Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Computer Organization and Microprocessor (ENCS2380)

Report

Project: Single Cycle Processor Design

Prepared by:

Raseel Jafar 1220724

Rand Saleh 1221124

Basmala Abuhakema 1220184

Instructor: Ismail Khater

Section: 1

Date: 11-6-2024

Abstract

In this project, we will design single cycle processor.

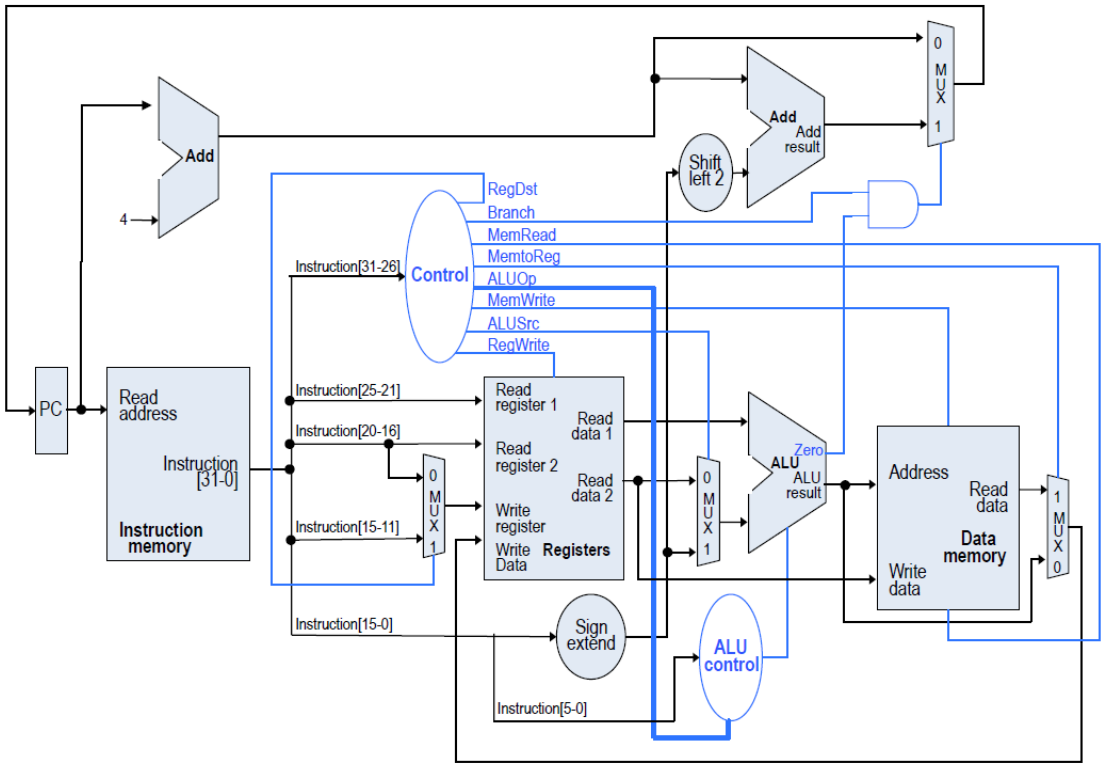


Figure 1: single cycle processor schematic

Table of Contents

Table of Figures

List of Tables

Register File

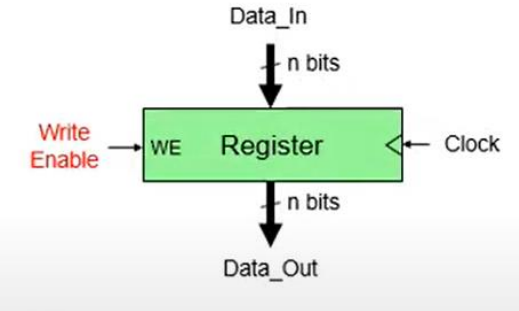


Figure 2: register element

A register file in a single-cycle processor is a critical component responsible for holding the processor's register set. It plays a crucial role in instruction execution, providing fast access to operand data for arithmetic and logical operations.

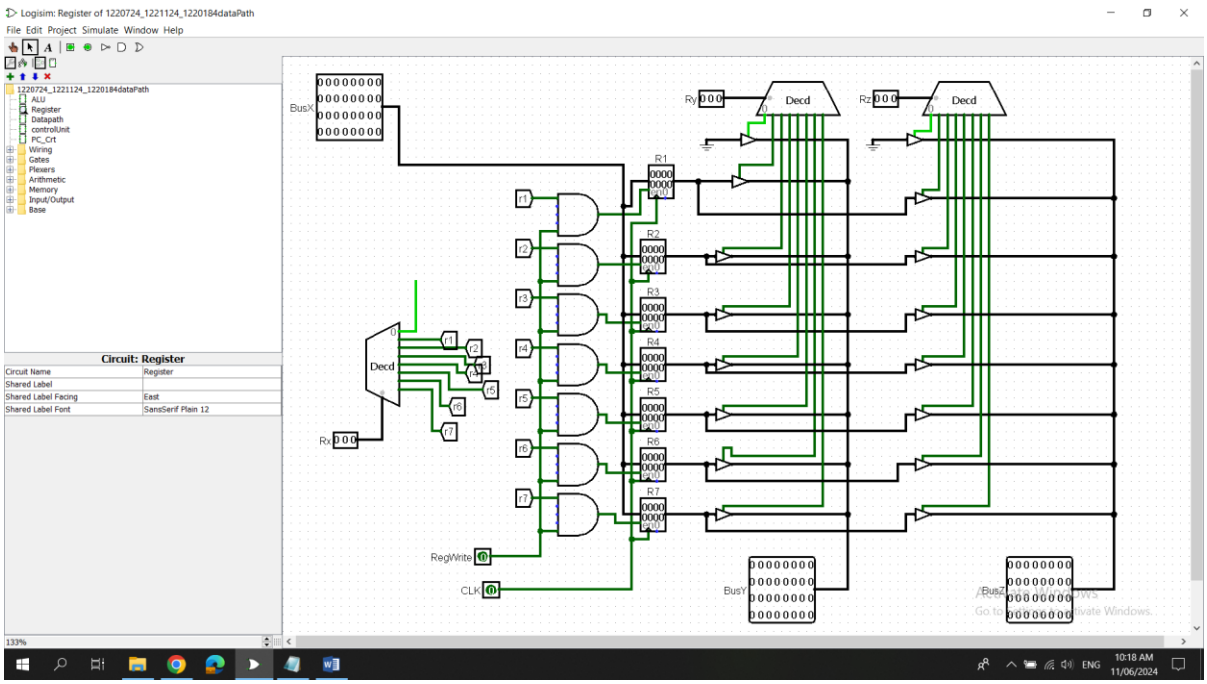


Figure 3: Register on logisim

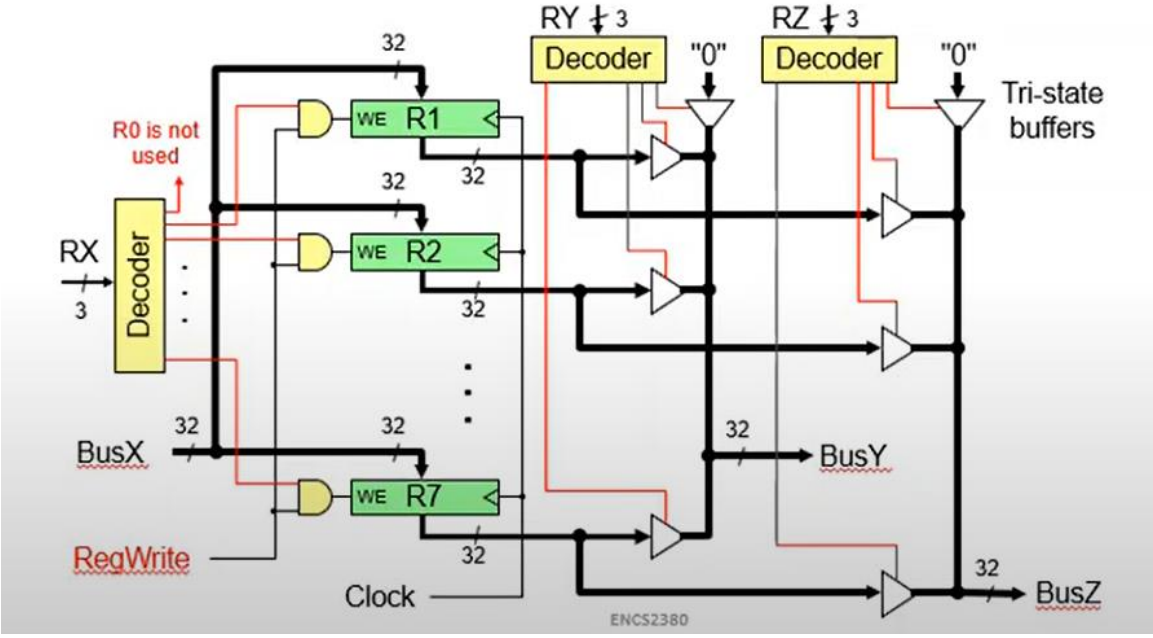


Figure 4: Circuit schematic of our register design

Register File consists of 7x32-bit registers

BusY and BusZ: 32-bit output busses for reading 2 registers.

BusX: 32-bit input bus for writing a register when RegWrite is 1.

Two registers read and one written in a cycle.

Registers are selected by:
Ry selects register to be read on BusY

Rz selects register to be read on BusZ

Rx selects the register to be written

Clock Input

The clock input is used ONLY during write operation.

During read, register file behaves as a combinational logic block
Ry or Rz valid => BusY or BusZ valid after access time

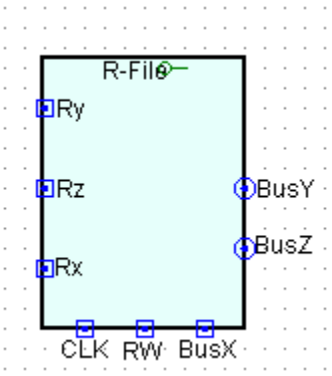


Figure 5: Register block on logisim

Testing the Register File:

Here is the final result for writing on R2 the value (4), and reading it.
we will discuss every step.

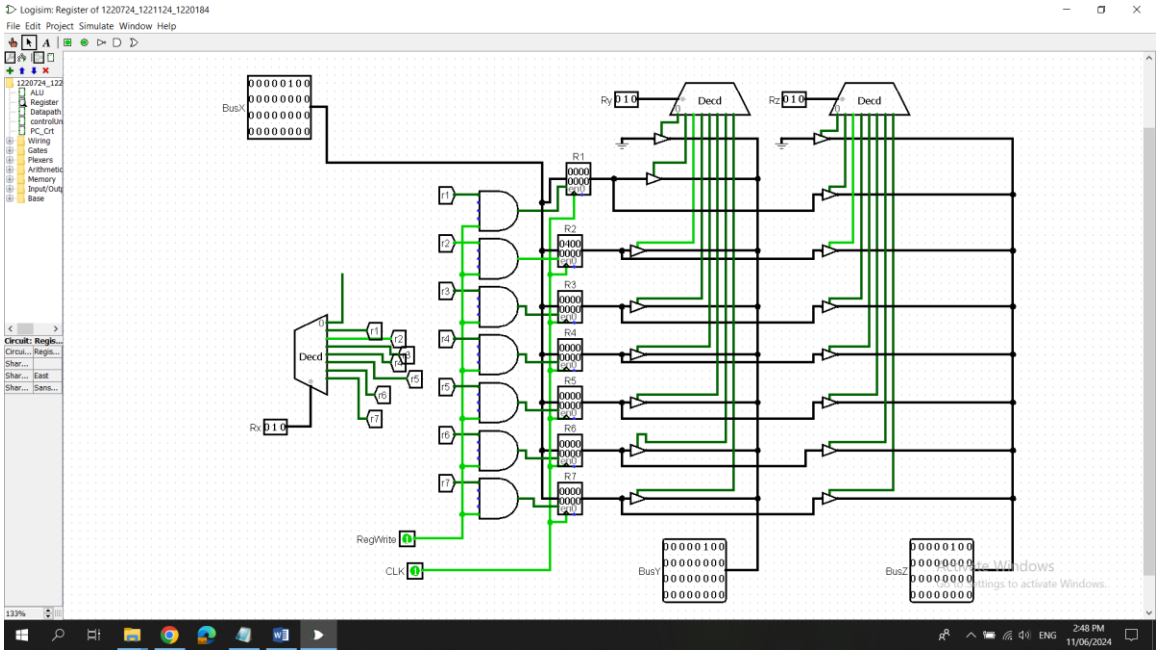


Figure 6: Testing the Register

First, we put the value we want to write it on the register on the BusX as shown below.

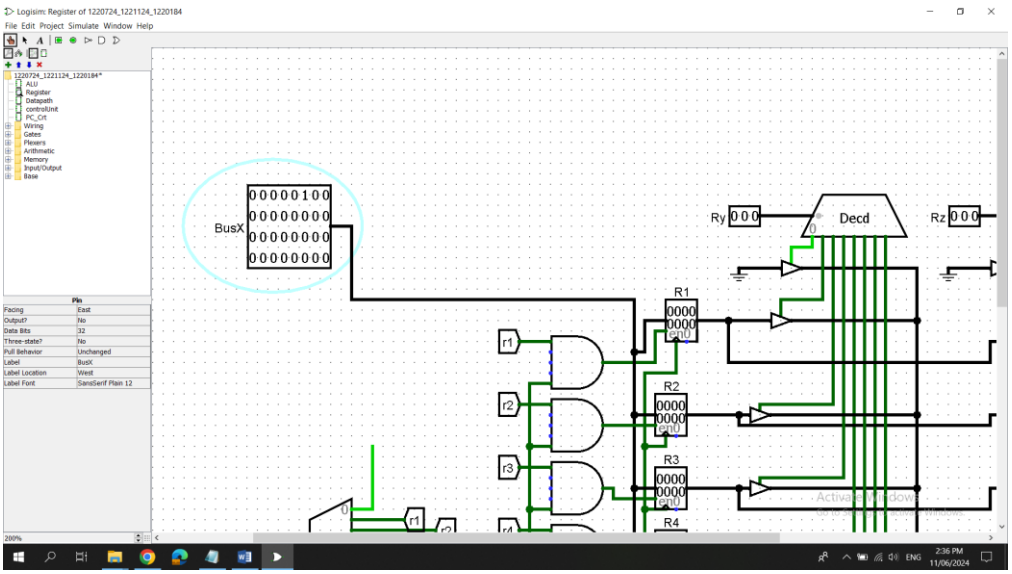


Figure 7: BusX for writing a register

Rx selects the register to be written, here we chose R2.

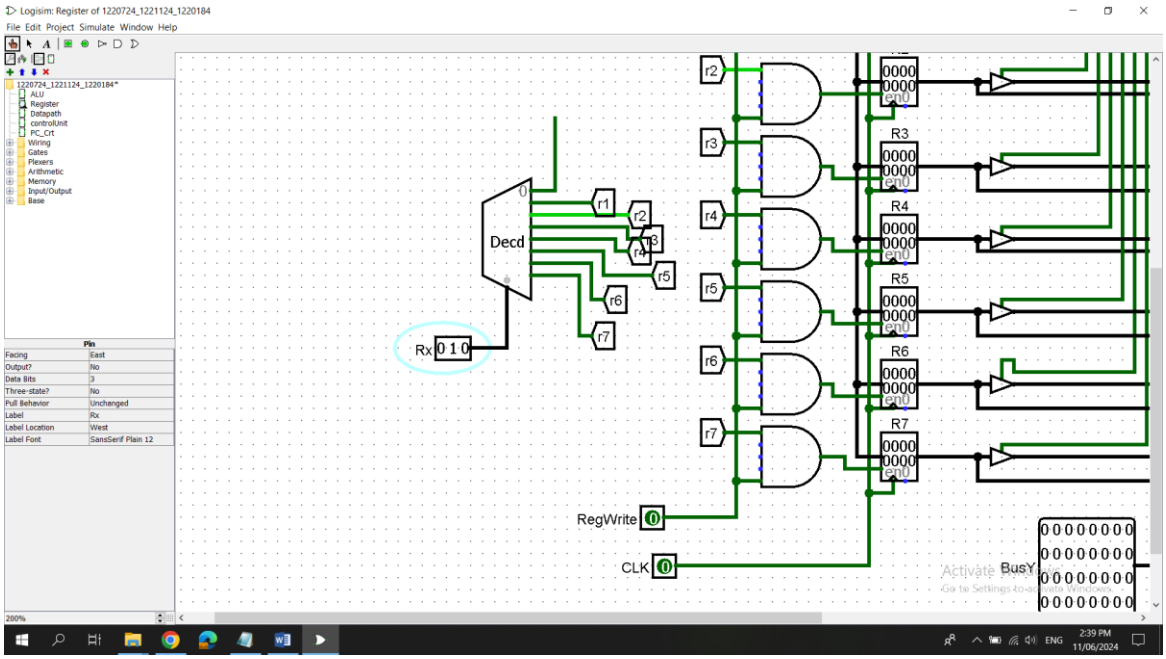


Figure 8: Rx

RegWrite enabled , and the clock.

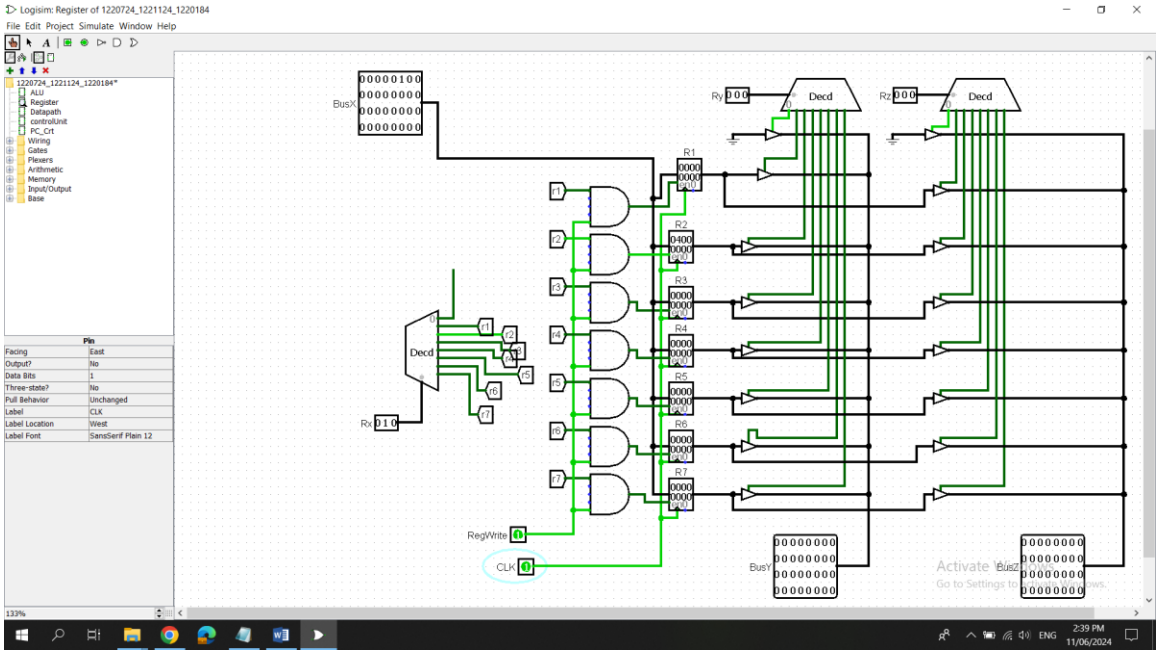


Figure 9: RegWrite/clock

Now, the value is written on R2 successfully as shown in the figure.

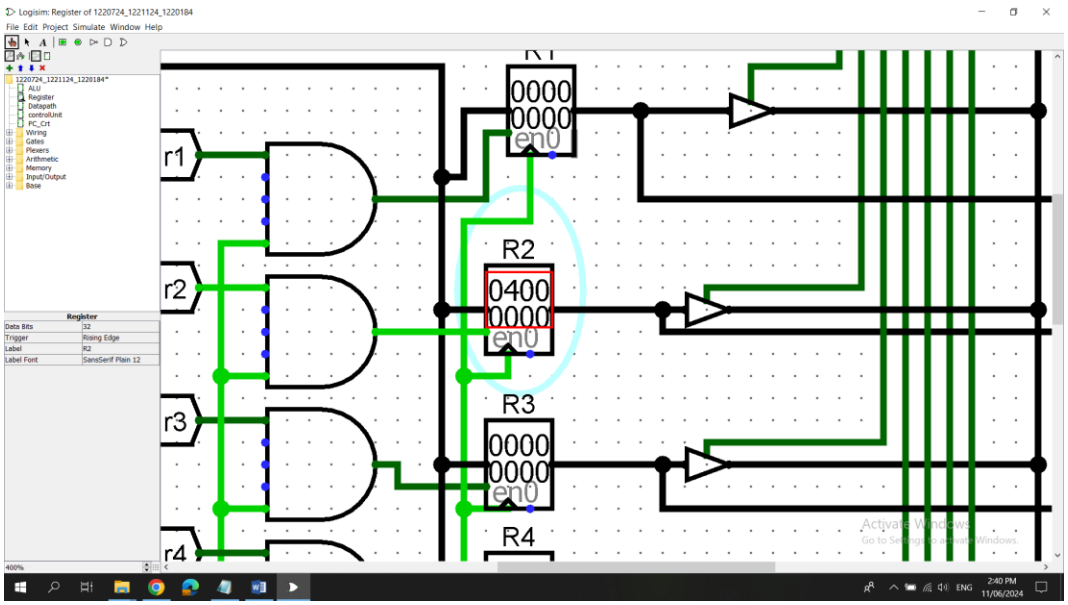


Figure 10: value written on the register

After writing the value on the register, we want to read it, we can choose to read it on BusY or BusZ, we will do both.

Ry selects register to be read on BusY, so in the figure we select R2.

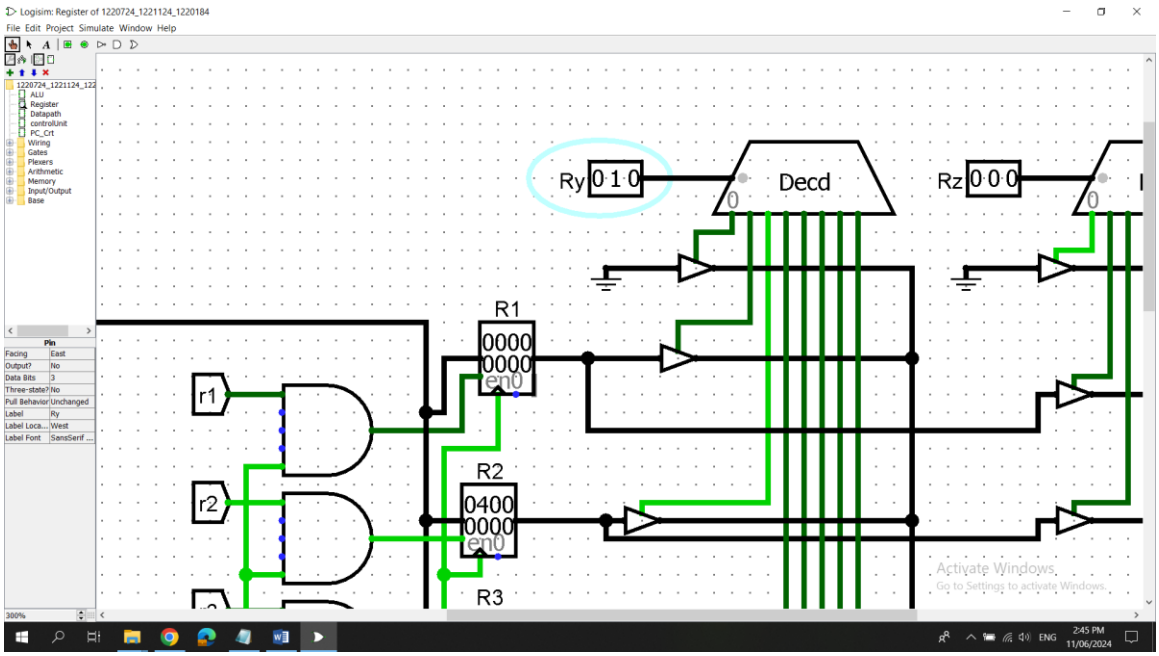


Figure 11:Ry selects register to be read on BusY

We did the same for Rz to read on BusZ.

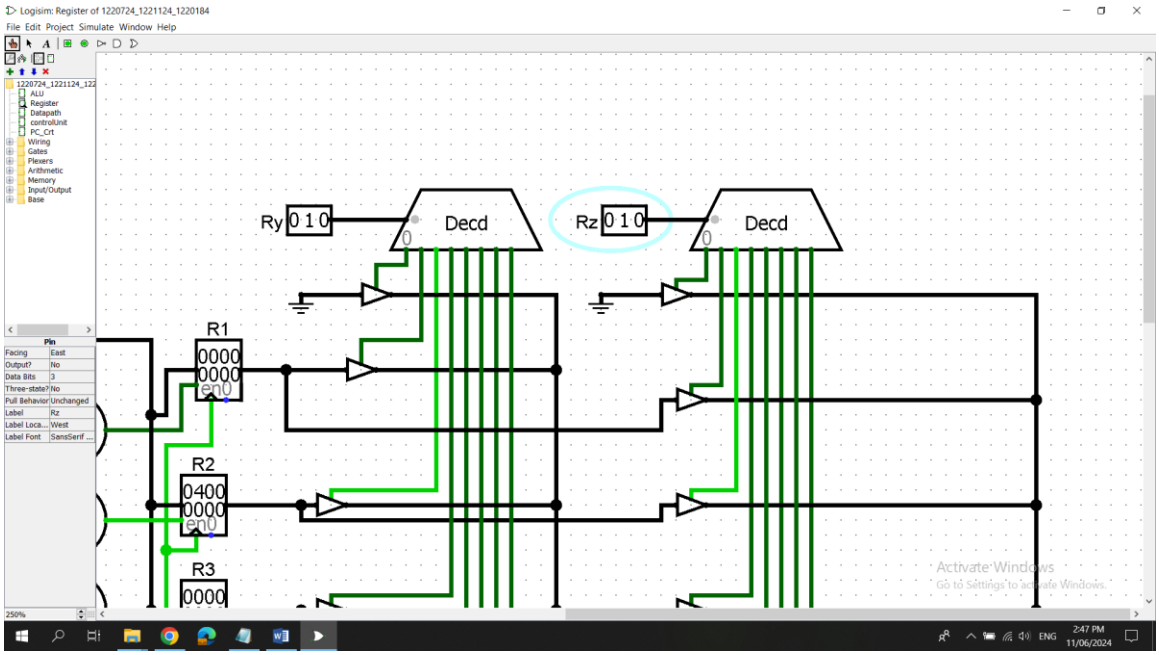


Figure 12:Rz selects register to be read on BusZ

The value read successfully!

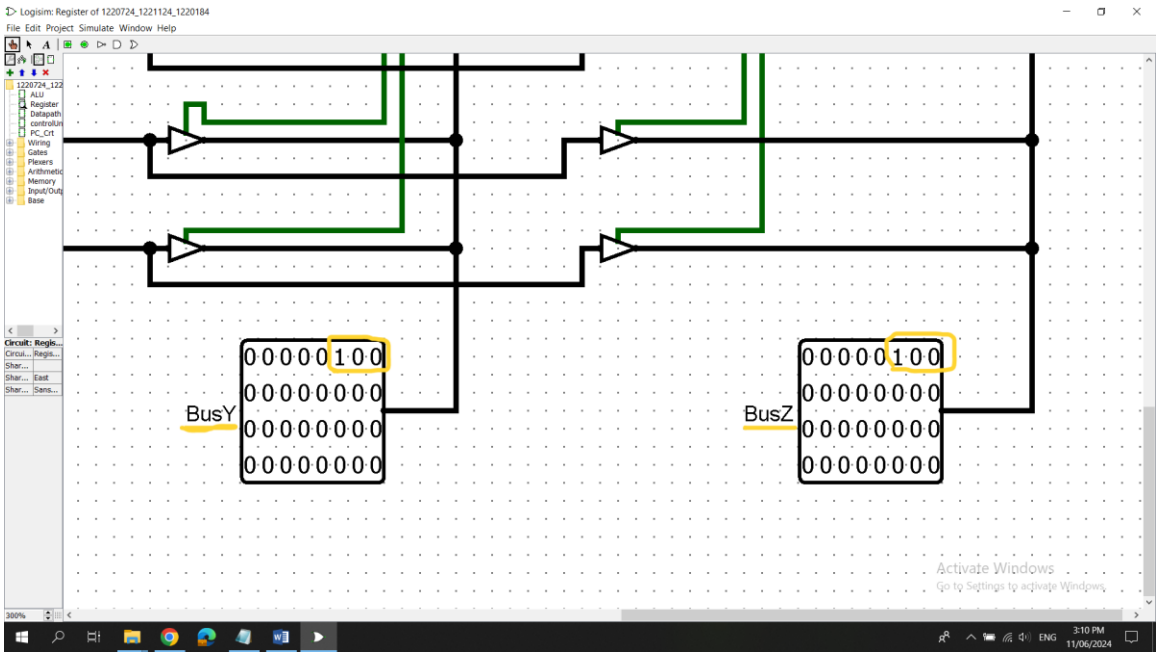


Figure 13: output busses for reading 2 registers.

Arithmetic-logic unit (ALU)

The Arithmetic -logic unit is the part of the Control processing unit (CPU) that plays a crucial role in excusing instructions in Single Cycle Processor ,it carries out arithmetic and logic operations on the operands provided to it . , Arithmetic operations like Addition, Subtraction ,Multiplication ,Division ,as well as Logical operations such as AND,OR,XOR, also we can execute Shift and Rotate operations within the ALU for example, Logical Shift Left/Right, Arithmetic Shift Left/Right,Rotate left/right. The ALU is going to execute these operations within one clock cycle which means it ensures that each instruction is processed and done in a single cycle and that affects the performance of the computer.

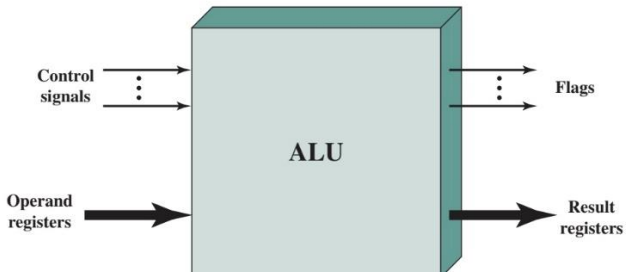


Figure 14: Arithmetic logic unit

In our project we designed a 32-bit ALU,this ALU consisting of 16 to 1 Multiplexer each input of the multiplexer implementing one instruction , and the ALU 4-bit selection line to select which instruction is going to be executed.

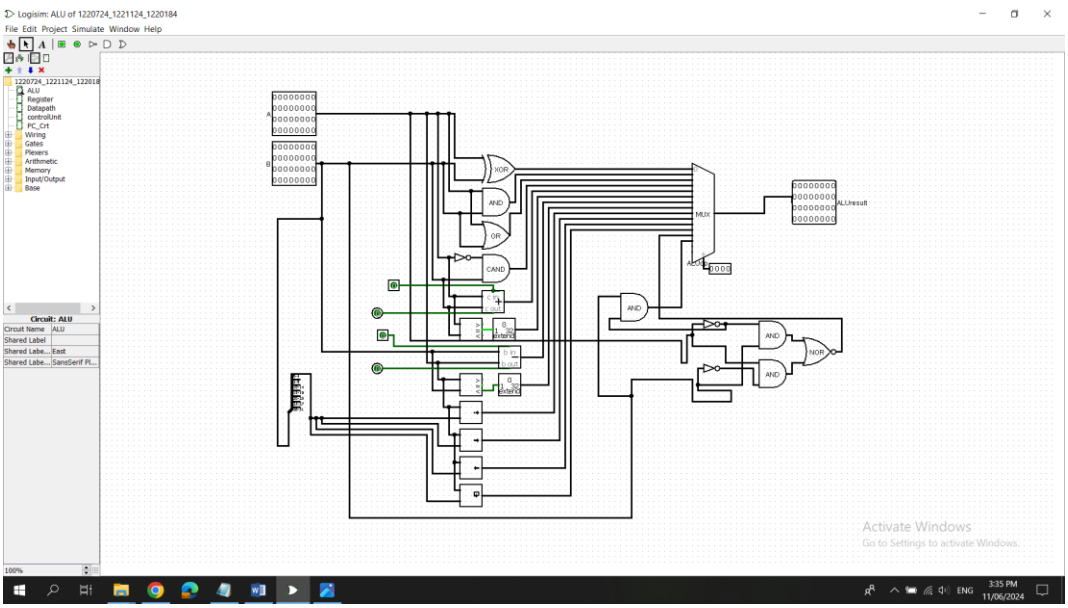


Figure 15: ALU circuit on logisim

Testing the ALU:

In our ALU testing we took the value of A as (01000010) and B as (00110110) , for the operations having opcode from (0000) to (0111).

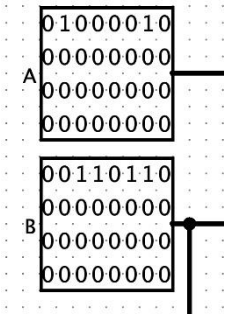


Figure 16: values of A & B in the ALU

Note here that the ALU has a 4-bit control line called ALU operation. The first two bits indicate whether a and b need to be inverted, respectively. The last two bits indicate the operation.

Remember subtract is implemented as add (thus the inversion).

ALU control line	Function
0000	XOR
0001	AND
0010	OR
0011	CAND
0100	ADD
0101	SEQ
0110	NADD
0111	SLT
1000	SRA
1001	SRL
1010	SLL
1011	ROR

For example if we select 0000,XOR operation will be executed bit by bit , so as we can see in the ALU result=01110110 because 0 XOR 0=0,1 XOR 0= 1,1 XOR 1=0

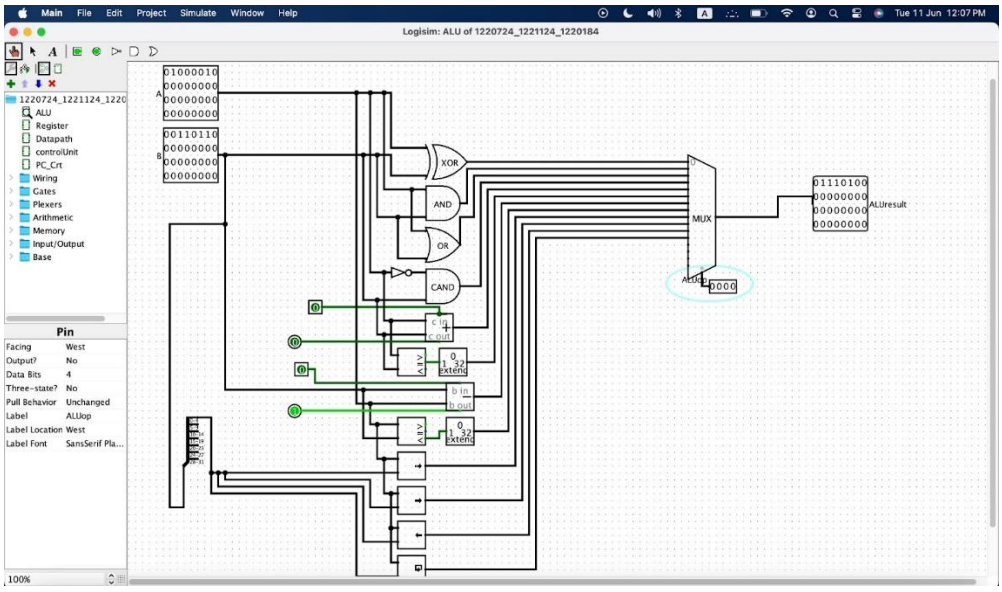


Figure 17: XOR

if we select 0001,AND operation will be executed bit by bit , so as we can see in the ALU result=00000010 because the output will be one only if the inputs are 1.

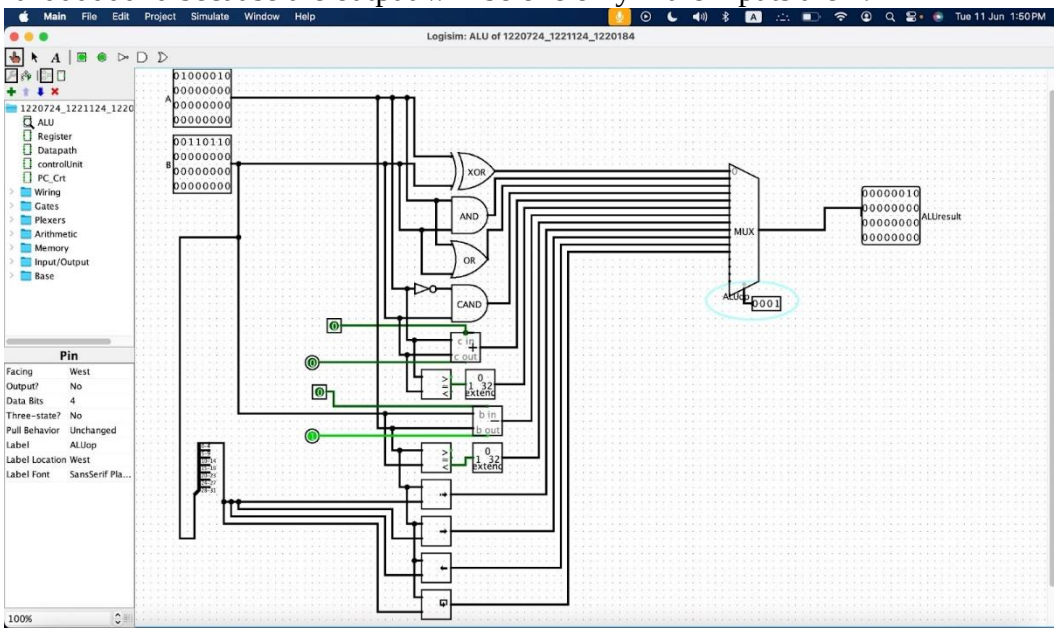


Figure 18: AND

if we select 0010,,OR operation will be executed bit by bit , so as we can see in the ALU result=01110110 because the output will be zero only if the inputs are 0 otherwise it will be one.

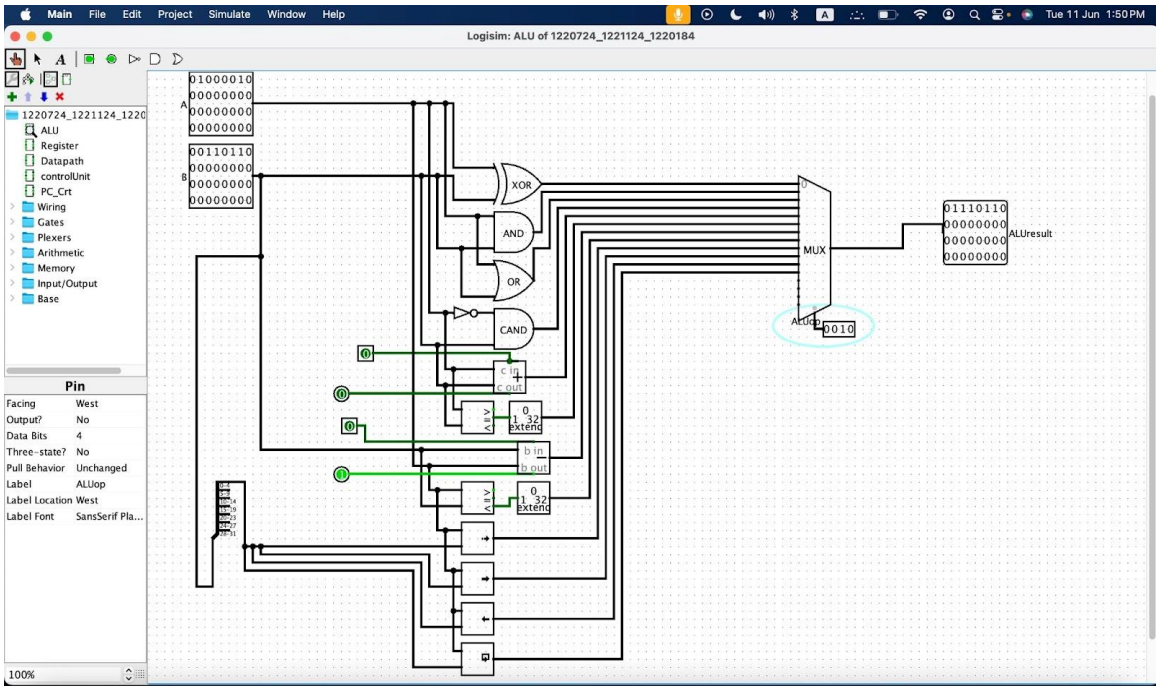


Figure 19: AND

if we select 0011,CAND operation will be executed bit by bit ,it will take the complement of the first input and the AND operation with the second input so as we can see the complement of(A =01000010) is 10111101 and then AND with (B =00110110).the ALU result =00110100.

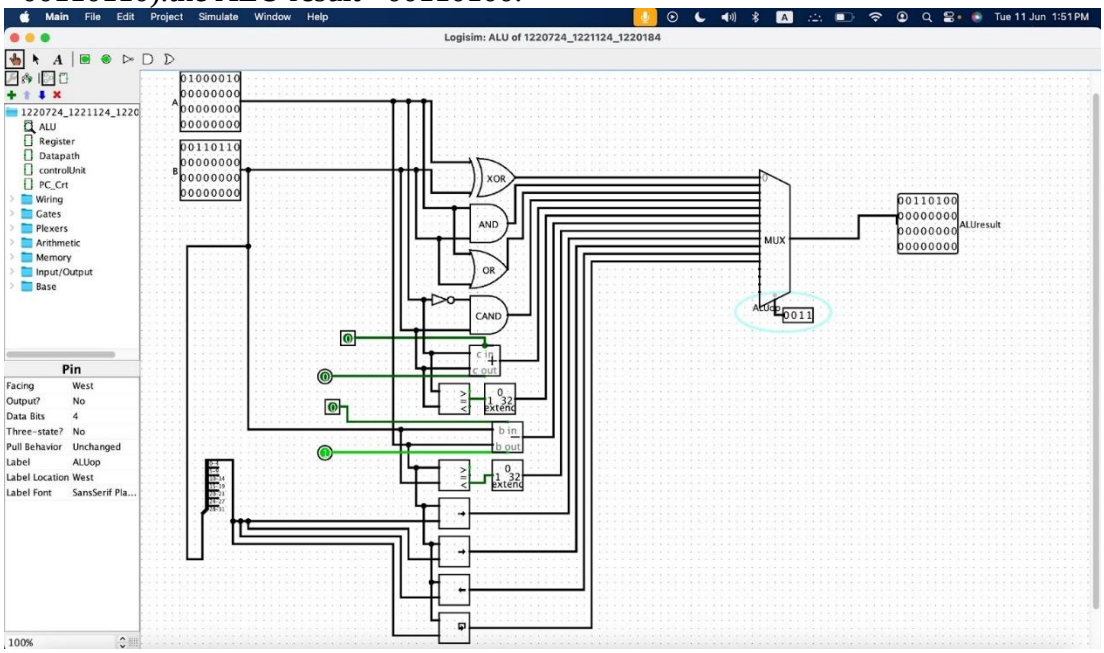


Figure 20: CAND

if we select 0100,ADD operation will be executed bit by bit ,it works as the normal full adder,0+0=0,1+0=1,1+1=0 and the carry =1, A=01000010 + B=00110110 —>Result=01111000

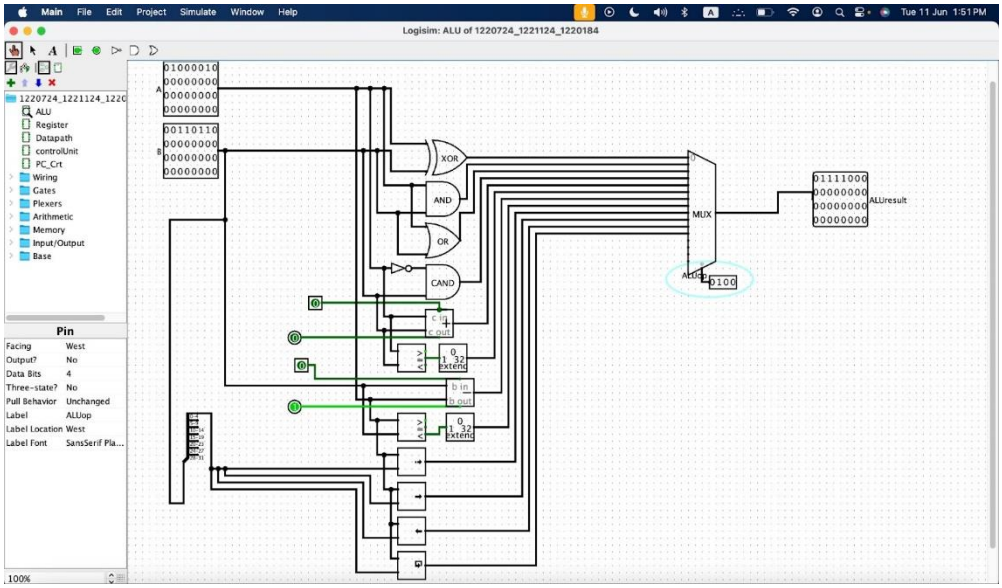


Figure 21: CAND

if we select 0101,SEQ(Set if Equal) operation will be executed which means Set the flag to 1 if the content of Ry equal to Rz,in other words if A=B,but in our example A!=B so the output is zero.if A=B we're going to see 1 in the LSB.

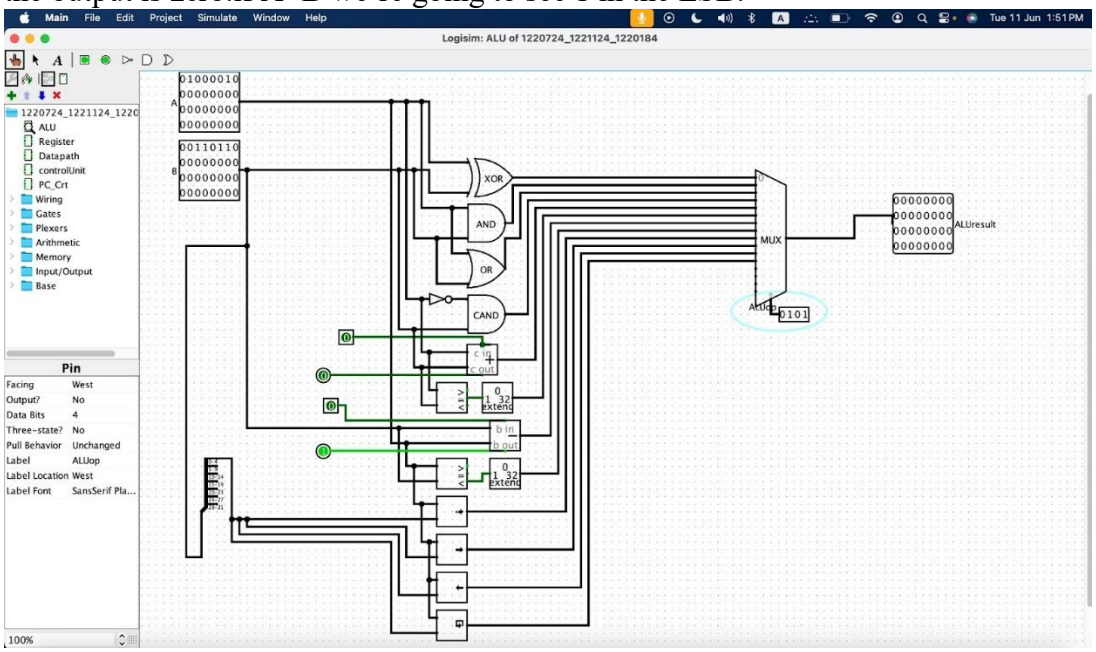


Figure 22: SEQ

if we select 0110,NADD(Negative ADD) operation will be executed which means it will take the 2's complement of A and the ADD it to B it is like the subtraction operation.

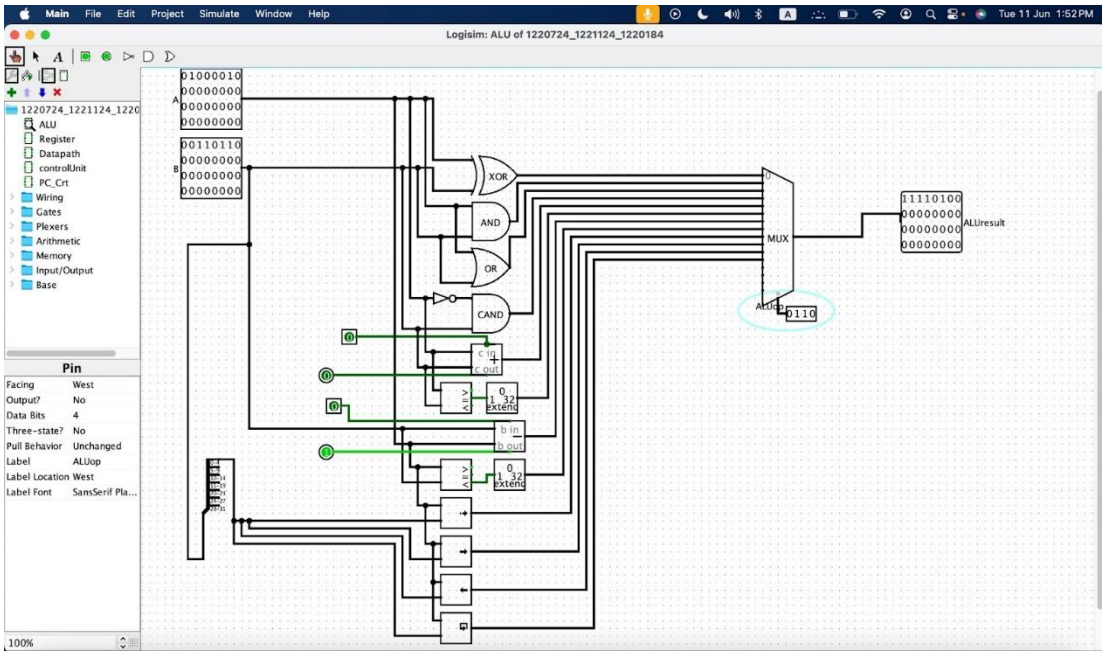


Figure 23: NADD

if we select 0111,SLT(Set if Less Than) operation will be executed which means Set the flag to 1 if the content of Ry Less than Rz,in other words if A<B.
A=00110110 < B=00110111.

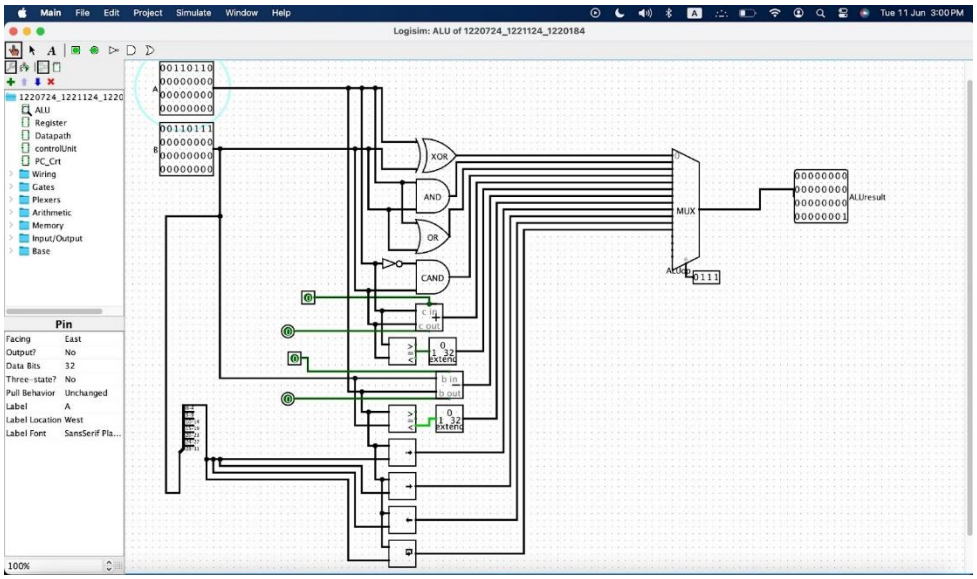
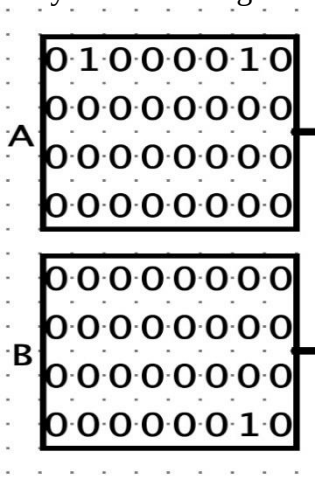


Figure 24: SLT

The next instructions will be the shift And Rotate operations

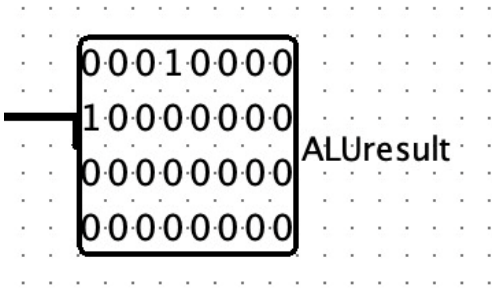
-Arithmetic Right Shift
-Logical Right Shift
-Logical Left Shift /Arithmetic Left Shift
-Rotate Right

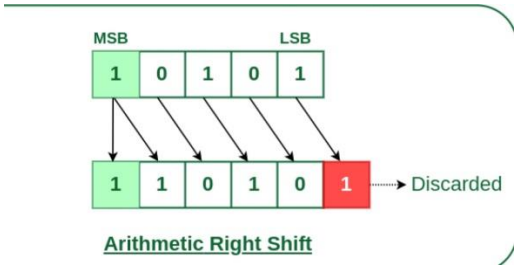
- In these instructions we're going to shift the input A (01000010..0) -Shown clearly in the next figure - by the value that we select in input B (in the LSB)



1)Arithmetic Right Shift

In this shift, each bit is moved to the right one by one and the least significant(LSB) bit is rejected and the most significant bit(MSB) is filled with the value of the previous MSB.





As we can see after we select 1000 the value of A=01000010 shifted 2 bits to the right arithmetically because we chose the value of B 0000...010.

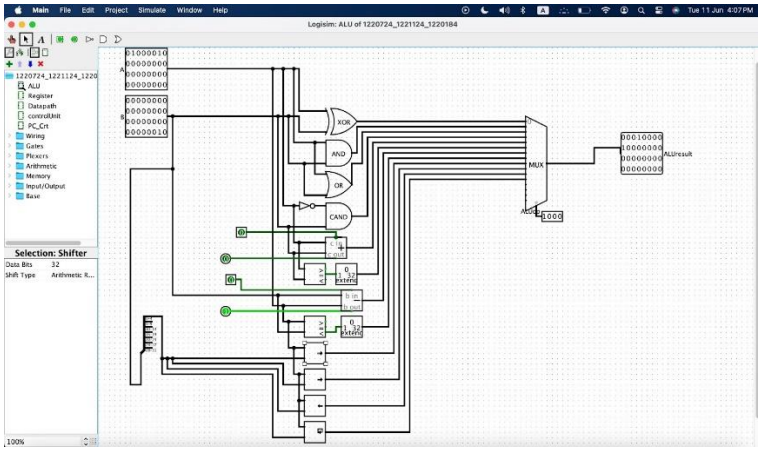
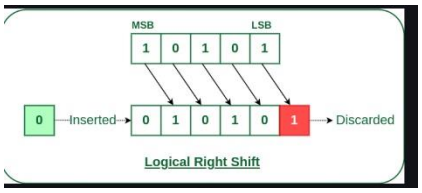


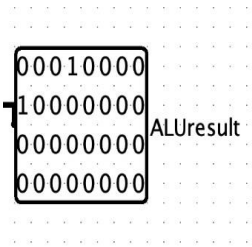
Figure 25: Arithmetic Right Shift

2) Logical Shift right

The logical shift right in general will shift a number towards the right by 1 bit it divides that number by 2 as shown in the figure below.



A=01000010 shifted 2 bits to the right so the ALU result



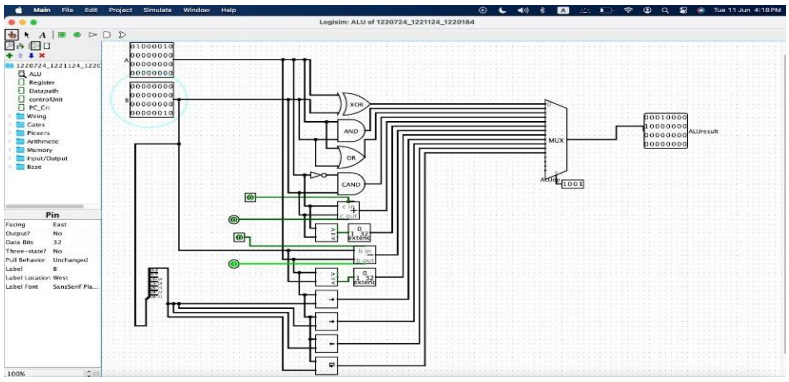
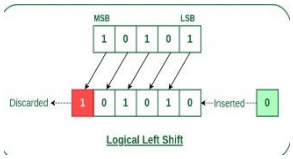


Figure 26: logical shift right

3) Logical and Arithmetic left Shift

The logical and Arithmetic left operations are the same the figure below shows us how the operation executed, In this shift, one position moves each bit to the left one by one. The least significant bit (LSB) is filled with zero and the most significant bit (MSB) is discarded.



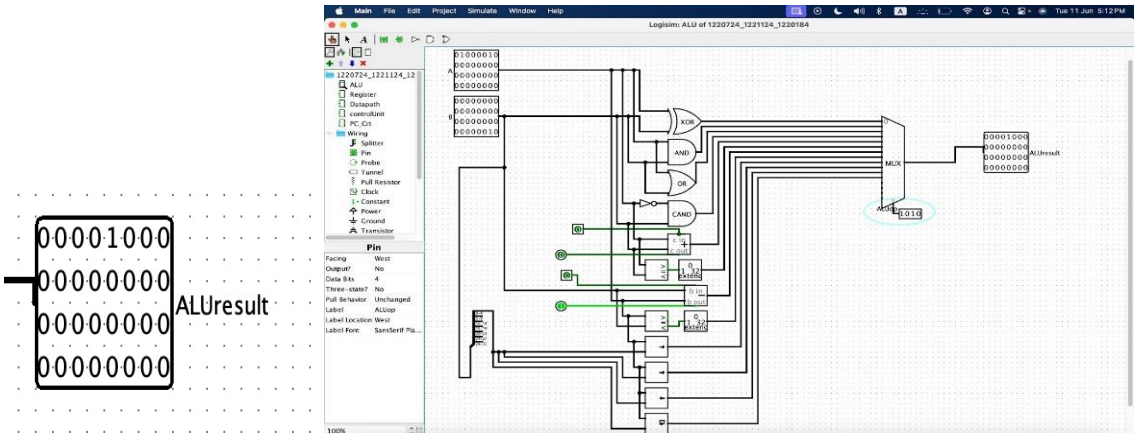


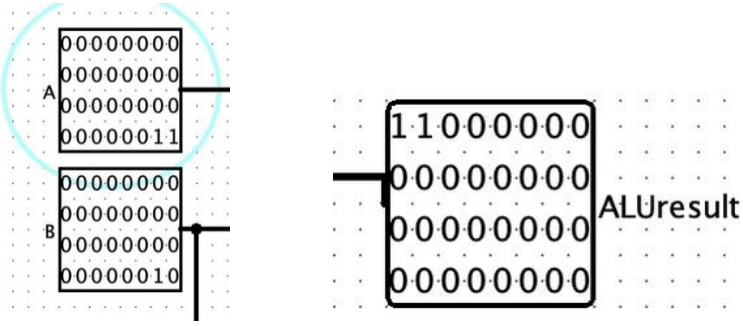
Figure 27: logical and arithmetic left shift

4) Bit Rotation :

A rotation (or circular shift) is an operation similar to shift except that the bits that fall off at one end are put back to the other end.

In right rotation the bits that fall off at right end are put back at left end.

As shown in our ALU Result.



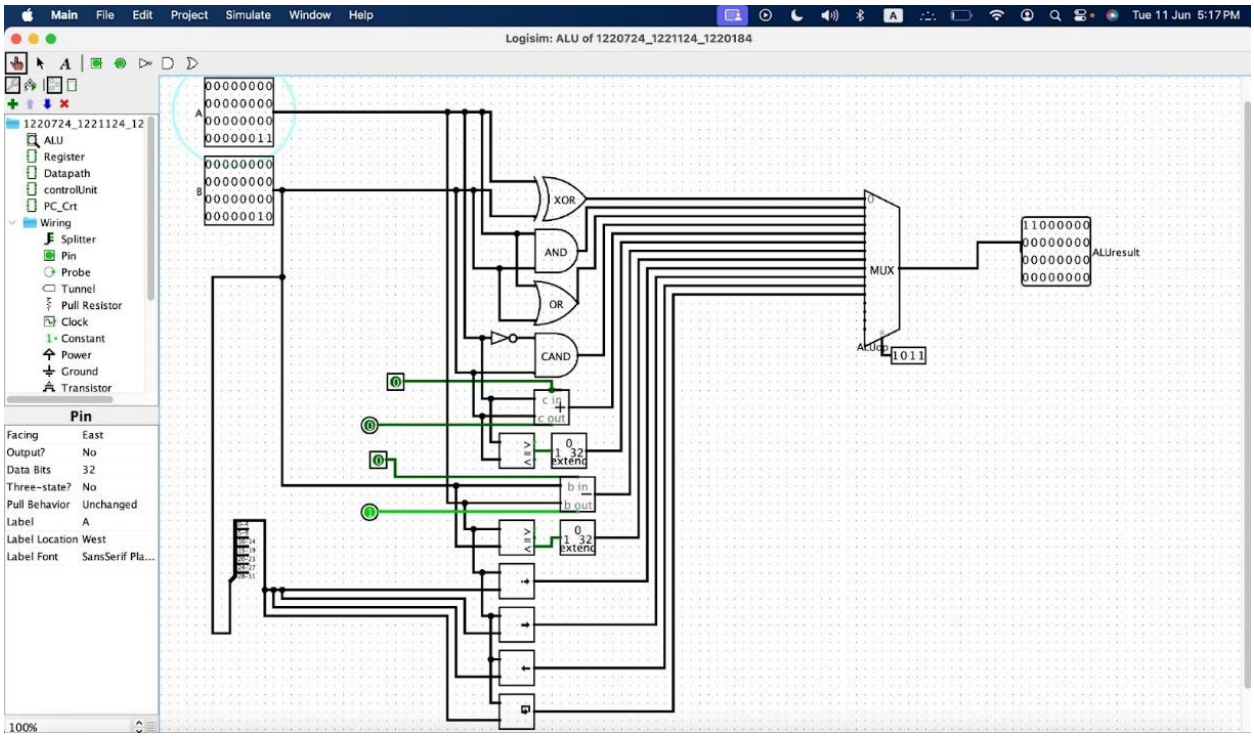


Figure 28: rotate right

Data Path:

A reasonable way to start a datapath design is to examine the major components required to execute each class of MIPS instructions. Let's start at the top by looking at which datapath elements each instruction needs, and then work our way down through the levels of abstraction. When we show the datapath elements, we will also show their control signals. We use abstraction in this explanation, starting from the bottom up.

To start, we will look at the datapath elements needed by every instruction.
First, we have instruction memory. Instruction memory is a state element that provides read-access to the instructions of a program and, given an address as input, supplies the corresponding instruction at that address. Code can also be written, e.g., self-modifying code.

Next, we have the program counter or PC. The PC is a state element that holds the address of the current instruction. Essentially, it is just a 20-bit register which holds the instruction address and is updated at the end of every clock cycle. Normally PC increments sequentially except for branch instructions The arrows on either side indicate that the PC state element is both readable and writeable.

Lastly, we have the adder. The adder is responsible for incrementing the PC to hold the address of the next instruction. It takes two input values, adds them together and outputs the result.

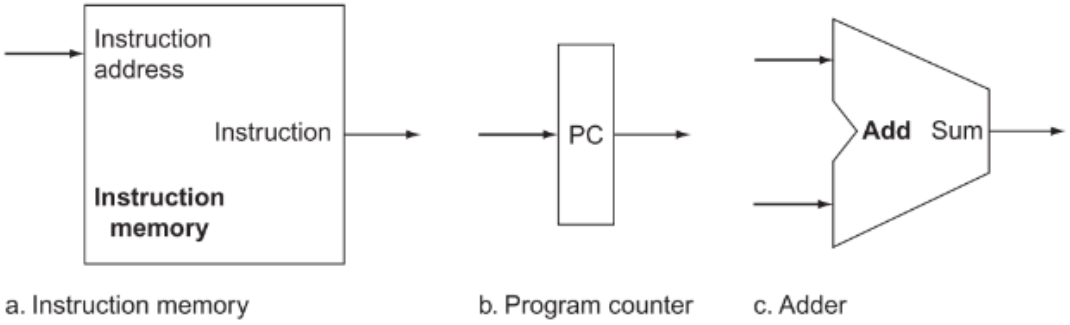


Figure 29: components of data path

So now we have instruction memory, PC, and adder datapath elements. Now, we can talk about the general steps taken to execute a program.
• **Instruction fetching**: use the address in the PC to fetch the current instruction from instruction memory.
• **Instruction decoding**: determine the fields within the instruction
• **Instruction execution**: perform the operation indicated by the instruction.
• Update the PC to hold the address of the next instruction.

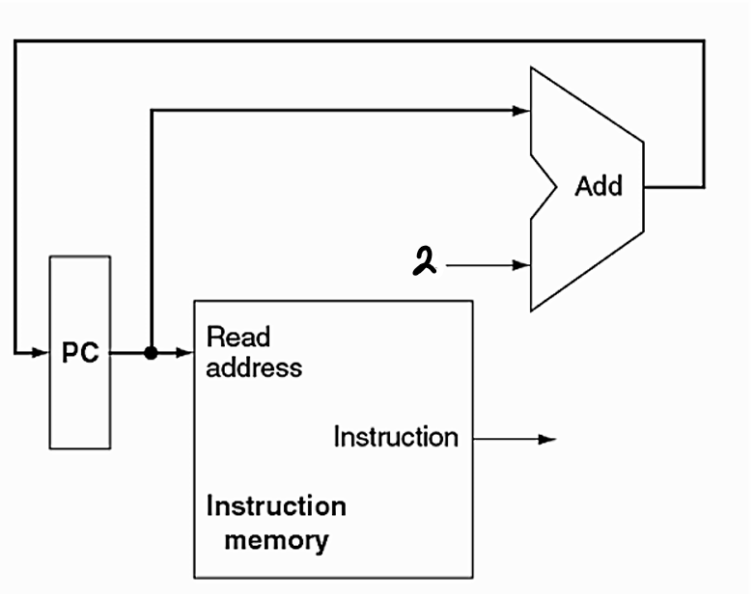


Figure 30: data path

Note: we perform PC+2 because instructions are 16-Bit long.

Components of the Datapath

- Combinational Elements
    - ALU, Adder
    - Immediate extender
    - Multiplexers
- Storage Elements
    - Instruction memory
    - Data memory
    - PC register
    - Register file
- Clocking methodology
    - Timing of writes

The processor's 32 general-purpose registers are stored in a structure called a register file. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the computer. In addition, we will need an ALU to operate on the values read from the registers.

To support ALL-format instructions, we'll need to add a state element called a register file. A register file is a collection readable/writeable registers.

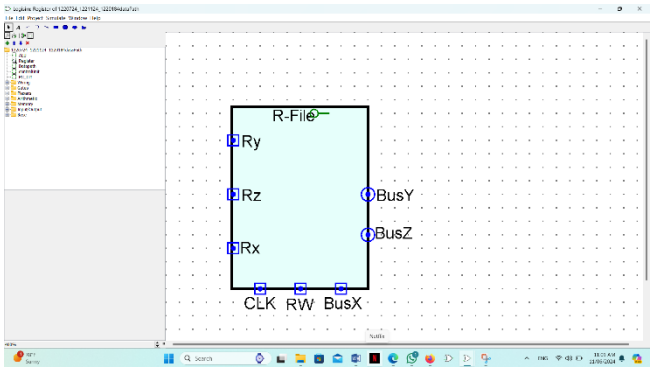


Figure 31: R-file block

To actually execute R-format instructions, we need to include the ALU element. The ALU performs the operation indicated by the instruction. It takes two operands, as well as a 4-bit wide operation selector value. The result of the operation is the output value. ALU operation is a part of the control. We discuss datapath first. We have an additional output specifically for branching we will cover this in a minute.

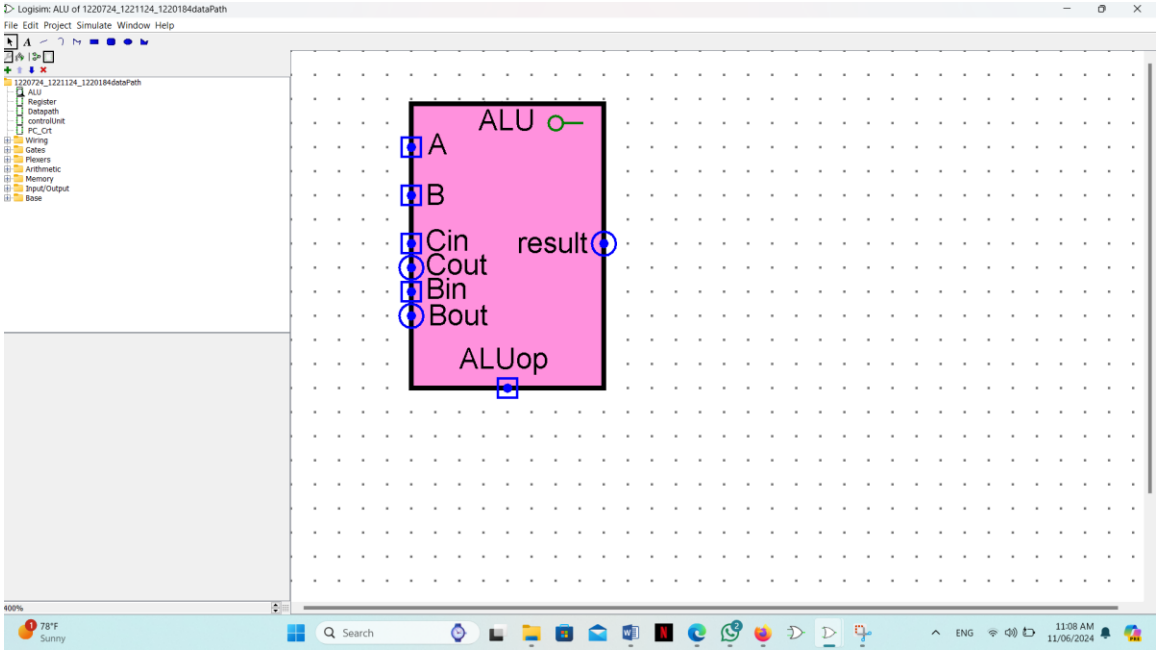


Figure 32: ALU block

R-FORMAT INSTRUCTIONS

Now, let's consider R-format instructions. In our limite instruction set, these are AND, OR, XOR, ADD, SUB, SEQ and SLT.

All R-format instructions read two registers, Ry and Rz, and write to a register Rx.

5-bit Operation code (Op), 3-bit register numbers (x, y, and z), and 2-bit function field f

5-opcode	3-x	3-y	3-z	2-f
MSB				LSB

Figure 33: R-format Instructions

Here is our datapath for R-format instructions.

1. Grab instruction address from PC.

2. Fetch instruction from instruction memory.

3. Decode instruction.

4. Pass Ry, Rz, and Rx into read register and write register arguments.

5. Retrieve data from BusY and rBusZ (Ry and Rz).

6. Pass contents of Ry and Rz into the ALU as operands of the operation to be performed.

7. Retrieve result of operation performed by ALU and pass back as the write data argument of the register file (with the RegWrite bit set).

8. Add 2 bytes to the PC value to obtain the Half-word-aligned address of the next instruction.

I-FORMAT INSTRUCTIONS

Now that we have a complete datapath for R-format instructions, let's add in support for I-format instructions. In our limited instruction set, these are SLL, SRL, SRA, ROR, BEQ, BNE, BLT, BGE, and LW

I-type

5-bit Op, 3-bit register numbers (x and y), and 5-bit Immediate

5-opcode	3-x	3-y	Imm5

Figure 34: I-format instructions

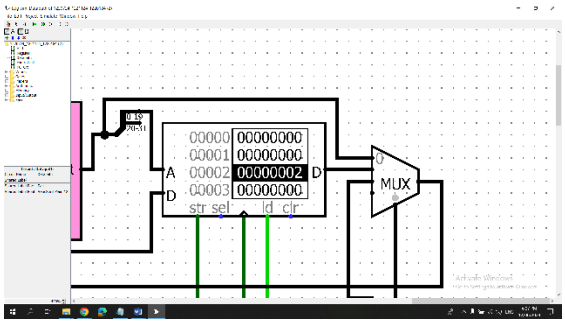
Let's start with accommodating the data transfer. For LW, we have the following format:
LW Rx, [Ry + sign_extend(Imm5)]

• The memory address is computed by sign-extending the 5-bit immediate to 32-bits, which is added to the contents of Ry. •

In LW, Rx represents the register that will be assigned the memory value. Bottom line: we need two more datapath elements to access memory and perform sign-extending.

The data memory element implements the functionality for reading and writing data to/from memory. There are two inputs. One for the address of the memory location to access, the other for the data to be written to memory if applicable. The output is the data read from the memory location accessed, if applicable. Reads and writes are signaled by MemRead and MemWrite, respectively, which must be asserted for the corresponding action to take place.

And this goes for all branching instructions.



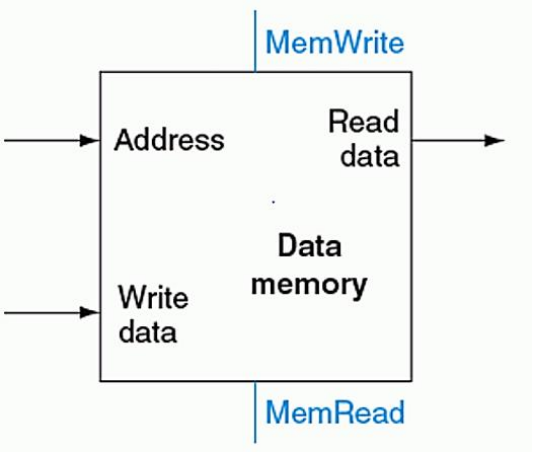


Figure 35: Data Memory schematic and in logisim

To perform sign-extending, we can add a sign extension element. The sign extension element takes as input a 5-bit wide value to be extended to 32-bits. To sign extend, we simply replicate the most-significant bit of the original field until we have reached the desired field width.

Here, we have modified the datapath to work only for the lw instruction.
Lw (Rx ← 4byte MEM[Ry + sign_extend(Imm5)]) The registers have been added to the datapath for added clarity.

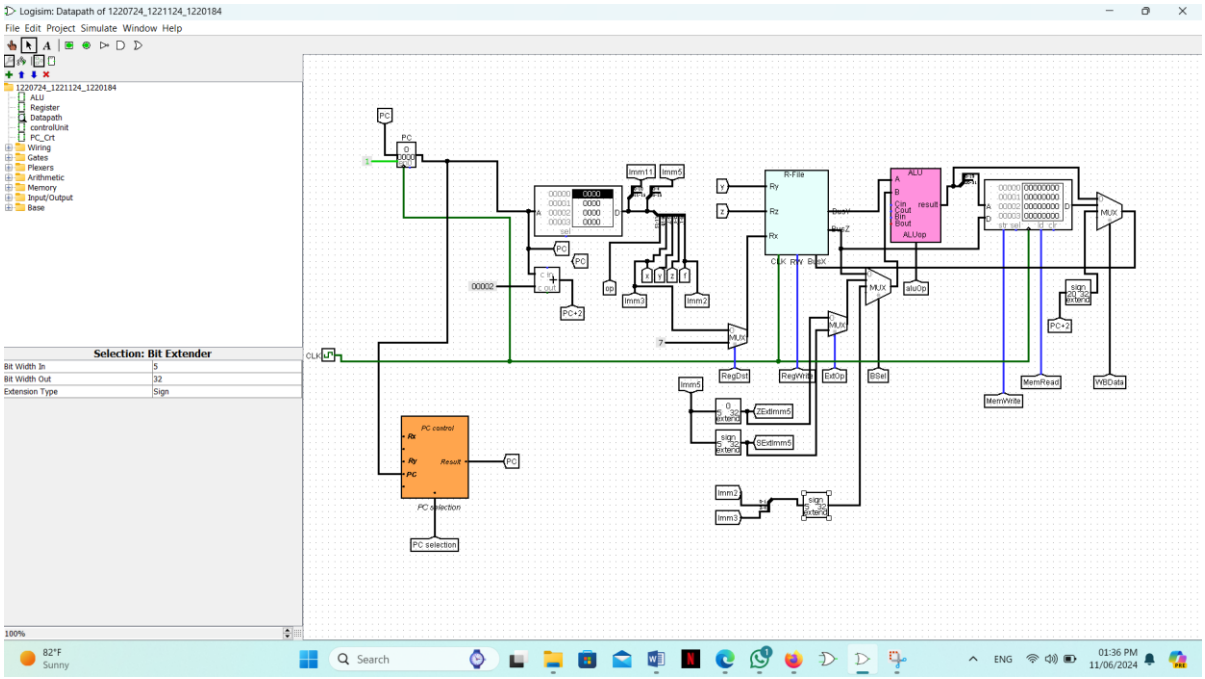


Figure 36: DataPath on logisim

Now we'll turn out attention to a branching instruction. In our limited instruction set, we have the BEQ instruction which has the following form: BEQ Branch if (Rx == Ry) (PC += sign_extend(Imm5<<1)) target This instruction compares the contents of Rx and Ry for equality and uses the 5-bit immediate field to compute the target address of the branch relative to the current address.

Note that our immediate field is only 5-bits so we can't specify a full 20-bit target address. So we have to do a few things before jumping.
• The immediate field is left-shifted by one because the immediate represents the number of words offset from PC+2, not the number of bytes (and we want to get it in number of bytes!).
• We sign-extend the immediate field to 20-bits and add it to PC+2.

S-FORMAT INSTRUCTIONS

Now that we have a complete datapath for R-format and I-format instructions, let's add in support for S-format instructions. In our limited instruction set, these are SW.

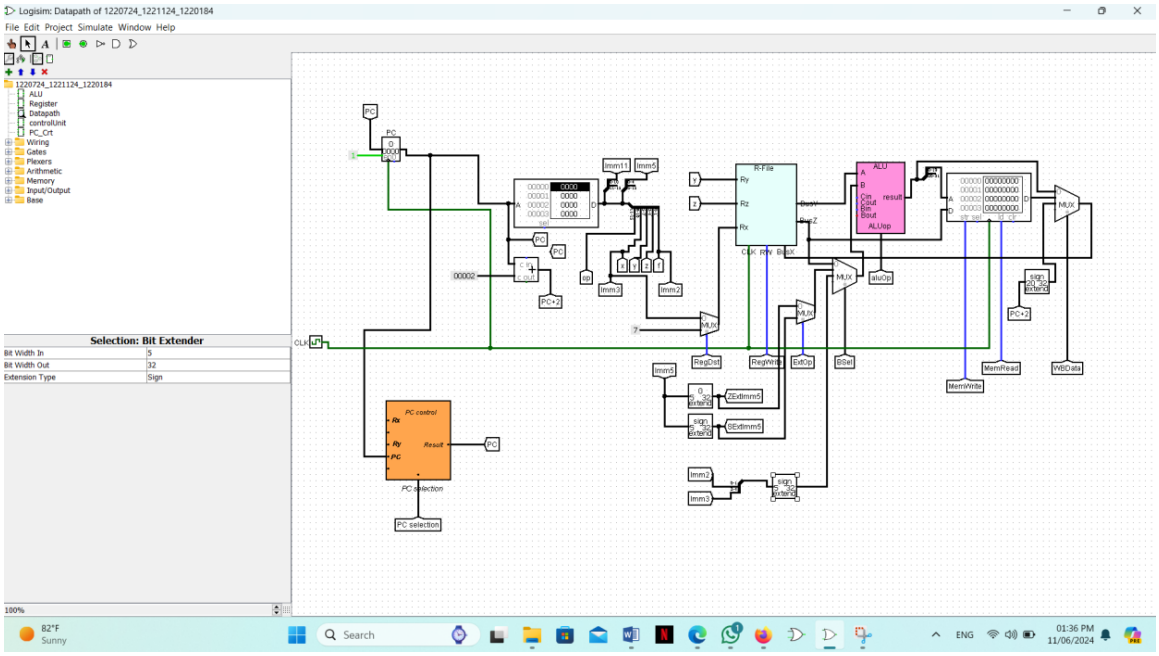
S-type

5-bit Op, 3-bit register numbers (y and z), and 5-bit Immediate split into (Imm3 and Imm2)

5-opcode	Imm3	3-y	3-z	Imm2

Let's start with accommodating the data transfer. For SW, we have the following format:
SW MEM, [Ry + sign_extend({Imm3,Imm2})]

Here, we have modified the datapath to work only for the SW instruction.
The registers have been added to the datapath for added clarity.

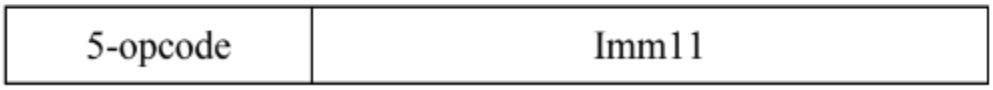


J-FORMAT INSTRUCTIONS

The last instruction we have to implement is the jump instruction. An example jump instruction is j L1. This instruction indicates that the next instruction to be executed is at the address of label L1.

J-type

5-bit Op and 11-bit Immediate



Note, we do not have enough space in the instruction to specify a full target address.
• Branching solves this problem by specifying an offset in words.
• Jump instructions solve this problem by specifying a portion of an absolute address
• Take the 11-bit target address field of the instruction, left-shift by one (instructions are Half-word-aligned), and add PC+2.

Control Unit:

Now we have a complete datapath for our simple Instruction subset – we will show the whole diagram in just a couple of minutes. Before that, we will add the control. The control unit is responsible for taking the instruction and generating the appropriate signals for the datapath elements. Signals that need to be generated include

• Operation to be performed by ALU.
• Whether register file needs to be written.
• Signals for multiple intermediate multiplexors.
• Whether data memory needs to be written.

For the most part, we can generate these signals using only the opcode and funct fields of an instruction.

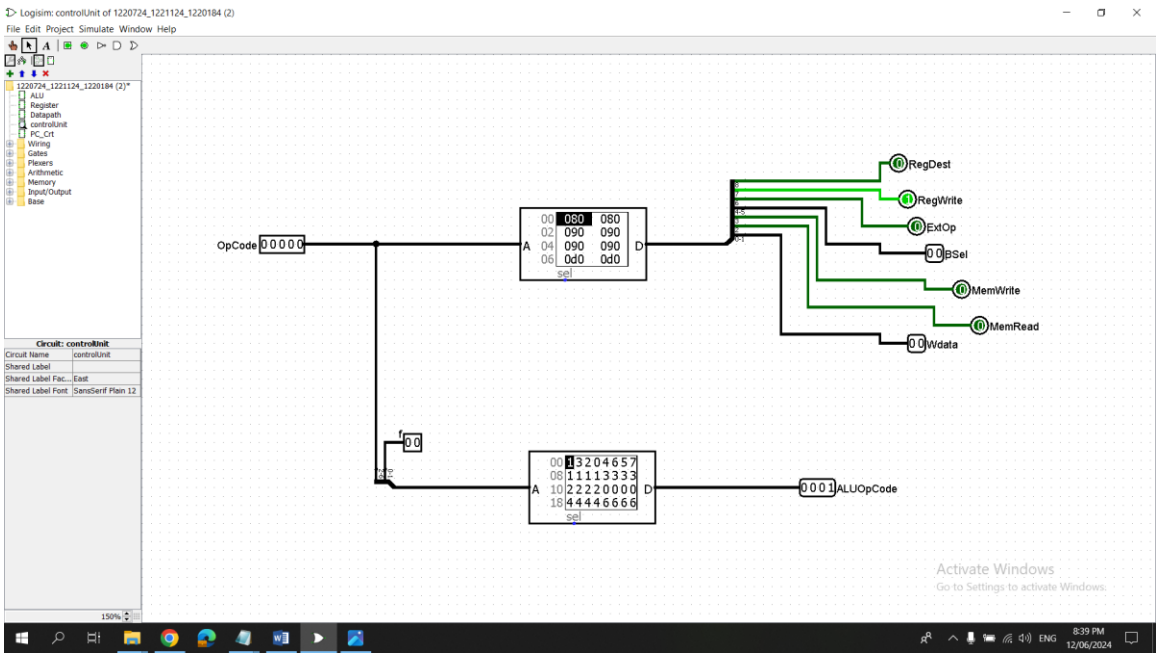


Figure 37: control unit on logisim

The Control memory is assumed to be a ROM, within which all control information is permanently stored. The control register holds the microinstruction fetched from the memory. The micro-instruction contains a control word that specifies one or more micro-operations for the data processor.
First memory has the controls below:

Table 1: Controls

RegDst	1-bit	0 → Rx
		1 → R7
RegWrite	1-bit	0 → no write on Reg
		1 → write on Reg
ExtOp	1-bit	0 → zero ext. Imm5
		1 → sign ext. Imm5
BSel	2-bit	00 → BusZ
		01 → Imm5
		10 → Imm3Imm2
		11 → Imm11
MemWrite	1-bit	0 → noWrite
		1 → Write
MemRead	1-bit	0 → noRead
		1 → Read
Wdata	2-bit	00 → ALU result
		01 → DataMem
		10 → PC+2

And we constructed the hexadecimal values of these controls in Table (2), and we wrote the values on the memory as shown in figure (37).

Table 2: hexa values of the controls for each instruction

Controls→	OP	RegDes	RegWrite	ExtOp	Bsel	MemWrite	MemRead	Wdata	Binary code	Hexa_value
AND	0	0	1	x	00	0	0	00	0_1000_0000 0_1100_0000	0x080 0x0c0
CAND	0	0	1	x	00	0	0	00	0_1000_0000 0_1100_0000	0x080 0x0c0
OR	0	0	1	x	00	0	0	00	0_1000_0000 0_1100_0000	0x080 0x0c0
XOR	0	0	1	x	00	0	0	00	0_1000_0000 0_1100_0000	0x080 0x0c0
ADD		0	1	x	00	0	0	00	0_1000_0000 0_1100_0000	0x080 0x0c0
NADD		0	1	x	00	0	0	00	0_1000_0000 0_1100_0000	0x080 0x0c0
SEQ		0	1	x	00	0	0	00	0_1000_0000 0_1100_0000	0x080 0x0c0
SLT		0	1	x	00	0	0	00	0_1000_0000 0_1100_0000	0x080 0x0c0

ANDI		0	1	0	01	0	0	00	0_1001_0000	0x090
CANDI		0	1	0	01	0	0	00	0_1001_0000	0x090
ORI		0	1	0	01	0	0	00	0_1001_0000	0x090
XORI		0	1	0	01	0	0	00	0_1001_0000	0x090
ADDI		0	1	1	01	0	0	00	0_1101_0000	0x0d0
NADDI		0	1	1	01	0	0	00	0_1101_0000	0x0d0
SEQI		0	1	1	01	0	0	00	0_1101_0000	0x0d0
SLTI		0	1	1	01	0	0	00	0_1101_0000	0x0d0
SLL		0	1	x	01	0	0	00	0_1101_0000 0_1001_0000	0x0d0 0x090
SRL		0	1	x	01	0	0	00	0_1101_0000 0_1001_0000	0x0d0 0x090
SRA		0	1	x	01	0	0	00	0_1101_0000 0_1001_0000	0x0d0 0x090
ROR		0	1	x	01	0	0	00	0_1101_0000 0_1001_0000	0x0d0 0x090
BEQ		0	1	1	01	0	0	10	0_1101_0010	0x0d2
BNE		0	1	1	01	0	0	10	0_1101_0010	0x0d2
BLT		0	1	1	01	0	0	10	0_1101_0010	0x0d2
BGE		0	1	1	01	0	0	10	0_1101_0010	0x0d2
LW		0	1	1	01	0	1	01	0_1101_0101	0x0d5
SW		x	1	1	10	1	0	01	0_1110_1001 1_1110_1001	0x0e9 0x1e9
J		X	0	1	11	0	0	10	0_0111_0010 1_0111_0010	0x072 0x172
JAL		1	1	1	11	0	0	10	1_1111_0010	0x1f2

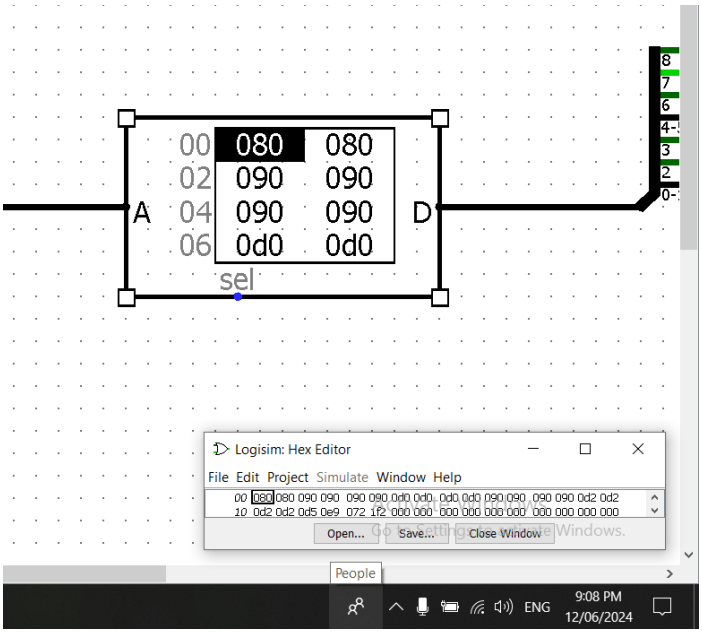


Figure 38:the values written on the memory

Same for the ALU opcode memory, for the instructions that have the same opcode so the value of (f) determine each instruction.

operation	op	f	AluOp	Binary code **{Opcode, f}**	Hexa value
AND	0	0	0001	000_0000	0x00
CAND	0	1	0011	000_0001	0x01
OR	0	2	0010	000_0010	0x02
XOR	0	3	0000	000_0011	0x03
ADD	1	0	0100	000_0100	0x04
NADD	1	1	0110	000_0101	0x05
SEQ	1	2	0101	000_0110	0x06
SLT	1	3	0111	000_0111	0x07
ANDI	2	X	0001	000_1000 000_1001 000_1010 000_1011	0x08 0x09 0x0a 0x0b
CANDI	3	X	0011	000_1100 000_1101 000_1110 000_1111	0x0c 0x0d 0x0e 0x0f
ORI	4	X	0010	001_0000 001_0001 001_0010 001_0011	0x10 0x11 0x12 0x13
XORI	5	X	0000	001_0100 001_0101 001_0110 001_0111	0x14 0x15 0x16 0x17
ADDI	6	X	0100	001_1000 001_1001 001_1010 001_1011	0x18 0x19 0x1a 0x1b
NADDI	7	X	0110	001_1100 001_1101 001_1110 001_1111	0x1c 0x1d 0x1e 0x1f
SEQI	8	X	0101	010_0000 010_0001 010_0010 010_0011	0x20 0x21 0x22 0x23
SLTI	9	X	0111	010_0100 010_0101 010_0110 010_0111	0x24 0x25 0x26 0x27

SLL	10	X	1010	010_1000	0x28
				010_1001	0x29
				010_1010	0x2a
				010_1011	0x2b
SRL	11	X	1001	010_1100	0x2c
				010_1101	0x2d
				010_1110	0x2e
				010_1111	0x2f
SRA	12	X	1000	011_0000	0x30
				011_0001	0x31
				011_0010	0x32
				011_0011	0x33
ROR	13	X	1011	011_0100	0x34
				011_0101	0x35
				011_0110	0x36
				011_0111	0x37

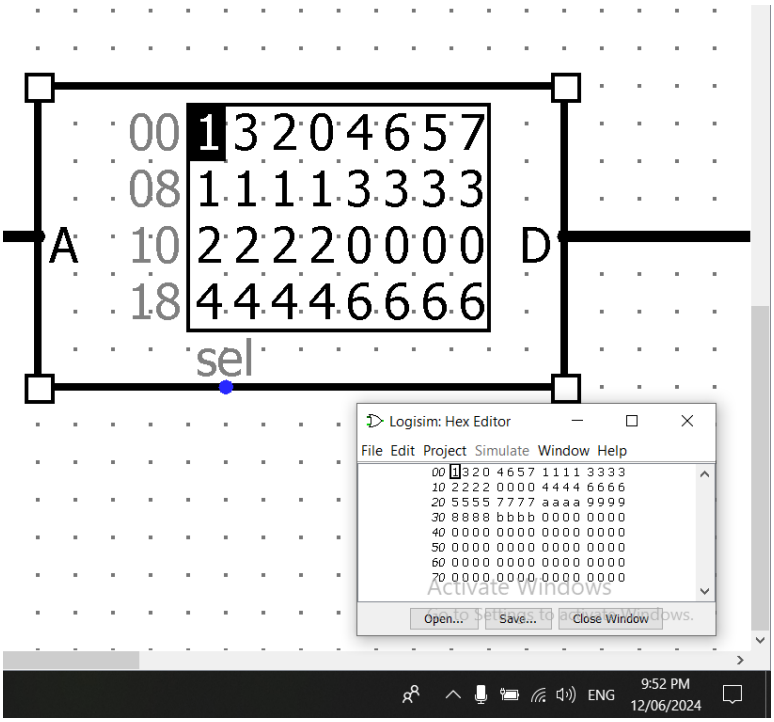


Figure 39: values written on memory

Testing the Control Unit:

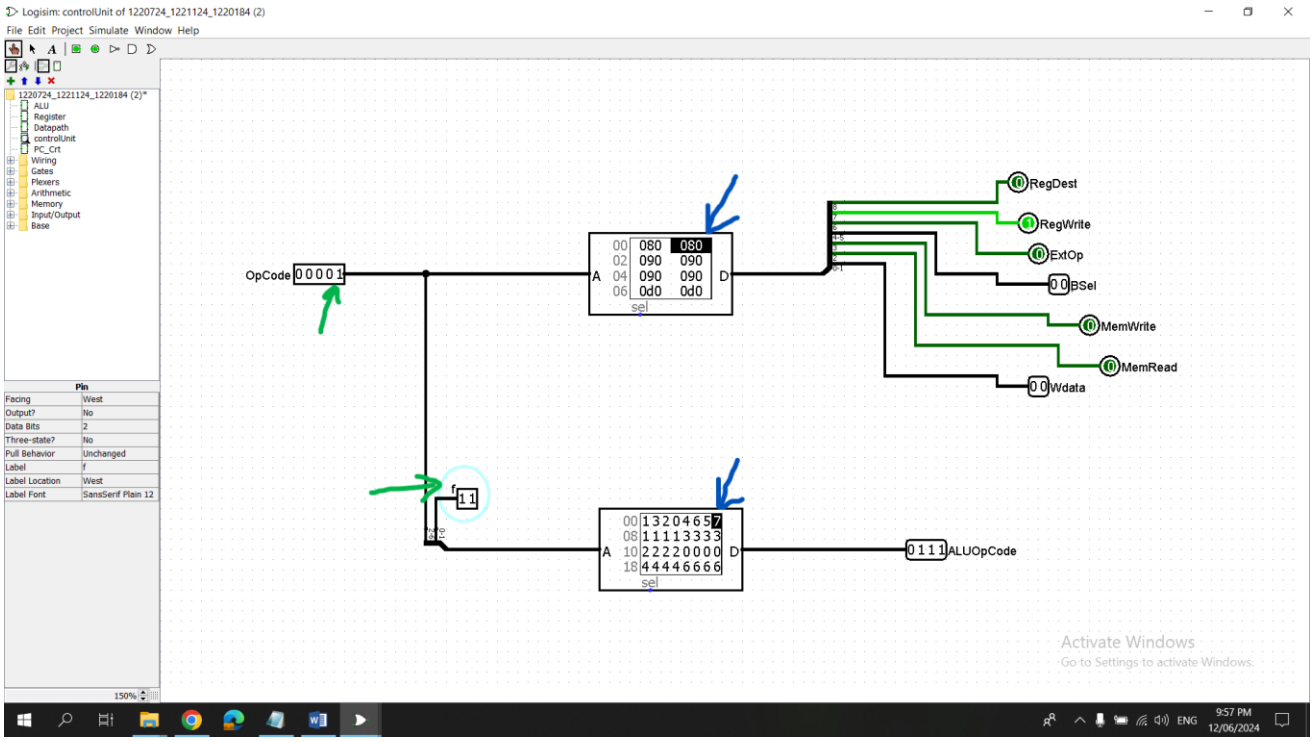


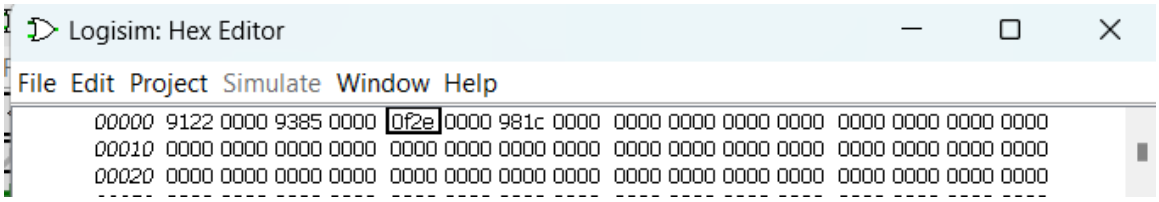
Figure 40: testing the CU

Testing and Verification:

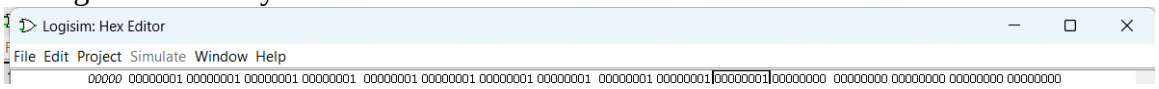
Testing SEQ Instruction:
Code:
LW R1, R2, Imm5
LW R3, R4, Imm5
SEQ R7, R1, R3, F
SW Imm3, R0,R7, Imm2

Binary code to Hex:
1001 0001 0010 0010 → 0x9122
1001 0011 1000 0101 → 0x9385
0000 1111 0010 1110 → 0x0F2E
1001 1000 0001 1100 → 0x981C

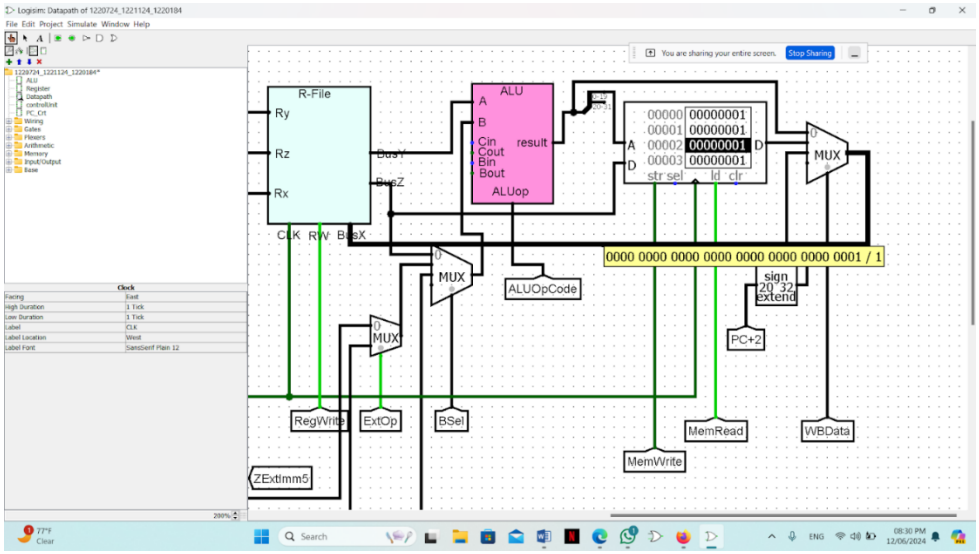
PC Instruction:

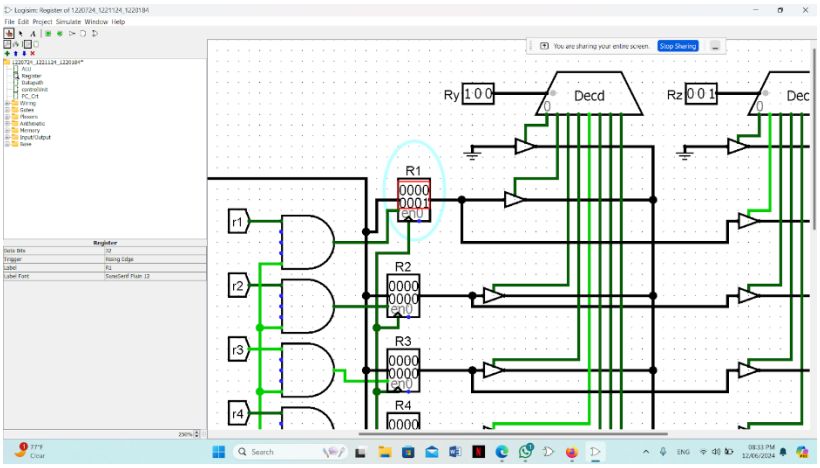


Filling Data Memory:

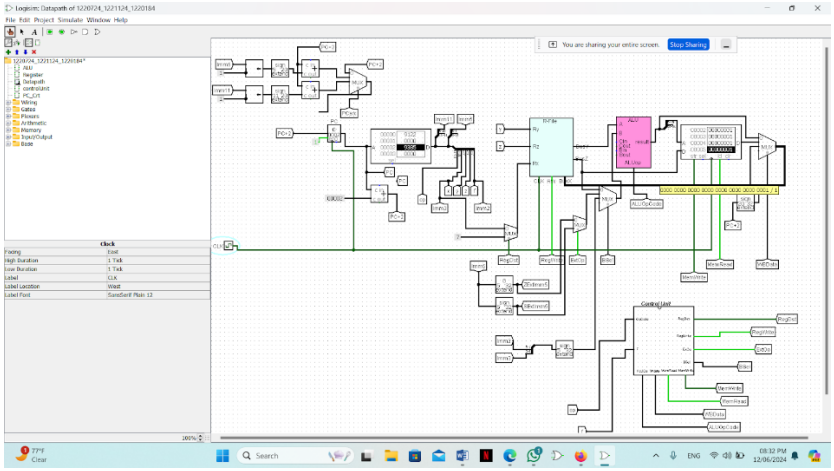


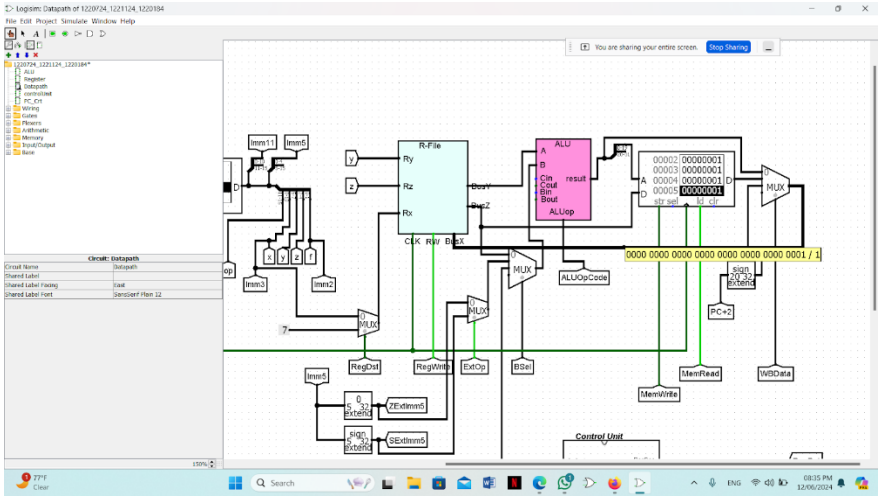
LOAD #1 to R1

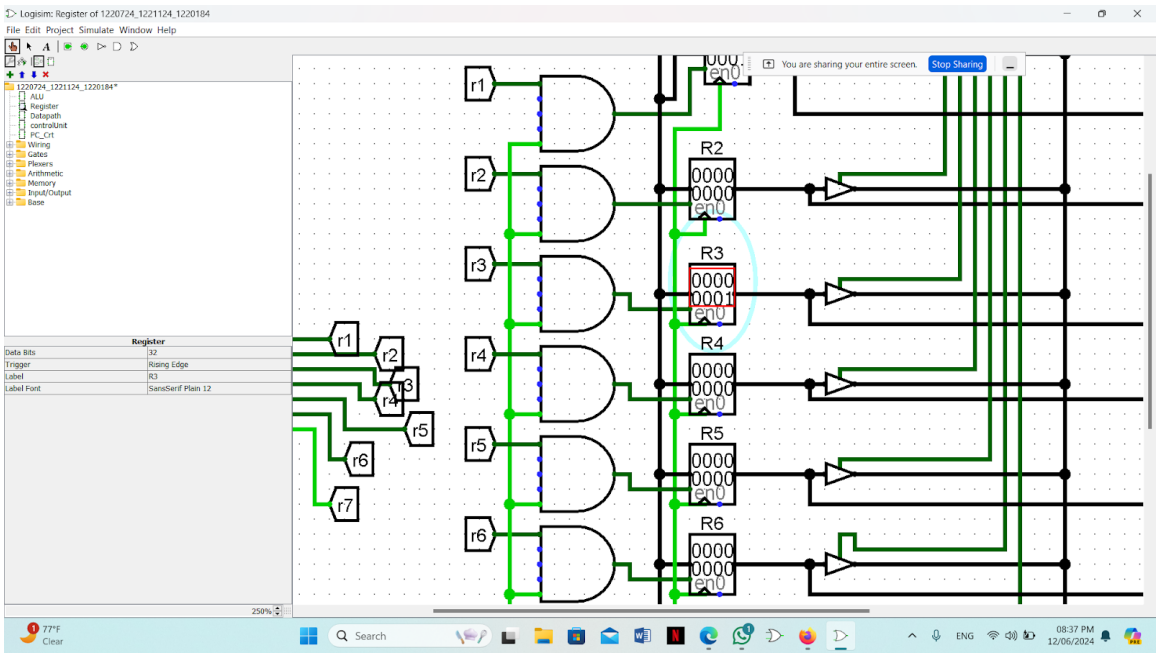



Ry 1 0 0
Decd
Rz 0 0 1
Dec
R1
0000
0001
R2
0000
0000
R3
0000
0000
R4
0000
r1
r2
r3
r4

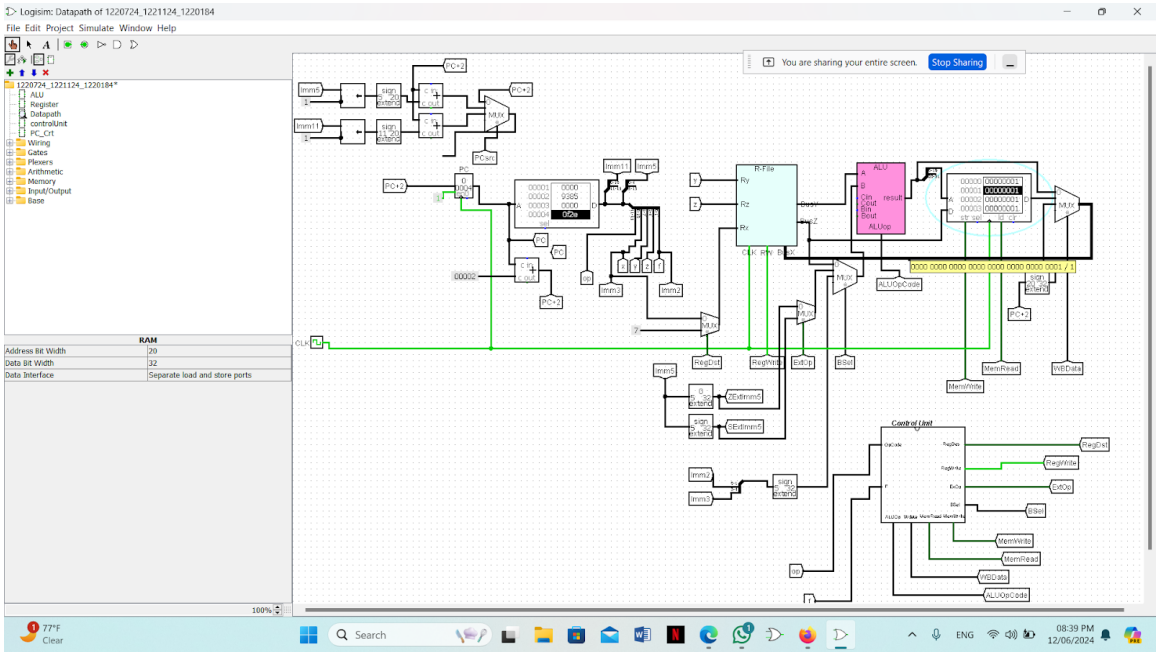
LOAD #1 TO R3:

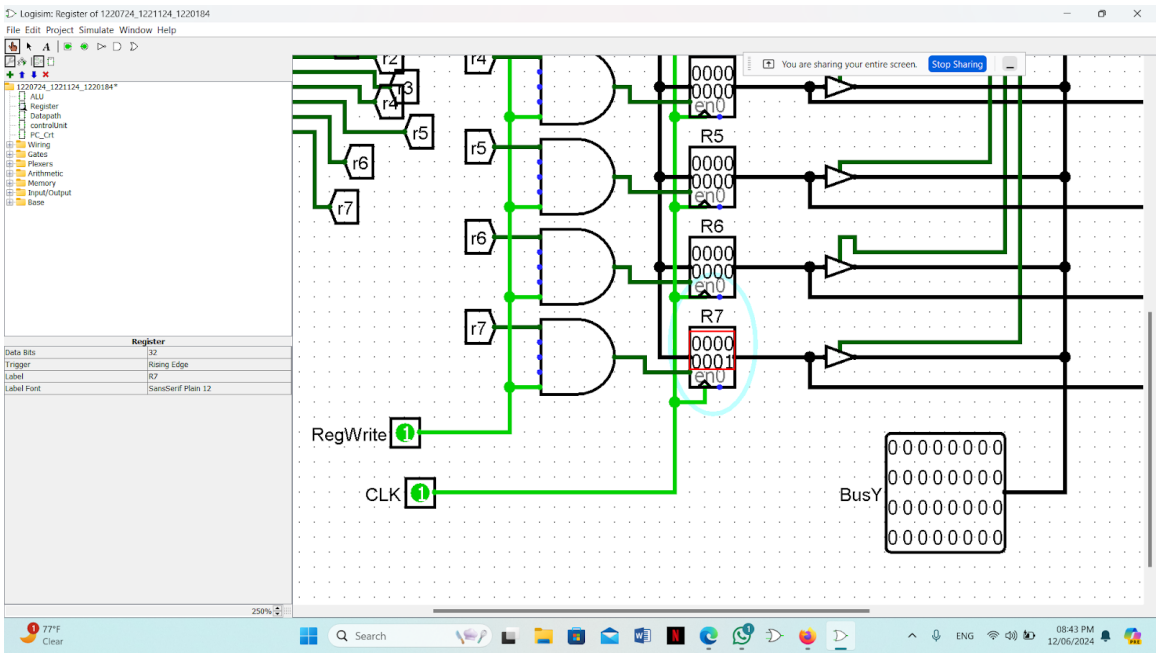



R-File
ALU
MUX
ALUOpCode
0000 0000 0000 0000 0000 0000 0000 0001 / 1
RegDst
RegWrite
ExtOp
BSel
MemRead
MemWrite
WBData
PC+2
Control Unit

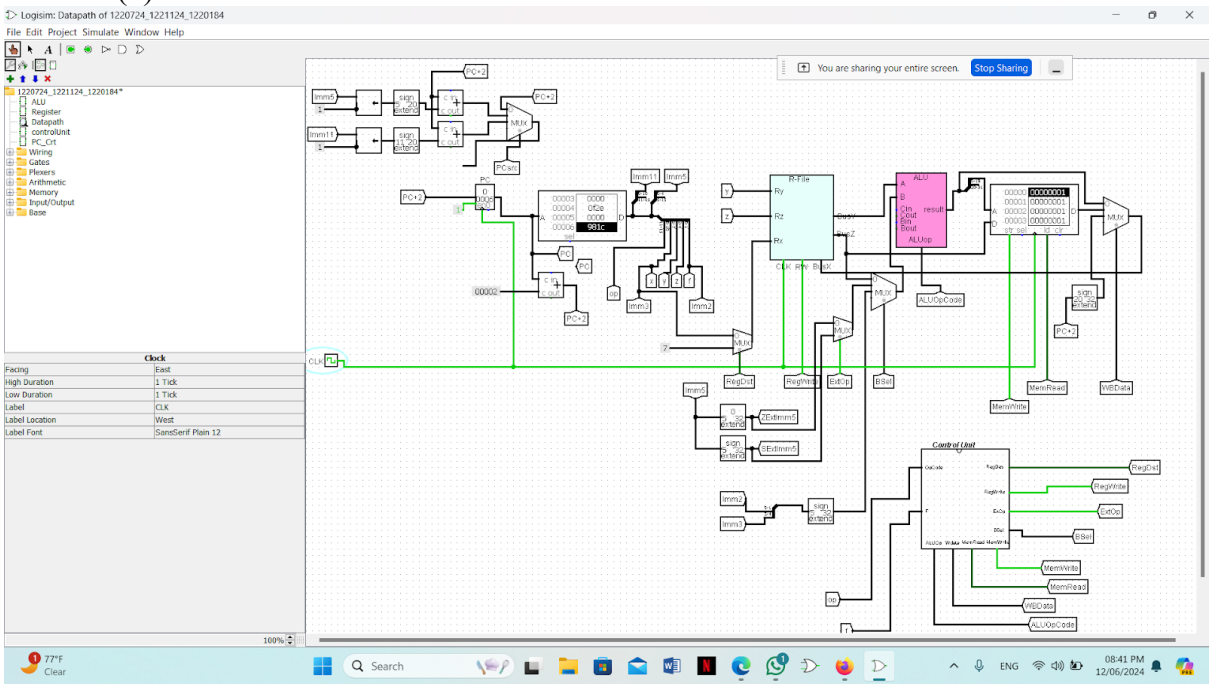


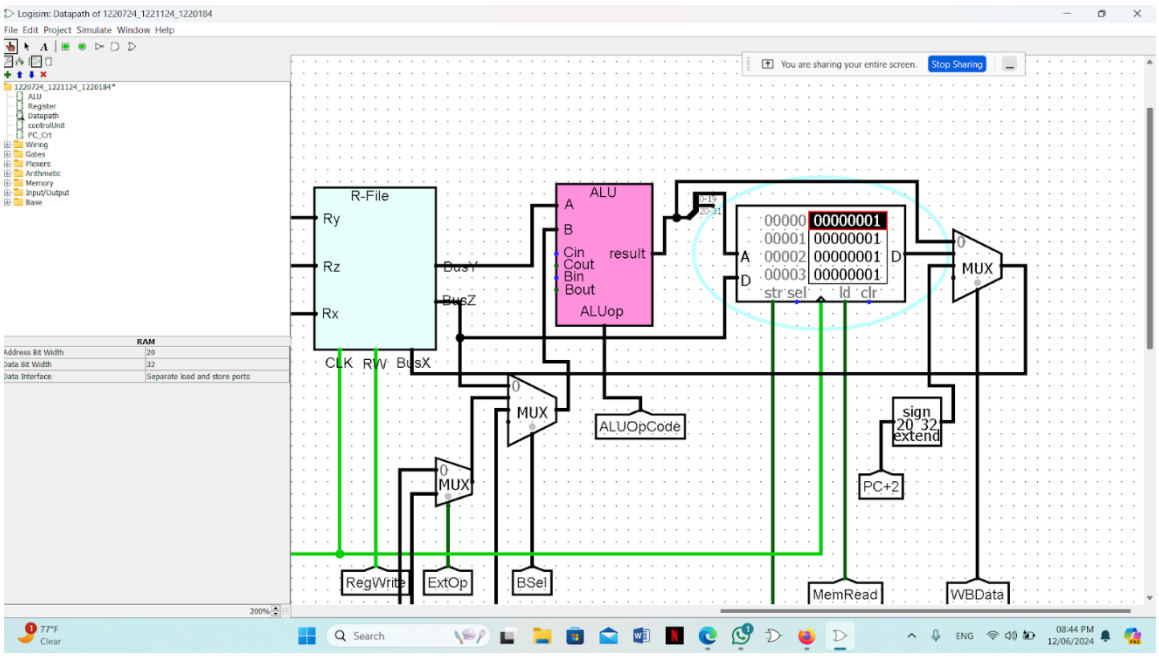
SEQ R7=(R1==R3) (Result is either 0 or 1)
result is 1 bec (R1(1)==R3(1))





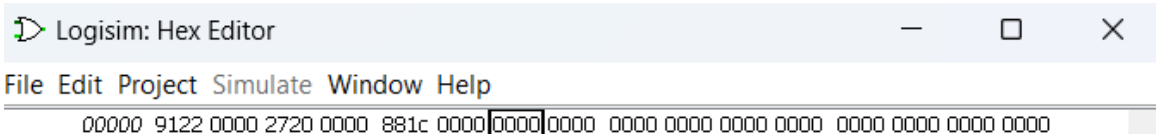
Store R7(1)→ MEM



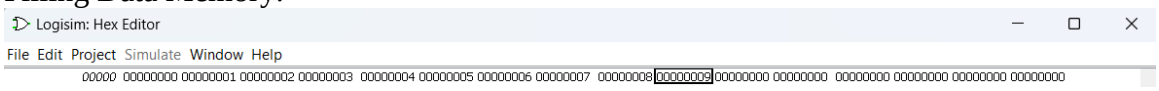


Testing ORI Instruction:
Code:
LW R1, R2, Imm5
ORI R7, R1, Imm5(00000)
SW Imm3, R0,R7, Imm2
Binary code to Hex:
1001 0001 0010 0010 → 0x9122
0010 0111 0010 0000 → 0x2720
1001 1000 0001 1100 → 0x981C

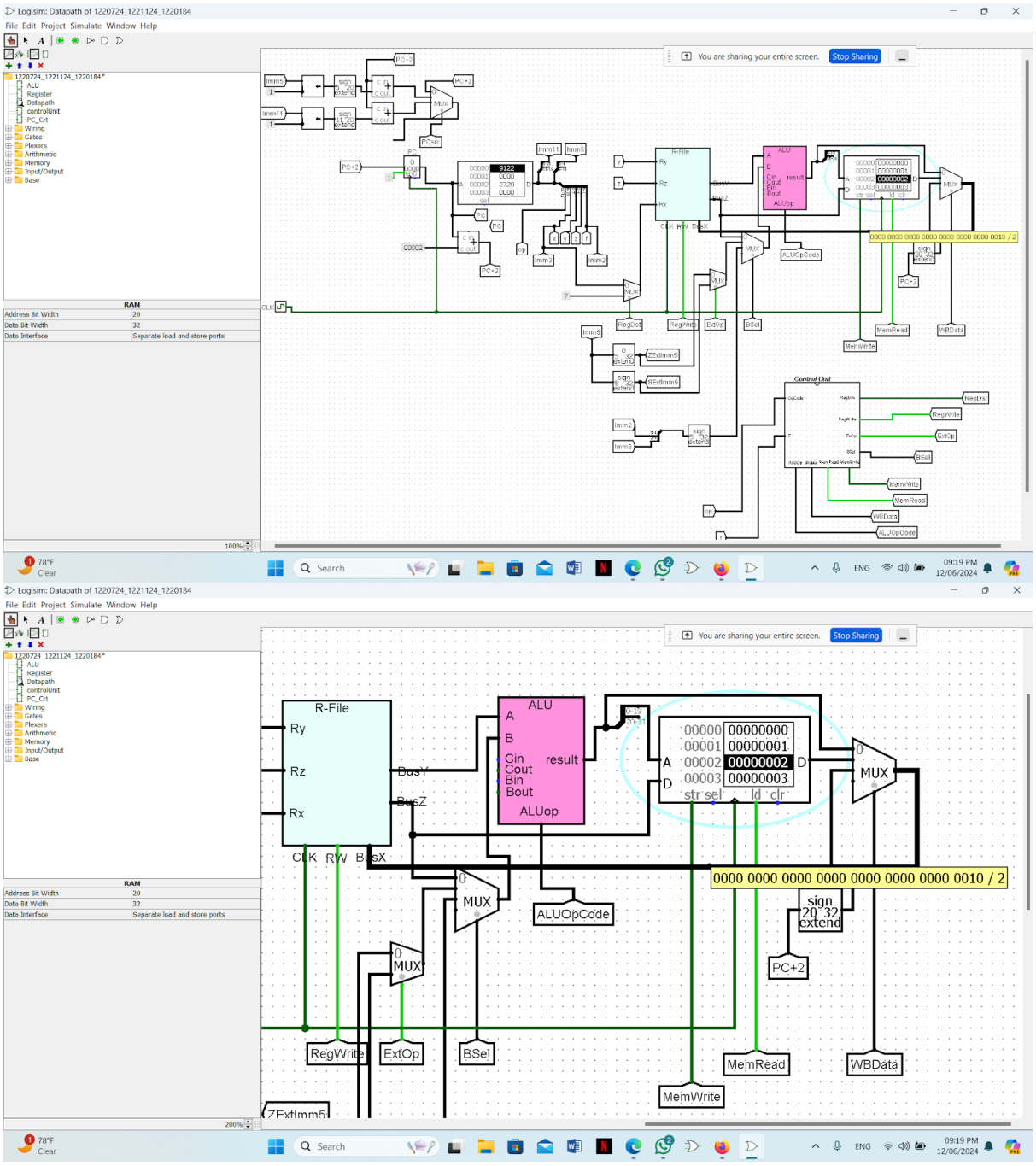
PC Instruction:

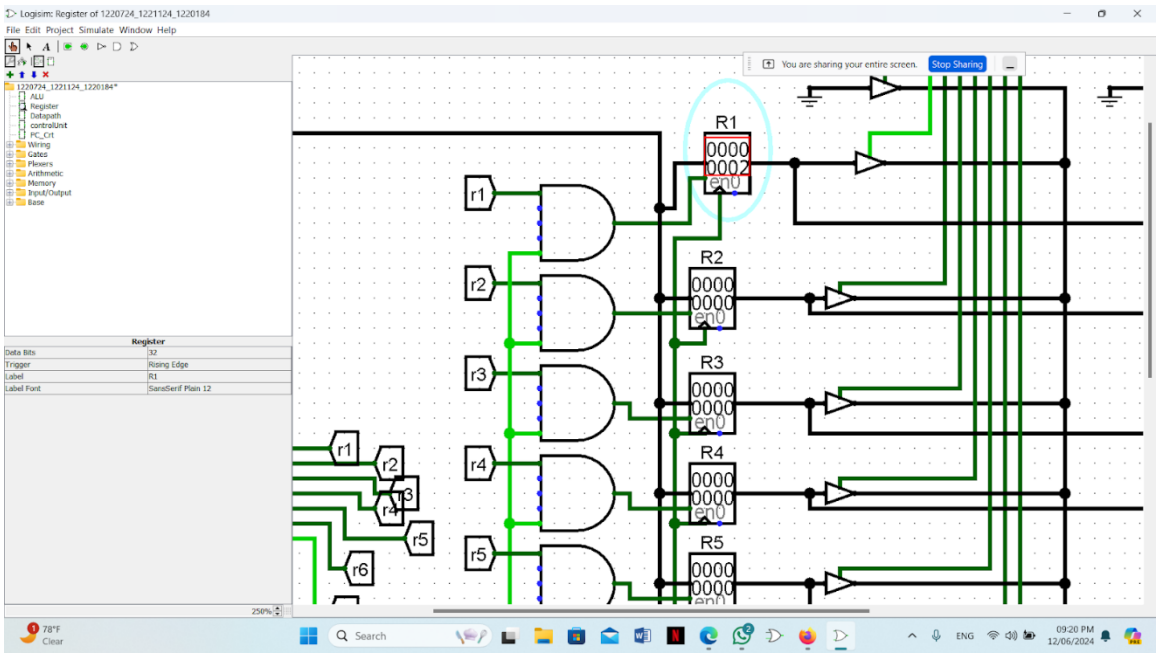


Filling Data Memory:

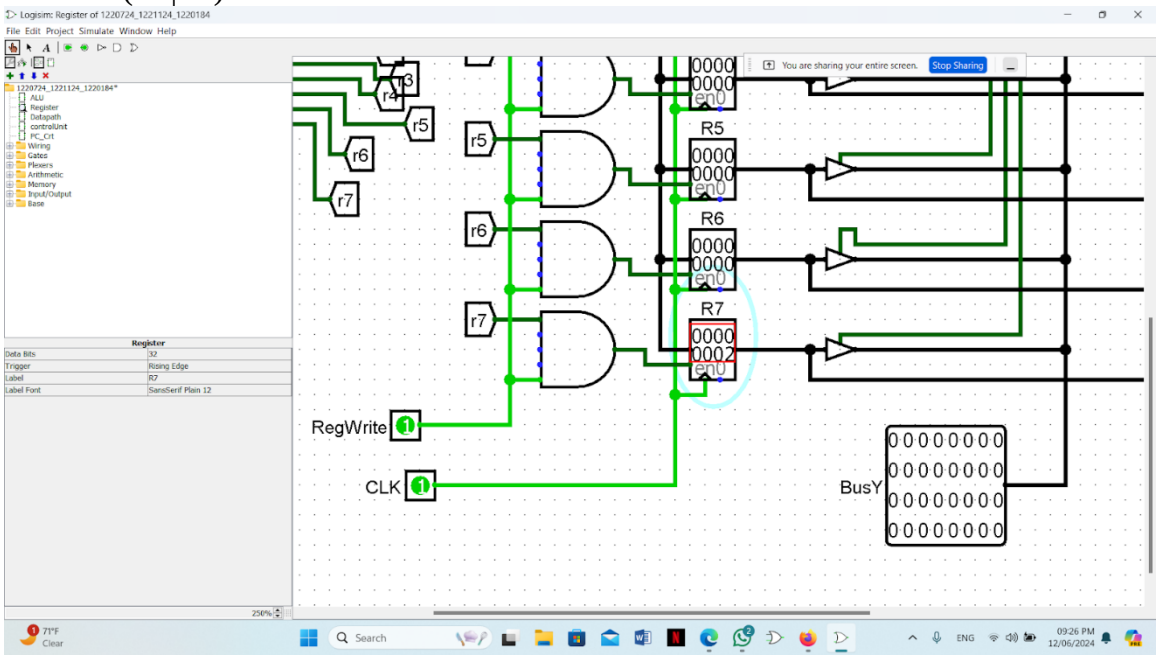


LOAD #2 to R1

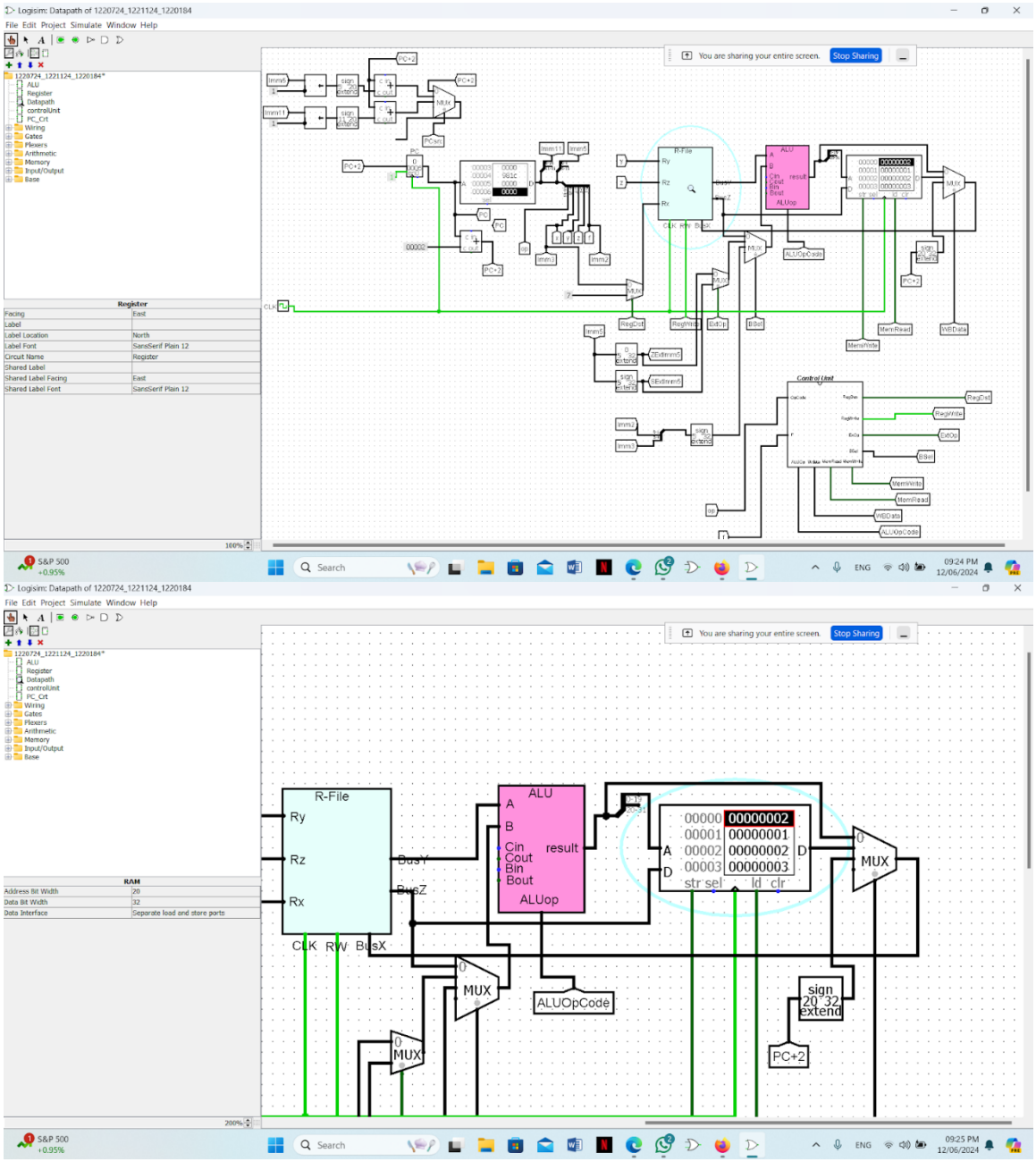

Logisim: Datapath of 1220724_1221124_1220184
File Edit Project Simulate Window Help
You are sharing your entire screen.
Stop Sharing
R-File
Ry
Rz
Rx
BusY
BusZ
CLK RW BusX
ALU
A
B
Cin
Cout
Bin
Bout
result
ALUop
ALUOpCode
MUX
RegWrite
ExtOp
BSel
MemRead
MemWrite
WBData
PC+2
sign 20_32 extend
0000 0000 0000 0000 0000 0000 0000 0010 / 2
00000 00000000
00001 00000001
00002 00000002
00003 00000003
str sel ld clr
RAM
Address Bit Width 20
Data Bit Width 32
Data Interface Separate load and store ports


Logisim: Register of 1220724_1221124_1220184
File Edit Project Simulate Window Help
You are sharing your entire screen.
Stop Sharing
R1
R2
R3
R4
R5
r1
r2
r3
r4
r5
r6
Register
Data Bits 32
Trigger Rising Edge
Label R1
Label Font SansSerif Plain 12

ORI R7=(R1|R3)


Logisim: Register of 1220724_1221124_1220184
File Edit Project Simulate Window Help
You are sharing your entire screen.
Stop Sharing
R5
R6
R7
r4
r5
r6
r7
RegWrite
CLK
BusY
Register
Data Bits 32
Trigger Rising Edge
Label R7
Label Font SansSerif Plain 12

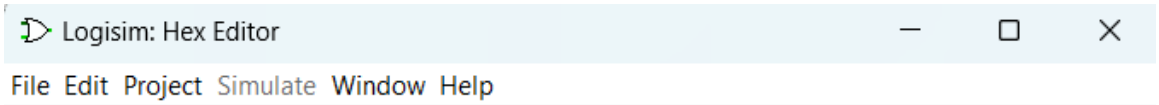
Store R7→ MEM


R-File
Ry
Rz
Rx
BusY
BusZ
CLK RW BusX
ALU
A
B
Cin
Cout
Bin
Bout
result
ALUop
00000 00000002
00001 00000001
00002 00000002
00003 00000003
str sel ld clr
MUX
ALUOpCode
sign 20 32 extend
PC+2

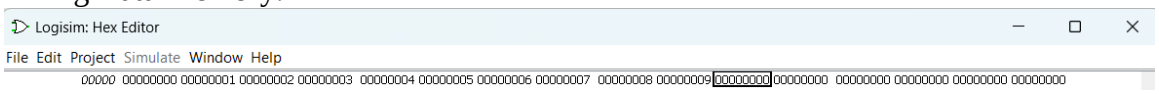
Testing SLTI Instruction:

Code:
LW R1, R2, Imm5
SLTI R7, R1, Imm5
SW Imm3, R0,R7, Imm2
Binary code to Hex:
1001 0001 0010 0010 → 0x9122
0100 1111 0010 0000 → 0x4F20
1001 1000 0001 1100 → 0x981C

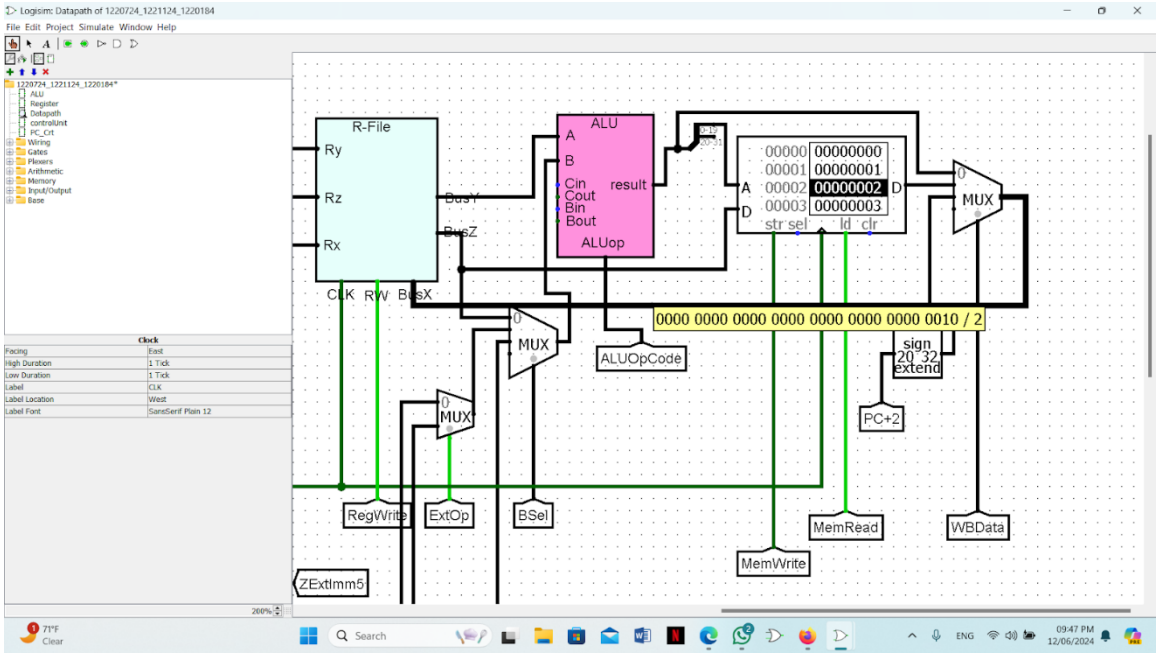
PC Instruction:

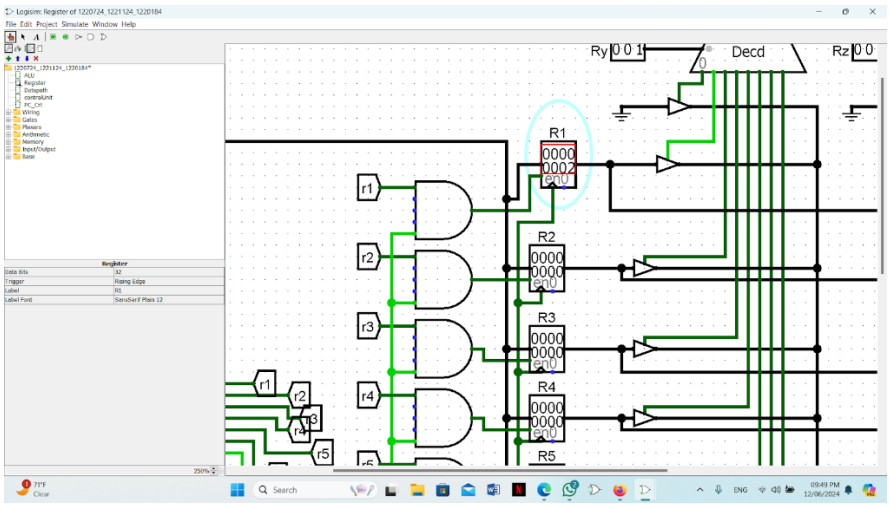


Filling Data Memory:

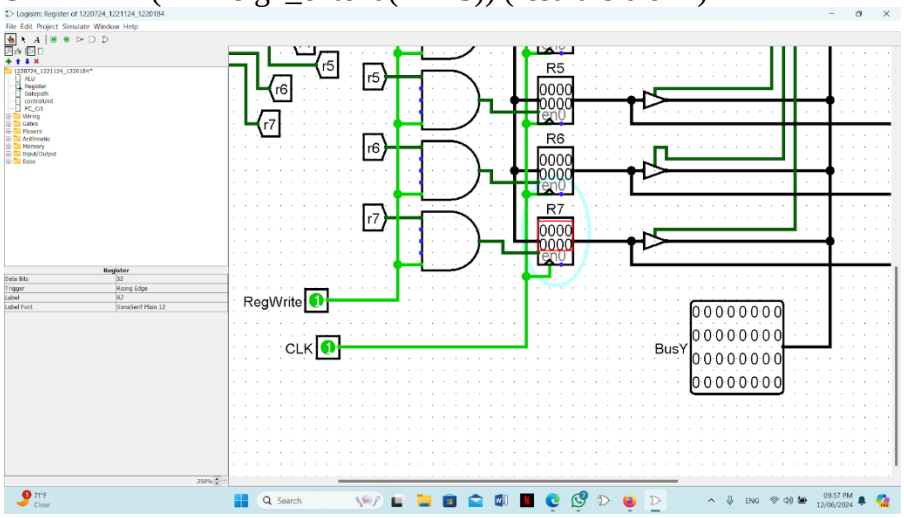


LOAD #2 to R1

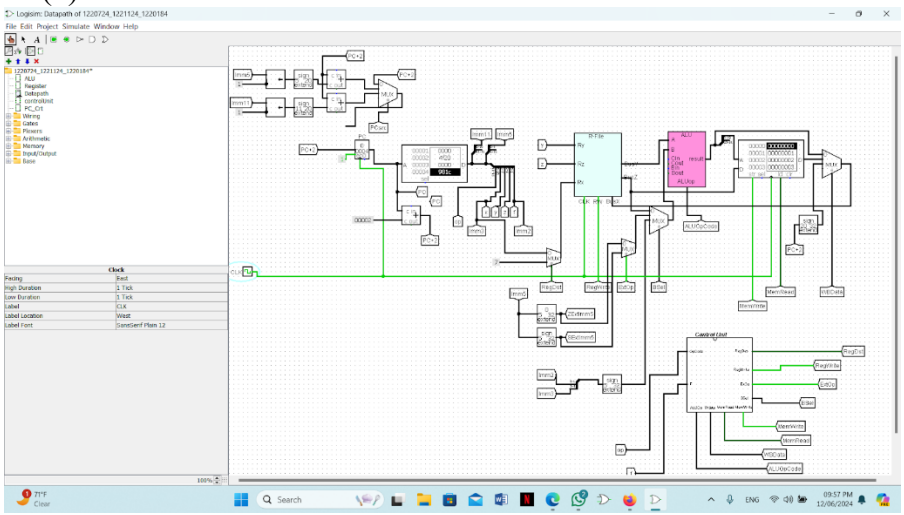




SLTI R7 = (R1 < sign_extend(Imm5)) (result is 0 or 1)



R7(0) → MEM



Testing AND Instruction:

Code:

LW R1, R2, Imm5

LW R3, R4, Imm5

SEQ R7, R1, R3, F

SW Imm3, R0,R7, Imm2

Binary code to Hex:

1001 0001 0010 0010 → 0x9122

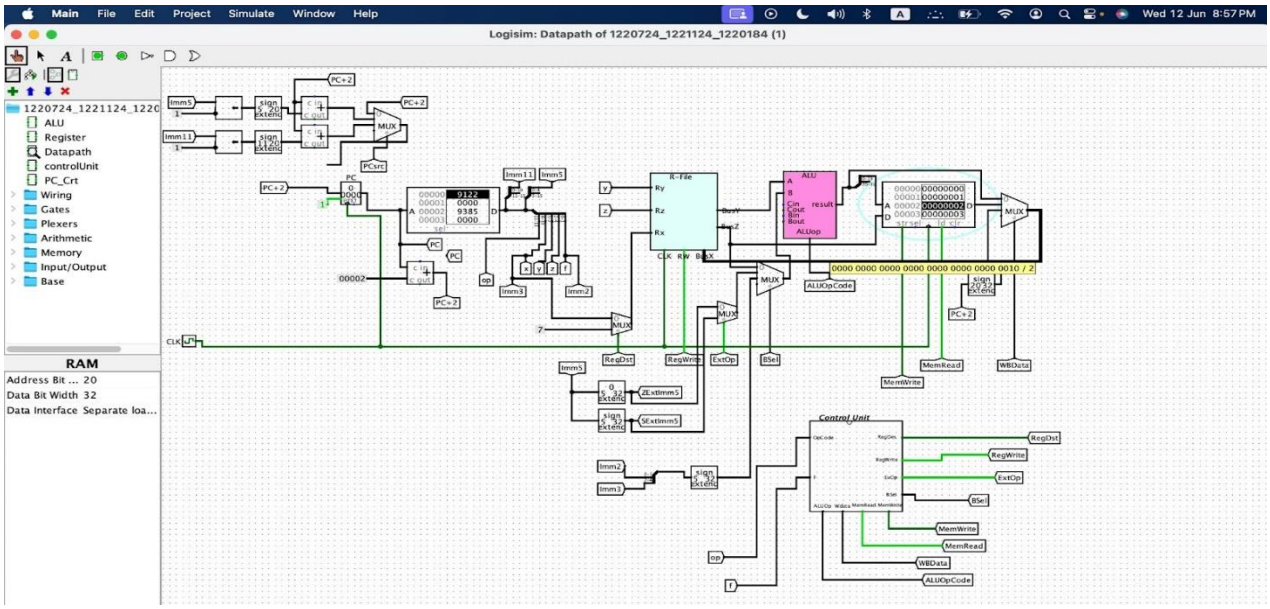
1001 0011 1000 0101 → 0x9385

0000_0111_0010_1100→0X72C

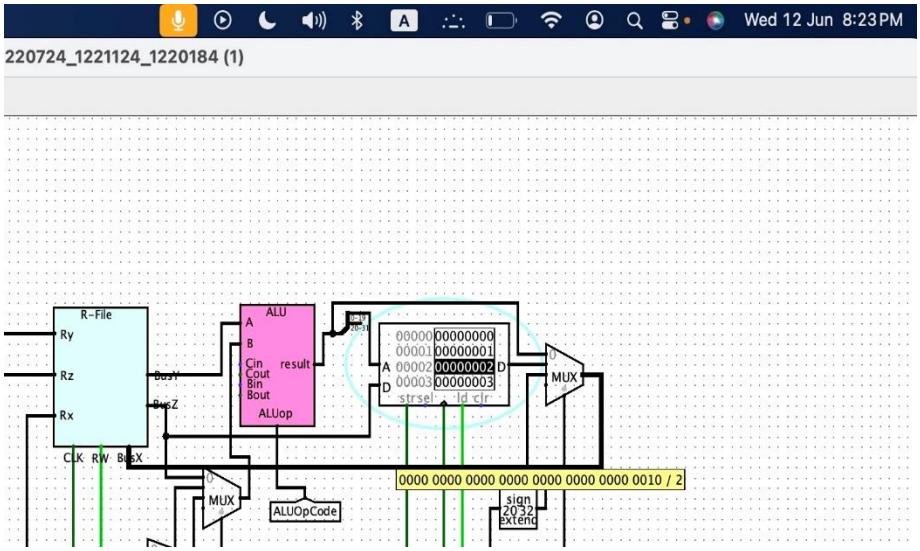
1001 1000 0001 1100 → 0x981C

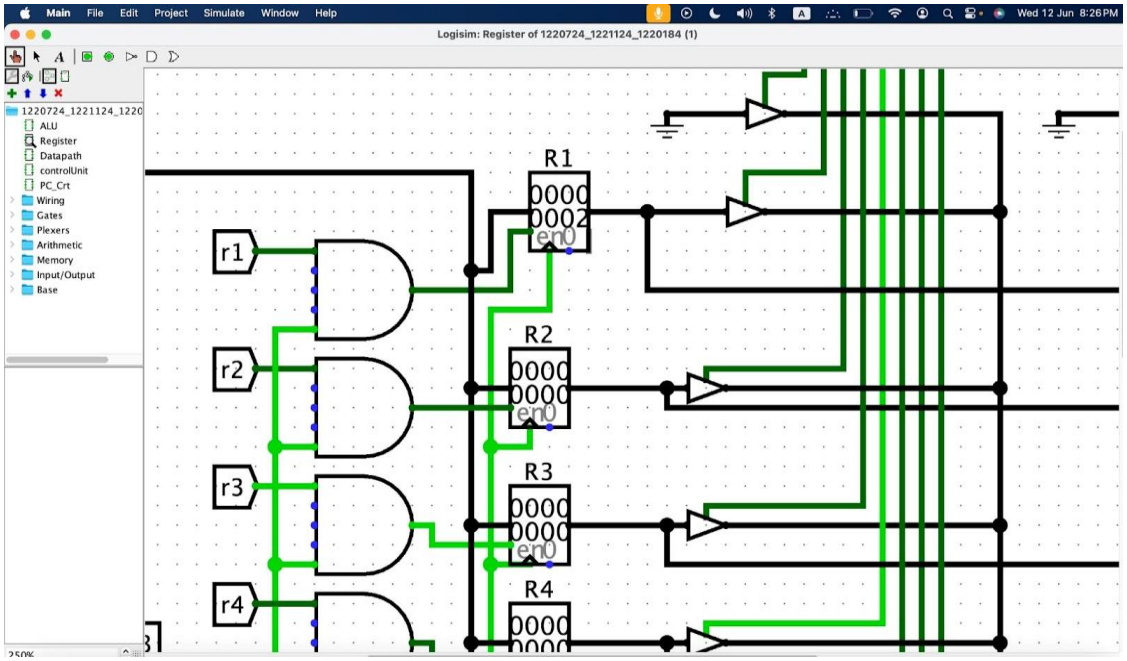
PC instruction

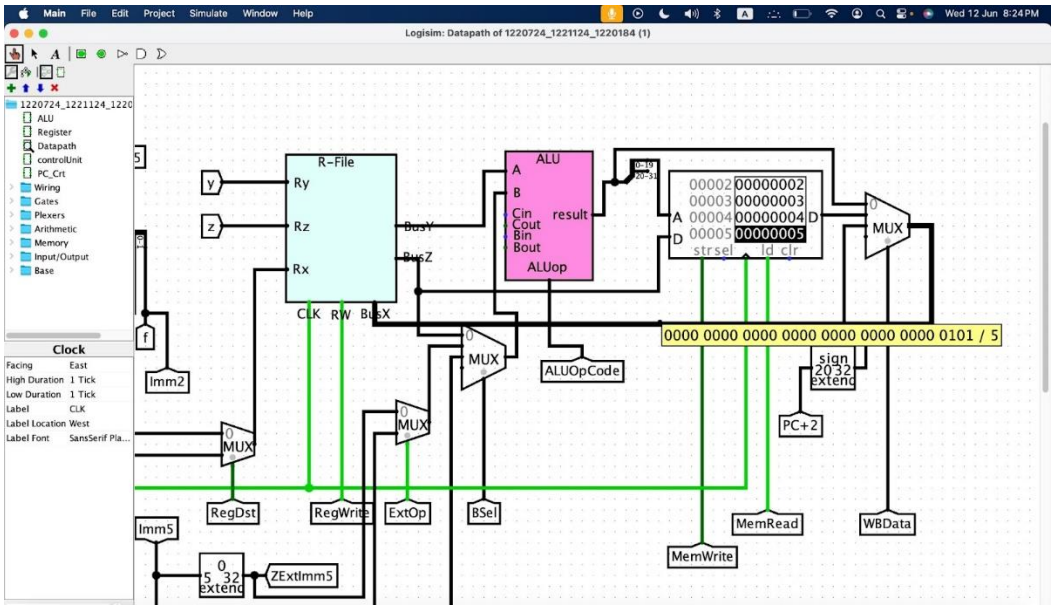
Filling Data memory



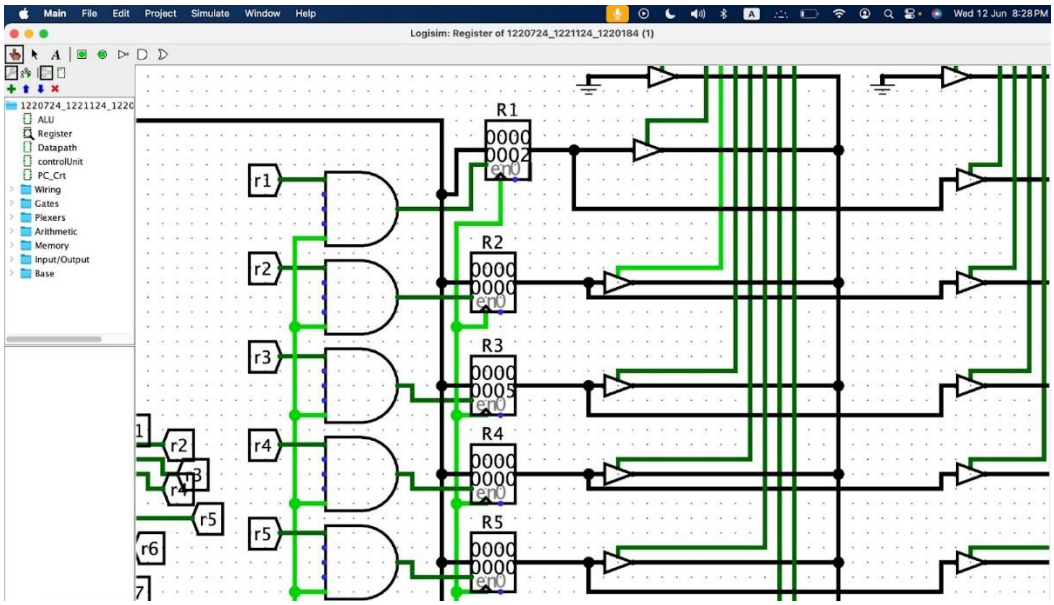
Load #2 from Data memory to R1 :

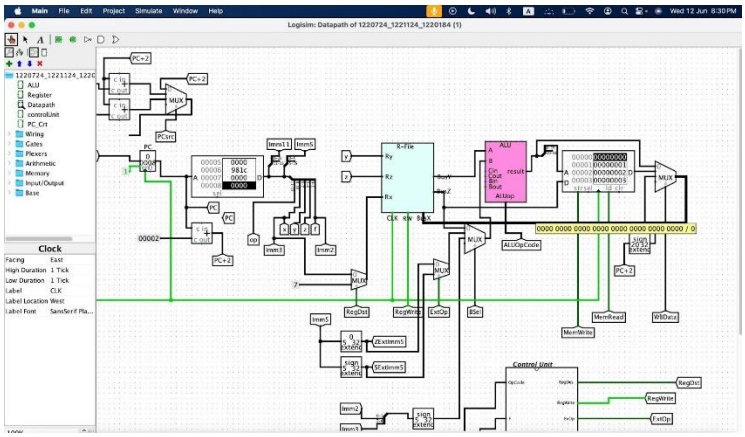






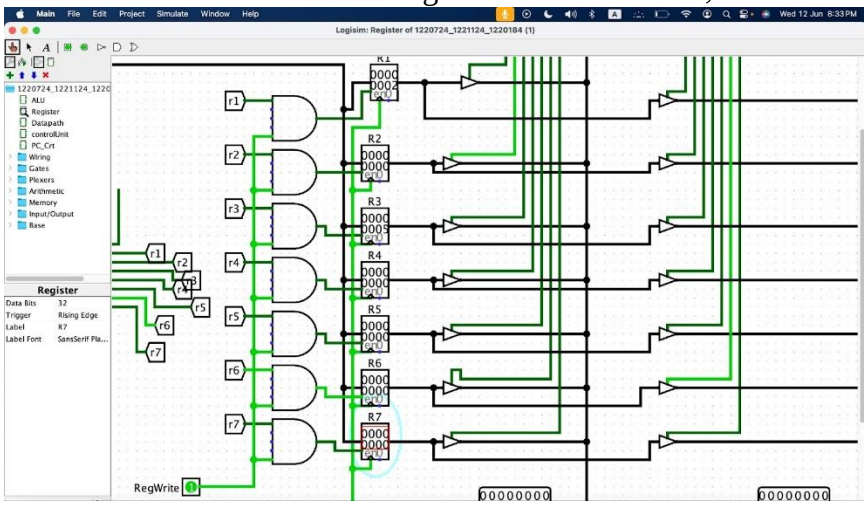
Load #5 from Data memory to R3 :



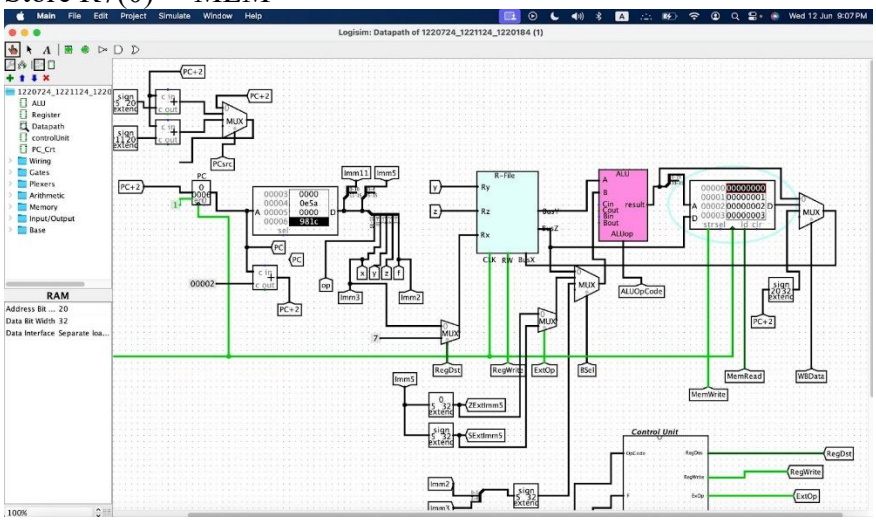


AND Rx = Ry & Rz

Result is zero because the Anding between 2 and 5 =0 ,we stored the result in R7



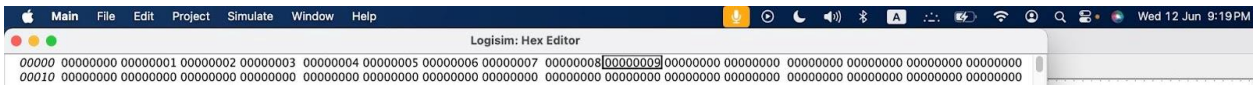
Store R7(0)→ MEM

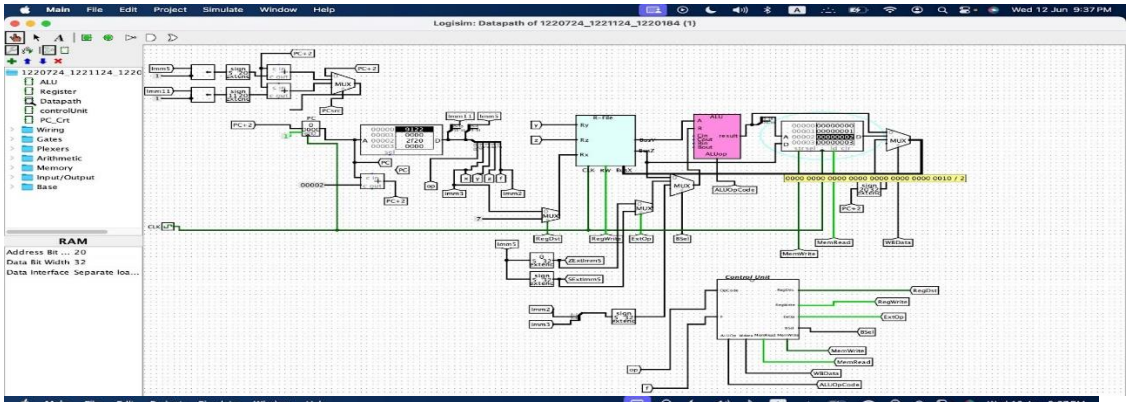


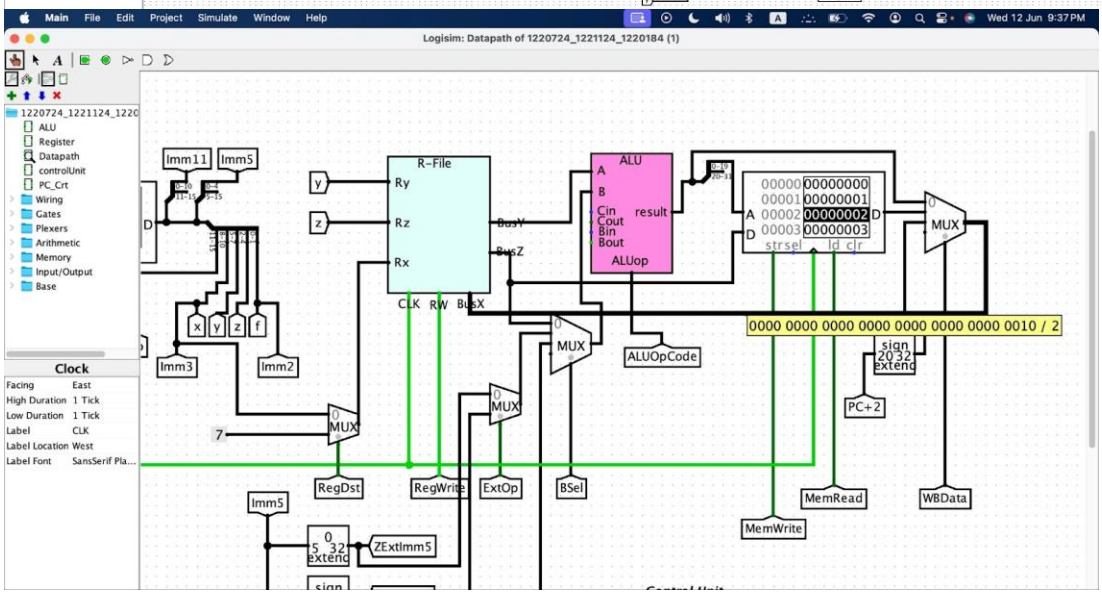
Testing XORI Instruction:
Code:
LW R1, R2, Imm5
ORI R7, R1, Imm5(00000)
SW Imm3, R0,R7, Imm2
Binary code to Hex:
1001 0001 0010 0010 → 0x9122
 0010_1111_0010_0000 → 2F20
1001 1000 0001 1100 → 0x981C

PC instruction

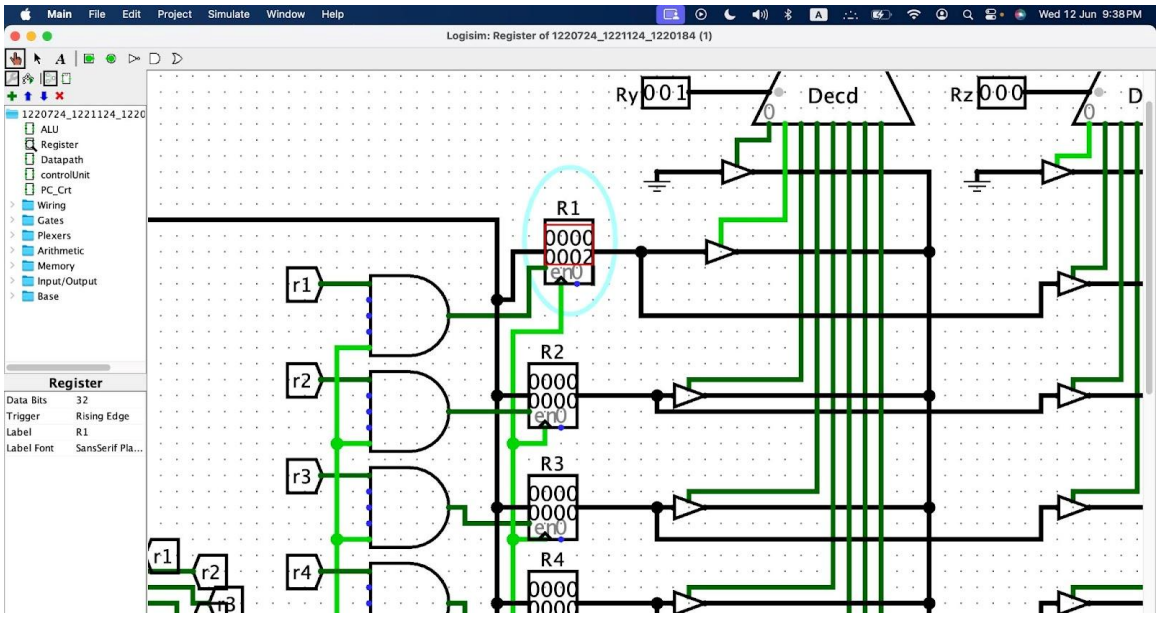
Filling data memory

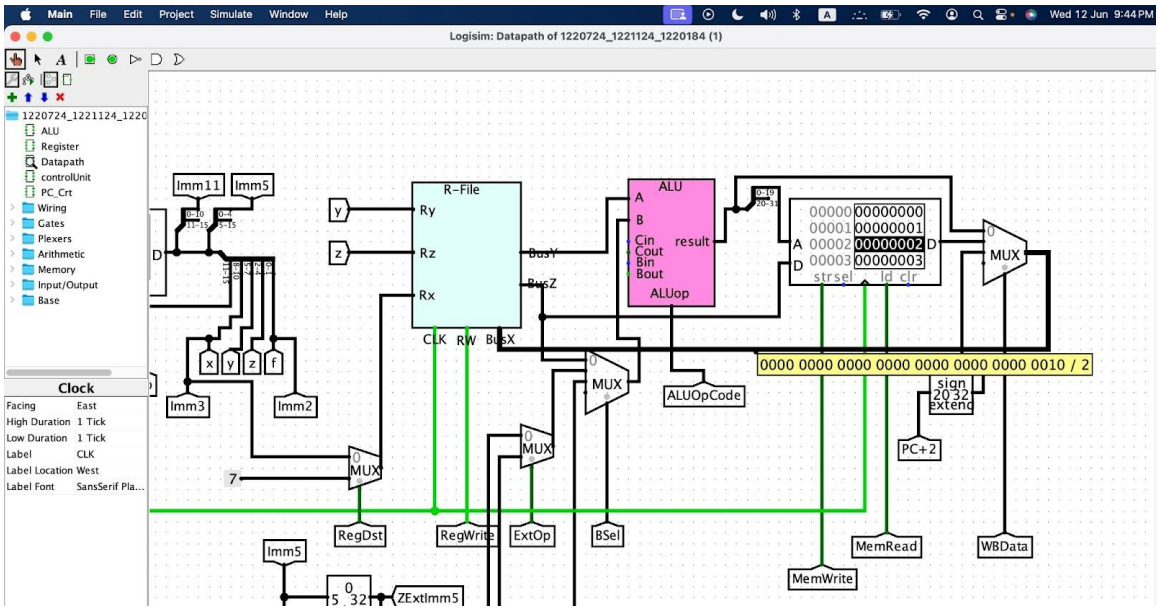






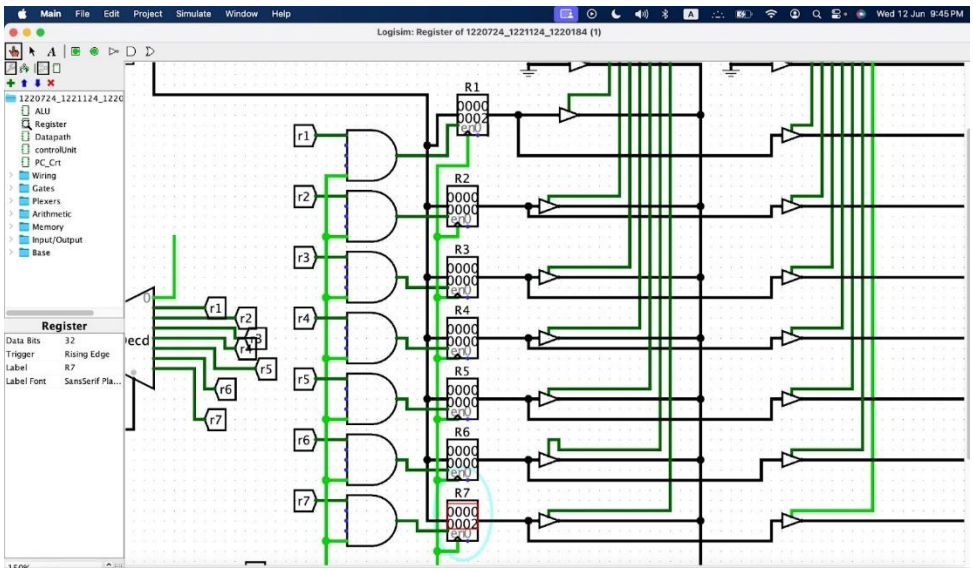
LOAD #2 to R1


Logisim: Register of 1220724_1221124_1220184 (1)
Ry 001
Decd
Rz 000
R1
0000
0002
en0
R2
R3
R4
r1
r2
r3
r4
Register
Data Bits 32
Trigger Rising Edge
Label R1
Label Font SansSerif Pla...

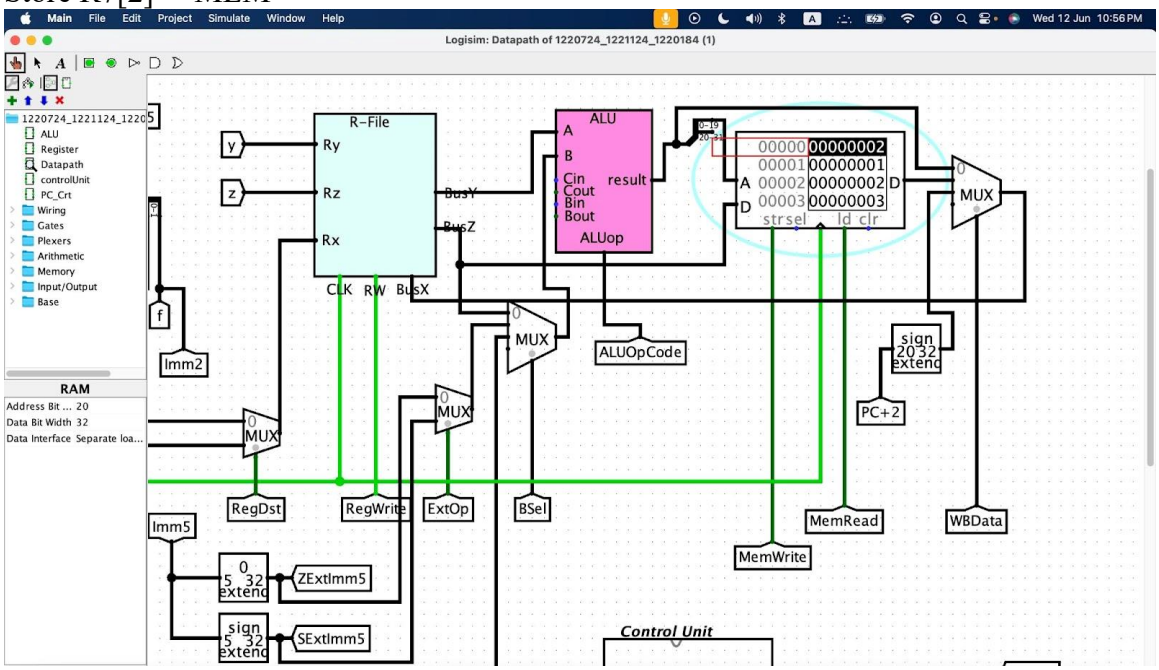

Logisim: Datapath of 1220724_1221124_1220184 (1)
Imm11
Imm5
R-File
Ry
Rz
Rx
BusY
BusZ
CLK RW BusX
ALU
A
B
Cin
Cout
Bin
Bout
result
ALUop
00000 00000000
00001 00000001
00002 00000002
00003 00000003
strsel ld clr
MUX
0000 0000 0000 0000 0000 0000 0000 0010 / 2
sign 2032 extend
PC+2
ALUOpCode
Imm3
Imm2
RegDst
RegWrite
ExtOp
BSel
MemRead
MemWrite
WBData
ZExtImm5
Clock
Facing East
High Duration 1 Tick
Low Duration 1 Tick
Label CLK
Label Location West
Label Font SansSerif Pla...

XORI R7 = R1 ^ zero_extend(Imm5)

R1 XOR 00000 ..result→ R1=2



Store R7[2]→ MEM

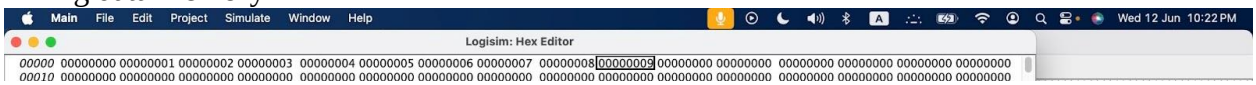


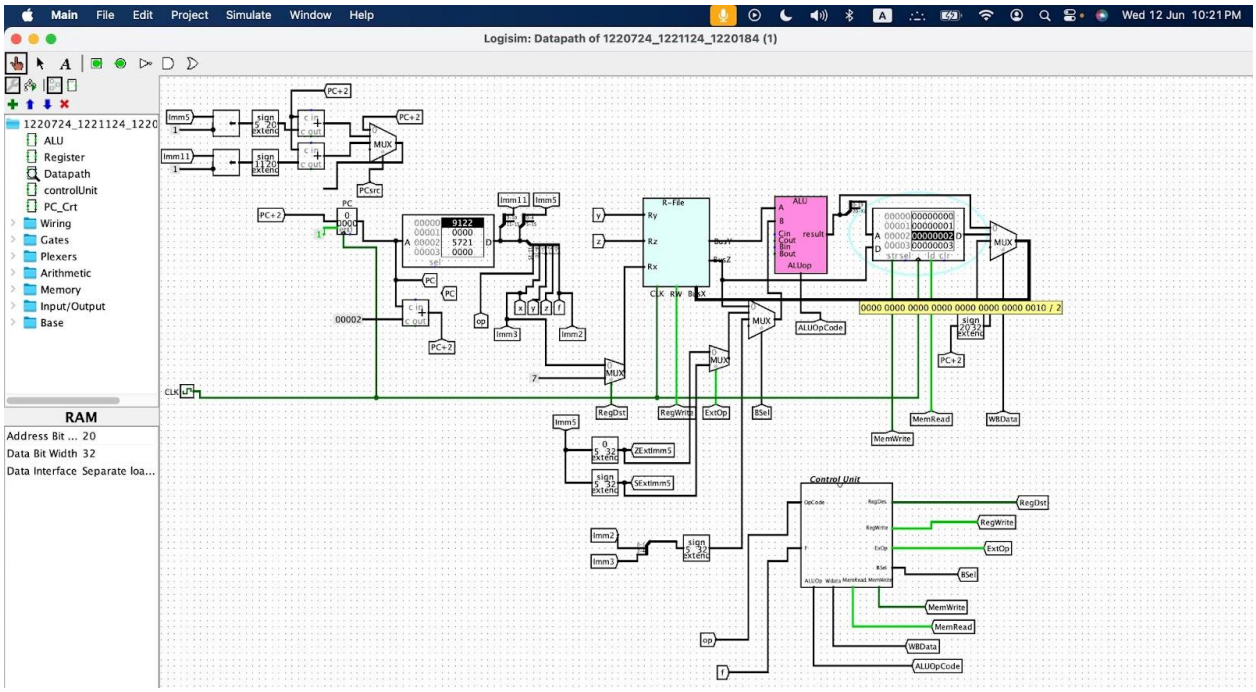
Testing SLL Instruction:

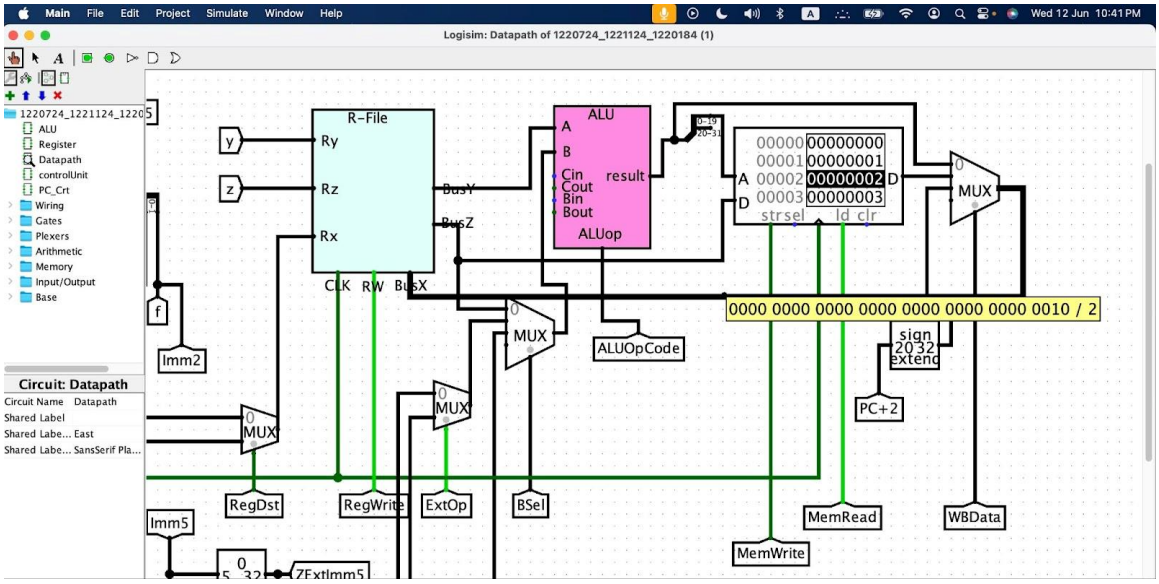
Code:
LW R1, R2, Imm5
SLL R7, R1, Imm5(00001)
SW Imm3, R0,R7, Imm2
Binary code to Hex:
1001 0001 0010 0010 → 0x9122
0101_0111_0010_0001 →5720
1001 1000 0001 1100 → 0x981C
010→100 logical shift left (2→4)

PC instruction

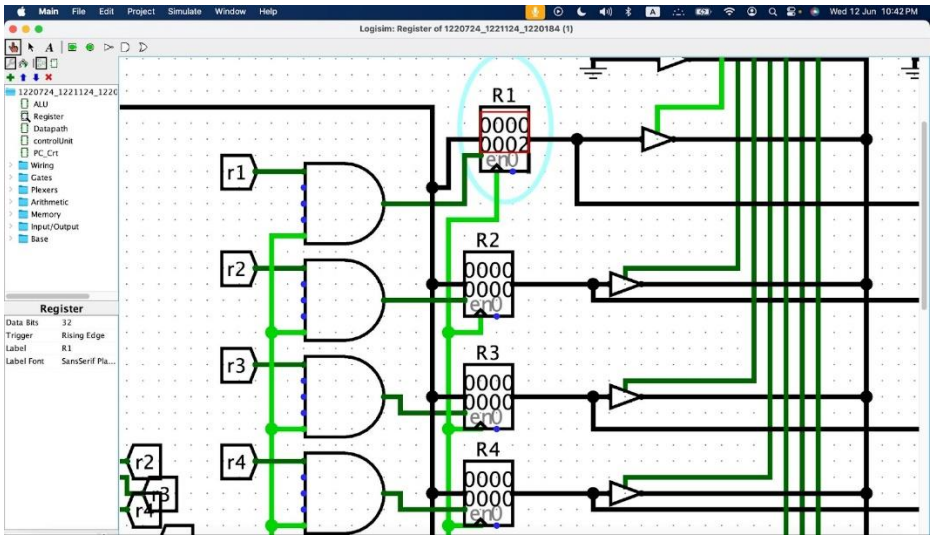
Filling data memory

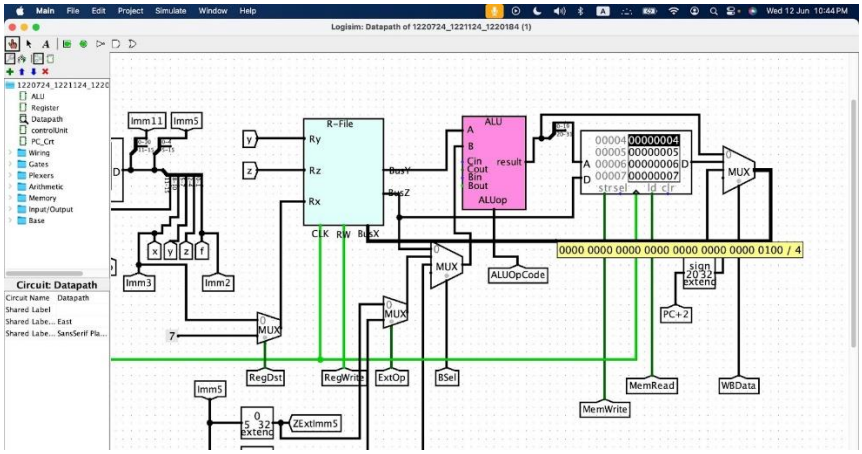




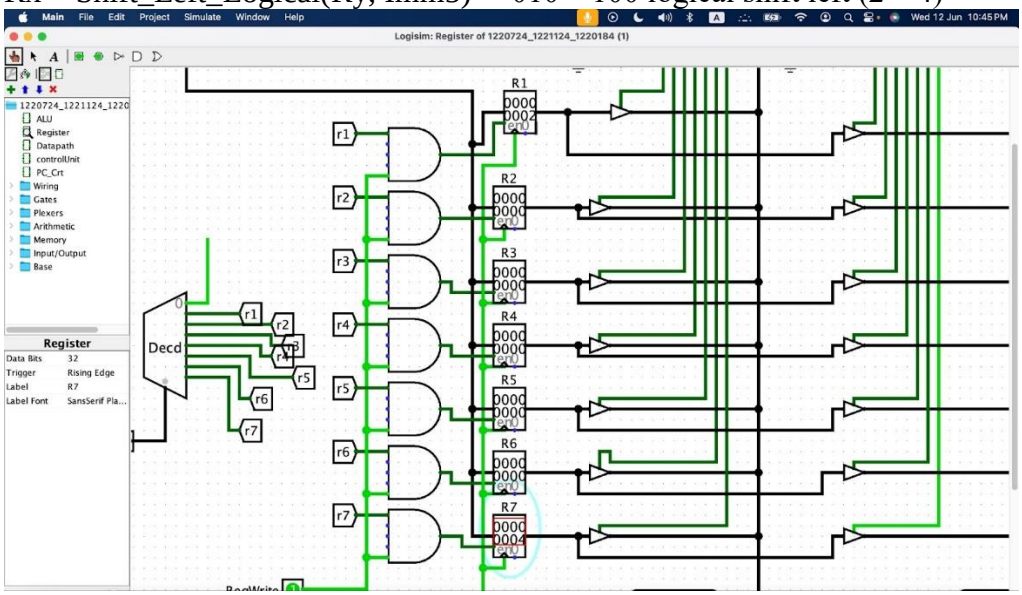

Logisim: Datapath of 1220724_1221124_1220184 (1)
R-File
ALU
ALUop
ALUOpCode
MUX
RegDst
RegWrite
ExtOp
BSel
MemRead
MemWrite
WBData
PC+2
Imm5
Imm2
0000 0000 0000 0000 0000 0000 0000 0010 / 2

LOAD #2 to R1

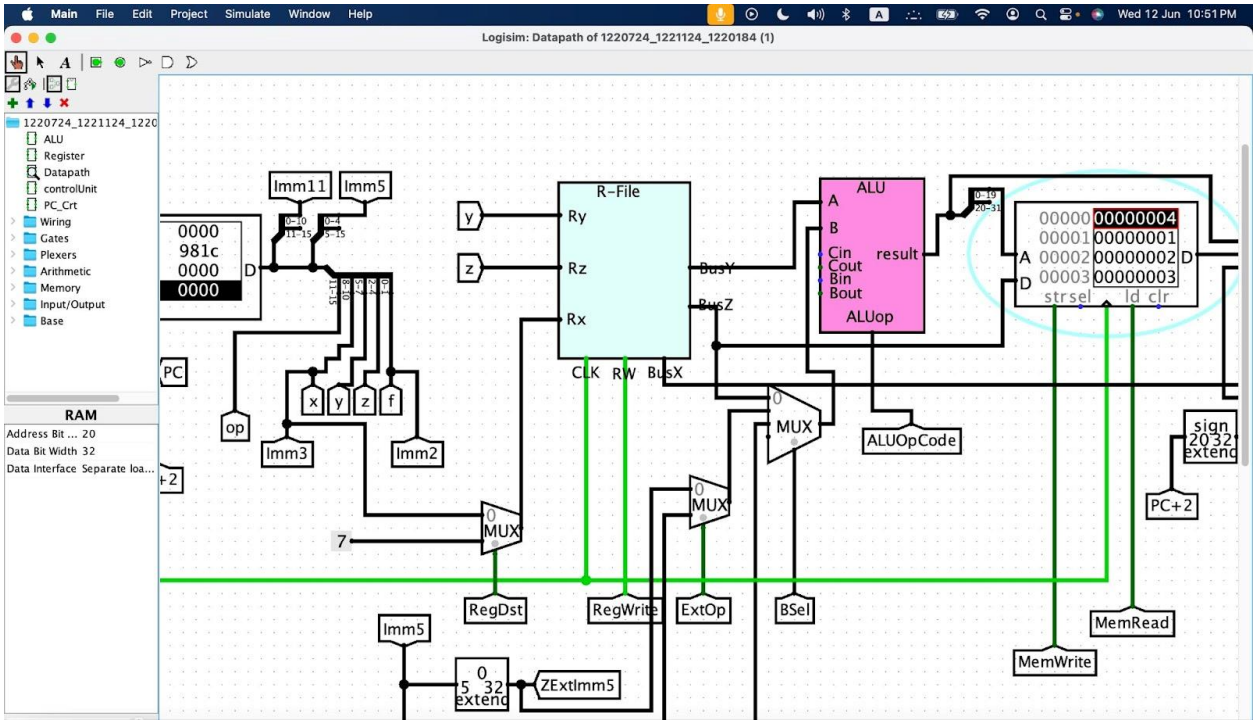

Logisim: Register of 1220724_1221124_1220184 (1)
R1
R2
R3
R4
r1
r2
r3
r4


Logisim: Datapath of 1220724_1221124_1220184 (1)
R-File
ALU
ALUop
ALUOpCode
MUX
RegDst
RegWrite
ExtOp
BSel
MemRead
MemWrite
WBData
PC+2
Imm5
Imm2
Imm3
0000 0000 0000 0000 0000 0000 0000 0100 / 4

Rx = Shift_Left_Logical(Ry, Imm5) →010→100 logical shift left (2→4)



Store R7[4]→ MEM



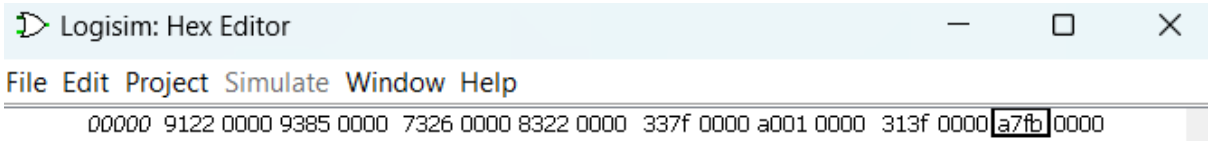
Simple while Loop Test with if-else statements for the branches and jump:

```
While(R1 != R3)
    if(R1>R3)
        R1=R3-1
    else
        R3=R1-1
```

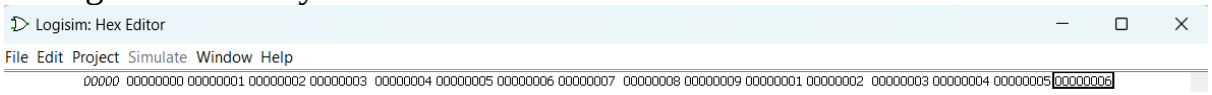
```
LW R1,R2,Imm5
LW R3,R4,Imm5
BEQ R3,R1,6
BLT R3,R1,2
ADDI R3,R3,-1
J1
ADDI R1,R1,-1
J -5
```

Binary code to Hex:
1001 0001 0010 0010 → 0x9122
1001 0011 1000 0101 → 0x9385
0111 0011 0010 0110 → 0x7326
1000 0011 0010 0010 → 0x8322
0011 0011 0111 1111 → 0x337F
1010 0000 0000 0001 → 0xA001
0011 0001 0011 1111 → 0x313F
1010 0111 1111 1011 → 0xA7FB

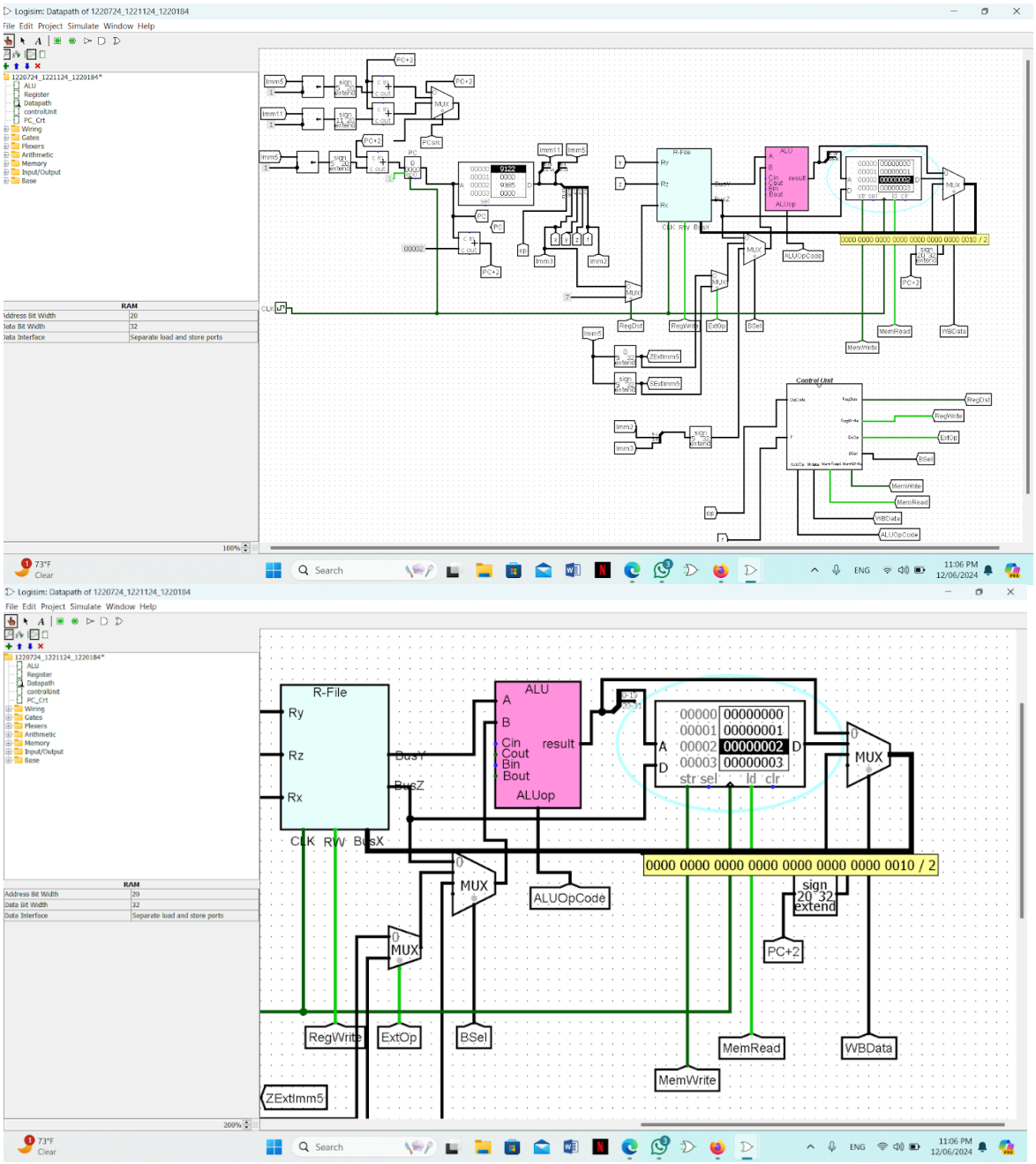
PC Instruction:



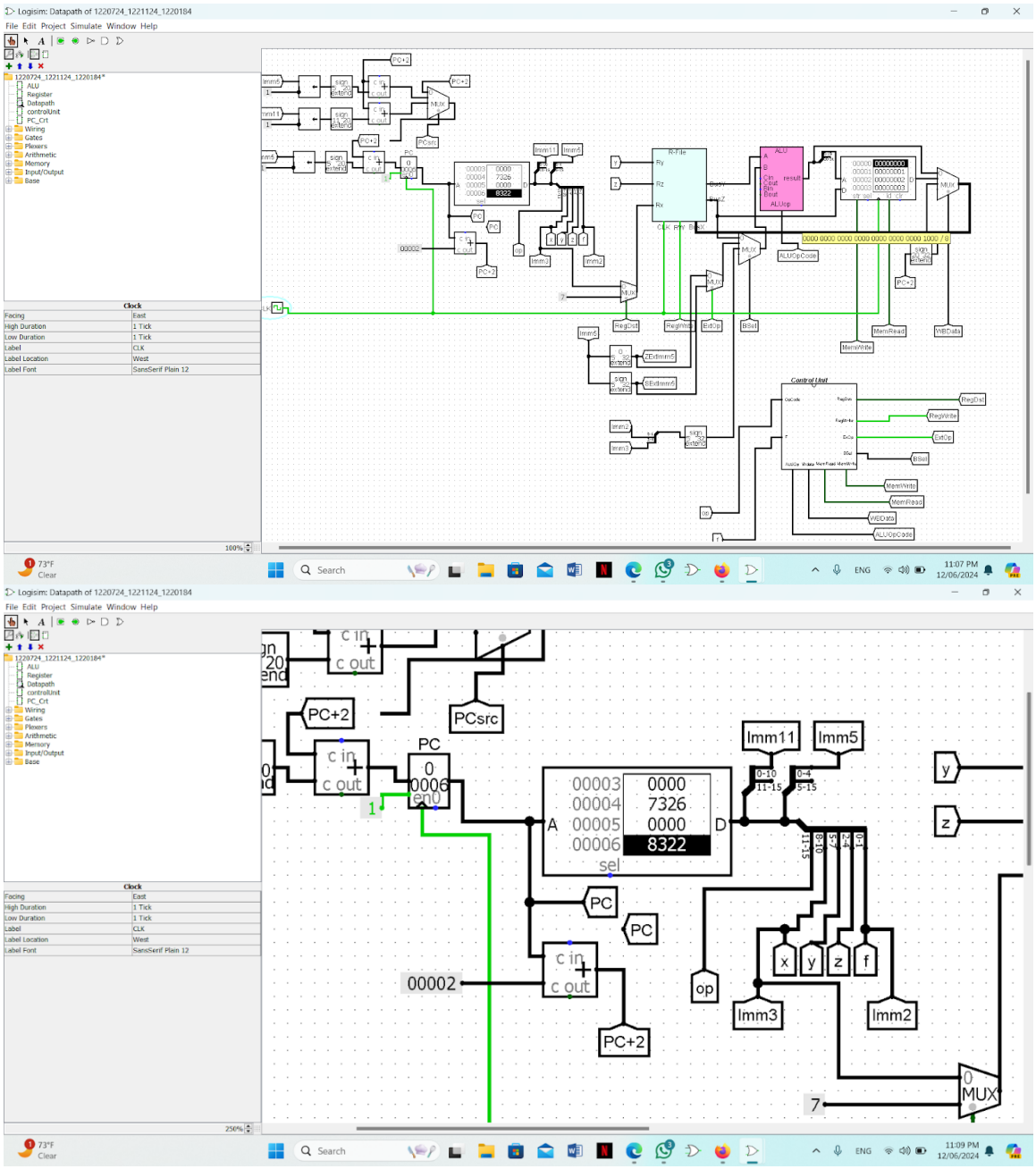
Filling Data memory:



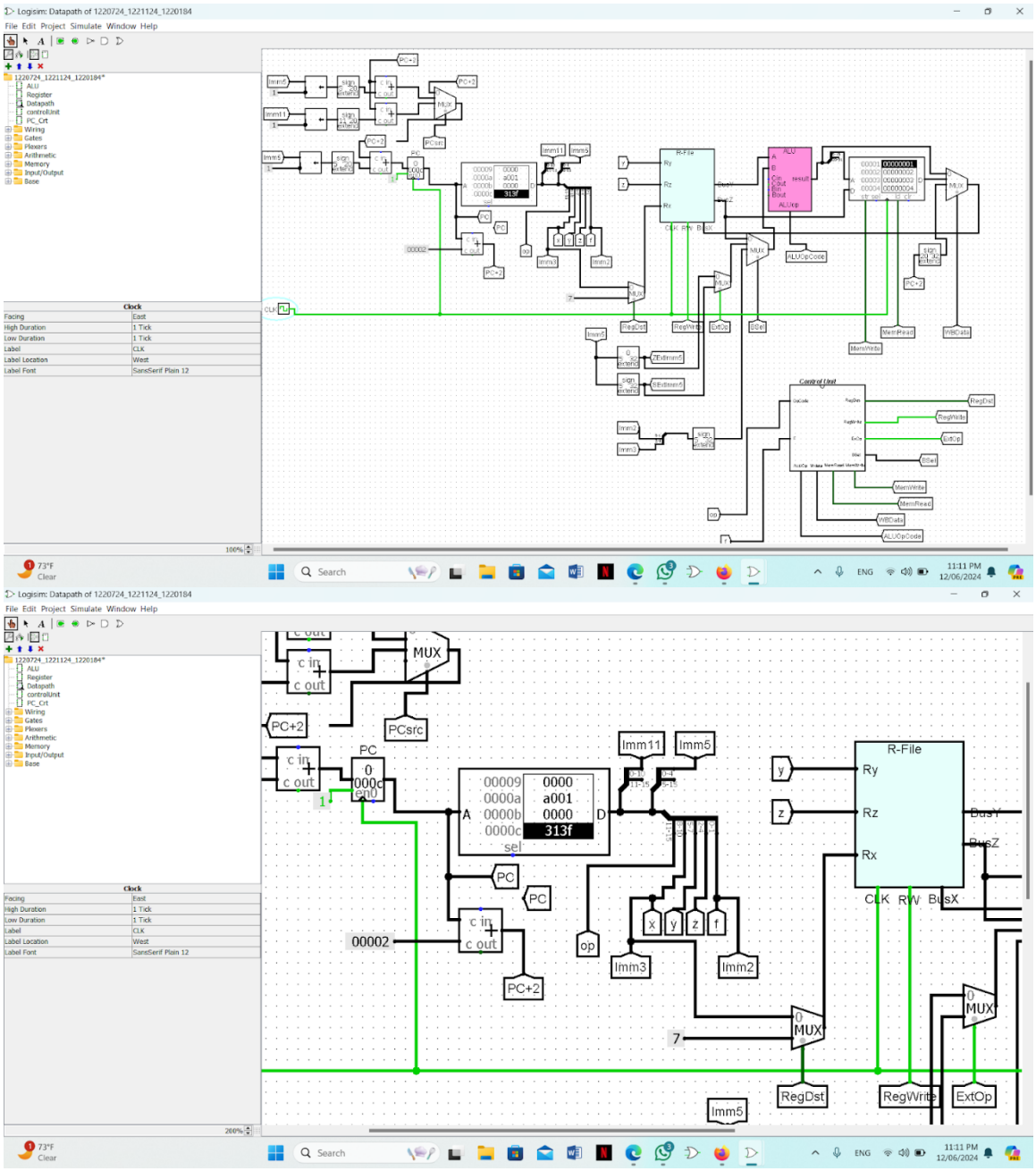
LOAD #2 to R1



here it branch the pc to = 0xooo6



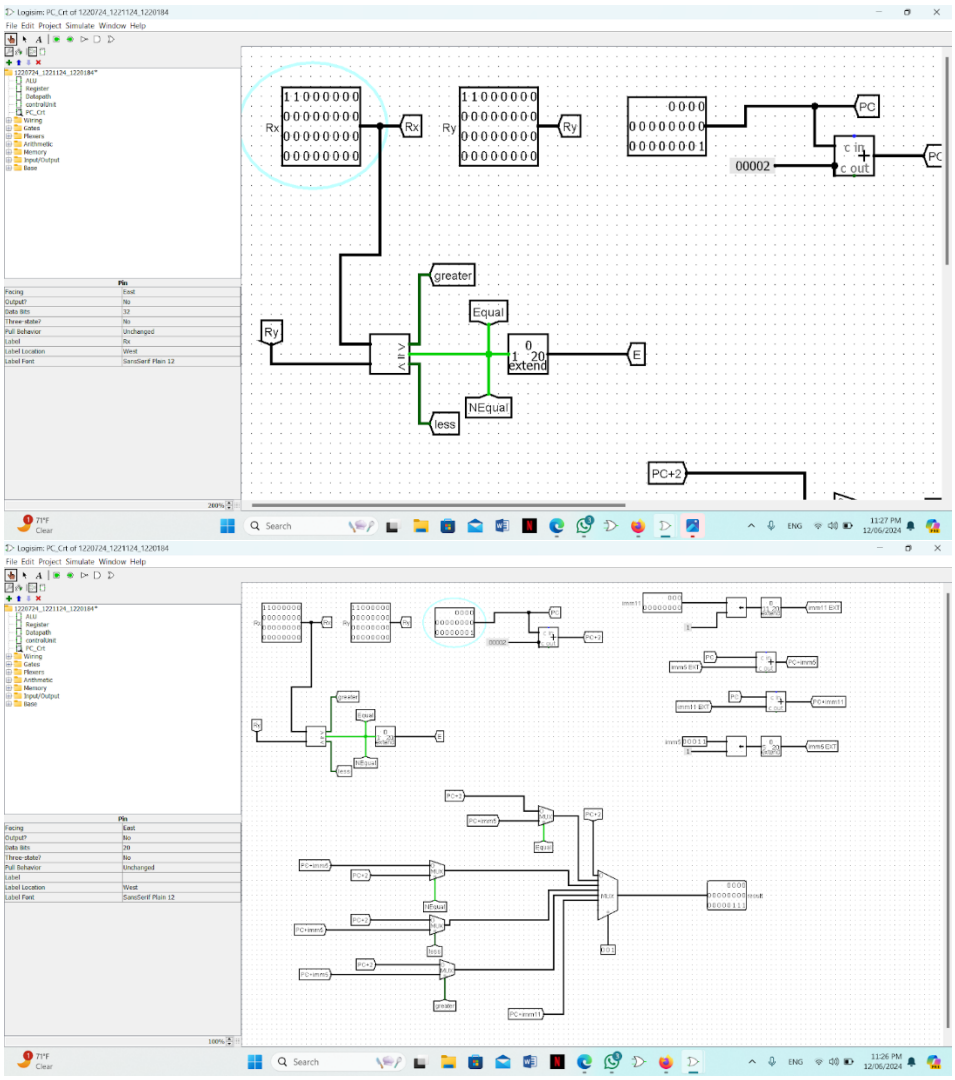
here we again give a clock it branches to pc=000c

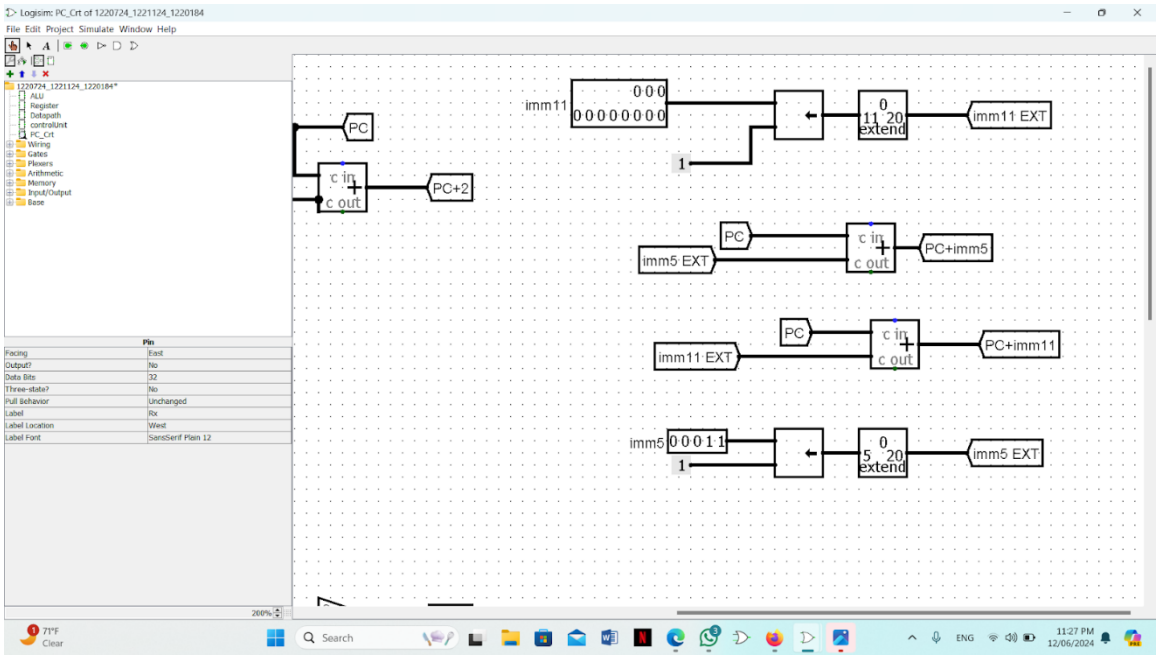


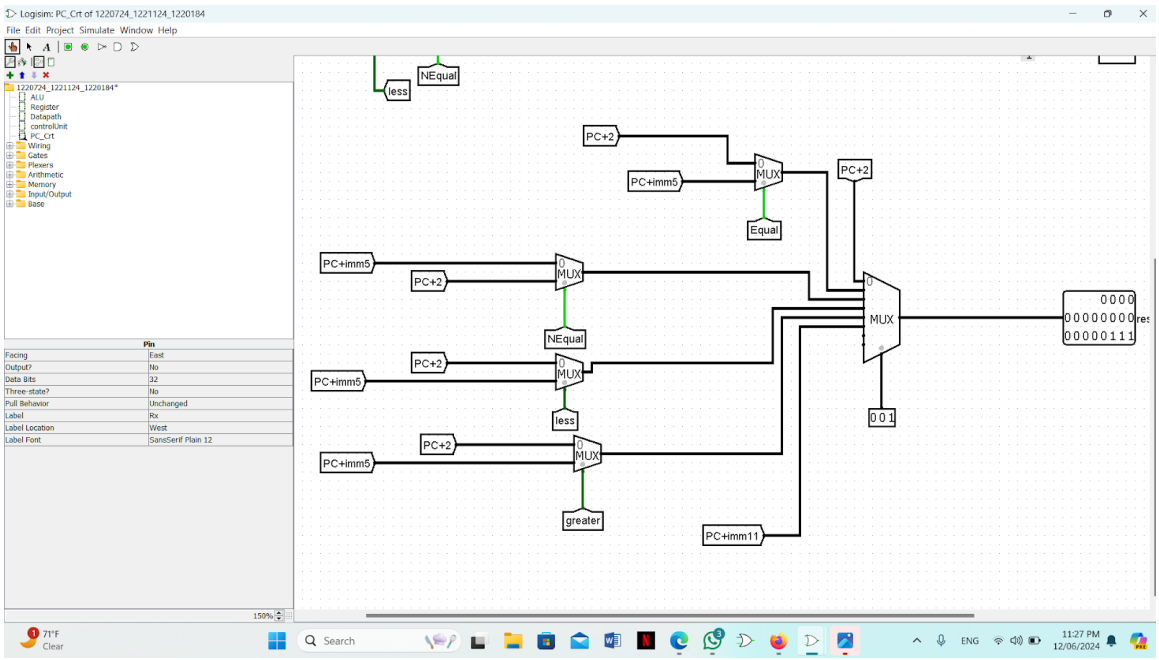
We will discuss the problem we faced below.

Issues and Limitations Part

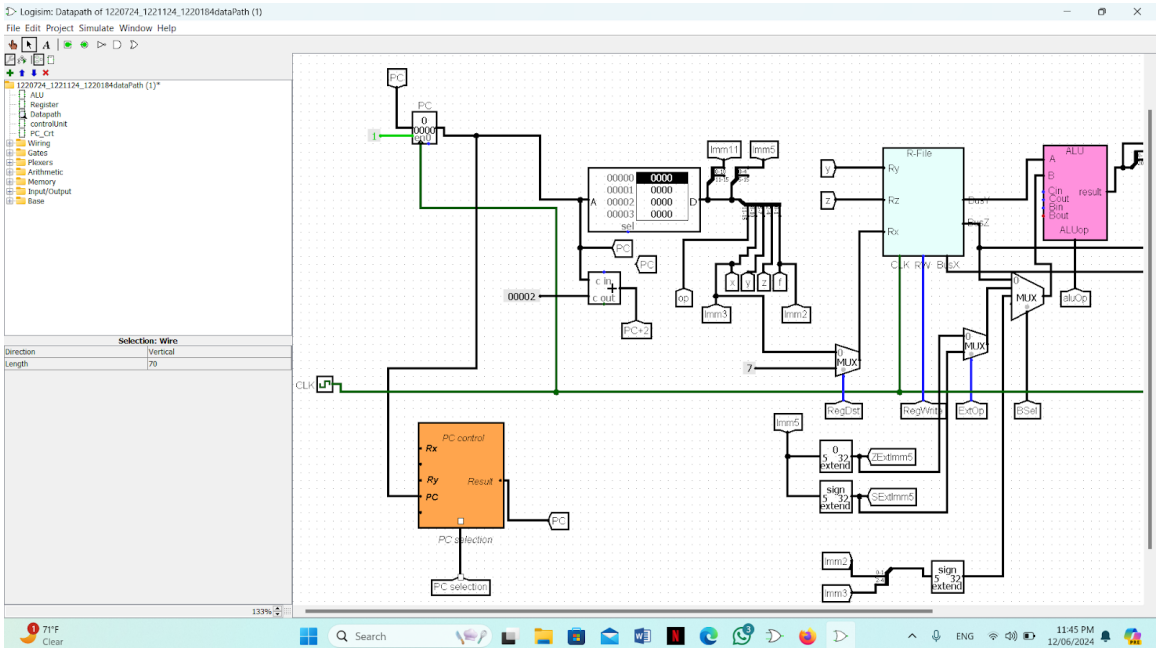
First of all, the pc component (PC control circuit) was working separately without the data path, once we added it to the data path it didn't work properly.
Maybe the problem was in connecting the Pc_control in the dataPath ,or maybe there was a logical error while connecting it ALU.
Alu consisting of many flags like carry flag ,zero flag, negative flag and overflow flag but we did not add to our ALU because the time didn't save us.




Logisim: PC_Crt of 1220724_1221124_1220184
File Edit Project Simulate Window Help
imm11
000
00000000
PC
c in
c out
PC+2
imm11 EXT
0
11 20
extend
1
imm5 EXT
PC+imm5
imm11 EXT
PC+imm11
imm5
00011
imm5 EXT
5 20
Search
ENG
11:27 PM
12/06/2024


Logisim: PC_Crt of 1220724_1221124_1220184
File Edit Project Simulate Window Help
less
NEqual
PC+2
PC+imm5
MUX
Equal
NEqual
less
greater
0000
00000000
00000111
001
PC+imm11
Search
ENG
11:27 PM
12/06/2024

so thats why our branch and jump instruction is not working(well its working in the pics i showed jump and branch but the PC counter in not all right)
we tried to add it but i didnt work here a pic of it:



Feedback

Overall the project was highly educational and engaging, we developed our knowledge in single cycle processor in computer organization as it allowed for a clear understanding of each element in the DataPath like the register file, ALU,and control unit.

All the group members worked together in an efficient way to submit this project ,so we used to meet in the campusand or do 'Google meet ' meetings to finish this work .