Chapter 7 – String Operation

Computers are frequently used to manipulate characters strings as well as numeric data. In data processing applications names, addresses, and so forth must be stored and sometimes rearranged. Text editor and word processor programs must be capable of searching for and moving strings of characters.

An assembler must be able to separate assembly language statement elements, identifying those that are reserved mnemonics.

Chapter 7 – String Operation

Even when computation is primarily numerical, it is often necessary to convert either a character string to an internal numerical format when a number is entered at the keyboard or an internal format to a character string for display purposes.

An 80x86 microprocessor has instructions to manipulate character strings. The same instructions can manipulate strings of doublewords or words. This chapter covers 80x86 instructions that are used to handle strings, with emphasis on character strings. A variety of applications are given, including procedures that are similar to those in some high-level languages and the procedure called by the dtoa macro.

Chapter 7 – String Operation

7.1 Using String Instructions

Five 80x86 instructions are designed for string manipulation:

- **movs (move string)** **(The movs instruction is used to copy a string from one memory location to another.**)
- **cmps (compare string)** (The cmps instruction is designed to compare the contents of two strings.)
- **scas (scan string)** **(**The cmps instruction is designed to compare the contents of two strings.)
- **stos (store string)** **(**The scas instruction can be used to search a string for one particular value.)
- **lods (load string)** **(**The stos instruction can store a new value in some position of a string.)

Chapter 7 – String Operation

7.1 Using String Instructions

A **string** in the 80x86 architecture refers to a contiguous collection of bytes, words, or doublewords in memory.

```
Response      BYTE    20 DUP (?)
label1        BYTE    'The results are ', 0
wordString    WORD    50 DUP (?)
arrayD        DWORD   60 DUP (0)
```

Note that strings and arrays are actually the same except for the way we look at them.

Chapter 7 – String Operation

7.1 Using String Instructions

Each string instruction applies to a source string, a destination string, or both.

The bytes, words, or doublewords of these strings are processed one at a time by the string instruction.

Register indirect addressing is used to locate the individual byte, word, or doubleword elements.

The 80x86 instructions access elements of the source string using the address in the source index register ESI.

Since the source and destination addresses of string elements are always given by ESI and EDI, respectively, no operands are needed to identify these locations. Without any operand, however, the assembler cannot tell the size of the string element to be used.

Chapter 7 – String Operation

7.1 Using String Instructions

For example, just movs by itself could say to move a byte, a word, or a doubleword. The Microsoft Macro Assembler offers two ways around this dilemma.

- The first method is to use destination and source operands; these are ignored except that MASM notes their type (both operands must be the same type) and uses that element size.

- The second method is to use special versions of the mnemonics that define the element size-instructions that operate on bytes use a *b* suffix, word string instructions use a *w* suffix, and doubleword string instructions use a *d* suffix.

Chapter 7 – String Operation

7.1 Using String Instructions

Although a string instruction operates on only one string element at a time, it always gets ready to operate on the next element. It does this by changing the source index register ESI and/or the destination index register EDI to contain the address of the next element of the string(s).

When byte-size elements are being used, the index registers are changed by one; for words, ESI and EDI are changed by two; and for doublewords, the registers are changed by four.

The 80x86 can move either forward through a string, from lower to higher addresses, or backward, from higher to lower addresses.

The movement direction is determined by the value of the direction flag DF, bit 10 of the EFLAGS register. If DF is set to 1, then the addresses in ESI and EDI are decremented by string instructions, causing right to left string operations. If DF is clear (0), then the values in ESI and EDI are incremented by string instructions, so that strings are processed left to right.

Chapter 7 – String Operation

7.1 Using String Instructions

The 80x86 has two instructions whose sole purpose is to reset or set the direction flag DF.

- The cld instruction clears DF to 0 so that ESI and EDI are incremented by string instructions and strings are processed left to right.
- The std instruction sets DF to 1 so that strings are processed backward.

	Clock Cycles				
Operand	**386**	**486**	**Pentium**	**Number of Bytes**	**opcode**
cld	2	2	2	1	FC
std	2	2	2	1	FD

Figure 7.1 cld and std instructions

7.1 Using String Instructions

Finally it is time to present all the details about a string instruction.

The move string instruction movs transfers one string element (byte, word, or doubleword) from a source string to a destination string.

The source element at address DS:ESI is copied to address ES:EDI.

After the string element is copied, both index registers are changed by the element size (1, 2, or 4), incremented if the direction flag DF is 0 or decremented if DF is 1.

The movs instruction does not affect any flag. It comes in movsb, movsw, and movsd versions; Fig. 7.2 gives information about each form.

7.1 Using String Instructions

Operand	Element Size	Clock Cycles 386	486	Pentium	Number of Bytes	opcode
movsb	Byte	7	7	4	1	A4
movsw	Word	7	7	4	1	A5
movsd	doubleword	7	7	4	1	A5

Figure 7.2 movs instructions (use ESI and EDI)

Figure 7.3 gives an example of a program that uses the movs instruction. The important part of the example is the procedure *strcopy*. This procedure has two parameters passed on the stack, which give the destination and source addresses of byte or character strings. The source string is assumed to be null-terminated. Procedure *strcopy* produces an exact copy of the source string at the destination location, terminating the destination string by a null byte.

Chapter 7 – String Operation

7.2 Repeat Prefixes and More String Instructions

Each 80x86 string instruction operates on one string element at a time. However, the 80x86 architecture includes three repeat prefixes that change the string instructions into versions that repeat automatically either for a fixed number of iterations or until some condition is satisfied.

The three repeat prefixes actually correspond to two different single-byte codes; these are not themselves instructions, but supplement machine codes for the primitive string instructions, making new instructions.

Figure 7.4 shows two program fragments, each of which copies a fixed number of characters from *sourceStr* to *destStr*. The number of characters is loaded into the ECX register from *count*. The code in part (a) uses a loop. Since the count of characters might be zero, the loop is guarded by a jecxz instruction. The body of the loop uses movsb to copy one character at a time. The loop instruction takes care of counting loop iterations. The program fragment in part (b) is functionally equivalent to the one in part (a). After the count is copied into ECX, it uses the repeat prefix rep with a movsb instruction; the rep movsb instruction does the same thing as the last four lines in part (a).

Chapter 7 – String Operation

7.2 Repeat Prefixes and More String Instructions

```
        lea    esi,sourceStr       ;source string
        lea    edi,destStr         ;destination
        cld                        ;forward movement
        mov    ecx,count           ;count of characters to copy
        jecxz  endCopy             ;skip loop if count is zero
copy:   movsb                      ;move 1 character
        loop   copy                ;decrement count and continue
endCopy:
```

(a) movsbiterated in a loop

figure 7.4 Copying a fixed number of characters of a string

7.2 Repeat Prefixes and More String Instructions

```
lea     esi,sourceStr       ;source string
lea     edi,destStr         ;destination
cld                         ;forward movement
mov     ecx,count           ;count of characters to copy
rep     movsb               ;move characters
```

(b) repeat prefix with movsb

figure 7.4 Copying a fixed number of characters of a string

Chapter 7 – String Operation

7.2 Repeat Prefixes and More String Instructions

The rep prefix is normally used with the movs instructions and with the stos instruction. It causes the following design to be executed:

```
while count in ECX > 0
    loop perform primitive instruction;
    decrement ECX by 1;
end while;
```

Note that this is a *while* loop. The primitive instruction is not executed at all if ECX contains zero. It is not necessary to guard a repeated string instruction as it often is with an ordinary for loop implemented with the loop instruction.

7.2 Repeat Prefixes and More String Instructions

The other two repeat prefixes are

- repe, with equivalent mnemonic repz, and

  "repeat while equal“, "repeat while zero."

- repne, which is the same as repnz

  "repeat while not equal“, "repeat while not zero“

Each of these repeat prefixes is appropriate for use with the two string instructions cmps and scas, which affect the zero flag ZF.

Each instruction works the same as rep, iterating a primitive instruction while ECX is not zero. However, each also examines ZF after the string instruction is executed. The repe and repz continue iterating while ZF=1. The repne and repnz continue iterating while ZF=0. Repeat prefixes themselves do not affect any flag.

7.2 Repeat Prefixes and More String Instructions

The three repeat prefixes are summarized in Fig. 7.5. Note that rep and repz (repe) generate exactly the same code.

Operand	Element Size	Number of Bytes	opcode
rep	ECX>0	1	F3
repz/repe	ECX>0 and ZF=1	1	F3
repnz/repne	ECX>0 and ZF=0	1	F2

Figure 7.5 Repeat prefixes

The repz and repnz prefixes do not quite produce true *while* loops with the conditions shown in Fig. 7.5. The value in ECX is checked *prior* to the first iteration of the primitive instruction, as it should be with a *while* loop. However, ZF is not checked until *after* the primitive instruction is executed.

7.2 Repeat Prefixes and More String Instructions

Figure 7.6 shows how the repeat prefix rep combines with the movs instructions.

		Clock Cycles				
Operand	**Element Size**	**386**	**486**	**Pentium**	**Number of Bytes**	**opcode**
movsb	Byte	7	7	4	1	A4
movsw	Word	7	7	4	1	A5
movsd	doubleword	7	7	4	1	A5

Figure 7.6 rep movs instructions

7.2 Repeat Prefixes and More String Instructions

The cmps instructions, summarized in Fig. 7.7, compare elements of source and destination strings.

Operand	Element Size	Clock Cycles 386	486	Pentium	Number of Bytes	opcode
cmpsb	Byte	10	8	5	1	A6
cmpsw	Word					A7
cmpsd	doubleword	5+9n	7+7n	9+4n	2	F7
repe cmpsb	Byte					F3 A6
repe cmpsw	Word					F3 A7
repe cmpsd	doubleword					F3 A7
repne cmpsb	Byte	5+9n	7+7n	9+4n	2	F2 A6
repne cmpsw	Word					F2 A7
repne cmpsd	Doubleword					F2 A7

Fig. 7.7 cmps instructions

7.2 Repeat Prefixes and More String Instructions

It is often necessary to search for one string embedded in another.

```
position := 1;
while position < (targetLength - keyLength + 1) loop
      if key matches the substring of target starting at position
      then
            report success;
            exit process;
      end if;
      add 1 to position;
end while;
report failure;
```

Chapter 7 – String Operation

7.2 Repeat Prefixes and More String Instructions

This algorithm checks to see if the key string matches the portion of the target string starting at each possible position. Using 80x86 registers, checking for one match can be done as follows:

```
ESI := address of key;
EDI := address of target + position - 1;
ECX := length of key;

forever loop
     if ECX = 0 then exit loop; end if;
     compare [ESI] and [EDI] setting ZF;
     increment ESI;
     increment EDI;
     decrement ECX;
     if ZF = 0 then exit loop; end if;
end loop;

if ZF = 1
Then
      match was found;
end if;
```

7.2 Repeat Prefixes and More String Instructions

The scan string instruction scas is used to scan a string for the presence or absence of a particular string element.

The string that is examined is a destination string; that is, the address of the element being examined is in the destination index register EDI.

- With a scasb instruction, the element searched for is the byte in the AL register;
- with a scasw, it is the word in the AX register; and
- with a scasd, it is the doubleword in the EAX register.

The scasb, scasw, and scasd instructions use no operand since the mnemonics tell the element size.

Chapter 7 – String Operation

7.2 Repeat Prefixes and More String Instructions

Figure 7.9 summarizes the scas instructions; as with the previous repeated instructions

Operand	Element Size	Clock Cycles 386	486	Pentium	Number of Bytes	opcode
scasb	Byte	7	6	4	1	A6
scasw	Word					A7
scasd	doubleword					F7
repe scasb	Byte	5+8n	7+5n	9+4n	2	F3 A6
repe scasw	Word					F3 A7
repe scasd	doubleword					F3 A7
repne scasb	Byte	5+8n	7+5n	9+4n	2	F2 A6
repne scasw	Word					F2 A7
repne scasd	Doubleword					F2 A7

Fig. 7.9 scas instructions (use EDI)

7.2 Repeat Prefixes and More String Instructions

The program shown in Fig. 7.10 inputs a string and a character and uses repne scasb to locate the position of the first occurrence of the character in the string.

It then displays the part of the string from the character to the end.

The length of the string is calculated using the *strlen* procedure that previously appeared in Fig. 7.8; this time we assume that *strlen* is separately assembled.

Chapter 7 – String Operation

7.2 Repeat Prefixes and More String Instructions

The store string instruction stos copies a byte, a word, or a doubleword from AL, AX, or EAX to an element of a destination string. A stos instruction affects no flag, so that when it is repeated with rep, it copies the same value into consecutive positions of a string.

For example, the following code will store spaces in the first 30 bytes of string.

```
mov     ecx,30          ;30 bytes
mov     al,' '          ;character to store
lea     edi,string      ;address of string
cld                     ;forward direction
rep stosb               ;store spaces
```

7.2 Repeat Prefixes and More String Instructions

Information about the stos instructions is in Fig. 7.11. As with previous repeated string instructions.

Operand	Element Size	Clock Cycles 386	486	Pentium	Number of Bytes	opcode
stosb	Byte	7	5	3	1	AA
stosw	Word					AB
stosd	doubleword					FB
rep stosb	Byte	5+5n	7+4n	9n	2	F3 A6
rep stosw	Word					F3 A6
rep stosd	doubleword					F3 A7

Fig. 7.11 stos instructions (use EDI)

7.2 Repeat Prefixes and More String Instructions

The load string instruction lods is the final string instruction.

This instruction copies a source string element to the AL, AX, or EAX register, depending on the string element size. A lods instruction sets no flag.

A lods instruction is useful in a loop set up with other instructions, making it possible to easily process string elements one at a time.

The lods instructions are summarized in Fig. 7.12. Repeated versions are not included since they are not used.

Operand	Element Size	Clock Cycles 386	486	Pentium	Number of Bytes	opcode
lodsb	Byte	5	5	2	1	AC
lodsw	Word					AD
lodsd	doubleword					FD

Fig. 7.12 lods instructions (use EDI)

Chapter 7 – String Operation

7.3 Character Translation

Sometimes character data are available in one format but need to be in another format for processing. One instance of this occurs when characters are transmitted between two computer systems, one normally using ASCII character codes and the other normally using EBCDIC character codes.

The 80x86 instruction set includes the xlat instruction to translate one character to another character. In combination with other string-processing instructions, it can easily translate all the characters in a string.

The xlat instruction requires only one byte of object code, the opcode D7. Prior to execution, the character to be translated is in the AL register. The instruction works by using a translation table in the data segment to look up the translation of the byte in AL. This translation table normally contains 256 bytes of data, one for each possible 8-bit value in AL.
The xlat instruction has no operand. The EBX register must contain the address of the translation table.

Chapter 7 – String Operation

7.3 Character Translation

```
; Translate uppercase letters to lowercase; don't change lower
; case letters and digits. Translate other characters to spaces.
; author: R. Detmer revised: 10/97

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
INCLUDE io.h
PUBLIC _start
cr EQU 0dh ; carriage return character
Lf EQU 0ah ; linefeed character

.STACK 4096 ; reserve 4096-byte stack

.DATA
String    BYTE      'This is a #!$& STRING',0
strLength EQU       $ - string - 1
Label1    BYTE      'Original string ->',0
Label2    BYTE      cr, Lf, 'Translated string ->',0
Crlf      BYTE      cr, Lf, 0
Table     BYTE      48 DUP (' '), '0123456789', 7 DUP (' ')
          BYTE      'abcdefghijklmnopqrstuvwxyz', 6 DUP (' ')
          BYTE      'abcdefghijklmnopqrstuvwxyz', 133 DUP (' ')
```

Chapter 7 – String Operation

7.3 Character Translation

```
.CODE
_start: output  label1          ;display original string
        output  string
        output  crlf
        mov     ecx, strLength  ;string length
        lea     ebx,table       ;address of translation table
        lea     esi,string      ;address of string
        lea     edi,string      ;destination also string
forIndex:
        lodsb                   ;copy next character to AL
        xlat                    ;translate character
        stosb                   ;copy character back into string
        loop    forIndex        ;repeat for all characters
        output  label2          ;display altered string
        output  string
        output  crlf
        INVOKE ExitProcess, 0
        END
```

7.3 Character Translation

Figure 7.14 shows the output of the program in Fig. 7.13. Notice that "strange" characters are not deleted, they are replaced by blanks.

```
Original string ->     This is a #!$& STRING
Translated string ->   this is a      string
```

Chapter 7 – String Operation

7.4 Converting a 2's Complement Integer to an ASCII String

The dtoa and itoa macros have been used to convert 2's complement integers to strings of ASCII characters for output. The code for these operations is similar. In this section we examine the slightly shorter code for itoa.

The itoa macro expands into the following sequence of instructions.

```
push    ebx             ;save EBX
mov     bx,source
push    bx              ;source parameter
lea     ebx,dest        ;destination address
push    ebx             ;destination parameter
call    itoaproc        ;call itoaproc(source,dest)
pop     ebx             ;restore EBX
```

Chapter 7 – String Operation

7.4 Converting a 2's Complement Integer to an ASCII String

These instructions call procedure *itoaproc* after pushing the source value and the destination address on the stack. The actual source and destination are used in the expanded macro, not the names *source* and *dest*. So that the user does not need to worry about any register contents being altered, EBX is initially saved on the stack and is restored at the end of the sequence.

The parameters are removed from the stack by procedure *itoaproc* since the alternative add esp,6 following the call instruction potentially changes the flags.

The real work of 2's complement integer to ASCII conversion is done by the procedure *itoaproc*. The assembled version of this procedure is contained in the file IO.OBJ. The source code from file IO.ASM is shown in Fig. 7.15.

7.5 Other Architectures: CISC versus RISC Designs

Early digital computers had very simple instruction sets. When designers began to use microcode to implement instructions in the 1960s, it became possible to have much more complex instructions. At the same time high-level programming languages were becoming popular, but language compilers were fairly primitive. This made it desirable to have machine language statements that almost directly implemented high-level language statements, increasing the pressure to produce computer architectures with many complex instructions.

The Intel 80x86 machines use complex instruction set computer (CISC) designs. Instructions such as the string instructions discussed in this chapter would never have appeared in early computers. CISC machines also have a variety of memory addressing modes, and the 80x86 family is typical in this respect, although you have only seen a few of its modes so far. Often CISC instructions take several clock cycles to execute.

7.5 Other Architectures: CISC versus RISC Designs

Reduced instruction set computer (RISC) designs began to appear in the 1980s. These machines have relatively few instructions and few memory addressing modes. Their instructions are so simple that any one can be executed in a single clock cycle. As compiler technology improved, it became possible to produce efficient code for RISC machines. Of course, it often takes many more instructions to implement a given high-level language statement on a RISC than on a CISC machine, but the overall operation is often faster because of the speed with which individual instructions execute.

In RISC architectures, instructions are all the same format; that is, the same number of bytes are encoded in a common pattern. This is not the case with CISC architectures. If the 80x86 chips were RISC designs, then this book would have no questions asking "How many clock cycles?" or "How many bytes?"

7.5 Other Architectures: CISC versus RISC Designs

One unusual feature of many RISC designs is a relatively large collection of registers (sometimes over 500), of which only a small number (often 32) are visible at one time. Registers are used to pass parameters to procedures, and the registers that are used to store arguments in the calling program overlap the registers that are used to receive the parameter values in the procedure. This provides a simple but very efficient method of communication between a calling program and a procedure.

There are proponents of both CICS and RISC designs. At this point in time it is not obvious that one is clearly better than the other. However, the popular Intel 80x86 and Motorola 680x0 families are both CISC designs, so we will be dealing with CISC systems at least in the near future.

Chapter 7 – String Operation

Chapter Summary

The word string refers to a collection of consecutive bytes, words, or doublewords in memory. The 80x86 instruction set includes five instructions for operating on strings: movs (to move or copy a string from a source to a destination location), cmps (to compare two strings), scas (to scan a string for a particular element), stos (to store a given value in a string), and lods (to copy a string element into EAX, AX, or AL). Each of these has mnemonic forms ending with *b*, *w*, or *d* to give the size of the string element.

A string instruction operates on one string element at a time. When a source string is involved, the source index register ESI contains the address of the string element. When a destination string is involved, the destination index register EDI contains the address of the string element. An index register is incremented or decremented after the string element is accessed, depending on whether the direction flag DF is reset to zero or set to one; the cld and std instructions are used to give the direction flag a desired value.

Chapter 7 – String Operation

Chapter Summary

Repeat prefixes rep, repe (repz), and repne (repnz) are used with some string instructions to cause them to repeat automatically. The number of times to execute a primitive instruction is placed in the ECX register. The conditional repeat forms use the count in ECX but will also terminate instruction execution if the zero flag gets a certain value; these are appropriate for use with the cmps and scas instructions that set or reset ZF.

The xlat instruction is used to translate the characters of a string. It requires a 256-byte-long translation table that starts with the destination byte to which the source byte 00 is translated and ends with the destination byte to which the source byte FF is translated. The xlat instruction can be used for such applications as changing ASCII codes to EBCDIC codes or for changing the case of letters within a given character coding system.

The itoa macro expands to code that calls a procedure *itoaproc*. Basically this procedure works by repeatedly dividing a non-negative number by 10 and using the remainder to get the rightmost character of the destination string.