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.

A simple 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`	

Fig. 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 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.

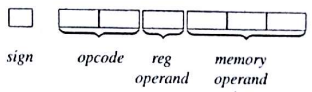


Fig. 4.2 Instruction format

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

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. Design a suitable data structure to record the information.
3. Determine the processing necessary to obtain and maintain the information.
4. Determine the processing necessary to perform the task.

The fundamental information requirements arise in the synthesis phase of an assembler. Hence it is best to begin by considering the information requirements of the synthesis tasks. We then consider how to make this information available, i.e. whether it should be collected during analysis or derived during synthesis.

Synthesis phase

Consider the assembly statement

```
MOVER    BREG, ONE
```

in Fig. 4.3. We must have the following information to synthesize the machine instruction corresponding to this statement:

1. Address of the memory word with which name `ONE` is associated,
2. Machine operation code corresponding to the mnemonic `MOVER`.

The first item of information depends on the source program. Hence it must be made available by the analysis phase. The second item of information does not depend on the source program, it merely depends on the assembly language. Hence the synthesis phase can determine this information for itself.

Based on the above discussion, we consider the use of two data structures during the synthesis phase:

1. Symbol table
2. Mnemonics table.

Each entry of the symbol table has two primary fields—*name* and *address*. The table is built by the analysis phase. An entry in the mnemonics table has two primary fields—*mnemonic* and *opcode*. The synthesis phase uses these tables to obtain the machine address with which a name is associated, and the machine opcode corresponding to a mnemonic, respectively. Hence the tables have to be searched with the symbol name and the mnemonic as keys.

Analysis phase

The primary function performed by the analysis phase is the building of the symbol table. For this purpose it must determine the addresses with which the symbolic names used in a program are associated. It is possible to determine some addresses directly, e.g. the address of the first instruction in the program, however others must be inferred. Consider the assembly program of Fig. 4.3. To determine the address of

N, we must fix the addresses of all program elements preceding it. This function is called *memory allocation*.

To implement memory allocation a data structure called *location counter* (LC) is introduced. The location counter is always made to contain the address of the next memory word in the target program. It is initialized to the constant specified in the `START` statement. Whenever the analysis phase sees a label in an assembly statement, it enters the label and the contents of LC in a new entry of the symbol table. It then finds the number of memory words required by the assembly statement and updates the LC contents. (Hence the word 'counter' in 'location counter'.) This ensures that LC points to the next memory word in the target program even when machine instructions have different lengths and `DS/DC` statements reserve different amounts of memory. To update the contents of LC, analysis phase needs to know lengths of different instructions. This information simply depends on the assembly language, hence the mnemonics table can be extended to include this information in a new field called *length*. We refer to the processing involved in maintaining the location counter as *LC processing*.

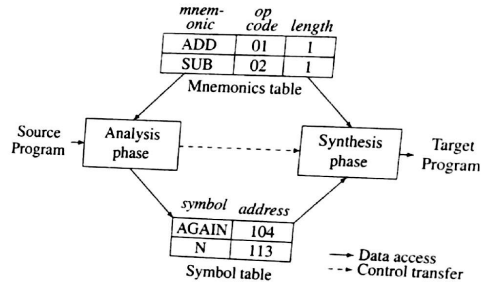


Fig. 4.6 Data structures of the assembler

Figure 4.6 illustrates the use of the data structures by the analysis and synthesis phases. Note that the Mnemonics table is a fixed table which is merely accessed by the analysis and synthesis phases, while the Symbol table is constructed during analysis and used during synthesis. The tasks performed by the analysis and synthesis phases are as follows:

Analysis phase
1. Isolate the label, mnemonic opcode and operand fields of a statement.

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
```

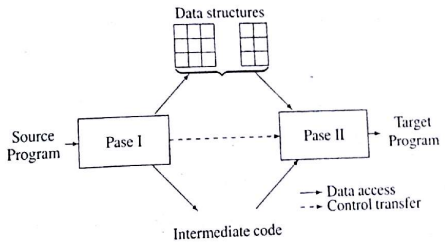



Fig. 4.7 Overview of two pass assembly

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 16th statement, viz.

BC LT, BACK

is assembled as '+ 07 1 202'.


```
 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

OPTAB

mnemonic opcode	*class*	*mnemonic info*
MOVER	IS	(04,1)
DS	DL	R#7
START	AD	R#11
	:	

SYMTAB

symbol	*address*	*length*
LOOP	202	1
NEXT	214	1
LAST	216	1
A	217	1
BACK	202	1
B	218	1

LITTAB

	literal	*address*
1	='5'	
2	='1'	
3	='1'	

POOLTAB

literal no
#1
#3
–

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.

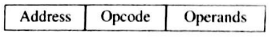


Fig. 4.10 An IC unit

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	
`DC`	01
`DS`	02

Assembler directives	
`START`	01
`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,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 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                   (DL,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.

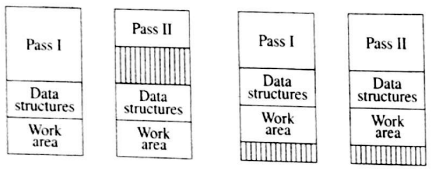


Fig. 4.14 Memory requirements using (a) variant I, (b) variant II

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:

```
            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 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 erroneous 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.

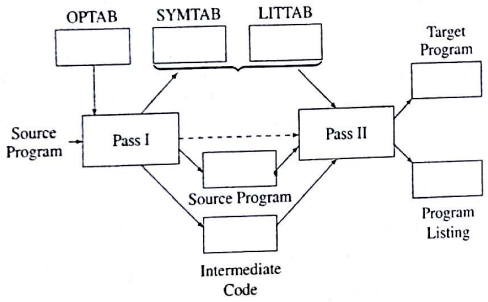


Fig. 4.18 Data structures and files in a two pass assembler

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.

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

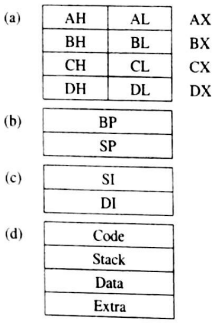


Fig. 4.19 (a) Data, (b) Base, (c) Index and (d) Segment registers

Two stack pointer registers called SP and BP are provided to address the stack. SP points into the stack implicitly used by the architecture to store subroutine and interrupt return addresses. BP can be used by the programmer in any desired manner. Push and Pop instructions are provided for this purpose.

The Intel 8088 provides addressing capability for 1 MB of primary memory. The memory is used to store three components of a program, program code, data and stack. The Code, Stack and Data segment registers are used to contain the start addresses of these three components. The Extra segment register points to another memory area which can be used to store data. To address a memory location, an instruction designates a segment register and provides a 16 bit logical address. The address contained in the segment register is extended by adding four lower order zeroes to yield the segment base address. The logical address is now added to it to obtain a 20 bit memory address. The size of each segment is thus limited to 2^{16}, i.e. 64K, bytes. A large program may contain many segments, only four of which can be addressed at any given time. Segment-based addressing facilitates easy relocation of programs. If the memory area allocated to a program is changed, it is sufficient to change the addresses loaded in the segment registers. The memory addresses generated would now automatically lie in the new memory area occupied by the program.

The 8088 architecture provides 24 addressing modes. These are summarized in Fig. 4.20. In the immediate addressing mode, the instruction itself contains the data that is to participate in the instruction. This data can be 8 or 16 bits in length. In the direct addressing mode, the instruction contains a 16 bit number which is taken to be a displacement from the segment base contained in a segment register. The segment

Addressing mode	*Example*	*Remarks*
Immediate	`MOV SUM, 1234H`	Data = 1234H
Register	`MOV SUM, AX`	AX contains the data
Direct	`MOV SUM, [1234H]`	Data disp. = 1234H
Register indirect	`MOV SUM, [BX]`	Data disp. = (BX)
Register indirect	`MOV SUM, CS: [BX]`	Segment override : Segment base = (CS) Data disp. = (BX)
Based	`MOV SUM, 12H[BX]`	Data disp. = 12H+(BX)
Indexed	`MOV SUM, 34H[SI]`	Data disp. = 34H+(SI)
Based & indexed	`MOV SUM, 56H[SI][BX]`	Data disp. `= 56H + (SI) + (BX)`

Fig. 4.20 Addressing modes of 8088 ('(..)' implies 'contents of')

register may be explicitly indicated in a prefix of the instruction, else a default segment register is used. In the indexed mode, contents of the index register indicated in the instruction (SI or DI) are added to the 8 or 16 bit displacement contained in the instruction. The result is taken to be the displacement from the segment base of the data segment. In the based mode, contents of the base register (BP or BX) are added to the displacement. The result is taken to be the displacement from the data segment base unless BP is specified, in which case it is taken as a displacement from the stack segment base. The based-and-indexed-with-displacement mode combines the effect of the based and indexed modes.

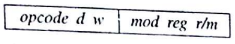
4.5.2 Intel 8088 Instructions

Arithmetic instructions

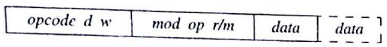
The operands can be in one of the four 16 bit registers, or in a memory location designated by one of the 24 addressing modes. Three instruction formats shown in Fig. 4.21 are supported. The *mod* and *r/m* fields specify the first operand, which can be in a register or in memory, while the *reg* field describes the second operand, which is always a register. The instruction opcode indicates which instruction format is applicable. The direction field (d) in the instruction indicates which operand is the destination operand in the instruction. If $d = 0$, the register/memory operand is the destination, else the register operand indicated by *reg* is the destination. The *width* field (w) indicates whether 8 or 16 bit arithmetic is to be used. The conventions for determining the operands are described in Fig. 4.21.

Figure 4.22 contains some sample instructions. Note that in the first statement, AL could be encoded into the first or the second operand of the instruction. In the second statement, however, it has to be encoded into second operand since the first operand has to be `12H[SI]`. Here, *mod* = 01 since only one byte of displacement is

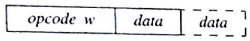
(a) *Register/Memory to Register*



(b) *Immediate to Register/Memory*



(c) *Immediate to Accumulator*



r/m	*mod* = 00	*mod* = 01	*mod* = 10	*mod* = 11	
				w=0	*w*=1
000	(BX) + (SI)	(BX) + (SI) + d8	Note 2	AL	AX
001	(BX) + (DI)	(BX) + (DI) + d8	Note 2	CL	CX
010	(BP) + (SI)	(BP) + (SI) + d8	Note 2	DL	DX
011	(BP) + (DI)	(BP) + (DI) + d8	Note 2	BL	BX
100	(SI)	(SI) + d8	Note 2	AH	SP
101	(DI)	(DI) + d8	Note 2	CH	BP
110	Note 1	(BP) + d8	Note 2	DH	SI
111	(BX)	(BX) + d8	Note 2	BH	DI

Note 1 : (BP) + DISP for indirect addressing, d16 for direct
Note 2 : Same as previous column, except d16 instead of d8

	Register	
reg	*8-bit* (*w*=0)	*16-bit* (*w*=1)
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Fig. 4.21 Instruction formats of Intel 8088

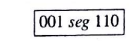
Assembly statement	*Opcode d w*	*mod reg r/m*	*data/ displacement*
`ADD AL, BL`	000000 0 0	11 011 000	
`ADD AL, 12H[SI]`	000000 1 0	01 000 100	00010010
`ADD AX, 3456H`	100000 0 1	11 000 000	01010110
			00110100
`ADD AX, 3456H`	000001 0 1	01 010 110	00110100

Fig. 4.22 Sample instructions of 8088

adequate. The third instruction contains 16 bits of immediate data. Note that the low byte of immediate data comes first, followed by its high byte. The fourth assembly statement is identical to the third, however it has been encoded using the immediate to accumulator instruction format. Here, w = 1 implies that the accumulator is the AX register. This instruction is only 3 bytes in length as against the previous instruction which is 4 bytes. This illustrates the fact that the assembler has to perform an analysis of the available options before determining the best instruction to use in a specific case.

Segment overrides

For arithmetic and `MOV` instructions, the architecture uses the data segment by default. To override this, an instruction can be preceded by a 1-byte segment override prefix with the following format:



where *seg*, represented in 2 bits, has the meanings shown in Fig. 4.23.

seg	*segment register*
00	ES
01	CS
10	SS
11	DS

Fig. 4.23 Segment codes

Example 4.8 If the code segment is to be used instead of the data segment in the second statement of Fig. 4.22, it can be rewritten as

```
ADD     AL, CS:12H[SI]
```

The assembler would encode this as

segment override	*instruction*
001 01 110	000000 1 0 01 000 100 00010010

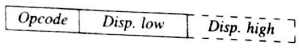
Control transfer instructions

Two groups of control transfer instructions are supported. These are:

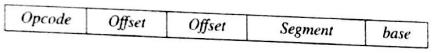
1. Calls, jumps and returns
2. Iteration control instructions.

The calls, jumps and returns can occur within the same segment, or can cross segment boundaries. Intra-segment transfers are preferably assembled using a self-relative displacement in the range of −128 to +127. The longer form of intra-segment transfer uses a 16 bit logical address within the segment. Inter-segment transfers indicate a new segment base and an offset. Their execution is expensive since the code segment register has to be modified. Control transfers can be both direct and indirect. Their instruction formats are as shown in Fig. 4.24.

(a) *Intrasegment*



(b) *Intersegment*



(c) *Indirect*

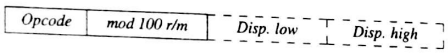


Fig. 4.24 Formats of control transfer instructions

A call pushes the offset of the next instruction's address from the segment base on the stack. This address is used to return to the calling program. In the case of an inter-segment call, the CS segment register is pushed first, followed by the offset.

Iteration control operations perform looping decisions in string operations. A self-relative jump of −128 to +127 permits the decision to be made at the beginning or end of the loop.

Example 4.9 Consider the program

```
MOV     SI, 100H  ; Source address
MOV     DI, 200H  ; Destination address
MOV     CX, 50H   ; No. of bytes
CLD               ; Clear direction flag
REP     MOVSB     ; Move 80 bytes
```

The MOVSB instruction moves one byte from the address contained in SI to the address contained in DI. After the move, the SI and DI registers are incremented/decremented by one byte depending on the direction flag. Since the CLD instruction resets the direction flag, the registers will be incremented in this case. The REP prefix of MOVSB will cause CX to be decremented after MOVSB is performed. The new value of CX is tested and the loop is repeated until CX becomes zero.

4.5.3 The Assembly Language of Intel 8088

Statement format

The format of an assembly statement is as follows:

```
[Label:]  opcode     operand(s)   ; comment string
```

where the label is optional. In the operand field, operands are separated by commas. Figure 4.22 contains some examples of assembly statements. The parentheses [..] in the operand field represent the words 'contents of'. Base and index register specifications, as also direct addresses specified as numeric offsets from the segment base are enclosed in these parentheses. A segment override is specified in the operand to which it applies, viz. `CS:12H[SI]`. More details of operand specification can be found in Sections 4.5.1 and 4.5.2.

Assembler directives

The ORG, EQU and END directives are analogous to the ORIGIN, EQU and END directives described in Sections 4.4 and 4.1. The start directive is not supported since ORG subsumes its functionality. The concept of literals, and the LTORG directive, are redundant since the 8088 architecture supports immediate operands.

Declarations

Declaration of constants and reservation of storage are both achieved in the same directive, viz.

```
A    DB    25          ;Reserve byte & initialize
B    DW    ?           ;Reserve word, no initialization
C    DD    6DUP(0)     ;6 Double words, all 0's
```

DQ and DT reserve a quad-word (8 bytes) and ten bytes respectively. Since logical addresses (16 bits) are required to occupy a word, the DW declaration is used for some special purposes, viz.

```
ADDR_A    DW        A
```

initializes the word to the logical address of A (i.e. offset from the segment base).

EQU *and* PURGE

EQU defines symbolic names to represent values or other symbolic names as described in Section 4.4. The names so defined can be 'undefined' through a PURGE statement. Such a name can be reused for other purposes later in the program.

Example 4.10 Following program illustrates EQU and PURGE.

```
XYZ     DB      ?
ABC     EQU     XYZ   ; ABC represents name XYZ
        PURGE   ABC   ; ABC no longer XYZ
ABC     EQU     25    ; ABC now stands for '25'
```

EGMENT, ENDS *and* ASSUME

All memory addressing is segment directed. Hence as assembly program consists of a set of segments. SEGMENT and ENDS directives demarcate the segments in an assembly program. To assemble a symbolic reference, the assembler must determine the offset of the symbol from the start of the segment containing it. To facilitate this, the programmer must perform the following actions in the assembly program: (a) load a segment register with the segment base, and (b) let the assembler know which segment register contains the segment base. The second task is performed using the ASSUME directive, which has the syntax

ASSUME *<register>* : *<segment name>*

and tells the assembler that it can 'assume' the address of the indicated segment to be present in *<register>*. The directive ASSUME *<register>* : NOTHING cancels any prior assumptions indicated for *<register>*.

Example 4.11 Consider the following program:

```
SAMPLE_DATA   SEGMENT
ARRAY         DW        100 DUP ?
SUM           DW        0
SAMPLE_DATA   ENDS
SAMPLE_CODE   SEGMENT

              ASSUME    DS:SAMPLE_DATA
HERE:         MOV       AX, SAMPLE_DATA
              MOV       DS, AX
              MOV       AX, SUM
              - -
SAMPLE_CODE   ENDS
              END       HERE
```

The program consists of two segments, a code segment and a data segment. The ASSUME directive tells the assembler that the start address of SAMPLE_DATA can be assumed to be in the DS register. While assembling the statement MOV AX, SUM the assembler first computes the offset of SUM from the start of its segment. This is 200 bytes. It now finds whether the segment in which SUM exists is addressable at the current moment. Since it is, it would encode SUM to be an offset of 200 bytes from the DS register. Note that it is the programmer's responsibility to ensure that the correct address is loaded in the DS register before executing the reference to SUM. If the address of SAMPLE_DATA were to be loaded into some other segment register, e.g. register ES, this fact would be indicated through the statement ASSUME ES:SAMPLE_DATA. The assembler would then generate a segment override prefix while assembling the statement MOV AX, SUM.

PROC, ENDP, NEAR *and* FAR

PROC and ENDP delimit the body of a procedure. The keywords NEAR and FAR appearing in the operand field of PROC indicate whether the call to the procedure is to be assembled as a *near* or *far* call. A RET statement must appear in the body of the procedure to return execution control to the calling program. Parameters for the called procedure can be passed through registers or on the stack.

Example 4.12 Consider the following assembly program:

```
SAMPLE_CODE   SEGMENT
CALCULATE     PROC      FAR         ; a FAR procedure
              - -
              RET
CALCULATE     ENDP
SAMPLE_CODE   ENDS
PGM           SEGMENT
              - -
              CALL      CALCULATE   ; a FAR call
              - -
PGM           ENDS
              END
```

Since CALCULATE is a far procedure, it need not be addressable at the point of call. The assembler will encode a far call instruction which specifies the segment base and the offset of CALCULATE within the segment.

PUBLIC *and* EXTRN

When a symbolic name declared in one assembly module is to be accessible in other modules, it is specified in a PUBLIC statement. Another module wishing to use this name must specify it in an EXTRN statement which has the syntax

EXTRN *<symbolic name>*:*<type>*

For labels of DC, DS statements, the type can be word, byte, etc. For labels of instructions, the type can be FAR or NEAR.

Analytic operators

The analytic operators split a memory address into its components, or provide information regarding the type and memory requirements of operands. Five analytic

operators exist. These are SEG, OFFSET, TYPE, SIZE and LENGTH. SEG and OFFSET provide the segment and offset components of the memory address of an operand.

Example 4.13 The instruction

```
MOV     AX, OFFSET ABC
```

loads the offset of symbol ABC within its segment into the AX register.

TYPE indicates the manner in which an operand is defined and returns the following numeric codes: 1 (byte), 2 (word), 4 (double word), 8 (quad-word), 10 (ten bytes), −1 (near instruction) and −2 (far instruction). SIZE indicates the number of units declared for an operand, while LENGTH indicates the number of bytes allocated to the operand.

Example 4.14 The symbol BUFFER defined in

```
BUFFER    DW        100 DUP (0)
```

has the SIZE of 100 and LENGTH of 200 bytes. SIZE and LENGTH can be used as in

```
MOV       CX, LENGTH XYZ
```

which loads length of XYZ into the CX register.

Synthetic operators

It is sometimes necessary to have different types associated with the same memory operand, e.g. when a byte in an operand of type 'word' is to be accessed as an operand of type 'byte'. This is achieved through the PTR and THIS operators. The PTR operator creates a new memory operand with the same segment and offset addresses as an existing operand, but having a different type. No memory allocation is implied by its use. The THIS operator performs the special function of creating a new memory operand with the same address as the next memory byte available for allocation.

Example 4.15 Consider the program

```
XYZ         DW      312
NEW_NAME    EQU     BYTE PTR XYZ
LOOP:       CMP     AX, 234
            JMP     LOOP
FAR_LOOP    EQU     FAR PTR LOOP
            JMP     FAR_LOOP
```

Here, NEW_NAME is a byte operand with the same address as XYZ, while FAR_LOOP is a FAR symbolic name with the same address as LOOP. Thus, while JMP LOOP is a near jump, JMP FAR_LOOP is a far jump. Exactly the same effect could be achieved by rewriting the program using THIS as follows:


```
            DW
NEW_NAME    EQU     THIS BYTE
XYZ         DW      312
FAR_LOOP    EQU     THIS FAR
LOOP        CMP     AX, 234
            JMP     LOOP
            -- --
            JMP     FAR_LOOP
```

4.5.4 Problems of Single Pass Assembly

A single pass assembler for Intel 8088 shares some problems with other single pass assemblers, viz. problems in assembling forward references and in error reporting. The forward reference problem is aggravated by the nature of the 8088 architecture. We discuss two aspects of this problem. The sample program of Fig. 4.25 is used to illustrate these problems.

```
Sr. No.                 Statement                       Offset
001     CODE    SEGMENT
002             ASSUME  CS:CODE, DS:DATA
003             MOV     AX, DATA                        0000
004             MOV     DS, AX                          0003
005             MOV     CX, LENGTH STRNG                0005
006             MOV     COUNT,0000                      0008
007             MOV     SI, OFFSET STRNG                0011
008             ASSUME  ES:DATA, DS:NOTHING
009             MOV     AX, DATA                        0014
010             MOV     ES, AX                          0017
011     COMP:   CMP     [SI],'A'                        0019
012             JNE     NEXT                            0022
013             MOV     COUNT, 1                        0024
014     NEXT:   INC     SI                              0027
015             DEC     CX                              0029
016             JNE     COMP                            0030
017     CODE    ENDS

018     DATA    SEGMENT
019             ORG     1
020     COUNT   DB      ?                               0001
021     STRNG   DW      50 DUP (?)                      0002
022     DATA    ENDS
023             END
```

Fig. 4.25 Sample assembly program of Intel 8088

Forward references

A symbolic name may be forward referenced in a variety of ways. When used as a data operand in a statement, its assembly is straightforward. An entry can be made in the table of incomplete instructions (TII) discussed in Section 4.3. This entry would identify the bytes in code where the address of the referenced symbol should be put. When the symbol's definition is encountered, this entry would be analysed to complete the instruction. However, use of a symbolic name as the destination in a branch instruction gives rise to a peculiar problem. Some generic branch opcodes like JMP in the 8088 assembly language can give rise to instructions of different formats and different lengths depending on whether the jump is *near* or *far*—that is, whether the destination symbol is less than 128 bytes away from the JMP instruction. However, this would not be known until sometime later in the assembly process! This problem is solved by assembling such instructions with a 16 bit logical address unless the programmer indicates a short displacement, e.g. JMP SHORT LOOP. The program of Fig. 4.25 contains the forward branch instruction JNE NEXT. However, the above problem does not arise here since the opcode JNE dictates that the instruction should be in the self-relative format.

A more serious problem arises when the type of a forward referenced symbol is used in an instruction. The type may be used in a manner which influences the size/length of a declaration. Such usage will have to be disallowed to facilitate single pass assembly.

Example 4.16 Consider the statements

```
XYZ     DB      LENGTH ABC DUP(0)
- -
ABC     DD      ?
```

Here the forward reference to ABC makes it impossible to assemble the DB statement in a single pass.

Segment registers

An ASSUME statement indicates that a segment register contains the base address of a segment. The assembler represents this information by a pair of the form (*segment register, segment name*). This information can be stored in a *segment registers table* (SRTAB). SRTAB is updated on processing an ASSUME statement. For processing the reference to a symbol *symb* in an assembly statement, the assembler accesses the symbol table entry of *symb* and finds (seg_{symb}, $offset_{symb}$) where seg_{symb} is the name of the symbol containing the definition of *symb*. It uses the information in SRTAB to find the register which contains seg_{symb}. Let it be register *r*. It now synthesizes the pair (r, $offset_{symb}$). This pair is put in the address field of the target instruction.

However, this strategy would not work while assembling forward references. Consider statements 6 and 13 in Fig. 4.25 which make forward references to COUNT.

When the definition of COUNT is encountered in statement 20, information concerning these forward references can be found in the table of incomplete instructions (TII). What segment register should be used to assemble these references? The first reference was made in statement 6 when DS was the segment register containing the segment base of DATA. However, SRTAB presently contains the pair (ES, DATA) as a result of statement 8, viz. ASSUME ES:DATA . . . A similar problem may arise while assembling forward references contained in branch instructions. The following provisions are made to handle this problem:

1. A new SRTAB is created while processing an ASSUME statement. This SRTAB differs from the old SRTAB only in the entries for the segment registers named in the ASSUME statement. Since many SRTAB's exist at any time, an array named SRTAB_ARRAY is used to store the SRTAB's. This array is indexed using a counter *srtab_no*.
2. Instead of TII, a *forward reference table* (FRT) is used. Each entry of FRT contains the following entries:
   (a) Address of the instruction whose operand field contains the forward reference
   (b) Symbol to which forward reference is made
   (c) Kind of reference (e.g. T : analytic operator TYPE, D : data address, S : self relative address, L : length, F : offset, etc.)
   (d) Number of the SRTAB to be used for assembling the reference.

Example 4.17 illustrates how these provisions are adequate to handle the problem concerning forward references mentioned earlier.

Example 4.17 Two SRTAB's would be built for the program of Fig. 4.25. SRTAB#1 contains the pairs (CS, CODE) and (DS, DATA) while SRTAB#2 contains the pairs (CS, CODE) and (ES, DATA). While processing statement 6, SRTAB#1 is the current SRTAB. Hence the FRT entry for this statement is (008, COUNT, D, SRTAB#1). Similarly the FRT entry for statement 13 is (024, COUNT, D, SRTAB#2). These entries are processed on encountering the definition of COUNT, giving the address pairs (DS, 001) and (ES, 001). (Note that FRT entries would also exist for statements 5, 7 and 12. However, none of them require the use of a base register.)

4.5.5 Design of the Assembler

Algorithm for the Intel 8088 assembler, Algorithm 4.3, is given at the end of this section. LC processing in this algorithm differs from LC processing in the first pass of a two pass assembler (see Algorithm 4.1) in one significant respect. In Intel 8088, the unit for memory allocation is a byte, however certain entities require their starting byte to be aligned on specific boundaries in the address space. For example, a word requires alignment on an even boundary, i.e. it must have an even start address. Such alignment requirements may force some bytes to be left unused during memory allocation. Hence while processing DB statements and imperatives, assembler first aligns

LC on the requisite boundary. We call this *LC alignment*. Allocation of memory for a statement and entering its label in the symbol table is performed after LC alignment.

The data structures of the assembler are illustrated in Fig. 4.26, where numbers in parentheses indicate the number of bytes required for a field. The mnemonics table (MOT) is hash organized and contains the following fields: *mnemonic opcode, machine opcode, alignment/format info* and *routine id*. The *routine id* field of an entry specifies the routine which processes that opcode. *Alignment/format info* is specific to a given routine. For example, the code of '00H' for routine R2 implies that only one instruction format, that with self-relative displacement, is supported. 'FFH' for the same routine implies that all formats are supported, hence the routine must decide which machine opcode to use. The symbol table (SYMTAB) is also hash-organized and contains all relevant information about symbols defined and used in the source program. The contents of some important fields are as follows: The *owner segment* field indicates id of the segment in which a symbol is defined. It contains the SYMTAB entry # of the segment name. For a non-EQU symbol the *type* field indicates the alignment information. For an EQU symbol, *type* field indicates whether the symbol is to be given a numeric value or a textual value. The owner segment and offset fields are used to accommodate the value.

An SRTAB can contain upto four entries, one for each register. The current SRTAB exists in the last entry of SRTAB_ARRAY. SRTAB's are accessed by their entry numbers in SRTAB_ARRAY.

Forward references

Information concerning forward references to a symbol is organized in the form of a linked list. Thus, the *forward reference table* (FRT) contains a set of linked lists. The *FRT pointer* field of a SYMTAB entry points to the head of this list. Since ordering of FRT entries is not important, for efficiency reasons new entries are added at the beginning of the list. Each FRT entry contains SRTAB # to be used to assemble the forward reference. It also contains the instruction address and a *usage code* indicating where and how the reference is to be assembled. When the definition of a symbol is encountered, its forward references (if any) are processed, and the forward references list is discarded. This minimizes the size of FRT at any time.

Cross references

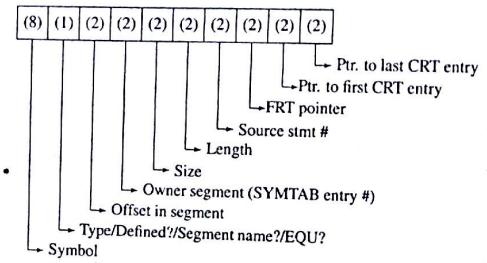
A cross reference directory is a report produced by the assembler which lists all references to a symbol sorted in the ascending order of statement numbers. The assembler uses the *cross reference table* (CRT) to collect the information concerning references to all symbols in the program. Each SYMTAB entry points to the head and tail of a linked list in the CRT. New entries are added at the end of the list.

Being linked lists, FRT and CRT can be organized in a single memory area. The tables grow from the high end of storage to its low end. The freed entries of FRT are reused by maintaining a free list. The target code generated by the assembler grows

(a) Mnemonics table (MOT)

Mnemonic opcode (6)	*Machine opcode* (2)	*Alignment/ format info* (1)	*Routine id* (4)
JNE	75H	OOH	R2

(b) Symbol table (Symtab)



(c) Segment Register Table Array (SRTAB_ARRAY)

Segment Register (1)	*Segment name* (2)	
00(ES)	23	SRTAB #1
:		SRTAB #2

(d)Forward Reference table (FRT)

Pointer (2)	*SRTAB #* (1)	*Instruction address* (2)	*Usage code* (1)	*Source stmt #* (2)

(e) Cross Reference table (CRT)

Pointer (2)	*Source Stmt #* (2)

Fig. 4.26 Data structures of the assembler

from the low end to the high end of storage. As a result, no size restrictions need to be placed on individual tables. The assembler fails to handle a source program only if its target code overlaps with its tables.

Example 4.18 Figure 4.27 illustrates contents of important data structures after processing Statement 19 of the source program of Fig. 4.25. The symbol table contains entries for symbols COMP and NEXT whose definitions have already been processed. The *defined* flag of these entries is = 'Yes' and the address and type fields contain appropriate values. NEXT was forward referenced in Statement 12 of the program. An FRT entry was created for this reference with the *usage code* ='S' to indicate that a self-relative displacement is desired. When the definition of NEXT was processed (Statement 14), the validity of the forward reference in terms of this requirement was checked and the corresponding instruction was completed. The FRT entry was then discarded.

FRT entries currently exist for symbols COUNT and STRNG. Both references to COUNT in the source program are forward references (Statements 6 and 13). Hence, two entries exist for COUNT in FRT and CRT. The first FRT entry has #1 in the SRTAB field, while the second entry has #2 in it. Similarly two FRT and CRT entries exist for STRNG. The *usage code* fields of the FRT entries indicate what information is required in the referencing instruction, e.g. data address ('D'), self relative address ('S'), length ('L'), offset ('F'), etc. Note that some CRT entries are not shown in Fig. 4.27.

Subsequent processing of the program is as follows: At the end of processing statement 20 (but before incrementing LC), its label, viz. COUNT, will be looked up in SYMTAB. An entry exists for it with *defined* = '*no*'. This implies that COUNT has been forward referenced. Its *segment* and *offset* fields are now set. The forward reference chain is then traversed and each forward reference is processed to perform error detection, and to complete the machine instruction containing the forward reference. The second forward reference to COUNT passes the error detection step and leads to completion of the machine instruction with offset 0024. The first forward reference, however, expects a word alignment for COUNT (since immediate data is 2 bytes in length) which is not the case. An error is indicated at this point. The *FRT pointer* field of COUNT's SYMTAB entry is now reset and the FRT entries for COUNT are destroyed.

Listing and error indication

The program listing and error reporting function faces the problems discussed in Section 4.4.7. The program listing can contain the statement in source form, its serial number in the program and the memory address assigned to it. The target code can only be printed at the end of assembly. Error reporting also suffers due to the single pass assembly scheme. An error cannot be reported against the statement containing a forward reference since the statement would have been already listed out. If the error is simply reported at the end of the source program, it is rather cumbersome for the programmer to identify the erroneous statement.

The following strategy is used to overcome this problem (see Ex. 4.19):

1. The serial number of the source statement containing a forward reference is stored in the FRT entry along with other relevant information (see Fig. 4.27).

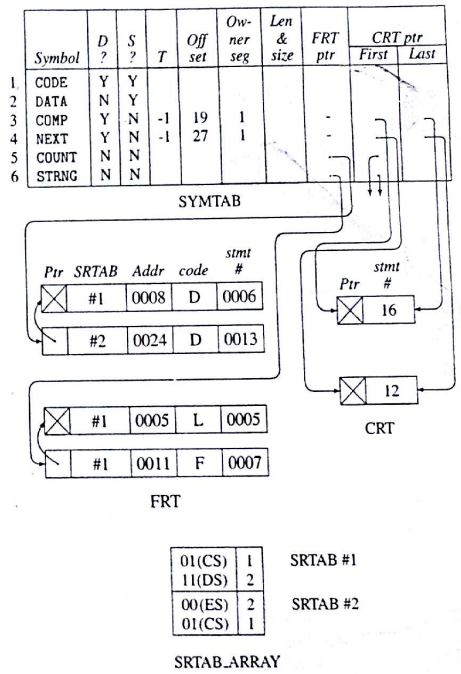


Fig. 4.27 Data structures after processing Statement 19

2. Whenever a symbol's definition is encountered, all forward references to the symbol are processed and errors, if any, are reported against this statement. Though this is not as effective as reporting the error against the erroneous statement, it is the next best thing.

Algorithm 4.3 (Single pass assembler of 8088)

1. *code_area_address* := address of *code_area*;
   srtab_no := 1;
   LC := 0;
   stmt_no := 1;
   SYMTAB_segment_entry := 0;
   Clear ERRTAB, SRTAB_ARRAY.
2. While next statement is not an END statement
   (a) Clear *machine_code_buffer*.
   (b) If label is present then
   this_label := symbol in label field;
   (c) If an EQU statement
   (i) *this_address* := value of operand expression;
   (ii) Make an entry for *this_label* in SYMTAB with
   offset := *this_addr*;
   Defined := '*yes*';
   owner_segment := *owner_segment* of operand symbol;
   source_stmt_# := *stmt_no*;
   (iii) Enter *stmt_no* in the CRT list of the label in the operand field.
   (iv) Process forward references to *this_label*;
   (v) *size* := 0;
   (d) If an ASSUME statement
   (i) Copy the SRTAB in SRTAB_ARRAY [*srtab_no*] into SRTAB-_ARRAY [*srtab_no*+1];
   (ii) *srtab_no* := *srtab_no*+1;
   (iii) *this_register* := register mentioned in the statement.
   (iv) *this_segment* := entry number of SYMTAB entry of the segment appearing in operand field.
   (v) Make the entry (*this_register, this_segment*) in SRTAB_ARRAY [*srtab_no*]. (This overwrites an existing entry for *this_register*.)
   (vi) *size* := 0;
   (e) If a SEGMENT statement

   (i) Make an entry for *this_label* in SYMTAB.
   (ii) Set *segment name ?* := *true*;
   (iii) *SYMTAB_segment_entry* := entry no. in SYMTAB;
   (iv) LC := 0;
   (v) *size* := 0;
   (f) If an ENDS statement then
   SYMTAB_segment_entry := 0;
   (g) If a declaration statement
   (i) Align LC according to the specification in the operand field.
   (ii) Assemble the constant(s), if any, in the *machine_code_buffer*.
   (iii) *size* := size of memory area required;
   (h) If an imperative statement
   (i) If operand is a symbol *symb* then
   enter *stmt_no* in CRT list of *symb*.
   (ii) If operand symbol is already defined then
   Check its alignment and addressibility.
   Generate the address specification (segment register, *offset*) for the symbol using its SYMTAB entry and SRTAB_ARRAY [*srtab_no*].
   else
   Make an entry for *symbol* in SYMTAB.
   Defined := '*no*';
   Enter (*srtab_no*, LC, *usage code, stmt_no*) in FRT.
   (iii) Assemble instruction in *machine_code_buffer*.
   (iv) *size* := size of the instruction;
   (i) If *size* ≠ 0 then
   (i) If label is present then
   Make an entry for *this_label* in SYMTAB.
   owner_segment := *SYMTAB_segment_entry*;
   Defined := '*yes*';
   offset := LC;
   source_stmt_# := *stmt_no*;
   (ii) Move contents of *machine_code_buffer* to the address *code_area-_address*;
   (iii) *code_area_address* := *code_area_address* + *size*;
   (iv) Process forward references to the symbol. Check for alignment and addressability errors. Enter errors in ERRTAB.
   (v) List the statement with errors contained in ERRTAB.
   (vi) Clear ERRTAB.

3. (Processing of END statement)
   (a) Report undefined symbols from SYMTAB.
   (b) Produce cross reference listing.
   (c) Write *code_area* into output file.

Example 4.19 Error in the forward reference to COUNT in Statement 6 of Fig. 4.25 would be reported as:

Stmt no.	*Source statement*			*Offset*	*Instrn*
006		MOV	COUNT, 0000	0005	...
...		...		...	...
020	COUNT	DB	?	0001	...

** error ** Illegal forward reference (alignment) from Stmt 6.

BIBLIOGRAPHY

Most books on computer architecture discuss assembly languages of particular computer systems, e.g. Stone and Siewiorek (1975), Gear (1980), and Macewen (1980). Leeds and Weinberg (1966) is one of the first books devoted to assembly language programming. Flores (1971) covers assembly language of IBM/360. Books by Leventhal deal with assembly language programming for microcomputers.

(a) Assembly language programming

1. Leeds, H.D. and M.W. Weinberg (1966): *Computer Programming Fundamentals*, McGraw-Hill, New York.
2. Leventhal, L.A., A. Osborne, and C. Collins (1980): *Z8000 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley.
3. Rudd, W.G. (1976): *Assembly Language Programming and the IBM 360/370 Computers*, Prentice-Hall, Englewood Cliffs.
4. Stone, H.S. and D.P. Siewiorek (1975): *Introduction to Computer Organization and Data Structures*, McGraw-Hill, New York.
5. Weller, W.J. (1975): *Assembly Level Programming for Small Computers*, Heath & Co., Lexington.
6. Yarmish, R. and J. Yarmish (1979): *Assembly Language Fundamentals 360/370, OS/VS, DOS/VS*, Addison-Wesley, Reading.

(b) Assemblers

1. Barron, D.W. (1969): *Assemblers and Loaders*, Macdonald Elsevier, London.
2. Calingaert, P. (1979): *Assemblers, Compilers and Program Translation*, Computer Science Press, Maryland.
3. Donovan, J.J. (1972): *Systems Programming*, McGraw-Hill Kogakusha, Tokyo.
4. Flores, I. (1971): *Assemblers and BAL*, Prentice-Hall, Englewood Cliffs.

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.