Macros and Conditional Assembly

Chapter 10
S. Dandamudi

Outline

- What are macros?
- Macros with parameters
  * Parameter passing
  * Types of parameters
  * Invocation mechanism
  * When are macros better?
- Labels in macros
- Comments in macros
- Macro operators
- List control directives
- Repeat block directives
  * REPT directive
  * WHLE directive
  * IRP and IRPC directives
- Conditional assembly
  * IF and IFE
  * IFDEF and IFNDEF
  * IFB and IFNB
  * IFIDN and IFDIF
- Nested macros
- Performance: Macros vs procedures
What Are Macros?

• Macros provide a means to represent a block of text (code, data, etc.) by a name (*macro name*)

• Macros provide a sophisticated text substitution mechanism

• Three directives

  * =
  
  Example:  \texttt{CLASS\_SIZE} = 90  (can be redefined later)

  * \texttt{EQU}
  
  Example:  \texttt{CLASS\_SIZE EQU 90}

  * \texttt{MACRO}

What Are Macros? (cont’d)

• Macros can be defined with \texttt{MACRO} and \texttt{ENDM}

• Format

\[
\text{macro\_name} \quad \text{MACRO} \quad [\text{parameter1, parameter2, …}]
\]
\[
\text{macro body}
\]
\[
\text{ENDM}
\]

• A macro can be invoked using

\[
\text{macro\_name} \quad [\text{argument1, argument2, …}]
\]

Example:

<table>
<thead>
<tr>
<th>Definition</th>
<th>Invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{multAX_by_16}\texttt{ MACRO}</td>
<td>...</td>
</tr>
<tr>
<td>\texttt{sal AX, 4} \texttt{ mov AX, 27}</td>
<td>\texttt{multAX_by_16}</td>
</tr>
<tr>
<td>\texttt{ENDM}</td>
<td>...</td>
</tr>
</tbody>
</table>
Macros with Parameters

- Macros can be defined with parameters
  - More flexible
  - More useful

Example

\[
\text{mult\_by\_16} \quad \text{MACRO} \quad \text{operand} \\
\quad \text{sal} \quad \text{operand, 4} \\
\quad \text{ENDM}
\]

- To multiply a byte in DL register
  \[
  \text{mult\_by\_16} \quad \text{DL}
  \]
- To multiply a memory variable \text{count}
  \[
  \text{mult\_by\_16} \quad \text{count}
  \]

Macros with Parameters (cont’d)

Example: To exchange two memory words

\[
\text{Wmxchg} \quad \text{MACRO} \quad \text{operand1, operand2} \\
\quad \text{xchg} \quad \text{AX, operand1} \\
\quad \text{xchg} \quad \text{AX, operand2} \\
\quad \text{xchg} \quad \text{AX, operand1} \\
\quad \text{ENDM}
\]

Example: To exchange two memory bytes

\[
\text{Bmxchg} \quad \text{MACRO} \quad \text{operand1, operand2} \\
\quad \text{xchg} \quad \text{AL, operand1} \\
\quad \text{xchg} \quad \text{AL, operand2} \\
\quad \text{xchg} \quad \text{AL, operand1} \\
\quad \text{ENDM}
\]
Macros vs. Procedures

• Similar to procedures in some respects
  ∗ Both improve programmer productivity
  ∗ Aids development of modular code
  ∗ Can be used when a block of code is repeated in the source program

• Some significant differences as well
  ∗ Parameter passing
  ∗ Types of parameters
  ∗ Invocation mechanism

Macros vs. Procedures (cont’d)

Parameter passing
• In macros, similar to a procedure call in a HLL
  \texttt{mult	extunderscore by	extunderscore 16} \texttt{AX}

• In procedures, we have to push parameters onto the stack
  \begin{verbatim}
  push    AX
  call    times16
  \end{verbatim}

• Macros can avoid
  ∗ Parameter passing overhead
    – Proportional to the number of parameters passed
  ∗ \texttt{call/ret} overhead
Macros vs. Procedures (cont’d)

Types of parameters

* Macros allow more flexibility in the types of parameters that can be passed
  – Result of it being a text substitution mechanism

Example

```
shift    MACRO    op_code, operand, count
         op_code, operand, count
         ENDM
```

can be invoked as

```
shift    sal, AX, 3
```

which results in the following expansion

```
sal    AX, 3
```

Macros vs. Procedures (cont’d)

Invocation mechanism

* Macro invocation
  » Done at assembly time by text substitution

* Procedure invocation
  » Done at run time

• Tradeoffs

<table>
<thead>
<tr>
<th>Type of overhead</th>
<th>Procedure</th>
<th>Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory space</td>
<td>lower</td>
<td>higher</td>
</tr>
<tr>
<td>Execution time</td>
<td>higher</td>
<td>lower</td>
</tr>
<tr>
<td>Assembly time</td>
<td>lower</td>
<td>higher</td>
</tr>
</tbody>
</table>
When Are Macros Better?

• Useful to extend the instruction set by defining macro-instructions

```assembly
times16 PROC
    push BP
    mov BP, SP
    push count
    push AX
    call times16
    mov AX, [BP+4]
    pop count
    sal AX, 4
    pop [BP+4], AX
    pop BP
ret 2

times16 ENDP
```

Invocation

| push   | count
|-------|-----|
| call   | times16
| pop    | count

Too much overhead
Use of procedure is impractical

When Are Macros Better? (cont’d)

• Sometimes procedures cannot be used
  » Suppose we want to save and restore BX, CX, DX, SI, DI, and BP registers
  » Cannot use `pusha` and `popa` as they include AX as well

```assembly
save_regs MACRO
    push BP
    push DI
    push SI
    push DX
    push CX
    push BX

restore_regs MACRO
    pop BX
    pop CX
    pop SI
    pop DI
    pop BP

ENDM
```

<table>
<thead>
<tr>
<th>save_regs</th>
<th>MACRO</th>
</tr>
</thead>
<tbody>
<tr>
<td>push BP</td>
<td>pop BP</td>
</tr>
<tr>
<td>push DI</td>
<td>pop CX</td>
</tr>
<tr>
<td>push SI</td>
<td>pop SI</td>
</tr>
<tr>
<td>push DX</td>
<td>pop DI</td>
</tr>
<tr>
<td>push CX</td>
<td>pop BP</td>
</tr>
<tr>
<td>push BX</td>
<td>ENDM</td>
</tr>
</tbody>
</table>
Labels in Macros

• Problem with the following macro definition

```
to_upper0   MACRO    ch
    cmp    ch, 'a'
    jb     done
    cmp    ch, 'z'
    ja     done
    sub    ch, 32

    done:
    ENDM
```

* If we invoke it more than once, we will have duplicate label `done`

Labels in Macros (cont’d)

• Solution: Use LOCAL directive

• Format: `LOCAL local_label1 [,local_label2, ...]`

```
to_upper   MACRO  ch
    LOCAL  done
    cmp    ch, 'a'
    jb     done
    cmp    ch, 'z'
    ja     done
    sub    ch, 32

    done:
    ENDM
```

Assembler uses labels ??XXXX where XXXX is between 0 and FFFFH

To avoid conflict, do not use labels that begin with ??
Comments in Macros

- We don’t want comments in a macro definition to appear every time it is expanded
  » The ;; operator suppresses comments in the expansions

```assembly
; Converts a lowercase letter to uppercase.
to_upper MACRO ch
    LOCAL done
    ; case conversion macro
    cmp     ch,'a'  ;; check if ch >= 'a'
    jb      done
    cmp     ch,'z'  ;; and if ch >= 'z'
    ja      done
    sub     ch,32   ;; then ch := ch - 32
    done:
ENDM
```

Comments in Macros (cont’d)

- Invoking the `to_upper` macro by

```assembly
mov     AL, 'b'
to_upper AL
mov     BL, AL
mov     AH, '1'
to_upper AH
mov     BH, AH
```

generates the following macro expansion
Comments in Macros (cont’d)

    17 0000  B0 62                  mov     AL, 'b'
    18                              to_upper  AL
    19                              ; case conversion macro
    20 0002  3C 61                  cmp     AL, 'a'
    21 0004  72 06                  jb      ??0000
    22 0006  3C 7A                  cmp     AL, 'z'
    23 0008  77 02                  ja      ??0000
    24 000A  2C 20                  sub     AL, 32
    25 000C                  ??0000:
                    26 000C  8A D8                  mov     BL, AL
                    27 000E  B4 31                  mov     AH, '1'

Comments in Macros (cont’d)

    28                              to_upper  AH
    29                              ; case conversion macro
    30 0010  80 FC 61               cmp     AH, 'a'
    31 0013  72 08                  jb      ??0001
    32 0015  80 FC 7A               cmp     AH, 'z'
    33 0018  77 03                  ja      ??0001
    34 001A  80 EC 20               sub     AH, 32
    35 001D                  ??0001:
                    36 001D  8A FC                  mov     BH, AH
Macro Operators

• Five operators
  ;; Suppress comment
  & Substitute
  < > Literal-text string
  ! Literal-character
  % Expression evaluate

* We have already seen ;; operator
* We will discuss the remaining four operators

Macro Operators (cont’d)

Substitute operator (&)

* Substitutes a parameter with the actual argument

• Syntax: &name

sort2  MACRO  cond, num1, num2
LOCAL  done
push    AX
mov     AX, num1
cmp     AX, num2
j&cond  done
xchg    AX, num2
mov     num1, AX
done:
  pop    AX
ENDM
Macro Operators (cont’d)

• To sort two unsigned numbers value1 and value2, use

\[ \text{sort2 \ ae, value1, value2} \]

This generates the following macro expansion:

\[
\begin{align*}
push & \ AX \\
mov & \ AX, value1 \\
cmp & \ AX, value2 \\
jae & ??0000 \\
xchg & \ AX, value2 \\
mov & \ value1, AX \\
??0000: & \\
pop & \ AX \\
\end{align*}
\]

• To sort two signed numbers value1 and value2, use

\[ \text{sort2 \ ge, value1, value2} \]

Macro Operators (cont’d)

Literal-text string operator (< >)

* Treats the enclosed text as a single string literal rather than separate arguments

* Syntax: \(<text>\)

\[
\begin{align*}
\text{range_error1} & \ \text{MACRO \ number, variable, range} \\
\text{err_msg}&\text{number} & \ \text{DB} & \ \&\text{variable: out of range}, 0 \\
\text{range_msg}&\text{number} & \ \text{DB} & \ \&\text{Correct range is &range}, 0 \\
\text{ENDM} & \\
\end{align*}
\]

• Invoking with

\[ \text{range_error1 \ 1, <Assignment mark>, <0 to 25>} \]

produces

\[
\begin{align*}
\text{err_msg1} & \ \text{DB} & \ \&\text{Assignment mark: out of range}, 0 \\
\text{range_msg1} & \ \text{DB} & \ \&\text{Correct range is 0 to 25}, 0 \\
\end{align*}
\]
Macro Operators (cont’d)

Literal-character operator (!)
* Treats the character literally without its default meaning
* Syntax: !character

range_error2 MACRO number,variable,range
err_msg&number DB ' &variable: out of range - &range',0 ENDM

• Invoking with
range_error2 3,mark,<can!’t be !> 100>

produces
err_msg3 DB ‘mark: out of range - can’’t be > 100’,0

* Without the ! operator, two single quotes will produce a single quote in the output

Macro Operators (cont’d)

Expression Evaluate operator (%)
* Expression is evaluated and its value is used to replace the expression itself
* Syntax: %expression

init_array MACRO element_size,name,size,init_value
name D&element_size size DUP (init_value)
ENDM

• Assuming NUM_STUDENTS EQU 47
NUM_TESTS EQU 7

Invoking with
init_array W,marks,%NUM_STUDENTS*NUM_TESTS,-1
produces
marks DW 329 DUP (-1)
List Control Directives

- Control the contents of `.LST` file

- Two directives control the source lines
  - `.LIST` Allows listing of subsequent source lines
    - Default mode
  - `.XLIST` Suppresses listing of subsequent source lines

- Macro invocation call directives
  - `.LALL` Enables listing of macro expansions
  - `.SALL` Suppresses listing of all statements in macro expansions
  - `.XALL` Lists only the source statements in a macro expansion that generates code or data

Repeat Block Directives

- Three directives to repeat a block of statements
  - `REPT`
  - `WHILE`
  - `IRP/IRPC`

- Mostly used to define and initialize variables in a data segment

- Each directive identifies the beginning of a block
  - `ENDM` indicates the end of a repeat block
Repeat Block Directives (cont’d)

REPT directive
- Syntax:
  ```assembly
  REPT expression
  macro-body
  ENDM
  *
  macro-body is repeated expression times
  ```

<table>
<thead>
<tr>
<th>mult_16</th>
<th>MACRO</th>
<th>operand</th>
<th>mult_16</th>
<th>MACRO</th>
<th>operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPT 4</td>
<td>sal operand,1</td>
<td></td>
<td>sal operand,1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENDM</td>
<td></td>
<td></td>
<td>ENDM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

WHILE directive
- Syntax:
  ```assembly
  WHILE expression
  macro-body
  ENDM
  *
  macro-body is executed until expression is false (0)
  ```
- Following code produces cubed data table
  ```assembly
  WHILE int_value LT NUM_ENTRIES
  DW int_value*int_value*int_value
  int_value = int_value+1
  ENDM
  ```
Repeat Block Directives (cont’d)

IRP and IRPC directives

IRP - Iteration RePeat
IRPC - Iteration RePeat with Character substitution

• IRP directive
• Syntax:

\[
\text{IRP } \text{parameter}, \langle \text{argument1[,argument2,…]} \rangle \\
\text{macro-body}
\]

ENDM

* Angle brackets are required
* Arguments are given as a list separated by commas

» During the first iteration \text{argument1} is assigned to \text{parameter}, \text{argument2} during the second iteration, ...

Repeat Block Directives (cont’d)

IRP example

```
.DAT A
IRP value, <9,6,11,8,13>
DB   value
ENDM
```

produces

```
.DAT A
DB   9
DB   6
DB   11
DB   8
DB   13
```
Repeat Block Directives (cont’d)

- IRPC directive

  Syntax:

  \[
  \text{IRPC parameter, string} \\
  \quad \text{macro-body} \\
  \text{ENDM}
  \]

  * `macro-body` is repeated once for each character in `string`
  * `string` specifies
    - the number of iterations
    - the character to be used in each iteration
  * During the first iteration first character of `string` is assigned to `parameter`, second character during the second iteration, ...

Repeat Block Directives (cont’d)

- IRPC example

  * To generate a sequence of DB statements in the order a, A, e, E, …

    \[
    \text{defineDB MACRO value} \\
    \text{DB value} \\
    \text{ENDM} \\
    \text{IRPC char, aeiou} \\
    \text{defineDB '&char'} \\
    \text{defineDB %'&char'-32} \\
    \text{ENDM}
    \]

  * Can also use

    \[
    \text{IRP char, <a,e,i,o,u>} \\
    \text{in place of IRPC statement}
    \]
Conditional Assembly

• Several conditional directives are available

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF/IFE</td>
<td>Assembles if condition is true (IF) or false (IFE)</td>
</tr>
<tr>
<td>IFDEF/IFNDEF</td>
<td>Assembles if symbol is defined (IFDEF) or undefined (IFNDEF)</td>
</tr>
<tr>
<td>IFB/IFNB</td>
<td>Assembles if arguments are blank (IFB) or not blank (IFNB)</td>
</tr>
<tr>
<td>IFIDN/IFDIF</td>
<td>Assembles if arguments are same (IFIDN) or different (IFDIF) - case sensitive</td>
</tr>
<tr>
<td>IFIDNI/IFDIFI</td>
<td>Assembles if arguments are same (IFIDNI) or different (IFDIFI) - case insensitive</td>
</tr>
</tbody>
</table>

Nested Macros

• Macros can be nested

```
shifty MACRO  lr,operand,count
   IF PROC_TYPE EQ 8086
      IF count LE 4
         REPT count
            sh&lr operand,1
         ENDM
      ELSE
         mov CL,count
         sh&lr operand,CL
      ENDIF
   ELSE
      sh&lr operand,count
  ENDIF
ENDM
```
Nested Macros (cont’d)

```assembly
shift MACRO operand,count
    ;; positive count => left shift
    ;; negative count => right shift
    IFE count EQ 0
        IF count GT 0     ;; left shift
            shifty l,operand,count
        ELSE              ;; right shift
            shifty r,operand,-count
        ENDIF
    ENDIF
ENDM
```

Performance: Macros vs Procedures

![Graph showing performance comparison between macros and procedures](image-url)

1998 © S. Dandamudi Macros: Page 35

To be used with S. Dandamudi, “Introduction to Assembly Language Programming,” Springer-Verlag, 1998.
Performance: Macros vs Procedures (cont’d)

![Graph showing performance comparison between macros and procedures](image)

- **Y-axis**: Sort time (seconds)
- **X-axis**: Number of elements

The graph illustrates the sort time in seconds for both macros and procedures as the number of elements increases. The trend shows that procedures generally outperform macros in terms of efficiency, especially as the number of elements grows.