PARALLEL

Assembly Language Macros

By Ken Skier, March 01, 1991

Assembly language macros make code more readable without sacrificing the traditional assembly language benefits of small code size and top performance.

Ken learned the program by hand-assembling object code for a KIM-1 microcomputer with 1K of RAM in 1978. Now he writes word processing software, desktop publishing, and utility software for PCs. His word processor, Eye Relief, won two Excellence in Software Awards from the Software Publishers Association in 1990. Ken can be reached at SkiSoft Publishing Corporation, 1644 Massachusetts Avenue, Suite 79, Lexington, MA 02173. Phone: 617-863-1876, Fax 617-861-0086.

There are a lot of myths about assembly language. Perhaps the most persistent of these is that assembly language programs are hard to write, hard to debug, and nigh onto impossible to maintain because they are hard to read.

(Of course, in many cases it's not a myth. Sometimes when I look through assembly language code in a book or magazine, I find myself subvocalizing, writing in the margins, trying to interpret the programmer's intent from a series of MOV and COMPARE instructions. What is he trying to do? And what is actually going on here?)

No wonder people avoid assembly language.

But I write only in assembly language, and have done so for over a decade. During that time I have created whole applications from assembly language, including graphical word processors and a desktop publishing program. They're smaller than competitive applications, they run faster, and I've found that it takes me less time to develop applications in assembly language than for others to do so in C.

Yet you can look through all of my source code and never find a COMPARE instruction. In fact, you might look through my code and scratch your head, trying to figure out what language it's written in. Certainly not C or Pascal, but it doesn't look much like assembly language, either.

That's because I write in macros. I've developed, in effect, a language of macros so I can write readable source, with all the traditional benefits of assembly language -- small code size and excellent performance -- without the penalty of impenetrable code.

Comparisons

Let me give you an example. Suppose we need a routine to let us know whether some character we've got is an ASCII digit. We'll pass it the character in AL, and we want it to return TRUE if AL is in the range '0'...'9'; FALSE otherwise.

Throughout my code, I use Carry as a TRUE flag, because it's easy to set, easy to test, and doesn't interfere with any

of the 80x6 general registers. So we can write that routine like this:

```
  A_DIGIT      PROC
           CMP   AL, '0'
           JB      false
           CMP   AL, '9'
           JA      false
           STC
           RET

   __false:     CLC
           RET
    endp
```

This little routine is about as simple as standard assembly language gets, but it does require you to interpret the 8086 instructions, and from them figure out the programmer's intent. I wrote these few lines just a moment ago, and yet when I look at them, I don't find their function clear. I have to subvocalize: "Compare AL to an ASCII zero. Jump if Below. Below? Ah, if AL is Below the ASCII zero ... then jump to __false. So if AL is less than an ASCII zero, go to __false."

I have to go through that kind of interpretation with every couple of lines of standard assembly language. I don't like that. I want to take a single thought and write it down as a single line of code -- not as two cryptic lines that have to be put together to make sense. Even the act of returning a status code takes two lines in standard assembly language: One to set (or clear) the carry flag, and another line to return. So a single thought ("Return TRUE") is encrypted as two lines of code: STC, RET.

I really want to write that subroutine like this:

```
  A_DIGIT:
      IF_AL < '0', return FALSE.
      IF_AL > '9', return FALSE.
      Else ... return TRUE.
```

In practice, my code doesn't look quite that clear, but it's getting there. I have a subroutine in my code library that looks like this:

```
  A_DIGIT      PROC
      IF_AL {,   '0',   __false
      IF_AL },   '9',   __false

    RET_TRUE

   __false: RET_FALSE

   endp
```

Does this look like assembly language? Probably not. But it assembles as exactly the same code shown in the first example. It's just a lot easier to write and to read.

This routine incorporates three of my most heavily-used macros: IF_AL, RET_TRUE, and RET_FALSE.

RET_TRUE and RET_FALSE are pretty simple. RET_TRUE returns with carry set; RET_FALSE returns with carry clear. Why bother with a macro to do something so trivial? Because the most important effect of these macros is not on the object code, but on the READER. Look at the line "RET_TRUE" in a subroutine and you can tell immediately what the programmer wants to happen; look at some carry manipulation and you've got to think about it.

I use macros often to make my intent clear.

The other macro, IF_AL, is a real workhorse. It lets me express a single thought in a single line of code -- and it lets me do so visually, using a natural algebraic notation (as opposed to such confusing mnemonics as JLE, JGE, and JBE).

When I want to write:

```
  IF_AL < 12, goto __label
```

I can write it like this:

```
  IF_AL {, 12, __label
```

The macro name "IF_AL" start's the line. It is followed by a visual notation implying a relationship, IF_AL "understands" the following notation:

```
   {    less than
   {=   less than or equal
    =   equal
   {}   not equal
   }    greater than
   }=   greater than or equal
```

(If I had my druthers, I would use "<" and ">" instead of "{" and "}", but TASM treats angle brackets as delimiters, so I can't use them as arguments to a macro. Still, braces look enough like "<" and ">" for me to feel pretty comfortable with this implementation.)

The macro expands in a pretty straight forward way:

```
  IF_AL arg1, arg2, arg3
```

expands to a CMP AL, arg2, followed by a conditional jump to arg3. (Arg1 defines which conditional jump to use.) Some extra bookkeeping within the macro allows it to accept four arguments, if one of them is OFFSET or PTR.

As you might expect, I have similar macros named IF_AH, IF_AX, IF_BL, IF_BH, IF_BX, IF_CL, and so on for every register in the 80x6. So I never have to write a CMP instruction, and I never have to figure out which conditional JMP instruction to use. I leave all of those details to my macros.

Procedure Calls

I mentioned earlier that throughout my code, I use the Carry flag to indicate whether a routine has returned TRUE or FALSE. For example, I have a procedure that asks the user to confirm whether the program should go ahead and do something. That procedure (called "CONFIRM") displays a message and waits for a key. The user can press Enter or 'Y' or "y" for yes; ESC or "N" or "n" for no. (Any other key makes CONFIRM beep; it then waits until the user presses a legal key.) When it receives a legal key, CONFIRM returns TRUE or FALSE, to report the user's intent.

I could use the CALL instruction to invoke CONFIRM, and then use JC or JNC to branch based on carry. But doing so would take two lines of code, and involve testing carry in a way whose meaning might not be clear. After all, I don't really care about carry as a bit; I care about whether a procedure is returning "true" or "false."

So I have two macros that handle this situation: IF_ and IF_NOT.

```
  IF_ arg1, arg2
```

expands to:

```
  CALL  arg1
  JC    arg2
```

and

```
  IF_NOT arg1, arg2
```

expands to:

```
  CALL  arg1
  JNC   arg2
```

These extremely simple macros contribute greatly to the readability of my code. Now, instead of writing two lines of conventional assembly language such as this:

```
;
CALL CONFIRM
JC   do_it
```

I write one line using my macro:

```
    IF_CONFIRM, do_it
```

In this line of code, the intent of the programmer is clear. "If the user confirms, then go ahead and do it!"

Tables

I use a lot of tables in my code. Tables are small, efficient, and lend themselves to easy modification. For example, I will often use a table to translate an 8-bit character to a 16-bit procedure address. "The user just pressed this function key. What procedure should I invoke?" The table consists of a series of entries, where each entry is a BYTE followed by a WORD.

I would really like to be able to set up the table like this:

```
    db a_key
    dw a_procedure
    db other_key
    dw other_procedure
       .
       .
       .
    db 0
```

This would let me have each key and its associated procedure on the same line of the table. But TASM won't let me put two instructions on the same line. So I have to do this:

```
    db a_key
    dw a_procedure
    db other_key
    dw other_procedure
       .
       .
       .
    db 0
```

I don't like that at all. It fails to show the direct relationship between a key and its corresponding procedure. (And I will be in terrible trouble if I delete a DB line without deleting the DW line below it!) But I can't put a DB and a DW on the same line ... or I thought I couldn't, until I decided to create a macro.

My macro, DBW, lets me put a DB and then a DW on the same line.

```
   DBW   arg1, arg2
```

expands to:

```
    db    arg1
    dw    arg2
```

Using the DBW macro, I can make my tables look the way they should:

```
    dbw   a_key,       a_procedure
    dbw   other_key,   other_procedure
             .              .
             .              .
  dbw   0            0
```

Equates

Macros aren't the only way to make your code more readable. In many cases you can use equates, instead.

I use equates to help me write code in a style that I find more natural. For example, I learned to program in 6502 assembly language, so I am accustomed to writing JSR instead of CALL, RTS instead of RET, and SEC instead of STC. Rather than reprogram my brain to type the correct mnemonics for Intel 8Ox6 assembly language, I just put a few lines into the MACROS file (see the macro definition file in Listing One) that I include at the start of every module:

```
JSR   equ   <CALL>
RTS   equ   <RET>
SEC   equ   <STC>
```

Now when I type JSR, TASM knows I mean "CALL" ... when I type SEC, TASM knows I mean "STC" ... and when I type RTS, TASM knows I mean RET. That doesn't enhance readability, exactly (except to another 80x6 programmer who cut his teeth on the 6502!) but I like the idea of teaching TASM to accommodate me, rather than vice versa.

Before I ever wrote a line of code, I was a writer and a teacher of writing. I believed very strongly that writing should be read aloud, and that a good piece of writing is clear to the person who reads it. Now I write word processors instead of novels, screen drivers instead of dialogues, but I find the basic process of writing is the same, whether I am writing English narrative or assembly language source code. It has to be readable. It has to make sense to someone else.

Fortunately, as an assembly language programmer I can use macros to make my code more readable. I think it makes me more productive. And it makes it possible for me to understand my code long after I've written it.

After all, the most efficiently-programmed code in the world isn't going to do you much good if you can't understand it when you look at it in your editor.

ASSEMBLY LANGUAGE MACROS
by Ken Skier

[LISTING ONE]

```
<a name="00a7_0009">

;-------------------------------------------;
;               SkiSoft Macros              ;
;                                           ;
;     Copyright (c) 1990 by SkiSoft, Inc.   ;
;            All rights reserved.           ;
;                                           ;
;                                           ;
;            Created by Ken Skier           ;
;                                           ;
;   SkiSoft, Inc.                           ;
;   1644 Massachusetts Avenue, Suite 79     ;
;   Lexington, MA 02173                     ;
;   Tel: (617) 863-1876  Fax:  (617 861-0086 ;
;                                           ;
;-------------------------------------------;



@       EQU     OFFSET

JSR     equ     CALL
RTS     equ     RET
SEC     equ     STC


IF_  MACRO  sub, dest
        CALL sub
```

```
         JC dest
   ENDM


IF_NOT  MACRO  sub, dest
          CALL sub
         JNC dest
 ENDM



RET_FALSE  MACRO
         CLC
         RET
 ENDM



RET_TRUE  MACRO
          STC
         RET
 ENDM



;--------------------------------------;
;                                      ;
;    16-bit Register Compare macros    ;
;                                      ;
;--------------------------------------;


IF_AX   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
          IF_REG16 AX, exp, val, dest, last
 ENDM

IF_BP   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
          IF_REG16 BP, exp, val, dest, last
 ENDM

IF_BX   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
          IF_REG16 BX, exp, val, dest, last
 ENDM

IF_CX   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
          IF_REG16 CX, exp, val, dest, last
 ENDM

IF_DX   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
          IF_REG16 DX, exp, val, dest, last

 ENDM

IF_SI   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
          IF_REG16 SI, exp, val, dest, last
 ENDM

IF_SP   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
          IF_REG16 SP, exp, val, dest, last
 ENDM

IF_DI   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
```

```
          IF_REG16 DI, exp, val, dest, last
 ENDM


IF_REG16 MACRO reg, exp, val, dest, last
          %POPLCTL       ;; Restore source-level listing parameters.
        IFIDNI <val>, <@>
           CMP reg, @ dest
          %PUSHLCTL
          %NOLIST
          IFITS_ exp, last
        ELSE
           CMP reg, word ptr val
          %PUSHLCTL
          %NOLIST
          IFITS_ exp, dest
        ENDIF
 ENDM


;---------------------------------------;
;                                       ;
;     8-bit Register Compare macros     ;
;                                       ;
;---------------------------------------;


IF_AL   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
        IF_REG8 AL, exp, val, dest, last
 ENDM


IF_AH   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
        IF_REG8 AH, exp, val, dest, last
 ENDM


IF_BL   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
        IF_REG8 BL, exp, val, dest, last
 ENDM

IF_BH   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
        IF_REG8 BH, exp, val, dest, last
 ENDM

IF_CL   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
        IF_REG8 CL, exp, val, dest, last
 ENDM

IF_CH   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
        IF_REG8 CH, exp, val, dest, last
 ENDM

IF_DL   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
        IF_REG8 DL, exp, val, dest, last
 ENDM

IF_DH   MACRO exp, val, dest, last
          %PUSHLCTL
          %NOLIST
        IF_REG8 DH, exp, val, dest, last
 ENDM
```

```
IF_REG8 MACRO reg, exp, val, dest, last
          %POPLCTL       ;; Restore source-level listing parameters.
        IFIDNI <val>, <@>
           CMP reg, @ dest
          %PUSHLCTL
          %NOLIST
          IFITS_ exp, last
        ELSE
           CMP reg, byte ptr val
          %PUSHLCTL
          %NOLIST
          IFITS_ exp, dest
        ENDIF
 ENDM


IFITS_  MACRO exp, dest
        %POPLCTL              ;;      Restore source-level listing parameters.
        IFIDNI  <exp>, <{>    ;;  <
           JB dest
    elseIFIDNI  <exp>, <=>     ;;  =
           JE dest
    elseIFIDNI  <exp>, <}>    ;;  >
           JA dest
    elseIFIDNI  <exp>, <{=>   ;;  <
           JBE dest
    elseIFIDNI  <exp>, <{}>  ;;
           JNE dest
    elseIFIDNI  <exp>, <}=>  ;;  >=
           JAE dest
        ENDIF
  ENDM



IFITS  MACRO exp, dest
        %PUSHLCTL
        %LIST
        IFIDNI  <exp>, <{>    ;;  <
           JB dest
    elseIFIDNI  <exp>, <=>     ;;  =
           JE dest
    elseIFIDNI  <exp>, <}>    ;;  >
           JA dest
    elseIFIDNI  <exp>, <{=>   ;;  <
           JBE dest
    elseIFIDNI  <exp>, <{}>  ;;
           JNE dest
    elseIFIDNI  <exp>, <}=>  ;;  >=
           JAE dest
        ENDIF
        %POPLCTL
  ENDM

IF_ITS  EQU     IFITS


;--------------------------------------;
;                                      ;
;    End of SkiSoft macros.            ;
;                                      ;
;--------------------------------------;
```

Related Reading

- News
- Commentary

- **Java Plumbr Unlocks Threads**
- **Parallels Supports Docker Apps**
- **Google's Data Processing Model Hardens Up**
- **Devart dbForge Studio For MySQL With Phrase Completion**

More News»

- Slideshow
- Video

- **Jolt Awards 2015: Coding Tools**
- **2014 Developer Salary Survey**
- **C++ Reading List**
- **2012 Jolt Awards: Mobile Tools**

More Slideshows»

- Most Popular

- **RESTful Web Services: A Tutorial**
- **Lambda Expressions in Java 8**
- **Developer Reading List: The Must-Have Books for JavaScript**
- **An Algorithm for Compressing Space and Time**

More Popular»