Chapter 11: String Instruction Applications

Now that we've got a solid understanding of what the string instructions do, let's look at a few applications to get a sense of what they're particularly good for. The applications we'll look at include copying arrays, searching strings for characters, looking up entries in tables, comparing strings, and animation.

There's a lot of meat in this chapter, and a lot of useful code. The code isn't fully fleshed out, since I'm trying to illustrate basic principles rather than providing you with a library from A to Z, but that's actually all to the good. You can build on this code to meet your specific needs or write your own code from scratch once you understand the ins and outs of the string instructions. In either case, you'll be better off with code customized to suit your purposes than you would be using any one-size-fits-all code I could provide.

I'll frequently contrast the string instruction-based implementations with versions built around non-string instructions. This should give you a greater appreciation for the string instructions, and may shed new light on the non-string instructions as well. I'll tell you ahead of time how the comparisons will turn out: in almost every case the string instructions will prove to be vastly superior. The lesson we learned in the last chapter holds true: use the string instructions to the hilt! There's nothing like them under the (8088) sun.

Contrasting string and non-string implementations also reinforces an important point. There are many, many ways to accomplish any given task on the 8088. It's knowing which approach to choose that separates the journeyman programmer from the guru.

STRING HANDLING WITH lods AND stos

lods is an odd bird among string instructions, being the only string instruction that doesn't benefit in the least from **rep**. While **rep** does work with **lods**, in that it causes **lods** to repeat multiple times, the combination of the two is nonetheless totally impractical: what good could it possibly do to load AL twice (to say nothing of 64 K times)? Without **rep**, **lods** is still better than **mov**, but not that much better; **lods** certainly doesn't generate the quantum jump in performance that **rep stos** and **rep movs** do. So--when does **lods** really shine?

It turns out that **lods** is what might be called a "synergistic" instruction, at its best when used with **stos** (or sometimes **scas**, or even non-string instructions) in a loop. Together, **lods** and **stos** let you load an array or string element into AL, test and/or modify it, and then write the element back to either the original array or a new array, as shown in Figure 11-1. You might think of the **lods**-process-**stos** combination as being a sort of "meta-**movs**," whereby you can whip up customized memory-to-memory moves as needed. Of course, **lods**/**stos** is slower than **movs** (especially **rep movs**), but by the same token **lods**/**stos** is far more flexible. Besides, **lods**/**stos** isn't that slow--all of the 8088's memory-accessing instructions

suffer by comparison with **movs**. Placed inside a loop, the **lods**/**stos** combination makes for fairly speedy array and string processing.

For example, Listing 11-1 copies a string to a new location, converting all characters to uppercase in the process, by using a loop containing **lods** and **stos**. Listing 11-1 takes just 773 us to copy and convert. By contrast, Listing 11-2, which uses non- string instructions to perform the same task, takes 921 us to perform the copy and conversion.

By the way, Listing 11-1 could just as easily have converted **SourceString** to uppercase in place, rather than copying the converted text to **DestString**. This would be accomplished simply by loading both DS:SI and ES:DI to point to **SourceString**, as shown in Listing 11-3, which changes nothing else from Listing 11-1.

Why is this interesting? It's interesting because two pointers--DS:SI and ES:DI--are used to point to a single array. It's often faster to maintain two pointers and use **lods** and **stos** than it is to use a single pointer with non-string instructions, as in Listing 11-4. Listing 11-3 runs in 771 us, about the same as Listing 11-1 (after all, they're virtually identical). However, Listing 11-4 takes 838 us, even though it uses only one pointer to point to the array being converted to uppercase. The **lods**/**stos** pair lies somewhere between the repeated string instructions and the non-string instructions in terms of performance and flexibility. **lods**/**stos** isn't as fast as any of the repeated string instructions, both because two instructions are involved and because it can't be used with a **rep** prefix but must instead be placed in a loop. However, **lods**/**stos** is a good deal more flexible than any

repeated string instruction, since once a memory operand is loaded into AL or AX it can be tested and manipulated easily (and often quickly as well, thanks to the accumulator-specific instructions).

On the other hand, the **lods**/**stos** pair is certainly faster than non-string instructions, as Listings 11-1 through 11-4 illustrate. However, **lods**/**stos** is not as flexible as the non- string instructions, since DS:SI and ES:DI must be used as pointer registers and only the accumulator can be loaded from and stored to memory.

On balance, the **lods**/**stos** pair overcomes some but not all of the limitations of repeated string instructions, and does so at a substantial performance cost vis-a-vis the repeated string instructions. One thing that **lods**/**stos** doesn't do particularly well is modify memory directly. For example, suppose that we want to set the high bit of every byte in a 1000-byte array. We could of course do this with **lodsb** and **stosb**, setting the high bit of each word while it's loaded into AL. Listing 11-5, which does exactly that, takes 10.07 us per word.

However, we could also use a plain old **or** instruction working directly with a memory operand to do the same thing, as shown in Listing 11-6. Listing 11-6 is just as fast as Listing 11-5 at 10.06 us per word, and it's also considerably shorter at 13 rather than 21 bytes, with 1 less byte inside the loop. **lods**/**stos** isn't disastrously worse in this case, but it certainly isn't the preferred solution--and there are plenty of other situations in which **lods**/**stos** is less than ideal.

For instance, when registers are tight, the extra pointer register

lods/**stos** takes can be sorely missed. If the accumulator is reserved for some specific purpose and can't be modified, **lods**/**stos** can't very well be used. If a pointer to far data is needed by other instructions in the same routine, the limitation of **stos** to operating in the ES segment would become a burden. In other words, while the **lods**/**stos** pair is more flexible than the repeated string instructions, its limitations are significant nonetheless.

The point is not simply that the **lods**/**stos** pair is not as flexible as the non-string instructions. The real point is that you shouldn't assume you've come up with the best solution just because you've used string instructions. Yes, I know that I've been touting string instructions as the greatest thing since sliced bread, and by and large that's true. However, because the string instructions have a sharply limited repertoire and often require a good deal of preliminary set-up, you must consider your alternatives before concluding that a string instruction-based implementation is best.

BLOCK HANDLING WITH movs

Simply put, **movs** is the king of the block copy. There's no other 8088 instruction that can hold a candle to **movs** when it comes to copying blocks of data from one area of memory to another. It does take several instructions to set up for **movs**, so if you're only moving a few bytes and DS:SI and ES:DI don't happen to be pointing to your source and destination, you might want to use a regular **mov**. Whenever you want to move more than a few bytes, though, **movs**--or better yet **rep movs**--is the ticket.

Let's look at the archetypal application for **movs**, a subroutine

which copies a block of memory from one memory area to another. What's special about the subroutine we'll look at is that it handles copying a block when the destination of the copy overlaps the source. This is a bit tricky because the direction in which the copy must proceed--from the start of the block toward the end, or vice-versa--depends on the direction of overlap.

If the destination block overlaps the source block and starts at a lower memory address than the source block, then the copy can proceed in the normal direction, from lower to higher addresses, as shown in Figure 11-2. If the destination block overlaps the source block and starts at a higher address, however, the block must be copied starting at its highest address and proceeding toward the low end, as shown in Figure 11-3. Otherwise, the first data copied to the destination block would wipe out source data that had yet to be copied, resulting in a corrupted copy, as shown in Figure 11-4. Finally, if the blocks don't overlap, the copy can proceed in either direction, since the two blocks can't conflict.

The block-copy subroutine **BlockCopyWithOverlap** shown in Listing 11-7 handles potential overlap problems exactly as described above. In cases where the destination block starts at a higher address than the source block, **BlockCopyWithOverlap** performs an **std** and uses **movs** to copy the source block starting at the high end and proceeding to the low end. Otherwise, the source block is copied from the low end to the high end with **cld**/**movs**. **BlockCopyWithOverlap** is both remarkably compact and very fast, clocking in at 5.57 ms for the cases tested in Listing 11-7. The subroutine could actually be more compact still, but I've chosen to improve

performance at the expense of a few bytes by copying as much of the block as possible a word rather than a byte at a time.

There are two points of particular interest in Listing 11-7. First, **BlockCopyWithOverlap** only handles blocks that reside in the same segment, and then only if neither block wraps around the end of the segment. While it would certainly be possible to write a version of the subroutine that properly handled both potentially overlapping copies between different segments and segment wrapping, neither of those features is usually necessary, and the additional code would reduce overall performance. If you need such a routine, write it, but as a general practice don't write extra, slower code just to handle cases that you can readily avoid.

Second, **BlockCopyWithOverlap** nicely illustrates a nasty aspect of the use of word-sized string instructions when the Direction flag is set to 1. The basic problem is this: if you point to the last byte of a block of memory and perform a word- sized operation, the byte after the end of the memory block will be accessed along with the last byte of the block, rather than the last two bytes of the block, as shown in Figure 11-5.

This problem of accessing the byte after the end of a memory block can occur with all word-sized instructions, not just string instructions. However, it's especially liable to happen with a word-sized string instruction that's moving its pointer or pointers backward (with the Direction flag equal to 1) because the temptation is to point to the end of the block, set the Direction flag, and let the string instruction do its stuff in repeated word-sized chunks for maximum performance. To avoid this problem, you must

always be sure to point to the last word rather than byte when you point to the last element in a memory block and then access memory with a word-sized instruction.

Matters get even more dicey when byte- and word-sized string instructions are mixed when the Direction flag is set to 1. This is done in Listing 11-7 in order to use **rep movsw** to move the largest possible portion of odd-length memory blocks. The problem here is that when a string instruction moves its pointer or pointers from high addresses to low, the address of the next byte that we want to access (with **lodsb**, for example) and the address of the next word that we want to access (with **lodsw**, for example) differ, as shown in Figure 11-6. For a byte-sized string instruction such as **lodsb**, we do want to point to the end of the array. After that **lodsb** has executed with the Direction flag equal to 1, though, where do the pointers point? To the address 1 byte--not 1 word--lower in memory. Then what happens when **lodsw** is executed as the next instruction, with the intent of accessing the word just above the last byte of the array? Why, the last byte of the array is incorrectly accessed again, as shown in Figure 11-7.

The solution, as shown in Listing 11-7, is fairly simple. We must perform the initial **movsb** and then adjust the pointers to point 1 byte lower in memory--to the start of the next word. Only then can we go ahead with a **movsw**, as shown in Figure 11-8.

Mind you, all this only applies when the Direction Flag is 1. When the Direction flag is 0, **movsb** and **movsw** can be mixed freely, since the address of the next byte is the same as the address of the next word

when we're counting from low addresses to high, as shown in Figure 11-9. Listing 11-7 reflects this, since the pointer adjustments are only made when the Direction flag is 1.

Listing 11-8 contains a version of **BlockCopyWithOverlap** that does exactly what the version in Listing 11-7 does, but does so without string instructions. While Listing 11-8 doesn't look all that much different from Listing 11-7, it takes a full 15.16 ms to run--quite change from the time of 5.57 ms we measured for Listing 11-7. Think about it: Listing 11-7 is nearly three times as fast as Listing 11-8, thanks to **movs**--and it's shorter too.

Enough said.

SEARCHING WITH scas

scas is often (but not always, as we shall see) the preferred way to search for either a given value or the absence of a given value in any array. When **scas** is well-matched to the task at hand, it is the best choice by a wide margin. For example, suppose that we want to count the number of times the letter 'A' appears in a text array. Listing 11-9, which uses non-string instructions, counts the number of occurrences of 'A' in the sample array in 475 us. Listing 11-10, which does exactly the same thing with **repnz scasb**, finishes in just 203 us. That, my friends, is an improvement of 134%. What's more, Listing 11- 10 is shorter than Listing 11-9.

Incidentally, Listing 11-10 illustrates the subtlety of the pitfalls associated with forgetting that **scas** repeated zero times (with CX equal to zero) doesn't alter the flags. If the **jcxz** instruction in Listing 11-10 were to

be removed, the code would still work perfectly--except when the array being scanned was exactly 64 K bytes long and every byte in the array matched the byte being searched for. In that one case, CX would be zero when **repnz scasb** was restarted after the last match, causing **repnz scasb** to drop through without altering the flags. The Zero flag would be 0 as a result of DX previously incrementing from 0FFFFh to 0, and so the **jnz** branch would not be taken. Instead, DX would be incremented again, causing a non-existent match to be counted. The result would be that 1 rather than 64 K matches would be returned as the match count, an error of considerable magnitude.

If you could be sure that no array longer than 64 K-1 bytes would ever be passed to **ByteCount**, you could eliminate the **jcxz** and speed the code considerably. Trimming the fat from your code until it's matched exactly to an application's needs is one key to performance.

scas AND ZERO-TERMINATED STRINGS

Clearly, then, when you want to find a given byte or word value in a buffer, table, or array of a known fixed length, it's often best to load up the registers and let a repeated **scas** do its stuff. However, the same is not always true of searching tasks that require multiple comparisons for each byte or word, such as a loop that ends when either the letter 'A' or a zero byte is found. Alas, **scas** can perform just one comparison per memory location, and **repz** or **repnz** can only terminate on the basis of the Zero flag setting after that one comparison. This is unfortunate because multiple

comparisons are exactly what we need to handle C-style strings, which are of no fixed length and are terminated with zeros. **rep scas** can still be used in such situations, but its sheer power is diluted by the workarounds needed to allow it to function more flexibly than it is normally capable of doing. The choice between repeated **scas** instructions and other approaches then must be made on a case-by-by case basis, according to the balance between the extra overhead needed to coax **scas** into doing what is needed and the inherent speed of the instruction.

For example, suppose we need a subroutine that returns either the offset in a string of the first instance of a selected byte value or the value zero if a zero byte (marking the end of the string) is encountered before the desired byte is found. There's no simple way to do this with **scasb**, for in this application we have to compare each memory location first to the desired byte value and then to zero. **scasb** can perform one comparison or the other, but not both.

Now, we could use **rep scasb** to find the zero byte at the end of the string, so we'd know how long the string was, and then use **rep scasb** again with CX set to the length of the string to search for the selected byte value. Unfortunately, that involves processing every byte in the string once before even beginning the search. On average, this double-search approach would read every element of the string being searched once and would then read one-half of the elements again, as shown in Figure 11-10. By contrast, an approach that reads each byte and immediately compares it to both the desired value and zero would read only one-half of the elements in the

string, as shown in Figure 11-11. Powerful as repeated **scasb** is, could it possibly run fast enough to allow the double-search approach to outperform an approach that accesses memory only one-third as many times?

The answer is yes...conditionally. The double-search approach actually is slightly faster than a **lodsb**-based single- search string-searching approach for the average case. The double-search approach performs relatively more poorly if matches tend to occur most frequently in the first half of the strings being searched, and relatively better if matches tend to occur in the second half of the strings. Also, the more flexible **lodsb**- based approach rapidly becomes the solution of choice as the termination condition becomes more complex, as when a case- insensitive search is desired. The same is true when modification as well as searching of the string is desired, as when the string is converted to uppercase.

Listing 11-11 shows **lodsb**-based code that searches a zero-terminated string for the character 'z'. For the sample string, which has the first match right in the middle of the string, Listing 11-11 takes 375 us to find the match. Listing 11-12 shows **repnz scasb**-based code that uses the double-search approach. For the same sample string as Listing 11-11, Listing 11-12 takes just 340 us to find the match, despite having to perform about three times as many memory accesses as Listing 11- 11--a tribute to the raw power of repeated **scas**. Finally, Listing 11-13, which performs the same search using non-string instructions, takes 419 us to find the match.

It is apparent from Listings 11-11 and 11-12 that the performance margin between **scas**-based string searching and other approaches is

considerably narrower than it was for array searching, due to the more complex termination conditions. Given a still more complex termination condition, **lods** would likely become the preferred solution due to its greater flexibility. In fact, if we're willing to expend a few bytes, the greater flexibility of **lods** can be translated into higher performance for Listing 11-11, as follows.

Listing 11-14 shows an interesting variation on Listing 11- 11. Here **lodsw** rather than **lodsb** is used, and AL and AH, respectively, are checked for the termination conditions. This technique uses a bit more code, but the replacement of two **lodsb** instructions with a single **lodsw** and the elimination of every other branch pays off handsomely, as Listing 11-14 runs in just 325 us, 15% faster than Listing 11-11 and 5% faster than Listing 11-12. The key here is that **lods** allows us leeway in designing code to work around the slow memory access and slow branching of the 8088, while **scas** does not. In truth, the flexibility of **lods** can make for better performance still through in-line code...but that's a story for the next few chapters.

MORE ON scas AND ZERO-TERMINATED STRINGS

While repeated **scas** instructions aren't ideally suited to string searches involving complex conditions, they do work nicely with strings whenever brute force scanning comes into play. One such application is finding the offset of the last element of some sort in a string. For example, Listing 11-15, which finds the last non-blank element of a string by using **lodsw** and remembering the offset of the most recent non-blank character

encountered, takes 907 us to find the last non-blank character of the sample string, which has the last non-blank character in the middle of the string. Listing 11-16, which does the same thing by using **repnz scasb** to find the end of the string and then **repz scasw** with the Direction flag set to 1 to find the first non- blank character scanning backward from the end of the string, runs in just 386 us.

That's an amazing improvement given our earlier results involving the relative speeds of **lodsw** and repeated **scas** in string applications. The reason that repeated **scas** outperforms **lodsw** by a tremendous amount in this case but underperformed it earlier is simple. The **lodsw**-based code always has to check every character in the string-- right up to the terminating zero-- when searching for the last non-blank character, as shown in Figure 11-12. While the **scasb**-base code also has to access every character in the string, and then some, as shown in Figure 11-13, the worst case is that Listing 11-16 accesses string elements no more than twice as many times as Listing 11-15. In our earlier example, the best case was a two-to-one ratio. The timing results for Listings 11-15 and 11-16 show that the superior speed, lack of prefetching, and lack of branching associated with repeated **scas** far outweigh any performance loss resulting from a memory-access ratio of less than two-to-one.

By the way, Listing 11-16 is an excellent example of the need to correct for pointer overrun when using the string instructions. No matter which direction we scan in, it's necessary to undo the last advance of DI performed by **scas** in order to point to the byte on which the comparison

ended.

Listing 11-16 also shows the use of **jcxz** to guard against the case where CX is zero. As you'll recall from the last chapter, repeated **scas** doesn't alter the flags when started with CX equal to zero. Consequently, we must test for the case of CX equal to zero before performing **repz scasw**, and we must treat that case if we had never found the terminating condition (a non- blank character). Otherwise, the leftover flags from an earlier instruction might give us a false result following a **repz scasw** which doesn't change the flags because it is repeated zero times. In Listing 11-21 we'll see that we need to do the same with repeated **cmps** as well.

Bear in mind, however, that there are several ways to solve any problem in assembler. For example, in Listing 11-16 I've chosen to use **jcxz** to guard against the case where CX is zero, thereby compensating for the fact that **scas** repeated zero times doesn't change the flags. Rather than thinking defensively, however, we could actually take advantage of that particular property of repeated **scas**. How? We could set the Zero flag to 1 (the "match" state) by placing **sub dx,dx** before **repz scasw**. Then if **repz scasw** is repeated zero times because CX is zero the following conditional jump will reach the proper conclusion, that the desired non-match (a non-blank character) wasn't found.

As it happens, **sub dx,dx** isn't particularly faster than **jcxz**, and so there's not much to choose from between the two solutions. With **sub dx,dx** the code is 3 cycles faster when CX isn't zero but is the same number of bytes in length, and is considerably slower when CX is zero. (There's

really no reason to worry about performance here when CX is zero, however, since that's a rare case that's always handled relatively quickly. Rather, our focus should be on losing as little performance as possible to the test for CX being zero in the more common case-- when CX isn't zero.) In another application, though, the desired Zero flag setting might fall out of the code preceding the repeated **cmps**, and no extra code at all would be required for the test for CX equal to zero. Listing 11-24, which we'll come to shortly, is such a case.

What's interesting here is that it's instinctive to use **jcxz**, which is after all a specialized and fast instruction that is clearly present in the 8088's instruction set for just such a purpose as protecting against repeating a string comparison zero times. The idea of presetting a flag and letting the comparison drop through without changing the flag, on the other hand, is anything but intuitive--but is just about as effective as **jcxz**, more so under certain circumstances.

Don't let your mind be constrained by intentions of the designers of the 8088. Think in terms of what instructions do rather than what they were intended to do.

USING REPEATED scasw ON BYTE-SIZED DATA

Listing 11-16 is also a fine example of how to use repeated **scasw** on byte-sized data. You'll recall that one of the rules of repeated string instruction usage is that word-sized string instructions should be used wherever possible, due to their faster overall speed. It turns out, however,

that it's rather tricky to apply this rule to **scas**.

For starters, there's hardly ever any use for **repnz scasw** when searching for a specific byte value in memory. Why? Well, while we could load up both AH and AL with the byte we're looking for and then use **repnz scasw**, we'd only find cases where the desired byte occurs at least twice in a row, and then we'd only find such 2-byte cases that didn't span word boundaries. Unfortunately, there's no way to use **repnz scasw** to check whether either AH or AL--but not necessarily both--matched their respective bytes. With **repnz scasw**, if AX doesn't match all 16 bits of memory, the search will continue, and individual byte matches will be missed.

On the other hand, we can use **repz scasw** to search for the first non-match, as in Listing 11-16. Why is it all right to search a word at a time for non-matches but not matches? Because if either byte of each word compared with **repz scasw** doesn't match the byte of interest (which is stored in both AH and AL), then **repz scasw** will stop, which is what we want. Of course, there's a bit of cleaning up to do in order to figure out which of the 2 bytes was the first non-match, as illustrated by Listing 11-16. Yes, it is a bit complex and does add a few bytes, but it also speeds things up, and that's what we're after.

In short, **repz scasw** can be used to boost performance when scanning for non-matching byte-sized data. However, **repnz scasw** is generally useless when scanning for matching byte-sized data.

scas AND LOOK-UP TABLES

One common application for table searching is to get an element number or an offset into a table that can be used to look up related data or a jump address in another table. We saw look- up tables in Chapter 7, and we'll see them again, for they're a potent performance tool.

scas is often excellent for look-up code, but the pointer and counter overrun characteristic of all string instructions make it a bit of a nuisance to calculate offsets and/or element numbers after repeated **scas** instructions. Listing 11-17 shows a subroutine that calculates the offset of a match in a word-sized table in the process of jumping to the associated routine from a jump table. Notice that it's necessary to subtract the 2-byte overrun from the difference between the final value of DI and the start of the table. The calculation would be the same for a byte-sized table scanned with **scasb**, save that **scasb** has only a 1-byte overrun and so only 1 would be subtracted from the difference between DI and the start of the table.

Finding the element number is a slightly different matter. After a repeated **scas**, CX contains the number of elements that weren't scanned. Since CX counts down just once each time **scas** is repeated, there's no difference between **scasw** and **scasb** in this respect.

Well, if CX contains the number of elements that weren't scanned, then subtracting CX from the table length in elements must yield the number of elements that were scanned. Subtracting 1 from that value gives us the number of the last element scanned. (The first element is element number 0, the second element is element number 1, and so on.) Listing 11-18 illustrates the calculation of the element number found in a

look-up table as a step in the process of jumping to the associated routine from a jump table, much as in Listing 11-17.

CONSIDER YOUR OPTIONS

Don't assume that **scas** is the ideal choice even for all memory-searching tasks in which the search length is known. Suppose that we simply want to know if a given character is any of, say, four characters: 'A', 'Z', '3', or '!'. We could do this with **repnz scasb**, as shown in Listing 11-19. Alternatively, however, we could simply do it with four comparisons and conditional jumps, as shown in Listing 11-20. Even with the prefetch queue cycle-eater doing its worst, each compare and conditional jump pair takes no more than 16 cycles when the jump isn't taken (the jump is taken at most once, on a match), which stacks up pretty well against the 15 cycle per comparison and 9 cycle set-up time of **repnz scasb**. What's more, the compare-and- jump approach requires no set-up instructions. In other words, the less sophisticated approach might well be better in this case.

The Zen timer bears this out. Listing 11-19, which uses **repnz scasb**, takes 183 us to perform five checks, while Listing 11-20, which uses the compare-and-jump approach, takes just 119 us to perform the same five checks. Listing 11-20 is not only 54% faster than Listing 11-19 but is also 1 byte shorter. (Don't forget to count the look-up table bytes in Listing 11-19.)

Of course, the compare-and-jump approach is less flexible than the look-up approach, since the table length and contents can't be passed as parameters or changed as the program runs. The compare-and-jump

approach also becomes unwieldy when more entries need to be checked, since 4 bytes are needed for each additional compare-and-jump entry where the **repnz scasb** approach needs just 1. The compare-and-jump approach finally falls apart when it's no longer possible to short-jump out of the comparison/jump code and so jumps around jumps must be used, as in:

```
cmp     al,'Z'
jnz     $+5
jmp     CharacterFound
cmp     al,'3'
```

When jumps around jumps are used, the comparison time per character goes from 16 to 24 cycles, and **rep scasb** emerges as the clear favorite.

Nonetheless, Listings 11-19 and 11-20 illustrate two important points. Point number 1: the repeated string instructions tend to have a greater advantage when they're repeated many times, allowing their speed and compact size to offset the overhead in set-up time and code they require. Point number 2: specialized as the string instructions are, there are ways to program the 8088 that are more specialized still. In certain cases, those specialized approaches can even outperform the string instructions. Sure, the specialized approaches, such as the compare-and-jump approach we just saw, are limited and inflexible--but when you don't need the flexibility, why pay for it in lost performance?

COMPARING MEMORY TO MEMORY WITH cmps

When **cmps** does exactly what you need done it can't be beat,

although to an even greater extent than with **scas** the cases in which that is true are relatively few. **cmps** is used for applications in which byte-for-byte or word-for-word comparisons between two memory blocks of a known length are performed, most notably array comparisons and substring searching. Like **scas**, **cmps** is not flexible enough to work at full power on other comparison tasks, such as case-insensitive substring searching or the comparison of zero-terminated strings, although with a bit of thought **cmps** can be made to serve adequately in some such applications.

cmps does just one thing, but it does far better than any other 8088 instruction or combination of instructions. The one transcendent ability of **cmps** is the direct comparison of two fixed-length blocks of memory. The obvious use of **cmps** is in determining whether two memory arrays or blocks of memory are the same, and if not, where they differ. Listing 11-21, which runs in 685 us, illustrates **repz cmpsw** in action. Listing 11-22, which performs exactly the same task as Listing 11-21 but uses **lodsw** and **scasw** instead of **cmpsw**, runs in 1298 us. Finally, Listing 11-23, which uses non-string instructions, takes a leisurely 1798 us to complete the task. As you can see, **cmps** blows away not only non-string instructions but also other string instructions under the right circumstances. (As I've said before, there are many, many different sequences of assembler code that will work for any given task. It's the choice of implementation that makes the difference between adequate code and great code.)

By the way, in Listings 11-21 though 11-23 I've used **jcxz** to make sure the correct result is returned if zero-length arrays are compared. If you

use this routine in your code and you can be sure that zero-length arrays will never be passed as parameters, however, you can save a few bytes and cycles by eliminating the **jcxz** check. After all, what sense does it make to compare zero-length arrays...and what sense does it make to waste precious bytes and cycles guarding against a contingency that can never arise?

Make the comparison a bit more complex, however, and **cmps** comes back to the pack. Consider the comparison of two zero- terminated strings, rather than two fixed-length arrays. As with **scas** in the last section, **cmps** can be made to work in this application by first performing a **scasb** pass to determine one string length and then comparing the strings with **cmpsw**, but the double pass negates much of the superior performance of **cmps**. Listing 11-24 shows an implementation of this approach, which runs in 364 us for the test strings.

We found earlier that **lods** works well for string searching when multiple termination conditions must be dealt with. That is true of string comparison as well, particularly since there we can benefit from the combination of **scas** and **lods**. The **lodsw**/**scasw** approach, shown in Listing 11-25, runs in just 306 us--19% faster than the **rep scasb**/**repz cmpsw**-based Listing 11-24. For once, I won't bother with a non-string instruction-based implementation, since it's perfectly obvious that replacing **lodsw** and **scasw** with non-string sequences such as:

```
mov     ax,[si]
inc     si
inc     si
```

and:

```
        cmp             [di],ax
                        :
        inc             di
        inc             di
```

can only reduce performance.

cmps and even **scas** become still less suitable if a highly complex operation such as case-insensitive string comparison is required. Since both source and destination must be converted to the same case before being compared, both must be loaded into the registers for manipulation, and only **lods** among the string instructions will do us any good at all. Listing 11-26 shows code that performs case-insensitive string comparison. Listing 11-26 takes 869 us to run, which is not very fast by comparison with Listings 11-21 through 11-25. That's to be expected, though, given the flexibility required for this comparison. The more flexibility required for a given task, the less likely we are to be able to bring the full power of the highly-specialized string instructions to bear on that task. That doesn't mean that we shouldn't try to do so, just that we won't always succeed.

If we're willing to expend 200 extra bytes or so, we can speed Listing 11-26 up considerably with a clever trick. Making sure a character is uppercase takes a considerable amount of time even when all calculations are done in the registers, as is the case in Listing 11-26. Fast as the instructions in the macro **TO_UPPER** in Listing 11-26 are, two to five of them

are executed every time a byte is made uppercase, and a time-consuming conditional jump may also be performed.

So what's better than two to five register-only instructions with at most one jump? A look-up table, that's what. Listing 11-27 is a modification of Listing 11-26 that looks up the uppercase version of each character in **ToUpperTable** with a single instruction--and the extremely fast and compact **xlat** instruction, at that. (It's possible that **mov** could be used instead of **xlat** to make an even faster version of Listing 11-27, since **mov** can reference any general-purpose register while **xlat** can only load AL. As I've said, there are many ways to do anything in assembler.) For most characters there is no uppercase version, and the same character that we started with is looked up in **ToUpperTable**. For the 26 lowercase characters, however, the character looked up is the uppercase equivalent.

You may well be thinking that it doesn't make much sense to try to speed up code by adding a memory access, and normally you'd be right. However, **xlat** is very fast--it's a 1-byte instruction that executes in 10 cycles--and it saves us the trouble of fetching the many instruction bytes of **TO_UPPER**. (Remember, instruction fetches are memory accesses too.) What's more, **xlat** eliminates the need for conditional jumps in the uppercase-conversion process.

Sounds good in theory, doesn't it? It works just as well in the real world, too. Listing 11-27 runs in just 638 us, a 36% improvement over Listing 11-26. Of course, Listing 11-27 is also a good deal larger than Listing 11-26, owing to the look-up table, and that's a dilemma the assembler

programmer faces frequently on the PC: the choice between speed and size. More memory, in the form of look-up tables and in-line code, often means better performance. It's actually relatively easy to speed up most code by throwing memory at it. The hard part is knowing where to strike the balance between performance and size.

Although both look-up tables and in-line code are discussed elsewhere in this volume, a broad discussion of the issue of memory versus performance will have to wait until Volume II of The Zen of Assembly Language. The mechanics of translating memory into performance--the knowledge aspect, if you will--is quite simple, but understanding when that tradeoff can and should be made is more complex and properly belongs in the discussion of the flexible mind.

STRING SEARCHING

Perhaps the single finest application of **cmps** is in searching for a sequence of bytes within a data buffer. In particular, **cmps** is excellent for finding a particular text sequence in a buffer full of text, as is the case when implementing a find-string capability in a text editor.

One way to implement such a searching capability is by simply starting **repz cmps** at each byte of the buffer until either a match is found or the end of the buffer is reached, as shown in Figure 11-14. Listing 11-28, which employs this approach, runs in 2995 us for the sample search sequence and buffer. That's not bad, but there's a better way to go. Suppose we load the first byte of the search string into AL and use **repnz**

scasb to find the next candidate for the full **repz cmps** comparison, as shown in Figure 11-15. By so doing we could use a fast repeated string instruction to disqualify most of the potential strings, rather than having to loop and start up **repz cmps** at each and every byte in the buffer. Would that make a difference?

It would indeed! Listing 11-29, which uses the hybrid **repnz scasb**/**repz cmps** technique, runs in just 719 us for the same search sequence and buffer as Listing 11-28. Now, the margin between the two techniques could vary considerably, depending on the contents of the buffer and the search sequence. Nonetheless, we've just seen an improvement of more than 300% over already- fast string instruction-based code! That improvement is primarily due to the use of **repnz scasb** to eliminate most of the instruction fetches and branches of Listing 11-28.

Even when you're using string instructions, stretch your mind to think of still-better approaches...

As for non-string implementations, Listing 11-30, which performs the same task as do Listings 11-28 and 11-29 but does so with non-string instructions, takes a full 3812 us to run. It should be very clear that non-string instructions should be used in searching applications only when their greater flexibility is absolutely required.

Make no mistake, there's more to searching performance than simply using the right combination of string instructions. The right choice of algorithm is critical. For a list of several thousand sorted items, a poorly-coded binary search might well beat the pants off a slick **repnz scasb**/**repz**

cmps implementation. On the other hand, the **repnz scasb**/**repz cmps** approach is excellent for searching free-form data of the sort that's found in text buffers.

The key to searching performance lies in choosing a good algorithm for your application and implementing it with the best possible code. Either the searching algorithm or the implementation may be the factor that limits performance. Ideally, a searching algorithm would be chosen with an eye toward using the strengths of the 8088--and that usually means the string instructions.

cmps WITHOUT rep

In the last chapter I pointed out that **scas** and **cmps** are slower but more flexible when they're not repeated. Although **repz** and **repnz** only allow termination according to the state of the Zero flag, **scas** and **cmps** actually set all the status flags, and we can take advantage of that when **scas** and **cmps** aren't repeated. Of course, we should use **repz** or **repnz** whenever we can, but non-repeated **scas** and **cmps** let us tap the power of string instructions when **repz** and **repnz** simply won't do.

For instance, suppose that we're comparing two arrays that contain signed 16-bit values representing signal measurements. Suppose further that we want to find the first point at which the waves represented by the arrays cross. That is, if wave A starts out above wave B, we want to know when wave A becomes less than or equal to wave B, as shown in Figure 11-16. If wave B starts out above wave A, then we want to know when wave

B becomes less than or equal to wave A.

There's no way to perform this comparison with repeated **cmps**, since greater-than/less-than comparisons aren't in the limited repertoire of the **rep** prefix. However, plain old non- repeated **cmpsw** is up to the task, as shown in Listing 11-31, which runs in 1232 us. As shown in Listing 11-31, we must initially determine which array starts out on top, in order to set SI to point to the initially-greater array and DI to point to the other array. Once that's done, all we need do is perform a **cmpsw** on each data point and check whether that point is still greater with **jg**. **loop** repeats the comparison for however many data points there are--and that's the whole routine in a very compact package! The 3-instruction, 5-byte loop of Listing 11-31 is hard to beat for this fairly demanding task.

By contrast, Listing 11-32, which performs the same crossing search but does so with non-string instructions, has 6 instructions and 13 bytes in the loop and takes considerably longer--1821 us--to complete the sample crossing search. Although we were unable to use repeated **cmps** for this particular task, we were nonetheless able to improve performance a great deal by using the string instruction in its non-repeated form.

A NOTE ABOUT RETURNING VALUES

Throughout this chapter I've been returning "not found" statuses by passing zero pointers (pointers set to zero) back to the calling routine. This is a commonly used and very flexible means of returning such statuses, since the same registers that are used to return pointers when searches are successful can be used to return zero when searches are not successful.

The success or failure of a subroutine can then be tested with code like:

```
        call    FindCharInString
        and     si,si
        jz      CharNotFound
```

Returning failure statuses as zero pointers is particularly popular in high-level languages such as C, although C returns pointers in either AX, DX:AX, or memory, rather than in SI or DI.

However, there are many other ways of returning statuses in assembler. One particularly effective approach is that of returning success or failure in either the Zero or Carry flag, so that the calling routine can immediately jump conditionally upon return from the subroutine, without the need for any anding, oring, or comparing of any sort. This works out especially well when the proper setting of a flag falls out of the normal functioning of a subroutine. For example, consider the following subroutine, which returns the Zero flag set to 1 if the character in AL is whitespace:

```
Whitespace:
        cmp     al,' '      ;space
        jz      WhitespaceDone
        cmp     al,9                ;tab
        jz      WhitespaceDone
        and     al,al       ;zero byte
WhitespaceDone:
        ret
```

The key point here is that the Zero flag is automatically set by the comparisons preceding the **ret**. Any test for whitespace would have to

perform the same comparisons, so practically speaking we didn't have to write a single extra line of code to return the subroutine's status in the Zero flag. Because the return status is in a flag rather than a register, **Whitespace** could be called and the outcome handled with a very short sequence of instructions, as follows:

```
mov     al,[Char]
call    Whitespace
jnz     NotWhitespace
```

The particular example isn't important here. What is important is that you realize that in assembler (unlike high- level languages) there are many ways to return statuses, and that it's possible to save a great deal of code and/or time by taking advantage of that. Now is not the time to pursue the topic further, but we'll return to the issues of passing values and statuses both to and from assembler subroutines in Volume II of The Zen of Assembly Language.

PUTTING STRING INSTRUCTIONS TO WORK IN UNLIKELY PLACES

I've said several times that string instructions are so powerful that you should try to use them even when they don't seem especially well-matched to a particular application. Now I'm going to back that up with an unlikely application in which the string instructions have served me well over the years: animation.

This section is actually a glimpse into the future. Volume II of

The Zen of Assembly Language will take up the topic of animation in much greater detail, since animation truly falls in the category of the flexible mind rather than knowledge. Still, animation is such a wonderful example of what the string instructions can do that we'll spend a bit of time on it here and now. It'll be a whirlwind look, with few details and nothing more than a quick glance at theory, for the focus isn't on animation per se. What's important is not that you understand how animation works, but rather that you get a feel for the miracles string instructions can perform in places where you wouldn't think they could serve at all.

ANIMATION BASICS

Animation involves erasing and redrawing one or more images quickly enough to fool the eye into perceiving motion, as shown in Figure 11-17. Animation is a marginal application for the PC, by which I mean that the 8088 barely has enough horsepower to support decent animation under the best of circumstances. What that means is that the Zen of assembler is an absolute must for PC animation.

Traditionally, microcomputer animation has been performed by exclusive-oring images into display memory; that is, by drawing images by inserting the bits that control their pixels into display memory with the **xor** instruction. When an image is first exclusive-ored into display memory at a given location, the image becomes visible. A second exclusive-oring of the image at the same location then erases the image. Why? That's simply the nature of the exclusive-or operation.

Consider this. When you exclusive-or a 1 bit with another bit once, the other bit is flipped. When you exclusive-or the same 1 bit with that other bit again, the other bit is again flipped--right back to its original state, as shown in Figure 11- 18. After all, a bit only has two possible states, so a double flip must restore the bit back to the state in which it started. Since exclusive-oring a 0 bit with another bit never affects the other bit, exclusive-oring a target bit twice with either a 1 or a 0 bit always leaves the target bit in its original state.

Why is exclusive-oring so popular for animation? Simply because no matter how many images overlap, the second exclusive- or of an image always erases it without interfering with any other images. In other words, the perfect reversibility of the exclusive-or operation means that you could exclusive-or each of 10 images once at the same location, drawing the images right on top of each other, then exclusive-or them all again at the same place--and they would all be erased. With exclusive-oring, the drawing or erasing of one image never interferes with the drawing or erasing of other images it overlaps.

If you're catching all this, great. If not, don't worry. I'm not going to spend time explaining animation now--better we should wait until Volume II, when we have the time to do it right. The important point is that exclusive-oring is a popular animation technique, primarily because it eliminates the complications of drawing and erasing overlapping images.

Listing 11-33, which bounces 10 images around the screen, illustrates animation based on exclusive-oring. When run on an Enhanced

Graphics Adapter (EGA), Listing 11-33 takes 30.29 seconds to move and redraw every image 500 times. (Note that the long-period Zen timer was used to time Listing 11-33, since we can't perform much animation within the 54 ms maximum period of the precision Zen timer.)

Listing 11-33 isn't a general-purpose animation program. I've kept complications to a minimum in order to show basic exclusive-or animation. Listing 11-33 allows us to observe the fundamental strengths and weaknesses (primarily the latter) of the exclusive-or approach.

When you run Listing 11-33, you'll see why exclusive-oring is less than ideal. While overlapping images don't interfere with each other so far as drawing and erasing go, they do produce some unattractive on-screen effects. In particular, unintended colors and patterns often result when multiple images are exclusive-ored into the same bytes of display memory. Another problem is that exclusive-ored images flicker because they're constantly being erased and redrawn. (Each image could instead be redrawn at its new location before being erased at the old location, but the overlap effects characteristic of exclusive- oring would still cause flicker.) That's not all, though. There's a still more serious problem with exclusive-or based animation...

Exclusive-oring is slow.

The problem isn't that the **xor** instruction itself is particular slow; rather, it's that the **xor** instruction isn't a string instruction. **xor** can't be repeated with **rep**, it doesn't advance its pointers automatically, and it just isn't as speedy as, say, **movs**. Still, neither **movs** nor any other string

instruction can perform exclusive-or operations, so it would seem we're stuck.

We're hardly stuck, though. On the contrary, we're bound for glory!

STRING INSTRUCTION-BASED ANIMATION

If string instructions can't perform exclusive-oring, then we'll just have to figure out a way to animate without exclusive- oring. As it turns out, there's a very nice way to do this. I learned this approach from Dan Illowsky, who developed it before string instructions even existed, way back in the early days of the Apple II.

First, we'll give each image a small blank fringe. Then we'll make it a rule never to move an image by more than the width of its fringe before redrawing it. Finally we'll draw images by simply copying them to display memory, destroying whatever they overwrite, as shown in Figure 11-19. Now, what does that do for us?

Amazing things. For starters, each image will, as it is redrawn, automatically erase its former incarnation. That means that there's no flicker, since images are never really erased, but only drawn over themselves. There are also no color effects when images overlap, since only the image that was drawn most recently at any given pixel is visible.

In short, this sort of animation (which I'll call "block- move animation") actually looks considerably better than animation based on exclusive-oring. That's just frosting on the cake, though--the big payoff is

speed. With block-move animation we suddenly don't need to exclusive-or anymore--in fact, **rep movs** will work beautifully to draw a whole line of an image in a single instruction. We also don't need to draw each image twice per move--once to erase the image at its old location and once to draw it at its new location--as we did with exclusive-oring, since the act of drawing the image at a new location serves to erase the old image as well. But wait, there's more! **xor** accesses a given byte of memory twice per draw, once to read the original byte and once to write the modified byte back to memory. With block-move animation, on the other hand, we simply write each byte of an image to memory once and we're done with that byte. In other words, between the elimination of a separate erasing step and the replacement of read-**xor**-write with a single write, block-move animation accesses display memory only about one-third as many times as exclusive-or animation. (The ratio isn't quite 1 to 4 because the blank fringe makes block-move animation images somewhat larger.)

Are alarm bells going off in your head? They should be. Think back to our journey beneath the programming interface. Think of the cycle-eaters. Ah, you've got it! Exclusive-or animation loses about three times as much performance to the display adapter cycle-eater as does block-move animation. What's more, block-move animation uses the blindingly fast **movs** instruction. To top it off, block-move animation loses almost nothing to the prefetch queue cycle-eater or the 8088's slow branching speed, thanks to the **rep** prefix.

Sounds almost too good to be true, doesn't it? It is true, though:

block-move animation relies almost exclusively on one of the two most powerful instructions of the 8088 (**cmps** being the other), and avoids the gaping maws of the prefetch queue and display adapter cycle-eaters in the process. Which leaves only one question:

How fast is block-move animation? Remember, theory is fine, but we don't trust any code until we've timed it. Listing 11-34 performs the same animation as Listing 11-34, but with block-move rather than exclusive-or animation. Happily, Listing 11-34 lives up to its advance billing, finishing in just 10.35 seconds when run on an EGA. Block-move animation is close to three times as fast as exclusive-oring in this application--and it looks better, too. (You can slow down the animation in order to observe the differences between the two sorts of animation more closely by setting **DELAY** to a higher value in each listing.)

Let's not underplay the appearance issue just because the performance advantage of block-move animation is so great. If you possibly can, enter and run Listings 11-33 and 11-34. The visual impact of block-move animation's flicker-free, high-speed animation is startling. It's hard to imagine that any programmer would go back to exclusive-oring after seeing block-move animation in action.

That's not to say that block-move animation is perfect. Unlike exclusive-oring, block-move animation wipes out the background unless the background is explicitly redrawn after each image is moved. Block-move animation does produce flicker and fringe effects when images overlap. Block-move animation also limits the maximum distance by which an image

can move before it's redrawn to the width of its fringe.

If block-move animation isn't perfect, however, it's much better than exclusive-oring. What's really noteworthy, however, is that we looked at an application--animation--without preconceived ideas about the best implementation, and came up with an approach that merged the application's needs with one of the strengths of the PC--the string instructions--while avoiding the cycle-eaters. In the end, we not only improved performance remarkably but also got better animation, in the process turning a seeming minus--the limitations of the string instructions--into a big plus. All in all, what we've just done is the Zen of assembler working on all levels: knowledge, flexible mind, and implementation.

Try to use the string instructions for all your time- critical code, even when you think they just don't fit. Sometimes they don't--but you can never be sure unless you try...and if they can be made to fit, it will pay off big.

NOTES ON THE ANIMATION IMPLEMENTATIONS

Spend as much time as you wish perusing Listings 11-33 and 11-34, but do not worry if they don't make complete sense to you right now. The point of this exercise was to illustrate the use of the string instructions in an unusual application, not to get you started with animation. In Volume II of The Zen of Assembly Language we'll return to animation in a big way.

The animation listings are not full-featured, flexible implementations, nor were they meant to be. My intent in creating these

programs was to contrast the basic operation and raw performance of exclusive-or and block-move animation. Consequently, I've structured the two listings along much the same lines, and while the code is fast, I've avoided further optimizations (notably the use of in-line code) that would have complicated matters. We'll see those additional optimizations in Volume II.

One interesting point to be made about the animation listings is that I've assumed in the drawing routines that images always start on even rows of the screen and are always an even number of rows in height. Many people would consider the routines to be incomplete, since they lack the extra code needed to handle the complications of odd start rows and odd heights in 320x200 4-color graphics mode. Of course, that extra code would slow performance and increase program size, but would be deemed necessary in any "full" animation implementation.

Is the handling of odd start rows and odd heights really necessary, though? Not if you can structure your application so that images can always start on even rows and can always be of even heights, and that's actually easy to do. No one will ever notice whether images move 1 or 2 pixels at a time; the nature of animation is such that the motion of an image appears just as smooth in either case. And why should there be a need for odd image heights? If necessary, images of odd height could be padded out with an extra line. In fact, an extra line can often be used to improve the appearance of an image.

In short, "full" animation implementations will not only run slower

than the implementation in Listings 11-33 and 11-34 but may not even yield any noticeable benefits. The lesson is this: only add features that slow your code when you're sure you need them. High-performance assembler programming is partly an art of eliminating everything but the essentials.

By the way, Listings 11-33 and 11-34 move images a full 4 pixels at a time horizontally, and that's a bit too far. 2 pixels is a far more visually attractive distance by which to move animated images, especially those that move slowly. However, because each byte of 320x200 4-color mode display memory controls 4 pixels, alignment of images to start in columns that aren't multiples of 4 is more difficult, although not really that hard once you get the hang of it. Since our goal in this section was to contrast block-move and exclusive-or animation, I didn't add the extra code and complications required to bit-align the images. We will discuss bit-alignment of images at length in Volume II, however.

A NOTE ON HANDLING BLOCKS LARGER THAN 64 K BYTES

All the string instruction-based code we've seen in this chapter handles only blocks or strings that are 64 K bytes in length or shorter. There's a very good reason for this, of course--the infernal segmented architecture of the 8088--but there are nonetheless times when larger memory blocks are needed.

I'm going to save the topic of handling blocks larger than 64 K bytes for Volume II of The Zen of Assembly Language. Why? Well, the trick with code that handles larger memory blocks isn't getting it to work; that's

relatively easy if you're willing to perform 32-bit arithmetic and reload the segment registers before each memory access. No, the trick is getting code that handles large memory blocks to work reasonably fast.

We've seen that a key to assembler programming lies in converting difficult problems from approaches ill-suited to the 8088 to ones that the 8088 can handle well, and this is no exception. In this particular application, we need to convert the task at hand from one of independently addressing every byte in the 8088's 1-megabyte address space to one of handling a series of blocks that are each no larger than 64 K bytes, so that we can process up to 64 K bytes at a time very rapidly without touching the segment registers.

The concept is simple, but the implementation is not so simple and requires the flexible mind...and that's why the handling of memory blocks larger than 64 K bytes will have to wait until Volume II.

CONCLUSION

This chapter had two objectives. First, I wanted you to get a sense of how and when the string instructions can best be applied. Second, I wanted you to heighten your regard for these instructions, which are the best the 8088 has to offer. With any luck, this chapter has both broadened your horizons for string instruction applications and increased your respect for these unique and uniquely powerful members of the 8088's instruction set.