Chapter 9

Hints My Readers Gave Me

Chapter 9

Optimization Odds and Ends from the Field

Back in high school, I took a pre-calculus class from Mr. Bourgeis, whose most notable characteristics were incessant pacing and truly enormous feet. My friend Barry, who sat in the back row, right behind me, claimed that it was because of his large feet that Mr. Bourgeis was so restless. Those feet were *so* heavy, Barry hypothesized, that if Mr. Bourgeis remained in any one place for too long, the floor would give way under the strain, plunging the unfortunate teacher deep into the mantle of the Earth and possibly all the way through to China. Many amusing cartoons were drawn to this effect.

Unfortunately, Barry was too busy drawing cartoons, or, alternatively, sleeping, to actually learn any math. In the long run, that didn't turn out to be a handicap for Barry, who went on to become vice-president of sales for a ham-packing company, where presumably he was rarely called upon to derive the quadratic equation. Barry's lack of scholarship caused some problems back then, though. On one memorable occasion, Barry was half-asleep, with his eyes open but unfocused and his chin balanced on his hand in the classic "if I fall asleep my head will fall off my hand and I'll wake up" posture, when Mr. Bourgeis popped a killer problem:

"Barry, solve this for X, please." On the blackboard lay the equation:

```
X - 1 = 0
```

"Minus 1," Barry said promptly.

Mr. Bourgeis shook his head mournfully. "Try again." Barry thought hard. He knew the fundamental rule that the answer to most mathematical questions is either 0, 1, infinity, -1, or minus infinity (do not apply this rule to balancing your checkbook, however); unfortunately, that gave him only a 25 percent chance of guessing right.

"One," I whispered surreptitiously.

"Zero," Barry announced. Mr. Bourgeis shook his head even more sadly.

"One," I whispered louder. Barry looked still more thoughtful—a bad sign—so I whispered "one" again, even louder. Barry looked so thoughtful that his eyes nearly rolled up into his head, and I realized that he was just doing his best to convince Mr. Bourgeis that Barry had solved this one by himself.

As Barry neared the climax of his stirring performance and opened his mouth to speak, Mr. Bourgeis looked at him with great concern. "Barry, can you hear me all right?"

"Yes, sir," Barry replied. "Why?"

"Well, I could hear the answer all the way up here. Surely you could hear it just one row away?"

The class went wild. They might as well have sent us home early for all we accomplished the rest of the day.

I like to think I know more about performance programming than Barry knew about math. Nonetheless, I always welcome good ideas and comments, and many readers have sent me a slew of those over the years. So in this chapter, I think I'll return the favor by devoting a chapter to reader feedback.

Another Look at LEA

Several people have pointed out that while **LEA** is great for performing certain additions (see Chapter 6), it isn't a perfect replacement for **ADD**. What's the difference? **LEA**, an addressing instruction by trade, doesn't affect the flags, while the arithmetic **ADD** instruction most certainly does. This is no problem when performing additions that involve only quantities that fit in one machine word (32 bits in 386 protected mode, 16 bits otherwise), but it renders **LEA** useless for multiword operations, which use the Carry flag to tie together partial results. For example, these instructions

```
ADD   EAX,EBX
ADC   EDX,ECX
```

could *not* be replaced

```
LEA   EAX,[EAX+EBX]
ADC   EDX,ECX
```

because **LEA** doesn't affect the Carry flag.

The no-carry characteristic of **LEA** becomes a distinct advantage when performing pointer arithmetic, however. For instance, the following code uses **LEA** to advance the pointers while adding one 128-bit memory variable to another such variable:

```
        MOV   ECX,4   ;# of 32-bit words to add
        CLC
;no carry into the initial ADC
ADDLOOP:

        MOV   EAX,[ESI]       ;get the next element of one array
        ADC   [EDI],EAX       ;add it to the other array, with carry
        LEA   ESI,[ESI+4]     ;advance one array's pointer
        LEA   EDI,[EDI+4]     ;advance the other array's pointer
              LOOP  ADDLOOP
```

(Yes, I could use **LODSD** instead of **MOV/LEA**; I'm just illustrating a point here. Besides, **LODS** is only 1 cycle faster than **MOV/LEA** on the 386, and is actually more than twice as slow on the 486.) If we used **ADD** rather than **LEA** to advance the pointers, the carry from one **ADC** to the next would have to be preserved with either **PUSHF/POPF** or **LAHF/SAHF**. (Alternatively, we could use multiple **INC**s, since **INC** doesn't affect the Carry flag.)

In short, **LEA** is indeed different from **ADD**. Sometimes it's better. Sometimes not; that's the nature of the various instruction substitutions and optimizations that will occur to you over time. There's no such thing as "best" instructions on the x86; it all depends on what you're trying to do.

But there sure are a lot of interesting options, aren't there?

The Kennedy Portfolio

Reader John Kennedy regularly passes along intriguing assembly programming tricks, many of which I've never seen mentioned anywhere else. John likes to optimize for size, whereas I lean more toward speed, but many of his optimizations are good for both purposes. Here are a few of my favorites:

John's code for setting AX to its absolute value is:

```
CWD
XOR   AX,DX
SUB   AX,DX
```

This does nothing when bit 15 of AX is 0 (that is, if AX is positive). When AX is negative, the code "nots" it and adds 1, which is exactly how you perform a two's complement negate. For the case where AX is not negative, this trick usually beats the stuffing out of the standard absolute value code:

```
        AND   AX,AX        ;negative?
        JNS   IsPositive ;no
        NEG   AX           ;yes,negate it
IsPositive:
```

However, John's code is slower on a 486; as you're no doubt coming to realize (and as I'll explain in Chapters 12 and 13), the 486 is an optimization world unto itself.

Here's how John copies a block of bytes from DS:SI to ES:DI, moving as much data as possible a word at a time:

```
SHR   CX,1        ;word count
REP   MOVSW       ;copy as many words as possible
ADC   CX,CX       ;CX=1 if copy length was odd,
                  ;0 else
REP   MOVSB       ;copy any odd byte
```

(**ADC CX,CX** can be replaced with **RCL CX,1**; which is faster depends on the processor type.) It might be hard to believe that the above is faster than this:

```
      SHR   CX,1        ;word count
      REP   MOVSW       ;copy as many words as
                        ;possible
      JNC   CopyDone    ;done if even copy length
      MOVSB             ;copy the odd byte
CopyDone:
```

However, it generally is. Sure, if the length is odd, John's approach incurs a penalty approximately equal to the **REP** startup time for **MOVSB**. However, if the length is even, John's approach doesn't branch, saving cycles and not emptying the prefetch queue. If copy lengths are evenly distributed between even and odd, John's approach is faster in most x86 systems. (Not on the 486, though.)

John also points out that on the 386, multiple **LEA**s can be combined to perform multiplications that can't be handled by a single **LEA**, much as multiple shifts and adds can be used for multiplication, only faster. **LEA** can be used to multiply in a single instruction on the 386, but only by the values 2, 3, 4, 5, 8, and 9; several **LEA**s strung together can handle a much wider range of values. For example, video programmers are undoubtedly familiar with the following code to multiply AX times 80 (the width in bytes of the bitmap in most PC display modes):

```
SHL   AX,1         ;*2
SHL   AX,1         ;*4
SHL   AX,1         ;*8
SHL   AX,1         ;*16
MOV   BX,AX
SHL   AX,1         ;*32
SHL   AX,1         ;*64
ADD   AX,BX        ;*80
```

Using **LEA** on the 386, the above could be reduced to

```
LEA   EAX,[EAX*2]      ;*2
LEA   EAX,[EAX*8]      ;*16
LEA   EAX,[EAX+EAX*4]  ;*80
```

which still isn't as fast as using a lookup table like

```
MOV   EAX,MultiplesOf80Table[EAX*4]
```

but is close and takes a great deal less space.

Of course, on the 386, the shift and add version could also be reduced to this considerably more efficient code:

```
SHL   AX,4        ;*16
MOV   BX,AX
SHL   AX,2        ;*64
ADD   AX,BX       ;*80
```

Speeding Up Multiplication

That brings us to multiplication, one of the slowest of x86 operations and one that allows for considerable optimization. One way to speed up multiplication is to use shift and add, **LEA**, or a lookup table to hard-code a multiplication operation for a fixed multiplier, as shown above. Another is to take advantage of the early-out feature of the 386 (and the 486, but in the interests of brevity I'll just say "386" from now on) by arranging your operands so that the multiplier (always the rightmost operand following **MUL** or **IMUL**) is no larger than the other operand.

Why? Because the 386 processes one multiplier bit per cycle and immediately ends a multiplication when all significant bits of the multiplier have been processed, so fewer cycles are required to multiply a large multiplicand times a small multiplier than a small multiplicand times a large multiplier, by a factor of about 1 cycle for each significant multiplier bit eliminated.

(There's a minimum execution time on this trick; below 3 significant multiplier bits, no additional cycles are saved.) For example, multiplication of 32,767 times 1 is 12 cycles faster than multiplication of 1 times 32,727.

Choosing the right operand as the multiplier can work wonders. According to published specs, the 386 takes 38 cycles to multiply by a multiplier with 32 significant bits but only 9 cycles to multiply by a multiplier of 2, a performance improvement of more than four times! (My tests regularly indicate that multiplication takes 3 to 4 cycles longer than the specs indicate, but the cycle-per-bit advantage of smaller multipliers holds true nonetheless.)

This highlights another interesting point: **MUL** and **IMUL** on the 386 are so fast that alternative multiplication approaches, while generally still faster, are worthwhile only in truly time-critical code.

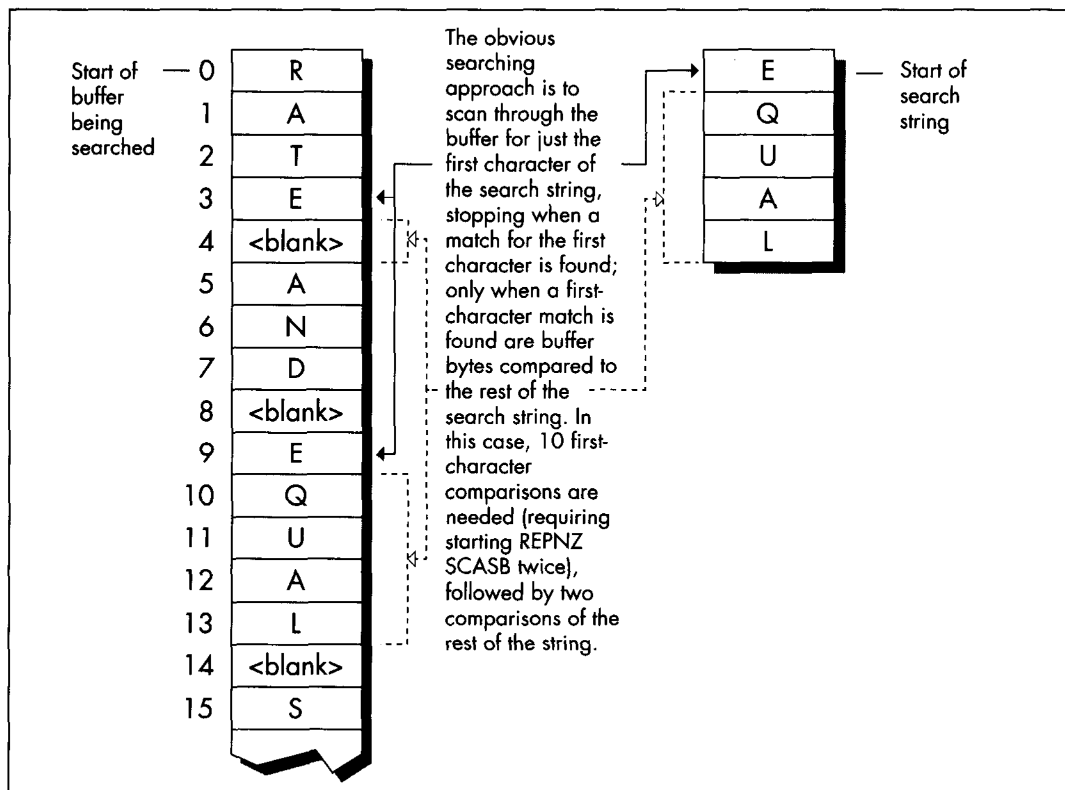
*On 386SXs and uncached 386s, where code size can significantly affect performance due to instruction prefetching, the compact **MUL** and **IMUL** instructions can approach and in some cases even outperform the "optimized" alternatives.*

All in all, **MUL** and **IMUL** are reasonable performers on the 386, no longer to be avoided in most cases—and you can help that along by arranging your code to make the smaller operand the multiplier whenever you know which operand is smaller.

That doesn't mean that your code should test and swap operands to make sure the smaller one is the multiplier; that rarely pays off. I'm speaking more of the case where you're scaling an array up by a value that's always in the range of, say, 2 to 10; because the scale value will always be small and the array elements may have any value, the scale value is the logical choice for the multiplier.

Optimizing Optimized Searching

Rob Williams writes with a wonderful optimization to the **REPNZ SCASB**-based optimized searching routine I discussed in Chapter 5. As a quick refresher, I described searching a buffer for a text string as follows: Scan for the first byte of the text string with **REPNZ SCASB**, then use **REPZ CMPS** to check for a full match whenever **REPNZ**



Simple searching method for locating a text string.
Figure 9.1

SCASB finds a match for the first character, as shown in Figure 9.1. The principle is that most buffer characters won't match the first character of any given string, so **REPNZ SCASB**, by far the fastest way to search on the PC, can be used to eliminate most potential matches; each remaining potential match can then be checked in its entirety with **REPZ CMPS**.

Rob's revelation, which he credits without explanation to Edgar Allen Poe (search nevermore?), was that by far the slowest part of the whole deal is handling **REPNZ SCASB** matches, which require checking the remainder of the string with **REPZ CMPS** and restarting **REPNZ SCASB** if no match is found.

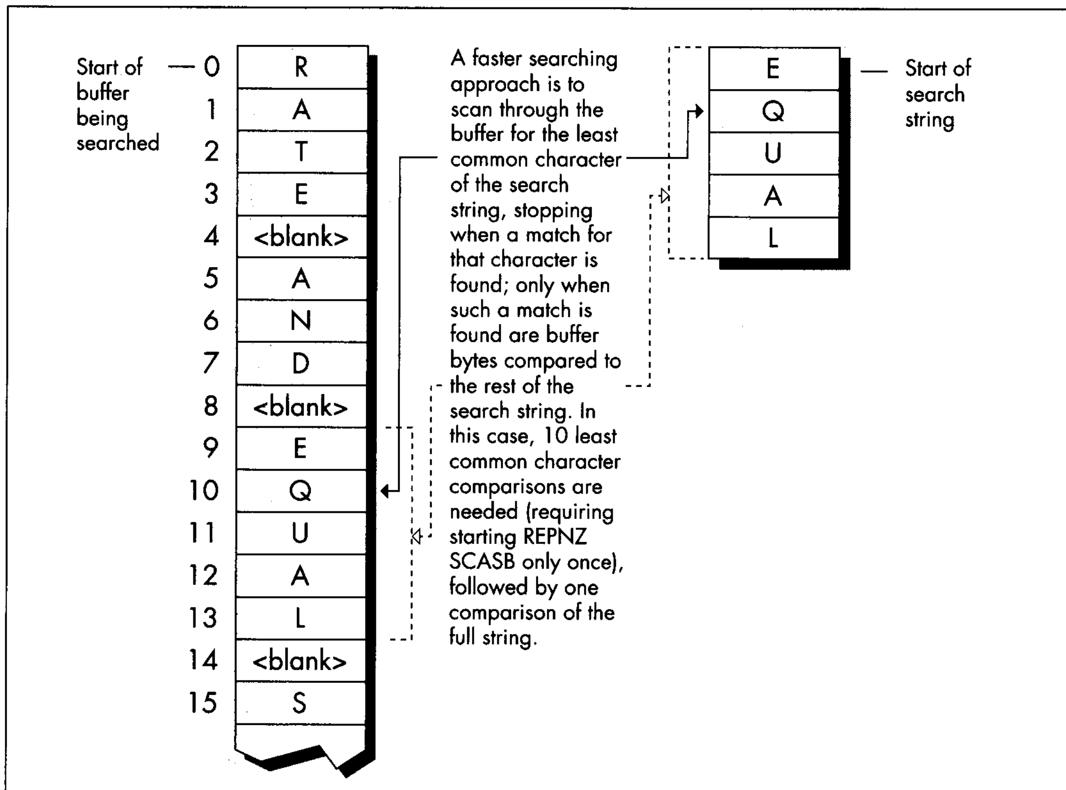
*Rob points out that the number of **REPNZ SCASB** matches can easily be reduced simply by scanning for the character in the searched-for string that appears least often in the buffer being searched.*

Imagine, if you will, that you're searching for the string "EQUAL." By my approach, you'd use **REPNZ SCASB** to scan for each occurrence of "E," which crops up quite often in normal text. Rob points out that it would make more sense to scan for "Q," then back up one character and check the whole string when a "Q" is found, as shown in Figure 9.2. "Q" is likely to occur much less often, resulting in many fewer whole-string checks and much faster processing.

Listing 9.1 implements the scan-on-first-character approach. Listing 9.2 scans for whatever character the caller specifies. Listing 9.3 is a test program used to compare the two approaches. How much difference does Rob's revelation make? Plenty. Even when the entire C function call to **FindString** is timed—**strlen** calls, parameter pushing, calling, setup, and all—the version of **FindString** in Listing 9.2, which is directed by Listing 9.3 to scan for the infrequently-occurring "Q," is about 40 percent faster on a 20 MHz cached 386 for the test search of Listing 9.3 than is the version of **FindString** in Listing 9.1, which always scans for the first character, in this case "E." However, when only the search loops (the code that actually does the searching) in the two versions of **FindString** are compared, Listing 9.2 is more than *twice* as fast as Listing 9.1—a remarkable improvement over code that already uses **REPNZ SCASB** and **REPZ CMPS**.

What I like so much about Rob's approach is that it demonstrates that optimization involves much more than instruction selection and cycle counting. Listings 9.1 and 9.2 use pretty much the same instructions, and even use the same approach of scanning with **REPNZ SCASB** and using **REPZ CMPS** to check scanning matches.

The difference between Listings 9.1 and 9.2 (which gives you more than a doubling of performance) is due entirely to understanding the nature of the data being handled, and biasing the code to reflect that knowledge.



Faster searching method for locating a text string.
Figure 9.2

LISTING 9.1 L9-1.ASM

```
; Searches a text buffer for a text string. Uses REPNZ SCASB to scan
; the buffer for locations that match the first character of the
; searched-for string, then uses REPZ CMPS to check fully only those
; locations that REPNZ SCASB has identified as potential matches.
;
; Adapted from Zen of Assembly Language, by Michael Abrash
;
; C small model-callable as:
;    unsigned char * FindString(unsigned char * Buffer,
;     unsigned int BufferLength, unsigned char * SearchString,
;     unsigned int SearchStringLength);
;
; Returns a pointer to the first match for SearchString in Buffer,or
; a NULL pointer if no match is found. Buffer should not start at
; offset 0 in the data segment to avoid confusing a match at 0 with
; no match found.
Parms struc
                        dw    2 dup(?)    ;pushed BP/return address
Buffer                  dw      ?         ;pointer to buffer to search
BufferLength            dw      ?         ;length of buffer to search
```

```
SearchString          dw      ?       ;pointer to string for which to search
SearchStringLength    dw      ?       ;length of string for which to search
Parms ends
      .model     small
      .code
      public _FindString
_FindString proc  near
      push  bp    ;preserve caller's stack frame
      mov   bp,sp ;point to our stack frame
      push  si    ;preserve caller's register variables
      push  di
      cld         ;make string instructions increment pointers
      mov   si,[bp+SearchString]        ;pointer to string to search for
      mov   bx,[bp+SearchStringLength]  ;length of string
      and   bx,bx
      jz    FindStringNotFound          ;no match if string is 0 length
      mov   dx,[bp+BufferLength]        ;length of buffer
      sub   dx,bx                       ;difference between buffer and string lengths
      jc    FindStringNotFound          ;no match if search string is
                                        ; longer than buffer
      inc   dx    ;difference between buffer and search string
                  ; lengths, plus 1 (# of possible string start
                  ; locations to check in the buffer)
      mov   di,ds
      mov   es,di
      mov   di,[bp+Buffer]    ;point ES:DI to buffer to search thru
      lodsb                   ;put the first byte of the search string in AL
      mov   bp,si             ;set aside pointer to the second search byte
      dec   bx                ;don't need to compare the first byte of the
                              ; string with CMPS; we'll do it with SCAS
FindStringLoop:
      mov   cx,dx ;put remaining buffer search length in CX
      repnz scasb ;scan for the first byte of the string
      jnz   FindStringNotFound ;not found, so there's no match
                               ;found, so we have a potential match-check the
                               ; rest of this candidate location
      push  di                 ;remember the address of the next byte to scan
      mov   dx,cx              ;set aside the remaining length to search in
                               ; the buffer
      mov   si,bp              ;point to the rest of the search string
      mov   cx,bx              ;string length (minus first byte)
      shr   cx,1               ;convert to word for faster search
      jnc   FindStringWord     ;do word search if no odd byte
      cmpsb                    ;compare the odd byte
      jnz   FindStringNoMatch  ;odd byte doesn't match, so we
                               ; haven't found the search string here
FindStringWord:
      jcxz  FindStringFound    ;test whether we've already checked
                               ; the whole string; if so, this is a match
                               ; bytes long; if so, we've found a match
      repz  cmpsw              ;check the rest of the string a word at a time
      jz    FindStringFound    ;it's a match
FindStringNoMatch:
      pop   di               ;get back pointer to the next byte to scan
      and   dx,dx            ;is there anything left to check?
      jnz   FindStringLoop   ;yes-check next byte
FindStringNotFound:
      sub   ax,ax            ;return a NULL pointer indicating that the
      jmp   FindStringDone   ; string was not found
```

```
FindStringFound:
      pop   ax     ;point to the buffer location at which the
      dec   ax     ; string was found (earlier we pushed the
                   ; address of the byte after the start of the
                   ; potential match)
FindStringDone:
      pop   di     ;restore caller's register variables
      pop   si
      pop   bp     ;restore caller's stack frame
      ret
_FindString endp
      end
```

LISTING 9.2 L9-2.ASM

```
; Searches a text buffer for a text string. Uses REPNZ SCASB to scan
; the buffer for locations that match a specified character of the
; searched-for string, then uses REPZ CMPS to check fully only those
; locations that REPNZ SCASB has identified as potential matches.
;
; C small model-callable as:
;    unsigned char * FindString(unsigned char * Buffer,
;     unsigned int BufferLength, unsigned char * SearchString,
;     unsigned int SearchStringLength,
;     unsigned int ScanCharOffset);
;
; Returns a pointer to the first match for SearchString in Buffer,or
; a NULL pointer if no match is found. Buffer should not start at
; offset 0 in the data segment to avoid confusing a match at 0 with
; no match found.
Parms struc
                       dw    2 dup(?)    ;pushed BP/return address
Buffer                 dw    ?           ;pointer to buffer to search
BufferLength           dw    ?           ;length of buffer to search
SearchString           dw    ?           ;pointer to string for which to search
SearchStringLength     dw    ?           ;length of string for which to search
ScanCharOffset         dw    ?           ;offset in string of character for
                                         ; which to scan
Parms ends
      .model     small
      .code
      public _FindString
_FindString proc  near
      push  bp          ;preserve caller's stack frame
      mov   bp,sp       ;point to our stack frame
      push  si          ;preserve caller's register variables
      push  di
      cld               ;make string instructions increment pointers
      mov   si,[bp+SearchString]          ;pointer to string to search for
      mov   cx,[bp+SearchStringLength]    ;length of string
      jcxz  FindStringNotFound            ;no match if string is 0 length
      mov   dx,[bp+BufferLength]          ;length of buffer
      sub   dx,cx                         ;difference between buffer and search
                                          ; lengths
      jc    FindStringNotFound ;no match if search string is
                  ; longer than buffer
      inc   dx    ; difference between buffer and search string
                  ; lengths, plus 1 (# of possible string start
                  ; locations to check in the buffer)
      mov   di,ds
      mov   es,di
```

```
        mov   di,[bp+Buffer]          ;point ES:DI to buffer to search thru
        mov   bx,[bp+ScanCharOffset]  ;offset in string of character
                                      ; on which to scan
        add   di,bx       ;point ES:DI to first buffer byte to scan
        mov   al,[si+bx]  ;put the scan character in AL
        inc   bx          ;set BX to the offset back to the start of the
                          ; potential full match after a scan match,
                          ; accounting for the 1-byte overrun of
                          ; REPNZ SCASB
FindStringLoop:
        mov   cx,dx               ;put remaining buffer search length in CX
        repnz scasb               ;scan for the scan byte
        jnz   FindStringNotFound  ;not found, so there's no match
                                  ;found, so we have a potential match-check the
                                  ; rest of this candidate location
        push  di                  ;remember the address of the next byte to scan
        mov   dx,cx               ;set aside the remaining length to search in
                                  ; the buffer
        sub   di,bx               ;point back to the potential start of the
                                  ; match in the buffer
        mov   si,[bp+SearchString]          ;point to the start of the string
        mov   cx,[bp+SearchStringLength]    ;string length
        shr   cx,1                          ;convert to word for faster search
        jnc   FindStringWord                ;do word search if no odd byte
        cmpsb                               ;compare the odd byte
        jnz   FindStringNoMatch             ;odd byte doesn't match, so we
                                            ; haven't found the search string here
FindStringWord:
        jcxz  FindStringFound   ;if the string is only 1 byte long,
                                ; we've found a match
        repz  cmpsw             ;check the rest of the string a word at a time
        jz    FindStringFound   ;it's a match
FindStringNoMatch:
        pop   di                ;get back pointer to the next byte to scan
        and   dx,dx             ;is there anything left to check?
        jnz   FindStringLoop    ;yes-check next byte
FindStringNotFound:
        sub   ax,ax             ;return a NULL pointer indicating that the
        jmp   FindStringDone    ; string was not found
FindStringFound:
        pop   ax    ;point to the buffer location at which the
        sub   ax,bx ; string was found (earlier we pushed the
                    ; address of the byte after the scan match)
FindStringDone:
        pop   di    ;restore caller's register variables
        pop   si
        pop   bp    ;restore caller's stack frame
        ret
_FindString endp
        end
```

LISTING 9.3 L9-3.C

```
/* Program to exercise buffer-search routines in Listings 9.1 & 9.2 */
#include <stdio.h>
#include <string.h>

#define DISPLAY_LENGTH  40
extern unsigned char * FindString(unsigned char *, unsigned int,
      unsigned char *, unsigned int, unsigned int);
void main(void);
```

```
static unsigned char TestBuffer[] = "When, in the course of human \
events, it becomes necessary for one people to dissolve the \
political bands which have connected them with another, and to \
assume among the powers of the earth the separate and equal station \
to which the laws of nature and of nature's God entitle them...";

void main() {
   static unsigned char TestString[] = "equal";
   unsigned char TempBuffer[DISPLAY_LENGTH+1];
   unsigned char *MatchPtr;

   /* Search for TestString and report the results */
   if ((MatchPtr = FindString(TestBuffer,
         (unsigned int) strlen(TestBuffer), TestString,
         (unsigned int) strlen(TestString), 1)) == NULL) {
      /* TestString wasn't found */
      printf("\"%s\" not found\n", TestString);
   } else {
      /* TestString was found. Zero-terminate TempBuffer; strncpy
         won't do it if DISPLAY_LENGTH characters are copied */
      TempBuffer[DISPLAY_LENGTH] = 0;
      printf("\"%s\" found. Next %d characters at match:\n\"%s\"\n",
            TestString, DISPLAY_LENGTH,
            strncpy(TempBuffer, MatchPtr, DISPLAY_LENGTH));
   }
}
```

You'll notice that in Listing 9.2 I didn't use a table of character frequencies in English text to determine the character for which to scan, but rather let the caller make that choice. Each buffer of bytes has unique characteristics, and English-letter frequency could well be inappropriate. What if the buffer is filled with French text? Cyrillic? What if it isn't text that's being searched? It might be worthwhile for an application to build a dynamic frequency table for each buffer so that the best scan character could be chosen for each search. Or perhaps not, if the search isn't time-critical or the buffer is small.

The point is that you can improve performance dramatically by understanding the nature of the data with which you work. (This is equally true for high-level language programming, by the way.) Listing 9.2 is very similar to and only slightly more complex than Listing 9.1; the difference lies not in elbow grease or cycle counting but in the organic integrating optimizer technology we all carry around in our heads.

Short Sorts

David Stafford (recently of Borland and Borland Japan) who happens to be one of the best assembly language programmers I've ever met, has written a C-callable routine that sorts an array of integers in ascending order. That wouldn't be particularly noteworthy, except that David's routine, shown in Listing 9.4, is exactly *25 bytes* long. Look at the code; you'll keep saying to yourself, "But this doesn't work...oh, yes, I guess it does." As they say in the Prego spaghetti sauce ads, *it's in there*—and what a job of packing. Anyway, David says that a 24-byte sort routine eludes him, and he'd like to know if anyone can come up with one.

LISTING 9.4 L9-4.ASM

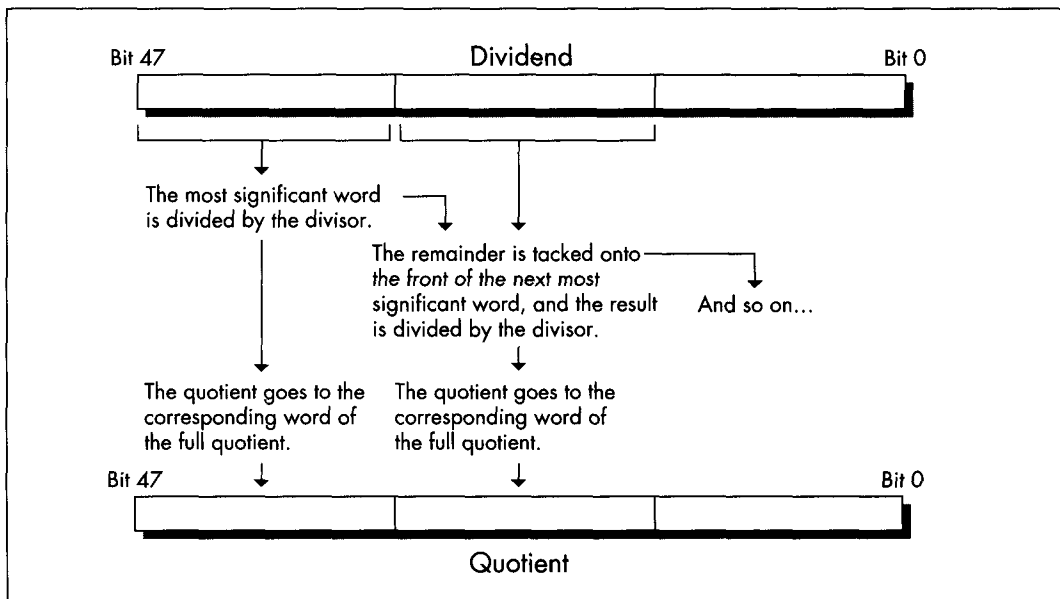
```
.
;------------------------------------------------------------------------
; Sorts an array of ints.  C callable (small model).  25 bytes.
; void sort( int num, int a[] );
;
; Courtesy of David Stafford.
;------------------------------------------------------------------------

      .model small
      .code
        public _sort

top:    mov     dx,[bx]         ;swap two adjacent integers
        xchg    dx,[bx+2]
        xchg    dx,[bx]

        cmp     dx,[bx]         ;did we put them in the right order?
        jl      top             ;no, swap them back

        inc     bx              ;go to next integer
        inc     bx
        loop    top

_sort:  pop     dx              ;get return address (entry point)
        pop     cx              ;get count
        pop     bx              ;get pointer
        push    bx              ;restore pointer
        dec     cx              ;decrement count
        push    cx              ;save count
        push    dx              ;restore return address
        jg      top             ;if cx > 0

        ret

      end
```

Full 32-Bit Division

One of the most annoying limitations of the x86 is that while the dividend operand to the **DIV** instruction can be 32 bits in size, both the divisor and the result must be 16 bits. That's particularly annoying in regards to the result because sometimes you just don't know whether the ratio of the dividend to the divisor is greater than 64K-1 or not—and if you guess wrong, you get that godawful Divide By Zero interrupt. So, what is one to do when the result might not fit in 16 bits, or when the dividend is larger than 32 bits? Fall back to a software division approach? That will work—but oh so slowly.

There's another technique that's much faster than a pure software approach, albeit not so flexible. This technique allows arbitrarily large dividends and results, but the divisor is still limited to16 bits. That's not perfect, but it does solve a number of problems, in particular eliminating the possibility of a Divide By Zero interrupt from a too-large result.

This technique involves nothing more complicated than breaking up the division into word-sized chunks, starting with the most significant word of the dividend. The



Fast multiword division on the 386.
Figure 9.3

most significant word is divided by the divisor (with no chance of overflow because there are only 16 bits in each); then the remainder is prepended to the next 16 bits of dividend, and the process is repeated, as shown in Figure 9.3. This process is equivalent to dividing by hand, except that here we stop to carry the remainder manually only after each word of the dividend; the hardware divide takes care of the rest. Listing 9.5 shows a function to divide an arbitrarily large dividend by a 16-bit divisor, and Listing 9.6 shows a sample division of a large dividend. Note that the same principle can be applied to handling arbitrarily large dividends in 386 native mode code, but in that case the operation can proceed a dword, rather than a word, at a time.

As for handling signed division with arbitrarily large dividends, that can be done easily enough by remembering the signs of the dividend and divisor, dividing the absolute value of the dividend by the absolute value of the divisor, and applying the stored signs to set the proper signs for the quotient and remainder. There may be more clever ways to produce the same result, by using **IDIV**, for example; if you know of one, drop me a line c/o Coriolis Group Books.

LISTING 9.5 L9-5.ASM

```
; Divides an arbitrarily long unsigned dividend by a 16-bit unsigned
; divisor. C near-callable as:
;    unsigned int Div(unsigned int * Dividend,
```

```
;          int DividendLength, unsigned int Divisor,
;          unsigned int * Quotient);
;
; Returns the remainder of the division.
;
; Tested with TASM 2.

parms struc
           dw    2 dup (?)   ;pushed BP & return address
Dividend   dw    ?           ;pointer to value to divide, stored in Intel
                             ; order, with lsb at lowest address, msb at
                             ; highest. Must be composed of an integral
                             ; number of words
DividendLength dw ?          ;# of bytes in Dividend. Must be a multiple
                             ; of 2
Divisor    dw    ?           ;value by which to divide. Must not be zero,
                             ; or a Divide By Zero interrupt will occur
Quotient   dw    ?           ;pointer to buffer in which to store the
                             ; result of the division, in Intel order.
                             ; The quotient returned is of the same
                             ; length as the dividend
parms ends

           .model      small
           .code
           public      _Div
_Div  proc near
           push    bp        ;preserve caller's stack frame
           mov     bp,sp     ;point to our stack frame
           push    si        ;preserve caller's register variables
           push    di

           std               ;we're working from msb to lsb
           mov   ax,ds
           mov   es,ax       ;for STOS
           mov   cx,[bp+DividendLength]
           sub   cx,2
           mov   si,[bp+Dividend]
           add   si,cx       ;point to the last word of the dividend
                             ; (the most significant word)
           mov   di,[bp+Quotient]
           add   di,cx       ;point to the last word of the quotient
                             ; buffer (the most significant word)
           mov   bx,[bp+Divisor]
           shr   cx,1
           inc   cx          ;# of words to process
           sub   dx,dx       ;convert initial divisor word to a 32-bit
                             ;value for DIV
DivLoop:
           lodsw             ;get next most significant word of divisor
           div   bx
           stosw             ;save this word of the quotient
                             ;DX contains the remainder at this point,
                             ; ready to prepend to the next divisor word
           loop  DivLoop
           mov   ax,dx       ;return the remainder

           cld               ;restore default Direction flag setting
           pop     di        ;restore caller's register variables
           pop     si
           pop     bp        ;restore caller's stack frame
```

```
        ret
_Div  endp
        end
```

LISTING 9.6 L9-6.C

```
/* Sample use of Div function to perform division when the result
   doesn't fit in 16 bits */

#include <stdio.h>

extern unsigned int Div(unsigned int * Dividend,
          int DividendLength, unsigned int Divisor,
          unsigned int * Quotient);

main() {
   unsigned long m, i = 0x20000001;
   unsigned int k, j = 0x10;

   k = Div((unsigned int *)&i, sizeof(i), j, (unsigned int *)&m);
   printf("%lu / %u = %lu r %u\n", i, j, m, k);
}
```

Sweet Spot Revisited

Way back in Volume 1, Number 1 of *PC TECHNIQUES,* (April/May 1990) I wrote the very first of that magazine's HAX (#1), which extolled the virtues of placing your most commonly-used automatic (stack-based) variables within the stack's "sweet spot," the area between +127 to -128 bytes away from BP, the stack frame pointer. The reason was that the 8088 can store addressing displacements that fall within that range in a single byte; larger displacements require a full word of storage, increasing code size by a byte per instruction, and thereby slowing down performance due to increased instruction fetching time.

This takes on new prominence in 386 native mode, where straying from the sweet spot costs not one, but two or three bytes. Where the 8088 had two possible displacement sizes, either byte or word, on the 386 there are three possible sizes: byte, word, or dword. In native mode (32-bit protected mode), however, a prefix byte is needed in order to use a word-sized displacement, so a variable located outside the sweet spot requires either two extra bytes (an extra displacement byte plus a prefix byte) or three extra bytes (a dword displacement rather than a byte displacement). Either way, instructions grow alarmingly.

Performance may or may not suffer from missing the sweet spot, depending on the processor, the memory architecture, and the code mix. On a 486, prefix bytes often cost a cycle; on a 386SX, increased code size often slows performance because instructions must be fetched through the half-pint 16-bit bus; on a 386, the effect depends on the instruction mix and whether there's a cache.

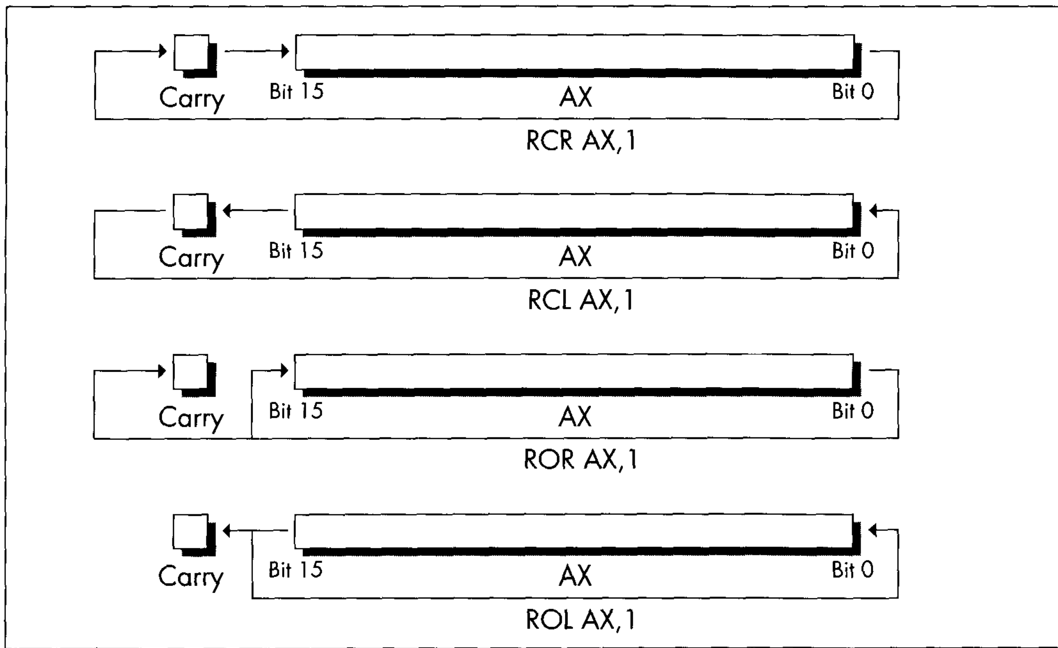
On balance, though, it's as important to keep your most-used variables in the stack's sweet spot in 386 native mode as it was on the 8088.

In assembly, it's easy to control the organization of your stack frame. In C, however, you'll have to figure out the allocation scheme your compiler uses to allocate automatic variables, and declare automatics appropriately to produce the desired effect. It can be done: I did it in Turbo C some years back, and trimmed the size of a program (admittedly, a large one) by several K—not bad, when you consider that the "sweet spot" optimization is essentially free, with no code reorganization, change in logic, or heavy thinking involved.

Hard-Core Cycle Counting

Next, we come to an item that cycle counters will love, especially since it involves apparently incorrect documentation on Intel's part. According to Intel's documents, all **RCR** and **RCL** instructions, which perform rotations through the Carry flag, as shown in Figure 9.4, take 9 cycles on the 386 when working with a register operand. My measurements indicate that the 9-cycle execution time almost holds true for *multibit* rotate-through-carries, which I've timed at 8 cycles apiece; for example, **RCR AX,CL** takes 8 cycles on *my* 386, as does **RCL DX,2**. Contrast that with **ROR** and **ROL**, which can rotate the contents of a register any number of bits in just 3 cycles.

However, rotating by *one* bit through the Carry flag does *not* take 9 cycles, contrary to Intel's *80386 Programmer's Reference Manual,* or even 8 cycles. In fact, **RCR** *reg*,1 and



Performing rotate instructions using the Carry flag.
Figure 9.4

RCL *reg*,1 take 3 cycles, just like **ROR**, **ROL**, **SHR**, and **SHL**. At least, that's how fast they run on my 386, and I very much doubt that you'll find different execution times on other 386s. (Please let me know if you do, though!)

Interestingly, according to Intel's *i486 Microprocessor Programmer's Reference Manual,* the 486 can **RCR** or **RCL** a register by one bit in 3 cycles, but takes between 8 and 30 cycles to perform a multibit register **RCR** or **RCL**!

No great lesson here, just a caution to be leery of multibit **RCR** and **RCL** when performance matters—and to take cycle-time documentation with a grain of salt.

Hardwired Far Jumps

Did you ever wonder how to code a far jump to an absolute address in assembly language? Probably not, but if you ever do, you're going to be glad for this next item, because the obvious solution doesn't work. You might think all it would take to jump to, say, 1000:5 would be **JMP FAR PTR 1000:5**, but you'd be wrong. That won't even assemble. You might then think to construct in memory a far pointer containing 1000:5, as in the following:

```
Ptr   dd    ?
      :
      mov   word ptr [Ptr],5
      mov   word ptr [Ptr+2],1000h
      jmp   [Ptr]
```

That will work, but at a price in performance. On an 8088, **JMP DWORD PTR [*mem*]** (an indirect far jump) takes at least 37 cycles; **JMP DWORD PTR *label*** (a direct far jump) takes only 15 cycles (plus, almost certainly, some cycles for instruction fetching). On a 386, an indirect far jump is documented to take at least 43 cycles in real mode (31 in protected mode); a direct far jump is documented to take at least 12 cycles, about three times faster. In truth, the difference between those two is nowhere near that big; the fastest I've measured for a direct far jump is 21 cycles, and I've measured indirect far jumps as fast as 30 cycles, so direct is still faster, but not by so much. (Oh, those cycle-time documentation blues!) Also, a direct far jump is documented to take at least 27 cycles in protected mode; why the big difference in protected mode, I have no idea.

At any rate, to return to our original problem of jumping to 1000:5: Although an indirect far jump will work, a direct far jump is still preferable.

Listing 9.7 shows a short program that performs a direct far call to 1000:5. (Don't run it, unless you want to crash your system!) It does this by creating a dummy segment at 1000H, so that the label **FarLabel** can be created with the desired far attribute at the proper location. (Segments created with "AT" don't cause the generation of any actual bytes or the allocation of any memory; they're just templates.) It's a little kludgey, but at least it does work. There may be a better solution; if you have one, pass it along.

LISTING 9.7 L9-7.ASM

```
; Program to perform a direct far jump to address 1000:5.
; *** Do not run this program! It's just an example of how ***
; *** to build a direct far jump to an absolute address    ***
;
; Tested with TASM 2 and MASM 5.

FarSeg     segment    at 01000h
     org   5
FarLabel label   far
FarSeg     ends

     .model     small
     .code
start:
     jmp   FarLabel
     end   start
```

By the way, if you're wondering how I figured this out, I merely applied my good friend Dan Illowsky's long-standing rule for dealing with MASM:

If the obvious doesn't work (and it usually doesn't), just try everything you can think of, no matter how ridiculous, until you find something that does—a rule with plenty of history on its side.

Setting 32-Bit Registers: Time versus Space

To finish up this chapter, consider these two items. First, in 32-bit protected mode,

```
sub   eax,eax
inc   eax
```

takes 4 cycles to execute, but is only 3 bytes long, while

```
mov   eax,1
```

takes only 2 cycles to execute, but is 5 bytes long (because native mode constants are dwords and the **MOV** instruction doesn't sign-extend). Both code fragments are ways to set **EAX** to 1 (although the first affects the flags and the second doesn't); this is a classic trade-off of speed for space. Second,

```
or    ebx,-1
```

takes 2 cycles to execute and is 3 bytes long, while

```
mov   ebx,-1
```

takes 2 cycles to execute and is 5 bytes long. Both instructions set **EBX** to -1; this is a classic trade-off of—gee, it's not a trade-off at all, is it? **OR** is a better way to set a 32-bit register to all 1-bits, just as **SUB** or **XOR** is a better way to set a register to all 0-bits. Who woulda thunk it? Just goes to show how the 32-bit displacements and constants of 386 native mode change the familiar landscape of 80×86 optimization.

Be warned, though, that I've found **OR**, **AND**, **ADD**, and the like to be a cycle slower than **MOV** when working with immediate operands on the 386 under some circumstances, for reasons that thus far escape me. This just reinforces the first rule of optimization: Measure your code in action, and place not your trust in documented cycle times.