The String Instructions Chapter Six

6.1 Chapter Overview

A string is a collection of objects stored in contiguous memory locations. Strings are usually arrays of bytes, words, or (on 80386 and later processors) double words. The 80x86 microprocessor family supports several instructions specifically designed to cope with strings. This chapter explores some of the uses of these string instructions.

The 80x86 CPUs can process three types of strings: byte strings , word strings, and double word strings. They can move strings, compare strings, search for a specific value within a string, initialize a string to a fixed value, and do other primitive operations on strings. The 80x86's string instructions are also useful for manipulating arrays, tables, and records. You can easily assign or compare such data structures using the string instructions. Using string instructions may speed up your array manipulation code considerably.

6.2 The 80x86 String Instructions

All members of the 80x86 family support five different string instructions: MOVS*x*, CMPS*x*, SCAS*x*, LODS*x*, and STOS*x*[1]. (*x*= B, W, or D for byte, word, or double word, respectively. This text will generally drop the x suffix when talking about these string instructions in a general sense.) They are the string primitives since you can build most other string operations from these five instructions. How you use these five instructions is the topic of the next several sections.

```
For MOVS:
    movsb();
    movsw();
    movsd();

For CMPS:
    cmpsb();   // Note: repz is a synonym for repe
    cmpsw();
    cmpsd();

    cmpsb();   // Note: repnz is a synonym for repne.
    cmpsw();
    cmpsd();

For SCAS:
    scasb();   // Note: repz is a synonym for repe
    scasw();
    scasd();

    scasb();   // Note: repnz is a synonym for repne.
    scasw();
    scasd();

For STOS:
    stosb();
    stosw();
    stosd();
```

1. The 80x86 processor support two additional string instructions, INS and OUTS which input strings of data from an input port or output strings of data to an output port. We will not consider these instructions since they are privileged instructions and you cannot execute them in a standard 32-bit OS application.


```
For LODS:
    lodsb();
    lodsw();
    lodsd();
```

6.2.1 How the String Instructions Operate

The string instructions operate on blocks (contiguous linear arrays) of memory. For example, the MOVS instruction moves a sequence of bytes from one memory location to another. The CMPS instruction compares two blocks of memory. The SCAS instruction scans a block of memory for a particular value. These string instructions often require three operands, a destination block address, a source block address, and (optionally) an element count. For example, when using the MOVS instruction to copy a string, you need a source address, a destination address, and a count (the number of string elements to move).

Unlike other instructions which operate on memory, the string instructions don't have any explicit operands. The operands for the string instructions include

- the ESI (source index) register,
- the EDI (destination index) register,
- the ECX (count) register,
- the AL/AX/EAX register, and
- the direction flag in the FLAGS register.

For example, one variant of the MOVS (move string) instruction copies a string from the source address specified by ESI to the destination address specified by EDI, of length ECX. Likewise, the CMPS instruction compares the string pointed at by ESI, of length ECX, to the string pointed at by EDI.

Not all instructions have source and destination operands (only MOVS and CMPS support them). For example, the SCAS instruction (scan a string) compares the value in the accumulator (AL, AX, or EAX) to values in memory.

6.2.2 The REP/REPE/REPZ and REPNZ/REPNE Prefixes

The string instructions, by themselves, do not operate on strings of data. The MOVS instruction, for example, will move a single byte, word, or double word. When executed by itself, the MOVS instruction ignores the value in the ECX register. The repeat prefixes tell the 80x86 to do a multi-byte string operation. The syntax for the repeat prefix is:

```
For MOVS:
    rep.movsb();
    rep.movsw();
    rep.movsd();

For CMPS:
    repe.cmpsb();   // Note: repz is a synonym for repe.
    repe.cmpsw();
    repe.cmpsd();

    repne.cmpsb();   // Note: repnz is a synonym for repne.
    repne.cmpsw();
    repne.cmpsd();

For SCAS:
    repe.scasb();   // Note: repz is a synonym for repe.
    repe.scasw();
    repe.scasd();
```



```
    repne.scasb();   // Note: repnz is a synonym for repne.
    repne.scasw();
    repne.scasd();


For STOS:
    rep.stosb();
    rep.stosw();
    rep.stosd();
```

You don't normally use the repeat prefixes with the LODS instruction.

When specifying the repeat prefix before a string instruction, the string instruction repeats ECX times[2]. Without the repeat prefix, the instruction operates only on a single byte, word, or double word.

You can use repeat prefixes to process entire strings with a single instruction. You can use the string instructions, without the repeat prefix, as string primitive operations to synthesize more powerful string operations.

6.2.3 The Direction Flag

Besides the ESI, EDI, ECX, and AL/AX/EAX registers, one other register controls the 80x86's string instructions – the flags register. Specifically, the *direction flag* in the flags register controls how the CPU processes strings.

If the direction flag is clear, the CPU increments ESI and EDI after operating upon each string element. For example, if the direction flag is clear, then executing MOVS will move the byte, word, or double word at ESI to EDI and will increment ESI and EDI by one, two, or four. When specifying the REP prefix before this instruction, the CPU increments ESI and EDI for each element in the string. At completion, the ESI and EDI registers will be pointing at the first item beyond the strings.

If the direction flag is set, then the 80x86 decrements ESI and EDI after processing each string element. After a repeated string operation, the ESI and EDI registers will be pointing at the first byte or word before the strings if the direction flag was set.

The direction flag may be set or cleared using the CLD (clear direction flag) and STD (set direction flag) instructions. When using these instructions inside a procedure, keep in mind that they modify the machine state. Therefore, you may need to save the direction flag during the execution of that procedure. The following example exhibits the kinds of problems you might encounter:

```
procedure Str2; nodisplay;
begin Str2;

       std();
   <Do some string operations>
        .
        .
        .
end Str2;
        .
        .
        .
       cld();
   <do some operations>
       Str2();
   <do some string operations requiring D=0>
```

2. Except for the cmps instruction which repeats *at most* the number of times specified in the cx register.

This code will not work properly. The calling code assumes that the direction flag is clear after *Str2* returns. However, this isn't true. Therefore, the string operations executed after the call to *Str2* will not function properly.

There are a couple of ways to handle this problem. The first, and probably the most obvious, is always to insert the CLD or STD instructions immediately before executing a sequence of one or more string instructions. The other alternative is to save and restore the direction flag using the PUSHFD and POPFD instructions. Using these two techniques, the code above would look like this:

Always issuing CLD or STD before a string instruction:

```
procedure Str2; nodisplay;
begin Str2;

        std();
    <Do some string operations>
         .
         .
         .
end Str2;
         .
         .
         .
        cld();
    <do some operations>
        Str2();
        cld();
    <do some string operations requiring D=0>
```

Saving and restoring the flags register:

```
procedure Str2; nodisplay;
begin Str2;

        pushfd();
        std();
    <Do some string operations>
         .
         .
         .
        popfd();
end Str2;
         .
         .
         .
        cld();
    <do some operations>
        Str2();
    <do some string operations requiring D=0>
```

If you use the PUSHFD and POPFD instructions to save and restore the flags register, keep in mind that you're saving and restoring all the flags. Therefore, such subroutines cannot return any information in the flags. For example, you will not be able to return an error condition in the carry flag if you use PUSHFD and POPFD.

6.2.4 The MOVS Instruction

The MOVS instruction uses the following syntax:

```
movsb()
```



```
movsw()
movsd()
rep.movsb()
rep.movsw()
rep.movsd()
```

The MOVSB (move string, bytes) instruction fetches the byte at address ESI, stores it at address EDI and then increments or decrements the ESI and EDI registers by one. If the REP prefix is present, the CPU checks ECX to see if it contains zero. If not, then it moves the byte from ESI to EDI and decrements the ECX register. This process repeats until ECX becomes zero.

The MOVSW (move string, words) instruction fetches the word at address ESI, stores it at address EDI and then increments or decrements ESI and EDI by two. If there is a REP prefix, then the CPU repeats this procedure as many times as specified in ECX.

The MOVSD instruction operates in a similar fashion on double words. Incrementing or decrementing ESI and EDI by four for each data movement.

When you use the *rep* prefix, the MOVSB instruction moves the number of bytes you specify in the ECX register. The following code segment copies 384 bytes from *CharArray1* to *CharArray2*:

```
CharArray1: byte[ 384 ];
CharArray2: byte[ 384 ];
     .
     .
     .
    cld();
    lea( esi, CharArray1 );
    lea( edi, CharArray2 );
    mov( 384, ecx );
    rep.movsb();
```

If you substitute MOVSW for MOVSB, then the code above will move 384 words (768 bytes) rather than 384 bytes:

```
WordArray1: word[ 384 ];
WordArray2: word[ 384 ];
     .
     .
     .
    cld();
    lea( esi, WordArray1 );
    lea( edi, WordArray2 );
    mov( 384, ecx );
    rep.movsw();
```

Remember, the ECX register contains the element count, not the byte count. When using the MOVSW instruction, the CPU moves the number of words specified in the ECX register. Similarly, MOVSD moves the number of double words you specify in the ECX register, not the number of bytes.

If you've set the direction flag before executing a MOVSB/MOVSW/MOVSD instruction, the CPU decrements the ESI and EDI registers after moving each string element. This means that the ESI and EDI registers must point at the end of their respective strings before issuing a MOVSB, MOVSW, or MOVSD instruction. For example,

```
CharArray1: byte[ 384 ];
CharArray2: byte[ 384 ];
     .
     .
     .
    cld();
    lea( esi, CharArray1[383] );
    lea( edi, CharArray2[383] );
```



```
mov( 384, ecx );
rep.movsb();
```

Although there are times when processing a string from tail to head is useful (see the CMPS description in the next section), generally you'll process strings in the forward direction since it's more straightforward to do so. There is one class of string operations where being able to process strings in both directions is absolutely mandatory: processing strings when the source and destination blocks overlap. Consider what happens in the following code:

```
CharArray1: byte;
CharArray2: byte[ 384 ];
     .
     .
     .
    cld();
    lea( esi, CharArray1 );
    lea( edi, CharArray2 );
    mov( 384, ecx );
    rep.movsb();
```

This sequence of instructions treats *CharArray1* and *CharArray2* as a pair of 384 byte strings. However, the last 383 bytes in the *CharArray1* array overlap the first 383 bytes in the *CharArray2* array. Let's trace the operation of this code byte by byte.

When the CPU executes the MOVSB instruction, it copies the byte at ESI (*CharArray1*) to the byte pointed at by EDI (*CharArray2*). Then it increments ESI and EDI, decrements ECX by one, and repeats this process. Now the ESI register points at *CharArray1+1* (which is the address of *CharArray2*) and the EDI register points at *CharArray2+1*. The MOVSB instruction copies the byte pointed at by ESI to the byte pointed at by EDI. However, this is the byte originally copied from location *CharArray1*. So the MOVSB instruction copies the value originally in location *CharArray1* to both locations *CharArray2* and *CharArray2+1*. Again, the CPU increments ESI and EDI, decrements ECX, and repeats this operation. Now the movsb instruction copies the byte from location *CharArray1+2* (*CharArray2+1*) to location *CharArray2*+2. But once again, this is the value that originally appeared in location *CharArray1*. Each repetition of the loop copies the next element in *CharArray1[0]* to the next available location in the *CharArray2* array. Pictorially, it looks something like that shown in Figure 6.1.

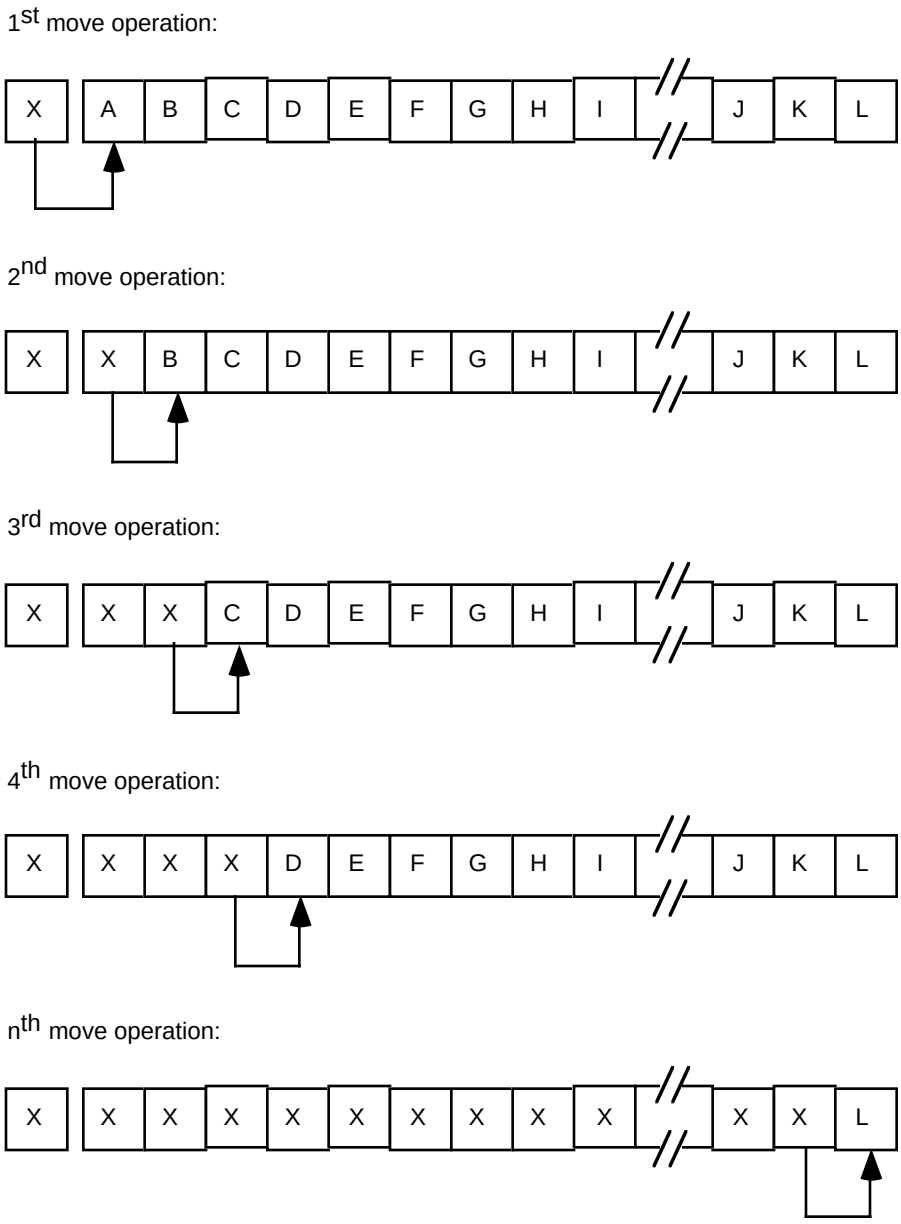

Figure 6.1 Copying Data Between Two Overlapping Arrays (forward direction)

The end result is that the MOVSB instruction replicates *X* throughout the string. The MOVSB instruction copies the source operand into the memory location which will become the source operand for the very next move operation, which causes the replication.

If you really want to move one array into another when they overlap, you should move each element of the source string to the destination string starting at the end of the two strings as shown in Figure 6.2.

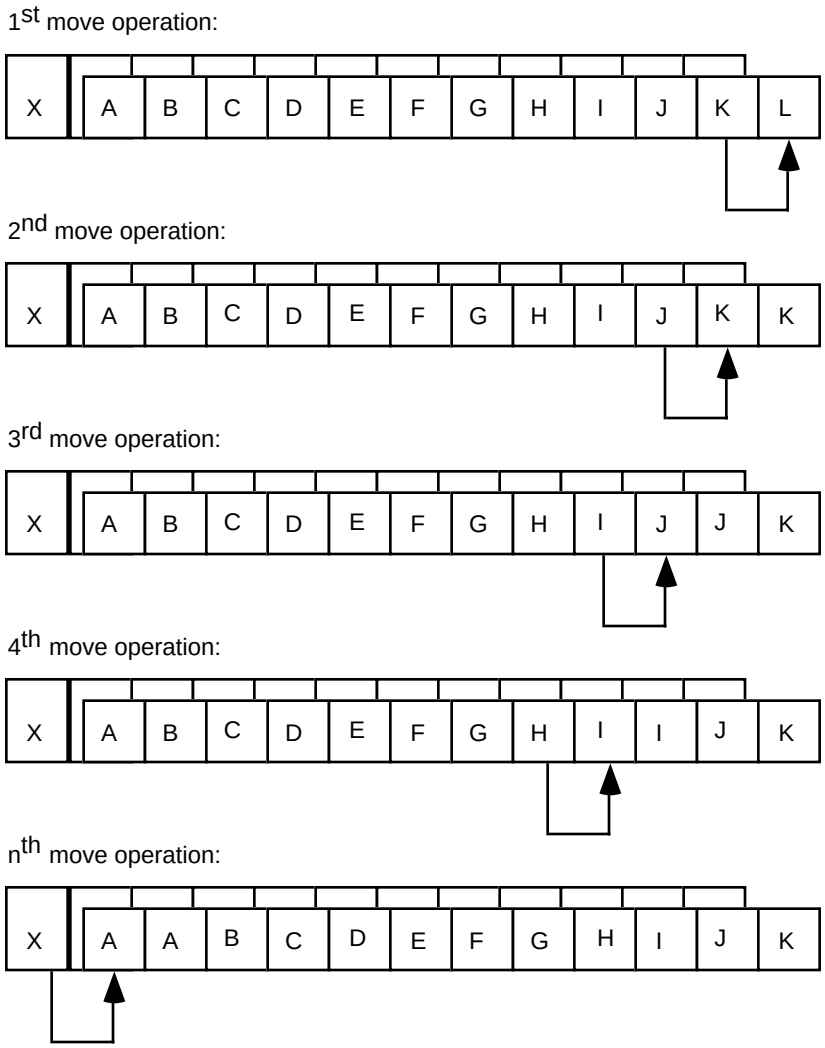

Figure 6.2 Using a Backwards Copy to Copy Data in Overlapping Arrays

Setting the direction flag and pointing ESI and EDI at the end of the strings will allow you to (correctly) move one string to another when the two strings overlap and the source string begins at a lower address than the destination string. If the two strings overlap and the source string begins at a higher address than the destination string, then clear the direction flag and point ESI and EDI at the beginning of the two strings.

If the two strings do not overlap, then you can use either technique to move the strings around in memory. Generally, operating with the direction flag clear is the easiest, so that makes the most sense in this case.

You shouldn't use the MOVS*x* instruction to fill an array with a single byte, word, or double word value. Another string instruction, STOS, is much better for this purpose. However, for arrays whose elements are larger than four bytes, you can use the MOVS instruction to initialize the entire array to the content of the first element.

The MOVS instruction is generally more efficient when copying double words than it is copying bytes or words. In fact, it typically takes the same amount of time to copy a byte using MOVSB as it does to copy a double word using MOVSD[3]. Therefore, if you are moving a large number of bytes from one array to another, the copy operation will be faster if you can use the MOVSD instruction rather than the MOVSB

3. This is true for MOVSW, as well.

instruction. Of course, if the number of bytes you wish to move is an even multiple of four, this is a trivial change; just divide the number of bytes to copy by four, load this value into ECX, and then use the MOVSB instruction. If the number of bytes is not evenly divisible by four, then you can use the MOVSD instruction to copy all but the last one, two, or three bytes of the array (that is, the remainder after you divide the byte count by four). For example, if you want to efficiently move 4099 bytes, you can do so with the following instruction sequence:

```
lea( esi, Source );
lea( edi, Destination );
mov( 1024, ecx );          // Copy 1024 dwords = 4096 bytes
rep.movsd();
movsw();                   // Copy bytes 4097 and 4098.
movsb();                   // Copy the last byte.
```

Using this technique to copy data never requires more than three MOVS*x* instructions since you can copy one, two, or three bytes with no more than two MOVSB and MOVSW instructions. The scheme above is most efficient if the two arrays are aligned on double word boundaries. If not, you might want to move the MOVSB or MOVSW instruction (or both) before the MOVSD so that the MOVSD instruction works with dword-aligned data (see Chapter Three for an explanation of the performance benefits of double word aligned data).

If you do not know the size of the block you are copying until the program executes, you can still use code like the following to improve the performance of a block move of bytes:

```
lea( esi, Source );
lea( edi, Dest );
mov( Length, ecx );
shr( 2, ecx );       // divide by four.
if( @nz ) then       // Only execute MOVSD if four or more bytes.

    rep.movsd();      // Copy the dwords.

endif;
mov( Length, ecx );
and( %11, ecx );     // Compute (Length mod 4).
if( @nz ) then       // Only execute MOVSB if #bytes/4 <> 0.

    rep.movsb();      // Copy the remaining one, two, or three bytes.

endif;
```

On most computer systems, the MOVSD instruction provides about the fastest way to copy bulk data from one location to another. While there are, arguably, faster ways to copy the data on certain CPUs, ultimately the memory bus performance is the limiting factor and the CPUs are generally much faster than the memory bus. Therefore, unless you have a special system, writing fancy code to improve memory to memory transfers is probably a waste of time. Also note that Intel has improved the performance of the MOVSx instructions on later processors so that MOVSB operates almost as efficiently as MOVSW and MOVSD when copying the same number of bytes. Therefore, when working on a later x86 processor, it may be more efficient to simply use MOVSB to copy the specified number of bytes rather than go through all the complexity outlined above.

6.2.5 The CMPS Instruction

The CMPS instruction compares two strings. The CPU compares the string referenced by EDI to the string pointed at by ESI. ECX contains the length of the two strings (when using the REPE or REPNE prefix). Like the MOVS instruction, HLA allows several different forms of this instruction:

```
cmpsb();
cmpsw();
```



```
cmpsd();

repe.cmpsb();
repe.cmpsw();
repe.cmpsd();

repne.cmpsb();
repne.cmpsw();
repne.cmpsd();
```

Like the MOVS instruction you specify the actual operand addresses in the ESI and EDI registers.

Without a repeat prefix, the CMPS instruction subtracts the value at location EDI from the value at ESI and updates the flags. Other than updating the flags, the CPU doesn't use the difference produced by this subtraction. After comparing the two locations, CMPS increments or decrements the ESI and EDI registers by one, two, or four (for CMPSB/CMPSW/CMPSD, respectively). CMPS increments the ESI and EDI registers if the direction flag is clear and decrements them otherwise.

Of course, you will not tap the real power of the CMPS instruction using it to compare single bytes, words, or double words in memory. This instruction shines when you use it to compare whole strings. With CMPS, you can compare consecutive elements in a string until you find a match or until consecutive elements do not match.

To compare two strings to see if they are equal or not equal, you must compare corresponding elements in a string until they don't match. Consider the following strings:

"String1"

"String1"

The only way to determine that these two strings are equal is to compare each character in the first string to the corresponding character in the second. After all, the second string could have been "String2" which definitely is not equal to "String1". Of course, once you encounter a character in the destination string which doesn't equal the corresponding character in the source string, the comparison can stop. You needn't compare any other characters in the two strings.

The REPE prefix accomplishes this operation. It will compare successive elements in a string as long as they are equal and ECX is greater than zero. We could compare the two strings above using the following 80x86 assembly language code:

```
cld();
mov( AdrsString1, esi );
mov( AdrsString2, edi );
mov( 7, ecx );
repe.cmpsb();
```

After the execution of the CMPSB instruction, you can test the flags using the standard conditional jump instructions. This lets you check for equality, inequality, less than, greater than, etc.

Character strings are usually compared using *lexicographical ordering*. In lexicographical ordering, the least significant element of a string carries the most weight. This is in direct contrast to standard integer comparisons where the most significant portion of the number carries the most weight. Furthermore, the length of a string affects the comparison only if the two strings are identical up to the length of the shorter string. For example, "Zebra" is less than "Zebras", because it is the shorter of the two strings, however, "Zebra" is greater than "AAAAAAAAAAH!" even though it is shorter. Lexicographical comparisons compare corresponding elements until encountering a character which doesn't match, or until encountering the end of the shorter string. If a pair of corresponding characters do not match, then this algorithm compares the two strings based on that single character. If the two strings match up to the length of the shorter string, we must compare their length. The two strings are equal if and only if their lengths are equal and each corresponding pair of characters in the two strings is identical. Lexicographical ordering is the standard alphabetical ordering you've grown up with.

For character strings, use the CMPS instruction in the following manner:

- The direction flag must be cleared before comparing the strings.
- Use the CMPSB instruction to compare the strings on a byte by byte basis. Even if the strings contain an even number of characters, you cannot use the CMPSW or CMPSD instructions. They do not compare strings in lexicographical order.
- You must load the ECX register with the length of the smaller string.
- Use the REPE prefix.
- The ESI and EDI registers must point at the very first character in the two strings you want to compare.

After the execution of the CMPS instruction, if the two strings were equal, their lengths must be compared in order to finish the comparison. The following code compares a couple of character strings:

```
mov( AdrsStr1, esi );
mov( AdrsStr2, edi );
mov( LengthSrc, ecx );
if( ecx > LengthDest ) then  // Put the length of the shorter string in ECX.

    mov( LengthDest, ecx );

endif;
repe.cmpsb();
if( @z ) then  // If equal to the length of the shorter string, cmp lengths.

    mov( LengthSrc, ecx );
    cmp( ecx, LengthDest );

endif;
```

If you're using bytes to hold the string lengths, you should adjust this code appropriately (i.e., use a MOVZX instruction to load the lengths into ECX). Of course, HLA strings use a double word to hold the current length value, so this isn't an issue when using HLA strings.

You can also use the CMPS instruction to compare multi-word integer values (that is, extended precision integer values). Because of the amount of setup required for a string comparison, this isn't practical for integer values less than six or eight double words in length, but for large integer values, it's an excellent way to compare such values. Unlike character strings, we cannot compare integer strings using a lexicographical ordering. When comparing strings, we compare the characters from the least significant byte to the most significant byte. When comparing integers, we must compare the values from the most significant byte (or word/double word) down to the least significant byte, word or double word. So, to compare two 32-byte (256-bit) integer values, use the following code on the 80x86:

```
std();
lea( esi, SourceInteger[28] );
lea( edi, DestInteger[28] );
mov( 8, ecx );
rep.cmpsd();
```

This code compares the integers from their most significant word down to the least significant word. The CMPSD instruction finishes when the two values are unequal or upon decrementing ECX to zero (implying that the two values are equal). Once again, the flags provide the result of the comparison.

The REPNE prefix will instruct the CMPS instruction to compare successive string elements as long as they do not match. The 80x86 flags are of little use after the execution of this instruction. Either the ECX register is zero (in which case the two strings are totally different), or it contains the number of elements compared in the two strings until a match. While this form of the CMPS instruction isn't particularly useful for comparing strings, it is useful for locating the first pair of matching items in a couple of byte, word, or double word arrays. In general, though, you'll rarely use the REPNE prefix with CMPS.

One last thing to keep in mind with using the CMPS instruction – the value in the ECX register determines the number of elements to process, not the number of bytes. Therefore, when using CMPSW, ECX

specifies the number of words to compare. This, of course, is twice the number of bytes to compare. Likewise, for CMPSD, ECX contains the number of double words to process.

6.2.6 The SCAS Instruction

The CMPS instruction compares two strings against one another. You do not use it to search for a particular element within a string. For example, you could not use the CMPS instruction to quickly scan for a zero throughout some other string. You can use the SCAS (scan string) instruction for this task.

Unlike the MOVS and CMPS instructions, the SCAS instruction only requires a destination string (pointed at by EDI) rather than both a source and destination string. The source operand is the value in the AL (SCASB), AX (SCASW), or EAX (SCASD) register. The SCAS instruction compares the value in the accumulator (AL, AX, or EAX) against the value pointed at by EDI and then increments (or decrements) EDI by one, two, or four. The CPU sets the flags according to the result of the comparison. While this might be useful on occasion, SCAS is a lot more useful when using the REPE and REPNE prefixes.

With the REPE prefix (repeat while equal), SCAS scans the string searching for an element which does not match the value in the accumulator. When using the REPNE prefix (repeat while not equal), SCAS scans the string searching for the first string element which is equal to the value in the accumulator.

You're probably wondering "why do these prefixes do exactly the opposite of what they ought to do?" The paragraphs above haven't quite phrased the operation of the SCAS instruction properly. When using the REPE prefix with SCAS, the 80x86 scans through the string while the value in the accumulator is equal to the string operand. This is equivalent to searching through the string for the first element which does not match the value in the accumulator. The SCAS instruction with REPNE scans through the string while the accumulator is not equal to the string operand. Of course, this form searches for the first value in the string which matches the value in the accumulator register. The SCAS instructions take the following forms:

```
scasb()
scasw()
scasd()

repe.scasb()
repe.scasw()
repe.scasd()

repne.scasb()
repne.scasw()
repne.scasd()
```

Like the CMPS and MOVS instructions, the value in the ECX register specifies the number of elements to process, not bytes, when using a repeat prefix.

6.2.7 The STOS Instruction

The STOS instruction stores the value in the accumulator at the location specified by EDI. After storing the value, the CPU increments or decrements EDI depending upon the state of the direction flag. Although the STOS instruction has many uses, its primary use is to initialize arrays and strings to a constant value. For example, if you have a 256-byte array you want to clear out with zeros, use the following code:

```
cld();
lea( edi, DestArray );
mov( 64, ecx );        // 64 double words = 256 bytes.
xor( eax, eax );       // Zero out EAX.
rep.stosd();
```

This code writes 64 double words rather than 256 bytes because a single STOSD operation is faster than four STOSB operations.

The STOS instructions take four forms. They are

```
stosb();
stosw();
stosd();

rep.stosb();
rep.stosw();
rep.stosd();
```

The STOSB instruction stores the value in the AL register into the specified memory location(s), the STOSW instruction stores the AX register into the specified memory location(s) and the STOSD instruction stores EAX into the specified location(s).

Keep in mind that the STOS instruction is useful only for initializing a byte, word, or double word array to a constant value. If you need to initialize an array to different values, you cannot use the STOS instruction. See the exercises for additional details.

6.2.8 The LODS Instruction

The LODS instruction is unique among the string instructions. You will probably never use a repeat prefix with this instruction. The LODS instruction copies the byte, word, or double word pointed at by ESI into the AL, AX, or EAX register, after which it increments or decrements the ESI register by one, two, or four. Repeating this instruction via the repeat prefix would serve no purpose whatsoever since the accumulator register will be overwritten each time the LODS instruction repeats. At the end of the repeat operation, the accumulator will contain the last value read from memory.

Instead, use the LODS instruction to fetch bytes (LODSB), words (LODSW), or double words (LODSD) from memory for further processing. By using the STOS instruction, you can synthesize powerful string operations.

Like the STOS instruction, the LODS instructions take four forms:

```
lodsb();
lodsw();
lodsd();

rep.lodsb();
rep.lodsw();
rep.lodsd();
```

As mentioned earlier, you'll rarely, if ever, use the REP prefixes with these instructions[4]. The 80x86 increments or decrements ESI by one, two, or four depending on the direction flag and whether you're using the LODSB, LODSW, or LODSD instruction.

6.2.9 Building Complex String Functions from LODS and STOS

The 80x86 supports only five different string instructions: MOVS, CMPS, SCAS, LODS, and STOS[5]. These certainly aren't the only string operations you'll ever want to use. However, you can use the LODS and STOS instructions to easily generate any particular string operation you like. For example, suppose you wanted a string operation that converts all the upper case characters in a string to lower case. You could use the following code:

```
mov( StringAddress, esi );  // Load string address into ESI.
mov( esi, edi );            // Also point EDI here.
```

4. They appear here simply because they are allowed. They're not veryuseful, but they are allowed.
5. Not counting INS and OUTS which we're ignoring here.


```
mov( (type str.strrec [esi].length, ecx );

repeat

    lodsb();             // Get the next character in the string.
    if( al in 'A'..'Z' ) then

        or( $20, al );   // Convert upper case character to lower case.

    endif;
    stosb();            // Store converted character back into string.
    dec( ecx );

until( ecx == 0 );
```

Since the LODS and STOS instructions use the accumulator as an intermediary, you can use any accumulator operation to quickly manipulate string elements.

6.3 Putting It All Together

In this chapter we took a quick look at the 80x86's string instructions. We studied their implementation and saw how to use them. These instructions are quite useful for synthesizing character set functions (see the source code for the HLA Standard Library string module for examples). We also saw how to use these instructions for non-character string purpose such as moving large blocks of memory (i.e., assigning one array to another) and comparing large integer values. For more information on the use of these instructions, please see the volume on Advanced String Handling.

