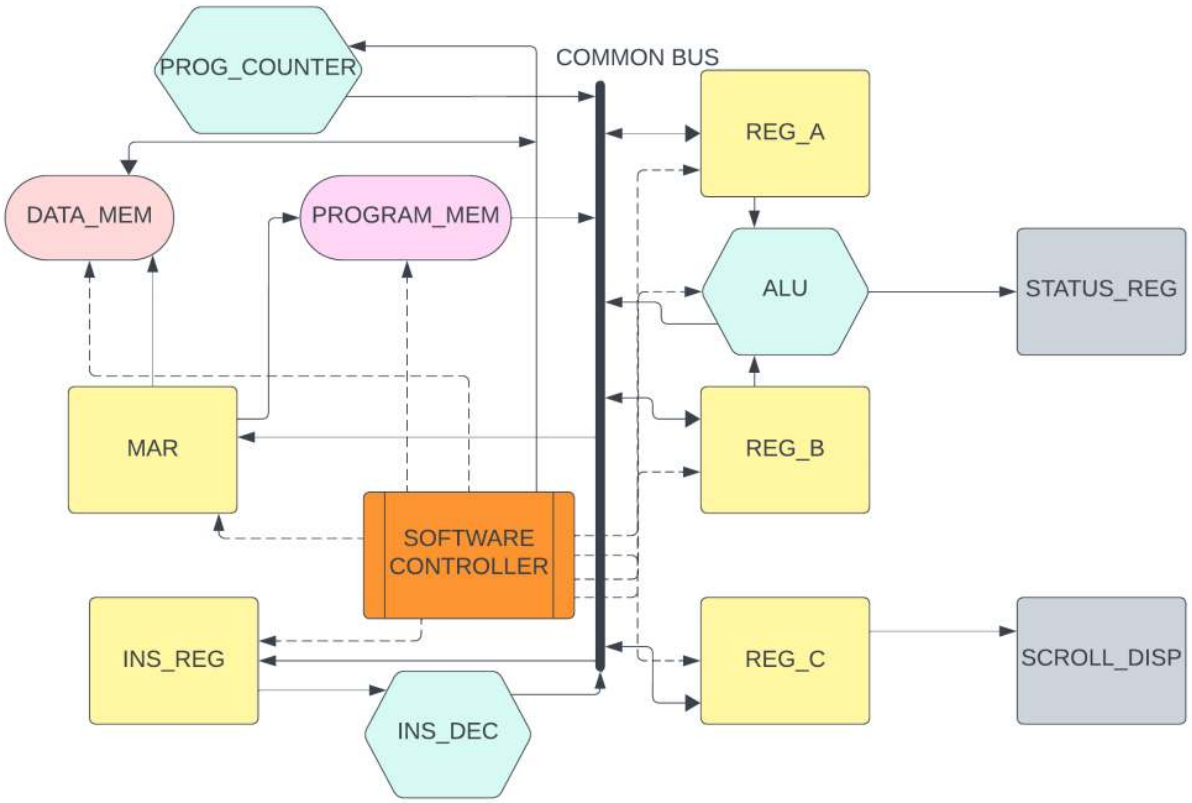
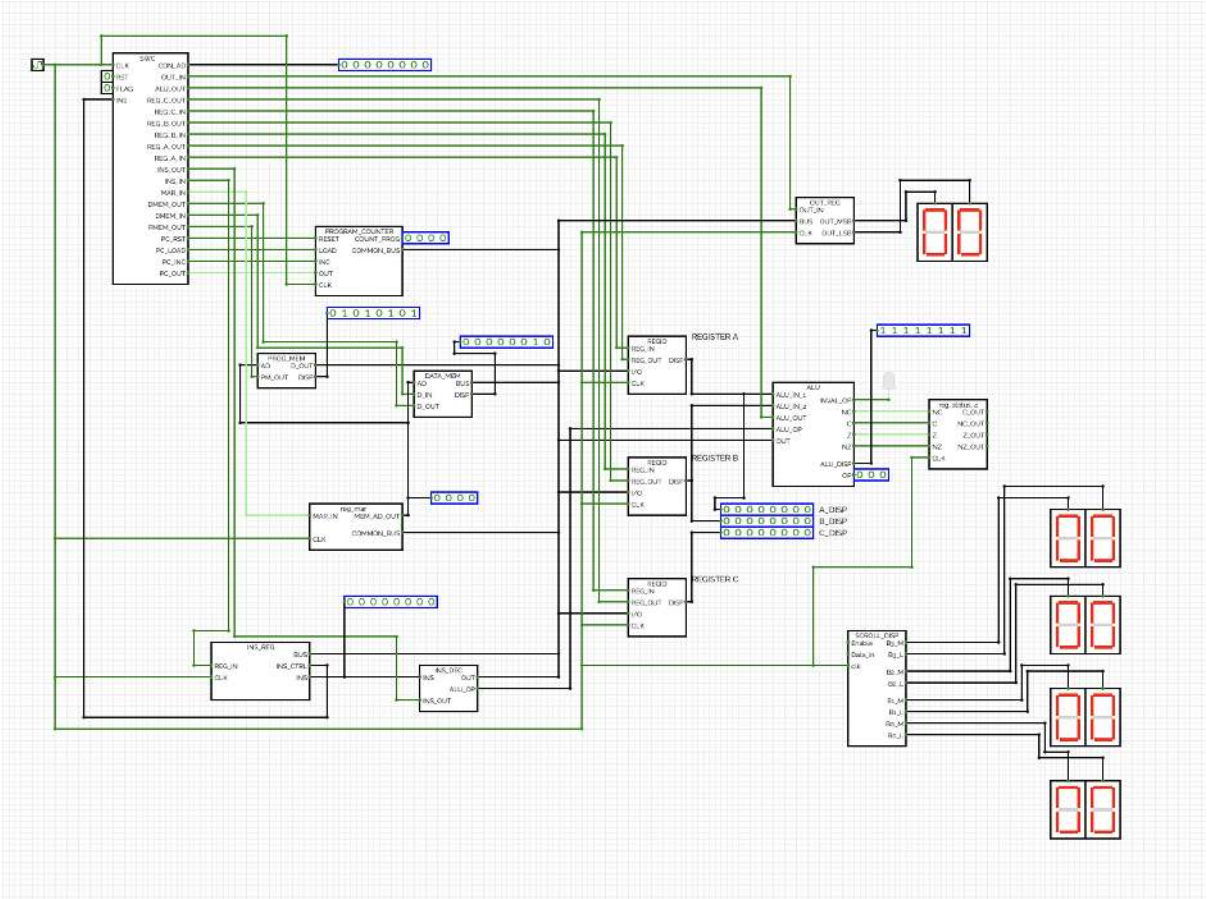
ARCH Gajendra

CS2300 Project

- *Kishore Kumar E (CS22B041)*
- *Nishok RP (CS22B035)*





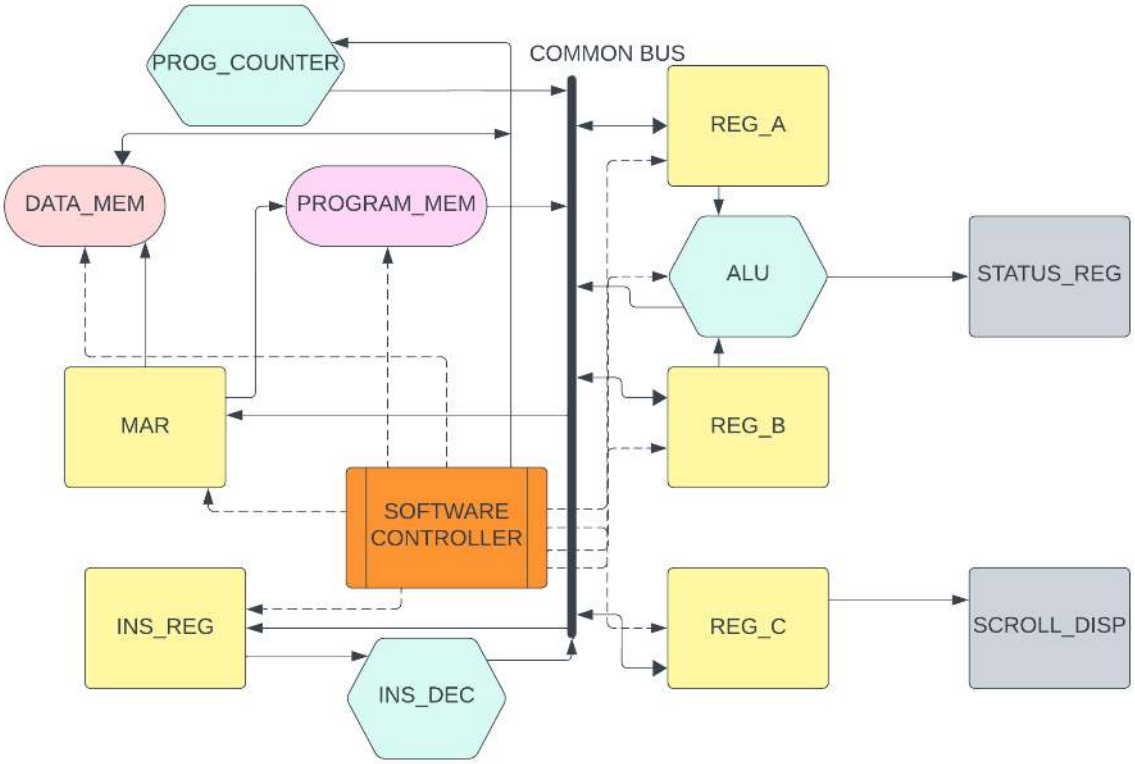
OUR COMPUTER

Section 1.1 : Contents

Section 2 : Features of our version of Gajendra

- A maximum of 16 instructions can be programmed into the Program Memory as there is a separate Data Memory.
- A maximum of 32 (8 bit each) data slots are available in the data memory.
- Three registers, A (the Accumulator), B and C, each of size 8 bit are available for internal working of the computer.
- The Arithmetic Logic Unit (ALU) supports operations like NOT, XOR, AND, Addition, Subtraction, Shift and Comparisons. Multiplication can be achieved by repeated addition in the user end.
- Both Hardware Controller and a Software Controller have been designed as a backup if one of them fails to work and also for our own practice.
- 5 T-states, (T0, T1, T2, T3 and T4) have been implemented in the controllers using a ring counter which is necessary for some operations like SWAP.
- Instruction Decoder is designed to make sure some of the control signals are set all over the timing of the controller to restrict gate delays affecting the working of the Computer.
- Status Register indicates the exceptions received by the ALU by analysing the output of the ALU.
- Scroll Display displays the values loaded to Register C along with the previous 3 values that were stored in Register C.
- Displays have been connected to each and every component of the Computer for effective Debugging.

Section 3 : Explanation of Flow (Data and Control)



1. Initially, the Software Controller is Reset which makes T0 = 1 and rest T-states are set to 0.
2. T0 refers to the 1st Fetch Cycle, where Program Counter(PC) outputs to the Common Bus and Memory Address Register(MAR) takes in this address.
3. T1 refers to the 2nd Fetch Cycle, where the program that MAR points to in PROGRAM_MEM is loaded into the Instruction Register(IR).
4. The Instruction Decoder (INS_DEC) takes in the instruction and makes sure the ALU operation (ALU_OP) is set as desired. It also performs other functions which are mentioned in <IR module desc>.
5. The Software Controller (SWC) takes in the Instruction from IR and processes what has to happen in the next 3 T-states of the RIng Counter.
6. According to the Instruction, the Software Controller outputs control signals to different modules for them to act the way the Instruction says it to. <Explained better in Programs>

Section 4: The Instruction Set

An Instruction in the Program Memory consists of 8 bits, the first 4 bits refer to the instruction and the last 4 bits refer to data or an address in the Data Memory.
As a result, we have 2^4 = 16 instructions.

4.1 NOP (No Operation)

Does nothing and moves on to the next instruction.

1) Syntax : NOP
2) OPT Code : 0000 XXXX (All Xs in the instructions are Don't Cares)
3) Operands : NIL
4) Operation : NIL
5) Program Counter : PC <- PC + 1

4.2 LDA (Load Accumulator A)

Loads the value in the given address of the Data Memory into Accumulator (REG_A).

1) Syntax : LDA AD (AD refers to Address in Machine Code)
2) OPT Code : 0001 AAAA (A refers to address bit)
3) Operands : 0 <= AD <= 15
4) Operation : A <- *AD (*AD refers to the data stored in address AD of DATA_MEM)
5) Program Counter : PC <- PC + 1

4.3 STA (Store Accumulator A)

Loads the value in the given address of the Data Memory with Accumulator data.

1) Syntax : STA AD
2) OPT Code : 0010 AAAA
3) Operands : 0 <= AD <= 15
4) Operation : AD <- A
5) Program Counter : PC <- PC + 1

4.4 ADD

Loads the value in Accumulator as the sum of itself and the value at the address specified in DATA_MEM.

1) Syntax : ADD AD
2) OPT Code : 0011 AAAA
3) Operands : 0 <= AD <= 15
4) Operation : A <- *AD + A
5) Program Counter : PC <- PC + 1

4.5 SUB

Loads the value in Accumulator as the difference of itself and the value at the address specified in DATA_MEM.

1) Syntax : SUB AD
2) OPT Code : 0100 AAAA
3) Operands : 0 <= AD <= 15
4) Operation : A <- A - *AD
5) Program Counter : PC <- PC + 1

4.6 LDI (Load Immediate)

Loads the value in Accumulator with the last 4 bits of the instruction, the Least Significant Half.

1) Syntax : LDI DT (DT refers to Data)
2) OPT Code : 0101 DDDD
3) Operands : 0 <= DT <= 15
4) Operation : A <- DT
5) Program Counter : PC <- PC + 1

4.7 OUT

Loads the value in Accumulator to the Output Register.

1) Syntax : OUT
2) OPT Code : 0110 XXXX
3) Operands : NIL
4) Operation : OUT <- A
5) Program Counter : PC <- PC + 1

4.8 JMP (Jump)

Loads the Program Counter with the given address, unconditionally.

1) Syntax : JMP AD
2) OPT Code : 0111 AAAA
3) Operands : 0 <= AD <= 15
4) Operation : NIL
5) Program Counter : PC <- AD

4.9 JNZ (Jump Conditional)

Loads the Program Counter with the given address, if controller Flag is set to 1.

1) Syntax : JNZ AD
2) OPT Code : 1000 AAAA
3) Operands : 0 <= AD <= 15
4) Operation : NIL
5) Program Counter : PC <- AD (If F == 1)

4.10 SWAP

Swaps the values stored in Register A and Register B.

1) Syntax : SWAP
2) OPT Code : 1001 XXXX
3) Operands : NIL
4) Operation : C <- A followed by A <- B followed by B <- C
5) Program Counter : PC <- PC + 1

4.11 MOVAB

Loads Register B with the value stored in Accumulator.

1) Syntax : MOVAB
2) OPT Code : 1010 XXXX
3) Operands : NIL
4) Operation : B <- A
5) Program Counter : PC <- PC + 1

4.12 MOVAC

Loads Register C with the value stored in Accumulator.

1) Syntax : MOVAC
2) OPT Code : 1011 XXXX
3) Operands : NIL
4) Operation : C <- A
5) Program Counter : PC <- PC + 1

4.13 MOVBA

Loads Accumulator with the value stored in Register B.

1) Syntax : MOVBA
2) OPT Code : 1100 XXXX
3) Operands : NIL
4) Operation : A <- B
5) Program Counter : PC <- PC + 1

4.14 MOVBC

Loads Accumulator with the value stored in Register B.

1) Syntax : MOVBC
2) OPT Code : 1101 XXXX
3) Operands : NIL
4) Operation : C <- B
5) Program Counter : PC <- PC + 1

4.15 MOVCB

Loads Register B with the value stored in Register C.

1) Syntax : MOVCB
2) OPT Code : 1110 XXXX
3) Operands : NIL
4) Operation : B <- C
5) Program Counter : PC <- PC + 1

4.16 HALT

Resets the Program Counter to 0000 and the Program repeats itself.

1) Syntax : HALT
2) OPT Code : 1111 XXXX
3) Operands : NIL
4) Operation : NIL
5) Program Counter : PC <- 0000

Note : When the HALT is encountered, the program repeats rather than stopping as the name suggests. This has been implemented for a better visualisation of the program.

Instead it could be modified by passing the CLK as (CLK(the actual clock) AND STOP Register). When HALT is encountered, this STOP Register can be set to 0 after which the CLK passed to the Program Counter will stop ticking and the program ends.

To achieve this in the current implementation, the RESET button of Controller can be set to 1. This essentially stops the ring counter inside the controller.

Section 5: Micro Instructions

NOP:
T0:1<<PC_OUT | 1<<MAR_IN
T1:1<<PC_inc | 1<<MEM_out | 1<<IR_in

LDA:
T0:1<<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

ADD:
T0:1<<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<<ALU_out | 1<<REGA_in

SUB:
T0:1<<PC_OUT | 1<<MAR_IN
T1:1<<PC_inc | 1<<MEM_out | 1<<IR_in
T2:1<<MAR_in | 1<<INS_OUT
T3:1<<DM_out | 1<<REGB_in
T4:1<<REGA_IN<<ALU_OUT

LDI:
T0:1<<PC_OUT | 1<<MAR_IN
T1:1<<PC_inc | 1<<MEM_out | 1<<IR_in
T2:1<<INS_OUT | 1<<REGA_IN

OUT:
T0:1<<PC_OUT | 1<<MAR_IN
T1::1<<PC_inc | 1<<MEM_out | 1<<IR_in
T2:1<<REGA_OUT | 1<< OUT_IN

JMP:
T0:1<<PC_OUT | 1<<MAR_IN
T1:1<<PC_inc | 1<<MEM_out | 1<<IR_in
T2:1<<LOAD | 1<<INS_OUT

JNZ:
T0:1<<PC_OUT | 1<<MAR_IN
T1:1<<PC_inc | 1<<MEM_out | 1<<IR_in
T2:1<<LOAD | 1<<INS_OUT (IF FLAG == 1)

SWAP:
T0:1<<PC_OUT | 1<<MAR_IN
T1:1<<PC_inc | 1<<MEM_out | 1<<IR_in
T2:1<<REGA_OUT | 1<<REGC_IN
T3:1<<REGA_IN | 1<<REGB_OUT
T4:1<<REGB_IN | 1<<REGC_OUT

MOV AB
T0:1<<PC_OUT | 1<<MAR_IN
T1:1<<PC_inc | 1<<MEM_out | 1<<IR_in
T2:1<<REGA_OUT | 1<<REGB_IN

MOV AC
T0:1<<PC_OUT | 1<<MAR_IN
T1:1<<PC_inc | 1<<MEM_out | 1<<IR_in
T2:1<<REGA_OUT | 1<<REGC_IN

MOV BA
T0:1<<PC_OUT | 1<<MAR_IN
T1:1<<PC_inc | 1<<MEM_out | 1<<IR_in
T2:1<<REGB_OUT | 1<<REGA_IN

MOV BC
T0:1<<PC_OUT | 1<<MAR_IN
T1:1<<PC_inc | 1<<MEM_out | 1<<IR_in
T2:1<<REGB_OUT | 1<<REGC_IN

MOV CB
T0:1<<PC_OUT | 1<<MAR_IN
T1:1<<PC_inc | 1<<MEM_out | 1<<IR_in
T2:1<<REGC_OUT | 1<<REGB_IN

HALT
T0:1<<PC_OUT | 1<<MAR_IN
T1:1<<PC_inc | 1<<MEM_out | 1<<IR_in

T2:1<<RST

We need a maximum of 5 T-states, which is for swap.

Section 6: The Instruction Routines

Refer:

https://docs.google.com/spreadsheets/d/1Gir48bnO4zN9vgMmWHv8s2wTx4zOdztkxXkNTxjRDyA/edit?usp=sharing)

	PC				PM	DM		MAR	IR		Data Registers						ALU	OR
	O	IN	L	R	O	I	O	I	I	O	I	O	I	O	I	O	O	I
T0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
T1	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
NOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LDA	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
STA	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
ADD	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
SUB	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
LDI	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
OUT	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1
JMP	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
JNZ	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
SWAP	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
MOV AB	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
MOV AC	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
MOV BA	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0
MOV BC	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0

	PC				PM	DM		MAR	IR		Data Registers						ALU	OR
	O	IN	L	R	O	I	O	I	I	O	I	O	I	O	I	O	O	I
MOV CB	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	**1**	0	0
HALT	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Note - I/O denotes Input and Output. For PC, L dentes LOAD, R denotes RESET and IN denotes INC

Note- For each Instruction, the first row denotes T2 state and so on, since the fetch cycles T0 and T1 are common to all.

Section 7: Machine Language to Assembly Code Converter

Code reference:
https://colab.research.google.com/drive/147ukic09iyaXmWMBRoZqui6Q6-T9GZVe?usp=sharing

```
def Ins(instruction):
  instruction_dict = {
      "NOP": "0",
      "LDA": "1",
      "STA": "2",
      "ADD": "3",
      "SUB": "4",
      "LDI": "5",
      "OUT": "6",
      "JMP": "7",
      "JNZ": "8",
      "SWAP": "9",
      "MOVAB": "A",
      "MOVAC": "B",
      "MOVBA": "C",
      "MOVBC": "D",
      "MOVCB": "E",
      "HALT": "F"
  }
  return instruction_dict.get(instruction, "-")

def main():
  prog_mem = ""
  while True:
      input_str = input("Enter a Machine Instruction (or type 'END' to finish): ")

      words = input_str.split()
      instruction = words[0]
      if len(words) == 2:
        data_ad = words[1]
        data_ad = data_ad[2:]
        prog_mem += "0x" + Ins(instruction).lower() + data_ad.lower() + "\n"
      else:
        prog_mem += "0x" + Ins(instruction).lower() + "0" + "\n"
      #print(Ins(instruction))

      if input_str == "HALT":
        break
```

```
  print(prog_mem)

if __name__ == "__main__":
    main()
```

Section 8: Running Different Programs

Note : The programs are loaded in the corresponding computer files in the **software controlled version.**

1) **Adding Two Numbers** *(Refer to final.cv)*

   Machine Code :

   ```
   LDA 0x0
   ADD 0x1
   OUT
   HALT
   ```

   Assembly Code: (Loaded in PMEM)

   ```
   0x10
   0x31
   0x60
   0xf0
   ```

   Memory Preset: Store any two numbers in DATA_MEM 0x0 and 0x1 (say p and q)

   Result : Outputs p + q in the Output Panel. Accumulator also stores p + q after the program.

2) **Adding and Subtracting Number in a sequence***(Refer to finalcombosum.cv)*

   Machine Code :

   ```
   LDA 0x0
   SUB 0x1
   ADD 0x2
   SUB 0x3
   OUT
   HALT
   ```

   Assembly Code: (Loaded in PMEM)

   ```
   0x10
   0x41
   0x32
   0x43
   0x60
   0xf0
   ```

   Memory Preset: Store any four numbers in DATA_MEM 0x0, 0x1, 0x2 and 0x3/ (say p, q, r and s)

Result : Outputs p - q + r - s in the Output Panel. Accumulator also stores p - q + r - s after the program.

3) **RepeatSwap23** *(Refer to repeatswapsw.cv)*

Machine Code :

```
 LDI 0x3
MOVAB
LDI 0x2
OUT
SWAP
JMP 0x3
HALT
```

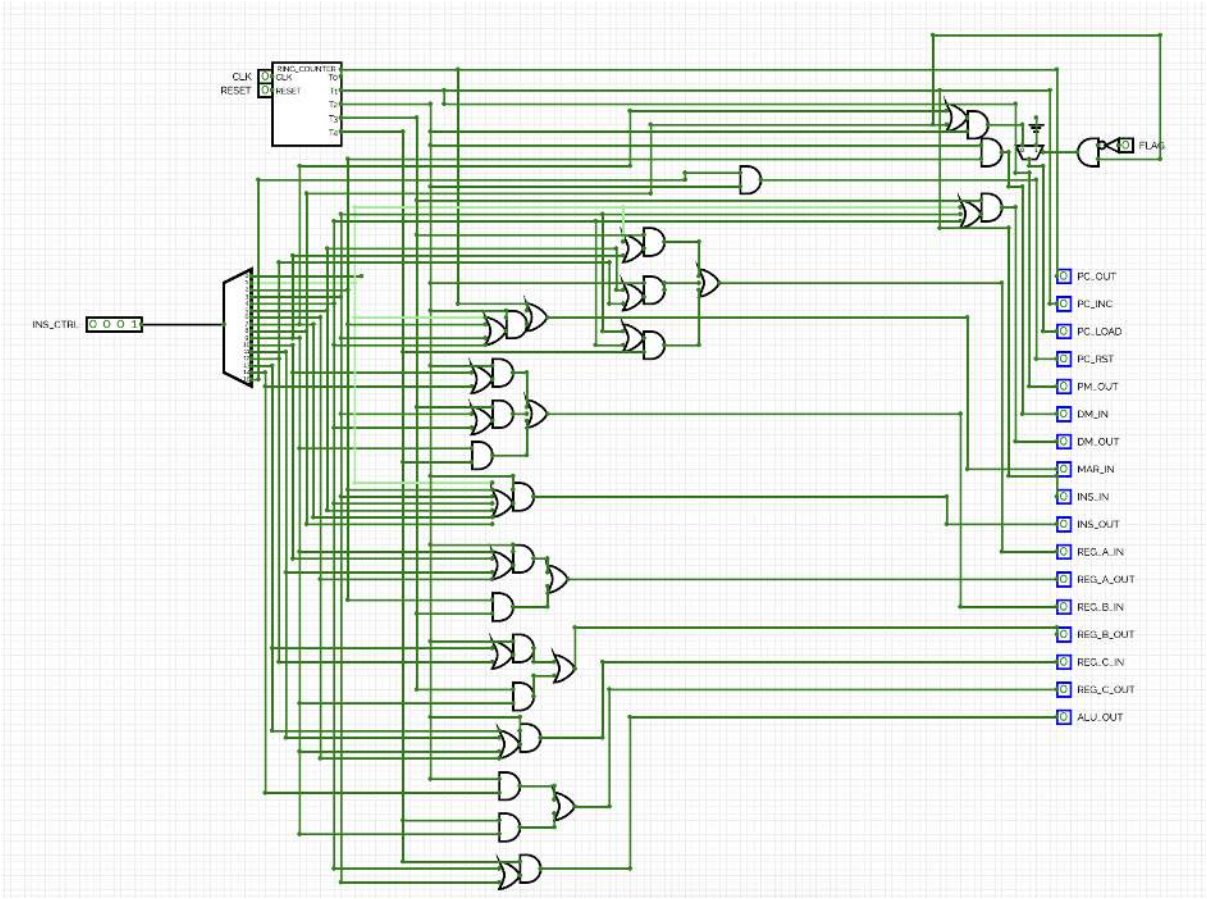
Assembly Code: (Loaded in PMEM)

```
0x53
0xa0
0x52
0x60
0x90
0x73
0xf0
```

Memory Preset : NIL

Result : 2 and 3 swapping alternatively between REG_A and REG_B in a forever while loop

Section 9: Hardware Controller



Truth Table (Note that each functionality given below starts after the two fetch cycles mentioned in the first two rows)

	Program Counter				Program Memory	Data Memory		Memory Address Register	Instruction Register	
	OUT	INC	LOAD	RST	PM_OUT	DM_IN	DM_OUT	MAR_IN	INS_IN	INS_OUT
Fetch1 (T0)	1	0	0	0	0	0	0	1	0	0
Fetch2	0	1	0	0	1	0	0	0	1	0
NOP(0000)										
LDA (0001 AAAA)	0	0	0	0	0	0	0	1	0	1
	0	0	0	0	0	0	1	0	0	0
STA(0010 AAAA)	0	0	0	0	0	0	0	1	0	1
	0	0	0	0	0	1	0	0	0	0
ADD(0011 AAAA)	0	0	0	0	0	0	0	1	0	1
	0	0	0	0	0	0	1	0	0	0
	0	0	0	0	0	0	0	0	0	0
SUB(0100 AAAA)	0	0	0	0	0	0	0	1	0	1
	0	0	0	0	0	0	1	0	0	0
	0	0	0	0	0	0	0	0	0	0
LDI(0101 DDDD)	0	0	0	0	0	0	0	0	0	1
OUT(0110 XXXX)	0	0	0	0	0	0	0	0	0	0

	Program Counter				Program Memory	Data Memory		Memory Address Register	Instruction Register	
	OUT	INC	LOAD	RST	PM_OUT	DM_IN	DM_OUT	MAR_IN	INS_IN	INS_OUT
JMP (0111 AAAA)	0	0	1	0	0	0	0	0	0	1
JNZ (1000 AAAA)	0	0	1	0	0	0	0	0	0	1
SWAP(1001 DDDD)	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
MOV AB (1010 XXXX	0	0	0	0	0	0	0	0	0	0
MOV AC (1011 XXXX	0	0	0	0	0	0	0	0	0	0
MOV BA (1100 XXXX	0	0	0	0	0	0	0	0	0	0
MOV BC(1101 XXXX)	0	0	0	0	0	0	0	0	0	0
MOV CB(1110 XXXX)	0	0	0	0	0	0	0	0	0	0
HALT(1111 XXXX)	0	0	0	1	0	0	0	0	0	0

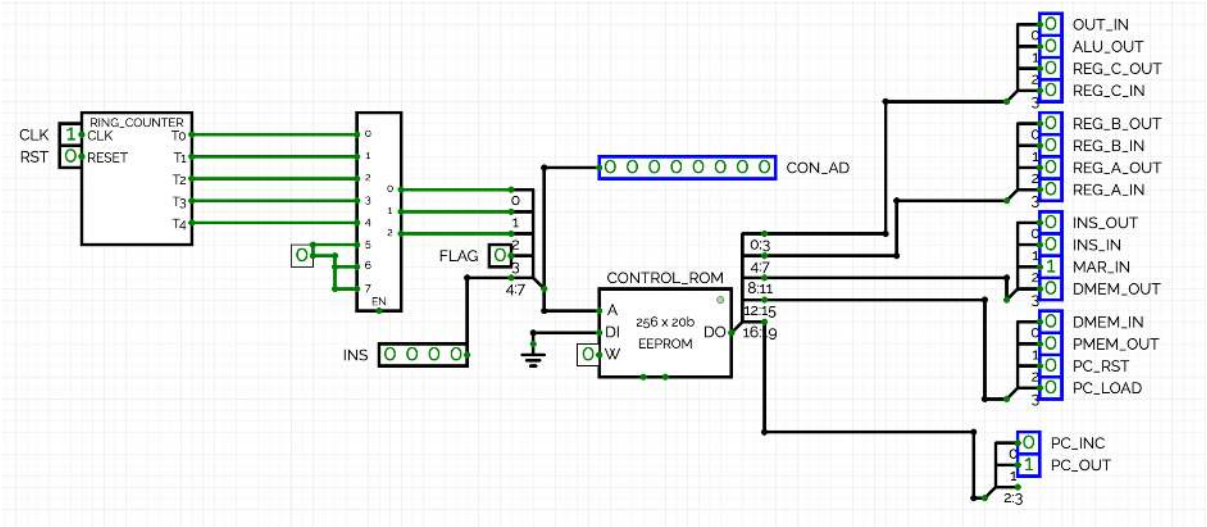
	Data Registers						ALU	
	REGA_IN	REGA_OUT	REGB_IN	REGB_OUT	REGC_IN	REGC_OUT	ALU_OUT	OUT_IN
Fetch1 (T0)	0	0	0	0	0	0	0	0
Fetch2	0	0	0	0	0	0	0	0
NOP(0000)								
LDA (0001 AAAA)	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0
STA(0010 AAAA)	0	0	0	0	0	0	0	0
	0	1	0	0	0	0	0	0
ADD(0011 AAAA)	0	0	0	0	0	0	0	0
	0	0	1	0	0	0	0	0
	1	0	0	0	0	0	1	0
SUB(0100 AAAA)	0	0	0	0	0	0	0	0
	0	0	1	0	0	0	0	0
	1	0	0	0	0	0	1	0
LDI(0101 DDDD)	1	0	0	0	0	0	0	0
OUT(0110 XXXX)	0	1	0	0	1	0	0	1

	Data Registers						ALU	
	REGA_IN	REGA_OUT	REGB_IN	REGB_OUT	REGC_IN	REGC_OUT	ALU_OUT	OUT_IN
JMP (0111 AAAA)	0	0	0	0	0	0	0	0
JNZ (1000 AAAA)	0	0	0	0	0	0	0	0
SWAP(1001 DDDD)	0	1	0	0	1	0	0	0
	1	0	0	1	0	0	0	0
	0	0	1	0	0	1	0	0
MOV AB (1010 XXXX	0	1	1	0	0	0	0	0
MOV AC (1011 XXXX	0	1	0	0	1	0	0	0
MOV BA (1100 XXXX	1	0	0	1	0	0	0	0
MOV BC(1101 XXXX)	0	0	0	1	1	0	0	0
MOV CB(1110 XXXX)	0	0	1	0	0	1	0	0
HALT(1111 XXXX)	0	0	0	0	0	0	0	0

SOPs of Individual Control Signals (JMP / JNZ are for example decoded using INS code of 4 bits as shows in the circuit)

- **OUT** - T0
- **INC** -T1
- **PC_LOAD** - JMP OR JNZ
- **PC_RET** - HALT
- **PM_OUT** - T1
- **DM_IN** - T3 AND STA
- **DM_OUT** - T3 AND (LDA OR ADD OR SUB)
- **MAR_IN** - T0 OR (T2 AND(LDA OR STA OR ADD OR SUB))
- **INS_IN** - T1
- **INS_OUT** - T2 AND (LDA OR STA OR ADD OR SUB OR SUB OR LDI OR JMP OR JNZ)
- **REGA_IN** - T3(AND(LDA SWAP)) OR (T2 AND(LDI MOV_BA)) OR (T4 AND(ADD OR SUB))
- **REGA_OUT** - (T2(AND(MOV_AC OR MOV_AB OR SWAP OR OUT)) OR (T3 AND STA)
- **REGB_IN** - T2(AND(MOV_CB,MOV_AB) OR T3 AND(ADD OR SUB) OR T4 AND SWAP OR T3 AND SWAP
- **REGB_OUT** - T2 AND(MOV_BC OR MOV_BA) OR T3 AND SWAP
- **REGC_IN** - T2 AND MOV_CB OR T4 AND SWAP
- **REGC_OUT** - T2 AND MOV_CB OR T4 AND SWAP
- **ALU_OUT** - T4 AND(ADD OR SUB)
- **OUT_IN** - T2 AND OUT

Section 10: Software Controller



How the Control ROM is built?

https://docs.google.com/spreadsheets/d/1LCxr4U4q90d7B5hXfeZPO0wUye1pitOnagIR8pElNsA/edit?usp=sharing

- T states are encoded into 3 bits.
- Instructions consist of 4 bits.
- Flag is one bit. Together they comprise 8 bits.
- In the left most column, 005 refers to 0th instruction (NOP), flag = 0 and Tstate = 5
- For each such state, a control word in hex is generated by combining the control signals and stored in a control ROM.

Details								
Flag	5th bit							
Instruction	1-4 bits							
T state	6 - 8 bits	T0 - 0	T1 - 1	T2 - 2	T3 - 3	T4 - 4		
		Program Counter			Program_MEM	Data_MEM		MAR
	OUT	INC	LOAD	RST	MEM_OUT	MEM_IN	MEM_OUT	MAR_IN
000	1	0	0	0	0	0	0	1
001	0	1	0	0	1	0	0	0
002	0	0	0	0	0	0	0	0
003	0	0	0	0	0	0	0	0
004	0	0	0	0	0	0	0	0
005	0	0	0	0	0	0	0	0
006	0	0	0	0	0	0	0	0
007	0	0	0	0	0	0	0	0

Example : LDA

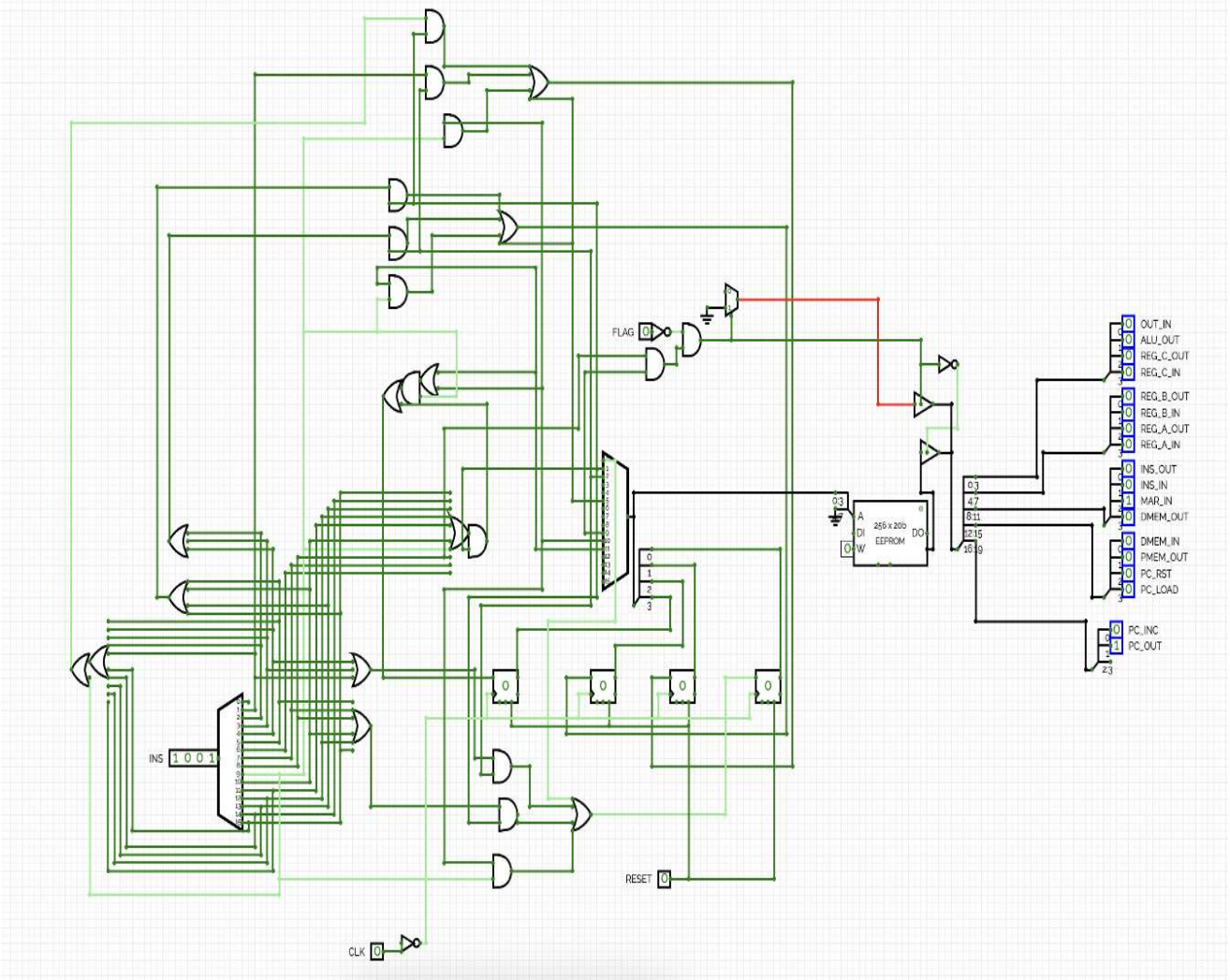
0x12200,
0x00500,
0x00880,

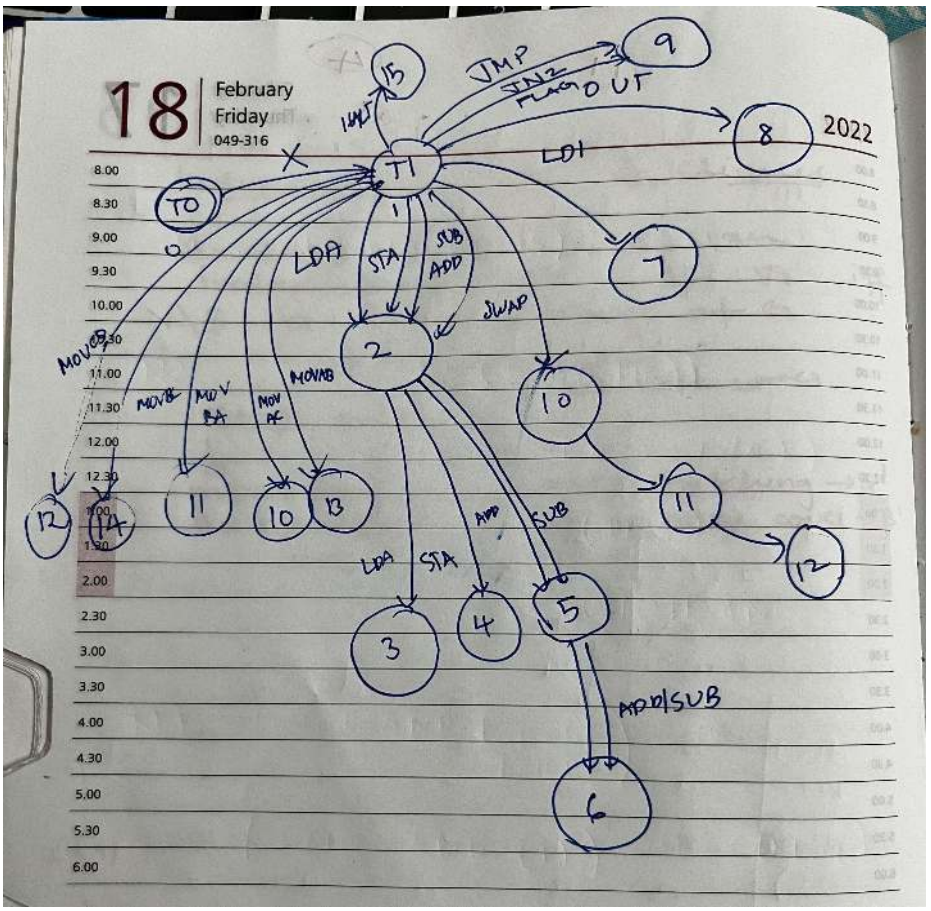
0x00000,
0x00000,
0x00000,
0x00000,

What each control word means?

- Fetch cycle 1 - **0X20400**
- Fetch cycle 2- **0X12200**
- MAR_IN =1 , INS_OUT=1 - **0X00500**
- A_IN=1 , INS_OUT=1 - **0X00880**
- A_OUT=1 , DMEM_IN=1 - **0X01040**
- B_IN=1 , DMEM_OUT=1 - **0X00820**
- ALU_OUT=1 , A_IN=1 - **0X00082**
- INS_OUT=1 , A_IN=1 - **0X00180**
- ALU_OUT=1 , OUT_IN=1 - **0X00041**
- INS_OUT=1 - **0X08100**
- A->C - **0X00048**
- B->A - **0X00090**
- C->B - **0X00024**
- A->B - **0X00060**
- B->C - **0X00018**
- PC_RST - **0X04000**

Section 11: Implementation of Finite State Machine




18
February
Friday
049-316
2022
8.00
8.30
9.00
9.30
10.00
10.30
11.00
11.30
12.00
12.30
1.00
1.30
2.00
2.30
3.00
3.30
4.00
4.30
5.00
5.30
6.00
T0
T1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
JMP
OUT
LDA
STA
ADD
SUB
JMAP
MOVB
MOVA
ADD/SUB

Fetch cycle 1 - 0x20400 (0)

Fetch cycle 2 - 0x 12288 (1)

2022 February Saturday 050-315 | 19

8.00
8.30 MAR IN INSOUT - 0x00500 (2)
9.00 A IN DMEM OUT - 0x00880 (3)
9.30
10.00 AOUT DMEM IN - 0x0104:0 (4)
10.30
11.00 B IN DMEM OUT - 0x00820 (5)
11.30
12.00 ALU OUT AIN - 0x00082 (6)
12.30
1.00 INS OUT AIN - 0x00180 (7)

OUT IN - 0x0004 (8)

February 051-314 Sunday | 20

~~PC~~ ROAD INS OUT - 0x08100 (9)

A → C - 0x00048 (10)

B → A - 0x00090 (11)

C → B - 0x00024 (12)

A → B - 0x00060 (13)

B → C - 0x00018 (14)

PC RST - 0x0400 (15)

JULY

Week	27	28	29	30	31

AUGUST

Week	32	33	34	35	36
Mon	1	8	15	22	29

SEPTEMBER

Week	36	37	38	39	40
Mon		5	12	19	26
		6	13	20	27

OCTOBER

Week	40	41	42	43	44
Mon	31	3	10	17	24
Tue		4	11	18	25
Wed		5	12	19	26

NOVEMBER

Week	45	46	47	48	49
Mon		7	14	21	28
Tue	1	8	15	22	29
Wed	2	9	16	23	30
Thu	3	10	17	24	

DECEMBER

Week	49	50	51	52	53
Mon		5	12	19	26
Tue		6	13	20	27
Wed		7	14	21	28
Thu	1	8	15	22	29
Fri	2	9	16	23	30
	3	10	17	24	31

SOP FOR THE FSM IMPLEMENTATION

- **D3**: (0001 AND(6 OR 7 OR 8 OR 9 OR 10 OR 11 OR 12 OR 13 OR 14 OR 15)) OR ((8 OR 9) AND(9))
- **D2**: (0001 AND(5 OR 10 OR 12 OR 15)) OR (0010 AND(2 OR 3 OR 4)) OR 0101 OR (1011 AND 9)
- **D1**: (0001 AND(1 OR 2 OR 3 OR 4 OR 5 OR 9 OR 11 OR 12 OR 13 OR 14 OR 15)) OR (0010 AND(1)) OR 0101 IR (1010 AND 9)
- **D0**: 0000 OR (0001 AND(5 OR 7 OR 8 OR 10 OR 12 OR 15)) OR (0010 AND(1 OR 4 OR 3)) OR 1010 AND 9

IMPLEMENTATION OF FSMs on CODE (Refer COMPUTER_FSM in fsm321.cv)

Machine Code :

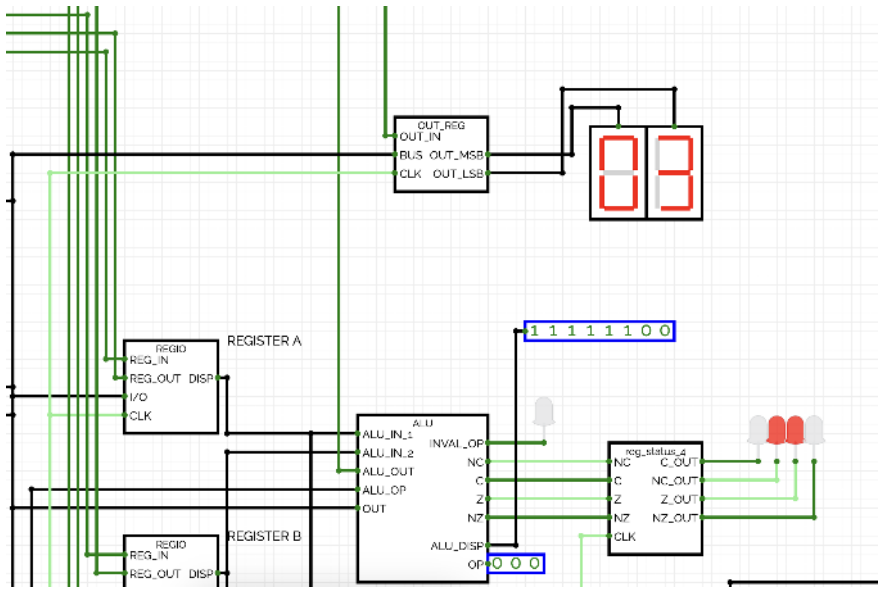
```
LDI 0x3
OUT
LDI 0x2
OUT
LDI 0x1
OUT
LDI 0x0
OUT
JMP 0x0
HALT
```

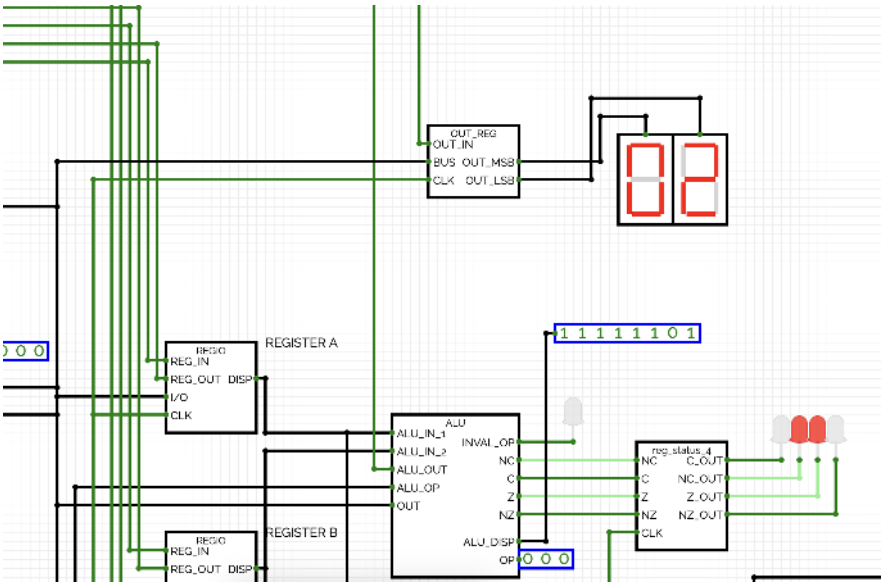
Assembly Code: (Loaded in P_MEM)

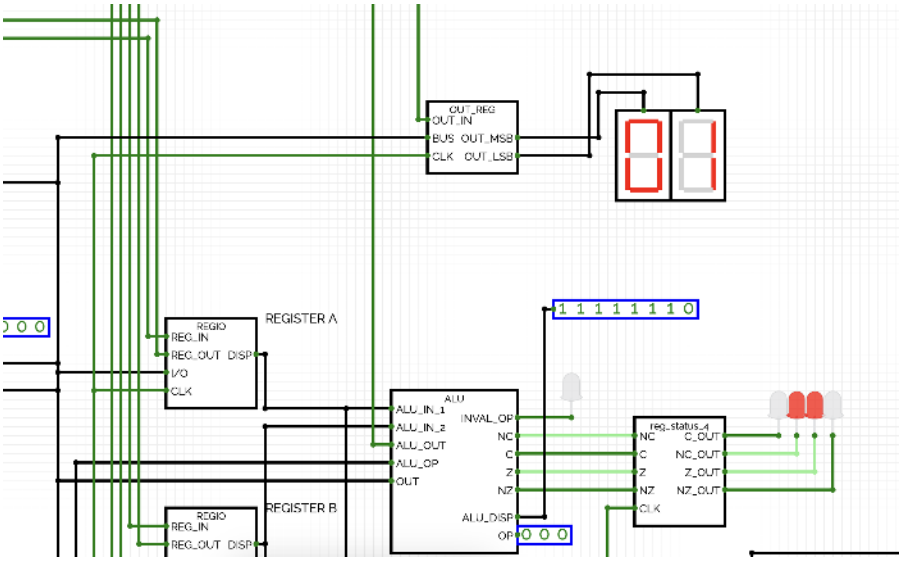
```
0x53
0x60
0x52
0x60
0x51
0x60
0x50
0x60
0x70
0xf0
```

Memory Preset: No memory preset required

Result : 3 -> 2 -> 1 -> 0 -> 3 counter


OUT_REG
OUT_IN
BUS_OUT_MSB
CLK OUT_LSB
REGIO
REG_IN
REG_OUT DISP
I/O
CLK
REGISTER A
REGISTER B
ALU
ALU_IN_1
ALU_IN_2
ALU_OUT
ALU_OP
OUT
INVAL_OP
NC
C
Z
NZ
ALU_DISP
OP
1 1 1 1 1 1 0 0
0 0 0
reg_status_4
C_OUT
NC_OUT
Z_OUT
NZ_OUT
CLK


OUT_REG
OUT_IN
BUS_OUT_MSB
CLK OUT_LSB
REGIO
REG_IN
REG_OUT DISP
I/O
CLK
REGISTER A
REGISTER B
ALU
ALU_IN_1
ALU_IN_2
ALU_OUT
ALU_OP
OUT
INVAL_OP
NC
C
Z
NZ
ALU_DISP
OP
1 1 1 1 1 1 0 1
0 0 0
reg_status_4
C_OUT
NC_OUT
Z_OUT
NZ_OUT
CLK

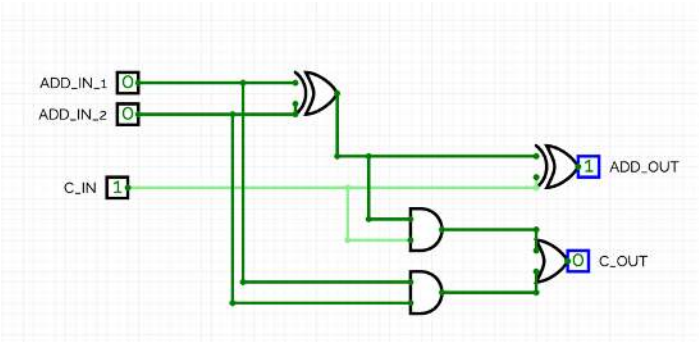

OUT_REG
OUT_IN
BUS_OUT_MSB
CLK OUT_LSB
REGIO
REG_IN
REG_OUT DISP
I/O
CLK
REGISTER A
REGISTER B
ALU
ALU_IN_1
ALU_IN_2
ALU_OUT
ALU_OP
OUT
INVAL_OP
NC
C
Z
NZ
ALU_DISP
OP
1 1 1 1 1 1 1 0
0 0 0
reg_status_4
C_OUT
NC_OUT
Z_OUT
NZ_OUT
CLK

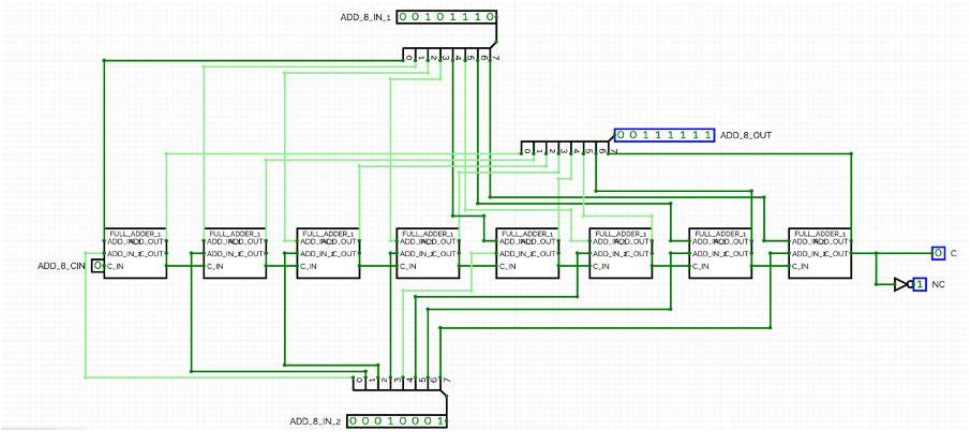
FSM vs Software

- FSM doesn’t waste empt T-cycles, time efficient and less latency.
- Costlier due to the amount of gates used.

Section 12: Explanation of each module

ADDERS





INPUT:

- 8 Bit input data
- C_IN

OUTPUT:

- NC and C (NC is 1 if carry is 0 else C is 1)
- ADD_OUT (The 8 bit output)

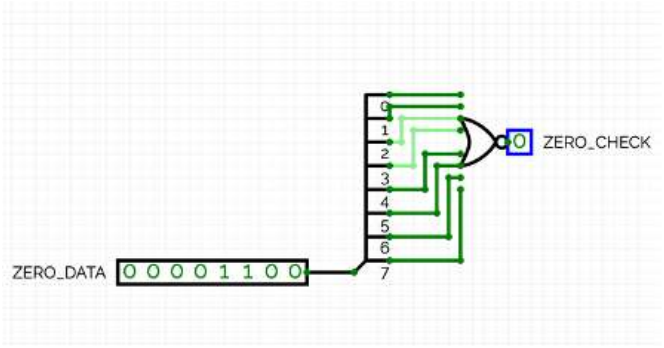
NOTE:

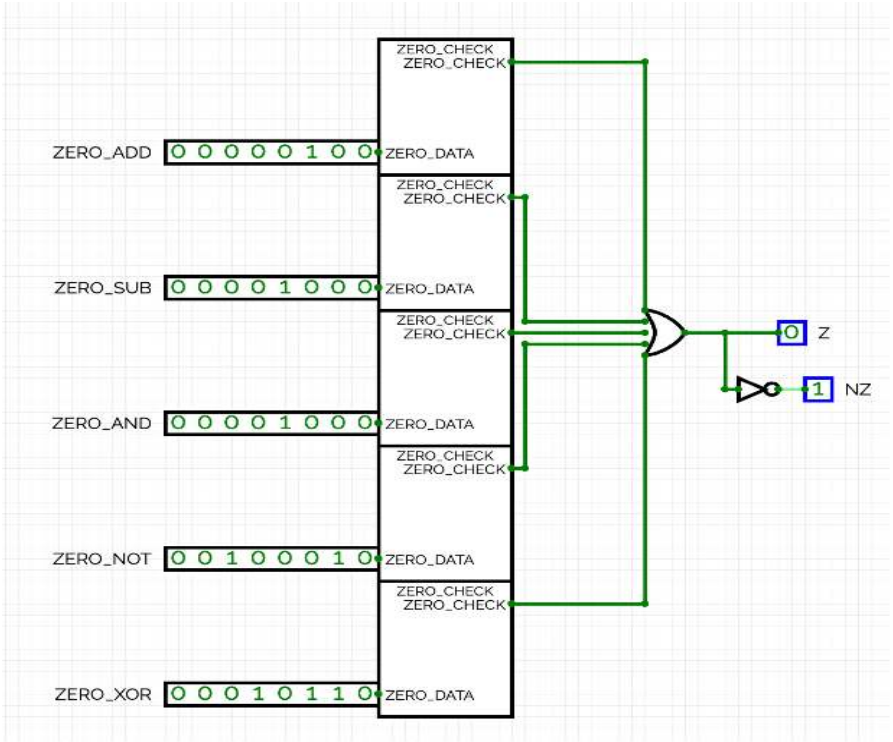
- As there is no clock, output is always made available.

Inputs			Outputs	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- S= ((A XOR B) XOR C_IN)
- C_OUT = ((A XOR B) AND C_IN) OR (A AND B)

ZERO CHECK





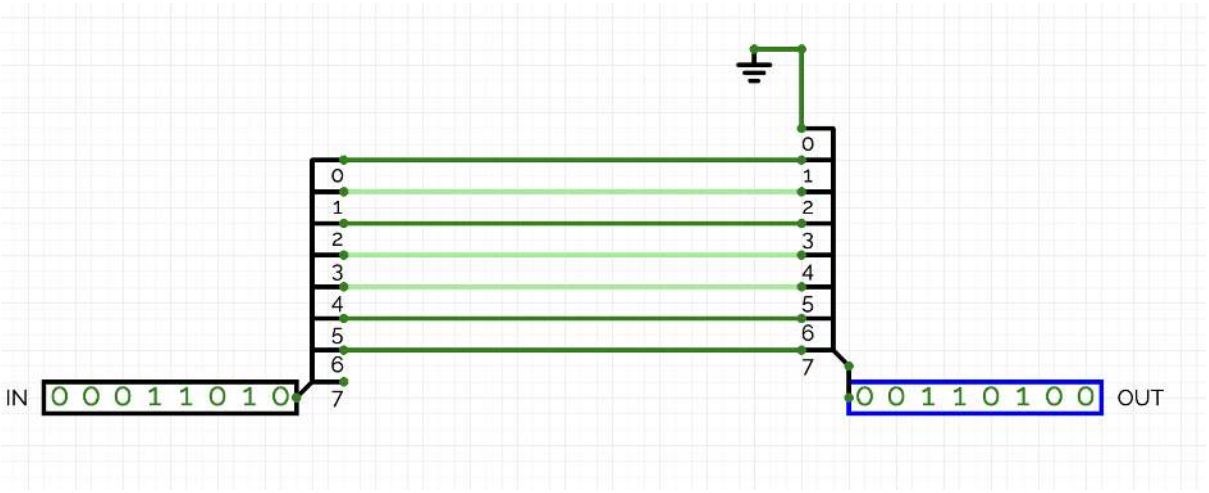
INPUT:

- Five 8 bit input data

OUTPUT:

- Z and NZ (Z is 1 if any one of the 5 input datas are 0, else NZ is 1)

LEFT SHIFT



INPUT:

- 8 Bit input

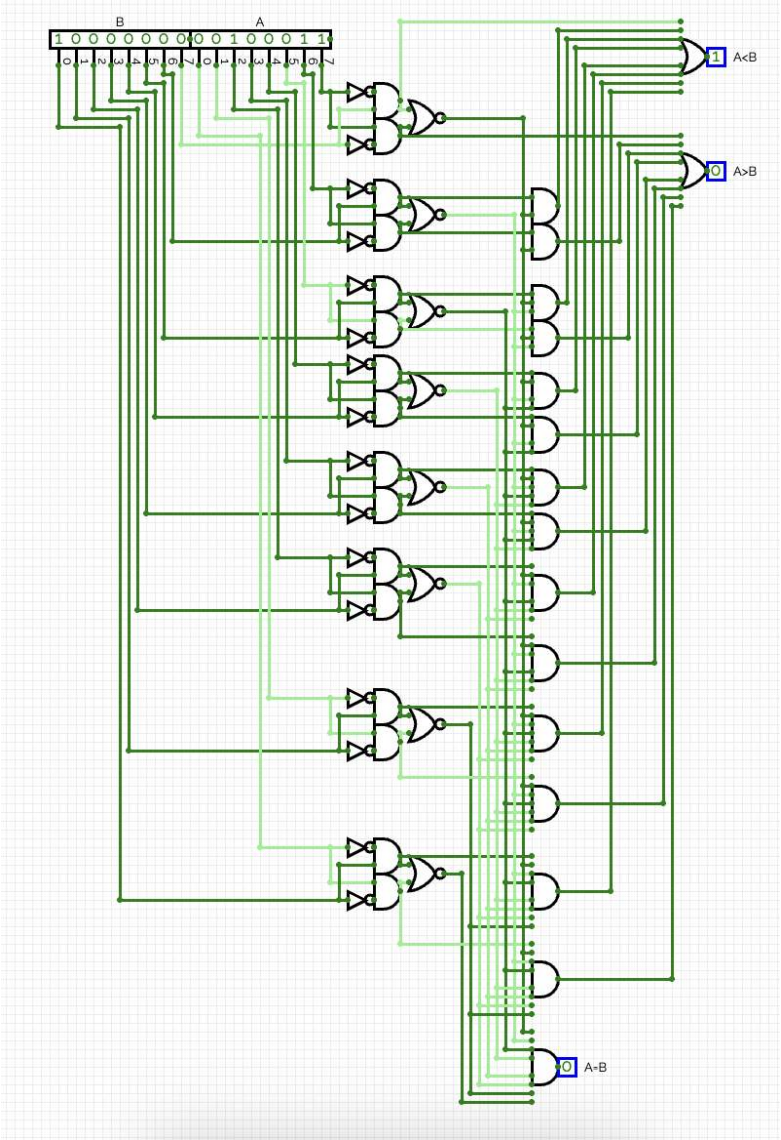
OUTPUT:

- 8 Bit output

NOTE:

- The last 7 bits of the output are the first 7 bits of the output and the first bit of the output is grounded, 0.
- The output is the number we get on multiplying the input by two.

COMPARATOR



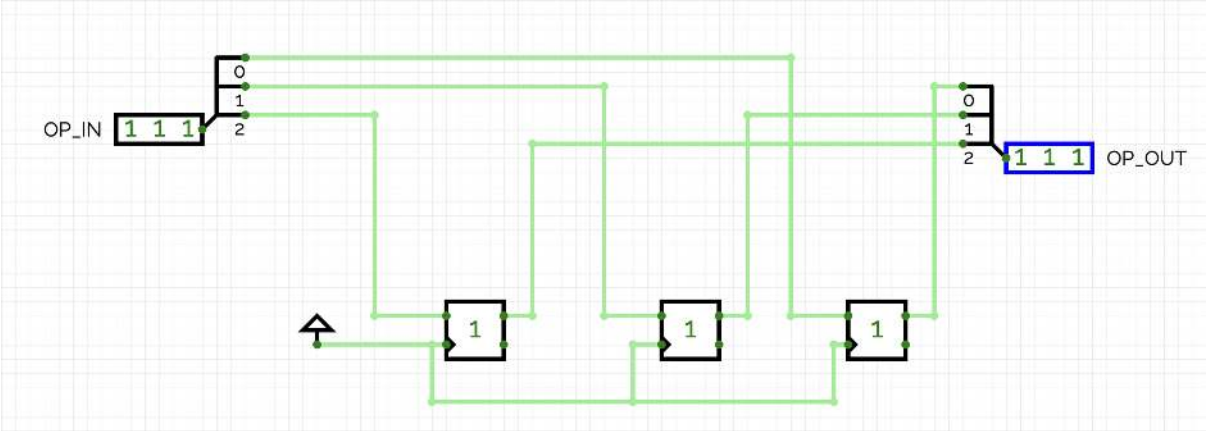
INPUT:

- Two 8 bit input data

OUTPUT:

- 1 if A<B and 0 if A>=B

ALU OPERATION REGISTER



INPUT:

- 3 Bit operation input

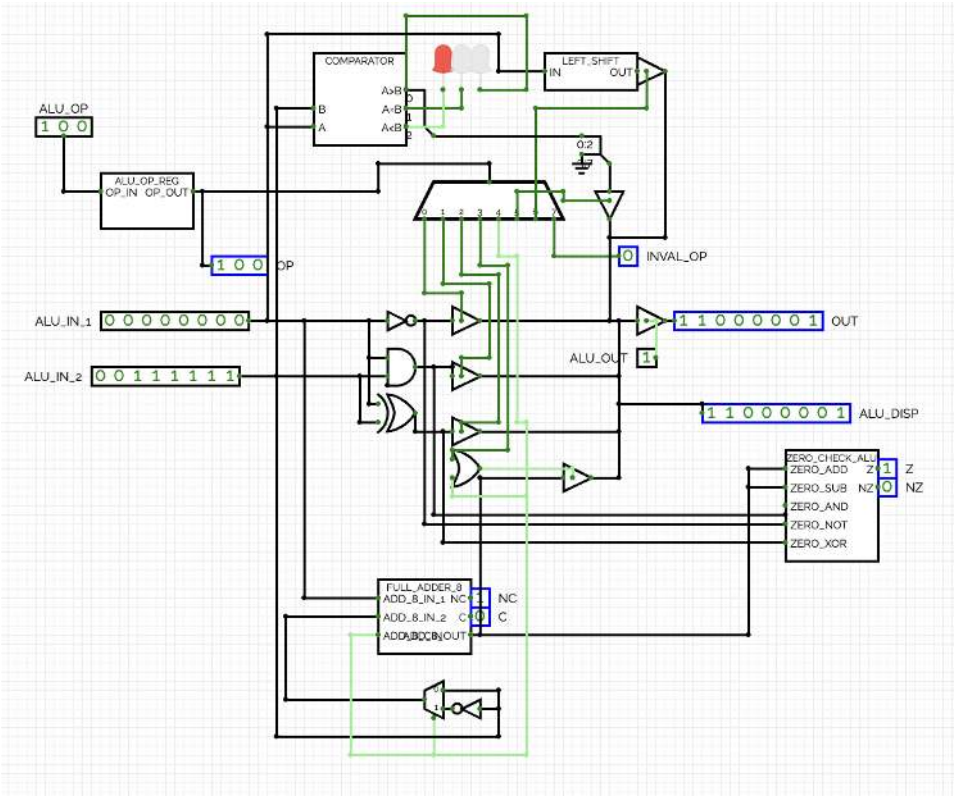
OUTPUT:

- 3 Bit operation output

NOTE:

- The operation input is stored in a register to make it available over 5 t-cycles
- D-latches are used instead of D-flip flops so that the clock is always enabled and level trigger avoids clock issues.

ALU



INPUT:

- ALU_OP - Operation input
- Two 8 bit input data

OUTPUT:

- Digital LED
- NC and C
- Z and NZ
- Operation input
- 8 Bit output after operation

NOTE:

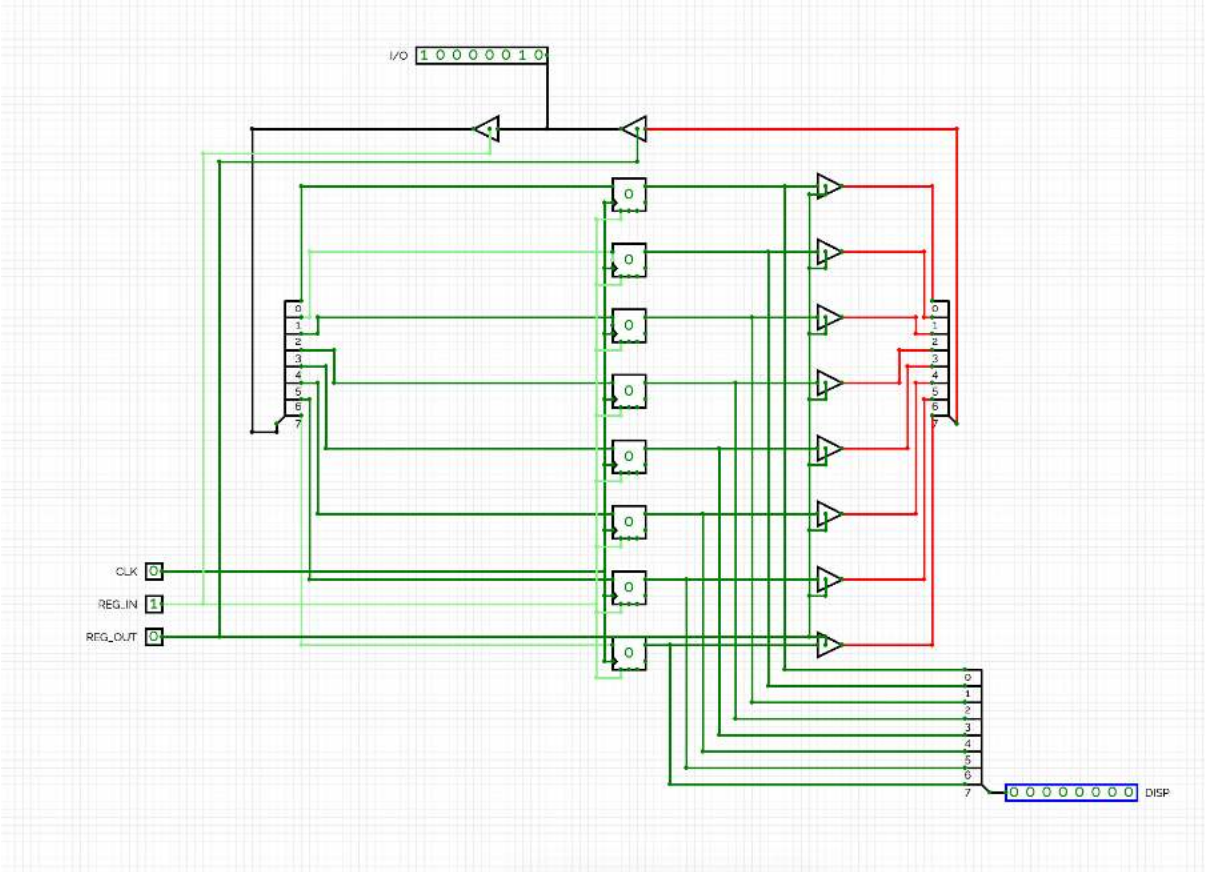
- Based on the operation input the ALU would perform the respective operations.

Operation input	Operation
0	Complement of A
1	A and B
2	A XOR B
3	Addition
4	Subtraction

5	Comparator
6	Left shift
7	Invalid

- The LED glows depending on the comparison between A and B
- The output is displayed after the operation

REGISTER



INPUT:

- 8 Bit input data
- Clock
- REG_IN
- REG_OUT

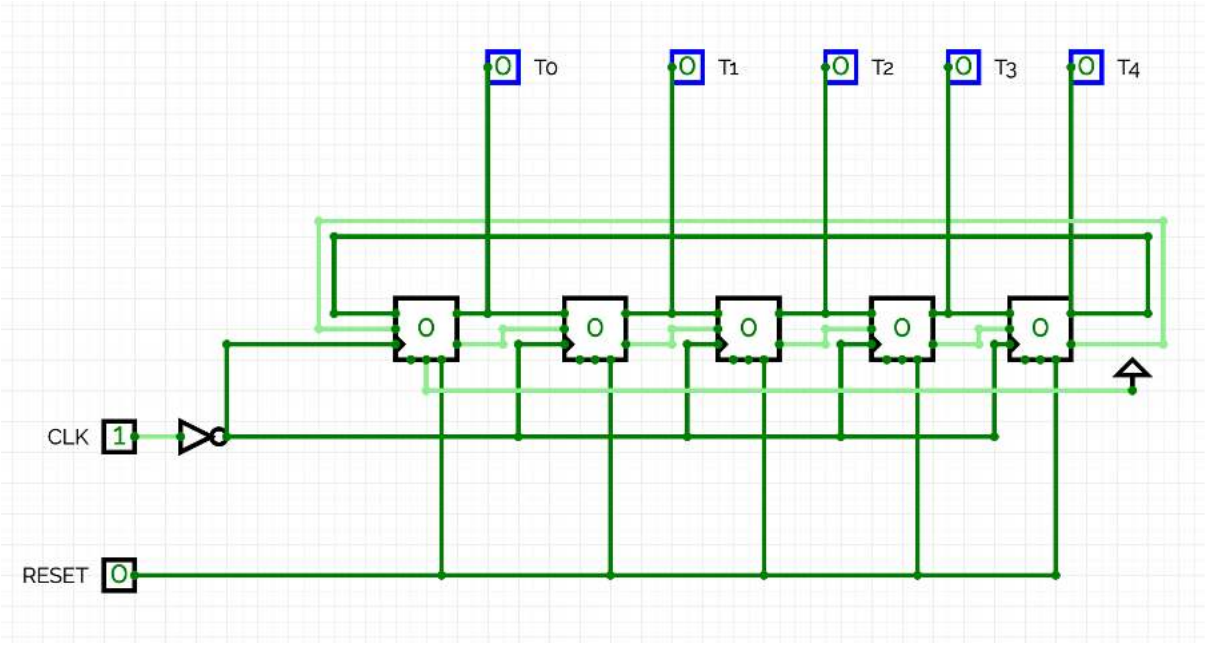
OUTPUT:

- 8 Bit output data display

NOTE:

- When REG_IN is enabled, the input is stored in memory on the positive going edge of the clock and when REG_OUT is enabled, it passes the data again to the bi directional common line on the positive going edge of the clock.
- The common bus serves as a common line for both input and output.
- Display is not triggered by the tri state.

RING COUNTER



INPUT:

- CLOCK
- RESET INPUT

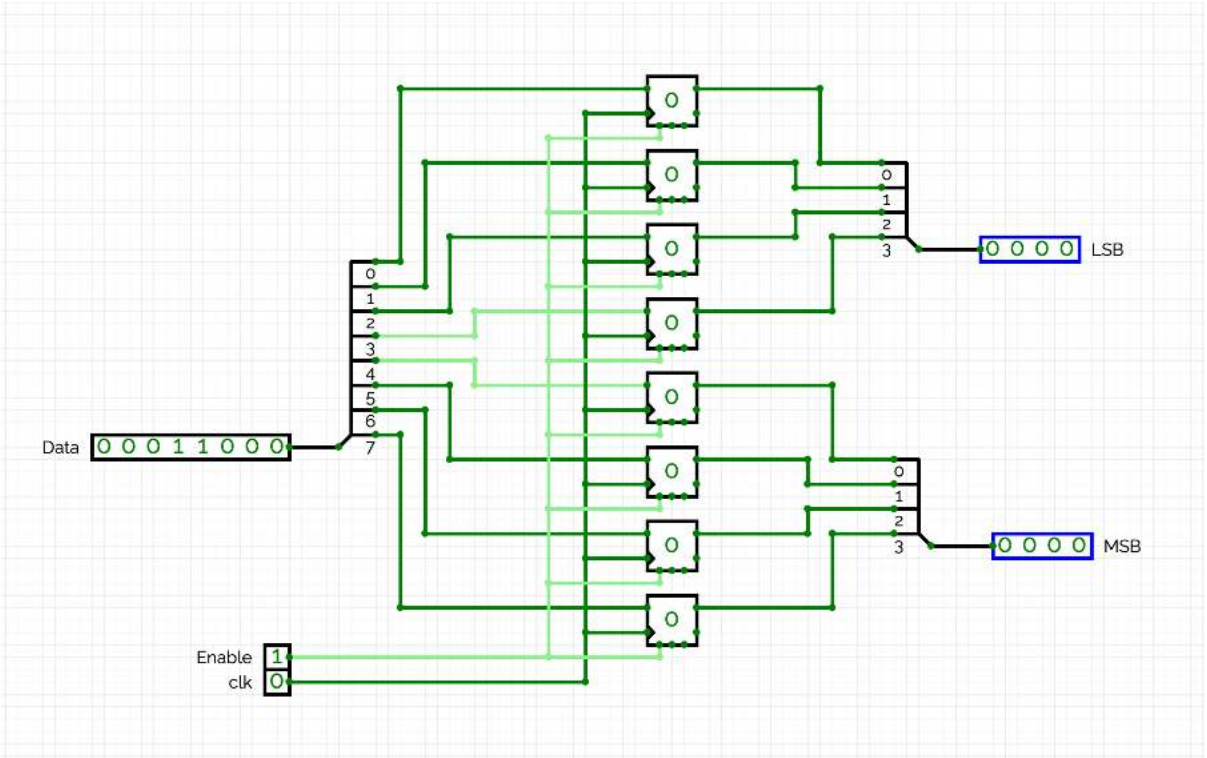
OUTPUT:

- Output of each of the JK flip flops used

NOTE:

- At each positive going edge of the clock the data bits are shifted from one flip flop to the another. The output of the last flip flop is passed to the first flip flop again creating a loop.
- When rest is enabled the counter is reset to 00000.
- Note that the **ring counter is passed with negation of the system clock.** This is for the negative edge trigger which makes sure each T-cycle operation is carried out properly.

BYTE SPLIT



INPUT:

- 8 Bit input data
- Clock
- Enable

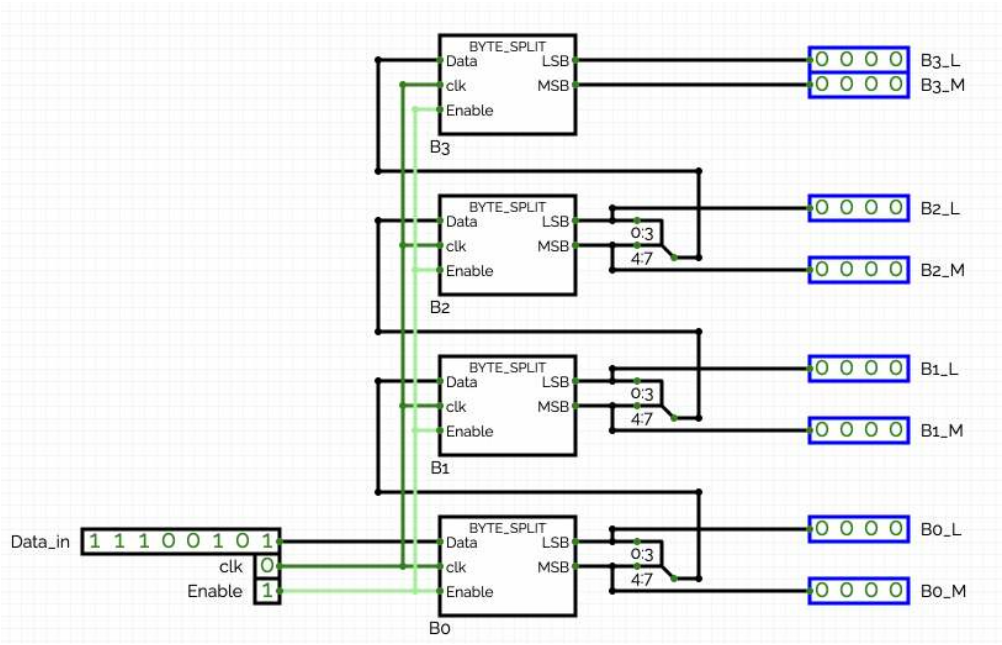
OUTPUT:

- Two 4 bit outputs

NOTE:

- At positive high of a clock the 8 bit data is split into its LSB and MSB

SCROLL DISPLAY



INPUT:

- 8 Bit input data
- Clock
- Enable

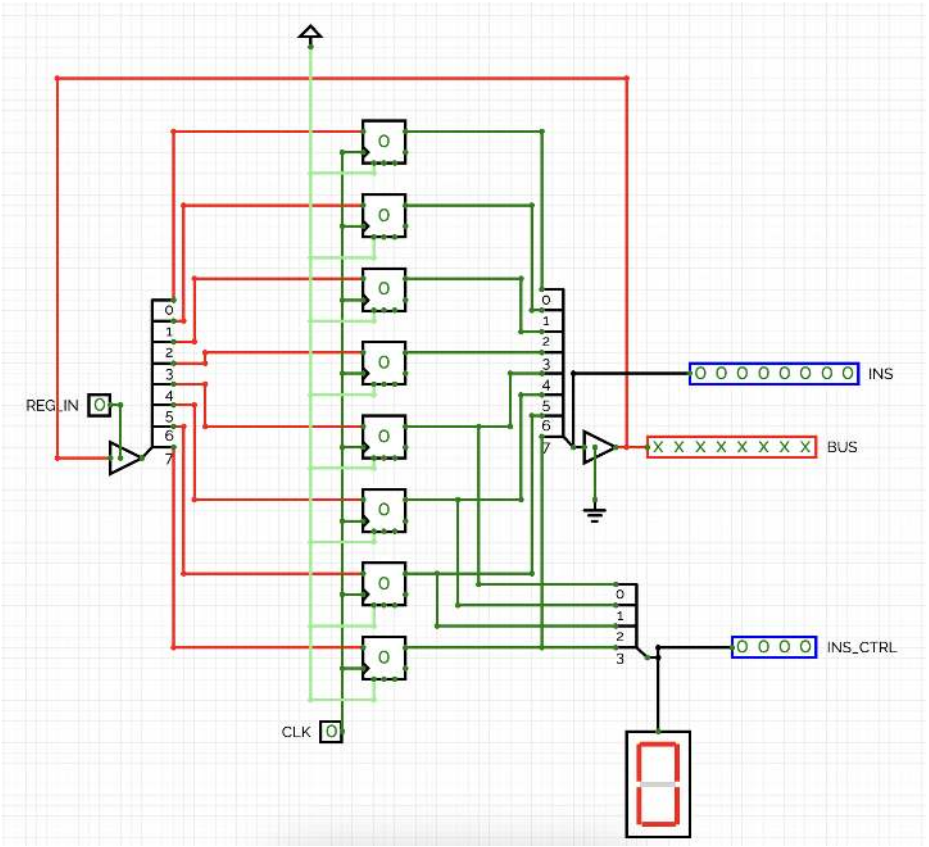
OUTPUT:

- Eight 4 bit output data

WORKING:

- It's a 4-byte display that scrolls up the old values as new values come in. The display unit acts like a FIFO buffer, where the latest byte, A, is displayed at byte 0 location. When a new byte, B, comes in, byte 1 location is updated to A and byte 0 to B. Similarly, when a byte C arrives next, C takes up byte 0 position while B and A holds byte 1 and byte 2 position.

INSTRUCTION REGISTER



INPUT:

- REG_IN
- Clock

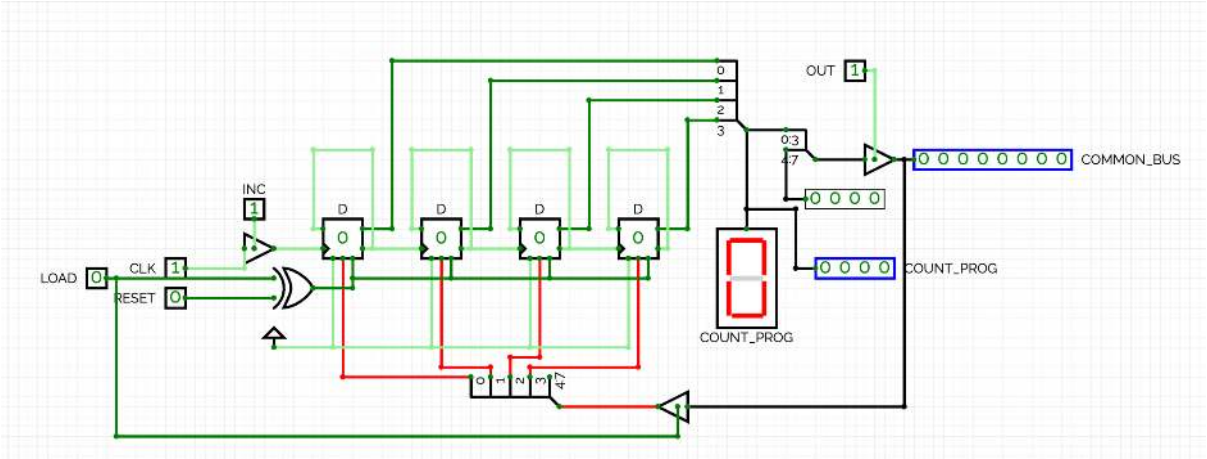
OUTPUT:

- Common bus
- Instruction control
- Instruction

WORKING:

- 8 Bit data from the common bus is stored in the memory on the positive edge of a clock cycle and sends the first 4 bits, MSB, that is the instruction control to the instruction decoder and hardware or software controller.

PROGRAM COUNTER



INPUT:

- RESET
- LOAD
- INC
- OUT
- Clock

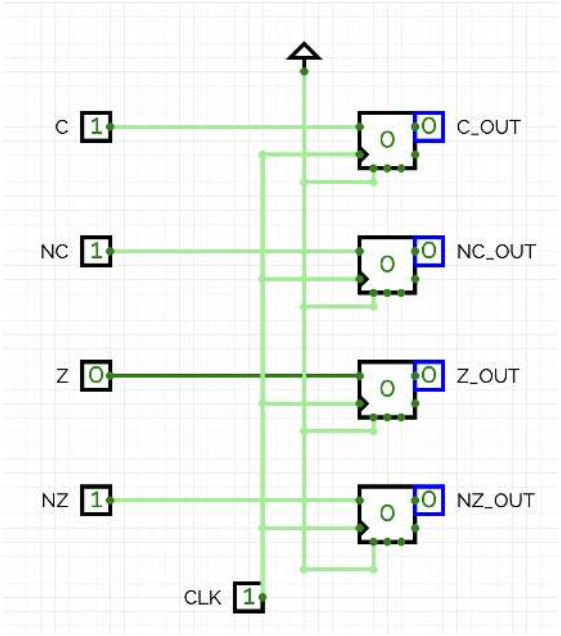
OUTPUT:

- Common Bus
- COUNT_PROG

WORKING:

- When load is set to 1, it loads the data directly from the common bus.
- When increment is set to 1, on the positive edge of a clock cycle the counter is incremented by 1.
- When reset is set to 1, it resets the counter to 0.
- The value in the counter is even displayed by a 7 segment display LED.
- When OUT is set to 1 the data is passed to the common bus.

REGISTER STATUS 4



INPUT:

- Four input data
- Clock

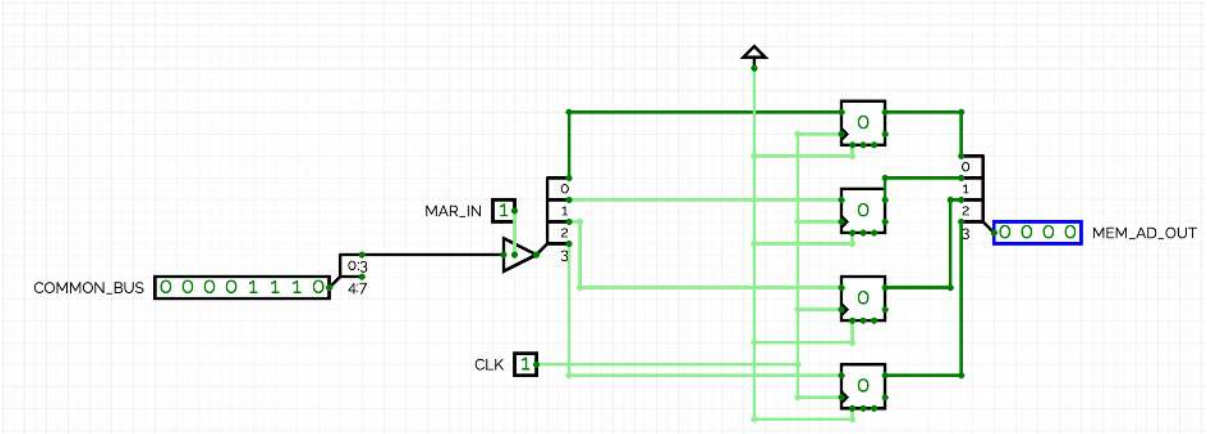
OUTPUT:

- Four 1 bit output data

WORKING:

- When clock is enabled the inputs, C,NC,Z,NZ are stored in the memory.

MEMORY ADDRESS REGISTER



INPUT:

- Clock
- MAR_IN
- Point to some address in the prog and data mem, whenever

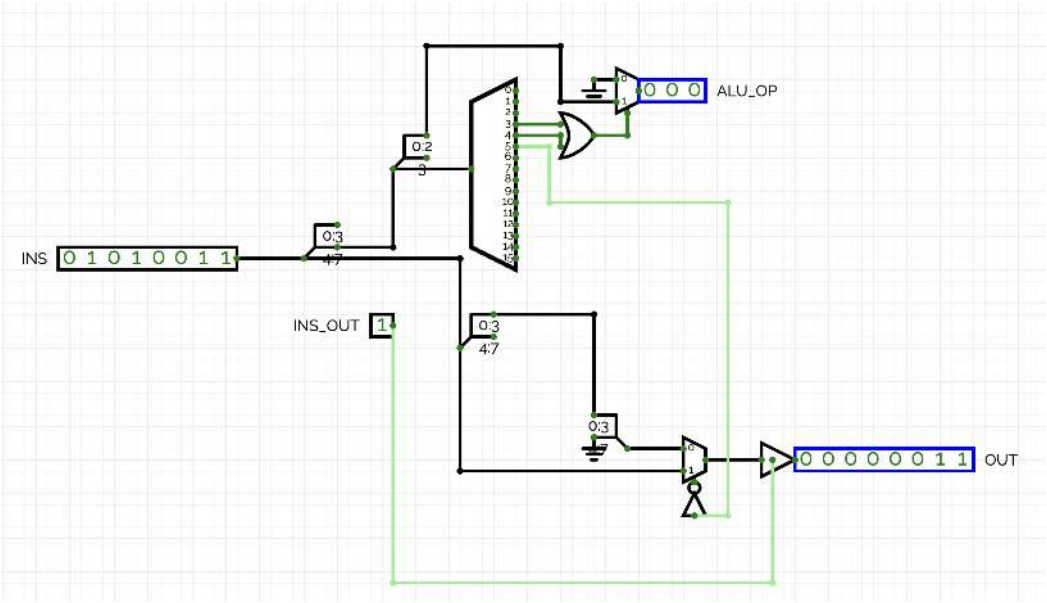
OUTPUT:

- Common bus
- MEM_AD_OUT

WORKING:

- It points to some address in the program and data memory
- There is no tristate buffer to control MEM_AD_OUT.
- Whenever the clock is enabled the address is stored. The address can be used when data has to to be loaded or stored in the ROM.

INSTRUCTION DECODER



INPUT:

- INS_OUT
- 8 bit INSTRUCTION

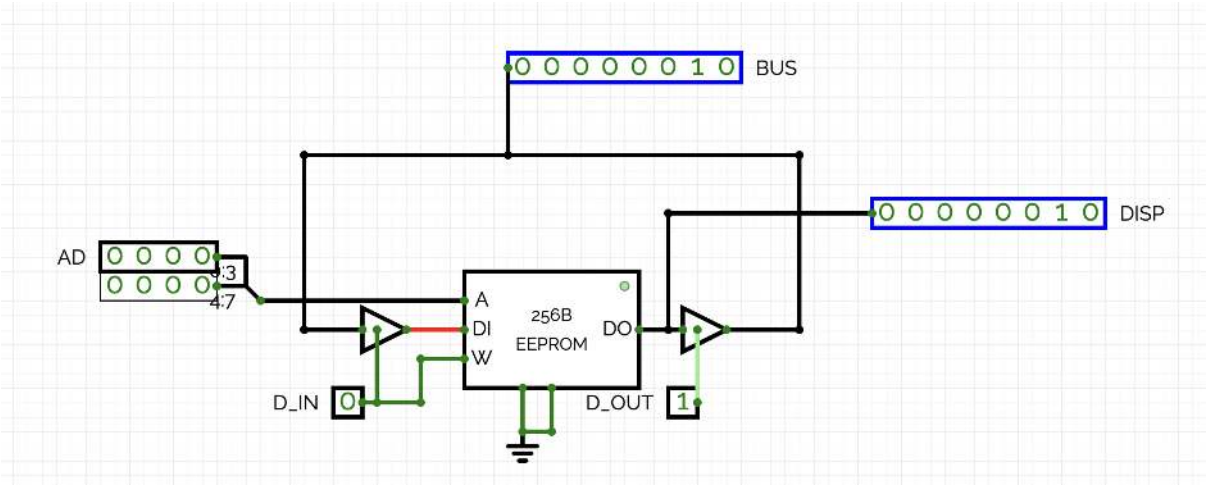
OUTPUT:

- ALU_OP
- OUT

WORKING:

- It stores the instruction received from the program memory.
- The ALU operation passes to the ALU ans stored in the ALU register.
- When INS_OUT is set to 1, the last four bits, that is the LSB is outputted.
- There is no tristate buffer controlling the register.

DATA MEMORY



INPUT:

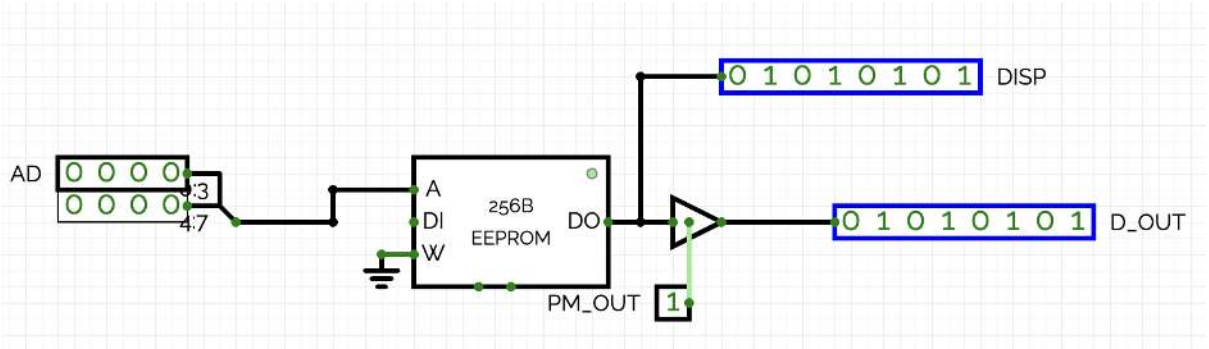
- AD
- D_IN
- D_OUT

OUTPUT:

WORKING:

- It reads and writes but not loads.
- The address is pointed by the MAR.
- When D_IN is enabled it writes data from the memory into the common bus.
- When D_OUT is enabled it reads the data from the common bus
- There is no tristate display.

PROGRAM MEMORY



INPUT:

- AD
- PM_OUT

OUTPUT:

- DISP
- D_OUT

WORKING:

- It reads and loads but can't write.
- Address -> program data(to obtain) comprised of ins
- When PM_OUT is set to 1, it loads the 8 bit instruction plus address or data into the common bus.
- Address is pointed by the MAR
- Displays the output without tristate.