Assembly Language *Tricks of the Trade*

Hand-picked code for smaller, faster programs

Tim Paterson

It is the nature of assembly language programmers to always look for ways to make their programs faster and smaller. Over the years, the individual programmer develops a personal catalog of tricks and techniques that squeeze out a few bytes here or a few clocks there. My own catalog of 8086 tricks has been 13 years in the making, including a few from the 8080 that survived the translation.

One of the original motivations for finding some of these alternatives to the obvious approach is the severe "branch penalty" of the 8086 and 8088. When a conditional jump is taken on the 8086/8088, four times as many clock cycles are required (16) as when the jump is not taken. However, this penalty has been reduced on the 286 and 386. When taking a conditional jump, the newer processors require only seven clocks, plus one clock for each byte in the instruction at the target of the jump. That is, if you're jumping to an instruction that is 2 bytes long, the conditional jump takes nine clocks. This improvement means that several of the nine tricks presented here are of little or no value on the 286 and 386. However, I have presented them anyway so you'll know what they do if you see them. They are also still useful for code targeted to the 8086/8088.

For each of these tricks, I have compared its size and speed to the "direct" approach. Because the 286 is now the largest selling processor in PCs, I have used 286 clock counts to compare timing. When conditional jumps branch out of the presented code sequence, I assume the target instruction is 2-bytes long so that the branch would take nine clocks.

Tim is the original author of MS-DOS, Versions 1.x, which he wrote in 1980–82 while employed by Seattle Computer Products and Microsoft. He was also the founder of Falcon Technology, which was eventually sold, to Phoenix Technologies, the ROM BIOS maker. He can be reached through the DDJ *office.*

#1 Binary-to-ASCII Conversion

Converts a binary number in *AL*, range *0* to *0FH*, to the appropriate ASCII character.

```
    add    al,"0"              ;Handle 0 - 9
    cmp    al,"9"              ;Did it work?
    jbe    HaveAscii
    add    al,"A" - ("9" + 1)  ;Apply correction for 0AH -0FH
HaveAscii:
```

Direct approach: 8 bytes, 12 clocks for *0AH-0FH*, 15 clocks for *0–9*.

```
    add    al,90H    ;90H - 9FH
    daa              ;90H - 99H, 00H -05H +CY
    adc    al,40H    ;0D0H - 0D9H +CY, 41H - 46H
    daa              ;30H - 39H, 41H -46H = "0"-"9", "A"-"F"
```

Trick: 6 bytes, 12 clocks.

#2 Absolute Value

Find absolute value of signed integer in AX.

```
    or     ax,ax         ;Set flags
    jns    AxPositive    ;Already the right answer if positive
    neg    ax            ;It was negative, so flip sign
AxPositive:
```

Direct approach: 6 bytes, 7 clocks if negative, 11 clocks if positive.

```
    cwd                  ;Extend sign through dx
    xor    ax,dx         ;Complement ax if negative
    sub    ax,dx         ;Increment ax if it was negative
```

Trick: 5 bytes, 6 clocks.

(continued on page 32)

(continued from page 38)

#3 Smaller of Two Values ("MIN")

Given signed integers in AX and BX, return smaller in AX.

```
 cmp    ax,bx
 jl     AxSmaller
 xchg   ax,bx          ;Swap smaller into ax
AxSmaller:
```

Direct approach: 5 bytes, 8 clocks if *ax* >= *bx*, 11 clocks otherwise.

```
 sub    ax,bx     ;Could overflow if signs are different!!
 cwd              ;dx = 0 if ax >= bx, dx = 0FFFFH if ax < bx
 and    ax,dx     ;ax = 0 if ax >= bx, ax = ax - bx if ax < bx
 add    ax,bx     ;ax = bx if ax >=bx, ax = ax if ax < bx
```

Trick: 7 bytes, 8 clocks. Doesn't work if |*ax* - *bx*| > 32K. Not recommended.

#4 Convert to Uppercase

Convert ASCII character in AL to uppercase if it's lowercase, otherwise leave unchanged.

```
    cmp   al,"a"
    jb    CaseOk
    cmp   al,"z"
    ja    CaseOk
    sub   al,"a" - "A" ;In range "a" - "z", apply correction
CaseOk:
```

Direct approach: 10 bytes, 12 clocks if less than "a" (number, capital letter, control character, most symbols), 15 clocks if lowercase, 18 clocks if greater than "z" (a few symbols and graphics characters).

```
 sub    al,"a"              ;Lowercase now 0 - 25
 cmp    al,"z" - "a" +1     ;Set CY flag if lowercase
 sbb    ah,ah               ;ah = 0FFH if lowercase, else 0
 and    ah,"a" - "A"        ;ah = correction or zero
 sub    al,ah               ;Apply correction, lower to upper
 add    al,"a"              ;Restore base
```

Trick: 13 bytes, 16 clocks. Although occasionally faster, it is bigger and slower on the average. Not recommended. Used by Microsoft C 5.1 *stricmp( )* routine.

#5 Fast String Move

Assume setup for a standard string move, with *DS:SI* pointing to source, *ES:DI* pointing to destination, and byte count in CX. Double the speed by moving words, accounting for a possible odd byte.

```
      shr      cx,1       ;Convert to word count
rep   movsw               ;Move words
      jnc      AllMoved   ;CY clear if no odd byte
      movsb               ;Copy that last odd byte
AllMoved:
```

Direct: 7 bytes, 10 clocks if odd, 11 clocks if even (plus time for repeated move).

```
      shr      cx,1       ;Convert to word count
rep   movsw               ;Move words
      adc      cx,cx      ;Move carry back into cx
rep   movsb               ;Move one more if odd count
```

Trick: 8 bytes, 9 clocks if even, 13 clocks if odd (plus time for repeated move). Not recommended.

#6 Binary/Decimal Conversion

The 8086 instruction *AAM* (ASCII adjust for multiplication) is actually a binary-to-decimal conversion instruction. Given a binary number in *AL* less than 100, *AAM* will convert it directly to unpacked BCD digits in *AL* and *AH* (ones in *AL*, tens in *AH*). If the value in *AL* isn't necessarily less than 100, then *AAM* can be applied twice to return three *BCD* digits. For example:

```
 aam               ;al = ones, ah = tens & hundreds
 mov    cl,al      ;Save ones in cl
 mov    al,ah      ;Set up to do it again
 aam               ;ah = hundreds, al = tens, cl = ones
```

AAM is really a divide-by-ten instruction, returning the quotient in *AH* and the remainder in *AL*. It takes 16 clocks, which are actually two clocks more than a byte *DIV*. However, you easily save those two clocks and more with reduced setup. There's no need to extend the dividend to 16 bits, nor to move the value 10 into a register.

The inverse of the *AAM* instruction is *AAD* (ASCII adjust for division). It multiplies *AH* by 10 and adds it to *AL*, then zeros *AH*. Given two unpacked *BCD* digits (tens in *AH* and ones in *AL*), *AAD* will convert them directly into a binary number. Of course, given only two digits, the resulting binary number will be less than 100. But *AAD* can be used twice to convert three unpacked *BCD* digits, provided the result is less than 256. For example:

```
;ah = hundreds, al = tens, cl = ones
    aad             ;Combine hundreds and tens
    mov   ah,al
    mov   al,cl     ;Move ones to al
    aad             ;Binary result in ax, mod 256
```

AAD takes 14 clocks, which is one clock more than a byte MUL. Again, that time can be saved because of reduced setup.

#7 Multiple Bit Testing

Test for all four combinations of 2 bits of a flag byte in memory.

```
    mov     al,[Flag]
    test    al,Bit1
    jnz     Bit1Set
    test    al,Bit2
    jz      BothZero
Bit2Only:
    . . .

Bit1Set:
    test    al,Bit2
    jnz     BothOne
Bit1Only:
```

Direct approach: 15 bytes, up to 29 clocks (to *BothOne*).

The parity flag is often thought of as a holdover from earlier days, useful only for error detection in communications. However, it does have a useful application to cases such as this bit testing. Recall that the parity flag is EVEN if there are an even number of "one" bits in the byte being tested, and ODD otherwise. When testing only 2 bits, the parity flag will tell you if they are equal — it is EVEN for no "one" bits or for 2 "one" bits, ODD for 1 "one" bit.

The sign flag is also handy for bit testing, because it directly gives you the value of bit 7 in the byte. The obvious drawback is you only get to use it on 1 bit.

(continued on page 34)

(continued from page 32)

```
        test        [Flag],Bit1 + Bit2
        jz          BothZero
        jpe         BothOne ;Bits are equal, but not both zero
;One (and only one) bit is set
.erre  Bit1 EQ 80H           ;Verify Bit1 is the sign bit
        js          Bit1Only
Bit2Only:
```

Trick: 11 bytes, up to 21 clocks (to *Bit1Only*).

Note that the parity flag is only set on the low 8 bits of a 16-bit (or 32-bit 386) operation. Suppose you test 2 bits in a 16-bit word, where 1 bit is in the low byte while the other is in the high byte. The parity flag will be set on the value of the 1 bit in the low byte — EVEN if zero, ODD if one. This is potentially useful in certain cases of bit testing, as long as you are aware of it!

Another example of using dedicated bit positions is to assign flags to bits 6 and 7 of a byte. Then test it by shifting it left 1 bit. The carry and sign flags will directly hold the values in those 2 bits. In addition, the overflow flag will be set if the bits are different (because the sign has changed).

Finally, there is a way to test up to 4 bits at once. Loading the flag byte into *AH* and executing the *SAHF* instruction will copy bits 0, 2, 6, and 7 directly into the carry, parity, zero, and sign flags, respectively.

#8 Function Dispatcher

Given a function number in a register with value 0 to $n-1$, dispatch to the respective one of n functions.

```
;Function number in cx
    jcxz        Function0
    dec         cx
    jz          Function1
    dec         cx
    jz          Function2
    . . .
```

Direct approach 1: $3*n$ - 4 bytes, $5*n$ clocks maximum. Not bad for small n $(n < 10)$.

```
;Function number in bx
    shl         bx,1
    jmp         tDispatch[bx]
```

Direct approach 2: $2*n$ + 6 bytes, 15 clocks. The best approach for large n when speed is a consideration.

```
;Function number in cx
    jcxz    Function0
    loop    NotFunc1
Function1:
    . . .
NotFunc1:
    loop    NotFunc2
Function2:
    . . .
NotFunc2:
    loop    NotFunc3
Function3:
    . . .
```

Trick: $2*n$ - 2 bytes, $10*n$ - 16 clocks maximum. Slow, but compact.

#9 Skipping Instructions

Sometimes a routine will have two or more entry points, but the only difference between the entry points is the first instruction. For example, the instruction that differs from one entry point to the next could be initializing a register to different values to be used as a flag later on in the routine.

(continued on page 36)

(continued from page 34)

```
Entry1:
        mov     al,0
        jmp     Body

Entry2:
        mov     al,1
        jmp     Body

Entry3:
        mov     al,-1
Body:
```

Direct approach: 10 bytes, 11 clocks (from *Entry1*).

Instead of using jump instructions to skip over the alternative entry points, a somewhat sleazy trick allows you to simply skip over those instructions. The technique goes back at least to 1975 with the first Microsoft Basic for the 8080. It became known as a "LXI trick" (pronounced "liksee"), after the 8080 mnemonic for a 16-bit *move-immediate* into register. Essentially, it allows you to skip a 2-byte instruction by hiding it as immediate data. A variation, the "MVI trick" (pronounced "movie"), uses an 8-bit immediate instruction to hide a 1-byte instruction.

Applied to the 8086, there is another variation. The skip can use a *move-immediate* instruction and destroy the contents of one register, or it can use a *compare-immediate* instruction and destroy the flags. Using the latter case the example above could be code such as this:

```
SKIP2F  MACRO
        db      3DH     ;Opcode byte for CMP AX,<immed>
        ENDM
Entry1:
        mov     al,0
        SKIP2F          ;Next 2 bytes are immediate data
Entry2:
        mov     al,1
        SKIP2F          ;Next 2 bytes are immediate data
Entry3:
        mov     al,-1
Body:
```

The effect of this when entered at *Entry1* is:

```
Entry1:
     mov          al,0
     cmp          ax,01B0H     ;Data is MOV AL,1
     cmp          ax,0FFB0H    ;Data is MOV AL,-1
Body:
```

Trick: 8 bytes, 8 clocks (from *Entry1*).

This trick should always be hidden in a macro. Here is a more complete macro that requires an argument specifying what register or flags to destroy. The argument is any 16-bit general register or "F" for flags.

```
SKIP2   MACRO      ModReg
IFIDNI  <ModReg>,<f>       ;Modify flags?
        db         3DH     ;Opcode byte for CMP AX,<immed>
ELSE
?_i     =          0
        IRP        Reg,<ax,cx,dx,bx,sp,bp,si,di>
IFIDN   <ModReg>,<Reg>     ;Find the register in list yet?
        db         0B8H + ?_i
        EXITM
ELSE
?_i     =          ?_i + 1
ENDIF   ;IF ModReg = Reg
        ENDM       ;IRP
.errnz  ?_i EQ 8           ;Flag an error if no match
ENDIF   ;IF ModReg = F
        ENDM       ;SKIP2

;Examples
        SKIP2      f       ;Modify flags only
        SKIP2      ax      ;Destroy ax, flags preserved
```

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**