# Chapter 7: Memory Addressing

The 8088's registers are very powerful, and critically important to writing high-performance code--but there are scarcely a dozen of them, and they certainly can't do the job by themselves. We need more than seven--or seventy, or seven hundred or even seven thousand--general-purpose storage locations. We need storage that's capable of storing characters, numbers, and instruction bytes in great quantities (remember that instruction bytes are just another sort of data)--and, of course, that's just what we get by way of the 1 megabyte of memory that the 8088 supports.

(The PC has only 640 Kb of system RAM, but nonetheless does support a full megabyte of addressable memory. The memory above the 640 K mark is occupied by display memory and by BIOS code stored in ROM (read-only memory); this memory can always be read from and can in some cases--display memory, for example--be written to as well.)

Not only does the 8088 support 1 Mb of memory, but it also provides many powerful and flexible ways to get at that memory. We'll skim through the many memory addressing modes and instructions quickly, but we're not going to spend a great deal of time on their basic operation.

Why not spend more time describing the memory addressing modes and instructions? One reason is that I've assumed throughout The Zen of Assembly Language that you're at least passingly familiar with assembler, thereby avoiding a lot of rehashing and explaining--and memory addressing is fundamental to almost any sort of assembler programming. If you really don't know the basic memory addressing modes, a refresher on assembler in general might be in order before you continue with The Zen of Assembly Language.

The other reason for not spending much time on the operation of the memory addressing modes is that we have another--and sadly neglected--aspect of memory addressing to discuss: performance.

You see, while the 8088 lets you address a great deal of memory, it isn't particularly fast at accessing all that memory. This is especially true when dealing with blocks of memory larger than 64 Kb, but is always true to some extent. Memory-accessing instructions are often very long and are always very slow.

Worse, many people don't seem to understand the sharp distinction between memory and registers. Some "experts" would have you view memory locations as extensions of your register set. With this sort of thinking, the instructions:

```
	mov	dx,ax
```

and:

```
	mov	dx,MemVar
```

are logically equivalent. Well, the instructions are logically equivalent in the sense that they both move data into DX--but they're polar opposites when it comes to performance. The register-only **mov** is half the length in bytes and anywhere from two to seven times faster than the **mov** from memory...and that's fairly typical of the differences between register-only and memory-addressing instructions.

So you see, saying that memory is logically equivalent to registers is something like saying that a bus is logically equivalent to a 747. Sure, you can buy a ticket to get from one place to another with either mode of transportation...but which would _you_ rather cross the country in?

As we'll see in this chapter, and indeed throughout the rest of the _Zen of Assembly Language_, one key to optimizing 8088 code is using the registers heavily while avoiding memory whenever you can. Pick your spots for such optimizations carefully, though. Optimize instructions in tight loops and in time-critical code, but let initialization and set-up code slide; it's just not worth the time and effort to optimize code that doesn't much affect overall performance or response time.

Slow and lengthy as memory accessing instructions are, you're going to end up using them a great deal in your code. (Just try to write a useful program that doesn't access memory!) In light of that, we're going to review the memory-addressing architecture and modes of the 8088, then look at the performance implications of accessing memory. We'll see why memory accesses are slow, and we'll see that not all memory addressing modes or memory addressing instructions are created equal in terms of size and performance. (In truth, the differences between the various memory-addressing modes and instructions are just about as large as those between register-only and memory-accessing instructions.) Along the way, we'll come across a number of useful techniques for writing high-performance code for the PC, most notably look-up tables. By the end of this chapter, we'll be ready to dive into the instruction set in a big way.

We've got a lot of ground to cover, so let's get started.

## DEFINITIONS

I'm going to take a moment to define some terms I'll use in this chapter. These terms will be used to describe operands to various instructions; for example, **mov ax,segreg** refers to copying the contents of a segment register into AX.

reg refers to any 8- or 16-bit general-purpose register. reg8 refers to any 8-bit (byte-sized) general-purpose register, and reg16 refers to any 16-bit (word-sized) general-purpose register.

segreg refers to any segment register.

mem refers to any 8-, 16-, or 32-bit memory operand. mem8 refers to any byte-sized memory operand, mem16 refers to any word-sized memory operand, and mem32 refers to any doublewordsized memory operand.

reg/mem refers to any 8- or 16-bit register or memory operand. As you'd expect, reg/mem8 refers to any byte-sized register or memory operand, and reg/mem16 refers to any word- sized register or memory operand.

immed refers to any immediate (constant) instruction operand. (Immediate addressing is discussed in detail below.) immed8 refers to any byte-sized immediate operand, and immed16 refers to any word-sized immediate operand.

## SQUARE BRACKETS MEAN MEMORY ADDRESSING

The use of square brackets is optional when a memory location is being addressed by name. That is, the two following instructions assemble to exactly the same code:

```
	mov	dx,MemVar
	mov	dx,[MemVar]
```

However, addressing memory without square brackets is an extension of the "memory and registers are logically equivalent" mindset. I strongly recommend that you use square brackets on all memory references in order to keep the distinction between memory and registers clear in your mind. This practice also helps distinguish between immediate and memory operands.

**THE MEMORY ARCHITECTURE OF THE 8088** The ability to address 1 Mb of memory, while unimpressive by today's standards, was quite remarkable when the PC was first introduced, 64 Kb then being standard for "serious" microcomputers. In fact, an argument could be made that the

8088's 1 Mb address space is the single factor most responsible for the success of the IBM PC and for the exceptional software that quickly became available for it. Realistically, the letters "IBM" were probably more important, but all that memory didn't hurt; quantities of memory make new sorts of software possible, and can often compensate for limited processor power in the form of lookup tables, RAM disks, data caching, and in-line code. All in all, the PC's then-large memory capacity made possible a quantum leap in software quality.

On the other hand, the 8088 actually addresses all that memory in what is perhaps the most awkward manner ever conceived--by way of addressing 64 Kb blocks off each of the four segment registers. This scheme means that programs must perform complex and time-consuming calculations in order to access the full 1 Mb of memory in a general way. One of the ways in which assembler programs can outstrip compiled programs is by cleverly structuring code and data so that sequential memory accesses generally involve only memory within the four segments addressable at any one time, thereby avoiding the considerable overhead associated with calculating full addresses and frequently reloading the segment registers.

In short, the 8088's memory architecture is the best of worlds and the worst of worlds: the best because a great deal of memory is addressable (at least by 1981 standards), the worst because it's hard to access all that memory quickly. That said, let's look at the 8088's memory architecture in detail. Most likely you know what we're about to discuss, but bear with me; I want to make sure we're all speaking the same language before I go on to more advanced subjects.

**SEGMENTS AND OFFSETS**

20 bits are needed to address 1 Mb of memory, and every one of the one-million-plus memory addresses the 8088 can handle can indeed be expressed as a 20-bit number. However, programs do not address memory with 20-bit addresses. There's a good reason for that: 20-bit addresses would be most impractical. For one thing, the 8088's registers are only 16 bits in size, so they couldn't be used to point to 20-bit addresses. For another, three rather than two bytes would be needed to store each address loaded by a program, making for bloated code. In general, the 8088 just wasn't designed to handle straight 20-bit addresses.

(You may well ask why the 8088 wasn't designed better. "Better" is a slippery term, and the 8088 certainly has been successful...nonetheless, that's a good question, which I'll answer in Chapter 8. A hint: much of the 8088's architecture is derived from the 8080, which could only address 64 Kb in all. The 8088 strongly reflects long-ago microcomputer technology, not least in its limitation to 1 Mb in total.) Well, if the PC doesn't use straight 20-bit addresses, what does it use? It uses paired segments and offsets, which together form an address denoted as segment:offset. For example, the address 23F0:1512 is the address composed of the segment value 23F0 hex and the offset value 1512 hex. (I'll always show segment:offset pairs in hexadecimal, which is by far the easiest numbering scheme for memory addressing.) Both segments and offsets are 16-bit values.

Wait one minute! We're just looking for 20-bit addresses, not 32-bit addresses. Why do we need 16 bits of segment and 16 bits of offset?

Actually, we <u>don't</u> need 16 bits of segment. We could manage to address 1 Mb perfectly well with a mere 4 bits of segment, but that's not the way Intel set up the segment:offset addressing scheme. I might add that there's some justification for using segments and offsets. The segment:offset approach is a reasonable compromise between the needs to use memory efficiently and keep chip costs down that predominated in the late 1970s and the need to use an architecture that could stretch to accommodate the far more sophisticated memory demands of the 8088's successors. The 80286 uses an extension of the segment:offset approach to address 16 Mb of memory in a fully protected multitasking environment, and the 80386 goes far beyond that, as we'll see in Chapter 15.

Anyway, although we only need 4 bits of segment, we get 16 bits, and none of them are ignored by the 8088. 20-bit addresses are formed from segment:offset pairs by shifting the segment 4 bits to the left and adding it to the offset, as shown in Figure 7-1.

I'd like to take a moment to note that for the remainder of this book, I'll use light lines to signify memory addressing in figures and heavy lines to show data movement, as illustrated by Figure 7-1. In the future, I'll show segment:offset memory addressing by simply joining the lines from the segment register and any registers and/or displacements (fixed values) used to generate an offset, as in Figure 7-7, avoiding the shift-and-add complications of Figure 7-1a; the 4-bit left shift of the segment and the addition to the offset to generate a 20-bit memory address, which occurs whenever a segment:offset address is used, is implied. Also, when the segment isn't germane to the discussion at hand, I may omit it and show only the offset component or components, as in Figure 7-4; although unseen, the segment is implied, since one segment register must participate in forming virtually every 20-bit memory address, as we'll see shortly.

Figure 7-1 also illustrates another practice I'll follow in figures that involve memory addressing: the shading of registers and memory locations that change value. This makes it easy to spot the effects of various operations. In Figure 7-1, only the contents of AL are altered; consequently, only AL is shaded.

I'll generally follow the sequence of Figure 7-1--memory address, memory access, final state of the PC--in memory addressing figures. While this detailed, step-by-step approach may seem like a bit of overkill right now, it will be most useful for illustrating the 8088's more complex instructions, particularly the string instructions.

Finally, the numbers in Figure 7-1--including both addresses and data--are in hexadecimal. Numbers in all figures involving memory addressing will be in hexadecimal unless otherwise noted.

To continue with our discussion of segment:offset addressing, shifting a segment value left 4 bits is equivalent to shifting it left 1 hexadecimal digit--one reason that hexadecimal is a useful notation for memory addresses. Put another way, if the segment is the hexadecimal value ssss and the offset is the hexadecimal value xxxx, then the 20-bit memory address mmmmm is calculated as follows:

```
                ssss0
+   xxxx
-------
=               mmmmm
```

For example, the 20-bit memory address corresponding to 23F0:1512 is 25412 (hex) arrived at as follows:

```
                23F00
+   1512
-------
=               25412
```

By the way, it happens that the 8088 isn't particularly fast at calculating 20-bit addresses from segment:offset pairs. Although it only takes the 8088's Bus Interface Unit 4 cycles to complete a memory access, the fastest memory-accessing instruction the PC has to offer (**xlat**) takes 10 cycles to run. Other memory-accessing instructions take longer, some much longer. We'll delve into the implications of the 8088's lack of memory-access performance shortly.

Several questions should immediately leap into your mind if you've never encountered segments and offsets before. Where do these odd beasts live? What's to prevent more than one segment:offset pair from pointing to the same 20-bit address? What happens when the sum of the two gets too large to fit in 20 bits?

To answer the first question first, segment values reside in the four segment registers: CS, DS, ES, and SS. One (and only one) of these four registers participates in calculating the address for almost every single memory access the PC makes. (Interrupts are exceptions to this rule, since interrupt vectors are read from fixed locations in the first 1 Kb of memory.) Segments are, practically speaking, part of every memory access your code will ever make.

CS is always used for code addresses, such as addresses involved in instruction fetching and branching. DS is usually used for accessing memory operands; most instructions can use any segment to access memory operands, but DS is generally the most efficient register for data access. SS is used for maintaining the stack, and is used to access data in stack frames. Finally, ES is used to access data anywhere in the 8088's address space; since it's not dedicated to any other purpose, it's useful for pointing to rarely-used segments. ES is particular useful in conjunction with the string instructions, as we'll see in Chapter 10. In Chapter 6 we discussed exactly what sort of memory accesses operate relative to each segment register by default; we'll continue that discussion later in this chapter, and look at ways to override the default segment selections in some cases.

Offsets are not so simple as segments. The 8088 can calculate offsets in a number of different ways, depending on the addressing mode being used. Both registers and instructions can contain offsets, and registers and/or constant values can be added together on the fly by the 8088 in order to calculate offsets. In various addressing modes, components of offsets may reside in BX, BP, SI, DI, SP, and AL, and offset components can be built into instructions as well.

We'll discuss the loading and use of the segment registers and the calculation and use of offsets below. First, though, let's answer our two remaining questions.

## SEGMENT:OFFSET PAIRS AREN'T UNIQUE

In answer to question number two, "What's to prevent more than one segment:offset pair from pointing to the same 20-bit address?" the answer is: nothing. There's no rule that says two segment:offset pairs can't point to the same address, and in fact many segment:offset pairs do evaluate to any given address--4096 segment:offset pairs for every address, to be precise. For example, the following segment:offset pairs all point to the 20- bit address 00410: 0000:0410, 0001:0400, 0002:03F0, 0003:03E0, and so on up to 0041:0000.

You may have noticed that we've only accounted for 42h segment:offset pairs, not 4096 of them, and that leads in neatly to the answer to our third and final question. When the sum of a segment shifted left 4 bits and an offset exceeds 20 bits, it wraps back around to address 00000. Basically, any bits that carry out of bit 19 (into what would be bit 20 if the 8088 had 21 addressing bits) are thrown away. The segment:offset pair FFFF:0010 points to the address 00000 as follows:

```
              FFFF0
  +   0010
  -------
   100000
   ^
 carry
```

with the 1 that carries out of bit 19 discarded to leave 00000.

Now we can see what the other 4,000-odd segment:offset pairs that point to address 00410 are. FFFF:0420 points to 00410, as do FFFE:0430, F042:FFF0, and a host of segment:offset pairs in between. I doubt you'll want to take advantage of that knowledge (in fact, there is a user-selectable trick that can be played on the 80286 and 80386 to disable wrapping at FFFFF, so you shouldn't count on wrapping if you can help it), but if you do ever happen to address past the end of memory, that's how it works on the 8088.

## GOOD NEWS AND BAD NEWS

Now that we know how segments and offsets work, what are the implications for assembler programs? The obvious implication is that we can address 1 Mb of memory, and that's good news, since we can use memory in myriad ways to improve performance. For example, we'll see how look-up tables can turn extra memory into improved performance later in this chapter. Likewise, in Chapter 13 we'll see how in-line code lets you trade off bytes for performance. Much of top-notch assembler programming involves balancing memory requirements against performance, so the more memory we have available, the merrier.

The bad news is this: while there's a lot of memory, it's only available in 64 Kb chunks. The four segment registers can only point to four 64 Kb segments at any one time, as shown in Figure 7-2. If you want to access a memory location that's not in any of the four currently pointed-to segments, there is no way to do that with a single instruction. You must first load a segment register to point to a segment containing the desired memory location, a process which takes a minimum of 1 and often 2 instructions. Only then can you access the desired memory location.

Worse, there are problems dealing with blocks of memory larger than 64 Kb, because there's no easy way to perform calculations involving full 20-bit addresses, and because 64 Kb is the largest block of memory that can be addressed by way of a single segment register without reloading the segment register. It's easy enough to access a block up to 64 Kb in size; point a register to the start of the block, and then point wherever you wish. For example, the following bit of code would calculate the 16-bit sum of all the bytes in a 64 Kb array:

```
            mov     bx,seg TestArray
            mov     ds,bx               ;point to segment:offset of start of
            mov     bx,offset TestArray ;array to sum
            sub     cx,cx               ;count 64 K bytes
            mov     ax,cx               ;set initial sum to 0
            mov     dh,ah               ;set DH to 0 for summing later
SumLoop:
            mov     dl,[bx]             ;get the next array element
            add     ax,dx               ;add the array element to the sum
            inc     bx                          ;point to the next array element
            loop    SumLoop
```

Easy enough, eh? Ah, but it all falls apart when a block of memory is larger than 64 Kb, or when a block crosses a segment boundary. The problem is that in either of those cases the segment must change as well as the offset, for there's simply no way for an offset to reach more than 64 K bytes away from any given segment register setting. If a register containing an offset reaches the end of a segment (reaches the value 0FFFFh), then it simply wraps back to zero when it's incremented. Likewise, the instruction sequence:

```
mov     si,0ffffh
mov     al,[si+1]
```

merely manages to load AL with the contents of offset 0. Basically, whenever an offset exceeds 16 bits in size, the excess bits are ignored, just as the excess bits are ignored when a segment:offset pair adds up to an address past the 1 Mb overall limit on 8088 memory.

So we need to work with the whole segment:offset pair in order to handle blocks larger than 64 Kb. Is that such a problem? Unfortunately, the answer is yes. The 8088 has no particular aptitude for calculations involving more than 16 bits, and is very bad at handling segments. There's no way to increment a segment:offset pair as a unit, and in fact there's no way to modify a segment register other than copying it to a general-purpose register, modifying that register, and copying the result back to the segment register. All in all, it's as difficult to work with blocks of memory larger than 64 Kb as it is easy to work with blocks no larger than 64 Kb.

For example, here's typical code to calculate the 16-bit sum of a 128 Kb array, of the sort that a high-level language might generate (actually, the following code is a good deal better than most high-level languages would generate, but what the heck, let's give them the benefit of the doubt!):

```
          mov     bx,seg TestArray
          mov     ds,bx               ;point to segment:offset of start of
          mov     bx,offset TestArray ;array to sum
          sub     cx,cx               ;count 128 K bytes with SI:CX
          mov     si,2
          mov     ax,cx               ;set initial sum to 0
          mov     dh,ah               ;set DH to 0 for summing later
SumLoop:
          mov     dl,[bx]             ;get the next array element
          add     ax,dx               ;add the array element to the sum
          inc     bx                          ;point to the next array element
          and     bx,0fh              ;time to advance the segment?
          jnz     SumLoopEnd          ;not yet
          mov     di,ds               ;advance the segment by 1; since BX has
          inc     di                          ; just gone from 15 to 0, we've advanced
          mov     ds,di               ; 1 byte in all
SumLoopEnd:
          loop    SumLoop             ;count down 32-bit counter
          dec     si
          jnz     SumLoop
```

## MORE GOOD NEWS

While the above is undeniably a mess, things are not quite so grim as they might seem. In fact, the news is quite good when it comes to handling multiple segments in assembler. For one thing, assembler is much better than other languages at handling segments efficiently. Only in assembler do you have complete control over all your segments; that means that you can switch the segments as needed in order to make sure that they are pointing to the data you're currently interested in. What's more, in assembler you can structure your code and data so that it falls naturally into 64 Kb blocks, allowing most of your accesses at any one time to fall within the currently loaded segments.

In high-level languages you almost always suffer both considerable performance loss and significant increase in code size when you start using multiple code or data segments, but in assembler it's possible to maintain near-peak performance even with many segments. In fact, segment-handling is one area in which assembler truly distinguishes itself, and we'll see examples of assembler's fine touch with segments in this chapter, Chapter 14, and Volume II of The Zen of Assembly Language.

There's one more reason that handling multiple code or data segments isn't much of a problem in assembler, and that's that the assembler programmer knows exactly what his code needs to do and can optimize accordingly. For example, suppose that we know that the array **TestArray** in the last example is guaranteed to start at offset 0 in the initial data segment. Given that extra knowledge, we can put together the following version of the above code to sum a 128 Kb array:

```
        mov     bx,seg TestArray
        mov     ds,bx           ;point to segment:offset of start of
        sub     bx,bx           ;array to sum, which we know starts
                                ; at offset 0
        mov     cx,2                    ;count two 64 Kb blocks
        sub     ax,ax           ;set initial sum to 0
        mov     dh,ah           ;set DH to 0 for summing later
SumLoop:
        mov     dl,[bx]         ;get the next array element
        add     ax,dx           ;add the array element to the sum
        inc     bx                      ;point to the next array element
        jnz     SumLoop         ;until we wrap at the end of a 64 Kb block
        mov     si,ds
        add     si,1000h        ;advance the segment by 64 K bytes
        mov     ds,si
        loop    SumLoop         ;count off this 64 Kb block
```

Compare the code within the inner loop above to that in the inner loop of the previous version of this example--the difference is striking. This inner loop is every bit as tight as that of the code for handling blocks 64 Kb-and-less in size; in fact, it's slightly tighter, as **jnz** is faster than **loop**. Consequently, there shouldn't be much difference in performance between the last example and the 64 Kb and less version. Nonetheless, a basic rule of the Zen of assembler is that we should check our assumptions, so let's toss the three approaches to summing arrays into the Zen timer and see what comes out.

Listing 7-1 measures the time required to calculate the 16- bit sum of a 64 Kb block without worrying about segments. This code runs in 619 ms, or 9.4 us per byte summed. (Note that Listings 7-1 through 7-3 must be timed with the long-period Zen timer--via LZTIME.BAT--since they take more than 54 ms to run.)

Listing 7-2 measures the time required to calculate the 16- bit sum of a 128 Kb block. As is always the case with a memory block larger than 64 Kb, segments must be dealt with, and that shows in the performance of Listing 7-2: 2044 ms, or 15.6 us per byte summed. In other words, Listing 7-1, which doesn't concern itself with segments, sums bytes 66% faster than Listing 7-2.

Finally, Listing 7-3 implements 128 Kb-block-handling code that takes advantage of the knowledge that the block of memory being summed starts at offset 0 in the initial data segment. We've speculated that Listing 7-3 should perform on a par with Listing 7-3, since their inner loops are similar...and the Zen timer bears that out, reporting that Listing 7-3 runs in

1239 ms--9.5 us per byte summed.

Assumptions confirmed.

**NOTES ON OPTIMIZATION**

There are several points to be made about Listings 7-1 through 7-3. First, these listings graphically illustrate that you should focus your optimization efforts on inner loops. Listing 7-3 is considerably bigger and more complex than Listing 7-1, but by moving the complexity and extra bytes out of the inner loop, we've managed to keep performance high in Listing 7- 3.

Now, you may well object that in the process of improving the performance of Listing 7-3, we've altered the code so that it will only work under certain circumstances, and that's my second point. Truly general-purpose code runs slowly, no matter whether it's written in assembler, C, BASIC, or COBOL. Your advantage as a programmer--and your great advantage as an assembler programmer--is that you know exactly what your code needs to do...so why write code that wastes cycles and bytes doing extra work? I stipulated that the start offset was at 0 in the initial data segment, and Listing 7-3 is a response to that stipulation. If the conditions to be met had been different, then we would have come up with a different solution.

Do you see what I'm driving at? I hope so, for it's central to the Zen of assembler. A key to good assembler code is to write lean code. Your code should do everything you need done-- and nothing more.

I'll finish up by pointing out that Listings 7-1 through 7-3 are excellent examples of both the hazards of using memory blocks larger than 64 Kb and of the virtues of using assembler when you must deal with large blocks. It's rare that you'll be able to handle larger-than-64 Kb blocks as efficiently as blocks that fit within a single segment; Listing 7-3 does take advantage of a very convenient special case. However, it's equally rare that you won't be able to handle large blocks much more efficiently in assembler than you ever could in a high-level language.

## A FINAL WORD ON SEGMENT:OFFSET ADDRESSING

Let's review what we've learned about segment:offset addressing and assembler. The architecture of the 8088 limits us to addressing at most four segments--64 Kb blocks of memory--at any time, with each segment pointed to by a different segment register. Accessing data in a segment that is not currently pointed to by any segment register is a time-consuming, awkward process, as is handling data that spans multiple blocks. Fortunately, assembler is adept at handling segments, and gives us considerable freedom to structure our programs so that we're usually working within the currently loaded segments at any one time.

On balance, segment:offset addressing is one of the less attractive features of the 8088. For us, however, it's actually an advantage, since it allows assembler, with its superb control over the 8088, to far outstrip high-level languages. We won't deal with segments a great deal in the remainder of this volume, since we'll be focusing on detailed optimizations, but the topic will come up from time to time. In Volume II, we'll tackle the subject of segment management in a big way.

The remainder of this chapter will deal only with data addressing--that is, the addressing of instruction operands. Code addressing--in the forms of instruction fetching and branching--is a very real part of PC performance (heck, instruction fetching is perhaps the single most important performance factor of all!), but it's also very different from the sort of memory addressing we'll be discussing. We learned as much as we'll ever need to know (and possibly more) about instruction fetching back in Chapters 4 and 5, so we won't pursue that aspect of code addressing any further. However, Chapters 12 through 14 discuss code addressing as it relates to branching in considerable detail.

## SEGMENT HANDLING

Now that we know what segments are, let's look at ways to handle the segment registers, in particular how to load them quickly. What we are not going to do is discuss the directives that let you create segments and the storage locations within them.

Why not discuss the segment directives? For one thing, there are enough directives, segment and otherwise, to fill a book by themselves. For another thing, there are already several such books, including both the manuals that come with MASM and TASM and the other books in this series. The Zen of Assembly Language is about writing efficient code, not using MASM, so I'll assume you already know how to use the **segment**, **ends**, and

**assume** directives to define segments and **db**, **dw**, and the like to create and reserve storage. If that's not the case, brush up before you continue reading. We'll use all of the above directives in The Zen of Assembly Language, and we'll discuss **assume** at some length later in this chapter, but we won't spend time covering the basic functionality of the segment and data directives.

## WHAT CAN YOU DO WITH SEGMENT REGISTERS? NOT MUCH

Segment registers are by no means as flexible as general-purpose registers. What can't you do with segment registers that you can do with general-purpose registers? Let me answer that question by way of a story.

There's a peculiar sort of "find the mistake" puzzle that's standard fare in children's magazines. Such puzzles typically consist of a drawing with a few intentional mistakes (a farmer milking a donkey, for example--a risky proposition at best), captioned, "What's wrong with this picture?" Invariably, the answer is printed upside down at the bottom of the page.

I dimly recall from my childhood a takeoff that MAD magazine did on those puzzles. MAD showed a picture in which everything-and I do mean everything--was wrong. Just as with the real McCoy, this picture was accompanied by the caption, "What's wrong with this picture?", and by the answer at the bottom of the page.

In MAD, the answer was: "Better yet, what's right with this picture?"

Segment registers are sort of like MAD's puzzles. What can't you do with segment registers? Better yet, what can you do with segment registers? Well, you can use them to address memory--and that's about it.

Any segment register can be copied to a general-purpose register or memory location. Any segment register other than CS can be loaded from a general-purpose register or memory location. Any segment register can be pushed onto the stack, and any segment register but CS can be popped from the stack.

And that's all.

Segment registers can't be used for arithmetic. They can't be operands to logical instructions, and they can't take part in comparisons. One segment register can't even be copied directly to another segment register. Basically, segment registers can't do a blessed thing except get loaded and get copied to a register or memory.

Now, there are reasons why segments are so hard to work with. For one thing, it's not all that important that segment registers be manipulated quickly. Segment registers aren't changed as often as general-purpose registers--at least, they shouldn't be, if you're interested in decent performance. Segment registers rarely need to be manipulated arithmetically or logically, and when the need does arise, they can always be copied to general-purpose registers and manipulated there. Nonetheless, greater flexibility in handling segment registers would be nice; however, a major expansion of the 8088's instruction set--requiring additional circuitry inside the 8088-- would have been required in order to allow us to handle segment registers like general-purpose registers, and it seems likely that the 8088's designers had other, higher-priority uses for their limited chip space.

There's another reason why segments can only be loaded and copied, nothing else, and it has to do with the protected mode of the 80286 and 80386 processors. Protected mode, which we'll return to at a bit more length in Chapter 15, is a second mode of the 80286 and 80386 that's not compatible with either MS-DOS or the 8088, but which makes much more memory available for program use than the familiar 1 Mb of MS-DOS/8088-compatible real mode.

In protected mode, the segment registers don't contain memory addresses; instead, they contain segment selectors, which the 80286 and 80386 use to look up the actual segment information--location and attributes such as writability--in a table. Not only would it make no sense to perform arithmetic and the like on segment selectors, since selectors don't correspond directly to memory addresses, but because the segment registers are central to the memory protection scheme of the 80286 and 80386, they simply <u>cannot</u> be loaded arbitrarily--the 80286 and 80386 literally don't allow that to happen by instantly causing a trap whenever an invalid selector is loaded.

What's more, it can take quite a while to load a segment register in protected mode. In real mode, moves to and from segment registers are just as fast as transfers involving general-purpose registers, but that's not the case in protected mode. For example, **mov es,ax** takes 2 cycles in real mode and 17 cycles in protected mode.

Given all of the above, all you'd generally want to do in protected mode is load the segment registers with known-good segment selectors provided to you by the operating system. That doesn't affect real mode, which is all we care about, but since real mode and protected mode share most instructions, the segment-register philosophy of protected mode (which Intel no doubt had as a long-range goal even before they designed the 8088) carries over to real mode.

And now you know why the 8088 offers so little in the way of segment-register manipulation capability.

## USING SEGMENT REGISTERS FOR TEMPORARY STORAGE

That brings us to another interesting point: the use of segment registers for temporary storage. The 8088 has just 7 available general-purpose registers (remember, we can't use SP for anything but the stack most of the time), and sometimes it would be awfully handy to have somewhere to store a 16-bit value for a little while. Can we use the segment registers for that purpose?

Some people would answer that "No," because code that uses segments for temporary storage can't easily be ported to protected mode. I don't buy that, for reasons I'll explain when we get to **les**. My answer is "Yes...when they're available." Two of the segment registers are never available, one is occasionally available, and one may or may not be readily available, depending on your code.

Some segments are always in use. CS is always busy pointing to the segment of the next instruction to be executed; if you were to load CS with an arbitrary value for even 1 instruction, your program would surely crash. Clearly, it's not a good idea to use CS for temporary storage. (Actually, this isn't even a potential problem, as Intel has thoughtfully not implemented the instructions--**mov** and **pop**--that might load CS directly; MASM will simply generate an error if you try to assemble **pop cs** or **mov cs,[mem16]**. CS can only be loaded by far branches: far calls, far returns, far jumps, and interrupts.)

SS isn't in use during every cycle as CS is, but unless interrupts are off, SS might be used on any cycle. Even if interrupts are off, non-maskable interrupts can occur, and of course your code will often use the stack directly. The risks are too great, the rewards too few. Don't use SS for temporary storage.

DS can be used for temporary storage whenever it's free. However, DS is usually used to point to the default data segment. It's rare that you'll have a tight loop in which memory isn't accessed (it's not worth bothering with such optimizations outside the tightest, most time-critical code), and memory is usually most efficiently accessed via DS. There certainly are loops in which DS is free--loops which use **scas** to scan the segment pointed to by ES, for example--but such cases are few and far between. Far more common is the case in which DS is saved and then pointed to another segment, as follows:

```
	push	ds				;preserve normal DS setting
```

```
                mov     bx,seg TestArray
                mov     ds,bx                   ;point DS:BX to array in which
                mov     bx,offset TestArray ; to flip all bits
                mov     cx,TEST_ARRAY_LENGTH ;# of bytes to flip
        FlipLoop:
                not     byte ptr [bx]           ;flip all bits in current byte
                inc     bx                              ;point to next byte
                loop    FlipLoop
                pop     ds                              ;restore normal DS setting
```

This approach allows instructions within the loop to access memory without the segment override prefix required when ES is used. (More on segment override prefixes shortly.)

In short, feel free to use DS for temporary storage if it's free, but don't expect that to come up too often.

Which brings us to the use of ES for temporary storage. ES is by far the best segment register to use for temporary storage; not being dedicated to any full-time function, it's usually free for any sort of use at all, including temporary storage.

Let's look at an example of code that uses ES for temporary storage to good effect. This sample code sums selected points in a two-dimensional word-sized array. Let's start by tallying up the registers this code will use. (A bit backwards, true, but we're focusing on the use of ES for temporary storage at the moment, and this is the best way to go about it.)

In the sample code, the list of subscripts of points to be added in the major dimension will be stored at DI, and the list of subscripts in the minor dimension will stored at BX. CX will contain the number of points to be summed, and BP will contain the final sum. AX and DX will be used for multiplying, and, as usual, SP will be used to point to the stack. Finally, when the code begins, SI will contain the offset of the start of the array.

Let's see...that covers all eight general-purpose registers. Unfortunately, we need yet another storage location, this one to serve as a working pointer into the array. There are many possible solutions to this problem, including using the **xchg** instruction (which we'll cover in the next chapter), storing values in memory (slow), pushing and popping SI (also slow), or disabling interrupts and using SP (can unduly delay interrupts and carries some risk). Instead, here's a solution that uses ES for temporary storage; it's not necessarily the best solution, but it does nicely illustrate the use of ES for temporary storage:

```
;
; Sums selected points in a two-dimensional array.
;
; Input:
;               BX = list of minor dimension coordinates to sum
;               CX = number of points to sum
;               DS:SI = start address of array
;               DI = list of major dimension coordinates to sum
;
; Output:
;               BP = sum of selected points
;
; Registers altered: AX, BX, CX, DX, SI, DI, BP, ES
;
                mov     es,si           ;set aside the array start offset
                sub     bp,bp           ;initialize sum to 0
TwoDimArraySumLoop:
                mov     ax,ARRAY_WIDTH  ;convert the next major dimension
                mul     word ptr [di]   ;coordinate to an offset in the array
                                        ; (wipes out DX)
                add     ax,[bx]         ;add in the minor dimension coordinate
                shl     ax,1                    ;make it a word-sized lookup
                mov     si,es           ;point to the start of the array
                add     si,ax           ;point to the desired data point
                add     bp,[si]         ;add it to the total
                inc     di                      ;point to the next major dimension coordinate
                inc     di
                inc     bx                      ;point to the next minor dimension coordinate
                inc     bx
                loop    TwoDimArraySumLoop
```

If you find yourself running out of registers in a tight loop and you're not using the segment pointed to by ES, by all means reload one of your registers from ES if that will help.

## SETTING AND COPYING SEGMENT REGISTERS

As I've said, loading segment registers is one area in which assembler has a tremendous advantage over high-level languages. High-level languages tend to use DS to point to a default data segment all the time, loading ES every single time any other segment is accessed. In assembler, we can either load a new segment into DS as needed, or we can load ES and leave it loaded for as long as we need to access a given segment.

We'll see examples of efficient segment use throughout The Zen of Assembly Language, especially when we discuss strings, so I'm not going to go into more detail here. What I am going to do is discuss the process of loading segment registers, because it is by no means obvious what the most efficient segment-loading mechanism is.

For starters, let's divide segment loading into two categories: setting and copying. Segment setting refers to loading a segment register to point to a certain segment, while segment copying refers to loading a segment register with the contents of another segment register. I'm making this distinction because the instruction sequences used for the two sorts of segment loading differ considerably.

Let's tackle segment copying first. Segment copying is useful when you want two segment registers to point to the same segment. For example, you'll want ES to point to the same segment as DS if you're using

**rep movs** to copy data within the segment pointed to by DS, because DS and ES are the default source and destination segments, respectively, for **movs**. There are two good ways to load ES to point to the same segment as DS, given that we can't copy one segment register directly to another segment register:

```
push	ds
pop	es
```

and:

```
mov	ax,ds
mov	es,ax
```

(Any general-purpose register would serve as well as AX.)

Each of the above approaches has its virtues. The **push**/**pop** approach is extremely compact, at just 2 bytes, and affects no other registers. Unfortunately, it takes a less-than-snappy 27 cycles to run. By contrast, the **mov**/**mov** approach officially takes just 4 cycles to run; 16 cycles (4 bytes at 4 cycles to fetch each byte) is a more realistic figure, but either way, **mov**/**mov** is clearly faster than **push**/**pop**. On the other hand, **mov**/**mov** takes twice as many bytes as **push**/**pop**, and destroys the contents of a general-purpose register as well.

There's no clear winner here. Use the **mov**/**mov** approach to copy segment registers when you're interested in speed and can spare a general-purpose register, and use the **push**/**pop** approach when bytes and/or registers are at a premium. I'll use both approaches in this book, generally using **push**/**pop** in non-time- critical code and **mov**/**mov** when speed really counts. Why waste the bytes when the cycles don't matter?

That brings us to an important point about assembler programming. There is rarely such a beast as the "best code" in assembler; instead, there's code that's good in a given context. In any situation, the choice between fast code, small code, understandable code, portable code, maintainable code, structured code, and whatever other sort of code you can dream up is purely up to you. If you make the right decisions, your code will beat high-level language code hands down, because you know more about your code and can think far more flexibly than any high-level language possibly can.

Now let's look at ways to set segment registers. Segment registers can't be loaded directly with a segment value, but they can be loaded either through a general-purpose register or from memory. The two approaches aren't always interchangeable: one requires that the segment name be available as an immediate operand, while the other requires that a memory variable be set to the desired segment value. Nonetheless, you can generally set things up so that either approach can be used, if you really want to--so which is best?

Well, loading a segment register through a general-purpose register, as in:

```
mov     ax,DATA
```

```
        mov     es,ax
```

officially takes 6 cycles. Since the two instructions together are 5 bytes long, however, this approach could take as much a 20 cycles if the prefetch queue is empty. By contrast, loading from memory, as in:

```
        mov     es,[DataSeg]
```

officially takes only 18 cycles, is only 4 bytes long, and doesn't destroy a general-purpose register. (Note that the last approach assumes that the memory variable **DataSeg** has previously been set to point to the desired segment.) Loading from memory sounds better, doesn't it?

It isn't.

Remember, it's not just the number of instruction byte fetches that affects performance--it's the number of memory accesses of all sorts. When a segment register is loaded from memory, 2 memory accesses are performed to read the segment value; together with the 4 instruction bytes, that means that 6 memory accesses in all are performed when a segment register is loaded from memory. What that means is that loading a segment register from memory takes anywhere from 18 to 24 (6 memory accesses at 4 cycles per access) cycles, which stacks up poorly against the 6 to 20 cycles required to load a segment register through a general-purpose register.

In short, it's clearly fastest to load segment registers through general-purpose registers.

That's not to say that there aren't times when you'll want to load a segment register directly from memory. If you're <u>really</u> tight on space, you can save a byte every time you load a segment by using the 4-byte load from memory rather than the 5- byte load through a general-purpose register. (This is only worthwhile if there are multiple segment load instructions, since the memory variable containing the segment address takes 2 bytes.) Also, if the segment you want to work with varies as your program runs (for example, if your code can access either display memory or a display buffer in system RAM), then loading the segment register from memory is the way to go. The following code is clearly the best way to load ES to point to a display buffer that may be at any of several segments:

```
mov     es,[DisplayBufferSegment]
```

Here, **DisplayBufferSegment** is set externally to point to the segment in which all screen drawing should be performed at any given time.

Finally, segments are often passed as stack frame parameters from high-level languages to assembler subroutines--to point to far data buffers and the like--and in those cases segments can best be loaded directly from stack frames into segment registers. (We'll discuss stack frames later in this chapter.) It's easy to forget that segments can be loaded directly from <u>any</u> addressable memory location, as we'll see in Chapter 16; all too many people load segments from stack frames like this:

```
mov     ax,[bp+BufferSegment]
mov     es,ax
```

when the following is shorter, faster, and doesn't use any general-purpose registers:

```
mov     es,[bp+BufferSegment]
```

As it happens, though, lone segment values are rarely passed as stack frame parameters. Instead, segment:offset pairs that provide a full 20-bit pointer to a specific data element are usually passed. These can be loaded as follows:

```
mov     es,[bp+BufferSegment]
mov     di,[bp+BufferOffset]
```

However, the designers of the 8088 anticipated the need for loading 20-bit pointers, and gave us two most useful instructions for just that purpose: **lds** and **les**.

## LOADING 20-BIT POINTERS WITH lds AND les

**lds** loads both DS and any one general-purpose register from a doubleword of memory, and **les** similarly loads both ES and a general-purpose register, as shown in Figure 7-3.

While both instructions are useful, **les** is by far the more commonly used of the two. Since most programs leave DS pointing to the default data segment whenever possible, it's rare that we'd want to load DS as part of a segment:offset pointer. True, it does happen, but generally only when we want to point to a block of far memory temporarily for faster processing in a tight loop.

ES, on the other hand, is the segment of choice when a segment:offset pointer is needed, since it's not generally reserved for any other purpose. Consequently, **les** is usually used to load segment:offset pointers.

**lds** and **les** actually don't come in for all that much use in pure assembler programs. The reason for that is that efficient assembler programs tend to be organized so that segments rarely need to be changed, and so such programs tend to work with 16-bit pointers most of the time. After all, while **lds** and **les** are efficient considering all they do, they're still slow, with official execution times of at least 29 cycles. If you need to load segment:offset pointers, use **lds** and **les**, but try to load just offsets whenever you can.

One place where there's no way to avoid loading segments is in assembler code that's called from a high-level language, especially when the large data model (the model that supports more than 64 Kb of data) is used. When a high-level language passes a far pointer as a parameter to an assembler subroutine, the full 20-bit pointer must be loaded from memory before it can be used, and there **lds** and **les** work beautifully.

Suppose that we have a C statement that calls the assembler subroutine **AddTwoFarInts** as follows:

```
int Sum;
int far *FarPtr1, far *FarPtr2;
 :
Sum = AddTwoFarInts(FarPtr1, FarPtr2);
```

**AddTwoFarInts** could be written without **les** as follows:

```
Parms           struc
                        dw      ?       ;pushed BP
                        dw      ?       ;return address
Ptr1Offset      dw      ?
Ptr1Segment     dw      ?
Ptr2Offset      dw      ?
Ptr2Segment     dw      ?
Parms           ends
;
AddTwoFarInts   proc    near
                push    bp                      ;save caller's BP
                mov     bp,sp                   ;point to stack frame
                mov     es,[Ptr1Segment]        ;load segment part of Ptr1
                mov     bx,[Ptr1Offset]         ;load offset part of Ptr1
                mov     ax,es:[bx]              ;get first int to add
                mov     es,[Ptr2Segment]        ;load segment part of Ptr2
                mov     bx,[Ptr2Offset]         ;load offset part of Ptr2
                add     ax,es:[bx]              ;add the two ints together
                pop     bp                      ;restore caller's BP
                ret
AddTwoFarInts   endp
```

The subroutine is considerably more efficient when **les** is used, however:

```
Parms           struc
                dw      ?       ;pushed BP
                dw      ?       ;return address
Ptr1            dd      ?
Ptr2            dd      ?
Parms           ends
;
AddTwoFarInts   proc    near
                push    bp              ;save caller's BP
                mov     bp,sp           ;point to stack frame
                les     bx,[Ptr1]       ;load both segment and offset of Ptr1
                mov     ax,es:[bx]      ;get first int to add
                les     bx,[Ptr2]       ;load both segment and offset of Ptr2
                add     ax,es:[bx]      ;add the two ints together
                pop     bp                      ;restore caller's BP
                ret
AddTwoFarInts   endp
```

(We'll talk about **struc**, stack frames, and segment overrides-- such as **es:**-- later in this chapter.)

High-level languages use **les** all the time to point to data that's not in the default data segment, and that hurts performance significantly. Most high-level languages aren't very smart about using **les**, either. For example, high-level languages tend to load a full 20-bit pointer into ES:BX every time through a loop, even though ES never gets changed from the last pass through the loop. That's one reason why high-level languages don't perform very well with more than 64 Kb of data.

You can usually easily avoid **les**-related performance problems in assembler. Consider Listing 7-4, which adds one far array to another far array in the same way that most high-level languages would, storing both far pointers in memory variables and loading each pointer with **les** every time it's used. (Actually, Listing 7-4 is better than your average high-level language subroutine because it uses **loop**, while most high-level languages use less efficient instruction sequences to handle looping.) Listing 7-4 runs in 43.42 ms, or 43 us per array element addition.

Now look at Listing 7-5, which does exactly the same thing that Listing 7-4 does...except that it loads the far pointers <u>outside</u> the loop and keeps them in the registers for the duration of the loop, using the segment-loading techniques that we learned earlier in this chapter. How much difference does it make to keep the far pointers in registers at all times? Listing 7-5 runs in 19.69 ms--more than twice as fast as Listing 7-4.

Now you know why I keep saying that assembler can handle segments much better than high-level languages can. Listing 7-5 isn't the ultimate in that regard, however; we can carry that concept a step further still, as shown in Listing 7-6.

Listing 7-6 brings the full power of assembler to bear on the task of adding two arrays. Listing 7-6 sets up the segments so that they never once need to be loaded within the loop. What's more, Listing 7-6 arranges the registers so that the powerful **lodsb** string instruction can be used in place of a **mov** and an **inc**. (We'll discuss the string instructions in Chapter 10. For now, just take my word that the string instructions are good stuff.) In short, Listing 7-6 organizes segment and register usage so that as much work as possible is moved out of the loop, and so that the most efficient instructions can be used. The results are stunning.

Listing 7-6 runs in just 13.79 ms, more than three times as fast as Listing 7-4, even though Listing 7-4 uses the efficient **loop** and **les** instructions. This example is a powerful reminder of two important aspects of the Zen of assembler. First, you must strive to play to the strengths of the 8088 (such as the string instructions) while sidestepping its weaknesses (such as the segments and slow memory access speed). Second, you must always concentrate on moving cycles out of loops. The **lds** and **les** instructions outside the loop in Listing 7-6 effectively run 1000 times faster than the **les** instructions inside the loop in Listing 7-4, since the latter are executed 1000 times but the former are executed only once.

## LOADING DOUBLEWORDS WITH les

While **les** isn't often used to load segment:offset pointers in pure assembler programs, it has another less obvious use: loading doubleword values into the general-purpose registers.

Normally, a doubleword value is loaded into two general- purpose registers with two instructions. Here's the standard way to load DX:AX from the doubleword memory variable **DVar**:

```
mov     ax,word ptr [DVar]
mov     dx,word ptr [DVar+2]
```

There's nothing wrong with this approach, but it does take between 4 and 8 bytes and between 34 and 48 cycles. We can cut the time nearly in half, and can usually reduce the size as well, by using **les** in a most unusual way:

```
les     ax,[DVar]
mov     dx,es
```

The only disadvantage of using **les** to load doubleword values is that it wipes out the contents of ES; if that isn't a problem, there's simply no reason to load doubleword values any other way.

Once again, there are those people who will tell you that it's a bad idea to load ES with anything but specific segment values, because such code won't work if you port it to run in protected mode on the 80286 and 80836. While that's a consideration, it's not an overwhelming one. For one thing, most code will never be ported to protected mode. For another, protected mode programming, which we'll touch on in Chapter 15, differs from normal 8088 assembler programming in a number of ways; using **les** to load doubleword values is unlikely to be the most difficult part of porting code to protected mode, especially if you have to rewrite the code to run under a new operating system. Still, if protected mode concerns you, use a macro such as:

```
LOAD_32_BITS    macro   Address
ifdef PROTECTED_MODE
                mov     ax,word ptr [Address]
                mov     dx,word ptr [Address+2]
else
                les     ax,dword ptr [Address]
                mov     dx,ax
endif
                endm
                 :
                LOAD_32_BITS    DwordVar
```

to load 32-bit values.

The **les** approach to loading doubleword values is not only fast but has a unique virtue: it's indivisible. In other words, there's no way an interrupt can occur after the lower word of a doubleword is read but before the upper word is read. For example, suppose we want to read the timer count the BIOS maintains at 0000:046C. We <u>could</u> read the count like this:

```
                sub     ax,ax
                mov     es,ax
                mov     ax,es:[46ch]
                mov     dx,es:[46eh]
```

There's a problem with this code, though. Every 54.9 ms, the timer generates an interrupt which starts the BIOS timer tick handler. The BIOS handler then increments the timer count. If an interrupt occurs right after **mov ax,es:[46ch]** in the above code--before **mov dx,es:[46eh]** can execute--we would read half of the value before it's advanced, and half of the value after it's advanced. If this happened as an hour or a day turned over, we could conceivably read a count that's seriously wrong, with potentially disastrous implications for any program that relies on precise time synchronization. Over time, such a misread of the timer is bound to happen if we use the above code.

We could solve the problem by disabling interrupts while we read the count:

```
sub     ax,ax
mov     es,ax
cli
mov     ax,es:[46ch]
mov     dx,es:[46eh]
sti
```

but there's a better solution. There's no way **les** can be interrupted as it reads a doubleword value, so we'll just load our doubleword thusly:

```
sub     ax,ax
mov     es,ax
les     ax,es:[46ch]
mov     dx,es
```

This last bit of code is shorter, faster, and uninterruptible--in short, it's perfect for our needs. In fact, we could have put **les** to good use reading the BIOS timer count in the long-period Zen timer, way back in Listing 2-5. Why didn't I use it there? The truth is that I didn't know about using **les** to load doublewords when I wrote the timer (which just goes to show that there's always more to learn about the 8088). When I did learn about loading doublewords with **les**, it didn't make any sense to tinker with code that worked perfectly well just to save a few bytes and cycles, particularly because the timer count load isn't time-critical.

Remember, it's only worth optimizing for speed when the cycles you save make a significant difference...which usually means inside tight loops.

## SEGMENT:OFFSET AND BYTE ORDERING IN MEMORY

Our discussion of **les** brings up the topic of how multi-byte values are stored in memory on the 8088. That's an interesting topic indeed; on occasion we'll need to load just the segment part of a 20-bit pointer from memory, or we'll want to modify only the upper byte of a word variable. The answer to our question is simple but by no means obvious: <u>multi-byte values are always stored with the least-significant byte at the lowest address</u>.

For example, when you execute **mov ax,[WordVar]**, AL is loaded from address **WordVar**, and AH is loaded from address **WordVar+1**, as shown in Figure 7-4. Put another way, this:

```
mov     ax,[WordVar]
```

is logically equivalent to this:

```
mov     al,byte ptr [WordVar]
mov     ah,byte ptr [WordVar+1]
```

although the single-instruction version is much faster and smaller. All word-sized values (including address displacements, which we'll get to shortly) follow this least-significant-byte- first memory ordering.

Similarly, segment:offset pointers are stored with the least-significant byte of the offset at the lowest memory address, the most-significant byte of the offset next, the least- significant byte of the segment after that, and the most- significant byte of the segment at the highest memory address, as shown in Figure 7-5. This:

```
les     dx,dword ptr [FarPtr]
```

is logically equivalent to this:

```
mov     dx,word ptr [FarPtr]
mov     es,word ptr [FarPtr+2]
```

which is in turn logically equivalent to this:

```
mov     dl,byte ptr [FarPtr]
mov     dh,byte ptr [FarPtr+1]
mov     al,byte ptr [FarPtr+2]
mov     ah,byte ptr [FarPtr+3]
mov     es,ax
```

This organization applies to all segment:offset values stored in memory, including return addresses placed on the stack by far calls, far pointers used by far indirect calls, and interrupt vectors.

There's nothing sacred about having the least-significant byte at the lowest address; it's just the approach Intel chose. Other processors store values with most-significant byte at the lowest address, and there's a sometimes heated debate about which memory organization is better. That debate is of no particular interest to us; we'll be using an Intel chip, so we'll always be using Intel's least-significant-byte-first organization.

So, to load just the segment part of the 20-bit pointer **FarPtr**, we'd use:

```
mov     es,word ptr [FarPtr+2]
```

and to increment only the upper byte of the word variable **WordPtr**, we'd use:

```
inc     byte ptr [WordVar+1]
```

Remember that the least-significant byte of any value (the byte that's closest to bit 0 when the value is loaded into a register) is always stored at the lowest memory address, and that offsets are stored at lower memory addresses than segments, and you'll be set.

## LOADING SS

I'd like to take a moment to remind you that SP must be loaded whenever SS is loaded, and that interrupts should be disabled for the duration of the load, as we discussed in the last chapter. It would have been handy if Intel had given us an **lss** instruction, but they didn't. Instead, we'll load SS and SP with code along the lines of:

```
	cli
	mov	ss,[NewSS]
	mov	sp,[NewSP]
	sti
```

## EXTRACTING SEGMENT VALUES WITH THE seg DIRECTIVE

Next, we're going to look very quickly at a MASM operator and a MASM directive. As I've said, this is not a book about MASM, but these directives are closely related to the efficient use of segments.

The **seg** operator returns the segment within which the following symbol (label or variable name) resides. In the following code, **seg WordVar** returns the segment **Data**, which is then loaded into ES and used to assume ES to that segment:

```
Data		segment
```

```
WordVar         dw      0
Data            ends
Code            segment
                assume  cs:Code, es:Nothing
                  :
                mov     ax,seg WordVar
                mov     es,ax
                assume  es:seg WordVar
                  :
Code            ends
```

You may well ask why it's worth bothering with **seg**, when we could simply have used the segment name **Data** instead. The answer is that you may not know or may not have direct access to the segment name for variables that are declared in other modules. For example, suppose that **WordVar** were external in our last example:

```
                extrn WordVar:word
Code            segment
                assume  cs:Code, es:Nothing
                  :
                mov     ax,seg WordVar
                mov     es,ax
                assume  es:seg WordVar
                  :
Code            ends
```

This code still returns the segment of **WordVar** properly, even though we don't necessarily have any idea at all as to what the name of that segment might be.

In short, **seg** makes it easier to work with multiple segments in multi-module programs.

## JOINING SEGMENTS

Selected assembler modules can share the same code and/or data segments even when multiple code and data segments are used. In other words, in assembler you can choose to share segments between modules or not as you choose, by contrast with high-level languages, which generally force you to choose between all or no modules sharing segments. (This is not always the case, however, as we'll see in Chapter 14.)

The mechanism for joining or separating segments is the **segment** directive. If each of two modules has a segment of the same name, and if those segments are created as public segments (via the **public** option to the **segment** directive), then those segments will be joined into a single, shared segment. If the segments are code segments, you can use near calls (faster and smaller than far calls) between the modules. If the segments are data segments, then there's no need for one module to load segment registers in order to access data in the other module.

All in all, shared segments allow multiple-module programs to produce code that's as efficient as single-module code, with the segment registers changed as infrequently as possible. In the same program in which multiple modules share a given segment, however, other modules--or even other parts of the same modules-- may share segments of different names, or may have segments that are private (unique to that module). As a result, assembler programs can strike an effective balance between performance and available memory: efficient offset-only addressing most of the time, along with access to as many segments and as much memory as the PC can handle on an as-needed basis.

There are many ways to join segments, including grouping them and declaring them common, and there are many options to the **segment** directive. We need to get on with our discussion of memory addressing, so we won't cover MASM's segment-related directives further, but I strongly suggest that you carefully read the discussion of those directives in your assembler's manual. In fact, you should make it a point to read your assembler's manual cover to cover--it may not be the most exciting reading around, but I guarantee that there are tricks and tips in there that you'll find nowhere else.

While we won't discuss MASM's segment-related directives again, we will explore the topic of effective segment use again in Chapter 10 (as it relates to the string instructions), Chapter 14 (as it relates to branching), and in Volume II of The Zen of Assembly Language.

## SEGMENT OVERRIDE PREFIXES

As we saw in Chapter 6, all memory accesses default to accessing memory relative to one of the four segment registers. Instructions come from CS, stack accesses and memory accesses that use BP as a pointer occur within SS, string instruction accesses via DI are in ES, and everything else is normally in DS. In some--but by no means all--cases, segments other than the default segments can be accessed by way of segment override prefixes, special bytes that can precede--prefix--instructions in order to cause those instructions to use any one of the four segment registers.

Let's start by listing the types of memory accesses segment override prefixes can't affect. Instructions are always fetched from CS; there's no way to alter that. The stack pointer is always used as a pointer into SS, no matter what. ES is always the segment to which string instruction accesses via DI go, regardless of segment override prefixes. Basically, it's accesses to explicitly named memory operands and string instruction accesses via SI that are affected by segment override prefixes. (The segment accessed by the unusual **xlat** instruction, which we'll encounter later in this chapter, can also be overridden.)

The default segment for a memory operand is overridden by placing the prefix **CS:**, **DS:**, **ES:**, or **SS:** on that memory operand. For example:

```
sub     bx,bx
mov     ax,es:[bx]
```

loads AX with the word at offset 0 in ES, as opposed to:

```
sub     bx,bx
mov     ax,[bx]
```

which loads AX with the word at offset 0 in DS. Segment override prefixes are handy in a number of situations. They're good for accessing data out of CS when you're not sure where DS is pointing, or when DS is temporarily pointing to some segment that doesn't contain the data you want. (CS is the one segment upon whose setting you can absolutely rely at any given time, since you know that if a given instruction is being executed, CS <u>must</u> be pointing to the segment containing that instruction. Consequently, CS is a good place to put jump tables and temporary variables in multi-segment programs, and is a particularly handy segment in which to stash data in interrupt handlers, which start up with only CS among the four segment registers set to a known value.)

In many programs, especially those involving high-level languages, DS and SS normally point to the same segment, since it's convenient to have both stack frame variables and static/global variables in the same segment. When that's the case, **ss:** prefixes can be used to point to data in the default data segment when DS is otherwise occupied. Even when SS doesn't point to the default data segment, segment override prefixes still let you address data on the stack using pointer registers other than BP.

Segment override prefixes are particularly handy when you need to access data in two to four segments at once. Suppose, for example, that we need to add two far word-sized arrays together and store the resulting array in the default data segment. Assuming that SS and DS both point to the default data segment, segment override prefixes let us keep all our pointers and counters in the registers as we add the arrays, as follows:

```
        push    ds                      ;save normal DS
        les     di,[FarPtr2]            ;point ES:DI to one source array
        mov     bx,[DestPtr]            ;point SS:BX to the destination array
        mov     cx,[AddLength]          ;array length
        lds     si,[FarPtr1]            ;point DS:SI to the other source array
        cld                             ;make LODSW count up
Add3Loop:
        lodsw                           ;get the next entry from one array
        add     ax,es:[di] ;add it to the other array
        mov     ss:[bx],ax              ;save the sum in a third array
        inc     di                      ;point to the next entries
        inc     di
        inc     bx
        inc     bx
        loop    Add3Loop
        pop     ds                      ;restore normal DS
```

Had we needed to, we could also have stored data in CS by using **cs:**.

Handy as segment override prefixes are, you shouldn't use them too heavily if you can help it. They're fine for one-shot instructions such as branching through a jump table in CS or retrieving a byte from the BIOS data area by way of ES, but they're to be avoided whenever possible inside tight loops. The reason: segment override prefixes officially take 2 cycles to execute and, since they're 1 byte long, they can actually take up to 4 cycles to fetch and execute--and 4 cycles is a significant amount of time inside a tight loop.

Whenever you can, organize your segments outside loops so that segment override prefixes aren't needed inside loops. For example, consider Listing 7-7, which uses a segment override prefix while stripping the high bit of every byte in an array in the segment addressed via ES. Listing 7-7 runs in 2.95 ms.

Now consider Listing 7-8, which does the same thing as Listing 7-7, save that DS is set to match ES outside the loop. Since DS is the default segment for the memory accesses we perform inside the loop, there's no longer any need for a segment override prefix...and that one change improves performance by nearly 14%, reducing total execution time to 2.59 ms.

The lesson is clear: don't use segment override prefixes in tight loops unless you have no choice.

## assume AND SEGMENT OVERRIDE PREFIXES

Segment override prefixes can find their way into your code even if you don't put them there, courtesy of the assembler and the **assume** directive. **assume** tells MASM what segments are currently addressable via the segment registers. Whenever MASM doesn't think the default segment register for a given instruction can reach the desired segment but another segment register can, MASM sticks in a segment override prefix without telling you it's doing so. As a result, your code can get bigger and slower without you knowing about it.

Take a look at this code:

```
Code            segment
                assume    cs:code
Start proc      far
                jmp       Skip
ByteVar         db        0
Skip:
                push      cs
                pop       ds        ;set DS to point to the segment Code
                inc       [ByteVar]
                  :
Code            ends
```

You know and I know that DS can be used to address **ByteVar** in the above code, since the first thing the code does is set DS equal to CS, thereby loading DS to point to the segment **Code**. Unfortunately, the assembler does not know that--the **assume** directive told it only that CS points to **Code**, and **assume** is all the assembler has to go by. Given this correct but not complete information, the assembler concludes that **ByteVar** must be addressed via CS and inserts a **cs:** segment override prefix, so the **inc** instruction assembles as if **inc cs:[ByteVar]** had been used.

The result is a wasted byte and several wasted cycles. Worse yet, you have no idea that the segment override prefix has been inserted unless you either generate and examine a listing file or view the assembled code as it runs in a debugger. The assembler is just trying to help by taking some of the burden of segment selection away from you, but the outcome is all too often code that's invisibly bloated with segment override prefixes.

The solution is simple. Keep the assembler's segment assumptions correct at all times by religiously using the **assume** directive every time you load a segment. The above example would have assembled correctly--without a segment override prefix--if only we had inserted the line:

```
        assume  ds:Code
```

before we had attempted to access **ByteVar**.

## OFFSET HANDLING

At long last, we've completed our discussion of segments. Now it's time to move on to the other half of the memory- addressing equation: offsets.

Offsets are handled somewhat differently from segments. Segments are simply loaded into the segment registers, which are then used to address memory as half of a segment:offset address. Offsets can also be loaded into registers and used directly as half of a segment:offset address, but just as often offsets are built into instructions, and they can also be calculated on the fly by summing the contents of one or two registers and/or offsets built into instructions.

At any rate, we'll quickly cover offset loading, and then we'll look at the many ways to generate offsets for memory addressing. The offset portion of memory addressing is one area in which the 8088 is very flexible, and, as we'll see, there's no one best way to address memory.

## LOADING OFFSETS

Offsets are loaded with the **offset** operator. **offset** is analogous to the **seg** operator we encountered earlier; the difference, of course, is that **offset** extracts the offset of a label or variable name rather than the segment. For example:

```
	mov	bx,offset WordVar
```

loads BX with the offset of the variable **WordVar**. If some segment register already points to the segment containing **WordVar**, then BX can be used to address memory, as for example in:

```
	mov	bx,seg WordVar
	mov	es,bx
	mov	bx,offset WordVar
	mov	ax,es:[bx]
```

We'll discuss the many ways in which offsets can be used to address memory next.

Before we get to using offsets to address memory, there are a couple of points I'd like to make. The first point is that the **lea** instruction can also be used to load offsets into registers; however, an understanding of **lea** requires an understanding of the 8088's addressing modes, so we'll defer the discussion of **lea** until later in this chapter.

The second point is a shortcoming of MASM that you must be aware of when you use **offset** on variables that reside in segment groups. If you are using the **group** directive to make segment groups, you must always specify the group name as well as the variable name when you use the offset operator. For example, if the segment **_DATA** is in the group **DGROUP**, and **WordVar** is in **_DATA**, you must load the offset of **WordVar** as follows:

```
	mov	di,offset DGROUP:WordVar
```

If you don't specify the group name, as in:

```
	mov	di,offset WordVar
```

the offset of **WordVar** relative to **_DATA** rather than **DGROUP** is loaded; given the way segment groups are organized (with all segments in the group addressed in a single combined segment), an offset relative to **_DATA** may not work at all.

I realize that the above discussion won't make much sense if you haven't encountered the **group** directive (lucky you!). I've never found segment groups to be necessary in pure assembler code, but they are often needed when sharing segments between high-level language code and assembler. If you do find yourself using segment groups, all you need to remember is this: when loading the offset of a variable that resides within a segment group with the **offset** operator, always specify the group name along with the variable name.

## mod-reg-rm ADDRESSING

There are a number of ways in which the offset of an instruction operand can be specified. Collectively, the ways of specifying operand offsets are known as addressing modes. Most of the 8088's addressing modes fall into a category known as mod- reg-rm addressing modes. We're going to discuss mod-reg-rm addressing modes next; later in the chapter we'll discuss non- mod-reg-rm addressing modes.

mod-reg-rm addressing modes are so named because they're specified by a second instruction byte, known as the mod-reg-rm byte, that follows instruction opcodes in order to specify the memory and/or register operands for many instructions. The mod- reg-rm byte gets its name because the various fields within the byte are used to specify the memory addressing mode, the register used for one operand, and the register or memory location used for the other operand, as shown in Figure 7-6. (Figure 7-6 should make it clear that at most only one mod-reg-rm operand can be a memory operand; one or both operands must be register operands, for there just aren't enough bits in a mod-reg-rm byte to specify two memory operands.)

Simply put, the mod-reg-rm byte tells the 8088 where to find an instruction's operand or operands. (It's up to the opcode byte to specify the data size, as well as which operand is the source and which is the destination.) When a memory operand is used, the mod-reg-rm byte tells the 8088 how to add together the contents of registers (BX or BP and/or SI or DI) and/or a fixed value built into the instruction (a displacement) in order to generate the operand's memory offset. The offset is then combined with the contents of one of the segment registers to make a full 20-bit memory address, as we saw earlier in this chapter, and that 20-bit address serves as the instruction operand. Figure 7-7 illustrates the operation of the complex base+index+displacement addressing mode, in which an offset is generated by adding BX or BP, SI or DI, and a fixed displacement. (Note that displacements are built right into instructions, coming immediately after mod-reg-rm bytes, as illustrated by Figure 7-9.)

For example, if the opcode for **mov reg8,[reg/mem8]** (8Ah) is followed by the mod-reg-rm byte 17h, that indicates that the register DL is to be loaded from the memory location pointed to by BX, as shown in Figure 7-8. Put the other way around, **mov dl,[bx]** assembles to the two byte sequence 8Ah 17h, where the first byte is the opcode for **mov reg8,[reg/mem8]** and the second byte is the mod-reg-rm byte that selects DL as the destination and the memory location pointed to by BX as the source.

You may well wonder how the mod-reg-rm byte works with one-operand instructions, such as **neg word ptr ds:[140h]**, or with instructions that have constant data as one operand, such as **sub [WordVar],1**. The answer is that in these cases the reg field isn't used for source or destination control; instead, it's used as an extension of the opcode byte. So, for instance, **neg [reg/mem16]** has an opcode byte of 0F7h and always has bits 5-3 of the mod-reg-rm byte set to 011b. Bits 7-6 and 2-0 of the mod-reg-rm byte still select the memory addressing mode for the single operand, but bits 5-3, together with the opcode byte, now simply tell the 8088 that the instruction is **neg [reg/mem16]**, as shown in Figure 7-9. **not [reg/mem16]** also has an opcode byte of 0F7h, but is distinguished from **neg [reg/mem16]** by bits 5-3 of the mod-reg-rm byte, which are 010b for **not** and 011b for **neg**.

At any rate, the mechanics of mod-reg-rm addressing aren't what we need to concern ourselves with; the assembler takes care of such details, thank goodness. We do, however, need to concern ourselves with the implications of mod-reg-rm addressing, particularly size and performance issues.

## WHAT'S mod-reg-rm ADDRESSING GOOD FOR?

The first thing to ask is, "What is mod-reg-rm addressing good for?" What mod-reg-rm addressing does best is address memory in a very flexible way. No other addressing mode approaches mod-reg-rm addressing for sheer number of ways in which memory offsets can be generated.

Look at Figure 7-6, and try to figure out how many source/destination combinations are possible with mod-reg-rm addressing. The answer is simple, since there are 8 bits in a mod-reg-rm byte; 256 possible source/destination combinations are supported. Any general-purpose register can be one operand, and any general-purpose register or memory location can be the other operand.

If we look at memory addressing alone, we see that there are 24 distinct ways to generate a memory offset. (8 of the 32 possible selections that can be made with bits 7-6 and 3-0 of the mod-reg-rm byte select general-purpose registers.) Some of those 24 selections differ only in whether 1 or 2 displacement bytes are present, leaving us with the following 16 completely distinct memory addressing modes:

| | |
|---|---|
| [disp16] | [bp+disp] |
| [bx] | [bx+disp] |
| [si] | [si+disp] |
| [di] | [di+disp] |
| [bp+si] | [bp+si+disp] |
| [bp+di] | [bp+di+disp] |
| [bx+si] | [bx+si+disp] |
| [bx+di] | [bx+di+disp] |

For two-operand instructions, each of those memory addressing modes can serve as either source or destination, with either a constant value or one of the 8 general-purpose registers as the other operand.

Basically, mod-reg-rm addressing lets you select a memory offset in any of 16 ways (or a general-purpose register, if you prefer), and say, "Use this as an operand." The other operand can't involve memory, but it can be any general-purpose register or (usually) a constant value. (There's no inherent support in <u>mod-reg-rm</u> addressing for constant operands. Special, separate opcodes must used to specify constant operands for instructions that support such operands, and a few <u>mod-reg-rm</u> instructions, such as **mul**, don't accept constant operands at all.)

<u>mod-reg-rm</u> addressing is flexible indeed.

## DISPLACEMENTS AND SIGN-EXTENSION

I've said that displacements can be either 1 or 2 bytes in size. The obvious question is: what determines which size is used? That's an important question, since displacement bytes directly affect program size, which in turn indirectly affects performance via the prefetch queue cycle-eater.

Except in the case of direct addressing, which we'll discuss shortly, displacements in the range -128 to +127 are stored as one byte, then automatically sign-extended by the 8088 to a word when the instructions containing them are executed. (Expressed in unsigned hexadecimal, -128 to +127 covers two ranges: 0 to 7Fh and 0FF80h to 0FFFFh.) Sign-extension involves copying bit 7 of the byte to bits 15-8, so a byte value of 80h sign-extends to 0FF80h, and a byte value of 7Fh sign-extends to 0007Fh. Basically, sign-extension converts signed byte values to signed word values; since the maximum range of a signed byte is -128 to +127, that's the maximum range of a 1-byte displacement as well.

The implication of this should be obvious: you should try to use displacements in the range -128 to +127 whenever possible, in order to reduce program size and improve performance. One caution, however: displacements must be either numbers or symbols equated to numbers in order for the assembler to be able to assemble them as single bytes. (Numbers and symbols work equally well. In:

```
SAMPLE_DISPLACEMENT     equu    1
                        :
                        mov     ax,[bx+SAMPLE_DISPLACEMENT]
                        mov     ax,[bx+9]
```

both **mov** instructions assemble with 1-byte displacements.)

Displacements must be constant values in order to be stored in sign-extended bytes because when a named memory variable is used, the assembler has no way of knowing where in the segment the variable will end up. Other parts of the segment may appear in other parts of the module or may be linked in from other modules, and the linker may also align the segment to various memory boundaries; any of these can have the effect of moving a given variable in the segment to an offset that doesn't fit in a sign-extended byte. As a result, the following **mov** instruction assembles with a 2-byte displacement, even though it appears to be at offset 0 in its segment:

```
Data            segment
MemVar          db      10 dup (?)
Data            ends
                 :
                mov     al,[MemVar+bx]
```

**NAMING THE mod-reg-rm ADDRESSING MODES** The 16 distinct memory addressing modes supported by the mod-reg-rm byte are often given a slew of confusing names, such as "implied addressing," "based relative addressing," and "direct indexed addressing." Generally, there's little need to name addressing modes; you'll find you use them much more than you talk about them. However, we will need to refer to the modes later in this book, so let me explain my preferred addressing mode naming scheme.

I find it simplest to give a name to each of the three possible components of a memory offset--base for BX or BP, index for SI or DI, displacement for a 1- or 2-byte fixed value--and then just refer to an addressing mode with all the components of that mode. That way, **mov [bx],al** uses base addressing, **add ax,[si+1]** uses index+displacement addressing, and **mov dl,[bp+di+1000h]** uses base+index+displacement addressing. The names may be long at times, but they're never ambiguous or hard to remember.

## DIRECT ADDRESSING

There is one exception to the above naming scheme, and that's direct addressing. Direct addressing is used when a memory address is referenced with just a 16-bit displacement, as in **mov bx,[WordVar]** or **mov es:[410h],al**. You might expect direct addressing to be called displacement addressing, but it's not, for three reasons. First, the address used in direct addressing is not, properly speaking, a displacement, since it isn't relative to any register. Second, direct addressing is a time- honored term that came into use long before the 8088 was around, so experienced programmers are more likely to speak of "direct addressing" than "displacement addressing."

Third, direct addressing is a bit of an anomaly in mod-reg- rm addressing. It's pretty obvious why we'd want to have direct addressing available; surely you'd rather do this:

```
	mov	dx,[WordVar]
```

than this:

```
	mov	bx,offset WordVar
	mov	dx,[bx]
```

It's just plain handy to be able to access a memory location directly by name.

Now look at Figure 7-6 again. Direct addressing really doesn't belong in that figure at all, does it? The mod-reg-rm encoding for direct addressing should by all rights be taken by base addressing using only BP. However, there is no addressing mode that can use only BP--if you assemble the instruction **mov [bp],al**, you'll find that it actually assembles as **mov [bp+0],al**, with a 1-byte displacement.

In other words, the designers of the 8088 rightly considered direct addressing important enough to build it into mod-reg-rm addressing in place of a little-used addressing mode. (BP is designed to point to stack frames, as we'll see shortly, and there's rarely any use for BP-only base addressing in stack frames.)

Along the same lines, note that direct addressing always uses a 16-bit displacement. Direct addressing does not use an 8- bit sign-extended displacement even if the address is in the range -128 to +127.

## MISCELLANEOUS INFORMATION ABOUT MEMORY ADDRESSING

Be aware that all mod-reg-rm addressing defaults to accessing the segment pointed to by DS--except when BP is used as part of the mod-reg-rm address. Any mod-reg-rm addressing involving BP accesses the segment pointed to by SS by default. (If DS and SS point to the same segment, as they often do, you can use BP-based addressing modes to point to normal data if necessary, and you can use the other mod-reg-rm addressing modes to point to data on the stack.) However, mod-reg-rm addressing can always be forced to use any segment register with a segment override prefix.

There are a few other addressing terms that I should mention now. Indirect addressing is commonly used to refer to any sort of memory addressing that uses a register (BX, BP, SI, or DI, or any of the valid combinations) to point to memory. We'll also use indirect to refer to branches that branch to destinations specified by memory operands, as in **jmp word ptr [SubroutinePointer]**. We'll discuss indirect branching in detail in Chapter 14.

Immediate addressing is a non-mod-reg-rm form of addressing in which the operand is a constant value that's built right into the instruction.

We'll cover immediate addressing when we're done with mod-reg-rm addressing.

Finally, I'd like to make it clear that a displacement is nothing more than a fixed (constant) value that's added into the memory offset calculated by a mod-reg-rm byte. It's called a displacement because it specifies the number of bytes by which the addressed offset should be displaced from the offset specified by the registers used to point to memory. In **mov si,[bx+1]**, the displacement is 1; the address from which SI is loaded is displaced 1 byte from the memory location pointed to by BX. In **mov ax,[si+WordVar]**, the displacement is the offset of **WordVar**. We won't know exactly what that offset is unless we look at the code with a debugger, but it's a constant value nonetheless.

Don't get caught up worrying about the exact meaning of the term displacement, or indeed of any of the memory addressing terms. In a way, the terms are silly; **mov ax,[bx]** is base addressing and **mov ax,[si]** is index addressing, but both load AX from the address pointed to by a register, both are 2 bytes long, and both take 13 cycles to execute. The difference between the two is purely semantic from a programmer's perspective.

Notwithstanding, we needed to establish a common terminology for the mod-reg-rm memory addressing modes, and we've done so. Now that we understand how mod-reg-rm addressing works and how wonderfully flexible it is, let's look at its dark side.

## mod-reg-rm ADDRESSING: THE DARK SIDE

Gee, if mod-reg-rm addressing is so flexible, why don't we use it for all memory accesses? For that matter, why does the 8088 even have any other addressing modes?

One reason is that mod-reg-rm addressing doesn't work with all instructions. For example, the string instructions can't use mod-reg-rm addressing, and neither can **xlat**, which we'll encounter later in this chapter. Nonetheless, most instructions, including **mov**, **add**, **adc**, **sub**, **sbb**, **cmp**, **and**, **or**, **xor**, **neg**, **not**, **mul**, **div**, and more, do support mod-reg-rm addressing, so it would seem that there must be some other reason for the existence of other addressing modes.

And indeed there is another reason for the existence of other addressing modes. In fact, there are two reasons: speed and size. mod-reg-rm addressing is more flexible than other addressing modes--and it also produces the largest, slowest code around.

It's easy to understand why mod-reg-rm addressing produces larger code than other memory addressing modes. The bits needed to encode mod-reg-rm addressing's many possible source, destination, and addressing mode combinations increase the size of mod-reg-rm instructions, and displacement bytes can make modreg-rm instructions larger still. It stands to reason that the string instruction **lods**, which always loads AL from the memory location pointed to by DS:SI, should have fewer instruction bytes than the mod-reg-rm instruction **mov al,[si]**, which selects AL from 8 possible destination registers, and which selects the memory location pointed to by SI from among 32 possible source operands.

It's less obvious why mod-reg-rm addressing is slower than other memory addressing modes. One major reason falls out from the larger size of mod-reg-rm instructions; we've already established that instructions with more instruction bytes tend to run more slowly, simply because it takes time to fetch those extra instruction bytes. That's not the whole story, however. It takes the 8088 a variable but considerable amount of time--5 to 12 cycles--to calculate memory addresses from mod-reg-rm bytes. Those lengthy calculations, known as effective address (EA) calculations, are our next topic.

Before we proceed to EA calculations, I'd like to point out that slow and bulky as mod-reg-rm addressing is, it's still the workhorse memory addressing mode of the 8088. It's also the addressing mode used by many register-only instructions, such as **add dx,bx** and **mov al,dl**, with the mod-reg-rm byte selecting register rather than memory operands. My goodness, some instructions don't even have a non-mod-reg-rm addressing mode. Without a doubt, you'll be using mod-reg-rm addressing often in your code, so we'll take the time to learn how to use it well. Nonetheless, the less-flexible addressing modes are generally shorter and faster than mod-reg-rm addressing. As we'll see throughout The Zen of Assembly Language, one key to high-performance code is avoiding mod-reg-rm addressing as much as possible.

## WHY MEMORY ACCESSES ARE SLOW

As I've already said, mod-reg-rm memory accesses are slow partly because instructions that use mod-reg-rm addressing tend to have many instruction bytes. The mod-reg-rm byte itself adds 1 byte beyond the opcode byte, and a displacement, if used, will add 1 or 2 more bytes. Remember, 4 cycles are required to fetch each and every one of those instruction bytes.

Taken a step farther, that line of thinking reveals why all instructions that access memory are slow: memory is slow. It takes 4 cycles per byte to access memory in any way. That means that an instruction like **mov bx,[WordVar]**, which is 4 bytes long and reads a word-sized memory variable, must perform 6 memory accesses in all; at 4 cycles a pop, that adds up to a minimum execution time of 24 cycles. Even a 2-byte memory-accessing instruction spends a minimum of 12 cycles just accessing memory. By contrast, most register-only operations are 1 to 2 bytes in length and have Execution Unit execution times of 2 to 4 cycles, so the maximum execution times for register-only instructions tend to be 4 to 8 cycles.

I've said it before, and I'll say it again: avoid accessing memory whenever you can. Memory is just plain slow.

In actual use, many memory-accessing instructions turn out to be even slower than memory access times alone would explain. For example, the fastest possible mod-reg-rm memory-accessing instruction, **mov reg8,[bx]** (BP, SI, or DI would do as well as BX), has an Execution Unit execution time of 13 cycles, although only 3 memory accesses (requiring 12 cycles) are performed. Similarly, string instructions, **xlat**, **push**, and **pop** take more cycles than can be accounted for solely by memory accesses.

The full explanation for the poor performance of the 8088's memory-accessing instructions lies in the microcode of the 8088 (the built-in bit patterns that sequence the 8088 through the execution of each instruction), which is undeniably slower than it might be. (Check out the execution times of the 8088's instructions on the 80286 and 80386, and you'll see that it's possible to execute the 8088's instructions in many fewer cycles than the 8088 requires.) That's not something we can change; about all we can do is choose the fastest available instruction for each task, and we'll spend much of The Zen of Assembly Language doing just that.

There is one aspect of memory addressing that we can change, however, and that's EA addressing time--the amount of time it takes the 8088 to calculate memory addresses.

## SOME mod-reg-rm MEMORY ACCESSES ARE SLOWER THAN OTHERS

A given instruction that uses mod-reg-rm addressing doesn't always execute in the same number of cycles. The Execution Unit execution time of mod-reg-rm instructions comes in two parts: a fixed Execution Unit execution time and an effective address (EA) execution time that varies depending on the mod-reg-rm addressing mode used. The two times added together determine the overall execution time of each mod-reg-rm instruction.

Each mod-reg-rm instruction has its own fixed Execution Unit execution time, which remains the same for all addressing modes. For example, the fixed execution time of **add bl,[mem]** is 9 cycles, as shown in Appendix A; this value is constant, no matter what mod-reg-rm addressing mode is used.

The EA calculation time, on the other hand, depends not in the least on which instruction is being executed. EA calculation time is determined solely by the mod-reg-rm addressing mode used, and nothing else, as shown in Figure 7-10. As you can see from Figure 7-10, the time it takes the 8088 to calculate an effective address can vary greatly, ranging from a mere 5 cycles if a single register is used to point to memory all the way up to 11 or 12 cycles if the sum of two registers and a displacement is used to point to memory. (Segment override prefixes require an additional 2 cycles each, as we saw earlier.) When I discuss the performance of an instruction that uses mod-reg-rm addressing, I'll often say that it takes at least a certain number of cycles to execute. What "at least" means is that the instruction will take that many cycles if the fastest mod-reg-rm addressing mode-- base- or index-only--is used, and longer if some other mod-reg-rm addressing mode is selected.

Only mod-reg-rm memory operands require EA calculations. There is no EA calculation time for register operands, or for memory operands accessed with non-mod-reg-rm addressing modes.

In short, EA calculation time means that the choice of mod- reg-rm addressing mode directly affects performance. Let's look more closely at the performance implications of EA calculations.

## PERFORMANCE IMPLICATIONS OF EFFECTIVE ADDRESS CALCULATIONS

There are a number of interesting points to be made about EA calculation time. For starters, it should be clear that EA calculation time is a big reason why instructions that use mod- reg-rm addressing are slow. The minimum EA calculation time of 5 cycles, on top of 8 or more cycles of fixed execution time, is no bargain; the maximum EA calculation time of 12 cycles is a grim prospect indeed.

For example, **add bl,[si]** takes 13 cycles to execute (8 cycles of fixed execution time and 5 cycles of EA calculation time), which is certainly not terrific by comparison with the 3- cycle execution time of **add bl,dl**. (Instruction fetching alters the picture somewhat, as we'll see shortly.) At the other end of the EA calculation spectrum, **add bl,[bx+di+100h]** takes 20 cycles to execute, which is horrendous no matter what you compare it to.

The lesson seems clear: use faster mod-reg-rm addressing modes whenever you can. While that's true, it's not necessarily obvious which mod-reg-rm addressing modes are faster. Base-only addressing or index-only addressing are the mod-reg-rm addressing modes of choice, because they add only 5 cycles of EA calculation time and 1 byte, the mod-reg-rm byte. For instance, **mov dl,[bp]** is just 2 bytes long and takes a fairly reasonable 13 cycles to execute.

Direct addressing, which has an EA calculation time of 6 cycles, is only slightly slower than base or index addressing so far as official execution time goes. However, direct addressing requires 2 additional instruction bytes (the 16-bit displacement) beyond the mod-reg-rm byte, so it's actually a good deal slower than base or index addressing. **mov dl,[ByteVar]** officially takes 14 cycles to execute, but given that the instruction is 4 bytes long and performs a memory access, 20 cycles is a more accurate execution time.

Base+index addressing (**mov al,[bp+di]** and the like) takes 1 to 2 cycles more for EA calculation time than does direct addressing, but is nonetheless superior to direct addressing in most cases. The key: base+index addressing requires only the 1 mod-reg-rm byte. Base+index addressing instructions are 2 bytes shorter than equivalent direct addressing instructions, and that translates into a considerable instruction-fetching/performance advantage.

The rule is: use displacement-free mod-reg-rm addressing modes whenever you can. Instructions that use displacements are always 1 to 2 bytes longer than those that use displacement-free mod-reg-rm addressing modes, and that means that there's generally a prefetching penalty for the use of displacements. There's also a substantial EA calculation time penalty for base+displacement, index+displacement, or base+index+displacement addressing. If you must use displacements, use 1-byte displacements as much as possible; we'll see an example of this when we get to stack frames later in this chapter.

Now, bear in mind that the choice of mod-reg-rm addressing mode really only matters inside loops, or in time-critical code. If you're going to load DX from memory just once in a long subroutine, it really doesn't much matter if you take a few extra cycles to load it with direct addressing rather than base or index addressing. It certainly isn't worth loading, say, BX to point to memory, as in:

```
mov     bx,offset MemVar
mov     dx,[bx]
```

just to use base or index addressing once--the **mov** instruction used to load BX takes 4 cycles and 3 bytes, more than negating any advantage base addressing has over direct addressing.

Inside loops, however, it's well worth using the most efficient addressing mode available. Listing 7-9, which adds up the elements of a byte-sized array using base+index+displacement addressing every time through the loop, runs in 1.17 ms. Listing 7-10, which changes the addressing mode to base+index by adding the displacement into the base outside the loop, runs in 1.01 ms, nearly 16% faster than Listing 7-9. Finally, Listing 7-11, which performs all the addressing calculations outside the loop and uses plain old base-only addressing, runs in just 0.95 ms, 6% faster still. (The string instruction **lods** is even faster than **mov al,[bx]**, as we'll see in Chapter 10. Always think of your non-mod-reg-rm alternatives.) Clearly, the choice of addressing mode matters considerably inside tight loops.

We've learned two basic rules, then: 1) use displacement- free mod-reg-rm addressing modes whenever you can, and 2) calculate memory addresses outside loops and use base-only or index-only addressing whenever possible. The **lea** instruction, which we'll get to shortly, is most useful for calculating memory addresses outside loops.

## mod-reg-rm ADDRESSING: SLOW, BUT NOT QUITE AS SLOW AS YOU THINK

There's no doubt about it: mod-reg-rm addressing is slow. Still, relative to register operands, mod-reg-rm operands might not be quite so slow as you think, for a very strange reason--the prefetch queue. mod-reg-rm addressing executes so slowly that it allows time for quite a few instruction bytes to be prefetched, and that means that instructions that use mod-reg-rm addressing often run at pretty much their official speed.

Consider this. **mov al,bl** is a 2-byte, 2-cycle instruction. String a few such instructions together and the prefetch queue empties, making the actual execution time 8 cycles--the time it takes to fetch the instruction bytes. By contrast, **mov al,[bx]** is a 2-byte, 13-cycle instruction. Counting both the memory access needed to read the operand pointed to by BX and the two instruction fetches, only 3 memory accesses are incurred by this instruction. Since 3 memory accesses take only 12 cycles, the 13-cycle official execution time of **mov al,[bx]** is a fair reflection of the instruction's true performance.

That doesn't mean that **mov al,[bx]** is faster than **mov al,bl**, or that memory-accessing instructions are faster than register- only instructions--they're not. **mov al,bl** is a minimum of about 50% faster than **mov al,[bx]** under any circumstances. What it does mean is that memory-accessing instructions tend to suffer less from the prefetch queue cycle-eater than do register-only instructions, because the considerably longer execution times of memory-accessing instructions often allow a good deal of prefetching per instruction byte executed. As a result, the performance difference between the two is often not quite so great as official execution times would indicate.

In short, memory-accessing instructions, especially those that use _mod-reg-rm_ addressing, generally have a better balance between overall memory access time and execution time than register-only instructions, and consequently run closer to their rated speeds. That's a mixed blessing, since it's a side effect of the slow speed of memory-accessing instructions, but it does make memory access--which is, after all, a necessary evil--somewhat less unappealing than it might seem. Let me emphasize that the basic reason that instructions that use _mod-reg-rm_ memory accesses suffer less from the prefetch queue cycle-eater than do equivalent register-only instructions is that both sorts of instructions have _mod-reg-rm_ bytes. True, register-only _mod-reg-rm_ instructions don't have EA calculation times, but they do have at least 2 bytes, making them as long as the shortest _mod-reg-rm_ memory-accessing instructions. (A number of non-_mod-reg-rm_ instructions are just 1 byte long; we'll meet them over the next few chapters.) Since register-only instructions are much faster than memory-accessing instructions, it's just common sense that if they're the same length in bytes then they can be hit much harder by the prefetch queue cycle-eater.

Still and all, register-only _mod-reg-rm_ instructions are _never_ longer than memory-accessing _mod-reg-rm_ instructions, and are shorter than memory-accessing instructions that use displacements. What's more, since memory-accessing instructions must by definition access memory at least once apart from fetching instruction bytes, register-only mod-reg-rm instructions must be at least 50% faster than their memory-accessing equivalents--100% when word-sized operands are used. To sum up, register-only instructions are always much faster and often smaller than equivalent mod-reg-rm memory-accessing instructions. (Register-only instructions are faster than, although not necessarily shorter than or even as short as, non-mod-reg-rm instructions--even the string instructions--as well.) Avoid memory. Use the registers as much as you possibly can.

## THE IMPORTANCE OF ADDRESSING WELL

When you do use mod-reg-rm addressing, do so efficiently. As we've discussed, that means using base- or index-only addressing whenever possible, and avoiding displacements when you can, especially inside loops. If you're only going to access a memory location once and you don't have a pointer to that location already loaded into BX, BP, SI, or DI, just use direct addressing; base- and index-only addressing aren't so much faster than direct addressing that it pays to load a pointer. As we've seen, however, don't use direct addressing inside a loop if you can load a pointer register outside the loop and then use base- or index-only addressing inside the loop.

It's often surprising how much more efficient than direct addressing base- and index-only addressing are. Consider this simple bit of code:

```
mov     dl,[ByteVar]
and     dl,0fh
mov     [ByteVar],dl
```

You wouldn't think that code could be improved upon by <u>adding</u> an instruction, but we can cut the code's size from 10 to 9 bytes by using base-only addressing:

```
mov     bx,offset ByteVar
mov     dl,[bx]
and     dl,0fh
mov     [bx],dl
```

The cycle count is 2 higher for the latter version, but a 2-byte advantage in instruction fetching could well overcome that.

The point is not that base-only addressing is always the best solution. In fact, the latter example could be made much more efficient simply by anding 0Fh directly with memory, as in:

```
and     [ByteVar],0fh
```

(Always bear in mind that memory can serve as the destination operand as well as the source operand. When only one modification is involved, it's always faster to modify a memory location directly, as in the last example, than it is to load a register, modify the register, and store the register back to memory. However, the scales tip when two or more modifications to a memory operand are involved, as we'll see in Chapter 8.) The special accumulator-specific direct-addressing instructions that we'll discuss in the next chapter make direct addressing more desirable in certain circumstances as well.

The point is that for repeated accesses to the same memory location, you should arrange your code so that the most efficient possible instruction--base-only, a string instruction, whatever fills the bill--can be used. In the last example, base-only addressing was superior to direct addressing when just two accesses to the same byte were involved. Multiply the number of accesses by ten, or a hundred, or a thousand, as is often the case in a tight loop, and you'll get a feel for the importance of selecting the correct memory addressing mode in your time- critical code.

## THE 8088 IS FASTER AT MEMORY ADDRESS CALCULATIONS THAN YOU ARE

You may recall that we found earlier that when you must access a word-sized memory operand, it is better to let the 8088 access the second byte than to do it with a separate instruction; the 8088 is simply faster at accessing two adjacent bytes than any two instructions can be. Much the same is true of mod-reg-rm addressing; the 8088 is faster at performing memory address calculations than you are. If you must add registers and/or constant values to address memory, the 8088 can do it faster during EA calculations than you can with separate instructions.

Suppose that we have to initialize a doubleword of memory pointed to by BX to zero. We could do that with:

```
mov     word ptr [bx],0
inc     bx
inc     bx
mov     word ptr [bx],0
```

However, it's better to let the 8088 do the addressing calculations, as follows:

```
mov     word ptr [bx],0
mov     word ptr [bx+2],0
```

True, the latter version involves a 1-byte displacement, but that displacement is smaller than the 2 bytes required to advance BX in the first version. Since the incremental cost of base+displacement addressing over base-only addressing is 4 cycles, exactly the same number of cycles as two **inc** instructions, the code that uses base+displacement addressing is clearly superior.

Similarly, you're invariably better off letting EA calculations add one register to another than you are using **add**. For example, consider two approaches to scanning an array pointed to by BX+SI for the byte in AL:

```
                mov     dx,bx       ;set aside the base address
ScanLoop:
                mov     bx,dx       ;get back the base address
                add     bx,si       ;add in the index
                cmp     [bx],al     ;is this a match?
                jz      ScanFound           ;yes, we're done
                inc     si                  ;advance the index to the next byte
                jmp     ScanLoop            ;scan the next byte
ScanFound:
```

and:

```
ScanLoop:
              cmp     [bx+si],al          ;is this a match?
              jz      ScanFound           ;yes, we're done
              inc     si                  ;advance the index to the next byte
              jmp     ScanLoop            ;scan the next byte
ScanFound:
```

It should be pretty clear that the approach that lets the 8088 add the two memory components together is far superior.

While the point is perhaps a little exaggerated--I seriously doubt anyone would use the first approach--it is nonetheless valid. The 8088 can add BX to SI in just 2 extra cycles as part of an EA calculation, and at the cost of no extra bytes at all. What's more, EA calculations leave all registers unchanged. By contrast, at least one register must be changed to hold the final memory address when you perform memory calculations yourself. That's what makes the first version above so inefficient; we have to reload BX from DX every time through the loop because it's altered by the memory-address calculation.

I hope you noticed that neither example above is particularly efficient. We'd be better off simply adding the two memory components <u>outside</u> the loop and using base- or index-only addressing inside the loop. (We'd be even better off using string instructions, but we'll save that for another chapter.) To wit:

```
              add     si,bx       ;add together the memory address components
                                  ; outside the loop
```

```
ScanLoop:
                cmp     [si],al     ;is this a match?
                jz      ScanFound       ;yes, we're done
                inc     si              ;point to the next byte
                jmp     ScanLoop        ;scan the next byte
ScanFound:
```

Although EA calculations can add faster than separate instructions can, it's faster still not to add at all. Whenever you can, perform your calculations outside loops.

Which brings us to **lea**.

## CALCULATING EFFECTIVE ADDRESSES WITH lea

**lea** is something of an odd bird, as the only mod-reg-rm memory-addressing instruction that doesn't access memory. **lea** calculates the offset of the memory operand...and then loads that offset into one of the 8 general-purpose registers, without accessing memory at all. Basically, **lea** is nothing more than a means by which to load the result of an EA calculation into a register.

For example, **lea bx,[MemVar]** loads the offset of **MemVar** into BX. Now, we wouldn't generally want to use **lea** to load simple offsets, since **mov** can do that more efficiently; **mov bx,offset MemVar** is 1 byte shorter and 4 cycles faster than **lea bx,[MemVar]**. (Since **lea** involves EA calculation, it's not particularly fast; however, it's faster than any mod-reg-rm memory-accessing instruction, taking only 2 cycles plus the EA calculation time.)

**lea** shines when you need to load a register with a complex memory address, preferably without disturbing any of the registers that make up the memory address. Suppose that we want to push the address of an array element that's indexed by BP+SI. We could use:

```
mov     ax,offset TestArray
add     ax,bp
add     ax,si
push    ax
```

which is 8 bytes long. On the other hand, we could simply use:

```
lea     ax,[TestArray+bp+si]
push    ax
```

which is only 5 bytes long. One of the primary uses of **lea** is loading offsets of variables in stack frames, because such variables are addressed with base+displacement addressing.

Refer back to the example we examined in the last section. Suppose that we wanted to scan memory without disturbing either BX or SI. In that case, we could use DI, with an assist from **lea**:

```
        lea     di,[bx+si] ;add together the memory address components
                        ; outside the loop
ScanLoop:
        cmp     [di],al     ;is this a match?
        jz      ScanFound           ;yes, we're done
        inc     di                  ;point to the next byte
        jmp     ScanLoop            ;scan the next byte
ScanFound:
```

**lea** is particularly handy in this case because it can add two registers--BX and SI--and place the result in a third register-- DI. That enables us to replace the two instructions:

```
mov     di,bx
add     di,si
```

with a single **lea**.

**lea** should make it clear that offsets are just 16-bit numbers. Adding offsets stored in BX and SI together with **lea** is no different from adding any two 16-bit numbers together with **add**, because offsets are just 16-bit numbers.   0 is a valid offset; if we execute:

```
sub     bx,bx           ;load BX with 0
mov     al,[bx]         ;load AL with the byte at offset 0 in DS
```

we'll read the byte at offset 0 in the segment pointed to by DS. It's important that you understand that offsets are just numbers, and that you can manipulate offsets every bit as flexibly as any other values.   The flip side is that you could, if you wished, add two registers and/or a constant value together with **lea** and place the result in a third register.   Of course, the registers would have to be BX or BP and SI or DI, but since offsets and numbers are one and the same, there's no reason that **lea** couldn't be used for arithmetic under the right circumstances.   For example, here's one way to add two memory variables and 52 together and store the result in DX:

```
mov     bx,[MemVar1]
mov     si,[MemVar2]
lea     dx,[bx+si+52]
```

That's not to say this is a good way to perform this particular task; the following is faster and uses fewer registers:

```
mov     dx,[MemVar1]
add     dx,[MemVar2]
add     dx,52
```

Nonetheless, the first approach does serve to illustrate the flexibility of **lea** and the equivalence of offsets and numbers.

## OFFSET WRAPPING AT THE ENDS OF SEGMENTS

Before we take our leave of mod-reg-rm addressing, I'd like to repeat a point made earlier that may have slipped past unnoticed. That point is that offsets wrap at the ends of segments. Offsets are 16-bit entities, so they're limited to the range 0 to 64 K-1. However, it is possible to use two or three mod-reg-rm address components that together add up to a number that's larger than 64 K. For example, the sum of the memory addressing components in the following code is 18000h:

```
mov     bx,4000h
mov     di,8000h
mov     ax,[bx+di+0c000h]
```

What happens in such a case? We found earlier that segments are limited to 64 Kb in length; is this a clever way to enlarge the effective size of a segment?

Alas, no. If the sum of two offset components won't fit in 16 bits, bits 16 and above of the sum are simply ignored. In other words, <u>mod-reg-rm</u> address calculations are always performed modulo 64 K (that is, modulo 10000h), as shown in Figure 7-11. As a result, the last example will access not the word at offset 18000h but the word at offset 8000h. Likewise, the following will access the byte at offset 0:

```
mov     bx,0ffffh
mov     dl,[bx+1]
```

The same rule holds for all memory-accessing instructions, <u>mod-reg-rm</u> or otherwise: <u>offsets are 16-bit values; any additional bits that result from address calculations are ignored</u>. Put another way, memory addresses that reach past the end of a segment's 64 K limit wrap back to the start of the segment. This allows the use of negative displacements, and is the reason a displacement can always reach anywhere in a segment, including addresses lower than those in the base and/or index registers, as in **mov ax,[bx-1]**.

## NON-<u>mod-reg-rm</u> MEMORY ADDRESSING

<u>mod-reg-rm</u> addressing is the most flexible memory addressing

mode of the 8088, and the most widely-used as well, but it's certainly not the only addressing mode. The 8088 also offers a number of specialized addressing modes, including stack addressing and the string instructions. These addressing modes are supported by fewer instructions than mod-reg-rm instructions, and are considerably more restrictive about the operands they'll accept--but they're also more compact and/or faster than the mod-reg-rm instructions.

Why are instructions that use the non-mod-reg-rm addressing modes generally superior to mod-reg-rm instructions? Simply this: being less flexible than mod-reg-rm instructions, they have fewer possible operands to specify, and so fewer instruction bits are needed. Non-mod-reg-rm instructions also don't require any EA calculation time, because they don't support the many addressing modes of the mod-reg-rm byte.

We'll discuss five sorts of non-mod-reg-rm memory-addressing instructions next: special forms of common instructions, string instructions, immediate-addressing instructions, stack-oriented instructions, and **xlat**, which is in a category all its own. For all these sorts of instructions, the rule is that if they're well matched to your application, they're almost surely worth using in preference to mod-reg-rm addressing. Some of the non-mod-reg-rm instructions, especially the string instructions, are so much faster than mod-reg-rm instructions that they're worth going out of your way for, as we'll see throughout The Zen of Assembly Language.

## SPECIAL FORMS OF COMMON INSTRUCTIONS

The 8088 offers special shorter, faster forms of several commonly used mod-reg-rm instructions, including **mov**, **inc**, and **xchg**. These special forms are both shorter and less flexible than the mod-reg-rm forms. For example, the special form of **inc** is just 1 byte long and requires only 2 cycles to execute, but can only work with 16-bit registers. By contrast, the mod-reg-rm form of **inc** is at least 2 bytes long and takes at least 3 cycles to execute, but can work with 8- or 16-bit registers or memory locations.

You don't have to specify that a special form of an instruction is to be used; the assembler automatically selects the shortest possible form of each instruction it assembles. That doesn't mean that you don't need to be familiar with the special forms, however. To the contrary, you need to be well aware of the sorts of instructions that have special forms, as well as the circumstances under which those special forms will be assembled. Armed with that knowledge, you can arrange your code so that the special forms will be assembled as often as possible.

We'll get a solid feel for the various special forms of mod- reg-rm instructions as we discuss them individually in Chapters 8 and 9.

## THE STRING INSTRUCTIONS

The string instructions are without question the most powerful instructions of the 8088. String instructions can initialize, copy, scan, and compare arrays of data at speeds far beyond those of mortal mod-reg-rm instructions, and lend themselves well to almost any sort of repetitive processing. In fact, string instructions are so important that they get two full chapters of The Zen of Assembly Language--Chapters 10 and 11--to themselves. We'll defer further discussion of these extremely important instructions until then.

## IMMEDIATE ADDRESSING

Immediate addressing is a form of memory addressing in which the constant value of one operand is built right into the instruction. You should think of immediate operands as being addressed by IP, since they directly follow opcode bytes or mod- reg-rm bytes, as shown in Figure 7-12.

Instructions that use immediate addressing are clearly faster than instructions that use mod-reg-rm addressing. In fact, according to official execution times, immediate addressing would seem to be much faster than mod-reg-rm addressing. For example, **add ax,1** is a 4-cycle instruction, while **add ax,[bx]** is an 18-cycle instruction. What's more, **add reg,immed** is just 1 cycle slower than **add reg,reg**, so immediate addressing seems to be nearly as fast as register addressing.

The official cycle counts are misleading, however. While immediate addressing is certainly faster than mod-reg-rm addressing, it is by no means as fast as register-only addressing, and the reason is a familiar one: the prefetch queue cycle-eater. You see, immediate operands are instruction bytes; when we use an immediate operand, we increase the size of that instruction, and that increases the number of cycles needed to fetch the instruction's bytes.

Looked at another way, immediate operands need to be fetched from the memory location pointed to by IP, so immediate addressing could be considered a memory addressing mode. Granted, immediate addressing is an efficient memory addressing mode, with no EA calculation time or the like--but memory accesses are nonetheless required, at the inescapable 4 cycles per byte.

The upshot is simply that register operands are superior to immediate operands in loops and time-critical code, although immediate operands are still much better than mod-reg-rm memory operands. Back in Listing 7-11, we set DL to 0 outside the loop so that we could use register-register **adc** inside the loop. That approach allowed the code to run in 0.95 ms. Listing 7-12 is similar to Listing 7-11, but is modified to use an immediate operand of 0 rather than a register operand containing 0. Even though the immediate operand is only byte-sized, Listing 7-12 slows down to 1.02 ms. In other words, the need to fetch just 1 immediate operand byte every time through the loop slowed the entire loop by about 7%. What's more, the performance loss would have been approximately twice as great if we had used a word- sized immediate operand.

On the other hand, immediate operands are certainly preferable to memory operands. Listing 7-13, which adds the constant value 0 from memory, runs in 1.26 ms. (I should hope you'll never use code as obviously inefficient as Listing 7-13; I'm just presenting it for illustrative purposes.)

To sum up: when speed matters, use register operands rather than immediate operands if you can. If registers are at a premium, however, immediate operands are reasonably fast, and are certainly better than memory operands. If bytes rather than cycles are at a premium, immediate operands are excellent, for it takes fewer bytes to use an immediate operand than it does to load a register with a constant value and then use that register. For example:

```
LoopTop:
            or      byte ptr [bx],80h
            loop    LoopTop
```

is 1 byte shorter than:

```
            mov     al,80h
LoopTop:
            or      [bx],al
            loop    LoopTop
```

However, the latter, register-only version is faster, because it moves 2 bytes out of the loop.

There are many circumstances in which we can substitute register-only instructions for instructions that use immediate operands without adding any extra instructions. The commonest of these cases involve testing for zero. There's almost never a need to compare a register to zero; instead, we can simply **and** or **or** the register with itself and check the resulting flags. We'll discuss ways to handle zero in the next two chapters, and we'll see similar cases in which immediate operands can be eliminated throughout The Zen of Assembly Language.

By the way, you should be aware that you can use an immediate operand even when the other operand is a memory variable rather than a register. For example, **add [MemVar],16** is a valid instruction, as is **mov [MemVar],52**. As I mentioned earlier, we're better off performing single operations directly to memory than we are loading from memory into a register, operating on the register, and storing the result back to memory. However, we're generally better off working with a register when multiple operations are involved.

Ideally, we'd load a memory value into a register, perform multiple operations on it there, store the result back to memory...and then have some additional use for the value left in the register, thereby getting double use out of our memory accesses. For example, suppose that we want to perform the equivalent of the C statement:

```
i = ++j + k;
```

We could do this as follows:

```
inc     [j]
mov     ax,[j]
add     ax,[k]
mov     [i],ax
```

However, we can eliminate a memory access by incrementing **j** in a register:

```
mov     ax,[j]
inc     ax
mov     [j],ax
add     ax,[k]
```

```
        mov     [i],ax
```

While the latter version is one instruction longer than the original version, it's actually faster and shorter. One reason for this is that we get double use out of loading **j** into AX; we increment **j** in AX and store the result to memory, then immediately use the incremented value left in AX as part of the calculation being performed.

The other reason the second example above is superior to the original version is that it used two of the special, more efficient instruction forms: the accumulator-specific direct- addressed form of **mov** and the 16-bit register-only form of **inc**. We'll study these instructions in detail in Chapters 8 and 9.

## SIGN-EXTENSION OF IMMEDIATE OPERANDS

I've already noted that immediate operands tend to make for compact code. One key to this property is that like displacements in mod-reg-rm addressing, word-sized immediate operands can be stored as a byte and then extended to a word by replicating bit 7 as bits 15-8; that is, word-sized immediate operands can be sign-extended. Almost all instructions that support immediate operands allow word-sized operands in the range -128 to +127 to be stored as single bytes. That means that while **and dx,1000h** is a 4-byte instruction (1 opcode byte, 1 mod-reg-rm byte, and a 2-byte immediate operand), **and dx,0fffeh** is just 3 bytes long; since the signed value of the immediate operand 0FFFEh is -2, 0FFFEh is stored as a single immediate operand byte.

Not all values of the form 000<u>nn</u>h and 0FF<u>nn</u>h (where <u>nn</u> is any two hex digits) can be stored as a single byte and sign- extended. 0007Fh can be stored as a single byte; 00080h cannot. 0FF80h can be stored as a single byte; 0FF7Fh cannot. Watch out for cases where you're using a word-sized immediate operand that can't be stored as a byte, when a byte-sized immediate operand would serve as well.

For example, suppose we want to set the lower 8 bits of DX to 0. **and dx,0ff00h** is a 4-byte instruction that accomplishes the desired result. **and dl,000h** produces the same result in just 3 bytes. (Of course, **sub dl,dl** does the same thing in just 2 bytes--there are <u>many</u> ways to skin a cat in assembler.) Recognizing when a word-sized immediate operand can be handled as a byte-sized operand is still more important when using accumulator-specific immediate-operand instructions, which we'll explore in the next chapter.

## mov DOESN'T SIGN-EXTEND IMMEDIATE OPERANDS

Along the same lines, **or bh,0ffh** does the same thing as **or bx,0ff00h** and is shorter, while **mov bh,0ffh** is also equivalent and is shorter still...and that brings us to the one instruction which cannot sign-extend immediate operands: **mov**. Word-sized operands to **mov** are always stored as words, no matter what size they may be. However, there's a compensating factor, and that's that there's a special, non-<u>mod-reg-rm</u> form of **mov <u>reg,immed</u>** that's 1 byte shorter than the <u>mod-reg-rm</u> form.

Let me put it this way. **and dx,1000h** is a 4-byte instruction, with 1 opcode byte, 1 mod-reg-rm byte, and a 2-byte immediate operand. **mov dx,1000h**, on the other hand, is only 3 bytes long. There's a special form of the **mov** instruction, used only when a register is loaded with an immediate value, that requires just the 1 opcode byte in addition to the immediate value.

There's also the standard mod-reg-rm form of **mov**, which is 4 bytes long for word-sized immediate operands. This form does exactly the same thing as the special form, but is a different instruction, with a different opcode and a mod-reg-rm byte. The 8088 offers a number of duplicate instructions, as we'll see in the next chapter. Don't worry about selecting the right form of **mov**, however; the assembler does that for you automatically.

In short, you're no worse off--and often better off--moving immediate values into registers than you are using immediate operands with instructions such as **add** and **xor**. It takes just 2 or 3 bytes, for byte- or word-sized registers, respectively, to load a register with an immediate operand. **mov al,2** is actually the same size as **mov al,bl** (both are 2 bytes), although the official execution time of the register-only **mov** is 2 cycles shorter.

On balance, immediate operands used with **mov reg,immed** perform at nearly the speed of register operands, especially when the register is byte-sized; consequently, there's less need to avoid immediate operands with **mov** than with other instructions. Nonetheless, register-only instructions are never slower, so you won't go wrong using register rather than immediate operands.

## DON'T mov IMMEDIATE OPERANDS TO MEMORY IF YOU CAN HELP IT

One final note, and then we're done with immediate addressing. There is no special form of **mov** for moving an immediate operand to a memory operand; the special form is limited to register operands only. What's more, **mov [mem16],immed16** has no sign-extension capability. This double whammy means that storing immediate values to memory is the single least desirable way to use immediate operands. Over the next few chapters, we'll explore several ways to set memory operands to given values. The one thing that the various approaches have in common is that they all improve performance by avoiding immediate operands to **mov**.

Don't move immediate values to memory unless you have no choice.

## STACK ADDRESSING

While SP can't be used to point to memory by mod-reg-rm instructions, it is nonetheless a memory-addressing register. After all, SP is used to address the top of the stack. Surely you know how the stack works, so I'll simply note that SP points to the data item most recently pushed onto the top of the stack that has not yet been popped off the stack. Consequently, stack data can only be accessed in Last In, First Out (LIFO) order via SP (that is, the order in which data is popped off the stack is the reverse of the order in which it was pushed on). However, other addressing modes--in particular mod-reg-rm BP-based addressing--can be used to access stack data in non-LIFO order, as we'll see when we discuss stack frames.

What's so great about the stack? Simply put, the stack is terrific for temporary storage. Each named memory variable, as in:

```
MemVar          dw      0
```

takes up 1 or more bytes of memory for the duration of the program. That's not the case with stack data, however; when data is popped from the stack, the space it occupied is freed up for other use. In other words, stack memory is a reusable resource. This makes the stack an excellent place to store temporary data, especially when large data elements such as buffers and structures are involved.

Space allocated on the stack is also unique for each invocation of a given subroutine, which is useful for any subroutine that needs to be capable of being called directly or indirectly from itself. Stack-based storage is how C implements automatic (dynamic) variables, which are unique for each invocation of a given subroutine. In fact, stack-based storage is the heart of the parameter-passing mechanism used by most C implementations, as well as the mechanism used for automatic variables, as we'll see shortly.

Don't underestimate the flexibility of the stack. I've heard of programs that actually compile code right into a buffer on the stack, then execute that code in place, on the stack. While that's a strange concept, stack memory is memory like any other, and instruction bytes are data;

obviously, those programs needed a temporary place in which to compile code, run it, and discard it, and the stack fits those requirements nicely.

Similarly, suppose that we need to pass a pointer to a variable from an assembler program to a C subroutine...but there's no variable to point to in the assembler code, because we keep the variable in a register. Suppose also that the C subroutine actually modifies the pointed-to variable, so we need to retrieve the altered value after the call. The stack is admirably suited to the job; at the beginning of the following code, the variable of interest is in DX, and that's just where the modified result is at the end of the code:

```
;
; Calls: int CSubroutine(int *Count, char *BufferPointer).
;
        mov     dx,MAX_COUNT    ;store the maximum # of bytes to handle
                                ; in the count variable
        push    dx              ;store the count variable on the stack
                                ; for the duration of the call
        mov     dx,sp           ;put a pointer to the just-pushed temporary
                                ; count variable in DX
        mov     ax,offset TestBuffer
        push    ax              ;pass the buffer pointer parameter
        push    dx              ;pass the count pointer parameter
        call    CSubroutine     ;do the count
        add     sp,4            ;clear the parameter bytes from the stack
        pop     dx              ;get the actual count back into DX
```

The important point in the above code is that we created a temporary memory variable on the stack as we needed it; then, when the call was over, we simply popped the variable back into DX, and its space on the stack was freed up for other use. The code is compact, and not a single byte of memory storage had to be reserved permanently.

Compact code without the need for permanent memory space is the hallmark of stack-based code. It's often possible to write amazingly complex code without using mod-reg-rm addressing or named variables simply by pushing and popping registers. The code tends to be compact because **push** **reg16** and **pop** **reg16** are each only 1 byte long. **push** **reg16** and **pop** **reg16** are so compact because they don't need to support the complex memory-addressing options of mod-reg-rm addressing; there are only 8 possible register operands, and each instruction can only address one location, by way of the stack pointer, at any one time. (**push** **mem16** and **pop** **mem16** are mod-reg-rm instructions, and so they're 2-4 bytes long; **push** **reg16** and **pop** **reg16**, and **push** **segreg** and **pop** **segreg** as well, are special, shorter forms of **push** and **pop**.)

For once, though, shorter isn't necessarily better. You see, **push** and **pop** are memory-accessing instructions, and although they don't require EA calculation time, they're still slow--like all instructions that access memory. **push** and **pop** are fast considering that they are word-sized memory-accessing instructions--**push** takes 15 cycles, **pop** takes just 12--and they make for good prefetching, since only 3 memory accesses (including instruction fetches) are performed during an official execution time of 12 to 15 cycles. Nonetheless, they're clearly slower than register-only instructions. This is basically the same case we studied when we looked into copying segments; it's faster but takes more bytes and requires a free register to preserve a register by copying it to another register:

```
	mov	dx,ax
```

```
  :
mov         ax,dx
```

than it is to preserve it by pushing and popping it:

```
push        ax
  :
pop         ax
```

What does all this mean to you? Simply this: use a free register for temporary storage if speed is of the essence, and **push** and **pop** if code size is your primary concern, if speed is not an issue, or if no registers happen to be free. In any case, it's faster and far more compact to store register values temporarily by pushing and popping them than it is to store them to memory with mod-reg-rm instructions. So use **push** and **pop**...but remember that they come with substantial performance overhead relative to register-only instructions.

## AN EXAMPLE OF AVOIDING push AND pop

Let's quickly look at an example of improving performance by using register-only instructions rather than **push** and **pop**. When copying images into display memory, it's common to use code like:

```
;
; Copies an image into display memory.
;
; Input:
;               BX = width of image in bytes
;               DX = height of image in lines
;               BP = number of bytes from the start of one line to the
;                start of the next
;               DS:SI = pointer to image to draw
;               ES:DI = display memory address at which to draw image
;                 Direction flag must be cleared on entry
;
; Output:
;                 none
;
DrawLoop:
                  push     di                     ;remember where the line starts
                  mov      cx,bx          ;# of bytes per line
                  rep      movsb          ;copy the next line
                  pop      di                     ;get back the line start offset
                  add      di,bp          ;point to the next line in display memory
                  dec      dx                     ;repeat if there are any more lines
                  jnz      DrawLoop
```

That's fine, but 1 **push** and 1 **pop** are performed per line, which seems a shame...all the more so given that we can eliminate those pushes and pops altogether, as follows:

```
;
; Copies an image into display memory.
;
; Input:
;                 BX = width of image in bytes
;                 DX = height of image in lines
;                 BP = number of bytes from the start of one line to the
;                  start of the next
;                 DS:SI = pointer to image to draw
;                 ES:DI = display memory address at which to draw image
;                 Direction flag must be cleared on entry
;
; Output:
;                 none
;
                  sub      bp,bx          ;# of bytes from the end of 1 line of the
                                          ; image in display memory to the start of
                                          ; the next line of the image
DrawLoop:
                  mov      cx,bx          ;# of bytes per line
                  rep      movsb          ;copy the next line
                  add      di,bp          ;point to the next line in display memory
                  dec      dx                     ;repeat if there are any more lines
                  jnz      DrawLoop
```

Do you see what we've done?  By converting an obvious solution (advancing 1 full line at a time) to a less-obvious but fully equivalent solution (advancing only the remaining portion of the line), we've saved about 27 cycles per loop...at no cost. Given inputs like the width of the screen and instructions like **push** and **pop**, we tend to use them; it's just human nature to frame solutions in familiar terms. By rethinking the problem just a little, however, we can often find a simpler, better solution.

Saving 27 cycles not by knowing more instructions but by not using two powerful instructions is an excellent example indeed of the Zen of assembler.

## MISCELLANEOUS NOTES ABOUT STACK ADDRESSING

Before we proceed to stack frames, I'd like to take a moment to review a few important points about stack addressing.

SP always points to the next item to be popped from the stack. When you push a value onto the stack, SP is first decremented by 2, and then the value is stored at the location pointed to by SP. When you pop a value off of the stack, the value is read from the location pointed to by SP, and then SP is incremented by 2. It's useful to know this whenever you need to point to data stored on the stack, as we did when we created and pointed to a temporary variable on the stack a few sections back, and as we will need to do when we work with stack frames.

**push** and **pop** can work with mod-reg-rm-addressed memory variables as easily as with registers, albeit more slowly and with more instruction bytes. **push [WordVar]** is perfectly legitimate, as is **pop word ptr [bx+si+100h]**. Bear in mind, however, that only 16-bit values can be pushed and popped; **push bl** won't work, and neither will **pop byte ptr [bx]**.

Finally, please remember that once you've popped a value from the stack, it's gone from memory. It's tempting to look at the way the stack pointer works and think that the data is still in memory at the address just below the new stack pointer, but that's simply not the case, as shown in Figure 7-13. Sure, sometimes the data is still there--but whenever an interrupt occurs, it uses the top of the stack, wiping out the values that were most recently popped. Interrupts can happen at any time, so unless you're willing to disable interrupts, accessing popped stack memory is a sure way to get intermittent bugs.

Even if interrupts are disabled, it's really not a good idea to access popped stack data. Why bother, when stack frames give you the same sort of access to stack data, but in a straightforward, risk-free way? Not coincidentally, stack frames are our next topic, but first let me emphasize: once you've popped data off the stack, it's gone from memory. Vanished. Kaput. Extinct. For all intents and purposes, that data is nonexistent.

Don't access popped stack memory. Period.

## STACK FRAMES

Stack frames are transient data structures, usually local to specific subroutines, that are stored on the stack. Two sorts of data are normally stored in stack frames: parameters that are passed from the calling routine by being pushed on the stack, and variables that are local to the subroutine using the stack frame.

Why use stack frames? Well, as we discussed earlier, the stack is an excellent place to store temporary data, a category into which both passed parameters and local storage fall. **push** and **pop** aren't good for accessing stack frames, which often contain many variables and which aren't generally accessed in LIFO order; however, there are several mod-reg-rm addressing modes that are perfect for accessing stack frames--the mod-reg-rm addressing modes involving BP. (We can't use SP for two reasons: it can't serve as a memory pointer with mod-reg-rm addressing modes, and it changes constantly during code execution, making offsets from SP hard to calculate.)

If you'll recall, BP-based addressing modes are the only mod-reg-rm addressing modes that don't access DS by default. BP- based addressing modes access SS by default, and now we can see why--in order to access stack frames. Typically, BP is set to equal the stack pointer at the start of a subroutine, and is then used to point to data in the stack frame for the remainder of the subroutine, as in:

```
push    bp              ;save caller's BP
mov     bp,sp           ;point to stack frame
mov     ax,[bp+4]       ;retrieve a parameter
 :
pop     bp              ;restore caller's BP
ret
```

If temporary local storage is needed, SP is moved to allocate the necessary room:

```
push    bp              ;save caller's BP
mov     bp,sp           ;point to stack frame
sub     sp,10           ;allocate 10 bytes of local storage
```

```
mov     ax,[bp+4]           ;retrieve a parameter
mov     [bp-2],ax ;save it in local storage
 :
mov     sp,bp     ;dump the temporary storage
pop     bp                  ;restore caller's BP
ret
```

I'm not going to spend a great deal of time on stack frames, for one simple reason: they're not all that terrific in assembler code. Stack frames are ideal for high-level languages, because they allow regular parameter-passing schemes and support dynamically allocated local variables. For assembler code, however, stack frames are quite limiting, in that they require a single consistent parameter-passing convention and the presence of code to create and destroy stack frames at the beginning and end of each subroutine. In particular, the ability of assembler code to pass pointers and variables in registers (which is much more efficient than pushing them on the stack) is constrained by standard stack frames conventions. In addition, the BP register, which is dedicated to pointing to stack frames, normally cannot be used for other purposes when stack frames are used; the loss of one of a mere seven generally-available 16-bit registers is not insignificant.

High-level language stack frame conventions also generally mandate the preservation of several registers--always BP, usually DS, and often SI and DI as well--and that requires time-consuming pushes and pops. Finally, while stack frame addressing is compact (owing to the heavy use of **bp+disp** addressing with 1-byte displacements), it is rather inefficient, even as memory- accessing instructions go; **mov ax,[bp+disp8]** is only 3 bytes long, but takes 21 cycles to execute.

In short, stack frames are powerful and useful--but they don't make for the best possible 8088 code. The best compiled code, yes, but not the best assembler code.

What's more, compilers handle stack frames very efficiently. If you're going to work within the constraints of stack frames, you may have a difficult time out-coding compilers, which rarely miss a trick in terms of generating efficient stack frame code. Handling stack frames well is not so simple as it might seem; you have to be sure not to insert unneeded stack-frame-related code, such as code to load BP when there is no stack frame, and you need to be sure that you always preserve the proper registers when they're altered, but not otherwise. It's not hard, but it's tedious, and it's easy to make mistakes that either waste bytes or lead to bugs as a result of registers that should be preserved but aren't.

When you work with stack frames, you're trying to out- compile a compiler while playing by its rules, and that's hard to do. In pure assembler code, I generally recommend against the use of stack frames, although there are surely exceptions to this rule. Personally, I often use C for the sort of code that requires stack frames, building only the subroutines that do the time-critical work in pure assembler. Why not let a compiler do the dirty work, while you focus your efforts on the code that really makes a difference?

## WHEN STACK FRAMES ARE USEFUL

That's not to say that stack frames aren't useful in assembler.

Stack frames are not only useful but mandatory when assembler subroutines are called from high-level language code, since the stack frame approach is the sole parameter-passing mechanism for most high-level language implementations.

Assembler subroutines for use with high-level languages are most useful; together, assembler subroutines and high-level languages provide relatively good performance and fast development time. The best code is written in assembler, but the best code within a reasonable time frame is often written in a high-level language/assembler hybrid. Then, too, high-level languages are generally better than assembler for managing the complexities of very large applications.

In short, stack frames are generally useful in assembler when assembler is interfaced to a high-level language. High- level language interfacing and stack frame organization varies from one language to another, however, so I'm not going to cover stack frames in detail, although I will offer a few tips about using stack frames in the next section. Before I do that, I'd like to point out an excellent way to mix assembler with high- level language code: in-line assembler. Many compilers offer the option of embedding assembler code directly in high-level language code; in many cases, high-level language and assembler variables and parameters can even be shared. For example, here's a Turbo C subroutine to set the video mode:

```
void SetVideoMode(unsigned char ModeNumber) {
                asm     mov     ah,0
                asm     mov     al,byte ptr [ModeNumber]
                asm     int     10h
```

```
    }
```

What makes in-line assembler so terrific is that it lets the compiler handle all the messy details of stack frames while freeing you to use assembler. In the above example, we didn't have to worry about defining and accessing the stack frame; Turbo C handled all that for us, saving and setting up BP and substituting the appropriate BP+disp value for **ModeNumber**. In- line assembler is harder to use for large tasks than is pure assembler, but in most cases where the power of assembler is needed in a high-level language, in-line assembler is a very good compromise.

One warning: many compilers turn off some or all code optimization in subroutines that contain in-line assembler. For that reason, it's often a good idea not to mix high-level language and in-line assembler statements when performance matters. Write your time-critical code either entirely in in- line assembler or entirely in pure assembler; don't let the compiler insert code of uncertain quality when every cycle counts.

Still and all, when you need to create the fastest or tightest code, try to avoid stack frames except when you must interface your assembler code to a high-level language. When you must use stack frames, bear in mind that assembler is infinitely flexible; there are more ways to handle stack frames than are dreamt of in high-level languages. In Chapter 16 we'll see an unusual but remarkably effective way to handle stack frames in a Pascal-callable assembler subroutine.

## TIPS ON STACK FRAMES

Before we go on to **xlat**, I'm going to skim over a few items that you may find useful should you need to use stack frames in assembler code.

MASM provides the **struc** directive for defining data structures. Such data structures can be used to access stack frames, as in:

```
Parms           struc
                dw      ?       ;pushed BP
                dw      ?       ;return address
X               dw      ?       ;X coordinate parameter
Y               dw      ?       ;Y coordinate parameter
Parms           end
                 :
DrawXY          proc    near
                push    bp              ;save caller's stack frame pointer
                mov     bp,sp   ;point to stack frame
                mov     cx,[bp+X]       ;get X coordinate
                mov     dx,[bp+Y]       ;get Y coordinate
                 :
                pop     bp
                ret
DrawXY          endp
```

MASM structures have a serious drawback when used with stack frames, however: they don't allow for negative displacements from BP, which are generally used to access local variables stored on the stack. While it is possible to access local storage by accessing all variables in the stack frames at positive offsets from BP, as in:

```
Parms           struc
Temp            dw      ?       ;temporary storage
OldBP dw        ?       ;pushed BP
                dw      ?       ;return address
X               dw      ?       ;X coordinate parameter
Y               dw      ?       ;Y coordinate parameter
Parms           end
                 :
DrawXY          proc    near
                push    bp              ;save caller's stack frame pointer
                sub     sp,OldBP ;make room for temp storage
                mov     bp,sp   ;point to stack frame
                mov     cx,[bp+X]           ;get X coordinate
                mov     dx,[bp+Y]           ;get Y coordinate
                mov     [bp+Temp],dx ;set aside Y coordinate
                  :
                add     sp,OldBP ;dump temp storage space
                pop     bp
                ret
DrawXY          endp
```

this approach has two disadvantages. First, it prevents us from dumping temporary storage with **mov sp,bp**, requiring instead that we use the less efficient **add sp,OldBP**. Second, and more important, it makes it more likely that parameters will be accessed with a 2-byte displacement.

Why? Remember that a 1-byte displacement can address memory in the range -128 to +127 bytes away from BP. If our entire stack frame is addressed at positive offsets from BP, then we've lost the use of a full one-half of the addresses that we can access with 1-byte displacements.

Now, we <u>can</u> use negative stack frame offsets in assembler; it's just a bit more trouble than we'd like. There are many possible solutions, ranging from a variety of ways to use equated symbols for stack frame variables, as in:

```
Temp            equu    -2      ;temporary storage
X               equ     4       ;X coordinate parameter
Y               equ     6       ;Y coordinate parameter
```

and:

```
Temp            equ     -2      ;temporary storage
X               equ     4       ;X coordinate parameter
Y               equ     X+2     ;Y coordinate parameter
```

up to ways to get the assembler to adjust structure offsets for us. See my "On Graphics" column in the July 1987 issue of Programmer's Journal (issue 5.4) for an elegant solution, provided by John Navas. (Incidentally, TASM provides special directives--**arg** and **local**--that handle many of the complications of stack frame addressing and allow negative offsets.)

While we're discussing stack frame displacements, allow me to emphasize that you should strive to use 1-byte displacements into stack frames as much as possible. If you have so many parameters or local variables that 2-byte displacements must be used, make an effort to put the least frequently used variables at those larger displacements. Alternatively, you may want to put large data elements such as arrays and structures in the stack frame areas that are addressed with 2-byte displacements, since such data elements are often accessed by way of pointer registers such as BX and SI, rather than directly via **bp+disp** addressing. Finally, you should avoid forward references to structures; if you refer to elements of a structure before the structure itself is defined in the code, you'll always get 2- byte displacements, as we'll see in Chapter 14.

Whenever you're uncertain whether 1- or 2-byte displacements are being used, simply generate a listing file, or look at your code with a debugger.

By the way, it's worth examining the size of your stack frame displacements even in high-level languages. If you can figure out the order in which your compiler organizes data in a stack frame, you can often speed up and shrink your code simply by reorganizing your local variable declarations so that arrays and structures are at 2-byte offsets, allowing most variables to be addressed with 1-byte offsets.

## STACK FRAMES ARE OFTEN IN DS

While it's not always the case, often enough the stack segment pointed to by SS and the default data segment pointed to by DS are one and the same. This is true in most high-level language memory models, and is standard for COM programs.

If DS and SS are the same, the implication is clear: all mod-reg-rm addressing modes can be used to point to stack frames. That's a real advantage if you need to scan stack frame arrays and the like, because SI or DI can be loaded with the array start address and used to address the array without the need for segment override prefixes. Similarly, BX could be set to point to a stack frame structure, which could then be accessed by way of **bx+disp** addressing without a segment override. In short, be sure to take advantage of the extra stack frame addressing power that you have at your disposal when SS equals DS.

## USE BP AS A NORMAL REGISTER IF YOU MUST

When stack frame addressing is in use, BP is normally dedicated to addressing the current stack frame. That doesn't mean you can't use BP as a normal register in a tight loop, though, and use it as a normal register you should; registers are too scarce to let even one go to waste when performance matters. Just push BP, use it however you wish in the loop, then pop it when you're done, as in:

```
                push    bp                      ;preserve stack frame pointer
                mov     bp,LOOP_COUNT   ;get # of times to repeat loop
LoopTop:
                 :
                dec     bp                      ;count off loops
                loop    LoopTop
                pop     bp                      ;restore stack frame pointer
```

Of course, the stack frame can't be accessed while BP is otherwise occupied, but you don't want to be accessing memory inside a tight loop anyway if you can help it.

Using BP as a normal register in a tight loop can make the difference between a register-only loop and one that accesses memory operands, and that can translate into quite a performance improvement. Also, don't forget that BP can be used in mod-reg- rm addressing even when stack frames aren't involved, so BP can come in handy as a memory-addressing register when BX, SI, and DI are otherwise engaged. In that usage, however, bear in mind that there is no BP-only memory addressing mode; either a 1- or 2-byte displacement or an index register (SI or DI) or both is always involved.

## THE MANY WAYS OF SPECIFYING mod-reg-rm ADDRESSING

There are, it seems, more ways of specifying an operand addressed with mod-reg-rm addressing than you can shake a stick at. For example, **[bp+MemVar+si]**, **MemVar[bp+si]**, **MemVar[si][bp]**, and **[bp]**

**[MemVar+si]** are all equivalent. Now stack frame addressing introduces us to a new form, involving the dot operator: **[bp.MemVar+si]**. Or [**bp.MemVar.si].** What's the story with all these mod-reg-rm forms?

It's actually fairly simple. The dot operator does the same thing as the plus operator: it adds two memory addressing components together. Any memory-addressing component enclosed in brackets is also added into the memory address. The order of the operands doesn't matter, since everything resolves to a mod-reg- rm byte in the end; **mov al,[bx+si]** assembles to exactly the same instruction as **mov al,[si+bx]**. All the constant values and symbols (variable names and equated values) in an address are added together into a single displacement, and that's used with whatever memory addressing registers are present (from among BX, BP, SI, and DI) to form a mod-reg-rm address. (Of course, only valid combinations--the combinations listed in Figure 7-6--will assemble.) Lastly, if memory addressing registers are present, they must be inside square brackets, but that's optional for constant values and symbols.

There are a few other rules about constructing memory addressing operands, but I avoid those complications by making it a practice to use a single simple mod-reg-rm memory address notation. As I said at the start of this chapter, I prefer to put square brackets around all memory operands, and I also prefer to use only the plus operator. There are three reasons for this: it's not complicated, it reminds me that I'm programming in assembler, not in a high-level language where complications such as array element size are automatically taken care of, and it reminds me that I'm accessing a memory operand rather than a register operand, thereby losing performance and gaining bytes.

You can use whatever mod-reg-rm addressing notation you wish. I do suggest, however, that you choose a single notation and stick with it. Why confuse yourself?

## xlat

At long last, we come to the final addressing mode of the 8088. This addressing mode is unique to the **xlat** instruction, an odd and rather limited instruction that can nonetheless outperform every other 8088 instruction under the proper circumstances.

The operation of **xlat** is simple: AL is loaded from the offset addressed by the sum of BX and AL, as shown in Figure 7- 14. DS is the default data segment, but a segment override prefix may be used.

As you can see, **xlat** bears no resemblance to any of the other addressing modes. It's certainly limited, and it always wipes out one of the two registers it uses to address memory (AL). In fact, the first thought that leaps to mind is: why would we ever want to use **xlat**?

If **xlat** were slow and large, the answer would be never. However, **xlat** is just 1 byte long, and, at 10 cycles, is as fast at accessing a memory operand as any 8088 instruction. As a result, **xlat** is excellent for a small but often time-critical category of tasks.

**xlat** excels when byte values must be translated from one representation to another. The most common example occurs when one character set must be translated to another, as for example when the ASCII character set used by the PC is translated to the EBCDIC character set used by IBM mainframes. In such a case **xlat** can form the heart of an extremely efficient loop, along the lines of the following:

```
;
; Converts the contents of an ASCII buffer to an EBCDIC buffer.
; Stops when a zero byte is encountered, but copies the zero byte.
;
; Input:
;               DS:SI = pointer to ASCII buffer.
;
; Output: none
;
; Registers altered: AL, BX, SI, DI, ES
;
                mov     di,ds
                mov     es,di
                mov     di,si           ;point ES:DI to the ASCII buffer as well
                mov     bx,offset ASCIIToEBCDICTable
                                        ;point to the table containing the EBCDIC
                                        ; equivalents of ASCII codes
                cld
ASCIIToEBCDICLoop:
                lodsb                   ;get the next ASCII character
                xlat                            ;convert it to EBCDIC
                stosb                   ;put the result back in the buffer
                and     al,al           ;zero byte is the last byte
                jnz     ASCIIToEBCDICLoop
```

Besides being small and fast, **xlat** has an advantage in that byte-sized look-up values don't need to be converted to words before they can be used to address memory. (Remember, mod-reg-rm addressing modes allow only word-sized registers to be used to address memory.) If we were to implement the look-up in the last example with mod-reg-rm instructions, the code would become a good deal less efficient no matter how efficiently we set up for mod-reg-rm addressing:

```
                sub     bh,bh           ;for use in converting a byte in BL
                                        ; to a word in BX
                mov     si,offset ASCIIToEBCDICTable
                                        ;point to the table containing the EBCDIC
                                        ; equivalents of ASCII codes
ASCIIToEBCDICLoop:
                lodsb                   ;get the next ASCII character
                mov     bl,al           ;get the character into BX, where
                                        ; we can use it to address memory
                mov     al,[si+bx] ;convert it to EBCDIC
                stosb                   ;put the result back in the buffer
                and     al,al           ;zero byte is the last byte
                jnz     ASCIIToEBCDICLoop
```

In short, **xlat** is clearly superior when a byte-sized look-up is performed, so long as it's possible to put both the look-up value and the result in AL. Shortly, we'll see how **xlat** can be used to good effect in a case where it certainly isn't the obvious choice.

## MEMORY IS CHEAP: YOU COULD LOOK IT UP

**xlat**, simply put, is a table look-up instruction. A table look-up occurs whenever you use an index value to look up a result in an array, or table, of data. A rough analogy might be using the number on a ballplayer's uniform to look up his name in a program.

Look-up tables are a superb way to improve performance. The basic premise of look-up tables is that it's faster to precalculate results, either by letting the assembler do the work or by calculating the results yourself and inserting them in the source code, than it is to have the 8088 calculate them at run time. The key factor is this: the 8088 is relatively fast at looking up data in tables and slow at performing almost any kind of calculation. Given that, why not perform your calculations before run time, when speed doesn't matter, and let the 8088 do what it does best at run time?

Now, look-up tables do have a significant disadvantage--they require extra memory. This is a trade-off we'll see again and again in The Zen of Assembly Language: cycles for bytes. If you're willing to expend more memory, you can almost always improve the performance of your code. One trick to generating top-notch code is knowing when that trade-off is worth making.

Let's look at an example that illustrates the power of look- up tables. In the process, we'll see an unusual but effective use of **xlat**; we'll also see that there are many ways to approach any programming task, and we'll get a first-hand look at the cycles-for-bytes tradeoff that arises so often in assembler programming.

## FIVE WAYS TO DOUBLE BITS

The example we're about to study is based on the article "Optimizing for Speed," by Michael Hoyt, which appeared in Programmer's Journal in March, 1986 (issue 4.2). This is the article I referred to back in Chapter 2 as an example of a programmer operating without full knowledge about code performance on the PC. By no means am I denigrating Mr. Hoyt; his article simply happens to be an excellent starting point for examining both look-up tables and the hazards of the prefetch queue cycle-eater.

The goal of Mr. Hoyt's article was to expand a byte to a word by doubling each bit, for the purpose of converting display memory pixels to printer pixels in order to perform a screen dump. So, for example, the value 01h (00000001b) would become 0003h (0000000000000011b), the value 02h (00000010b) would become 000Ch (0000000000001100b), and the value 5Ah (01011010b) would become 33CCh (0011001111001100b). Now, in general this isn't a particularly worthy pursuit, given that the speed of the printer is likely to be the limiting factor; however, speed could matter if the screen dump code is used by a background print spooler. At any rate, bit-doubling is an ideal application for look-up tables, so we're going to spend some time studying it.

Mr. Hoyt started his article with code that doubled each bit by testing that bit and branching accordingly to set the appropriate doubled bit values. He then optimized the code by eliminating branches entirely, instead using fast shift and rotate instructions, in a manner similar to that used by Listing 7-14.

Eliminating branches isn't a bad idea in general, since, as we'll see in Chapter 12, branching is very slow. However, as we've already seen in Chapter 4, instruction fetching is also very slow...and the code in Listing 7-14 requires a lot of instruction fetching. 70 instruction bytes must be fetched for each byte that's doubled, meaning that this code can't possibly run in less than about 280 (70 times 4) cycles per byte doubled, even though its official Execution Unit execution time is scarcely 70 cycles.

The Zen timer confirms our calculations, reporting that Listing 7-14 runs in 6.34 ms, or about 300 cycles per byte doubled. (The excess cycles are the result of DRAM refresh.) As a result of this intensive instruction fetching, Mr. Hoyt's optimized shift-and-rotate code actually ran slower than his original test-and-jump code, as discussed in my article "More Optimizing for Speed," Programmer's Journal, July, 1986 (issue 4.4).

So far, all we've done is confirm that the prefetch queue cycle-eater can cause code to run much more slowly than the official execution times would indicate. This is of course not news to us; in fact, I haven't even bothered to show the test- and-jump code and contrast it with the shift-and-rotate code, since that would just restate what we already know. What's interesting is not that Mr. Hoyt's optimization didn't make his code faster, but rather that a look-up table approach can make the code *much* faster. So let's plunge headlong into look-up tables, and see what we can do with this code.

## TABLE LOOK-UPS TO THE RESCUE

Bit-doubling is beautifully suited to an approach based on look-up tables. There are only 256 possible input values, all byte-sized, and only 256 possible output values, all word-sized. Better yet, each input value maps to one and only one output value, and all the input values are consecutive, covering the range 0 to 255, inclusive.

Given those parameters, it should be clear that we can create a table of 256 words, one corresponding to each possible byte to be bit-doubled. We can then use each byte to be doubled as a look-up index into that table, retrieving the appropriate bit-doubled word with just a few instructions. Granted, 512 bytes would be needed to store the table, but the 50 or so instruction bytes we would save would partially compensate for the size of the table. Besides, surely the performance improvement from eliminating all those shifts, rotates, and especially instruction fetches would justify the extra bytes...wouldn't it?

It would indeed. Listing 7-15, which uses the table look-up approach I've just described, runs in just 1.32 ms--more than four times as fast as Listing 7-14! When performance matters, trading less than 500 bytes for a more than four-fold speed increase is quite a deal. Listing 7-15 is so fast that it's faster than Listing 7-14 would be even if there were no prefetch queue cycle-eater; in other words, the official execution time of Listing 7-15 is faster than that of Listing 7-14. Factor in instruction fetch time, though, and you have a fine example of the massive performance improvement that look-up tables can offer.

The key to Listing 7-15, of course, is that I precalculated all the doubled bit masks when I wrote the program. As a result, the code doesn't have to perform any calculation more complex than looking up a precalculated bit mask at run time. In a little while, we'll see how MASM can often perform look-up table calculations at assembly time, relieving us of the drudgery of precalculating results.

## THERE ARE MANY WAYS TO APPROACH ANY TASK

Never assume that there's only one way, or even one "best" way, to approach any programming task. There are always many ways to solve any given programming problem in assembler, and different solutions may well be superior in different situations.

Suppose, for example, that we're writing bit-doubling code in a situation where size is more important than speed, perhaps because we're writing a memory-resident program, or perhaps because the code will be used in a very large program that's squeezed for space. We'd like to improve our speed, if we can-- but not at the expense of a single byte. In this case, Listing 7-14 is preferable to Listing 7-15--but is Listing 7-14 the best we can do?

Not by a long shot.

What we'd like to do is somehow shrink Listing 7-15 a good deal. Well, Listing 7-15 is so large because it has a 512-byte table that's used to look up the bit-doubled words that can be selected by the 256 values that can be stored in a byte. We can shrink the table a great deal simply by converting it to a 16- byte table that's used to look up the bit-doubled _bytes_ that can be selected by the 16 values that can be stored in a _nibble_ (4 bits), and performing two look-ups into that table, one for each half of the byte being doubled.

Listing 7-16 shows this double table look-up solution in action. This listing requires only 23 bytes of code for each byte doubled, and even if you add the 16-byte size of the table in, the total size of 39 bytes is still considerably smaller than the 70 bytes needed to bit-double each byte in Listing 7-14. What's more, the table only needs to appear once in any program, so practically speaking Listing 7-16 is _much_ more compact than Listing 7-14.

Listing 7-16 also is more than twice as fast as Listing 7- 14, clocking in at 2.52 ms. Of course, Listing 7-16 is nearly twice as _slow_ as Listing 7-15--but then, it's much more compact.

There's that choice again: cycles or bytes.

In truth, there are both cycles and bytes yet to be saved in Listing 7-16. If we apply our knowledge of mod-reg-rm addressing to Listing 7-16, we'll realize that it's a waste to use base+displacement addressing with the same displacement twice in a row; we can save a byte and a few cycles by loading SI with the displacement and using base+index addressing instead. Listing 7- 17, which incorporates this optimization, runs in 2.44 ms, a bit faster than Listing 7-16.

There's yet another optimization to be made, and this one brings us full circle, back to the start of our discussion of look-up tables. Think about it: Listing 7-17 basically does nothing more than use two nibble values as look-up indices into a table of byte values. Sound familiar? It should--that's an awful lot like a description of **xlat**. (**xlat** can handle byte look-up values, but this task is just a subset of that.)

Listing 7-18 shows an **xlat**-based version of our bit-doubling code. This code runs in just 1.94 ms, still about 50% slower than the single look-up approach, but a good deal faster than anything else we've seen. Better yet, this approach takes just 16 instruction bytes per bit-doubled byte (32 if you count the table)--which makes this by far the shortest approach we've seen. Comparing Listing 7-18 to Listing 7-14 reveals that we've improved the code to an astonishing degree: Listing 7-18 runs more than three times as fast as Listing 7-14, and yet it requires less than one-fourth as many instruction bytes per bit- doubled byte.

There are many lessons here. First, **xlat** is extremely efficient at performing the limited category of tasks it can manage; when you need to use a byte index into a byte-sized look- up table, **xlat** is often your best bet. Second, the official execution times aren't a particularly good guide to writing high- performance code. (Of course, you already knew _that_!) Third, there is no such thing as the best code, because the fastest code is rarely the smallest code, and vice-versa.

Finally, there are an awful lot of solutions to any given programming problem on the 8088. Don't fall into the trap of thinking that the obvious solution is the best one. In fact, we'll see yet another solution to the bit-doubling problem in Chapter 9; this solution, based on the **sar** instruction, isn't like _any_ of the solutions we've seen so far.

We'll see look-up tables again in Chapter 14, in the form of jump tables.

## INITIALIZING MEMORY

Assembler offers excellent data-definition capabilities, and look-up tables can benefit greatly from those capabilities. No high-level language even comes close to assembler so far as flexible definition of data is concerned, both in terms of arbitrarily mixing different data types and in terms of letting the assembler perform calculations at assembly time; given that, why not let the assembler generate your look-up tables for you?

For example, consider the multiplication of a word-sized value by 80, a task often performed in order to calculate row offsets in display memory. Listing 7-19 does this with the compact but slow **mul** instruction, at a pace of 30.17 us per multiply. Listing 7-20 improves to 15.08 us per multiply by using a faster shift-and-add approach. However, the performance of the shift-and-add approach is limited by the prefetch queue cycle-eater; Listing 7-21, which looks the multiplication results up in a table, is considerably faster yet, at 12.26 us per multiply. Once again, the look-up approach is faster even than tight register-only code, but that's not what's most interesting here. What's really interesting about Listing 7-21 is that it's the assembler, not the programmer, that generates the look-up table of multiples of 80. Back in Listing 7-15, I had to calculate and type each entry in the look-up table myself. In Listing 7-21, however, I've used the **rept** and **=** directives to instruct the assembler to build the table automatically. That's even more convenient than you might think; not only does it save the tedium of a lot of typing, but it avoids the sort of typos that inevitably creep in whenever a lot of typing is involved.

Another area in which assembler's data-definition capabilities lend themselves to good code is in constructing and using mini-interpreters, which are nothing less than task- specific mini-languages that are easily created and used in assembler. We'll discuss mini-interpreters at length in Volume II of The Zen of Assembly Language.

You can also take advantage of assembler's data definition capabilities by assigning initial values to variables when they're defined, rather than initializing them with code. In other words:

```
MemVar          dw      0
```

takes no time at all at run time; **MemVar** simply is 0 when the program starts. By contrast:

```
        MemVar          dw      ?
                         :
                        mov     [MemVar],0
```

takes 20 cycles at run time, and adds 6 bytes to the program as well.

In general, the rule is: calculate results and initialize data at or before assembly time if you can, rather than at run time. What makes look-up tables so powerful is simply that they provide an easy way to shift the overhead of calculations from run time to assembly time.

## A BRIEF NOTE ON I/O ADDRESSING

You may wonder why we've spent so much time on memory addressing but none on input/output (I/O) addressing. The answer is simple: I/O addressing is so limited that there's not much to know about it. There aren't any profound performance implications or optimizations associated with I/O addressing simply because there are only two ways to perform I/O.

**out**, which writes data to a port, always uses the accumulator for the source operand: AL when writing to byte- sized ports, AX when writing to word-sized ports. The destination port address may be specified either by a constant value in the range 0-255 (basically direct port addressing with a byte-sized displacement) or by the value in DX (basically indirect port addressing). Here are the two possible ways to send the value 5Ah to port

99:

```
mov     al,5ah
out     99,al
mov     dx,99
out     dx,al
```

Likewise, **in**, which reads data from a port, always uses AL or AX for the destination operand, and may use either a constant port value between 0 and 255 or the port pointed to by DX as the source operand. Here are the two ways to read a value from port 255 into AL:

```
in      al,0ffh
mov     dx,0ffh
in      al,dx
```

And that just about does it for I/O addressing. As you can see, there's not much flexibility or opportunity for Zen here. All I/O data must pass through the accumulator, and if you want to access a port address greater than 255, you <u>must</u> address the port with DX. What's more, there are no substitutes for the I/O instructions; when you need to perform I/O, what we've just seen is all there is.

While the I/O instructions are a bit awkward, at least they aren't particularly slow, at 8 (DX-indirect) or 10 (direct- addressed) cycles apiece, with no EA calculation time. Neither are the I/O instructions particularly lengthy; in fact, **in** and **out** are considerably more compact than the memory-addressing instructions, which shouldn't be surprising given that the I/O instructions provide such limited functionality. The DX-indirect forms of both **in** and **out** are just 1 byte long, while the direct- addressed forms are 2 bytes long.

Each I/O access takes over the bus and thereby briefly prevents prefetching, much as each memory access does. However, the ratio of total bus accesses (including instruction byte fetches) to execution time for **in** and **out** isn't bad. In fact, byte-sized DX-indirect I/O instructions, which are only 1 byte long and perform only one I/O access, should actually run in close to the advertised 8 cycles per out.

Among our limited repertoire of I/O instructions, which is best? It doesn't make all <u>that</u> much difference, but given the choice between DX-indirect I/O instructions and direct-addressed I/O instructions for heavy I/O, choose DX-indirect, which is slightly faster and more compact. For one-shot I/O to ports in the 0-255 range, use direct-addressed I/O instructions, since it takes three bytes and 4 cycles to set up DX for a DX-indirect I/O instruction.

On balance, though, don't worry about I/O--just do it when you must. Rare indeed is the program that spends an appreciable amount of its time performing I/O--and given the paucity of I/O addressing modes, there's not much to be done about performance in such cases anyway.

**VIDEO PROGRAMMING AND I/O** I'd like to make one final point about I/O addressing. This section won't mean much to you if you haven't worked with video programming, and I'm not going to explain it further now; we'll return to the topic when we discuss video programming in Volume II. For those of you who are involved with video programming, however, here goes.

Word-sized **out** instructions--**out dx,ax**--unquestionably provide the fastest way to set the indexed video registers of the CGA, EGA, and VGA. Just put the index of the video register you're setting in AL and the value you're setting the register to in AH, and **out dx,ax** sets both the index and the register in a single instruction. Using byte-sized **out** instructions, we'd have to do all this to achieve the same results:

```
out     dx,al
inc     dx
xchg    ah,al
out     dx,al
dec     dx
xchg    ah,al
```

(Sometimes you can leave off the final **dec** and **xchg**, but the word-sized approach is still much more efficient.)

However, there's a potential pitfall to the use of word- sized **out** instructions to set indexed video registers. The 8088 can't actually perform word-sized I/O accesses, since the bus is only 8 bits wide. Consequently, the 8088 breaks 16-bit I/O accesses into two 8-bit accesses, one sending AL to the addressed port, and a second one sending AH to the addressed port plus one. (If you think about it, you'll realize that this is exactly how the 8088 handles word-sized memory accesses too.)

All well and good. Unfortunately, on computers built around the 8086, 80286, and the like, the processors do not automatically break up word-sized I/O accesses, since they're fully capable of outputting 16 bits at once. Consequently, when word-sized accesses are made to 8-bit adapters like the EGA by code running on such computers, it's the bus, not the processor, that breaks up those accesses. Generally, that works perfectly well--but on certain PC-compatible computers, the bus outputs the byte in AH to the addressed port plus one first, and then sends the byte in AL to the addressed port. The correct values go to the correct ports, but here sequence is critical; **out dx,ax** to an indexed video register relies on the index in AL being output before the data in AH, and that simply doesn't happen. As a result, the data goes to the wrong video register, and the video programming works incorrectly--sometimes disastrously so.

You may protest that any computer that gets the sequencing of word-sized **out** instructions wrong isn't truly a PC-compatible, and I suppose that's so. Nonetheless, if a computer runs everything except your code that uses word-sized **out** instructions, you're going to have a tough time selling that explanation. Consequently, I recommend using byte-sized **out** instructions to indexed video registers whenever you can't be sure of the particular PC-compatible models on which your code will run.

## AVOID MEMORY!

We've come to the end of our discussion of memory addressing. Memory addressing on the 8088 is no trivial matter, is it? Now that we've familiarized ourselves with the registers and memory addressing capabilities of the 8088, we'll start exploring the instruction set, a journey that will occupy most of the rest of this volume.

Before we leave the realm of memory addressing, let me repeat:

avoid memory. Use the registers to the hilt; register- only instructions are shorter and faster. If you must access memory, try not to use mod-reg-rm addressing; the special memory- accessing instructions, such as the string instructions and **xlat**, are generally shorter and faster. When you do use mod-reg-rm addressing, try not to use displacements, especially 2-byte displacements.

Last but not least, choose your spots. Don't waste time optimizing non-critical code; focus on loops and other chunks of code in which every cycle counts. Assembler programming is not some sort of game where the object is to save cycles and bytes blindly. Rather, the goal is a dual one: to produce whole programs that perform well and to produce those programs as quickly as possible. The key to doing that is knowing how to optimize code, and then doing so in time-critical code--and only in time-critical code.