Vidya Vikas Education Trust's

Universal College of Engineering, Kaman Road, Vasai-401212

Accredited by B+ Grade by NAAC

Department of Computer Engineering

MICROPROCESSOR LAB
(CSL 404)

Lab Manual

Name of the student:

VIVEK. SHIVAKUMAR. HOTTI

Roll No:

31

Class & Division:

S.E Computer Engineering : SE-A

Semester:

Sem IV

2020-21

Lab Code	Lab Name	Credits
CSL404	Microprocessor Lab	1

Lab Outcomes: At the end of the course, the students will be able to	
1	Use appropriate instructions to program microprocessor to perform various task
2	Develop the program in assembly/ mixed language for Intel 8086 processor
3	Demonstrate the execution and debugging of assembly/ mixed language program

Term Work:	
1	The distribution of marks for term work shall be as follows: Lab/ Experimental Work: 15 Assignments: 05 Attendance (Theory & Practical): 05

List of Experiments	
Sr. No.	**Title**
1	Write an assembly language program to perform 8-bit addition.
2	Assembly Program to transfer n block of data to one segment to another segment.
3	Assembly language program for SORTING 8-bit numbers in ascending and descending order.
4	Assembly language program for the Fibonacci Series and Factorial of a number
5	To implement mixed language programming using Assembly and C.
6	Assembly language program for implementing DOS Interrupt.
7	Assembly language program using Macros.
8	Interfacing of 8255 with 8086
9	Assembly language program for string reverse.
10	Case study on Motherboard

I N D E X

Name: **Vivek S Hotti** Roll: **31** Class: **SE-Sem4** Div.: **A**

Subject: **Microprocessor (CSL404)**

EXPERIMENT 01

Write an assembly language program to perform 8-bit addition.

AIM:

Use of programming tools (Debug/TASM/MASM/8086kit) to perform basic arithmetic operations on 8-bit/16-bit data.

THEORY:

Arithmetic Instructions are the instructions which perform basic arithmetic operations such as addition, subtraction and a few more. Unlike in 8085 microprocessors, in 8086 microprocessor the destination operand need not be the accumulator.

8086 microprocessors is a 16-bit microprocessor which was developed by Intel in 1976. It has a 16-bit data bus and a 20-bit address bus. It also has 14 16-bit registers.

We can perform 8-bit addition of two numbers using an 8086 microprocessor. The following instructions are needed:

➔ **data segment**

8086's memory is segmented into four portions each of 64 kb. They are - code segment, data segment, extra segment and stack segment. A data segment of an object file is that portion of memory i.e address space of program which contains initialized static variables. The size of this segment is determined by the size of variables in the program.

➔ **db**

DB assembler directive is used to declare a byte type variable or to store a byte in memory location. A byte consists of 8 bits.
Example:
no db 02h
Here, we declare a variable 'no' of type byte having a value of '02h'. In our program we declare two variables 'no1' and 'no2' of data type 'byte' and values '04h' and '02h' respectively.

➔ **ends**

ENDS assembler directive is used with the name of a logical segment to indicate the end of that logical segment.

Example:

```
data segment
...
data ends
```

➔ **code segment**

8086's memory is segmented into four portions each of 64 kb. They are - code segment, data segment, extra segment and stack segment. The code segment which is also known as text segment / text is a portion of an object file or the corresponding section of the program's virtual address space that contains executable instructions.

➔ **assume**

ASSUME assembler directive is used to tell the assembler the name of the logical segment to be used as the physical segment. It simply assigns a new name for the segment registers.

Example:

```
assume cs: code, ds: data, es: extra, ss: stack
```

Here, the CS register is associated with the name 'code', DS register is associated with 'data', ES register with 'extra' and SS register with 'stack'.

➔ **cs**

It stands for 'code segment register'.

➔ **ds**

It stands for 'data segment register'.

➔ **start**

START is a label that we use to tell the assembler where the execution of the actual program begins. We also need to mark the end of this label with an END directive.

Example:

```
start:
...
end start
```

➔ **mov**

'mov' is a data copy / transfer instruction which is used to transfer data from one register or memory location to another register or memory location. The source can be any register, memory location or immediate

data. The destination can be any register. Although in immediate addressing mode, a segment register cannot be the destination register.

Syntax:

mov destination, source

Example:

mov ax,5000h; Immediate Addressing mode
mov ax,bx; Register Addressing mode
mov ax,[SI]; Register based Indirect Addressing mode
mov ax,[2000h]; Direct Addressing mode

➔ add

'add' is an arithmetic instruction which is used to add immediate data or contents of memory location specified in the instruction or a register to the contents of another register or memory location. The result is stored in the destination operand. Memory to memory addition is not possible which means that both source and destination operands cannot be memory locations. Also, contents of segment registers cannot be added using this instruction.

Syntax:

add destination,source

Example:

add ax, 1000h; Immediate Addressing mode
add ax,bx; Register Addressing mode
add ax,[SI]; Register based Indirect Addressing mode
add ax,[4000h]; Direct Addressing mode

➔ ax

It stands for '16-bit accumulator'.

➔ al

It stands for 'Lower 8-bit accumulator'.

➔ bl

It stands for 'Lower 8-bit base register'.

➔ int 3

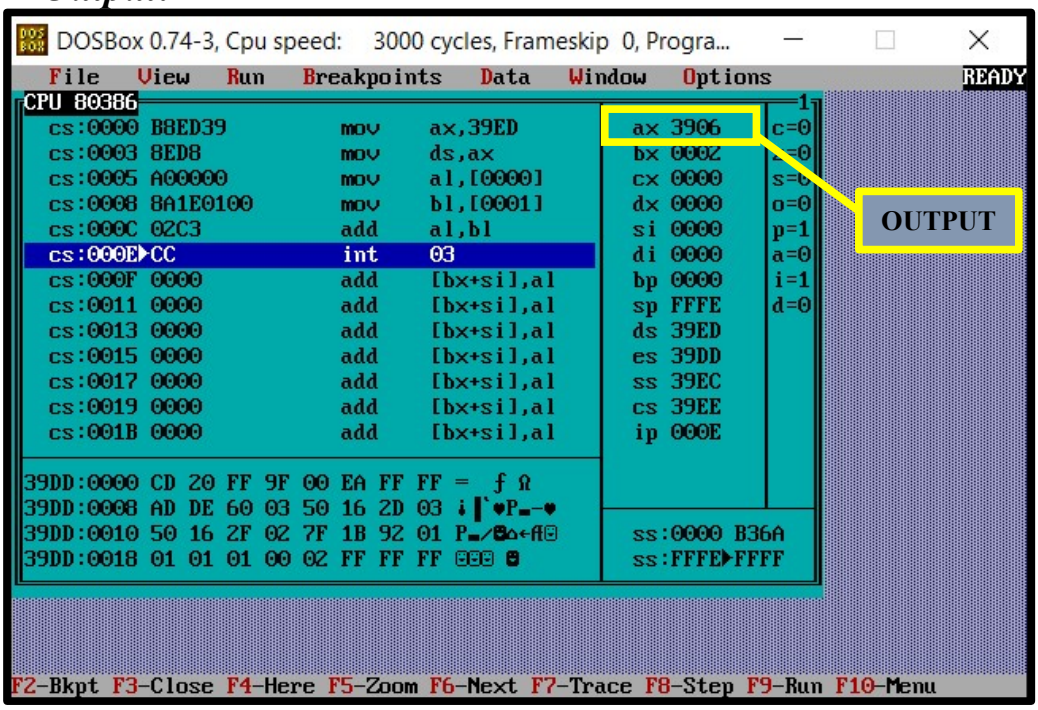
It stands for Interrupt Type 3 which is the breakpoint interrupt. It is used to implement a breakpoint function in the system. When a breakpoint is inserted, the system executes instructions upto breakpoint and then goes to breakpoint procedure. Depending on the system, it either displays register contents or waits for input from the user. Also new CS and IP are loaded.

EXPERIMENTAL RESULTS:

➔ *Code:*

```
data segment
no1 db 04h
no2 db 02h
data ends
code segment
assume cs:code,ds:data
start:mov ax,data
mov ds,ax
mov al,no1
mov bl,no2
add al,bl
int 3
code ends
end start
```

➔ *Output:*



➔ *Explanation:*

According to our code, result is stored in ax register. The inputs are 03h and 02h. We know, **4+2=6**. Result we got in ax is **06. Hence, our result is verified.**

CONCLUSION:

Thus, we have implemented a program that perform basic arithmetic operations on 8-bit/16-bit data.

EXPERIMENT 02

Assembly Program to transfer n block of data to one segment to another segment.

AIM:

To write an Assembly Program to transfer n block of data to one segment to another segment.

THEORY:

The new Instructions learnt in-order to perform this experiment were: -

- **INC:**
  It increments the byte or word by one. The operand can be a register or memory location.

  It affects AF, OF, PF, SF, ZF flags. Whereas, CF is not affected.

  E.g.: INC AX

- **DEC:**
  It decrements the byte or word by one. The operand can be a register or memory location.

  It affects AF, OF, PF, SF, ZF flags. Whereas, CF is not affected.

  E.g.: DEC AX

- **JNE/JNZ:**
  Stands for 'Jump if Not Equal' or 'Jump if Not Zero'. It checks whether the zero flag is reset or not. If yes, then jump takes place.

  If ZF = 0, then jump.

- **LEA:**

  It loads a 16-bit register with the offset address of the data specified by the source.

  E.g.: LEA BX, [DI] This instruction loads the contents of DI (offset) into the BX register.

ALGORITHM:

Step 1: Initialize data segment and store the first block of data. End the data segment.

Step 2: Initialize extra segments and store the second block of data. End the extra segment.

Step 3: Initialize code segment. Assume cs:code, ds:data and es:extra.

Step 4: Move the blocks of data into ds and es with the help of ax. Set counter register cl to 03h.

Step 5: Load effective address of block and block2 into si and di index pointer registers respectively.

Step 6: Transfer element of first block pointed by si into ah and element of second block pointed by di to bh. Transfer content of ah to block2 and content of bh to block1.

Step 7: Decrement counter register cl and increment si and di

Step 8: Repeat Steps VI and VII till ZF = 0.

Step 9: Interrupt Type 3. End of code segment.

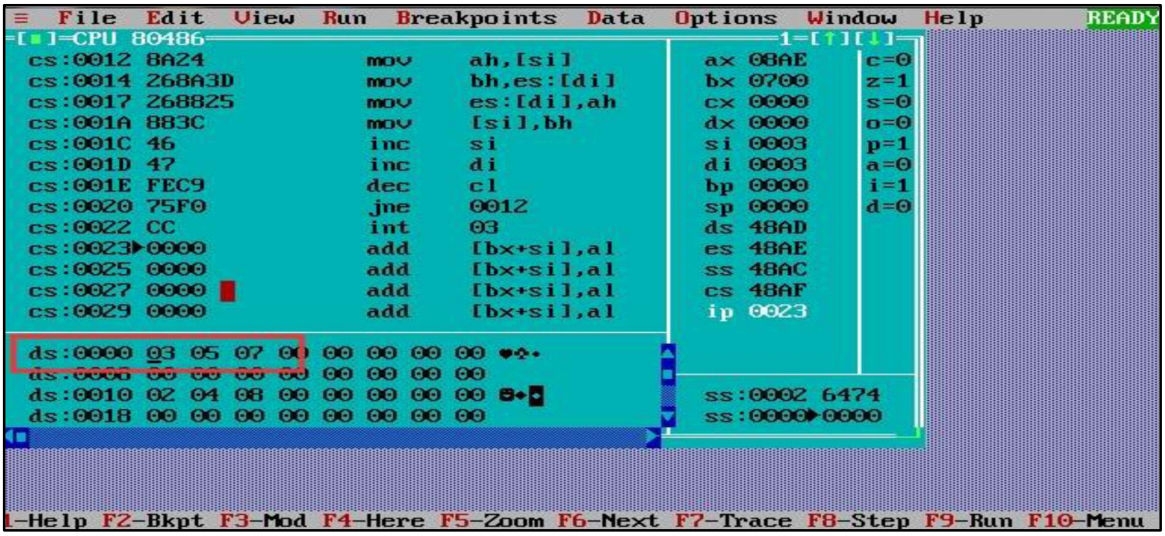
EXPERIMENTAL RESULTS:

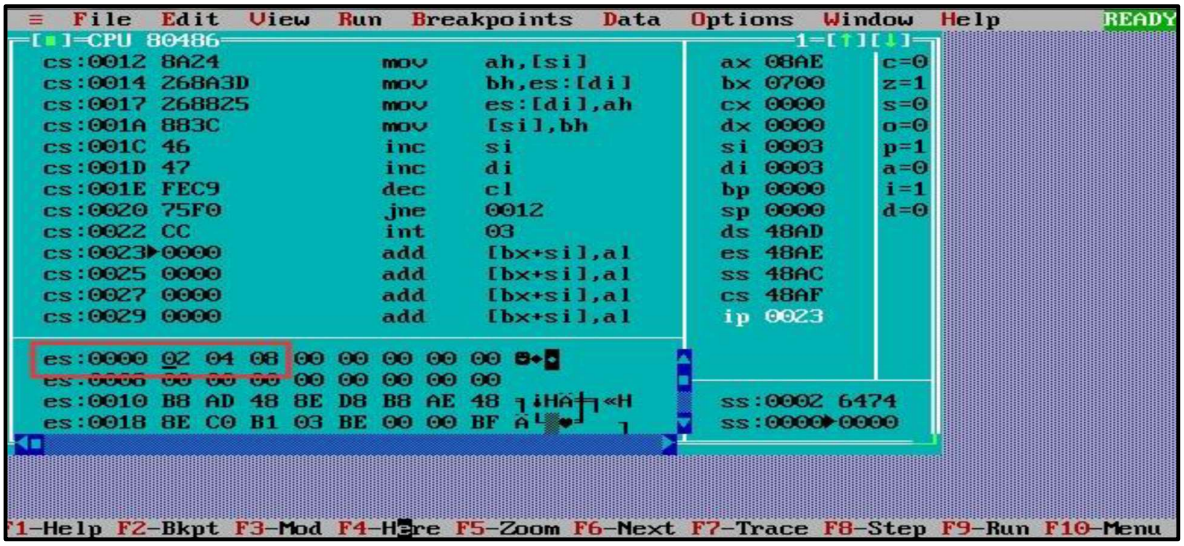
➔ ***Code:***

```
data segment
block1 db 02h,04h,08h
data ends
extra segment
block2 db 03h,05h,07h
extra ends
code segment
assume cs:code,ds:data,es:extra
start:mov ax,data
mov ds,ax
mov ax,extra
mov es,ax
mov cl,03h
lea si,block1
lea di,block2
x:mov ah,ds:[si]
mov bh,es:[di]
```

```
mov es:[di],ah
mov ds:[si],bh
inc si
inc di
dec cl
jnz x
int 3
code ends
end start
```

➔ ***Output:***





➔ ***Explanation:***

From the above output we can say that data between two blocks has been transferred. Data has been transferred from ds register (03h 05h 07h) fo es register (02h 04h 08h).

Hence our results stand verified.

CONCLUSION:

Thus, we have implemented an assembly Program to transfer n block of data to one segment to another segment.

EXPERIMENT 03

Assembly language program for SORTING 8-bit numbers in ascending and descending order.

AIM:

To write an assembly language program for SORTING 8-bit numbers in ascending and descending order.

THEORY:

The new Instructions learnt in-order to perform this experiment were: -

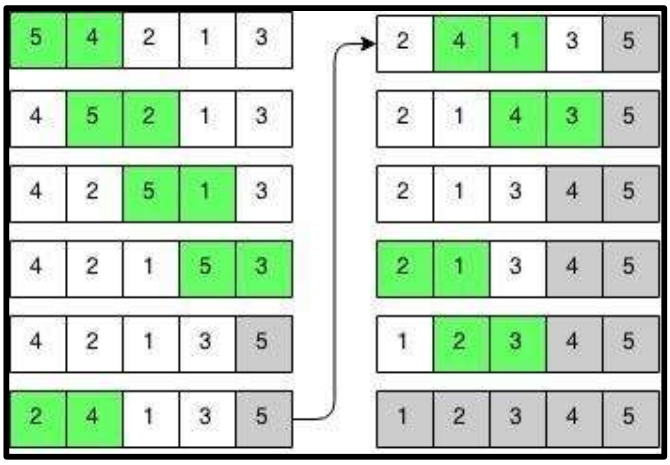
- **CMP:**
  CMP is a mnemonic that stands for "Compare Accumulator" and here R stands for any of the following registers, or memory location M pointed by HL pair.
  This instruction is used to compare contents of the Accumulator with given register R. The result of compare operation will be stored in the Temp register.

- **JC:**
  (Conditional jump) The program sequence is transferred to a particular level or a 16-bit address if C=1 (or carry is 1)
  Eg: - JC ABC (jump to the level abc if C=1)

- **JNC:**
  (Conditional jump) The program sequence is transferred to a particular level or a 16-bit address if C=0 (or carry is 0)
  Eg: JNC ABC (jump to the level abc if C=0)

- **JNZ:**
  (Conditional jump) The program sequence is transferred to a particular level or a 16-bit address if Z=0 (or zero flag is 0)
  Eg: - JNZ ABC (jump to the level abc if Z=0)

- **XCHG:**
  This instruction exchanges Src with Des. It cannot exchange two memory locations directly. E.g.: XCHG DX, AX



ALGORITHM:

A) ASCENDING ORDER

Step 1: Store the series to be sorted into the data segment.

Step 2: Initialize code segment. Assume cs:code and ds:data

Step 3: Move data from data segment to ds with the help of accumulator.

Step 4: Set counter register ch to 04h.

Step 5: Set counter register cl to 04h.

Step 6: Load effective address of the series into St register.

Step 7: Move the element pointed by SI and the consecutive element into al and bl respectively.

Step 8: Compare these elements. If carry is generated go to Step X.

Step 9: Exchange both the elements.

Step 10: Increment SI and decrement cl register

Step 11: Repeat Steps VI to IX till zero flag is not set

Step 12: Decrement ch register.

Step 13: Repeat steps V to XII till zero flag is not set

Step 14: Interrupt Type 3. End of code segment.

B) DESCENDING ORDER

Step 1: Store the series to be sorted into the data segment.

Step 2: Initialize code segment. Assume cs:code and ds:data

Step 3: Move data from data segment to ds with the help of accumulator.

Step 4: Set counter register ch to 04h.

Step 5: Set counter register cl to 04h.

Step 6: Load effective address of the series into St register.

Step 7: Move the element pointed by SI and the consecutive element into al and bl respectively.

Step 8: Compare these elements. If carry is generated go to Step X.

Step 9: Exchange both the elements.

Step 10: Increment SI and decrement cl register

Step 11: Repeat Steps VI to IX till zero flag is not set

Step 12: Decrement ch register.

Step 13: Repeat steps V to XII till zero flag is not set

Step 14: Interrupt Type 3. End of code segment.

EXPERIMENTAL RESULTS:

➔ *A) Code – Ascending Order:*

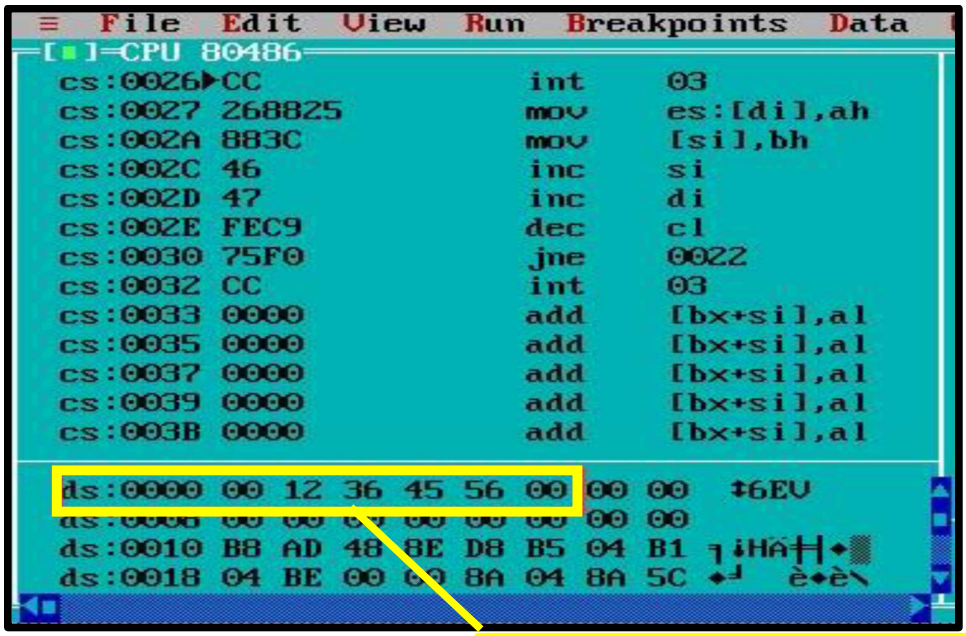
```
series DB 99h,12h,56h,45h,36h
data ends
code segment
assume cs:code,ds:data
start :mov ax,data
mov ds,ax
mov ch,04h
y :mov cl,04h
lea SI,series
x :mov al,[SI]
mov bl,[SI+1]
cmp al,bl
JC next
mov dl,[SI+1]
XCHG [SI],dl
mov [SI+1],dl
next : inc SI
deccl
jnz x
decch
jnz y
int 3
code ends
end start
```

➔ *B) Code – Descending Order:*

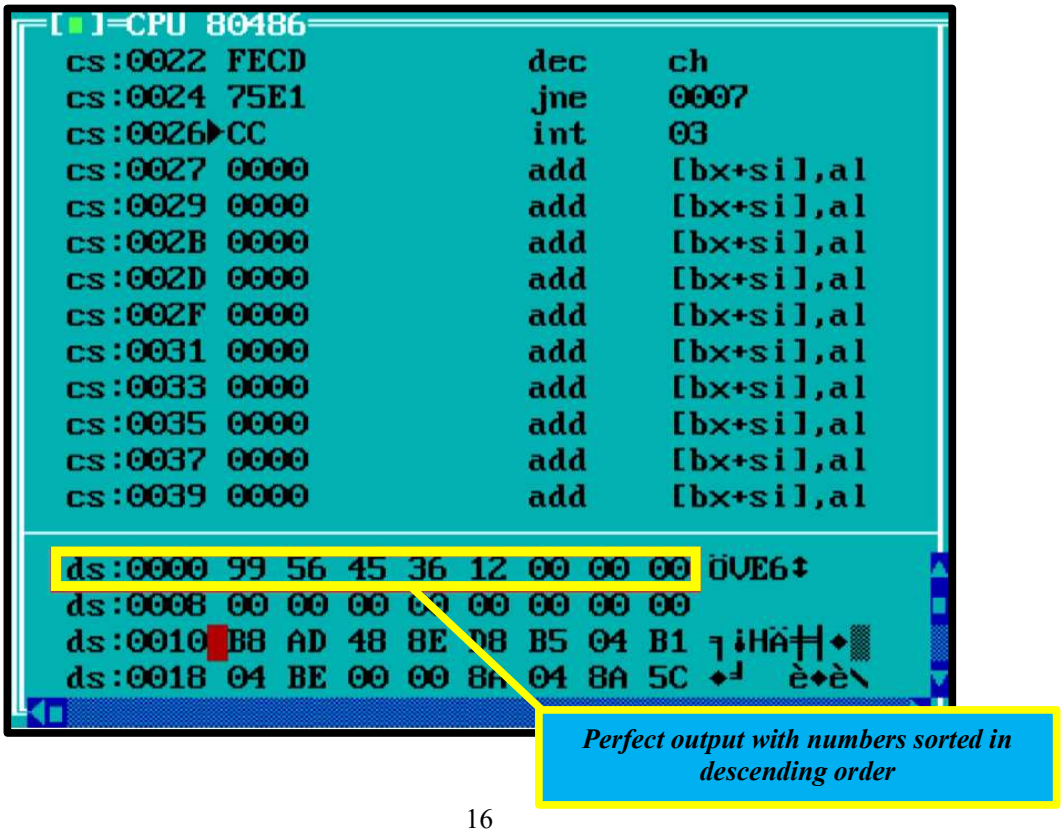
```
data segment
series DB 99h,12h,56h,45h,36h
data ends
code segment
assume cs:code, ds:data
start :mov ax,data
mov ds,ax
mov ch,04h
y :mov cl,04h
lea SI,series
x :mov al,[SI]
mov bl,[SI+1]
cmp al,bl
JNC next
mov dl,[SI+1]
XCHG [SI],dl
mov [SI+1],dl
next : inc SI
dec cl
jnz x
dec ch
jnz y
int 3
code ends
end start
```

➔ ***Output:***

A) ASCENDING ORDER: -



B) DESCENDING ORDER: -



➔ ***Explanation:***

A) Our results are stored in ds and series to be sorted is: 99,12,56,45,36. The series in ascending order should be: 12,36,45,56,99.
And the result in DS is:12,36,45,56,99; Hence result is verified.

B) Our results are stored in ds and series to be sorted is: 99,12,56,45,36. The series in descending order should be: 99,56,,45,36,12. And the result in DS is:99,56,,45,36,12 ; Hence result is verified.

CONCLUSION:

Thus, we have implemented a program to perform Sorting both ascending and descending type using Assembly Language Programming.

EXPERIMENT 04

Write a program To Perform Fibonacci series & Factorial Using Assembly Programming.

AIM:

Write a program To Perform:
A) Fibonacci series Using Assembly Programming
B) Factorial Using Assembly Programming.

THEORY:

The new Instructions learnt in-order to perform this experiment were: -

- **DUP:**

  The DUP operator is very often used in the declaration of arrays This operator works with any of the data allocation directives. the count value sets the number of times to repeat all values within the parentheses.

- **LOOP:**
  This instruction is used to repeat a series of instructions some number of times. The number of times the instruction sequence is to be repeated is loaded into CX.

  Each time the LOOP instruction executes, CX is automatically decremented by 1. If CX is not 0, execution will jump to a destination specified by a label in the instruction. If CX = 0 after the auto decrement, execution will simply go on to the next instruction after LOOP.

  The destination address for the jump must be in the range of –128 bytes to +127 bytes from the address of the instruction after the LOOP instruction. This instruction does not affect any flag.

- **MUL:**

  This instruction multiplies an unsigned byte (word) from some source with an unsigned byte (word) in the AL (AX) register.

The source can be a register or a memory location addressed by any of the 24 standard modes of addressing.

When doing byte-multiplication, the result is stored in a 16-bit location AX since the result can be as large as 16 bits. Word-multiplication results in the most significant word being placed in the DX register and the least significant word placed in the AX register.

ALGORITHM:

A) FIBONACCI SERIES

Step 1: Initialize data segment. Store the desired value (here 05h) and uninitialized array of size 10 using variables numl and res respectively. End data segment.

Step 2: Initialize code segment. Assume es code and ds data. Move data into ds.

Step 3: Store counter 05h into ex register.

Step 4: Load effective address of array res into Sl register: 22/37.

Step 5: Move data 00h and 01h into res array one after the other win the help of al, bl and S1 registers.

Step 6: Add the data pointed by [si-2] and [si-1] with the help of al and bl registers. Store it at current memory location of res pointed by SI register

Step 7: Increment contents of St register.

Step 8: Repeat Steps VI to VII till the counter register becomes zero.

Step 9: Interrupt Type 03. End of code segment.

B) FACTORIAL OF A NUMBER

Step 1: Initialize data segment. Store number whose factorial is to be found (here 05h) into a variable num1. End data segment.

Step 2: Initialize code segment. Assume es code and ds data. Move data into ds.

Step 3: Move 01h into ax register and numl into bi register.

Step 4: Multiply contents of bl with ax register.

Step 5: Decrement bl register.

Step 6: Compare contents of bl with data 01h. Repeat Steps IV and V till zero flag is not set.

Step 7: Interrupt Type 3. End of code segment.

EXPERIMENTAL RESULTS:

➔ *A) Code – Fibonacci Series:*

```
data segment
num1 db 05h
res db 10 dup(?)
data ends
code segment
assume cs:code,ds:data
start: mov ax,data
mov ds,ax
mov cx,05h
lea si,res
mov al,00h
mov bl,01h
mov [si],al
inc si
mov [si],bl
inc si
up: mov al,[si-2]
mov bl,[si-1]
add al,bl
mov [si],al
inc si
loop up
int 3
code ends
end start
```

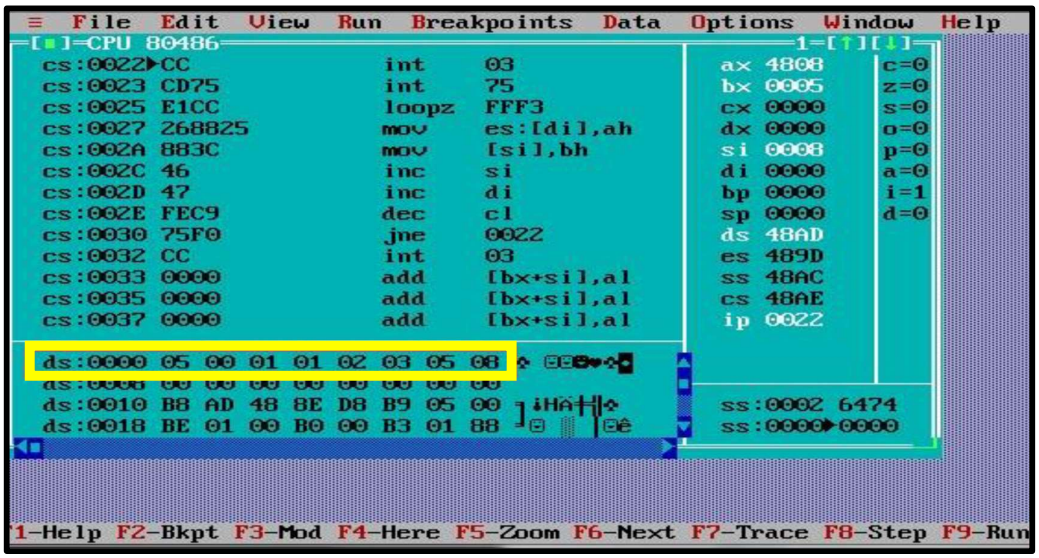
➔ *B) Code – Factorial of a Number:*

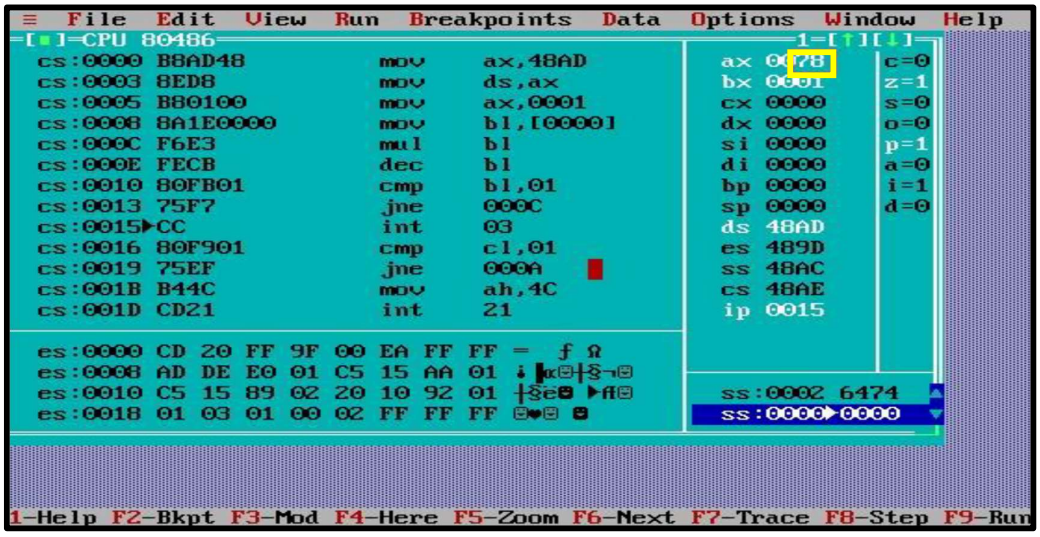
```
data segment
num1 db 05h
data ends
code segment
```

```
assume ds:data,cs:code
start:mov ax,data
mov ds,ax
mov ax,01h
mov bl,num1
y: mul bl
dec bl
cmp bl,01
jnz y
int 3
code ends
end start
```

➔ ***Output:***

A) Fibonacci Series: -





➔ ***Explanation:***

A) Our result will be generated in the res array according to our code. Since our input is 05h we must receive the 1st 7 terms of the fibonacci series i.e.:0,1,1,2,3,5,8. The output we get in ds is: 0,1,1,2,3,5,8. Hence our results are verified.

B) Our result will be generated in ax register as we have performed byte operation. Our input is 05h so factorial of 5 is to be calculated i.e., =120. After execution ax register has data 0078h which hexadecimal and on conversion to decimal is 120, the result we get is 120. Hence our result is verified.

CONCLUSION:

Thus, we have implemented a program that outputs the Fibonacci series & Factorial of a given number using assembly programming.

EXPERIMENT 05

Write a program to implement mixed language programming using Assembly and C

AIM:

Write a program to implement mixed language programming using Assembly and C.

THEORY:

- **Mixed Language Theory:**

  There are times when programs need to call programs written in other languages referred as mixed language programming. For example, when a particular subprogram is available in a language other than language you are using, or when algorithms are described more naturally in a different language, you need to use more than one language.
  Mixed-language programming always involves a call to a function, procedure, or subroutine. Mixed-language calls involve calling functions in separate modules. Instead of compiling all source programs with same compiler, different compilers or assemblers are used as per the language used in the programs.
  Microsoft C supports this mixed language programming. So it can combine assembly code routines in C as a separate language.
  C program calls assembly language routines that are separately assembled by-MASM (MASM Assembler). These assembled modules are linked with the compiled C modules to get executable file. Fig shows the compile, assemble and link processes using C compiler, MASM assembler, and TUNIC.

- **ASM Keyword:**

  The asm keyword allows you to embed assembler instructions within C code. GCC provides two forms of inline asm statements. A basic asm statement is one with no operands (see Basic Asm), while an extended asm statement (see Extended Asm) includes one or more operands. The extended form is preferred for mixing C and assembly language within a

function, but to include assembly language at top level you must use basic asm.

- **Inline assembly:**

  Inline assembly (typically introduced by the asm keyword) gives the ability to embed assembly language source code within a C program. Unlike in C++, inline assembly is treated as an extension in C. It is conditionally supported and implementation defined, meaning that it may not be present and, even when provided by the implementation, it does not have a fixed meaning.
  Syntax
  asm (string_literal) ;

- **how to use assembly + c?**
  We can write assembly program code inside a C language program. In such case, all the assembly code must be placed inside asm{} block

ALGORITHM:

Step 1: Start.

Step 2: Declare three variable a, b and c

Step 3: Input a and b from the user.

Step 4: Move a into ax and b into box.

Step 5: Add ax and bx.

Step 6: Move result from ax to c.

Step 7: Print c.

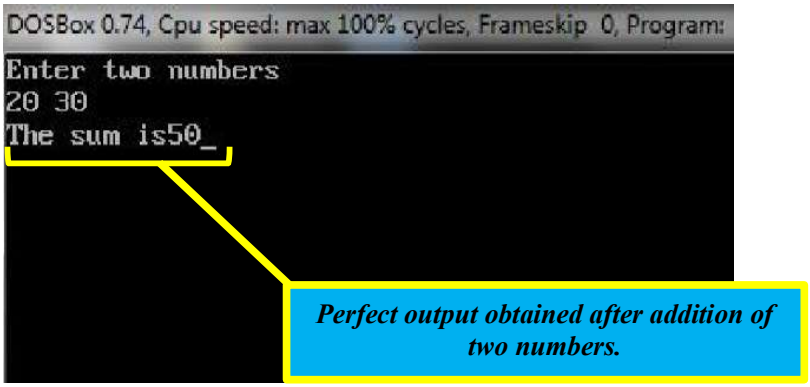
Step 8: Stop.

EXPERIMENTAL RESULTS:

➔ *Code:*

```
#include<iostream.h>
#include<conio.h>
void main()
{
int a,b,c;
clrscr();
cout<<" Enter two numbers\n";
cin>>a>>b;
asm mov ax,a;
asm mov bx,b;
asm add ax,bx;
asm mov c,ax;
cout<<"The sum is"<<c;
getch();
}
```

➔ *Output:*



CONCLUSION:

Thus, we have written a program to implement mixed language programming using Assembly and C.

EXPERIMENT 06

Assembly language program for implementing DOS Interrupt

AIM:

Write a Program to input string from keyboard and display it.

THEORY:

- **DOS Interrupt:**

  DOS and BIOS interrupts are used to perform some very useful functions, such as displaying data to the monitor, reading data from keyboard, etc.
  They are used by identifying the interrupt option type, which is the value stored in register AH and providing, whatever extra information that the specific option requires.
  While the Microprocessor is executing a program, an 'interrupt' breaks the normal sequence of execution of instructions, diverts its execution to some other program called Interrupt Service Routine (ISR). After executing ISR, IRET returns the control back again to the main program. Interrupt processing is an alternative to polling.

- **Use of Interrupt:**

  How to get key typed in the keyboard or a keypad? Polling :- The CPU executes a program that checks for the availability of data If a key is pressed then read the data, otherwise keep waiting or looping.

- **JE / JZ:**

  A conditional jump instruction, like "je" (jump-if-equal), does a goto somewhere if the two values satisfy the right condition. For example, if the values are equal, subtracting them results in zero, so "je" is the same as "jz".

ALGORITHMS:

Step 1: Create a stack of size 100H and an array "STRING" of size 50 with the initial value of each element as '$'.

Step 2: Move offset of "STRING" into Sl register.

Step 3: Compare cach character passed through the keyboard with "q".

Step 4: If result of compare is zero, then go to Step VIIL

Step 5: Move the entered character into the location pointed by the SI register.

Step 6: Increment SI register.

Step 7: Jump to Step III.

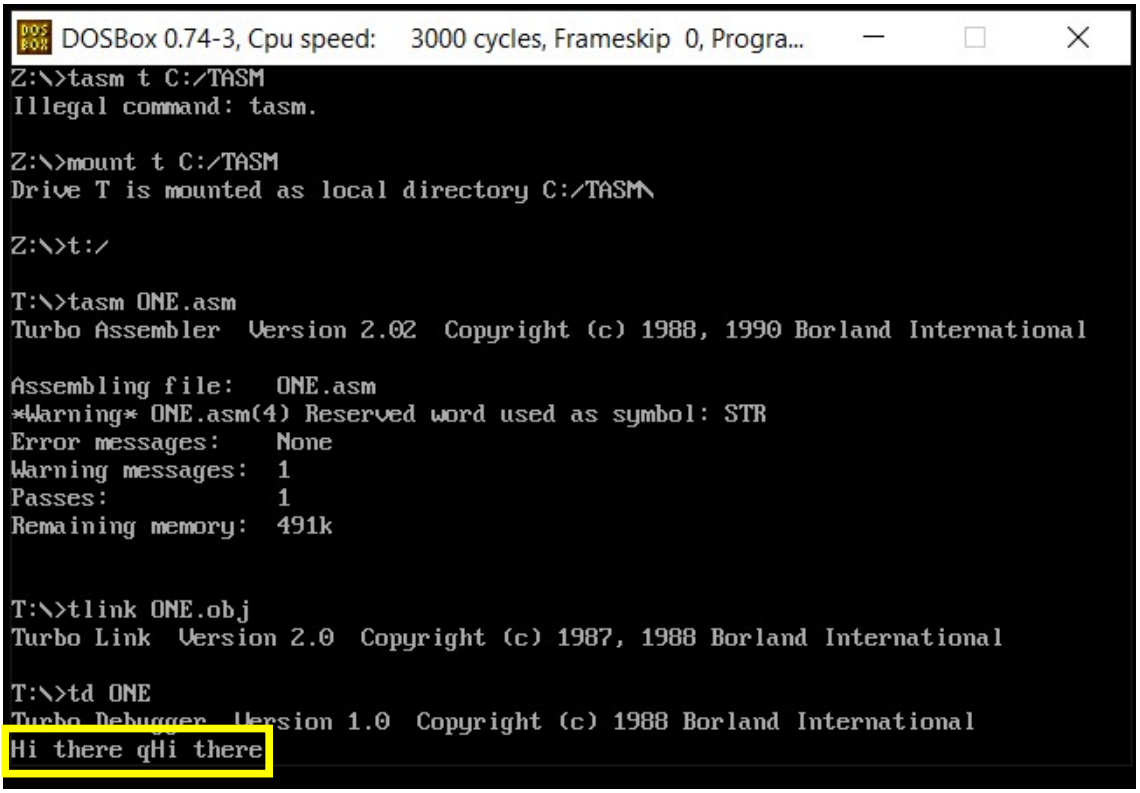
Step 8: Display the entered string.

EXPERIMENTAL RESULTS:

➔ ***Code:***

```
.MODEL SMALL
.STACK 100H
.DATA
STR db 50 DUP ('$')
.CODE
MAIN PROC FAR
MOV AX, @ DATA
MOV DS, AX
MOV SI, OFFSET STR
L2:
MOV AH, 01H
INT 21H
CMP AL, 'q'
JE L1
MOV [SI] , AL
INC SI
JMP L2
L1:
MOV AH, 09H
MOV DX, OFFSET STR
INT 21H
MOV AX, 4C00H
INT 21H
MAIN ENDP
END MAIN
```

➔ ***Output:***



CONCLUSION:

Thus, we have implemented a program to input string from keyboard and display it.

EXPERIMENT 07

Write an Assembly language program for factorial & Addition using macros.

AIM:

Write an Assembly language program for factorial & Addition using macros.

THEORY:

Writing a macro is another way of ensuring modular programming in assembly language.

- A **macro** is a sequence of instructions, assigned by a name and could be used anywhere in the program.
- In NASM, macros are defined with **%macro** and **%endmacro** directives.
- The macro begins with the %macro directive and ends with the %endmacro directive.

The Syntax for macro definition –

%macro macro_name number_of_params
<macro body>
%endmacro

Where, *number_of_params* specifies the number parameters, *macro_name* specifies the name of the macro.

The macro is invoked by using the macro name along with the necessary parameters. When you need to use some sequence of instructions many times in a program, you can put those instructions in a macro and use it instead of writing the instructions all the time.

For example, a very common need for programs is to write a string of characters in the screen. For displaying a string of characters, you need the following sequence of instructions −

```
mov     edx,len         ;message length
mov     ecx,msg         ;message to write
mov     ebx,1           ;file descriptor (stdout)
mov     eax,4           ;system call number (sys_write)
int     0x80            ;call kernel
```

In the above example of displaying a character string, the registers EAX, EBX, ECX and EDX have been used by the INT 80H function call. So, each time you need to display on screen, you need to save these registers on the stack, invoke INT 80H and then restore the original value of the registers from the stack. So, it could be useful to write two macros for saving and restoring data.
We have observed that, some instructions like IMUL, IDIV, INT, etc., need some of the information to be stored in some particular registers and even return values in some specific register(s).

EXPERIMENTAL RESULTS:

A)

➔ *Code:*

```
mov ax,01h
mov bl,05h
y:mul bl
dec bl
cmp bl,01
jnz y
endm
data segment
num db 05h
data ends
code segment
assume
ds:data,cs:code
start:mov ax,data
mov ds,ax
factorial 05h
int 3
code ends
end start
```

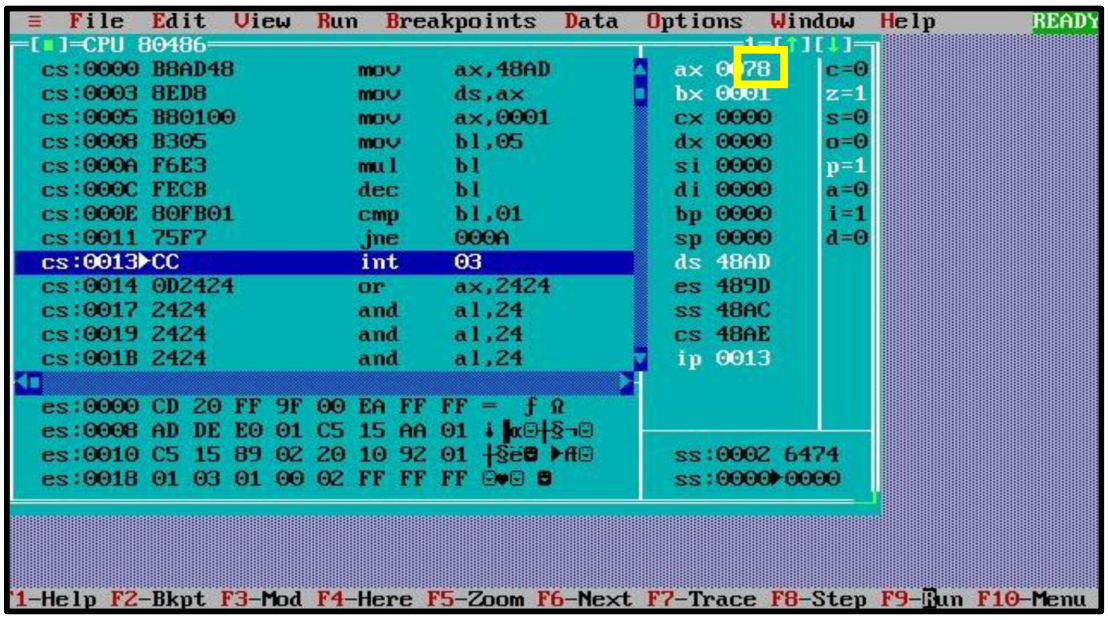
B)

➔ *Code:*

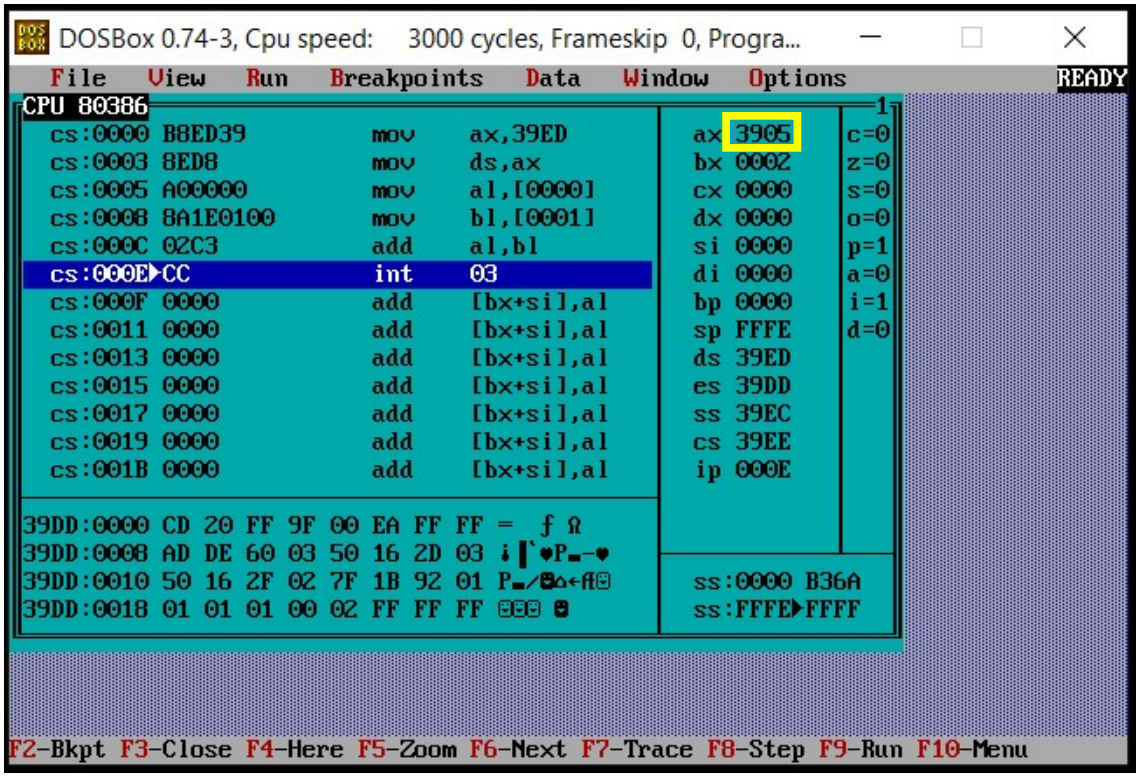
```
addition macro
num1,num2
mov al,num1
mov bl,num2
add al,bl
endm
data segment
num1 db 03h
num2 db 02h
data ends
code segment
assume
cs:code,ds:data
start: mov ax,data
mov ds,ax
addition num1, num2
int 3h
code ends
end start
```

➔ ***Output:***

A)


File Edit View Run Breakpoints Data Options Window Help READY
CPU 80486
cs:0000 B8AD48 mov ax,48AD
cs:0003 8ED8 mov ds,ax
cs:0005 B80100 mov ax,0001
cs:0008 B305 mov bl,05
cs:000A F6E3 mul bl
cs:000C FECB dec bl
cs:000E 80FB01 cmp bl,01
cs:0011 75F7 jne 000A
cs:0013 CC int 03
cs:0014 0D2424 or ax,2424
cs:0017 2424 and al,24
cs:0019 2424 and al,24
cs:001B 2424 and al,24
es:0000 CD 20 FF 9F 00 EA FF FF
es:0008 AD DE E0 01 C5 15 AA 01
es:0010 C5 15 89 02 20 10 92 01
es:0018 01 03 01 00 02 FF FF FF
ax 0078
bx 0001
cx 0000
dx 0000
si 0000
di 0000
bp 0000
sp 0000
ds 48AD
es 489D
ss 48AC
cs 48AE
ip 0013
c=0
z=1
s=0
o=0
p=1
a=0
i=1
d=0
ss:0002 6474
ss:0000 0000
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

B)


DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Progra...
File View Run Breakpoints Data Window Options READY
CPU 80386
cs:0000 B8ED39 mov ax,39ED
cs:0003 8ED8 mov ds,ax
cs:0005 A00000 mov al,[0000]
cs:0008 8A1E0100 mov bl,[0001]
cs:000C 02C3 add al,bl
cs:000E CC int 03
cs:000F 0000 add [bx+si],al
cs:0011 0000 add [bx+si],al
cs:0013 0000 add [bx+si],al
cs:0015 0000 add [bx+si],al
cs:0017 0000 add [bx+si],al
cs:0019 0000 add [bx+si],al
cs:001B 0000 add [bx+si],al
39DD:0000 CD 20 FF 9F 00 EA FF FF
39DD:0008 AD DE 60 03 50 16 2D 03
39DD:0010 50 16 2F 02 7F 1B 92 01
39DD:0018 01 01 01 00 02 FF FF FF
ax 3905
bx 0002
cx 0000
dx 0000
si 0000
di 0000
bp 0000
sp FFFE
ds 39ED
es 39DD
ss 39EC
cs 39EE
ip 000E
c=0
z=0
s=0
o=0
p=1
a=0
i=1
d=0
ss:0000 B36A
ss:FFFE FFFF
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

ALGORITHM:

A) FACTORIAL using MACROS

Step 1: Initialize data segment. Store number whose factorial is to be found in a variable "num". End data segment.

Step 2: Initialize code segment. Assume cs code and ds:data.

Step 3: Move data from data segment into ds.

Step 4: factorial 05h.

Step 5: Move 01h into al and 05h into bl.

Step 6: Multiply contents of bl and ax.

Step 7: Decrement bl.

Step 8: Compare bl with 01h.

Step 9: Repeat Steps VI to VIII until zero flag is not set.

Step 10: Interrupt Type 3.

B) FACTORIAL using MACROS

Step 1: Initialize data segment. Store the numbers to be added into two variables numl and num2. End data segment.

Step 2: Initialize code segment. Assume cs:code and ds:data.

Step 3: Move data from data segment into ds.

Step 4: addition num1, num2.

Step 5: Move num1 and num2 into al and bl respectively.

Step 6: Add the contents of al and bl.

Step 7: Interrupt Type 3.

EXPLANATION:

A) FACTORIAL using MACROS

Our result will be generated in ax register. Our input is 05h so factorial of 5 is to be calculated i.e., =120. After execution ax register has data 0078h which hexadecimal and on conversion to decimal is 120, the result we get is 120. Hence our result is verified.

B) FACTORIAL using MACROS

According to our code, result is stored in ax register. The inputs are 03h and 02h. We know, 3+2=5. Result we got in ax is 05. Hence, our result is verified.

CONCLUSION:

Thus, we have implemented an Assembly language program for factorial & Addition using macros.

EXPERIMENT 08

Interfacing of 8255 with 8086.

AIM:

Write a program to Interface 8255 PPI with 8086.

EXPERIMENTAL RESULTS:

Port A – (as input)

Data transferred from 8255 to 8086

```
MOV AL,90H
MOV DX,01E6
OUT DX,AL
MOV DX,01EO
IN AL,DX
INT 5
```

Port B – (as output)

Data transferred from 8086 to 8255

```
MOV AL,80H
MOV DX,01E6
OUT DX,AL
INT 5
```

➔ ***Output:***

RESULT- (PART A)

DATA BUS		RD	WR	A0	A1	COMMENT
90	L	-	L	1	1	Control word mode 0
DATA	L	-	L	1	-	PORT A- i\P
-	-	-	-	-	-	Displays i\p data on kit. Goes to command mode

RESULT- (PART B)

DATA BUS		RD	WR	A0	A1	COMMENT
80	L	-	L	1	1	Control word mode 0 port B o\p
DATA	L	-	L	1	-	Data entered through keyboard will be displayed on data bus
-	-	-	-	-	-	Data displayed on port A

CONCLUSION:

Hence, we have interfaced 8255 with 8086 as O/I port and I/O port.

EXPERIMENT 09

Assembly language program for string reverse.

AIM:

To write an assembly language program to obtain the reverse of a entered string.

THEORY:

The new Instructions learnt in-order to perform this experiment were: -

- **O2H:**
  It helps in displaying single character by sending the characters in DL to display
  Example:
  mov ah, 02h mov dl. "A
  INT 21H

ALGORITHM:

Step 1: Create an array "strl" of size 10 which is uninitialized.

Step 2: Store appropriate message for display in two variables "msgl" and "msg2" in data segment.

Step 3: Display content of msg1 using 09h function with INT 21h.

Step 4: Set counter register el to 00h.

Step 5: Move offset of entered string "strl" to Sl register.

Step 6: Compare each entered character with "Odh" If the result is equal then goto Step VIII.

Step 7: Load content of al into address pointed by SI and increment SI and cl. Repeat Step VI

Step 8: Print contents of msg2.

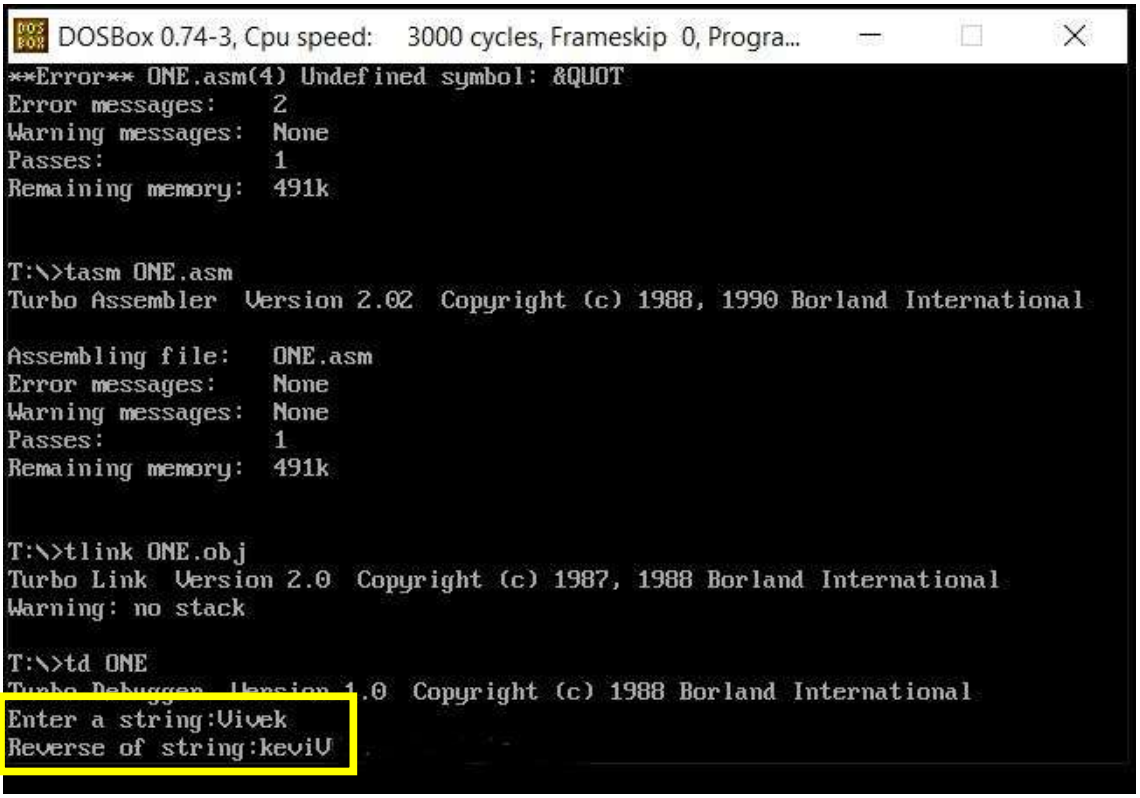
Step 9: Reverse the entered string using appropriate logic and display it.
Step 10: End of code segment.

EXPERIMENTAL RESULTS:

➔ *Code:*

```
data segment
str1 db 10 dup(?)
msg1 db "Enter a string:$"
msg2 db "Reverse of string:$"
data ends
code segment
assume cs:code,ds:data
start:
mov ax, data
mov ds, ax
mov dx, offset msg1;
mov ah, 09
int 21h
mov cl, 00h
mov si, offset str1;
nex: mov ah, 01h
int 21h;
cmp al, 0dh;
je out1
mov [si], al
inc si;
inc cl
jmp nex;
out1: mov dx, offset msg2
mov ah, 09
int 21h;
back: dec si
mov dl,[si]
mov ah, 02;
int 21h
loop back;
mov ax, 4c00h
int 21h
code ends
end start
```

➔ ***Output:***



➔ ***Explanation:***

The input given is "vivek" as the string on termination the string should be reversed i.e. "keviv". We got "keviv" as output, hence our results are verified.

CONCLUSION:

Hence, we have written an assembly language program to reverse a string and implemented it.

EXPERIMENT 10

A Case Study on Motherboard.

AIM:

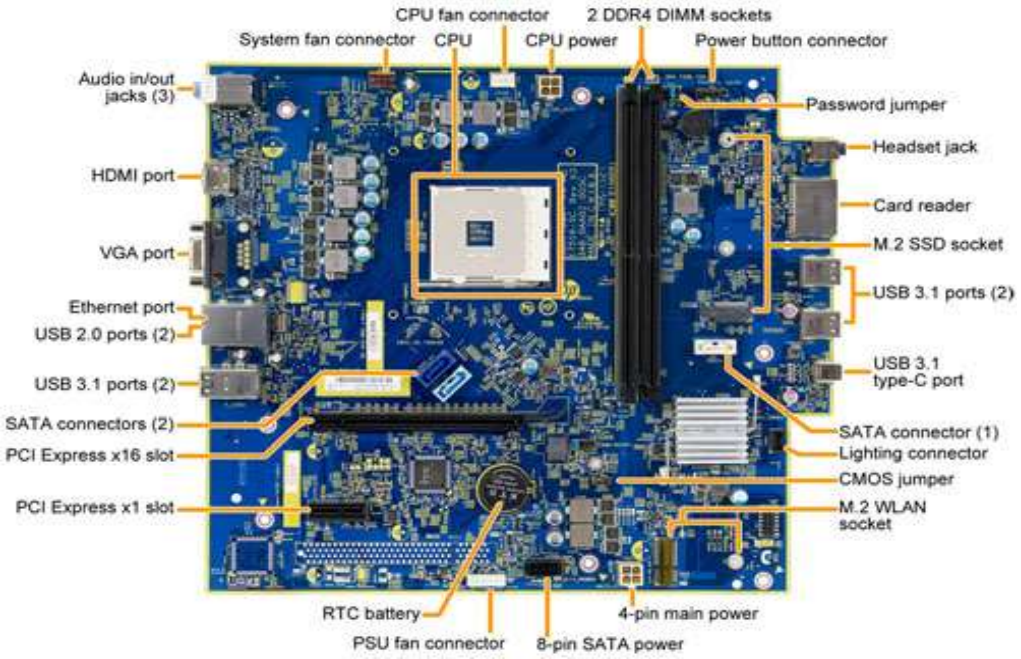
A case study on Motherboard.

THEORY:

- ***MOTHERBOARD:***

A motherboard is the main printed circuit board (PCB) in general-purpose computers and other expandable systems. It holds and allows communication between many of the crucial electronic components of a system, such as the central processing unit (CPU) and memory, and provides connectors for other peripherals. Unlike a backplane, a motherboard usually contains significant sub-systems, such as the central processor, the chipset's input/output and memory controllers, interface connectors, and other components integrated for general use.

Motherboard means specifically a PCB with expansion capabilities. As the name suggests, this board is often referred to as the "mother" of all components attached to it, which often include peripherals, interface cards, and daughtercards: sound cards, video cards, network cards, host bus adapters, TV tuner cards, IEEE 1394 cards; and a variety of other custom components.



Modern motherboards include:

- CPU sockets (or CPU slots) in which one or more microprocessors may be installed. In the case of CPUs in ball grid array packages, such as the VIA Nano and the Goldmont Plus, the CPU is directly soldered to the motherboard
- Memory slots into which the system's main memory is to be installed, typically in the form of DIMM modules containing DRAM chips can be DDR3, DDR4 or DDR5
- The chipset which forms an interface between the CPU, main memory, and peripheral buses
- Non-volatile memory chips (usually Flash ROM in modern motherboards) containing the system's firmware or BIOS
- The clock generator which produces the system clock signal to synchronize the various components
- Slots for expansion cards (the interface to the system via the buses supported by the chipset)
- Power connectors, which receive electrical power from the computer power supply and distribute it to the CPU, chipset, main memory, and expansion cards. As of 2007, some graphics cards (e.g. GeForce 8 and Radeon R600) require more power than the motherboard can provide, and thus dedicated connectors have been introduced to attach them directly to the power supply
- Connectors for hard disk drives, optical disc drives, or solid-state drives, typically SATA and NVMe now.

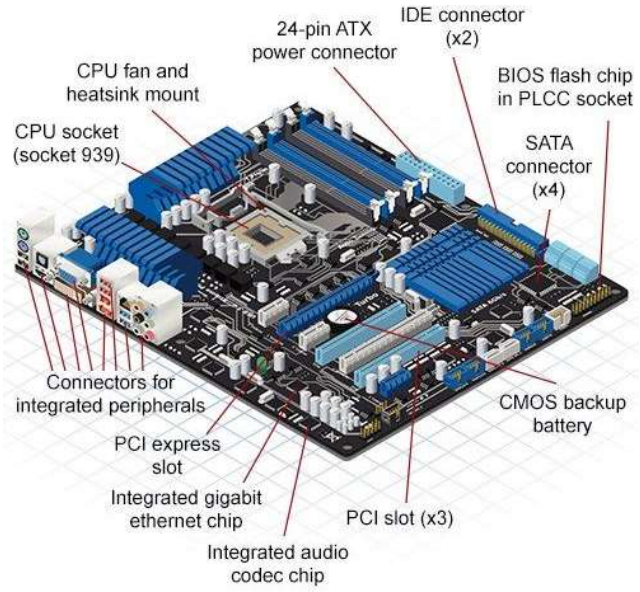
Additionally, nearly all motherboards include logic and connectors to support commonly used input devices, such as USB for mouse devices and keyboards. Early personal computers such as the Apple II or IBM PC included only this minimal peripheral support on the motherboard. Occasionally video interface hardware was also integrated into the motherboard; for example, on the Apple II and rarely on IBM-compatible computers such as the IBM PC Jr. Additional peripherals such as disk controllers and serial ports were provided as expansion cards.

Given the high thermal design power of high-speed computer CPUs and components, modern motherboards nearly always include heat sinks and mounting points for fans to dissipate excess heat.

Integrated peripherals

With the steadily declining costs and size of integrated circuits, it is now possible to include support for many peripherals on the motherboard. By combining many functions on one PCB, the physical size and total cost of the system may be reduced; highly integrated motherboards are thus especially popular in small form factor and budget computers.

- Disk controllers for SATA drives, and historical PATA drives.
- Historical floppy-disk controller
- Integrated graphics controller supporting 2D and 3D graphics, with VGA, DVI, HDMI, DisplayPort and TV output
- integrated sound card supporting 8-channel (7.1) audio and S/PDIF output
- Ethernet network controller for connection to a LAN and to receive Internet
- USB controller
- Wireless network interface controller
- Bluetooth controller
- Temperature, voltage, and fan-speed sensors that allow software to monitor the health of computer components.



CONCLUSION:

Thus, we have explored the motherboard: its different components and their functions.