FS19CO042

MP MANUALS

Exp 1 to 9

FS19CO042 First shift
Omkar Phansopkar

Experiment No 1

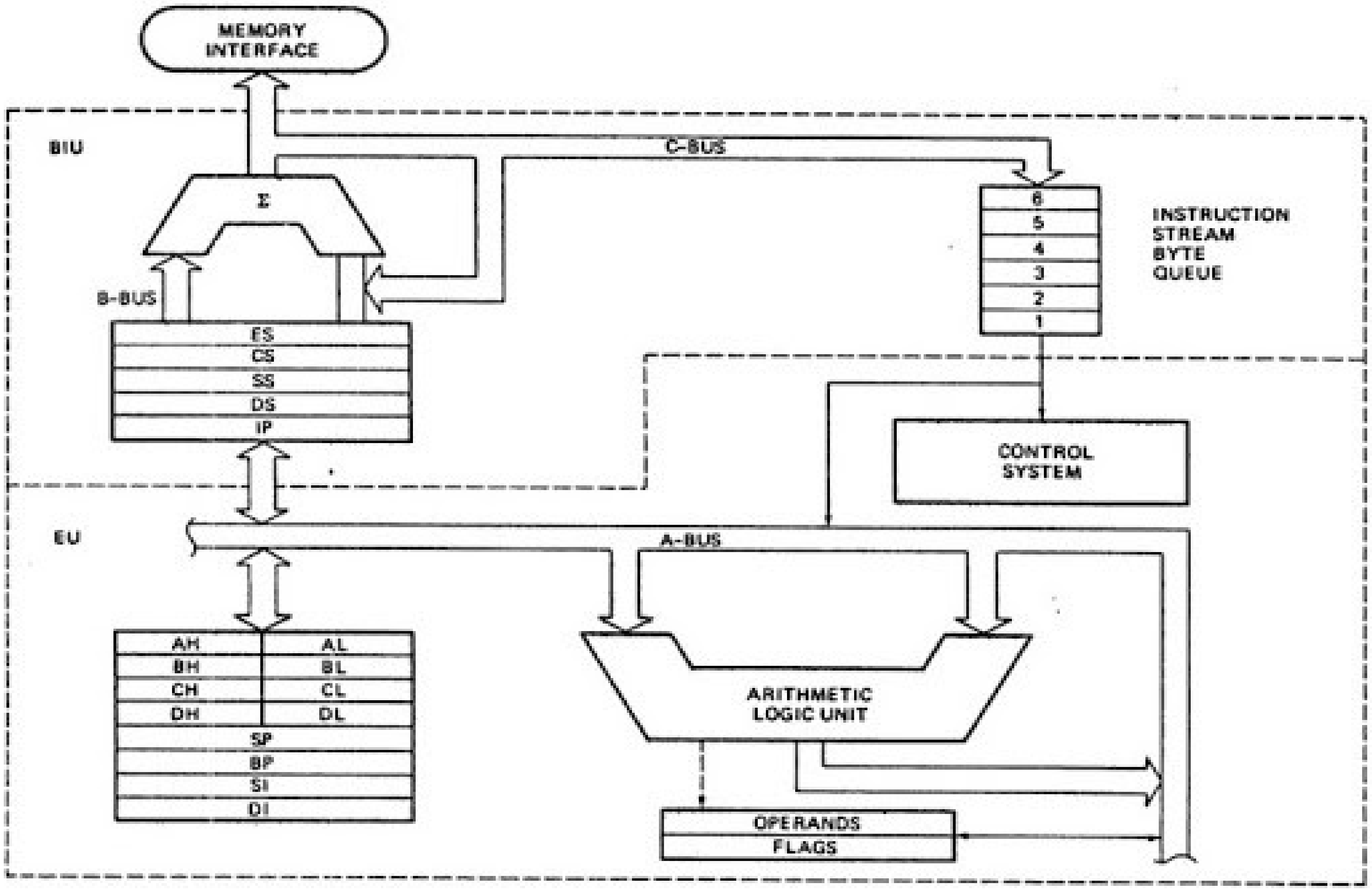
Aim : Understand 8086 Development Board and Simulation Software

Theory :

8086 Microprocessor is an enhanced version of 8085Microprocessor that was designed by Intel in 1976. It is a 16-bit Microprocessor having 20 address lines and16 data lines that provides up to 1MB storage. It consists of powerful instruction set, which provides operations like multiplication and division easily

EMU8086 - MICROPROCESSOR EMULATOR is a free emulator for multiple platforms. It provides its user with the ability to emulate old 8086 processors, which were used in Macintosh and Windows computers from the 1980s and early 1990s. It can emulate a large amount of software that was used on these microprocessors, but a savvy user can also program their own assembly code to run on it.

Block diagram of 8086:



BIU does the following:

- Fetch the instruction or data from memory.
- Write the data to the port.
- Read data from the port.
- Write the data to memory.

Execution unit does the following:

- To tell BIU where to fetch the instructions or data from.
- To decode the instructions.
- To execute the instructions.

Registers:

- General Purpose register (Usable by programmer to store values i.e. variables)

  These are usable as whole for storing 16 bit data or their sub divisions, eg. AH,AL etc to store 8 bit data individually

  1. **AX (AL,AH)** : Also used as accumulator in operations.
  2. **BX (BL, BH)** : Generally used to store base locations
  3. **CX (CL, CH)** : Acts as a counter in loops
  4. **DX (DL, DH)** : Generally used to contain I/0 port addresses

- Segment registers

  The pointers will always store some address or memory location. In **8086 Microprocessor**, they usually store the offset through which the actual address is calculated.

  1. CS (Code Segment): The user cannot modify the content of these registers. Only the microprocessor's compiler can do this.
  2. DS (Data Segment): The user can modify the content of the data segment.
  3. SS (Stack Segment): The SS is used to store the information about the memory segment.

     The operations of the SS are mainly Push and Pop.
  4. ES (Extra Segment): By default, the control of the compiler remains in the DS where the user can add and modify the instructions. If there is less space in that segment, then ES is used. ES is also used for copying purpose.

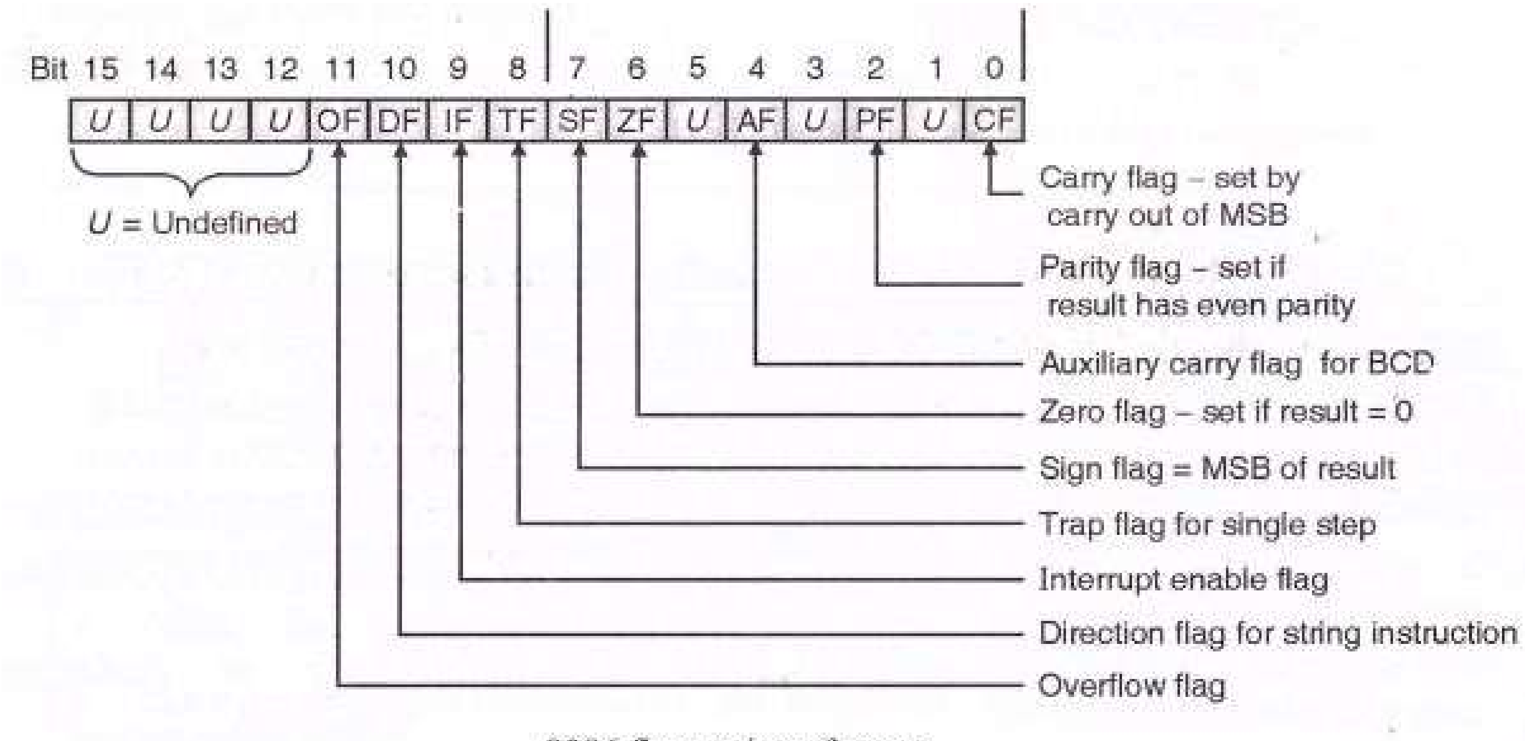
- Pointers and Index Registers

  The pointers will always store some address or memory location. In **8086 Microprocessor**, they usually store the offset through which the actual address is calculated.

  1. **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 (FFFE)H. It’s offset address relative to stack segment
  2. **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.
  3. **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. It’s offset is relative to data segment.
  4. **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.It’s offset is relative to extra segment.

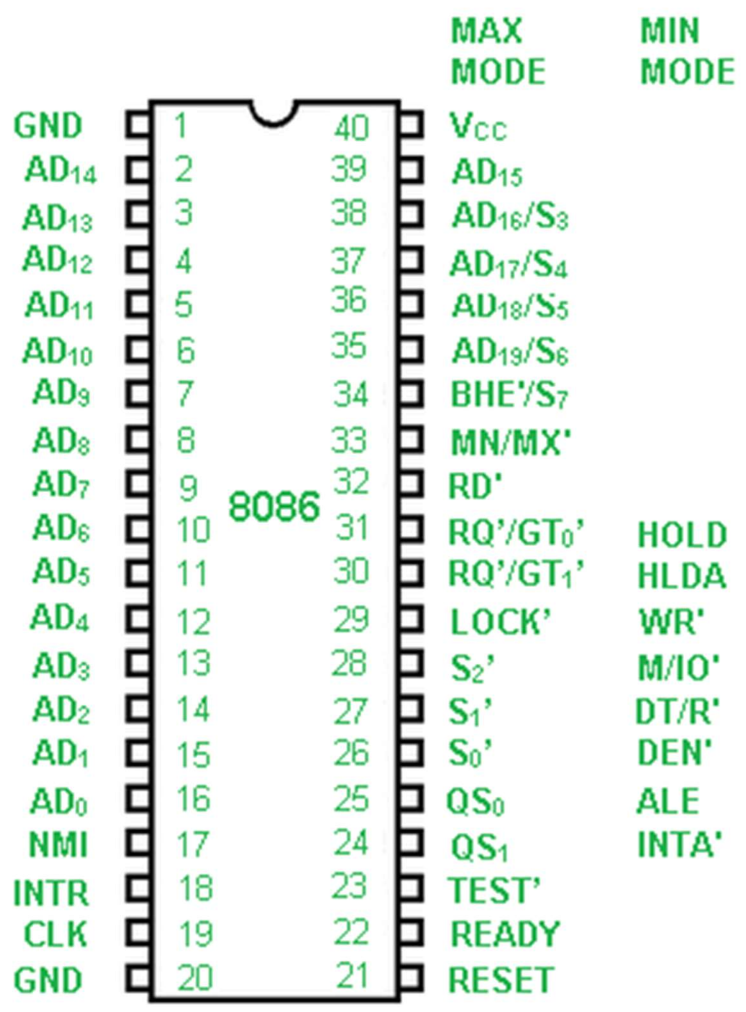
- <u>Flag or Status Registers</u>

  The Flag or Status register is a 16-bit register which contains 9 flags, and the remaining 7 bits are idle in this register. These flags tell about the status of the processor after any arithmetic or logical operation. IF the flag value is 1, the flag is set, and if it is 0, it is said to be reset.


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

8086 flag register format

Pin Diagram of 8086 Microprocessor:



Signals:

Power supply and frequency signals

It uses 5V DC supply at V_{CC} pin 40, and uses ground at V_{SS} pin 1 and 20 for its operation.

Clock signal

Clock signal is provided through Pin-19. It provides timing to the processor for operations. Its frequency is different for different versions, i.e. 5MHz, 8MHz and 10MHz.

Address/data bus

AD0-AD15. These are 16 address/data bus. AD0-AD7 carries low order byte data and AD8AD15 carries higher order byte data. During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.

Address/status bus

A16-A19/S3-S6. These are the 4 address/status buses. During the first clock cycle, it carries 4-bit address and later it carries status signals.

S7/BHE

BHE stands for Bus High Enable. It is available at pin 34 and used to indicate the transfer of data using data bus D8-D15. This signal is low during the first clock cycle, thereafter it is active.

Read (*$\overline {RD}$*)

It is available at pin 32 and is used to read signal for Read operation.

Ready

It is available at pin 22. It is an acknowledgement signal from I/O devices that data is transferred. It is an active high signal. When it is high, it indicates that the device is ready to transfer data. When it is low, it indicates wait state.

RESET

It is available at pin 21 and is used to restart the execution. It causes the processor to immediately terminate its present activity. This signal is active high for the first 4 clock cycles to RESET the microprocessor.

INTR

It is available at pin 18. It is an interrupt request signal, which is sampled during the last clock cycle of each instruction to determine if the processor considered this as an interrupt or not.

NMI

It stands for non-maskable interrupt and is available at pin 17. It is an edge triggered input, which causes an interrupt request to the microprocessor.

$\overline {TEST}$

This signal is like wait state and is available at pin 23. When this signal is high, then the processor has to wait for IDLE state, else the execution continues.

MN/*$\overline {MX}$*

It stands for Minimum/Maximum and is available at pin 33. It indicates what mode the processor is to operate in; when it is high, it works in the minimum mode and vice-versa.

INTA

It is an interrupt acknowledgement signal and id available at pin 24. When the microprocessor receives this signal, it acknowledges the interrupt.

ALE

It stands for address enable latch and is available at pin 25. A positive pulse is generated each time the processor begins any operation. This signal indicates the availability of a valid address on the address/data lines.

DEN

It stands for Data Enable and is available at pin 26. It is used to enable Trans receiver 8286. The trans receiver is a device used to separate data from the address/data bus.

DT/R

It stands for Data Transmit/Receive signal and is available at pin 27. It decides the direction of data flow through the trans receiver. When it is high, data is transmitted out and vice-a-versa.

M/IO

This signal is used to distinguish between memory and I/O operations. When it is high, it indicates I/O operation and when it is low indicates the memory operation. It is available at pin 28.

WR

It stands for write signal and is available at pin 29. It is used to write the data into the memory or the output device depending on the status of M/IO signal.

HLDA

It stands for Hold Acknowledgement signal and is available at pin 30. This signal acknowledges the HOLD signal.

HOLD

This signal indicates to the processor that external devices are requesting to access the address/data buses. It is available at pin 31.

QS_1 and QS_0

These are queue status signals and are available at pin 24 and 25. These signals provide the status of instruction queue. Their conditions are shown in the following table −

QS_0	QS_1	Status
0	0	No operation

0	1	First byte of opcode from the queue
1	0	Empty the queue
1	1	Subsequent byte from the queue

S_0, S_1, S_2

These are the status signals that provide the status of operation, which is used by the Bus Controller 8288 to generate memory & I/O control signals. These are available at pin 26, 27, and 28. Following is the table showing their status –

S_2	S_1	S_0	Status
0	0	0	Interrupt acknowledgement
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Passive

LOCK

When this signal is active, it indicates to the other processors not to ask the CPU to leave the system bus. It is activated using the LOCK prefix on any instruction and is available at pin 29.

RQ/GT_1 and RQ/GT_0

These are the Request/Grant signals used by the other processors requesting the CPU to release the system bus. When the signal is received by CPU, then it sends acknowledgment. RQ/GT_0 has a higher priority than RQ/GT_1.

Instruction Set Of 8086:

1. Data Transfer Instructions

These instructions are used to transfer the data from the source operand to the destination operand. List of data transfer instructions:

Instruction to transfer a word

- **MOV** – Used to copy the byte or word from the provided source to the provided destination.
- **PPUSH** – Used to put a word at the top of the stack.
- **POP** – Used to get a word from the top of the stack to the provided location.
- **PUSHA** – Used to put all the registers into the stack.
- **POPA** – Used to get words from the stack to all registers.
- **XCHG** – Used to exchange the data from two locations.
- **XLAT** – Used to translate a byte in AL using a table in the memory.

Instructions to transfer the address

- **LEA** – Used to load the address of operand into the provided register.
- **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 for input and output port transfer

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

Instructions to transfer flag registers

- **LAHF** − Used to load AH with the low byte of the flag register.
- **SAHF** − Used to store AH register to low byte of the flag register.
- **PUSHF** − Used to copy the flag register at the top of the stack.
- **POPF** − Used to copy a word at the top of the stack to the flag register.

2. Arithmetic Instructions

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

Following is the list of instructions under this group −

Instructions to perform addition

- **ADD** − Used to add the provided byte to byte/word to word.
- **ADC** − Used to add with carry.
- **INC** − Used to increment the provided byte/word by 1.
- **AAA** − Used to adjust ASCII after addition.
- **DAA** − Used to adjust the decimal after the addition/subtraction operation.

Instructions to perform subtraction

- **SUB** − Used to subtract the byte from byte/word from word.
- **SBB** − Used to perform subtraction with borrow.
- **DEC** − Used to decrement the provided byte/word by 1.
- **NPG** − Used to negate each bit of the provided byte/word and add 1/2's complement.
- **CMP** − Used to compare 2 provided byte/word.
- **AAS** − Used to adjust ASCII codes after subtraction.
- **DAS** − Used to adjust decimal after subtraction.

Instruction to perform multiplication

- **MUL** − Used to multiply unsigned byte by byte/word by word.
- **IMUL** − Used to multiply signed byte by byte/word by word.
- **AAM** − Used to adjust ASCII codes after multiplication.

Instructions to perform division

- **DIV** − Used to divide the unsigned word by byte or unsigned double word by word.
- **IDIV** − Used to divide the signed word by byte or signed double word by word.
- **AAD** − Used to adjust ASCII codes after division.
- **CBW** − Used to fill the upper byte of the word with the copies of sign bit of the lower byte.
- **CWD** − Used to fill the upper word of the double word with the sign bit of the lower word.

3. Bit Manipulation Instructions

These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc.

Following is the list of instructions under this group −

Instructions to perform logical operation

- **NOT** − Used to invert each bit of a byte or word.
- **AND** − Used for adding each bit in a byte/word with the corresponding bit in another byte/word.
- **OR** − Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.
- **XOR** − Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.
- **TEST** − Used to add operands to update flags, without affecting operands.

Instructions to perform shift operations

- **SHL/SAL** − Used to shift bits of a byte/word towards left and put zero(S) in LSBs.

- **SHR** − Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.
- **SAR** − Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

Instructions to perform rotate operations

- **ROL** − Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF].
- **ROR** − Used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF].
- **RCR** − Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB.
- **RCL** − Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB.

4. String Instructions

String is a group of bytes/words and their memory is always allocated in a sequential order.

Following is the list of instructions under this group −

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

5. Program Execution Transfer Instructions (Branch and Loop Instructions)

These instructions are used to transfer/branch the instructions during an execution. It includes the following instructions −

Instructions to transfer the instruction during an execution without any condition −

- **CALL** − Used to call a procedure and save their return address to the stack.
- **RET** − Used to return from the procedure to the main program.
- **JMP** − Used to jump to the provided address to proceed to the next instruction.

Instructions to transfer the instruction during an execution with some conditions −

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

6. Processor Control Instructions

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

Following are the instructions under this group −

- **STC** − Used to set carry flag CF to 1

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

7. Iteration Control Instructions

These instructions are used to execute the given instructions for number of times. Following is the list of instructions under this group −

- **LOOP** − Used to loop a group of instructions until the condition satisfies, i.e., CX = 0
- **LOOPE/LOOPZ** − Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0
- **LOOPNE/LOOPNZ** − Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0
- **JCXZ** − Used to jump to the provided address if CX = 0

8. Interrupt Instructions

These instructions are used to call the interrupt during program execution.

- **INT** − Used to interrupt the program during execution and calling service specified.
- **INTO** − Used to interrupt the program during execution if OF = 1
- **IRET** − Used to return from interrupt service to the main program

Conclusion: Thus, we understood working, construction and use of 8086 microprocessor development board and stimulation software

Experiment No 2

Aim : 8086 Assembly language programming for Addition and subtraction of two 16 bit numbers

Requirement : Emu8086 (Assembler and Microprocessor emulator)

Theory :

1. **Data transfer instructions**

   Data transfer instructions are used to transfer data from source operand to destination operand

   MOV instruction

   MOV instruction moves data from one location to another. It also has the widest variety of parameters; so it the assembler programmer can use MOV effectively, the rest of the commands are easier to understand.

   Format: `MOV destination,source`

   Eg. MOV AX,6

2. **Arithmetic instructions**

   a. **ADD instruction**

   ADD adds the contents of the source to the destination. The source and destination may be either bytes or words but both operands must be the same type or the assembler will generate an error.

   If the sum of the two numbers cannot fit in the destination, an extra bit is required and this is signalled by the ADD operation setting the carry flags (CF) to 1. If the sum fits without spillage, CF=0. Other registers can be affected by addition operations as well; ZF=0 if the sum is zero, SF=1 if the sum is negative, etc. The logic of the basic addition command is:

   Format: **ADD** `destination,source`

   Logically, destination = destination + source

   Eg. ADD AX,BX

   b. **SUB instruction**

   SUB subtracts the source value from the destination. Operation is almost identical to addition, except that the CF flag is used as a borrow in the case of the SBB (subtract with borrow) instruction.

   Format: **SUB** `destination,source`

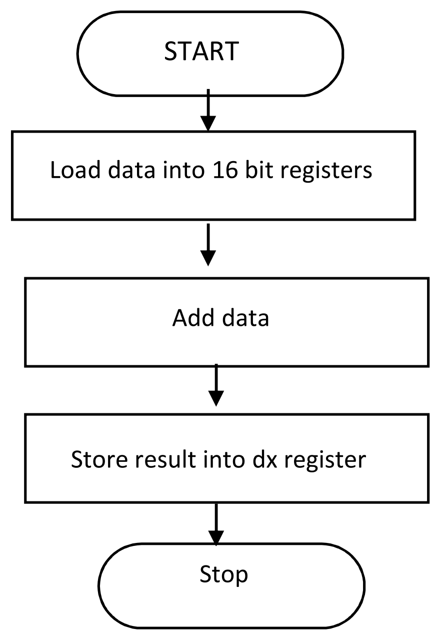
   Logically, it is destination = destination - source

   destination = destination - source - carry (if required)

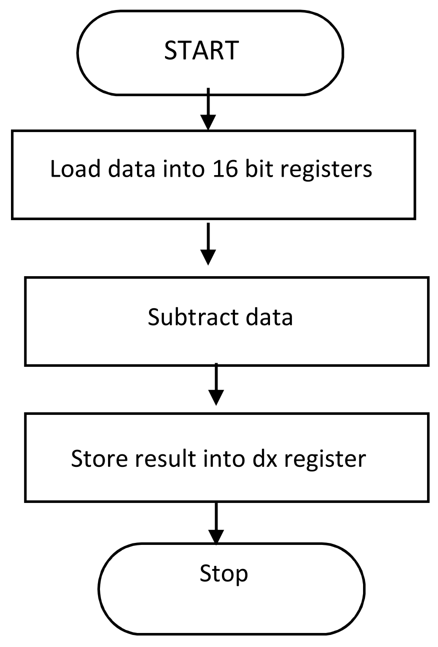
   Eg. SUB AX,BX

Flowchart :

Flowchart for 16 bit numbers addition


START
Load data into 16 bit registers
Add data
Store result into dx register
Stop

Flowchart for 16 bit numbers subtraction


START
Load data into 16 bit registers
Subtract data
Store result into dx register
Stop

16 Bit Addition :

Program:

```
data segment
   a dw 4121h
   b dw 1742h
   c dw ?
   data ends
   code segment
      assume cs:code,ds:data
      start:
      mov ax,data
      mov ds,ax
      mov ax,a
      mov bx,b
      add ax,bx
      mov cx,ax
      mov c,cx
      int 3
      code ends
   end start
```

MEMORY LOCATION	OP-CODE	LABEL	MNEMOIC
O7110H	B8		mov ax,data
O7111H	10		
O7112H	07		
07113H	8E		mov ds,ax
07114H	D8		
07115H	A1		mov ax,a
07116H	00		
07117H	00		
			mov bx,b
07118H	8B		
07119H	1E		
0711AH	02		
0711BH	00		
0711CH	03		add ax,bx
0711DH	C3		
0711EH	8B		mov cx,ax
0711FH	C8		
07120H	89		mov c,cx
07121H	0E		
07122H	04		
07123H	00		
07124H	CC		int 3

16 Bit Subtraction :

Program:

```
data segment
   a dw 21A6h
   b dw 1022h
   c dw ?
   data ends
   code segment
      assume cs:code,ds:data
      start:
      mov ax,data
      mov ds,ax
      mov ax,a
      mov bx,b
      sub ax,bx
      mov cx,ax
      mov c,cx
      int 3
      code ends
   end start
```

MEMORY LOCATION	OP-CODE	LABEL	MNEMOIC
07110H	B8		mov ax,data
07111H	10		
07112H	07		

07113H	8E		**mov ds,ax**
07114H	D8		
07115H	A1		**mov ax,a**
07116H	00		
07117H	00		
07118H	8B		**mov bx,b**
07119H	1E		
0711AH	02		
0711BH	00		
0711CH	2B		**sub ax,bx**
0711DH	C3		
0711EH	8B		**mov cx,ax**
0711FH	C8		
07120H	89		**mov c,cx**
07121H	0E		
07122H	04		
07123H	00		
07124H	CC		**int 3**

Result :

Output OF 16 BIT ADDITION :

INPUT		**OUTPUT**	
REGISTER	**DATA**	**REGISTER**	**DATA**
AX	**4121**	**CX**	**5863**
BX	**1742**		

Affected flags: Parity flag

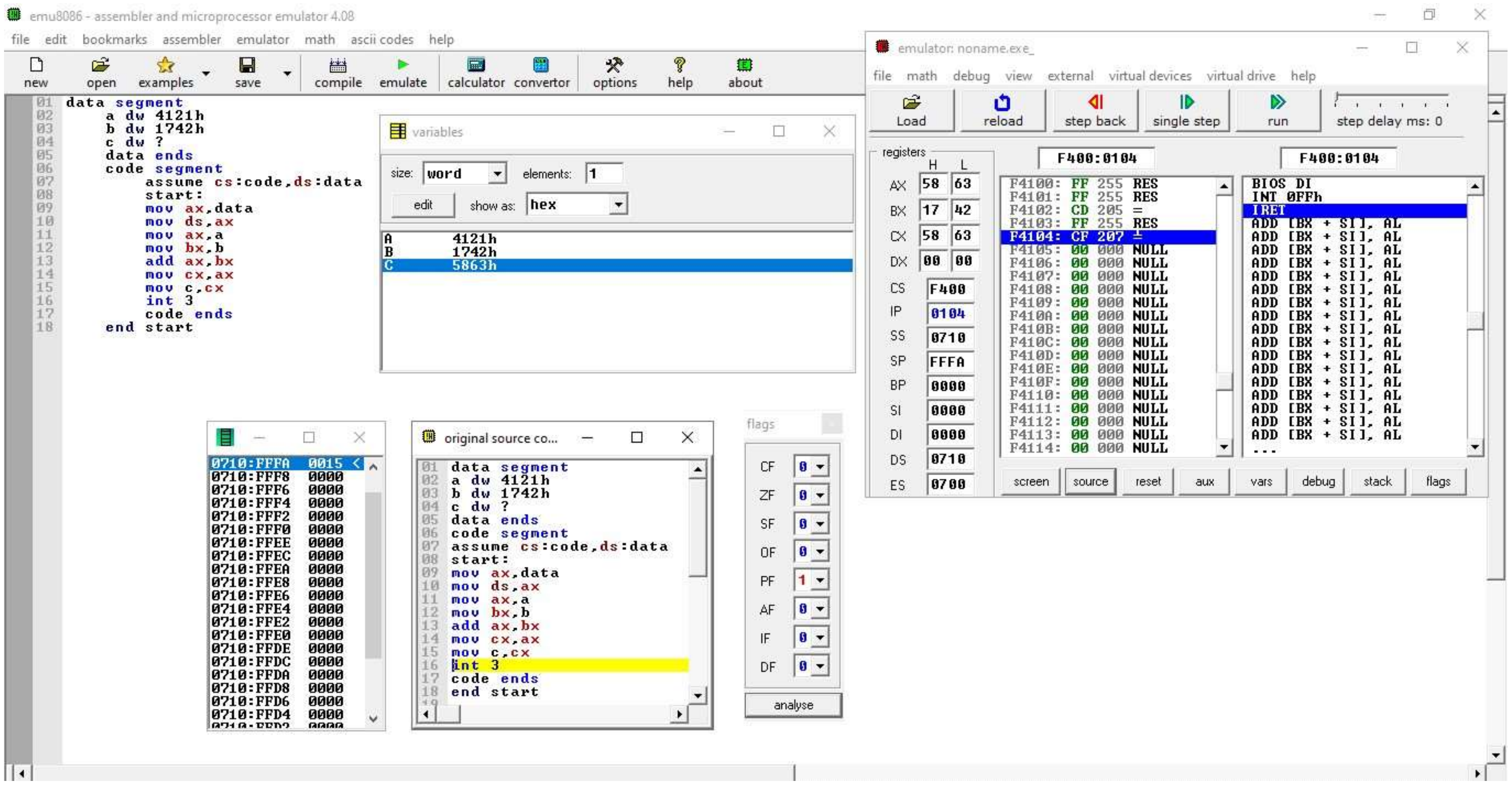
Output OF 16 BIT SUNTRACTION :

INPUT		**OUTPUT**	
REGISTER	**DATA**	**REGISTER**	**DATA**
AX	**21A6**	**CX**	**1184**
BX	**1022**		

Affected flags: Parity flag

SCREENSHOT :

16 Bit Addition :



```
data segment
    a dw 4121h
    b dw 1742h
    c dw ?
    data ends
    code segment
        assume cs:code,ds:data
        start:
        mov ax,data
        mov ds,ax
        mov ax,a
        mov bx,b
        add ax,bx
        mov cx,ax
        mov c,cx
        int 3
        code ends
    end start
```

variables — size: word, elements: 1, show as: hex

```
A        4121h
B        1742h
C        5863h
```

```
0710:FFFA  0015
0710:FFF8  0000
0710:FFF6  0000
0710:FFF4  0000
0710:FFF2  0000
0710:FFF0  0000
0710:FFEE  0000
0710:FFEC  0000
0710:FFEA  0000
0710:FFE8  0000
0710:FFE6  0000
0710:FFE4  0000
0710:FFE2  0000
0710:FFE0  0000
0710:FFDE  0000
0710:FFDC  0000
0710:FFDA  0000
0710:FFD8  0000
0710:FFD6  0000
0710:FFD4  0000
```

flags: CF 0, ZF 0, SF 0, OF 0, PF 1, AF 0, IF 0, DF 0

registers:

```
     H   L
AX   58  63
BX   17  42
CX   58  63
DX   00  00
CS   F400
IP   0104
SS   0710
SP   FFFA
BP   0000
SI   0000
DI   0000
DS   0710
ES   0700
```

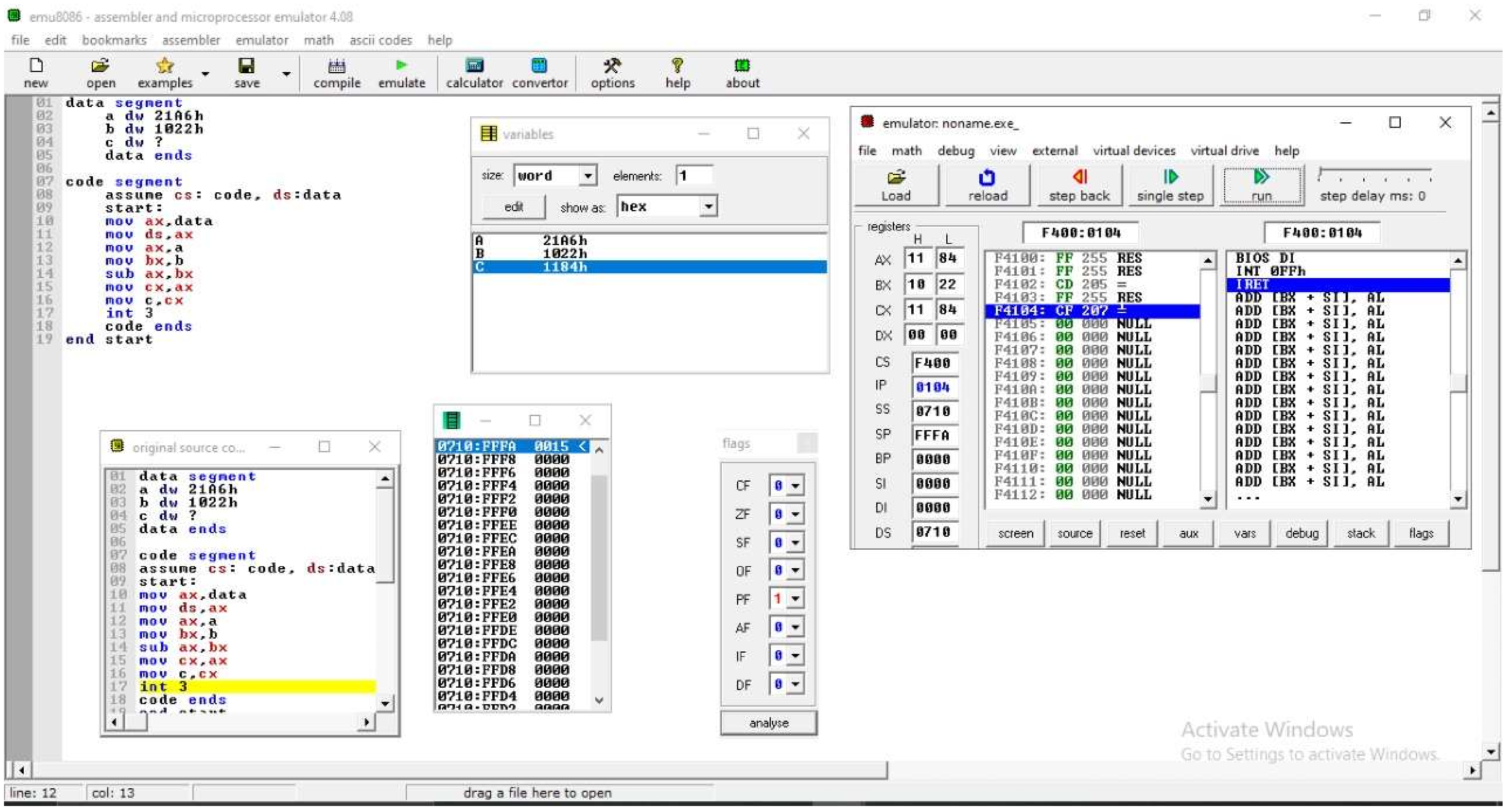
```
F400:0104

F4100: FF 255 RES
F4101: FF 255 RES
F4102: CD 205 =
F4103: FF 255 RES
F4104: CF 207 ±
F4105: 00 000 NULL
F4106: 00 000 NULL
F4107: 00 000 NULL
F4108: 00 000 NULL
F4109: 00 000 NULL
F410A: 00 000 NULL
F410B: 00 000 NULL
F410C: 00 000 NULL
F410D: 00 000 NULL
F410E: 00 000 NULL
F410F: 00 000 NULL
F4110: 00 000 NULL
F4111: 00 000 NULL
F4112: 00 000 NULL
F4113: 00 000 NULL
F4114: 00 000 NULL
```

```
F400:0104

BIOS DI
INT 0FFh
IRET
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
...
```

16 bit subtraction :



```
data segment
    a dw 21A6h
    b dw 1022h
    c dw ?
    data ends

code segment
    assume cs: code, ds:data
    start:
    mov ax,data
    mov ds,ax
    mov ax,a
    mov bx,b
    sub ax,bx
    mov cx,ax
    mov c,cx
    int 3
    code ends
end start
```

variables — size: word, elements: 1, show as: hex

```
A        21A6h
B        1022h
C        1184h
```

```
0710:FFFA  0015
0710:FFF8  0000
0710:FFF6  0000
0710:FFF4  0000
0710:FFF2  0000
0710:FFF0  0000
0710:FFEE  0000
0710:FFEC  0000
0710:FFEA  0000
0710:FFE8  0000
0710:FFE6  0000
0710:FFE4  0000
0710:FFE2  0000
0710:FFE0  0000
0710:FFDE  0000
0710:FFDC  0000
0710:FFDA  0000
0710:FFD8  0000
0710:FFD6  0000
0710:FFD4  0000
```

flags: CF 0, ZF 0, SF 0, OF 0, PF 1, AF 0, IF 0, DF 0

registers:

```
     H   L
AX   11  84
BX   10  22
CX   11  84
DX   00  00
CS   F400
IP   0104
SS   0710
SP   FFFA
BP   0000
SI   0000
DI   0000
DS   0710
```

```
F400:0104

F4100: FF 255 RES
F4101: FF 255 RES
F4102: CD 205 =
F4103: FF 255 RES
F4104: CF 207 ±
F4105: 00 000 NULL
F4106: 00 000 NULL
F4107: 00 000 NULL
F4108: 00 000 NULL
F4109: 00 000 NULL
F410A: 00 000 NULL
F410B: 00 000 NULL
F410C: 00 000 NULL
F410D: 00 000 NULL
F410E: 00 000 NULL
F410F: 00 000 NULL
F4110: 00 000 NULL
F4111: 00 000 NULL
F4112: 00 000 NULL
```

```
F400:0104

BIOS DI
INT 0FFh
IRET
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
...
```

line: 12 col: 13 drag a file here to open

CONCLUSION : Thus ,we performed 16 bit addition and 16 bit subtraction using AX ,BX and CX register.

Experiment No 3

Aim : 8086 Assembly language programming for Addition of series of 16 bit numbers

Requirement : Emu8086 (Assembler and Microprocessor emulator)

Theory :

1. **Data transfer instructions**

   Data transfer instructions are used to transfer data from source operand to destination operand

a. MOV instruction

MOV instruction moves data from one location to another. It also has the widest variety of parameters; so it the assembler programmer can use MOV effectively, the rest of the commands are easier to understand.

Format: `MOV destination,source`

Eg. MOV AX,6

b. LEA instruction

LEA instruction determines the offset of the variable or memory location named as the source and puts this offset in the indicated 16-bit register. LEA does not affect any flag.

Syntax : LEA Register, Source

Example : LEA AX,14

2. **Arithmetic instructions**

a. ADD instruction

ADD adds the contents of the source to the destination. The source and destination may be either bytes or words but both operands must be the same type or the assembler will generate an error.

If the sum of the two numbers cannot fit in the destination, an extra bit is required and this is signalled by the ADD operation setting the carry flags (CF) to 1. If the sum fits without spillage, CF=0. Other registers can be affected by addition operations as well; ZF=0 if the sum is zero, SF=1 if the sum is negative, etc. The logic of the basic addition command is:

Format: **ADD** `destination,source`

Logically, destination = destination + source

Eg. ADD AX,BX

b. DEC instruction

DEC instruction subtracts 1 from the destination word or byte. The destination can be a register or a memory location. AF, OF, SF, PF, and ZF are updated, but CF is not affected. This means that if an 8-bit destination containing 00H or a 16-bit destination containing 0000H is decremented, the result will be FFH or FFFFH with no carry (borrow).

Syntax : DEC Destination

Example : DEC AX

3. **Logical instructions**

CMP instruction

This instruction comes under **Logical Instruction.** This instruction compares a byte / word in the specified source with a byte / word in the specified destination. The source can be an immediate number, a register, or a memory location. The destination
can be a register or a memory location. However, the source and the destination cannot both be memory
locations. The comparison is actually done by subtracting the source byte or word from the destination

byte or word. The source and the destination are not changed, but the flags are set to indicate the results of the comparison. AF, OF, SF, ZF, PF, and CF are updated by the CMP instruction. For the instruction CMP CX, BX, the values of CF, ZF, and SF will be as follows:

Syntax : CMP Destination, Source

Example : CMP AL, 01H

4. Transfer Of Control Instruction

JNZ instruction

This instruction comes under **Transfer Of Control Instruction.** This instruction is usually used after a Compare instruction. If the zero flag is 0, then this instruction will cause a jump to the label given in the instruction.

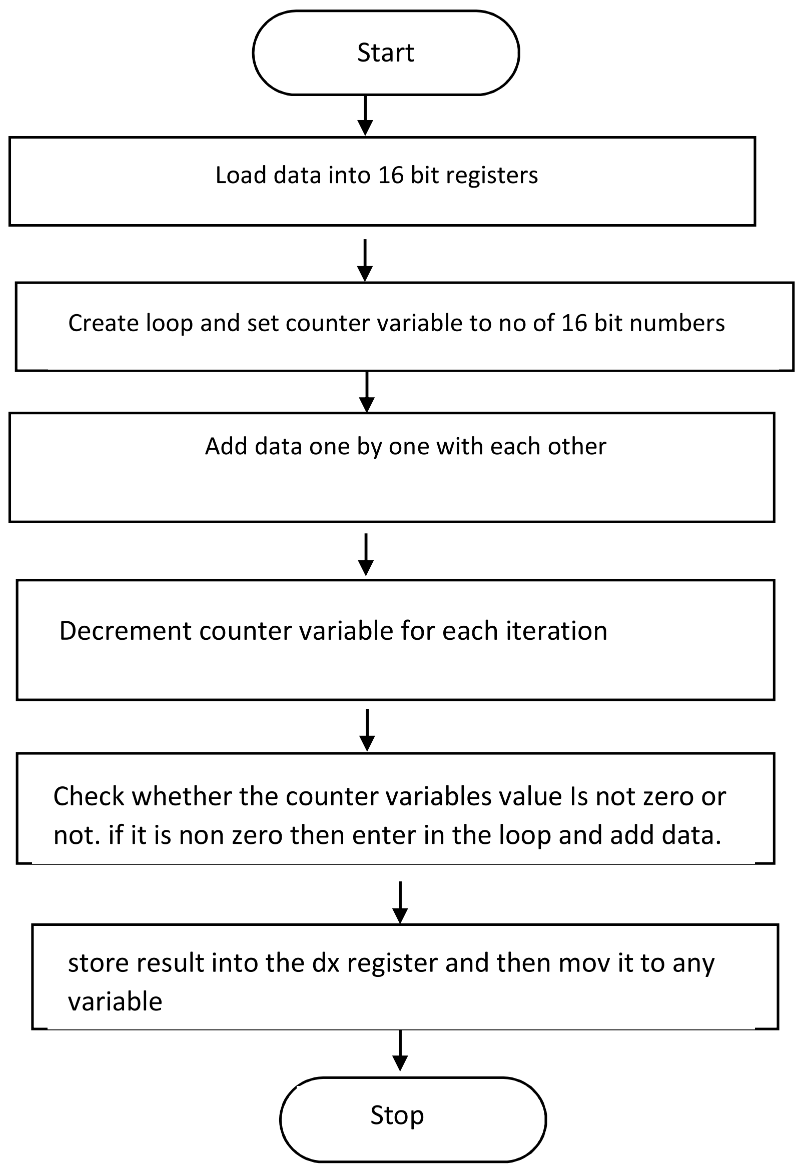
Example :

ADD AX, 0002H

DEC BX

JNZ NEXT

Flowchart :


Start
Load data into 16 bit registers
Create loop and set counter variable to no of 16 bit numbers
Add data one by one with each other
Decrement counter variable for each iteration
Check whether the counter variables value Is not zero or not. if it is non zero then enter in the loop and add data.
store result into the dx register and then mov it to any variable
Stop

Program :

```
data segment
   a dw 4200h, 5300h, 1600h, 8000h, 1900h
   b dw ?
   data ends
code segment
      assume cs:code,ds:data
      start:
      mov ax,data
      mov ds,ax
      mov cl,5
      lea bx,a
      mov ax,00

      OP: add ax,word ptr[bx]
         add bx,02
         dec cl
         cmp cl,00
         jnz OP

      mov b,ax
      int 3
      code ends
end start
```

MEMORY LOCATION	**OP-CODE**	**LABEL**	**MNEMOIC**
07110h	**B8**		**mov ax,data**
07111h			
07112h			
07113h	**8E**		**mov ds,ax**
07114h	**D8**		
07115h	**B1**		**mov cl,5**
07116h	**05**		
07117h	**BB**		**lea bx,a**
07118h	**00**		
07119h	**00**		
0711Ah	**B8**		**mov ax,00**
0711Bh	**00**		
0711Ch	**00**		
0711Dh	**03**	**OP:**	**add ax,word ptr[bx]**
0711Eh	**07**		
0711Fh	**83**		**add bx,02**
07120h	**C3**		
07121h	**02**		
07122h	**FE**		**dec cl**
07123h	**C9**		
07124h	**80**		**cmp cl,00**
07125h	**F9**		
07126h	**00**		

07127h 07128h	75 F4		jnz OP
07129h 0712Ah 0712Bh	A3 0A 00		mov b,ax
0712Ch	CC		int 3

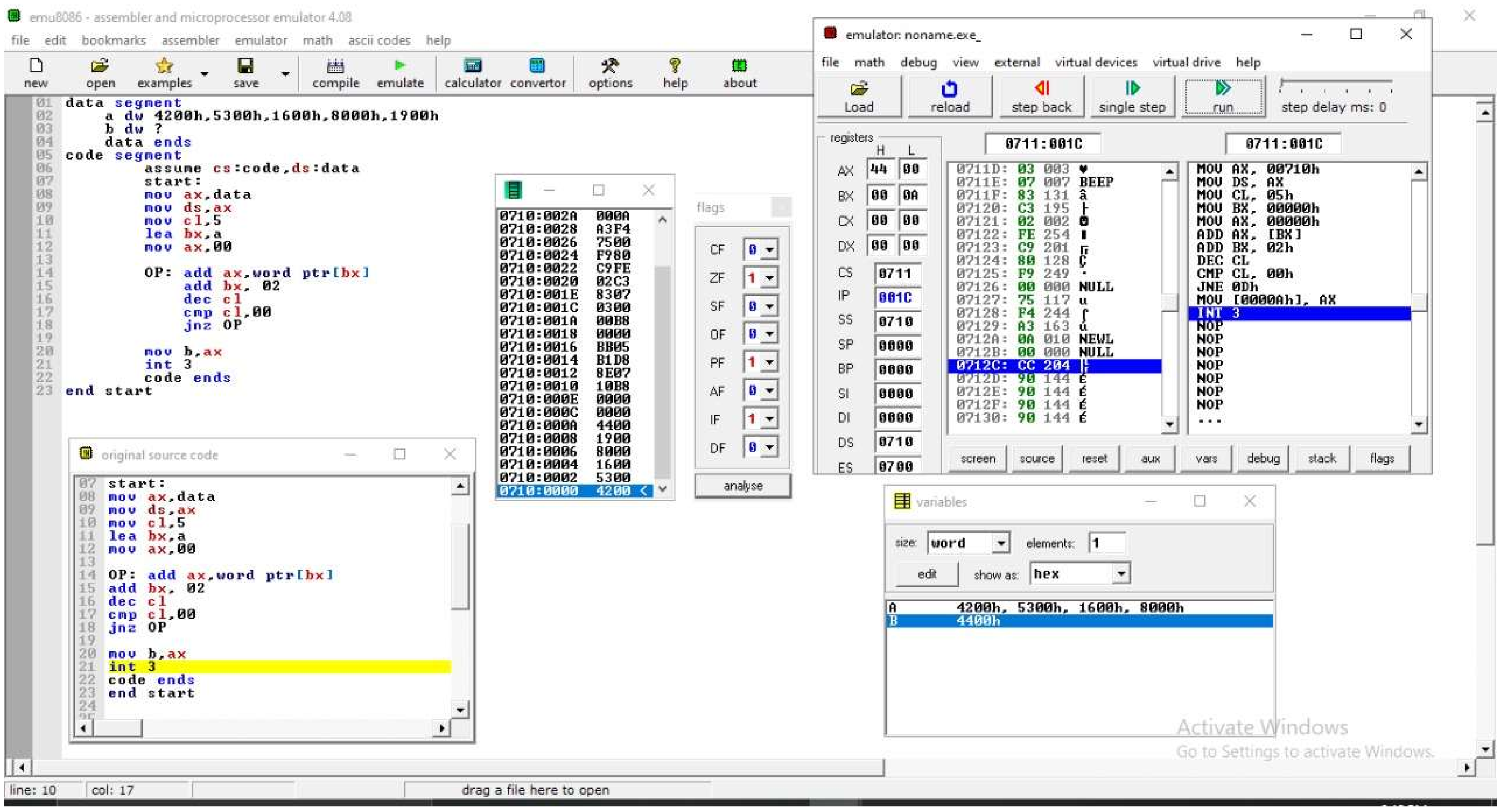
Result :

Output OF 16 BIT ADDITION :

INPUT		OUTPUT	
REGISTER	**DATA**	**REGISTER**	**DATA**
AX	4200h, 5300h, 1600h, 8000h, 1900h	**AX**	**4400h**

Affected flags: Parity flag, Zero flag, Interrupt enable flag

SCREENSHOT :



emu8086 - assembler and microprocessor emulator 4.08
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
emulator: noname.exe_
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
screen source reset aux vars debug stack flags
analyse
line: 10 col: 17 drag a file here to open

```
data segment
    a dw 4200h,5300h,1600h,8000h,1900h
    b dw ?
    data ends
code segment
        assume cs:code,ds:data
        start:
        mov ax,data
        mov ds,ax
        mov cl,5
        lea bx,a
        mov ax,00

        OP: add ax,word ptr[bx]
            add bx, 02
            dec cl
            cmp cl,00
            jnz OP

        mov b,ax
        int 3
        code ends
end start
```

original source code

```
start:
mov ax,data
mov ds,ax
mov cl,5
lea bx,a
mov ax,00

OP: add ax,word ptr[bx]
add bx, 02
dec cl
cmp cl,00
jnz OP

mov b,ax
int 3
code ends
end start
```

```
0710:002A  000A
0710:0028  A3F4
0710:0026  7500
0710:0024  F980
0710:0022  C9FE
0710:0020  02C3
0710:001E  8307
0710:001C  0300
0710:001A  00B8
0710:0018  0000
0710:0016  BB05
0710:0014  B1D8
0710:0012  8E07
0710:0010  10B8
0710:000E  0000
0710:000C  0000
0710:000A  4400
0710:0008  1900
0710:0006  8000
0710:0004  1600
0710:0002  5300
0710:0000  4200 <
```

flags

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

registers

	H	L
AX	44	00
BX	00	0A
CX	00	00
DX	00	00
CS	0711	
IP	001C	
SS	0710	
SP	0000	
BP	0000	
SI	0000	
DI	0000	
DS	0710	
ES	0700	

```
0711:001C
0711D: 03 003 ♥
0711E: 07 007 BEEP
0711F: 83 131 â
07120: C3 195 ├
07121: 02 002 ☻
07122: FE 254 ■
07123: C9 201 ╔
07124: 80 128 Ç
07125: F9 249 ·
07126: 00 000 NULL
07127: 75 117 u
07128: F4 244 ⌠
07129: A3 163 ú
0712A: 0A 010 NEWL
0712B: 00 000 NULL
0712C: CC 204 ╠
0712D: 90 144 É
0712E: 90 144 É
0712F: 90 144 É
07130: 90 144 É
```

```
0711:001C
MOV AX, 00710h
MOV DS, AX
MOV CL, 05h
MOV BX, 00000h
MOV AX, 00000h
ADD AX, [BX]
ADD BX, 02h
DEC CL
CMP CL, 00h
JNE 0Dh
MOV [0000Ah], AX
INT 3
NOP
NOP
NOP
NOP
NOP
NOP
NOP
...
```

variables

size: word elements: 1
show as: hex

```
A      4200h, 5300h, 1600h, 8000h
B      4400h
```

Activate Windows
Go to Settings to activate Windows.

Conclusion: Thus, we understood and performed addition of series of 16 bit numbers.

Experiment No 4

Aim : 8086 Assembly language programming for multibyte addition of two numbers

Requirement : Emu8086 (Assembler and Microprocessor emulator)

Theory :

1. **Data transfer instructions**

   Data transfer instructions are used to transfer data from source operand to destination operand

   a. **MOV instruction**

   MOV instruction moves data from one location to another. It also has the widest variety of parameters; so it the assembler programmer can use MOV effectively, the rest of the commands are easier to understand.

   Format: `MOV destination,source`

   Eg. MOV AX,6

2. **Arithmetic instructions**

   a. **ADD instruction**

   ADD adds the contents of the source to the destination. The source and destination may be either bytes or words but both operands must be the same type or the assembler will generate an error.

   If the sum of the two numbers cannot fit in the destination, an extra bit is required and this is signalled by the ADD operation setting the carry flags (CF) to 1. If the sum fits without spillage, CF=0. Other registers can be affected by addition operations as well; ZF=0 if the sum is zero, SF=1 if the sum is negative, etc. The logic of the basic addition command is:

   Format: **ADD** `destination,source`

   Logically, destination = destination + source

   Eg. ADD AX,BX

   b. **ADC instruction**

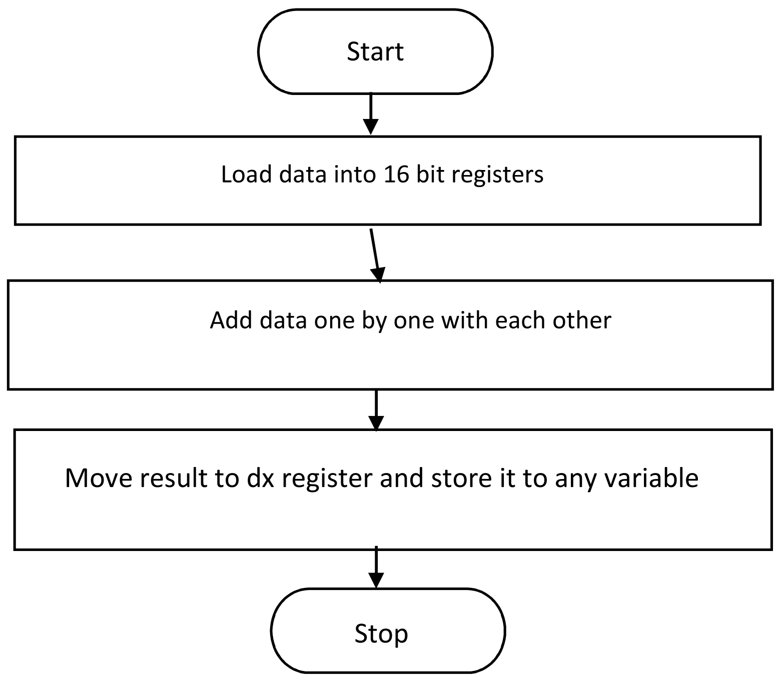
   ADC instructions add a number from some source to a number in specified destination and put result in destination like ADD instruction. It also adds the carry flag value 0 or 1 to the result, hence called Add with carry

   Syntax : ADC Destination, Source

   Logically, destination = destination + source + CF

   Example : ADC BX, AX

Flowchart:


Start
Load data into 16 bit registers
Add data one by one with each other
Move result to dx register and store it to any variable
Stop

Program:

```
data segment
   a dw 4200h
   b dw 5300h
   c dw 1600h
   d dw 8000h
   p dw ?
   data ends
code segment
   assume cs:code,ds:data
   start:
   mov ax,data
   mov ds,ax

   mov ax,a
   mov bx,b
   mov cx,c
   mov dx,d

   add ax,bx
   adc ax,cx
   adc ax,dx
   mov p,ax

   int 3
   code ends
end start
```

MEMORY LOCATION	OP-CODE	LABEL	MNEMOIC
7110h **7111h** **7112h**	**B8** **10** **07**		**mov ax,data**
7113h **7114h**	**8E** **D8**		**mov ds,ax**
7115h **7116h** **7117h**	**A1** **00** **00**		**mov ax,a**
7118h **7119h** **711Ah** **711Bh**	**8B** **1E** **02** **00**		**mov bx,b**

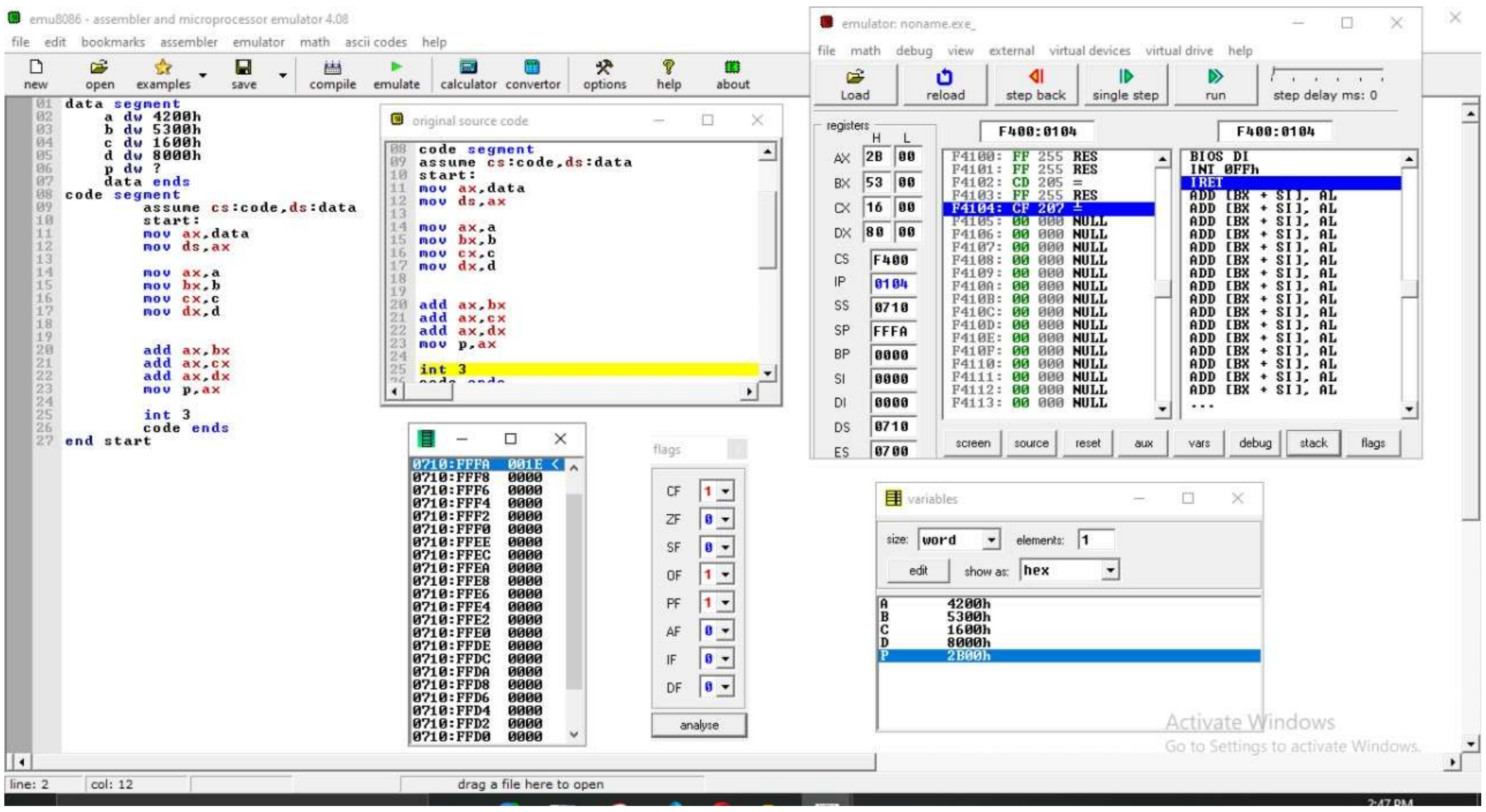
711Ch 711Dh 711Eh 711Fh	8B 0E 04 00		mov cx,c
7120h 7121h 7122h 7123h	8B 16 06 00		mov dx,d
7124h	03		add ax,bx
7126h 7127h	13 C1		adc ax,cx
7128h 7129h	13 C2		adc ax,dx
712Ah 712Bh 712Ch	A3 08 00		mov p, ax
712Dh	CC		int 3

Result:

INPUT		OUTPUT	
REGISTER	DATA	REGISTER	DATA
AX	4200h	AX	2B00h
BX	5300h		
CX	1600h		
DX	0000h		

Affected flag : Parity flag, Carry flag, Overflow flag

Output:



```
data segment
     a dw 4200h
     b dw 5300h
     c dw 1600h
     d dw 8000h
     p dw ?
     data ends
code segment
        assume cs:code,ds:data
        start:
        mov ax,data
        mov ds,ax

        mov ax,a
        mov bx,b
        mov cx,c
        mov dx,d


        add ax,bx
        add ax,cx
        add ax,dx
        mov p,ax

        int 3
        code ends
end start
```

```
registers
     H   L
AX   2B  00
BX   53  00
CX   16  00
DX   80  00
CS   F400
IP   0104
SS   0710
SP   FFFA
BP   0000
SI   0000
DI   0000
DS   0710
ES   0700
```

```
F400:0104
F4100: FF 255 RES
F4101: FF 255 RES
F4102: CD 205 =
F4103: FF 255 RES
F4104: CF 207 ±
F4105: 00 000 NULL
F4106: 00 000 NULL
F4107: 00 000 NULL
F4108: 00 000 NULL
F4109: 00 000 NULL
F410A: 00 000 NULL
F410B: 00 000 NULL
F410C: 00 000 NULL
F410D: 00 000 NULL
F410E: 00 000 NULL
F410F: 00 000 NULL
F4110: 00 000 NULL
F4111: 00 000 NULL
F4112: 00 000 NULL
F4113: 00 000 NULL
```

```
F400:0104
BIOS DI
INT 0FFh
IRET
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
...
```

```
0710:FFFA   001E <
0710:FFF8   0000
0710:FFF6   0000
0710:FFF4   0000
0710:FFF2   0000
0710:FFF0   0000
0710:FFEE   0000
0710:FFEC   0000
0710:FFEA   0000
0710:FFE8   0000
0710:FFE6   0000
0710:FFE4   0000
0710:FFE2   0000
0710:FFE0   0000
0710:FFDE   0000
0710:FFDC   0000
0710:FFDA   0000
0710:FFD8   0000
0710:FFD6   0000
0710:FFD4   0000
0710:FFD2   0000
0710:FFD0   0000
```

```
flags
CF 1
ZF 0
SF 0
OF 1
PF 1
AF 0
IF 0
DF 0
```

```
variables
size: word   elements: 1
show as: hex
A        4200h
B        5300h
C        1600h
D        8000h
P        2B00h
```

emu8086 - assembler and microprocessor emulator 4.08
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
original source code
emulator: noname.exe_
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
screen source reset aux vars debug stack flags
edit
analyse
line: 2 col: 12
drag a file here to open

Conclusion: Thus, we performed and understood multibyte addition of 2 numbers.

Experiment No 5

Aim : 8086 Assembly language programming for multiplication of two 16 bit signed and unsigned numbers

Requirement : Emu8086 (Assembler and Microprocessor emulator)

Theory :

1. **Data transfer instructions**

   Data transfer instructions are used to transfer data from source operand to destination operand

   a. **MOV instruction**

   MOV instruction moves data from one location to another. It also has the widest variety of parameters; so it the assembler programmer can use MOV effectively, the rest of the commands are easier to understand.

   Format: `MOV destination,source`
   Eg. MOV AX,6

2. **Arithmetic instructions**

   a. **MUL : -** MUL is used to multiply an unsigned byte in some source with an unsigned byte in AL register or an unsigned word in some source with an unsigned word in AX register.

   Flags affected : CF, OF

   Syntax : - MUL source

   Example : - MUL BH

   b. **IMUL : -** IMUL is used to multiply a signed byte from source with a signed byte in AL or a signed word from some source with a signed word in AX.

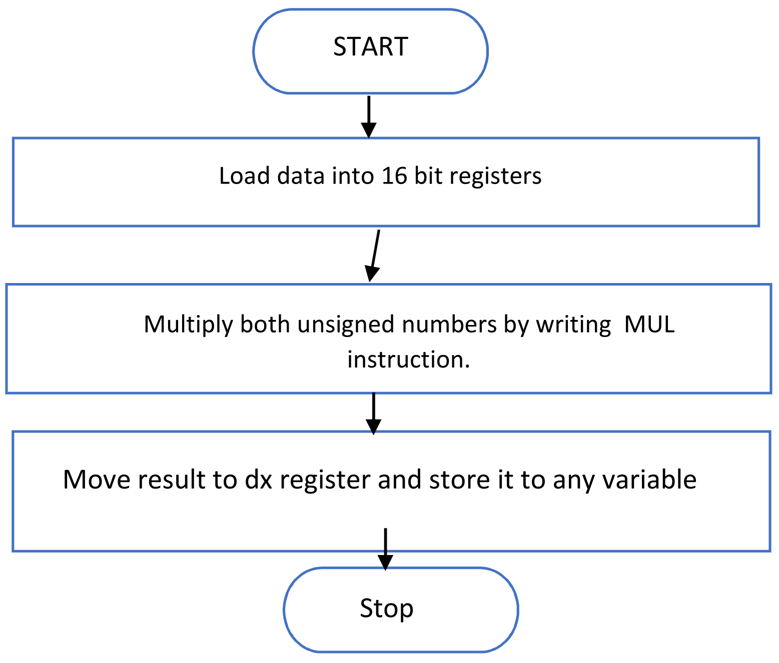
   Flags affected : CF, OF

   Syntax : - MUL source

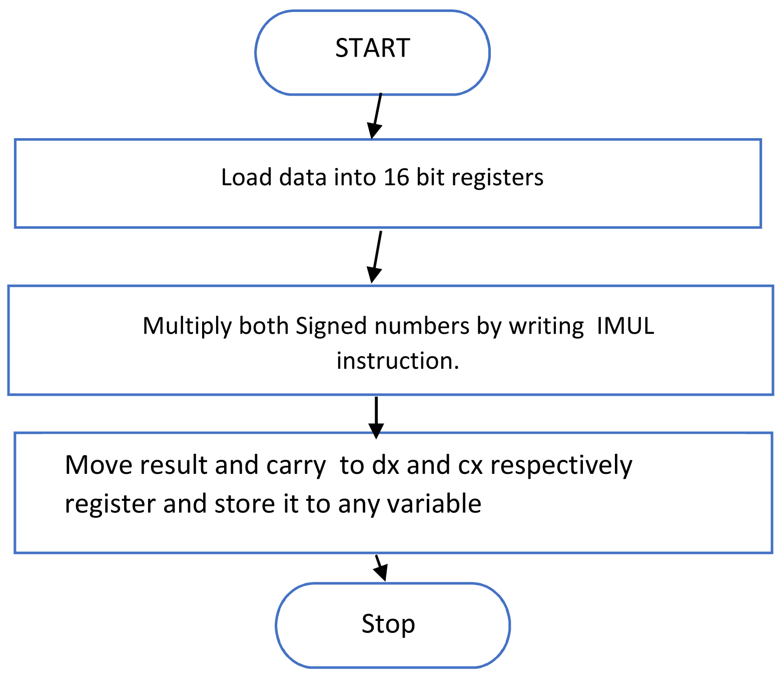
   Example : - IMUL BH

Flowchart:

1. Unsigned Numbers:


START
Load data into 16 bit registers
Multiply both unsigned numbers by writing MUL instruction.
Move result to dx register and store it to any variable
Stop

2. Signed Numbers:


START
Load data into 16 bit registers
Multiply both Signed numbers by writing IMUL instruction.
Move result and carry to dx and cx respectively register and store it to any variable
Stop

Programs:

Unsigned multiplication

```
data segment

   a dw 2342h

   b dw 3312h

   c dw ?

   d dw ?

   data ends

code segment

   assume cs: code, ds:data

   start:

   mov ax, data

   mov ds, ax

   mov ax, a

   mov bx, b

   mul bx

   mov c, dx

   mov d, ax

   int 3

   code ends

end start
```

MEMORY LOCATION	OP-CODE	LABEL	MNEMONIC
07110h	B8	-	MOV AX, DATA
07111h	10		
07112h	07		
07113h	8E	-	MOV DS, AX
07114h	D8		
07115h	A1	-	MOV AX, A
07116h	00		
07117h	00		
07118h	8B	-	MOV BX, B
07119h	1E		
0711Ah	02		
0711Bh	00		
0711Ch	F7	-	MUL BX
0711Dh	E3		
0711E	89	-	MOV C, DX
0711Fh	16		
07120h	04		
07121h	00		
07122h	A3	-	MOV D, AX
07123h	06		
07124h	00		
07125h	CC		INT 3

Input		Output	
Register	**Data**	**Register**	**Data**
AX	2342h	DX	0A04h
BX	3312h	AX	0708h

Flags Affected : Carry Flag, Overflow flag, Interrupt Flag

Signed multiplication

```
data segment
   a dw 2342h
   b dw 00d2h
   c dw ?
   d dw ?

   data ends

code segment
   assume cs: code, ds:data
   start:
   mov ax, data
   mov ds, ax
   mov ax, a
   mov bx, b
   imul bx
```

```
    mov c, dx
    mov d, ax
    int 3
    code ends
end start
```

MEMORY LOCATION	OP-CODE	LABEL	MNEMONIC
07110h 07111h 07112h	B8 10 07	-	MOV AX, DATA
07113h 07114h	8E D8	-	MOV DS, AX
07115h 07116h 07117h	A1 00 00	-	MOV AX, A
07118h 07119h 0711Ah 0711Bh	8B 1E 02 00	-	MOV BX, B
0711Ch 0711Dh	F7 EB	-	IMUL BX
0711Eh 0711Fh 07120h 07121h	89 16 04 00	-	MOV C, DX
07122h 07123h 07124h	A3 06 00	-	MOV D, AX
07125h	CC	-	Int 3

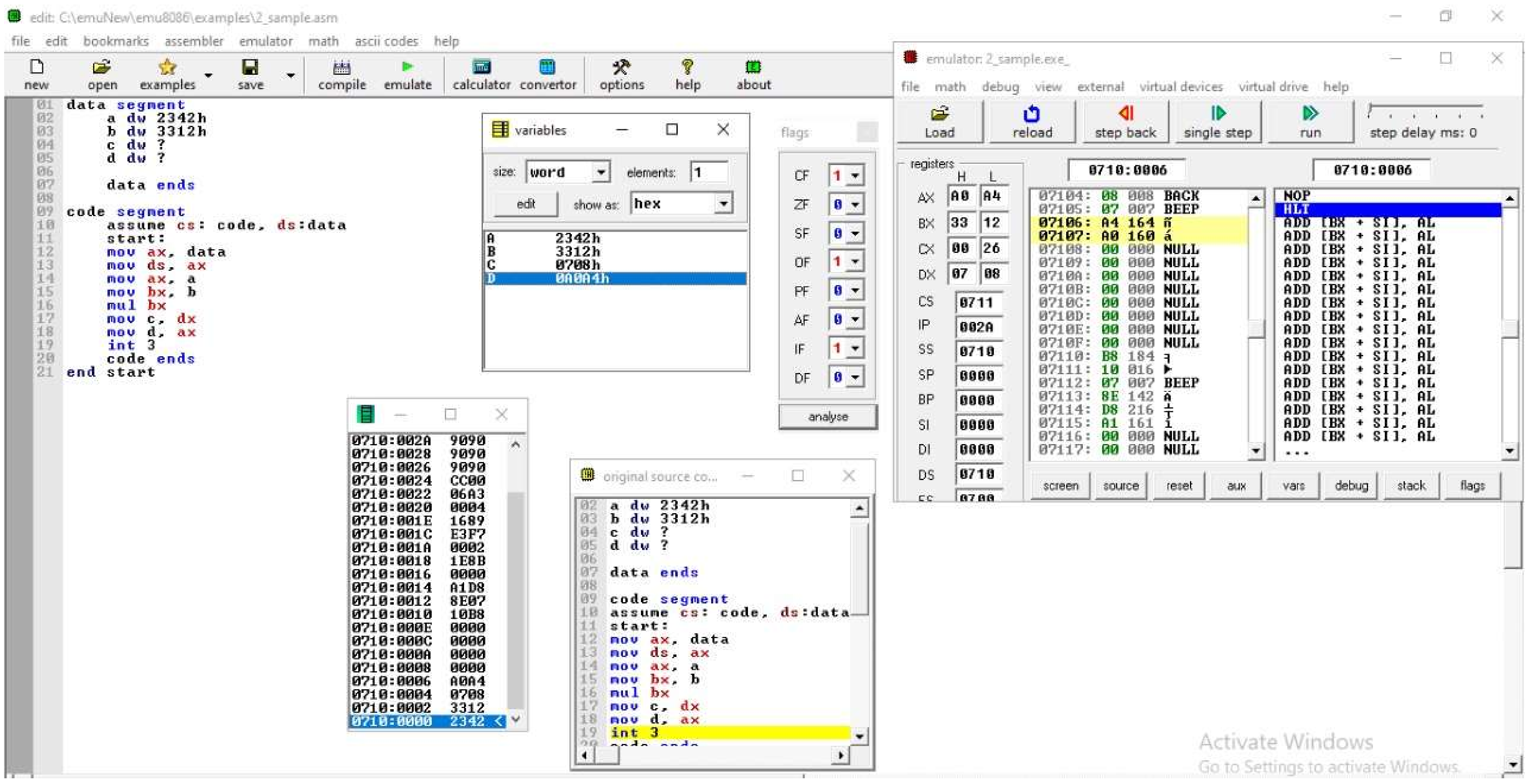
Result:

Input		Output	
Register	**Data**	**Register**	**Data**
AX	2342h	DX	001Ch
BX	00D2h	AX	EC24h

Flags Affected : Carry flag, Overflow flag

Screenshots:

Unsigned multiplication:



edit: C:\emuNew\emu8086\examples\2_sample.asm

```
data segment
      a dw 2342h
      b dw 3312h
      c dw ?
      d dw ?

      data ends

code segment
      assume cs: code, ds:data
      start:
      mov ax, data
      mov ds, ax
      mov ax, a
      mov bx, b
      mul bx
      mov c, dx
      mov d, ax
      int 3
      code ends
end start
```

variables — size: word, elements: 1, show as: hex

```
A        2342h
B        3312h
C        0708h
D        0A0A4h
```

flags: CF 1, ZF 0, SF 0, OF 1, PF 0, AF 0, IF 1, DF 0

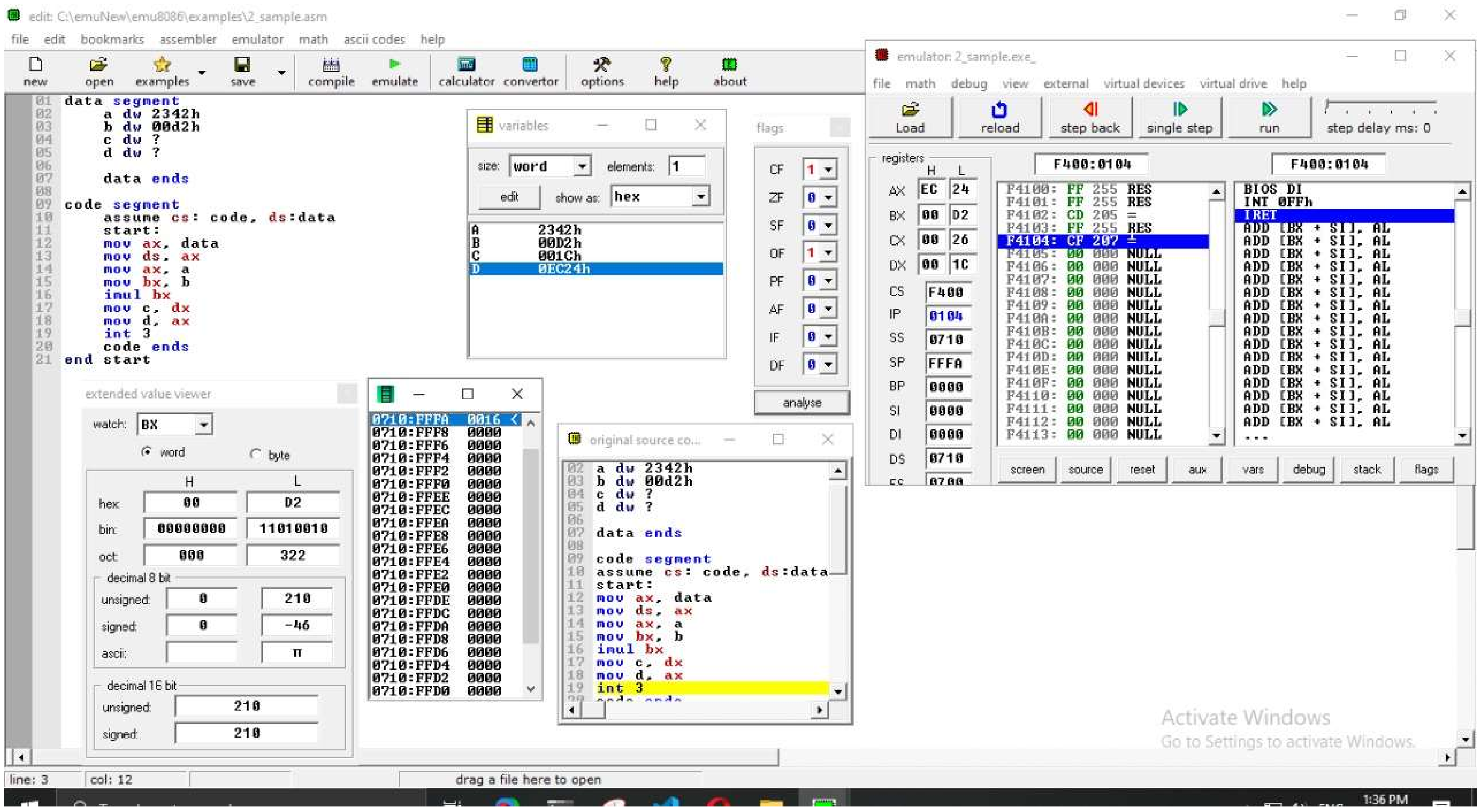
```
0710:002A   9090
0710:0028   9090
0710:0026   9090
0710:0024   CC00
0710:0022   06A3
0710:0020   0004
0710:001E   1689
0710:001C   E3F7
0710:001A   0002
0710:0018   1E8B
0710:0016   0000
0710:0014   A1D8
0710:0012   8E07
0710:0010   10B8
0710:000E   0000
0710:000C   0000
0710:000A   0000
0710:0008   0000
0710:0006   A0A4
0710:0004   0708
0710:0002   3312
0710:0000   2342 <
```

emulator: 2_sample.exe_

```
registers
     H   L
AX   A0  A4
BX   33  12
CX   00  26
DX   07  08
CS   0711
IP   002A
SS   0710
SP   0000
BP   0000
SI   0000
DI   0000
DS   0710
ES   0700
```

```
0710:0006                          0710:0006
07104: 08 008 BACK                 NOP
07105: 07 007 BEEP                 HLT
07106: A4 164 ñ                    ADD [BX + SI], AL
07107: A0 160 á                    ADD [BX + SI], AL
07108: 00 000 NULL                 ADD [BX + SI], AL
07109: 00 000 NULL                 ADD [BX + SI], AL
0710A: 00 000 NULL                 ADD [BX + SI], AL
0710B: 00 000 NULL                 ADD [BX + SI], AL
0710C: 00 000 NULL                 ADD [BX + SI], AL
0710D: 00 000 NULL                 ADD [BX + SI], AL
0710E: 00 000 NULL                 ADD [BX + SI], AL
0710F: 00 000 NULL                 ADD [BX + SI], AL
07110: B8 184 ╕                    ADD [BX + SI], AL
07111: 10 016 ►                    ADD [BX + SI], AL
07112: 07 007 BEEP                 ADD [BX + SI], AL
07113: 8E 142 Ä                    ADD [BX + SI], AL
07114: D8 216 ╪                    ADD [BX + SI], AL
07115: A1 161 í                    ADD [BX + SI], AL
07116: 00 000 NULL                 ADD [BX + SI], AL
07117: 00 000 NULL                 ...
```

Signed multiplication:



edit: C:\emuNew\emu8086\examples\2_sample.asm

```
data segment
      a dw 2342h
      b dw 00d2h
      c dw ?
      d dw ?

      data ends

code segment
      assume cs: code, ds:data
      start:
      mov ax, data
      mov ds, ax
      mov ax, a
      mov bx, b
      imul bx
      mov c, dx
      mov d, ax
      int 3
      code ends
end start
```

variables — size: word, elements: 1, show as: hex

```
A        2342h
B        00D2h
C        001Ch
D        0EC24h
```

flags: CF 1, ZF 0, SF 0, OF 1, PF 0, AF 0, IF 0, DF 0

extended value viewer — watch: BX, (•) word ( ) byte

	H	L
hex:	00	D2
bin:	00000000	11010010
oct:	000	322
decimal 8 bit unsigned:	0	210
decimal 8 bit signed:	0	-46
ascii:		π
decimal 16 bit unsigned:	210	
decimal 16 bit signed:	210	

```
0710:FFFA   0016 <
0710:FFF8   0000
0710:FFF6   0000
0710:FFF4   0000
0710:FFF2   0000
0710:FFF0   0000
0710:FFEE   0000
0710:FFEC   0000
0710:FFEA   0000
0710:FFE8   0000
0710:FFE6   0000
0710:FFE4   0000
0710:FFE2   0000
0710:FFE0   0000
0710:FFDE   0000
0710:FFDC   0000
0710:FFDA   0000
0710:FFD8   0000
0710:FFD6   0000
0710:FFD4   0000
0710:FFD2   0000
0710:FFD0   0000
```

emulator: 2_sample.exe_

```
registers
     H   L
AX   EC  24
BX   00  D2
CX   00  26
DX   00  1C
CS   F400
IP   0104
SS   0710
SP   FFFA
BP   0000
SI   0000
DI   0000
DS   0710
ES   0700
```

```
F400:0104                          F400:0104
F4100: FF 255 RES                  BIOS DI
F4101: FF 255 RES                  INT 0FFh
F4102: CD 205 =                    IRET
F4103: FF 255 RES                  ADD [BX + SI], AL
F4104: CF 207 ╧                    ADD [BX + SI], AL
F4105: 00 000 NULL                 ADD [BX + SI], AL
F4106: 00 000 NULL                 ADD [BX + SI], AL
F4107: 00 000 NULL                 ADD [BX + SI], AL
F4108: 00 000 NULL                 ADD [BX + SI], AL
F4109: 00 000 NULL                 ADD [BX + SI], AL
F410A: 00 000 NULL                 ADD [BX + SI], AL
F410B: 00 000 NULL                 ADD [BX + SI], AL
F410C: 00 000 NULL                 ADD [BX + SI], AL
F410D: 00 000 NULL                 ADD [BX + SI], AL
F410E: 00 000 NULL                 ADD [BX + SI], AL
F410F: 00 000 NULL                 ADD [BX + SI], AL
F4110: 00 000 NULL                 ADD [BX + SI], AL
F4111: 00 000 NULL                 ADD [BX + SI], AL
F4112: 00 000 NULL                 ADD [BX + SI], AL
F4113: 00 000 NULL                 ...
```

line: 3   col: 12   drag a file here to open

Conclusion: Thus we performed and understood multiplication of signed and unsigned 16 bit numbers

Experiment No 6

Aim : 8086 Assembly language programming for multiplication of two 16 bit signed and unsigned numbers

Requirement : Emu8086 (Assembler and Microprocessor emulator)

Theory :

1. **Data transfer instructions**

   Data transfer instructions are used to transfer data from source operand to destination operand

   a. **MOV instruction**

   MOV instruction moves data from one location to another. It also has the widest variety of parameters; so it the assembler programmer can use MOV effectively, the rest of the commands are easier to understand.

   Format: `MOV destination,source`

   Eg. MOV AX,6

2. **Arithmetic instructions**

   a. **DIV : -** DIV instruction is used to divide an *unsigned* word by a byte or to divide an *unsigned* double word (32 bits) by a word. When a word is divided by a byte, the word must be in the AX register.

   Flags affected :- All of the arithmetic status flags are undefined after the operation.

   Syntax : - MUL source

   Example : - MUL BH

   b. **IDIV:-** This instruction is used to divide a *signed* word by a *signed* byte, or to divide a *signed* double word by a *signed* word. When dividing a signed word by a signed byte, the word must be in the AX register

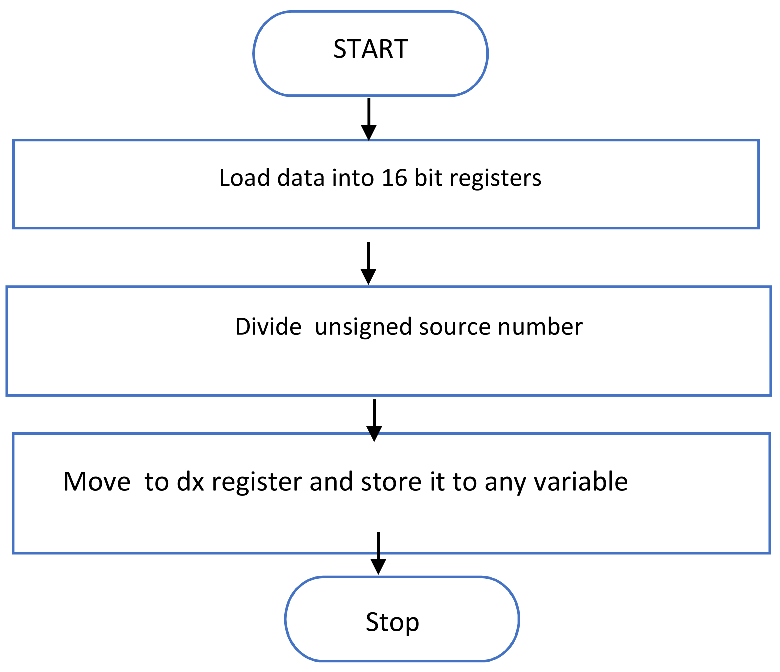
Flags affected :- All of the arithmetic status flags are undefined after the operation.

Syntax :- IDIV source

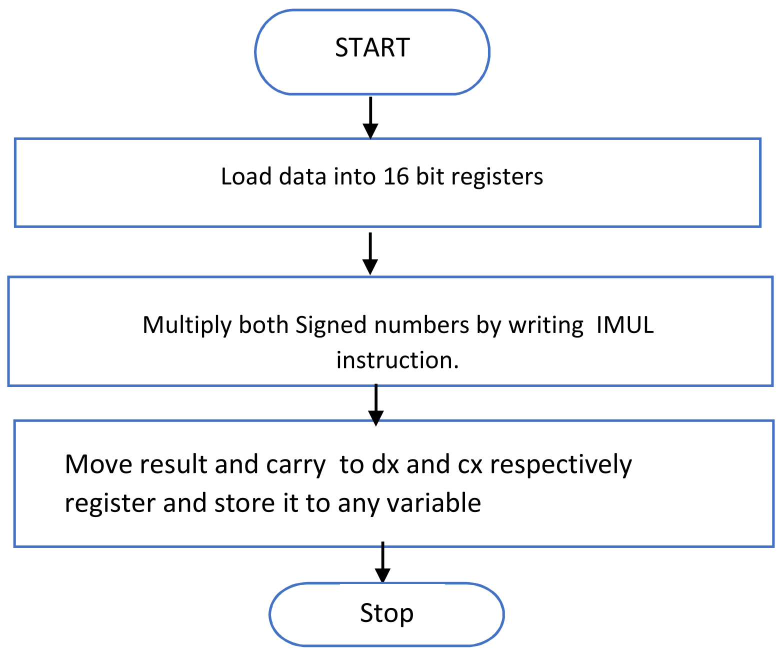
Example:- IDIV BL

Flowchart:

Unsigned division


START
Load data into 16 bit registers
Divide unsigned source number
Move to dx register and store it to any variable
Stop

Signed division:


START
Load data into 16 bit registers
Multiply both Signed numbers by writing IMUL instruction.
Move result and carry to dx and cx respectively register and store it to any variable
Stop

Program:

```
data segment
   a dw 4235h
   b dw 2264h
   c dw ?
   d dw ?
   data ends
code segment
   assume cs:code ds:data
   start:
   mov ax,data
   mov ds,ax
   mov ax,a
   mov bx,b
   div bx
   mov cx,ax
   mov c,cx
   mov d,dx
   int 3
   code ends
end start
```

Unsigned division

MEMORY LOCATION	OP-CODE	LABEL	MNEMONIC
7110h 7111h 7112h	B8 10 07	-	MOV AX, DATA
7113h 7114h	8E D8	-	MOV DS, AX
7115h 7116hh 7117h	A1 00 00	-	MOV AX, A
7118h 7119h 711Ah 711Bh	8B 1E 02 00	-	MOV BX, B
711Ch 711Dh	F7 FB	-	DIV BX
711Eh 711Fh	8B C8	-	MOV CX, AX
7120hh 7121h 7122h 7123h	89 0E 04 00	-	MOV C, CX
7124h 7125h 7126h 7127h	89 16 06 00	-	MOV D, DX
7128h	CC		Int 3

Result:

Input		Output	
Register	**Data**	**Register**	**Data**
AX	000Ah	CX	0003h
BX	0003h	DX	0001h

Flags Affected : Interrupt Flag

Signed division

MEMORY LOCATION	OP-CODE	LABEL	MNEMONIC
7110h 7111h 7112h	B8 10 07	-	MOV AX, DATA
7113h 7114h	8E D8	-	MOV DS, AX
7115h 7116h 7117h	A1 00 00	-	MOV AX, A
7118h 7119hh 711Ah 711Bh	8B 1E 02 00	-	MOV BX, B
711Chh 711Dh	F7 FB	-	IDIV BX
711Eh 711Fhh	8B C8	-	MOV CX, AX
7120h 7121h 7122h 7123h	89 0E 04 00	-	MOV C, CX
7124h 7125h 7126h	89 16 06	-	MOV D, DX

7127h	00		

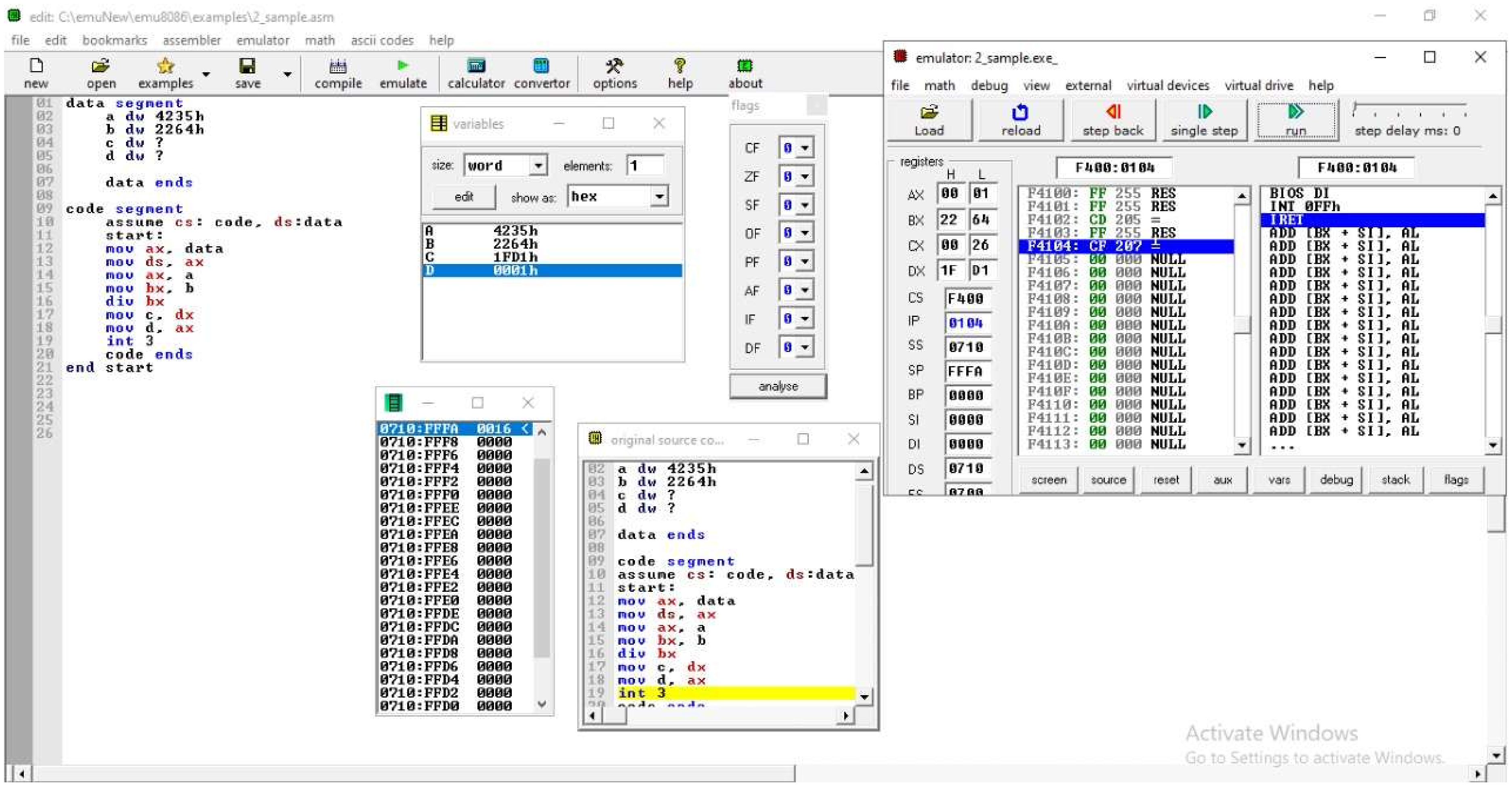
Result :

Input		Output	
Register	**Data**	**Register**	**Data**
AX	00F6h	CX	0052h
BX	0003h	DX	0000h

Flags Affected : Interrupt Flag

Output:

Unsigned division:



edit: C:\emuNew\emu8086\examples\2_sample.asm

```
data segment
      a dw 4235h
      b dw 2264h
      c dw ?
      d dw ?

      data ends

code segment
      assume cs: code, ds:data
      start:
      mov ax, data
      mov ds, ax
      mov ax, a
      mov bx, b
      div bx
      mov c, dx
      mov d, ax
      int 3
      code ends
end start
```

variables — size: word, elements: 1, show as: hex

```
A        4235h
B        2264h
C        1FD1h
D        0001h
```

flags: CF 0, ZF 0, SF 0, OF 0, PF 0, AF 0, IF 0, DF 0

```
0710:FFFA   0016 <
0710:FFF8   0000
0710:FFF6   0000
0710:FFF4   0000
0710:FFF2   0000
0710:FFF0   0000
0710:FFEE   0000
0710:FFEC   0000
0710:FFEA   0000
0710:FFE8   0000
0710:FFE6   0000
0710:FFE4   0000
0710:FFE2   0000
0710:FFE0   0000
0710:FFDE   0000
0710:FFDC   0000
0710:FFDA   0000
0710:FFD8   0000
0710:FFD6   0000
0710:FFD4   0000
0710:FFD2   0000
0710:FFD0   0000
```

emulator: 2_sample.exe_

registers:

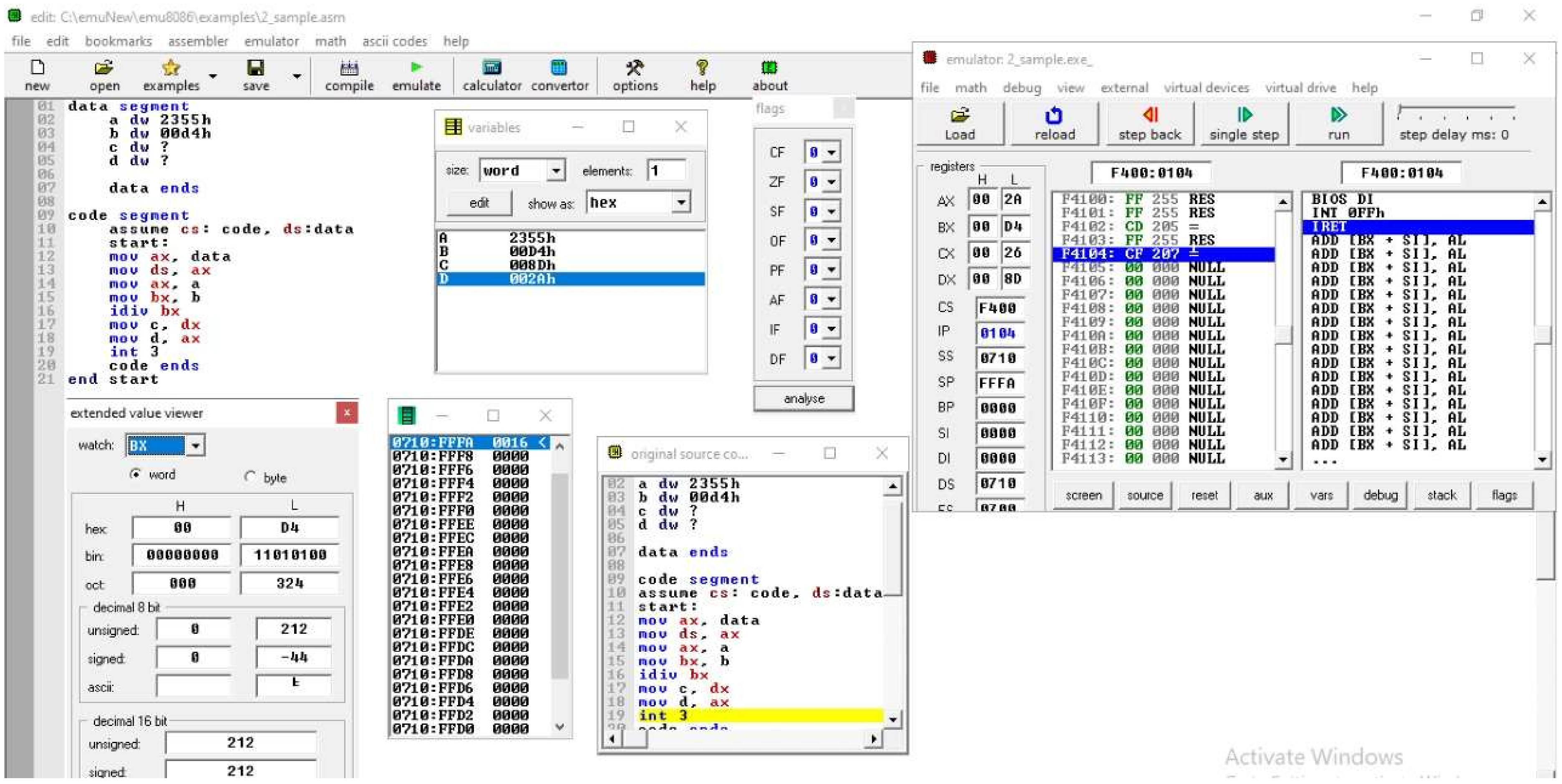
```
     H  L
AX  00 01
BX  22 64
CX  00 26
DX  1F D1
CS  F400
IP  0104
SS  0710
SP  FFFA
BP  0000
SI  0000
DI  0000
DS  0710
ES  0700
```

```
F400:0104
F4100: FF 255 RES
F4101: FF 255 RES
F4102: CD 205 =
F4103: FF 255 RES
F4104: CF 207 ±
F4105: 00 000 NULL
F4106: 00 000 NULL
F4107: 00 000 NULL
F4108: 00 000 NULL
F4109: 00 000 NULL
F410A: 00 000 NULL
F410B: 00 000 NULL
F410C: 00 000 NULL
F410D: 00 000 NULL
F410E: 00 000 NULL
F410F: 00 000 NULL
F4110: 00 000 NULL
F4111: 00 000 NULL
F4112: 00 000 NULL
F4113: 00 000 NULL
```

```
F400:0104
BIOS DI
INT 0FFh
IRET
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
...
```

file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
screen source reset aux vars debug stack flags
analyse
original source co...
Activate Windows
Go to Settings to activate Windows.

Signed division:



edit: C:\emuNew\emu8086\examples\2_sample.asm

```
data segment
      a dw 2355h
      b dw 00d4h
      c dw ?
      d dw ?

      data ends

code segment
      assume cs: code, ds:data
      start:
      mov ax, data
      mov ds, ax
      mov ax, a
      mov bx, b
      idiv bx
      mov c, dx
      mov d, ax
      int 3
      code ends
end start
```

variables — size: word, elements: 1, show as: hex

```
A        2355h
B        00D4h
C        008Dh
D        002Ah
```

flags: CF 0, ZF 0, SF 0, OF 0, PF 0, AF 0, IF 0, DF 0

extended value viewer — watch: BX, (• word) (byte)

	H	L
hex:	00	D4
bin:	00000000	11010100
oct:	000	324
decimal 8 bit		
unsigned:	0	212
signed:	0	-44
ascii:		╘
decimal 16 bit		
unsigned:	212	
signed:	212	

```
0710:FFFA   0016 <
0710:FFF8   0000
0710:FFF6   0000
0710:FFF4   0000
0710:FFF2   0000
0710:FFF0   0000
0710:FFEE   0000
0710:FFEC   0000
0710:FFEA   0000
0710:FFE8   0000
0710:FFE6   0000
0710:FFE4   0000
0710:FFE2   0000
0710:FFE0   0000
0710:FFDE   0000
0710:FFDC   0000
0710:FFDA   0000
0710:FFD8   0000
0710:FFD6   0000
0710:FFD4   0000
0710:FFD2   0000
0710:FFD0   0000
```

emulator: 2_sample.exe_

registers:

```
     H  L
AX  00 2A
BX  00 D4
CX  00 26
DX  00 8D
CS  F400
IP  0104
SS  0710
SP  FFFA
BP  0000
SI  0000
DI  0000
DS  0710
ES  0700
```

```
F400:0104
F4100: FF 255 RES
F4101: FF 255 RES
F4102: CD 205 =
F4103: FF 255 RES
F4104: CF 207 ±
F4105: 00 000 NULL
F4106: 00 000 NULL
F4107: 00 000 NULL
F4108: 00 000 NULL
F4109: 00 000 NULL
F410A: 00 000 NULL
F410B: 00 000 NULL
F410C: 00 000 NULL
F410D: 00 000 NULL
F410E: 00 000 NULL
F410F: 00 000 NULL
F4110: 00 000 NULL
F4111: 00 000 NULL
F4112: 00 000 NULL
F4113: 00 000 NULL
```

```
F400:0104
BIOS DI
INT 0FFh
IRET
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
...
```

file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
screen source reset aux vars debug stack flags
analyse
original source co...
Activate Windows

Conclusion: – Thus we have understood 8086 Assembly language programming for division of two 16 bit signed and unsigned numbers.

Experiment no. 7

Aim: 8086 Assembly language programming for arranging 16 bit numbers in ascending order.

Requirements: 8086 microprocessor kit/MASM ----1 2.RPS (+5V) ----1,]and Emu8086 Emulator

Theory:

Data Transfer Instructions : -

1) **MOV :-** MOV instruction is used to trasnfer byte or word from register to register, memory to register, register to memory or with immediate addressing.
   Flags affected :- None

   Syntax :- MOV destination, source

   Example :- MOV AX,00F4H

2) **XCHG :-** The XCHG instruction exchanges the content of a register with the content of another register. It cannot directly exchange the content of two memory locations.
   Flags affected :- None

   Syntax :- XCHG Destination, Source

   Example :- XCHG AX, DX

3) **LEA :-** LEA instruction computes the effective address of the second operand (source operand) and stores it in the first operand (destination operand).
   Flags affected :- None

   Syntax :- LEA destination, source

   Example :- LEA AX, BX

Arithmetic Instructions : -

4) **ADD :-** ADD instruction is used to add the current contents of destination with that of source and store the result in destination.
   Flags affected :- AF, CF, OF, PF, SF and ZF

   Syntax :- ADD destination, source

   Example :- ADD AL,0FH

5) **DEC :-** DEC instruction is used to decrement the content of the specified destination by one.
   Flags affected :- AF, CF, OF, PF, SF and ZF

   Syntax :- DEC destination, source

   Example :- DEC AX

6) **CMP :-** CMP instruction is used to compare the source operand, which maybe a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location. For comparison, it subtracts the source operand from the destination operand but does not store the result anywhere.
   Flags affected :- The flags are affected depending upon the result of the subtraction.
   If both of the operands are equal, then zero flag is affected.
   If the source operand is greater than the destination operand, then carry flag is set or else, carry flag is reset.
   Syntax :- CMP destination, source

   Example :- CMP AX,BX

Transfer-Of-Control Instructions : -

7) **JC (JUMP IF CARRY):-** If, after a compare or some other instructions which affect flags, the carry flag is a 1, this instruction will cause execution to jump to a label given in the instruction. If CF is 0, the instruction will have no effect on program execution.
   Flags affected :- None

   Syntax : - LOOP label

   Example:- ADD BX, CX
   JC NEXT

8) **JNZ :-** JNZ instruction is a conditional jump that follows a test. It jumps to the specified location if the Zero flag is cleared (0).

Flags affected :- None

Syntax :- JNZ location

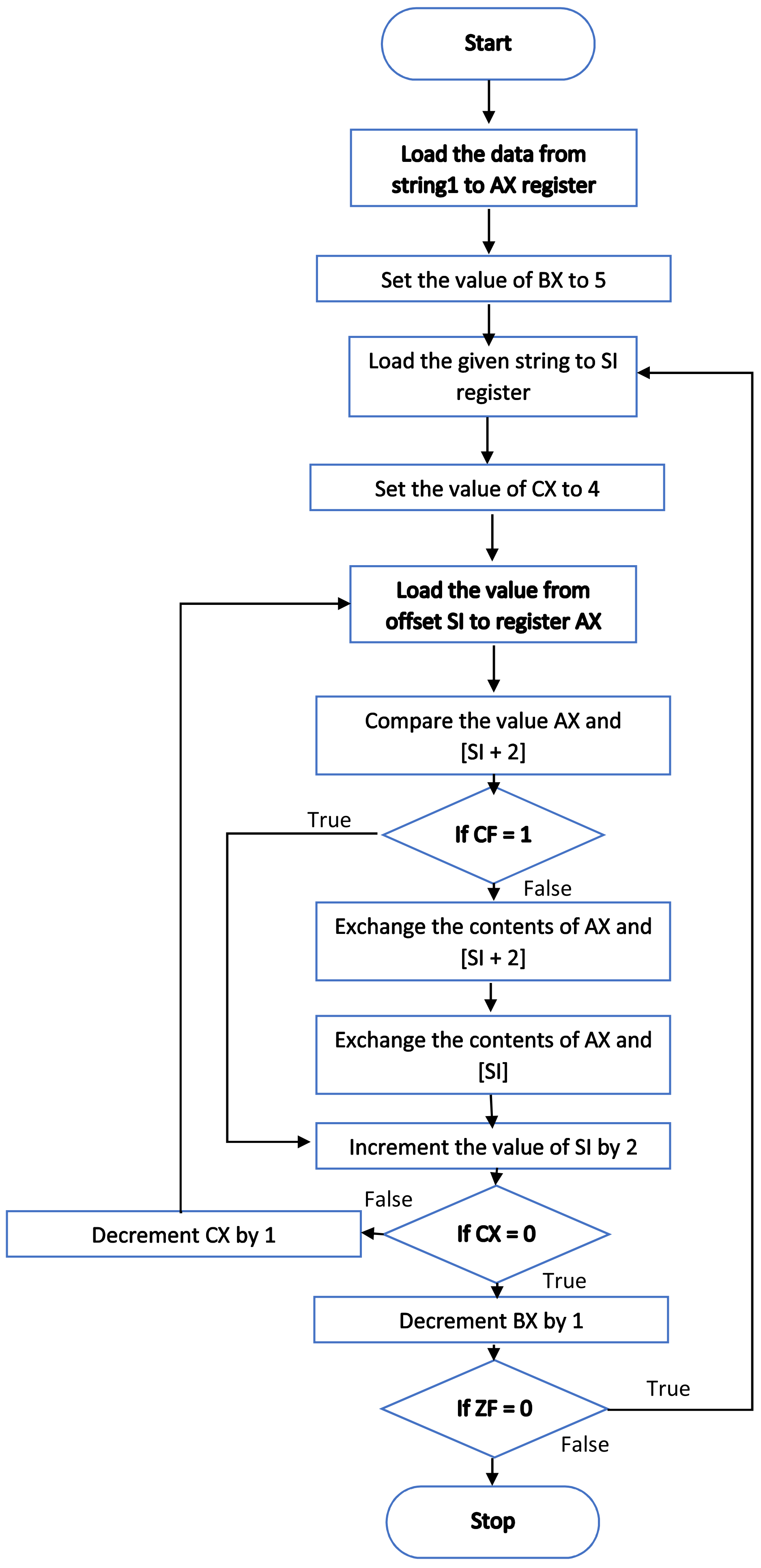
Example :- JNZ 4000

9) LOOP :- This instruction is used to repeat a series of instructions some number of time.Loops through a sequence of instructions until CX=0
Flags affected :- None

Syntax : - LOOP label

Example :-
```
MOV BX, OFFSET PRICES
 MOV CX, 40
 NEXT: MOV AL, [BX]
INC AL
 MOV [BX], AL
 INC BX
LOOP NEXT
```

➢ **Flowchart :**


Start
Load the data from string1 to AX register
Set the value of BX to 5
Load the given string to SI register
Set the value of CX to 4
Load the value from offset SI to register AX
Compare the value AX and [SI + 2]
If CF = 1
True
False
Exchange the contents of AX and [SI + 2]
Exchange the contents of AX and [SI]
Increment the value of SI by 2
If CX = 0
False
Decrement CX by 1
True
Decrement BX by 1
If ZF = 0
True
False
Stop

Code :

```
data segment
   string1 dw 14h,10h,5h,15h,21h
   data ends
   code segment
      assume cs:code,ds:data
      start:
      mov ax,data
      mov ds,ax
     mov bx,5
     up1: lea si,string1
     mov cx,4
     up: mov ax,[si]
     cmp ax,[si+2]
     jc down

    xchg ax,[si+2]
    xchg ax,[si]
    down: add si,2
    loop up
    dec bx
    jnz up1
    int 3
     code ends
end start
```

Program:

MEMORY LOCATION	OP-CODE	LABEL	MNEMONIC
07110h 07111h 07112hh	B8 10 07		MOV AX, DATA
07113h 07114h	8E D8		MOV DS, AX
07115h 07116h 07117h	BB 05 00		MOV BX, 5
07118h 07119h 0711Ah	BE 00 00	UP1	LEA SI, STRING1
0711Bh 0711Ch 0711Dh	B9 04 00		MOV CX, 4
0711Eh 0711Fh	8B 04	UP	MOV AX, [SI]
07120h 07121h 07122h	3B 44 02		CMP AX, [SI + 2]
07123h 07124h	72 05		JC DOWN

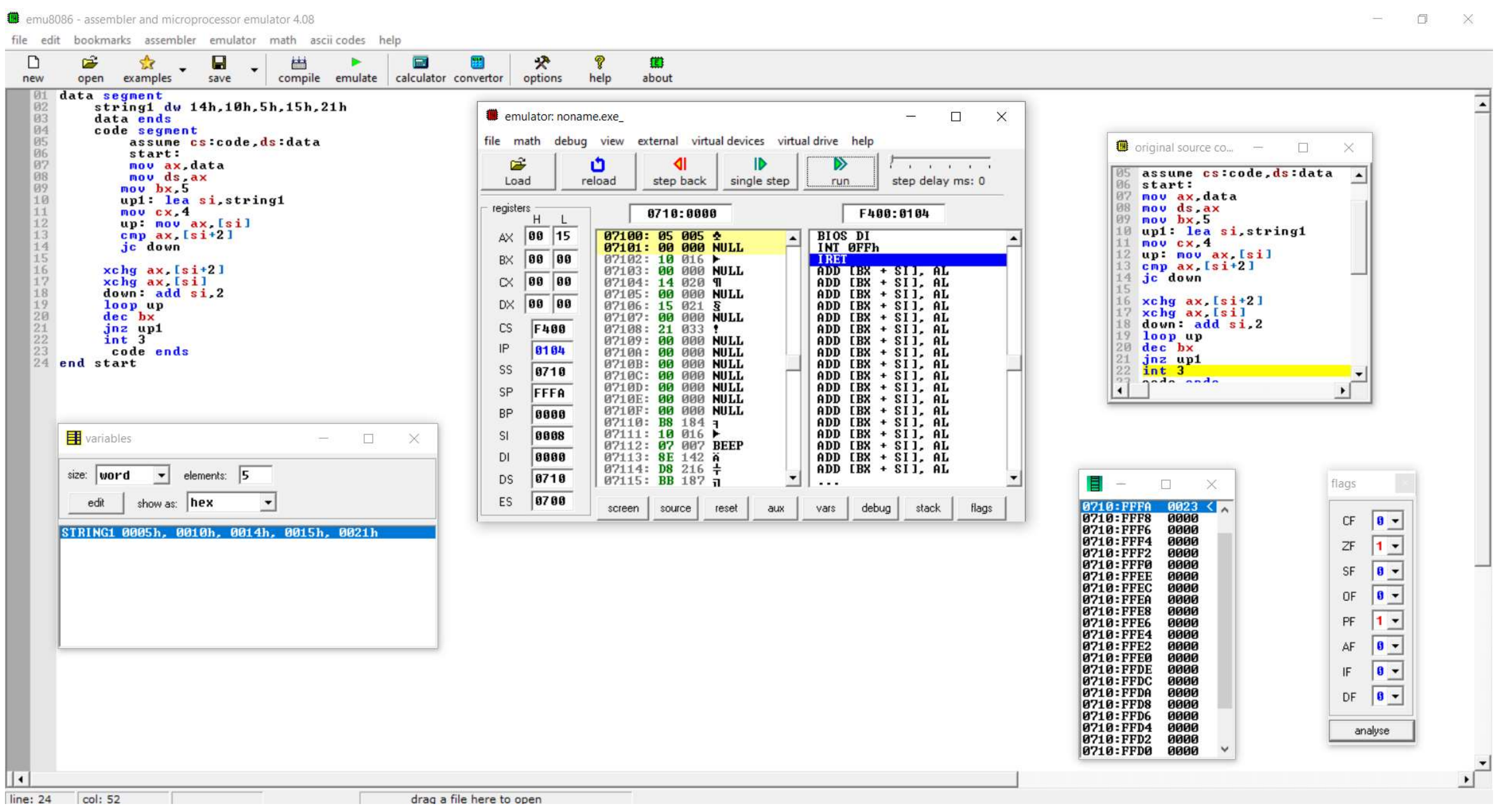
07125h 07126hh 07127h	87 44 02		XCHG AX, [SI + 2]
07128h 07129h	87 04		XCHG AX, [SI]
0712Ah 0712Bh 0712Chh	83 C6 02	DOWN	ADD SI, 2
0712Dh 0712Eh	E2 EF		LOOP UP
0712Fh	4B		DEC BX
07130h 07131h	75 E6		JNZ UP1
07132h	CC		INT 3

Result:

Input		Output	
Register	**Data**	**Register**	**Data**
AX	0014h,0010h,0005h, 0015h,0021h	AX	0005h, 0010h, 0014h, 0015h, 0021h

Flag Affected : Parity flag, Zero flag

Screenshot:


emu8086 - assembler and microprocessor emulator 4.08
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
emulator: noname.exe_
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
screen source reset aux vars debug stack flags
original source co...
analyse
line: 24 col: 52 drag a file here to open

```
data segment
     string1 dw 14h,10h,5h,15h,21h
     data ends
     code segment
         assume cs:code,ds:data
         start:
         mov ax,data
         mov ds,ax
        mov bx,5
        up1: lea si,string1
        mov cx,4
        up: mov ax,[si]
        cmp ax,[si+2]
        jc down

      xchg ax,[si+2]
      xchg ax,[si]
      down: add si,2
      loop up
      dec bx
      jnz up1
      int 3
       code ends
end start
```

```
registers  H   L
AX        00  15
BX        00  00
CX        00  00
DX        00  00
CS        F400
IP        0104
SS        0710
SP        FFFA
BP        0000
SI        0008
DI        0000
DS        0710
ES        0700
```

```
0710:0000
07100: 05 005 ♣
07101: 00 000 NULL
07102: 10 016 ►
07103: 00 000 NULL
07104: 14 020 ¶
07105: 00 000 NULL
07106: 15 021 §
07107: 00 000 NULL
07108: 21 033 !
07109: 00 000 NULL
0710A: 00 000 NULL
0710B: 00 000 NULL
0710C: 00 000 NULL
0710D: 00 000 NULL
0710E: 00 000 NULL
0710F: 00 000 NULL
07110: B8 184 ╕
07111: 10 016 ►
07112: 07 007 BEEP
07113: 8E 142 Ä
07114: D8 216 ╪
07115: BB 187 ╗
```

```
F400:0104
BIOS DI
INT 0FFh
IRET
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
...
```

variables
size: word elements: 5
edit show as: hex

```
STRING1 0005h, 0010h, 0014h, 0015h, 0021h
```

```
0710:FFFA  0023 <
0710:FFF8  0000
0710:FFF6  0000
0710:FFF4  0000
0710:FFF2  0000
0710:FFF0  0000
0710:FFEE  0000
0710:FFEC  0000
0710:FFEA  0000
0710:FFE8  0000
0710:FFE6  0000
0710:FFE4  0000
0710:FFE2  0000
0710:FFE0  0000
0710:FFDE  0000
0710:FFDC  0000
0710:FFDA  0000
0710:FFD8  0000
0710:FFD6  0000
0710:FFD4  0000
0710:FFD2  0000
0710:FFD0  0000
```

flags

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

Conclusion: Thus we have understood 8086 Assembly language programming for arranging 16 bit numbers in ascending order.

Experiment no. 8

Aim: 8086 Assembly language programming for arranging 16 bit numbers in descending order.

Requirements: 8086 microprocessor kit/MASM ----1 2.RPS (+5V) ----1,]and Emu8086 Emulator

Theory:

Data Transfer Instructions : -

1) **MOV :-** MOV instruction is used to transfer byte or word from register to register, memory to register, register to memory or with immediate addressing.
   Flags affected :- None

   Syntax :- MOV destination, source

   Example :- MOV AX,BX

2) **LEA :-** LEA instruction computes the effective address of the second operand (source operand) and stores it in the first operand (destination operand).
   Flags affected :- None

   Syntax :- LEA destination, source

   Example :- LEA AX, BX

3) **XCHG :-** The XCHG instruction exchanges the content of a register with the content of another register. It cannot directly exchange the content of two memory locations.
   Flags affected :- None

   Syntax :- XCHG Destination, Source

   Example :- XCHG AX, DX

Arithmetic Instructions : -

4) **CMP -** CMP instruction is used to compare the source operand, which maybe a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location. For comparison, it subtracts the source operand from the destination operand but does not store the result anywhere.
   Flags affected :- The flags are affected depending upon the result of the subtraction.
   If both of the operands are equal, then zero flag is affected.
   If the source operand is greater than the destination operand, then carry flag is set or else, carry flag is reset.

   Syntax :- CMP destination, source

   Example :- CMP AX,B

5) **DEC :-** DEC instruction is used to decrement the content of the specified destination by one.
   Flags affected :- AF, CF, OF, PF, SF and ZF

   Syntax :- DEC destination

   Example :- DEC AX

6) **ADD :-** ADD instruction is used to add the current contents of destination with that of source and store the result in destination.
   Flags affected :- AF, CF, OF, PF, SF and ZF

   Syntax :- ADD destination, source

   Example :- ADD AL,0FH

Program Execution Transfer Instructions : -

7) **JNC :-** The JNC (Jump if No Carry) instruction transfers program control to the specified address if the carry flag is 0.
   Flags affected :- None

Syntax :-

Example :- ADD AL, BL
JNC NEXT

8) **JNZ :-** JNZ instruction is a conditional jump that follows a test. It jumps to the specified location if the Zero flag is cleared (0).
Flags affected :- None

Syntax :- JNZ location

Example :- JNZ 4000

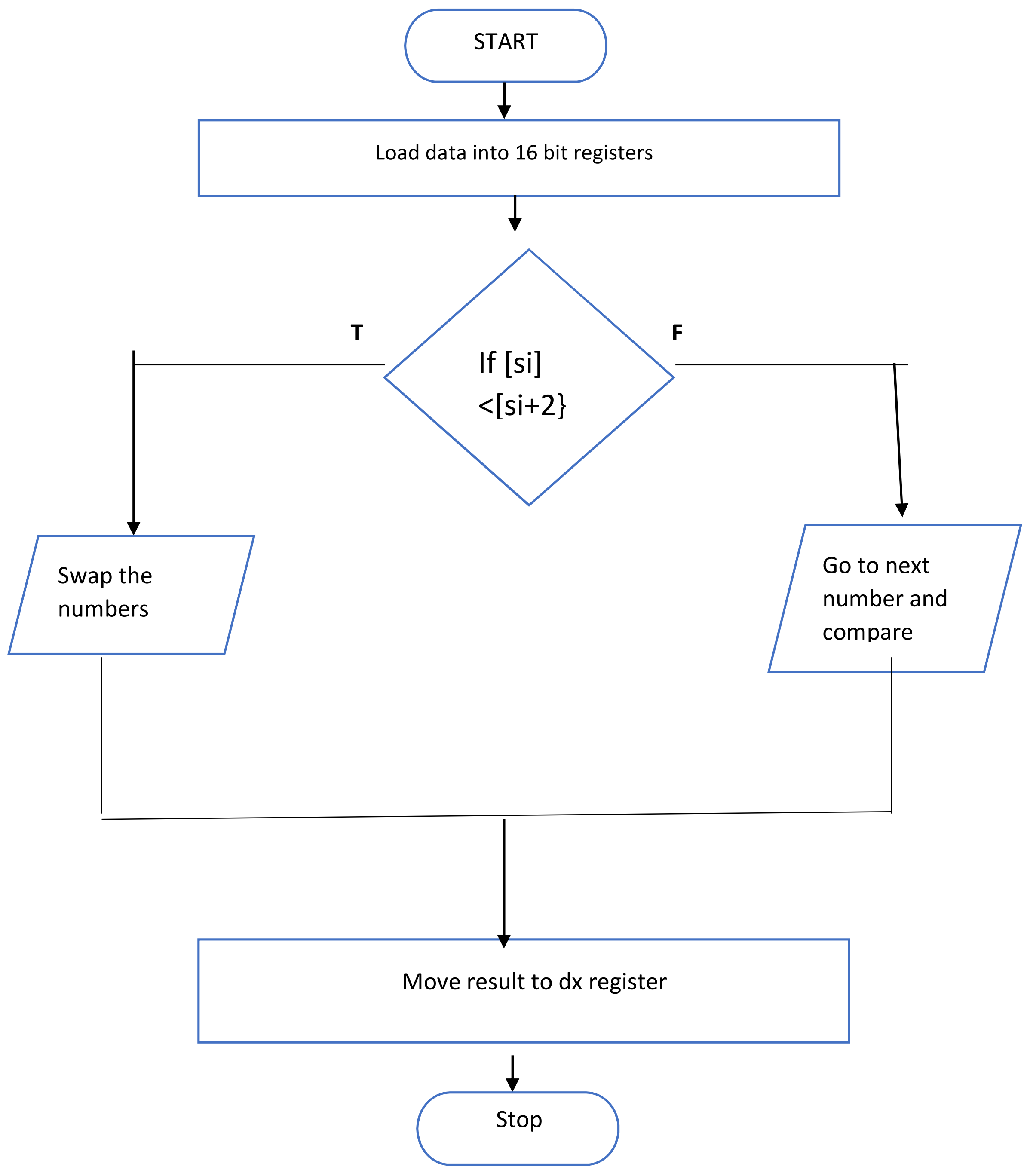
Transfer-Of-Control Instructions : -

9) **LOOP :-** The Loop instruction provides a simple way to repeat a block of statements a specific number of times. The Loop instructions use the CX register to indicate the loop count.
Flags affected :- None

Syntax :- LOOP label

Example :- LOOP 2050

Flowchart:


START
Load data into 16 bit registers
T
If [si] <[si+2}
F
Swap the numbers
Go to next number and compare
Move result to dx register
Stop

Code:

```
data segment
   string1 dw 14h,10h,5h,15h,21h
   data ends
   code segment
      assume cs:code,ds:data
      start:
```

```
     mov ax,data
     mov ds,ax
    mov bx,5
    up1: lea si,string1
    mov cx,4
    up: mov ax,[si]
    cmp ax,[si+2]
    jnc down

   xchg ax,[si+2]
   xchg ax,[si]
   down: add si,2
   loop up
   dec bx
   jnz up1
   int 3
    code ends
end start
```

Program:

MEMORY LOCATION	OP-CODE	LABEL	MNEMONIC
07110h 07111h 07112h	B8 10 07		MOV AX, DATA
07113h 07114hh	8E D8		MOV DS, AX
07115h 07116h 07117h	BB 05 00		MOV BX, 5
07118h 07119hh 0711Ah	BE 00 00	UP1	LEA SI, STRING1
0711Bh 0711Ch 0711Dh	B9 04 00		MOV CX, 4
0711Ehh 0711Fh	8B 04	UP	MOV AX, [SI]
07120h 07121h 07122h	3B 44 02		CMP AX, [SI + 2]
07123h 07124h	72 05		JNC DOWN

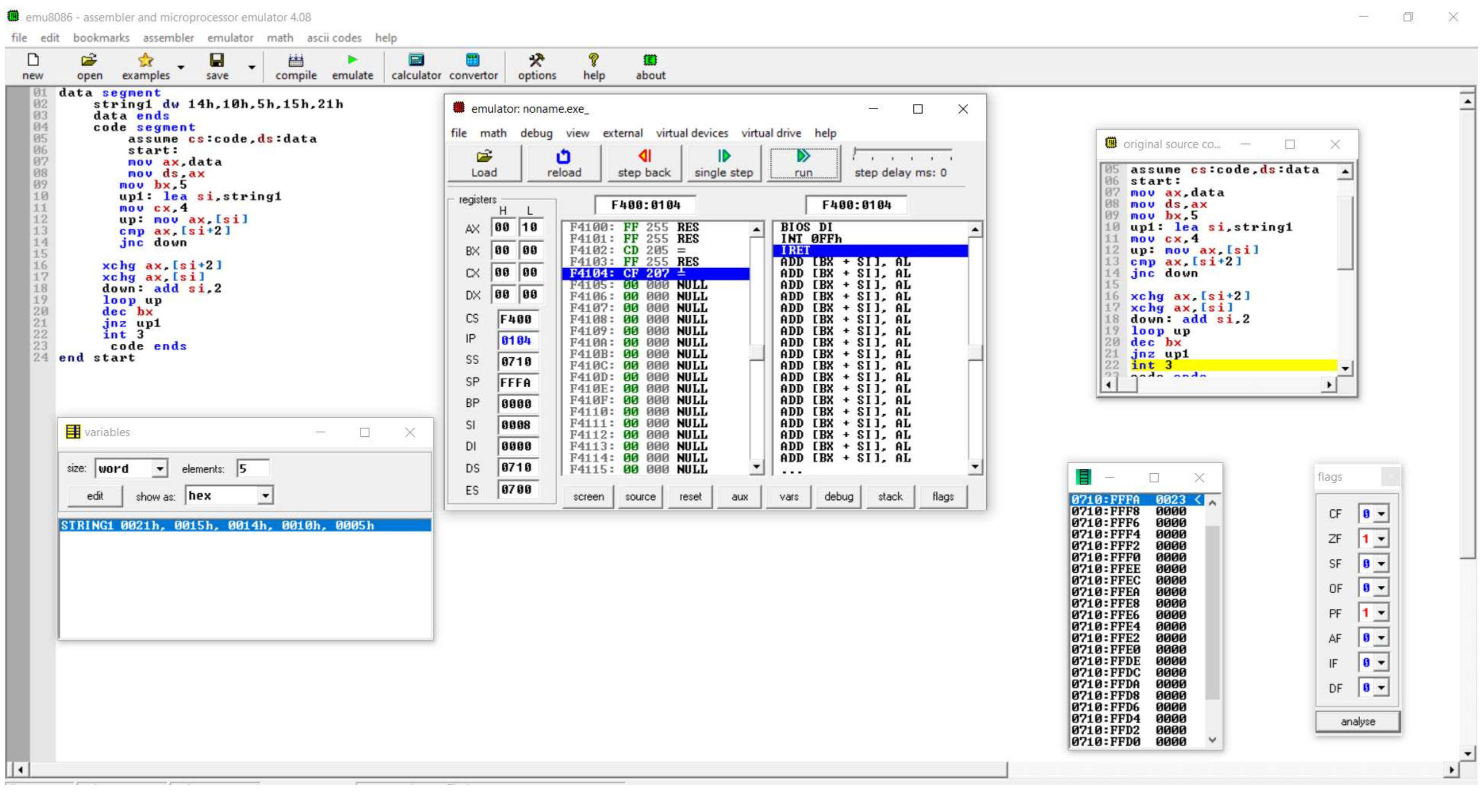
0712Ah 0712Bh 0712Ch	83 C6 02	DOWN	ADD SI, 2
0712Dh 0712Eh	E2 EF		LOOP UP
07125h 07126h 07127h	87 44 02		XCHG AX, [SI + 2]
07128h 07129h	87 04		XCHG AX, [SI]
0712Fh	4B		DEC BX
07130h 07131h	75 E6		JNZ UP1
07132h	CC		INT 3

Result:

Input		Output	
Register	**Data**	**Register**	**Data**
AX	0014h, 0010h, 0005h, 0015h, 0021h	AX	0021h, 0015h, 0014h, 0010h, 0005h

Flags Affected : Zero flag, Parity flag

Screenshot:



```
data segment
     string1 dw 14h,10h,5h,15h,21h
     data ends
     code segment
        assume cs:code,ds:data
        start:
        mov ax,data
        mov ds,ax
       mov bx,5
       up1: lea si,string1
       mov cx,4
       up: mov ax,[si]
       cmp ax,[si+2]
       jnc down

      xchg ax,[si+2]
      xchg ax,[si]
      down: add si,2
      loop up
      dec bx
      jnz up1
      int 3
       code ends
end start
```

```
size: word   elements: 5
show as: hex
STRING1 0021h, 0015h, 0014h, 0010h, 0005h
```

```
AX 00 10
BX 00 00
CX 00 00
DX 00 00
CS F400
IP 0104
SS 0710
SP FFFA
BP 0000
SI 0008
DI 0000
DS 0710
ES 0700
```

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

Conclusion : - Thus we have understood 8086 Assembly language programming for arranging 16 bit numbers in descending order.

Experiment no. 9

Aim: 8086 Assembly language programming for block transfer of 16 bit data.

Requirements: 8086 microprocessor kit/MASM ----1 2.RPS (+5V) ----1,]and Emu8086 Emulator

Theory:

Data Transfer Instructions : -

1) **MOV :-** MOV instruction is used to transfer byte or word from register to register, memory to register, register to memory or with immediate addressing.
   Flags affected :- None

   Syntax :- MOV destination, source

   Example :- MOV AX,BX

2) **LEA :-** LEA instruction computes the effective address of the second operand (source operand) and stores it in the first operand (destination operand).
   Flags affected :- None

   Syntax :- LEA destination, source

   Example :- LEA AX, BX

Transfer-Of-Control Instructions : -

3) **LOOP :-** The Loop instruction provides a simple way to repeat a block of statements a specific number of times. The Loop instructions use the CX register to indicate the loop count.
   Flags affected :- None

   Syntax :- LOOP label

   Example :- LOOP 2050

String Instructions : -

4) **MOVSB :-** The MOVSB (move string, byte) instruction fetches the byte at address SI, stores it at address DI and then increments or decrements the SI & DI registers by 1.
   Flags affected :- None

   Syntax :- LEA destination, source

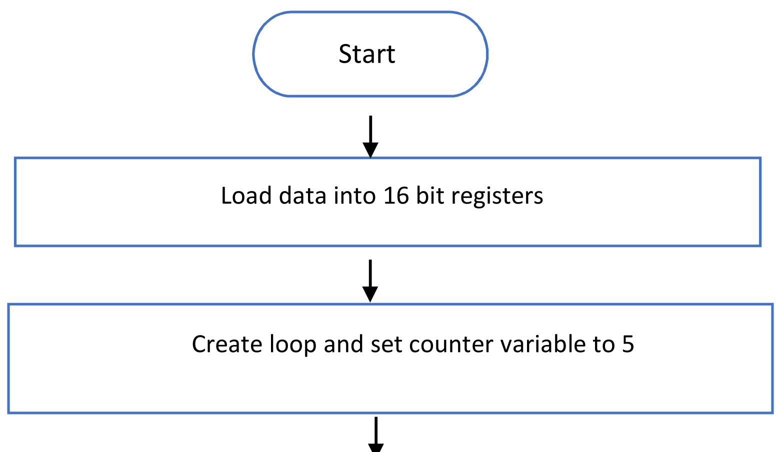
   Example :- MOV SI, OFFSET SOURCE
   MOV DI, OFFSET DESTINATION
   CLD
   MOV CX, 04H
   REP MOVSB

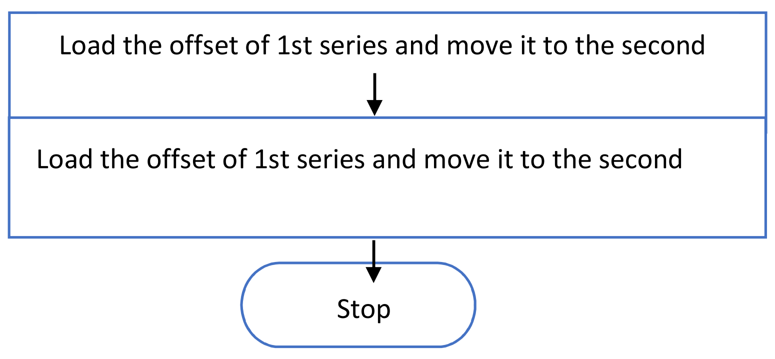
Miscellaneous Instructions : -

5) **INT** :- Used to interrupt the program during execution and calling service specified.

   Example:- INT 3 -> This is a special form, which has the single-byte code of CCH;

Flowchart:


Start
Load data into 16 bit registers
Create loop and set counter variable to 5


Load the offset of 1st series and move it to the second
Load the offset of 1st series and move it to the second
Stop

Code:

```
data segment
   string1 dw 0101h,0303h,0505h,0707h,0909h
   string2 dw 5DUP<0>
   data ends
code segment
   assume cs:code ds:data
   start:
   mov ax,data
   mov ds,ax
   mov es,ax
   mov cx,5
   lea si,string1
   lea di,string2

   up: movsw
   loop up
   int 3
   code ends
end start
```

Program :

MEMORY LOCATION	OP-CODE	LABEL	MNEMONIC
07110h 07111h 07112hh	B8 10 07		MOV AX, DATA
07113h 07114h	8E D8		MOV DX, AX
07115h 07116h	8E C0		MOV ES, AX
07118h 07119h	B1 05		MOV CX, 5
07110h 0711Ah 0711Bh	BE 00 00		LEA SI, STRING1
0711Ch 0711Dh 0711Eh	BF 05 00		LEA DI, STRING2
0711Fh	A4	UP	MOVSW
07120h 07121h	E2 FD		LOOP UP
07122h	CC		INT 3

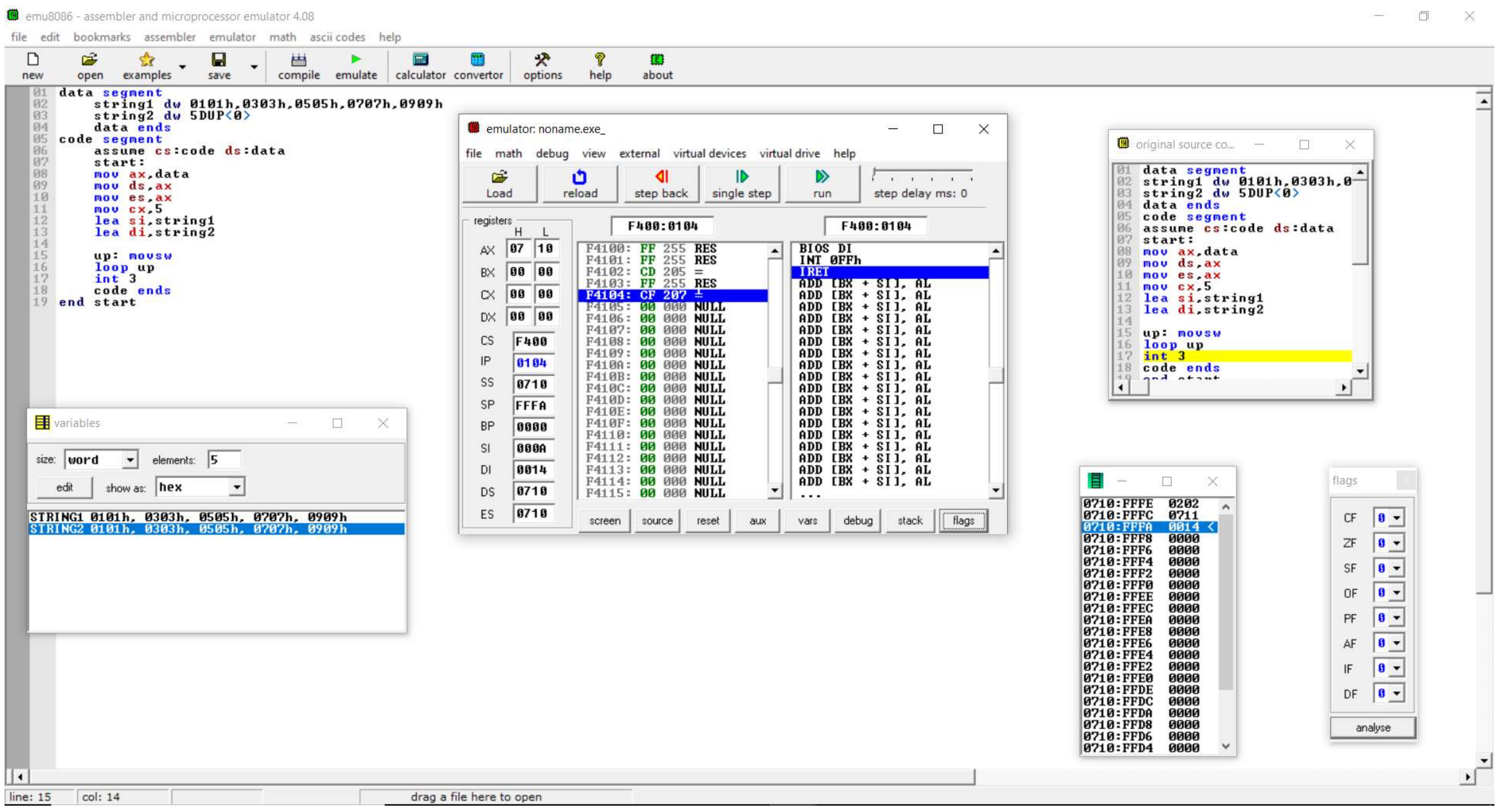
Result:

INPUT		OUTPUT	
REGISTER	**DATA**	**REGISTER**	**DATA**

SI	**0101h,0303h,0505h, 0707h,0909h**	**DI**	**0101h,0303h,0505h, 0707h,0909h**

Flag Affected : No flags affected

Screenshot :



```
data segment
     string1 dw 0101h,0303h,0505h,0707h,0909h
     string2 dw 5DUP(0)
     data ends
code segment
     assume cs:code ds:data
     start:
     mov ax,data
     mov ds,ax
     mov es,ax
     mov cx,5
     lea si,string1
     lea di,string2

     up: movsw
     loop up
     int 3
     code ends
end start
```

```
registers
     H   L
AX   07  10
BX   00  00
CX   00  00
DX   00  00
CS   F400
IP   0104
SS   0710
SP   FFFA
BP   0000
SI   000A
DI   0014
DS   0710
ES   0710
```

```
F400:0104                      F400:0104
F4100: FF 255 RES              BIOS DI
F4101: FF 255 RES              INT 0FFh
F4102: CD 205 =                IRET
F4103: FF 255 RES              ADD [BX + SI], AL
F4104: CF 207 ±                ADD [BX + SI], AL
F4105: 00 000 NULL             ADD [BX + SI], AL
F4106: 00 000 NULL             ADD [BX + SI], AL
F4107: 00 000 NULL             ADD [BX + SI], AL
F4108: 00 000 NULL             ADD [BX + SI], AL
F4109: 00 000 NULL             ADD [BX + SI], AL
F410A: 00 000 NULL             ADD [BX + SI], AL
F410B: 00 000 NULL             ADD [BX + SI], AL
F410C: 00 000 NULL             ADD [BX + SI], AL
F410D: 00 000 NULL             ADD [BX + SI], AL
F410E: 00 000 NULL             ADD [BX + SI], AL
F410F: 00 000 NULL             ADD [BX + SI], AL
F4110: 00 000 NULL             ADD [BX + SI], AL
F4111: 00 000 NULL             ADD [BX + SI], AL
F4112: 00 000 NULL             ADD [BX + SI], AL
F4113: 00 000 NULL             ADD [BX + SI], AL
F4114: 00 000 NULL             ADD [BX + SI], AL
F4115: 00 000 NULL             ...
```

size: word   elements: 5   show as: hex

```
STRING1 0101h, 0303h, 0505h, 0707h, 0909h
STRING2 0101h, 0303h, 0505h, 0707h, 0909h
```

```
0710:FFFE  0202
0710:FFFC  0711
0710:FFFA  0014 <
0710:FFF8  0000
0710:FFF6  0000
0710:FFF4  0000
0710:FFF2  0000
0710:FFF0  0000
0710:FFEE  0000
0710:FFEC  0000
0710:FFEA  0000
0710:FFE8  0000
0710:FFE6  0000
0710:FFE4  0000
0710:FFE2  0000
0710:FFE0  0000
0710:FFDE  0000
0710:FFDC  0000
0710:FFDA  0000
0710:FFD8  0000
0710:FFD6  0000
0710:FFD4  0000
```

flags

```
CF 0
ZF 0
SF 0
OF 0
PF 0
AF 0
IF 0
DF 0
```

line: 15   col: 14   drag a file here to open

emu8086 - assembler and microprocessor emulator 4.08
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
emulator: noname.exe_
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
screen source reset aux vars debug stack flags
variables
edit
original source co...
analyse

Conclusion : - Thus we have understood 8086 Assembly language programming for block transfer of 16 bit data.