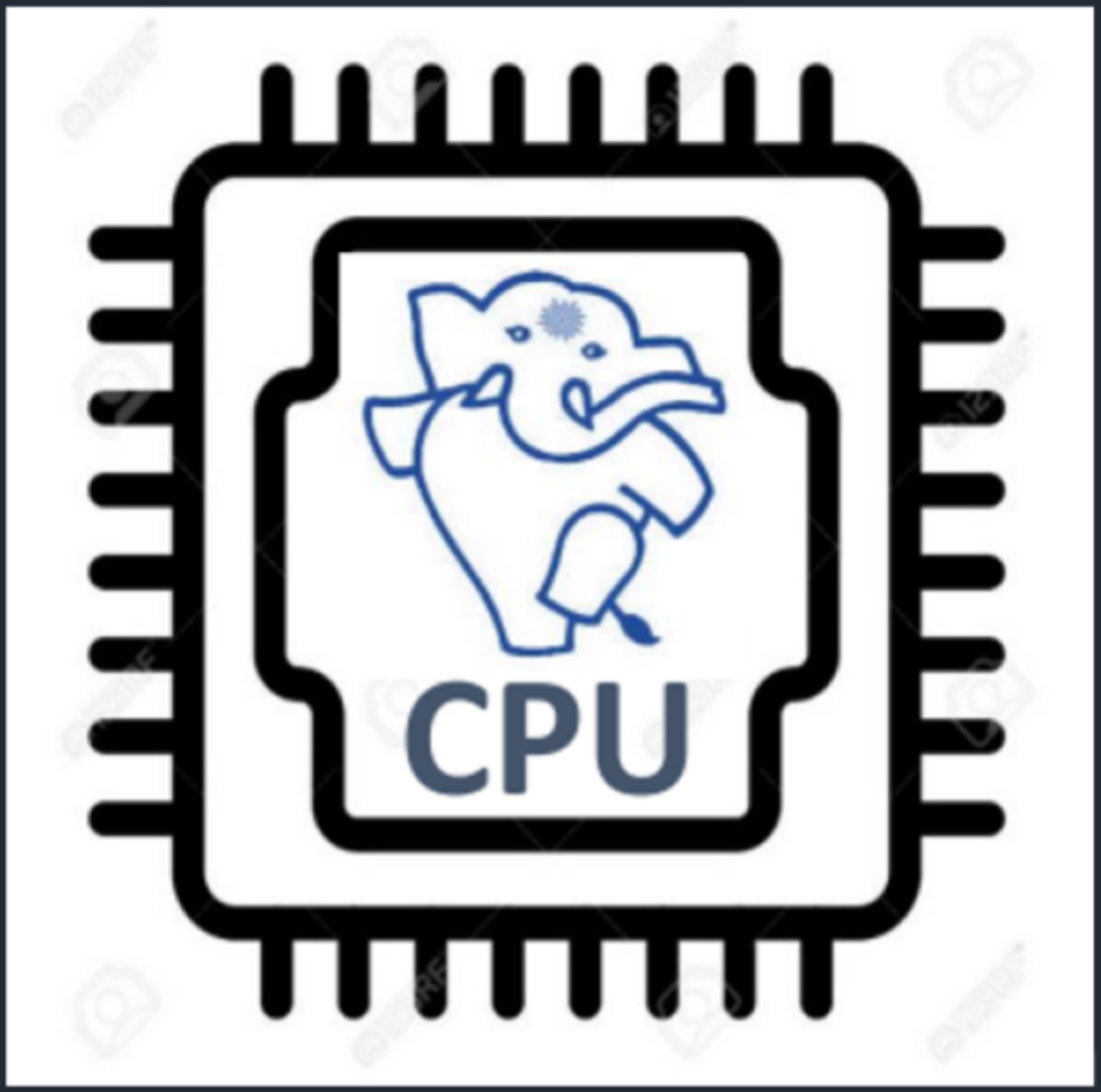


GAJENDRA-I

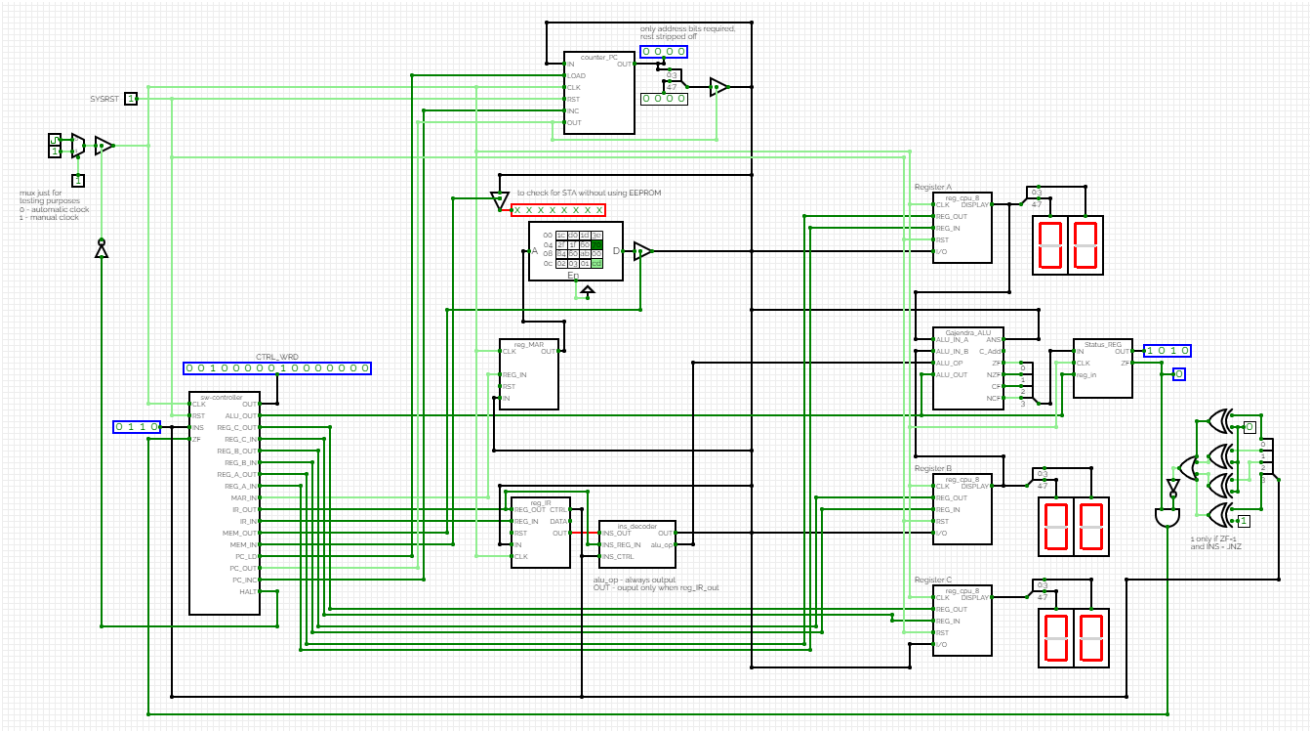
Professor -Ayon Chakraborty

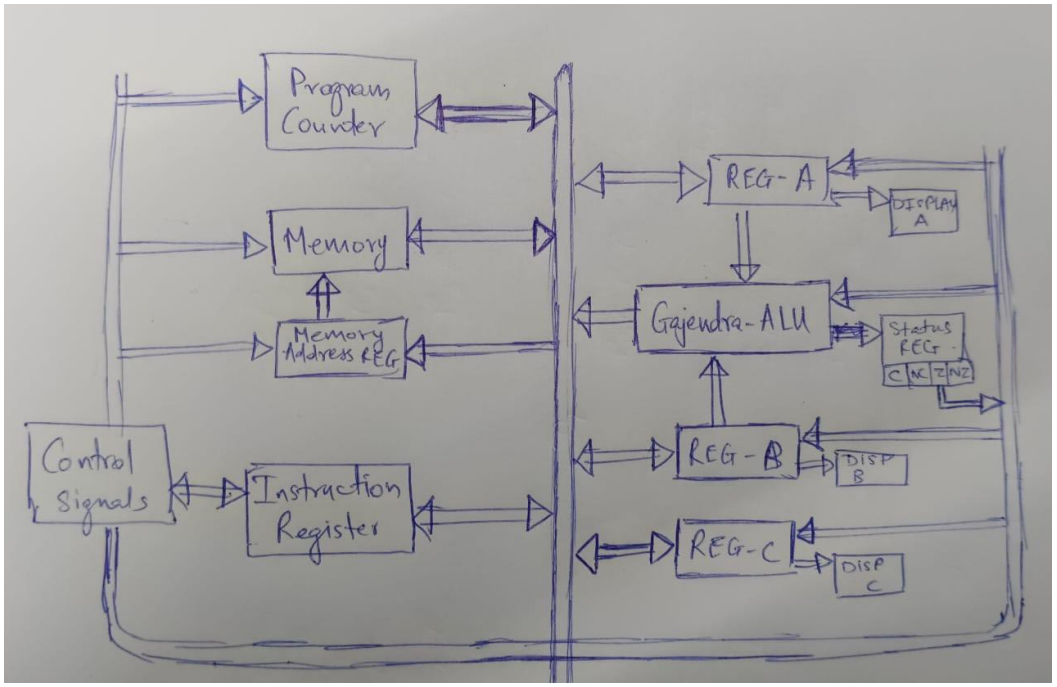
Contributors-

1. Dev Mehta-CS22B007
2. Shreyas Bargale-CS22B016

Features of Gajendra-I CPU-

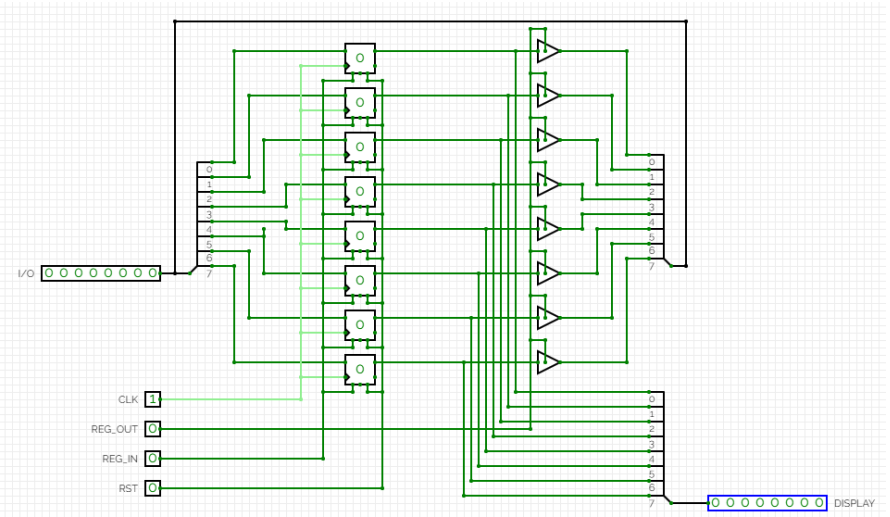
1. Supports 16 bytes of memory(ROM)
2. Recommended 8 locations of program space and 8 locations of data space.
3. CPU has an 8 bit bus which carries both instructions and data.
4. 8 bit ALU for arithmetic operations
5. Software controller used for controlling the execution of program.
6. Also implemented FSM controller (but with fewer instructions).
7. Instructions-
    - Supports instructions like CMP(compare), SWAP, MOV instructions, SHIFT (Left/Right).
    - Supports STA
    - Supports add, subtract, load, etc.
    - Supports JNZ.(Jump if not zero)





COMPONENTS OF GAJENDRA-I

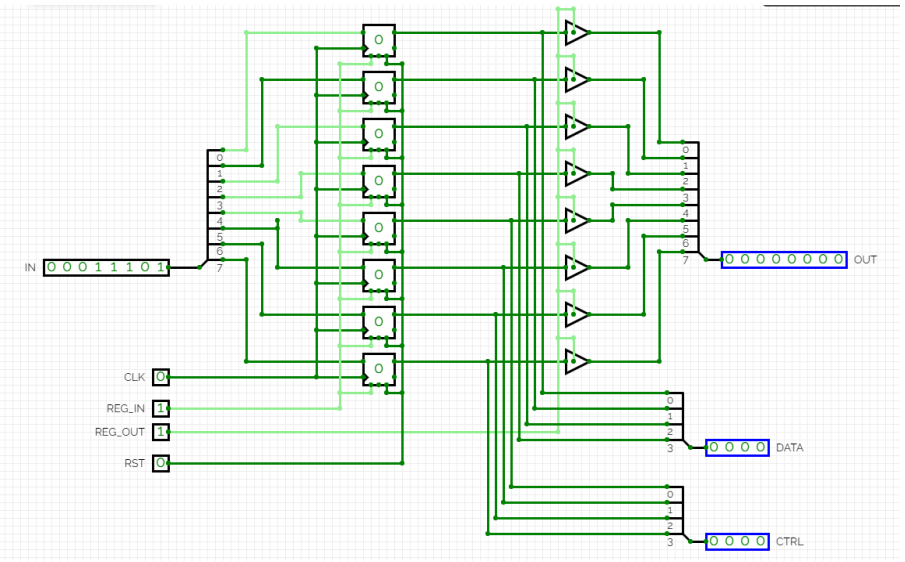
1. General CPU registers
(reg_cpu_8)



- General CPU registers are used to store data on which operations like ADD, SUB, CMP, left shift, right shift are to be performed.
- Data can also be transferred between the General CPU registers through operations like SWAP and MOV.

I/O : common input/output bus (8 bit)
REG_IN : if 1, register takes input from the bus
REG_OUT : if 1, register gives output to the bus
(Note : both REG_IN & REG_OUT must not be 1 together)
RST : resets all bits (D - flip flops) to 0
CLK : clock controlling the D - flip flops (connected to system clock)

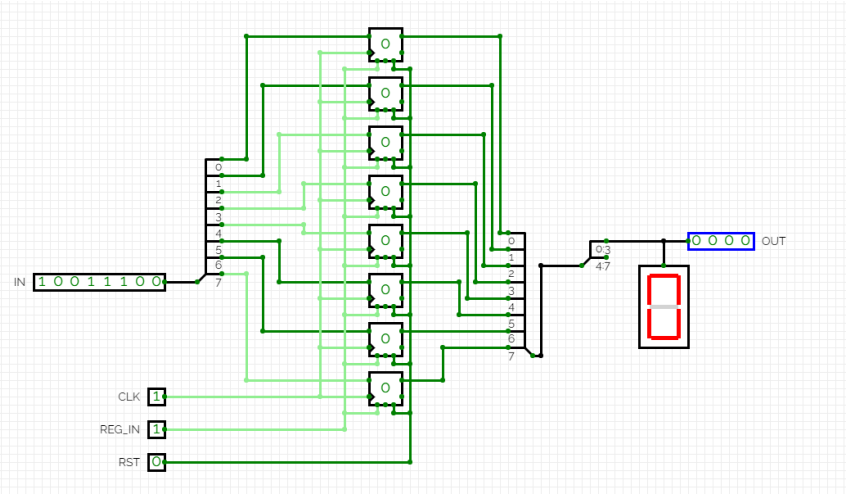
2. Instruction Register
(reg_IR)



- The instruction register is a part of the control unit.
- To fetch an instruction, the computer does a memory read operation - MEM-OUT
- This places the contents of the addressed memory location on the bus - IR_IN

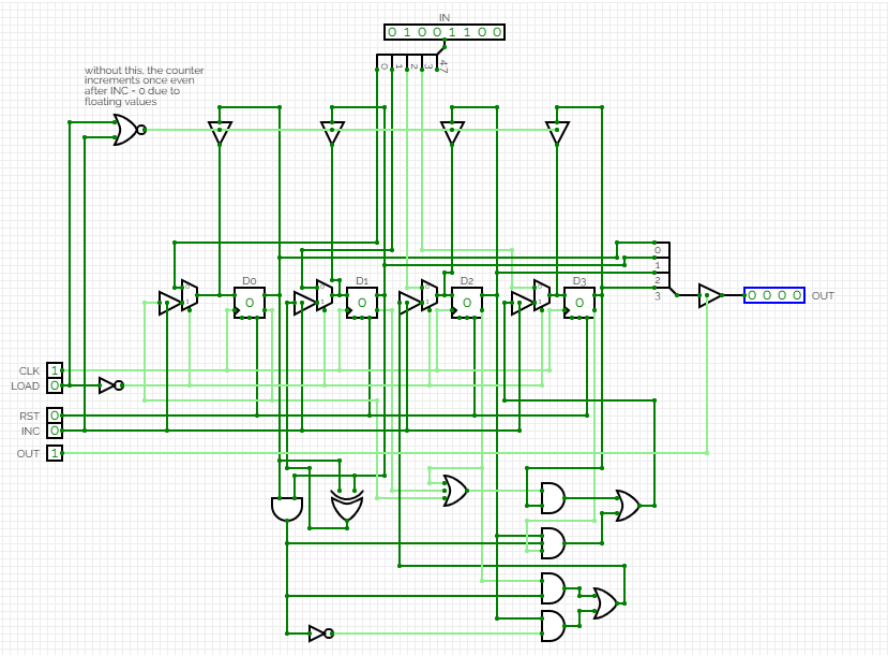
The design of the instruction register is similar to that of a general CPU register.
However, the input and output lines are different as the output of the instruction register goes to the instruction decoder and not directly to the bus.

3. Memory Address Register (reg_MAR)



- As the name suggests, this register stores the 4 - bit address for the corresponding location in the memory.
- The least significant 4 bits correspond to the address.
- The MAR output is directly connected to the address input of the memory.

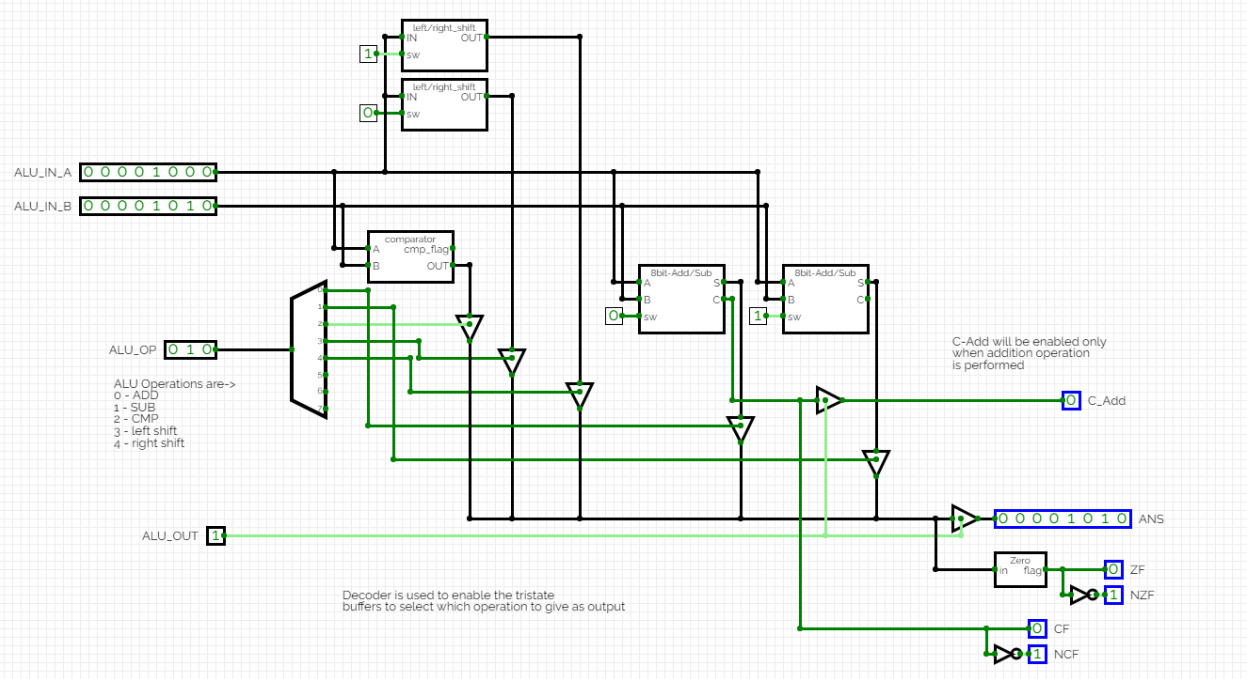
4. Program Counter (counter_PC)



- The program is stored at the beginning of memory with the first instruction at 0000, second at 0001 and so on.
- The program counter, which is a part of the control unit, counts from 0000 to 1111.
- It sends the memory address of the next instruction to be fetched and executed.
- It is essentially a pointer which points to the address of the instruction to be executed.

The program counter is reset to 0000 before each computer run (by SYS_RST)
When the computer is run, the program counter sends address 0000 to the memory address register. (PC_OUT and MAR_IN)
The program counter is then incremented to 0001 (PC_INC) simultaneously as the first instruction is read from memory into the instruction register.
This constitutes the fetch cycle.
The first instruction is then executed.
Then address 0001 is sent to the memory address register followed by an increment to 0010 and so on. The second instruction is then executed.
The program counter is thus keeping track of the next instruction to be fetched and executed.

5. **Arithmetic and Logical Unit** (Gajendra_ALU)



ALU is composed of subcircuits for addition, subtraction, left shift, right shift and comparison. It uses a decoder to select one of these 5 operations from the 3 bit ALU_OP.

Zero flag - 1 if answer is 0 by the operation performed in the ALU

Carry flag - 1 if carry is 1 on adding the two inputs A and B

The operations are mapped to the decoder as follows:

0 - ADD

1 - SUB

2 - CMP (Note: outputs A if A>=B, else outputs B)

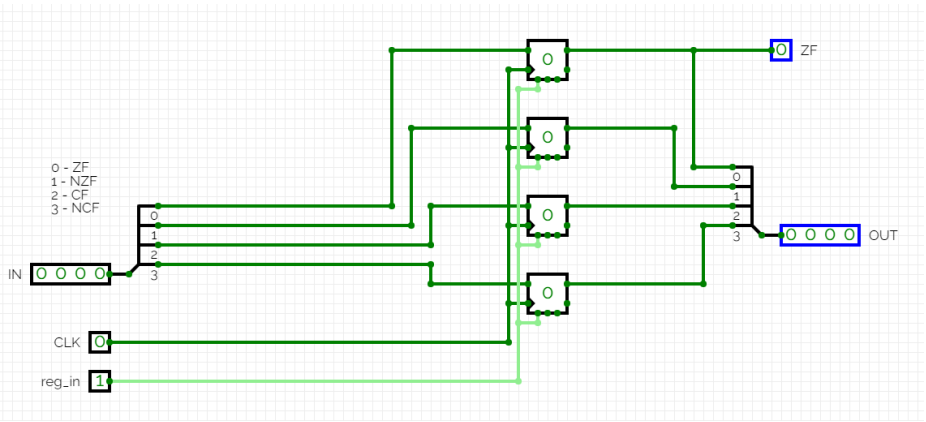
3 - left shift

4 - right shift

6. **Status Register** (reg_status_4)

The status register stores the values of the status flags in the ALU, namely -

a. Zero Flag
b. Not Zero Flag
c. Carry Flag
d. Not Carry Flag



- The input of the status register are the 4 flag outputs from the ALU
- The status registers takes input only when ALU gives output.
- This is ensured by connecting the reg_in of the status register to the ALU_OUT of the ALU.

INSTRUCTION OP CODES

NAME	MSBs	LSBs
NOP	0000	XXXX
LDA	0001	ADDR
STA	0010	ADDR
ADD	0011	ADDR
SUB	0100	ADDR
LDI	0101	DATA
COUT	0110	XXXX
JMP	0111	ADDR
JNZ	1000	ADDR
SWAP	1001	XXXX
CMP	1010	ADDR
LSA	1011	XXXX
RSA	1100	XXXX
MOVAB	1101	XXXX
MOVBC	1110	XXXX
HALT	1111	XXXX

General execution
<previous instruction>

1010 000 <control word> fetch 1
1010 001 <control word> fetch 2
1010 010 <control word> actual stuff 1
1010 011 <control word> actual stuff 2
1010 100 <control word> actual stuff 3
1010 101 <control word> zeros
1010 110 <control word> zeros
1010 111 <control word> zeros

<next instruction>

Instruction Set for Gajendra-I

R_A – Register A
R_B – Register B
R_C – Register C
K – Data or address
(K) – Content of memory at address K

1. **NOP** (No operation)

Description: Does not perform any operation. Only fetches the instruction from memory and increments the program counter by 1.		
Operation: No		
Syntax:	**Operands:**	**Program Counter:**
NOP	None	PC ← PC + 1
8 bit op code:		
0000	0000	

2. **LDA**

Description: Loads the contents of the addressed memory location into register A.		
Operation: R_A ← (K)		
Syntax:	**Operands:**	**Program Counter:**
LDA K	8 <= K <= 15	PC ← PC + 1
8 bit op code:		
0001	kkkk	

3. **STA**

Description: Loads the contents of register A to the addressed memory location.		
Operation: (K)←R_A		
Syntax:	**Operands:**	**Program Counter:**
STA K	8<=K<=15	PC ← PC + 1
8 bit op code:		
0010	kkkk	

(we are using ROM in our implementation, but we have tested STA using a tri-state buffer enabled by mem-in)-(in eeprom we were getting a simulation stack exceeded error)

4. **ADD**

Description: Adds the contents of the register A with the contents of the addressed memory location and stores the result in register A		
Operation: R_B ← (K), R_A ← R_B+R_A		
Syntax:	**Operands:**	**Program Counter:**
ADD K	8<=K<=15	PC ← PC + 1
8 bit op code:		
0011	kkkk	

5. **SUB**

Description: Subtracts the contents of the register A with the contents of the addressed memory location and stores the result in register A		
Operation: R_B ← (K), R_A ← R_A-R_B		
Syntax:	**Operands:**	**Program Counter:**
SUB K	8<=K<=15	PC ← PC + 1
8 bit op code:		
0100	kkkk	

6. LDI

Description: Loads the data given as argument to the instruction to register A		
Operation: $R_A \leftarrow K$		
Syntax:	**Operands:**	**Program Counter:**
LDI K	0<=K<=15	PC ← PC + 1
8 bit op code:		
0101	kkkk	

7. COUT

Description: Outputs the content of register A by first transferring it to register C and displays it on the hex display		
Operation: $R_C \leftarrow R_A$, Display$\leftarrow R_C$		
Syntax:	**Operands:**	**Program Counter:**
COUT	None	PC ← PC + 1
8 bit op code:		
0110	xxxx	

8. JMP

Description: Unconditional jump to the address mentioned in the instruction. Sets program counter to address of instruction		
Operation: None		
Syntax:	**Operands:**	**Program Counter:**
JMP K	0<=K<=7 (8 memory locations for program and 8 for data but can be varied depending on length of program)	PC ← K
8 bit op code:		
0111	kkkk	

9. JNZ(Jump Not Zero)

Description: Conditional jump to the address mentioned in the instruction. Sets program counter to address of instruction if zero flag is not set.		
Operation: None		
Syntax:	**Operands:**	**Program Counter:**
JNZ K	0<=K<=7 (same as JMP)	PC ← K (if zero flag not set) PC←PC+1(if zero flag set)
8 bit op code:		
1000	kkkk	

10. SWAP

Description: Swaps the contents of register A with register C.		
Operation: $R_B \leftarrow R_A$ $R_A \leftarrow R_C$ $R_c \leftarrow R_B$		
Syntax:	**Operands:**	**Program Counter:**
SWAP	None	PC ← PC + 1
8 bit op code:		
1001	xxxx	

11. CMP

Description: Stores the greater value among the contents of register A and contents of address in register A		
Operation: $R_A \leftarrow K$ (if $R_A < K$) $R_A \leftarrow R_A$ (if $R_A >= K$)		
Syntax:	**Operands:**	**Program Counter:**
CMP K	8<=K<=15	PC ← PC + 1
8 bit op code:		
1010	kkkk	

12. LSA

Description: Left shifts the contents of register A by one bit		
Operation: Logical left shift of 8 bit contents of register A		
Syntax:	**Operands:**	**Program Counter:**
LSA	None	PC ← PC + 1
8 bit op code:		
1011	xxxx	

13. RSA

Description: Right shifts the contents of register A by one bit		
Operation: Logical right shift of 8 bit contents of register A		
Syntax:	**Operands:**	**Program Counter:**
RSA	None	PC ← PC + 1
8 bit op code:		
1100	xxxx	

14. MOVAB

Description: Does not perform any operation. Only fetches the instruction from memory and increments the program counter by 1.		
Operation: $R_B \leftarrow R_A$		
Syntax:	**Operands:**	**Program Counter:**
MOVAB	None	PC ← PC + 1
8 bit op code:		
1101	xxxx	

15. MOVBC

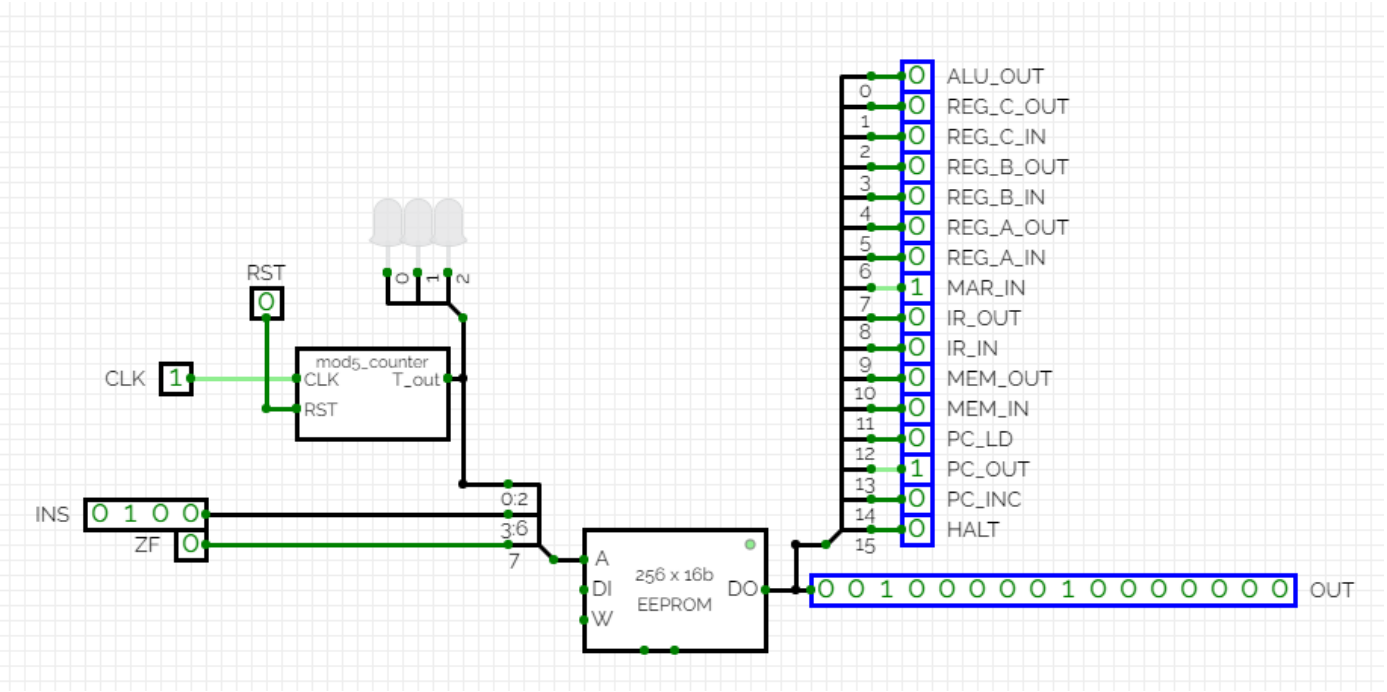
Description: Does not perform any operation. Only fetches the instruction from memory and increments the program counter by 1.		
Operation: $R_C \leftarrow R_B$		
Syntax:	**Operands:**	**Program Counter:**
MOVBC	None	PC ← PC + 1
8 bit op code:		
1110	xxxx	

16. HALT

Description: Stops the program.(stops the clock)		
Operation: No		
Syntax:	**Operands:**	**Program Counter:**
HALT	None	PC ← PC + 1
8 bit op code:		
1111	xxxx	

Control Logic and Microinstructions

Software Controller



16 bit Control words-
MSB to LSB bit representation.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Halt	PC	PC	PC	MEM	MEM	IR	IR	MAR_IN	REG_A	REG_A	REG_B	REG_B	REG_C	REG_C	ALU
	INC	IN	OUT	IN	OUT	IN	OUT		IN	OUT	IN	OUT	IN	OUT	OUT

Control words stored in the software controller.
Each instruction has been padded to 8 microinsructions.
Each instruction's T state's control word has to be accessed in memory based on the address given by the instruction along with zero flag and T state.
Each new instruction starts at 8*n where n is an integer>=0.
We have loaded 16 instructions followed by NOPs.

```
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0180,0x0440,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0180,0x0820,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0180,0x0410,0x0041,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0180,0x0410,0x0041,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0140,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0024,0x0002,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x1100,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x1100,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0030,0x0042,0x000C,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0180,0x0410,0x0041,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0100,0x0005,0x0042,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0100,0x0005,0x0042,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0030,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x000C,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x8000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x2080,0x4600,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
```

NOTE:
We have also implemented using FSM, details and pictures of which are at the end of the document.

Instruction	Control word	T states
NOP:(0000)		
2 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	0	T3
	0	T4
	0	T5
LDA:(0001)		
4 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<IR_OUT | 1<<MAR_IN	T3
	1<<MEM_OUT | 1<<REGA_IN	T4
	0	T5
STA:(0010)		
4 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<IR_OUT | 1<<MAR_IN	T3
	1<<MEM_IN | 1<<REGA_OUT	T4
	0	T5
ADD:(0011)		
5 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<IR_OUT | 1<<MAR_IN	T3
	1<<MEM_OUT | 1<<REGB_IN	T4
	1<<REGA_IN | 1<<ALU_OUT	T5

Instruction	Control word	T states
SUB:(0100)		
5 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<IR_OUT | 1<<MAR_IN	T3
	1<<MEM_OUT | 1<<REGB_IN	T4
	1<<REGA_IN | 1<<ALU_OUT	T5
LDI:(0101)		
3 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<IR_OUT | 1<<REGA_IN	T3
	0	T4
	0	T5
COUT:(0110)		
4 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<REGC_IN|1<<REGA_OUT	T3
	1<<REGC_OUT	T4
	0	T5
JMP:(0111)		
3 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<IR_OUT | 1<<PC_LOAD	T3
	0	T4
	0	T5

Instruction	Control word	T states
JNZ:(1000)		
3 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<IR_OUT | 1<<PC_LOAD	T3
	0	T4
	0	T5
SWAP:(1001)		
5 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<REGB_IN | 1<<REGA_OUT	T3
	1<<REGA_IN | 1<<REGB_OUT	T4
	1<<REGC_IN | 1<<REGB_OUT	T5
CMP:(1010)		
5 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<IR_OUT | 1<<MAR_IN	T3
	1<<MEM_OUT | 1<<REGB_IN	T4
	1<<REGA_IN | 1<<ALU_OUT	T5
LSA:(1011)		
5 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<IR_OUT	T3
	1<<ALU_OUT | 1<<REGC_IN	T4
	1<<REGA_IN | 1<<REGC_OUT	T5

Instruction	Control word	T states
RSA:(1100)		
5 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<IR_OUT	T3
	1<<ALU_OUT | 1<<REGC_IN	T4
	1<<REGA_IN | 1<<REGC_OUT	T5
MOVAB(1101)		
3 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<REGA_OUT | 1<<REGB_IN	T3
	0	T4
	0	T5
MOVBC(1110)		
3 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<REGC_IN | 1<<REGB_OUT	T3
	0	T4
	0	T5
HALT(1111)		
3 T states	1<<PC_OUT | 1<<MAR_IN	T1
	1<<PC_INC | 1<<MEM_OUT | 1<<IR_IN	T2
	1<<HALT	T3
	0	T4
	0	T5

Programs

A. Add 2 numbers and display the sum

	Instruction	Address	
a.	LDA	0xe	→ 1e
b.	ADD	0xf	→ 3f
c.	COUT	0x0	→ 60

B. Adding and subtracting 4 numbers in some combination

a.	LDA	0xc	→ 1c
b.	ADD	0xd	→ 3d
c.	SUB	0xe	→ 4e
d.	ADD	0xf	→ 3f
e.	COUT	0x0	→ 60

C. Multiplication using repeated addition

a.	LDA	0xb	→ 1b	// store 00 at memory address b(initialising sum)
b.	MOVAB	0x0	→ d0	
c.	LDA	0xd	→ 1d	// number of times to be added at mem. address d
d.	SWAP	0x0	→ 90	
e.	ADD	0xc	→ 3c	// number to be added at mem. address c
f.	SWAP	0x0	→ 90	
g.	SUB	0xe	→ 4e	// store 01 at memory address e
h.	SWAP	0x0	→ 90	
i.	JNZ	0x5	→ 85	
j.	COUT	0x0	→ 60	

D. Left shift A and display

a.	LDA	0xb	→ 1b	// loads register A with contents at b
b.	LSA	0x0	→ b0	//Left shifts contents of register A
c.	COUT	0x0	→ 60	

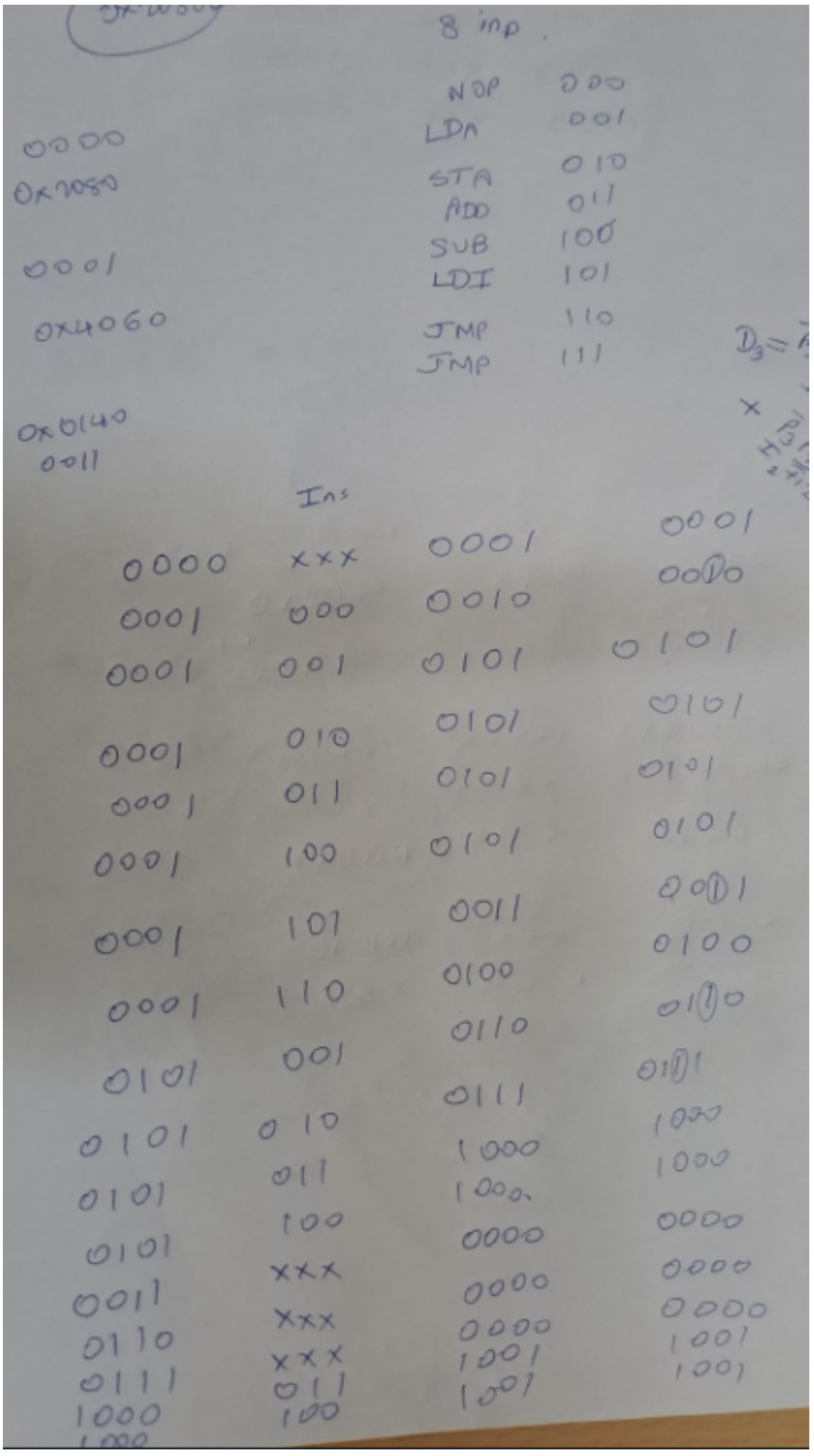
E. Compare contents of 2 memory locations and output the larger one

a.	LDA	0xb	→ 1b	// loads register A with contents at b
b.	CMP	0xc	→ ac	//compare register a components and address at c and stores the larger in register a.
c.	COUT	0x0	→ 60	

FSM controller logic working-

Taken reference from slides and used the states naming convention from the slides.

The controller supports NOP,LDA,STA,ADD,SUB,LDI,JMP,JNZ(will require external logic)



D_3

$\overline{P_3}P_2\overline{P_1}P_0\overline{I_2}I_1I_0$

$+ \overline{P_3}P_2P_1P_0I_2\overline{I_1}\overline{I_0}$

$+ P_3\overline{P_2}\overline{P_1}\overline{P_0}I_2\overline{I_1}\overline{I_0}$

$+ P_3\overline{P_2}P_1\overline{P_0}\overline{I_2}I_1I_0$

D_1

$\overline{P_3}\overline{P_2}P_1P_0\overline{I_2}\overline{I_1}\overline{I_0}$

$I_2\overline{I_1}I_0$

$\overline{P_3}P_2\overline{P_1}P_0\overline{I_2}\overline{I_1}I_0$

$\overline{P_3}P_2\overline{P_1}P_0\overline{I_2}I_1\overline{I_0}$

D_2

$\overline{P_3}P_2\overline{P_1}P_0\overline{I_2}\overline{I_1}I_0$

$\overline{P_3}\overline{P_2}\overline{P_1}P_0\overline{I_2}I_1$

$\overline{P_3}\overline{P_2}P_1P_0I_2\overline{I_0}$

$\overline{P_3}P_2\overline{P_1}P_0\overline{I_2}I_1I_0$

$P_3P_2\overline{P_1}P_0\overline{I_2}I_1\overline{I_0}$

D_0

$\overline{P_3}\overline{P_2}\overline{P_1}\overline{P_0}$

$+ P_3\overline{P_2}\overline{P_1}P_0\overline{I_2}I_1$

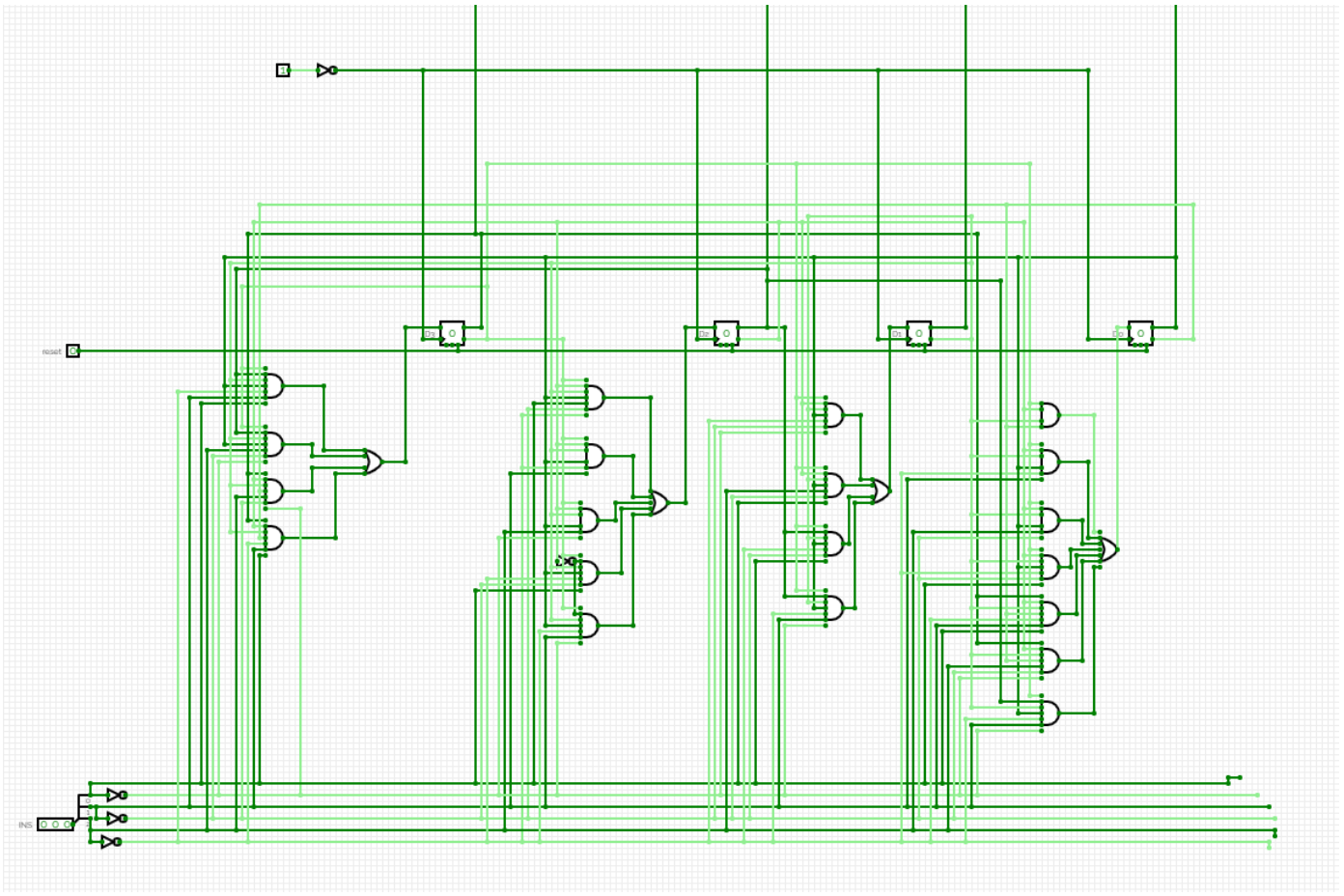
$+ \overline{P_3}\overline{P_2}P_1P_0I_2\overline{I_1}$

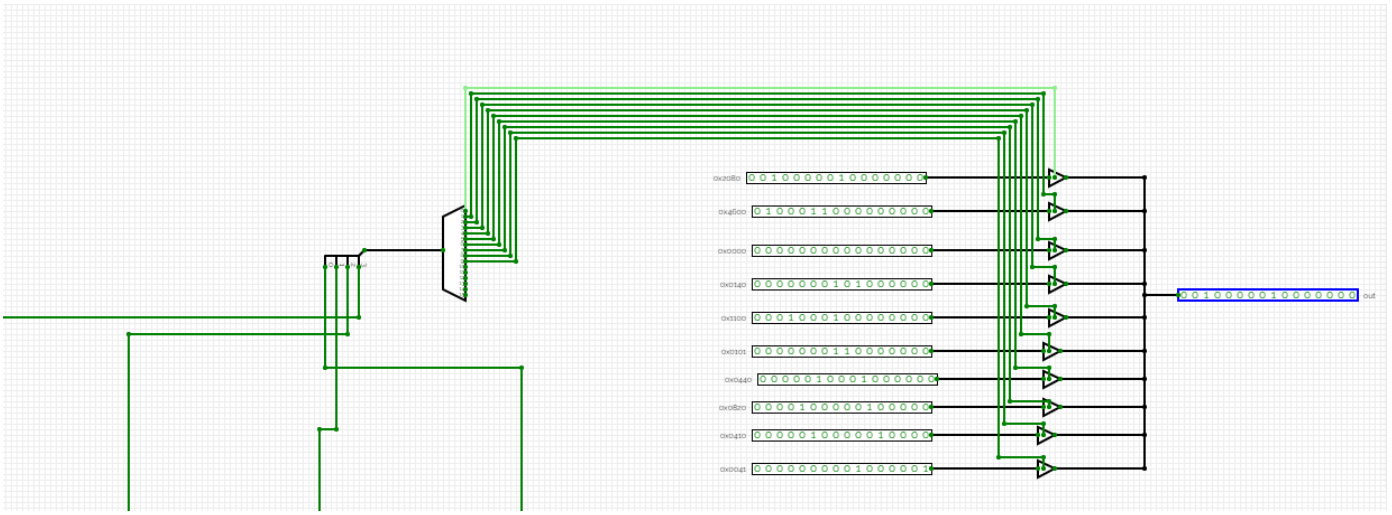
$+ \overline{P_3}\overline{P_2}\overline{P_1}P_0\overline{I_2}\overline{I_1}\overline{I_0}$

$+ P_3\overline{P_2}\overline{P_1}P_0\overline{I_2}I_1I_0$

$+ P_3\overline{P_2}\overline{P_1}\overline{P_0}I_2\overline{I_1}\overline{I_0}$

$+ \overline{P_3}P_2\overline{P_1}P_0\overline{I_2}I_1\overline{I_0}$





FSM is faster because it does not require the additional T-states that were getting wasted in the software controller.
However, it requires more D-flip flops and combinational circuit elements.s

We have tested all our components and run different porgrams to test our circuit.
It was an extremely exciting journey of creating Gajendra-I and we were elated when the programs worked on our homebrew CPU.
Thank you.

END.