Assembling, Linking and Executing

1) Assembling:

- Assembling converts source program into object program if syntactically correct and generates an intermediate **.obj** file or module.
- It calculates the offset address for every data item in data segment and every instruction in code segment.
- A header is created which contains the incomplete address in front of the generated **obj** module during the assembling.
- Assembler complains about the syntax error if any and does not generate the object module.
- Assembler creates **.obj .lst** and **.crf** files and last two are optional files that can be created at run time.

Assembler Types:

a) One pass assembler:

- This assembler scans the assembly language program once and converts to object code at the same time.
- Works fine with backward referencing
- Problem with forward referencing

Backward referencing Forward Referencing

```
L1:……..          JNZ L2
  ……             ……
  JNZ L1         L2:……..
```

b) Two pass assembler

- This type of assembler scans the assembly language twice.
- First pass generates symbol table of names and labels used in the program and calculates their relative address.
- This table can be seen at the end of the list file and here user need not define anything.
- Second pass uses the table constructed in first pass and completes the object code creation.
- This assembler is more efficient and easier than earlier.

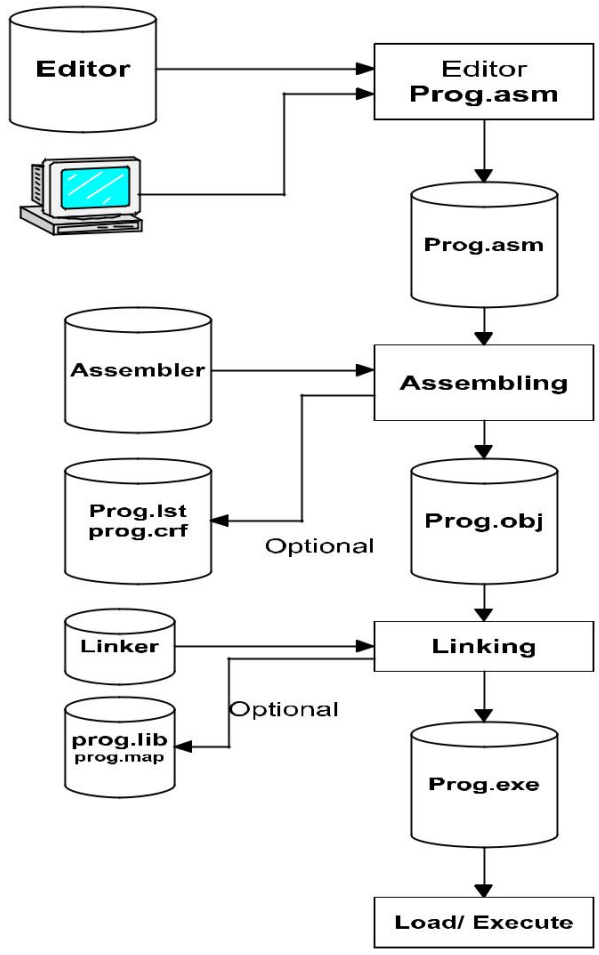


Fig: Steps in assembling, linking & Executing

2) Linking:

- This involves the converting of **.OBJ** module into **.EXE**(executable) module i.e. executable machine code.
- It completes the address left by the assembler.
- It combines separately assembled object files.
- Linking creates **.EXE**, **.LIB**, **.MAP** files among which last two are optional files.

3) Loading and Executing:

- It Loads the program in memory for execution.
- It resolves remaining address.
- This process creates the program segment prefix (PSP) before loading.
- It executes to generate the result.

Sample program → assembling → object Program → linking → executable program

Writing .COM programs:

- It fits for memory resident programs.
- Code size limited to 64K.
- **.com** combines PSP, CS, DS in the same segment
- SP is kept at the end of the segment (FFFF), if 64k is not enough, DOS Places stack at the end of the memory.
- The advantage of **.com** program is that they are smaller than **.exe** program.
- A program written as **.com** requires ORG 100H immediately following the code segment's SEGMENT statement. The statement sets the offset address to the beginning of execution following the PSP.

```
.MODEL TINY
.CODE
ORG 100H                        ; start at end of PSP
BEGIN:JMP MAIN                  ;Jump Past data
A1 DW215
B1DW125
C1DW ?
MAIN PROC
MOV AX, A1
ADD AX, B1
MOV C1, AX
MOV AX, 4C00H
INT 21H
MAIN ENDP
END BEGIN
```

Macro Assembler:

- A macro is an instruction sequence that appears repeatedly in a program assigned with a specific name.
- The macro assembler replaces a macro name with the appropriate instruction sequence each time it encounters a macro name.
- When same instruction sequence is to be executed repeatedly, macro assemblers allow the macro name to be typed instead of all instructions provided the macro is defined.
- Macro are useful for the following purposes:

oTo simplify and reduce the amount of repetitive coding.
oTo reduce errors caused by repetitive coding.
OTo make an assembly language program more readable.
O Macro executes faster because there is no need to call and return.
O Basic format of macro definition:

```
Macro nameMACRO [Parameter list]
……..
……..
[Instructions]
……..
……..
ENDM
```

E.g.

```
Addition  MACRO

IN AX, PORT
ADD AX, BX
OUT PORT, AX
ENDM
```

Instructions

1) Data Transfer Instructions

- **MOV**

  MOV AX,BX
  MOV AL,CH
  MOV CX,2050H
  MOV BH,33H

- **XCHG**

  XCHG BH, CL
  XCHG AX, DX

- **PUSH and POP**

  PUSH BX POP BX
  PUSH DS POP DS
  Note: POP CS is illegal

- **LEA: Load Effective Address**

  - Loads effective address of operand into specified register
    LEA AX, data1
  Alternative Instruction:
  MOV AX, offset data1

2) Arithmetic Instructions

- **ADD:-** Adds byte+byte OR word+word
  ADC:-Adds byte+byte+CY OR Adds word+word+CY
  Flags affected: AF, CF, OF, PF, SF, ZF
  E.g.

ADD AL, 25H;	AL <- AL+25H
ADD BL, AH;	BL <- BL+AH
ADD DX, BX;	DX <- DX+BX
ADD CH, [2050H];	CH<- CH+(byte from 2050H)
ADC AL, DL;	AL<- AL+DL+CY
ADD data1, AL;	data1<- data1+AL

  ADD data1, data2 ; (illegal: two variables cannot be added directly)
- **SUB:-** Subtracts byte-byte OR word-word
  SBB:-Subtracts byte-byte-CY OR Adds word-word-CY
  Flags affected: AF, CF, OF, PF, SF, ZF
  E.g.

SUB CX, BX;	CX <- CX-BX
SBB DH, AL;	DH <- DH-AL-CY
SUB CX, 2356H;	CX <- CX-2356H
SBB DX, [3427H];	DX <- DX-(word from 3427H)-CY

- **INC & DEC**

  INC reg; reg<-reg+1
  DEC reg; reg<-reg-1
  Flags affected: AF, OF, PF, SF, ZF
  E.g.

INC AH	DEC BL
INC BX	DEC DX

- **MUL & IMUL**
  MUL: Multiplies an unsigned byte from source times an unsigned byte in AL register or an unsigned word from source times an unsigned word in AX.
  In byte multiplication, result is kept in AX and in word multiplication, result is kept in DX:AX.
  IMUL: Same as **MUL,** except it does for signed value.
  E.g.
  MUL BL; AX <- AL*BL
  MUL CX; DX:AX<- AX*CX
  IMUL AH; AX <- AL*AH
  Flags affected: CF, OF; AF, PF, SF and ZF are undefined
- **DIV & IDIV**
  DIV: Divides unsigned word by byte or unsigned double word by word.
  When word is divided by byte, word must be in AX register and divisor is given in instruction.
  After division, AL<- quotient, AH<- remainder
  When double word is divided by word, most significant word must be in DX register and least significant word must be in AX register. Divisor is given in instruction.
  After division, AX<- quotient, DX<- remainder
  IDIV: Same as **DIV,** except it does for signed value.
  E.g.
  DIV CX; DX:AX/CX
  IDIV BL; AX/BL
  Flags affected: all flags are undefined
- **NEG: Negate (2's complement)**
  E.g.
  NEG AL
  NEG BX
  Flags affected: AF, OF, PF, SF, CF
- **CMP: Compare Bytes or words**
  E.g.
  CMP CX,AX CMP AL,20H CMP BX, 2050H
  If CX=AX; CF=0, ZF=1, SF=0
  If CX>AX; CF=0, ZF=0, SF=0
  If CX<AX; CF=1, ZF=0, SF=1
  AF, OF, PF are affected according to the result

3) Logical Instructions

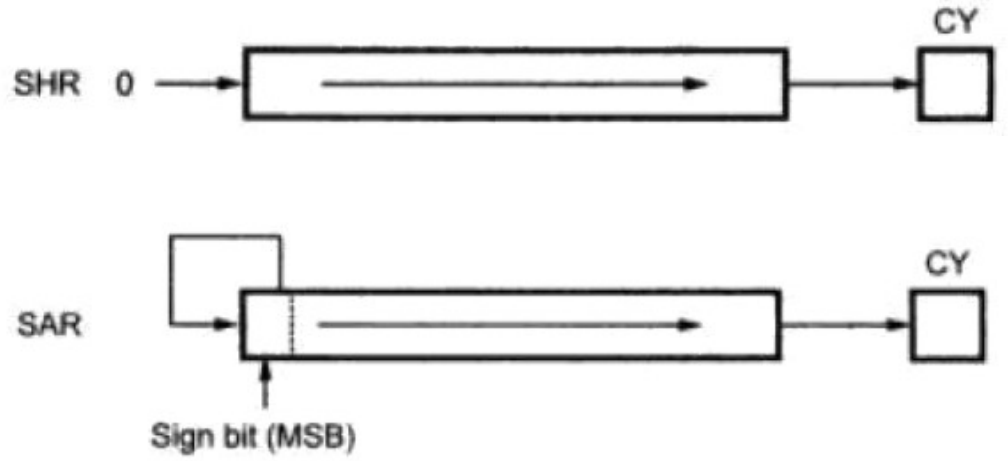
- **NOT – Flags:- No flags affected**
- **AND – Flags:- CF=0, OF=0, PF, SF, ZF according to the result**
- **OR – Flags:- CF=0, OF=0, PF, SF, ZF according to the result**
- **XOR – Flags:- CF=0, OF=0, PF, SF, ZF according to the result**
- **TEST:- AND operation to update flag, but neither operand is changed**

E.g.

NOT BX	AND BH,CL	AND BX,00ffH
OR CH,CL	OR BL,80H,	XOR BP, DI
XOR CL,0BH	TEST AL,BH	TEST CX,0010H

4) Shift Instructions

- **SHR/SAR**
  SHR (Shift Logical Right) -> for unsigned data
  SAR (Shift Arithmetic Right) -> for signed data
  Note: If the no. of shift is more than one, it should be loaded in CL register.



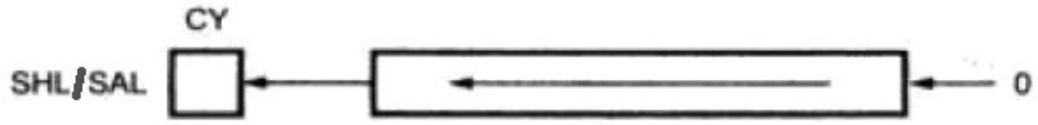
E.g.1

```
MOV BH, 10110111B
SHR BH, 01          ;BH=01011011, CY=1
MOV CL, 02
SHR BH, CL          ;BH=00101101, CY=1; 1st shift
                    ;BH=00010110, CY=1; 2nd shift
```

E.g.2

```
MOV BH, 00110111B
SAR BH, 01          BH=00011011,CY=1
MOV CL, 02
SAR BH, CL          BH=00001101, CY=1; 1st shift
                    BH=00000110, CY=1; 2nd shift
```

- **SHL/SAL**
  SHL (Shift Logical Left) -> for unsigned data
  SAL (Shift Arithmetic Left) -> for signed data
  (Both instructions do same operation)
  Note: If the no. of shift is more than one, it should be loaded in CL register.

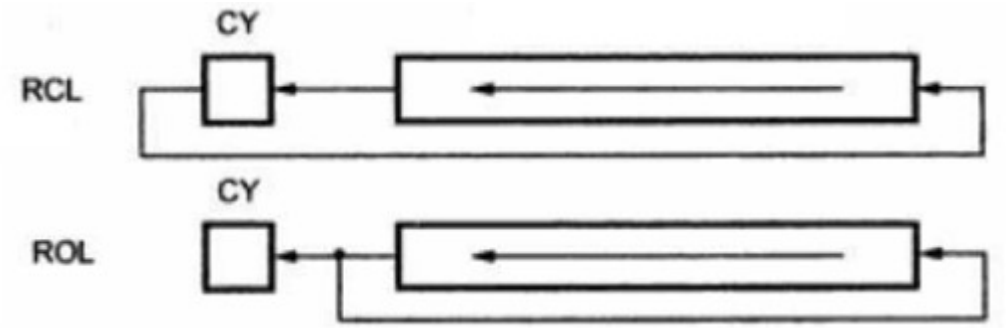


E.g.

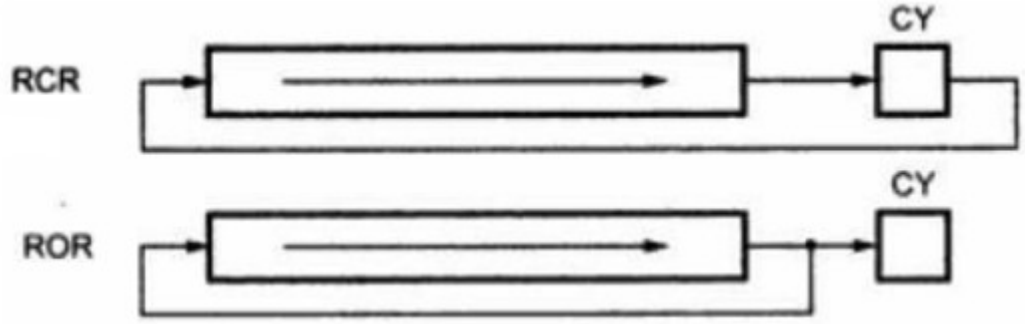
```
MOV BH, 00000101B
SHL BH, 01          BH=00001010, CY=0
MOV CL, 02
SAL BH, CL          BH=00010100, CY=0; 1st shift
                    BH=00101000, CY=0; 2nd shift
```

5) Rotate Instructions

- **ROL/RCL (Rotate left without carry/ Rotate left with carry)**



- **ROR/RCR (Rotate right without carry/ Rotate right with carry)**



E.g.1

```
MOV AL, 10100101B
ROL AL, 01      ;AL=01001011, CY=1
MOV CL, 02
ROL AL, CL      ;AL=10010110, CY=0; 1st rotate
                ;AL=00101101, CY=1; 2nd rotate
```

E.g.2

```
MOV AL, 10100101B
ROR AL, 01      ;AL=11010010, CY=1
MOV CL, 02
ROR AL, CL      ;AL=01101001, CY=0; 1st rotate
                ;AL=10110100, CY=1; 2nd rotate
```

6) Unconditional Transfer

- **Call:** Call a procedure (sub-program), save return address on stack
  E.g.
  Call convert Call BX
- **RET:** Return from procedure to calling program
  Near Call: within same segment
  Far Call: For different segment
- **JMP:** Goto specified address to get next instruction
  E.g.
  JMP UP JMP PASS JMP BX

7) Conditional Transfer Instructions

- **JE/JZ:-** Jump if equal/Jump if zero (Z=1)
- **JNE/JNZ:-** Jump if not equal/Jump if not zero (Z=0)
- **JC:-** Jump if carry (CY=1)
- **JNC:-** Jump if no carry (CY=0)
- **JO:-** Jump if overflow (OF=1)
- **JNO:-** Jump if not overflow (OF=0)
- **JP/JPE:-** Jump if parity/Jump if parity even (PF=1)
- **JNP/JPO:-** Jump if not parity/Jump if parity odd (PF=0)
- **JS:-** Jump if sign (SF=1)
- **JNS:-** Jump if no sign (SF=0)
- **JCXZ:-** Jump if CX is zero

8) Iteration Control Instructions

- **LOOP:-** loop through a sequence of instructions until CX=0
  Each time, loop instruction executes, CX is automatically decremented by 1. If CX is not zero, execution will jump to a destination specified by a label in the instruction otherwise simply goes to next instruction after loop.
  E.g.

```
        MOV AL, 20
        MOV CX, 0040
UP:     ADD AL, 2
        MOV DH, AL
        LOOP UP
```

- **LOOPE/LOOPZ:** loop through a sequence of instructions if ZF=1 and CX $\neq 0$
- **LOOPNE/LOOPNZ:** loop through a sequence of instructions if ZF=0 and CX $\neq 0$

9) Processor Control Instructions

- **STC:-** Set carry flag
- **CLC:-** Clear carry flag
- **CMC:-** Complement carry flag
- **STD:-** Set direction flag

- **CLD:-** Clear direction flag
- **STI:-** Set Interrupt flag
- **CLI:-** Clear Interrupt flag

Programs

Title to find sum of an array

```
dosseg
.model small
.stack 64H
.data
        array db 01H, 05H, 7AH, 2BH, 25H
        sum db ?
.code
        main proc
        mov ax, @data
        mov ds, ax
        mov si, 0000H
        mov al, 00H
        mov cx, 0005H
up:     add al, array[si]
        inc si
        loop up
        mov ax, 4c00H
        int 21H
        main endp
end
```

Title program to sort array

```
dosseg
.model small
.stack 64h
.data
        buf1 db 01,03,05,04,02,07
.code
        main proc
        mov ax,@data
        mov ds,ax
        mov si,0000h
l4:     cmp si,05h
        jz l5
        mov di,si
        inc di
l2:     cmp di,06h
        jz l6
        mov ah,buf1[di]
        cmp buf1[si],ah
        jnc l3
        mov ah,buf1[di]
        xchg buf1[si],ah
        mov buf1[di],ah
l3:     inc di
        jmp l2
l6:     incsi
        jmp l4
l5:     mov ax,4c00h
        int 21h
        main endp
end
```

Software Interrupt

Software interrupt is call to a subroutine located in the operating system. The common software interrupts used here are INT 10H for video services and INT 21H for DOS services.

INT 21H (DOS Servies):

87 different functions supported by this interrupt, specified by a function number placed in AH register.

Function No.	Description
# 00H-	**It terminates the current program.** Generally not used, function 4CH is used instead.
# 01H-	**Console(character) input with echo** Character read is returned in AL in ASCII value

# 02H-	**Display single character** 	
	Sends the characters in DL to display
	MOV AH, 02H
	MOV DL, 'A' ; move Dl, 65
	INT 21H

# 03H and 04H-	**Auxiliary input/output** INT 14H is preferred.
# 05H-	**Printer service** Sends the character in DL to printer
# 06H-	**Direct Console Input** Displays the character in DL.
# 07H-	**Console input without echo (doesn't respond to Ctrl+Break)**
# 08H-	**Console input without echo (responds to Ctrl+Break)**
# 09H-	**string display** Displays string until '$' is reached. DX should have the address of the string to be displayed.
# 0AH-	**Read string** DX points location whose first byte gives the maximum character allowed to enter. The next byte is reserved to store actual no. of character entered and rest for entered character

Max. no. of character allowed	**Actual no. of character**	**Character storage starts from here**			

Str[0] Str[1] Str[2] Str[3]

Character Storage Format

E.g.

```
title program to read string from user and display it
dosseg
.model small
.stack 64H
.data
        buf1 db 30 dup(?)
        buf2 db 0dh,0ah,'$'
```

```
.code
        main proc
        mov ax,@data
        mov ds,ax                   ;initialize data segment register
        mov buf1,30                 ;maximum size of the buffer
        mov dx,offset buf1          ;load offset address in DX register
        mov ah,0ah                  ;Function No. 0AH for string input
        int 21h                     ;Execute the instruction
        mov ah,09h                  ;Function No. 09H for string output
        mov dx,offset buf2          ;load offset in DX register (buf2 contains code for next line)
        int 21h                     ;Execute the instruction
        mov si,buf1[01]             ;2nd byte of buffer contains no. of typed characters
        mov buf1[si+2],'$'          ;load '$' at the end of string
        mov dx,offset buf1[2]       ;load offset address of 3rd byte of buffer
        mov ah,09h                  ;string output
        int 21h                     ;Execute the instruction
        mov ax,4c00h                ;Function No. 4CH to terminate program
        int 21h
        main endp
end
```

INT 10H (Video Servies)

Function No.	**Description**
# **00H-**	**Set Video Mode (also clears screen)**
	AL = display mode
	00H for 40X25 black and white text
	01H for 40X25 color text
	02H for 80X25 black and white text
	03H for 80X25 color text
	(colsXrows)
# **01H-**	**Set cursor shape (size)**
# **02H-**	**Set Cursor position**
	BH =video page
	DH = row (y-co-ordinate)
	DL = column (x-co-ordinate)
# **03H-**	**Read Cursor position**
	Returns: DH= current row (y-co-ordinate)
	DL= current column (x-co-ordinate)
	CH= starting line for cursor
	CL= ending line for cursor
# **04H-**	**Read light pen position**
# **05H-**	**Set active video page**
# **06H-**	**Scroll (Initialize) rectangle window up**
	AL = no. of lines to scroll up

(if AL = zero, entire window if cleared or blanked)
BH = blanked area attributes
CH = y-co-ordinate, upper left corner of window
CL = x-co-ordinate, upper left corner of window
DH = y-co-ordinate, lower right corner of window
DL = x-co-ordinate, lower right corner of window

07H- **Scroll rectangle window down**

08H- **Read character and attribute at cursor**
Returns: AH = attribute, AL = ASCII character code

# **09H-** **Write character and attribute at cursor**
AL = ASCII Character code
BH = Video page
BL = attribute or color
CX = count of character to write (replication factor)

# **0AH-** **Write character only at cursor**
AL = ASCII Character code
BH = Video page
BL = color
CX = count of character to write (replication factor)

Attribute

	Background	Foreground
Attribute:	BL b bb	f f f f
Bit number:	7 6 5 4	3 2 1 0

BL – Blink (1 – enable; 0 – disable)

Background/Foreground
0 – Black
1 – Blue
2 – Green
3 – Cyan
4 – Red
5 – Magenta
6 – Brown
7 – Light Grey

Foreground
8 – Dark Grey
9 – Light Blue
10 – Light Green
11 – Light Cyan
12 – Light Red
13 – Light Magenta
14 – Yellow
15 – White

Programs

Title to print "Test String" with blue background and light green text color in center of screen

```
dosseg
.model small
.data
        msg db 'Test String$'

.code
        main proc
        mov ax,@data
        mov ds,ax
        mov ah,00h;             set video services
        mov al,01h;             set video mode, 40 cols and 25 rows
        int 10h;                execute video function
        mov ah, 02H;            set cursor position
        mov bh, 00H;            set video page
        mov dh, 12;             row no. of new position
        mov dl, 20;             column no. of new position
        int 10h
        mov ah,06h;
        mov bh,1Ah;             blue background and light green text color
        mov ch,0;               y-co-ordinate, upper left corner of window
        mov cl,0;               x-co-ordinate, upper left corner of window
        mov dh,25;              y-co-ordinate, lower right corner of window
        mov dl,40;              x-co-ordinate, lower right corner of window
        int 10h
        mov dx,offset msg;      load offset of string
        mov ah,09h;             string output
        int 21h
        mov ax,4c00h
        int 21h
        main endp
end
```

Title to change uppercase to lowercase

```
dosseg
.model small
.stack 64H
.data
        buf1db 255 dup(?)
        newline db 0dh,0ah,'$'
.code
        main proc
        mov ax,@data
```

```
        mov ds,ax
        mov buf1,50H;                   maximum size of buffer
        mov dx,offset buf1;             load offset to dx register
        mov ah,0ah;                     read string
        int 21h
        mov cl,buf1[1];                 2nd byte contains actual no. of characters
        mov ch,00
label2: mov ah,09h
        lea dx,newline
        int 21h
        mov ah,02h
        mov si,02
loop1:  mov al,buf1[si]
        cmp al,41h;                     ASCII value of A=41H
        jc pass;                        if less than 41H, jump to pass
        cmp al,5bh;                     ASCII value of Z=5AH
        jnc pass;                       if greater than or equals to 5bH, jump to pass
        add al,20h;                     convert to lowercase
        mov buf1[si],al
pass:   mov dl,buf1[si]
        int 21h
        incsi
        loop loop1
        mov ax,4c00h
        int 21h
        main endp
end
```

Title program to count vowel and display the count in clear screen

```
dosseg
.model small
.stack 64H
.data
        str1 db 100 dup(?)
        str2 db 'The no. of vowels is:$'
        newline db 0dh,0ah,'$'
.code
        main proc
        mov ax,@data
        mov ds,ax
        mov str1,30h
        mov dx,offset str1
        mov ah,0ah
        int 21h
```

```
        mov cl,str1[1]
        mov ch,00h
        mov si,cx
        mov str1[si+2],'$';          load $ at end of string
        mov al,00h;                  register to count vowel
        mov si,0002H;
up:     cmp str1[si],'A'
        jz count
        cmp str1[si],'a'
        jz count
        cmp str1[si],'E'
        jz count
        cmp str1[si],'e'
        jz count
        cmp str1[si],'I'
        jz count
        cmp str1[si],'i'
        jz count
        cmp str1[si],'O'
        jz count
        cmp str1[si],'o'
        jz count
        cmp str1[si],'U'
        jz count
        cmp str1[si],'u'
        jz count
        jmp pass
count:  add al,01h
        daa
pass:   inc si
        loop up
        mov ch,al
        mov ah,00h;                  set video mode to clear screen
        mov al,00h;                  40X25 screen
        int 10h
        lea dx,str1[2]
        mov ah,09h
        int 21h
        lea dx,newline;              load offset for newline
        mov ah,09h;                  string output
        int 21h
        lea dx,str2
        mov ah,09h
        int 21h
```

```
        mov ah,02h;             character output
        mov dl,ch
        and dl,0f0h;            extract higher nibble
        mov cl,04h
        ror dl,cl;              rotate 4 times to make higher nibble, lower
        add dl,30h;             convert decimal number into ASCII code
        int 21h
        and ch,0fh;             extract lower nibble
        add ch,30h;             convert decimal number into ASCII code
        mov dl,ch
        int 21h
        mov ax,4c00h
        int 21h
        main endp
end
```

Title to read string, separate words from string, display each word at center of each line of clear screen with blue background and cyan foreground

```
dosseg
.model small
.stack 64H
.data
        str1 db 100 dup(?)
        newline db 0dh,0ah,'$'
.code
        Main proc
        mov ax,@data
        mov ds,ax
        mov str1,30h
        mov dx,offset str1
        mov ah,0ah
        int 21h
        mov ah,09h
        mov dx,offset newline
        int 21h
        mov ah,00h
        mov al,00h
        int 10h
        mov ah,06h
        mov bh,00010011b
        mov cx,0
        mov dh,25
        mov dl,40
        int 10h
```

```
        mov cl,str1[1];         counter, total no. of characters
        mov ch,0
        mov si,02
        mov bl,01h;             row number for cursor to set
up:     mov ah,02h
        mov bh,00
        mov dh,bl
        mov dl,20;              column number for cursor to set
        int 10h
        mov ah,02h
repeat:mov dl,str1[si]
        cmp dl,' ';             compare with space to separate the word
        jnz pass
        inc bl;                 increase the row number when space is found
        dec cx;                 decrease the counter
        incsi;                  increase the index
        jmp up;                 goto up to set the cursor again
pass:   int 21h
        inc si
        loop repeat
        mov ax,4c00h
        int 21h
        main endp
end
```

Title toread a string from user and display only alphabetic characters in a clear screen.

```
dosseg
.model small
.stack 64H
.data
        str1 db 100 dup(?)
        str2 db ‘enter a string$’
        newline db 0dh,0ah,'$'
.code
        main proc
        mov ax,@data
        mov ds,ax
        mov dx, offset str2
        mov ah,09h
        int 21h
        mov str1,30h
        mov dx,offset str1
        mov ah,0ah
        int 21h
        mov ah,09h
```

```
        mov dx,offset newline
        int 21h
        mov ah,00h      ;set video mode for clear screen
        mov al,00h
        int 10h
        mov cl,str1[1]
        mov ch,0
        mov si,02
        mov ah,02h
up:     mov dl,str1[si]
        cmp dl,41h      ;ascii value of A=41h
        jc pass1
        cmp dl,5bh      ;ascii value of Z=5Ah
        jnc pass1
        jmp pass
pass1:  cmp dl,61h      ;ascii value of a=61h
        jc down
        cmp dl,7bh      ;ascii value of z=7Ah
        jnc down
        jmp pass
        dec cx
        inc si
        jmp up
pass:   int 21h
down:   inc si
        loop up
        mov ax,4c00h
        int 21h
        main endp
end
```