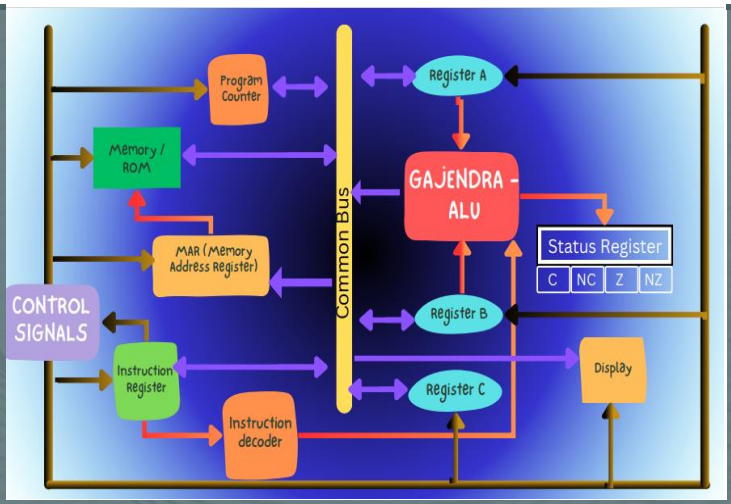
Designing an 8 bit processor

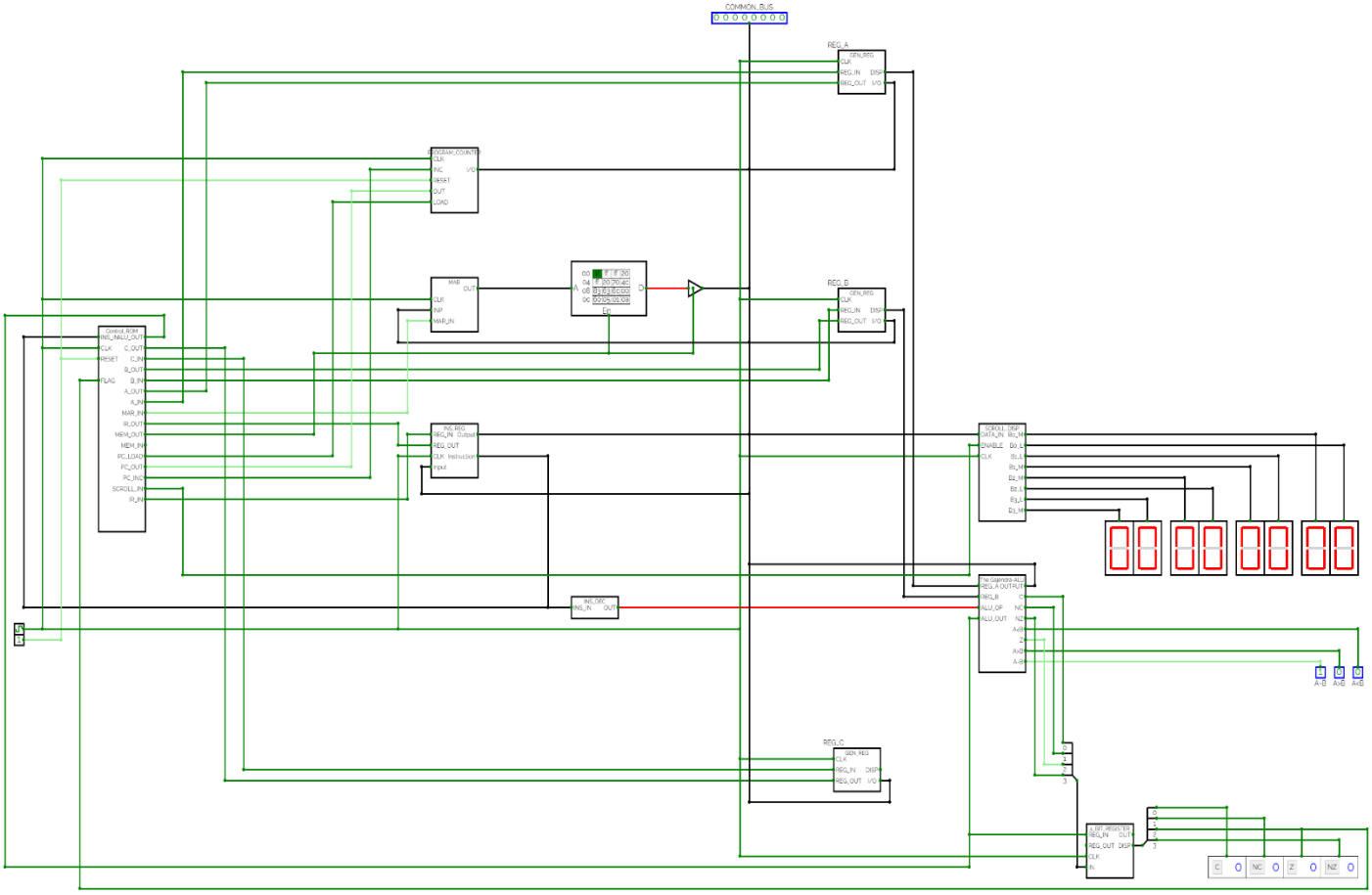
Kritang Kothari
(CS22B012)
Aditya Jain
(CS22B065)
Group 32

CS2310 Assignment Report

Overview

This Processor can be used to perform addition, subtraction, loading values in registers (loading values in memory was not possible as we used ROM for that purpose), output of stored values, multiplication through repeated addition, etc. The bus Outlay of our design is described below and the circuit diagram is on next page.
It is designed using some registers, EEPROM and ROM as memory units, and using some other registers for storing memory addresses, instructions to be executed, status of the last ALU operation, etc.





“cpu_core_arch_gajendra” of our circuit

General Registers

We have used 3 8-Bit Registers in our CPU, with Reg_A being the accumulator, Reg_C being the secondary register (for storing the values that may be of use in later part of program), and a Reg_B, which serves as a temporary register for helping in functions such as Swap, etc.

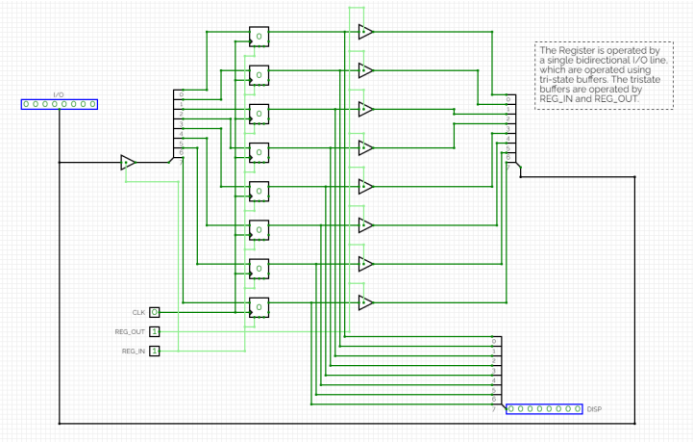


Figure 1 General Registers

The components of registers are described below:

a) Bidirectional Input-Output line
   This line takes in or gives out the output, depending on the REG_OUT and REG_IN bits provided to the register. Together with Tri-state buffers, D-flip-flops, they insure smooth functioning of I/O line.
b) REG_OUT
   This is the bit given to the register which decides whether the register will output the value stored or not. When REG_OUT is 1, register outputs the value stored.
c) REG_IN
   This is the bit given to the register which decides whether the register takes in the input value provided or not. When REG_IN is 1, register takes in the input value provided.
d) Display
   This functions as the display for the current value stored in the register, and always outputs the value stored, irrespective of REG_OUTREG/IN/CLK.

Instruction Registers

- The Instruction register is used to store the instructions that our processor has to perform. It feeds those instruction in Control UNIT, which with the help of an EEPROM carries out all the instructions.
- The instruction register is similar as general 8-bit registers, the only difference is that instruction register is used to store the most significant 4-bits of the 8-bit code provided.

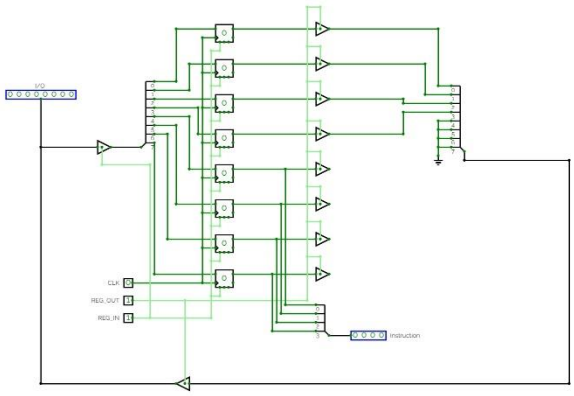
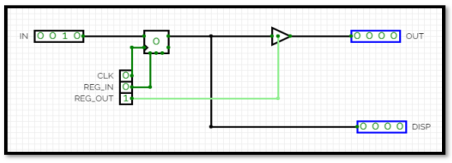


Figure 2 Instruction Register

- The display output is of 4 bits. But as the I/O line is connected to the common bus, to avoid contention error, we use 8 bits for that.

Memory Address Register

- The MAR is used to store the 4 bit address pointing to the memory location that the processor has to access to execute the instruction.



- The MAR is made up of a 4 bit Register which functions the same way as an 8-bit register, the only difference being that the number of bits stored in MAR is 4 instead of 8.

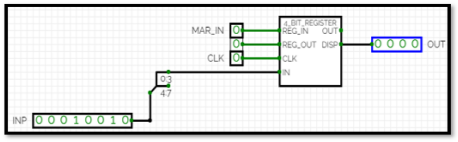


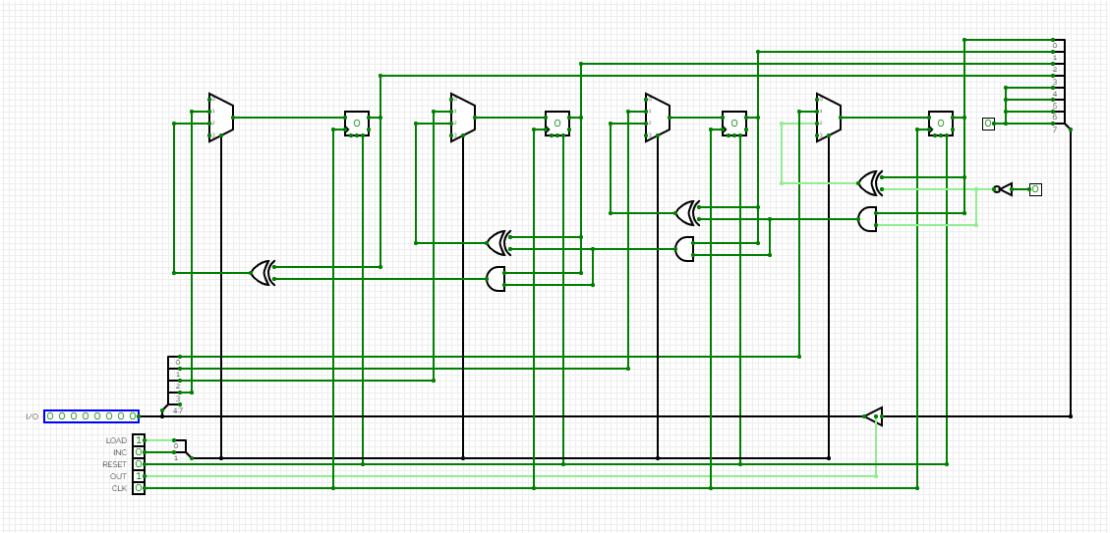
Figure 3 MAR

- The circuit Diagram of 4 Bit register and MAR are given below. Note that the flip flop used in 4-bit register is of 4 bits.

Program Counter

- The program counter is used to keep track the number of instructions that have been executed and helps the MAR to point to the address in the memory unit

- The components of the programming counter are as follows:



a) Load: This bit denotes whether the program counter is storing the value being provided to the Programmer counter by the I/O line.
b) INC : When INC bit is high, the value in program counter increments by 1 after every clock pulse.

Figure 4 Program Counter

c) RESET: When reset bit is high, it resets the value stored in program counter by 0x0.
d) OUT: When this bit is high, it outputs the value stored in program counter through the I/O line.
e) I/O line: This serves as the bidirectional input-output line for the program counter. As it is connected to the common bus, it carries 8 bits, where the 4 MSBs are of no importance in both input and output states.

Arithmetic and Logical Unit

- The ALU is used to perform various operations such as Addition, Subtraction, AND, NOT, XOR, Left Shift, Right shift, comparator, etc.
- The ALU takes in 4 inputs for its functioning:
    a) REG_A and REG_B: ALU takes in the value stored in register A and B through these input ports and uses them for various ALU operations.
    b) ALU_OUT: This input controls the output line of ALU, i.e., the ALU will give output only when this bit is set as high

c) ALU_OP:
   i. 000 => ADD
   ii. 001 => XOR
   iii. 010 => AND
   iv. 011 => NOT
   v. 100 => SUB
   vi. 101 => Rshift
   vii. 110 => Lshift

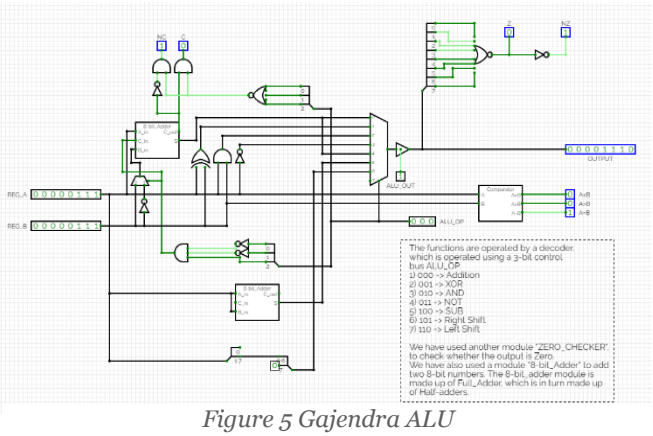
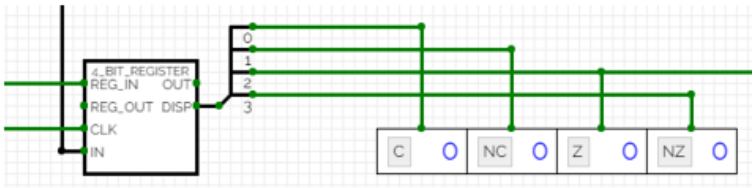


Figure 5 Gajendra ALU

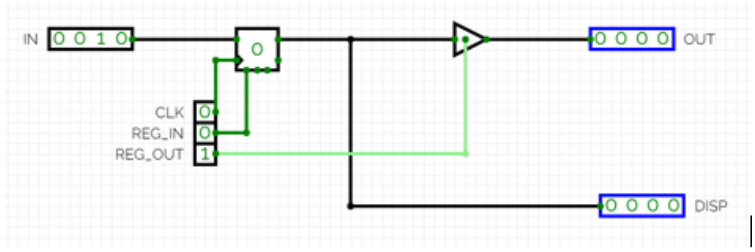
This input is of 3 bits and denotes what instruction is to be performed by the ALU. For our ALU, we have given the ALU codes above.

Status Register

- It is used to keep track of status of the last ALU operation performed.



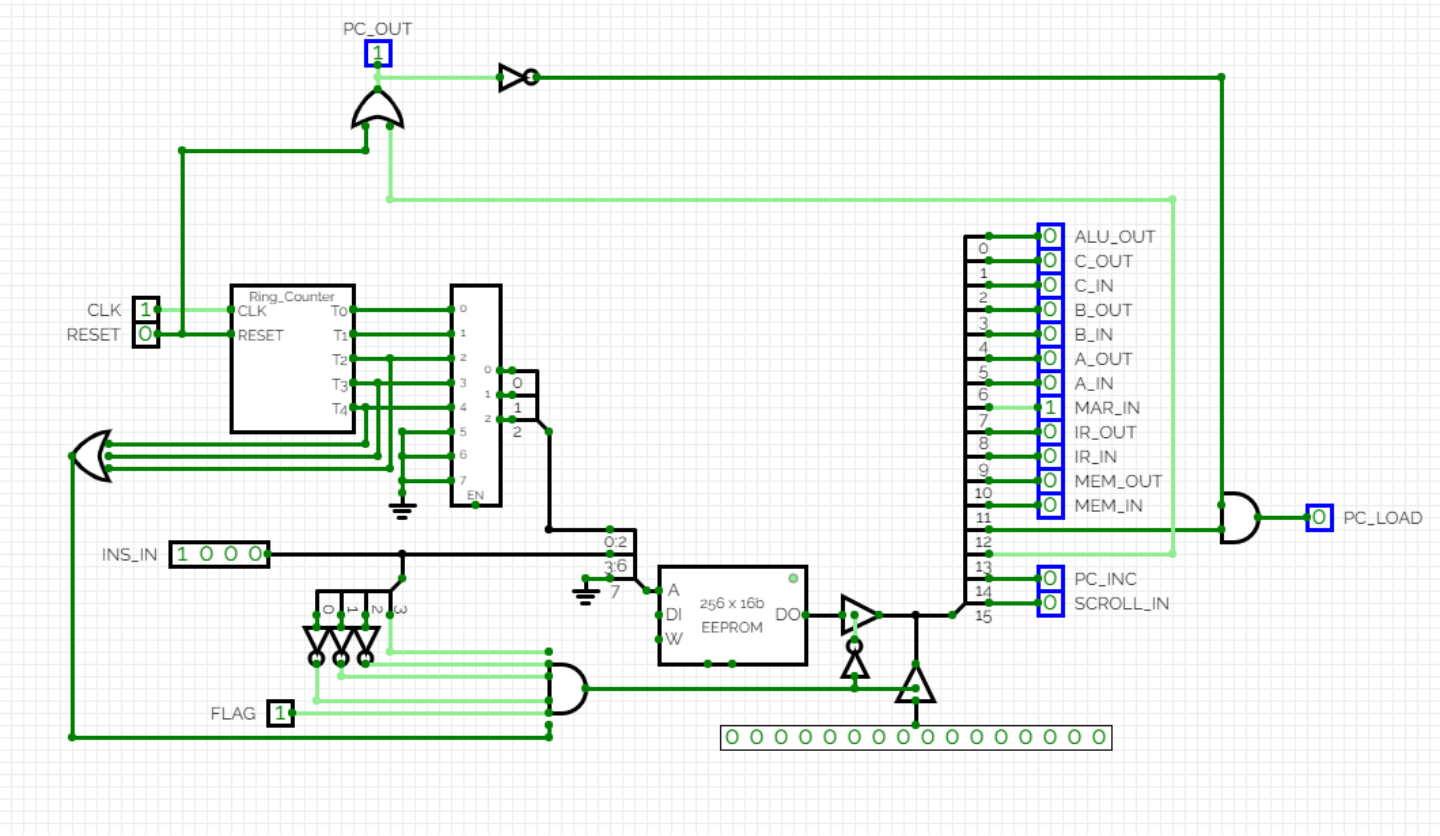
- The status stored are Zero(Z), Not-Zero (NZ), Carry(C) and Not-Carry (NC). These status are stored for instructions like JNZ, etc.



- It is made using a 4 bit-register, and its input and display lines are both of 4 bits.

CPU_TIMING_CONTROLLER

- This component is used to determine what bits are high for the circuit components in certain machine cycles.
- The EEPROM contains all the data of size 16 bits and as per the t-states, instruction, and flag, the cpu_controller gives us the bits for various inputs, outputs.



End of CPU ARCHITECTURE

Instruction Set

NOP (No operation)

- As the name suggests, this instruction performs nothing and all the data remains same except the program counter, which increments to one and points to the next instruction after this set of cycle is completed.
- Operation: Nothing
- Syntax: Operands: Program Counter:

  Nop (any 4-bit value) 0≤X≤15 PC←PC+1
- **No register data is affected.**
  The count in program counter increments by 1.
- 8-bit Opcode

0000	XXXX

LDA (Load A)

- Load 8-bit value in register A which is stored in the memory address pointed by the 4 LSBs of LDA instruction.
- Operation:
  - Reg_A <- Memory Address r
-

  Syntax Operands: Program Counter

  LDA, Memory Address r 0≤r≤15 PC←PC+1
- **The data stored in Reg_A is overwritten by the data from the memory location.**
  The count in program counter increments by 1.
- 8-bit Opcode

0001	rrrr

OUT (Output of register A)

- Outputs the value stored in Register A into the scroll display
- Operation:
    - Scroll_Display <- Reg_A

-

Syntax	Operands:	Program Counter
Out, any 4-bit value	0≤X≤15	PC<--PC+1

- **No stored data is affected, just outputs the value stored in Reg_A. The count in program counter increments by 1.**
- 8-bit Opcode

0010	XXXX

ADD (Add without Carry)

- Adds the 8-bit value stored in memory address to the value stored in register A and store it in register A.
- Operation:
    - Reg_A <- Memory Address r + Reg_A

-

Syntax	Operands:	Program Counter
Add, Memory Address r	0≤r≤15	PC← PC+1

- **Data stored in register A is overwritten by the sum of values of Reg_A and data provided. The data stored in our temporary register Reg_B is also overwritten by the value to be added. The count in program counter increments by 1.**
- 8-bit Opcode

0011	rrrr

SUB (Subtract without Sign)

- Subtracts the value stored in memory address from the value stored in register A. In case of negative results, it stores the complement of result + 1 in register A.

- Operation:
    - Reg_A <- Reg_A – Memory Address r

-

Syntax	Operands:	Program Counter
SUB, Memory Address r	0≤r≤15	PC←PC+1

- **Data stored in register A is overwritten by the difference of values of Reg_A and data provided. The data stored in our temporary register Reg_B is also overwritten by the value to be subtracted.**
  The count in program counter increments by 1.
- 8-bit Opcode

0100	rrrr

LDI (load Immediate)

- Loads the 4 LSBs of the instruction into register A. The 4 MSBs to be loaded are set to 0.
- Operation:
  - Reg_A <- K

-

Syntax	Operands:	Program Counter
LDI, K	0≤K≤15	PC←PC+1

- **The data stored in Reg_A is overwritten by 4 LSBs of instruction.**
  The count in program counter increments by 1.
- 8-bit Opcode

0101	KKKK

JMP (Jumps PC to loaded address)

- Loads the 4-bit value to program counter and thus it jumps to the loaded address.
- Operation:
  - PC <- K

-

Syntax	Operands:	Program Counter
JMP, K	0≤K≤15	PC←K

- **The value of the program counter is updated to the new address provided, rest all remains same.**
- 8-bit Opcode

0110	KKKK

SWP(Swaps value of Reg_C & Reg_A)

- Swaps the value stored in Register C and Register A.
- Operation:
  - Reg_C <- Reg_A Reg_A<- Reg_C
-

Syntax	Operands:	Program Counter
SWP, (any 4-bit value)	X=Any 4 bit value	PC←PC+1

- **The values stored in Register A and C are interchanged with the help of register B. As a result, the data stored in B gets lost.**
  The count in program counter increments by 1.
- 8-bit Opcode

0111	XXXX

JNZ (Jumps PC to loaded address, if flag is 0)

- Loads the 4-bit value to program counter and thus it jumps to the loaded address, if the flag is 0.
- Operation:
  - PC <- K
-

Syntax	Operands:	Program Counter
JNZ, K	0≤K≤15	PC←K (if flag=0)

- **If fag is 0,the value of the program counter is updated to the new address provided, rest all remains same.**
- 8-bit Opcode

1000	KKKK

STA (Stores value of register A)

- Stores the value of register A in memory
- Operation:
  - Memory Address r <- Reg_A

Syntax	Operands:	Program Counter
STA, Memory address Rr	r=Address, 0≤r≤15	PC←PC+1

- **No register data is modified.**
  The data in memory address is updated with register A's data.
- 8-bit Opcode

1001	rrrr

LST (Left Shifts the value in register A)

- Left Shifts the value stored in Reg_A and overwrite A with the modified value.
- Operation:
  - Reg_A <- Reg_A*2

Syntax	Operands:	Program Counter
LST,(any 4-bit value)	0≤X≤15	PC<--PC+1

- **The data stored in Reg_A is modified.**
  It is updated with data obtained by doing the left bit shift operation on data stored in A.
- 8-bit Opcode

1010	XXXX

RST (Right Shifts the value in register A)

- Left Shifts the value stored in Reg_A and overwrite A with the modified value.
- Operation:
  - Reg_A <- Reg_A/2

Syntax	Operands:	Program Counter
RST (any 4-bit value)	0≤X≤15	PC<--PC+1

- **The data stored in Reg_A is modified.**
  It is updated with data obtained by doing the right bit shift operation on data stored in A.
- 8-bit Opcode

1011	XXXX

MAB (Copies data of Reg_A to Reg_B)

- Copies the data in register A into register B
- Operation:
  - Reg_B <- Reg_A

Syntax	Operands:	Program Counter
MAB, (any 4-bit value)	0≤X≤15	PC<--PC+1

- **Data in Reg_B is overwritten with data of Reg_A.**
- 8-bit Opcode

1100	XXXX

MCA (Copies data of Reg_C to Reg_A)

- Copies the data in register C into register A
- Operation:
  - Reg_A <- Reg_C

Syntax	Operands:	Program Counter
MCA, (any 4-bit value)	0≤X≤15	PC<--PC+1

- **Data in Reg_B is overwritten with data of Reg_A.**
- 8-bit Opcode

1101	XXXX

MAC (Copies data of Reg_A to Reg_C)

- Copies the data in register A into register C
- Operation:
  - Reg_C <- Reg_A

Syntax	Operands:	Program Counter
MAC, any 4-bit value	0≤X≤15	PC<--PC+1

- **Data in Reg_B is overwritten with data of Reg_A.**
- 8-bit Opcode

1110	XXXX

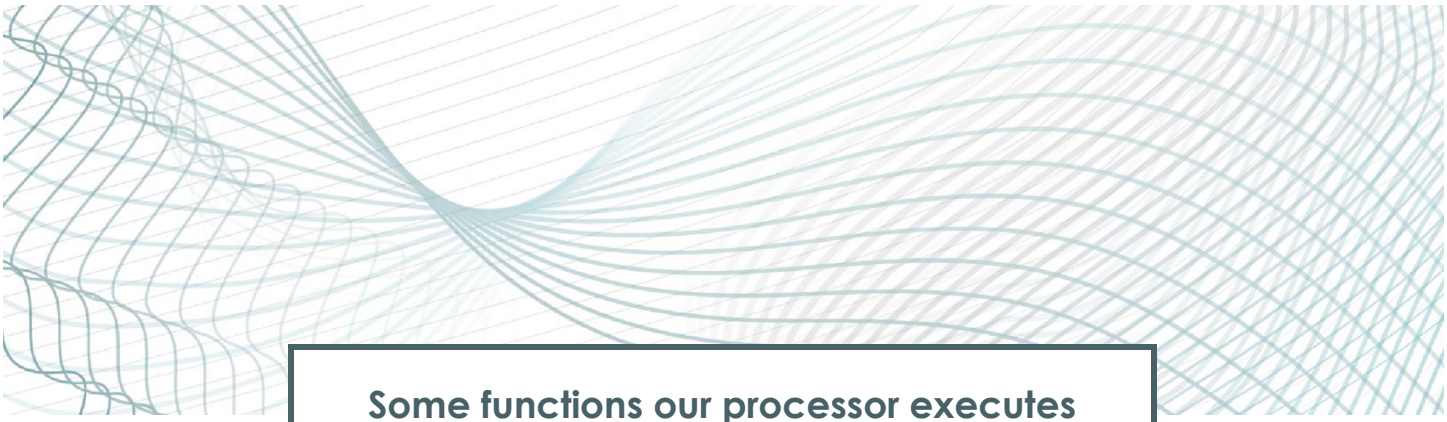
HLT (Halt)

- Stops the incrementation of program counter
-

Syntax	Operands:	Program Counter
Halt (any 4-bit value)	0≤X≤15	PC<--PC

- **Everything remains same, even the address given by program counter. The processor stops executing any more functions.**
- 8-bit Opcode

1111	XXXX



Following is the instruction set of our processor:

Assembly	Machine Code
NOP	0x0
LDA	0x1
OUT	0x2
ADD	0x3
SUB	0x4
LDI	0x5
JMP	0x6
SWP	0x7

Assembly	Machine Code
JNZ	0x8
STA	0x9
LST	0xA
RST	0xB
MAB	0xC
MCA	0xD
MAC	0xE
HLT	0xF

Example 1: Adding two numbers and displaying the output:

Consider the case when we want to add 2 numbers present at addresses 0xF and 0XE.

Address	Assembly Code		Machine Code
0x0	LDA	0xF	0x1F
0x1	ADD	0xE	0x3E
0x2	OUT	(any hexadecimal)	0x2 (any hexadecimal)
0x3	HLT	(any hexadecimal)	0xF (any hexadecimal)

Example 2: Adding and subtracting four numbers in some combination

Suppose you want to do a-b+c-d, where a is stored at 0xF, b at 0xE, c at 0xD, and d at 0xC.and display the output

Address	Assembly Code		Machine Code
0x0	LDA	0xF	0x1F
0x1	SUB	0xE	0x3E
0x2	ADD	0xD	0x4D
0x3	SUB	0xC	0x3C
0x4	OUT	(any hexadecimal)	0x2(any hexadecimal)
0x5	HLT	(any hexadecimal)	0xF (any hexadecimal)

Example 3: Adding numbers from a starting address to an ending address and displaying the result

Suppose we want to add all numbers from 0x9 to 0xF.

Address	Assembly Code		Machine Code
0x0	LDA	0x9	0x19
0x1	ADD	0xA	0x3A
0x2	ADD	0xB	0x3B
0x3	ADD	0xC	0x3C
0x4	ADD	0xD	0x3D
0x5	ADD	0xE	0x3E
0x6	ADD	0xF	0x3F
0x7	OUT	(any hexadecimal)	0x2(any hexadecimal)
0x8	HLT	(any hexadecimal)	0xF (any hexadecimal)

Example 4: Multiplication Through Repeated Addition

Suppose we want to obtain 3 + 7*5, such that 0xF= 0x05, 0xE =0x03, 0xD= 0x07, 0xC=0x01

Address	Assembly Code		Machine Code
0x0	LDA	0xF	0x1F
0x1	SWP	(any hexadecimal)	0x70
0x2	LDA	0xE	0x1E
0x3	ADD	0xD	0x3D
0x4	SWP	(any hexadecimal	0x70
0x5	SUB	0xC	0x4C
0x6	SWP	(any hexadecimal	0x70
0x7	JNZ	0x3	0x83
0x8	OUT	(any hexadecimal)	0x2(any hexadecimal)
0x9	HLT	(any hexadecimal)	0xF (any hexadecimal)

Microinstructions and Controller Logic Design

The control bits of our processor are as follows. If i is assigned to some bit, it is the $(16-i)^{th}$ most significant bit in our control word. For example, PC_INC is the 2^{nd} most significant bit in our control ROM.

```
unsigned int SCROLL_IN  = 15;
unsigned int PC_INC     = 14;
unsigned int PC_OUT     = 13;
unsigned int PC_LOAD    = 12;
unsigned int MEM_IN     = 11;
unsigned int MEM_OUT    = 10;
unsigned int IR_IN      = 9;
unsigned int IR_OUT     = 8;
unsigned int MAR_IN     = 7;
unsigned int REGA_IN    = 6;
unsigned int REGA_OUT   = 5;
unsigned int REGB_IN    = 4;
unsigned int REGB_OUT   = 3;
unsigned int REGC_IN    = 2;
unsigned int REGC_OUT   = 1;
unsigned int ALU_OUT    = 0;
```

NOP

- Cycles needed= 2

```
1<<PC_OUT|1<<MAR_IN,                    T0   0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,          T1   0x4600
0,                                      T2   0x0000
0,                                      T3   0x0000
0,                                      T4   0x0000
```

LDA

- Cycles needed= 4

```
1<<PC_OUT|1<<MAR_IN,                    T0    0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,          T1    0x4600
1<<IR_OUT|1<<MAR_IN,                    T2    0x0180
1<<MEM_OUT|1<<REGA_IN,                  T3    0x0440
0,                                      T4    0x0000
```

STA

- Cycles needed= 4

```
1<<PC_OUT|1<<MAR_IN,                    T0    0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,          T1    0x4600
1<<IR_OUT|1<<MAR_IN,                    T2    0x0180
1<<MEM_IN|1<<REGA_OUT,                  T3    0x0820
0,                                      T4    0x0000
```

ADD

- Cycles needed= 5

```
1<<PC_OUT|1<<MAR_IN,                    T0    0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,          T1    0x4600
1<<IR_OUT|1<<MAR_IN,                    T2    0x0180
1<<MEM_OUT|1<<REGB_IN,                  T3    0x0410
1<<REGA_IN|1<<ALU_OUT,                  T4    0x0041
```

SUB

- Cycles needed= 5

```
1<<PC_OUT|1<<MAR_IN,                    T0    0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,          T1    0x4600
1<<IR_OUT|1<<MAR_IN,                    T2    0x0180
1<<MEM_OUT|1<<REGB_IN,                  T3    0x0410
1<<REGA_IN|1<<ALU_OUT,                  T4    0x0041
```

LDI

- Cycles needed= 3

```
1<<PC_OUT|1<<MAR_IN,                    T0    0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,          T1    0x4600
1<<IR_OUT|1<<REGA_IN,                   T2    0x0140
0,                                      T3    0x0000
0,                                      T4    0x0000
```

JMP

- Cycles needed= 3

```
1<<PC_OUT|1<<MAR_IN,                        T0     0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,              T1     0x4600
1<<PC_LOAD|1<<IR_OUT,                       T2     0x1100
0,                                          T3     0x0000
0,                                          T4     0x0000
```

SWP

- Cycles needed= 5

```
1<<PC_OUT|1<<MAR_IN,                        T0     0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,              T1     0x4600
1<<REGA_OUT|1<<REGB_IN,                     T2     0x0030
1<<REGA_IN|1<<REGC_OUT,                     T3     0x0042
1<<REGB_OUT|1<<REGC_IN,                     T4     0x000C
```

JNZ

- Cycles needed= 3

```
1<<PC_OUT|1<<MAR_IN,                        T0     0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,              T1     0x4600
1<<PC_LOAD|1<<IR_OUT,                       T2     0x1100
0,                                          T3     0x0000
0,                                          T4     0x0000
```

OUT

- Cycles needed= 3

```
1<<PC_OUT|1<<MAR_IN,                        T0     0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,              T1     0x4600
1<<SCROLL_IN|1<<REGA_OUT,                   T2     0x8020
0,                                          T3     0x0000
0,                                          T4     0x0000
```

LST

- Cycles needed= 4

```
1<<PC_OUT|1<<MAR_IN,                        T0     0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,              T1     0x4600
1<<IR_OUT|1<<MAR_IN,                        T2     0x0180
```

```
1<<REGA_IN|1<<ALU_OUT,                  T3      0x0041
0,                                      T4      0x0000
```

RST

- Cycles needed= 4

```
1<<PC_OUT|1<<MAR_IN,                    T0      0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,          T1      0x4600
1<<IR_OUT|1<<MAR_IN,                    T2      0x0180
1<<REGA_IN|1<<ALU_OUT,                  T3      0x0041
0,                                      T4      0x0000
```

MAB

- Cycles needed= 3

```
1<<PC_OUT|1<<MAR_IN,                    T0      0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,          T1      0x4600
1<<REGB_IN|1<<REGA_OUT,                 T2      0x0030
0,                                      T3      0x0000
0,                                      T4      0x0000
```

MCA

- Cycles needed= 3

```
1<<PC_OUT|1<<MAR_IN,                    T0      0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,          T1      0x4600
1<<REGA_IN|1<<REGC_OUT,                 T2      0x0042
0,                                      T3      0x0000
0,                                      T4      0x0000
```

MAC

- Cycles needed= 3

```
1<<PC_OUT|1<<MAR_IN,                    T0      0x2080
1<<PC_INC|1<<MEM_OUT|1<<IR_IN,          T1      0x4600
1<<REGA_OUT|1<<REGC_IN,                 T2      0x0024
0,                                      T3      0x0000
0,                                      T4      0x0000
```

HLT

- Cycles needed= 2

```
1<<MAR_IN,                          T0      0x0080
1<<MEM_OUT|1<<IR_IN,                T1      0x0600
0,                                  T2      0x0000
0,                                  T3      0x0000
0,                                  T4      0x0000
```

Conclusion

We have constructed a processor that can perform functions like addition, multiplication, left shift, right shift, writing in registers, etc.

By accessing the “MAIN” module and filling the cells with the correct instructions and addresses, we can get the output.

Below is the screenshot of multiplication through repeated addition. By taking this as a reference, we can run the desired program, provided that it is possible to execute that instruction through our processor.