Faculty of Engineering and Technology Electrical and Computer Engineering Department

COMPUTER ORGANIZATION AND MICROPROCESSOR (ENCS2380)

Project: Single Cycle Processor

Prepared by:

Student Name: Sama Alkhader **Student NO.** 1230079

Student Name: Sahar Hani **Student NO.** 1230971

Instructor: Dr. Ibrahim Nemer

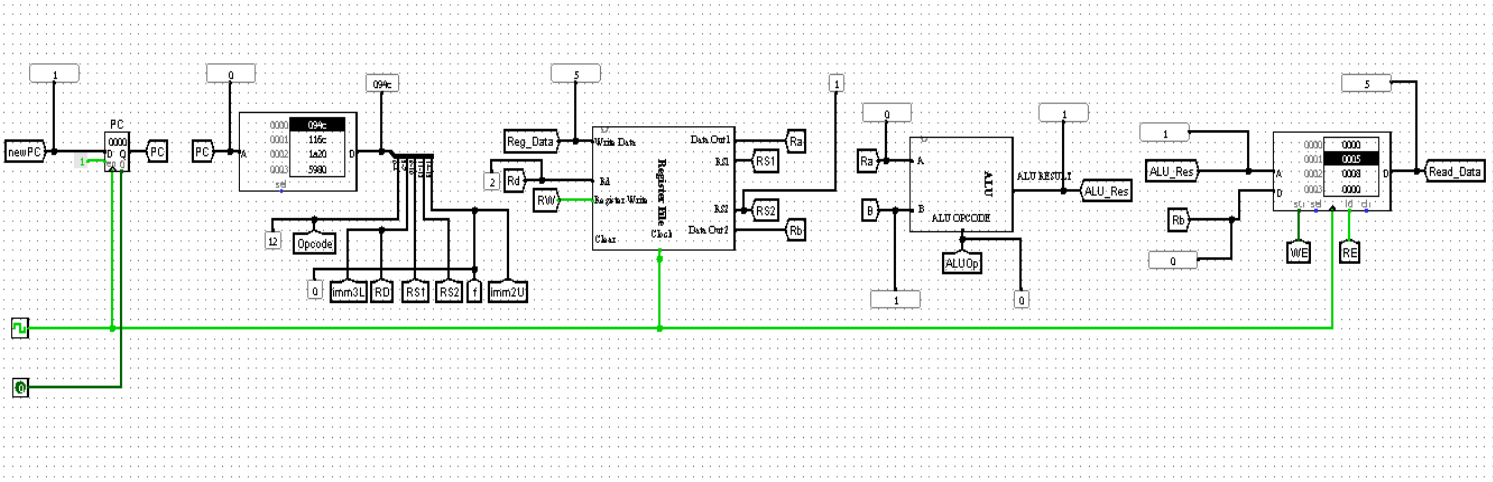
Section: 1

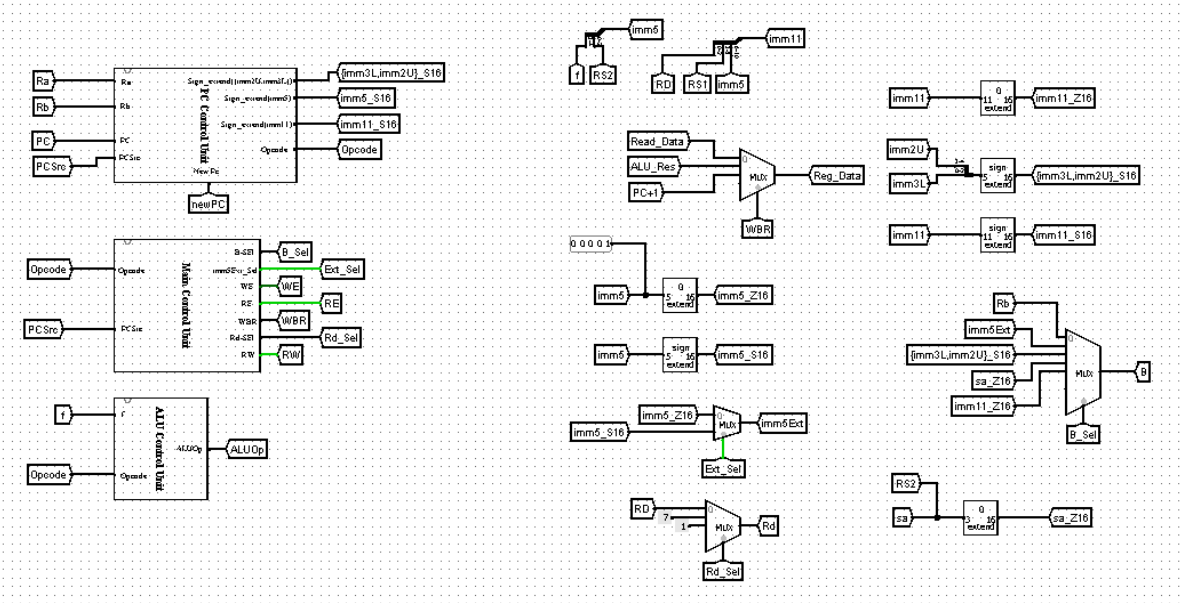
Date: 25/12/2024.

Design and Implementation

In this project, a 16-bit single cycle processor was designed using Logisim software. The processor can implement four instruction formats: R-type, I-type, SB-type, and J-type instructions. These instructions can perform different types of operations, which involve arithmetic and logical operations, branching and jumping, and memory accessing mechanisms.

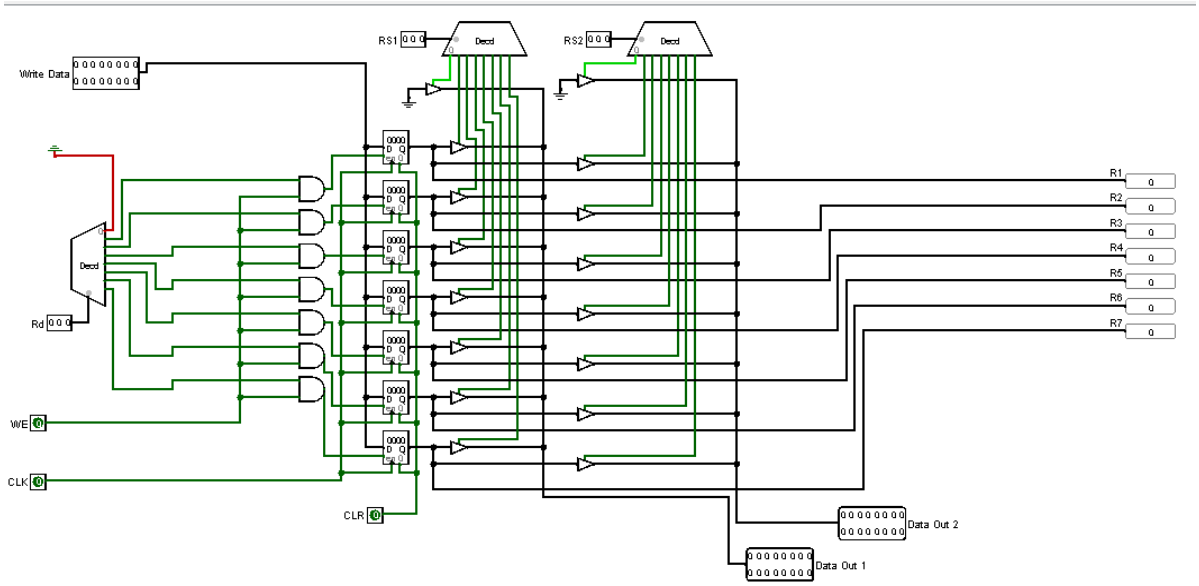
Overall Datapath implementation:





The main components of the single cycle processor design are:

1-Register File:



The register file is the part of the processor responsible for storing the data used during the execution of a program. It's considered quicker to access than the memory, which is why in most cases, the registers are used to store the operands of an instruction.

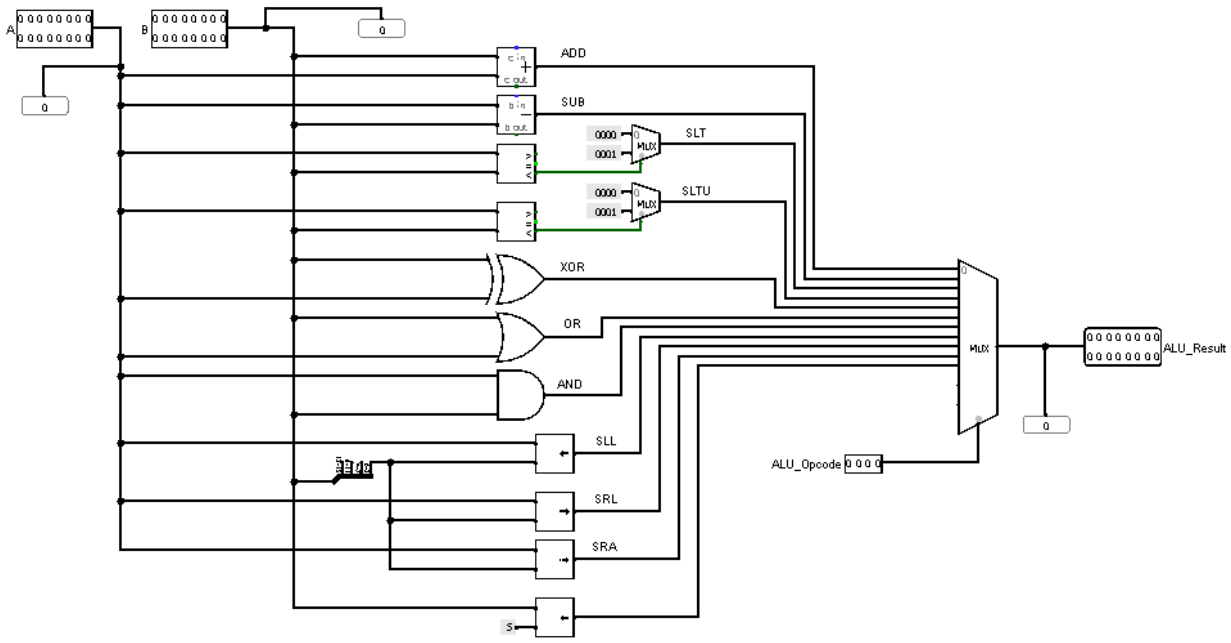
In our design, the register file is made up of 8 16-bit registers (R0-R7), 3 3-8 decoders, two output ports called data out 1 and data out 2, two input ports RS1 and RS2, and a Rd input port.

At the rising edge of the clock, input port Rd is used to choose the register which we want to write on, and if the write enable input is high (WE=1), the data on the write data input is stored in the specified register.

However, to access the stored data, output ports RS1 and RS2 are used. The number of the register that has the operands is specified using RS1 for register A, and RS2 for register B, and with the help of the decoders, which enable the chosen registers for read, the data on these registers is fetched using output ports data out 1 and data out 2, making it available for other components to use.

Additionally, all registers in the file can be read or written except for R0, which is a special register that can be read only because it's always hardwired to 0. Finally, all registers are connected to a clear (CLR) input, which is used to clear the values of all registers when it's high.

2-ALU (Arithmetic and Logic Unit):



The ALU is responsible for performing all arithmetic and logical operations in the processor. It depends mainly on the ALU opcode obtained from the main control unit to specify the operation to be executed.

The ALU was designed using two input ports (which receive the operands A and B), some logic gates (XOR, AND,OR), 16-bit arithmetic and logical shifters, and a 16-bit output port that stores the result and sends it to other parts of the processor.

In our design, the ALU is capable of performing arithmetic operations, such as addition and subtraction when provided with the opcode. Also, it performs logical operations using the logic gates XOR, AND and OR. Moreover, it is capable of performing arithmetic and logic shifting, whether it's left or right. Finally, the ALU compares two values and decides if they are greater than, less than or equal.

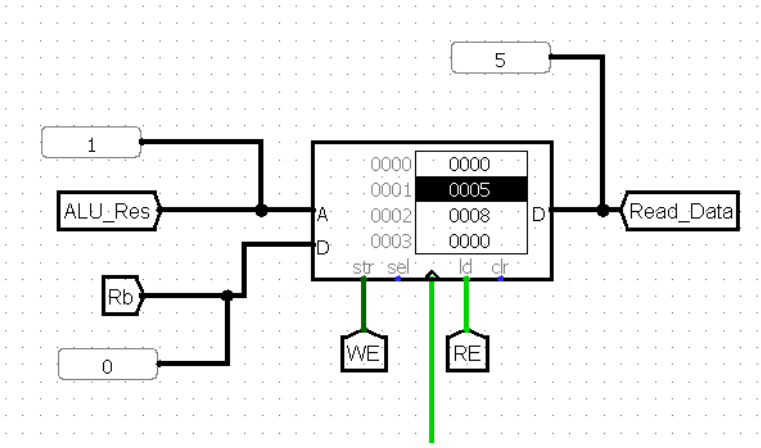
The ALU receives input A directly from the register file (data out 1), while input B is received from a mux, which chooses between: register B, sign extended imm11, sign extended imm2U,imm3L, sign extended imm5, and the shift amount. This allows the ALU to use immediate values to perform its operations.

Below, a table that shows the ALU opcode control signals description.

NOTE: we needed to implement 11 operations, so we used a 16x1 MUX.

ALU_Opcode encoding	Arithmetic or logical operation	Meaning
0000	ADD	Rd=Ra+Rb
0001	SUB	Rd=Ra-Rb
0010	SLT	Rd= (Ra<Rb)
0011	SLTU	Rd= (Ra<Rb) unsigned
0100	XOR	Rd= Ra^Rb (bitwise XORing)
0101	OR	Rd= Ra | Rb (bitwise ORing)
0110	AND	Rd= Ra & Rb (bitwise ANDing)
0111	SLL	Rd=ShiftLeftLogical(Ra, Rb[3:0])
1000	SRL	Rd=ShiftRightLogical(Ra, Rb[3:0])
1001	SRA	Rd=ShiftRightArith(Ra, Rb[3:0])
1010	LUI	R1= imm11 << 5 Shift (imm11) left by 5-bits
1011-1111	XXXX	Don't care

3-Data Memory:



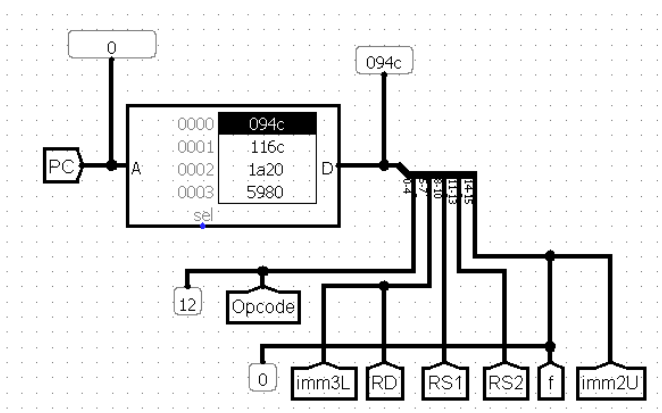
The data memory is the part of the processor where data is stored. During program execution, the ALU calculates the effective address, and depending on the instruction, the data memory either accepts data and stores it in the specified address (Store instruction) or loads the data from the specified address to the register file (Load instruction).

In our design, the data memory is implemented using 16-bit RAM, two 16-bit input ports, (write data and address), three 1-bit output ports (clock, clear, read enable, write enable), and one 16-bit output port (read data).

The data memory receives the effective address calculated by the ALU. At the rising edge of the clock, if the write enable is high, the memory writes the data on the write data port and stores it in the specified address. However, if memory read was high, the data on the specified address is sent to the read data output.

Additionally, the data memory is byte-addressable, so each location corresponds to a 16-bit word, and the size of the data memory is 128KB.

4-Instruction Memory:



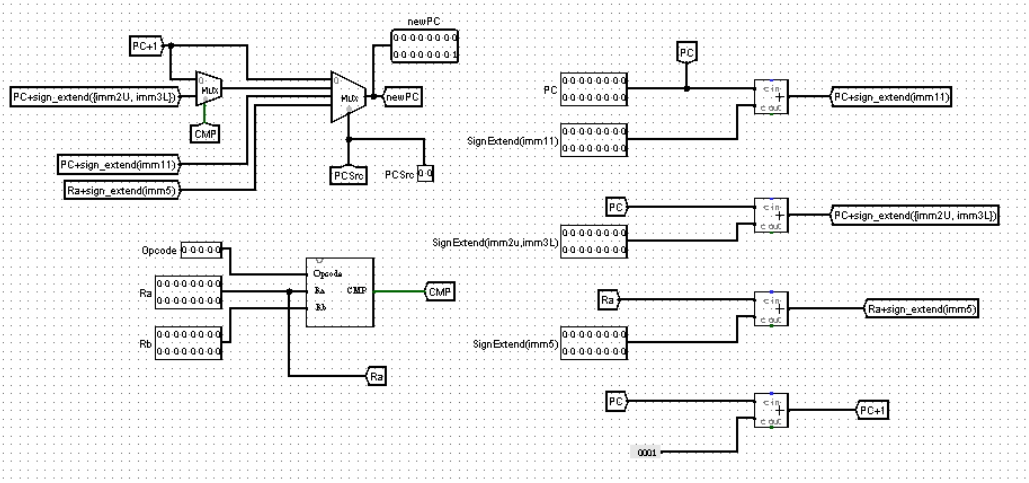
The instruction memory stores the instructions of the single cycle processor. It receives the address of the instruction from the PC, and locates the instruction in ROM, and then splits it to five parts: the 5 least significant bits represent the opcode, bits 5-7 represent the Rd register for I-type and R-type instructions, but in SB type they represent imm3L, and in J-type they represent the first three bits of imm11. Additionally, bits 8-10 represent RS1 in I-type, R-type and SB-type instructions, but represent the bits 8-10 of imm11 in J-type instructions.

Next, bits 11-13 represent RS2 for R-type and SB-type instructions, the first 3 bits of imm5 in I-type instructions, and bits 11-13 of imm11 in J-type instructions. Finally, the last two bits (14 and 15) represent f for R-type, the last two bits of imm5 in I-type, imm2U for SB-type, and the last two bits of imm11 in J-type.

In our design, the instruction memory uses output ports that connect the splitted part of the instruction to the appropriate destination.

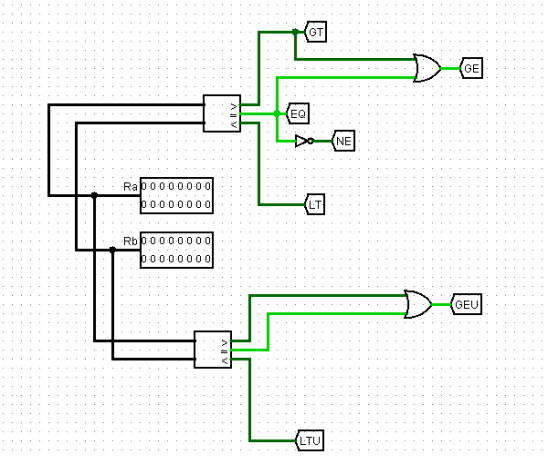
5-PC-Control Unit:

The PC-Control unit calculates the address of the next instruction based on a control signal it receives from the main memory specifying the instruction type. However, the PC-control is made up of the following parts:



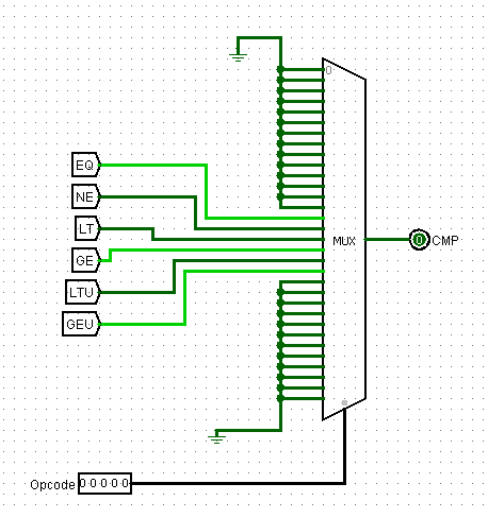
1-Comparator unit

a-Comparator:



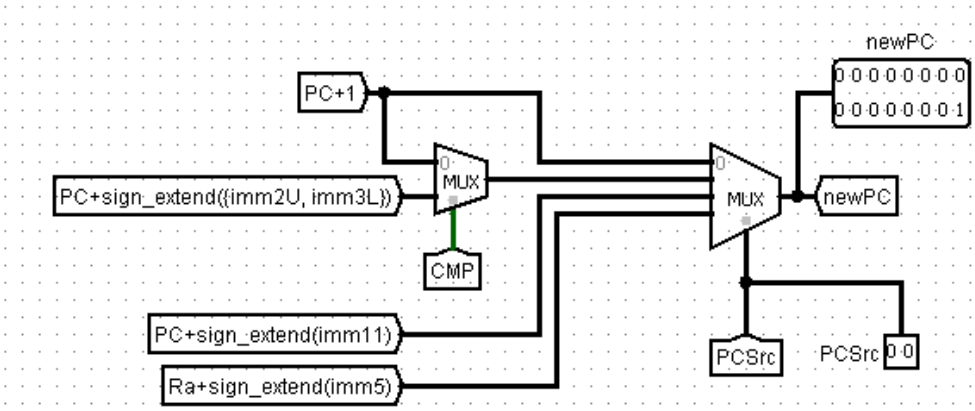
This comparator receives two inputs from register A (Ra) and register B (Rb) and compares their values. It decides whether the value in register A is greater than, greater than or equal, equal, less than, less than or equal to the value of register B.

b-Mux:



The mux receives the outputs of the comparator, which are connected to the instruction number they correspond to. Based on the opcode, the mux decides which value is the one to be sent to the output port. For example, if the opcode is 14 (which corresponds to the BEQ instruction) and the greater than signal from the comparator is 1, then CMP would take this value to the final mux, which calculates the new PC address based on the instruction type and if the condition is met.

2-New-PC Deciding Mux:

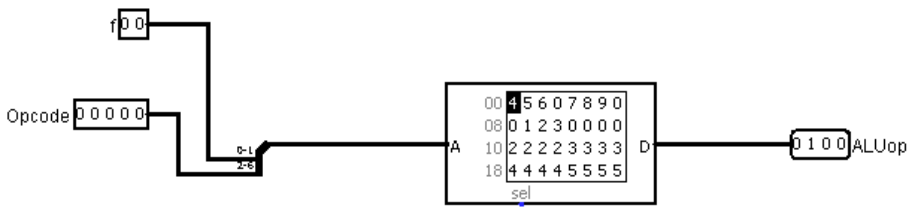


This mux receives the PC opcode signal and treats it as a selection line that decides the new PC address based on the instruction type. On input 00 of the mux, there's PC+1, which tells the processor to move to the next instruction if the previous instruction was of type R or type I. On input 01, there's a 2-1 mux that uses CMP as a selection line. If CMP is 0, this means that the branching condition was not met, and the processor should move to the next instruction. On the other hand, if CMP is 1, it tells the processor to branch to a new address calculated by adding the last value of the PC to the concatenation of imm2U and imm3L because the branching condition was met.

However, on input 10 of the mux, we have PC+sign_extend(imm11), which corresponds to the jumping address. This type of branching is unconditional, so the processor simply jumps to this address if the PC opcode signal represents a J-type instruction. Finally, the value on input 11 of the mux (PC= Ra+sign_extend(imm5)) is used for the Jump-and-Link instruction, which saves the return address in Rd register and jumps to the new address.

In our design, the PC-control unit receives the previous PC, and an input signal from the main control unit, representing a special code that specifies the type of the instruction, and based on it the new PC address is calculated and sent to the output port (New PC), which sends it directly to the instruction memory.

6-ALU-Control Unit:



This unit is used to control the ALU operation based on the opcode. It decides which operation should be performed by the ALU by taking the opcode and f from the instruction memory, and using them to produce an output called ALUOP, which is connected directly to the ALU mux that decides between operations.

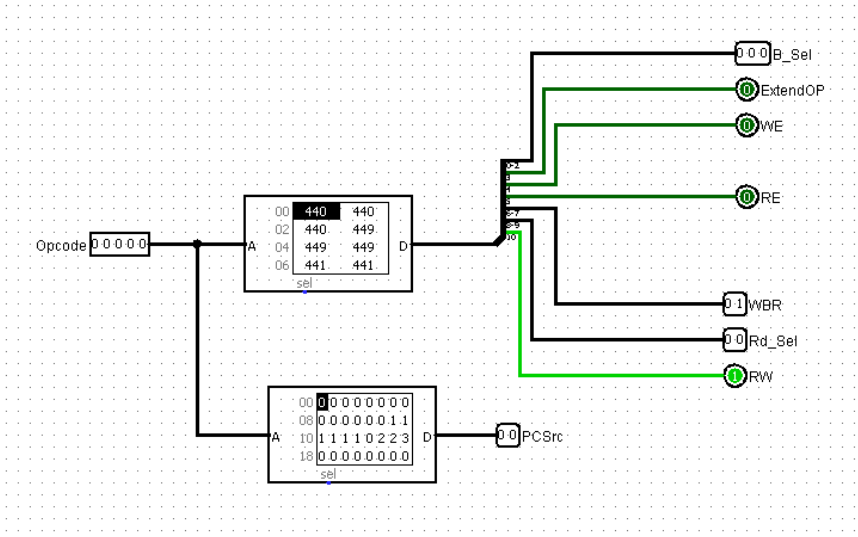
For example, if the opcode is 00000 and f is 00 (which corresponds to the XOR instruction), this unit produces 0100. This value is then sent to the ALU telling it to perform an XOR operation on values A and B.

The following table shows the ALU opcode generated for each instruction:

Instruction	Opcode	f	HEX
XOR	00000	00	0x00
OR	00000	01	0x01
AND	00000	01	0x02
SLL	00001	00	0x04
SRL	00001	01	0x05
SRA	00011	10	0x06
ADD	00010	00	0x08
SUB	00010	01	0x09
SLT	00010	10	0x0A
SLTU	00010	11	0x0B
LUI	10100	XX	0x50 or 0x51 or 0x52 or 0x53
ADDI	00011	XX	0x0C or 0x0D 0r 0x0E or 0x0F
SLTI	00100	XX	0x10 or 0x11 or 0x12 or 0x13
SLTIU	00101	XX	0x14 or 0x15 or 0x16 or 0x17
XORI	00110	XX	0x18 or 0x19 or 0x1A or 0x1B

ORI	00111	XX	0x1C or 0x1D or 0x1E or 0x1F
ANDI	01000	XX	0x20 or 0x21 or 0x22 or 0x23
SLLI	01001	XX	0x24 or 0x25 or 0x26 or 0x27
SRLI	01010	XX	0x28 or 0x29 or 0x2A or 0x2B
SRAI	01011	XX	0x2C or 0x2D or 0x2E or 0x2F
LW	01100	XX	0x30 or 0x31 or 0x32 or 0x33
SW	01101	XX	0x34 or 0x35 or 0x36 or 0x37

7-Main-Control Unit:



The main control unit sends signals to different parts of the processor to control the execution of the program. First, it decodes the instruction based on the opcode, and decides where in memory the decoded instruction is stored. Then, it generates signals that enable or disable other components.

The description of the control signals generated by the (upper) general control unit are as following:

Control signal	**Number of bits**	**Description**	**Signals meaning**
B_Sel	3	Determines the second source (operand) of the ALU	000 → Rb 001 → imm5 (extended) 010 → {imm3L,imm2U} (sign extended) 011 → sa (shift amount) 100 → imm11 (zero extended) 101-111 → don't care
imm5Ext_Sel	1	Determines whether to extend imm5 by zeros or by sign	0 → imm5 zero extended 1 → imm5 sign extended
WE (Memory write enable)	1	Enables write on memory (for STORE instruction)	0 → no WRITE 1 → WRITE
RE (Memory read enable)	1	Enables read from memory (for LOAD instruction)	0 → no READ 1 → READ
WBR	2	Determines what data to be written back on register file	00 → Data read from memory 01 → ALU result 10 → PC+1 11 → XX
Rd_Sel	2	Determines the destination register	00 → RD 01 → 7 10 → 1 11 → XX
RW	1	Enables WRITE on register file	0 → WRITE on rfile 1→ no WRITE on rfile

Additionally, the second ROM in the main unit is used to generate a single 2-bits signal based on the next PC state called PCSrc. This signal is connected directly to the PC-control unit, and it generates the following signals:

Instruction	PCSrc Signal
R-type, I-type, and LUI instruction	00
SB-type	01
J-type (J and JAL)	10
J-type (JALR)	11

NOTE: since only four PC-update states were present in this SCP design, we chose not to increase the complexity of our implementation (e.g., by adding more control signals or taking other more complex approaches).

The following table shows how each instruction is decoded, and the signals generated:Note: All R-type instructions follow the same format as the XOR.

	RW	Rd_Sel	WBR	RE	RE	imm5Ext_Sel	B_Sel	HEX
XOR	1	00	01	0	0	0	000	0x440
ADDI	1	00	01	0	0	1	001	0x449
SLTI	1	00	01	0	0	1	001	0x449
SLTUI	1	00	01	0	0	1	001	0x449
XORI	1	00	01	0	0	0	001	0x441
ORI	1	00	01	0	0	0	001	0x441
ANDI	1	00	01	0	0	0	001	0x441
SLLI	1	00	01	0	0	0	011	0x443
SRLI	1	00	01	0	0	0	011	0x443
SRAI	1	00	01	0	0	0	011	0x443
LW	1	00	00	1	X (0)	1	001	0x429
SW	X (0)	X (11)	X (11)	X	1	X (0)	010	0x3D2
BEQ	0	X (11)	X (11)	0	0	0	010	0x3C2

BNE	0	X (11)	X (11)	0	0	0	010	0x3C2
BLT	0	X (11)	X (11)	0	0	0	010	0x3C2
BEG	0	X (11)	X (11)	0	0	0	010	0x3C2
BLTU	0	X (11)	X (11)	0	0	0	010	0x3C2
BGEU	0	X (11)	X (11)	0	0	0	010	0x3C2
LUI	1	10	01	0	0	0	100	0x644
J	0	X (11)	X (11)	0	0	0	100	0x3C4
JAL	1	01	10	0	0	0	100	0x584
JALR	1	00	01	0	0	0	001	0x481

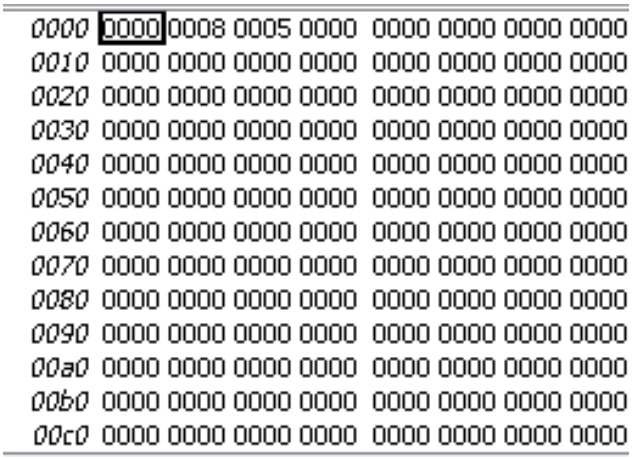
Simulation and Testing

NOTE: instructions #1 - #24 were executed sequentially,

Instruction Set 1 (R-type instructions):

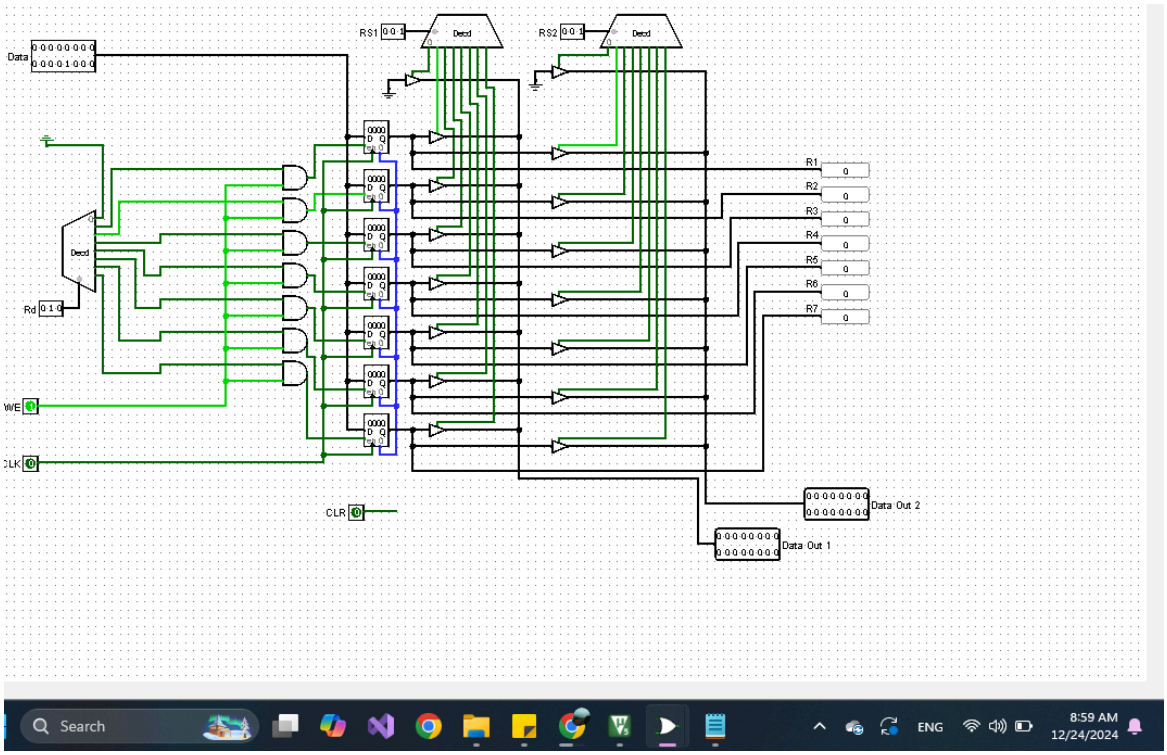
#	Address of instruction	Assembly	Binary	Hexadecimal
1	0x0000	LW R2, R1, #0x00001	00001 001 010 01100	0x094c
2	0x0001	LW R3, R1, #0x00002	00010 001 011 01100	0x116c
3	0x0002	XOR R1, R2, R3	00 011 010 001 00000	0x1a20
4	0x0003	OR R4, R1, R3	01 011 001 100 00000	0x5980
5	0x0004	AND R4, R1, R3	10 011 001 100 00000	0x9980
6	0x0005	SLL R5, R1, R3	00 011 001 101 00001	0x19a1
7	0x0006	SUB R5, R2, R3	01 011 010 101 00010	0x5aa2
8	0x0007	SRL R6, R1, R3	01 011 001 110 00001	0x59c1
9	0x0008	SRA R7, R1, R3	10 011 001 111 00001	0x99e1
10	0x0009	ADD R6, R4, R1	00 001 100 110 00010	0x0cc2
11	0x000A	SLT R6, R4, R1	10 001 100 110 00010	0x8cc2
12	0x000B	SLTU R6, R4, R1	11 001 100 110 00010	0xccc2

Data memory:



```
0000 094c 116c 1a20 5980 9980 19a1 5aa2 59c1 99e1 0cc2 8cc2 ccc2 e6e3 c4e4 c4e5 3ce6
0010 3ce7 3ce8 6483 1cc9 1cca 1ccb 756d 7b6e 636f 0000 0000 0000 0000 0000 0000 0000
0020 0000 0000 6670 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 636f 0000 0000
0030 0000 0000 0000 0000 0000 0000 0000 0000 7473 0000 0000 0000 0000 0000 0000 0000
0040 0000 0000 0000 6672 0000 0000 0000 0000 0000 0000 0000 0000 0000 0015 0000 0000
0050 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0060 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0070 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

Register File before any instruction implementation:

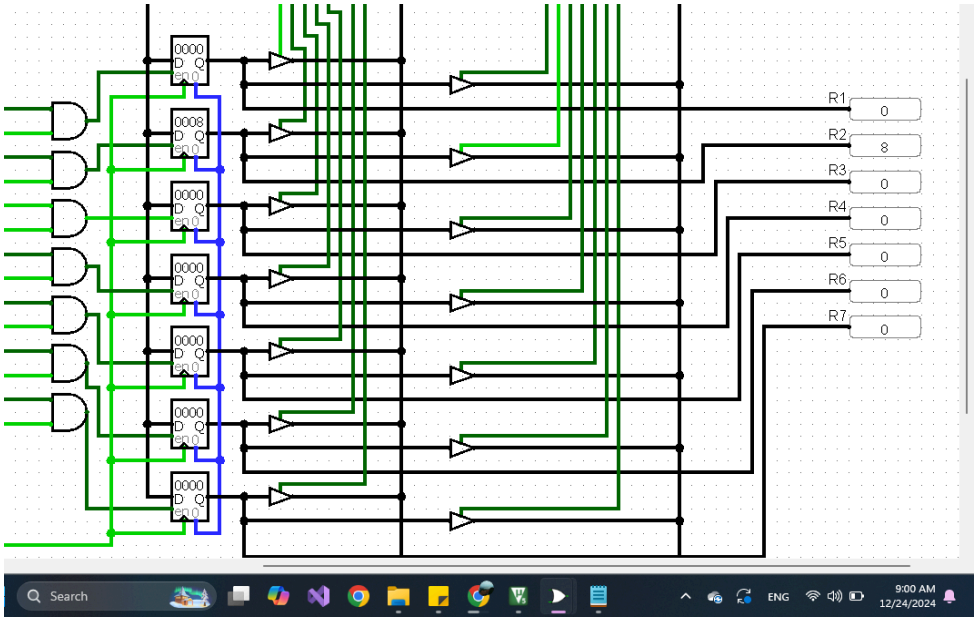


>> All registers are set to zero

Instruction #1: LW R2, R1, #0x00001 → R2 = Mem[R1+0x00001] = 8

This instruction adds the value in R1 to the immediate value #0x00001, and uses the result as a memory address. Then, it loads the value in that memory address into R2.

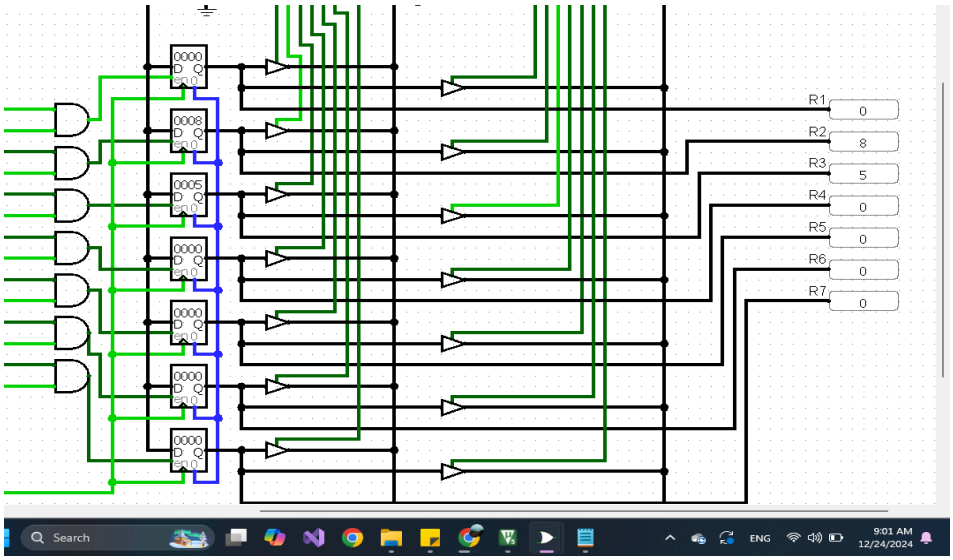
Register File:



Instruction #2: LW R3, R1, #0x00002 → R2 = Mem[R1+0x00002] = 5

This instruction adds the value in R1 to the immediate value #0x00002, and uses the result as a memory address. Then, it loads the value in that memory address into R3.

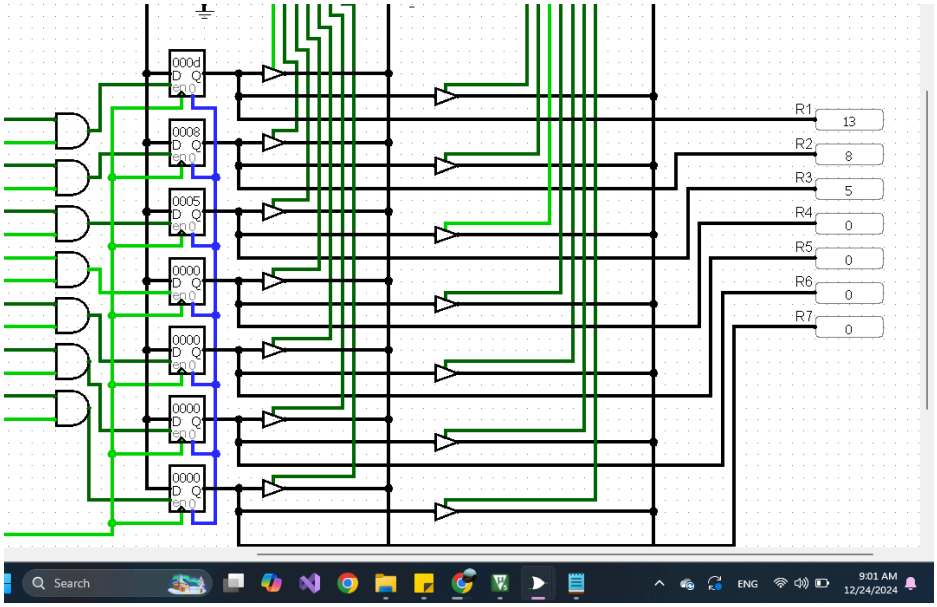
Register File:



Instruction #3: XOR R1, R2, R3 → R1 = R2^R3 = 1000^0101 = 1101b = 0xd

This instruction performs a bitwise XOR operation on the values in R2 and R3, then it stores the result in R1.

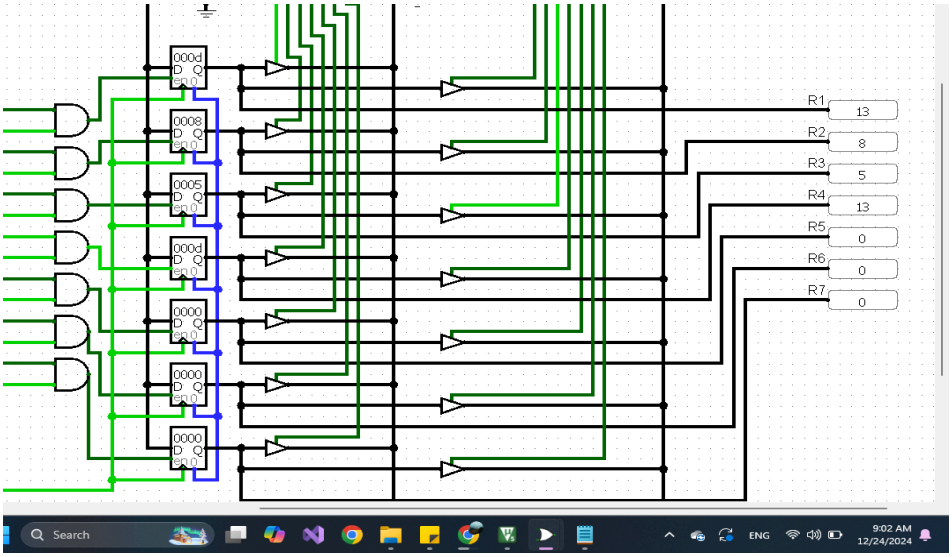
Register File:



Instruction #4: OR R4, R1, R3 → R4 = R1 | R3 = 1101b = 0xd

This instruction performs a bitwise OR operation on the values in R1 and R3, then it stores the result in R4.

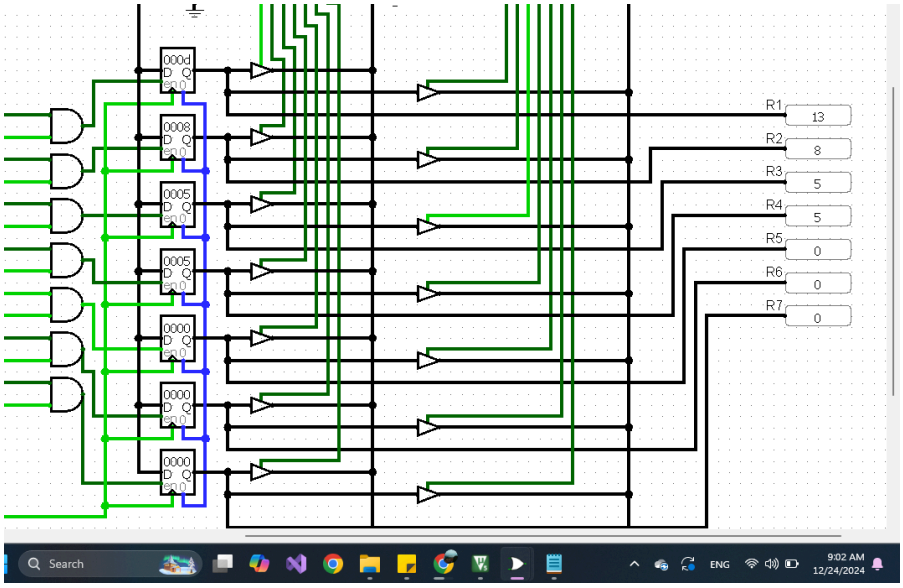
Register File:



Instruction #5: AND R4, R1, R3 → R4 = R1 & R3 = 0101b = 0x5

This instruction performs a bitwise AND operation on the values in R1 and R3, then it stores the result in R4.

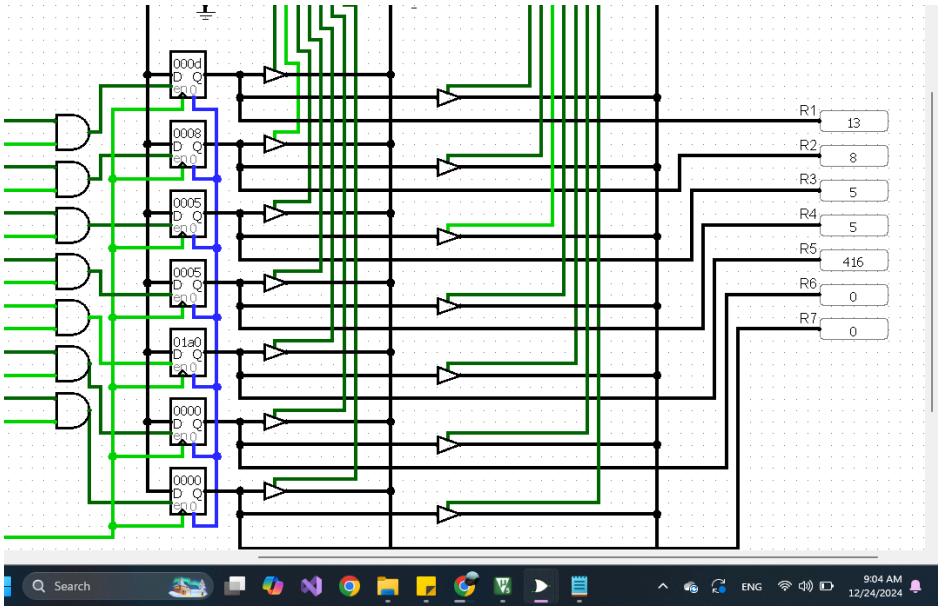
Register File:



Instruction #6: SLL R5,R1,R3

This instruction uses the first four bits of R3 to determine the shift amount of the value in R1. It shifts the value in R1 to the left and fills the empty spots with zeros (logical shift), then it stores the result in R5.

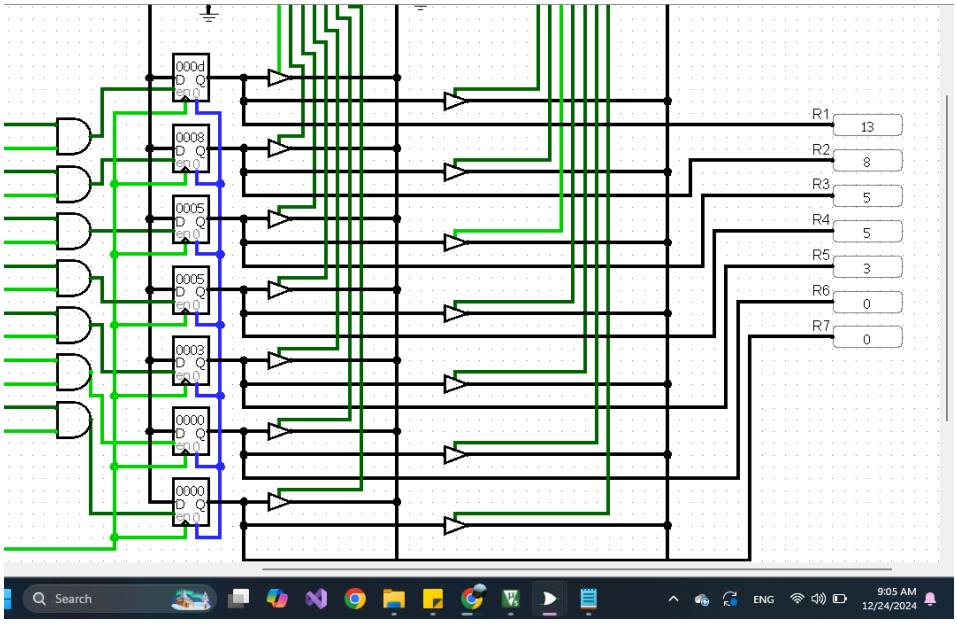
Register File:



Instruction #7: SUB R5,R2,R3

This instruction subtracts the value in R3 from the value in R2, and stores the result in R5.

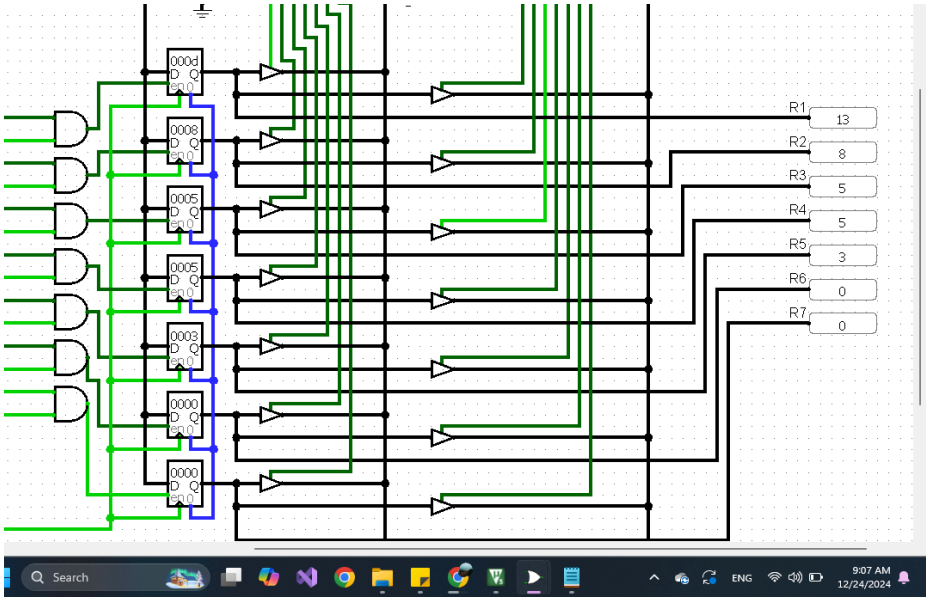
Register File:



Instruction #8: SRL R6,R1,R3

This instruction uses the first four bits of R3 to determine the shift amount of the value in R1. It shifts the value in R1 to the right and fills the empty spots with zeros (logical shift), then it stores the result in R6.

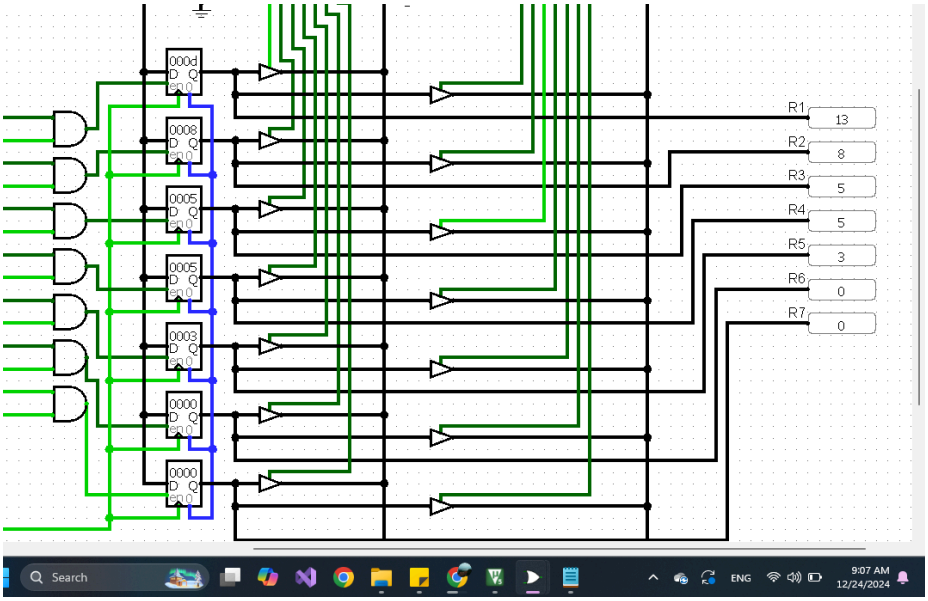
Register File:



Instruction #9: SRA R7,R1,R3

This instruction uses the first four bits of R3 to determine the shift amount of the value in R1. It shifts the value in R1 to the right and fills the empty spots with the sign (arithmetic shift), then it stores the result in R7.

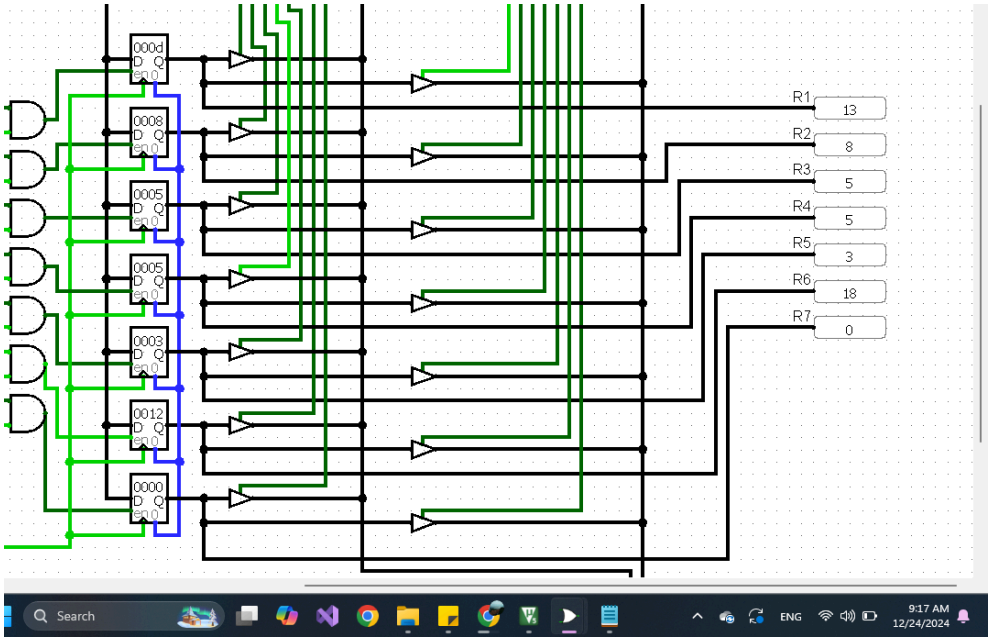
Register File:



Instruction #10: ADD R6,R4,R1

This instruction adds the values in R1 and R4, then stores the result in R6.

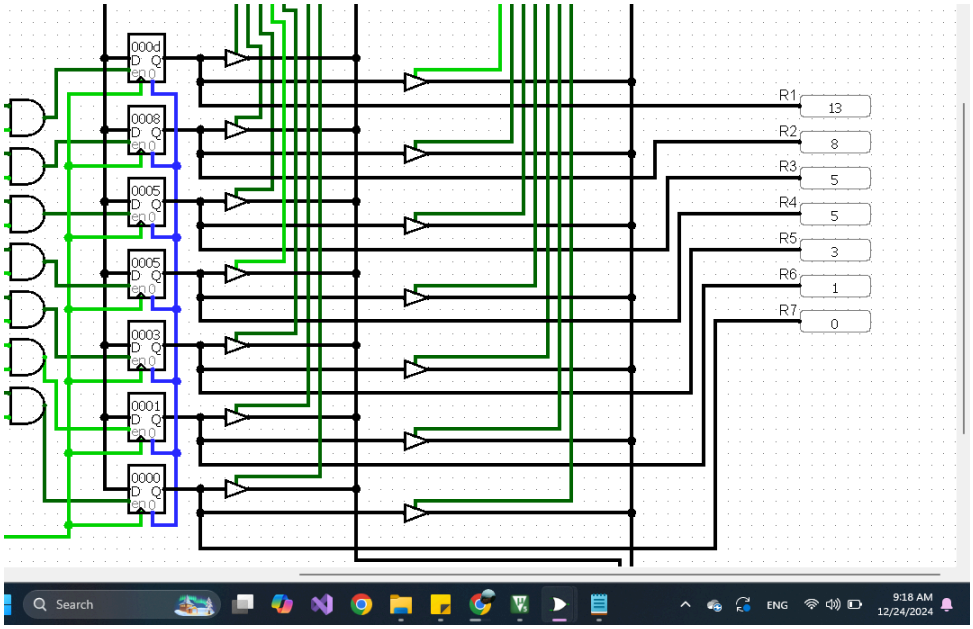
Register File:



Instruction #11: SLT R6,R4,R1

This instruction compares the signed values in R1 and R4. If the value in R4 is less than the value in R1, it sets the value in R6 to 1.

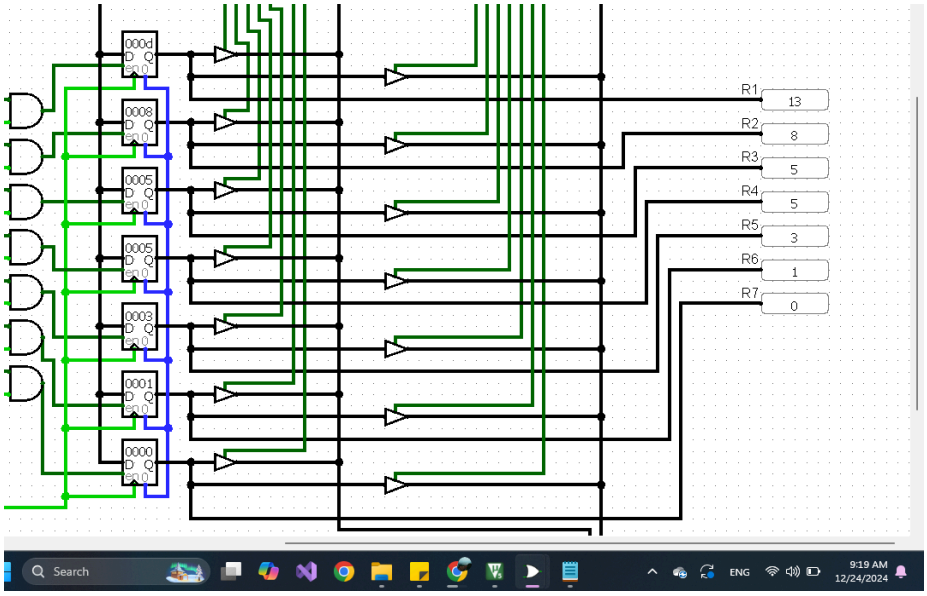
Register File:



Instruction #12: SLTU R6,R4,R1

This instruction compares the unsigned values in R1 and R4. If the value in R4 is less than the value in R1, it sets the value in R6 to 1.

Register File:



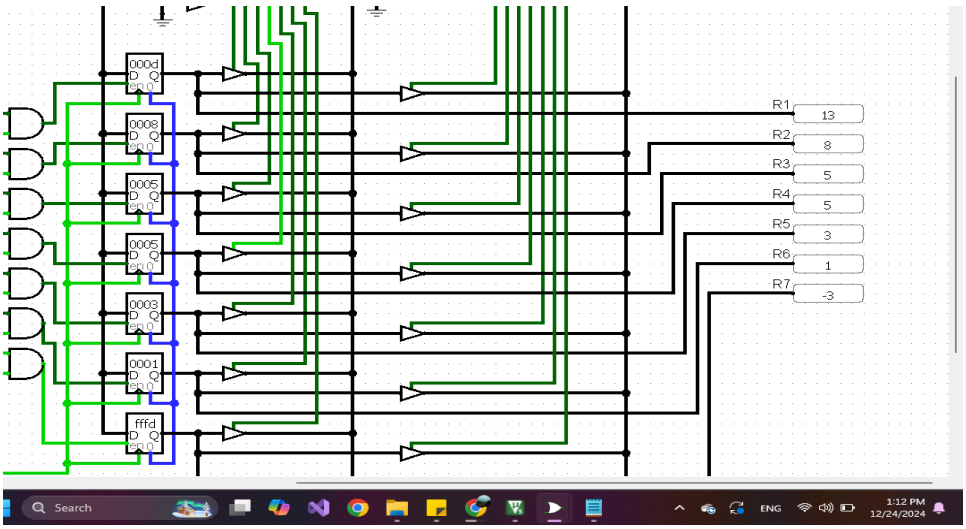
Instruction Set 2 (I-type instructions):

#	Address of instruction	Assembly	Binary	Hexadecimal
13	0x000C	ADDI R7, R6, #11100b	1110/0 110 /111 0/0011	0xe6e3
14	0x000D	SLTI R7, R4, #11000b	1100/0 100 /111 0/0100	0xc4e4
15	0x000E	SLTIU R7, R4, #11000b	1100/0 100 /111 0/0101	0xc4e5
16	0x000F	XORI R7, R4, #00111b	0011/1 100 /111 0/0110	0x3ce6
17	0x0010	ORI R5, R4, #00111b	0011/1 100 /111 0/0111	0x3ce7
18	0x0011	ANDI R5, R4, #00111b	0011/1 100 /111 0/1000	0x3ce8
19	0x0012	ADDI R4, R4, #01100b	0110/0 100 /100 0/0011	0x6483
20	0x0013	SLLI R6, R4, #011b	0001/1 100 /110 0/1001	0x1cc9
21	0x0014	SRLI R6, R4, #011b	0001/1 100 /110 0/1010	0x1cca
22	0x0015	SRAI R6, R4, #011b	0001/1 100 /110 0/1011	0x1ccb

Instruction #13: ADDI R7, R6, #11100b

This instruction adds the value in R6 to the immediate value #11100 (after sign_extending it), then it stores the result in R7.

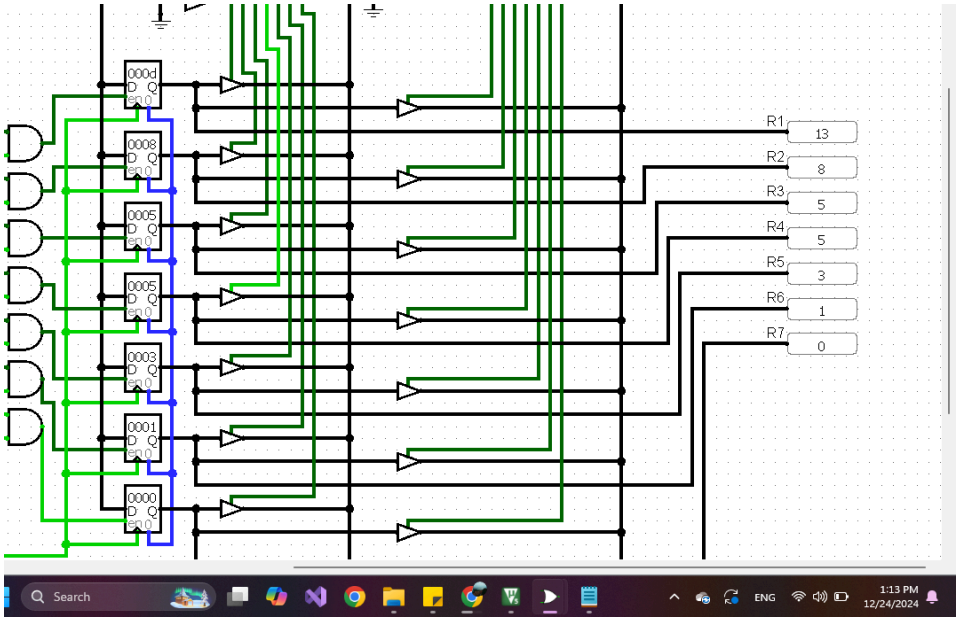
Register File:



Instruction #14: SLTI R7, R4, #11000b

This instruction compares the signed value in R4 to the immediate value #11000. If the immediate value is greater it sets the value in R7 to 1.

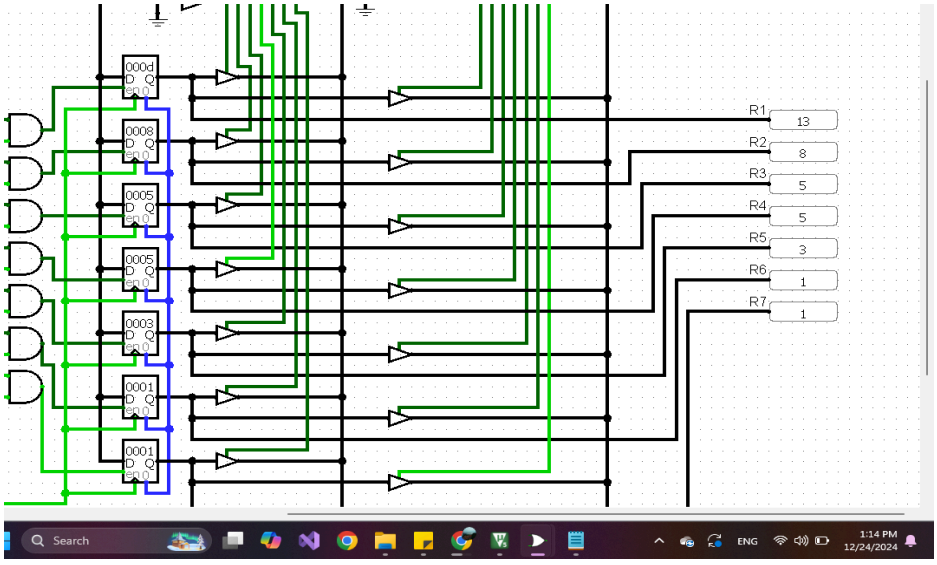
Register File:



Instruction #15: SLTI R7, R4, #11000b

This instruction compares the unsigned value in R4 to the immediate value #11000. If the immediate value is greater it sets the value in R7 to 1.

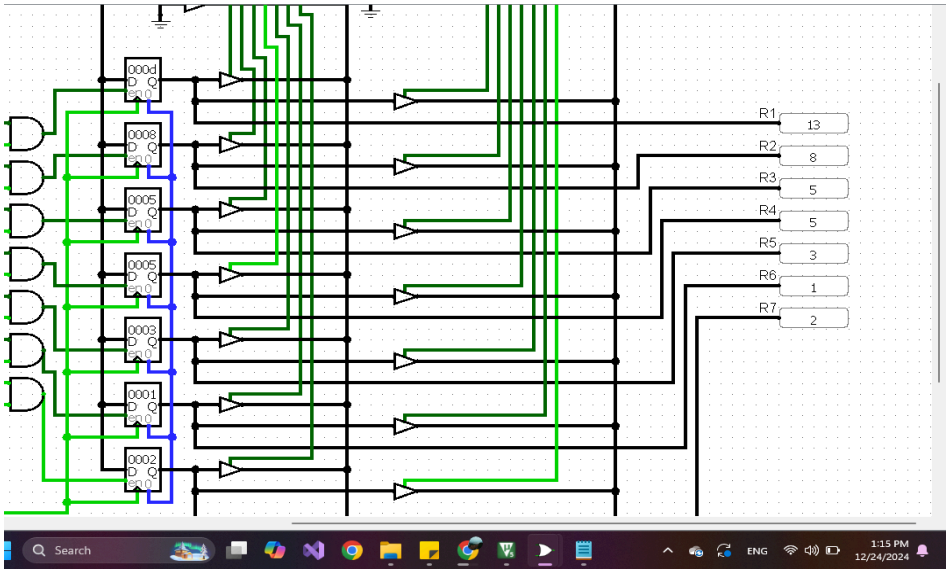
Register File:



Instruction #16: XORI R7, R4, #00111b

This instruction performs a bitwise XOR operation between the immediate value 00111 and the value in R4, then it stores the result in R7.

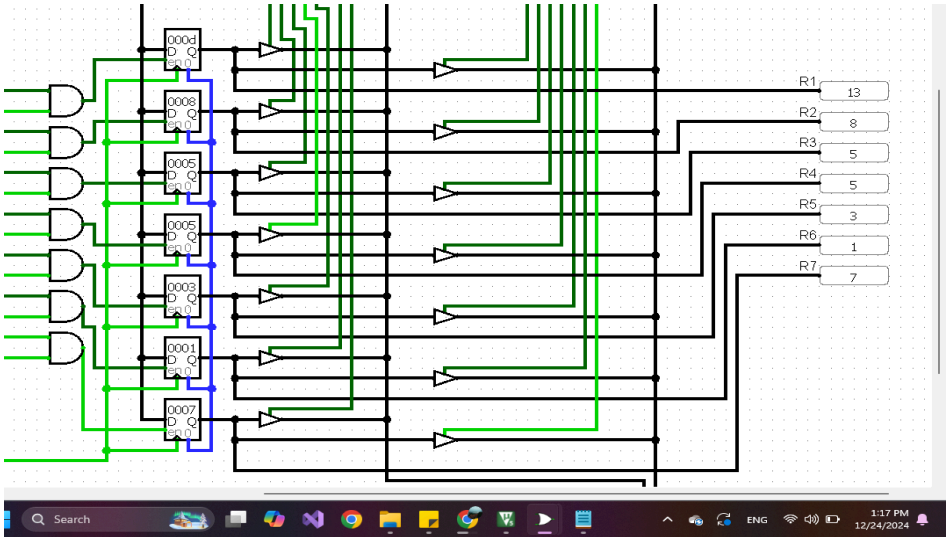
Register File:



Instruction #17: ORI R5, R4, #00111b

This instruction performs a bitwise OR operation between the immediate value 00111 and the value in R4, then it stores the result in R5.

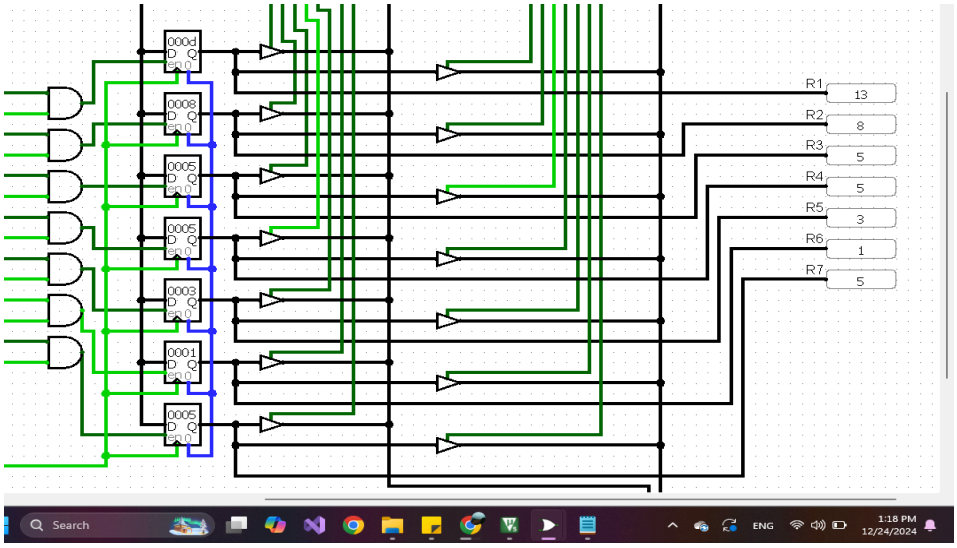
Register File:



Instruction #18: ANDI R5, R4, #00111b

This instruction performs a bitwise AND operation between the immediate value 00111 and the value in R4, then it stores the result in R5.

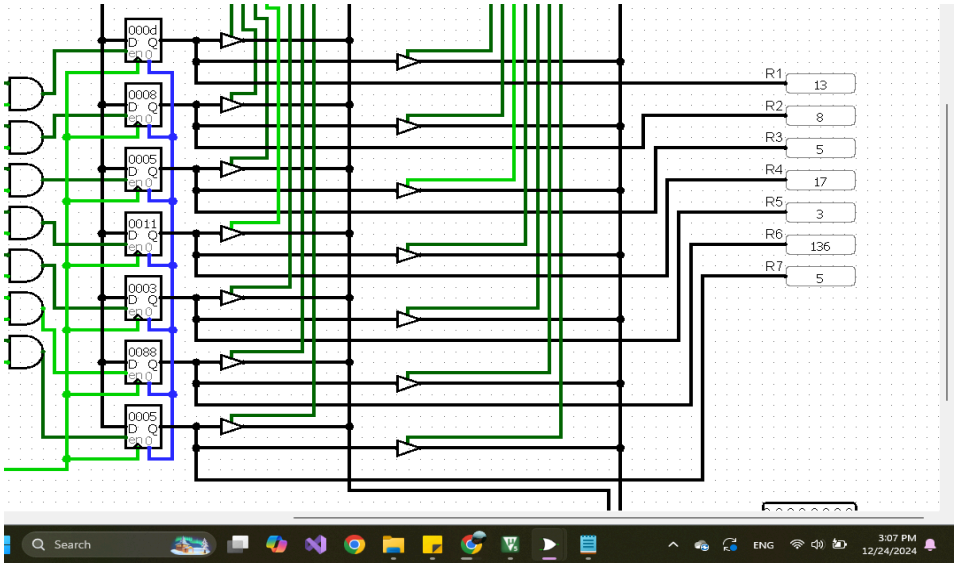
Register File:



Instruction #19: ADDI R4, R4, #01100b

This instruction adds the value in R4 to the immediate value 01100, then stores the result in R4.

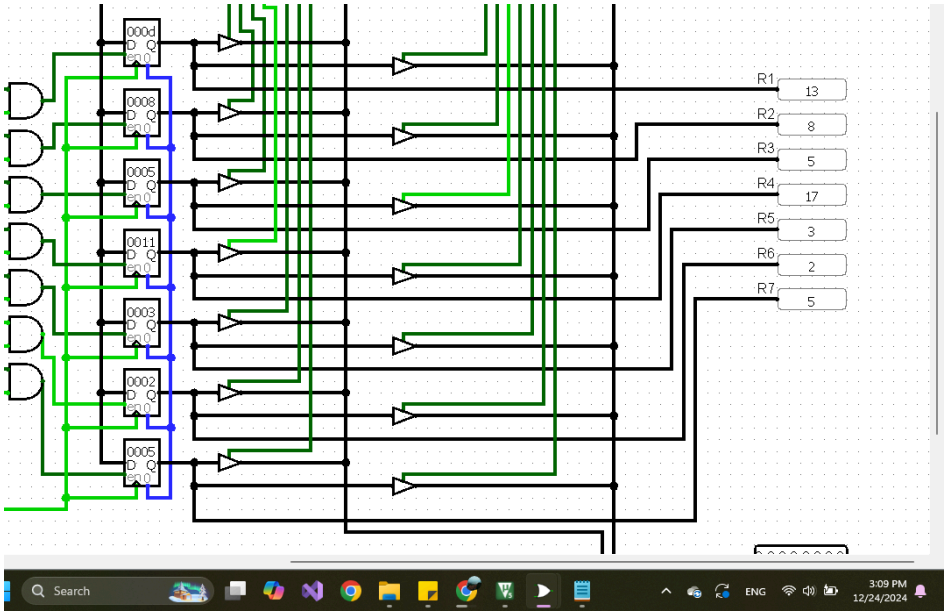
Register File:



Instruction #20: SLLI R6, R4, #011b

This instruction shifts the value in R4 to the left by using the immediate value 011 to specify the shift amount, then it stores the result in R6 after filling the empty spots with zeros (logical shift).

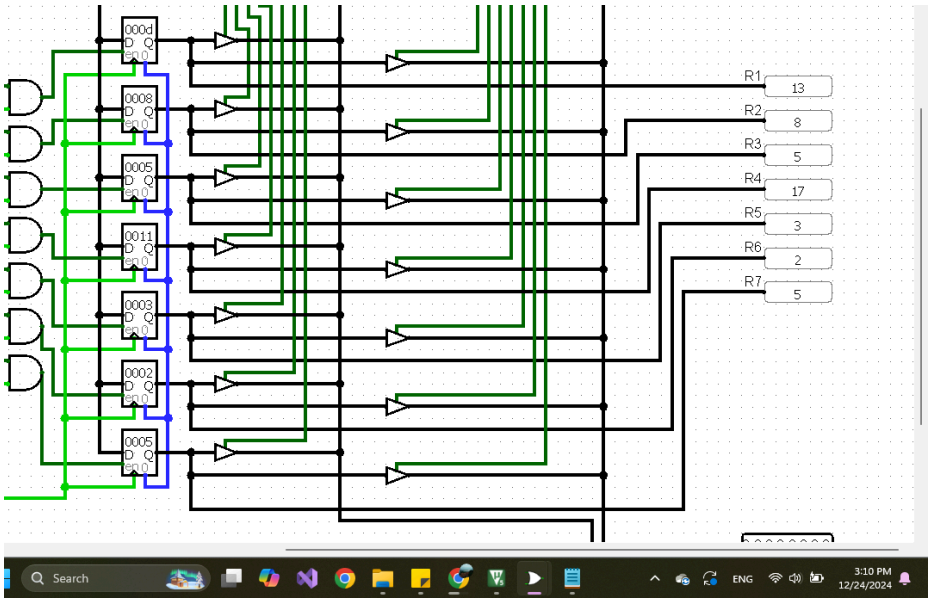
Register File:



Instruction #21: SRLI R6, R4, #011b

This instruction shifts the value in R4 to the right by using the immediate value 011 to specify the shift amount, then it stores the result in R6 after filling the empty spots with zeros (logical shift).

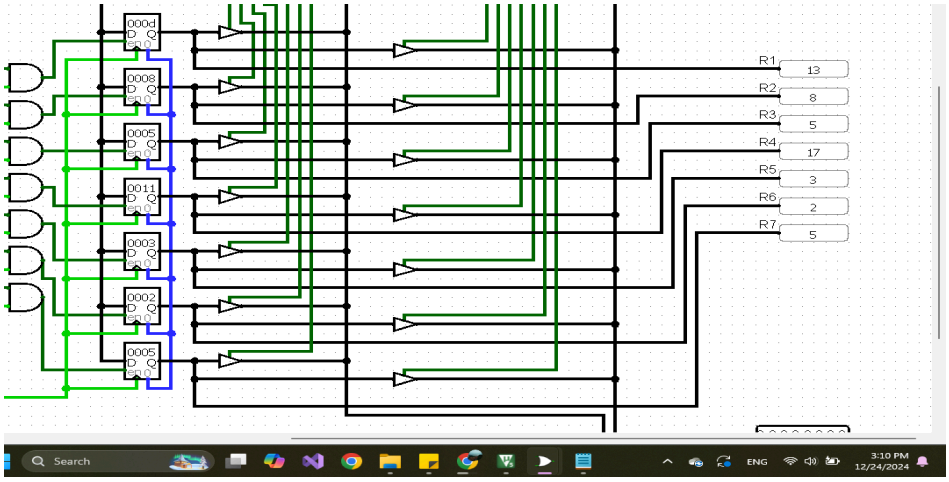
Register File:



Instruction #22: SRAI R6, R4, #011b

This instruction shifts the value in R4 to the right by using the immediate value 011 to specify the shift amount, then it stores the result in R6 after filling the empty spots with the sign (arithmetic shift).

Register File:



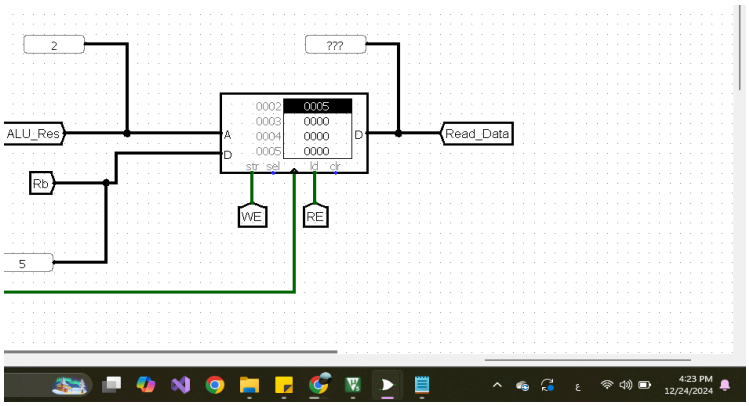
Instruction Set 3 (SB-type instructions):

#	Address of instruction	Assembly	Binary	Hexadecimal
23	0x0016	SW #011b, R5, R6, #01b	01 11/0 101 /011 0/1101	0x756d
24	0x0017	BEQ #011, R3, R7, #01	01 11/1 011 /011 0/1110	0x7b6e
25	0x0018	BNE #011, R3, R4, #01	01 10/0 011 /011 0/1111	0x636f
26	0x0023	BLT #011, R6, R4, #01	01 10/0 110 /011 1/0000	0x6670
27	0x002e	BGE #011, R4, R6, #01	01 11/0 100 /011 1/0001	0x7471
28	0x0039	BLTU #011, R6, R4, #01	01 10/0 110 /011 1/0010	0x6672
29	0x0044	BGEU #011, R4, R6, #01	01 11/0 100 /011 1/0011	0x7473

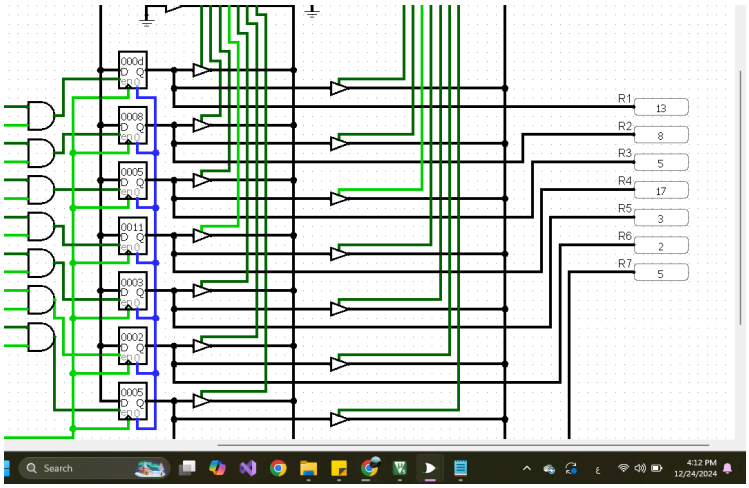
Instruction #23: SW #011b, R5, R6, #01b

This instruction concatenates the immediate values {imm2U,imm3L}, then it adds this value to R6 to produce the memory address where the value in R5 will be stored.

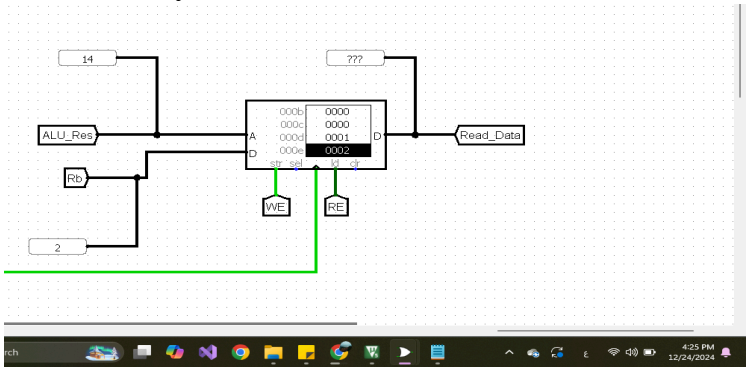
Data Memory Before:



Register File:



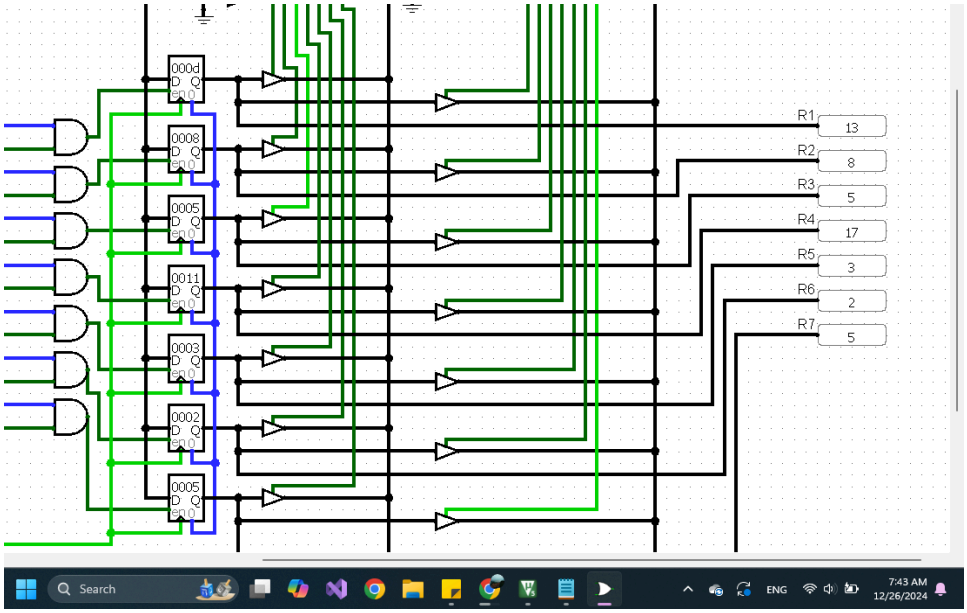
Data Memory After:



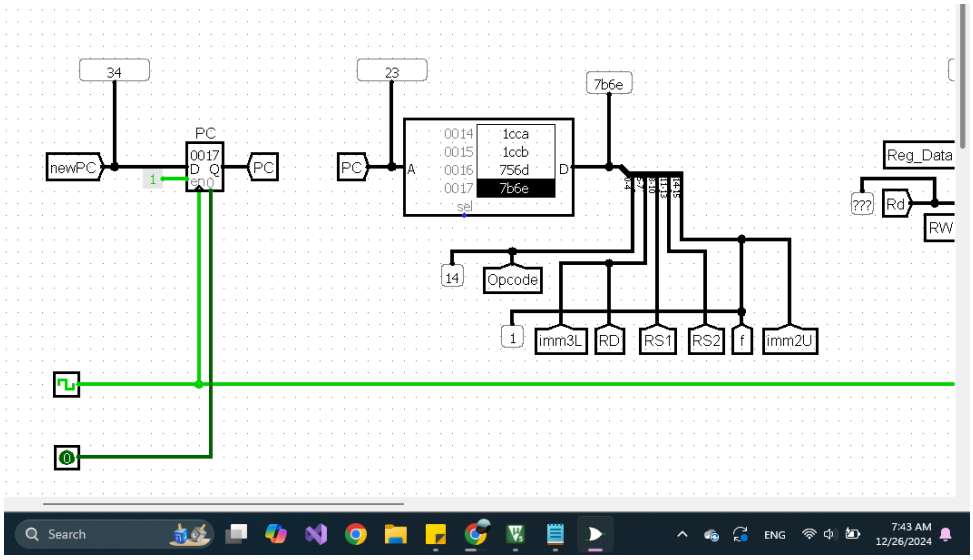
Instruction #24: BEQ #011, R3, R7, #01

This instruction compares the values in R3 and R7. If the values are equal, it branches to a new PC, which equals PC+sign_extend({imm2U,imm3L}) = 23+11=34.

Register File:



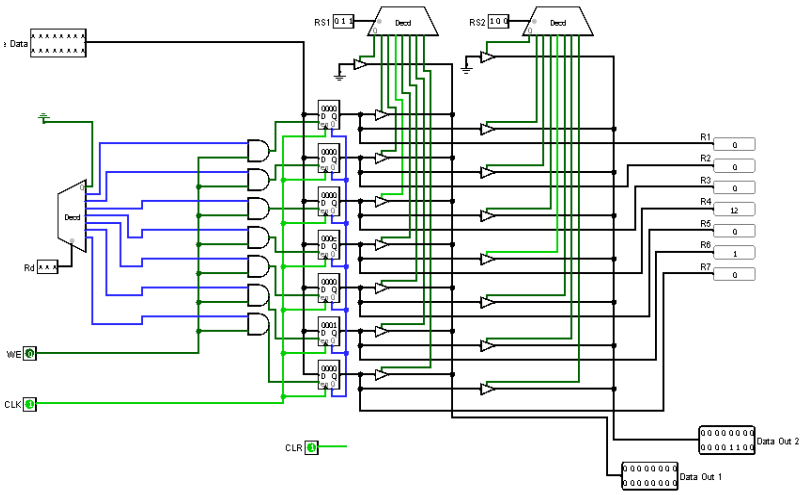
Datapath:



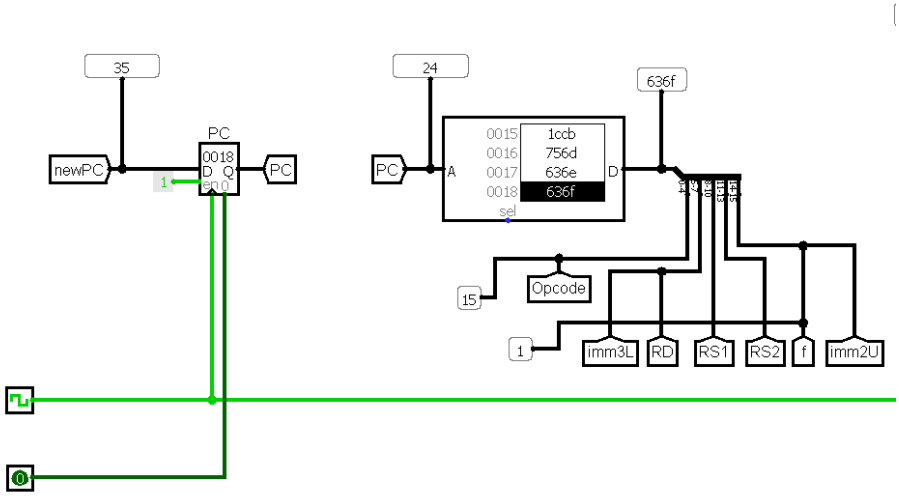
Instruction #25: BNE #011, R3, R4, #01

This instruction checks if the values in registers R4 and R3 are not equal, if the condition is met, then the processor branches to a new instruction, whose address is PC+sign_extend({imm2U,imm3L}) = 24+11=35.

Register File:



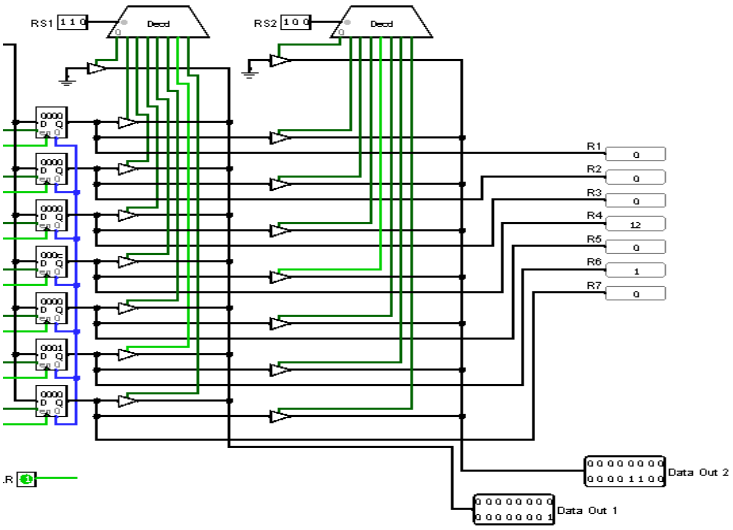
Datapath:



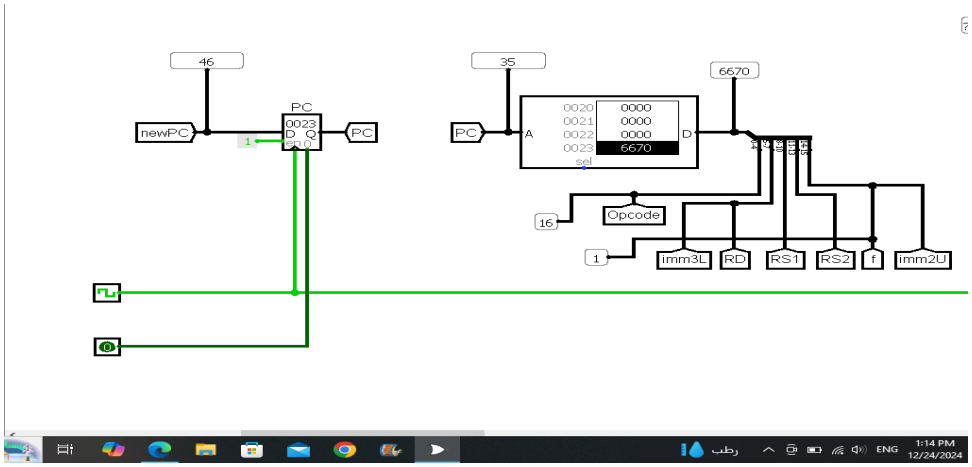
Instruction #26: BLT #011, R6, R4, #01

This instruction checks if the signed value in R6 is greater than the signed value in R4, if the condition is met, then the processor branches to a new instruction, whose address is PC+sign_extend({imm2U,imm3L}) = 35+11=46.

Register File:



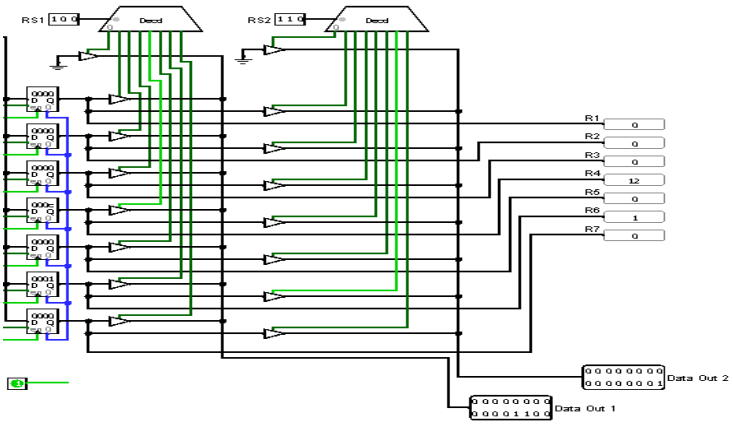
Datapath:



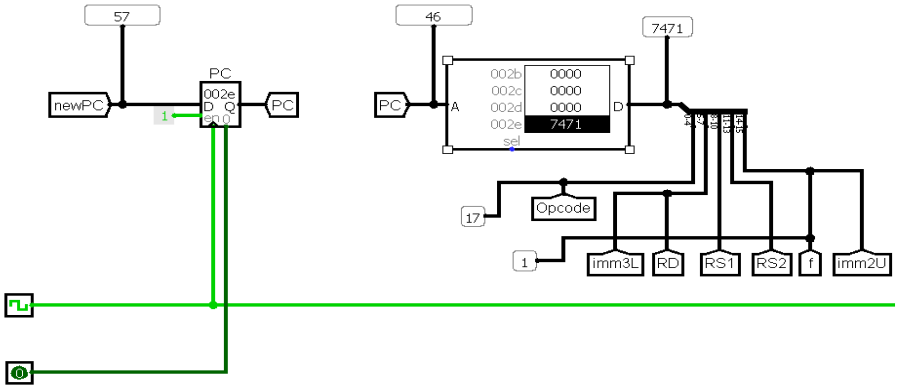
Instruction #27: BGE #011, R4, R6, #01

This instruction checks if the value in R4 is greater than or equal to the value in R6, if the condition is met, then the processor branches to a new instruction, whose address is PC+sign_extend({imm2U,imm3L}) = 46+11=57.

Register File:



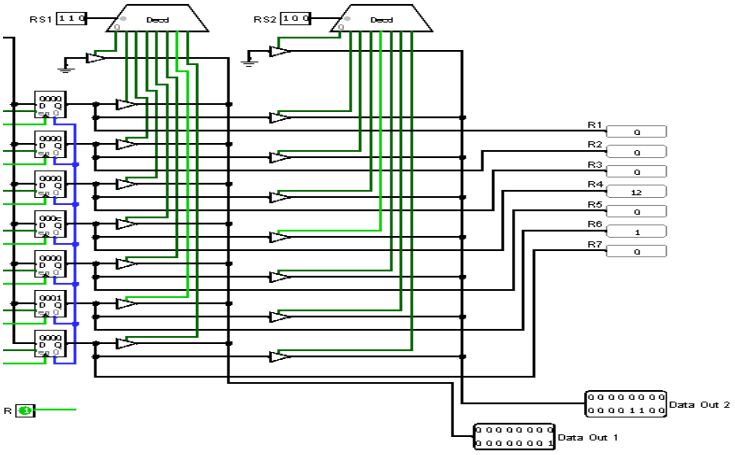
Datapath:

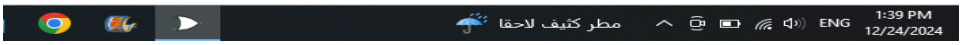


Instruction #28: BLTU #011, R6, R4, #01

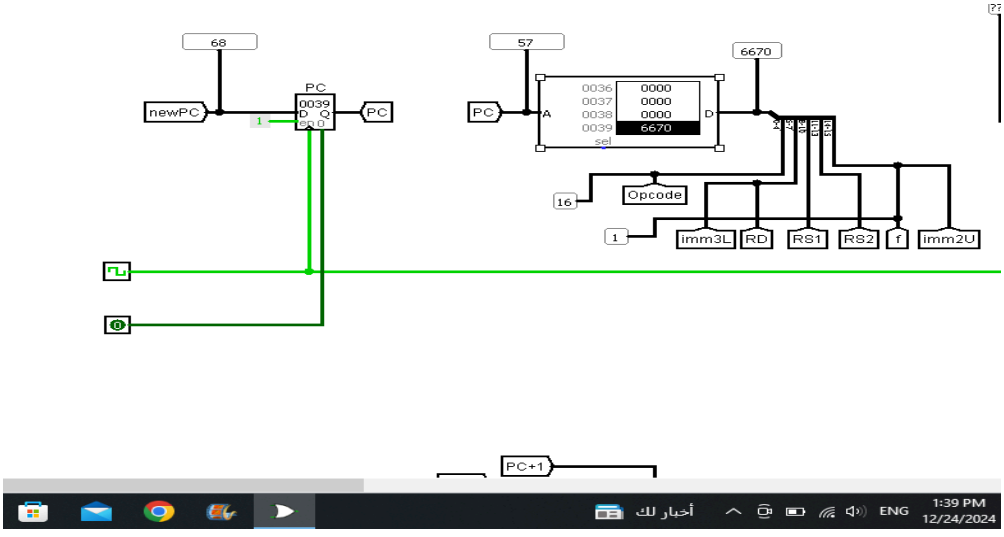
This instruction checks if the unsigned value in R6 is greater than the unsigned value in R4, if the condition is met, then the processor branches to a new instruction, whose address is PC+sign_extend({imm2U,imm3L}) = 57+11=68.

Register File:





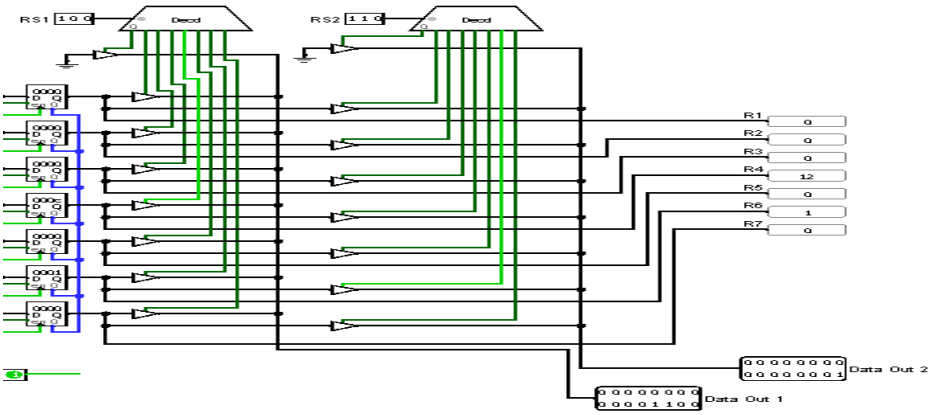
Datapath:



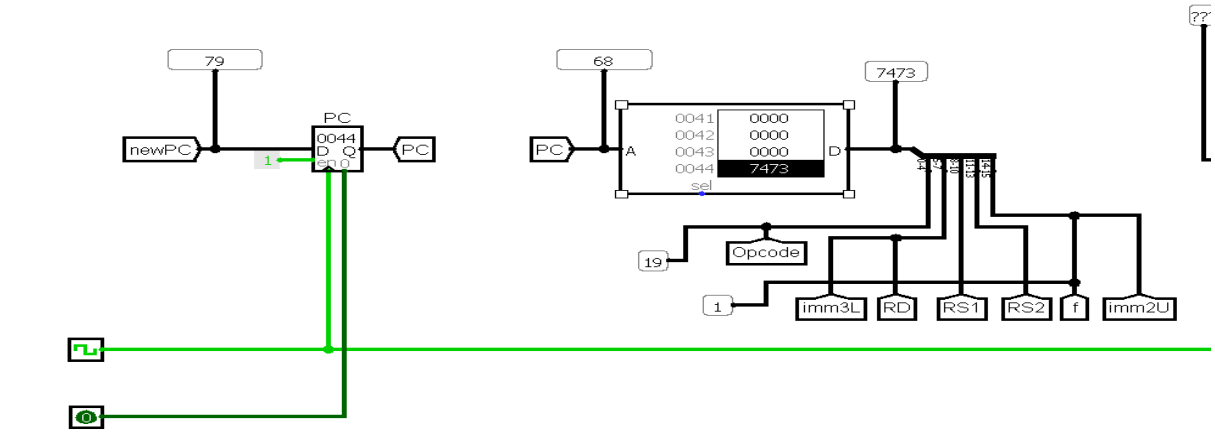
Instruction #29: BGEU #011, R4, R6, #01

This instruction checks if the unsigned value in R4 is greater than or equal to the unsigned value in R6, if the condition is met, then the processor branches to a new instruction, whose address is PC+sign_extend({imm2U,imm3L}) = 68+11=79.

Register File:



Datapath:



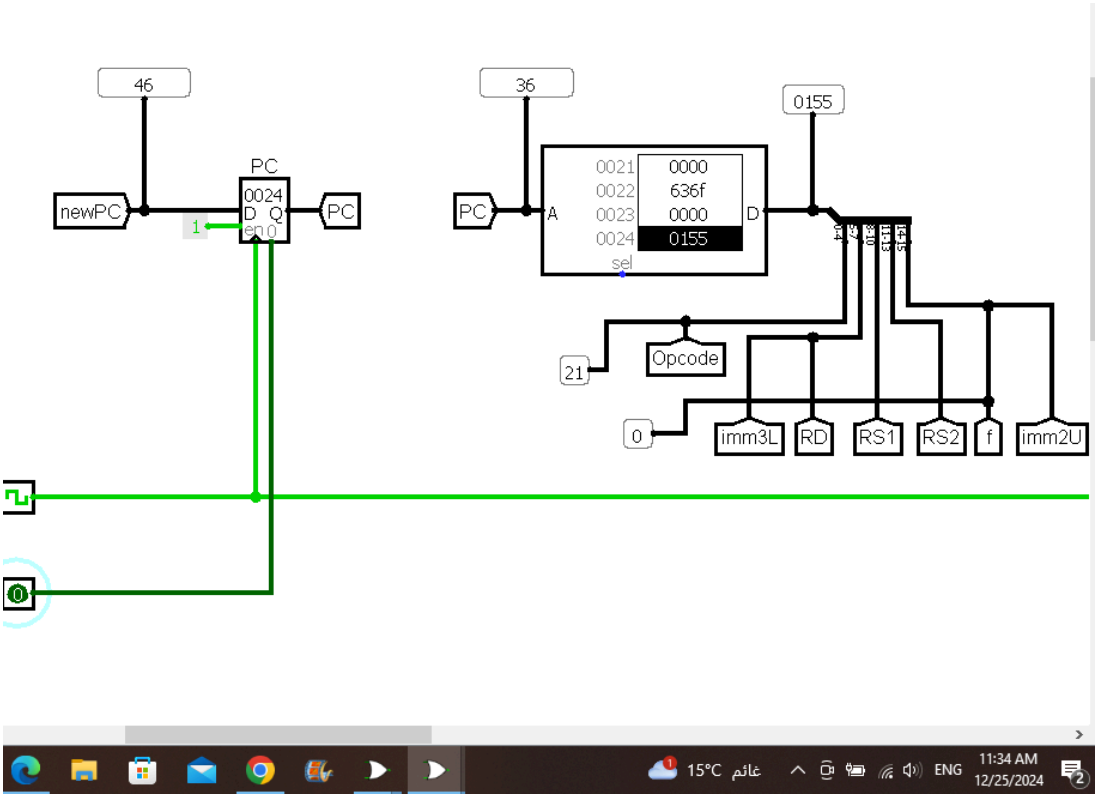
Instruction Set 4 (J-type instructions):

#	Address of instruction	Assembly	Binary	Hexadecimal
30	0x0024	J #00000001010	0000/0001/010 1/0101	0x0155
31	0x002E	JAL #00000001100	0000/0001 /100 1/0101	0x0195
32	0x003A	JALR R1,R6,#00001	0000/1 110 /001 1/0111	0x0E37
33	0x0000	LUI #00000001111	0000/ 0001 /111 1/0100	0x01F4

Instruction #30: J #00000001010

This instruction adds the immediate value #00000001010 to the current PC value to produce the new PC where the program jumps. The current PC is 36, and after adding the immediate value 10 to it, it becomes 46.

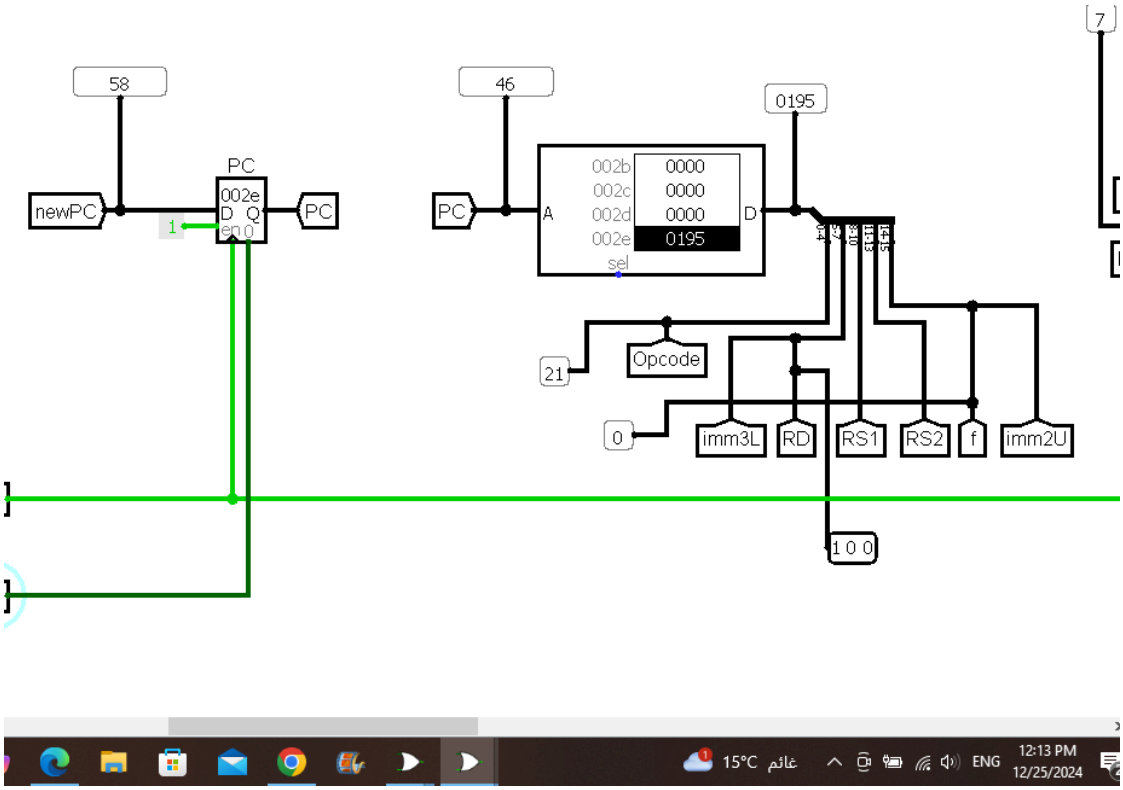
Datapath:



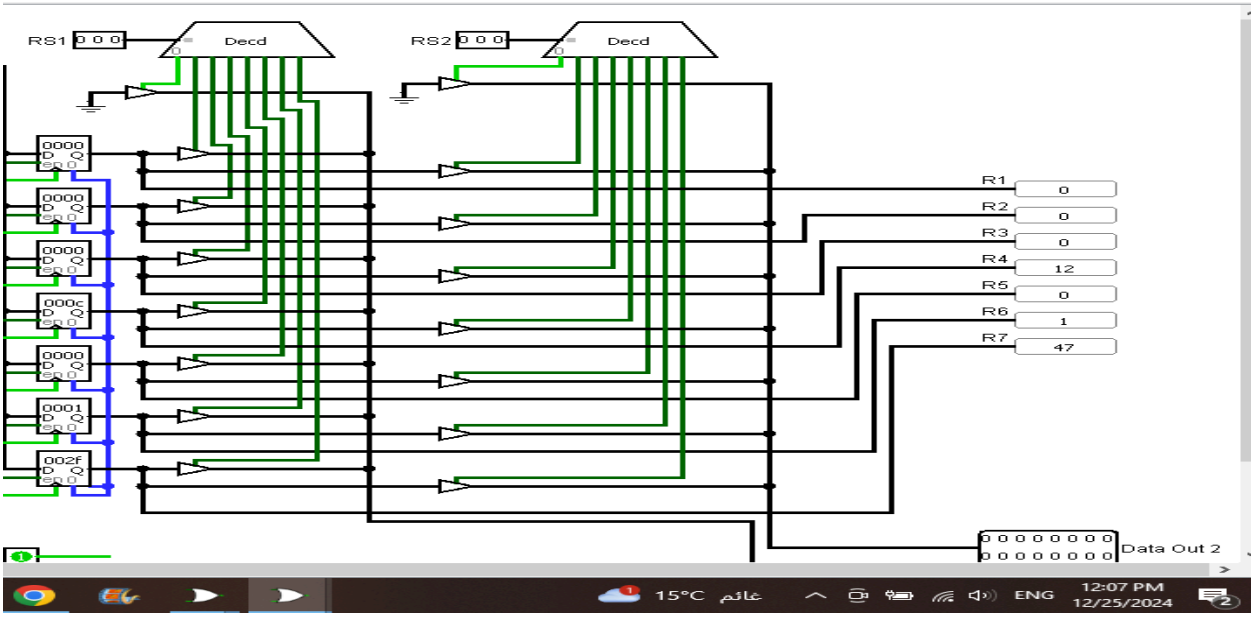
Instruction #31: JAL #00000001100

This instruction adds the immediate value #00000001100 to the current PC value to produce the new PC which the program jumps to. The current PC is 46, and after adding the immediate value 12 to it, it becomes 58. The program stores the next instruction address (PC+1) in R7.

Datapath:



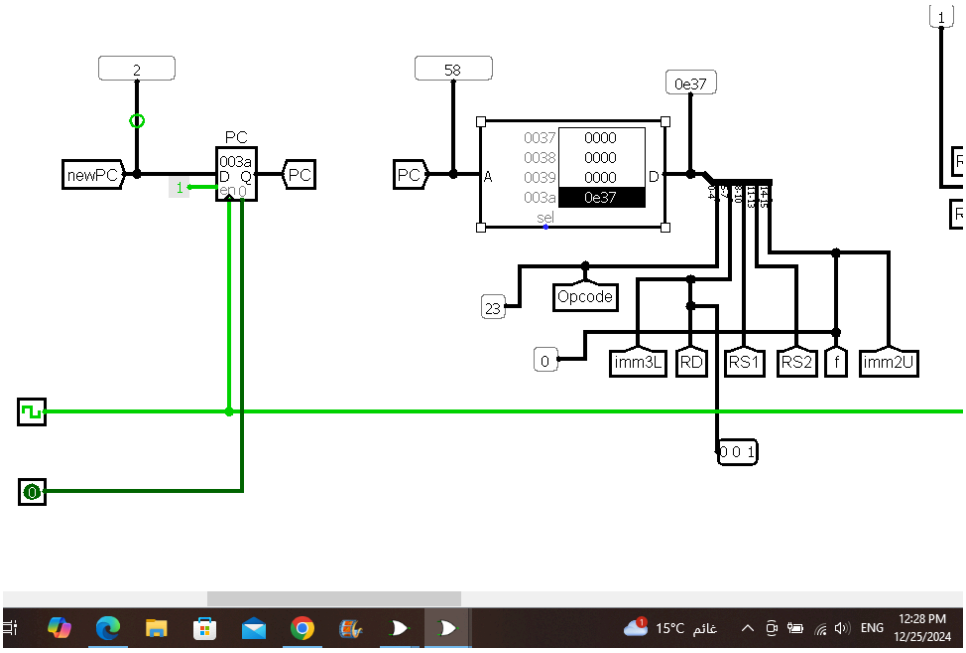
Register File:



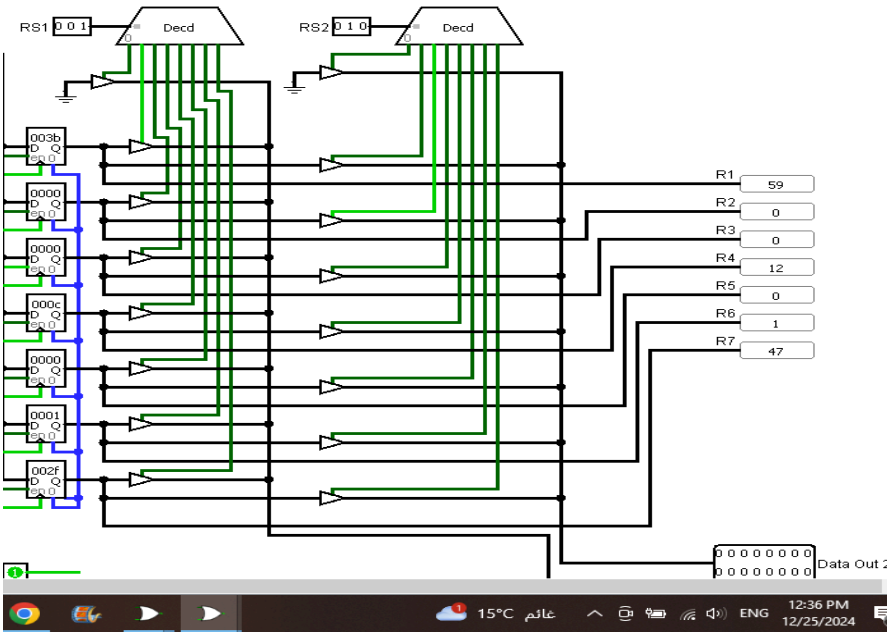
Instruction #32: JALR R1,R6,#00001

This instruction adds the immediate value #00001 to the current value in R6 (which is 1) to produce the new PC where the program jumps (in this case it returns to address 2). The program stores the next instruction address (PC+1) in R1.

Datapath:



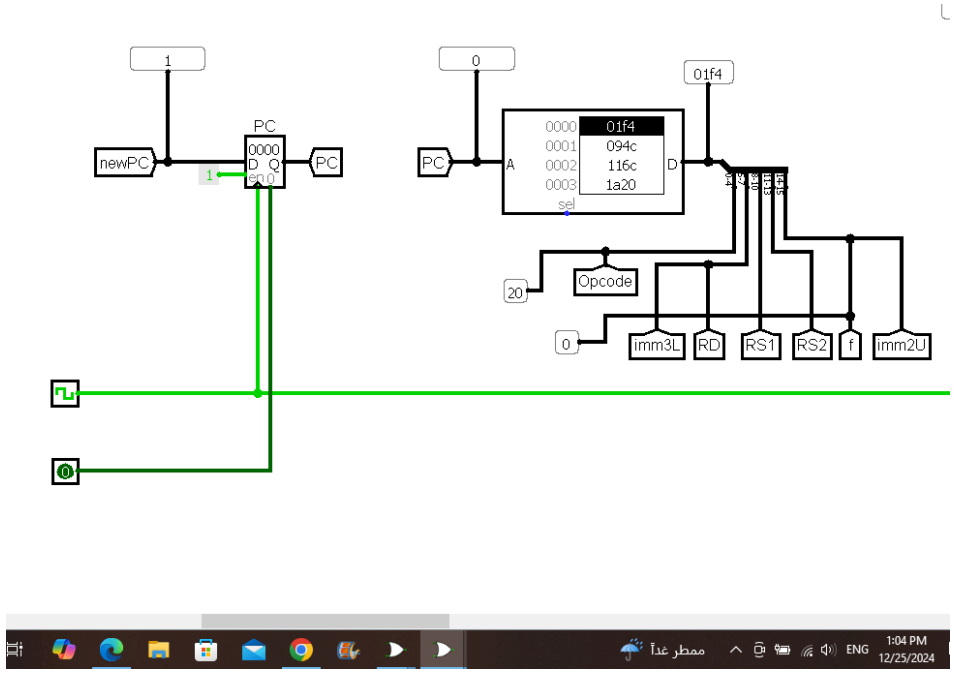
Register File:



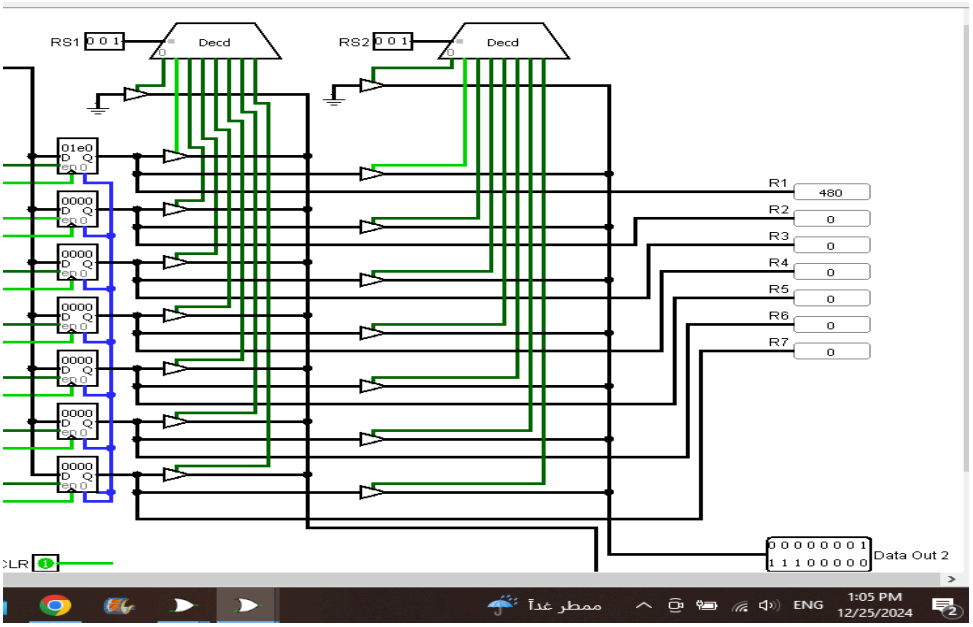
Instruction #33: LUI #00000001111

This instruction loads the first five bits of the immediate value into the upper 5 bits of register 1, then it fills the empty spots with zeros.

Datapath:



Register File:



Testing Scenario 1 (Summation of array elements):

The following program adds 5 values in an array in the data memory, and then stores the result back in memory. During each loop, it checks if it has reached the end of the array, if yes, it branches to the address where the memory STORE instruction is located, if no, it jumps back to the beginning of the loop to keep on adding the values to the sum.

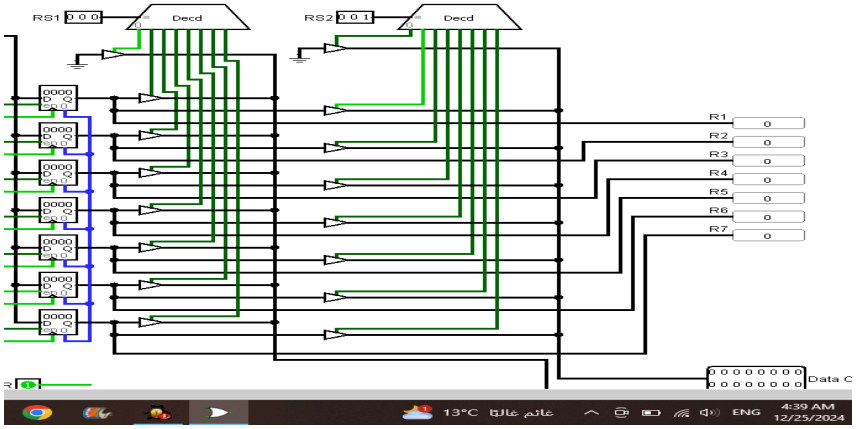
#	Instruction Address	Instruction Format	Binary	HEX	Instruction Description
1	0X0010	ADDI R6,R0,#01	0000100011000011	0X08C3	Initialize R6 to 1 (address of the first element in memory)
2	0X0011	ADDI R3,R0,#00	0000000001100011	0X0063	Initialize the sum (R3) to 0
3	0X0012	ADDI R7,R0,#00	0000000011100011	0X00E3	Initialize the counter R7 to 0
4	0X0013	ADDI R4,R0,#101	0010100010000011	0X2883	Initialize R4 to 5 (elements number)
5	0X0014	ADDI R5,R6,#00	0000011010100011	0X06A3	Initialize R5 (base address) to the address of the first element
6	0X0015	LW R2,R6,#00	0000011001001100	0X064C	Load the value in the address stored in R6 (first element) into R2
7	0X0016	ADD R3,R3,R2	0001001101100010	0X1362	Adds the new element stored in R2 to the previous sum in R3
8	0X0017	ADDI R5,R5,#10	0001010110100011	0X15A3	Increment the base address by 2 during each loop
9	0X0018	ADDI R7,R7,#01	0000111111100011	0X0FE3	Increment the elements counter by 1 during each loop
10	0X0019	SLT R1,R7,R4	1010011100100010	0XA722	If the elements count R7 is less than the total elements number R4, set R1 to 1
11	0X001A	BEQ #11,R1,R0,#011	1100000101100010	0XC162	If R1 equals R0 (both are zeros) then branch to the address of the store instruction
12	0X001B	J #11111111011	1111111100110101	0XFF35	Jump to the load instruction in address 0X0015 to loop again

13	0X0022	SW #000,R3,R6,#00	0001111000001101	0X1E0D	Stores the sum in the memory address directly after the last element in the array
14	0X0023	J #00000000000	0000000000010101	0X0015	This instruction terminates the execution of the program by jumping to itself indefinitely

Data Memory:

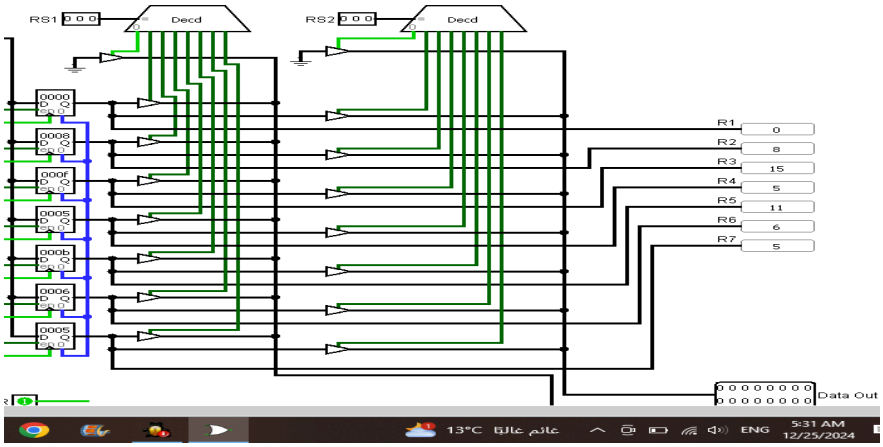
Address	Decimal Value	HEX
0X0001	5	0X0005
0X0002	-3	0XFFFD
0X0003	12	0X000C
0X0004	-7	0XFFF9
0X0005	8	0X0008

Register File Before:



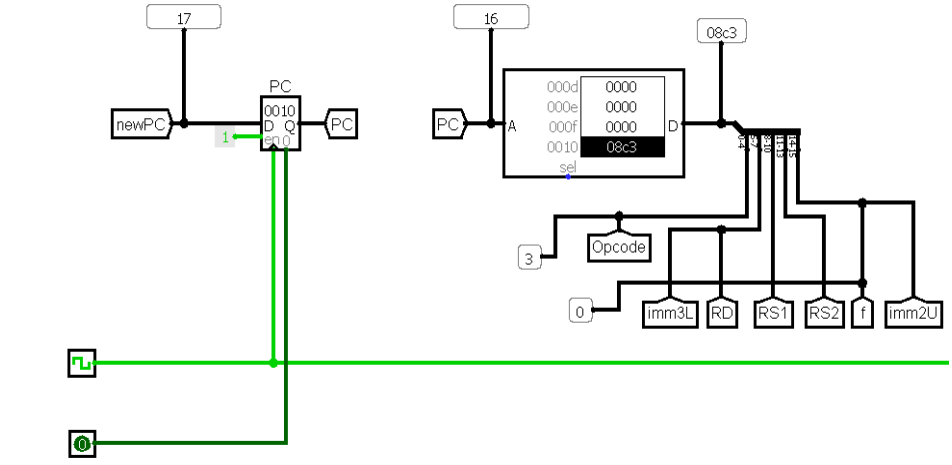
All registers are initialized to 0.

Register File After:

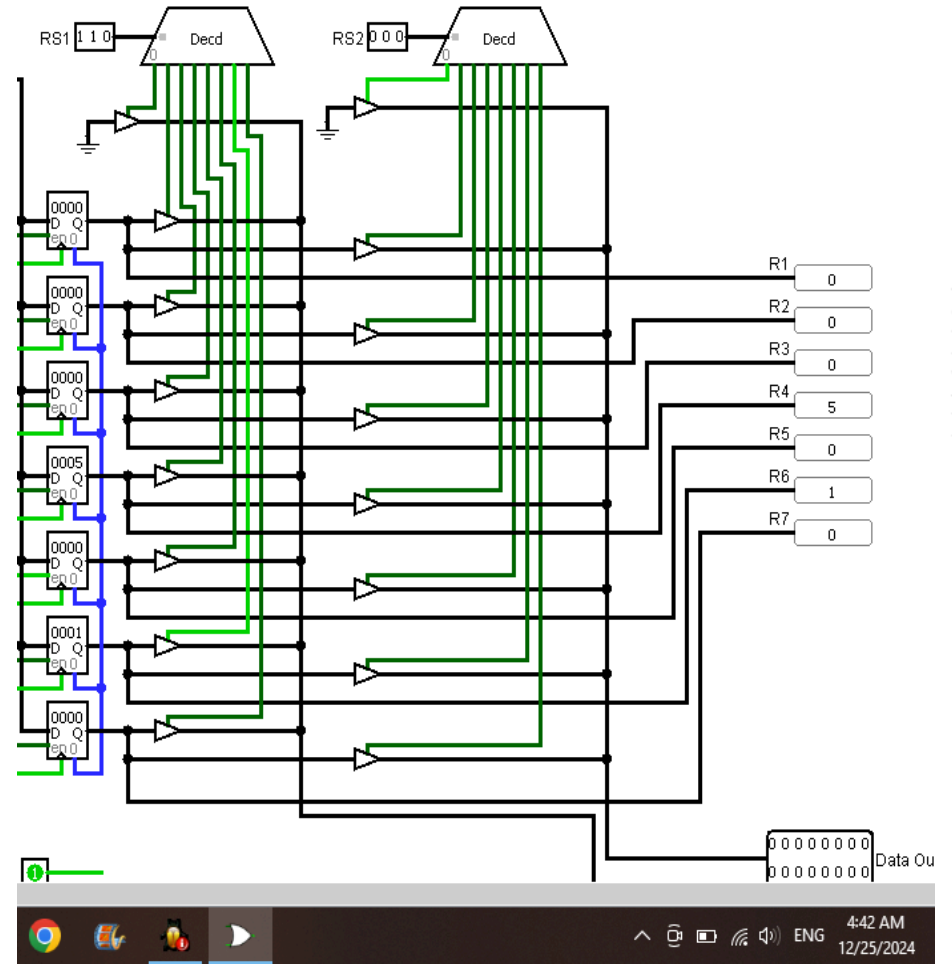


R1=0 → Because R7=R4
R2=8 → final element
R3=15 → final sum
R4= 5 → array size
R5=11 → last element's base address
R6= 6 → element's memory address
R7= 5 → counter

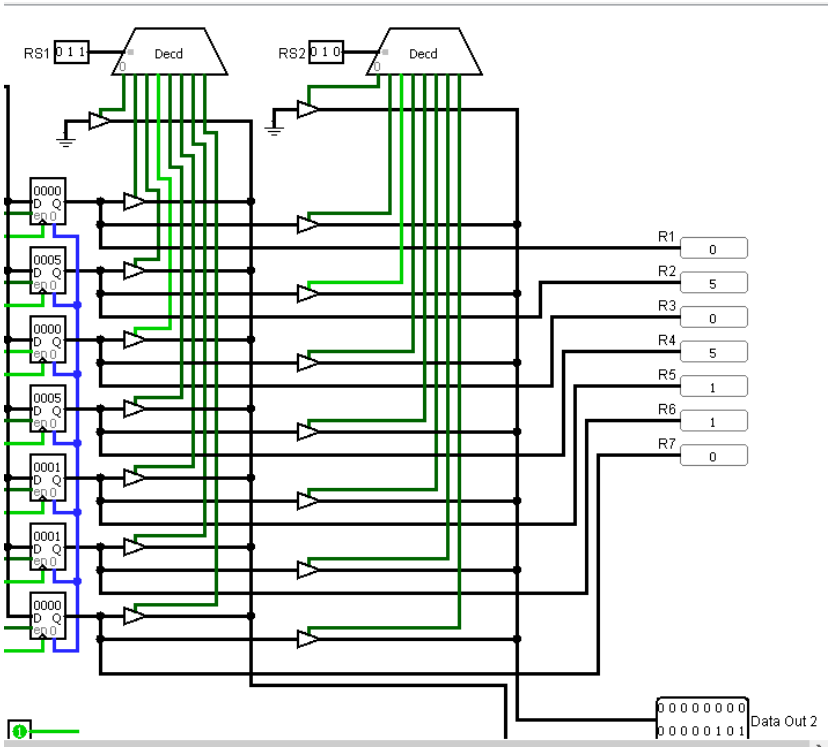
First Loop:



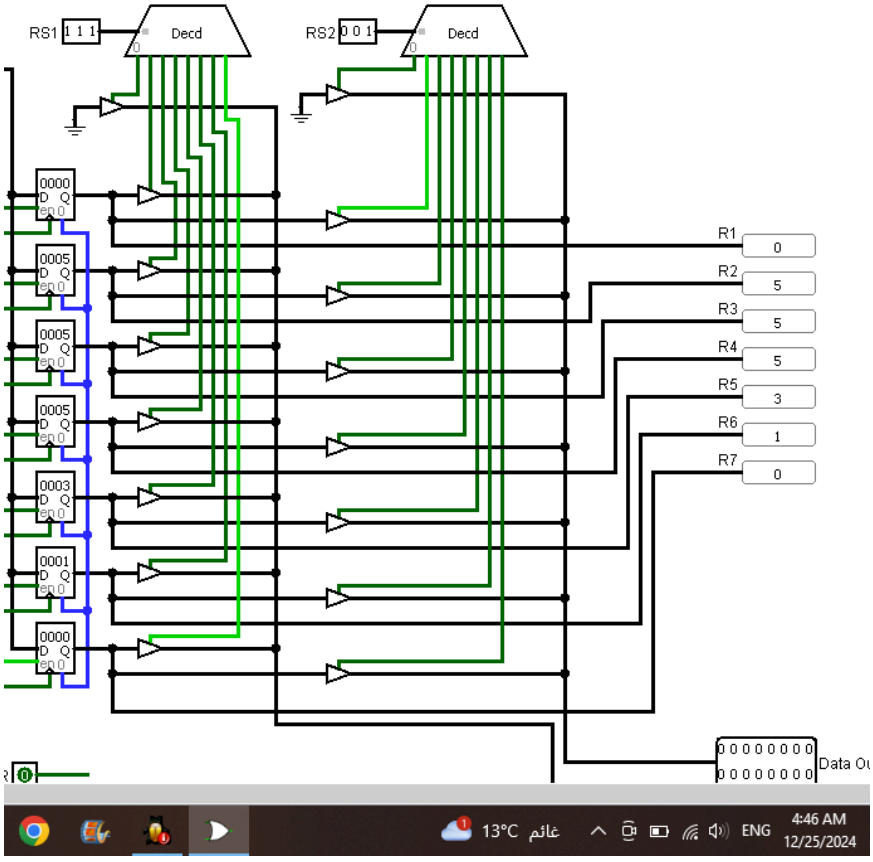
The program starts by initializing R6 to the address of the first element in the data memory



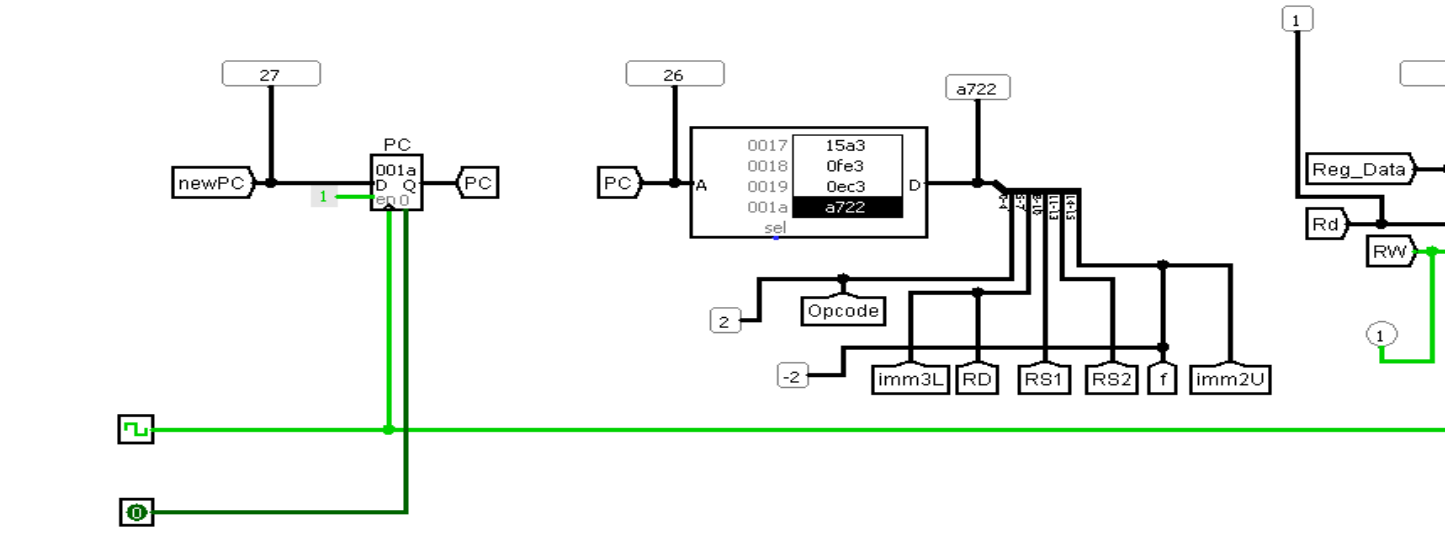
R6=1
R3 (sum) is initialized to 0
R7 (counter) is initialized to 0
R4 is initialized to 5 (elements count)

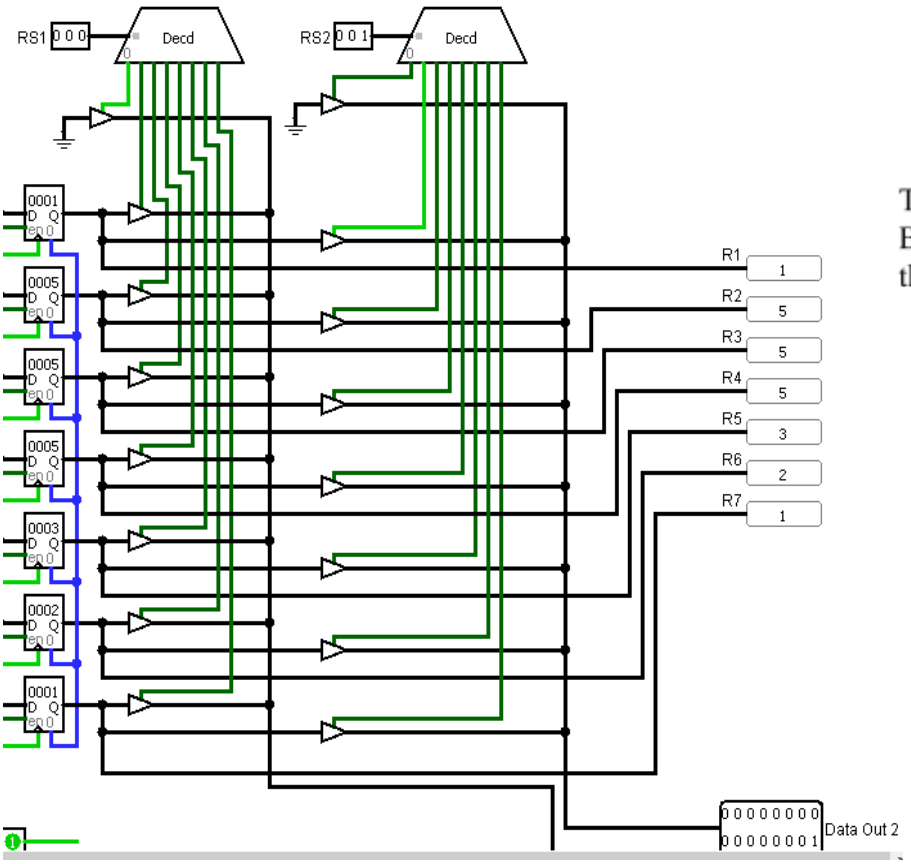


R5= 1 (base address)
The first element in the array is loaded into R2

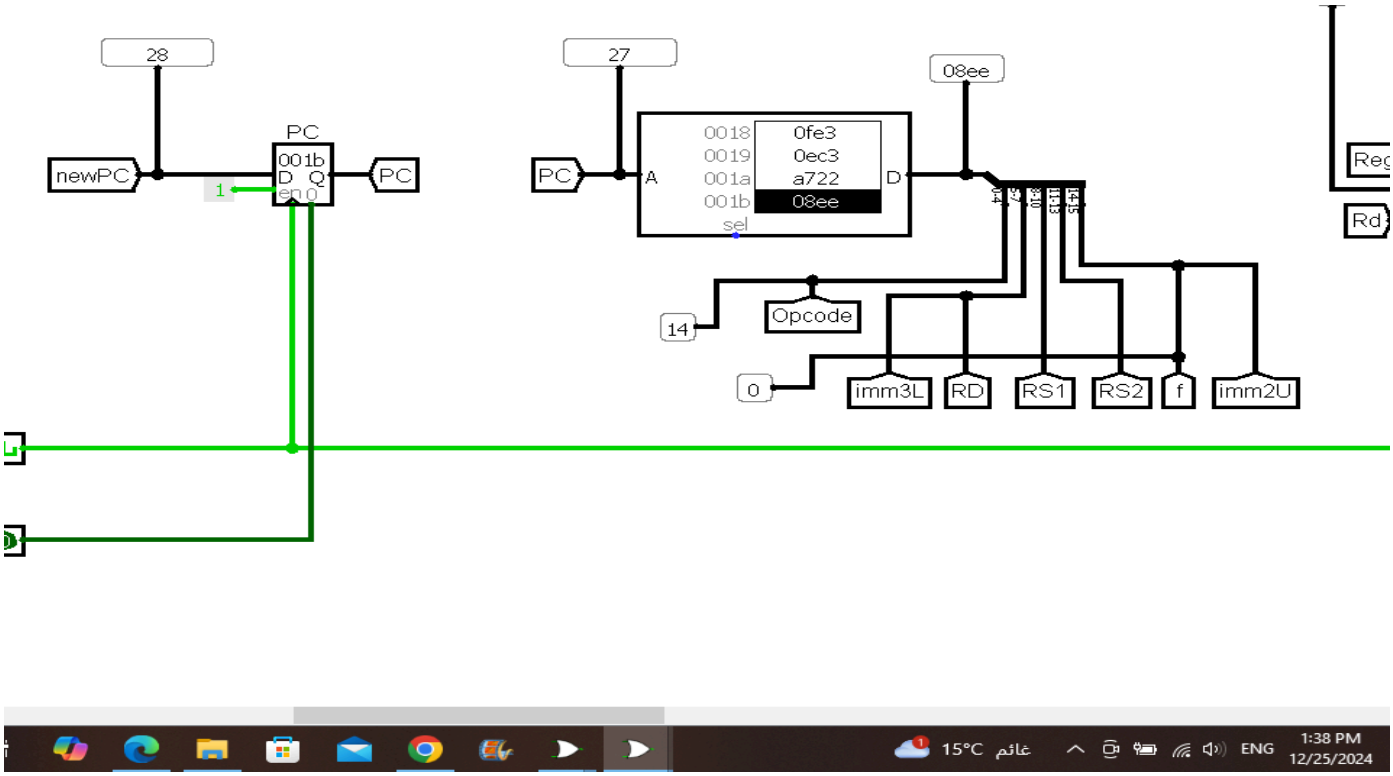


R3=R3+R2
R3=0+5

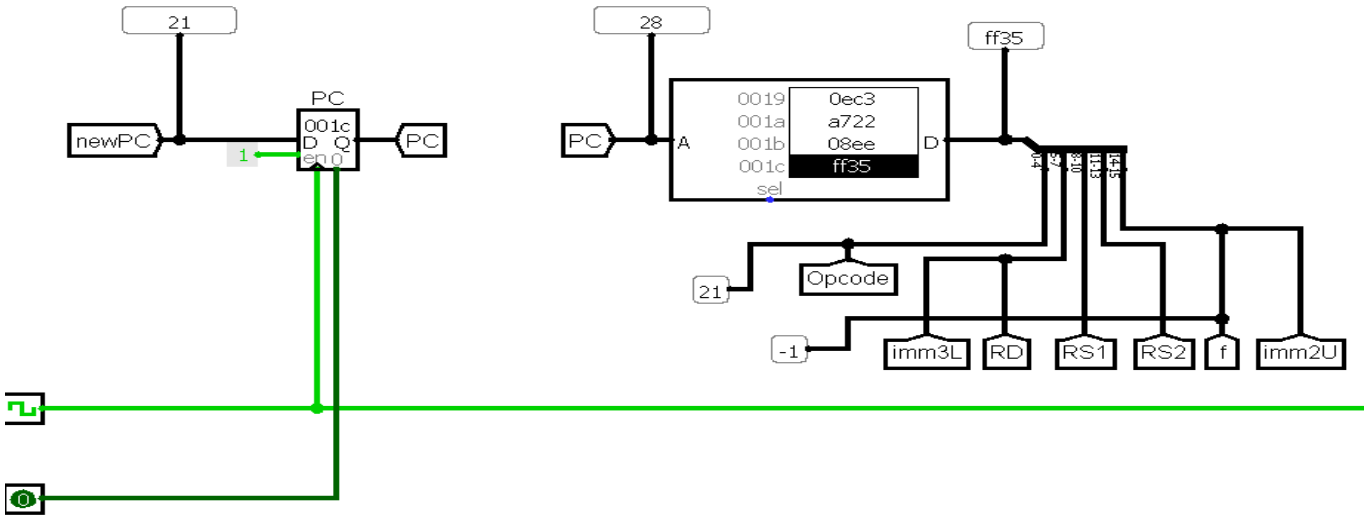




The loop counter R7 is incremented. Because the elements count (R7) is less than the array size (R4) R1 is set to 1

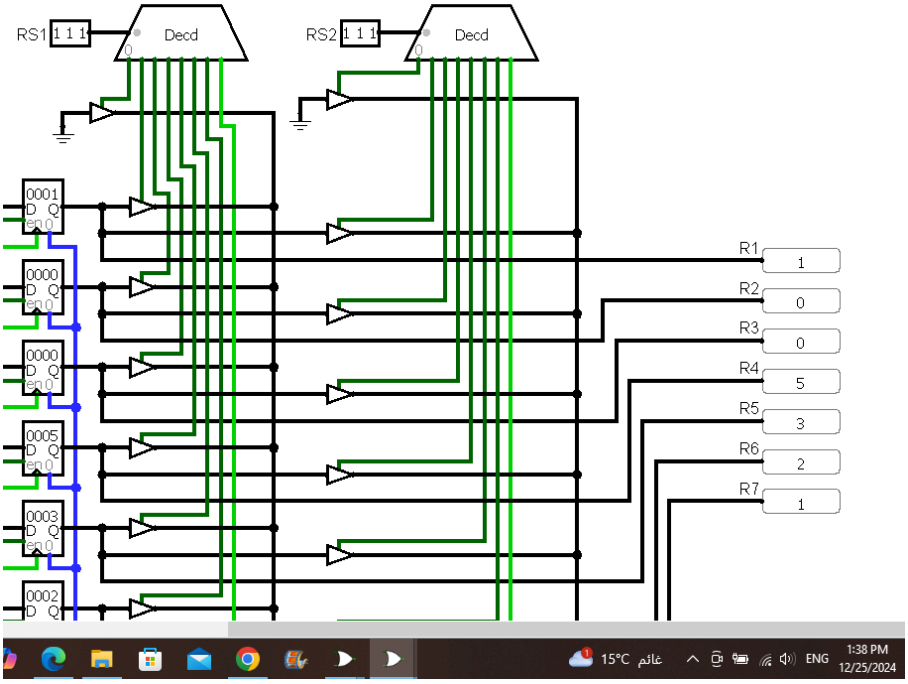

28
PC
001b
D Q
en 0
newPC
1
PC
27
0018 0fe3
0019 0ec3
001a a722
001b 08ee
sel
PC
A
D
08ee
14
Opcode
0
imm3L
RD
RS1
RS2
f
imm2U
Rd

15°C غائم
ENG
1:38 PM
12/25/2024

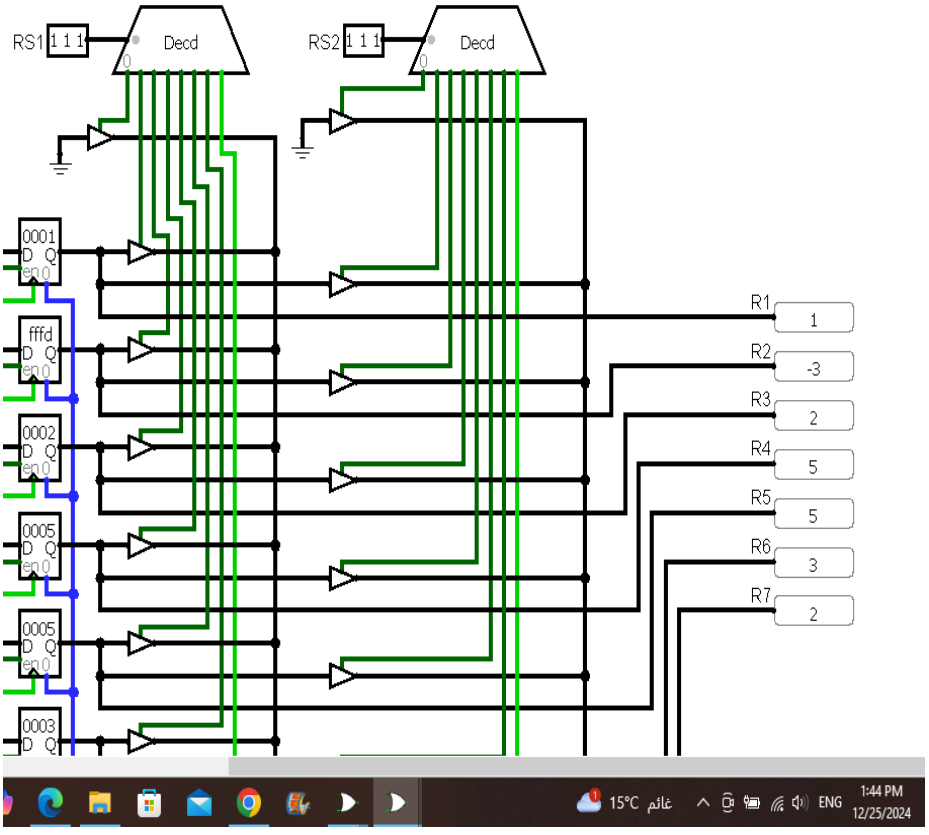

21
PC
001c
D Q
en 0
newPC
1
PC
28
0019 0ec3
001a a722
001b 08ee
001c ff35
sel
PC
A
D
ff35
21
Opcode
-1
imm3L
RD
RS1
RS2
f
imm2U


15°C غائم
ENG
1:38 PM
12/25/2024

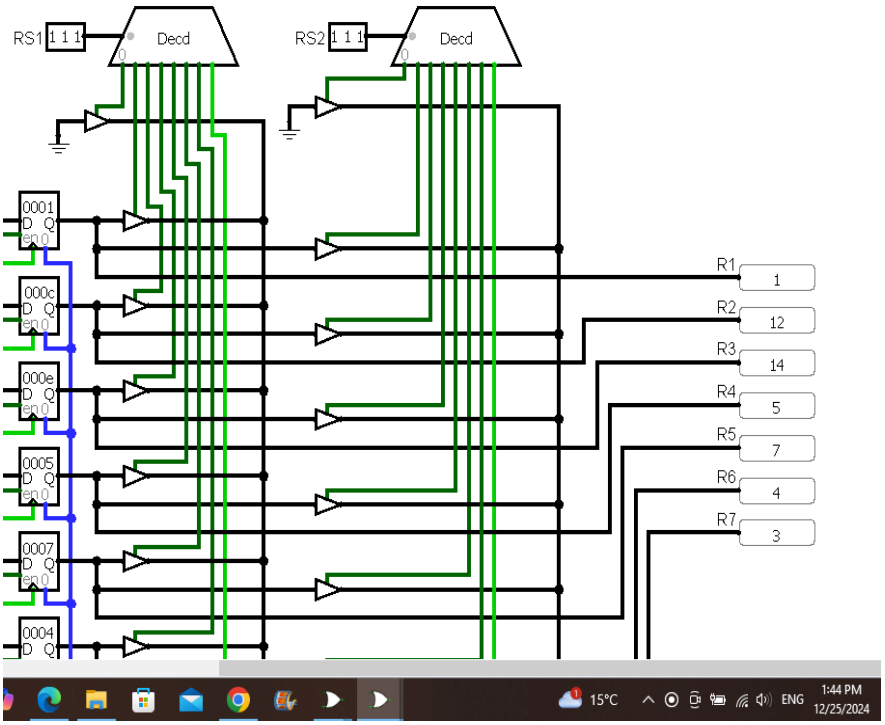
Because the branching condition is false (R1 != R0), the program moves to the next address in memory PC+1 instead of branching.



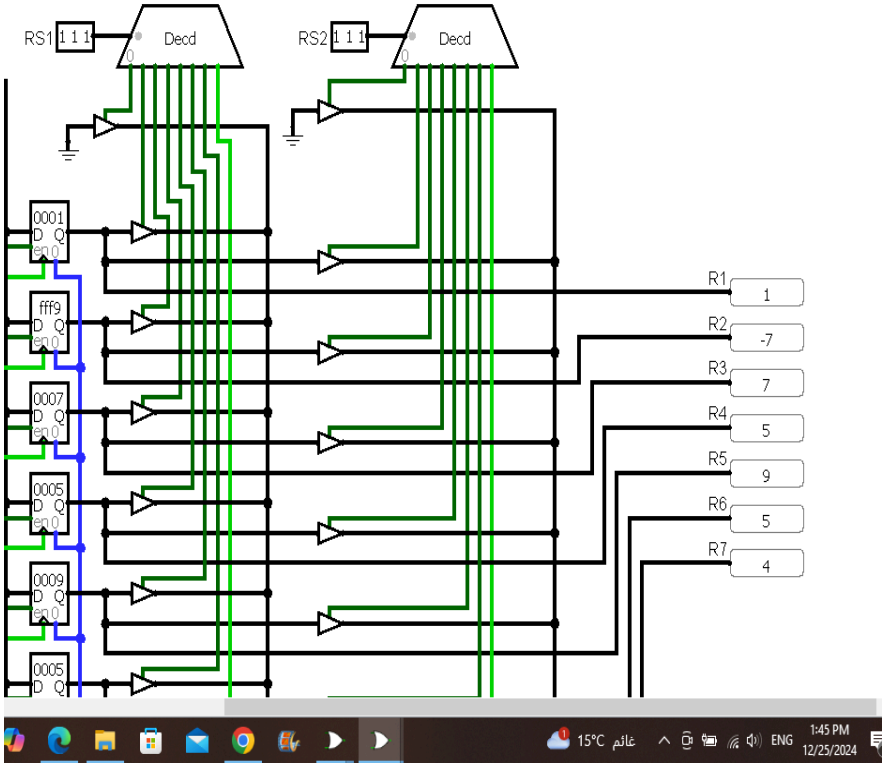
After the first loop



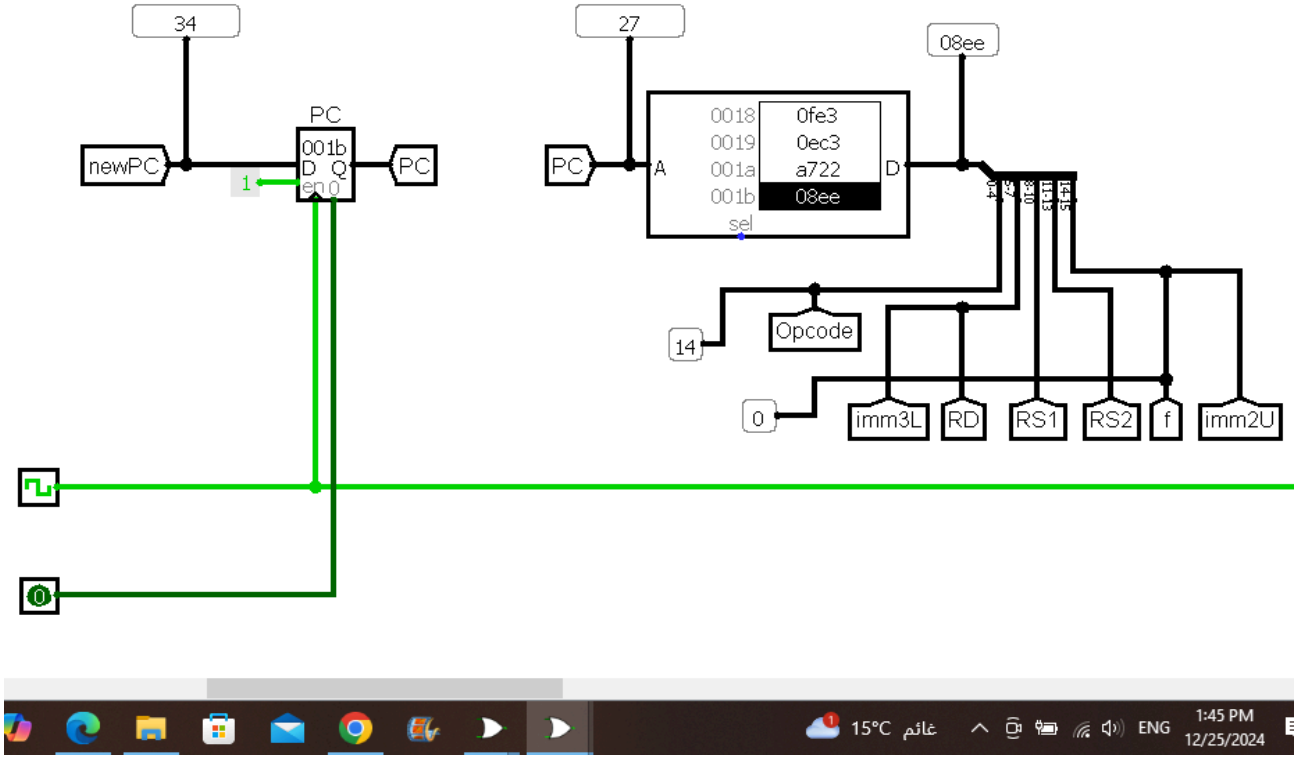
After the second loop



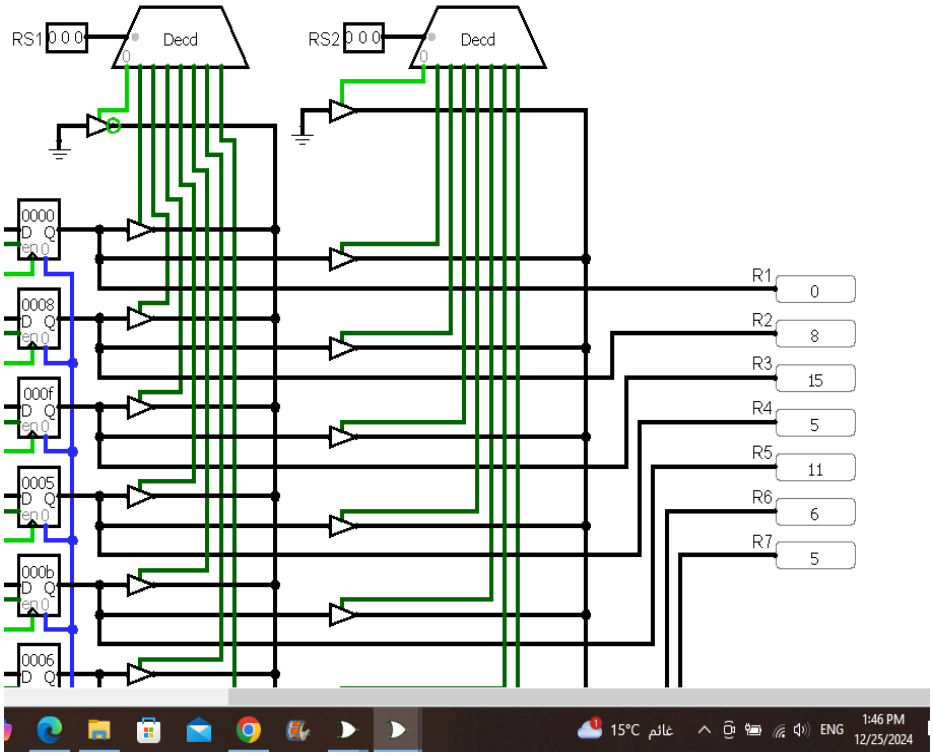
After the third loop



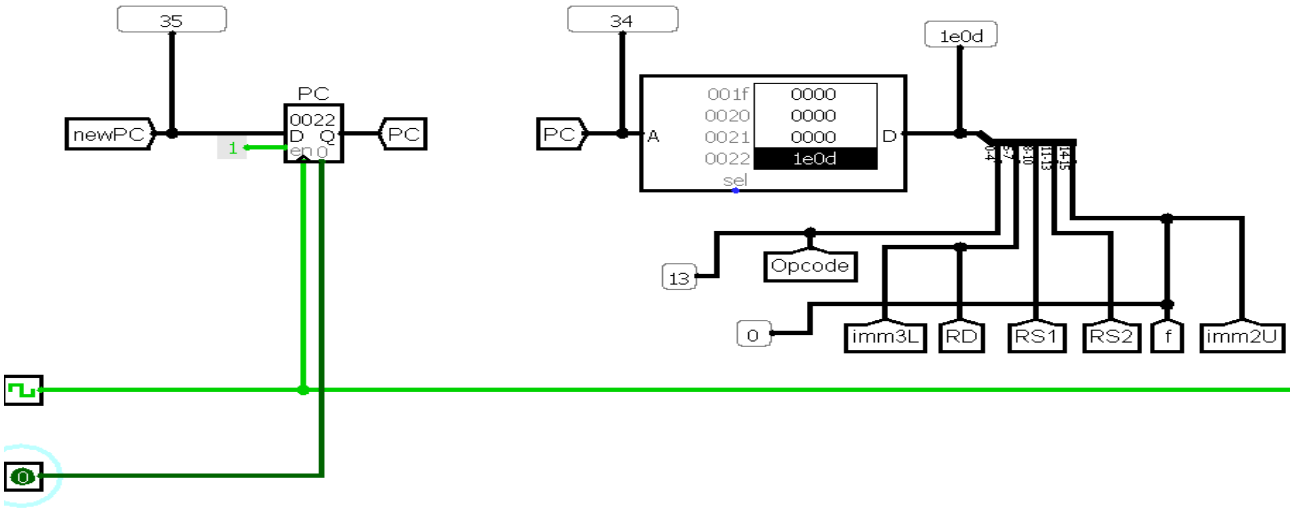
After the forth loop



During the fifth loop, the branching condition becomes true, so the program branches to a new address where the store instruction is stored.

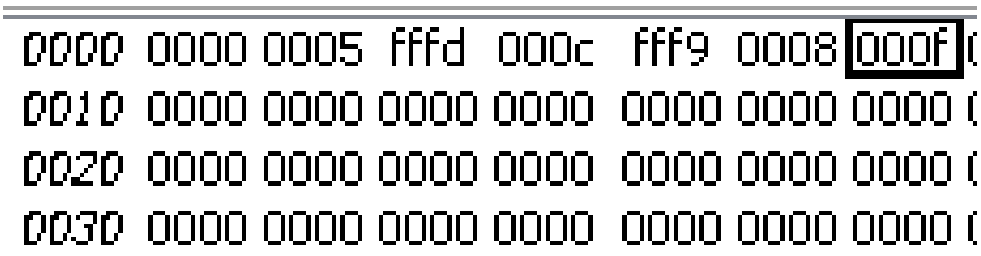


The final sum of 5+-3+12+-7+8= 15, which is stored in R3



The program branches to the store address, which stores the result in address 0X0006 in memory (directly after the last element in the array).

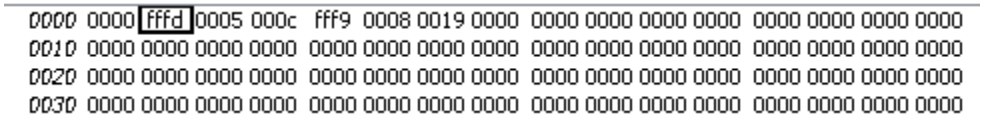
Data memory after:



Testing Scenario 2 (Conditional summation of array elements based on the sign of the element):

The following program checks if a number in an array is positive or negative. If it's positive, it adds it to the sum in R3, and if it's negative it adds it to the sum in R5.

Data Memory Before:



The first value is -3, so the program should add it to the negative sum in R5.

#	Instruction Address	Instruction Format	Binary	HEX	Instruction Description
1	0X0010	ADDI R6,R0,#01	0000100011000011	0X08C3	Initialize R6 to 1 (address of the first element in memory)
2	0X0011	ADDI R3,R0,#00	0000000001100011	0X0063	Initialize the sum (R3) to 0
3	0X0012	ADDI R7,R0,#00	0000000011100011	0X00E3	Initialize the counter R7 to 0
4	0X0013	ADDI R4,R0,#101	0010100010000011	0X2883	Initialize R4 to 5 (elements number)
5	0X0014	LW R2,R6,#00	0000011001001100	0X064C	Load the value in the address stored in R6 (first element) into R2
6	0X0015	BLT #01,R2,R0,#010	0100001001010000	0X4250	Checks if the current element is less than zero, if yes, it branches to a new address PC+10
7	0X0016	ADD R3,R3,R2	0001001101100010	0X1362	Adds the new element stored in R2 to the previous sum in R3
8	0X0017	ADDI R7,R7,#01	0000111111100011	0X0FE3	Increment the elements counter by 1 during each loop
9	0X0018	ADDI R6,R6,#00001	00000111011000011	0X0EC3	Increment R6 by 1 so that it points to the next element in the array

10	0X0019	SLT R1,R7,R4	1010011100100010	0XA722	If the elements count R7 is less than the total elements number R4, set R1 to 1
11	0X001A	BEQ #11,R1,R0,#011	0000100011101110	0X08EE	If R1 equals R0 (both are zeros) then branch to the address of the store instruction
12	0X001B	J #11111111011	1111111100110101	0XFF35	Jump to the load instruction in address 0X0015 to loop again
13	0X001F	ADD R5,R5,R2	0001010110100010	0X15A2	If the element is negative, it adds it to R5.
14	0X0020	J #11111111001	1111111011110101	0XFEF5	Jumps back to the loop to address 20
15	0X0022	SW #000,R3,R6,#00	0001111000001101	0X1E0D	Stores the positive sum in the memory address directly after the last element in the array
16	0X0023	SW #001,R5,R6,#00	0010111000101101	0X2E2D	Stores the negative sum after the positive sum
17	0X0024	J #00000000000	0000000000010101	0X0015	This instruction terminates the execution of the program by jumping to itself indefinitely

Data Memory After:

After the program execution is completed, the sum of positive numbers (25) is stored in address 0X0006, and the sum of negative numbers (-10) is stored in address 0X0007.

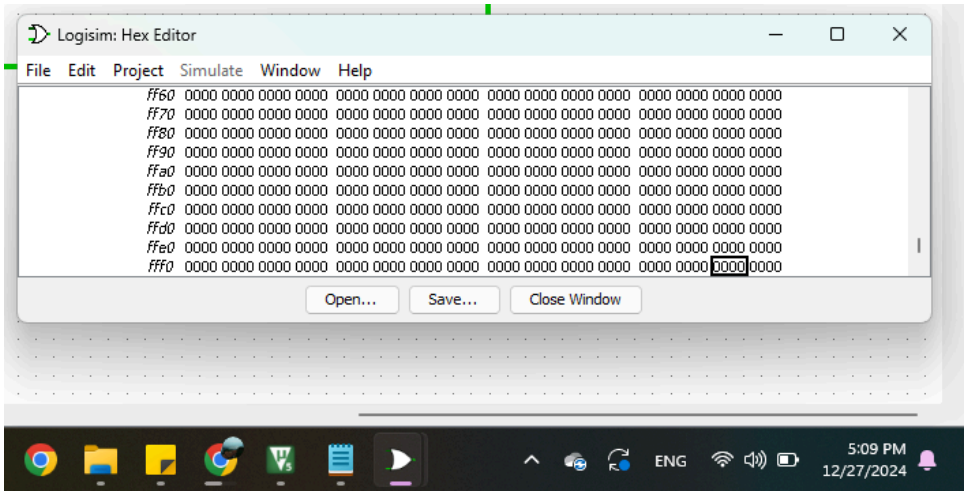
NOTE: In the two scenarios, instruction execution doesn't start from PC=0, and the instructions are not stored in address 0X0000 in the instruction memory. This approach is justified entirely in the improvements section of this report. (See page 54)

Testing Scenario 3 (Swapping Registers) program to demonstrate the usage of stack segment structure and execution on procedures:

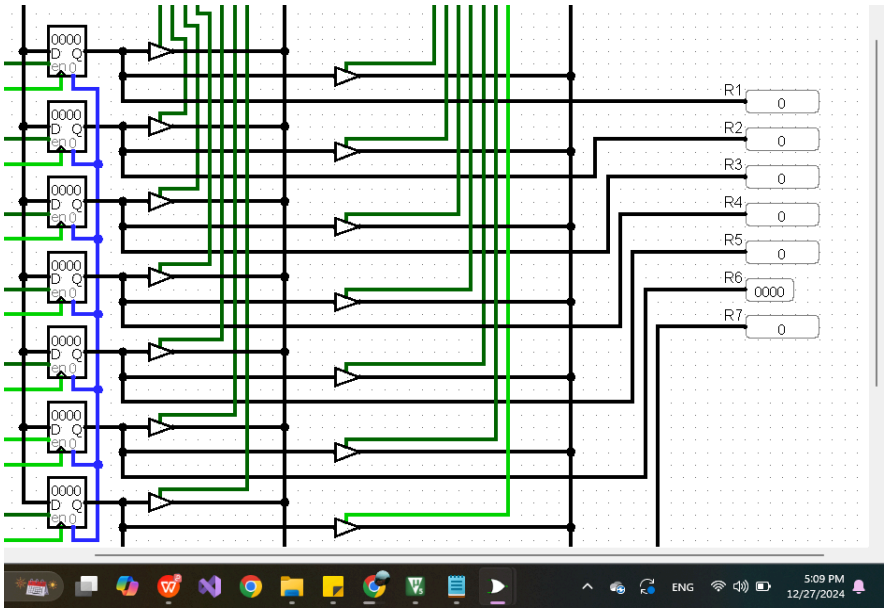
Instruction memory:

```
0000 f8c3 5023 1043 00b6 0015 0000 0000 0000 0e0d fec3 160d fec3 0ec3 062c 0ec3 064c
0010 0717 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0020 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0030 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0040 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0050 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

Data memory before:



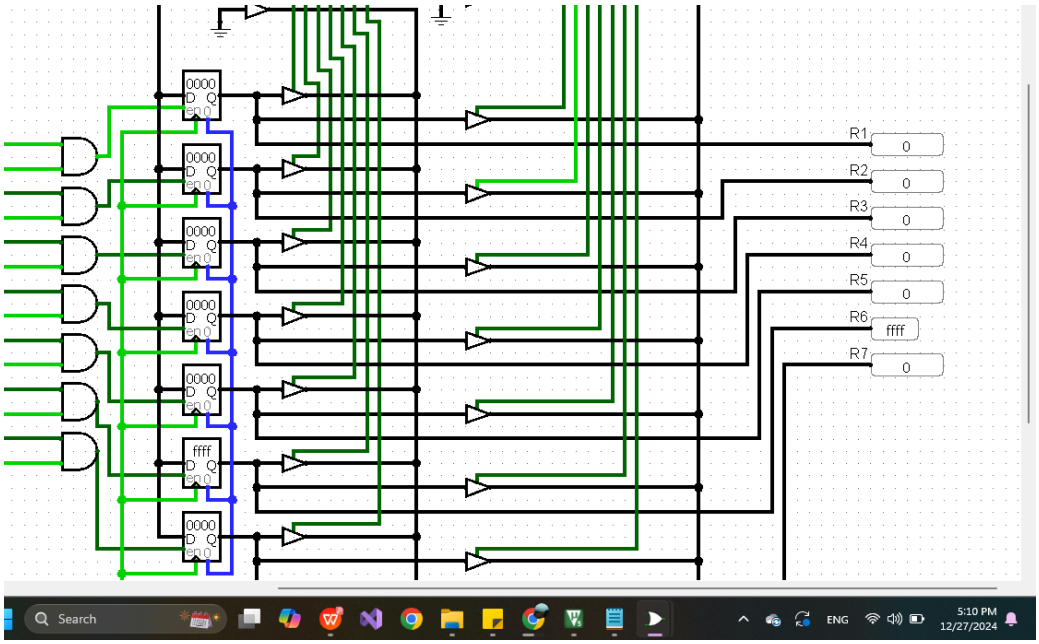
Register file before:



Swap Program: swap values of R1 (10) and R2 (2)

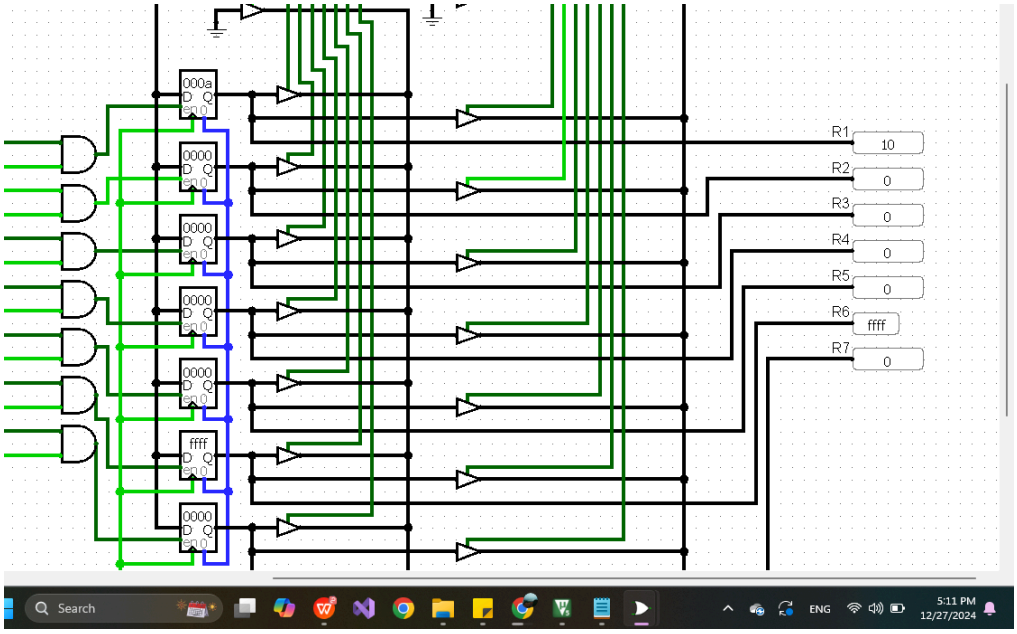
0x0 > ADDI R6, R0, #0x1F → 1111/1 000 /110 0/0011 -- F8C3

initialize R6 stack pointer (SP)



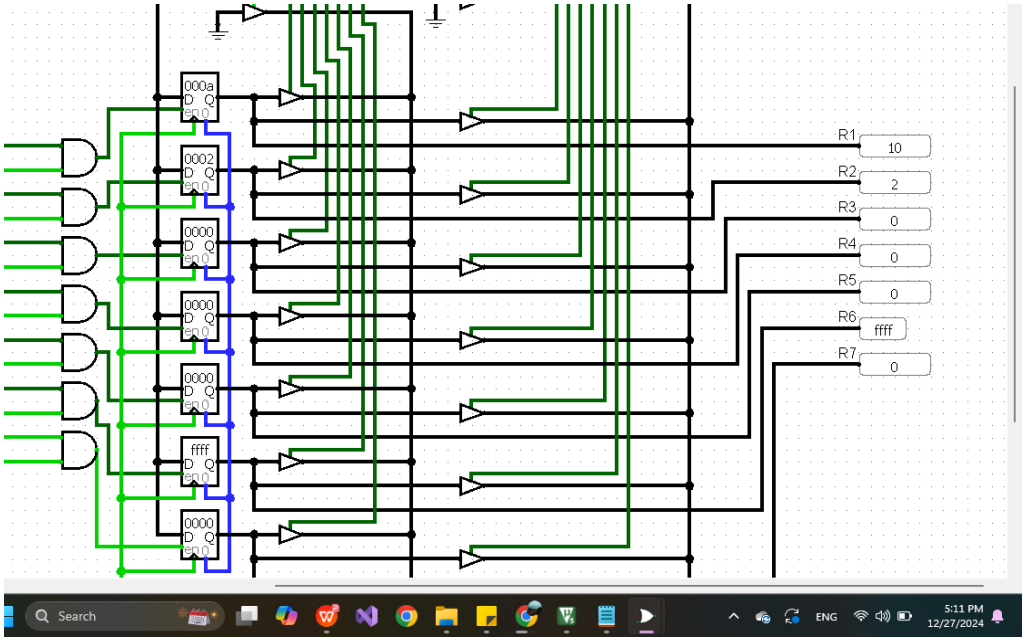
0x1 > ADDI R1, R0, #01010 → 0101/0 000 /001 0/0011 - 5023

Load R1 with 10



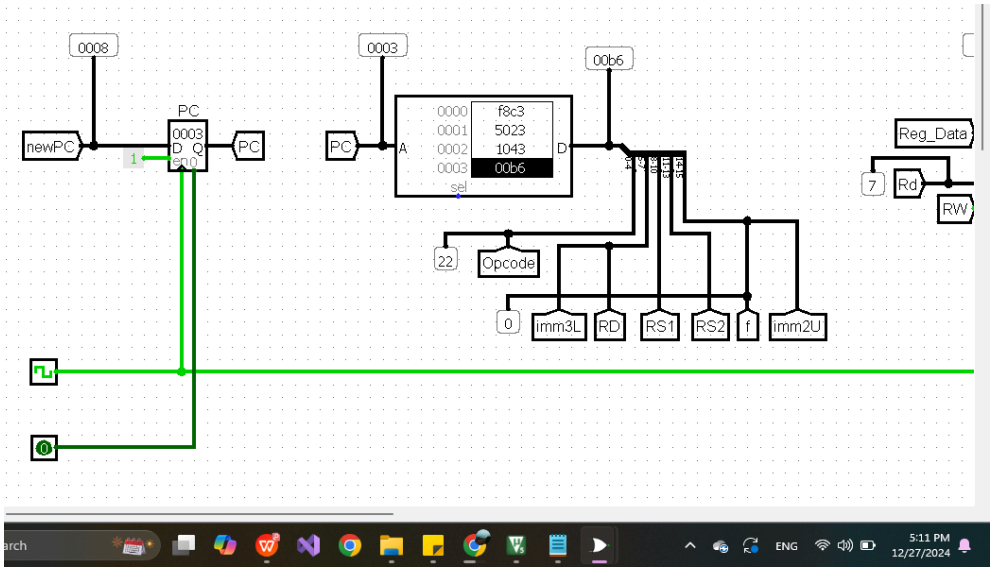
0x2 > ADDI R2, R0, #00010 → 0001/0 000 /010 0/0011 - 1043

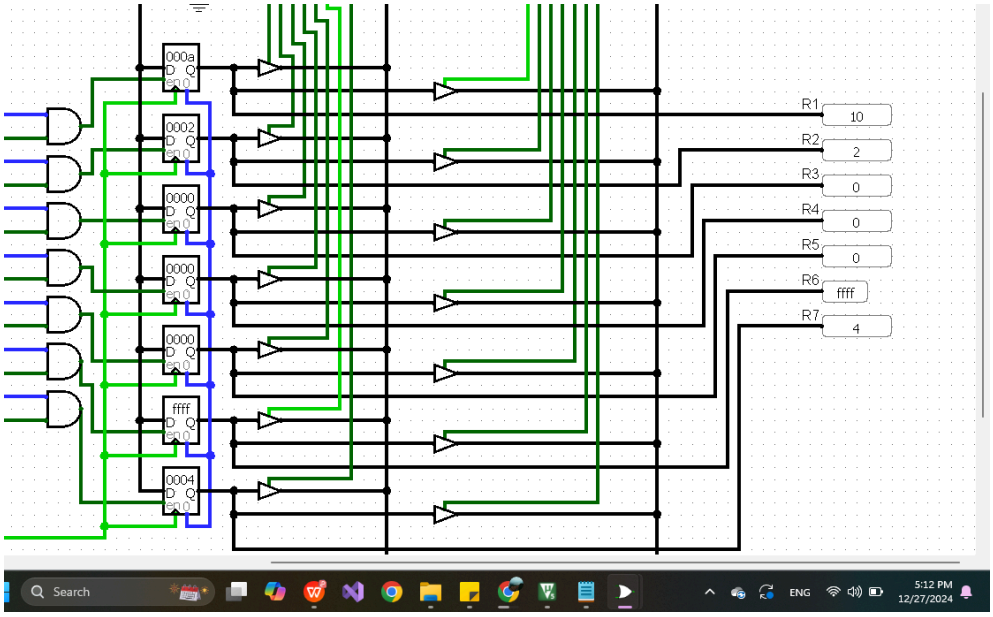
Load R2 with 2



0x3 > JAL #101 → 0000 /0000 /101 1/0110 - 00B6

Call SwapSubroutine in 0x8 and store the return address in R7



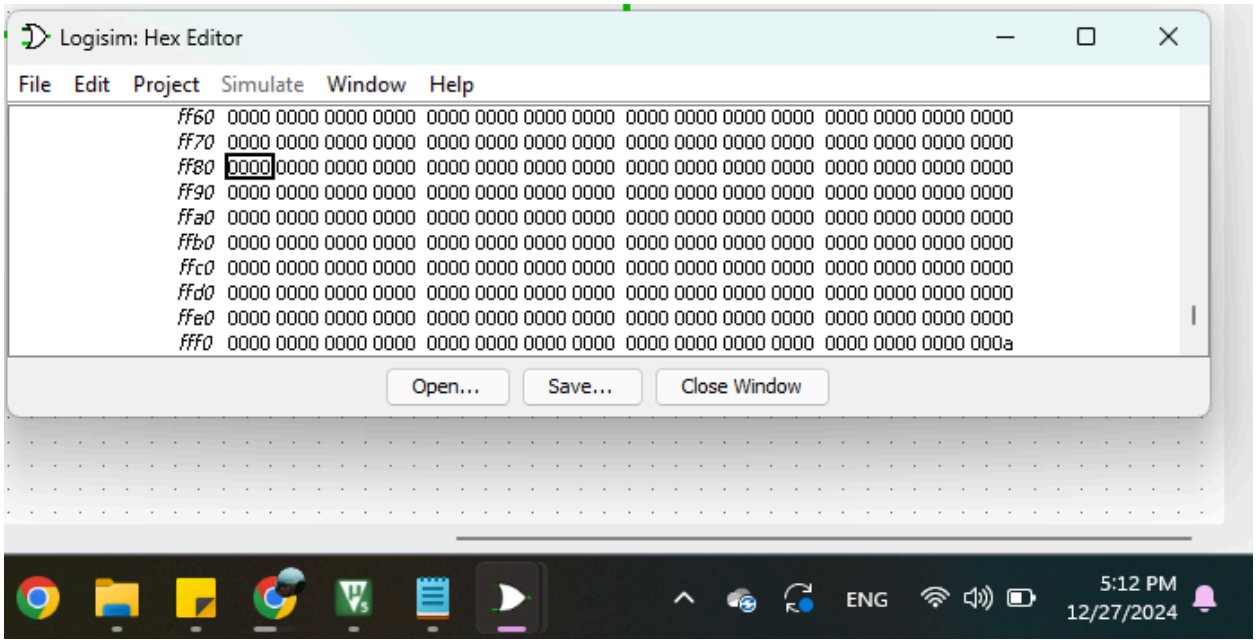


SwapSubroutine:

*Save R1 and R2 to the stack

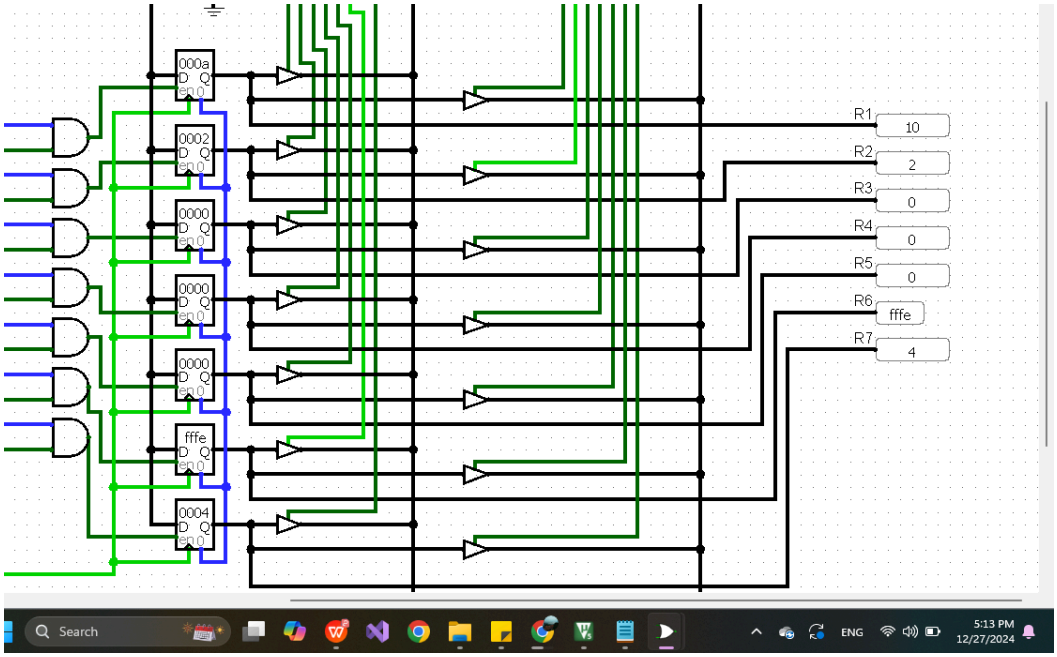
0x8 > SW 000, R6, R1, 00 → 00 00/1 110 /000 0/1101 - 0E0D

Push R1 onto the stack



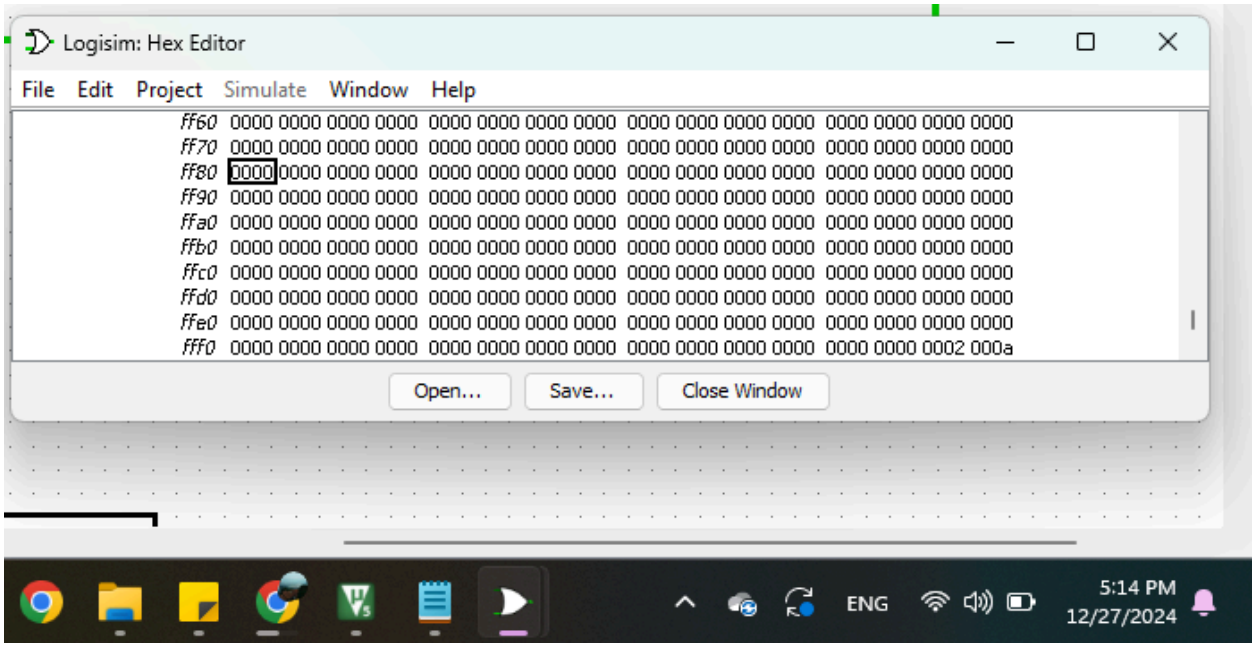
0x9 > ADDI R6, R6, #11111 → 1111/1 110 /110 0/0011 - FEC3

Decrement SP



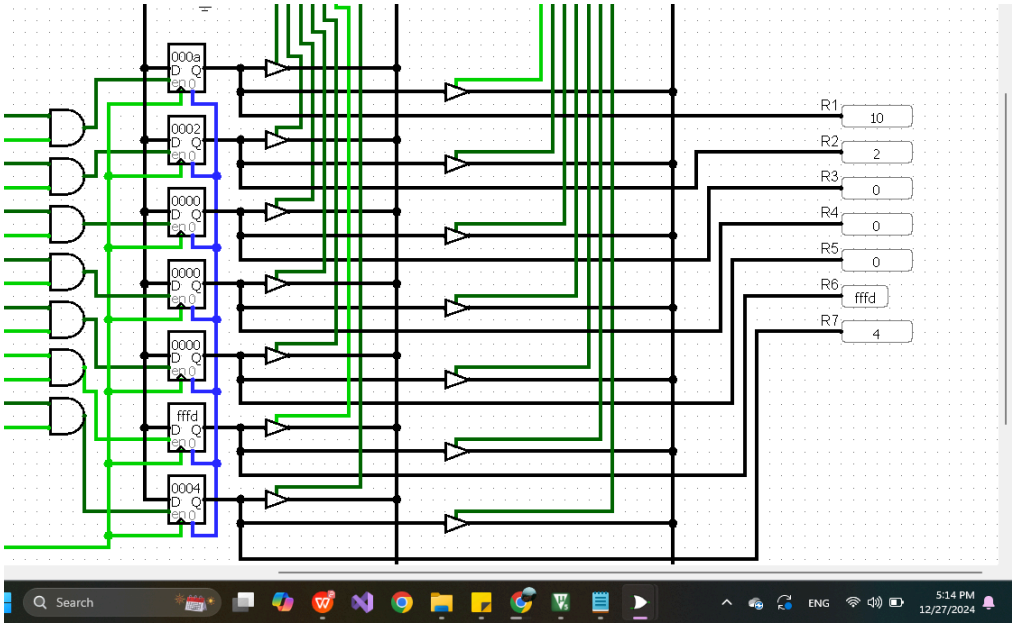
0xa > SW 000, R6, R2, 00 → 00 01/0 110 /000 0/1101 - 160D

Push R2 onto the stack



0xb > ADDI R6, R6, #11111 → 1111/1 110 /110 0/0011 - FEC3

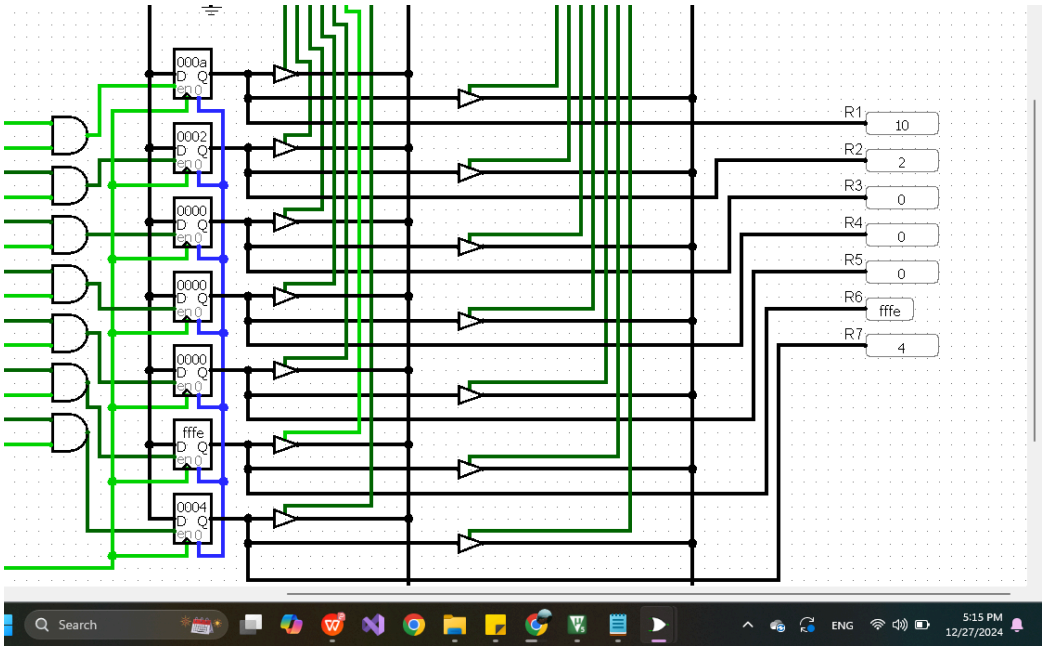
Decrement SP



*Restore R1 and R2 in reverse order

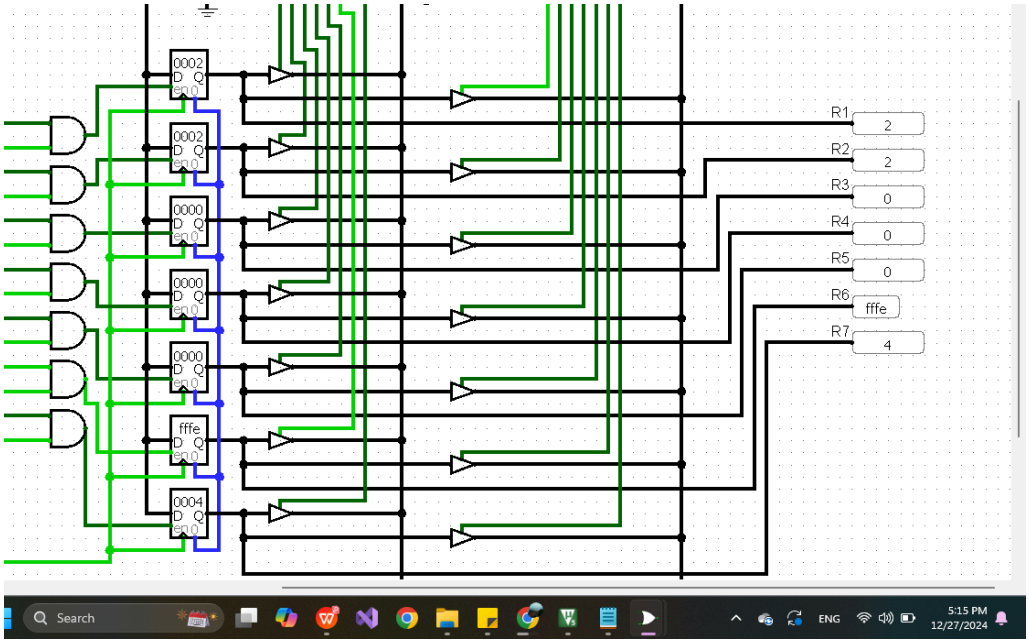
0xc > ADDI R6, R6, #1 → 0000/1 110 /110 0/0011 - 0EC3

Increment SP



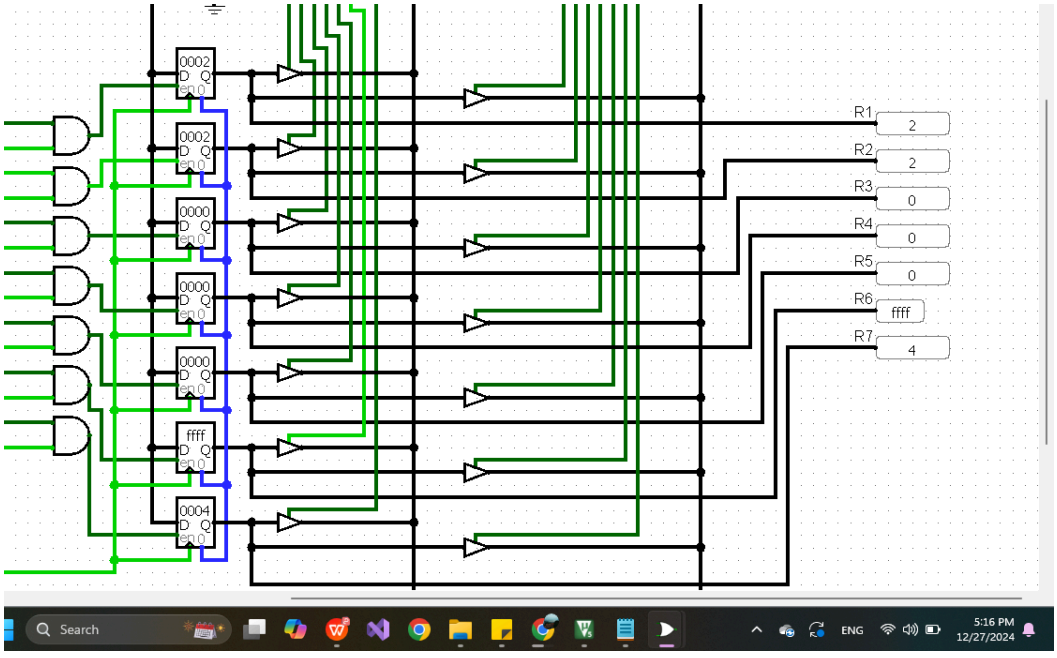
0xd > LW R1, R6, #0 → 0000/0 110 /001 0/1100 - 062C

Pop R2 into R1



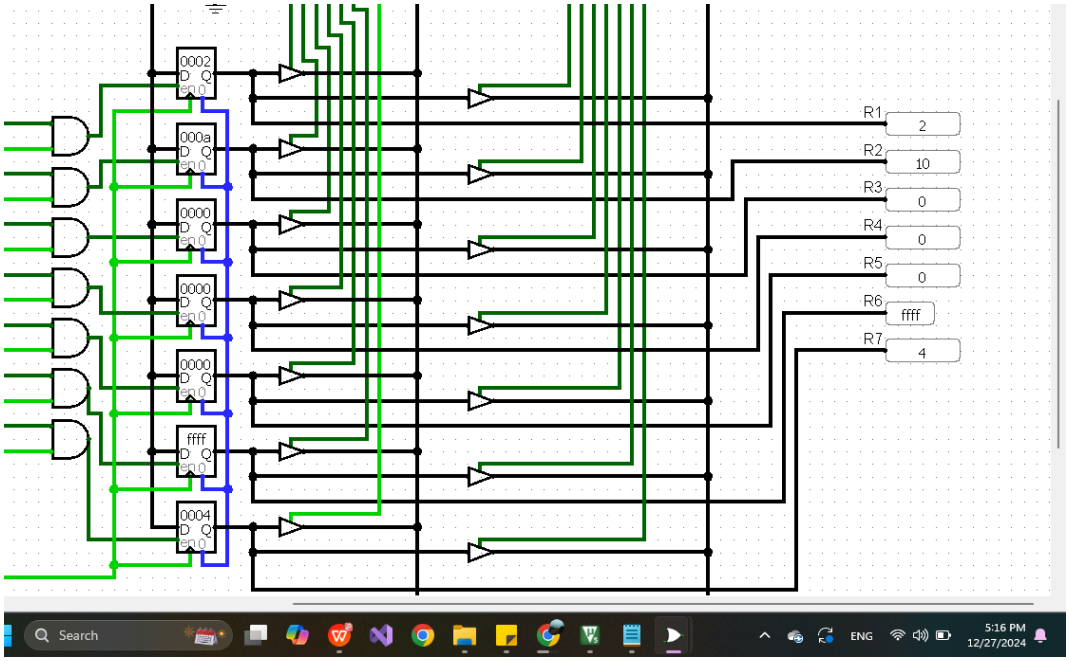
0xe > ADDI R6, R6, #1 → 0000/1 110 /110 0/0011 - 0EC3

Increment SP



0xf > LW R2, R6, #0 → 0000/0 110 /010 0/1100 - 064C

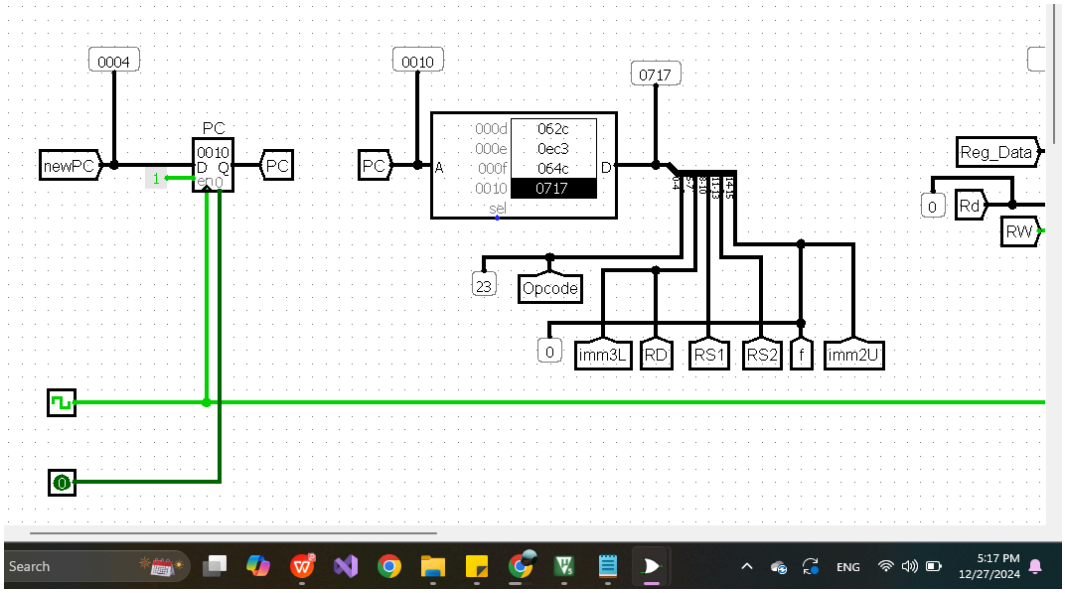
Pop R1 into R2



*Return to main program

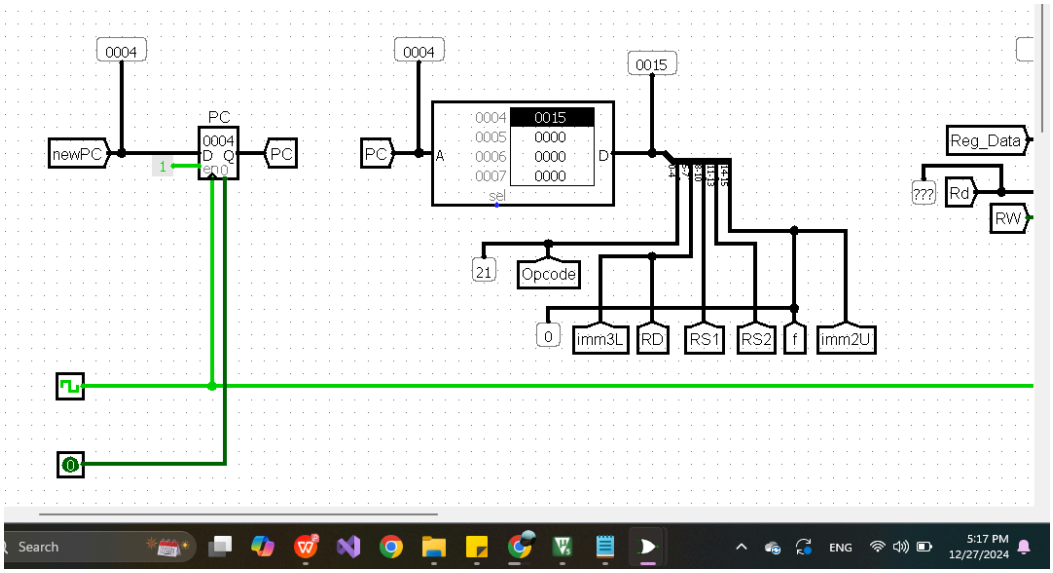
JALR R0, R7, #0000 → 0x10 > 0000/0 111 /000 1/0111 - 0717

Return to main



0x4 > J #0 - 0015

HALT

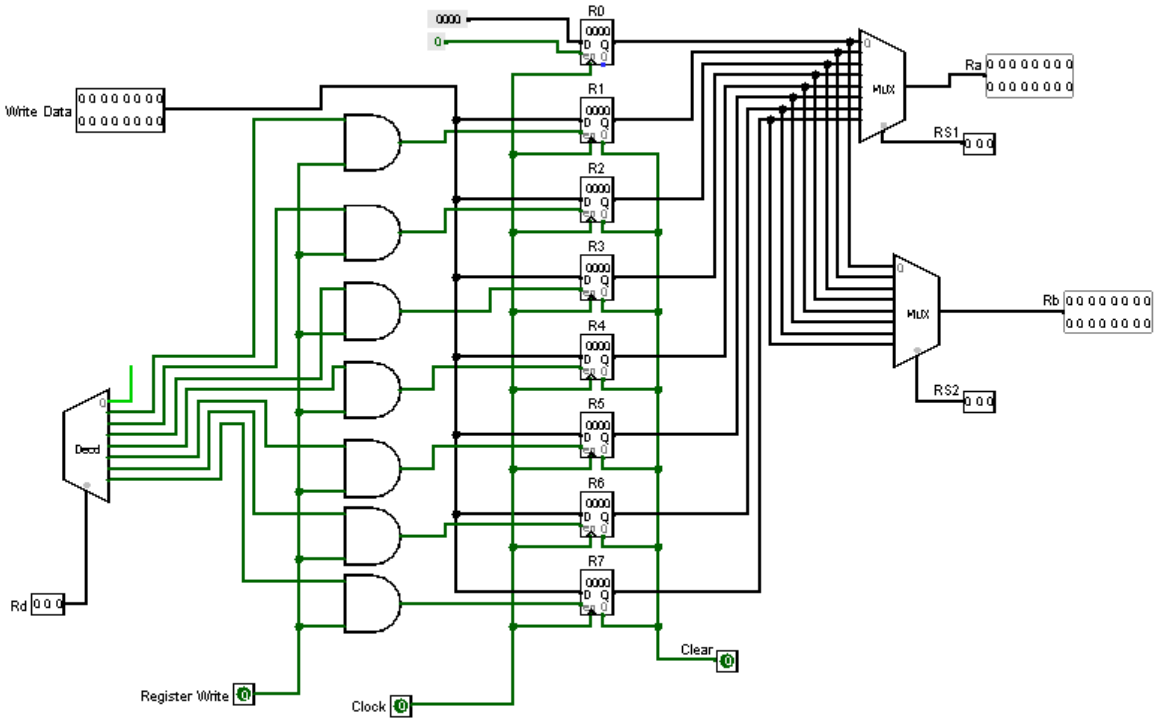


Alternative Designs

1-Register File:

In our design, we used three state gates and decoders to implement the register file. The two decoders generate signals that enable only one three state gate at a time, therefore only one register can be accessed by each decoder while the others remain in high impedance.

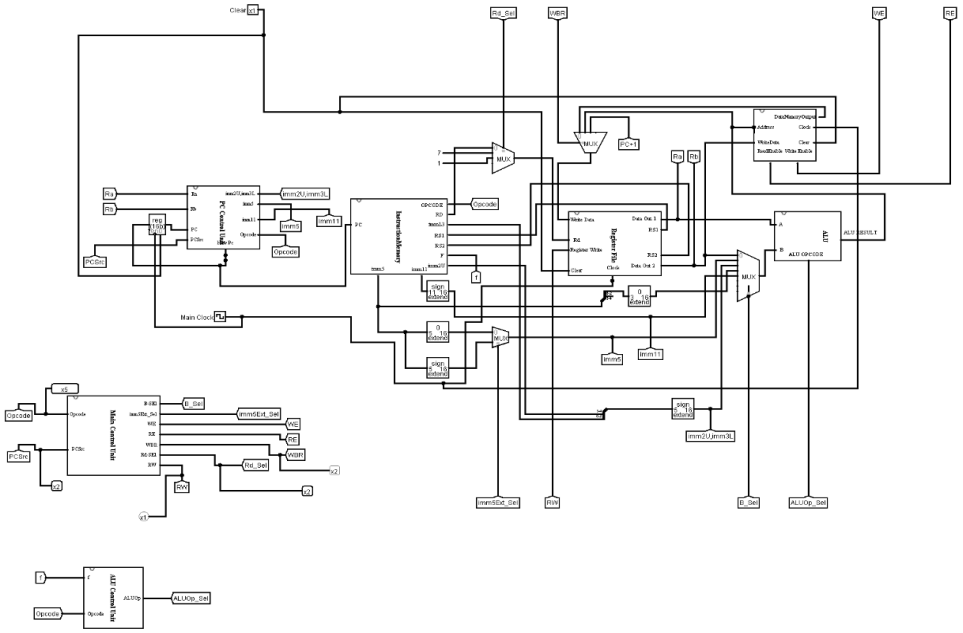
An alternative design can be implemented using AND gates and two muxes that decide which register is enabled for writing. The two approaches yield the same results but the three state gate with decoders method is better for large designs, while the AND gates and multiplexers method is better for smaller designs that require fewer connections.



The main decoder uses Rd to decide which AND gate is set to 1, and if the Register Write signal is high, then the AND gate corresponding to the chosen register enables the register for writing, and the data on Write Data input port is transferred to the targeted register.

Additionally, the two muxes use RS1 and RS2 as selection lines, and based on their values, one of the registers is chosen and its value is sent to the output port.

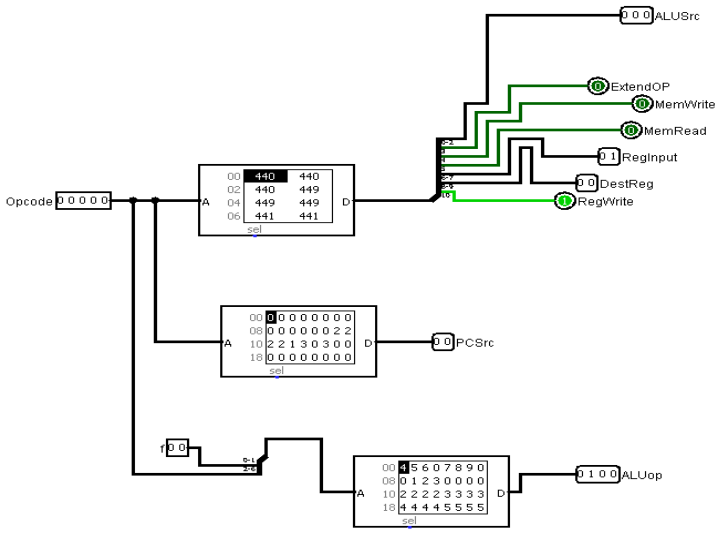
2-Datapath:



Our first datapath design as in the figure above used blocks to represent each component in the datapath. All elements are connected together using wires and the output of one element is the input to one or more others. This design is complex and harder to track in case of an error. Also, it is difficult to observe both instruction and data memory in real time. On the other hand, a more modular design has the advantage of reusability and modification, and is easier to track for more complex designs. However, our design clearly supports modularity in most components, and uses tunnels to reduce the number of connecting wires, which makes the design simpler and easier to debug.

3-Main Control Unit and ALU Control Unit:

In our design, we implemented separate modules for each unit. The main control unit generates control signals for the register file, for enabling/ disabling reading and writing on the data memory, and controlling the data flow between components in the datapath. On the other hand, the ALU control unit generates only a 3-bit signal that controls the ALU operations.



However, an alternative design approach can be implemented by using a single block that has a separate ROM for the ALU operations in addition to the main ROM, as in the figure above. We chose to take the approach of separating control unit modules, which is only a matter of preference and doesn't change the overall design, breaking the design into smaller components can make it easier to manage sometimes.

Issues and Limitations

Designing the single cycle processor on Logisim was a challenge due to the lack of resources that provided guidance on how to fix common errors and test the final hierarchical design. However, the overall experience was informative and provided a deep understanding of how the processor works, and how different addressing modes are implemented.

In addition, we implemented all the listed instructions in the project document successfully, and we even tested a real scenario that shows how the elements of an array are added. In this part, we faced some issues because some instructions were not properly encoded, or didn't work properly with others, and the branching instructions required more effort because in order to test them, we had to expect their output beforehand. However, we still managed to find alternative designs for some components that can be implemented with more modularity.

Finally, we encountered some issues while designing the PC control unit due to the complexity of the design and the difficulty of connecting all parts so that it can work together. Also, the limited number of registers and limited range of branch instructions was also a bit of a challenge while performing test scenarios that included loops, add to that the fact that no branch instruction saves PC address.

But despite all the difficulties, the recorded tutorials provided guidance on how to design several components and how to think of the design as a whole.

Improvements

An additional feature that can be added to the current design is to have an option that asks the user if he/she wants to start instruction execution from a specific address or start sequentially from the beginning.

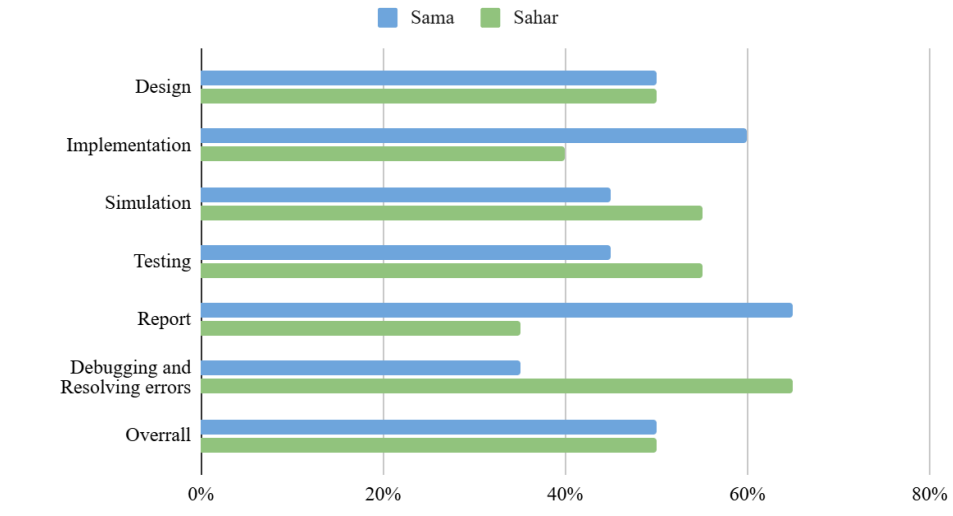
This feature is especially important because the instruction memory stores a number of programs that start at different locations in memory, so the user might want to execute only a specific program (for example he wants to sum the elements in an array), without having to go through all other instructions that comes before this specific program stored (e,g,. in location 0X0010 in memory).

However, this feature can be implemented using a mux that has two options to choose from, one is starting the execution sequentially from address 0X0000 in memory, and the other is giving the user the option of choosing a starting address where the desired program is stored.

Teamwork

The workload was divided between the two of us, and we both participated in the process of thinking, designing, testing, debugging, and report writing.

The bar chart below shows an approximate percentage of the work done in each stage among the team.



تم بحمد الله