

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

Experiments	Page No.
1. Introduction of Emulator 8086	2-6
2. Instruction set: Programming and Illustration	7-26
3. Program for addition of 8/16/32-bit number. Program for subtraction of 8/16/32-bit number. Program for multiplication of 8/16-bit number. Program for division of 8/16-bit number.	27-29 30-32 33-34 35-36
4. Program for logical operation (XOR, OR, AND, NOT) and comparison of 8/16-bit number.	37-46
5. Program to find the maximum of N given numbers.	47-48
6. Program to find the minimum of N given numbers.	49-50
7. Program to arrange a given number in ascending order.	51-52
8. Program to arrange a given number in descending order.	53-54
9. Program to do square of the given series.	55-56
10. Program to generate the Fibonacci series.	57-58
11. Program to find out EVEN and ODD numbers in a series.	59-60
12. Program to count the length of a string.	61-62
13. Program to display data on LED.	63-64
14. Program to transfer content of memory location to another memory location.	65-66

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 and 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. Its offset address relative to stack segment.

6. BP – This is the base pointer. It is of 16 bits. It is primarily used in accessing parameters passed by the stack. Its 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	H	L
AX	00	00
BX	00	00
CX	00	00
DX	00	00
CS	0100	
IP	0000	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	

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

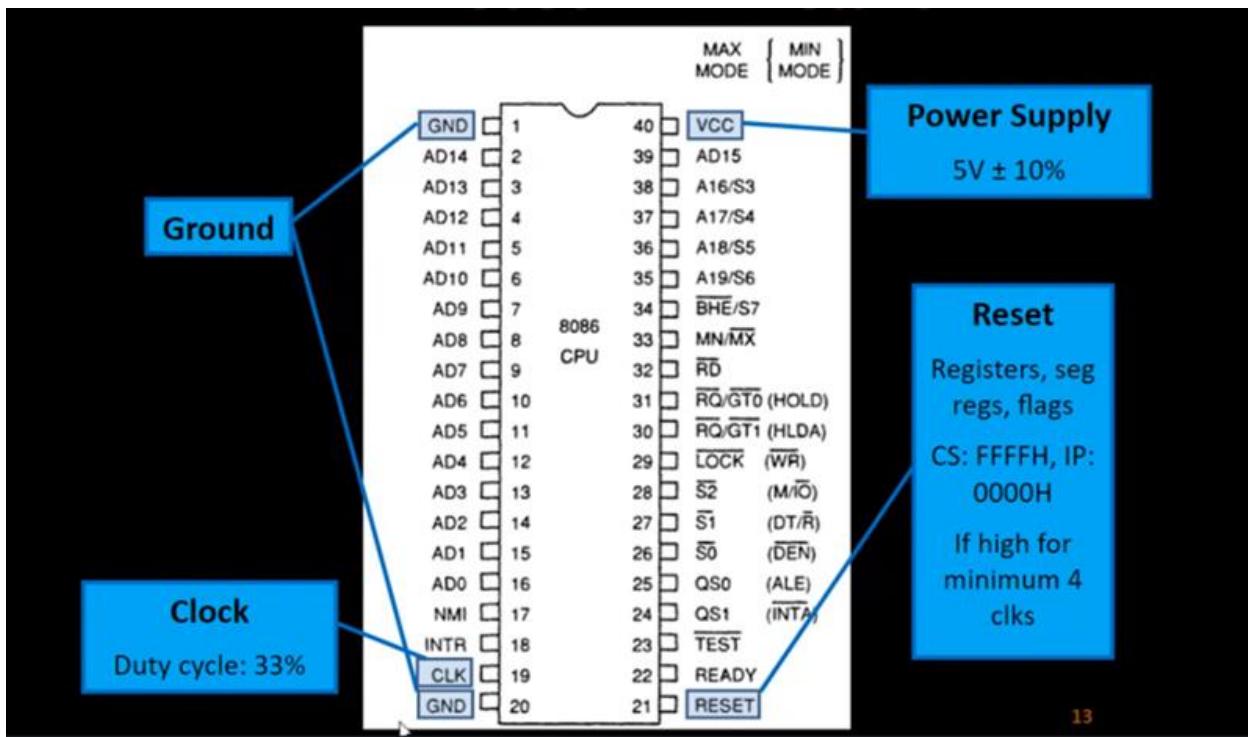
EXPERIMENT – 2

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.

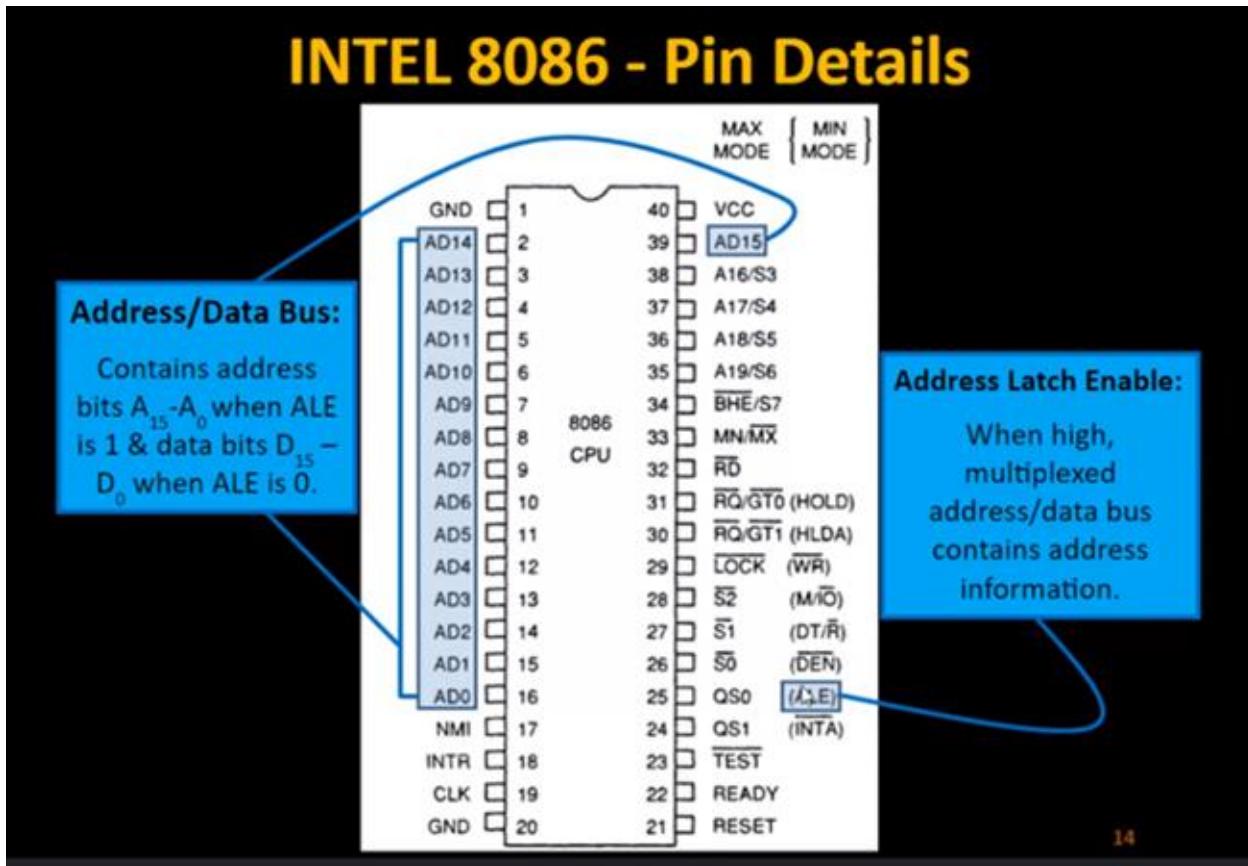
PIN DESCRIPTION



1. **VCC – 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. AD₀ – AD₁₅ – 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. A₁₆/S₃, A₁₇/S₄, A₁₈/S₅ and A₁₉S₆ – 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 S₃ and S₄ 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 S₃ and S₄:

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

- S₅, when enabled, shows the presence of an interrupts in the microprocessor. So, basically, it serves as an **interrupt flag**.
- S₆ 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' / S₇ – Pin number 34 – BHE is an acronym for Bus High Enable. The combination of the BHE signal and S₇ 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. S₀', S₁' and S₂' – ***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. QS₀ and QS₁ – 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'/ GT₀'and RQ'/ GT₁' – 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

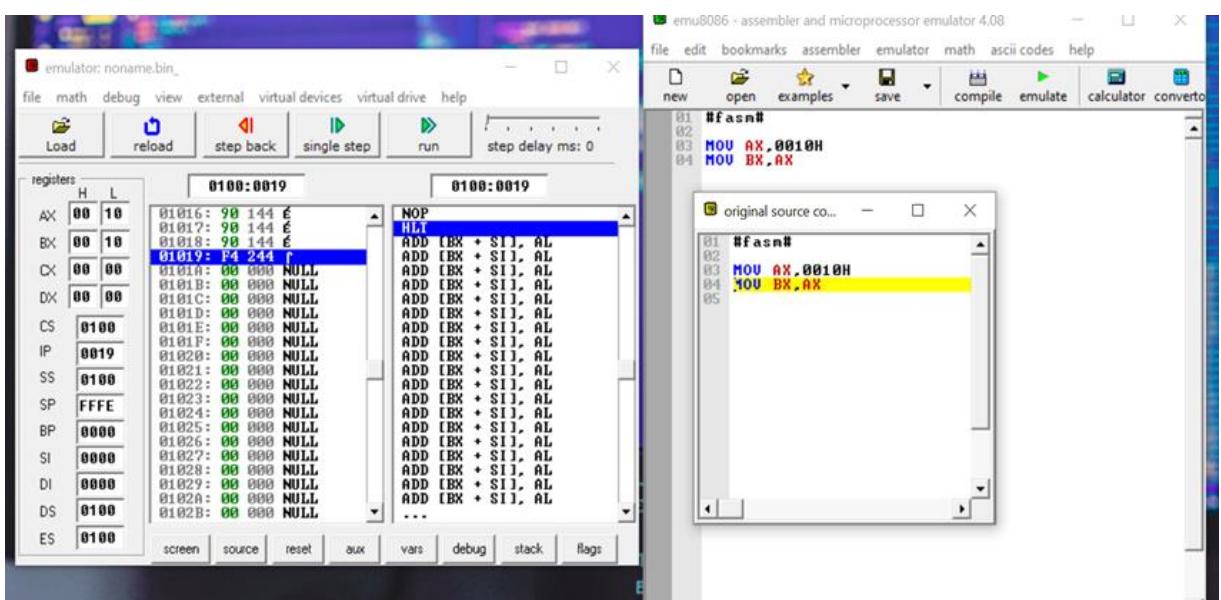
Data Transfer Instructions

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 –

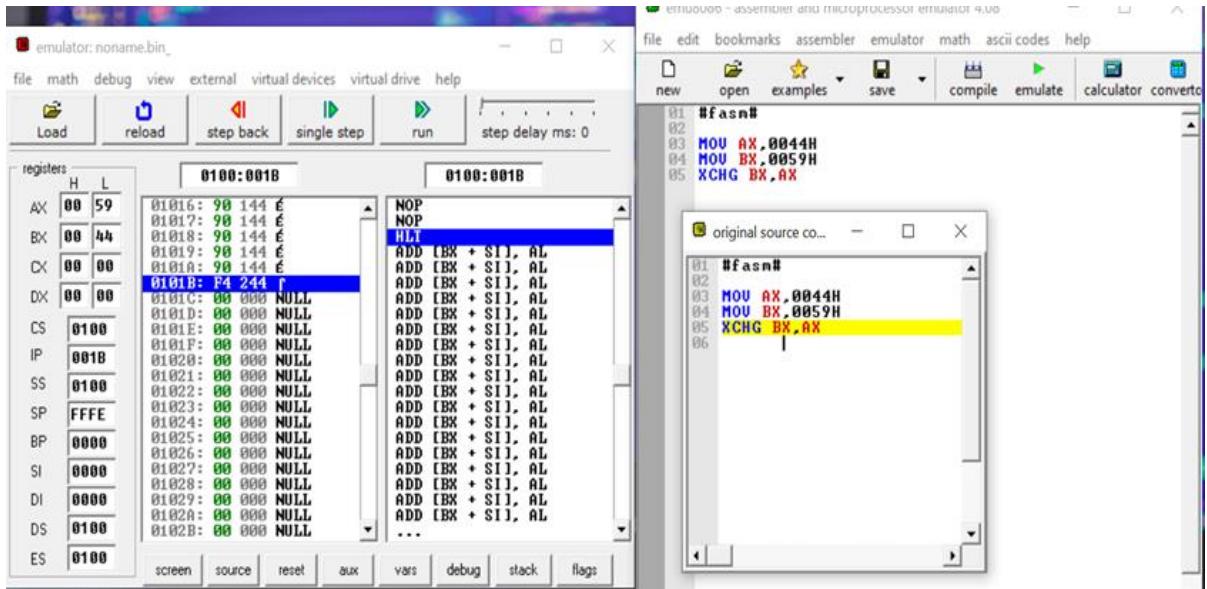
Instruction to transfer a word :

- 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 exchange 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.

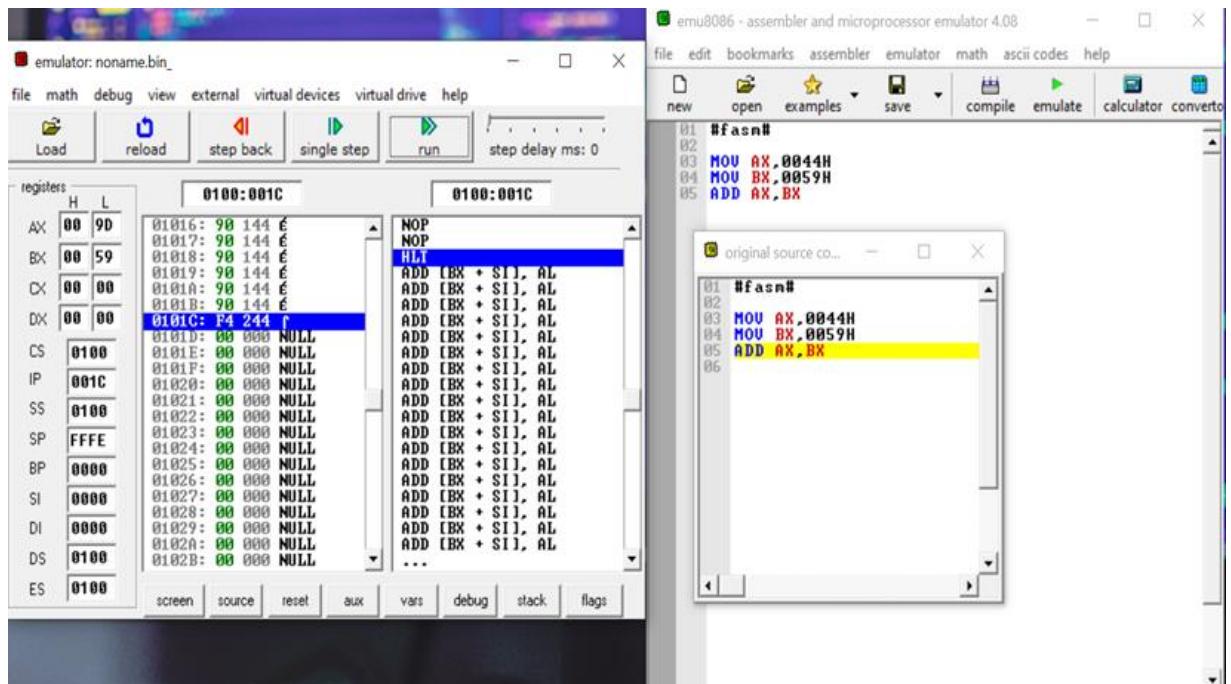
e.g.: POPF

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.

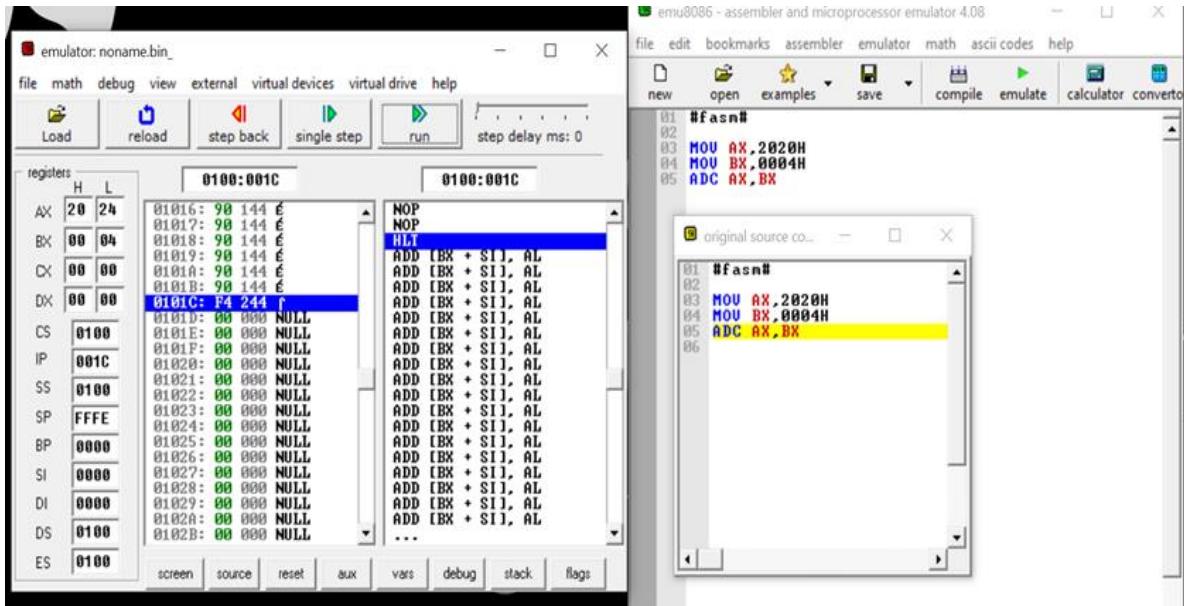
e.g.: ADD AX,2020H;
ADD AX, BX;



- **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;

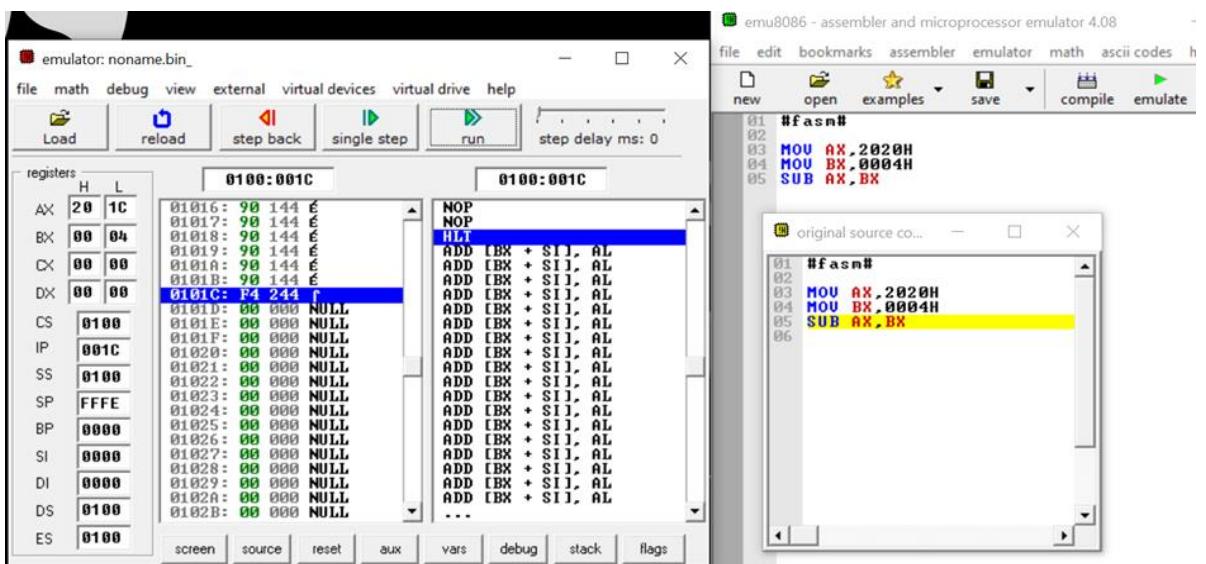
ADC AX, BX;



- **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;

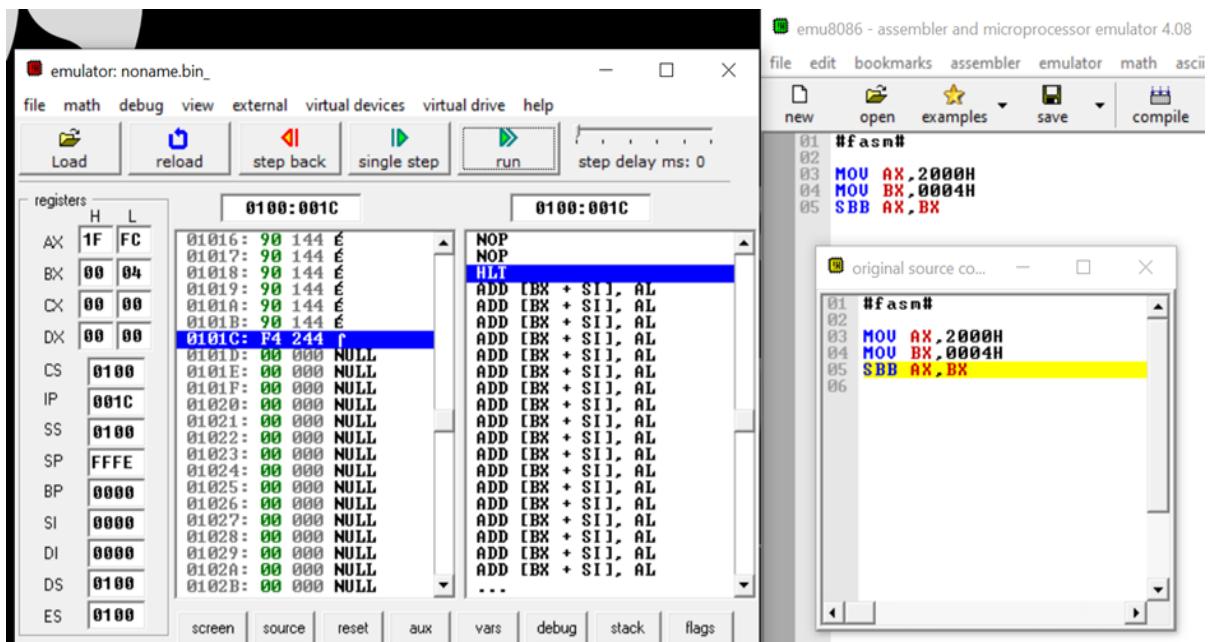
SUB AX, BX;



- **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];

The screenshot shows the emu8086 interface. The assembly pane on the right contains the following code:

```

#fasm#
MOU AX,0008H
INC AX

```

The registers pane shows the following values:

	H	L
AX	00	09
BX	00	00
CX	00	00
DX	00	00
CS	0100	
IP	0018	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The memory dump pane on the left shows the memory starting at address 0100:0018, where the instruction F4 2441 is highlighted.

- **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];

The screenshot shows the emu8086 interface after changing the assembly code. The assembly pane now contains:

```

#fasm#
MOU AX,0008H
DEC AX

```

The registers pane shows the following values:

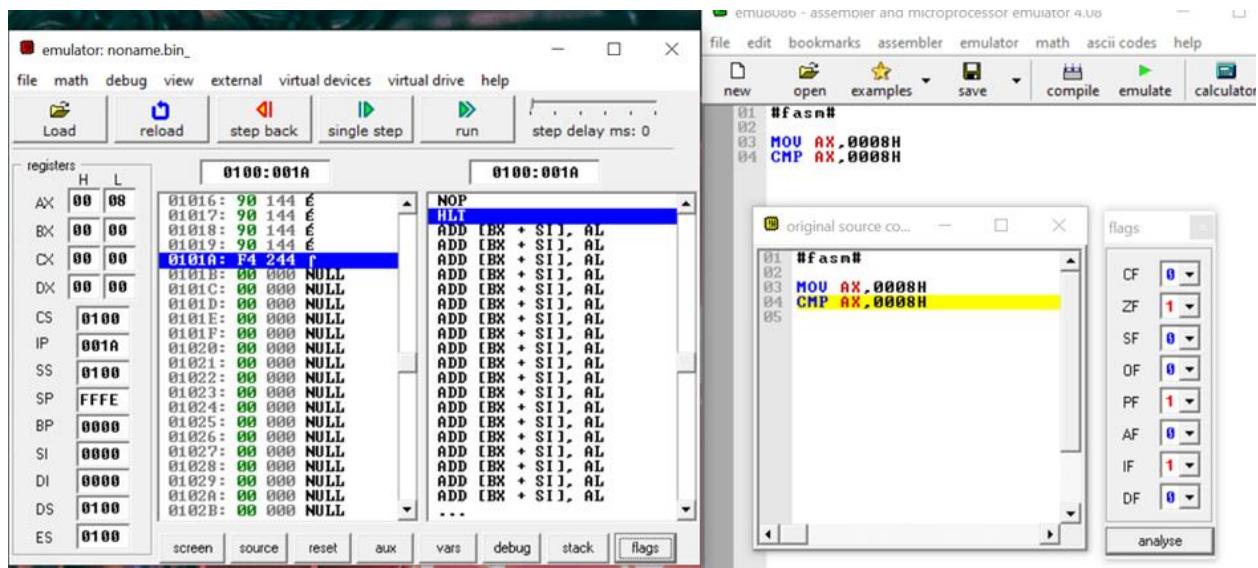
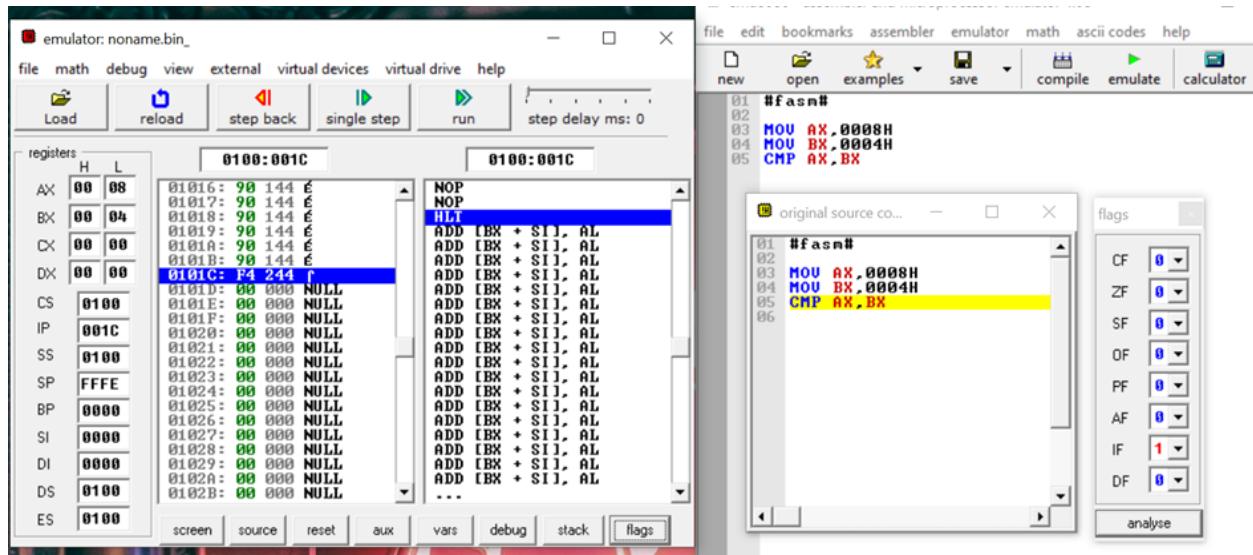
	H	L
AX	00	07
BX	00	00
CX	00	00
DX	00	00
CS	0100	
IP	0018	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The memory dump pane on the left shows the memory starting at address 0100:0018, where the instruction F4 2441 has been replaced by F4 2440.

- **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.

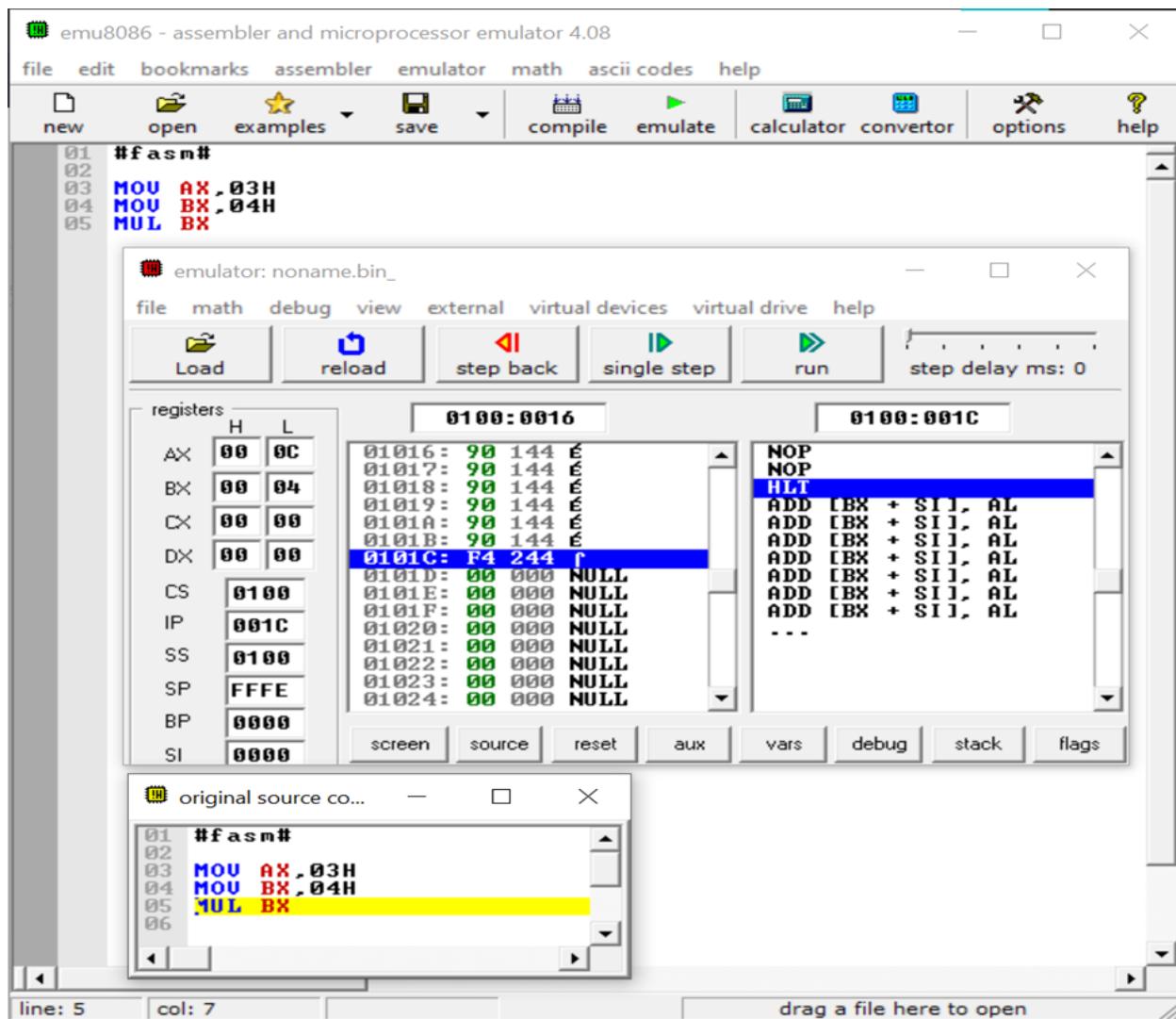
(1). IF lower bits of AL<=09 then, Higher bits of AL should be loaded with zeroes. There should be no change in lower bits of AL. AH also must be cleared.

(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.

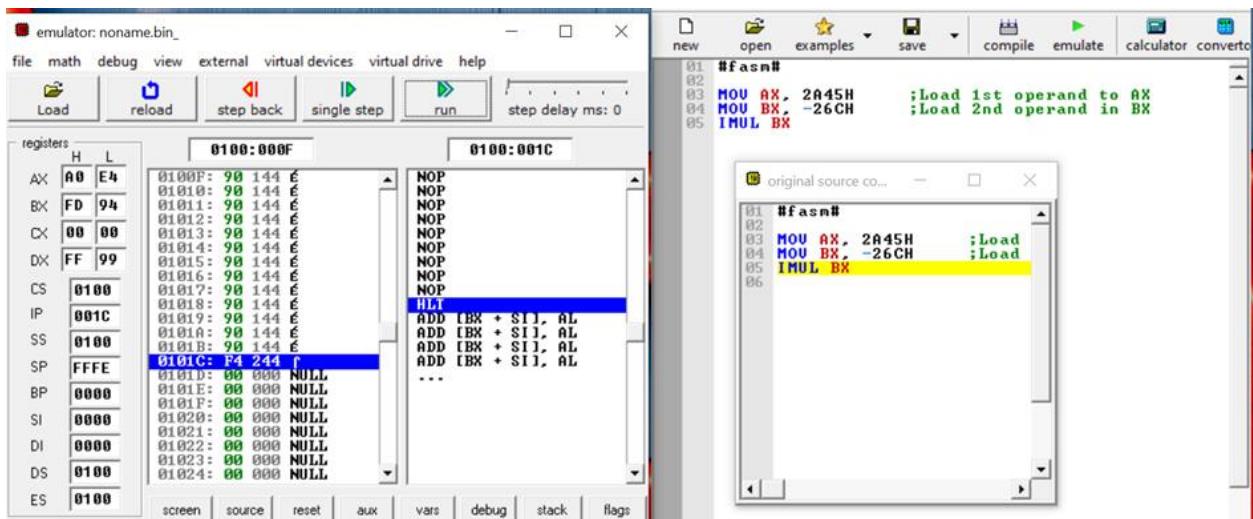
MUL CX // AX=AX*CX.



- **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)

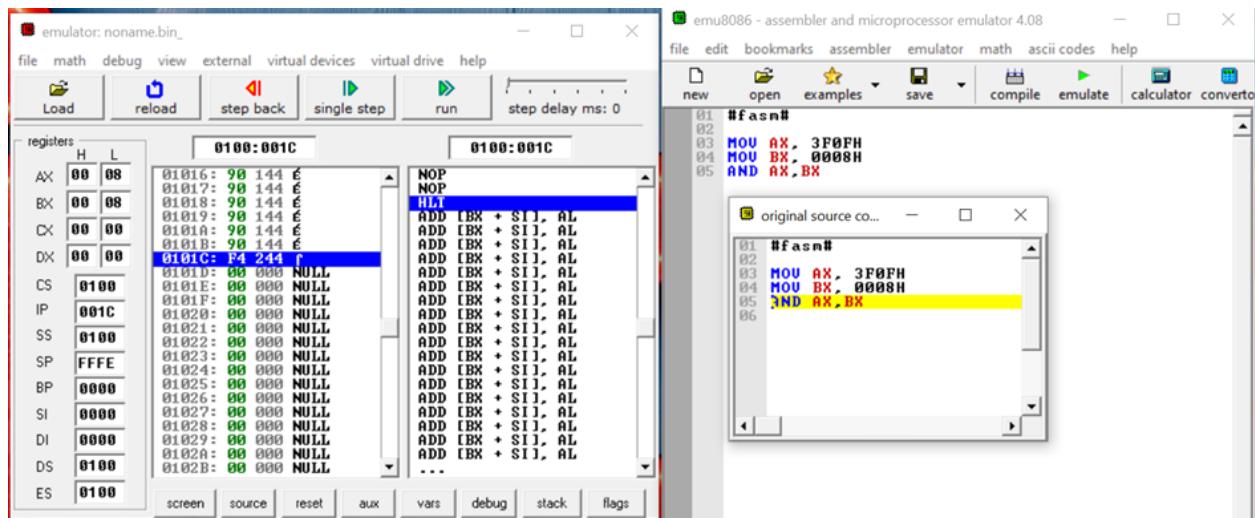
Logical Instructions

- **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.

The screenshot shows the emu8086 software interface. On the left, the 'Registers' window displays CPU register values. The AX register is set to 00 00. The right side shows the assembly code window with the following instructions:

```

    01 #fasm#
    02
    03 MOU AX, 3F0FH
    04 MOU BX, 0098H
    05 AND AX,BX

```

- **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

The screenshot shows the emu8086 software interface. The AX register is now set to C0 F0. The right side shows the assembly code window with the following instructions:

```

    01 #fasm#
    02
    03 MOU AX, 3F0FH
    04 not AX

```

- **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.

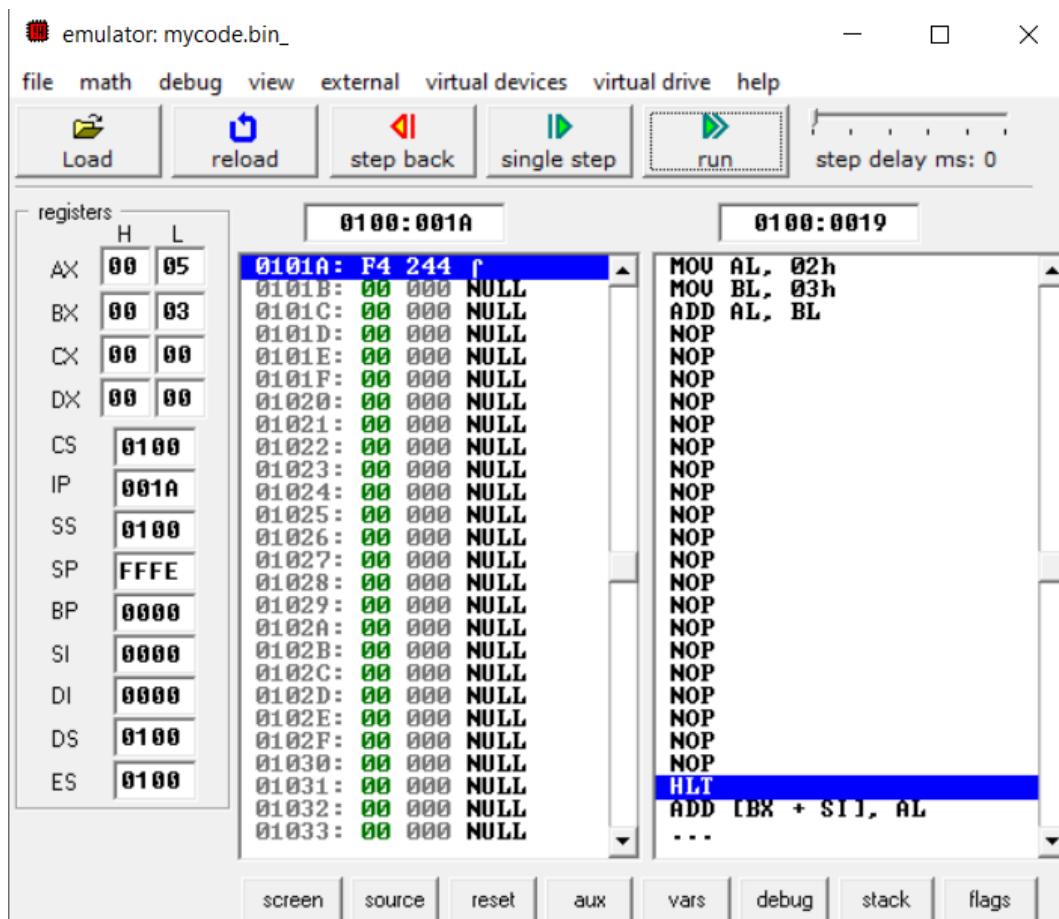
CODE:

1. For 8 bit number:

```
MOV AL, 02
```

```
MOV BL, 03
```

```
ADD AL,BL
```



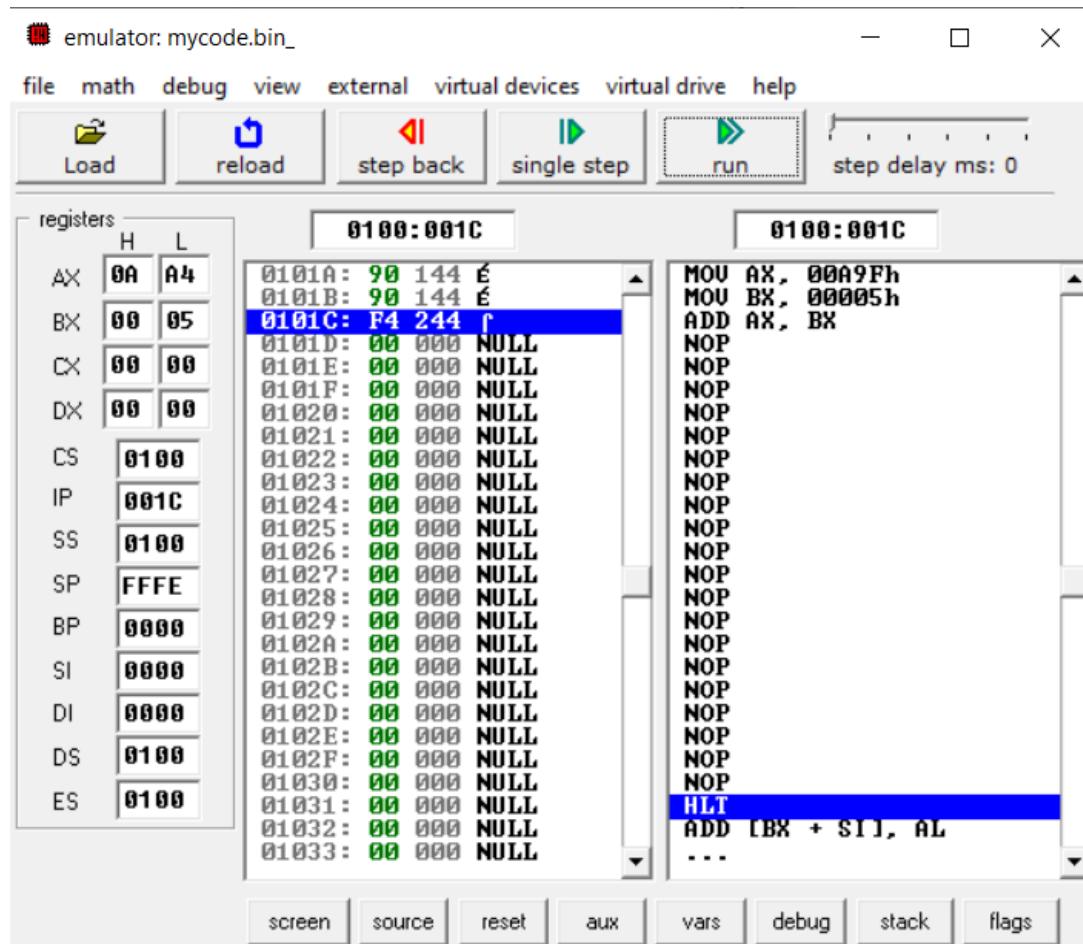
Output - The result will be stored in the AL register i.e.05

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

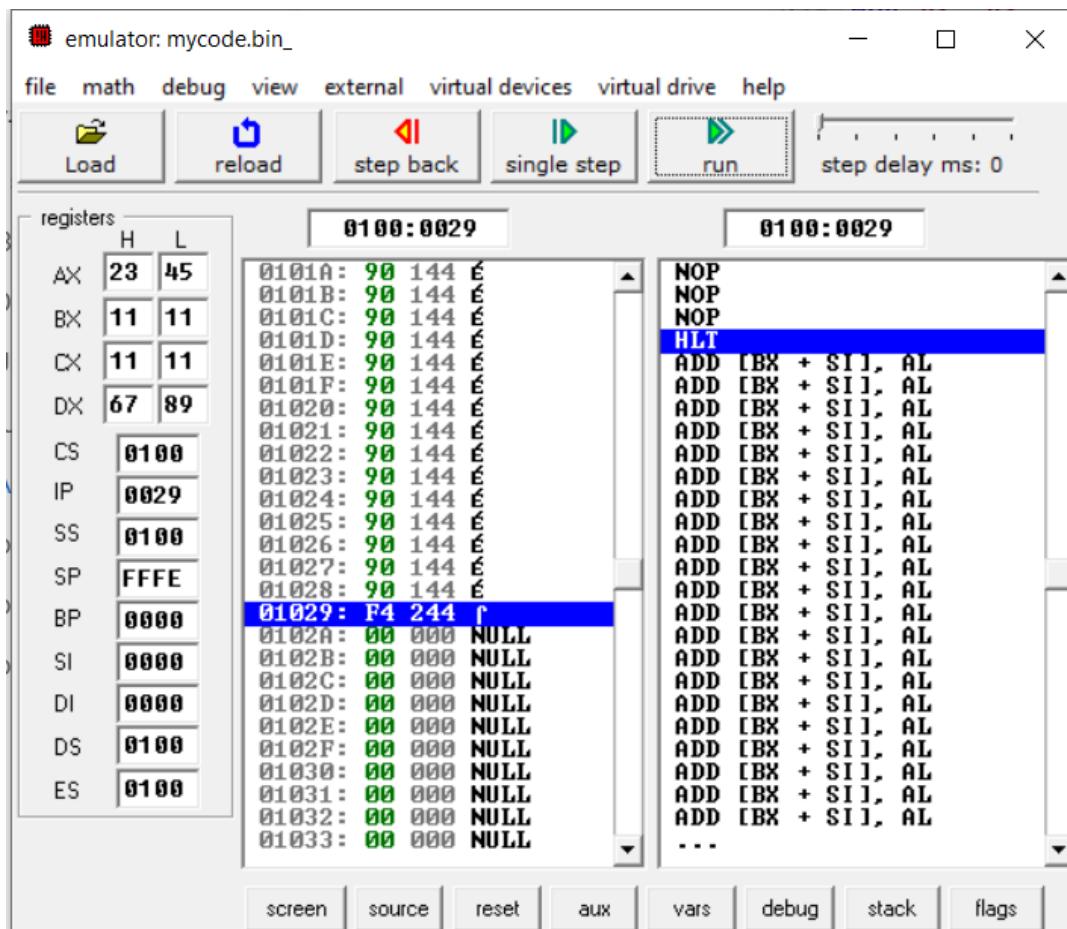
MOV BX, 1111H ; Move no 2 LSB in BX register

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.

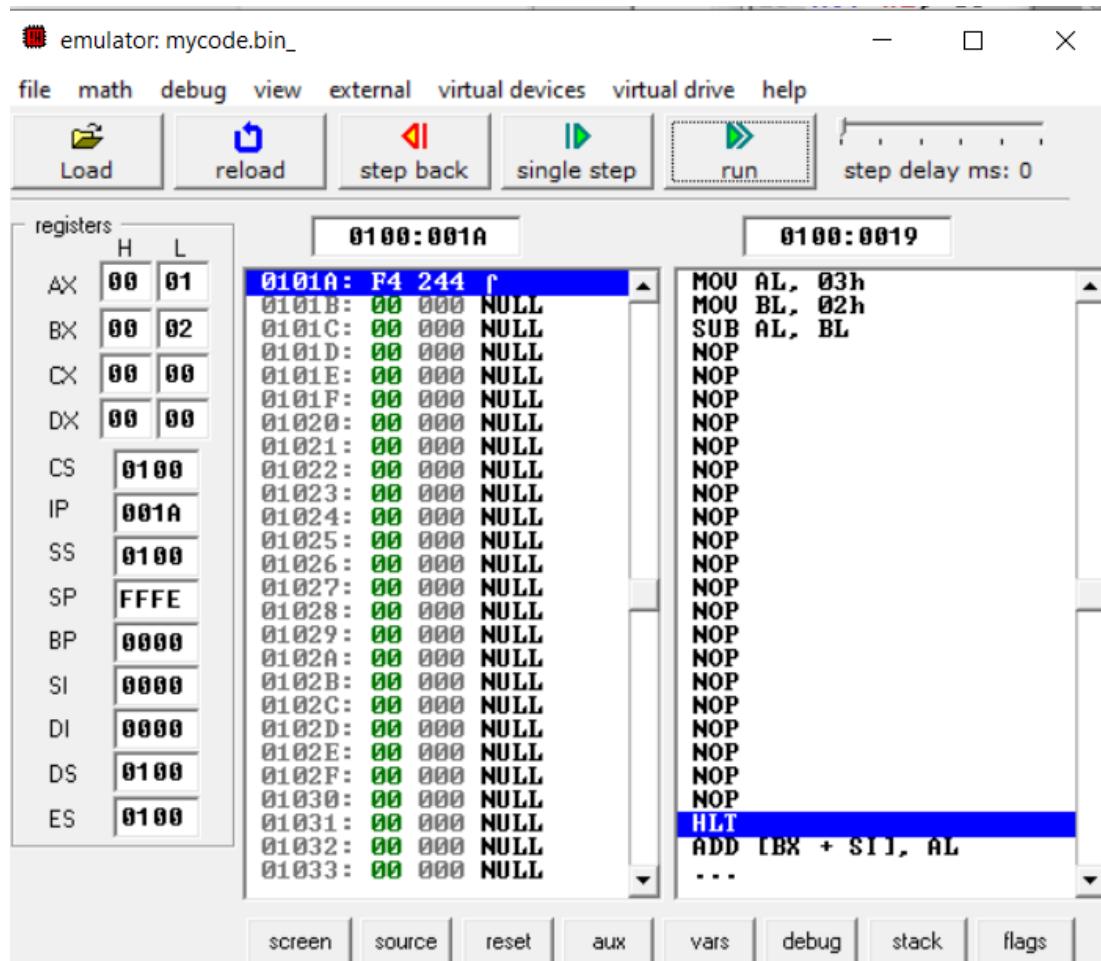
CODE:

1. For 8 bit number -

MOV AL, 03

MOV BL, 02

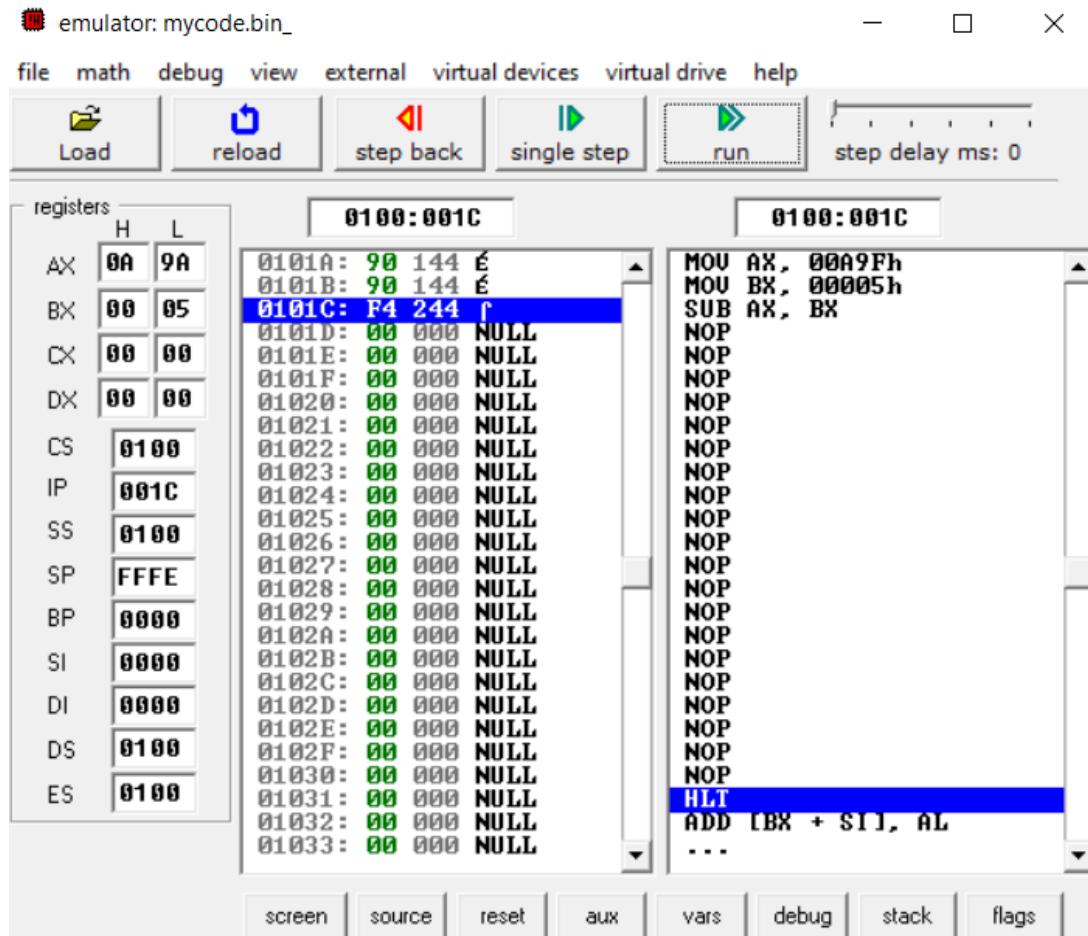
SUB AL, BL



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
```

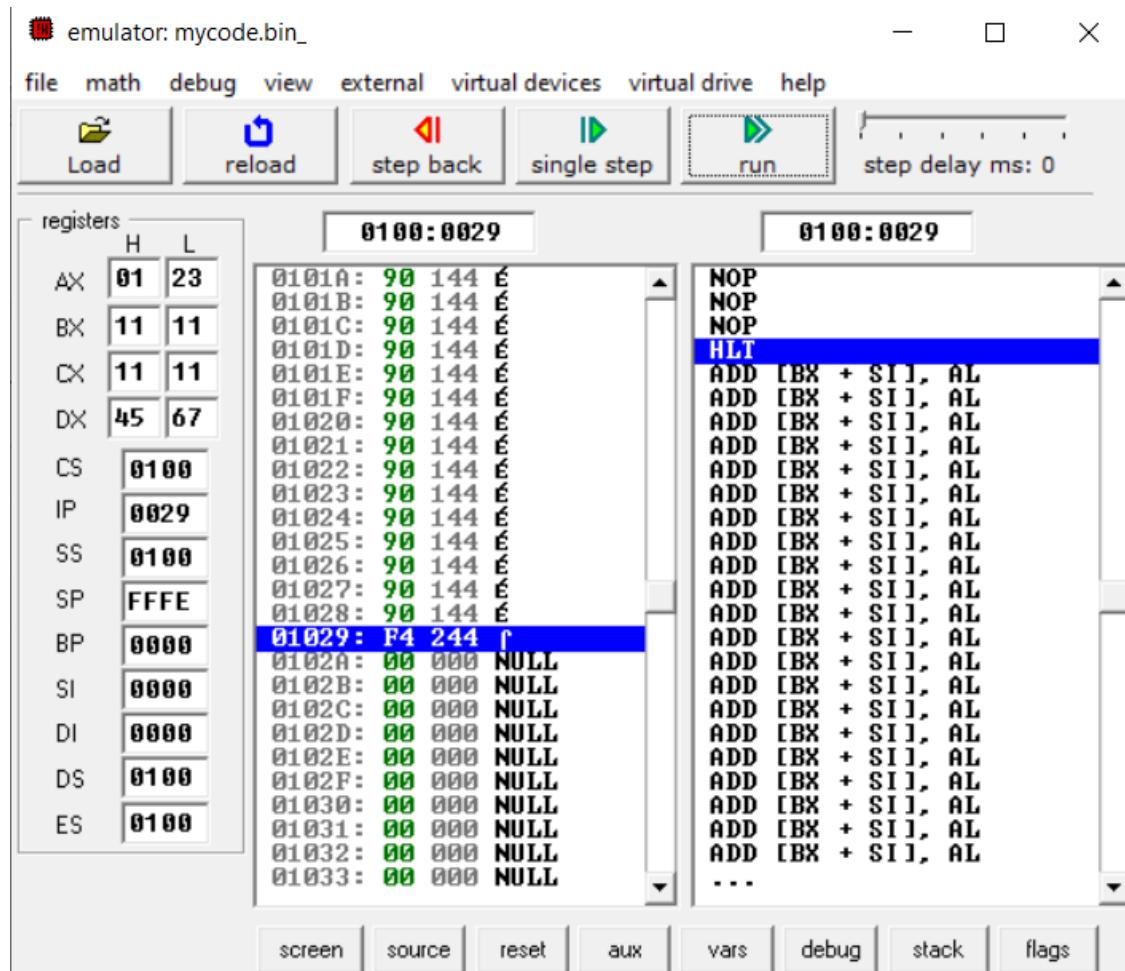


Output - The result will be stored in the AX register i.e 0A9A.

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



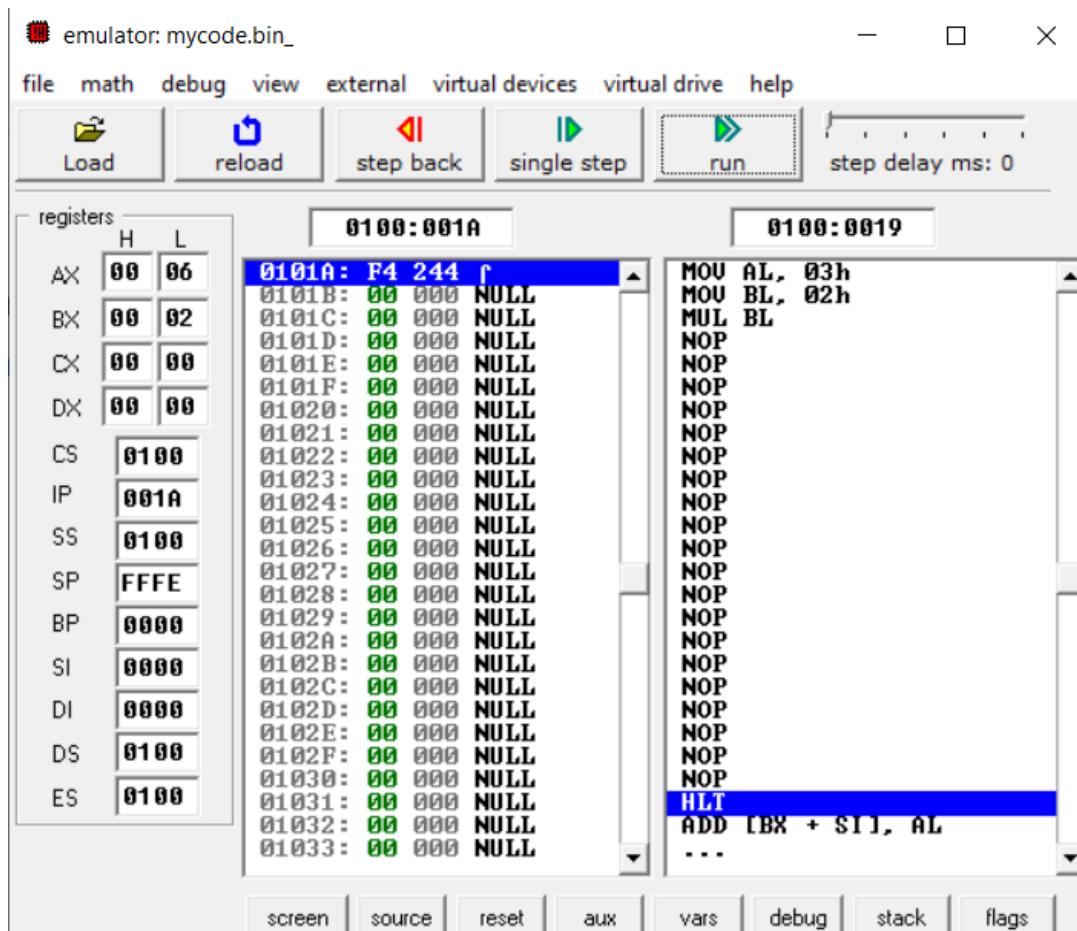
Output - The LSB and MSB of the result will be stored in the DX and AX register respectively i.e 01234567H.

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

CODE:

1. For 8 bit number -

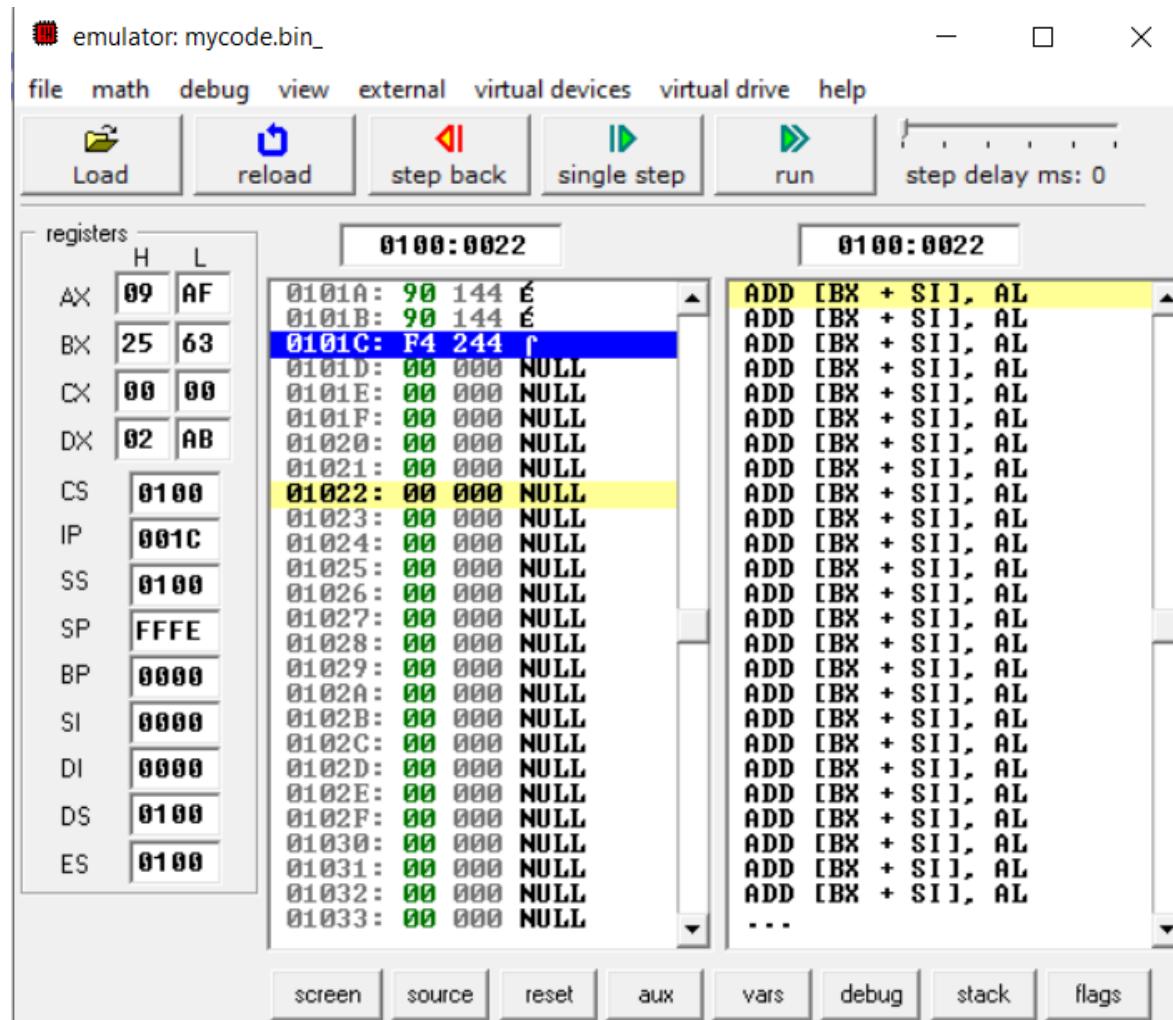
```
MOV AL, 03  
MOV BL, 02  
MUL BL
```



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



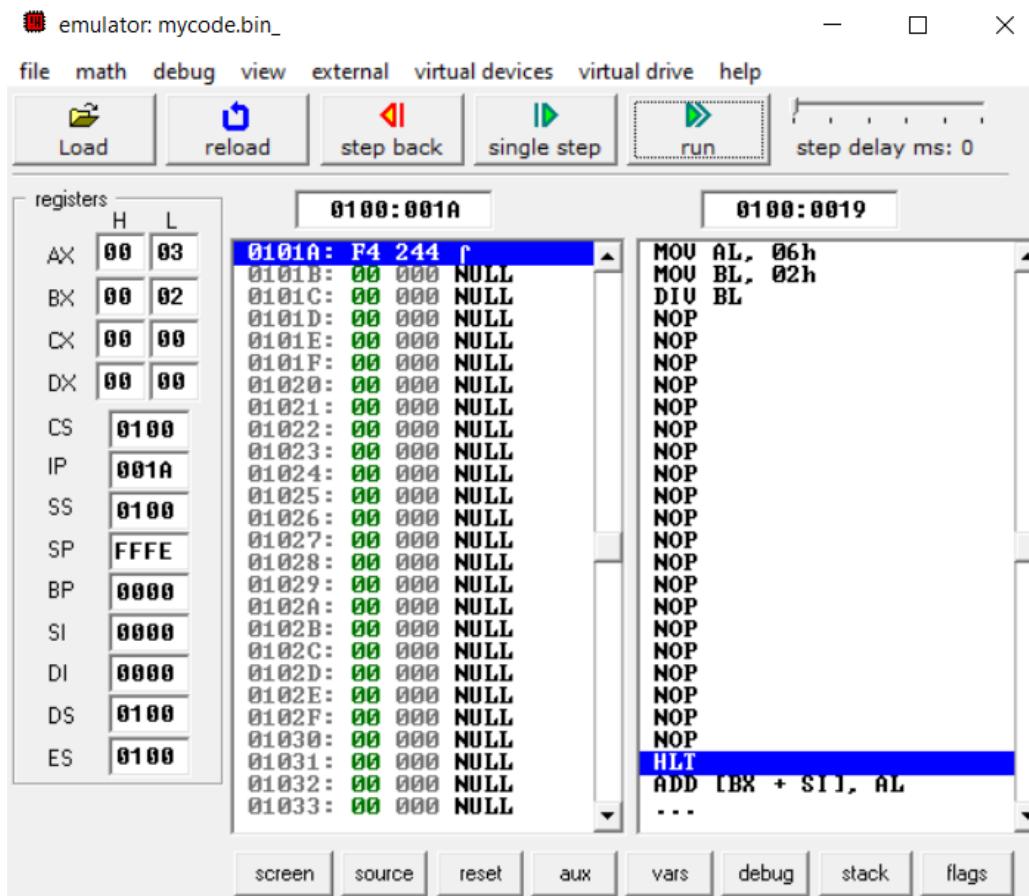
Output - The result will be stored in the DX and AX register i.e 02AB09AF.

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
```



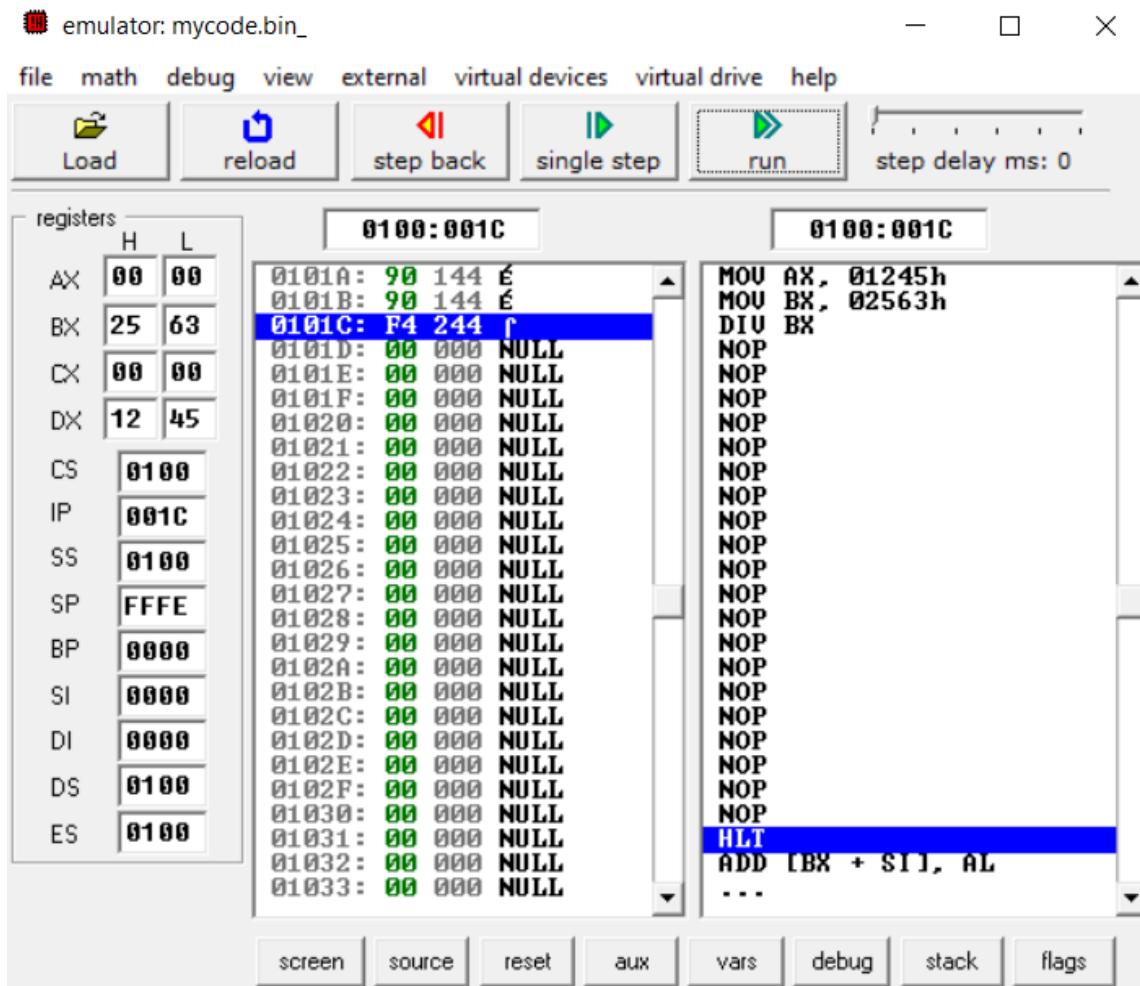
Output - The quotient will be stored in the AL register i.e 03 and the remainder in AH register i.e 0.

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

The screenshot shows a debugger window with the title "emulator: ANDOR.bin_". The menu bar includes file, math, debug, view, external, virtual devices, virtual drive, and help. The toolbar features Load, reload, step back, single step, run, and step delay ms: 0. The registers pane on the left displays the following values:

	H	L
AX	00	FD
BX	00	00
CX	00	00
DX	00	00
CS	0100	
IP	0018	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The assembly code pane shows the following sequence of instructions at address 0100:0018:

```
01000: B0 176
01001: 02 002 ⊞
01002: F6 246 ⊞
01003: D0 208 ⊞
01004: 90 144 ⊞
01005: 90 144 ⊞
01006: 90 144 ⊞
01007: 90 144 ⊞
01008: 90 144 ⊞
01009: 90 144 ⊞
0100A: 90 144 ⊞
0100B: 90 144 ⊞
0100C: 90 144 ⊞
0100D: 90 144 ⊞
0100E: 90 144 ⊞
0100F: 90 144 ⊞
01010: 90 144 ⊞
01011: 90 144 ⊞
01012: 90 144 ⊞
01013: 90 144 ⊞
01014: 90 144 ⊞
01015: 90 144 ⊞
01016: 90 144 ⊞
01017: 90 144 ⊞
01018: F4 244 ⊞
01019: 00 000 NULL
```

The instruction at address 01018 is highlighted in blue. The right pane shows the stack dump starting at address 0100:0018:

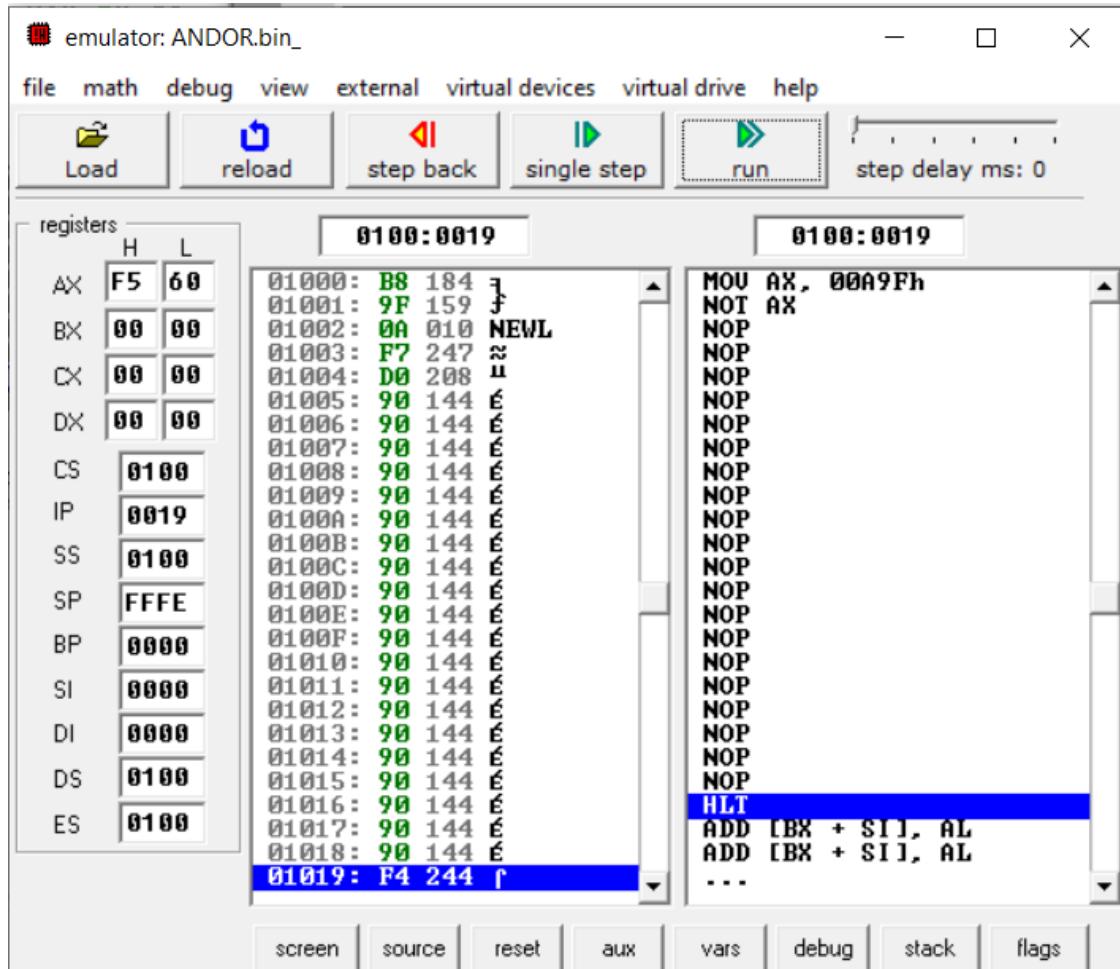
Address	Value	Content
0100:0018	MOU AL, 02h	
0100:0019	NOT AL	
0100:001A	NOP	
0100:001B	NOP	
0100:001C	NOP	
0100:001D	NOP	
0100:001E	NOP	
0100:001F	NOP	
0100:0020	NOP	
0100:0021	NOP	
0100:0022	NOP	
0100:0023	NOP	
0100:0024	NOP	
0100:0025	NOP	
0100:0026	NOP	
0100:0027	NOP	
0100:0028	NOP	
0100:0029	NOP	
0100:002A	NOP	
0100:002B	NOP	
0100:002C	NOP	
0100:002D	NOP	
0100:002E	NOP	
0100:002F	NOP	
0100:0030	NOP	
0100:0031	NOP	
0100:0032	NOP	
0100:0033	NOP	
0100:0034	NOP	
0100:0035	NOP	
0100:0036	NOP	
0100:0037	NOP	
0100:0038	NOP	
0100:0039	NOP	
0100:003A	NOP	
0100:003B	NOP	
0100:003C	NOP	
0100:003D	NOP	
0100:003E	NOP	
0100:003F	NOP	
0100:0040	NOP	
0100:0041	NOP	
0100:0042	NOP	
0100:0043	NOP	
0100:0044	NOP	
0100:0045	NOP	
0100:0046	NOP	
0100:0047	NOP	
0100:0048	NOP	
0100:0049	NOP	
0100:004A	NOP	
0100:004B	NOP	
0100:004C	NOP	
0100:004D	NOP	
0100:004E	NOP	
0100:004F	NOP	
0100:0050	NOP	
0100:0051	NOP	
0100:0052	NOP	
0100:0053	NOP	
0100:0054	NOP	
0100:0055	NOP	
0100:0056	NOP	
0100:0057	NOP	
0100:0058	NOP	
0100:0059	NOP	
0100:005A	NOP	
0100:005B	NOP	
0100:005C	NOP	
0100:005D	NOP	
0100:005E	NOP	
0100:005F	NOP	
0100:0060	NOP	
0100:0061	NOP	
0100:0062	NOP	
0100:0063	NOP	
0100:0064	NOP	
0100:0065	NOP	
0100:0066	NOP	
0100:0067	NOP	
0100:0068	NOP	
0100:0069	NOP	
0100:006A	NOP	
0100:006B	NOP	
0100:006C	NOP	
0100:006D	NOP	
0100:006E	NOP	
0100:006F	NOP	
0100:0070	NOP	
0100:0071	NOP	
0100:0072	NOP	
0100:0073	NOP	
0100:0074	NOP	
0100:0075	NOP	
0100:0076	NOP	
0100:0077	NOP	
0100:0078	NOP	
0100:0079	NOP	
0100:007A	NOP	
0100:007B	NOP	
0100:007C	NOP	
0100:007D	NOP	
0100:007E	NOP	
0100:007F	NOP	
0100:0080	NOP	
0100:0081	NOP	
0100:0082	NOP	
0100:0083	NOP	
0100:0084	NOP	
0100:0085	NOP	
0100:0086	NOP	
0100:0087	NOP	
0100:0088	NOP	
0100:0089	NOP	
0100:008A	NOP	
0100:008B	NOP	
0100:008C	NOP	
0100:008D	NOP	
0100:008E	NOP	
0100:008F	NOP	
0100:0090	NOP	
0100:0091	NOP	
0100:0092	NOP	
0100:0093	NOP	
0100:0094	NOP	
0100:0095	NOP	
0100:0096	NOP	
0100:0097	NOP	
0100:0098	NOP	
0100:0099	NOP	
0100:009A	NOP	
0100:009B	NOP	
0100:009C	NOP	
0100:009D	NOP	
0100:009E	NOP	
0100:009F	NOP	
0100:00A0	NOP	
0100:00A1	NOP	
0100:00A2	NOP	
0100:00A3	NOP	
0100:00A4	NOP	
0100:00A5	NOP	
0100:00A6	NOP	
0100:00A7	NOP	
0100:00A8	NOP	
0100:00A9	NOP	
0100:00AA	NOP	
0100:00AB	NOP	
0100:00AC	NOP	
0100:00AD	NOP	
0100:00AE	NOP	
0100:00AF	NOP	
0100:00B0	NOP	
0100:00B1	NOP	
0100:00B2	NOP	
0100:00B3	NOP	
0100:00B4	NOP	
0100:00B5	NOP	
0100:00B6	NOP	
0100:00B7	NOP	
0100:00B8	NOP	
0100:00B9	NOP	
0100:00BA	NOP	
0100:00BB	NOP	
0100:00BC	NOP	
0100:00BD	NOP	
0100:00BE	NOP	
0100:00BF	NOP	
0100:00C0	NOP	
0100:00C1	NOP	
0100:00C2	NOP	
0100:00C3	NOP	
0100:00C4	NOP	
0100:00C5	NOP	
0100:00C6	NOP	
0100:00C7	NOP	
0100:00C8	NOP	
0100:00C9	NOP	
0100:00CA	NOP	
0100:00CB	NOP	
0100:00CC	NOP	
0100:00CD	NOP	
0100:00CE	NOP	
0100:00CF	NOP	
0100:00D0	NOP	
0100:00D1	NOP	
0100:00D2	NOP	
0100:00D3	NOP	
0100:00D4	NOP	
0100:00D5	NOP	
0100:00D6	NOP	
0100:00D7	NOP	
0100:00D8	NOP	
0100:00D9	NOP	
0100:00DA	NOP	
0100:00DB	NOP	
0100:00DC	NOP	
0100:00DD	NOP	
0100:00DE	NOP	
0100:00DF	NOP	
0100:00E0	NOP	
0100:00E1	NOP	
0100:00E2	NOP	
0100:00E3	NOP	
0100:00E4	NOP	
0100:00E5	NOP	
0100:00E6	NOP	
0100:00E7	NOP	
0100:00E8	NOP	
0100:00E9	NOP	
0100:00EA	NOP	
0100:00EB	NOP	
0100:00EC	NOP	
0100:00ED	NOP	
0100:00EE	NOP	
0100:00EF	NOP	
0100:00F0	NOP	
0100:00F1	NOP	
0100:00F2	NOP	
0100:00F3	NOP	
0100:00F4	NOP	
0100:00F5	NOP	
0100:00F6	NOP	
0100:00F7	NOP	
0100:00F8	NOP	
0100:00F9	NOP	
0100:00FA	NOP	
0100:00FB	NOP	
0100:00FC	NOP	
0100:00FD	NOP	
0100:00FE	NOP	
0100:00FF	NOP	

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

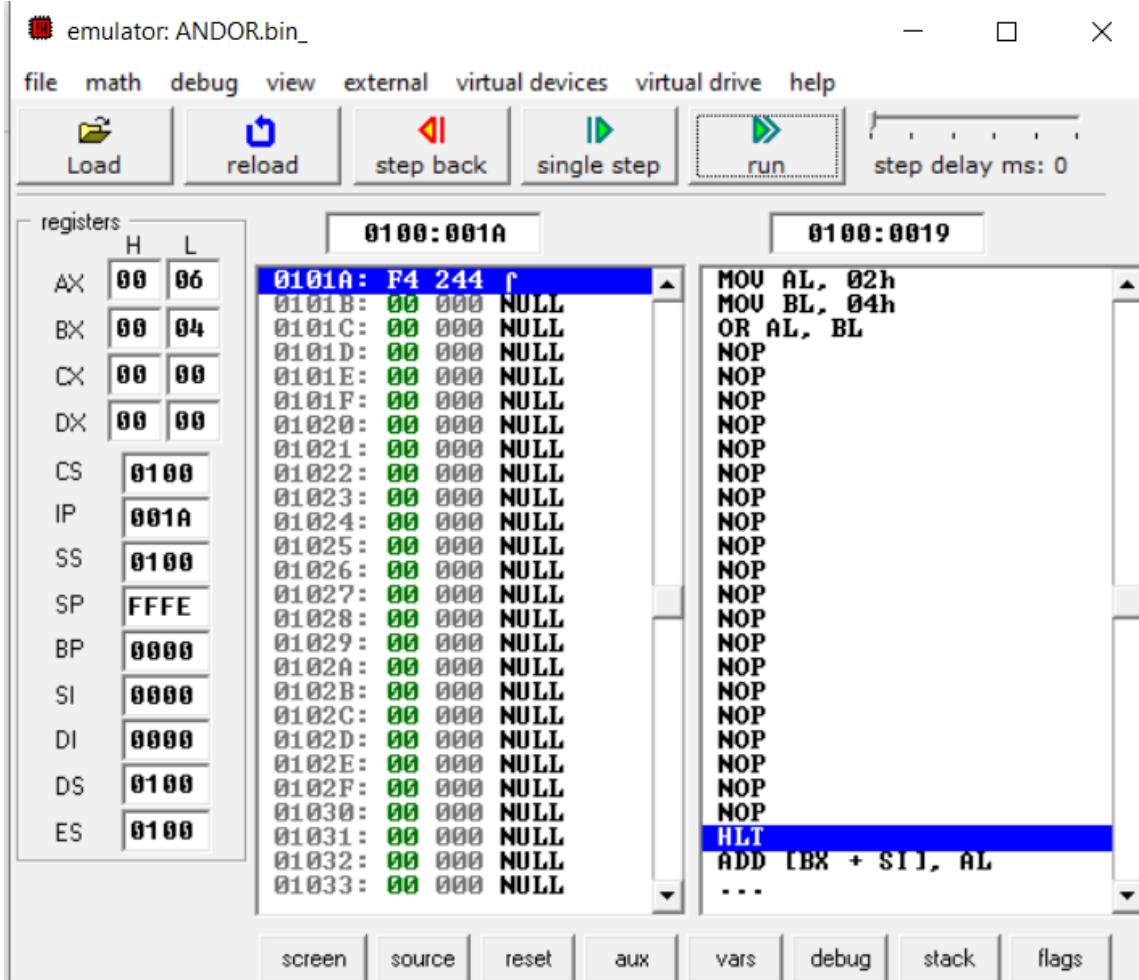
- OR

1. For 8 bit number -

MOV AL,02

MOV BL,04

OR AL,BL



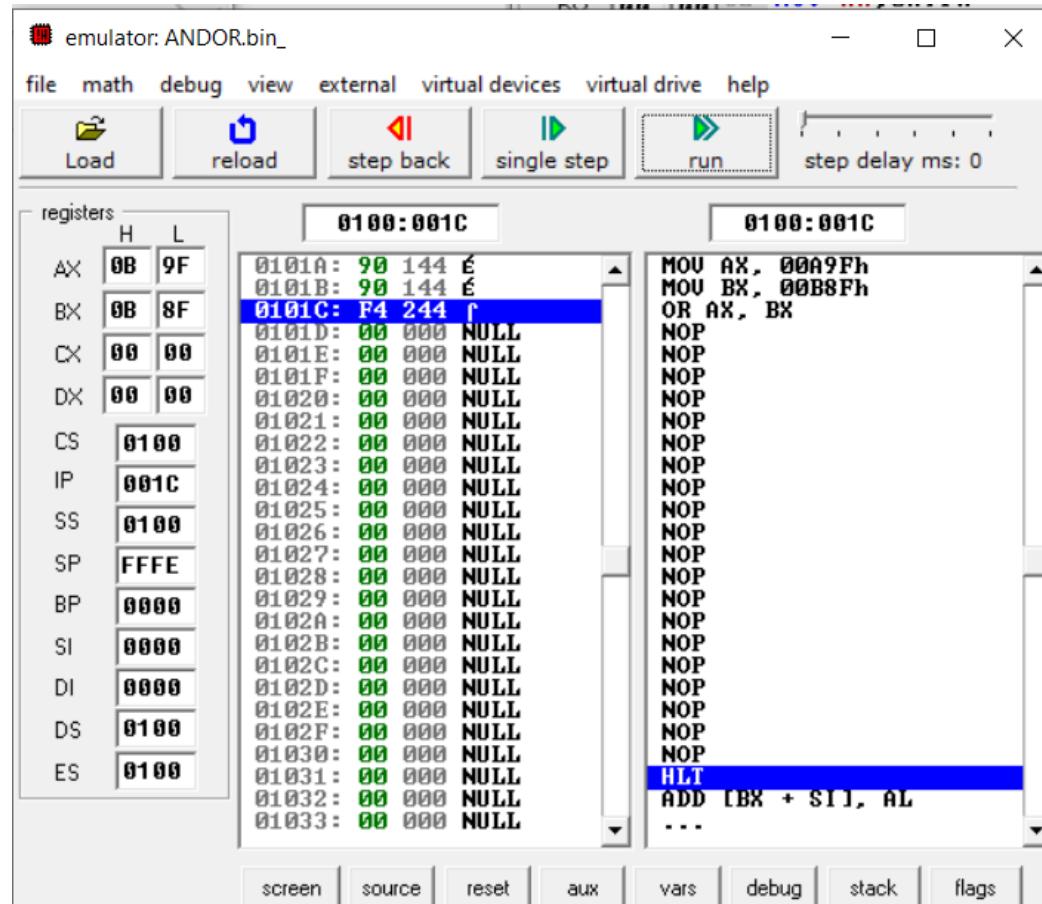
Output - Output is stored in AL register i.e 06

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

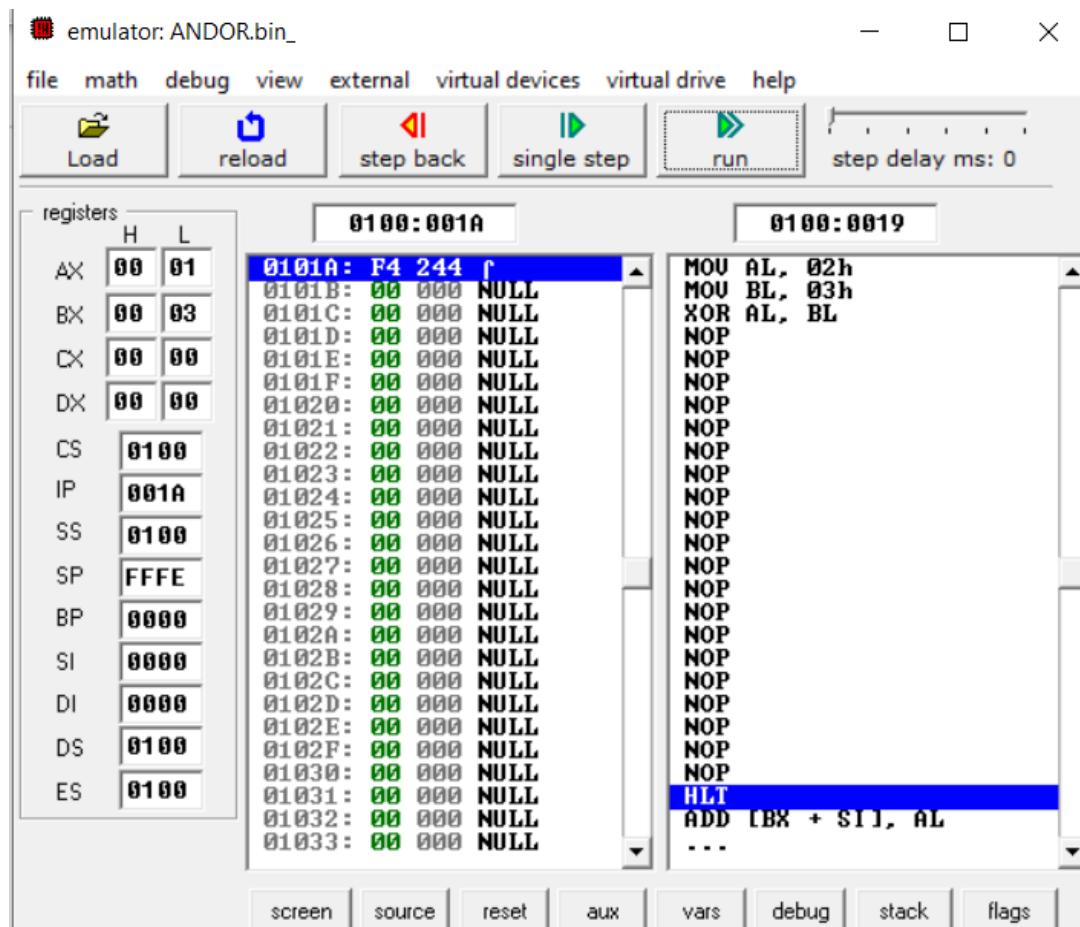
- XOR

1. For 8 bit number-

MOV AL,02

MOV BL,03

XOR AL,BL



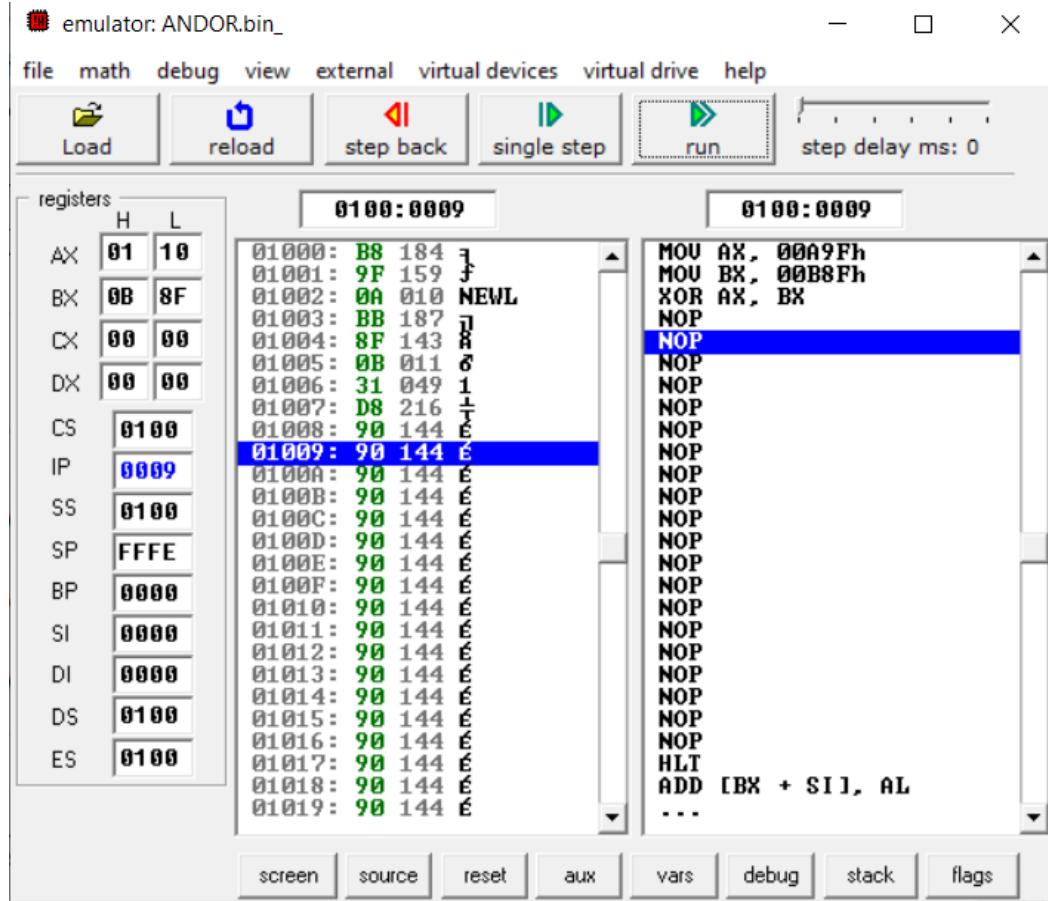
Output - Output is stored in AL register i.e 01

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

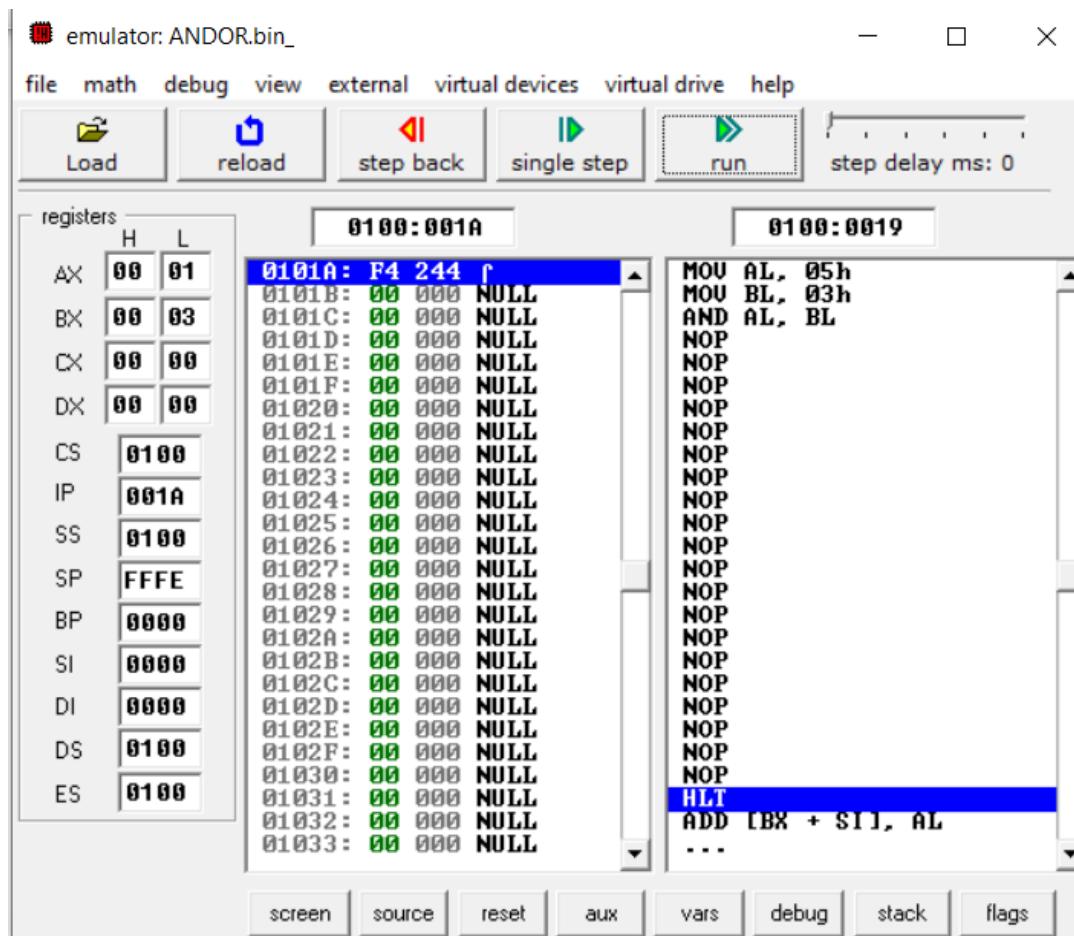
- AND

1. For 8 bit number-

MOV AL,05

MOV BL,03

AND AL,BL



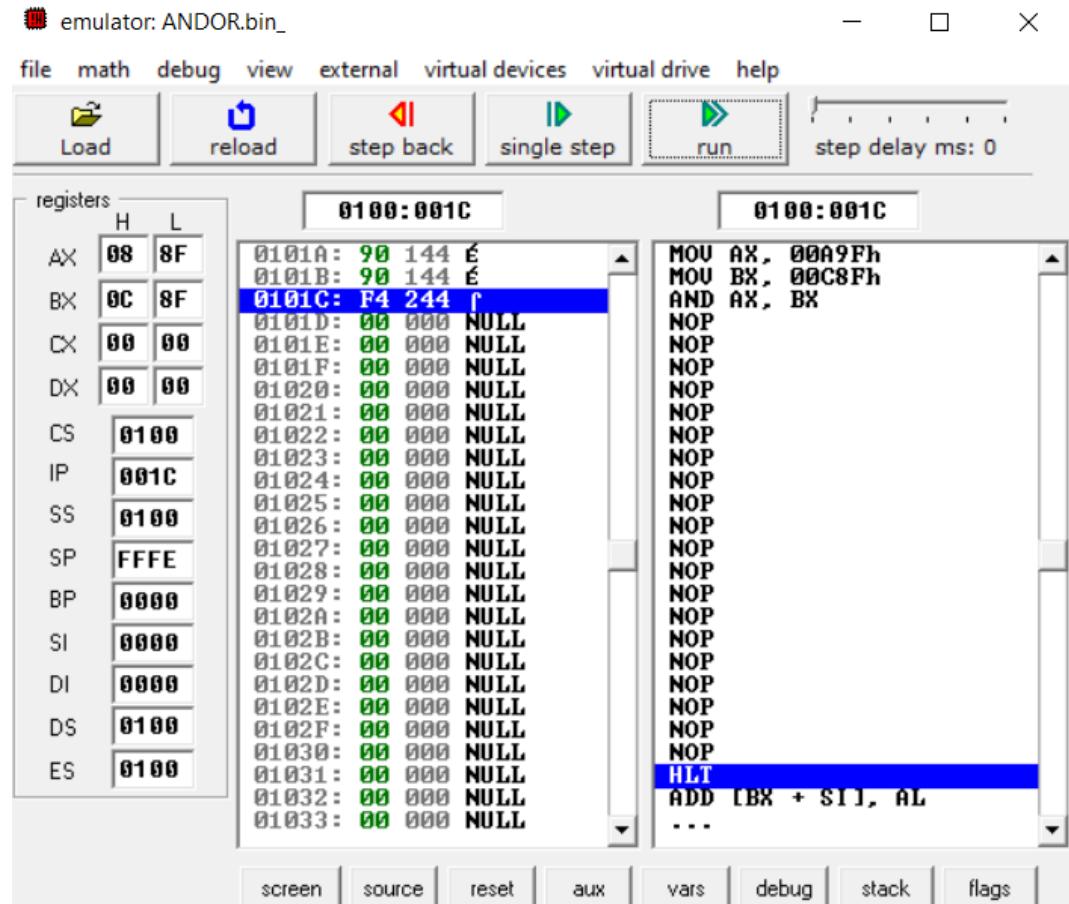
Output - Output is stored in AL register i.e 01

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

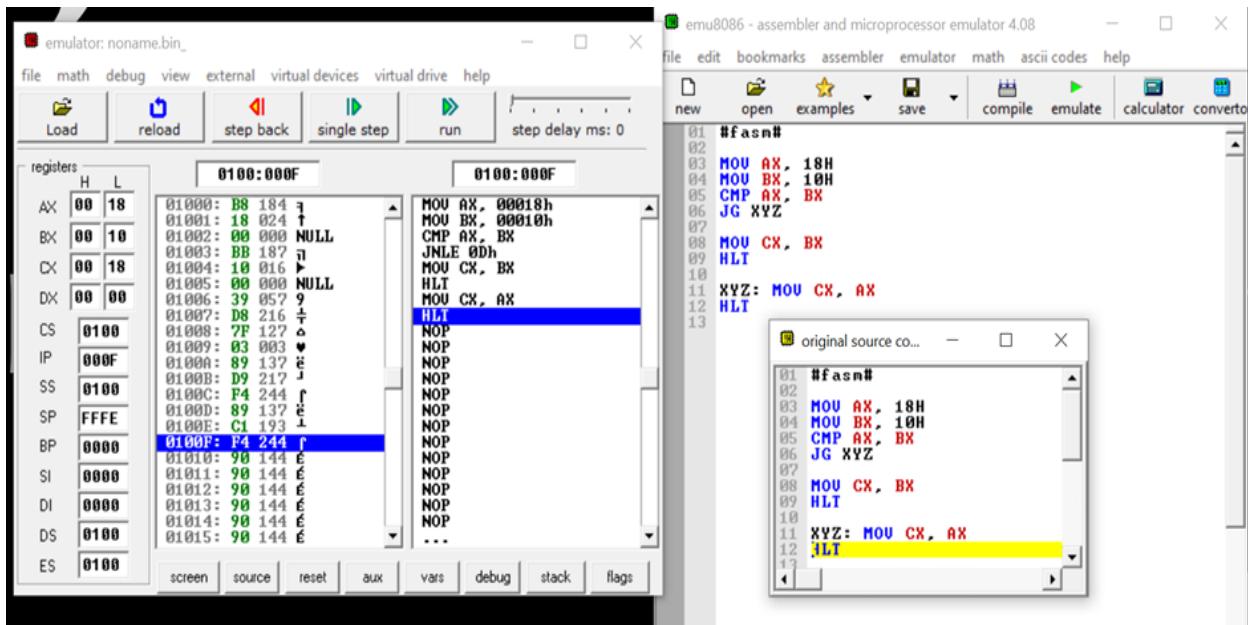
CMP AX, BX

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

The screenshot shows the emu8086 interface with two main windows. The left window displays the assembly code and memory dump, while the right window shows the register state.

Registers Window:

	H	L
AX	12	78
BX	12	90
CX	12	90
DX	00	00
CS	0100	
IP	000C	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

Code Window:

```

01 #fasm#
02
03 MOU AX, 1278H
04 MOU BX, 1290H
05 CMP AX, BX
06 JG XYZ
07
08 MOU CX, BX
09 HLT
10
11 XYZ: MOU CX, AX
12 HLT
13

```

Original Source Code Window:

```

01 #fasm#
02
03 MOU AX, 1278H
04 MOU BX, 1290H
05 CMP AX, BX
06 JG XYZ
07
08 MOU CX, BX
09 HLT
10
11 XYZ: MOU CX, AX
12 HLT
13

```

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

emulator: MaxFromArray.com_

file math debug view external virtual devices virtual drive help

Load reload step back single step run step delay ms: 0

registers		0700:011E	0700:011E
	H L		
AX	00 06	0710F: 46 070 F	MOU SI, 00102h
BX	00 06	07110: 8A 138 è	MOU CL, 04h
CX	00 06	07111: 1C 028 L	MOU AL, [SI]
DX	00 00	07112: 3A 058 :	DEC CL
CS	0700	07113: C3 195 T	INC SI
IP	011E	07114: 73 115 s	MOU BL, [SI]
SS	0700	07115: 02 002 @	CMP AL, BL
SP	FFFE	07116: 8A 138 è	JNB 0118h
BP	0000	07117: C3 195 T	MOU AL, BL
SI	0105	07118: FE 254 I	DEC CL
DI	0000	07119: C9 201 R	JNE 010Fh
DS	0700	0711A: 75 117 u	MOU CL, AL
ES	0700	0711B: F3 243 S	HLT
		0711C: 8A 138 è	RET
		0711D: C8 200 L	NOP
		0711E: F4 244 T	NOP
		0711F: C3 195 T	NOP
		07120: 90 144 É	NOP
		07121: 90 144 É	NOP
		07122: 90 144 É	NOP
		07123: 90 144 É	NOP
		07124: 90 144 É	NOP
		07125: 90 144 É	NOP
		07126: 90 144 É	NOP
		07127: 90 144 É	NOP
		07128: 90 144 É	NOP
			...

screen source reset aux vars debug stack flags

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

emulator: MinFromArray.com_

file math debug view external virtual devices virtual drive help

Load reload step back single step run step delay ms: 0

registers		0700:011E	0700:011E
AX	H L	0710F: 46 070 F	MOU SI, 00102h
BX	H L	07110: 8A 138 è	MOU CL, 04h
CX	H L	07111: 1C 028 ↴	MOU AL, [SI]
DX	H L	07112: 3A 058 :	DEC CL
CS	0700	07113: C3 195 ↴	INC SI
IP	011E	07114: 72 114 r	MOU BL, [SI]
SS	0700	07115: 02 002 ø	CMP AL, BL
SP	FFFE	07116: 8A 138 è	JB 0118h
BP	0000	07117: C3 195 ↴	MOU AL, BL
SI	0105	07118: FE 254 ↴	DEC CL
DI	0000	07119: C9 201 ↴	JNE 010Fh
DS	0700	0711A: 75 117 u	MOU CL, AL
ES	0700	0711B: F3 243 ↴	HLT
		0711C: 8A 138 è	RET
		0711D: C8 200 u	NOP
		0711E: F4 244 ↴	NOP
		0711F: C3 195 ↴	NOP
		07120: 90 144 É	NOP
		07121: 90 144 É	NOP
		07122: 90 144 É	NOP
		07123: 90 144 É	NOP
		07124: 90 144 É	NOP
		07125: 90 144 É	NOP
		07126: 90 144 É	NOP
		07127: 90 144 É	NOP
		07128: 90 144 É	...

screen source reset aux vars debug stack flags

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
```

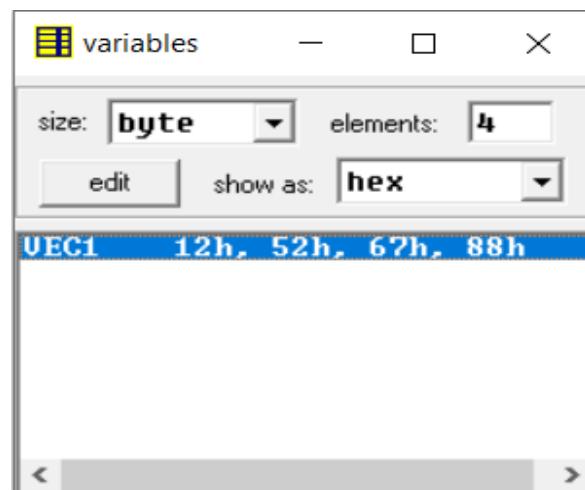
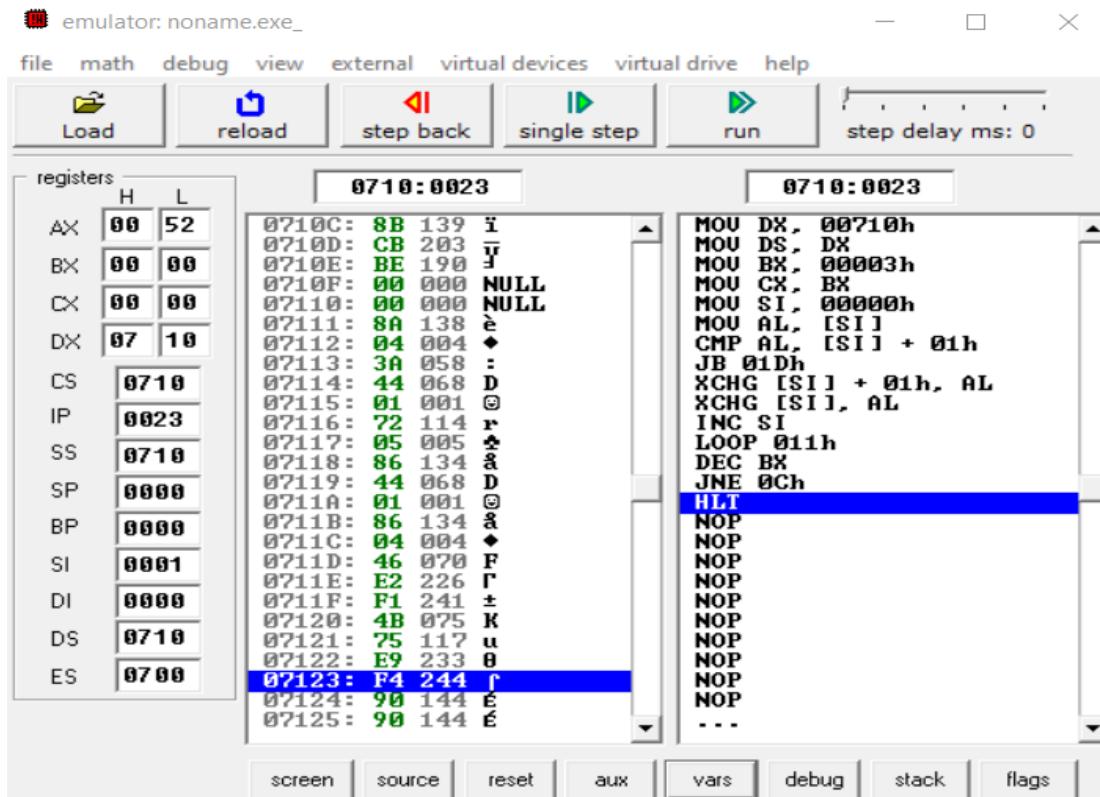
```
l2:  
    mov al,[si]  
    cmp al, [si+1]  
    jc xx;  
    xchg al,[si+1]  
    xchg al,[si]
```

```
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
```

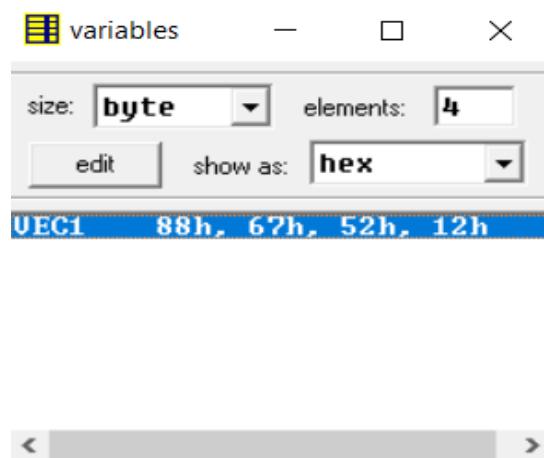
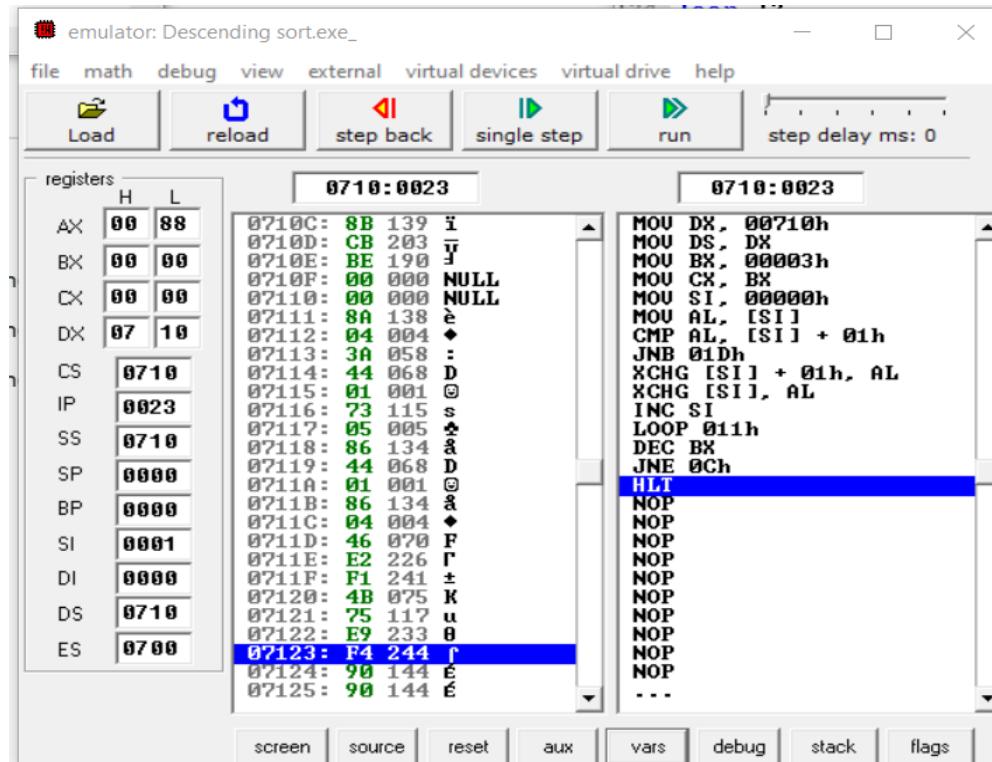
```
start:  
    mov dx, data  
    mov ds, dx  
    mov bx,3
```

```
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

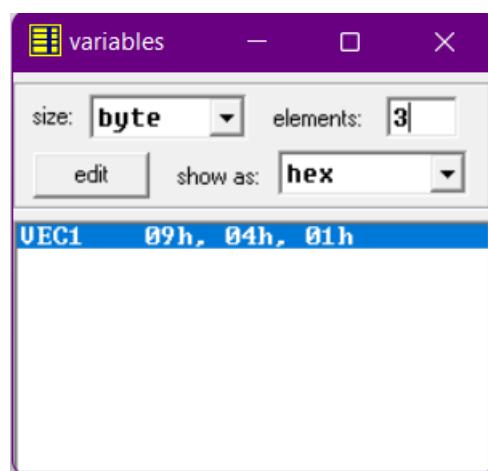
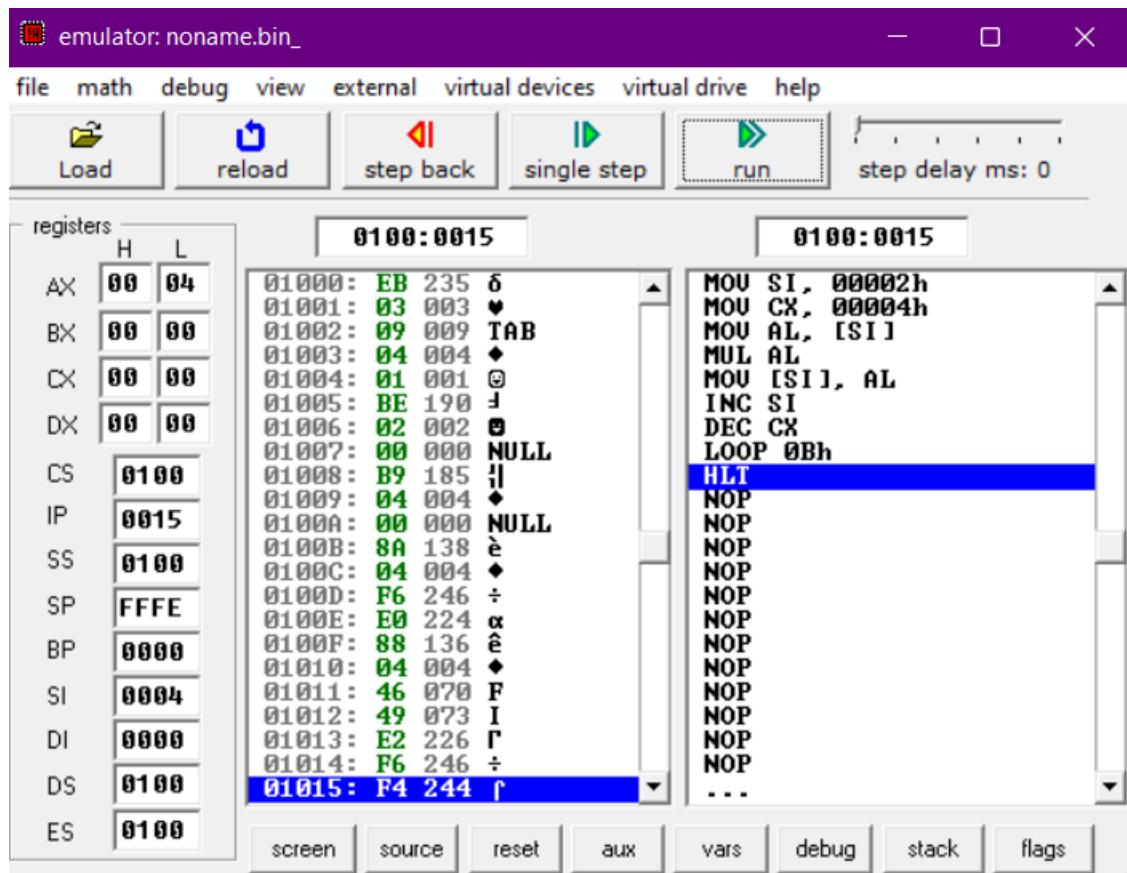
MOV [SI],AL

INC SI

DEC CX

LOOP L1

HLT



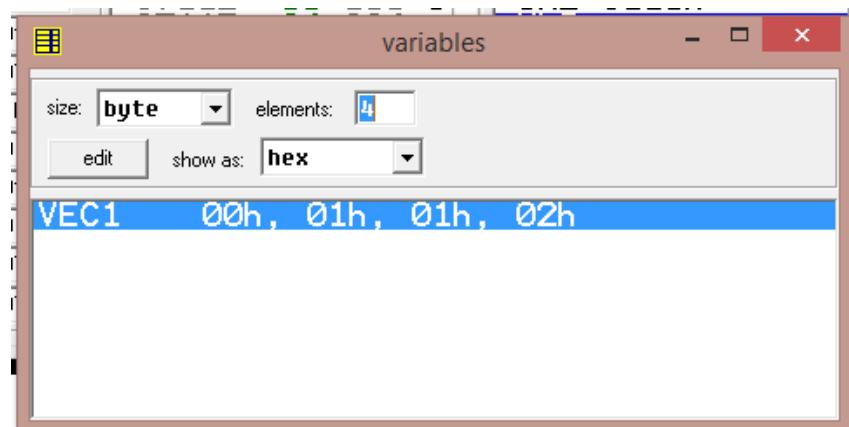
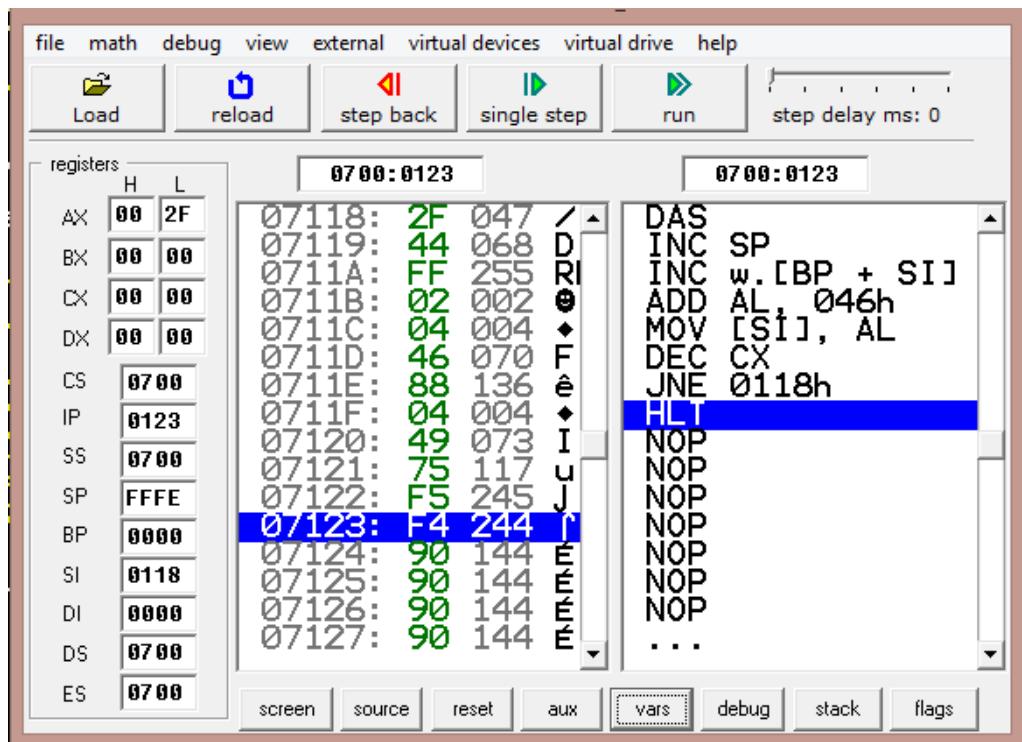
Output – All the elements of the series are squared and stored in vec1.

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
```

The screenshot shows a debugger interface with two main windows. The top window is titled "emulator: oddnumber.com_" and displays assembly code. The assembly code starts at address 0700:0106 and includes instructions like INC DI, INC SI, DEC CX, JNE 0215h, and HLT. The bottom window is titled "variables" and shows two variables: VEC1 (02h) and VEC2 (00h, 00h, 00h, 01h). The registers window on the left shows various CPU registers with their current values.

	H	L
AX	01	08
BX	00	02
CX	00	00
DX	00	00
CS	0700	
IP	0126	
SS	0700	
SP	FFFE	
BP	0000	
SI	0106	
DI	010A	
DS	0700	
ES	0700	

Registers:

	0700:0106	0700:0126
07106:	00 000 NI	INC DI
07107:	00 000 NI	INC SI
07108:	00 000 NI	DEC CX
07109:	01 001 ☺	JNE 0215h
0710A:	BE 190 ↴	HLT
0710B:	02 002 ☺	MOV b.[DI], 01h
0710C:	01 001 ☺	LOOP 0121h
0710D:	BF 191 ↴	RET
0710E:	06 006 ♠	NOP
0710F:	01 001 ☺	NOP
07110:	B9 185 ↴	NOP
07111:	05 005 ♣	NOP
07112:	00 000 NI	NOP
07113:	B3 179 ↴	NOP
07114:	02 002 ☺	NOP
07115:	8A 138 è	...

Instructions:

Variables window:

	size: byte	elements: 4
VEC1	02h	
VEC2	00h, 00h, 00h, 01h	

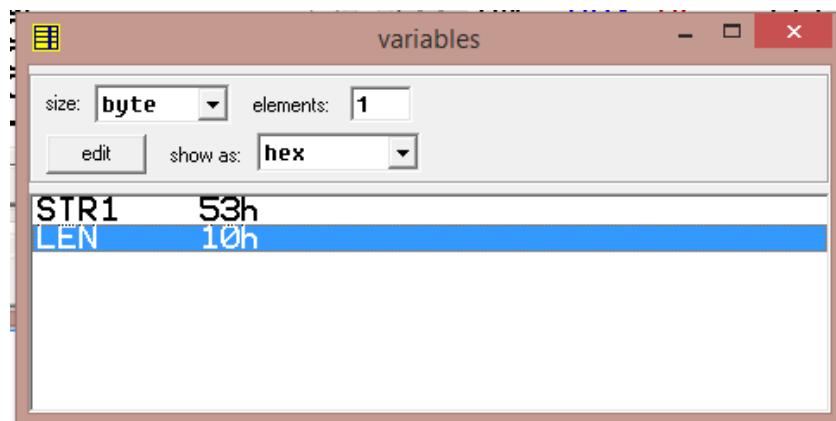
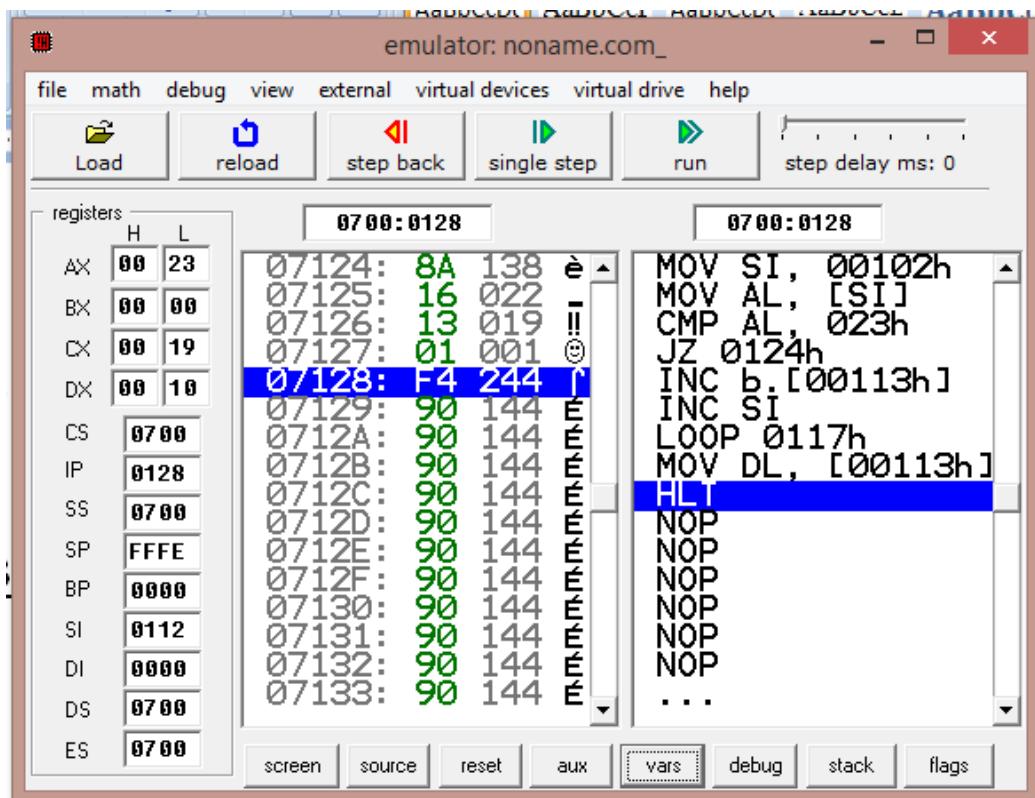
Output- From the variables window we can see that the 0 shows even, 1 shows odd numbers in vec2.

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
```



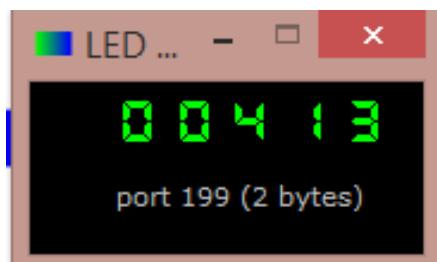
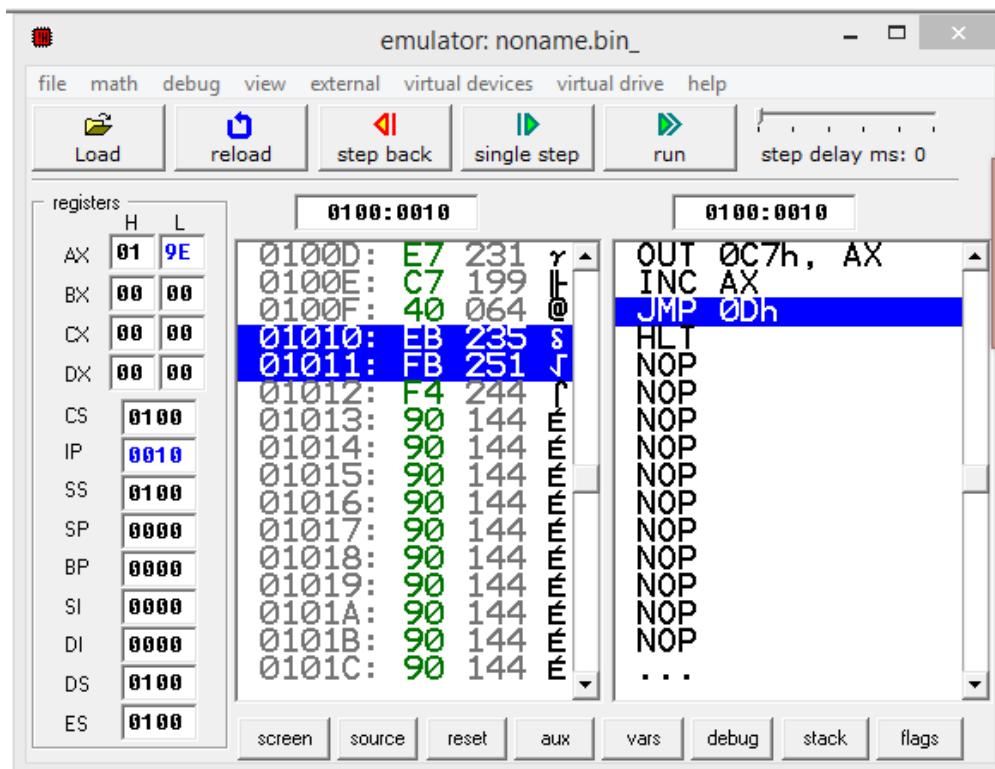
Output- From the variables window we can see the string length.

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
```



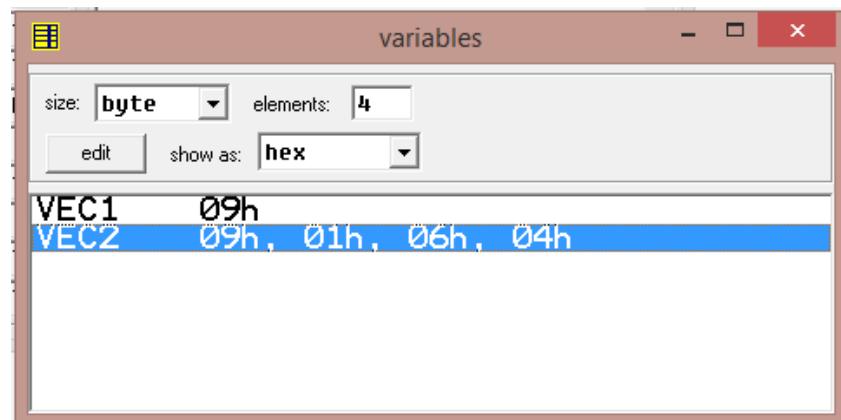
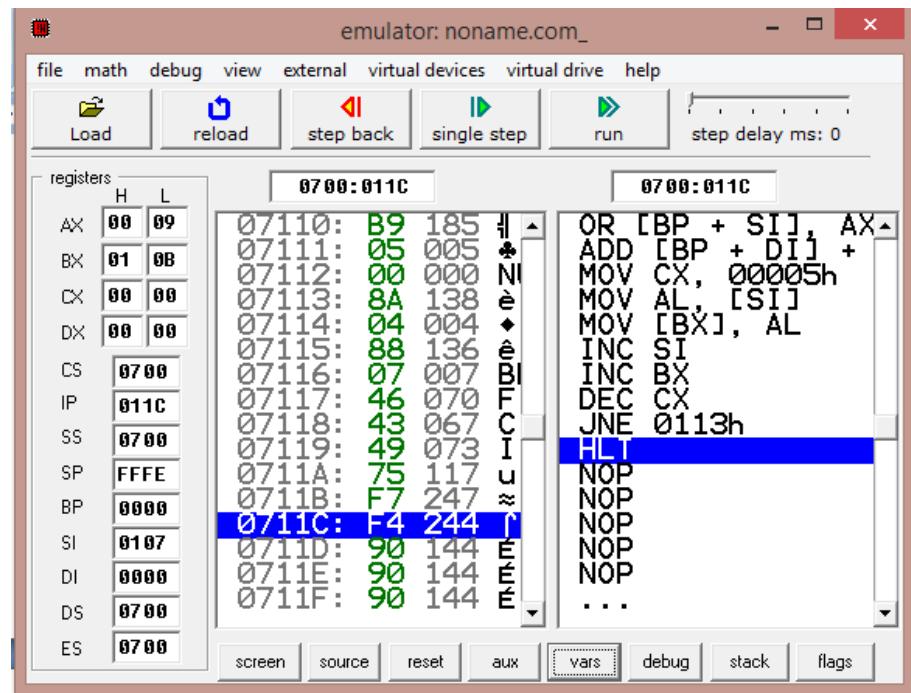
Output: The led display with data is shown below.

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.