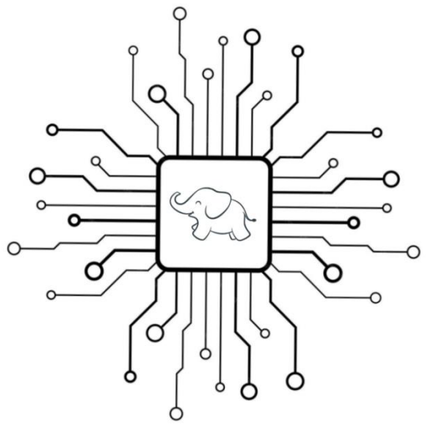
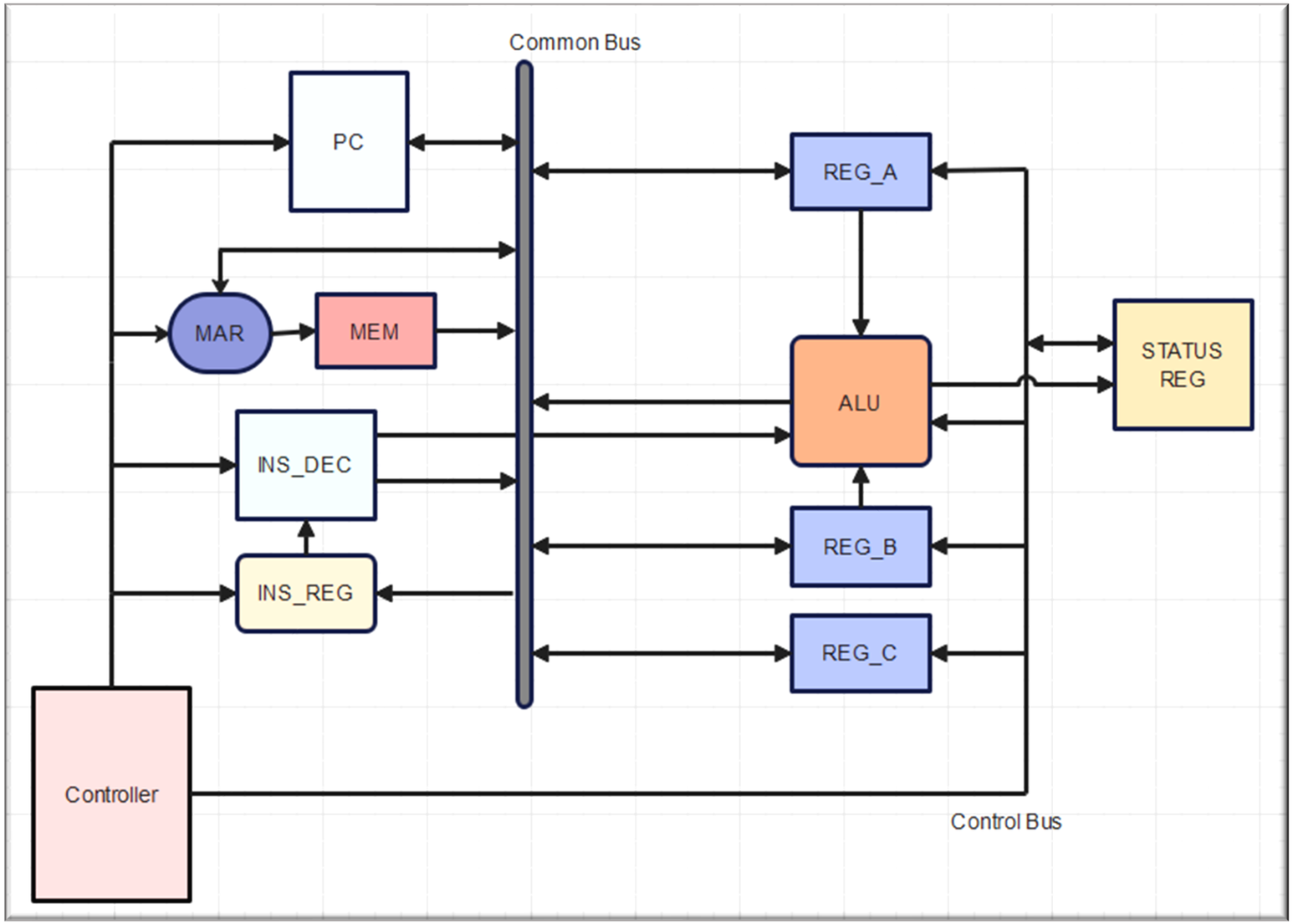


GAJENDRA-1

Contents

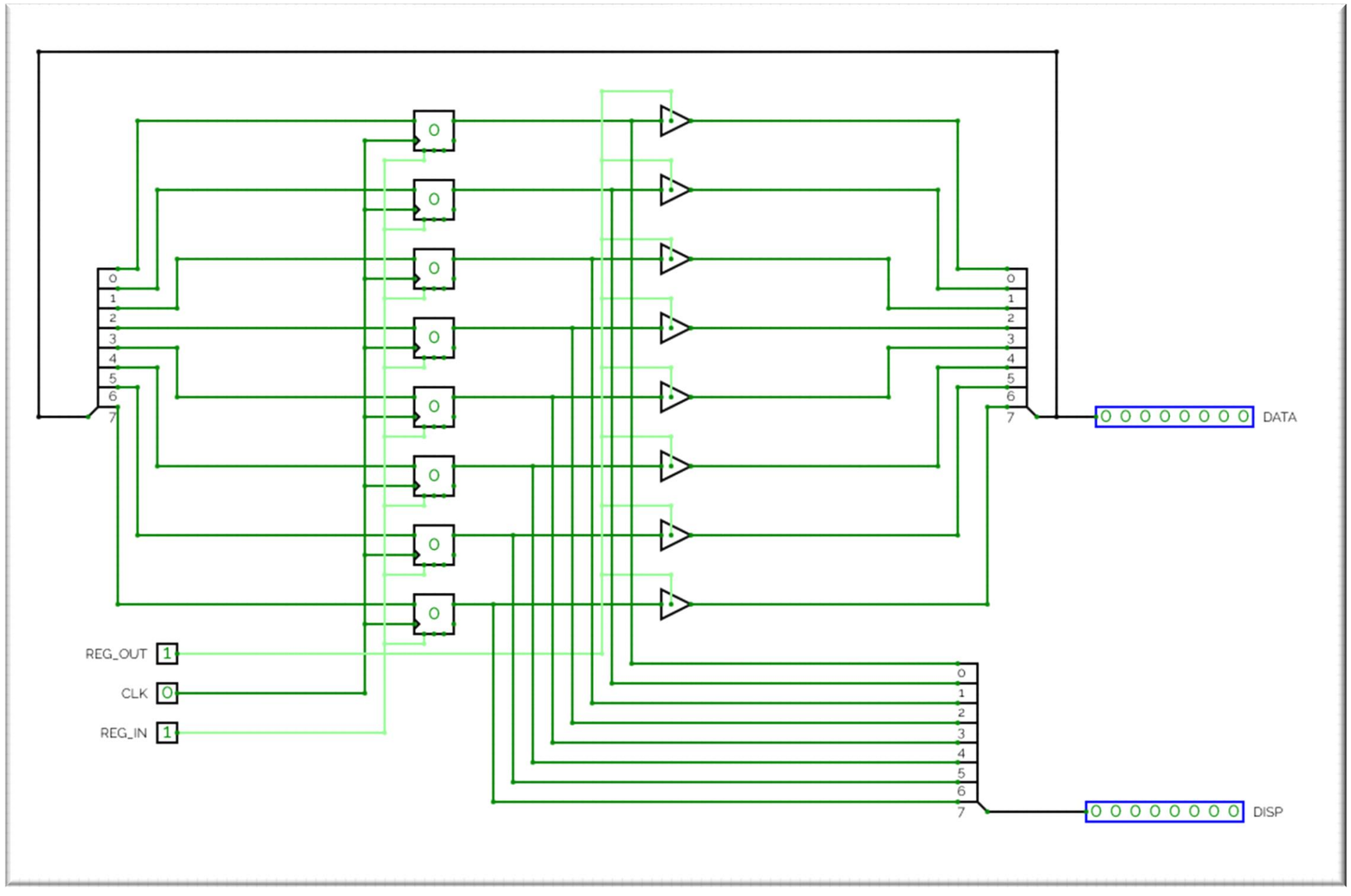
1. ARCHITECTURE OF THE PROCESSOR



- In the design of the processor, various components have been used according to the requirement to run desired programs.
- The individual components used, and their description is in the later part of this Manual.
- As per the requirment of the design, three Registers have been used for storing data while running programs.

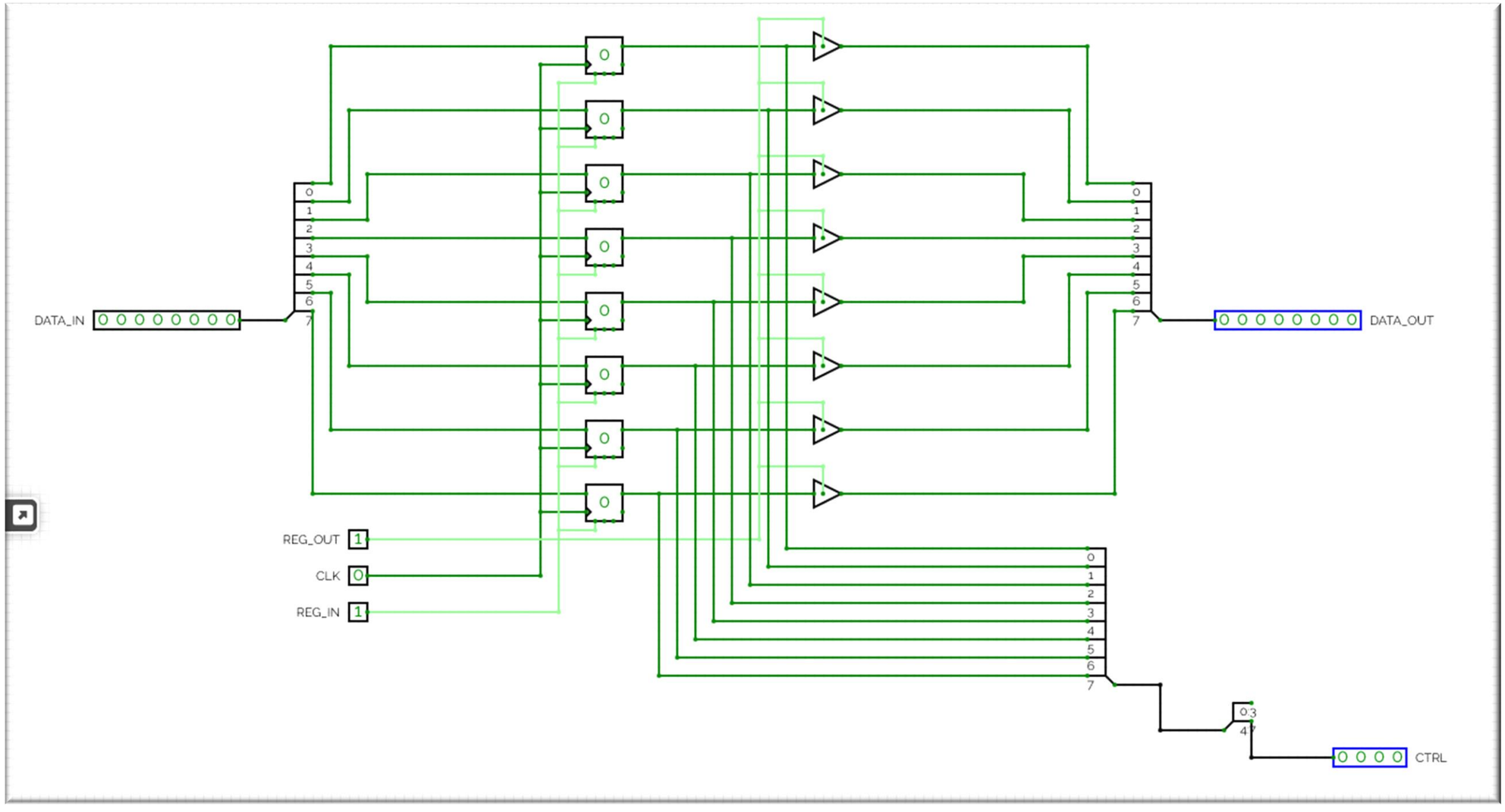
- Using 3 registers makes up the ideal design since the maximum number of registers an operation that can be executed using this processor demands is three.
- To run a particular program, load the program into the ROM and plug it in the circuit.
- The processor runs each instruction step by step starting from the address 0x0 of the ROM and proceeds till it encounters a HALT instruction.
- If the program does not consist of any HALT instruction, the program will be running forever.
- Anyhow, the program can be stopped by activating the **SYS_RESET** switch. Once this switch is activated and deactivated, the program starts from the beginning all over again.
- But when the program encounters a HALT instruction, the program stops at that particular instruction itself. In this case, **New program** button is used.
- This button must be used whenever the program is stopped due to HALT instruction.
- This button brings the control back to the first memery loaction of the ROM and the new program can be executed.

2. GENERAL CPU REGISTERS (reg_cpu_8)



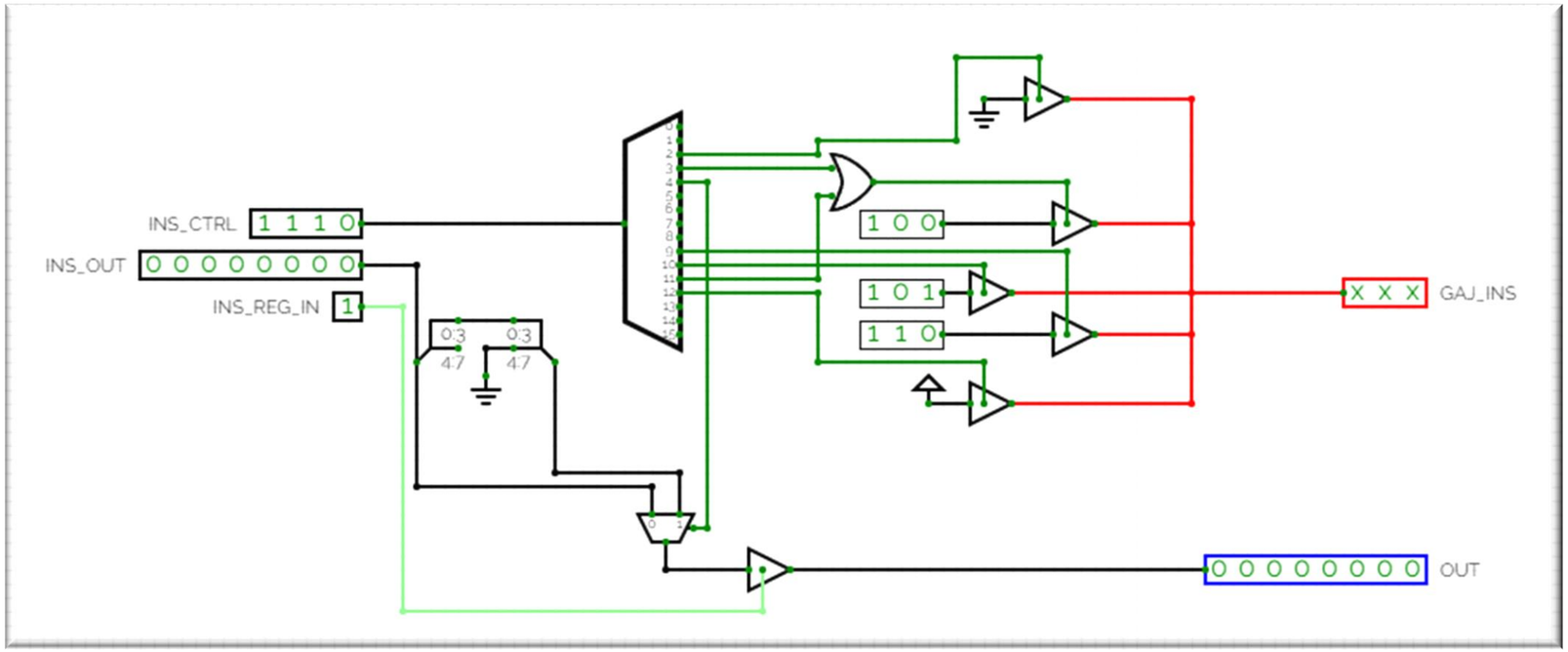
- The register reg_cpu_8 is bidirectional.
- There are three control signals :
  1. REG_IN: When it is active, the register stores the input coming from the DATA. This is implemented using the Enable of each individual D-flipflop.
  2. REG_OUT: When it is active, the register outputs to the DATA.
  3. CLK: It is the clock input to individual D-flipflops used in the circuit.
- There are 3 general purpose registers in the CPU. They are Accumulator(Ra), REG_B and REG_C.
  1. Accumulator is the register which is used to carry the sum or difference over multiple instructions.
  2. REG_B is used to get the input from the memory. This is the only register that communicates with the memory first hand.
  3. REG_C is Output register as well as used for implementing SWAP instruction for Accumulator and REG_B.
- Data can be stored and can be looked up when needed using the control signals.

3. INSTRUCTION REGISTER (reg_IR)



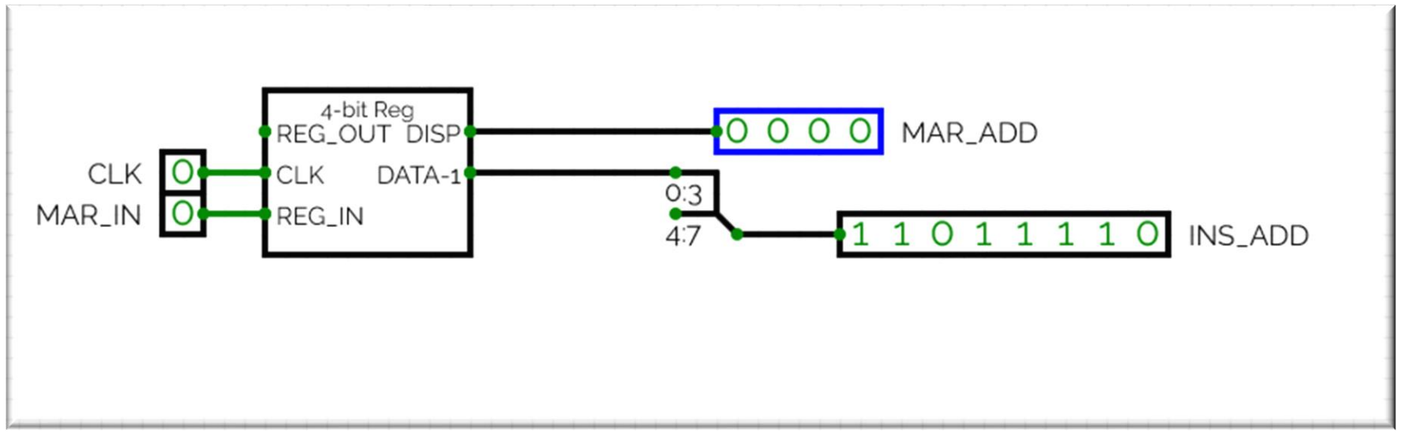
- Instruction register is similar to general purpose registers as they both have the same bit size.
- This is not bidirectional. But the control words are same as described for general purpose registers.
- The instruction register takes input from the common bus in the processor and stores it.
- The instruction register is responsible for sending the 4 MSBs of the opcode to the Controller.
- The instruction register sends the same 4 MSB's to instruction decoder which decodes the operation that is to be done by the ALU.
- It can even send the address for certain instructions to the MAR if needed in certain instruction.
- Like general purpose registers D-flipflops are used for storing the data. The CTRL as shown is an unrestricted output.

4. INSTRUCTION DECODER

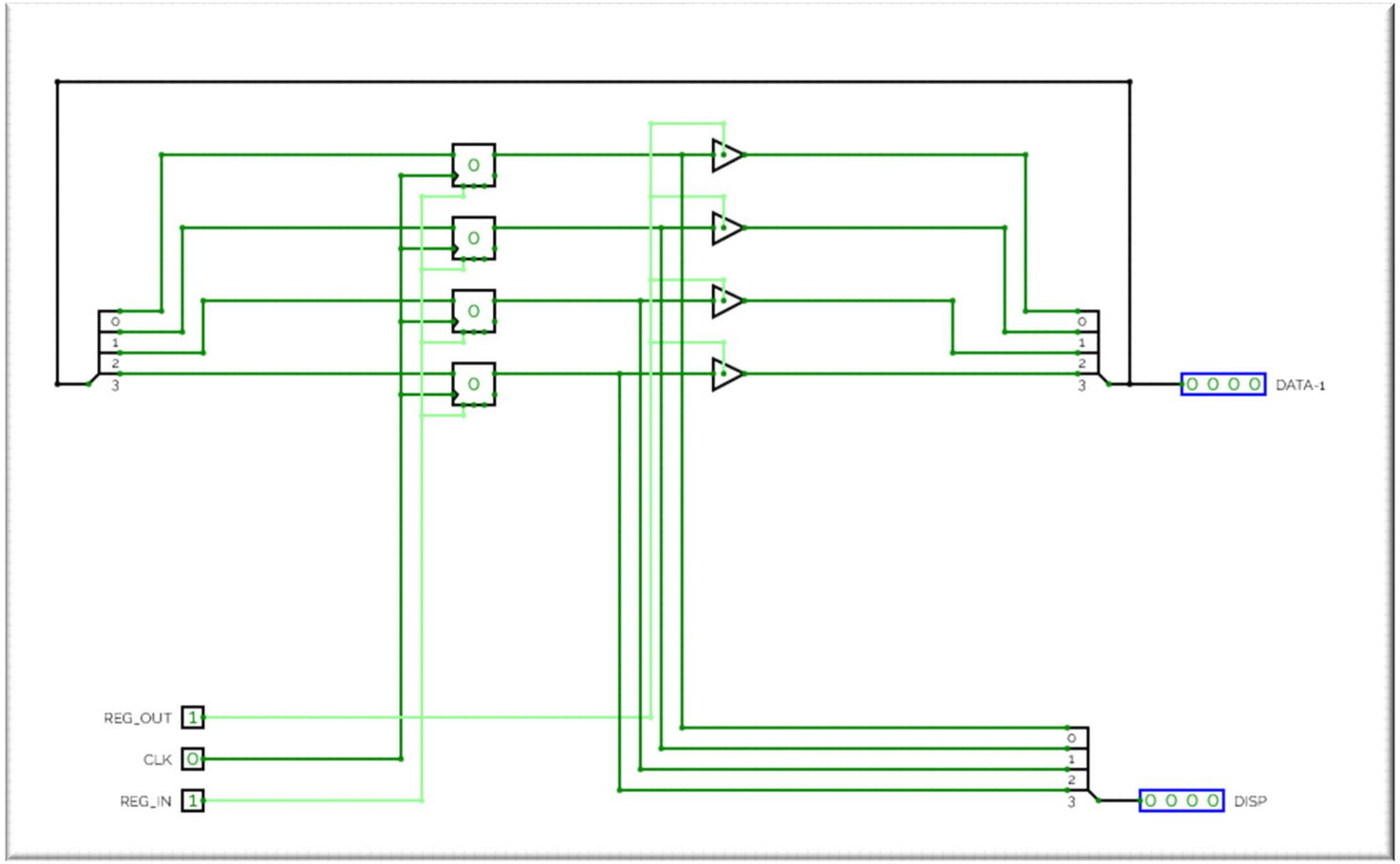


- The Instruction Decoder takes the 4-bit instruction from Instruction Register and based on the instruction given, it sends the relevent control word to the bus.
- If any of the instruction needs ALU to perform a task, then it sends the control word for the the particular action ot ALU.
- And also in the case of executing LDI, as the data is directly loaded to the Accumulator, the most significant 4 bits are stripped off which consist of the instruction machine code.
- The control signal which enables the decoder to pass data to Common bus, is active whenever the Instruction Register gives an output.

5. MEMORY ADDRESS REGISTER (reg_MAR)

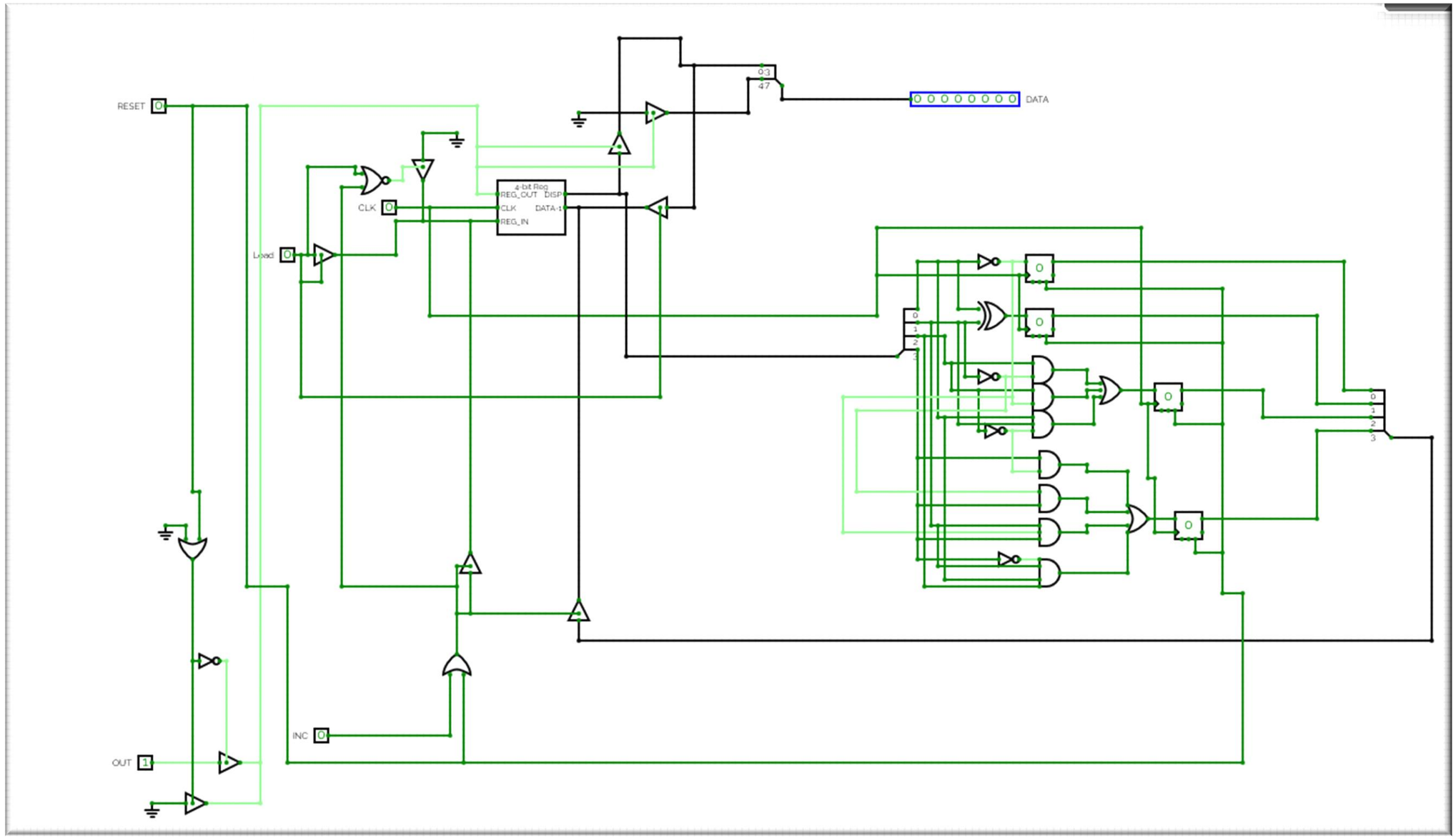


- Memory Address Register is a 4 bit register which stores the address to be sent to the ROM in the processor.
- This register is built using another module which is 4-bit Reg, this module is a bidirectional 4-bit register.
- Here the 4-bit Reg module takes 4 LSBs and stores it when the MAR_IN control is active.
- The MAR_ADD output is always active. This 4-bit output is what points to the correct address in the ROM.
- The existence of this is very important because when program counter sends the address to the ROM, it should simultaneously output it to the control bus. But since 2 components cannot talk to the common bus, we need a register to store the address and this is where MAR comes in.



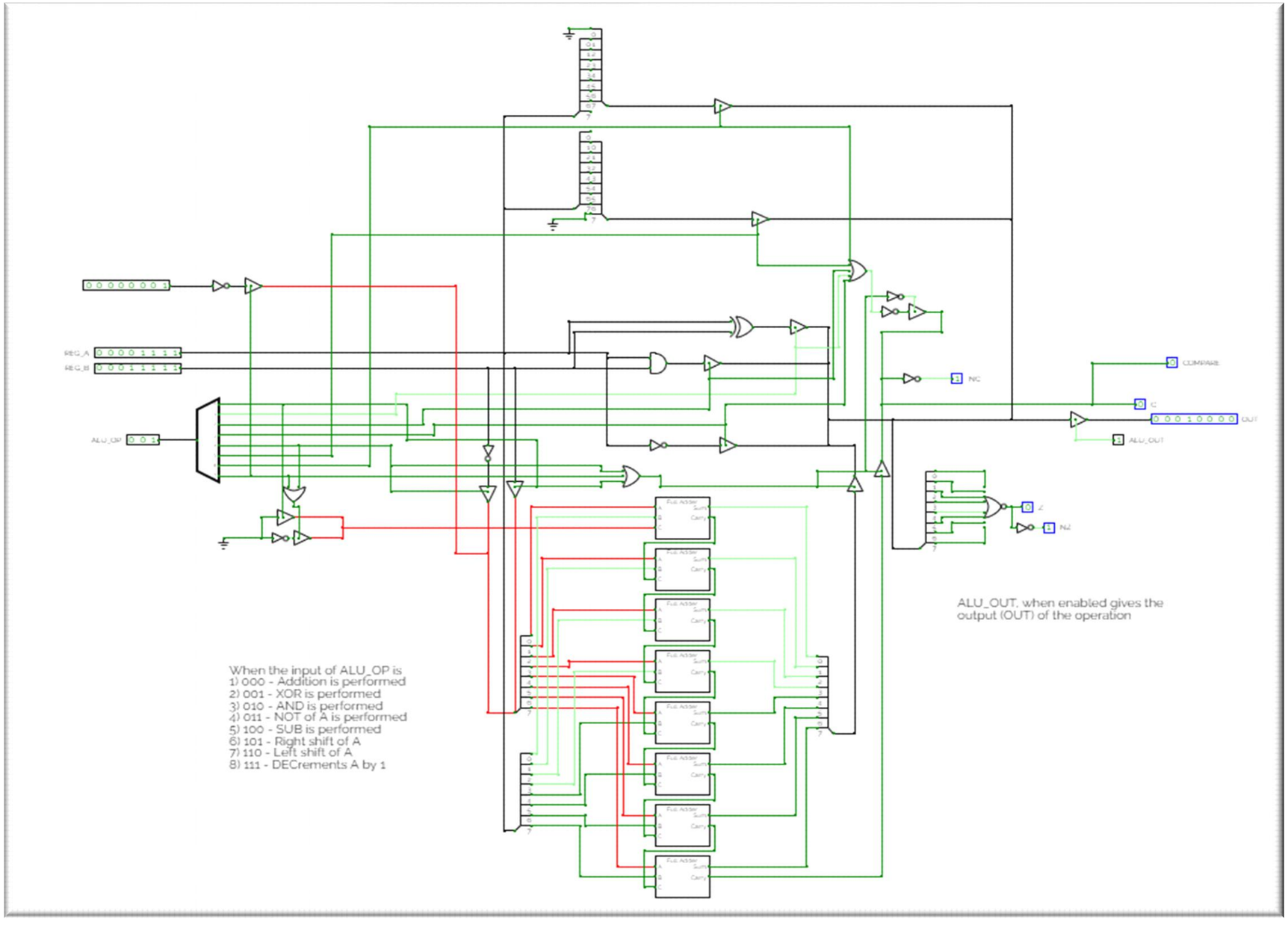
4 – bit Register

6. PROGRAM COUNTER (counter_PC)



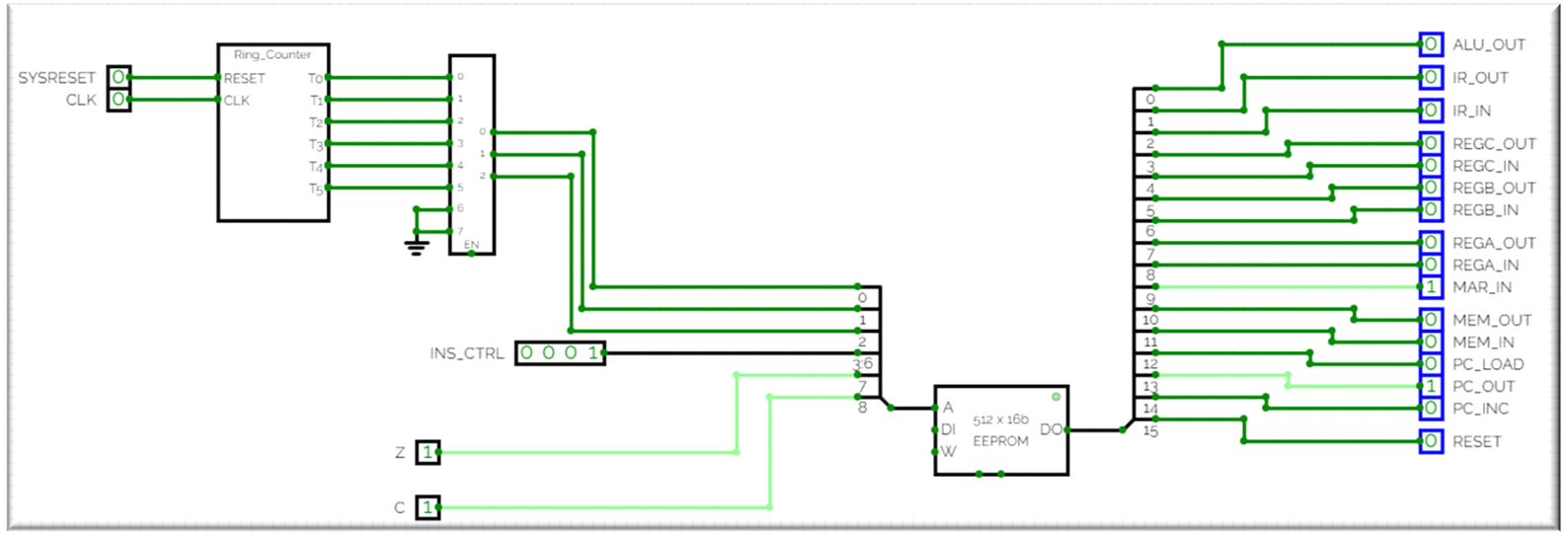
- Program Counter is a 4-bit register. It has the same module as MAR which is 4-bit Reg.
- The function of program counter is to send the address of the instruction that is stored in the ROM.
- Program counter can load an address from the common bus. For this, **LOAD** has to be activated.
- The flipflops and the combinational logic on the right side is for incrementing the value by one and storing it back into the register.
- For sending the output to the common bus, we need to make the **OUT** active. Here, the program counter is Bidirectional.
- There is another control word **RESET** which resets the counter to 0. So, when we want to start the program from the first instruction, we just need to make the **RESET** active.
- To freeze our processor the system reset control signal actually makes the **RESET** active.
- All the control signals in this component are exclusive. We cannot have multiple things active at once.
- At its heart, the program counter is just a counter which points to the next instruction in the program after every instruction cycle.
- The implementation of the Program counter is whenever it gives output the first 4 MSBs are 0.

7. ARITHMETIC AND LOGICAL UNIT (ALU)



- ALU is one of the most important components in any processor. Here in Gajendra-1, ALU can support 8 instructions. They are
- 1) 000 - Addition is performed
- 2) 001 - XOR is performed
- 3) 010 - AND is performed
- 4) 011 - NOT of A is performed
- 5) 100 - SUB is performed
- 6) 101 - Right shift of A
- 7) 110 - Left shift of A
- 8) 111 - Decrements A by 1
- The instruction for this will be sent by instruction decoder when an instruction that requires operation of ALU is needed. For addition, decrements and subtraction, we are using 8 1-bit full adders.
- XOR, AND, NOT are performed by using the respective gates.
- The right shift, left shift are performed by using splitter.
- When an operation is performed in ALU it sets 4 status registers namely C, NC, Z, NZ.
- We are using C and Z flags to send them to the controller for doing conditional instructions.
- There is an instruction named JG, which should be followed after a SUB or CMP instruction so that the C and Z flags are set accordingly and the instruction is carried out.

8. CONTROLLER

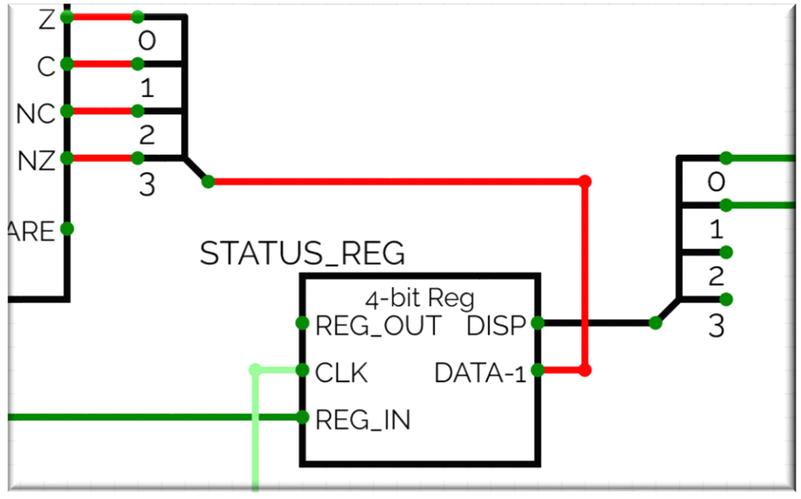


- Controller is the main important component in our Microprocessor. It is the component which controls everything else after receiving an instruction from instruction register.
- This Controller is a software-based controller which uses an EEPROM to store the control words in it. The Control word used for this processor is of size 16 bits.
- The controller stores the control word and the ring counter is a 6-T-state counter, and when an instruction is passed from the instruction decoder, the address of the corresponding is 3 LSBs are state of the ring counter, next four bits are instruction opcode, next MSB is Zero flag from ALU, and the last MSB is Carry flag from the ALU.
- The controller simply makes the corresponding control signal active during the execution of the program.

- The following are the place values of each control signal of the 16 bit control word :-

0	-	alu_out	8	-	rega_in
1	-	ir_out	9	-	mar_in
2	-	ir_in	10	-	mem_out
3	-	regc_out	11	-	mem_in
4	-	regc_in	12	-	pc_load
5	-	regb_out	13	-	pc_out
6	-	regb_in	14	-	pc_inc
7	-	rega_out	15	-	reset

9. STATUS REGISTER (reg_status_4)



- Status Register is implemented using the same 4-bit Reg which was used for Memory Address Register and Program Counter.
- The status register takes 4-bit input from ALU and stores it when the REG_IN is active. The data from the status register is used in controller for doing conditional instructions like JNZ and JG.
- The 4 bits are Z, C, NZ, and NC. Z is set when the output of the ALU is zero, and C is set when there is an addition or subtraction is performed then if there was a carry to the 9^{th} bit then the C flag is set.
- Though our register is bidirectional, it never gives output to the ALU, so the REG_OUT is always zero or never active.

- Status Register is essential in our processor design because the flags that it stores are useful for making conditional instructions, and the utility of these instructions is very high for supporting multiplication of two numbers whose product is less than 256.

General Notations used :

- Ra : Accumulator (aka Register-A)
- Rb : Register - B
- Rc : Register – C
- PC : Program Counter
- Ad : Constant 4-bit Address
- ROM(Ad) : Contents of the ROM at the address ‘Ad’
- K : Constant 4-bit Data

8. INSTRUCTION SET SUMMARY:

MACHINE CODE	MNEMONIC	OPERANDS	DESCRIPTION	OPERATION
0000	NOP	–	No Operation	–
0001	LDA	Ra, Ad	Load accumulator	Ra ← ROM(Ad)
0010	ADD	Ra, Rb, Ad	Add without carry	Rb ← ROM(Ad) Ra ← Ra + Rb
0011	SUB	Ra, Rb, Ad	Subtraction of positive numbers	Rb ← ROM(Ad) Ra ← Ra – Rb
0100	LDI	Ra, K	Load Immediate	Ra ← K
0101	OUT	Rc	Display on Hex Display	Rc ← Ra
0110	JMP	Ad	Jump to address	PC ← ROM(Ad)

0111	JNZ	Ad	Jump when Not Zero	PC ← ROM(Ad)
1000	SWAP	Ra, Rb	Swap the values in Registers	Rc ← Rb Rb ← Ra Ra ← Rc
1001	LSL	Ra	Logical Shift Left	Ra ← Ra << 1
1010	LSR	Ra	Logical Shift Right	Ra ← Ra >> 1
1011	CMP	Ra, Rb	Compare	Ra ← Ra – Rb
1100	DEC	Ra	Decrement	Ra ← Ra – 1
1101	MOV	Ra, Rc	Move the data	Rc ← Ra
1110	JG	Ra, Rb, Ad	Jump when Greater	PC ← ROM(Ad)
1111	HALT	–	Halt	RESET = 1

9. Description of INSTRUCTION SET :

LDA – Load Accumulator

Description

This instruction loads the Accumulator with the contents from the memory by looking up at the 4-bit address provided.

Operation:

1) Ra ← ROM(XXXX)

Syntax:	Operands:	Program Counter:
1) LDA XXXX	X = 0 or X = 1	PC ← PC + 1

8-bit Opcode:

0001	XXXX

ADD – Add without Carry

Description

This instruction loads the Register-B with the contents from the memory by looking up at the 4-bit address provided and then adds up the contents of Accumulator and Register-B. And stores the final sum in Accumulator.

Operation:

1) Rb ← ROM(XXXX)
2) Ra ← Ra + Rb

Syntax:	Operands:	Program Counter:
1) ADD XXXX	X = 0 or X = 1	PC ← PC + 1

8-bit Opcode:

0010	XXXX

SUB – Subtraction of positive numbers

Description

This instruction loads the Register-B with the contents from the memory by looking up at the 4-bit address provided and then subtracts the contents of Register-B from Accumulator. And stores the final difference in Accumulator, provided the difference non-negative.

Operation:

1) Rb ← ROM(XXXX)
2) Ra ← Ra - Rb

Syntax:	Operands:	Program Counter:
1) SUB XXXX	X = 0 or X = 1	PC ← PC + 1

8-bit Opcode:

0011	XXXX

LDI – Load Immediate

Description

Loads the 4-bit constant data directly to the Accumulator.

Operation:

1) Ra ← K

Syntax:	Operands:	Program Counter:
1) LDI XXXX	X = 0 or X = 1	PC ← PC + 1

8-bit Opcode:

0100	XXXX

OUT – Display on Hex display

Description

Copies the contents of Accumulator to Register-C and shows the value on Hex display attached to it.

Operation:

1) Rc ← Ra

Syntax:	Operands:	Program Counter:
1) OUT XXXX	X =0	PC ← PC + 1

8-bit Opcode:

0101	XXXX

JMP – Jump to address

Description

This instruction takes the control to the given memory location.

Operation:

1) ROM address : XXXX

Syntax:	Operands:	Program Counter:
1) JMP XXXX	X = 0 or X = 1	PC ← XXXX

8-bit Opcode:

0110	XXXX

JNZ – Jump when Non-zero

Description

This instruction takes the control to the given memory location when the value returned after performing an operation is non-zero.

Operation:

1) If NZ == 1 : ROM address : XXXX

Syntax:	Operands:	Program Counter:
1) JNZ XXXX	X = 0 or X = 1	If NZ == 1 : PC ← XXXX else PC ← PC + 1

8-bit Opcode:

0111	XXXX

SWAP(AB) – Swap A and B

Description

Swaps the contents present in Accumulator and Register-B by temporarily storing contents of Register-B in Register-C during swapping.

Operation:

1) Rc ← Rb
2) Rb ← Ra
3) Ra ← Rc

Syntax:	Operands:	Program Counter:
1) SWAP XXXX	X = 0	PC ← PC + 1

8-bit Opcode:

1000	XXXX

LSL – Logical Shift Left

Description

This instruction performs the logical left shift to the contents present in the Accumulator. Used to perform multiplication by 2.

Operation:

1) Ra ← Ra << 1

Syntax:	Operands:	Program Counter:
1) LSL XXXX	X = 0	PC ← PC + 1

8-bit Opcode:

1001	XXXX

LSR – Logical Shift Right

Description

This instruction performs the logical right shift to the contents present in the Accumulator. Used to perform division by 2.

Operation:

1) Ra ← Ra >> 1

Syntax:	Operands:	Program Counter:
1) RSL XXXX	X = 0	PC ← PC + 1

8-bit Opcode:

1010	XXXX

CMP – Compare

Description

The contents of the Accumulator are compared with the value present at the given memory location in the memory.

Operation:

1) Rb ← ROM(XXXX)
2) Ra ← Ra – Rb
3) If C == 1 : Value in Ra ≥ Value in Rb

Syntax:	Operands:	Program Counter:
1) CMP XXXX	X = 0	PC ← PC + 1

8-bit Opcode:

1011	XXXX

DEC – Decrement

Description

This instruction reduces the value present in the Accumulator by 1. This is performed by loading Register-B with 1 and subtracting it form contents of Accumulator.

Operation:

1) Rb ← 1
2) Ra ← Ra - Rb

Syntax:	Operands:	Program Counter:
1) DEC XXXX	X = 0	PC ← PC + 1

8-bit Opcode:

1100	XXXX

MOV(CA) – Copy data from C to A

Description

The contents present in Register-C are loaded into the Accumulator.

Operation:

1) Ra ← Rc

Syntax:	Operands:	Program Counter:
1) MOV XXXX	X = 0	PC ← PC + 1

8-bit Opcode:

1101	XXXX

JG – Jump when Greater

Description

This instruction takes the control to the given memory location if the value stored in the Accumulator is greater than the value stored in Register-B.

Operation:

1) If Ra > Rb : ROM Address : XXXX

Syntax:

1) JG XXXX

Operands:

X = 0 or X = 1

Program Counter:

If Z = 0 and C = 1 : PC ← XXXX
else PC ← PC + 1

8-bit Opcode:

1110	XXXX

HALT – Halt the program

Description

This instruction terminates the program by freezing the program counter. This is done by resetting the program counter. The control is taken back to the first memory location.

Operation:

1) RESET = 1

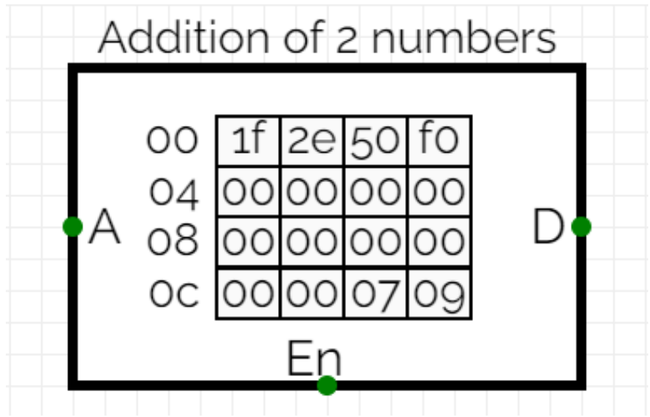
Syntax:	Operands:	Program Counter:
1) HALT XXXX	X = 0	—

8-bit Opcode:

1111	XXXX

10. Example Assembly programs implemented using the Instruction set.

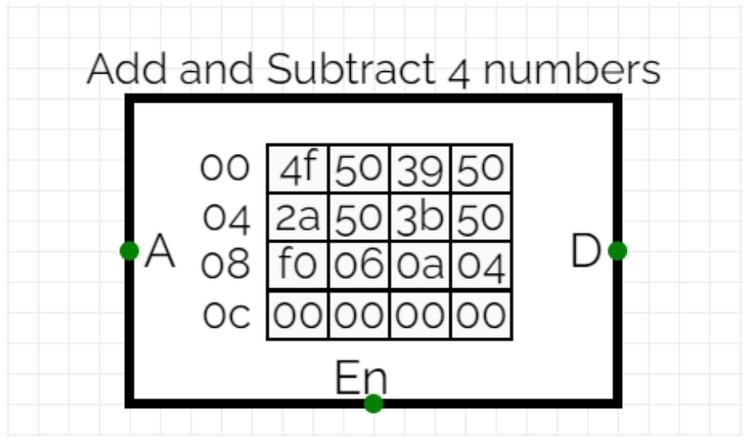
a) Adding two numbers and displaying the result.



- The program shown in the ROM implements the addition operation.
- The content of ROM at the address 0xf is loaded to Accumulator and the content at the address 0xe is loaded to Register-B and then the ALU performs addition.
- Finally, the sum is loaded to Register-C and displayed on Hex Display. And the programs halts.

```
0x0  LDA  0xf
0x1  ADD  0xe
0x2  OUT  0x0
0x3  HALT 0x0
0x4  0x0
0x5  0x0
0x6  0x0
0x7  0x0
0x8  0x0
0x9  0x0
0xa  0x0
0xb  0x0
0xc  0x0
0xd  0x0
0xe  0x7
0xf  0x9
```

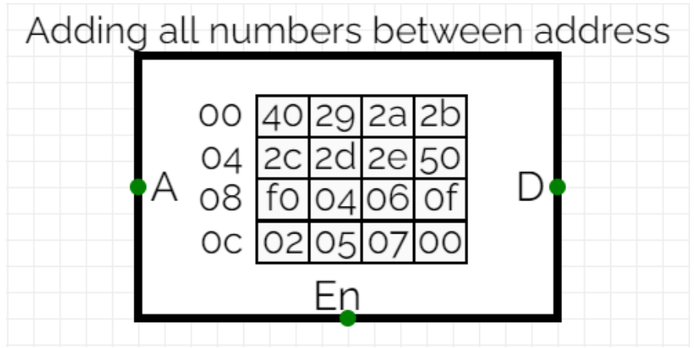
b) Adding and subtracting four numbers in some combination.



- The program shown in the ROM implements the addition and subtraction operation.
- First, the value 0xf is loaded directly to accumulator and the data at the address 0x9 is subtracted from it.
- Then the data at address 0xa is added and finally the data at 0xb is subtracted from the result.
- After every operation, the result is being displayed on Hex Display.
- After the final operation, the program halts.

```
0x0  LDI   0xf
0x1  OUT   0x0
0x2  SUB   0x9
0x3  OUT   0x0
0x4  ADD   0xa
0x5  OUT   0x0
0x6  SUB   0xb
0x7  OUT   0x0
0x8  HALT  0x0
0x9  0x6
0xa  0xa
0xb  0x4
0xc  0x0
0xd  0x0
0xe  0x0
0xf  0x0
```

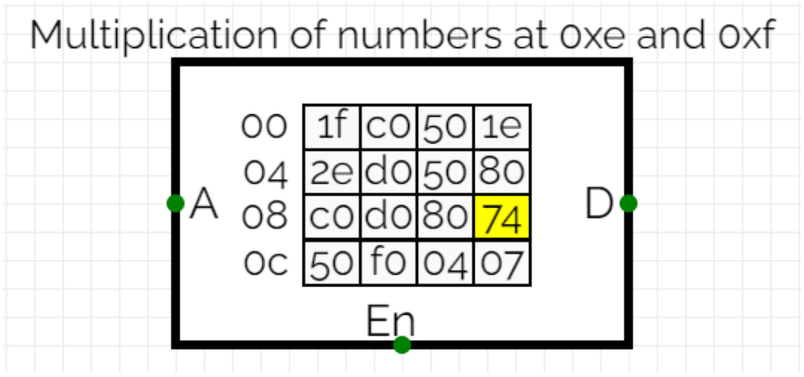
c) Adding numbers from a starting address to an ending address and displaying the result.



- The program shown in the ROM finds the sum of the numbers stored in the ROM from address 0x9 to 0xe
- First, the value 0x0 is loaded directly to accumulator and the numbers at the address location from 0x9 to 0xe are loaded to Register-B one by one and added.
- The final sum is loaded to Register-C and displayed on the Hex Display.
- The program halts after showing the final sum.

0x0	LDI	0x0
0x1	ADD	0x9
0x2	ADD	0xa
0x3	ADD	0xb
0x4	ADD	0xc
0x5	ADD	0xd
0x6	ADD	0xe
0x7	OUT	0x0
0x8	HALT	0x0
0x9	0x4	
0xa	0x6	
0xb	0xf	
0xc	0x2	
0xd	0x5	
0xe	0x7	
0xf	0x0	

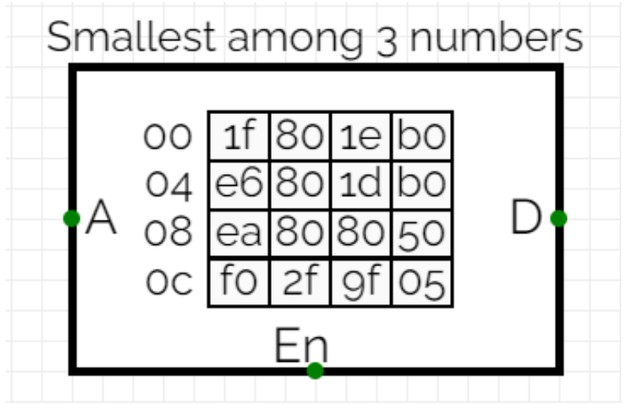
d) Multiplication of two numbers using repeated addition.



- The program shown in the ROM multiplies the numbers stored in the ROM at the address 0xe and 0xf.
- This is done by adding the number at location 0xe, the number at 0xf times.
- At the start of program, the data at address 0xf is loaded to Accumulator and decremented by one and then stored in Register-C.
- Every time the number at 0xe is added, the decremented value stored in Register-C is loaded to Accumulator and decremented. During this process, sum is stored in Register-C.
- When this value reaches zero, the programs halt after showing the final result on Hex Display.

```
0x0  LDA   0xf
0x1  DEC   0x0
0x2  OUT   0x9
0x3  LDA   0xe
0x4  ADD   0xe
0x5  MOV   0x0
0x6  OUT   0xb
0x7  SWAP  0x0
0x8  DEC   0x0
0x9  MOV   0x0
0xa  SWAP  0x0
0xb  JNZ   0x4
0xc  OUT   0x0
0xd  HALT  0x0
0xe  0x4
0xf  0x7
```

e) Finding the smallest among three given numbers.



- The program shown in the ROM compares the 3 numbers at address 0xd, 0xe, 0xf and displays the smallest among them.
- First, the Accumulator loads data at 0xf and compares it with the data at address 0xe.
- After comparison, the smaller one is retained in the Register-B and the data at address oxd is load to Accumulator.
- And the comparison takes place again and the smaller number is stored in Accumulator.
- Finally, the value in Accumulator is moved to Register-C and displayed on Hex Display. And the program halts.

```
0x0  LDA   0xf
0x1  SWAP  0x0
0x2  LDA   0xe
0x3  CMP   0x0
0x4  JG    0x6
0x5  SWAP  0x0
0x6  LDA   0xd
0x7  CMP   0x0
0x8  JG    0xa
0x9  SWAP  0x0
0xa  SWAP  0x0
0xb  OUT   0x4
0xc  HALT  0x0
0xd  0x2f
0xe  0x9f
0xf  0x5
```

II. Sequence of Micro instructions for Instructions:

NOP :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   0
T3 :   0
T4 :   0
T5 :   0
```

LDA :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   1 << ir_out | 1 << mar_in
T3 :   1 << mem_out | 1 << rega_in
T4 :   0
T5 :   0
```

ADD :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   1 << ir_out | 1 << mar_in
T3 :   1 << mem_out | 1 << regb_in
T4 :   1 << alu_out | 1 << rega_in
T5 :   0
```

SUB :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   1 << ir_out | 1 << mar_in
T3 :   1 << mem_out | 1 << regb_in
T4 :   1 << alu_out | 1 << rega_in
T5 :   0
```

LDI :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   1 << ir_out | 1 << rega_in
T3 :   0
T4 :   0
T5 :   0
```

OUT :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   1 << rega_out | 1 << regc_in
T3 :   1 << regc_out
T4 :   0
T5 :   0
```

JMP :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   1 << ir_out | 1 << pc_load
T3 :   0
T4 :   0
T5 :   0
```

JNZ (when Z ≠ 1) :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   1 << ir_out | 1 << pc_load
T3 :   0
T4 :   0
T5 :   0
```

JNZ (when Z = 1) :

```
1 << pc_out | 1 << mar_in
1 << mem_out | 1 << ir_in | 1 << pc_inc
0
0
0
0
```

SWAP :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   1 << rega_out | 1 << regc_in
T3 :   1 << regb_out | 1 << rega_in
T4 :   1 << regc_out | 1 << regb_in
T5 :   0
```

LSL :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   1 << alu_out | 1 << rega_in
T3 :   0
T4 :   0
T5 :   0
```

LSR :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   1 << alu_out | 1 << rega_in
T3 :   0
T4 :   0
T5 :   0
```

CMP :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   1 << alu_out
T3 :   0
T4 :   0
T5 :   0
```

DEC :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   1 << alu_out | 1 << rega_in
T3 :   0
T4 :   0
T5 :   0
```

MOV :

```
T0 :   1 << pc_out | 1 << mar_in
T1 :   1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :   1 << regc_out | 1 << regb_in
T3 :   0
T4 :   0
T5 :   0
```

JG (when Z = 0 and C = 1) :

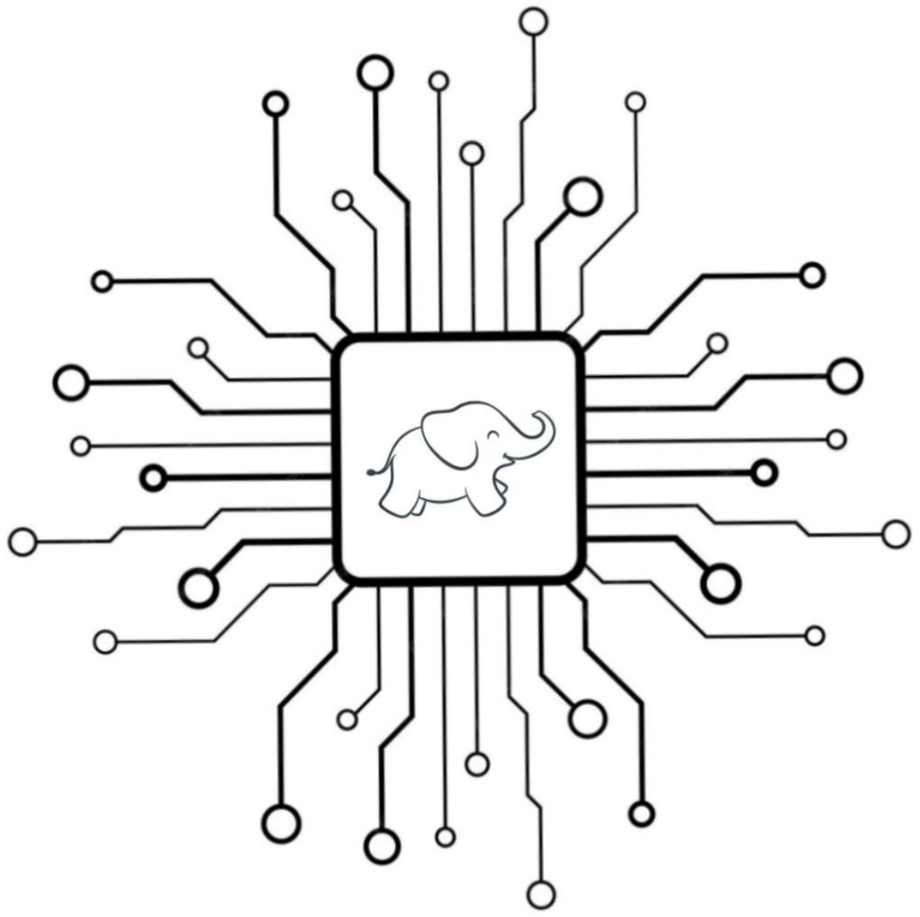
```
T0 :  1 << pc_out | 1 << mar_in
T1 :  1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :  1 << pc_load | 1 << ir_out
T3 :  0
T4 :  0
T5 :  0
```

JG (Otherwise) :

```
1 << pc_out | 1 << mar_in
1 << mem_out | 1 << ir_in | 1 << pc_inc
0
0
0
0
```

HALT :

```
T0 :  1 << pc_out | 1 << mar_in
T1 :  1 << mem_out | 1 << ir_in | 1 << pc_inc
T2 :  1 << reset
T3 :  0
T4 :  0
T5 :  0
```



For any queries contact :
Varun Teja CS22B050
Sri Charith CS22B052