MODULE 4

Macro Processors: Basic macro processor functions, machine dependent and machine independent macro processor features, macro processor design options.

MACRO PREPROCESSOR

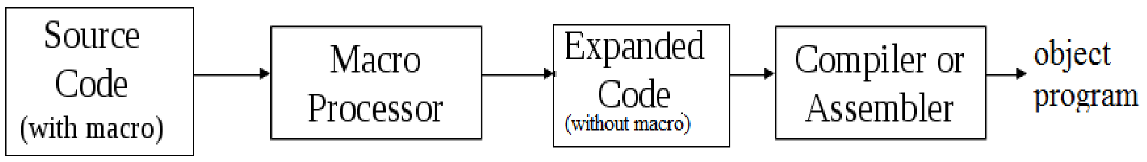
Macro

- A macro (or macro instruction)
  - It is simply a notational convenience for the programmer.
  - It allows the programmer to write shorthand version of a program
  - It represents a commonly used group of statements in the source program.
- For example:
  - Suppose a program needs to add two numbers frequently. This requires a sequence of instructions. We can define and use a macro called SUM, to represent this sequence of instructions.

```
SUM     MACRO   &X,&Y
        LDA     &X
        MOV     B
        LDA     &Y
        ADD     B
        MEND
```

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.



BASIC MACRO PROCESSOR FUNCTIONS

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

- Macro **D**efinition
- Macro **I**nvocation
- 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

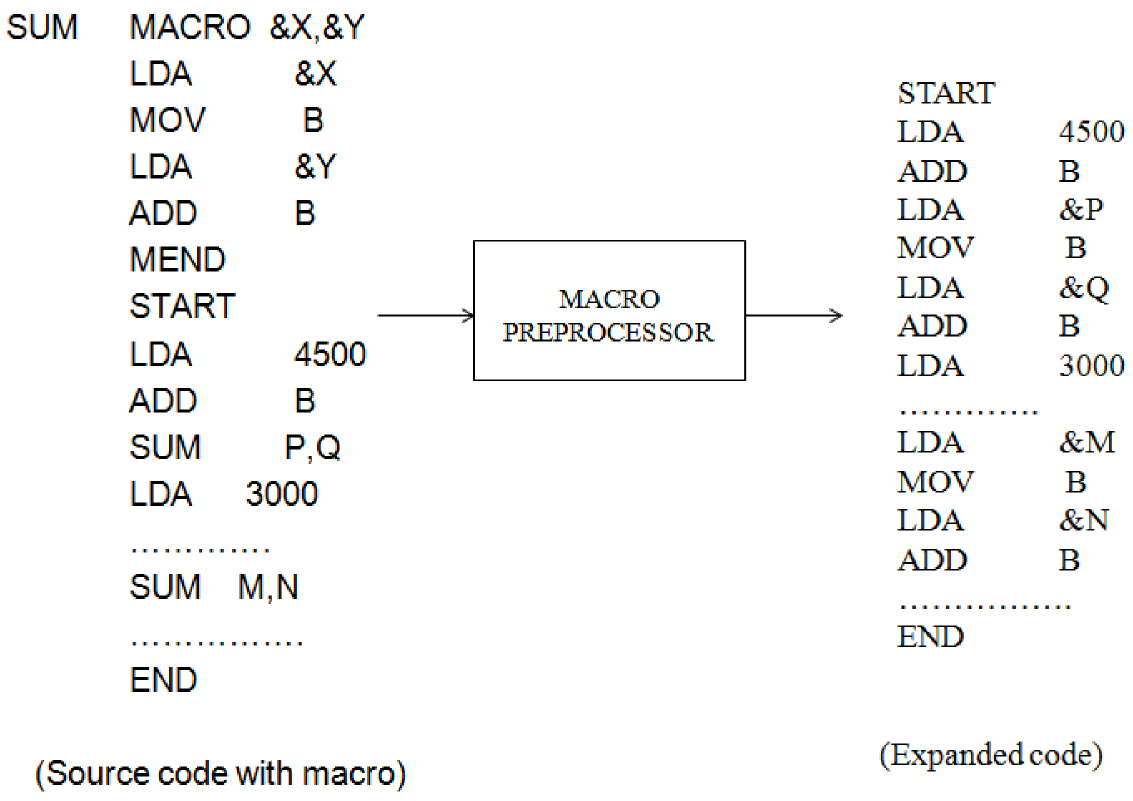
```
LDA     &P
MOV     B
```

```
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



```
SUM     MACRO  &X,&Y
        LDA     &X
        MOV     B
        LDA     &Y
        ADD     B
        MEND
        START
        LDA     4500
        ADD     B
        SUM     P,Q
        LDA     3000
        ............
        SUM     M,N
        ...............
        END
```

```
START
LDA     4500
ADD     B
LDA     &P
MOV     B
LDA     &Q
ADD     B
LDA     3000
............
LDA     &M
MOV     B
LDA     &N
ADD     B
...............
END
```

Difference between Macro and Subroutine

Macros differ from subroutines in two fundamental aspects:

1. A macro call leads to its expansion, whereas subroutine call leads to its execution. So there is difference in size and execution efficiency.
2. Statements of the macro body are expanded each time the macro is invoked. But the statements of the subroutine appear only once, regardless of how many times the subroutine is called.

MACRO PROCESSOR ALGORITHM AND DATA STRUCTURES

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                                 outer macro
        STX      &X         Inner macro
        MEND
        MEND
```

One pass macro processor

- 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

- There are 3 main data structures involved in the design of one pass macro processor:

  DEFTAB

  NAMTAB

  ARGTAB

Definition table (DEFTAB)

- All the macro definitions in the program are stored in DEFTAB, which includes macro prototype and macro body statements.
- Comment lines from macro definition are not entered into DEFTAB because they will not be a part of macro expansion.
- References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.

Name table (NAMTAB)

- The macro names are entered into NAMTAB
- NAMTAB contains pointers to beginning and end of definition in DEFTAB.

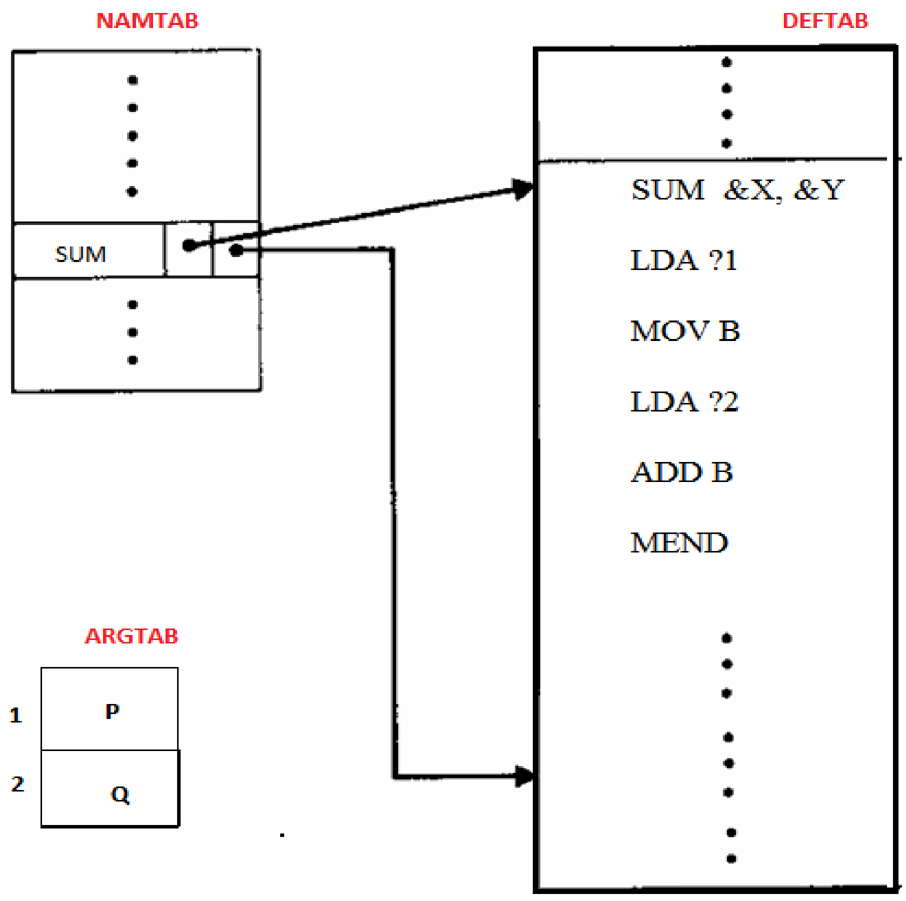
Argument table (ARGTAB)

- The third data structure is an argument table (ARGTAB), which is used during expansion of macro invocations.
- When macro invocation statements are recognized, the arguments are stored in ARGTAB according to their position in argument list.
- As the macro is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body.
- Example: Consider the following source code

```
SUM    MACRO  &X,&Y
       LDA     &X
       MOV      B
       LDA     &Y
       ADD     B
       MEND
       START
       LDA     4500
       ADD     B
       SUM    P,Q
       LDA   3000
       ………….
       END
```

- When the macro definition for SUM is encountered, the macro name SUM along with its parameters X and Y are entered into DEFTAB. Then the statements in the body of macro is also entered into DEFTAB. The positional notation is used for the parameters. The parameter &X has been converted to ?1, &Y has been converted to ?2.
- The macro name SUM is entered into NAMTAB and the beginning and end pointers are also marked.
- On processing the input code, opcode in each statement is compared with the NAMTAB, to check whether it is a macro call. When the macro call SUM P,Q is recognized, the arguments P and Q will entered into ARGTAB. The macro is expanded by taking the statements from DEFTAB using the beginning and end pointers of NAMTAB.
- When the ?n notation is recognized in a line from DEFTAB, the corresponding argument is taken from ARGTAB.

Figure shows the different data structures used



Algorithm for one pass macro processor

```
begin    //macro processor main function
    EXPANDING := FALSE
    while OPCODE ≠ 'END' do
        begin
            GETLINE
            PROCESSLINE
        end
end
```

```
procedure PROCESSLINE
    begin
        search NAMTAB for OPCODE
        if found then
            EXPAND
        else if OPCODE = 'MACRO' then
            DEFINE
        else write source line to expanded file
    end
```

```
procedure DEFINE
    begin
        enter macro name into NAMTAB
        enter macro prototype into DEFTAB
        LEVEL  := 1
        while LEVEL > 0 do
            begin
                GETLINE
                if this is not a comment line then
                    begin
                        substitute positional notation for parameters
                        enter line into DEFTAB
                        if OPCODE = 'MACRO' then
                            LEVEL := LEVEL + 1
                        else if OPCODE = 'MEND' then
                            LEVEL  := LEVEL - 1
                    end {if not comment}
            end {while}
        store in NAMTAB pointers to beginning and end of definition
    end {DEFINE}
```

```
procedure EXPAND
    begin
        EXPANDING  := TRUE
        get first line of macro definition {prototype} from DEFTAB
        set up arguments from macro invocation in ARGTAB
        write macro invocation to expanded file as a comment
        while not end of macro definition do
            begin
                GETLINE
                PROCESSLINE
            end {while}
        EXPANDING := FALSE
    end {EXPAND}
```

```
procedure GETLINE
    begin
        if EXPANDING then
            begin
                get next line of macro definition from DEFTAB
                substitute arguments from ARGTAB for positional notation
            end {if}
        else
            read next line from input file
    end {GETLINE}
```

Explanation of algorithm

- The algorithm uses 5 procedures
    - MACROPROCESSOR (main function)
    - DEFINE
    - EXPAND
    - PROCESSLINE
    - GETLINE

MACROPROCESSOR (MAIN function)

- This function initialize the variable named EXPANDING to false.
- It then calls GETLINE procedure to get next line from the source program and PROCESSLINE procedure to process that line.
- This process will continue until the END of program.

PROCESSLINE

- This procedure checks
  - If the opcode of current statement is present in NAMTAB. If so it is a macro invocation statement and calls the procedure EXPAND
  - Else if opcode =MACRO, then it indicates the beginning of a macro definition and calls the procedure DEFINE
  - Else it is identified as a normal statement(not a macro definition or macro call) and write it to the output file.

DEFINE

- The control will reach in this procedure if and only if it is identified as a macro definition statement.Then:
  - Macro name is entered into NAMTAB
  - Then the macro name along with its parameters are entered into DEFTAB.
  - The statements in body of macro is also enterd into DEFTAB. References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.
  - Comment lines from macro definition are not entered into DEFTAB because they will not be a part of macro expansion.
  - Store in NAMTAB the pointers to beginning and end of definition in DEFTAB.

- To deal with Nested macro definitions DEFINE procedure maintains a counter named LEVEL.
  - When the assembler directive MACRO is read, the value of LEVEL is incremented by 1
  - When MEND directive is read, the value of LEVEL is decremented by 1
  - That is, whenever a new macro definition is encountered within the current definition, the value of LEVEL will be incremented and the while loop which is used to process the macro definition will terminate only after the value of LEVEL =0. With this we can ensure the nested macro definitions are properly handled.

EXPAND

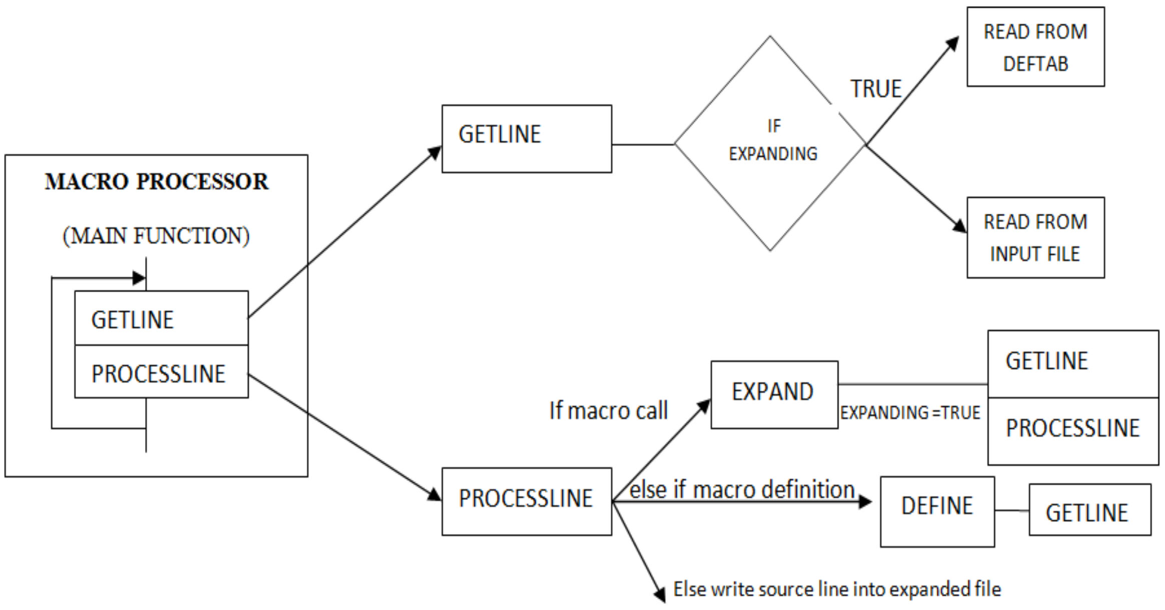
- The control will reach in this procedure if and only if it is identified as a macro call.

- In this procedure, the variable EXPANDING is set to true. It actually indicates the GETLINE procedure that it is going to expand the macro call. So that GETLINE procedure will read the next line from DEFTAB instead of reading from input file.
- The arguments of macro call 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.

GETLINE

- This procedure is used to get the next line.
- If EXPANDING = TRUE, the next line is fetched from DEFTAB. (It means we are expanding the macro call)
- If EXPANDING = False, the next line is read from input file.

Flow Diagram of a one pass macroprocessor



I

MACHINE INDEPENDENT MACRO PROCESSOR FEATURES

- The features of macro which doesn’t depends on the architecture of machine is called machine independent macro processor features.
- The features includes:
    1. Concatenation of Macro Parameters
    2. Generation of Unique Labels
    3. Conditional Macro Expansion
    4. Keyword Macro Parameters

1. Concatenation of Macro Parameters

- Most macro processor allows parameters to be concatenated with other character strings.
- Suppose that a program contains a series of variables named by the symbols XA1, XA2, XA3,..., and another series of variables named XB1, XB2, XB3,..., etc.
- If similar processing is to be performed on each series of labels, the programmer might put this as a macro instruction.
- The parameter to such a macro instruction could specify the series of variables to be operated on (A, B, etc.). The macro processor would use this parameter to construct the symbols required in the macro expansion (XA1, XB1, etc.).
- For example, suppose that the parameter to such a macro instruction is named &ID. The body of the macro definition contain a statement like LDA X&ID1, which means &ID is concatenated after the string “X” and before the string “1”

```
TOTAL   MACRO &ID
        LDA     X&ID1
        ADD     X&ID2
        STA     X&ID3
        MEND
```

The macro call TOTAL A will be expanded as:

```
        LDA     XA1
        ADD     XA2
        STA     XA3
```

- **Ambiguity Problem**: In the statement LDA X&ID1, & is the starting character of the macro parameter; but the end of the parameter is not marked.
- So X&ID1 may mean

  X + &ID + 1 or X +&ID1

- Most of the macro processors deal with this problem by providing a special concatenation operator → to specify the end of parameter.
- Thus LDA X&ID1 can be rewritten as LDA X&ID→1. So that end of the parameter &ID is clearly defined.
- Example:

```
TOTAL   MACRO  &ID
        LDA    X&ID→1
        ADD    X&ID→2
        STA    X&ID→3
        MEND
```

```
                  LDA   XA1
TOTAL A     ⇨     ADD   XA2
                  STA   XA3
```

```
                        LDA   XBETA1
TOTAL BETA  ⇨           ADD   XBETA2
                        STA   XBETA3
```

- The example given above shows a macro definition that uses the concatenation operator as previously described. The statement TOTAL A and TOTAL BETA shows the invocation statements and the corresponding macro expansion.
- The macroprocessor deletes all occurrences of the concatenation operator immediately after performing parameter substitution, so the symbol → will not appear in the macro expansion.

2. Generation of unique labels

- It is not possible to use labels for the instructions in the macro definition, since every expansion of macro would include the label repeatedly.
- This results in *duplicate labels problem* if macro is invoked and expanded multiple times.(which is not allowed by the assembler)

- To avoid this we can use the technique of generating unique labels for every macro invocation and expansion.
- During macro expansion each $ will be replaced with $XX, where XX is a two character alphanumeric counter of the number of macro instructions expansion.
- For the first macro expansion in a program, XX will have the value AA. For succeeding macro expansions XX will be set to AB,AC etc. This allows 1296 macro expansions in a single program
- Consider the following program:

```
SAMPLE   START    0
COPY     MACRO    &A,&B
         LDA      &A
         .........
$LOOP    ADD      &B
         ..........
         JLE      $LOOP
         STA      &B
         MEND
         ............
         COPY     X,Y
         LDA      M
         COPY     P,Q
         ..........
         END
```

After the macro expansion above code becomes:

```
SAMPLE   START    0
         ............
         LDA      &X        ⎫
         .........          ⎪
$AALOOP  ADD      &Y        ⎬ Expansion of  COPY  X,Y
         ..........         ⎪
         JLE      $AALOOP   ⎪
         STA      &Y        ⎭
```

```
            LDA     M
            LDA     &P         }
            .........          }
$ABLOOP     ADD     &P         }
            ..........         }  Expansion of COPY P,Q
            JLE     $ABLOOP    }
            STA     &Q         }
            ...........
            END
```

In the example, for the first invocation of COPY X,Y the label generated is $AALOOP and for the second invocation COPY P,Q the label generated is $ABLOOP. Thus for each invocation of macro unique label is generated.

3. **Conditional Macro Expansion**

- Arguments in macro invocation can be used to:
    - Substitute the parameters in the macro body without changing the sequence of statements expanded.(sequential macro expansion)
    - Modify the sequence of statements for conditional macro expansion. This capability increases power and flexibility of a macro language.
- **Macro-Time Variables:**
    - Macro-time variables (often called as SET Symbol) can be used to store working values during the macro expansion.
    - It can also be used to store the evaluation result of Boolean expression
    - Any symbol that begins with & and not a macro instruction parameter is considered as macro-time variable. All such variables are initialized to zero.
    - The value of macro-time variables can be modified by using the macro processor directive **SET**
    - The macro processor must maintain a symbol table to store the value of all macro-time variables used. The table is used to look up the current value of the macro-time variable whenever it is required.

Implementation of Macro-Time Conditional Structure IF-ELSE-ENDIF

- Structure of IF-ELSE_ENDIF:

```
Macroname   MACRO    &COND
            ........
             IF (&COND NE ’’)
                 .part I
             ELSE
                 .part II
            ENDIF
             .........
             MEND
```

- When an IF statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated.
- **If the value of this expression TRUE,**
    - The macro processor continues to process lines from the DEFTAB until it encounters the ELSE or ENDIF statement.
    - If an ELSE is found, macro processor skips lines in DEFTAB until the next ENDIF.
    - Once it reaches ENDIF, it resumes expanding the macro in the usual way.
- **If the value of the expression is FALSE,**
    - The macro processor skips ahead in DEFTAB until it encounters next ELSE or ENDIF statement.
    - The macro processor then resumes normal macro expansion.

- Example for conditional macro:

```
COPY   START    0
EVAL   MACRO    &X,&Y,&Z
       IF       (&Y LE &X)
       LDA      &X
       SUB      &Z
       ELSE
       LDA      &Y
       ADD       &Z
       ENDIF
       MEND
       STA     P
       ...................
```

```
                EVAL 2 ,3 ,4
                STA       Q
                ...............
                END
```

After expansion the above code becomes:

```
          COPY    START      0
                  STA        P
                  ..................
                  LDA        3
                  ADD        4
                  STA        Q
                  ........................
                  END
```

Implementation of Macro-time looping(or Expansion time looping) structure: WHILE-ENDW

- WHILE statement specifies that the following lines until the next ENDW statement, are to be generated repeatedly as long as a particular condition is true. The testing of this condition, and the looping are done during the macro under expansion.
- When a WHILE statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated.
- If expression is TRUE, the macro processor continues to process lines from DEFTAB until it encounters the next ENDW statement.When ENDW is encountered, the macro processor returns to the preceding WHILE, re-evaluates the Boolean expression, and takes action based on the new value.
- If the expression is FALSE, the macro processor skips ahead in DEFTAB until it finds the next ENDW statement and then resumes normal macro expansion.
- The example given below shows the usage of Macro-Time Looping statement. In this example, the variable &LIMIT is a macro time variable.

```
COPY      START     0
EVAL      MACRO     &X,&Y,&Z
&LIMIT    SET       1
          WHILE     (&LIMIT LE &Y)
          LDA       &X
          ADD       &Z
&LIMIT    SET       &LIMIT+1
          ENDW
          MEND
FIRST     STL       RETADR
          EVAL  0 ,3 ,2
RETADR    RESW      1
          END  FIRST
```

Above loop after expansion

```
COPY      START    0
FIRST     STL      RETADR
          .EVAL  0 ,3 ,2
          LDA      0
          ADD      2
          LDA      0
          ADD      2
RETADR    RESW     1
          END  FIRST
```

4. Keyword Macro Parameters

- The parameters used in macro can be of two types
  - a)Positional Parameters
  - b)Keyword Parameters

a) Positional Parameters

- Parameters and arguments are associated according to their positions in the macro prototype and invocation. The programmer must specify the arguments in proper order.
- Syntax : In macro definition,

  macroname MACRO &parameter1, &parameter2,……

  In macro invocation,

  macroname argument1, argument2,….

  Example: In macro definition,

  EVAL MACRO &X, &Y

  In macro invocation,

  EVAL P,Q

  Here &X recieves the value of P and &Y recieves the value of Q
- If an argument is to be omitted, a null argument should be used to maintain the proper order in macro invocation statement.
- For example: Suppose a macro named EVAL has 5 possible parameters, but in a particular invocation of the macro only the 1st and 4th parameters are to be specified. Then the macro call will be EVAL P,,,S,

- Positional parameter is not suitable if a macro has a large number of parameters, and only a few of these are given values in a typical invocation.

b) Keyword Parameters

- Each argument value is written with a keyword that names the corresponding parameter.
- Arguments may appear in any order. That is not any importance to the position of arguments.
- Null arguments no longer need to be used.
- For macros using keyword parameters the macro prototype statement specification is different. Here each parameter name is followed by equal sign, which identifies a keyword parameter and a default value is specified for some of the parameters.
- Keyword parameters are easier to read and much less error-prone than the positional parameters.
- It is of the form

  &formal parameter name = <ordinary string>
- Consider the example:

```
INCR   MACRO   &X=, &Y=, &Z=
       MOV     &Z, &X
       ADD     &Z, &Y
       MOV     &Z, &X
       MEND
```

  The following calls are now equivalent

  1) INCR X=A, Y=B, Z=C
  2) INCR Y=B, X= A, Z= C

Default specification of parameters

- Default specification of parameters can also be possible in macros.
- This specification 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 that macro call.
- Consider the example:

Here the default value R is specified for the parameter Z

```
INCR    MACRO     &X=, &Y=, &Z=R
        MOV       &Z, &X
        ADD       &Z, &Y
        MOV       &Z, &X
      MEND
```

- Then the macro call INCR X=A, Y=B will take the values A for parameter X, B for parameter Y and R for the parameter Z.
- The macro call INCR X=A, Y=B, Z=C will take the values A for parameter X, B for parameter Y and C for the parameter Z.

MACRO PROCESSOR DESIGN OPTIONS

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.

```
SWAP   MACRO   &X,&Y
       LDA     &X
       LDX     &Y
STORE  MACRO  &X,&Y                                  outer macro
       STA     &Y
       STX     &X          Inner macro
       MEND
       MEND
```

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.

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:
    - The procedure EXPAND would be called recursively, thus the invocation arguments in the ARGTAB will be overwritten.
    - 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.
    - A similar problem would occur with PROCESSLINE since this procedure too would be called recursively.
- Solutions:
    - Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained.
    - 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.

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.
    - 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.**
    - 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.

Macro Processing within Language Translators

- Macros can be processed
    1. Outside the language translators
    2. Within the Language translators

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.

TYPES OF MACRO

Different types of macro are 1. Parameterized Macro 2. Nested Macro 3. Recursive Macro

1. Parameterized Macro

- Macros which uses parameters are called parameterized macro.This type of macro has the capability to insert the given parameters into its expansion.Macros we studied so far belongs to this type.
- Example:

```
SUM     MACRO   &X,&Y
        LDA     &X
        MOV     B
        LDA     &Y
        ADD     B
        MEND
```

2. Nested Macros

- 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    }      }  outer macro
        STX     &X    }  Inner macro
        MEND          }      }
        MEND                 }
```

3. Recursive Macro

- 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.