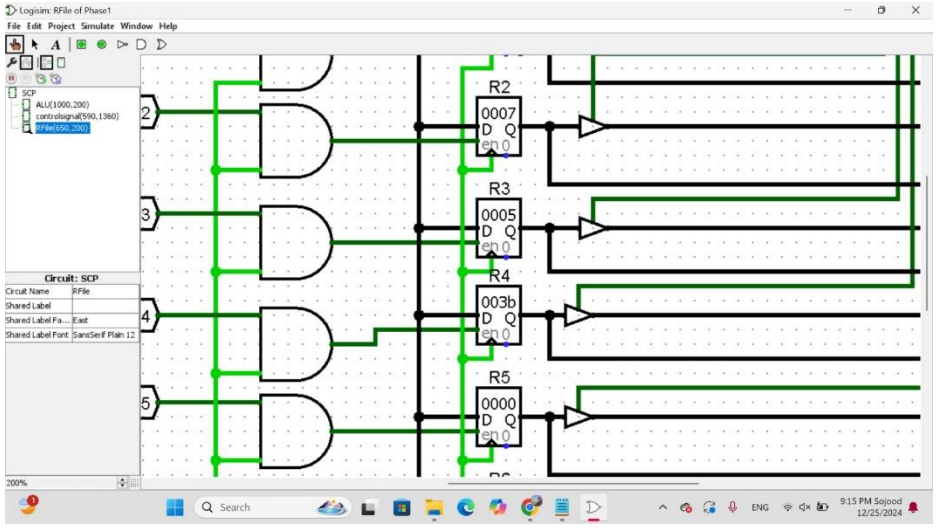


Figure 63: Value of R4.

JALR PC= Ra + sign_extend (imm5), Rd=PC+1 >> R6 = PC+1 = 0X003C + 1 = 0X003D and PC = R4 + 0000 0000 0000 0010 = 0X003B + 0X0002 = 0X003D.

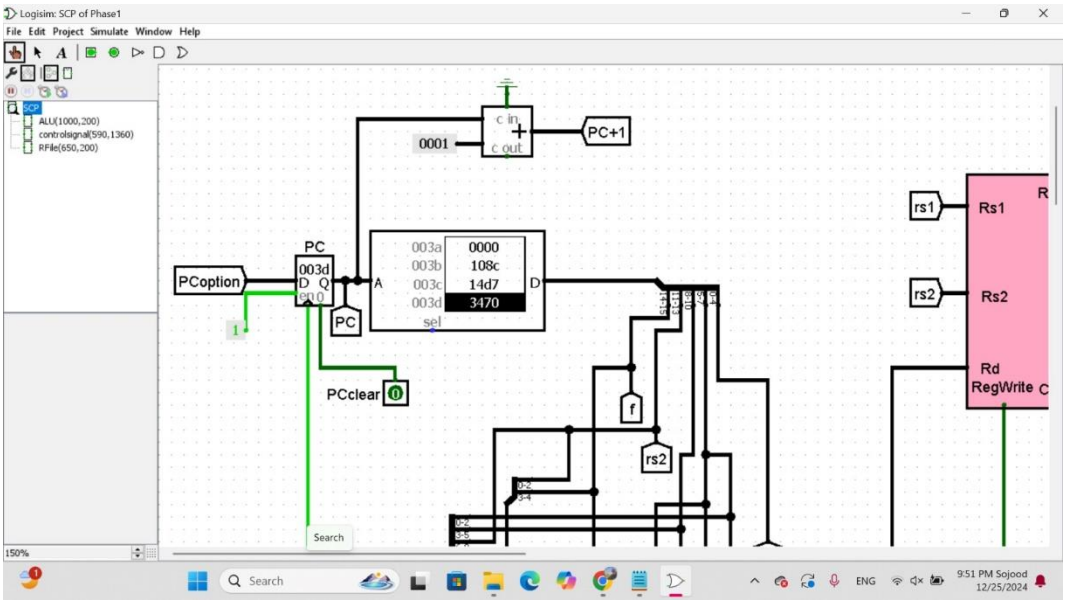


Figure 64: JALR test1.

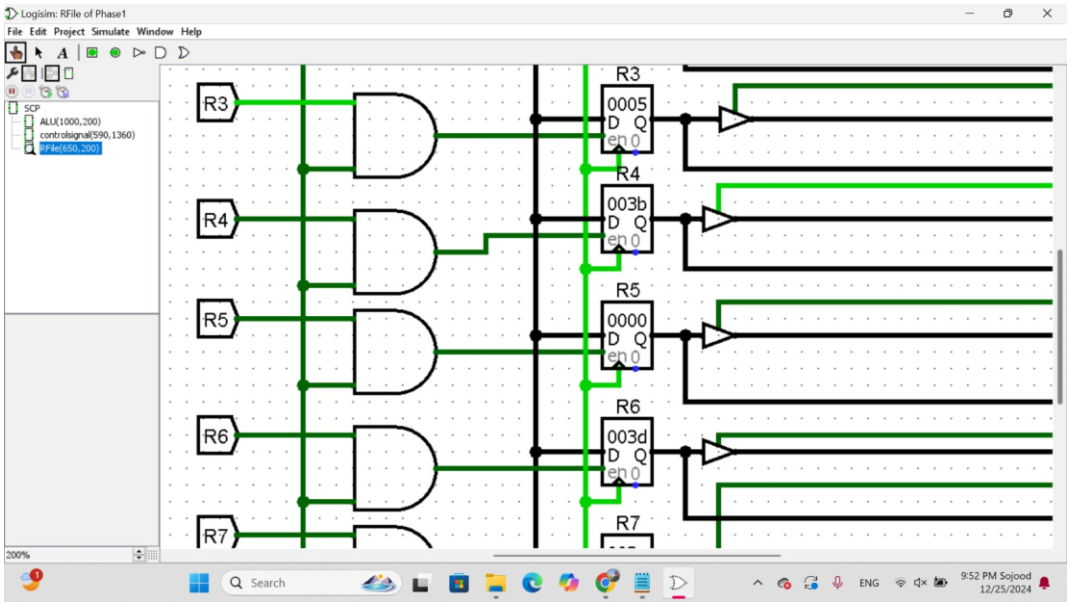


Figure 65: JALR test2.

Comparison / Conditional Test

Simple if-else Statement Test (check Branch and Jump instructions)

if (R6 > R4)

R4 = R6 + 4

else

R6 = R4 AND 5

}

0x 003D: BLT 0, R6, R4, 3 >> 0011 0100 0111 0000 >> 0x3470

0x 003E: ANDI 5, R4, R6 >> 0010 1100 1100 1000 >> 0X2CC8

0x 003F: J 2 >> 0000 0000 0101 0101 >> 0X0055

0x 0040: ADDI 4, R6, R4 >> 0010 0110 1000 0011 >> 0X2683

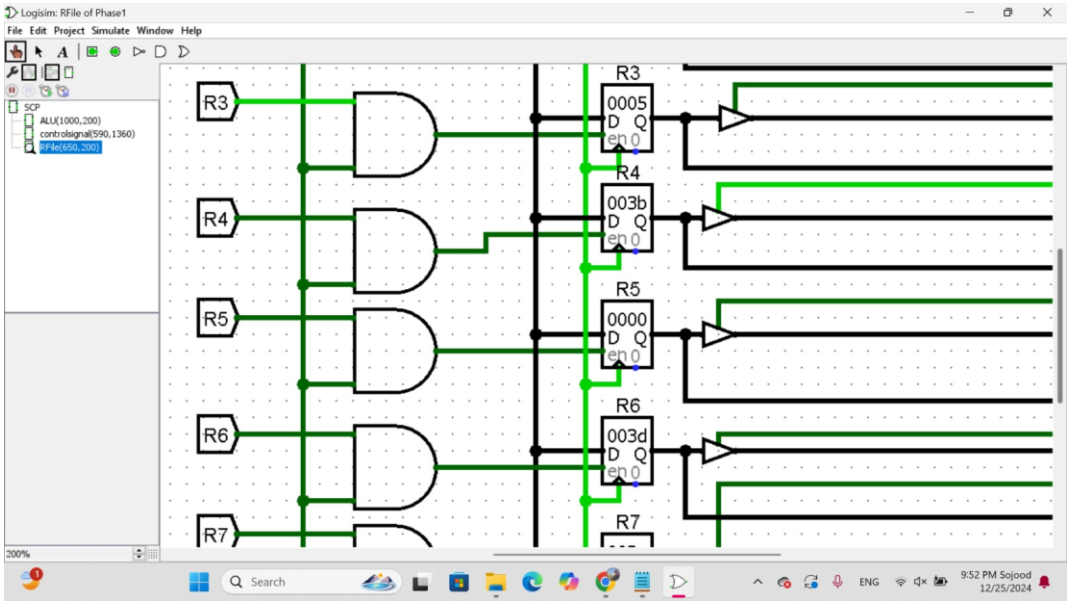


Figure 66: Values of R4 & R6.

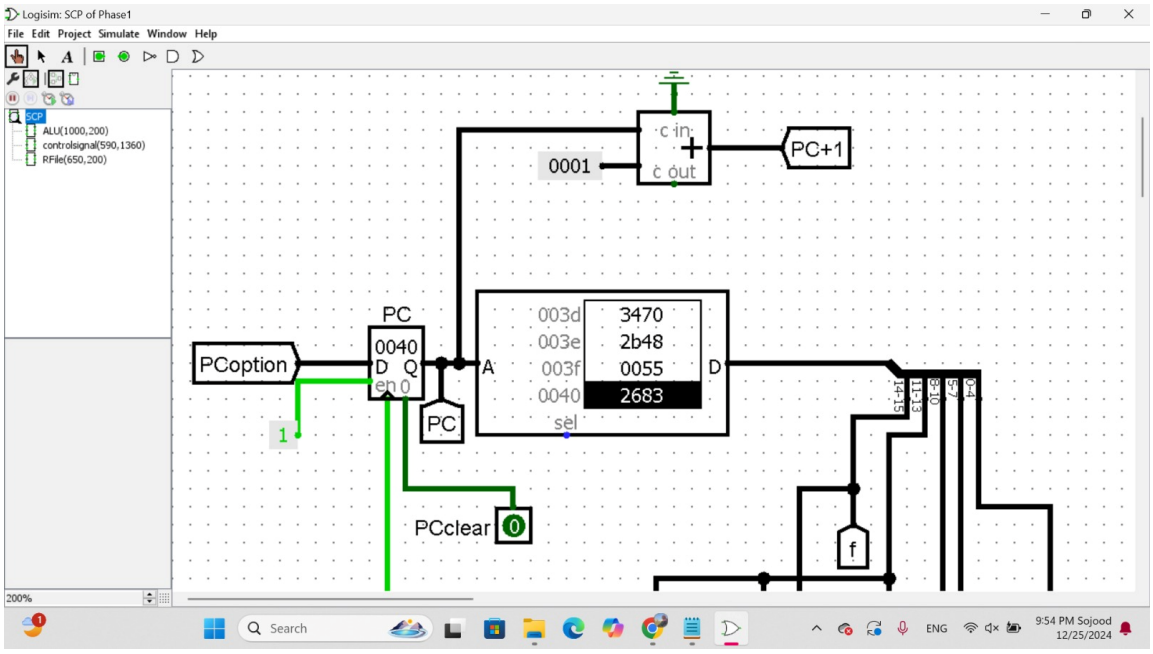


Figure 67: BLT test.

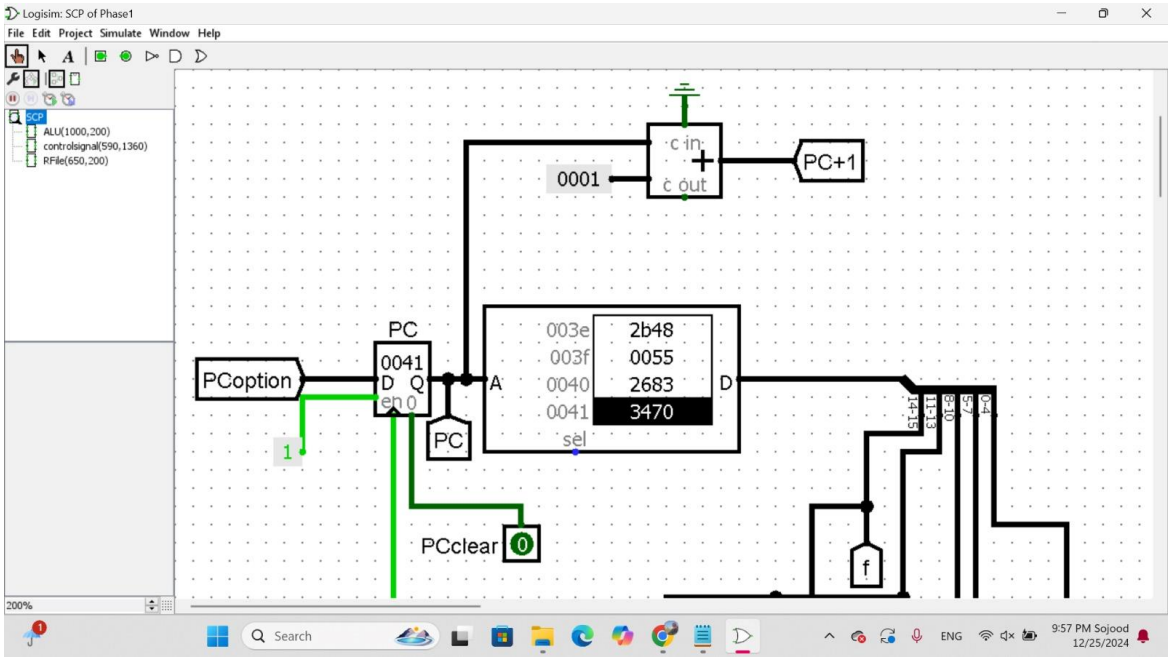


Figure 68: ADDI.

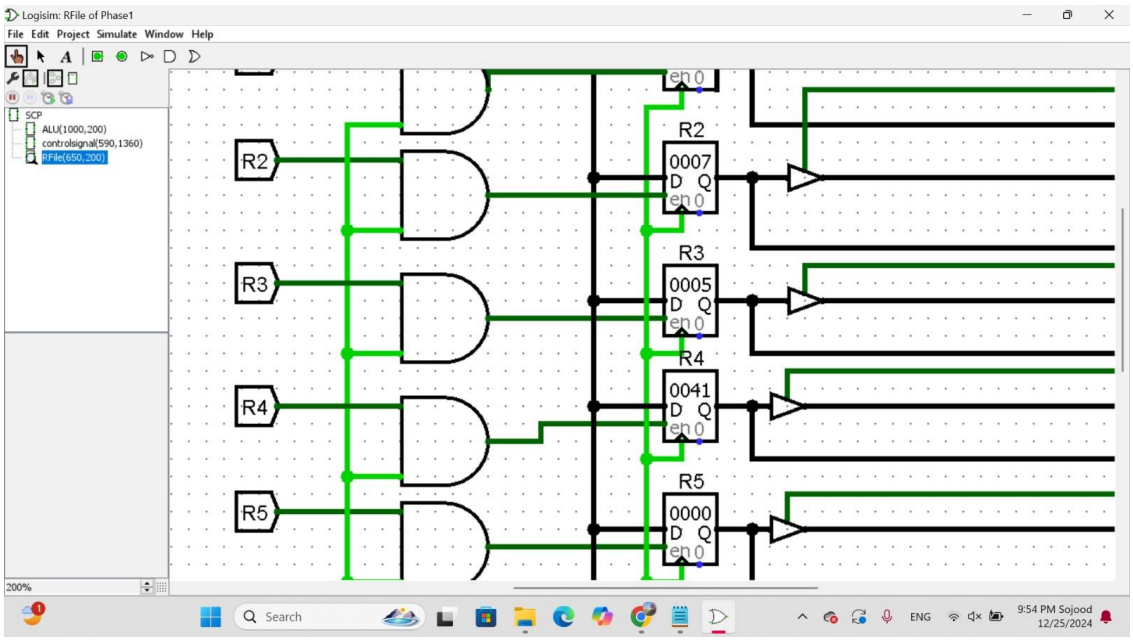


Figure 69: Value of R4 (R4 = R6 +4).

Now R4 = 0x0041, R6 = 0x003D, so we retested the conditional test:

If (R6 > R4)

R4 = R6 + 4

else

R6 = R4 AND 5

}

0x 0041: BLT 0, R6, R4, 3 >> 0011 0100 0111 0000 >> 0X3470

0x 0042: ANDI 5, R4, R6 >> 0010 1100 1100 1000 >> 0X2CC8

0x 0043: J 2 0000 0000 0101 0101 >> 0X0055

0x 0044: ADDI 4, R6, R4 >> 0010 0110 1000 0011 >> 0X2683

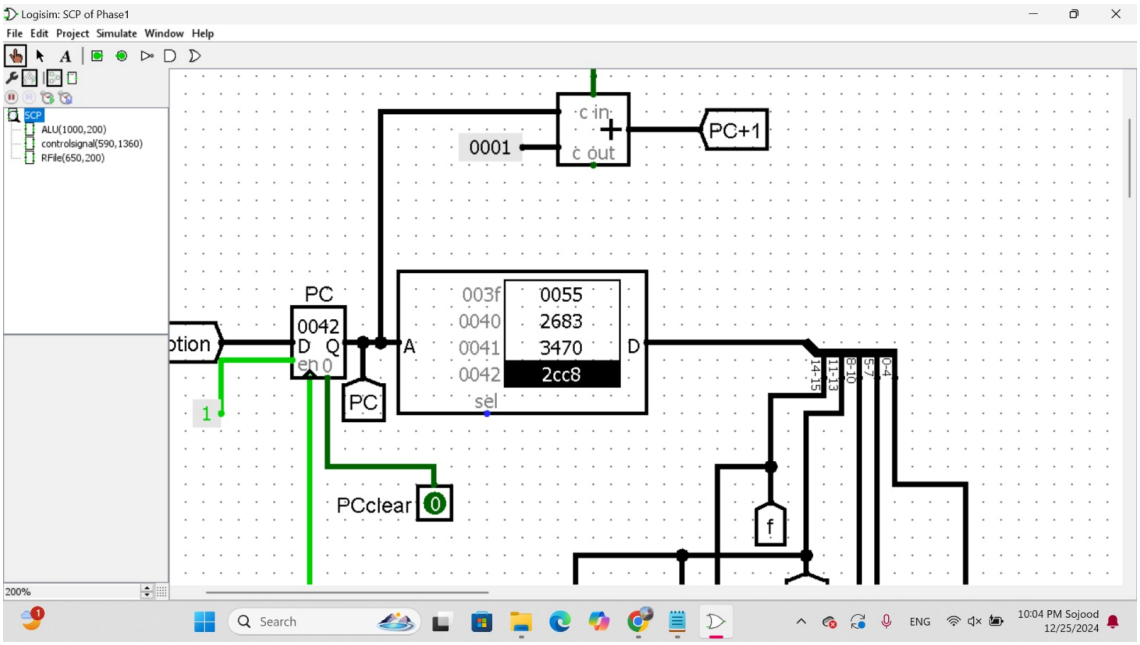


Figure 70: BLT.

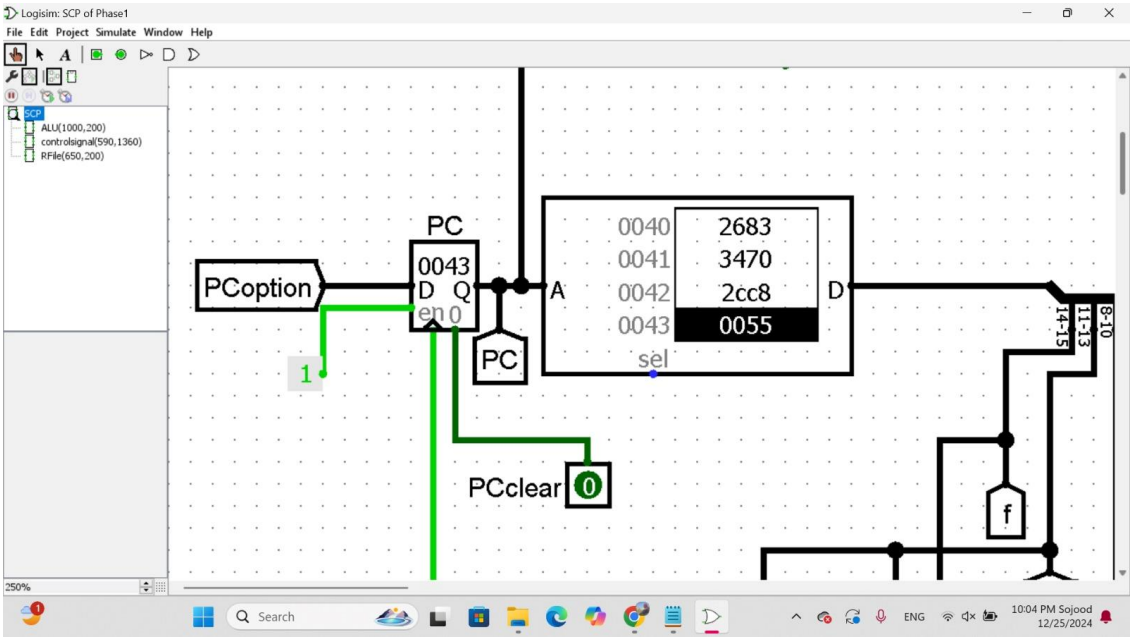


Figure 71: ANDI.

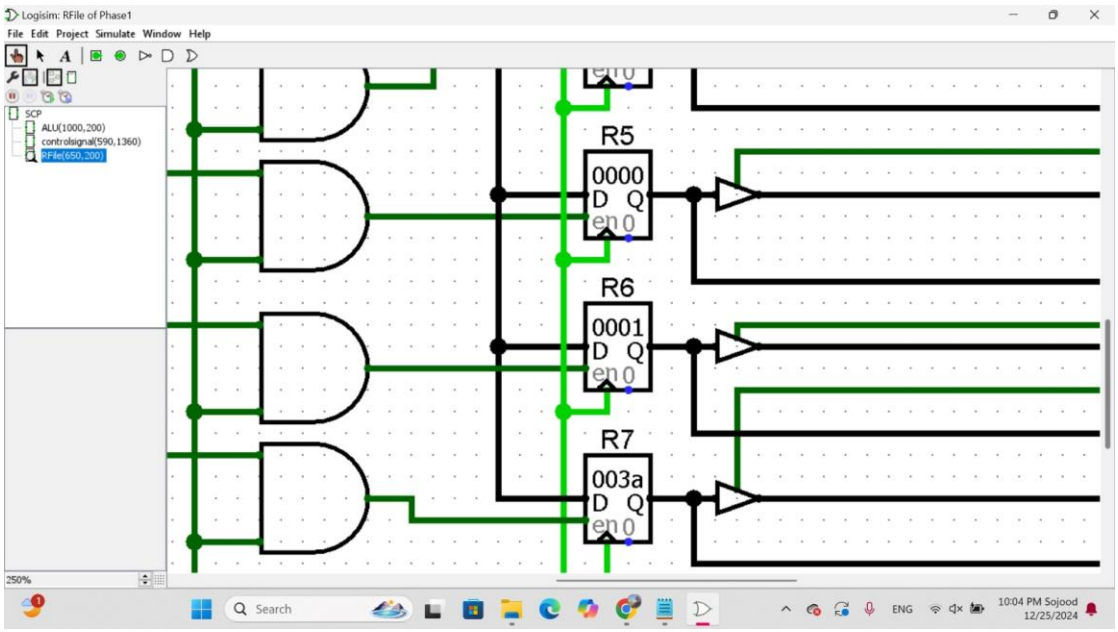


Figure 72: Value of R6 (R6 = R4 &5).

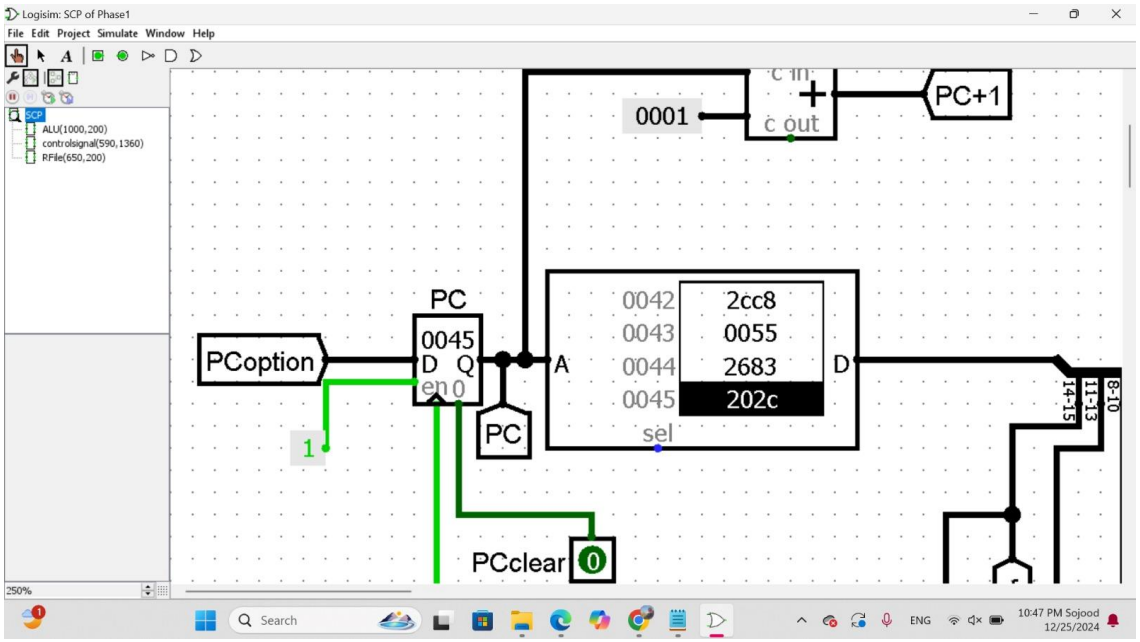


Figure 73: JUMP.

0x 0045: LW 4, R0, R1 >> 0010 0000 0010 1100 >> 0X202C

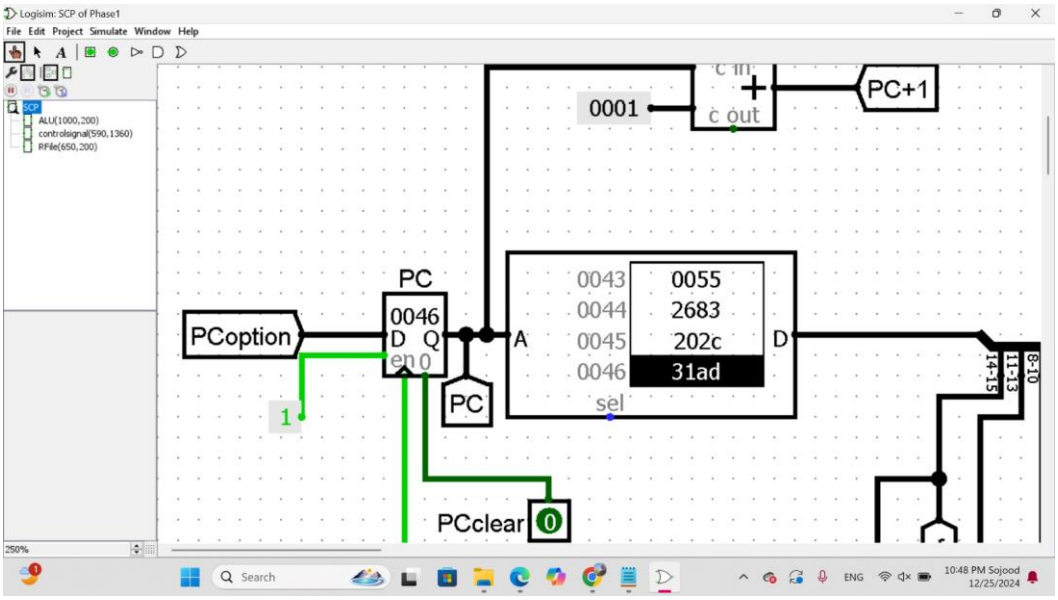


Figure 74: LW.

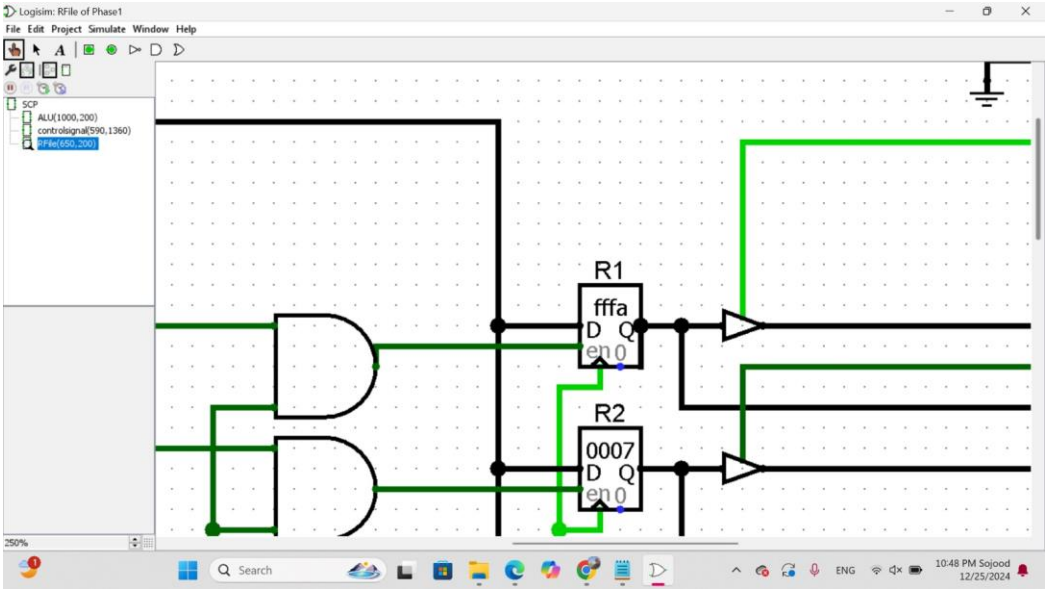


Figure 75: R1 after load.

0x 0046: SW 0, R6, R1, 5 >> 0011 0001 1010 1101 >> 0X31AD

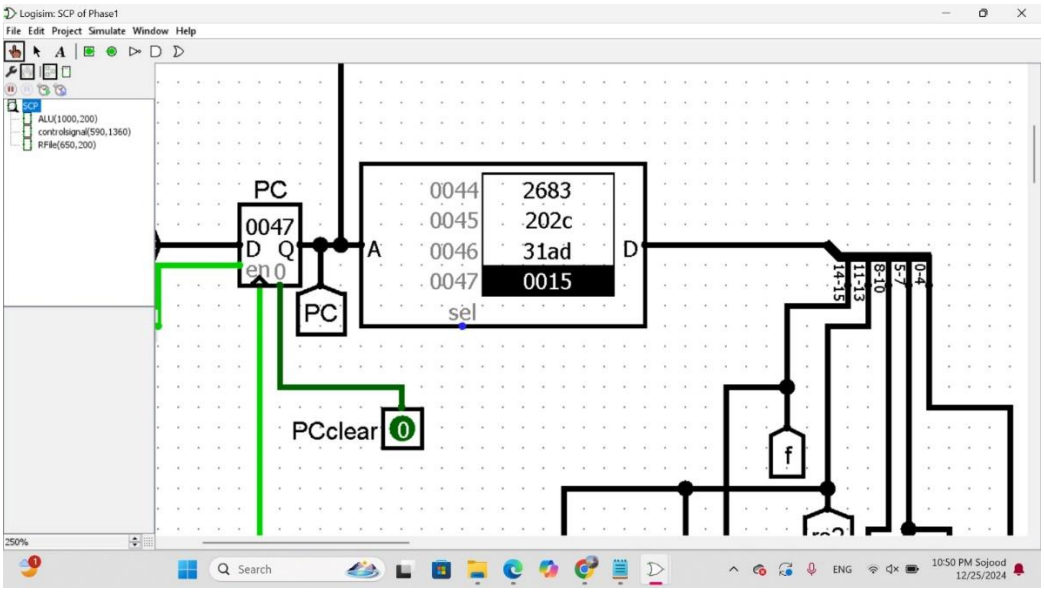


Figure 76: SW (R6 value to address 0xffff).

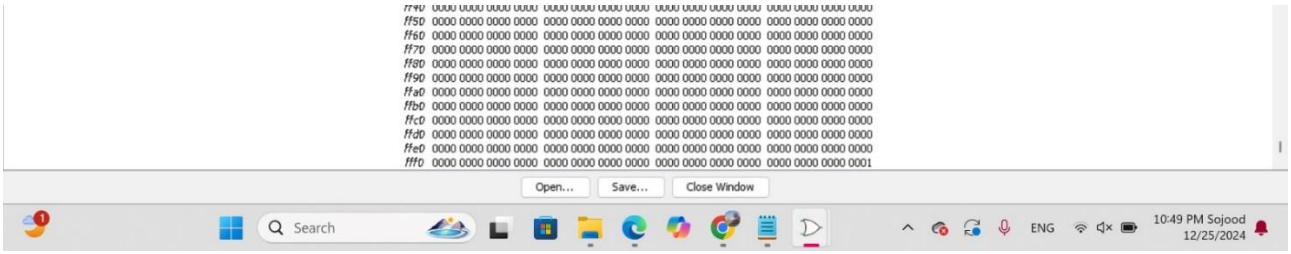


Figure 77: Address 0xffff in Memory.

The last instruction in the program can jump to itself indefinitely.

0x0047: J 0 >>0000 0000 0001 0101 >> 0X0015

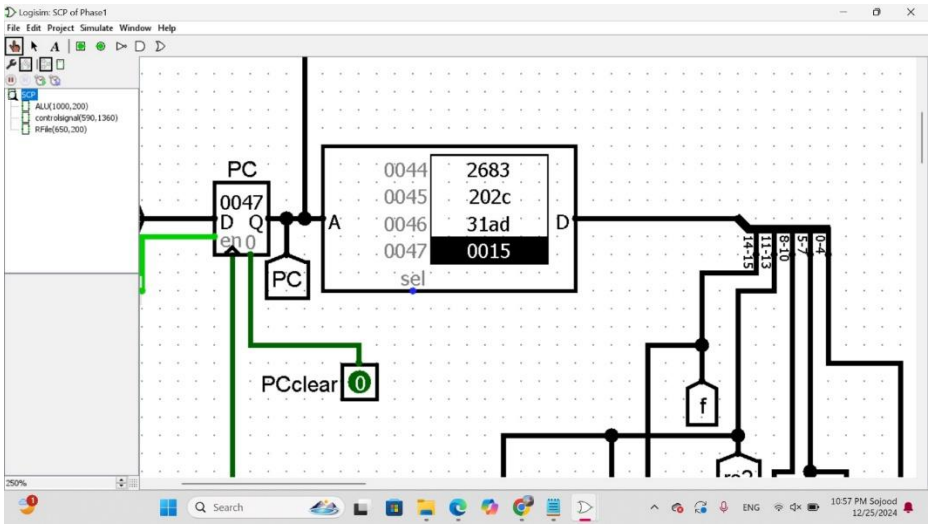


Figure 78: J0.

Design Alternatives, Issues and Limitations

Design Alternatives:

1- An alternative design for the program counter (PC) can be implemented using a decoder and multiplexers (MUXs). A decoder can be used to interpret control signals such as branching, jumping, or sequential execution based on the opcode or specific fields in the instruction. The outputs of the decoder can then generate control signals to activate the appropriate input of a MUX. The MUX selects between various inputs for the PC update, such as PC+1 for sequential execution, a branch target address for conditional branch instructions, or a jump address derived from imm11 or the contents of a register for jump instructions. This approach simplifies the logic, as the decoder handles instruction decoding while the MUX routes the correct value to the PC based on the operation being performed. This modular design ensures efficient control flow and reduces complexity in managing different PC updates.

2- The immediate values Imm11 and Imm5 can either be supplied separately from dedicated inputs or derived indirectly from registers such as rs1, rs2, or other sources within the circuit. This flexibility allows the immediate values to adapt to the specific needs of the instruction being executed.

3- A single ROM can be used to generate all control signals in the design by encoding the control signals as a control word corresponding to each instruction. The opcode and function bits (f) serve as the ROM's address input, with each unique combination mapping to a specific control word. This control word contains all necessary signals, such as RegDst, RegWrite, MemRead, MemWrite, and ALUOp Using a single ROM simplifies the control unit by centralizing the generation of control signals, reducing wiring complexity, and ensuring consistency across instructions.

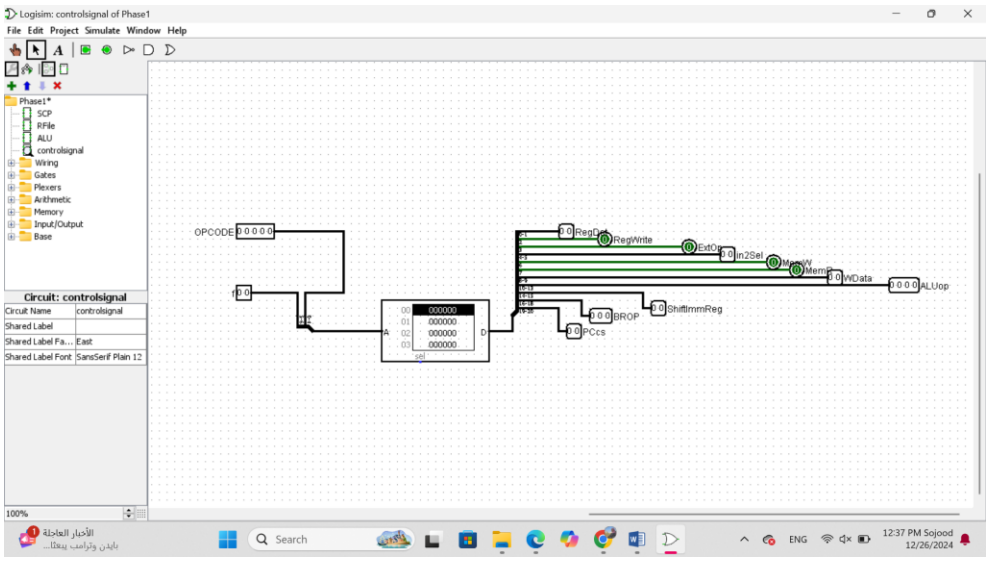


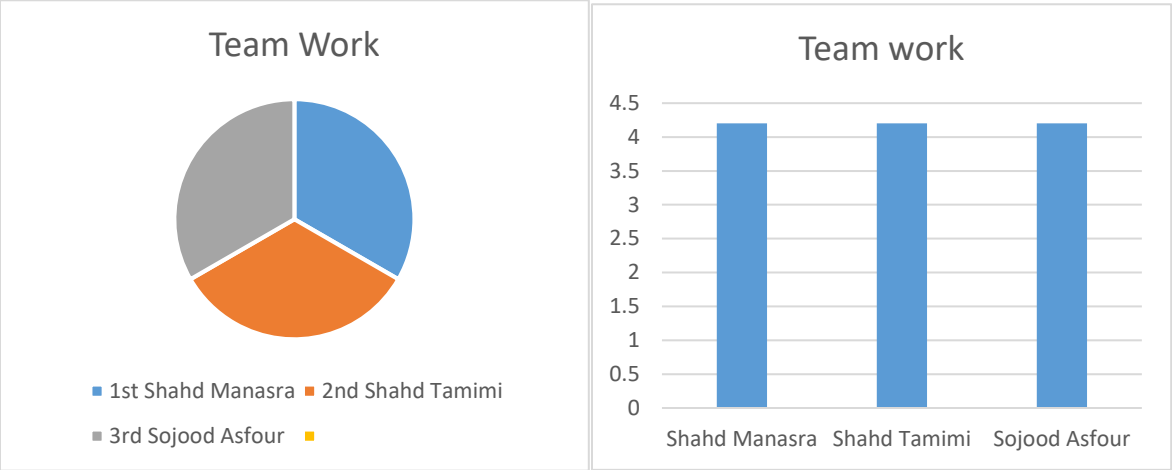
Figure 79: Alternatives design

Issues and Limitations

1- We used a control signal called ShiftImmReg (2 bits) to manage the shift immediate functionality, allowing us to input 1, 2, 3 and 4bits into sa. However, this design turned out to be incorrect. Since we had already progressed significantly in the work, we resolved the issue by defaulting the shift to always handle 4 bits, setting shiftImmReg = 00 in all cases to avoid any impact.

2- As for the second issue, we were initially taking imm11 in the ALU from input1 in most of the previous configurations. However, we corrected this and started taking it from input2.

Teamwork



We worked on the project collaboratively across all its parts without dividing tasks. This was because we worked together at the university and also held google meetings to continue working on the project.

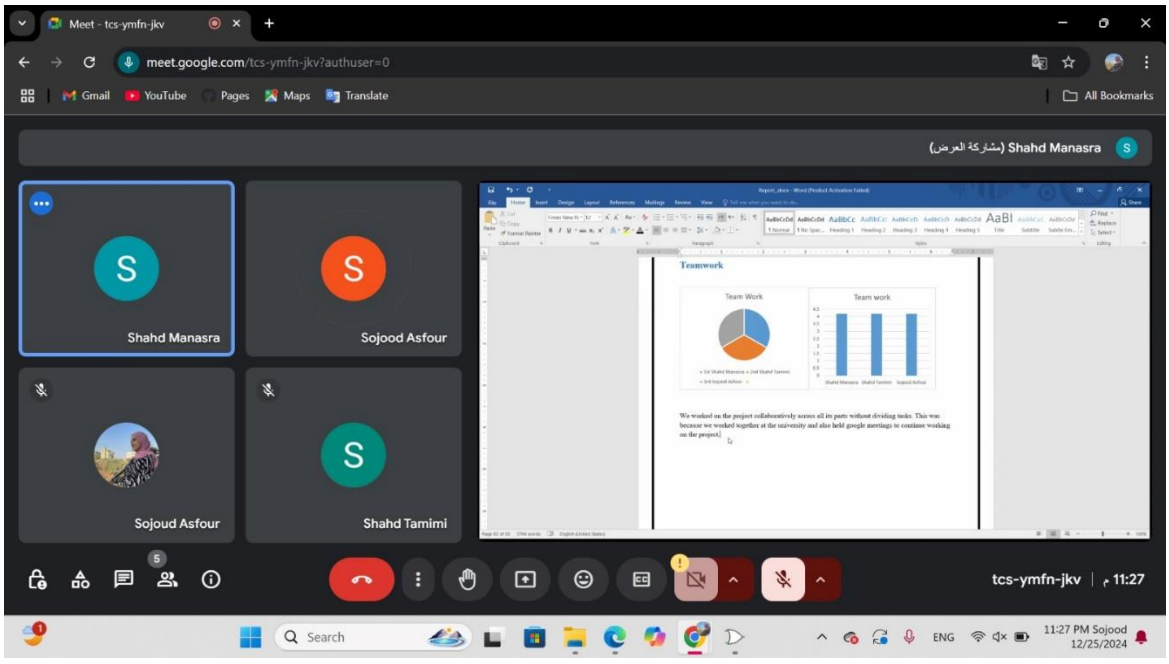


Figure 80: TeamWork.