UNIT – I

Introduction: Components of System Software: Text editors, Loaders, Assemblers, Macro processors, Compilers, Debuggers. Machine Structure, Machine language and Assembly Language. Assemblers: General design procedure, design of two pass assembler

1. INTRODUCTION :

System programming involves designing and writing computer programs that allow the computer hardware to interface with the programmer and the user, leading to the effective execution of application software on the computer system. Typical system programs include the operating system and firmware, programming tools such as compilers, assemblers, I/O routines, interpreters, scheduler, loaders and linkers as well as the runtime libraries of the computer programming languages.

2. Component of System Software :

System software is a type of computer program that is designed to run a computer's hardware and application programs. If we think of the computer system as a layered model, the system software is the interface between the hardware and **user applications**

Components are :

- Text editors
- Loaders
- Assemblers
- Macro processors
- Compilers
- Debuggers
- Machine Structure
- Machine language and Assembly Language.

i. Text Editor :

A **text editor** is a type of program used for editing plain text files. Such programs are sometimes known as "**notepad**" software, following the Microsoft Notepad.

ii. Loaders :

a **loader** is the part of an operating system that is responsible for loading programs and libraries. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution. Loading a program involves reading the contents of the executable file containing the program instructions into memory, and then carrying out other

required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code.

iii. **Linkers :**

Linker is a **program** that takes one or more objects generated by a compiler and combines them into a single executable **program**. Loader is the part of an operating **system** that is responsible for loading programs from executables (i.e., executable files) into memory, preparing them for execution and then executing them.

iv. **Assembler :**

Assembler is a computer program which is used to translate program written in Assembly Language in to machine language. The translated program is called as object program. Assembler checks each instruction for its correctness and generates diagnostic messages, if there are mistakes in the program. Various steps of assembling are:

1. Input source program in Assembly Language through an input device.
2. Use Assembler to produce object program in machine language.
3. Execute the program.

v. **Compiler :**

A compiler is a program that translates a programme written in HLL to executable machine language. The process of transferring HLL source program in to object code is a lengthy and complex process as compared to assembling. Compliers have diagnostic capabilities and prompt the programmer with appropriate error message while compiling a HLL program. The corrections are to be incorporated in the program, whenever needed, and the program has to be recompiled. The process is repeated until the program is mistake free and translated to an object code. Thus the job of a complier includes the following:

1. To translate HLL source program to machine codes.
2. To trace variables in the program
3. To include linkage for subroutines.
4. To allocate memory for storage of program and variables.
5. To generate error messages, if there are errors in the program.

vi. **Interpreter :**

The basic purpose of interpreter is same as that of complier. In compiler, the program is translated completely and directly executable version is generated. Whereas interpreter translates each instruction, executes it and then the next instruction is translated and this goes on until end of the program. In this case, object code is not stored and reused. Every time the program is executed, the interpreter translates each

instruction freshly. It also has program diagnostic capabilities. However, it has some disadvantages as below:

1. Instructions repeated in program must be translated each time they are executed.

2. Because the source program is translated fresh every time it is used, it is slow process or execution takes more time. Approx. 20 times slower than complier.

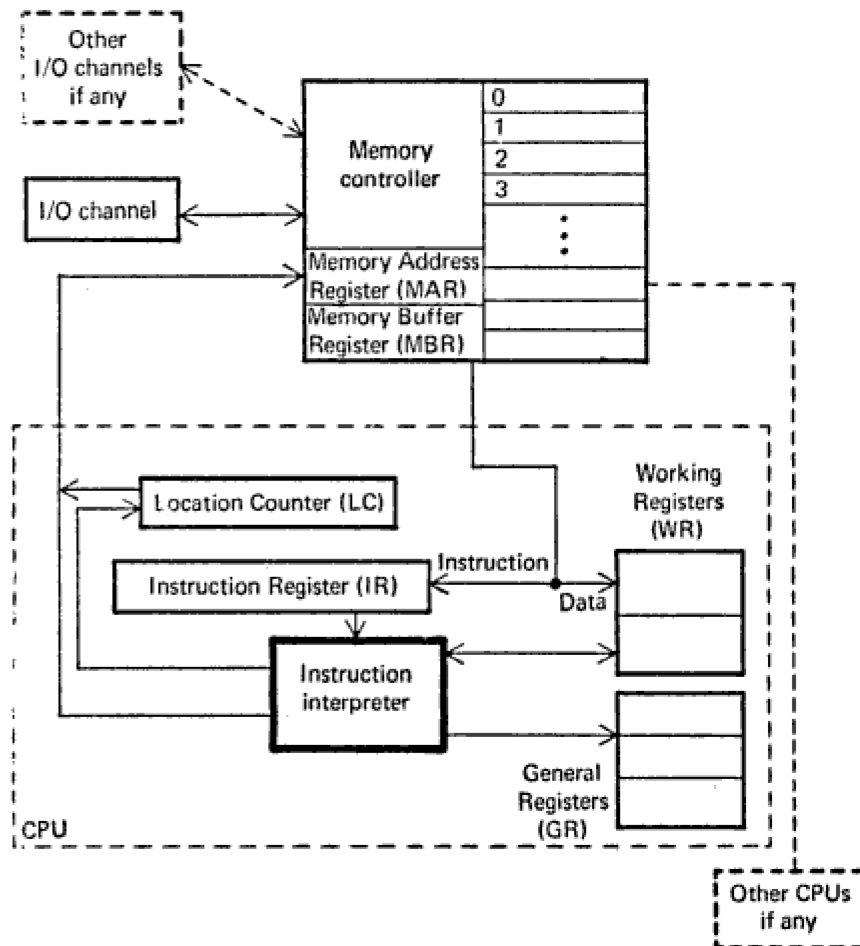
vii. **Macro Processor :**

A **macro processor** is a program that copies a stream of text from one place to another, making a systematic set of replacements as it does so. **Macro processors**are often embedded in other programs, such as assemblers and compilers. Sometimes they are standalone programs that can be used to process any kind of text.

viii. **Debugger :**

A **debugger** is a computer **program** used by **programmers** to test and debug a target **program**. **Debuggers** may use instruction-set simulators, rather than running a **program** directly on the processor to achieve a higher level of control over its execution.

ix. **Machine structure :**



The above structure consist of..

1. Instruction interpreter
2. Location counter
3. Instruction register
4. Working register
5. General register

The Instruction Interpreter Hardware is basically a group of circuits that perform the operation specified by the instructions fetched from the memory.
The Location Counter can also be called as Program/Instruction Counter simply points to the current instruction being excuted.
The working registers are often called as the "scratch pads" because they are used to store temporary values while calculation is in progress.
This CPU interfaces with Memory through MAR & MBR
MAR (Memory Address Register) - contains address of memory location (to be read from or stored into)
MBR (Memory Buffer Register) - contains copy of address specified by MAR
Memory controller is used to transfer data between MBR & the memory location specified by MAR
The role of I/O Channels is to input or output information from memory.

Machine language :

Sometimes referred to as **machine code** or **object code**, **machine language** is a collection of binary digits or bits that the computer reads and interprets. Machine language is the only language a computer is capable of understanding.

The exact machine language for a program or action can differ by operating system on the computer. The specific operating system will dictate how a compiler writes a program or action into machine language.

Computer programs are written in one or more programming languages, like C++, Java, or Visual Basic. A computer cannot directly understand the programming languages used to create computer programs, so the program code must be compiled. Once a program's code is compiled, the computer can understand it because the program's code has been turned into machine language.

Machine language example:

Below is an example of machine language (binary) for the text "Hello World".

```
01001000 01100101 01101100 01101100 01101111 00100000 01010111 01101111 01110010
01101100 01100100
```

Below is another example of machine language (non-binary), which will print the letter "A" 1000 times to the computer screen.

```
169 1 160 0 153 0 128 153 0 129 153 130 153 0 131 200 208 241 96
```

Assembly Language :

Each personal computer has a microprocessor that manages the computer's arithmetical, logical, and control activities.

Each family of processors has its own set of instructions for handling various operations such as getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instructions'.

A processor understands only machine language instructions, which are strings of 1's and 0's. However, machine language is too obscure and complex for using in software development. So, the low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

Advantages of Assembly Language :

- It requires less memory and execution time;
- It allows hardware-specific complex jobs in an easier way;
- It is suitable for time-critical jobs;
- It is most suitable for writing interrupt service routines and other memory resident programs.

SOFTWARE

A program or group of programs designed for end users.

TYPES

1. System Software
2. Application Software

DIFFERENCES BETWEEN SYSTEM SOFTWARE & APPLICATION SOFTWARE

SYSTEM SOFTWARE	APPLICATION SOFTWARE
1. It consists of low level programs that interact with the computer at the very basic level.	It sits at the top of the system software because it is unable to run without the operating system & system utilities.
2. It controls and coordinates the computer operations.	It is used for special and general purpose Operations.
3. **Functions:**	
Programs Manages Resources Controls I/O Communications	- Word Processing - Desktop Publishing - Spreadsheets - Databases - Telecommunications

4. **Examples:**

- OS
  - Windows
  - Macintosh system
  - Unix
- Compilers
- Assemblers
- Loaders
- Linkers

- MS Word
- Excel
- Lotus
- Access
- Dbase

Language Translators :

Figure : 1

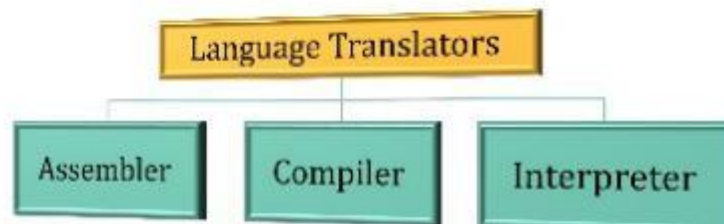
A **Language Translator** ***(Figure : 1)*** is the program which converts a users program written in some language to another language.

- The language in which the users program is written is called the source language.
- The language to which the source language is converted is called the target language.

Why Language Translators ?

Computer only understands object code (machine code). It does not understand any source code.
The Programmer writes the source code and then translator converts it in machine readable format (object code).

Types of Language Translators: ***(Figure : 2)***



Assembler :

An **assembler** ***(Figure : 3)*** is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations

Figure : 3

Characteristics of Assembler :

Assembly language programming is
difficult Takes longer time
Takes longer to
debug Difficult to
maintain

Compiler :

Complier *(Figure : 4)* is a system program that translates an input program in a high level language into its machine language equivalent

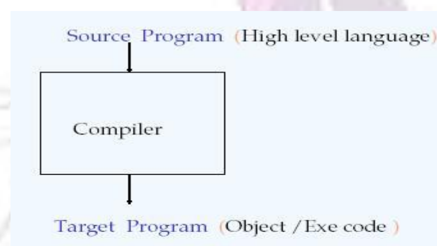


Figure : 4

High Level Language Features :

High degree of machine
independence Good data
structures

Improved debugging
capability Good
documentation

Need of assembly language programming?

1. **Performance issues**

   For some applications, speed and size of the code are critical. An expert assembly language programmer can often produce code that is much smaller and much faster than a high level programmer can. Example – embedded applications such as the code on a smart card, the code in a cellular telephone, BIOS routines, inner loops of performance critical applications etc.

2. **Access to the machine**

   Some procedures need complete access to the hardware, something which is impossible in high level languages.
   Example – Low levels interrupts and traps handlers in an operating system etc.

Loader : It is the part of the operating system (a system software) that is responsible for loading programs from secondary storage devices into memory, preparing them for execution and then executing them.

Linker : *(Figure : 5)*

- A large program can be splitted up into many modules. All these modules have to be connected logically and linked to form a single program.

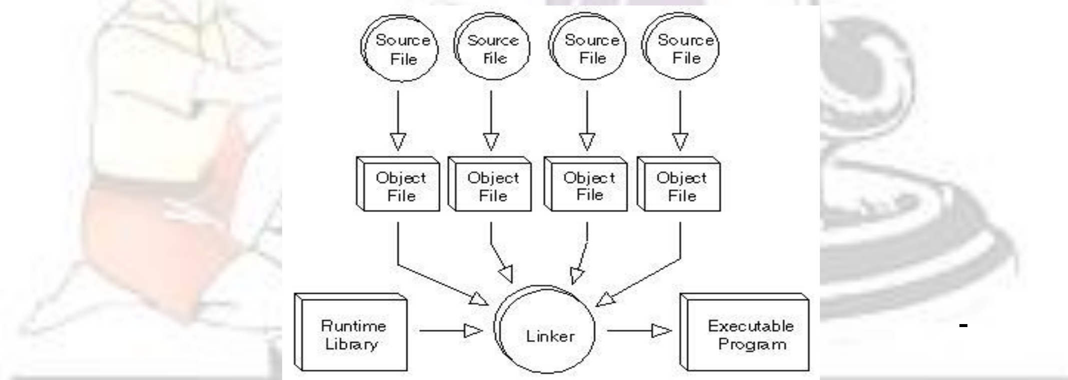


Figure 5

Elements of Assembly Language

An assembly language programming provides three basic features which simplify programming when compared to machine language.

1. ***Mnemonic Operation Codes :***

   Mnemonic operation code / Mnemonic Opcodes for machine instruction eliminates the need to memorize numeric operation codes. It enables assembler to provide helpful error diagnostics. Such as 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 assembeler performes memory bindinding to these names; the programmer need not 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 -5 into $(11111010)_2$ or 10.5 into $(41A80000)_{16}$

Statement format :

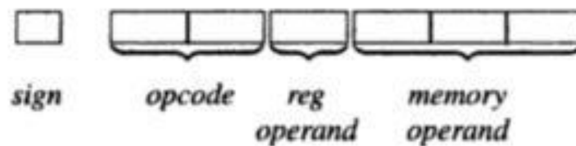
An assembly language statement has the following format :

[Label] <Opcode> <operand Spec> [, operand Spec> ..]

Mnemonic Operation Codes :

Instruction opcode	*Assembly mnemonic*	*Remarks*
00	STOP	Stop execution
01	ADD	*First operand is modified*
02	SUB	*Condition code is set*
03	MULT	
04	MOVER	Register ← memory move
05	MOVEM	Memory ← register move
06	COMP	Sets condition code
07	BC	Branch on condition
08	DIV	Analogous to SUB
09	READ	*First operand is not used*
10	PRINT	

Instruction Format :



Sign is not a part of Instruction

An Assembly and equivalent machine language program :

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

Assembly Language Statements :

Three Kinds of Statements

1. Imperative Statements
2. Declaration Statements
3. Assembler Directives

a) **Imperative Statements :** It indicates an action to be performed during the execution of the assembled program. Each imperative statement typically translates into one machine instruction.

b) **Declaration Statements :** Two types of declaration statements is as follows

[Lable] DS <Constant>

[Label] DC <Value>

The DS (Declare Storage) statement reserves areas of memory and associates names with them.

E.g. A DS 1

B DS 150

First statement reserves a memory of 1 word and associates the name of the memory as A.

Second statement reserves a memory of 150 word and associates the name of the memory as B.

The DC (Declare Constant) Statement constructs memory word containing constants

e.g. ONE DC '1'

Associates the name ONE with a memory word containing the value '1' . The programmer can declare constants in decimal,binary, hexadecimal forms etc., These values are not protected by the assembler. In the above assembly language program the value of ONE Can be changed by executing an instruction MOVEM BREG,ONE

c. Assembler Directives :

Assembler directives instruct the assembler to perform certain actions during the assembly of a program. Some Assembler directives are described in the following

START <Constant>

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>]

It Indicates the end of the source program

Pass Structure of Assembler :

One complete scan of the source program is known as a pass of a Language Processor.

Two types 1) Single Pass Assembler 2) Two Pass Assembler.

Single Pass Assembler :

First type to be developed Most Primitive

Source code is processed only once.

The operand field of an instruction containing forward reference is left blank intially

Eg) MOVER BREG,ONE

Can be only partially synthesized since ONE is a forward reference

During the scan of the source program, all the symbols will be stored in a table called **SYMBOL TABLE.** Symbol table consists of two important fields, they are symbol name and address.

All the statements describing forward references will be stored in a table called Table of Incompleted Instructions (TII)

TII (Table of Incomplete instructions)

Instruction Address	Symbol
101	ONE

By the time the END statement is processed the symbol table would contain the address of all symbols defined in the source program

Two Pass Assembler :

Can handle forward reference problem easily.

First Phase : ***(Analysis)***

- Symbols are entered in the table called Symbol table
- Mnemonics and the corresponding opcodes are stored in a table called Mnemonic table
- LC Processing

Second Phase : ***(Synthesis)***

- Synthesis the target form using the address information found in Symbol table.
- First pass constructs an Intermediated Representation (IR) of the source program for use by the second pass.

Data Structure used during Synthesis Phase :

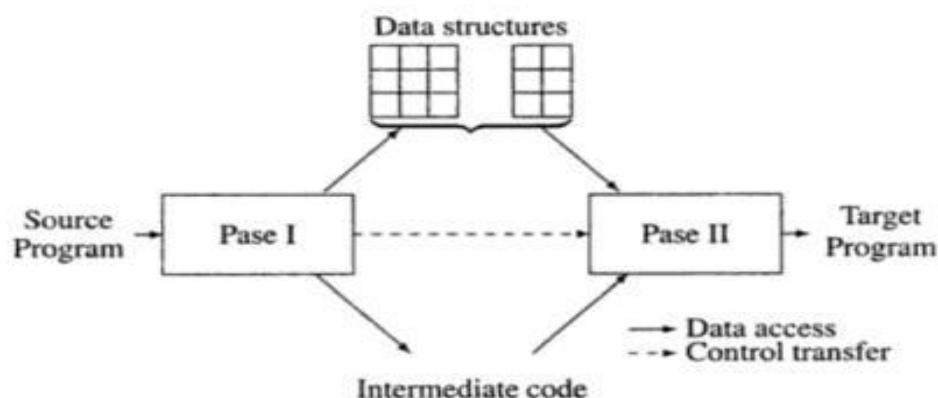
1. Symbol table

symbol	address
AGAIN	104
N	113

2. Mnemonics table

mnem-onic	op code	length
ADD	01	1
SUB	02	1

Processed form of the source program called Intermediate Code (IC)



ADVANCED ASSEMBLER DIRECTIVES

1. ORIGIN
2. EQU
3. LTROG

ORIGIN :

Syntax : ORIGIN < address spec>

< address spec>can be an <operand spec> or constant

Indicates that Location counter should be set to the address given by < address spec>

This statement is useful when the target program does not consist of consecutive memory words.

Eg) ORIGIN Loop + 2

EQU :

Syntax

<symbol> EQU <address spec>

<address spec>operand spec (or) constant

Simply associates the name symbol with address specification No Location counter processing is implied

Eg) Back EQU Loop

LTORG : (Literal Origin)

Where should the assembler place literals ?

It should be placed such that the control never reaches it during the execution of a program.

By default, the assembler places the literals after the END statement.

LTROG statement permits a programmer to specify where literals should be placed.

DESIGN OF TWO PASS ASSEMBLER:

Pass I : (Analysis of Source Program)

1) Separate the symbol, mnemonic opcode and operand fields
2) Build the symbol table.
3) Perform LC processing.
4) Construct intermediate representation

PASS 2:-

Processes the intermediate representation (IR) to synthesize the target program.

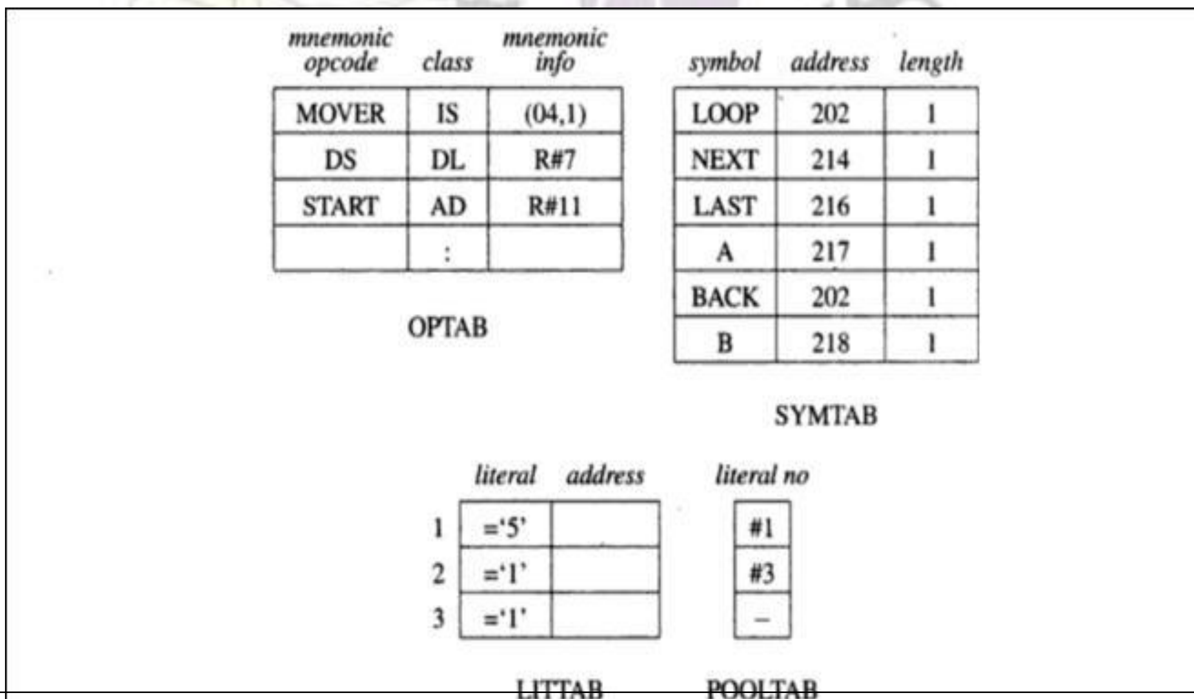
Pass 1 of the Assembler:-

Pass 1 uses the following data structures:-

OPTAB: - A table of mnemonic opcodes and related info.

SYMTAB: - Symbol Table.

LITTAB: - A table of literals used in the program.



OPTAB

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

SYMTAB

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

LITTAB

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

POOLTAB

literal no
#1
#3
–

Data structures of assembler Pass I

OPTAB :

contains the fields mnemonic opcode, class and mnemonic information.

The 'class' field indicates whether the opcode corresponds to an imperative statement (IS), a declaration statement (DL) or an assembler directive (AD).

If an imperative statement is present, then the mnemonic info field contains the pair (machine opcode, instruction length) else it contains the pair id of a routine to handle the declaration or directive statement.

SYMTAB :

contains the fields address and length.

The processing of an assembly statement begins with the processing of its label field. If it contains a symbol, the symbol and the value in LC is copied into a new entry of SYMTAB.

If it is an imperative statement, then length of the machine instruction is simply added to the LC. The length is also entered into the symbol table

LITTAB :

The LITTAB is used to collect all literals used in the program.

The awareness of different literals pools in maintained by an auxiliary table POOLTAB.

POOLTAB :

This table contains the literal no. of starting literal of each literal pool.

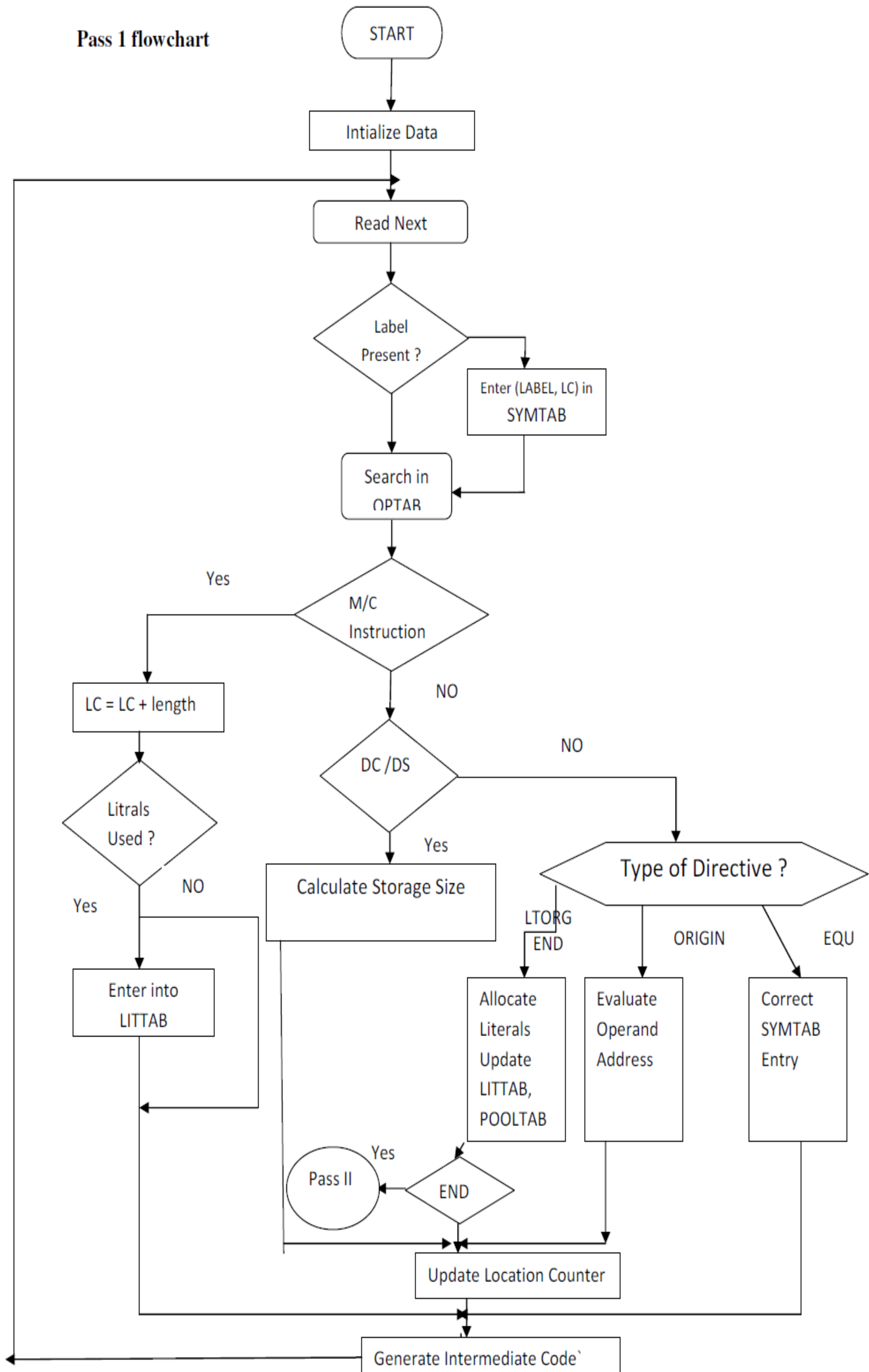
The detailed design of pass1 assembler and pass2 assembler are explained in detail with the flowcharts.

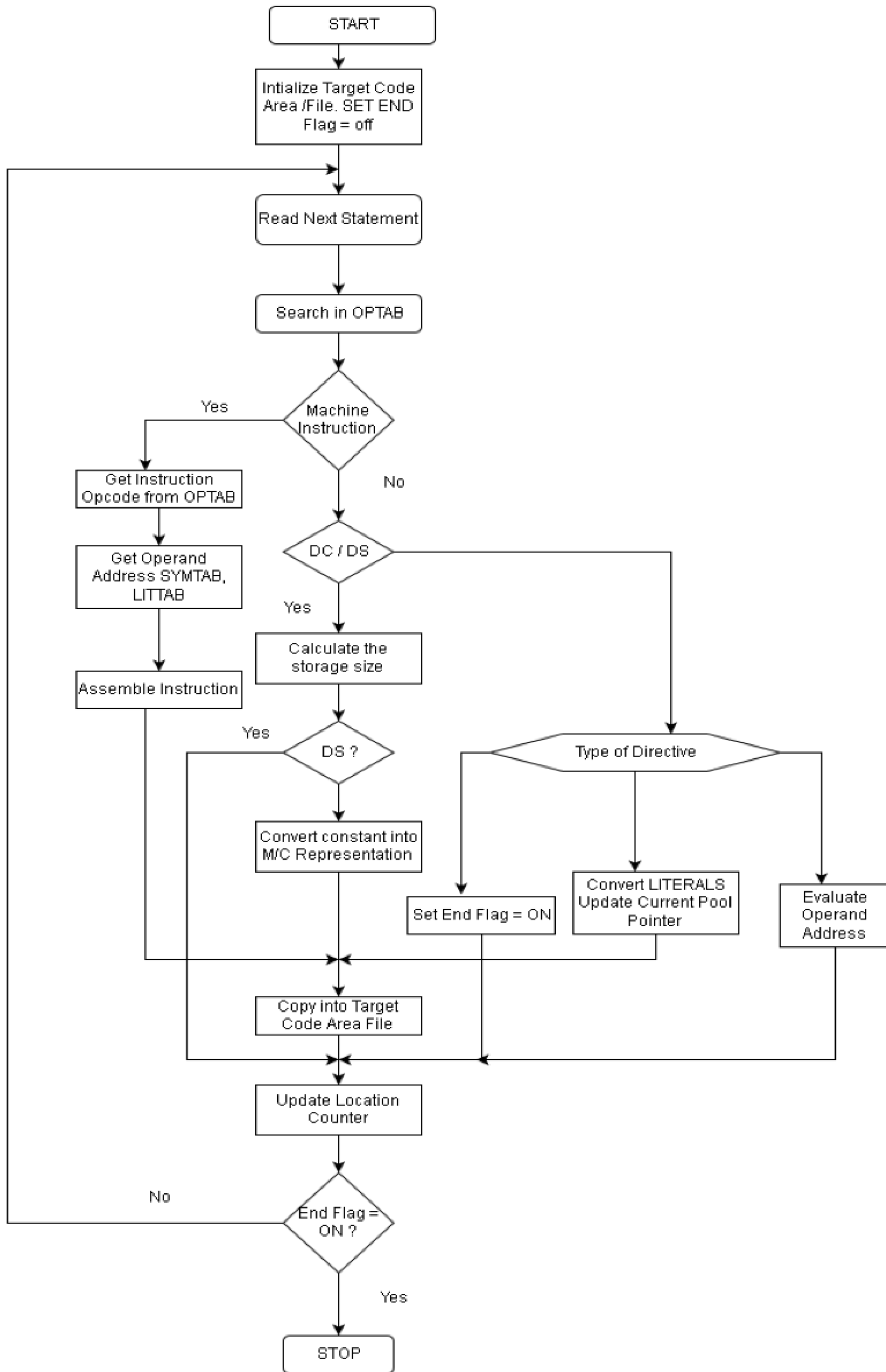
Pass 1:

Input:Source program
Output:Intermediate code

Pass 2:

Input:Intermediate code
Output:object code


Pass 1 flowchart
START
Intialize Data
Read Next
Label
Present ?
Enter (LABEL, LC) in
SYMTAB
Search in
OPTAB
Yes
M/C
Instruction
LC = LC + length
NO
DC /DS
NO
Litrals
Used ?
Yes
Type of Directive ?
NO
Yes
Calculate Storage Size
LTORG
END
ORIGIN
EQU
Enter into
LITTAB
Allocate
Literals
Update
LITTAB,
POOLTAB
Evaluate
Operand
Address
Correct
SYMTAB
Entry
Yes
Pass II
END
Update Location Counter
Generate Intermediate Code`


START
Intialize Target Code Area /File. SET END Flag = off
Read Next Statement
Search in OPTAB
Machine Instruction
Yes
No
Get Instruction Opcode from OPTAB
Get Operand Address SYMTAB, LITTAB
Assemble Instruction
DC / DS
Yes
Calculate the storage size
DS ?
Yes
Convert constant into M/C Representation
Type of Directive
Set End Flag = ON
Convert LITERALS Update Current Pool Pointer
Evaluate Operand Address
Copy into Target Code Area File
Update Location Counter
End Flag = ON ?
No
Yes
STOP

The output of pass 1 assembler is intermediate code(IR)

The output of pass 2 assembler is machine code.

Intermdeiate Code Forms :

Consists of a set of (IC) units. Each unit consisting of the following three fields.

1. Address
2. Mnemonic Opcode
3. Operands

	Mnemonic	
Address	Opcode	Operands

Mnemonic Opcode :

This field consists of the form (Statement Class,Code)

Statement class can be an

1. Imperative Statement (IS)
2. Declaration Statement (DL)
3. Assembler Statement (AD)

Code :

1. DC01
2. DS02

AD

1. Start01
2. END02
3. ORIGIN03
4. EQU04
5. LTORG05

For IS, the code will be mnemonic opcode

There are two variants of intermediate code which differs in the information contained in their operand fields.

Variant 1

Operand Field

It can be an

1. Register operand
2. Memory operand

Register Operand	(or) Conditional Code (CC)
A1	LT1
B2	LE2
C3	EQ3
D4	GT4
	GE5
	ANY6

Memory Operand

It is represented by a pair of the form (operand class, code) Operand class can be constant (C), Literal (L), Symbol (S)

For a constant, the code field contains the internal representation of the constant itself.

Eg) Generate the intermediate code of the followin program (or)

Generate the output of Pass I of the Assembly language program.

```
        START 200
        READ  A
Loop    MOVER AREG, A
        SUB  AREG, ='1'
        BC  GT, Loop
        STOP
```

A DS
LTOR
G

Solution :

Label	Statement	Intermediate Code
	START 200	(AD,01) (C,200)
	READ A	(IS,09) (S,02)
Loop	MOVER AREG, A	(IS,04) (1) (S,01)
	SUB AREG, ='1'	(IS,02) (1) (L,01)
	BC GT, Loop	(IS,07) (4) (S,01)
	STOP	(IS,00)
A	DS	(DL,02) (C,01)
	LTORG	(DL,05)