The LTORG directive has very little relevance for the simple assembly language we have assumed so far. The need to allocate literals at intermediate points in the program rather than at the end is critically felt in a computer using a base displacement mode of addressing, e.g. computers of the IBM 360/370 family.

EXERCISE 4.4.1

1. An assembly program contains the statement

```
X       EQU     Y+25
```

Indicate how the EQU statement can be processed if
(a) Y is a back reference,
(b) Y is a forward reference.

2. Can the operand expression in an ORIGIN statement contain forward references? If so, outline how the statement can be processed in a two pass assembly scheme.

4.4.2 Pass I of the Assembler

Pass I uses the following data structures:

OPTAB	A table of mnemonic opcodes and related information
SYMTAB	Symbol table
LITTAB	A table of literals used in the program

Figure 4.9 illustrates sample contents of these tables while processing the program of Fig. 4.8. OPTAB contains the fields *mnemonic opcode*, *class* and *mnemonic info*. The *class* field indicates whether the opcode corresponds to an imperative statement (IS), a declaration statement (DL) or an assembler directive (AD). If an imperative, the *mnemonic info* field contains the pair (*machine opcode, instruction length*), else it contains the id of a routine to handle the declaration or directive statement. A SYMTAB entry contains the fields *address* and *length*. A LITTAB entry contains the fields *literal* and *address*.

Processing of an assembly statement begins with the processing of its label field. If it contains a symbol, the symbol and the value in LC is copied into a new entry of SYMTAB. Thereafter, the functioning of Pass I centers around the interpretation of the OPTAB entry for the mnemonic. The *class* field of the entry is examined to determine whether the mnemonic belongs to the class of imperative, declaration or assembler directive statements. In the case of an imperative statement, the length of the machine instruction is simply added to the LC. The length is also entered in the SYMTAB entry of the symbol (if any) defined in the statement. This completes the processing of the statement.

For a declaration or assembler directive statement, the routine mentioned in the *mnemonic info* field is called to perform appropriate processing of the statement. For example, in the case of a DS statement, routine R#7 would be called. This routine

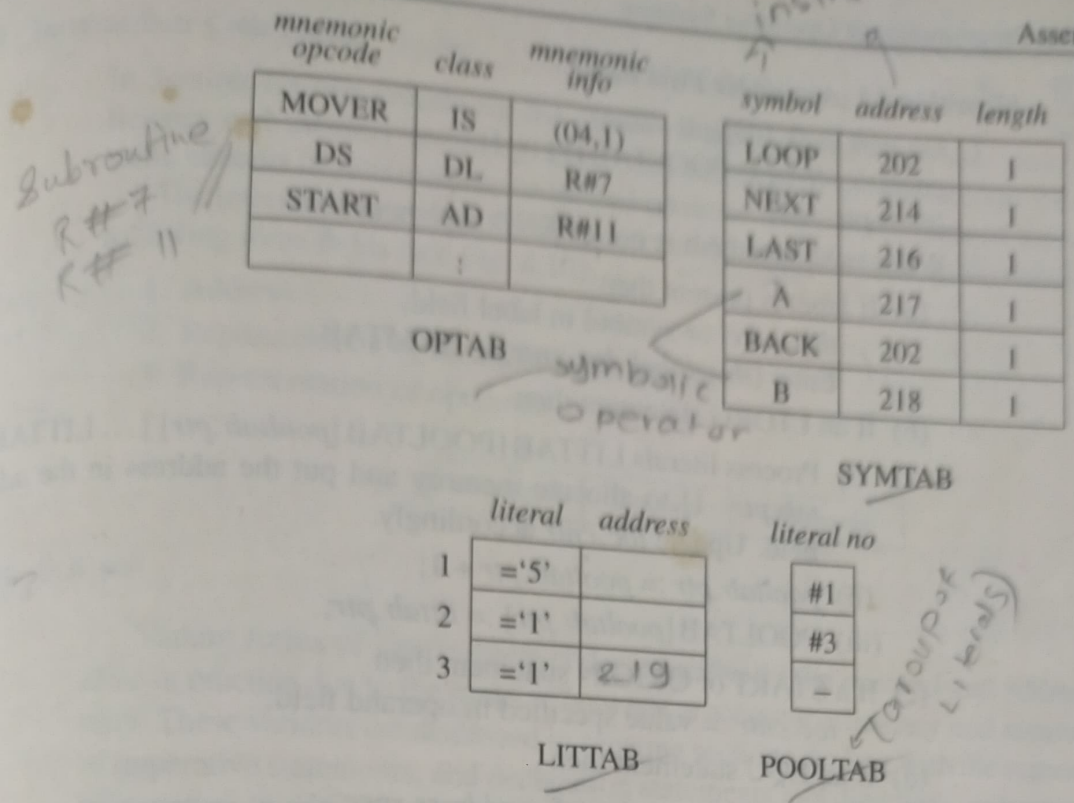


Fig. 4.9 Data structures of assembler Pass I

processes the operand field of the statement to determine the amount of memory required by this statement and appropriately updates the LC and the SYMTAB entry of the symbol (if any) defined in the statement. Similarly, for an assembler directive the called routine would perform appropriate processing, possibly affecting the value in LC.

The use of LITTAB needs some explanation. The first pass uses LITTAB to collect all literals used in a program. Awareness of different literal pools is maintained using the auxiliary table POOLTAB. This table contains the literal number of the starting literal of each literal pool. At any stage, the current literal pool is the last pool in LITTAB. On encountering an LTORG statement (or the `END` statement), literals in the current pool are allocated addresses starting with the current value in LC and LC is appropriately incremented. Thus, the literals of the program in Fig. 4.8(a) will be allocated memory in two steps. At the `LTORG` statement, the first two literals will be allocated the addresses 211 and 212. At the `END` statement, the third literal will be allocated address 219.

We now present the algorithm for the first pass of the assembler. Intermediate code forms for use in a two pass assembler are discussed in the next section.

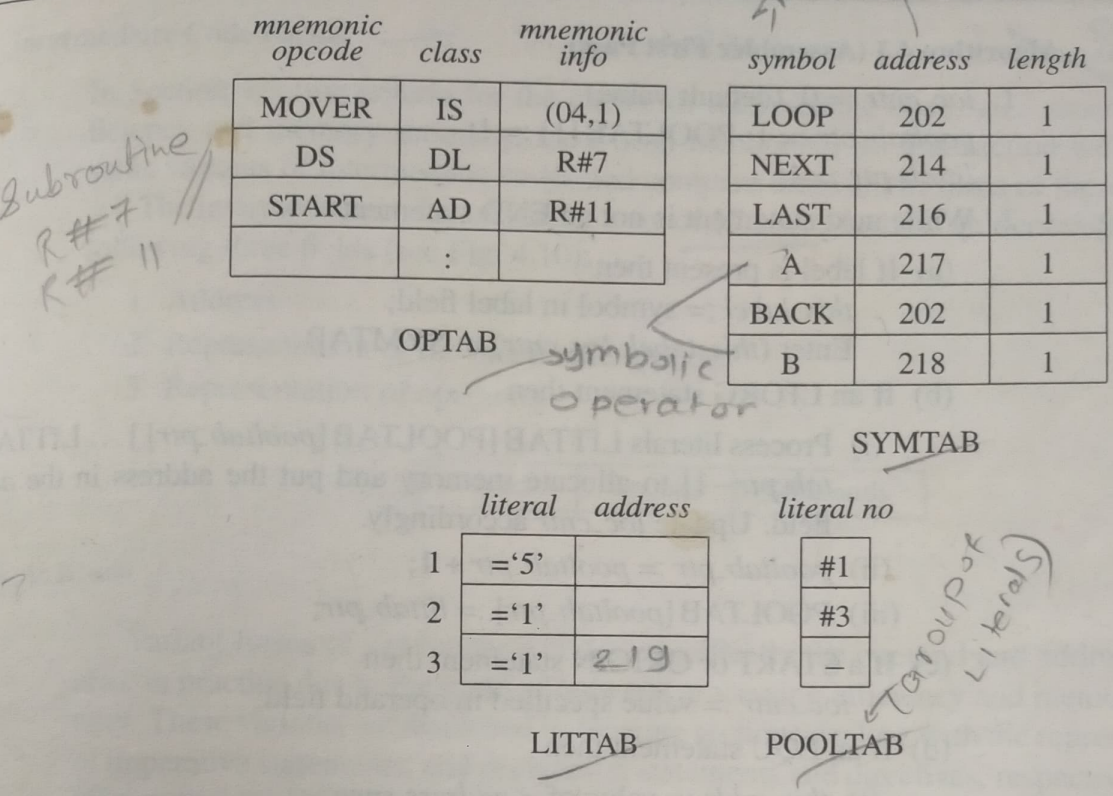


Fig. 4.9 Data structures of assembler Pass I

processes the operand field of the statement to determine the amount of memory required by this statement and appropriately updates the LC and the SYMTAB entry of the symbol (if any) defined in the statement. Similarly, for an assembler directive the called routine would perform appropriate processing, possibly affecting the value in LC.

The use of LITTAB needs some explanation. The first pass uses LITTAB to collect all literals used in a program. Awareness of different literal pools is maintained using the auxiliary table POOLTAB. This table contains the literal number of the starting literal of each literal pool. At any stage, the current literal pool is the last pool in LITTAB. On encountering an LTORG statement (or the `END` statement), literals in the current pool are allocated addresses starting with the current value in LC and LC is appropriately incremented. Thus, the literals of the program in Fig. 4.8(a) will be allocated memory in two steps. At the `LTORG` statement, the first two literals will be allocated the addresses 211 and 212. At the `END` statement, the third literal will be allocated address 219.

We now present the algorithm for the first pass of the assembler. Intermediate code forms for use in a two pass assembler are discussed in the next section.

IMP

Algorithm 4.1 (Assembler First Pass)

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). any literal declare after end stmt
   (b) Generate IC '(AD,02)'.
   (c) Go to Pass II.

4.4.6 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.

CHAPTER 5

Macros and Macro Processors

Macros are used to provide a *program generation* facility (see Section 1.2.1) through *macro expansion*. Many languages provide built-in facilities for writing macros. Well known examples of these 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 macro facilities, a programmer may achieve an equivalent effect by using generalized preprocessors or software tools like Awk of Unix. The discussion in this chapter is confined to macro facilities provided in assembly languages.

Definition 5.1 (Macro) *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 a macro name with a set of actual parameters is replaced by some code generated from its body. This is called *macro expansion*. Two kinds of expansion can be readily identified.

1. *Lexical expansion:* Lexical expansion implies replacement of a character string by another character string during program generation. Lexical expansion 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—for example, generation of type specific instructions for manipulation of byte and word operands. Semantic expansion is characterized by the fact that different uses of a macro can lead to codes which differ in the number, sequence and opcodes of instructions.

Example 5.1 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 a machine register.
2. Increment the value in the machine register.
3. Move the new value into the memory word.

Since the instruction sequence MOVE-ADD-MOVE may be used a number of 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 example, for Intel 8088, an INC instruction could be generated if A is a byte operand and B has the value '1', while a MOV-ADD-MOV sequence can be generated in all other situations.

Note that 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*, 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. In fact, macros can be said to trade program size for execution efficiency of a program. Other differences between macros and subroutines will be discussed along with the discussion of advanced macro facilities in Section 5.4.

5.1 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 a 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*>] (5.1)

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

$$\langle \textit{macro name} \rangle \quad [\langle \textit{actual parameter spec} \rangle [,..]] \tag{5.2}$$

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

Example 5.2 Figure 5.1 shows the definition of macro INCR. MACRO and MEND are the macro header and macro end statements, respectively. The prototype statement indicates that three parameters called MEM_VAL, INCR_VAL and REG exist for the macro. Since parameter kind is not specified for any of the parameters, they are all of the default kind 'positional parameter'. Statements with the operation codes MOVER, ADD and MOVEM are model statements. No preprocessor statements are used in this macro.

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

Fig. 5.1 A macro definition

5.2 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 statement following the macro prototype statement and ending with the statement preceding the MEND statement. A preprocessor statement can alter the flow of control during expansion such that some model statements

are either never visited during expansion, or are repeatedly visited during expansion. The former results in *conditional expansion* and the latter in *expansion time loops*. The flow of control during macro expansion is implemented using a *macro expansion counter* (MEC).

Algorithm 5.1 (Outline of macro expansion)

1. MEC := statement number 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.

MEC is set to point at the statement following the prototype statement. It is incremented by 1 after expanding a model statement. Execution of a preprocessor statement can set MEC to a new value to implement conditional expansion or expansion time loops. These features are discussed in Section 5.4.

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 '&'.

During lexical expansion, strings of type 1 are retained without substitution. Strings of types 2 and 3 are replaced by the 'values' of the formal parameters or preprocessor variables. The value of a formal parameter is the corresponding actual parameter string. The rules for determining the value of a formal parameter depend on the kind of parameter.

Positional parameters

A positional formal parameter is written as &<*parameter name*>, e.g. &SAMPLE where SAMPLE is the name of a parameter. In other words, <*parameter kind*> of syntax rule (5.1) is omitted. The <*actual parameter spec*> in a call on a macro using positional parameters [see syntax rule (5.2)] is simply an <*ordinary string*>.

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

```
SUMUP     A,B,G=20,H=X
```

A, B are positional parameters while G, H are keyword parameters. Correspondence between actual and formal parameters is established by applying the rules governing positional and keyword parameters separately.

Other uses of parameters

The model statements of Examples 5.2-5.5 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.

Example 5.6

```
        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
```

5.3 NESTED MACRO CALLS

A model statement in a macro may constitute a call on another macro. Such calls are known as *nested macro calls*. We refer to the macro containing the nested call as the *outer* macro and the called macro as the *inner* macro. Expansion of nested macro calls follows the *last-in-first-out* (LIFO) rule. 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.

Example 5.7 Macro COMPUTE of Fig. 5.4 contains a nested call on macro INCR_D of Fig. 5.3. Figure 5.5 shows the expanded code for the call

```
COMPUTE   X, Y
```

After 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 in Fig. 5.5. The third model statement of COMPUTE is now expanded. Thus the expanded code for the call on COMPUTE is:

```
+          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
```

Fig. 5.4 A nested macro call

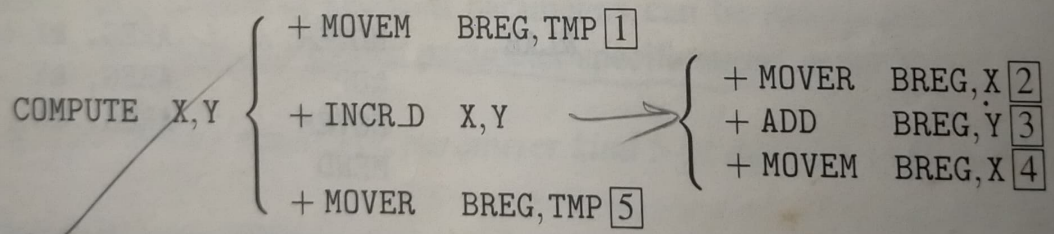


Fig. 5.5 Expanded code for a nested macro call

5.4 ADVANCED MACRO FACILITIES

Advanced macro facilities are aimed at supporting semantic expansion. These facilities can be grouped into

1. Facilities for alteration of flow of control during expansion
2. Expansion time variables
3. Attributes of parameters.

CHAPTER 7

Linkers

Execution of a program written in a language L involves the following steps:

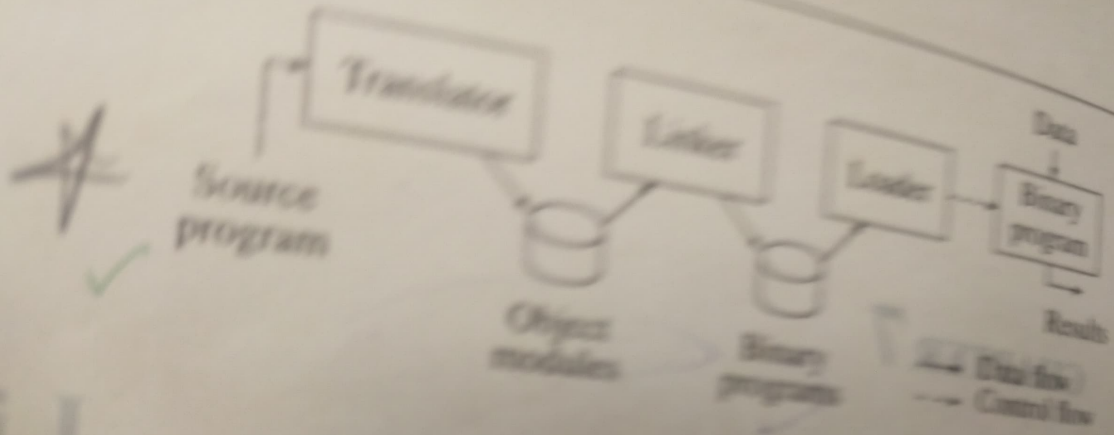
1. *Translation* of the program
2. *Linking* of the program with other programs needed for its execution
3. *Relocation* of the program to execute from the specific memory area allocated to it
4. *Loading* of the program in the memory for the purpose of execution.

These steps are performed by different language processors. Step 1 is performed by the translator for language L. Steps 2 and 3 are performed by a *linker* while Step 4 is performed by a *loader*. The terms linking, relocation and loading are defined in a later section.

Figure 7.1 contains a schematic showing steps 1–4 in the execution of a program. The translator outputs a program form called *object module* for the program. The linker processes a set of object modules to produce a ready-to-execute program form, which we will call a *binary program*. The loader loads this program into the memory for the purpose of execution. As shown in the schematic, the object module(s) and ready-to-execute program forms can be stored in the form of files for repeated use.

Translated, linked and load time addresses

While compiling a program P, a translator is given an origin specification for P. This is called the *translated origin* of P. (In an assembly program, the programmer can specify the origin in a `START` or `ORIGIN` statement.) The translator uses the value of the translated origin to perform memory allocation for the symbols declared in P. This results in the assignment of a *translation time address* t_{symb} to each symbol *symb* in the program. The *execution start address* or simply the *start address* of a program is the address of the instruction from which its execution must begin. The start address specified by the translator is the *translated start address* of the program



A schematic of program execution

The origin of a program may have to be changed by the linker or loader for one of two reasons. First, the same set of translated addresses may have been used in different object modules constituting a program, e.g. object modules of library routines often have the same translated origin. Memory allocation to such programs would conflict unless their origins are changed. Second, an operating system may require that a program should execute from a specific area of memory. This may require a change in its origin. The change of origin leads to changes in the execution start address and in the addresses assigned to symbols. The following terminology is used to refer to the address of a program entity at different times:

1. *Translation time* (or *translated*) *address*: Address assigned by the translator.
2. *Linked address*: Address assigned by the linker.

Fig. 7.2 A

7.1 REL

7.1.1 Pr

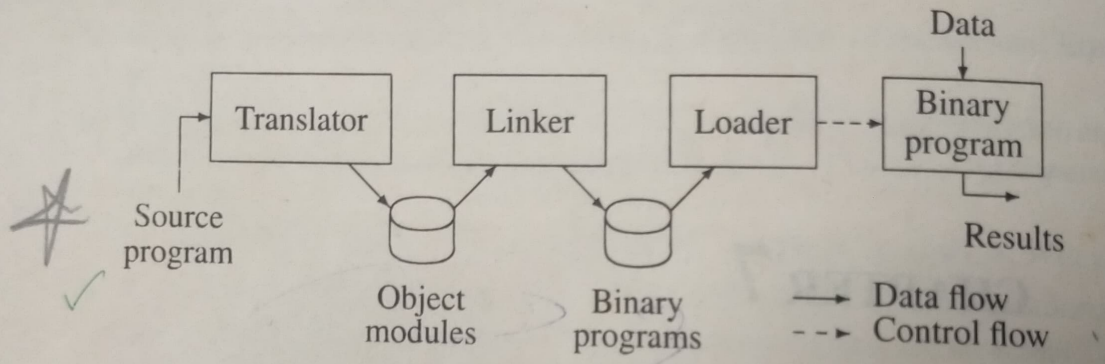


Fig. 7.1 A schematic of program execution

The origin of a program may have to be changed by the linker or loader for one of two reasons. First, the same set of translated addresses may have been used in different object modules constituting a program, e.g. object modules of library rotines often have the same translated origin. Memory allocation to such programs would conflict unless their origins are changed. Second, an operating system may require that a program should execute from a specific area of memory. This may require a change in its origin. The change of origin leads to changes in the execution start address and in the addresses assigned to symbols. The following terminology is used to refer to the address of a program entity at different times:

1. *Translation time* (or *translated*) *address:* Address assigned by the translator.
2. *Linked address:* Address assigned by the linker.
3. *Load time* (or *load*) *address:* Address

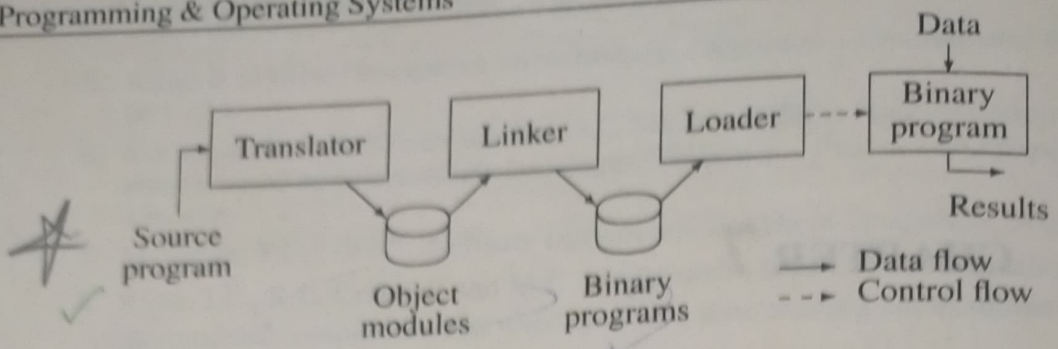


Fig. 7.1 A schematic of program execution

The origin of a program may have to be changed by the linker or loader for one of two reasons. First, the same set of translated addresses may have been used in different object modules constituting a program, e.g. object modules of library rotines often have the same translated origin. Memory allocation to such programs would conflict unless their origins are changed. Second, an operating system may require that a program should execute from a specific area of memory. This may require a change in its origin. The change of origin leads to changes in the execution start address and in the addresses assigned to symbols. The following terminology is used to refer to the address of a program entity at different times:

1. *Translation time* (or *translated*) *address:* Address assigned by the translator.
2. *Linked address:* Address assigned by the linker.
3. *Load time* (or *load*) *address:* Address assigned by the loader.

The same prefixes *translation time* (or *translated*), *linked* and *load time* (or *load*) are used with the origin and execution start address of a program. Thus,

1. *Translated origin:* Address of the origin assumed by the translator. This is the address specified by the programmer in an ORIGIN statement.
2. *Linked origin:* Address of the origin assigned by the linker while producing a binary program.
3. *Load origin:* Address of the origin assigned by the loader while loading the program for execution.

The linked and load origins may differ from the translated origin of a program due to one of the reasons mentioned earlier.

Example 7.1 Consider the assembly program and its generated code shown in Fig. 7.2. The translated origin of the program is 500. The translation time address of LOOP is therefore 501. If the program is loaded for execution in the memory area starting with the address 900. the load time origin is 900. The load time address of LOOP would be 901.

```
                Statement                Address     Code
                START   500
                ENTRY   TOTAL
                EXTRN   MAX, ALPHA
                READ    A                500)    + 09 0 540
LOOP                                     501)
                  :
                MOVER   AREG, ALPHA      518)    + 04 1 000
                BC      ANY, MAX         519)    + 06 6 000
                  :
                BC      LT, LOOP         538)    + 06 1 501
                STOP                     539)    + 00 0 000
A               DS      1                540)
TOTAL           DS      1                541)
                END
```

Fig. 7.2 A sample assembly program and its generated code

7.1 RELOCATION AND LINKING CONCEPTS

7.1.1 Program Relocation

Let AA be the set of absolute addresses—instruction or data addresses—used in the instructions of a program P. AA $\neq \phi$ implies that program P assumes its instructions and data to occupy memory words with specific addresses. Such a program—called an *address sensitive program*—contains one or more of the following:

1. An *address sensitive instruction:* an instruction which uses an address $a_i \in$ AA.
2. An *address constant:* a data word which contains an address $a_i \in$ AA.

In the following, we discuss relocation of programs containing address sensitive instructions. Address constants are handled analogously.

An address sensitive program P can execute correctly only if the start address of the memory area allocated to it is the same as its translated origin. To execute correctly from any other memory area, the address used in each address sensitive instruction of P must be 'corrected'.

Definition 7.1 (Program relocation) Program relocation *is the process of modifying the addresses used in the address sensitive instructions of a program such that the program can execute correctly from the designated area of memory.*

If linked origin $\neq$ translated origin, relocation must be performed by the linker. If load origin $\neq$ linked origin, relocation must be performed by the loader. In general, a linker always performs relocation, whereas some loaders do not. For simplicity, in the first part of the chapter it has been assumed that loaders do not perform