Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Computer Organization and Microprocessor

ENCS2380

Project Report

Single Cycle Processor Design

Prepared by :

Manar Zitawi **ID : 1221806**

Nawras Yaqoob **ID : 1221904**

Mohammed Saada **ID : 1221972**

Instructor : Dr. Abualseoud Hanani

Section : **2**

Date : 11 / 06 / 2024

Design and Implementation :

In this Project, we implemented a single cycle processor which can be able to implement and deal with set of instructions (ISA).

This processor contains of four basics component: Arithmetic and Logical Unit (ALU), Number of Registers (Registers File), Main Control Unit (CU), and Datapath.

1) Arithmetic and Logical Unit (ALU) :

This Component can implement all the arithmetic and logical operations which required in this processor, they are :

XOR, AND, OR, CAND, ADD, SEQ, NADD, SLT, SRA, SRL, SLL, and ROR.

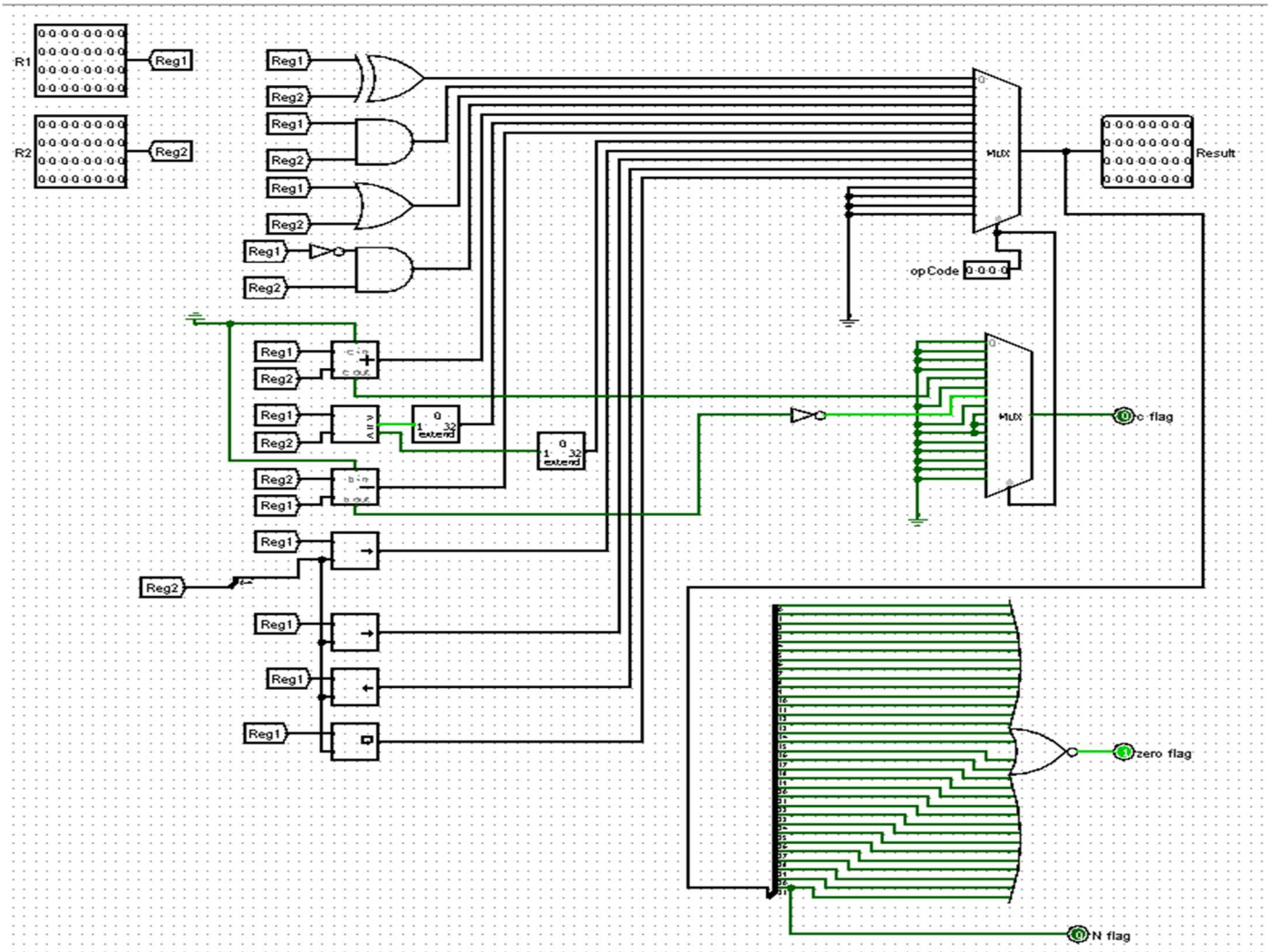


Figure 1 : ALU Implementation

As we show in Figure 1 (ALU implementation) above, this unit contain all the logical and arithmetic unit that we mentioned previously, and the result of all these operations are input for the Mux, this Mux depends on the opCode to determine which operation is required and give it's result as the final ALU result.

Additionally, we implement a simple design to give the ALU flags (carry flag (C flag), zero flag, sign flag (N flag)) these flags are output to the ALU, besides to the main result for the ALU.

2) Registers File :

In this component we bring all the Registers in this CPU together with some connections. This Registers are general purpose registers.

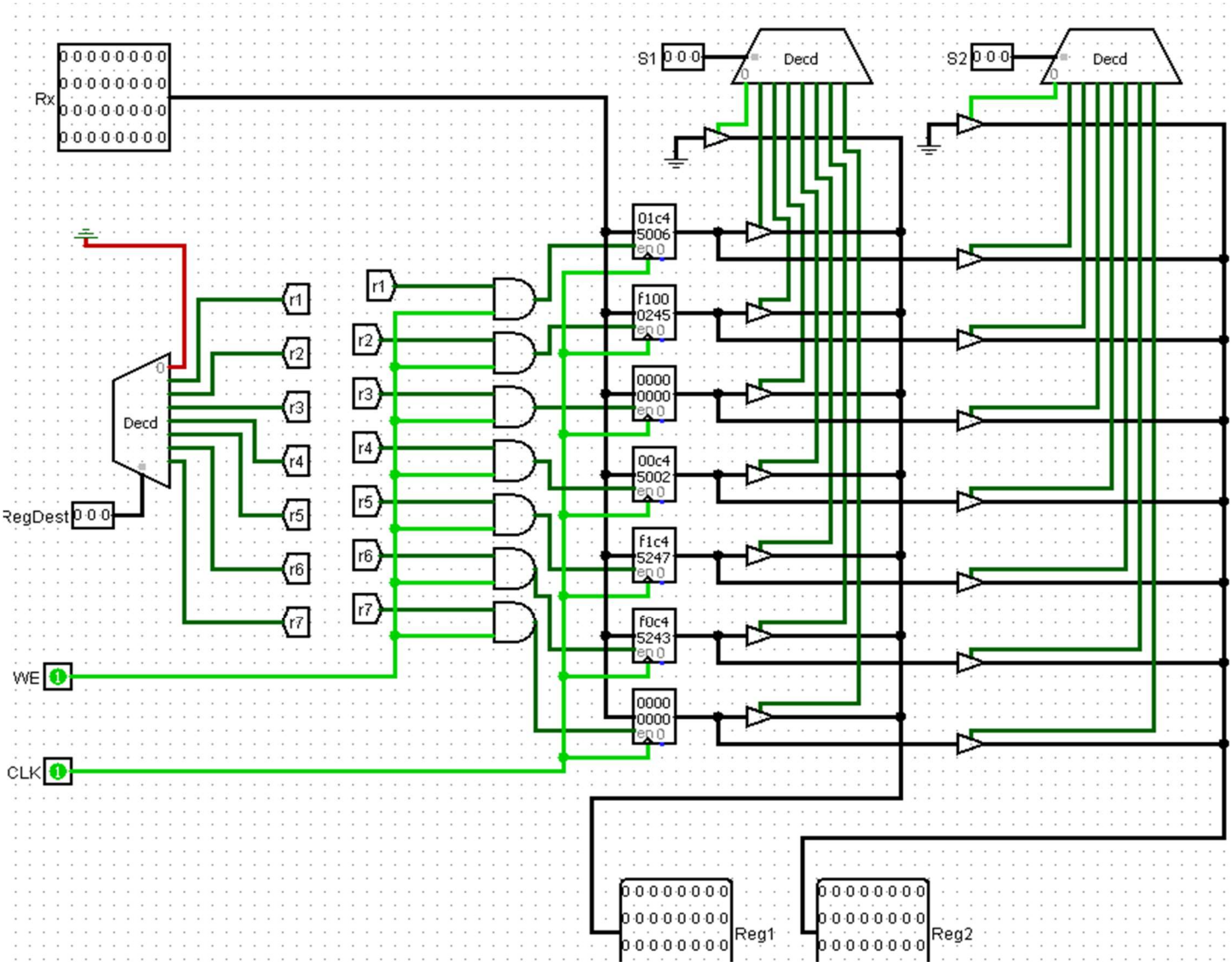


Figure 2 : Register s File Implementation

In the Figure 2 above that show the implementation of the Registers File clearly. This Registers File has an ability to deal with 8 Registers each one is 32-bit Register (R0 – R7) we need 3 bits to deal with these registers (000 – 111), while register R0 has no accessed to write or read from it (R0 always reads as 32-bit zeros), other registers (R1 – R7) can be accessed to read or write on them. In one clock cycle we can access one register to write and two registers to read, so we have a decoder to decode a 3-bits and determine which register to be written if the write enable is (1), otherwise we can't write on any of registers. In addition to this we have two decoders to decode and determine which registers to be readable on a time and connect the contents of these two registers with two outputs.

3) 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 20-bits register, because the address of the instructions memory is 20-bits [due to the Logisim limitations], the instructions memory is 20-bits addresses bus, and each cell is 16-bits because each instruction is 16-bits, while the data memory is 20-bits addresses bus and each cell is 32-bits because the data in registers is also 32-bits. The PC registers value increment by 2 at every clock cycle [PC + 2], because the instructions stored in the memory at the even addresses only, but when there is a branch (conditional) instruction if the condition is true the PC value changes as PC += sign_extend(Imm5), otherwise the PC increment by 2 (e.g. PC += 2), and when there is a jump (unconditional) instruction the PC changes as PC += sign_extend(Imm11).

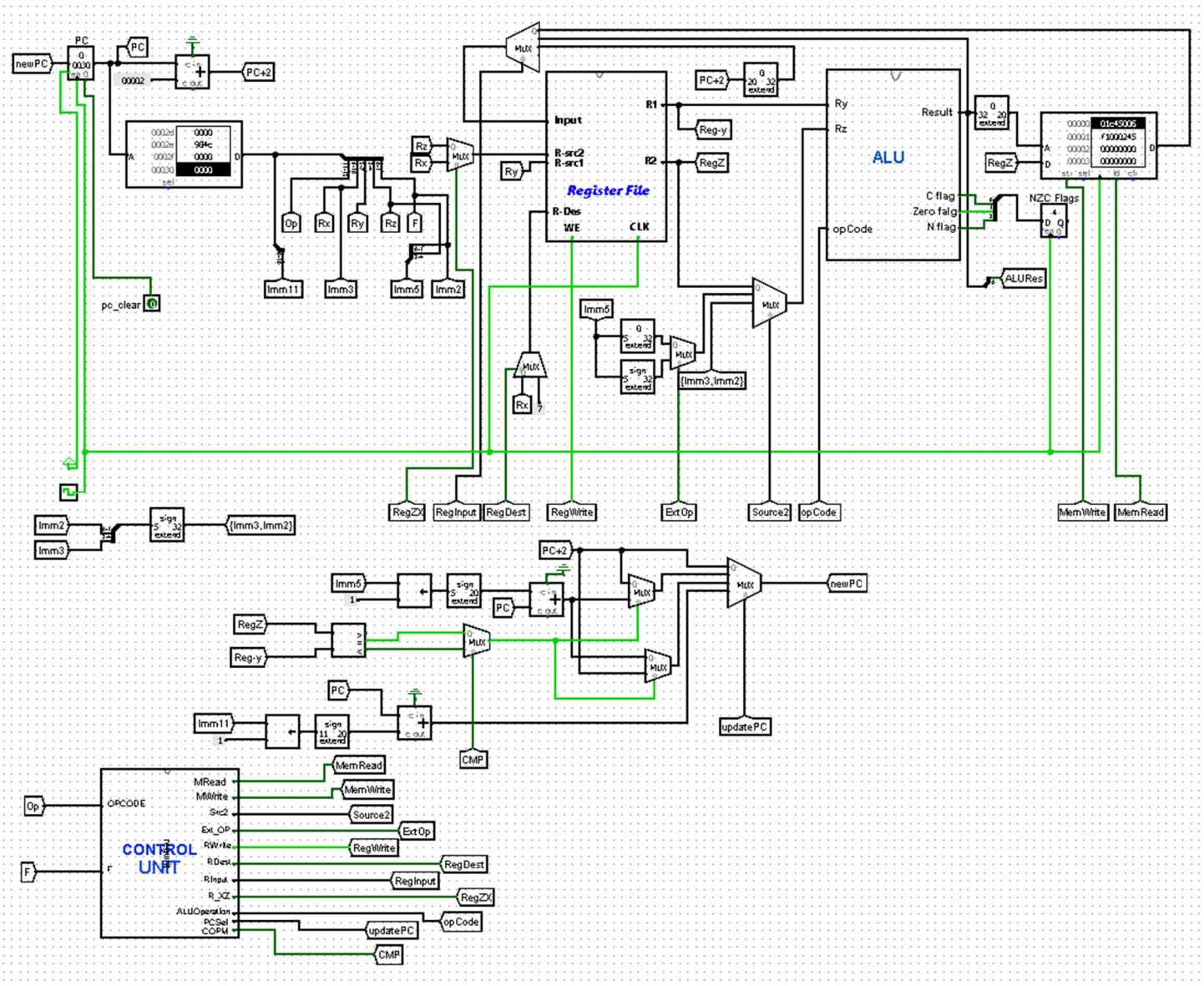


Figure 3 : Datapath Implementation

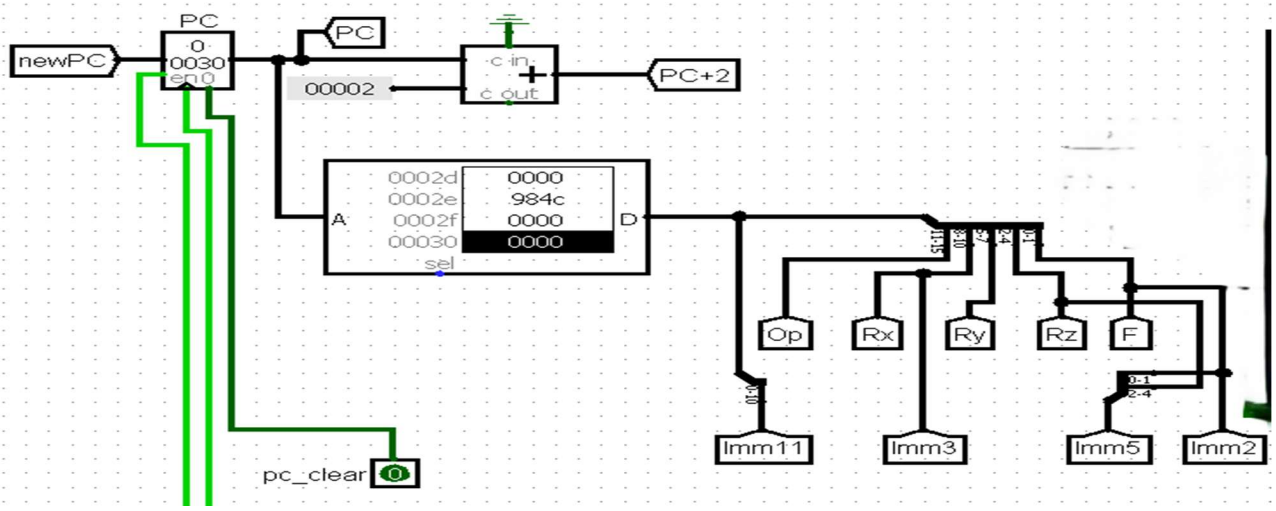


Figure 4 : PC Reg and Instructions memory

In this part of the Datapath, the PC Register contain the address of the next instruction to be executed from the instructions memory, so the PC Register give the address of the instruction to be read from the instructions memory, and the instructions memory return this instruction, to split it as we want, then we split this instruction to many parts to cover all the instruction types that our CPU must deal with them, such as (5-bits) for opCode (for all types),

R-type :

5 - opcode	3 - Rx	3 - Ry	3 - Rz	2 - f

So we split as : 5-bits for opcode, then 3-bits for the Register destination (Rx), 3-bits for the first source (Ry), 3-bits for the second source (Rz), and the last 2-bits for the function (F).

I-type :

5 – opcode	3 - Rx	3 - Ry	Imm5

We split it for this type as : the same MSB 5-bits for opcode, then the same 3-bits for destination (Rx), the same 3-bits for the first source (Ry), and the LSB 5-bits for the immediate 5.

S-type :

5 - opcode	Imm3	3 - Ry	3 - Rz	Imm2

This type is the similar to the first type (R-type) with opcode and the two sources, but replacing the 3-bit to the destination in R-type to be 3-bits for the immediate 3 in this type, and replacing the last 2-bit (function in R-type) to be 2-bits for the immediate 2 in this type.

J-type :

5 - opcode	Imm11

Finally, in this type we split the instruction for 2 parts only the first on is the MSB 5-bits for the opcode, and the remaining 11-bits to the immediate 11.

So, we split the instruction to cover all this types and the control unit determine the type of the instruction will be executed.

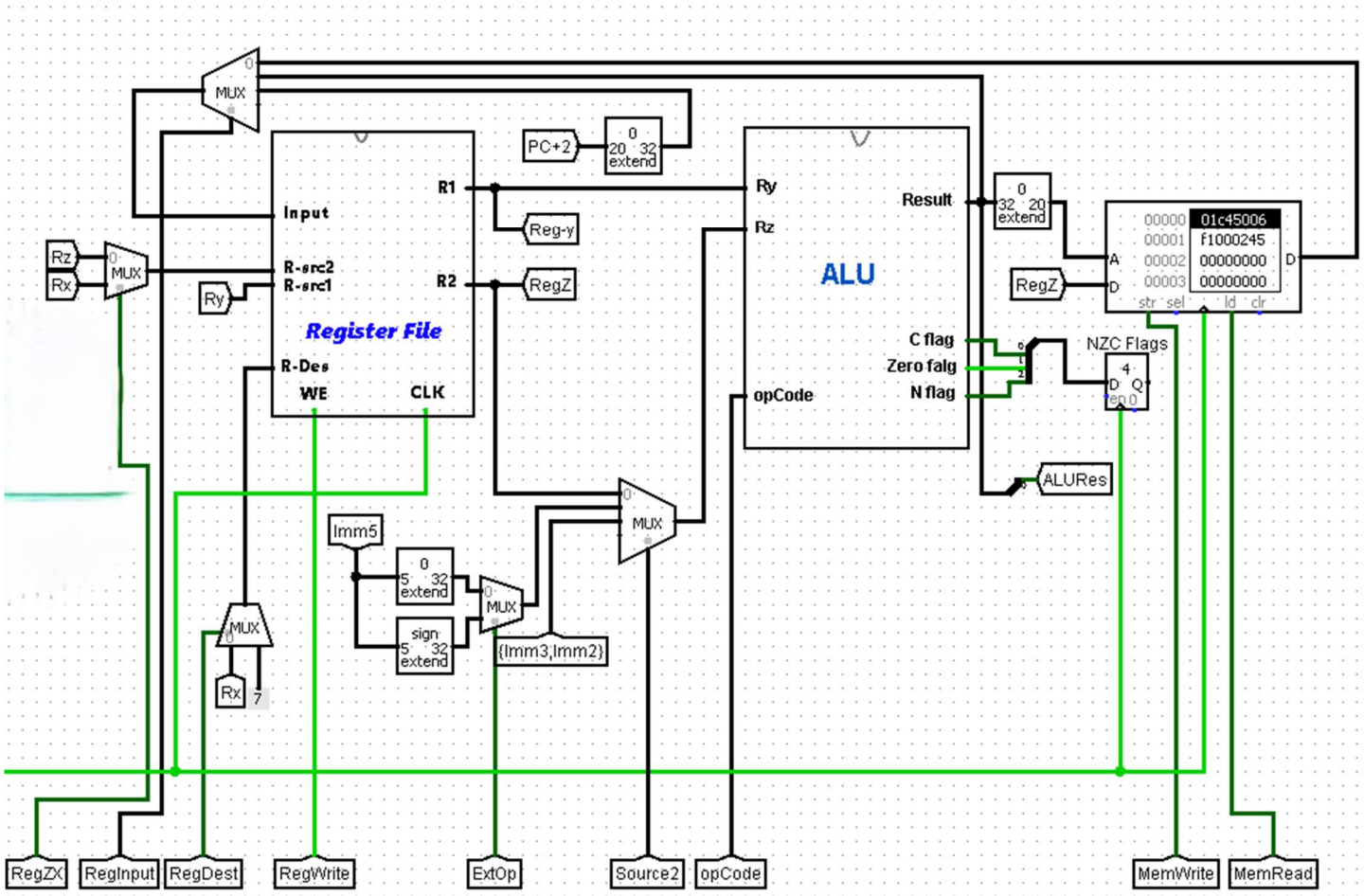


Figure 5 : RegFile, ALU and Data memory connection

After splitting the instruction, then we connect the other components (e.g. Register File, ALU and Data memory), as it shown in Figure 5 above.

Register File :

the source 1 that the CPU must read in each instruction that depends on the registers value is Ry, so Ry connected to the R-src1, and the source 2 is Rz, but in some cases like branches we need to read Rx and don't need Ry, so we connected Rx and Rz by multiplexer to choose on of them depends on the instruction type, we controlled the selection line for this Mux by control signal that control unit generate it, and the destination register that we write on it in the most instruction is Rx, but in case of (JAL) we need to write on R7, so connect them on another Mux, and the Write enable (WE) is a control signal also.

The Value that we want to write it on any register is one of three cases (e.g. ALU result [in R-type and I-type], data from Data memory [LW instruction, or PC+2 – next instruction – [JAL instruction]), so we connect them by another Mux.

ALU :

The First operand (input) for the ALU is always Ry so connected directly with ALU, but the second one may be one of three (e.g. Rz from Register File, zero or sign extend of Immediate 5, 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. And the other output of the ALU (e.g. flags) are stored in special purpose register (NZC Flags Reg).

Data Memory:

This component is used to read from or write to Data memory, the address that should be accessed generated from the ALU (such as Ry + sign_extend(Imm5)), so this will be the output of the ALU to read from it in LW instruction. And the data that will be stored in memory is the content of Rz, so it connected directly to the input data of memory. And there is two control signals for this memory (e.g. Write Enable {memWrite} and Read Enable {memRead}), the control unit will generate them.

PC controller :

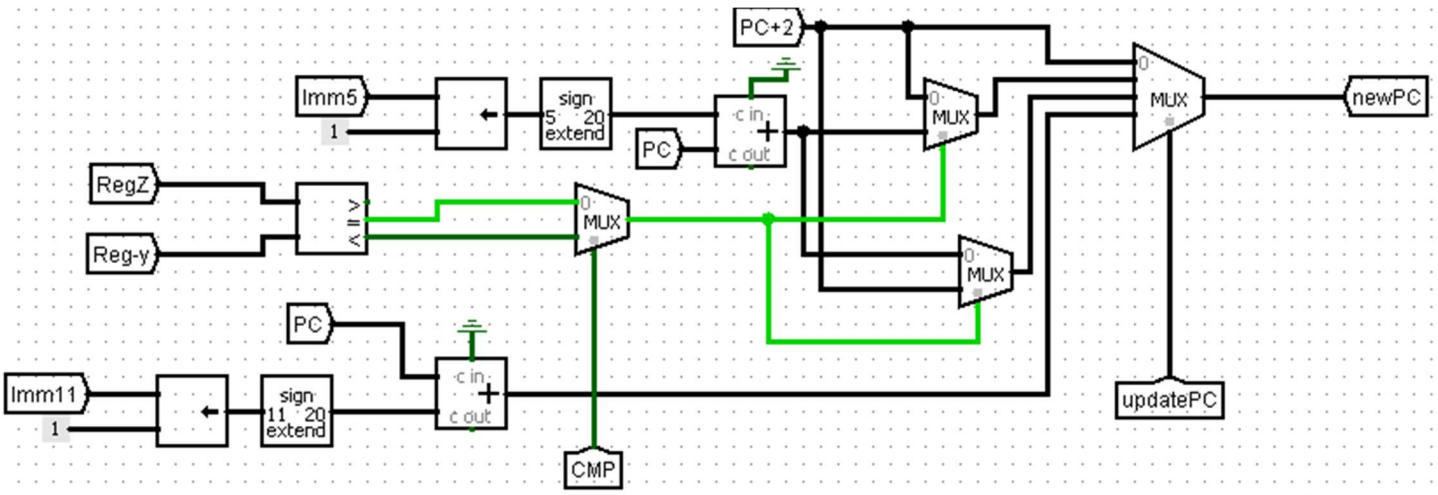


Figure 6 : PC Controller

This component used to control the PC register value (e.g. to control what is the next instruction to be executed). In all instructions except the Branch and Jump the PC increment by 2 (PC += 2) so this is the first choice in the Mux.

In Branch instructions (conditional Branch) if the condition is true then update the value of PC to PC = PC + sign_extend(Imm5 <<1)

If the condition is false then update the value of PC to the next instruction (e.g. PC = PC + 2), we check this conditions by comparator (such as if Rx == Ry), where the tunnel RegZ in Figure 6 above represent the value of Rx and controlled the Mux to choose the condition [equal, less than, …] by control signal (e.g. CMP).

In Jump instructions (unconditional) the value of the PC will change to PC = PC + sign_extend(Imm11<<1).

And another control signal (UpdatePC) to determine how the PC increment (auto increment (PC+2), Branch and check the condition, or Jump without conditions).

These control signals will be generated from the control unit.

4) Main 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.

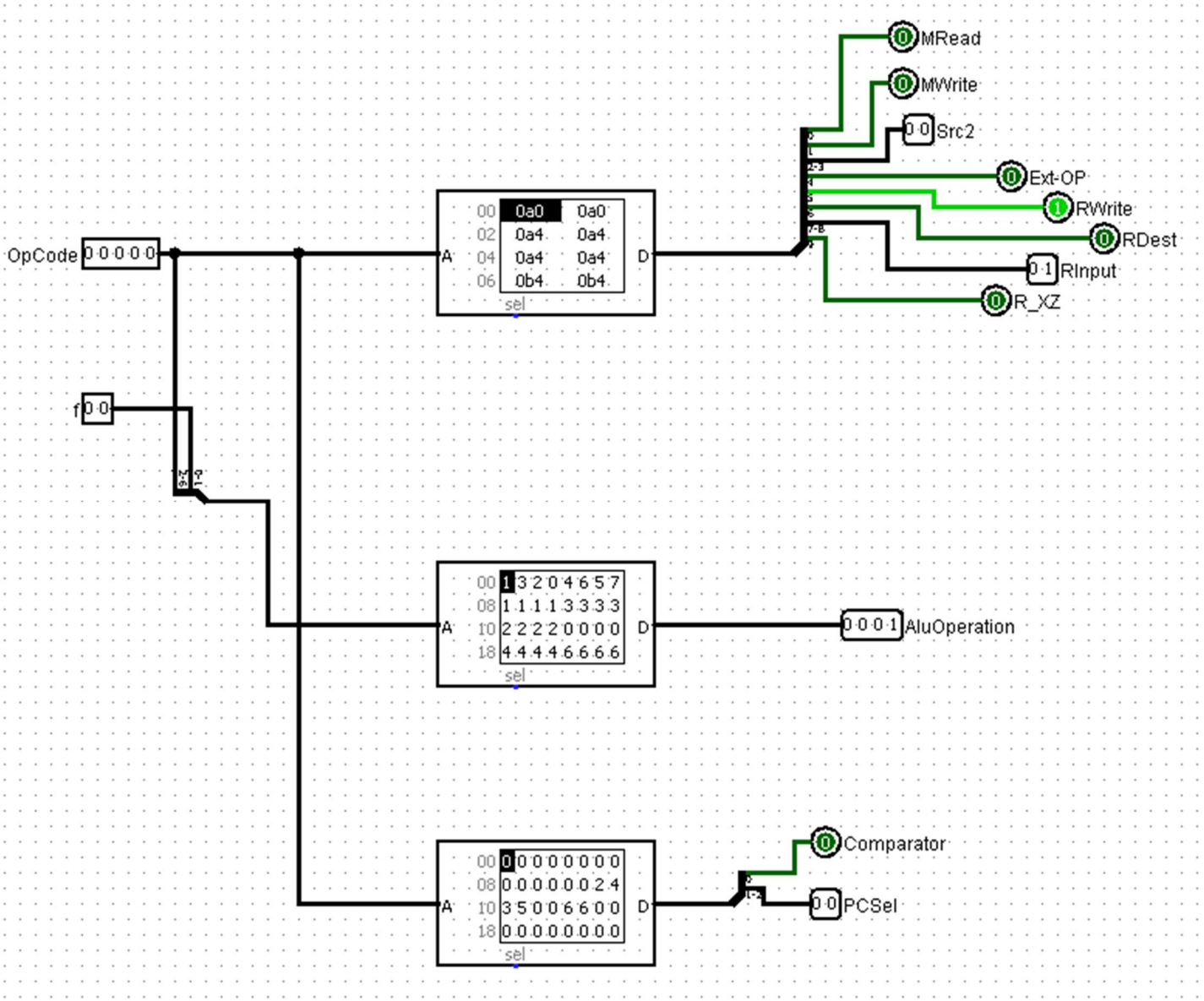


Figure 7 : Main Control Unit

1) General Control Unit :

We implement this Microprogrammed control unit to generate the general control signals (e.g. memWrite, RegWrite, …). We use a ROM to store this control signals for each different opcode – the address of thise memory is 5-bits (opcode).

The Control Signals that generated by this control unit as following :

Table 1 : General Control Signals

Control Signal	Number of bits	Description	Signals meaning
RegZX	**1**	Choose what Register to read Rz or Rx	**0 >> Rz** **1 >> Rx**
RegInput	**2**	Choose what to write on the Register, from memory, ALU Result, or PC+2	**00 >> DataMemory** **01 >> ALU Result** **10 >> PC + 2** **11 >> xxxx**
RegDest	**1**	Determine which Register to write on it, Rx or R7	**0 >> Rx** **1 >> R7**
RegWrite	**1**	Enable write on the Register or disable it	**0 >> no write (D)** **1 >> write (E)**
ExtOp	**1**	Determine the type of extend to the Imm5, zero or sign extend	**0 >> zero extend** **1 >> sign extend**
Source2	**2**	Determine the source (input) 2 to the ALU	**00 >> Rz** **01 >> extend(Imm5)** **10 >> extend({Imm3,Imm2})** **11 >> xxxx**
memWrite	**1**	Enable write on memory or disable it	**0 >> no Write (D)** **1 >> Write (E)**
memRead	**1**	Enable read from memory or disable it	**0 >> no Read (D)** **1 >> Read (E)**

The next table show the control signals for each opcode represented as this :

RegZX_RegInput_RegDest_RegWrite_ExtOp_Source2_memWrite_memRead

Table 2 : General Control Signals Values

Opcode (Address) (5 bits)	Control signals (10 bits)	Control signals in Hexadecimal
00000 = 0x 00	0_01_0_1_0_00_0_0	0x 0A0
00001 = 0x 01	0_01_0_1_0_00_0_0	0x 0A0
00010 = 0x 02	0_01_0_1_0_01_0_0	0x 0A4
00011 = 0x 03	0_01_0_1_0_01_0_0	0x 0A4
00100 = 0x 04	0_01_0_1_0_01_0_0	0x 0A4
00101 = 0x 05	0_01_0_1_0_01_0_0	0x 0A4
00110 = 0x 06	0_01_0_1_1_01_0_0	0x 0B4
00111 = 0x 07	0_01_0_1_1_01_0_0	0x 0B4
01000 = 0x 08	0_01_0_1_1_01_0_0	0x 0B4
01001 = 0x 09	0_01_0_1_1_01_0_0	0x 0B4
01010 = 0x 0A	0_01_0_1_0_01_0_0	0x 0A4
01011 = 0x 0B	0_01_0_1_0_01_0_0	0x 0A4
01100 = 0x 0C	0_01_0_1_0_01_0_0	0x 0A4
01101 = 0x 0D	0_01_0_1_0_01_0_0	0x 0A4
01110 = 0x 0E	1_00_0_0_1_00_0_0	0x 210
01111 = 0x 0F	1_00_0_0_1_00_0_0	0x 210
10000 = 0x 10	1_00_0_0_1_00_0_0	0x 210
10001 = 0x 11	1_00_0_0_1_00_0_0	0x 210
10010 = 0x 12	0_00_0_1_1_01_0_1	0x 035
10011 = 0x 13	0_00_0_0_0_10_1_0	0x 00A
10100 = 0x 14	0_00_0_0_0_00_0_0	0x 000
10101 = 0x 15	0_10_1_1_1_00_0_0	0x 170

2) ALU Control Unit :

We implement this Microprogrammed control unit to generate the opCode control signal for the ALU (4-bits) to determine the Arithmetic or logical operation to be executed in the ALU. We use another ROM to store this control signal for each different opcode and different function (f) – the address of this memory is 7-bits {opcode,f}.

Table 3 : ALU opcode control signal description

opCode of the ALU	Arithmetic or Logical operation	Description
0000 = 0x 0	**XOR**	Bitwise XOR Rx = Ry ^ Rz
0001 = 0x 1	**AND**	Bitwise AND Rx = Ry & Rz
0010 = 0x 2	**OR**	Bitwise OR Rx = Ry | Rz
0011 = 0x 3	**CAND**	Rx = ~Ry & Rz
0100 = 0x 4	**ADD**	Rx = Ry + Rz
0101 = 0x 5	**SEQ**	SET Rx = 1 if (Ry == Rz) Otherwise Rx = 0
0110 = 0x 6	**NADD**	Rx = -Ry + Rz
0111 = 0x 7	**SLT**	SET Rx = 1 if (Ry < Rz) Otherwise Rx = 0
1000 = 0x 8	**SRA**	Shift Right Arithmetic
1001 = 0x 9	**SRL**	Shift Right Logical
1010 = 0x A	**SLL**	Shift Left Logical
1011 = 0x B	**ROR**	Rotate Right
1100 = 0x C	**XXXX**	Don’t Care
1101 = 0x D	**XXXX**	Don’t Care
1110 = 0x E	**XXXX**	Don’t Care
1111 = 0x F	**XXXX**	Don’t Care

The next table show the ALU opcode control signal for each instruction opcode and function (f) :

Table 4 : ALU opcode Control signal values

{ Opcode, f } (Address) (7 bits)	ALU opcode (4 bits)	Control signals in Hexadecimal
0000000 = 0x 00	0001	0x 1
0000001 = 0x 01	0011	0x 3
0000010 = 0x 02	0010	0x 2
0000011 = 0x 03	0000	0x 0
0000100 = 0x 04	0100	0x 4
0000101 = 0x 05	0110	0x 6
0000110 = 0x 06	0101	0x 5
0000111 = 0x 07	0111	0x 7
00010XX 0001000 = 0x 08 0001001 = 0x 09 0001010 = 0x 0A 0001011 = 0x 0B	0001	0x 1
00011XX 0001100 = 0x 0C 0001101 = 0x 0D 0001110 = 0x 0E 0001111 = 0x 0F	0011	0x 3
00100XX 0010000 = 0x 10 0010001 = 0x 11 0010010 = 0x 12 0010011 = 0x 13	0010	0x 2
00101XX 0010100 = 0x 14 0010101 = 0x 15 0010110 = 0x 16 0010111 = 0x 17	0000	0x 0

00110XX **0011000 = 0x 18** **0011001 = 0x 19** **0011010 = 0x 1A** **0011011 = 0x 1B**	**0100**	**0x 4**
00111XX **0011100 = 0x 1C** **0011101 = 0x 1D** **0011110 = 0x 1E** **0011111 = 0x 1F**	**0110**	**0x 6**
01000XX **0100000 = 0x 20** **0100001 = 0x 21** **0100010= 0x 22** **0100011 = 0x 23**	**0101**	**0x 5**
01001XX **0100100 = 0x 24** **0100101 = 0x 25** **0100110= 0x 26** **0100111 = 0x 27**	**0111**	**0x 7**
01010XX **0101000 = 0x 28** **0101001 = 0x 29** **0101010= 0x 2A** **0101011 = 0x 2B**	**1010**	**0x A**
01011XX **0101100 = 0x 2C** **0101101 = 0x 2D** **0101110= 0x 2E** **0101111 = 0x 2F**	**1001**	**0x 9**
01100XX **0110000 = 0x 30** **0110001 = 0x 31** **0110010= 0x 32** **0110011 = 0x 33**	**1000**	**0x 8**

01101XX **0110100 = 0x34** **0110101 = 0x35** **0110110 = 0x36** **0110111 = 0x37**	**1011**	**0x B**
01110XX **0111000 = 0x 38** **0111001 = 0x 39** **0111010 = 0x 3A** **0111011 = 0x 3B**	**0101**	**0x 5**
01111XX **0111100 = 0x 3C** **0111101 = 0x 3D** **0111110 = 0x 3E** **0111111 = 0x 3F**	**0101**	**0x 5**
10000XX **1000000 = 0x 40** **1000001 = 0x 41** **1000010 = 0x 42** **1000011 = 0x 43**	**0111**	**0x 7**
10001XX **1000100 = 0x 44** **1000101 = 0x 45** **1000110 = 0x 46** **1000111 = 0x 47**	**0111**	**0x 7**
10010XX **1001000 = 0x 48** **1001001 = 0x 49** **1001010 = 0x 4A** **1001000 = 0x 4B**	**0100**	**0x 4**
10011XX **1001100 = 0x 4C** **1001101 = 0x 4D** **1001110 = 0x 4E** **1001111 = 0x 4F**	**0100**	**0x 4**

3) PC Control Unit :

We implement this Microprogrammed control unit to generate the updatePC control signal for the Mux that determine the value of the next instruction (auto increment e.g. PC + 2, conditional Branch, or unconditional Jump) (2-bits) and another control signal (CMP) that determine the condition of the Branch to check it (e.g. Rx == Ry, or Rx < Ry,…). We use another ROM to store this control signals for each different opcode – the address of this memory is 5-bits (opcode).

Table 5 : PC control Signals Description

Control Signal	Number of bits	Description	Signals meaning
UpdatePC	**2**	Determine the next instruction (auto increment PC+2 , or there is a branch if the condition is true or jump without condition 01 >> to check (Equal or less than) 10 >> to check (not Equal or not less than – greater than or equal -	**00 >> PC + 2** **01 >> PC + 2 if condition false or PC + sign_extend (Imm5) if condition true** **10 >> PC + 2 if condition false or PC + sign_extend (Imm5) if condition true** **11 >> PC + sign_extend(Imm11) – Jump**
CMP	**1**	Determine what to check for the condition of branch	**0 >> check Equal, not equal** **1 >> check less than, not less than – greater or equal**

The next table show the PC control signals for each different opcode, represented as :

updatePC_CMP

Table 6 : PC control Signals Values

Opcode (Address) (5 bits)	Control Signals (3 bits)	Control Signals in Hexadecimal
00000 = 0x 00	000	0x 0
00001 = 0x 01	000	0x 0
00010 = 0x 02	000	0x 0
00011 = 0x 03	000	0x 0
00100 = 0x 04	000	0x 0
00101 = 0x 05	000	0x 0
00110 = 0x 06	000	0x 0
00111 = 0x 07	000	0x 0
01000 = 0x 08	000	0x 0
01001 = 0x 09	000	0x 0
01010 = 0x 0A	000	0x 0
01011 = 0x 0B	000	0x 0
01100 = 0x 0C	000	0x 0
01101 = 0x 0D	000	0x 0
01110 = 0x 0E	010	0x 2
01111 = 0x 0F	100	0x 4
10000 = 0x 10	011	0x 3
10001 = 0x 11	101	0x 5
10010 = 0x 12	000	0x 0
10011 = 0x 13	000	0x 0
10100 = 0x 14	110	0x 6
10101 = 0x 15	110	0x 6

As we see in Figure 7, we connect the three control units (e.g. General control unit, ALU control unit, and PC control unit) in main control unit that generates all the control signals to the SCP (single cycle processor) in the Datapath, as clarify in Figure 8 bellow :

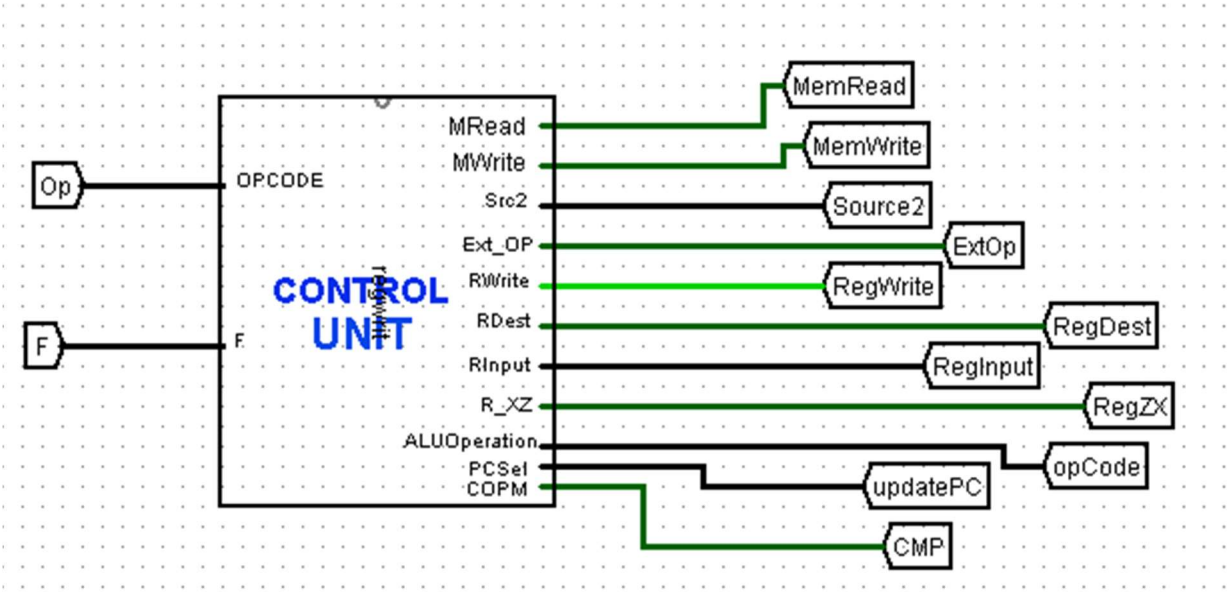


Figure 8 : Main Control unit in Datapath

As we see in Figure 8 above this is the main control unit that generates all the required control signals to Datapath, and this component which controls the single cycle processor to work probably.

Simulation and Testing :

Set one of Instructions to check the correctness of the CPU :

Table 7 : R-type Instructions for Test

Address of instruction	Assembly	Binary	Hexadecimal
0x 00000	LW R1,R3,0	1001 0001 0110 0000	0x 9160
0x 00002	LW R2,R3,1	1001 0010 0110 0001	0x 9261
0x 00004	AND R3,R2,R1	0000 0011 0100 0100	0x 0344
0x 00006	CAND R4,R2,R1	0000 0100 0100 0101	0x 0445
0x 00008	OR R5,R2,R1	0000 0101 0100 0110	0x 0546
0x 0000A	XOR R6,R2,R1	0000 0110 0100 0111	0x 0647
0x 0000C	ADD R3,R2,R1	0000 1011 0100 0100	0x 0B44
0x 0000E	NADD R3,R2,R1	0000 1011 0100 0101	0x 0B45
0x 00010	SEQ R3,R2,R1	0000 1011 0100 0110	0x 0B46
0x 00012	SLT R3,R2,R1	0000 1011 0100 0111	0x 0B47

Instructions Memory :

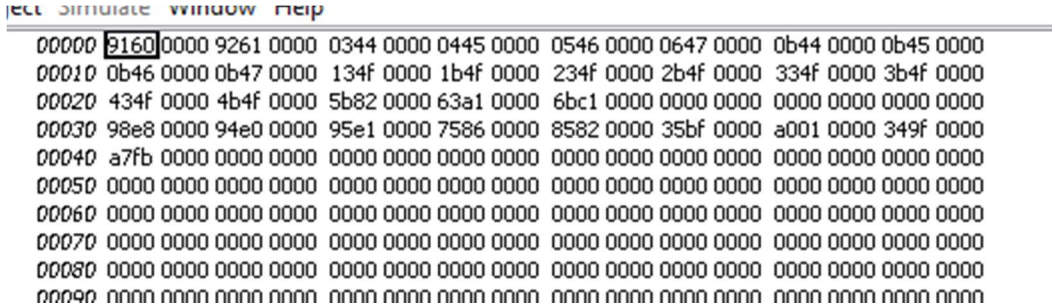


Figure 9 : Instructions Memory

The initially Registers File (Before Any Instructions) :

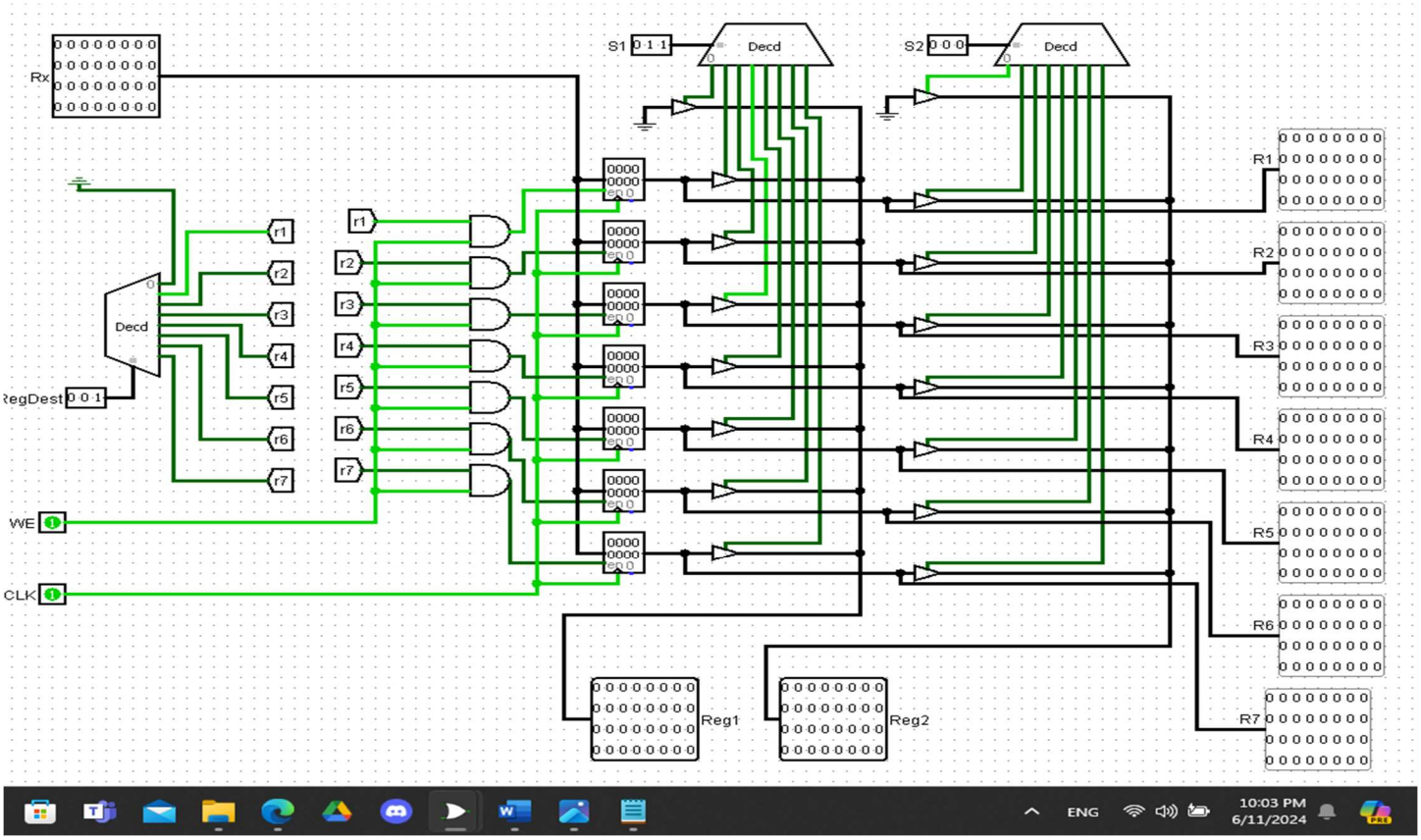


Figure 10 : Initially Registers File

The initially Data Memory :

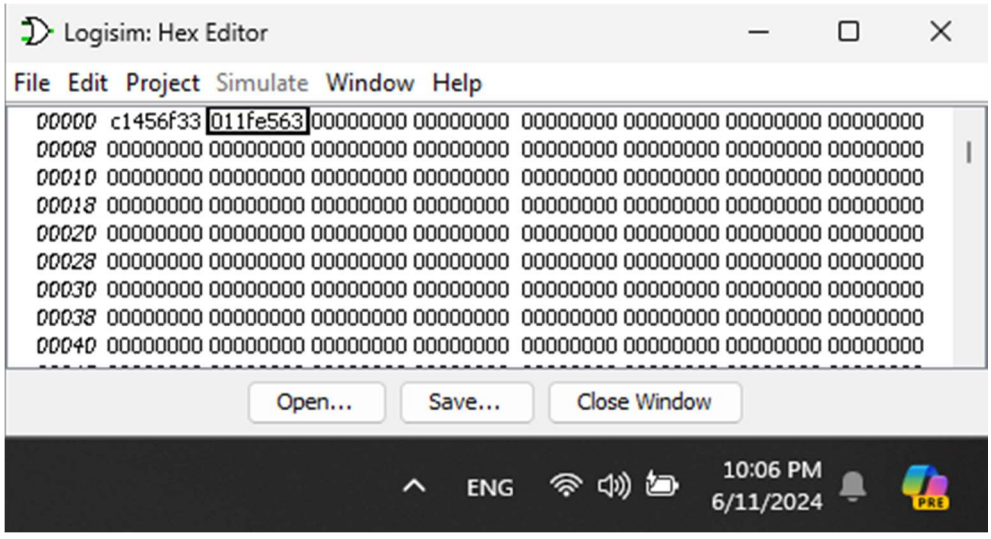
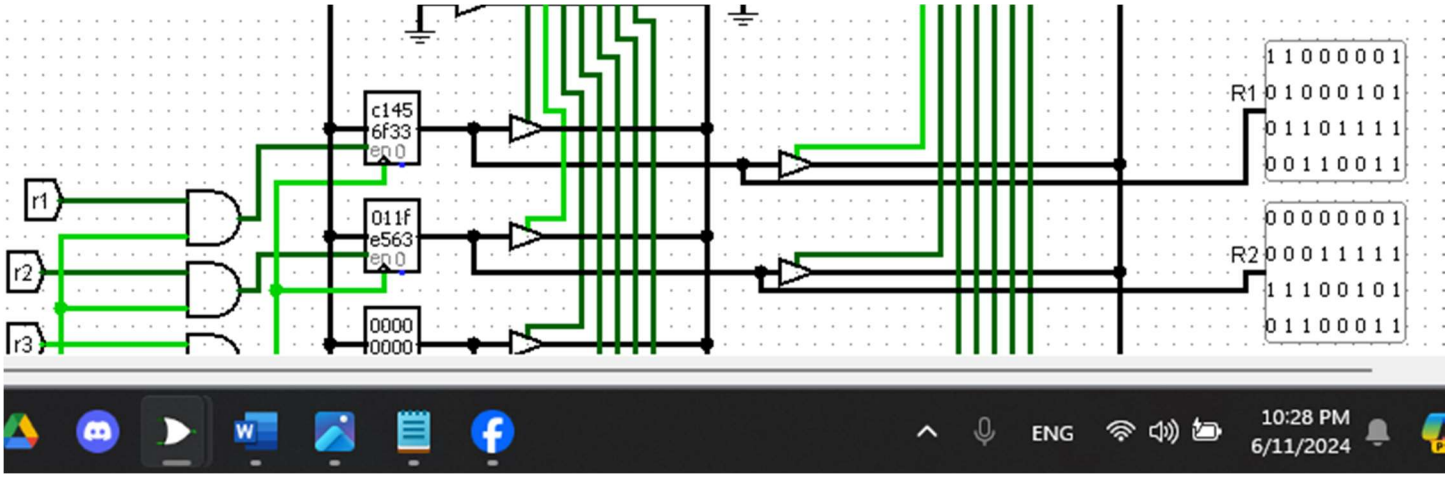


Figure 11 : Initially Data Memory

After Simulate the First Two instructions :

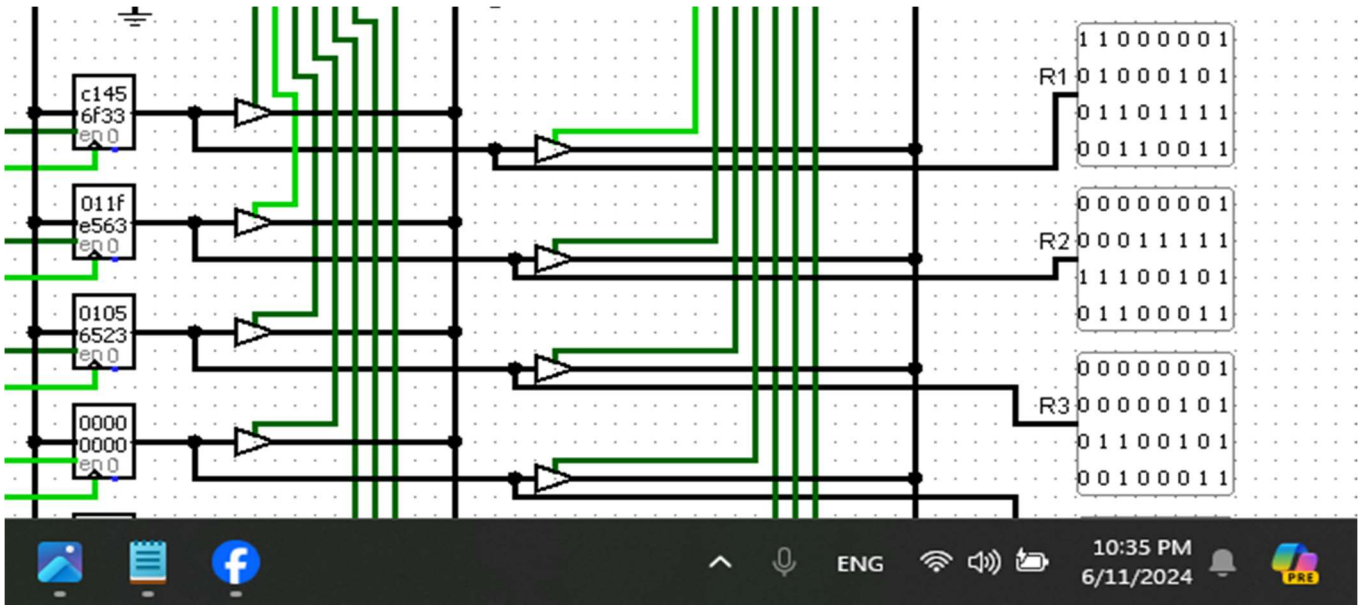
LW R1,R3,0 ➔ Load to R1 the content of the Address R3 + sign_extend(00000), the content of R3 is 0 so Load the Data at address 0x 00000 to R1, so the content of **R1 will be 0x C1456F33**

LW R2,R3,1 ➔ Load to R2 the content of the Address R3 + sign_extend(00001), the content of R3 is 0 so Load the Data at address 0x 00001 to R2, so the content of **R2 will be 0x 011FE563**



After Simulate the Third instruction :

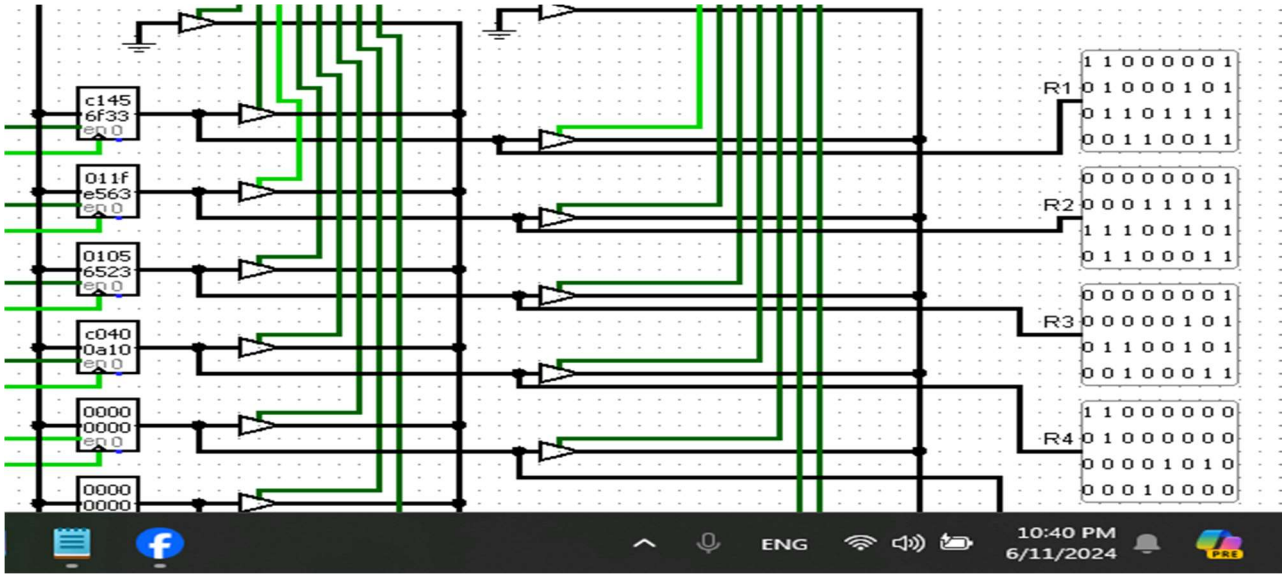
AND R3,R2,R1 ➔ R3 = R2 & R1,
so **R3 will be 00000001000001010110010100100011 = 0x 01056523**



After Simulate the Fourth instruction :

CAND R4,R2,R1 ➔ R4 = ~R2 & R1 ,

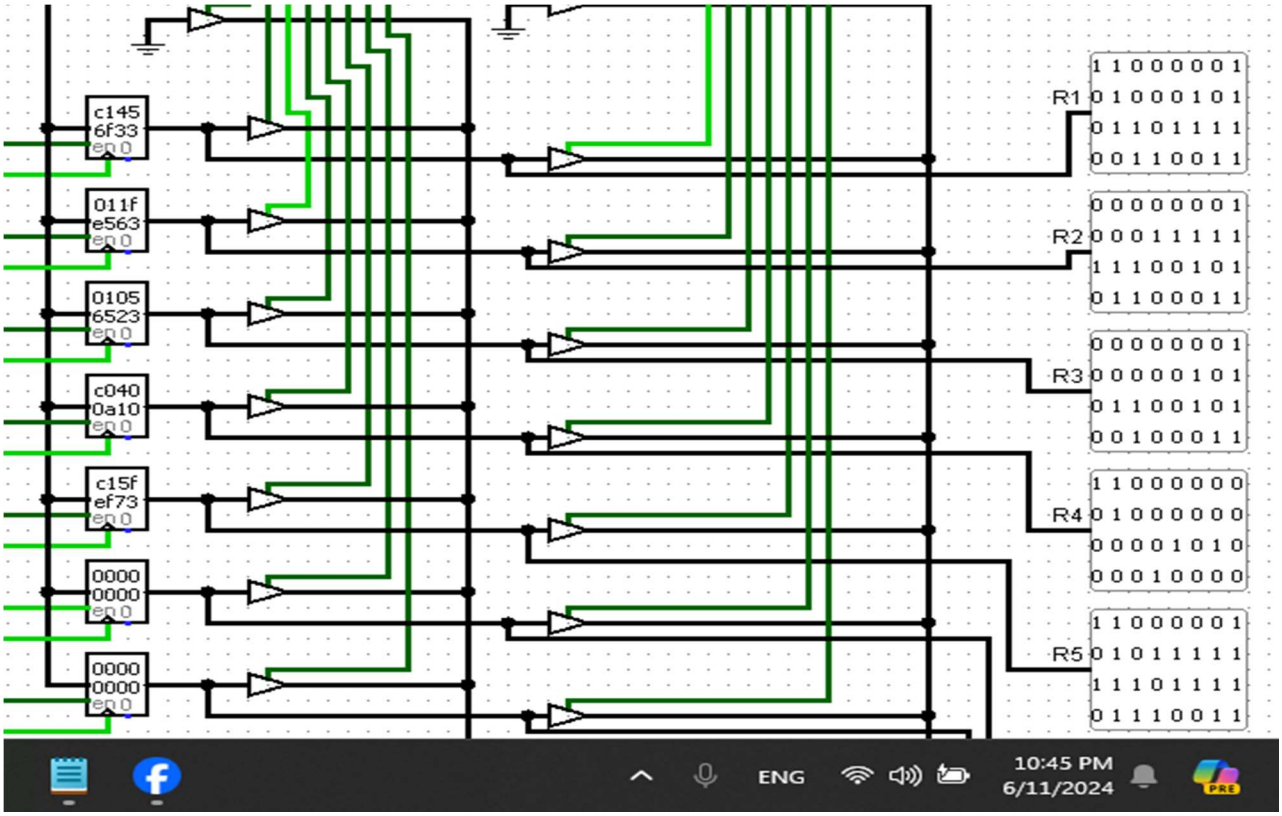
So **R4 will be 11000000010000000000101000010000 = 0x C0400A10**



After Simulate the Fifth instruction :

OR R5,R2,R1 ➔ R5 = R2 | R1 ,

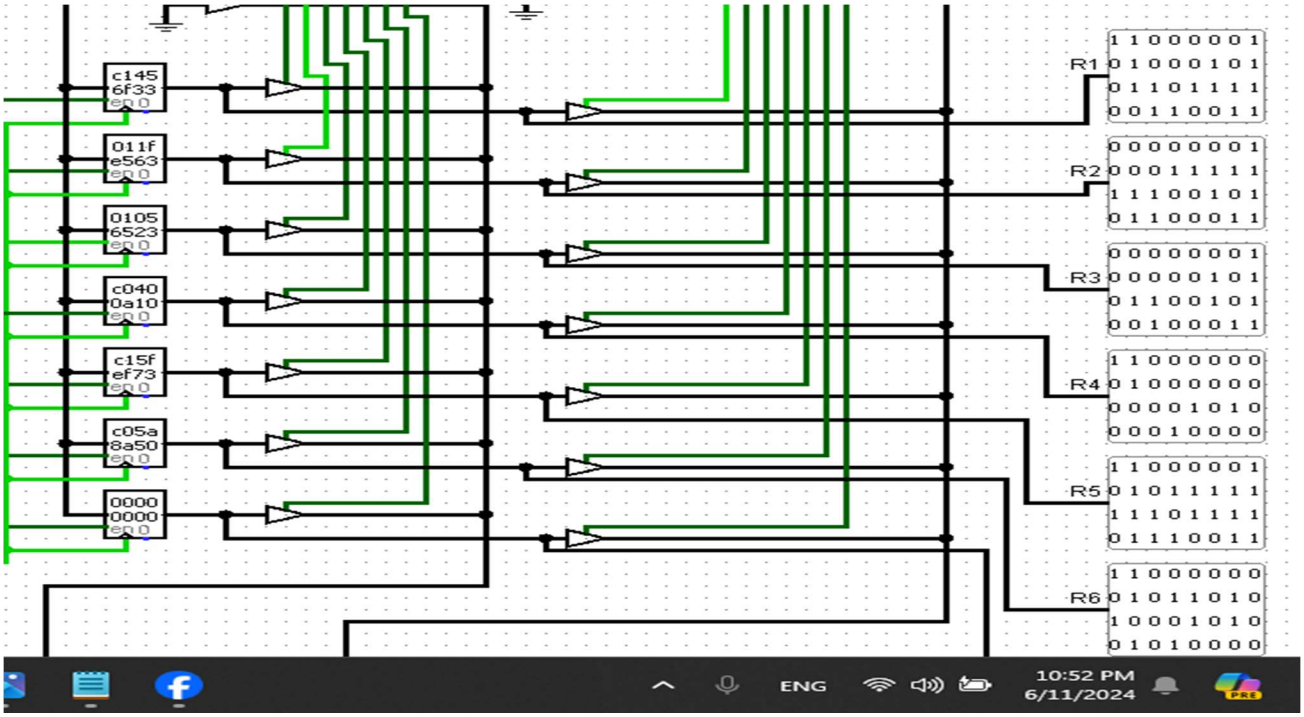
So, **R5 will be 11000001010111111110111101110011= 0x C15FEF73**



After Simulate the sixth instruction :

XOR R6,R2,R1 ➔ R6 = R2 ^ R1,

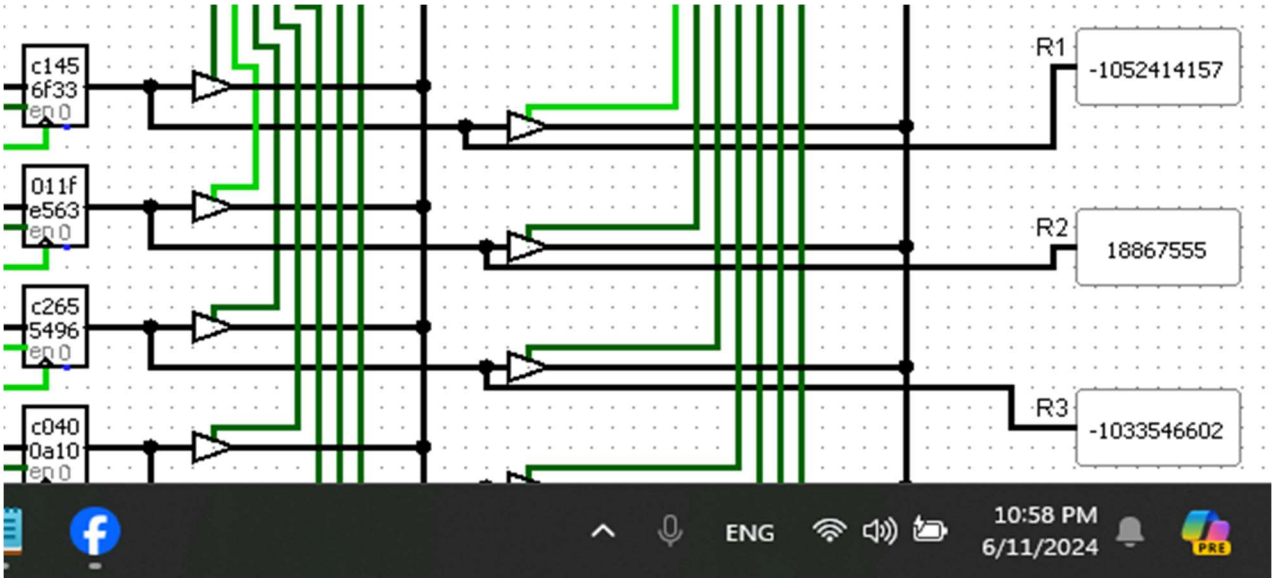
So, **R6 will be 11000000010110101000101001010000 = 0x C05A8A50**



After Simulate the seventh instruction :

ADD R3,R2,R1 ➔ R3 = R2 + R1,

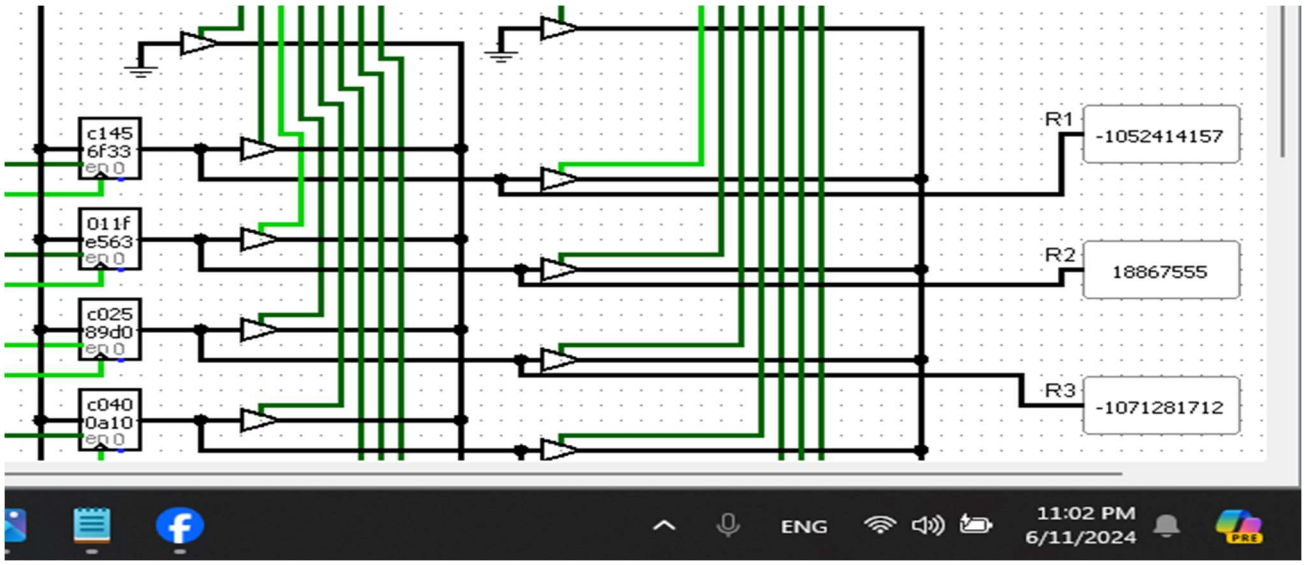
So, **R3 = 18867555 + -1052414157 = -1033546602 (in decimal)**



After Simulate the eighth instruction :

NADD R3,R2,R1 ➔ R3 = -R2 + R1,

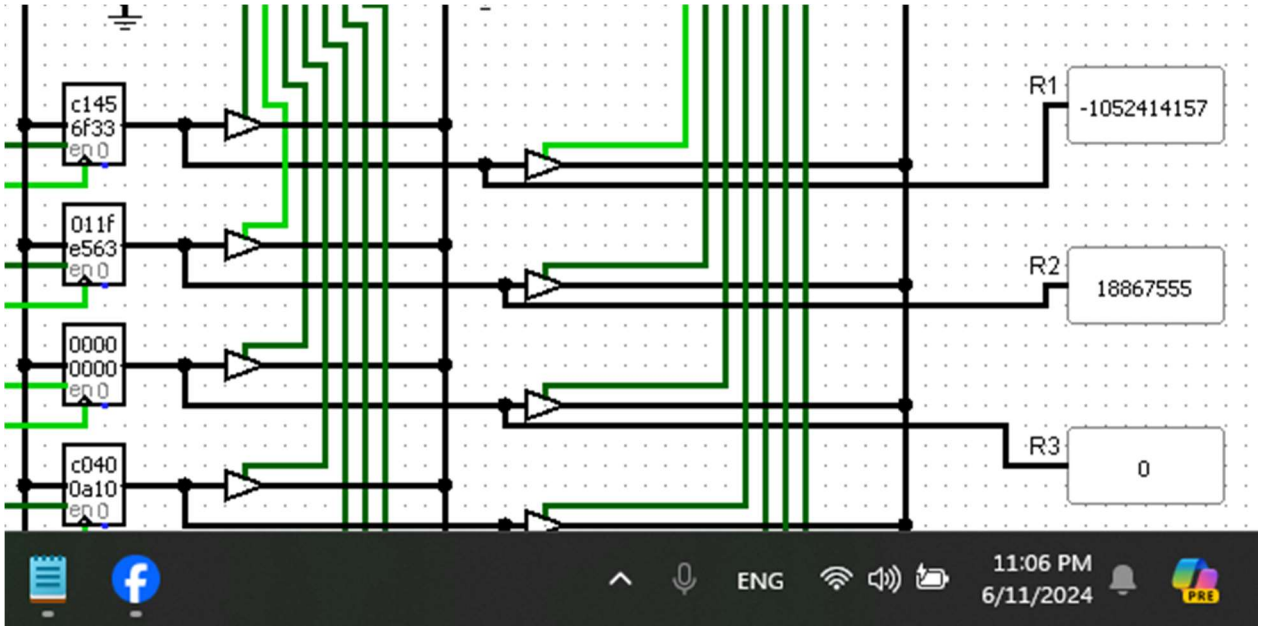
So, **R3 = -(18867555) + -1052414157 = -1071281712 (in decimal)**



After Simulate the ninth instruction :

SEQ R3,R2,R1 ➔ if R2 == R1, then R3 = 1, otherwise R3 = 0

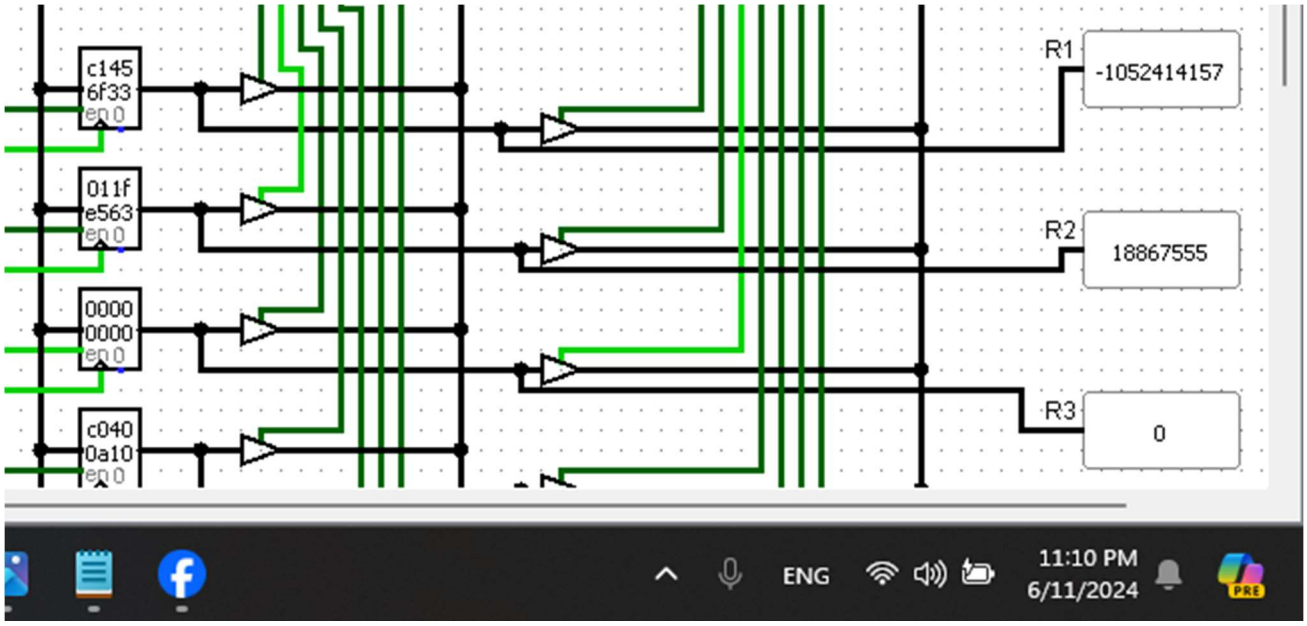
So, **R3 will be 00000000000000000000000000000000 = 0x 00000000**



After Simulate the tenth instruction :

SLT R3,R2,R1 ➔ if R2 < R1, then R3 = 1, otherwise R3 = 0

So, **R3 will be 00000000000000000000000000000000 = 0x 00000000**



Set two of Instructions to check the correctness of the CPU :

Table 8 : Set of Instructions to Test

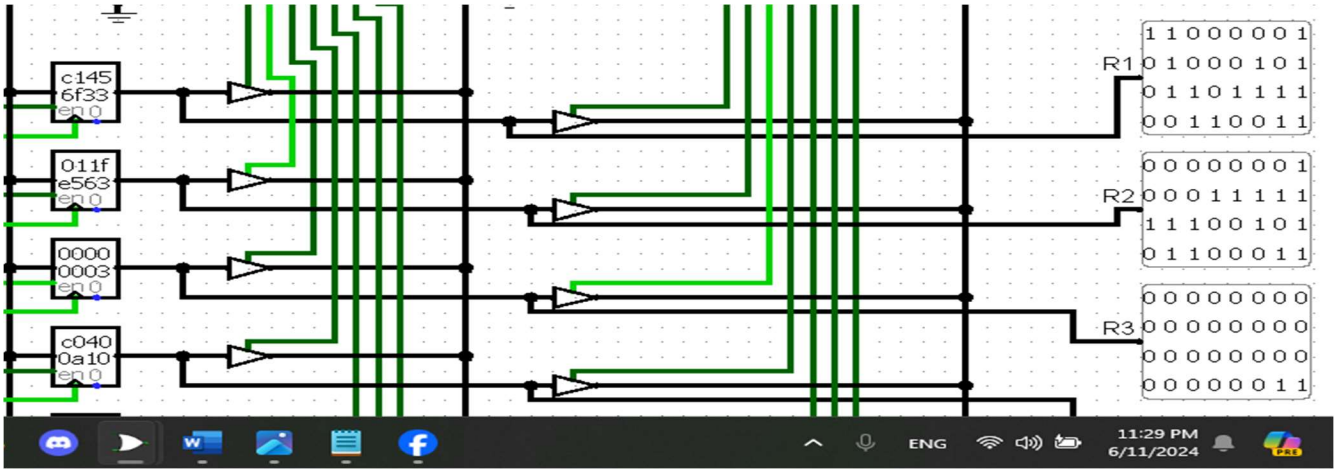
Address of instruction	Assembly	Binary	Hexadecimal
0x 00014	ANDI R3,R2,15	0001 0011 0100 1111	0x 134F
0x 00016	CANDI R3,R2,15	0001 1011 0100 1111	0x 1B4F
0x 00018	ORI R3,R2,15	0010 0011 0100 1111	0x 234F
0x 0001A	XORI R3,R2,15	0010 1011 0100 1111	0x 2B4F
0x 0001C	ADDI R3,R2,15	0011 0011 0100 1111	0x 334F
0x 0001E	NADDI R3,R2,15	0011 1011 0100 1111	0x 3B4F
0x 00020	SEQI R3,R2,15	0100 0011 0100 1111	0x 434F
0x 00022	SLTI R3,R2,15	0100 1011 0100 1111	0x 4B4F
0x 00024	SRL R3,R4,2	0101 1011 1000 0010	0x 5B82
0x 00026	SRA R3,R5,1	0110 0011 1010 0001	0x 63A1
0x 00028	ROR R3,R6,1	0110 1011 1100 0001	0x 6BC1
0x 00030	SW 0,R7,R2,0	1001 1000 1110 1000	0x 98E8

First instruction :

ANDI R3,R2,15 ➔ R3 = R2 & (15), So

R3 = 00000001000111111110010101100011 & 00000000000000000000000000001111

R3 will be 00000000000000000000000000000011 = 0x 00000003

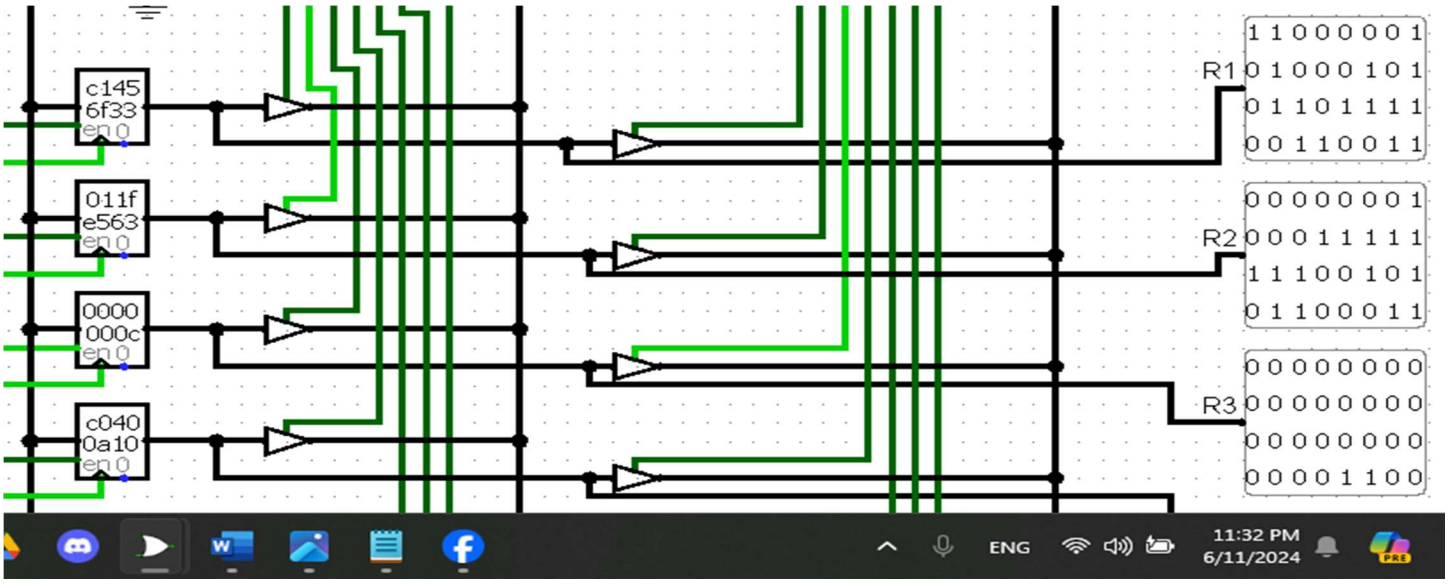


Second instruction :

CANDI R3,R2,15 ➔ R3 = ~R2 & (15), So

R3 = 11111110111000000001101010011100 & 00000000000000000000000000001111

R3 will be 00000000000000000000000000001100 = 0x 0000000C

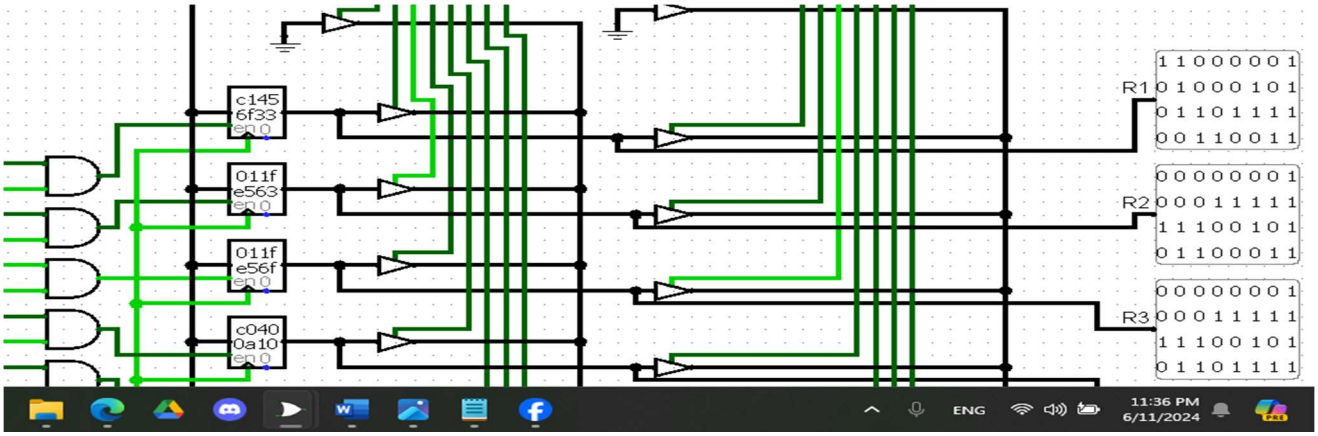


Third instruction :

ORI R3,R2,15➔ R3 = R2 | (15), So

R3 = 00000001000111111110010101100011 | 00000000000000000000000000001111

R3 will be 00000001000111111110010101101111 = 0x 011FE56F

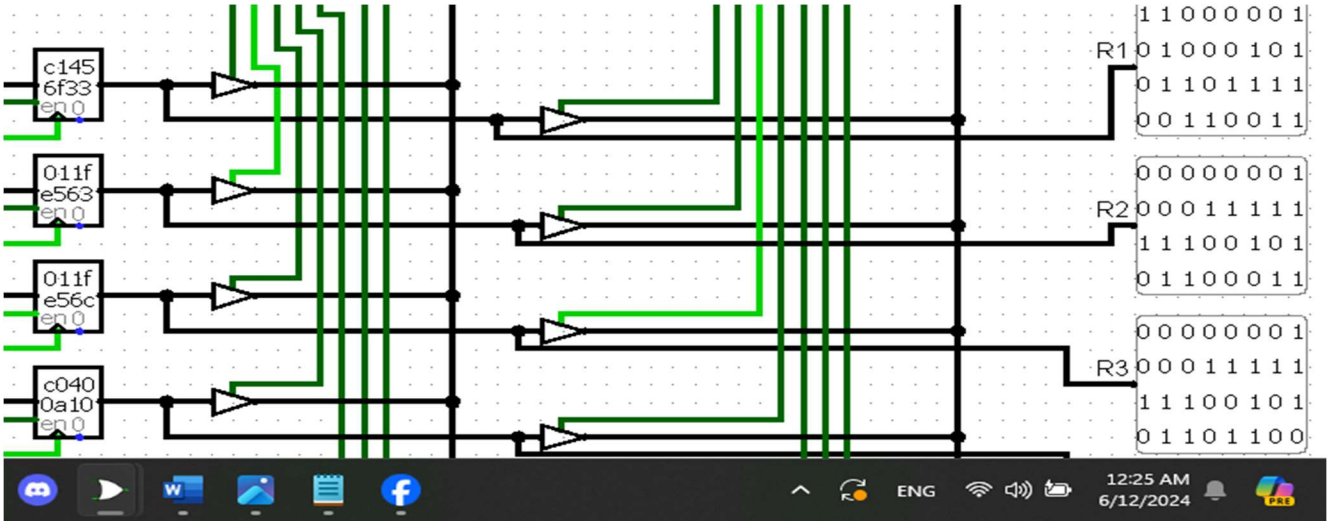


Fourth instruction :

XORI R3,R2,15➔ R3 = R2 ^ (15), So

R3 = 00000001000111111110010101100011 ^ 00000000000000000000000000001111

R3 will be 00000001000111111110010101101100 = 0x 011FE56C

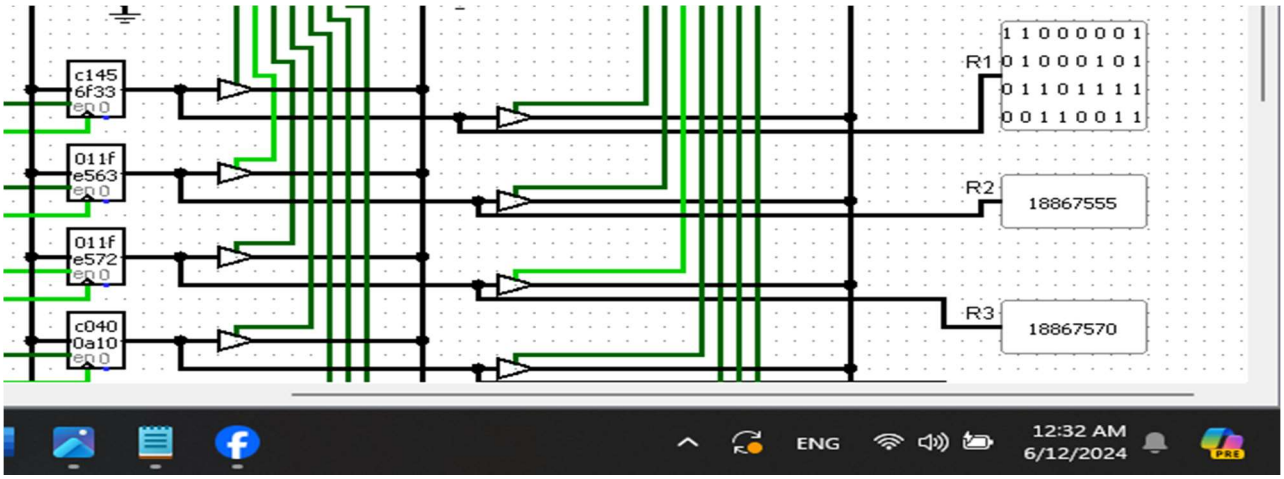


Fifth instruction :

ADDI R3,R2,15➔ R3 = R2 + (15), So

R3 = 18867555 + 15 = 18867570 (in decimal)

R3 will be 18867570 (in decimal)

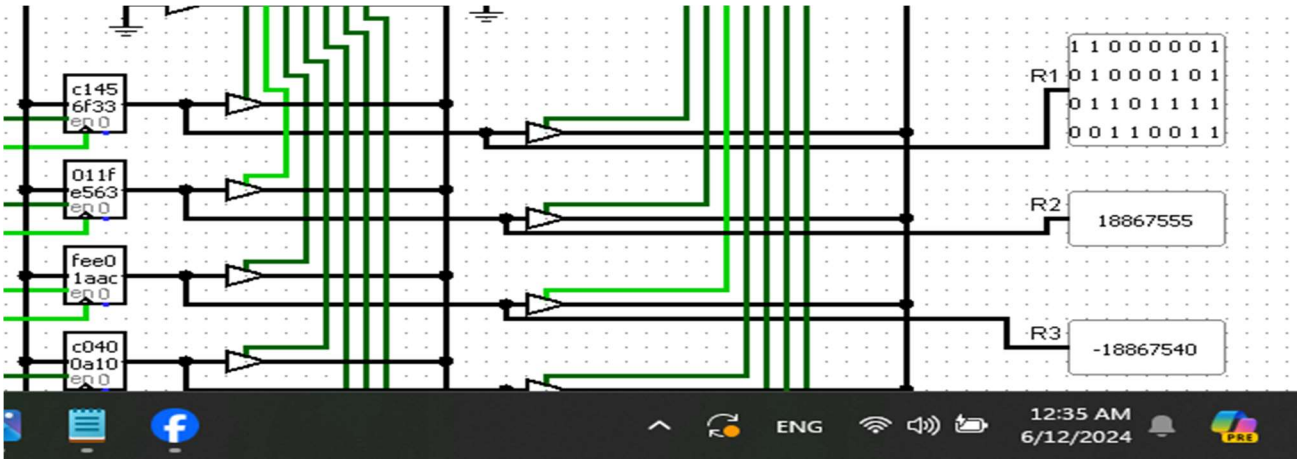


Sixth instruction :

NADDI R3,R2,15➔ R3 = -R2 + (15), So

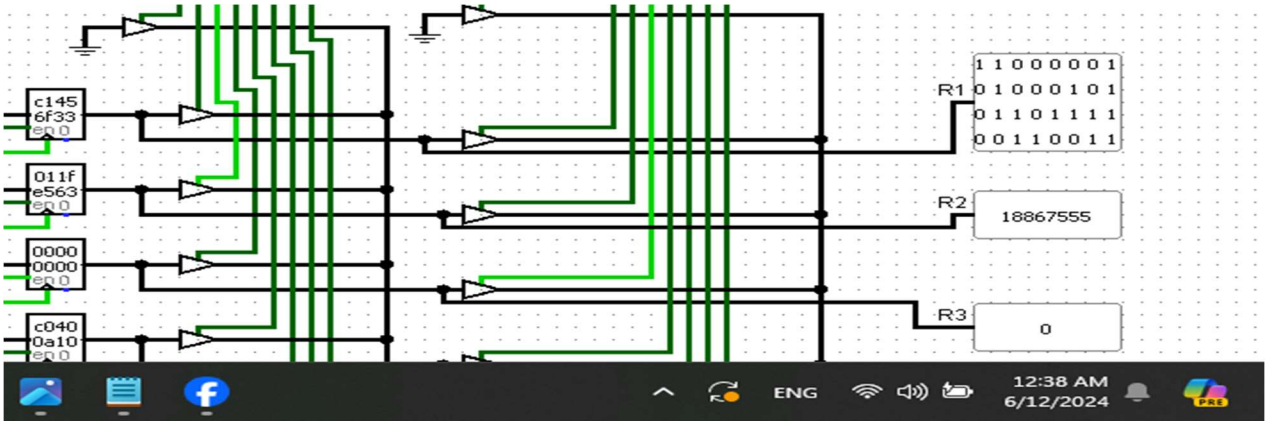
R3 = -18867555 + 15 = -18867540 (in decimal)

R3 will be -18867540 (in decimal)



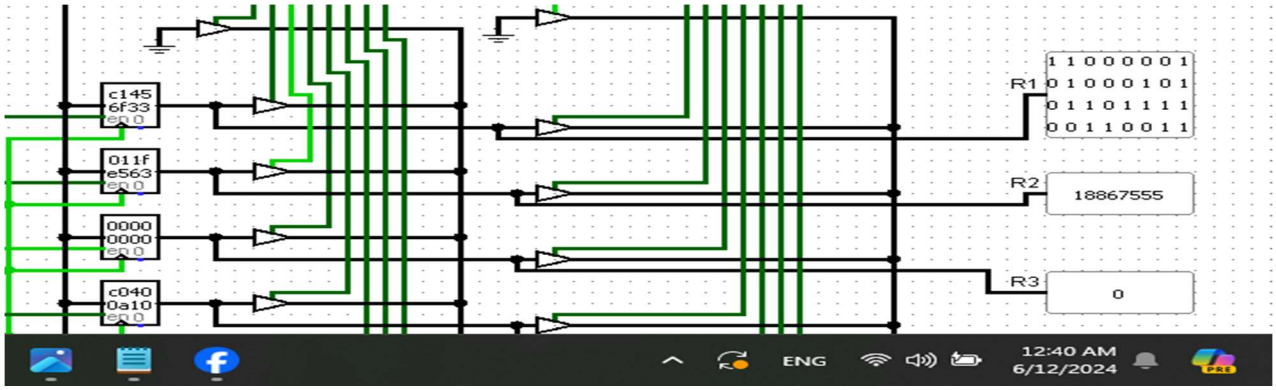
Seventh instruction :

SEQI R3,R2,15➔ R3 = (R2 == 15) so **R3 will be 0**



Eighth instruction :

SLTI R3,R2,15➔ R3 = (R2 < 15) so **R3 will be 0**

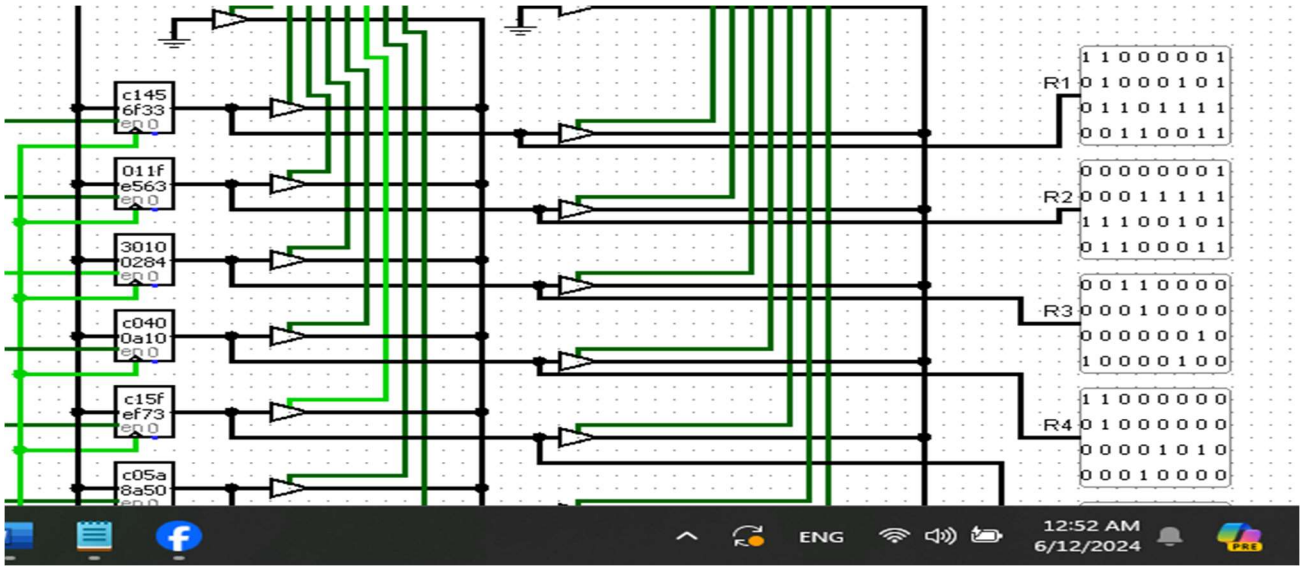


Ninth instruction :

SRL R3,R4,2➔ R3 = shift right logical for R4 by 2 bits

R3 = shift_right_2bits(11000000010000000000101000010000)

R3 will be 00110000000100000000001010000100 = 0x 30100284

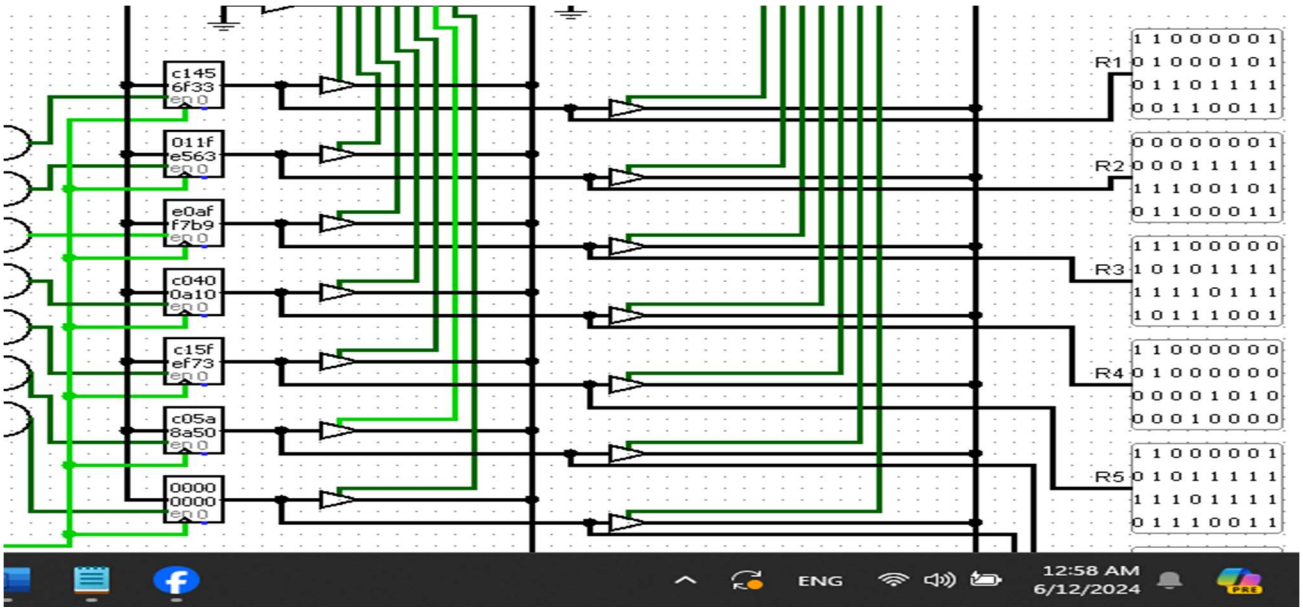


Tenth instruction :

SRA R3,R5,1 ➔ R3 = shift right Arithmetic for R5 by 1 bit

R3 = shift_right_Arith_1bit(11000001010111111110111101110011)

R3 will be 11100000101011111111011110111001 = 0x E0AFF7B9

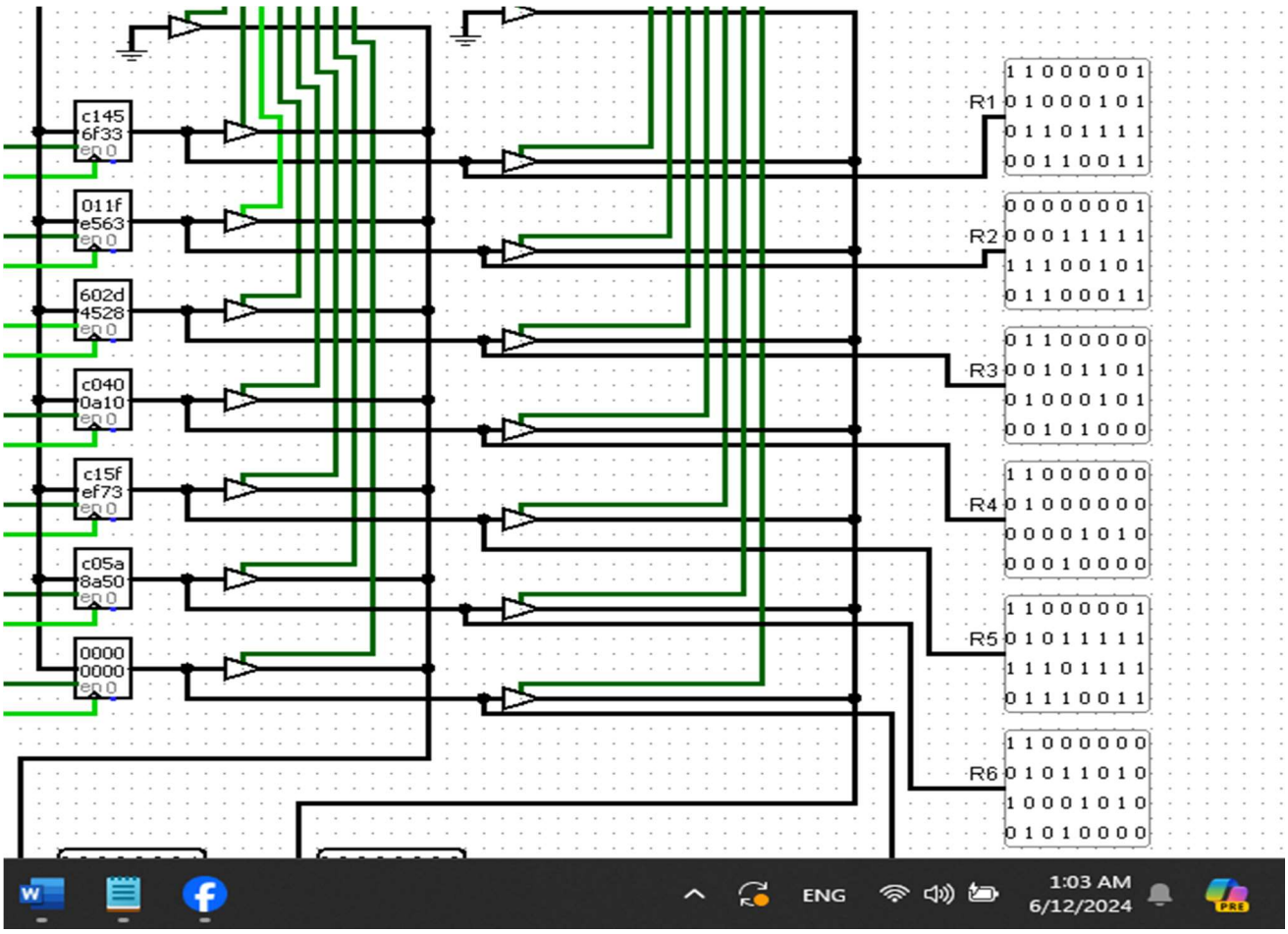


Eleventh instruction :

ROR R3,R6,1➔ R3 = Rotate Right for R6 by 1 bit

R3 = rotate_right_1bit(11000000010110101000101001010000)

R3 will be 01100000001011010100010100101000 = 0x 602D4528



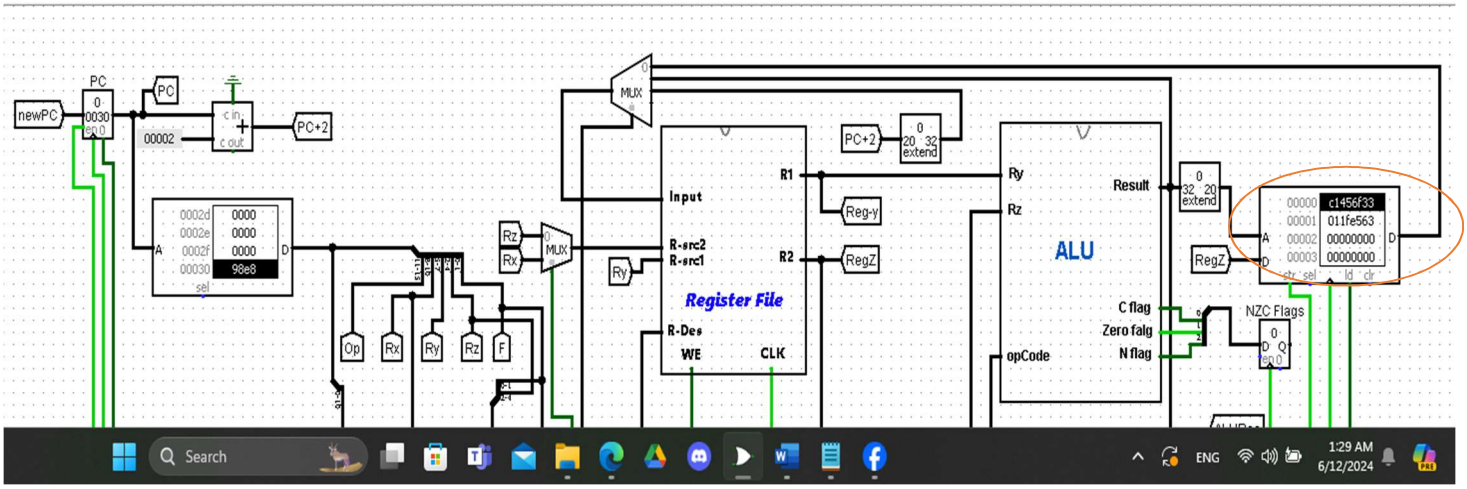
Twelfth instruction :

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

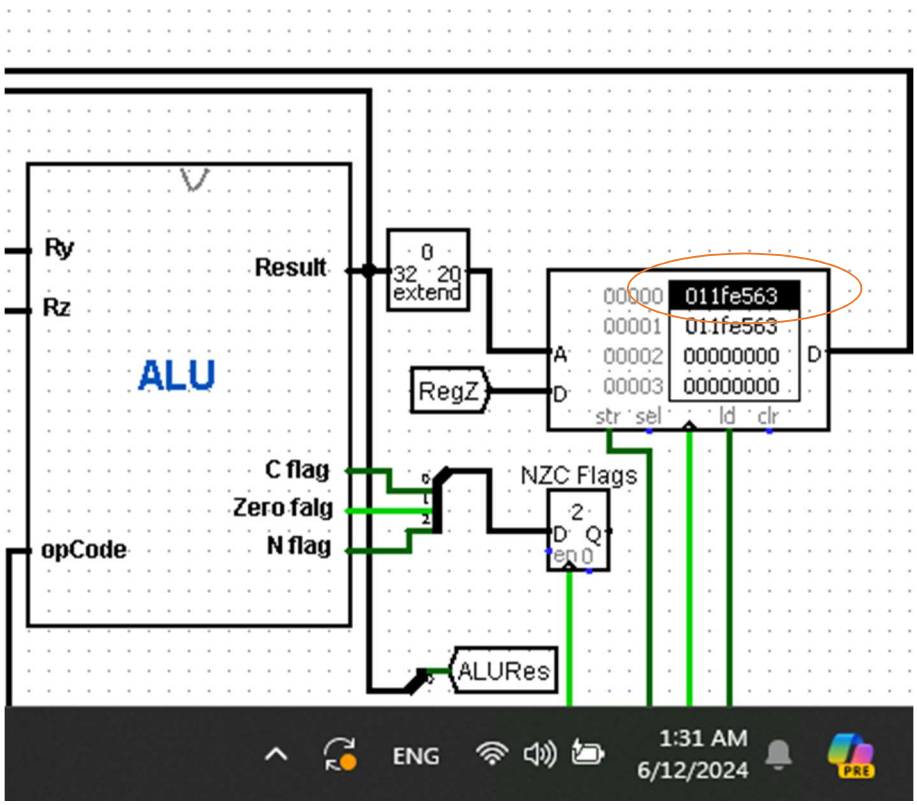
Then, Mem[0x 00000] = 0000 0001 0001 1111 1110 0101 0110 0011

= 0x 011FE563

Before Store :



After Store :



Simple While Loop Test with if-else Statements :

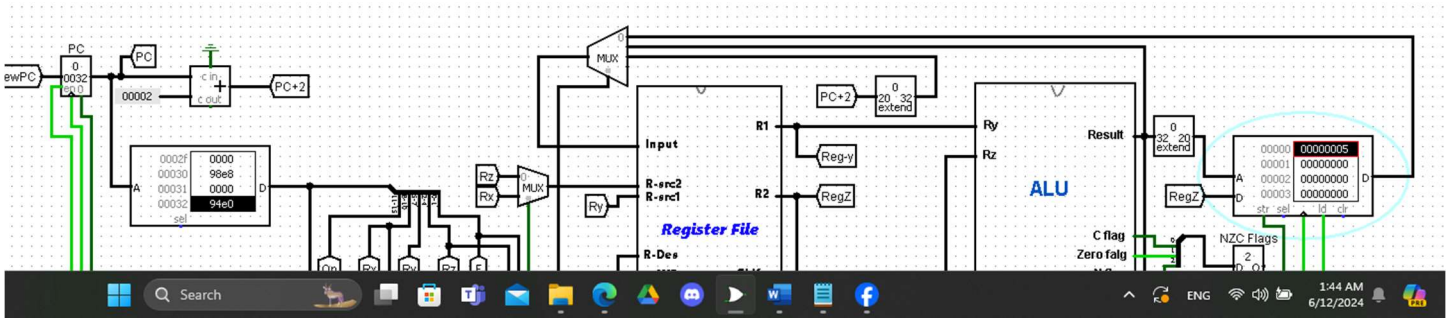
{Test the Branches}

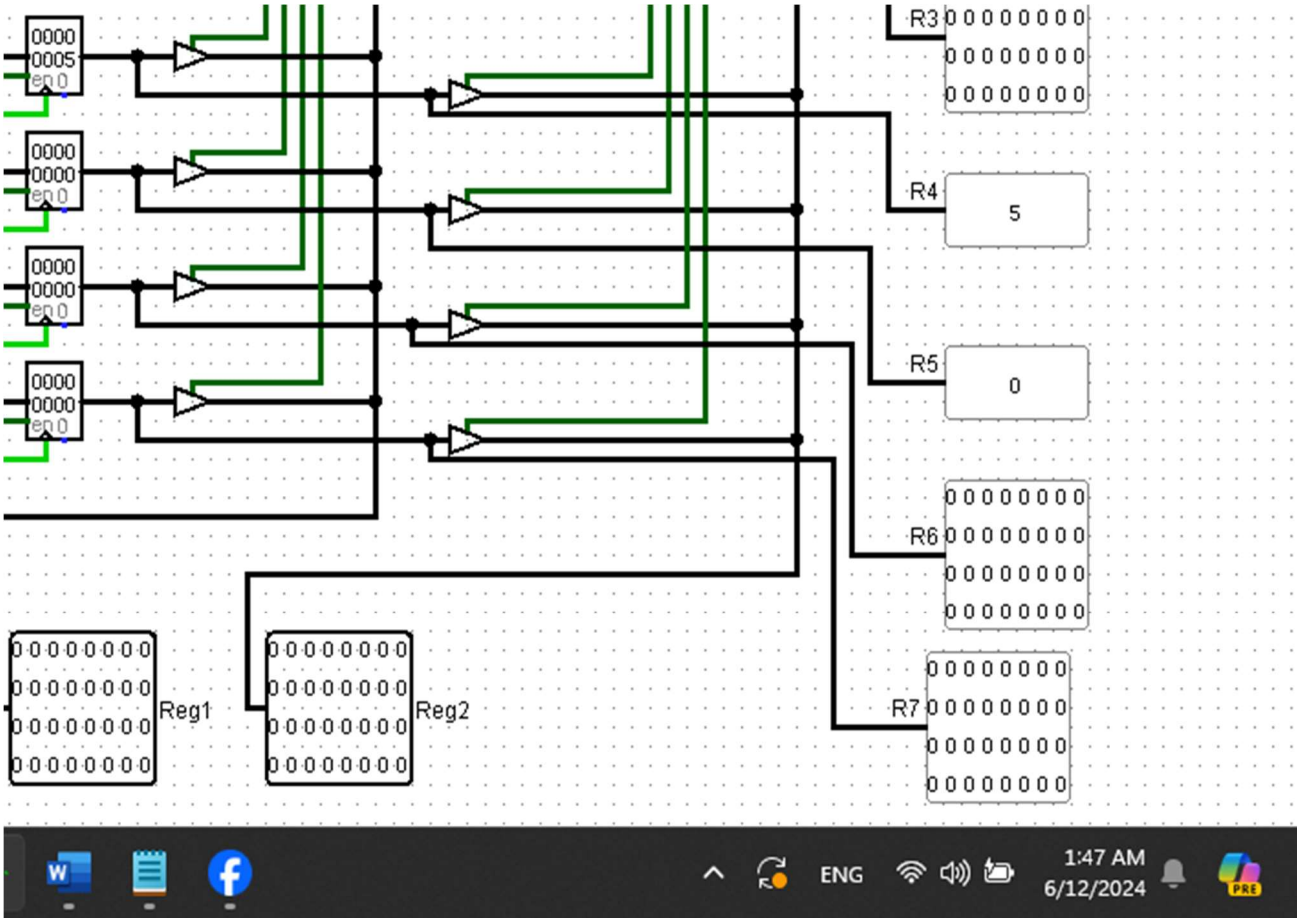
0x 00032 LW R4,R7,0 ➔ 0x 94E0

0x 00034 LW R5,R7,0 ➔ 0x 95E1

These instructions load from Memory address 0 to R4, and memory address 1 to R5, because R7 contains 0.

So, R4 will contains 5 in decimal (0x 00000005) and R5 will contains 0 (0x 00000000)





```
while(R5 != R4){
        if(R5 > R4)
                R5 = R5 - 1
        else
                R4 = R5 - 1
}
```

0x 00036 : BEQ R5,R4,6 ➔ 0111 0101 1000 0110 ➔ 0x 7586

0x 00038 : BLT R5,R4,2 ➔ 1000 0101 1000 0010 ➔ 0x 8582

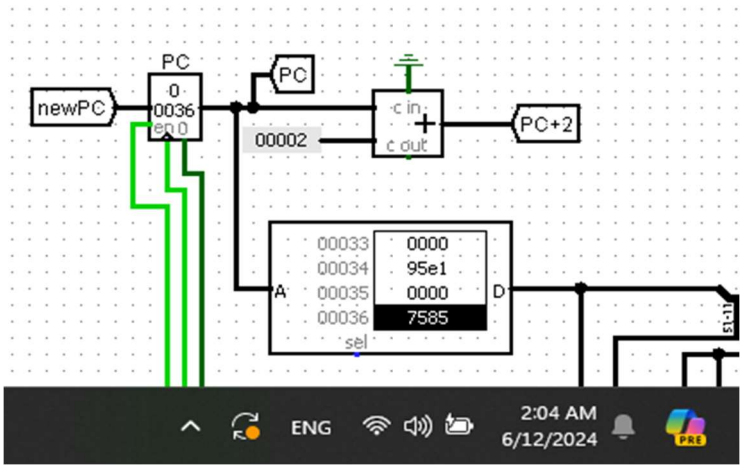
0x 0003A : ADDI R5,R5,-1 ➔ 0011 0101 1011 1111 ➔ 0x 35BF

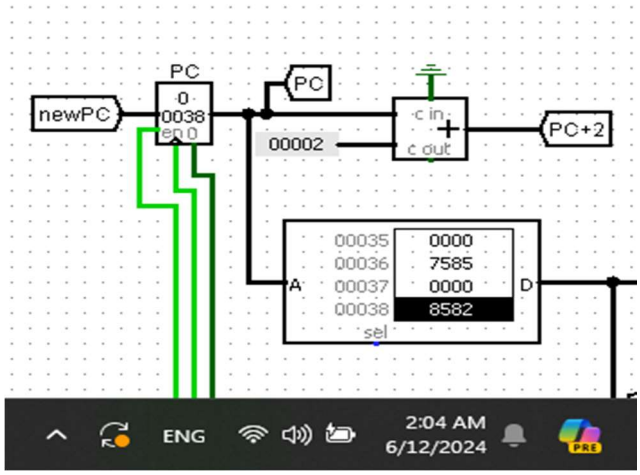
0x 0003C : J 1 ➔ 1010 0000 0000 0001 ➔ 0x A001

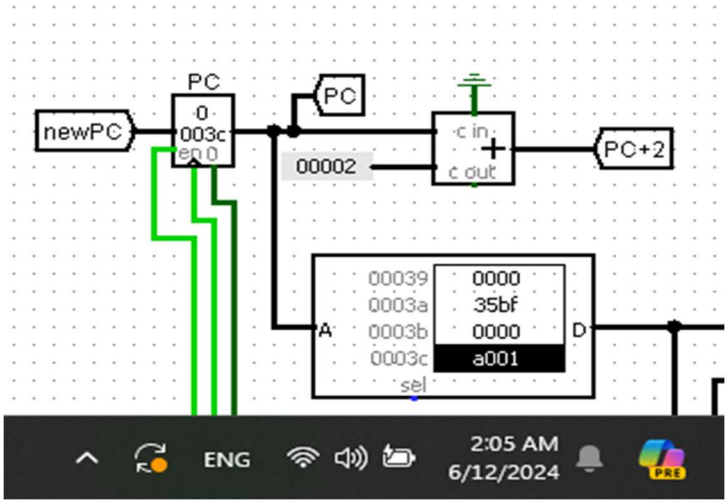
0x 0003E: ADDI R4,R4,-1 ➔ 0011 0100 1001 1111 ➔ 0x 349F

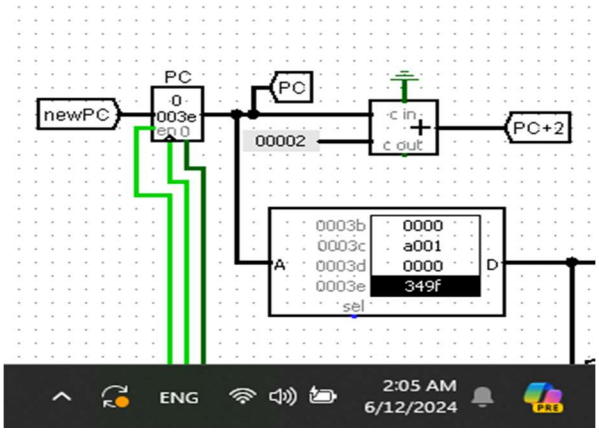
0x 00040: J -5 ➔ 1010 0111 1111 1011 ➔ 0x A7FB

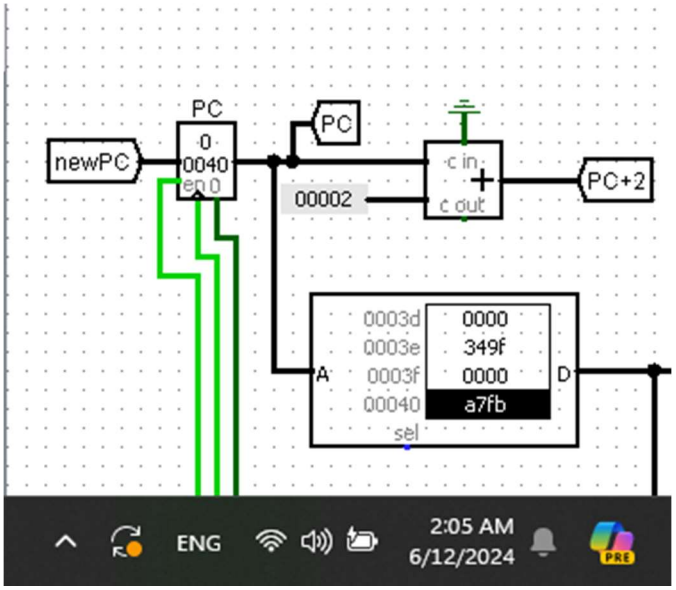
The Sequence of simulate as following :

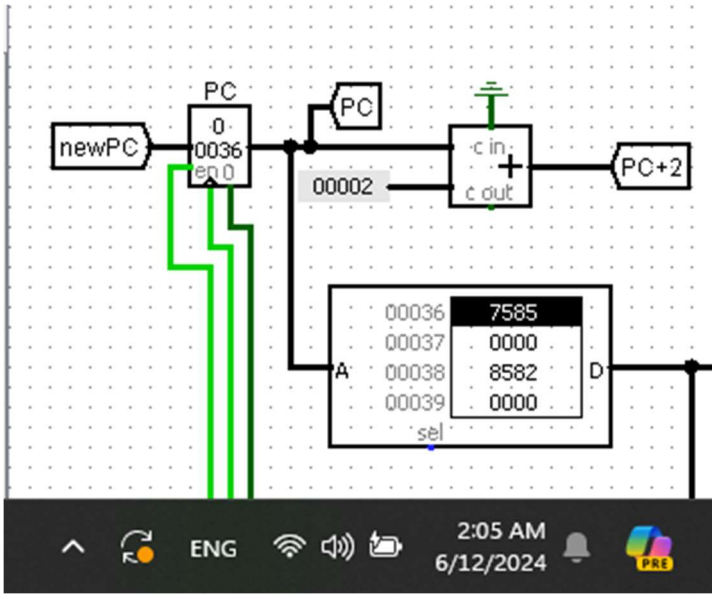


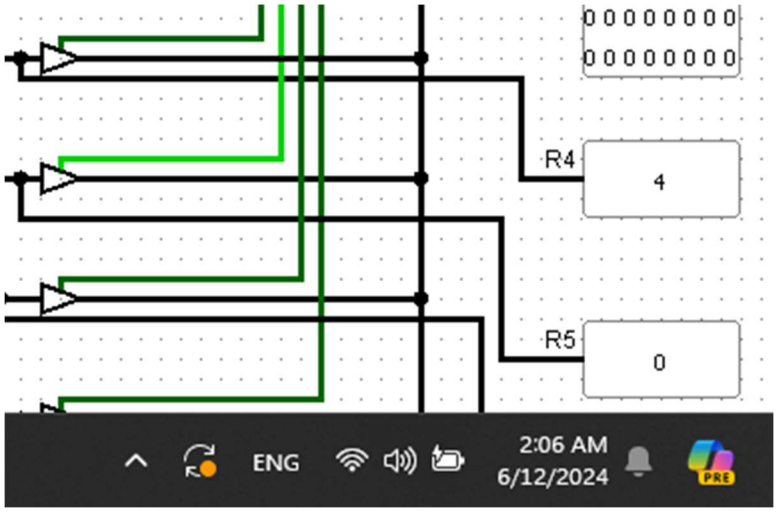






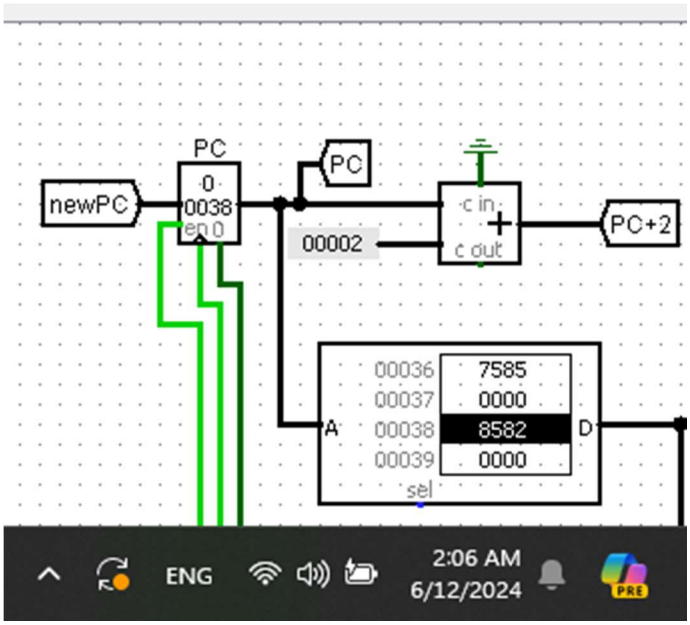


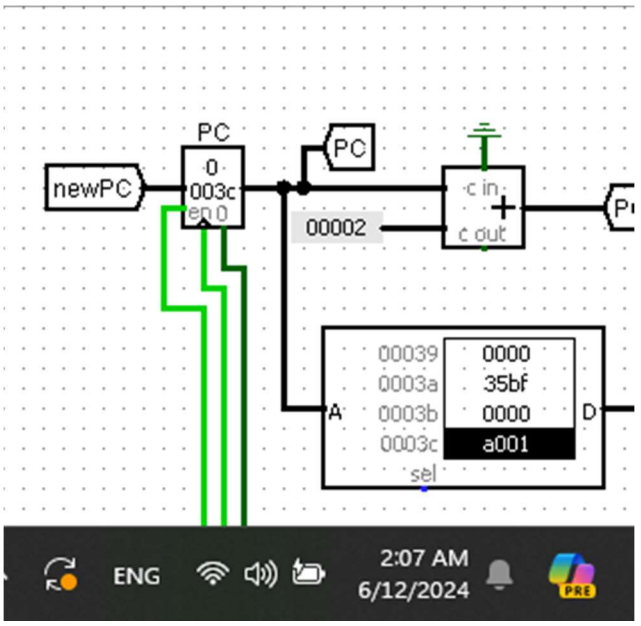


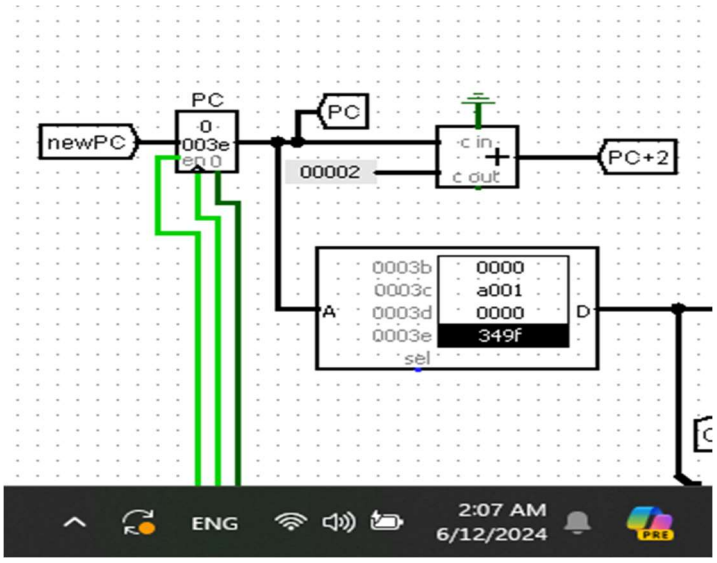


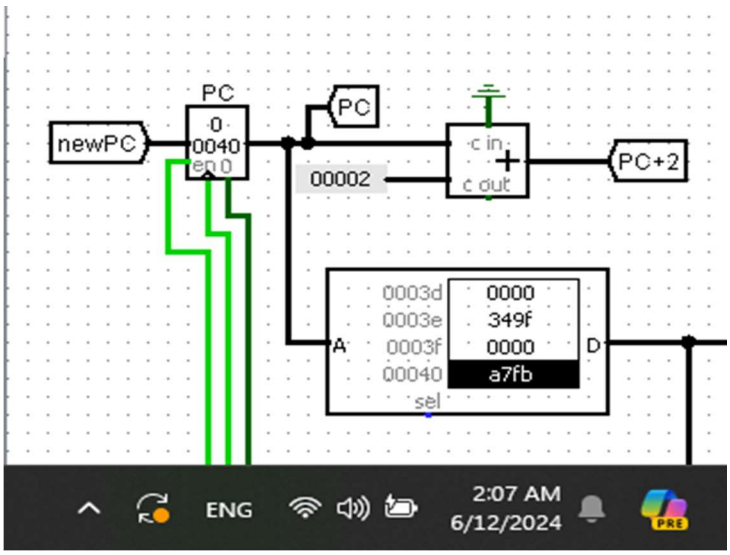
After the first loop

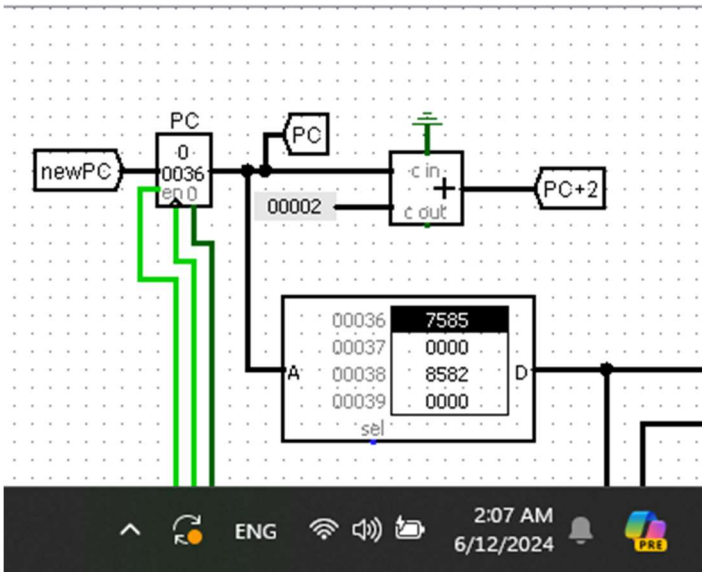
The Second Loop :

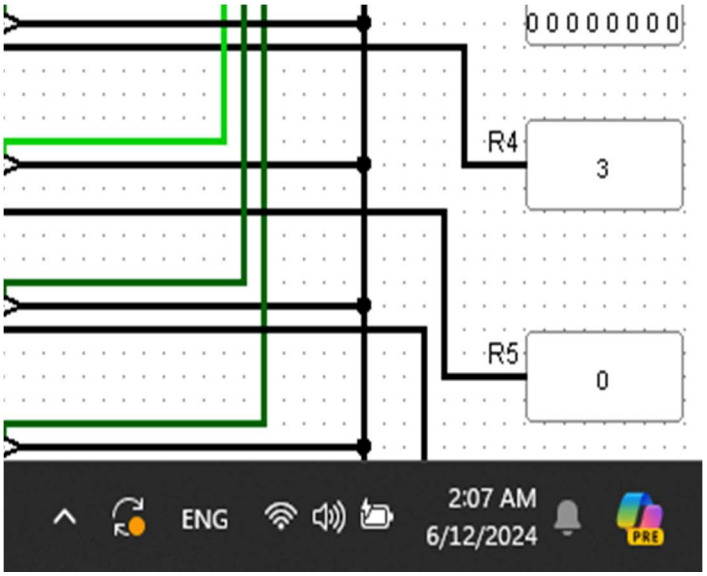




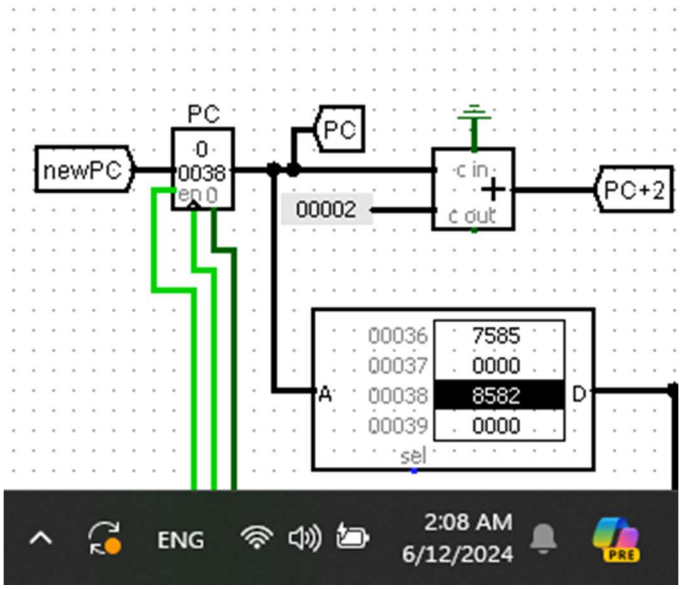


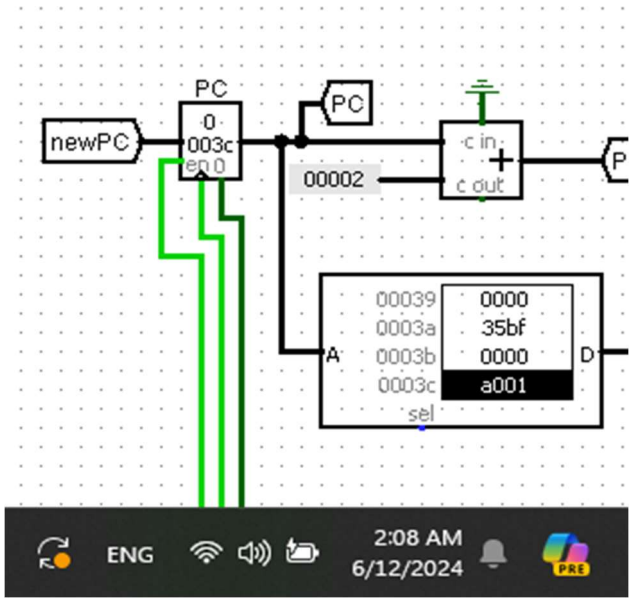


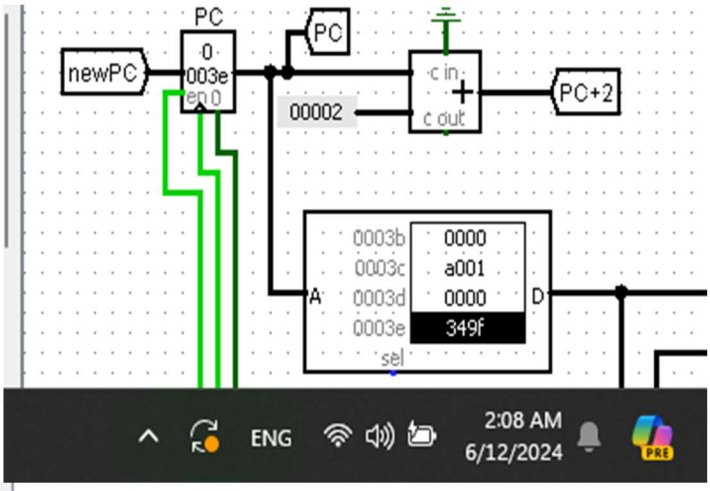


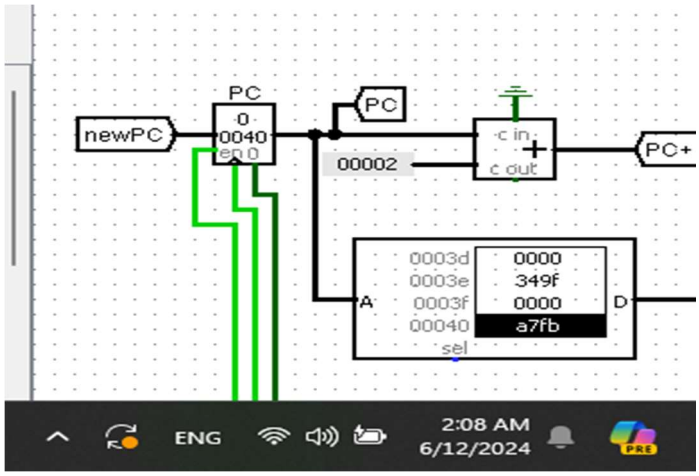


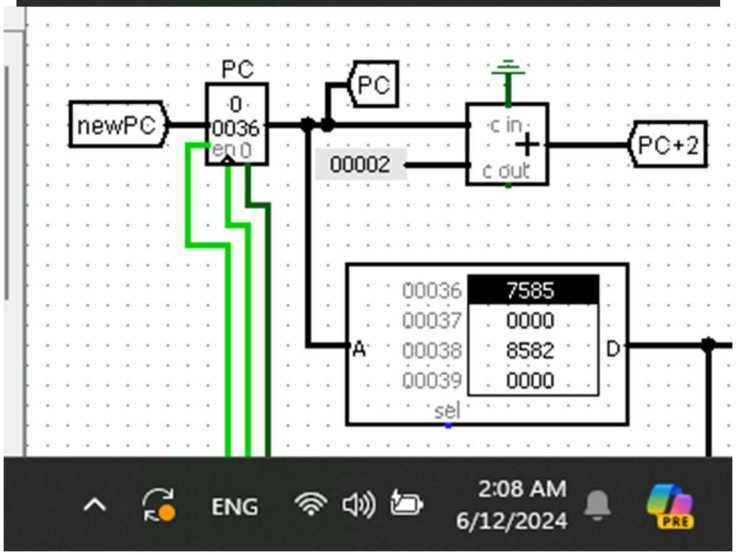
The Third Loop :

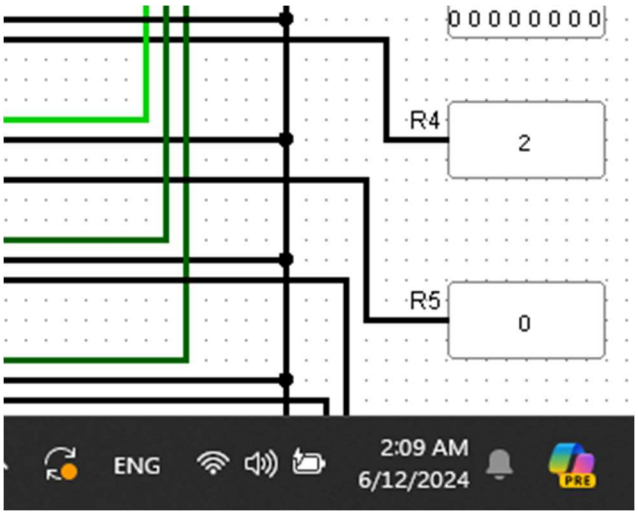




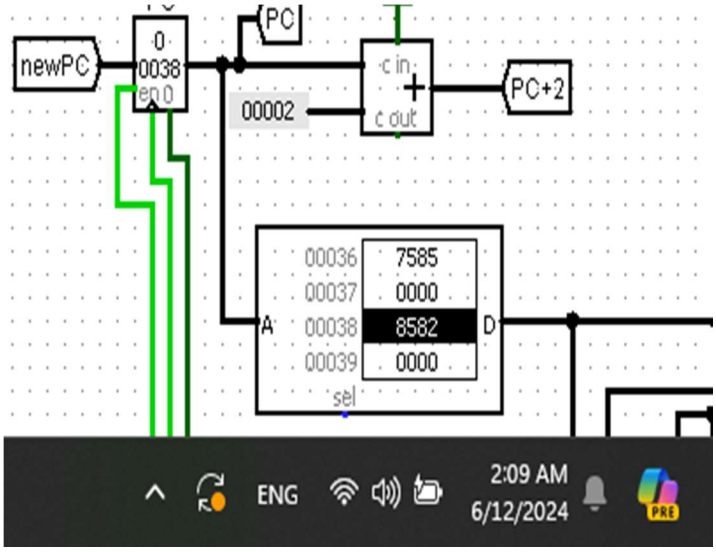


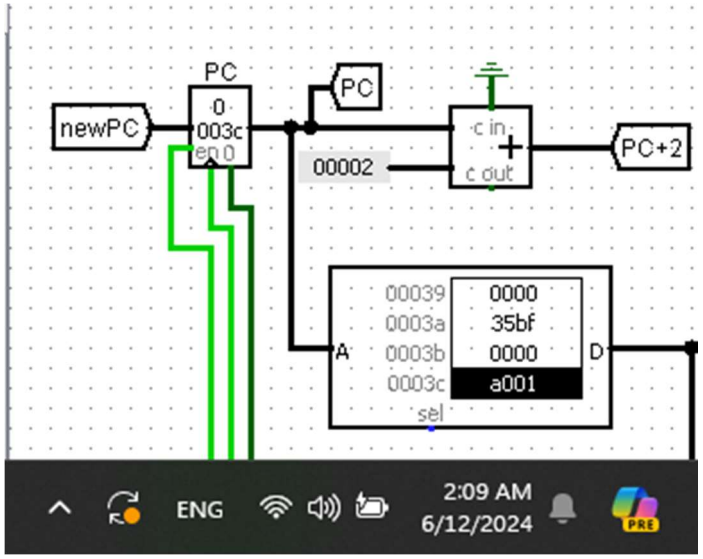


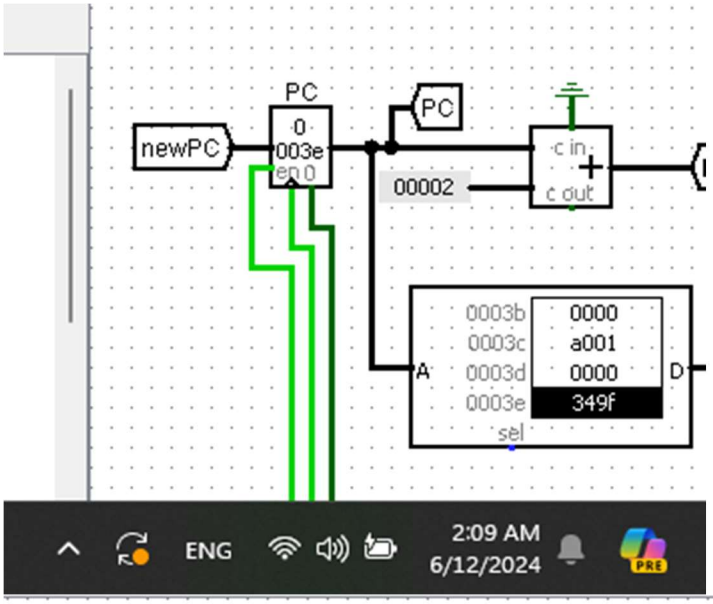


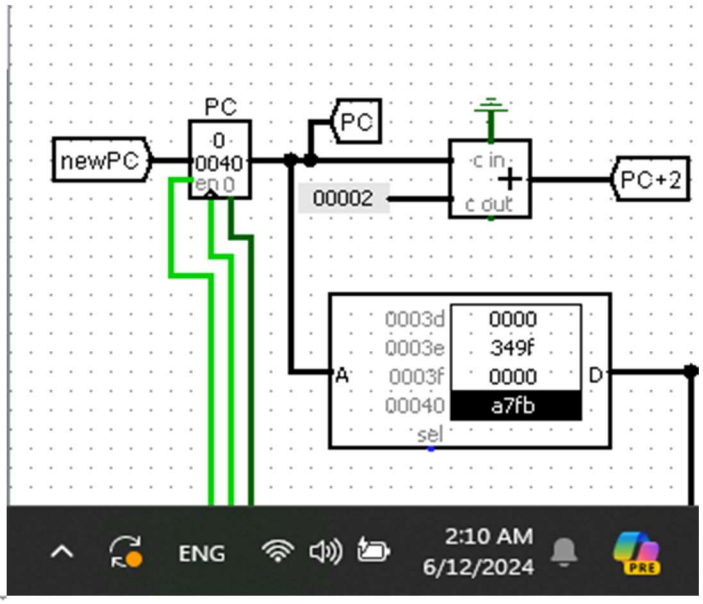


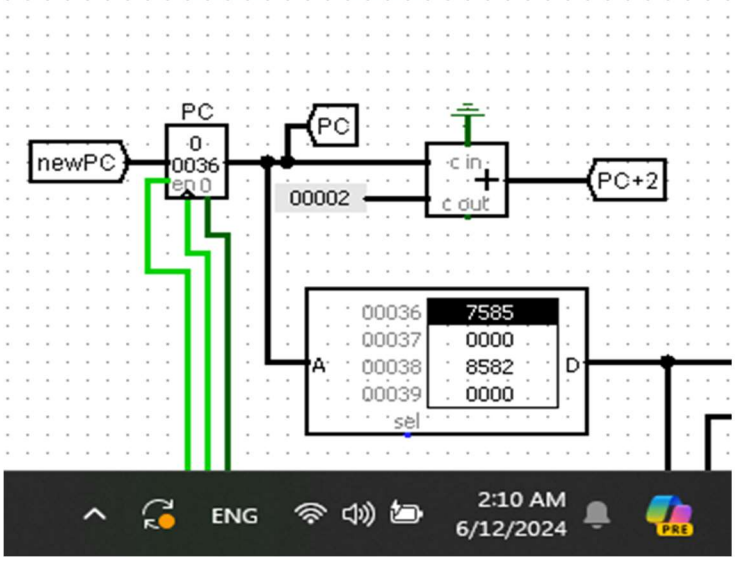
The Fourth Loop :

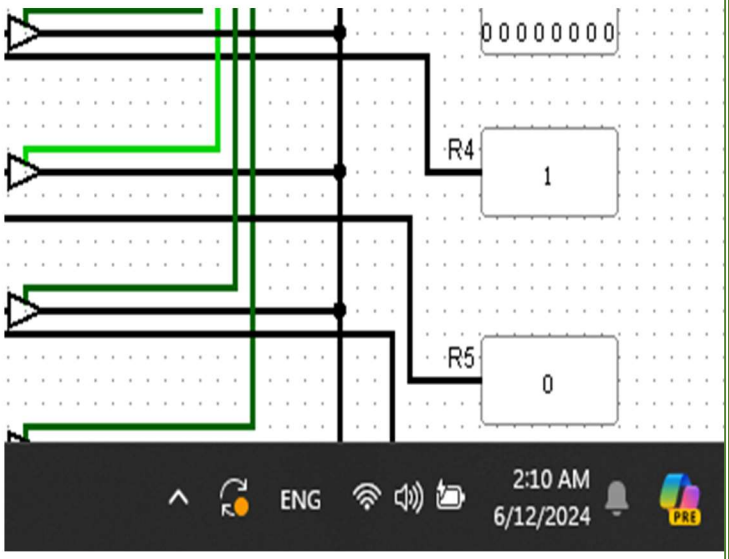




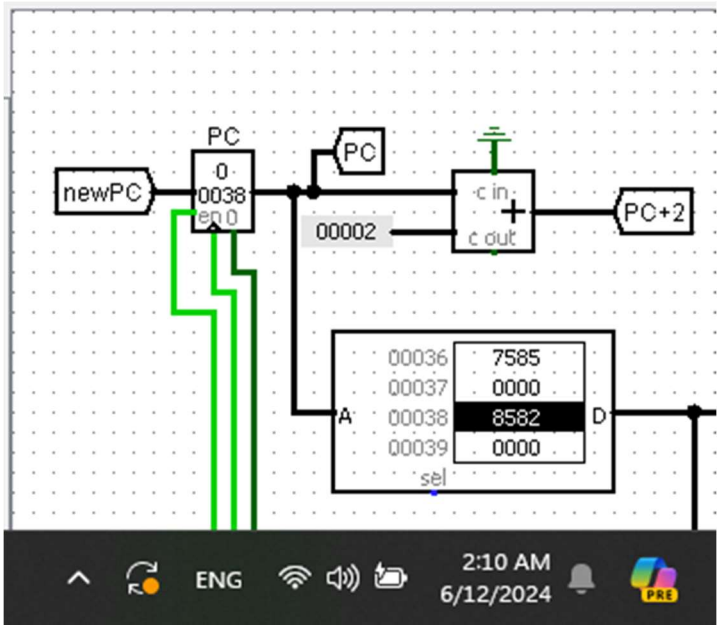


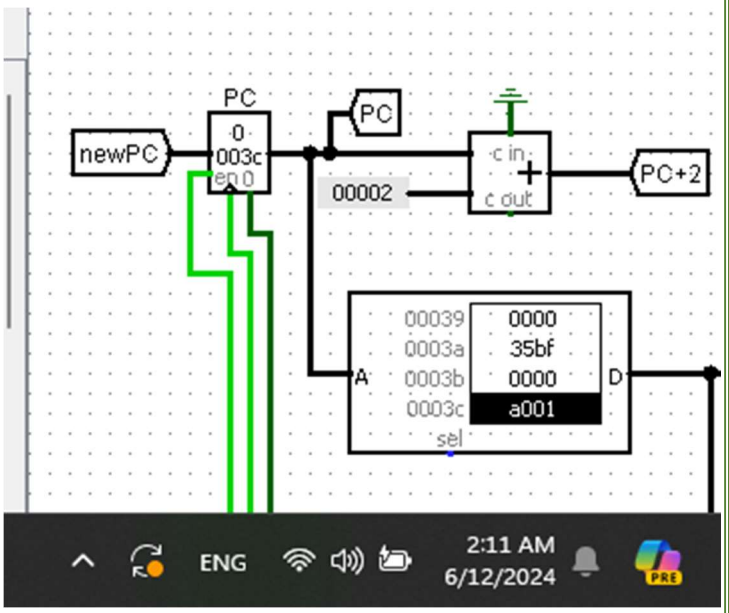


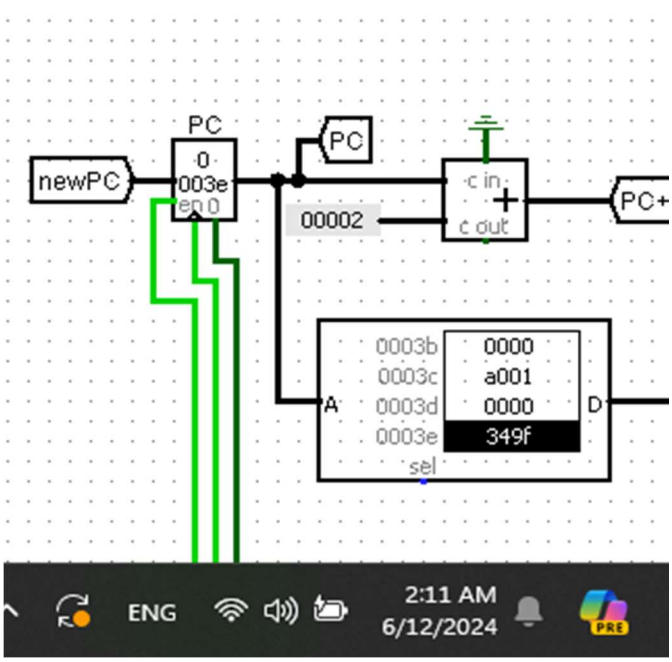


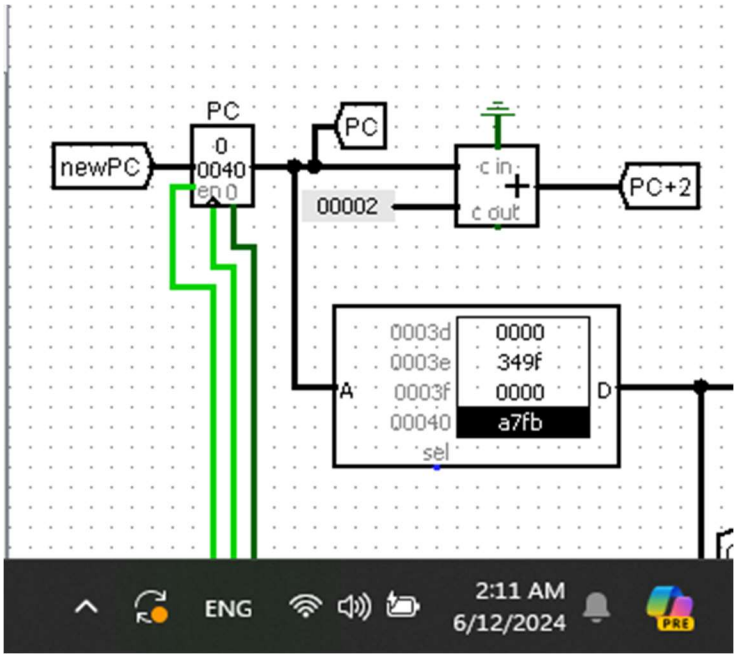


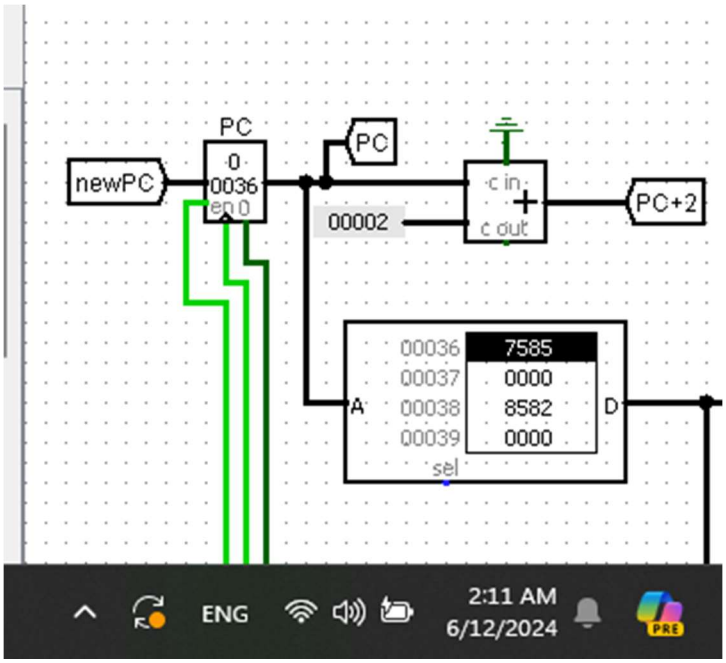
The Fifth Loop :

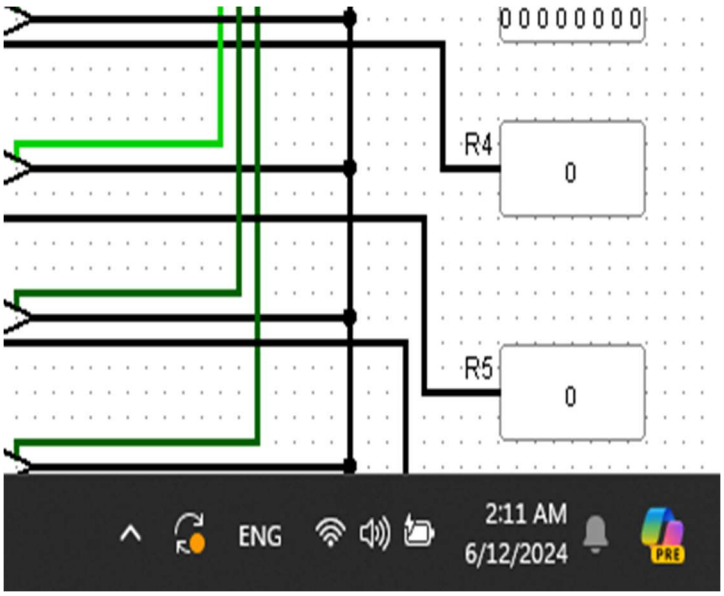


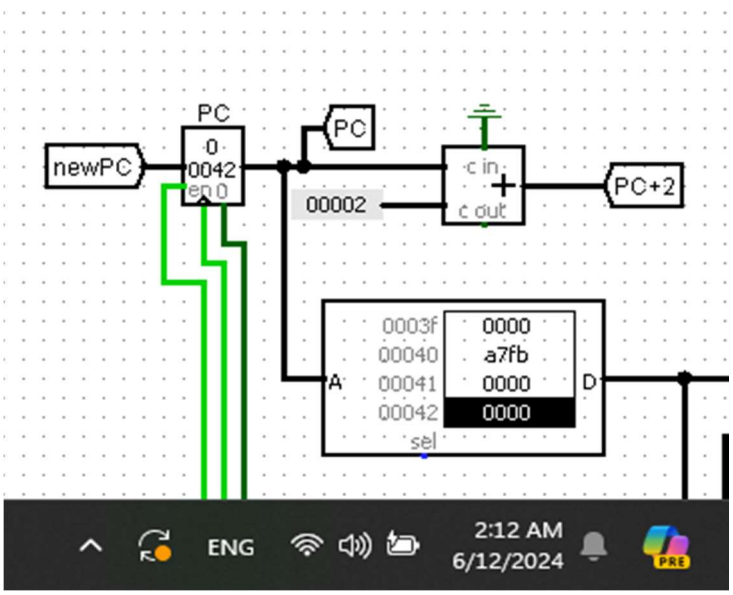












Then (0 = 0) the condition for loop is false, so branch outside the loop to the address 0x 00042

Issues and Limitations Part:

we have two primary things in procedure (subroutine) : **call** and **return** .

Call verifies without problem , when we have a procedure we will call an address of this procedure , and store the value of pc (next instruction) in R7 so this can be done with instruction **JAL** without any problem.

The problem will be in **return,** When we want to return from procedure we can't do this .because the PC must return to the address that stored in R7 and there is no instruction to do that in our processor.

To solve this problem we can add an instruction to transform the value of R7 to the PC register .

Teamwork Part:

Names:	Work:	Teamwork:
Mohammed Saada	ALU, Test cases	Text file (instructions) , Report
Nawras Yaqoob	Register File ,part of Datapath	
Manar Zitawi	Main control unit, part of Datapath	