Assemblers

Introduction

Assembly language is a machine dependent, low level

Programming language which is specific to certain computer

Each statement is called instruction which is processed by assembler

Elements of assembly language programming

It provides 3 basic facilities which simplify programming

1.Mnemonic operation codes:

Mnemonic codes are easier to remember than numeric codes.

2.Symbolic operands:

Symbolic names can be used for data or instruction.so programmer no need to think about the numeric address of

operands

3.Data declaration:

Data can be declared in various notations

Statement format

[Label] <Opcode> <operand spec> [<operand spec>..]

Label and subsequent operands are optional

where the notation […] indicates the optional

<operand spec> is given by

<Symbolic name> [+ |- <displacement>] [(<index register>)]

A simple assembly language

Instruction opcode	*Assembly mnemonic*	*Remarks*
00	STOP	Stop execution
01	ADD	Perform addition
02	SUB	Perform subtraction
03	MULT	Perform multiplication
04	MOVER	Move from memory to register
05	MOVEM	Move from register to memory
06	COMP	Compare and set condition code
07	BC	Branch on condition
08	DIV	Perform division
09	READ	Read into register
10	PRINT	Print contents of register

Figure 3.1 Mnemonic operation codes

Comparison instruction sets a condition code after comparison

The condition code can be tested by BC instruction

BC <condition code spec>, <memory address>

Assembly language statements

It consists of three types of statements

1.Imperative statements

It indicates action to be performed

Ex: Arithmetic operations

Each imperative statement translates into one machine instruction

2.Declarative statement

It is of the following statement

[Label] DS <constant>

[Label] DC `<value>’

DS: Declare storage – reserves an area of memory and associates a symbolic name with it.

A DS 1

B DS 200

DC: Declare constant – it constructs memory words containing constants

C DC ‘10’

The DC actually will not create constant. It initializes

the memory with some value

Ex: MOVER BREG, C

Constants in assembly language

Two ways constant values can be used

Using immediate operand in instruction

Using literals

Use of immediate operand requires a special addressing mode

ADDI AREG,10

A literal is an operand with the syntax

='<value>'

ADD AREG, ='5'

3.Assembler Directives

It instructs the assembler to perform certain actions while assembling the program.

Two common directives: START and STOP

START <constant>

END [<Operand spec>]

Benefits of assembly language

Use of symbolic operand specification

It provides the ability to use special features in the architecture of computer. Example: Special instruction supported by CPU

Pass structure of Assemblers

1.A pass in a language processor performs language

processing functions on every statement in a source program or in its equivalent representation.

2.A language processor may make multiple passes over a source program for handling forward reference and reduce memory requirement.

3.Multiple passes makes the language processor slower, so the design should include only fewer passes.

Two Pass Translation

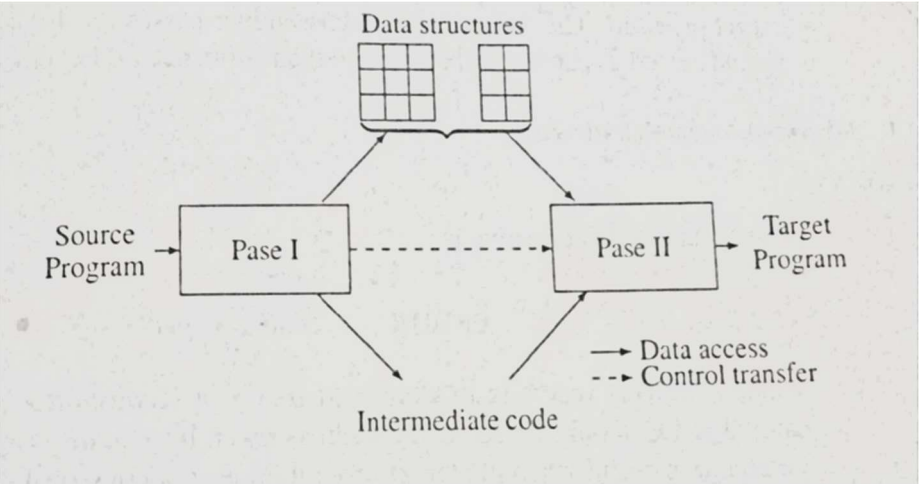


Fig. overview of two pass assembly

1.The first pass constructs intermediate code from the source program which is used by second pass

2.The intermediate representation consists of two main components

A data structure representation called symbol table

Processed form of source program

3.The second pass creates the target program by using the address information found in the symbol table.

Pass 1 – Analysis of source program

Pass 2 – Synthesis of target program

Single pass Translation

1.Location counter-(Memory allocation of data structure)

processing and symbol table construction is done in the pass one.

2.In addition the target program also constructed in the same pass

3.The problem of forward reference is tackled by back patching.

4.In back patching the assembler leaves the operand field of an instruction blank if that instruction contains a forward reference to a symbol.

Implementation of backpatching

1.The assembler builds a table of incomplete instructions (TII)

to record information about instructions whose operands fields were left blank

2.Each entry in TII is of the pair (instruction address, symbol) to indicate that the address of the symbol should be put in the operand field of the instruction with address instruction address.

3.By the time the END statement is processed, the symbol table would contain the addresses of all symbols from source

program and the TII would contain information describing all forward references.

4.The assembler now process each entry in TII to complete the concerned instruction

Design of two pass assembler

1.The tasks performed by two-pass assembler are,

Pass 1:

Separation of symbol, opcode and operand

Build the symbol table

Perform LC processing

Construct intermediate representation

Pass II

Synthesis the target program

Advanced Assembler Directives

ORIGIN

The syntax of this is

ORIGIN <Address Specification>

<Address Specification> is an <Operand Specification>

Or <Constant>

This directive instruct the assembler to put the address given by <Address Specification> in the LC (Location Counter)

EQU

The syntax of this is

<symbol> EQU <Address Specification>

<Address Specification> is either a <constant> or a <symbolic name> with <displacement>

The EQU directive simply associates the name <symbol> with the address specified by the <Address Specification>

The address of LC is not affected

LTORG

The LTORG stands for Origin for Literals, used to specify where the literals are placed

When there is a literal in a statement, then assembler enters it into literal pool unless a matching literal already exists in the pool.

At every LTORG/END statement the assembler allocates memory to the literals of the literal pool and clears the literal pool

Pass 1 of the Assembler

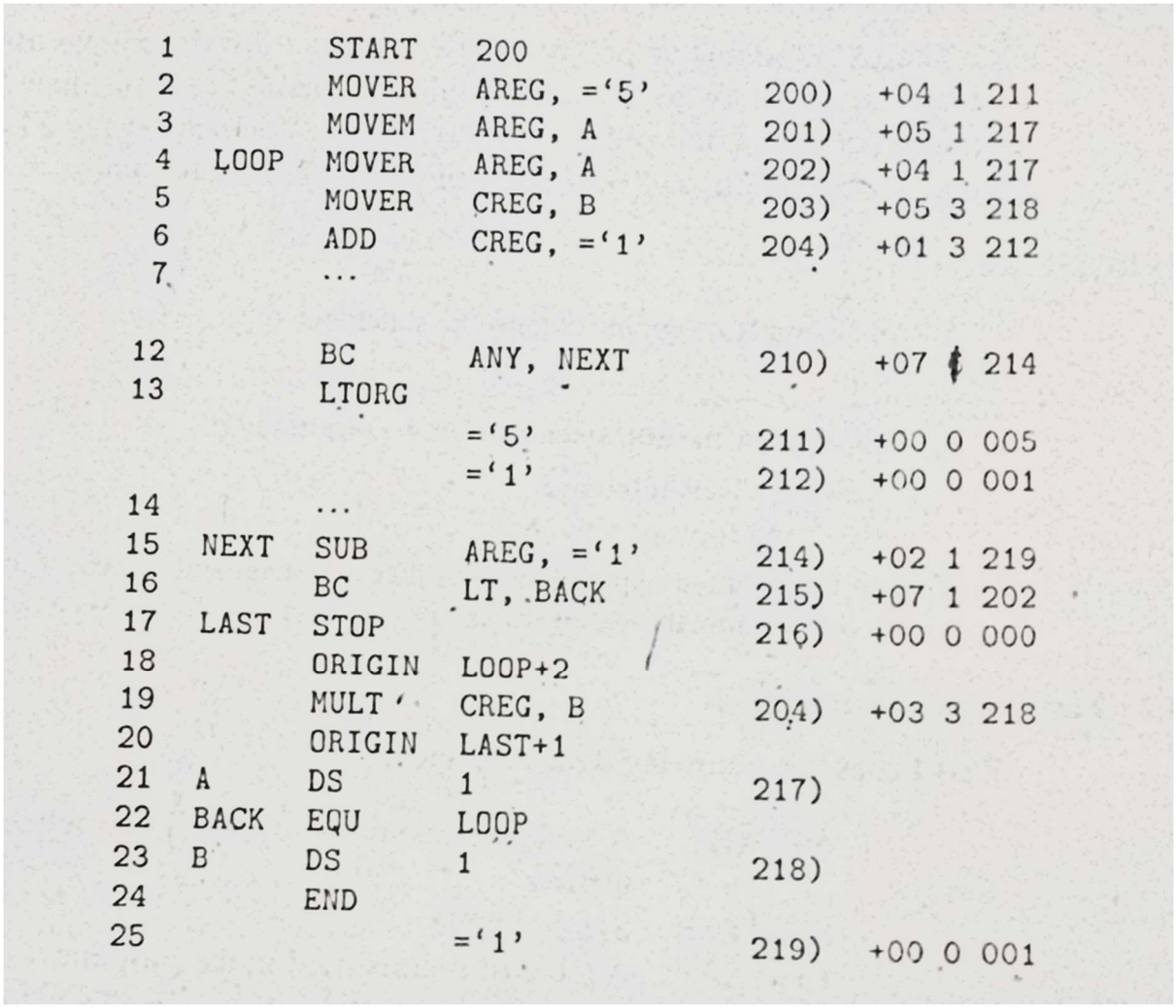
Pass 1 uses the following data structures

OPTAB – a table of mnemonic opcodes and their information

SYMTAB- Symbol Table

LITTAB – a table of literals used in the program

POOLTAB – a table of information concerning literal pool



```
 1          START   200
 2          MOVER   AREG, ='5'      200)  +04 1 211
 3          MOVEM   AREG, A         201)  +05 1 217
 4  LOOP    MOVER   AREG, A         202)  +04 1 217
 5          MOVER   CREG, B         203)  +05 3 218
 6          ADD     CREG, ='1'      204)  +01 3 212
 7          ...

12          BC      ANY, NEXT       210)  +07 6 214
13          LTORG
                    ='5'            211)  +00 0 005
                    ='1'            212)  +00 0 001
14          ...
15  NEXT    SUB     AREG, ='1'      214)  +02 1 219
16          BC      LT, BACK        215)  +07 1 202
17  LAST    STOP                    216)  +00 0 000
18          ORIGIN  LOOP+2
19          MULT    CREG, B         204)  +03 3 218
20          ORIGIN  LAST+1
21  A       DS      1               217)
22  BACK    EQU     LOOP
23  B       DS      1               218)
24          END
25                  ='1'            219)  +00 0 001
```

Fig. An Assembly program

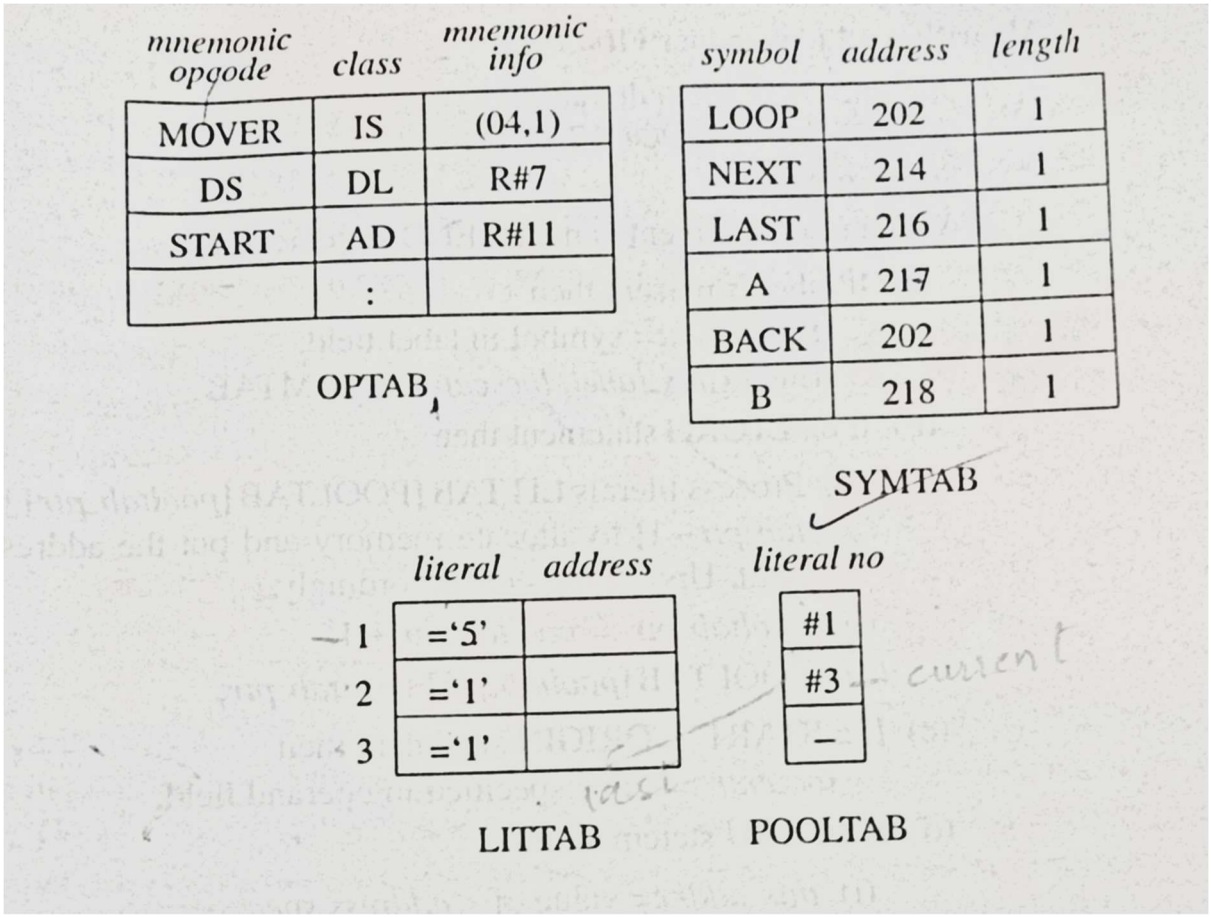


Fig. Data structures of assembler pass 1

Pass 1 Algorithm

Pass 1 algorithm uses the following data structures

LC : Location counter

littab_ptr: Points to an entry in LITTAB

Pooltab_ptr: Points to an entry in POOLTAB

1. *loc_cntr* := 0; (default value)
   pooltab_ptr := 1; POOLTAB[1] := 1;
   littab_ptr := 1;
2. While next statement is not an END statement
   (a) If label is present then
       this_label := symbol in label field;
       Enter (*this_label*, *loc_cntr*) in SYMTAB.
   (b) If an LTORG statement then
       (i) Process literals LITTAB[POOLTAB[*pooltab_ptr*]] ... LITTAB[*littab_ptr*−1] to allocate memory and put the address in the *address* field. Update *loc_cntr* accordingly.
       (ii) *pooltab_ptr* := *pooltab_ptr* + 1;
       (iii) POOLTAB[*pooltab_ptr*] := *littab_ptr*;
   (c) If a START or ORIGIN statement then
       loc_cntr := value specified in operand field;
   (d) If an EQU statement then
       (i) *this_addr* := value of <*address spec*>;
       (ii) Correct the symtab entry for *this_label* to (*this_label*, *this_addr*).
   (e) If a declaration statement then
       (i) *code* := code of the declaration statement;
       (ii) *size* := size of memory area required by DC/DS.
       (iii) *loc_cntr* := *loc_cntr* + *size*;
       (iv) Generate IC '(DL, *code*) ...'.
   (f) If an imperative statement then
       (i) *code* := machine opcode from OPTAB;
       (ii) *loc_cntr* := *loc_cntr* + instruction length from OPTAB;
       (iii) If operand is a literal then
           this_literal := literal in operand field;
           LITTAB[*littab_ptr*] := *this_literal*;
           littab_ptr := *littab_ptr* + 1;
       else (i.e. operand is a symbol)
           this_entry := SYMTAB entry number of operand;
           Generate IC '(IS, *code*)(S, *this_entry*)';
3. (Processing of END statement)
   (a) Perform step 2(b).
   (b) Generate IC '(AD,02)'.
   (c) Go to Pass II.

Intermediate Code Forms

The intermediate consists of a sequence of intermediate code units (IC units)

Each IC unit consists of the following

Address

Representation of mnemonic code

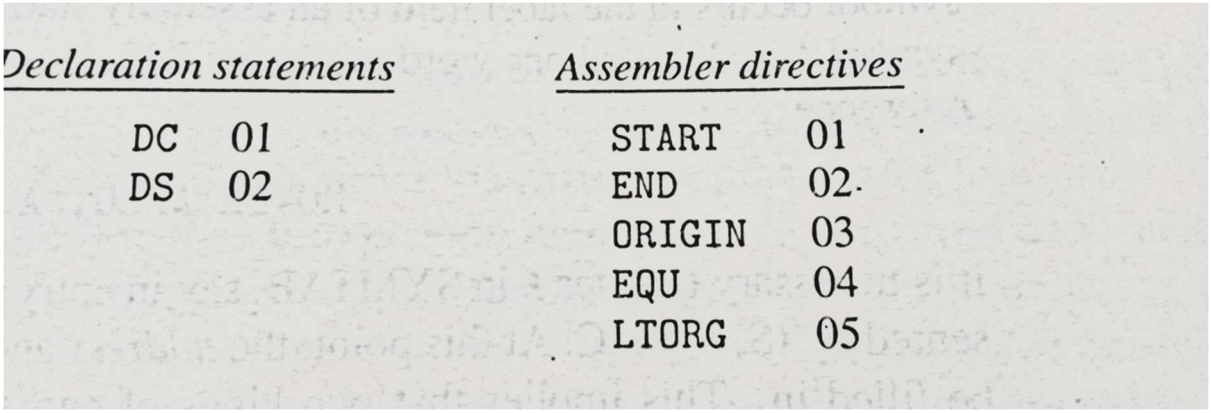
Representation of operands

Address	Opcode	Operands

The mnemonic opcode field contains a pair of the form

(Statement class, code)

Statement class can be IS, DL or AD a code is the instruction opcode in the machine language



Declaration statements

DC	01
DS	02

Assembler directives

START	01
END	02
ORIGIN	03
EQU	04
LTORG	05

Intermediate code for IS

In two ways intermediate code is represented for imperative statements

Way 1

For first operand of IS, the codes can be used as 1 to 4 for indicating CPU registers

The second operand is a memory operand, which can be represented as

(Operand class, code)

Operand class is either

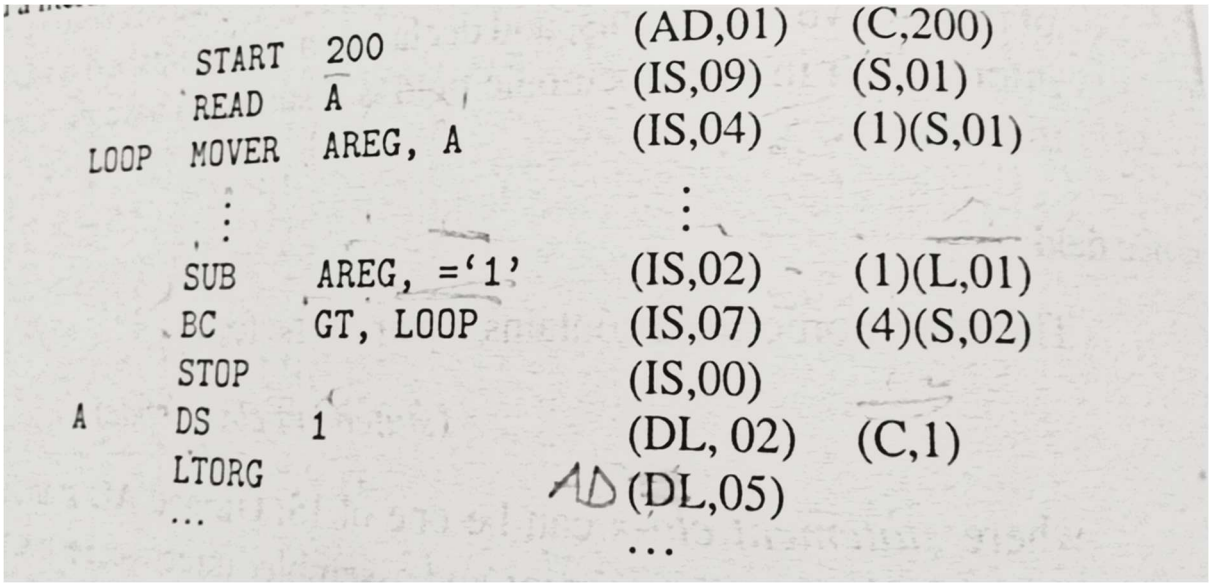
C – Constant

L – Literal

S – Symbol

For constants the code field contain the constant value

For literal and symbol, the literal/symbol table serial no is used as code value



```
        START   200             (AD,01)   (C,200)
        READ    A               (IS,09)   (S,01)
LOOP    MOVER   AREG, A         (IS,04)   (1)(S,01)
          :                       :
        SUB     AREG, ='1'      (IS,02)   (1)(L,01)
        BC      GT, LOOP        (IS,07)   (4)(S,02)
        STOP                    (IS,00)
A       DS      1               (DL, 02)  (C,1)
        LTORG                   (DL,05)
        ...                     ...
```

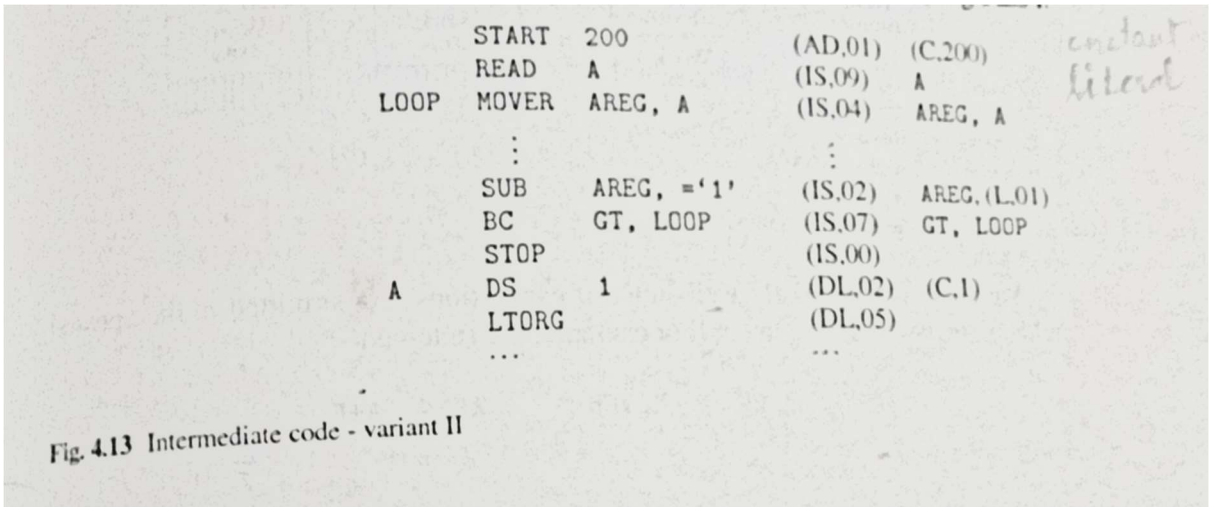
Fig Intermediate code variant I

Way 2

In this method the operand field of the IS is either in processed form or in the source form itself

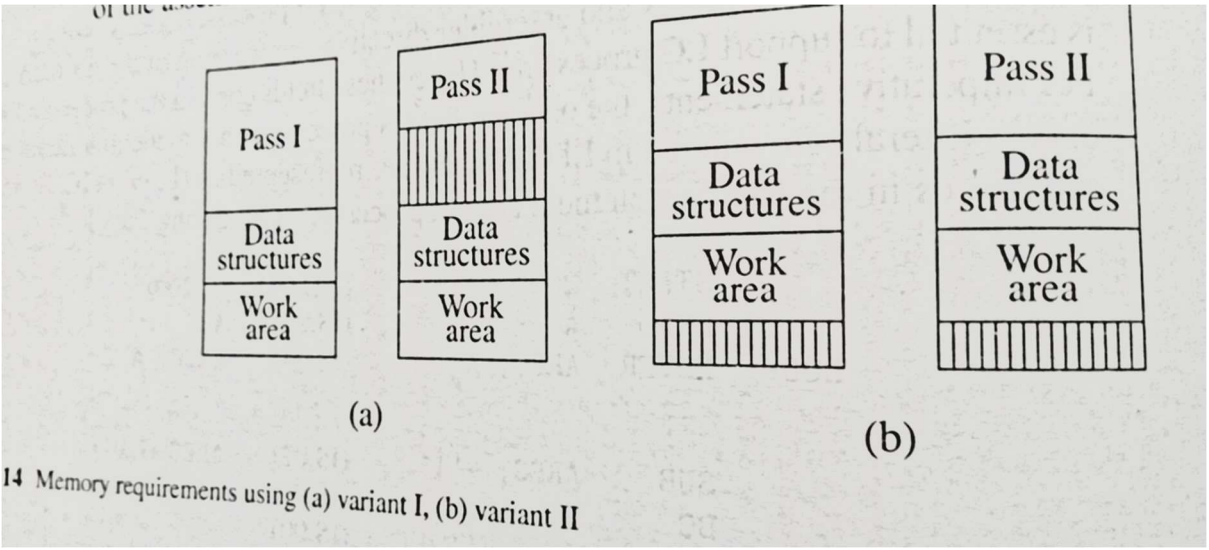
If there is any symbolic references in the operand field, then they can be placed in the source form itself

In this case the processing of these statements are done in Pass II



```
        START   200             (AD,01)   (C,200)
        READ    A               (IS,09)   A
LOOP    MOVER   AREG, A         (IS,04)   AREG, A
          :                       :
        SUB     AREG, ='1'      (IS,02)   AREG, (L,01)
        BC      GT, LOOP        (IS,07)   GT, LOOP
        STOP                    (IS,00)
A       DS      1               (DL,02)   (C,1)
        LTORG                   (DL,05)
        ...                     ...
```

Fig. 4.13 Intermediate code - variant II



14 Memory requirements using (a) variant I, (b) variant II

The second variant (way2) is normally used in the instruction that contains expressions in the operand field

Ex: MOVER AREG, A+5

If variant 1 (way1) is used then the processed form of the statement is

(IS, 05) (1) (S, 01)+5

This will not simplify the task of Pass II and memory requirement

So variant2 (way2) is better

Processing of Declarations and AD

Consider the following

```
        START   200          —)   (AD,01)   (C,200)
AREA1   DS      20     ⇒    200) (DL,02)   (C,20)
SIZE    DC      5           220) (DL,01)   (C,5)
```

It is not needed to have the representation for START and DS in intermediate code, since DC statement have the necessary address 220

The DC (Declare Constant) statement must be represented in the intermediate code.

If a DC statement is in the following form

DC ‘5,3,-7’

Then many (DL,01) units can be used

The START and ORGIN directives set the new value into LC. So it is not necessary to retain these statements in the intermediate code, if the intermediate code contains an address field

LTORG must be processed

Pass II of the assembler

The important data structures used by the PASS II are

SYMTAB, LITTAB and POOLTAB

LC	: Location counter
Littab_ptr	: Points to an entry in LITTAB
Pooltab_ptr	: Points to an entry in POOLTAB
Machine_code_buffer	: Area for constructing code for one statement
Code_area	: Area for assembling the target progam
Code_area_address	: Contains address of code areas

Algorithm for Pass II

Pass II of the Assembler

Algorithm 4.2 is the algorithm for assembler Pass II. Minor changes may be needed to suit the IC being used. It has been assumed that the target code is to be assembled in the area named *code_area*.

Algorithm 4.2 (Assembler Second Pass)

1. *code_area_address* := address of *code_area*;
   pooltab_ptr := 1;
   loc_cntr := 0;
2. While next statement is not an END statement
   (a) Clear *machine_code_buffer*;
   (b) If an LTORG statement
      (i) Process literals in LITTAB [POOLTAB [*pooltab_ptr*]] ... LITTAB [POOLTAB [*pooltab_ptr*+1]]−1 similar to processing of constants in a DC statement, i.e. assemble the literals in *machine_code_buffer*.
      (ii) *size* := size of memory area required for literals;
      (iii) *pooltab_ptr* := *pooltab_ptr* + 1;
   (c) If a START or ORIGIN statement then
      (i) *loc_cntr* := value specified in operand field;
      (ii) *size* := 0;
   (d) If a declaration statement
      (i) If a DC statement then
         Assemble the constant in *machine_code_buffer*.
      (ii) *size* := size of memory area required by DC/DS;
   (e) If an imperative statement
      (i) Get operand address from SYMTAB or LITTAB.
      (ii) Assemble instruction in *machine_code_buffer*.
      (iii) *size* := size of instruction;
   (f) If *size* ≠ 0 then
      (i) Move contents of *machine_code_buffer* to the address *code_area_address* + *loc_cntr*;
      (ii) *loc_cntr* := *loc_cntr* + *size*;
3. (Processing of END statement)
   (a) Perform steps 2(b) and 2(f).
   (b) Write *code_area* into output file.

Program listing & error reporting

The assembler produces a program listing that shows a source statement and the target code

The listing also represents the errors in the source program

Some errors like syntax errors are identified in PASS I and some other kinds are identified in PASS II.

Macros and macro pre-processor

Introduction

- A macro is a facility for extending a programming language
- A macro either defines a new operation or a new method for declaring data
- The language processor replaces a call on a macro by a sequence of statements that implements the defined operation or the methos of declaring data
- Many languages that support macro C, C++

Two kinds of macro expansion

- Lexical substitution: It replaces the formal parameters with actual parameters
- Semantic Expansion: Generation of statements that are tailored to the requirements of a specific macro call

Macro Definition and call

A macro definition consists of

Macro name

Set of formal parameters

Set of statement for defining the macro-operation

Use of macro name in the assembly language instruction is called macro call

The operand field of macro call can include the values called actual parameters

The language processor replaces a macro call by its definition- macro expansion

- Macro definition appears in the starting of the program
- Each macro is enclosed by a header (MACRO) and end (MEND) statements.
- Macro can use formal parameter along with the symbol &
- A macro definition can have 3 kinds of statements

1.Macro Prototype

It declares macro name and formal parameters

2.Model Statement

Using this assembly language statement may be generated during macro expansion

3.Macro Pre-processor statement

Used to perform auxiliary functions during macro expansion

The macro prototype has the following syntax:

<Macro name>[<formal parameter spec.> […]]

Macro name appears in the assembly language instruction as opcode, formal parameters appear in the operand field

A <formal parameter spec,> is of the form

&<parameter name> [<parameter kind>]

A parameter may either positional parameter or keyword parameter

By default the parameters are positional parameters and they doesn't include parameter kind.

A macro call has the following syntax,

<Macro name> [<Actual parameter spec, > […]]

A macro definition for increment operation

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

Corresponding Macro call

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

Macro Expansion

Macro expansion can be performed by two kinds of language processor

1.macro assembler

It performs expansion of each macro call in a program into a sequence of assembly language statements and also assembles the resultant assembly language program

2.macro pre-processor

It only processes the macro call. Other statements are processed with the help of assembler.

There are two key notions used in implementing macro expansion

1.Flow of control during expansion

2.Lexical substitution

Flow of control during expansion

Default flow of control during macro expansion in sequential

A pre-processor statement can alter the flow of control, so that some statements from macro may not visited or some may be executed repeatedly

Ex: use of conditional expansion or expansion time loops

The flow control of macro expansion is implemented by

MEC(Macro Expansion Counter)

Lexical substitution

A model statement may use the three types of strings

1.An ordinary string: which is any string other than type 2 or type 3

2. The name of formal parameter preceded by &

3.The name of pre-processor variable preceded by &

During lexical substitution ordinary string is retained in its original form.

The name of formal parameter is replaced by its value from the actual parameter of a macro call

The name of the pre-processor variable is replaced with its value. This value is readily known to the pre-processor

Positional parameters

The formal parameter without <parameter kind> is called positional parameter.

Example:

consider the macro

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

Macro call is: INCR A, B, AREG

Keyword parameter:

These formal parameters include <parameter kind> which is actually the string '='

Consider the macro definition

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

The macro calls for the above definition is

```
    INCR_M      MEM_VAL=A,  INC_VAL=B,     REG=AREG
    INCR_M      INCR_VAL=B,   REG=AREG, MEM_VAL=A
```

Specifying default values of parameters

The default values for the keyword parameters can be specified as

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

Macro definition

```
MACRO
INCR_D      &MEM_VAL=, &INCR_VAL=,  &REG=AREG
```

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

Macro call

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

Macro with mixed parameter list

Here a macro can use both positional and keyword parameters

```
SUM A, B,    C=10, D=X
```

Nested macro calls

A model statement in a macro constitute a call to another macro

The macro that contains the nested call is called outer

macro and the macro that is called in the nested call is inner macro.

The pre-processor performs expansion of nested macro calls

LIFO order

Example

```
MACRO
COMPUTE    &FIRST,    &SECOND
```

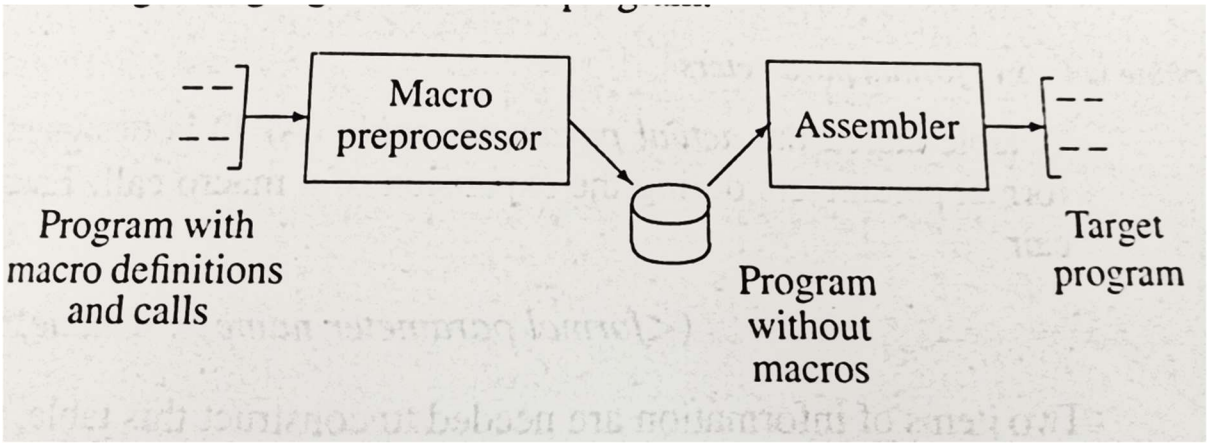
```
MOVEM     BREG,    TMP
INCR_D    &FIRST,  &SECOND,    REG=BREG
MOVER     BREG,    TMP
MEND

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

For this macro call, COMPUTE X, Y

Design of macro-preprocessor

The macro pre-processor accepts an assembly program containing definitions and calls and translates it into an assembly program which does not contain any macro definition or calls.



Design overview

The following tasks in macro expansion

1.Recognizing a macro call

2.Deciding what values formal parameters should have in the expansion of the macro call

3.Maintaining values of expansion time variables during the expansion of the macro call

4.Organising expansion time control flow

5.Finding the statement that defines a specific sequencing symbol

6.Performing expansion of a model statement

<u>Recognizing a macro call</u>

Macro call is identified using the mnemonic field of the statement

A Macro Name Table (MNT) used to store all macro names defined in the program

Determining values of formal parameter:

The value of formal parameter is the corresponding actual parameter from a macro call or default value specified in the macro definition.

The pre-processor needs to know the names of formal parameters and the default values of keyword parameter

A table Parameter Default Table (PDT), that stores formal parameter name and its default value.

(<formal parameter name> <default name>)

Once formal parameters are identified whose values are maintained in Actual Parameter Table (APT).

(<formal parameter name> <value>)

Maintaining values of expansion time variables:

During a pre-processor statement or a model statement expansion, expansion variable values are needed

A table Expansion time Variable Table (EVT) is used to store the values of expansion time variables.

(<expansion time variable name> <value>)

Organising expansion time control flow:

The flow of control determines whether a model statement or a pre-processor statement of macro definition is visited for expansion

To manage flow of control MEC (Macro Expansion Counter) is used.

The body of macro code which contains model statements and pre-processor statements are stored in a table called MDT (Macro Definition Table)

Each entry in MDT is for one statement in the Macro definition

The MNT entry points to the first statement of its corresponding MDT, MEC also points to the MDT

Finding the model statement that defines a sequencing symbol

The pre-processor will identify this while processing a macro definition and store it in a table called SST (Sequencing Symbol Table)

It is used during macro expansion

Each entry in this table is a pair of the following form

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

Performing expansion of a model statement

It involves a lexical substitution for the parameters and expansion time variables used in the model statements

1.The macro expansion counter (MEC) points to the MDT entry that contains the model statement

2.Values of formal parameters and expansion time variables are available in the actual parameter table and the expansion time variables table respectively

3.The model statement that defines a sequencing symbol can be found by using the sequencing symbol table

Data structures of a macro pre-processor

The APT is split into two tables

PNTAB (Parameter Name Table) which stores the names of all formal parameters.

APTAB (Actual Parameter Table) which stores the values for formal parameters

PNTAB is used during the process of macro definition and APTAB is used during macro expansion

Similarly EVT is split into EVNTAB & EVTAB and SST is split into SSNTAB & SSTAB

The PDT is replaced by KPDTAB (Keyword Parameter Default table)

Below diagram shows the tables of macro preprocessor

Table	*Fields in each entry*
Macro name table (MNT)	*Macro name*, *Number of positional parameters* (#PP), *Number of keyword parameters* (#KP), *Number of expansion time variables* (#EV), *MDT pointer* (MDTP), *KPDTAB pointer* (KPDTP), *SSTAB pointer* (SSTP)
Parameter Name Table (PNTAB)	*Parameter name*
EV Name Table (EVNTAB)	*EV name*
SS Name Table (SSNTAB)	*SS name*
Keyword Parameter Default Table (KPDTAB)	*parameter name*, *default value*
Macro Definition Table (MDT)	*Label*, *Opcode*, *Operands*
Actual Parameter Table (APTAB)	*Value*
EV Table (EVTAB)	*Value*
SS Table (SSTAB)	*MDT entry #*

Consider a macro definition and call

```
        MACRO
        CLEARMEM    &X, &N, &REG=AREG
        LCL         &M
&M      SET         0
        MOVER       &REG, = ‘0’
.MORE   MOVEM       &REG, &X+&M
&M      SET         &M+1
        AIF         (&M NE N).MORE
        MEND
```

Macro call

```
        CLEARMEM    AREA,10
```

Data structures

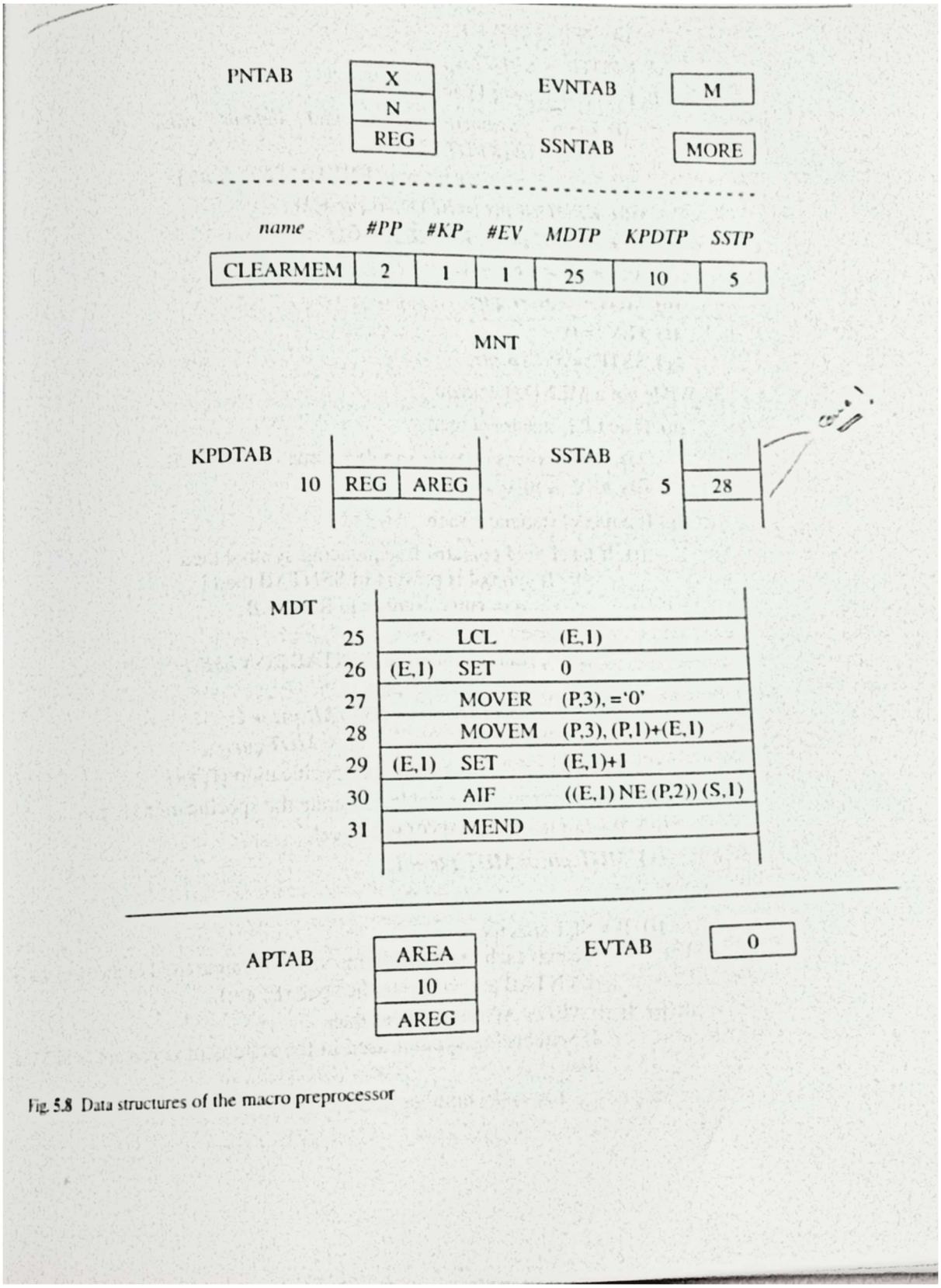


Fig. 5.8 Data structures of the macro preprocessor

Processing of macro definition(Algorithm)

The pre-processor uses three global variables and which are initialized as

KPDTAB_ptr =1;

SSTAB_ptr =1;

MDT_ptr =1;

Algorithm 5.2 (Processing of a macro definition)

1. *SSNTAB_ptr* := 1;
   PNTAB_ptr := 1;
2. Process the macro prototype statement and form the MNT entry
   (a) *name* := macro name;
   (b) For each positional parameter
      (i) Enter *parameter name* in PNTAB [*PNTAB_ptr*].
      (ii) *PNTAB_ptr* := *PNTAB_ptr* + 1;

(iii) #PP := #PP + 1;

(c) KPDTP := *KPDTAB_ptr*;

(d) For each keyword parameter

(i) Enter *parameter name* and *default value* (if any), in KPDTAB[*KPDTAB_ptr*].

(ii) Enter *parameter name* in PNTAB[*PNTAB_ptr*].

(iii) *KPDTAB_ptr* := *KPDTAB_ptr* + 1;

(iv) *PNTAB_ptr* := *PNTAB_ptr* + 1;

(v) #KP := #KP + 1;

(e) MDTP := *MDT_ptr*;

(f) #EV := 0;

(g) SSTP := *SSTAB_ptr*;

3. While not a MEND statement

(a) If an LCL statement then

(i) Enter expansion time variable name in EVNTAB.

(ii) #EV := #EV + 1;

(b) If a model statement then

(i) If label field contains a sequencing symbol then

If symbol is present in SSNTAB then

q := entry number in SSNTAB;

else

Enter symbol in SSNTAB[*SSNTAB_ptr*].

q := *SSNTAB_ptr*;

SSNTAB_ptr := *SSNTAB_ptr* + 1;

SSTAB[SSTP + *q* − 1] := *MDT_ptr*;

(ii) For a parameter, generate the specification (P, #*n*).

(iii) For an expansion variable, generate the specification (E, #*m*).

(iv) Record the IC in MDT[*MDT_ptr*];

(v) *MDT_ptr* := *MDT_ptr* + 1;

(c) If a preprocessor statement then

(i) If a SET statement

Search each expansion time variable name used in the statement in EVNTAB and generate the spec (E, #*m*).

(ii) If an AIF or AGO statement then

If sequencing symbol used in the statement is present in SSNTAB then

q := entry number in SSNTAB;

else
Enter symbol in SSNTAB[*SSNTAB_ptr*].
q := *SSNTAB_ptr*;
SSNTAB_ptr := *SSNTAB_ptr* + 1;
Replace the symbol by (S, SSTP + *q* − 1).
(iii) Record the IC in MDT [*MDT_ptr*].
(iv) *MDT_ptr* := *MDT_ptr* + 1;

4. (MEND statement)
If *SSNTAB_ptr* = 1 (i.e. SSNTAB is empty) then
SSTP := 0;
Else *SSTAB_ptr* := *SSTAB_ptr* + *SSNTAB_ptr* − 1;
If #KP = 0 then KPDTP = 0;

Macro expansion(Algorithm)

The following data structures are used for expansion

APTAB	Actual parameter table
EVTAB	Expansion variables table
MEC	Macro expansion counter
APTAB_ptr	APTAB pointer
EVTAB_ptr	EVTAB pointer

Algorithm 5.3 (Macro expansion)

1. Perform initializations for the expansion of a macro
   (a) MEC := MDTP field of the MNT entry;
   (b) Create EVTAB with #EV entries and set *EVTAB_ptr*.
   (c) Create APTAB with #PP+#KP entries and set *APTAB_ptr*.
   (d) Copy keyword parameter defaults from the entries KPDTAB [KPDTP] ... KPDTAB[KPDTP+#KP−1] into APTAB[#PP+1] ... APTAB[#PP+#KP].
   (e) Process positional paremeters in the actual parameter list and copy them into APTAB[1] ... APTAB[#PP].

(f) For keyword parameters in the actual parameter list
Search the keyword name in *parameter name* field of KPDTAB[KPDTP] ... KPDTAB[KPDTP+#KP−1]. Let KPTDAB[*q*] contain a matching entry. Enter value of the keyword parameter in the call (if any) in APTAB[#PP+*q*−KPDTP+1].

2. While statement pointed by MEC is not MEND statement
   (a) If a model statement then
      (i) Replace operands of the form (P, #*n*) and (E, #*m*) by values in APTAB[*n*] and EVTAB[*m*] respectively.
      (ii) Output the generated statement.
      (iii) MEC := MEC + 1;
   (b) If a SET statement with the specification (E, #*m*) in the label field then
      (i) Evaluate the expression in the operand field and set an appropriate value in EVTAB[*m*].
      (ii) MEC := MEC + 1;
   (c) If an AGO statement with (S, #*s*) in operand field then
      MEC := SSTAB[SSTP+*s* − 1];
   (d) If an AIF statement with (S, #*s*) in operand field then
      If condition in the AIF statement is *true* then
         MEC := SSTAB[SSTP+*s* − 1];
3. Exit from macro expansion.

Figure 5.9 shows the data structures at some point during the expansion of macro CLEARMEM of Ex. 5.16.

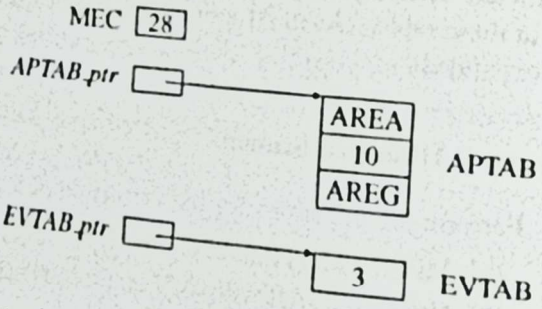


Fig. 5.9 Data structures during macro expansion

5.5.5 Nested Macro Calls

Figure 5.5 in Section 5.3 illustrates nested macro calls. Two basic alternatives exist for processing nested macro calls. We can apply the macro expansion schematic of

Design of a macro assembler

Pass I:

- Macro definition processing
- Entering of names and types of symbols in SYMBOL

Pass II:

- Macro Expansion
- Memory allocation and LC processing
- Processing of literals
- Intermediate code generation

Pass III:

- Target code generation