CHAPTER 4

Assemblers

4.1 ELEMENTS OF ASSEMBLY LANGUAGE PROGRAMMING

An assembly language is a machine dependent, low level programming language which is specific to a certain computer system (or a family of computer systems). Compared to the machine language of a computer system, it provides three basic features which simplify programming:

1. *Mnemonic operation codes:* Use of mnemonic operation codes (also called *mnemonic opcodes*) for machine instructions eliminates the need to memorize numeric operation codes. It also enables the assembler to provide helpful diagnostics, for example indication of misspelt operation codes.
2. *Symbolic operands:* Symbolic names can be associated with data or instructions. These symbolic names can be used as operands in assembly statements. The assembler performs memory bindings to these names; the programmer need not know any details of the memory bindings performed by the assembler. This leads to a very important practical advantage during program modification as discussed in Section 4.1.2.
3. *Data declarations:* Data can be declared in a variety of notations, including the decimal notation. This avoids manual conversion of constants into their internal machine representation, for example, conversion of -5 into $(11111010)_2$ or 10.5 into $(41A80000)_{16}$.

Statement format

An assembly language statement has the following format:

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

where the notation [..] indicates that the enclosed specification is optional. If a label is specified in a statement, it is associated as a symbolic name with the memory word(s) generated for the statement. <*operand spec*> has the following syntax:

<*symbolic name*> [+<*displacement*>][(<*index register*>)]

Thus, some possible operand forms are: `AREA`, `AREA+5`, `AREA(4)`, and `AREA+5(4)`. The first specification refers to the memory word with which the name `AREA` is associated. The second specification refers to the memory word 5 words away from the word with the name `AREA`. Here '5' is the *displacement* or *offset* from `AREA`. The third specification implies indexing with index register 4—that is, the operand address is obtained by adding the contents of index register 4 to the address of `AREA`. The last specification is a combination of the previous two specifications.

mple assembly language

In the first half of the chapter we use a simple assembly language to illustrate features of assembly languages and techniques used in assemblers. In this language, each statement has two operands, the first operand is always a register which can be any one of `AREG`, `BREG`, `CREG` and `DREG`. The second operand refers to a memory word using a symbolic name and an optional displacement. (Note that indexing is not permitted.)

Instruction opcode	*Assembly mnemonic*	*Remarks*
00	STOP	Stop execution
01	ADD	*First operand is modified*
02	SUB	*Condition code is set*
03	MULT	
04	MOVER	Register ← memory move
05	MOVEM	Memory ← register move
06	COMP	Sets condition code
07	BC	Branch on condition
08	DIV	Analogous to SUB
09	READ	*First operand is not used*
10	PRINT	

. 4.1 Mnemonic operation codes

Figure 4.1 lists the mnemonic opcodes for machine instructions. The `MOVE` instructions move a value between a memory word and a register. In the MOVER instruction the second operand is the source operand and the first operand is the target operand. Converse is true for the MOVEM instruction. All arithmetic is performed in a register (i.e. the result replaces the contents of a register) and sets a *condition code*. A comparison instruction sets a condition code analogous to a subtract instruction without affecting the values of its operands. The condition code can be tested by a Branch on Condition (`BC`) instruction. The assembly statement corresponding to it has the format

`BC` <*condition code spec*>, <*memory address*>

It transfers control to the memory word with the address <*memory address*> if the current value of condition code matches <*condition code spec*>. For simplicity, we assume <*condition code spec*> to be a character string with obvious meaning, e.g. GT, EQ, etc. A BC statement with the condition code spec ANY implies unconditional transfer of control. In a machine language program, we show all addresses and constants in decimal rather than in octal or hexadecimal.

Figure 4.2 shows the machine instructions format. The opcode, register operand and memory operand occupy 2, 1 and 3 digits, respectively. The sign is not a part of the instruction. The condition code specified in a BC statement is encoded into the first operand position using the codes 1-6 for the specifications LT, LE, EQ, GT, GE and ANY, respectively. Figure 4.3 shows an assembly language program and an equivalent machine language program.

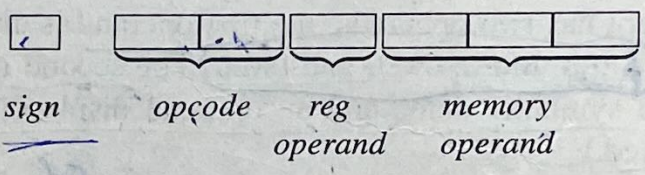

Fig. 4.2 Instruction format

```
                START   101
                READ    N
                MOVER   BREG, ONE
                MOVEM   BREG, TERM
AGAIN           MULT    BREG, TERM
                MOVER   CREG, TERM
                ADD     CREG, ONE
                MOVEM   CREG, TERM
                COMP    CREG, N
                BC      LE, AGAIN
                MOVEM   BREG, RESULT
                PRINT   RESULT
                STOP
N               DS      1
RESULT          DS      1
ONE             DC      '1'
TERM            DS      1
                END
```

```
101)  + 09 0 113
102)  + 04 2 115
103)  + 05 2 116
104)  + 03 2 116
105)  + 04 3 116
106)  + 01 3 115
107)  + 05 3 116
108)  + 06 3 113
109)  + 07 2 104
110)  + 05 2 114
111)  + 10 0 114
112)  + 00 0 000
113)
114)
115)  + 00 0 001
116)
```

Fig. 4.3 An assembly and equivalent machine language program

4.1.1 Assembly Language Statements

An assembly program contains three kinds of statements:

1. Imperative statements

2. Declaration statements
3. Assembler directives.

Imperative statements

An imperative statement indicates an action to be performed during the execution of the assembled program. Each imperative statement typically translates into one machine instruction.

Declaration statements

The syntax of declaration statements is as follows:

```
[Label]  DS   <constant>
[Label]  DC   '<value>'
```

The DS (short for *declare storage*) statement reserves areas of memory and associates names with them. Consider the following DS statements:

```
A        DS   1
G        DS   200
```

The first statement reserves a memory area of 1 word and associates the name A with it. The second statement reserves a block of 200 memory words. The name G is associated with the first word of the block. Other words in the block can be accessed through offsets from G, e.g. G+5 is the sixth word of the memory block, etc.

The DC (short for *declare constant*) statement constructs memory words containing constants. The statement

```
ONE      DC      '1'
```

associates the name ONE with a memory word containing the value '1'. The programmer can declare constants in different forms—decimal, binary, hexadecimal, etc. The assembler converts them to the appropriate internal form.

Use of constants

Contrary to the name 'declare constant', the DC statement does not really implement constants, it merely initializes memory words to given values. These values are not protected by the assembler; they may be changed by moving a new value into the memory word. For example, in Fig. 4.3 the value of ONE can be changed by executing an instruction MOVEM BREG, ONE.

An assembly program can use constants in the sense implemented in an HLL in two ways—as immediate operands, and as literals. Immediate operands can be used in an assembly statement only if the architecture of the target machine includes the necessary features. In such a machine, the assembly statement

```
ADD        AREG,5
```

is translated into an instruction with two operands—AREG and the value '5' as an immediate operand. Note that our simple assembly language does not support this feature, whereas the assembly language of Intel 8086 supports it (see Section 4.5).

```
ADD   AREG, ='5'   ⇒        ADD    AREG, FIVE
                            -- --
                    FIVE    DC     '5'
```

(a) (b)

Fig. 4.4 Use of literals in an assembly program

A *literal* is an operand with the syntax ='<*value*>'. It differs from a constant because its location cannot be specified in the assembly program. This helps to ensure that its value is not changed during execution of a program. It differs from an immediate operand because no architectural provision is needed to support its use. An assembler handles a literal by mapping its use into other features of the assembly language. Figure 4.4(a) shows use of a literal ='5'. Figure 4.4(b) shows an equivalent arrangement using a DC statement `FIVE  DC  '5'`. When the assembler encounters the use of a literal in the operand field of a statement, it handles the literal using an arrangement similar to that shown in Fig. 4.4(b)—it allocates a memory word to contain the value of the literal, and replaces the use of the literal in a statement by an operand expression referring to this word. The value of the literal is protected by the fact that the name and address of this word is not know to the assembly language programmer.

Assembler directives

Assembler directives instruct the assembler to perform certain actions during the assembly of a program. Some assembler directives are described in the following.

```
START        <constant>
```

This directive indicates that the first word of the target program generated by the assembler should be placed in the memory word with address <*constant*>.

```
END          [<operand spec>]
```

This directive indicates the end of the source program. The optional <*operand spec*> indicates the address of the instruction where the execution of the program should begin. (By default, execution begins with the first instruction of the assembled program.)

4.1.2 Advantages of Assembly Language

The primary advantages of assembly language programming vis-a-vis machine language programming arise from the use of symbolic operand specifications. Consider

the machine and assembly language statements of Fig. 4.3 once again. The programs presently compute N!. Figure 4.5 shows a changed program to compute $\frac{1}{2} \times$N!, where rectangular boxes are used to highlight changes in the program. One statement has been inserted before the PRINT statement to implement division by 2. In the machine language program, this leads to changes in addresses of constants and reserved memory areas. Because of this, addresses used in most instructions of the program had to change. Such changes are not needed in the assembly program since operand specifications are symbolic in nature.

```
          START   101
          READ    N              101)  + 09 0 114
          MOVER   BREG, ONE      102)  + 04 2 116
          MOVEM   BREG, TERM     103)  + 05 2 117
AGAIN     MULT    BREG, TERM     104)  + 03 2 117
          MOVER   CREG, TERM     105)  + 04 3 117
          ADD     CREG, ONE      106)  + 01 3 116
          MOVEM   CREG, TERM     107)  + 05 3 117
          COMP    CREG, N        108)  + 06 3 114
          BC      LE, AGAIN      109)  + 07 2 104
          DIV     BREG, TWO      110)  + 08 2 118
          MOVEM   BREG, RESULT   111)  + 05 2 115
          PRINT   RESULT         112)  + 10 0 115
          STOP                   113)  + 00 0 000
N         DS      1              114)
RESULT    DS      1              115)
ONE       DC      '1'            116)  + 00 0 001
TERM      DS      1              117)
TWO       DC      '2'            118)  + 00 0 001
          END
```

Fig. 4.5 Modified assembly and machine language programs

Assembly language programming holds an edge over HLL programming in situations where it is necessary or desirable to use specific architectural features of a computer—for example, special instructions supported by the CPU.

4.2 A SIMPLE ASSEMBLY SCHEME

The fundamental translation model is motivated by Definition 1.2. In this section we use this model to develop preliminary ideas on the design of an assembler. We will use these ideas in Sections 4.4 and 4.5.

Design specification of an assembler

We use a four step approach to develop a design specification for an assembler:

1. Identify the information necessary to perform a task.

2. If a label is present, enter the pair (*symbol*, <*LC contents*>) in a new entry of symbol table.
3. Check validity of the mnemonic opcode through a look-up in the Mnemonics table.
4. Perform LC processing, i.e. update the value contained in LC by considering the opcode and operands of the statement.

Synthesis phase

1. Obtain the machine opcode corresponding to the mnemonic from the Mnemonics table.
2. Obtain address of a memory operand from the Symbol table.
3. Synthesize a machine instruction or the machine form of a constant, as the case may be.

4.3 PASS STRUCTURE OF ASSEMBLERS

In Section 1.3 we have defined a pass of a language processor as one complete scan of the source program, or its equivalent representation (see Definition 1.4). We discuss two pass and single pass assembly schemes in this section.

Two pass translation

Two pass translation of an assembly language program can handle forward references easily. LC processing is performed in the first pass and symbols defined in the program are entered into the symbol table. The second pass synthesizes the target form using the address information found in the symbol table. In effect, the first pass performs analysis of the source program while the second pass performs synthesis of the target program. The first pass constructs an intermediate representation (IR) of the source program for use by the second pass (see Fig. 4.7). This representation consists of two main components—data structures, e.g. the symbol table, and a processed form of the source program. The latter component is called *intermediate code* (IC).

Single pass translation

LC processing and construction of the symbol table proceed as in two pass translation. The problem of forward references is tackled using a process called *backpatching*. The operand field of an instruction containing a forward reference is left blank initially. The address of the forward referenced symbol is put into this field when its definition is encountered. In the program of Fig. 4.3, the instruction corresponding to the statement

```
MOVER    BREG, ONE
```

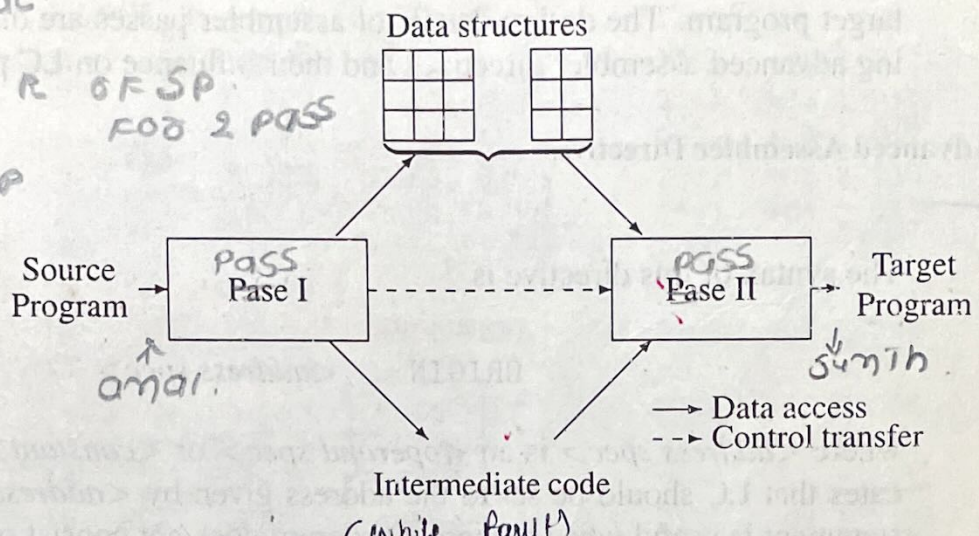

Fig. 4.7 Overview of two pass assembly

can be only partially synthesized since ONE is a forward reference. Hence the instruction opcode and address of BREG will be assembled to reside in location 101. The need for inserting the second operand's address at a later stage can be indicated by adding an entry to the Table of Incomplete Instructions (TII). This entry is a pair (<*instruction address*>, <*symbol*>), e.g. (101, ONE) in this case.

By the time the END statement is processed, the symbol table would contain the addresses of all symbols defined in the source program and TII would contain information describing all forward references. The assembler can now process each entry in TII to complete the concerned instruction. For example, the entry (101, ONE) would be processed by obtaining the address of ONE from symbol table and inserting it in the operand address field of the instruction with assembled address 101. Alternatively, entries in TII can be processed in an incremental manner. Thus, when definition of some symbol *symb* is encountered, all forward references to *symb* can be processed.

4.4 DESIGN OF A TWO PASS ASSEMBLER

Tasks performed by the passes of a two pass assembler are as follows:

Pass I
1. Separate the symbol, mnemonic opcode and operand fields.
2. Build the symbol table.
3. Perform LC processing.
4. Construct intermediate representation.

Pass II Synthesize the target program.

Pass I performs analysis of the source program and synthesis of the intermediate representation while Pass II processes the intermediate representation to synthesize the

target program. The design details of assembler passes are discussed after introducing advanced assembler directives and their influence on LC processing.

4.4.1 Advanced Assembler Directives

ORIGIN

The syntax of this directive is

ORIGIN <*address spec*>

where <*address spec*> is an <*operand spec*> or <*constant*>. This directive indicates that LC should be set to the address given by <*address spec*>. The ORIGIN statement is useful when the target program does not consist of consecutive memory words. The ability to use an <*operand spec*> in the ORIGIN statement provides the ability to perform LC processing in a *relative* rather than *absolute* manner. Example 4.1 illustrates the differences between the two.

Example 4.1 Statement number 18 of Fig. 4.8(a), viz. ORIGIN LOOP+2, sets LC to the value 204, since the symbol LOOP is associated with the address 202. The next statement, viz.

MULT CREG, B

is therefore given the address 204. The statement ORIGIN LAST+1 sets LC to address 217. Note that an equivalent effect could have been achieved by using the statements ORIGIN 202 and ORIGIN 217 at these two places in the program, however the absolute addresses used in these statements would need to be changed if the address specification in the START statement is changed.

EQU

The EQU statement has the syntax

<*symbol*> EQU <*address spec*>

where <*address spec*> is an <*operand spec*> or <*constant*>.

The EQU statement defines the symbol to represent <*address spec*>. This differs from the DC/DS statement as no LC processing is implied. Thus EQU simply associates the name <*symbol*> with <*address spec*>.

Example 4.2 Statement 22 of Fig. 4.8(a), viz. BACK EQU LOOP introduces the symbol BACK to represent the operand LOOP. This is how the 16^{th} statement, viz.

BC LT, BACK

is assembled as '+ 07 1 202'.

(Jump)

```
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. 4.8 An assembly program illustrating ORIGIN

LTORG

Fig. 4.4 has shown how literals can be handled in two steps. First, the literal is treated as if it is a <*value*> in a DC statement, i.e. a memory word containing the value of the literal is formed. Second, this memory word is used as the operand in place of the literal. Where should the assembler place the word corresponding to the literal? Obviously, it should be placed such that control never reaches it during the execution of a program. The LTORG statement permits a programmer to specify where literals should be placed. By default, assembler places the literals after the END statement.

At every LTORG statement, as also at the END statement, the assembler allocates memory to the literals of a *literal pool*. The pool contains all literals used in the program since the start of the program or since the last LTORG statement.

Example 4.3 In Fig. 4.8, the literals ='5' and ='1' are added to the literal pool in statements 2 and 6, respectively. the first LTORG statement (statement number 13) allocates the addresses 211 and 212 to the values '5' and '1'. A new literal pool is now started. The value '1' is put into this pool in statement 15. This value is allocated the address 219 while processing the END statement. The literal ='1' used in statement 15 therefore refers to location 219 of the second pool of literals rather than location 212 of the first pool. Thus, all references to literals are forward references by definition.

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

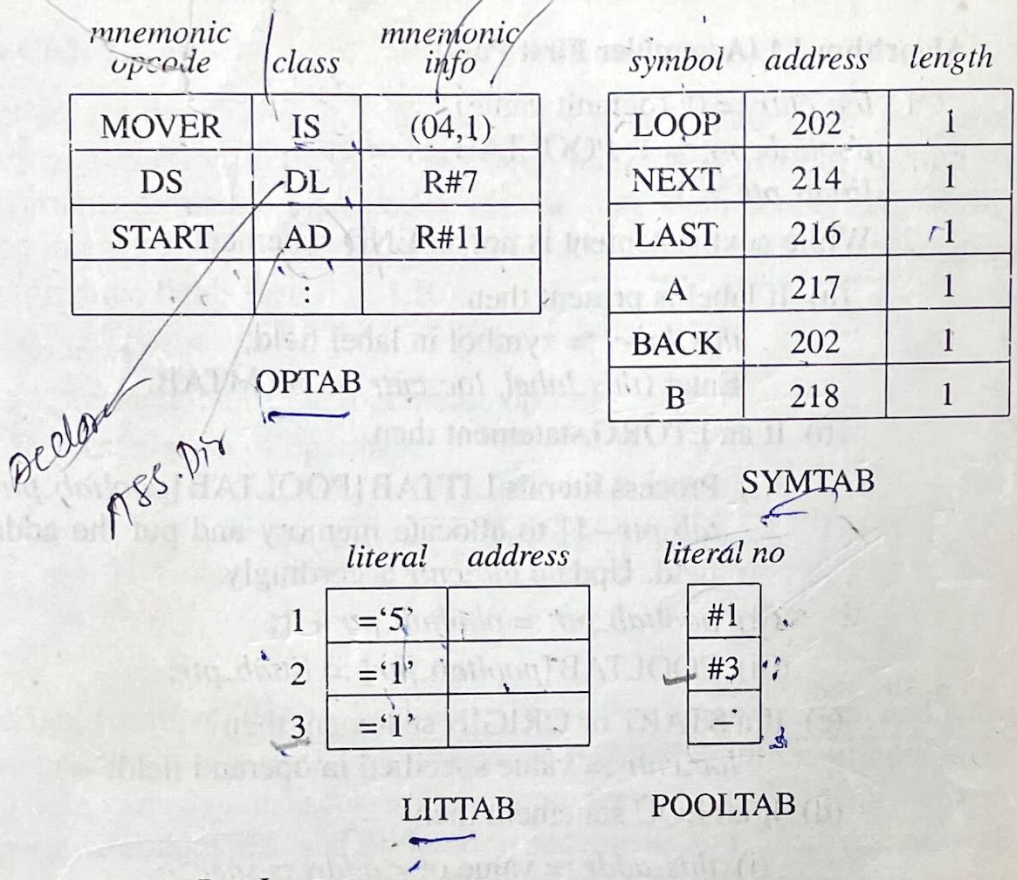

Fig. 4.9 Data structures of assembler Pass I

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.

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).
   (b) Generate IC '(AD,02)'.
   (c) Go to Pass II.

4.4.3 Intermediate Code Forms

In Section 1.3 two criteria for the choice of intermediate code, viz. processing efficiency and memory economy, have been mentioned. In this section we consider some variants of intermediate codes and compare them on the basis of these criteria.

The intermediate code consists of a set of IC units, each IC unit consisting of the following three fields (see Fig. 4.10):

1. Address
2. Representation of the mnemonic opcode
3. Representation of operands.

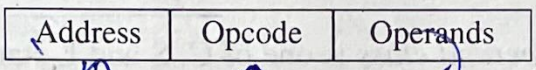

Fig. 4.10 An IC unit

Variant forms of intermediate codes, specifically the operand and address fields, arise in practice due to the tradeoff between processing efficiency and memory economy. These variants are discussed in separate sections dealing with the representation of imperative statements, and declaration statements and directives, respectively. The information in the mnemonic field is assumed to have the same representation in all the variants.

Mnemonic field

The mnemonic field contains a pair of the form

(*statement class, code*)

where *statement class* can be one of IS, DL and AD standing for imperative statement, declaration statement and assembler directive, respectively. For an imperative statement, *code* is the instruction opcode in the machine language. For declarations and assembler directives, *code* is an ordinal number within the class. Thus, (AD, 01) stands for assembler directive number 1 which is the directive `START`. Figure 4.11 shows the codes for various declaration statements and assembler directives.

Declaration statements		*Assembler directives*	
DC	01	START	01
DS	02	END	02
		ORIGIN	03
		EQU	04
		LTORG	05

Fig. 4.11 Codes for declaration statements and directives

4.4.4 Intermediate Code for Imperative Statements

We consider two variants of intermediate code which differ in the information contained in their operand fields. For simplicity, the address field is assumed to contain identical information in both variants.

Variant I

The first operand is represented by a single digit number which is a code for a register (1-4 for AREG-DREG) or the condition code itself (1-6 for LT-ANY). The second operand, which is a memory operand, is represented by a pair of the form

(*operand class, code*)

where *operand class* is one of C, S and L standing for constant, symbol and literal, respectively (see Fig. 4.12). For a constant, the *code* field contains the internal representation of the constant itself. For example, the operand descriptor for the statement START 200 is (C, 200). For a symbol or literal, the *code* field contains the ordinal number of the operand's entry in SYMTAB or LITTAB. Thus entries for a symbol XYZ and a literal ='25' would be of the form (S, 17) and (L, 35) respectively.

```
        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,01)
        STOP                 (IS,00)
A       DS      1            (DL, 02)  (C,1)
        LTORG                (AD,05)
          :                    :
```

Fig. 4.12 Intermediate code - variant I

Note that this method of representing symbolic operands gives rise to one peculiarity. We have so far assumed that an entry is made in SYMTAB only when a symbol occurs in the label field of an assembly statement, e.g. an entry (A, 345, 1) if symbol A is allocated one word at address 345. However, while processing a forward reference

```
MOVER AREG, A
```

it is necessary to enter A in SYMTAB, say in entry number *n*, so that it can be represented by (S, *n*) in IC. At this point, the *address* and *length* fields of A's entry cannot be filled in. This implies that two kinds of entries may exist in SYMTAB at any time—for defined symbols and for forward references. This fact should be noted for use during error detection (see Section 4.4.7).

Variant II

This variant differs from variant I of the intermediate code in that the operand fields of the source statements are selectively replaced by their processed forms (see Fig. 4.13). For declarative statements and assembler directives, processing of the operand fields is essential to support LC processing. Hence these fields contain the processed forms. For imperative statements, the operand field is processed only to identify literal references. Literals are entered in LITTAB, and are represented as (L, *m*) in IC. Symbolic references in the source statement are not processed at all during Pass I.

```
        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                   (AD,05)
        ...                     ...
```

Fig. 4.13 Intermediate code - variant II

Comparison of the variants

Variant I of the intermediate code appears to require extra work in Pass I since operand fields are completely processed. However, this processing considerably simplifies the tasks of Pass II—a look at the IC of Fig. 4.12 confirms this. The functions of Pass II are quite trivial. To process the operand field of a declaration statement, we only need to refer to the appropriate table and obtain the operand address. Most declarations do not require any processing, e.g. `DC`, `DS` (see Section 4.4.5), and `START` statements, while some, e.g. `LTORG`, require marginal processing. The IC is quite compact—it can be as compact as the target code itself if each operand reference like (S, *n*) can be represented in the same number of bits as an operand address in a machine instruction.

Variant II reduces the work of Pass I by transferring the burden of operand processing from Pass I to Pass II of the assembler. The IC is less compact since the memory operand of a typical imperative statement is in the source form itself. On the other hand, by making Pass II to perform more work, the functions and memory requirements of the two passes get better balanced. Figure 4.14 illustrates the advantages of this aspect. Part (a) of Fig. 4.14 shows memory utilization by an assembler using variant I of IC. Some data structures, viz. symbol table, are passed in the memory while IC is presumably written in a file. Since Pass I performs much more processing than Pass II, its code occupies more memory than the code of Pass II. Part

(b) of Fig. 4.14 shows memory utilization when variant II of IC is used. The code sizes of the two passes are now comparable, hence the overall memory requirement of the assembler is lower.

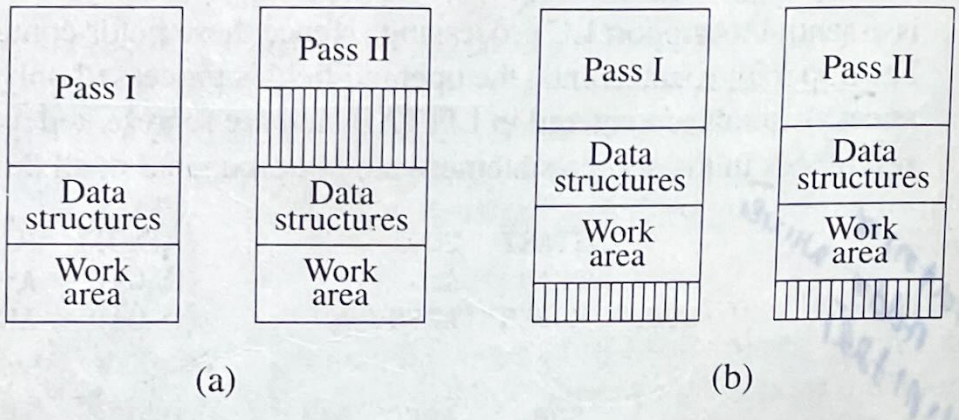

Fig. 4.14 Memory requirements using (a) variant I, (b) variant II

Variant II is particularly well-suited if expressions are permitted in the operand fields of an assembly statement. For example, the statement

```
MOVER      AREG, A+5
```

would appear as

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

in variant I of IC. This does not particularly simplify the task of Pass II or save much memory space. In such situations, it would have been preferable not to have processed the operand field at all.

4.4.5 Processing of Declarations and Assembler Directives

The focus of this discussion is on identifying alternative ways of processing declaration statements and assembler directives. In this context, it is useful to consider how far these statements can be processed in Pass I of the assembler. This depends on answers to two related questions:

1. Is it necessary to represent the address of each source statement in IC?
2. Is it necessary to have an explicit representation of DS statements and assembler directives in IC?

Let the answer to the first question be 'yes'. Now consider the following source program fragment and its intermediate code:

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

Here, it is redundant to have the representations of the START and DS statements in IC, since the effect of these statements is implied in the fact that the DC statement has the address 220 ! Thus, it is not necessary to have a representation for DS statements and assembler directives in IC if the IC contains an *address* field. If the *address* field of the IC is omitted, a representation for the DS statements and assembler directives becomes essential. Now, Pass II can determine the address for SIZE only after analyzing the intermediate code units for the START and DS statements. The first alternative avoids this processing but requires the existence of the address field. Yet another instance of space–time tradeoff !

DC statement

A DC statement must be represented in IC. The mnemonic field contains the pair (DL,01). The operand field may contain the value of the constant in the source form or in the internal machine representation. No processing advantage exists in either case since conversion of the constant into the machine representation is required anyway. If a DC statement defines many constants, e.g.

```
DC      '5, 3, -7'
```

a series of (DL,01) units can be put in the IC.

START and ORIGIN

These directives set new values into the LC. It is not necessary to retain START and ORIGIN statements in the IC if the IC contains an address field.

LTORG

Pass I checks for the presence of a literal reference in the operand field of every statement. If one exists, it enters the literal in the current literal pool in LITTAB. When an LTORG statement appears in the source program, it assigns memory addresses to the literals in the current pool. These addresses are entered in the *address* field of their LITTAB entries.

After performing this fundamental action, two alternatives exist concerning Pass I processing. (Pass I could simply construct an IC unit for the LTORG statement and leave all subsequent processing to Pass II. Values of literals can be inserted in the target program when this IC unit is processed in Pass II. This requires the use of POOLTAB and LITTAB in a manner analogous to Pass I.)

Example 4.4 Figure 4.9 shows the LITTAB and POOLTAB for the program of Fig. 4.8 at the end of Pass I. Literals of the first pool are copied into the target program when the IC unit for LTORG is encountered in Pass II. Literals of the second pool are copied into the target program when the IC unit for END is processed.

Alternatively, Pass I could itself copy out the literals of the pool into the IC. This avoids duplication of Pass I actions in Pass II. The IC for a literal can be made

identical to the IC for a DC statement so that no special processing is required in Pass II.

Example 4.5 Figure 4.15 shows the IC for the first half of the program of Fig. 4.8. The literals of the first pool (see Fig. 4.9) are copied out at LTORG statement. Note that the opcode field of the IC units, i.e (DL,01), is same as that for DC statements.

```
        START   200             (AD,01)   (C,200)
        MOVER   AREG, ='5'      (IS,04)   (1)(L,01)
        MOVEM   AREG, A         (IS,05)   (1)(S,01)
LOOP    MOVER   AREG, A         (IS,04)   (1)(S,01)
          :
        BC      ANY, NEXT       (IS,07)   (6)(S,04)
        LTORG                   (DL,01)   (C,5)
                                (DL,01)   (C,1)
          :
```

Fig. 4.15 Copying of literal values into intermediate code

However, this alternative increases the tasks to be performed by Pass I, consequently increasing its size. This might lead to an unbalanced pass structure for the assembler with the consequences illustrated in Fig. 4.14. Secondly, the literals have to exist in two forms simultaneously, in the LITTAB along with the address information, and also in the intermediate code.

EXERCISE 4.4

1. Given the following source program:

```
100   A       START   100
103   L1      DS      3
              MOVER   AREG, B
              ADD     AREG, C
              MOVEM   AREG, D
105           EQU     A+1
106   L2      PRINT   D

              ORIGIN  A-1
99    C       DC      '5'
              ORIGIN  L2+1
107           STOP
108   B       DC      '19'
              END     L1
```

   (a) Show the contents of the symbol table at the end of Pass I.

   (b) Explain the significance of EQU and ORIGIN statements in the program and explain how they are processed by the assembler.

   (c) Show the Intermediate code generated for the program.

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.

Output interface of the assembler

It has been assumed that the assembler produces a target program which is the machine language of the target computer. This is rarely (if ever !) the case. The assembler produces an *object module* in the format required by a linkage editor or loader. The information contained in object modules is discussed in Chapter 7.

4.4.7 Listing and Error Reporting

Design of an error indication scheme involves some decisions which influence the effectiveness of error reporting and the speed and memory requirements of the assembler. The basic decision is whether to produce program listing and error reports in Pass I or delay these actions until Pass II. Producing the listing in the first pass has the advantage that the source program need not be preserved till Pass II. This conserves memory and avoids some amount of duplicate processing.

This design decision also has very important implications from a programmer's viewpoint. A listing produced in Pass I can report only certain errors in the most relevant place, that is, against the source statement itself. Examples of such errors are syntax errors like missing commas or parentheses and semantic errors like duplicate definitions of symbols. Other errors like references to undefined variables can only be reported at the end of the source program (see Fig. 4.16). The target code can be printed later in Pass II, however it is difficult to locate the target code corresponding to a source statement and vice versa. All these factors make debugging difficult.

```
Sr. No.      Statement          Address
001          START  200
002          MOVER  AREG, A     200
               :
003
009          MVER   BREG, A     207
         ** error ** Invalid opcode
010          ADD    BREG, B     208
014      A   DS     1           209
               :
015
021      A   DC     '5'         227
         ** error ** Duplicate definition of symbol A
               :
022
035          END
         ** error ** Undefined symbol B in statement 10
```

Fig. 4.16 Error reporting in pass I

For effective error reporting, it is necessary to report all errors against the erro-

neous statement itself. This can be achieved by delaying program listing and error reporting till Pass II. Now the error reports as well as the target code can be printed against each source statement (see Ex. 4.6).

Example 4.6 Figure 4.16 illustrates error reporting in Pass I. Detection of errors in statements 9 and 21 is straightforward. In statement 9, the opcode is known to be invalid because it does not match with any mnemonic in OPTAB. In statement 21, A is known to be a duplicate definition because an entry for A already exists in the symbol table. Use of the undefined symbol B is harder to detect because at the end of Pass I we have no record that a forward reference to B exists in statement 10. This problem can be resolved by making an entry for B in the symbol table with an indication that a forward reference to B exists in statement 10. All such entries would be processed at the end of Pass I to check if a definition of the symbol has been encountered. If not, the symbol table entry contains sufficient information for error reporting. Note that the target instructions cannot be printed because they have not yet been generated. The memory address is printed against each statement in a weak attempt to provide a cross-reference between source statements and target instructions.

Example 4.7 Figure 4.17 illustrates error reporting performed in Pass II. Indication of errors in statements 9 and 21 is as easy as in Ex. 4.6. Indication of the error in statement 10 is equally easy—the symbol table is searched for an entry of B and an error is reported when no matching entry is found. Note that target program instructions appear against the source statements to which they belong.

```
Sr. No.      Statement              Address   Instruction
001          START  200
002          MOVER  AREG, A         200       + 04 1 209
               :
003
009          MVER   BREG, A         207       + -- 2 209
          ** error ** Invalid opcode
010          ADD    BREG, B         208       + 01 2 ---
          ** error ** Undefined symbol B in operand field
014     A    DS     1               209
               :
015
021     A    DC     '5'             227       + 00 0 005
          ** error ** Duplicate definition of symbol A
               :
022
035          END
```

g. 4.17 Error reporting in pass II

EXERCISE 4.4.7

1. A two pass assembler performs program listing and error reporting in Pass II using the following strategy: Errors detected in Pass I are stored in an error table. These are reported along with Pass II errors while producing the program listing.

   (a) Design the error table for use by Pass I. What is its entry format? What is the table organization?

   (b) Let the error messages (e.g. `DUPLICATE LABEL...`) be stored in an error message table. Comment on the organization of this table.

   (*Note:* Readers may refer to Dhamdhere (1983) for some interesting error reporting strategies.)

4.4.8 Some Organizational Issues

We discuss some organizational issues in assembler design, like the placement and access of tables and IC, with respect to the schematic shown in Fig. 4.18.

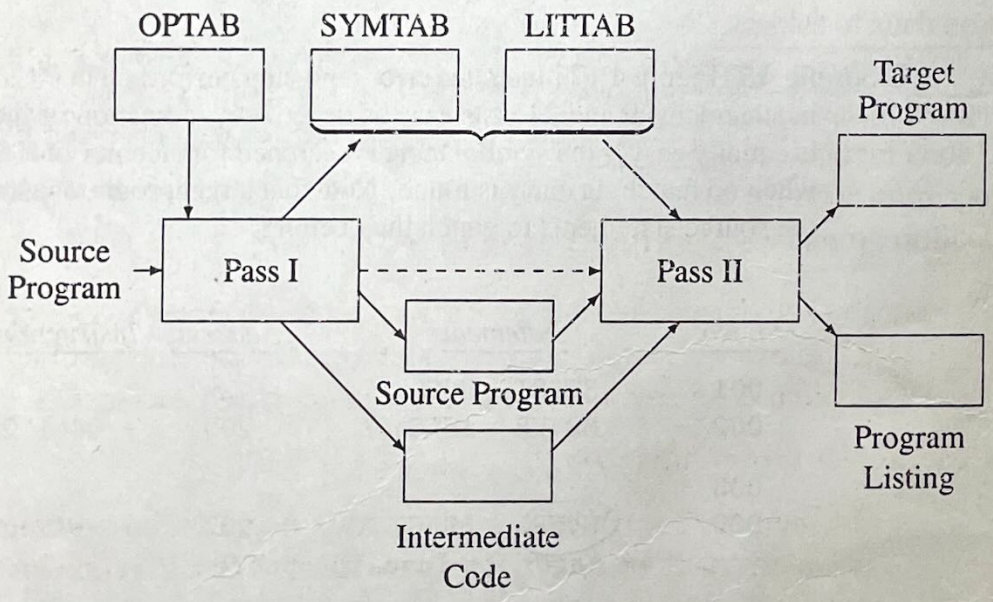

Fig. 4.18 Data structures and files in a two pass assembler

Tables

For efficiency reasons SYMTAB must remain in main memory throughout Passes I and II of the assembler. LITTAB is not accessed as frequently as SYMTAB, however it may be accessed sufficiently frequently to justify its presence in the memory. If memory is at a premium, it is possible to hold only part of LITTAB in the memory because only the literals of the current pool need to be accessible at any time. For obvious reasons, no such partitioning is feasible for SYMTAB. OPTAB should be in memory during Pass I.

Source program and intermediate code

The source program would be read by Pass I on a statement by statement basis. After processing, a source statement can be written into a file for subsequent use in Pass II. The IC generated for it would also be written into another file. The target code and the program listings can be written out as separate files by Pass II. Since all these files are sequential in nature, it is beneficial to use appropriate blocking and buffering of records.

EXERCISE 4.4.8

1. Develop complete program specifications for the passes of a two pass assembler indicating

   (a) Tables for internal use of the passes
   (b) Tables to be shared between passes
   (c) Inputs (files and tables) for every pass
   (d) Outputs (files and tables) of every pass.

   You must clearly specify why certain information is in the form of tables in main memory while other information is in the form of files.
2. Recommend appropriate organizations for the tables and files used in the two pass assembler of problem 1.

4.5 A SINGLE PASS ASSEMBLER FOR IBM PC

We shall discuss a single pass assembler for the intel 8088 processor used in the IBM PC. The discussion focuses on the design features for handling the forward reference problem in an environment using segment-based addressing.

4.5.1 The architecture of Intel 8088

The intel 8088 microprocessor supports 8 and 16 bit arithmetic, and also provides special instructions for string manipulation. The CPU contains the following features (see Fig. 4.19):

- Data registers AX, BX, CX and DX
- Index registers SI and DI
- Stack pointer registers BP and SP
- Segment registers Code, Stack, Data and Extra.

Each data register is 16 bits in size, split into the upper and lower halves. Either half can be used for 8 bit arithmetic, while the two halves together constitute the data register for 16 bit arithmetic. The architecture supports stacks for storing subroutine and interrupt return addresses, parameters and other data. The index registers SI and DI are used to index the source and destination addresses in string manipulation instructions. They are provided with the auto-increment and auto-decrement facility.