Macro Expander

μSPEC 5

JUNE 1975

NATIONAL μSPEC 5

MACRO EXPANDER

BENEFITS – increases programmer productivity:

- Provides a convenient "shorthand," which enables faster assembly language coding.
- Reduces programmer errors because the *macro* can be checked out once and used successfully many times.
- Enhances the readability of the source code, thus lessening the work of program documentation.

FEATURES

- LOCAL SYMBOLS – variables can be declared as local to a *macro* instruction to prevent conflicts of names within a program.
- PARAMETER DRIVEN – optional parameters allow a *macro* to be defined with variable values that can be replaced when the *macro* is called or expanded.
- CONDITIONAL EXPANSION – the lines of code generated by a single *macro* can be varied by simply changing the parameter values when the *macro* is called.
- NESTED MACRO CALLS – a *macro* instruction can call other *macros* or even call itself.

INTRODUCTION

Programming in assembly language allows the programmer to produce highly efficient, machine-language code, making the best use of the microprocessor resources. However, programming in assembly language can be very time consuming.

The Macro Expander provides the programmer with many of the capabilities found in higher-level languages while maintaining the efficiency and control of assembly language code. Using the Macro Expander, the programmer can define his own *macro* instructions and then use them throughout his assembly language source program. The Macro Expander then generates the expanded assembly language source statements for each *macro* and produces an expanded, complete, and annotated source program.

The Macro Expander is available through timesharing service bureaus and is compatible with any of National Semiconductor's IMP series assembly languages. The following paragraphs illustrate the flexibility and power of the Macro Expander as utilized to speed and simplify assembly language programming and, thus, to reduce development time and costs.

B15M65/Printed in U.S.A.

CAPABILITIES

Input to the Macro Expander is an assembly language source program with interspersed *macro* directives and calls. Output is an assembly language source program with the *macros* expanded and the *macro* directives and calls echoed as comments. All *macro* directives and calls are identified by closing parenthesis ")" as the first nonblank character in a line.

Defining a Macro

The following form is used to define a *macro:*

```
)MACRO   macname, param1, param2, . . . param32
                 .
                 .
              macro body
                 .
                 .
)ENDM
```

where:

- *)MACRO* is the directive that causes the Macro Expander to accept the ensuing information as a *macro* definition.
- *macname* is the name assigned by the user to the *macro* being defined, and this name is used later to 'call' the *macro.*
- *param1 - param32* are optional parameters that are used to further define elements within the *macro.*
- *macro body* consists of assembly language statements that define the function of the *macro.*
- *)ENDM* designates the end of the *macro.*

Example #1:
The following is a simple *macro* that uses no parameters. It loads R0, R1, and R2 with constants and calls a subroutine (ADELAY) which then generates a delay based on the values of AC0, AC1, and AC2.

```
)MACRO    DELAY
LI        0, 1
LI        1, 4
LI        2, 1
JSR       ADELAY
)ENDM
```

Calling a Macro

Once a *macro* has been defined, it may then be 'called'. Calling consists of telling the Macro Expander to generate and insert the previously defined code into the user's source program. The following form is used for a *macro* call:

```
)macname     param1, param2, . . . param32
```

where:

- *macname* is the name previously assigned in the *macro* definition.
- *param1 - param32* are values that are to be assigned to the parameters declared in the *macro* definition.

Example #2:
The *macro* defined in Example #1 is called by)DELAY.

Use of the)DELAY *macro* results in the following code being generated and inserted:

```
;          )DELAY
           LI          0, 1
           LI          1, 4
           LI          2, 1
           JSR         ADELAY
```

Using Parameters

The power of a *macro* can be increased tremendously through the use of the optional parameters. The parameters allow variable values to be declared when the *macro* is first defined. The variable values are then replaced with constant values when the *macro* is called.

Example #3:
The following *macro* performs the same function as the *macro* defined in Example #1, but instead of providing constants to be loaded into the three registers, three variables (D1, D2, D3) are designated.

```
)MACRO      DELAY2, D1, D2, D3
LI          0, D1
LI          1, D2
LI          2, D3
JSR         ADELAY
)ENDM
```

Now, a variable delay can be generated by varying the values of D1, D2, and D3 when the *macro* is called. The following call to DELAY2 generates the same delay (the same code) as Example #2.

```
)DELAY2      1, 4, 1
```

Local Variables

The Macro Expander allows variables to be designated as 'local' to a particular *macro*; so each time that a *macro* is used (that is, called and expanded), selected symbols may be assigned unique, nonconflicting meanings. This permits the programmer to use *macros* containing labels more than once without causing a name to be multiple-defined.

Local variables are defined by inserting the)LOCAL directive immediately following the)MACRO directive in a *macro* definition as shown below.

```
        )MACRO      macname
        )LOCAL      varname
                       .
                       .
                    macro body
                       .
        )ENDM
```

The usefulness of this capability is illustrated in the following examples.

Example #4:
The following *macro* generates a jump to parameter A if R0 < 0, or to parameter B if R0 > 0. If R0 = 0, control passes to the next statement.

```
        )MACRO      BPN, A, B
        BOC         ZR, LABEL
        BOC         SIGNBIT, A
        JMP         B
LABEL:
        )ENDM
```

If this *macro* is called more than once, it generates duplicate labels each time it is called. (LABEL is defined with each call.)

Example #5:
The problem of duplicate labels may be solved by making LABEL a local variable. This is accomplished by inserting the statement

```
        )LOCAL      LABEL
```

after the *macro* definition directive in Example #4.

Now, each call to the *macro* BPN generates a unique label for each LABEL in the form MGLXXX, where XXX is a unique 3-digit decimal number assigned by the Macro Expander (MGL = Macro Generated Label). The following macro calls cause code to be generated:

```
        )BPN        X, Y
        )BPN        M, N
```

The)BPN X, Y call generates the following code:

```
;           )BPN        X, Y
            BOC         ZR, MGL001
            BOC         SIGNBIT, X
            JMP         Y
MGL001:
```

The)BPN M, N call generates the following code:

```
;           )BPN        M, N
            BOC         ZR, MGL002
            BOC         SIGNBIT, M
            JMP         N
MGL002:
```

Conditional Expansion

The versatility and power of the Macro Expander are further enhanced by a *Conditional Expansion Capability*. This feature allows the programmer to generate easily different lines of code from the same *macro* by simply varying the parameter values used in the *macro* calls. Two relational operators are provided, EQ (equal) and NE (not equal); these operators cause comparisons to be made between input parameters when a *macro* is called. The following directives control conditional expansion:

```
        )IF
        )ELSE
        )ENDIF
```

When the Macro Expander encounters the)IF directive, it evaluates the relational operation that follows. If the relational operation (EQ or NE) is satisfied, the lines following the)IF are expanded until the)ELSE is encountered and the subsequent lines are ignored. If the relational operation is not satisfied, only the lines from)ELSE to)ENDIF are expanded.

Example #6:
The following *macro* generates a BOC instruction. The parameter COND is the condition to be tested, REG is the register to be tested, and DEST is the label to branch to if the condition is satisfied. If REG = 0, a simple BOC is generated. If REG ≠ 0, a series of register exchanges is required to preserve register contents.

```
            )MACRO      BOC, COND, REG,
                          DEST
            )LOCAL      LABEL1, LABEL2
            )IF         EQ, REG, 0
            BOC         COND, DEST
            )ELSE
            RXCH        0, REG
            BOC         COND, LABEL1
            RXCH        0, REG
            JMP         LABEL2
LABEL1:     RXCH        0, REG
            JMP         DEST
LABEL2:
            )ENDIF
            )ENDM
```

Now, the macro call ")BOC ZR, 0, LABEL" causes the lines between)IF and)ELSE to be expanded, since the parameter provided for REG equals zero. If the macro is called as follows

```
                    )BOC           ZR, 1, LABEL
```

the lines between)ELSE and)ENDIF are expanded since REG ≠ 0.

Nested Macro Calls

The Macro Expander allows nested *macro* calls, so a *macro* definition may contain a call to other *macros* or even to itself. This capability may be used to even further reduce programming time because it provides a power similar to that of subroutines. A single *macro* definition can be called by several others, and it will be expanded and its code inserted within the calling *macros*. Up to 64 levels of macro nesting are allowed.

Example #7:
The following *macros* are written for the IMP-4 and illustrate the value of the nesting capability.

The first *macro* is named LDREG and loads registers 1, 2, and 3 with the left, middle, and lower 4-bit bytes of an address.

```
)MACRO     LDREG, ADDRESS
LI         1, < ADDRESS
LI         2, #ADDRESS
LI         3, > ADDRESS
)ENDM
```

The next *macro* is named WRITE and uses the LDREG *macro* to load the registers with the address of a message; WRITE then causes a jump to an output routine to print the message.

```
)MACRO     WRITE, MESSAGE
)LDREG     MESSAGE
JSR        PRINT
)ENDM
```

Now, a simple call to the WRITE *macro*

```
)WRITE     ENDMSG
```

generates the following code:

```
;          )WRITE     ENDMSG
           LI         1, < ENDMSG
           LI         2, #ENDMSG
           LI         3, > ENDMSG
           JMP        PRINT
```

Manufactured under one or more of the following U.S. patents: 3083262, 3189758, 3231797, 3303356, 3317671, 3323071, 3381071, 3408542, 3421025, 3426423, 3440498, 3518750, 3519897, 3557431, 3560765, 3566218, 3571630, 3575609, 3579059, 3593069, 3597640, 3607469, 3617859, 3631312, 3633052, 3638131, 3648071, 3651565, 3693248.

National Semiconductor Corporation
2900 Semiconductor Drive, Santa Clara, California 95051, (408) 732-5000/TWX (910) 339-9240

National Semiconductor GmbH
808 Fuerstenfeldbruck, Industriestrasse 10, West Germany, Tele. (08141) 1371/Telex 27649

National Semiconductor (UK) Ltd
Larkfield Industrial Estate, Greenock, Scotland, Tele. (0475) 33251/Telex 778-632

NATIONAL