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.

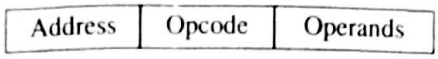


Fig. 4.10 An IC unit

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

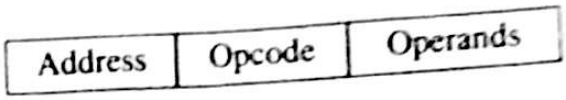
Fig. 4.11 Codes for declaration statements and directives

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.



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

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 c tained in their operand fields. For simplicity, the address field is assumed to con identical information in both variants.

Variant I

The first operand is represented by a single digit number which is a code for a r ister (1-4 for AREG-DREG) or the condition code itself (1-6 for LT-ANY). The seco 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 liter respectively (see Fig. 4.12). For a constant, the *code* field contains the internal repr sentation of the constant itself. For example, the operand descriptor for the stateme START 200 is (C, 200). For a symbol or literal, the *code* field contains the ordin number of the operand's entry in SYMTAB or LITTAB. Thus entries for a symb 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,02)
        STOP                 (IS,00)
A       DS      1            (DL, 02)  (C,1)
        LTORG                (DL,05)
        ...                  ...
```

Fig. 4.12 Intermediate code - variant I

Note that this method of representing symbolic operands gives rise to one pe culiarity. 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 repre sented 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).

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                   (DL,05)
        ...                     ...
```

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.

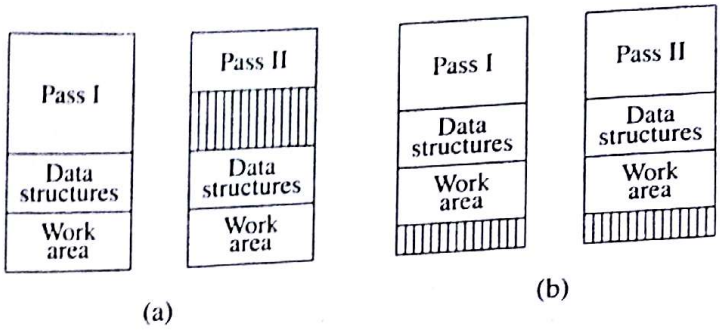


Fig. 4.14 Memory requirements using (a) variant I, (b) variant II

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 Pa II.

Example 4.5 Figure 4.15 shows the IC for the first half of the program of Fig. 4.8. T literals of the first pool (see Fig. 4.9) are copied out at LTORG statement. Note that t 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:

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

        ORIGIN  A-1
C       DC      '5'
        ORIGIN  L2+1
        STOP
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 m chine language of the target computer. This is rarely (if ever !) the case. The asso bler produces an *object module* in the format required by a linkage editor or load 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 th effectiveness of error reporting and the speed and memory requirements of the a sembler. The basic decision is whether to produce program listing and error repor in Pass I or delay these actions until Pass II. Producing the listing in the first pa has the advantage that the source program need not be preserved till Pass II. Thi conserves memory and avoids some amount of duplicate processing.

This design decision also has very important implications from a programmer' viewpoint. A listing produced in Pass I can report only certain errors in the mo relevant place, that is, against the source statement itself. Examples of such errors ar syntax errors like missing commas or parentheses and semantic errors like duplicat definitions of symbols. Other errors like references to undefined variables can onl be reported at the end of the source program (see Fig. 4.16). The target code can b printed later in Pass II, however it is difficult to locate the target code correspondin 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-

...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
```

Fig. 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.

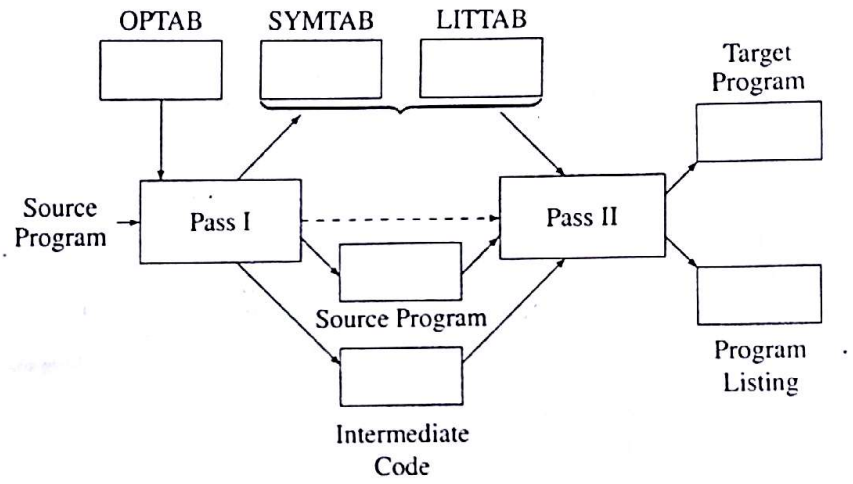


Fig. 4.18 Data structures and files in a two pass assembler

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.

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.

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.

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.