Chapter-11

The String Instructions

Overview

In this chapter we consider a special group of instructions called the string instructions. In assembly language, a **memory string** or **string** is simply a byte or word array. Thus, string instructions are designed for array processing.

Here are examples of operations that can be performed with the string instructions:

- Copy a string into another string.
- Search a string for a particular byte or word.
- Store characters in a string.
- Compare string of characters alphabetically.

The task carried out by the string instructions can be performed using register indirect mode.

11.1 The Direction Flag

The control flags of the FLAGS register of 8086 processor are used to control the processor's operations.

One of the control flags is the **direction flag** (DF). Its purpose is to determine the direction in which string operations will proceed. These operations are implemented by the two index registers SI and DI.

Suppose for example, that the following string has been declared:

```
STRING1    DB    'ABCDE'
```

If DF = 0, SI and DI proceed in the direction of increasing memory address from left to right across the string. Conversely, IF DF = 1, SI and DI proceed from right to left.

In the DEBUG display, DF = 0 is symbolized by UP, an DF = 1 by DN.

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 other flags.

11.2 Moving a String

Suppose we have defined two strings follows:

```
.DATA
STRING1    DB  'HELLO'
STRING2    DB  5  DUP  (?)
```

and we would like to move the contents of STRING1 (the source string) into STRING2 (the destination string). This operation is needed

for many string operations, such as duplicating a string or concatenating strings.

The **MOVSB** instruction

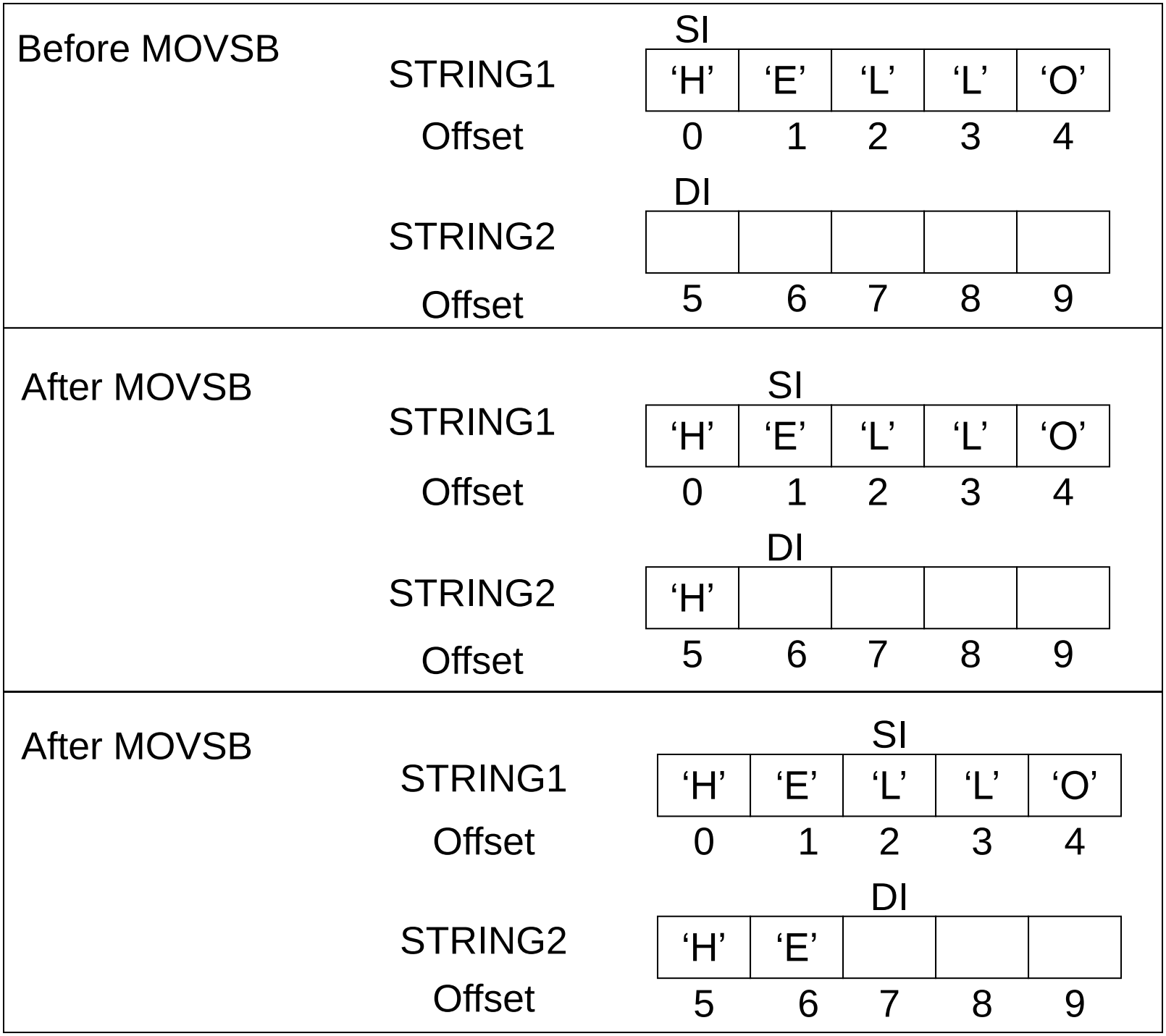
MOVSB ; move a string byte

copies the contents of the byte addressed by DS:SI, to the byte addressed by ES:DI. The contents of the source byte are unchanged. After the byte has been moved, both SI and DI are automatically incremented if DF = 0, or decremented if DF = 1.

For example, to move the first two bytes of STRING1 to STRING2, we execute the following instructions:

```
MOV  AX, @DATA
MOV  DS, AX
MOV  ES, AX
LEA  SI, STRING1
LEA  DI, STRING2
```

Figure 11.1 MOVSB



```
CLD
MOVSB
MOVSB
```

MOVSB is the first instructions we have seen that permits a memory-memory operation.

The REP Prefix

MOVSB moves only a single byte from the source string to the destination string. 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 cause MOVSB to executed N times. After each MOVSB, CX is decremented until it becomes 0. For example, to copy STRING1 into STRING2 we execute

```
CLD
LEA  SI, STRING1
LEA  DI, STRING2
MOV  CX, 5
REP  MOVSB
```

Example 11.1: Write instructions to copy STRING1 of the preceding section into STRING2 in reverse order.

Solution:

```
LEA  SI, STRING1+4
LEA  DI, STRING2
STD
MOV  CX, 5
```

```
MOVE:
MOVSB
ADD  DI,2
LOOP MOVE
```

MOVSW

There is a word from of MOVSB. It is

```
MOVSW                    ; move string word
```

MOVSW moves a word from the source string to the destination string. Like MOVSB, it expects DS:SI to point to a source string word, and ES:DI to point to destination string word. After a string word has been moved, both SI and DI are increased by 2 if DF = 0, or are decreased by 2 if DF = 1.

MOVSB and MOVSW have no effect on the flags.

Example 11.2: For the following array,

ARR DW 10, 20, 40, 50, 60, ?

write instructions to insert 30 between 20 and 40. (Assume DS and ES have been initialized to the data segment.)

Solution:

```
STD
LEA  SI, ARR+8H
LEA  DI, ARR+AH
MOV  CX, 3
REP  MOVSW
MOV  WORD  PTR  [DI], 30
```

Note: the PTR operator was introduced in section 10.2.3.

In general, PTR operator forces expression to be treated as a pointer of specified type:

```
.DATA
Num  DWORD  0
 .CODE
 mov ax, WORD  PTR [num] ; Load a word-size value from a DWORD
```

11.3 Store String

The STOSB Instruction

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, the **STOSW** instruction

STOSW ; store string word

moves the contents of AX register to the word at address ES:DI and updates DI by 2, according to the direction flag setting.

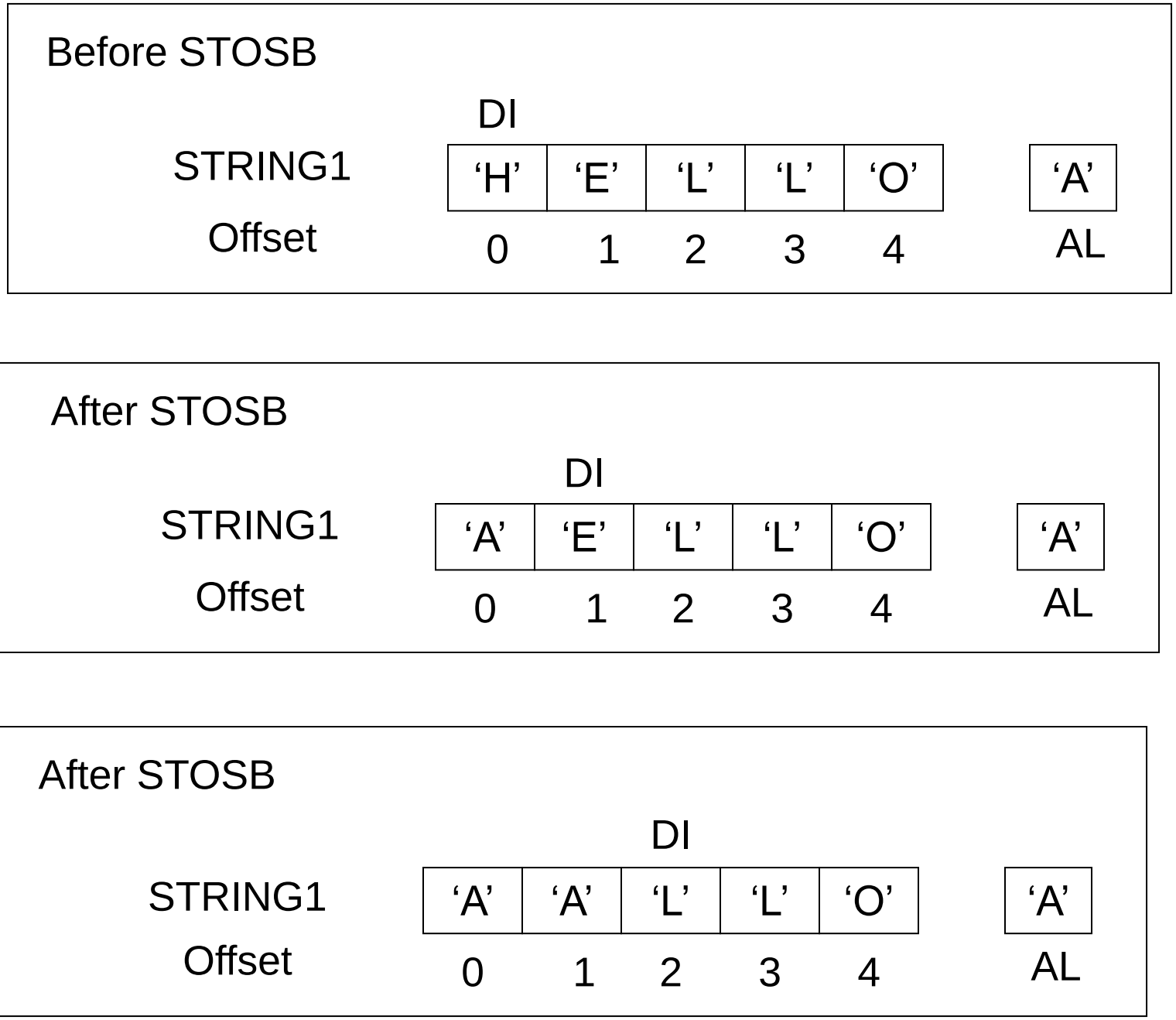
STOSB and STOSW have no effect on flags.

As an example of STOSB, the following instructions will store two “A”s in STRING1:

```
MOV   AX, @DATA
MOV  ES, AX
```

Figure 11.2

STOSB



```
LEA   DI, STRING1
CLD
MOV  AL, ‘A’
STOSB
STOSB
```

Reading and Storing a Character String

```
READ_STR  PROC  NEAR
PUSH AX
PUSH DI
CLD
XOR BX,BX
MOV AH,1
INT 21H
WHILE1:
CMP AL,0DH
JE END_WHILE1
```

```
CMP AL,8H
JNE ELSE1
DEC DI
DEC BX
JMP READ
ELSE1:
STOSB
INC BX
READ:
INT 21H
JMP WHILE1
END_WHILE1:
POP DI
POD AX
RET
READ_STR  ENDP
```

11.4 Load String

The LODSB Instruction

LODSB ; load string byte

moves the byte addressed by DS:SI into AL. SI is then incremented if DF = 0 or decremented if DF = 1. The word form is

LODSW ; load string word

It moves the word addressed by DS:SI into AX; SI is increased by 2 if DF = 0 or decreased by 2 if DF = 1.

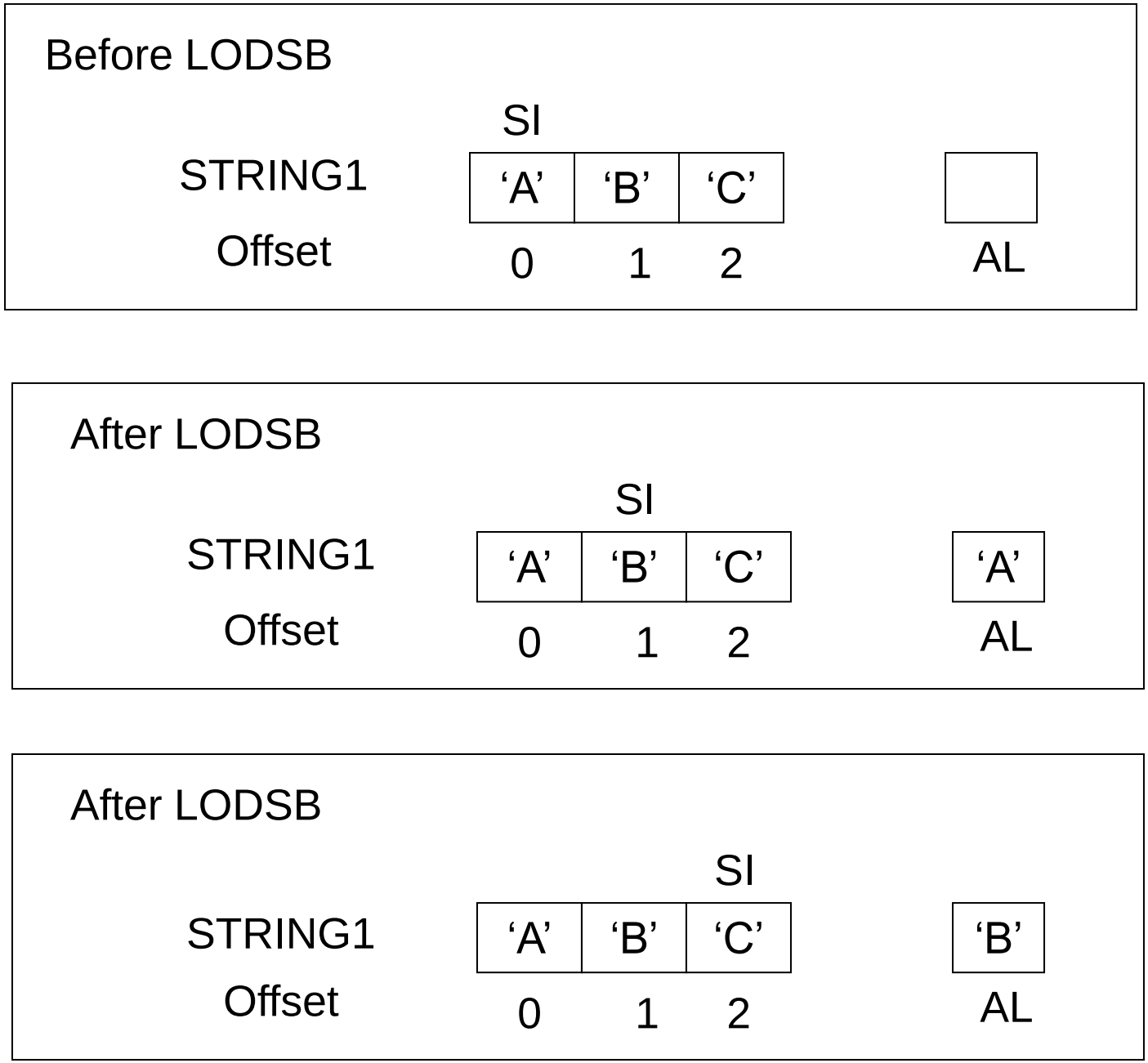
LODSB and LODSW have no effect on flags.

To illustrate LODSB, suppose STRING1 is defined as

STRING1 DB 'ABC'

Figure 11.3

LODSB



The following code successively loads the first and second bytes of STING1 into AL.

```
MOV   AX, @DATA
MOV  DS, AX
LEA   SI, STRING1
CLD
LODSB
LODSB
```

Displaying a Character String

```
DISP_STR  PROC  NEAR
PUSH AX
PUSH BX
PUSH CX
DUSH DX
PUSH SI
```

```
MOV CX,BX
JCXZ P_EXIT
CLD
MOV AH,2
TOP:
LODSB
MOV DL,AL
INT 21H
LOOP TOP
P_EXIT:
POP SI
POP DX
POP CX
POP BX
POP AX
RET
DISP_STR  ENDP
```

To demonstrate READ_STR and DISP_STR, we will write a program that reads a string (up to 80 characters) and displays the first 10 characters on the next line.

```
TITLE   PGM11_:  TEST READ_STR and DISP_STR
.MODEL SMALL
.STACK 100H
.DATA
STRING  DB  80  DUP (0)
CRLF  DB  0DH,0AH,'$'
.CODE
MAIN PROC
MOV AX,@DATA
MOV DS,AX
MOV ES,AX
LEA DI,STRING
```

```
CALL READ_STR
LEA DX,CRLF
MOV H,9
INT 21H
LEA SI,STRING
MOV BX,10
CALL DISP_STR
MOV AH,4CH
INT 21H
MAIN ENDP
; READ_STR goes here
; DISP_STR goes here
END MAIN
```

11.5 Scan String

The SCASB Instruction

The instruction

```
SCASB                    ; scan string byte
```

can be used to examine a string for a target byte. The target byte is contained in AL. SCASB subtracts the string byte pointed to by ES:DI from the content of AL and uses the result to set the flags. The result is not stored. Then, DI is incremented if DF = 0 or decremented if DF = 1.

The word form is

```
SCASW                    ; scan string word
```

in this case the target word is in AX. SCASW subtracts the word addressed by ES:DI from AX and sets the flags. DI is increased by 2 if DF = 0 or decreased by 2 if DF = 1.

All the status flags are affected by SCASB and SCASW.

For example, if the string

```
STRING1    DB  'ABC'
```

is defined, then these instructions examine the first two bytes of STRING1, looking for "B".

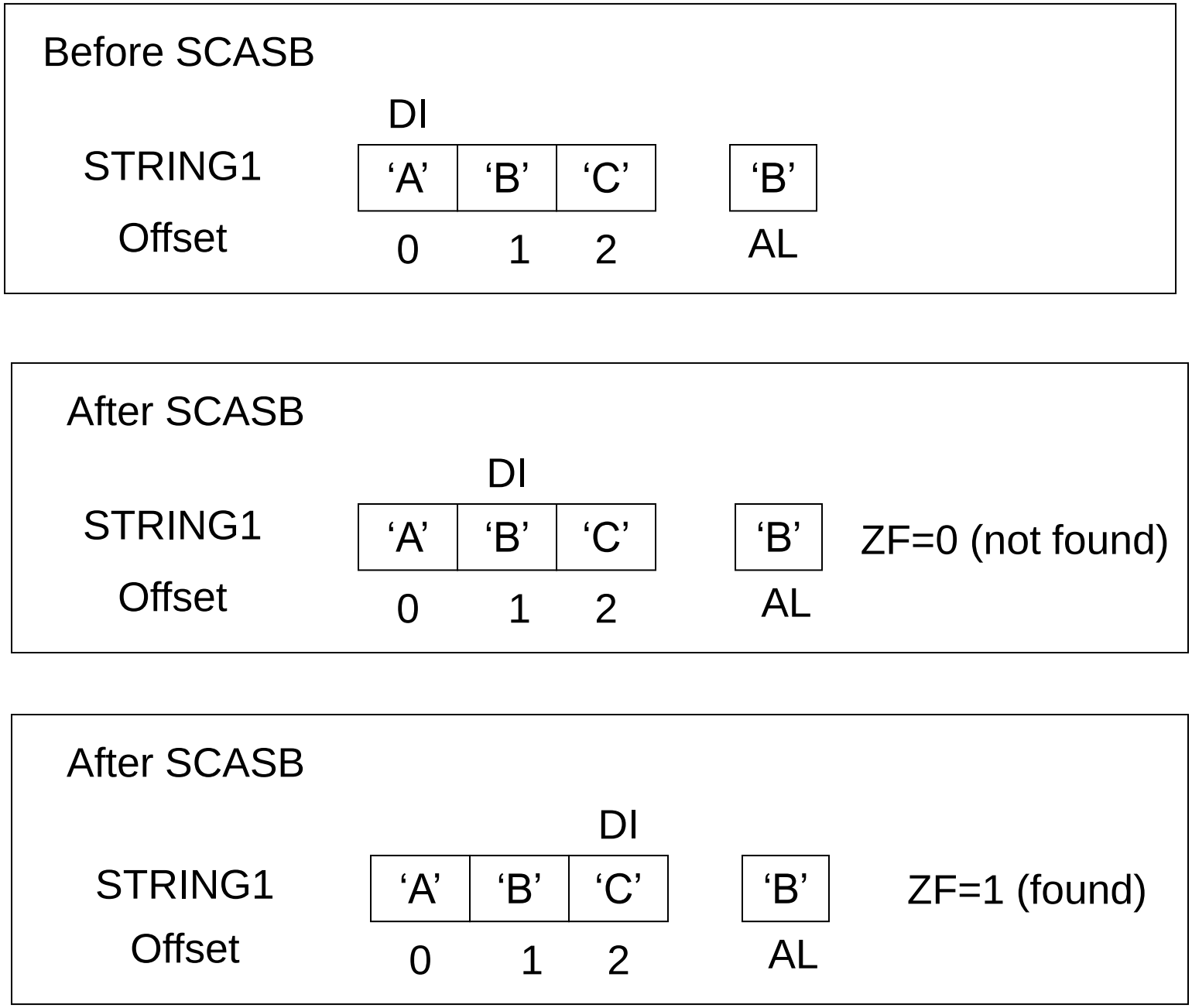
```
MOV AX, @DATA
MOV ES,AX
CLD
LEA DI, STRING1
MOV AL, 'B'
SCASB
SCASB
```

In looking for a target byte in a string, the string is traversed until the byte is found of the string ends. If CX is initialized to the number of bytes in the string,

```
REPNE   SCASB          ; repeat SCASB while not equal (to target)
```

Figure 11.4

SCASB



will repeatedly subtract each string byte from Al, update DI, and decremented CX until there is a zero result (the target is found) or CX = 0 (the string ends).

Note: **REPNZ** (repeat while not zero) generates the same machine code as **REPNE**.

Home Task :

Program: PGM11_4.ASM

COUNT VOWELS AND CONSONANTS

11.6 Compare String

The CMPSB Instruction

```
CMPSB                    ; compare string byte
```

subtracts the byte with address ES:DI from the byte address DS:SI, and sets the flags. The result is not stored. After that, both SI and DI

are incremented if DF = 0, or decremented if DF = 1.

The word version of CMPSB is

```
CMPSW                    ; compare string word
```

It subtracts the word with address ES:DI from the word whose address is DS:SI, and sets the flags. If DF = 0, SI and DI re increased by 2 if DF = 1, they are decreased by 2. CMPSW is useful in comparing word arrays of numbers.

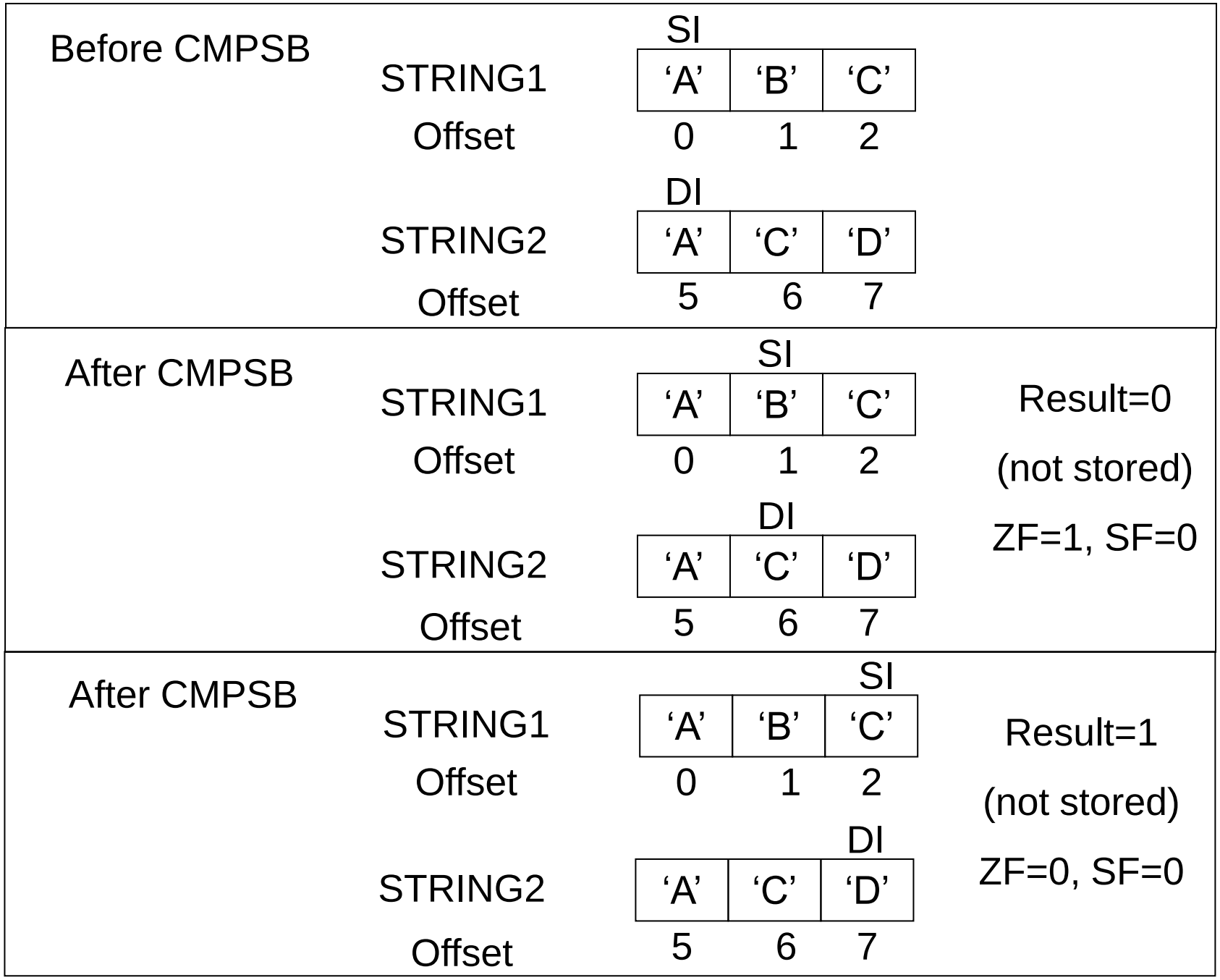
All the status flags are affected by CMPSB and CMPSW.

For example, suppose

```
.DATA
STRING1  DB  'ACD'
SRTING2  DB  'ABC'
```

The following instructions compare the first two bytes of the preceding string:

Figure 11.5 CMPSB



```
MOV  AX, @DATA
MOV  DS, AX
MOV  ES, AX
CLD
LEA  SI, STRING1
LEA  DI, STRING2
CMPSB
CMPSB
```

REPE and REPZ

String comparison may be done by attaching the prefix **REPE** (repeat while equal) or **REPZ** (repeat while zero) to CMPSB or CMPSW. CX is initialized to the number of bytes in the shorter string, then

```
REPE  CMPSB          ; compare string byte while equal
```

or

```
REPE   CMPSW         ; compare string words while equal
```

repeatedly executes CMPSB or CMPSW and decrements CX until

(1) there is a mismatch between corresponding string bytes or words,

(2) CX = 0.

The flags are set according to the result of the last comparison.

As an example, suppose STR1 and STR2 re strings of length 10. The following instructions put 0 in AX if the strings are identical, put 1 in AX if STR1 comes first alphabetically, or put 2 in AX if STR2 comes first alphabetically (assume DS and ES are initialized).

```
MOV CX,10
LEA SI,STR1
LEA DI,STR2
CLD
REPE CMPSB
JL STR1_FIRST
JG STR2_FIRST
```

```
MOV AX,0
JMP EXIT
STR1_FIRST:
MOV AX,1
JMP EXIT
STR2_FIRST:
MOV AX,2
EXIT:
```

Home Task **:**

11.6.1

Finding a Substring of a String