MM LAB FILE- ELE306L

Banasthali Vidyapith
BTech, CS-C
Microprocessors and Microcontrollers Lab
C1

Name: Shivani Paliwal
Roll No.: 1913107
ID: BTBTC19123
Class: Computer Science

Submitted to: Dr Urvashi Prakash Shukla Ma'am

TABLE OF CONTENT

EXPERIMENT - 1

AIM: INTRODUCTION OF EMULATOR 8086

Microprocessor Emulator, also known as EMU8086, is an emulator of the program 8086 microprocessor. It is developed with a built-in 8086 assembler. This application is able to run programs on both PC desktops and laptops. This tool is primarily designed to copy or emulate hardware. These include the memory of a program, CPU, RAM, input and output devices, and even the display screen.

MICROPROCESSOR 8086

It is a **16-bit microprocessor** and was created by Intel in 1978. It has 20 address lines and 16 data lines that provides up to 1MB storage. It consists of powerful instruction set, which provides operations like multiplication and division easily.

8086 has 4 general purpose registers, each register has its own name. General purpose registers are used to store temporary data within the microprocessor. There are 8 general purpose registers in 8086 microprocessors and they are discussed below.

GENERAL PURPOSE REGISTERS

1. AX – This is the accumulator. It is of 16 bits and is divided into two 8-bit registers AH and AL to also perform 8-bit instructions. It is generally used for arithmetical and logical instructions but in 8086 microprocessor it is not mandatory to have accumulator as the destination operand.

2. BX - This is the base register. It is of 16 bits and is divided into two 8-bit registers BH and BL to also perform 8-bit instructions. It is used to store the value of the offset**.**

3. CX – This is the counter register. It is of 16 bits and is divided into two 8- bit registers CH and CL to also perform 8-bit instructions. It is used in looping and rotation.

4. DX - This is the data register. It is of 16 bits and is divided into two 8-bit registers DH and DL to also perform 8-bit instructions. It is used in multiplication an input/output port addressing.

POINTER REGISTERS

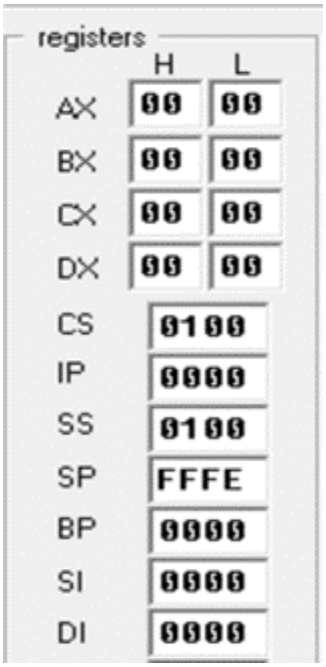
5. SP – This is the stack pointer. It is of 16 bits. It points to the topmost item of the stack. If the stack is empty the stack pointer will be at (FFFE)H. It's offset address relative to stack segment.

6. BP – This is the base pointer. It is of 16 bits. It is primary used in accessing parameters passed by the stack. It's offset address relative to stack segment.

INDEX REGISTERS

7. SI – This is the source index register. It is of 16 bits. It is used in the pointer addressing of data and as a source in some string related operations. Its offset is relative to data segment.

8. DI – This is the destination index register. It is of 16 bits. It is used in the pointer addressing of data and as a destination in some string related operations. Its offset is relative to extra segment.



REGISTERS IN EMU 8086

- The main purpose of a register is to store value of a variable. Therefore, when we modify any of the 8-bit registers 16-bit register is also updated, and vice-versa.
- "H" is for the high part of the register and "L" is for low part of the register.
- Registers are located inside the CPU; they are much faster than memory. Accessing a memory location requires the use of a system bus, so it takes much longer. Accessing data in a register usually takes no time. Therefore, it is preferred to keep variables in the registers.
- Register sets are very small and most registers have special purposes which limit their use as variables, but they are still an excellent place to store temporary data of calculations.

FLAG REGISTERS

It is a 16-bit register that behaves like a flip-flop, i.e., it changes its status according to the result stored in the accumulator. It has 9 flags and they are divided into 2 groups − Conditional Flags and Control Flags.

CONDITIONAL FLAGS

It represents the result of the last arithmetic or logical instruction executed. Following is the list of conditional flags:

1. CARRY FLAG - This flag indicates an overflow condition for arithmetic operations. It is the carry out from the MSB bit on addition or borrow into MSB bit on subtraction.

2. AUXILIARY FLAG - When an operation is performed at ALU, it results in a carry/barrow from lower nibble (i.e., D0 – D3) to upper nibble (i.e., D4 – D7), then this flag is set.

3. ZERO FLAG - This flag is set to 1 when the result of arithmetic or logical operation is zero else it is set to 0.

4. PARITY FLAG - This flag is used to indicate the parity of the result, i.e., when the lower order 8-bits of the result contains even number of 1's, then the Parity Flag is set.

5. SIGN FLAG - This flag holds the sign of the result, i.e., when the result of the operation is negative, then the sign flag is set to 1 else set to 0.

6. OVERFLOW FLAG - This flag represents the result when the system capacity is exceeded.

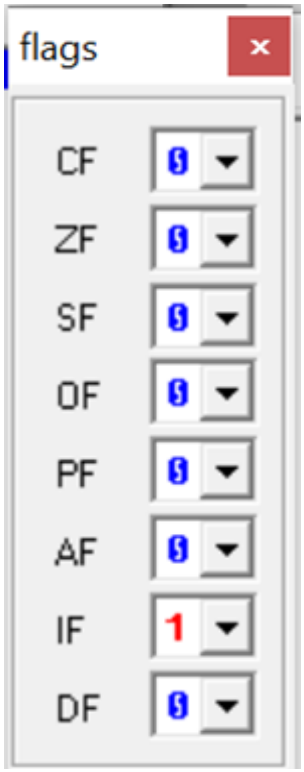
CONTROL FLAGS

Control flags controls the operations of the execution unit. Following is the list of control flags −

1. **TRAP FLAG**− It is used for single step control and allows the user to execute one instruction at a time for debugging. If it is set, then the program can be run in a single step mode.

2. **INTERRUPT FLAG**− It is an interrupt enable/disable flag, i.e., used to allow/prohibit the interruption of a program. It is set to 1 for interrupt enabled condition and set to 0 for interrupt disabled condition.

3. **DIRECTION FLAG** − It is used in string operation. As the name suggests when it is set then string bytes are accessed from the higher memory address to the lower memory address and vice-a-versa.



FLAGS IN EMU 8086

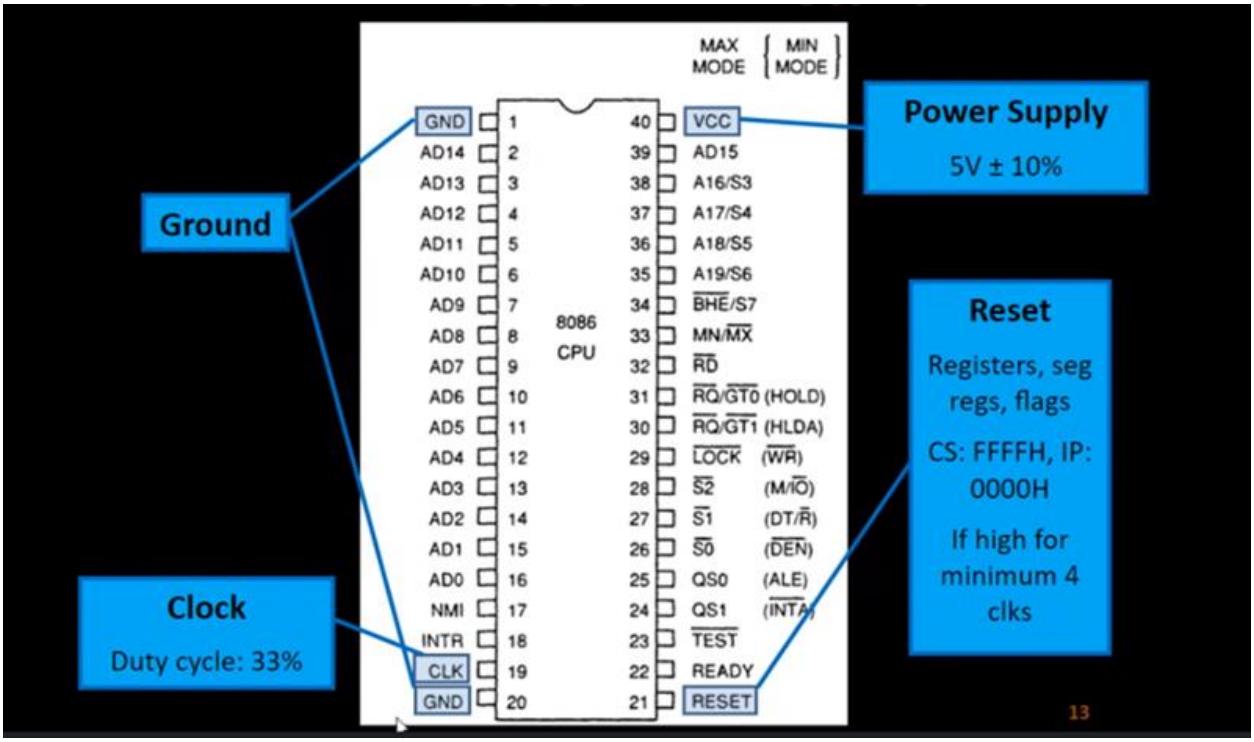
EXPERIMENT – 2

AIM: Instruction Set: Programming and Illustration

PIN DETAILS – 8086

An 8086 microprocessor is a 40 pin IC and has few separate pin configurations for minimum and maximum mode.

PIN DESCRIPTION

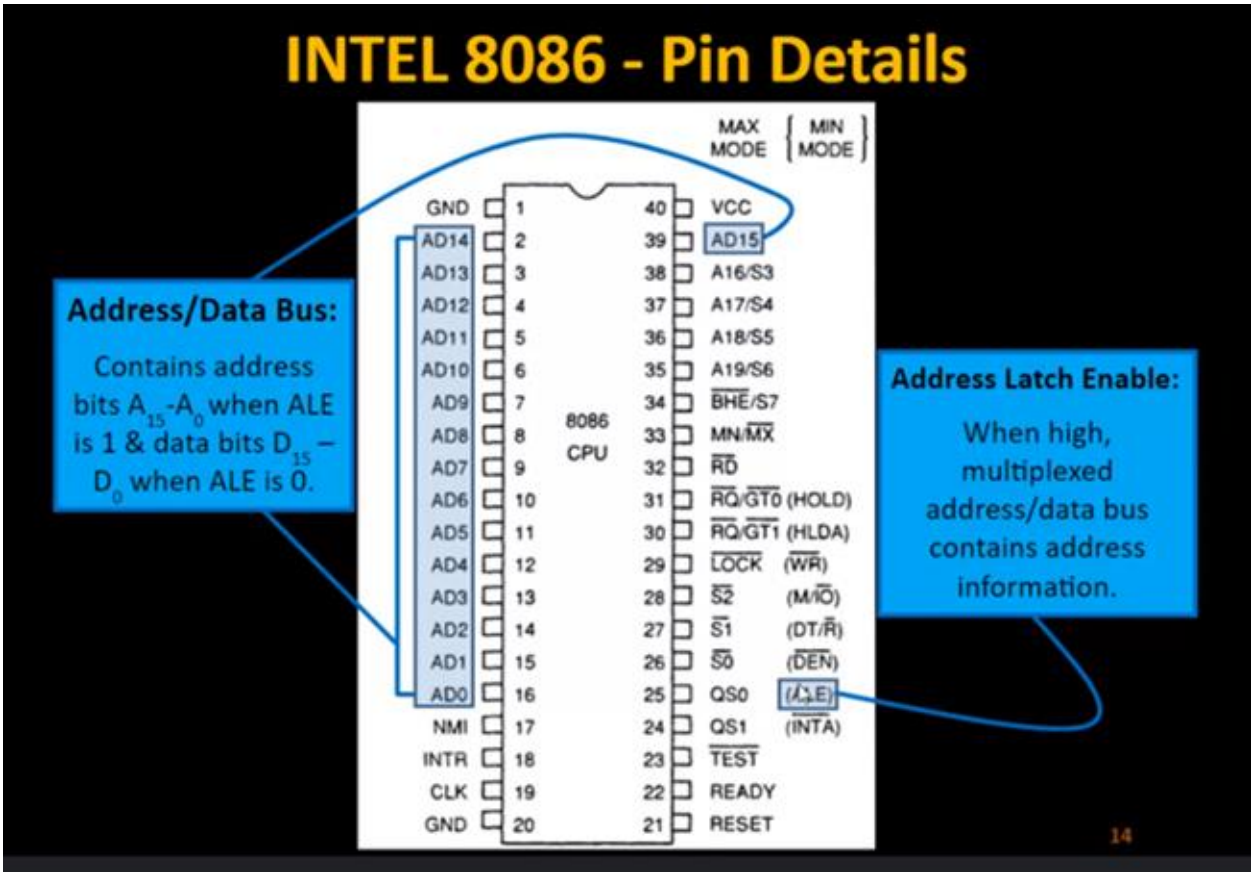


1. V_{CC} – Pin number 40 – At this pin, the external power supply of +5V is provided to the processor.

2. GND– Pin number 1 and 20 – These two pins act as the ground. This pin directs the extra current of the microprocessor to ground.

3. RESET – **Pin number 21** – Whenever this pin is enabled then it resets the processor and other devices connected to the system by immediately terminating the recent task.

4. CLK – **Pin number 19** – A signal at this pin provides the timing to the internal operations that are being executed inside the microprocessor.



5. AD_0 – AD_{15} – Pin number 2 to 16 and 39 – These are the multiplexed address and data bus. Since, 8086 microprocessor has 20-bit address bus and 16-bit data bus. So, the 16 lines of the address and data bus are multiplexed together so as to reduce the number of lines inside the IC.

6. ALE – ***Pin number 25*** – ALE is an abbreviation for address latch enable. Whenever an address is present in the multiplexed address and data bus, then the microprocessor enables this pin. If ALE is 0, AD0-AD15 represents data lines and if ALE is 1, they are Address lines.

7. A_{16}/S_3, A_{17}/S_4, A_{18}/S_5 and $A_{19}S_6$ – Pin number 35 to 38 – Out of 20 address bits, 4 are present in the multiplexed form with the status signals. In the case of memory operations, these pins act as an address bus and contain the memory address of any particular instruction or data. However, from I/O operations these pins are low that shows the status of the processor.

- The signal at S_3 and S_4 show that which segment is currently accessed by the microprocessor among the four segments present in it.

The table below will show the encoding of S_3 and S_4:

S3	S4	STATUS
0	0	Extra Segment
0	1	Stack Segment
1	0	Code Segment/idle
1	1	Data Segment

- S_5, when enabled, shows the presence of an interrupts in the microprocessor. So, basically, it serves as an **interrupt flag**.
- S_6 shows the status of the bus master for the current operation. It shows whether the 8086 is the bus master or any other proficient device is acting as the bus master.
- When 0 is present as the signal at this pin then it indicates the 8086 is holding the access of the bus otherwise it is high i.e., 1.

8. BHE' / S_7 – ***Pin number 34*** – BHE is an acronym for Bus High Enable. The combination of the BHE signal and S_7 status informs about the existence of the

data on the bus. Also, different combinations show whether the bus is containing overall 16-bit, upper byte or lower byte of the data.

9. MN/MX' – ***Pin number 33*** –The status at this particular pin shows whether the processor is operating in the minimum mode or maximum mode.
A signal 0 at this pin informs that the 8086 is operating in maximum mode i.e., multiple processors. While signal 1 shows the operation under minimum mode i.e., single processor.

10. RD' – ***Pin number 32*** – An active low signal at this pin shows that the microprocessor is performing read operation with either memory or I/O devices.

11. NMI – ***Pin number 17*** – NMI is non-maskable interrupt. These are basically uncontrollable interrupts generated inside the processor. When an NMI occurs, then an interrupt service routine is generated by the interrupt vector table.

12. TEST – ***Pin number 23*** – This pin basically shows the wait instruction. Whenever a low signal at this pin occurs then the processing inside the processor continues. As against, in case of the high signal, the processor has to wait for the disabling of this pin.

13. INTR – ***Pin number 18*** – INTR stands for an interrupt request. The processor after each clock cycle samples the INTR and if the signal at this pin is found to be high then the processor controls that interrupt internally.

14. READY – ***Pin number 22*** – This signal is used by the peripherals and memory devices in order to show the readiness for the next operation.

15. RESET – ***Pin number 21*** – Whenever this pin is enabled then it resets the processor and other devices connected to the system by immediately terminating the recent task.

PINS IN MINIMUM MODE

16. INTA' – ***Pin number 24*** – It is an interrupt acknowledge pin. Whenever an INTR signal is generated, then the microprocessor generates INTA signal, as a response to that interrupt.

17. DEN' – ***Pin number 26*** – DEN is used for data enable. This is an active low pin that means whenever a 0 is present at this pin then the transceiver gets enabled and it separates the data from the multiplexed address and data bus.

18. DT/R' – ***Pin number 27*** – This pin is used to show whether the data is getting transmitted or is received. A high signal at this pin provides the information regarding the transmission of data. While a low indicates reception of data.

19. M/IO' – ***Pin number 28*** – This pin indicates whether the processor is performing an operation with memory or I/O devices. Whenever a high is present at this pin then it shows the operation is carried out through the memory. While a low signal shows operation through I/O devices.

20. WR' – ***Pin number 29*** – An active low signal at this pin indicates that the processor is performing write operation from either memory or I/O devices.

21. HOLD – ***Pin number 31*** – When an external device enables this pin then the processor stops accessing the buses immediately after the recent task gets over.

22. HLDA – ***Pin number 30*** – This pin is used as a response pin for the hold request. Once request for accessing the buses is produced by an external entity. Then the microprocessor acknowledges the device that its request will be considered once it gets over by the current operation.

PINS IN MAXIMUM MODE

23. S_0', S_1'and S_2'– ***Pin number 26 to 28*** – These are basically 3 status pins and are active low. This means that if the status at all the 3 pins is 0 then it shows that multiple interrupts are to be handled in maximum mode.

The table below is representing the status of the processor in different combinations:

S0	S1	S2	STATUS
0	0	0	INTA
0	0	1	R/IO
0	1	0	W/IO
0	1	1	HALT
1	0	0	R/M
1	1	0	W/M
1	1	1	NONE

24. QS_0 and QS_1 – ***Pin number 24 and 25*** – These two pins indicate the status of the 6-byte pre-fetch queue present in the architecture of 8086.

QS_0	QS_1	STATUS
0	0	No Operation
0	1	First Byte from queue
1	0	Empty queue
1	1	Subsequent byte from queue

25. LOCK' – ***Pin number 29*** –This pin is involved in maximum mode operation. So, basically, when a single processor is accessing the buses and peripherals then it locks the resources being used by it. So, that no other entity can access it until the recent processor frees it.

26. RQ'/ GT_0'and RQ'/ GT_1'– Pin ***number 30 and 31*** – Due to the involvement of multiple processors, these pins indicate the request and grant permission for accessing the buses, memory and peripherals.

8086 INSTRUCTION SETS

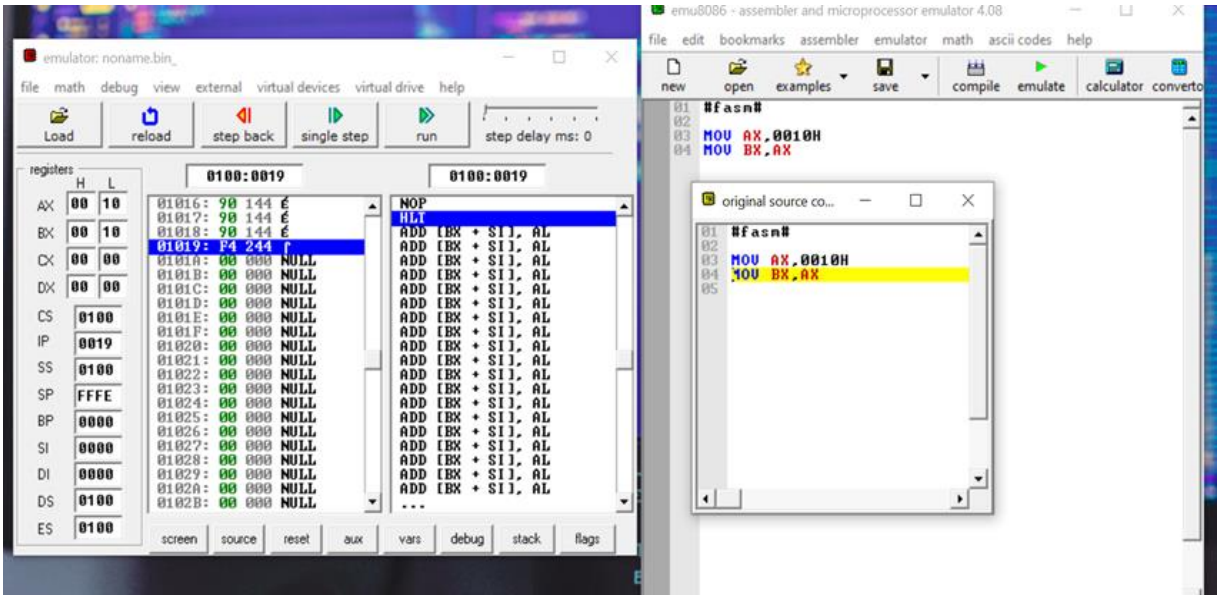
Data Transfer Instructions

These instructions are used to transfer the data from the source operand to the destination operand. Following are the list of instructions under this group −

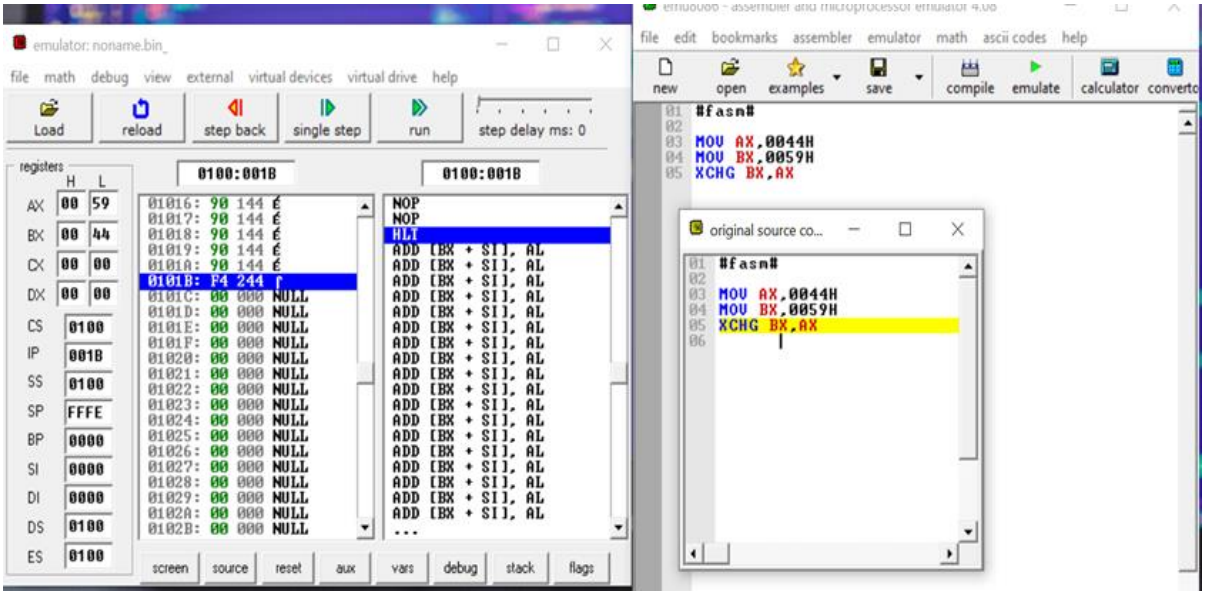
Instruction to transfer a word :

- MOV − Used to copy the byte or word from the provided source to the provided destination.

  MOV reg, data / MOV reg1, reg2 / MOV reg, mem / MOV mem, data



- XCHG − Used to exchanges contents of Source with destination. It cannot exchange two memory locations directly. The contents of AL are exchanged with BL. The contents of AH are exchanged with BH.
  e.g.: XCHG BX, AX; XCHG [5000H], AX;



- **XLAT** − Used to translate a byte in AL using a table in the memory. It is also known as translate instruction.
- **PUSH**: Source can be register, segment register or memory. This instruction pushes the contents of specified source on to the stack. In this stack pointer is decremented by 2. The higher byte data is pushed first (SP-1).Then lower byte data is pushed (SP-2)

  e.g.: PUSH AX;

  PUSH DS;

  PUSH [5000H];

- **POP:** Destination can be register, segment register or memory. This instruction pops the contents of specified destination. In this stack pointer is incremented by 2. The lower byte data is popped first (SP+1). Then higher byte data is popped (SP+2)

  e.g.: POP AX;

POP DS;

POP [5000H];

Instructions for input and output port transfer :

- **IN** − Used to read a byte or word from the provided port to the accumulator.

  e.g.: IN AL, 80H; IN AX, DX; //DX contains address of 16-bit port.

- **OUT** − Used to send out a byte or word from the accumulator to the provided port.

  e.g.: OUT 80H, AL; OUT DX, AX; //DX contains address of 16-bit port.

Instructions to transfer the address

- **LEA** − LEA Also known as Load Effective Address (LEA). It loads effective address formed by the destination into the source register.

  e.g.: LEA BX, Address;

  LEA SI, Address [BX];

- **LDS** − Used to load DS register and other provided register from the memory
- **LES** − Used to load ES register and other provided register from the memory.

Instructions to transfer flag registers

- **LAHF** − Used to load AH with the low byte of the flag register.

  e.g.: LAHF

- **SAHF** − Used to store AH register to low byte of the flag register.

  e.g.: SAHF

- **PUSHF** − Used to copy the flag register at the top of the stack.

  e.g.: PUSHF

- **POPF** − Used to copy a word at the top of the stack to the flag register.

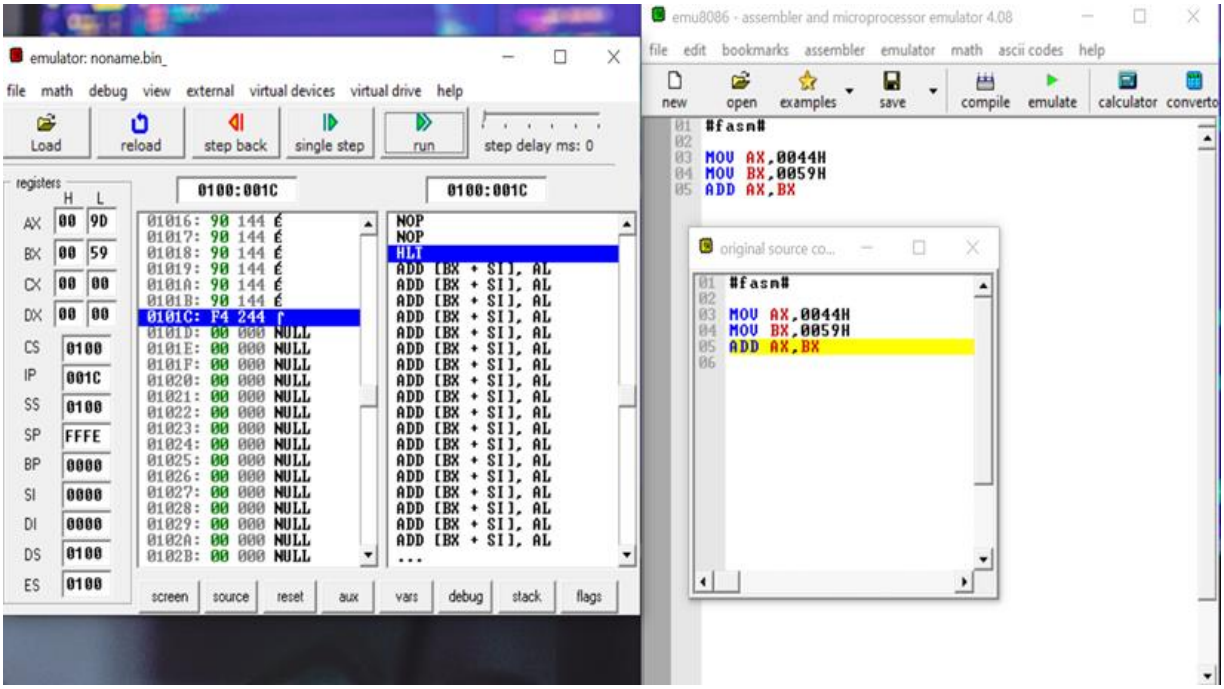
e.g.: POPF

Arithmetic Instructions

These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.

- **ADD**: This instruction adds the contents of source operand with the contents of destination operand. The source may be immediate data, memory location or register. The destination may be memory location or register. The result is stored in destination operand.AX is the default destination register.

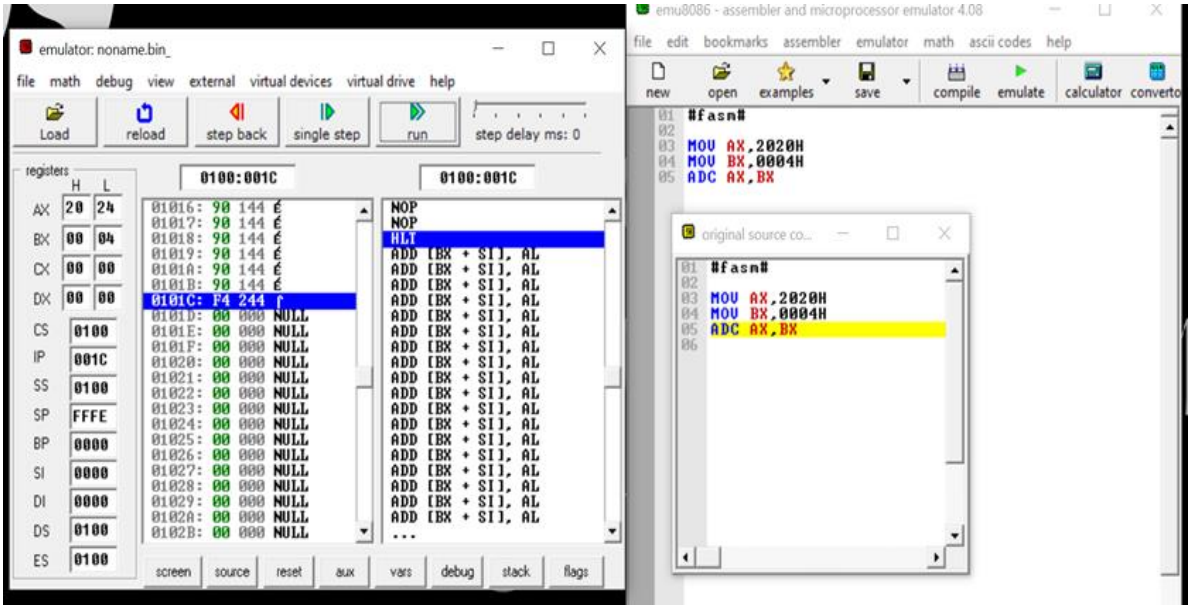
  e.g.: ADD AX,2020H;
  ADD AX, BX;



- **ADC**: This instruction adds the contents of source operand with the contents of destination operand with carry flag bit. The source may be immediate data, memory location or register. The destination may be memory location or register. The result is stored in destination operand.AX is the default destination register.

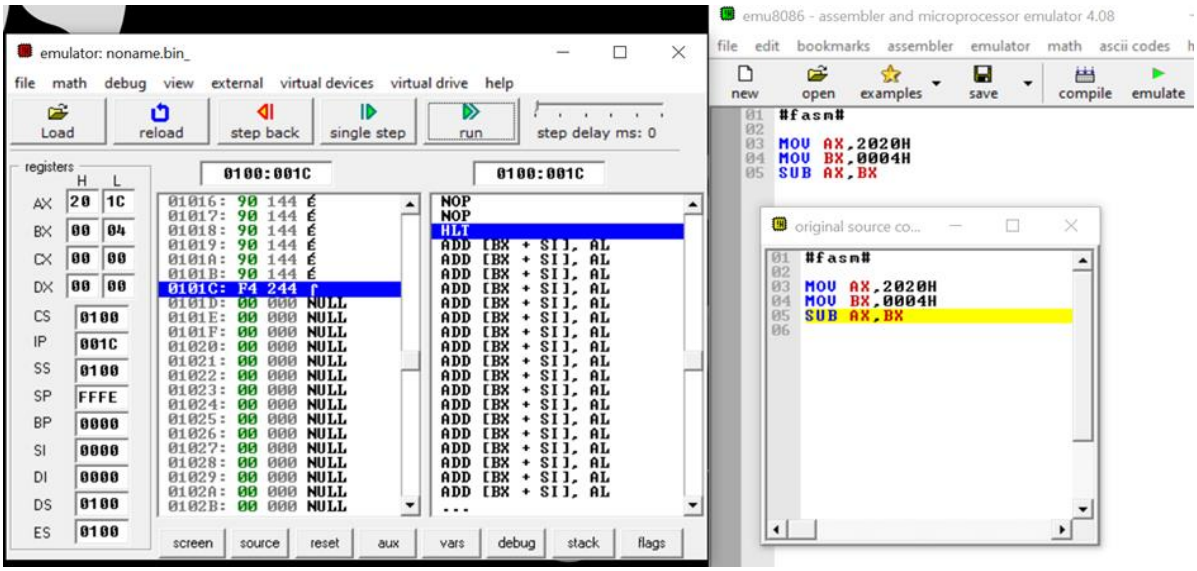
e.g.: ADC AX,2020H;

ADC AX, BX;



- **SUB**: This instruction subtracts the contents of source operand from contents of destination. The source may be immediate data, memory location or register. The destination may be memory location or register. The result is stored in the destination place.
  e.g.: SUB AX,1000H;

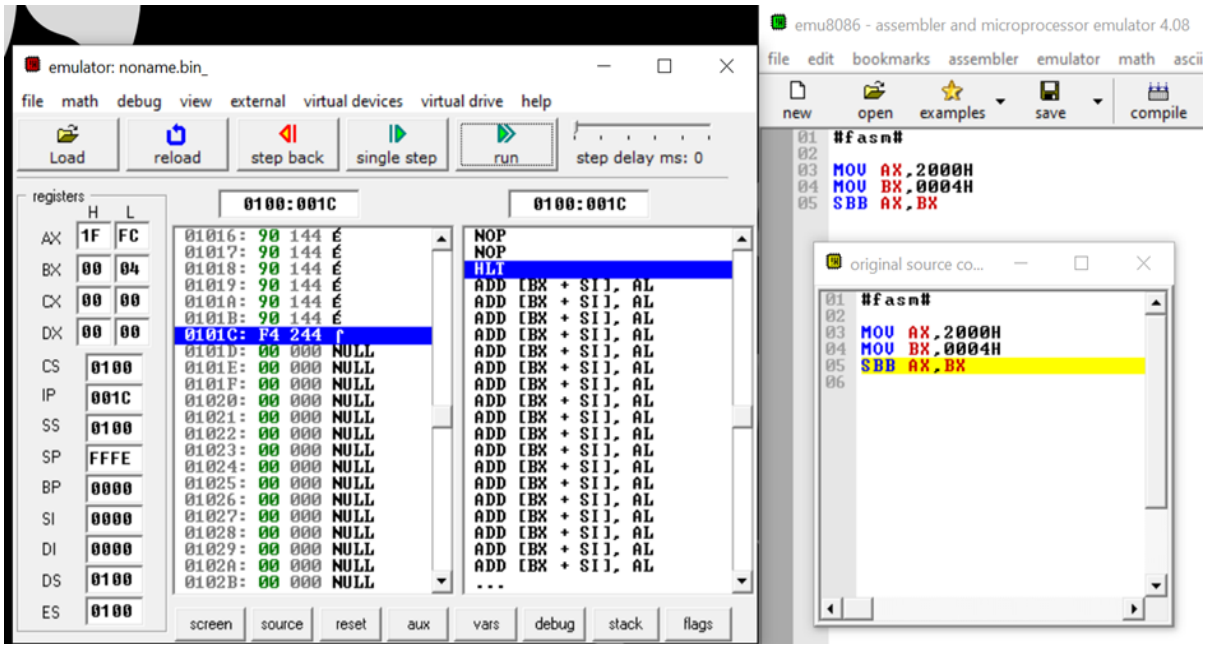
  SUB AX, BX;



- **SBB**: Also known as Subtract with Borrow. This instruction subtracts the contents of source operand & borrow from contents of destination operand. The source may be immediate data, memory location or register. The destination may be memory location or register. The result is stored in the destination place.

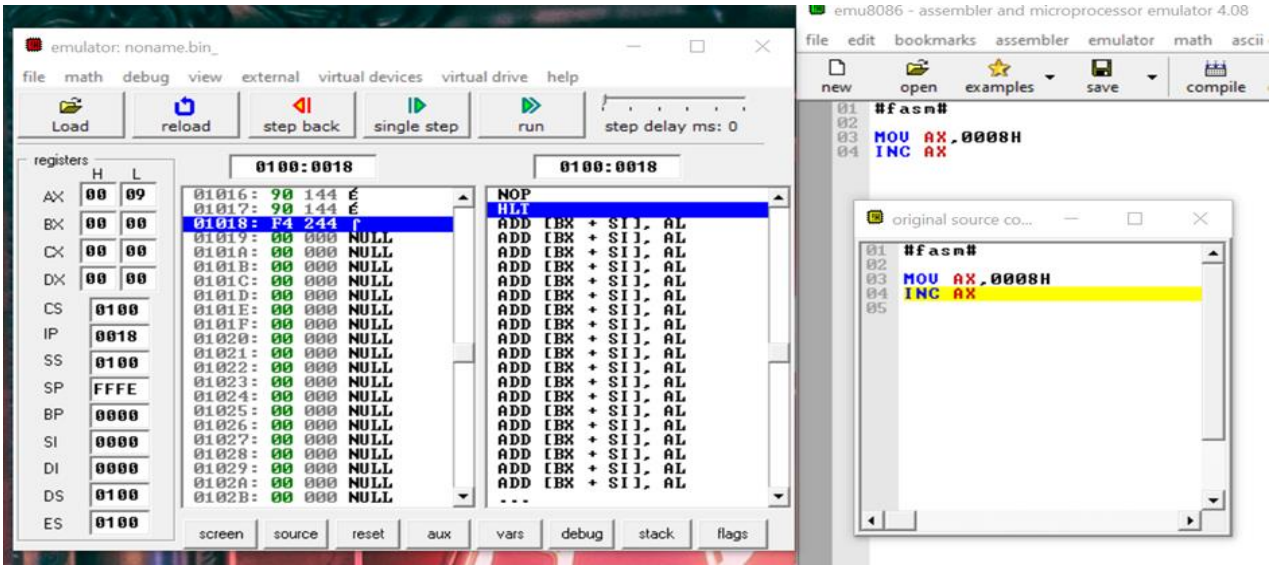
  e.g.: SBB AX,1000H;

  SBB AX, BX;

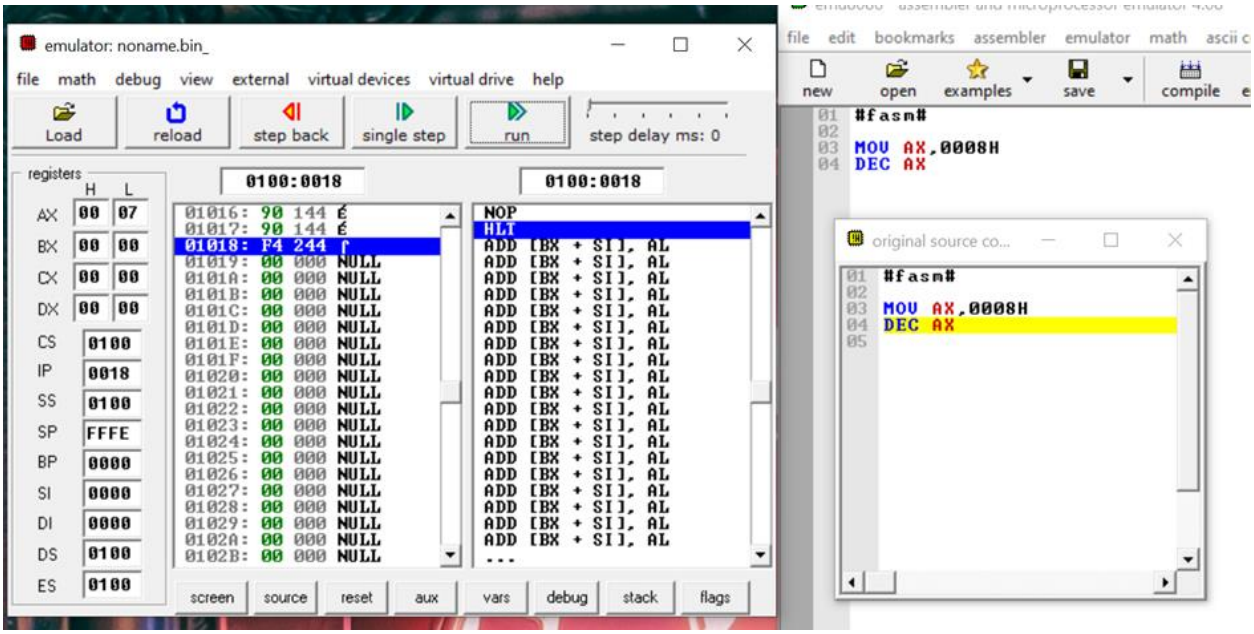


- **INC**: This instruction increases the contents of source operand by 1. The source may be memory location or register. The source cannot be immediate data. The result is stored in the same place.

  e.g.: INC AX; INC [5000H];



- **DEC:** This instruction decreases the contents of source operand by 1. The source may be memory location or register. The source cannot be immediate data. The result is stored in the same place.
  e.g.: DEC AX;
  DEC [5000H];

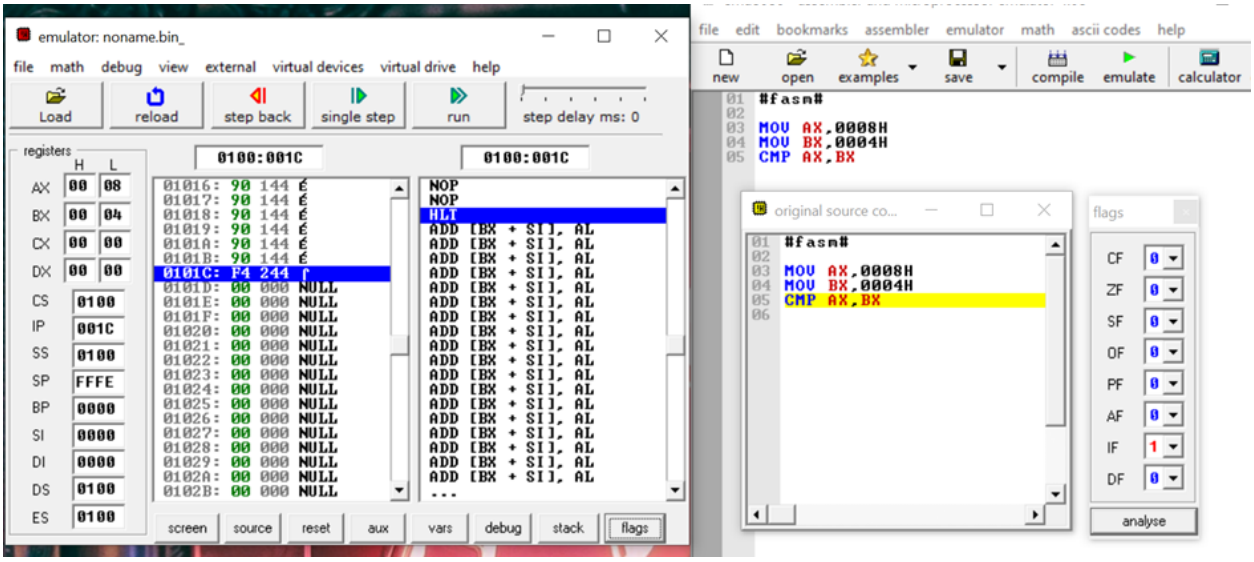


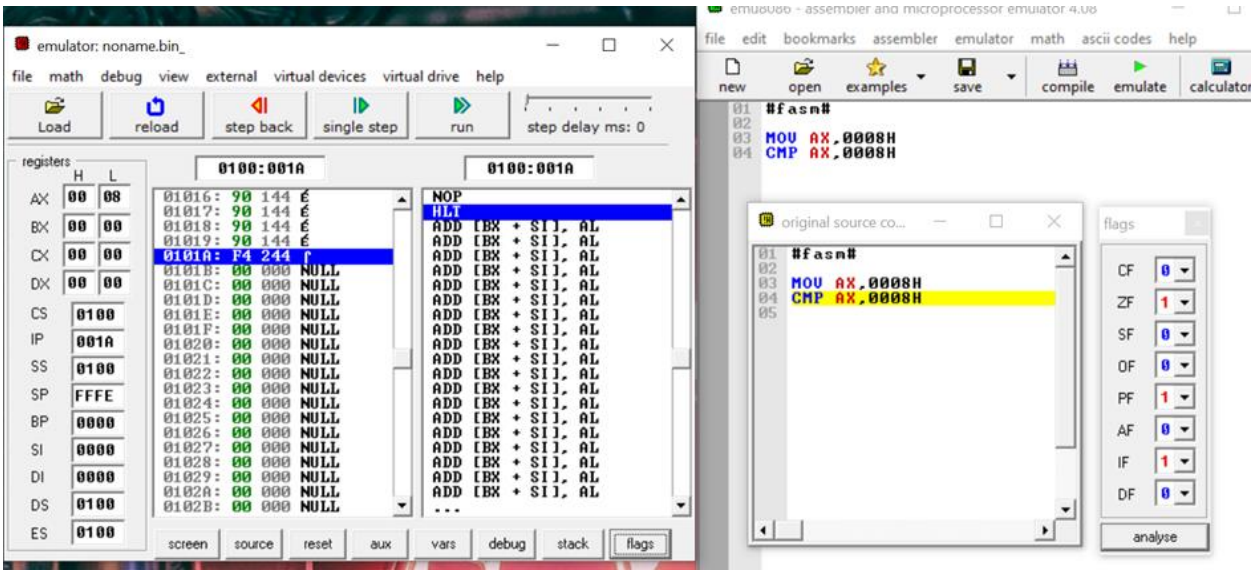
- **CMP**: Known as Compare. This instruction compares the contents of source operand with the contents of destination operands. The source may be immediate data, memory location or register. The destination may be

memory location or register. Then resulting carry & zero flag will be set or reset.

e.g.: CMP AX,1000H;

CMP AX, BX;





- **AAA**: known as ASCII Adjust After Addition. This instruction is executed after ADD instruction.

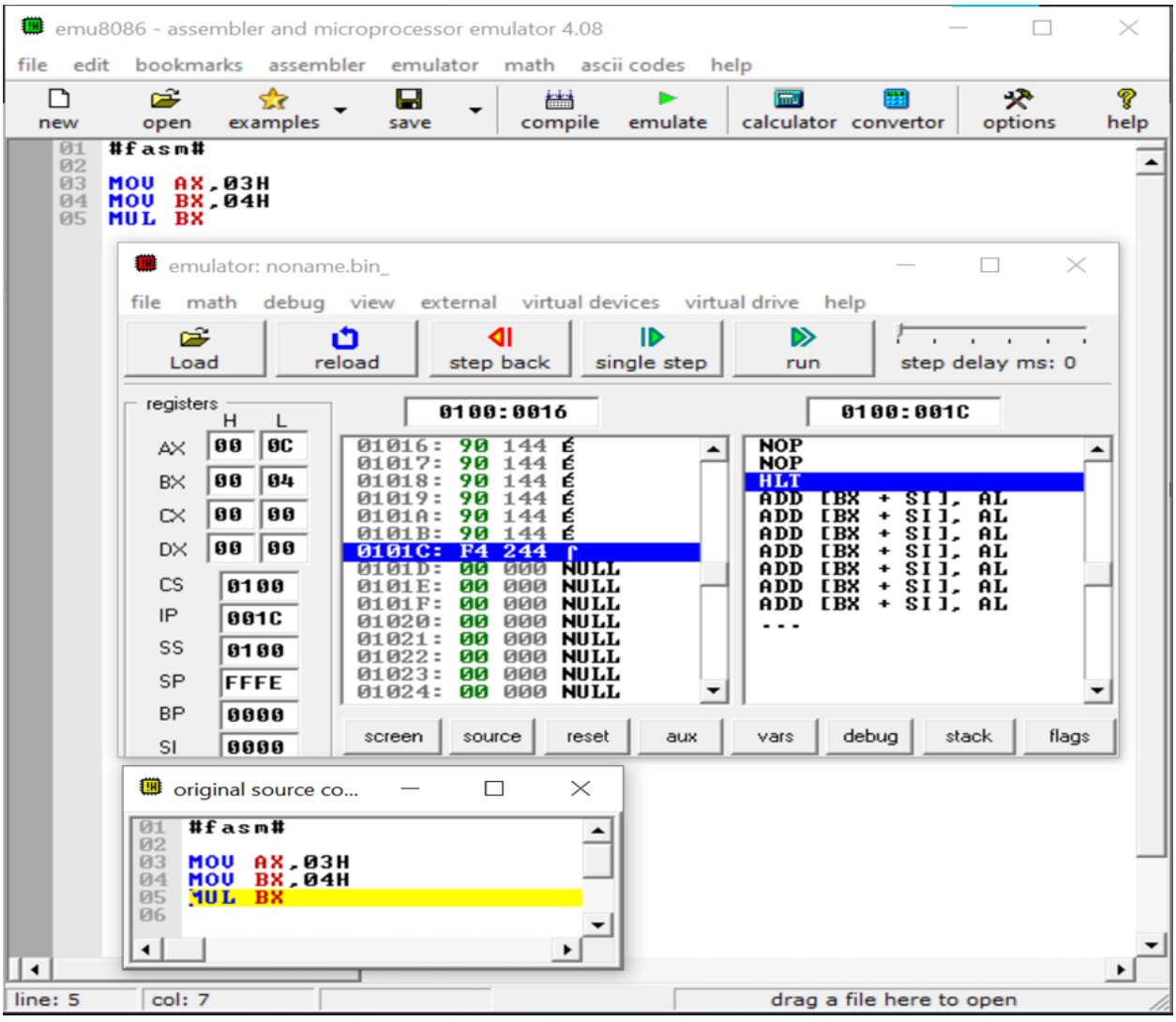
(1). IF lower bits of AL<=09 then, Higher bits of AL should be loaded with zeroes. There should be no change in lower bits of AL. AH also must be cleared.

(2). IF lower bits of AL>09 then, Bits of AL must be incremented by 06 (i.e., AL+0110). Bits of AH must be incremented by 01 (i.e., AH+0001). Then higher bits of AL should be loaded with 0000. e.g.: AAA;

- **MUL**: Unsigned Multiplication: Operand contents are positively signed. Operand may be general purpose register or memory location. Result is stored in accumulator (AX). When operand is a byte: AX = AL * operand. when operand is a word:(DX AX) = AX * operand.

  e.g.: MUL BH // AX= AL*BH; // (+3) * (+4) = +12.

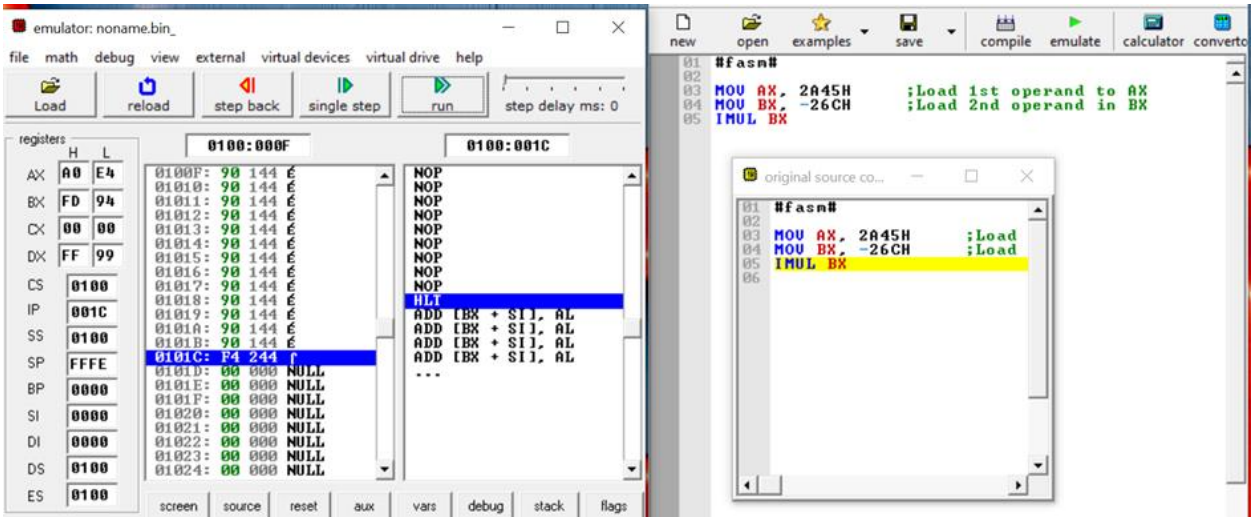
  MUL CX // AX=AX*CX.



- **IMUL**: It is the signed multiplication. Operand contents are negatively signed. Operand may be general purpose register, memory location or index register. If operand is of 8-bit then multiply it with contents of AL. If operand is of 16-bit then multiply it with contents of AX. Result is stored in accumulator (AX).

  e.g.: IMUL BH // AX= AL*BH; // (-3) * (-4) = 12.

  IMUL CX // AX=AX*CX;



- **DIV**: Unsigned Division: Operand may be register or memory. Operand contents are positively signed. Operand may be general purpose register or memory location.AL=AX/Operand (8-bit/16-bit) & AH= Remainder.

  e.g.: MOV AX, // AX=0203MOV BL, // BL=04DIV BL // AX=0203/04=80 (i.e., AL=50 & AH=00)

- **IDIV**: Signed Division-Operand may be register or memory. Operand contents are negatively signed. Operand may be general purpose register or memory location. when operand is a byte: AL = AX / operand AH = remainder (modulus)when operand is a word: AX = (DX AX) / operand DX = remainder (modulus).

  e.g.: MOV AX, // AX=-0203MOV BL, // BL=04DIV BL // AL=-0203/04=-50 (i.e., AL=-80 & AH=00)

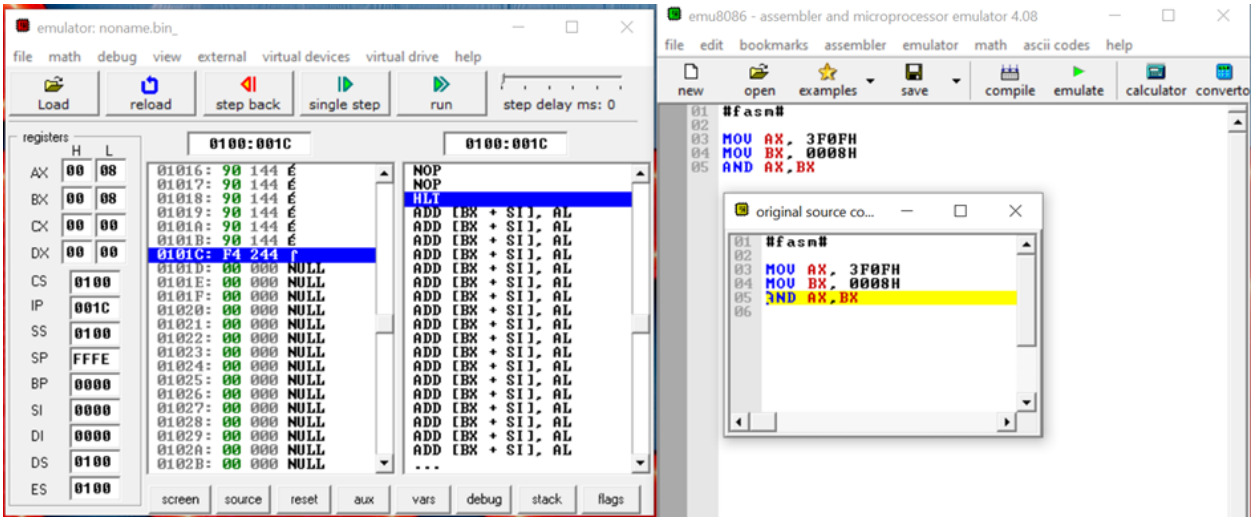
Logical Instructions

- **AND**: Destination operand may be register, memory location. Source operand may be register, immediate data or memory location. Result is stored in destination operand.

  e.g.: MOV AX, 3F0FH // AX=3F0FH

  MOV BX, 0008H // BX=0008H

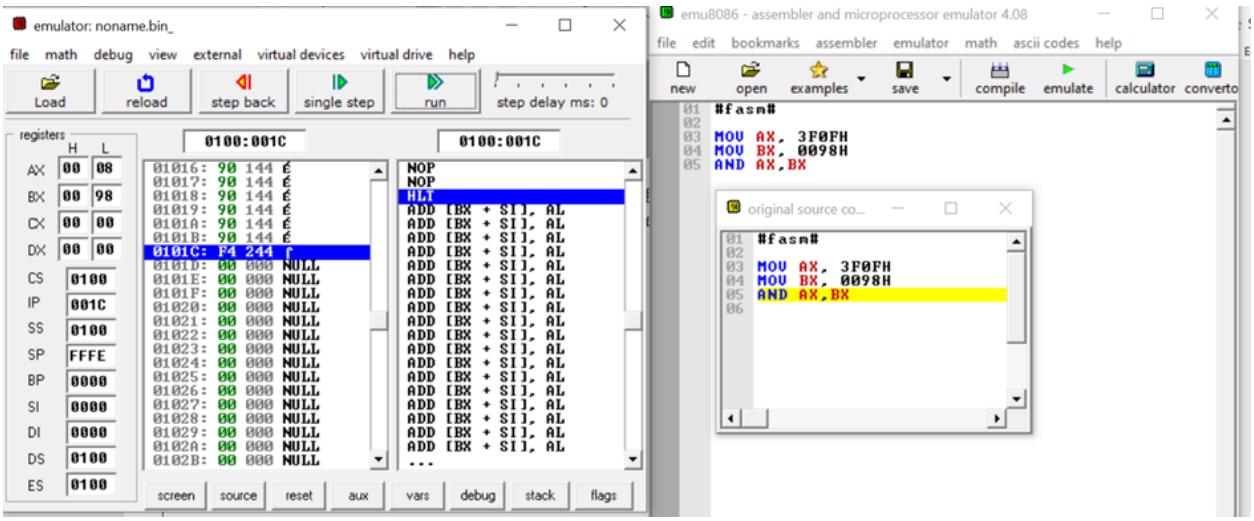
  AND AX, BX // AX=0008H.



- **OR:** Destination operand may be register, memory location. Source operand may be register, immediate data or memory location. Result is stored in destination operand.
  e.g.: MOV AX, 3F0FH // AX=3F0FH

  MOV BX, 0098H // BX=0098H

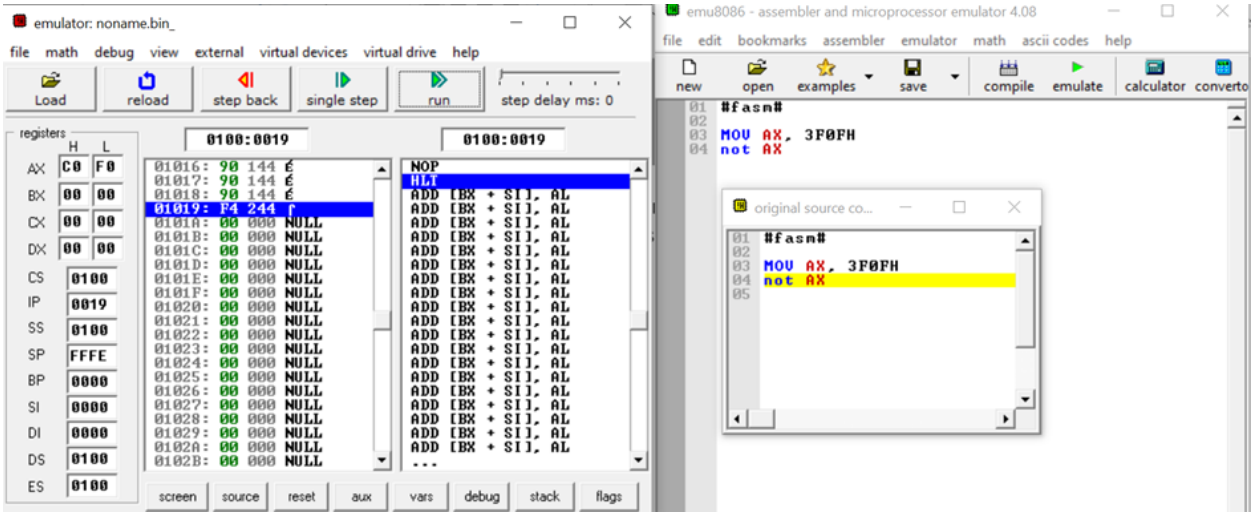
  OR AX, BX // AX=3F9FH.



- **NOT**: Operand may be register, memory location. This instruction inverts (complements) the contents of given operand. Result is stored in Accumulator (AX).

  e.g.: MOV AX, 0200FH // AX=200FH

  NOT AX // AX=DFF0H



- **TEST**: Both operands may be register, memory location or immediate data. This instruction performs bit by bit logical AND operation for flags only (i.e., only flags will be affected). If the corresponding 0th bit of result contains '1' then result will be non-zero & zero flag will be cleared/reset

(i.e. ZF=0). If the corresponding 0th bit of result contains '0' then result will be zero & zero flag will be set (i.e. ZF=1).

e.g.: TEST AX, BX

TEST [0500],06H

Shift and Rotate Instructions: SHL/SAL: shift logical left/shift arithmetic left

SHR: shift logical right

SAR: shift arithmetic right

ROL: rotate left

ROR: rotate right

RCL: rotate left through carry

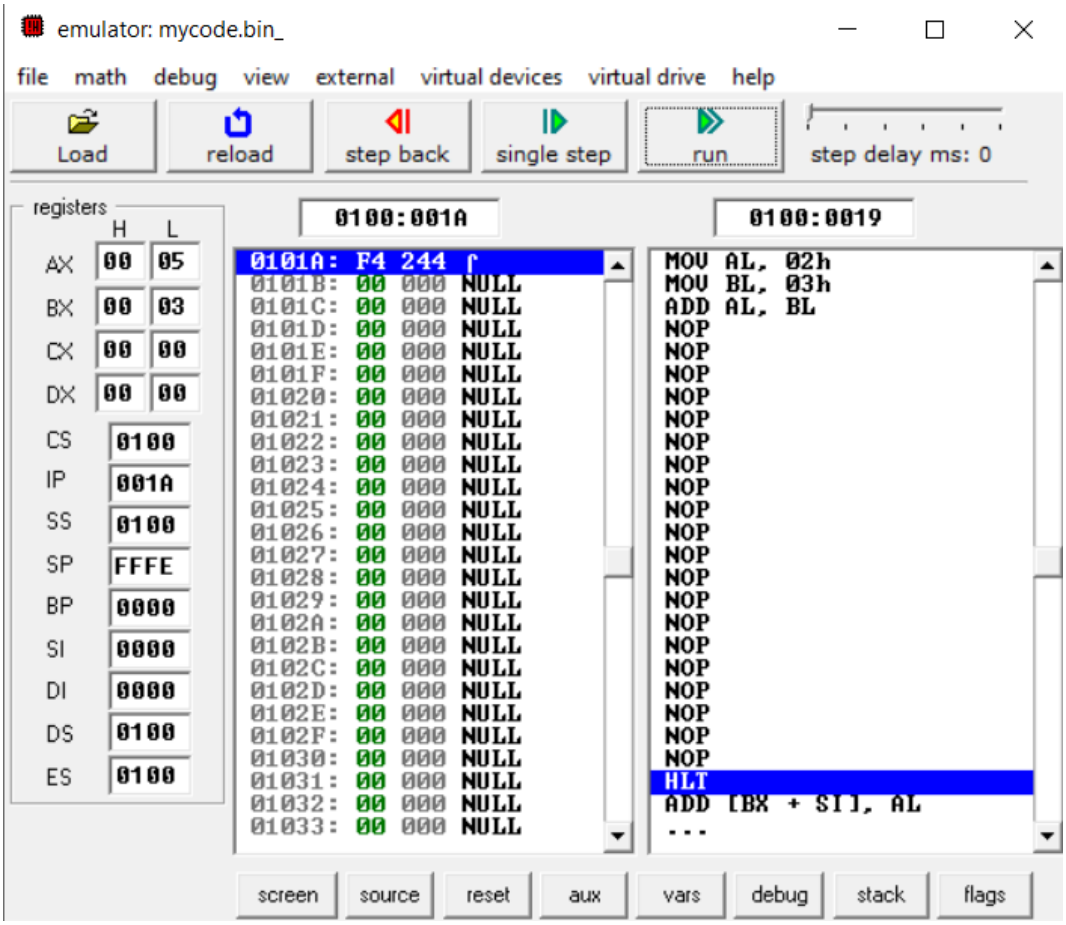
RCR: rotate right through carry

EXPERIMENT - 3

AIM: Write a program for the addition of 8/16/32-bit number.

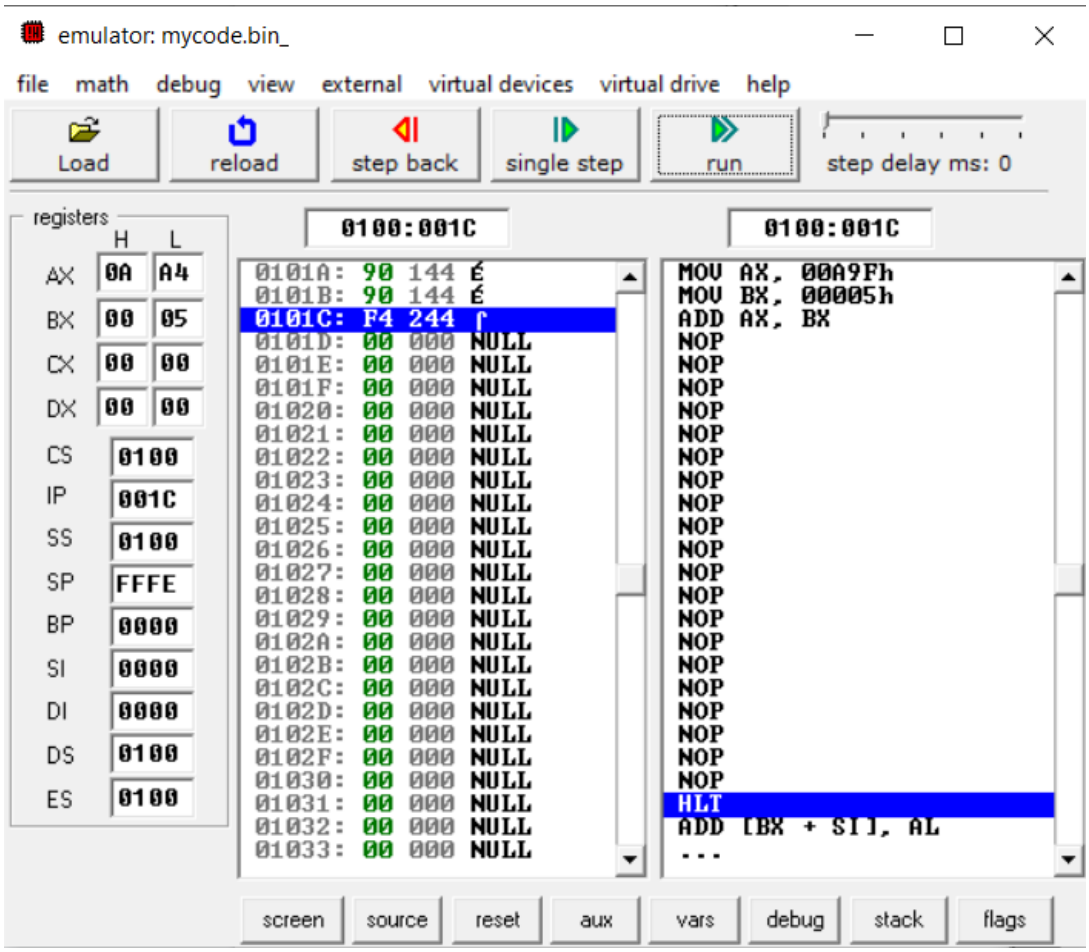
CODE:

1. **For 8 bit number:**
   MOV AL, 02
   MOV BL, 03
   ADD AL,BL



Output - The result will be stored in the AL register i.e.05

2. **For 16 bit number:**
   MOV AX, 0A9FH
   MOV BX, 0005H
   ADD AX,BX

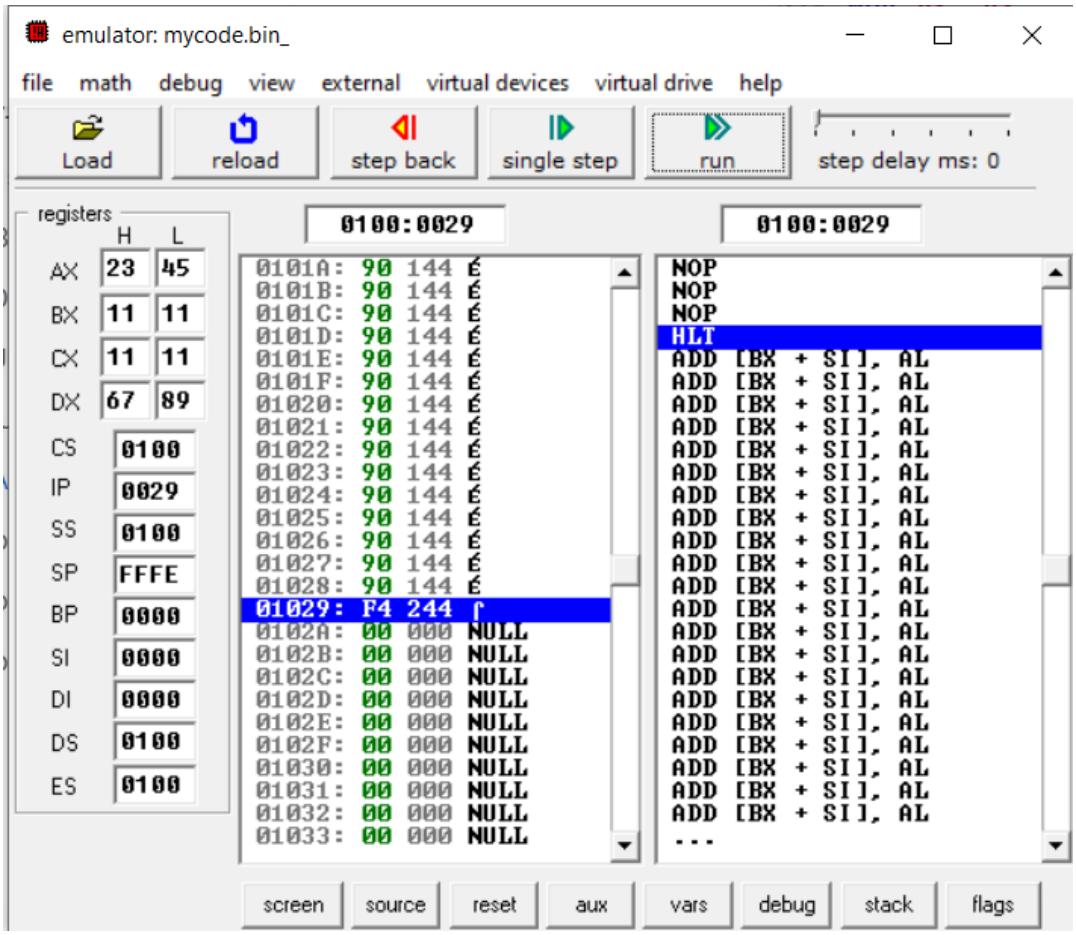


Output - The result will be stored in the AX register i.e. 0AA4

3. **For 32 bit number:**
   First number- 12345678H Second Number- 11111111H

   MOV AX, 5678H ; Move no 1 LSB in AX register
   MOV BX, 1111H ; Move no 2 LSB in BX register
   ADD AX, BX

```
MOV DX, AX
MOV AX, 0000H        ; clearing register
MOV AX, 1234H        ; Move no 1 MSB in AX register
MOV CX, 1111h        ; Move no 2 MSB in CX register
ADC AX,CX            ; final result will be stored in AX and DX
```

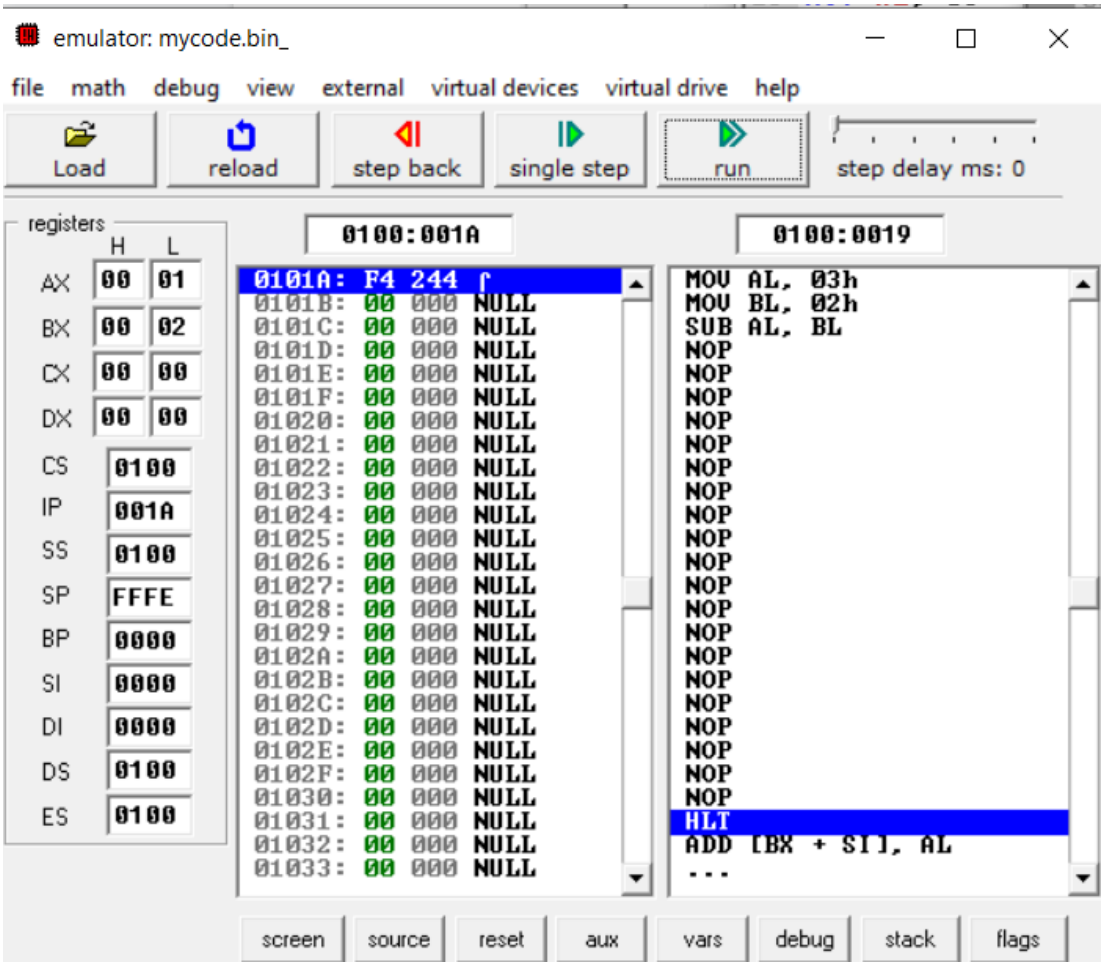


Output - The LSB and MSB of the result will be stored in the DX and AX register respectively i.e. 23456789H

AIM: Write a program for the subtraction of 8/16/32-bit number.

CODE:

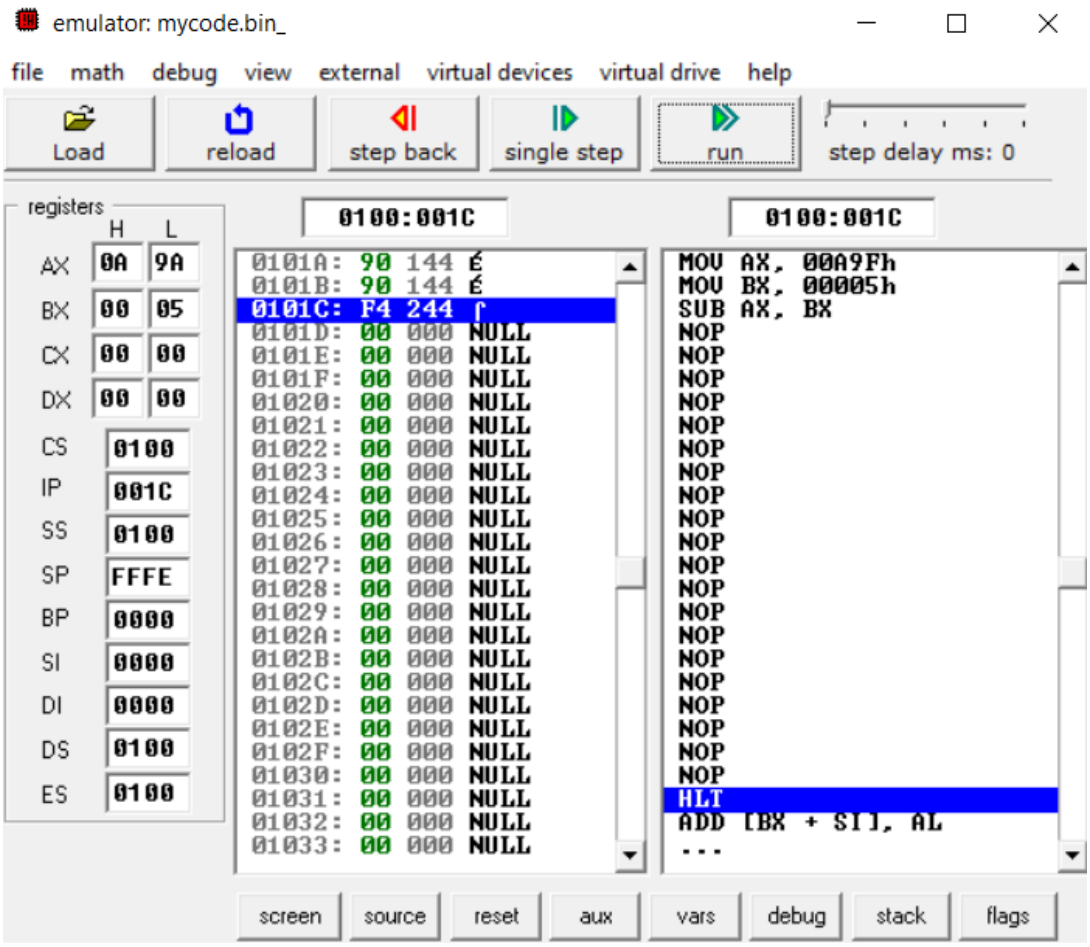
1. **For 8 bit number -**
   MOV AL, 03
   MOV BL, 02
   SUB AL, BL



Output - The result will be stored in the AL register i.e 01.

2. **For 16 bit number -**

MOV AX, 0A9FH
MOV BX, 0005
SUB AX,BX

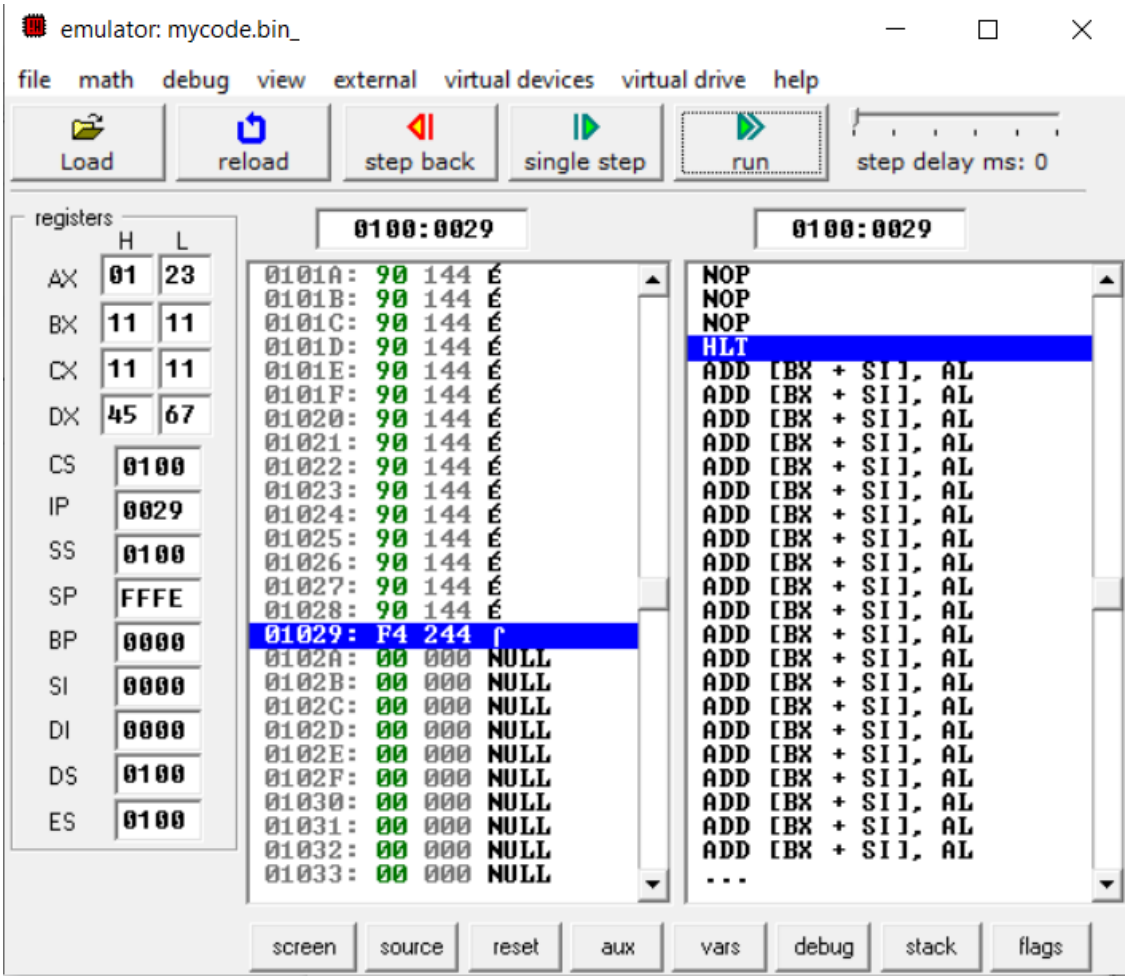


Output - The result will be stored in the AX register i.e 0A9A.

3. **For 32 bit number -**

MOV AX, 5678H ; Move no 1 LSB in AX register
MOV BX, 1111H ; Move no 2 LSB in BX register
SUB AX, BX
MOV DX, AX
MOV AX, 0000H ; clearing register
MOV AX, 1234H ; Move no 1 MSB in AX register
MOV CX, 1111h ; Move no 2 MSB in CX register

SBB AX,CX ; final result will be stored in AX and DX



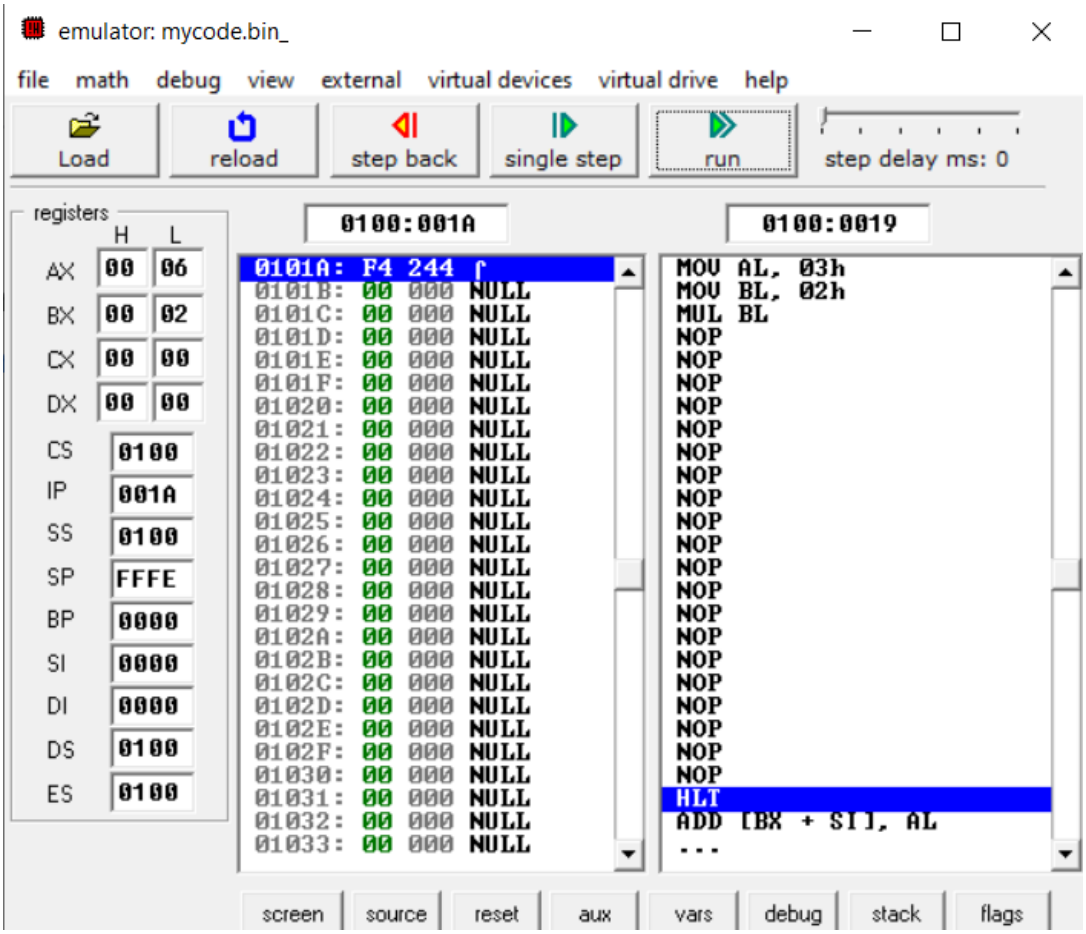
Output - The LSB and MSB of the result will be stored in the DX and AX register respectively i.e 01234567H.

AIM: Write a program for the multiplication of 8/16-bit number.

CODE:

1. For 8 bit number -

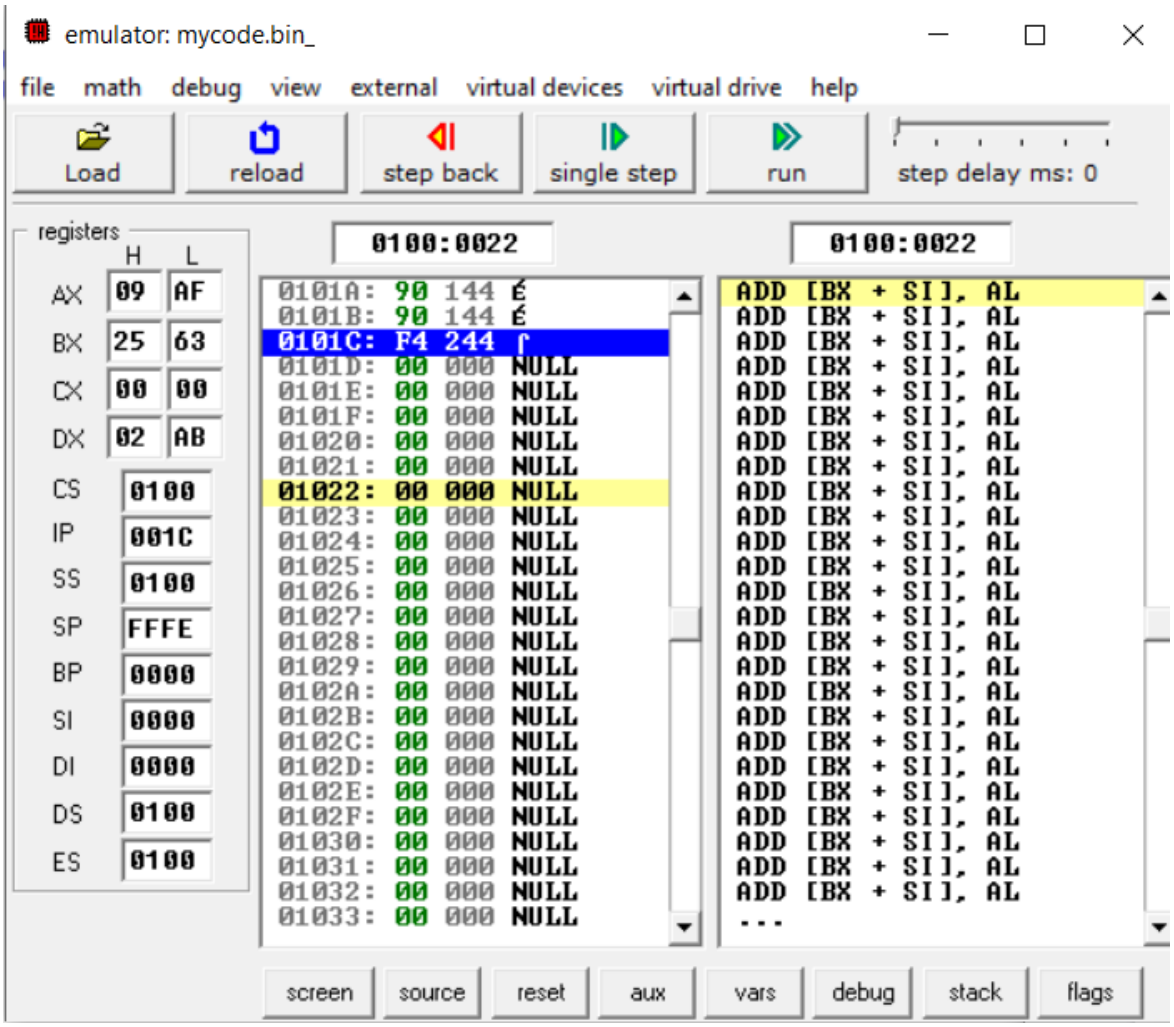
```
MOV AL, 03
MOV BL, 02
MUL BL
```



Output - The result will be stored in the AL register i.e 06.

2. **For 16 bit number -**

MOV AX, 1245h
MOV BX, 2563h
MUL BX

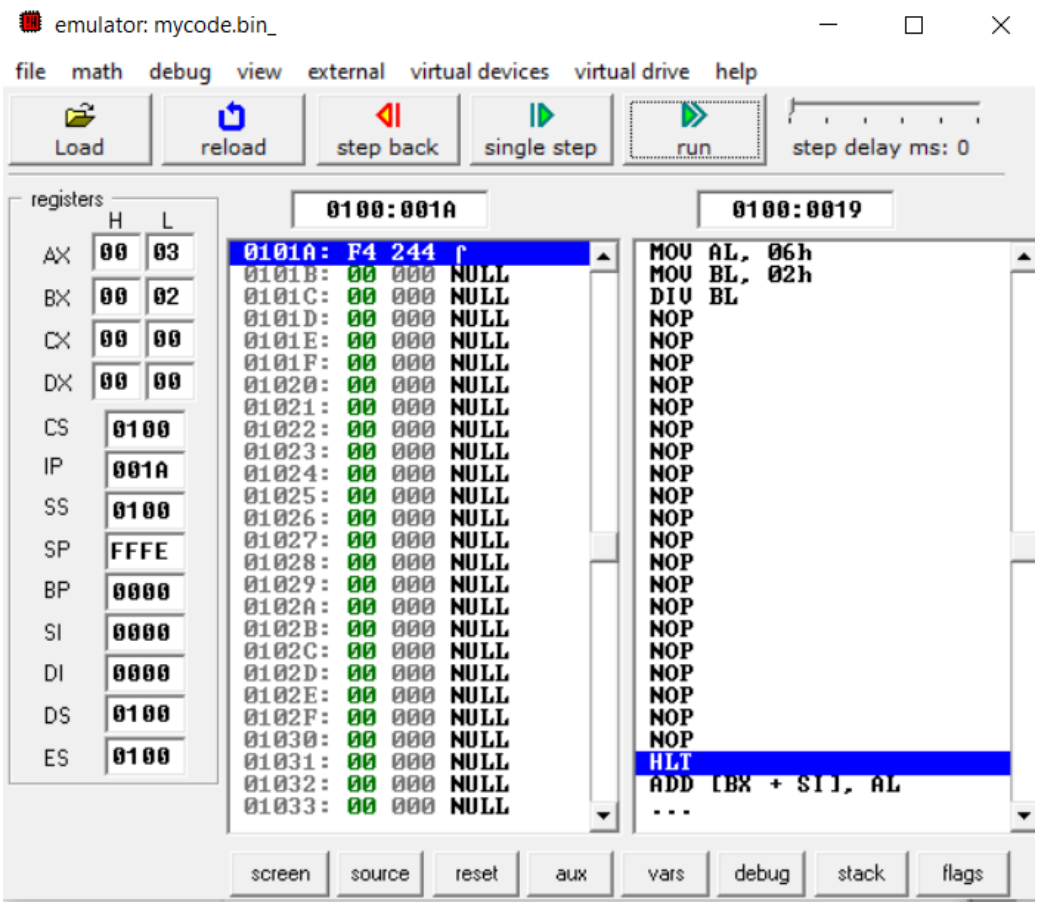


Output - The result will be stored in the DX and AX register i.e 02AB09AF.

AIM: Write a program for the division of 8/16-bit number.

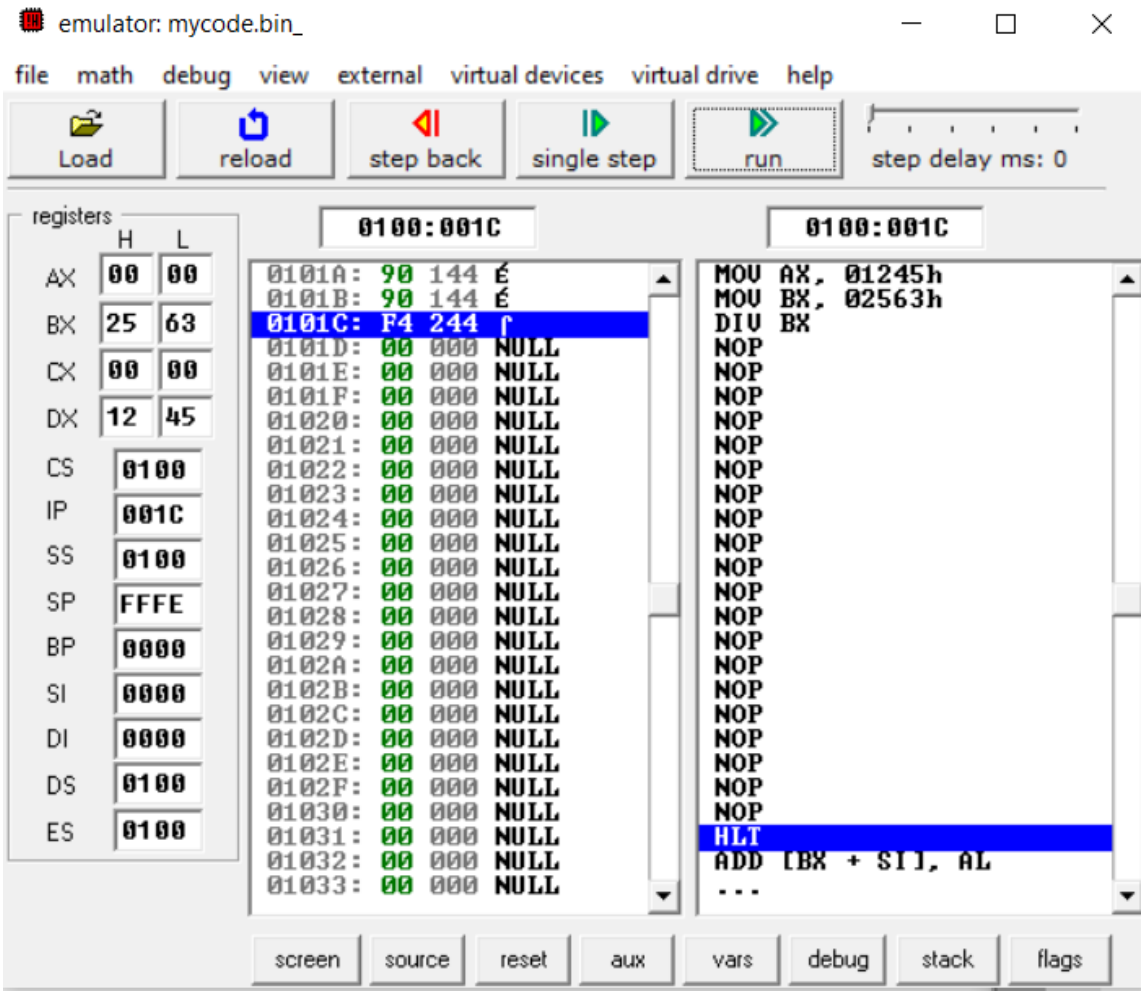
CODE:

1. **For 8 bit number -**
   MOV AL, 06
   MOV BL, 02
   DIV BL



Output - The quotient will be stored in the AL register i.e 03 and the remainder in AH register i.e 0.

2. **For 16 bit number -**
   MOV AX, 1245h
   MOV BX, 2563h
   DIV BX



Output - The quotient will be stored in the AX register i.e 00 and the remainder in DX register i.e 1245.

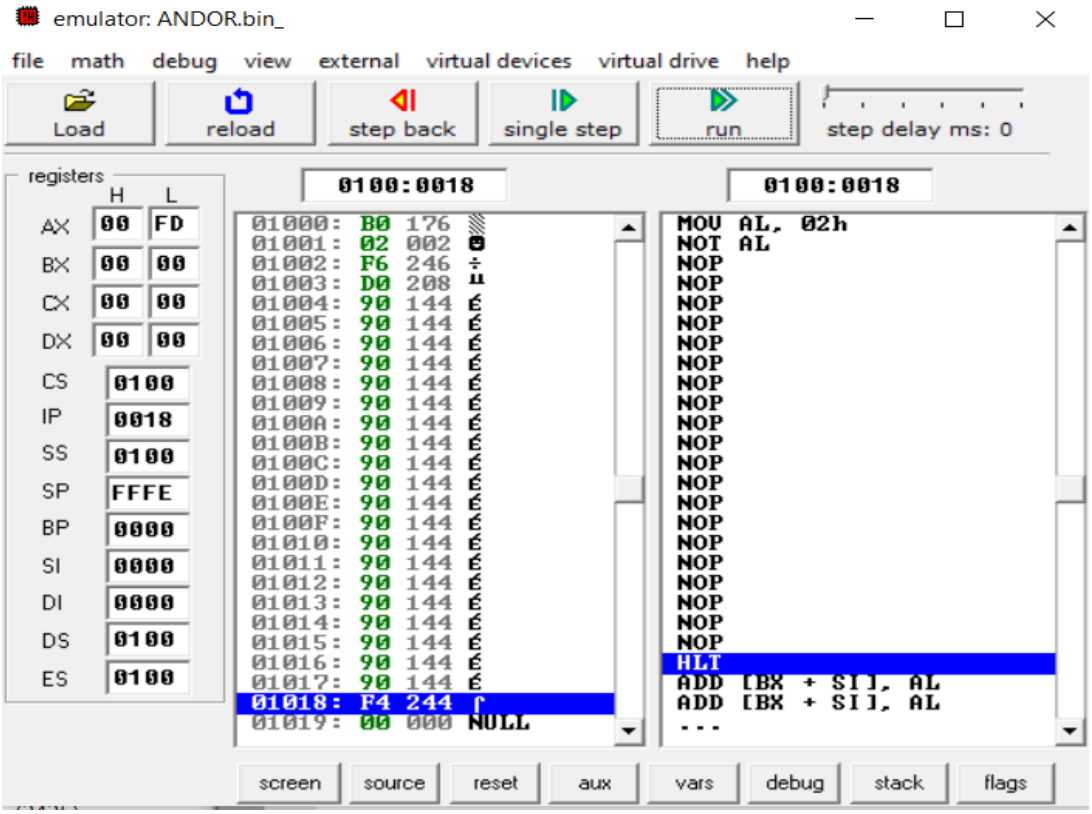
EXPERIMENT - 4

AIM: Write a program for logical operation (XOR, OR, AND, NOT) and comparison of 8/16 -bit number.

CODE:

- **NOT**

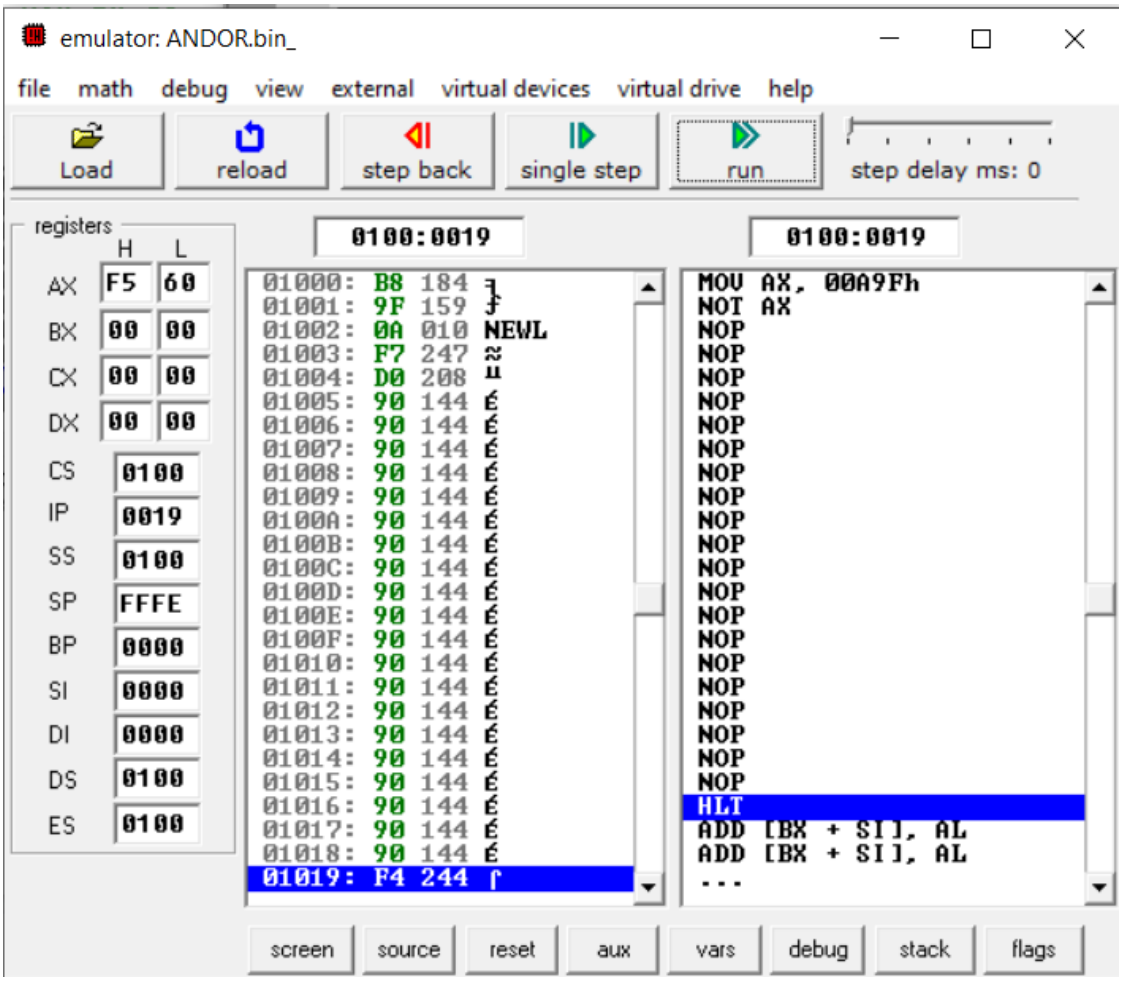
1. **For 8 bit number -**
   MOV AL,02
   NOT AL



Output - Output is stored in AL register i.e FD

2. For 16 bit number -

MOV AX,0A9FH
NOT AX

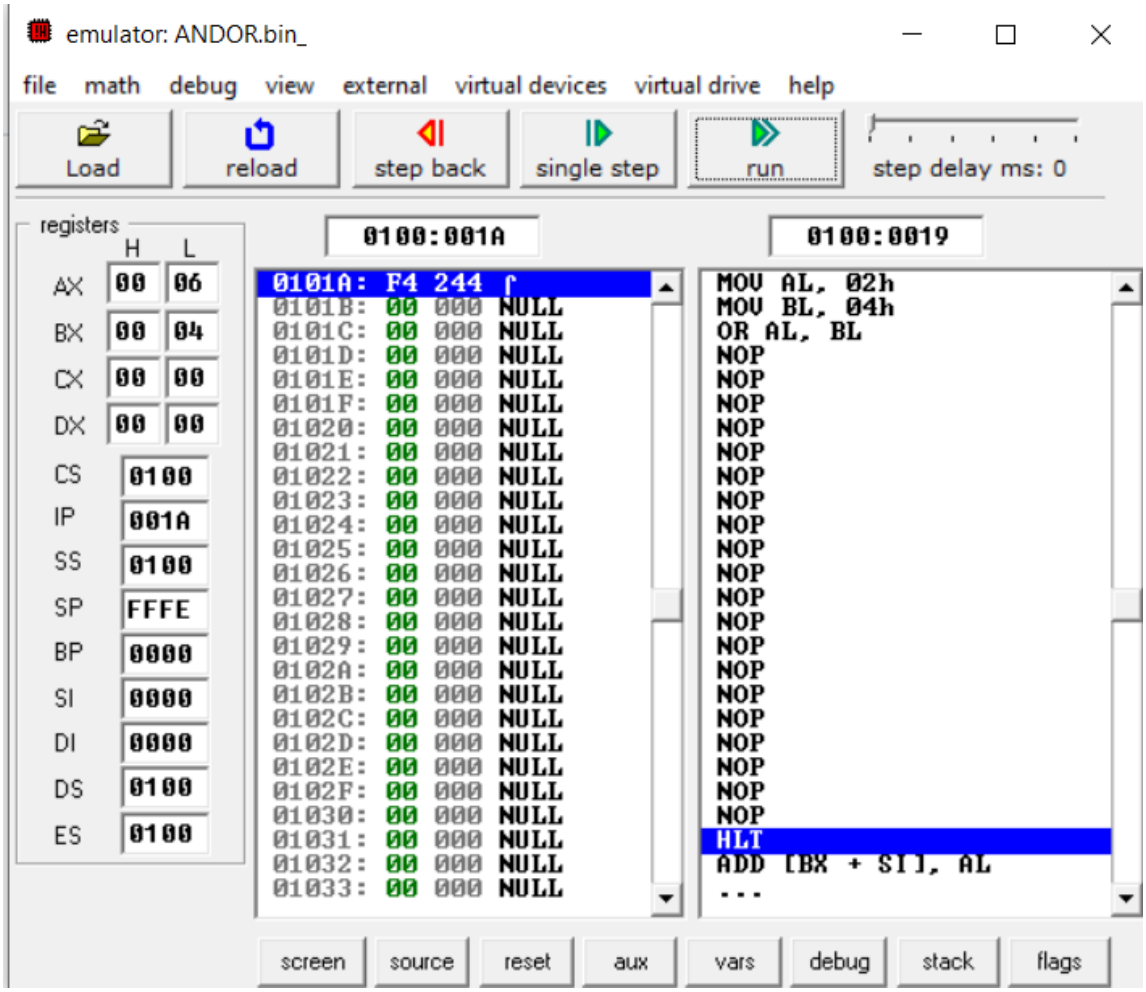


Output - Output is stored in AX register i.e F560

- **<u>OR</u>**

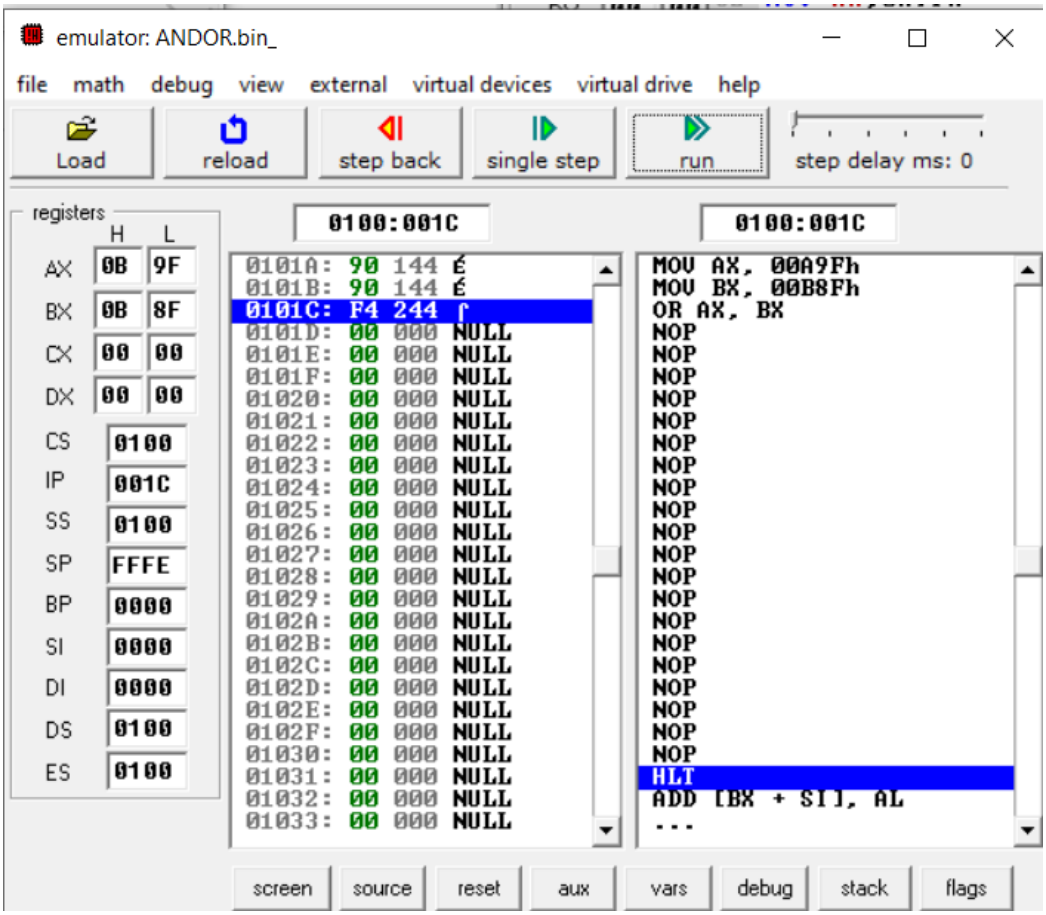
1. **For 8 bit number -**

MOV AL,02
MOV BL,04
OR AL,BL



Output - Output is stored in AL register i.e 06

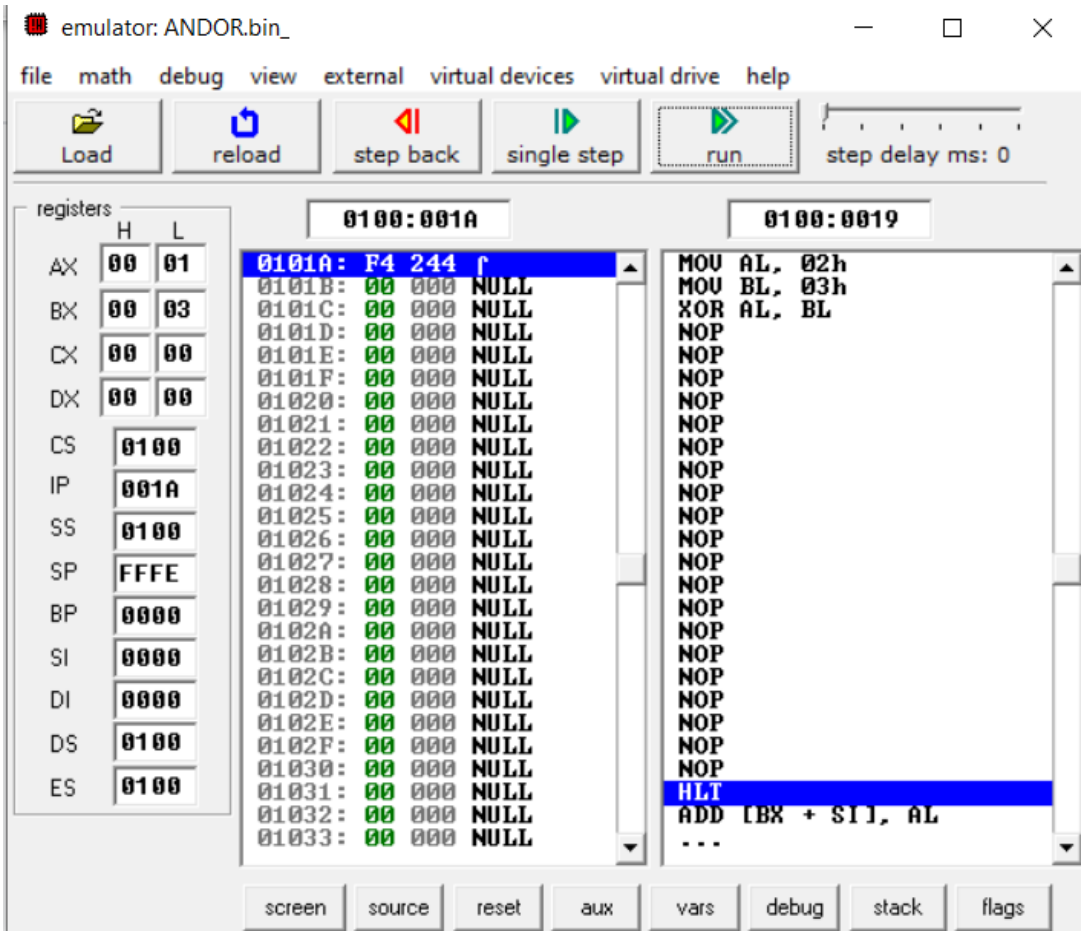
2. **For 16 bit number-**
   MOV AX,0A9FH
   MOV BX,0B8FH
   OR AX,BX



Output - Output is stored in AX register i.e 0B9F h

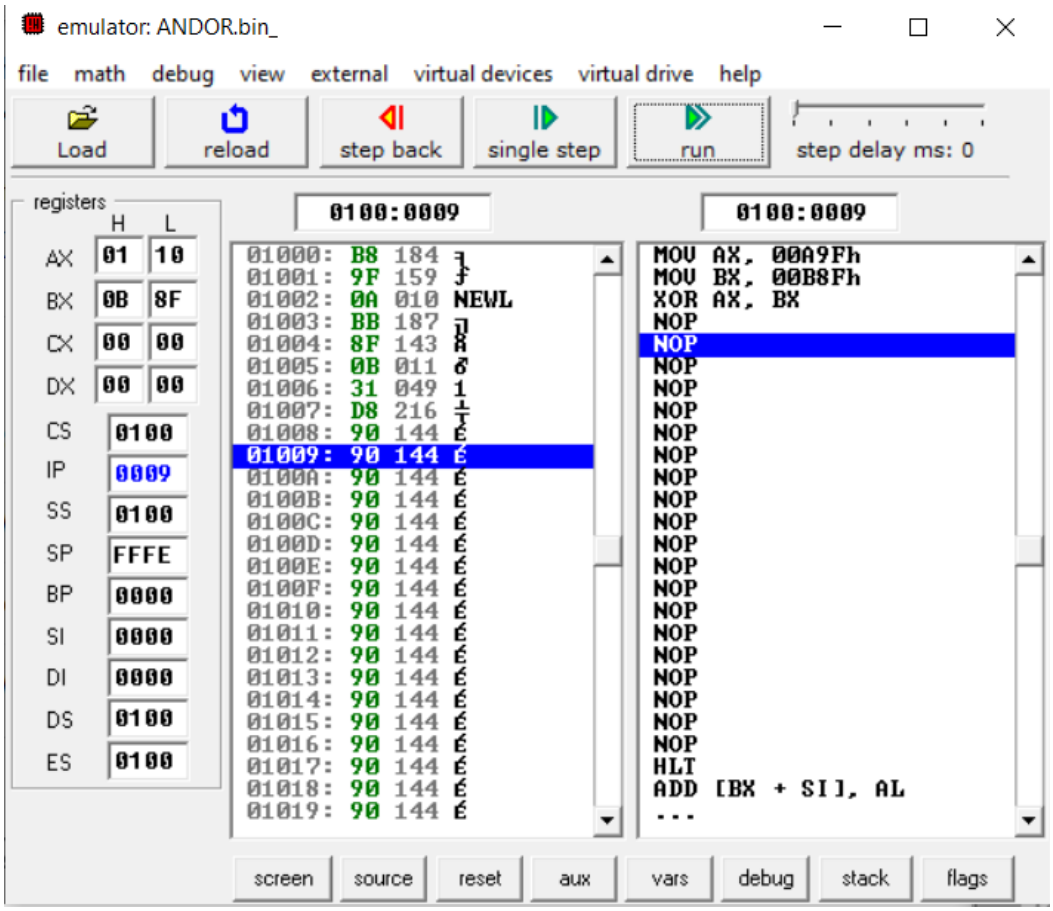
- **<u>XOR</u>**

1. **For 8 bit number-**
   MOV AL,02
   MOV BL,03
   XOR AL,BL



Output - Output is stored in AL register i.e 01

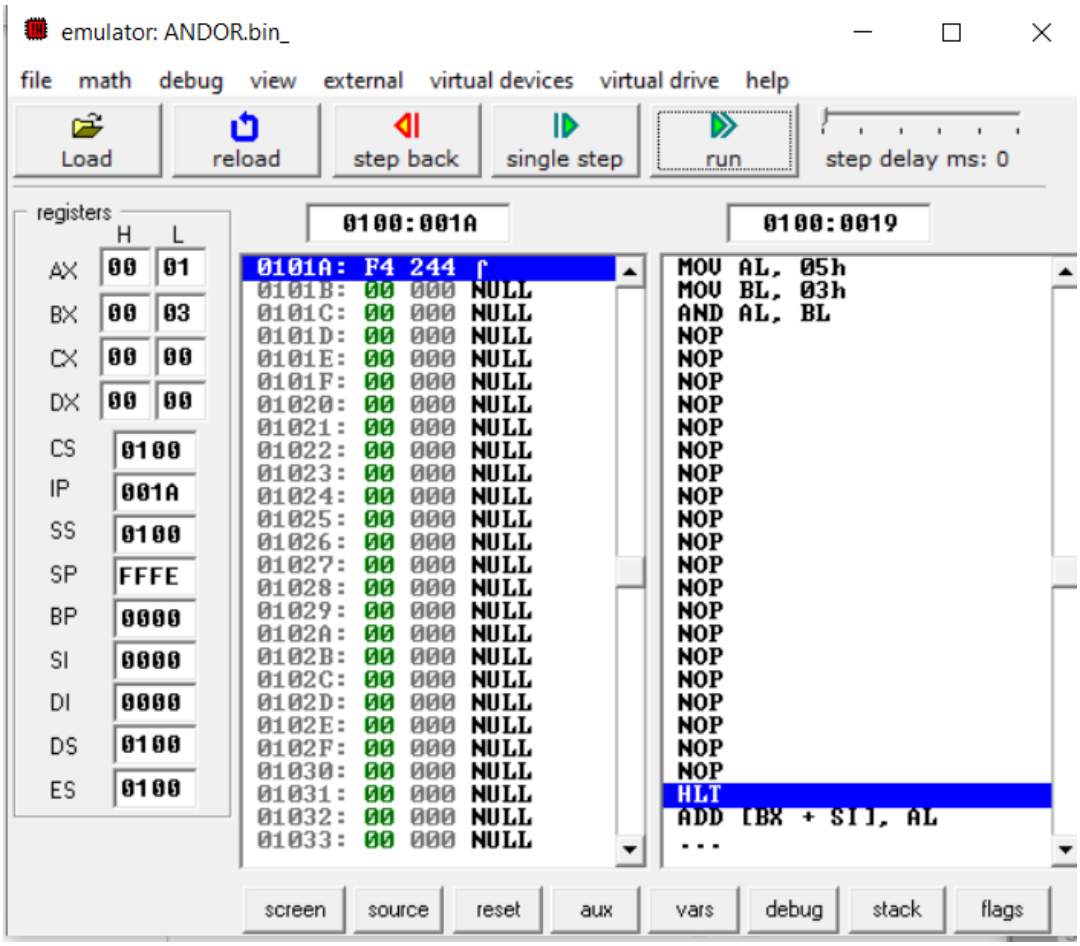
2. **For 16 bit number-**
   MOV AX,0A9FH
   MOV BX,0B8FH
   XOR AX,BX



Output - Output is stored in AX register i.e 0110 h

- **AND**

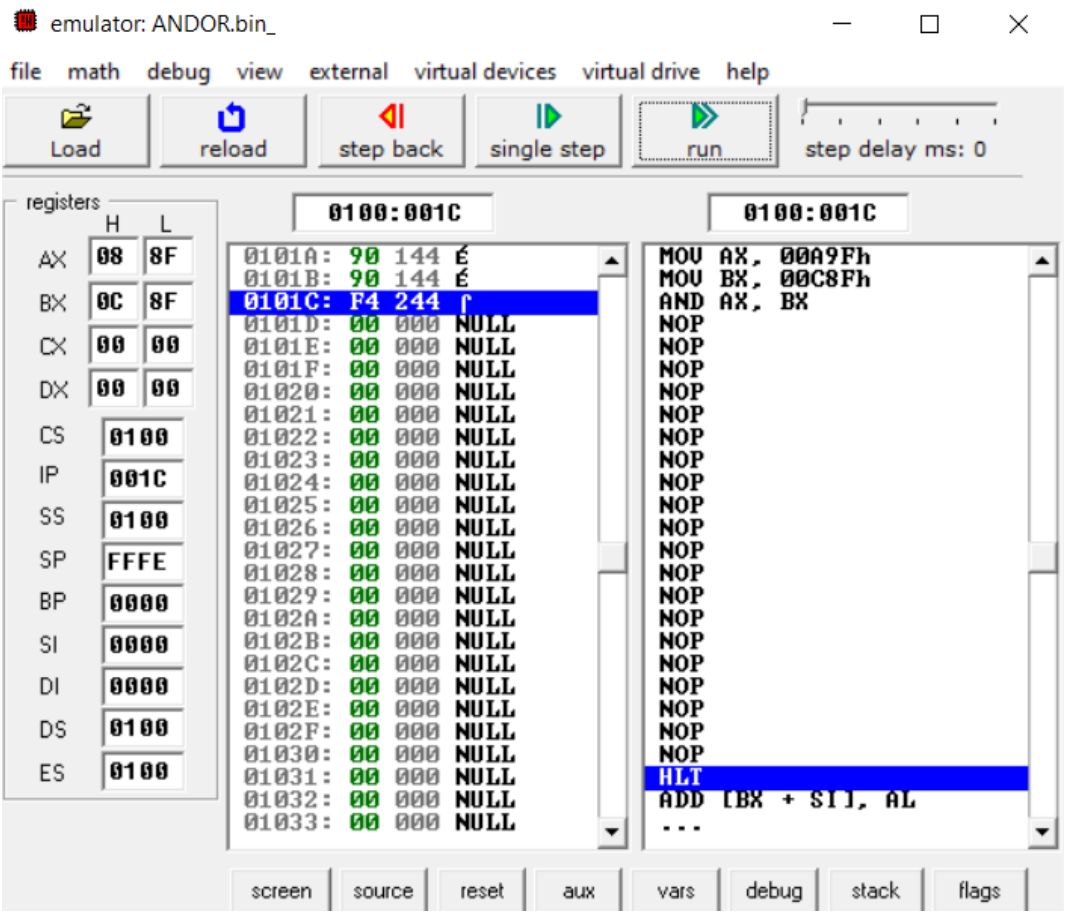
1. **For 8 bit number-**
   MOV AL,05
   MOV BL,03
   AND AL,BL



Output - Output is stored in AL register i.e 01

2. **For 16 bit number-**

   MOV AX,0A9FH
   MOV BX,0C8FH
   AND AX,BX



Output - Output is stored in AX register i.e 088F h

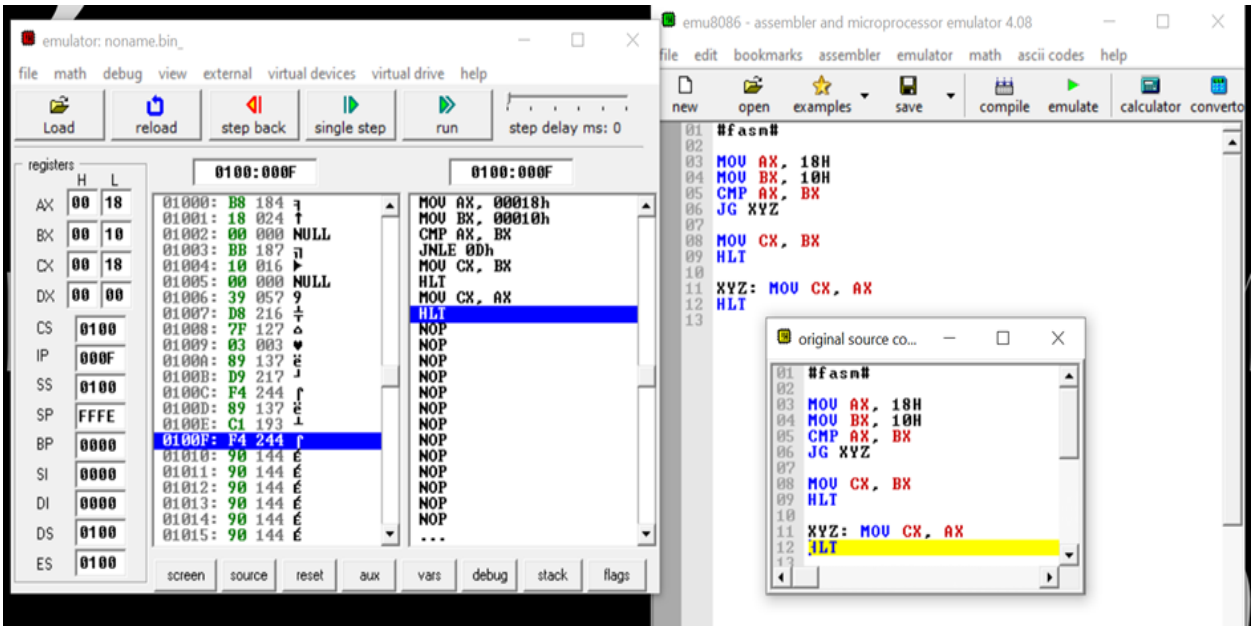
- **CMP**

1. For 8 bit number-

MOV AX, 18H
MOV BX, 10H
CMP AX, BX
JG XYZ

MOV CX, BX
HLT

XYZ: MOV CX, AX
HLT



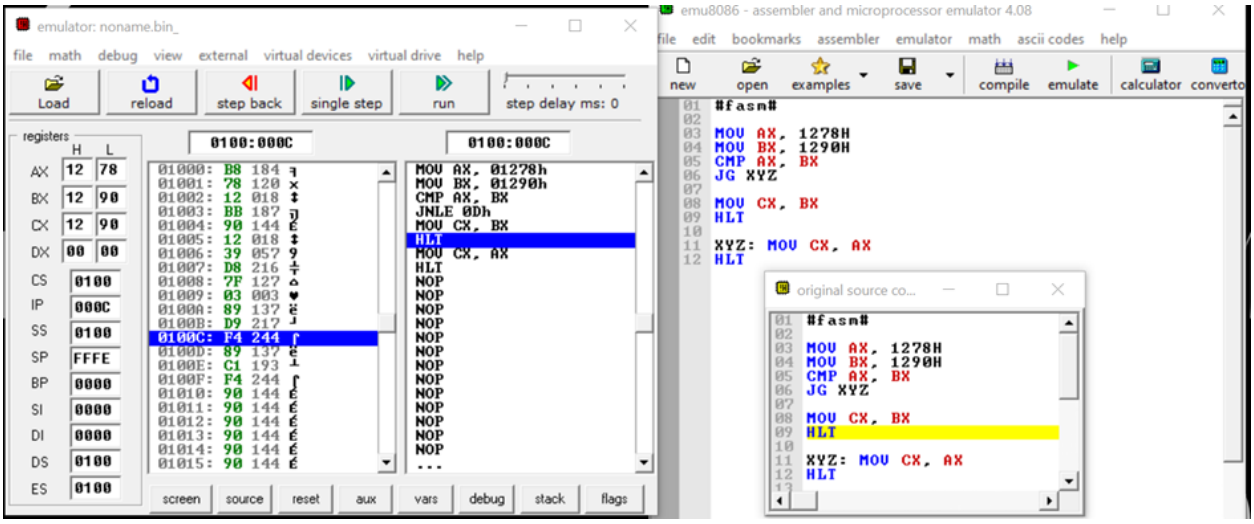
Output: The greater number is stored in register CX, i.e.,18.

2. For 16 bit number-

MOV AX, 1278H
MOV BX, 1290H
CMP AX, BX
JG XYZ

MOV CX, BX
HLT

XYZ: MOV CX, AX
HLT



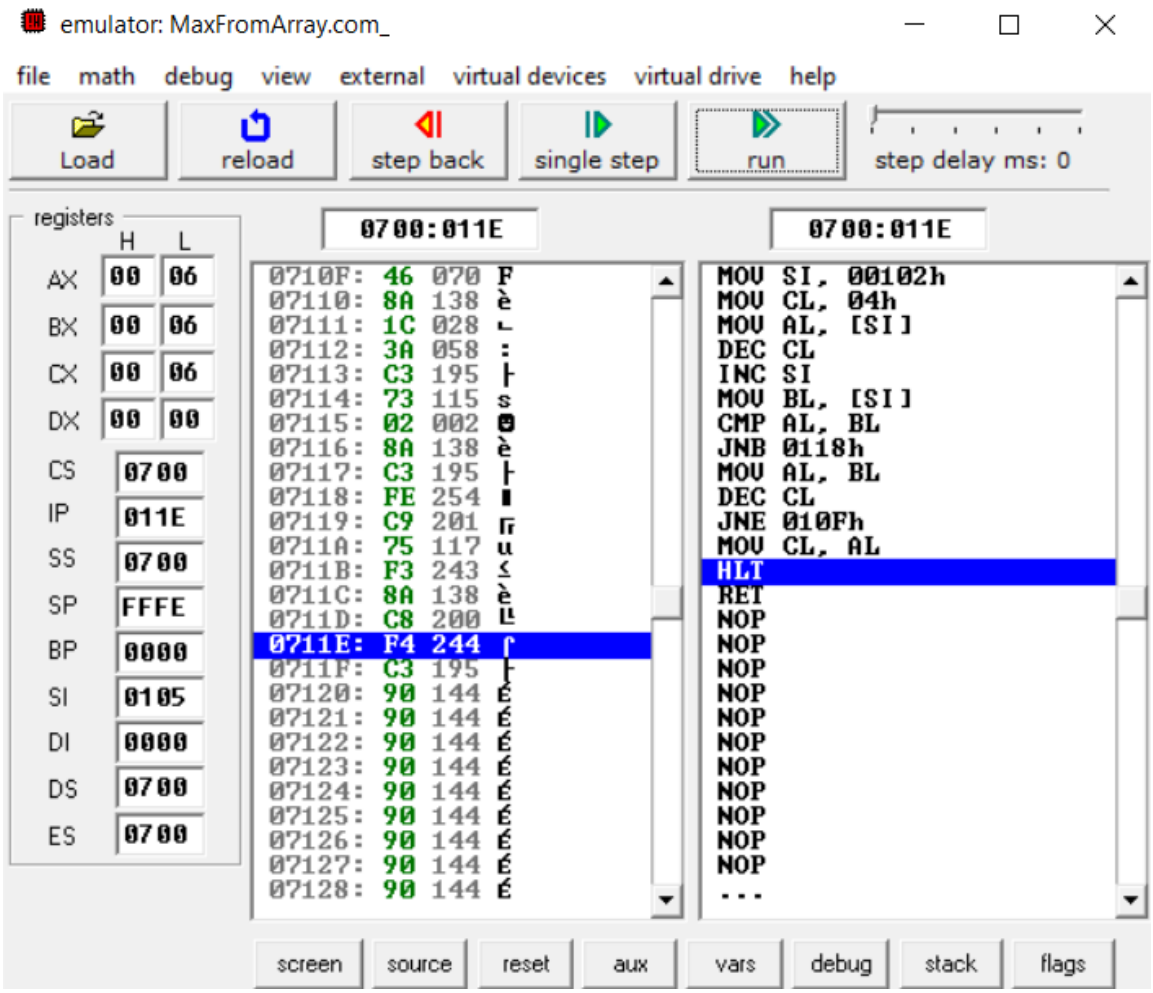
Output: The greater number is stored in register CX, i.e.,1290.

EXPERIMENT - 5

AIM: Write a program to find the maximum of N given numbers.

CODE:

```
org 100h
jmp start
vec1 db 1,2,5,6  ;[1 2 5 6]

start:
   lea si,vec1
   mov cl,04      ;Keeping count of numbers in array
   mov al,[si]
   dec cl

again:
   inc si
   mov bl,[si]
   cmp al,bl
   jnc ahead
   mov al,bl

ahead:
   dec cl
   jnz again       ; If the elements still remain then continue the process
   mov cl, al      ; move the result i.e maximum no to cl register
   hlt

ret
```



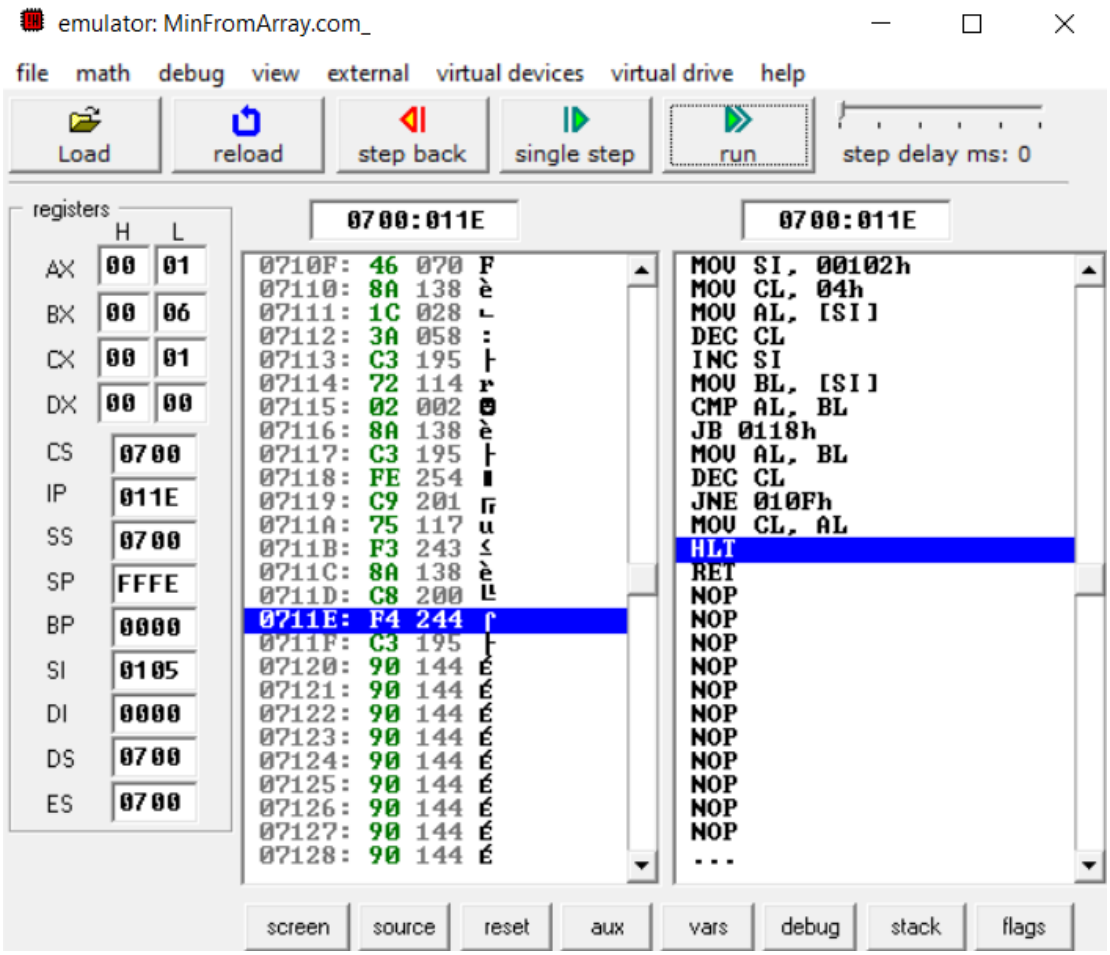
Output - The maximum number will be stored in the CL register.
So, the answer will be 06.

EXPERIMENT - 6

AIM: Write a program to find the minimum of N given numbers.

CODE:

```
org 100h
jmp start
vec1 db 1,2,5,6  ;[1 2 5 6]

start:
   lea si,vec1
   mov cl,04
   mov al,[si]
   dec cl

again:
   inc si
   mov bl,[si]
   cmp al,bl
   jc ahead
   mov al,bl

ahead:
   dec cl
   jnz again
   mov cl,al
   hlt

ret
```



Output - The minimum number will be stored in the CL register.
So, the answer will be 01.

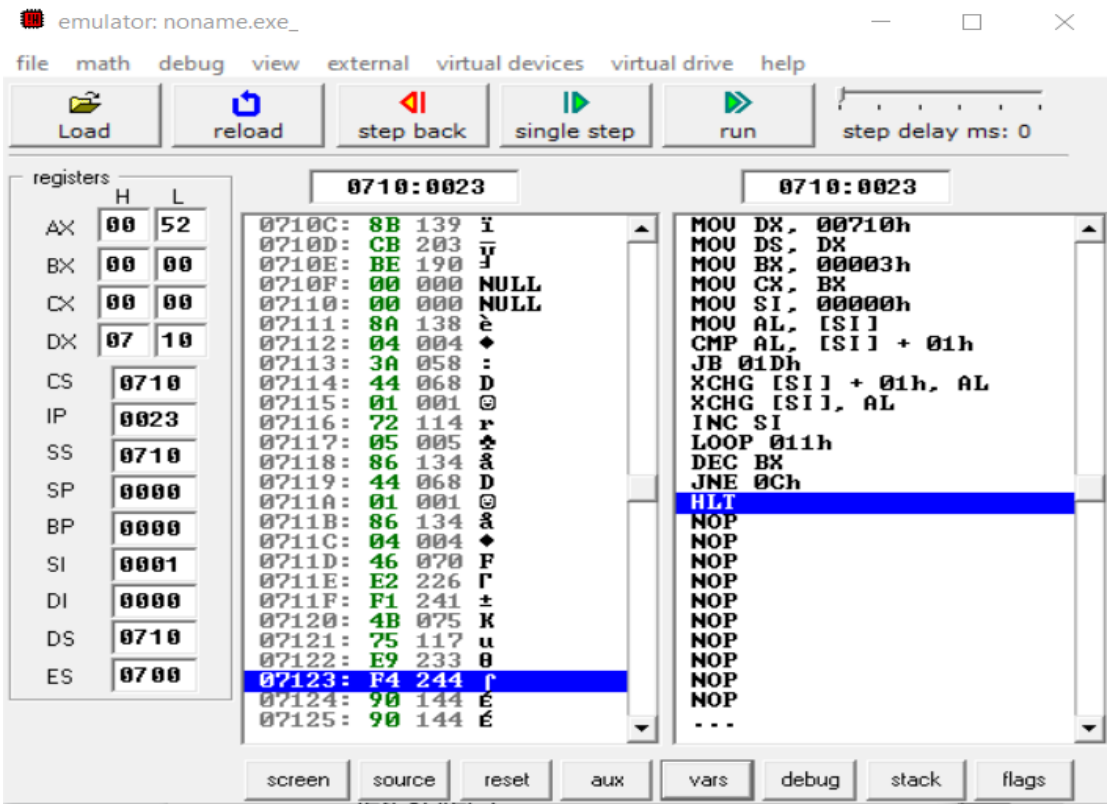
EXPERIMENT - 7

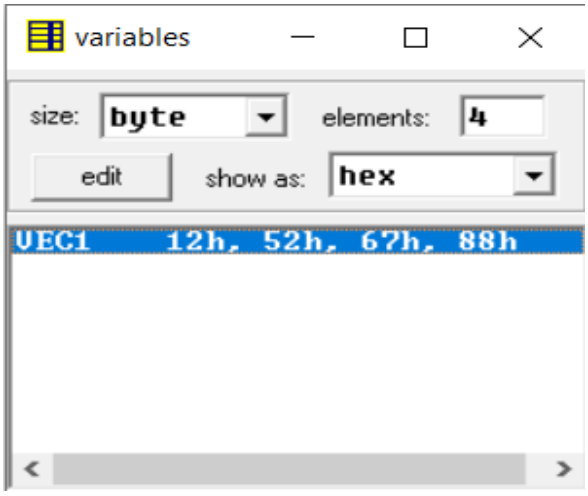
AIM: Write a program to arrange a given number in ascending order.

CODE:

```
data segment
vec1 db 67h, 52h, 88h, 12h    ;67,88,52,12
ends

start:
mov dx, data
mov ds, dx
mov bx,3

l1:
mov cx,bx
lea si,vec1

l2:
mov al,[si]
cmp al, [si+1]
jc xx;
xchg al,[si+1]
xchg al,[si]

xx:
inc si
loop l2
dec bx
jnz l1
```

```
hlt
ends
end start
```





Output - The vector vec1 is sorted in ascending order as shown in variables window.

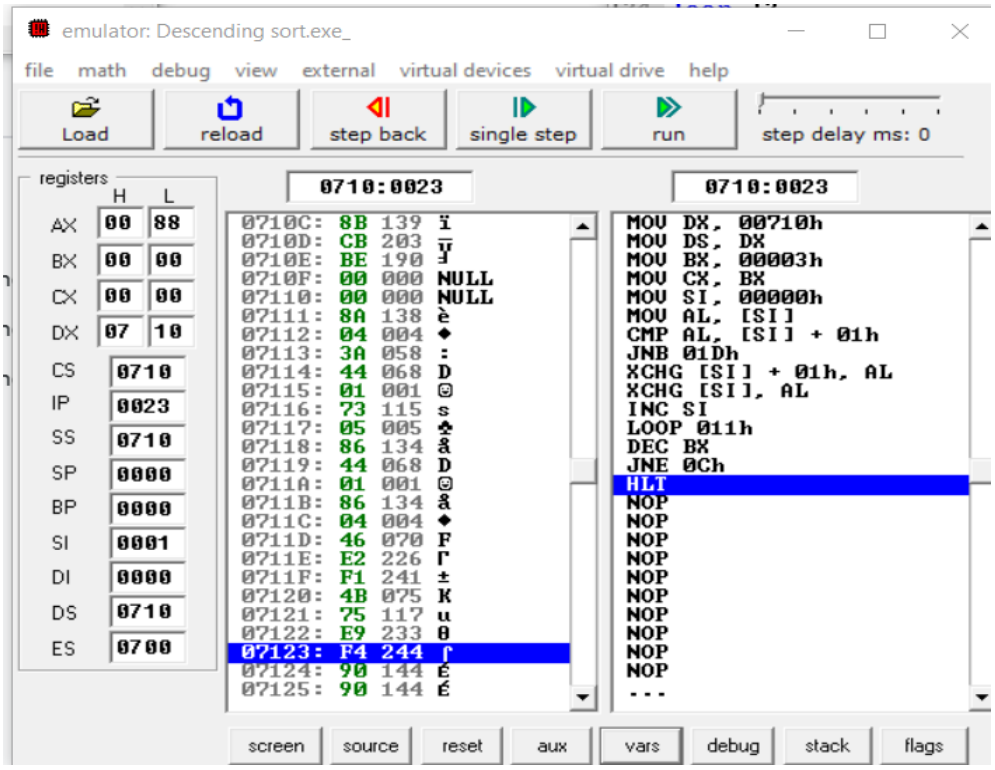
EXPERIMENT - 8

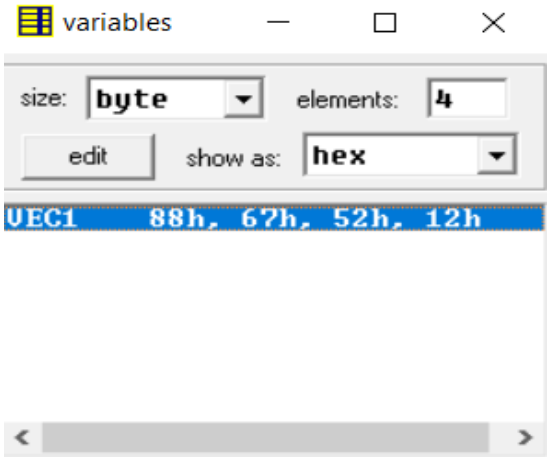
AIM: Write a program to arrange a given number in descending order.

CODE:

```
data segment
vec1 db 67h, 52h, 88h, 12h    ;67,88,52,12
ends

start:
mov dx, data
mov ds, dx
mov bx,3

l1:
mov cx,bx
lea si,vec1

l2:
mov al,[si]
cmp al, [si+1]
jnc xx;
xchg al,[si+1]
xchg al,[si]

xx:
inc si
loop l2
dec bx
jnz l1
```

hlt
ends
end start





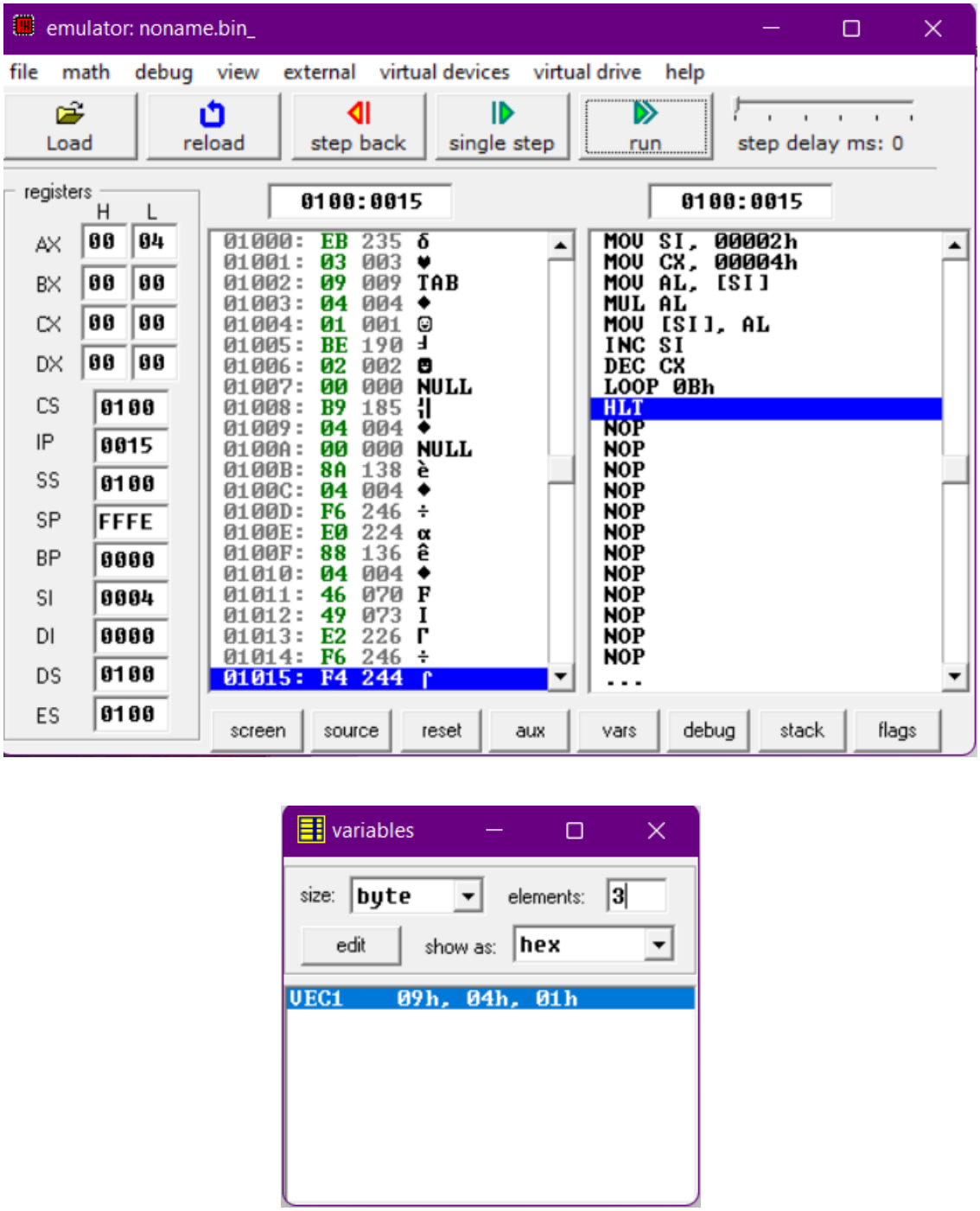
Output - The vector vec1 is sorted in descending order as shown in variables window.

EXPERIMENT - 9

AIM: Program to do square of the given series.

CODE:

```
JMP START

VEC1 DB 3,2,1

START:

LEA SI,VEC1
MOV CX,4

L1: MOV AL,[SI]
   MUL AL
   MOV [SI],AL
   INC SI
   DEC CX
   LOOP L1
   HLT
```



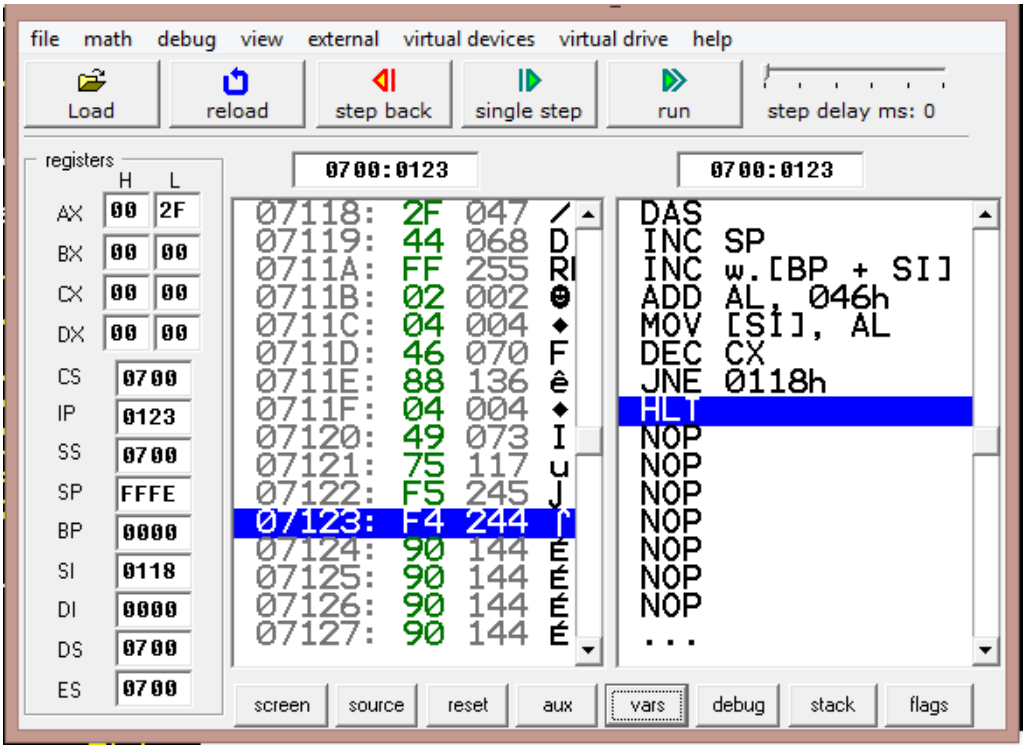
Output – All the elements of the series are squared and stored in vec1.

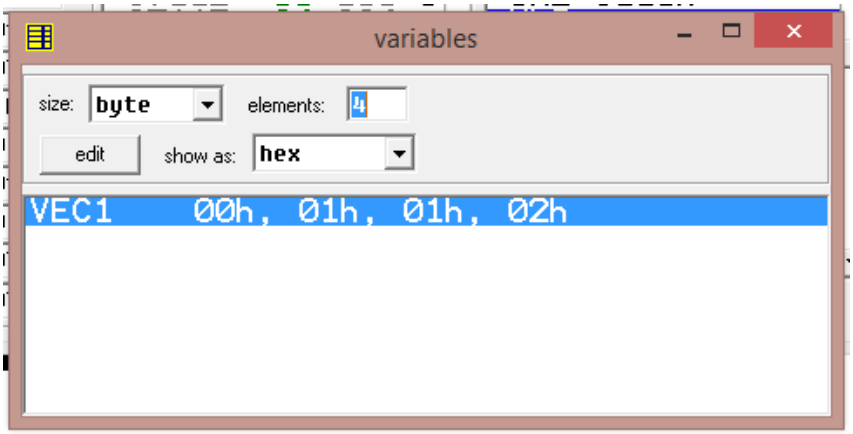
EXPERIMENT 10

AIM: Program to generate the Fibonacci series.

CODE:

```
ORG 100H
JMP START
VEC1 DB ?,?,?,?
START:
LEA SI,VEC1
MOV CX,5
MOV CX,15H
MOV AL,00H
MOV [SI],AL
INC SI
INC AL
MOV [SI],AL
L1: MOV AL,[SI-1]
   ADD AL,[SI]
   INC SI
   MOV [SI],AL
   DEC CX
   JNZ L1
   HLT
```





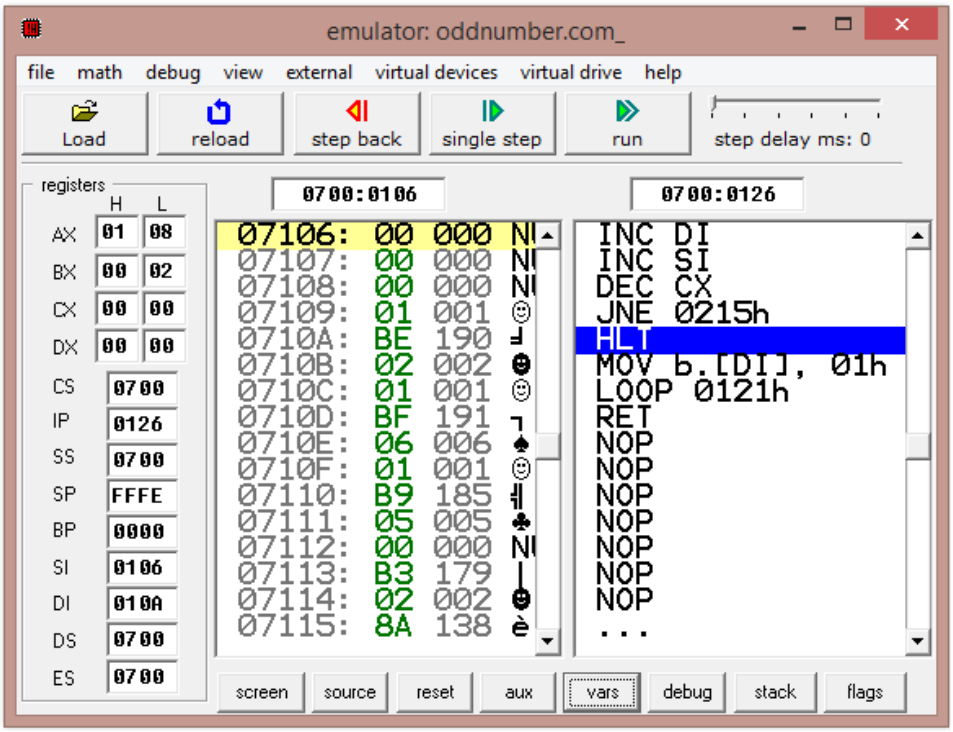
Output- From the variables window we can see the generated Fibonacci series.

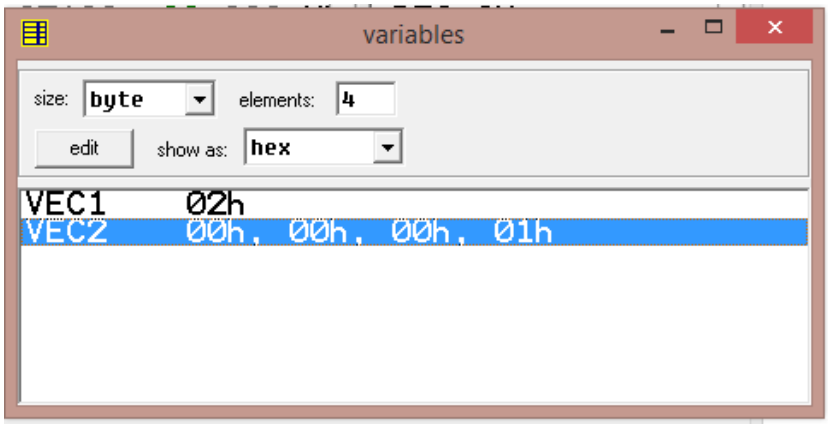
EXPERIMENT 11

AIM: Program to find out EVEN and ODD numbers in a series.

CODE:

```
;EVEN AND ODD
ORG 100H
JMP START
VEC1 DB 2,4,6,17
VEC2 DB ?,?,?,?
START:
LEA SI, VEC1  ;
LEA DI, VEC2   ;
MOV CX,5  ;
MOV BL,02H
L2: MOV AL,[SI]
   DIV BL
   CMP AH,00H
   JNZ L1
   MOV [DI],0
L3: INC DI
   INC SI
   DEC CX
   JNZ L2
   HLT
L1: MOV [DI],1
   LOOP L3
RET
```





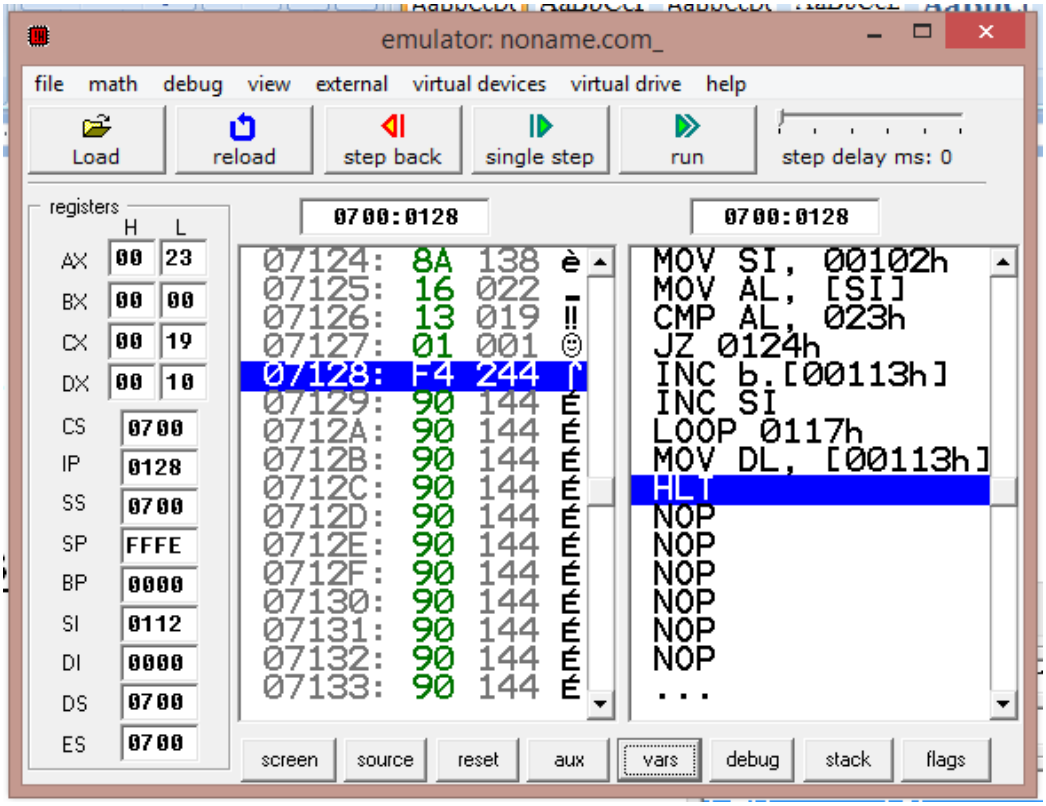
Output- From the variables window we can see that the 0 shows even, 1 shows odd numbers in vec2.

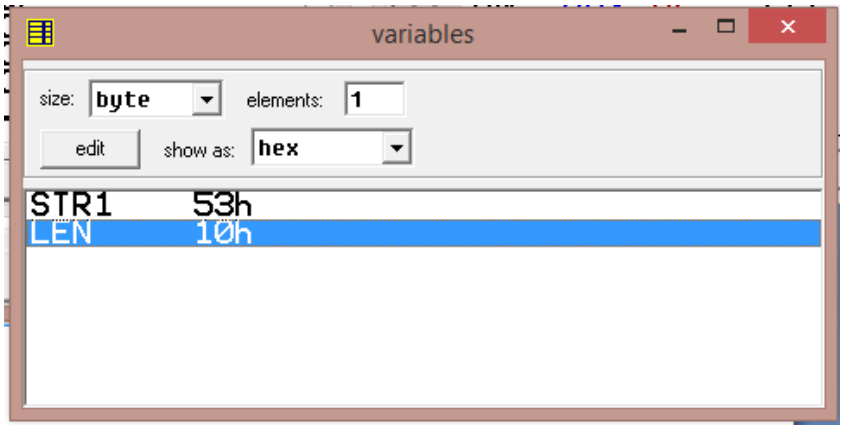
EXPERIMENT 12

AIM: Program to count the length of a string.

CODE:

```
ORG 100H
JMP START
 STR1 DB "Simran Sachdeva #"
 LEN DB 0
START:
 MOV SI , OFFSET STR1
 UP:  MOV AL , [SI]
     CMP AL , '#'
     JZ DN         ; IF DESTINATION == SOURCE THEN ZF = 1
     INC LEN
     INC SI
     LOOP UP
DN: MOV DL , LEN
HLT
```





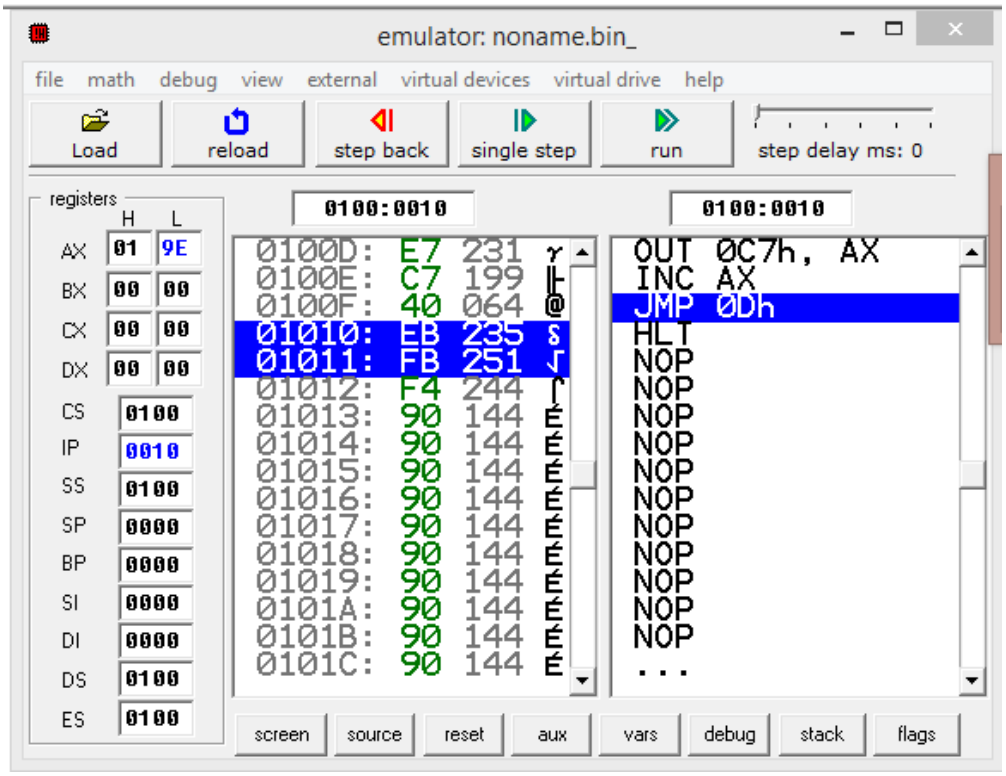
Output- From the variables window we can see the string length.

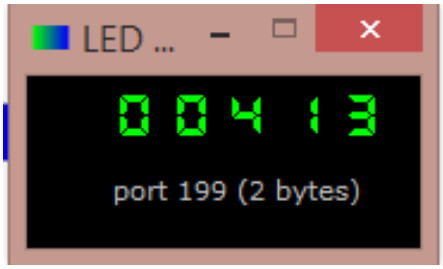
EXPERIMENT 13

AIM: Program to display data on LED

CODE:

```
#start=led_display.exe#
#make_bin#
MOV AX, 1234
OUT 199, AX
MOV AX, -5678
OUT 199, AX
; eternal loop to write
; values to port:
MOV AX, 0
X1:
  OUT 199, AX
  INC AX
JMP X1
HLT
```





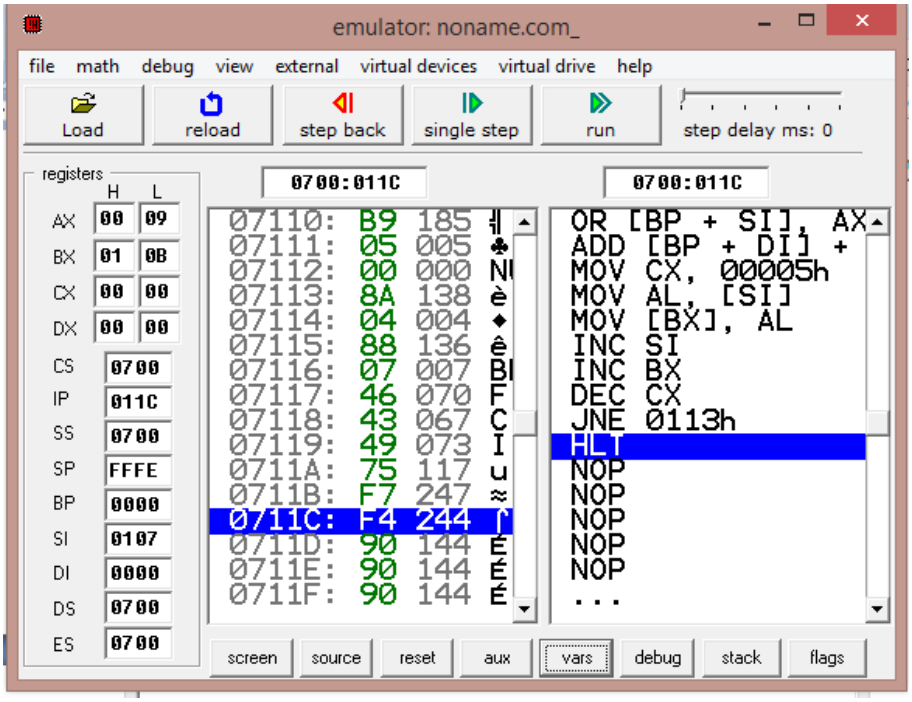
Output: The led display with data is shown below.

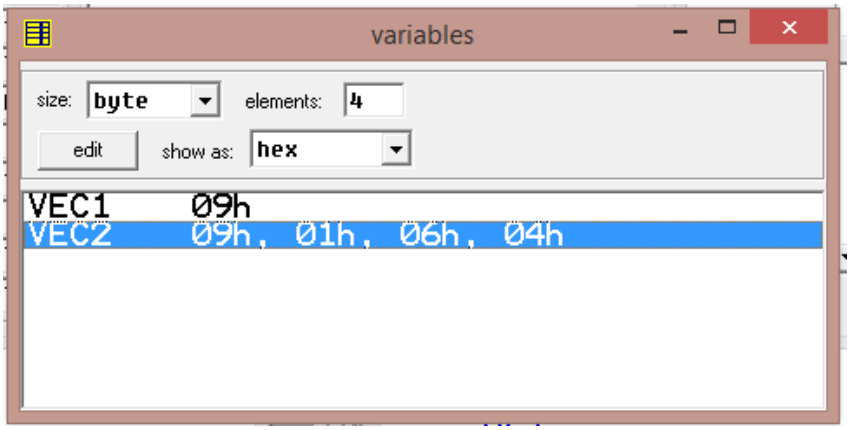
EXPERIMENT 14

AIM: Program to transfer the content of one memory location to another memory location.

CODE:

```
ORG 100H
JMP START
VEC1 DB 9,1,6,4
VEC2 DB ?,?,?,?
START:
LEA SI, VEC1
LEA BX, VEC2
MOV CX,5
L1: MOV AL,[SI] ;AL=3 ; 7;1
   MOV [BX],AL
   INC SI
   INC BX
   DEC CX
   JNZ L1
HLT
```





Output: From the variables window we can see that the data is moved from vec 1 to vec 2.