Local Optimization
You might not think it, but there’s much to learn about performance programming from the Great Buffalo Sauna Fiasco. To wit:

The scene is Buffalo, New York, in the dead of winter, with the snow piled several feet deep. Four college students, living in typical student housing, are frozen to the bone. The third floor of their house, uninsulated and so cold that it’s uninhabitable, has an ancient bathroom. One fabulously cold day, inspiration strikes:

“Hey—we could make that bathroom into a sauna!”

Pandemonium ensues. Someone rushes out and buys a gas heater, and at considerable risk to life and limb hooks it up to an abandoned but still live gas pipe that once fed a stove on the third floor. Someone else gets sheets of plastic and lines the walls of the bathroom to keep the moisture in, and yet another student gets a bucket full of rocks. The remaining chap brings up some old wooden chairs and sets them up to make benches along the sides of the bathroom. Voila—instant sauna!

They crank up the gas heater, put the bucket of rocks in front of it, close the door, take off their clothes, and sit down to steam themselves. Mind you, it’s not yet 50 degrees Fahrenheit in this room, but the gas heater is roaring. Surely warmer times await.

Indeed they do. The temperature climbs to 55 degrees, then 60, then 63, then 65, and finally creeps up to 68 degrees.
And there it stops.

68 degrees is warm for an uninsulated third floor in Buffalo in the dead of winter. Damn warm. It is not, however, particularly warm for a sauna. Eventually someone acknowledges the obvious and allows that it might have been a stupid idea after all, and everyone agrees, and they shut off the heater and leave, each no doubt offering silent thanks that they had gotten out of this without any incidents requiring major surgery.

And so we see that the best idea in the world can fail for lack of either proper design or adequate horsepower. The primary cause of the Great Buffalo Sauna Fiasco was a lack of horsepower; the gas heater was flat-out undersized. This is analogous to trying to write programs that incorporate features like bitmapped text and searching of multisegment buffers without using high-performance assembly language. Any PC language can perform just about any function you can think of—eventually. That heater would eventually have heated the room to 110 degrees, too—along about the first of June or so.

The Great Buffalo Sauna Fiasco also suffered from fundamental design flaws. A more powerful heater would indeed have made the room hotter—and might well have burned the house down in the process. Likewise, proper algorithm selection and good design are fundamental to performance. The extra horsepower a superb assembly language implementation gives a program is worth bothering with only in the context of a good design.

---

Assembly language optimization is a small but crucial corner of the PC programming world. Use it sparingly and only within the framework of a good design—but ignore it and you may find various portions of your anatomy out in the cold.

---

So, drawing fortitude from the knowledge that our quest is a pure and worthy one, let’s resume our exploration of assembly language instructions with hidden talents and instructions with well-known talents that are less than they appear to be. In the process, we’ll come to see that there is another, very important optimization level between the algorithm/design level and the cycle-counting/individual instruction level. I’ll call this middle level local optimization; it involves focusing on optimizing sequences of instructions rather than individual instructions, all with an eye to implementing designs as efficiently as possible given the capabilities of the x86 family instruction set.

And yes, in case you’re wondering, the above story is indeed true. Was I there? Let me put it this way: If I were, I’d never admit it!

---

When LOOP Is a Bad Idea

Let’s examine first an instruction that is less than it appears to be: LOOP. There’s no mystery about what LOOP does; it decrements CX and branches if CX doesn’t decrement to zero. It’s so beautifully suited to the task of counting down loops that any
experienced x86 programmer instinctively stuffs the loop count in CX and reaches for LOOP when setting up a loop. That’s fine—LOOP does, of course, work as advertised—but there is one problem:

On half of the processors in the x86 family, LOOP is slower than DEC CX followed by JNZ. (Granted, DEC CX/JNZ isn’t precisely equivalent to LOOP, because DEC alters the flags and LOOP doesn’t, but in most situations they’re comparable.)

How can this be? Don’t ask me, ask Intel. On the 8088 and 80286, LOOP is indeed faster than DEC CX/JNZ by a cycle, and LOOP is generally a little faster still because it’s a byte shorter and so can be fetched faster. On the 386, however, things change; LOOP is two cycles slower than DEC/JNZ, and the fetch time for one extra byte on even an uncached 386 generally isn’t significant. (Remember that the 386 fetches four instruction bytes at a pop.) LOOP is three cycles slower than DEC/JNZ on the 486, and the 486 executes instructions in so few cycles that those three cycles mean that DEC/JNZ is nearly twice as fast as LOOP. Then, too, unlike LOOP, DEC doesn’t require that CX be used, so the DEC/JNZ solution is both faster and more flexible on the 386 and 486, and on the Pentium as well. (By the way, all this is not just theory; I’ve timed the relative performances of LOOP and DEC CX/JNZ on a cached 386, and LOOP really is slower.)

Things are stranger still for LOOP’s relative JCXZ, which branches if and only if CX is zero. JCXZ is faster than AND CX,CX/JZ on the 8088 and 80286, and equivalent on the 80386—but is about twice as slow on the 486!

By the way, don’t fall victim to the lures of JCXZ and do something like this:

```assembly
and cx,0fh ;Isolate the desired field
jcxz SkipLoop ;If field is 0, don’t bother
```

The AND instruction has already set the Zero flag, so this

```assembly
and cx,0fh ;Isolate the desired field
jz SkipLoop ;If field is 0, don’t bother
```

will do just fine and is faster on all processors. Use JCXZ only when the Zero flag isn’t already set to reflect the status of CX.

The Lessons of LOOP and JCXZ

What can we learn from LOOP and JCXZ? First, that a single instruction that is intended to do a complex task is not necessarily faster than several instructions that together do the same thing. Second, that the relative merits of instructions and optimization rules vary to a surprisingly large degree across the x86 family.
In particular, if you’re going to write 386 protected mode code, which will run only on the 386, 486, and Pentium, you’d be well advised to rethink your use of the more esoteric members of the x86 instruction set. **LOOP, JCXZ**, the various accumulator-specific instructions, and even the string instructions in many circumstances no longer offer the advantages they did on the 8088. Sometimes they’re just not any faster than more general instructions, so they’re not worth going out of your way to use; sometimes, as with **LOOP**, they’re actually slower, and you’d do well to avoid them altogether in the 386/486 world. Reviewing the instruction cycle times in the MASM or TASM manuals, or looking over the cycle times in Intel’s literature, is a good place to start; published cycle times are closer to actual execution times on the 386 and 486 than on the 8088, and are reasonably reliable indicators of the relative performance levels of x86 instructions.

**Avoiding LOOPS of Any Stripe**
Cycle counting and directly substituting instructions (**DEC CX/JNZ** for **LOOP**, for example) are techniques that belong at the lowest level of optimization. It’s an important level, but it’s fairly mechanical; once you’ve learned the capabilities and relative performance levels of the various instructions, you should be able to select the best instructions fairly easily. What’s more, this is a task at which compilers excel. What I’m saying is that you shouldn’t get too caught up in counting cycles because that’s a small (albeit important) part of the optimization picture, and not the area in which your greatest advantage lies.

**Local Optimization**
One level at which assembly language programming pays off handsomely is that of **local optimization**; that is, selecting the best sequence of instructions for a task. The key to local optimization is viewing the 80x86 instruction set as a set of building blocks, each with unique characteristics. Your job is to sequence those blocks so that they perform well. It doesn’t matter what the instructions are intended to do or what their names are; all that matters is what they do.

Our discussion of **LOOP** versus **DEC/JNZ** is an excellent example of optimization by cycle counting. It’s worth knowing, but once you’ve learned it, you just routinely use **DEC/JNZ** at the bottom of loops in 386/486-specific code, and that’s that. Besides, you’ll save at most a few cycles each time, and while that helps a little, it’s not going to make all that much difference.

Now let’s step back for a moment, and with no preconceptions consider what the x86 instruction set can do for us. The bulk of the time with both **LOOP** and **DEC/JNZ** is taken up by branching, which just happens to be one of the slowest aspects of every processor in the x86 family, and the rest is taken up by decrementing the count register and checking whether it’s zero. There may be ways to perform those tasks a
little faster by selecting different instructions, but they can get only so fast, and branching can’t even get all that fast.

The trick, then, is not to find the fastest way to decrement a count and branch conditionally, but rather to figure out how to accomplish the same result without decrementing or branching as often. Remember the Kobiyashi Maru problem in Star Trek? The same principle applies here: Redefine the problem to one that offers better solutions.

Consider Listing 7.1, which searches a buffer until either the specified byte is found, a zero byte is found, or the specified number of characters have been checked. Such a function would be useful for scanning up to a maximum number of characters in a zero-terminated buffer. Listing 7.1, which uses LOOP in the main loop, performs a search of the sample string for a period (‘.’) in 170 μs on a 20 MHz cached 386.

When the LOOP in Listing 7.1 is replaced with DEC CX/JNZ, performance improves to 168 μs, less than 2 percent faster than Listing 7.1. Actually, instruction fetching, instruction alignment, cache characteristics, or something similar is affecting these results; I’d expect a slightly larger improvement—around 7 percent—but that’s the most that counting cycles could buy us in this case. (All right, already; LOOPNZ could be used at the bottom of the loop, and other optimizations are surely possible, but all that won’t add up to anywhere near the benefits we’re about to see from local optimization, and that’s the whole point.)

LISTING 7.1  L7-1.ASM
: Program to illustrate searching through a buffer of a specified length until either a specified byte or a zero byte is encountered.
: A standard loop terminated with LOOP is used.

.model small
.stack 100h
.data
SampleString label byte
   db 'This is a sample string of a long enough length so that raw searching speed can outweigh any extra set-up time that may be required.',0
SAMPLE_STRING_LENGTH equ $-SampleString

Prompt db 'Enter character to search for:$'

ByteFoundMsg db 0dh,0ah
   db 'Specified byte found.',0dh,0ah,'$'
ZeroByteFoundMsg db 0dh,0ah
   db 'Zero byte encountered.',0dh,0ah,'$'
NoByteFoundMsg db 0dh,0ah
   db 'Buffer exhausted with no match.',0dh,0ah,'$'

Local Optimization 141
.code
Start proc near
mov ax,@data  ;point to standard data segment
mov ds,ax
mov dx,offset Prompt
mov ah,9      ;DOS print string function
int 21h       ;prompt the user
mov ah,1      ;DOS get key function
int 21h       ;get the key to search for
mov ah,al     ;put character to search for in AH
mov cx,SAMPLE_STRING_LENGTH ;# of bytes to search
mov si,offset SampleString  ;point to buffer to search
call SearchMaxLength ;search the buffer
mov dx,offset ByteFoundMsg ;assume we found the byte
jc  PrintStatus
;jc: assume we found the byte
;jc: didn't find the byte, figure out
;jc: whether we found a zero byte or
;jc: ran out of buffer
mov dx,offset NoByteFoundMsg
jc  PrintStatus
;jc: we didn't find a zero byte
;jc: we didn't find a zero byte
mov dx,offset ZeroByteFoundMsg
;jc: we found a zero byte
mov ah,9      ;DOS print string function
int 21h       ;DOS print string function
mov ah,4ch     ;return to DOS
int 21h       ;return to DOS
PrintStatus:
mov ah,9      ;DOS print string function
int 21h       ;DOS print string function
Start endp

; Function to search a buffer of a specified length until either a
; specified byte or a zero byte is encountered.
; Input:
;  AH - character to search for
;  CX - maximum length to be searched (must be > 0)
;  DS:SI - pointer to buffer to be searched
; Output:
;  CX - 0 if and only if we ran out of bytes without finding
;        either the desired byte or a zero byte
;  DS:SI - pointer to searched-for byte if found, otherwise byte
;        after zero byte if found, otherwise byte after last
;        byte checked if neither searched-for byte nor zero
;        byte is found
;  Carry Flag - set if searched-for byte found, reset otherwise

SearchMaxLength proc near
  cld
SearchMaxLengthLoop:
  lodsb         ;get the next byte
  cmp al,ah     ;is this the byte we want?
  jz  ByteFound ;yes, we're done with success
  and al,al     ;is this the terminating 0 byte?
  jz  ByteNotFound ;yes, we're done with failure
  loop SearchMaxLengthLoop ;it's neither, so check the next
    byte, if any
ByteNotFound:
  clc            ;return "not found" status
  ret
ByteFound:
  dec si        ;point back to the location at which
                ;we found the searched-for byte
  stc            ;return "found" status
142 Chapter 7
Unrolling Loops

Listing 7.2 takes a different tack, unrolling the loop so that four bytes are checked for each LOOP performed. The same instructions are used inside the loop in each listing, but Listing 7.2 is arranged so that three-quarters of the LOOPS are eliminated. Listings 7.1 and 7.2 perform exactly the same task, and they use the same instructions in the loop—the searching algorithm hasn’t changed in any way—but we have sequenced the instructions differently in Listing 7.2, and that makes all the difference.

**LISTING 7.2  L7-2.ASM**

: Program to illustrate searching through a buffer of a specified length until a specified zero byte is encountered.
: A loop unrolled four times and terminated with LOOP is used.

```
.model small
.stack 100h
.data
SampleString label byte
  db 'This is a sample string of a long enough length '  
  db 'so that raw searching speed can outweigh any '  
  db 'extra set-up time that may be required.',0
SAMPLE_STRING_LENGTH equ $-Samplestring
Prompt db 'Enter character to search for:$'
ByteFoundMsg db 0dh,0ah
  db 'Specified byte found.',0dh,0ah,'$'
ZeroByteFoundMsg db 0dh,0ah
  db 'Zero byte encountered.',0dh,0ah,'$'
NoByteFoundMsg db 0dh,0ah
  db 'Buffer exhausted with no match.',0dh,0ah,'$'

SearchMaxLengthEntryTable label word
  dw SearchMaxLengthEntry4
  dw SearchMaxLengthEntry1
  dw SearchMaxLengthEntry2
  dw SearchMaxLengthEntry3

.code
Start proc near
  mov ax,@data :point to standard data segment
  mov ds,ax
  mov dx,offset Prompt
  mov ah,9 :DOS print string function
  int 21h :prompt the user
  mov ah,1 :DOS get key function
  int 21h :get the key to search for
  mov ah,al :put character to search for in AH
```
mov cx, SAMPLE_STRING_LENGTH ; # of bytes to search
mov si, offset SampleString ; point to buffer to search
call SearchMaxLength ; search the buffer
mov dx, offset ByteFoundMsg ; assume we found the byte
jc PrintStatus
: we did find the byte
: we didn't find the byte, figure out whether we found a zero byte or ran out of buffer
mov dx, offset NoByteFoundMsg
jc xz PrintStatus
: assume we didn't find a zero byte
mov dx, offset ZeroByteFoundMsg
: we found a zero byte
mov ah, 9 ; DOS print string function
int 21h
mov ah, 4ch ; return to DOS
int 21h
Start endp

; Function to search a buffer of a specified length until either a specified byte or a zero byte is encountered.
; Input:
; AH - character to search for
; CX - maximum length to be searched (must be > 0)
; DS: SI - pointer to buffer to be searched
; Output:
; CX = 0 if and only if we ran out of bytes without finding either the desired byte or a zero byte
; DS: SI = pointer to searched-for byte if found, otherwise byte after last byte checked if neither searched-for byte nor zero byte is found
; Carry Flag = set if searched-for byte found, reset otherwise

SearchMaxLength proc near
cld
mov bx, cx
add cx, 3
: calculate the maximum # of passes
shr cx, 1
: through the loop, which is unrolled 4 times
shr cx, 1
and bx, 3
: calculate the index into the entry point table for the first, possibly partial loop
shl bx, 1
: prepare for a word-sized look-up
jmp SearchMaxLengthEntryTable[bx]
: branch into the unrolled loop to do the first, possibly partial loop

SearchMaxLengthLoop:

SearchMaxLengthEntry4:
1 lodsb ; get the next byte
cmp al, ah ; is this the byte we want?
jz ByteFound ; yes, we're done with success
and al, al
jz ByteNotFound ; yes, we're done with failure

SearchMaxLengthEntry3:
1 lodsb ; get the next byte
cmp al, ah ; is this the byte we want?
jz ByteFound ; yes, we're done with success
and al, al
jz ByteNotFound ; yes, we're done with failure

144 Chapter 7
SearchMaxLengthEntry2:
  lodsb ; get the next byte
  cmp al, ah ; is this the byte we want?
  jz ByteFound ; yes, we’re done with success
  and al, al ; is this the terminating 0 byte?
  jz ByteNotFound ; yes, we’re done with failure

SearchMaxLengthEntry1:
  lodsb ; get the next byte
  cmp al, ah ; is this the byte we want?
  jz ByteFound ; yes, we’re done with success
  and al, al ; is this the terminating 0 byte?
  jz ByteNotFound ; yes, we’re done with failure
  loop SearchMaxLengthLoop ; it’s neither, so check the next
    ; four bytes, if any

ByteNotFound:
  clc ; return “not found” status
  ret

ByteFound:
  dec si ; point back to the location at which
  ; we found the searched-for byte
  stc ; return “found” status
  ret

SearchMaxLength endp
end Start

How much difference? Listing 7.2 runs in 121 μs—40 percent faster than Listing 7.1, even though Listing 7.2 still uses LOOP rather than DEC CX/JNZ. (The loop in Listing 7.2 could be unrolled further, too; it’s just a question of how much more memory you want to trade for ever-decreasing performance benefits.) That’s typical of local optimization; it won’t often yield the order-of-magnitude improvements that algorithmic improvements can produce, but it can get you a critical 50 percent or 100 percent improvement when you’ve exhausted all other avenues.

The point is simply this: You can gain far more by stepping back a bit and thinking of the fastest overall way for the CPU to perform a task than you can by saving a cycle here or there using different instructions. Try to think at the level of sequences of instructions rather than individual instructions, and learn to treat x86 instructions as building blocks with unique characteristics rather than as instructions dedicated to specific tasks.

Rotating and Shifting with Tables
As another example of local optimization, consider the matter of rotating or shifting a mask into position. First, let’s look at the simple task of setting bit N of AX to 1.

The obvious way to do this is to place N in CL, rotate the bit into position, and OR it with AX, as follows:

```
MOV BX, 1
SHL BX, CL
OR AX, BX
```
This solution is obvious because it takes good advantage of the special ability of the x86 family to shift or rotate by the variable number of bits specified by CL. However, it takes an average of about 45 cycles on an 8088. It's actually far faster to precalculate the results, pass the bit number in BX, and look the shifted bit up, as shown in Listing 7.3.

**LISTING 7.3  L7-3.ASM**

```
SHL BX, 1 ; prepare for word sized look up
OR AX, ShiftTable[BX] ; look up the bit and OR it in

ShiftTable LABEL WORD
BIT_PATTERN=0001H
REPT 16
DW BIT_PATTERN
BIT_PATTERN-BIT_PATTERN SHL 1
ENDM
```

Even though it accesses memory, this approach takes only 20 cycles—more than twice as fast as the variable shift. Once again, we were able to improve performance considerably—not by knowing the fastest instructions, but by selecting the fastest sequence of instructions.

In the particular example above, we once again run into the difficulty of optimizing across the x86 family. The table lookup is faster on the 8088 and 286, but it's slightly slower on the 386 and no faster on the 486. However, 386/486-specific code could use enhanced addressing to accomplish the whole job in just one instruction, along the lines of the code snippet in Listing 7.4.

**LISTING 7.4  L7-4.ASM**

```
OR EAX, ShiftTable[EBX*4] ; look up the bit and OR it in

ShiftTable LABEL DWORD
BIT_PATTERN=0001H
REPT 32
DD BIT_PATTERN
BIT_PATTERN-BIT_PATTERN SHL 1
ENDM
```

Besides illustrating the advantages of local optimization, this example also shows that it generally pays to precalculate results; this is often done at or before assembly time, but precalculated tables can also be built at run time. This is merely one aspect of a fundamental optimization rule: Move as much work as possible out of your critical code by whatever means necessary.

**NOT Flips Bits—Not Flags**

The **NOT** instruction flips all the bits in the operand, from 0 to 1 or from 1 to 0. That's as simple as could be, but **NOT** nonetheless has a minor but interesting talent: It doesn't affect the flags. That can be irritating; I once spent a good hour tracking
down a bug caused by my unconscious assumption that \texttt{NOT} does set the flags. After all, every other arithmetic and logical instruction sets the flags; why not \texttt{NOT}? Probably because \texttt{NOT} isn’t considered to be an arithmetic or logical instruction at all; rather, it’s a data manipulation instruction, like \texttt{MOV} and the various rotates. (These are \texttt{RCR}, \texttt{RCL}, \texttt{ROR}, and \texttt{ROL}, which affect only the Carry and Overflow flags.) \texttt{NOT} is often used for tasks, such as flipping masks, where there’s no reason to test the state of the result, and in that context it can be handy to keep the flags unmodified for later testing.

\textbf{Tip} Besides, if you want to \texttt{NOT} an operand and set the flags in the process, you can just \texttt{XOR} it with \texttt{-1}. Put another way, the only functional difference between \texttt{NOT AX} and \texttt{XOR AX,0FFFFH} is that \texttt{XOR} modifies the flags and \texttt{NOT} doesn’t.

The x86 instruction set offers many ways to accomplish almost any task. Understanding the subtle distinctions between the instructions—whether and which flags are set, for example—can be critical when you’re trying to optimize a code sequence and you’re running out of registers, or when you’re trying to minimize branching.

\textbf{Incrementing with and without Carry}

Another case in which there are two slightly different ways to perform a task involves adding \texttt{1} to an operand. You can do this with \texttt{INC}, as in \texttt{INC AX}, or you can do it with \texttt{ADD}, as in \texttt{ADD AX,1}. What’s the difference? The obvious difference is that \texttt{INC} is usually a byte or two shorter (the exception being \texttt{ADD AL,1}, which at two bytes is the same length as \texttt{INC AL}), and is faster on some processors. Less obvious, but no less important, is that \texttt{ADD} sets the Carry flag while \texttt{INC} leaves the Carry flag untouched.

Why is that important? Because it allows \texttt{INC} to function as a data pointer manipulation instruction for multi-word arithmetic. You can use \texttt{INC} to advance the pointers in code like that shown in Listing 7.5 without having to do any work to preserve the Carry status from one addition to the next.

\textbf{Listing 7.5 L7-5.ASM}

```
CLC ;clear the Carry for the initial addition

LOOP_TOP:
  MOV AX,[SI];get next source operand word
  ADC [DI],AX;add with Carry to dest operand word
  INC SI ;point to next source operand word
  INC SI
  INC DI ;point to next dest operand word
  INC DI
  LOOP LOOP_TOP
```

If \texttt{ADD} were used, the Carry flag would have to be saved between additions, with code along the lines shown in Listing 7.6.
LISTING 7.6  L7-6.ASM

CLC  ;clear the carry for the initial addition
LOOP_TOP:
    MOV AX,[SI]  ;get next source operand word
    ADC [DI],AX  ;add with carry to dest operand word
    LAHF  ;set aside the carry flag
    ADD SI.2  ;point to next source operand word
    ADD DI.2  ;point to next dest operand word
    SAHF  ;restore the carry flag
    LOOP LOOP_TOP

It's not that the Listing 7.6 approach is necessarily better or worse; that depends on the processor and the situation. The Listing 7.6 approach is different, and if you understand the differences, you'll be able to choose the best approach for whatever code you happen to write. (DEC has the same property of preserving the Carry flag, by the way.)

There are a couple of interesting aspects to the last example. First, note that LOOP doesn't affect any flags at all; this allows the Carry flag to remain unchanged from one addition to the next. Not altering the arithmetic flags is a common characteristic of program control instructions (as opposed to arithmetic and logical instructions like SUB and AND, which do alter the flags).

The rule is not that the arithmetic flags change whenever the CPU performs a calculation; rather, the flags change whenever you execute an arithmetic, logical, or flag control (such as CLC to clear the Carry flag) instruction.

Not only do LOOP and JCXZ not alter the flags, but REP MOVS, which counts down CX to 0, doesn't affect the flags either.

The other interesting point about the last example is the use of LAHF and SAHF, which transfer the low byte of the FLAGS register to and from AH, respectively. These instructions were created to help provide compatibility with the 8080's (that's 8080, not 8088) PUSH PSW and POP PSW instructions, but turn out to be compact (one byte) instructions for saving and restoring the arithmetic flags. A word of caution, however: SAHF restores the Carry, Zero, Sign, Auxiliary Carry, and Parity flags—but not the Overflow flag, which resides in the high byte of the FLAGS register. Also, be aware that LAHF and SAHF provide a fast way to preserve the flags on an 8088 but are relatively slow instructions on the 486 and Pentium.

There are times when it's a clear liability that INC doesn't set the Carry flag. For instance

    INC AX
    ADC DX.0

does not increment the 32-bit value in DX:AX. To do that, you'd need the following:

    ADD AX.1
    ADC DX.0

As always, pay attention!