FS19CO042

FS19CO042 First shift

Omkar Phansopkar

MP MANUALS

Exp 1 to 9

**Experiment No 1**

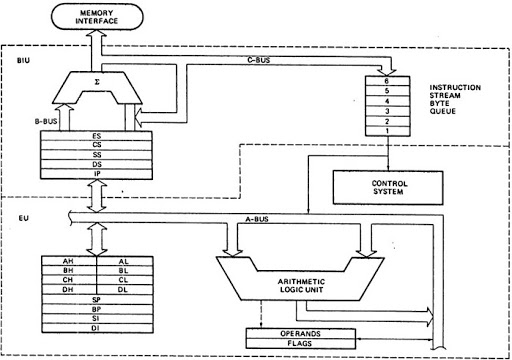
**Aim :** Understand 8086 Development Board and Simulation Software

**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.

1. DS (Data Segment): The user can modify the content of the data segment.
2. 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.

1. 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

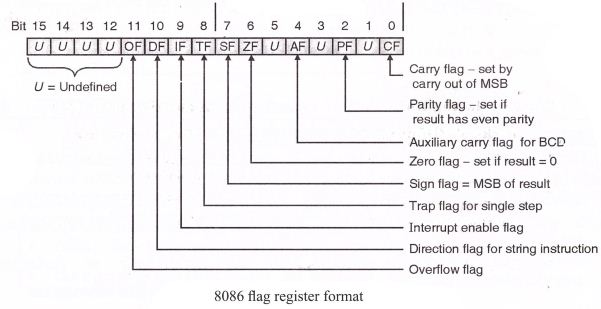
stack pointer will be (FFFE)H. It’s offset address relative to stack segment

* 1. **BP:** This is the base pointer. It is of 16 bits. It is primary used in accessing parameters passed by the stack. It’s

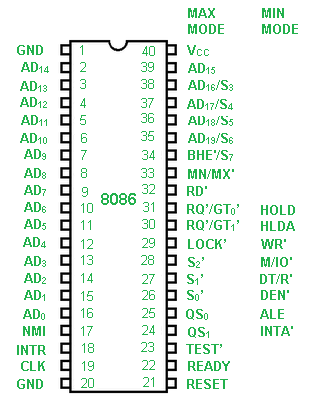
offset address relative to stack segment.

* 1. **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.
  2. **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.
* Flag or Status Registers

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.



Pin Diagram of 8086 Microprocessor:



Signals:

**Power supply and frequency signals**

It uses 5V DC supply at VCC pin 40, and uses ground at VSS 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.

**QS1 and QS0**

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 −

|  |  |  |
| --- | --- | --- |
| QS0 | QS1 | 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 |

**S0, S1, S2**

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 –

|  |  |  |  |
| --- | --- | --- | --- |
| S2 | S1 | S0 | 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/GT1 and RQ/GT0**

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/GT0 has a higher priority than RQ/GT1.

**Instruction Set Of 8086:**

## **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.

## **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.

## **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.

## **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.

## **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

## **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.

## **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

## **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

1. **Arithmetic instructions** 
   1. **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

* 1. **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

Stop

Store result into dx register

Add data

**Flowchart for 16 bit numbers subtraction**

START

Load data into 16 bit registers

Stop

Store result into dx register

Subtract data

**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**  **O7111H**  **O7112H**  **07113H**  **07114H**  **07115H**  **07116H**  **07117H**  **07118H**  **07119H**  **0711AH**  **0711BH**  **0711CH**  **0711DH**  **0711EH**  **0711FH**  **07120H**  **07121H**  **07122H**  **07123H**  **07124H** | **B8**  **10**  **07**  8E  D8  A1  00  00  8B  1E  02  00  03  C3  8B  C8  89  0E  04  00  CC |  | **mov ax,data**    **mov ds,ax**      **mov ax,a**    **mov bx,b**      **add ax,bx**      **mov cx,ax**    **mov c,cx**    **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**  **07111H**  **07112H**  **07113H**  **07114H**  **07115H**  **07116H**  **07117H**  **07118H**  **07119H**  **0711AH**  **0711BH**  **0711CH**  **0711DH**  **0711EH**  **0711FH**  **07120H**  **07121H**  **07122H**  **07123H**  **07124H** | **B8**  **10**  07  8E  D8  A1  00  00  8B  1E  02  00  2B  C3  8B  C8  89  0E  04  00  CC |  | **mov ax,data**    **mov ds,ax**      **mov ax,a**    **mov bx,b**    **sub ax,bx**    **mov cx,ax**    **mov c,cx**    **int 3** |

**Result :**

**Output OF 16 BIT ADDITION :**

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

**Affected flags: Parity flag**

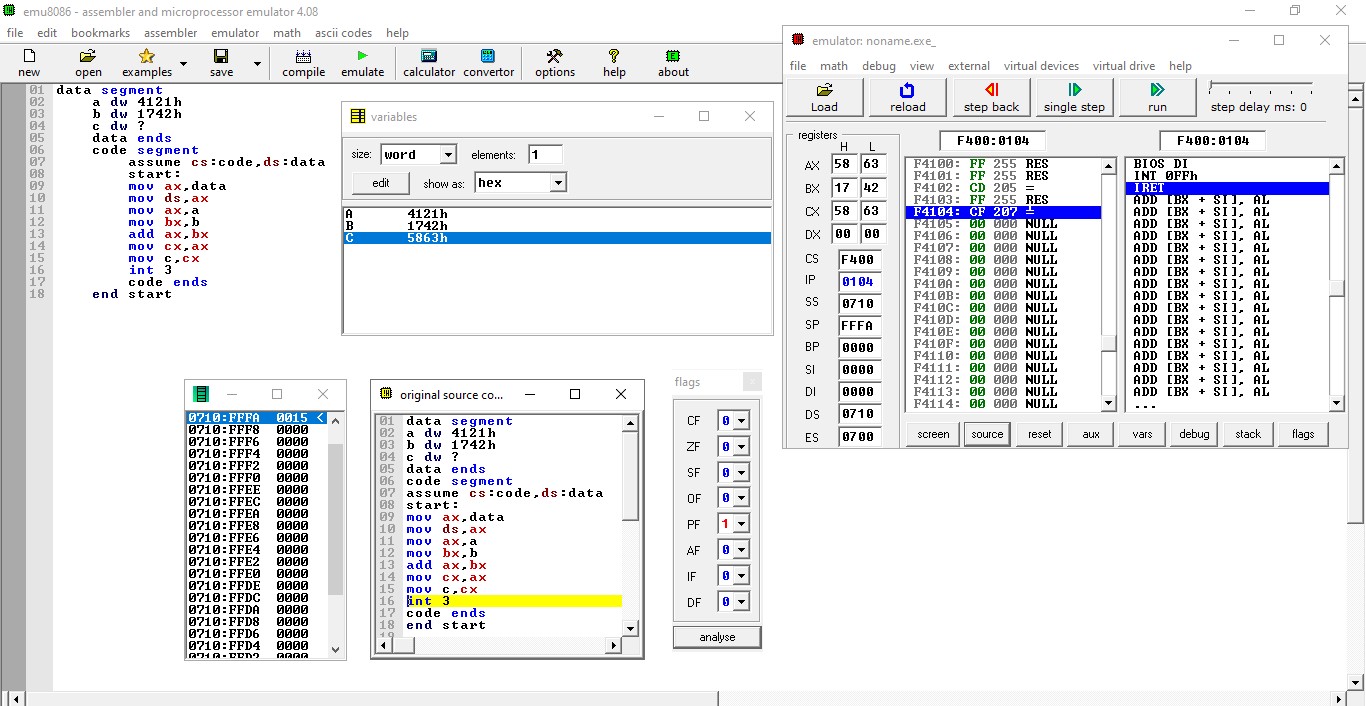
**Output OF 16 BIT SUNTRACTION :**

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

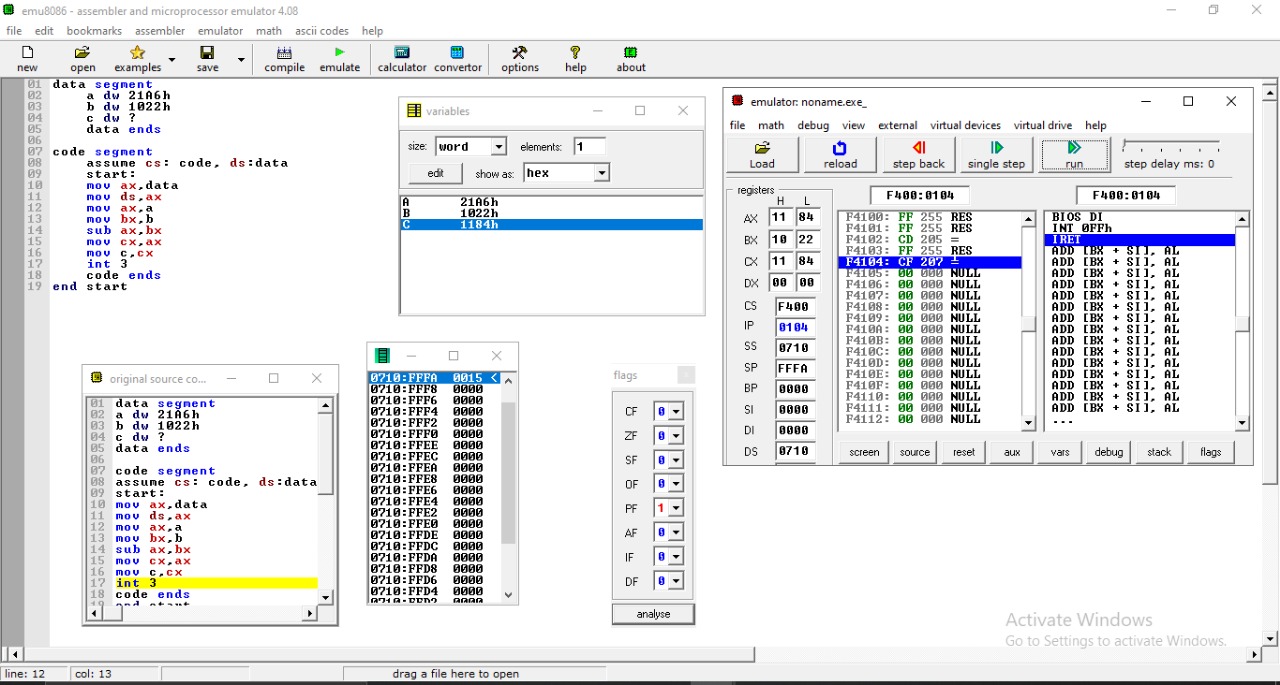
**Affected flags: Parity flag**

**SCREENSHOT :**

**16 Bit Addition :**

****

**16 bit subtraction :**

****

**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

* 1. **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

* 1. **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

1. **Arithmetic instructions** 
   1. **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

* 1. **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

1. **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

1. **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**  **07111h**  **07112h**  **07113h**  **07114h**  **07115h**  **07116h**  **07117h**  **07118h**  **07119h**  **0711Ah**  **0711Bh**  **0711Ch**  **0711Dh**  **0711Eh**  **0711Fh**  **07120h**  **07121h**  **07122h**  **07123h**  **07124h**  **07125h**  **07126h**  **07127h**  **07128h**  **07129h**  **0712Ah**  **0712Bh**  **0712Ch** | **B8**  **8E**  **D8**  **B1**  **05**  **BB**  **00**  **00**  **B8**  **00**  **00**  **03**  **07**  **83**  **C3**  **02**  **FE**  **C9**  **80**  **F9**  **00**  **75**  **F4**  **A3**  **0A**  **00**  **CC** | **OP:** | **mov ax,data**    **mov ds,ax**  **mov cl,5**  **lea bx,a**  **mov ax,00**  **add ax,word ptr[bx]**    **add bx,02**    **dec cl**  **cmp cl,00**    **jnz OP**    **mov b,ax**    **int 3** |

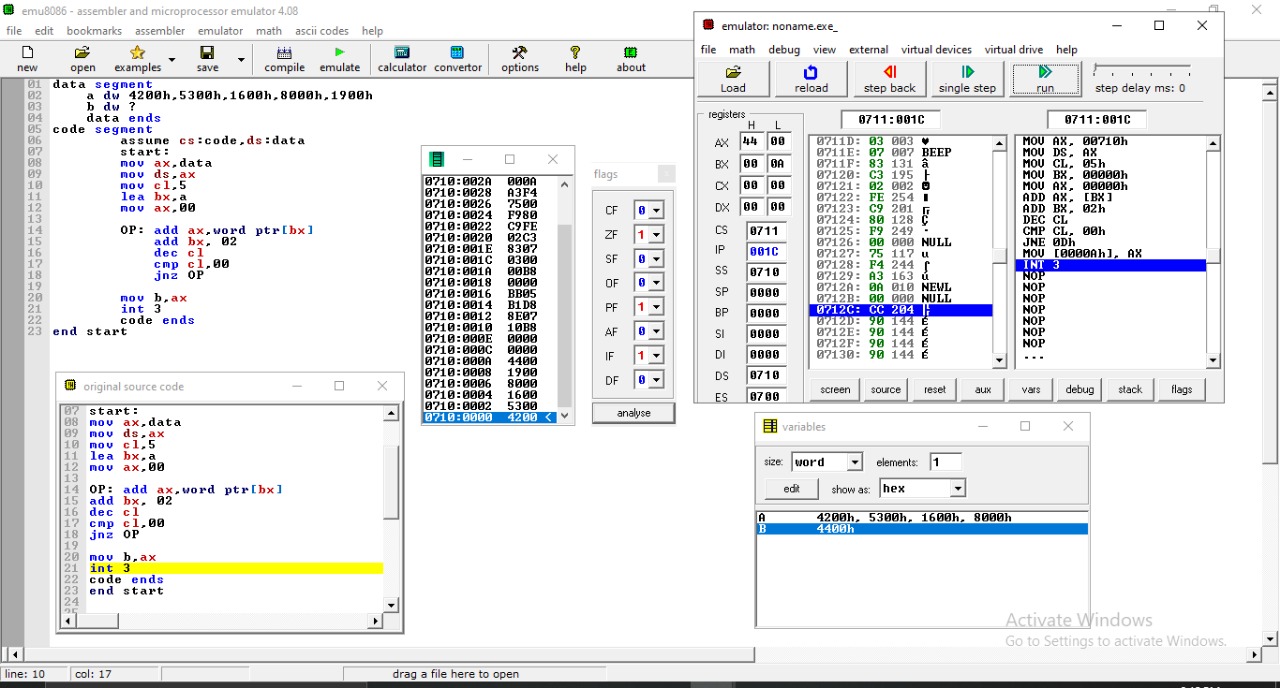
**Result :**

**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 :**

****

**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

* 1. **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

1. **Arithmetic instructions** 
   1. **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

* 1. **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

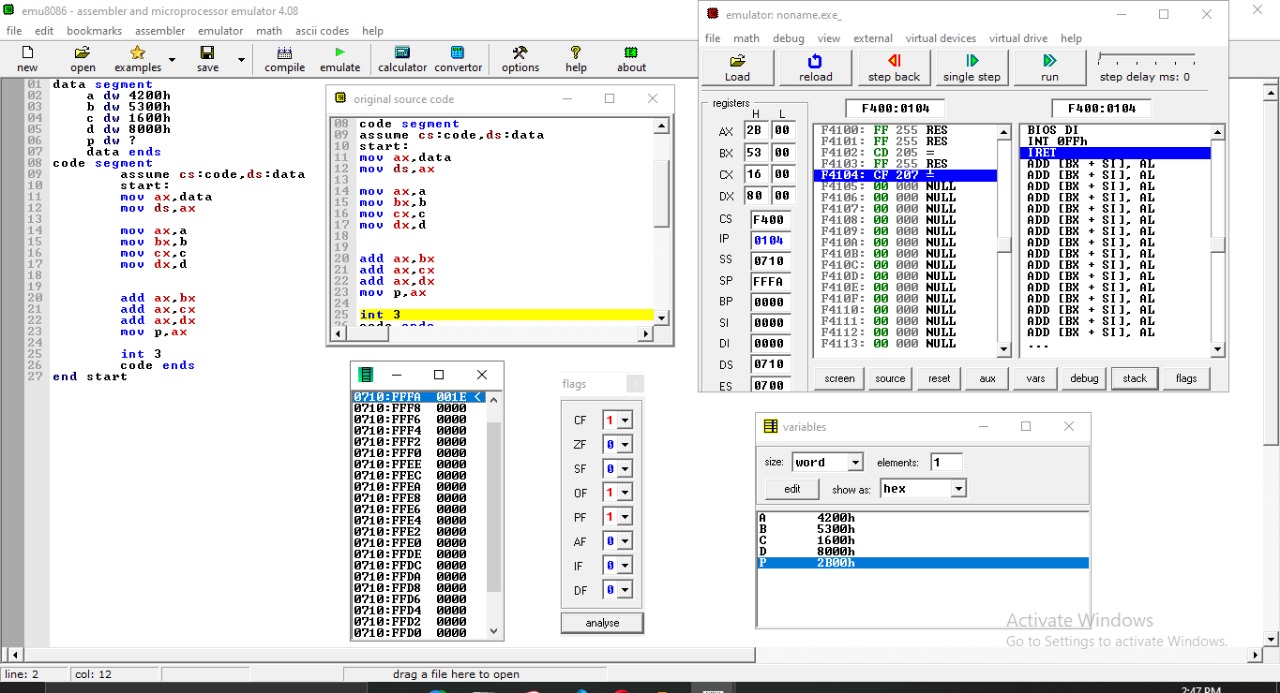
|  |  |  |  |
| --- | --- | --- | --- |
| **MEMORY LOCATION** | **OP-CODE** | **LABEL** | **MNEMOIC** |
| **7110h**  **7111h**  **7112h** | **B8**  **10**  **07** |  | **mov ax,data** |
| **7113h**  **7114h** | **8E**  **D8** |  | **mov ds,ax** |
| **7115h**  **7116h**  **7117h** | **A1**  **00**  **00** |  | **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:**

****

**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

* 1. **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

1. **Arithmetic instructions** 
   1. **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

* 1. **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

1. 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  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  E3 | - | MUL BX |
| 0711E  0711Fh  07120h  07121h | 89  16  04  00 | - | MOV C, DX |
| 07122h  07123h  07124h | A3  06  00 | - | MOV D, AX |
| 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 |

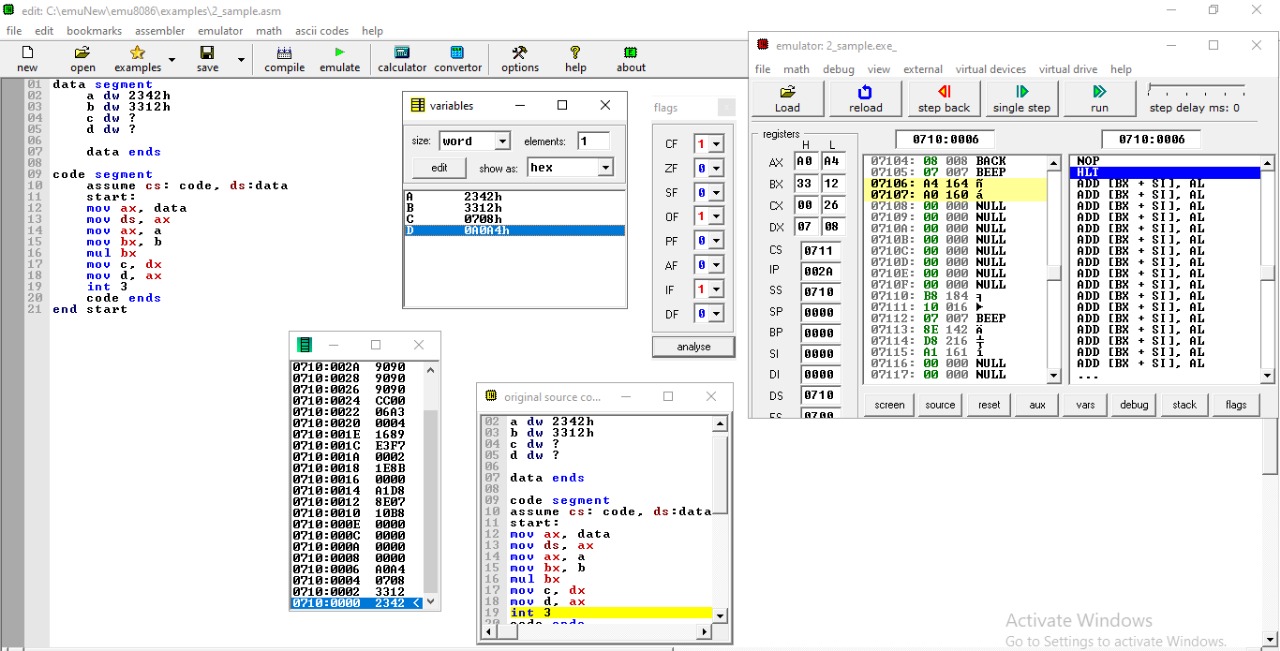
Result:

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

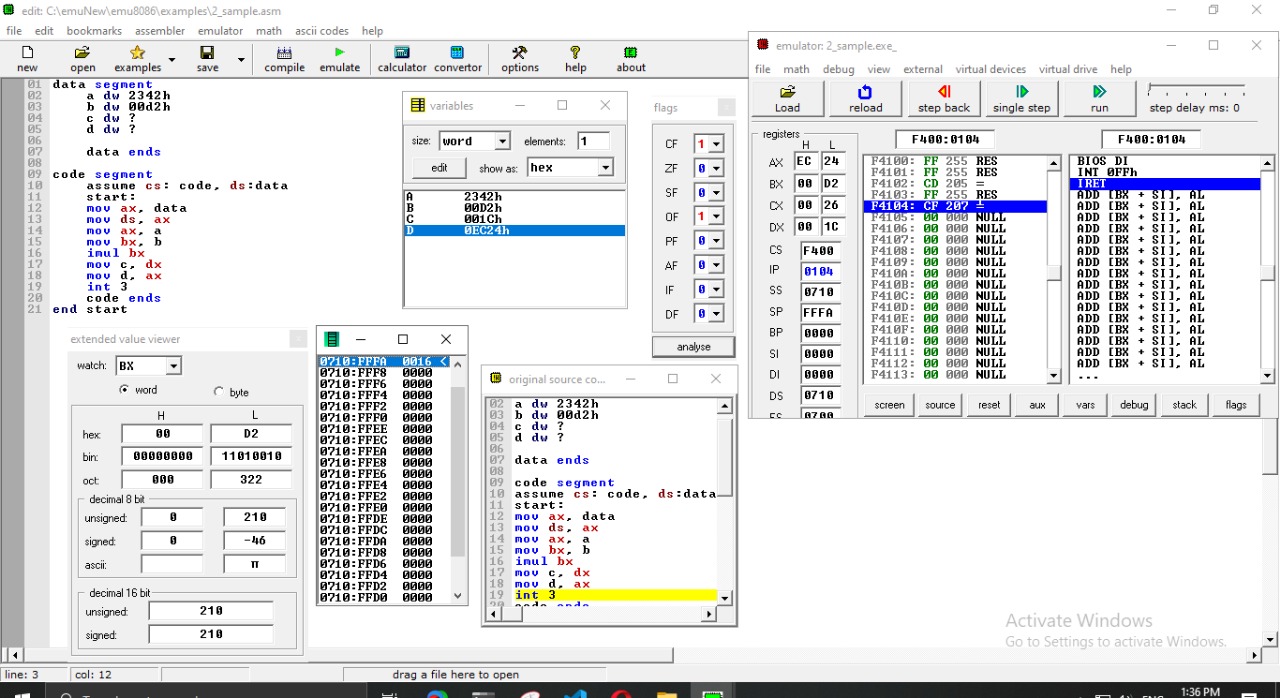
Flags Affected : Carry flag, Overflow flag

**Screenshots:**

**Unsigned multiplication:**

****

**Signed multiplication:**

****

**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

* 1. **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

1. **Arithmetic instructions** 
   1. **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

* 1. **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 |
| 7110  7111  7112 | B8  10  07 | - | MOV AX, DATA |
| 7113  7114 | 8E  D8 | - | MOV DS, AX |
| 7115  7116  7117 | A1  00  00 | - | MOV AX, A |
| 7118  7119  711A  711B | 8B  1E  02  00 | - | MOV BX, B |
| 711C  711D | F7  FB | - | DIV BX |
| 711E  711F | 8B  C8 | - | MOV CX, AX |
| 7120  7121  7122  7123 | 89  0E  04  00 | - | MOV C, CX |
| 7124  7125  7126  7127 | 89  16  06  00 | - | MOV D, DX |
| 7128 | 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 |
| 7110  7111  7112 | B8  10  07 | - | MOV AX, DATA |
| 7113  7114 | 8E  D8 | - | MOV DS, AX |
| 7115  7116  7117 | A1  00  00 | - | MOV AX, A |
| 7118  7119  711A  711B | 8B  1E  02  00 | - | MOV BX, B |
| 711C  711D | F7  FB | - | IDIV BX |
| 711E  711F | 8B  C8 | - | MOV CX, AX |
| 7120  7121  7122  7123 | 89  0E  04  00 | - | MOV C, CX |
| 7124  7125  7126  7127 | 89  16  06  00 | - | MOV D, DX |

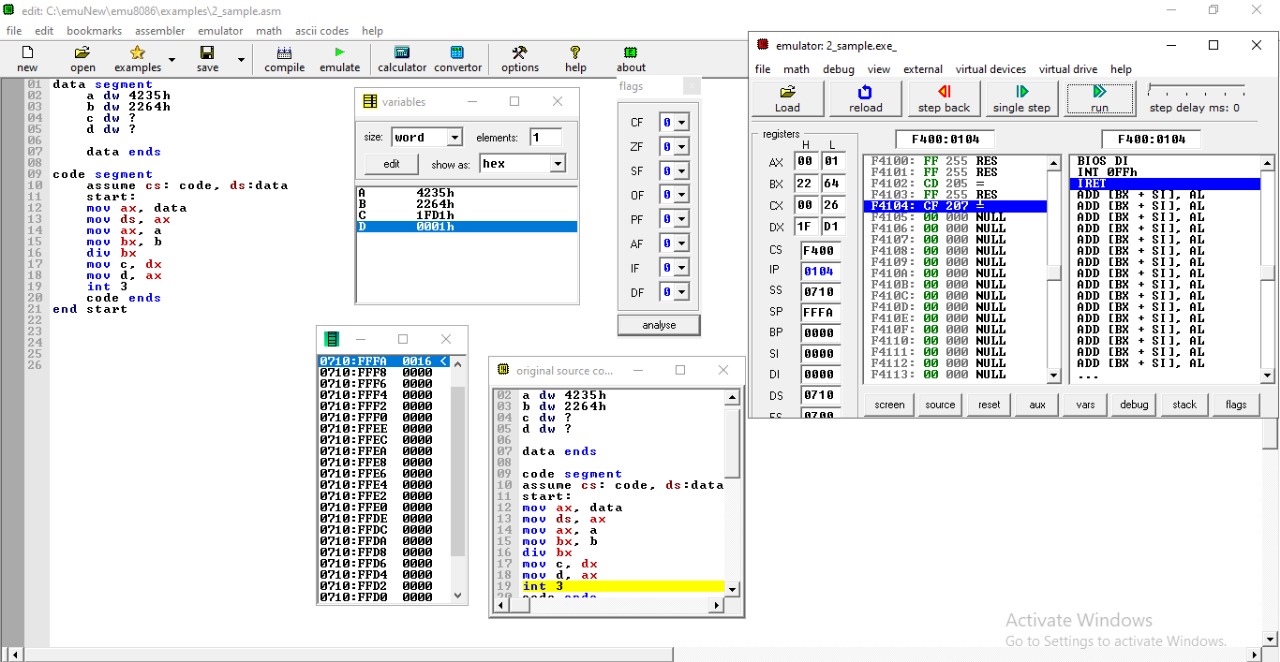
Result :

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

Flags Affected : Interrupt Flag

Output:

Unsigned division:



Signed division: 