Unit 5 String Manipulation

Structure:

5.1 Introduction

In the previous unit, we studied the various types of addressing modes, various types of instructions like data transfer and arithmetic instructions, logical and branch instructions. One very important computer application is *text processing,* which is the manipulation of sequences of bytes that contain the alphanumeric codes for characters, i.e., character strings. In the design of a text editor or text formatting program it is necessary to have program sequences for moving and comparing character strings, and for inserting strings into and deleting them from other strings. It is often necessary to search a string for a given substring or to replace a substring with a different substring. A text editor is a requirement on any general purpose computer system, and features which improve the ability of such a system to work with character strings can be a definite advantage. This unit is concerned primarily with those features of the 8086 that facilitate the handling of character strings. In this unit, you will also study about procedures and macros.

Objectives:

After studying this unit, you should be able to:

- explain 8086 string instructions
- explain the significance of REP prefix
- discuss on table translation
- explain the purpose of number format conversions
- define procedure
- discuss about macros

5.2 String Instructions

A series of data byte or word available in memory at consecutive locations, to be referred as Byte String or Word String. A String of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent. The 8086 supports a set of more powerful instructions for string manipulations for referring to a string, two parameters are required.

i. Starting and End Address of the String.
ii. Length of the String.

The length of the string is usually stored as count in the CX register. The incrementing or decrementing of the pointer, in string instructions, depends upon the Direction Flag (DF) Status. If it is a Byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation, the index registers are updated by two.

The 8086 microprocessor is equipped with special instructions to handle string operations. By string we mean a series of data words or bytes that reside in consecutive memory locations. The usefulness of string instructions can be seen when in the memory, data has to be moved, searched or compared in blocks. Consider the case of 100 (say) words or bytes in a particular memory area that is to be moved to another memory area. This can very well be done using pointer registers and looping using a counter. However, this whole process can be automated with lesser number of instructions if we use string instructions. Similarly, we may need to compare two blocks of data for equality, or search a data block for a particular data. In all these cases, string instructions make our task easier and our code shorter. However, before using these instructions, we have to include a few initialization steps in our code. The string instructions of the 8086 permit a programmer to implement operations such as to move data from one block of memory to a block elsewhere in memory. A second type of operation that is easily performed is to scan a string and data elements stored in memory looking for a specific value. Other examples are to compare the elements and two strings together in order to determine whether they are the same or different. The string instructions are summarized in figure 5.1. Because the string instructions can operate on only a single byte or word unless they are used with the REP prefix discussed in section.5.3, they are often referred to as string primitives, or simply *primitives.* All of the primitives are 1 byte long, with bit 0 indicating whether a byte (bit 0 = 0) or a word (bit 0 = 1) is being manipulated.

There are five basic primitives and each may appear in one of the following three forms:

Operation Operand(s)

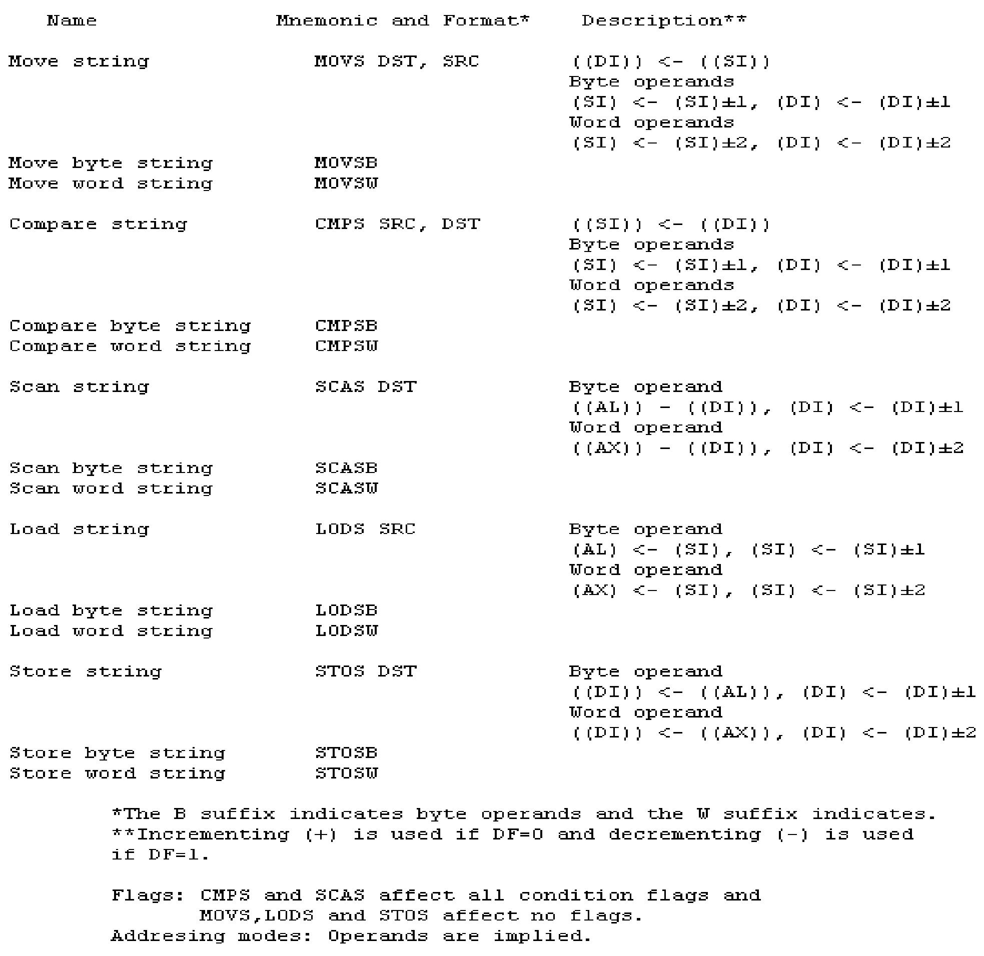
Or

Operation B

Or

Operation W

If the first form is used, whether bytes or words are to be operated on is determined implicitly by the type of the operand(s). The second and third forms explicitly indicate byte and word operations, respectively.



Name	Mnemonic and Format*	Description**
Move string	MOVS DST, SRC	((DI)) <- ((SI)) Byte operands (SI) <- (SI)±1, (DI) <- (DI)±1 Word operands (SI) <- (SI)±2, (DI) <- (DI)±2
Move byte string	MOVSB	
Move word string	MOVSW	
Compare string	CMPS SRC, DST	((SI)) <- ((DI)) Byte operands (SI) <- (SI)±1, (DI) <- (DI)±1 Word operands (SI) <- (SI)±2, (DI) <- (DI)±2
Compare byte string	CMPSB	
Compare word string	CMPSW	
Scan string	SCAS DST	Byte operand ((AL)) - ((DI)), (DI) <- (DI)±1 Word operand ((AX)) - ((DI)), (DI) <- (DI)±2
Scan byte string	SCASB	
Scan word string	SCASW	
Load string	LODS SRC	Byte operand (AL) <- (SI), (SI) <- (SI)±1 Word operand (AX) <- (SI), (SI) <- (SI)±2
Load byte string	LODSB	
Load word string	LODSW	
Store string	STOS DST	Byte operand ((DI)) <- ((AL)), (DI) <- (DI)±1 Word operand ((DI)) <- ((AX)), (DI) <- (DI)±2
Store byte string	STOSB	
Store word string	STOSW	

*The B suffix indicates byte operands and the W suffix indicates.
**Incrementing (+) is used if DF=0 and decrementing (-) is used if DF=1.

Flags: CMPS and SCAS affect all condition flags and MOVS,LODS and STOS affect no flags.
Addresing modes: Operands are implied.

Figure 5.1: String Instructions

Regardless of the form of the primitive the actual operands are determined solely by the contents of the SI, DI, DS, and ES registers. For a source operand the address of the operand is the sum of (SI) and (DS) x 16 unless the DS register is overridden by explicitly specifying the ES, CS, or SS register as the segment register. The address of a destination operand is always the sum of (DI) and (ES) x 16_{10}. Now lets us consider some string instructions.

Move String: MOV SB, MOV SW:

These instructions copy a byte or a word from a location in the data segment to a location in extra segment. An element of the string specified by the source index (SI) register with respect to the current data segment (DS) register is moved to the location specified by the destination index (DI) register with respect to the current extra segment (ES) register. The way to tell the assembler whether to code the instruction for moving a byte or word is to add a “B” or a “W” to the MOVS mnemonic. So the move can be performed on a byte (MOV SB) or a word (MOV SW) of data. After the move is complete, the contents of both SI & DI are automatically incremented or decremented by 1 for a byte move and by 2 for a word move. Address pointers SI and DI increment or decrement depends on how the direction flag DF is set.

Example : Block move program using the move string instruction

```
MOV AX, DATA SEG ADDR
MOV DS, AX
MOV ES, AX
MOV SI, BLK 1 ADDR
MOV DI, BLK 2 ADDR
MOV CK, N
CDF; DF=0
NEXT: MOV SB
LOOP NEXT
HLT
```

Load and store strings: (LOD SB/LOD SW and STO SB/STO SW)

LOD SB/LOD SW instructions copies a byte or a word from a location pointed by SI to AX.

LOD SB: Loads a byte from a string in memory into AL. The address in SI is used relative to DS to determine the address of the memory location of the string element.
(AL) ← [(DS) + (SI)]
(SI) ← (SI) ±1

LOD SW: The word string element at the physical address derived from DS and SI is to be loaded into AX. SI is automatically incremented by 2.
(AX) ← [(DS) + (SI)]
(SI) ← (SI) ± 2

STO SB : Stores a byte from AL into a string location in memory. This time the contents of ES and DI are used to form the address of the storage location in memory
[(ES) + (DI)] ← (AL)
(DI) ← (DI) + 1

STO SW: [(ES) + (DI)] ← (AX)
(DI) ← (DI) ± 2

Example: Clearing a block of memory with a STOSB operation.

```
MOV AX, 0
MOV DS, AX
MOV ES, AX
MOV DI, A000
MOV CX, OF
CDF
AGAIN: STO SB
LOOP NE AGAIN

NEXT:
```

Clear A000 to A00F to 0016
When working with strings, the advantages of the MOVS and CMPS instructions over the MOV and CMP instructions are:

1. They are only 1 byte long.
2. Both operands are memory operands.
3. Their auto-indexing removes the the need for separate incrementing or decrementing instructions, thus decreasing overall processing time.

As an example consider the problem of moving the contents of a block of memory to another area in memory. A solution that uses only the MOV instruction, which cannot perform a memory-to-memory transfer, is shown in figure 5.2(a).

```
            MOV SI, OFFSET STRING1     ;USE SI AS SOURCE INDEX
            MOV DI, OFFSET STRING2     ;USE DI AS DESTINATION INDEX
            MOV CX, LENGTH STRING1     ;PUT LENGTH IN CX
MOVE:       MOV AL, (SI)               ;MOVE BYTE FROM SOURCE
            MOV (DI), AL               ;TO DESTINATION
            INC SI                     ;INCREMENT SOURCE INDEX
            INC DI                     ;INCREMENT DESTINATION INDEX
            LOOP MOVE
```

(a)Uses the MOV instruction

```
            MOV SI, OFFSET STRING1     ;ASUME (DS)=(ES)
            MOV DI, OFFSET STRING2
            MOV CX, LENGTH STRING1
            CLD                        ;CLEARALL FLAG
MOVE:       MOVS STRING2, STRING1      ;FOR AUTO-INCREMENTING
            LOOP MOVE
```

(B)Uses primitive MOVS

Figure 5.2: Program sequences for moving a block of data

A solution that employs the MOVS instruction is given in figure 5.2 (B). Note that the second program sequence may move either bytes or words, depending on the type of STRING1 and STRING2.

CMPS: Compare string byte or string word.

The CMPS instruction can be used to compare two strings of byte or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero flag is set. The REP instruction prefix is used to repeat the operation till CX (counter) becomes zero or the condition specified by the REP prefix is false.

STOS: Store string byte or string word.

The STOS instruction stores the AL /AX register contents to a location in the string pointed by the ES: DI pair. The DI is modified accordingly. No flags are affected by this instruction. The direction flag controls the string instruction execution. The SI and DI are modified after each iteration automatically. If DI = 1, then the execution follows auto decrement mode, SI and DI are decremented automatically after each iteration. IF DS = 0, then the execution follows auto increment mode. In this mode, SI and DI are incremented automatically after each iteration.

SCAN : Scan String Byte or String Word

This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The String is pointed to by ES:DI register pair. The length of the string s stored in CX. The DF controls the mode for scanning of the string. Whenever a match to the specified operand, is found in the string, execution stops and the zero Flag is set. If no match is found, the zero flag is reset.

Self Assessment Questions

1. The string instructions can operate on only a single byte or word unless they are used with the ____________ prefix.
2. Regardless of the form of the primitive the actual ________________ are determined solely by the contents of the SI, DI, DS, and ES registers.

5.3 REP Prefix

Because string operations inherently involve looping, the 8086 machine language includes a prefix that considerably simplifies the use of string primitives with loops. This prefix has the machine code

1 1 1 1 10 0 1 Z

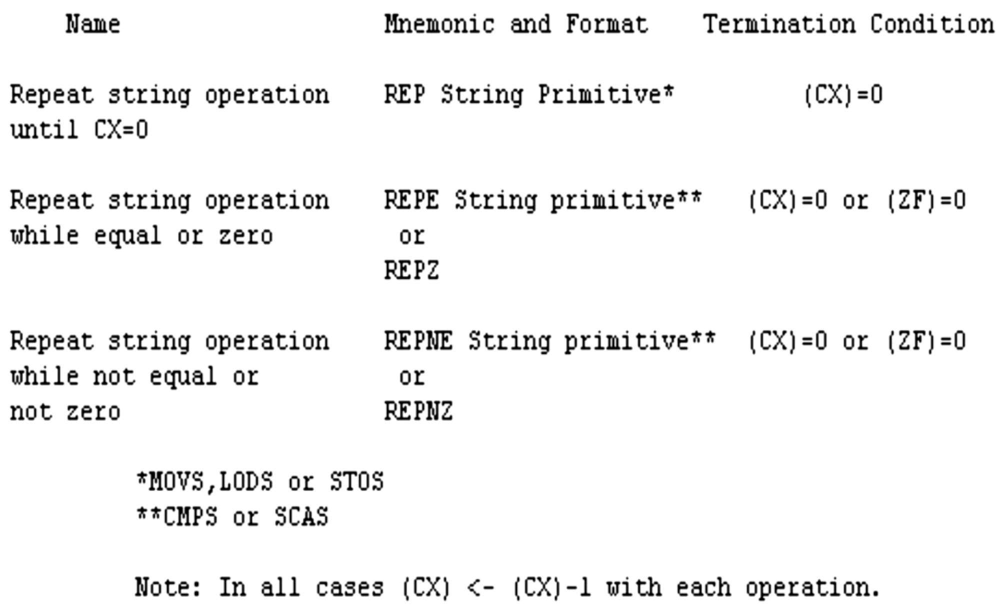
where, for the CMPS and SCAS primitives, the Z bit helps control the loop. By prefixing MOVS, LODS and STOS, which do not affect the flags, with the REP prefix 11110011, they are repeated the number of times indicated by the CX register according to the following steps:

1. If (CX) = 0, exit the REP operation.
2. Perform the specified primitive.
3. Decrement CX by 1.
4. Repeat steps 1 through 3.

For the CMPS and SCAS primitives, which do affect the flags, the prefix causes them to be repeated the number of times indicated by the CX register or until the Z-bit does not match the ZF flag, whichever occurs first.

The Z-bit/ZF flag comparison is made after each repetition. In any case the CX register is decremented with each repetition and, unless a Z-bit/ZF flag mismatch causes the termination, CX will be 0 when the loop is exited. Therefore, after repetitively executing the CMPS and SCAS primitives, the contents of ZF can be examined to determine the cause of termination.

In assembly language the prefix is invoked by placing the appropriate repeat (REP) mnemonic before the primitive. The REP mnemonics are defined in figure 5.3.



Name	Mnemonic and Format	Termination Condition
Repeat string operation until CX=0	REP String Primitive*	(CX)=0
Repeat string operation while equal or zero	REPE String primitive** or REPZ	(CX)=0 or (ZF)=0
Repeat string operation while not equal or not zero	REPNE String primitive** or REPNZ	(CX)=0 or (ZF)=0

*MOVS,LODS or STOS
**CMPS or SCAS

Note: In all cases (CX) <- (CX)-1 with each operation.

Figure 5.3: REP prefix

As an example of the use of the REP prefix let us reconsider the program sequence for moving a string within memory given in figure 5.2(b). By replacing the explicit loop

MOVE:	MOVS	STRING2,STRING1
	LOOP	MOVE
with		
REP	MOVS	STRING2, STRING1

5.4 Table Translation

It is sometimes necessary to translate from one code to another. A terminal may communicate with the computer using the EBCDIC (Extended Binary Coded Decimal Interchange Code) alphanumeric code even though the computer's software is designed to work with the ASCII (American Standard Code for Information Interchange) code, or vice versa. Code conversions involving fewer than 8 bits (which accommodates up to 256 distinct entities) can be performed most easily by storing the desired code in an array of up to 256 bytes and letting the original code be the index within the array of the desired code values. If the EBCDIC code were being converted to the ASCII code, then the EBCDIC code value for "A", which is 11000001, would be added to the address of the beginning of the array. Then, by putting the ASCII code for A, which is 01000001, in the array element having the address of the array plus 00C1, the code conversion is readily accomplished.

The 8086 has an instruction specifically designed for executing this procedure. It is the XLAT instruction. This instruction may appear either as XLATB or XLAT OPR. Both forms produce the following 1-byte machine language instruction:

1110101111

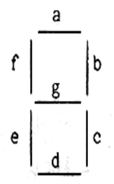
In the XLAT OPR form, OPR is a dummy operand that is normally the variable name associated with the translation table and serves only to improve the readability of the program. The XLAT instruction assumes the base address of the byte array is in the BX register and the byte to be converted is in the AL register. The desired code value is taken from the array and put in the AL register. None of the flags are affected.

As an example, suppose that a string of unpacked BCD digits with zeros in the most significant 4- bits were to be converted to bit combinations to be used for lighting a seven-segment display. The arrangement of the seven segments in the display are shown in figure 5.4(a) along with the symbols for representing the segments. Figure 5.4(b) summarizes the code conversion needed to *form* the various digits on the display. In the figure, bit 0 of the resulting code corresponds to segment a of the display, bit 1 to segment b, and so on. It is seen that a string such as

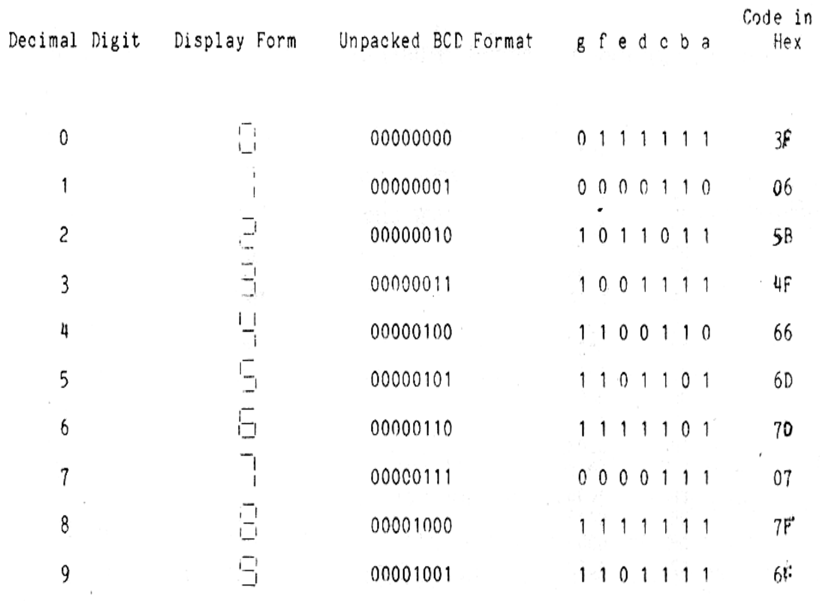
05 07 09 00 00 01 03 04

which corresponds to the number 57900134, would produce the output string

SD 07 SF 3F 3F OS 4F 66



(a) Seven-segment display unit



Decimal Digit	Display Form	Unpacked BCD Format	g f e d c b a	Code in Hex
0		00000000	0 1 1 1 1 1 1	3F
1		00000001	0 0 0 0 1 1 0	06
2		00000010	1 0 1 1 0 1 1	5B
3		00000011	1 0 0 1 1 1 1	4F
4		00000100	1 1 0 0 1 1 0	66
5		00000101	1 1 0 1 1 0 1	6D
6		00000110	1 1 1 1 1 0 1	7D
7		00000111	0 0 0 0 1 1 1	07
8		00001000	1 1 1 1 1 1 1	7F
9		00001001	1 1 0 1 1 1 1	6F

(b) Seven-segment code equivalent for BCD digits

Figure 5.4: Unpacked BCD to seven segment code conversion

Self Assessment Questions

3. LODS instruction affects flags. (True/False).
4. In assembly language the prefix is invoked by placing the appropriate repeat (REP) mnemonic before the primitive. (True/False)
5. A terminal may communicate with the computer using the EBCDIC alphanumeric code even though the computer's software is designed to work with the ASCII code, or vice versa. (True/False)
6. The XLAT instruction assumes the base address of the byte array is in the ____________ register.

5.5 Number Format Conversions

We know that computers do all the calculations using binary arithmetic, but we are accustomed to do calculations in decimal form since we are familiar with decimal number system. We see numbers printed out in the decimal form, and enter data using the keyboard as decimal numbers. Our conclusion obviously is that though arithmetic data is processed mostly in binary form by computers, there are methods to convert binary data to forms better suited for display and understanding. Also, if we want to process numeric data in a format other than binary, that should also be possible. As such, let us examine some of the number formats frequently used. Two of the most widely used formats are BCD and ASCII. BCD is 'binary coded decimal' – two versions of which are packed BCD and unpacked BCD. Terminals, printers, card readers, and other devices communicate with a computer system through a code, often the ASCII code. However, the internal arithmetic operations are not normally done with numbers in their coded formats, but usually operate on numbers in their binary or packed BCD formats. Therefore, it is necessary to convert from the external form of the data which the I/O equipment is familiar with, to the internal form, which can easily be operated on by the machine language instructions of the computer.

Although the 8086 is capable of performing arithmetic on ASCII-coded numbers directly, anything more than simple arithmetic calculations on such numbers can be quite complicated. Arithmetic using the packed BCD format is acceptable if only addition and subtraction are involved. Binary arithmetic is, of course, the easiest and fastest of the three choices. Unfortunately, to use binary arithmetic all numbers must be converted from their ASCII form to their binary form. Therefore, there is a trade-off between taking the extra time to perform the conversions to and from binary, and using more time in carrying out the arithmetic operations. The packed BCD format is a good compromise for some problems since the conversion from ASCII to packed BCD is relatively simple and action and subtraction on packed BCD numbers can be performed fairly quickly.

The entire process of inputting ASCII character strings, converting them to binary numbers, performing the necessary arithmetic, and reconverting the results to ASCII strings is shown in figure 5.5.

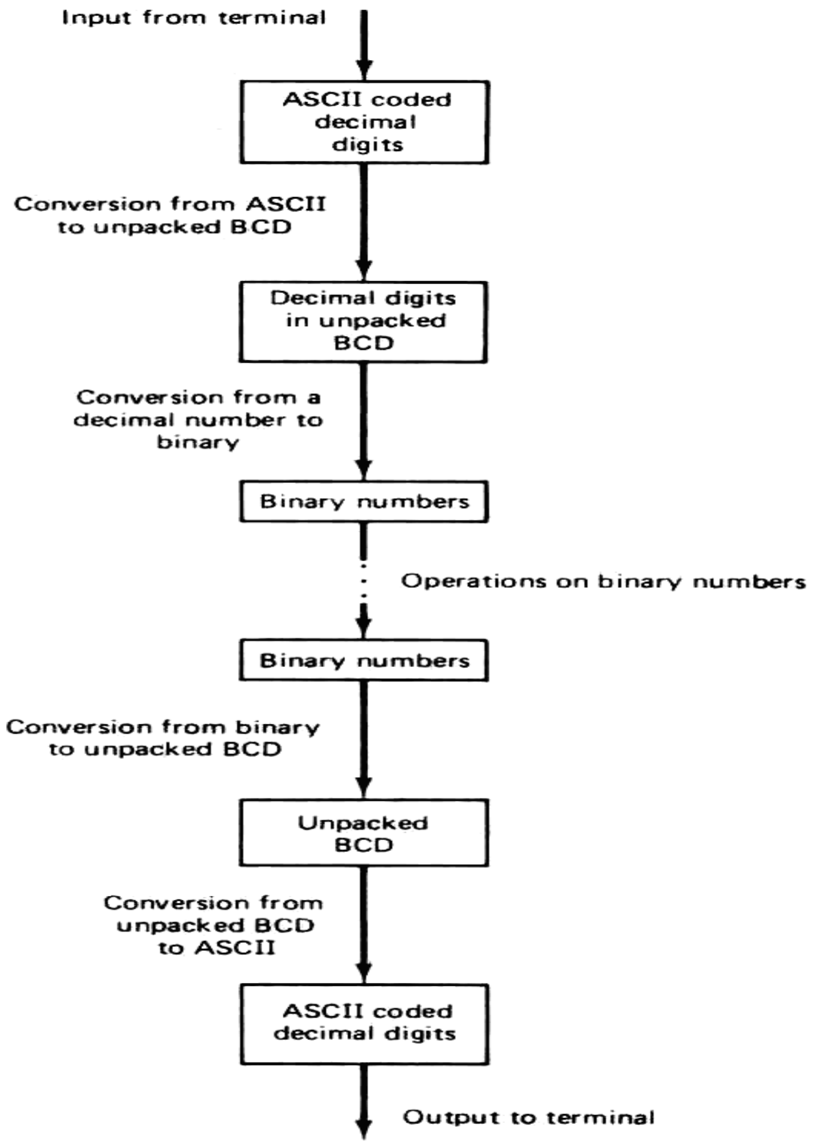

Input from terminal
ASCII coded decimal digits
Conversion from ASCII to unpacked BCD
Decimal digits in unpacked BCD
Conversion from a decimal number to binary
Binary numbers
Operations on binary numbers
Binary numbers
Conversion from binary to unpacked BCD
Unpacked BCD
Conversion from unpacked BCD to ASCII
ASCII coded decimal digits
Output to terminal

Figure 5.5: Conversion process required for inputting and outputting decimal numbers

5.6 Procedures

In high level languages (i.e., C, C++) you might have come across 'functions'. A function is a program which does a specific task. When this task is to be done repeatedly, the function is used again and again. When a 'main' program considers this as a subsidiary task, the function (or sub routine) is 'called' whenever its service is required. This is applicable to assembly language programming also. In Intel's assembly language programming terminology, this is called a 'procedure'.

A procedure (or subroutine) is a set of code that can be branched to and returned from in such a way that code is as if it was inserted at the point from which it is branched to. The Branch to a procedure is referred to as the call, and the corresponding branch back is known as the return. The return is always made to the instruction immediately following the call regardless of where the call is located. If, as shown in figure 5.6 (a), several calls are made to the same procedure, the return after each call is made to the instruction following that call. Therefore, only one copy of the procedure needs to be stored in memory even though the procedure may be called several times. When procedures are nested as shown in figure 5.6 (b), each return is made to the corresponding calling procedure, not to procedure higher in the hierarchy.

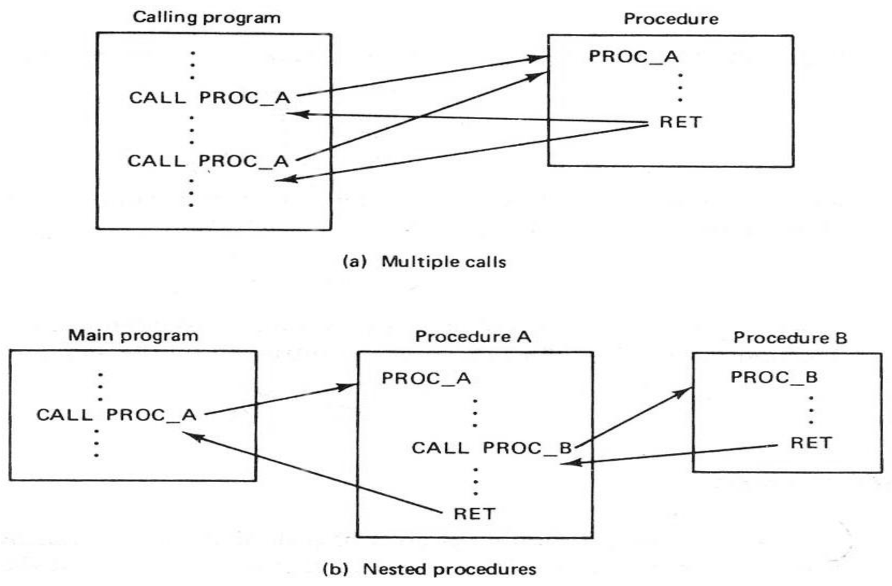


Figure 5.6: Use of procedures

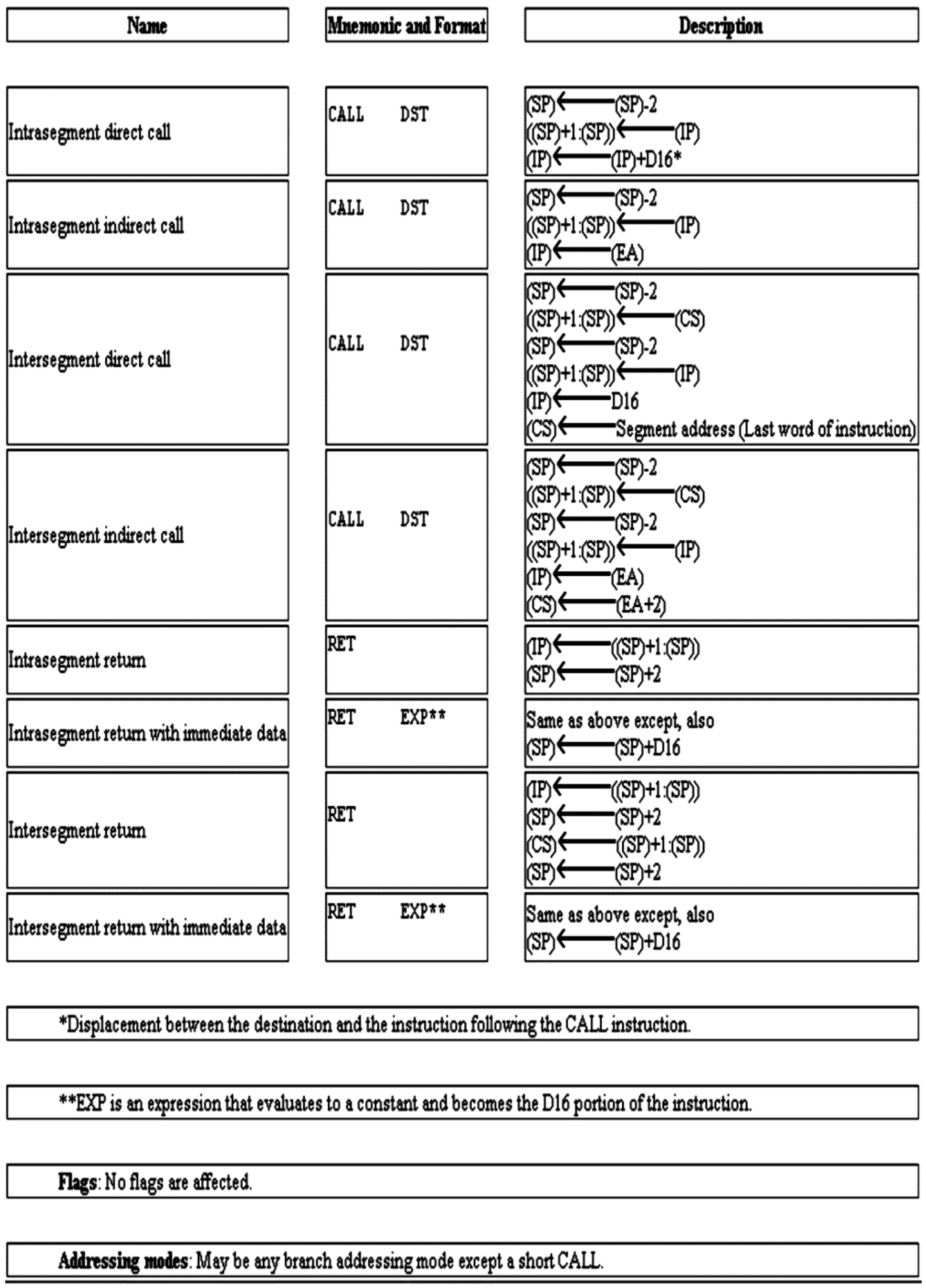
Procedures provide the primary means of breaking the code in a program into modules.

The following three requirements must be satisfied when calling a procedure:

1. Unlike other branch instructions, a procedure call must save the address of the next instruction so that the return will be able to branch back to the proper place in the calling program.
2. The registers used by the procedure need to be stored before their contents are changed, and then restored just before the procedure is exited.
3. A procedure must have a means of communicating or sharing data with the routine that calls it and other procedures.

Calls, Returns, and Procedure Definitions

The first of the above requirements is met by having special call and return branch instructions. These instructions for the 8086 are given in figure 5.7.



Name	Mnemonic and Format	Description
Intrasegment direct call	CALL DST	$(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (IP)$ $(IP) \leftarrow (IP)+D16$*
Intrasegment indirect call	CALL DST	$(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (IP)$ $(IP) \leftarrow (EA)$
Intersegment direct call	CALL DST	$(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (CS)$ $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (IP)$ $(IP) \leftarrow D16$ $(CS) \leftarrow$ Segment address (Last word of instruction)
Intersegment indirect call	CALL DST	$(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (CS)$ $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (IP)$ $(IP) \leftarrow (EA)$ $(CS) \leftarrow (EA+2)$
Intrasegment return	RET	$(IP) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP)+2$
Intrasegment return with immediate data	RET EXP**	Same as above except, also $(SP) \leftarrow (SP)+D16$
Intersegment return	RET	$(IP) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP)+2$ $(CS) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP)+2$
Intersegment return with immediate data	RET EXP**	Same as above except, also $(SP) \leftarrow (SP)+D16$

*Displacement between the destination and the instruction following the CALL instruction.

**EXP is an expression that evaluates to a constant and becomes the D16 portion of the instruction.

Flags: No flags are affected.

Addressing modes: May be any branch addressing mode except a short CALL.

Figure 5.7: Call and Return instructions

Procedures are delimited within the source code by placing a statement of the form

Procedure name PROC Attribute

At the beginning of the procedure and by terminating the procedure with a statement

Procedure name ENDP

The procedure name is the identifier used for calling the procedure and the attribute is either NEAR or FAR. The attribute is needed to determine the type of RET statement that is to be assembled. If the attribute is NEAR, the RET instruction will only pop a word into the IP register, but if it is FAR, it will also pop a word into the CS register.

A procedure may be in:

1. The same code segment as the statement that calls it.
2. A code segment that is different from the one containing the statement that calls it, but in the same source module as the calling statement.
3. A different source module and segment from the calling statement.

In the first case the attribute could be NEAR provided that all calls are in the same code segment as the procedure. For the latter two cases the attribute must be FAR. If a procedure is given a FAR attribute, then all calls to it must be intersegment calls even if the call is from the same code segment. Otherwise, the return would pop two words from the stack even though only one word was pushed onto the stack by the call. In addition, for the third case, the procedure name must be declared in EXTRN and PUBLIC statements.

Self Assessment Questions

7. Arithmetic using the ________________ format is acceptable if only addition and subtraction are involved.
8. Branch to a procedure is referred to as the _____________, and the corresponding branch back is known as the ______________.
9. If the attribute is NEAR, the RET instruction will only pop a word into the ___________ register.

5.7 Macros

The advantages of procedures are that they save memory and programming time by allowing code to be reused and provide a modularity that makes it easier to debug and modify a program. The disadvantage of procedures is the linkage associated with them. It sometimes requires more code to program the linkage than is needed to perform the task. When this is the case a procedure may not save memory and the execution time is considerably increased. For situations in which similar, but short, code segments are to appear at several places within a program, a means is needed for providing the programming ease of a procedure while avoiding the linkage. This need is fulfilled by macros. Usually, a macro, like a procedure, is defined for performing a specific function. However, the 'overheads' involved in invoking a procedure are not incurred here. A procedure call causes pushing and popping of addresses/data in stack. A macro when invoked just expands the code by putting in all the instructions corresponding to the called macro. It does not have to 'call' and 'return'. Thus, when we write our code with macros, it may look small, but when assembling the code, each macro is replaced by the full set of instructions it consists of. Thus we can say that macros execute faster but the assembled code takes more memory. This is because a procedure is written only once in memory, but the macro statements are written as part of the code every time the macro is invoked.

A *macro* is a segment of code that needs to be written only once but whose basic structure can be caused to be repeated several times within a source module by placing a single statement at the point of each appearance. A macro is unlike a procedure in that the machine instructions are repeated each time the macro is referenced; therefore, no memory is saved, but programming time is conserved (no linkage is required) and some degree of modularity is achieved.

The code that is to be repeated is called the *prototype code,* and the prototype code along with the statements for referencing and terminating it is called the *macro definition.* Included in the first statement of a macro definition is the macro's name. The procedure for using a macro is to give the macro definition and then cause the macro to be inserted at various points within a program by placing a statement that includes the macro's name at these points. These statements are known as *macro calls.* When a macro call is encountered by the assembler, the assembler replaces the call with the macro's code. This replacement action is referred to as a *macro expansion.*

In order to allow the prototype code to be used in a variety of situations, portions of the statements in this code (e.g., operands) are such that they can be replaced by different character strings each time the macro is expanded. These statement portions are called *dummy parameters* and are listed in the first statement of a macro's definition. The statement that calls the macro includes a matching list of *actual parameters* which are character strings that are to replace the dummy parameters when the macro is expanded. During a macro expansion, the first actual parameter replaces the first dummy parameter in the prototype code, the second actual parameter replaces the second dummy parameter, and so on. The correspondence of actual parameters to dummy parameters is similar to the matching of arguments to parameters when calling a subprogram in a high-level language such as FORTRAN.

ASM-86 Macro Facilities

For the ASM-86 assembler a macro definition is constructed as follows:

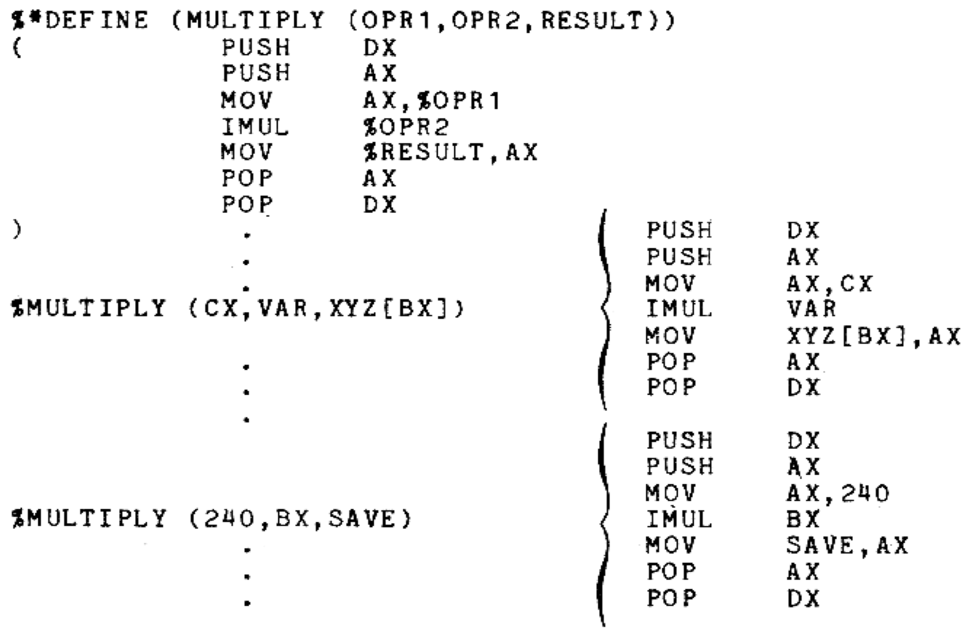
```
%*DEFINE        (Macro name (Dummy parameter list))
(
                Prototype code
)
```

where the macro name (which must be an identifier that begins with a letter and contains only letters, numbers, and underscore characters) is used to reference the macro, and the dummy parameters in the parameter list are separated by commas. ASM-86 requires that each dummy parameter appearing in the prototype code be preceded by a % character. A macro call has the form

% Macro name (Actual parameter list)
with the actual parameters being separated by commas.

As an example, the definition and use of a macro for multiplying two word - operands and storing the result, which is assumed not to exceed 16 bits, in a third operand is shown in figure 5.8. The macro definition shows that the macro first temporarily stores the contents of the AX and DX registers on the stack, carries out the multiplication, and then pops the stack so that AX and

DX are restored. The dummy parameters are OPR1, OPR2, and RESULT. In the first macro call they are replaced by CX, VAR, and XYZ[BX], respectively, and in the second call they are replaced by 240, BX, and SAVE. If a dummy parameter appears in a statement, then the matching actual parameter must be such that the statement is valid; otherwise, an assembler error will occur.



```
%*DEFINE (MULTIPLY (OPR1,OPR2,RESULT))
(            PUSH    DX
             PUSH    AX
             MOV     AX,%OPR1
             IMUL    %OPR2
             MOV     %RESULT,AX
             POP     AX
             POP     DX
)              .                        ( PUSH    DX
               .                        ( PUSH    AX
               .                        ( MOV     AX,CX
%MULTIPLY (CX,VAR,XYZ[BX])              { IMUL    VAR
                                        ( MOV     XYZ[BX],AX
               .                        ( POP     AX
               .                        ( POP     DX
               .
                                        ( PUSH    DX
                                        ( PUSH    AX
                                        ( MOV     AX,240
%MULTIPLY (240,BX,SAVE)                 { IMUL    BX
               .                        ( MOV     SAVE,AX
               .                        ( POP     AX
               .                        ( POP     DX
```

Figure 5.8: Example of use of Macro

Self Assessment Questions

10. A macro is unlike a ____________in that the machine instructions are repeated each time the macro is referenced.
11. ____________ requires that each dummy parameter appearing in the prototype code be preceded by a % character.

5.8 Summary

- String instructions help in efficient handling of strings.
- REP prefix makes a string instruction to repeat execution several times depending on the content of CX register.
- In assembly language the prefix is invoked by placing the appropriate repeat (REP) mnemonic before the primitive.

- The STOS instruction stores the AL /AX register contents to a location in the string pointed by the ES: DI pair.
- XLAT instruction helps in code conversion by accessing the codes from a look up table.
- It is necessary to convert from the external form of the data which the I/O equipment is familiar with, to the internal form, which can easily be operated on by the machine language instructions of the computer.
- A procedure (or subroutine) is a set of code that can be branched to and returned from in such a way that code is as if it were inserted at the point from which it is branched to.
- A *macro* is a segment of code that needs to be written only once but whose basic structure can be caused to be repeated several times within a source module by placing a single statement at the point of each appearance.

5.9 Terminal Questions

1. Discuss about string instructions.
2. Explain why REP Prefix is used in 8086.
3. Write a note on table translation.
4. What do you mean by procedure? Explain.
5. What do you mean by macro? Explain why is it required?

5.10 Answers

Self Assessment Questions

1. REP
2. Operands
3. False
4. True
5. True
6. BX
7. packed BCD
8. Call, Return
9. IP
10. Procedure
11. ASM-86

Terminal Questions

1. The 8086 microprocessor is equipped with special instructions to handle string operations. Refer to section 5.2 for details.
2. Because string operations inherently involve looping, the 8086 machine language includes a prefix that considerably simplifies the use of string primitives with loops. Refer to section 5.3 for details.
3. It is sometimes necessary to translate from one code to another. A terminal may communicate with the computer using the EBCDIC alphanumeric code even though the computer's software is designed to work with the ASCII code, or vice versa. Refer to section 5.4 for details.
4. A procedure (or subroutine) is a set of code that can be branched to and returned from in such a way that code is as if it were inserted at the point from which it is branched to. Refer to section 5.6 for details.
5. A *macro* is a segment of code that needs to be written only once but whose basic structure can be caused to be repeated several times within a source module by placing a single statement at the point of each appearance. Refer to section 5.7 for details.