10

Working with Strings

For humans, written communication is crucial. Unfortunately, computers were not specifically designed for ease of communication with humans. Most high-level languages provide specific functions for helping programmers write programs to interact with humans. As an assembly language programmer, you do not have the luxury of these functions. It is up to you to code your programs to interact with humans in their own language.

The use of strings helps programs communicate with humans in their own language. While using strings is not a simple matter in assembly language programming, it is not impossible. This chapter guides you through working with strings. The first section discusses the instructions used to move strings around in memory by copying them from one memory location to another. The next section shows how, similar to integers, strings can be compared for equality. The last section describes the instructions that are used to scan strings for a search character or character string. This feature comes in handy when searching for specific characters within text.

> *Note that the string instructions presented in this chapter also can be applied to nonstring data. Moving, modifying, and comparing blocks of numerical data can also be accomplished using the IA-32 string instructions.*

Moving Strings

One of the most useful functions when dealing with strings is the capability to copy a string from one memory location to another. If you remember from Chapter 5, "Moving Data," you cannot use the `MOV` instruction to move data from one memory location to another.

Fortunately, Intel has created a complete family of instructions to use when working with string data. This section describes the IA-32 string moving instructions, and shows how to use them in your programs.

The MOVS instruction

The `MOVS` instruction was created to provide a simple way for programmers to move string data from one memory location to another. There are three formats of the `MOVS` instruction:

- **MOVSB:** Moves a single byte
- **MOVSW:** Moves a word (2 bytes)
- **MOVSL:** Moves a doubleword (4 bytes)

*The Intel documentation uses **MOVSD** for moving a doubleword. The GNU assembler decided to use **MOVSL**.*

The `MOVS` instructions use implied source and destination operands. The implied source operand is the `ESI` register. It points to the memory location for the source string. The implied destination operand is the `EDI` register. It points to the destination memory location to which the string is copied. The obvious way to remember this is that the `S` in `ESI` stands for source, and the `D` in `EDI` stands for destination.

With the GNU assembler, there are two ways to load the `ESI` and `EDI` values. The first way is to use indirect addressing (see Chapter 5). By placing a dollar sign in front of the memory location label, the address of the memory location is loaded into the `ESI` or `EDI` registers:

```
movl $output, %edi
```

This instruction moves the 32-bit memory location of the `output` label to the `EDI` register.

Another method of specifying the memory locations is the `LEA` instruction. The `LEA` instruction loads the effective address of an object. Because Linux uses 32-bit values to reference memory locations, the memory address of the object must be stored in a 32-bit destination value. The source operand must point to a memory location, such as a label used in the .data section. The instruction

```
leal output, %edi
```

loads the 32-bit memory location of the `output` label to the EDI register.

The following `movstest1.s` program uses the `MOVS` instruction to move some strings:

```
# movstest1.s - An example of the MOVS instructions
.section .data
value1:
   .ascii "This is a test string.\n"
.section .bss
   .lcomm output, 23
.section .text
.globl _start
_start:
```

```
    nop
    leal value1, %esi
    leal output, %edi
    movsb
    movsw
    movsl

    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

The `movstest1.s` program loads the location of the `value1` memory location into the `ESI` register, and the location of the `output` memory location into the `EDI` register. When the `MOVSB` instruction is executed, it moves 1 byte of data from the `value1` location to the `output` location. Because the output variable was declared in the .bss section, any string data placed there will automatically be null terminated.

The interesting thing is what happens when the `MOVSW` instruction is executed. If you run the program in the debugger, you can see the output after each of the `MOVS` instructions:

```
(gdb) s
13          movsb
(gdb) s
14          movsw
(gdb) x/s &output
0x80490b0 <output>:      "T"
(gdb) s
15          movsl
(gdb) x/s &output
0x80490b0 <output>:      "Thi"
(gdb) s
17          movl $1, %eax
(gdb) x/s &output
0x80490b0 <output>:      "This is"
(gdb)
```

The `MOVSB` instruction moved the "T" from the `value1` location to the `output` location as expected. However, without changing the `ESI` and `EDI` registers, when the `MOVSW` instruction ran, instead of moving the "Th" (the first 2 bytes of the string), it moved the "hi" from the `value1` location to the `output` location. Then the `MOVSL` instruction continues by adding the next 4 byte values. There is a reason for this.

Each time a `MOVS` instruction is executed, when the data is moved, the `ESI` and `EDI` registers are automatically changed in preparation for another move. While this is usually a good thing, sometimes it can be somewhat tricky.

One of the tricky parts of this operation is the direction in which the registers are changed. The `ESI` and `EDI` registers can be either automatically incremented or automatically decremented, depending on the value of the `DF` flag in the `EFLAGS` register.

If the `DF` flag is cleared, the `ESI` and `EDI` registers are incremented after each `MOVS` instruction. If the `DF` flag is set, the `ESI` and `EDI` registers are decremented after each `MOVS` instruction. Because the

movstest.s program did not specifically set the DF flag, we are at the mercy of the current setting. To ensure that the DF flag is set in the proper direction, you can use the following commands:

- CLD to clear the DF flag
- STD to set the DF flag

When the STD instruction is used, the ESI and EDI registers are decremented after each MOVS instruction, so they should point to the end of the string locations instead of the beginning. The movstest2.s program demonstrates this:

```
# movstest2.s - A second example of the MOVS instructions
.section .data
value1:
   .ascii "This is a test string.\n"
.section .bss
   .lcomm output, 23
.section .text
.globl _start
_start:
   nop
   leal value1+22, %esi

   leal output+22, %edi

   std
   movsb
   movsw
   movsl

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

This time, the address location of the value1 memory location is placed in the EAX register, the length of the test string (minus one because it starts at address 0) is added to it, and the value is placed in the ESI register. This causes the ESI register to point to the end of the test string. The same is done for the EDI register, so it points to the end of the output memory location. The STD instruction is used to set the DF flag so the ESI and EDI registers are decremented after each MOVS instruction.

The three MOVS instructions move 1, 2, and 4 bytes of data between the two string locations. However, this time, there is a difference. After the three MOVS instructions have been executed, you can look at the end of the output memory location using the debugger:

```
(gdb) x/23b &output
0x80490b8 <output>:    0x00   0x00   0x00   0x00   0x00   0x00   0x00  0x00
0x80490c0 <output+8>:  0x00   0x00   0x00   0x00   0x00   0x00   0x00  0x00
0x80490c8 <output+16>: 0x00   0x00   0x00   0x6e   0x67   0x2e   0x0a
(gdb)
```

Notice that the output string was beginning to fill in from the end of the string, but after the three MOVS instructions, only four memory locations have been filled in. In the movstest.s program, which ran forward, seven memory locations were filled in using the same three instructions. Why is that?

The answer relates to how the values are copied. Even though the `ESI` and `EDI` registers are counting backward, the `MOVW` and `MOVL` instructions are getting the memory locations in forward order. When the `MOVSB` instruction finished, it decremented the `ESI` and `EDI` registers one, but the `MOVSW` instruction gets two memory locations. Likewise, when the `MOVSW` instruction finished, it decremented the `ESI` and `EDI` registers two, but the `MOVSL` instruction gets four memory locations. This is demonstrated in Figure 10-1.

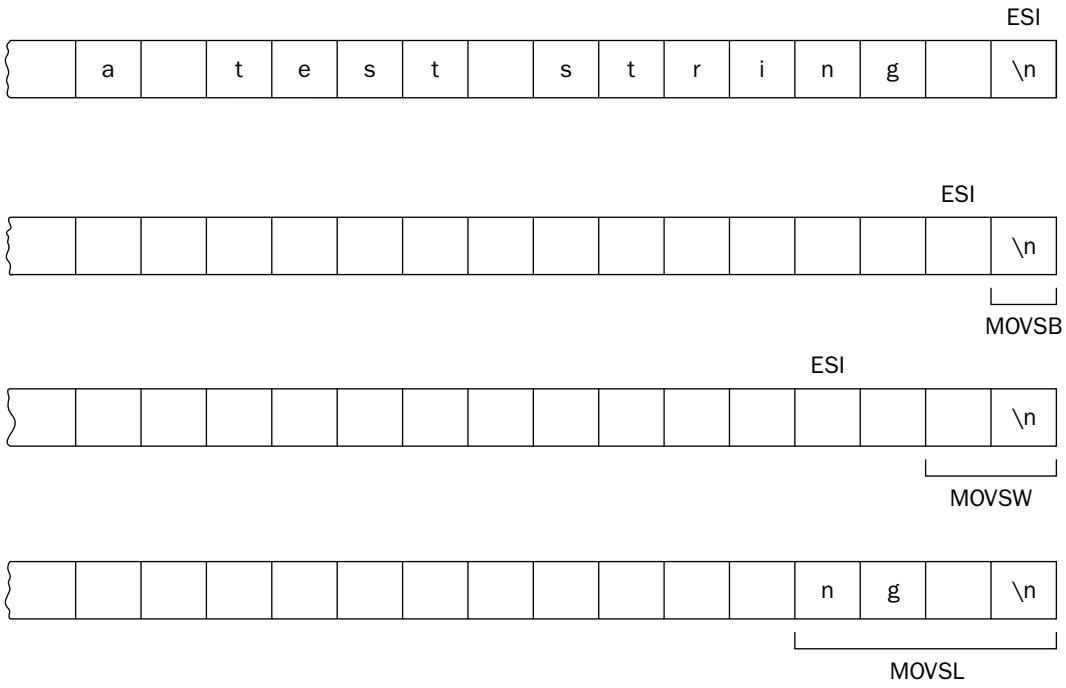


Figure 10-1

Of course, the way to solve this problem is to consistently use the same size blocks to move for every instruction. If all three instructions were `MOVSW` instructions, the `ESI` value would be decremented by two each time, and 2 bytes would be moved.

> *It is important to remember that if you use the* ***`STD`*** *instruction to work backward from a string, the* ***`MOVSW`*** *and* ***`MOVSL`*** *instructions are still working forward in retrieving memory locations.*

If you are copying a large string, it is easy to see that it could take a lot of `MOVL` instructions to get all of the data. To make things easier, you might be tempted to put the `MOVL` instruction in a loop, controlled by the `ECX` register set to the length of the string. The `movstest3.s` program demonstrates this:

```
# movstest3.s - An example of moving an entire string
.section .data
value1:
   .ascii "This is a test string.\n"
.section .bss
   .lcomm output, 23
```

```
.section .text
.globl _start
_start:
   nop
   leal value1, %esi
   leal output, %edi
   movl $23, %ecx
   cld
loop1:
   movsb
   loop loop1

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

The `ESI` and `EDI` registers are loaded as before with the source and destination memory locations. The `ECX` register is loaded with the length of the string to move. The loop section continually performs the `MOVSB` instruction until the entire string has been moved. Viewing the string value at the output memory location can check this:

```
(gdb) x/s &output
0x80490b0 <output>:      "This is a test string.\n"
(gdb)
```

While this method works, Intel has provided a simpler way to do this: using the `REP` instruction.

The REP prefix

The `REP` instruction is special in that it does nothing by itself. It is used to repeat a string instruction a specific number of times, controlled by the value in the `ECX` register, similar to using a loop, but without the extra `LOOP` instruction. The `REP` instruction repeats the string instruction immediately following it until the value in the `ECX` register is zero. That is why it is called a prefix.

Moving a string byte by byte

The `MOVSB` instruction can be used with the `REP` instruction to move a string 1 byte at a time to another location. The `reptest1.s` program demonstrates how this works:

```
# reptest1.s - An example of the REP instruction
.section .data
value1:
   .ascii "This is a test string.\n"
.section .bss
   .lcomm output, 23
.section .text
.globl _start
_start:
   nop
   leal value1, %esi
   leal output, %edi
```

```
movl $23, %ecx
cld
rep movsb

movl $1, %eax
movl $0, %ebx
int $0x80
```

The size of the string to move is loaded into the `ECX` register, and the `REP` instruction is used with the `MOVSB` instruction to move a single byte of data 23 times (the length of the string). When you step through the program in the debugger, the `REP` instruction still only counts as one instruction step, not 23. You should see the following output from the debugger:

```
$ gdb -q reptest1
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file reptest1.s, line 11.
(gdb) run
Starting program: /home/rich/palp/chap10/reptest1

Breakpoint 1, _start () at reptest1.s:11
11          leal value1, %esi
Current language:  auto; currently asm
(gdb) s
12          leal output, %edi
(gdb) s
13          movl $23, %ecx
(gdb) s
14          cld
(gdb) s
15          rep movsb
(gdb) s
17          movl $1, %eax
(gdb) x/s &output
0x80490b0 <output>:      "This is a test string.\n"
(gdb)
```

While stepping through the instructions, the `REP` instruction took only one step, but after that step, all 23 bytes of the source string were moved to the destination string location.

Moving strings block by block

You are not limited to moving the strings byte by byte. You can also use the `MOVSW` and `MOVSL` instructions to move more than 1 byte per iteration.

If you are using the `MOVSW` or `MOVSL` instructions, the `ECX` register should contain the number of iterations required to walk through the string. For example, if you are moving an 8-byte string, you would need to set `ECX` to 8 if you are using the `MOVSB` instruction, to 4 if you are using the `MOVSW` instruction, or to 2 if you are using the `MOVSL` instruction.

When walking through the string using `MOVSW` or `MOVSL`, be careful that you do not overstep the string boundaries. If you do, look at what happens in the `reptest2.s` program:

```
# reptest2.s - An incorrect example of using the REP instruction
.section .data
value1:
   .ascii "This is a test string.\n"
value2:
   .ascii "Oops"
.section .bss
   .lcomm output, 23
.section .text
.globl _start
_start:
   nop
   leal value1, %esi
   leal output, %edi
   movl $6, %ecx
   cld
   rep movsl

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

The preceding example attempts to be resourceful by looping only six times to move blocks of 4 bytes of data. The only problem is that the total data size of the source string is not an even multiple of four. The last time the `MOVSL` instruction executes, it will not only get the end of the `value1` string, it will also erroneously pick up a byte from the next string defined. You can see this from the debugger output:

```
$ gdb -q reptest2
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file reptest2.s, line 13.
(gdb) run
Starting program: /home/rich/palp/chap10/reptest2

Breakpoint 1, _start () at reptest2.s:13
13         leal value1, %esi
Current language:  auto; currently asm
(gdb) s
14         leal output, %edi
(gdb) s
15         movl $6, %ecx
(gdb) s
16         cld
(gdb) s
17         rep movsl
(gdb) s
19         movl $1, %eax
(gdb) x/s &output
0x80490b0 <output>:      "This is a test string.\nO"
(gdb)
```

The `output` string now contains the first character from the `value2` string tacked onto the data from the `value1` string. That is not at all what we would want to have happen.

Moving large strings

Obviously, it is more efficient to move string characters using the MOVSL instruction as much as possible. The problem, as demonstrated in the reptest2.s program, is that you must know when to stop using the MOVSL instruction and convert back to the MOVSB. The trick is in matching the length of the string.

When you know the length of the string, it is easy to perform an integer division (see Chapter 8, "Basic Math Functions") to determine how many doublewords would be in the string. The remainder can then use the MOVSB instruction (which should be no more than three iterations). This is demonstrated in the following reptest3.s program:

```
# reptest3.s - Moving a large string using MOVSL and MOVSB
.section .data
string1:
   .asciz "This is a test of the conversion program!\n"
length:
   .int 43
divisor:
   .int 4
.section .bss
   .lcomm buffer, 43
.section .text
.globl _start
_start:
   nop

   leal string1, %esi
   leal buffer, %edi
   movl length, %ecx
   shrl $2, %ecx

   cld
   rep movsl
   movl length, %ecx
   andl $3, %ecx
   rep movsb

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

The reptest3.s program loads the source and destination memory locations into the ESI and EDI registers as normal, but then loads the string length value into the AX register. To divide the string length by four, the SHR instruction is used to shift the length value right 2 bits (which is the same as dividing by four), which leaves the quotient value loaded into the ECX register. The REP MOVSL instruction pair is then performed that number of times. After that completes, the remainder value is determined using a common math trick.

If the divisor is a power of two (which four is) you can quickly find the remainder by subtracting one from the divisor and ANDing it with the dividend. This value is then loaded into the ECX register, and the REP MOVSB instruction pair is performed to move the remaining characters.

You can watch this as it happens in the debugger. First, stop the program after the REP MOVSL instruction pair are performed, and display the buffer memory location contents:

```
(gdb) s
22          movl %edx, %ecx
(gdb) x/s &buffer
0x80490d8 <buffer>:      "This is a test of the conversion program"
(gdb)
```

Notice that the first 40 characters were moved from the source string to the destination string. Next, execute the REP MOVSB instruction pair, and look at the buffer memory location contents again:

```
(gdb) s
23          rep movsb
(gdb) s
25          movl $1, %eax
(gdb) x/s &buffer
0x80490d8 <buffer>:      "This is a test of the conversion program!\n"
(gdb)
```

The final two characters in the string were successfully moved. Again, try different combinations of strings and string lengths to verify that this method works properly.

Moving a string in reverse order

The REP instruction works equally well backward and forward. The DF flag can be set to work backward on a string, moving it in reverse order between memory locations. This is demonstrated in the reptest4.s program:

```
# reptest4.s - An example of using REP backwards
.section .data
value1:
   .asciz "This is a test string.\n"
.section .bss
   .lcomm output, 24
.section .text
.globl _start
_start:
   nop
   leal value1+22, %esi
   leal output+22, %edi
   movl $23, %ecx
   std
   rep movsb

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

Similar to the movstest2.s program, the reptest4.s program loads the location of the end of the source and destination strings into the ESI and EDI registers, and then sets the DF flag using the STD instruction. This causes the destination string to be stored in reverse order (although you won't be able to see that from the debugger, as the REP instruction still moves all of the bytes in one step):

```
(gdb) s
20          std
(gdb) s
21          rep movsb
(gdb) s
23          movl $1, %eax
(gdb) x/s &output
0x80490c0 <output>:      "This is a test string.\n"
(gdb)
```

Other REP instructions

While the REP instruction is handy, you can use a few other versions of it when working with strings. Besides monitoring the value of the ECX register, there are REP instructions that also monitor the status of the zero flag (ZF). The following table describes the other REP instructions that can be used.

Instruction	Description
REPE	Repeat while equal
REPNE	Repeat while not equal
REPNZ	Repeat while not zero
REPZ	Repeat while zero

The REPE and REPZ instructions are synonyms for the same instruction, and the REPNE and REPNZ instructions are synonymous.

*While the **MOVS** instructions do not lend themselves to using these **REP** variations, the comparing and scanning string functions discussed later in this chapter make extensive use of them.*

Storing and Loading Strings

Besides moving strings from one memory location to another, there are also instructions for loading string values in memory into registers, and then back into memory locations. This section describes the STOS and LODS instructions that are used for this purpose.

The LODS instruction

The LODS instruction is used to move a string value in memory into the EAX register. As with the MOVS instruction, there are three different formats of the LODS instruction:

- **LODSB:** Loads a byte into the AL register
- **LODSW:** Loads a word (2 bytes) into the AX register
- **LODSL:** Loads a doubleword (4 bytes) into the EAX register

*The Intel documents use **LODSD** for loading doublewords. The GNU assembler uses **LODSL**.*

The LODS instructions use an implied source operand of the ESI register. The ESI register must contain the memory address of the location of the string to load. The LODS instruction increments or decrements (depending on the DF flag status) the ESI register by the amount of data loaded after the data is moved.

The STOS and SCAS instructions described later in this chapter both utilize data stored in the EAX register. The LODS instruction is useful in placing string values into the EAX register for these instructions. While you can use the REP instruction to repeat LODS instructions, it is unlikely you would ever do that, as the most you can load into the EAX register is 4 bytes, which can be done with a single LODSL instruction.

The STOS instruction

After the LODS instruction is used to place a string value in the EAX register, the STOS instruction can be used to place it in another memory location. Similar to the LODS instruction, the STOS instruction has three formats, depending on the amount of data to move:

- **STOSB:** Stores a byte of data from the AL register
- **STOSW:** Stores a word (2 bytes) of data from the AX register
- **STOSL:** Stores a doubleword (4 bytes) of data from the EAX register

The STOS instruction uses an implied destination operand of the EDI register. When the STOS instruction is executed, it will either increment or decrement the EDI register value by the data size used.

The STOS instruction by itself is not too exciting. Just placing a single byte, word, or doubleword string value into a memory location is not too difficult of a task. Where the STOS instruction really comes handy is when it is used with the REP instruction to replicate a string value multiple times within a large string value — for example, copying a space character (ASCII value 0x20) to a 256-byte buffer area.

The stostest1.s program demonstrates this concept:

```
# stostest1.s - An example of using the STOS instruction
.section .data
space:
   .ascii " "
.section .bss
   .lcomm buffer, 256
.section .text
.globl _start
_start:
   nop
   leal space, %esi
   leal buffer, %edi
   movl $256, %ecx
   cld
   lodsb
   rep stosb

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

The stostest1.s program loads an ASCII space character into the AL register, then copies it 256 times into the memory locations pointed to by the buffer label. You can see the before and after values of the buffer memory locations using the debugger:

```
(gdb) s
15          lodsb
(gdb) s
16          rep stosb
(gdb) print/x $eax
$1 = 0x20
(gdb) x/10b &buffer
0x80490a0 <buffer>:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x80490a8 <buffer+8>:   0x00    0x00
(gdb) s
18          movl $1, %eax
(gdb) x/10b &buffer
0x80490a0 <buffer>:     0x20    0x20    0x20    0x20    0x20    0x20    0x20    0x20
0x80490a8 <buffer+8>:   0x20    0x20
(gdb)
```

This output shows that the space character was loaded into the AL register by the LODSB instruction. Before the STOSB instruction, the buffer memory location contained zeros (which is what we would expect because it was constructed in the .bss section). After the STOSB instruction, the buffer contained all spaces.

Building your own string functions

The STOS and LODS instructions come in handy for a variety of string operations. By pointing the ESI and EDI registers to the same string, you can perform simple functions on the string. You can use the LODS instruction to walk your way through a string, load each character one at a time into the AL register, perform some operation on that character, and then load the new character back into the string using the STOS instruction.

The convert.s program demonstrates this principle by converting an ASCII string into all capital letters:

```
# convert.s - Converting lower to upper case
.section .data
string1:
   .asciz "This is a TEST, of the conversion program!\n"
length:
   .int 43
.section .text
.globl _start
_start:
   nop
   leal string1, %esi
   movl %esi, %edi
   movl length, %ecx
   cld
```

```
loop1:
   lodsb
   cmpb $'a', %al
   jl skip
   cmpb $'z', %al
   jg skip
   subb $0x20, %al
skip:
   stosb
   loop loop1
end:
   pushl $string1
   call printf
   addl $4, %esp
   pushl $0
   call exit
```

The `convert.s` program loads the `string1` memory location into both the `ESI` and `EDI` registers, and the string length into the `ECX` register. It then uses the `LOOP` instruction to perform a character check for each character in the string. It checks the characters by loading each individual character into the `AL` register, and determining whether it is less than the ASCII value for the letter `a` (`0x61`) or greater than the ASCII value for the letter `z` (`0x7a`). If the character is within these ranges, it must be a lowercase letter that can be converted to uppercase by subtracting `0x20` from it.

Whether the character was converted or not, it must be placed back into the string to keep the `ESI` and `EDI` registers in sync. The `STOSB` instruction is run on each character, and then the code loops back for the next character until it runs out of characters in the string.

After assembling the program and linking it with the C library, you can run it to see if it works right:

```
$ ./convert
THIS IS A TEST, OF THE CONVERSION PROGRAM!
$
```

Indeed, it performed as expected. You can test the program by using different types of ASCII characters within the string (remember to change the length value to match your new string).

Comparing Strings

Moving strings from one place to another is useful, but there are other string functions that can really help out when working with strings. One of the most useful string functions available is the capability to compare strings. Many programs need to compare input values from users, or compare string records with search values. This section describes the methods used to compare string values in assembly language programs.

The CMPS instruction

The `CMPS` family of instructions is used to compare string values. As with the other string instructions, there are three formats of the `CMPS` instruction:

- **CMPSB:** Compares a byte value
- **CMPSW:** Compares a word (2 bytes) value
- **CMPSL:** Compares a doubleword (4 bytes) value

As with the other string instructions, the locations of the implied source and destination operands are again stored in the `ESI` and `EDI` registers. Each time the `CMPS` instruction is executed, the `ESI` and `EDI` registers are incremented or decremented by the amount of the data size compared, depending on the `DF` flag setting.

The `CMPS` instruction subtracts the destination string from the source string, and sets the carry, sign, overflow, zero, parity, and adjust flags in the `EFLAGS` register appropriately. After the `CMPS` instruction, you can use the normal conditional jump instructions to branch, depending on the values of the strings.

The `cmpstest1.s` program demonstrates a simple example of using the `CMPS` instruction:

```
# cmpstest1.s - A simple example of the CMPS instruction
.section .data
value1:
   .ascii "Test"
value2:
   .ascii "Test"
.section .text
.globl _start
_start:
   nop
   movl $1, %eax
   leal value1, %esi
   leal value2, %edi
   cld
   cmpsl
   je equal
   movl $1, %ebx
   int $0x80
equal:
   movl $0, %ebx
   int $0x80
```

The `cmptest1.s` program compares two string values, and sets the return code for the program depending on the result of the comparison. First the exit system call value is loaded into the `EAX` register. After loading the location of the two strings to test into the `ESI` and `EDI` registers, the `cmpstest1.s` program uses the `CMPSL` instruction to compare the first four bytes of the strings. The `JE` instruction is used to jump to the `equal` label if the strings are equal, which sets the program result code to 0 and exits. If the strings are not equal, the branch is not taken, and execution falls through to set the result code to 1 and exit.

After assembling and linking the program, you can test it by running it and checking the result code:

```
$ ./cmpstest1
$ echo $?
0
$
```

The result code was 0, indicating that the strings matched. To test this, you can change one of the strings and assemble the program again to see whether the result code changes to 1.

This technique works well for matching strings up to four characters long, but what about longer strings? The answer lies in the REP instruction, described in the next section.

Using REP with CMPS

The REP instruction can be used to repeat the string comparisons over multiple bytes, but there is a problem. Unfortunately, the REP instruction does not check the status of the flags between repetitions; remember that it is only concerned about the count value in the ECX register.

The solution is to use the other instructions in the REP family: REPE, REPNE, REPZ, and REPNZ. These instructions check the zero flag for each repetition and stop the repetitions if the zero flag is set. This enables you to check strings byte by byte to determine whether they match up. As soon as one of the character pairs does not match, the REP instruction will stop repeating.

The cmpstest2.s program demonstrates how this is accomplished:

```
# cmpstest2.s - An example of using the REPE CMPS instruction
.section .data
value1:
   .ascii "This is a test of the CMPS instructions"
value2:
   .ascii "This is a test of the CMPS Instructions"
.section .text
.globl _start
_start:
   nop
   movl $1, %eax
   lea value1, %esi
   leal value2, %edi
   movl $39, %ecx
   cld
   repe cmpsb
   je equal
   movl %ecx, %ebx
   int $0x80
equal:
   movl $0, %ebx
   int $0x80
```

The cmpstest2.s program loads the source and destination string locations into the ESI and EDI registers, as well as the string length in the ECX register. The REPE CMPSB instructions repeat the string compare byte by byte until either the ECX register runs out or the zero flag is set, indicating a nonmatch.

After the REPE instruction, the JE instruction is used as normal to check the EFLAGS registers to determine whether the strings were equal. If the REPE instruction exited, the zero flag will be set, and the JE instruction will not branch, indicating the strings were not the same. The ESI and EDI registers will then contain the location of the mismatched character in the strings, and the ECX register will contain the position of the mismatched character (counting back from the end of the string).

This example also demonstrates how sensitive the string comparisons are. The two strings differ only in the capitalization of one character, which will be detected by the comparison:

```
$ ./cmpstest2
$ echo $?
11
$
```

The `CMPS` instructions subtract the hexadecimal values of the source and destination strings. The ASCII codes for each individual character are different, so any difference between the strings will be detected.

String inequality

While on the topic of comparing strings, it is a good idea to discuss the concept of unequal strings. When comparing strings, it is easy to determine when two strings are equal. The string "test" is always equal to the string "test." However, what about the string "test1"? Should that be less than or greater than the string "test"?

Unlike integers, where it is easy to understand why the value 100 would be greater than 10, determining string inequalities is not as simple of a concept. Trying to determine whether the string "less" is less than or greater than the string "greater" is not easy to understand. If your application must determine string inequalities, you must have a specific method for determining them.

The method most often used to compare strings is called *lexicographical ordering*. This is most often referred to as *dictionary ordering*, as it is the standard by which dictionaries order words. As you page through a dictionary, you can see how the words are ordered. The basic rules for lexicographical ordering are as follows:

- Alphabetically lower letters are less than alphabetically higher letters
- Uppercase letters are less than lowercase letters

You may notice that these rules follow the standard ASCII character coding values. It is easy to apply these rules to strings that are the same length. The string `"test"` would be less than the string `"west"`, but greater than the string `"Test"`. When working with strings of different lengths, things get tricky.

When comparing two strings of different lengths, the comparison is based on the number of characters in the shorter string. If the shorter string would be greater than the same number of characters in the longer string, then the shorter string is greater than the longer string. If the shorter string would be less than the same number of characters in the longer string, then the shorter string would be less than the longer string. If the shorter string is equal to the same number of characters in the longer string, the longer string is greater than the shorter string.

Using this rule, the following examples would be true:

- `"test"` is greater than `"boomerang"`
- `"test"` is less than `"velocity"`
- `"test"` is less than `"test1"`

The `strcomp.s` program demonstrates comparing strings for both inequality and equality:

```
# strcomp.s - An example of comparing strings
.section .data
string1:
   .ascii "test"
length1:
   .int 4
string2:
   .ascii "test1"
length2:
   .int 5
.section .text
.globl _start
_start:
   nop
   lea string1, %esi
   lea string2, %edi
   movl length1, %ecx
   movl length2, %eax
   cmpl %eax, %ecx
   ja longer
   xchg %ecx, %eax
longer:
   cld
   repe cmpsb
   je equal
   jg greater
less:
   movl $1, %eax
   movl $255, %ebx
   int $0x80
greater:
   movl $1, %eax
   movl $1, %ebx
   int $0x80
equal:
   movl length1, %ecx
   movl length2, %eax
   cmpl %ecx, %eax
   jg greater
   jl less
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

The `strcomp.s` program defines two strings, `string1` and `string2`, along with their lengths (`length1` and `length2`). The result code produced from the program reflects the comparison of the two strings, shown in the following table.

Result Code	Description
255	string1 is less than string2
0	string1 is equal to string2
1	string1 is greater than string2

To perform the lexicographical ordering, first the shorter string length must be determined. This is done by loading the two string lengths into registers and using the `CMP` instruction to compare them. The shorter number is loaded into the `ECX` register for the `REPE` instruction.

Next, the two strings are compared byte by byte using the `REPE` and `CMPSB` instructions for the length of the shorter string. If the first string is greater than the second string, the program branches to the `greater` label and sets the result code to 1 and exits. If the first string is less than the second string, the program falls through and sets the result code to 255 and exits.

If the two strings are equal, there is more work to be done — the program must still determine which of the two strings is longer. If the first string is longer, then it is greater, and the program branches to the `greater` label, sets the result code to 1, and exits. If the second string is longer, then the first string is less, and the program branches to the `less` label, sets the result code to 255, and exits. If the two lengths are not greater or less, then not only did the strings match, but their lengths are equal, so the strings are equal.

You can set various strings in the `string1` and `string2` locations to test the lexicographical ordering (just remember to change the string length values accordingly). Using the values shown in the `strcomp.s` program listing, the output should be as follows:

```
$ ./strcomp
$ echo $?
255
$
```

as the first string, `"test"`, is less than the second string, `"test1"`.

Scanning Strings

Sometimes it is useful to scan a string for a specific character or character sequence. One method would be to walk through the string using the `LODS` instruction, putting each character in the `AL` register, and comparing the characters with the search character. This would certainly work, but would be a time-consuming process.

Intel provides a better way to do this with another string instruction. The `SCAS` instruction provides a way for you to scan a string looking for a specific character, or group of characters. This section describes how to use the `SCAS` instruction in your programs.

The SCAS instruction

The SCAS family of instructions is used to scan strings for one or more search characters. As with the other string instructions, there are three versions of the SCAS instruction:

- **SCASB:** Compares a byte in memory with the AL register value
- **SCASW:** Compares a word in memory with the AX register value
- **SCASL:** Compares a doubleword in memory with the EAX register value

The SCAS instructions use an implied destination operand of the EDI register. The EDI register must contain the memory address of the string to scan. As with the other string instructions, when the SCAS instruction is executed, the EDI register value is incremented or decremented (depending on the DF flag value) by the data size amount of the search character.

When the comparison is made, the EFLAGS adjust, carry, parity, overflow, sign, and zero flags are set accordingly. You can use the standard conditional branch instructions to detect the outcome of the scan.

By itself, the SCAS instruction is not too exciting. It will only check the current character pointed to by the EDI register against the character in the AL register, similar to the CMPS instruction. Where the SCAS instruction comes in handy is when it is used with the REPE and REPNE prefixes.

These two prefixes enable you to scan the entire length of a string looking for a specific search character (or character sequence). The REPE and REPNE instructions are usually used to stop the scan when the search character is found. Be careful, however, when using these two instructions, as their behavior might be opposite from what you would think:

- REPE: Scans the string characters looking for a character that does not match the search character
- REPNE: Scans the string characters looking for a character that matches the search character

For most string scans, you would use the REPNE instruction, as it will stop the scan when the search character is found in the string. When the character is found, the EDI register contains the memory address immediately after where the character is located. This is because the REPNE instruction increments the EDI register after the SCAS instruction is performed. The ECX register contains the position from the end of the string that contains the search character. Be careful with this value, as it is counted from the end of the string. To get the position from the start of the string, subtract the string length from this value and reverse the sign.

The scastest1.s program demonstrates searching for a character in a string using the REPNE and SCAS instructions:

```
# scastest1.s - An example of the SCAS instruction
.section .data
string1:
   .ascii "This is a test - a long text string to scan."
length:
   .int 44
string2:
   .ascii "-"
```

```
.section .text
.globl _start
_start:
   nop
   leal string1, %edi
   leal string2, %esi
   movl length, %ecx
   lodsb
   cld
   repne scasb
   jne notfound
   subw length, %cx
   neg %cx
   movl $1, %eax
   movl %ecx, %ebx
   int $0x80
notfound:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

The `scastest1.s` program loads the memory location of the string to scan into the `EDI` register, uses the `LODSB` instruction to load the `AL` register with the character to search for, and places the length of the string in the `ECX` register. When all of that is done, the `REPNE SCASB` instruction is used to scan the string for the location of the search character. If the character is not found, the `JNE` instruction will branch to the `notfound` label. If the character is found, its location from the end of the string is now in `ECX`. The length of the string is subtracted from `ECX`, and the `NEG` instruction is used to change the sign of the value to produce the location in the string where the search character is found. The location is loaded into the `EBX` register so it becomes the result code after the program terminates:

```
$ ./scastest1
$ echo $?
16
$
```

The output shows that the "-" character was found in position 16 of the string.

Scanning for multiple characters

While the `SCASW` and `SCASL` instructions can be used to scan for a sequence of two or four characters, care must be taken when using them. They might not perform just as you are expecting.

The `SCASW` and `SCASL` instructions walk down the string looking for the character sequence in the `AX` or `EAX` registers, but they do not perform a character-by-character comparison. Instead, the `EDI` register is incremented by either 2 (for `SCASW`) or 4 (for `SCASL`) after each comparison, not by 1. This means that the character sequence must also be present in the proper sequence within the string. The `scastest2.s` program demonstrates this problem:

```
# scastest2.s - An example of incorrectly using the SCAS instruction
.section .data
string1:
   .ascii "This is a test - a long text string to scan."
```

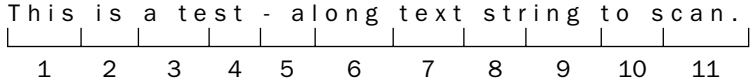
```
length:
   .int 11
string2:
   .ascii "test"
.section .text
.globl _start
_start:
   nop
   leal string1, %edi
   leal string2, %esi
   movl length, %ecx
   lodsl
   cld
   repne scasl
   jne notfound
   subw length, %cx
   neg %cx
   movl $1, %eax
   movl %ecx, %ebx
   int $0x80
notfound:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

The `scastest2.s` program attempts to find the character sequence "test" within the string. It loads the complete search string into the `EAX` register, and uses the `SCASL` instruction to check 4 bytes of the string at a time. Note that the `ECX` register is not set to the string length, but instead to the number of iterations the `REPNE` instruction needs to take to complete the string walking. Because each iteration checks 4 bytes, the `ECX` register is one-fourth the total string length of 44.

If you run the program and check the result code, you will see something that you might not expect to see:

```
$ ./scastest2
$ echo $?
0
$
```

The `SCASL` instruction did not find the character sequence `"test"` within the string. Obviously, something went wrong. That something was the way in which the `REPNE` instruction performed the iterations. Figure 10-2 demonstrates how this was done.



REPNE SCASL Iteration

Figure 10-2

The first iteration of the `REPNE` instruction checked the 4 bytes `"This"` against the character sequence in `EAX`. Because they did not match, it incremented the `ECX` register by four, and checked the next 4 bytes,

`" is "`. As you can see from Figure 10-2, no groups of 4 bytes tested matched the character sequence, even though the sequence was in the string.

> *While the* ***SCASW*** *and* ***SCASL*** *instructions don't work well with strings, they are useful when searching for nonstring data sequences in data arrays. The* ***SCASW*** *instruction can be used for 2-byte arrays, while the* ***SCASL*** *instruction can be used with 4-byte arrays.*

Finding a string length

One extremely useful function of the `SCAS` instruction is to determine the string length of zero-terminated (also called null-terminated) strings. These strings are most commonly used in C programs, but are also used in assembly language programs by using the `.asciz` declaration. With a zero-terminated string, the obvious thing to search for is the location of the zero, and count how many characters were processed looking for the zero. The `strsize.s` program demonstrates this:

```
# strsize.s - Finding the size of a string using the SCAS instruction
.section .data
string1:
   .asciz "Testing, one, two, three, testing.\n"
.section .text
.globl _start
_start:
   nop
   leal string1, %edi
   movl $0xffff, %ecx
   movb $0, %al
   cld
   repne scasb
   jne notfound
   subw $0xffff, %cx
   neg %cx
   dec %cx
   movl $1, %eax
   movl %ecx, %ebx
   int $0x80
notfound:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

The `strsize.s` program loads the memory location of the string to test into the `EDI` register and loads a fictitious string length into the `ECX` register. The `0xffff` string length value indicates that this utility will only work on strings up to 65,535 bytes in length. The `ECX` register will keep track of how many iterations it takes to find the terminating zero in the string. If the zero is found by the `SCASB` instruction, the position must be calculated from the value of the `ECX` register. Subtracting it from the original value and changing the sign of the result does this. Because the length includes the terminating zero, the final value must be decreased by one to show the actual string size. The result of the calculation is placed in the `EBX` register so it can be retrieved by checking the result code from the program:

```
$ ./strsize
$ echo $?
35
$
```

Summary

Strings are an important part of working with programs that must interact with humans. Creating, moving, comparing, and scanning strings are vital functions that must be performed within assembly language programs. This chapter presented the IA-32 string instructions and demonstrated how they can be used in assembly language programs.

The MOVS instruction family provides methods to move strings from one memory location to another. These instructions move a string referenced by the ESI register to a location referenced by the EDI register. The three formats of the MOVS instruction, MOVSB (for bytes), MOVSW (for words), and MOVSL (for doublewords) provide quick methods for moving large strings in memory. To move even more memory, the REP instruction enables the repetition of the MOVS instructions a preset number of times. The ECX register is used as a counter to determine how many iterations of the MOVS instruction will be performed. It is important to remember that the ECX register counts iterations and not necessarily the string size. The ESI and EDI registers are incremented or decremented automatically by the amount of the data size moved. The EFLAGS DF flag is used to control the direction in which the registers are changed.

The LODS instruction family is used to load a string value into the EAX register. The LODSB instruction moves a byte into the AL register, the LODSW instruction moves a word into the AX register, and the LODSL instruction moves a doubleword into the EAX register. Once a string value is placed in the register, many different functions can be performed on it, such as changing the ASCII value from a lowercase letter to an uppercase letter. After the function is applied, the string character can be moved back into memory using the STOS instruction. The STOS instructions include STOSB (store a byte), STOSW (store a word), and STOSL (store a doubleword).

A very handy function to perform with strings is the capability to compare different string values, which is made possible by the CMPS instruction. The memory locations of the strings to compare are loaded into the ESI and EDI registers. The result of the CMPS instructions is set in the EFLAGS registers, using the usual flags. This enables you to use the standard conditional branching instructions to check the EFLAGS flag settings and branch accordingly. The REPE instruction can be used to compare longer strings, comparing each individual character until two characters are not the same. The length of the strings to compare is set in the ECX register. Again, the EFLAGS registers can be used to determine whether the two strings are the same.

Besides comparing strings, it is often useful to scan a string for a specific character or character sequence. The SCAS instructions are used to scan strings for characters that are stored in the EAX register. The SCAS instruction is often used with the REPNE instruction to repeat the scan function over an entire string length. The scan will continue until the search character is found, or the end of the string is reached. The length of the string to scan is placed in the ECX register, and the results are set in the EFLAGS register.

With all of these specialized string functions, along with the special math functions presented in the previous chapters, you can start building a library of assembly language functions to use in all your programs. Chapter 11, "Using Functions," demonstrates how to separate code functions into self-contained libraries that can be used in any assembly program where they are needed. This can save you a lot of time by not having to repeat the same code functions in all of your programs.