BHCS15B: System Programming

Assembler

Mahesh Kumar
(maheshkumar@andc.du.ac.in)

Course Web Page
(www.mkbhandari.com/mkwiki)

Outline

1. Introduction to System Software
2. Introduction to Assembler
3. A Simple Manual Assembler
4. Assembler Design Process
    1. *Major Data Structures Used.*
    2. *Two-Pass Assembler*
    3. *Single-Pass Assembler*
5. Load-and-go Assembler
6. Object File Formats [self study]

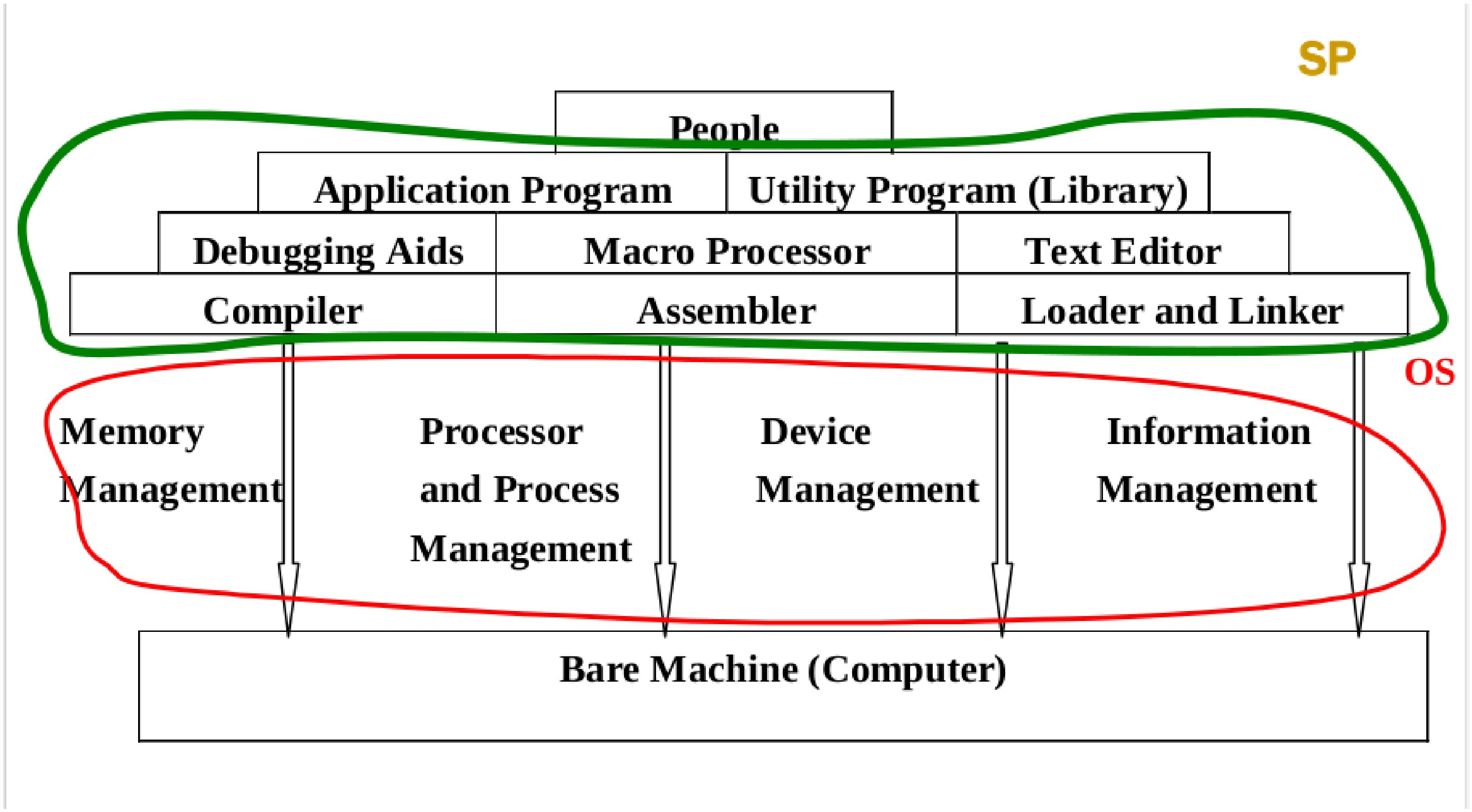
Introduction

- *System Software*:

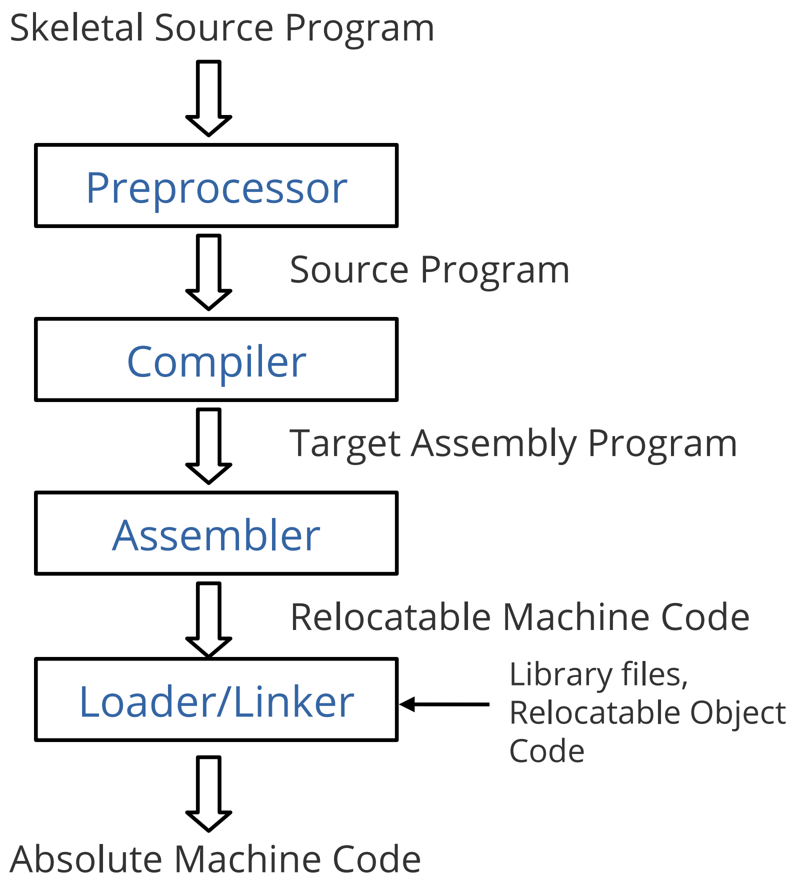
  - Consists of a variety of programs that support the operation of a computer.

  - The software makes it possible for the user to focus on an application without needing to know the details of how the machine works internally.

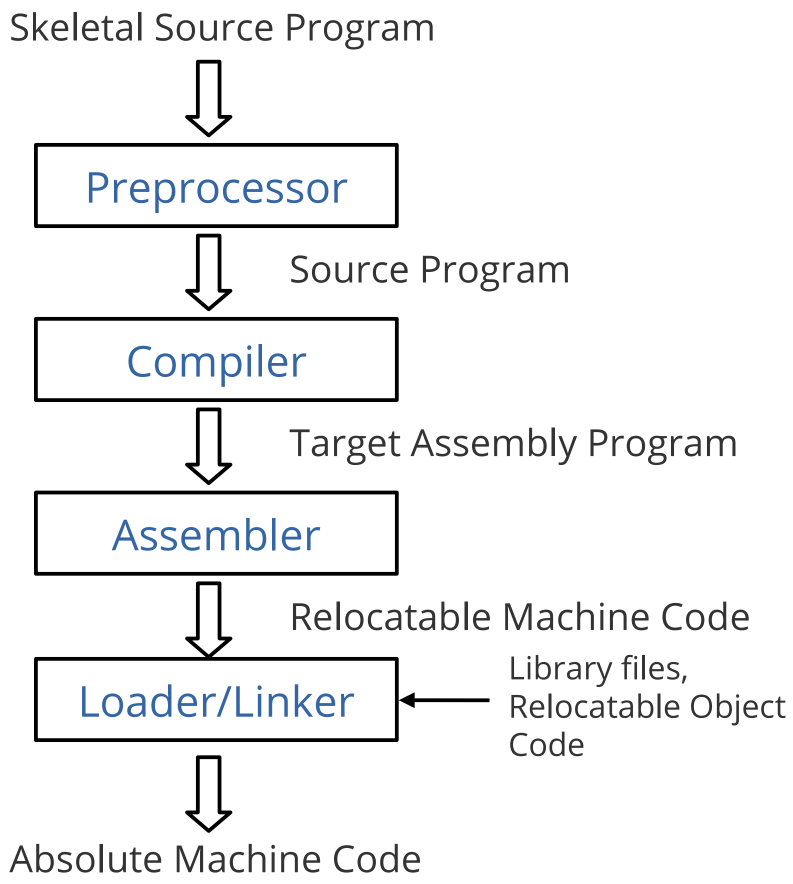
Introduction (2)



Introduction (3)



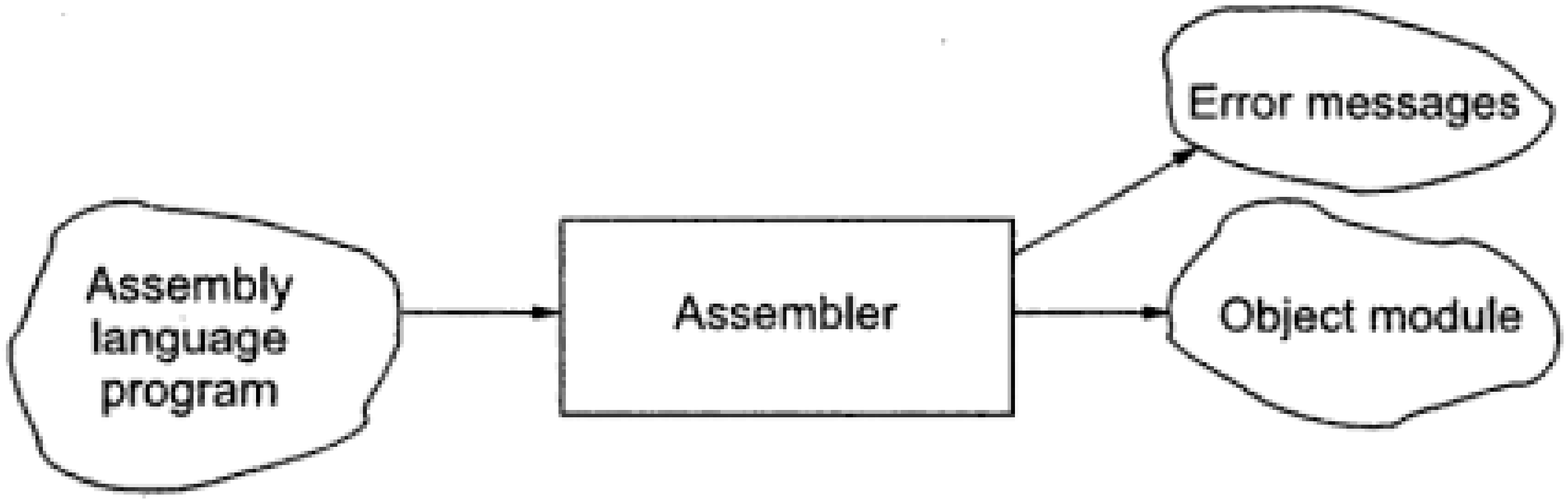
Introduction (4)



- *Different types of System Software*:
  - ***Text editor:*** create and modify the programs
  - ***Compiler:*** translate programs into machine language
  - ***Linker:*** performs the linking task
  - ***Loader:*** load machine language program into memory and prepares for execution
  - ***Assembler:*** translate assembly program into machine language
  - ***Macro processor:*** translate macros instructions into its definition
  - ***Debugger:*** detect errors in the program
  - ***OS:*** *You* control all the above by interacting with the operating system.

Introduction (5)

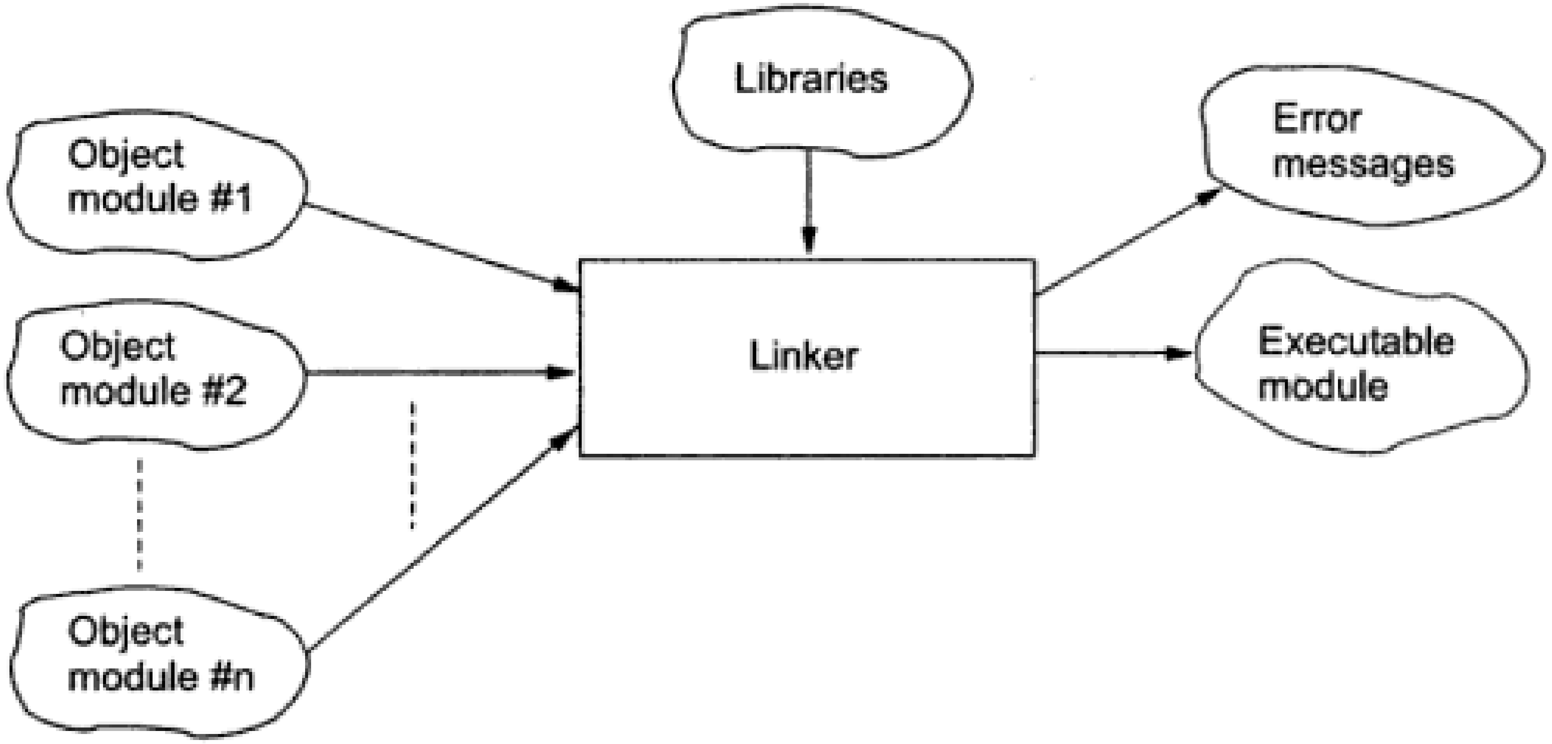
- *Input-Output of an Assembler*



[Source 3]

Introduction (6)

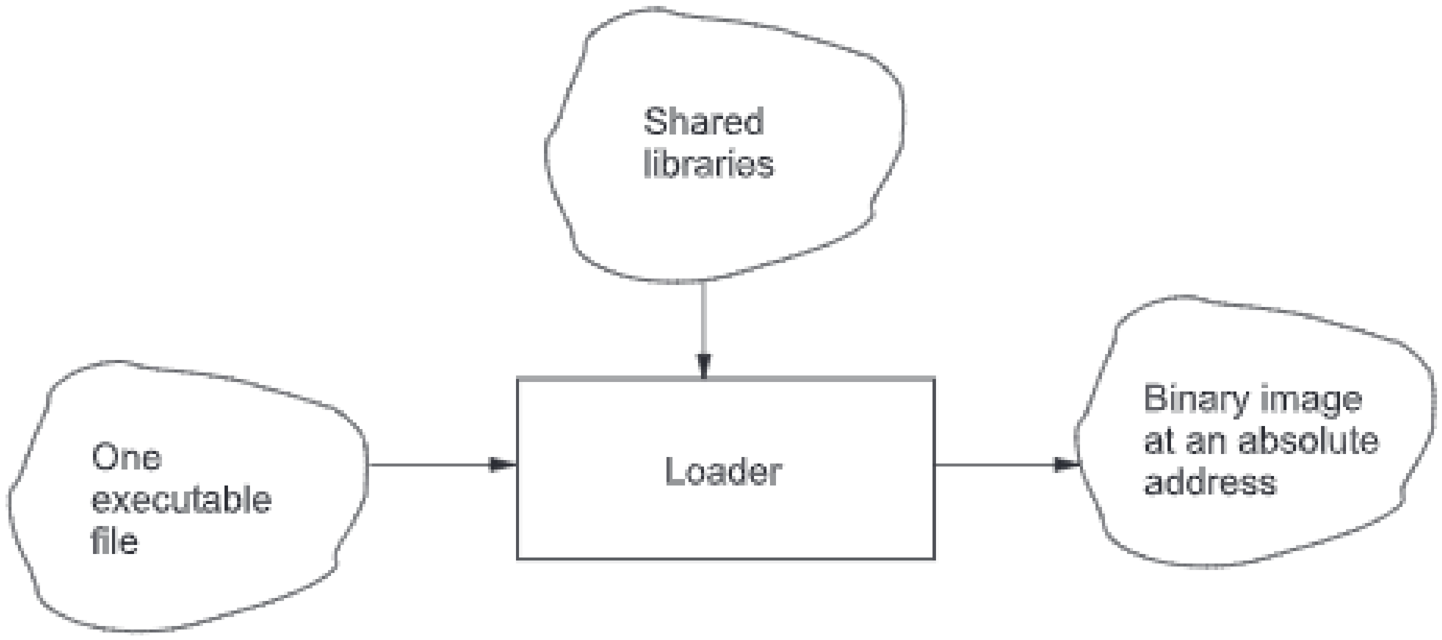
- *Input-Output of an Linker*



[Source 3]

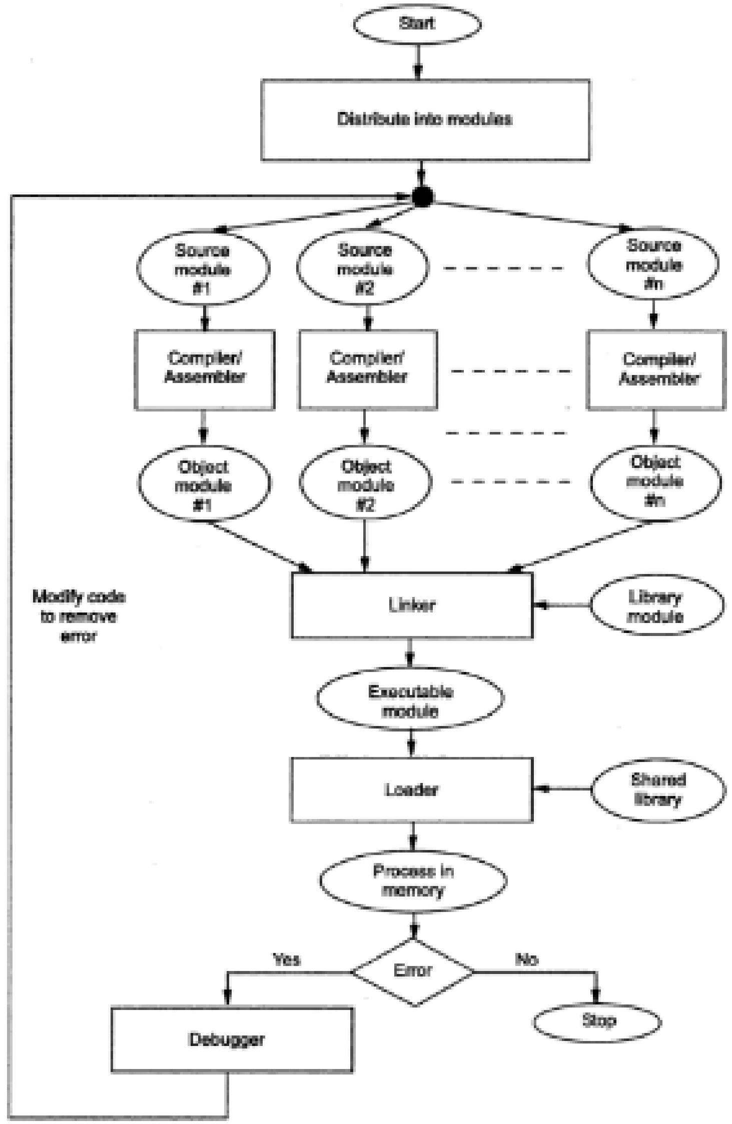
Introduction (7)

- *Input-Output of an Loader*



[Source 3]

- *Program Development Flow ->*



[Source 2]

System Software and Machine Architecture

- One characteristic in which most system software differ from application software is *machine dependency.*
  - System programs are intended to support the operation and use of the computer itself, rather than any particular application.
  - Application programs are primary concerned with the solution of some problem, using the computer as a tool.
- *Example:*
  - Assembler translates mnemonic instructions into machine code.
    - *instruction formats, addressing modes, etc..*
  - Compilers must generate machine language code.
    - *number and type of registers, machine instructions available, etc..*
  - OS is directly concerned with the management of nearly all of the resources of a computing system.

System Software and Machine Architecture (2)

- Important machine structures to the design of system software:
  - Memory Structure
  - Registers
  - Data Formats
  - Instruction Formats
  - Addressing Modes
  - Instruction set

System Software and Machine Architecture (3)

- *Some aspects of system software* that do not directly depend upon the machine architecture are:
  - The general design and logic of an *assembler* is basically the same on most computers.
  - Some of the code optimization techniques used by *compilers* are independent of the target machine.
  - The process of linking together independently assembled subprograms do not usually depend on the computer being used.

System Software and Machine Architecture (4)

- *While understanding any system software, we should identify:*
  - Features that are fundamental
  - Features that are architecture dependent
  - Extended features that are relatively machine independent
  - Major design options for structuring the software
  - Optional features

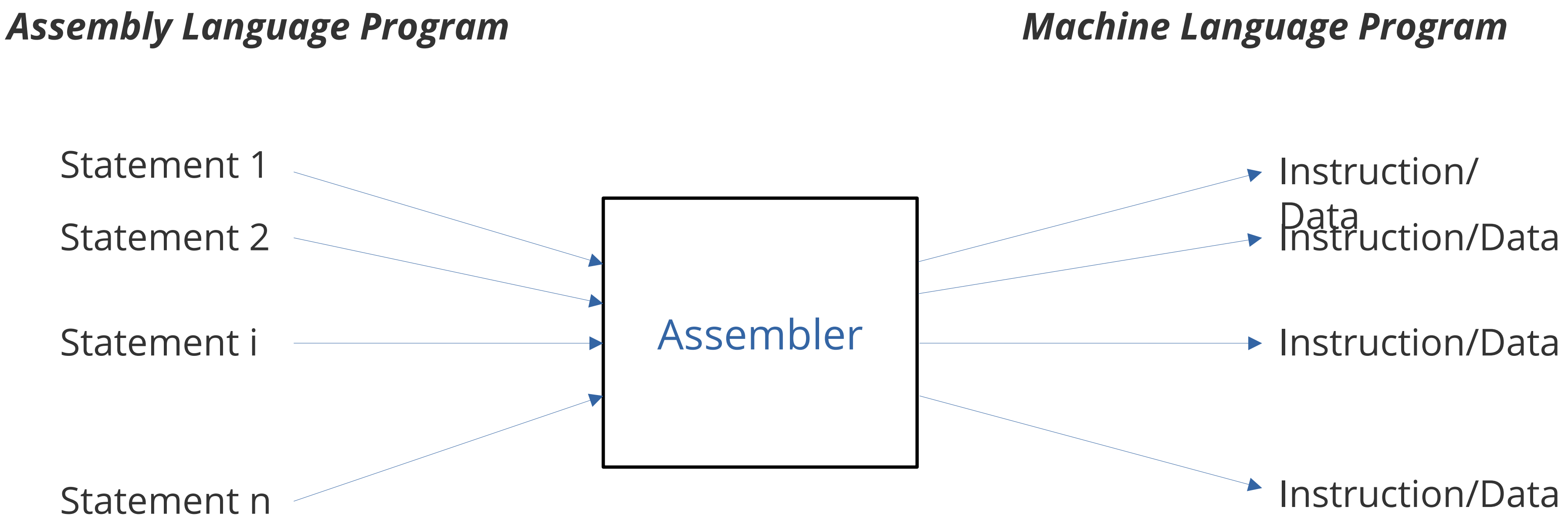
Introduction to Assembler

- Assembler is the tool (*translator/language processor*) to convert *assembly language* into the *machine language* one:
  - *Understandable by the processor that executes it.*

- The complexity of the process depends upon various factors:
  - Size of the *instruction set*
  - Different *addressing modes* supported
  - Length of the *program being translated*, and so on.

- For the simplest case, the assembly may be done manually (*hand assembly*) however, for the complex processors the automation is a must.

The role of an Assembler



The role of an Assembler (2)

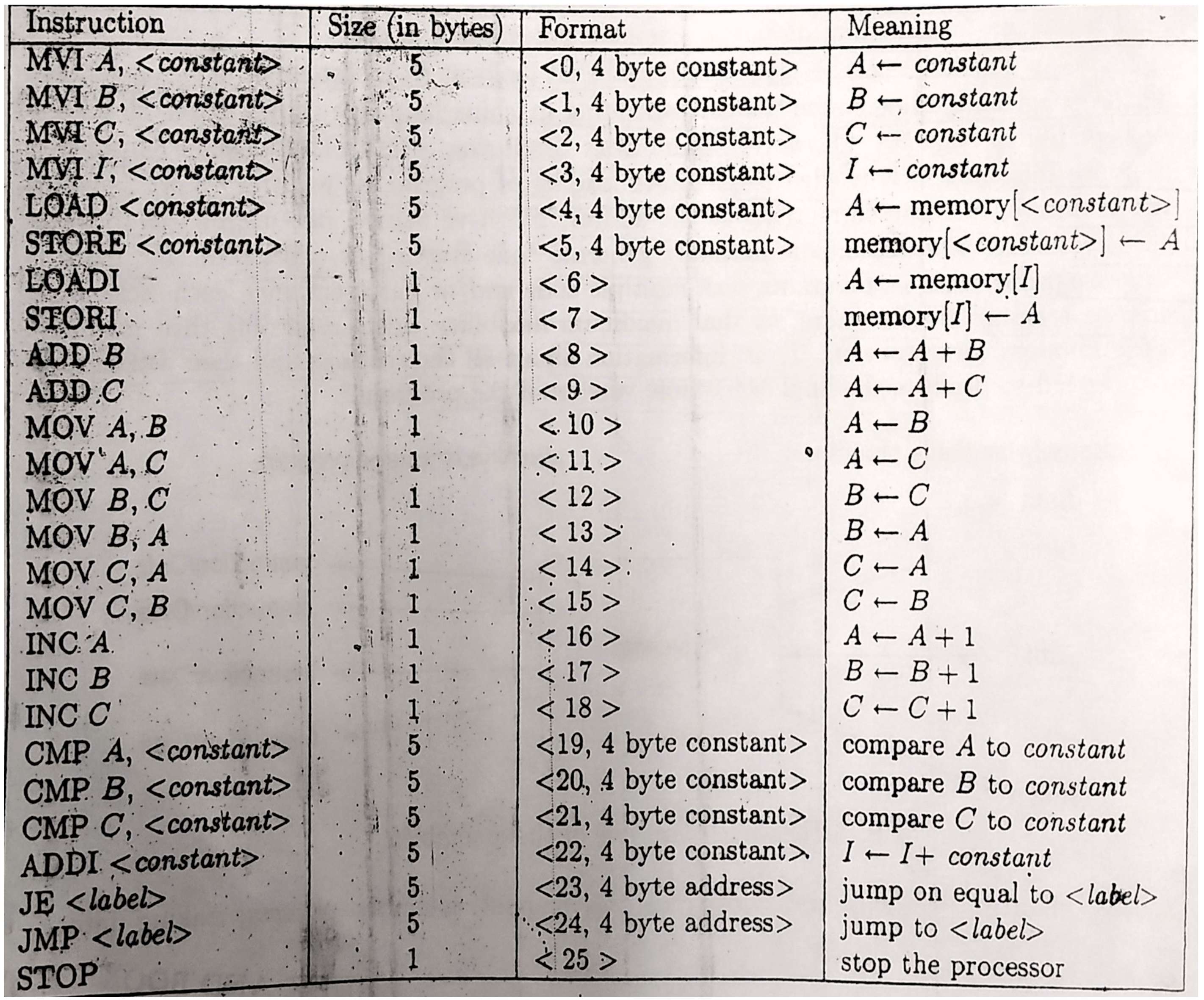
- Apart from producing the machine language code, some more information is needed to facilitate loading of program at arbitrary start address in the memory.
- It needs corrections to the address sensitive values like:
  - *operand addresses*
  - *jump targets, etc.*
- The assemblers normally generate code starting at offset *zero.*
- For multi-section programs, each section is assembled starting at offset zero, so that maximum flexibility exists regarding their loading into the memory for execution.
- Information about all the sections and their attributes is also needed to produce the final executable version of the program.

A Simple Manual Assembler

- ***A simple hypothetical accumulator based processor.***
  - ***Accumulator register (A)***: all memory *load* and *store* operations.
  - ***Arithmetic and logic operations***: mostly use **A** as source register and also the destination register.
  - ***B and C registers***: 32-bit *general purpose* registers.
  - ***Indexed registers (I)***: Arithmetic is permitted on **I** also.
  - ***Addressing modes***: *immediate* and *indirect* through the indexed register.
  - All ***memory addresses*** are 32-bit wide.
  - ***Register-to-register*** *data movement* is also supported.

- *The instructions and the associated codes shown →*

  The machine instructions,is called as ***Machine Opcode Table*** *(****MOT****) 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

A Simple Manual Assembler (2)

- Our Assembler supports only one *pseudo-opcode* ***dd*** to reserve and initialize four bytes memory locations.

- In a general assembler there may be a number of pseudo-opcodes supported, stored in a table known as ***Pseudo Opcode Table*** (POT).

- Let us look at a simple program written in this language to sum ten numbers stored in the memory. -->>

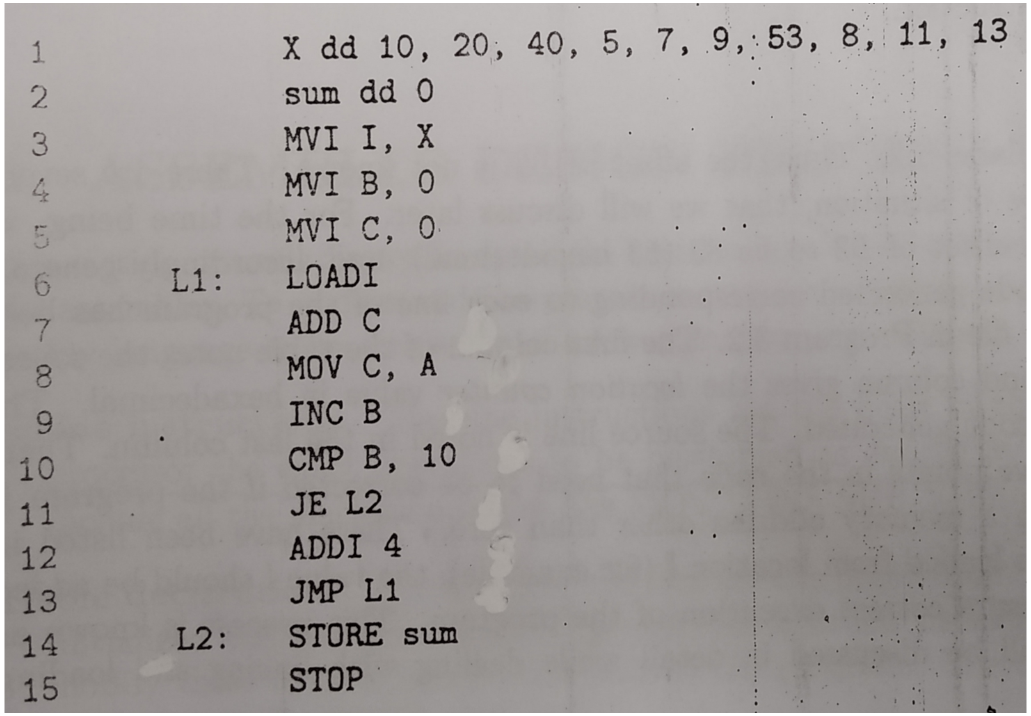
- ***Symbol Table:***

  - contains all variables and labels defined in the program.

  - helps in the assembly process and is also is needed by linker and loader.

A Simple Manual Assembler (3)

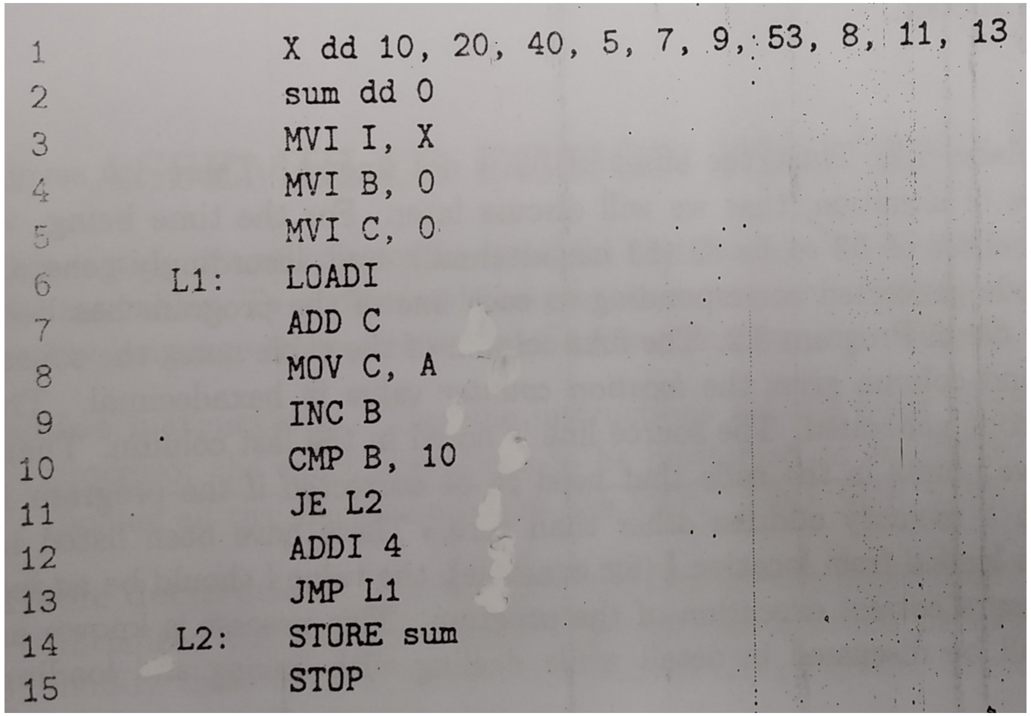
- A program to add 10 numbers



```
        X dd 10, 20, 40, 5, 7, 9, 53, 8, 11, 13
        sum dd 0
        MVI I, X
        MVI B, 0
        MVI C, 0
L1:     LOADI
        ADD C
        MOV C, A
        INC B
        CMP B, 10
        JE L2
        ADDI 4
        JMP L1
L2:     STORE sum
        STOP
```

A Simple Manual Assembler (4)

- A program to add 10 numbers



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

- The assembly process start at *offset zero*.
- *Location Counter*: a variable to keep the track of next offset in which the generated code is to be placed, is also initialized to zero.
- *Line 1*: a declaration variable ***X*** having 10, 32-bit numbers.

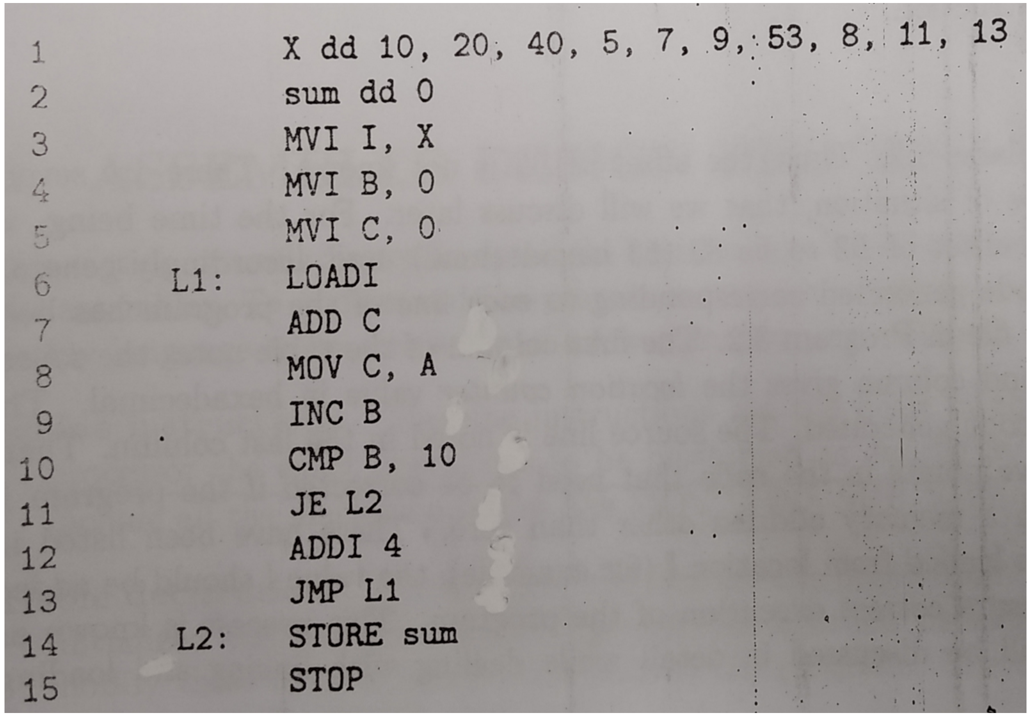
  Assembler reserves space and initializes accordingly ***(dd)***.

  Location counter is incremented to 40.

  The variable ***X*** along with its attributes is entered into the *symbol table.*

A Simple Manual Assembler (5)

- A program to add 10 numbers



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

- *Line 2:* The next four locations are reserved to store ***sum***.

  Assembler reserves space and initializes it to zero.

  Location counter is incremented to 44.

  The variable ***sum*** is also entered into the *symbol table.*

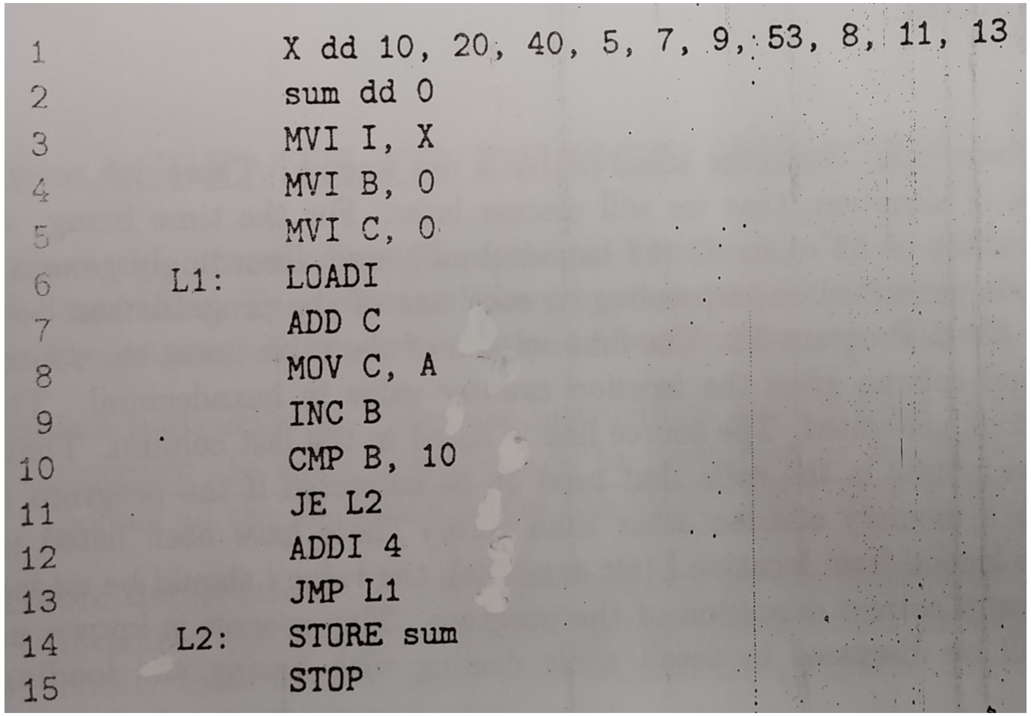
- *Line 3:* This instruction is assembled into five bytes

  The location 44 contains the ***opcode 3***, and location 45-48 holds the ***offset of X***, which is zero in this case

  The location counter is updated accordingly.

A Simple Manual Assembler (6)

- A program to add 10 numbers



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

- *Line 6:* The ***label L1*** is entered into symbol table along with its offset from the beginning of the program.

  When we assemble the instruction ***JMP L1***, we already know the *offset of **L1** and the code can be generated accordingly.*

- *Line 11:* When we try to assemble the instruction ***JE L2,*** we have not seen the ***L2***. Thus, the offset of ***L2*** is not known.

  There are several ways of handling this type of situation.

  For the time being, we assume the offset of L2 to be 83(53 hex.), and accordingly generate the machine code.

- The *list file* of the program to add 10 numbers.

```
1    00000000   0A000000 14000000            X  dd  10,  20,  40,  5,  7,  9,  53,  8,  11,  13
                28000000 05000000
                07000000 09000000
                35000000 08000000
                0B000000 0D000000
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         L2:       STORE  sum
15   00000058   1900000000                   STOP
```

A Simple Manual Assembler (7)

- The *list file* table consists of four columns:
  1. The first column of the table notes the *source line number*.
  2. The second column gives the *location counter value in hexadecimal*.
  3. The third column contains the *code generated.*
  4. 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. (Program Relocation)

A Simple Manual Assembler (8)

- ***The symbol table of the program***

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

A Simple Manual Assembler (9)

- ***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

- ***Program Relocation:*** 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. (will be covered in detail in Linking and loading techniques)*

Assembler Design Process

- There are various design issues and techniques involved.
- To understand the process better, we can think of program to consisting of a few components as follows:

1. ***Machine instructions:*** tells what to do, includes all the machine opcode like: MOV, ADD, SUB, JMP, and so on.
2. ***Variable declarations:*** the declarations of the storage space, machine instructions may modify these data.
3. ***Assembler directives:*** to produce the code in structured manner so that different sections of the program can be handled efficiently and perhaps independently.
4. ***Comments:*** used by programmer for documentation purposes. For example:

```
ADD  B        ; Adds contents of B with A and stores sum in A.
```

Major Data Structures Used

- The major tables involved with the assembly process are:
  1. *Machine Opcode Table* (MOT)
  2. *Pseudo Opcode Table* (POT)
  3. *Symbol Table* (SYMTAB)

- *Machine Opcode Table* (MOT) and *Pseudo Opcode Table* (POT) are static in nature.

- On the other hand contents of *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

- It hold the opcodes used by the processor for different instruction mnemonics. Typical entries of this table are:

Mnemonic	*Size*	*Opcode*

  - *Mnemonics* field contains various instruction mnemonics.
  - *Size* field contains the size of the instructions (*typically in bytes*).
  - *Opcode* field contains the machine code corresponding to the mnemonics.

- The basic structure of MOT may be varied significantly based upon *instruction set* and also on the *designer of the assembler*.

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 dictate the organization to be used.

- Since MOT is static, *insertions, deletions, and updates are not there*. Only *search* operation is there.

- 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 <u>*search* operation can be carried out very fast</u>

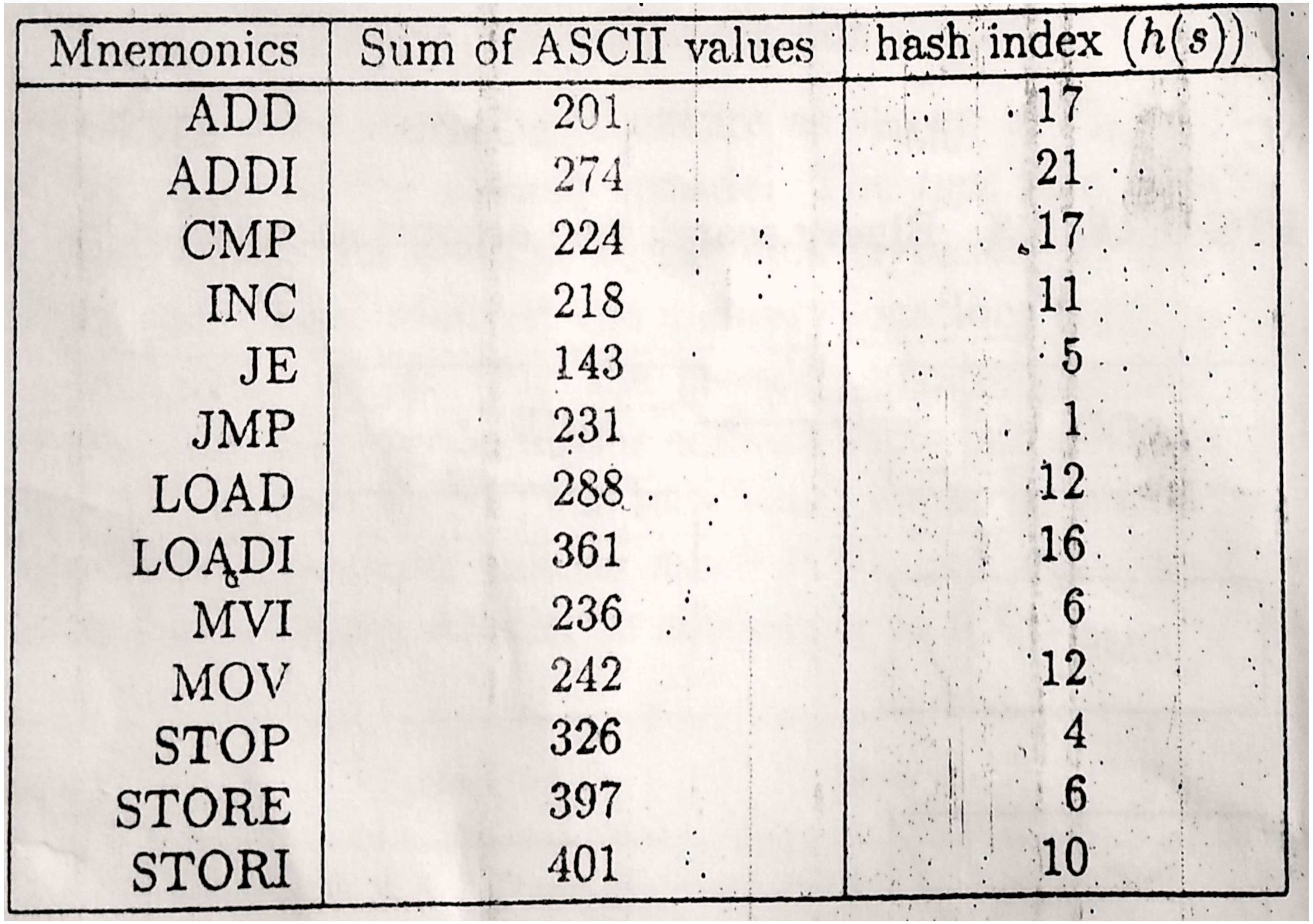
- A very good data structure providing *constant time* O(1) searching algorithm is the hash table.

Organization of MOT (2)

- A suitable choice of *hash function* has to be made to convert a mnemonic into an integer values to be used as index to the table.
- A basic problem with hash table is the *collision* of data items (*two or more mnemonics mapping to the same location in the MOT*)
- Good collision resolution strategies are needed to take care of this situation
- The calculation of hash Indexes for a set of mnemonics using hash function *h(s)* (where s is the mnemonic) given by:

h(s)=(Sum of ASCII values of the characters of s) mod 23

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

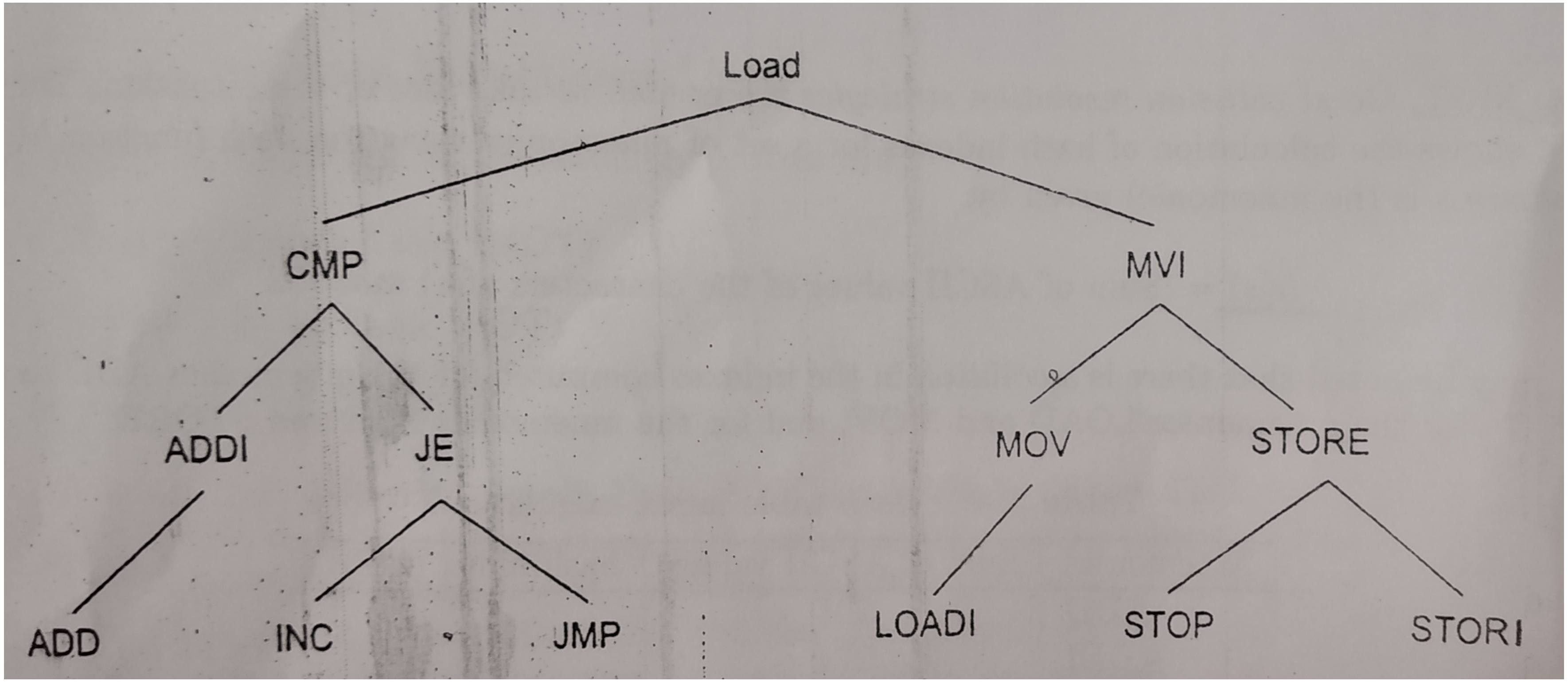
Organization of MOT (3)

- Some other alternative data structures can be used:

  1. *Binary Search Tree* (BST)
     - *Mnemonics* are sorted in the alphabetical order and then inserted into a BST 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).
  2. *Linked List* (LL) : indexed by the first alphabet of the mnemonic.
     - The array is indexed by the first letter of the mnemonic.

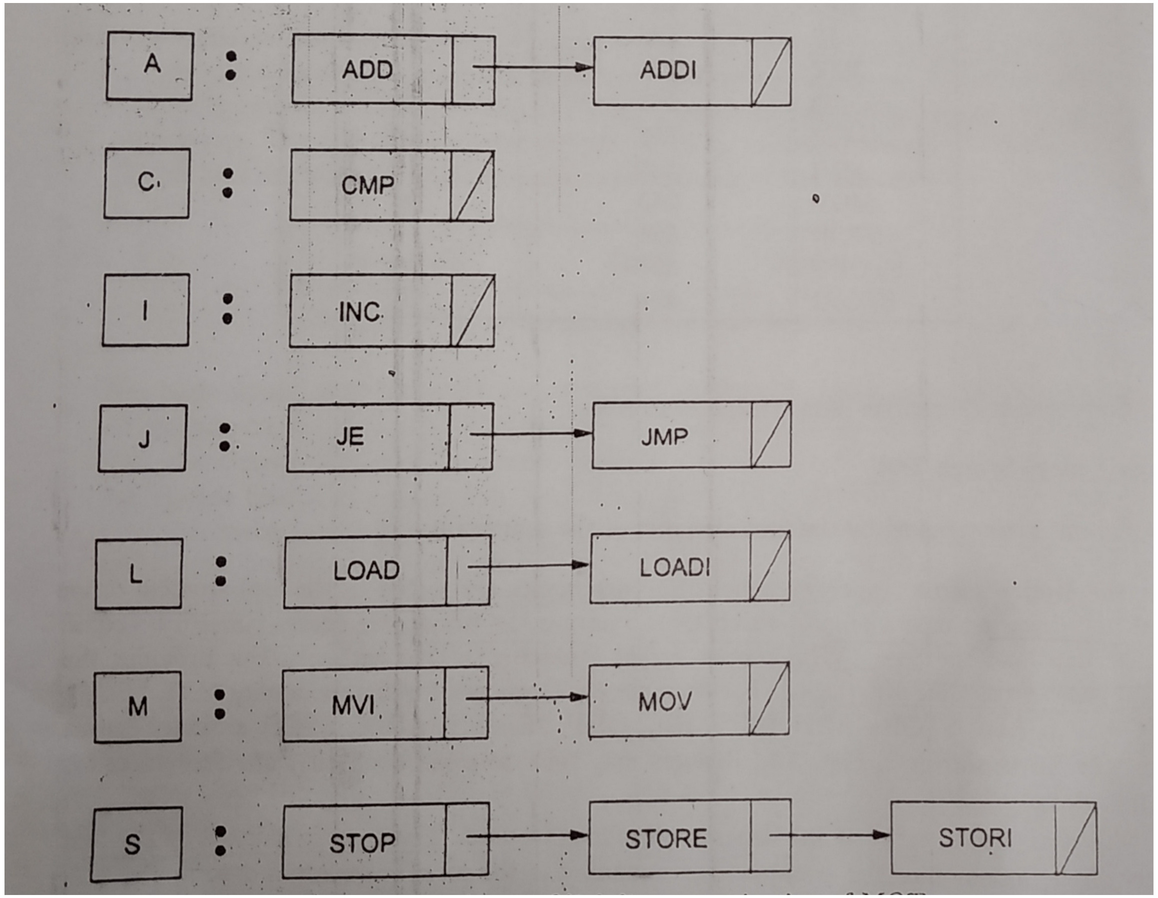
Binary Search Tree organization of MOT

- For example processor with mnemonics MVI, LOAD, STORE, LOADI, STORI, ADD, MOV, INC, CMP, JE, JMP, ADDI, STOP, a BST may be as shown:



Array of Linked List organization of MOT

- The array is indexed by the first letter of the mnemonic:



Pseudo Opcode Table

- It contains the pseudo opcodes supported by the assembler, used to reserve memory space and possibly initialize it.
- There can be several pseudo opcodes as imagined by the assembler designer.
- ***NASM*** pseudo opcodes are as shown: ->

DB :	define byte
DW :	define word
DD :	define double-word / float
DQ :	define double-precision float
DT :	define extended-precision float
RESB :	reserve byte
RESW :	reserve word
RESD :	reserve double-word / float
RESQ :	reserve double-precision float
REST :	reserve extended-precision float

Pseudo Opcode Table (2)

- A typical POT may have the structure as shown below:

Pseudo-opcode	*Type*	*Size*	*Initializable*

- *Pseudo-opcode* field contains the name of the pseudo-opcode.
- *Type* field identifies type of data.
- *Size* field holds the size of the field (*typically in bytes*).
- *Initializable* field is boolean, specifying whether the memory locations corresponding to the pseudo opcode can have initial values.

  - *DB, DW, DD, DQ, and DT, the corresponding locations contain some initial values.*

  - *RESB, RESW, RESD, RESQ, and REST, the locations cannot be initialized.*

Symbol Table (SYMTAB)

- An essential data structure used by the assembler to remember information about *identifiers* appearing in the program to be assembled.

- The type of symbols that are stored in symbol table are: *variables, procedures, defined constants, labels*, etc.

- Symbol table is designed by the assembler writer to facilitate the assembly process.

- The identifiers stored in the symbol table may vary widely from one assembler to another, even for the same assembly language.

Information in Symbol Table

- For an identifier stored in the symbol table, following are the associated information stored:
  - *Name:* The name of the identifier, may be stored *directly in the table* or *the table entry may point to another character string* (in an associated string table) .
  - *Type:* The type of the identifier, it defines whether it is a variable, a label, a procedure name, and so on. Variables are further identified by type like *byte, word, double word*, etc.
  - *Locaton:* This is an offset within the program where the identifier is defined.

Fundamental operations and Data Structures on SYMTAB

- The fundamental operations on SYMTAB are:
  - *Enter* a new symbol in the table.
  - *Lookup* for a symbol.
  - *Modify* information about a symbol stored earlier in the table.
- To create a SYMTAB several data structures exist, the commonly used are:
  - *Linear Table*
  - *Ordered list*
  - *Tree*
  - *Hash table*

Linear Table

- 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: Consider the following definitions:

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

The **linear Table** will store the information as shown:

Symbo 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

Others: Ordered List, Tree, Hash Table

- Ordered List: a variation of linear tables in which a list organization is used.
  - Sort the list, then use Binary Search to access table in O(log n) time.
  - Insertion becomes costly.
- Tree organization of the SYMTAB may be used <u>to speed up the access</u>.
  - Each entry is represented by a node of the tree.
  - Based on string comparison(*less-than / greater-than*) of names with reference node, entries are put on appropriate left/right subtree.
  - Average lookup time for an entry in the tree is O(log n); however if tree becomes unbalanced height, will require O(n) search time. (*AVL trees for height balancing*)

Others: Ordered List, Tree, Hash Table (2)

- Hash Table: provides the fastest mechanism to access the symbols.
  - Similar to those for MOT, except it is dynamic in nature.
  - Designing good hash function and good collision resolution strategies are needed.

Two-Pass Assembler

- The two-pass assembly process scans the input assembly language program twice, called as *Pass I* and *Pass II.*

- *Pass I*: different data structures (*tables*) are filled up with the information related to the symbols and sections defined in the program.

- *Pass II:* generates the actual code.

- Both the passes uses a variable, *location counter* (***lc***) to computer current offset from the beginning of a *section* while traversing the input file

Two-Pass Assembler – *Pass I*

- The main task of this pass is to scan the input file and compute the offset of all symbols appearing in the program, in the following tables:

1. ***Section table:*** Detailed information regarding *all sections* appearing in the input program.

Name	*Size*	*Attributes*	*Pointer to content*

- *Section name:* the name of the section
- *Size:* the total size
- *Attributes:* the section declaration may have several attributes like *name, start address, size, align*, etc.
- *Pointer to content:* at the end of *Pass II,* pointer will point to the content of the section after translation.

Two-Pass Assembler – *Pass I*

2 ***Symbol table:*** holds information about the *symbols* defined in the program.

Name	*Type*	*Location*	*Size*	*Section-id*	*Is-global*

- *Name:* the name of the symbol
- *Type:* stores the type such as *variable, label*, etc.
- *Location:* the offset of the symbol from the start of the section containing it.
- *Size:* stores the size of symbol in bytes
- *Section -id:* identifies the section to which the section belongs.
- *Is-global:* is a Boolean, identifying whether the symbol has been declared as global.

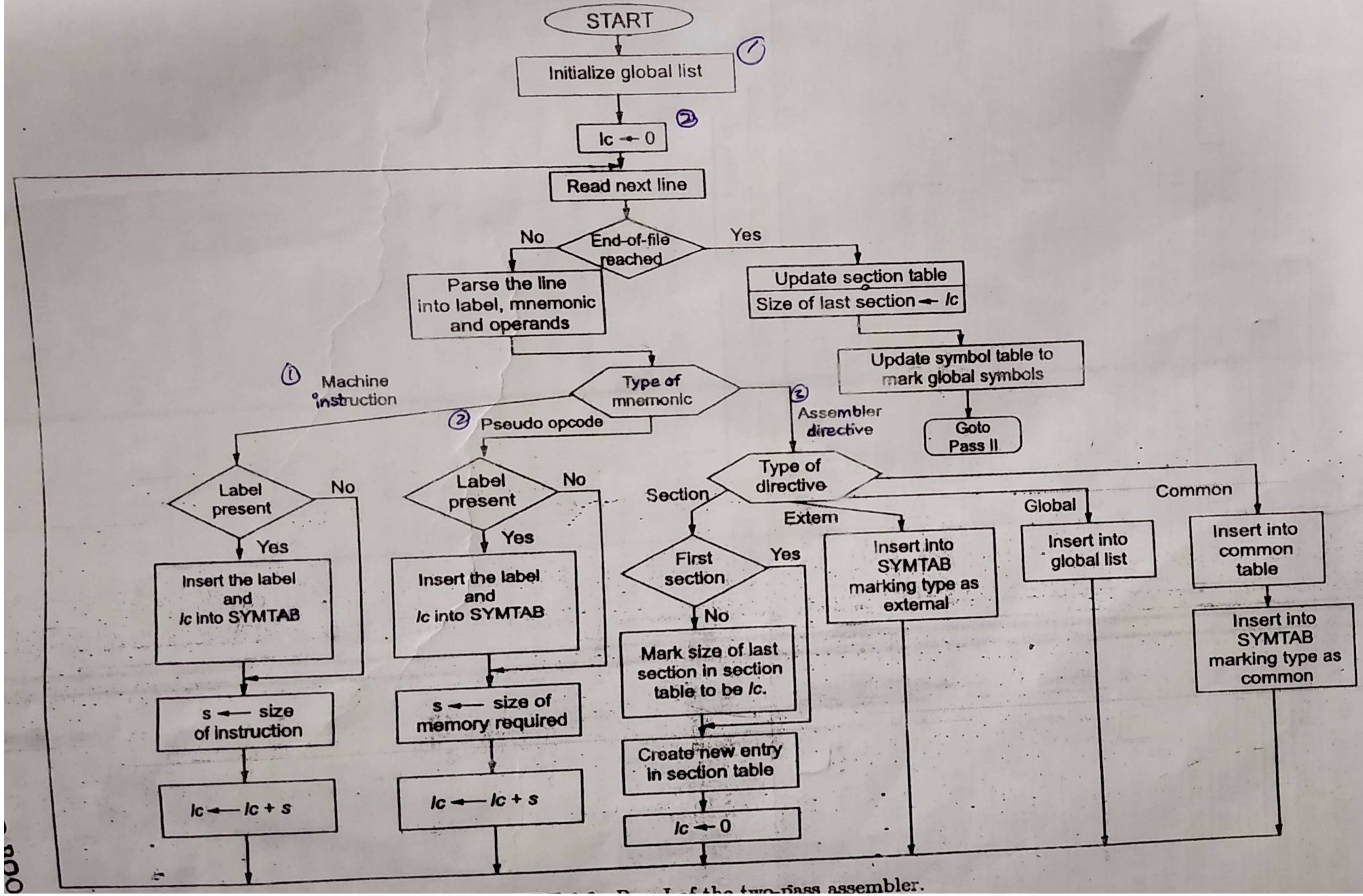
Two-Pass Assembler – *Pass I*

3 ***Common table:*** holds information about the variables declared *COMMON*.

Name	*Size*

Two-Pass Assembler – *Pass I* (Flowchart)

- *Please see the flowchart: Pass I* of the two-pass assembler


START
Initialize global list
1
Ic ← 0
2
Read next line
End-of-file reached
No
Yes
Update section table
Size of last section ← Ic
Update symbol table to mark global symbols
Goto Pass II
Parse the line into label, mnemonic and operands
Type of mnemonic
1 Machine instruction
2 Pseudo opcode
3 Assembler directive
Label present
No
Yes
Insert the label and Ic into SYMTAB
s ← size of instruction
Ic ← Ic + s
Label present
No
Yes
Insert the label and Ic into SYMTAB
s ← size of memory required
Ic ← Ic + s
Type of directive
Section
Extern
Global
Common
First section
Yes
No
Mark size of last section in section table to be Ic.
Create new entry in section table
Ic ← 0
Insert into SYMTAB marking type as external
Insert into global list
Insert into common table
Insert into SYMTAB marking type as common

... of the two-pass assembler.

Two-Pass Assembler – *Pass I* (Flowchart explained..)

1. Initialize the *Global list,* to store all *global* declarations. The *lc* is also initialized to zero.

2. Read the next line and update one or more tables as needed, till the *end-of-file* marker is reached by *physical end* or by *some special symbol*.

3. The process *Read next line* scans the input file and returns the line to be assembled next skipping over the comments.

4. The *line* is then *parsed* into components like possible *label, mnemonic, and operands.* The *mnemonic* can be a:
    - *machine instruction*
    - *pseudo opcode*
    - *assembler directive.*

Two-Pass Assembler – *Pass I* (Flowchart explained..)

5. If a *label* is present in the line:
   - Put *label* and *current lc* value in Symbol table.
   - *Type* is set as *label* for machine instructions and *variable* for pseudo opcode.
   - *Size* is not requreid for *label,* for a *variable* it is set to the amount of memory required to store the variable.
   - Section-id is set to the index of current section in *section table*.
   - Is-global is set to *false* for the time being.
   - Next, *lc* is updated by the size of the instruction for machine instructions and the amount of memory required for pseudo opcode.

Two-Pass Assembler – *Pass I* (Flowchart explained..)

6 For the *assembler directive* 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 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.

- For an *external* symbol defined via an *extern* declaration, it is put into *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 also put into the *Symbol table,* marking its type as *common.*

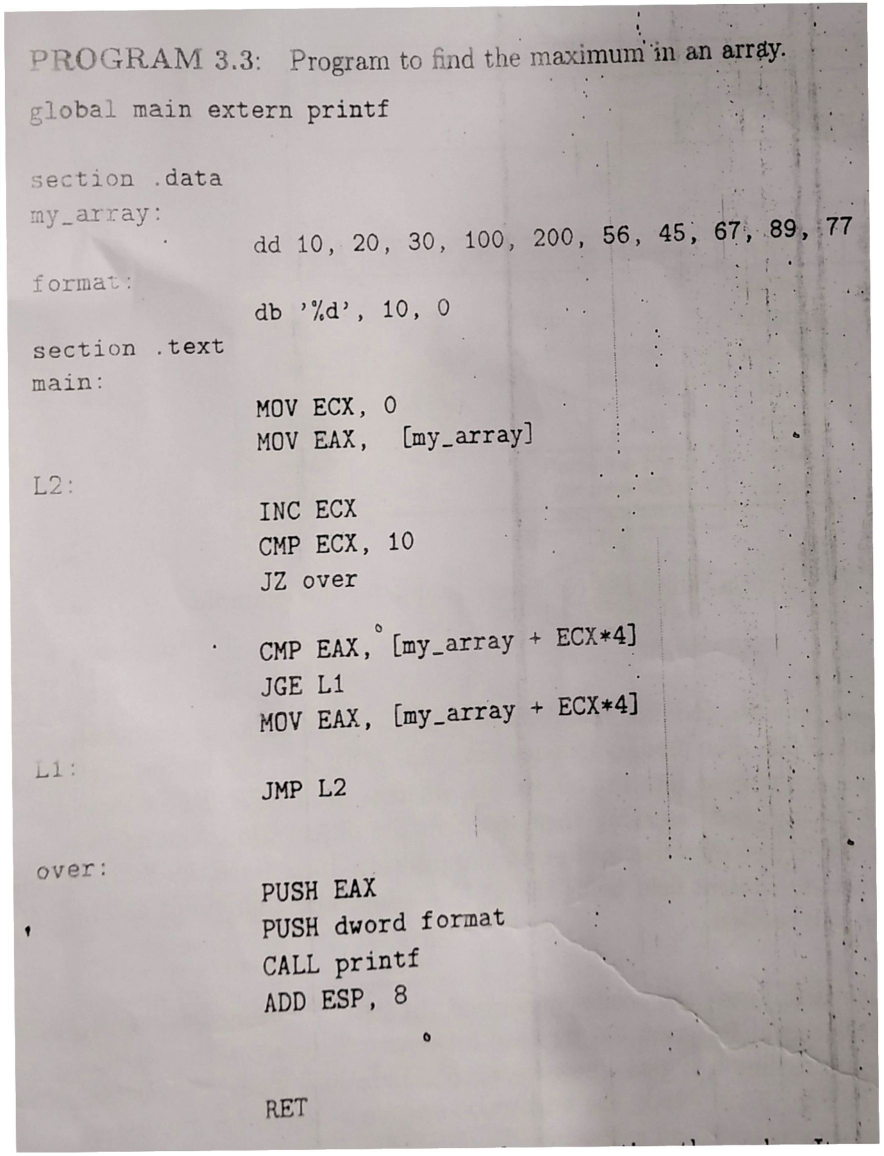
Two-Pass Assembler – *Pass I* (Flowchart explained..)

7 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 .***

Two-Pass Assembler – *Pass I* (Example)

- *Please see the Program:* to find the maximum in an array.



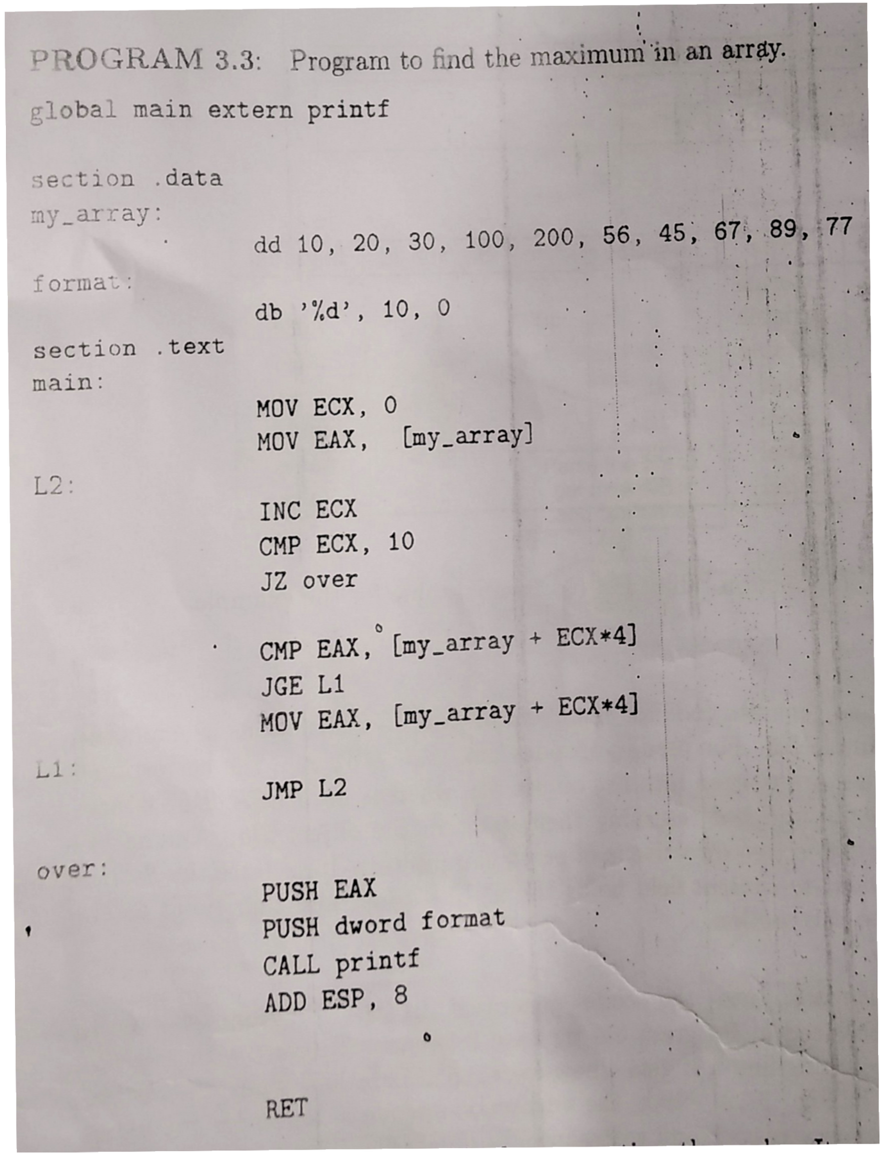
PROGRAM 3.3: Program to find the maximum in an array.

```
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
```

- To start with, *Global list* is initialized and *lc* is set to zero.
- Line 1: global main, puts the entry main in *Global list.*
- Line 2: extern printf, puts the entry printf into the *Symbol table,* with type set as external.
- Line 3: section .data, an entry is made into the *Section table,* and *lc* is reset to zero.
- Line 4: my_array, *label* entry is made into the *Symbol table* with:

  Type as variable
  Size as 40 bytes
  Location as zero(current *lc* value)
  Section-id as 1
  Is-global set to false

  and *lc* is set to 40.


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
```

- Line 5: format, *label* entry is made into the *Symbol table* with

  Type as variable
  Size as 3 bytes
  Location as 40
  Section-id as 1
  Is-global set to false

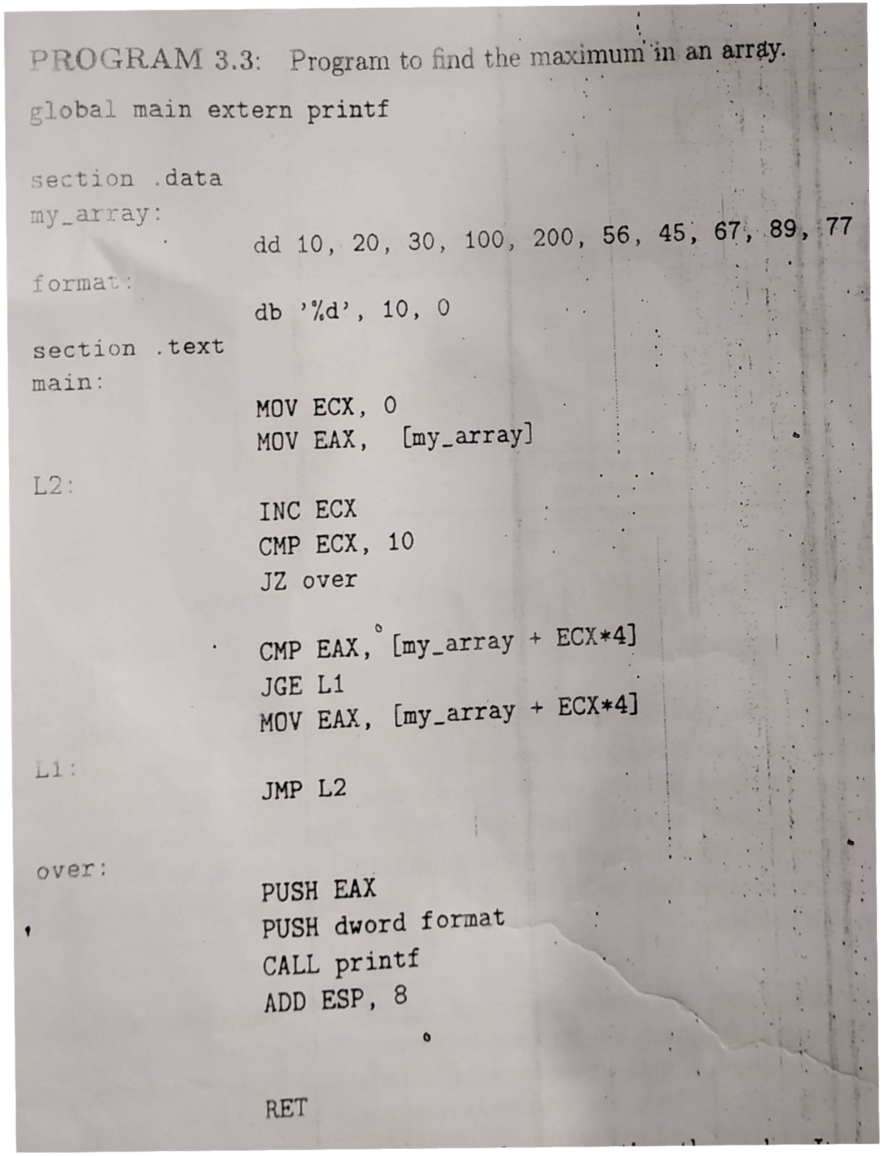
  and *lc* is set to 43.

- Line 6: section .text, the current value of *lc* is stored as Size of Section 1 in the *Section table.*

  A new entry is created for section .text into the *Section table,* and *lc* is reset to zero.

- Line 7: main, the *label* entry is made into the *Symbol table* with

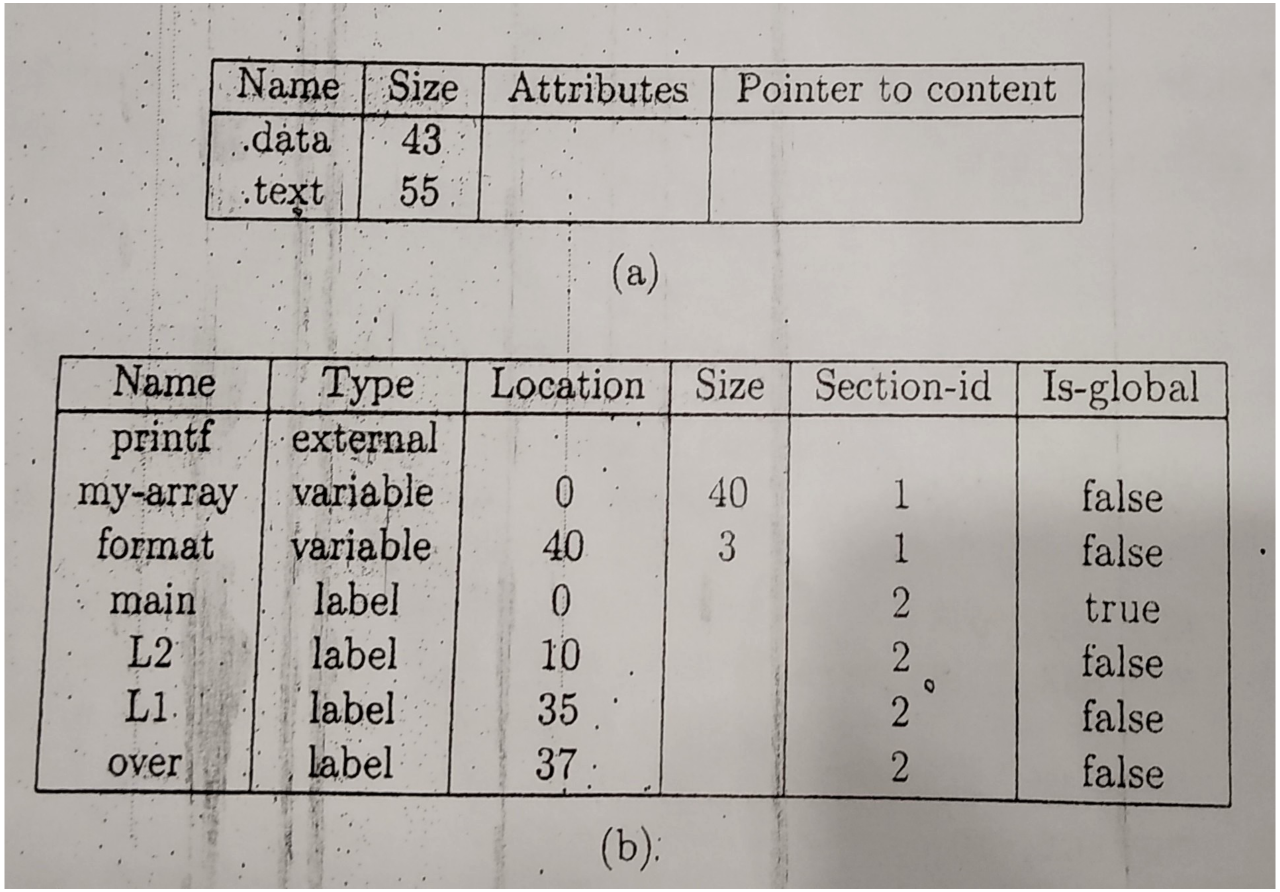
  Size as 5 bytes
  lc is incremented to 5.


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 I* continues by computing the length of instructions and incrementing *lc* accordingly.
- The label L2, L1, and over are stored in *Symbol table.*
- Finally after processing RET instruction, the *end-of-file* is reached.
- The current lc value 55 is entered as Size of the section .text.
- *Global list* is consulted to mark the label *main* as the global in the *Symbol table*, by setting Is-global to true.

Two-Pass Assembler – *Pass I* (tables)



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)

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

Two-Pass Assembler – *Pass II*

- The main task of this pass is to generate the machine code
- It uses tables created in *Pass I* and writes the generated code into an object file.

  - *Please see the flowchart: Pass II* of the two-pass assembler

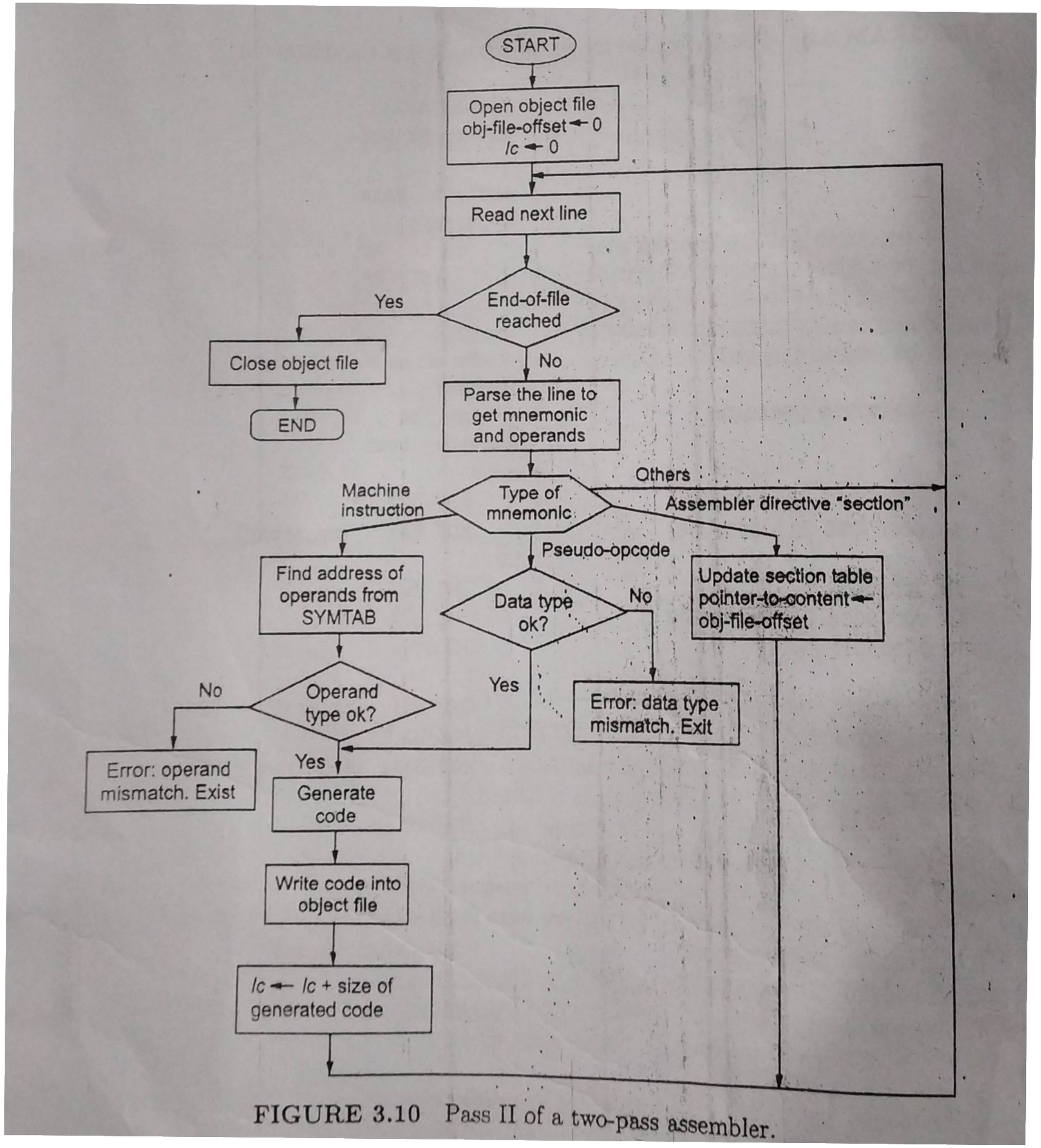

START
Open object file
obj-file-offset ← 0
lc ← 0
Read next line
End-of-file reached
Yes
No
Close object file
END
Parse the line to get mnemonic and operands
Type of mnemonic
Machine instruction
Others
Assembler directive "section"
Pseudo-opcode
Find address of operands from SYMTAB
Data type ok?
No
Yes
Update section table pointer-to-content ← obj-file-offset
Error: data type mismatch. Exit
Operand type ok?
No
Yes
Error: operand mismatch. Exist
Generate code
Write code into object file
lc ← lc + size of generated code

FIGURE 3.10 Pass II of a two-pass assembler.

- The *object-file-offset* and *lc* are initialized with zero.
- The source program lines are read one by one sequentially and corresponding code is generated.
- The actual code generation will definitely depend upon the target processor.

Two-Pass Assembler – *Code generated for the Program*

- *Please see the Code generated: for the program to find the maximum*

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

```
                                        global main
                                        extern printf

                                        section .data
                                        my_array:
00000000 0A000000140000001E-                dd 10, 20, 30, 100, 200, 56, 45,
00000009 00000064000000C800-                   67, 89, 77
00000012 0000380000002D0000-
0000001B 004300000059000000-
00000024 4D000000

                                        format:
00000028 25640A00                           db '%d', 10, 0
                                        section .text
                                        main:
00000000 B900000000                         MOV ECX, 0
00000005 A1[00000000]                       MOV EAX,  [my_array]
                                        L2:
0000000A 41                                 INC ECX
0000000B 81F90A000000                       CMP ECX, 10
00000011 7412                               JZ over

00000013 3B048D[00000000]                   CMP EAX, [my_array + ECX*4]
0000001A 7D07                               JGE L1
0000001C 8B048D[00000000]                   MOV EAX, [my_array + ECX*4]
                                        L1:
00000023 EBE5                               JMP L2


                                        over:
00000025 50                                 PUSH EAX
00000026 68[28000000]                       PUSH dword format
0000002B E8(00000000)                       CALL printf
00000030 81C408000000                       ADD ESP, 8


00000036 C3                                 RET

```

Two-Pass Assembler – *Code generated for the Program*

- The updated Section table

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

- External reference list

Name	***Offsets to be corrected***
printf	43

Single-Pass Assembler

- The single-pass assembler scans the input assembly language program only once. Thus, it is 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.

:	
X: db 10	→ *makes entry in SYMTAB and simultaneously generates code to reserve one bye initialized to 10.*
:	
MOV AL, X	→ *here address of **X** is already available in the SYMTAB which can be use to generate code.*
MOV Y, AL	→ *Address of **Y** is not known at this point of time. It will only be available when **Y:RESB** is seen. This type of situation is known as **Forward Reference.***
:	
Y: RESB	

Single-Pass Assembler

- The single-pass assembler scans the input assembly language program only once. Thus, it is 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.

```
:
X:  db 10     → makes entry in SYMTAB and simultaneously generates code to reserve one bye initialized
                to 10.
:
MOV AL, X     → here address of X is already available in the SYMTAB which can be use to generate code.
MOV Y, AL     → Address of Y is not known at this point of time. It will only be available when Y:RESB is
                seen. This type of situation is known as Forward Reference.
:
Y:  RESB
```

Q. What about Two-Pass Assembler?

Single-Pass Assembler (1)

- There will be no problem of forward reference in a two-pass assembler.
  - → Since, the *Pass I* finds the symbols defined in the input program, and at the end of *Pass I,* information is available in the *SYMTAB,* which is used by *Pass II* to generates the code.

- To resolve the problem of forward referencing, the assembler, <u>upon seeing a symbol not yet defined, should make an entry into the *SYMTAB*</u> .
  - → The value of Type field for symbol should be inferred from the context and marked as undefined.
  - → When the symbol gets defined, it should be checked with the entry already present in the *SYMTAB*.
  - → Any discrepancy may be resolved accordingly, or reported as an error.

Single-Pass Assembler (2)

- There exists another potential problem regarding the code generation
  - → An instruction containing forward reference cannot be translated entirely at the time it is encounters
  - → Since, address of the operand symbol in not available at that time.
  - → The Address will be known only after the definition has been seen, and at that time, all the locations corresponding to the reference of the symbol cab be fixed.
  - → The process is known as ***backpatching.***

- *Backpatching*
  - → For each froward referenced symbol, a corresponding list of locations requiring corrections is maintained. The list is called as *forward reference list* .
  - → As and when a symbol gets defined, the corresponding list be traversed and the locations corrected.

Single-Pass Assembler (*Flowchart*)

- *Please see the flowchart:* single-pass assembler
  - → The flow of logic combines both *Pass I* and *Pass II* of a two-pass assembler
  - → Some extra processing is needed.

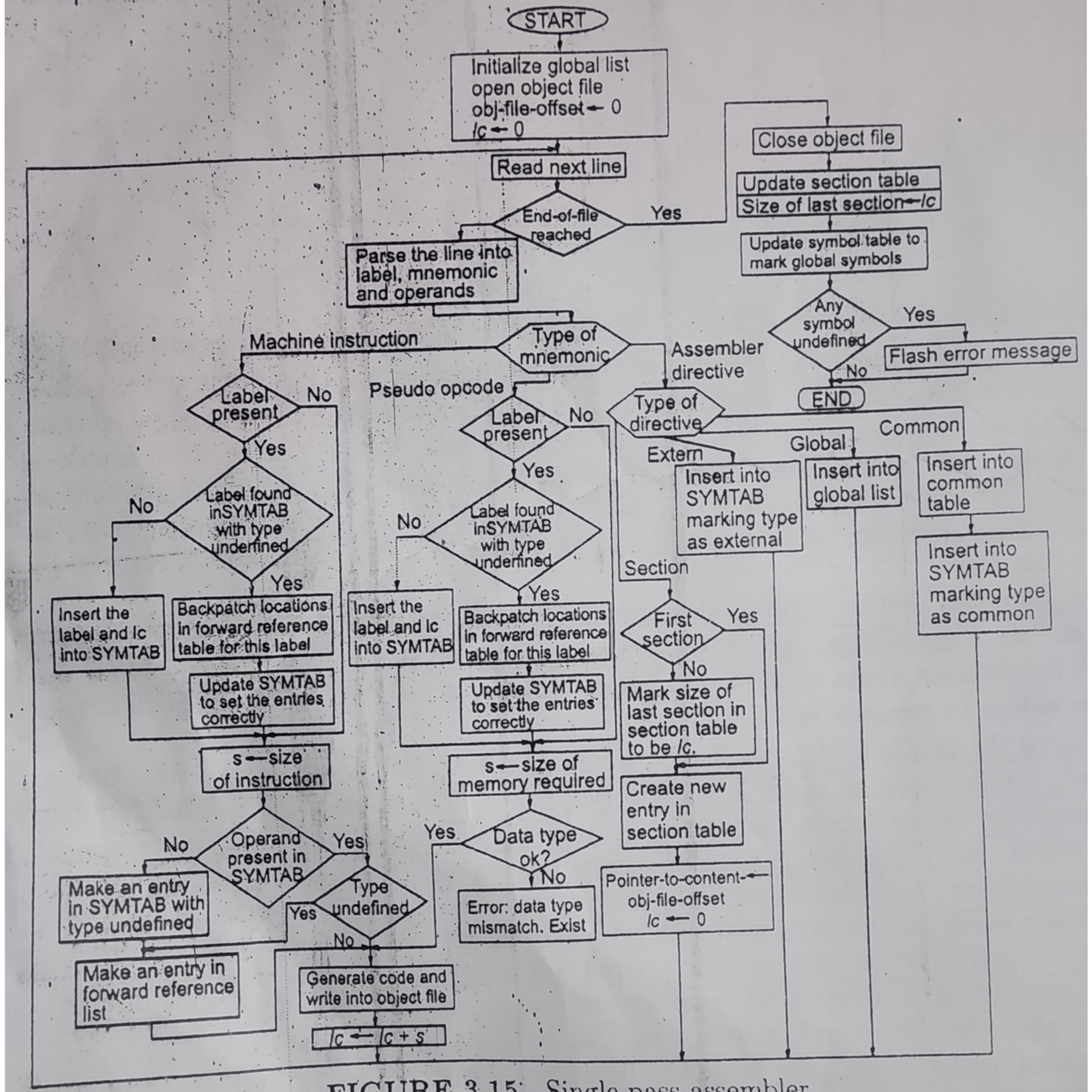

START
Initialize global list
open object file
obj-file-offset ← 0
lc ← 0
Read next line
End-of-file reached
Yes
Close object file
Update section table
Size of last section ← lc
Update symbol table to mark global symbols
Any symbol undefined
Yes
Flash error message
No
END
Parse the line into label, mnemonic and operands
Type of mnemonic
Machine instruction
Pseudo opcode
Assembler directive
Label present
No
Yes
Label found inSYMTAB with type undefined
No
Yes
Insert the label and lc into SYMTAB
Backpatch locations in forward reference table for this label
Update SYMTAB to set the entries correctly
s ← size of instruction
Operand present in SYMTAB
No
Yes
Make an entry in SYMTAB with type undefined
Type undefined
Yes
No
Make an entry in forward reference list
Generate code and write into object file
lc ← lc + s
Label present
No
Yes
Label found inSYMTAB with type undefined
No
Yes
Insert the label and lc into SYMTAB
Backpatch locations in forward reference table for this label
Update SYMTAB to set the entries correctly
s ← size of memory required
Data type ok?
Yes
No
Error: data type mismatch. Exist
Type of directive
Extern
Global
Common
Insert into SYMTAB marking type as external
Insert into global list
Insert into common table
Insert into SYMTAB marking type as common
Section
First section
Yes
No
Mark size of last section in section table to be lc.
Create new entry in section table
Pointer-to-content ← obj-file-offset
lc ← 0

FIGURE 3.15 Single-pass assembler.

Single-Pass Assembler (*Extra Processing*)

1. On reaching the *end-of-file* marker:

   - Apart from closing the object file, updating Section table, and *SYMTAB*, it also needs to check whether some forward referenced symbol is still left unidentified.

     → *Situation may occur is 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 he later part of the program. An entry is made into the *SYMTAB* 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

Single-Pass Assembler (*Extra Processing*)

2 In the *code generation* process:

- While translating an instruction, its operands are looked for in the *SYMTAB*, and If the operand is not found:

  → *A new entry is made into the SYMTAB with type undefined, assuming that it is a froward reference.*

  → *Also, an entry is created in the forward reference list, for this label along with the location requiring correction.*

- If he operand is already present in the *SYMTAB* with type undefined:

  → *The label is searched for in the froward reference list, and the current location is added to the list of locations requiring corrections for this label.*

Single-Pass Assembler (*Extra Processing*)

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 *SYMTAB*

- At the same time, it should be checked, whether it is a forward referenced symbol or not ?

  → *If the label is a froward referenced, the backpatching procedure is invoked <u>to correct the locations noted in the forward reference table entry corresponding to this symbol.</u>*

Single-Pass Assembler (*Program*)

- *Please see the Program:* To find the maximum of a set of numbers.

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
```

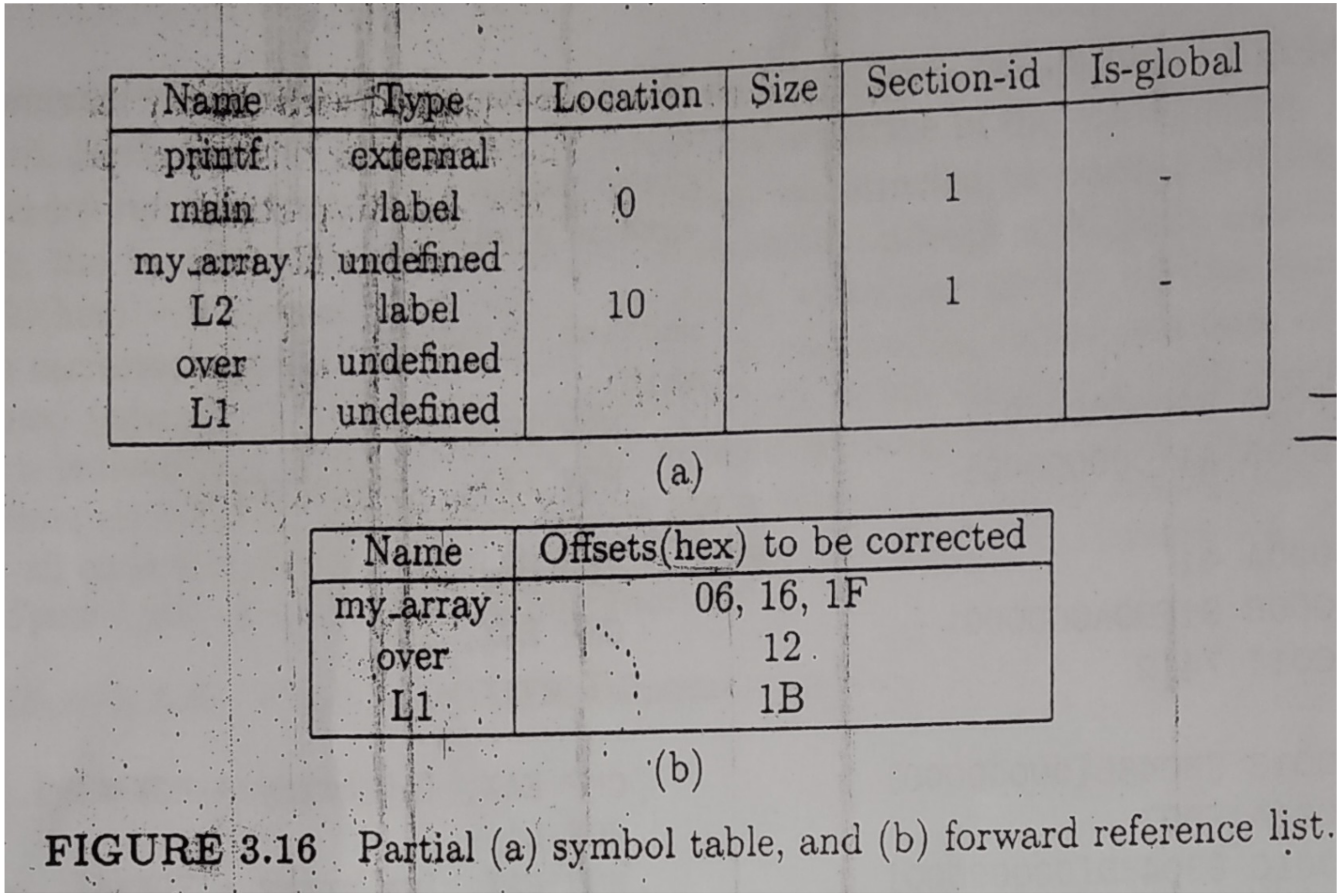
Single-Pass Assembler – *Code generated for the Program*

- *Please see the Code generated: for the program to find the maximum*

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 00000000 B900000000                  MOV ECX, 0
 7 00000005 A1[00000000]                MOV EAX,  [my_array]
 8                                  L2:
 9 0000000A 41                          INC ECX
10 0000000B 81F90A000000                CMP ECX, 10
11 00000011 7412                        JZ over
12
13 00000013 3B048D[00000000]            CMP EAX, [my_array + ECX*4]
14 0000001A 7D07                        JGE L1
15 0000001C 8B048D[00000000]            MOV EAX, [my_array + ECX*4]
16                                  L1:
17 00000023 EBE5                        JMP L2
18
19                                  over:
20 00000025 50                          PUSH EAX
21 00000026 68[28000000]                PUSH dword format
22 0000002B E8(00000000)                CALL printf
23 00000030 81C408000000                ADD ESP, 8
24
25
26 00000036 C3                          RET
27                                  section .data
28                                  my_array:
29 00000000 0A000000140000001E-         dd 10, 20, 30, 100, 200, 56, 45,
30 00000009 00000064000000C800-             67, 89, 77
31 00000012 0000380000002D0000-
32 0000001B 004300000059000000-
33 00000024 4D000000
34                                  format:
35 00000028 25640A00                    db '%d', 10, 0
36
```

Single-Pass Assembler (*Tables*)



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				

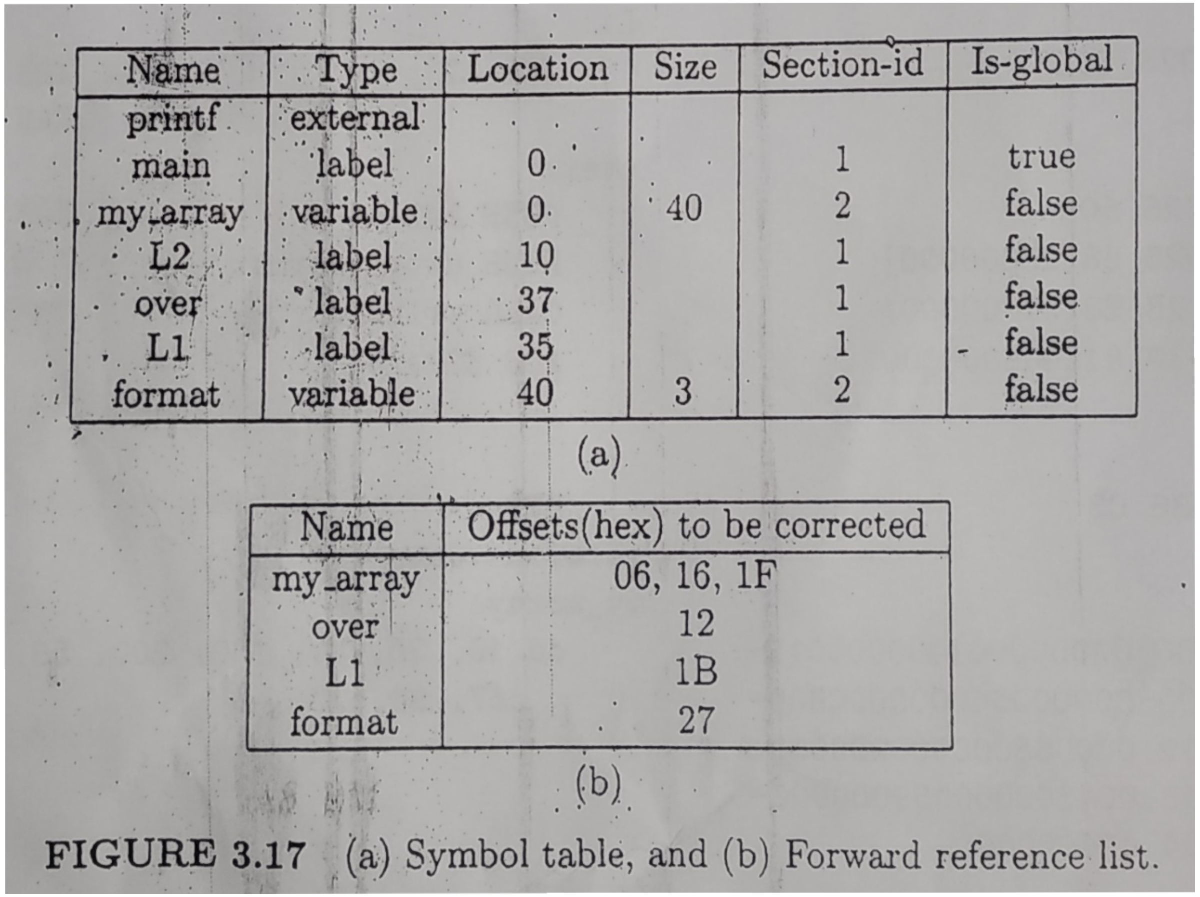
(a)

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.

Single-Pass Assembler (*Tables*)



Name	Type	Location	Size	Section-id	Is-global
printf	external				
main	label	0		1	true
my_array	variable	0	40	2	false
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.

Load-and-go Assembler

- Assemblers producing the object code directly into the memory and the code is ready for execution.

- Suitable for small programs in their development stages:
  - → *Where after small modification, it is required to check the results.*

- The development time is reduced since:
  - → *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 (discussed in the next slide)

Load-and-go Assembler (*Drawbacks*)

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.

3. 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.(should be acceptable to linker, etc.)

Object File Formats

1. Intel hex format
2. obj – Microsoft OMF object files
3. win32 – Microsoft Win32 object files
4. coff – Common object file format
5. elf – Executable and linkable
6. aout – Linux a.out object files.

References

- **Book:** System Software: An Introduction to System Programming, *Third Edition*, Leland L. Beck and D. Manjula, Pearson Education.

- **Book:** System Software, S. Chattopadhyaya (2011), PHI Learning.

- **Book:** Alfred V. Aho, Monica S. Lam, Ravi Sethi, J D Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd Edition, Prentice Hall, 2006.

- **PPT:** Hsung-Pin Chang, Department of Computer Science, National Chung Hsing University, **Chapter 1: Background**

- **PPT:** Hsung-Pin Chang, Department of Computer Science, National Chung Hsing University, **Chapter 2:** Assembler (for additional reading)