# Module 5: String Data and Defined Tables Processing

## Lesson 1: String Instructions

### Learning Outcomes

- LO 5.1.1 Explain the importance of string operations over conventional iteration.
- LO 5.1.2 Familiarize the needed parameters before executing a string operation
- LO 5.1.3 Explain the various string operations and apply them in manipulating a specific string.

### String Instructions

It is often necessary, however, to move or compare data fields that exceed the length of a single byte, word or doubleword. Items of this type are known as string data and may be in either character or numeric format. Assemble language provides these string instructions for processing string data.

**MOVS** – moves one byte of, word, or doubleword from one location in memory to another
**LODS** – Loads from memory a byte into AL, word into AX, or doubleword to EAX
**STOS** – Stores the contents of AL, AX, and EAX into memory
**CMPS** – Compares bytes, Word, or Doubleword to memory.
**SCAS** – Compares the contents of AL, AX, or EAX with contents of items in memory

An association instruction REP prefix, causes a string instruction to perform repetitively so that it may process any number of bytes, words, doublewords a specified number of times.

#### Features of String Operations

Each string instruction has a byte, word, and doubleword version of repetitive processing and assumes use of the ES:DI or DS:DI pair of registers. Thus you could select a byte operation for a string with an odd number of bytes and a word operation for a string with an even number of bytes. The use of word and doubleword operations can provide faster processing.

String instruction expect that the DI and SI registers contain valid offset addresses that reference bytes in memory. SI is normally associated with DS (data segment) as DS:SI whereas DI is always associated with ES (Extra Segment) as ES:DI. For this reason, MOVS, STOS, CMPS, and SCAS require that an .EXE program.

```
MOV AX, @data
MOV DS, AX
MOV ES, AX
```

#### REP: Repeat Prefix

The REP prefix immediately before a string instruction, such as REP MOVSB, provides for repeated execution based on an initial count that you set in CX. REP executes the string instruction, decrements CX, and repeats this operation until CX is zero. In this way you can process string virtually any length. The Direction Flag (DF) determines the direction of a repeated operation.

- To process from left to right, use CLD to clear the DF to 0
- To process from right to left, use STD to set DF to 1.

See example below:

```
SEND_STR DB 20 DUP ('*')      ; Send Field
         RECV_STR DB 20 DUP (' ')      ; Receive Field
         ….
```

```
CLD                     ; Clear Direction Flag
MOV CX, 20          ; Initialize for 20Bytes
LEA SI, SEND_STR    ; send address(DS:SI)
LEA DI, RECV_STR    ; receive address(ES:DI)
REP MOVSB               ; copy send to receive
```

MOV, LODS, and STOS always fully repeat the specified number of times. However, CMPS and SCAS make comparisons that set status flags so that the operations can end immediately on finding a specified condition. The variations of REP that CMPS and SCAS use for this purpose are the following:

- REP: Repeat operation until CX is zero
- REPE/REPZ: Repeat the operation while Zero Flag (ZF) indicates equal/zero. Stop when ZF indicates not equal/zero or when CX is decremented to zero
- REPNE or REPNZ: Repeat the operation while ZF indicates not equal/zero. Stop when ZF indicates equal or zero or when CX is decremented to zero.

**MOVS: Move String Instruction**

MOVSB, MOVSW, and MOVSD combined with REP prefix and a length in CX can move a specified number of characters. The segment:offset registers are ES: DI for the receiving string and DS:SI for the sending string. As a result, at the start of an .EXE program, be sure to initialize ES along with DS, and prior to executing MOVS, also initialize DI and SI. Depending on the Direction flag, MOVS increment or decrements DI and SI by 1 for byte, 2 for word, and 4 for double word.

```
CLD
MOV CX, 12          ; Number of Words
LEA DI, RECV_STR    ; Address (ES:DI)
LEA SI, SEND_STR    ; Address (DS:SI)
REP MOVSB               ; Move 12 bytes
```

A MOV instruction initializes CX with 12 (the length of SEND_STR and RECV_STR). Two LEA instruction load SI and DI with the offset address of SEND_STR and RECV_STR respectively. The REP MOVSB now performs the following.

Moves the leftmost byte of SEND_STR (addressed by DS:SI) to the leftmost byte of RECV_STR (addressed by ES:DI)

- Increments DI and SI by 1 for the next bytes to the right
- Decrements CX by 1
- Repeats this operation 12 times until CX becomes 0.

Because the Direction flag is 0 and MOVSB increments DI and SI, each iteration processes one byte farther to the right, as SEND_STR+1 to RECV_STR+1, and so on. At the end of the execution, CX would be 0, SI and DI would contain SEND_STR+12 and RECV_STR+12 respectively.

To process from right to left, set the Direction Flag to 1. MOVSB then decrement DI and SI, but initialize SI and DI with the end portion of the string, thus SI with SEND_STR+11 and DI with RECV_STR+11

The instructions below are the equivalent to the REP MOVSB operation:

```
        LEA DI, RECV_STR        ; Address (ES:DI)
        LEA SI, SEND_STR        ; Address (DS:SI)
MOV CX, 12                      ; Initialize count to 12
L30: MOV AH, [SI]               ; Move first element to AH
```

```
            MOV [DI], AH            ; Move contents of AH to DI
            INC DI                  ; Move to the next character
            INC SI
        LOOP L30                    ; Loop until CX is zero
```

**LODS: Load String Instruction**

LODS simply loads AL with a byte, AX with a word, EAX with a doubleword from memory. The memory address is subject to DS:SI registers, although you can override SI. Depending on the Direction Flag, the operation also increments or decrements SI by 1 for byte, 2 for word, and 4 for doublewords.

Because LODS operation fills the register, there is no practical reason to use the REP prefix with it. For most purposes, a simple MOV instruction is adequate. But MOV generates three bytes of machine code, whereas LODS generates only one, although you have to initialize SI. You could use LODS to step through a string one byte, word, or doubleword at a time, examining successively for a particular character.

The instructions equivalent to LODSB are:

```
      MOV AL, [SI]              ; Transfer one byte to AL
      INC SI                    ; Increment SI for next byte
```

The example below would transfer a given string in reverse order to the destination string

```
  String1 db "Interstellar"     ; Data Items
            String2 db 12 DUP (20H)
            CLD                                 ; Left to right
            MOV CX, 12
            LEA SI, String1         ; Address of string (DS:SI)
            LEA DI, String2+11            ;   Address   of   string
  (ES:DI)
      L20:    LODS                        ; Get character in AL
               MOV [DI], AL         ; Store in String2
               DEC DI                     ; right to left
      LOOP A20                            ;    Loop    until    12
  characters
```

**STOS: Store String Instruction**

STOS stores the contents of AL, AX, or EAX into a byte, word, or doubleword in memory. The memory address is always subject to ES:DI. Depending on the Direction Flag, STOS also increments or decrements DI by 1 for byte, 2 for word, and 4 for double words.

A practical use of STOS with a REP prefix is to initialize a data area to any specified value, such as clearing an area to blanks. You set the number of bytes, words, or doublewords in CX. The instruction equivalent to REP STOSB are:

```
      JCXZ              ; Jump if CX Zero
  L20: MOV [DI], AL     ; Store in AL
      INC/DEC DI  ; Depends on Direction
      LOOP L20          ; Repeat until CX is zero
  L30:                  ; Operation complete
```

The STOSB instruction in the following example repeatedly stores a word containing 20H 12 times through STRING1. The operation stores AL in the first byte. At the end, all STRING1 is blank, CX contains 00, and DI contains the address of STRING1+12

```
CLD               ; Left to right
MOV AL, 20H       ; Move
MOV CX, 12        ; 12 Blank bytes
LEA DI, STRING1   ; to String
REP STOSW
```

**CMPS: Compare String Instruction**

CMPS compares the contents of one memory location (addressed by DS:SI) with that of another memory location (addressed by ES:DI). Depending on the Direction Flag, CMPS, increments or decrements SI and DI, by 1for byte, 2 for word, and 4 for doubleword. The operation then sets the AF, CF, OF, PF, SF, and ZF flags depending on the result, if any, of the compare. When combined with a REPnn prefix and length in CX, CMPS for byte, CMPSW for word, and CMPSD for doubleword.

Note that CMP compares operand 2 to operand 1, whereas CMPS compares operand 1 to 2. Also, CMPS provides alphanumeric comparisons according to ASCII values. The operation is not suited to algebraic comparisons, which consist of signed numeric values.

Consider the comparison of two strings containing “TASM” and “MASM”. A comparison from left to right, one byte at a time, results in the following:

```
T: M  Unequal (M is higher)
A: A  Equal
S: S  Equal
M: M  Equal
```

A comparison to the entire four bytes ends with a comparison of “M” with “M” (equal). Now since the two names are not identical, the operation should end as soon as it makes a comparison between two different characters. For this purpose, the REP variation, REPE (Repeat if Equal), repeats the operation as long as the comparison is between equal characters, or until CX is decremented to 0. The coding for a repeated one-byte comparison is REPE CMPSB. The examples below use CMPSB. The first example continues CMPSB operation until CX would become 0, since each character are of equal values.

```
String1 DB “Interstellar”
String2 DB “Interstellar”
String3 DB 12 DUP(‘*’)
CLD
MOV CX, 12
LEA DI, String2
LEA SI, String1
REPE CMPSB
JE exit
```

The second example compares STRING2 and STRING3, which contain different values. CMPSB operation ends after comparing the first byte and results in a high/unequal condition: CX contains 11, DI contains STRING3+1, SI contains the address STRING2+1. Sign Flag is positive, and the Zero Flag indicates unequal.

```
MOV CX, 12
LEA DI, String3
```

```
LEA SI, String2
REPE CMPSB
JE exit
```

### SCAS: Scans String Instruction

SCAS differs from CMPS in that SCAS scans for a specified value. To this end, SCAS compares the contents of a memory location (addressed by ES:DI) with the contents of AL, AX, or EAX. Depending on the Direction Flag, SCAS also increments or decrements DI by 1 for byte, 2 for word and 4 for doubleword. The operation ends on a successful compare or when REP reduces CX to zero. SCAS the sets AF, CF, OF, PF, and ZF flags, depending on the result of the compare. When combined with a REPnn prefix and a length in CX, SCAS can scan virtually any string length. The three SCAS operation are SCASB for byte, SCASW for word, and SCASD for doubleword.

SCAS would be particularly useful for a text editing application in which the program has to scan for punctuation, such as periods, commas, and blanks

The following example scan STRING1 for the lowercase letter r. Because the SCASB operation is to continue scanning while the comparison is not equal or until CX is 0, the operation in this case is REPNE SCASB

```
String1 DB 'Interstellar'
…
CLD
MOV AL, 'r'        ; Scan String for 'r'
MOV CX, 12         ; 12 characters
LEA DI, String1    ; ES: DI
REPNE SCASB        ; Start Scan
JE exit                  ; Found
```

## Lesson 2: The XLAT (Translate) Instruction and manipulating table entries

### Learning Outcomes

- LO 5.2.1 Implement the in the searching and sorting of table entries
- LO 5.2.2 Explain and implement the XLAT (translate) instruction. Implement addressing and data manipulation of a two dimensional array.

### Defining Tables

Many program application require table and arrays containing such data as names, descriptions, quantities and rates. The definition and use of tables largely involves applying what you have already learned such as searching and sorting. Most tables are arranged in a consistent manner, with each entry defined with the same format (Character or Numeric), with the same length and in either ascending or descending order. For example:

```
MONTH_TBL DB 'JAN' 'FEB','MAR'
CUST_TBL DB 205, 206 209
```

All entries in MONTH_TBL are three characters. However, although all entries in CUST_TBL are defined in as three digits, the assembler would convert then decimal numbers to binary format, provided that they would not exceed the length of 255.

A table may also contain a mixture of numeric character and character values, provided that their definition are consistent. For example:

```
STOCK_TBL DB 12, 'Computers',14,'Paper....',17,'Diskettes'
```

The four dots following the description “Paper” are to show the spaces that should be present. For clarity you may code each pair of table entries separated by line.

```
STOCK_TBL DB 12, 'Computers'
          DB 14,'Paper....'
          DB 17,'Diskettes'
```

In real-world situation, many programs are table driven. That is, tables are stored as disk files, which any number of programs may require for processing.

### Type, Length and Size Operators

The assembler supplies a number of special operators you may find useful. For example, the length of a table may change over time and you may have to modify a program to account the new definition and add routines that check for the end of the table. The use of TYPE, LENGTH and SIZE operators can help reduce the number of instructions that have changed. For example:

```
RAIN_TBL DW 12 DUP(?)   ; Table of 12 words
```

The length use the TYPE operator to determine the definition (DW in this case), the LENGTH operator to determine the DUP factor (12) and the SIZE operator to determine the number of bytes (12x2=24)

### Direct Addressing of Tables

Suppose that a user enters a numeric month such as 03 and that program is to convert it into alphanumeric format. The routine to perform this conversion involves defining a table of alphabetic months, all of equal length. The length of each entry should be that of the longest name, September in this format:

```
MONTH_TBL DB 'January..'
          DB 'February.'
          DB 'March....'
          DB 'April....'
          DB 'May......'
          DB 'June.....'
          DB 'July.....'
          DB 'August...'
          DB 'September'
          DB 'October..'
          DB 'November.'
          DB 'December.'
```

The entry ‘January’ is MONTH_TBL+00, ‘February’ is at MONTH_TBL+09, ‘March’ is at MONTH_TBL+18, and so forth. Let us say that a user keys in 3 (March), which the program is to locate in the table. The algorithm is a follows:

1. Convert the entered month from ASCII 33 to binary 3
2. Deduct 1 from this number: 3 – 1 = 2 (because month 01 is at MONTH_TBL+00)
3. Multiply the new number by 9 (the length of each entry): 2 x 9 = 18

4. Add this product (18) to the address of MONTH_TBL; the result is the address of the required description: MONTH_TBL+18 where the entry "March" begins.

This algorithm is known as direct table addressing. Because it calculates the required table address directly, you don't have to define the numeric months in the table and the program doesn't have to search successively to the table.

```
;Convert ASCII to month
XOR word ptr MONTH_IN,3030H   ; Clear the 3's in ASCII
MOV AX,0
MOV BL,MONTH_IN+1    ;Store the one's digit in BL
MOV AL,MONTH_IN+0    ;Store the ten's digit in AL
MOV DL,10
MUL DL              ;Multiply the ten's digit with 10
ADD AL,BL           ;Add the one's digit with the ten's digit

;Locate the month table
DEC AL              ;Decrement AL since the starting address is at
0
MOV DL, LEN_ENTRY   ;Set the length of DL as multiplier offset
MUL DL
LEA BP,MONTH_TBL    ;load the address of MONTH_TBL
ADD BP,AX           ;locate the address of 12th entry on table

MOV AX,1301H        ;Request interrupt
MOV BX,0016H        ;Page:attribute
MOV CX,LEN_ENTRY    ;9 Characters
MOV DX,0812H        ;Row:column position
INT 10H
```

**The XLAT Instruction**

The XLAT instruction translates the bit configuration of a byte into another predefined configuration. You can use XLAT, for example to, validate the contents of data items or to encrypt data.

| [label:] | XLAT | ;no operand |
|---|---|---|

To use XLAT, you define a translation table that accounts for all 256 possible characters XLAT requires that the address of the table is in BX and the byte to be translated is in AL. Example:

```
XLAT_TBL DB 45 DUP (40H)
             DB 60H, 4BH
             DB 40H
             DB 0F0H,0F1H,0F2H,0F3H,0F4H
             DB 0F5H,0F6H,0F7H,0F8H,0F9H
             DB 198 DUP (40H)
```

XLAT uses the value of AL as an offset address, in effect, BX contains the starting address of the table and AL contains an offset within the table.

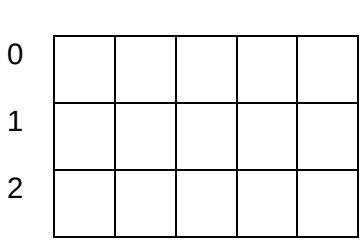
```
MOV AX,@data
MOV ES,AX
MOV DS,AX

LEA BX,XLAT_TBL
```

```
MOV AL,50
XLAT
XLAT
XLAT
```

This example would first address the content of 50 (32H) which contains 0F2H then use 0F2H as offset which contains the address 00H then address 00H contains 40H.

## Two-Dimensional Arrays

The two-dimensional array consist of y rows and x columns, as following example 3 rows by 4 columns.




The array is the form [row, column] and contains 3x5 = 15 cells. In the memory each row of data can be considered a one-dimensional array, that is each row successively follows the other. However, it helps to think of the array as being two-dimensional and could be defined like

```
DATA_ARRAY DW 3 DUP(5 DUP(?))
```

Accessing an element in the array, such as [2,3] that row 2, column 3, involves the following steps.

1. Multiply row by number of elements in a column 2 x 5 =10
2. Add the column: 10 + 3