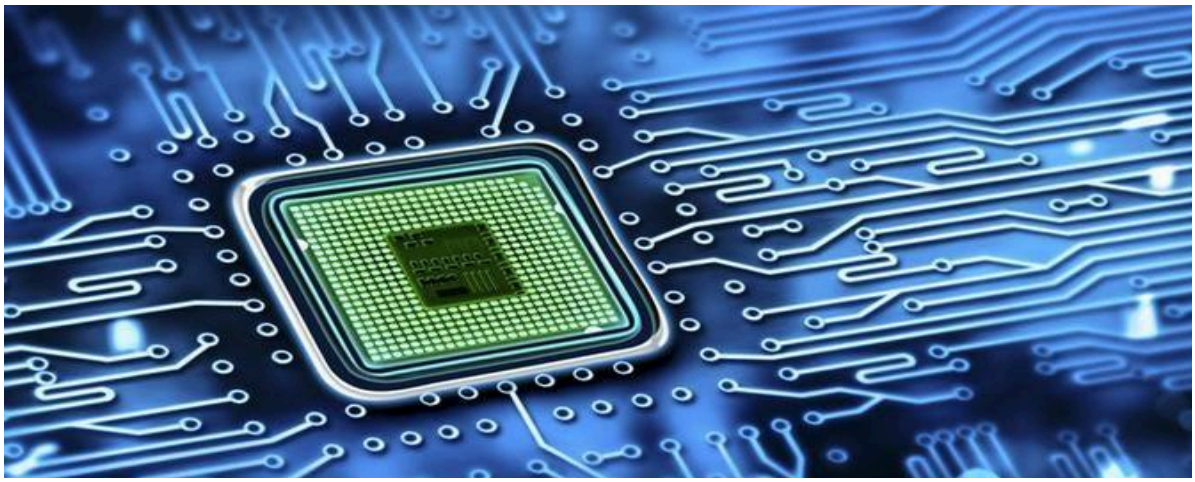
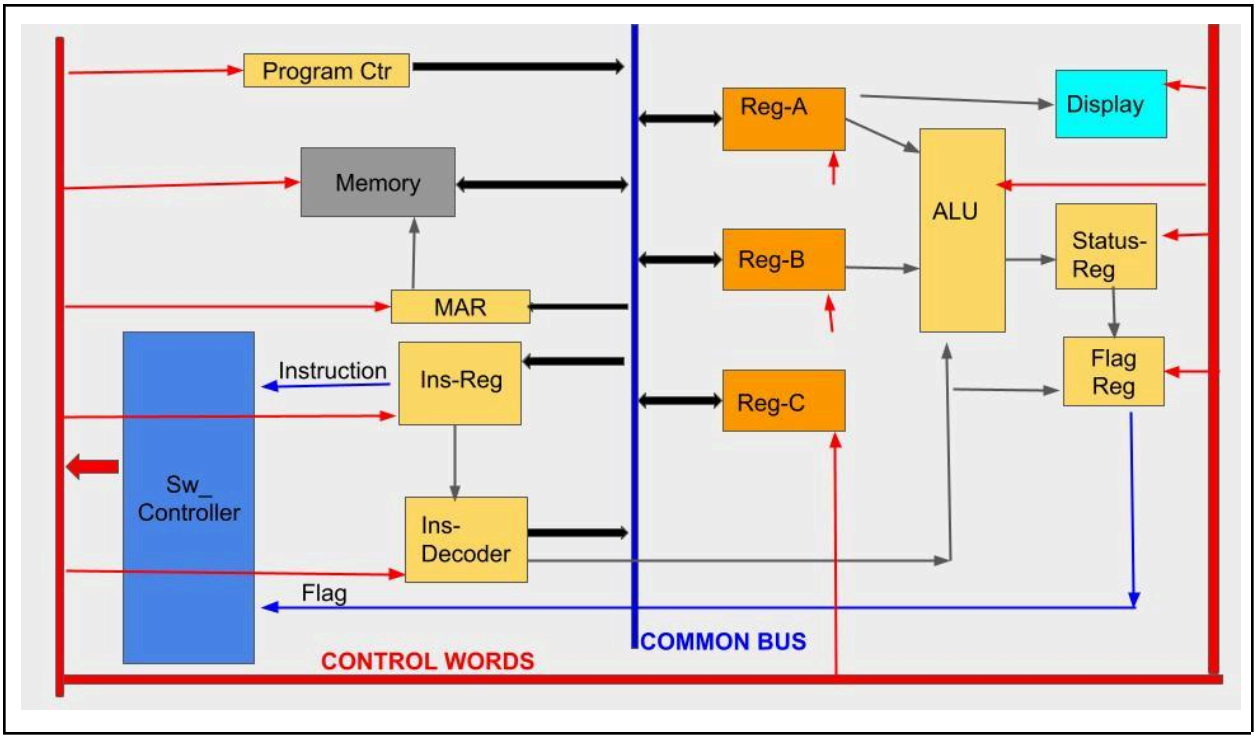


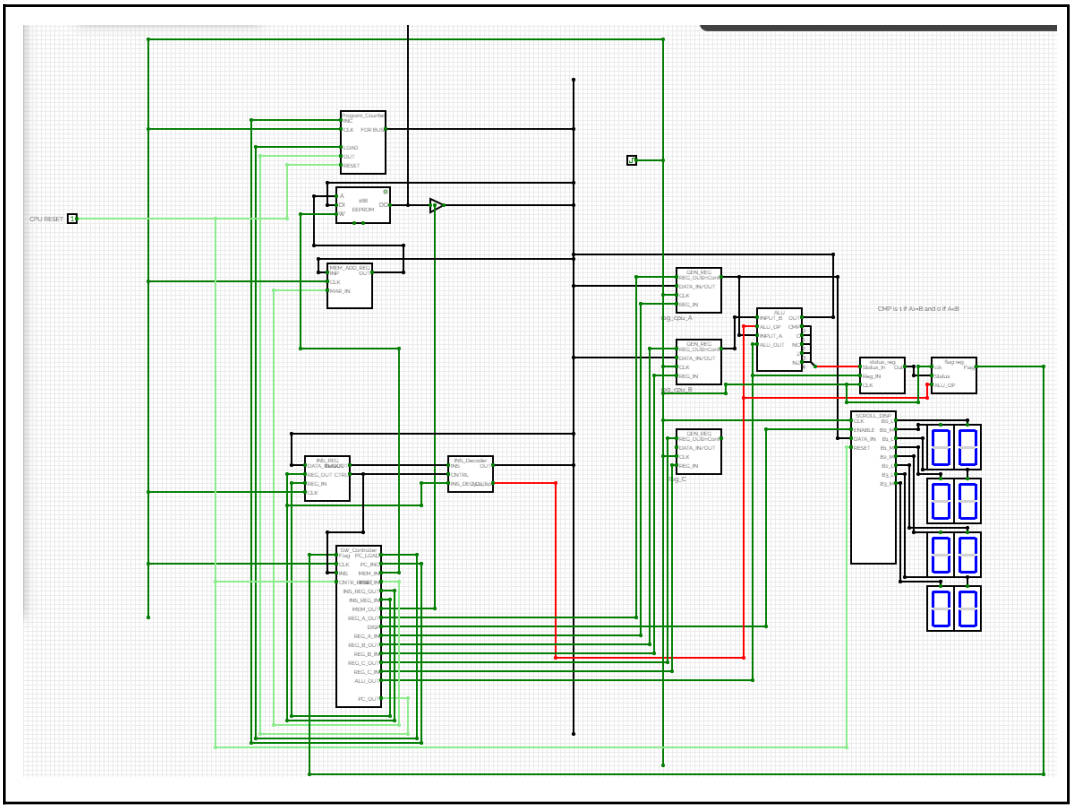
Gajendra-1 Report

Devi Sumanth CS22B073
Harsh Vardhan Daga CS22B075

Section 1 State the overall architecture for your CPU Core

Our processor's overall architecture looks something like this.

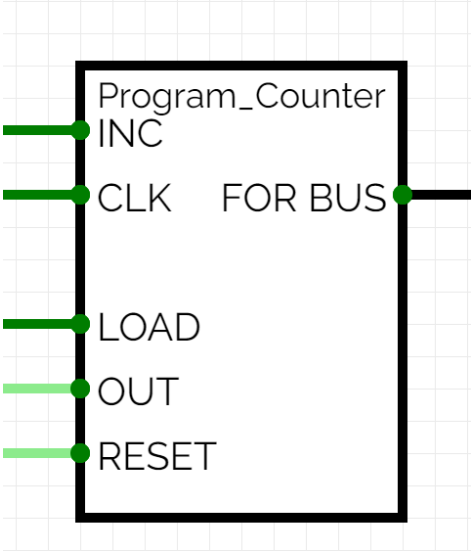


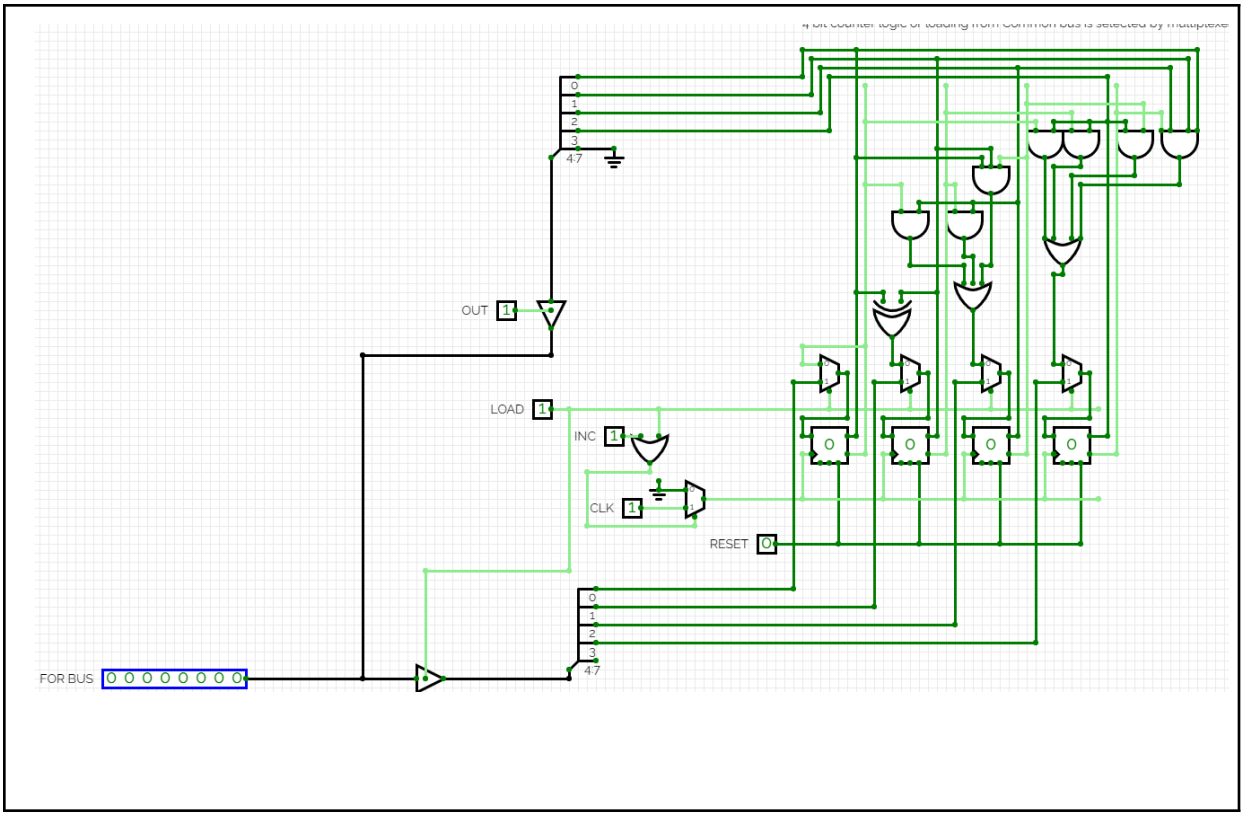


Components

Program Counter (PC)

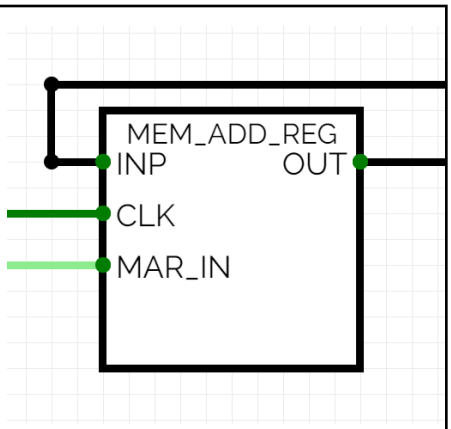
- Program counter basically tells memory to which location it should point to (via MAR)
- When INC is 1 value in the counter increases by one.
- When LOAD is 1, value of counter is loaded with the data coming from bus
- When OUT is 1, value of the counter is written to the bus
- When RESET is 1 counter gets reset to 0
- The design logic we used is using multiplexers with the selection lines as LOAD where if LOAD is 1 the input to flip flop comes from bus else normal counter logic is the input .





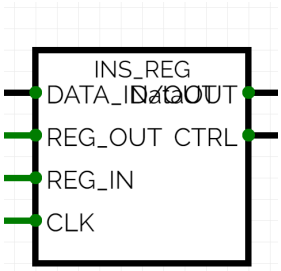
Memory Address Register (MAR)

- This is just a normal 4 bit register, which takes data from the bus (stripping off the most significant 4 bits) and stores it.
- The data is directly sent as address input to the memory without any control
- In general MAR takes data from either PC -> BUS-> or INS_REG -> BUS ->



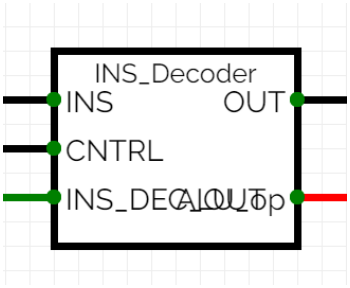
Instruction Register (IR)

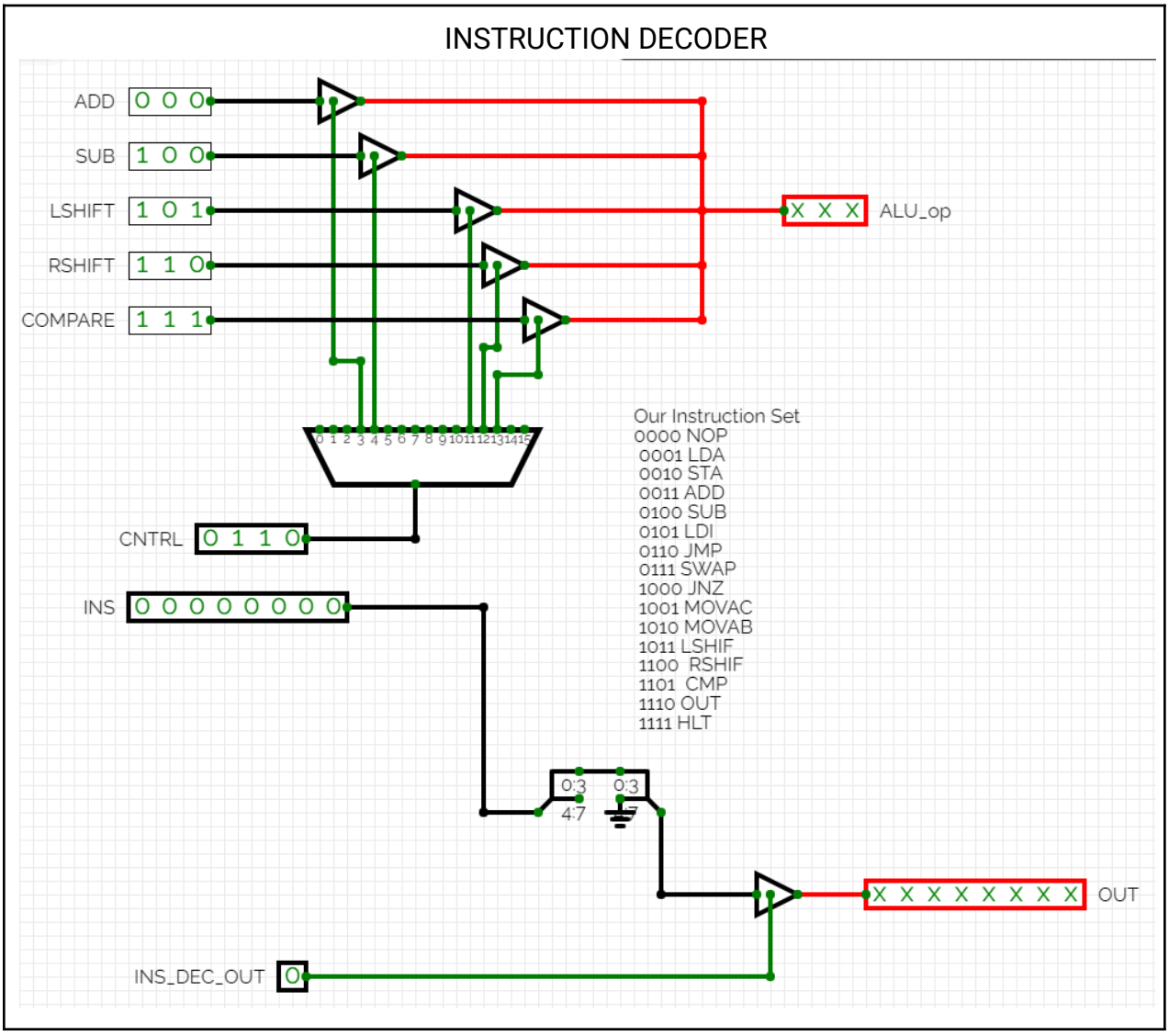
- It stores the the 8 bit opcode instruction+data (that comes from memory)
- The 4 bits of instruction is sent to controller and 8 bits are sent to Instruction Decoder
- Controller gives the control words based on the instruction, which it takes from IR



Instruction Decoder

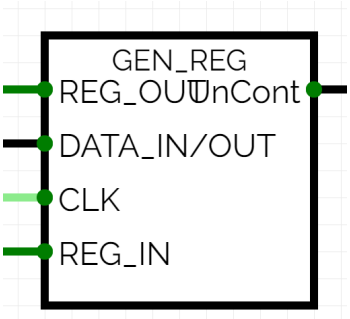
- It takes the data from IR and does mainly these two jobs
- 1) Stripping of the instruction from the toptal opcode and sending remaining data whenever IR_OUT is 1
- 2) Process the instruction and gives ALU_OP to ALU based on which ALU knows what operation to perform
- The logic we used to map ALU_OP with instruction can be understood by below circuit of instruction decoder





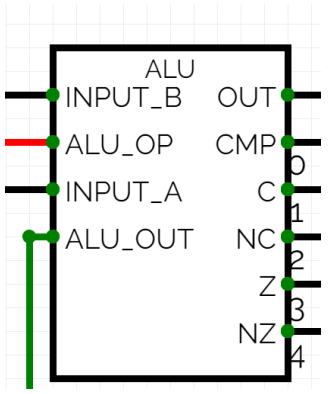
General Purpose Registers (REG_A, REG_B, REG_C)

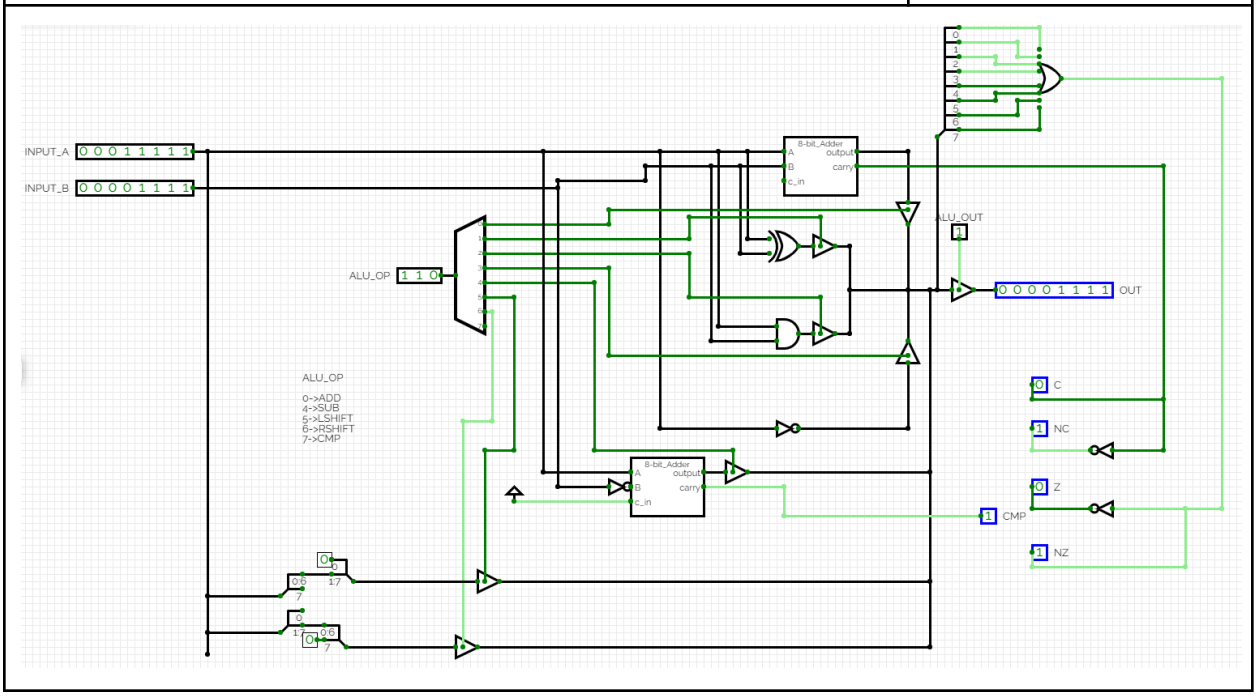
- These are used for storing data for computation
- We made these bi directional, i.e both reading and writing (from and) to the bus will be done through a single wire(connected through DATA_IN/OUT)
- When REG_IN is 1 data is written into register from bus
- When REG_OUT is 1 data is read from register to the bus



Arithmetic and Logical Unit (ALU)

- Our ALU takes INPUT (DATA) from registers A and B and (OPERATION) ALU_OP from INS_DECODER.
- Based on ALU_OP it selects what data to be outputted to the bus and outputs it when ALU_OUT is 1.
- The Z and NZ tells if the computed result of the operation is zero or not
- The C and NC tells if we'll have a carry when the data in both A and B are added
- CMP is 1 if data in A is >= B and is 0 if A < B
- These will be used as flags and the explanation of which flag will be selected will be in the FLAG_REG module.

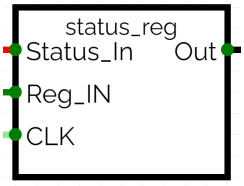




* We have other operations like bitwise or ,bitwise and, bitwise not included in ALU, but we're not going to use them as instructions for our instruction set, so for our PC, these will never be selected.

Status Register

- This is a normal 4 bit register, which stores all the flags. We update flags whenever ALU_OUT is 1.
- Status_In have CMP, C, NC, Z, NZ (5 bit).
- Reg_IN is controlled by ALU_OUT.

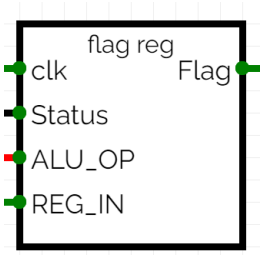


Flag Register

First I'll explain why we used this register. Our controller can take only one flag (controller is **flag + 4 bit ins + 3 bit T-states**), but depending on the instruction of our instruction set flag has to be corresponding to CMP when the previous instruction is CMP else it has to be of NZ. So I'm using a register which stores the flag corresponding to the most recent SUB instruction or CMP instruction.

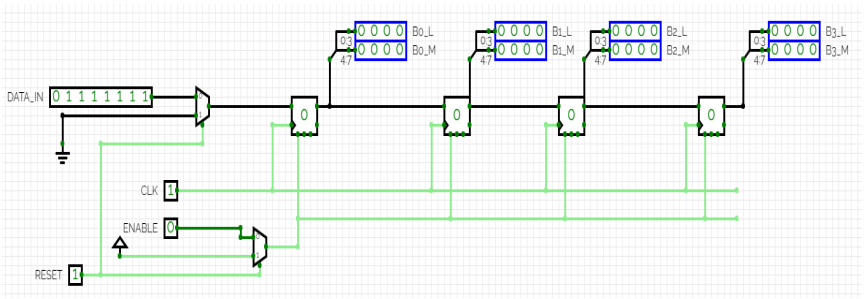
If we have bits available in the controller word for all the flags then we don't need to have this as a special register.

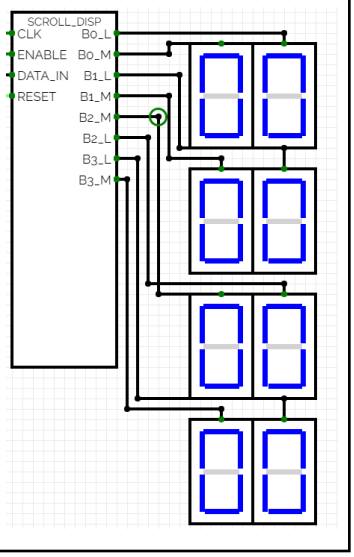
- If ALU_OP corresponds to SUB then it stores the NZ
- If ALU_OP corresponds to CMP then it stores the CMP
- ALU_OUT acts as REG_IN for this.
- So for CMP instruction even though ew won't write anything to the bus from ALU we should have ALU_OUT as 1 for the architecture we followed for the flag being correct.



Display

- We used a scroll display that displays the contents of accumulator (REG_A) when ever the instruction is OUT
  (in which we have Enable as 1, so it displays)





Controller

We used a software based controller in which control words are hard-coded in control rom which can be addressed by the state given by FLAG + INS + T-STATE.
Design of controller and filling control rom's data is in Section 4.

Section 2. Define your Instruction Set (IS)

We followed 8 bit opcodes, with most significant 4 bits as instruction and least significant 4 bits as data.

So we're having a maximum of 16 instructions

The instructions are

1. 0000 - NOP
2. 0001 - LDA
3. 0010 - STA
4. 0011 - ADD
5. 0100 - SUB
6. 0101 - LDI
7. 0110 - JMP
8. 0111 - SWAP
9. 1000 - JNZ
10. 1001 - JZ
11. 1010 - MOVAC
12. 1011 - LSHIF
13. 1100 - RSHIF
14. 1101 - CMP
15. 1110 - OUT
16. 1111 - HALT

The instructions in Red require the 4 bit data, for others the 4 bit data is don't care.

1) <u>NOP</u>

Description:

Does nothing just increase the program counter by 1.

PC ← PC + 1

Syntax:

NOP XXXX == 0000 XXXX

2) <u>LDA</u>

Description:

Load the data present corresponding to the address in the opcode to accumulator

PC ← PC + 1

Reg_A ← *(aaaa)

Syntax:

LDA aaaa == 0001 aaaa

3) STA

Description:

Store the data present in the accumulator to the corresponding address in the opcode.

PC ← PC + 1

*(aaaa) ← Reg_A

Syntax:

STA aaaa == 0010 aaaa

4) ADD

Description:

Replace the value in the accumulator by adding the value in it to the data present corresponding to the address in the opcode.

PC ← PC + 1

Reg_A ← Reg_A + *(aaaa)

Syntax:

ADD aaaa == 0011 aaaa

Note : The data corresponding to (aaaa) is loaded into Reg_B in this instruction, so any data present in B will be erased.

5) SUB

Description:

Replace the value in the accumulator by subtracting the value in it with the data present corresponding to the address in the opcode.

PC ← PC + 1

Reg_A ← Reg_A - *(aaaa)

Syntax:

SUB aaaa == 0100 aaaa

Note : The data corresponding to (aaaa) is loaded into Reg_B in this instruction, so any data present in B will be erased.

6) LDI

Description:

Load the data present in least significant 4 bits of opcode to accumulator

PC ← PC + 1

Reg_A ← dddd

Syntax:

LDI dddd == 0101 dddd

7) JMP

Description:

Load the PC with the data present in the least significant 4 bits of the opcode.

PC ← dddd

Syntax:

JMP dddd == 0110 dddd

8) SWAP

Description:

Swap the data present in Reg_A and Reg_C.

PC ← PC + 1

Reg_A ← Reg_C

Reg_C ← Reg_A

Syntax:

SWAP XXXX == 0111 XXXX

Note: We use Reg_B for his instruction, hence data in Reg_B will be erased during the instruction.

9) JNZ

Description:

If flag is 1

Load the PC with the data present in the least significant 4 bits of the opcode.

PC ← dddd

If flag is 0

Does nothing just increase the program counter by 1.

PC ← PC + 1

Syntax:

JNZ dddd == 1000 dddd

Note: In the design I used, the flag comes from a flag register which stores NZ flag if we have SUB instruction and CMP flag if we have CMP instruction and for other instructions, flag in the flag register won't change.

10) JZ

Description:

If flag is 0

Load the PC with the data present in the least significant 4 bits of the opcode.

PC ← dddd

If flag is 1

Does nothing just increase the program counter by 1.

PC ← PC + 1

Syntax:

JZ dddd == 1000 dddd

Note: In the design I used, the flag comes from a flag register which stores NZ flag if we have SUB instruction and CMP flag if we have CMP instruction and for other instructions, flag in the flag register won't change.

11) MOVAC

Description:

Copies the value in Reg_A to Reg_C.

PC ← PC + 1

Reg_C ← Reg_A

Syntax:

MOVAC XXXX == 1001 XXXX

12) LSHIFT

Description:

Value in Reg_A gets left shifted by 1 bit.

PC ← PC + 1

Reg_A ← Reg_A << 1

Syntax:

LSHIFT XXXX == 1011 XXXX

13) RSHIFT

Description:

Value in Reg_A gets right shifted by 1 bit.

PC ← PC + 1

Reg_A ← Reg_A >> 1

Syntax:

RSHIFT XXXX == 1100 XXXX

14) CMP

Description:

Compares the value in the accumulator with the value corresponding to the address in the opcode and updates the flag.

If A >= *(dddd) flag set to 1.

If A < *(dddd) flag set to 0.

PC ← PC + 1

Syntax:

CMP dddd == 1101 dddd

Note: This instruction loads the data into Reg_B, so any data that is present in Reg_B will be lost.

15) OUT

Description:

Displays the data present in the accumulator.

PC ← PC + 1

Syntax:

OUT XXXX == 1110 XXXX

16) HALT

Description:

Program Counter Freezes and we should reboot to make the CPU work.

Syntax:

HALT XXXX == 1111 XXXX

Section 3. Give a few example assembly programs implemented using your IS

Assembly code	**Data into the Rom (Machine code and memory)**	**Expected result** **Green - Successful** **Red - Failed:(**
LDI 0111 **STA 1111** **LDI 1011** **LDA 1111** **OUT XXXX** **HLT XXXX**	0x57,0x2f,0x5b,0x1f,0xe0,0xf0	0XF location in memory is written as 7 and 7 should be displayed in the scroll display

LDI 0010 **ADD 1111** **OUT XXXX** **JMP 0001** **HLT XXXX**	0x52,0x3f,0xe0,0x61,0xf0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x03	Infinite addition of 3 to 2 Basically 2+3*n will go through display
LDI 1001 **OUT XXXX** **RSHIFT XXXX** **OUT XXXX** **LSHIFT XXXX** **OUT XXXX** **HALT XXXX**	0x59,0xe0,0xc0,0xe0,0xb0,0xe0,0xf0	Loads 9 to the accumulator and displays 9. Then right shift 9, it becomes 4 Displays 4 Then left shift 4, it becomes 8 Displays 8.
LDA 1111 **MOVAC XXXX** **LDI 0000** **ADD 1110** **OUT XXXX** **SWAP XXXX** **SUB 1101** **SWAP XXXX** **JNZ 0011** **HALT XXXX**	0x1f,0xa0,0x50,0x3e,0xe0,0x70,0x4d,0x70,0x83,0xf0,0x00,0x00,0x00,0x01,0x03,0x0a	Should display multiplication table of 3 from 3 to 30 3 6 9 12 ….. 30 and stop
LDI 0000 **MOVAC XXXX** **LDI 0000** **ADD 1110** **OUT XXXX** **SWAP XXXX** **ADD 1101** **CMP 1111** **SWAP XXXX** **JZ 0011** **HLT XXXX**	0x50,0xa0,0x50,0x3e,0xe0,0x70,0x3d,0xdf,0x70,0x93,0xf0,0x00,0x00,0x01,0x03,0x0a	Should display the same multiplication table of 3 as above But I used the CMP and corresponding CMP flag will be used for conditional jump and here conditional jump used is JZ

The above example programs cover all the instructions in our instruction set and every instruction seems to be working properly.

Section 4. Microinstructions and Controller Logic Design

Controller takes instruction from IR and flag from Flag Register and it has a T-state counter in it.
The **Flag + Instruction + T-State (8bits)** act as address to control rom in which control words are hard-coded and control words corresponding to the address are outputted which are used by other components of the processor.

Our Control word 2 bytes == 16 bits

15	14	13	12	11	10	9	8
DISP	ALU_OUT	Reg_C_IN	Reg_C_OUT	Reg_B_IN	Reg_B_OUT	Reg_A_IN	Reg_A_OUT

7	6	5	4	3	2	1	0
Mem_OUT	IR_IN	IR_OUT	MAR_IN	MEM_IN	PC_OUT	PC_LOAD	PC_INC

In out instruction set except JNZ and JZ remaining are flag independent

Control words for different Instructions

1) NOP (flag 0 and 1) - 2 machine cycles

```
1<<PC_OUT|1<<MAR_IN,
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,
0,0,0,0,0,0,
```

2) LDA (flag 0 and 1) - 4 machine cycles

```
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<<REGA_IN,
0,0,0,0,
```

3) STA (flag 0 and 1) - 4 machine cycles

```
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,
0,0,0,0,
```

4) ADD (flag 0 and 1) - 5 machine cycles

```
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,
0,0,0,
```

5) SUB (flag 0 and 1) - 5 machine cycles

```
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,
0,0,0,
```

6) LDI (flag 0 and 1) - 3 machine cycles

```
1<<PC_OUT|1<<MAR_IN,
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,
 1<<IR_OUT|1<<REGA_IN,
0,0,0,0,0,
```

7) JMP (flag 0 and 1) - 3 machine cycles

```
1<<PC_OUT|1<<MAR_IN,
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,
1<<IR_OUT|1<<PC_LOAD,
0,0,0,0,0,
```

8) SWAP (flag 0 and 1) - 5 machine cycles

```
1<<PC_OUT|1<<MAR_IN,
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,
1<<REGA_OUT|1<<REGB_IN,
1<<REGC_OUT|1<<REGA_IN,
1<<REGB_OUT|1<<REGC_IN,
0,0,0,
```

9) JNZ - 3 machine cycles

Flag 0

```
1<<PC_OUT|1<<MAR_IN,
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,
0,0,0,0,0,0,
```

Flag 1

```
1<<PC_OUT|1<<MAR_IN,
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,
1<<IR_OUT|1<<PC_LOAD,
0,0,0,0,0,
```

10) JZ - 3 machine cycles

Flag 0

```
1<<PC_OUT|1<<MAR_IN,
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,
1<<IR_OUT|1<<PC_LOAD,
0,0,0,0,0,
```

Flag 1

```
1<<PC_OUT|1<<MAR_IN,
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,
0,0,0,0,0,0,
```

11) MOVAC (flag 0 and 1) - 3 machine cycles

```
1<<PC_OUT|1<<MAR_IN,
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,
1<<REGA_OUT|1<<REGC_IN,
0,0,0,0,0,
```

12) LSHIFT (flag 0 and 1) - 3 machine cycles

```
1<<PC_OUT|1<<MAR_IN,
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,
1<<ALU_OUT|1<<REGA_IN,
0,0,0,0,0,
```

13) RSHIFT (flag 0 and 1) - 3 machine cycles

```
1<<PC_OUT|1<<MAR_IN,
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,
1<<ALU_OUT|1<<REGA_IN,
0,0,0,0,0,
```

14) CMP (flag 0 and 1) - 5 machine cycles

```
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,
0,0,0,
```

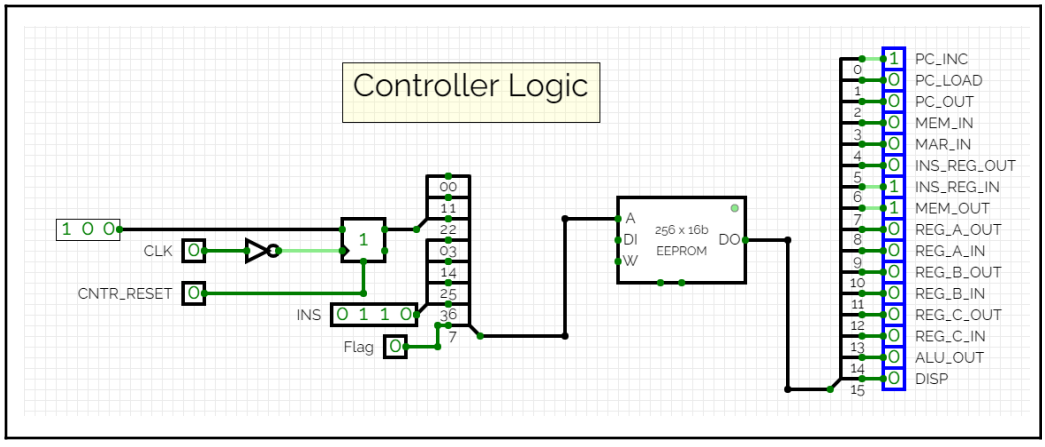
15) OUT (flag 0 and flag 1) - 3 machine cycles

```
1<<PC_OUT|1<<MAR_IN,
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,
1<<DISP,
0,0,0,0,0,
```

16) HALT (flag 0 and flag 1) - 2 machine cycles

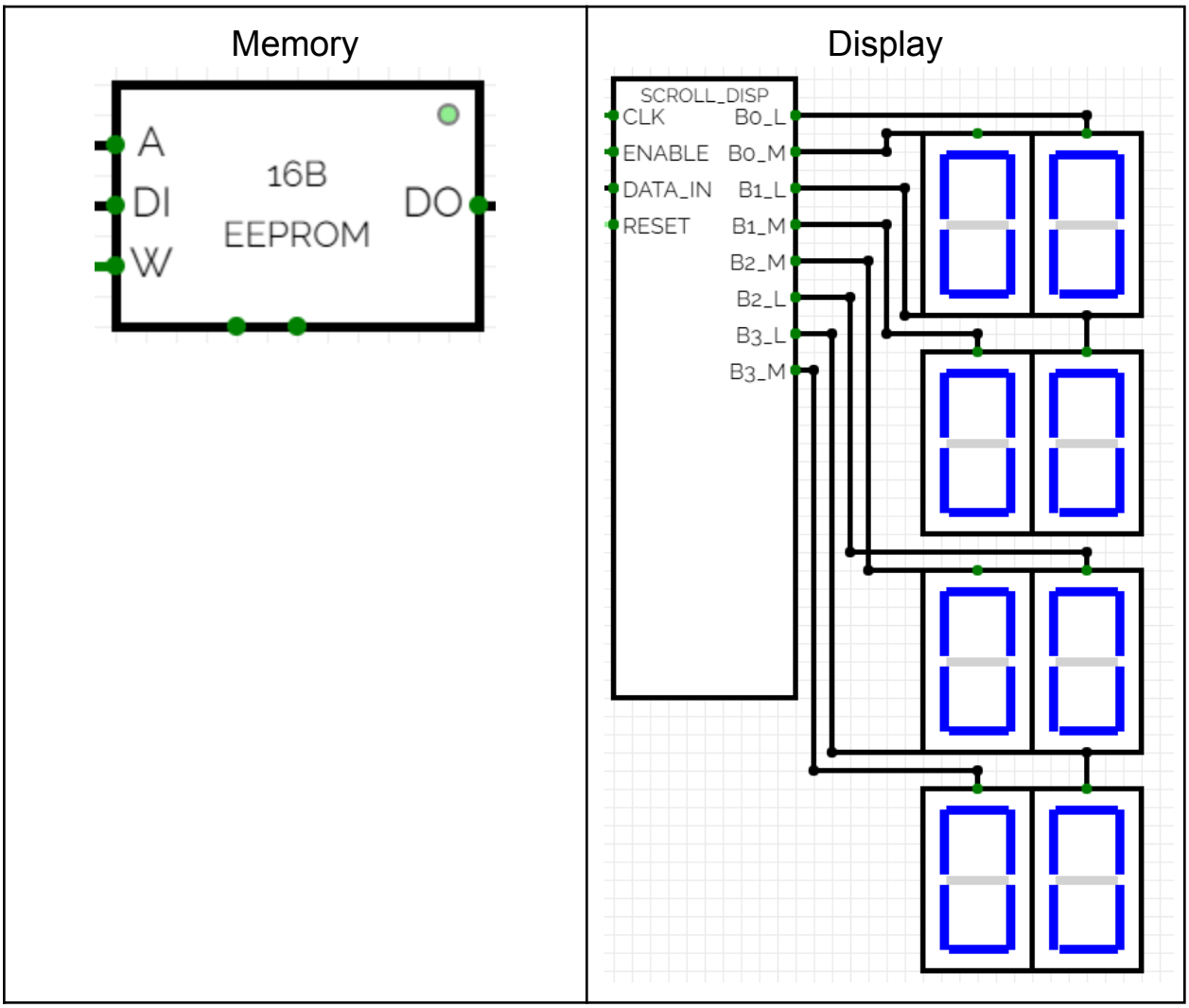
```
1<<MAR_IN,
1<<MEM_OUT|1<<IR_IN,
0,0,0,0,0,0
```

Therefore the maximum number of machine cycles required for any instruction is 5. Hence we have a mod 5 counter as T- state counter and we used not clock as clock for T-state counter because it ensures controls are ready when positive trigger happens.



Section 5. Connect the data/program memory and display, System Reset

Our program memory is a 16 byte ROM and we used a scroll display (scrolls the data in the accumulator whenever the instruction OUT is performed).



And we have a system reset which resets the program counter to 0 and T-state counter in controller to 0 and also IR to 0.
We are also resetting IR because during the start of the program, IR should not have instruction corresponding to HALT, if previously HALT is executed the fetch cycle won't occur properly since halt doesn't have the same micro instructions like the others for the 1st two T-States.