“Design of Assembler for 8086 using Python”

Special Assignment Report

Submitted in Partial Fulfillment of the
Requirements for completion of

Course on
2EC601 Computer Architecture

By

Sanjaykumar Parmar [18BEC069]

Department of Electronics and Communication Engineering,
Institute of Technology,
Nirma University,
Ahmedabad - 382481

April 2021

CERTIFICATE

This is to certify that the Comprehensive Evaluation Report entitled **"Design of Assembler for 8086 using Python"** submitted by *Mr. Sanjaykumar Jashvantbhai Parmar* [*18BEC069*] towards the partial fulfillment of for completion of Course on **2EC601 Computer Architecture** is the record of work carried out by him individually.

Date : 15th April, 2021
Faculty Coordinator :

Undertaking for Originality of the Work

I, **Sanjaykumar J. Parmar, 18BEC069** , give undertaking that the Comprehensive Evaluation Report entitled **"Design of Assembler for 8086 using Python"** submitted by me, towards the partial fulfillment of for completion of Course on **2EC601 Computer Architecture**, is the original work carried out by me and I give assurance that no attempt of plagiarism has been made. I understand that in the event of any similarity found subsequently with any other published work or any report elsewhere; it will result in severe disciplinary action.

Signature of the Student

Date : 15th April, 2021

Place : Ahmedabad

Abstract

An assembler is a program that converts assembly language into machine code. It takes the basic commands and operations from assembly code and converts them into binary code that can be recognized by a specific type of processor. Assemblers are similar to compilers in that they produce executable code. Each assembly language is designed for a specific processor, assembling a program is performed using a simple one-to-one mapping from assembly code to machine code.

In this project, We are working on the Assembly language instructions used for 8086 microprocessor. Usually the instructions for 8086 consists of an opcode mnemonic followed by an operand, which might be a list of data, arguments or parameters. The operand are either source or destination resistors.

Here, Firstly we are checking for the error if any. The error is detected by checking whether the opcode is valid or not. Then we will be looking for the number of arguments required for that particular instruction. Once the code/instructions are error free, it will convert the given instruction into the low-level language to the binary machine level language. For, Each and every valid mnemonics there is a unique opcode available and for every register the hex address value transferred as a machine instruction.

After the creation of hex file, The code will get interpreted by the processor and internally it will update the value of registers (GPRs and Flag Registers) on the basis of instructions passed.

INDEX

LIST OF IMAGES

1. Introduction

This project is implemented in python using GUI. Here, The user will be able to write the code in the given space(*textbox area*), user will be able to save the written code to the selected folder, open the already saved file, edit the old file and save it. After saving the code, user will be able to run/compile the code via given options. After, Clicking the compile option the assembler will check for the error if any. When it is error free, it will generated the hex file with machine level code with the name '***filename_hex.h***'. After the compilation is completed you can check the values of all the registers through the option '***RegisterValues***'.

For the primary use we have just implemented it for the basis 18 instructions. The list of instructions are as follows : *'MOV' , 'XCHG' , 'ADD' , 'INC' , 'SUB' , 'DEC' , 'NEG' , 'MUL' , 'NOT' , 'OR' , 'AND' , 'NAND' , 'NOR' , 'SHR' , 'STC' , 'CLC' , 'CMC' , 'CLD'.*

2. Theory

A. Intel 8086 Microprocessor Architecture

- 16 Bit Microprocessor
- 20 Bit Address Bus
- 2^20 Memory Locations [1 MB]
- It can provide 14, 16, bit Registers
- Word size 2 Byte(16 Bits) & Double Word Size 4 Bytes
- Multiplexed Address AD0-AD15 & Data Bus A16-A19
- Requires +5 V Supply
- It uses two stages of pipelining, i.e. Fetch Stage and Execute Stage, which improves performance.
- Fetch stage can pre-fetch up to 6 bytes of instructions and stores them in the queue.
- It has 256 vectored interrupts.
- It consists of 29,000 transistors.

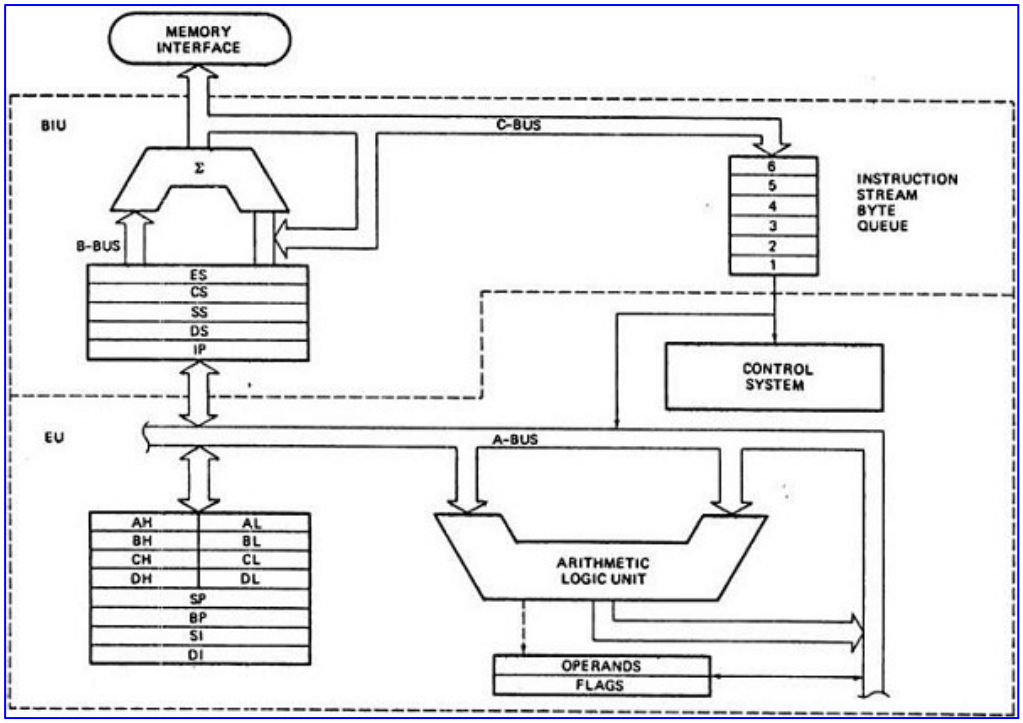

Image 1 : 8086 Microprocessor Architecture

B. General Purpose Registers

AX : Word Multiply, Divide

AL & AH : Byte Multiply, Byte Divide, Decimal Arithmetic

BX : Store Address Information

CX : String Operation, Loops

CL : Variable shift & Rotate

DX : Used to hold I/O address

- If result is more than 16 bits, the lower 16 bits are stored in AX(Accumulator) & Higher order 16 bits are stored in DX.

C. Flag Registers

EU(Execution Unit) contain 16 bit flag Register

9 Active Flags

- 6 Flag Indicates some conditions
- 3 Control Flags

7 Undefined

OF : Overflow

DF : Direction Flag [C]

IF : Interrupt Flag [C]

TF : Trap Flag [C]

SF : Sign Flag

ZF : Zero Flag

AF : Auxilary Flag

PF : Parity Flag

CF : Carry Flag

D. Instruction Sets

I. Data Transfer Instructions

These instructions are used to transfer the data from the source operand to the destination operand. These are also known as copy instructions. They are also called copy instructions.

Here D stands for destination and S stands for source. D and S can either be register, data or memory address.

Opcode	Operand	Description
MOV	D,S	Used to copy the byte or word from the provided source to the provided destination.
PUSH	D	Used to put a word at the top of the stack.
POP	D	Used to get a word from the top of the stack to the provided location.
PUSHA	----	Used to put all the registers into the stack.
POPA	----	Used to get words from the stack to all registers.
XCHG	D,S	Used to exchange the data from two locations.
IN	D,S	Used to read a byte or word from the provided port to the accumulator.
OUT	D,S	Used to send out a byte or word from the accumulator to the provided port.
XLAT	----	Used to translate a byte in AL using a table in the memory.
LAHF	----	Used to load AH with the low byte of the flag register.
SAHF	----	Used to store AH register to low byte of the flag register.
PUSHF	----	Used to copy the flag register at the top of the stack.
POPF	----	Used to copy a word at the top of the stack to the flag register.

Image 2 : Data Transfer Instructions

II. **Arithmetic Instructions**

These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc. In 8086 the destination address is need not to be the accumulator.

ADD	D,S	Used to add the provided byte to byte/word to word.
ADC	D,S	Used to add with carry.
INC	D	Used to increment the provided byte/word by 1.
AAA	----	Used to adjust ASCII after addition.
DAA	----	Used to adjust the decimal after the addition/subtraction operation.
SUB	D,S	Used to subtract the byte from byte/word from word.
SBB	D,S	Used to perform subtraction with borrow.
DEC	D	Used to decrement the provided byte/word by 1.
NEG	D	Used to negate each bit of the provided byte/word and add 1/2's complement.
CMP	D	Used to compare 2 provided byte/word.
AAS	----	Used to adjust ASCII codes after subtraction.
DAS	----	Used to adjust decimal after subtraction.
MUL	8-bit reg	Used to multiply unsigned byte by byte/word by word.
IMUL	8 or 16-bit reg	Used to multiply signed byte by byte/word by word.
AAM	----	Used to adjust ASCII codes after multiplication.
DIV	8-bit reg	Used to divide the unsigned word by byte or unsigned double word by word.
IDIV	8 or 16-bit reg	Used to divide the signed word by byte or signed double word by word.
AAD	----	Used to adjust ASCII codes after division.
CBW	----	Used to fill the upper byte of the word with the copies of sign bit of the lower byte.

Image 3 : Arithmetic Instructions

III. **Bit Manipulation Instruction**

These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc. We can say that these instructions are logical instructions. In 8086, the destination register may or may not the Accumulator.

Opcode	Operand	Description
AND	D,S	Used for adding each bit in a byte/word with the corresponding bit in another byte/word.
OR	D,S	Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.
NOT	D	Used to invert each bit of a byte or word.
XOR	D,S	Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.
TEST	D,S	Used to add operands to update flags, without affecting operands.
SHR	D,C	Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.
SHL/SAL	D,C	Used to shift bits of a byte/word towards left and put zero(S) in LSBs.
ROR	D,C	Used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF].
ROL	D,C	Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF].
RCR	D,C	Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB.
RCL	D,C	Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB.

Image 4 : Bit Manipulation Instructions

IV. String Instructions

String is a group of bytes/words and their memory is always allocated in a sequential order. String is either referred as byte string or word string. Here we will see some instructions which are used to manipulate the string related operations.

Opcode	Operand	Description
REP	Instruction	Used to repeat the given instruction till CX ≠ 0.
REPE/REPZ	Instruction	Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
REPNE/REPNZ	Instruction	Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
MOVS/MOVSB/MOVSW	----	Used to move the byte/word from one string to another.
COMS/COMPSB/COMPSW	----	Used to compare two string bytes/words.
INS/INSB/INSW	----	Used as an input string/byte/word from the I/O port to the provided memory location.
OUTS/OUTSB/OUTSW	----	Used as an output string/byte/word from the provided memory location to the I/O port.
SCAS/SCASB/SCASW	----	Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.
LODS/LODSB/LODSW	----	Used to store the string byte into AL or string word into AX.

Image 5 : String Instructions

V. Program Execution Transfer Instructions

These instructions are used to transfer/branch the instructions during an execution. There are two types of branching instructions. The unconditional branch and conditional branch.

The Unconditional Program execution transfer instructions are as follows.

Opcode	Operand	Description
CALL	address	Used to call a procedure and save their return address to the stack.
RET	----	Used to return from the procedure to the main program.
JMP	address	Used to jump to the provided address to proceed to the next instruction.
LOOP	address	Used to loop a group of instructions until the condition satisfies, i.e., CX = 0

Image 6 : Unconditional Transfer Instructions

Opcode	Operand	Description
JC	address	Used to jump if carry flag CY = 1
JNC	address	Used to jump if no carry flag (CY = 0)
JE/JZ	address	Used to jump if equal/zero flag ZF = 1
JNE/JNZ	address	Used to jump if not equal/zero flag ZF = 0
JO	address	Used to jump if overflow flag OF = 1
JNO	address	Used to jump if no overflow flag OF = 0
JP/ JPE	address	Used to jump if parity/parity even PF = 1
JNP/ JPO	address	Used to jump if not parity/parity odd PF = 0
JS	address	Used to jump if sign flag SF = 1
JNS	address	Used to jump if not sign SF = 0
JA/ JNBE	address	Used to jump if above/not below/equal instruction satisfies.
JAE/ JNB	address	Used to jump if above/not below instruction satisfies.
JBE/ JNA	address	Used to jump if below/equal/ not above instruction satisfies.
JG/ JNLE	address	Used to jump if greater/not less than/equal instruction satisfies.
JGE/ JNL	address	Used to jump if greater than/equal/not less than instruction satisfies.
JL/ JNGE	address	Used to jump if less than/not greater than/equal instruction satisfies.
JLE/ JNG	address	Used to jump if less than/equal/if not greater than instruction satisfies.
JCXZ	address	Used to jump to the provided address if CX = 0

Image 7 : Conditional Transfer Instructions

VI. **Process Control Instructions**

These instructions are used to control the processor action by setting/resetting the flag values.

Opcode	Operand	Description
STC	----	Used to set carry flag CY to 1
CLC	----	Used to clear/reset carry flag CY to 0
CMC	----	Used to put complement at the state of carry flag CY.
STD	----	Used to set the direction flag DF to 1
CLD	----	Used to clear/reset the direction flag DF to 0
STI	----	Used to set the interrupt enable flag to 1, i.e., enable INTR input.
CLI	----	Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

Image 8 : Process Control Instructions

3. Flow Chart

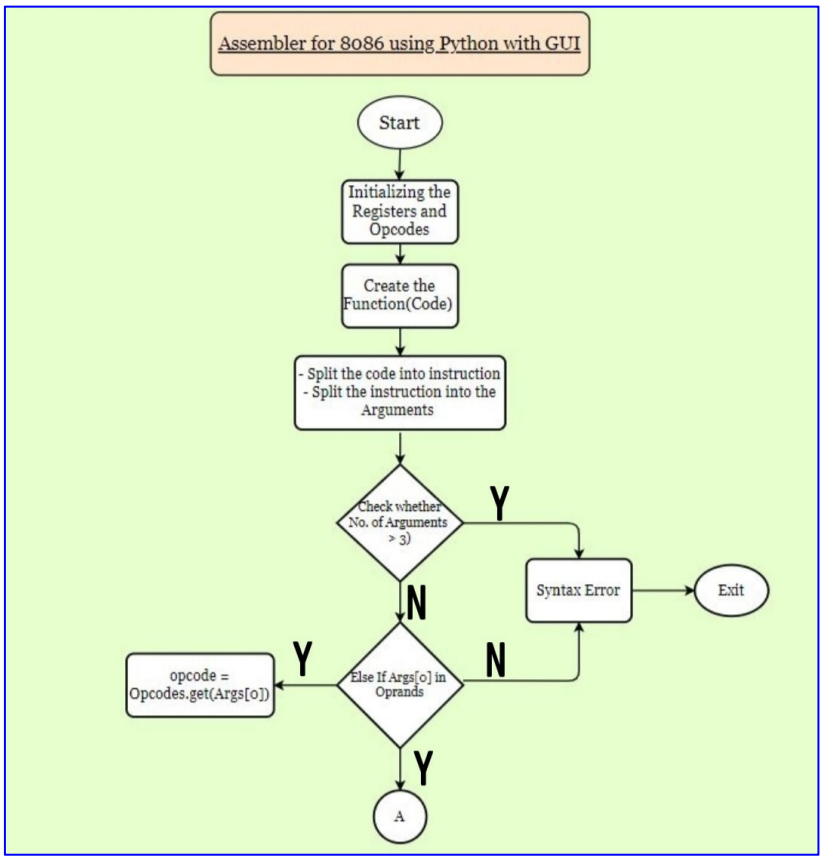

Image 9 : Flow Chart 1

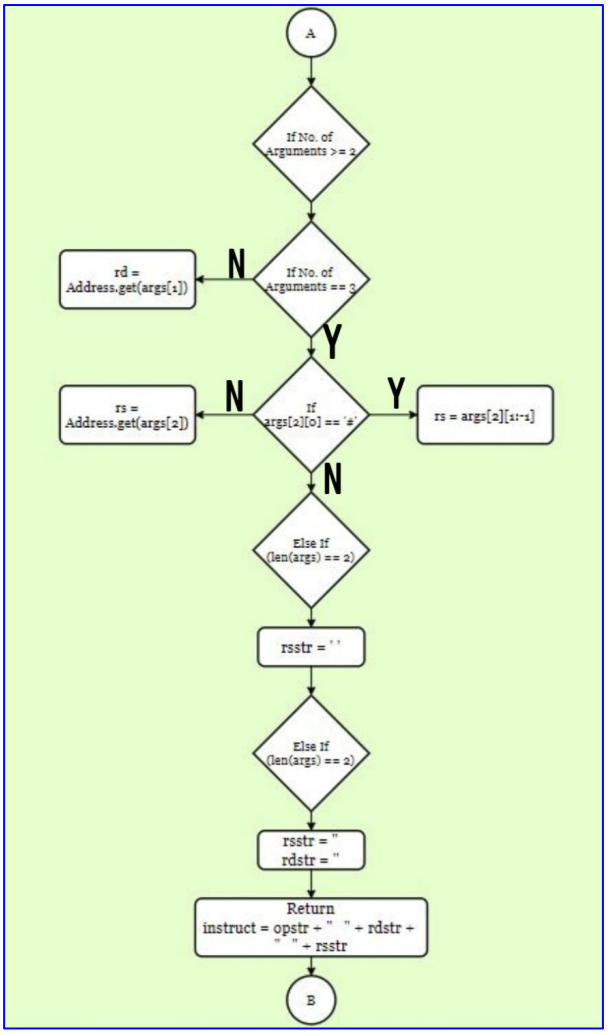

Image 10 : Flow Chart 2

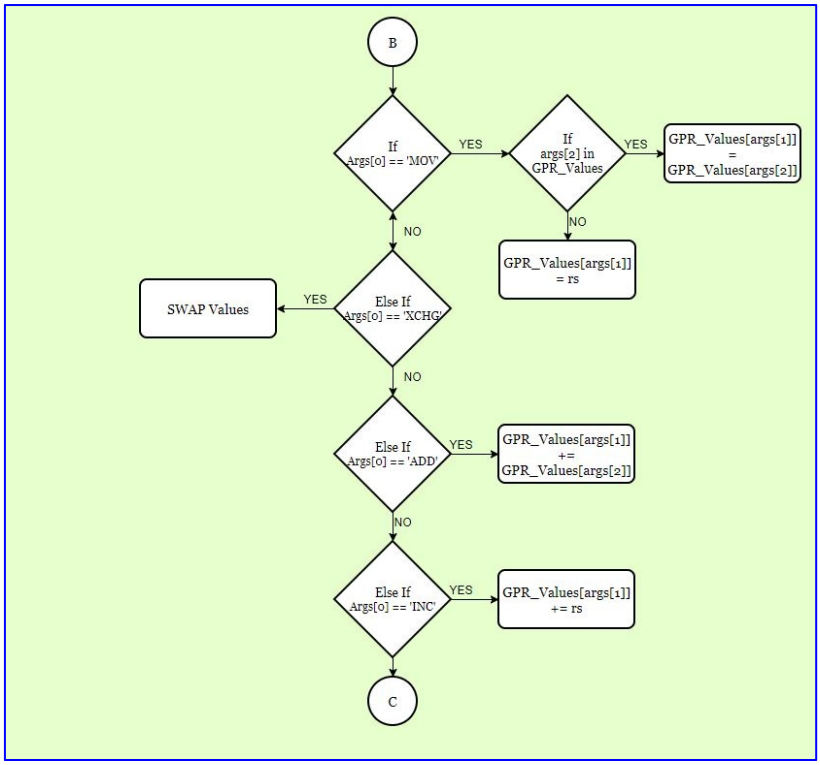

Image 11 : Flow Chart 3

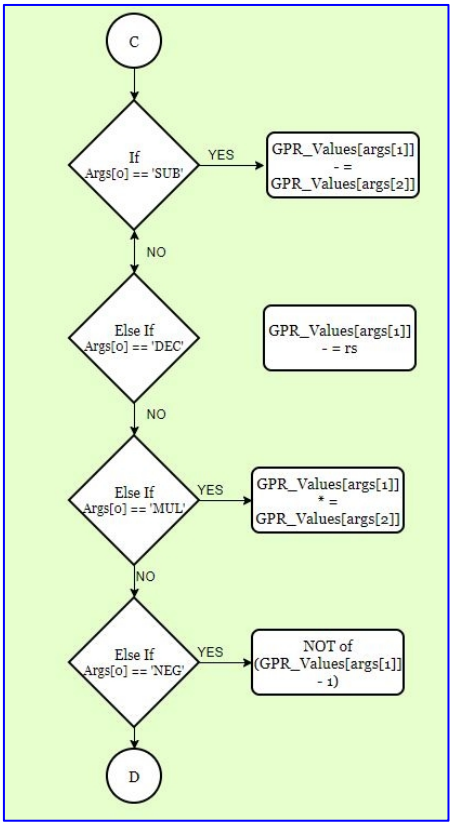

Image 12 : Flow Chart 4

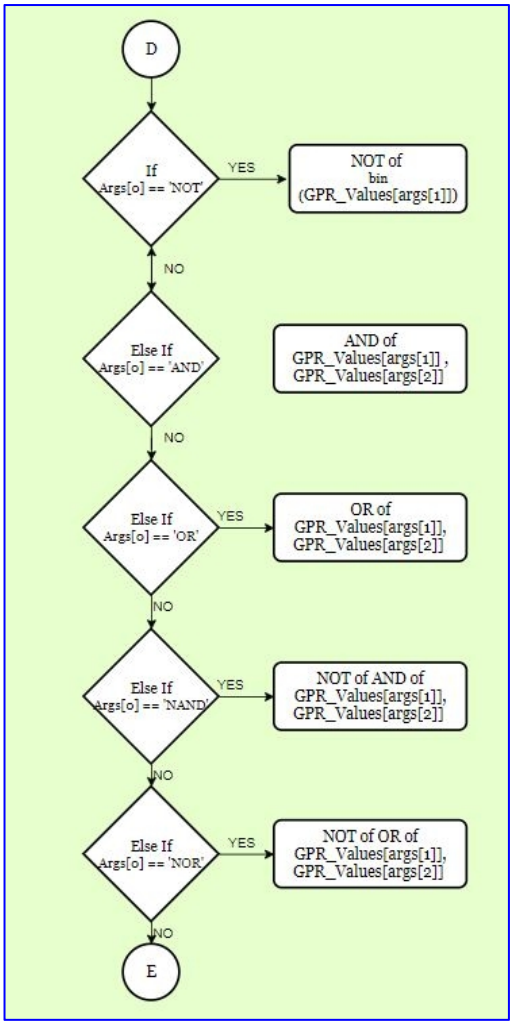

Image 13 : Flow Chart 5

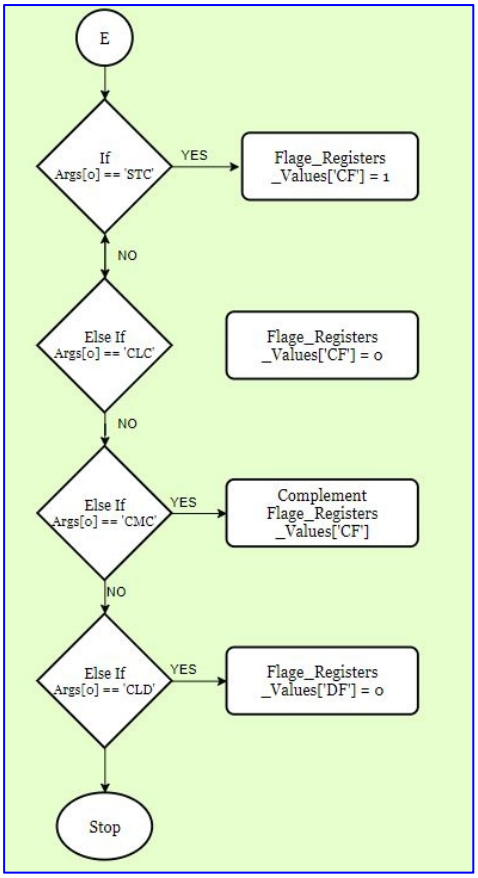

Image 14 : Flow Chart 6

4. Result

8086 Assembler [C:/Users/maulik parmar/Desktop/SANJU/Projects/8086 Assembler/Testing]

File Run

```
MOV AL #80H
MOV BL AL
ADD AL BL
SUB AL CL
MUL AL BL
AND BL #40H
OR CL #40H
```

Image 15 : Sample Code

8086 Assembler [C:/Users/maulik parmar/Desktop/SANJU/Projects/8086 Assembler/Testing_hex.txt]

File Run

```
0 | 0x101a      0x201a      #50 | MOV AL #80H
1 | 0x101a      0x201e      0x201a | MOV BL AL
2 | 0x1022      0x201a      0x201e | ADD AL BL
3 | 0x102a      0x201a      0x2022 | SUB AL CL
4 | 0x1032      0x201a      0x201e | MUL AL BL
5 | 0x1c16      0x201e      #28 | AND BL #40H
6 | 0x1c12      0x2022      #28 | OR CL #40H
```

Image 16 : Generated Hex File

GENERAL PURPOSE REGISTERS

AL	0x3e8
AH	0x44c
BL	0x4b0
BH	0x514
CL	0x578
CH	0x5dc
DL	0x640
DH	0x6a4

POINTERS

SP	8040
BP	8044

FLAG REGISTERS

CF	PF	AF	ZF	SF	TF	IF	DF	OF
1	1	0	1	1	0	0	0	0

Image 17 : Register Values

5. Conclusion

In this project we have successfully implemented the "**Assembler for 8086 using Python**" on Jupiter Notebook. The implemented assembler is comparatively faster than that of regular ones. The assembler based on python is easier, because python is open sourced, easily accessible, good runtime environment & also flexible with the available inbuilt functions.

References

[1] Microprocessor - 8086 Overview, *By Tutorials Point Website*

[2] Compiling Python Syntax, *by BEN HOYT Website*

[3] ASM8086, *By Matti J. Kärki,* GITHUB Website