Assembler/simulator for instructions of 8086 microprocessor using C++

Dishank Garg
Dept. of Electronics and Communication Engineering
Institute of Technology Nirma University
20bec030@nirmauni.ac.in

Dixit Dudhat
Dept. of Electronics and Communication Engineering
Institute of Technology Nirma University
20bec033@nirmauni.ac.in

Abstract -The 8086 microprocessor architecture is a popular and significant microprocessor design that has been extensively applied in a variety of fields. The 8086 microprocessor software development and testing process requires an accurate and trustworthy simulator. The concept and implementation of an 8086-microprocessor simulator in C++ are presented in this work. The simulator replicates the 8086 microprocessor's features, including its instruction set, registers, and memory layout. It offers users a simple interface via which they may enter programmes written in machine code, simulate their execution, and track the outcomes. Users of the simulator may fully comprehend and debug their code thanks to features like step-by-step execution, memory and register inspection, and debugging tools.

I. INTRODUCTION

Testing and debugging assembly language programs on physical hardware can be challenging and time-consuming. Microprocessor simulators are frequently used to test, debug, and analyze the behavior of microprocessors in a virtual setting, without the requirement for the actual hardware, to address this problem. The simulator is developed in C++, utilizing the principles of object-oriented programming for effective and reusable code. With distinct classes for the CPU, memory, and user interface components, the architecture adopts a modular approach. By simulating a microprocessor's behavior, a simulator enables programmers to run assembly language code, monitor the microprocessor's status, and evaluate the outcomes.

There are 8 types of instructions supported by 8086 microprocessor, some of which are data transfer, arithmetic, bit manipulation, branch, and loop instructions. Here, we have implemented some of these instructions and generated their equivalent results.

II. ASSEMBLY LANGUAGE

An assembly language gives instructions to the processors for performing various tasks. It is unique for any processor. Assembly language is almost similar to Machine language but has easy language and code. Since machine language comprises 0s and 1s, it is difficult to write a program using it. Assembly language code can be written by using a compiler. It makes use of opcode for the instructions. Opcode primarily provides information about the specific instruction. Opcode is represented in terms of symbols. This symbolic representation of opcode is known as Mnemonics which is used by the programmer to remember the operation.[1]

A. Function of Assembler

Assembler converts assembly language programs into their corresponding object code. Code written in assembly language is given as an input to the assembler and it gives object code as an output. Since the language used is mnemonic language, assembler design depends on machine architecture. [2]

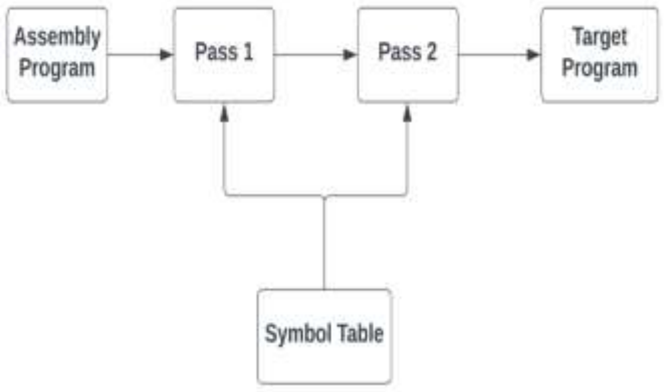


Fig 1. Assembler [2]

The function of a basic assembler is:

1. Translation of mnemonic language code to its corresponding object code.
2. Assignment of machine addresses to corresponding symbolic labels.[3]

The processes performed inside the assembler are:

1. Scanning (also known as tokenizing).
2. Parsing is the process of validating the instructions.
3. Creating the symbol table.
4. Resolving the forward references.
5. Converting into the machine language. [4]

In other words, the Design of Assembler is:

1. Converting mnemonic opcodes to their equivalent machine language.
2. Converting symbolic operands to their corresponding machine address.
3. Converting data constants to the corresponding internal machine representation.
4. Writing object programs and assembly listing.[4]

B. Type of Assemblers

"Assembler is classified based on several stages it uses to convert the assembly-level language to machine-level language:

1. One-Pass Assembler: A one-pass assembler is a type of assembler that processes the assembly language code in a single pass, from beginning to end. It translates the assembly language instructions into machine code as it reads through the source code. One-pass assemblers are typically used for simple, small-scale projects where the source code can be processed in a single pass without the need for multiple passes or complex symbol resolution.[4]

Pass 1

1. Defines Symbol table and Opcode table.
2. Keep track of the location counter.
3. Processing of pseudo instructions.
4. Allocate the address to each statement.
5. Save the address allocated to all labels which are to be used in Pass-2.[4]

2. Two-Pass Assembler: A two-pass assembler is a type of assembler that processes the assembly language code in two passes. In the first pass, it scans the source code to collect information about symbols, labels, and addresses and builds a symbol table. In the second pass, it uses the information from the symbol table to generate the machine code. Two-pass assemblers are typically used for larger, more complex projects where multiple passes are needed to resolve symbols and addresses.

Pass 2

1. Conversion of the symbolic opcode into its corresponding numeric opcode.
2. Generation of machine code according to the values of symbols and literals.
3. Processes the assembler directives not done during Pass-1.
4. Writing object programs and assembly listing".[4]

3. Multi-Pass assembler: An assembler that can run the source code through more than two passes is known as a multi-pass assembler. For complicated projects that call for sophisticated capabilities like macro expansion, conditional assembly, and comprehensive symbol resolution, multi-pass assemblers are utilized. They can build the machine code and manage these complex features through numerous rounds.

III.8086 ARCHITECTURE

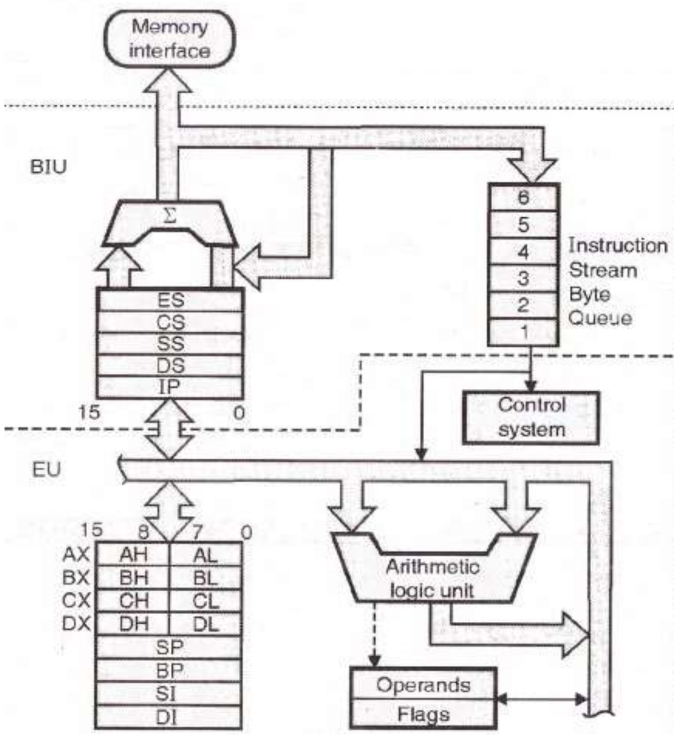


Fig. 2. 8086 Architecture [5]

“The 8086 microprocessor is a 16-bit microprocessor architecture developed by Intel. It has a 16-bit data bus, which means it can process data in 16-bit chunks at a time, allowing for larger data manipulation compared to 8-bit microprocessors. It has a 20-bit address bus, allowing it to address up to 1 MB of memory. This made it suitable for addressing large memory spaces and paved the way for advanced memory management techniques.

The 8086 uses a segmented memory model, where the 1 MB memory space is divided into multiple segments of up to 64 KB each. The combination of a segment register and an offset address is used to access memory, providing flexibility in memory addressing.

The 8086 has a rich set of registers, including four 16-bit general-purpose registers (AX, BX, CX, and DX), four 16-bit segment registers (CS, DS, ES, and SS) for addressing memory segments, and a 16-bit instruction pointer (IP) for soring the address of the next instruction to be executed” .[6]

IV. FLAG RESISTORS

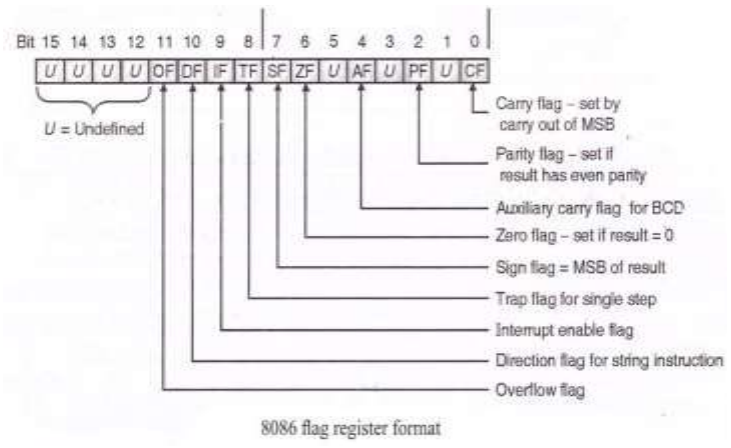

Bit 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
U U U U OF DF IF TF SF ZF U AF U PF U CF
U = Undefined
Carry flag – set by carry out of MSB
Parity flag – set if result has even parity
Auxiliary carry flag for BCD
Zero flag – set if result = 0
Sign flag = MSB of result
Trap flag for single step
Interrupt enable flag
Direction flag for string instruction
Overflow flag
8086 flag register format

Fig. 3. Flag Resistors [7]

“The 8086 flag register contents indicate the results of the computation in the ALU. It also contains some flag bits to control the CPU operations.

1. A 16-bit flag register is used in 8086. It is divided into two parts.

   - Condition code or status flags
   - Machine control flags

   A. The condition code flag register is the lower byte of the 16-bit flag register. The condition code flag register is identical to the 8085 flag register, with an additional overflow flag.

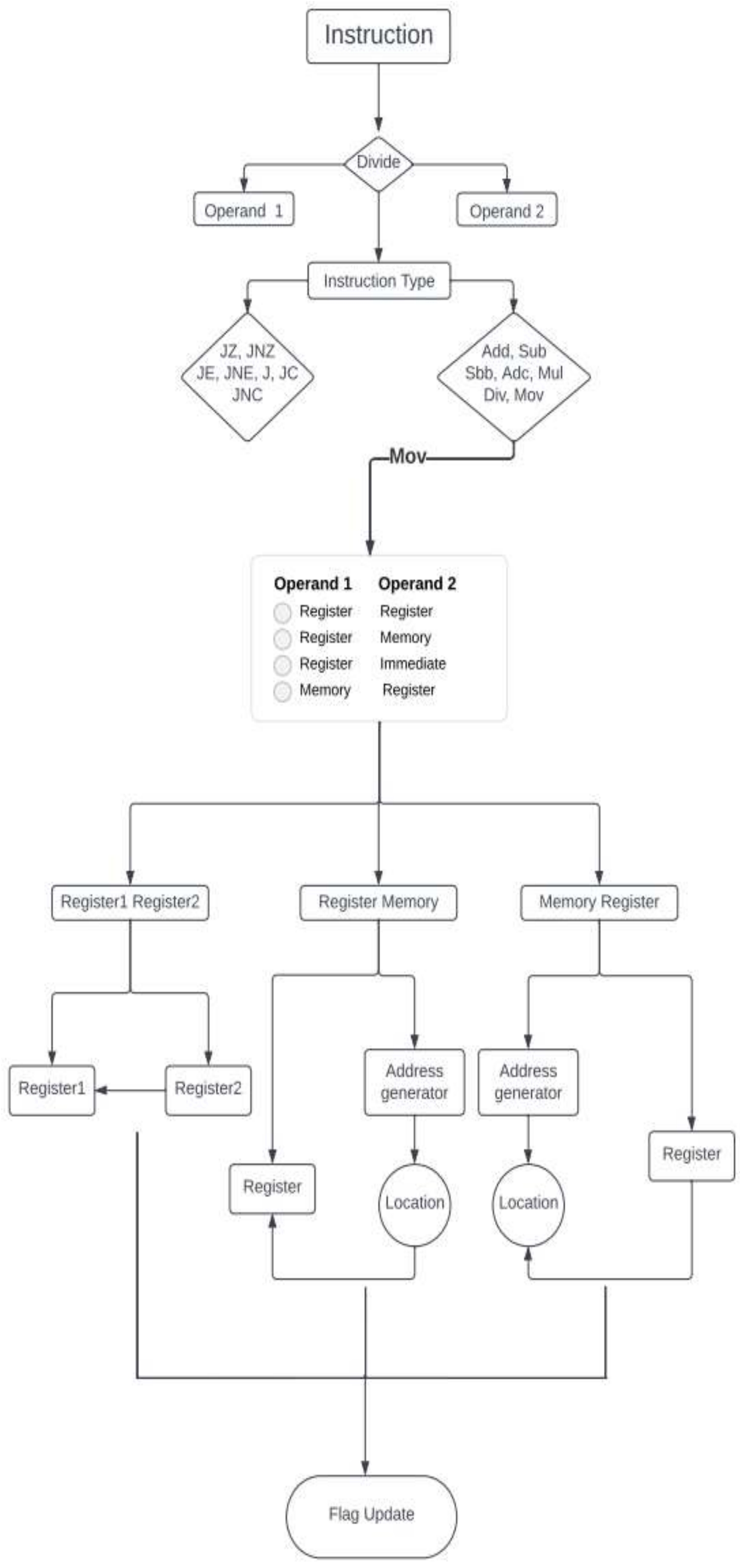
   B. The control flag register is the higher byte of the flag register. It contains three flags namely direction flag (D), interrupt flag (I), and trap flag (T).

2. It consists of 9 active flags out of 16. The remaining 7 flags marked ‘U’ are undefined flags”.[7]

   These 9 flags are of two types:

   - 6 Status flags
   - 3 Control flags

V. BLOCK DIAGRAM



VI. WORKING

The instruction will be breakdown into strings. first string signifies the instruction type, which is going to be executed, the second string refers to the 1st operand, and the third string refers rs the 2nd operand. The instruction set is divided into 2 categories. Cateory1 consists of [ADD, SUB, SUB, ADC, MUL, DIV, MOV] and category 2 contains jump type instructions (JZ, JNZ, JE, JNE, J, JC, JNC).

For example, in the case of Mov instruction, operands can be:
1. [register, register]
2. [register, memory]
3. [register, immediate]
4. [memory, register]

1. In the case of [register, register] type operands, the data is directly taken from the registers and loaded directly into the register.

The 8086 processor has a 20-bit address bus, which gives a physical address space of up to 1 MB (220), addressed as 00000h to FFFFFh. However, the maximum linear address space was limited to 64 KB because the internal registers are only 16 bits wide. A technique called "internal segmentation" has to be used for applications that are above the 64 KB boundary. There are four 16-bit segment registers (CS for "code segment," DS for "data segment," ES for "extra data segment," and SS for "stack segment").
All memory references are of the form [segment: offset], relative to the base address contained in the corresponding segment register.[8]

In particular, given a [segment: offset] pair, a 20-bit external (or physical) address is produced by segment × 10H + offset, where segment × 10H is called the segment address, which has its 16 most significant bits from the 16-bit segment register, and its four LSB's are all zeros. A 16-bit offset is always added to the 20-bit segment address to yield an external address. [8]

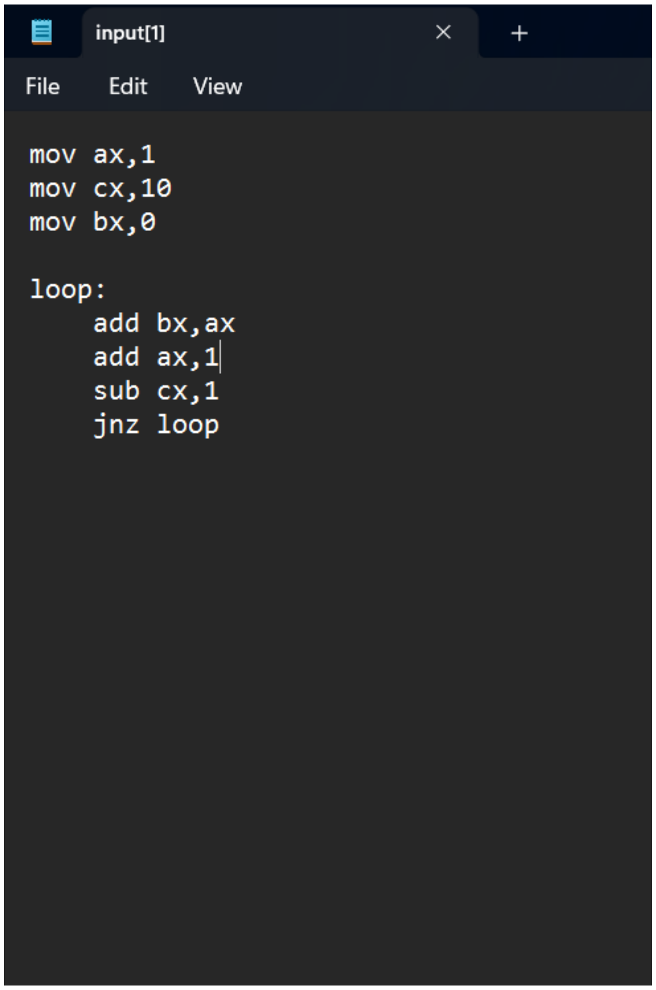
2. In the case of [register, memory] type operands, the physical address will calculate first then the data of the physical address will load in the register in the instruction.

3. In the case of [register, immediate] type operands, the immediate data will be stored in the physical address of 20-bit.

4. In the case of [Memory, register] type of operands, the physical address of 20 bits will be calculated first, then the data stored in the register will be stored in the location, addressed by the 2- bit's physical address.

VII. IMPLEMENTATION

Program:
Assembly language code for the addition of the first 10 natural numbers.

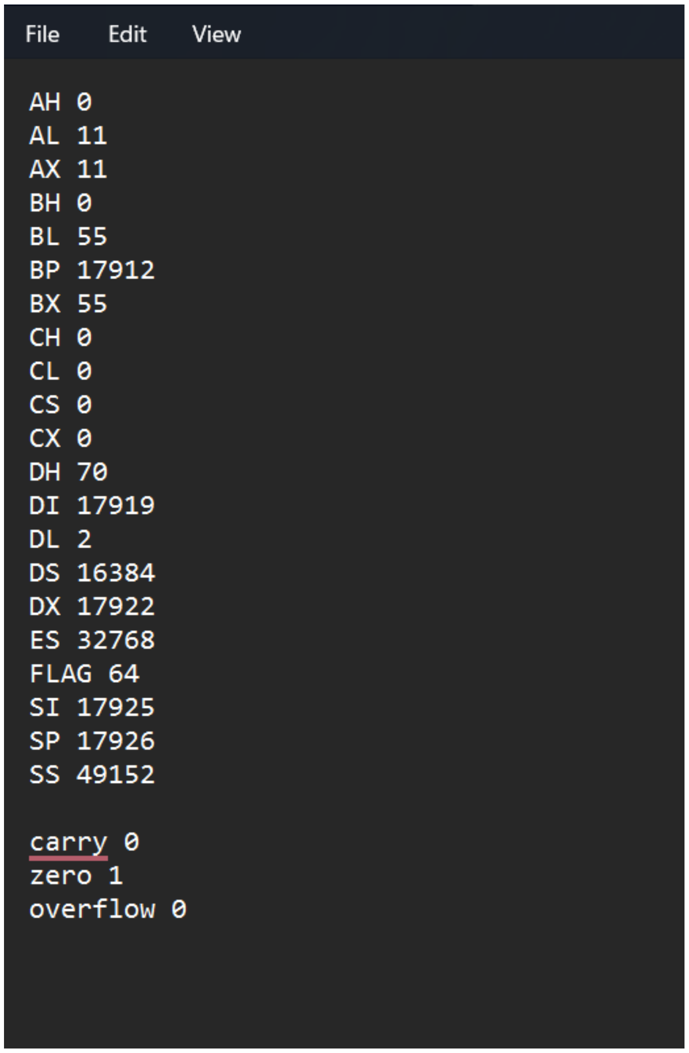


```
mov ax,1
mov cx,10
mov bx,0

loop:
    add bx,ax
    add ax,1
    sub cx,1
    jnz loop
```

Initially, the AX register is loaded with 1, the CX register is stored with 10 and the BX register is stored with zero,
Here BX is the Output or the sum of the first 10 natural number summations. CX is the counter which will decrease after every time enters the loop and terminate the loop when the CX= =0.

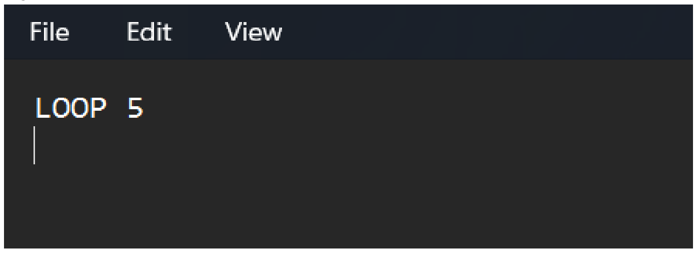
Output:



```
AH 0
AL 11
AX 11
BH 0
BL 55
BP 17912
BX 55
CH 0
CL 0
CS 0
CX 0
DH 70
DI 17919
DL 2
DS 16384
DX 17922
ES 32768
FLAG 64
SI 17925
SP 17926
SS 49152

carry 0
zero 1
overflow 0
```

Symbol table:



```
LOOP 5
```

VIII. CONCLUSION

The paper has discussed the design of the 8086 assembler and simulator which is capable of simulating various instructions such as data movement, arithmetic, logical, and jump instructions. This simulator is capable of detecting errors in the input instructions up to some extent.

This simulator supports every addressing mode for each instruction. This assembler generates the symbol table for handling the jump instruction. In the output section, we can see the value of every register after the execution of the program. The implementation of the simulator is done by designing various libraries such as registers, memory, alu, and so on to ease the development process.

IX. REFERENCES

[1] https://en.wikipedia.org/wiki/Assembly_language

[2] https://www.geeksforgeeks.org/introduction-of-assembler/

[3] https://sites.google.com/site/assignmentssolved/mca/semester3/mc0073/29

[4] https://www.studocu.com/in/document/g-h-raisoni-college-of-engineering/computer-networking/assembler-design-nice-to-teach/38405902

[5] https://www.includehelp.com/embedded-system/architecture-of-8086-microprocessor.aspx

[6] https://electronicsdesk.com/8086-microprocessor.html

[7] https://www.ques10.com/p/10786/explain-the-flag-register-of-8086-1/

[8] https://www.sciencedirect.com/topics/engineering/address-offset#:~:text=The%208086%20processor

Project Link

- ➢ https://github.com/dixit122/Simulator_8086_C-.git