Design Aspects and Trends among the Recent Maco-Processors as Compared to Function and Subroutines

Dharmendra Kumar
M. Tech. Student
Rajiv Gandhi Proudyogiki Vishwavidyalaya, Bhopal, India

Abstract— In this research paper we discuss those macro-processors that do not dependent on (tied to, or integrated with) any particular programming language or piece of software, but can be used with a variety of different languages. These are the general purpose macro-processors. The different factors considered while designing a general purpose macro-processors are: comments, grouping of statements, tokens and syntax used for macro definitions. A common case of preprocessing is the processing, performed on the code before the next step of compilation. In that sense, Lexical level of preprocessors is the lowest level of preprocessors in so far as they only require Lexical analysis.

Key words: Preprocessor, Maco-Processors

I. INTRODUCTION

In some programming languages like C/C++, there is a prior phase of translation known as preprocessing and the program for this purpose is called a Preprocessor or Macro-processor. This output is often used by some subsequent programs like a language translator (compiler, assembler).

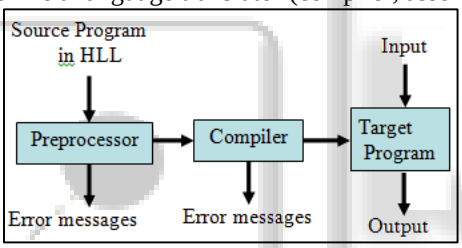

Fig. 1: Preprocessor

When the preprocessing or macro processing facility is provided to the assemblers, it is done by a program mainly known as the Macro assemblers.

A preprocessor is a program and the macro or preprocessor directive is the command to the preprocessor. The preprocessor looks for the macros. A macro is basically the programmer defined symbol to define pseudo operations or an abbreviation to provide a notational convenience by which one can develop a compact version of a program (module programming). Once a macro is defined using the appropriate pseudo-operation, its name may be used in place of an op-code. Macros can be used in many programming languages like C, C++, assembly etc.

II. MACRO EXPANSION

A macro is a unit of specification for program generation through expansion. A macro invocation statement is expanded into the statements that form the body of the macro, with the arguments from the macro invocation statement substituted for the parameters in the macro prototype. For example, in C language or assembly language, a name defines a set of statements which is substituted for macro name whenever the name appears in a program when the program is translated.

Macro expansion process basically consists of macro substitution and language expansion. In its simplest form, a macro-processor is a program that copies a stream of text from one place to another, making some kind of systematic set of replacements as it does, e.g. defining new constructs that can be expressed in terms of existing components of the same language. Macro-processor replaces each occurrence of macro template with the corresponding macro body (statements). After the macro expansion, the macro definition statements are deleted since they are no longer needed.

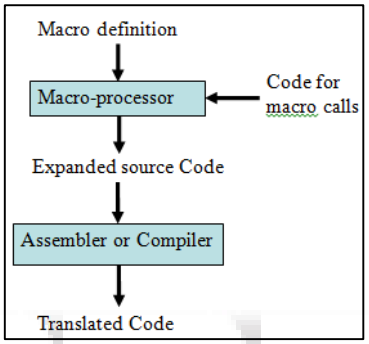

Fig. 2: Macro expansion on a source program.

After expansion the result is a new temporary file called as the expanded source code that we normally can't see, but that we can instruct the translator to save and is dumped on a disk file so that we can examine it, if we want.

Normally the macro performs no analysis of the text, it handles and does not concern with the meaning of the involved statements during macro expansion. The amount and kind of processing done depends on the nature of the preprocessor. Some preprocessors are only capable of performing relatively simple textual substitutions and macro expansions, while others have the power of full-fledged programming languages.

Note: There is a problem regarding the expansion of the labels in the body of macro in that, if the same macro is expanded multiple times at different places in the program, there will be duplicate labels, which will be treated as errors by the translator. The solutions to this problem are:

- Do not use labels in the body of macro
- Explicitly use PC-relative addressing instead

III. MACRO FUNCTION: MACROS WITH ARGUMENTS

A macro definition in a program defines either a new operation or a new method of declaring data. A macro call is an invocation of the new operation or the new method of declaring data defined in the macro. A macro function consists of the three parts: macro name, set of formal parameters and the body of the Macro as dictated following:

A. Macro Name

It is the name provided to a particular macro. It leads to the program generation activity during which the macro call

(macro with parameters) is substituted into the macro body (consisting a sequence of statements i.e. a large number of instructions or data definition) during expansion in the program. The macro name with set of actual parameters is replaced by some code generated by the body of the Macro Function; e.g. a C macro.

B. *Arguments (Parameters)*

Macros can have arguments, just as functions can, and Macros with arguments are used to replace small functions. Macro parameters support code reuse and, one macro definition to implement multiple algorithms. Parameter names are local symbols, which are known within the macro only, outside the macro, they have no meaning. The parameters are associated with one another according to their positions. The first parameter in the macro matches with the first one in the macro prototype and so on.

C. *Macro Body*

Macro body consists of the group of statements that will be replaced as the expansion of the macro. A Macro body is a commonly used group of statements or some sequence of related code (a sequence of instructions or the data storage pseudo-ops) in the source program.

D. *Macro Call*

A macro invocation is referred as a Macro call. After the macro call, the *Macro Invocation* statement is considered as comments and the statement generated from expansion is treated exactly as though they had been written directly by the programmer.

IV. GENERAL-PURPOSE MACRO-PROCESSORS

These are the macro-processors that do not depend on (tied to, or integrated with) any particular programming language or piece of software, but can be used with a variety of different languages. The different factors considered while designing a general purpose macro-processors are: comments, grouping of statements, tokens and syntax used for macro definitions.

A. *Pros*

- Programmers do not need to learn about a macro facility for many languages or language translators.
- Overall saving in software development cost and maintenance cost.

Note: Although its development costs are somewhat greater than those for a language specific macro-processor initially. But, those expenses need not to be repeated for each language later on, thus save substantial overall cost.

B. *Cons*

- Large amount of details from the various languages must be dealt with in a real general purpose macro-processor.
- Situations where a normal macro parameter substitution does not occur, e.g., comments.
- Facilities for grouping together different terms, expressions, statements, tokens, (e.g., identifiers, constants, operators, keywords, puntuations)
- Syntax had better be consistent with the source programming language.

1) Syntax for defining the Macro

The syntax for the general purpose macro function is given as:

```
MACRO <Macro-name>  <formal parameters>
    :
<Macro body>
    :
ENDM
```

Here two new assembler directives are used in macro definition:

a) Macro

Identify the beginning of a macro definition

ENDM: identify the end of a macro definition

The formal parameters can be any in number. When we call a macro, the format will be:

<Macro-name> <Actual parameters>

The tasks involved in macro expansion are:

- Identify the macro calls in the program
- Identify the values of formal parameters (if they are there)
- Maintain the values of expansion time variables declared in a macro
- Organize the expansion time control flow
- Determine the values of sequencing symbols
- Perform the expansion of a model statement

2) Valid Macro Arguments

The following types of parameters/values are the valid arguments for the general purpose Macros:

- Single printable character, preceded by "!" as an escape character
- Quoted strings (in single or double quotes)
- String consisting of printable characters, not containing blanks, tabs, commas or semicolons
- Character sequences, enclosed in literal brackets <…>, which may be arbitrary sequences of valid macro arguments as blanks, commas and semicolons
- Expressions preceded by a % character (e.g. in NASM)

a) Note

- The syntax for the macro for the SIC or SIC/XE machine involves the keyword MACRO after the name of the macro.
- NASM assembler uses the symbol % in the macro definition. Also it uses the % symbol while using the name of the parameters in the body of the macro. With the call we need to precede the argument number by the %symbol.

In the General-Purpose macro-processors, we use the macro arguments preceded by the symbol '&' in the macro definition and its body. But when we call the macro, this '&' symbol is not used.

b) Example 1

Consider the following macro definition:

```
MACRO M1 &D1, &D2
STA &D1
STB &D2
ENDM
```

Here, M1 is a macro name with two parameters D1 and D2. This macro M1 stores the contents of register A in

D1 and the contents of register B in D2. Now consider the following statements,

M1 x, y
M1 p, q

The first statement is a macro invocation statements that gives the name of the macro instruction being invoked and the arguments x and y to be used in expanding. Here M1 is invoked with the parameters x and y and second time with p and q. The code below shows the expanded macro for the above program, replacing the macro call with its block of executable instruction. The above first and second calls are replaced as:

STA x
STB y

And

STA p
STB q

c) Example 2

Consider the following Macro named DG:

MACRO DG &Ct, &Reg
MOV A, &Ct
SUB A, &Reg
ENDM

When we call the macro as:

DG 15, R7

The arguments for this macro code are replaced as under:

MOV A, 15
SUB A, R7

After the above replacement, the program is assembled.

C. Macro Processing within Language Translators

We may also combine the macro processing functions with the language translator. The macro-processors process the macro definitions, expand the macro invocations and produce an expanded version of the source program, which is then can be used as input to a translator (assembler or compiler). Macro processing within translators can be performed in the following two ways:

- Line-by-line macro-processor
- Integrated macro-processor

1) Line-by-Line Macro-processor

It is used as a sort of input routine for the translator as it reads the source program and pass the output lines to the translator. It processes the macro definitions and expands macro invocations. For example the C language preprocessor works on this rule.

a) Benefits

- Avoid making an extra pass over the source program during translation.
- Some of the utility can be used and shared by both the macro-processor and the language translators so that utility can be loaded once.
  - Data structures required can be combined (e.g., OPTAB and MNT)
  - Utility subroutines for:
    1) Scanning input lines
    2) Searching tables
    3) Data format conversion
- Easier to give diagnostic messages related to the source statements

2) Integrated Macro-processor

It can potentially make use of any information about the source program that is extracted by the translator. It can support macro instructions that depend upon the context in which they occur.

D. Default Parameters to the Macros

We can also have the default parameters to the macros. The number of actual arguments passed to a macro can be less (but not greater) than the number of its formal parameters. If an argument is omitted, the corresponding formal parameter is replaced by an empty string. If arguments other than the last ones are to be omitted, they can be represented by the commas.

1) Example 3

```
MACRO DG &A1, &A2, &A3, &A4, &A5, &A6, &A7, &A8
...
<macro body>
...
ENDM
```

Let the call is made as follows:

DG 31, 12, , ,23, 7

The formal parameters A1, A2, A5 and A6 are replaced by the arguments 31, 12, 23 and 7 during substitution. The parameters A3, A4, A7 and A8 are replaced by a zero length string.

V. NESTED MACRO

Though not supported fully by NASM, a powerful feature of many macro-processors including MASM is to handle nested macros. Nesting can come in two different ways as described following:

A. Macro Definition Nesting

In this process, the new macros may be defined within the body of the another macro. Consider the following example for this purpose:

1) Example 4

```
MACRO DG  &D1, &D2
MOV R1, &D1
MOV R2, &D2
ADD R1, R2
MACRO SM &D
MOV R3, &D
INC R3
ENDM
MOV D2, R1
ENDM
```

Here the macro SM is encapsulated within macro DG and is not directly visible to the code outside the DG. However, once DG is called, SM also becomes visible. From that point onwards, SM can also be called directly without calling DG anymore. Thus the calling sequence,

```
DG x, y.
.
.
SM z
```

is a valid one. But the sequence,

```
SM z
.
.
```


DG x, y
will produce an error as SM at the first place is undefined.

B. Macro call Nesting

The statement, in which a macro calls on another macro, is called nested macro call. In it, the call is done by outer macro and the macro called is the inner macro. Macro calls may be nested in the sense that from within the body of one macro, there can be a call to a previously defined macro. For example in the following, we have first defined the macro SM and then in the body of DG, a call has been made to macro SM.

1) Example 5

```
MACRO SM &S
MOV R3, &S
INC R3
ENDM
MACRO  DG  &D1, &D2
MOV R1, &D1
MOV R2, &D2
ADD R1, R2
SM B
MOV D2, R1
ENDM
```

VI. Comparison between functions and macros

Macros and functions can be compared based on the many standard criteria as described below:

A function call is performed by transferring a control at the place in the memory where the function code is stored. From this place the function call is made.

On the other hand, a macro call is replaced by the actual commands (or code or body) they represent. Macro body is copied into the program at the place of its (macro) calling.

In particular there is a special kind of function called inline function which performs approximately similar to the macros.

VII. Conclusion

Macros play a vital role in completion of a program. Back in the days the macros had many limitations but over the years it has been improvised and worked upon and now it's able to handle a lot more complex functions like adding support for modern features including dynamically linked shared libraries and the unusual demands of C++. The design of macro-processor generally doesn't depend lot on the architecture of the machine. Macros has many features that are better in performance as compared to the functions. In the case of a macro, a smaller code fragment may better be implemented, since substitution of it will not increase the code size significantly, at the same time, saving in context switching will result in faster execution of the program. Hence macros are very important for a programmer.

References

[1] Brown, P. J., A Survey of Macro Processors, Ann. Rev. in Aut. Prog. Vol. 6. Oxford & New York, NY.: Pergamon Press, 1966, pp. 37–88.
[2] Campbell-Kelly, M., An Introduction to Macros, New York, NY.: American Elsevier, 1973.
[3] Cole, A. J., Macro Processors, Cambridge: Cambridge University Press, 1976.
[4] McIlroy, M. D., Macro Instruction Extensions of Compiler Languages, in Comm. ACM 3,(4), p. 214 (1960).
[5] Graham, M. L., P. Z. Ingerman, An Assembly Language for Reprogramming, Comm. ACM 8,(12) p. 769, (1965).
[6] Ferguson, D. E., The Evolution of the Meta-Assembly Program, Comm. ACM 9, p. 190 (1966).
[7] Freeman, D.N., Macro Language Design for System/360, IBM Sys. J. 5, (1966)6–77.
[8] IBM System/360 Operating System Assembler Language, IBM Form No. GC28-6514.
[9] Wegner, P., Programming Languages, Information Structures, and Machine Organization. New York, NY.: McGraw-Hill 1968.
[10] PDP-11 Macro-11 Language Reference Manual, Order #AA-5075B-TC.
[11] IBM PC Macro Assembler, IBM #6172234.
[12] Greenwald, I. D., Handling of Macro Instructions, Comm. ACM 2,11,21–23(1959).
[13] Donovan, J., Systems Programming, New Yor,k NY.: McGraw-Hill, 1972.
[14] Revesz, G., A Note on Macro Generation, Soft. Pract. & Exp. 15(5),423–426(May 1985).
[15] Beck, L. L., System Software, Reading, Mass.: Addison-Wesley,
[16] Knuth, D. E., The Art of Computer Programming, vol. I, Reading, Mass.: Addison-Wesley, 1973.
[17] VAX Macro & Instruction Set Reference Manual. Order #AA-Z700A-TE, Digital Equipment Corp., Maynard, Mass., 1984.
[18] Microsoft Macro Assembler 5.1 Programmer's Guide, Microsoft Corp., 1987.

