Chapter 13: Not-Branching

Now we know why we don't want to branch, but we haven't a clue as to how to manage that trick. After all, decisions still have to be made, loops still have to be iterated through, and so on. Branching is the way we've always performed those tasks, and it's certainly not obvious what the alternatives are, or, for that matter, that alternatives even exist.

While alternatives to branching do indeed exist, they are anything but obvious. Programming without branches--not- branching, in Zen-speak--is without question one of the stranger arts you must master in your growth as a Zen programmer.

Strange--but most rewarding. So let's get to it!

THINK FUNCTIONALLY

The key to not-branching lies in understanding each programming task strictly in terms of what that task needs to do, not in terms of how the task will ultimately be implemented. Put another way, you should not consider how you might implement a task, even in a general way, until you have a clear picture of exactly what results the implementation must produce.

Once you've separated the objective from the implementation, you're free to bring all the capabilities of the 8088--in their limitless combinations and permutations--to bear in designing the implementation,

rather than the limited subset of programming techniques you've grown accustomed to using. This is one of the areas in which assembler programmers have a vast advantage over compilers, which can use only the small and inflexible set of techniques their designers built in. Compilers operate by translating human-oriented languages to machine language along a few fixed paths; there's no way such a rigid code-generation mechanism can properly address the boundless possibilities of the 8088.

Of course, separating the objective and the implementation is more easily said than done, especially given an instruction set in which almost every instruction seems to have been designed for a specific purpose. For example, it's hard not to think of the **loop** instruction when you need to exclusive-or together all the bytes in a block of memory 64 bytes long, and do so as quickly as possible. (Such a cumulative exclusive-or might be used as a check against corrupted data in a block of data about to be transmitted or stored. The speed at which the cumulative exclusive-or could be generated might well determine the maximum error-checked transfer rate supported by the program.)

In this case, as in many others, the objective--a fast cumulative exclusive-or--and the implementation--64 loops by way of the **loop** instruction, with each loop exclusive-oring 1 byte into the cumulative result--are inseparable to the experienced non-Zen programmer.

Why? Consider the solution shown in Listing 13-1. Listing 13-1 is obviously well-matched to the task of generating the cumulative exclusive-or for a block of 64 bytes. In fact, it's so well-matched that few

programmers would even contemplate alternatives. The code in Listing 13-1 works, it's easy to write, and it runs in just 503 us. Surely that's just about as fast as the 8088 can manage to perform this task--after all, the loop involves just three instructions: one **lodsb** (string instructions are the fastest around), one register-register **xor** (register-register instructions are short and fast), and one **loop** (the 8088's special, fast looping instruction). Who would ever think that performance could be nearly doubled by literally duplicating the code inside the loop 64 times and executing that code sequentially--thereby eliminating branching entirely?

Only a Zen programmer would even consider the possibility, for not-branching simply has no counterpart in non-Zen programming. Not-branching just plain feels wrong at first to any programmer raised on high-level languages. Not-branching goes against the grain and intent of both the 8088 instruction set and virtually all computer-science teachings and high-level languages. That's only to be expected; language designers and computer-science teachers are concerned with the form of programs, for they're most interested in making programming more amenable to people--that is, matching implementations to the way people think.

By contrast, Zen programmers are concerned with the functionality of programs. Zen programmers focus on performance and/or program size, and are most interested in matching implementations to the way computers think. The desired application is paramount, but the true Zen comes in producing the necessary result (the functionality) in the best possible way given the computer's resources.

Zen programmers understand that the objective in generating the cumulative exclusive-or of 64 bytes actually has nothing whatsoever to do with looping. The objective is simply to exclusive-or together the 64 bytes in whatever way the PC can most rapidly accomplish the task, and looping is just one of many possible means to that end. Most programmers have seen and solved similar problems so many times, however, that they instinctively--almost unconsciously--select the **loop** instruction from their bag of tricks the moment they see the problem. To these programmers, repetitive processing and **loop** are synonymous.

Zen programmers have a bigger bag of tricks, however, and a more flexible view of the world. Listing 13-2 shows a Zen solution to the array-sum problem. Listing 13-2 performs no branches at all, thanks to the use of in-line code, which we'll discuss in detail later in this chapter.

Functionally, there's not much difference between Listings 13-1 and 13-2. Both listings leave the same cumulative result in AH, leave the same value in SI, and even leave the flags set to the same values. Listing 13-1 leaves CX set to zero, while Listing 13-2 doesn't touch CX, but that's really a point in the favor of Listing 13-2, and could in any case be remedied simply by placing a **sub cx,cx** at the start of Listing 13-2 if necessary.

No, there's not much to choose from between the two listings...until you see them in action. Listing 13-2 calculates the 64-byte cumulative exclusive-or value in just 275 us--more than 82% faster than Listing 13-1. A 5% increase might not be worth worrying about, but we're talking about nearly doubling the performance of a well-coded three-

instruction loop! Clearly, there's something to this business of Zen programming.

You may object that Listing 13-2 is many bytes longer than Listing 13-1, and indeed it is: 184 bytes, to be exact. If you need speed, though, a couple of hundred bytes is a small price to pay for nearly doubling performance--certainly preferable to requiring a more powerful (and expensive) processor, such as an 80286. You may also object that Listing 13-2 can only handle blocks that are exactly 64 bytes in length, while the loop in Listing 13-1 can be made to handle blocks of any size simply by loading CX with different values. That, too, is true...but you're missing the point.

Listing 13-2 is constructed to meet a specific goal as well as possible on the PC. If the goal was different, then Listing 13-2 would be different. If blocks of different sizes were required, then we would modify our approach accordingly, possibly by jumping into the series of exclusive-or operations at the appropriate place. If space was tight, perhaps we would use partial in-line code (which we'll discuss later in this chapter), combining the space-saving qualities of loops with the speed of in-line code. If space was at a premium and performance was not an issue, we might well decide that **loop** was the best solution after all. The point is that the Zen programmer has a wide range of approaches to choose from, and in most cases at least one of those choices will handily outperform any standard, one-size- fits-all solution.

In the context of not-branching (which is after all how we got into

all this), Zen programming means replicating the functionality of branches without branching. That's certainly not a goal we'd want to achieve all the time--in many cases branches really are the best (or only) choice--but you'll be surprised at how often it's possible to find good substitutes for branches in time-critical code.

For all their reputation as number-crunching machines, computers typically spend most of their time moving data, scanning data, and branching. In the Chapters 10 and 11 we learned how to minimize the time spent moving and scanning data. Now we're going to attack the other part of the performance equation by learning how to minimize branching.

rep: LOOPING WITHOUT BRANCHING

It's a popular misconception that **loop** is the 8088's fastest instruction for looping. Not so. In truth, it's **rep** that supports far and away the most powerful looping possible on the 8088. In Chapters 10 and 11 we saw again and again that repeated string instructions perform repetitive tasks much, much faster than normal loops do. Not only do repeated string instructions not empty the prefetch queue on every repetition as **loop** and other branching instructions do, but they actually eliminate the prefetch queue cycle-eater altogether, since no instruction fetching at all is required while a string instruction repeats.

As we saw in Chapter 9, shifts and rotates by CL also eliminate the prefetch queue cycle-eater, although those instructions don't pack quite the punch that repeated string instructions do, both because they perform

relatively specialized tasks and because there's not much point to repeating a shift or rotate more than 16 times.

We've already discussed repeated string instructions and repeated shifts and rotates in plenty of detail, so I'm not going to spend much more time on them here. However, I would like to offer one hint about using shifts and rotates by CL. As we found in Chapter 9, repeated shifts and rotates are generally faster than individual shifts and rotates when a shift or rotate of 3 or more bits is required. Repeated shifts and rotates are also much faster than shifting 1 bit at a time in a loop; the sequence:

```
BitShiftLoop:
                shr     ax,1
                loop    BitShiftLoop
```

is far inferior to **shr ax,cl**.

Nonetheless, repeated shifts and rotates still aren't fast-- instead, you might think of them as less slow than the alternatives. It's easy to think that shifts and rotates by CL are so fast that they can be used with impunity, since they avoid looping and prefetching, but that's just not true. A repeated shift or rotate takes 8 cycles just to start, and then takes 4 cycles per bit shifted. Even a 4-bit shift by CL takes 24 cycles, which is not insignificant, and a 16-bit shift by CL takes a full 72 cycles. Use shifts and rotates by CL sparingly, and keep them out of loops whenever you can. Look-up tables, our next topic, are often a faster alternative to multi-bit shifts and rotates.

LOOK-UP TABLES: CALCULATING WITHOUT BRANCHING

Like the use of repeated string instructions, the use of look-up tables is a familiar technique that can help avoid branching. Whenever you're using branching code to perform a calculation, see if you can't use a look-up table instead; tight as your branching code may be, look-up tables are usually faster still. Listings 11-26 and 11-27 pit a five-instruction sequence that branches no more than once against an equivalent table look-up; you can't get branching code that's much tighter than that, and yet the table look-up is much faster.

In short, if you have a calculation to make--even a simple one--see if it isn't faster to precalculate the answer at assembly time and just look it up at run time.

TAKE THE BRANCH LESS TRAVELLED BY One of the best ways to avoid branching is to arrange your code so that conditional jumps rarely jump. Usually you can guess which way a given conditional test will most often go, and if that's the case, you can save a good deal of branching simply by arranging your code so that the conditional jump will fall through--that is, not branch--in the more common case. Sometimes the choice is made on the basis of which case is most time- critical rather than which is most common, but the principle remains the same.

Why is it that falling through conditional jumps is desirable? Simple: none of the horrendous speed loss associated with branching

applies to conditional jumps that fall through, because conditional jumps don't branch when they fall through.

Let's look at the statistics. It always takes a conditional jump at least 16 cycles to branch, and the total cost in cycles is usually somewhat greater because the prefetch queue is emptied. On the other hand, it takes a conditional jump a maximum of just 8 cycles not to jump, that being the case if the prefetch queue is empty and both bytes of the instruction must be fetched before they can be executed. The official execution time of a conditional jump that doesn't branch is just 4 cycles, so it is particularly fast to fall through a conditional jump if both bytes of the instruction are waiting in the prefetch queue when it comes time to execute them.

In other words, falling through a conditional jump can be anywhere from 100% to 700% faster than branching, depending on the exact state and behavior of the prefetch queue. As you might imagine, it's worth going out of your way to reap cycle savings of that magnitude...and that's why you should arrange your conditional jumps so that they fall through as often as possible.

For example, you'll recall that in Chapter 11--in Listing 11-20, to be precise--we tested several characters for inclusion in a small set via repeated **cmp**/**jz** instruction pairs. We arranged the conditional jumps so that a jump occurred only when a match was made, meaning that at most one branch was performed during any given inclusion test. Put another way, we branched out of the main stream of the subroutine on the less common condition.

You may not have thought much of it at the time, but the arrangement of branches in Listing 11-20 was no accident. Tests for four potential matches are involved when testing for inclusion in a set of four characters, and no more than one of those matches can occur during any given test. Given an even distribution of match characters, matching is clearly less common than not matching. If we jumped whenever we didn't get a match (the more common condition), we'd end up branching as many as three times during a single test, with significantly worse performance the likely result.

Listing 13-3 shows Listing 11-20 modified to branch on non-matches rather than matches. The original branch-on-match version ran in 119 us, and, as predicted, that's faster than Listing 13-3, which runs in 133 us. That's not the two- or three-times performance improvement we've grown accustomed to seeing (my, how jaded we've become!), but it's significant nonetheless, especially since we're talking about a very small number of conditional jumps. We'd see a more dramatic difference if we were dealing with a long series of tests.

Another relevant point is that the worst-case performance of Listing 13-3 is much worse than that of Listing 11-20. Listing 13-3 actually has a shorter best-case time than Listing 11-20, because no branches at all are performed when the test character is 'A'. On the other hand, Listing 13-3 performs three branches when the test character is '!' or is not in the set, and that's two branches more than Listing 11-20 ever performs. When you're trying to make sure that code always responds within a certain time,

worst-case performance can matter more than average performance.

Then, too, if the characters tested are often not in the set, as may well be the case with such a small set, the branching-out approach of Listing 11-20 will far outperform the branch-branch-branch approach of Listing 13-3. When Listing 11- 20 is modified so that none of the five test characters is in the set, its overall execution time scarcely changes, rising by just 8 us, to 127 us. When Listing 13-3 is modified similarly, however, its overall execution time rises by a considerably greater amount--26 us--to 159 us. This neatly illustrates the potential worst-case problem of repeated branching that we just discussed. There are two lessons here. The first and obvious lesson is that you should arrange your conditional jumps so that they fall through as often as possible. The second lesson is that you must understand the conditions under which your code will operate before you can truly optimize it.

For instance, there's no way you can evaluate the relative merits of the versions of **CheckTestSetInclusion** in Listings 11-20 and 13-3 until you know the mix of characters that will be tested. There's no such beast as an absolute measure of code speed, only code speed in context. You've heard that before as it relates to instruction mix and the prefetch queue, but here we're dealing with a different aspect of performance. What I mean now is that you must understand the typical and worst-case conditions under which a block of code will run before you can get a handle on its performance and consider possible alternatives.

Your ability to understand and respond to the circumstances

under which your assembler code will run gives you a big leg up on high-level language compilers. There's no way for a compiler to know the typical and/or worst-case conditions under which code will run, let alone which of those conditions is more important in your application.

For instance, suppose that we have one loop which repeats 10 times on average and another loop which repeats 10000 times on average, with both loops executed a variable (not constant) number of times. A C compiler couldn't know that cycles saved in the second loop would have a 1000-times-greater payoff than cycles saved in the first loop, so it would have to approach both loops in the same way, generating the same sort of code in both cases. What this means is that compiled code is designed for reasonable performance under all conditions...hardly the ticket for greatness.

PUT THE LOAD ON THE UNIMPORTANT CASE

When arranging branching code to branch on the less critical case, don't be afraid to heap the cycles on that case if that will help the more critical case.

For example, suppose that you need to test whether CX is zero at the start of a long subroutine and return if CX is in fact zero. You'd normally do that with something like:

```
LongSubroutine  proc    near
                jcxz    LongSubroutineEnd
                  :
; *** Body of subroutine ***
                  :
LongSubroutineEnd:
```

```
                    ret
LongSubroutine      endp
```

Now, however, assume that the body of the subroutine is more than 127 bytes long. In that case, the 1-byte displacement of **jcxz** can't reach **LongSubroutineEnd**, so the last bit of code won't work.

Well, then, the obvious alternative is:

```
LongSubroutine      proc        near
                    and         cx,cx
                    jnz         DoLongSubroutine
                    jmp         LongSubroutineEnd
DoLongSubroutine:
                     :
; *** Body of subroutine ***
                     :
LongSubroutineEnd:
                    ret
LongSubroutine      endp
```

There's a problem here, though. Every time CX isn't zero we end up branching, and that's surely wrong. The case where CX is zero is most likely rare, and is probably of no real interest to us anyway, since it's a do-nothing case for the subroutine. (At any rate, for the purposes of this example we'll assume that the CX equal to 0 case is rare and uninteresting.) What's more, whether the CX equal to 0 case is rare or not, the body of the subroutine is skipped when CX is 0, so that case is bound to be much faster than the other cases. That means that the CX equal to zero case is not only unimportant, but also doesn't affect the worst-case performance of the subroutine. Yet here we are, adding an extra branch to every single invocation of this subroutine simply to protect against the quick and unimportant case of CX equal to zero.

The tail is wagging the dog.

Instead, let's heap the branches on the CX equal to zero case, sparing the other, more important cases as much as possible. One solution is:

```
LongSubroutineExit      proc    near
                        ret
LongSubroutineExit      endp
;
LongSubroutine  proc    near
                jcxz    LongSubroutineExit
                 :
; *** Body of subroutine ***
                 :
                ret
LongSubroutine  endp
```

This restores the code to its original, saner state, where the shortest possible time--6 cycles for a single **jcxz** that falls through--is used to guard against the case of CX equal to zero.

If you prefer that your subroutines be exited only from the end, as is for example necessary when a stack frame must be deallocated, there's another solution:

```
LongSubroutineExit      proc    near
                        jmp     LongSubroutineEnd
LongSubroutineExit      endp
;
LongSubroutine  proc    near
                jcxz    LongSubroutineExit
                 :
; *** Body of subroutine ***
                 :
LongSubroutineEnd:
                ret
LongSubroutine  endp
```

Now we've <u>really</u> put the load on the CX equal to zero case, for two branches

must be performed in that case. So what? As far as we're concerned, the CX equal to zero case can take as long as it pleases, so long as it doesn't slow down the real work of the subroutine, which is done when CX isn't equal to zero.

YES, VIRGINIA, THERE IS A FASTER 32-BIT NEGATE!

In Chapter 9 we came across an extremely fast and compact way to negate 32-bit values, as follows:

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

This very short sequence involves two register-only negations, one constant-from-register subtraction--and no branches. At the time, I told you that, fast as that code was, at some later point we'd run across a still faster way to negate a 32-bit value.

That time has come. Incredibly, we're going to speed up 32- bit negates by using a branching instruction. Yes, I know that I've been telling you to avoid branching like the plague, but there's a trick here: we're not really going to branch. The branching instruction we're going to use is a conditional jump, and we're going to fall through the jump almost every time.

There's a bit of history to this trick, and it's worth reviewing for the lesson about the Zen of assembler it contains. The story goes as follows:

Having worked out to my satisfaction how the above 32-bit

negation worked, I (somewhat egotistically, I admit) asked Dan Illowsky if he knew how to negate a 32-bit value in three instructions.

Well, it took him a while, but he did come up with a working three-instruction solution. Interestingly enough, it wasn't the solution I had found. Instead, he derived the second solution I mentioned in Chapter 9:

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

This solution is equivalent to the first solution in functionality, length, and cycle count.

That's not the end of the tale, however. Taken aback because Dan had come up with a different and equally good solution (demonstrating that my solution wasn't so profound after all), I commented that while he had managed to match my solution, he surely could never surpass it.

Ha!

If there's one word that should set any Zen programmer off like a rocket, it's "never." The 8088 instruction set is so rich and varied that there are dozens of ways to do just about anything. For any but the simplest task several of those approaches--and not necessarily the obvious ones--are bound to be good. Whenever you think that you've found the best possible solution for anything more complex than incrementing a register, you're most likely in for a humbling experience.

At any rate, "never" certainly set Dan off. He got a thoughtful

look on his face, walked off, and came back five minutes later with a faster implementation. Here it is:

```
                        not     dx
                        neg     ax
                        jnc     Negate32BitsCarry
Negate32BitsDone:
                         :
Negate32BitsIncDX:
                        inc     dx
                        jmp     short Negate32BitsDone
```

where the code at **Negate32BitsCarry** is somewhere--anywhere-- within a 1-byte displacement (+127 to -128 bytes) of the byte after the **jnc** instruction.

It may not look like working 32-bit negation code, but working code it is, believe me. Brilliant working code.

HOW 32-BIT NEGATION WORKS

In order to understand the brilliance of Dan's code, we first need to get a firm grasp on the mechanics of 32-bit negation. The basic principle of two's complement negation is that the value to be negated is first notted (that is, all its bits are flipped, from 1 to 0 or 0 to 1), and then incremented. For a 32-bit value stored in DX:AX, negation would ideally follow one of the two sequences shown in Figure 13-1, with all operations performed 32 bits at a time.

Unfortunately, the 8088 can only handle data 16 bits at a time, so we must perform negation with a series of 16-bit operations like:

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

as shown in Figure 13-2. The purpose of the first operation, notting DX with the **not** instruction, is obvious enough: flipping all the bits in the high word of the value. The purpose of the second operation, negating AX, is equally obvious: negating the low word of the value with the **neg** instruction, which both nots AX and increments it all at once.

After two instructions, we've successfully notted the entire 32-bit value in DX:AX, and we've incremented AX as well. All that remains to be done is to complete the full 32-bit increment by incrementing DX if necessary.

When does DX need to be incremented? In one case only--when AX is originally 0, is notted to 0FFFFh, and is incremented back to 0, with a carry out from bit 15 of AX indicating that AX has turned over to 0 and so the notted value in DX must be incremented as well, as shown in Figure 13-3. In all other cases, incrementing the 32-bit notted value in DX:AX doesn't alter DX at all, since incrementing AX doesn't cause a carry out of bit 15 unless AX is 0FFFFh.

However, due to the way that **neg** sets the Carry flag (as if subtraction from zero had occurred), the Carry flag is set by **neg** in all cases except the one case in which DX needs to be incremented. Consequently, after **neg ax** we subtract -1 from DX with borrow, with the 1 value of the

Carry flag normally offsetting the -1, resulting in a subtraction of 0 from DX. In other words, DX remains unchanged when **neg ax** sets the Carry flag to 1, which is to say in all cases except when AX is originally zero. That's just what we want; in all those cases the 32-bit negation was actually complete after the first two instructions, since the increment of the notted 32-bit value doesn't affect DX, as shown in Figure 13-4.

In the case where AX is originally 0, on the other hand, **neg ax** doesn't set the Carry flag. This is the one case in which DX must be incremented. In this one case only, **sbb dx,-1** succeeds in subtracting -1 from DX, since the Carry flag is 0. Again, that's what we want; in this one case DX is affected when the 32- bit value is incremented, and so incrementing DX completes the 32-bit negation, as shown in Figure 13-5.

HOW FAST 32-BIT NEGATION WORKS

Now that we understand what our code has to do, we're in a position to think about optimizations. We'll do just what Dan did--look at negation from a functional perspective, understanding exactly what needs to be done and tailoring our code to do precisely that and nothing more.

The breakthrough in Dan's thinking was the realization that DX only needs to be incremented when AX originally was 0, which normally happens only once in a blue moon (once out of every 64 K evenly-distributed values, to be exact). For all other original values of AX, the bits in DX simply flip in the process of 32-bit negation, and nothing more needs to be done to DX after the initial **not**. As we found above, the 32-bit negation is actually

complete after the first two instructions for 64 K-1 out of every 64 K possible values to be negated, with the final **sbb** almost always leaving DX unchanged.

Improving the code is easy once we've recognized that the first two instructions usually complete the 32-bit negation. The only question is how to minimize the overhead taken to check for the rare case in which DX needs to be incremented. A once-in-64 K-times case is more than rare enough to absorb a few extra cycles, so we'll branch out to increment DX in the case where it needs to be adjusted. The payoff for branching in that one case is that in all other cases a 3-byte, 4-cycle **sbb** instruction is replaced by a 2-byte, 4-cycle fall-through of **jnc**. In tight code, the 1-byte difference will usually translate into 4 cycles, thanks to the prefetch queue cycle-eater.

Essentially, **jnc** is a faster way of doing nothing in the 64 K-1 cases where DX:AX already contains the negated value than **sbb dx,-1** is. Granted, **jnc** is also a slower way of incrementing DX in the one case where that's necessary, but that's so infrequent that we can readily trade those extra cycles for the cycles we save on the other cases.

Let's try out the two 32-bit negates to see how they compare in actual use. Listing 13-4, which uses the original nonbranching 32-bit negation code, runs in 2264 us. Listing 13-5, which uses the branch-on-zero-AX approach to 32-bit negation, runs in 2193 us. A small improvement, to be sure--but it is nonetheless an improvement, and since the test code's 100:1 ratio of zero to non-zero values is much less than the real world's ratio of 64 K-1:1 (assuming evenly distributed values), the superiority of the

branch-on-zero-AX approach is somewhat greater than this test indicates.

By itself, speeding the negation of 32-bit values by a few cycles isn't particularly noteworthy. On the other hand, you must surely realize that if it was possible to speed up even the three-instruction, non-branching sequence that we started off with, then it must be possible to speed up just about any code, and that perception is important indeed.

Code for almost any task can be implemented in many different ways, and can in the process usually be made faster than it currently is. It's not always worth the cost in programming time and/or bytes to speed up code--you must pick your spots carefully, concentrating on loops and other time- critical code--but it can almost always be done. The key to improved performance lies in understanding exactly what the task at hand requires and understanding the context in which the code performs, and then matching that understanding to the resources of the PC.

My own experience is that no matter how many times I study a time-critical sequence of, say, 20-100 instructions, I can always save at least a few more cycles--and sometimes many more--by viewing the code differently and reworking it to match the capabilities of the 8088 more closely. That's why way back in Chapter 2 I said that "optimize" was not a word to be used lightly. When programming in assembler for the PC, only fools and geniuses consider their code optimized. As for the rest of us...well, we'll just have to keep working on our time-critical code, trying new approaches and timing the results, with the attitude that our code is good and getting better.

And have we finally found the fastest possible code for 32- bit negation, never to be topped? Lord knows I don't expect to come across anything faster in the near future. But never?

Don't bet on it.

ARRANGE YOUR CODE TO ELIMINATE BRANCHES

There are many, many ways to arrange your code to eliminate branches. I'm going to discuss a few here, but don't consider this to be anything like an exhaustive list. Whenever you use branching instructions where performance matters, take it as a challenge to arrange those instructions for maximum performance and minimum code size.

PRELOADING THE LESS COMMON CASE

One of my favorite ways to eliminate jumps comes up when a register must be set to one of two values based on a test condition. For example, suppose that we want to set AL to 0 if DL is less than or equal to 10, and set AL to 1 if DL is greater than 10.

The obvious solution is:

```
                cmp     dl,10                   ;is DL greater than 10?
                ja      DLGreaterThan10  ;yes, so set AL to 1
                sub     al,al                   ;DL is less than or equal to 10
                jmp     short DLCheckDone
DLGreaterThan10:
                mov     al,1                                    ;DL is greater than 10
DLCheckDone:
```

Here we either branch or don't branch to reach the code that sets AL to the

appropriate value; after setting AL, we rejoin the main flow of the code, branching if necessary. Whether DL is greater than 10 or not, a branch is always performed.

Now let's try this out:

```
                sub     al,al           ;assume DL will not be greater than 10
                cmp     dl,10           ;is DL greater than 10?
                jbe     DLCheckDone     ;no, so AL is already correct
                mov     al,1                    ;DL is greater than 10
DLCheckDone:
```

Here we've loaded AL with one of the two possible results before the test. In one of the two possible cases, we've guessed right and AL is already correct, so a single branch ends the test-and- set code. In the other possible case, we've guessed wrong, so the conditional jump falls through and AL is set properly. (By the way, **inc ax** would be faster than and logically equivalent to **mov al,1** in the above code. Right now, though, we're focusing on a different sort of optimization, and I've opted for clarity rather than maximum speed; I also want you to see that the preload approach is inherently faster, whether or not tricks like **inc ax** are used.)

I'll admit that it's more than a little peculiar to go out of our way to set AL twice in some cases; the previous example set AL just once per test-and-set, and that would logically seem to be the faster approach. While we sometimes set AL an extra time with the preload approach, however, we also avoid a good bit of branching, and that's more than enough to compensate for the extra times AL is set.

Consider this. If DL is less than or equal to 10, then the first example (the "normal" test-and-branch code) performs a **cmp dl,10** (4 cycles/2 bytes), a **ja DLGreaterThan10** that falls through (4 cycles/2 bytes), a **sub al,al** (3 cycles/2 bytes), and a **jmp short DLCheckDone** (15 cycles/2 bytes). The grand total: 26 cycles, 8 instruction bytes and one branch, as shown in Figure 13-6a.

On the other hand, the preload code of the second example handles the same case with a **sub al,al** (3 cycles/2 bytes), a **cmp dl,10** (4 cycles/2 bytes), and a **jbe DLCheckDone** that branches (16 cycles/2 bytes). The total: 23 cycles, 6 instruction bytes and one branch, as shown in Figure 13-7a. That's not much faster than the normal approach, but it is faster.

Now let's look at the case where DL is greater than 10. Here the test-and-branch code of the first example performs a **cmp dl,10** (4 cycles/2 bytes), a **ja DLGreaterThan10** that branches (16 cycles/2 bytes), and a **mov al,1** (4 cycles/2 bytes), for a total of 24 cycles, 6 instruction bytes and one branch, as shown in Figure 13-6b.

The preload code of the second example handles the same DL greater than 10 case with a **sub al,al** (3 cycles/2 bytes), a **cmp dl,10** (4 cycles/2 bytes), a **jbe DLCheckDone** that doesn't branch (4 cycles/2 bytes), and a **mov al,1** (4 cycles/2 bytes). The total: 8 instruction bytes--2 bytes more than the test-and- branch code--but just 15 cycles...and no branches, as shown in Figure 13-7b. The lack of a prefetch queue-flushing branch should more than compensate for the two additional instruction bytes that must be fetched.

In other words, the preload code is either 3 or 9 cycles faster than the more familiar test-and-branch code, is 2 bytes shorter overall, and sometimes branches less while never branching more. That's a clean sweep for the preload code--all because always performing one extra register load made it possible to do away with a branch.

Let's run the two approaches through the Zen timer. Listing 13-6, which times the test-and-branch code when DL is 10 (causing AL to be set to 0), runs in 10.06 us per test-and-branch. By contrast, Listing 13-7, which times the preload code for the same case, runs in just 8.62 us.

That's a healthy advantage for the preload code, but perhaps things will change if we test a case where AL is set to 1, by altering Listings 13-6 and 13-7 to set DL to 11 rather than 10 prior to the tests.

Things do indeed change when DL is set to 11. Listing 13-6 speeds up to 8.62 ms per test, matching the performance of Listing 13-7 when DL was 10. When DL is 11, however, Listing 13- 7 speeds up to 8.15 us, again comfortably outperforming Listing 13-6.

In short, the preload approach is superior in every respect. While it's counterintuitive to think that by loading a register an extra time we can actually speed up code, it does work, and that sort of unorthodox but effective technique is what the Zen of assembler is all about.

A final note on the preload approach: arrange your preload code so that the more common case is not preloaded. Once again this is counterintuitive, since it seems that we're going out of our way to guess wrong about the outcome of the test. Remember, however, that it's much

faster to fall through a conditional jump, and you'll see why preloading the less common value makes sense. It's actually faster to fall through the conditional jump and load a value than it is just to branch at the conditional jump, even if the correct value is already loaded.

The results from the two executions of Listing 13-7 confirm this. The case where the value preloaded into AL is correct actually runs a good bit more slowly than the case where the conditional jump falls through and a new value must be loaded.

Think of your assembler programs not just in terms of their logic but also in terms of how that logic can best be expressed-- in terms of cycles and/or bytes--in the highly irregular language of the 8088. The first example in this section--the "normal" approach--seems at first glance to be the ideal expression of the desired test-and-set sequence in 8088 assembler. However, the poor performance of branching instructions renders the normal approach inferior to the preload approach on the 8088, even though preloading is counter to common sense and most programming experience. In short, the best 8088 code can only be arrived at by thinking in terms of the 8088; superior 8088 solutions often seem to be lunacy in other logic systems.

Thinking in terms of the 8088 can be particularly difficult for those of us used to high-level languages, in which programs are pure abstractions far removed from the ugly details of the processor. When programming in a high-level language, it would seem to be faster to preload the correct value and test than to preload an incorrect value, test, and load

the correct value. In fact, in any high-level language it would seem most efficient to use an **if...then...else** structure to handle a test-and-set case such as the one above.

That's not the way it works on the 8088, though, because not all tests are created equal--tests that branch are much slower than tests that fall through. When you're programming the 8088 in assembler, the maddening and fascinating capabilities of the processor must become part of your logic system, however illogical the paths down which that perspective leads may seem at times to be.

USE THE CARRY FLAG TO REPLACE SOME BRANCHES

Unlike the other flags, the Carry flag can serve as a direct operand to some arithmetic instructions, such as **rcr** and **adc**. This gives the Carry flag a unique property--it can sometimes be used to alter the value in a register conditionally without branching.

For instance, suppose that we want to count the number of negative values in a 1000-word array, maintaining the count in DX. One way to do this is shown in Listing 13-8, which runs in 12.29 ms. In this code, each value is anded with itself. The resulting setting of the Sign flag indicates whether the value is positive or negative. With the help of a conditional jump, the Sign flag setting controls whether DX is incremented or not.

Speedy and compact as it is, Listing 13-8 does involve a conditional jump that branches about half the time...and by now you should

be developing a distinct dislike for branching. By using the Carry flag to eliminate branching entirely, we can speed things up quite a bit.

Listing 13-9 does just that, shifting the sign bit of each tested value into the Carry flag and then adding it--along with zero, since **adc** requires two source operands--to DX, as shown in Figure 13-8. (Note that the constant zero is stored in BX for speed, since **adc dx,bx** is 1 byte shorter and 1 cycle faster than **adc dx,0**.) The result is that DX is incremented only when the sign bit of the value being tested is 1--that is, only when the value being tested is negative, which is exactly what we want.

Listing 13-9 runs in 10.80 ms. That's about 14% faster than Listing 13-8, even though the instruction that increments DX in Listing 13-9 (**adc dx,bx**) is actually 1 byte longer and 1 cycle slower than its counterpart in Listing 13-8 (**inc dx**). The key to the improved performance is, once again, avoiding branching. In this case that's made possible by recognizing that a Carry flag- based operation can accomplish a task that we'd usually perform with a conditional jump. You wouldn't normally think to substitute **shl**/**adc** for **and**/**jns**/**inc**--they certainly don't look the least bit similar--but in this particular context the two instruction sequences are equivalent.

The many and varied parts of the 8088's instruction set are surprisingly interchangeable. Don't hesitate to mix and match them in unusual ways.

NEVER USE TWO JUMPS WHEN ONE WILL DO

Don't use a conditional jump followed by an unconditional jump

when the conditional jump can do the job by itself. Generally, a conditional jump should only be paired with an unconditional jump when the 1-byte displacement of the conditional jump can't reach the desired offset--that is, when the offset to be branched to is more than -128 to +127 bytes away.

For example:

```
        jz      IsZero
```

works fine unless **IsZero** is more than -128 or +127 bytes away from the first byte of the instruction immediately following the **jz** instruction. (You'll recall that we found in the last chapter that conditional jumps, like all jumps that use displacements, actually branch relative to the offset of the start of the following instruction.) If, however, **IsZero** is more than -128 or +127 bytes away, the polarity of the conditional jump must be reversed, and the conditional jump must be used to skip around the unconditional jump:

```
        jnz     NotZero
        jmp     IsZero
NotZero:
```

When the conditional jump falls through (in the case that resulted in a branch in the first example), the 2-byte displacement of the unconditional jump can be used to jump to **IsZero** no matter where in the code segment **IsZero** may be.

Logically, the two examples we've just covered are equivalent, branching in exactly the same cases. There's an obvious difference in the

way the two examples run, though--the first example branches in only one of the two cases, while the second example always branches, and is larger too.

In this case, it's pretty clear which is the code of choice (at least, I hope it is!)--you'd only use a conditional jump around an unconditional jump when a conditional jump alone can't reach the target label. However, paired jumps can also be eliminated in a number of less obvious situations.

For example, suppose that you want to scan a string until you come to either a character that matches the character in AH or a zero byte, whichever comes first. You might conceptualize the solution as follows:

1) Get the next byte.
2) If the next byte matches the desired byte, we've got a match and we're done.
3) If the next byte is zero, we're done without finding a match.
4) Repeat 1).

That sort of thinking is likely to produce code like that shown in Listing 13-10, which is a faithful line-by-line reproduction of the above sequence.

Listing 13-10 works perfectly well, finishing in 431 us. However, the loop in Listing 13-10 ends with a conditional jump followed by an unconditional jump. With a little code rearrangement, the conditional jump can be made to handle both the test-for-zero and repeat-loop functions, and the unconditional jump can be done away with entirely. All we need do is

put the "no-match" handling code right after the conditional jump and change the polarity of the jump from **jz** to **jnz**, so that the one conditional jump can either fall through if the terminating zero is found or repeat the loop otherwise.

Back in Chapter 11 we saw Listing 11-11, which features just such rearranged code. (Listing 13-10 is actually Listing 11-11 modified to illustrate the perils of using two jumps when one will do.) Listing 11-11 runs in just 375 us. Not only is Listing 11-11 faster than Listing 13-10, it's also shorter by two bytes--the length of the eliminated jump.

Look to streamline your code whenever you see a short unconditional jump paired with a conditional jump. Of course, it's not always possible to eliminate paired jumps, but you'd be surprised at how often loops can be compacted and speeded up with a little rearrangement.

JUMP TO THE LAND OF NO RETURN

It's not uncommon that the last action before returning at the end of a subroutine is to call another subroutine, as follows:

```
                        call    SaveNewSymbol
                        ret
PromptForSymbol endp
```

What's wrong with this picture? That's easy: there's a branch to a branch here. The **ret** that ends **SaveNewSymbol** branches directly to the **ret** that follows the call to **SaveNewSymbol** at the end of **PromptForSymbol**.

Surely there's a better way!

Indeed there is a better way, and that is to end **PromptForSymbol** by jumping to **SaveNewSymbol** rather than calling it. To wit:

```
                jmp     SaveNewSymbol
PromptForSymbol endp
```

The **ret** at the end of **SaveNewSymbol** will serve perfectly well to return to the code that called **PromptForSymbol**, and by doing this we'll save one complete **ret** plus the performance difference between **jmp** and **call**--all without changing the logic of the code in the least.

One caveat regarding **jmp** in the place of **call**/**ret**: make sure that the types--near or far--of the two subroutines match. If **SaveNewSymbol** is near-callable but **PromptForSymbol** happens to be far-callable, then the **ret** instructions at the ends of the two subroutines are not equivalent, since near and far **ret** instructions perform distinctly different actions. Mismatch **ret** instructions in this way and you'll unbalance the stack, in the process most likely crashing your program--so exercise caution when replacing **call**/**ret** with **jmp**.

DON'T BE AFRAID TO DUPLICATE CODE

Whenever you use an unconditional jump, ask yourself, "Do I really need that jump?" Often the answer is yes...but not always.

What are unconditional jumps used for? Generally, they're used to allow a conditionally-executed section of code to rejoin the main flow of program execution. For example, consider the following:

```
;
; Subroutine to set AH to 1 if AL contains the
; character 'Y', AH to 0 otherwise.
;
; Input:
;                       AL = character to check
;
; Output;
;                       AH = 1 if AL contains 'Y', 0 otherwise
;
; Registers altered: AH
;
CheckY          proc    near
                cmp     al,'Y'
                jnz     CheckYNo
                mov     ah,1                    ;it is indeed 'Y'
                jmp     short CheckYDone
CheckYNo:
                sub     ah,ah   ;it's not 'Y'
CheckYDone:
                ret
CheckY          endp
```

(You'll instantly recognize that the whole subroutine could be speeded up simply by preloading one of the values, as we learned a few sections back. In this particular case, however, we have a still better option available.) You'll notice that **jmp short CheckYDone**, the one unconditional jump in the above subroutine, doesn't actually serve much purpose. Sure, it rejoins the rest of the code after handling the case where AL is 'Y', but all that happens at that point is a return to the calling code. Surely it doesn't make sense to expend the time and 2 bytes required by a **jmp short** just to get to a **ret** instruction. Far better to simply replace the **jmp short** with a **ret**:

```
CheckY          proc    near
                cmp     al,'Y'
                jnz     CheckYNo
                mov     ah,1                ;it is indeed 'Y'
                ret
CheckYNo:
                sub     ah,ah       ;it's not 'Y'
CheckYDone:
                ret
CheckY          endp
```

The net effect: the code is 1 byte shorter, the time required for a branch is saved about half the time--and there is absolutely no change in the logic of the code. It's important that you understand that **jmp short** was basically a **nop** instruction in the first example, since all it did was unconditionally branch to another branching instruction, as shown in Figure 13-9. We removed the unconditional jump simply by replacing it with a copy of the code that it branched to.

The basic principle here is that of duplicating code. Many unconditional jumps can be eliminated by replacing the jump with a copy of the code at the jump destination. (Unconditional jumps used for looping are an exception. As we found earlier, however, unconditional jumps used to end loops can often be replaced by conditional jumps, improving both performance and code size in the process.) Often the destination code is many bytes long, and in such cases code duplication doesn't pay. However, in many other cases, such as the example shown above, code duplication is an unqualified winner, saving both cycles and bytes.

There are also cases where code duplication saves cycles but

costs bytes, and then you'll have to decide which of the two matters more on a case-by-case basis. For instance, suppose that the last example required that AL be anded with 0DFh (not 20h) after the test for 'Y'. The standard code would be:

```
;
; Subroutine to set AH to 1 if AL contains the
; character 'Y', AH to 0 otherwise. AL is then forced to
; uppercase. (AL must be a letter.)
;
; Input:
;                       AL = character to check (must be a letter)
;
; Output;
;                       AH = 1 if AL contains 'Y', 0 otherwise
;                       AL = character to check forced to uppercase
;
; Registers altered: AX
;
CheckY          proc    near
                cmp     al,'Y'
                jnz     CheckYNo
                mov     ah,1                            ;it is indeed 'Y'
                jmp     short CheckYDone
CheckYNo:
                sub     ah,ah               ;it's not 'Y'
CheckYDone:
                and     al,not 20h          ;make it uppercase
                ret
CheckY          endp
```

The duplicate-code implementation would be:

```
CheckY          proc    near
                cmp     al,'Y'
                jnz     CheckYNo
                mov     ah,1        ;it is indeed 'Y'
                and     al,not 20h          ;make it uppercase
                ret
CheckYNo:
                sub     ah,ah       ;it's not 'Y'
CheckYDone:
                and     al,not 20h          ;make it uppercase
                ret
CheckY          endp
```

with both **and** and **ret** duplicated at the end of each of the two possible

paths through the subroutine.

The decision as to which of the two above implementations is preferable is by no means cut and dried. The duplicated-code implementation is certainly faster, since it still avoids a branch in half the cases. On the other hand, the duplicated-code implementation is also 1 byte longer, since a 2-byte **jmp short** is replaced with a 3-byte sequence of **and** and **ret**. Neither sequence is superior on all counts, so the choice between the two depends on context and your own preferences.

Duplicated code is counter to all principles of structured programming. As we've learned, that's not inherently a bad thing--when you need performance, it can be most useful to discard conventions and look for fresh approaches.

Nonetheless, it's certainly possible to push the duplicated- code approach too far. As the code to be duplicated becomes longer and/or more complex, the duplicated-code approach becomes less appealing. In addition to the bytes that duplicating longer code can cost, there's also the risk that you'll modify the code at only one of the duplicated locations as you alter the program. For this reason, duplicated code sequences longer than a **ret** and perhaps one other instruction should be used only when performance is at an absolute premium.

INSIDE LOOPS IS WHERE BRANCHES REALLY HURT

Branches always hurt performance, but where they really hurt is inside loops. There, the performance loss incurred by a single branching

instruction is magnified by the number of loop repetitions. It's important that you understand that not all branches are created equal, so that you can focus on eliminating or at least reducing the branches that most affect performance-- and those branches are usually inside loops.

How can we apply this knowledge? By making every effort to use techniques such as duplicated code, in-line code (which we'll see shortly), and preloading values inside loops, and by simply moving decision-making out of loops whenever we can. Let's take a look at an example of using duplicated code within a loop, in order to see how easily cycle-saving inside a loop can pay off.

TWO LOOPS CAN BE BETTER THAN ONE

Suppose that we want to determine whether there are more negative or non-negative values in an array of 8-bit signed values. Listing 13-11 does that in 3.60 ms for the sample array by using a straightforward and compact test-and-branch approach. There's nothing wrong with Listing 13-11, but there is an unconditional jump. We'd just as soon do away with that unconditional jump, especially since it's in a loop. Unfortunately, the instruction the unconditional jump branches to isn't a simple **ret**--it's a **loop** instruction, and we all know that loops must end in one place, at the loop bottom.

Hmmmm. Why must loops end in one place? There's no particular reason that I can think of, apart from habit, so let's try duplicating some code and ending the loop in two places. Listing 13-12, which does

exactly that, runs in just 3.05 ms. That's an improvement of 18%--quite a return for the 1 byte the duplicated-code approach adds.

It's evident that eliminating branching instructions inside loops can result in handsome performance gains for relatively little effort. That's why I urge you to focus your optimization efforts on loops. While we're on this important topic, let's look at another way to eliminate branches inside loops.

MAKE UP YOUR MIND ONCE AND FOR ALL

If you find yourself making a decision inside a loop, for heaven's sake see if you can manage to make that decision before the loop. Why decide every time through the loop when you can decide just once at the outset?

Consider Listing 13-13, in which the contents of DL are used to decide whether to convert each character to uppercase while copying one string to another string. Listing 13-13, which runs in 3.03 ms for the sample string, is representative of the situation in which a parameter passed to a subroutine selects between different modes of operation.

The failing of Listing 13-13 is that the decision as to whether to convert to uppercase is made over and over, once for each character. We'd be much better off if we could make the decision just once at the start of the subroutine, moving the decision-making (particularly the branching) out of the loop.

There are a number of ways to do this. One is shown in Listing

13-14. Here, a single branch outside the loop is used to force the test for inclusion in the lowercase to function also as the test for whether conversion is desired. If conversion isn't desired, AH, which normally contains the start of the lowercase range, is set to 0FFh. This has the effect of causing the lowercase test always to fail on the first conditional jump if conversion isn't desired, just as was the case in Listing 13-13. Consequently, performance stays just about the same when conversion to uppercase isn't desired.

However, when lowercase conversion is desired, Listing 13-14 performs one less test each time through the loop than does Listing 13-13, because a separate test to find out whether conversion is desired is no longer needed. We've already performed the test for whether conversion is desired at the start of the subroutine--outside the loop--so the code inside the loop can sail through the copy-and-convert process at full speed. The result is that Listing 13-14 runs in 2.76 ms, significantly faster than Listing 13-13.

In Listing 13-14, we've really only moved the test as to whether conversion is desired out of the loop in the case where conversion is indeed desired. When conversion isn't desired, a branch is still performed every time through the loop, just as in Listing 13-13. If we're willing to duplicate a bit of code, we can also move the branch out of the loop when conversion isn't desired, as shown in Listing 13-15. There's a cost in size for this optimization--7 bytes--but execution time is cut to just 2.35 us, a 29% improvement over Listing 13-13.

Moreover, Listing 13-15 could easily be speeded up further by

using the word-at-a-time or **scas**/**movs** techniques we encountered in Chapter 11. Why is it easier to do this to Listing 13-15 than to Listing 13-13? It's easier because we've completely separated the instruction sequences for the two modes of operation of the subroutine, so we have fewer instructions and simpler code to optimize in whichever case we try to speed up.

Remember, not all branches are created equal. If you have a choice between branching once before a loop and branching once every time through the loop, it's really like choosing between one branch and dozens or hundreds (however many times the loop is repeated) of branches. Even when it costs a few extra bytes, that's not a particularly hard choice to make, is it?

DON'T COME CALLING

Jumps aren't the 8088's only branching instructions. Calls, returns, and interrupts branch as well. Interrupts aren't usually repeated unnecessarily inside loops, although you should try to handle data obtained through DOS interrupts in large blocks, rather than a character at a time, as we'll see in the next chapter.

By definition, returns can't be executed repeatedly inside loops, since a return branches out of a loop back to the calling code.

That leaves calls...and calls in loops are in fact among the great cycle-wasters of the 8088.

Consider what the **call** instruction does. First it pushes the

Instruction Pointer onto the stack, and then it branches. That's like pairing a **push** and a **jmp**--a gruesome prospect from a performance perspective. Actually, things aren't that bad; the official execution time of **call**, at 23 cycles, is only 8 cycles longer than that of **jmp**. Nonetheless, you should cast a wary eye on any instruction that takes 23 cycles to execute and empties the prefetch queue.

The cycles spent executing **call** aren't the end of the performance loss associated with calling a subroutine, however. Once you're done with a subroutine, you have to branch back to the calling code. The instruction that does that, **ret**, takes another 20 cycles and empties the prefetch queue again. On balance, then, a subroutine call expends 43 cycles on overhead operations and empties the prefetch queue not once but twice!

Fine, you say, but what's the alternative? After all, subroutines are fundamental to good programming--we can't just do away with them altogether.

By and large, that's true, but inside time-critical loops there's no reason why we can't eliminate calls simply by moving the called code into the loop. Replacing the subroutine call with a macro is the simplest way to do this. For example, suppose that we have a subroutine called **IsPrintable**, which tests whether the character in AL is a printable character (in the range 20h to 7Eh). Listing 13-16 shows a loop that calls this subroutine in the process of copying only printable characters from one string to another string. Call and all, Listing 13-16 runs in 3.48 ms for the test string.

Listing 13-17 is functionally identical to Listing 13-16. In Listing 13-17, however, the call to the subroutine **IsPrintable** has been converted to the expansion of the macro **IS_PRINTABLE**, eliminating the **call** and **ret** instructions. How much difference does that change from call to macro expansion make? Listing 13- 17 runs in 2.21 ms, 57% faster than Listing 13-16. Listing 13- 16 spends over one-third of its entire execution time simply calling **IsPrintable** and returning from that subroutine!

While the superior performance of Listing 13-17 clearly illustrates the price paid for subroutine calls, that listing by no means applies all of the optimizations made possible by the elimination of the calls that plagued Listing 13-16. It's true that the macro **IS_PRINTABLE** eliminates the subroutine call, but there are still internal branches in **IS_PRINTABLE**, and there's still a **cmp** instruction that sets the Zero flag on success. In other words, Listing 13-17 hasn't taken full advantage of moving the code into the loop; it has simply taken the call and return overhead out of determining whether a character is printable.

Listing 13-18 does take full advantage of moving the test code into the loop, by eliminating the macro and thereby eliminating the need to place a return status in the Zero flag. Instead, Listing 13-18 branches directly to **NotPrintable** if a character is found to be non-printable, eliminating the intermediate conditional jump that Listing 13-17 performed. It's also no longer necessary to test the Zero flag to see whether the character is printable before storing it in the destination array, since any character that passes the two comparisons for inclusion in the printable range must be

printable. The upshot is that Listing 13-18 runs in just 1.74 ms, 27% faster than Listing 13-17 and 100% faster than Listing 13-16.

Listing 13-18 illustrates two useful optimizations in the case where a character is found to be printable. First, there's no need to branch to the bottom of the loop just to branch back to the top of the loop, so Listing 13-18 just branches directly to the top of the loop after storing each printable character. The same is done when a non-printable character greater than 7Eh is detected. The point here is that it's fine to branch back to the top of a loop from multiple places. Second, there's no way that a printable character can end a string (zero isn't a printable character), so we don't bother testing for the terminating zero after storing a printable character; again, the same is true for non-printable characters greater than 7Eh. When you duplicate code, it's not necessary to duplicate any portion of the code that performs no useful function in the new location.

Whenever you use a subroutine or a macro, you're surrendering some degree of control over your code in exchange for ease of programming. In particular, the use of subroutines involves a direct trade-off of decreased performance for reduced code size and greater modularity. In general, ease of programming, reduced code size, and modularity are highly desirable attributes...but not in time-critical code.

Try to eliminate calls from your tight loops and time- critical code. If the code called is large, that may not be possible, but then you have to ask yourself what such a large subroutine is doing in your time-critical code in the first place. It may also be beneficial to eliminate macros in time- critical

code. Whether or not that's the case depends on the nature of the macros, but at least make sure you understand what code you're really writing. In this pursuit, it can be useful to generate a listing file in order to see the code the assembler is actually generating.

As I mentioned above, there are three objections to moving subroutines into loops: size, modularity, and ease of programming. Let's quickly address each of these points.

Sure, code gets bigger when you move subroutines into loops: performance is often a balancing of program size and performance. That's why you should concentrate on applying the techniques in this chapter (and, indeed, all the performance-enhancing techniques presented in The Zen of Assembly Language) to time- critical code, where a few extra bytes can buy a great many cycles.

On the other hand, code doesn't really have to be less modular when subroutines are moved into loops. Macros are just as modular as subroutines, in the sense that in your code both are one-line entries that perform a well-defined set of actions. In any case, in discussing moving subroutine code into loops we're generally talking about moving relatively few instructions into any given loop, since the call/return overhead becomes proportionately less significant for longer subroutines (although never insignificant, if you're really squeezed for cycles). Modularity shouldn't be a big issue with short instruction sequences.

Finally, as to ease of programming: if you want easy programming, program in C or Pascal, or, better yet, COBOL. Assembler

subroutine and macro libraries are fine for run-of-the- mill code, but when it comes to the high-performance, time- critical parts of your programs, it's your ability to write the hard assembler code that will set you apart. Assembler isn't easy, but any competent programmer can eventually get almost any application to work in assembler. The Zen of assembler lies not in making an application work, but in making it work as well as it possibly can, given the strengths and limitations of the PC.

SMALLER ISN'T ALWAYS BETTER

You've no doubt noticed that this chapter seems to have repeatedly violated the rule that "smaller is better." Not so, given the true meaning of the rule. "Smaller is better" applies to instruction prefetching, where fewer bytes to be fetched means less time waiting for instruction bytes. Subroutine calls don't fall into this category, even though they reduce overall program size.

Subroutines merely allow you to run the same instructions from multiple places in a program. That reduces program size, since the code only needs to appear in one place, but there are no fewer bytes to be fetched on any given call than if the code of the subroutine were to be placed directly into the calling code. In fact, instruction fetching becomes more of a problem with subroutines, since the prefetch queue is emptied twice, and the call and return instruction bytes must be fetched, as well.

In short, while subroutines are great for reducing program size and have a host of other virtues as regards program design, modularity, and maintenance, they don't come under the "smaller is better" rule, and are, in

fact, lousy for performance. Much the same--smaller is slower--can be said of branches of many sorts. Of all the branching instructions, loops are perhaps the worst "smaller is slower" offender. We're going to close out this chapter with a discussion of the potent in-line-code alternative to looping--yet another way to trade a few bytes for a great many cycles.

loop MAY NOT BE BAD, BUT LORD KNOWS IT'S NOT GOOD: IN-LINE CODE

One of the great misconceptions of 8088 programming is that **loop** is a good instruction for looping. It's true that **loop** is designed especially for looping. It's also true that **loop** is the 8088's best looping instruction. But good?

No way.

You see, **loop** is a branching instruction, and not an especially fast branching instruction, at that. The official execution time of **loop** is 17 cycles, which makes it just 1 cycle faster than the similar construct **dec cx**/**jnz**, although **loop** is also 1 byte shorter. Like all branching instructions, **loop** empties the prefetch queue, so it is effectively even slower than it would appear to be. I don't see how you can call an instruction that takes in the neighborhood of 20 cycles just to repeat a loop good. Better than the obvious alternatives, sure, and pleasantly compact and easy to use if you don't much care about speed--but not good.

Look at it this way. Suppose you have a program containing a loop that zeros the high bit of each byte in a 100-byte array, as shown in

Listing 13-19, which runs in 1023 us. What percent of that overall execution time do you suppose this program spends just decrementing CX and branching back to the top of the loop-- that is, looping? Ten percent?

No. Twenty percent?

No.

Thirty percent?

No, but you're getting warm...Listing 13-19 spends forty- five percent of the total execution time looping. (That figure was arrived at by comparing the execution time of Listing 13-20, which uses no branches and which we'll get to shortly, to the execution time of Listing 13-19.) Yes, you read that correctly-- in a loop which accesses memory twice and which contains a second instruction in addition to the memory-accessing instruction, **loop** manages to take nearly one-half of the total execution time. Appalling?

You bet.

Still, while **loop** may not be much faster than other branching instructions, it is nonetheless somewhat faster, and it's also more compact. We know we're losing a great deal of performance to the 8088's abysmal branching speed, but there doesn't seem to be much we can do about it.

But of course there is something we can do, as is almost always the case with the 8088. Let's look at exactly what **loop** is used for, and then let's see if we can produce the same functionality in a different way.

Well, **loop** is used to repeat a given sequence of instructions multiple times...and that's about all. What can we do with that job

description?

Heck, that's easy. We'll eliminate branching and loop counting entirely by literally repeating the instructions, as shown in Figure 13-10. Instead of using **loop** to execute the same code, say, 10 times, we'll just line up 10 repetitions of the code inside the loop, and then execute the 10 repetitions one after another. This is known as in-line code, because the repetitions of the code are lined up in order rather than being separated by branches. (In-line code is sometimes used to refer to subroutine code that's brought into the main code, eliminating a call, a technique we discussed in the last section. However, I'm going to use the phrase "in-line code" only to refer to code that's repeated by assembling multiple instances and running them back-to-back rather than in a loop.)

Listing 13-20 shows in-line code used to speed up Listing 13-19. The **loop** instruction is gone, replaced with a **rept** directive that creates 100 back-to-back instances of the code inside the loop of Listing 13-19. The performance improvement is dramatic: Listing 13-20 runs in 557 us, more than 83% faster than Listing 13-19.

Often-enormous improvement in performance is the good news about in-line code. Often-enormous increase in code size-- depending on the number of repetitions and the amount of code in the loop--is the bad news. Listing 13-20 is nearly 300 bytes larger than Listing 13-19. On the other hand, we're talking about nearly doubling performance by adding those extra bytes. Yes, once again we've encountered the trade-off between bytes and cycles that pops up so often when we set out to improve

performance: in-line code can be used to speed up just about any loop, but the cost in bytes ranges from modest to prohibitively high. Still, when you need flat-out performance, in-line code is a tried and true way to get a sizable performance boost.

In-line code has another benefit beside eliminating branching. When in-line code is used, CX (or whatever register would otherwise have been used as a loop counter) is freed up. An extra 16-bit register is always welcome in high-performance code.

You may well object at this point that in-line code is fine when the number of repetitions of a loop is known in advance and is always the same, but how often is that the case? Not all that often, I admit, but it does happen. For example, think back to our animation examples in Chapter 11. The example that used exclusive-or-based animation looped once for each word exclusive- ored into display memory, and always drew the same number of words per line. That sounds like an excellent candidate for in-line code, and in fact it is.

Listing 13-21 shows the **XorImage** subroutine from Listing 11- 33 revised to use in-line code to draw each line without branching. Instead, the four instructions that draw the four words of the image are duplicated four times, in order to draw a whole line at a time. This frees up not only CX but also BP, which in Listing 11-33 was used to reload the number of words per line each time through the loop. That has a ripple effect which lets us avoid using BX, saving a **push** and a **pop**, and also allows us to store the offset from odd lines to even lines in a register for added speed.

The net effect of the in-line code in Listing 13-21 is far from trivial. When this version of **XorImage** is substituted for the version in Listing 11-33, execution time drops from 30.29 seconds to 24.21 seconds, a 25% improvement in overall performance. Put another way, the **loop** instructions in the two loops that draw the even and odd lines in Listing 11-33 take up about one out of every five cycles that the entire program uses! Bear in mind that we're not talking now about a program that zeros the high bits of bytes in three-instruction loops; we're talking about a program that performs complex animation and accesses display memory heavily...in other words, a program that does many time-consuming things besides looping.

To drive the point home, let's modify Listing 11-34 to use in-line code, as well. Listing 11-34 uses **rep movsw** to draw each line, so there are no branches to get rid of during line drawing, and consequently no way to put in-line code to work there. There is, however, a loop that's used to repeat the drawing of each pair of rows in the image. That's not nearly so intensive a loop as the line-drawing loop was in Listing 11-33; instead of being repeated once for every word that's drawn, it's repeated just once every two lines, or 10 words.

Nonetheless, when the in-line version of **BlockDrawImage** shown in Listing 13-22 is substituted for the version in Listing 11-34, overall execution time drops from 10.35 seconds to 9.69 seconds, an improvement of nearly 7%. Not earthshaking--but in demanding applications such as animation, where every cycle counts, it's certainly worth expending a few hundred extra bytes to get that extra speed.

The 7% improvement we got with Listing 13-22 is more impressive when you consider that the bulk of the work in Listing 11-34 is done with **rep movsw**. If you take a moment to contemplate the knowledge that 7% of overall execution time in Listing 11-34 is used by just 20 **dec dx**/**jnz** pairs per image draw (and remember that cycle-eating display memory is accessed 400 times for every 20 **dec dx**/**jnz** pairs executed), you'll probably reach the conclusion that **loop** really isn't a very good instruction for high-performance looping.

And you'll be right.

BRANCHED-TO IN-LINE CODE: FLEXIBILITY NEEDED AND FOUND

What we've just seen is "pure" in-line code, where a loop that's always repeated a fixed number of times is converted to in-line code by simply repeating the contents of the loop however many times the loop was repeated. The above animation examples notwithstanding, pure in-line code isn't used very often. Why? Because loops rarely repeat a fixed number of times, and pure in- line code isn't flexible enough to handle a variable number of repetitions. With pure in-line code, if you put five repetitions of a loop in-line, you'll always get five repetitions, no more and no less. Most looping applications demand more flexibility than that.

As it turns out, however, it's no great trick to modify pure in-line code to replace loops that repeat a variable number of times, so long as you know the maximum number of times you'll ever want to repeat the loop. The basic concept is shown in Figure 13-11. The loop code is repeated in-

line as many times as the maximum possible number of loop repetitions. Then the specified repetition count is used to jump right into the in- line code at the distance from the end of the in-line code that will produce the desired number of repetitions. This mechanism, known as branched-to in-line code, is almost startlingly simple, but powerful nonetheless.

Let's convert the in-line code example of Listing 13-20 to use branched-to in-line code. Listing 13-23 shows this implementation. First, in-line code to support up to the maximum possible number of repetitions (in this case, 200) is created with **rept**. Then the start offset in the in-line code that will result in the desired number of repetitions is calculated, by multiplying the number of instruction bytes per repetition by the desired number of repetitions, and subtracting the result from the offset of the end of the table. As a result, Listing 13-23 can handle any number of repetitions between 0 and 200, and does so with just one branch, the **jmp cx** that branches into the in- line code.

The performance price for the flexibility of Listing 13-23 is small; the code runs in 584 us, just 27 us slower than Listing 13-20. Moreover, Listing 13-23 could be speeded up a bit by multiplying by 3 with a shift-and-add sequence rather than the notoriously slow **mul** instruction; I used **mul** in order to illustrate the general case and because I didn't want to obscure the workings of branched-to in-line code.

Branched-to in-line code retains almost all of the performance advantages of in-line code, without the inflexibility. Branched-to in-line code does everything **loop** does, and does it without branching inside the loop.

Branched-to in-line code is sort of the poor man's **rep**, capable of repeating any instruction or sequence of instructions without branching, just as **rep** does for string instructions. It's true that branched-to in-line code doesn't really eliminate the prefetch- queue cycle-eater as **rep** does, since each instruction byte in branched-to in-line code must still be fetched. On the other hand, it's also true that branched-to in-line code eliminates the constant prefetch-queue flushing of **loop**, and that's all to the good.

In short, branched-to in-line code allows repetitive processing based on non-string instructions to approach its performance limits on the 8088 by eliminating branching, thereby doing away with not only the time required to branch but also the nasty prefetch-queue effects of branching. When you need flat- out speed for repetitive tasks, branched-to in-line code is often a good bet.

That's not to say that branched-to in-line code is perfect. The hitch is that you must allow for the maximum number of repetitions when setting up branched-to in-line code. If you're performing checksums on data blocks no larger than 64 bytes, the maximum size is no problem, but if you're working with large arrays, the maximum size can easily be either unknown or so large that the resulting in-line code would simply be too large to use. For example, the in-line code in Listing 13-23 is 600 bytes long, and would swell to 60,000 bytes long if the maximum number of repetitions were 20,000 rather than 200. In-line code can also become too large to be practical after just a few repetitions if the code to be repeated is lengthy. Finally, lengthy branched-to in-line code isn't well-suited for tasks such as scanning arrays,

since the in-line code can easily be too long to allow the 1-byte displacements of conditional jumps to branch out of the in-line code when a match is found.

Clearly, branched-to in-line code is not the ideal solution for all situations. Branched-to in-line code is great if both the maximum number of repetitions and the code to be repeated are small, or if performance is so important that you're willing to expend a great many bytes to speed up your code. For applications that don't fit within those parameters, however, a still more flexible in-line solution is needed.

Which brings us to partial in-line code.

PARTIAL IN-LINE CODE

Partial in-line code is a hybrid of normal looping and pure in-line code. Partial in-line code performs a few repetitions back-to-back without branching, as in-line code does, and then loops. As such, partial in-line code offers much of the performance improvement of in-line code, along with much of the compactness of normal loops. While partial in-line code isn't as fast as pure or branched-to in-line code, it's still fast, and because it's relatively compact, it overcomes most of the size- related objections to in-line code.

Let's go back to our familiar example of zeroing the high bit of each byte in an array to see partial in-line code in action. In Listing 13-19 we saw this example implemented with a loop, in Listing 13-20 we saw it implemented with pure in-line code, and in Listing 13-23 we saw it

implemented with branched-to in-line code. Listing 13-24 shows yet another version, this time using partial in-line code.

The key to Listing 13-24 is that it performs four in-line bit-clears, then loops. This means that Listing 13-24 loops just once for every four bits cleared. While that means that Listing 13-24 still branches 25 times, that's 75 times fewer than the loop-only version, Listing 13-19, certainly a vast improvement. And while the **ClearHighBits** subroutine is 13 bytes larger in Listing 13-24 than in Listing 13-19, it's nearly 300 bytes smaller than in the pure in-line version, Listing 13-20. If Listing 13-24 can run anywhere near as fast as Listing 13-20, it'll be a winner.

Listing 13-24 is indeed a winner, running in 688 us. That's certainly slower than pure in-line code--Listing 13-20 is about 24% faster--but it's a whole lot faster than pure looping. Listing 13-24 outperforms Listing 13-19 by close to 50%--at a cost of just 13 bytes. That's a terrific return for the extra bytes expended, proportionally much better than the 83% improvement Listing 13-20 brings at a cost of 295 bytes. To put it another way, in this example the performance improvement of partial in-line code over pure looping is about 49%, at a cost of 13 bytes, while the improvement of pure in-line code over partial in-line code is only 24%, at a cost of 282 bytes.

If you need absolute maximum speed, in-line code is the ticket...but partial in-line code offers similar performance improvements in a far more generally usable form. If size is your driving concern, then **loop** is the way to go.

As always, no one approach is perfect in all situations. The three approaches to handling repetitive code that we've discussed--in-line code, partial in-line code, and looping--give you a solid set of tools to use for handling repetitive tasks, but it's up to you to evaluate the trade-offs between performance, size, and program complexity and then select the proper techniques for your particular applications. There are no easy answers in top-notch assembler programming--but at least now you have a set of tools with which to craft good solutions.

There are many, many ways to use in-line code. We've seen some already, we'll see more over the remainder of this chapter, and you'll surely discover others yourself. Whenever you must loop in time-critical code, take a moment to see if you can't use in-line code in one of its many forms instead.

The rewards can be rich indeed.

PARTIAL IN-LINE CODE: LIMITATIONS AND WORKAROUNDS

The partial in-line code implementation in Listing 13-24 is somewhat more flexible than the pure in-line code implementation in Listing 13-20, but not by much. The partial in-line code in Listing 13-24 is capable of handling only repetition counts that happen to be multiples of four, since four repetitions are performed each time through the loop. That's fine for repetitive tasks that always involve repetition counts that happen to be multiples of four; unfortunately, such tasks are the exception rather than the rule. In order to be generally useful, partial in-line code must be able to

support any number of repetitions at all.

As it turns out, that's not a problem. The flexibility of branched-to in-line code can easily be coupled with the compact size of partial in-line code. As an example, let's modify the branched-to in-line code of Listing 13-23 to use partial in-line code.

The basic principle when branching into partial in-line code is similar to that for standard branched-to in-line code. The key is still to branch to the location in the in-line code from which the desired number of repetitions will occur. The difference with branched-to partial in-line code is that the branching-to process only needs to handle any odd repetitions that can't be handled by a full loop, as shown in Figure 13-12. In other words, if partial in-line code performs n repetitions per loop and we want to perform m repetitions, the branching-to process only needs to handle m modulo n repetitions.

For example, if we want to perform 15 repetitions with partial in-line code that performs 4 repetitions per loop, we need to branch so as to perform the first 15 modulo 4 = 3 repetitions during the first, partial pass through the loop. After that, 3 full passes through the loop will handle the other 12 repetitions.

Listing 13-25, a branched-to partial in-line code version of our familiar bit-clearing example, should help to make this clear. The version of **ClearHighBits** in Listing 13-25 first calculates the number of repetitions modulo 4. Since each pass through the loop performs 4 repetitions, the number of repetitions modulo 4 is the number of repetitions to be performed

on the first, partial pass through the loop in order to handle repetition counts that aren't multiples of 4. Listing 13-25 then uses this value to calculate the offset in the partial in-line code to branch to in order to cause the correct number of repetitions to occur on that first pass.

Incidentally, multiplication by 3 in Listing 13-25 is performed not with **mul**, but with a much faster shift-and-add sequence. As we mentioned earlier, the same could have been done in Listing 13-23, but **mul** was used there in order to handle the general case and avoid obscuring the mechanics of the branching- to process. In the next chapter we'll see a jump-table-based approach that does away with the calculation of the target offset in the in-line code entirely, in favor of simply looking up the target address.

Next, Listing 13-25 divides the repetition count by 4, since 4 repetitions are performed each time through the loop. That value must then be incremented to account for the first pass through the loop--and that's it! All we need do is branch to the correct location in the partial in-line code and let it rip. And rip it does, with Listing 13-25 running in just 713 us. Yes, that is indeed considerably slower than the 584 us time of the branched-to in-line code of Listing 13-23, but it's much faster than the 1023 us of Listing 13-19. Then, too, Listing 13-25 is only 32 bytes larger than Listing 13-19, while Listing 13-23 is more than 600 bytes larger.

Listing 13-25, the branched-to partial in-line code, has an additional advantage over Listing 13-23, the branched-to in-line code, and that's the ability to handle an array of <u>any</u> size up to 64 K-1. With in-line code, the largest number of repetitions that can be handled is determined by

the number of times the code is physically repeated. Partial in-line code suffers from no such restriction, since it loops periodically. In fact, branched-to partial in-line code implementations can handle any case normal loops can handle, tend to be only a little larger, and are much faster for all but very small repetition counts. Listing 13-25 itself isn't quite equivalent to a **loop**-based loop. Given an initial count of zero, **loop** performs 64 K repetitions, while Listing 13-25 performs 0 repetitions in the same case. That's not necessarily a disadvantage; **loop**-based loops are often preceded with **jcxz** in order to cause zero counts to produce 0 repetitions. However, Listing 13-25 can easily be modified to treat an initial count of zero as 64 K; I chose to perform 0 repetitions given a zero count in Listing 13-25 only because it made for code that was easier to explain and understand. Listing 13-26 shows the **ClearHighBits** subroutine of Listing 13-25 modified to perform 64 K repetitions given an initial count of zero.

It's worth noting that the **inc ax** in Listing 13-26 could be eliminated if the line:

```
        mov     dx,offset InLineBitClearEnd
```

were changed to:

```
        mov     dx,offset InLineBitClearEnd-3
```

This change has no effect on overall functionality, because the net effect of **inc ax** in Listing 13-26 is merely to subtract 3 from the offset of the end of the partial in-line code. I omitted this optimization in the interests of making Listing 13- 26 comprehensible, but as a general practice arithmetic should be performed at assembly time rather than at run time whenever possible.

By the way, there's nothing special about using 4 repetitions in partial in-line code. 8 repetitions or even 16 could serve as well, and, in fact, speed increases as the number of partial in-line repetitions increases. However, size increases proportionately as well, offsetting part of the advantage of using partial in-line code. Partial in-line code using 4 repetitions strikes a nice balance between size and speed, eliminating 75% of the branches without adding too many instruction bytes.

PARTIAL IN-LINE CODE AND STRINGS: A GOOD MATCH

One case in which the poor repetition granularity of partial in-line code (that is, the inability of partial in-line loops to deal unaided with repetition counts that aren't exact multiples of the number of repetitions per partial in-line loop) causes no trouble at all is in handling zero-terminated strings. Since there is no preset repetition count for processing such strings, it doesn't matter in the least that the lengths of the strings won't always be multiples of the number of repetitions in a single partial in-line loop. When handling zero-terminated strings, it doesn't matter if the terminating condition occurs at the start of partial in-line code, the end, or somewhere in-between, since a conditional jump will branch out equally well from

anywhere in partial in-line code. As a result, there's no need to branch into partial in-line code when handling zero- terminated strings.

As usual, an example is the best explanation. Back in Listing 11-25, we used **lodsw** and **scasw** inside a loop to find the first difference between two zero-terminated strings. We used word- rather than byte-sized string instructions to speed processing; interestingly, much of the improvement came not from accessing memory a word at a time but rather from cutting the number of loops in half, since two bytes were processed per loop. We're going to use partial in-line code to speed up Listing 11-25 further by eliminating still more branches.

Listing 13-27 is our partial in-line version of Listing 11- 25. I've chosen a repetition granularity of 8 repetitions per loop both for speed and to show you that granularities other than 4 can be used. There's no need to add code to branch into the partial in-line code, since there's no repetition count for a zero-terminated string. Note that I've separated the eighth repetition of the partial in-line code from the first seven, so that the eighth repetition can jump directly back to the top of the loop if it doesn't find the terminating zero. If I lumped all 8 repetitions together in a **rept** block, an unconditional jump would have to follow the partial in-line code in order to branch back to the top of the loop. While that would work, it would result in a conditional jump/unconditional jump pair...and well we know to steer clear of those when we're striving for top performance. Listing 13-27 runs in 278 us, 10% faster than Listing 11-25. Considering how heavily optimized Listing 11-25 already was, what with the use of word-sized string

instructions, that's a healthy improvement. What's more, Listing 13-27 isn't markedly more complicated than Listing 11-25; actually, the only difference is that the contents of the loop are repeated 8 times rather than once.

As you can see, partial in-line code is ideal for the handling of zero-terminated strings. Once again, partial in-line code is a poor man's **rep**; in fact, in string and similar applications, you might think of partial in-line code as a substitute for the sorely-missed **rep** prefix for the flexible but slow **lods**/**stos** and **lods**/**scas** instruction pairs.

LABELS AND IN-LINE CODE

That just about does it for our discussion of in-line code. However, there's one more in-line code item we need to discuss, and that's the use of labels in in-line code.

Suppose that for some reason you need to use a label somewhere inside in-line code. For example, consider the following:

```
                rept    4
                lodsb
                cmp     al,'a'
                jb      NotUppercase
                cmp     al,'z'
                ja      NotUppercase
                and     al,not 20h
NotUppercase:
                stosb
                endm
```

In this example, the label **NotUppercase** is inside in-line code used to convert 4 characters in a row to uppercase. While the code seems simple enough, it nonetheless has one serious problem:

It won't assemble.

Why is that? The problem is that the line defining the label is inside a **rept** block, so it's literally assembled multiple times. As it would at any time, MASM complains when asked to define two labels with the same name.

The solution should be straightforward: declare the label local to the **rept** block with the **local** directive, which exists for just such emergencies. For example, the following code should do the trick:

```
                rept    4
                local   NotUppercase
                lodsb
                cmp     al,'a'
                jb      NotUppercase
                cmp     al,'z'
                ja      NotUppercase
                and     al,not 20h
NotUppercase:
                stosb
                endm
```

It should--but it doesn't, at least not with MASM 5.0. While the **local** directive does indeed solve our problem when assembled with TASM, it just doesn't work correctly when assembled with MASM 5.0. There's no use asking why--the bugs and quirks of MASM are just a fact of life in assembler programming.

So, what's the solution to our local label problem when using MASM? One possibility is counting bytes and jumping relative to the program counter, as in:

```
                rept    4
```

```
        lodsb
        cmp     al,'a'
        jb      $+8
        cmp     al,'z'
        ja      $+4
        and     al,not 20h
        stosb
        endm
```

It's not elegant, but it does work. Another possibility is defining a macro that contains the code in the **rept** block, since **local** does work in macros. For example, the following assembles properly under MASM 5.0:

```
MAKE_UPPER      macro
                local   NotUppercase
                lodsb
                cmp     al,'a'
                jb      NotUppercase
                cmp     al,'z'
                ja      NotUppercase
                and     al,not 20h
NotUppercase:
                stosb
                endm
                 :
                rept    4
                MAKE_UPPER
                endm
```

A NOTE ON SELF-MODIFYING CODE

Just so you won't think I've forgotten about it, let's briefly discuss self-modifying code. For those of you unfamiliar with this demon of modern programming, self-modifying code is a once-popular coding technique whereby a program modifies its own code--changes its own instruction bytes--on the fly in order to alter its operation without the need for tests and branches. (Remember how back in Chapter 3 we learned that code is just one kind of data? Self-modifying code is a logical extension of that concept.)

Nowadays, self-modifying code is strongly frowned- upon, on the grounds that it makes for hard-to-follow, hard-to debug programs.

"Frowned upon, eh?" you think. "Sounds like fertile ground for a little Zen programming, doesn't it?" Yes, it does. Nonetheless, I don't recommend that you use self-modifying code, at least not self-modifying code in the classic sense. Not because it's frowned-upon, of course, but rather because I haven't encountered any cases where in-line code, look-up tables, jump vectors, jumping through a register or some other 8088 technique didn't serve just about as well as self-modifying code.

Granted, there may be a small advantage to, say, directly modifying the displacement in a **jmp** instruction rather than jumping to the address stored in a word-sized memory variable, but in-line code really is hard to debug and follow, and is hard to write, as well (consider the complexities of simply calculating a jump displacement). I haven't seen cases where in- line code brings the sort of significant performance improvement that would justify its drawbacks. That's not to say such cases don't exist; I'm sure they do. I just haven't encountered them.

Self-modifying code has an additional strike against it in the form of the prefetch queue. If you modify an instruction byte after it's been fetched by the Bus Interface Unit, it's the original, unmodified byte that's executed, since that's the byte that the 8088 read. That's particularly troublesome because the various members of the 8086 family have prefetch queues of differing lengths, so self-modifying code that works on the PC might not work at all on an AT or a Model 80. A branch always empties the

prefetch queue and forces it to reload, but even that might not be true with future 8086-family processors.

To sum up, my experience is that in the context of the 8086 family, self-modifying code offers at best small performance improvements, coupled with significant risk and other drawbacks. That's not the case with some other processors, especially those with less-rich instruction sets and no prefetch queue. However, The Zen of Assembly Language is concerned only with the 8086 family, and in that context my final word on self-modifying code of the sort we've been discussing is:

Why bother?

On the other hand, I've only been discussing self-modifying code in the classic sense, where individual instructions are altered. For instance, the operand to **cmp al,immed8** might be modified to change an inclusion range; in such a case, why not just use **cmp al,reg** and load the new range bound into the appropriate register? It's simpler, easier to follow, and actually slightly faster.

There's another sort of self-modifying code, however, that operates on a grander scale. Consider a program that uses code overlays. Code is swapped in from disk to memory and then executed; obviously the instruction bytes in the overlay region are changed, so that's self-modifying code. Or consider a program that builds custom code for a special, complex purpose in a buffer and then executes the generated code; that's self-modifying code as well. Some programs are built out of loosely- coupled, relocatable blocks of code residing in a heap under a memory manager, with

the blocks moved around memory and to and from disk as they're needed; that's certainly self-modifying code, in the sense that the instructions stored at particular memory locations change constantly. Finally, loadable drivers, such as graphics drivers for many windowing environments, are self-modifying code of a sort, since they are loaded as data from the disk into memory by the driver-based program and then executed.

My point is that you shouldn't think of code as immovable and unchangeable. I've found that it's not worth the trouble and risk to modify individual instructions, but in large or complex programs it can be most worthwhile to treat blocks of code as if they were data. The topic is a large one, and this is not the place to explore it, but always keep in mind that even if self- modifying code in its classic sense isn't a great idea on the 8088, the notion that code is just another sort of data is a powerful and perfectly valid concept.

CONCLUSION

Who would have thought that not-branching could offer such variety, to say nothing of such substantial performance improvements? You'll find that not-branching is an excellent exercise for developing your assembler skills, requiring as it does a complete understanding of what your code needs to do, thorough knowledge of the 8088 instruction set, the ability to approach programming problems in non-intuitive ways, knowledge as to when the effort involved in not-branching is justified by the return, and a balancing of relative importance of saving bytes and cycles in a given

application.

In other words, not-branching is a perfect Zen exercise. Practice it often and well!