# Chapter 9: Around and About the Instruction Set

So far, we've covered assembler programming in a fairly linear fashion, with one topic leading neatly to the next and with related topics grouped by chapter. Alas, assembler programming isn't so easily pigeonholed. For one thing, the relationships between the many facets of assembler programming are complex; consider how often I've already mentioned the string instructions, which we have yet to discuss formally. For another, certain aspects of assembler stand alone, and are simply not particularly closely related to any other assembler topic.

Some interesting members of the 8088's instruction set fall into the category of stand-alone topics, as do unusual applications of a number of instructions. For example, while the knowledge that **inc ax** is a byte shorter than **inc al** doesn't have any far-reaching implications, that knowledge can save a byte and a few cycles when applied properly. Likewise, the use of **cbw** to convert certain unsigned byte values to word values is a self-contained programming technique.

Over the last few chapters, we've covered the 8088's registers, memory addressing, and 8080-influenced instructions. In this chapter, we'll touch on more 8088 instructions. Not all the instructions, by any means (remember, I'm assuming you already know 8088 assembler) but rather those instructions with subtle, useful idiosyncracies. These instructions fall into the class described above--well worth knowing but unrelated to one

another--so this chapter will be a potpourri of assembler topics, leaping from one instruction to another.

In the next chapter we'll return to a more linear format as we discuss the string instructions. After that we'll get into branching, look-up tables, and more. For now, though, hold on to your hat as we bound through the instruction set.

## SHORTCUTS FOR HANDLING ZERO AND CONSTANTS

The instruction set of the 8088 can perform any of a number of logical and arithmetic operations on byte- and word-sized, signed and unsigned integer values. What's more, those values may be stored either in registers or in memory. Much of the complexity of the 8088's instruction set results from this flexibility--and so does the slow performance of many of the 8088's instructions. However, some of the 8088's instructions can be used in a less flexible--but far speedier--fashion. Nowhere is this more apparent than in handling zero.

Zero pops up everywhere in assembler programs. Up counters are initialized to zero. Down counters are counted down to zero. Flag bytes are compared to zero. Parameters of value zero are passed to subroutines. Zero is surely the most commonly-used value in assembler programming--and the easiest value to handle, as well.

### MAKING ZERO

For starters, there is almost never any reason to assign the immediate value zero to a register. Why assign zero to a register when **sub**

**reg,reg** or **xor reg,reg** always zeros the register in fewer cycles (and also in fewer bytes for 16-bit registers)? The only time you should assign the value zero to a register rather than clearing the register with **sub** or **xor** is when you need to preserve the flags, since **mov** doesn't affect the flags but **sub** and **xor** do.

## INITIALIZING CONSTANTS FROM THE REGISTERS

As we discussed in the last chapter, it pays to clear a direct-addressed memory variable by zeroing AL or AX and storing that register to the memory variable. If you're setting two or more direct-addressed variables to any specific value (and here we're talking about any value, not just zero), it's worth storing that value in the accumulator and then storing the accumulator to the memory variables. (When initializing large blocks of memory, **rep stos** works better still, as we'll see in Chapter 10.) The basic principle is this: avoid extra immediate-operand bytes by storing frequently-used constants in registers and using the registers as operands.

Listing 9-1 provides an example of initializing multiple memory variables to the same value. This listing, which stores 0FFFFh in AX and then stores AX to three memory variables, executes in 17.60 us per three-word initialization. That's more than 28% faster than the 22.63 us per initialization of Listing 9-2, which stores the immediate value 0FFFFh to each of the three words. Listing 9-1 is that much faster than Listing 9-2 even though Listing 9-1 is one instruction longer per initialization. The difference? Each of the three **mov** instructions in Listing 9- 2 is 3 bytes longer than the

corresponding **mov** in Listing 9-1: two bytes are taken up by the immediate value 0FFFFh, and one extra byte is required because the accumulator-specific direct- addressing form of **mov** isn't used. That's a total of 9 extra bytes for the three **mov** instructions of Listing 9-2, more than offsetting the 3 bytes required by the extra instruction **mov ax,0ffffh** of Listing 9-1. (Remember, the 8088 doesn't sign- extend immediate operands to **mov**.) As always, those extra bytes take 4 cycles each to fetch.

Shorter is better.

If you're initializing more than one register to zero, you can save 1 cycle per additional register by initializing just one of the registers, then copying it to the other registers, as follows:

```
sub     si,si     ;point to offset 0 in DS
mov     di,si     ;point to offset 0 in ES
mov     dx,si     ;initialize counter to 0
```

While **mov reg,reg** is 2 bytes long, the same as **sub reg,reg**, according to the official specs **mov** is the faster of the two by 1 cycle. Whether this translates into any performance advantage depends on the code mix--if the prefetch queue is empty, code fetching time will dominate and **mov** will have no advantage--but it can't hurt and might help.

Similarly, if you're initializing multiple 8-bit registers to the same non-zero value, you can save up to 2 cycles per additional register by initializing one of the registers and copying it to the other(s). While **mov reg,immed8** is 2 cycles slower than **mov reg,reg**, both instructions are the

same size.

Finally, if you're initializing multiple 16-bit registers to the same non-zero value, it always pays to initialize one register and copy it to the other(s). The reason: **mov reg,immed16**, at 3 bytes in length, is a byte longer (and 2 cycles slower) than **mov reg,reg**.

**INITIALIZING TWO BYTES WITH A SINGLE mov**

While we're on the topic of initializing registers and variables, let's take a quick look at initializing paired bytes. Suppose we want to initialize AH to 16h and AL to 1. The obvious solution is to set each register to the desired value:

```
mov     ah,16h
mov     al,1
```

However, a better solution is to set the pair of registers with a single **mov**:

```
mov     ax,1601h
```

The paired-register initialization is a byte shorter and 4 cycles faster...and does exactly the same thing as the separate initializations!

A trick that makes it easier to initialize paired 8-bit registers is to shift the value for the upper register by 8 bits. For example, the last initialization could be performed as:

```
mov     ax,(16h shl 8) + 1
```

This method has two benefits. First, it's easy to distinguish between the values for the upper and lower registers; 16 and 1 are easy to pick out in the above example. Second, it's much simpler to handle non-hexadecimal values by shifting and adding. You must admit that:

```
mov     dx,(201 shl 8) + 'A'
```

is easier to write and understand than:

```
mov     dx,0c941h       ;DH=201, DL='A'
```

You need not limit paired-byte initializations to registers. Adjacent byte-sized memory variables can be initialized with a single word access as well. If you do use paired-byte initializations of memory variables, though, be sure to place prominent comments around the memory variables; otherwise, you or someone else might accidentally separate the pair at a later date, ruining the initialization.

## MORE FUN WITH ZERO

What else can we do with zero? Well, we can test the zero/non-

zero status of a register with either **and reg,reg** or **or reg,reg**. Both of these instructions set the Zero flag just as **cmp reg,0** would...and they execute faster and are anywhere from 0 to 2 bytes shorter than **cmp**. (Both **and reg,reg** and **or reg,reg** are guaranteed to be at least 1 byte shorter than **cmp reg,0** except when **reg** is AL, in which case all three instructions are the same length.) Listing 9-3, which uses **and dx,dx** to test for the zero status of DX, clocks in at 3.62 us per test. That's 25% faster than the 4.53 us per test of Listing 9-4, which uses **cmp dx,0**.

As described in the last chapter, it is (surprisingly) faster to load the accumulator from a direct-addressed memory variable and **and** or **or** the accumulator with itself in order to test whether that memory variable is zero than it is to simply compare the memory variable with an immediate operand. For instance:

```
mov     al,[ByteFlag]
and     al,al
jnz     FlagNotZero
```

is equivalent to and faster than:

```
cmp     [ByteFlag],0
jnz     FlagNotZero
```

Finally, there are some cases in which tests that are really not zero/non-zero tests can be converted to tests for zero. For example, consider a test to check whether or not DX is 0FFFFh. We could use **cmp**

**dx,0ffffh**, which is three bytes long and takes 4 cycles to execute.  On the other hand, if we don't need to preserve DX (that is, if we're performing a one-time-only test) we could simply use **inc dx**, which is only one byte long and takes just 2 cycles to execute, and then test for a zero/non-zero status. So, if we don't mind altering DX in the course of the test:

```
cmp     dx,0ffffh
jnz     NotFFFF
```

and:

```
inc     dx
jnz     NotFFFF
```

are functionally the same...save that the latter version is much smaller and faster.

A similar case of turning a test into a zero/non-zero test occurs when testing a value for membership in a short sequence of consecutive numbers--the equivalent of a C switch construct with just a few cases consisting of consecutive values.  (Longer and/or non-consecutive sequences should be handled with look-up tables.)  For example, suppose that you want to perform one action if CX is 4, another if CX is 3, a third action if CX is 2, and yet another if CX is 1.  Listing 9-5, which uses four **cmp** instructions to test for the four cases of interest, runs in 17.01 us per switch handled.  That's a good 4.94 us slower per switch than the 12.07 us of

Listing 9-6, so Listing 9-5 runs at less than 75% of the speed of Listing 9-6. Listing 9-6 gets its speed boost by using the 1-byte **dec cx** instruction rather than the 3-byte **cmp cx,immed8** instruction to test for each of the four cases, thereby turning all the tests into zero/non-zero tests.

Unorthodox, yes--but very effective. The moral is clear: even when the 8088 has an instruction that's clearly intended to perform a given task (such as **cmp** for comparing), don't assume that instruction is the best way to perform that task under all conditions.

**inc AND dec**

**inc** and **dec** are simple, unpretentious instructions--and more powerful than you might imagine. Since **inc** and **dec** require only one operand (the immediate value 1 that's added or subtracted is implied by the instruction), they are among the shortest (1 to 4 bytes) and fastest (2 to 3 cycles for a register operand, but up to 35 for a word-sized memory operand--keep your operands in registers!) instructions of the 8088. In particular, when working with 16-bit register operands, **inc** and **dec** are the fastest arithmetic instructions of the 8088, with an execution time of 2 cycles paired with a length of just 1 byte.

How much difference does it make to use **inc** or **dec** rather than **add** or **sub**? When you're manipulating a register, the answer is: a lot. In fact, it's actually better to use two **inc** instructions to add 2 to a 16-bit register than to add 2 with a single **add**, because a single **add** with an immediate operand of 2 is 3 bytes long, three times the length of a 16-bit

register **inc**. (Remember, shorter is better, thanks to the prefetch queue cycle- eater.)

The same is true of **dec** versus **sub** as of **inc** versus **add**. For example, the code in Listing 9-7, which uses a 16-bit register **dec** instruction, clocks in at 5.03 us per loop, 33% faster than the 6.70 us of the code in Listing 9-8, which uses a **sub** instruction to decrement DX.

The difference between the times of Listings 9-7 and 9-8 is primarily attributable to the 8 cycles required to fetch the two extra bytes of the **sub** instruction. To illustrate that point, consider Listing 9-9, which decrements DX twice per loop. Listing 9-9 executes in 5.80 us per loop, approximately halfway between the times of Listings 9-7 and 9-8. That's just what we'd expect, since the loop in Listing 9-9 is 1 byte longer than the loop in Listing 9-7 and 1 byte shorter than the loop in Listing 9-8.

Use **inc** or **dec** in preference to **add** or **sub** whenever possible.

(Actually, when SP is involved there's an exception to the above rule for code that will run on 80286- or 80386-based computers. Such code should use **add**, **sub**, **push**, and **pop** to alter SP in preference to **inc** and **dec**, because an odd stack pointer is highly undesirable on 16- and 32-bit processors. I'll cover this topic in detail in Chapter 15.)

I'd like to pause at this point to emphasize that the 16-bit register versions of **inc** and **dec** are different beasts from the run-of-the-mill **inc** and **dec** instructions. As with the 16-bit register **xchg**-with-AX instructions we discussed in the last chapter, there are actually two separate **inc** instructions on the 8088, one of which is a superset of the other. (The same is true of

**dec**, but we'll just discuss **inc** for now.)

Figure 9-1 illustrates the two forms of **inc**. While the special form is limited to 16-bit register operands, it has the advantage of being a byte shorter and a cycle faster than the mod-reg-rm register form, even when both instructions operate on the same register. As you'd expect, 8088 assemblers automatically use the more efficient special version whenever possible, so you don't need to select between the two forms explicitly. However, it's up to you to use 16-bit register **inc** (and **dec**) instructions whenever you possibly can, since only then can the assembler assemble the more efficient form of those instructions.

For example, Listing 9-7, which uses the 1-byte-long 16-bit register form of **dec** to decrement the 16-bit DX register, executes in 5.03 us per loop, 15% faster than Listing 9-10, which uses the 2-byte-long mod-reg-rm form of **dec** to decrement the 8- bit DL register and executes in 5.79 us per loop.

## USING 16-BIT inc AND dec INSTRUCTIONS FOR 8-BIT OPERATIONS

If you're clever, you can sometimes use the 16-bit form of **inc** or **dec** even when you only want to affect an 8-bit register. Consider Listing 9-11, which uses AL to count from 0 to 8. Since AL will never pass 0FFh and turn over (the only circumstance in which **inc ax** modifies AH), it's perfectly safe to use **inc ax** rather than **inc al**. In this case, both instructions always produce the same result; however, **inc ax** produces that result considerably more rapidly than **inc al**. If you do use such a technique, however,

remember that the flags are set on the basis of the whole operand. For example, **dec ax** will set the Zero flag only when both AH and AL--not AL alone--go to zero. This seems obvious, but if you're thinking of AL as the working register, as in Listing 9-11, it's easy to forget that **dec ax** sets the flags to reflect the status of AX, not AL.

To carry the quest for **inc** and **dec** efficiency to the limit, suppose we're constructing code which contains nested countdown loops. Suppose further that all registers but CX are in use, so all we've got available for counters are CH and CL. Normally, we would expect to use two 8-bit **dec** instructions here. However, we know that the counter for the inner loop is 0 after the loop is completed, so we've got an opportunity to perform a 16-bit **dec** for the outer loop if we play our cards right.

Listing 9-12 shows how this trick works. CH is the counter for the inner loop, and we are indeed stuck with an 8-bit **dec** for this loop. However, by the time we get around to using CL as a counter, CH is guaranteed to be 0, so we can use a 16-bit **dec cx** for the outer loop. Granted, it would be preferable to place the 16-bit **dec** in the time-critical inner loop, and if that loop were long enough, we might well do that by pushing CX for the duration of the inner loop; nonetheless, a 16-bit **dec** is preferable in any loop, and in Listing 9-12 we get the benefits of a 16-bit **dec** at no cost other than a bit of careful register usage.

By the way, you've likely noticed that Listing 9-12 fairly begs for a **loop** instruction at the bottom of the outer loop. That's certainly the most efficient code in this case; I've broken the **loop** into a **dec** and a **jnz** only for

illustrative purposes.

**HOW inc AND add (AND dec AND sub) DIFFER--AND WHY**

**inc** and **dec** are not exactly the same as **add 1** and **sub 1**. Unlike addition and subtraction, **inc** and **dec** don't affect the Carry flag. This can often be a nuisance, but there is a good use for this quirk of **inc** and **dec**, and that's in adding or subtracting multi-word memory values.

Multi-word memory values are values longer than 16 bits that are stored in memory. On the 8088 such values can only be added together by a series of 16- and/or 8-bit additions. The first addition--of the least-significant words--must be performed with **add**, or with **adc** with the Carry flag set to 0. Subsequent additions of successively more-significant words must be performed with **adc**, so that the carry-out can be passed from one addition to the next via the Carry flag. The same is true of **sub**, **sbb**, and borrow for subtraction of multi-word memory variables.

Some way is needed to address each of the words in a multi-word memory value in turn, so that each part of the value may be used as an operand. Consequently, multi-word memory values are often pointed to by registers (BP or BX and/or SI or DI), which can be advanced to point to successively more-significant portions of the values as addition or subtraction proceeds. If, however, there were no way to advance a memory-addressing register without modifying the Carry flag, then **adc** and **sbb** would only work properly if we preserved the Carry flag around the **inc** instructions, with **pushf** and **popf** or **lahf** and **sahf**.

**inc** and **dec** don't affect the Carry flag, however, and that greatly simplifies the process of adding multi-word memory variables. The code in Listing 9-13, which adds together two 64bit memory variables--one pointed to by SI and the other pointed to by DI--only works because the **inc** instructions that advance the pointers don't affect the Carry flag values that join the additions of the various parts of the variables. (It's equally important that **loop** doesn't affect any flags, as we'll see in Chapter 14.)

## CARRYING RESULTS ALONG IN A FLAG

As mentioned in Chapter 6 and illustrated in the last section, many instructions don't affect all the flags, and some don't affect any flags at all. You can take advantage of this by carrying a status along in the FLAGS register for several instructions before testing that status. Of course, if you do choose to carry a status along, all of the instructions executed between setting the status and testing it must leave the status alone.

For example, the following code tests AL for a specific value, then sets AL to 0 even before branching according to the results of the test:

```
cmp     al,RESET_FLAG   ;sets Z to reflect test result
mov     al,0                    ;set AL for the code following the
                        ; branch
                        ;*** NOTE: THIS INSTRUCTION MUST ***
                        ;*** NOT ALTER THE Z FLAG!       ***
jz      IsReset         ;branch according the to Z flag set
                        ; by CMP
```

In this example, AL must be set to 0 no matter which way the branch goes. If we were to set AL after the branch rather than before, two **mov al,0**

instructions--one for each code sequence that might follow **jz IsReset**--would be needed. If we set AL before the **cmp** instruction, the test couldn't even be performed because the value under test in AL would be lost. In very specific cases such as this, clear advantages result from carrying a status flag along for a few instructions.

One caution when using the above approach: never set a register to zero via **sub reg,reg** or **xor reg,reg** while carrying a status along. With time, you'll get in the habit of setting registers to zero with **sub reg,reg** or **xor reg,reg**, either of which is faster (and often smaller) than **mov reg,0**. Unfortunately, **sub** and **xor** affect the flags, while **mov** doesn't. For example:

```
cmp     al,RESET_FLAG   ;sets Z to reflect status under test
sub     al,al           ;alters Z, causing the code to
                        ; malfunction
jz      IsReset         ;won't jump properly
```

fails to preserve the Zero flag between the **cmp** and the **jz**, and wouldn't work properly. In cases such as this, always be sure to use **mov**.

The bugs that can arise from the use of a carried-along status that is accidentally wiped out are often hard to reproduce and difficult to track down, so all possible precautions should be taken whenever this technique is used. No more than a few instructions--and no branches--should occur between the setting and the testing of the status. The use of a carried-along status should always be clearly commented, as in the first example in this section. Careful commenting is particularly important in

order to forestall trouble should you (or worse, someone else) alter the code at a later date without noticing that a status is being carried along.

If you do need to carry a status along for more than a few instructions, store the status with either **pushf** or **lahf**, then restore it later with **popf** or **sahf**, so there's no chance of the intervening code accidentally wiping the status out.

## BYTE-TO-WORD AND WORD-TO-DOUBLEWORD CONVERSION

On the 8088 the need frequently arises to convert byte values to word values.  A byte value might be converted to a word in order to add it to a 16-bit value, or in order to use it as a pointer into a table (remember that only 16-bit registers can be used as pointers, with the lone exception of AL in the case of **xlat**).  Occasionally it's also necessary to convert word values to doubleword values.  One application for word-to-doubleword conversion is the preparation of a 16-bit dividend for 32-bit by 16-bit division.

Unsigned values are converted to a larger data type by simply zeroing the upper portion of the desired data type.  For example, an unsigned byte value in DL is converted to an unsigned word value in DX with:

```
sub     dh,dh
```

Likewise, an unsigned byte value in AL can be converted to a doubleword value in DX:AX with:

```
sub     dx,dx
mov     ah,dh
```

In principle, conversion of a signed value to a larger data type is more complex, since it requires replication of the high (or sign) bit of the original value throughout the upper portion of the desired data type. Fortunately, the 8088 provides two instructions that handle the complications of signed conversion for us: **cbw** and **cwd**. **cbw** sets all the bits of AH to the value of bit 7 of AL, performing signed byte-to-word conversion. **cwd** sets all the bits of DX to the value of bit 15 of AX, performing signed word-to-doubleword conversion.

There's nothing tricky about **cbw** and **cwd**, and you're doubtless familiar with them already. What's particularly interesting about these instructions is that they're each only 1 byte long, 1 byte shorter than **sub reg,reg**. What's more, the official execution time of **cbw** is only 2 cycles, so it's 1 cycle faster than **sub** as well. **cwd**'s official execution time is 5 cycles, but since it's shorter than **sub**, it will actually often execute more rapidly than **sub**, thanks to the prefetch queue cycle-eater.

What all this means is that **cbw** and **cwd** are the preferred means of converting values to larger data types, and should be used whenever possible. In particular, you should use **cbw** to convert unsigned bytes in the range 0-7Fh to unsigned words. While it may seem strange to use a signed type-conversion instruction to convert unsigned values, there's no distinction

between unsigned bytes in the range 0 to 7Fh and signed bytes in the range 0 to +127, since they have the same values and have bit 7 set to 0.

Listing 9-14 illustrates the use of **cbw** to convert an array of unsigned byte values between 0 and 7Fh to an array of word values. Note that values are read from memory and written back to memory and the loop counter is decremented, so this is a realistic usage of **cbw** rather than an artificial situation designed to show the instruction in the best possible light. Despite all the other activity occurring in the loop, Listing 9- 14 executes in 10.06 us per loop, 12% faster than Listing 9-15, which executes in 11.31 us per loop while using **sub ah,ah** to perform unsigned byte-to-word conversion.

**cwd** can be used in a similar manner to speed up the conversion of unsigned word values in the range 0-7FFFh to doubleword values. Another clever use of **cwd** is as a more efficient way than **sub reg,reg** to set DX to 0 when you're certain that bit 15 of AX is 0 or as a better way than **mov reg,0FFFFh** to set DX to 0FFFFh when you're sure that bit 15 of AX is 1. Similarly, **cbw** can be used as a faster way to set AH to 0 whenever bit 7 of AL is 0 or to 0FFh when bit 7 of AL is 1.

Viewed objectively, there's no distinction between using **cbw** to convert AL to a signed word, to zero AH when bit 7 of AL is 0, and to set AH to 0FFh when bit 7 of AL is 1. In all three cases each bit of AH is set to the value of bit 7 of AL. Viewed conceptually, however, it can be useful to think of **cbw** as capable of performing three distinct functions: converting a signed value in AL to a signed value in AX, setting AH to 0 when bit 7 of AL is

0, and setting AH to 0FFh when bit 7 of AL is 1. After all, an important aspect of the Zen of assembler is the ability to view your resources (such as the instruction set) from the perspective most suited to your current needs. Rather than getting locked in to the limited functionality of the instruction set as it was intended to be used, you must tap into the functionality of the instruction set as it is capable of being used.

Listing 9-14 is an excellent example of how focusing too closely on a particular sort of optimization or getting too locked into a particular meaning for an instruction can obscure a better approach. In Listing 9-14, aware that the values in the array are less than 80h, we cleverly use **cbw** to set AH to 0. This means that AH is set to zero every time through the loop--even though AH never changes from one pass through the loop to the next! This makes sense only if you view each byte-to-word conversion in isolation. Listing 9-16 shows a more sensible approach, in which AH is set to 0 just once, outside the loop. In Listing 9-16, each byte value is automatically converted to a word value in AX simply by being loaded into AL.

In the particular case of Listing 9-16, it happens that moving the setting of AH to 0 outside the loop doesn't improve performance; Listing 9-16 runs at exactly the same speed as Listing 9-14, no doubt thanks to the prefetch queue and DRAM refresh cycle-eaters. That's just a fluke, though--on average, an optimization such as the one in Listing 9-16 will save about 4 cycles. Don't let the quirks of the 8088 deter you from the pursuit of saving bytes and cycles--but do remember to always time your code to make sure you've improved it!

If for any reason AH did change each time through the loop, we could no longer use the method of Listing 9-16, and Listing 9- 14 would be a good alternative. That's why there are no hard- and-fast rules that produce the best assembler code. Instead, you must respond flexibly to the virtually infinite variety of assembler coding situations that arise. The bigger your bag of tricks, the better off you'll be.

**xchg IS HANDY WHEN REGISTERS ARE TIGHT**

One key to good assembler code is avoiding memory and using the registers as much as possible. When you start juggling registers in order to get the maximum mileage from them, you'll find that **xchg** is a good friend.

Why? Because the 8088's general-purpose registers are actually fairly special-purpose. BX is used to point to memory, CX is used to count, SI is used with **lods**, and so on. As a result, you may want to use a specific register for two different purposes in a tight loop. **xchg** makes that possible.

Consider the case where you need to handle both a loop count and a shift count. Ideally, you would want to use CX to store the loop count and CL to store the shift count. Listing 9-17 uses CX for both purposes by pushing and popping the loop count around the use of the shift count. However, this solution is less than ideal because **push** and **pop** are relatively slow instructions. Instead, we can use **xchg** to swap the lower byte of the loop count with the shift count, giving each a turn in CL, as shown in Listing

9-18. Listing 9-18 runs in 15.08 us per byte processed, versus the 20.11 us time of Listing 9-17. That's a 33% improvement from a seemingly minor change! The secret is that **push** and **pop** together take 27 cycles, while a register- register **xchg** takes no more than 4 cycles to execute once fetched and only 8 cycles even when the prefetch queue is empty.

Neither Listing 9-17 or Listing 9-18 is the most practical solution to this particular problem. A better solution would be to simply store the loop count in a register other than CX and use **dec**/**jnz** rather than **loop**. The object of this exercise wasn't to produce ideal code, but rather to illustrate that **xchg** gives you both speed and flexibility when you need to use a single register for more than one purpose.

**xchg** is also useful when you need more memory pointers in a loop than there are registers that can point to memory. See Chapter 8 for an example of the use of **xchg** to allow BX to point to two arrays. As the example in Chapter 8 also points out, the form of **xchg** used to swap AX with another general-purpose register is 1 byte shorter than the standard form of **xchg**.

Finally, **xchg** is useful for getting and setting a memory variable at the same time. For example, suppose that we're maintaining a flag that's used by an interrupt handler. One way to get the current flag setting and force the flag to zero is:

```
cli                             ;turn interrupts off
mov     al,[Flag]  ;get the current flag value
mov     [Flag],0   ;set the flag to 0
sti                             ;turn interrupts on
```

(It's necessary to disable interrupts to ensure that the interrupt handler doesn't change **Flag** between the instruction that reads the flag and the instruction that resets it.)

With **xchg**, however, we can do the same thing with just two instructions:

```
sub     al,al       ;set AL to 0
xchg    [Flag],al   ;get the current flag value and
                    ; set the flag to 0
```

Best of all, we don't need to disable interrupts in the **xchg**- based code, since interrupts can only occur between instructions, not during them! (Interrupts can occur between repetitions of a repeated string instruction, but that's because a single string instruction is actually executed multiple times when it's repeated. We'll discuss repeated string instructions at length in Chapters 10 and 11.)

## DESTINATION: REGISTER

Many arithmetic and logical operations can be performed with a register as one operand and a memory location as the other, with either one being the source and the other serving as the destination. For example, both of the following forms of **sub** are valid:

```
sub     [bx],al
```

```
sub         al,[bx]
```

The two instructions are not the same, of course. Memory is the destination in the first case, while AL is the destination in the second case. That's not the only distinction between the two instructions, however. There's also a major difference in the area of performance.

Consider this. Any instruction, such as **sub**, that has a register source operand and a memory destination operand must access memory twice: once to fetch the destination operand prior to performing an operation, and once to store the result of the operation to the destination operand. By contrast, the same instruction with a memory source operand and a register destination operand must access memory just once, in order to fetch the source value from memory. Consequently, having a memory operand as the destination imposes an immediate penalty of at least 4 cycles per instruction, since each memory access takes a minimum of 4 cycles.

As it turns out, however, the extra time required to access destination memory operands with such instructions--which include **adc**, **add**, **and**, **or**, **sbb**, **sub**, and **xor**--is not 4 but 7 cycles, according to the official specs in Appendix A. We can measure the actual difference by timing the code in Listings 9-19 and 9- 20. As it turns out, the code with AL as the destination takes just 5.03 us per instruction. That's 1.00 us (4.77 cycles) or nearly 20% faster than the code with memory as the destination operand, which takes 6.03 us per instruction.

The moral of the story? Simply to keep those operands which

tend to be destination operands most frequently--counters, pointers, and the like--in registers whenever possible. The ideal situation is one in which both destination and source operands are in registers.

By the way, remember that an instruction with a word-sized memory operand requires an additional 4 cycles per memory access to access the second byte of the word. Consequently:

```
add     [si],dx     ;performs 2 word-sized accesses
                    ; (= 4 byte-sized accesses)
```

takes 8 cycles longer than:

```
add     [si],dl     ;performs 2 byte-sized accesses
```

However:

```
add     dx,[si]     ;performs 1 word-sized access
                    ; (= 2 byte-sized accesses)
```

takes only 4 cycles longer than:

```
add     dl,[si]     ;performs 1 byte-sized access
```

since only one memory access is performed by each.

A final note: at least one 8088 reference lists **cmp** as requiring

the same 7 additional cycles as **sub** when used with a memory operand that is the destination rather than the source. Not so--**cmp** requires the same time no matter which operand is a memory operand. That makes sense, since **cmp** doesn't actually modify the destination operand and so has no reason to perform a second memory access. The same is true for **test**, which doesn't modify the destination operand.

**neg AND not**

**neg** and **not** are short, fast instructions that are sometimes undeservedly overlooked. Each instruction is 2 bytes long and executes in just 3 cycles when used with a register operand, and each instruction can often replace a longer instruction or several instructions.

**not mem/reg** is similar to **xor mem/reg,0ffffh** (or **xor mem/reg,0ffh** for 8-bit operands), but is usually 1 byte shorter and 1 cycle faster. (If mem/reg is AL, **not** and **xor** are the same length, but **not** is still 1 cycle faster.) Another difference between the two instructions is that unlike **xor**, **not** doesn't affect any of the status flags. This can be useful for, say, toggling the state of a flag byte without disturbing the statuses that an earlier operation left in the FLAGS register.

**neg** negates a signed value in a register or memory variable. You can think of **neg** as subtracting the operand from 0 and storing the result back in the operand. The flags are set to reflect this subtraction from 0, so **neg ax** sets the flags as if:

```
mov     dx,ax
mov     ax,0
sub     ax,dx
```

had been performed.

One interesting consequence of the way in which **neg** sets the flags is that the Carry flag is set in every case except when the operand was originally 0. (That's because in every other case a value larger than 0 is being subtracted from 0, resulting in a borrow.) This is very handy for negating 32-bit operands quickly. In the following example, DX:AX contains a 32-bit operand to be negated:

```
neg     dx
neg     ax
sbb     dx,0
```

Although it's not obvious, the above code does indeed negate DX:AX, and does so very quickly indeed. (You might well think that there couldn't possibly be a faster way to negate a 32-bit value, but in Chapter 13 we'll see a decidedly unusual approach that's faster still. Be wary of thinking you've found the fastest possible code for any task!)

How does the above negation code work? Well, normally we would want to perform a two's complement negation by flipping all bits of the operand and then adding 1 to it, as follows:

```
not     dx      ;flip all bits...
not     ax      ;...of the operand
add     ax,1    ;remember, INC doesn't set the Carry flag!
adc     dx,0    ;then add 1 to finish the two's complement
```

However, this code is 10 bytes long, a full 3 bytes longer than our optimized negation code. In the optimized code, the first negation word flips all bits of DX and adds 1 to that result, and the second negation flips all bits of AX and adds 1 to that result. At this point, we've got a perfect two's complement result, except that 1 has been added to DX. That's incorrect-- unless AX was originally 0. Aha! Thanks to the way **neg** sets the flags, the Carry flag is always set except when the operand was originally 0. Consequently, we need only to subtract from DX the carry-out from **neg ax** and we've got a 32-bit two's-complement negation--in just 7 bytes!

By the way, 32-bit negation can also be performed with the three instruction, 7-cycle sequence:

```
not     dx
neg     ax
sbb     dx,-1
```

If you can understand why this sequence works, you've got a good handle on **neg**, **not**, and two's complement arithmetic. (Hint: the underlying principle in the last sequence is exactly the same as with the **neg**/**neg**/**sbb** approach we just discussed.) If not, wait until Chapter 13, in which we'll explore the workings of 32-bit negation in considerable detail.

**neg** is also handy for generating differences without using **sub** and without using other registers. For example, suppose that we're scanning a list for a match with AL. **repnz scasb** (which we'll discuss further in Chapter 10) is ideal for such an application. However, after **repnz scasb** has found a match, CX contains the number of entries in the list that weren't scanned, not the number that were scanned, and it's the latter number that we want in CX. Fortunately, we can use **neg** to convert the entries-remaining count in CX into an entries-scanned count, as follows:

```
; The value to search for is already in AL, and ES:DI
; already points to the list to scan.
        mov     cx,[NumberOfEntries] ;# of entries to scan
        cld                                     ;make SCASB count up
        repnz scasb                     ;look for the value
        jnz     ValueNotFound           ;the value is not in the list
        neg     cx                              ;the # of entries not scanned
                                        ; times -1
        add     cx,[NumberOfEntries] ;total # of entries - # of
                                        ; entries not scanned = # of
                                        ; entries scanned
```

Thanks to **neg**, this replaces the longer code sequence:

```
; The value to search for is already in AL, and ES:DI
; already points to the list to scan.
        mov     cx,[NumberOfEntries] ;# of entries to scan
        cld                                     ;make SCASB count up
        repnz scasb                     ;look for the value
        jnz     ValueNotFound           ;the value is not in the list
        mov     ax,[NumberOfEntries] ;total # of entries
        sub     ax,cx                   ;total # of entries - # of
                                        ; entries not scanned = # of
                                        ; entries scanned
        mov     cx,ax                   ;put the result back in CX
```

Another advantage of **neg** in the above example is that it lets us

generate the entries-remaining count without using another register. By contrast, the alternative approach requires the use of a 16-bit register for temporary storage. When registers are in short supply--as is usually the case--the register-conserving nature of **neg** can be most useful.

**ROTATES AND SHIFTS** Next, we're going to spend some time going over interesting aspects of the various shift and rotate instructions. To my mind, the single most fascinating thing about these instructions concerns their ability to shift or rotate by either 1 bit or the number of bits specified by CL; in particular, it's most informative to examine the relative performance of the two approaches for multi-bit operations.

It's much more desirable than you might think to perform multi-bit shifts and rotates by repeating the shift or rotate CL times, as opposed to using multiple 1-bit shift or rotate instructions. As is so often the case, the cycle counts in Appendix A are misleading in this regard. As it turns out, shifting or rotating multiple bits by repeating an instruction CL times, as in:

```
mov     cl,4
shr     ax,cl
```

is almost always faster than shifting by 1 bit repeatedly, as in:

```
shr     ax,1
shr     ax,1
shr     ax,1
shr     ax,1
```

This is true even though the official specs in Appendix A indicate that the latter approach is more than twice as fast.

Shifting or rotating by CL also requires fewer instruction bytes for shifts of more than 2 bits. In fact, that reduced instruction byte count is precisely the reason the shift/rotate by CL approach is faster. As we saw in Chapter 4, fetching the instruction bytes of **shr ax,1** takes up to four cycles per byte; each shift or rotate instruction is 2 bytes long, so **shr ax,1** can take as much as 8 cycles per bit shifted. By contrast, only 4 instruction bytes in total need to be fetched in order to load CL and execute **shr ax,cl**. Once those bytes are fetched, **shr ax,cl** runs at its Execution Unit speed of 4 cycles per bit shifted, since no additional instruction fetching is needed. Better yet, the next instruction's bytes can be prefetched while a shift or rotate by CL executes.

The point is not that shifts and rotates by CL are faster than you'd expect, but rather that 1-bit shifts and rotates are slower than you'd expect, courtesy of the prefetch queue cycle- eater. The question is, of course, at what point does it become faster to shift or rotate by CL instead of using multiple 1-bit shift or rotate instructions?

To answer that, I've timed the two approaches, shown in Listings 9-21 and 9-22, for shifts ranging from 1 to 7 bits, by altering the equated value of BITS_TO_SHIFT accordingly. The results are as follows:

| Bits shifted (BITS_TO_SHIFT) | Time taken to shift by CL (Listing 9-21) | Time taken to shift 1 bit at a time (Listing 9-22) |
|---|---|---|

| | | |
|---|---|---|
| 1 | 3.6 us | 1.8 us |
| 2 | 4.2 us | 3.6 us |
| 3 | 5.0 us | 5.4 us |
| 4 | 5.9 us | 7.2 us |
| 5 | 6.7 us | 9.1 us |
| 6 | 7.5 us | 10.9 us |
| 7 | 8.4 us | 12.7 us |

Astonishingly, it hardly ever pays to shift or rotate by multiple bit places with separate 1-bit instructions.  The prefetch queue cycle-eater exacts such a price on 1-bit shifts and rotates that it pays to shift or rotate by CL for shifts of 3 or more bits.  Actually, the choice is not entirely clear-cut for 3- to 5-bit shifts/rotates, since the 1-bit-at-a-time approach can become relatively somewhat faster if the prefetch queue is full when the shift/rotate sequence begins.  Still, there's no question but what shifting or rotating by CL is as good as or superior to using multiple 1-bit shifts for most multi-bit shifts.

By the way, you should be aware that the contents of CL are not changed when CL is used to supply the count for a shift or rotate instruction. This allows you to load CL once and then use it to control multiple shift and/or rotate instructions.

**SHIFTING AND ROTATING MEMORY**

One feature of the 8088 that for some reason is often overlooked is the ability to shift or rotate a memory variable. True, the 8088 doesn't shift or rotate memory variables very rapidly, but the capability is there should

you need it. If you should find the need to perform a multi-bit shift or rotate on a memory variable, for goodness sakes use a CL shift! Every 1-bit memory shift/rotate takes a minimum of 20 cycles. By contrast, a shift-by-CL memory shift/rotate takes a minimum of 25 cycles, but only 4 additional cycles per bit shifted. It doesn't take a genius to see that for, say, a 4-bit rotate, the 41 cycles taken by the CL shift would beat the stuffing out of the 80 cycles taken by the four 1-bit shifts.

**ROTATES**

You should be well aware that there are two sorts of rotates. One category, made up of **rol** and **ror**, consists of rotates that simply rotate the bits in the operand, as shown in Figure 9-2. These instructions are useful for adjusting masks, swapping nibbles, and the like. For example:

```
mov     cl,4
ror     al,cl
```

swaps the high and low nibbles of AL. Note that these instructions don't rotate through the Carry flag. However, they do copy the bit wrapped around to the other end of the operand to the Carry flag as well. The other rotate category, made up of **rcl** and **rcr**, consists of rotates that rotate the operand through the Carry flag, as shown in Figure 9-3. These instructions are useful for multi- word shifts and rotates. For example:

```
shr         dx,1
rcr         cx,1
rcr         bx,1
rcr         ax,1
```

shifts the 64-bit value in DX:CX:BX:AX right one bit.

The rotate instructions affect fewer flags than you might think, befitting their role as bit-manipulation rather than arithmetic instructions. None of the rotate instructions affect the Sign, Zero, Auxiliary Carry, or Parity flags. On 1-bit left rotates the Overflow flag is set to the exclusive-or of the value of the resulting Carry flag and the most-significant bit of the result. On 1-bit right rotates the Overflow flag is set to the exclusive-or of the two most-significant bits of the result. (These Overflow flag settings indicate whether the rotate has changed the sign of the operand.) On rotates by CL the setting of the Overflow flag is undefined.

**SHIFTS**

Similarly, there are two sorts of shift instructions. One category, made up of **shl** (also known as **sal**) and **shr**, consists of shifts that shift out to the Carry flag, shifting a 0 into the vacated bit of the operand, as shown in Figure 9-4. These instructions are used for moving masks and bits about and for performing fast unsigned division and multiplication by powers of 2. For example:

```
shl         ax,1
```

multiplies AX, viewed as an unsigned value, by 2.

The other shift category contains only **sar**. **sar** performs the same shift right as does **shr**, save that the most significant bit of the operand is preserved rather than zeroed after the shift, as shown in Figure 9-5. This preserves the sign of the operand, and is useful for performing fast signed division by powers of 2. For example:

```
sar         ax,1
```

divides AX, viewed as a signed value, by 2.

The shift instructions affect the arithmetic-oriented flags that the rotate instructions leave alone, which makes sense since the shift instructions can perform certain types of multiplication and division. Unlike the rotate instructions, the shift instructions modify the Sign, Zero, and Parity flags in the expected ways. The setting of the Auxiliary Carry flag is undefined. The setting of the Overflow flag by the shift instructions is identical to the Overflow settings of the rotate instructions. On 1-bit left shifts the Overflow flag is set to the exclusive-or of the resulting Carry flag and the most- significant bit of the result. On 1-bit right shifts the Overflow flag is set to the exclusive-or of the two most- significant bits of the result.

Basically, any given shift will set the Overflow flag to 1 if the sign of the result differs from the sign of the original operand, thereby signalling that the shift has not produced a valid signed multiplication or division result. **sar** always sets the Overflow flag to 0, since **sar** can never change the sign of an operand. **shr** always sets the Overflow flag to the high-order bit of the original value, since the sign of the result is always positive. On shifts by CL the setting of the Overflow flag is undefined.

**SIGNED DIVISION WITH sar**

One tip if you do use **sar** to divide signed values: for negative dividends, **sar** rounds to the integer result of the next largest absolute value. This can be confusing, since for positive values **sar** rounds to the integer result of the next smallest absolute value, just as **shr** does. That is:

```
mov     ax,1
sar     ax,1
```

returns 1/2=0, while:

```
mov     ax,-1
sar     ax,1
```

doesn't return -1/2=0, but rather -1/2=-1. Similarly, **sar** insists that -5/4=-2, not -1. This is actually a tendency to round to the next integer value less than the actual result in all cases, which is exactly what **shr** also does. While that may be consistent, it's nonetheless generally a nuisance, since we

tend to expect that, say, -1/2*-1 should equal 1/2*1, but with **sar** we actually get 1 for the former and 0 for the latter.

The solution? For a signed division by n of a negative number with **sar**, simply add n-1 to the dividend before shifting. This compensates exactly for the rounding **sar** performs. For example:

```
                mov     ax,-1     ;sample dividend
                and     ax,ax     ;is the dividend negative?
                jns     DoDiv     ;it's positive, so we're ready to divide
                add     ax,2-1    ;it's negative, so we need to compensate.
                                  ; This is division by 2, so we'll
                                  ; add n-1 = 2-1
DoDiv:
                sar     ax,1              ;signed divide by 2
```

returns 0, just what we'd expect from -1/2.

That's a quick look at what the shift and rotate instructions were designed to do. Now let's bring a little Zen of assembler to bear in cooking up a use for **sar** that you can be fairly sure was never planned by the architects of the 8088.

**BIT-DOUBLING MADE EASY**

Think back to the bit-doubling example of Chapter 7, where we found that a bit-doubling routine based on register-register instructions didn't run nearly as fast as it should have, thanks to the prefetch queue. We boosted the performance of the routine by performing a table look-up, and that's the best solution that I know of. There is, however, yet another bit-doubling technique (conceived by my friend Dan Illowsky) that's faster than the original shift-based approach. Interestingly enough, this new technique

uses **sar**.

Let's consider **sar** as a bit-manipulation instruction rather than as a signed arithmetic instruction. What does **sar** really do? Well, it shifts all the bits of the operand 1 bit to the right, and it shifts bit 0 of the operand into the Carry flag. The most significant bit of the operand is left unchanged--and it is also shifted 1 bit to the right.

In other words, the most significant bit is doubled!

Once we've made the conceptual leap from **sar** as arithmetic instruction to **sar** as "bit-twiddler," we've got an excellent tool for bit-doubling. The code in Listing 7-14 placed the byte containing the bits to be doubled in two registers (BL and BH) and then doubled the bits with 4 instructions:

```
shr     bl,1
rcr     ax,1
shr     bh,1
rcr     ax,1
```

By contrast, the **sar** approach, illustrated in Listing 9-23, requires only one source register and doubles the bits with just 3 instructions:

```
shr     bl,1
rcr     ax,1
sar     ax,1
```

The **sar** approach requires only 75% as many code bytes as the approach in Listing 7-14. Since instruction fetching dominates the execution time of Listing 7-14, the shorter **sar**-based code should be considerably

faster, and indeed it is. Listing 9-23 doubles bits in 47.07 us per byte doubled, more than 34% faster than the 63.36 us of Listing 7-14. (Note that the ratio of the execution times is almost exactly 3-to-4...which is the ratio of the code sizes of the two approaches. Keep your code short!)

Mind you, the **sar** approach of Listing 9-23 is still much slower than the look-up approach of Listing 7-15. What's more, the code in Listing 9-23 is both slower and larger than the **xlat**- based nibble look-up approach shown in Listing 7-18, so **sar** really isn't a preferred technique for doubling bits. The point to our discussion of bit-doubling with **sar** is actually this: all sorts of interesting possibilities open up once you start to view instructions in terms of what they do, rather than what they were designed to do.

## ASCII AND DECIMAL ADJUST

Now we come to the ASCII and decimal-adjust instructions: **daa**, **das**, **aaa**, **aas**, **aam**, and **aad**. To be honest, I'm covering these instructions only because many people have asked me what they are used for. In truth, they aren't useful very often, and there aren't any particularly nifty or non-obvious uses for them that I'm aware of, so I'm not going to cover them at great length, and you shouldn't spend too much time trying to understand them unless they fill a specific need of yours. Still, the ASCII and decimal-adjust instructions do have their purposes, so here goes.

## daa, das, AND PACKED BCD ARITHMETIC

**daa** ("decimal adjust AL after addition") and **das** ("decimal adjust

AL after subtraction") adjust AL to the correct value after addition of two packed BCD (binary coded decimal) operands. Packed BCD is a number-storage format whereby a digit between 0 and 9 is stored in each nibble, so the hex value 1000h interpreted in BCD is 1000 decimal, not 4096 decimal. (Unpacked BCD is similar to packed BCD, save that only one digit rather than two is stored in each byte.)

Naturally, the addition of two BCD values with the **add** instruction doesn't produce the right result. The contents of AL after **add al,bl** is performed with 09h (9 decimal in BCD) in AL and 01h (1 decimal in BCD) in BL is 0Ah, which isn't even a BCD digit. What **daa** does is take the binary result of the addition of a packed BCD byte (two digits) in AL and adjust it to the correct sum. If, in the last example, **daa** had been performed after **add al,bl**, AL would have contained 10h, which is 10 in packed BCD--the correct answer.

**das** performs a similar adjustment after subtraction of packed BCD numbers. The mechanics of **daa** and **das** are a bit complex, and I won't go into them here, since I know of no use for the instructions save to adjust packed BCD results. Yes, I do remember that I told you to look at instructions for what they can do, not what they were designed to do. As far as I know, though, the two are one and the same for **daa** and **das**. I'll tell you what: look up the detailed operation of these instructions, find an unintended use for them, and let me know what it is. I'll be delighted to hear! One possible hint: these instructions are among the very few that pay attention to the Auxiliary Carry flag.

I'm not going to spend any more time on **daa** and **das**, because they're just not used that often. BCD arithmetic is used primarily for working with on values to an exact number of decimal digits. (By contrast, normal binary arithmetic stores values to an exact number of binary digits, which can cause rounding problems with decimal calculations.) Consequently, BCD arithmetic is useful for accounting purposes, but not much else. Moreover, BCD arithmetic is decidedly slow. If you're one of the few who need BCD arithmetic, the BCD-oriented instructions are there, and BCD arithmetic is well-discussed in the literature-- it's been around for decades, and many IBM mainframes use it--so go to it. For the rest of you, don't worry that you're missing out on powerful and mysterious instructions--the BCD instructions are deservedly obscure.

**aam, aad, AND UNPACKED BCD ARITHMETIC**

**aam** and **aad** are BCD instructions of a slightly different flavor and a bit more utility. **aam** ("ASCII adjust AX after multiply") adjusts the result in AL of the multiplication of two single-digit unpacked BCD values to a valid two-digit unpacked BCD value in AX. This is accomplished by dividing AL by 10 and storing the quotient in AH and the remainder in AL. (By contrast, **div** stores the quotient in AL and the remainder in AH.)

**aad** ("ASCII adjust AL before division") converts a two-digit unpacked BCD value in AX into the binary equivalent in AX. This is performed by multiplying AH by 10, adding it to AL, and zeroing AH. The binary result of **aad** can then be divided by a single-digit BCD value to

generate a single-digit BCD result.

By the way, "ASCII adjust" really means unpacked BCD for these instructions, since ASCII digits with the upper nibble zeroed are unpacked BCD digits. **aaa** and **aas**, which we'll discuss shortly, explicitly convert ASCII digits into unpacked BCD, but **aam** and **aad** require that you use **and** to zero the upper nibble of ASCII digits before performing multiplication and division.

**aam** can be used to implement multiplication of arbitrarily long unpacked BCD operands one digit at a time. That is, with **aam** you can multiply decimal numbers just the way we do it with a pencil and paper, multiplying one digit of each product together at a time and carrying the results along. Presumably, **aad** can be used similarly in the division of two BCD operands, although I've never found an example of the use of **aad**.

At any rate, the two instructions do have some small use apart from unpacked BCD arithmetic. They can save a bit of code space if you need to perform exactly the specified division by 10 of **aam** or multiplication by 10 and addition of **aad**, although you must be sure that the result can fit in a single byte. In particular, **aam** has an advantage over **div** in that a **div** by an 8- bit divisor requires a 16-bit dividend in AX, while **aam** uses only an 8-bit dividend in AL. **aam** has another advantage in that unlike **div**, it doesn't require a register to store the divisor.

For example, Listing 9-24 shows code that converts a byte value to a three-digit ASCII string by way of **aam**. Listing 9-25, by contrast, converts a byte value to an ASCII string by using explicit division by 10 via **div**. Listing 9-24 is only 28 bytes long per byte converted, 2 bytes shorter

than Listing 9-25. Listing 9-24 also executes in 54.97 us per conversion, 2.65 us faster than the 57.62 us of Listing 9-25. Normally, an improvement of 2.65 us would have us jumping up and down, but the lengthy execution times of both conversion routines mean that the speed advantage of Listing 9-24 is only about 5%. That's certainly an improvement--but painfully slow nonetheless.

**aam** and **aad** would be more interesting if they provided significantly faster ways than **div** and **mul** to divide and multiply by 10. Unfortunately, that's not the case, as the above results illustrate. **aad** and **aam** must use the 8088's general-purpose multiplication and division capabilities, for they are just about as slow as **mul** and **div**. **aad** is the speedster of the two at 60 cycles per execution, while **aam** executes in 83 cycles.

**NOTES ON mul AND div**

I'd like to take a moment to note some occasionally annoying characteristics of **mul** and **div**. **mul** (and **imul**, but I'll refer only to **mul** from now on for brevity) has a tendency to surprise you by wiping out a register that you'd intuitively think it wouldn't, because the product is stored in twice as many bits as either factor. For example, **mul bl** stores the result in AX, not AL, and **mul cx** stores the result in DX:AX, not AX. While this sounds simple enough, it's easy to forget in the heat of coding.

Similarly, it's easy to forget that **div** requires that the dividend be twice as large as the divisor and quotient. (The following discussion applies

to **idiv** as well; again, I'll refer only to **div** for brevity.) In order to divide one 16-bit value by another, it's essential that the 16-bit dividend be extended to a 32-bit value, as in:

```
        mov     bx,[Divisor]
        mov     ax,[Dividend]
        sub     dx,dx           ;extend Dividend to an unsigned 32-bit value
        div     bx
```

(**cwd** can be used for sign-extension to a 32-bit value.) What's particularly tricky about 32-bit-by-16-bit division is that it leaves the remainder in DX. That means that if you perform multiple 16-bit-by-16-bit divisions in a loop, you must zero DX every time through the loop. For example, the following code to convert a binary number to five ASCII digits wouldn't work properly, because the dividend wouldn't be properly extended to 32 bits after the first division, which would leave the remainder in DL:

```
        mov     ax,[Count]      ;value to convert to ASCII
        sub     dx,dx           ;extend Count to an unsigned 32-bit value
        mov     bx,10           ;divide by 10 to convert to decimal
        mov     si,offset CountEnd-1 ;ASCII count goes here
        mov     cx,5                    ;we want 5 ASCII digits
DivLoop:
        div     bx                      ;divide by 10
        add     dl,'0'          ;convert this digit to ASCII
        mov     [si],dl         ;store the ASCII digit
        dec     si                      ;point to the next most significant digit
        loop    DivLoop
```

On the other hand, the following code would work perfectly well, because it extends the dividend to 32 bits every time through the loop:

```
        mov     ax,[Count]      ;value to convert to ASCII
        mov     bx,10           ;extend Count to an unsigned 32-bit value
        mov     si,offset CountEnd-1 ;ASCII count goes here
```

```
                mov     cx,5                    ;we want 5 ASCII digits
DivLoop:
                sub     dx,dx           ;extend the dividend to an unsigned 32-bit value
                div     bx                      ;divide by 10
                add     dl,'0'          ;convert this digit to ASCII
                mov     [si],dl         ;store the ASCII digit
                dec     si                      ;point to the next most significant digit
                loop    DivLoop
```

All of the above goes for 8-bit-by-8-bit division as well, except that in that case it's the 8-bit dividend in AL that you must extend to a word in AX before each division.

There's another tricky point to **div**: **div** can crash a program by generating a divide-by-zero interrupt (interrupt 0) under certain circumstances. Obviously, this can happen if you divide by zero, but that's not the only way **div** can generate a divide-by-zero interrupt. If a division is attempted for which the quotient doesn't fit into the destination register (AX for 32-bit-by-16-bit divides, AL for 16-bit-by-8-bit divides), a divide-by-zero interrupt occurs. So, for example:

```
mov     ax,0ffffh
mov     dl,1
div     dl
```

results in a divide-by-zero interrupt.

Often, you know exactly what the dividend and divisor will be for a particular division, or at least what range they'll be in, and in those cases you don't have to worry about **div** causing a divide-by-zero interrupt. If you're not sure that the dividend and divisor are safe, however, you must guard against potential problems. One way to do this is by intercepting

interrupt 0 and handling divide-by-zero interrupts. The alternative is to check the dividend and divisor before each division to make sure both that the divisor is non-zero and that the dividend isn't so much larger than the divisor that the result won't fit in 8 or 16 bits, whichever size the division happens to be.

This division-by-zero business is undeniably a nuisance to have to deal with--but it's absolutely necessary if you're going to perform division without knowing that the inputs can safely be used.

**aaa, aas, AND DECIMAL ASCII ARITHMETIC**

Finally, we come to **aaa** and **aas**, which support addition and subtraction of decimal ASCII digits. Actually, **aaa** and **aas** support addition and subtraction of any two unpacked BCD digits, or indeed of any two bytes at all the lower nibbles of which contain digits in the range 0-9.

**aaa** ("ASCII adjust after addition") adjusts AL to the correct decimal (unpacked BCD) result of the addition of two nibbles. Consider this: if you add two digits in the range 0-9, one of three things can happen. The result can be in the range 0-9, in which case no adjustment is needed and no decimal carry has occurred. Alternatively, the result can be in the range 0Ah- 0Fh, in which case the result can be corrected by adding 6 to the result, taking the result modulo 16 (decimal), and setting carry- out. Finally, the result can be in the range 10h-12h, in which case the result can be corrected in exactly the same way as for results in the range 0Ah-0Fh.

**aaa** handles all three cases with a single 1-byte instruction. **aaa**

assumes that an **add** or **adc** instruction has just executed, with the Auxiliary Carry flag set appropriately. If the Auxiliary Carry flag is set (indicating a result in the range 10h-12h) or if the lower nibble of AL is in the range 0Ah-0Fh, then 6 is added to AL, the Auxiliary Carry and Carry flags are set to 1, and AH is incremented. Finally, the upper nibble of AL is set to 0 in all cases.

What does all this mean? Obviously, it means that it's easy to add together unpacked BCD numbers. More important, though, is that **aaa** makes it fast (4 cycles per **aaa**) and easy to add together ASCII representations of decimal numbers. That's genuinely useful because it takes a slew of cycles to convert a binary number to an ASCII representation; after all, a division by 10 is required for each digit to be converted. ASCII numbers are necessary for all sorts of data displays for which speed is important, ranging from game scores to instrumentation readouts. **aaa** makes possible the attractive alternative of keeping the numbers in displayable ASCII forms at all times, thereby avoiding the need for any sort of conversion at all.

Listing 9-26 shows the use of **aaa** in adding the value 1-- stored as the ASCII decimal string "00001"--to an ASCII decimal count. Granted, it takes much longer to perform the ASCII decimal increment shown in Listing 9-26 than it does to execute an **inc** instruction--more than 100 times as long, in fact, at 93.00 us per ASCII decimal increment versus a maximum of 0.809 us per **inc**. However, Listing 9-26 maintains the count in instantly-displayable ASCII form, and for frequently-displayed but rarely- changed

numbers, a ready-to-display format can more than compensate for lengthier calculations.

If you do use **aaa**, remember that you have not one but two ways to use the carry-out that indicates that a decimal digit has counted from 9 back around to 0. The Carry flag is set on carry- out; that's what we use as the carry-out status in Listing 9-26. In addition, though, AH is incremented by **aaa** whenever decimal carry-out occurs. It's certainly possible to get some extra mileage by putting the next-most-significant digit in AH before performing **aaa** so that the carry-out is automatically carried. It's also conceivable that you could use **aaa** specifically to increment AH depending on either the value in AL or on the setting of the Auxiliary Carry flag, although I've never seen such an application. Since the Auxiliary Carry flag isn't testable by any conditional jump (or indeed by any instructions other than **daa**, **das**, **aaa**, and **aas**), **aaa** is perhaps the best hope for getting extra utility from that obscure flag.

**aas** ("ASCII adjust after subtraction") is well and truly the mirror image of **aaa**. **aas** is designed to be used after a **sub** or **sbb**, subtracting 6 from the result, decrementing AH, and setting the Carry flag if the result in AL is not in the range 0-9, and zeroing the high nibble of AL in any case. You'll find that wherever **aaa** is useful, so too will be **aas**.

## MNEMONICS THAT COVER MULTIPLE INSTRUCTIONS

As we've seen several times in this chapter and the last, 8088 assembler often uses a single mnemonic, such as **mov**, to name two or more

instructions that perform the same operations but are quite different in size and execution speed. When the assembler encounters such a mnemonic in assembler source code, it automatically chooses the most efficient instruction that fills the bill.

For example, earlier in this chapter we learned that there's a special 16-bit register-only version of **inc** that's shorter and faster than the standard mod-reg-rm version of **inc**. Whenever you use a 16-bit register **inc** in source code--for example, **inc ax**-- the assembler uses the more efficient 16-bit register-only **inc**; otherwise, the assembler uses the standard version.

Naturally, you'd prefer to use the most efficient version of a given mnemonic whenever possible. The only way to do that is to know the various instructions described by each mnemonic and to strive to use the forms of the mnemonic that assemble to the most efficient instruction. For instance, consider the choice between **inc ax** and **inc al**. Without inside knowledge, there's nothing to choose from between these two assembler lines. In fact, there might be a temptation to choose the 8-bit form on the premise that an 8-bit operation can't possibly be slower than a 16-bit one. Actually, of course, it can...but you'll only know that the 16-bit **inc** is the one to pick if you're aware of the two instructions **inc** describes.

This section is a summary of mnemonics that cover multiple instructions, many of which we've covered in detail elsewhere in this book. The mnemonics that describe multiple instructions are:

- **inc**, which has a mod-reg-rm version and a 16-bit register- only

version, as described earlier in this chapter (the same applies to **dec**).

- **xchg**, which has a mod-reg-rm version and a 16-bit exchange- with-AX-only version, as described in Chapter 8.
- **add**, which has two mod-reg-rm versions (one for adding a register and a memory variable or a second register together, and another for adding immediate data to a register or memory variable) and an accumulator-specific immediate-addressing version, as described in Chapter 8 (the same applies to **adc**, **and**, **cmp**, **or**, **sbb**, **sub**, **test** and **xor**, also as described in Chapter 8).
- **mov**, which requires further explanation.

**mov** covers several instructions, and it's worth understanding each one. The basic form of **mov** is a mod-reg-rm form that copies one register or memory variable to another register or memory variable. (Memory-to-memory moves are not permitted, however.) There's also a mod-reg-rm form of **mov** that allows the copying of a segment register to a general-purpose register or a memory variable, and vice-versa. Last among the mod-reg-rm versions of **mov**, there's a form of **mov** that supports the setting of a register or a memory variable to an immediate value.

There are two more versions of **mov**, both of which are non- mod-reg-rm forms of the instruction. There's an accumulator- specific version that allows the transfer of values between direct-addressed memory variables and the accumulator (AL or AX) faster and in fewer bytes than the mod-reg-rm instruction, as discussed in Chapter 8. There's also a register-

specific form of **mov**, as we discussed in Chapter 7; I'd like to discuss that version of **mov** further, for it's an important instruction indeed.

Every mod-reg-rm instruction requires at least 2 bytes, one for the instruction opcode and one for the mod-reg-rm byte. Consequently, the mod-reg-rm version of **mov mem/reg,immed8** is 3 bytes long, since the immediate value takes another byte. However, there's a register-specific immediate-addressing form of **mov** that doesn't have a mod-reg-rm byte. Instead, the register selection is built right into the opcode, so only 1 byte is needed to both describe the instruction and select the destination. The result: the register-specific immediate- addressing form of **mov** allows **mov reg,immed8** to assemble to just 2 bytes, and **mov reg,immed16** to assemble to just 3 bytes. The presence of the register-specific immediate-addressing version of **mov** makes loading immediate values into registers quite reasonable in terms of code size and performance. For example, **mov al,0** assembles to a 2-byte instruction, exactly the same length as **sub al,al**. Granted, **sub al,al** is 1 cycle faster than **mov al,0**, and **sub ax,ax** is both 1 cycle faster and 1 byte shorter than **mov ax,0**, but nonetheless the upshot is that registers can be loaded with immediate values fairly efficiently.

Be aware, however, that the same is not generally true of **add**, **sub**, or any of the logical or arithmetic instructions--the mod-reg-rm immediate-addressing forms of these instructions take a minimum of 3 bytes. As mentioned above, though, the accumulator-specific immediate-addressing forms of these instructions are fast and compact at 2 or 3 bytes

in length.

While there is a special form of **mov** for loading registers with immediate data, there is no such form for loading memory variables. The shortest possible instruction for loading memory with an immediate value is 3 bytes long, and such instructions can range all the way up to 6 bytes in length. In fact, thanks to the 8088's accumulator- and register-specific **mov** instructions:

```
mov     al,0
mov     [MemVar],al
```

is not only the same length as:

```
mov     [MemVar],0
```

but is also 2 cycles faster!

Learn well those special cases where a single mnemonic covers multiple instructions--and use them! They're one of the secrets of good 8088 assembler code.

## ON TO THE STRING INSTRUCTIONS

We've cut a wide swath through the 8088's instruction set in this chapter, but we have yet to touch on one important set of instructions--the string instructions. These instructions, which are perhaps the most important instructions the 8088 has to offer when it comes to high-

performance programming, are coming up next.  Stay tuned.