Stack

The stack segment of a program is used for temporary storage of data and addresses.

A stack is an 1 dimensional data structure. Items are added and removed from one end of the structure. That is it is processed in a LIFO(Last-In First-Out) manner. The most recent addition to the stack is called top of the stack.

A program must set aside a block of memory to hold the stack by declaring a stack segment

For example, `.stack 100h`

SS will contain the segment number of the stack segment -- **SP** will be initialized to **256** (**100h**). The stack grows from higher memory addresses to lower ones

PUSH and PUSHF

PUSH is used to add a new word to the stack. The syntax is

```
PUSH source
```

Here the source is a 16 bit register or memory word. Example is

```
PUSH AX
```

The execution of PUSH causes the following to happen

1. SP decreased by 2
2. A copy of the source content is moved to the address specified by `SS:SP`

PUSHF instruction has no operand and pushes the flag register onto stack.

Initially, SP contains the offset address of the memory location immediately following the stack segment; the first PUSH decreases SP by 2, making it to point to the last word in the stack segment. Because each PUSH instruction decreases SP, the stack grows toward the beginning of the memory. The figure below shows how stack works.

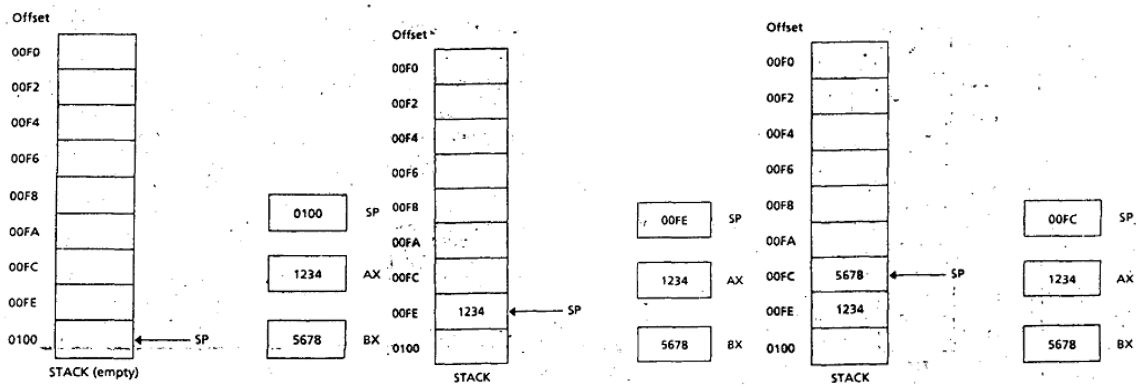


Figure 1 Stack (a) before and (b) after PUSH AX (c) after PUSH BX instruction

POP and POPF

To remove the top item from the stack, we POP it. The syntax is

```
POP destination
```

Here destination is a 16-bit register (Except IP) or memory word. Example of it os

```
POP BX
```

The execution of POP causes this to happen

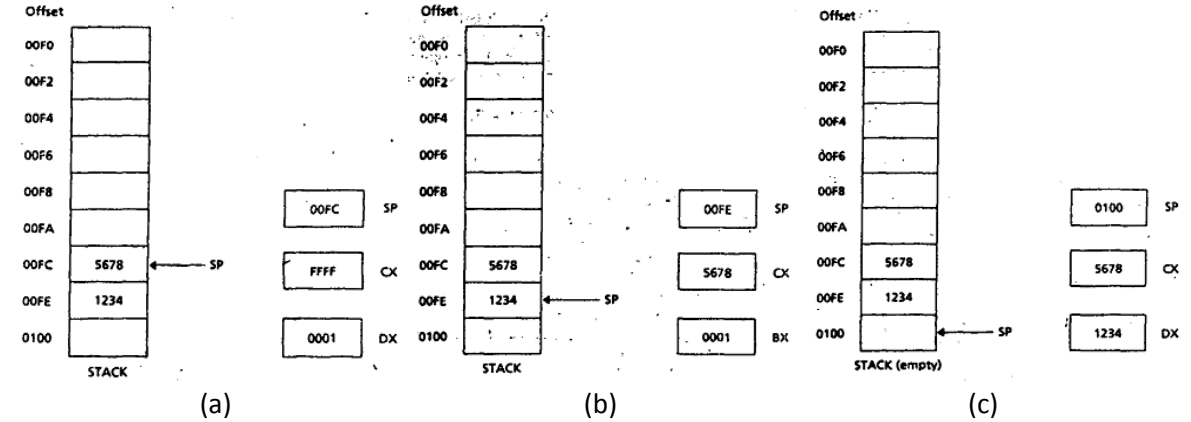
1. The content of SS:SP (THe top of the stack) is moved top destination
2. SP is increased by 2

The instruction POPF pops the top of the stack into FLAGS register.

PUSH, PUSHF, POP, POPF has no effect on the content of the Flags register. PUSH and POP instruction will not work with a byte instruction

PREPARED BY
SHAHADAT HUSSAIN PARVEZ

The figure below shows how POP works.



(a) (b) (c)

Figure 2 Stack (a) before POP, (b) after POP CX, (c) after POP DX

An application of STACK

Because the stack behaves in a last-in, first-out manner, the order that items come off the stack is the reverse of the order they enter.

Algorithm to Reverse Input.

```
Display a '?'
Initialize count to 0
Read a character
WHILE character is not a carriage return DO
                push character onto the stack
                Increment count
                read a character
END WHILE;
Go to a new line
FOR count times DO
pop a character from the stack;
display it;
END_FOR
```

Procedure

Procedure is used to decompose a problem into a series of sub-problems, which works similar to functions in high level languages like C. The syntax of procedure declaration is the following:

```
PROC name type
; body of procedure
ret
ENDP name
```

type can be **NEAR** (in same segment) or **FAR** (in a different segment) -- if omitted, **NEAR** is assumed

The CALL Instruction

CALL invokes a procedure CALL has two forms, ***direct***

```
call name
```

where ***name*** is the name of a procedure, and ***indirect***

```
call address_expression
```
(not generally recommended)

where ***address_expression*** specifies a register or memory location containing the address of a procedure

PREPARED BY
SHAHADAT HUSSAIN PARVEZ

Executing a CALL

- The return address to the calling program (the current value of the IP) is saved on the stack
- IP get the offset address of the first instruction of the procedure (this transfers control to the procedure)
- FAR procedures must process CS:IP instead of just IP

The RET instruction

To return from a procedure, the instruction

```
ret pop_value
```

is executed

The integer argument ***pop_value*** is optional

ret causes the stack to be popped into IP

If ***pop_value*** **N**is specified, it is added to SP -- in effect removes *N* additional bytes from the stack

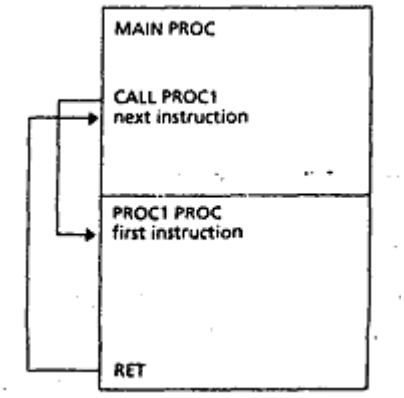


Figure 3 Procedure call and return

Example of procedure

Multiplication algorithm:

```
Product = 0
REPEAT
   IF lsb of B is 1   (Recall lsb = least
              significant bit)
   THEN
      Product = Product + A
   END_IF
   Shift left A
   Shift right B
UNTIL B = 0
```

Array

A one-dimensional array is an ordered list of elements, all of the same type.To define an array in assembly language

```
W dw 10,20,30,40,50,60
```

The address of the array variable is called the base address of the array

If the offset address of the array is 0200h, the array looks like this in memory:

element	offset address	symbolic address	contents
W[0]	0200h	W	10
W[1]	0202h	W+2h	20
W[2]	0204h	W+4h	30
W[3]	0206h	W+6h	40

PREPARED BY
SHAHADAT HUSSAIN PARVEZ

The DUP Operator

- Arrays whose elements share a common initial value are defined using the DUP pseudo-op
- It has the form:

  repeat_count DUP (*value*)
- `gamma dw 100 DUP(0)`
  - sets up an array of 100 words, with each entry initialized to 0
- `delta db 212 DUP(?)`
  - sets up an array of 212 uninitialized bytes

Location of Array Elements

- The address of an array element may be computed by adding a constant to the base address
- If A is an array and S denotes the number of bytes in an element, then the address of element A[i] is A + i*S (assuming zero-based arrays; for one-based arrays it would be A + (i-1)*S)
- To exchange W[9] and W[24] in an word array W:

```
mov  ax,W+18      ; ax has W[9]
xchg W+48,ax      ; ax has W[24]
mov  W+18,ax      ; complete exchange
```

Two-Dimensional Arrays

- A 2D array is an array of arrays
- Usually view them as consisting of rows and columns

Row \ Column	1	2	3	4
1	B[1,1]	B[1,2]	B[1,3]	B[1,4]
2	B[2,1]	B[2,2]	B[2,3]	B[2,4]
3	B[3,1]	B[3,2]	B[3,3]	B[3,4]

Elements may be stored in row-major order

```
B   DW   10,20,30,40
     DW   50,60,70,80
     DW   90,100,110,120
```

or column-major order

```
B   DW   10,50,90
     DW   20,60,100
     DW   30,70,110
     DW   40,80,120
```

Locating an Element in a 2D Array

- A is an *M* x *N* array in row-major order, where the size of the elements is S
- To find the location of A[i,j]
  - find where row i begins
  - find the location of the j^{th} element in that row
- Row 0 begins at location A -- row i begins at location A + i*N*S
- The j^{th} element is stored j*S bytes from the beginning of the row
- So, A[i,j] is in location A+(i*N + j)*S

PREPARED BY
SHAHADAT HUSSAIN PARVEZ

2D Arrays and Based-Indexed Addressing Mode

- Suppose A is a 5x7 word array stored in row-major order. Write code to clear row 2.

```
  mov   bx,28            ; bx indexes row 2
    xor   si,si            ; si will index columns
    mov   cx,7             ; # elements in a row
clear:
    mov   [bx + si + A],0  ; clear A[2,j]
    add   si,2             ; go to next column
    loop  clear            ; loop until done
```

Another Example:

- Write code to clear column 3 -- Since A is a 7 column word array we need to add 2*7 = 14 to get to the next row

```
        mov   si,6             &n bsp; ; si indexes column 3
        xor   bx,bx              ; bx will index rows
        mov   cx,5             &n bsp; ; #elements in a column
clear:
        mov   [bx + si + A],0   ; clear A[i,3]
        add   bx,14             ; go to next row
        loop  clear             ; loop until done
```

String Instructions

- A string is simply an array of bytes or words
- Here are some operations which may be performed with string instructions
    - copy a string into another string
    - search a string for a particular byte or word
    - store characters in a string
    - compare strings of characters alphabetically

The Direction Flag

- One of the control flags in the FLAGS register is the *direction flag* (DF)
- It determines the direction in which string operations will proceed
- The string operations are implemented by the two index registers SI and DI
- If DF = 0, SI and DI proceed in the direction of increasing memory addresses
- If DF = 1, they proceed in decreasing direction

CLD and STD

- To make DF = 0, use the cld instruction

```
cld ;clear direction flag
```

- To make DF = 1, use the std instruction

```
std ;set direction flag
```

- **cld** and **std** have no effect on the other flags

Moving a String

- Suppose we have defined two strings

```
DATASEG
string1 DB "HELLO"
string2 DB 5 DUP (?)
```

- The **movsb** instruction

```
movsb ;move string byte
```


PREPARED BY
SHAHADAT HUSSAIN PARVEZ

- copies the contents of the byte addressed by DS:SI to the byte addressed by ES:DI
- after the byte is moved, both SI and DI are incremented if DF=0; if DF=1, they are decremented

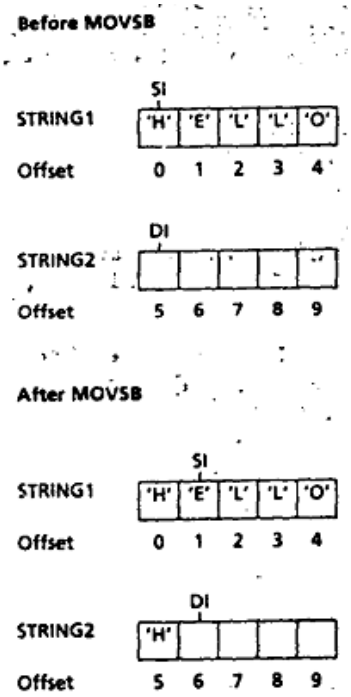


Figure 4 MOVSB

MOVSB example

- To copy the first two bytes of **str1** to **str2**, we use the following instructions:

```
mov ax,@data
mov ds,ax      ;initialize ds
mov es,ax      ; and es
lea si,[str1] ;si points to source string
lea di,[str2] ;di points to dest string
cld            ;set df=0 (increasing)
movsb          ;move first byte
movsb          ;move second byte
```

The REP Prefix

- **movsb** moves only a single byte from the source string to the destination
- To move the entire string, first initialize **cx** to the number *N* of bytes in the source string and execute **rep movsb**
- The **rep** prefix causes **movsb** to be executed *N* times
- After each **movsb**, **cx** is decremented until it becomes 0

REP Example

```
mov ax,@data
mov ds,ax      ;initialize ds
mov es,ax      ; and es
lea si,[str1] ;si points to source string
lea di,[str2] ;di points to dest string
cld            ;set df=0 (increasing)
mov cx,5       ;# of chars in string1
rep movsb      ;copy the string
```

MOVSW

- The word form of movsb is movsw

```
movsw ;move string word
```

- **movsw** moves words rather than bytes
- After the string word has been moved, both **SI** and **DI** are incremented (or decremented) by 2
- Neither **movsb** nor **movsw** have any effect on the flags

The STOSB and STOSW Instructions

```
stosb ;store string byte
```

- Moves the contents of the AL register to the byte addressed by ES:DI
- DI is incremented if DF=0 or decremented if DF=1
- Similarly,

```
stosw ;store string word
```

  - Moves the contents of AX register to the word addressed by ES:DI
  - DI is incremented or decremented by 2
- Neither **stosb** nor **stosw** have any effect on the flags

Code using STOSB

```
mov ax,@data
mov es, ax      ;initialize es
lea di,[str]    ;di points to str
cld             ;process to the right
mov al,'A'      ;al has char to store
stosb           ;store an 'A'
stosb           ;store another one
```

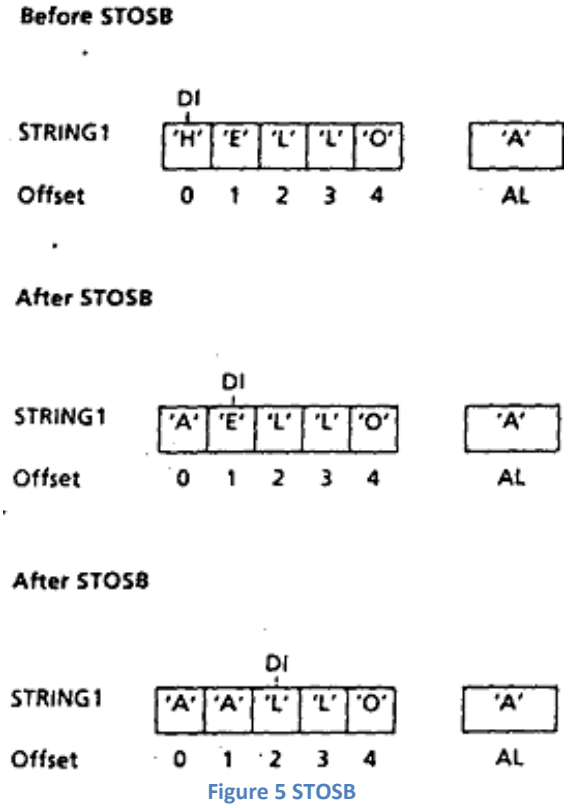


Figure 5 STOSB

PREPARED BY
SHAHADAT HUSSAIN PARVEZ

Reading and Storing a Character String

- Int 21h, function 1 reads a character from the keyboard into AL
- Use interrupt with **stosb** to read a character string

Pseudocode:

```
chars_read = 0
read a character (using int 21h, fcn 1)
while character is not CR do
    if char is BS then
        chars_read = chars_read - 1
        back up in string
    else
        store char in string
        chars_read = chars_read + 1
    endif
    read another character
endwhile
```

Code

```
cld         ;process from left
        xor bx,bx   ;BX holds no. of chars read
        mov ah,1    ;input char function
        int 21h     ;read a char into AL
WHILE1: cmp al,0Dh  ;<CR>?
        je ENDWHLE1 ;yes, exit
    ;if char is backspace
        cmp al,08h  ;is char a backspace?
        jne ELSE1   ;no, store in string
        dec di      ;yes, move string ptr back
        dec bx      ;decrement char counter
        jmp read    ;and go to read another char
ELSE1:  stosb       ;store char in string
        inc bx      ;increment char count
READ:   int 21h     ;read a char into AL
        jmp WHILE1  ;and continue loop
ENDWHLE1:
```

The LODSB Instruction

```
lodsb ;load string byte
```

- Moves the byte addressed by DS:SI into the AL register
- SI is incremented if DF=0 or decremented if DF=1
- Similarly,

```
lodsw ;store string word
```

- Moves the word addressed by DS:SI into the AX register
- SI is incremented or decremented by 2
- Neither **lodsb** nor **lodsw** have any effect on the flags

Code using LODSB

```
            DATASEG
str         DB  'ABC'     ;define string
            CODESEG
            mov ax,@data
            mov ds, ax    ;initialize ds
            lea si,[str]  ;si points to str
            cld           ;process left to right
            lodsb         ;load first byte in al
            lodsb         ;load second byte in al
```

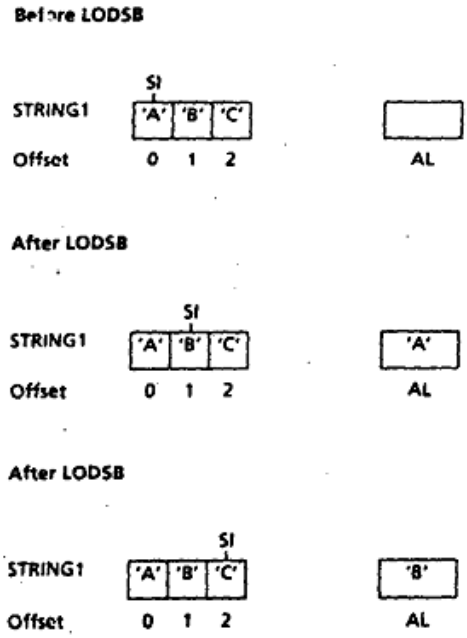


Figure 6 LODSB

Displaying a Character String

- Int 21h, function 2 displays the character in **dl**
- Use interrupt with **lodsb** to display a character string
- Pseudocode:

```
for count times do
    load string character into al
    move it to dl
    output the character
endfor
```

Code to Display a String

```
            cld           ;process from left
            mov cx,number ;cx holds no. of chars
            jcxz ENDFOR   ;exit if none
            mov ah,2      ;display char function
TOP:
            lodsb         ;char in al
            mov dl,al     ;move it to dl
            int 21h       ;display character
            loop TOP      ;loop until done
ENDFOR:
```


PREPARED BY
SHAHADAT HUSSAIN PARVEZ

Scan String

```
scasb      ;scan string byte
```

- examines a string for a target byte (contained in **al**)
- subtracts the string byte pointed to by **es:di** from **al** and sets the flags
- the result is not stored
- **di** is incremented if **df** = 0 or decremented if **df** = 1

SCASW

- **scasw** is the word form of scan string
- The target word is in **ax**
- **di** is incremented or decremented by 2 depending on the value of **df**
- All the status flags are affected by **scasb** and **scasw**

SCASB Example

```
            DATASEG
str         DB    'ABC'      ;define string
            CODESEG
            mov   ax,@data
            mov   es,ax      ;initialize es
            cld              ;process left to right
            lea   di,[str]   ;di points to str
            mov   al,'B'     ;target character
            scasb            ;scan first byte
            scasb            ;scan second byte
```

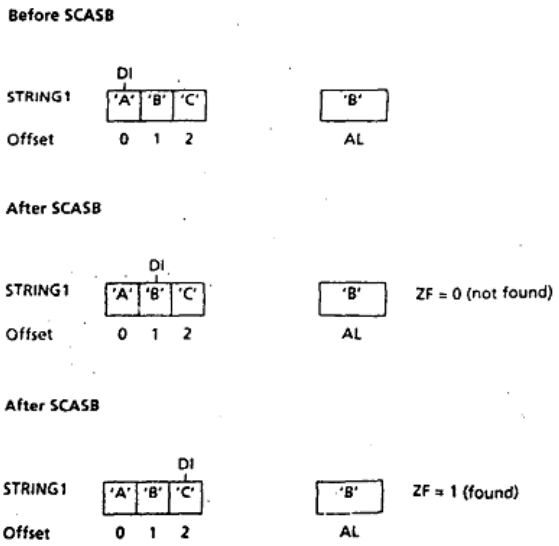


Figure 7 SCASB

REPNE, REPNZ, REPE, and REPZ

- In looking for a target byte, the string is traversed until a match is found or the string ends
- As with **rep**, **cx** is initialized to the length of the string
- **repne scansb** (*repeat while not equal*) will repeatedly subtract each string byte from **al**, update **di**, and decrement **cx** until either the target is found (**zf** = 1) or **cx** = 0
- **repnz** is a synonym for **repne**
- **repe** (*repeat while equal*) repeats a string instruction until **zf** = 0 or **cx** = 0
- **repz** is a synonym for **repe**

Comparing Strings

- The **cmpsb** instruction

```
cmpsb      ;compare string byte
```

- subtracts the byte addressed by **DS**:**SI** from the byte addressed by **ES**:**DI**, sets the flags, and throws the result away
- afterward, both **SI** and **DI** are incremented if **DF**=0; if **DF**=1, they are decremented
- The word version of **cmpsb** is

```
cmpsw      ;compare string word
```

- All status flags are affected by **cmpsb** and **cmpsw**

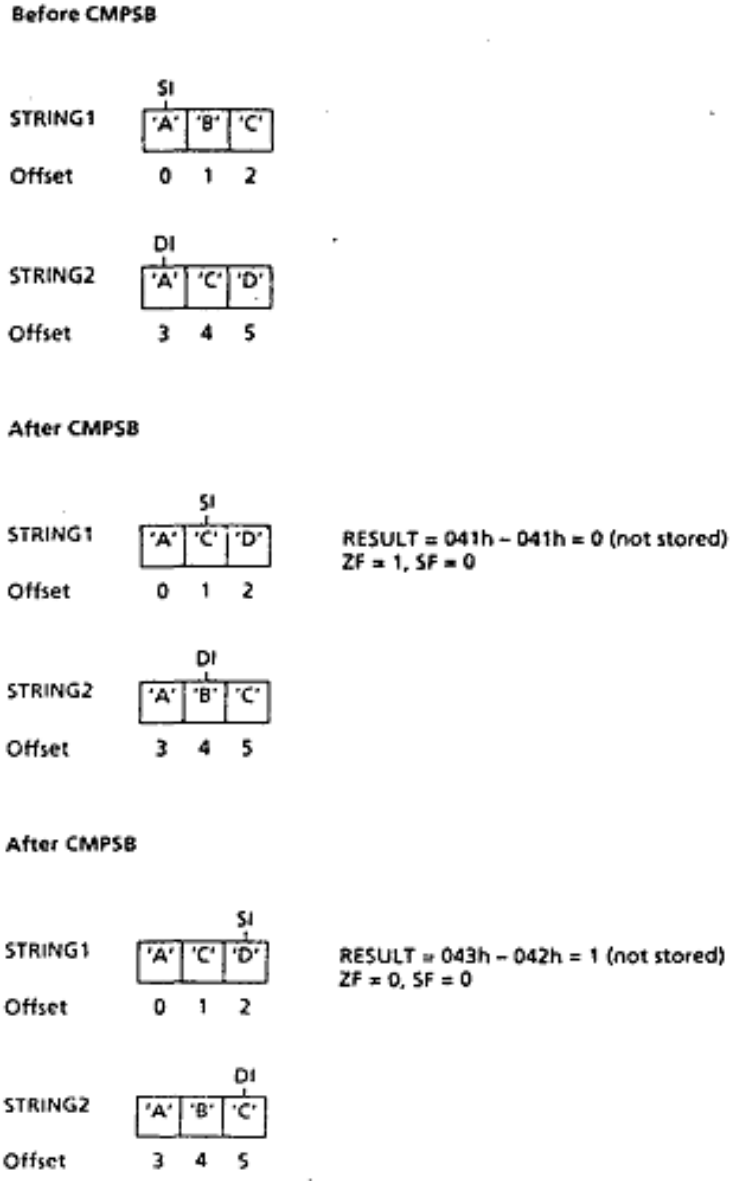


Figure 8 CMPSB

Example of CMPSB

```
mov   ax,@data
mov   ds,ax          ;initialize ds
mov   es,ax          ; and es
lea   si,[string1]   ;si points to first string
lea   di,[string2]   ;di points to second string
```


PREPARED BY
SHAHADAT HUSSAIN PARVEZ

```
        cld                 ;left to right processing
        mov  cx,10          ;# of chars in strings
        repe cmpsb          ;compare string bytes
        jl   S1_1st         ;string1 precedes string2
        jg   S2_1st         ;string2 precedes string1
        mov  ax, 0          ;put 0 in ax, string1=string2
        jmp  EXIT           ;and exit
S1_1st:
        mov  ax, 1          ;put 1 in ax, string1>string2
        jmp  EXIT           ;and exit
S2_1st:
        mov  ax, -1         ;put -1 in ax, string1<string2
EXIT:
```

General form of string instructions

Instruction		*Example*
MOVS	destination_string, source_string	MOVSB
CMPS	destination_string, source_string	CMPSB
STOS	destination_string	STOS STRING2
LODS	source_string	LODS STRING1
SCAS	destination_string	SCAS STRING2

1. Marut Chapter 8 exercise question 1-6
2. Marut Chapter 10 example 1, 12, 13
3. Marut Chapter 10 exercise question 1-4
4. Marut Chapter 11 example 1-2
5. Marut Chapter 11 exercise question 1-6

Programming assignment for LAB

1. Reversing input (Marut Chapter 8.2)
2. Procedure example (Marut Chapter 8.5)
3. Marut Chapter 8 exercise question 8,10
4. Marut program listing 11.1-11.5

Self Study

1. Marut chapter 10 section 10.6
2. Marut chapter 10 section 10.7
3. Marut chapter 11 section 11.6

PREPARED BY
SHAHADAT HUSSAIN PARVEZ