BANASTHALI VIDYAPITH

MICROPROCESSORS AND MICROCONTROLLERS (ELE306L)

LAB FILE

ACADEMIC YEAR 2020 – 2021

Submitted by:

Submitted to:

Dr. Rahul Kumar Vijay

Department of Computer Science and Engineering

Table Of Contents

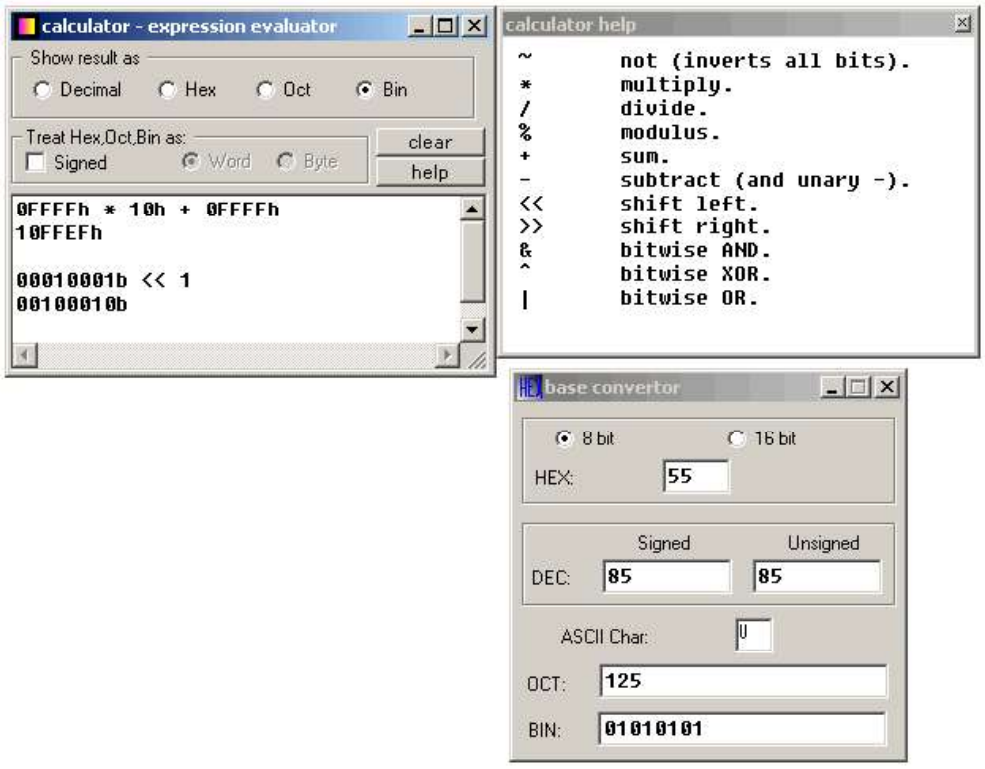
Introduction to emulator 8086

Emulator 8086 is a microprocessor emulator.Emu8086 combines an advanced source editor, assembler, disassembler, software emulator (Virtual PC) with debugger, and step by step tutorials. It compiles the source code and executes it on emulator step by step. One can watch registers, flags and memory while the program executes. Arithmetic & Logical Unit (ALU) shows the internal work of the central processor unit (CPU).
Emulator runs programs on a Virtual PC, this completely blocks your program from accessing real hardware, such as hard-drives and memory, since your assembly code runs on a virtual machine, this makes debugging much easier.8086 machine code is fully compatible with all next generations of Intel's microprocessors. This makes 8086 code very portable, since it runs both on ancient and on the modern computer systems. Another advantage of 8086 instruction set is that it is much smaller, and thus easier to learn.
There are some handy tools in emu8086 to convert numbers, and make calculations of any numerical expressions, all you need is a click on **Math** menu.
Base converter allows numbers conversion from any system and to any system. A value is typed in any text-box, and the value will be automatically converted to all other systems. It can work both with **8 bit** and **16 bit** values.
Multi base calculator can be used to make calculations between numbers in different systems and convert numbers from one system to another. Type an expression and press enter, result will appear in chosen numbering system. You can work with values up to **32 bits**.

Binary numbers must have "**b**" suffix, example:
00011011b

Hexadecimal numbers must have "**h**" suffix, and start with a zero
when first digit is a letter (A..F), example:
0ABCDh

Octal (base 8) numbers must have "**o**" suffix, example:
77o



Experiment 2.1

MOV Instruction

Aim:
To understand MOV instruction through the example of copying content of BL into AL register.

Software required:
Emu8086

Algorithm:
operand1 = operand2
(i.e. Copies operand2 to operand1)

Program:

```
CODE SEGMENT
START:
        MOV BL, 24H         ; SETS BL=24H
        MOV AL, BL          ; SETS AL=BL
        HLT
ENDS

END START
```

Output:

Before execution:

Register	Value
AL	00h
BL	00h

After execution:

Register	Value
AL	24h
BL	24h

Result:
The data 24h is moved to BL and then the data of BL is copied to AL successfully. After this the emulator is halted.

Conclusion:
Data can be copied from oprand2 to oprand1 by using MOV instruction.
Also, it is understood that MOV instruction cannot:

- Set the value of the CS and IP registers.
- Copy value of one segment register to another segment register (should copy to general register first).
- Copy immediate value to segment register (should copy to general register first).

Experiment 2.2

XCHG Instruction

Aim:
To understand XCHG instruction through the example of exchanging contents of AX and BX registers.

Software required:
Emu8086

Algorithm:
operand1 < - > operand2
(Exchange values of two operands.)

Program:

```
CODE SEGMENT
START:
        MOV AX, 2040H     ; SETS AX=2040H
        MOV BX, 5080H     ; SETS BX=5080H
        XCHG AX, BX       ; AX=5080H, BX=2040H
        HLT
ENDS

END START
```

Output:

Before execution:

Register	Value
AX	0000h
BX	0000h

After execution:

Register	Value
AX	5080h
BX	2040h

Result:
The content of two registers, AX and BX are exchanged successfully.

Conclusion:
Content of two operands can be exchanged by using XCHG instruction.

Experiment 2.3

ADD and ADC Instructions

Aim:
To understand the difference between ADD and ADC instruction by using them on same operand and observing the result.

Software required:
Emu8086

Algorithm:

ADD instruction: (Add)
operand1 = operand1 + operand2

ADC instruction: (Add with carry)
operand1 = operand1 + operand2 + CF

Program:

```
DATA SEGMENT

ENDS
CODE SEGMENT
START:
        MOV AX, DATA
        MOV DS, AX          ; SET DS= 0710H
        MOV [2476H], 90H
        MOV [2477H], 90H
        ADD  [2476H], 86H  ; PERFORMS 90H + 86H
        ADC  [2477H], 86H  ; PERFORMS 90H + 86H + 1 ,AS CF=1
        HLT
ENDS

END START
```

Output:

Before execution of add and adc:

Memory location	Value
0710:2476h	90h
0710:2477h	90h

After execution of add and adc:

Memory location	Value
0710:2476h	16h
0710:2477h	17h

CF=1

Result:
The result of ADD instruction is obtained at memory location 0710:2476 which is 16 while the result of ADC instruction is obtained at memory location 0710:2476 which is 17.Also CF is set to 1.

Conclusion:
The ADD instruction ignores the carry generated while ADC instruction adds carry to the result.

Experiment 2.4

SUB and SBB Instructions

Aim:
To understand the difference between SUB and SBB instruction by using them on same operand and observing the result.

Software required:
Emu8086

Algorithm:

SUB instruction: (Subtract)
operand1 = operand1 - operand2

SBB instruction: (Subtract with Borrow)
operand1 = operand1 - operand2 - CF

Program:

```
DATA SEGMENT

ENDS
CODE SEGMENT
START:
        MOV AX, DATA
        MOV DS, AX          ; SET DS= 0710H
        MOV [2476H], 20H
        MOV [2477H], 20H
        SUB  [2476H], 50H  ; PERFORMS 20H - 50H
        SBB  [2477H], 50H  ; PERFORMS 20H - 50 H - 1 ,AS CF=1
        HLT
ENDS

END START
```

Output:

Before execution of sub and sbb:

Memory location	Value
0710:2476h	20h
0710:2477h	20h

After execution of sub and sbb:

Memory location	Value
0710:2476h	0D0h
0710:2477h	0CFh

CF=1
SF=1

Result:
The result of SUB instruction is obtained at memory location 0710:2476 which is 0D0h while the result of SBB instruction is obtained at memory location 0710:2476 which is 0CFh.
Also, the carry flag and sign flag are set to 1.

Conclusion:
The SUB instruction ignores borrows while SBB instruction subtracts borrow from the result.