MACROS AND MACRO PROCESSORS

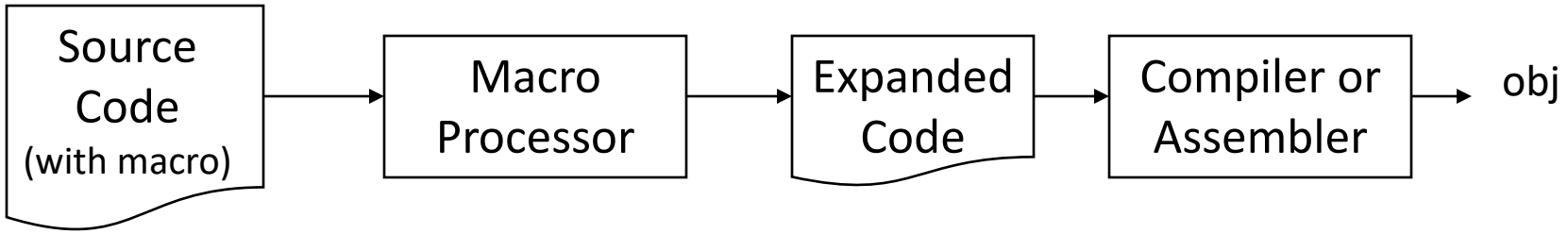
INTRODUCTION

→Macro allows programmer to write short hand programs (modular programming).

→The macro processor replaces each macro instruction with its equivalent block of instructions.

Macro Processor

- Recognize macro definitions
- Save the macro definition
- Recognize macro calls
- Expand macro calls



MACROS :

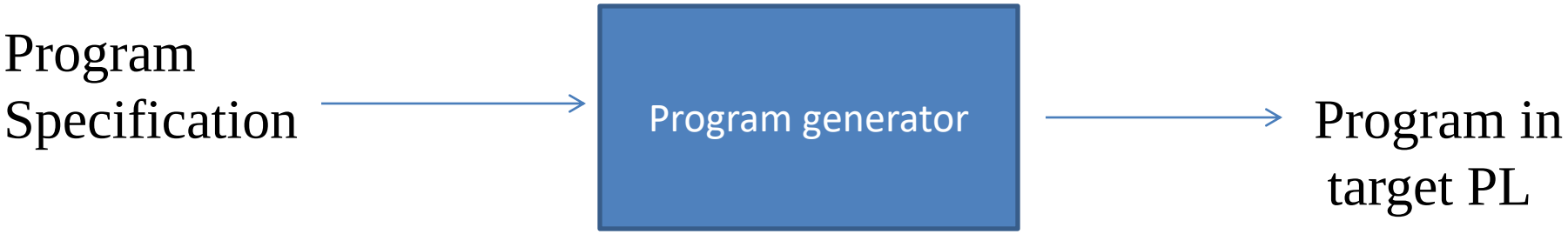
→A Macro is an extension to the basic ASSEMBLER language.

→They provide a means for generating a commonly used sequence of assembler instructions/statements.

→The sequence of instructions/statements will be coded ONE time within the macro definition. Whenever the sequence is needed within a program, the macro will be "called".

→Macros are used to provide a program generation facility through macro expansion.

→The program generator is a software system which accepts the specification of a program to be generated, and generates a program in the target PL.



→Many languages provide built in facilities for writing macros.

→Well known examples are the higher level languages PL/I, C, Ada and C++.

→Assembly languages of most computer systems also provide such facilities.

→When a language does not support built in macros facilities, a programmer may achieve an equivalent effect by using generalized preprocessors or software tools like Awk of Unix.

Standard Definitation:

→A macro is a unit of specification for program generation through expansion.

→A macro consists of a name, a set of formal parameters and a body of code.

→The use of macro name with a set of actual parameters is replaced by some code generated by its body.

This is called **Macro Expansion**.

There are two kinds of expansions:
1) **Lexical expansion**
2) **Semantic expansion**

1) **Lexical expansion**: lexical expansion implies replacement of a character string by another character string during program execution.

→It is typically employed to replace occurrences of formal parameters by corresponding actual parameters.

2) **Semantic expansion**: semantic expansion implies generation of instructions tailored to the requirements of a specific usage

→It 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.

Example:

→The following sequence of instructions is used to increment the value in a memory word by a constant :
1) Move the value from the memory word into the machine register.
2) Increment the value into the machine register.
3) Move the new value into the memory word.

→Since the instruction sequence MOVE-ADD-MOVE may be used a number times in a program, it is convenient to define a macro named INCR.

→ Using lexical expansion the macro call INCR A, B, AREG can lead to the generation of a MOVE-ADD-MOVE instruction sequence to increment A by the value of B using AREG to perform the arithmetic.

→ Use of semantic expansion can enable the instruction sequence to be adapted to the types of A and B.
for eg: for Intel 8088, an INC instruction could be generated if A is a byte operand and B has the value ‘1’, while a MOVE-ADD-MOVE sequence can be generated in all other situations.

Macro vs. Subroutine

→ The macros differ from subroutines in one fundamental respect.

→ Use of a macro name in the mnemonic field of an assembly statement leads to its expansion.

→ In other words, the statement of expansion are generated each time the macro are invoked

→ Whereas use of a subroutine name in a call instruction leads to its execution.

→ Thus programs using macros and subroutines differ significantly in terms of program size and execution efficiency.

MACRO DEFINITION AND CALL

Macro definition: A macro definition is enclosed between a ***macro header statement*** and a ***macro end statement***.

→ Macro definitions are typically located at the start of the program.

→ A macro definition consists of :
1) A macro prototype statement.
2) One or more model statements.
3) Macro preprocessor statements.

→The macro prototype statement declares the name of a macro and the names and kinds of its parameters.

→A model statement is a statement from which an assembly language statement may be generated during macro expansion.

→A preprocessor statement is used to perform auxiliary functions during macro expansion.

→The macro prototype statement has the following syntax :

<macro name> [<formal parameter spec> [,..]]

where <macro name> appears in the mnemonic field of an assembly statement and <formal parameter spec> is of the form &<parameter name> [<parameter kind>]

Macro call

→ A macro is called by writing the macro name in the mnemonic field of an assembly statement.

→ The macro call has the syntax :

<macro name> [<actual parameter spec> [,..]]

where an actual parameter typically resembles an operand specification in an assembly language statement.

Example showing the definition of macro INCR

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

→MACRO and MEND are the macro header & macro end statements.

Macro expansion

→ A macro call leads to macro expansion.

→ During macro expansion, the macro call statement is replaced by a sequence of assembly statements.

→To differentiate between the original statements of a program and the statements resulting from macro expansion, each expanded statement is marked with a '+' preceding its label field.

Two key notions concerning macro expansion are:

1) **Expansion time control flow**: This determines the order in which model statements are visited during macro expansion.

2) **Lexical substitution**: Lexical substitution is used to generate an assembly statement from a model statement.

Flow of control during expansion

→ The default flow of control during macro expansion is sequential.

→ Thus in the absence of preprocessor statements, the model statements of a macro are visited sequentially starting with the statements following the macro prototype statements and ending with the statements preceding the MEND statements.

→ A preprocessor statements can alter the flow of control during expansion.

→ Alters in such a way that model statements are either never visited during expansion, or are repeatedly visited during expansion.

→ The flow of control during macro expansion is implementated using a macro expansion counter (MEC).

Algorithm of macro expansion

1) MEC:=statement no of first statement following the prototype statement;

2) While statement pointed by MEC is not a MEND statement

(a) if a model statement then

(i) expand the statement

(ii) MEC:=MEC+1;

(b) else(i.e. a preprocessor statement)

(i) MEC:=new value specified in the statement

3) Exit from macro expansion

Lexical substitution

A model statement consists of 3 types of strings:

1) An ordinary string, which stands for itself.

2) The name of a formal parameter which is preceded by the character ‘&’.

3) The name of a preprocessor variable, which is also preceded by the character ‘&’.

Positional parameters

→ A positional formal parameter is written as &<parameter name>
e.g. &SAMPLE where SAMPLE is the name of parameter

→ In other words, <parameter kind> is omitted.

→ The <actual parameter spec> in a call on a macro using positional parameters is simply an <ordinary string>.

→ The value of a positional formal parameter XYZ is determined by the rule of positional association as follows :

1) Find the ordinal position of XYZ in the list of formal parameters in the macro prototype statement.

2) Find the actual parameter specification occupying the same ordinal position in the list of actual parameters in the macro call statement.

Consider the call : INCR A, B, AREG
on macro INCR

Following the rule of positional association, values of the formal parameters are:

formal parameters	value
MEM_VAL	A
INCR_VAL	B
REG	AREG

Lexical expansion of the model statements now leads to the code as follows :

```
+        MOVER        AREG, A
+        ADD          AREG, B
+        MOVEM        AREG, A
```

Keyword parameters

→ For keyword parameters, <parameter name> is an ordinary string and <parameter kind> is the string "=".

→ The <actual parameter spec> is written as <formal parameter name> = <ordinary string>

→ The value of a formal parameter XYZ is determined by the rule of keyword association as follows :

1) Find the actual parameter specification which has the form XYZ = <ordinary string>.

2) Let <ordinary string> in the specification be the string ABC. Then the value of formal parameter XYZ is ABC.

→ For eg : the macro INCR can be rewritten as macro INCR_M using keyword parameters.

→ The following macro calls are now equivalent

```
INCR_M      MEM_VAL=A,   INCR_VAL=B,   REG=AREG
…..
INCR_M      INCR_VAL=B,   REG=AREG,    MEM_VAL=A
```

A macro definition using keyword parameters

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

Default specifications of parameters

→ It 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 the call.

→ Default specification of keyword parameters can be written as:
&<parameter name> [<parameter kind> [<default value>]]

Eg: A macro definition with default parameter

→ Register AREG is used for all arithmetic in a program.

→ Hence most calls on macro INCR_M contain the specification &REG=AREG.

→ The macro can be redefined to use a default specification for the parameter REG.

→ Consider the following calls:

```
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
```

→ The first two calls are equivalent.

→ The third call overrides the default value for REG with the value BREG.

→ BREG will be used to perform the arithmetic in its expanded code.

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

Macros with mixed parameters lists

→ A macro may be defined to use both positional and keyword parameters.

→ In such a case, all positional parameters must precede all keyword parameters.

→ For example, in the macro call :
SUMUP A, B, G=20, H=X

A, B are positional parameters while
G, H are keyword parameters

Other uses of parameters

→ The model statements have used formal parameters only in operand fields.

→ However, use of parameters is not restricted to these fields.

→ Formal parameters can also appear in the label and opcode fields of model statements.

→ The example is shown below :

```
MACRO
CALC        &X,   &Y,    &OP=MULT,    &LAB=

&LAB        MOVER       AREG,  &X
            &OP         AREG,  &Y
            MOVEM       AREG,  &X
            MEND
```

Expansion of the call CALC A, B , LAB=LOOP leads to the following code:

```
+   LOOP    MOVER       AREG,  A
+           MULT        AREG,  B
+           MOVEM       AREG,  A
```

Nested macro calls

→ A macro statement in a macro may constitute a call on another macro. Such calls are known as nested calls.

→ We refer to the macro containing the nested call as the outer macro and the called macro as the inner macro.

→ Thus in a structure of nested macro calls, expansion of the latest macro call (i.e. the innermost macro call in the structure) is completed first.

→ Macro COMPUTE contains a nested call on macro INCR_D

COMPUTE X, Y

→ After the lexical expansion, the second model statement of COMPUTE is recognized to be a call on macro INCR_D.

→ Expansion of this macro is now performed. This leads to generation of statements marked 2, 3 and 4.

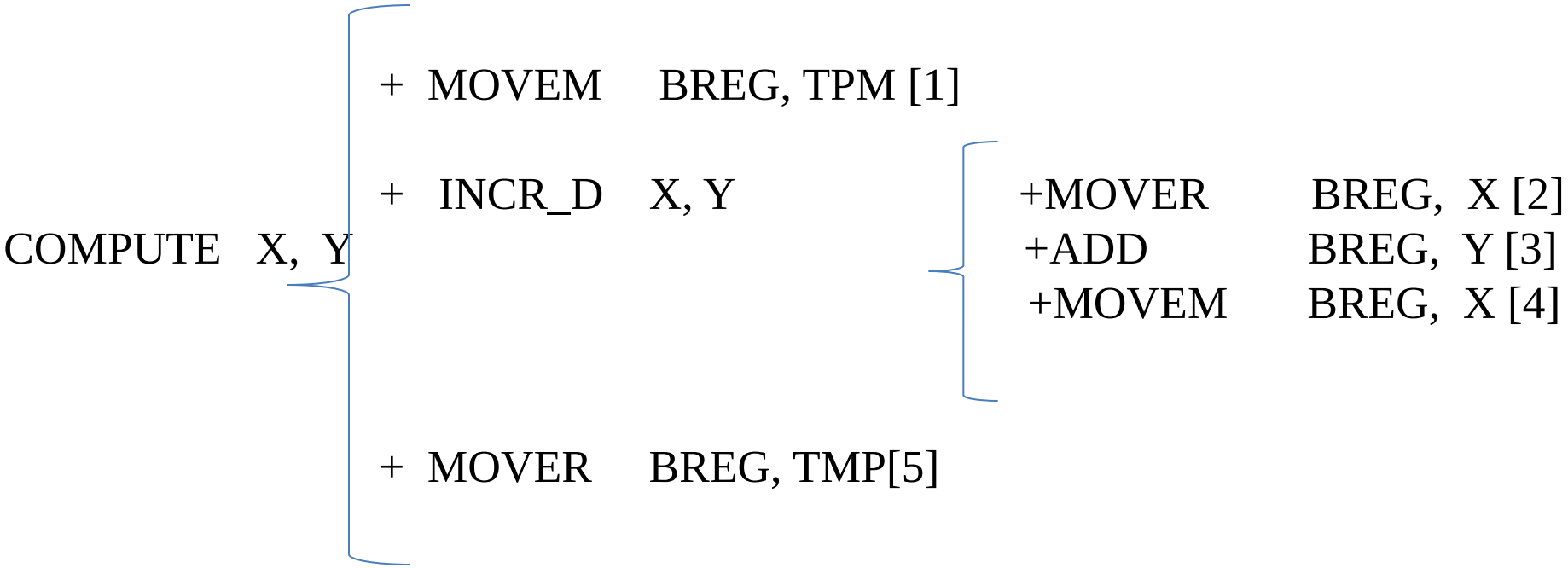
→ The third model statement of COMPUTE is now expanded. Thus the expanded code for the call on COMPUTE is :

A Nested macro call

```
+          MOVEM      BREG,  TMP
+          MOVER      BREG,    X
+          ADD        BREG,    Y
+          MOVEM      BREG,    X
+          MOVER      BREG,  TMP

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

Expanded code for a nested macro call



Advanced macro facilities

→ These facilities can be grouped into :
1) Facilities for alteration of flow of control during expansion.
2) Expansion time variables.
3) Attributes of parameters.

→ These advanced facilities are used in performing conditional expansion of model statements and in writing expansion time loops.

Alteration of flow of control during expansion

→ Two features are provide to facilitate alteration of flow of control during expansion :

1) Expansion time sequencing symbols.
2) Expansion time statements AIF, AGO and ANOP.

→ A sequencing symbol (SS) has the syntax :

.<ordinary string>

→ As SS is defined by putting it in the label field of a statement in the macro body.

→ It is used as an operand in an AIF or AGO statement to designate the destination of an expansion time control transfer.

→ An AIF statement has the syntax :

AIF (<expression>) <sequencing symbol>

where <expression> is a relational involving ordinary strings, formal parameters

→ If the relational expression evaluates to true, expansion time control is transferred to the statement containing <sequencing symbol> in its label field.

→ An AGO statement has the syntax :

AGO <sequencing symbol>

and unconditionally transfers expansion time control to the statement containing <sequencing symbol> in its label field.

→ An ANOP statement is written as :

< sequencing symbol > ANOP

and simply has the effect of defining the sequencing symbol.

```
        MACRO
        EVAL        &X, &Y, &Z
        AIF         (&Y  EQ  &X)   .ONLY
        MOVER       AREG, &X
        SUB         AREG, &Y
        ADD         AREG, &Z
        AGO         .OVER
.ONLY   MOVER       AREG, &Z
.OVER   MEND
```

```
        MACRO
        EX1     &VAR1=AREG,&VAR2=,&VAR3=B
        AIF     (‘&VAR2’ EQ ‘ ’) .NOOPE
        ADD     &VAR1,&VAR3
.NOOPE  ANOP
        MULT    &VAR1,&VAR3
        MEND
```

Expansion time variables

→ Expansion time variables (EV's) are variables which can only be used during the expansion of macro calls.

→ A local EV is created for use only during a particular macro call.

→ A global EV exists across all macro calls situated in a program.

→ Local and global EV's are created through declaration statements with the following syntax :

LCL <EV specification> [, <EV specification> …]
GBL <EV specification> [, <EV specification>…]

and <EV specification has the syntax &<EV name>, where <EV name> is an ordinary string.

→ Value of EV's can be manipulated through the preprocessor statement SET.

→ A SET statement is written as

<EV specification> SET <SET-expression>

where <EV specification appears in the label field and SET in the mnemonic field.

- Each SET variable can be either ***local*** or ***global***.
- A local SET variable is assigned an initial value every time the macro is called.
- It is known only to the macro it is created in.
- A global SET variable is initialized the first time the macro is called and retains a value from one call to another.

→ A SET statement assigns the value of <SET-expression> to the EV specified in <EV specification>.

→ The value of an EV can be used in any field of a model statement, and in the expression of an AIF statement.

```
          MACRO
          CONSTANTS
          LCL     &A
&A        SET      1
          DC      'A'
&A        SET      &A+1
          DC      'A'
          MEND
```

Attributes of formal parameters

→ An attribute is written using the syntax :

<attribute name>'<formal parameter spec>

and represents information about the value of the formal parameter, i.e. the corresponding actual parameter.

→ The type, length and size attributes have the names T, L and S.

```
            MACRO
            DCL_CONST       &A
             AIF            (L'&A  EQ   1)    .NEXT
             - -
.NEXT        - -
             - -
            MEND
```

→ Here expansion time control is transferred to the statement having .NEXT in its label field only if the actual parameter corresponding to the formal parameter A has the length of '1'.

Conditional Expansion

→ While writing a general purpose macro it is important to ensure execution efficiency of its generated code.

→ Conditional expansion helps in generating assembly code specifically suited to the parameters in a macro call.

→ This is achieved by ensuring that a model statement is visited only under specific conditions during the expansion of a macro.

→ The AIF and AGO statements are used for this purpose.

Consider the following call :

EVAL A, B, C

→ It is required to develop a macro EVAL such that a call generates efficient code to evaluate A-B+C in AREG.

→ When the first two parameters of a call are identical, EVAL should generate a single MOVER instruction to load the 3rd parameter into AREG.

→ Since the values of a formal parameters is simply the corresponding actual parameter, the AIF statement effectively compares names of the first two actual parameters.

→ If the names are same, expansion time control is transferred to the model statement MOVER AREG, &Z.

→ If not, the MOVE-SUB-ADD sequence is generated and expansion time control is transferred to the statement .OVER MEND which terminates the expansion.

→ Thus efficient code is generated under all conditions.

```
        MACRO
        EVAL        &X, &Y, &Z
        AIF         (&Y  EQ  &X)   .ONLY
        MOVER       AREG, &X
        SUB         AREG, &Y
        ADD         AREG, &Z
        AGO         .OVER
.ONLY   MOVER       AREG, &Z
.OVER   MEND
```

Expansion time loops

→ It is often necessary to generate many similar statements during the expansion of a macro.

→ This can be achieved by writing similar model statements in the macro.

```
MACRO
CLEAR     &A
MOVER     AREG,   =‘0’
MOVEM     AREG,   &A
MOVEM     AREG,   &A+1
MOVEM     AREG,   &A+2
MEND
```

→ When called as CLEAR B, the MOVER statement puts the value '0' in AREG, while the three MOVEM statements store this value in 3 consecutive bytes with the addresses B, B+1 and B+2.

→ Expansion time loops can be written using expansion time variables (EV's) and expansion time control transfer statements AIF and AGO.

```
            MACRO
            CLEAR     &X,  &N
            LCL       &M
&M          SET       0
            MOVER     AREG,   ='0'
.MORE       MOVEM     AREG,   &X+&M
&M          SET       &M+1
            AIF       (&M  NE  N)  .MORE
            MEND
```

Consider expansion of the macro call :

CLEAR B, 3

→ The LCL statement declares M to be a local EV.
At the start of expansion of the call, M is initialized to zero.

→ The expansion of model statements MOVEM AREG, &x+&M thus leads to generation of the statement MOVEM AREG, B.

→ The value of M is incremented by 1 and the model statement is expanded repeatedly until its value equals the value of N, which is 3 in this case.

→Thus the macro call leads to generation of the statements :

```
+    MOVER    AREG,  ='0'
+    MOVEM    AREG,    B
+    MOVEM    AREG,   B+1
+    MOVEM    AREG,   B+2
```

→ Most expansion time loops can be replaced by execution time loops.

For eg: instead of generating many MOVEM statements to clear the memory area starting on B, it is possible to write an execution time loop which moves 0 into B, B+1 and B+2.

→ It leads to more compact assembly programs. such programs would execute slower than programs containing expansion time loops.

→ Thus macro can be used to trade program size for execution efficiency.

Other facilities for expansion time loops

→ Many assemblers provide other facilities for conditional expansion, an ELSE clause in AIF being an obvious example.

→ The assemblers for M 68000 and Intel 8088 processors provide explicit expansion time looping constructs.

→ Two such facilities are :
1) the REPT statement
2) the IRP statement

→ The REPT statement can be written as :

REPT <expression>

→ Here expression should evaluate to a numerical value during macro expansion.

→ The statements between REPT and ENDM statement would be processed for expansion<expression> number of times.

→ The following example illustrates the use of this facility to declare 10 constants with the values 1, 2, ..10.

```
        MACRO
        CONST10
        LCL     &M
&M      SET     1
        REPT    10
        DC      '&M'
&M      SET     &M+1
        ENDM
        MEND
```

```
define macro count
        value=0
        REPT count
        db value
        value=value+1
        endm
endm
```

→ The IRP statement can be written as :

IRP <formal parameter>, <argument-list>

→ The formal parameter in the statement takes successive values from the argument list.

→ For each value, the statements between the IRP and ENDM statements are expanded once.

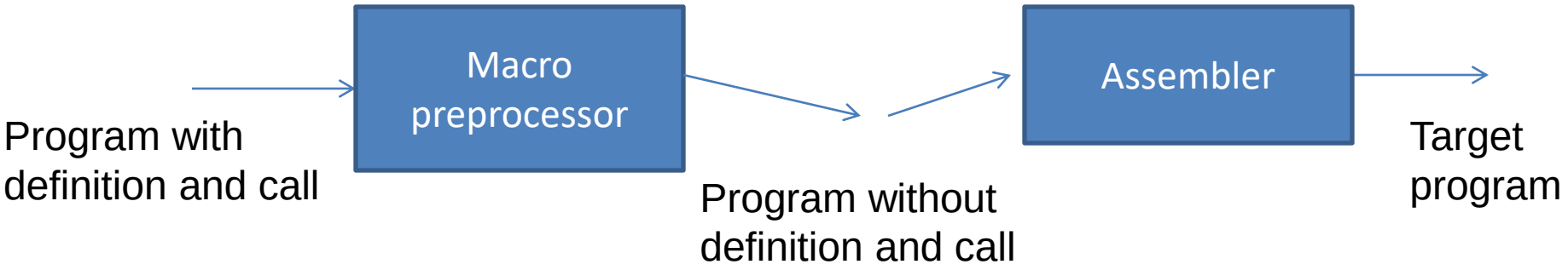
```
MACRO
CONSTS
IRP             arg,<4,10>
 DB             arg
ENDM
MEND
```

→ A macro call CONSTS 4,10 leads to declaration of 2 block of bytes with value 4 and 10.

Design of macro preprocessor

→ The macro preprocessor accepts an assembly program containing definitions and calls and translates it into an assembly program which does not contain any macro definitions and calls.

→this output is handed over to an assembler to obtain the target language form of the program.


Program with
definition and call
Macro
preprocessor
Program without
definition and call
Assembler
Target
program

Design overview

Tasks involved in macro expansion

1) Identify macro calls in the program.

2) Determine the values of formal parameters.

3) Maintain the values of expansion time variables declared in a macro.

4) Organize expansion time control flow.

5) Determine the values of sequencing symbol.

6) Perform expansion of a model statement.

The following 4 step procedure is followed to arrive at a design specification for each task :

1) Identify the information necessary to perform task.

2) Design a suitable data structure to record the information.

3) Determine the processing necessary to obtain the information.

4) Determine the processing necessary to perform the task.

1) Identify macro calls

→A table called the macro name table (MNT) is designed to hold the names of all macros defined in a program.

→ A macro name is entered in this table when a macro definition is processed.

→ While processing a statement in the source program, the preprocessor compares the string found in its mnemonic field with the macro names in MNT.

→ A match indicates that the current statement is a macro call.

2) Determine values of formal parameters

→ A table called the actual parameter table (APT) is designed to hold the values of formal parameters during the expansion of a macro call.

→ Each entry in the table is a pair

(< formal parameter name >, < value >)

→ Two items of information are needed to construct this table, name of formal parameters and default values of keyword parameters.

→ For this purpose, a table called the parameter default table (PDT) is used for each macro.

→ This table would be accessible from the MNT entry of a macro and would contain pairs of the form

(< formal parameter name >, < default value >).

→ If a macro call statement does not specify a value for some parameter par, its default value would be copied from PDT to APT.

3) Maintain expansion time variables

→ An expansion time variables table (EVT) is maintained for this purpose.

→ The table contains the pairs of the form

(< EV name >, < value >)

→ The value field of a pair is accessed when a preprocessor statement or a model statement under expansion refers to an EV.

4) Organize expansion time control flow

→ The body of a macro, i.e. the set of preprocessor statements and model statements in it is stored in a table called the macro definition table (MDT) for use during macro expansion.

→ The flow of control during expansion determines when a model statement is to be visited for expansion.

5) Determine values of sequencing symbols

→ A sequencing symbols table (SST) is maintained to hold this information.

→ The table contains pairs of the form

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

where <MDT entry #> is the number of the MDT entry which contains the model statements defining the sequencing symbol.

This entry is made on encountering a statement which contains the sequencing symbol in its label field

6) Perform expansion of a model statement

This is a trivial task given the following :

1) MEC points to the MDT entry containing the model statements.

2) Values of formal parameters and EV's are available in APT and EVT respectively.

3) The model statements defining a sequencing symbol can be identified from SST.

→ Expansion of model statements is achieved by performing a lexical substitution for the parameters & EV's used

Data structures

→ To obtain a detailed design of the data structures it is necessary to apply the practical criteria of processing efficiency and memory requirements.

→ The tables APT, PDT and EVT contain pairs which are searched using the first component of the pairs as a key---for eg., the formal parameter name is used as the key to obtain its value from APT.

→ This search can be eliminated if the position of an entity within a table is known when its value is to be accessed.

→ The value of a formal parameter ABC is needed while expanding a model statement using it., viz

MOVER AREG, &ABC

→ let the pair (ABC, ALPHA) occupy entry #5 in APT.

→ The search in APT can be avoided if the model statement appears as

MOVER AREG, (P,5)

in the MDT, where (P,5) stands for the words ‘parameter #5’.

→ Thus the macro expansion can be made more efficient by storing an intermediate code for a statement, rather than its source form in the MDT.

→ All parameter names could be replaced by pairs of the form (P,n) in model statements and preprocessor statements stored in MDT.

→ The first component of the pairs stored in APT is no longer used during macro expansion.

→ Eg. The information (P,5) appearing in a model statement is sufficient to access the value of formal parameter ABC.

→ Hence APT containing (<formal parameter name>, <value>) pairs is replaced by another table called APTAB which only contains <value>'s.

→ To implement this, ordinal no are assigned to all parameters of a macro.

→ A table named parameter name table (PNTAB) is used for this purpose.

→PNTAB is used while processing the definition of a macro.Parameter names are entered in PNTAB in the same order in which they appear in the prototype statement.

→ The entry # of a parameter's entry in PNTAB is now its ordinal no.

→ This entry is used to replace the parameter name in the model and preprocessor statements of the macro while storing it in the MDT.

→ This implements the requirement that the statement MOVER AREG, &ABC should appear as MOVER AREG, (P,5) in MDT.

In effect, the information (<formal parameter name>,<value>) in APT has been split into two tables :
1) PNTAB—which contains formal parameter names
2) APTAB---which contains formal parameters values

→ PNTAB is used while processing a macro definition while APTAB is used during macro expansion.

→ Similarly EVT is splitted into EVNTAB and EVTAB and SST into SSNTAB and SSTAB.

→ EV names are entered in EVNTAB while processing EV declarations.

→ SS names are entered in SSNTAB while processing an SS reference or definition, whichever occurs earlier.

→ This arrangement leads to some simplifications concerning PDT.

→ The positional parameters of a macro appear before keyword parameters in the prototype statement.

→ Hence in the prototype statement for a macro BETA which has p positional and k keyword parameters, the keyword parameters have the ordinal no p+1…p+k .

→ Due to this numbering, 2 kinds of redundancies appear in PDT.

→ The first component of each entry is redundant as in APTAB and EVTAB.

TABLE	FIELDS IN EACH ENTRY
1) MNT (Macro name table)	Macro name, Number of positional parameters (#PP), Number of keyword parameters (#KP), Number of expansion time variables (#EV), MDT pointer (MDTP), KPDTAB pointer (KPDTAB), SSTAB pointer (SSTAB).
2) PNTAB (parameter name table)	Parameter name
3) EVNTAB (EV name table)	EV name
4) SSNTAB (SS name table)	SS name
5) KPDTAB (keyword parameter default table)	Parameter name, default value
6) MDT (macro definition table)	Label, opcode, operands
7) APTAB (actual parameter table)	value
8) EVTAB (EV table)	value
9) SSTAB (SS table)	MDT entry #

SUMMARY

→ PNTAB and KPDTAB are constructed by processing the prototype statement.

→ Entries are added to EVNTAB and SSNTAB as EV declarations and SS definitions/references are encountered.

→ MDT are constructed while processing the model statements and preprocessor statements in macro body.

→ An entry is added to SSTAB when the definition of a sequencing symbol is encountered.

→APTAB is constructed while processing a macro call.

→EVTAB is constructed at start of expansion of a macro.

MACRO EXPANSION

→ We use the following data structure to perform macro expansion
  - APTAB Actual Parameter Table
  - EVTAB EV Table
  - MEC Macro expansion counter
  - APTAB_ptr APTAB pointer
  - EVTAB_ptr EVTAB pointer

→ The number of entry in APTAB equals the sum of values in the #PP and #KP fields of the MNT entry of macro.

→ Number of entries in EVTAB is given by value in #EV field of MNT.

→ APTAB and EVTAB are constructed when a macro call is recognized.

→ APTAB_ptr and EVTAB_ptr are set to point at these tables.

→ MEC always pointers to the next statement to be expanded.