CS202 – System Software

Dr. Manish Khare

Macro Processors

- Macros are special code fragments that are defined once in the program and are used repetitively by calling them from various places within the program.
- It is similar to the subprogram in the sense that both can be used to organize the program better by separating out the frequently used fragment into a different block.
- The main program calls this block of code as and when needed. Both of them can have an associated list of parameters.
- However, the implementation of macro differs significantly from subprograms.
- In the case of subprogram, during execution of the program, at each call to the subprogram, control branches to it and returns to the calling point at the end of executing the subprogram

- This involves saving a lot of context information (such as values of arguments passed, local variables, CPU registers, return address, etc.) into the stack.
- The context is to be restored when control returns after executing the subprogram.
- Microprocessors work by simple code substitution. Moreover, the substitution takes place at the time of compiling/assembling the program.
- Each call to the macro is replaced by the body of the macro directly with the parameters substituted by the arguments passed.
- Thus, once the expansion has taken place, there is no necessity to do context switching during execution.
- This saves a lot of time, particularly if the macro is called quite often.

- To decide upon whether a code fragment be realized as a subprogram or as a macro, the following points are to be noted.

- **1. Size:** The size of the code fragment is an important parameter. In the case of a macro, each call is replaced by the code fragment. Thus, if the size is significantly high, with all these substitutions, the compiled/assembled program may be quite big, which may result in slower execution of the program. On the other hand, a small fragment may better be implemented as macro, since substitution of it will not increase the code size significantly. At the same time, the saving in context switching will result in faster execution of the program.

- **2. Number of parameters:** A large number of parameters in a subprogram will necessitate creating all of them in the stack at the time of its invocation. This will require large stack-area and may slow down execution.
- **3. Debugging:** Debugging may be easier with a subprogram since the execution may be suspended at the entry to the subprogram. But, in the case of macros, the code substitution will change the source line numbers and also setting breakpoints at each of the substituted code fragments become cumbersome.

- **4. Recursion:** It cannot be implemented with a macro because the recursive calls are to be executed at the run-time and this is not possible at compile-time.
- **5. Parameter passing:** Since in the macro expansion, each reference to a parameter within the body is substituted by the argument passed from the corresponding macro call, the only type of parameter passing strategy that can be supported is call-by-name. On the other hand, subprograms in most of the programming languages support call-by-value and/or call-by-reference strategies.

Macro

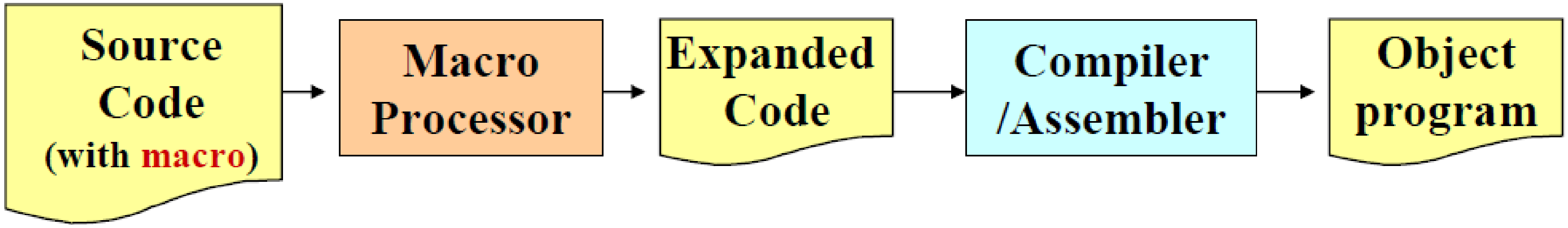
- A macro (or macro instruction)
  - It is simply a notational convenience for the programmer.
  - It allows the programmer to write shorthand version of a program
  - It represents a commonly used group of statements in the source program.

- 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.

- Three examples of actual macro processors:
  - A macro processor designed for use by assembler language programmers
  - Used with a high-level programming language
  - General-purpose macro processor, which is not tied to any particular language



- C uses a macro preprocessor to support language extensions, such as named constants, expressions, and file inclusion.

```
#define max(a,b)  ((a<b)?(a):(b))
#define MACBUF 4
#include <stdio.h>
```

Classification of MACROS

- Lexical expansion
  - Lexical expansion implies replacement of a character string by another character string during program generation.
  - Lexical expansion is to replace occurrences of formal parameters by corresponding actual parameters.

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

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

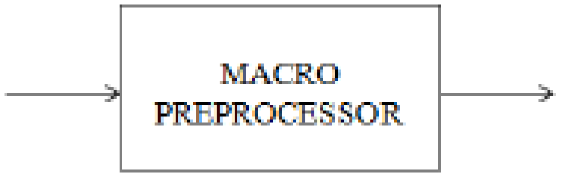
```
LDA     &P
MOV     B
LDA     &Q
ADD     B
MEND
```

- 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
```

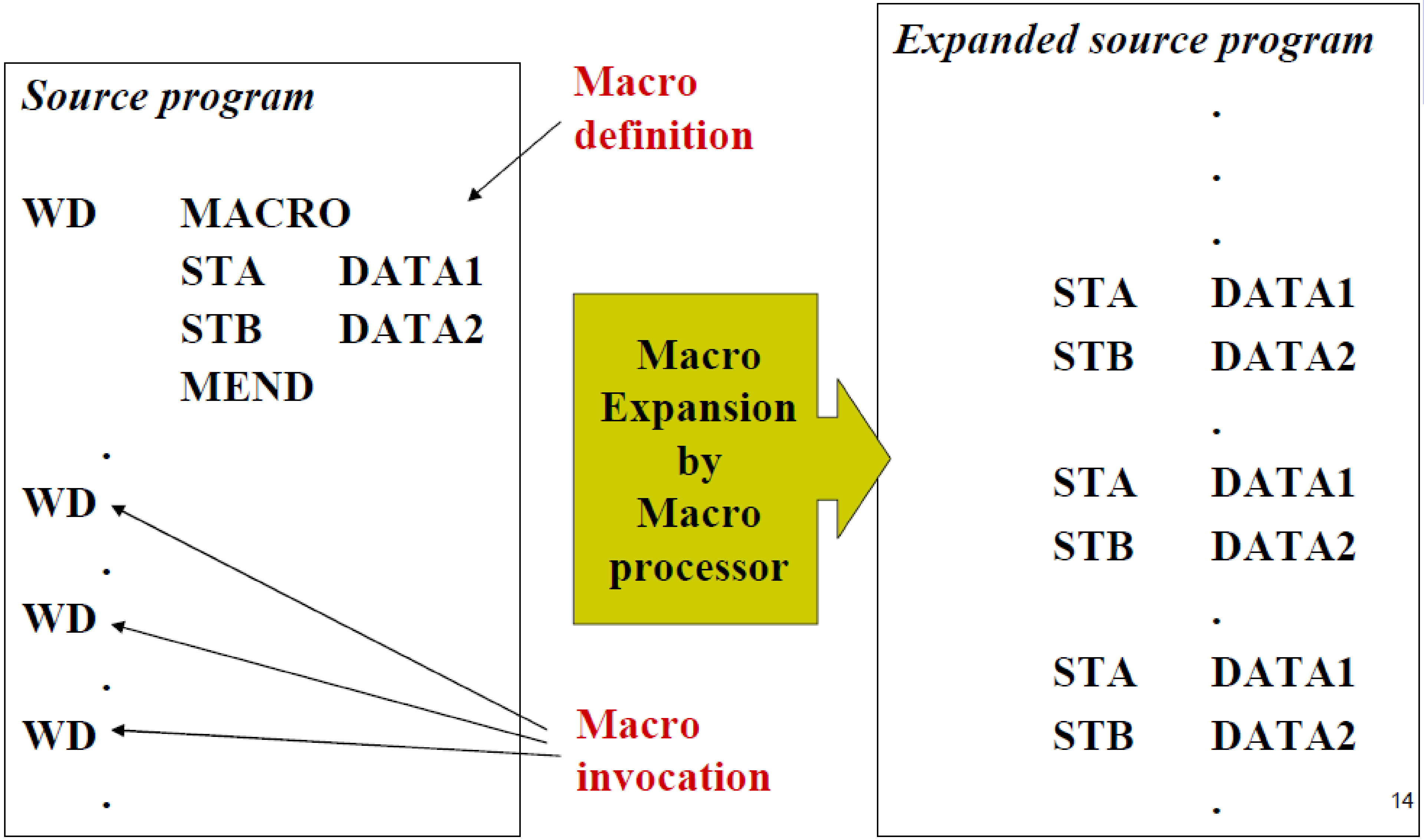
```
SUM    MACRO  &X,&Y
       LDA       &X
       MOV        B
       LDA       &Y
       ADD       B
       MEND
       START
       LDA       4500
       ADD       B
       SUM       P,Q
       LDA    3000
       .............
       SUM   M,N
       ................
       END
```

(Source code with macro)



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

(Expanded code)

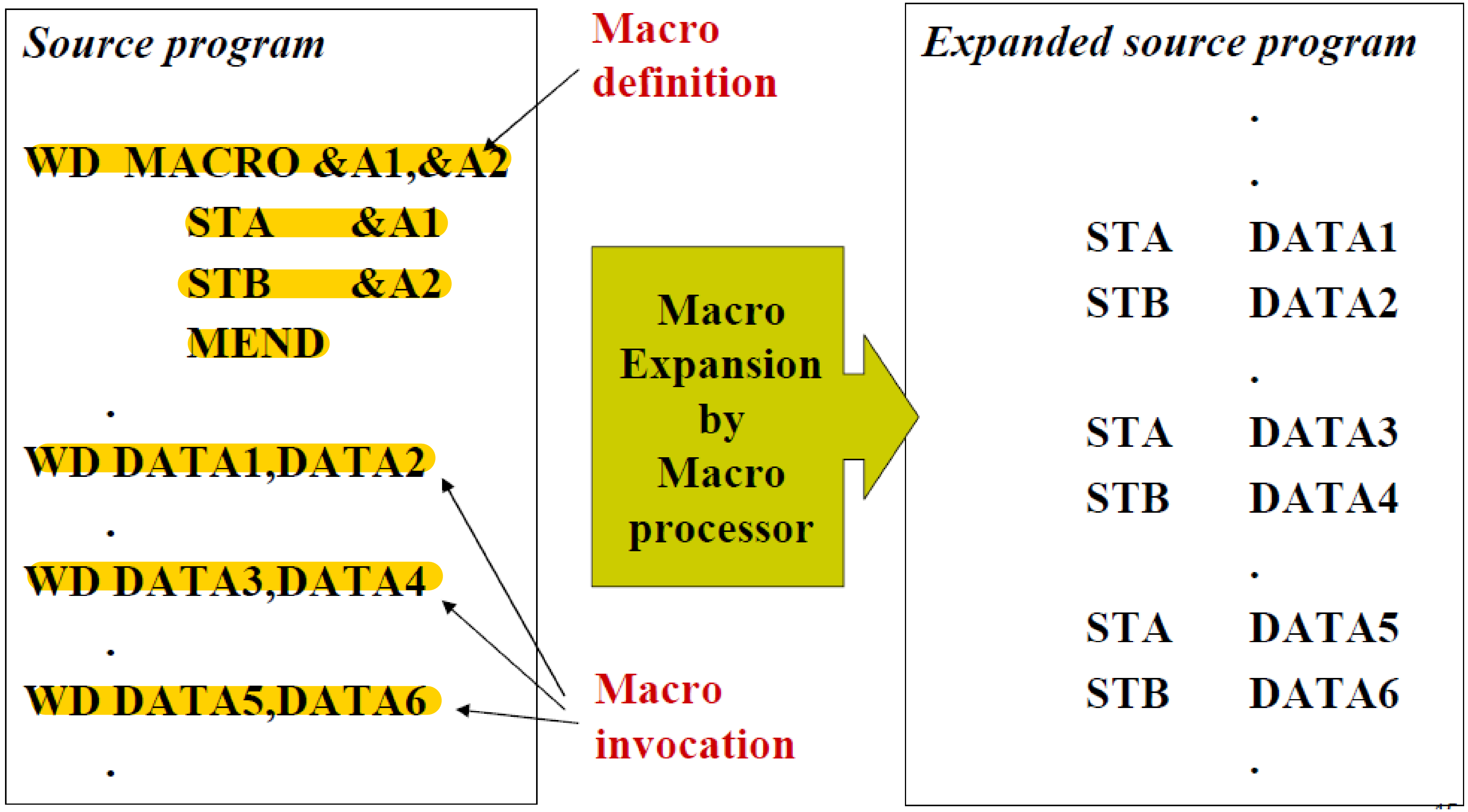


Source program

```
WD      MACRO
        STA     DATA1
        STB     DATA2
        MEND
   .
WD
   .
WD
   .
WD
   .
```

Expanded source program

```
                .
                .
                .
        STA     DATA1
        STB     DATA2
                .
        STA     DATA1
        STB     DATA2
                .
        STA     DATA1
        STB     DATA2
                .
```



Source program

```
WD  MACRO &A1,&A2
          STA      &A1
          STB      &A2
          MEND
    .
WD DATA1,DATA2
    .
WD DATA3,DATA4
    .
WD DATA5,DATA6
    .
```

Expanded source program

```
                .
                .
       STA      DATA1
       STB      DATA2
                .
       STA      DATA3
       STB      DATA4
                .
       STA      DATA5
       STB      DATA6
                .
```

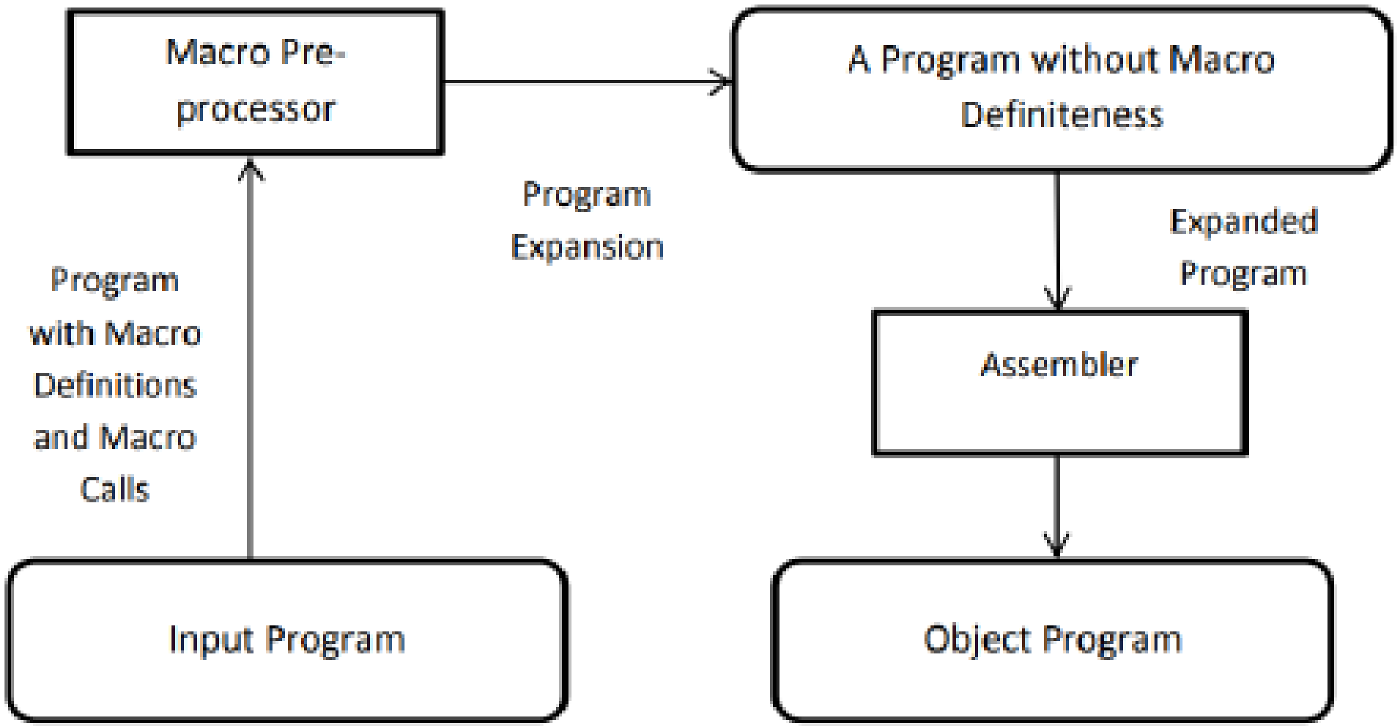
Difference between Macro and Subroutine

Macro	Subroutine
Macro name in the mnemonic field leads to expansion only.	Subroutine name in a call statement in the program leads to execution.
So there is difference in size and execution efficiency.	
Statements of the macro body are expanded each time the macro is invoked	Statements of the subroutine appear only once, regardless of how many times the subroutine is called.
Macros are completely handled by the assembler during assembly time.	Subroutines are completely handled by the hardware at runtime.
Macro definition and macro expansion are executed by the assembler. So, the assembler has to know all the features, options, and exceptions associated with them. The hardware knows nothing about macros.	Hardware executes the subroutine call instruction. So, it has to know how to save the return address and how to branch to the subroutine. The assembler knows nothing about subroutines.

Design of Macro Preprocessor

- Macro preprocessors are vital for processing all programs that contain macro definitions and/or calls. Language translators such as assemblers and compilers cannot directly generate the target code from the programs containing definitions and calls for macros.
- Therefore, most language processing activities by assemblers and compilers preprocess these programs through macro processors.
- A macro preprocessor essentially accepts an assembly program with macro definitions and calls as its input and processes it into an equivalent expanded assembly program with no macro definitions and calls.
- The macro preprocessor output program is then passed over to an assemble to generate the target object program.

- The general design semantics of a macro preprocessor is shown as below



- The design of a macro preprocessor is influenced by the provisions for performing the following tasks involved in macro expansion:

  - Recognize Macro Calls
  - Determine the values of formal parameters
  - Maintain the values of expansion time variables declared in a macro
  - Organize expansion time control flow
  - Determine the values of sequencing symbols
  - Perform expansion of a model statement:

Recognize Macro Calls

- A table is maintained to store names of all macros defined in a program.
- Such a table is called Macro Name Table (MNT) in which an entry is made for every macro definition being processed.
- During processing program statements, a match is done to compare strings in the mnemonic field with entries in the MNT.
- A successful match in the MNT indicates that the statement is a macro call.

Determine the values of formal parameters

- A table called Actual Parameter Table (APT) holds the values of formal parameters during the expansion of a macro call.
- The entry into this table will be in pair of the form (,).
- A table called Parameter Default Table (PDT) contains information about default parameters stored as pairs of the form (,) for each macro defined in the program.
- If the programmer does not specify value for any or some parameters, its corresponding default value is copied from PDT to APT.

- Maintain the values of expansion time variables declared in a macro
  - A table called Expansion time Variable Table (EVT) maintains information about expansion variables in the form (,).
  - It is used when a preprocessor statement or a model statement during expansion refers to an EV.

- Organize expansion time control flow
  - A table called Macro Definition Table (MDT) is used to store the body of a macro.
  - The flow of control determines when a model statement from the MDT is to be visited for expansion during macro expansion.
  - MEC {Macro Expansion Counter) is defined and initialized to the first statement of the macro body in the MDT.
  - MDT is updated following an expansion of a model statement by a macro preprocessor.

➢ Determine the values of sequencing symbols

- A table called Sequencing Symbols Table (SST) maintains information about sequencing symbols in pairs of the form

  (<sequencing symbol name>, <MDT entry #>)

- Where <MDT entry #> denotes the index of the MDT entry containing the model statement with the sequencing symbol. Entries are made on encountering a statement with the sequencing symbol in their label field or on reading a reference prior to its definition.

Perform expansion of a model statement

- The expansion task has the following steps:
  - MEC points to the entry in the MDT table with the model statements.
  - APT and EVT provide the values of the formal parameters and EVs, respectively.
  - SST enables identifying the model statement and defining sequencing.

Functions of Macro Processor

The design and operation of a macro processor greatly influence the activities performed by it.

In general, a macro processor will perform the following tasks:

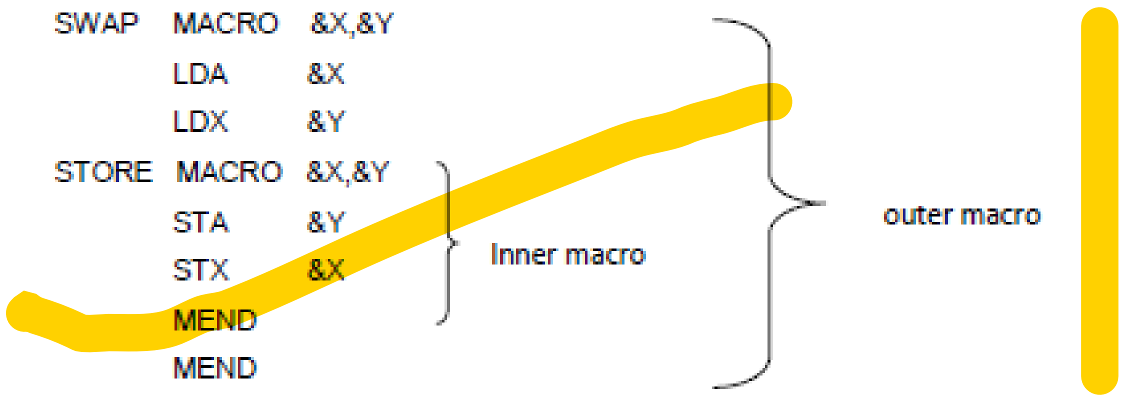
- Identifies macro definitions and calls in the program.
- Determines formal parameters and their values.
- Keeps track of the values of expansion time variables and sequencing symbols declared in a macro.
- Handles expansion time control flow and performs expansion of model statements.

Macro Processor Algorithm and Data Structures

- Two pass Macro Processor
- One pass Macro Processor

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
```

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

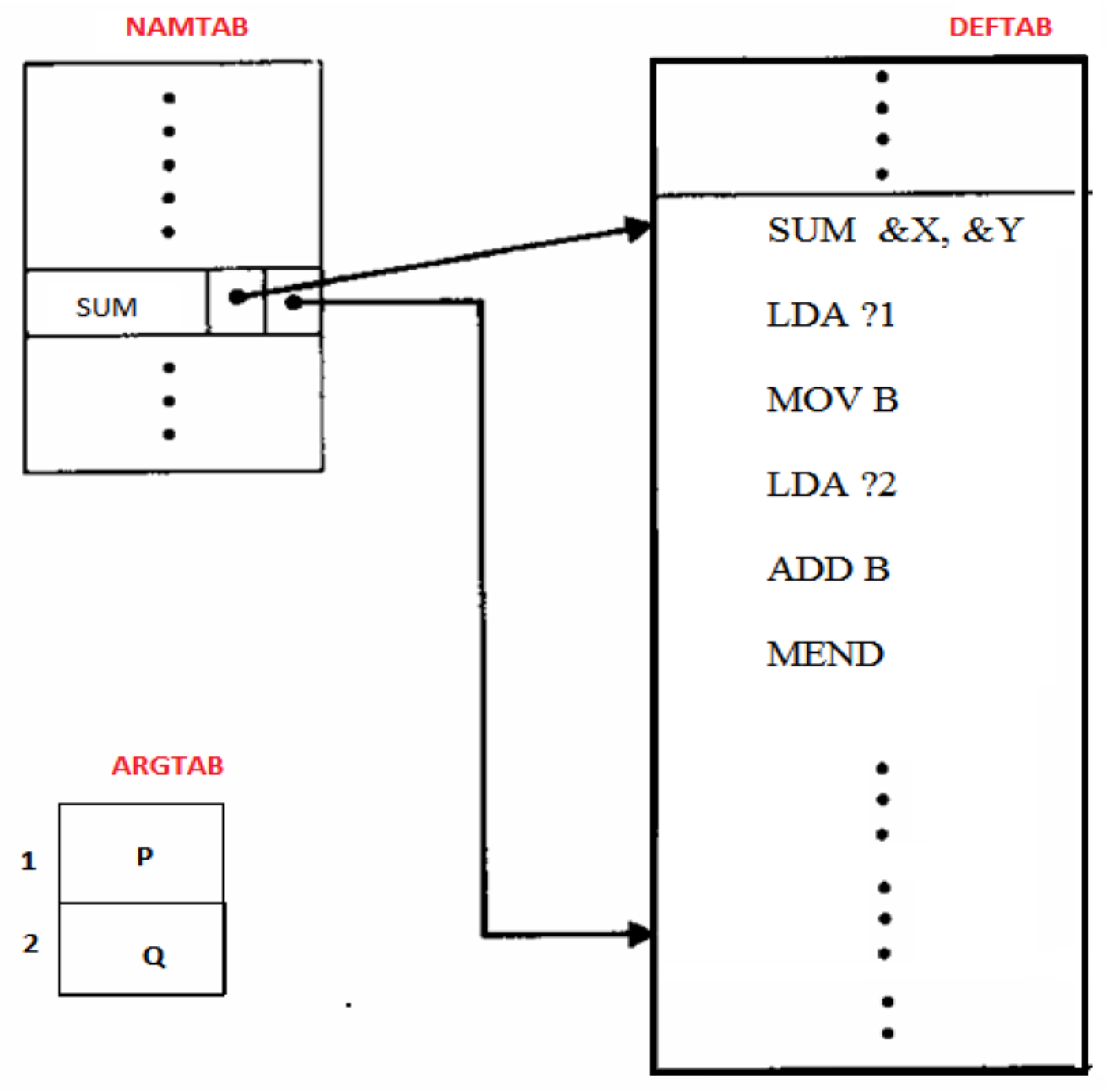
```
SUM         MACRO  &X,&Y
            LDA    &X
            MOV    B
            LDA    &Y
            ADD    B
            MEND
            START
            LDA    4500
            ADD    B
            SUM    P,Q
            LDA    3000
            ………….
            END
```

- 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.

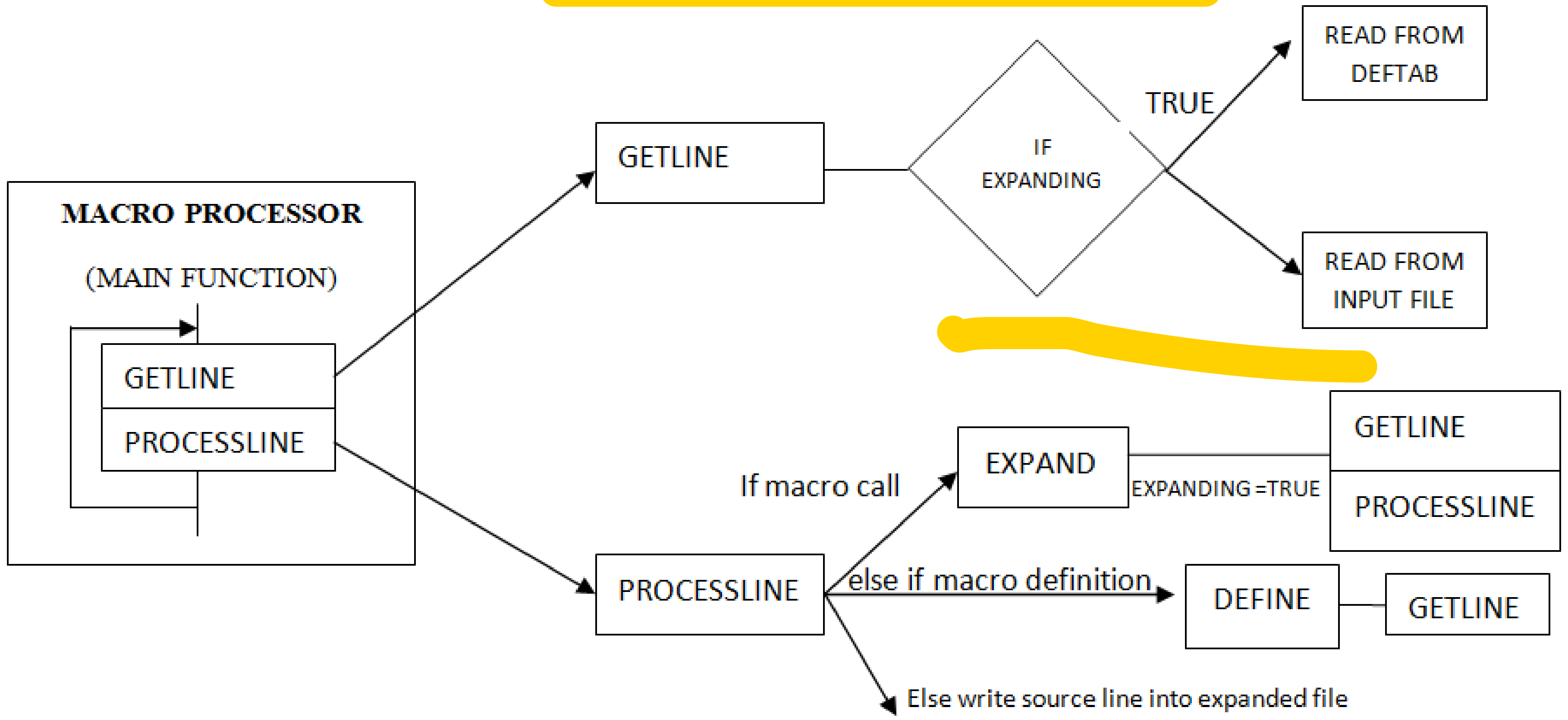
- The macro name SUM is entered into NAMTAB and the beginning and end pointers are also marked.

- 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.

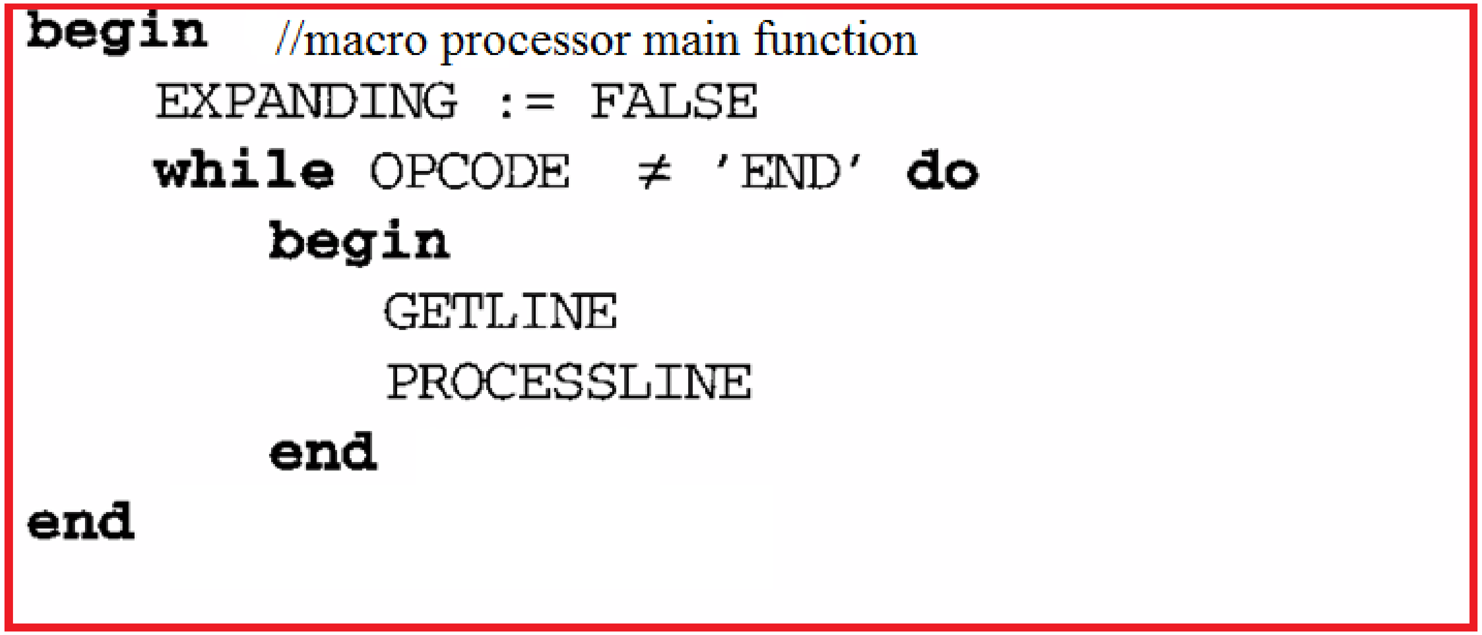
- When the ?n notation is recognized in a line from DEFTAB, the corresponding argument is taken from ARGTAB.


NAMTAB
SUM
DEFTAB
SUM &X, &Y
LDA ?1
MOV B
LDA ?2
ADD B
MEND
ARGTAB
1
P
2
Q

Flow Diagram of a one pass macroprocessor



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 entered 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.