2. Assemblers

2.1 ELEMENTS OF ASSEMBLY LANGUAGE PROGRAMMING

An assembly language is a machine dependent, low level programming language which is specific to a certain computer system. 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 needs not to know any details of the memory bindings performed by the assembler.
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.

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: This specification refers to the memory word with which the name AREA is associated

- AREA+5: This 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.
- AREA(4): This specification implies indexing with index register 4—that is, the operand address is obtained by adding the contents of register 4 to the address of AREA
- AREA+5(4): This specification is a combination of the previous two specifications.

A Simple Assembly Language

In assembly language, each statement has two operands, the first is always a register which can be any one of AREG, BREG, CREG, DREG. The second operand refers to a memory word using a symbolic name and an optional displacement.

Instruction Opcode	Assembly Mnemonic	Remarks
00	STOP	Stop Execution
01	ADD	Op1 ← Op1+ Op2
02	SUB	Op1 ← Op1 – Op2
03	MULT	Op1 ← Op1* Op2
04	MOVER	CPU Reg ← Memory operand
05	MOVEM	Memory ← CPU Reg
06	COMP	Sets Condition Code
07	BC	Branch on Condition
08	DIV	Op1 ← Op1/ Op2
09	READ	Operand 2 ← input Value
10	PRINT	Output ← Operand2

Figure 1: Mnemonic operation code

Figure 1 lists the mnemonic opcodes for machine instructions.

- The MOVE instruction moves a value between a memory word and a register.
- In the MOVER instruction the second operand is the source operand and the first is target operand.
- Converse is true for MOVEM instruction.
- A comparison instruction sets a condition code analogous to a subtract instruction without affecting the values of its operands.
- The condition code can be tested by a Branch on condition (BC) instruction. The assembly statement corresponding to it has a 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>.

  The condition code specified in a BC statement is encoded into the first operand using the codes 1-5 for the specification LT,LE,EQ,GT,GE and ANY respectively.

Figure 2 shows the machine instruction format. The opcode, register operand and memory operand occupy 2,1 and 3 digits, respectively. The sign is not a part of the instruction.

Figure 2: Instruction format

Figure 3 shows an assembly language program and an equivalent machine language program.

```
START    101
READ     X          101) + 09 0 108
READ     Y          102) + 09 0 109
MOVER    AREG, X    103) + 04 1 108
```

```
                ADD     AREG, Y         104) + 01 1 109
                MOVEM   AREG, RESULT    105) + 05 0 110
                PRINT   RESULT          106) + 10 0 110
                STOP                    107) + 00 0 000
        X       DS      1               108)
        Y       DS      1               109)
RESULT          DS      1               110)
                END
```

Figure 3: an assembly language and equivalent machine language program 1

```
                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 3 116
                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
```

Figure 4: an assembly language and equivalent machine language program 2

2.1.1 Assembly Language Statements

An assembly program consists of 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 transfers into one machine instruction.

Declaration Statements

The syntax of declaration statement is as follows:

[label] DS <constant>

[label] DC '<value>'

The DS 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.

The DC 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

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 e.g. in figure 4, the value of ONE can be changed by executing an instruction MOVEM BREG, ONE

An assembly program can use constants in two ways—as immediate operands, and as literals. Immediate operands can be used in an assembly statement only if the architecture of the largest 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. Simple assembly language does not support this feature, whereas language of Intel 8086 supports it.

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

Figure 5: Use of literals in assembly language

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 program execution. 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. 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 figure 5 —it allocates a memory word to contain the value of the literal, and replaces the use of the literal in 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 known to the assembly language programmer.

Assembler Directives

Assembler directives instruct the assembler to perform certain actions during the assembly of the 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.

2.1.2 Advantages of Assembly Language

The primary advantages of the assembly language programming vis-à-vis machine language programming arise from the use of symbolic operand specifications. Consider the machine and assembly language statements of figure 6. The programs presently compute n!. Figure 7 shows a changed program to compute ½ *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 0 113
        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 3 116
```

```
          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
```

Figure 6: an assembly language and equivalent machine language program (figure 4)

```
          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 3 116
          BC        LE, AGAIN       109) + 07 2 104
          DIV       BREG. TWO       110) + 08 2 118
          MOVEM     BREG, RESULT    111) + 05 2 114
          PRINT     RESULT          112) + 10 0 114
          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 0 002
          END
```

Figure 7: modified assembly and machine language program to compute ½*n!

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

2.2 A SIMPLE ASSEMBLY SCHEME

In this section we use model to develop preliminary ideas on the design of an assembler.

Design specification of an assembler

We use a four step approach to develop 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 the synthesis.

Synthesis Phase

Consider the assembly statement

```
MOVER    BREG, ONE
```

In figure 6, 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 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 address with which the symbolic names 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 6. To determine the address of N, we must fix the address of all program elements preceding it. This function is called memory allocation.

To implement memory allocation a data structure called location counter 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. 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.

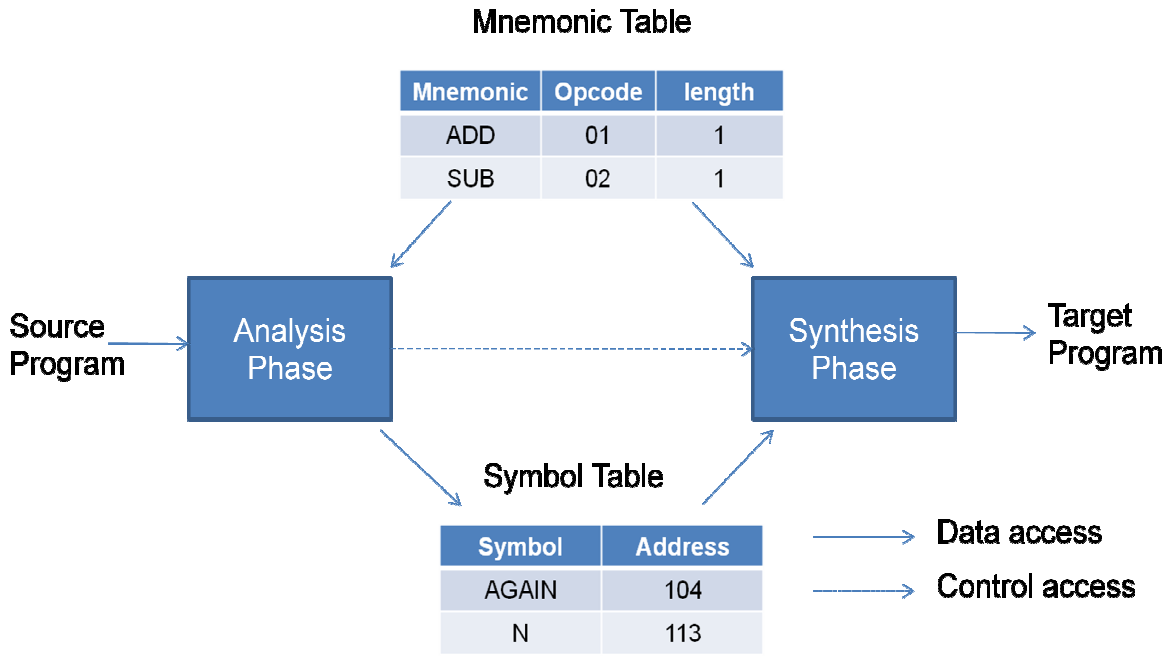


Figure 8: Data structure of the assembler

Figure 8 illustrated 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 counter>) in a new entry of the symbol table
3. Check the validity of the mnemonic opcode through a look-up in the mnemonic 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 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.

2.3 PASS STRUCTURE OF ASSEMBLERS

A pass of a language processor is one complete scan of the source program, or its equivalent representation. 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 synthesis of the target program. The first pass constructs an intermediate representation of the source program for use by the second pass as shown in figure 9. 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.

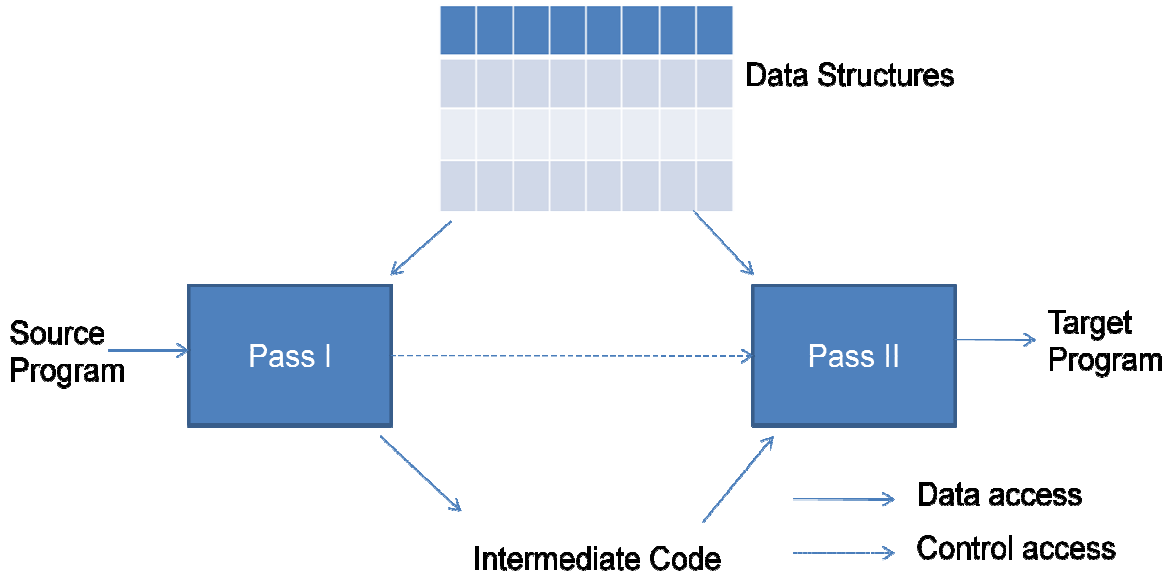


Figure 9: Overview of two pass assembly

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 figure 6, the instruction corresponding to the statement

MOVER BREG, ONE

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>)

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 the 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 the symbol table symb is encountered, all forward references to symb can be processed.

2.4 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.
    2. Build Symbol Table.
    3. Perform LC Processing.
    4. Construct Intermediate Representation.
- Pass II:
    1. Process IR to synthesize the target program.

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

2.4.1 Advanced Assembler Directives

Following instructions are the advanced assembler directives:

- ORIGIN
- EQU
- LTORG

ORIGIN:

The syntax is

ORIGIN <address specification>

where address specification is an <operand spec> or <constant>. This directive indicates that LC should be set to the address given by <address spec>. The ORIGIN is

useful when the target program does not consist of consecutive memory word. 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 1: Statement number 18 of the figure 10 sets LC to the value 204 since the symbol LOOP is associated with the address 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 2 212
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)

Figure 10: An assembly language illustrating ORIGIN

EQU

The syntax is

<symbol> ORIGIN <address specification>

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

The EQU statement defines the symbol to represent <address spec>. No LC processing is implied. Thus EQU simply associates the name <symbol> with <address spec>.

Example 2: Statement 22 of above program viz BACK EQU LOOP introduces the symbol BACK to represent the operand LOOP.

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 2 212
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)
```

Figure 11: An assembly language illustrating EQU

LTORG

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 start of the program or since the last LTORG statement.

Example 3: In program given below, the literals ='5' and ='1' are added to the literal pool in the statement 2 and 6 respectively. The first LTORG statement 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.

```
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 2 212
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)

Figure 12: An assembly language illustrating LTORG

2.4.2 Pass I of the Assembler

Pass I uses the following data structures:

OPTAB : A table of machine opcodes and related information

SYMTAB : Symbol table

LITTAB : A table of literals used in the program.

- OPTAB contains the fields mnemonic opcode, class and mnemonic info. The class indicates whether the opcode corresponds to an IS (Imperative statement), DL(Declaration statement) or AD (Assembler directives). If an imperative statement then the mnemonic info field contains the pair (mnemonic opcode, instruction length) else it contains the id of a routine to handle the declaration or directive statement.
- SYTAB entry contains the fields address and length.
- LITTAB entry contains the fields literal and address.

Processing of an assembly statement:

- Begins with the processing of label field.

- If label fields contains a symbol, the symbol and the value in LC is copied into a new entry of SYMTAB.
- For OPTAB entry, the class field of entry is examined to determine whether the mnemonics belongs to the class of imperative, declaration or assembler directive statements.
- In case of imperative statement, the length the machine instructions is simply added to LC and length is also entered in the SYMTAB.
- For declaration or assembler directives statements, the routine is mentioned in the mnemonic info field is called to perform appropriate processing of the statement.
- The pass I use the LITTAB (Literal Table) 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 END statement, literals in the current pool are allocated addresses starting with the current value in LC and LC is appropriately incremented.

Algorithm for Pass I

1. **location_counter := 0**
   pooltable_ptr := 1
   POOLTAB[1] := 1
   litertable_ptr := 1
2. **While next statement is not end statement**
   a. If label is present then
      this_label := symbol in label field
      Enter (this_label, location_counter) in SYMTAB

   b. If LTROG statement then

i. Process literals LITTAB[POOLTAL[pooltable_ptr]]… LITTAB[literaltable_ptr -1] to allocate memory and put the address in the address field. Update location_counter accordingly.

ii. Pooltable_ptr := pooltable_ptr + 1

iii. POOLTAB[pooltable_ptr] := literaltable_ptr

c. If START or ORIGIN statement then

location_counter := value specified in the operand field

d. If an EQU statement then

i. this_address := value of < address specification

ii. Correct symbol table entry

This_label (this_label, this_address)

e. If a declaration statement then

i. code := code of the declaration statement

ii. size := size of memory area required by DS or DC

iii. location_counter := location_counter + size

iv. Generate IC (DL, code)

f. If an imperative statement then

i. code := machine opcode from OPTAB

ii. location_counter := location_counter + instruction length from OPTAB

iii. If operand is literal then

this_literal := literal in operand field

LITTAB [literaltable_ptr] := this_literal

Literaltable_ptr := literaltable_ptr + 1

Else (i.e. operand is a symbol)

this_entry := SYMTAB entry number of operands

Generate IC (IS, code) (S, this_entry)

3. **Processing of end statement**
   i. **Perform step(2)**
   ii. **Generate intermediate code IC (AD, 02)**
   iii. **Go to Pass II.**

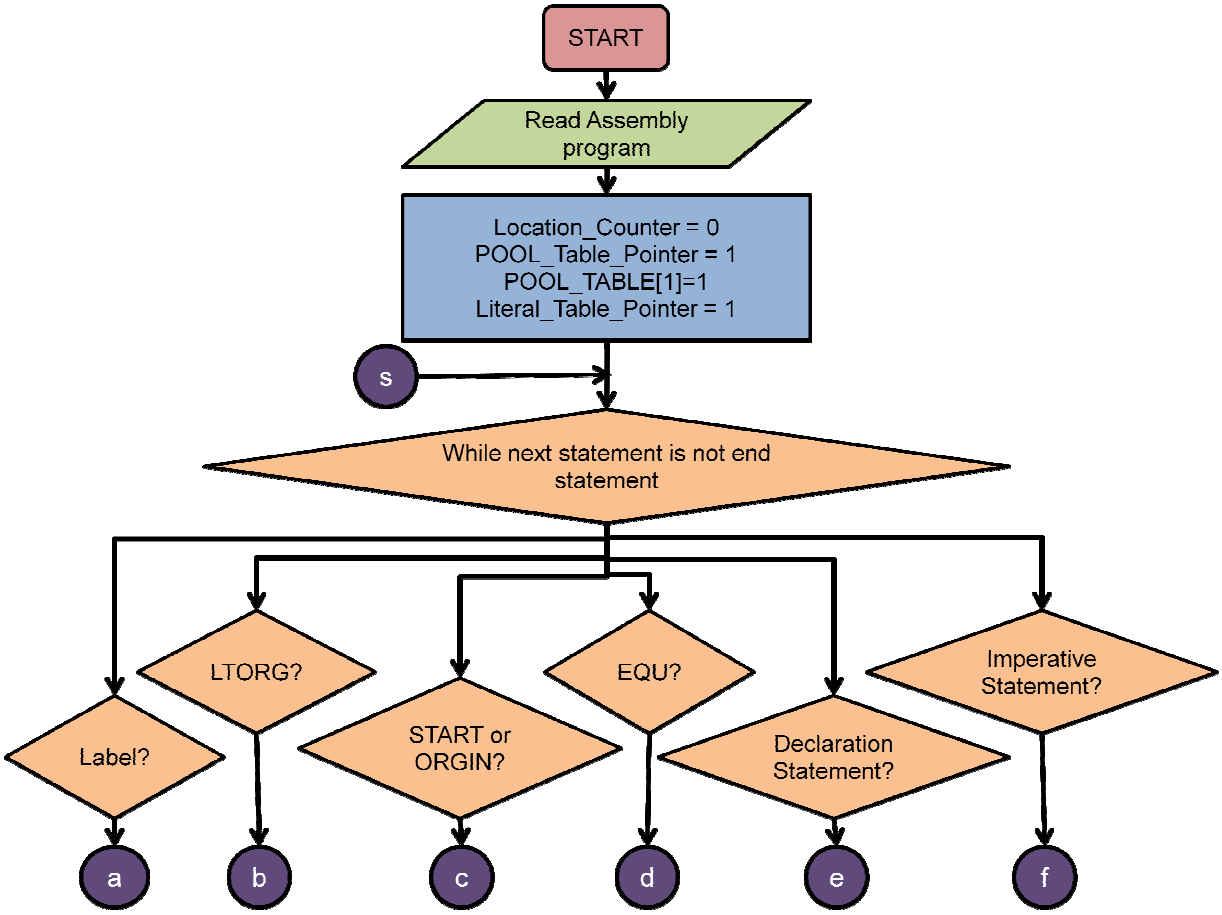


Figure 13: Flowchart for Pass I of two pass assembler

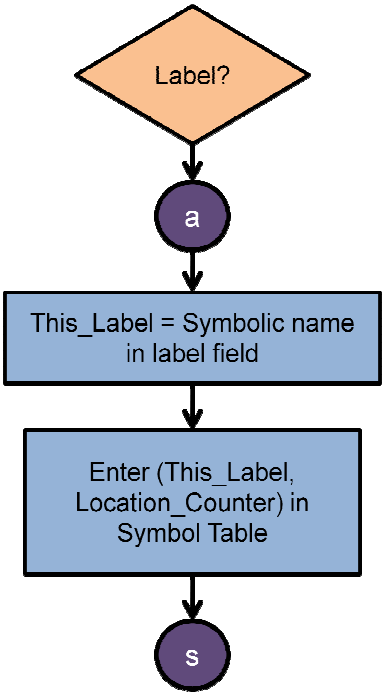


Figure 13 a: Flowchart for processing labels of assembly program

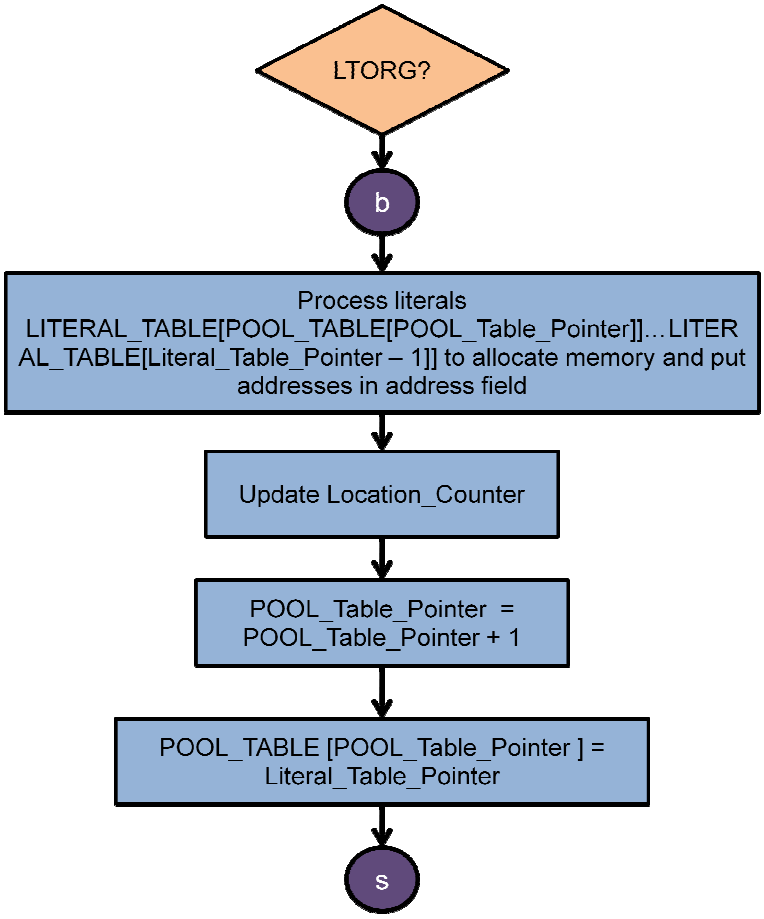


Figure 13 b: Flowchart for processing literals of assembly program

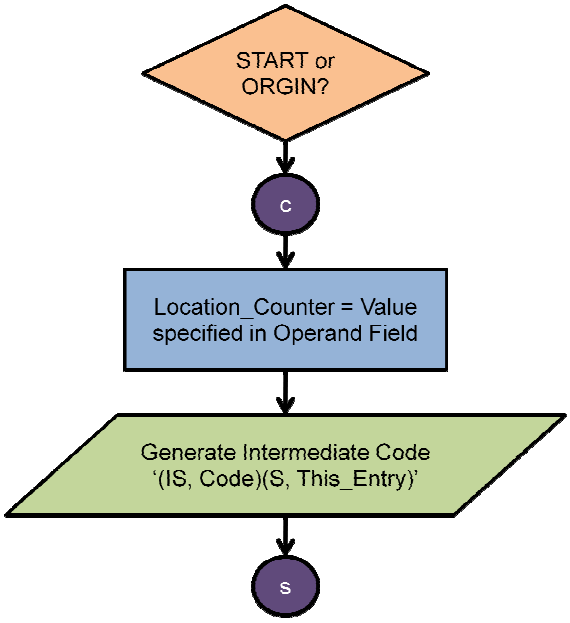


Figure 13 c: Flowchart for processing of START or ORIGIN statement of assembly program

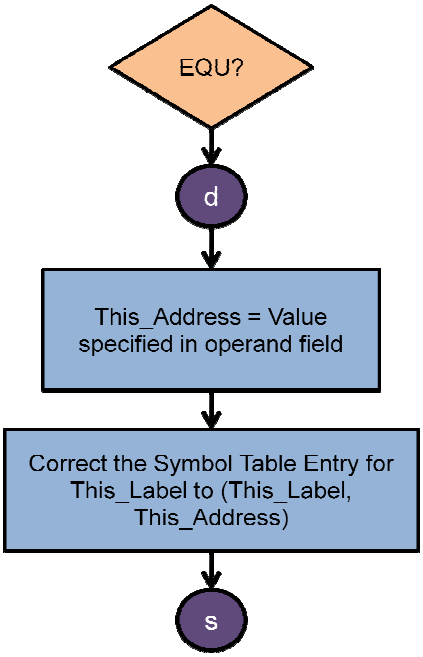


Figure 13 d: Flowchart for processing of EQU statement of assembly program

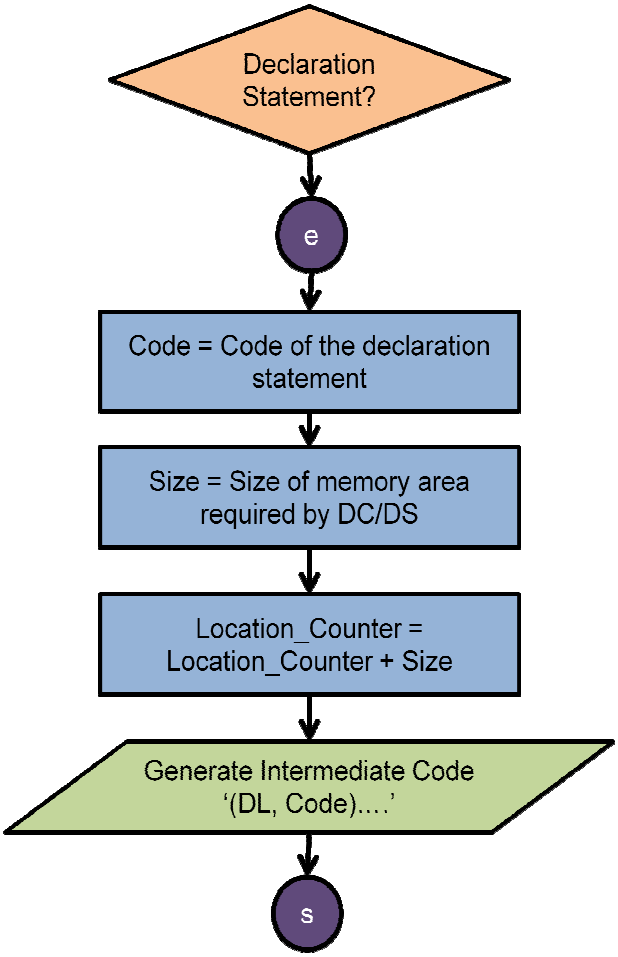


Figure 13 e: Flowchart for processing of declaration statement of assembly program

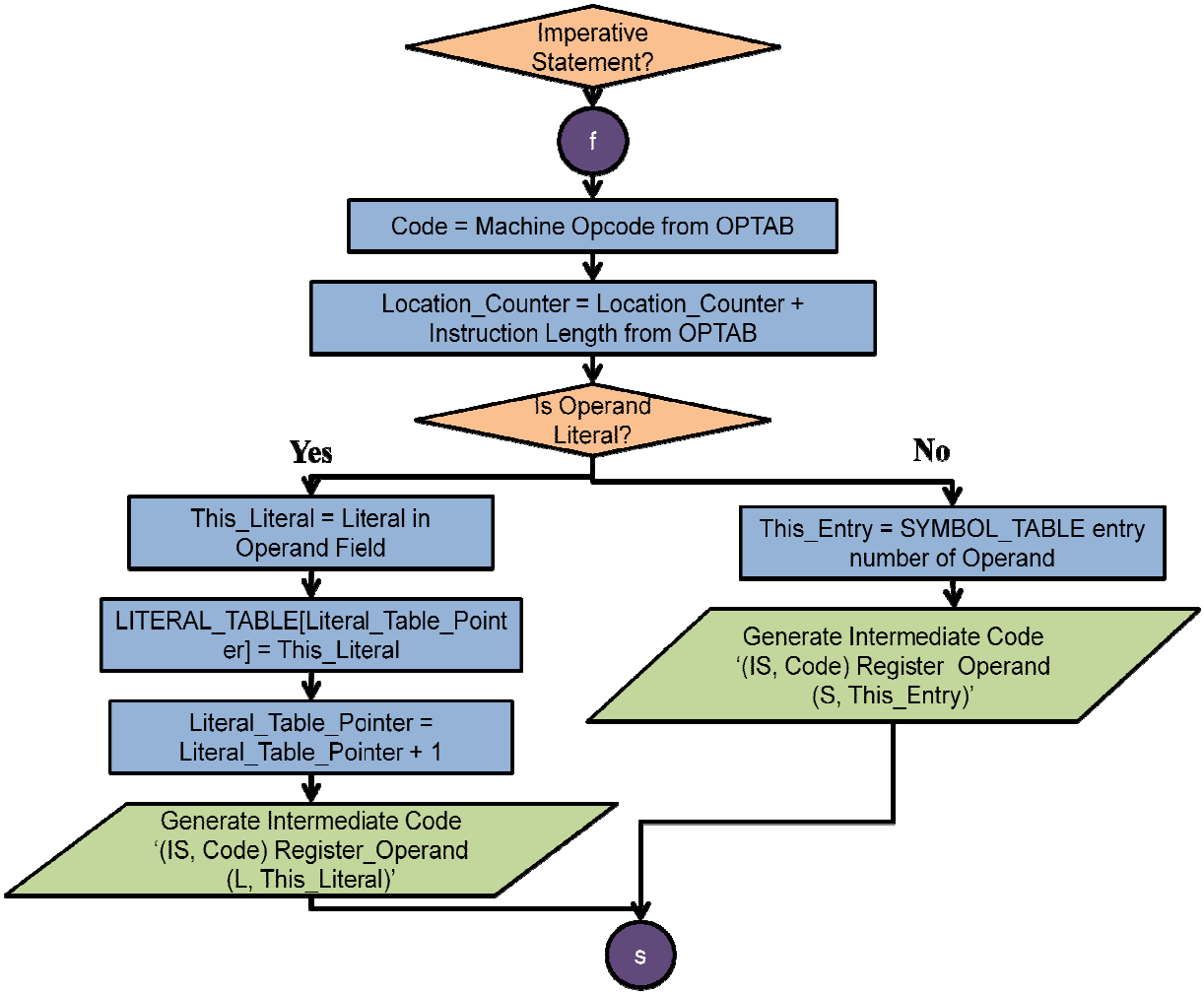


Figure 13 f: Flowchart for processing of imperative statement of assembly program

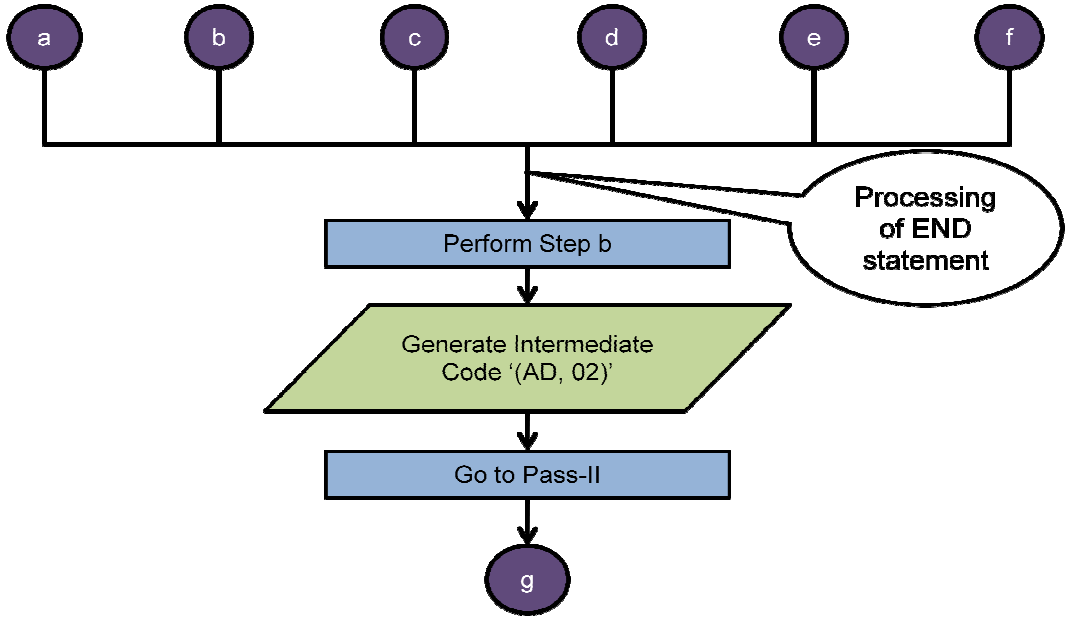
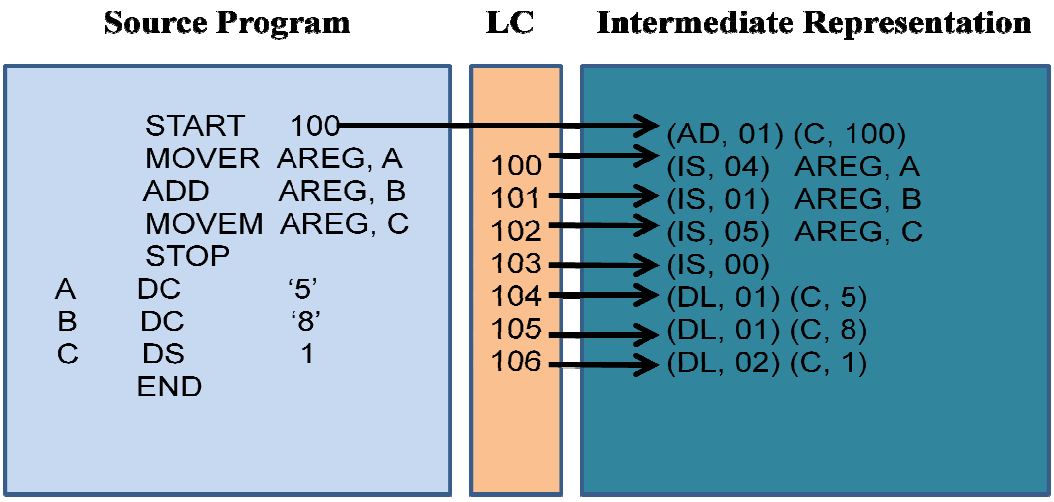


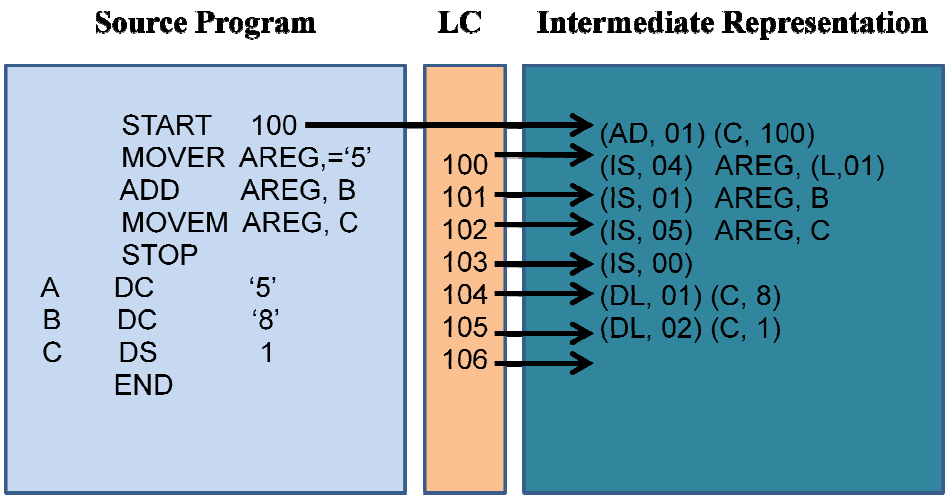
Figure 14: Flowchart for processing of END statement of assembly program



Symbol	Address	Length
A	104	1
B	105	1
C	106	1

Symbol Table

Figure 15 a: Example to generate intermediate representation using Pass I of two pass assembler



Literal_ Table_P ointer	Literal	Address
1	='5'	106
2		

Literal Table

POOL_Table_ Pointer	Literal_Table_ Pointer
1	1
2	2

POOL Table

Symbol	Address	Length
B	104	1
C	105	1

Symbol Table

Figure 15 b: Example to generate intermediate representation using Pass I of two pass assembler

2.4.3 Intermediate Code Forms

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 unit, each consisting of the following three fields.

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

Address	Opcode	Operands

Variant forms of intermediate codes, specifically the operand & address fields, arise in practice due to the tradeoff between processing efficiency & memory economy. These variants are discussed in separate section dealing with the representation of imperative statement, & declaration statement & directive, 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 IS (Imperative statement), DL(Declaration statement) or AD (Assembler directives).

For imperative statement, code is the instruction opcode in the machine language. For declaration and assembler directives, code is an ordinal number within the class. Figure 16 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

Figure 16: Codes for declaration statements and directives

2.4.4 Intermediate Code for Imperative Statements

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

The first operand is represented by a single digit

- 1-4 for AREG-DREG
- 1-6 for LT-ANY

The second operand which is a memory operand is represented by

(operand class, code)

where operand class is one of C (constants), S (symbol), L (literals).

- For constant, the code field contains the internal representation of the constant itself.
- For symbol or literals, code field contain the ordinal number of the operand's entry in SYMTAB or LITTAB

Variant I

In variant I, label, instruction and operand fields are completely processed.

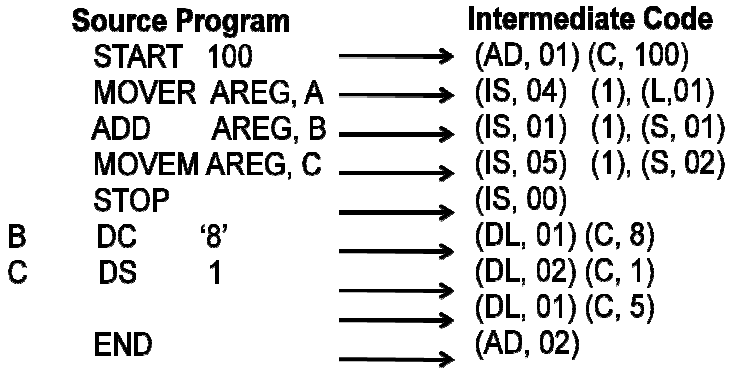


Figure 17: Variant I representation

Disadvantage: In Variant-I, two kinds of entries may exists in SYMTAB at any time for defined symbol and for forward references.

Variant II

In variant II,

- For declarative statements and assembler directives, processing of the operand fields is essential to support LC processing.
- For imperative statements, the operand field is processed only to identify literal references.
- Symbolic references in the source statement are not processed at all during Pass-I

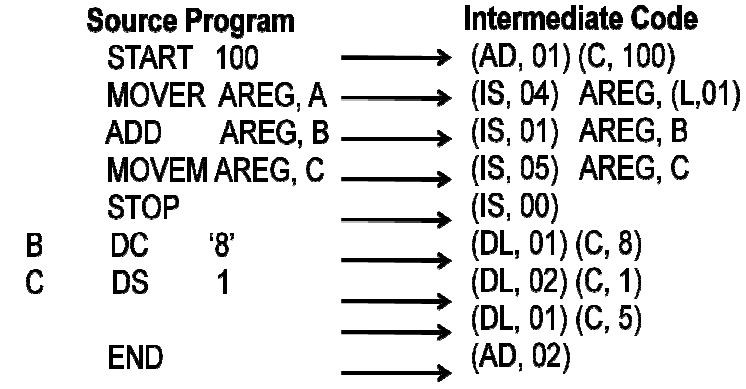


Figure 17: Variant II representation

Comparison of Variant-I and Variant-II

- Variant-I of the intermediate code appears to require extra work in Pass-I.
- IC is quite compact in Variant-I
- Variant-II reduces the work of pass-I by transferring the burden of operand processing from Pass-I to Pass-II of the assembler.
- IC is less compact in Variant-II.

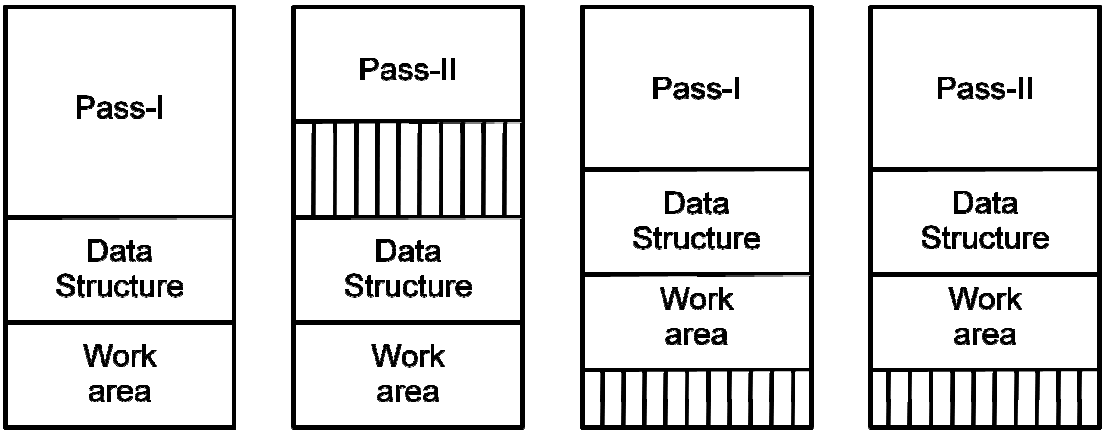


Figure 18: memory requirements using Variant I and Variant II

2.4.5 Processing of Declaration and Assembler Directives

Here we consider the alternative ways of processing declaration statement and assembler directives.

- **DC**: 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.
- **START or ORIGIN**: It is not necessary to retain START and ORIGIN statement in 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 literal 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.
- Pass I could itself copy out the literals of the pool into the IC. 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.

2.4.6 Pass II of the Assembler

Following algorithm is the algorithm for assembler Pass II

Algorithm for Pass II

1. **code_area_address := address of code area**
   pooltable_ptr := 1
   location_counter := 0
2. **While next statement is not an end statement then**
   a. Clear machine_code_buffer

   b. If an LTORG statement then
      i. Process literals in LITTAB[POOLTAB[pooltable_ptr]]…….. LITTAB[POOLTAB[pooltable_ptr + 1] -1 similarly 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. pooltable_ptr := polltable_ptr +1

   c. if a START or ORIGIN statement then
      i. location_counter := value specified in operand field
      ii. size := 0

d. If a declaration statement then
    i. If a DC statement then
        Assemble the constant in machine_code_buffer
    ii. size := size of memory area required by DC or DS

e. If an imperative statement then
    i. Get the 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 the contents of machine_code_buffer to the address code_area_address + location_counter
    ii. location_counter := location_counter +size

3. Processing end statement

a. Perform steps 2(b) and 2(f)

b. Write code_area into output file

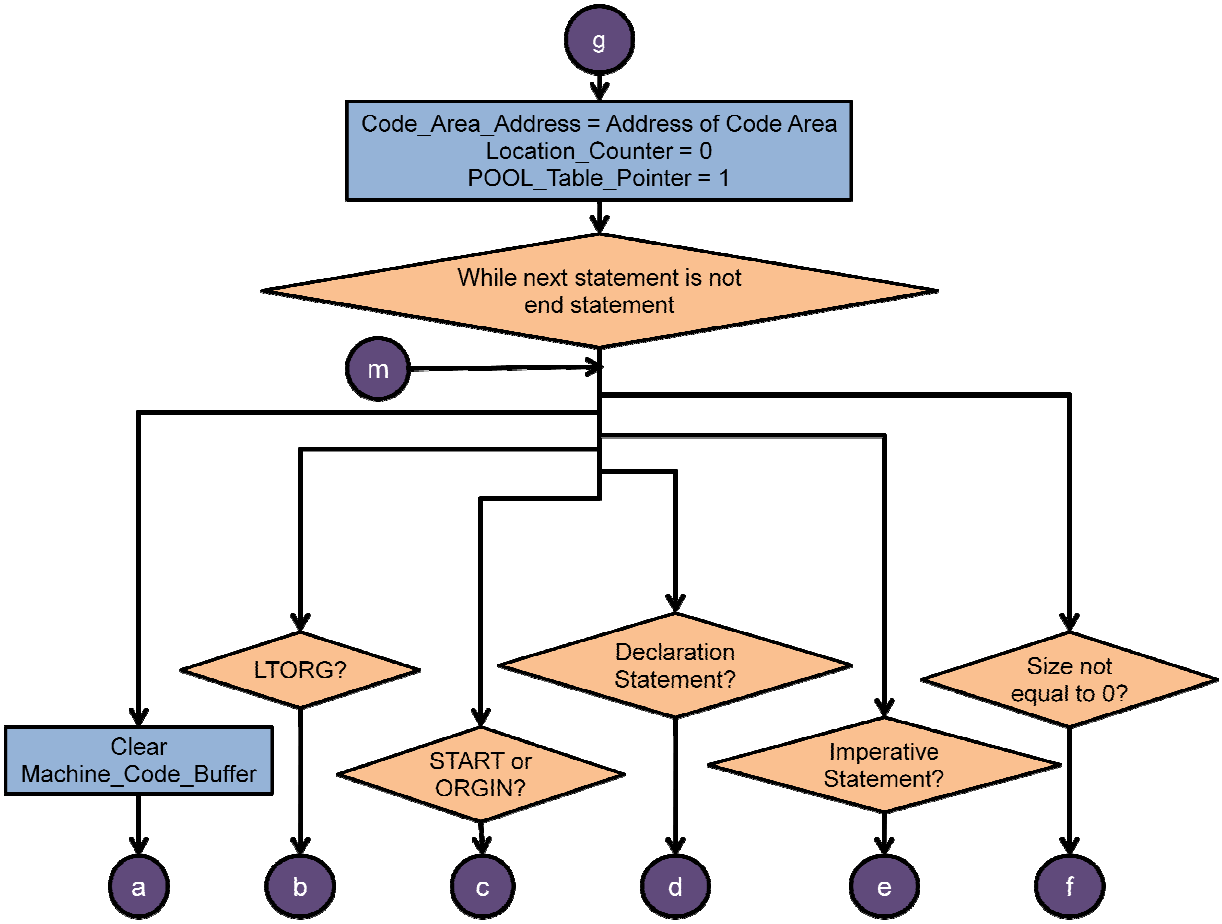


Figure 19: Flowchart for Pass I of two pass assembler

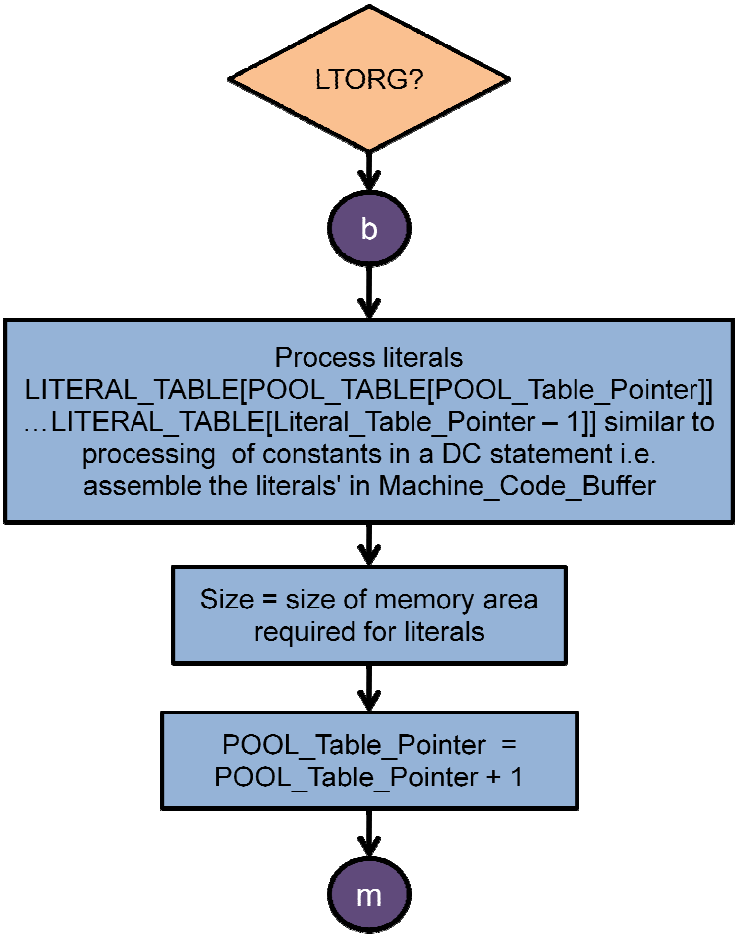


Figure 19 b: Flowchart for processing LTORG statement of assembly program

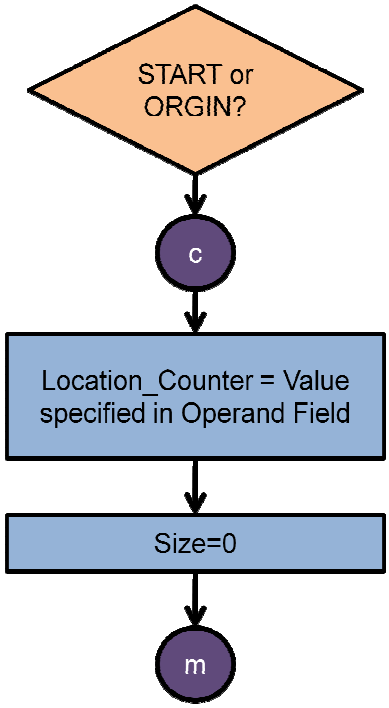


Figure 19 c: Flowchart for processing START or ORIGIN statement of assembly program

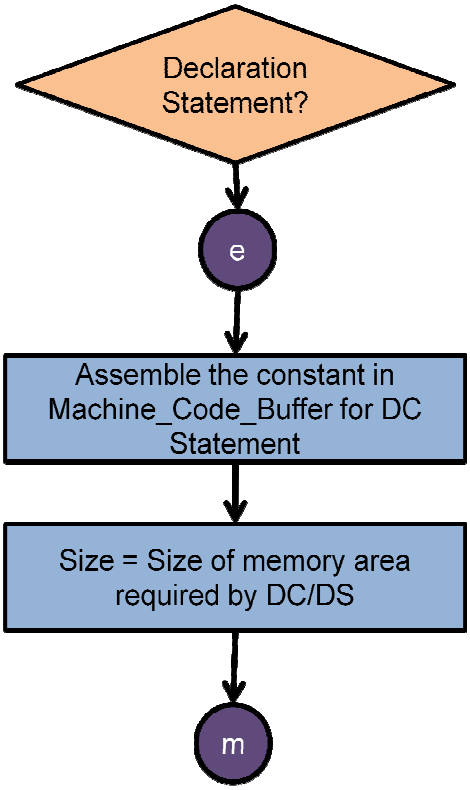


Figure 19 d: Flowchart for processing declaration statement of assembly program

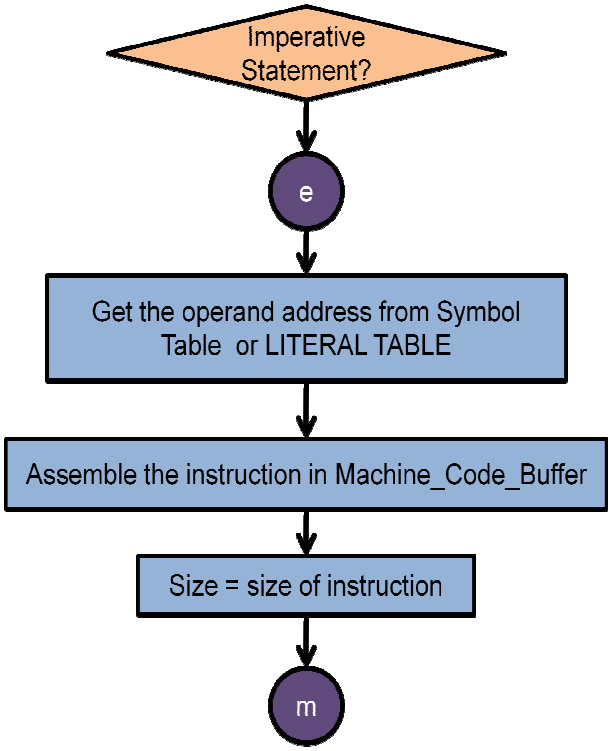


Figure 19 e: Flowchart for processing imperative statement of assembly program

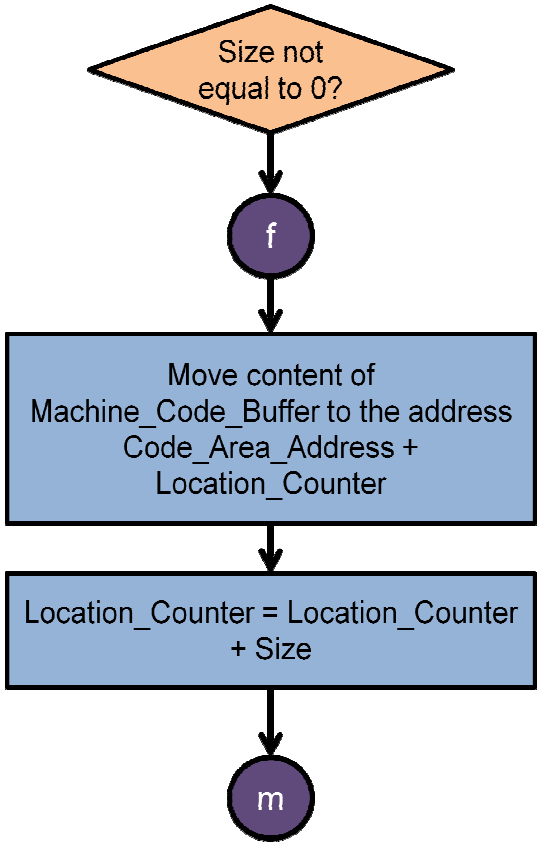


Figure 19 f: Flowchart for processing statement having size not equal to zero of assembly program

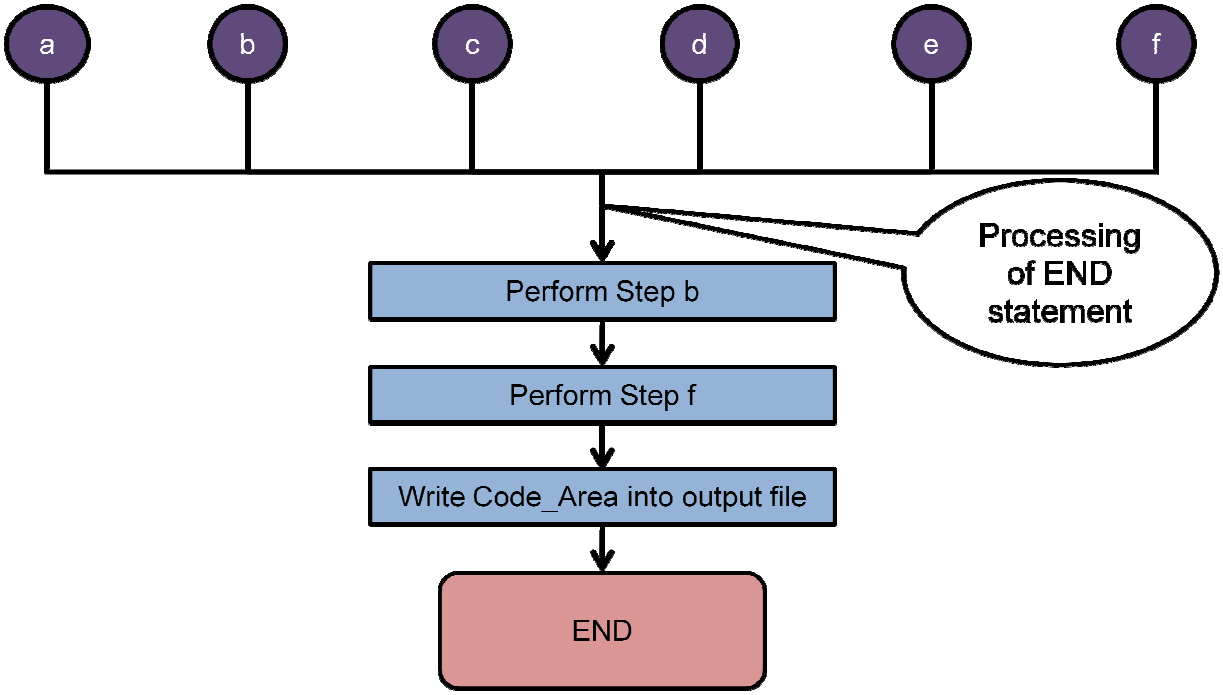
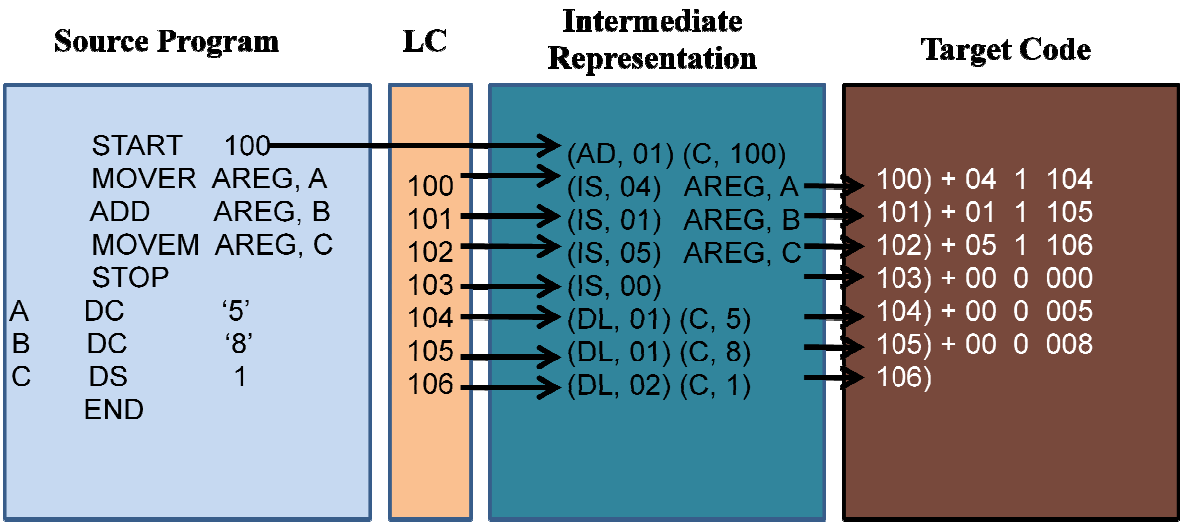


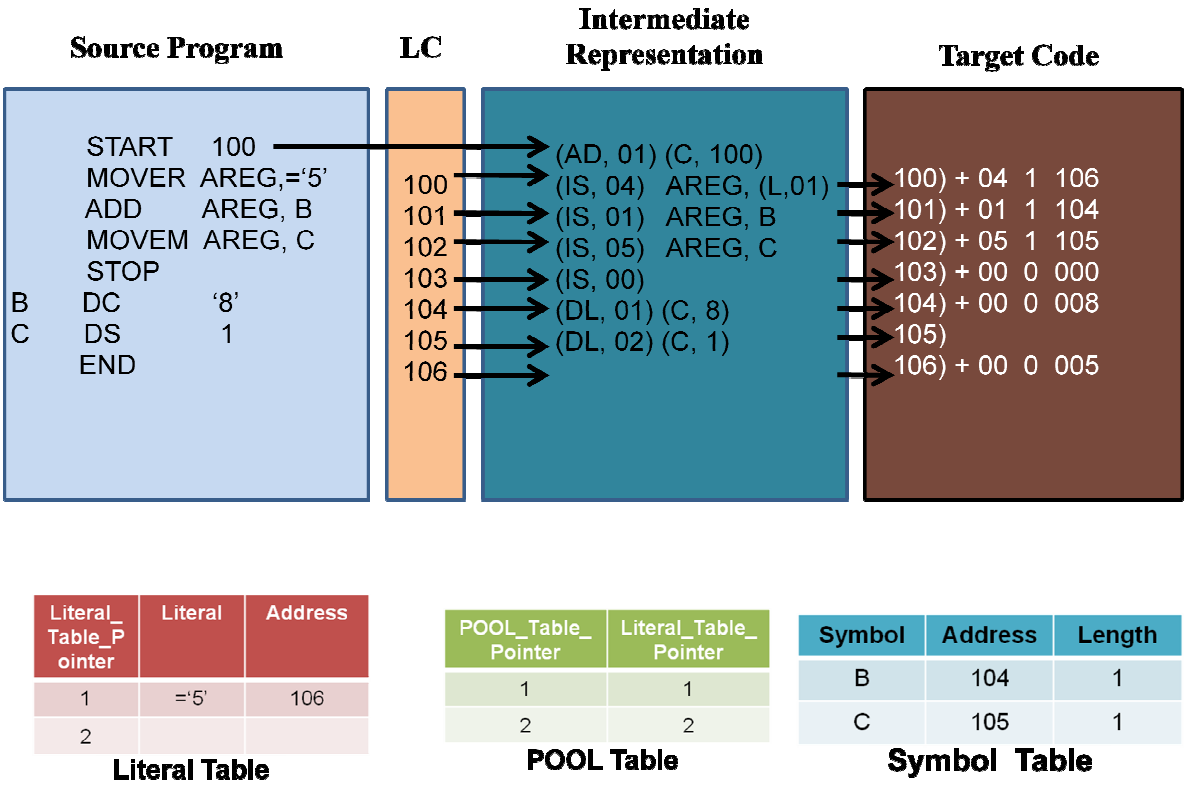
Figure 20: Flowchart for processing END statement of assembly program



Symbol	Address	Length
A	104	1
B	105	1
C	106	1

Symbol Table

Figure 21 a: Example of converting source program to machine language



Literal_ Table_P ointer	Literal	Address
1	='5'	106
2		

Literal Table

POOL_Table_ Pointer	Literal_Table_ Pointer
1	1
2	2

POOL Table

Symbol	Address	Length
B	104	1
C	105	1

Symbol Table

Figure 21 b: Example of converting source program to machine language

Output interface of the assembler

The assembler produces a target program which is the machine language of the target computer. This is rarely. The assembler produces an object module in the format required by a linkage editor or loader.

2.4.7 Listening and Error Reporting

The basic decision is whether to produce program listening and error reports in Pass I or delay these actions until Pass II. Producing the listening 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 listening 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 symbol. Other errors like references to undefined variables can only be represented at the end of source program (see below table). 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.

Example 4: Figure 22 illustrates the 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 mnemonics in OPTAB. In statement 21, A is known to be a duplicate definition. Use of 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.

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**			

Figure 22: 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 listening and error reporting till Pass II. Now the error reports as well as the target code can be printed against each source statement.

Example 5: Figure 23 illustrates error reporting performed in pass-II. Indication of errors in statements 9 and 21 is easy. Indication of error in statement 10 is equally easy- the symbol table is searched for an entry of B and error is reported when no matching entry is found. The target program instruction 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 DEFINATION OF SYMBOL A**			
022		:			
035		END			

Figure 23: Error reporting in pass II

2.4.8 Some Organizational Issues

Here we discuss some organizational issues in assembler design like the placement and access of tables and IC with respect to the schematic shown in figure 24.

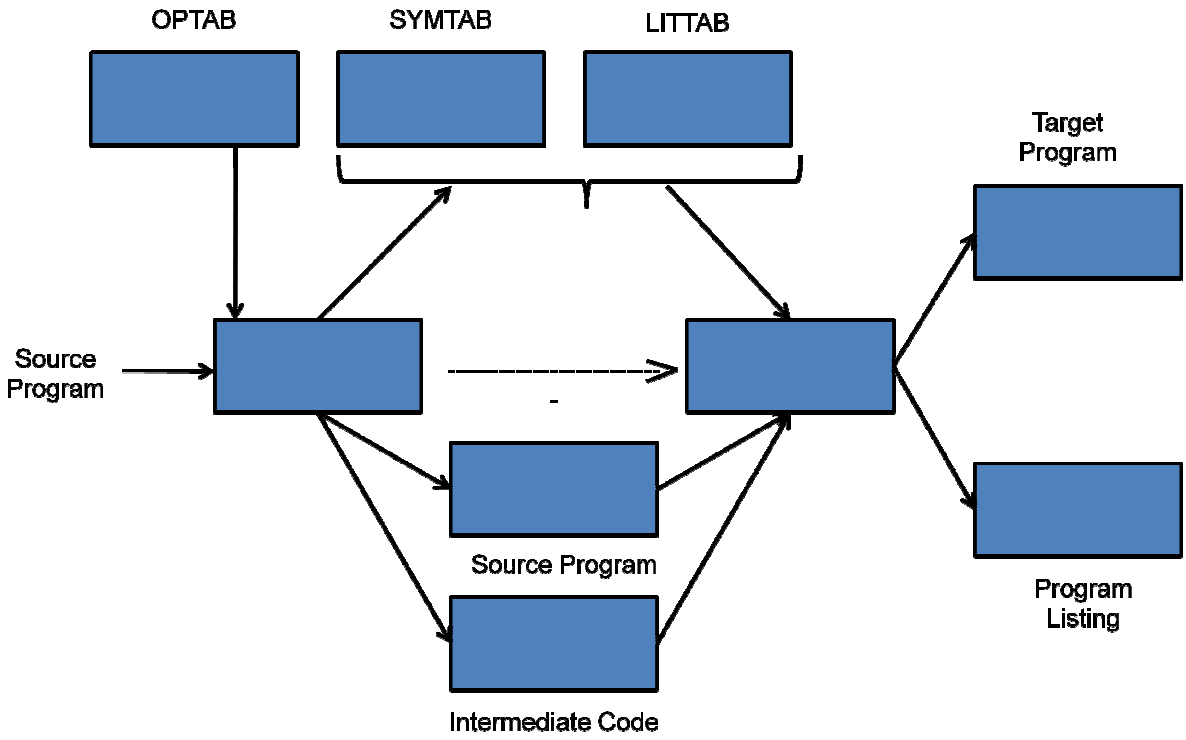


Figure 24: Data structures and files in a two pass assembler

Tables

- For efficiency reason, 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 memory. If memory is at a premium, it is possible to hold only part of LITTAB in memory because only the literals of the current pool need to be accessible at any time.
- 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 source program would also be written into another file.
- The target code and program listing can be written out as separate files by pass II.

2.5 A SINGLE PASS ASSEMBLER FOR IBM PC

2.5.1 Architecture of Intel 8088

The Intel 8088 microprocessor supports 8 and 16 bit arithmetic and also provides special instruction for string manipulation. CPU contains following features:

- **Data Registers AX, BX, CX and DX:** 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.
- **Index Registers SI and DI:** The index registers SI and DI are use to index the source and destination addresses in string manipulation instructions
- **Stack Pointers Registers SP and BP:** These two pointers are used to address 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 desired manner. Push and Pop instructions are provided for this purpose.
- **Segment registers Code Stack, Data and Extra:** The memory of the Intel 8088 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 address of these three components. The extra segment registers points to another memory area which can be used to store data.

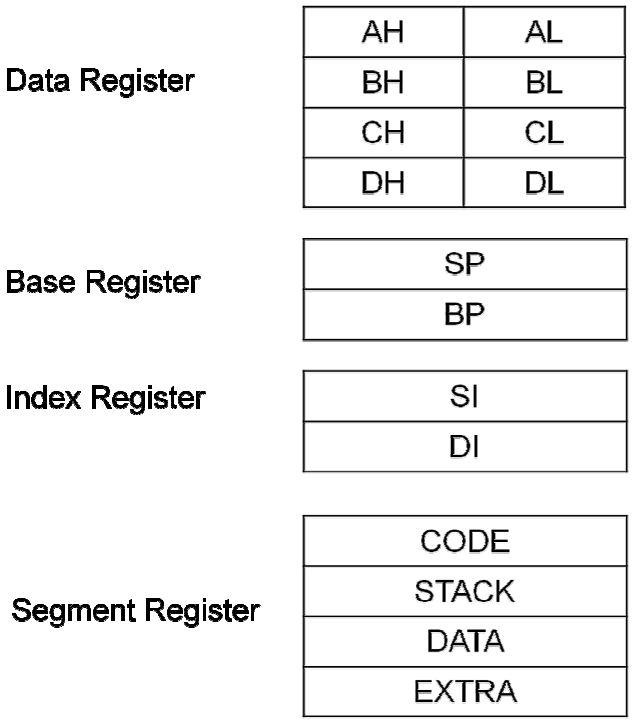


Figure 25: Data, Base, Index and Segment registers

The 8088 architecture provides 24 addressing modes as shown in the following figure.

Addressing Modes	Examples	Remarks
Immediate	MOV SUM, 1234H	Data = 1234H
Register	MOV SUM, AX	AX contains data
Direct	MOV SUM, [1234H]	Data Displacement = 1234H
Indirect	MOV SUM, [BX]	Data Displacement = (BX)
Register Indirect	MOV SUM, CS: [BX]	Segment Override: Segment Base = (CS) Data Displacement (BX)
Based	MOV SUM, 12H[BX]	Data Displacement = 12H + (BX)
Indexed	MOV SUM, 34H[SI]	Data Displacement = 34H + (SI)
Based and Indexed	MOV SUM, 56H[SI][BX]	Data Displacement = 56H + (SI) + (BX)

Figure 26: Addressing modes of 8088

2.5.2 Assembly Language of Intel the 8088

Statement format

The format of an assembly statement is as follows:

[label1:] opcode operand(s) ; comment string

where the label is optional. In the operand field, operands are separated by commas. 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].

Assembler directives

The ORG, EQU and END directives are analogous to the ORIGIN, EQU and END directives. 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 & PURGE

EQU defines symbolic names to represent values or other symbolic names. 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: 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 an assembly program consists of a set of segments. SEGMENTS and ENDS directives demarcate the segments in an assembly program. To assembly symbolic reference, the assembler must determine the offset of the symbol from start of segment containing it. To facilitate this, the programmer must perform the following actions in an assembly program: (a) load a

segment register with segment base, and (b) let the assembler know which segment register contains segment base. The second task is performed using the ASSUME directive, which has the syntax

ASSUME <register> : <segment name>

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

Example: Consider the following program

```
                SAMPLE_DATA     SEGMENT
                ARRAY           DW              100  UDP  ?
                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 DS register. While assembling the statement MOV_AX, SUM the assembler first computes the offset of SUM from 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 DS register. Note that it is the programmers responsibility to ensure that the correct address is loaded in the DS register before executing the reference to SM. 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 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 near or far call. A RET statement must appear in the body of procedure to return execution control to the calling program. Parameters for the called procedure can be passed through registers or on the stack.

Example: 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 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 operator:-

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, OFFST, TYPE, SIZE and LENGTH. SEG and OFFSET provide the segment and offset components of the memory address of an operand.

Example: 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(bytes),-(near instruction) and -2(far instruction), SIZE indicates the number of bytes allocated to the operand.

Example: 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 PTR and THIS operator. The PTR operator

creates a new memory operand with the same segment and offset address 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 the allocation.

Example: 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 the 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
```

2.4.3 Problems of Single Pass Assembly

A single pass assembler for Intel 8088 shares some problems with other single pass assembler's viz. problems in assembling forward reference and error reporting. The forward reference problem is aggravated by the nature of 8088 architecture. We discuss two aspects of this problem.

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 CS,LENGTH STRNG	0005
006		MOV COUNT,0000	0000
007		MOV SI, OFFSET STRING	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	

Figure 27: Sample assembly program of Intel 8088

Forward Preferences

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. This entry would identify the bytes in

code where the address of the referenced symbol should be put. When the symbols' definition is encountered, this entry would be analyzed to complete the instruction. However, use a symbolic name as the destination in a branch instruction gives rise to 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 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 figure 27 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 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 the 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 (segsymb, offsetsymb) where segsymb is the name of the symbol containing the definition of symb.
- It uses the information in SRTAB to find the register which contains segsymb. Let it be register r. it now synthesizes the pair(r,offsetsymb). This pair is put in the address field of the target instruction.

However, this strategy would not work while assembling forward references. **Example:** Consider statements 6 and 13 in figure 27 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 segment 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 the 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 the counter srtab_no.
2. Instead of TII, a forward reference table (FRT) is used. Each entry of FRT contains the following entries:
   i) Address of the instruction whose operand field contains the forward reference.
   ii) Symbol to which forward reference is made.
   iii) Kind of reference (e.g. T: analytic operator TYPE, D:data address, S: self relative address, L: length, F: offset, etc)
   iv) Number of the SRTAB to be used for assembling the reference.

Example: Two SRTAB's would be built for the program of figure 27. 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 the 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).

4.5.5 Design of the Assembler

LC processing in this algorithm differs from LC processing in the first pass of a two pass assembler 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 DR 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 assemblers are illustrated in figure 28, where numbers in the parenthesis indicate the number of bytes required for a field.

- The mnemonics table is hash organized and contains the following fields: mnemonic opcode, machine opcode, alignment info and routine id. The routine id field of an entry specifies the routine which processes the opcode. Alignment 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 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 whether the

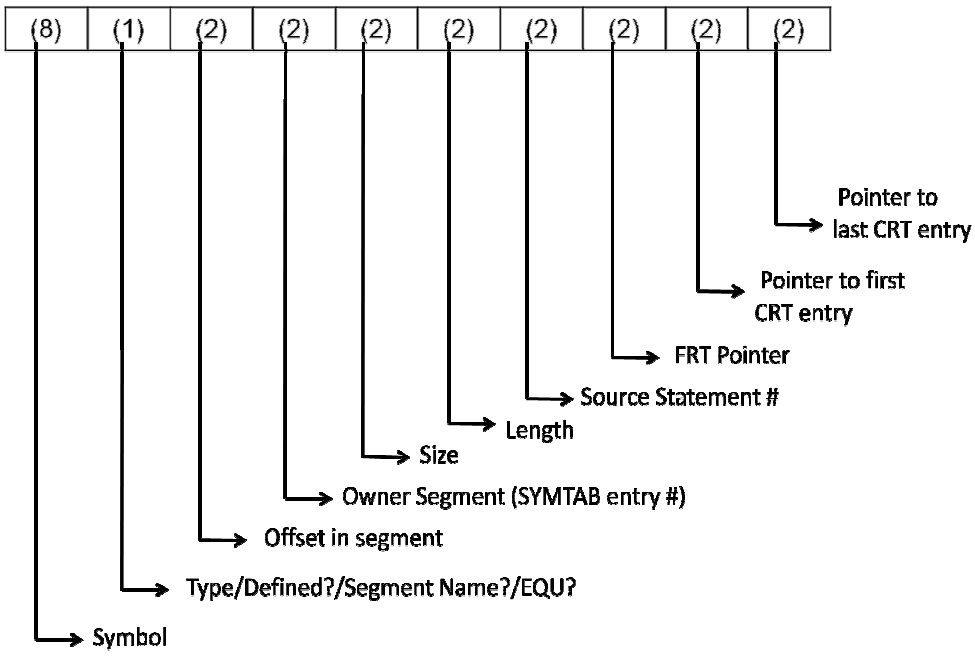
symbol is to be given a numeric value or a textual value. The owner segment and offset fields are used to accommodate the value.

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

Mnemonics table(MOT)

Mnemonic Opcode (6)	Machine Opcode (2)	Alignment/ Format info (1)	Routine Id (4)
JNE	75H	00H	R2

Symbol Table (SYMTAB)



Segment register table array (SRTAB_ARRAY)

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

Forward reference table(FRT)

Pointer (2)	SRTAB # (1)	Instruction Address (2)	Usage Code (1)	Source Stmt# (2)

Cross Reference Table (CRT)

Pointer (2)	Source Stmt# (2)

Figure 28: Data structures of the assembler

Forward References

Information concerning forward references to a symbol is organized in the form of a linked list. Thus the 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 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 are processed, and the forward references list is discarded. This minimizes the size of FRT at any time.

Cross References

A cross references directory is a report produced by the assembler which lists all references to a symbol sorted in the ascending order of the 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 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 from low end to the high end of the storage.

Example: Figure 29 illustrates contents of important data structures after processing statement 19 of the source program of figure 27.

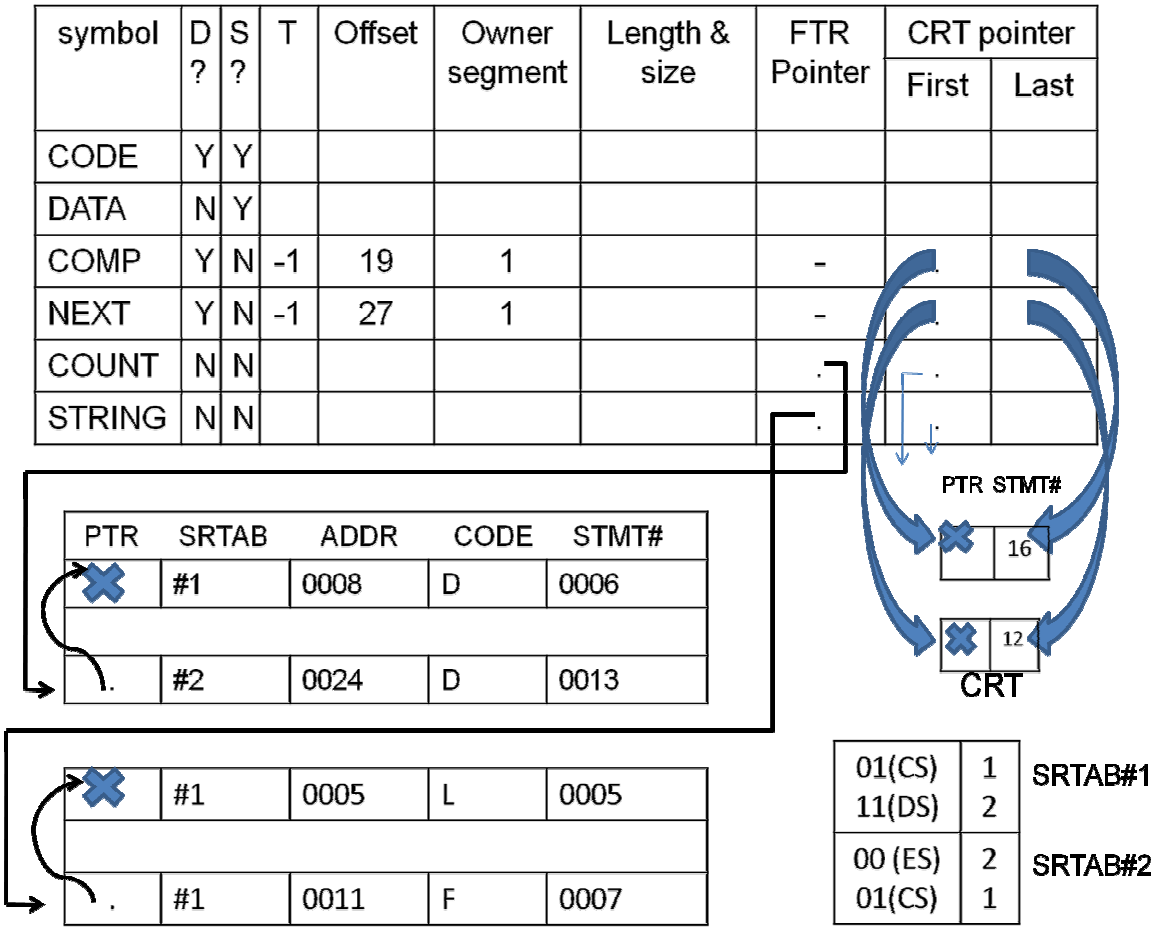


Figure 29: Data structures after processing statement 19.

- 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 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 COUNT in the source program are forward references. Hence two entries exist

for COUNT in FRT and CRT. The first FRT entry has #1 in the SRTAB field, while the second has #2 in it.

- Similarly two FRT and CRT entries exist for STRNG. The usage code fields of the FRT indicate what information is required in the referencing instruction, e.g. data address ('D'), self-relative address, length, offset, etc.
- Subsequent processing of the program is as follows: at the end of processing statement 20 its label, viz. COUNT will be looked up in the SYMTAB. An entry exists for it with defined='no'. This implies that count has been forward referenced. Its segments 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 machine instruction with offset 0024. The first forward reference however, expects a word alignment for COUNT 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 the COUNT are destroyed.

Listing and error detection

The program listing and error reporting function faces the problems.

- The program listing can contain the statement in the source form, its serial number in the program and the memory address assigned to it. The target code can only be printed 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

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

2. Whenever a symbols 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 (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 ERRORTAB, 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 level field;

c) If an EQU statement
i. this_address := value of operand expression;
ii. Make an array for this _lable 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 lable in the operand field.
iv. Process forward reference to this_lable.
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 statement pearing in operand field.
v. Make the entry (this_register, this_segment) in SRTAB_ARRAY[srtab_no] . (this overwrites an existing entry for this_register.)

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 the CRT list of symb.
ii. If operand symbol is already defined then
Check its alignment & addressability.

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 the 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 & 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 symbol from SYMTAB.

b) Produce cross reference listing.

c) Write code_area into output file.

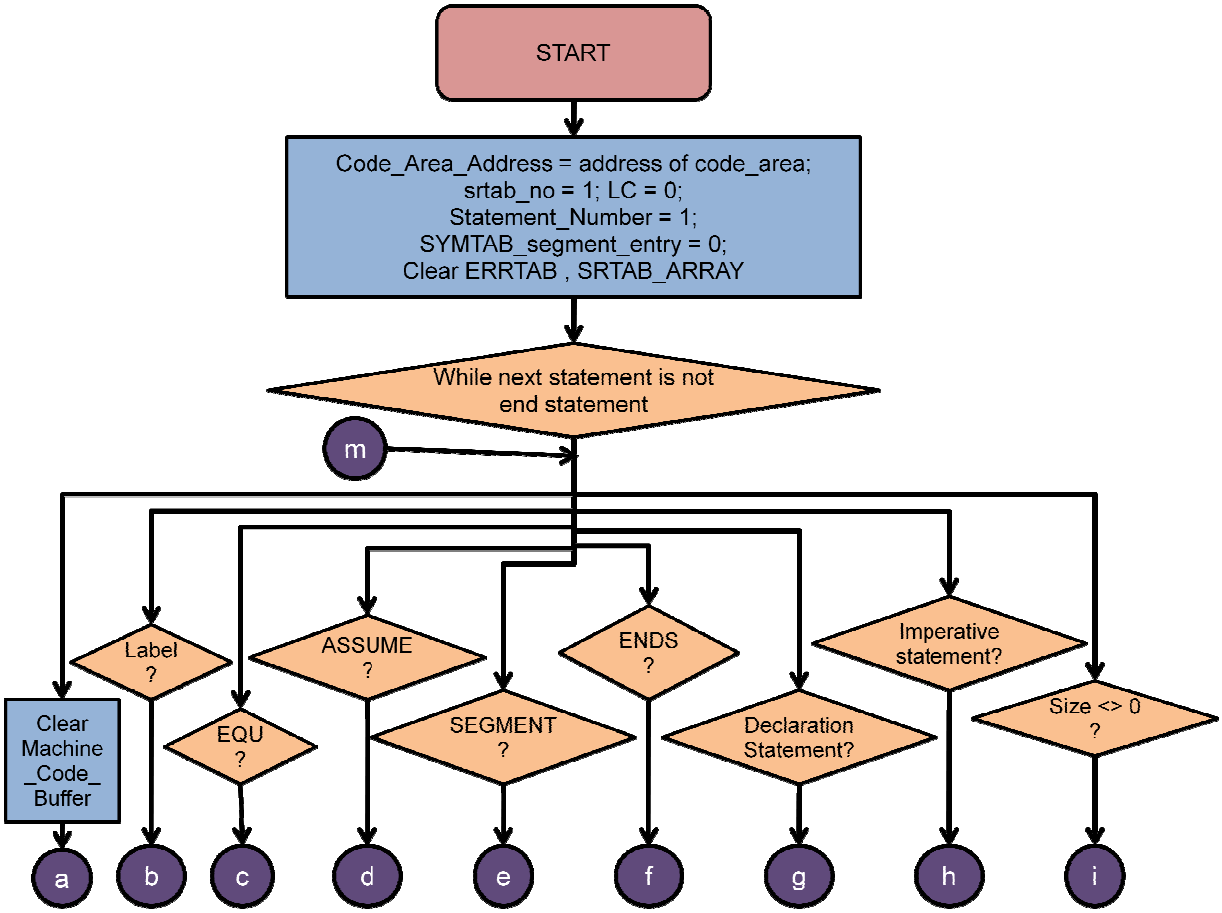


Figure 30: Flowchart for single pass assembler

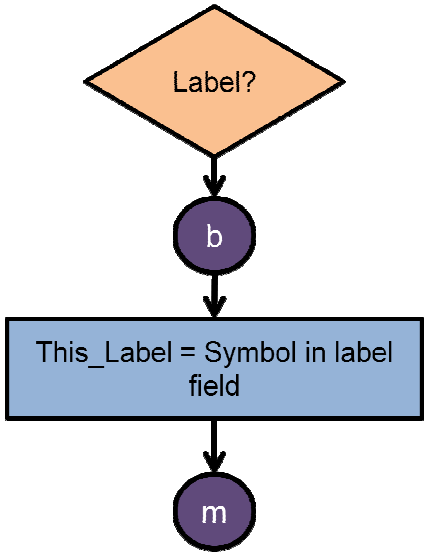


Figure 30b: Flowchart for processing the label of assembly program

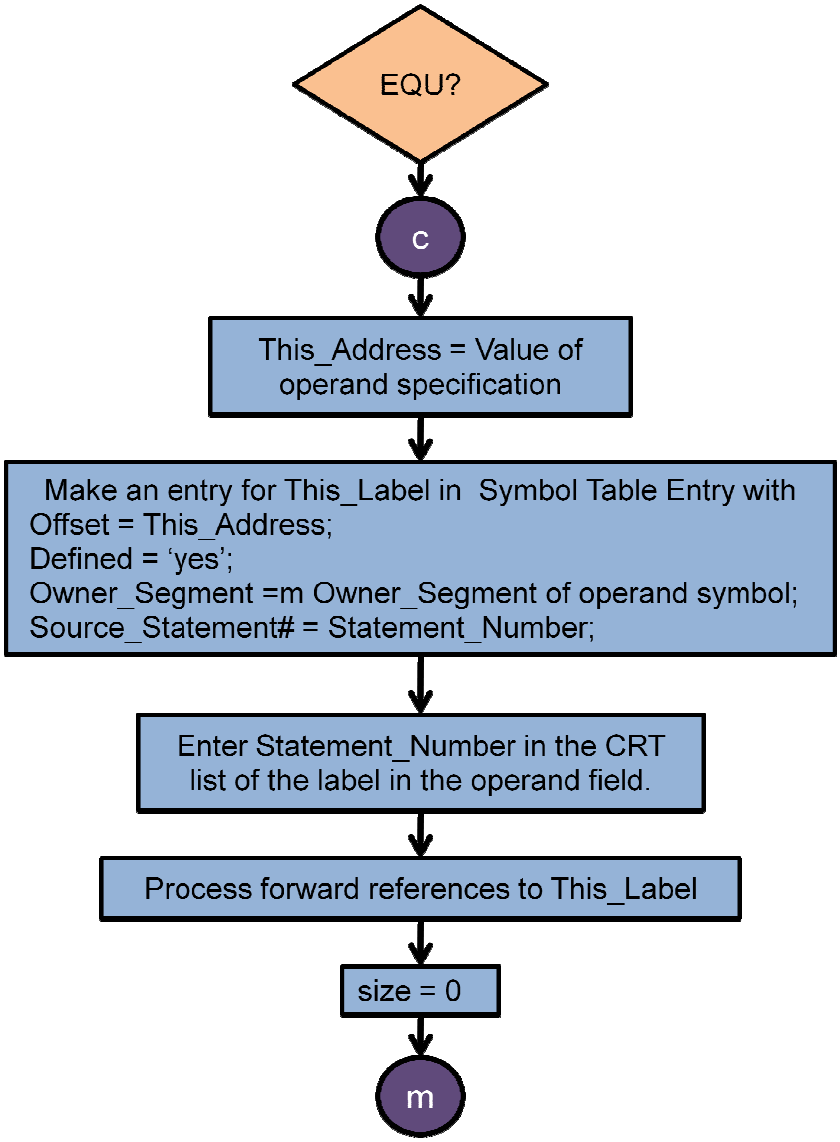


Figure 30c: Flowchart for processing the EQU statement of assembly program

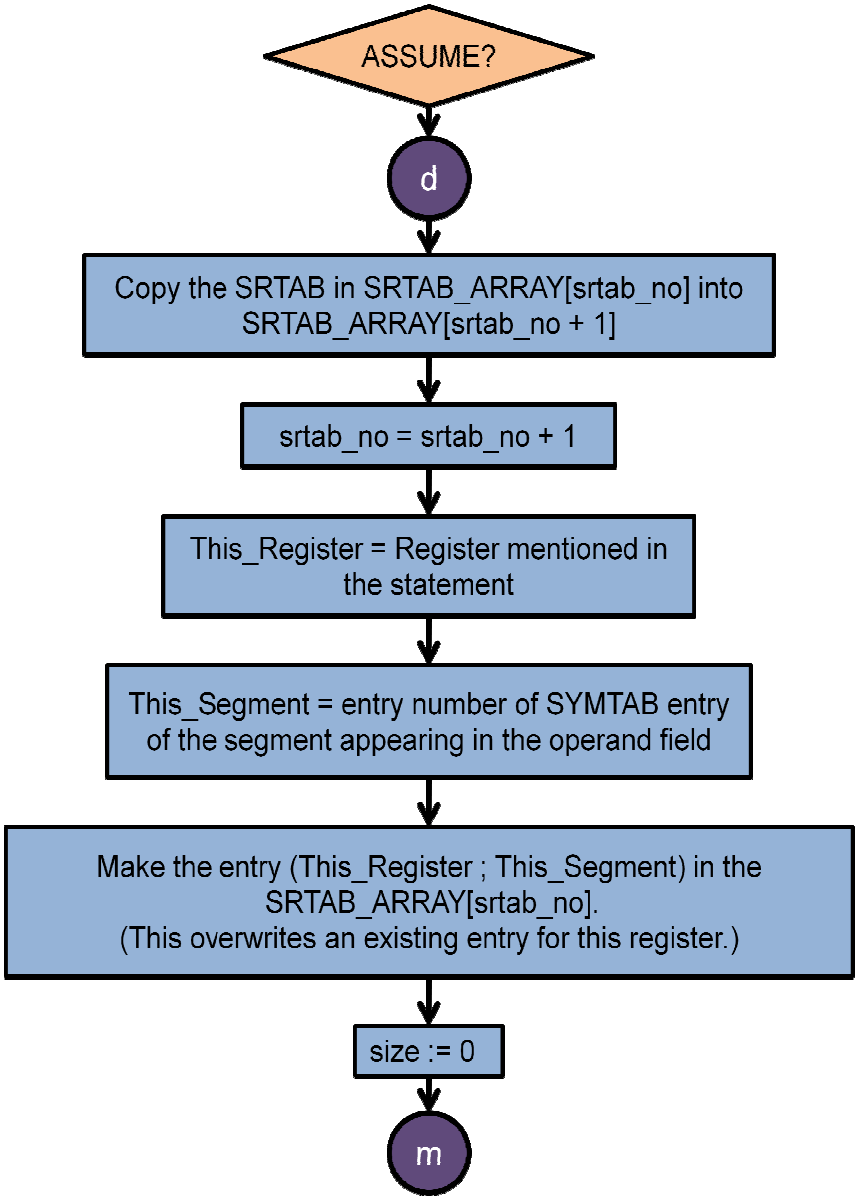


Figure 30d: Flowchart for processing the ASSUME statement of assembly program

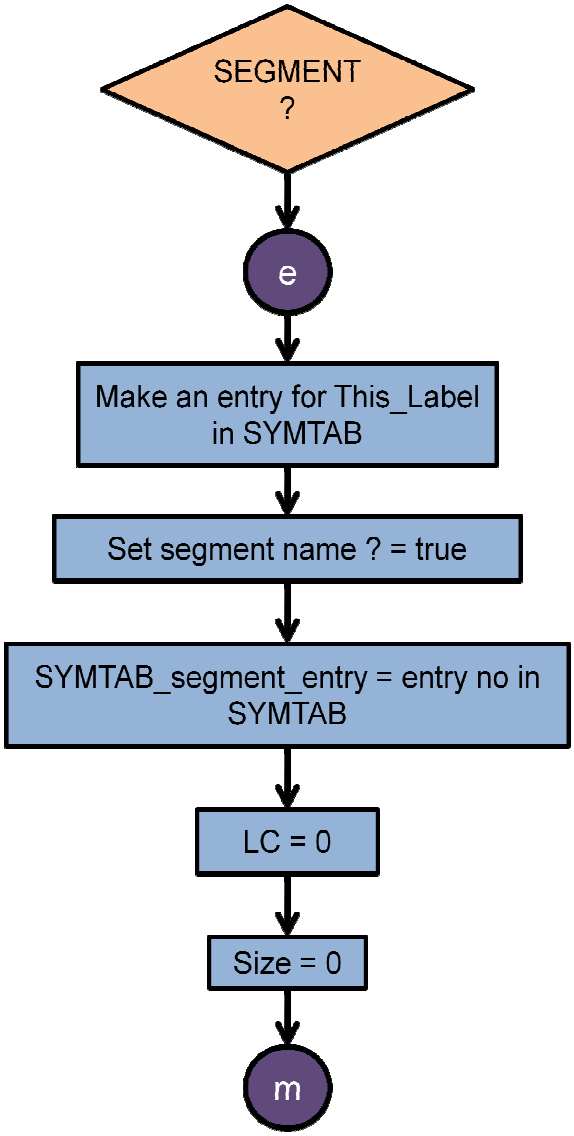


Figure 30e: Flowchart for processing the SEGMENT statement of assembly program

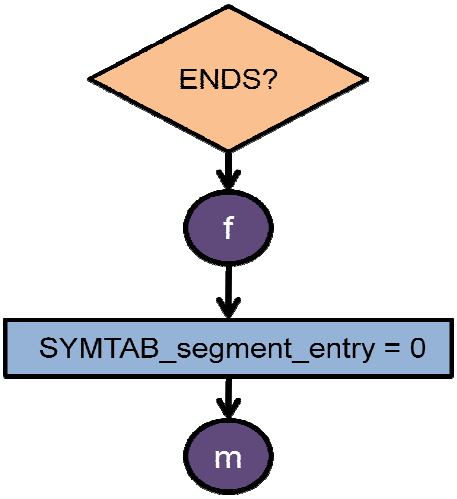


Figure 30f: Flowchart for processing the END statement of assembly program

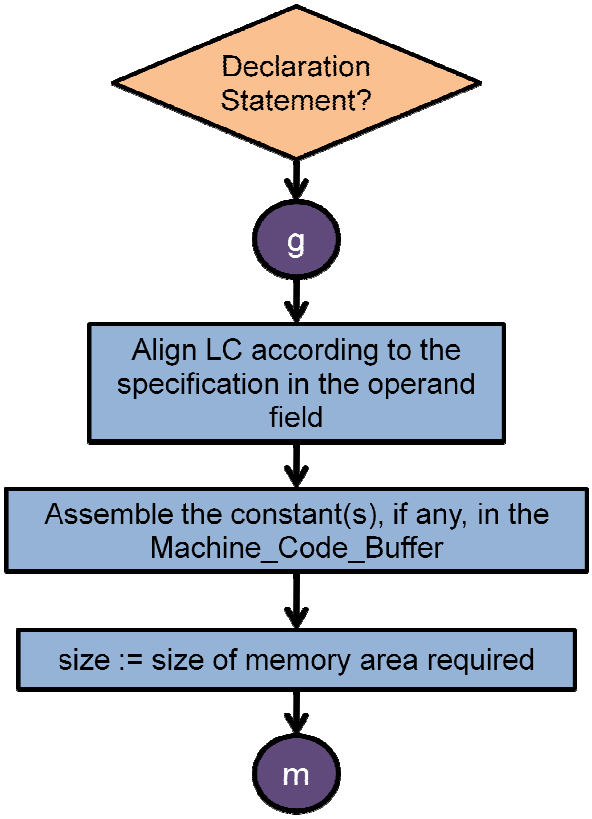


Figure 30g: Flowchart for processing the declaration statement of assembly program

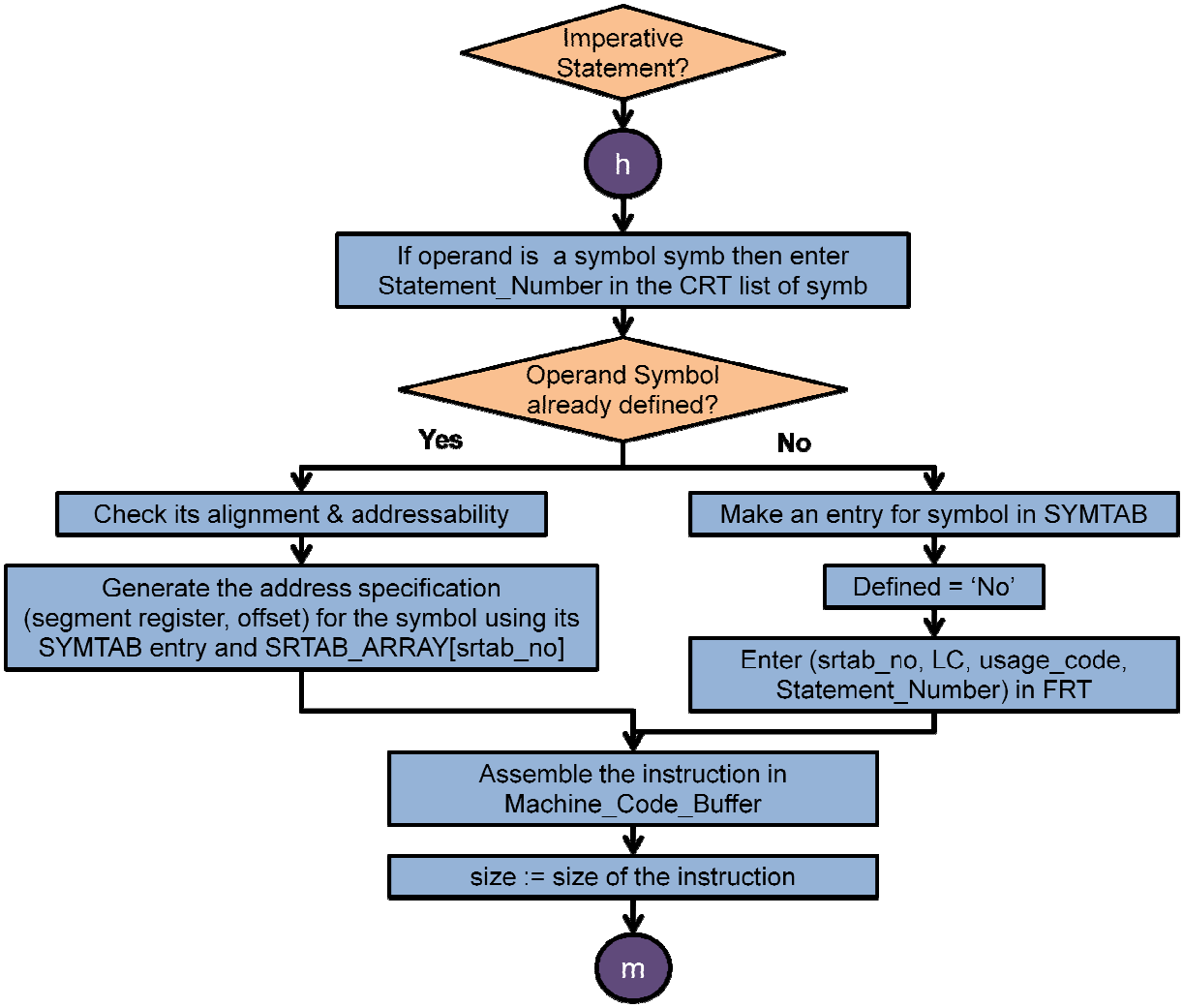


Figure 30h: Flowchart for processing the imperative statement of assembly program

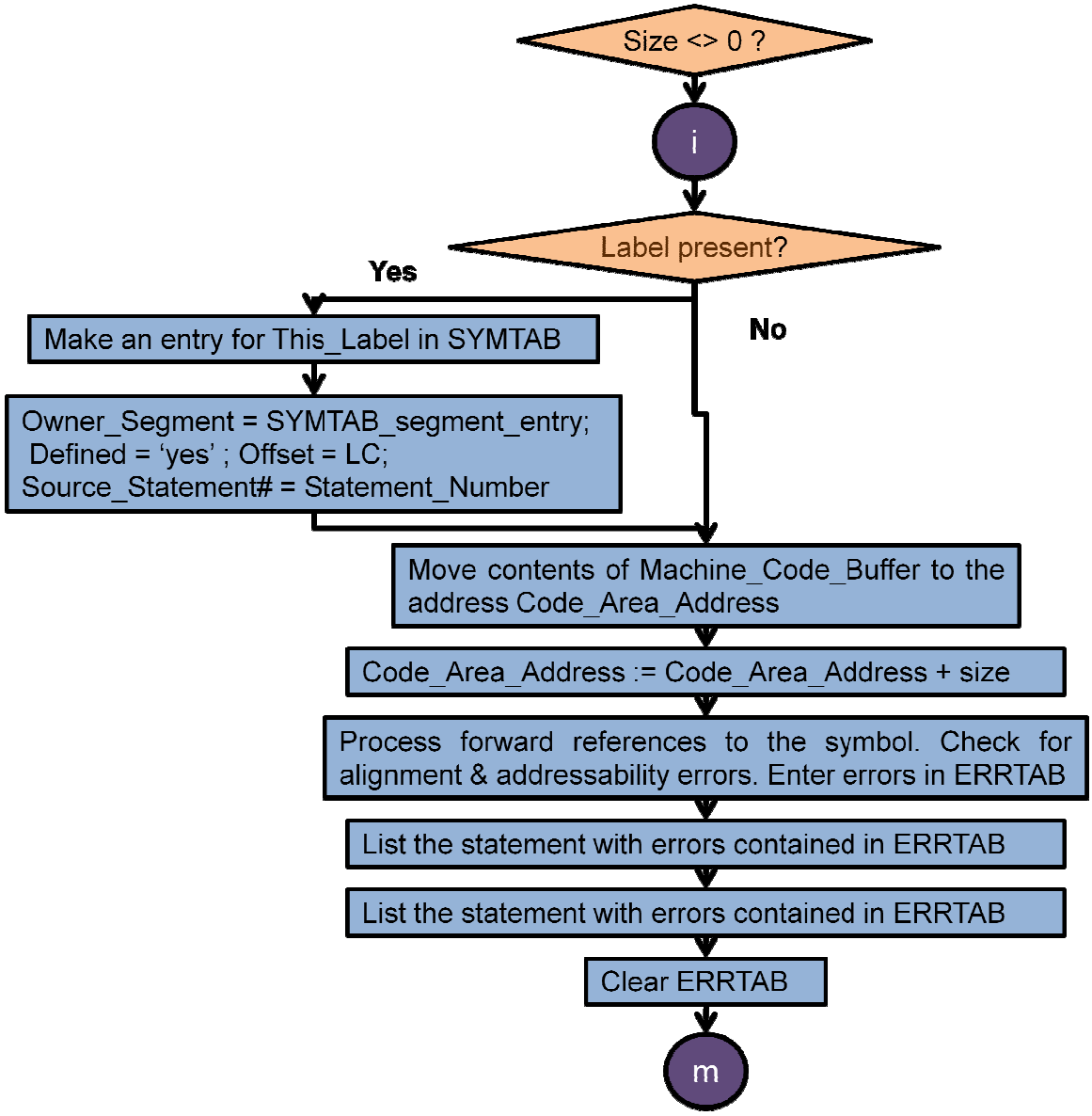


Figure 30i : Flowchart for processing the statement having size nonzero of assembly program

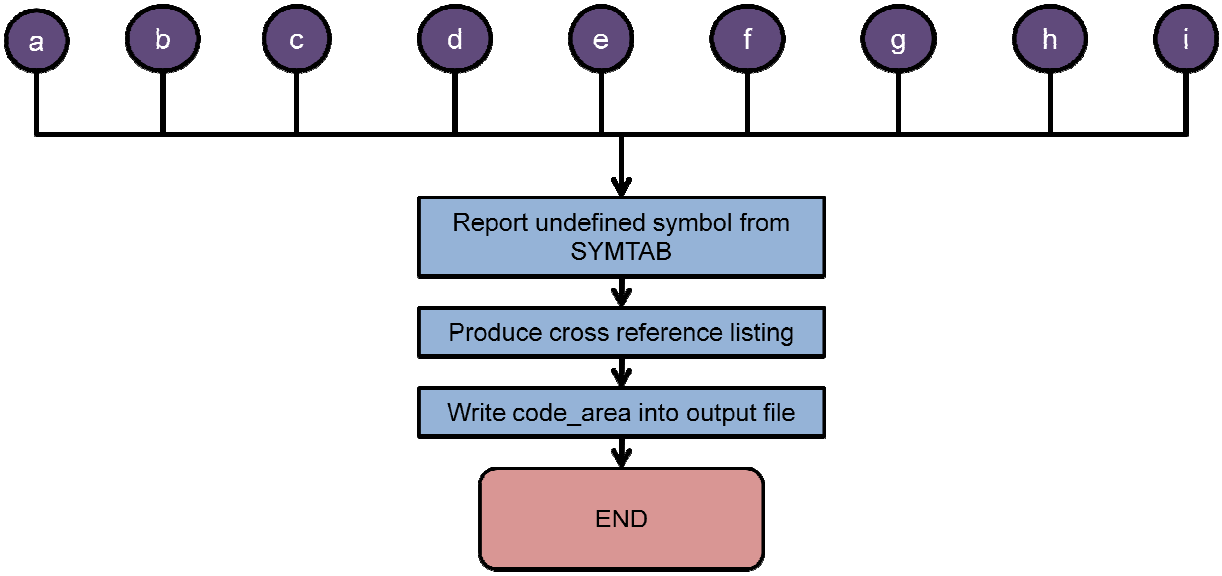


Figure 31: Flowchart for processing the END statement of assembly program