

## Chapter 3

# Assembler Design

(code of  
Assembly lang  
→  
Assembler  
→  
machine code)

Assembler is the tool to convert an assembly language program into the machine language one, understandable by the processor that executes it. The complexity of the process depends upon various factors, including the size of the instruction set, different addressing modes supported, length of program being translated, and so on. For the simplest case, the assembly may be done manually, aptly named as hand assembly. However, the job becomes difficult and error-prone for complex processors, like the Intel's 32-bit architectures discussed in Chapter 2. For this type of machines, the automation of the assembly process is a must.

Ideally, an assembler can be viewed as a tool that looks at each statement of the assembly language program and generates the corresponding machine language counterpart. This has been depicted in Fig. 3.1. However, apart from producing code in the machine language, some more information is needed to facilitate loading of program at arbitrary start address in the memory. It needs corrections to the address sensitive values, like operand addresses, jump targets, etc. The assemblers normally generate code starting at offset zero. For multi-section programs (that is, programs with multiple code and/or data sections), each section is assembled starting at offset zero, so that maximum flexibility exists regarding their loading into the memory for execution. Thus, information about all the sections and their attributes is also needed to produce the final executable version of the program.

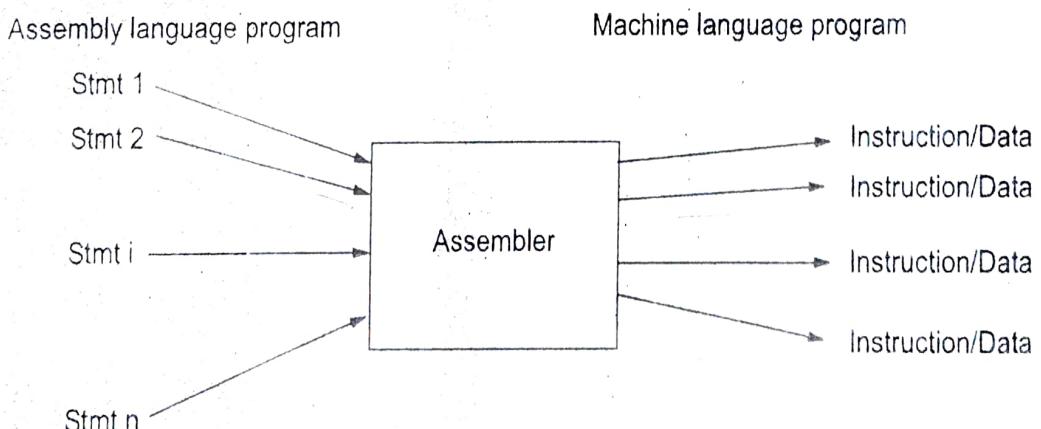


FIGURE 3.1 The role of the assembler.

In this chapter, we will first look into the manual assembly process, taking into

consideration a very simple assembly language. Then, we will look into the important data structures needed in the assembly process. This will be followed by a detailed discussion on the design of assemblers.

### 3.1 A SIMPLE MANUAL ASSEMBLER

Consider a simple hypothetical accumulator based processor. All memory load and store operations are through the accumulator register (call it  $A$ ). Arithmetic and logic operations mostly use  $A$  as a source register and also the destination register. Assume that there are two more 32-bit general purpose registers  $B$ ,  $C$ , and an index register  $I$ . Arithmetic is permitted on  $I$  also. The addressing modes supported are the immediate and indirect through the index register. All memory addresses are 32-bit wide. Register-to-register data movement is also supported. The instructions and the associated codes are as shown in Table 3.1. This table depicting the machine instructions, is called the Machine Opcode Table (MOT).

Table 3.1: Machine opcode table for the simple processor.

Instruction	Size (in bytes)	Format	Meaning
MVI $A, <constant>$	5	<0, 4 byte constant>	$A \leftarrow constant$
MVI $B, <constant>$	5	<1, 4 byte constant>	$B \leftarrow constant$
MVI $C, <constant>$	5	<2, 4 byte constant>	$C \leftarrow constant$
MVI $I, <constant>$	5	<3, 4 byte constant>	$I \leftarrow constant$
LOAD <constant>	5	<4, 4 byte constant>	$A \leftarrow memory[<constant>]$
STORE <constant>	5	<5, 4 byte constant>	$memory[<constant>] \leftarrow A$
LOADI	1	<6>	$A \leftarrow memory[I]$
STORI	1	<7>	$memory[I] \leftarrow A$
ADD $B$	1	<8>	$A \leftarrow A + B$
ADD $C$	1	<9>	$A \leftarrow A + C$
MOV $A, B$	1	<10>	$A \leftarrow B$
MOV $A, C$	1	<11>	$A \leftarrow C$
MOV $B, C$	1	<12>	$B \leftarrow C$
MOV $B, A$	1	<13>	$B \leftarrow A$
MOV $C, A$	1	<14>	$C \leftarrow A$
MOV $C, B$	1	<15>	$C \leftarrow B$
INC $A$	1	<16>	$A \leftarrow A + 1$
INC $B$	1	<17>	$B \leftarrow B + 1$
INC $C$	1	<18>	$C \leftarrow C + 1$
CMP $A, <constant>$	5	<19, 4 byte constant>	compare $A$ to $constant$
CMP $B, <constant>$	5	<20, 4 byte constant>	compare $B$ to $constant$
CMP $C, <constant>$	5	<21, 4 byte constant>	compare $C$ to $constant$
ADDI <constant>	5	<22, 4 byte constant>	$I \leftarrow I + constant$
JE <label>	5	<23, 4 byte address>	jump on equal to <label>
JMP <label>	5	<24, 4 byte address>	jump to <label>
STOP	1	<25>	stop the processor

We assume that our assembler supports only one pseudo-opcode  $[dd]$  to reserve and initialize four byte memory locations. In a general assembler, there may be a number of pseudo-opcodes supported which are stored in another table known as *Pseudo Opcode Table (POT)*.

Next, we consider a simple program (Program 3.1) written in this language to sum ten numbers stored in the memory. The assembly process starts at offset zero. To keep track of the next offset in which the generated code is to be placed, the assembler maintains a variable called *location counter*, which is analogous to the *offset* and is also initialized to zero. Since line 1 contains a declaration of variable *X* having 10, 32-bit numbers, the assembler reserves

**PROGRAM 3.1:** A program to add 10 numbers.

```

1   . X dd 10, 20, 40, 5, 7, 9, 53, 8, 11, 13 ] 40  0-39
2   . sum dd 0 40-58
3   . MVI I, X 5
4   . MVI B, 0 5
5   . MVI C, 0 5
6   . L1: LOADI 1 7
7   . ADD C 1 1
8   . MOV C, A 1 1
9   . INC B 1 1
10  . CMP B, 10 5 5
11  . JE L2 5
12  . ADDI 4 5
13  . JMP L1 5
14  . L2: STORE sum 5 ] 83-88
15  . STOP 1

```

space for the same and the locations are initialized accordingly. Next, *location counter* is incremented to 40, as the 40 locations are occupied by *X*. The variable *X* along with its attributes is entered into the *symbol table* as shown in Table 3.2. It may be noted that *symbol table* contains all variables and labels defined in the program. The table helps in the assembly process and is also needed by later tools like *linker* and *loader*. Next four locations are used to store *sum* as specified in the program. Thus, the assembler reserves the space and initializes it to zero. The variable *sum* is also entered into the *symbol table*. Current *location counter* is 44. Next, the instruction

MVI I, X

Table 3.2: The symbol table of the program.

SYMTAB.

Name	Type	Offset
X	variable	0
sum	variable	40
L1	label	59
L2	label	83

### 3.1. A SIMPLE MANUAL ASSEMBLER

39

is assembled into five bytes as depicted by the MOT. The location 44 contains the opcode 3, while 45 through 48 hold the offset of  $X$ , which is zero in this case. *Location counter* is updated accordingly. When the assembler reaches the instruction,

L1: LOADI

the label L1 is entered into the symbol table along with its offset from the beginning of the program. So that, when we assemble the instruction,

JMP L1

we already know the offset of L1 and the code can be generated accordingly. However, when we try to assemble the instruction,

JE L2

we have not yet seen the label L2. Thus, the offset of L2 is not known. There are several ways of handling this type of situation, that we will discuss later. For the time being, we assume that we know the offset of L2 to be 83 (53 hexadecimal), and accordingly generate the machine code. The code generated corresponding to each line of the program has been shown in the form of a list file in Program 3.2. The first column of the table notes the source line number, and the second column gives the *location counter* value in hexadecimal. The next column contains the code generated. The source line is noted in the last column. There are several address sensitive points in the code that need to be corrected if the program is to be loaded starting from a memory address other than zero. These have been listed in Table 3.3. If the program is loaded from location  $l$  (for example), the value  $l$  should be added to all these locations to ensure correct execution of the program. This process is known as program relocation, and will be discussed in detail while dealing with linking and loading techniques.

**PROGRAM 3.2:** The list file of the program to add to numbers.

1	00000000 0A00000014000000	X dd 10, 20, 40, 5, 7, 9, 53, 8, 11, 13
	2B00000005000000	
	0700000009000000	
	3500000008000000	
	0B0000000D000000	
2	00000028 00000000	sum dd 0
3	0000002C 03000000	MVI I, X
4	00000031 01000000	MVI B, 0
5	00000036 02000000	MVI C, 0
6	0000003B 06	L1: LOADI
7	0000003C 09	ADD C
8	0000003D 0E	MOV C, A
9	0000003E 11	INC B
10	0000003F 140A000000	CMP B, 10
11	00000044 1753000000	JE L2
12	00000049 1604000000	ADDI 4
13	0000004E 183B000000	JMP L1

```

14 00000053 0528000000
15 00000058 1900000000

```

L2: STORE sum  
STOP

Table 3.3: Address sensitive places for the code.

Address	Reason
0000002D	X needs correction
00000044	jump target L2 modified
0000004E	jump target L1 modified
00000054	sum needs correction

## 3.2 ASSEMBLER DESIGN PROCESS

In this section, we will discuss in detail the various issues and techniques involved in the design of the assembler. To understand the process better, we can think of a program written in the assembly language to be consisting of a few components as follows:

**Machine instructions.** Machine instructions are the heart of the program that tells the processor the exact operations to be performed as desired by the programmer. These include all the machine opcodes like MOV, ADD, SUB, JMP, and so on.

**Variable declarations.** Variable declarations are the declaration of the storage space (both initialized and non-initialized). The machine instructions specified in the program will modify these data to achieve the task of the program. The common examples of this category are db, dw, dd, resb, etc.

**Assembler directives.** Assembler directives direct the assembler to produce code in a structured manner so that different sections of the program can be handled efficiently and perhaps independently. For example, the section directive in NASM allows the programmer to demarcate different code and data segments of the program. Another very useful directive is END that tells the assembler to terminate the assembly process. It is useful because there may be some portion in the file in which the user is not interested for the time being. For example, some error-handling routines may not be included in the initial versions of the program during the developmental stage. These may be excluded from the assembly process simply by terminating the assembly before those. There can be many other such directives depending upon the facilities provided to the programmer by the assembler designer.

**Comments.** These are used by the programmer for documentation purposes. Most assemblers allow the user to put a comment by prefixing it with a ';' symbol. For example,

MOV AX, 39 ;move constant 39 to AX

Since the comments are not useful for program execution, the assembler simply discards all comments while doing the translation.

### 3.2.1 Major Data Structures Used

The major tables involved in the assembly process are:

- Machine Opcode Table (MOT)
- Pseudo Opcode Table (POT)
- Symbol Table (SYMTAB).

*static*

*dynamic*

Out of these, the *Machine Opcode Table (MOT)* and *Pseudo Opcode Table (POT)* are static in nature, in the sense that their content does not change during the lifetime of the assembler, until and unless a new processor with different opcodes is targeted, or new pseudo-opcodes are added to facilitate storage allocation. On the other hand, content of the *Symbol Table (SYMTAB)* depends upon the program being assembled—the variables and labels declared within it. Thus, *Symbol Table* is dynamic in nature.

**Machine Opcode Table (MOT):** As noted earlier, the machine opcode table holds the opcodes used by the processor for different instruction mnemonics. Typical entries of this table have been shown in Fig. 3.2. The *Mnemonic* field holds various instruction mnemonics. The *Size* field contains the size of the instructions (typically in bytes). The *Opcode* field contains the machine code corresponding to the mnemonic.

Mnemonic	5	MOV A,B
Mnemonic	Size	Opcode
MOV		

FIGURE 3.2 Machine opcode table structure.

The basic structure of the MOT may be varied significantly based upon the instruction set and also on the designer of the assembler. For simple machines, (like the one in Section 3.1), the *Opcode* field is quite simple. However, as noted in Chapter 2, Section 2.1.5, the *Opcode* field may also depend upon the operand type, direction of data movement etc. This makes the design of *MOT* complex. A very simplistic approach may be to just include the portion of the opcode that is independent of these sorts of variations. On the other hand, one can put all the alternative entries into the MOT and the assembler will just search out the correct combination.

**Organization of MOT:** Organization of any table is determined by the set of operations, namely, *insertion*, *deletion*, *search*, *update*, etc. to be performed on the table entries. The frequency of these operations often dictates the organization to be used. For MOT, *insertions* are not there, since the table is static in nature. Similarly, *deletion* is also absent. The only operation that is done is *searching*. In fact, the assembler needs to refer to this table to translate each line of the source file. Thus, the structure of MOT be such that the *search* operation can be carried out very fast. A very good data structure providing constant time  $O(1)$  searching is the *hash table*. A suitable choice of hash function has to be made to convert a mnemonic into an integer value to be used as the index to the table. A detailed discussion on such hashing functions and the general hashing scheme is beyond the scope of this book. Readers may consult some book on *Data Structures* for the same. A basic problem with hash table is the collision of data items, that is, two or more mnemonics mapping to the same location in

the MOT. Good *collision resolution strategies* are needed to take care of this situation. Table 3.4 shows the calculation of hash indexes for a set of mnemonics using the hash function  $h(s)$  (where  $s$  is the mnemonic) given by,

$$h(s) = (\text{Sum of ASCII values of the characters of } s) \bmod 23$$

It may be noted that there is a collision in the indexes computed for the mnemonics ADD and CMP, for the mnemonics LOAD and MOV, and for the mnemonics MVI and STORE.

Table 3.4: Hash table index calculation.

Mnemonics	Sum of ASCII values	hash index ( $h(s)$ )
ADD	201	17
ADDI	274	21
CMP	224	17
INC	218	11
JE	143	5
JMP	231	1
LOAD	288	12
LOADI	361	16
MVI	236	6
MOV	242	12
STOP	326	4
STORE	397	6
STORI	401	10

Some other alternative data structures can be,

- Binary Search Tree

- Link Lists indexed by the first alphabet of the mnemonic.

For the *binary search tree* organization, the mnemonics are sorted in the alphabetical order and then inserted into a binary search tree to ensure the worst case access time to be equal to the depth of the tree. With proper *height balancing*, a tree with  $n$  nodes provides the worst case access time of  $O(\log n)$ . For example, for a processor with mnemonics MVI, LOAD, STORE, LOADI, STORI, ADD, MOV, INC, CMP, JE, JMP, ADDI, STOP, a binary search tree may be as shown in Fig. 3.3. Readers may refer to some book on *Data Structures* for further details.

Alternatively, an array of link lists structure may be used. The array is indexed by the first letter of the mnemonic. For example, corresponding to the set of mnemonics used in Fig. 3.3, an array of link lists structure has been shown in Fig. 3.4. Since the number of mnemonics starting with an alphabet is expected to be small, it may provide a reasonably good access mechanism.

**Pseudo Opcode Table (POT):** Pseudo opcode table contains the pseudo opcodes supported by the assembler. These are used to reserve memory space and possibly initialize it.

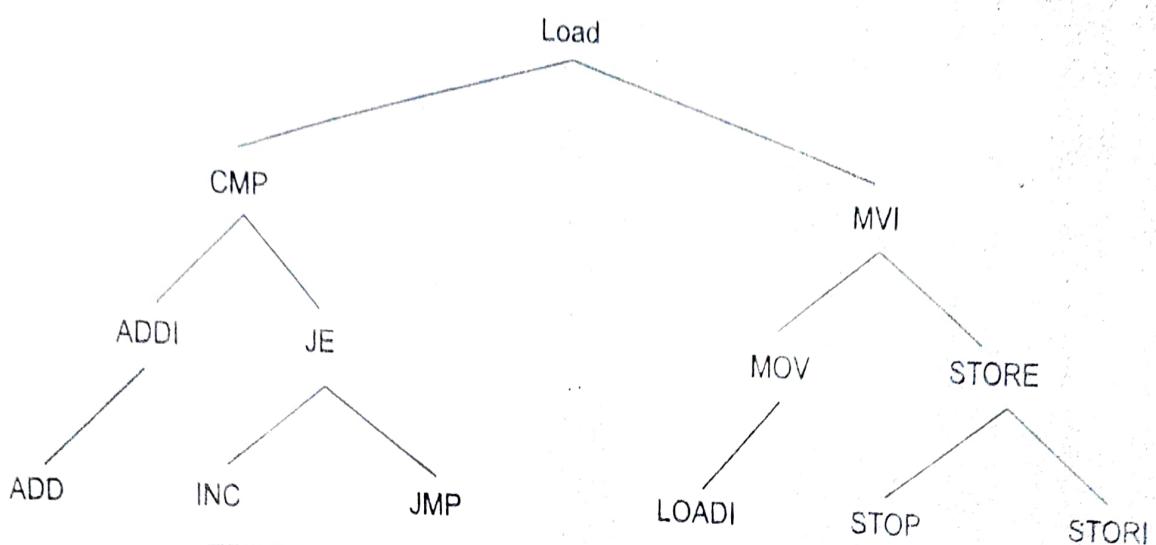


FIGURE 3.3 Binary search tree organization of MOT.

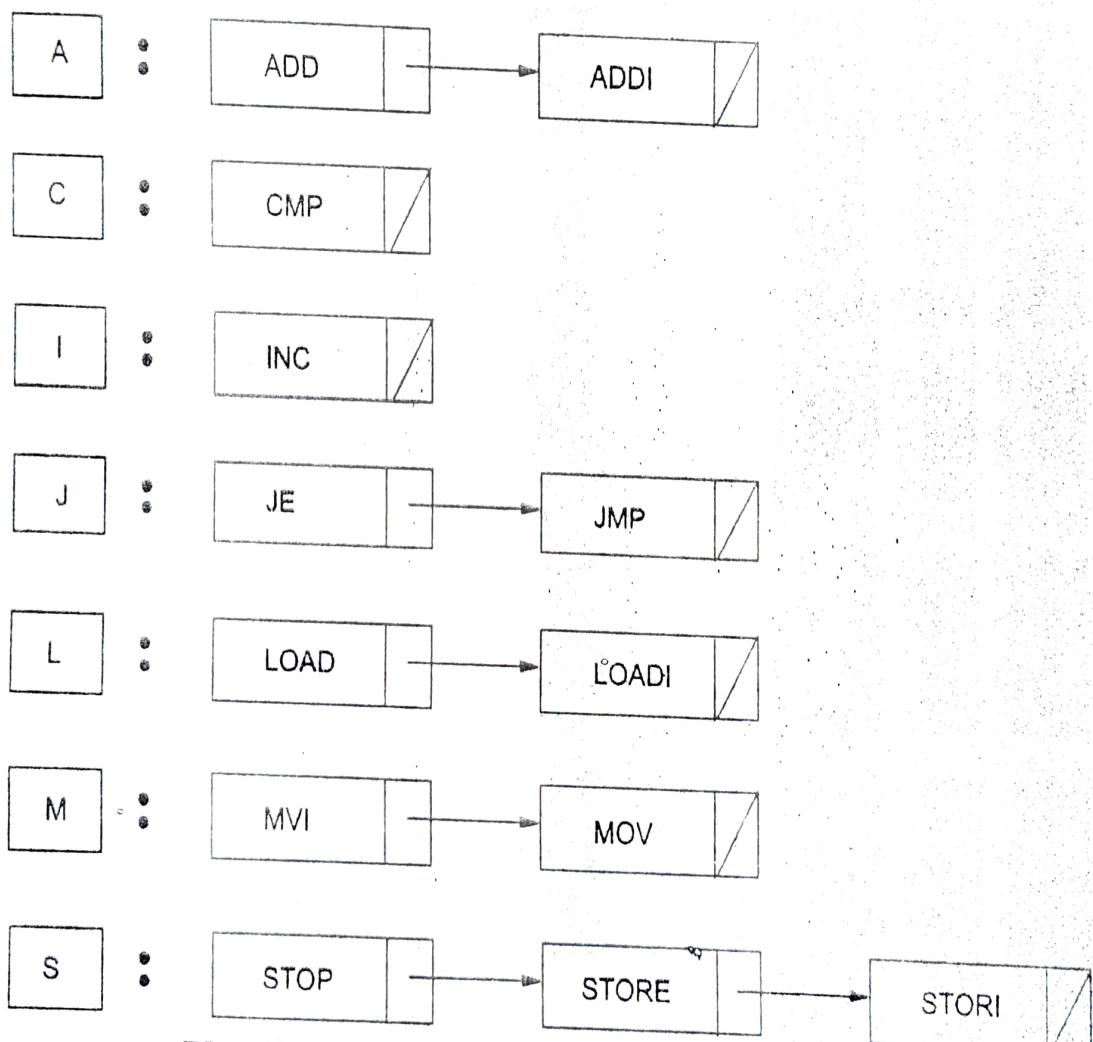


FIGURE 3.4 Array of link lists organization of MOT.

There can be several pseudo opcodes as envisaged by the assembler designer. The following is a list of such items supported in *NASM*:

for initialized val	DB	define byte
	DW	define word
	DD	define double-word/float
	DQ	define double-precision float
	DT	define extended-precision float
for uninitialized variables	RESB	reserve byte
	RESW	reserve word
	RESD	reserve double word/float
	RESQ	reserve double-precision float
	REST	reserve extended-precision float

Thus, a typical POT may have the structure as shown in Fig. 3.5. The field pseudo-opcode holds the name of the pseudo opcode. The type field identifies type of data it can contain. Size field holds the size of the field (typically in bytes). The field Initializable is boolean specifying whether the memory locations corresponding to the pseudo opcode can have initial values. For the pseudo opcodes like DB, DW, DD, DQ, and DT, the corresponding memory locations contain some initial values, whereas for RESB, RESW, RESD, RESQ, and REST, the locations cannot be initialized. Organization of the Pseudo-opcode table does not matter much in the performance of the assembler since the number of entries is much smaller as compared to MOT.

Pseudo-opcode	Type	Size	Initializable
			Boolean 0 or 1

FIGURE 3.5 Pseudo-opcode table structure.

**Symbol Table (SYMTAB):** Symbol table is an essential data structure used by the assembler to remember information about identifiers appearing in the program to be assembled. The types of symbols that are stored in the symbol table include variables, procedures, functions, defined constants, labels, etc. However, it should be kept in mind that the symbol table is designed by the assembler writer to facilitate the assembly process. Thus, the identifiers stored in the symbol table may vary widely from one assembler to another, even for the same assembly language.

Name      Type      offset

**Information in Symbol Table:** For an identifier stored in the symbol table, the following are the associated information stored.

- **Name:** The name of the identifier. The name may be stored directly in the table, or the table entry may point to another character string (possibly stored in an associated string table). The second indirect approach is particularly suitable if the names can be arbitrarily long and widely varying in length.
- **Type:** The type of the identifier. It defines, for example, whether the identifier is a variable, a label, a procedure name, and so on. For variables, it will further identify the type—the basic types like byte, word, double word, etc.
- **Location:** This is an offset within the program where the identifier is defined.

The fundamental operations on such a table are:

- Enter a new symbol in the table.
- Lookup for a symbol
- Modify information about a symbol stored earlier in the table.

There exist several alternative data structures to choose from to create such a table. The commonly used techniques are:

- Linear table ✓
- Ordered list ✓
- Tree ✓
- Hash table ✓

*Linear Table:* This is a simple array of records with each record corresponding to an identifier of the program. The entries are made in the same order in which they appear in the program.

**EXAMPLE 3.1** Consider the following definitions.

```

x : RESB
y : RESW
z : RESD
...
procedure abc
...
L1 : ...
    
```

The linear table will store the information as shown in Table 3.5.

**Table 3.5:** Symbol table.

Name	Type	Location
x	byte	offset of x
y	word	offset of y
z	double word	offset of z
abc	procedure	offset of abc
L1	label	offset of L1

If the language puts no restriction on the length of the string representing the name of an identifier, it may be convenient to store the names in a string table with the name field holding pointers to it. With this type of organization, the *lookup*, *insert* and *modify* operations are going to take  $O(n)$  time,  $n$  being the number of identifiers stored. Of course, *insertion* can be made  $O(1)$  by remembering the pointer to the next free index of the table where an entry can be inserted.

go. Also, if we can scan most recent entries first, it can probably speed up the access. This is because of program locality—a variable defined just inside a block is expected to be referred to more often than some earlier variable.

Ordered List: This is a variation of linear tables in which a list organization is used. The list may be sorted in some fashion, and then binary search can be used to access the table in  $O(\log n)$  time. However, an insertion needs to be done at an appropriate place to preserve the sorted form. Thus, insertion becomes costly.

Tree: Tree organization of the symbol table may be used to speed up the access. Each entry is represented by a node of the tree. Based on string comparison of names, the entries lesser than a reference node are kept in the left-subtree of it, whereas the entries greater than the reference node are put into the right subtree. The individual nodes, apart from having the regular information related to the symbol, also hold two more pointer fields to the left and right subtrees. Moreover, each lookup operation, apart from the equality check, also needs to be followed by a less-than/greater-than check for each node of the tree where a match is not found, and the search must proceed further. A possible tree representing the entries of Example 3.1 has been shown in Fig. 3.6.

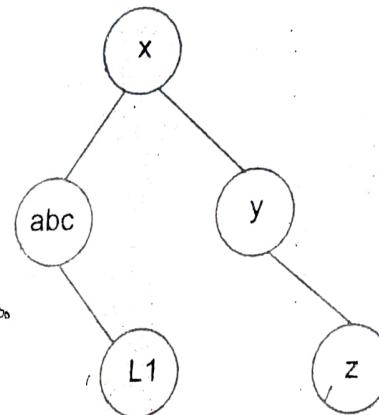


FIGURE 3.6 Tree symbol table.

The average lookup time for an entry in the tree is  $O(\log n)$ ; however, it could degrade to a single long branching, requiring  $O(n)$  search time. Proper height balancing strategies may be used (for example; AVL trees) to keep the tree balanced.

Hash Table: As discussed earlier, hash table provides the fastest mechanism to access the symbols. The techniques used can be similar to those for MOT. However, since the set of symbols appearing in a program are not fixed (unlike the MOT content which is fixed), designing a good hash function with lesser number of collisions may be difficult. Thus, good collision resolution strategies are needed to handle such situations.



### 3.2.2 Two-Pass Assembler

The two-pass assembly process scans the input assembly language program twice. These scans are normally called *Pass I* and *Pass II*. In *Pass I*, different data structures (tables) are filled up with the information pertaining to the symbols and sections defined in the program, whereas the second pass generates the actual code. Both the passes use a variable, *location counter*

(lc in short) to compute current offset from the beginning of a *section* while traversing the input file.

**Pass I** The main task of this pass is to scan the input file and compute the offsets of all symbols appearing in the program. In the process, it fills up a number of tables as mentioned below:

• **Section table:** It holds detailed information regarding all sections appearing in the input program. The typical entries in this table are *section name*, *size*, *attributes* and a *pointer* to the translated content of the section. The table is as shown in Fig. 3.7(a). *Section name* is the name of the section, *size* is the total size. The *section declaration* may have several attributes associated with them, which we will discuss in Chapter 4, Section 4.1.2. At the end of *Pass II*, *pointer* will point to the content of the section after translation.

• **Symbol table:** As explained earlier, symbol table holds information about the symbols defined in the program. The entries are *name*, *type*, *location*, *size*, and *section-id*. The *name* contains the name of the symbol, *type* stores the type such as *variable*, *label*, etc., *location* is the offset of the symbol from the start of the section containing it, *size* stores the size of the symbol in bytes, *section-id* identifies the section to which the symbol belongs. The field *Is-global* is Boolean, identifying whether the symbol has been declared as global. The symbol table structure is shown in Fig. 3.7(b).

• **Common table:** It holds information about the variables declared COMMON. The structure has been shown in Fig. 3.7(c).

COMMON var = 59

Name	Size	Attributes	Pointer to content

(a)

Name	Type	Location	Size	Section-id	Is-global

(b)

Name	Size

(c)

FIGURE 3.7 Structure of the (a) section table, (b) symbol table, and (c) common table.

The flowchart corresponding to the first pass of a two-pass assembler has been shown in Fig. 3.8. In the beginning, it initializes the *Global list* into which all *global declarations* will be stored. The location counter *lc* is also initialized to zero. The process then continues by reading the next line and updating one or more tables as needed, till the *end-of-file marker* is reached, which may be the physical end of the file or some special symbol marking the end of

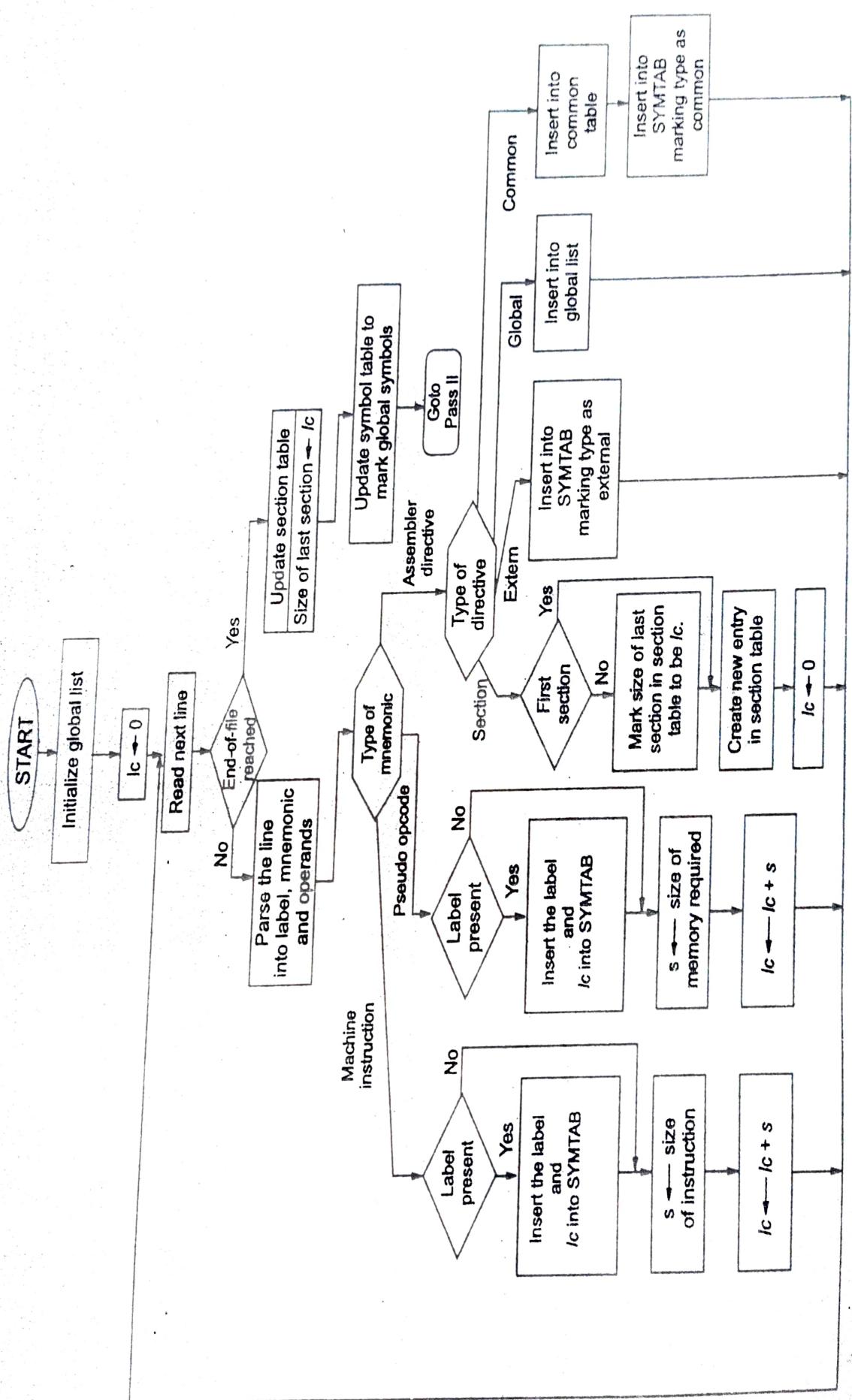


FIGURE 3.8 Pass I of the two-pass assembler.

the portion of the file to be assembled. The process *Read next line* scans the input file and returns the line to be assembled next skipping over the comments. The line is then parsed into components like a possible *label*, *mnemonic*, and *operands*. The mnemonic can be a machine instruction, a pseudo opcode, or an assembler directive. If a label is present in the line, it is put into the symbol table along with the current *lc* value as the *location*. *Type* field is set as *label* for machine instruction. For pseudo opcode, *type* field is set as *variable*. *Size* is not required for a label, whereas for a variable, it is set to the amount of memory required to store the variable. *Section-id* is set to the index of the current segment in the *Segment* table. The field *Is-global* is set to *false* for the time being. Next, *lc* is updated by the size of the instruction as computed by consulting Section 2.1.5 for IA32 processor, or similar information for other processors. For pseudo opcodes, *lc* is updated by the amount of memory required. For assembler directives, the course of action depends upon the type of the directive. For a *Section* directive indicating the beginning of a new section, the size of the last section in *Section* table is set to the current *lc* value. A new entry is created in the *Section* table and *lc* is reset to zero, as the new offsets will be calculated starting from this segment declaration. For an *external* symbol defined via an *extern* declaration, it is put into the symbol table, marking its type as *external*. For *global* symbols, it is put into the *global list*. Similarly, the common symbols are put into the *common* table and are also inserted into the *symbol* table marking its type as *common*. When the *end-of-file* is reached, *section* table is updated to mark the size of the last section as *lc*. *Symbol* table is updated to set the *Is-global* field to *true* for all entries corresponding to the symbols in the global list. Control then transfers to *Pass II*.

**EXAMPLE 3.2** To understand the assembly process, let us consider the program shown in Program 2.1, reproduced in Program 3.3 for the sake of easy reference. To start with, *Global list* is initialized and *lc* is set to zero. In the first line, the statement `global main` puts the entry *main* into the *Global list*. The next line `extern printf` puts the entry *printf* into symbol table with type set as *external*. Next, the new *section declaration* `section .data` is encountered. An entry is made into the *Section table* and *lc* is reset to zero. In the next line, label *my\_array* is found and entered into the *Symbol table* with *Type* as *variable*, *Size* as 40 bytes (as there are 10 double words, each of 4 bytes), *Location* as zero (current *lc* value), *Section-id* as 1 and *Is-global* set to *false*. *lc* is incremented to 40 (28 hexadecimal). Next, the variable *format* is found and entered into the *Symbol table* with *Type* as *variable*, *Location* 40, *Size* 3 bytes, *Section-id* 1, and *Is-global* *false*. *lc* is incremented to 43. A new section declaration `section .text` is encountered next. Thus, the current *lc* value 43 is entered as *Size* of *Section 1* in the *Section* table. A new entry is created for section *.text* in the *Section* table and *lc* is reset to zero. The next instruction has the label *main* associated with it. Thus, *main* is entered into the *Symbol* table. Size of the instruction is 5 bytes. Thus, *lc* is incremented to five. This way, the *Pass I* continues by computing the length of the instructions and incrementing *lc* accordingly. The labels *L2*, *L1*, and *over* are entered into the *Symbol* table in the process. At the end, after processing the RET instruction, *end-of-file* is reached. The current *lc* value 55 (37 hexadecimal) is entered as *Size* of the section *.text*. *Global list* is consulted to mark the label *main* as *global* in the *Symbol* table. This is done by setting *Is-global* to *true*. The *Section* table and *Symbol* table created through *Pass I* for the program are shown in Fig. 3.9(a) and (b) respectively. The *Common* table is empty in this case, and hence, it is not shown.

**PROGRAM 3.3:** Program to find the maximum in an array.

```

global main extern printf

section .data
my_array:
        dd 10, 20, 30, 100, 200, 56, 45, 67, 89, 77

format:
        db '%d', 10, 0

section .text
main:
        MOV ECX, 0
        MOV EAX, [my_array]

L2:
        INC ECX
        CMP ECX, 10
        JZ over

        CMP EAX, [my_array + ECX*4]
        JGE L1
        MOV EAX, [my_array + ECX*4]

L1:
        JMP L2

over:
        PUSH EAX
        PUSH dword format
        CALL printf
        ADD ESP, 8

RET

```

**Pass II** This phase is responsible for generating the code. It uses the tables created in Pass I and writes the generated code into an object file. A flowchart of the entire Pass II has been shown in Fig. 3.10. To start with, the obj-file-offset and lc are initialized to zero. The source program lines are now read one by one sequentially and the corresponding code is generated. The source line is parsed into its components to separate out the mnemonic and operands. If the mnemonic is a machine instruction, the operand addresses are found via the symbol table. If the operand types noted in the symbol table are acceptable for the instruction, code is generated else an error message is flashed. If an operand's type is *external*, the operand address cannot be determined by the assembler. Thus, it puts the address as zero and makes an entry into the *external reference list* as shown in Fig. 3.11. The actual code generation will definitely depend upon the target processor. For example, for the IA32 processor, the translation process will be as depicted in Section 2.1.5. Generated code is written into the object file which automatically updates the *obj-file-offset*. *lc* value is also updated. It may be noted that the *lc*

### 3.2. ASSEMBLER DESIGN PROCESS

51

Name	Size	Attributes	Pointer to content
.data	43		
.text	55		

(a)

Name	Type	Location	Size	Section-id	Is-global
printf	external				
my-array	variable	0	40	1	false
format	variable	40	3	1	false
main	label	0		2	true
L2	label	10		2	false
L1	label	35		2	false
over	label	37		2	false

(b)

FIGURE 3.9 (a) Section table, and (b) Symbol table for the example.

value will be needed to generate code for *relative JMP/CALL* instructions as explained in the following Example 3.3. For pseudo-opcodes, like *DB*, *DW*, etc., the corresponding initialization values are written into the *object* file, whereas for *RESB*, *RESW* etc., only the *obj-file-offset* is updated reserving their space in the *object* file. Among the assembler directives, the *section* directive is of particular interest. It updates the *Section* table, so that its *pointer-to-content* field holds the *obj-file-offset*, and thus point to the code corresponding to this section.

**EXAMPLE 3.3** Let us look into the code generated in the program shown in Program 2.2, which is reproduced in Program 3.4 for easy reference. To start with, *obj-file-offset* and *lc* are reset. On reaching line 4, it gets a new *section* declaration. Thus, the *Section* table is updated as shown in Fig. 3.12. Next, the *dd* pseudo-opcode is found for *my\_array*. Accordingly, the assembler generates code and writes into the object file. The *db* declaration for *format* is processed similarly. On getting the line *section .text*, the *Section* table is updated (Fig. 3.12). For the next few lines, code is generated and written into the file. For the instruction *JZ over* at line 20, the assembler looks into the symbol table to see the offset of *over* to be 37. Now, the value of *lc* after this instruction is 19. Thus, the distance of *over* from the current position is  $(37 - 19) = 18 = 12$  (in hex) bytes. In the generated code for *JZ over*, the first byte contains the opcode for *JZ*, that is 74 (hex), and the second byte contains 12. During execution of this instruction on an IA32 processor, this 12 will be added to the instruction pointer to compute the jump target. This facilitates program relocation as explained in Chapter 4. For the instruction *JMP L2* at line 26, a similar strategy is followed; however, the distance is negative in this case as the jump target *L2* occurs earlier in the program. For the instruction *CALL printf* at line 31, the address is put as zero, since *printf* is external to the module. An entry is made accordingly in the *external reference list* as shown in Fig. 3.13.

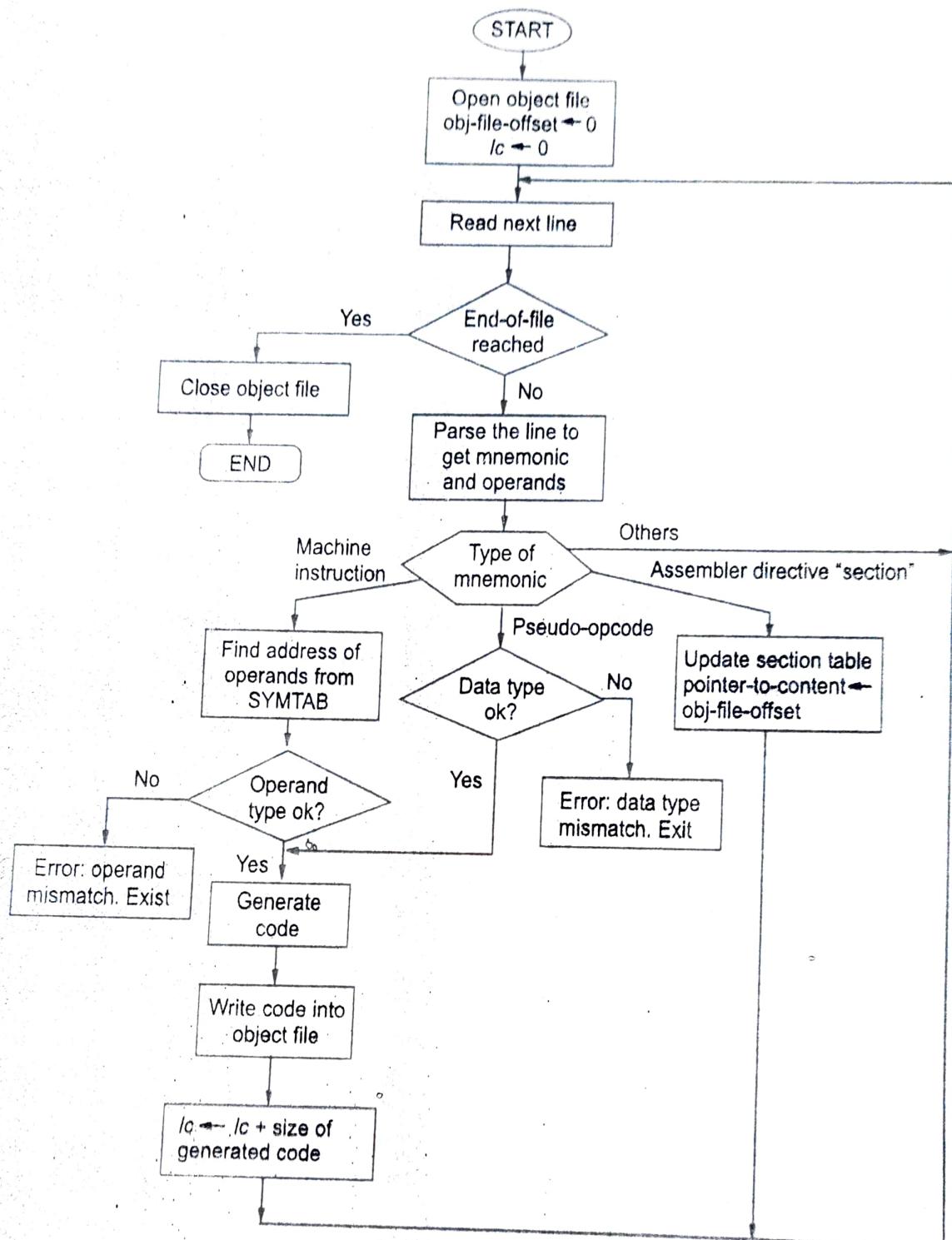


FIGURE 3.10 Pass II of a two-pass assembler.

Name	Offsets to be corrected

FIGURE 3.11 External reference list.

### 3.2. ASSEMBLER DESIGN PROCESS

PROGRAM 3.4: Code generated for the program to find the maximum.

```

1           global main
2           extern printf
3
4           section .data
5           my_array:
6           00000000 0A000000140000001E-
7           00000009 00000064000000C800-
8           00000012 0000380000002D0000-
9           0000001B 004300000059000000-
10          00000024 4D000000
11
12          00000028 25640A00
13
14
15          00000000 B900000000
16          00000005 A1[00000000]
17
18          0000000A 41
19          0000000B 81F90A000000
20          00000011 7412
21
22          00000013 3B048D[00000000]
23          0000001A 7D07
24          0000001C 8B048D[00000000]
25
26          00000023 EBE5
27
28
29          00000025 50
30          00000026 68[28000000]
31          0000002B E8(00000000)
32          00000030 81C408000000
33
34
35          00000036 C3
36
format:
    db '%d', 10, 0
section .text
main:
    MOV ECX, 0
    MOV EAX, [my_array]
L2:
    INC ECX
    CMP ECX, 10
    JZ over
    CMP EAX, [my_array + ECX*4]
    JGE L1
    MOV EAX, [my_array + ECX*4]
L1:
    JMP L2
over:
    PUSH EAX
    PUSH dword format
    CALL printf
    ADD ESP, 8
RET

```

Name	Size	Attributes	Pointer to content
.data	43		0
.text	55		44

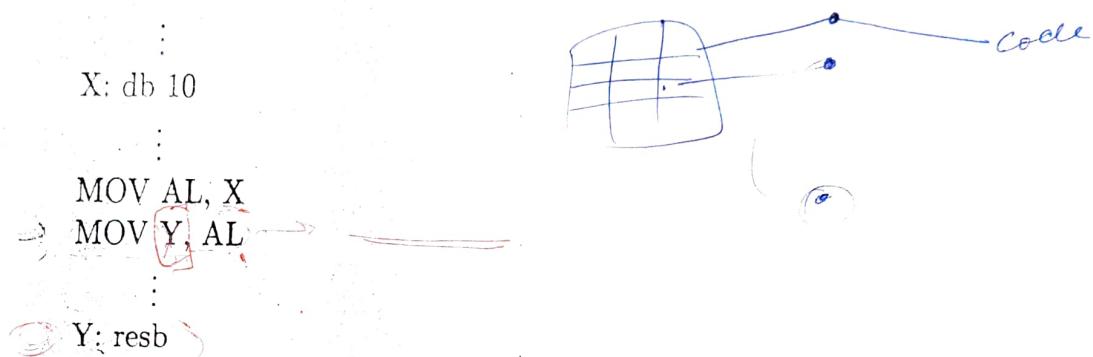
FIGURE 3.12 Updated Section table.

Name	Offsets to be corrected
printf	43

FIGURE 3.13 External reference list.

### 3.2.3 Single-Pass Assembler

A *single-pass* assembler scans the input file only once. Thus, it is generally faster than a two-pass assembler. It uses the same set of data structures as the two-pass assembler. However, it needs some extra data structures and processing to handle the references to symbols defined in the later part of the program (that is, references to symbols occurring before their definitions). For example, consider the following piece of code.



Here, when the assembler reaches the line "X: db 10", it makes entry into the symbol table and simultaneously generates code to reserve one byte initialized to 10. Thus, when the assembler reaches the line "MOV AL, X", the address of X is already available in the symbol table which can be used to generate code. However, at the very next line, "MOV Y, AL" is not simple to process as the type and address of Y is not known at this point of time. These information will only be available when the line "Y: resb" is seen. This type of situation is known as **forward reference**. Similar conditions will arise for branch targets, which are not yet seen by the assembler.

It may be noted that for a two-pass assembler, forward referencing is not a problem, since the entire *Pass I* is dedicated to find the symbols defined in the input program. At the end of *Pass I*, information about all the symbols are available in the *Symbol* table. *Pass II* simply uses these values to generate code.

To resolve the problem of forward referencing, the assembler, upon seeing a symbol not yet defined, should make an entry into the *Symbol* table. The value of the *type* field for the symbol should be inferred from the context. The entry should also be marked as *undefined*. Later on, when the symbol gets defined, it should be checked with the entry already present in the *Symbol* table. Any discrepancy may be resolved accordingly, or reported as an error. However, there exists another potential problem regarding the code generation. An instruction containing forward reference cannot be translated entirely at the time it is encountered. This is because the address of the operand symbol is not available at that time. The address will be known only after the definition has been seen. At that time, all the locations corresponding to the reference of the symbol can be fixed. This process is known as backpatching. For this purpose, corresponding to each such forward referenced symbol, we need to maintain the list

### 3.2. ASSEMBLER DESIGN PROCESS

55.

of locations requiring correction. As and when a symbol gets defined, the corresponding list is traversed and the locations corrected. This list may conveniently be called forward reference list. The structure of this list has been shown in Fig. 3.14.

Name	Offsets to be corrected

FIGURE 3.14 Forward reference list.

A flowchart of the single pass assembler has been shown in Fig. 3.15. As observed from it, the flow of logic combines both *Pass I* and *Pass II* of a two-pass assembler. However, the following extra processings are needed.

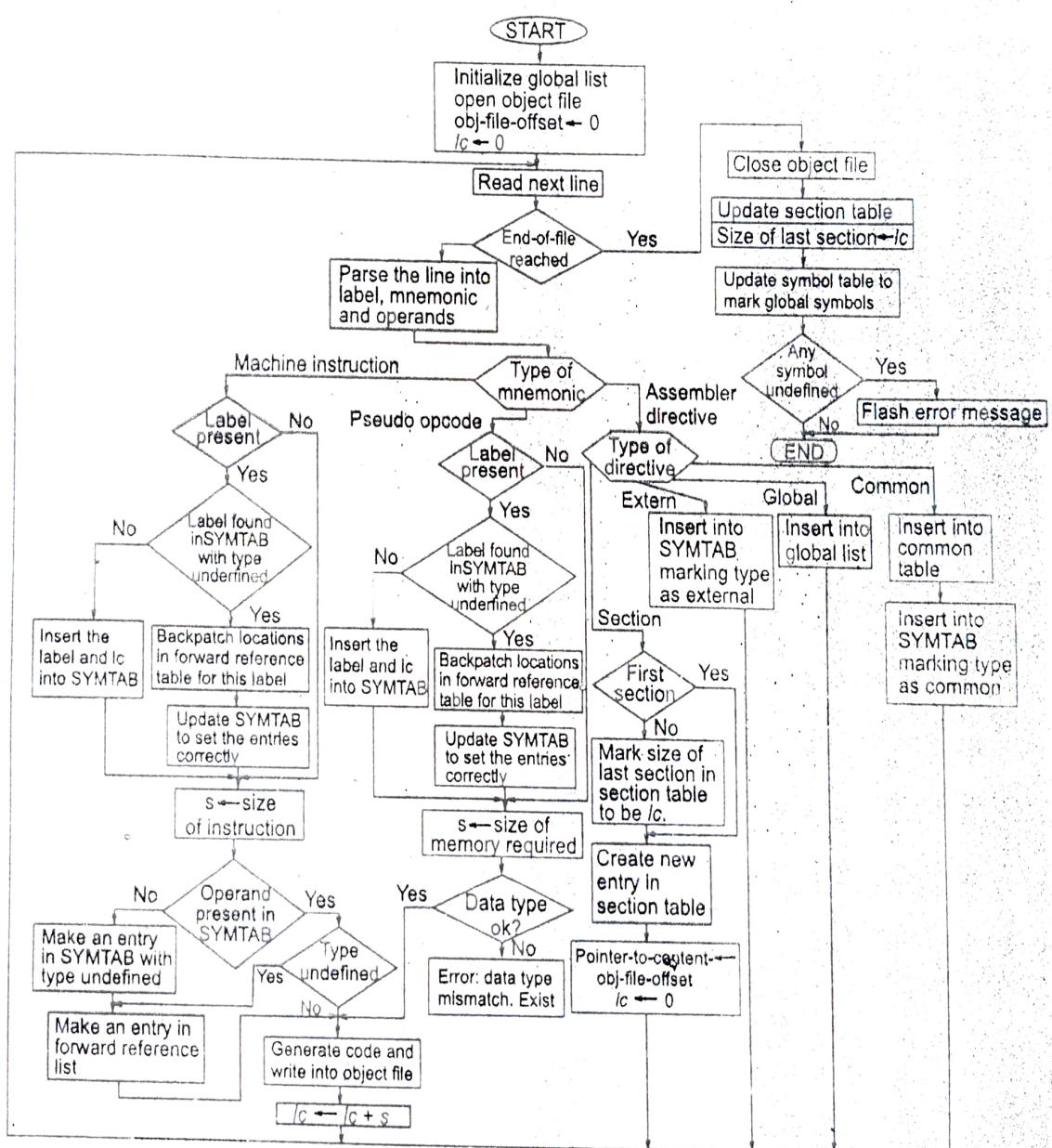


FIGURE 3.15 Single-pass assembler.

LINK AND BORON

1. On reaching the end-of-file marker, apart from closing the object file, updating *Section* and *Symbol* tables, it also needs to check whether some forward referenced symbol is still left undefined. The situation may occur if some symbol is referred to in the program but never defined. At the point of referencing, the assembler assumes it to be a forward referenced symbol and expects it to appear in the later part of the program. An entry is made into the *Symbol* table with type *left* as *undefined*. Thus, at the end of assembly process, if some symbols are still undefined, it is an error and should be reported to the programmer.
2. In the code generation process, while translating an instruction, its operands are looked for in the *Symbol* table. If the operand is not found, a new entry is made into the *Symbol* table with type *undefined*, assuming that it is a forward reference. Also, an entry is created in the *forward reference list* for this label along with the location requiring correction. On the other hand, if the operand is already present in the *Symbol* table with type *undefined*, then the label is searched for in the *forward reference list*, and the current location is added to the list of locations requiring corrections for this label.
3. Whenever a new symbol is defined in the form of a label to some instruction or pseudo-opcode, it is to be entered into the *Symbol* table. At the same time, it should be checked, whether it is a forward referenced symbol or not. If the label is a forward referenced one, the backpatching procedure is invoked to correct the locations noted in the *forward reference table* entry corresponding to this symbol.

**EXAMPLE 3.4** To understand the assembly process better, let us consider the modified program shown in Program 3.5 to find the maximum of a set of numbers. It has been created from Program 2.1 by moving the *data* section to the end of the program to make several instances of forward references. The corresponding list file showing the code generated has been noted in Program 3.6.

To start with, *Global list* is initialized, object file is opened with its offset set to zero, *lc* is also initialized to zero. The source lines are then scanned one-by-one and translation is carried out. Here, for the next two lines corresponding to the *global* declaration of *main* and *External* declaration of *printf*, processing done is similar to the case noted in Example 3.2. Then the *Section* declaration section *.text* is found that creates a new entry in the *Section* table. The field *pointer-to-content* is set to the current *obj-file-offset*, which is zero in this case. As usual, *lc* is reset to zero. The next line is processed by putting the symbol *main* into the *Symbol* table and generating code for the instruction. The next instruction is interesting, as it has a reference to the symbol *my\_array*, which is not yet defined. Thus, a tentative entry is made into the *Symbol* table with type field set to *undefined*. Also, an entry is made in the *forward reference list* marking the location 6 requiring correction. Next, label *L2* is encountered and entered into the *Symbol* table. Codes are generated as usual. Again in the line *JZ over*, *over* is assumed to be a forward reference and entries are made into the *Symbol* table and *forward reference list*. In the next statement, *CMP EAX, [my\_array + ECX \* 4]*, *my\_array* is found in the *Symbol* table; however, its type is *undefined*. Thus, in the entry for *my\_array* in *forward reference list*, another address location 16(hex) is added to indicate a location requiring correction. In the next line, *JGE L1* makes *L1* to be considered as a forward reference and appropriate action is taken to update *Symbol* table and *forward reference list*. The statement *MOV EAX, [my\_array + ECX \* 4]* puts the address 1F(hex) in the *forward reference list* for symbol *my\_array*. The

entries in the *Symbol* table and *forward reference list* upto this point has been shown in Fig. 3.16. Next, label *L1* gets defined. Corrections are made in the *Symbol* table. Consulting the *forward reference list*, code location 1B(hex) is corrected to contain 07(hex). This is because, the *lc* value after the instruction at location 1B(hex) is 1C(hex) and the offset of *L1* is 23(hex) – a distance of 07(hex). Similarly, when the symbol *over* is found, *Symbol* table is corrected and the code location 12(hex) corresponding to the reference of *over* is set to 12(hex) (again by computing the distance). Next, in the statement *PUSH dword format*, *format* is entered into the *Symbol* table and *forward reference table*. Later on, when *my\_array* and *format* are defined, *Symbol* table entries are corrected and backpatching is carried out to correct all code locations noted in *forward reference list*. Figure 3.17 shows the final content of the *Symbol* table and the *forward reference list*.

**PROGRAM 3.5:** Program to find maximum in an array.

```
global main extern printf

section .text
main:
    MOV ECX, 0
    MOV EAX, [my_array]
L2:
    INC ECX
    CMP ECX, 10
    JZ over

    CMP EAX, [my_array + ECX*4]
    JGE L1
    MOV EAX, [my_array + ECX*4]
L1:
    JMP L2

over:
    PUSH EAX
    PUSH dword format
    CALL printf
    ADD ESP, 8

    RET
section .data
my_array:
    dd 10, 20, 30, 100, 200, 56, 45, 67, 89, 77
format:
    db '%d', 10, 0
```

Two pass  
One pass

**PROGRAM 3.6:** List file showing code generated of the program to find the maximum.

```

1           global main
2           extern printf
3
4           section .text
5           main:
6           MOV ECX, 0
7           MOV EAX, [my_array]
8           L2:
9           INC ECX
10          CMP ECX, 10
11          JZ over
12
13          CMP EAX, [my_array + ECX*4]
14          JGE L1
15          MOV EAX, [my_array + ECX*4]
16
17          JMP L2
18
19          over:
20          PUSH EAX
21          PUSH dword format
22          CALL printf
23          ADD ESP, 8
24
25
26          RET
27
28          section .data
29          my_array:
30          dd 10, 20, 30, 100, 200, 56, 45,
31          67, 89, 77
32
33
34          format:
35          db '%d', 10, 0
36

```

*memory*

*run*

### 3.3 LOAD-AND-GO ASSEMBLER

Dunk n book

The assemblers producing the object code directly into the memory and the code is ready for execution are of *Load-and-go* type. This is particularly suitable for small programs in their developmental stages, where after every small modification, it is desirable to check the result. Often, an editor is also clubbed with it to provide an integrated program development

### 3.3. LOAD-AND-GO ASSEMBLER

Name	Type	Location	Size	Section-id	Is-global
printf	external				
main	label	0		1	
my_array	undefined				
L2	label	10		1	
over	undefined				
L1	undefined				

(a)

12	2
6	2
3	1

Name	Offsets(hex) to be corrected
my_array	06, 16, 1F
over	12
L1	1B

(b)

FIGURE 3.16 Partial (a) symbol table, and (b) forward reference list.

Name	Type	Location	Size	Section-id	Is-global
printf	external				
main	label	0		1	
my_array	variable	0	40	2	true
L2	label	10		1	false
over	label	37		1	false
L1	label	35		1	false
format	variable	40	3	2	false

(a)

Name	Offsets(hex) to be corrected
my_array	06, 16, 1F
over	12
L1	1B
format	27

(b)

FIGURE 3.17 (a) Symbol table, and (b) Forward reference list.

environment. The total software development time is reduced since after every modification, the object file need not be stored into the secondary storage and loaded again from there for execution. However, it has a number of drawbacks also as enumerated below.

1. The user program needs to be reassembled each time it is run.
2. The memory occupied by the load-and-go assembler is unavailable for use by the program they process.
3. As an extreme result, a load-and-go assembler cannot be used to reassemble itself!

To avoid these problems, most of the assemblers are equipped with the capability to write their output into a file in the secondary storage. The format of this *object file* should be acceptable to many other system softwares present in the system, particularly the linker. In the following, we discuss about some of the widely accepted object file formats.

### 3.4 OBJECT FILE FORMATS

There are several object file formats reported in the literature, out of which the following are the widely used ones.

- Intel hex format
- obj – Microsoft OMF object files
- win32 – Microsoft Win32 object files
- coff – Common object file format
- elf – Executable and linkable format
- aout – Linux a.out object files

**Intel hex format:** It is a very common blocked object code still in widespread use. In a blocked object code, each sequence of values to be loaded in consecutive addresses is placed in a separate block, along with the address of the first item in the sequence. Intel hex code is a textual code, with each line holding one block of load data, where all of the information on the line is encoded in hexadecimal. Each line must begin with a colon, followed by a 2-digit count of the number of data bytes on the line, a 4-digit address where the data should be stored, and a 2-digit block-type indicator; type 0 is loadable data, and type 1 is used for an end-of-file marker. Following the type-marker comes the data bytes, and then a 2-digit checksum, which is always the last item on the line. The checksum serves to make it possible to detect malformed object files. The checksum is the two's complement of the least significant 8 bits of the sum of all other bytes on the line. Figure 3.18 demonstrates an example of the Intel hex format.

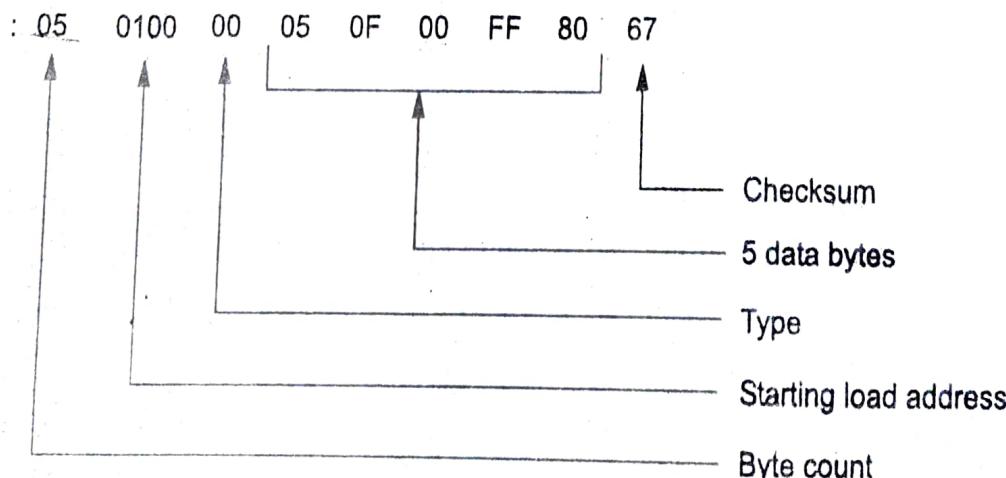


FIGURE 3.18 The Intel hex format example.

**Obj format:** The *obj* file format is the one used in assemblers like MASM, TASM, etc., which is typically fed to 16-bit DOS linkers to produce *.EXE* files. It is also used in OS/2. The default output file extension is *.obj*.

**Win32 format:** This format is used in Microsoft Win32 object files, suitable for passing to Microsoft linkers such as Visual C++. The default output file-name extension is again *.obj*.

**Coff format:** The coff object files are suitable for linking with DJGPP linker.

**Elf format:** This format is used by Linux as well as Unix System V, including Solaris x86, UnixWare and SCO Unix. The default output filename extension is *.o*. It allows the program to be viewed as a collection of *Sections* as shown in Fig. 3.19. An *ELF header* resides at the beginning and holds a description regarding the file organization. *Sections* hold the bulk of object file information, that is, instructions, data, symbol table, *relocation*(to be explained later) information, and so on. It also allows to specify *Position-Independent Code (PIC)* that does not need any modification if loaded at different portions of the memory, and thus useful for the libraries. The concept will be clarified in Chapter 4.

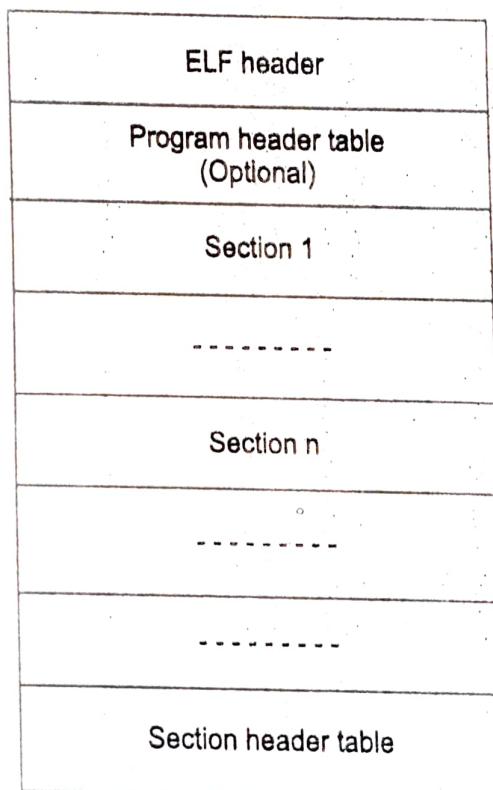


FIGURE 3.19 Elf object file format.

**Aout format:** The *a.out* format generates *a.out* object files, in the form used by early Linux systems. It is a very simple object format, supporting no special directives or symbols. It supports three standard section names *.text*, *.data*, and *.bss*.

### 3.5 CONCLUSION

In this chapter, we have discussed the assembler design process. The major data structures along with their possible organization have been introduced. A detailed discussion has been carried out on both two-pass and single-pass assembler design. In the next chapter, we will see how the object files created through an assembler can be combined along with the library modules to create the executable version of a program using the tools like *linker* and *loader*.

## EXERCISES

- 3.1 Discuss the role of the assembler in program development.
- 3.2 Using the Machine Opcode Table shown in Table 3.1, write a program to find the maximum of ten numbers stored in memory. Hand-assemble the program and show the list file along with the symbol table.
- 3.3 Discuss the major data structures used and their organization in an assembler.
- 3.4 What is forward reference? Explain why the problem of forward reference does not occur in a two-pass assembler, but comes in a single-pass one. How is it solved in the single-pass assembler?
- 3.5 Compare the relative advantages and disadvantages of a single-pass assembler over a two-pass assembler.
- 3.6 What is load-and-go type assembler? How is it helpful in program development?
- 3.7 What are the commonly available object file formats? Discuss their structures in brief.
- 3.8 Distinguish between a simple assembler and a cross-assembler. Do you think that cross-assemblers are definitely needed for system development?
- 3.9 For program in *Exercise 1.8*, show the operation of a two-pass assembler clearly, bringing out the operations carried out in the two passes.
- 3.10 Repeat *Exercise 3.9* for *Exercise 1.9*.
- 3.11 Repeat *Exercise 3.9* for *Exercise 1.10*.
- 3.12 For program in *Exercise 1.8*, show the operation of a single-pass assembler clearly explaining the assembly process.
- 3.13 Repeat *Exercise 3.12* for *Exercise 1.9*.
- 3.14 Repeat *Exercise 3.12* for *Exercise 1.10*.