GAJENDRA PROCESSOR

July - Nov 2023

Under the guidance of Professor : **Sir Ayon Chakraborty.**

Submitted by:

Group 41

- Malireddi Sri Sai Shanmukh raj (CS22B029)
- Mandipalli Lohith Reddy (CS22B030)

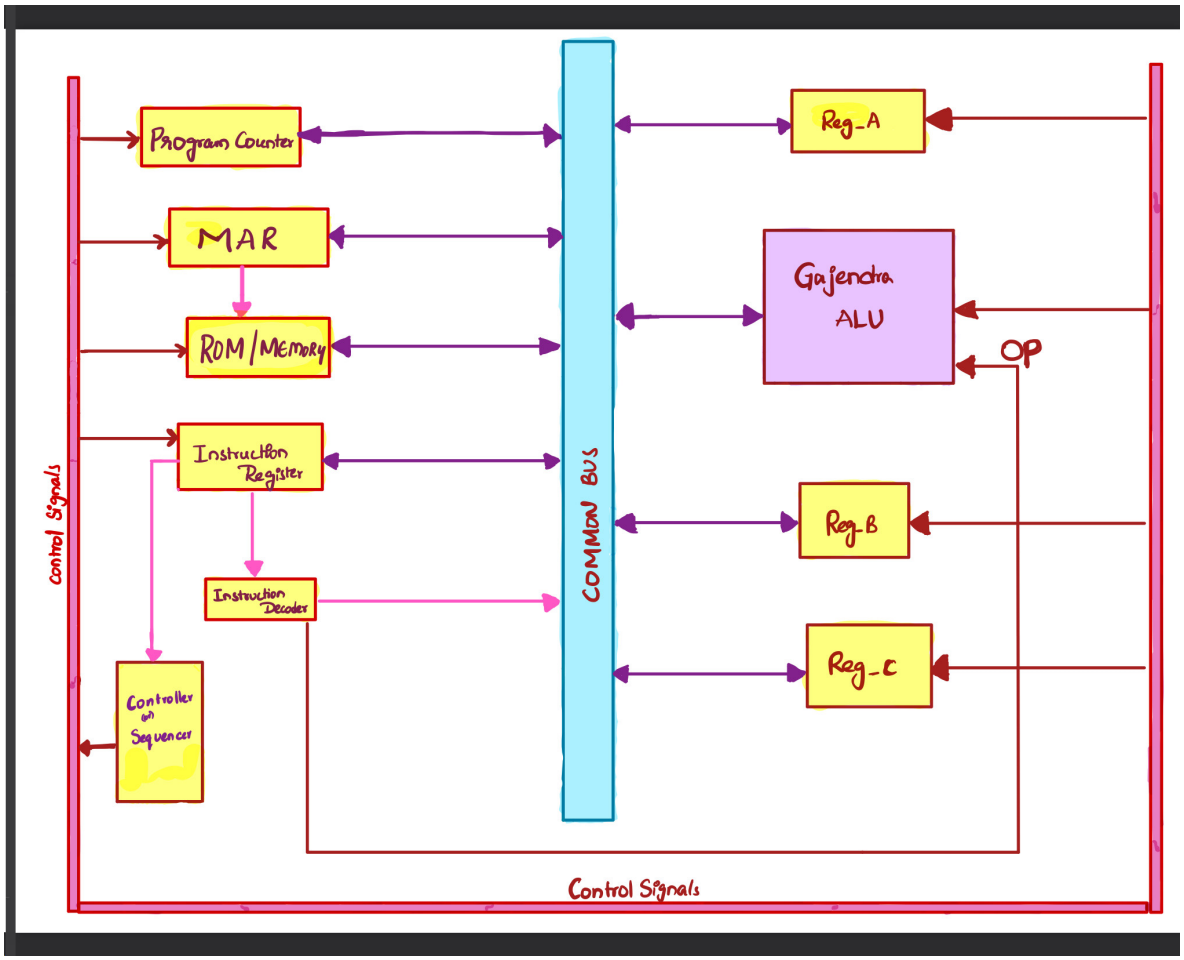
Contents

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

5.) Micro Instructions and Controller Logic Design

- NOP
- LDA
- STA
- ADD
- SUB
- LDI
- JMP
- SWAP
- JNZ
- MOVAC
- MOVBA
- MOVCB
- MOVAB
- MOVCA
- MOVBC
- HLT

Schematic Diagram of Gajendra

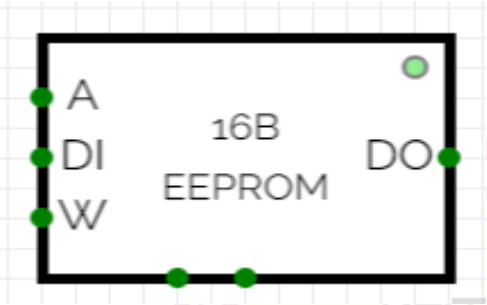


Internal Components

• Memory

- Memory is EEPROM used in controller which contains instructions and data required for the program.
- Memory is ROM which is used in cpu_core_arch_gajendra.

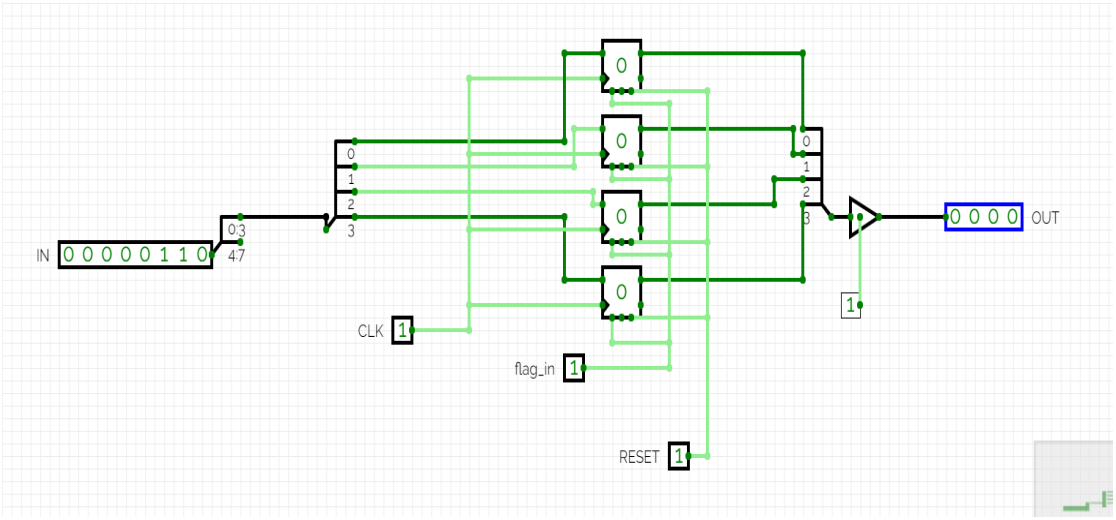
- **Circuit diagram:**



• Memory Address Register

- During execution the 4-bit address from the 8-bit input(LSB) in the program Counter is transferred into the MAR, then the MAR applies this 4-bit address to the Memory where a read operation is executed.

- **Circuit Diagram :**

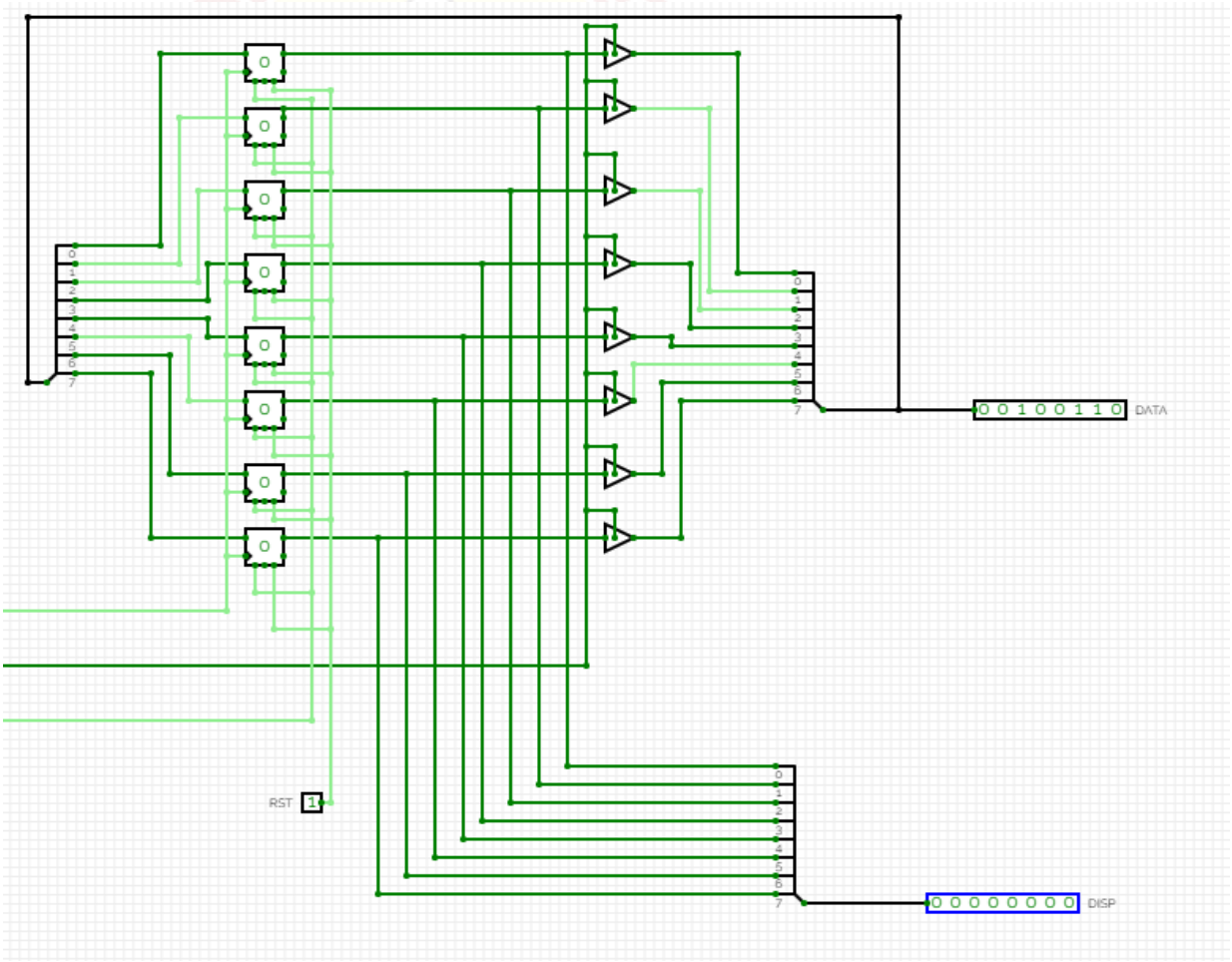


- # General CPU Register
  - We use D-Flip Flops in the register.

  - It takes an 8-bit input from the common bus and stores the provided value. The 8-bit data stored within it is then presented as the output.

  - Here the control inputs are REG_IN and REG_OUT. The working of control inputs are :
    1.) When REG_IN is active (usually set to 1), the data on the DATA bus is taken into the register.
    2.) When REG_OUT is active(usually set to 1), the data stored in the register is shown in the DATA bus.
    3.) The DISP output is connected to 8-bit output, and it shows the current value stored in the register. This output continuously changes with the data in the register.

  - **Circuit diagram :**



- Here we are using 3 Registers A,B,C. ‘A’ will also serve as our Accumulator.
- An accumulator is the register that is used to carry the sum or difference over multiple instructions.
- REG_B is used to get input from the memory. This is the only register that communicates with the memory firsthand.
- REG_C is used as an output register and as well as a temporary register in swapping Accumulator and B.

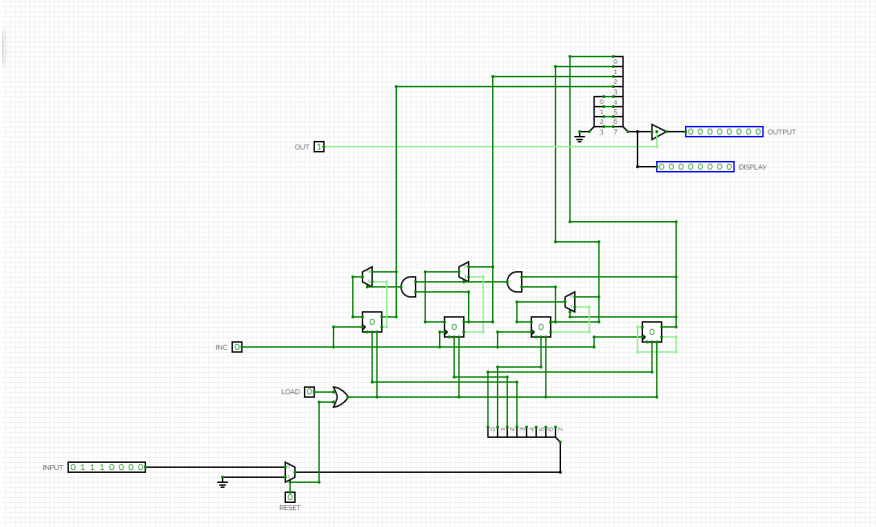
• Program Counter

- The program Counterpoints to the address of the instruction in the ROM, that has to be processed next.

- When PC_LD(Load) is set to 1, the 8-bit value from the common bus is taken as input to the Program Counter. The 4 least significant bits (LSB) of the input are set as the output, with the most significant bits(MSB) set to zero.

- When Reset is set to 1, all bits of the input change to 0.

- **Circuit Diagram :**



• Arithmetic Logic Unit (ALU)

- ALU is the most important component of the processor.

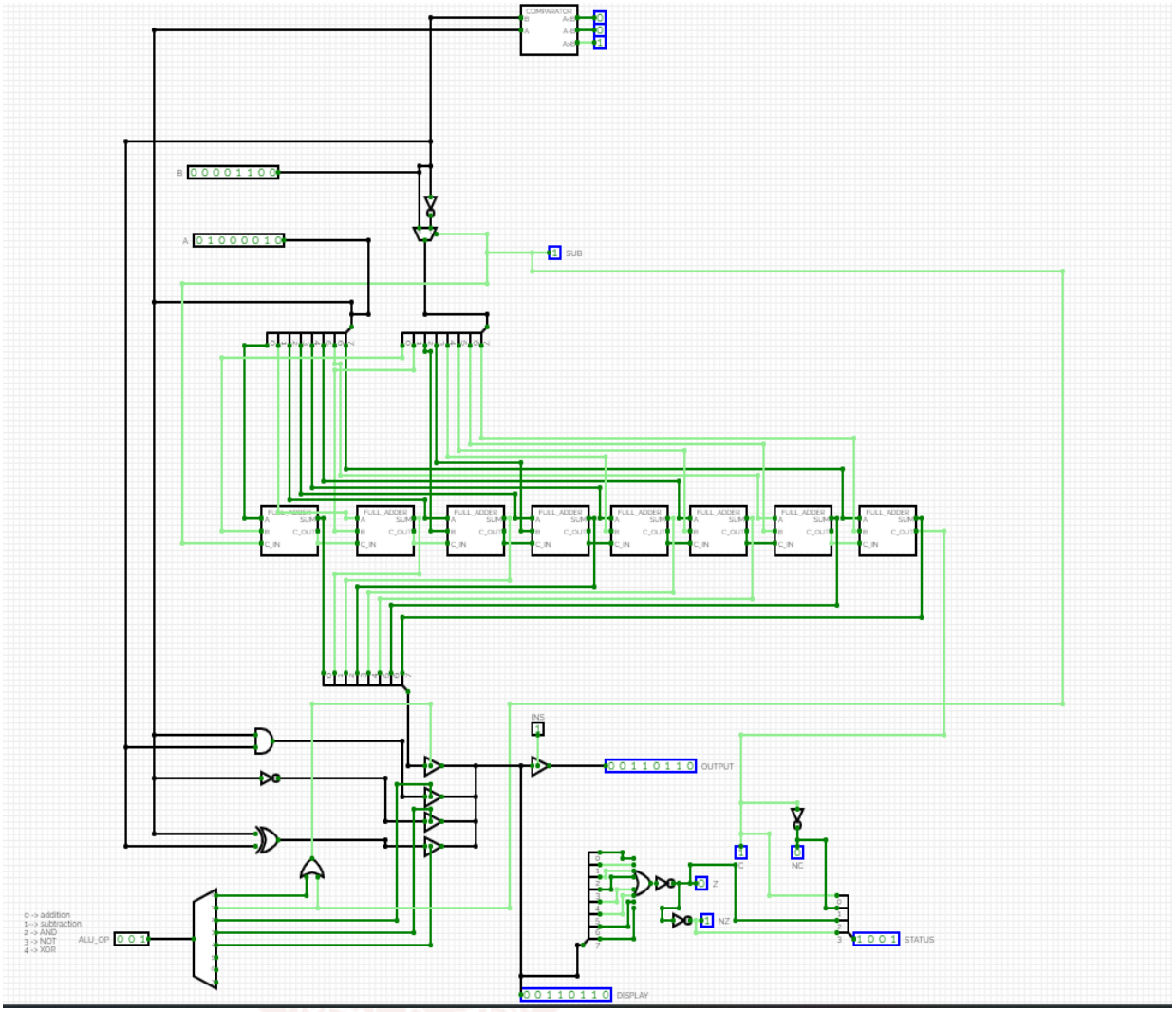
Gajendra can support the following 8 instructions :

- 000 - Addition is performed
- 001 - Subtraction is performed
- 010 - AND is performed
- 011 - NOT of A is performed
- 100 - XOR shift of A
- 101 - Right shift of A
- 110 - Left shift of A

- The instruction for this will be sent by the instruction decoder when an instruction that requires the operation of ALU is needed. For addition and subtraction, we are using 8 1-bit full adders.
- XOR, AND, NOT are performed by using the respective gates.
- 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.
- Here Z will serve as the flag to the Controller.
- If all the 8 bits in the display of ALU are zero then Z will be set as 0 otherwise it is set as 1.
- Here in this ALU, it automatically compares the values A and B. That means every time we pass two values A,B into the ALU, it displays ‘1’ in the A>B block if A>B, it displays ‘1’ in the A<B block if A<B, else it displays ‘1’ in A=B block if A=B.

- **Circuit diagram :**

(IN THE CIRCUIT DIAGRAM WE HAVE USED THE RIPPLE CARRY ADDER, COMPARATOR WHICH DISPLAYS A>B OR A<B OR A==B)

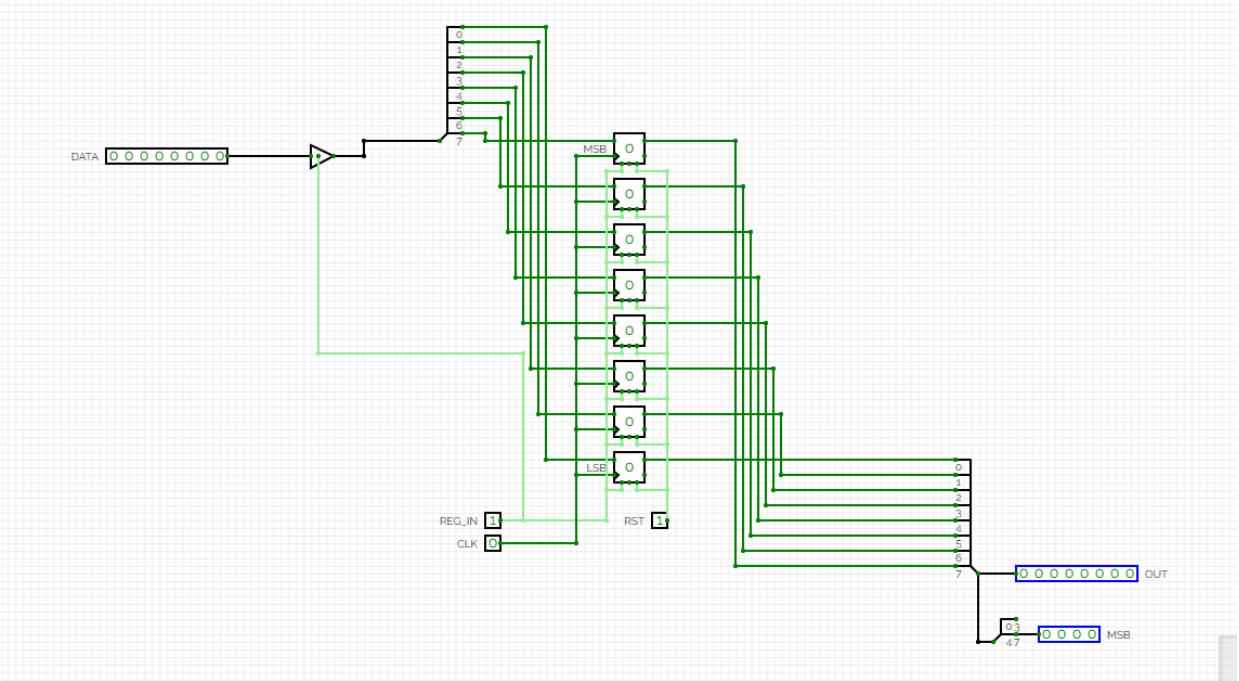


• Instruction Register

- Instruction register is similar to general purpose registers as they both have the same bit size.
- This is not bidirectional. But there are the same control words 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 MSBs to the instruction decoder which decodes which operation 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, we used D-flipflops for storing the data. The MSB as shown is an unrestricted output.

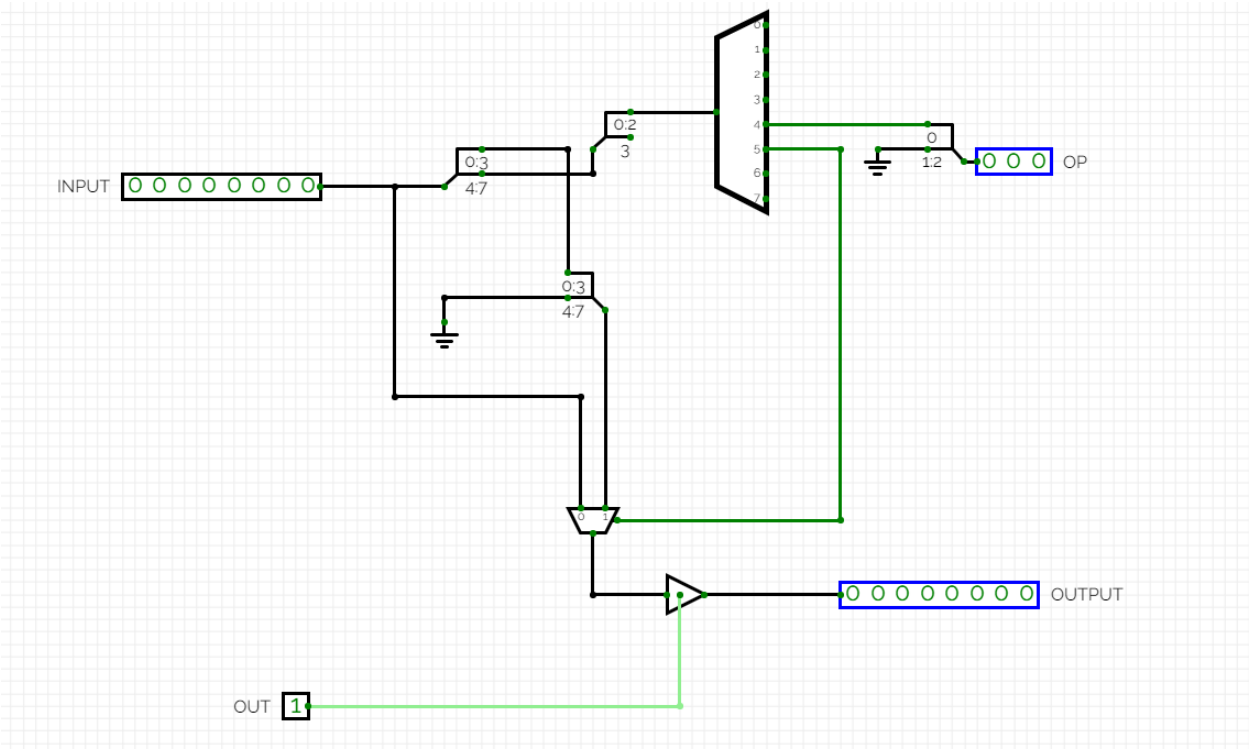
- **Circuit diagram :**



• Instruction Decoder

- Input is directly taken from reg IR without any tri-state buffer.
- The first 4 most significant bits decide what operation to be done.
- When IR_OUT is 1, output is displayed.
- When Operation is LDI 4 most significant bits of output become zeros'.

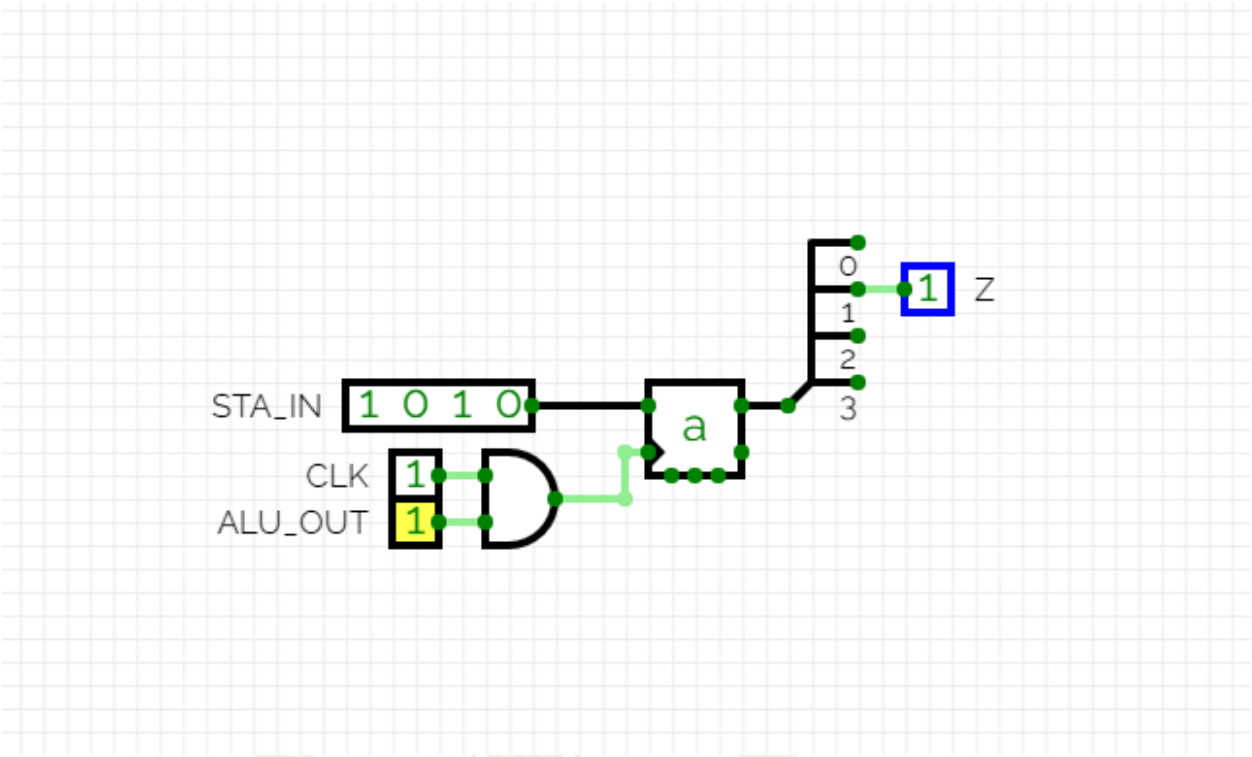
- **Circuit diagram :**



• Status Register

- A 4-bit status register for storing flags (C, NC, Z, NZ).
- When ALU OUT is set to 1, Output is displayed.

- **Circuit diagram :**



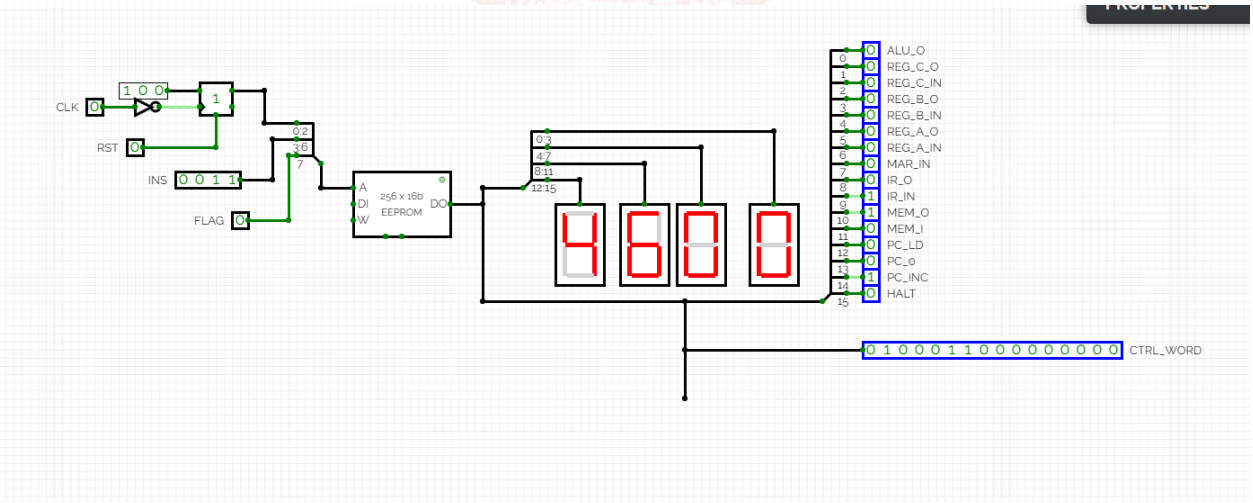
Controller

- The controller will give the commands in the proper sequence that has to be implemented.
- It contains EEPROM which is encoded by the below-shown data, and as the clock progresses, it passes the information to the splitter in our required order.

EEPROM encoded data: (EEPROM used is 16 bit width and 8 address width)

```
0 0000 NOP:  0x2080, 0x4600, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
1 0001 LDA:  0x2080, 0x4600, 0x0180, 0x0440, 0x0000, 0x0000, 0x0000, 0x0000
2 0010 STA:  0x2080, 0x4600, 0x0180, 0x0820, 0x0000, 0x0000, 0x0000, 0x0000
3 0011 ADD:  0x2080, 0x4600, 0x0180, 0x0410, 0x0041, 0x0000, 0x0000, 0x0000
4 0100 SUB:  0x2080, 0x4600, 0x0180, 0x0410, 0x0041, 0x0000, 0x0000, 0x0000
5 0101 LDI:  0x2080, 0x4600, 0x0140, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
6 0110 JMP:  0x2080, 0x4600, 0x1100, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
7 0111 SWAP: 0x2080, 0x4600, 0x0030, 0x0042, 0x000C, 0x0000, 0x0000, 0x0000
8 1000 JNZ:  0x2080, 0x4600, 0x1100, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
9 1001 MOVAC:0x2080, 0x4600, 0x0024, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
A 1010 MOVBA:0x2080, 0x4600, 0x0048, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
B 1011 MOVCB:0x2080, 0x4600, 0x0012, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
C 1100 MOVAB:0x2080, 0x4600, 0x0030, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
D 1101 MOVCA:0x2080, 0x4600, 0x0042, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
E 1110 MOVBC:0x2080, 0x4600, 0x000C, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
F 1111 HLT:  0x0080, 0x0600, 0x8000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
```

- **Circuit Diagram :**



Instruction and Machine Code

Instruction	Machine Code
NOP	0000 0X0
LDA	0001 0X1
STA	0010 0X2
ADD	0011 0X3
SUB	0100 0X4
LDI	0101 0X5
JMP	0110 0X6
SWAP	0111 0X7
JNZ	1000 0X8
MOVAC	1001 0X9
MOVBA	1010 0Xa
MOVCB	1011 0xb
MOVAB	1100 0xc
MOVCA	1101 0xd
MOVBC	1110 0xe
HLT	1111 0xf

Description of Instruction Set

NOP - No Operation

- **Description** : No operation is done.
- **Operation** : None

Syntax	Operands	Program Counter
NOP XXXX	X=0	PC ← PC+1

- **8 bit OPCode :**

0000	XXXX

LDA - Load Accumulator

- **Description** : This instruction loads the accumulator with the contents from the memory by looking at the 4-bit address provided.
- **Operation** : Ra ← ROM

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

- **8-bit OPCode :**

0001	XXXX

• STA - Store at Address A

- **Description** : This instruction takes the value at accumulator and stores in memory at address A.

- **Operation** : Ra ← ROM

Syntax	Operands	Program Counter
STA XXXX	X=0 or X=1	PC ← PC+1

- **8-bit OPCode :**

0010	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** : Rb ← ROM
  Ra ← Ra + Rb

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

- **8-bit OPCode:**

0011	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 up the contents of Register B from Accumulator and stores the final difference(non-negative) in Accumulator.

- **Operation** : Rb ← ROM
  Ra ← Ra - Rb

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

- **8-bit OPCode:**

0100	XXXX

• LDI - Load Immediate

- **Description** : Loads the 4-bit constant directly to Accumulator
- **Operation** : Ra ← k

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

- **8-bit OPCode:**

0101	XXXX

• JMP - Jump to Address

- **Description** : This Instruction takes the control to the given memory location.
- **Operation** : ROM Address: XXXX

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

- **8-bit OPCode:**

0110	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** : Rc ← Rb
  Rb ← Ra
  Ra ← Rc

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

- **8-bit OPCode:**

0111	XXXX

- ## JNZ - Jump when Non-Zero

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

- **Operation** : if(NZ=1) then ROM Address: XXXX

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

- **8-bit OPCode:**

1000	XXXX

• MOVAC - Copy data from A to C

- **Description** : The contents present in Register A are loaded into the Register C .
- **Operation** : Rc ← Ra

Syntax	Operands	Program Counter
MOVAC XXXX	X=0	PC ← PC+1

- **8-bit OPCode:**

1001	XXXX

• MOVBA - Copy data from B to A

- **Description** : The contents present in Register B are loaded into the Register A.

- **Operation** : Ra ← Rb

Syntax	Operands	Program Counter
MOVBA XXXX	X=0	PC ← PC+1

- **8-bit OPCode:**

1010	XXXX

- MOVCB - Copy data from C to B

- **Description** : The contents present in Register C are loaded into the Register B

- **Operation** : Rb ← Rc

Syntax	Operands	Program Counter
MOVCB XXXX	X=0	PC ← PC+1

- **8-bit OPCode:**

1011	XXXX

- ## MOVAB - Copy data from A to B

- **Description:** The contents present in Register A are loaded into the Register B

- **Operation** : Rb ← Ra

Syntax	Operands	Program Counter
MOVAB XXXX	X=0	PC ← PC+1

- **8-bit OPCode:**

1100	XXXX

- ## MOVCA - Copy data from C to A

- **Description:** The contents present in Register C are loaded into the Register A.
- **Operation** : Ra ← Rc

Syntax	Operands	Program Counter
MOVCA XXXX	X=0	PC ← PC+1

- **8-bit OPCode:**

1101	XXXX

- ## MOVBC - Copy data from B to C

- **Description:** The contents present in Register B are loaded into the Register C .

- **Operation** : Rc ← Rb

Syntax	Operands	Program Counter
MOVBC XXXX	X=0	PC ← PC+1

- **8-bit OPCode:**

1110	XXXX

- ## HLT - End the Instruction (Halt)

- **Description:** Fetch cycle stops.

Syntax	Operands	Program Counter
HLT XXXX	X=0	PC ← PC+1

- **8-bit OPCode:**

1111	XXXX

Assembly Programs implemented using the Instruction Set

- Add two numbers and display the result

Address	Assembly Code	Machine Code
0X0	LDA 0X5	0X15
0X1	ADD 0X6	0X36
0X2	MOVAC 0X0	0X90
0X3	HLT 0X0	0Xf0
0X4		0x00
0X5		0x34
0X6		0x12

- Value at address 0X5(say some 34) is loaded to accumulator.
- Value at address 0X6(say some 12) is added to the value of accumulator(to give 46) and store in it.
- Value in the accumulator is moved to Register C for displaying.
- Instructions end here.

- Add two numbers and display the result

Address	Assembly Code	Machine Code
0x0	LDA 0x5	0x15
0x1	SUB 0x6	0x46
0x2	ADD 0x7	0x37
0x3	SUB 0x8	0x48
0x4	HLT 0x0	0xf0
0x5		0x17
0x6		0x08
0x7		0x25
0x8		0x12

- Value at address 0X5(say 17) is loaded to accumulator.
- Value at address 0X6(say 08) is subtracted from the value of the accumulator(to give 09) and stored in it.
- Value at address 0X7(say 25) is added to value of the accumulator(to give 34) and stored in it.
- Value at address 0X8(say 12) is subtracted from the value of the accumulator(to give 22) and stored in it.
- Instructions end here

- Adding numbers from a starting address to an ending address and displaying the result

Address	Assembly Code	Machine Code
0x0	LDI 0x2	0x52
0x1	ADD 0x8	0x38
0x2	ADD 0x9	0x39
0x3	ADD 0xa	0x3a
0x4	ADD 0xb	0x3b
0x5	ADD 0xc	0x3c
0x6	ADD 0xd	0x3d
0x7	HLT 0x0	0xf0
0x8		0x19
0x9		0x26
0xa		0x4b
0xb		0x04
0xc		0x10
0xd		0x09
0xe		0x00
0xf		0x00

- Value of 02 is loaded to accumulator.
- Value at address 0x8(say 19) is added to value of the accumulator(to give 21) and stored in it.
- Value at address 0X9(say 26) is added to value of accumulator(to give 47) and stored in it.
- Value at address 0Xa(say 20) is added to value of accumulator(to give 67) and stored in it.

- Value at address 0Xb(say 04) is added to value of accumulator(to give 71) and stored in it.
- Value at address 0Xc(say 10) is added to value of accumulator(to give 81) and stored in it.
- Value at address 0Xd(say 09) is added to value of accumulator(to give 90) and stored in it.
- Instructions end here

Micro Instructions and Controller Logoc Design

- NOP

1 << PC_OUT | 1 << MAR_IN	T0
1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN	T1
0	T2
0	T3
0	T4

Micro Instruction for NOP

- LDA

1 << PC_OUT | 1 << MAR_IN	T0
1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN	T1
1 << IR_OUT | 1 << MAR_IN	T2
1 << MEM_OUT | 1<<REGA_IN	T3
0	T4

Micro Instruction for LDA

- STA

1 << PC_OUT | 1 << MAR_IN 1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN 1 << IR_OUT | 1 << MAR_IN 1 << MEM_IN | 1<<REGA_OUT 0	T0 T1 T2 T3 T4

Micro Instruction for STA

- ADD

1 << PC_OUT | 1 << MAR_IN 1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN 1 << IR_OUT | 1 << MAR_IN 1 << MEM_OUT | 1<<REGB_IN 1 << ALU_OUT | 1<< REGA_IN 0	T0 T1 T2 T3 T4 T5

Micro Instruction for ADD

- SUB

1 << PC_OUT | 1 << MAR_IN 1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN 1 << IR_OUT | 1 << MAR_IN 1 << MEM_OUT | 1<<REGB_IN 1 << ALU_OUT | 1<< REGA_IN	T0 T1 T2 T3 T4

Micro Instruction for SUB

- LDI

1 << PC_OUT | 1 << MAR_IN 1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN 1 << IR_OUT | 1 << REGA_IN 0 0	T0 T1 T2 T3 T4

Micro Instruction for LDI

- JMP

1 << PC_OUT | 1 << MAR_IN 1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN 1 << IR_OUT | 1 << PC_LOAD 0 0	T0 T1 T2 T3 T4

Micro Instruction for JMP

- SWAP

1 << PC_OUT | 1 << MAR_IN 1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN 1 << REGA_OUT | 1 << REGC_IN 1 << REGB_OUT | 1 << REGA_IN 1 << REGC_OUT | 1 << REGB_IN 0	T0 T1 T2 T3 T4 T5

Micro Instruction for SWAP

- JNZ

1 << PC_OUT | 1 << MAR_IN	T0
1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN	T1
1 << IR_OUT | 1 << PC_LOAD	T2
0	T3
0	T4

Micro Instruction for JNZ (If flag(Z)=0)

1 << PC_OUT | 1 << MAR_IN	T0
1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN	T1
0	T2
0	T3
0	T4

Micro Instruction for JNZ (If flag(Z)=1)

- MOVAC

1 << PC_OUT | 1 << MAR_IN	T0
1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN	T1
1 << REGA_OUT | 1 << REGC_IN	T2
0	T3
0	T4

Micro Instruction for MOVAC

- MOVBA

1 << PC_OUT | 1 << MAR_IN 1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN 1 << REGB_OUT | 1 << REGA_IN 0 0	T0 T1 T2 T3 T4

Micro Instruction for MOVBA

- MOVCB

1 << PC_OUT | 1 << MAR_IN 1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN 1 << REGC_OUT | 1 << REGB_IN 0 0	T0 T1 T2 T3 T4

Micro Instruction for MOVCB

- MOVAB

1 << PC_OUT | 1 << MAR_IN 1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN 1 << REGA_OUT | 1 << REGB_IN 0 0	T0 T1 T2 T3 T4

Micro Instruction for MOVAB

- MOVCA

1 << PC_OUT | 1 << MAR_IN 1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN 1 << REGC_OUT | 1 << REGA_IN 0 0	T0 T1 T2 T3 T4

Micro Instruction for MOVCA

- MOVBC

1 << PC_OUT | 1 << MAR_IN 1 << PC_INC | 1 << MEM_OUT | 1 << IR_IN 1 << REGB_OUT | 1 << REGC_IN 0 0	T0 T1 T2 T3 T4

Micro Instruction for MOVBC

- HLT

1 << MAR_IN 1 << MEM_OUT | 1 << IR_IN 0 0	T0 T1 T2 T3

Micro Instruction for HLT
(As per our circuit design we need two halts to stop our program)
Fetch cycle is not used in HALT instruction due to this circuit works only one time,to avoid this we have used RESET in reg IR so that at the start of function Fetch Cycle of NOP runs.

The final cpu_core_arch_gajendra