TPCT's

College of Engineering, Osmanabad

# Laboratory Manual

**Computer Science And Engineering**

**For**

**Second Year Students**

**Manual Prepared by**

**R.B.Randive**

**Author COE, Osmanabad**

**TPCT's**

**College of Engineering**
**Solapur Road, Osmanabad**
**Department Computer Science And Engineering**

## Vision of the Department:

To produce trained computer professionals who can successfully meet the demands of academia, IT Industry and research by building a strong teaching and research environment.

## Mission of the Department:

To provide industry and research oriented quality education to UG and PG students and train them to apply this knowledge for solving real world problems and make them competitive in the ever-changing and challenging global work environment

# College of Engineering

## Technical Document

This technical document is a series of Laboratory manuals of Computer Science And Engineering Department and is a certified document of College of engineering, Osmanabad. The care has been taken to make the document error-free. But still if any error is found, kindly bring it to the notice of subject teacher and HOD.

Recommended by,

HOD

Approved by,

Principal

# FOREWORD

It is my great pleasure to present this laboratory manual for second year engineering students for the subject of Microprocessor and Computer Organization to understand and visualize the basic concepts of microprocessor using 8086 emulator. Microprocessor and Computer Organization cover basic concepts of Microprocessor. This being a core subject, it becomes very essential to have clear theoretical and designing aspects.

This lab manual provides a platform to the students for understanding the basic concepts of Microprocessor and Computer Organization. This practical background will help students to gain confidence in qualitative and quantitative approach to Microprocessor.

H.O.D
CSE Dept

## LABORATORY MANUAL CONTENTS

This manual is intended for the Second Year students of CSE branches in the subject of Microprocessor and Computer Organization. This manual typically contains practical/ Lab Sessions related to Microprocessor and Computer Organization covering various aspects related to the subject for enhanced understanding.

Students are advised to thoroughly go through this manual rather than only topics mentioned in the syllabus as practical aspects are the key to understanding and conceptual visualization of theoretical aspects covered in the books.

SUBJECT INDEX:

## 1.Dos and Don'ts in Laboratory :-

1. Do not handle any equipment before reading the instructions /Instruction manuals.
2. Read carefully the power ratings of the equipment before it is switched ON, whether ratings 230 V/50 Hz or 115V/60 Hz. For Indian equipment, the power ratings are normally 230V/50Hz. If you have equipment with 115/60 Hz ratings, do not insert power plug, as our normal supply is 230V/50Hz., which will damage the equipment.
3. Observe type of sockets of equipment power to avoid mechanical damage.
4. Do not forcefully place connectors to avoid the damage.
5. Strictly observe the instructions given by the Teacher/ Lab Instructor.

## Instruction for Laboratory Teachers:-

1. Submission related to whatever lab work has been completed should be done during the next lab session.
2. Students should be instructed to switch on the power supply after getting the checked by the lab assistant / teacher. After the experiment is over, the students must hand over the circuit board, wires, CRO probe to the lab assistant/teacher.
3. The promptness of submission should be encouraged by way of marking and evaluation patterns that will benefit the sincere students.

## 2.Lab Exercises

1. Write assembly language program to perform 8 bit and 16 bit addition.
2. Write assembly language program to perform 8 bit and 16 bit subtraction.
3. Write assembly language program to perform 8 bit and 16 bit multiplication.
4. Write assembly language program to perform 8 bit and 16 bit division.
5. Write assembly language program to perform 8 bit and 16 bit AND ing.
6. Write assembly language program to perform 8 bit and 16 bit OR ing.
7. Write assembly language program to perform 8 bit and 16 bit XOR ing.
8. Write assembly language program to check whether entered number is even or odd
9. Write assembly language program to perform conversion from ASCII number to packed BCD.
10. Write assembly language program to calculate of temperature.
11. Study of BIOS and DOS interrupts.
12. Study of TSR.

# Experiment No.1
## Addition

**Aim:-** Write assembly language program to perform 8 bit and 16 bit addition

**Objective:** To add 8 bit and 16 bit binary numbers using addition rules for binary arithmetic instruction.

**Software**: 8086 Emulator

**Theory:** The 8086 has four groups of the user accessible internal registers. They are

1. general purpose registers
2. Segment registers
3. pointer and index registers
4. Flag register

**General Purpose Registers**

- **AX : Accumulator register consists of two 8-bit registers AL** and AH, which can be combined together and used as a 16- bit register AX. AL in this case contains the low-order byte of the word, and AH contains the high-order byte. Accumulator can be used for I/O operations and string manipulation.
- BX: **Base** register consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX. BL in this case contains the low-order byte of the word, and BH contains the high-order byte. BX register usually contains a data pointer used for based, based indexed or register indirect addressing.
- CX: **Count** register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX. When combined, CL register contains the low-order byte of the word, and CH contains the highorder byte. Count register can be used in Loop, shift/rotate instructions and as a counter in string manipulation,.
- **DX: Data** register consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX. When combined, DL register contains the low-order byte of the word, and DH contains the highorder byte. Data register can be used as a port number in I/O operations. In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

**Segment register:**

- **Code segment (CS)** is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.
- **Stack segment** (SS) is a 16-bit register containing address of 64KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction.
- **Data segment** (DS) is a 16-bit register containing address of 64KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX,

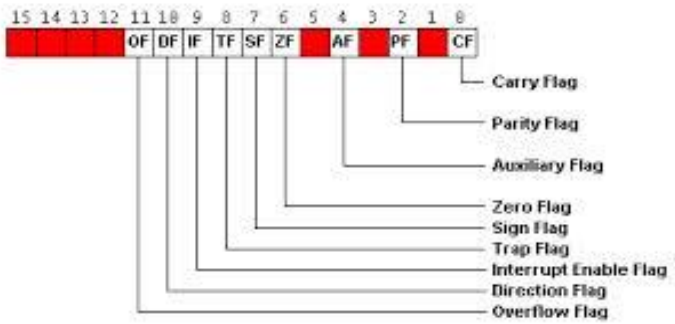
BX, CX, DX) and index register (SI, DI) is located in the data segment. DS register can be changed directly using POP and LDS instructions.

- **Extra segment** (ES) is a 16-bit register containing address of 64KB segment, usually with program data. By default, the processor assumes that the DI register references the ES segment in string manipulation instructions. ES register can be changed directly using POP and LES instructions

**Pointer and Index Registers**

- **Instruction Pointer** (IP) is a 16-bit register that contains the offset address. IP is combined with the CS to generate the address of the next instruction to be executed.
- **Stack Pointer (SP) is a 16-bit register pointing to program** stack.
- **Base Pointer** (BP) is a 16-bit register pointing to data in stack segment. BP register is usually used for based, based indexed or register indirect addressing.
- **Source Index** (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data address in string manipulation instructions.
- **Destination Index** (DI) is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data address in string manipulation instructions.

**Flag Register**




**Flags** is a 16-bit register containing nine 1-bit flags. 06 flags are status flags and 3 are Control Flags

- **Overflow Flag (OF)** - set if the result is too large positive number, or is too small negative number to fit into destination operand.
- **Direction Flag (DF)** - if set then string manipulation instructions will auto-decrement index registers. If cleared then the index registers will be auto-incremented.
- **Interrupt-enable Flag (IF)** - setting this bit enables maskable interrupts.
- **Single-step Flag (TF)** - if set then single-step interrupt will occur after the next instruction.
- **Sign Flag (SF)** - set if the most significant bit of the result is set.
- **Zero Flag (ZF)** - set if the result is zero.
- **Auxiliary carry Flag (AF)** - set if there was a carry from or borrow to bits 0-3 in the AL register.
- **Parity Flag (PF)** - set if parity (the number of "1" bits) in the low-order byte of the result is even.
- **Carry Flag (CF)** - set if there was a carry from or borrow to the most significant bit

during last result calculation.

**Data Transfer Instructions**

Data transfer is one of the most common tasks when programming in an assembly language. Data can be transferred between registers or between registers and the memory. Immediate data can be loaded to registers or to the memory. The transfer can be done on an octet or word. The two operands must have the same size. Data transfer instructions don't affect the condition indicators (excepting the ones that have this purpose). They are classified as follows:

6. classical transfer instructions
7. address transfer instructions
8. condition indicator transfer instructions
9. input/output instructions (peripheral register transfers)

One of the Classical transfer instructions Include the following instruction:

**MOV <d>, <s>**

The MOV instruction is used to transfer a byte or a word of data from a source operand to a destination operand. These operands can be internal registers of the 8086 and storage locations in memory.

| Mnemonic | Meaning | Format | Operation | Flags affected |
|---|---|---|---|---|
| MOV | Move | MOV D,S | (S) → (D) | None |

| **Destination** | **Source** | **Example** |
|---|---|---|
| Accumulator | Memory | MOV AX, TEMP |
| Register | Register | MOV AX, BX |
| Memory | Register | MOV COUNT [DI], CX |
| Register | Immediate | MOV CL, 04 |

**Arithmetic Instructions : Addition**

**ADD – ADD Destination, Source**

**ADC – ADC Destination, Source**

These instructions add a number from some *source* to a number in some *destination* and put the result in the specified destination. The ADC also adds the status of the carry flag to the result. The source may be an immediate number, a register, or a memory location. The destination may be a register or a memory location. The source and the destination in an instruction cannot both be memory locations. The source and the destination must be of the same type (bytes or words). If you want to add a byte to a word, you must copy the byte to a word location and fill the upper byte of the word with 0's before adding. Flags affected: AF, CF, OF, SF, ZF.

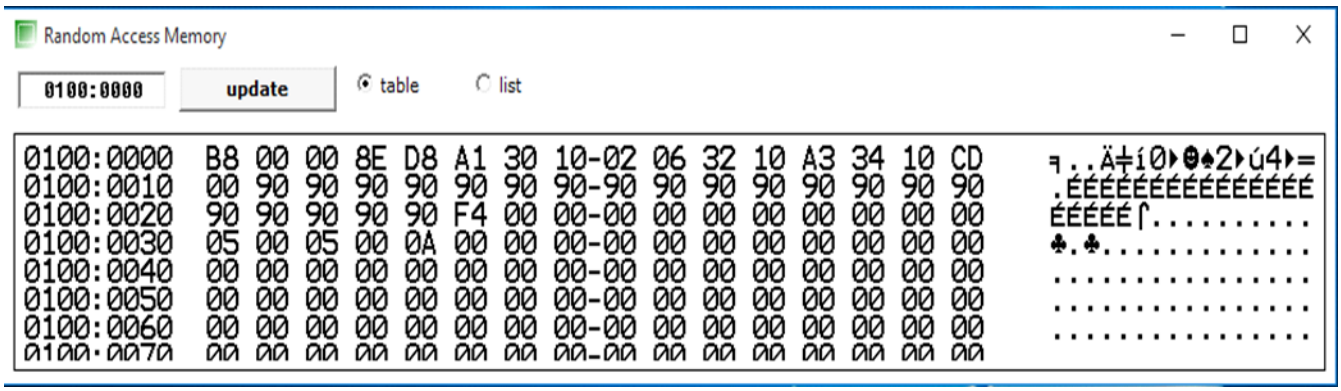
 ADD AL, 74H ; Add immediate number 74H to content of AL. Result in AL

 ADC CL, BL ;Add content of BL plus carry status to content of CL(CL = CL+BL+Carry Flag)
 ADD DX, BX ; Add content of BX to content of DX
 ADD DX, [SI] ;Add word from memory at offset [SI] in DS to content of DX

**Program:-** /* 8 BIT ADDITION */

```
MOV AX,0000H
MOV DS,AX
MOV AX,[1030H]
ADD AL,[1032H]
MOV [1034H],AX
INT
```

**Observations:-**




**Rules for Addition (8 bit and 16 bit addition)**

| A(i/p) | B(i/p) | Y(o/p) | Carry(o/p) |
|---|---|---|---|
| 0 | 0 | 0 | - |
| 0 | 1 | 1 | - |
| 1 | 0 | 1 | - |
| 1 | 1 | 0 | 1 |

**Result: -**

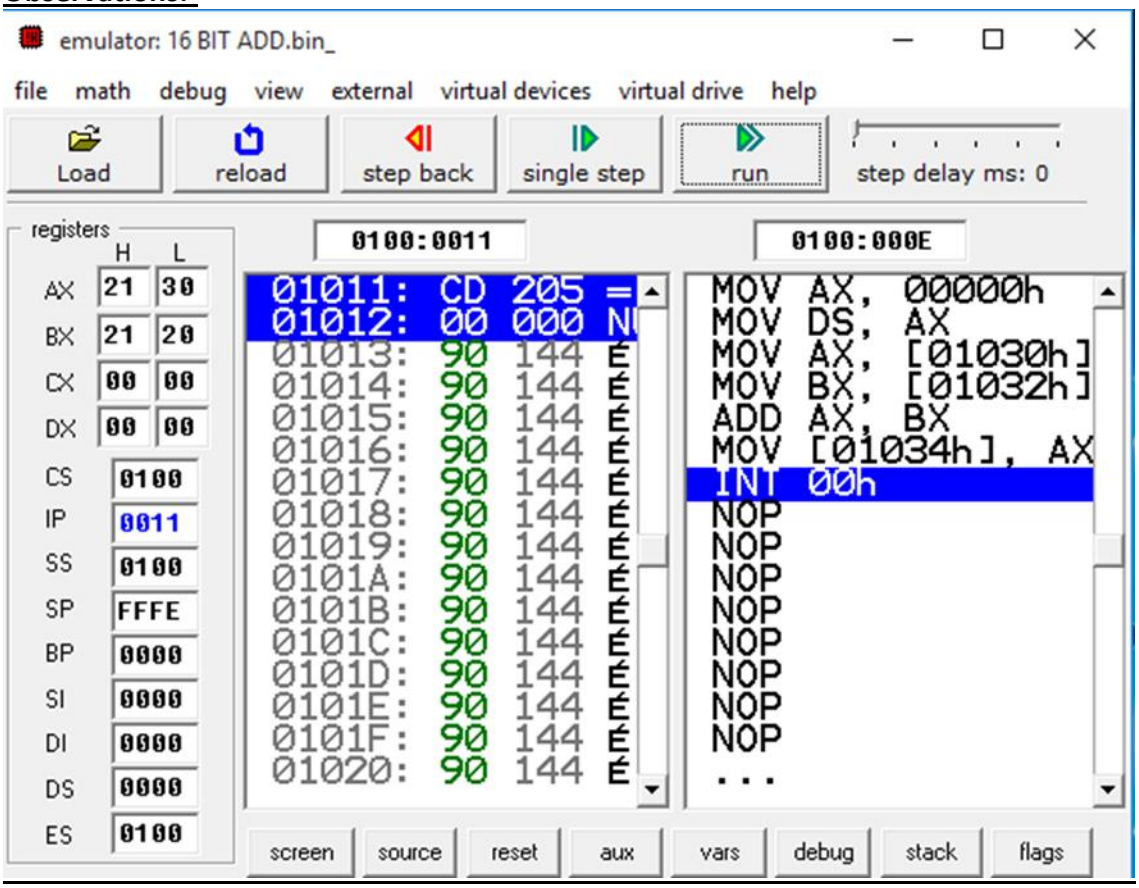
**[1030H]=05**
**[1032H]=05**
**[1034H]=0A.**

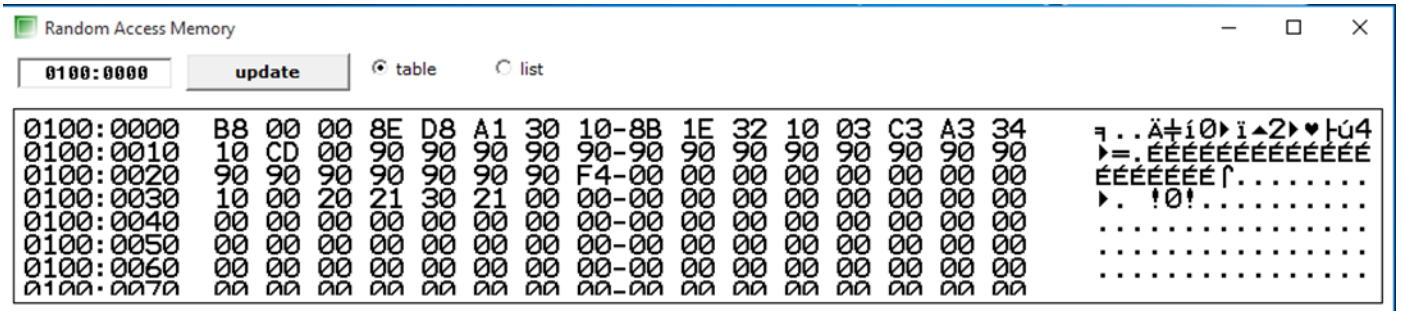
**Program:-** /* 16 BIT ADDITION */

```
MOV AX,0000H
MOV DS,AX
MOV AX,[1030H]
MOV BX,[1032H]
```

ADD AX,BX
MOV [1034H],AX
INT

**Observations:-**




**Result: -**

**[1030H]=1000**

**[1032H]=2021**

**[1034H]=3021**

**Conclusion:**

The internal registers along with FLAG register is understood and 8-bit and 16-bit addition is implemented.

# Experiment No.2
## Subtraction

**Aim:-** Write assembly language program to perform 8 bit and 16 bit subtraction

**Objective:** To subtract 8 bit and 16 bit binary numbers using subtraction rules for binary arithmetic instruction.

**Software**: 8086 Emulator

**Theory: 8086 ADDRESSING MODES**

**Immediate addressing mode:**
In this mode, 8 or 16 bit data can be specified as part of the instruction. OP Code Immediate Operand
Example 1 : MOV CL, 03 H
Moves the 8 bit data 03 H into CL

Example 2 : MOV DX, 0525 H
Moves the 16 bit data 0525 H into DX

In the above two examples, the source operand is in immediate mode and the destination operand is in register mode. A constant such as "VALUE" can be defined by the assembler EQUATE directive such as VALUE EQU 35H
Example : MOV BH, VALUE
Used to load 35 H into BH

**Register addressing mode :**
The operand to be accessed is specified as residing in an internal register of 8086.Internal registers can be used as a source or destination operand, however only the data registers can be accessed as either a byte or word.

Example 1 : MOV DX (Destination Register) , CX (Source Register)
Which moves 16 bit content of CS into DX.

Example 2 : MOV CL, DL
Moves 8 bit contents of DL into CL

MOV BX, CH is an illegal instruction.

- The register sizes must be the same.

**Direct addressing mode :**
The instruction Opcode is followed by an affective address, this effective address is directly used as the 16 bit offset of the storage location of the operand from the location specified by the current value in the selected segment register. The default segment is always DS.

The 20 bit physical address of the operand in memory is normally obtained as
PA = DS : EA
But by using a segment override prefix (SOP) in the instruction, any of the four segment registers can be referenced,

$$PA = \left\{ \begin{array}{l} CS \\ DS \\ SS \\ ES \end{array} \right\} : \left\{ \text{Direct Address} \right\}$$

In the direct addressing mode, the 16 bit effective address (EA) is taken directly from the displacement field of the instruction.
Example 1 : MOV CX, START
If the 16 bit value assigned to the offset START by the programmer using an assembler pseudo instruction such as DW is 0040 and [DS] = 3050.
Then BIU generates the 20 bit physical address 30540 H.
The content of 30540 is moved to CL
The content of 30541 is moved to CH

Example 2 : MOV CH, START
If [DS] = 3050 and START = 0040
8 bit content of memory location 30540 is moved to CH.

Example 3 : MOV START, BX
With [DS] = 3050, the value of START is 0040.
Physical address : 30540
MOV instruction moves (BL) and (BH) to locations 30540 and 30541 respectively.

**Register indirect addressing mode :**
The EA is specified in either pointer (BX) register or an index (SI or DI) register. The 20 bit physical address is computed using DS and EA.
Example : MOV [DI], BX
register indirect
If [DS] = 5004, [DI] = 0020, [Bx] = 2456 PA=50060.
The content of BX(2456) is moved to memory locations 50060 H and 50061 H.

$$PA = \left\{ \begin{array}{l} CS \\ DS \\ SS \\ ES \end{array} \right\} = \left\{ \begin{array}{l} BX \\ SI \\ DI \end{array} \right\}$$

**Based Addressing Mode:**

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ or \\ BP \end{Bmatrix} + displacement$$

when memory is accessed PA is computed from BX and DS when the stack is accessed PA is computed from BP and SS.

Example : MOV AL, START [BX]

or

MOV AL, [START + BX]

based mode

EA : [START] + [BX]

PA : [DS] + [EA]

The 8 bit content of this memory location is moved to AL.

Indexed addressing mode:

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} SI \\ or \\ DI \end{Bmatrix} + 8 \text{ or } 16\text{bit displacement}$$

Example : MOV BH, START [SI]

PA : [START] + [SI] + [DS]

The content of this memory is moved into BH.

**Based Indexed addressing mode:**

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ or \\ BP \end{Bmatrix} + \begin{Bmatrix} SI \\ or \\ DI \end{Bmatrix} + 8 \text{ or } 16\text{bit displacement}$$

Example : MOV ALPHA [SI] [BX], CL

If [BX] = 0200, ALPHA – 08, [SI] = 1000 H and [DS] = 3000

Physical address (PA) = 31208

8 bit content of CL is moved to 31208 memory address.

**Instructions:**

**SUB – SUB Destination, Source**

**SBB – SBB Destination, Source**

These instructions subtract the number in some *source* from the number in some *destination* and put the result in the destination. The SBB instruction also subtracts the content of carry flag from the destination. The source may be an immediate number, a register or memory location. The destination can also be a register or a memory location. However, the source and the destination cannot both be memory location. The source and the destination must both be of the same type (bytes or words). If you want to subtract a byte from a word, you must first move

the byte to a word location such as a 16-bit register and fill the upper byte of the word with 0's. Flags affected: AF, CF, OF, PF, SF, ZF.
 SUB CX, BX CX – BX; Result in CX
 SBB CH, AL Subtract content of AL and content of CF from content of CH. From BX

SUB PRICES [BX], 04H Subtract 04 from byte at effective address PRICES [BX],
if PRICES is declared with DB; Subtract 04 from word at effective address PRICES [BX], if it is declared with DW.
 SBB CX, TABLE [BX] Subtract word from effective address TABLE [BX]

and status of CF from CX.
 SBB TABLE [BX], CX Subtract CX and status of CF from word in memory at

effective address TABLE[BX].

Result in CH
 SUB AX, 3427H Subtract immediate number 3427H from AX
 SBB BX, [3427H] Subtract word at displacement 3427H in DS and content of CF

**Decrement Instruction**

**DEC – DEC Destination**

This 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).
 DEC CL Subtract 1 from content of CL register
 DEC BP Subtract 1 from content of BP register
 DEC BYTE PTR [BX] Subtract 1 from byte at offset [BX] in DS.
 DEC WORD PTR [BP] Subtract 1 from a word at offset [BP] in SS.
 DEC COUNT Subtract 1 from byte or word named COUNT in DS.

Decrement a byte if COUNT is declared with a DB;
Decrement a word if COUNT is declared with a DW.

Increment instruction

**INC – INC Destination**

The INC instruction adds 1 to a specified register or to a memory location. AF, OF, PF, SF, and ZF are updated, but CF is not affected. This means that if an 8-bit destination containing FFH or a 16-bit destination containing FFFFH is incremented, the result will be all 0's with no carry.

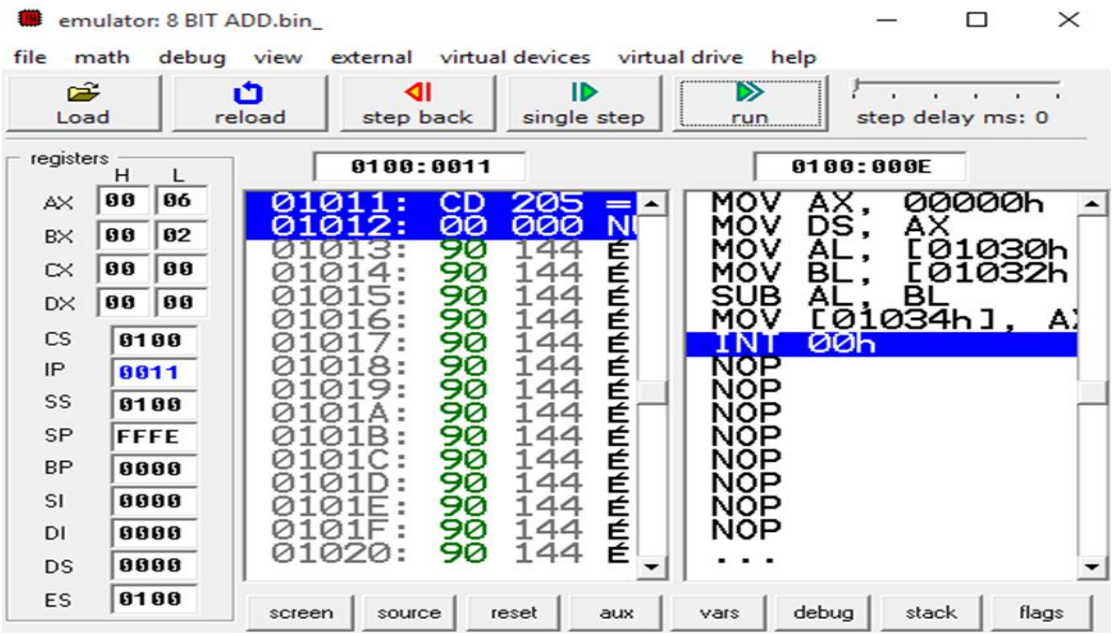
 INC BL Add 1 to contains of BL register
 INC CX Add 1 to contains of CX register
 INC BYTE PTR [BX] Increment byte in data segment at offset contained in BX.
 INC WORD PTR [BX] Increment the word at offset of [BX] and [BX + 1] in the data segment.
 INC TEMP Increment byte or word named TEMP in the data segment.

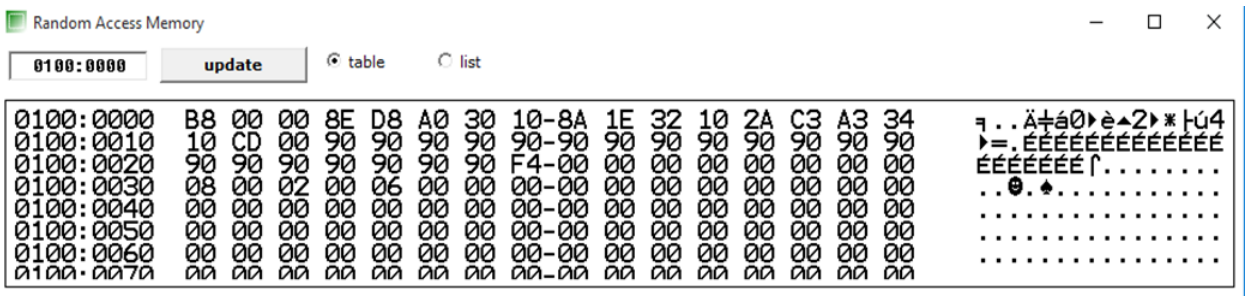
Increment byte if MAX_TEMP declared with DB.
Increment word if MAX_TEMP is declared with DW.
 INC PRICES [BX] Increment element pointed to by [BX] in array PRICES.

Increment a word if PRICES is declared as an array of words;
Increment a byte if PRICES is declared as an array of bytes.

**Program:-** /* 8 BIT SUBSTRACTION */
MOV AX,0000H
MOV DS,AX
MOV AL,[1030H]
MOV BL,[1032H]
SUB AL, BL
MOV [1034H],AX
INT

**Observations:-**



**Rules for Subtraction (8 bit and 16 bit subtraction)**

| A(i/p) | B(i/p) | Y(o/p) | Carry(o/p) |
|---|---|---|---|
| 0 | 0 | 0 | - |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | - |
| 1 | 1 | 0 | - |

**Result: -**

[1030H]=08

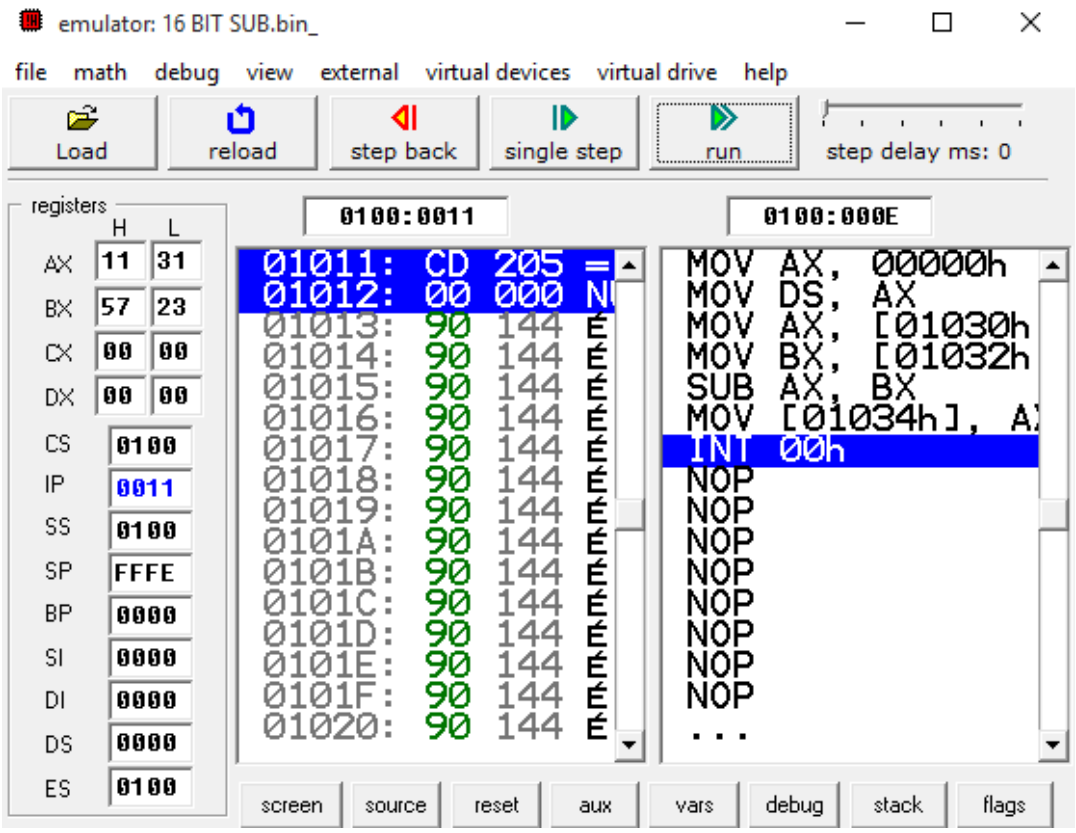
[1032H]=02

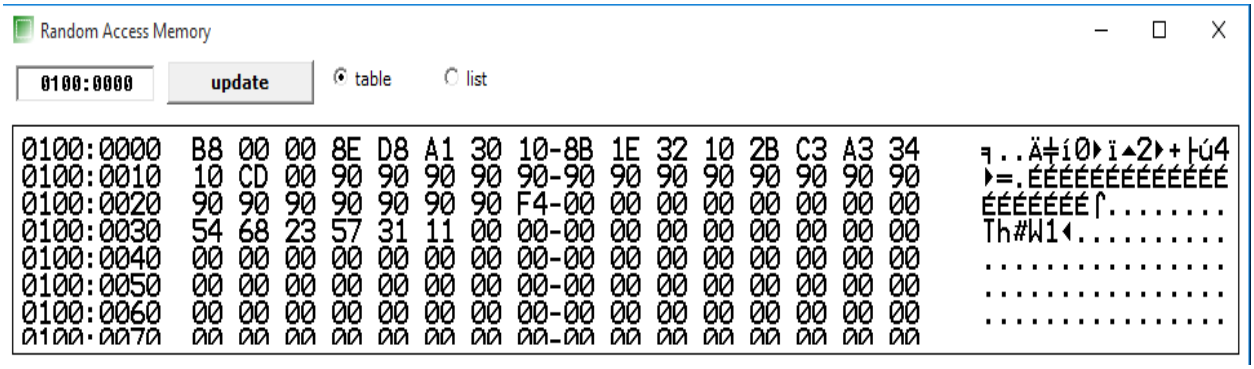
[1034H]=06.

**Program:-** /* 16 BIT SUBSTRACTION */

```
MOV AX,0000H
MOV DS,AX
MOV AX,[1030H]
MOV BX,[1032H]
SUB AX,BX
MOV [1034H],AX
INT
```

**Observations:-**






**Result: -**

[1030H]=5468

[1032H]=2357

[1034H]=3111

**Conclusion:** Thus the addressing modes are studied and the 8-bit and 16-bit subtraction is implemented,

# Experiment No.3
## Multiplication

**Aim:-** Write assembly language program to perform 8 bit and 16 bit multiplication

**Objective:** To study string related operations with the help of string instructions.
To use Multiplication instruction for 8 bit and 16 bit numbers.

**Software**: 8086 Emulator

**Theory:**

**MUL – MUL Source**

This instruction multiplies 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. The source can be a register or a memory location. When a byte is multiplied by the content of AL, the result (product) is put in AX. When a word is multiplied by the content of AX, the result is put in DX and AX registers. If the most significant byte of a 16-bit result or the most significant word of a 32-bit result is 0, CF and OF will both be 0's. AF, PF, SF and ZF are undefined after a MUL instruction.

If you want to multiply a byte with a word, you must first move the byte to a word location such as an extended register and fill the upper byte of the word with all 0's. You cannot use the CBW instruction for this, because the CBW instruction fills the upper byte with copies of the most significant bit of the lower byte.

MUL BH Multiply AL with BH; result in AX

MUL CX Multiply AX with CX; result high word in DX, low word in AX

MUL BYTE PTR [BX] Multiply AL with byte in DS pointed to by [BX]

MUL FACTOR [BX] Multiply AL with byte at effective address FACTOR [BX], if it is declared as type byte with DB. Multiply AX with word at effective address FACTOR [BX], if it is declared as type word with DW. MOV AX, MCAND_16 Load 16-bit multiplicand into AX

MOV CL, MPLIER_8 Load 8-bit multiplier into CL

MOV CH, 00H Set upper byte of CX to all 0's

MUL CX AX times CX; 32-bit result in DX and AX

**IMUL – IMUL Source**

This instruction multiplies a *signed* byte from *source* with a *signed* byte in AL or a *signed* word from some *source* with a *signed* word in AX. The source can be a register or a memory location. When a byte from source is multiplied with content of AL, the signed result (product) will be put in AX. When a word from source is multiplied by AX, the result is put in DX and AX. If the magnitude of the product does not require all the bits of the destination, the unused byte / word will be filled with copies of the sign bit. If the upper byte of a 16-bit result or the upper word of a 32-bit result contains only copies of the sign bit (all 0's or all 1's), then CF and the OF will both be 0; If it contains a part of the product, CF and OF will both be 1. AF, PF, SF and ZF are undefined after IMUL.

If you want to multiply a signed byte with a signed word, you must first move the byte into a word location and fill the upper byte of the word with copies of the sign bit. If you move the byte into AL, you can use the CBW instruction to do this.

IMUL BH Multiply signed byte in AL with signed byte in BH; result in AX.
IMUL AX Multiply AX times AX; result in DX and AX
 MOV CX, MULTIPLIER Load signed word in CX

STRING MANIPULATION INSTRUCTIONS

**MOVS – MOVS Destination String Name, Source String Name**
**MOVSB – MOVSB Destination String Name, Source String Name**
**MOVSW – MOVSW Destination String Name, Source String Name**

This instruction copies a byte or a word from location in the data segment to a location in the extra segment. The offset of the source in the data segment must be in the SI register. The offset of the destination in the extra segment must be in the DI register. For multiple-byte or multiple-word moves, the number of elements to be moved is put in the CX register so that it can function as a counter. After the byte or a word is moved, SI and DI are automatically adjusted to point to the next source element and the next destination element. If DF is 0, then SI and DI will incremented by 1 after a byte move and by 2 after a word move. If DF is 1, then SI and DI will be decremented by 1 after a byte move and by 2 after a word move. MOVS does not affect any flag. When using the MOVS instruction, you must in some way tell the assembler whether you want to move a string as bytes or as word. There are two ways to do this. The first way is to indicate the name of the source and destination strings in the instruction, as, for example. MOVS DEST, SRC. The assembler will code the instruction for a byte / word move if they were declared with a DB / DW. The second way is to add a “B” or a “W” to the MOVS mnemonic. MOVSB says move a string as bytes; MOVSW says move a string as words.

MOV SI, OFFSET SOURCE Load offset of start of source string in DS into SI
MOV DI, OFFSET DESTINATION Load offset of start of destination string in ES into DI
CLD Clear DF to auto increment SI and DI after move
MOV CX, 04H Load length of string into CX as counter
REP MOVSB Move string byte until CX = 0

**LODS / LODSB / LODSW (LOAD STRING BYTE INTO AL OR STRING WORD INTO AX)**

This instruction copies a byte from a string location pointed to by SI to AL, or a word from a string location pointed to by SI to AX. If DF is 0, SI will be automatically incremented (by 1 for a byte string, and 2 for a word string) to point to the next element of the string. If DF is 1, SI will be automatically decremented (by 1 for a byte string, and 2 for a word string) to point to the previous element of the string. LODS does not affect any flag.

CLD Clear direction flag so that SI is auto-incremented
MOV SI, OFFSET SOURCE Point SI to start of string
LODS SOURCE Copy a byte or a word from string to AL or AX

Note: The assembler uses the name of the string to determine whether the string is of type bye or type word. Instead of using the string name to do this, you can use the mnemonic LODSB to tell the assembler that the string is type byte or the mnemonic LODSW to tell the assembler that the string is of type word.

**STOS / STOSB / STOSW (STORE STRING BYTE OR STRING WORD)**

This instruction copies a byte from AL or a word from AX to a memory location in the extra segment pointed to by DI. In effect, it replaces a string element with a byte from AL or a word from AX. After the copy, DI is automatically incremented or decremented to point to next or previous element of the string. If DF is cleared, then DI will automatically incremented by 1 for a byte string and by 2 for a word string. If DI is set, DI will be automatically decremented by 1 for a byte string and by 2 for a word string. STOS does not affect any flag.

MOV DI, OFFSET TARGET
STOS TARGET
Note: The assembler uses the string name to determine whether the string is of type byte or type word. If it is a byte string, then string byte is replaced with content of AL. If it is a word string, then string word is replaced with content of AX.
MOV DI, OFFSET TARGET
STOSB
"B" added to STOSB mnemonic tells assembler to replace byte in string with byte from AL. STOSW would tell assembler directly to replace a word in the string with a word from AX.

**CMPS / CMPSB / CMPSW (COMPARE STRING BYTES OR STRING WORDS)**
This instruction can be used to compare a byte / word in one string with a byte / word in another string. SI is used to hold the offset of the byte or word in the source string, and DI is used to hold the offset of the byte or word in the destination string.
The AF, CF, OF, PF, SF, and ZF flags are affected by the comparison, but the two operands are not affected. After the comparison, SI and DI will automatically be incremented or decremented to point to the next or previous element in the two strings. If DF is set, then SI and DI will automatically be decremented by 1 for a byte string and by 2 for a word string. If DF is reset, then SI and DI will automatically be incremented by 1 for byte strings and by 2 for word strings. The string pointed to by SI must be in the data segment. The string pointed to by DI must be in the extra segment.
The CMPS instruction can be used with a REPE or REPNE prefix to compare all the elements of a string.
MOV SI, OFFSET FIRST Point SI to source string
MOV DI, OFFSET SECOND Point DI to destination string
CLD DF cleared, SI and DI will auto-increment after compare
MOV CX, 100 Put number of string elements in CX
REPE CMPSB Repeat the comparison of string bytes until end of string
or until compared bytes are not equal
CX functions as a counter, which the REPE prefix will cause CX to be decremented after each compare. The B attached to CMPS tells the assembler that the strings are of type byte. If you want to tell the assembler that strings are of type word, write the instruction as CMPSW. The REPE CMPSW instruction will cause the pointers in SI and DI to be incremented by 2 after each compare, if the direction flag is set.
**SCAS / SCASB / SCASW (SCAN A STRING BYTE OR A STRING WORD)**
SCAS compares a byte in AL or a word in AX with a byte or a word in ES pointed to by DI. Therefore, the string to be scanned must be in the extra segment, and DI must contain the offset of the byte or the word to be compared. If DF is cleared, then DI will be incremented by 1 for byte strings and by 2 for word strings. If DF is set, then DI will be decremented by 1 for byte strings and by 2 for word strings. SCAS affects AF, CF, OF, PF, SF, and ZF, but it does not change either the operand in AL (AX) or the operand in the string.
The following program segment scans a text string of 80 characters for a carriage return, 0DH, and puts the offset of string into DI:
MOV DI, OFFSET STRING

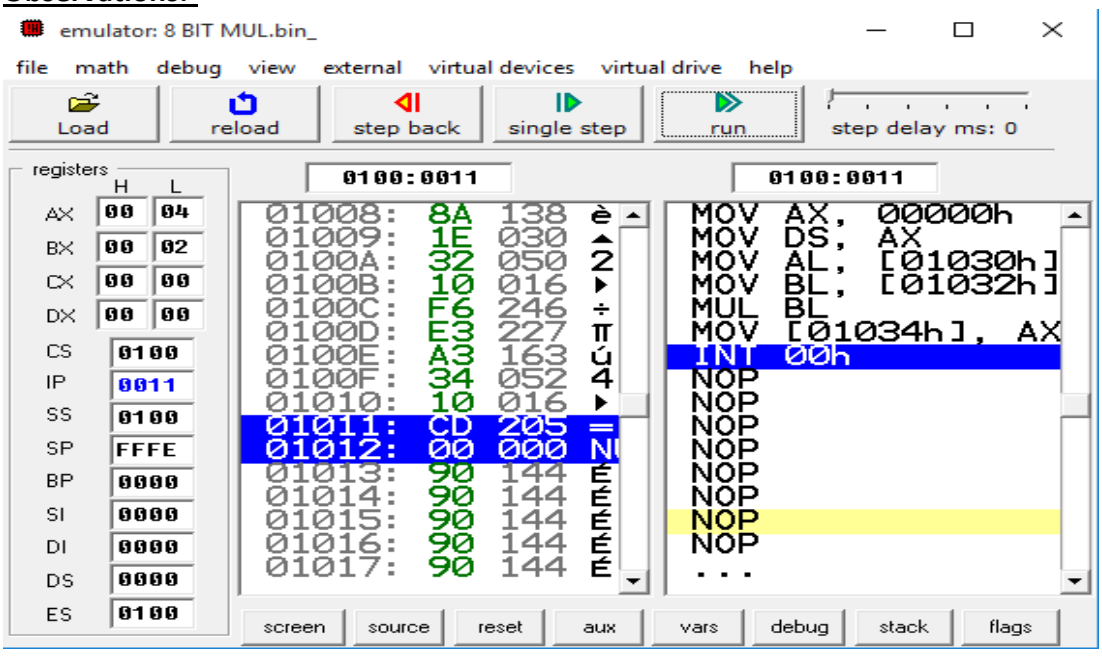
MOV AL, 0DH Byte to be scanned for into AL
MOV CX, 80 CX used as element counter

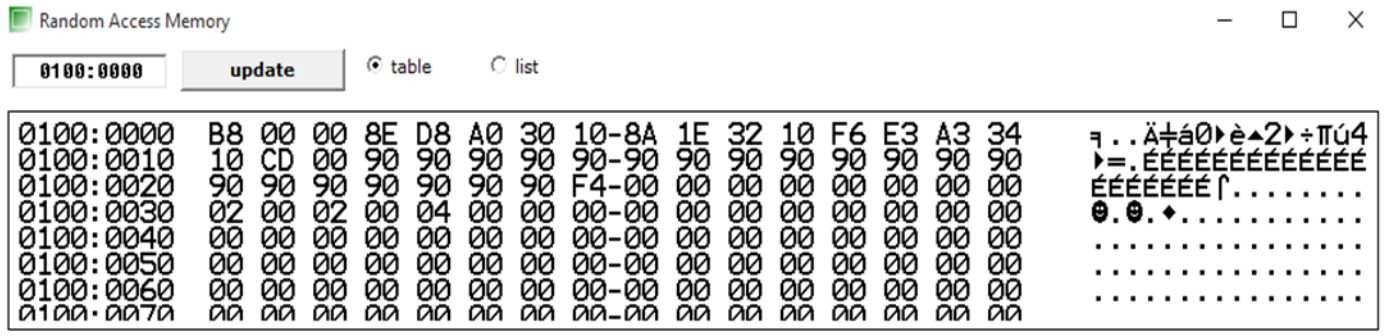
CLD Clear DF, so that DI auto increments
REPNE SCAS STRING Compare byte in string with byte in AL
**REP / REPE / REPZ / REPNE / REPNZ (PREFIX)**
**(REPEAT STRING INSTRUCTION UNTIL SPECIFIED CONDITIONS EXIST)**
REP is a prefix, which is written before one of the string instructions. It will cause the CX register to be decremented and the string instruction to be repeated until CX = 0. The instruction REP MOVSB, for example, will continue to copy string bytes until the number of bytes loaded into CX has been copied. REPE and REPZ are two mnemonics for the same prefix. They stand for *repeat if equal* and *repeat if zero*, respectively. They are often used with the Compare String instruction or with the Scan String instruction. They will cause the string instruction to be repeated as long as the compared bytes or words are equal (ZF = 1) and CX is not yet counted down to zero. In other words, there are two conditions that will stop the repetition: CX = 0 or string bytes or words not equal. REPE CMPSB Compare string bytes until end of string or until string bytes not equal.
REPNE and REPNZ are also two mnemonics for the same prefix. They stand for *repeat if not equal* and *repeat if not zero*, respectively. They are often used with the Compare String instruction or with the Scan String instruction. They will cause the string instruction to be repeated as long as the compared bytes or words are not equal (ZF = 0) and CX is not yet counted down to zero.
REPNE SCASW Scan a string of word until a word in the string matches the word
in AX or until all of the string has been scanned.
The string instruction used with the prefix determines which flags are affected.

**Program:-** /* 8 BIT MULTIPLICATION */

```
MOV AX,0000H
MOV DS,AX
MOV AL,[1030H]
MOV BL,[1032H]
MUL BL
MOV [1034H],AX
INT
```

**Observations:-**






**Rules for Multiplication (8 bit and 16 bit Multiplication)**

| A(i/p) | B(i/p) | Y(o/p) | Carry(o/p) |
|---|---|---|---|
| 0 | 0 | 0 | - |
| 0 | 1 | 0 | - |
| 1 | 0 | 0 | - |
| 1 | 1 | 1 | - |

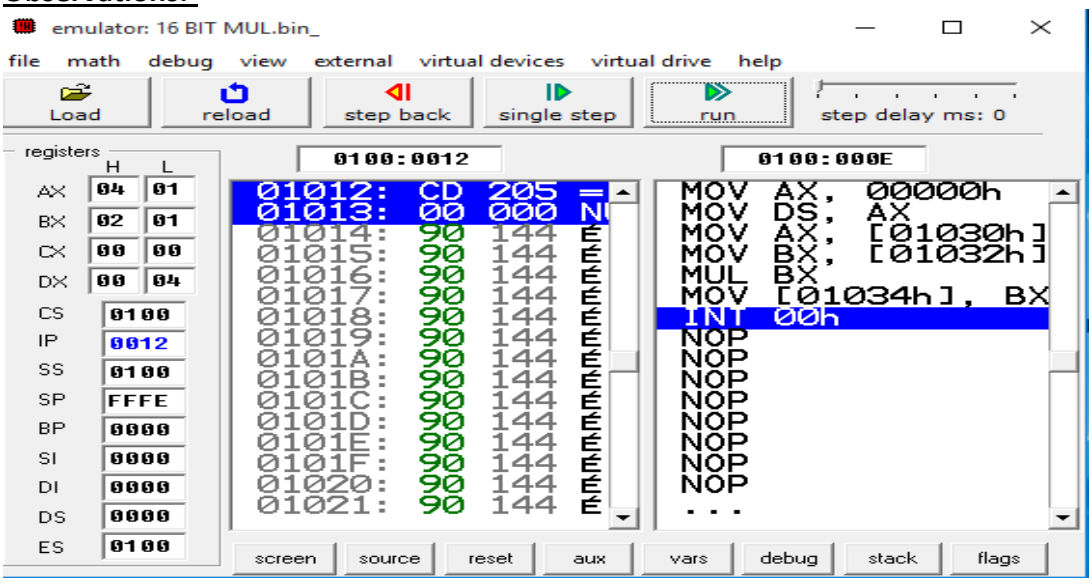
**Result:**

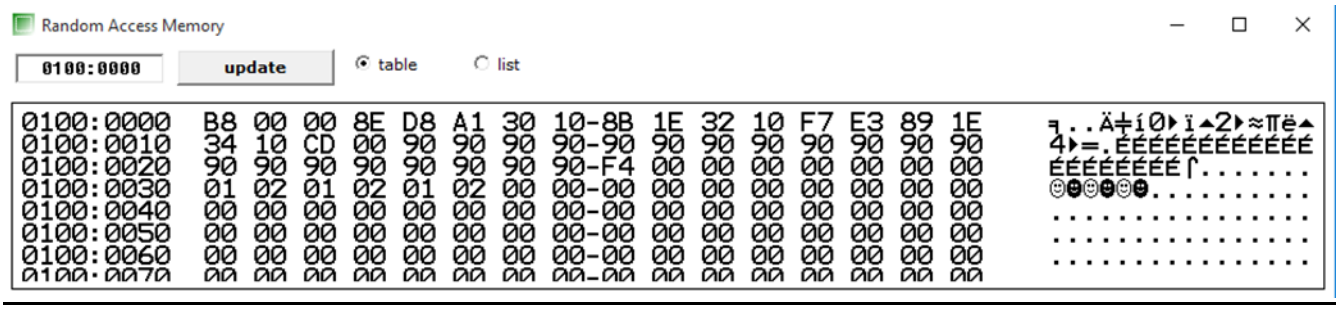
[1030H]=02
[1032H]=02
[1034H]=04

**Program:-** /* 16 BIT MULTIPLICATION */

```
MOV AX,0000H
MOV DS,AX
MOV AX,[1030H]
MOV BX,[1032H]
MUL BX
MOV [1034H],BX
INT
```

**Observations:-**






**Result:-**

[1030H]=0102

[1032H]=0102

[1034H]=0104.

**Conclusion:** Thus the string instructions are studied and multiplication for 8 bit and 16 bit.

# Experiment No.4
## Division

**Aim:-** Write assembly language program to perform 8 bit and 16 bit division.

**Objective:** Describe the conditional and unconditional jump instructions.
To study and implement the division instructions.

**Software**: 8086 Emulator

**Theory:**

**DIV – DIV Source**

This 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. The divisor can be in a register or a memory location. After the division, AL will contain the 8-bit quotient, and AH will contain the 8-bit remainder. When a double word is divided by a word, the most significant word of the double word must be in DX, and the least significant word of the double word must be in AX. After the division, AX will contain the 16-bit quotient and DX will contain the 16-bit remainder. If an attempt is made to divide by 0 or if the quotient is too large to fit in the destination (greater than FFH / FFFFH), the 8086 will generate a type 0 interrupt. All flags are undefined after a DIV instruction.
If you want to divide a byte by a byte, you must first put the dividend byte in AL and fill AH with all 0's. Likewise, if you want to divide a word by another word, then put the dividend word in AX and fill DX with all 0's.
DIV BL Divide word in AX by byte in BL; Quotient in AL, remainder in AH
DIV CX Divide down word in DX and AX by word in CX;
Quotient in AX, and remainder in DX.
DIV SCALE [BX] AX / (byte at effective address SCALE [BX]) if SCALE [BX] is of type
byte; or (DX and AX) / (word at effective address SCALE[BX]
if SCALE[BX] is of type word

**IDIV – IDIV Source**

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. The divisor can be in an 8-bit register or a memory location. After the division, AL will contain the signed quotient, and AH will contain the signed remainder. The sign of the remainder will be the same as the sign of the dividend. If an attempt is made to divide by 0, the quotient is greater than 127 (7FH) or less than –127 (81H), the 8086 will automatically generate a type 0 interrupt.
When dividing a signed double word by a signed word, the most significant word of the dividend (numerator) must be in the DX register, and the least significant word of the dividend must be in the AX register. The divisor can be in any other 16-bit register or memory location. After the division, AX will contain a signed 16-bit quotient, and DX will contain a signed 16-bit remainder. The sign of the remainder will be the same as the sign of the dividend. Again, if an attempt is

made to divide by 0, the quotient is greater than +32,767 (7FFFH) or less than –32,767 (8001H), the 8086 will automatically generate a type 0 interrupt.
All flags are undefined after an IDIV.
If you want to divide a signed byte by a signed byte, you must first put the dividend byte in AL and sign-extend AL into AH. The CBW instruction can be used for this purpose. Likewise, if you want to divide a signed word by a signed word, you must put the dividend word in AX and extend the sign of AX to all the bits of DX. The CWD instruction can be used for this purpose.
IDIV BL Signed word in AX/signed byte in BL
IDIV BP Signed double word in DX and AX/signed word in BP
IDIV BYTE PTR [BX] AX / byte at offset [BX] in DS
CWB
CWD
TRANSFER-OF-CONTROL INSTRUCTIONS

**JMP (UNCONDITIONAL JUMP TO SPECIFIED DESTINATION)**
This instruction will fetch the next instruction from the location specified in the instruction rather than from the next location after the JMP instruction. If the destination is in the same code segment as the JMP instruction, then only the instruction pointer will be changed to get the destination location. This is referred to as a *near jump.* If the destination for the jump instruction is in a segment with a name different from that of the segment containing the JMP instruction, then both the instruction pointer and the code segment register content will be changed to get the destination location. This referred to as a *far jump*. The JMP instruction does not affect any flag.
JMP CONTINUE

This instruction fetches the next instruction from address at label CONTINUE. If the label is in the same segment, an offset coded as part of the instruction will be added to the instruction pointer to produce the new fetch address. If the label is another segment, then IP and CS will be replaced with value coded in part of the instruction. This type of jump is referred to as *direct* because the displacement of the destination or the destination itself is specified directly in the instruction.

**JA / JNBE (JUMP IF ABOVE / JUMP IF NOT BELOW OR EQUAL)**
If, after a compare or some other instructions which affect flags, the zero flag and the carry flag both are 0, this instruction will cause execution to jump to a label given in the instruction. If CF and ZF are not both 0, the instruction will have no effect on program execution.
CMP AX, 4371H Compare by subtracting 4371H from AX
JA NEXT Jump to label NEXT if AX above 4371H
CMP AX, 4371H Compare (AX – 4371H)
JNBE NEXT Jump to label NEXT if AX not below or equal to 4371H
**JAE / JNB / JNC**
**(JUMP IF ABOVE OR EQUAL / JUMP IF NOT BELOW / JUMP IF NO CARRY)**
If, after a compare or some other instructions which affect flags, the carry flag is 0, this instruction will cause execution to jump to a label given in the instruction. If CF is 1, the instruction will have no effect on program execution.
CMP AX, 4371H Compare (AX – 4371H)
JAE NEXT Jump to label NEXT if AX above 4371H
CMP AX, 4371H Compare (AX – 4371H)

JNB NEXT Jump to label NEXT if AX not below 4371H
ADD AL, BL Add two bytes
JNC NEXT If the result with in acceptable range, continue

**JB / JC / JNAE (JUMP IF BELOW / JUMP IF CARRY / JUMP IF NOT ABOVE OR EQUAL)**

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.

CMP AX, 4371H Compare (AX – 4371H)
JB NEXT Jump to label NEXT if AX below 4371H
ADD BX, CX Add two words
JC NEXT Jump to label NEXT if CF = 1
CMP AX, 4371H Compare (AX – 4371H)
JNAE NEXT Jump to label NEXT if AX not above or equal to 4371H

**JBE / JNA (JUMP IF BELOW OR EQUAL / JUMP IF NOT ABOVE)**

If, after a compare or some other instructions which affect flags, either the zero flag or the carry flag is 1, this instruction will cause execution to jump to a label given in the instruction. If CF and ZF are both 0, the instruction will have no effect on program execution.

CMP AX, 4371H Compare (AX – 4371H)
JBE NEXT Jump to label NEXT if AX is below or equal to 4371H
CMP AX, 4371H Compare (AX – 4371H)
JNA NEXT Jump to label NEXT if AX not above 4371H

**JG / JNLE (JUMP IF GREATER / JUMP IF NOT LESS THAN OR EQUAL)**

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction, if the zero flag is 0 and the carry flag is the same as the overflow flag.

CMP BL, 39H Compare by subtracting 39H from BL
JG NEXT Jump to label NEXT if BL more positive than 39H
CMP BL, 39H Compare by subtracting 39H from BL
JNLE NEXT Jump to label NEXT if BL is not less than or equal to 39H

**JGE / JNL (JUMP IF GREATER THAN OR EQUAL / JUMP IF NOT LESS THAN)**

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction, if the sign flag is equal to the overflow flag.

CMP BL, 39H Compare by subtracting 39H from BL
JGE NEXT Jump to label NEXT if BL more positive than or equal to 39H
CMP BL, 39H Compare by subtracting 39H from BL
JNL NEXT Jump to label NEXT if BL not less than 39H

**JL / JNGE (JUMP IF LESS THAN / JUMP IF NOT GREATER THAN OR EQUAL)**

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction if the sign flag is not equal to the overflow flag.

CMP BL, 39H Compare by subtracting 39H from BL
JL AGAIN Jump to label AGAIN if BL more negative than 39H
CMP BL, 39H Compare by subtracting 39H from BL
JNGE AGAIN Jump to label AGAIN if BL not more positive than or equal to
39H

**JLE / JNG (JUMP IF LESS THAN OR EQUAL / JUMP IF NOT GREATER)**

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction if the zero flag is set, or if the sign flag not equal to the overflow flag.
CMP BL, 39H Compare by subtracting 39H from BL
JLE NEXT Jump to label NEXT if BL more negative than or equal to 39H
CMP BL, 39H Compare by subtracting 39H from BL

JNG NEXT Jump to label NEXT if BL not more positive than 39H

**JE / JZ (JUMP IF EQUAL / JUMP IF ZERO)**

This instruction is usually used after a Compare instruction. If the zero flag is set, then this instruction will cause a jump to the label given in the instruction.
CMP BX, DX Compare (BX-DX)

JE DONE Jump to DONE if BX = DX

**JNE / JNZ (JUMP NOT EQUAL / JUMP IF NOT ZERO)**

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.
ADD AX, 0002H Add count factor 0002H to AX
DEC BX Decrement BX
JNZ NEXT Jump to label NEXT if BX □ 0

**JS (JUMP IF SIGNED / JUMP IF NEGATIVE)**

This instruction will cause a jump to the specified destination address if the sign flag is set. Since a 1 in the sign flag indicates a negative signed number, you can think of this instruction as saying "jump if negative".
□ ADD BL, DH Add signed byte in DH to signed byte in DL
JS NEXT Jump to label NEXT if result of addition is negative number

**JNS (JUMP IF NOT SIGNED / JUMP IF POSITIVE)**

This instruction will cause a jump to the specified destination address if the sign flag is 0. Since a 0 in the sign flag indicate a positive signed number, you can think to this instruction as saying "jump if positive".
□ DEC AL Decrement AL
JNS NEXT Jump to label NEXT if AL has not decremented to FFH

**JP / JPE (JUMP IF PARITY / JUMP IF PARITY EVEN)**

If the number of 1's left in the lower 8 bits of a data word after an instruction which affects the parity flag is even, then the parity flag will be set. If the parity flag is set, the JP / JPE instruction will cause a jump to the specified destination address.

**JNP / JPO (JUMP IF NO PARITY / JUMP IF PARITY ODD)**

If the number of 1's left in the lower 8 bits of a data word after an instruction which affects the parity flag is odd, then the parity flag is 0. The JNP / JPO instruction will cause a jump to the specified destination address, if the parity flag is 0.

**JO (JUMP IF OVERFLOW)**

The overflow flag will be set if the magnitude of the result produced by some signed arithmetic operation is too large to fit in the destination register or memory location. The JO instruction will cause a jump to the destination given in the instruction, if the overflow flag is set.

ADD AL, BL Add signed bytes in AL and BL

JO ERROR Jump to label ERROR if overflow from add

**JNO (JUMP IF NO OVERFLOW)**

The overflow flag will be set if some signed arithmetic operation is too large to fit in the destination register or memory location. The JNO instruction will cause a jump to the destination given in the instruction, if the overflow flag is not set.

ADD AL, BL Add signed byte in AL and BL

JNO DONE Process DONE if no overflow

**JCXZ (JUMP IF THE CX REGISTER IS ZERO)**

This instruction will cause a jump to the label to a given in the instruction, if the CX register contains all 0's. The instruction does not look at the zero flag when it decides whether to jump or not.

JCXZ SKIP If CX = 0, skip the process

SUB [BX], 07H Subtract 7 from data value

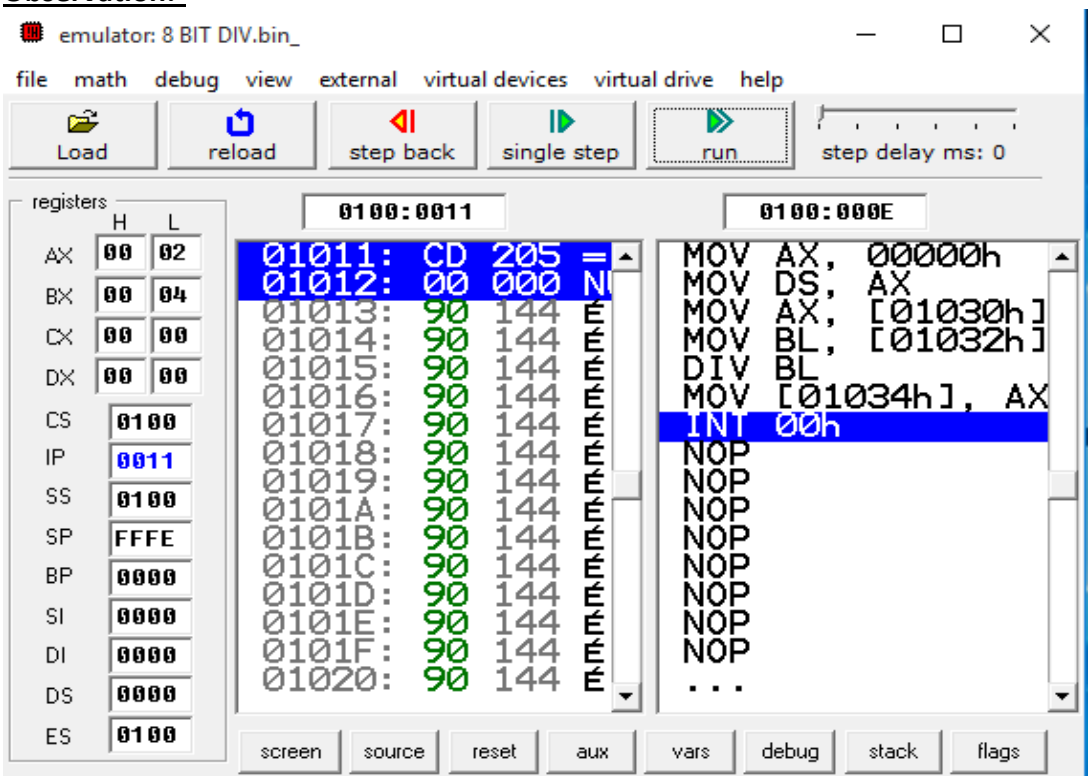
SKIP: ADD C Next instruction

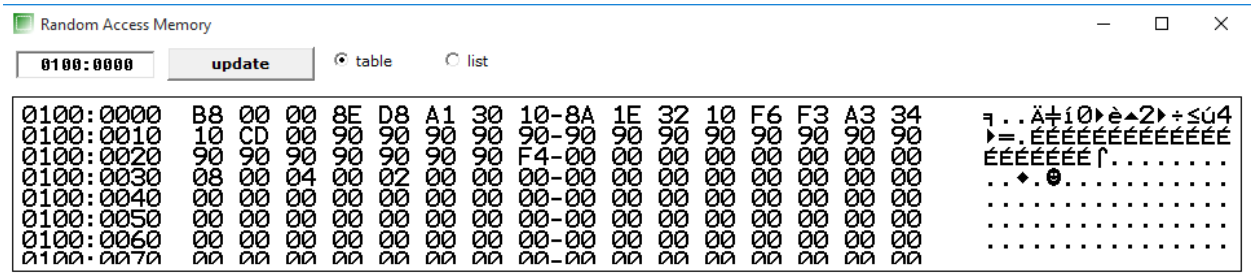
LOOP NEXT Repeat until all elements adjusted

**Program:-** /* 8 BIT DIVISION */

```
MOV AX,0000H
MOV DS,AX
MOV AX,[1030H]
MOV BL,[1032H]
DIV BL
MOV [1034H],AX
INT
```

**Observation:-**





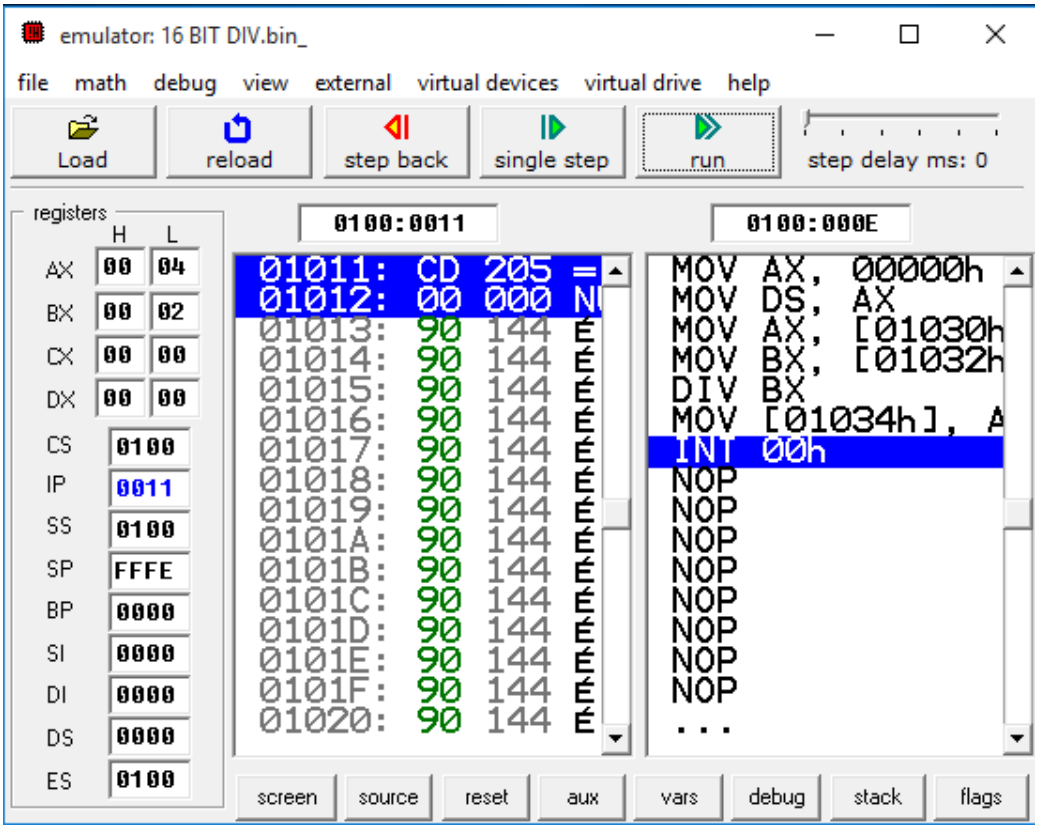

**Result:-**

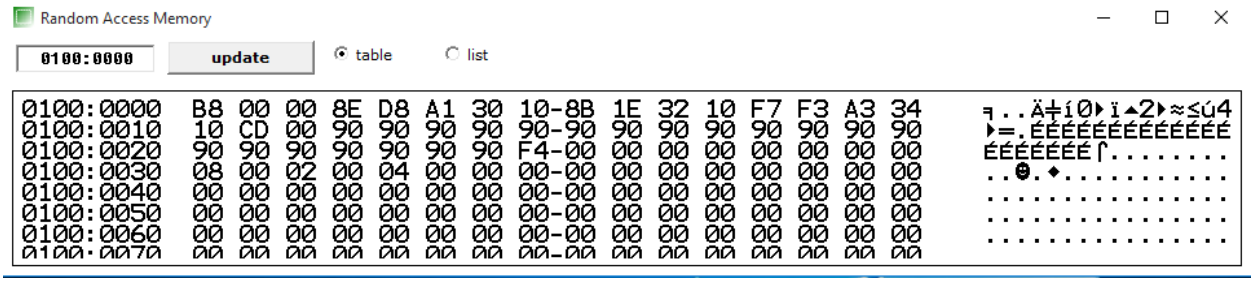
[1030H]=08
[1032H]=04
[1034H]=02

**Program:** /* 16 BIT DIVISION */

```
MOV AX,0000H
MOV DS,AX
MOV AX,[1030H]
MOV BX,[1032H]
DIV BX
MOV [1034H],AX
INT
```

**Observation:-**






**Result:-**

[1030H]=08

[1032H]=02

[1034H]=04.

**Conclusion:** In this way we perform 8 bit and 16 bit division.

# Experiment No.5
## AND Operation

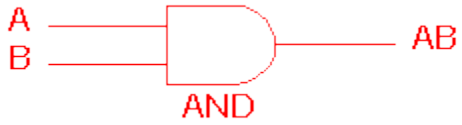
**Aim:-** Write assembly language program to perform 8 bit and 16 bit AND operation.

**Objective:** Describe the AND instructions Operation.

**Software**: 8086 Emulator

**Theory:**

**AND gate**



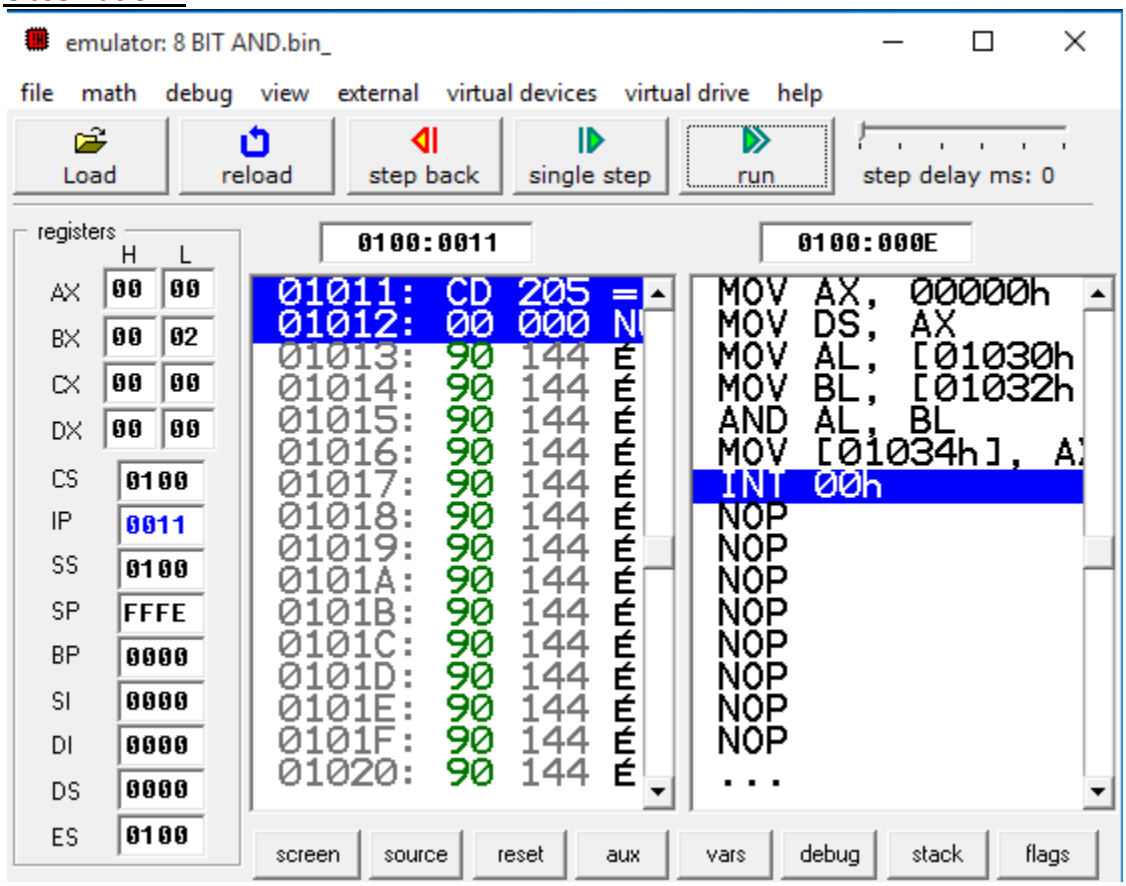

| 2 Input AND gate | | |
|---|---|---|
| A | B | A.B |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

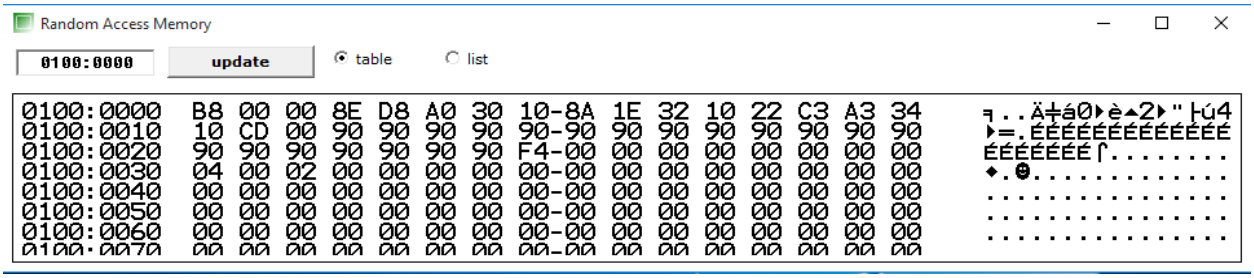
The AND gate is an electronic circuit that gives a **high** output (1) only if **all** its inputs are high. A dot (.) is used to show the AND operation i.e. A.B. Bear in mind that this dot is sometimes omitted i.e. AB

**Program:-** **/***8 BIT ANDING */

```
MOV AX,0000H
MOV DS,AX
MOV AL,[1030H]
MOV BL,[1032H]
AND AL, BL
MOV [1034H],AX
INT
```

**Observation:-**




**Result:-**

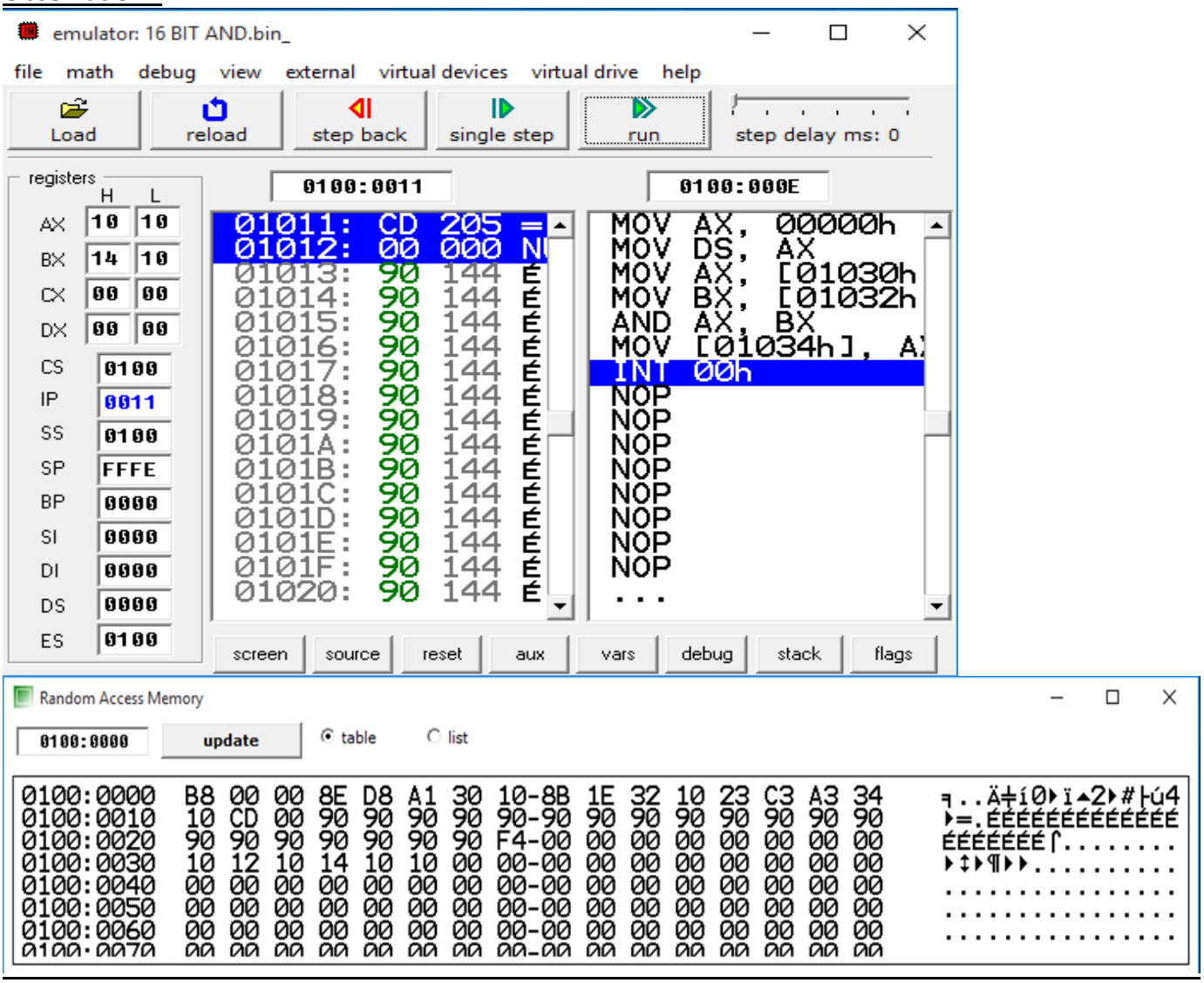
[1030H]=04
[1032H]=02
[1034H]=00

**Program:-**

/* 16 BIT ANDING */

```
MOV AX,0000H
MOV DS,AX
MOV AX,[1030H]
MOV BX,[1032H]
AND AX,BX
MOV [1034H],AX
INT
```

**Observation:-**




**Result:-**

[1030H]=1012

[1032H]=1014

[1034H]=1010.

**Conclusion:-**

In this way we executed assembly language program for ANDing of two 8 bit and 16 bit numbers.

# Experiment No.6
## OR Operation

**Aim:-** Write assembly language program to perform 8 bit and 16 bit OR operation.

**Objective:** Describe the OR instructions Operation.

**Software**: 8086 Emulator

**Theory:**

### OR gate

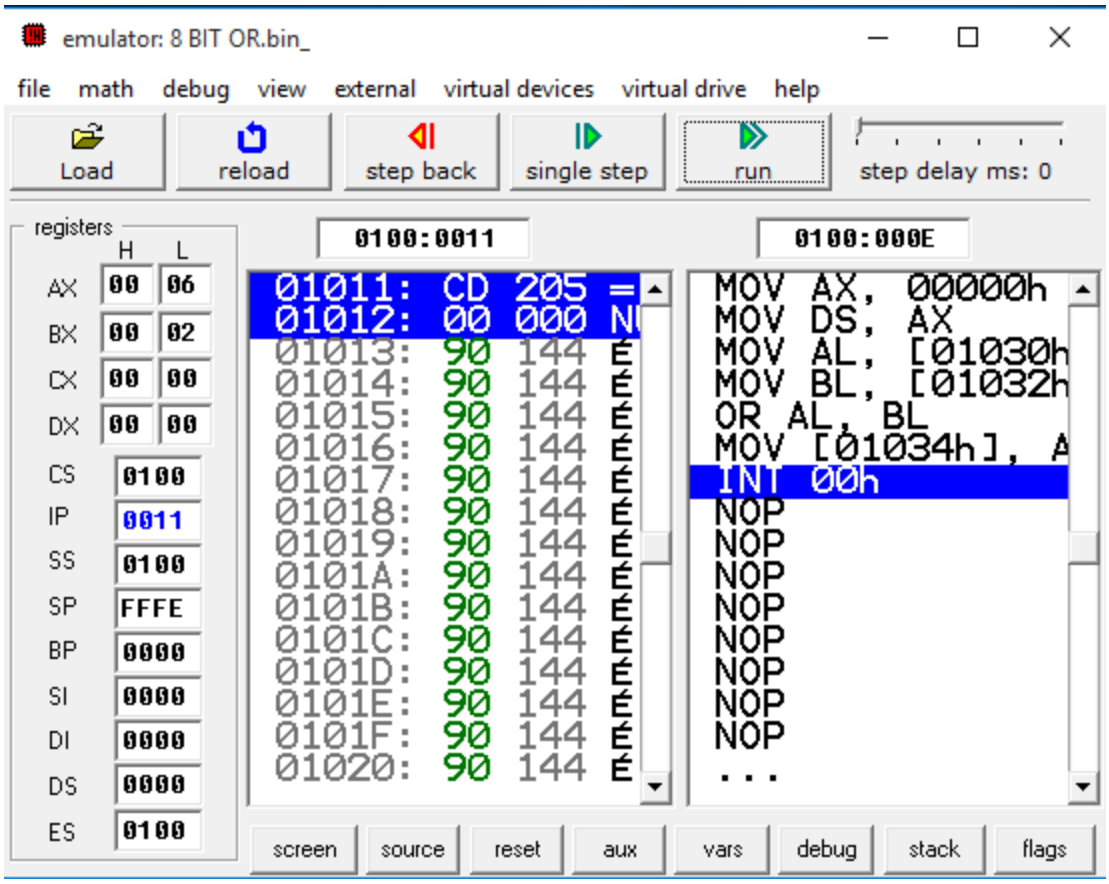

| 2 Input OR gate | | |
|---|---|---|
| A | B | A+B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

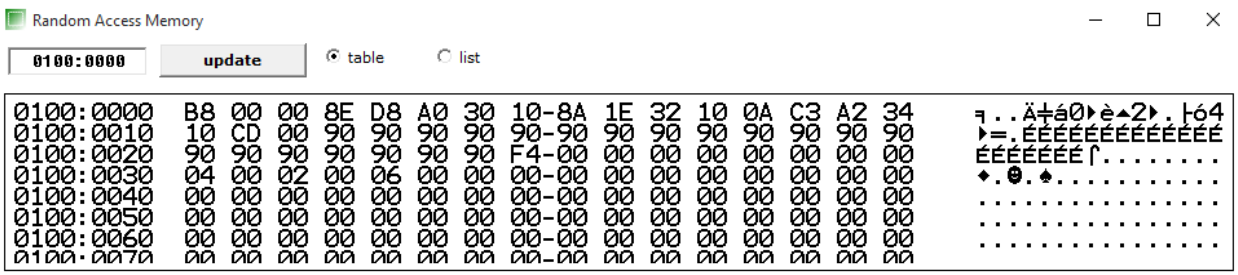
The OR gate is an electronic circuit that gives a high output (1) if **one or more** of its inputs are high. A plus (+) is used to show the OR operation.

**Program:-** /* 8 BIT ORING */

```
MOV AX,0000H
MOV DS,AX
MOV AL,[1030H]
MOV BL,[1032H]
OR AL,BL
MOV [1034H],AL
INT
```

**Observation:-**




**Result:-**

[1030H]=04

[1032H]=02

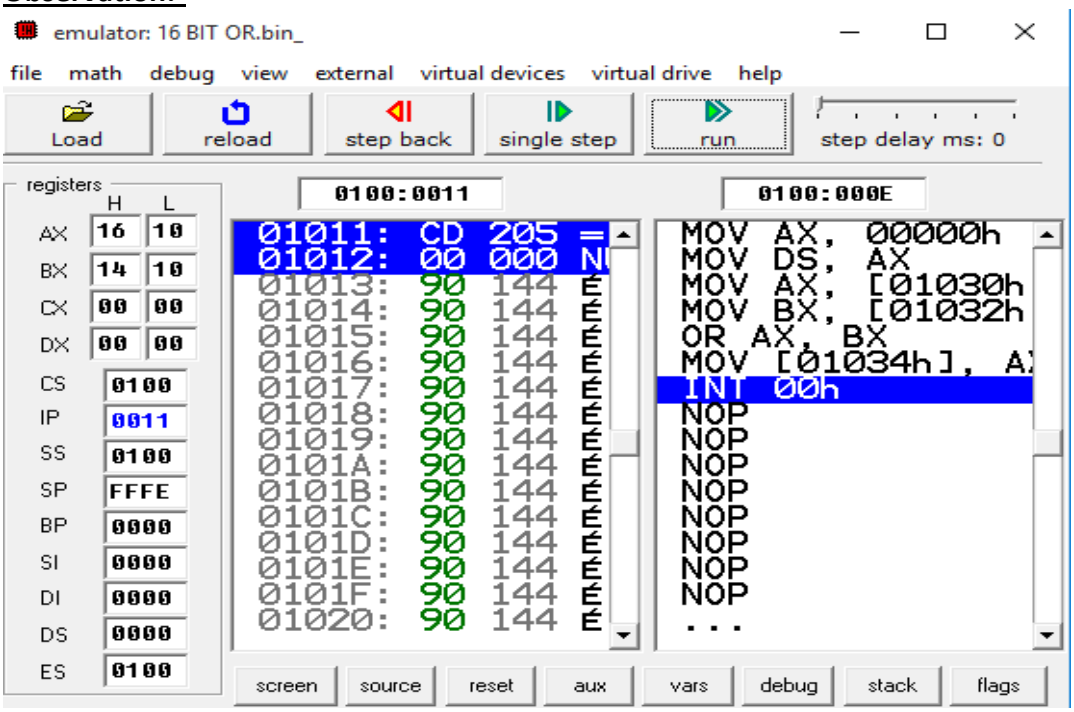
[1034H]=06.

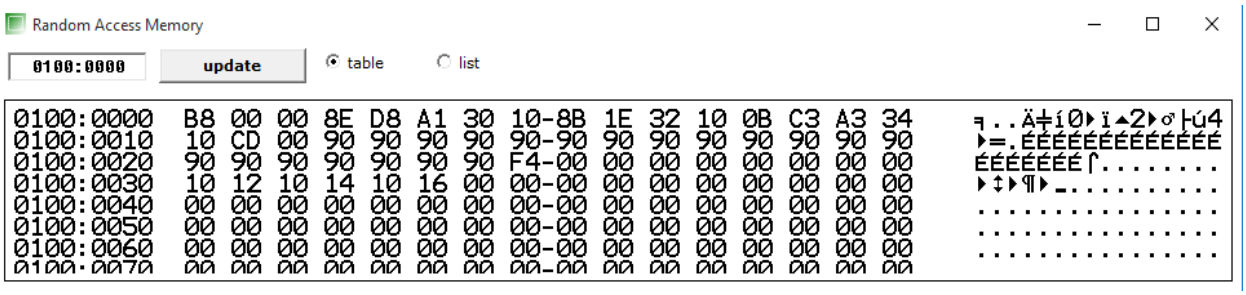
**Program:-**

/* 16 BIT ORING */

```
MOV AX,0000H
MOV DS,AX
MOV AX,[1030H]
MOV BX,[1032H]
OR AX,BX
MOV [1034H],AX
INT
```

**Observation:-**




**Result:-**

[1030H]=1012

[1032H]=1014

[1034H]=1016.

**Conclusion:-**

In this way we executed assembly language program for ORing of two 8 bit and 16 bit numbers.

# Experiment No.7

## XOR Operation

**Aim:-** Write assembly language program to perform 8 bit and 16 bit XOR operation.

**Objective:** Describe the XOR instructions Operation.

**Software**: 8086 Emulator

**Theory:**

### EXOR gate

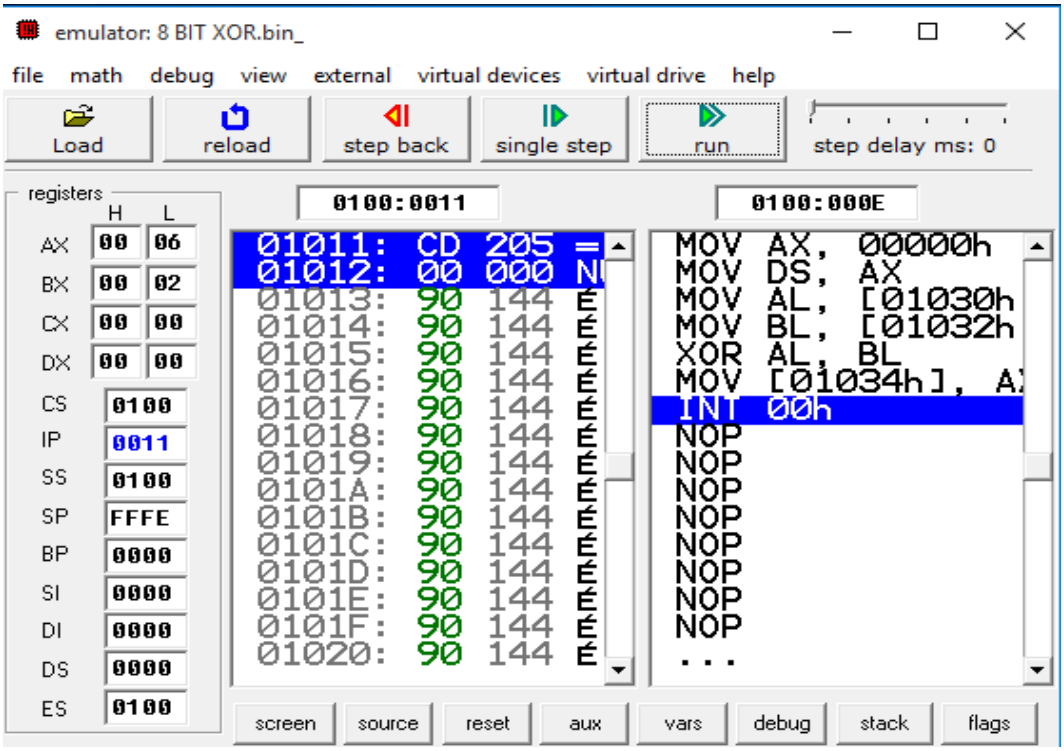

| 2 Input EXOR gate | | |
|---|---|---|
| A | B | A⊕B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

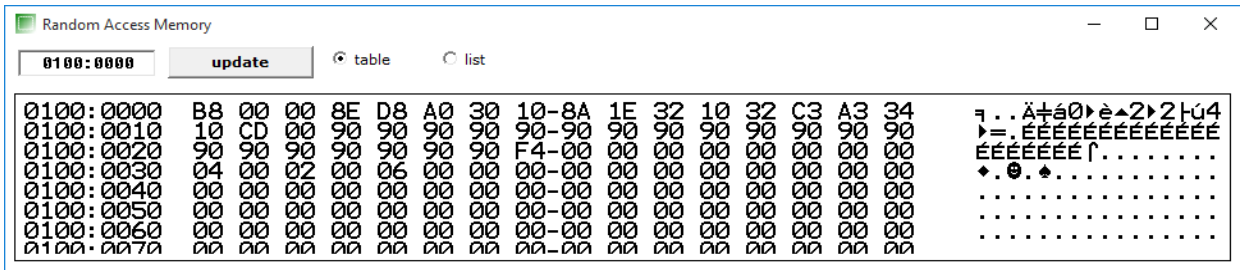
The '**Exclusive-OR**' gate is a circuit which will give a high output if **either, but not both**, of its two inputs are high. An encircled plus sign (⊕) is used to show the EOR operation.

**Program:-**

```
/*8 BIT XORING */
MOV AX,0000H
MOV DS,AX
MOV AL,[1030H]
MOV BL,[1032H]
XOR AL, BL
MOV [1034H],AX
INT
```

**Observation:-**



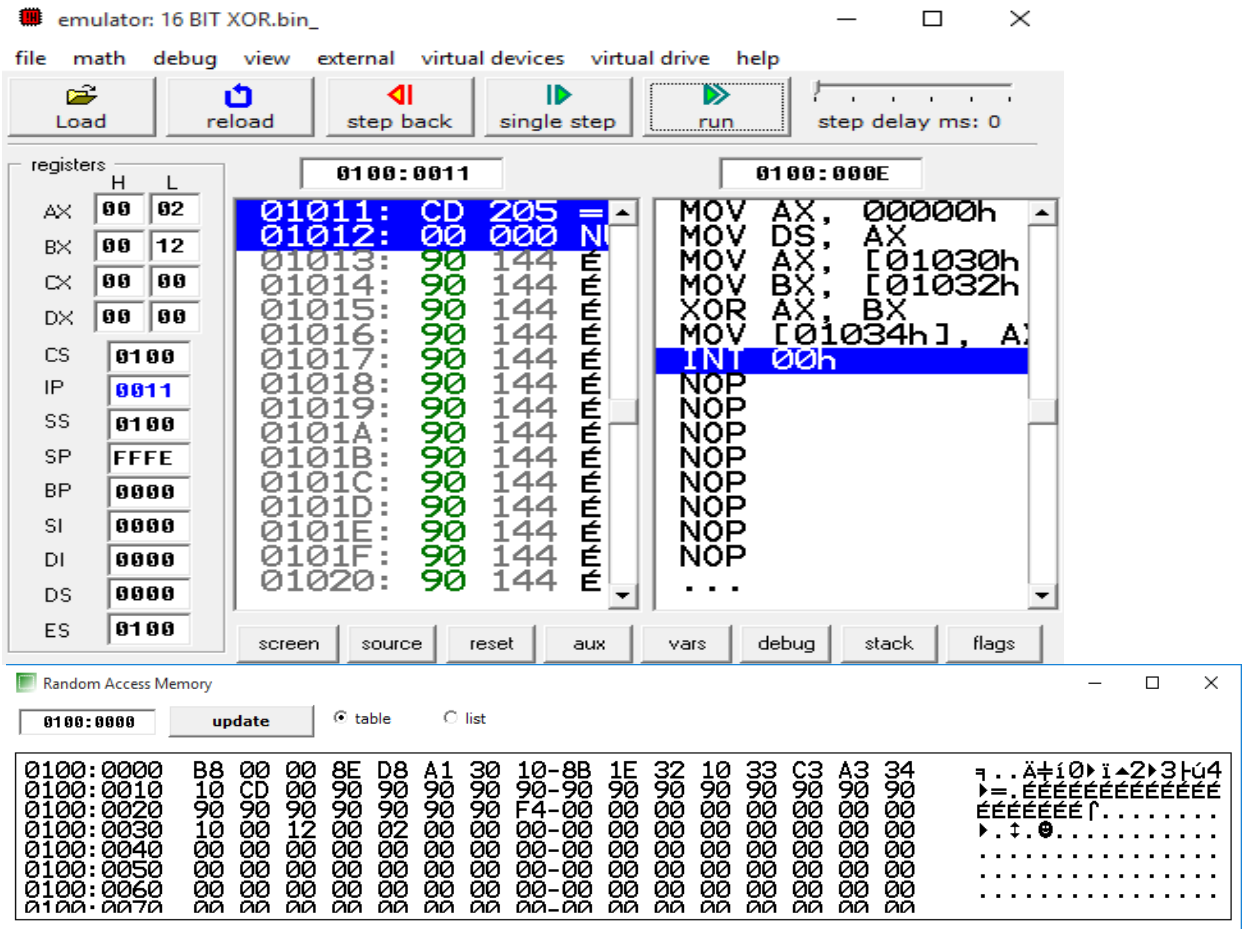

**Result:-**

[1030H]=04
[1032H]=02
[1034H]=06

**Program:-**

/ * 16 BIT XORING */
MOV AX,0000H
MOV DS,AX
MOV AX,[1030H]
MOV BX,[1032H]
XOR AX,BX
MOV [1034H],AX
INT

**Observation:-**




**Result:-**

[1030H]=10

[1032H]=12

[1034H]=02

**Conclusion:-**

In this way we executed assembly language program for 8 bit and 16 bit XORing number.

# Experiment No.8
## Even and Odd number

**Aim:-** Write An Assembly Language Program To Check Whether Entered Number Is Even Or Odd.

**Objective:** Describe even and odd instructions Operation.

**Software**: 8086 Emulator

**Theory:** Describe the Miscellaneous Data Transfer Instructions.

To study and implement the jump instructions to find whether entered number is even or odd.

**Miscellaneous Data Transfer Instructions**

**CMP – CMP Destination, Source**

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:

| | CF | ZF | SF | |
|---|---|---|---|---|
| CX = BX | 0 | 1 | 0 | Result of subtraction is 0 |
| CX > BX | 0 | 0 | 0 | No borrow required, so CF = 0 |
| CX < BX | 1 | 0 | 1 | Subtraction requires borrow, so CF = 1 |

- CMP AL, 01H Compare immediate number 01H with byte in AL
- CMP BH, CL Compare byte in CL with byte in BH
- CMP CX, TEMP Compare word in DS at displacement TEMP with word at CX

**XCHG – XCHG Destination, Source**

The XCHG instruction exchanges the content of a register with the content of another register or with the content of memory location(s). It cannot directly exchange the content of two memory locations. The source and destination must both be of the same type (bytes or words). The segment registers cannot be used in this instruction. This instruction does not affect any flag.

- XCHG AX, DX Exchange word in AX with word in DX
- XCHG BL, CH Exchange byte in BL with byte in CH

**LAHF (COPY LOW BYTE OF FLAG REGISTER TO AH REGISTER)**

The LAHF instruction copies the low-byte of the 8086 flag register to AH register. It can then be pushed onto the stack along with AL by a PUSH AX instruction. LAHF does not affect any flag.

**SAHF (COPY AH REGISTER TO LOW BYTE OF FLAG REGISTER)**

The SAHF instruction replaces the low-byte of the 8086 flag register with a byte from the AH register. SAHF changes the flags in lower byte of the flag register.

**XLAT / XLATB – TRANSLATE A BYTE IN AL**

The XLATB instruction is used to translate a byte from one code (8 bits or less) to another code (8 bits or less). The instruction replaces a byte in AL register with a byte pointed to by BX in a lookup table in the memory. Before the XLATB instruction can be executed, the lookup table containing the values for a new code must be put in memory, and the offset of the starting address of the lookup table must be loaded in BX. The code byte to be translated is put in AL. The XLATB instruction adds the byte in AL to the offset of the start of the table in BX. It then copies the byte from the address pointed to by (BX + AL) back into AL. XLATB instruction does not affect any flag.

8086 routine to convert ASCII code byte to EBCDIC equivalent: ASCII code byte is in AL at the start, EBCDIC code in AL after conversion.

MOV BX, OFFSET EBCDIC Point BX to the start of EBCDIC table in DS

XLATB Replace ASCII in AL with EBCDIC from table.

**IN – IN Accumulator, Port**

The IN instruction copies data from a port to the AL or AX register. If an 8-bit port is read, the data will go to AL. If a 16-bit port is read, the data will go to AX.

The IN instruction has two possible formats, fixed port and variable port. For fixed port type, the 8-bit address of a port is specified directly in the instruction. With this form, any one of 256 possible ports can be addressed.

IN AL, OC8H Input a byte from port OC8H to AL

IN AX, 34H Input a word from port 34H to AX

For the variable-port form of the IN instruction, the port address is loaded into the DX register before the IN instruction. Since DX is a 16-bit register, the port address can be any number between 0000H and FFFFH. Therefore, up to 65,536 ports are addressable in this mode.

MOV DX, 0FF78H Initialize DX to point to port

IN AL, DX Input a byte from 8-bit port 0FF78H to AL

IN AX, DX Input a word from 16-bit port 0FF78H to AX

The variable-port IN instruction has advantage that the port address can be computed or dynamically determined in the program. Suppose, for example, that an 8086-based computer needs to input data from 10 terminals, each having its own port address. Instead of having a separate procedure to input data from each port, you can write one generalized input procedure and simply pass the address of the desired port to the procedure in DX.

The IN instruction does not change any flag.

**OUT – OUT Port, Accumulator**

The OUT instruction copies a byte from AL or a word from AX to the specified port. The OUT instruction has two possible forms, fixed port and variable port.

For the fixed port form, the 8-bit port address is specified directly in the instruction. With this form, any one of 256 possible ports can be addressed.
OUT 3BH, AL Copy the content of AL to port 3BH
OUT 2CH, AX Copy the content of AX to port 2CH

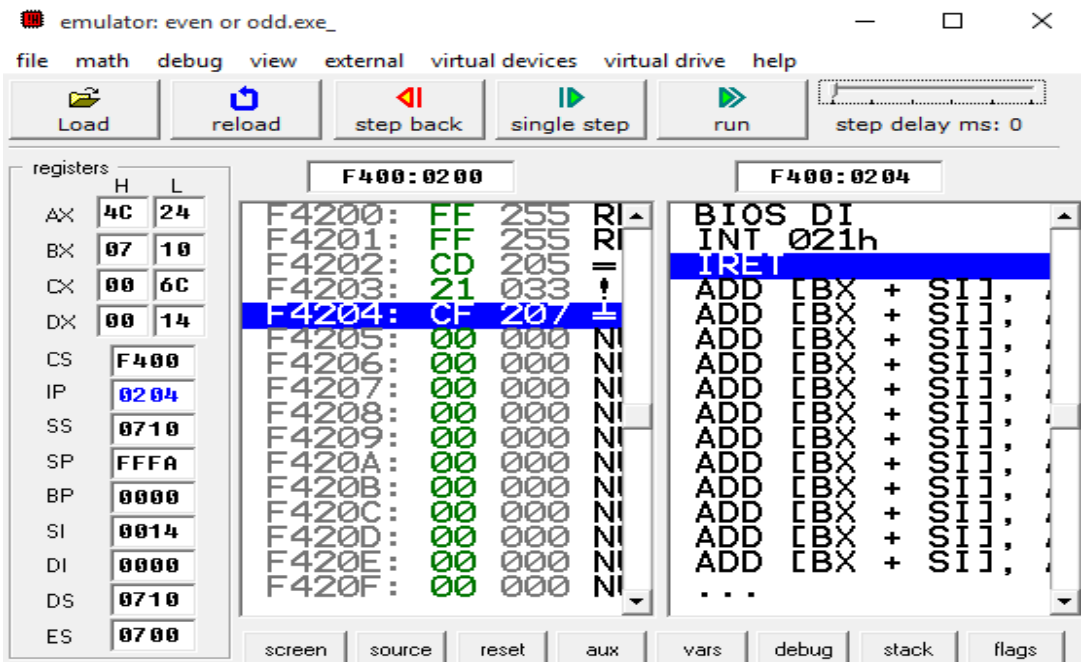
For variable port form of the OUT instruction, the content of AL or AX will be copied to the port at an address contained in DX. Therefore, the DX register must be loaded with the desired port address before this form of the OUT instruction is used.
MOV DX, 0FFF8H Load desired port address in DX
OUT DX, AL Copy content of AL to port FFF8H
OUT DX, AX Copy content of AX to port FFF8H
The OUT instruction does not affect any flag.

**PROGRAM:**

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
MSG DB 10,13,ENTER A NUMBER = $
MSG1 DB 10,13,NUMBER IS EVEN $
MSG2 DB 10,13,NUMBER IS ODD $
DATA ENDS

CODE SEGMENT
START:

MOV BX,DATA
MOV DS,BX

LEA SI,MSG
CALL PRINT

MOV AH,01H
INT 21H

SAR AL,01
JC ODD

LEA SI,MSG1
CALL PRINT

JMP TERMINATE

ODD:
LEA SI,MSG2
CALL PRINT

TERMINATE:
```

```
MOV AH,4CH
INT 21H

PRINT PROC
MOV DX,SI
MOV AH,09H
INT 21H
RET
PRINT ENDP

CODE ENDS
END START
```

**Observation:-**




**Conclusion:-**

In this way we executed assembly language program to check whether the number is odd or even.

# Experiment No.9
# ASCII number to packed BCD

**Aim:-** Write assembly language program to perform conversion from ASCII number to packed BCD

**Objective:** To study the concept of ASCII in Assembly Language programming.

To implement the instructions related to ASCII arithmetic.

**Software**: 8086 Emulator

**Theory:**

***ASCII Arithmetic***

***AAA (ASCII ADJUST FOR ADDITION)***

Numerical data coming into a computer from a terminal is usually in ASCII code. In this code, the numbers 0 to 9 are represented by the ASCII codes 30H to 39H. The 8086 allows you to add the ASCII codes for two decimal digits without masking off the "3" in the upper nibble of each. After the addition, the AAA instruction is used to make sure the result is the correct unpacked BCD.
Let AL = 0011 0101 (ASCII 5), and BL = 0011 1001 (ASCII 9)

ADD AL, BL AL = 0110 1110 (6EH, which is incorrect BCD)
AAA AL = 0000 0100 (unpacked BCD 4)
CF = 1 indicates answer is 14 decimal.
The AAA instruction works only on the AL register. The AAA instruction updates AF and CF; but OF, PF, SF and ZF are left undefined.

**AAS (ASCII ADJUST FOR SUBTRACTION)**

Numerical data coming into a computer from a terminal is usually in an ASCII code. In this code the numbers 0 to 9 are represented by the ASCII codes 30H to 39H. The 8086 allows you to subtract the ASCII codes for two decimal digits without masking the "3" in the upper nibble of each. The AAS instruction is then used to make sure the result is the correct unpacked BCD.
Let AL = 00111001 (39H or ASCII 9), and BL = 00110101 (35H or ASCII 5)

SUB AL, BL AL = 00000100 (BCD 04), and CF = 0
AAS AL = 00000100 (BCD 04), and CF = 0 (no borrow required)
Let AL = 00110101 (35H or ASCII 5), and BL = 00111001 (39H or ASCII 9)

SUB AL, BL AL = 11111100 (– 4 in 2's complement form), and CF = 1
AAS AL = 00000100 (BCD 06), and CF = 1 (borrow required)
The AAS instruction works only on the AL register. It updates ZF and CF; but OF, PF, SF, AF are left undefined.

***AAM (BCD ADJUST AFTER MULTIPLY)***

Before you can multiply two ASCII digits, you must first mask the upper 4 bit of each. This leaves unpacked BCD (one BCD digit per byte) in each byte. After the two unpacked BCD digits are multiplied, the AAM instruction is used to adjust the product to two unpacked BCD digits in AX.

AAM works only after the multiplication of two unpacked BCD bytes, and it works only the operand in AL. AAM updates PF, SF and ZF but AF; CF and OF are left undefined.
Let AL = 00000101 (unpacked BCD 5), and BH = 00001001 (unpacked BCD 9)
MUL BH AL x BH: AX = 00000000 00101101 = 002DH
AAM AX = 00000100 00000101 = 0405H (unpacked BCD for 45)

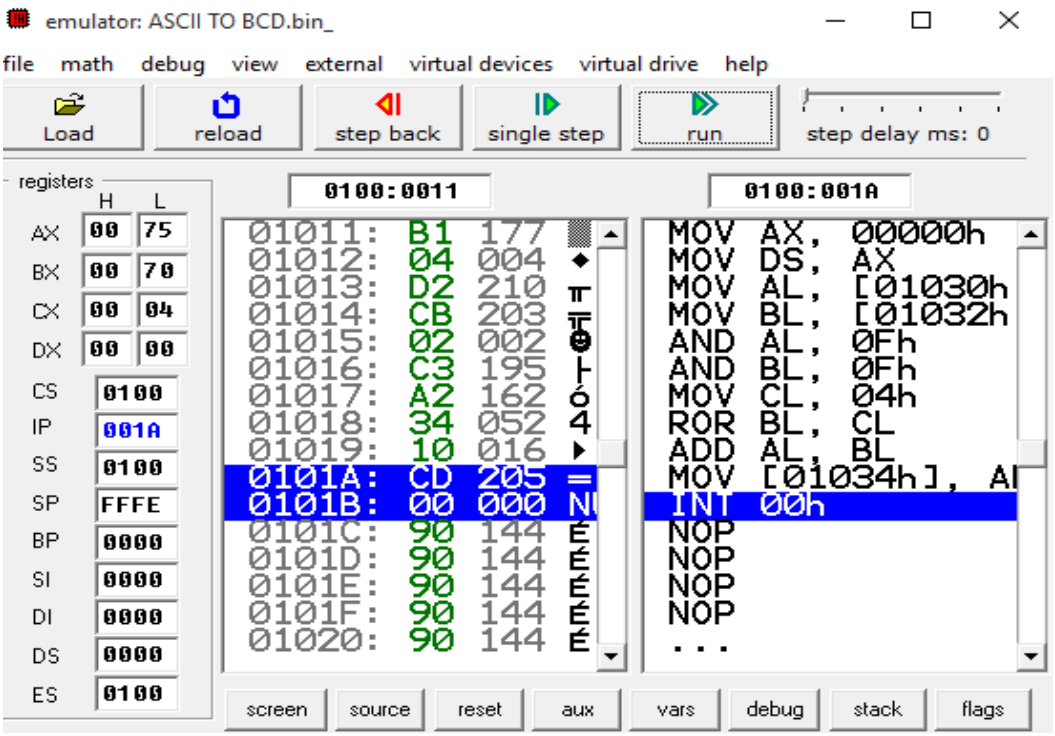
**AAD (BCD-TO-BINARY CONVERT BEFORE DIVISION)**

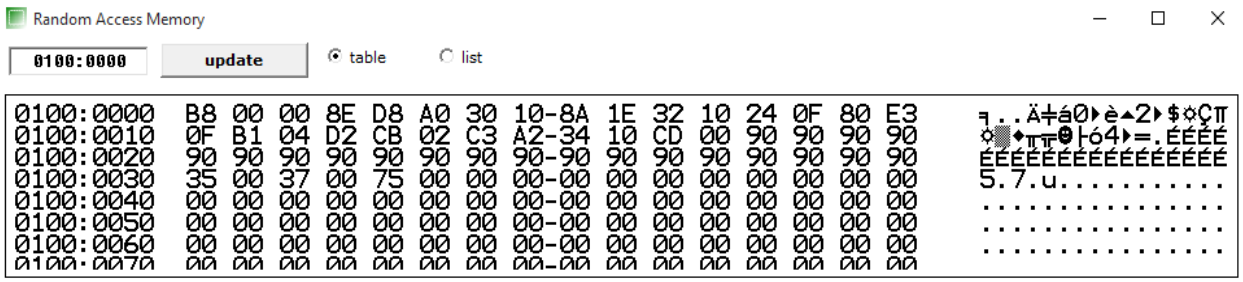
AAD converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. After the BCD division, AL will contain the unpacked BCD quotient and AH will contain the unpacked BCD remainder. AAD updates PF, SF and ZF; AF, CF and OF are left undefined.
Let AX = 0607 (unpacked BCD for 67 decimal), and CH = 09H
AAD AX = 0043 (43H = 67 decimal)
DIV CH AL = 07; AH = 04; Flags undefined after DIV
If an attempt is made to divide by 0, the 8086 will generate a type 0 interrupt.

**Program:-**

```
MOV AX,0000
MOV DS,AX
MOV AL,[1030H]
MOV BL,[1032H]
AND AL,0FH
AND BL,0FH
MOV CL,04H
ROR BL,CL
ADD AL,BL
MOV [1034H],AL
INT
```

**Observation:-**






**Result:-**

[1030H]=35

[1032H]=37

[1034H]=75

**Conclusion:-**

In this way we executed assembly language program to conversion from ASCII number to packed BCD.

# Experiment No.10
## Calculate conversion of temperature

**Aim:-** Write assembly language program to calculate conversion of temperature

**Objective:** Describe the Logical Instructions.
To study and implement the concept of arrays in Assembly Language Programming.

**Software**: 8086 Emulator

**Theory:**

**LOGICAL INSTRUCTIONS**

**AND – AND Destination, Source**
This instruction ANDs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.
The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and the destination cannot both be memory locations. CF and OF are both 0 after AND. PF, SF, and ZF are updated by the AND instruction. AF is undefined. PF has meaning only for an 8-bit operand.
AND CX, [SI] AND word in DS at offset [SI] with word in CX register;
Result in CX register
AND BH, CL AND byte in CL with byte in BH; Result in BH
AND BX, 00FFH 00FFH Masks upper byte, leaves lower byte unchanged.

**OR – OR Destination, Source**
This instruction ORs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.
The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and destination cannot both be memory locations. CF and OF are both 0 after OR. PF, SF, and ZF are updated by the OR instruction. AF is undefined. PF has meaning only for an 8-bit operand.
OR AH, CL CL ORed with AH, result in AH, CL not changed
OR BP, SI SI ORed with BP, result in BP, SI not changed
OR SI, BP BP ORed with SI, result in SI, BP not changed
OR BL, 80H BL ORed with immediate number 80H; sets MSB of BL to 1

**XOR – XOR Destination, Source**
This instruction Exclusive-ORs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.
The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and destination

cannot both be memory locations. CF and OF are both 0 after XOR. PF, SF, and ZF are updated. PF has meaning only for an 8-bit operand. AF is undefined.
XOR CL, BH Byte in BH exclusive-ORed with byte in CL.
Result in CL. BH not changed.
XOR BP, DI Word in DI exclusive-ORed with word in BP.
Result in BP. DI not changed.

**NOT – NOT Destination**
The NOT instruction inverts each bit (forms the 1's complement) of a byte or word in the specified destination. The destination can be a register or a memory location. This instruction does not affect any flag.
NOT BX Complement content or BX register

**NEG – NEG Destination**
This instruction replaces the number in a destination with its 2's complement. The destination can be a register or a memory location. It gives the same result as the *invert each bit and add one* algorithm. The NEG instruction updates AF, AF, PF, ZF, and OF.
 NEG AL Replace number in AL with its 2's complement
 NEG BX Replace number in BX with its 2's complement

**TEST – TEST Destination, Source**
This instruction ANDs the byte / word in the specified source with the byte / word in the specified destination. Flags are updated, but neither operand is changed. The test instruction is often used to set flags before a Conditional jump instruction.
The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and the destination cannot both be memory locations. CF and OF are both 0's after TEST. PF, SF and ZF will be updated to show the results of the destination. AF is be undefined.
TEST AL, BH AND BH with AL. No result stored; Update PF, SF, ZF.
TEST CX, 0001H AND CX with immediate number 0001H;
No result stored; Update PF, SF, ZF
TEST BP, [BX][DI] AND word are offset [BX][DI] in DS with word in BP.
No result stored. Update PF, SF, and ZF

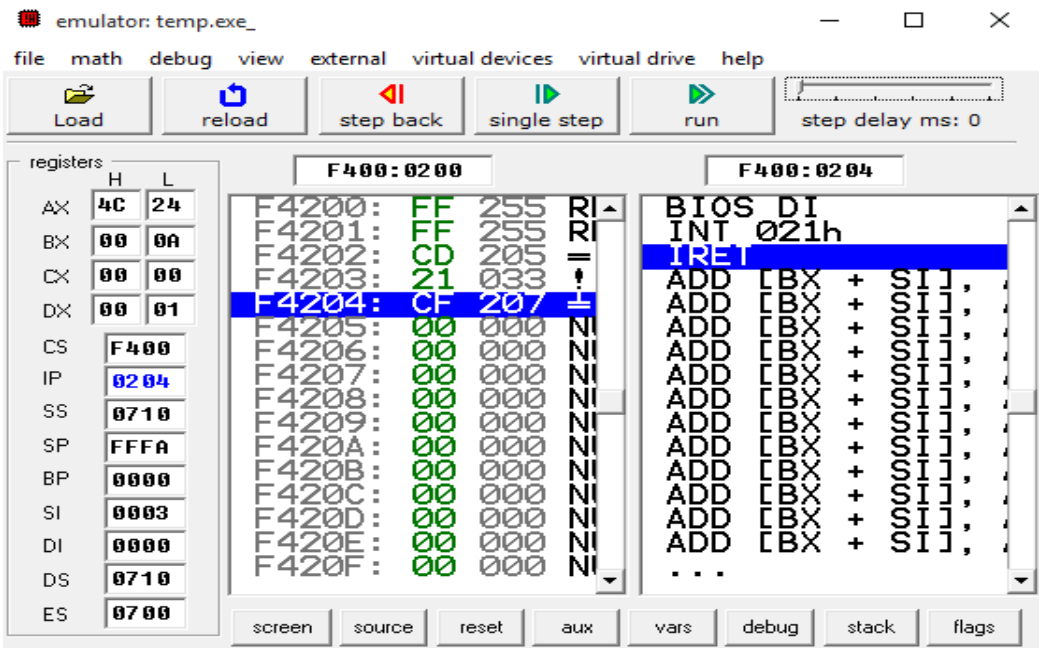
**Program:-**

```
DATA SEGMENT
    T    DB ?
    RES  DB 10 DUP ('$')
    MSG1 DB "ENTER TEMPERATURE IN CELSIUS (ONLY IN 2 DIGITS) : $"
    MSG2 DB 10,13,"CONVERTED IS FAHRENHEIT (TEMPERATURE) : $"

DATA ENDS
CODE SEGMENT
      ASSUME DS:DATA,CS:CODE
START:
   MOV AX,DATA
```

```
    MOV DS,AX

    LEA DX,MSG1
    MOV AH,9
    INT 21H

    MOV AH,1
    INT 21H

    SUB AL,30H
    MOV AH,0

    MOV BL,10
    MUL BL
    MOV BL,AL

    MOV AH,1
    INT 21H

    SUB AL,30H
    MOV AH,0
    ADD AL,BL
    MOV T,AL

    MOV DL,9
    MUL DL

    MOV BL,5
    DIV BL
    MOV AH,0

    ADD AL,32

    LEA SI,RES
    CALL HEX2DEC

    LEA DX,MSG2
    MOV AH,9
    INT 21H

    LEA DX,RES
    MOV AH,9
    INT 21H

    MOV AH,4CH
    INT 21H
CODE ENDS
```

```
HEX2DEC PROC NEAR
   MOV CX,0
   MOV BX,10

LOOP1: MOV DX,0
      DIV BX
      ADD DL,30H
      PUSH DX
      INC CX
      CMP AX,9
      JG LOOP1

      ADD AL,30H
      MOV [SI],AL

LOOP2: POP AX
      INC SI
      MOV [SI],AL
      LOOP LOOP2
      RET
HEX2DEC ENDP

END START
```

**Observation:-**




**Conclusion:-**

In this way we executed assembly language program to calculate conversion of temperature.

# Experiment No.11

## Study of BIOS and DOS

**Aim:- Study of BIOS and DOS**

**Objective:** Describe the basic input output system interrupts.
To study disk operating system interrupts.

**Theory:**

BIOS interrupt calls are a facility that operating systems and application programs use to invoke the facilities of the Basic Input/Output System on IBM PC compatible computers. Traditionally, BIOS calls are mainly used by MS-DOS programs and some other software such as boot loaders (including, mostly historically, relatively simple application software that boots directly and runs without an operating system—especially game software). BIOS only runs in the real address mode (Real Mode) of the x86 CPU, so programs that call BIOS either must also run in real mode or must switch from protected mode to real mode before calling BIOS and then switch back again. For this reason, modern operating systems that use the CPU in Protected Mode generally do not use the BIOS to support system functions, although some of them use the BIOS to probe and initialize hardware resources during their early stages of booting.

### Purpose of BIOS calls

The BIOS also frees computer hardware designers (to the extent that programs are written to use the BIOS exclusively) from being constrained to maintain exact hardware compatibility with old systems when designing new systems, in order to maintain compatibility with existing software. In addition to giving access to hardware facilities, BIOS provides added facilities that are implemented in the BIOS software.

### Calling BIOS: BIOS software interrupts

Operating systems and other software communicates with the BIOS software, in order to control the installed hardware, via software interrupts. A software interrupt is a specific variety of the general concept of an interrupt. An interrupt is a mechanism by which the CPU can be directed to stop executing the main-line program and immediately execute a special program, called an Interrupt Service Routine (ISR).

BIOS interrupt calls can be thought of as a mechanism for passing messages between BIOS and the operating system or other BIOS client software.

The BIOS software usually returns to the caller with an error code if not successful, or with a status code and/or requested data if successful. The data itself can be as small as one bit or as large as 65536 bytes of whole raw disk sectors (the maximum that will fit into one real-mode memory segment). BIOS has been expanded and enhanced over the years many times by many different corporate entities, and unfortunately the result of this evolution is that not all the BIOS functions that can be called use consistent conventions for formatting and communicating data

or for reporting results. Some BIOS functions report detailed status information, while others may not even report success or failure but just return silently, leaving the caller to assume success (or to test the outcome some other way). Sometimes it can also be difficult to determine whether or not a certain BIOS function call is supported by the BIOS on a certain computer, or what the limits of a call's parameters are on that computer.

**Invoking an interrupt**

Invoking an interrupt can be done using the INT x86 assembly language instruction. For example, to print a character to the screen using BIOS interrupt 0x10, the following x86 assembly language instructions could be executed:

```
mov ah, 0x0e    ; function number = 0Eh : Display Character
mov al, '!'     ; AL = code of character to display
int 0x10        ; call INT 10h, BIOS video service
```

Interrupt table

*Main article:* Interrupt vector table

A list of common BIOS interrupt classes can be found below. Note that some BIOSes (particularly old ones) do not implement all of these interrupt classes.

BIOS also uses some interrupts to relay hardware event interrupts to programs which choose to receive them or to route messages for its own use. The table below includes only those BIOS interrupts which are intended to be called by programs (using the "INT" assembly-language software interrupt instruction) to request services or information.

Interrupt types 0 to 1Fh are known as BIOS interrupts. This is because most of these service routines are BIOS routines residing in the ROM.

**Interrupt Types 0-7**Interrupt types 0-7 are reserved by Intel, with types 0-4 being predefined. IBM uses type 5 for print screen. Types 6 and 7 are not used.

**Interrupt 0 -- Divide Overflow**
A type 0 interrupt is generated when a DIV or IDIV operation produces an overflow. This occurs when the quotient can not fit in the destination register. The interrupt 0 routine displays the message "DIVIDE OVERFLOW" and returns control to DOS.

**Interrupt 1 -- Single Step**
Single-stepping is a useful debugging tool to observe the behavior of a program instruction by instruction. A type 1 interrupt is generated when the **Trap Flag (TF)** is set. The ISR for a type 1 interrupt can be used to display relevant information about the state of the program. For example, the contents of all registers can be displayed. To end single-stepping, the TF should be cleared. Note that there are dedicated instructions for setting and clearing the TF.

**Interrupt 2 -- Non-Maskable Interrupt**
Interrupt 2 is the hardware interrupt that cannot be masked out by clearing the TF. The IBM PC uses this interrupt to signal memory and I/O parity errors that indicate bad chips.

**Interrupt 3 -- Breakpoint**
The INT 3 instruction is the only single-byte interrupt instruction (opcode: CCh); other interrupt instructions are two-byte instructions. Inserting a breakpoint in a program involves replacing the program code byte by CCh while saving the program byte for later restoration to remove the breakpoint. Inserting breakpoints is used by program debuggers.

**Interrupt 4 -- Overflow**
A type 4 interrupt is generated by the instruction **INTO** (interrupt if overflow) when the OF is set. Programmers may write their own routines to handle unexpected overflows. Note that executing the instruction INT 4 will invoke the ISR for this interrupt type unconditionally while INTO invokes it conditionally on the OF. INTO is not used normally as the overflow condition is usually detected and processed using the instructions JO and JNO.

**Interrupt 5 -- Print Screen**
The BIOS interrupt 5 routine sends the video screen information to the printer. An INT 5 instruction is generated by the keyboard interrupt routine (interrupt type 9) when the PrtSC (print screen) key is pressed.

**Interrupt Types 8h-Fh**
The 8086 has only one pin, INTR pin, for maskable hardware interrupt signals. To allow more devices to interrupt the 8086, IBM uses an interrupt controller, the Intel 8259 Programmable Interrupt Controller chip, which can interface up to eight devices. Interrupt types 8h-Fh are generated by hardware devices connected to the 8259 chip. The original version of the PC uses only interrupts 8, 9, and Eh.

**Interrupt 8 -- Timer**
The IBM PC contains a timer circuit that generates an interrupt once every 54.92 milliseconds (about 18.2 times per second). The BIOS interrupt 8 routine services the timer circuit. It uses the timer signals (ticks) to keep track of the time of the day.

**Interrupt 9 -- Keyboard**
The interrupt 9 is generated by the keyboard each time a key is pressed or released. The BIOS interrupt 9 routine reads a scan code and stores it in the keyboard buffer. In addition, it also identifies special key combinations such ctrl-break. The keyboard buffer has the capacity to store up to 15 keys. When the buffer is full, pressing a key causes the BIOS to beep, indicating that the key stroke is lost.

The keyboard controller supplies the key identity by means of a *scan code*. The scan code of a key is simply an identification number given to the key based on its location in the keyboard. The scan code has no relation to the ASCII code. The interrupt 9 routine receives the scan code and generates the equivalent ASCII code, if there is one. Both the scan code and the ASCII code are placed in the keyboard buffer.

**Interrupt E -- Diskette Error**
The BIOS interrupt Eh handles diskette errors.

**Interrupt Types 10h-1Fh**

The interrupt routines 10h-1Fh can be called by application programs to perform various I/O operations and status checking.

**Interrupt 10h -- Video**
The BIOS interrupt 10h routine is the video driver. Associated with each I/O device. There is a device controller or I/O controller that acts as a hardware interface between the processor and the I/O device. The device controller performs many of the low-level tasks specific to the I/O device. This allows the CPU to interact with the device at higher level. For each device controller, there is a software interface that provides a clean interface to access the device. This software interface is called the *device driver*.

**Interrupt 11h -- Equipment Check**
The BIOS interrupt 11h routine returns the equipment configuration of the particular PC. The return code is placed in register AX.

**Interrupt 12h -- Memory Size**
The BIOS interrupt 12h routine returns in register AX the amount of conventional memory a computer has. Conventional memory refers to memory circuits with addresses below 640 KByte. The unit for the return value is in Kbytes.

**Interrupt 13h -- Disk I/O**
The BIOS interrupt 13h routine is the disk driver. It allows application programs to do disk I/O.

**Interrupt 14h -- Communications**
The BIOS interrupt 14h routine is the communications driver that interacts with the serial ports.

**Interrupt 15h -- Cassette**
This interrupt was used by the original PC for cassette interface.

**Interrupt 16h -- Keyboard I/O**
The BIOS interrupt 16h routine is the keyboard driver. BIOS provides keyboard service routines under INT 16H. We list here some examples.

To read a character from the keyboard, function 0 in AH is used as follows:

**Function 00H -- Read a character from the keyboard**
Input: AH=00H
Returns: if AL<>0 then
AL = ASCII code of the key entered
AH = Scan code of the key entered
if AL = 0
AH = Scan code of the extended key entered

This BIOS function can be used to read a character from the keyboard. If the keyboard buffer is empty, it waits for a character to be entered. The value returned in AL determines if the key represents an ASCII character or an extended key character. In both cases, the scan code is placed in the AH register and the ASCII and scan codes are removed from the keyboard buffer.

To check if the keyboard buffer is empty or not, we can use function 1 as follows:

**Function 01H -- Check keyboard buffer**
Input: AH=01H
Returns: ZF= 1 if the keyboard buffer is empty
ZF= 0 if there is at least one character available.
In this case, the ASCII and scan codes are placed in the AL and AH registers as in function 00. The codes, however, are not removed from the buffer.

Function 2 can be used to check keyboard status with regard to shift and toggle keys.

**Function 02H -- Check keyboard status**
Input: AH=02H
Returns: AL= status of the shift and toggle keys

The following table indicates the bit assignment for shift and toggle keys. A bit with a value of 1 indicates the presence of a condition.

| Bits in AL | Key Assignment |
|---|---|
| 7 | Ins lock switch is on |
| 6 | Caps lock switch is on |
| 5 | Num lock switch is on |
| 4 | Scroll lock switch is on |
| 3 | Alt key depressed |
| 2 | Ctrl key depressed |
| 1 | Left shift key depressed |
| 0 | Right shift key depressed |

**Interrupt 17h -- Printer I/O**
The BIOS interrupt 17h routine is the printer driver. The routine supports three functions: 0-2.

- Function 0 writes a character to a printer; input values are AH=0, AL=character to be printed, DX=printer number (0=LPT1, 1=LPT2, 3=LPT3).
- Function 1 initializes a printer port; input values are AH=1, DX=printer number.
- Function 2 gets printer status; input values are AH=2, DX= printer number.

For all functions the status is returned in AH. The following table shows the meaning of the bits returned in AH:

| Bits in AH | Meaning |
|---|---|
| 7 | = 1 printer not busy |
| 6 | = 1 print acknowledge |
| 5 | = 1 out of paper |
| 4 | = 1 printer online |
| 3 | = 1 I/O error |

| | |
|---|---|
| 2 | = 1 not used |
| 1 | = 1 not used |
| 0 | = 1 printer timed-out |

Next, we show an example for printing character 0 on the printer. Because printers contain buffers for data, the 0 will not be printed until a carriage return or line feed character is sent.

```
MOV AH, 0       ; function 0, print character
MOV AL, '0'     ; character to be printed in AL
MOV DX, 0       ; Printer 0 (LPT1)
INT 17H         ; AH contains return code
MOV AH, 0       ; function 0, print character
MOV AL, 0AH     ; line feed
INT 17H
```

**Interrupt 18h -- BASIC**
The BIOS interrupt 18h routine transfers control to ROM BASIC.

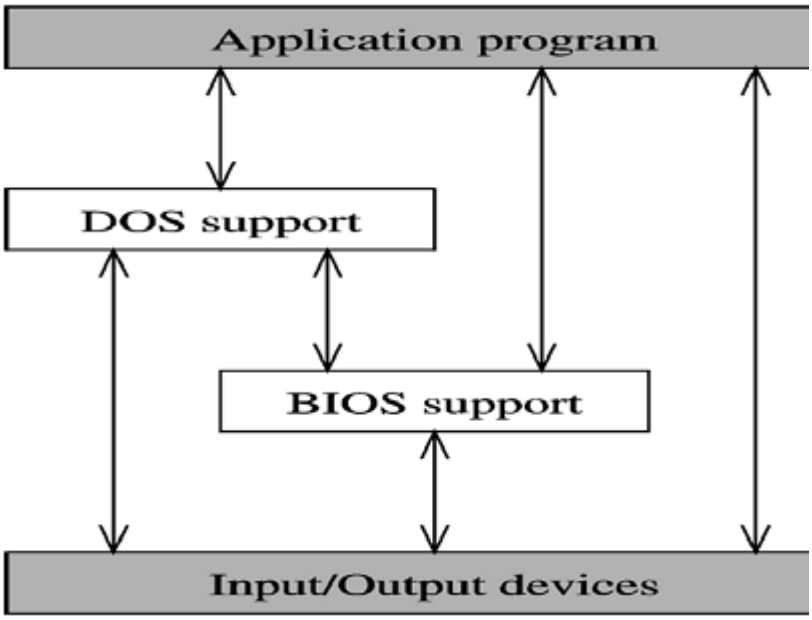
**Interrupt 19h -- Bootstrap**
The BIOS interrupt 19h routine reboots the system.

**Interrupt 1Ah -- Time of Day**
The BIOS interrupt 1Ah routine allows a program to get and set the timer tick count.

**Interrupt 1Bh -- Ctrl Break**
This interrupt is called by the INT 9 routine when the Ctrl-break key is pressed. The BIOS interrupt 1Bh routine contains only an IRET instruction. Users may write their own routine to handle the Ctrl-break key.

**Interrupt 1Ch -- Timer Tick**
Interrupt 1Ch is called by the INT 8 routine each time the timer circuit interrupts. The BIOS interrupt 1Ch routine contains only an IRET instruction. Users may write their own service routine to perform timing operations.

**Interrupts 1Dh-1Fh**
These interrupt vectors point to data instead of instructions. The interrupt 1Dh, 1Eh, and 1Fh point to video initialization parameters, diskette parameters, and video graphics characters, respectively.




**DOS Interrupts**

MS-DOS provides many common services through INT 21h. Entire books have been written about the variety of functions available; I will just list the most basic ones for console input and output here.

- **Input a character**.
- MOV AH, 01h
- INT 21h

  After the interrupt, AL contains the ASCII code of the input character. The character is echoed (displayed on the screen). Use function code 8 instead of 1 for no echo.

- **Input a string**.
- SECTION .data
- Buffer DB BUFSIZE ;BUFSIZE is max number of chars to read, <= 255
- RESB BUFSIZE + 1
-
-
- SECTION .text
- MOV DX, Buffer
- MOV AH, 0Ah
- INT 21h

  After the interrupt, BYTE [Buffer + 1] will contain the number of characters read, and the characters themselves will start at Buffer + 2. The characters will be terminated by a carriage return (ASCII code 13), although this will not be included in the count.

- **Output a character**.
- MOV DL, ...
- MOV AH, 02h

- INT 21h

  Load the desired character into DL, then call the interrupt with function code 2 in AH.

- **Output a string**.
- MOV DX, ...
- MOV AH, 09h
- INT 21h

  Load the address of a '$'-terminated string into DX, then call the interrupt with function code 9 in AH.

- **Exit**.
- MOV AL, ...
- MOV AH, 4Ch
- INT 21h

  Load the return code (0 for normal exit, non-zero for error) into AL, then call the interrupt with function code 4Ch in AH. This is the proper DOS exit routine; however, if you are running your program with DEBUG, this will exit DEBUG. An alternative is to go behind DOS's back and use the BIOS routine accessed by INT 20h, which will return control to the DEBUG prompt when executed.

**Conclusion:-** In this we studied BIOS and DOS interrupts

# Experiment No.12

## Study of TSR

**Aim:- Study of TSR**

**Objective:** To study Keyboard interrupt (int 9h), Control-C interrupt (int 23h), Control-break interrupt (int 1bh), Critical error interrupt (int 24h), BIOS disk interrupt (int 124 A to Z of C 13h), Timer interrupt (int 1ch) and DOS Idle interrupt (int 28h) using TSR

.

**Theory:**

TSR or "Terminate and Stay Resident" Programming is one of the interesting topics in DOS Programming. TSR programs are the one which seems to terminate, but remains resident in memory. So the resident program can be invoked at any time. Few TSR programs are written with the characteristic of TCR (Terminate Continue Running) i.e., TSR program seems to terminate, but continues to run in the background. TSR Programming is supposed to be an easy one, if you know the DOS internals.

TSR Programming concept in a simpler manner DOS's non-reentrancy Problem If a function can be called before it is finished, it is called reentrant. Unfortunately, DOS functions are non-reentrant. That is, we should not call a DOS function when it executes the same. Now, our intuition suggests us to avoid the DOS functions in TSR programs! 27.2 Switching Programs As we know, DOS is not a multitasking operating system. So DOS is not meant for running two or more programs simultaneously! One of the major problems we face in TSR programming is that DOS's nature of switching programs. DOS handles switching programs, by simply saving the swapped-out program's complete register set and replacing it with the swapped-in program's registers. In DOS, if a program is put to sleep its registers are stored in an area called TCB (Task Control Block). We must finish one process before another is undertaken. The main idea behind it is that, whenever we switch between programs, DOS switches our program's stack to its own internal set. And whatever that is pushed must be fully popped. For example, assume that we have a process currently running called previous-process, and we initiate another process in the meantime called current-process. In this case, the current-process will work fine, but when the previous-process just gets finished, it would find its stack data has been trashed by current process.

DOS Busy Flag From the above discussion, we understand that before popping up our TSR program, we must check whether DOS is currently executing an internal routine (i.e., busy) or not. Surprisingly DOS also checks its status using a flag called "DOS Busy Flag". This "DOS Busy Flag" feature is undocumented and some programmers refer this flag as "DOS Critical Flag". We can also use this flag in our TSR program to check whether DOS is busy or not. For that, we have to use undocumented DOS function 34h.BIOS Functions As BIOS functions are reentrant, some programmers use BIOS functions in TSR programs. But professional programmers don't use BIOS functions, as the implementation of BIOS functions is quite different from machine to machine. In other words, BIOS is not compatible and there is no guarantee for its reentrancy. So for professional TSR programming, avoid BIOS functions too! Popping up TSR TSR programs can be made to reside in memory with the keep( ) function. Then how does our TSR program understand, it is being requested by user? In other words, when to popup our TSR program? For that, we have

to capture few interrupts. We have already seen that interrupt routines will be called whenever an interrupt is been generated. So if we replace the existing interrupt routine with our routine,

TSR programmers capture Keyboard interrupt (int 9h), Control-C interrupt (int 23h), Control-break interrupt (int 1bh), Critical error interrupt (int 24h), BIOS disk interrupt (int 124 A to Z of C 13h), Timer interrupt (int 1ch) and DOS Idle interrupt (int 28h). Indian TSR programmers often use int 8h as Timer interrupt. But other international TSR programmers use int 1ch as Timer interrupt. The idea is that we have to block Control-C interrupt, Control-break interrupt and Critical error interrupt. Otherwise, there is a chance that the control will pass onto another program when our TSR program is in action. And it will spoil everything! We must also monitor other interrupts—Keyboard interrupt, BIOS disk interrupt, Timer interrupt and DOS Idle interrupt, and we have to chain them. I hope by looking at the figure, you can understand the concept better. 27.6 IBM's Interrupt-Sharing Protocol Almost all TSR utilities came with the property of unloading itself from the memory. But in order to unload the TSR, it must be the last TSR loaded. For example, if we run TSR utilities namely "X" and "Y", we can unload only the last TSR loaded i.e., "Y". The problem here is that of sharing of interrupts by TSR programs. IBM has suggested a protocol for sharing system interrupts. Even though, this protocol is meant for sharing hardware interrupts, it can be used for software interrupts too. It is especially useful for unloading TSR programs from memory, irrespective of its loading sequence. That is, if we follow this protocol standard, we can unload any TSR at any time! So, in order to unload any TSR at any time, all the TSR programs must use this protocol. But unfortunately, TSR programmers don't use this standard. If you are very particular to know more about this protocol, checkout the Intshare.doc file found on CD.

Rules for TSR Programming

It is wise to consider the following rules, when you programming TSR:

1. Avoid DOS functions. If possible, avoid BIOS functions too!
2. When DOS busy flag is non-zero, DOS is executing interrupt 21h function. So we must wait and watch DOS busy flag.
3. When DOS is busy waiting for console input, we can disturb DOS regardless of the DOS busy flag setting. So you should watch interrupt 28h.
4. Use "signature" mechanism to check whether the TSR is already loaded or not. And so prevent multiple copies.
5. Our TSR program must use its own stack, and not that of the running process.
6. Other TSR programs might be chained to interrupts. So we must also chain any interrupt vector that our program needs.
7. TSR programs should be compiled in Small memory model.
8. However you may need to compile in compact, large or huge memory model if you use file operations with get data( ) and set data( ) functions.
9. TSR programs should be compiled with stack checking turned off.

**Conclusion:- In this way we studied TSR.**

## 3.Quiz on the subject:-

1. **Explain architecture of 8086.**
2. **Write assembly language program to perform 8 bit addition.**
3. **Explain about flag register.**
4. **Write assembly language program to perform 16 bit addition.**
5. **Explain about general purpose register.**
6. **What is AX,BX,CX,DX?**
7. **What is EAX,EBX,ECX,EDX?**
8. **Write assembly language program to perform 8 bit subtraction.**
9. **What Is RAX,RBX,RDX,RCX?**
10. **Write assembly language program to perform 16 bit subtraction.**
11. **What will stands for A IN AX,B IN BX,C IN CX,D IN DX IN 8086 .**
12. **Write assembly language program to perform 8 bit multiplication.**
13. **What are different data transfer instructions?**
14. **Write assembly language program to perform 16 bit multiplication.**
15. **Which is clock generator series is used fro 8086.**
16. **Write assembly language program to perform 8 bit division.**
17. **What is bus?**
18. **Write assembly language program to perform 16 bit division.**
19. **Tell me different types of buses are used in 8086?**
20. **What is advantages of microprocessor?**
21. **Where it is situated in Computer?**

## 4. Conduction of Viva-Voce Examinations:

Teacher should conduct oral exams of the students with full preparation. Normally, the objective questions with guess are to be avoided. To make it meaningful, the questions should be such that depth of the students in the subject is tested. Oral examinations are to be conducted in cordial environment amongst the teachers taking the examination. Teachers taking such examinations should not have ill thoughts about each other and courtesies should be offered to each other in case of difference of opinion, which should be critically suppressed in front of the students.

## 5. Evaluation and marking system:

Basic honesty in the evaluation and marking system is absolutely essential and in the process impartial nature of the evaluator is required in the examination system to become. It is a primary responsibility of the teacher to see that right students who are really putting up lot of hard work with right kind of intelligence are correctly awarded.

The marking patterns should be justifiable to the students without any ambiguity and teacher should see that students are faced with just circumstances.