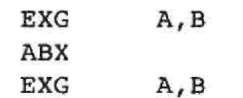
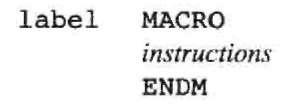
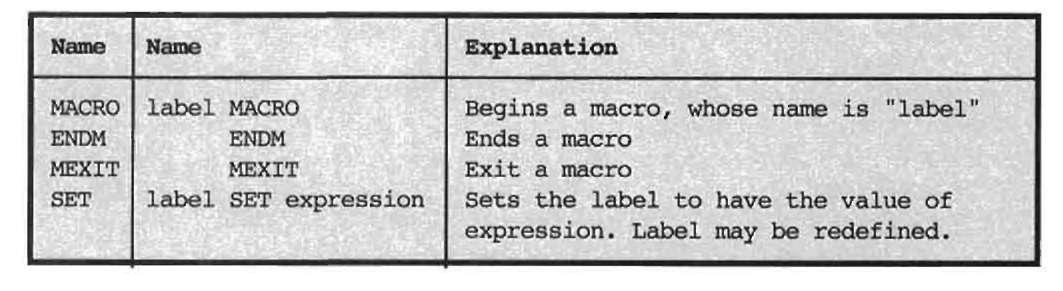
**Macro assembler**

 A macro assembler is able to generate a program segment, which is defined by a macro, when the name of the macro appears as an opcode in a program. The macro assembler is still capable of regular assembler functioning, generating a machine instruction for each line of assembly language code; but like a compiler, it can generate many machine instructions from one line of source code. Its instruction set can be expanded to include new mnemonics, which generate these program segments of machine code. The following discussion of how a macro works will show how this can be done.  
A frequently used program segment can be written just once, in the macro definition at the beginning of a program. For example, the macro  
  
allows the programmer to use the single mnemonic A AX to generate the sequence  
  
The assembler will insert this sequence each time the macro AAX is written in the assembler source code. If the mnemonic AAX is used ten times in the program, the three instructions above will be inserted into the program each time the mnemonic AAX is used. The advantage is clear. The programmer almost has a new instruction that adds the unsigned contents of A to X, and he or she can use it like the real instruction ABX that actually exists in the machine. The general form of a macro is  
  
Here the symbolic name “label” that appears in the label field of the directive is the name of the macro. It must not be the same as an instruction mnemonic or an assembler directive. The phrases MACRO and ENDM are assembler directives indicating the start and the end of the macro.



**Functions/basic Tasks  of Macro Processors**

The fundamental functions common to all macro processors are: ( Code to remember - **DIE**)

o   Macro **D**efinition

o   Macro **I**nvocation

o   Macro **E**xpansion

## Macro Definition

·         Macro definitions are typically located at the start of a program.

·         A macro definition is enclosed between a macro header statement(MACRO) and a macro end statement(MEND)

·         Format of macro definition

macroname MACRO parameters

:

body

:

MEND

·         A macro definition consist of macro prototype statement and body of macro.

·         A macro prototype statement declares the name of a macro and its parameters. It has following format:

*macroname MACRO parameters*

where *macroname*indicates the name of macro, *MACRO*indicates the beginning of macro definition and *parameters*indicates the list of formal parameters. *parameters*is of the form &parameter1, &parameter2,…Each parameter begins with ‘&’. Whenever we use the term macro prototype it simply means the macro name along with its parameters.

·         Body of macro consist of statements that will generated as the expansion of macro.

·         Consider the following macro definition: SUM                  MACRO    &X,&Y

LDA           &X

MOV          B

LDA           &Y

ADD          B MEND

Here, the macro named SUM is used to find the sum of two variables passed to it.

## Macro Invocation(or Macro Call)

·         A macro invocation statement (a macro call) gives the name of the macro instruction being invoked and the arguments to be used in expanding the macro.

·         The format of macro invocation

macroname    p1, p2,...pn

·         The above defined macro can be called as    SUM P,Q

## Macro Expansion

·        Each macro invocation statement will be expanded into the statements that form the body of the macro.

·        Arguments from the macro invocation are substituted for the parameters in the macro prototype.

·        The arguments and parameters are associated with one another according to their positions. The first argument in the macro invocation corresponds to the first parameter in the macro prototype, etc.

·        Comment lines within the macro body have been deleted, but comments on individual statements have been retained.   Macro invocation statement itself has been included as a comment line.

·         Consider the example for macro expansion on next page:

In this example, the macro named SUM is defined at the start of the program. This macro is invoked with the macro call SUM P,Q and the macro is expanded as

LDA           &P

MOV          B

LDA           &Q

ADD          B MEND

Again the same macro is invoked with the macro call SUM M,N and the macro is expanded as

LDA           &M

MOV          B

LDA           &N

ADD          B MEND

Figure: Example for macro expansion

|  |
| --- |
|  |  |

## 5.2    MACRO PROCESSOR DESIGN OPTIONS

**5.5.1        Two Pass Macro Processor**

·         It is easy to design a two-pass macro processor in which all macro definitions are processed during the first pass and all macro invocation statements are expanded during second pass.

·         Such a two pass macro processor cannot handle **nested macro definitions.**

·         Nested macros are macros in which definition of one macro contains definition of other macros.

·         Consider the macro definition example given below, which is used to swap two numbers.

·         The macro named SWAP defines another macro named STORE inside it. These type of macro are called nested macros.

SWAP    MACRO    &X,&Y

LDA         &X

LDX         &Y STORE MACRO &X,&Y

STA          &Y

STX          &X MEND MEND

Inner macro

outer macro

## 5.5.2        One Pass Macro Processor

·         Same as on page 4 (here only brief description is needed)

·         A one-pass macro processor uses only one pass for processing macro definitions and macro expansions.

·         It can handle nested macro definitions.

·         To implement one pass macro processor, the definition of a macro must appear in the source program before any statements that invoke that macro.

·         Data Structures involved in the design of one pass macro processor DEFTAB

NAMTAB ARGTAB

·         Whenever a macro definition is encountered, the macro prototype and body of macro is entered into DEFTAB. References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.

·         The macro name along with the begin and end pointers are entered into NAMTAB.

·         Whenever a macro invocation is encountered, the arguments are entered into ARGTAB.

·         The macro call is expanded with the lines from the DEFTAB. When the ?n notation is recognized in a line from DEFTAB, the corresponding argument is taken from ARGTAB.

## 5.5.3  Recursive Macro Expansion

·      Invocation of one macro by another macro is called recursive macro.

·      Example for recursive macro:

| SUM | MACRO | &X,&Y |
| --- | --- | --- |
|  | STA | &X |
|  | ADD | &Y |
|  | MEND |  |
| INPUT | MACRO | &A,&B |
|  | SUM | &A,&B      .i n v o k i n g the macro SUM |
|  | MEND |  |

Here the macro named INPUT is calling another macro named SUM. This is called as recursive macro.

·      The macro processor design algorithm discussed previously cannot handle recursive macro invocation and expansion.

·        Reasons are:

o   The procedure EXPAND would be called recursively, thus the invocation arguments in the ARGTAB will be overwritten.

o   The Boolean variable EXPANDING would be set to FALSE when the “inner” macro expansion is finished, that is, the macro process would forget that it had been in the middle of expanding an “outer” macro.

o   A similar problem would occur with PROCESSLINE since this procedure too would be called recursively.

·   Solutions:

o   Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained.

o   If we are writing in a language without recursion support, use a stack to take care of pushing and popping local variables and return addresses. So the recursive calls can be handled.

## 5.5.4  General-Purpose Macro Processors

·      The macro processor we discussed so far is related to assembly language programming. Macro processor for high level languages have also been developed. Macro processors which are designed for a specific language are called special purpose macro processors. Example for special purpose macro processor is MASM Macro processor

·      These special purpose macro processors are similar in general function and approach but the implementation details differ from language to language.

·      The general purpose macro processor do not dependent on any particular programming language, but can be used with a variety of different languages.

·      Example for a general purpose macro processor is **ELENA Macro processor**

## Advantages

·      Programmers do not need to learn many macro languages, so much of the time and expense involved in training are eliminated.

·      Although its development costs are somewhat greater than those for a specific language macro processor, this expense does not need to be repeated for each language, thus save substantial overall cost.

## Disadvantages

In spite of the advantages noted, there are still relatively few general purpose macro processors. The reasons are:

·      Large number of details must be dealt with in a real programming language.

·      There are several situations in which normal macro parameter substitution or normal macro expansion should not occur.

o  For example, **comments**are usually ignored by the macro processor. But each programming languages uses its own method for specifying comments. So a general purpose macro processor should be designed with the capability for identifying the comments in any programming languages.

·      Another problem is with the facilities for **grouping together terms, expressions, or statements.**

o  Some languages use keywords such as begin and end for grouping statements while some other languages uses special characters like { }. A general purpose macro processor may need to take these groupings into account in scanning the source statements.

·      Another problem is with the identification of **tokens**of the programming languages. The tokens means the identifiers, constants, operators and keywords in the programming language. Different languages uses different rules for the formation of tokens. So the design of a general purpose macro processor must take this into consideration.

·      Another problem is with the **syntax used for macro definitions and macro invocation**

statements.

## 1.Macro processing outside the language translators

·         The macro processors that we had discussed so far belongs to this group. They are called **macro preprocessors.**

·         They reads the source program statements, process the macro definitions and expand macro invocations, producing an expanded version of the source program.

·         This expanded program is then used as input to an assembler or compiler.

·         So this can be considered as macro processing outside the language translators.

## 2.Macro processing within the language translators

·         In this section we discuss the methods for combining the macro processing functions with the language translator itself.

·         Two common methods are

a)       Line –by- Line Macro Processors

b)       Integrated Macro Processors

## a)   Line-by-line macro processor

·         The simplest method for combining the macro processing functions with the language translator is a line-by-line approach.

·         Using this approach, the macro processor reads the source program statement, process the macro definitions and expand macro invocations. But it does not produce an expanded version of source program.

·         The processed lines are passed to the language translator(compiler or assembler) as they are generated, instead of being written to an expanded source file.

·         Thus macro processor operates as a sort of input routine for the assembler or compiler.

## Benefits of line –by-line macro processor

·                  It avoids making an extra pass over the source program. So it can be more efficient than using a macro preprocessor.

·         Some of the data structures required by the macro processor and the language translator can be combined (e.g., OPTAB and NAMTAB)

·         Many utility subroutines can be used by both macro processor and the language translator. This involves scanning input lines, searching tables , converting numeric values to internal representation etc.

·         It is easier to give diagnostic messages related to the source statements.

## b)   Integrated Macro Processors

·         Although a line-by-line macro processor may use some of the same utility routines as language translator, the functions of macro processing and program translation are still relatively independent.

·         It is possible to have even closer cooperation between the macro processor and the language translator. Such a scheme is called as language translator with an integrated macro processor.

·         An integrated macro processor can potentially make use of any information about the source program that is extracted by the language translator.

·         The macro processor may use the results of the translator operations such as scanning of symbols, constants, etc without involved in processing it.

·         For example in FORTRAN language, consider the statement:

DO 100 I = 1,20

where DO is a keyword, 100 is statement number, I is a variable name etc.

DO 100 I = 1

since in FORTRAN blanks are not significant, this statement is an assignment that gives the value 1 to the variable DO100I. Thus the proper interpretation of characters cannot be decided until the rest of the statement is examined.

## Disadvantages of line-by-line and integrated macro processors

·         They must be specially designed and written to work with a particular implementation of an assembler or compiler. The cost of macro processor development is added to the costs of the language translator, which results in a more expensive software. The assembler or compiler will be considerably larger and more complex than using a macro preprocessor. Size may be a problem if the translator is to run on a computer with limited memory.

## Macro Preprocessor

·         The macro pre-processor(or macro processor) is a system software which replaces each macro instruction with the corresponding group of source language statements. This operation is called **expanding the macro.**

·         It does not concern the meaning of the involved statements during macro expansion.

·         The design of a macro processor generally is machine independent.

## 5.3    MACRO PROCESSOR DESIGN OPTIONS

***5.5.5        Two Pass Macro Processor***

·         *Same as on page 4*

·         *It is easy to design a two-pass macro processor in which all macro definitions are processed during the first pass and all macro invocation statements are expanded during second pass.*

·         *Such a two pass macro processor cannot handle****nested macro definitions.***

·         *Nested macros are macros in which definition of one macro contains definition of other macros.*

·         *Consider the macro definition example given below, which is used to swap two numbers.*

·         *The macro named SWAP defines another macro named STORE inside it. These type of macro are called nested macros.*

## 