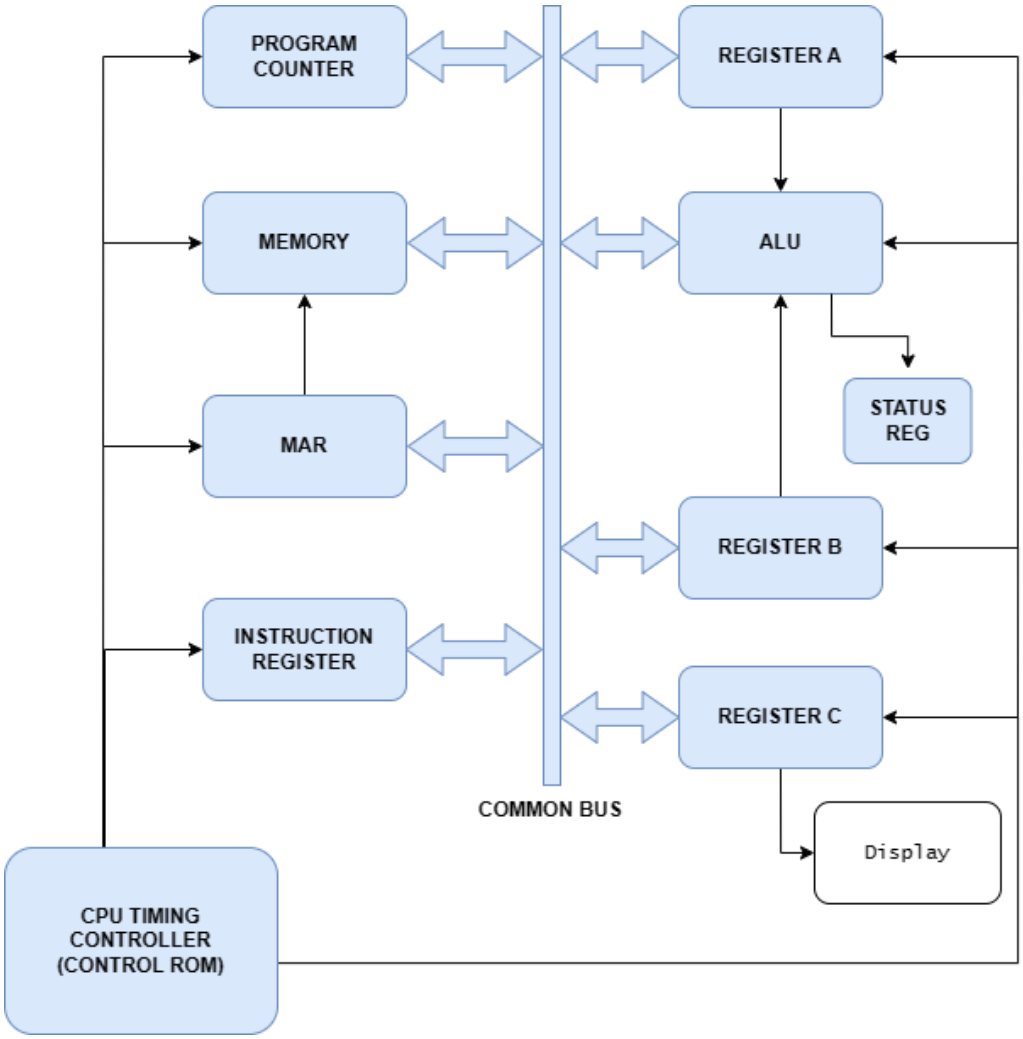
ARCHITECTURE OF THE GAJENDRA – PROCESSOR

Our processor consists of many modules, enlisted as follows:

- A Program Counter, whose purpose is to send the line of execution of the program to the Instruction Register.
- The Instruction Register (IR) (:8-bit), has many purposes. It firstly receives the line number from the counter (for a 4-bit program counter, it'll be from 0 to 15). Then it 'fetches' the required instruction from memory, that's accessed from MAR. The instruction sent to the IR consists of 8-bits, most significant 4 signifying the opcode of the instruction.

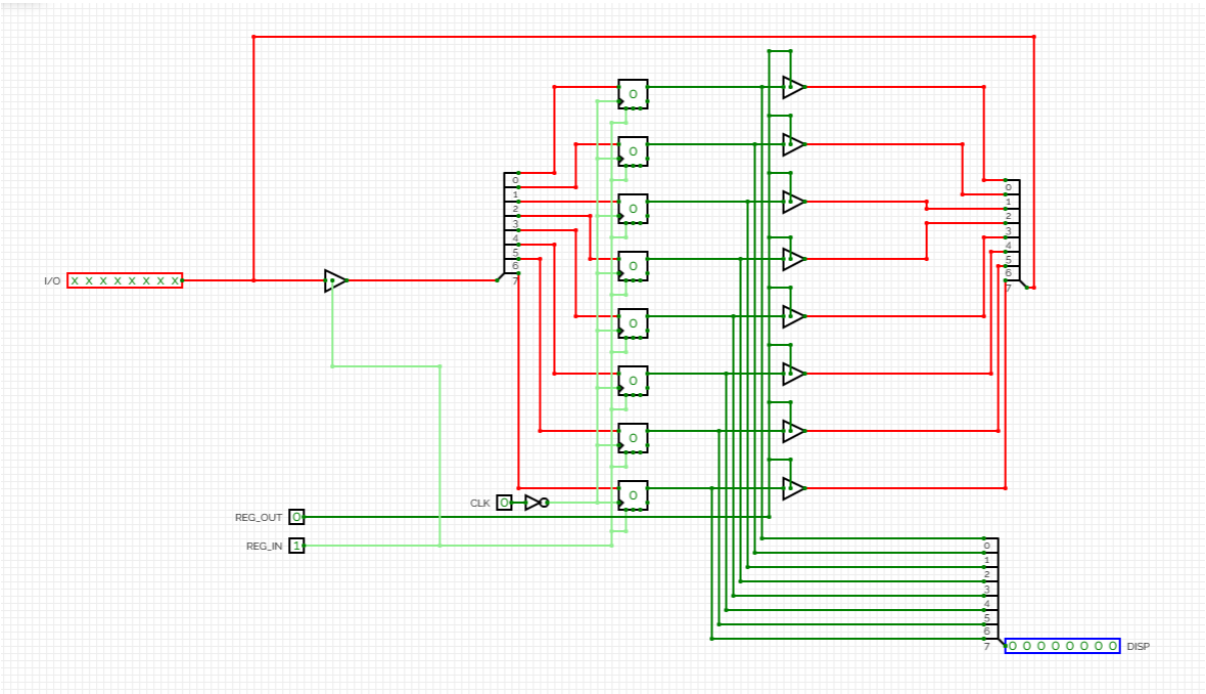
  [Elaborated further later.]
- The MAR is the Memory Address Register, which is an 8-bit register. To access an address from Memory, we send the address (typically from IR) via Common Bus to MAR, and its output is directly connected to the Memory.
- A ROM for Memory: Stores the program written in assembly code and also stores some of the data that'll needed to be accessed while the program is running. We currently have a ROM of size 16 bytes.
- An ALU: That supports addition, subtraction, comparing and some discontinued functions like XOR, AND, NOT. The output is processed to check for zero output, and for carry output. The zero/non-zero check and carry/no carry check is sent to the Status Register. There's also a check for negative output (intended to detect -ve output from subtraction).
- Status Register: 4-bit register that stores the four values described above (checks for zero and carry).
- Three general registers (A, B, C): A is used as the "accumulator". B is used typically for storing the second values in an expression. A and B's outputs are always sent to the ALU, so the ALU constantly receives these values as inputs. C is a register connected to a display. We have used a *"Scrolling Display"* unit.
- CPU timing controller is the software module that works on a ring counter that counts from 1 to 5. It has provisions for a 7-bit input (4 for opcode, 3 for T-state) and an extra bit for flag (Flag == 1 terminates the operation, basically no action is taken). This flag is sent from the IR after processing the

status register’s current values and the call of certain functions, that are described in a bit more detail later.



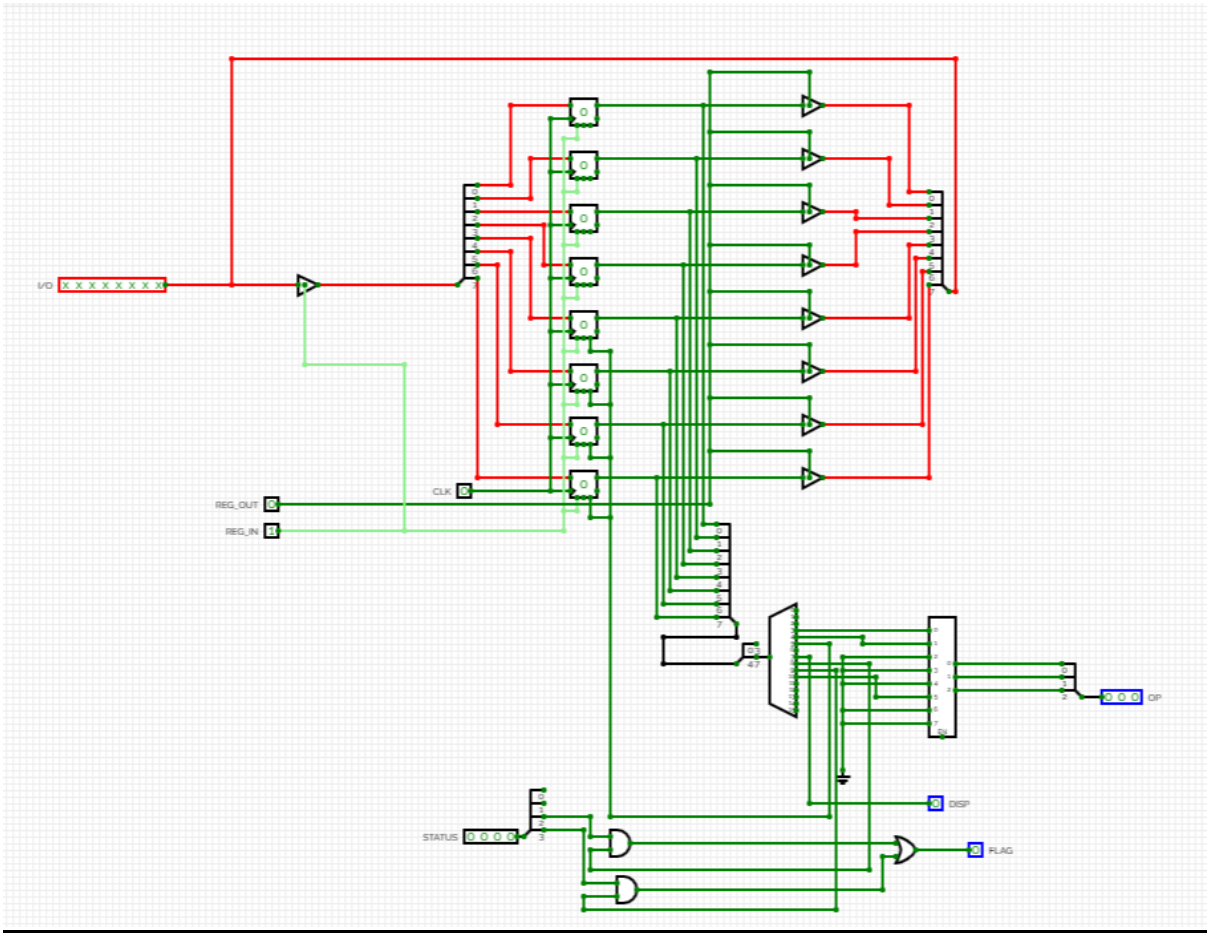
Schematic diagram of the arrangement of various components in the Processor

❖ General CPU Registers

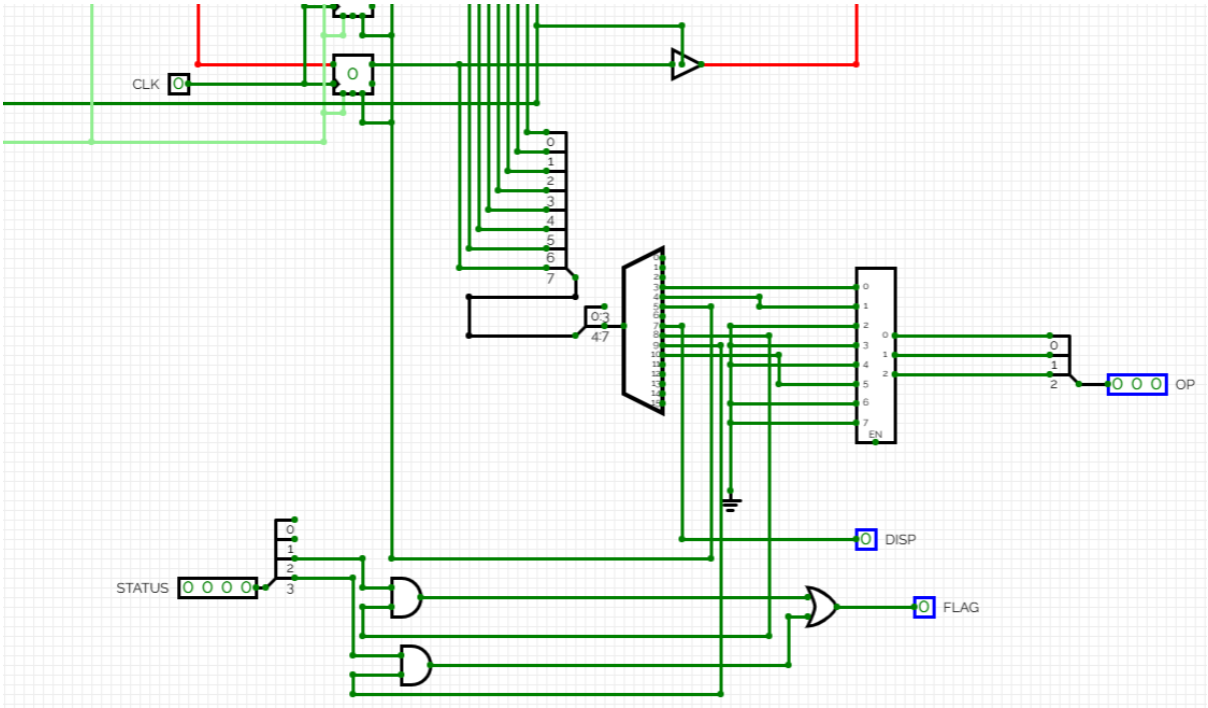


There is a common I/O bus, that is of 8-bits, and connects into the common bus. The flow of input is control by a single tristate buffer which is controlled by *REG_IN*. The flow of output is controlled by *REG_OUT*. So when REG_IN is 1, and REG_OUT is 0, the flip-flops are enabled, and at the next clock pulse (it's negative triggered) they store the input value sent to them. When both are 0, the value is held between the flip-flops and the tristate buffers. It's value can be displayed as shown in the figure. To send output to the common bus, we make REG_OUT as 1 and REG_IN as 0.
We have utilized this register as it is, in A, B and C, and modified it to make the IR.

❖ Instruction Register



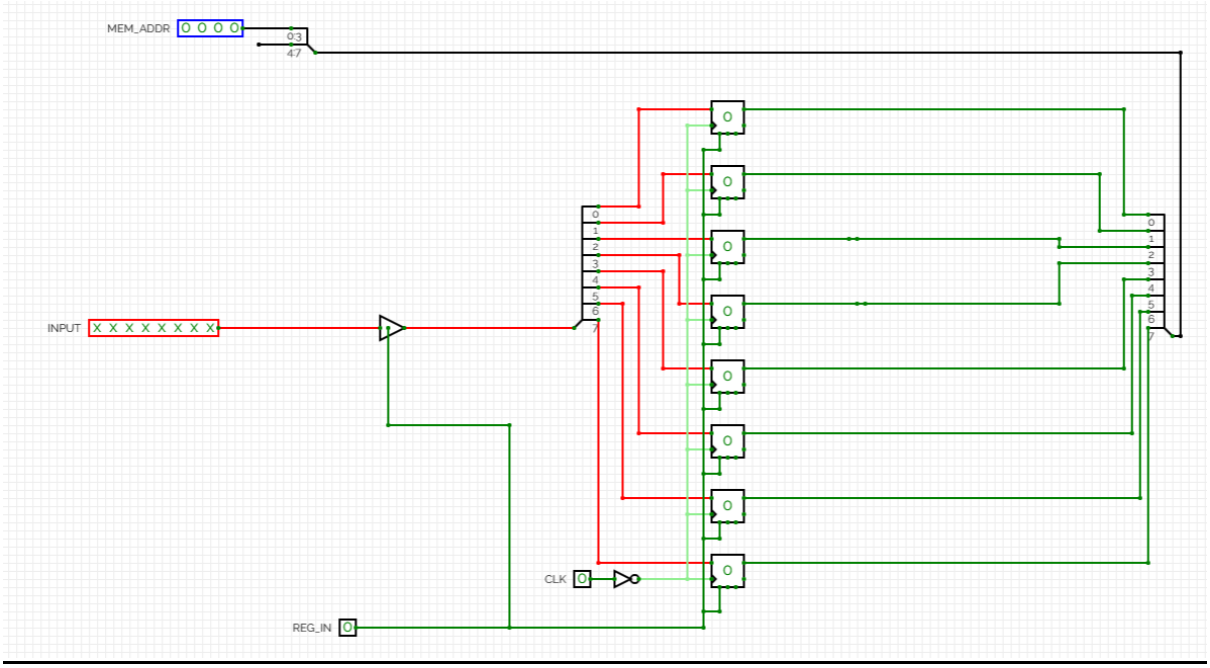
As seen in the figure, the IR is a modified version of the general purpose registers. The Input and Output work just like a normal register. Let's examine the lower part of the circuit with more detail.



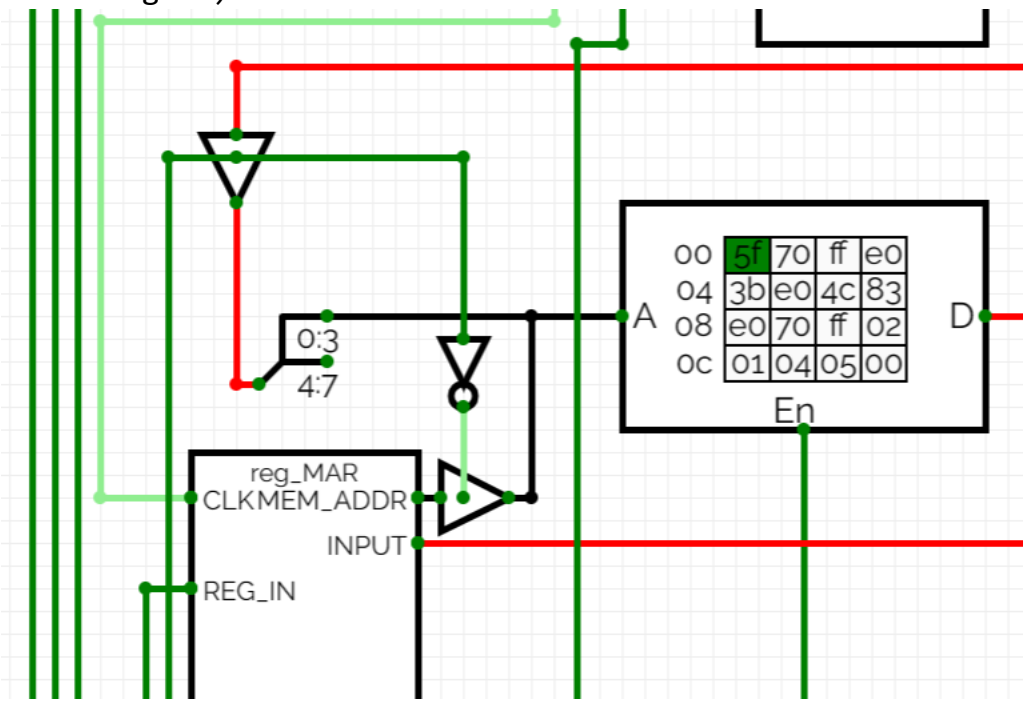
Unlike the general register, the display part of the register is connected to a decoder, and some of the outputs of the decoder are connected to an 8 to 3 encoder (we used priority encoder) whose 3-bit output gives the ALU the required operation *OP*.
[000: ADD, 001: SUB, 101: CMP. 010, 011 and 100 were initially reserved for XOR, AND and NOT, and so we went with 101 for CMP. But later on, due to limitations of using only a max of 16 instructions, we decided to omit these 3 bitwise operations.]
Notice that for opcode 0x5, the decoded line connects to an asynchronous reset, that resets the most 4 significant bits' flip-flops. 0x5 corresponds to LDI, and LDI x loads a 4 bit number immediately to register A. To get rid of the most four significant bits and pad them with 0, we make the connections in this way.
For opcode 0x7 (OUT), the decoded line controls the enable of the scrolling display. [When 0x7 is sent, then DISP becomes 1, and during the T-states, when *c_out* is made 1, only then the Scrolling Display's output is enabled.]

For the instructions 0x8 (JNZ) and 0x9 (JZ), we need to ensure that when JNZ is called, NZ should be 1 (only then we should jump), else we should ignore. We can say similar things for JZ.
So, when 0x8 is called, then we check that if NZ is 0 (or Z is 1), then it means it's a false call to jump to the required line number. The AND Gate returns 1, and the FLAG becomes 1. Similarly, when JZ is called and NZ is 1, then also, FLAG becomes 1. This *FLAG* is sent to the *CPU Timing Controller*, where it reacts by making the remaining control words as 0x00 when FLAG is 1.

❖ Memory Address Register (MAR)

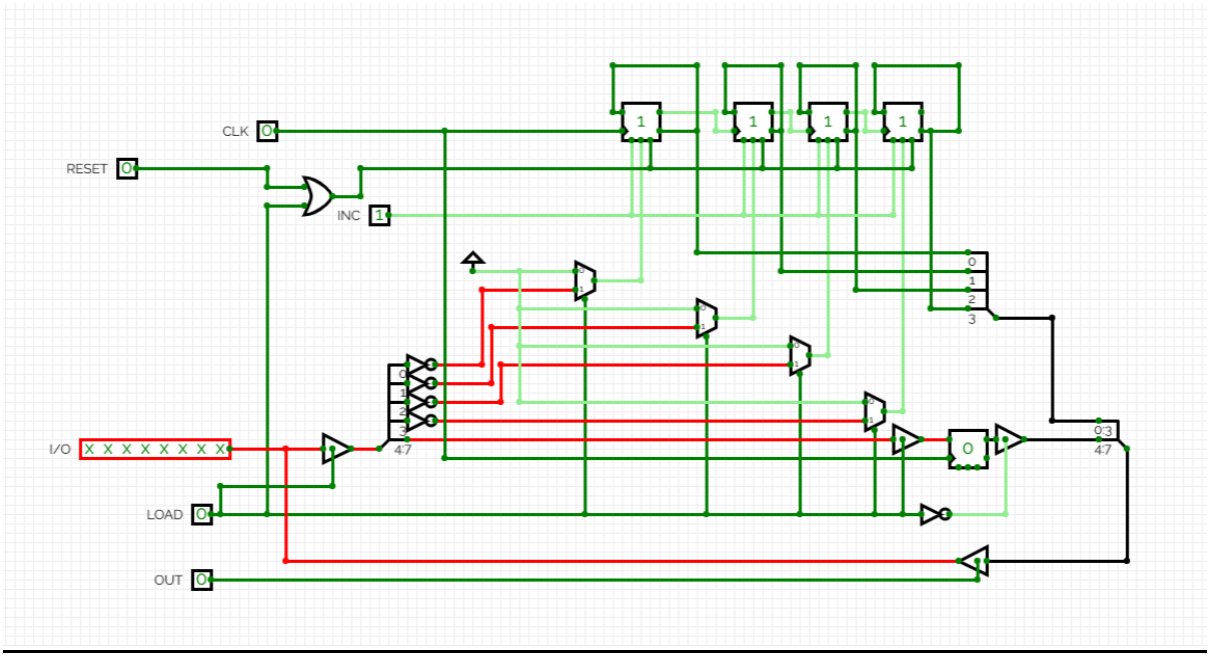


This is a modification of the 8-bit register, basically since OUT is always set to 1, we remove the buffers. And across the output, we connect a splitter, from which the [0:3] bits are used. These four least significant bits represent the address bits. Note that, this output is always connected to the input of the Memory ROM. However, at the end of the Memory, (see below figure)



When MEM_IN is set to 1, only then MAR's output is blocked, by setting control to the tristate buffer as zero. This is needed for immediate lookup of memory and/or modification, by functions like STA.

❖ Program Counter



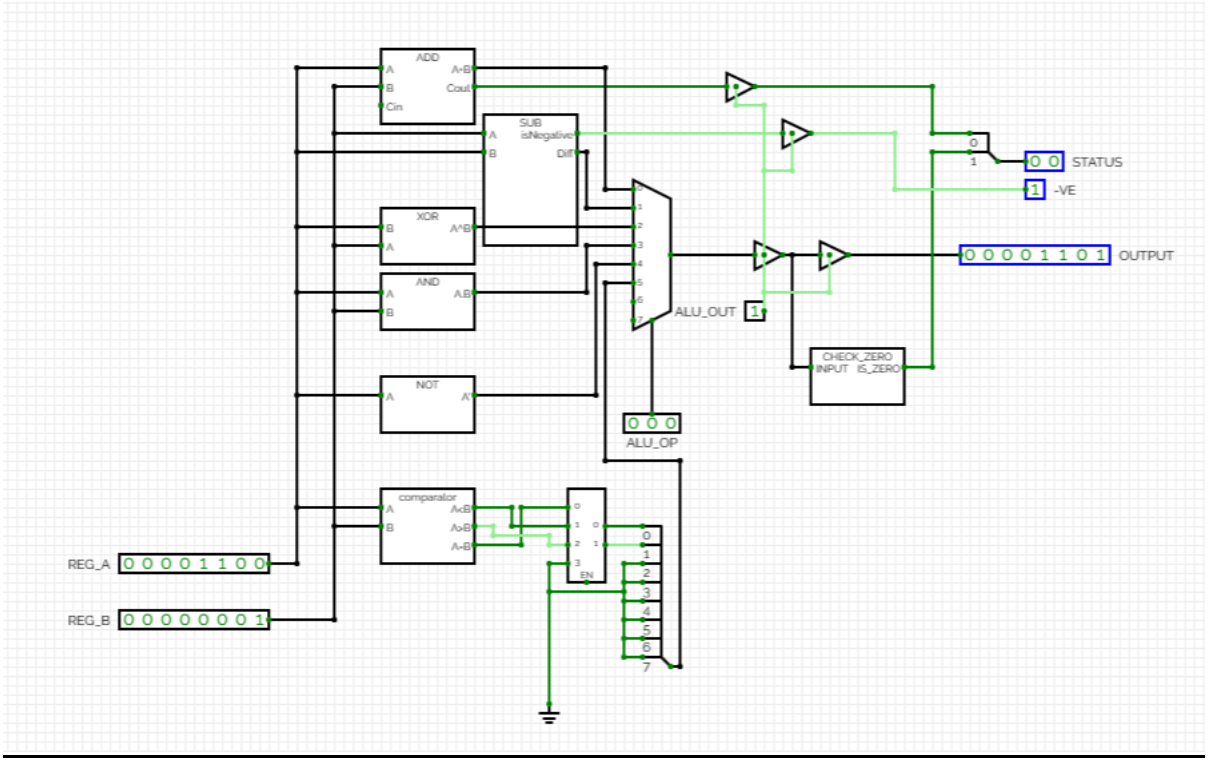
This program counter is a 4-bit up counter. The flip-flops are arranged from left to right indicating the LSB to MSB respectively. The complement of the values in the flip-flops gives us the actual count of the counter. (This is essentially the complement of a 4-bit down counter).

By default, the preset is set to 1. However, the preset value changes, when Load is set to 1. This is done to load an external value (we have used NOT gates because of the fact that the counter actually is the complement of our actual counter value). The bit range [4:7] remains unused and so is just stored in the flip flop that we have over there (to prevent it from getting a don't care value when OUT = 0 and LOAD = 0). OUT controls whether we want to send the stored program count value out or not.

Reset sets the value in the preset. When Load is 0, then Reset sets the counter to 0. But if Load is 1, then Load also invokes the reset function, and resets the counter to the value sent in the input (described above, by changing the preset value).

Note that the Program counter is positive-edge triggered.

❖ **Arithmetic & Logic Unit (ALU)**



The operation of the ALU depends on the input ALU_OP.

000 - addition,

001 - subtraction,

010 - XOR of A and B,

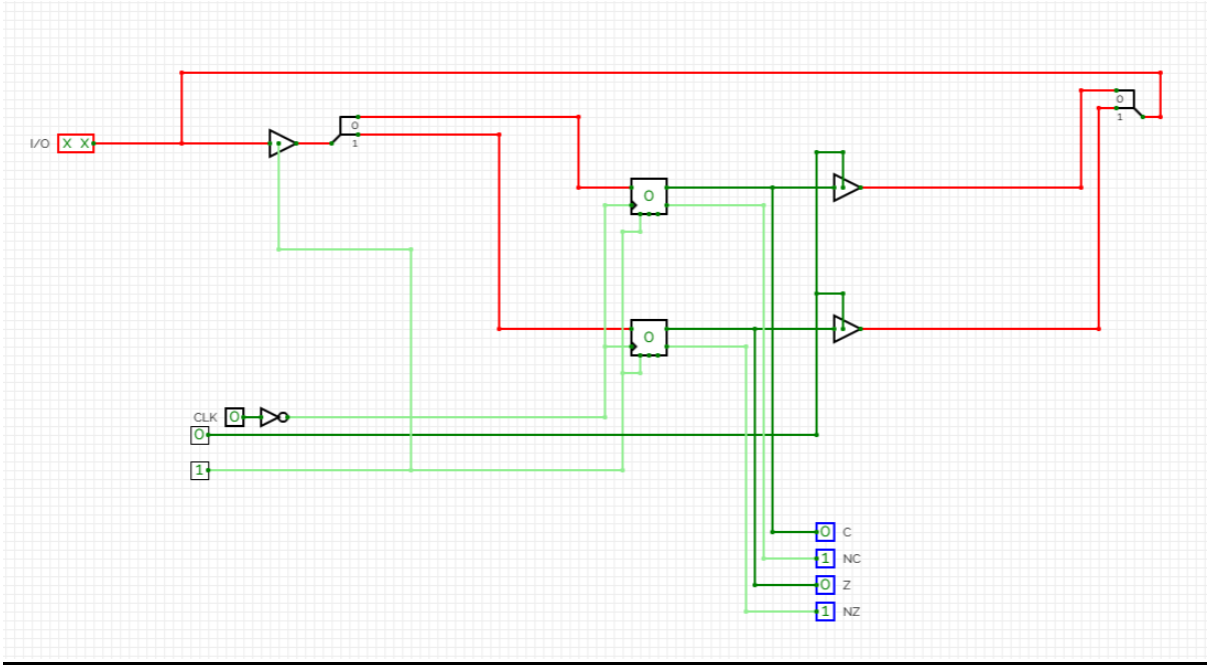
011 - AND of A and B,

100 - the output displays A'(complement of A),

101 - the output shows 0 if A = B, 1 if A < B and 2 if A > B.

Whenever ALU_OUT is 1, the ALU sends the calculated value to the register, which has its IN as 1.

❖ **Status Register**



The status register displays the status of the carry and output of the ALU. If carry of SUM of A and B is 1, C = 1 and NC = C' = 0 and vice versa. Similarly, if OUTPUT = 0, Z = 0 and NZ = Z' = 1 and vice versa.

The status register reads input only when it's sent some fresh input from the ALU using ALU_OUT = 1. The D Flip-flops store the value and is held from being outputted to the same bus using the tristate buffers which are permanently made inactive.

INSTRUCTION SET

I. NOP – No Operation

This instruction performs a single cycle No Operation.

Operation: Nil

Syntax: NOP

Opcode: 0000 XXXX

II. LDA – Load *A* direct

Loads a value form the address ROM to the accumulator.

Operation: A ← M

Syntax: LDA <ADDRESS>

Opcode: 0001 <ADDRESS>

III. STA – Store *A* direct

Store the value of the accumulator in an address.

Operation: M ← A

Syntax: STA <ADDRESS>

Opcode: 0010 <ADDRESS>

IV. ADD – Add without Carry

Adds the present value of accumulator and the value stored at a particular address without the carry and places the result in the accumulator.

Operation: A ← A + B

Syntax: ADD <ADDRESS>

Opcode: 0011 <ADDRESS>

V. SUB – Subtract without Carry

Subtracts the present value of accumulator and the value stored at a particular address and places the result in the accumulator.

Operation: A ← A - B

Syntax: SUB <ADDRESS>

Opcode: 0100 <ADDRESS>

VI. LDI – Load Immediate

Loads an 4-bit constant directly to the accumulator.

Operation: A ← K

Syntax: LDI <VALUE>

Opcode: 0101 <VALUE>

VII. JMP – Jump

Jump to an address within the entire Program memory.

Operation: PC ← K

Syntax: JMP <ADDRESS>

Opcode: 0110 <ADDRESS>

VIII. OUT – Output to Hex Display

Output the the value stored in register C to the Sliding-Hex display.

Operation: SCROLL_DISP ← C

Syntax: OUT

Opcode: 0111 XXXX

IX. JNZ – Jump if Non-Zero

Jump to an address within the entire Program memory if the accumulator stores a non-zero value.

Operation: PC ← K

Syntax: JNZ < ADDRESS >

Opcode: 1000 < ADDRESS >

X. JZ – Jump if Zero

Jump to an address within the entire Program memory if the accumulator stores a zero.

Operation: PC ← K

Syntax: JZ <ADDRESS>

Opcode: 1000 <ADDRESS>

XI. CMP – Compare

Jump to an address within the entire Program memory if the accumulator stores a zero.

Operation: PC ← K

Syntax: JZ < ADDRESS >

Opcode: 1000 < ADDRESS >

XII.MOVAB – Copy Register

This instruction makes a copy of A into B. The source register A is left unchanged, while the destination register B is loaded with a copy of A.

Operation: B ← A

Syntax: MOVAB

Opcode: 1011 XXXX

XIII. MOVBC – Copy Register

This instruction makes a copy of B into C. The source register B is left unchanged, while the destination register C is loaded with a copy of B.

Operation: C ← B

Syntax: MOVBC

Opcode: 1100 XXXX

XIV. MOVCA – Copy Register

This instruction makes a copy of C into A. The source register C is left unchanged, while the destination register A is loaded with a copy of C.

Operation: A ← C

Syntax: MOVCA

Opcode: 1101 XXXX

XV. SWAPAC – Swap A and C

This instruction swaps C and A.

Operation: A ← C, C ← A

Syntax: SWAPAC

Opcode: 1110 XXXX

XVI. HLT - Halt processor

This instruction stops the program counter.

Operation: STOP

Syntax: HLT

Opcode: 1111 XXXX

EXAMPLE PROGRAMS

Example 1:

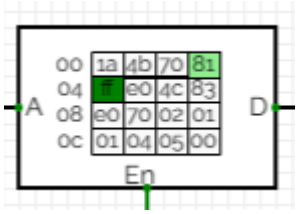
```
LDA 0xa
SUB 0xb
OUT
JNZ 0x1
HLT
```

0xa contains 0x02, and 0xb contains 0x01. Program will output 1,0 in the scrolling display.

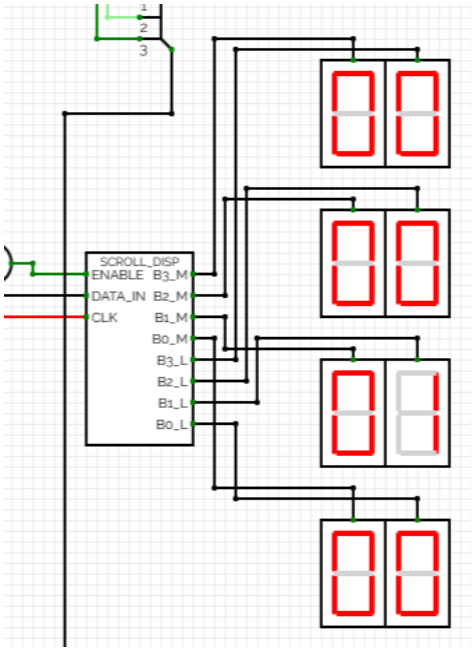
Equivalent C++ program:

```
int x = 2;
while (x != 0)
{
 x -= 1;
 cout << x << endl;
}
```

MEMORY:



OUTPUT:



01, followed by 00 was displayed.

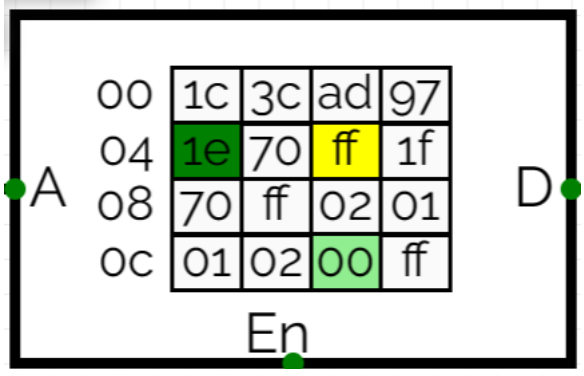
Example 2:

```
LDA 0xc
ADD 0xc
CMP 0xd
JZ 0x7
LDA 0xe
OUT
HLT
LDA 0xf
OUT
HLT
```

Equivalent C++ program:

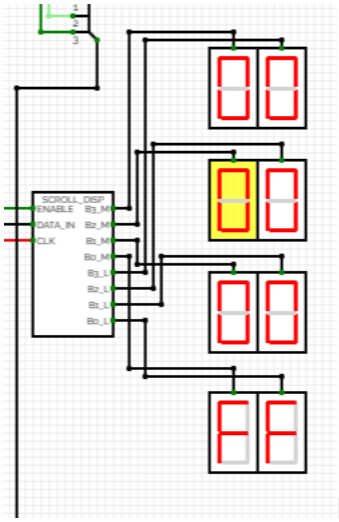
```
int x = 1;
x += 1;
if (x == 2)
{
 cout << ff << endl;
}
else
{
 cout << 00 << endl;
}
```

MEMORY:



OUTPUT:

Program should output an ff, because 01+01 (basically 0xc +0xc == 0xd) equals two, and should go to memory location f and display FF.



FF is correctly displayed.

Microinstructions and Controller Logic Design

The fetch cycle consists of 2 T-states T1 and T2 and is common to all: (some of these are deprecated instructions)

T1: (1 << pc_out | 1 << mar_in)

T2: (1 << ir_in | 1 << mem_out | 1<<pc_inc)

1.NOP: No operation performed

2.LDA:

- T3: (1 << mar_in | 1 << ir_out)
- T4: (1 << mem_out | 1 << a_in)

3.LDA:

- T3: (1 << mar_in | 1 << ir_out)
- T4: (1 << ir_in | 1 << mem_out | 1<<pc_inc)

4.ADD:

- T3: (1 << mar_in | 1 << ir_out)
- T4: (1 << mem_out | 1 << b_in)
- T5: (1 << alu_out | 1 << a_in)

5.SUB:

- T3: (1 << mar_in | 1 << ir_out)

- T4: (1 << mem_out | 1 << b_in)
- T5: (1 << alu_out | 1 << a_in)

6.STA:

- T3: (1 << mar_in | 1 << ir_out)
- T4: (1 << a_out | 1 << mem_in)

7.LDI:

- T3: (1 << ir_out | 1<<a_in)

8.OUT:

- T3: (1 << a_out | 1 << c_in)
- T4: (1 << c_out)

9.JMP:

- T3: (1 << ir_out | 1 << pc_load)

10.JNZ: Same as JMP, but with an extra condition, which is taken care of, by *FLAG*.

11.MOVAB:

- T3: (1 << a_out | 1 << b_in)

12.MOVAC

- T3: (1 << a_out | 1 << c_in)

13.MOVBA:

- T3: (1 << b_out | 1 << a_in)

14.MOVCA:

- T3: (1 << c_out | 1 << a_in)

15.MOVBC:

- T3: (1 << b_out | 1 << c_in)

16.MOVCB:

- T3: (1 << c_out | 1 << b_in)

17.SWAPAC, SWAPBC, SWAPAB: Simply combinations of MOVs.

18.HLT:

- T3: (1<<clk)

19. CMP:
    - T3: (1 << ir_out | 1 << mar_in)
    - T4: (1 << mem_out | 1 << b_in)
    - T5: (1 << alu_out | 1 << a_in)

Due to lack of space in the memory, we only stuck with 3 MOVs, and one SWAP function.

Program demonstrating SWAP and MOVs:

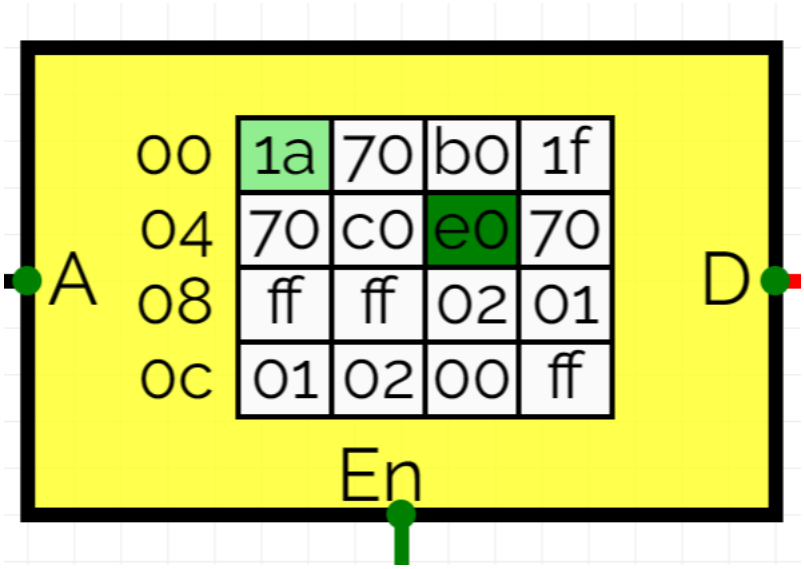
```
LDA 0xa
OUT
MOVAB
LDA 0xf
OUT
MOVBC
SWAPAC
OUT
HLT
```

Equivalent C++ program:

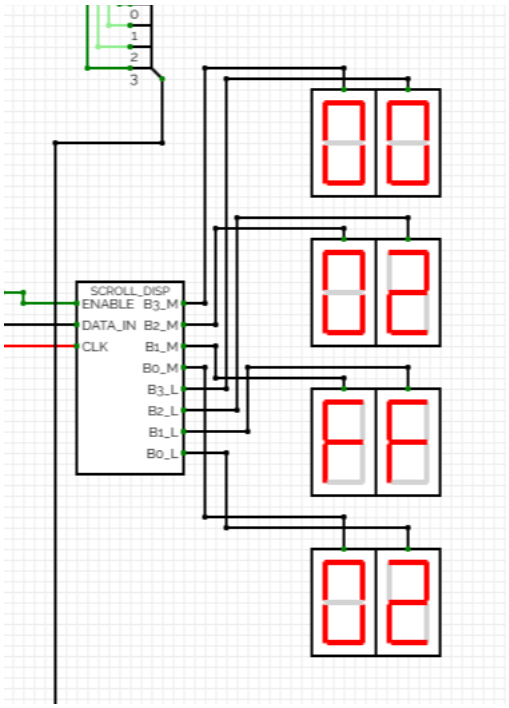
```
int x = 02;
cout << x << endl;
int y = x;
x = ff;
cout << x << endl;
int c = y;
x = c;
cout << x << endl;
```

[Please see next page for memory and output]

MEMORY:



OUTPUT:



The required output is 02, ff, 02. (Note that OUT is connected to register C in this architecture, hence, while OUT-ing, the value of A gets loaded onto C, and the previous value of C would get lost)