Vivekanand Education Society's Institute of Technology

Department of Computer Engineering

CPC501 – Microprocessor

REFERENCE LAB MANUAL

T.E.(CMPN) /SEM V/D12

ACADEMIC YEAR: 2017-2018

Vivekanand Education Society's Institute of Technology

Department of Computer Engineering

Name of the Course	**:**	**MICROPROCESSOR(MP) ,**	**Subject code: CPC501**
Year/ Semester / Class	**:**	**T.E. (Comp) / Sem V/ D12 A/B/C**	
Subject and/or Practical Teacher	**:**	**Dr. Gresha Bhatia, Mr. Prashant Kanade, Mrs. Poonam Gholap, Mrs. Sayali Salkade, Ms. Mukesh Yadav**	

LAB PLAN

Sr No	Lab Title	Course Outcomes (CO)
1	Study of Assembler	
2	Write a program to calculate Addition of two 8 and 16 bit number	
	Write a program to calculate Subtraction of two 8 and 16 bit number	
3	Write a program to calculate Multiplication of 8 by 8 bit and 16 by 16 bit number	
	Write a program to calculate Division of 16 by 8 bit and 32 by 16 bit number	
4	Write a program for finding largest and smallest number in an array	
5	Write a program to find factorial of a given number using mixed language	
6	Write a program to move a block of data from one memory location to another using string	
7	Write a program to count no of vowels in given string	
8	Write a program to compare two strings using MACRO and comparing 2 strings	
9	Write a program to display string with int 21h	
10	Write a program to check string is palindrome	
11	Interfacing of 8086 with Programmable Peripheral Interface 8255	
12	Perform interfacing of DAC	

13	Case study of RISC	
14	Case study of CISC	
1	Overall grade	
2	Assignment 1	
3	Assignment 2	
4	Test 1	
5	Test 2	

Experiment: 1

Aim: Study of Assembler

Objective: Details of TASM

Theory:
In general, programming of microprocessor usually takes several iterations before the right sequence of machine code instruction is written. The process, however is facilitated using a special program called an Assembler. The Assembler allows the user to write alphanumeric instructions. The Assembler in turn generates the desired instructions from the assembly language instructions. Assembly level programs generally abbreviated as ALP are written in text editor EDIT.
Assembly language programming consists of following steps:

1. **Editing**: produces source file
2. **Assembling**: produces Object file
3. **Linking**: produces Executable file
4. **Executing**: produces Results

Assembling the program
The assembler is used to convert the assembly language instructions to machine code. It is used immediately after writing the Assembly language program. The assembler starts by checking the syntax or validity of the structure of each instruction in the source file. If any errors are found, the assembler displays a report on these errors along with brief explanation of their nature. However if the program does not contain any error, the assembler produces an object file that has name as the original file but with the "obj" extension.

Linking the program
The linker is used to convert the object file to an executable file. The executable file is the set of machine code instructions that can directly be executed by the microprocessor. It is different than the object file in the sense that it is self-contained and re-locatable. An object file may represent one segment of a long program. This segment cannot operate by itself, and be integrated with object files representing the rest of the program in order to produce final self-contained executable file.

Executing the program
The executable contains the machine language code. It can be loaded in the RAM and executed by the microprocessor simply by typing, from the DOS prompt, the name of the file followed by carriage return key (Enter Key). If the program produces an output on the screen or sequence of control signals to control a piece of hardware, the effect should be noticed almost immediately. However, the program manipulates data in memory; nothing would seem to have happened as a result of executing the program.

Basic Procedure

1. To open Tasm: Double click on TASM1.4 icon on windows desktop. Or go to windows, in search type "tasm1.4", we will see tasm1.4 is installed on the system and click enter. Icon looks like shown below.

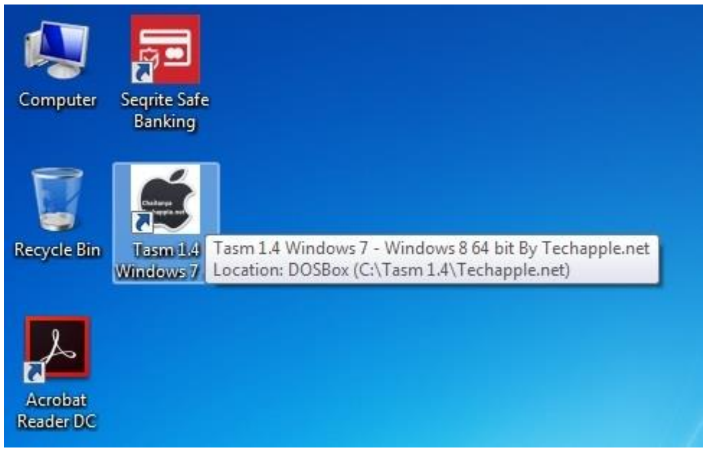
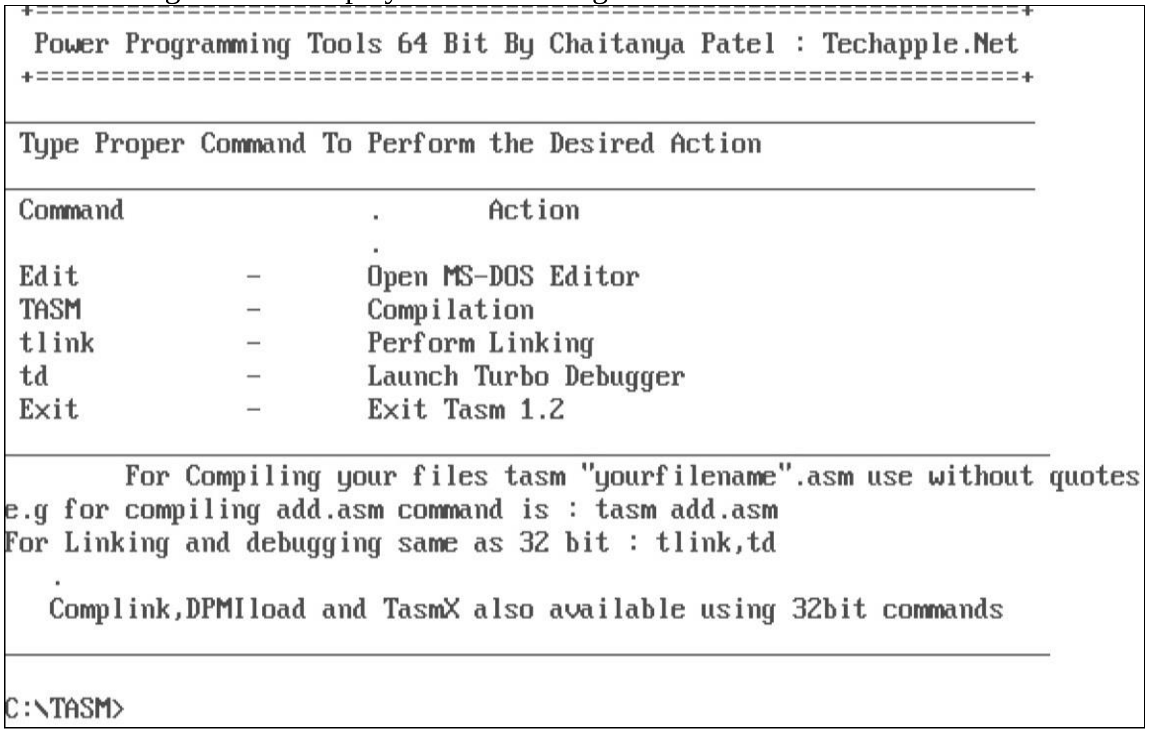

Computer
Seqrite Safe Banking
Recycle Bin
Tasm 1.4 Windows 7
Tasm 1.4 Windows 7 - Windows 8 64 bit By Techapple.net
Location: DOSBox (C:\Tasm 1.4\Techapple.net)
Acrobat Reader DC

Figure 1: Tasm1.4 windows 8 64 bit by Techapple.net

2. The following screen is displayed after clicking on TASM1.4 icon



```
+===================================================================+
 Power Programming Tools 64 Bit By Chaitanya Patel : Techapple.Net
+===================================================================+

Type Proper Command To Perform the Desired Action

Command                .        Action
                       .
Edit          -        Open MS-DOS Editor
TASM          -        Compilation
tlink         -        Perform Linking
td            -        Launch Turbo Debugger
Exit          -        Exit Tasm 1.2

       For Compiling your files tasm "yourfilename".asm use without quotes
e.g for compiling add.asm command is : tasm add.asm
For Linking and debugging same as 32 bit : tlink,td
   .
   Complink,DPMIload and TasmX also available using 32bit commands

C:\TASM>
```

Figure 2: Tasm1.4 start window screen

3. Type "***edit***" to open a blank text editor. Depending on the algorithm & the instructions in the pre-experiment exercise, write the program using Assembly language, save the program with an ***.ASM*** extensionusing***SAVE AS***option. You can also use the command "***edit filename.asm***" to open a blank text editor with "***filename***" as its name.

4. After saving the program use ***Exit*** in the ***File*** option to come back to the MS-DOS Editor as shown in step 2.

```
C:\TASM>edit
```

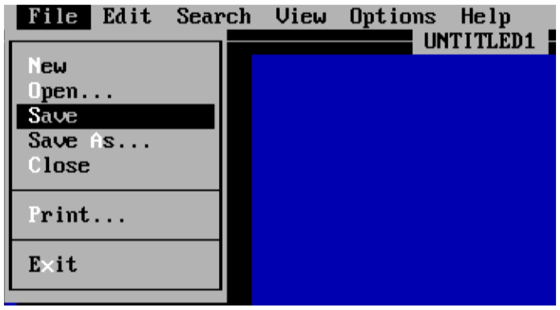
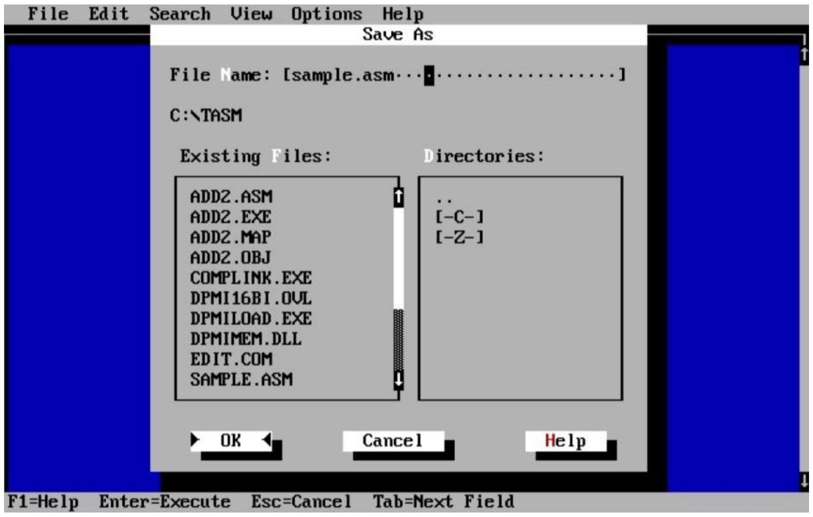


Figure 3: File menu to save the file



```
File Name: [sample.asm·················]

C:\TASM

Existing Files:          Directories:
ADD2.ASM                 ..
ADD2.EXE                 [-C-]
ADD2.MAP                 [-Z-]
ADD2.OBJ
COMPLINK.EXE
DPMI16BI.OVL
DPMILOAD.EXE
DPMIMEM.DLL
EDIT.COM
SAMPLE.ASM

F1=Help  Enter=Execute  Esc=Cancel  Tab=Next Field
```

Figure 4: Saving file name

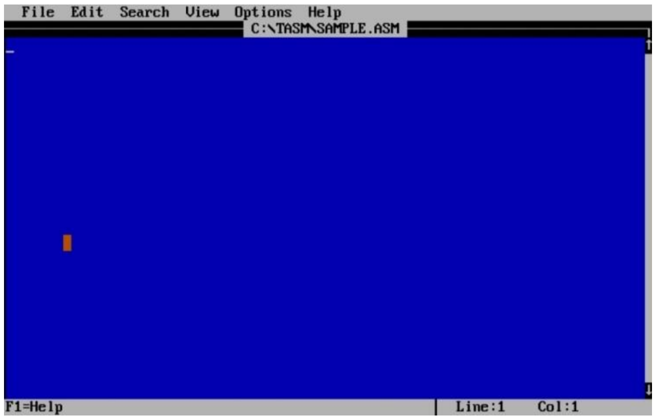


Figure 5: File saved

5. After saving the program use ***Exit*** in the ***File*** option to come back to the MS-DOS Editor.

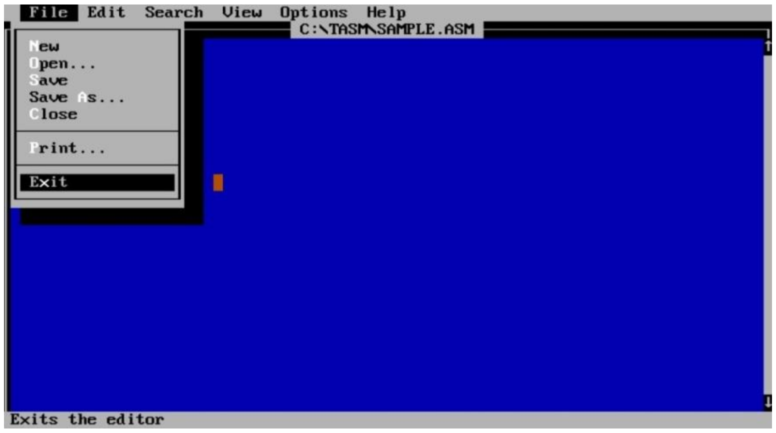
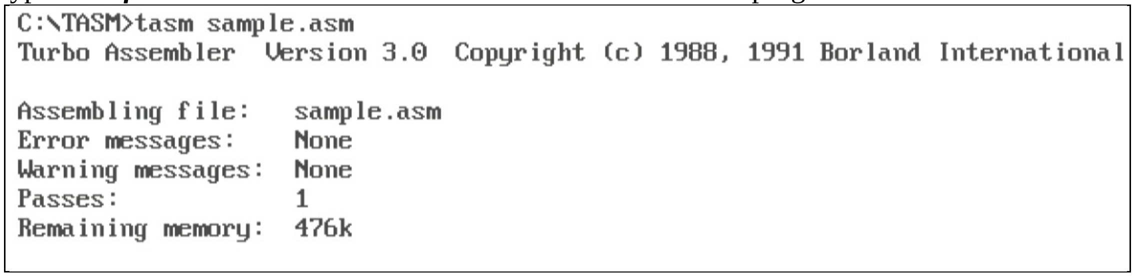


Figure 6: Exit from editor

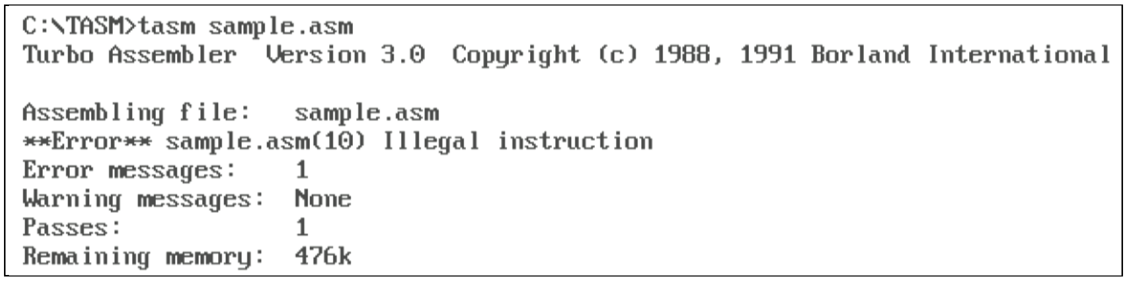
6. Type ***tasm filename.asm*** at the MS-DOS Editor to assemble the program and check for errors.



```
C:\TASM>tasm sample.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland International

Assembling file:   sample.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  476k
```

Figure 7: Assembling the file (having no error)

If the program has no syntax errors, the window shown in figure 7 should appear else the window in figure 8 appears. If the program is free from all syntactical errors, this command will give the OBJECT file. In case of errors it list out the number of errors, warnings and kind of error. Note: No object file is created until all errors are rectified.

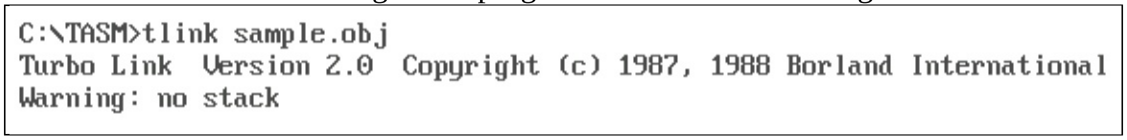


```
C:\TASM>tasm sample.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland International

Assembling file:   sample.asm
**Error** sample.asm(10) Illegal instruction
Error messages:    1
Warning messages:  None
Passes:            1
Remaining memory:  476k
```

Figure 8: Assembling the File (having error)

Note: The error message will vary depending on the error in the program.

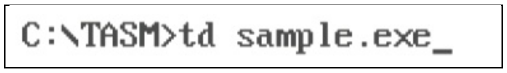
7. Create an object file by using the command ***tlink filename.obj*** or ***tlink filename***
   After successful assembling of the program we have to link it to get Executable file.



```
C:\TASM>tlink sample.obj
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
Warning: no stack
```

Figure 9: File Linking

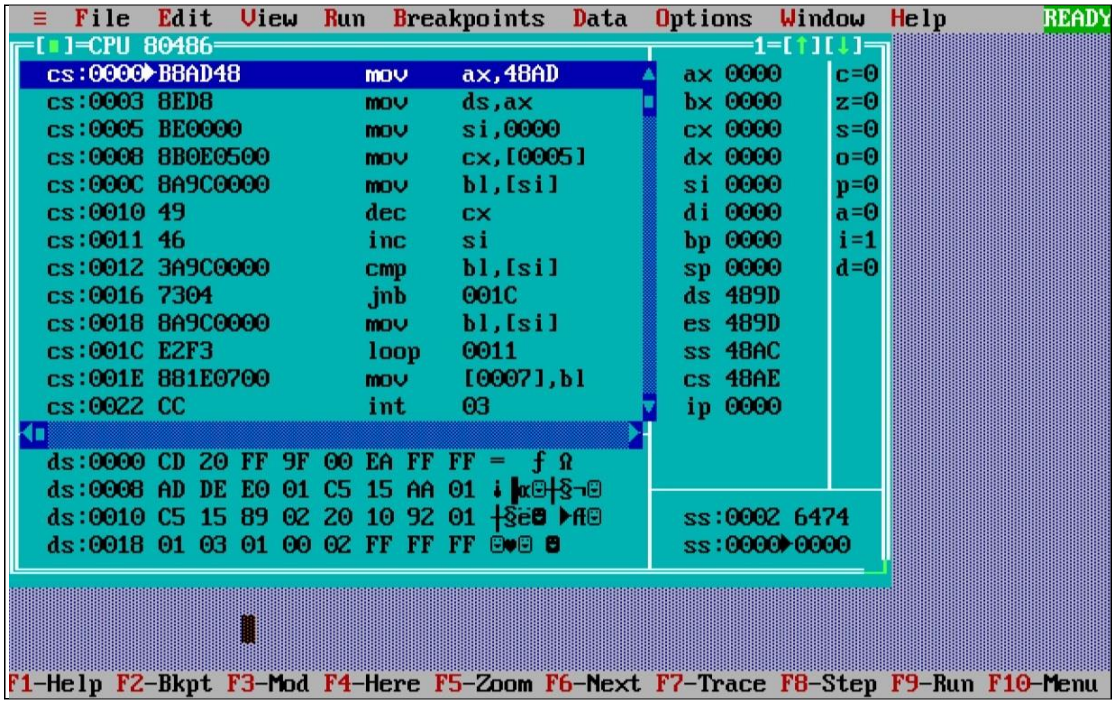
8. Execute the file by typing the command ***td filename.exe.*** The output window is shown in figure 11.



```
C:\TASM>td sample.exe_
```

Figure 10: Debugging the file

This will open the program in debugger screen where in you can view the assemble code with the CS and IP values at the left most side and the machine code. Register content, memory content also be viewed using VIEW option of the debugger. Execute option in the menu in the menu can be used to execute the program either in single steps(F7), Next step (F8) or burst execution(F9).



≡ File Edit View Run Breakpoints Data Options Window Help READY
[■]=CPU 80486
1=[↑][↓]

```
cs:0000▶B8AD48          mov     ax,48AD        ax 0000   c=0
cs:0003 8ED8            mov     ds,ax          bx 0000   z=0
cs:0005 BE0000          mov     si,0000        cx 0000   s=0
cs:0008 8B0E0500        mov     cx,[0005]      dx 0000   o=0
cs:000C 8A9C0000        mov     bl,[si]        si 0000   p=0
cs:0010 49              dec     cx             di 0000   a=0
cs:0011 46              inc     si             bp 0000   i=1
cs:0012 3A9C0000        cmp     bl,[si]        sp 0000   d=0
cs:0016 7304            jnb     001C           ds 489D
cs:0018 8A9C0000        mov     bl,[si]        es 489D
cs:001C E2F3            loop    0011           ss 48AC
cs:001E 881E0700        mov     [0007],bl      cs 48AE
cs:0022 CC              int     03             ip 0000

ds:0000 CD 20 FF 9F 00 EA FF FF = ƒ Ω
ds:0008 AD DE E0 01 C5 15 AA 01 ¡ ▐α☺┼§¬☺      ss:0002 6474
ds:0010 C5 15 89 02 20 10 92 01 ┼§ë☻ ►Æ☺      ss:0000▶0000
ds:0018 01 03 01 00 02 FF FF FF ☺♥☺ ☻
```

F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

Figure 11: Output window in the Editor

Go through the various options on the menu bar to run the program step by step and note the changes in the various registers and flags.

Experiment: 2

Aim: Write a program to calculate
1. Addition of two 8 and 16 bit number
2. Subtraction of two 8 and 16 bit number

Objective: Addition and subtraction of two 8 bit and two 16 bit numbers Using Tasm.

Description: Arithmetic operations Addition and Subtraction

Theory:

SEGMENT

The SEGMENT directive is used to indicate the start of a logical segment. Preceding the SEGMENT directive is the name you want to give the segment. For example, the statement CODE SEGMENT indicates to the assembler the start of a logical segment called CODE. The SEGMENT and ENDS directive are used to "bracket" a logical segment containing code of data.

Additional terms are often added to a SEGMENT directive statement to indicate some special way in which we want the assembler to treat the segment. The statement CODE SEGMENT WORD tells the assembler that we want the content of this segment located on the next available word (even address) when segments ate combined and given absolute addresses. Without this WORD addition, the segment will be located on the next available paragraph (16-byte) address, which might waste as much as 15 bytes of memory. The statement CODE SEGMENT PUBLIC tells the assembler that the segment may be put together with other segments named CODE from other assembly modules when the modules are linked together.

ENDS (END SEGMENT)

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

➢ CODE SEGMENT	Start of logical segment containing code instruction statements
➢ CODE ENDS	End of segment named CODE

ASSUME

The ASSUME directive is used tell the assembler the name of the logical segment it should use for a specified segment. The statement ASSUME CS: CODE, for example, tells the assembler that the instructions for a program are in a logical segment named CODE. The statement ASSUME DS: DATA tells the assembler that for any program instruction, which refers to the data segment, it should use the logical segment called DATA.

DB (DEFINE BYTE)

The DB directive is used to declare a byte type variable, or a set aside one or more storage locations of type byte in memory.

- ➢ PRICES DB 49H, 98H, 29H
  Description: Declare array of 3 bytes named PRICE and initialize them with specified values
- ➢ NAMES DB "THOMAS"
  Description: array of 6 bytes and initialize with ASCII codes for the letters in THOMAS.

- TEMP DB 100 DUP (?)
  Description: Set aside 100 bytes of storage in memory and give it the name TEMP. But leave the 100 bytes un-initialized.
- PRESSURE DB 20H DUP (0)
  Description: Set aside 20H bytes of storage in memory, give it the name PRESSURE and put 0 in all 20H locations.

DW (DEFINE WORD)
The DW directive is used to tell the assembler to define a variable of type word or to reserve storage locations of type word in memory. The statement MULTIPLIER DW 437AH, for example, declares a variable of type word named MULTIPLIER, and initialized with the value 437AH when the program is loaded into memory to be run.

- WORDS DW 1234H, 3456H
  Description: Declare an array of 2 words and initialize them with the specified values.
- STORAGE DW 100 DUP (0)
  Description: Reserve an array of 100 words of memory and initialize all 100 words with 0000. Array is named as STORAGE.
- STORAGE DW 100 DUP (?)
  Description: Reserve 100 word of storage in memory and give it the name STORAGE, but leave the words un-initialized.

DD (DEFINE DOUBLE WORD)
The DD directive is used to declare a variable of type double word or to reserve memory locations, which can be accessed as type double word. The statement ARRAY DD 25629261H, for example, will define a double word named ARRAY and initialize the double word with the specified value when the program is loaded into memory to be run. The low word, 9261H, will be put in memory at a lower address than the high word.

Data Transfer Instructions-MOV, XCHG, LEA, LDS, LES

MOV – MOV Destination, Source
The MOV instruction copies a word or byte of data from a specified source to a specified destination. The destination can be a register or a memory location. The source can be a register, a memory location or an immediate number. The source and destination cannot both be memory locations. They must both be of the same type (bytes or words). MOV instruction does not affect any flag.

MOV CX, 037AH	Put immediate number 037AH to CX
MOV BL, [437AH]	Copy byte in DS at offset 437AH to BL
MOV AX, BX	Copy content of register BX to AX
MOV DL, [BX]	Copy byte from memory at [BX] to DL
MOV DS, BX	Copy word from BX to DS register

- MOV RESULT [BP], AX
  Description: Copy AX to two memory locations; AL to the first location, AH to the second; EA of the first memory location is sum of the displacement represented by

RESULTS and content of BP. Physical address = EA + SS

- MOV ES: RESULTS [BP], AX
  Description: Same as the above instruction, but physical address = EA + ES, because of the segment override prefix ES

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
➢ XCHG AL, PRICES [BX]	Exchange byte in AL with byte in memory at EA=PRICE[BX] in DS.

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

➢ LEA BX, PRICES	Load BX with offset of PRICE in DS
➢ LEA BP, SS: STACK_TOP	Load BP with offset of STACK_TOP in SS
➢ LEA CX, [BX][DI]	Load CX with EA = [BX] + [DI]

LDS – LDS Register, Memory address of the first word
This instruction loads new values into the specified register and into the DS register from four successive memory locations. The word from two memory locations is copied into the specified register and the word from the next two memory locations is copied into the DS registers. LDS does not affect any flag.

- LDS BX, [4326]
  Meaning: Copy content of memory at displacement 4326H in DS to BL, content of 4327H to BH. Copy content at displacement of 4328H and 4329H in DS to DS register.
- LDS SI, SPTR
  Meaning: Copy content of memory at displacement SPTR and SPTR + 1 in DS to SI register. Copy content of memory at displacements SPTR + 2 and SPTR + 3 in DS to DS register. DS: SI now points at start of the desired string.

LES – LES Register, Memory address of the first word
This instruction loads new values into the specified register and into the ES register from four successive memory locations. The word from the first two memory locations is copied into the specified register, and the word from the next two memory locations is copied into the ES register. LES does not affect any flag.

- LES BX, [789AH]
  Meaning: Copy content of memory at displacement 789AH in DS to BL, content of 789BH to BH, content of memory at displacement 789CH and 789DH in DS is copied to ES register.
- LES DI, [BX]
  Meaning: Copy content of memory at offset [BX] and offset [BX] + 1 in DS to DI

register. Copy content of memory at offset [BX] + 2 and [BX] + 3 to ES register.

Arithmetic Instructions-ADD, ADC, SUB, SBB, MUL, IMUL, DIV, IDIV, INC, DEC, DAA,DAS, CBW, CWD, AAA, AAS, AAM, AAD

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
- 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
- ADC AL, PRICES [BX]
  Add byte from effective address PRICES [BX] plus carry status to content of AL
- ADD AL, PRICES [BX]
  Add content of memory at effective address PRICES [BX] to AL

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.

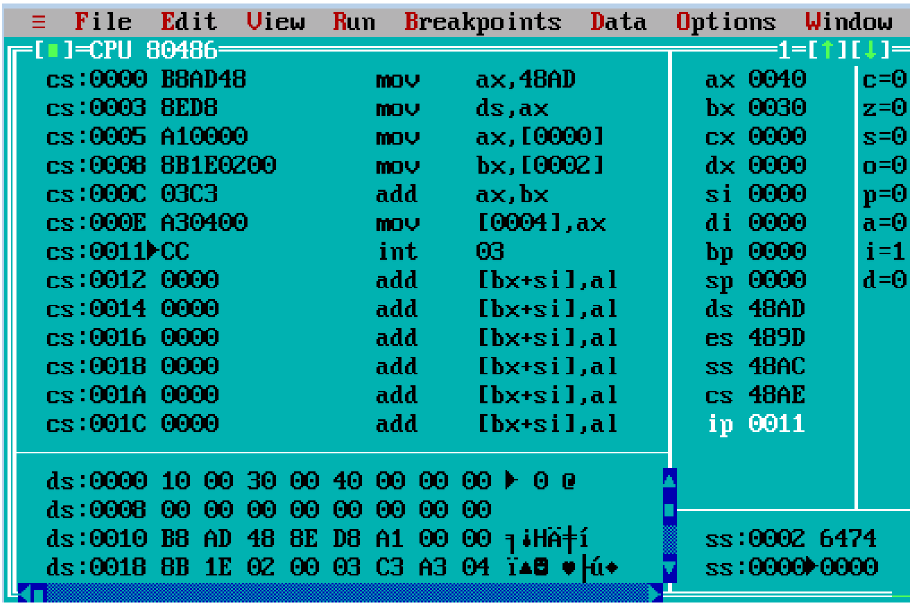
- SUB CX, BX — CX – BX; Result in CX
- SBB CH, AL
  Subtract content of AL and content of CF from content of CH. Result in CH
- SUB AX, 3427H — Subtract immediate number 3427H from AX
- SBB BX, [3427H]
  Subtract word at displacement 3427H in DS and content of CF from BX
- SUB PRICES [BX], 04H
  Description: 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]
  Description: 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].

cs: Address Opcode Operation
ds: Address LvalueHvalueLvalueHvalue..

ADDITION OF TWO 8-Bit NUMBERS
Algorithm
1. Initialize memory pointer to data location.
2. Get the first 8-bit number from memory in accumulator.
3. Get the second 8-bit number and add it to the accumulator.
4. Store the answer at another memory location.

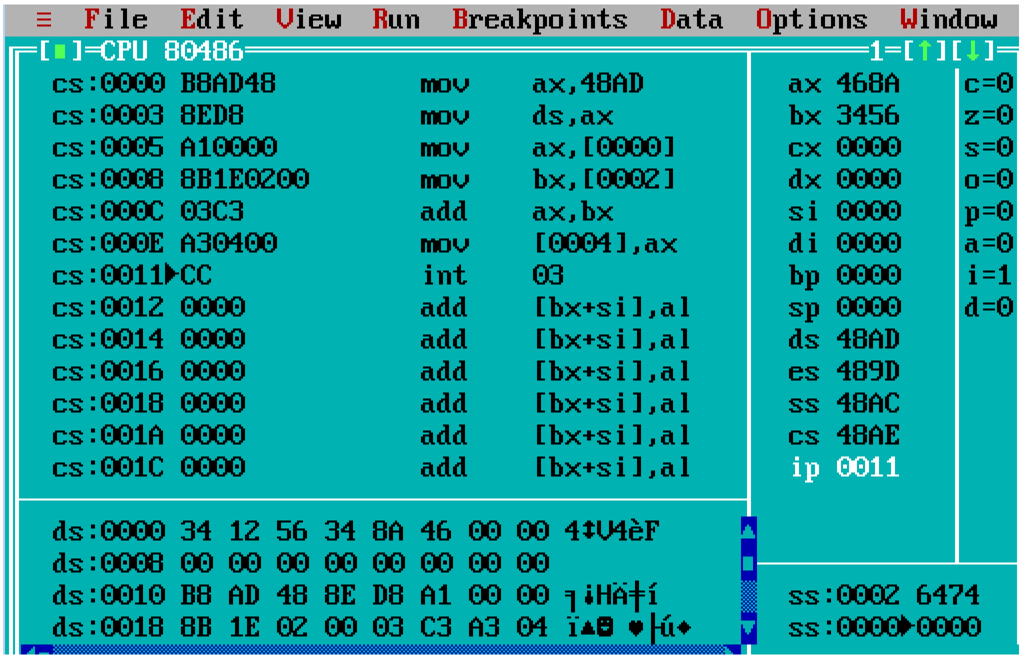
INPUT: 10h, 30h
OUTPUT: 40h



```
[■]=CPU 80486==================================1=[↑][↓]=
  cs:0000 B8AD48        mov    ax,48AD        ax 0040   c=0
  cs:0003 8ED8          mov    ds,ax          bx 0030   z=0
  cs:0005 A10000        mov    ax,[0000]      cx 0000   s=0
  cs:0008 8B1E0200      mov    bx,[0002]      dx 0000   o=0
  cs:000C 03C3          add    ax,bx          si 0000   p=0
  cs:000E A30400        mov    [0004],ax      di 0000   a=0
  cs:0011►CC            int    03             bp 0000   i=1
  cs:0012 0000          add    [bx+si],al     sp 0000   d=0
  cs:0014 0000          add    [bx+si],al     ds 48AD
  cs:0016 0000          add    [bx+si],al     es 489D
  cs:0018 0000          add    [bx+si],al     ss 48AC
  cs:001A 0000          add    [bx+si],al     cs 48AE
  cs:001C 0000          add    [bx+si],al     ip 0011

  ds:0000 10 00 30 00 40 00 00 00 ► 0 @
  ds:0008 00 00 00 00 00 00 00 00
  ds:0010 B8 AD 48 8E D8 A1 00 00 ┐↓HÄ╪í      ss:0002 6474
  ds:0018 8B 1E 02 00 03 C3 A3 04 ï▲☻ ♥├ú♦     ss:0000►0000
```

ADDITION OF TWO 16-Bit NUMBERS
Algorithm
1. Initialize memory pointer to data location.
2. Get the first 16-bit number from memory in accumulator.
3. Get the second 16-bit number and add it to the accumulator.
4. Store the answer at another memory location.

INPUT: 1234h, 3456h
OUTPUT: 468A



```
cs:0000 B8AD48        mov    ax,48AD        ax 468A   c=0
cs:0003 8ED8          mov    ds,ax          bx 3456   z=0
cs:0005 A10000        mov    ax,[0000]      cx 0000   s=0
cs:0008 8B1E0200      mov    bx,[0002]      dx 0000   o=0
cs:000C 03C3          add    ax,bx          si 0000   p=0
cs:000E A30400        mov    [0004],ax      di 0000   a=0
cs:0011►CC            int    03             bp 0000   i=1
cs:0012 0000          add    [bx+si],al     sp 0000   d=0
cs:0014 0000          add    [bx+si],al     ds 48AD
cs:0016 0000          add    [bx+si],al     es 489D
cs:0018 0000          add    [bx+si],al     ss 48AC
cs:001A 0000          add    [bx+si],al     cs 48AE
cs:001C 0000          add    [bx+si],al     ip 0011

ds:0000 34 12 56 34 8A 46 00 00 4‡V4èF
ds:0008 00 00 00 00 00 00 00 00
ds:0010 B8 AD 48 8E D8 A1 00 00 ╕¡HÄ╪í        ss:0002 6474
ds:0018 8B 1E 02 00 03 C3 A3 04 ï▲☻ ♥├ú♦      ss:0000►0000
```

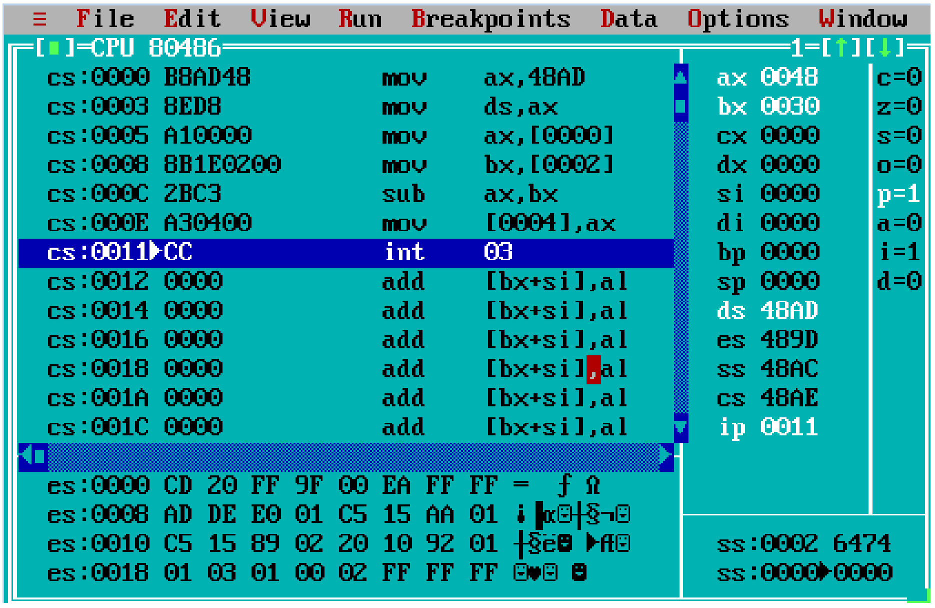
SUBTRACTION OF TWO 8-Bit NUMBERS

Algorithm

1. Initialize memory pointer to data location.
2. Get the first 8-bit number from memory in accumulator.
3. Get the second 8-bit number and subtract it to the accumulator.
4. Store the answer at another memory location.

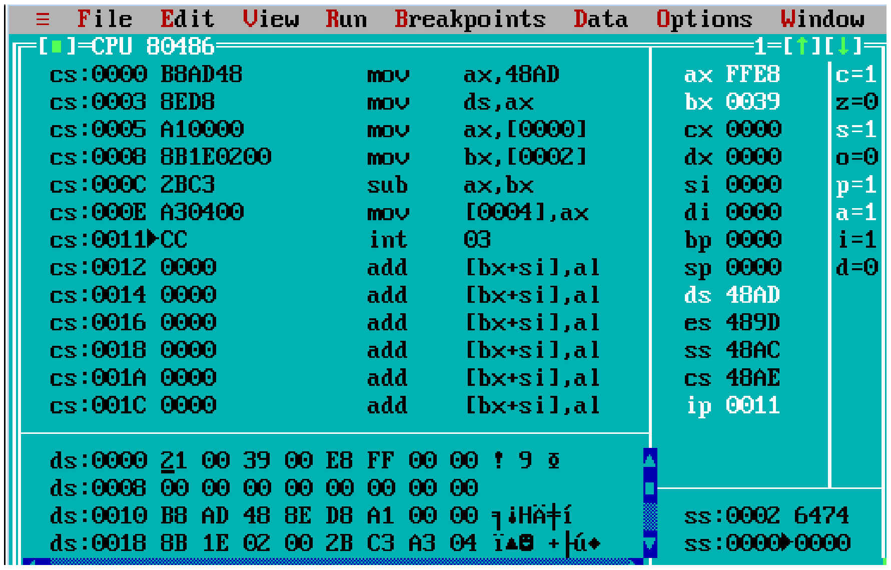
INPUT: 78h, 30h

OUTPUT: 48h



```
cs:0000 B8AD48        mov    ax,48AD        ax 0048   c=0
cs:0003 8ED8          mov    ds,ax          bx 0030   z=0
cs:0005 A10000        mov    ax,[0000]      cx 0000   s=0
cs:0008 8B1E0200      mov    bx,[0002]      dx 0000   o=0
cs:000C 2BC3          sub    ax,bx          si 0000   p=1
cs:000E A30400        mov    [0004],ax      di 0000   a=0
cs:0011►CC            int    03             bp 0000   i=1
cs:0012 0000          add    [bx+si],al     sp 0000   d=0
cs:0014 0000          add    [bx+si],al     ds 48AD
cs:0016 0000          add    [bx+si],al     es 489D
cs:0018 0000          add    [bx+si],al     ss 48AC
cs:001A 0000          add    [bx+si],al     cs 48AE
cs:001C 0000          add    [bx+si],al     ip 0011

es:0000 CD 20 FF 9F 00 EA FF FF ═  ƒ Ω
es:0008 AD DE E0 01 C5 15 AA 01 ¡▐α☺┼§¬☺
es:0010 C5 15 89 02 20 10 92 01 ┼§ë☻ ►Æ☺       ss:0002 6474
es:0018 01 03 01 00 02 FF FF FF ☺♥☺ ☻          ss:0000►0000
```

Input: 21h, 39h
Output: FFE8h



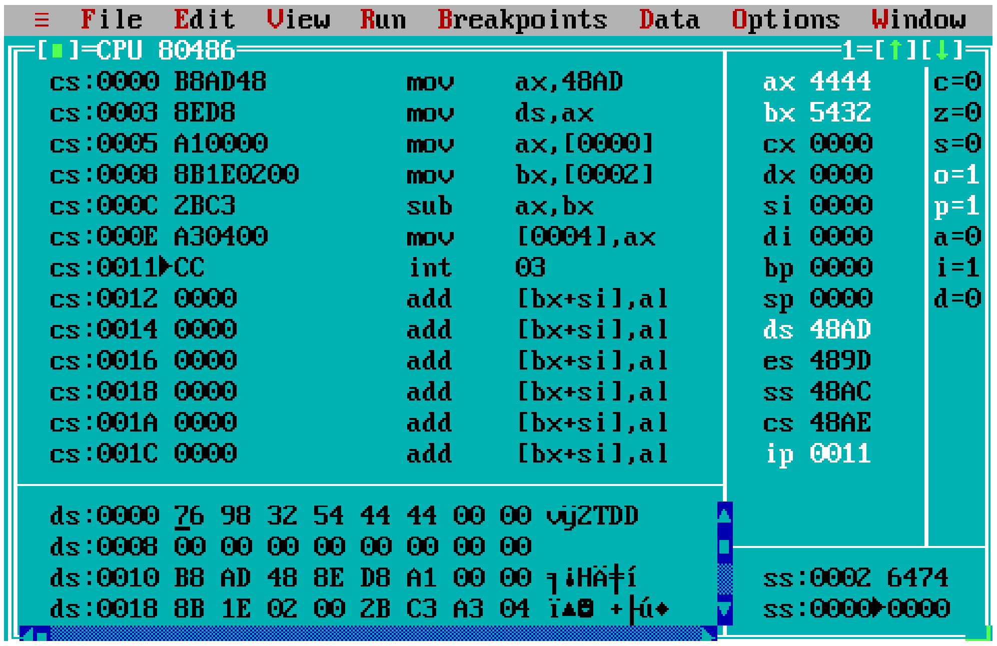
```
cs:0000 B8AD48         mov    ax,48AD      ax FFE8   c=1
cs:0003 8ED8           mov    ds,ax        bx 0039   z=0
cs:0005 A10000         mov    ax,[0000]    cx 0000   s=1
cs:0008 8B1E0200       mov    bx,[0002]    dx 0000   o=0
cs:000C 2BC3           sub    ax,bx        si 0000   p=1
cs:000E A30400         mov    [0004],ax    di 0000   a=1
cs:0011►CC             int    03           bp 0000   i=1
cs:0012 0000           add    [bx+si],al   sp 0000   d=0
cs:0014 0000           add    [bx+si],al   ds 48AD
cs:0016 0000           add    [bx+si],al   es 489D
cs:0018 0000           add    [bx+si],al   ss 48AC
cs:001A 0000           add    [bx+si],al   cs 48AE
cs:001C 0000           add    [bx+si],al   ip 0011

ds:0000 21 00 39 00 E8 FF 00 00 ! 9 Φ
ds:0008 00 00 00 00 00 00 00 00
ds:0010 B8 AD 48 8E D8 A1 00 00 ╕¡HÄ╪í      ss:0002 6474
ds:0018 8B 1E 02 00 2B C3 A3 04 ï▲☻ +├ú♦     ss:0000►0000
```

SUBTRACTION OF TWO 16-BIT NUMBERS

Algorithm

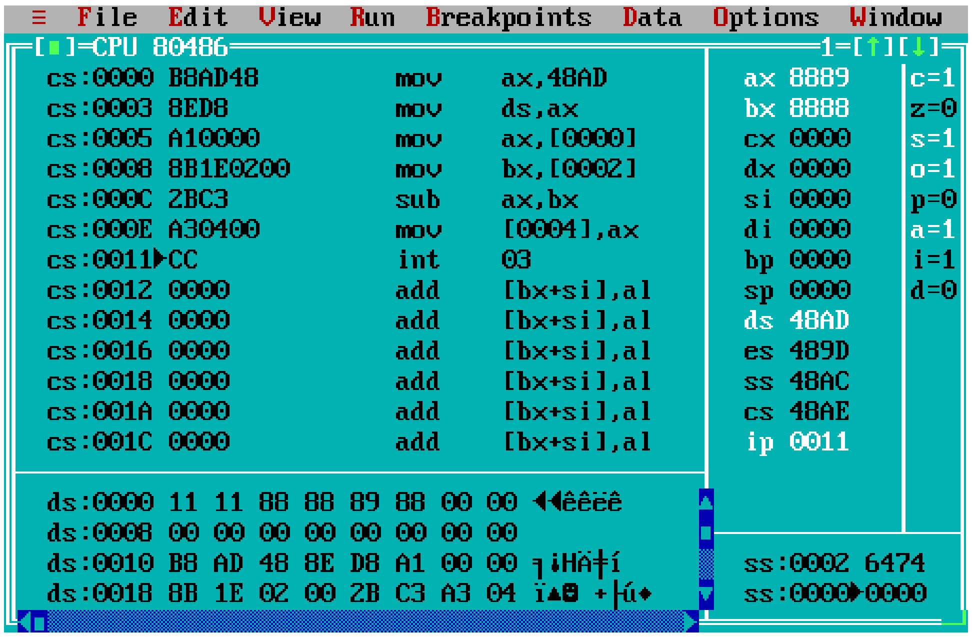
1. Initialize memory pointer to data location.
2. Get the first 16-bit number from memory in accumulator.
3. Get the second 16-bit number and subtract it to the accumulator.
4. Store the answer at another memory location.

INPUT: 9876h, 5432h
OUTPUT: 4444h



```
cs:0000 B8AD48         mov    ax,48AD      ax 4444   c=0
cs:0003 8ED8           mov    ds,ax        bx 5432   z=0
cs:0005 A10000         mov    ax,[0000]    cx 0000   s=0
cs:0008 8B1E0200       mov    bx,[0002]    dx 0000   o=1
cs:000C 2BC3           sub    ax,bx        si 0000   p=1
cs:000E A30400         mov    [0004],ax    di 0000   a=0
cs:0011►CC             int    03           bp 0000   i=1
cs:0012 0000           add    [bx+si],al   sp 0000   d=0
cs:0014 0000           add    [bx+si],al   ds 48AD
cs:0016 0000           add    [bx+si],al   es 489D
cs:0018 0000           add    [bx+si],al   ss 48AC
cs:001A 0000           add    [bx+si],al   cs 48AE
cs:001C 0000           add    [bx+si],al   ip 0011

ds:0000 76 98 32 54 44 44 00 00 vÿ2TDD
ds:0008 00 00 00 00 00 00 00 00
ds:0010 B8 AD 48 8E D8 A1 00 00 ╕¡HÄ╪í      ss:0002 6474
ds:0018 8B 1E 02 00 2B C3 A3 04 ï▲☻ +├ú♦     ss:0000►0000
```

INPUT: 1111h, 8888h
OUTPUT: 8889h



```
[■]=CPU 80486=======================================1=[↑][↓]=
  cs:0000 B8AD48         mov    ax,48AD          ax 8889   c=1
  cs:0003 8ED8           mov    ds,ax            bx 8888   z=0
  cs:0005 A10000         mov    ax,[0000]        cx 0000   s=1
  cs:0008 8B1E0200       mov    bx,[0002]        dx 0000   o=1
  cs:000C 2BC3           sub    ax,bx            si 0000   p=0
  cs:000E A30400         mov    [0004],ax        di 0000   a=1
  cs:0011►CC             int    03               bp 0000   i=1
  cs:0012 0000           add    [bx+si],al       sp 0000   d=0
  cs:0014 0000           add    [bx+si],al       ds 48AD
  cs:0016 0000           add    [bx+si],al       es 489D
  cs:0018 0000           add    [bx+si],al       ss 48AC
  cs:001A 0000           add    [bx+si],al       cs 48AE
  cs:001C 0000           add    [bx+si],al       ip 0011

  ds:0000 11 11 88 88 89 88 00 00 ◄◄êêëê
  ds:0008 00 00 00 00 00 00 00 00
  ds:0010 B8 AD 48 8E D8 A1 00 00 ╕¡HÄ╪í           ss:0002 6474
  ds:0018 8B 1E 02 00 2B C3 A3 04 ï▲☻ +├ú♦          ss:0000►0000
```

Experiment: 3

Aim: Write a program to calculate
1. Multiplication of 8 by 8 bit and 16 by 16 bit number
2. Division of 16 by 8 bit and 32 by 16 bit number

Objective: Multiplication and division of two 8 bit and two 16 bit numbers using Tasm.

Theory: Arithmetic operations Multiplication and Division

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.

MUL BH   Multiply AL with BH; result in AX
MUL CX   Multiply AX with CX; result high word in DX, low word in 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.
IMUL BH   Multiply signed byte in AL with signed byte in BH; result in AX.

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.

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.

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.

IDIV BL   Signed word in AX/signed byte in BL

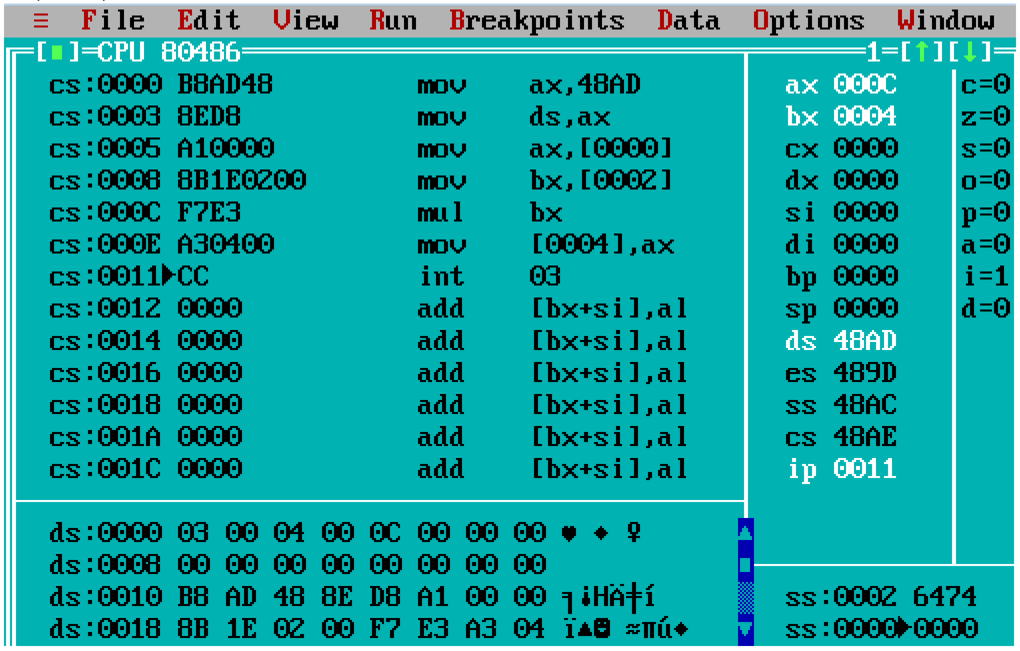
MULTIPLICATION OF TWO 8-BIT NOS

Algorithm:

1. Initialize memory pointer to data location
2. Get first no in al
3. Get second no in bl
4. Multiply contents of al with bl and put result in ax reg

MULTIPLICATION OF TWO 8-BIT UNSIGNED NUMBERS

INPUT: 03H, 04H, OUTPUT: 0CH



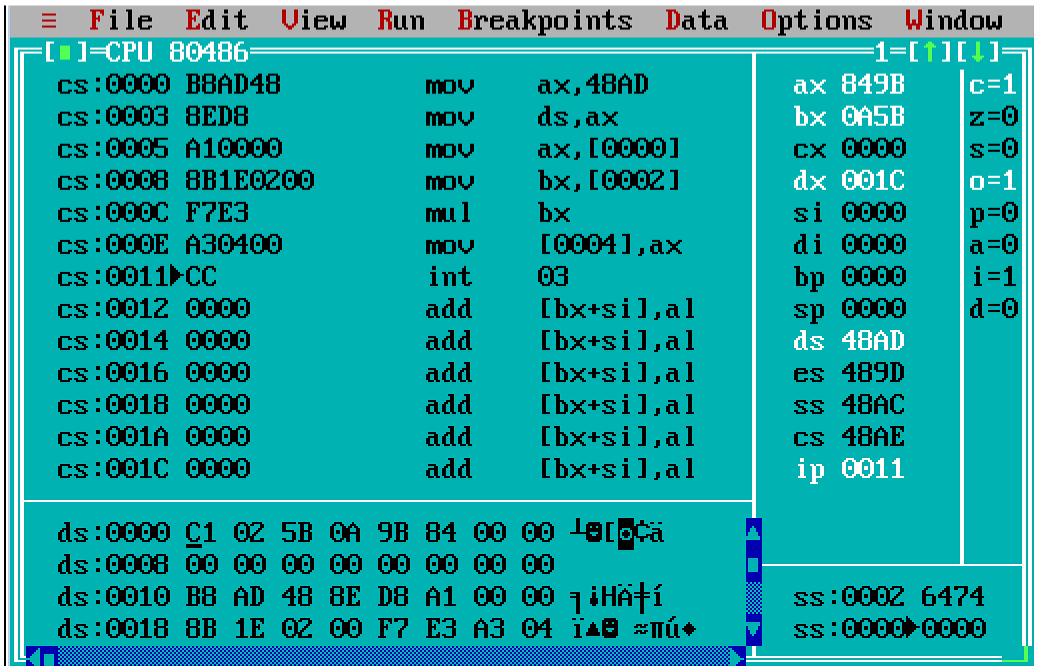
```
 ≡  File  Edit  View  Run  Breakpoints  Data  Options  Window
[■]=CPU 80486=========================================1=[↑][↓]=
  cs:0000 B8AD48        mov    ax,48AD        ax 000C   c=0
  cs:0003 8ED8          mov    ds,ax          bx 0004   z=0
  cs:0005 A10000        mov    ax,[0000]      cx 0000   s=0
  cs:0008 8B1E0200      mov    bx,[0002]      dx 0000   o=0
  cs:000C F7E3          mul    bx             si 0000   p=0
  cs:000E A30400        mov    [0004],ax      di 0000   a=0
  cs:0011►CC            int    03             bp 0000   i=1
  cs:0012 0000          add    [bx+si],al     sp 0000   d=0
  cs:0014 0000          add    [bx+si],al     ds 48AD
  cs:0016 0000          add    [bx+si],al     es 489D
  cs:0018 0000          add    [bx+si],al     ss 48AC
  cs:001A 0000          add    [bx+si],al     cs 48AE
  cs:001C 0000          add    [bx+si],al     ip 0011

  ds:0000 03 00 04 00 0C 00 00 00 ♥ ♦ ♀
  ds:0008 00 00 00 00 00 00 00 00
  ds:0010 B8 AD 48 8E D8 A1 00 00 ╕¡HÄ╪í        ss:0002 6474
  ds:0018 8B 1E 02 00 F7 E3 A3 04 ï▲☻ ≈πú♦       ss:0000►0000
```

MULTIPLICATION OF TWO 16-BIT UNSIGNED NUMBERS

Algorithm:

1. Initialize memory pointer to data location
2. Get first no in ax
3. Get second no in bx
4. Multiply contents of ax with bx and put result in ax:dx reg

INPUT: 02C1 H, 0A5B H, OUTPUT: 001C849B H



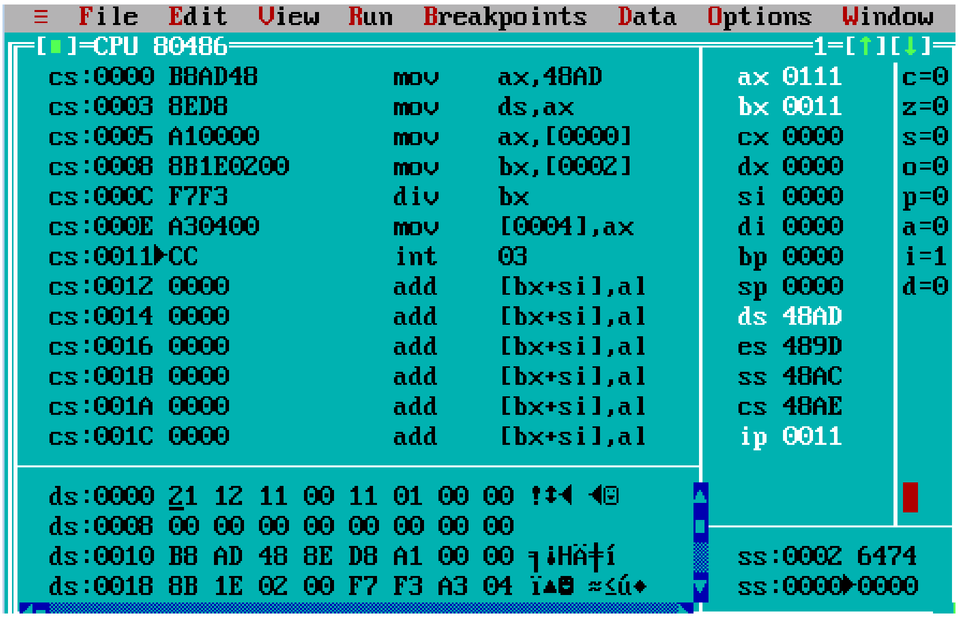
```
cs:0000 B8AD48        mov    ax,48AD        ax 849B   c=1
cs:0003 8ED8          mov    ds,ax          bx 0A5B   z=0
cs:0005 A10000        mov    ax,[0000]      cx 0000   s=0
cs:0008 8B1E0200      mov    bx,[0002]      dx 001C   o=1
cs:000C F7E3          mul    bx             si 0000   p=0
cs:000E A30400        mov    [0004],ax      di 0000   a=0
cs:0011►CC            int    03             bp 0000   i=1
cs:0012 0000          add    [bx+si],al     sp 0000   d=0
cs:0014 0000          add    [bx+si],al     ds 48AD
cs:0016 0000          add    [bx+si],al     es 489D
cs:0018 0000          add    [bx+si],al     ss 48AC
cs:001A 0000          add    [bx+si],al     cs 48AE
cs:001C 0000          add    [bx+si],al     ip 0011

ds:0000 C1 02 5B 0A 9B 84 00 00 ┴☻[◙¢ä
ds:0008 00 00 00 00 00 00 00 00
ds:0010 B8 AD 48 8E D8 A1 00 00 ╕¡HÄ╪í       ss:0002 6474
ds:0018 8B 1E 02 00 F7 E3 A3 04 ï▲☻ ≈πú♦     ss:0000►0000
```

DIVISION OF 16 BY 8 BIT NUMBERS

Algorithm:

1. Get first no in ax(dividend)
2. Get second no in bl(divisor)
3. Divide contents of ax with bl(div instruction)
4. Get quotient in ax and remainder in dx

INPUT: 1221 H (dividend), 11H (divisor),
OUTPUT: 111 H (Quotient), 0 (Remainder)



```
cs:0000 B8AD48        mov    ax,48AD        ax 0111   c=0
cs:0003 8ED8          mov    ds,ax          bx 0011   z=0
cs:0005 A10000        mov    ax,[0000]      cx 0000   s=0
cs:0008 8B1E0200      mov    bx,[0002]      dx 0000   o=0
cs:000C F7F3          div    bx             si 0000   p=0
cs:000E A30400        mov    [0004],ax      di 0000   a=0
cs:0011►CC            int    03             bp 0000   i=1
cs:0012 0000          add    [bx+si],al     sp 0000   d=0
cs:0014 0000          add    [bx+si],al     ds 48AD
cs:0016 0000          add    [bx+si],al     es 489D
cs:0018 0000          add    [bx+si],al     ss 48AC
cs:001A 0000          add    [bx+si],al     cs 48AE
cs:001C 0000          add    [bx+si],al     ip 0011

ds:0000 21 12 11 00 11 01 00 00 !↕◄ ◄☺
ds:0008 00 00 00 00 00 00 00 00
ds:0010 B8 AD 48 8E D8 A1 00 00 ╕¡HÄ╪í       ss:0002 6474
ds:0018 8B 1E 02 00 F7 F3 A3 04 ï▲☻ ≈≤ú♦     ss:0000►0000
```

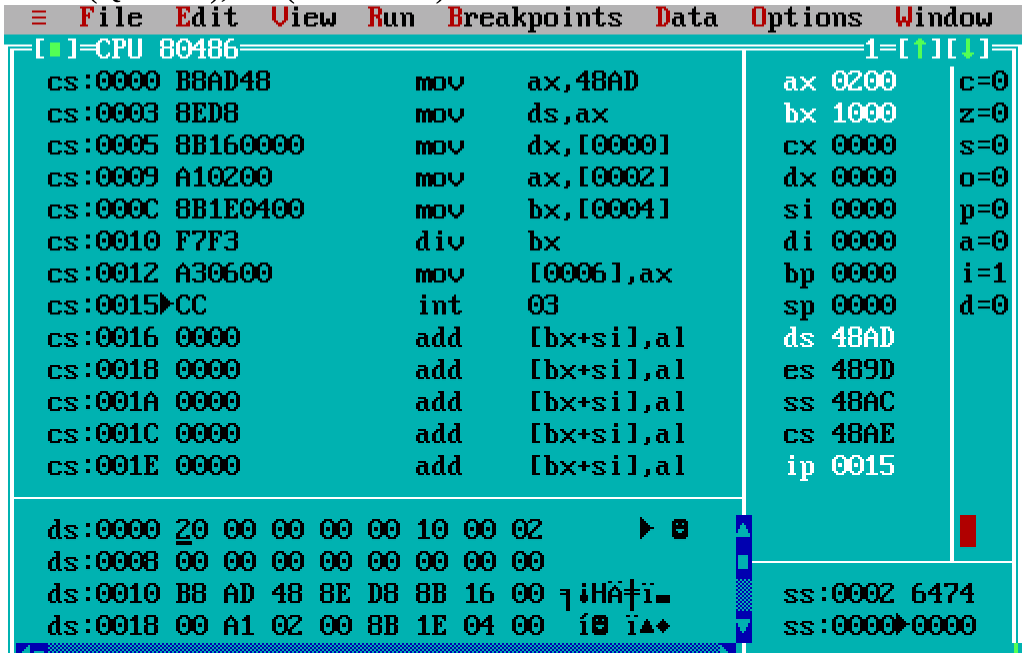
DIVISION OF 32 BY 16 BIT UNSIGNED NUMBERS

Algorithm:

1. Copy higher bits of register into data register
2. Copy lower bits of register into accumulator
3. Copy value of divisor to base register
4. Divide ax:dx contents by bx put quotient in dx and remainder in ax

INPUT: 200000 H (dividend), 1000 H (divisor)

OUTPUT: 200 (Quotient), 000(Remainder)



```
cs:0000 B8AD48        mov   ax,48AD        ax 0200   c=0
cs:0003 8ED8          mov   ds,ax          bx 1000   z=0
cs:0005 8B160000      mov   dx,[0000]      cx 0000   s=0
cs:0009 A10200        mov   ax,[0002]      dx 0000   o=0
cs:000C 8B1E0400      mov   bx,[0004]      si 0000   p=0
cs:0010 F7F3          div   bx             di 0000   a=0
cs:0012 A30600        mov   [0006],ax      bp 0000   i=1
cs:0015▶CC            int   03             sp 0000   d=0
cs:0016 0000          add   [bx+si],al     ds 48AD
cs:0018 0000          add   [bx+si],al     es 489D
cs:001A 0000          add   [bx+si],al     ss 48AC
cs:001C 0000          add   [bx+si],al     cs 48AE
cs:001E 0000          add   [bx+si],al     ip 0015

ds:0000 20 00 00 00 00 10 00 02     ▶ ☻
ds:0008 00 00 00 00 00 00 00 00
ds:0010 B8 AD 48 8E D8 8B 16 00 ╕¡HÄ╪ï▬
ds:0018 00 A1 02 00 8B 1E 04 00  í☻ ï▲♦       ss:0002 6474
                                              ss:0000▶0000
```

Experiment: 4

Aim: Write a program for finding largest and smallest number in an array

Objective: Find out largest and smallest number for given array of numbers.

Theory:

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.

Example-
DEC CL Subtract 1 from content of CL register
DEC BP Subtract 1 from content of BP register

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.

Example-
INC BL Add 1 to contains of BL register
INC CX Add 1 to contains of CX register

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. AF, OF, SF, ZF, PF, and CF are updated by the CMP instruction

Example-
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

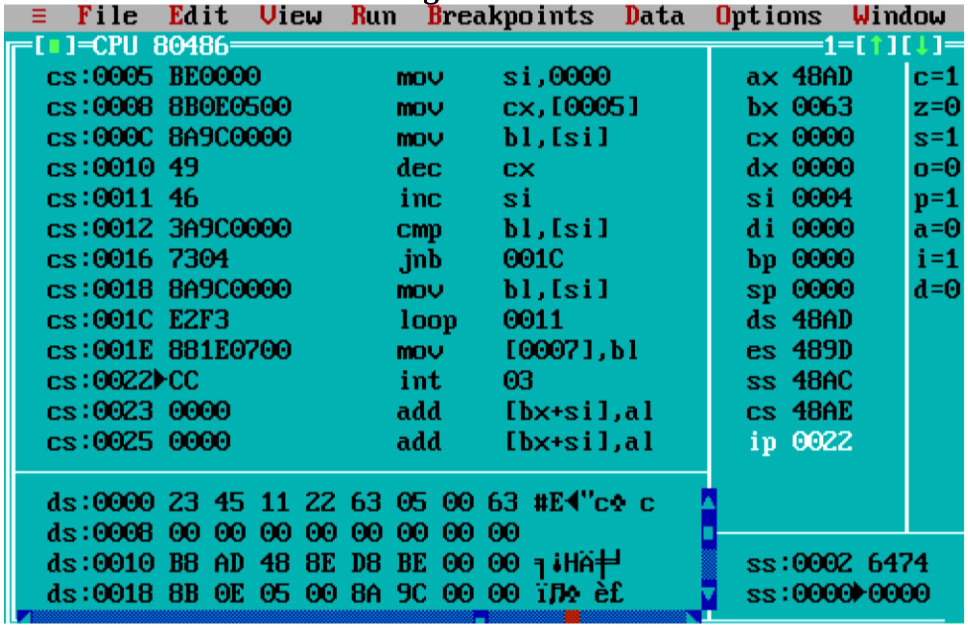
Example-
CMP AL, 01H Compare immediate number 01H with byte in AL
CMP BH, CL Compare byte in CL with byte in BH

<u>Largest number:</u>
Algorithm:

1. Initialise an array with an array of elements
2. Initialize total counter to number of elements
3. Initailise data segment
4. Initialize cx to counter
5. Load al with first element
6. Decrement counter
7. Load bl with a
8. compare bl with a[si]
9. carry is not generated then go to no swap
10. If first number > second number and carry is not generated goto no swap
11. Move bl into largest
12. Otherwise mov bl to a[si]

OUTPUT:

Largest number



```
cs:0005 BE0000        mov    si,0000        ax 48AD   c=1
cs:0008 8B0E0500      mov    cx,[0005]      bx 0063   z=0
cs:000C 8A9C0000      mov    bl,[si]        cx 0000   s=1
cs:0010 49            dec    cx             dx 0000   o=0
cs:0011 46            inc    si             si 0004   p=1
cs:0012 3A9C0000      cmp    bl,[si]        di 0000   a=0
cs:0016 7304          jnb    001C           bp 0000   i=1
cs:0018 8A9C0000      mov    bl,[si]        sp 0000   d=0
cs:001C E2F3          loop   0011           ds 48AD
cs:001E 881E0700      mov    [0007],bl      es 489D
cs:0022►CC            int    03             ss 48AC
cs:0023 0000          add    [bx+si],al     cs 48AE
cs:0025 0000          add    [bx+si],al     ip 0022

ds:0000 23 45 11 22 63 05 00 63 #E◄"c♠ c
ds:0008 00 00 00 00 00 00 00 00             ss:0002 6474
ds:0010 B8 AD 48 8E D8 BE 00 00 ╕¡HÄ╪╛      ss:0000►0000
ds:0018 8B 0E 05 00 8A 9C 00 00 ï♫♣ è£
```

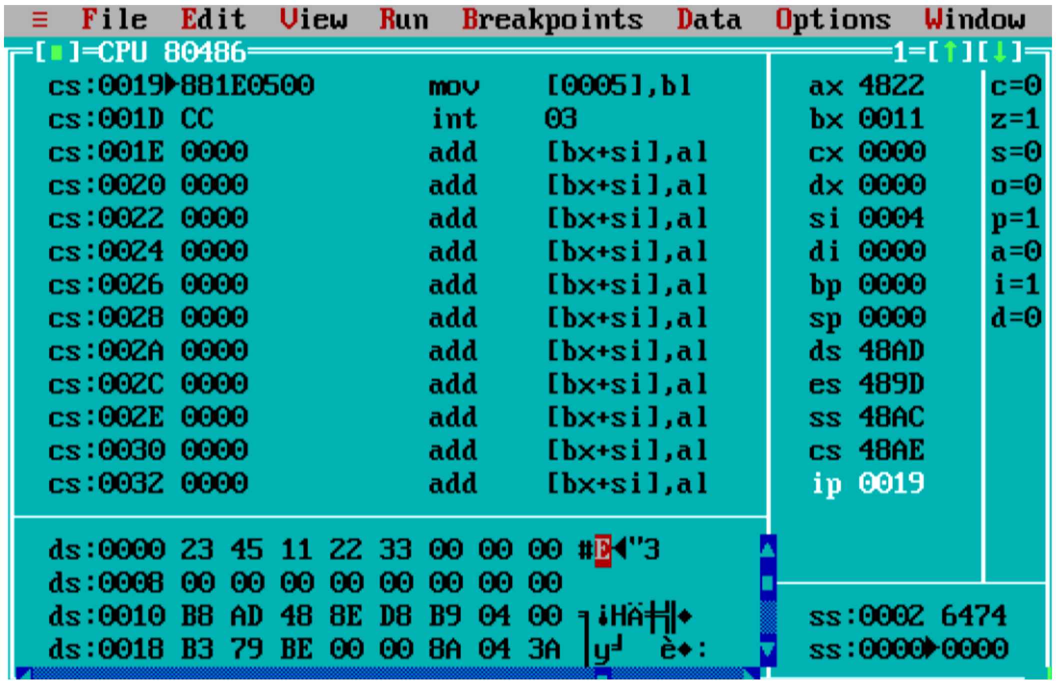
<u>Smallest number</u>

Algorithm:

1. Initialise an array with an array of elements
2. Initialize total counter to number of elements
3. Initailise data segment
4. Initialize cx to counter
5. Load al with first element
6. Decrement counter
7. Load bl with a
8. compare bl with a[si]
9. carry is generated then go to no swap
10. If first number > second number and carry is not generated goto no swap
11. Move bl into largest
12. Otherwise movbl to a[si]

OUTPUT:

Smallest number



```
cs:0019▶881E0500         mov    [0005],bl       ax 4822   c=0
cs:001D CC               int    03              bx 0011   z=1
cs:001E 0000             add    [bx+si],al      cx 0000   s=0
cs:0020 0000             add    [bx+si],al      dx 0000   o=0
cs:0022 0000             add    [bx+si],al      si 0004   p=1
cs:0024 0000             add    [bx+si],al      di 0000   a=0
cs:0026 0000             add    [bx+si],al      bp 0000   i=1
cs:0028 0000             add    [bx+si],al      sp 0000   d=0
cs:002A 0000             add    [bx+si],al      ds 48AD
cs:002C 0000             add    [bx+si],al      es 489D
cs:002E 0000             add    [bx+si],al      ss 48AC
cs:0030 0000             add    [bx+si],al      cs 48AE
cs:0032 0000             add    [bx+si],al      ip 0019

ds:0000 23 45 11 22 33 00 00 00 #E◄"3
ds:0008 00 00 00 00 00 00 00 00
ds:0010 B8 AD 48 8E D8 B9 04 00 ╕¡HÄ╪╣♦       ss:0002 6474
ds:0018 B3 79 BE 00 00 8A 04 3A │y╛  è♦:      ss:0000▶0000
```

Experiment 5

Aim: Write a program to find factorial of a given number using mixed language

Objective: Find out factorial of a given number using mixed language

Theory:

Mixed language programming

It involves a call to a function, procedure or subroutine. Mixed language call involves calling functions in separate modules. Instead of compiling all source programs with same compilers, different compilers or assemblers are used as per language used in program.

Microsoft supports this mixed language programming. So it combines assembly code routines in C as a separate language program calls assembly routines that are separately assembled by TASM. These assembled modules are linked with compiled C modules. Instead of compiling all source program with same compiler

METHOD 1

Built in Inline assembler is used to include assembly language routine in your C program without any need for separate assembler.

METHOD 2

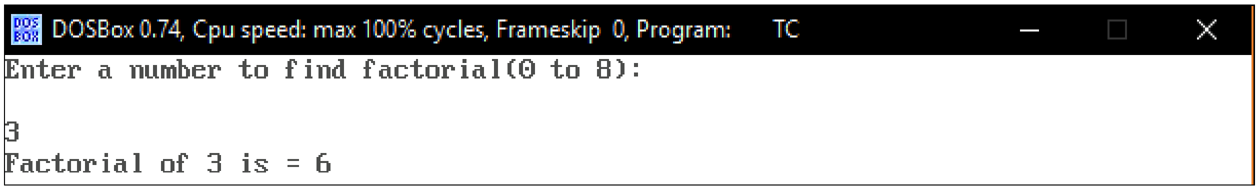
Program need to call program written in other languages referred as mixed language. Keyword asm is required to allow assembly language code in C program. Inline assembly instruction language format is:

```
asm opcode operands
asm mov ah,02h
asm int 21h
OR
asm{    mov ah,02
        mov dl,30h
        int 21h
      }
```

Algorithm:

1. Initialize al to 01.
2. Initialize cx to 0000
3. Move number whose factorial is to be taken in cx.(a in cx)
4. Multiply cx with 1
5. Decrement cx until zero if not zero jump to back1
6. Move value in ax to c,
7. print c(factorial).

OUTPUT:


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
Enter a number to find factorial(0 to 8):

3
Factorial of 3 is = 6
```

Experiment 6

Aim: Write a program to move a block of data from one memory location to another using string

Objective: Working on string instructions MOVS, REP, CLD

Theory:
Performing block transfer from one memory location to another memory location.

STRING DATA TRANSFERS

- Instructions for moving large blocks of data or *strings.*
- Five string data transfer instructions: LODS, STOS, MOVS, INS, and OUTS.
- Each allows data transfers as a single byte, word, or doubleword.
- Before the string instructions are presented, the operation of the D flag-bit (direction), DI, and SI must be understood as they apply to the string 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.

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.

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.

Example-

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

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.

MOV DI, OFFSET TARGET
STOS TARGET

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.

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

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 CMPSB // Compare string bytes until end of string or until string bytes not equal.

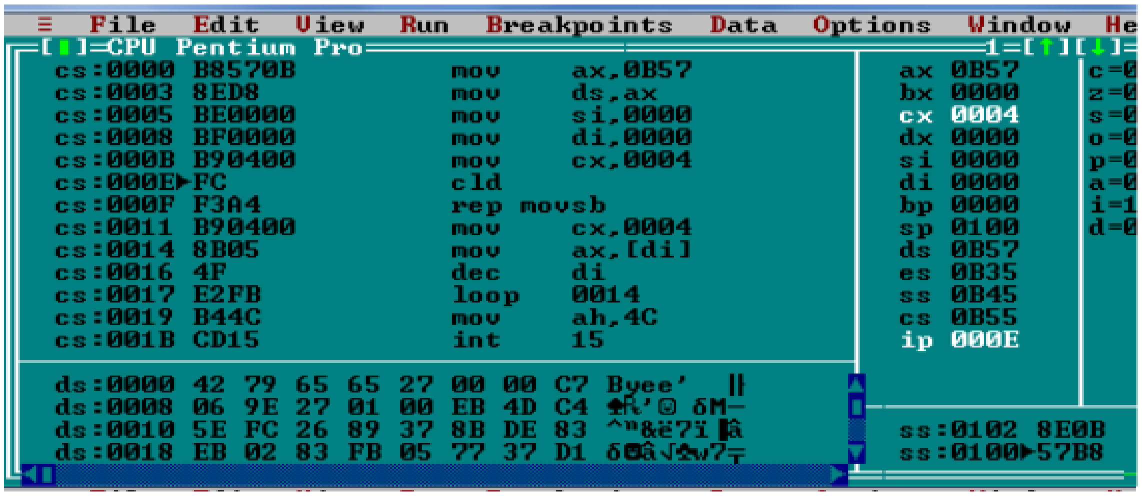
The Direction Flag

- The direction flag (D, located in the flag register) selects the auto-increment or the auto-decrement operation for the DI and SI registers during string operations.
- used only with the string instructions
- STD set direction flag so that the pointers are auto decremented.
- CLD clear direction flag so that the pointers are auto incremented.

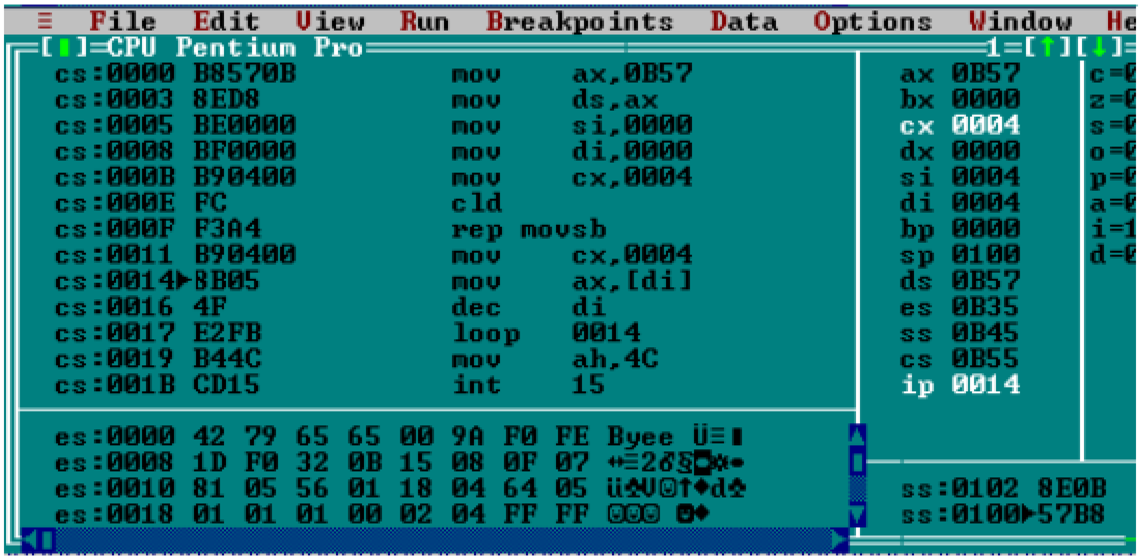
Algorithm

1. Initialise data segment.
2. Initialise si with 0000
3. Initialise di with 0000
4. Initialise cx counter
5. Clear direction flag so that you will automatically increment the si and di
6. repeat mov sb and transfer movement of string byte by byte in destination string. Loop until counter is 0.
7. Terminate program

OUTPUT: for moving a block of data from one memory location to another using string



```
 ≡   File   Edit   View   Run   Breakpoints   Data   Options   Window   He
=[■]=CPU Pentium Pro=                                        1=[↑][↓]=
  cs:0000 B8570B         mov     ax,0B57        ax 0B57    c=0
  cs:0003 8ED8           mov     ds,ax          bx 0000    z=0
  cs:0005 BE0000         mov     si,0000        cx 0004    s=0
  cs:0008 BF0000         mov     di,0000        dx 0000    o=0
  cs:000B B90400         mov     cx,0004        si 0000    p=0
  cs:000E►FC             cld                    di 0000    a=0
  cs:000F F3A4           rep movsb              bp 0000    i=1
  cs:0011 B90400         mov     cx,0004        sp 0100    d=0
  cs:0014 8B05           mov     ax,[di]        ds 0B57
  cs:0016 4F             dec     di             es 0B35
  cs:0017 E2FB           loop    0014           ss 0B45
  cs:0019 B44C           mov     ah,4C          cs 0B55
  cs:001B CD15           int     15             ip 000E

  ds:0000 42 79 65 65 27 00 00 C7 Byee'  ╟
  ds:0008 06 9E 27 01 00 EB 4D C4 ♠₧'☺ δM─
  ds:0010 5E FC 26 89 37 8B DE 83 ^ⁿ&ë7ï▐â     ss:0102 8E0B
  ds:0018 EB 02 83 FB 05 77 37 D1 δ☻â√♣w7╤     ss:0100►57B8
```



```
 ≡   File   Edit   View   Run   Breakpoints   Data   Options   Window   He
=[■]=CPU Pentium Pro=                                        1=[↑][↓]=
  cs:0000 B8570B         mov     ax,0B57        ax 0B57    c=0
  cs:0003 8ED8           mov     ds,ax          bx 0000    z=0
  cs:0005 BE0000         mov     si,0000        cx 0004    s=0
  cs:0008 BF0000         mov     di,0000        dx 0000    o=0
  cs:000B B90400         mov     cx,0004        si 0004    p=0
  cs:000E FC             cld                    di 0004    a=0
  cs:000F F3A4           rep movsb              bp 0000    i=1
  cs:0011 B90400         mov     cx,0004        sp 0100    d=0
  cs:0014►8B05           mov     ax,[di]        ds 0B57
  cs:0016 4F             dec     di             es 0B35
  cs:0017 E2FB           loop    0014           ss 0B45
  cs:0019 B44C           mov     ah,4C          cs 0B55
  cs:001B CD15           int     15             ip 0014

  es:0000 42 79 65 65 00 9A F0 FE Byee Ü≡■
  es:0008 1D F0 32 0B 15 08 0F 07 ↔≡2♂§◘☼•
  es:0010 81 05 56 01 18 04 64 05 ü♣V☺↑♦d♣     ss:0102 8E0B
  es:0018 01 01 01 00 02 04 FF FF ☺☺☺ ☻♦       ss:0100►57B8
```

Experiment 7

Aim: Write a program to count number of vowels in a given string

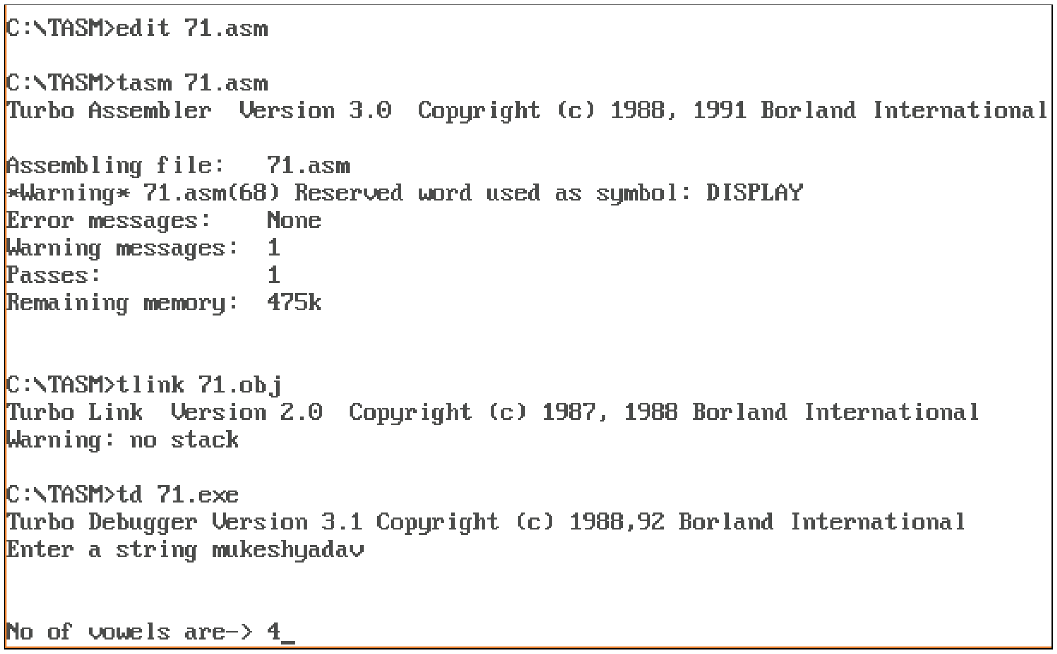
Objective: Count vowels in a given string

Theory:
English has five proper vowel letters (A, E, I, O, U), sometimes Y is also considered as a vowel and all alphabets except these characters are consonants. There are 10 digits in decimal number systems, from '0' to '9'. There is space character ' ' and white space characters like tab and new line. A string may contain combination of above. In this experiment: a, e, i, o, u, A, E, I, O, U is taken into consideration as a string may contain upper case or lower case letters.

Algorithm

1. Take input from the user
2. Store it in a register (say dx)
3. Store list of vowel in an array
4. Mov dx to si, now si is having input string
5. Initialize bl to 00H. This is vowel counter.
6. List of vowels are in di, input characters in al
7. Comparing al and di
8. Jump if destination (i.e. al) equals source (i.e. di) i.e. match found
9. Increment counter i.e. bl
10. If not equal, increment di (i.e. next vowel)
11. Increment si
12. Final bl count value is moved to dl
13. convert bcd to ascii by adding 30H to the dl
14. display output on the screen
15. Terminate the program

OUTPUT:



```
C:\TASM>edit 71.asm

C:\TASM>tasm 71.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland International

Assembling file:   71.asm
*Warning* 71.asm(68) Reserved word used as symbol: DISPLAY
Error messages:    None
Warning messages:  1
Passes:            1
Remaining memory:  475k


C:\TASM>tlink 71.obj
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
Warning: no stack

C:\TASM>td 71.exe
Turbo Debugger Version 3.1 Copyright (c) 1988,92 Borland International
Enter a string mukeshyadav


No of vowels are-> 4_
```

Experiment 8

Aim: Write a program to compare two strings using MACRO and comparing two Strings

Objective: Compare if two strings are same strings.

Theory:

MACRO

Macro can be defined as a group of repetitive instructions in a program that are codified only once but can be repeated n number of times.

Syntax:-

```
Macro_name MACRO argument1, argument2,…..argument n
        Statement 1
        Statement 2
        Statement k
EndM
```

- Example:

```
Read MACRO
      mov ah,01h
      int 21h
ENDM

Display MACRO
      mov dl,al
      mov ah,02h
      in 21h
ENDM
```

- Passing parameters to a macro

```
Display MACRO MSG
      mov dx, offset msg
      mov ah,09h
      int 21h
ENDM
```

- The parameter MSG can be replaced by msg1 or msg2 while calling..

  Calling macro:

```
      DISPLAY MSG1
      DISPLAY MSG2

      MSG1 db " I am Fine $"
      MSG2 db "Hello, How are you..? $"
```

- An actual argument can be any variable, immediate value or a register name. They may also be duplicates of other names like labels, variables etc.

- A macro call designed to generate assembly language instructions must be called in a code segment.
- A macro call designed to generate data definitions must appear in a portion o f a program where it will not be considered as an instruction.
- A macro definition can occur anywhere before END directive.
- A macro definition cannot occur inside another macro. Usually all macros are placed at beginning of program before segment definition.
- A macro definition is not assembled until macro is called. Each macro call is replaced by statements in the macro.

Nested macros.
A macro calling another macro is called a nested macro. TASM and MASM can contain a macro that calls a previously defined macros. All macros should be defined before segment definition though their order can be anything.

```
DISPLAY_CHAR  MACRO  CHAR
                    PUSH  AX
                    PUSH  DX
                    MOV  AH , 02H
                    MOV  DL ,  CHAR
                    INT  21H
                    POP  DX
                    POP  AX
ENDM
CRLF  MACRO
                DIPLAY_CHAR  0DH
                DISPLAY_CHAR  0AH
ENDM
```

<u>Procedure Vs Macros</u>

Procedure	Macro
Accessed by CALL and RET mechanism during execution	Accessed by name given to macro when defined during assembly
Machine code for instructions only put in memory space	Machine code generated for instructions each time called
Parameters are passed in registers, memory locations or stack	Parameters passed as part of statement which calls macro
Procedure uses stack	Macro does not utilize stack
A procedure can be defined anywhere in program using the directives PROC and ENDP	A macro can be defined anywhere in program using the directives MACRO and ENDM
Procedures take huge memory for CALL (3 byte each time CALL is used) instruction	Length of code is very huge if macro's are called for more number of times

<u>String Instructions</u>
Each string instruction may require a source operand, a destination operand or both. For 32-bit segments, string instructions use ESI and EDI registers to point to the source and destination operands, respectively.

For 16-bit segments, however, the SI and the DI registers are used to point to the source and destination, respectively.

CMPS

- This instruction compares two data items in memory. Data could be of a byte size, word or doubleword. This instruction has a byte, word, and doubleword version, and string instructions can be repeated by using a repetition prefix.
- These instructions use the ES:DI and DS:SI pair of registers, where DI and SI registers contain valid offset addresses that refers to bytes stored in memory. SI is normally associated with DS (data segment) and DI is always associated with ES (extra segment).
- The DS:SI (or ESI) and ES:DI (or EDI) registers point to the source and destination operands, respectively. The source operand is assumed to be at DS:SI (or ESI) and the destination operand at ES:DI (or EDI) in memory.
- For 16-bit addresses, the SI and DI registers are used, and for 32-bit addresses, the ESI and EDI registers are used.

Repetition Prefixes

The REP prefix, when set before a string instruction, for example - REP MOVSB, causes repetition of the instruction based on a counter placed at the CX register. REP executes the instruction, decreases CX by 1, and checks whether CX is zero. It repeats the instruction processing until CX is zero.

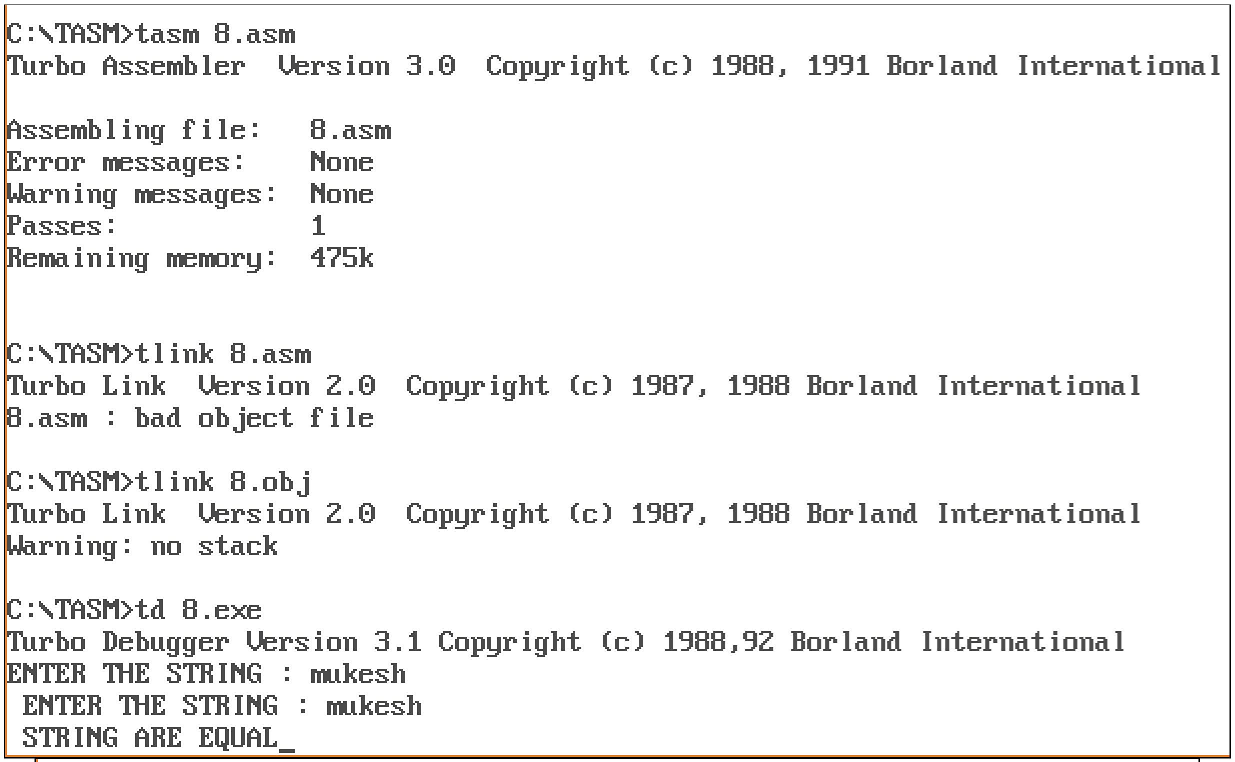
The Direction Flag (DF) determines the direction of the operation.

- Use CLD (Clear Direction Flag, DF = 0) to make the operation left to right.
- Use STD (Set Direction Flag, DF = 1) to make the operation right to left.

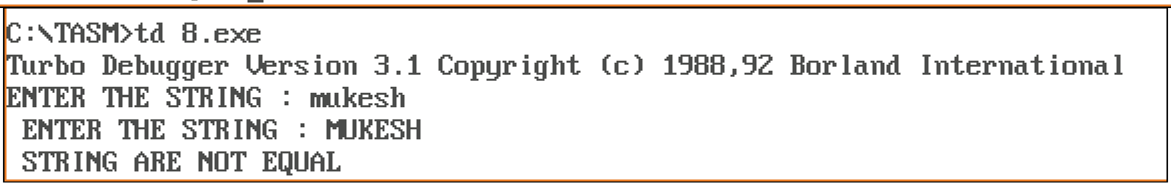
The REP prefix also has the following variations:

- REP: It is the unconditional repeat. It repeats the operation until CX is zero.
- REPE or REPZ: It is conditional repeat. It repeats the operation while the zero flag indicates equal/zero. It stops when the ZF indicates not equal/zero or when CX is zero.
- REPNE or REPNZ: It is also conditional repeat. It repeats the operation while the zero flag indicates not equal/zero. It stops when the ZF indicates equal/zero or when CX is decremented to zero.

OUTPUT:



```
C:\TASM>tasm 8.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland International

Assembling file:   8.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  475k


C:\TASM>tlink 8.asm
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
8.asm : bad object file

C:\TASM>tlink 8.obj
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
Warning: no stack

C:\TASM>td 8.exe
Turbo Debugger Version 3.1 Copyright (c) 1988,92 Borland International
ENTER THE STRING : mukesh
 ENTER THE STRING : mukesh
 STRING ARE EQUAL_
```



```
C:\TASM>td 8.exe
Turbo Debugger Version 3.1 Copyright (c) 1988,92 Borland International
ENTER THE STRING : mukesh
 ENTER THE STRING : MUKESH
 STRING ARE NOT EQUAL
```

Experiment 9

Aim: Write a program to display string with INT 21h
Objective: Study about 21h interrupt

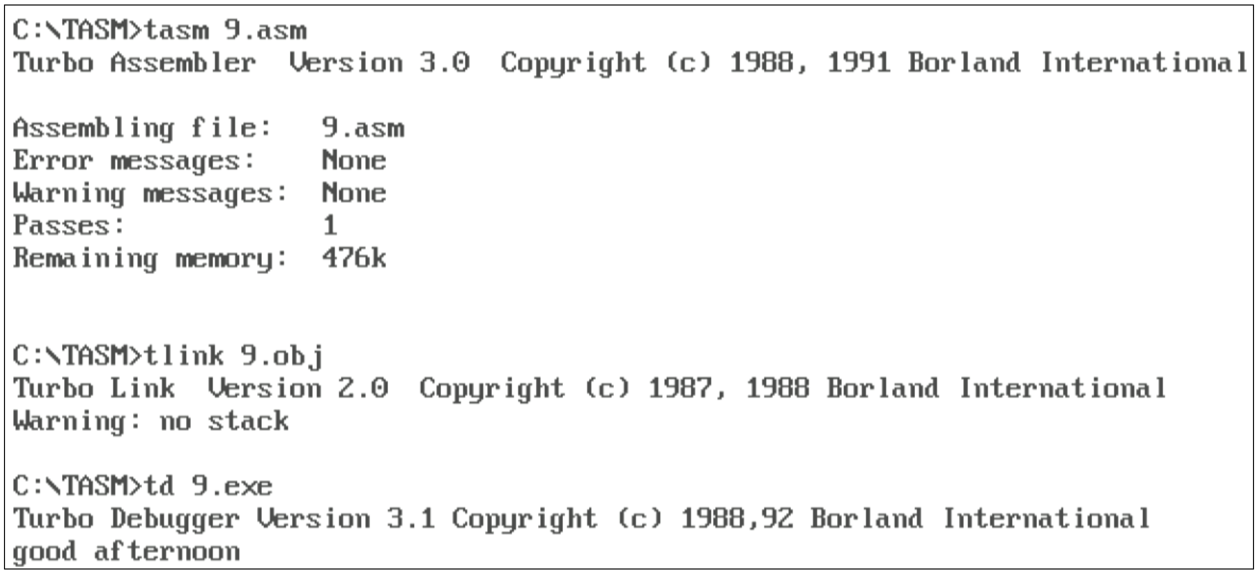
Theory:
Interrupt

- It is generated by the program
- The INT instruction generates a software interrupt which is provided by DOS.
- INT instruction uses a single operand to indicate which MS-DOS sub programs should be invoked.
- When MS-DOS is loaded into the computer, INT 21H can be invoked to perform some extremely useful functions.
- These functions are commonly referred to as DOS INT 21H function calls
- Data input and output I/O operations through the keyboard and monitor are the most commonly used functions.
- Here number stored in AH register is used to specify which kind of I/O operation is to be done.

Below are some examples that use DOS interrupts

- Display the message defined with variable
  ```
  DATA DB 'Microprocessor', '$'
  MOV AH, 09              ; option 9 to display string of data
  MOV DX, OFFSET DATA     ; DX= offset address of data
  INT 21H                 ; invoke the interrupt
  ```
- Inputting a single character, with echo
  ```
  MOV AH, 01              ; option 01 to input one character
  INT 21H                 ; Invoke the interrupt
  ```

OUTPUT:



```
C:\TASM>tasm 9.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland International

Assembling file:   9.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  476k


C:\TASM>tlink 9.obj
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
Warning: no stack

C:\TASM>td 9.exe
Turbo Debugger Version 3.1 Copyright (c) 1988,92 Borland International
good afternoon
```

Experiment 10

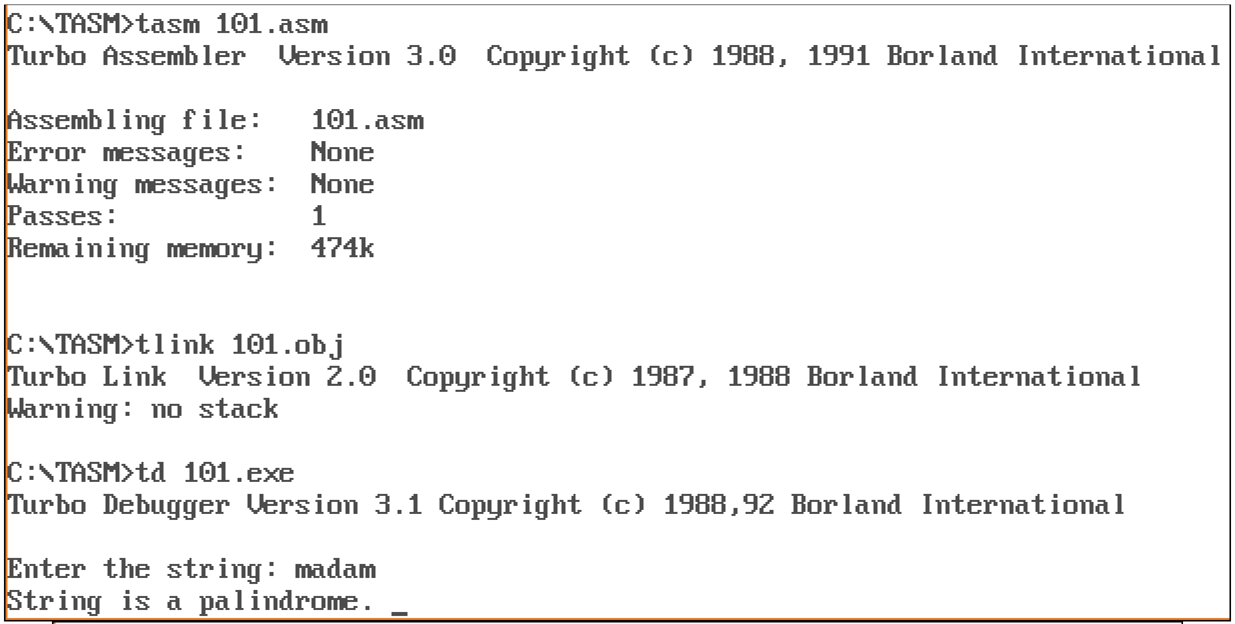
Aim: Write a program to check string is palindrome

Theory:
Palindrome means a word, phrase, or sequence that reads the same backwards as forwards, e.g. madam, malayalam ,etc.

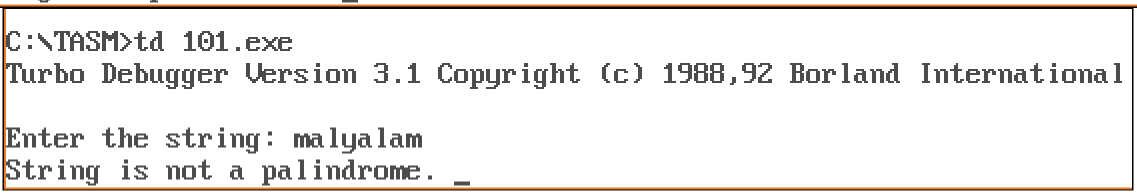
Algorithm

1. Take input from the user
2. Store string1
3. Store string2
4. Comparing character by character of string1 and string2
5. If equal i.e. all match found, display palindrome
6. If not equal, display not a palindrome
7. display output string on the screen
8. Terminate the program

OUTPUT:



```
C:\TASM>tasm 101.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland International

Assembling file:   101.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  474k


C:\TASM>tlink 101.obj
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
Warning: no stack

C:\TASM>td 101.exe
Turbo Debugger Version 3.1 Copyright (c) 1988,92 Borland International

Enter the string: madam
String is a palindrome. _
```



```
C:\TASM>td 101.exe
Turbo Debugger Version 3.1 Copyright (c) 1988,92 Borland International

Enter the string: malyalam
String is not a palindrome. _
```

Experiment 11

Aim: Interfacing of 8086 with Programmable Peripheral Interface 8255

Objective: Connect stepper motor through 8255 to 8086

Theory:

8255 Programmable Peripheral Interface and Interfacing

The 8255 is a widely used, programmable parallel I/O device. It can be programmed to transfer data under data under various conditions, from simple I/O to interrupt I/O. It is flexible, versatile and economical (when multiple I/O ports are required). It is an important general purpose I/O device that can be used with almost any microprocessor.

The 8255 has 24 I/O pins that can be grouped primarily into two 8 bit parallel ports: A and B, with the remaining 8 bits as Port C. The 8 bits of port C can be used as individual bits or be grouped into two 4 bit ports : CUpper (CU) and CLower (CL). The functions of these ports are defined by writing a control word in the control register. 8255 can be used in two modes: Bit set/Reset (BSR) mode and I/O mode. The BSR mode is used to set or reset the bits in port C. The I/O mode is further divided into 3 modes: mode 0, mode 1 and mode 2. In mode 0, all ports function as simple I/O ports. Mode 1 is a handshake mode whereby Port A and/or Port B use bits from Port C as handshake signals. In the handshake mode, two types of I/O data transfer can be implemented: status check and interrupt. In mode 2, Port A can be set up for bidirectional data transfer using handshake signals from Port C, and Port B can be set up either in mode 0 or mode 1.

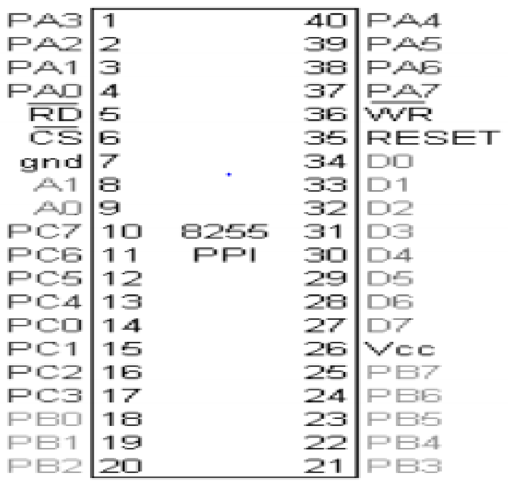


Fig. 6.1 Pin Configuration of 8255

Control Logic of 8255

RD (Read) : This signal enables the Read operation. When the signal is low, microprocessor reads data from a selected I/O port of 8255.

WR (Write) : This control signal enables the write operation.

RESET (Reset) : It clears the control registers and sets all ports in input mode.

CS, A0, A1: These are device select signals. is connected to a decoded address and A0, A1 are connected to A0, A1 of microprocessor.

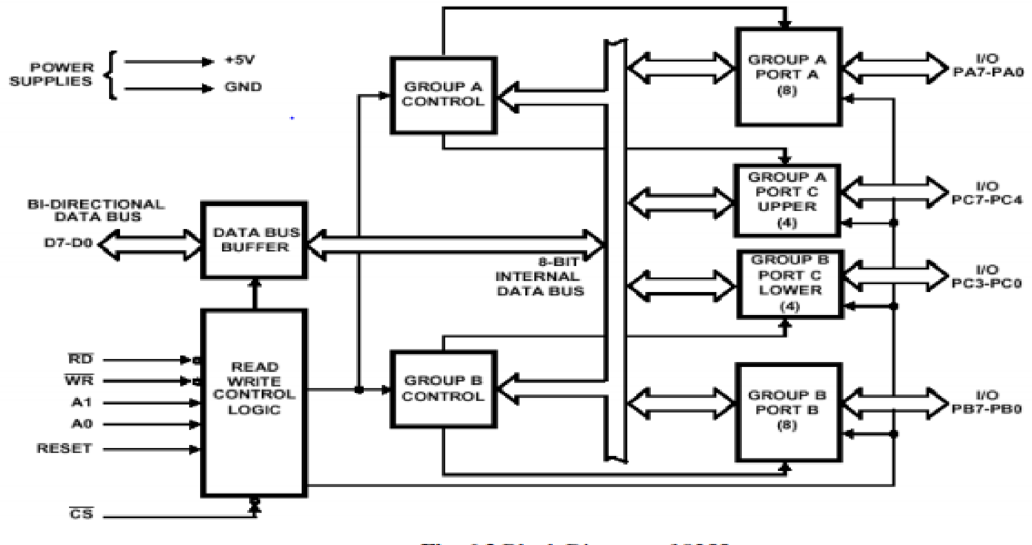
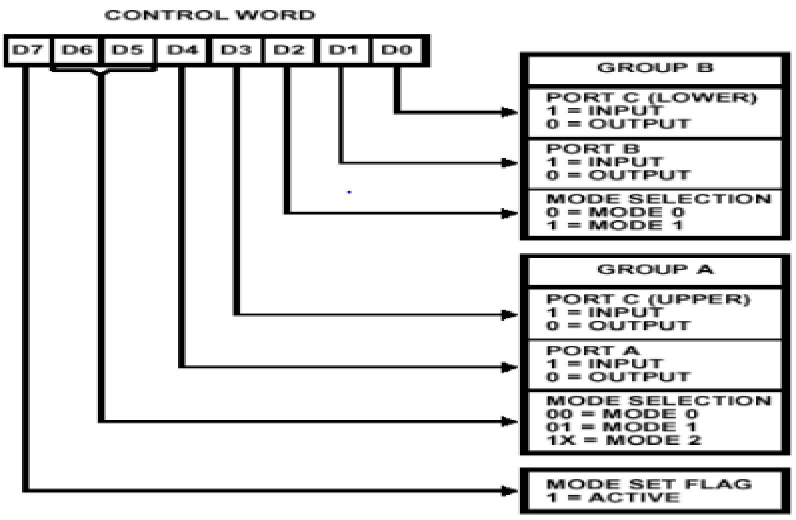


Figure : Block diagram of 8255


CONTROL WORD
D7
D6
D5
D4
D3
D2
D1
D0
GROUP B
PORT C (LOWER) 1 = INPUT 0 = OUTPUT
PORT B 1 = INPUT 0 = OUTPUT
MODE SELECTION 0 = MODE 0 1 = MODE 1
GROUP A
PORT C (UPPER) 1 = INPUT 0 = OUTPUT
PORT A 1 = INPUT 0 = OUTPUT
MODE SELECTION 00 = MODE 0 01 = MODE 1 1X = MODE 2
MODE SET FLAG 1 = ACTIVE

STEPPER MOTOR INTERFACING

A stepper motor is stepped from one position to the next by changing the currents through the fields in the motor. The two common field connections are referred to as two phase or four phase. There are three main areas of applications for stepper motor.

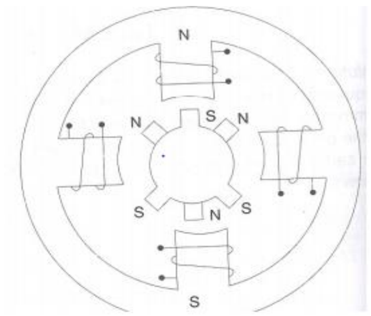
i. Instrumentation
ii. Computer peripherals
iii. Machine drives.

They are used in floppy drives, dot-matrix printers, X-Y plotters, digital watches, etc to rotate things in steps of small angles. The step size in typical stepper motor varies from 0.9deg to 30deg.

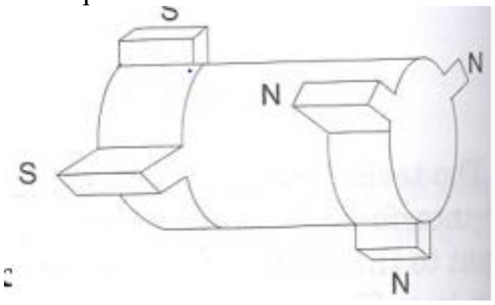
A stepper motor is a device used to obtain an accurate position control of rotating shafts. A stepper motor employs rotation of its shaft in terms of steps, rather than continuous rotation as in

case of AC or DC motors. To rotate the shaft of the stepper motor, a sequence of pulses is needed to be applied to the windings of the stepper motor, in proper sequence. The number of pulses required for one complete rotation of the shaft of the stepper motor are equal to its number of internal teeth on its rotor. The stator teeth and the rotor teeth lock with each other to fix a position of the shaft. With a pulse applied to the winding input, the rotor rotates by one teeth position or an angle x. The angle x may be calculated as.
x =360° /no. of rotor teeth

After the rotation of the shaft through angle x the rotor locks itself with the next tooth in the sequence on the internal surface of stator. The internal schematic of a typical stepper motor with four windings is shown in Figure.



The stepper motors have been designed to work with digital circuits. Binary level pulses of 0-5V are required at its winding inputs to obtain the rotation of shafts. The sequence of the pulses can be decided, depending upon the required motion of the shaft.



excitation Sequences of a Stepper Motor Using Wave Switching Scheme

Motion	Step	A	B	C	D
Clockwise	1	1	0	0	0
	2	0	1	0	0
	3	0	0	1	0
	4	0	0	0	1
	5	1	0	0	0
Anticlock wise	1	1	0	0	0
	2	0	0	0	1
	3	0	0	1	0
	4	0	1	0	0
	5	1	0	0	0

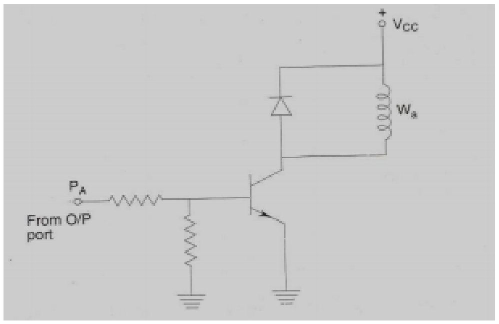


Figure: Interfacing Stepper Motor winding Wa.

Sample Code :

```
MOV  AX,0000H
MOV  DX,0FFE6H
MOV  AL,80H
OUT  DX,AL
MOV  DX,0FFE0H
MOV  AL,88H
OUT  DX,AL
CALL  2086
RCR   AL,1
JMP 200E
```

AT LOCATION 2086:

```
MOV CX,800H
LOOP 2089
RET
```

Experiment 12

Aim: Perform interfacing of DAC

Objective: Perform interfacing of DAC

Theory:

8086 TRAINER KIT

PS-TRAINER-8086A microprocessor trainer kit is proposed to smooth the progress of learning and developing designs of microprocessor from Intel. It has the facility to connect PC's 101/104 Keyboard, to enter user programs in Assembly languages. User verifies the programs through LCD or PC. User friendly Firmware confirms facilitating the beginners learns operations of a microprocessor quickly.

DAC

Fig 1 shows DAC0800 series are monolithic 8-bit high-speed current output digital-to-analog converters (DAC) featuring typical settling times of 100 ns.. The noise immune inputs will accept variety of logic levels. The performance and characteristics of the device are essentially unchanged over the ±4.5V to ±18V power supply range and power consumption at only 33 mW with ±5V supplies is independent of logic input levels.

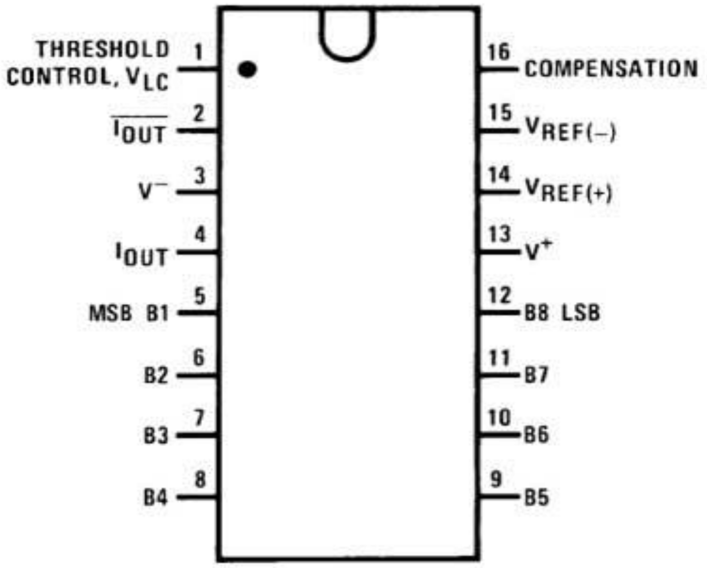


Fig 1: Pin diagram of DAC 0800

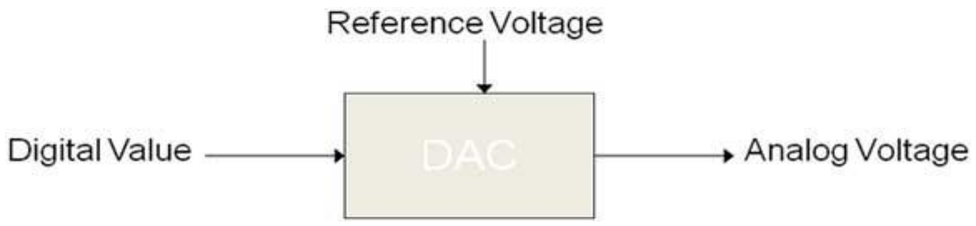
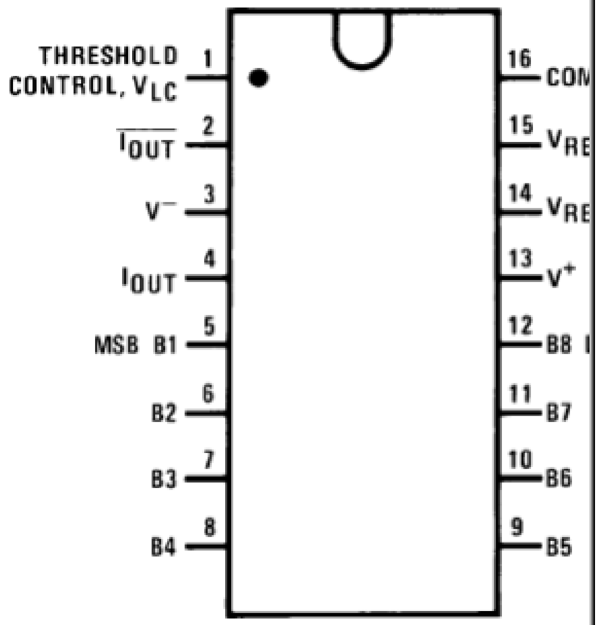


Fig. 2: How a DAC works

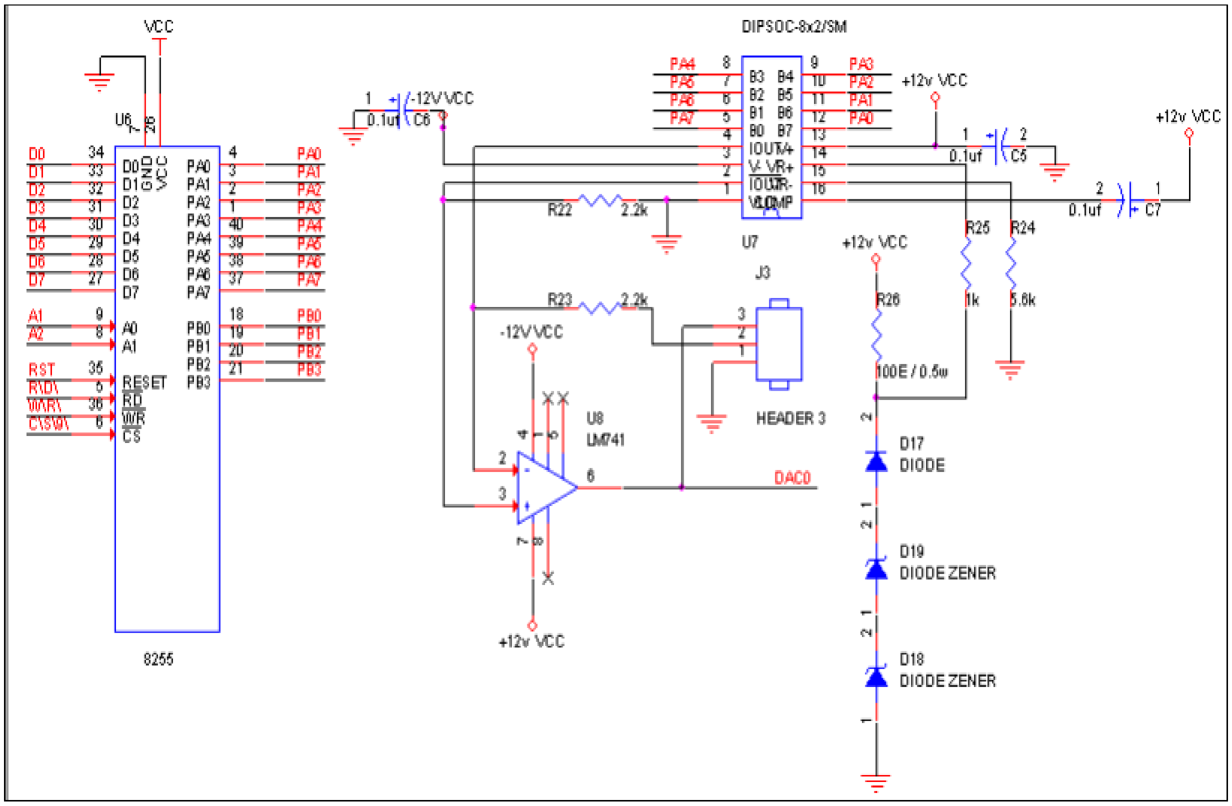
INTERFACING DAC WITH 8086

We now want to we now want to convert the Digital signal to analog voltage by using 8086 trainer board. Here we are using DAC 0800. The DAC 0800 consists of 8 data lines and REF voltage lines. When the DAC is given the digital input it converts the Digital data to corresponding current, to convert the I to V we use UA 741.

PIN ASSIGNMENT WITH PIN ASSIGNMENT WITH 8086

	8086	DAC 0800	DAC 0800 PIN DIAGRAM
REF VOLTAGE LINES	**V+** Line is given to **+12V** regulator(**7812**) **V-** Line is given to **-12V** regulator(**7912**) **78** series in the Regulator IC indicates + voltage **79** series in the Regulator IC indicates - voltage		 THRESHOLD CONTROL, V_{LC} 1 2 $\overline{I_{OUT}}$ 3 V^- 4 I_{OUT} 5 MSB B1 6 B2 7 B3 8 B4 16 COM 15 VRE 14 VRE 13 V^+ 12 B8 11 B7 10 B6 9 B5
DAC – DATA LINES	PA.0	D0	
	PA.1	D1	
	PA.2	D2	
	PA.3	D3	
	PA.4	D4	
	PA.5	D5	
	PA.6	D6	
	PA.7	D7	
VCC	40	16	Supply drawn from MCU/MPU/
GND	20	3	

CIRCUIT DIAGRAM TO INTERFACE DAC WITH 8086



ASSEMBLY PROGRAM TO INTERFACE DAC WITH 8086

MEMORY ADDRESS	OPCODE	MNEMONICS
1100	B0 80	MOV AL, 80H
1102	BA 36 FF	MOV DX,FF36
1105	EE	OUT DX, AL
1106	B0 00	**START:** MOV AL,00
1108	BA 30 FF	MOV DX, FF30
110B	EE	**LOOP1** :OUT DX,AL
110C	FE C0	INC AL
110E	75 FB	JNZ **LOOP1**
1110	EB F4	JMP **START**

Experiment 13

Aim: Case study of RISC

Theory:
INTRODUCTION:
During the 1980's, a new technology emerged in the computer industry – the Reduced Instruction Set Computer (RISC). This instruction set architecture surfaced as an alternative to the older, established technology already in the field – the Complex Instruction Set Computer (CISC).
RISC was presented as a solution to derive more instruction set power out of a computer. It is not just a means to reduce the instruction set, but it is a way of significantly enhancing the performance of a system while having minimal costs compared to that of a CISC implementation.
In 1980, the paper published by Patterson and Ditzel laid down the arguments in favor of RISC over CISC which started the RISC versus CISC debate that lasted for years.
Before we delve into the arguments for and against these architectures, we must first understand the differences between them. The definitions of RISC and of CISC are given in the next few subsections.

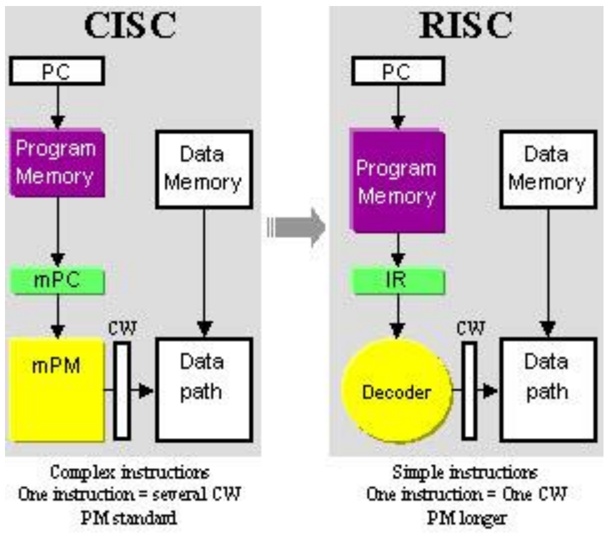
WHAT IS CISC?
The CISC design uses complex instructions intended for direct implementation of high-level language operations. These complex instructions that take time to be executed are made up of smaller, simpler instructions.
CISC microprocessors can be described as the opposite of RISC having these characteristics:

- Multi-cycle operation due to variable execution time of instructions
- Not a load/store machine
- Relies on micro-coded control engine
- More instructions and addressing modes like that of a high-level programming language
- Variable-length instruction format dependent on the number of operands and addressing modes used
- Difficult targets for optimizing compilers because of complex instructions

WHAT IS RISC?
RISC is a microprocessor design approach using simple instructions. Though it may seem less effective for a computational task to be executed with many simple instructions rather than a few complex instructions, the simple instructions take fairly the same amount of time to be performed, making them ideal for pipelining.
The other elements that define a RISC processor, according to, are the following:

- Single-cycle operation aids in fast execution of simple functions
- Load/Store architecture implemented due to the desired single-cycle operation
- Hardwired control provides for the fastest possible single-cycle operation
- Relatively few instructions and addressing modes makes it easy for the control unit to interpret instructions
- Fixed instruction format makes decoding of instructions fast and easy

- More compile-time effort offers opportunity to explicitly move static run-time complexity into compiler



ARGUMENTS AND COUNTER-ARGUMENTS

Having seen the difference between the two architectures, this section now presents the arguments made for each in different aspects of computer design.

Table below shows the summarized list of arguments for RISC and against CISC from. Notice that the arguments for RISC revolve around the central idea of it being a simple design. From this simple design spawns advantages that the CISC architecture would not be able to obtain due to its complexity.

ARGUMENTS FOR RISC and CISC

Factor	RISC	CISC
Implementation Feasibility	Simple architecture makes RISC being realizable at earlier date	Complex architecture make it difficult to implement on a single chip with the design rules at that time
Design Time	Design that can be easier to design and debug can use much superior technology than design that takes a long time to implement	Lengthened design time may lead to problems such as having a machine with an old technology, or trying to predict future technology and have a go at attempting to build that technology
Speed	Gains in speed from better use of chip area and from simple design – with less instructions and addressing modes, these would lead to a less complicated control structure	More instructions and addressing modes result to complicated control structure
Use of Chip Area	Area left on chip due to simple design may be used to make RISC improve performance even more	Complex design leaves no room on chip for enhancements

THE COUNTER-ARGUMENTS:

These are counter-arguments made against the RISC architecture.

- **False Dichotomy.** Complexity of a machine cannot be measured by instruction count alone, and cannot be the only decisive factor for categorizing a machine as RISC or CISC
- **Lengthy Design Time.** Designing products for a high-volume manufacturing company has time-consuming processes that do not exist in a small company. There are differences in the design environments for academic research and from companies that produce commercial products.
- **Language Multiplicity.** An instruction set that supports only a particular language can make the implementation of a different language difficult. Designing a machine for a particular language is different from designing one that accepts a wide range of languages.
- **Instruction Execution Time.** Frequency of execution of instruction must not be the focus of attention, but also to the length of time an instruction is executed.
- **Micro benchmarks.** Small programs (e.g. Towers of Hanoi, Fibonacci) that have been used to measure performance on RISC machines have a narrow scope on functions, although they execute millions of instructions. These benchmarks do not represent actual applications used in the industry
- **Specialized Hardware.** Higher-level hardware-software interface has the advantage of possible use of specialized hardware for better performance.
- **Chip Usage.** Chip area left by RISC's simple design can indeed be used by adding enhancements for better performance. But these enhancements are not inherent in RISC architectures.
- **Ignored Operating System.** The system is not only made of software, hardware, and application code. The operating system overhead and needs have been given too little attention in RISC research.
- **More Instructions.** Breaking up complex instructions of CISC results to more instructions per program.

According to these rebuttals, some arguments for RISC are misleading – for example the design time and chip usage – and therefore cannot be used as basis for comparison of the two architectures. Also, other areas of interest must be properly investigated to make an accurate comparison between the two, like that of the operating system of a machine. Reference stated that with just a paper design and lack of metrics to back the arguments, the claim that the RISC architecture can be superior over CISC will be difficult to prove.

PERFORMANCE COMPARISON

Bhandarkar and Clark published a paper in 1991 that did performance comparisons on implementations from the RISC and CISC architectures. They chose the MIPS M/2000 from the RISC architecture and Digital VAX 8700 from CISC since these machines possess organizational similarity. Nine out of ten SPEC89 benchmarks were run on both machines, and the results were noted down and analyzed.

The results show that the MIPS M/2000 had more instructions than that of VAX 8700, but achieved much less average CPI on all benchmarks. Also, analysis of results reveals that MIPS

M/2000 has a net advantage, and this advantage are attributed to architectural factors like filled delay slots and the number of registers.
Based on the results, they concluded that the MIPS has an advantage in processor performance over VAX with a comparable architecture.
In 1994, another paper was published by Bhandarkar comparing various design aspects of two machines from each architecture – the Alpha 21164 from RISC, and the Intel Pentium® Pro processor. Results show that the Alpha 21164 had better performance, performing over two times faster in floating point benchmarks, and a little better in performance in integer and transaction processing workloads than the Pentium® Pro.
The performed experiments for comparison did not only demonstrate that RISC machines can be at par with CISC, but may even outdo them in performance.

INDUSTRY REACTION

While the RISC versus CISC debate continued over the years, many companies have adopted the RISC design and produced a number of successful processors based on RISC designs. These include the ARM family of RISC processors, the IBM PowerPC family of RISC processors, Sun’s SPARC and UltraSparc, MIPS, and HP’s PA-RISC.
The CISC-based design was also continued to be supported and processors under the said architecture are the Motorola 68000 family, Intel processors, and the VAX family.
Processors with characteristics from both architectures have also materialized, as “Crisp” processors like Intel 80x86 processors and the Motorola 80486 and 68040 were designed to have RISC features. Also, advancements in technology such as simultaneous multi-threading and value prediction are used by machines under both architectures, and because the architectures have adopted the strategies of the other that the architecture classification becomes blurred.

Properties of RISC:
1. Small and simple instructions
2. In RISC Instructions are execute in one clock cycle per Instructions
3. All instructions have the same length
4. Load and Store architecture implemented due to the desired single-cycle operation
5. Have Pipelining
6. More register than CISC
7. Optimal compilation speed as compared to CISC
8. Emphasis on software
9. Compare to CISC a RISC Spends more transistors on memory registers

Advantages of RISC
- Implementation with simple instructions provides many advantages over implementing as compared to CISC Processors.
- Simple instruction set allow for pipeline superscalar designing RISC processor often achieved two to four times performance of CISC processors using [21-27] comparable semiconductor technology and similar clock rates.
- Simple hardware.

- Because instructions set of a (RISC) processor is so simple, it uses up much less chips spaces and extra functions i.e. memory management unit or floating point arithmetic units, can also be placed on the similar chip.
- Smaller chips allows a semiconductor manufacturers to placed more parts on single silicon wafer which can lower per chips cost dramatically and have short design cycles.
- Since RISC processors are simpler than corresponding CISC processors they can be design more quickly and take advantage of other technological [28-33] developments sooner than corresponds CISC design leading to great leaps in performance between generations.

CONCLUSION:

The RISC design had advantages that result to a machine's excellent performance that it has been adopted for commercial products, as we can see from the industry's response of designing RISC-based processors. The long RISC versus CISC debate was most certainly a good thing as it allowed the industry to explore and come up with solutions that have significantly raised the machine's performance we use today. Due to this exploration, both architectures have continuously developed and processors today have a little bit of something from each architecture.

Experiment 14

Aim: Case study of CISC

Theory:
Introduction
The Microprocessor chips are divided into two categories. In both of these architectures the main purpose is to optimal the performance of the system. Our Research on these two architecture which is complex but we do our best. CISC is stand for complex instruction set computer. Nowadays the PCs mostly uses CISC architecture Like AMD and Intel etc. CISC chips have large and complex instructions. We know that hardware is faster than software so therefore one should make a powerful instruction set which provides programmers with assembly instructions to do a lot with short program. In common CISC chips are relatively slow (as compared to RISC). RISC is stand for Reduced Instruction Set Computer. Nowadays mostly Mobile Phones Based on RISC architecture Like MIPS and ARM etc. RISC has simple and small Instruction. RISC chips Comes around the mid 80’s because the reaction of CISC chips. The philosophy behind that almost no one use complex instructions and mostly people uses compilers which never use complex instructions. So for Apple uses RISC chips. So therefore simple and faster instructions are better than large complex and slower (CISC) instructions. However, RISC required more instruction to complete a task than CISC. An advantage of RISC is that because it more simple instructions. RISC chips require less transistors which makes easier to design and cheaper to produce. So now it easier to write powerful optimal compilers since fewer instructions exists.

Properties of CISC:
1. Some simple and very complex instructions
2. In CISC instructions take more than 1 clock per Cycle to execute
3. Variable size instructions
4. No pipelining
5. Few registers
6. Not a load and store machine
7. For Compilation not so good in term of speed
8. Emphasis of Hardware
9. Transistors are used for storing complex instructions

Advantages of CISC
- At the time of their initial development CISC machines use technologies to optimize the performance of a computer.
- Microprogramming is easy as assembly Programming language to implement and less expensive than hardwiring a control unit.
- The ease of micro coding newly instructions allows designers to make (CISC) machines upwardly compatible new computer run the same programs as early computers because the new computers would contained a superset of instructions of earlier computers.
- As each instruction became more capable less instruction used to implement the given task. This made efficient uses of the relative slow main memory.

- Because micro program instructions set can be write to match the construct of high level languages the compilers doesn't have to be as complicated.

Examples of RISC and CISC processors are shown in table below:

CISC	**RISC**
IBM 370-169	MIPS R2000
VAX 11-780	SUN SPARC
MICROVAX 2	INTEL i860
INTEL 80386	MOTOROLOA 8800
INTEL 80286	POWERPC 601

Conclusion: Thus we have successfully studied CISC