BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 4

DCA2203

SYSTEM SOFTWARE

Unit 4

Macros and Macro Processor

Table of Contents

1. INTRODUCTION

We have studied various language processors in the previous units. In this unit discusses about macros and macro processors in detail. Before discussing macros and macro processors, some introduction about preprocessors and subroutines are given, which will help in understanding the concepts of macros and macro processors.

In this unit, we are discussing Macro definition and call, Macro expansion, and Nested macro calls. We are also discussing advanced macro facilities. In the last section of this unit, we are discussing the design of a Macro preprocessor.

1.1 Learning Objectives

After studying this unit, you should be able to:

- ❖ *Define a macro and macro processor*
- ❖ *Describe macro definition and call*
- ❖ *Explain macro expansion*
- ❖ *Describe the advanced macro facility*
- ❖ *Explain the design of the macro processor*

2. MACRO DEFINITION AND CALL

In computer science, a preprocessor is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers. The amount and kind of processing done depend 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.

A common example from computer programming is the processing performed on source code before the next step of compilation. In some computer languages, there is a phase of translation known as preprocessing.

A *general purpose macro processor* is a macro processor that is not tied to, or integrated with, a particular language or piece of software.

In its simplest form, a macro processor is a preprocessor, which handles the macros. Whenever there is a macro call the macro processor substitutes the definition in the place of the macro call. This is macro expansion. Macros are used to provide a program generation facility through macro expansion.

Definition: A Macro is a unit of specification of program generation through expansion.

A *macro consists* of a name, a set of formal parameters, and a body of code. A macro name is an abbreviation, which stands for some related lines of code. Macros are useful for the following purposes:

- To simplify and reduce the amount of repetitive coding
- To reduce errors caused by repetitive coding
- To make an assembly program more readable.

The use of a macro name with a set of actual parameters is replaced by some code generated by the program body. This is called macro expansion.

Macros allow a programmer to define pseudo operations, typically operations that are generally desirable, are not implemented as part of the processor instruction, and can be implemented as a sequence of instructions. Each use of a macro generates new program instructions; the macro has the effect of automating the writing of the program.

A *macro definition* is enclosed between a *macro header* statement and a *macro* end statement. Macro definitions are typically located at the start of a program. A macro definition consists of

1. A *macro prototype* statement
2. One or more model statements
3. *Macro preprocessor* statements.

The macro prototype statement declares the name of a macro and the names and kinds of its parameters. A model statement is a statement from which an assembly language statement may be generated during macro expansion. A preprocessor statement is used to perform auxiliary functions during macro expansion.

The macro prototype statement has the following syntax:

```
Macro_name<formal parameters spec>
```

Where <macro name> appears in the mnemonic field of an assembly statement and <formal parameter spec> is of the form

```
&<parameter name> [<parameter kind>]
```

Macros are similar to functions in that they can take arguments and in that, they are called to lengthier sets of instructions. Unlike functions, macros are replaced by the actual commands they represent when the program is prepared for execution. Function instructions are copied into a program only once.

Macro Call

A macro is called by writing the macro name in the mnemonic field of an assembly statement. The macro call has the syntax

<macro name> [<actual parameter spec>[,...]]

where an actual parameter typically resembles an operand specification in an assembly language statement.

Consider the example,

Example 4.1: Following figure 4.1 shows the definition of macro INCR. MACRO and MEND are the macro header and macro end statement, respectively. The prototype statement indicates that three parameters called MEM_VAL, INCR_VAL, and REG exist for the macro. Since parameter kind is not specified for any of the parameters. They are all of the default kind 'positional parameter'. Statements with the operation codes MOVER, ADD and MOVEM is model statements. No preprocessor statements are used in this macro. Figure 4.1 shows a macro definition.

```
        MACRO
        INCR        &MEM_VAL, &INCR_VAL, &REG
        MOVER       &REG, &MEM_VAL
        ADD         &REG, &INCR_VAL
        MOVEM       &REG, &MEM_VAL
        MEND
```

Fig 1: A macro definition.

Self-Assessment Questions - 1

1. A unit of specification of program generation through expansion is called __________.
2. The use of macro name with set of actual parameters is replaced by some code generated by its body is called __________.
3. A *macro definition* is enclosed between a __________ statement and a ______ statement.

3. MACRO EXPANSION

The use of a macro name with a set of actual parameters is replaced by some code generated from its body. This is called Macro expansion. Two kinds of expansion are:

1. **Lexical Expansion**

   Lexical expansion implies the replacement of a character string with another character string during program generation. Lexical expansion is typically employed to replace occurrences of formal parameters with corresponding actual parameters.

2. **Semantic Expansion**

   Semantic expansion implies the generation of instructions tailored to the requirements of a specific usage.- for example, the generation of type-specific instructions for the manipulation of byte and word operands. Semantic expansion is characterized by the fact that different uses of a macro can lead to codes that differ in the number, sequence, and opcodes of instructions.

A macro call leads to macro expansion. During macro expansion, the macro call statement is replaced by a sequence of assembly statements. To differentiate between the original statements of a program and the statements resulting from macro expansion, each expanded statement is marked with a '+' preceding its label field.

Two key notions concerning macro expansion are:

1. Expansion time control flow: This determines the order in which model statements are visited during macro expansion.
2. Lexical substitution: Lexical substitution is used to generate an assembly statement from a model statement.

Flow of control during expansion:

The default flow of control during macro expansion is sequential. Thus, in the absence of preprocessor statements, the model statements of a macro are visited sequentially starting with the statement following the macro prototype statement and ending with the statement preceding the MEND statement. A preprocessor statement can alter the flow of control during expansion such that some model statements are either never visited during expansion, or are repeatedly visited during expansion. The former results in *conditional expansion* and the latter in *expansion time loops*. The flow of control during macro expansion is implemented using a *macro expansion counter* (MEC).

Algorithm for Outline of Macro expansion is as follows:

1. MEC:= statement number of the first statement following the prototype statement;

```
2. While the statement pointed out by MEC is not a MEND statement
     a) If a model statement then
          (i) Expand the statement.
          (ii) MEC:= MEC+1;
     a) Else (i.e. a preprocessor statement)
          (i) MEC:=new value specified in the statement;
3. Exit from macro expansion.
```

MEC is set to point at the statement following the prototype statement. It is incremented by 1 after expanding a model statement. Execution of a preprocessor statement can set MEC to a new value to implement conditional expansion or expansion time loops.

Lexical Substitution

A model statement consists of 3 types of strings

1) An ordinary string, which stands for itself.
2) The name of a formal parameter is preceded by the character '&'.
3) The name of a preprocessor variable, which is also preceded by the character '&'.

During lexical expansion, Strings of type 1 are retained without substitution. Strings of types 2 and 3 are replaced by the 'values' of the formal parameters or preprocessor variables. The value of a formal parameter is the corresponding actual parameter string. The rules for determining the value of a formal parameter depend on the kind of parameter.

Positional Parameters

A positional formal parameter is written as &<parameter name>, e.g. &SAMPLE where SAMPLE is the name of a parameter. In other words, <parameter kind> of formal parameter specification is omitted. The actual parameter specification in a call on a macro using positional parameters is simply an <ordinary string>.

The value of a positional formal parameter XYZ is determined by the rule of positional association as follows:

1. Find the ordinal position of XYZ in the list of formal parameters in the macro proto-type statement.
2. Find the actual parameter specification occupying the same ordinal position in the list of actual parameters in the macro call statement.

Example 4.2: Consider a call

```
INCR    A, B, AREG
```

on macro INCR, Following the rule of positional association, values of the formal parameters are:

Formal Parameter	**value**
MEM_VAL	A
INCR_VAL	B
REG	AREG

Lexical expansion of the model statements now leads to the code

```
+     MOVER     AREG, A
+     ADD       AREG, B
+     MOVEM     AREG, A
```

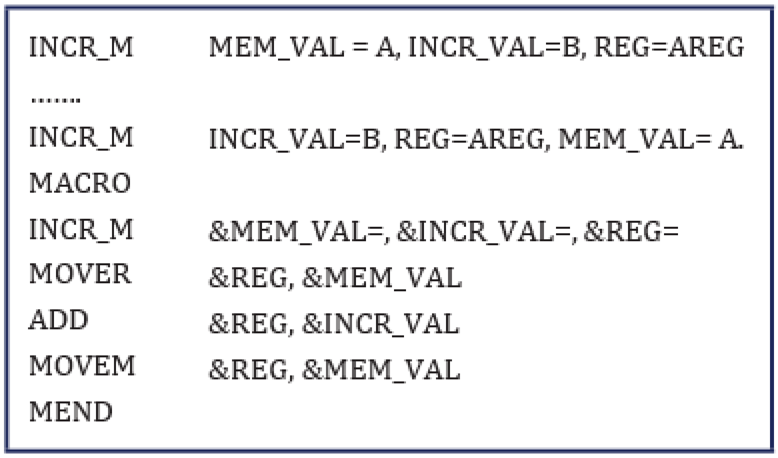
Keyword Parameters

For keyword parameters, <parameter name> is an ordinary string, and <parameter kind> is the string '='. The <actual parameter spec> is written as <formal parameter name>=<ordinary string>. The value of a formal parameter XYZ is determined by the rule of keyword association as follows:

1. Find the actual parameter specification which has the form XYZ = <ordinary string>.
2. Let <ordinary string> in the specification be the string ABC. Then the value of formal parameter XYZ is ABC.

Note that the ordinal position of the specification XYZ= ABC in the list of actual parameters is immaterial. This is very useful in situations where long lists of parameters have to be used.

Example 4.3: Figure 4.2 shows the Macro INCR of Figure 4.1. rewritten as Macro INCR_M using keyword parameters. The following macro calls are now equivalent.



```
INCR_M      MEM_VAL = A, INCR_VAL=B, REG=AREG
…….
INCR_M      INCR_VAL=B, REG=AREG, MEM_VAL= A.
MACRO
INCR_M      &MEM_VAL=, &INCR_VAL=, &REG=
MOVER       &REG, &MEM_VAL
ADD         &REG, &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
```

Fig 2: A macro definition using keyword parameters.

Default Specification of parameters

A default is a standard assumption in the absence of an explicit specification by the programmer. Default specification of parameters is useful in situations where a parameter has the same value in most calls. When the desired value is different from the default value, the desired value can be specified explicitly in a macro call. This specification overrides the default value of the parameter for the duration of the call.

Default specification of keyword parameters can be incorporated by extending syntax, the syntax for formal parameter specification, as follows:

&<parameter name>[<parameter kind>[<default value>]]

Example 4.4: Register AREG is used for all arithmetic in a program. Hence most calls on macro INCR_M contain the specification &REG=AREG. The macro can be redefined to use a default specification for the parameter REG shown in Figure 4.3. Consider the following calls.

```
INCR_D              MEM_VAL=A, INCR_VAL=B
INCR_D              INCR_VAL=B, MEM_VAL=A
INCR_D INCR_        VAL=B, MEM_VAL=A, REG=BREG
```

The first two calls are equivalent to the calls in example 4.3. The third call overrides the default value for REG with the value BREG. BREG will be used to perform the arithmetic in its expanded code.

```
        MACRO
        INCR_D          &MEM_VAL=,&INCR_VAL=, &REG=AREG
        MOVER           &REG, &MEM_VAL
        ADD             &REG, &INCR_VAL
        MOVEM           &REG, &MEM_VAL
        MEND
```

Fig 3: A macro definition with default parameter

Macros with mixed parameter lists

A macro may be defined to use both positional and keyword parameters. In such a case, all positional parameters must precede all keyword parameters. For example, in the macro call

```
SUMUP       A, B, G=20, H=X
```

A, B are positional parameters while G, H are keyword parameters. Correspondence between actual and formal parameters is established by applying the rules governing positional and keyword parameters separately.

Other uses of parameters

The model statements of Examples 4.1-4.4 have used formal parameters only in operand fields. However, the use of parameters is not restricted to these fields. Formal parameters can also appear in the label and opcode fields of model statements.

Example 4.5

```
         MACRO
         CALC           &X, &Y, &OP=MULT, &LAB=
&LAB     MOVER          AREG, &X
         &OP            AREG, &Y
        MOVEM           AREG, &X
         MEND
```

Expansion of the call CALC A, B, LAB=LOOP leads to the following code:

```
+ LOOP    MOVER     REG, A
+         MULT      REG, B
+         MOVEM     REG, A
```

4. NESTED MACRO CALLS

A model statement in a macro may constitute a call on another macro. Such calls are known as ***nested macro calls***. Calling macro is called outer macro and called macro is called Inner macro.

Macro bodies may also contain macro calls, and so may the bodies of those called macros and so forth. If a macro call is seen throughout the expansion of a macro, the assembler starts immediately with the expansion of the called macro. For this, its expanded body lines are simply inserted into the expanded macro body of the calling macro, until the called macro is completely expanded. Then the expansion of the calling macro is continued with the body line following the nested macro call.

Expansion of nested macro calls follows the ***last-in-first-out (LIFO)*** rule. Thus, in a structure of nested macro calls, expansion of the latest macro call (i.e. the innermost macro call in the structure) is completed first.

Example 4.6

```
INSIDE MACRO
SUBB A, R3
ENDM
OUTSIDE MACRO
MOV A, #42
INSIDE
MOV R7, A
ENDM
```

In the body of the macro OUTSIDE, the macro INSIDE is called. If OUTSIDE is called, then one gets something like the following expansion:

Line I	**Addr**	**Code**		**Source**
	15+ 1	0000	74 2A	MOV A, #42
	17+ 2	0002	9B	SUBB A, R3
	18+ 1	0003	FF	MOV R7, A

If macros are calling themselves, (nested/recursive macro calls), In this case, there must be some stop criterion, to prevent the macro of calling itself over and over until the assembler is running out of memory. Here again, conditional assembly is the solution.

Nested Macro Definitions

A macro body may also contain further macro definitions. However, these nested macro definitions aren't valid until the enclosing macro has been expanded! That means, the enclosing macro must have been called before the nested macros can be called.

Example 4.7

A macro, which can be used to define macros with arbitrary names, may look as follows:

```
DEFINE MACROMACNAME
MACNAME MACRO
DB 'I am the macro &MACNAME.'
ENDM
ENDM
```

In order not to overload the example with "knowhow", the nested macro only introduces itself kindly with a suitable character string in ROM. The call

```
        DEFINE Macc
```

Would define the macro

```
Macc MACRO
        DB 'I am the macro Macc.'
        ENDM
```

and the call

```
        DEFINE Macc2
```

would define the following macro:

```
Macc2 MACRO
        DB 'I am the macro Macc2.'
        ENDM
```

Self-Assessment Questions - 2

4. A preprocessor statement can alter the flow of control during expansion such that some model statements are never visited during expansion is called_____________.
5. The flow of control during macro expansion is implemented using __________.
6. Expansion of nested macro calls follows _________ rule.

5. ADVANCED MACRO FACILITY

Advanced macro facilities are aimed at supporting semantic expansion. These facilities can be grouped into

1. Facilities for alternation of the flow of control during expansion.
2. Expansion time variables.
3. Attributes of Parameters.

Alternation of the flow of control during expansion

Two features are provided to facilitate alternation of the flow of control during expansion:

1) Expansion time sequence symbols.
2) Expansion time statements AIF, AGO, and ANOP. A sequencing symbol (SS) has the syntax

. <ordinary string >

As **SS** is defined by putting it in the label field of a statement in the macro body. It is used as an operand in an AIF or AGO statement to designate the destination of an expansion time control transfer. It never appears in the expanded form of a model statement.

An AIF statement has the syntax

AIF(<expression>)<sequencing symbol>

Where <expression> is a relational expression involving ordinary strings, formal parameters and their attributes, and expansion time variables. If the relational expression evaluates to true, expansion time control is transferred to the statement containing <sequencing symbol> in its label field. An AGO statement has the syntax

AGO<sequencing symbol>

And unconditionally transfers expansion time control to the statement containing <sequencing symbol> in its label field. An ANOP statement is written as

<sequencing symbol> ANOP

And simply has the effect of defining the sequencing symbol.

Expansion Time Variables

Expansion time variables (EV's) are variables that can only be used during the expansion of macro calls. A local EV is created for use only during a particular macro call. A global EV exists across all macro calls situated in a program and can be used in any macro which has a declaration for it. Local and global EV's are created through declaration statements with the following syntax:

LCL <EV specification>[,<EV specification>....]

GBL <EV specification.[,<EV specification>....]

And <EV specification> has the syntax &<EV name>, where <EV name> is an ordinary string.

Values of EV's can be manipulated through the preprocessor statement SET. A SET statement is written as

<EV specification> SET <SET-expression>

Where <EV specification> appears in the label field and SET in the mnemonic field. A SET statement assigns the value of <SET-expression> to the EV specified in <EV specification>. The value of an EV can be used in any field of a model statement, and in the expression of an AIF statement.

Example 4.8

```
        MACRO
        CONSTANTS
        LCL         &A
&A      SET         1
        DB          &A
&A      SET         &A+1
        DB          &A
        MEND
```

A call on Macro CONSTANTS is expanded as follows: The local EV A is created. The first SET statement assigns the value '1' to it. The first DB statement thus declares a byte constant '1'. The second SET statement assigns the value '2' to A and the second DB statement declares a constant '2'.

Attributes of formal parameters

An attribute is written using the syntax

<attribute name>’<formal parameter spec>

and represents information about the value of the formal parameter, i.e. about the corresponding actual parameter. The type, length, and size attributes have the names T, L, and S.

Example 4.9

```
         MACRO
         DCL_CONST
         &A
         AIF            (L’&A EQ 1)   .NEXT
         …….
   NEXT  …….
         ……..
         MEND
```

Here, expansion time control is transferred to the statement having.NEXT in its label field only if the actual parameter corresponding to the formal parameter A has the length of ‘1’.

5.1 Conditional Expansion

While writing a general purpose macro it is important to ensure the execution efficiency of its generated code. Conditional expansion helps in generating assembly code specifically suited to the parameters in a macro call. This is achieved by ensuring that a model statement is visited only under specific conditions during the expansion of a macro. The AIF and AGO statements are used for this purpose.

Example 4.10: It is required to develop a Macro EVAL such that a call

```
EVAL        A, B, C
```

generates efficient code to evaluate A-B+C in AREG. When the first two parameters of a call are identical, EVAL should generate a single MOVER instruction to load the 3rd parameter into AREG. This is achieved as follows:

```
              MACRO
        EVAL        &X, &Y, &Z
        AIF         (&Y EQ &X) .ONLY
        MOVER       AREG, &X
        SUB         AREG, &Y
        ADD         AREG, &Z
        AGO         .OVER
.ONLY   MOVER       AREG, &Z
.OVER   MEND
```

Since the value of a formal parameter is simply the corresponding actual parameter, the AIF statement effectively compares the names of the first two actual parameters. If the names are the same, expansion time control is transferred to the model statement MOVER AREG, &Z. If not, the MOVE-SUB-ADD sequence is generated and expansion time control is transferred to the statement. OVER MEND which terminates the expansion. Thus efficient code is generated under all conditions.

Expansion time loops

It is often necessary to generate many similar statements during the expansion of a macro. This can be achieved by writing similar model statements in the macro.

Example 4.11

```
MACRO
CLEAR           &A
MOVER           AREG, ='0'
MOVEM           AREG, &A
MOVEM           AREG, &A+1
MOVEM           AREG, &A+2
MEND
```

When called CLEAR B, the MOVER statement puts the value '0' in AREG, while the three MOVEM statements store this value in 3 consecutive bytes with the addresses B, B+1, and B+2.

Alternatively, the same effect can be achieved by writing an expansion time loop that visits a model statement, or a set of model statements, repeatedly during macro expansion. Expansion time loops can be written using expansion time variables (EV's) and expansion time control transfer statements AIF and AGO.

Example 4.12

```
                MACRO
                CLEAR       &X,&N
                LCL         &M
&M              SET         0
                MOVER       AREG, ='0'
.MORE           MOVEM       AREG, &X+&M
&M              SET         &M+1
                AIF    (&M NE N) .MORE
                MEND
```

Consider expansion of the macro call

```
                CLEARB,      3
```

The LCL statement declares M to be a local EV. At the start of the expansion of the call, M is initialized to zero. The expansion of model statement MOVEM AREG, &X+&M thus leads to the generation of the statement MOVEM AREG, B. The value of M is incremented by 1 and the model statement MOVEM... is expanded repeatedly until its value equals the value of N, which is 3 in this case. Thus the macro call leads to the generation of the statements

```
        +     MOVER        AREG, ='0'
        +     MOVEM        AREG, B
        +     MOVEM        AREG, B+1
        +     MOVEM        AREG, B+2
```

Comparison with execution time loops

Most expansion time loops can be replaced by execution time loops. For example, instead of generating many MOVEM statements as in the above example. 4.12 to clear the memory area starting on B, it is possible to write an execution time loop that moves 0 into B, B+1, and B+2. An execution time loop leads to more compact assembly programs. However, such programs would execute slower than programs containing expansion time loops. Thus a macro can be used to trade program size for execution efficiency.

5.2 Other Facilities For Expansion Time Loops

Many assemblers provide other facilities for conditional expansion, an ELSE clause in AIF being an obvious example. The assemblers for M 68000 and Intel 8088 processors provide explicit expansion time looping constructs. We discuss two such facilities here.

The REPT statement

REPT <expression>

<expression> should evaluate to a numerical value during macro expansion. The statements between REPT and an ENDM statement would be processed for expansion <expression> number of times. Example 4.13 illustrates the use of this facility to declare 10 constants with the values 1,2, .. 10.

Example 4.13

```
        MACRO
        CONST 10
        LCL    &M
&M      SET             1
        REPT            10
        DC              '&M'
&M      SETA            &M+1
        ENDM
        MEND
```

The IRP statement

IRP <formal parameter>, <argument-list>

The formal parameter mentioned in the statement takes successive values from the argument list. For each value, the statements between the IRP and ENDM statements are expanded once.

Example 4.14

```
MACRO
CONSTS          &M, &N, &Z
IRP             &Z, &M, 7, &N
DC              '&Z'
ENDM
MEND
```

A macro call CONSTS 4, 10 leads to a declaration of 3 constants with the values 4, 7, and 10.

5.3 Semantic Expansion

Semantic expansion is the generation of instructions tailored to the requirements of specific usage. It can be achieved by a combination of advanced macro facilities like AIF, AGO statements, and expansion time variables. The CLEAR macro of Example. 4.12 is an instance of semantic expansion. Here, the number of MOVEM AREG, ..statements generated by a call on CLEAR is determined by the value of the second parameter of CLEAR. Macro EVAL of exam¬ple 4.10 is another instance of conditional expansion wherein one of two alternative code sequences is generated depending on the peculiarities of actual parameters of a macro call. Example 4.15 illustrates semantic expansion using the type attribute.

Example 4.15

```
MACRO
CREATE_CONST      &X,&Y
AIF               (T'&X EQ B).BYTE
```

```
&Y      DW      25
        AGO     .OVER
.BYTE   ANOP
&Y      DB      25
.OVER   MEND
```

This macro creates a constant ‘25’ with the name given by the 2nd parameter. The type of the constant matches the type of the first parameter.

6. DESIGN OF MACRO PROCESSOR

In this section, the design concepts of macro processors are discussed. The source program is read and is then transformed into another program from which all macro definitions have been removed. And also all macro calls have also been replaced with the corresponding macro bodies. The output of the macro processor will be an assembly language program containing no macros. Like in assemblers, the macro processor can also be designed in two passes. In the first pass, all the macro definitions are saved and expanded. The macro body, that is, the text is processed in the second pass. Figure 4.4 shows A macro processor design.

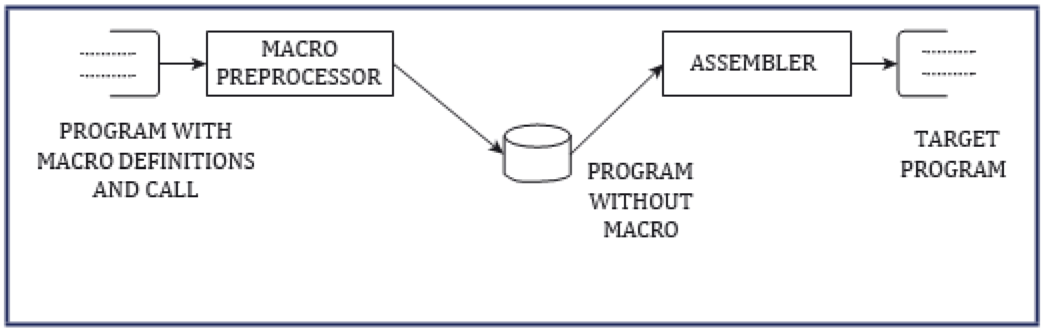


Fig 4: A macro processor design

The macro processor treats it, as though it is the original program. The macro processor is system software. It is actually a program, which is a part of the assembler program. The macro processor is designed in passes. In the design of a macro processor, there are two important functions to be considered. They are given below:

1. Saving the macro definitions
2. Expanding the macro calls.

These functions are done in two passes. Here pass refers to reading the source program once. The functions of each pass are given below:

Functions of Pass 1

1. Reads the source program.

2. If there is a macro call, store the definition of macro in the macro definition Table (MDT).
3. Stores the macro name in the Macro Name Table (MNT).
4. Prepares the List of formal Arguments (AL) in the macro.
5. Repeats the similar processes for all the other macros.

Functions of Pass 2

1. Reads the source program.
2. If there is a macro call, expand the macro with the help of the Macro Name Table.
3. Replaces the formal parameters in the Argument List with the actual parameters.
4. Repeats the similar processes for all the other macros.

These are the functions done by the macro processor in two different passes. The next section discusses the data structures used in the two passes.

6.1 Data Structures Used In Macro Processor Design

There are some important data structures used in the design of a macro processor. They are,

1. Macro Name Table (MNT)
2. Argument List (AL)
3. Macro Definition Table (MDT)

Macro Name Table (MNT)

It is a data structure, which maintains the names and addresses (pointers) where the macro definition is available. This address is helpful to retrieve the macro definition during the macro call.

Argument List (AL)

It also maintains the details of the formal parameters. A list of formal parameters is also maintained. The formal parameters are indicated by some special symbols.

```
MOV EBX,&R

MOV EAX,&S
```

Here ‘R’, ‘S’ are formal parameters, and a special symbol ‘&’ is used to precede these parameters so as to indicate that these are formal parameters.

Macro Definition Table

It is also a data structure that keeps the macro body i.e., text. The macro body can be retrieved and used during the macro expansion by the macro processor. Whenever a macro definition is encountered, the body is read and stored in the macro definition table.

The functions can be rewritten using the above-mentioned data structures as given below:

Functions of Pass 1 of the Macro processor

- Create the Macro Name Table (MNT), Macro Definition Table (MDT) during pass1.
- Create a list of formal parameters (AL) in Macro Name Table.

Functions of Pass 2 of the Macro processor

- During any macro call, reads the macro body from Macro Definition Table (MDT) with the help of the Macro Name Table (MNT)
- Extracts the formal parameters from the macro body and replaces them with the actual parameters provided in the macro call.

Self-Assessment Questions - 3

7. Variables which can only be used during the expansion of macro calls are called ____________ .
8. A model statement is visited only under specific conditions during the expansion of a macro is ensured by__________ and ___________ statements.
9. The generation of instructions tailored to the requirements of a specific usage is called_________ .
10. A data structure, which maintains the names and addresses (pointers) where the macro definition is available is called___________.

7. SUMMARY

Let us recapitulate the important concepts discussed in this unit:

- Macros are used to provide a program generation facility through macro expansion.
- A macro consists of a name, a set of formal parameters, and a body of code.
- Macro definitions are located at the start of a program.
- A macro call leads to macro expansion. During Macro expansion, the Macro call statement is replaced by a sequence of assembly statements.
- A statement in macro results in the call to another macro, then it is called nested macro calls.
- Facilities for alternation of the flow of control during expansion, Expansion time variables, and Attributes of Parameters are the advanced macro facilities.

8. GLOSSARY

Lexical Expansion: Lexical expansion implies the replacement of a character string with another character string during program generation

Macro: A predefined sequence of computer instructions that are inserted into a program, usually during assembly or compilation, at each place that its corresponding macroinstruction appears in the program.

Macro Expansion Counter: Execution of a preprocessor statement can increase the counter by one to implement conditional expansion or expansion time loops.

Macroinstruction: A source code instruction that is replaced by a predefined sequence of source instructions, usually in the same language as the rest of the program and usually during assembly or compilation.

Nested macro: A statement in macro results in the call to another macro

9. TERMINAL QUESTIONS

SHORT ANSWER QUESTIONS

Q1. Explain about Macro definition and call.

Q2. Discuss Macro expansion.

Q3. Write short notes about Nested macro calls.

Q4. Explain in detail about the Advanced Macro facility.

Q5. Explain various Data Structures used in Macro Processor Design.

10. ANSWERS

SELF ASSESSMENT QUESTIONS

1. Macro.
2. Macro expansion.
3. Macro header and Macro end.
4. Conditional expansion.
5. Macro expansion Counter.
6. Last-in-first-out (LIFO).
7. Expansion time variables (EV's).
8. AIF, AGO statements.
9. Semantic Expansion.
10. Macro Name Table.

TERMINAL QUESTIONS

SHORT ANSWER QUESTIONS

Answer 1: A Macro is a unit of specification of program generation through expansion. Refer section 2 for detail.

Answer 2: The use of a macro name with a set of actual parameters is replaced by some code generated from its body. This is called Macro expansion. Refer section 3 for detail.

Answer 3: A model statement in a macro may constitute a call on another macro. Such calls are known as nested macro calls. Refer section 4 for detail.

Answer 4: Advanced macro facilities are aimed at supporting semantic expansion. Refer section 5

Answer 5: There are some important data structures used in the design of a macro processor. They are macro name table(MNT), Argument List(AL) and Macro Definition Table(MDT). Refer subsection 6.1

11. SUGGESTED BOOKS AND E- REFERENCES

- Dhamdhere (2002). Systems programming and operating systems Tata McGraw-Hill.
- M. Joseph (2007). **System software**, Firewall Media.