CHAPTER 04

MACRO LANGUAGE AND THE MACRO PROCESSOR

MACRO INSTRUCTIONS

In assembly language programming it is often that some set or block of statements get repeated every now. In this context the programmer uses the concept of *macro instructions (*often called as *macro)* where a single line abbreviation is used for a set of line. For every occurrence of that single line the whole block of statements gets expanded in the main source code.

This is gives a high level feature to assembly language that makes it more convenient for the user to write code easily. Many computer gives use macro instructions to automate the writing of "tailored" operating systems in a process called as *system generation*.

In its simplest form, a macro is an abbreviation for a sequence of operations. Consider the program given in example 1. In this example we see that the three set of statements (add instruction with DATA) occurs twice. The macro processor effectively constitutes a separate language processor with its own language. It provides a name to the sequence of statements (called macro name) that are repeated and ever needed it make use of the name in the main program.

Example1:

```
        :
A           1, DATA
A           2, DATA
A           3, DATA
        :
A           1, DATA
A           2, DATA
A           3, DATA
        :
DATA  DC    F'5'
```

The macro definition consists of the following parts:

1. Start of definition MACRO
2. Macro name [ ]
3. Sequence of statements -----------
   ----
4. End of definition MEND

In the definition of the MACRO and MEND are the pseudo code that represents the start and the end of the macro definition. MACRO being the first statement, the second statement being the name of the macro instruction with is an identifier what will be used whenever the macro is called. The statements from the third till the MEND represent the body of the macro.

```
        Source                  |        Expanded source
MACRO                           |
INCR                            |
A         1, DATA               |
A         2, DATA               |
A         3, DATA               |
MEND                            |
.                               |
.                               |        :
INCR                            |    {   A     1,DATA
                                |    {   A     2,DATA
.                               |    {   A     3,DATA
.                               |        :
INCR                            |    {   A     1,DATA
                                |    {   A     2,DATA
.                               |    {   A     3,DATA
.                               |
DATA1     DC      F'5'          |  DATA1     DC    F'5'
```

Once the macro is defined then the name of macro instruction now acts as a mnemonic in assembly language that is equivalent to sequence of the statements. Now whenever the macro instruction is encountered, the whole of the macro gets substituted in place of the macro instruction. This process of substitution is called as *macro expansion*. This expansion takes place as the definition of the macro instruction is saved in the macro processer and is used whenever the macro instruction is encountered. A *macro call* represents the expansion of the macro in the source program.

FEATURES OF MACRO FACILITIES

Macro instruction has got a number of features depending on which we will be trying to design our macro processer or in other words we can say that the macro processor are designed in a manner to give the following features to the macro instructions.

MACRO INSTRUCTION ARGUMENTS

We always know that a macro instruction just expands the macro definition but does not allow any modification to the macro definition. But we can do the same with the data only by the help of arguments. As in the example 2 we can see the same set of statements makes use of different data in each block.

Example 2:

```
          :
A            1, DATA 1
A            2, DATA 1
A            3, DATA 1
          :
A            1, DATA 2
A            2, DATA 2
A            3, DATA 2
          :
DATA 1  DC   F'5'
DATA 2  DC   F'10'
```

In this case the macro is defined with help of arguments called as *dummy* or *macro instruction arguments.* In the example it is clear that the first block of statements makes use of DATA1 and the second set makes use of the DATA2. So here in the macro definition along with the macro name the arguments

```
        Source                                  |      Expanded source
MACRO                   Macro INCR has          |
                        one argument            |
INCR      &ARG                                  |
A         1,&ARG                                |
A         2,&ARG                                |
A         3,&ARG                                |
MEND                                            |
.                                               |
.                                               |          :
INCR      DATA1         Use DATA1 as            |     {  A     1,DATA1
                        operand                 |     {  A     2,DATA1
.                                               |     {  A     3,DATA1
.                                               |          :
INCR      DATA2         Use DATA2 as            |     {  A     1,DATA2
                        operand                 |     {  A     2,DATA2
.                                               |     {  A     3,DATA2
.                                               |
DATA1     DC      F'5'                          |  DATA1     DC    F'5'
DATA2     DC      F'10'                         |  DATA2     DC    F'10'
```

are also defined and statement in the body makes use of these arguments in place of data. In the first macro call INCR DATA1, DATA1 is the argument and the second macro call DATA2 is the argument that is passed.

Let us now consider a different case where the data in the block of statement are not the same and also the blocks make use of labels. We can define macro for this kind of code by passing sufficient number of arguments to handle the data in the set of statements.

```
SOURCE                                           EXPANDED SOURCE
          :
       MACRO
&LAB   INCR    &ARG1,&ARG2,&ARG3
&LAB   A       1,&ARG1
       A       2,&ARG2
       A       3,&ARG3
       MEND
          :                                          :
LOOP1  INCR    DATA1,DATA2,DATA3          LOOP1   A     1,DATA1
                                                  A     2,DATA2
                                                  A     3,DATA3
          :                                          :
LOOP2  INCR    DATA3,DATA2,DATA1          LOOP2   A     1,DATA3
                                                  A     2,DATA2
                                                  A     3,DATA1
          :                                          :
DATA1  DC      F'5'                       DATA1   DC    F'5'
DATA2  DC      F'10'                      DATA2   DC    F'10'
DATA3  DC      F'15'                      DATA3   DC    F'15'
          :
```

We also make of *label argument* that is used to define the label for a set of statement. The label arguments can be defined in two ways i.e. before the name of the macro (*positional argument*)or in the normal argument list (*dummy arguments)*. Whenever these kind of macros are called the corresponding values in the calls are passed the matching arguments in the definition.

CONDITIONAL MACRO EXPANSION

The AIF and AGO pseudo codes provides conditional operations within a macro. This will be clear from example 4 where we have set of statements having variable number of statements.

In this case we define a macro that makes use of AIF (conditional statement). Even in some cases AGO

Example 3:

```
          :
LOOP1  A   1, DATA1
       A   2, DATA2
       A   3, DATA3
          :
LOOP2  A   1, DATA3
       A   2, DATA2
       A   3, DATA1
          :
DATA1  DC  F'5'
DATA2  DC  F'10'
DATA3  DC  F'15'
```

```
Source                                        Expanded source
       MACRO
       INCR    &ARG1,&ARG2,&ARG3,&LAB
&LAB   A       1,&ARG1
       A       2,&ARG2
       A       3,&ARG3
       MEND
  ...                                         ...
       INCR    DATA1,DATA2,DATA3,LOOP1        LOOP1  A  1,DATA1
                                                     A  2,DATA2
                                                     A  3,DATA3
  ...                                         ...
       INCR    DATA3,DATA2,DATA1,LOOP2        LOOP2  A  1,DATA3
                                                     A  2,DATA2
                                                     A  3,DATA1
  ...                                         ...
```

Example 4:

```
          :
LOOP1  A   1, DATA1
       A   2, DATA2
       A   3, DATA3
          :
LOOP2  A   1, DATA3
       A   2, DATA2
          :
LOOP3  A   1, DATA1
          :
DATA1  DC  F'5'
DATA2  DC  F'10'
DATA3  DC  F'15'
          :
```

(unconditional statement) is used. In this macro we make use of one additional argument COUNT that keeps a count of statement used in the block. The AIF with help of COUNT finds the number of statements that is to be expanded.

Here .FINI is used as a *macro label*. Here the label is used to mark the end of the macro. With the help of conditional macros we can produce program equivalent to that of high level languages.

```
       MACRO
&ARG0  VARY    &COUNT,&ARG1,&ARG2,&ARG3
&ARG0  A       1,&ARG1
       AIF     (&COUNT EQ 1).FINI          Test if & COUNT = 1
       A       2,&ARG2
       AIF     (&COUNT EQ 2).FINI          Test if & COUNT = 2
       A       3,&ARG3
.FINI  MEND
                                           Expanded source
          :                                     :
LOOP1  VARY    3,DATA1,DATA2,DATA3         LOOP1  A  1,DATA1
                                                  A  2,DATA2
                                                  A  3,DATA3
          :                                     :
LOOP2  VARY    2,DATA3,DATA2               LOOP2  A  1,DATA3
                                                  A  2,DATA2
          :                                     :
LOOP3  VARY    1,DATA1                     LOOP3  A  1,DATA1
          :                                     :
DATA1  DC      F'5'
DATA2  DC      F'10'
DATA3  DC      F'15'
```

MACRO CALLS WITHIN MACRO

A macro can some time make use of some other macro within its definition. This will be clear from the example 5 where the macro ADDS make use of macro ADD1 within it. This is allowed with an assumption that the macro that is called within a macro should be defined before it is used in a second macro.

Since we have macro called with in other macro so there need to be more than one expansion for a particular macro calls. In the example 5 we have macro ADDS is expanded first at first level of expansions, followed by the expansions of ADD1 at the second level of expansion. This feature

Example 5:

```
MACRO
ADD1    &ARG
L       1, &ARG
A       1, =F'1'
ST      1, &ARG
MEND
MACRO
ADDS    &ARG1, &ARG2, &ARG3
ADD1    &ARG1
ADD1    &ARG2
ADD1    &ARG3
MEND
```

```
Source                          Expanded source        Expanded source
                                (Level 1)              (Level 2)
   :
MACRO
ADD1   &ARG
L      1,&ARG
A      1,=F'1'
ST     1,&ARG
MEND
MACRO
ADDS   &ARG1,&ARG2,
       &ARG3
ADD1   &ARG1
ADD1   &ARG2
ADD1   &ARG3
MEND
                                Expansion of ADDS      Expansion of ADD1
   :                                 :                      :
                                ADD1  DATA1            L   1,DATA1
                                                       A   1,=F'1'
                                                       ST  1,DATA1
ADDS   DATA1,DATA2,             ADD1  DATA2            L   1,DATA2
       DATA3                                           A   1,=F'1'
                                                       ST  1,DATA2
                                ADD1  DATA3            L   1,DATA3
                                                       A   1,=F'1'
                                                       ST  1,DATA3
   :
DATA1  DC   F'5'                                       DATA1  DC  F'5'
DATA2  DC   F'10'                                      DATA2  DC  F'10'
DATA3  DC   F'15'                                      DATA3  DC  F'15'
   :
```

also facilitates the process of *recursion*.

MACRO INSTRUCTIONS DEFINING MACROS

We can also define a macro with in a second macro. This is clear from example 6 where the macro DEFINE defines a macro named COS within it. Let there be a call like DEFINE COS, then the macro DEFINE is called or expanded and it is then the macro COS is defined. At this point of time we can access the macro COS but not before a call to the macro DEFINE. Each and every time the inner function is to be called, then this is followed by a call of the first or the outer macro.

```
Definition of macro DEFINE {
        MACRO
        DEFINE  &SUB            Macro name: DEFINE
  Definition of macro &SUB {
        MACRO
        &SUB    &Y              Dummy macro name
        CNOP    0,4             Align boundary
        BAL     1,*+8           Set register 1 to parameter list pointer
        DC      A(&Y)           Parameter list pointer
        L       15,=V(&SUB)     Address of subroutine
        BALR    14,15           Transfer control to subroutine
        MEND
  }
        MEND
}
```

IMPLEMENTATION

Whenever we think of a macro, there are two tasks associated with it i.e. macro definition and macro call. The following are the process that are employed in the implantation of a macro processor.

STATEMENT OF PROBLEM

There are four basic task that any macro processor is associated with are as follows:

1. *Recognizing macro definition:* We recognize the macro definition in a program by the statements that appear in between MACRO and corresponding MEND. The definition of macro includes these two pseudo codes.
2. *Save the definition:* The definitions of all the macro are saved onto a table so that they can be expanded when ever in future they are called.
3. *Recognize the calls:* A macro call is similar to a machine instruction whose work is predefined i.e. the definition is stored in some table.
4. *Expand calls and substitute arguments:* Each time a macro call is encountered a table consisting of macro name is searched and all the code in the macro definition except the MACRO, MEND and the macro name card is expanded in the main program. The actual source program now looks like as it was intended by the programmer.

IMPLEMENTATION TWO PASS ALGORITHM

In case of two pass macro processors we need to make some assumptions that they are functionally independent form the assembler and their output will be serving as he input for the assembler. Due to this we exclude the macro definition and macro call with in other macro as these constructions are much complex.

Macro languages are so closely related as assembly language (not much related with address and location within a program). Macro definition refers to nothing outside them i.e. a macro call refer only to macro definition and only text get substituted but not values for parameters.

Our macro processor algorithm will make two systematic scans, or passes over the inputs text, searching first for macro definition and saving them, and then for macro calls as macros cannot be expanded until they are defined unlike symbols in assemblers. Hence the first pass handles the definition and the second pass handles the macro calls or expansions. The macro definitions are stored in MDT(macro definition table) and the names of the macros are stored in MNT(macro name table).

SPECIFICATION OF DATA BASES

The following are the data bases used by the two passes of the macro processor.

Pass 1 data bases:

1. The input macro source deck
2. The output macro source deck copy for use by pass 2
3. The macro definition table (MBT), used to store the body of the macro definitions.
4. The macro name table (MNT), used to store the names of defined macros.
5. The macro definition table counter(MDTC), used to indicate the next available entry in the MDT.
6. The macro name table counter (MNTC), used to indicate the next available entry in the MNT.
7. The argument list array (ALA), used to substitute index markers for dummy arguments before storing a macro definition.

Pass 2 data bases:

1. The copy of the input macro source deck.
2. The output expanded source deck to be used as input to the assembler.
3. The macro definition table (MDT), created in pass1.
4. The macro name table (MNT), created in pass1.
5. The macro definition table pointer (MDTP), used to indicate the next line of text to be used during macro expansion.
6. The argument list array (ALA), used to substitute macro call arguments for the index markers in the stored macro definition.

SPECIFICATION OF DATA BASE FORMAT

The only data bases with non trivial format are the MDT, MNT and ALA. Others are of less importance.

ARGUMENT LIST ARRAY: used during both pass1 and pass2 but some the functions are just the reverse. During pass 1 the ALA stores the arguments in the macro definition with positional indicator when the definition is stored in MDT i.e. the ith argument is stored in the ALA as #i. This arrangement is in according to the order of argument in which they appear in the MDT. Later on in the pass2, when there is macro expansion the ALA fills the arguments of the corresponding index with its appropriate argument in the call. This will clear from the example. Here when LOOP INCR DATA1, DATA2, DATA3 is executed the ALA fills the argument fields of the corresponding index #0, #1, #2, #3 with LOOP, DATA1, DATA2 and DATA3.

Argument List Array

Index	Argument (8 bytes per entry)
0	"LOOP1*bbb*"
1	"DATA1*bbb*"
2	"DATA2*bbb*"
3	"DATA3*bbb*"

(*b* denotes the blank character)

MACRO DEFINTION TABLE: It is table of text lines. It consists of two fields, index that keep track of line numbers of the macro definition and the card that is 80 bytes of size and is responsible for storing the macro definition. Everything except the pseudo code MACRO is inserted into the MDT. MEND pseudo code marks the end of the macro definition.

Macro Definition Table

Index	Card (80 bytes per entry)		
⋮		⋮	
15	&LAB	INCR	&ARG1,&ARG2,&ARG3
16	#0	A	1, #1
17		A	2, #2
18		A	3, #3
19		MEND	
⋮		⋮	

MACRO NAME TABLE: It is similar to out MOT and POT in assembler. It has got three field, Index field that keep track of various macro that are defined, the Name field that keep track of names of the macros and the MDT index is a pointer to the entry in MDT.

Index	Name (8 bytes)	MDT index (4 bytes)
⋮	⋮	⋮
3	"INCR*bbbb*"	15
⋮	⋮	⋮

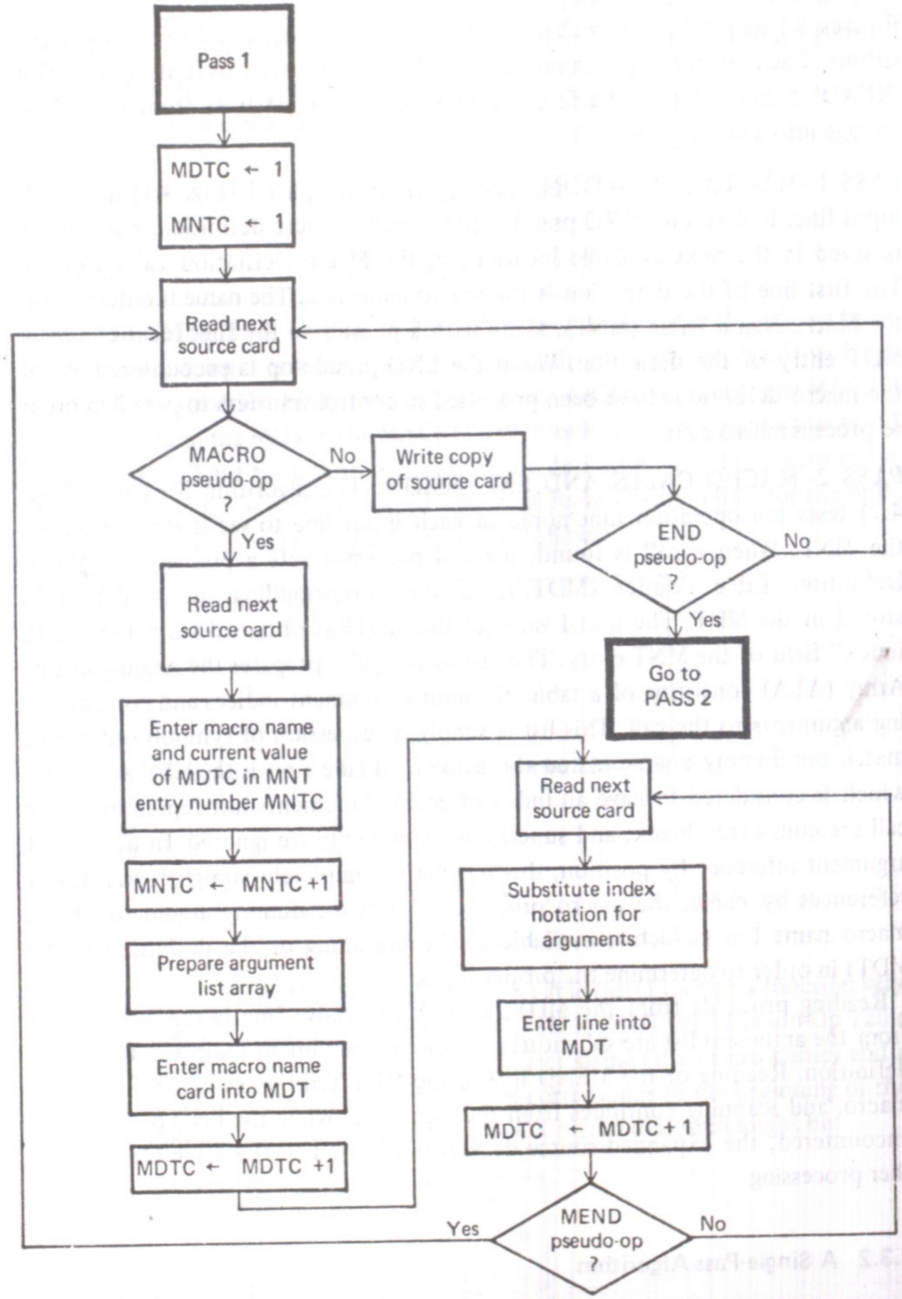

FIGURE 4.1 Pass 1 – processing macro definitions

ALGORITHM

The two figures show the flowchart of the macro definition and macro calls. Each of the algorithms makes a line by line scan over the input. The READ refers to fetching of successive input lines form secondary storage into a workspace.

Pass 1:

1. Initializes MDTC and MNTC to 1
2. Reads next source card.
3. If MACRO pseudo code then
   a. Read from next source card
   b. Enter macro name and current value of MDTC in MNT entry number MNTC.
   c. Increment MNTC.
   d. Prepare ALA
   e. Enter macro name card into MDT.
   f. Increment MDTC.
   g. Read next source card.
   h. Substitute index notation for the arguments.
   i. Enter line into MDT
   j. Increment MDTC
   k. If MEND GOTO 2. else GOTO3.g.
4. Else write copy of source card.
5. If END then GOTO pass2 else GOTO 2.

Pass 2:

1. Read next source card(copied by pass1).
2. Search MNT for match with operation code.
3. If Macro name found then
   a. MDTP<-MDT index form MNT entry.
   b. Setup ALA
   c. Increment MDTP.
   d. Get line form MDT.
   e. Substitute arguments from macro call.
   f. If MEND then GOTO 1 else GOTO 3.c.
4. Else write into expanded source card file
5. If END then, supply expanded source file to assembler processing, else GOTO 1.

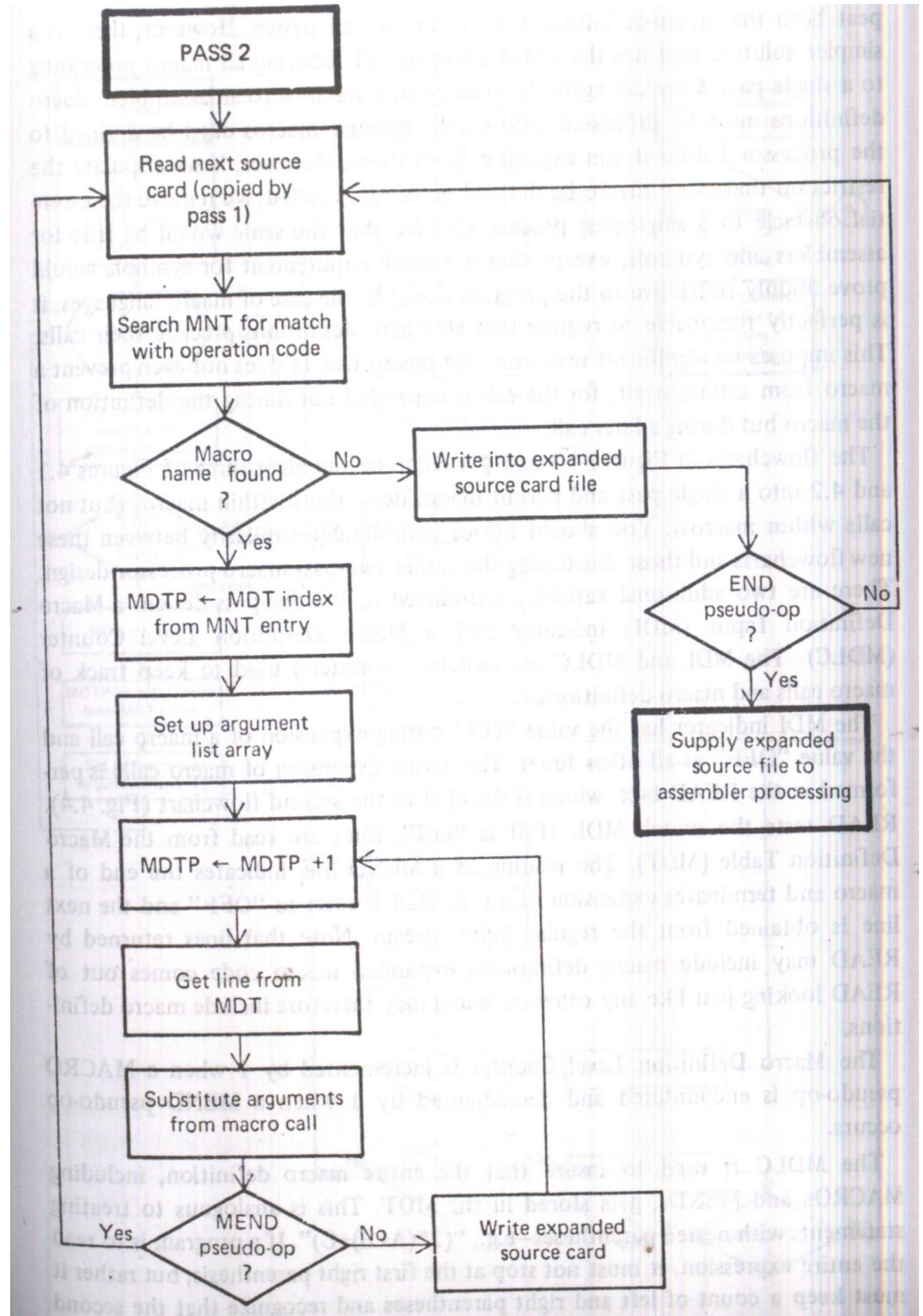

FIGURE 4.2 Pass 2 –processing macro calls and expansion

IMPLEMENTATION A SINGLE PASS ALGORITHM

Definition of a macro within other macro is possible in case of one pass macro processor. Here the inner macro is defined only after the outer one has been called: in order to provide for any use of the inner macro, we would have to repeat the both the macro definition and the macro call passes. This can be assumed by considering that the macros are never called before they are defined.

Here we make use of additional data structures like macro definition indicator (MDI) and macro definition level counter (MDLC). The MDI and MDLC are the switches used to keep track of macro calls and macro definition.

The MDI has status "ON" during the expansion of macro call and the value "OFF" all the other times. When its value is "ON" the cards are read from the MDT and when it is "OFF" the cards are read from the input source card. The use of MDLC is used keep track of the level of macros while defining the macros. Initially it is zero and it is incremented each time a MACRO code is found within a macro. The reverse process happens in case of MEND i.e. the valued of MDLC is decremented by one each time it encounters a MEND and the process continues till the MDLC is zero i.e. the completion of macro definition.

ALGORITHM

The process of one pass macro process can be clearly understood with the help of a MAIN algorithm that make use of a sub algorithm named READ.

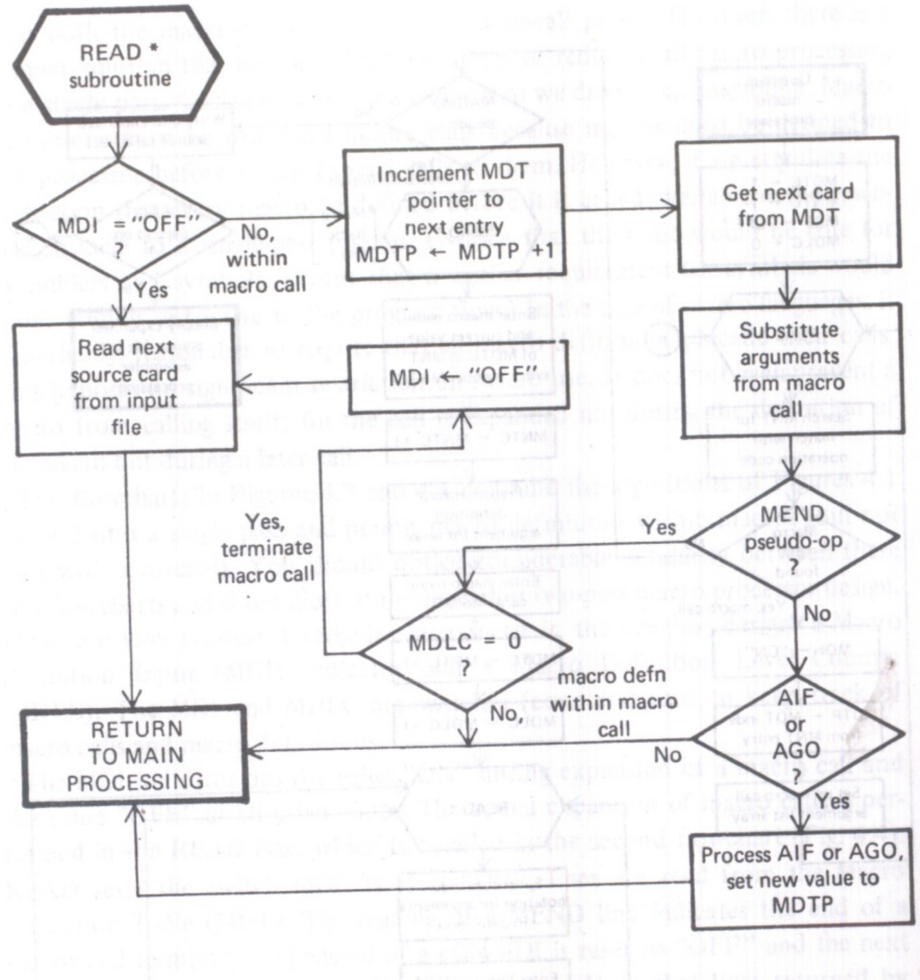

FIGURE 4.4 Detail of READ function used for macro expansion

READ: (Macro call expansion or read a next instruction form the source input card)

1. If MDI ="OFF", then
   a. Read next source card from input file.
   b. Return to MAIN algorithm.
2. Else increment MDT pointer to next entry MDTP<-MDTP+1.
3. Get next card from MDT.
4. Substitute arguments from macro call.
5. If MEND pseudo code
   a. Then if MDLC=0,
      i. Then MDI<-"OFF".
      ii. GOTO 1.a.
   b. Else return to MAIN algorithm.
6. Else if AIF or AGO present
   a. Then process AIF or AGO and update MDTP.
   b. Return to MAIN algorithm.
7. Else return to MAIN algorithm.

MAIN: (One pass macro processor)

1. Initialize MDTC and MNTC to 1, MDI to “OFF” and MDLC to 0.
2. READ
3. Search MNT for match with operation code.
4. If macro name found
    a. MDI<-“ON”.
    b. MDTP<-MDT index from MNT entry.
    c. Setup macro call ALA.
    d. GOTO 2.
5. Else if MACRO pseudo code
    a. Then READ. //macro name line.
    b. Enter macro name and current value of MDTC in MNT entry number MNTC.
    c. Increment MNTC <- MNTC+1.
    d. Prepare macro definition ALA.
    e. Enter macro name card into MDT.
    f. MDTC<-MDTC+1.
    g. MDLC<-MDLC+1.
    h. READ.
    i. Substitute index notation for arguments in definition.
    j. Enter line into MDT.
    k. MDTC<-MDTC+1
    l. If MACRO pseudo code
        i. MDLC<-MDLC+1
        ii. GOTO 5.h.
    m. Else If MEND pseudo code
        i. Then MDLC<-MDLC-1
            1. If MDLC=0 the
                a. Then GOTO 2.
            2. Else GOTO 5.h.
    n. Else GOTO 5.h.
6. Write into expanded source card file.
7. If END pseudo code
    a. Then Supply expanded source file to assembler processing.
8. Else GOTO 2.

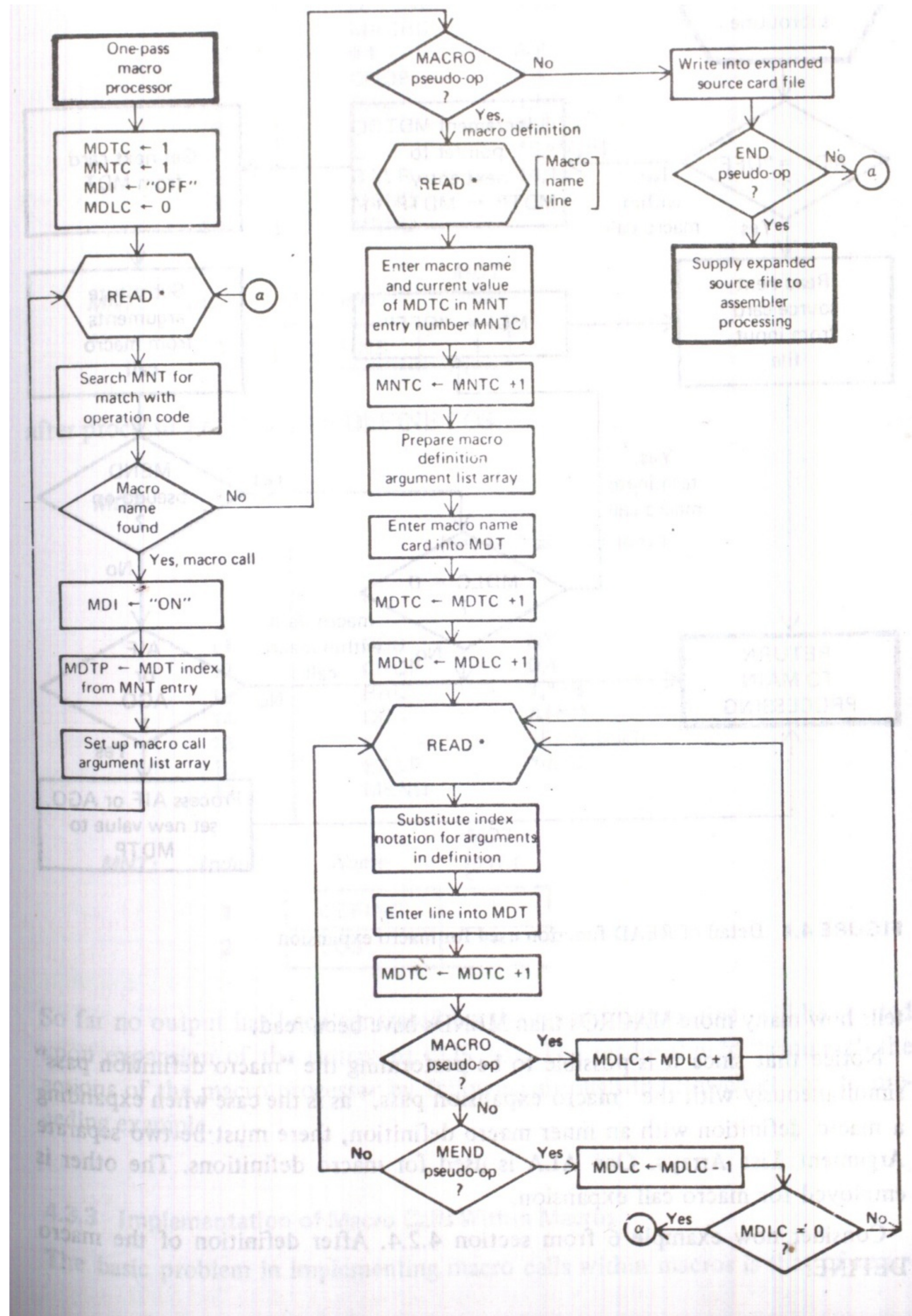

FIGURE 4.3 Simple one-pass macro processor

MDT:

Index		
1	DEFINE	&SUB
2	MACRO	
3	#1	&Y
4	CNOP	0,4
5	BAL	1,*+8
6	DC	A(&Y)
7	L	15,=V(#1)
8	BALR	14,15
9	MEND	
10	MEND	

MNT:

Index	Name	MDT Index
1	DEFINE	1

after processing of statement DEFINE COS:

MDT:

Lines 1-10 same as above

Index		
.	.	
.	.	
.	.	
11	COS	&Y
12	CNOP	0,4
13	BAL	1,*+8
14	DC	A(#1)
15	L	15,=V(COS)
16	BALR	14,15
17	MEND	

MNT:

Index	Name	MDT Index
1	DEFINE	1
2	COS	11

Here in this case we have "macro definition pass" simultaneously with the "macro expansion pass" as in the case when expanding a macro definition within an inner macro definition, then there must be two separate ALAs. One for macro definition and the other employed for macro call expansion. This will be clear from the example 6 as given below.

IMPLEMENTATION MACRO CALLS WITHIN MACROS

Here we try to implement a macro processor where a macro is called within another macro. This can be easily handled by making use of more than one macro processor, each responsible for expanding a particular macro. But the use of more than one macro processor makes the picture more complex. Hence something is done store the status of macro while they are expanded as the outer macro needs to be expanded form a position from where it was left while calling the inner macro.

What actually happens it that when a inner macro is called then the MDI is set "ON" and the inner macro gets expanded and after it is completed the MDI is turned "OFF", and the instruction are read from the source card, while the outer macro is still unread. The ALA containing the arguments of the outer macro is overwritten when the inner macro is called as the same ALA is reconstructed. The MDTP is out of track after the expansion of the inner macro, since now it should point to the unexpanded part of the outer macro. This three problems takes place can overcome by saving the status of the outer macro when the macro call with in it.

We make use of stack to store the state of macro when a macro call is made with in another macro. The stack being LIFO help in resolving the problem of nested macro calls. The stack is divided into *stack frames* that are responsible for storing each of the nested macro calls. There is a *stack pointer (SP)* that stores the address of previous stack frame. For the first frame SP =1 and S(SP)=-1. The condition that SP=1 is equivalent to MDLC=0, informs that processor is reading from the source card. S(SP+1) contains

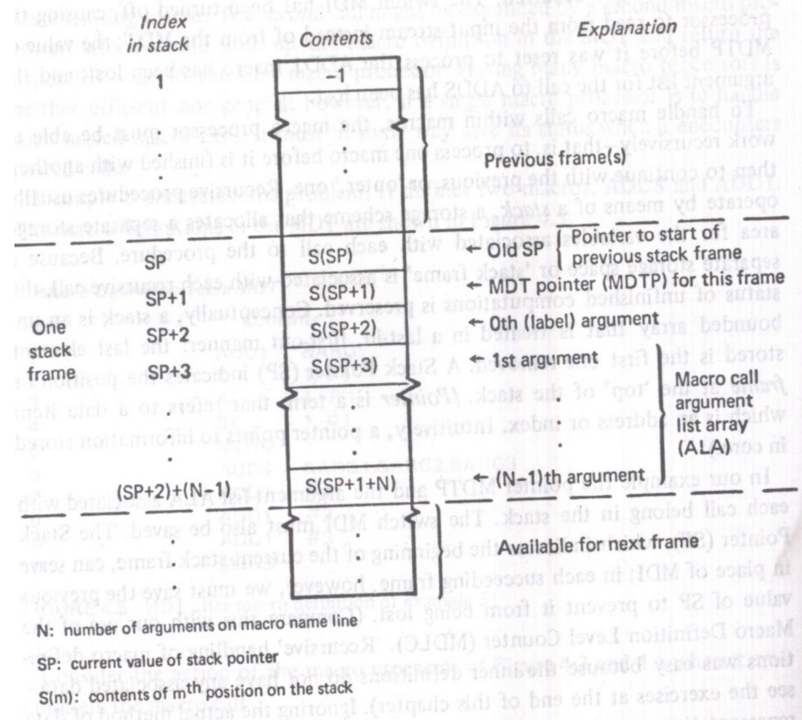

FIGURE 4.6 Stack organization

the MDTP value and S(SP+2)…S(SP+N+1) contain the N character string of the argument list.

The algorithm of this macro processor is similar to that of single pass macro processor but it makes use of SP instead of MDI and MDTC.

READ: (Macro call expansion or read a next instruction form the source input card)

1. If SP=-1, then
    a. Read next source card from input file.
    b. Return to MAIN algorithm.
2. Else increment MDT pointer to next entry S(SP+1)<-S(SP+1)+1.
3. Get next card from MDT, pointer is S(SP+1).
4. Substitute arguments from macro call. S(SP+2)… S(SP+N+1)
5. If MEND pseudo code
    a. Then if MDLC=0,
        i. Then N<-SP-S(SP)-2.
        ii. SP<-S(SP)
        iii. GOTO 1.
    b. Else return to MAIN algorithm.
6. Else if AIF or AGO present
    a. Then process AIF or AGO and set new value to MDTP, S(SP+1).
    b. Return to MAIN algorithm.
7. Else return to MAIN algorithm.

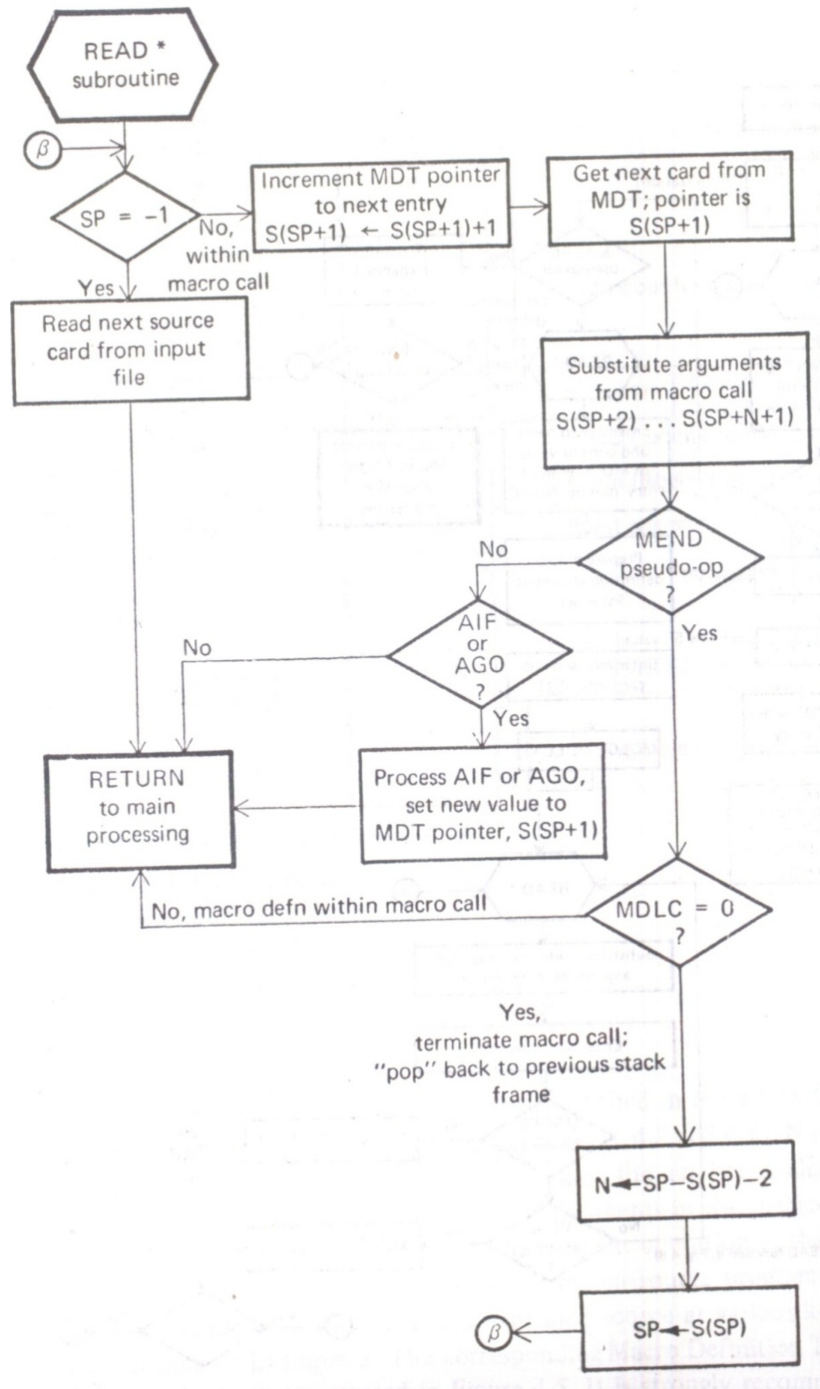

FIGURE 4.8 Detail of READ function for recursive macro expansion

MAIN: (One pass macro processor)

1. Initialize MDLC to 0, N to 0 and SP to -1.
2. READ
3. Search MNT for match with operation code.
4. If macro name found
    a. S(SP+N+2)<-SP.
    b. SP<-SP+N+2.
    c. S(SP=1)<-MDT index from MNT entry
    d. Setup macro call ALA in S(SP+2)…S(SP+N+1) where N=total number of argument.
    e. GOTO 2.
5. Else if MACRO pseudo code
    a. Then READ. //macro name line.
    b. Enter macro name and current value of MDTC in MNT entry number MNTC.
    c. Prepare macro definition ALA.
    d. Enter macro name card into MDT.
    e. MDLC<-MDLC+1.
    f. READ.
    g. Substitute index notation for arguments in definition.
    h. Enter line into MDT.
    i. If MACRO pseudo code
        i. MDLC<-MDLC+1
        ii. GOTO 5.f.
    j. Else If MEND pseudo code
        i. Then MDLC<-MDLC-1
            1. If MDLC=0 the
                a. Then GOTO 2.
            2. Else GOTO 5.f.
    k. Else GOTO 5.f.
6. Write into expanded source card file.
7. If END pseudo code
    a. Then Supply expanded source file to assembler processing.
8. Else GOTO 2.

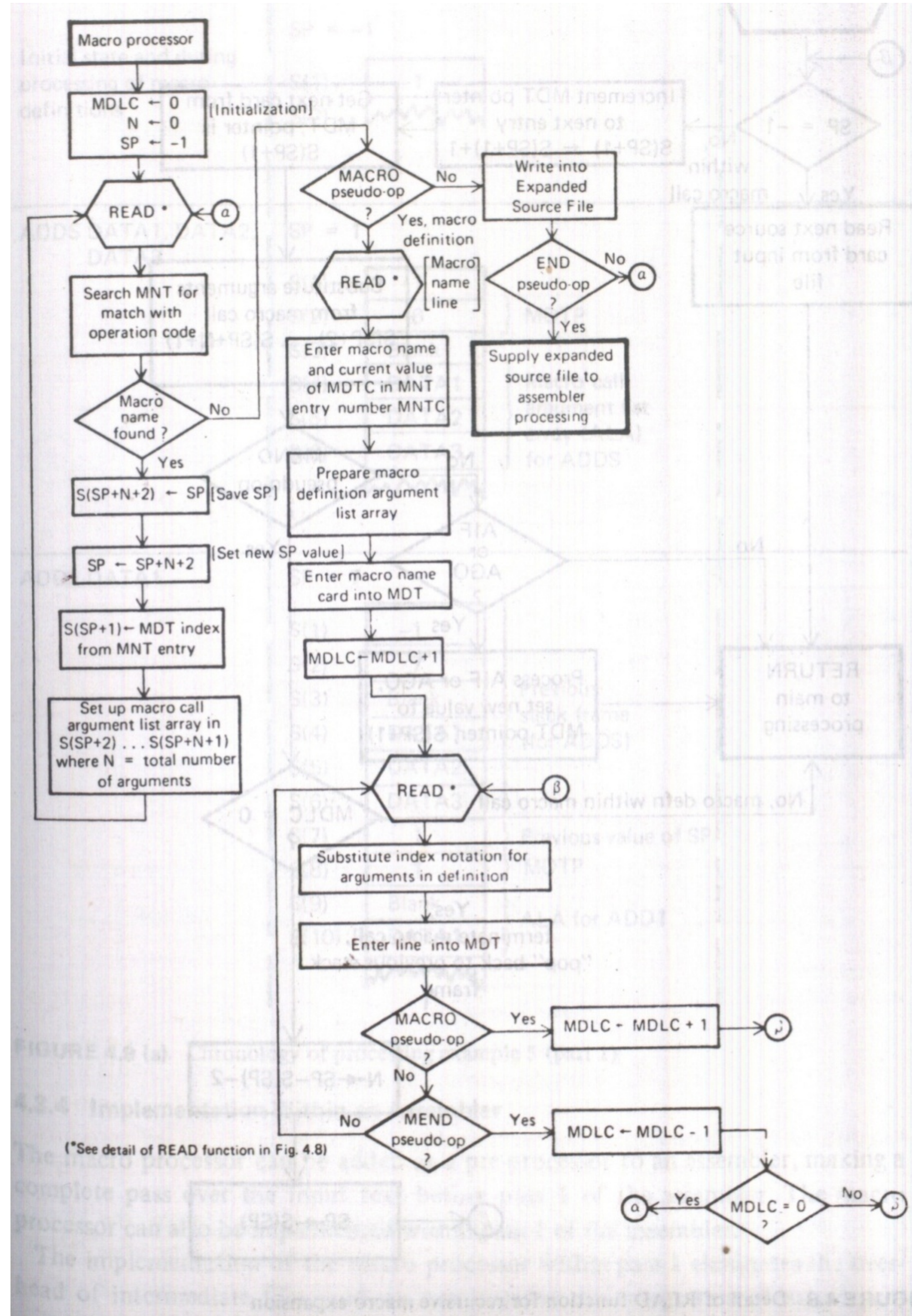

FIGURE 4.7 One-pass macro processor capable of handling macro calls within macro definitions

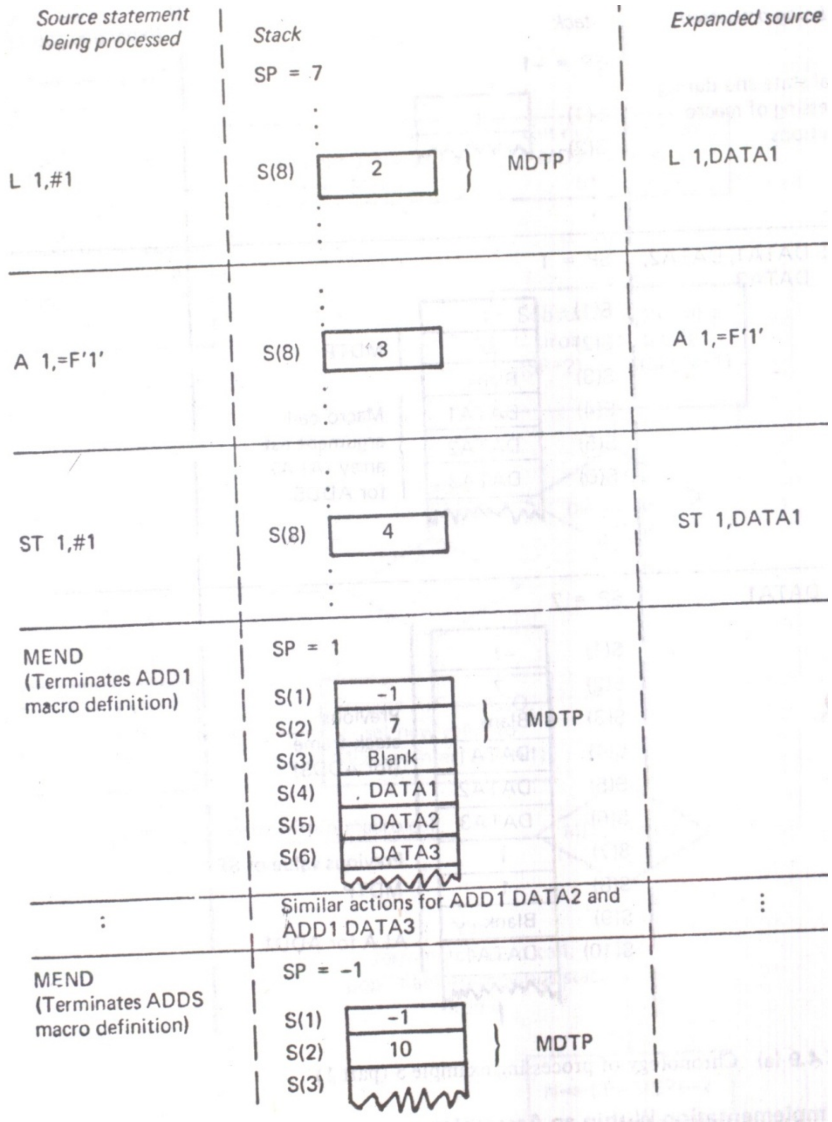

FIGURE 4.9 (b) Chronology of processing example 5 (part 2)

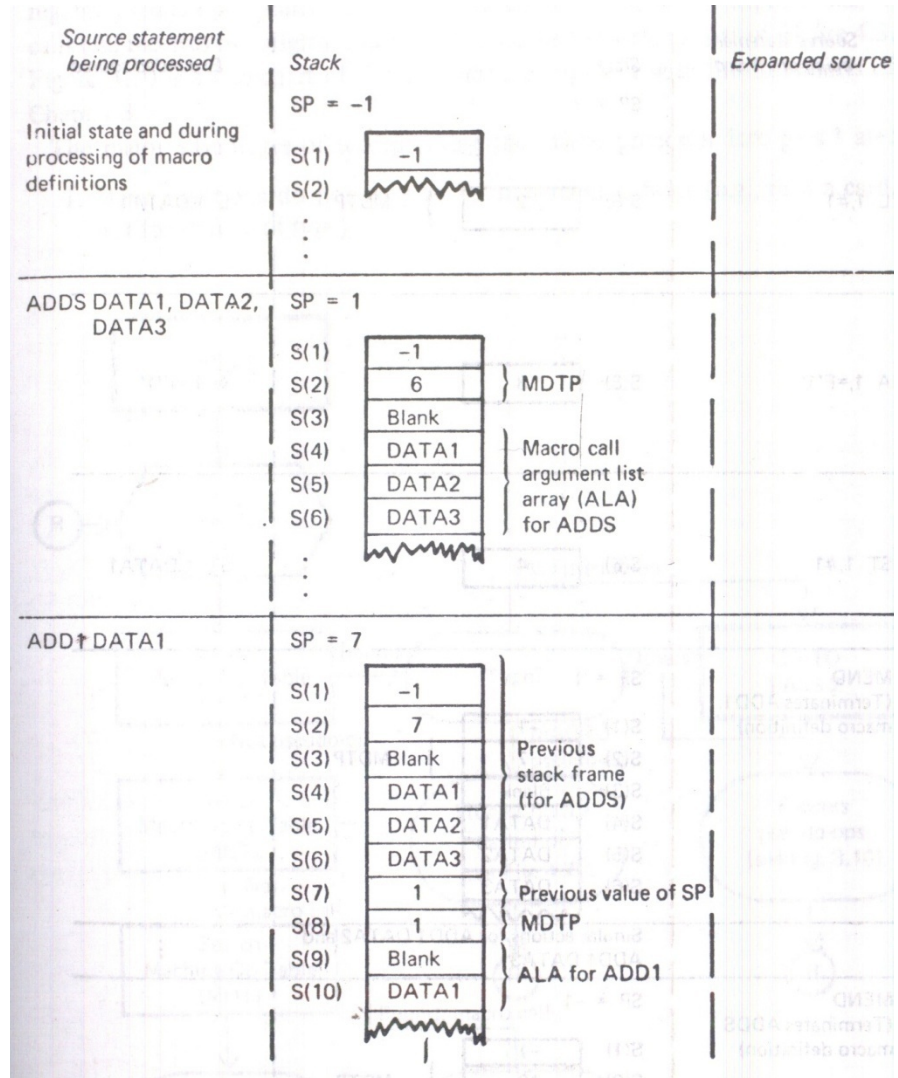

FIGURE 4.9 (a) Chronology of processing example 5 (part 1)

IMPLEMENTATION MACRO WITH IN ASSEMBLER

The macro processor can added as preprocessor to an assembler. The input of the macro processer serving as the input to the pass one of the assembler.

The other way of implanting a macro processor with an assembler is integrating macro processer with pass one of the assembler. This will remove the overheads associated with the input files that passed form macro processer to the assembler. This implantation will combine the similar functions of both like the MNT can be merged with the MOT or POT table with a flag stating that it is a macro. The read function will be similar to that of the macro processor with stack.

The advantages of this integration are as follows:

1. Many function need not be implanted twice i.e. read a card, test a statement etc.
2. There is less overhead, since there are no intermediate files needed over here.
3. More flexible since the user can make use of more features.

Disadvantages of this implementation are as follows:

1. The integration may be too complex and lengthy.
2. Problem with integration as two groups of persons for pass 1 of assembler and macro processor.

ALGOTITHM (Pass 1of Assembler)

1. READ (similar to read of macro processor with stack)
2. Search POT
3. If pseudo code found
    a. Then find type
        i. MACRO
            1. Then use macro defintion with stack.
            2. GOTO step 1.
        ii. Other
            1. Then similar task as that of pass 1 assembler.
            2. GOTO step 1.
        iii. END GOTO pass 2 of assembler.
4. Else search the MNT if found
    a. Set up macro call process
    b. GOTO step 1.
5. Else search MOT
6. Process machine instructions.
7. GOTO step 2.

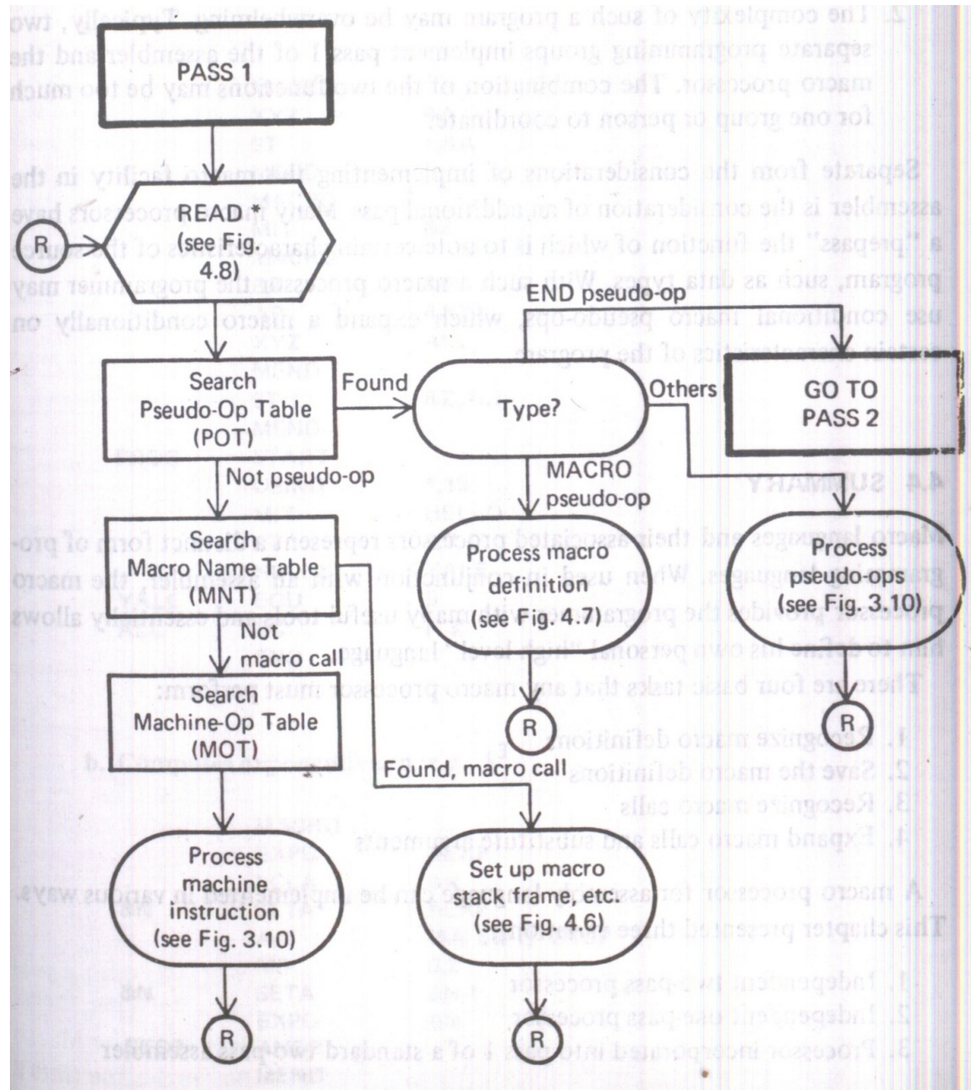

FIGURE 4.10 A macro processor combined with assembler pass 1