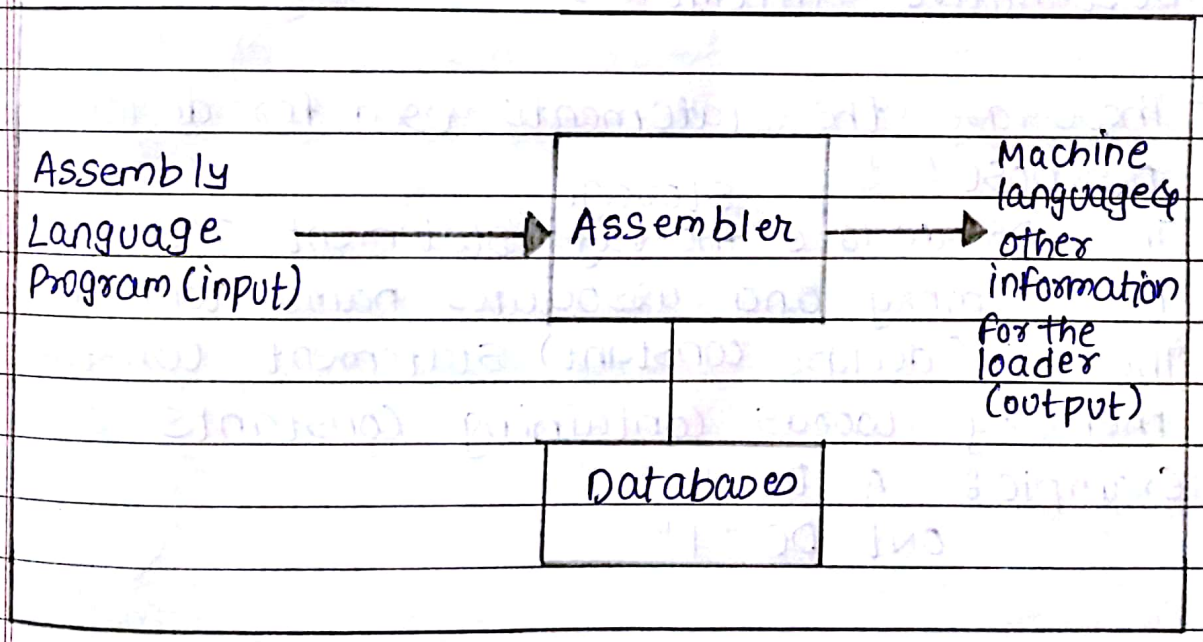
CHAPTER 2 : ASSEMBLERS

ASSEMBLER :

DEFINTION

An assembler is a program that accepts input as assembly language program and produces its machine language equivalent along with information for the loader.

DIAGRAM REPRESENTATION



Features of Assembly Language Programming [ALP]

I Language specification
II Statement in ALP
III Structure of ALP

Scanned by CamScanner

* TYPES OF ASSEMBLY LANGUAGE STATEMENTS

1.] IMPERATIVE STATEMENTS

- An imperative statement indicates an actions to be performed during the execution of the assembled program. Each imperative statement typically translates into one machine instruction
- In simple terms, these are the statements understood by the machine & executed by the machine.

Example: MOV A, B

2.] DECLARATIVE STATEMENTS

- These are the statements used for declaration purpose
- The DS (declare ~~shortage~~ storage) statement reserves areas for memory and associates names with them
- The DC (declare constant) statement constants memory words containing constants.

Example: A DS 1
ONE DC "1"

3.] ASSEMBLER DIRECTIVES

- These are the statements used to instruct the assembler to perform certain actions regarding how assembly of input program is to be performed

Example: START <constant>
END <operand spec.>

Scanned by CamScanner

Date
Page

* GENERAL DESIGN PROCEDURE OF ASSEMBLER

- Specify the problem
- Specify data structures
- Define format of data structures
- Specify Algorithm
- Look for modularity [capability of one program to be subdivided into independent program units]
- Repeat 1 through 5 on modules

* SYNTHESIS PHASE

The synthesis phase is concerned with the construction of target language statements which have the same meaning as a source statement. Synthesis phase uses the table to obtain the machine address with a name associated and machine opcode corresponding to the memonic

It performs two main activities
- memory Allocation
- Code Generation

* Types of Assembler

- Single Pass Assembler:

A single pass assembler scans the program only once and creates the equivalent binary program. The assembler substitute all of the symbolic instruction with machine code in one pass.

Scanned by CamScanner

2.] Multi Pass Assembler :
Does the work in two pass - Resolves the forward references

- First pass - Scans the code
  Validates the tokens
  Creates a symbol table
- Second pass - Solves forward references
  Converts the code to the machine code

Statement of Problem

- The assembler must do the following

  1.] Generate Instructions
  - Evaluate the mnemonic in the operation field
  - Evaluate sub fields

  2.] Process Pseudo ops

Scanned by CamScanner

Date
Page

ASSEMBLER PASS I

Step 1: Specify the problem

Pass 1: Define symbols & literals

1.) Determine length of machine instruction [MOT]
2.) Keep track of Location Counter [LC]
3.) Remember values of symbols [ST]
4.) Process some pseudo ops [EQU, DS etc] [POT]
5.) Remember literals [LT]

Step 2: Specify Data Structure

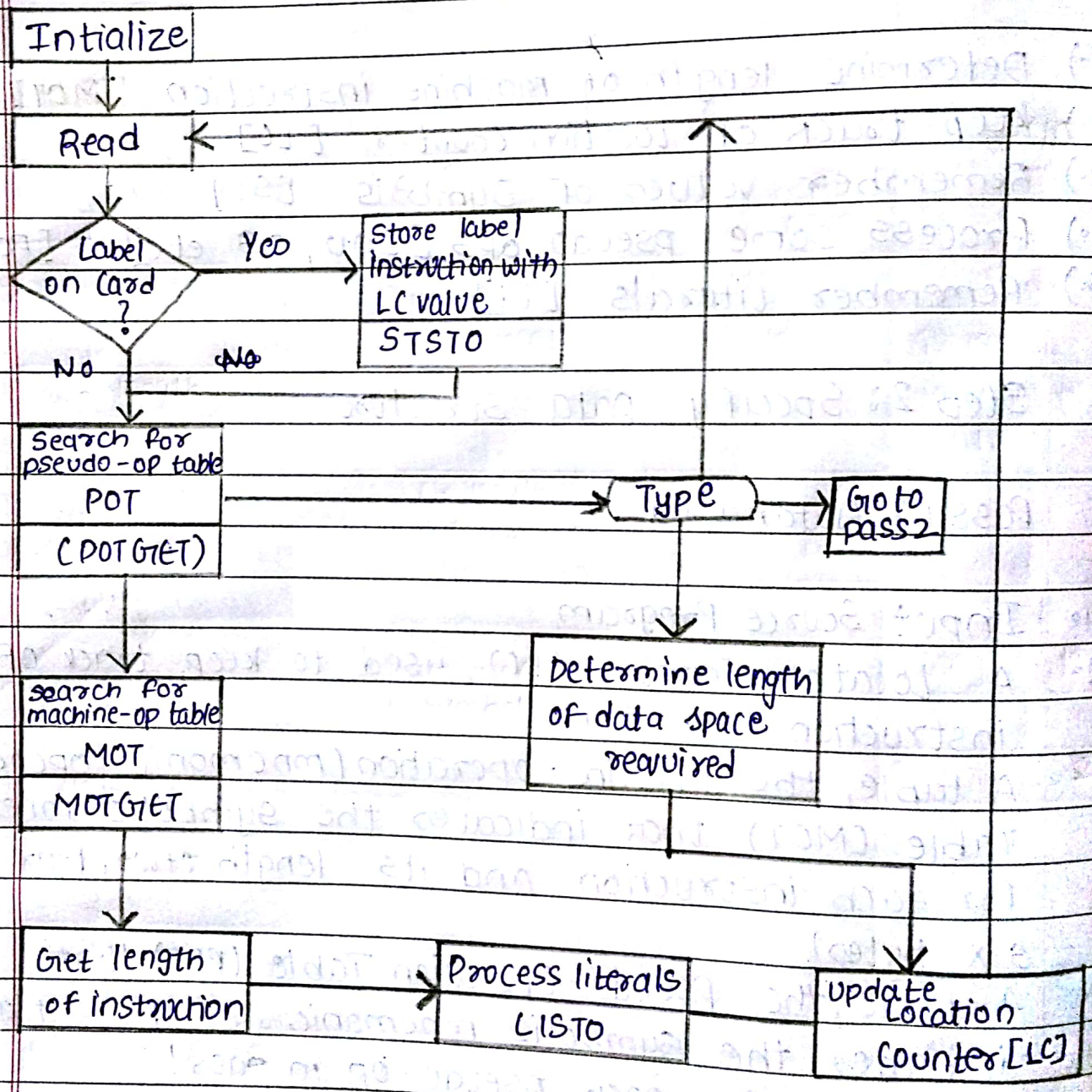
Pass 1: Databases

1. Input Source Program
2. A Location Counter (LC), used to keep track of each instruction's location
3. A table, the Machine-operation (mnemonic opcode) Table (MOT) that indicates the symbolic mnemonic, for each instruction and its length (two, four or six bytes)
4. A table, the Pseudo-Operation Table (POT) that indicates the symbolic mnemonic and the action to be taken for each pseudo-op in pass 1
5. A table, Symbol Table (ST) that is used to store each label and its corresponding value
6. A table, the literal table (LT) that is used to store each literal encountered and its corresponding assignment locations

Scanned by CamScanner

OVERVIEW OF ASSEMBLER PASS I

FLOW CHART



Scanned by CamScanner

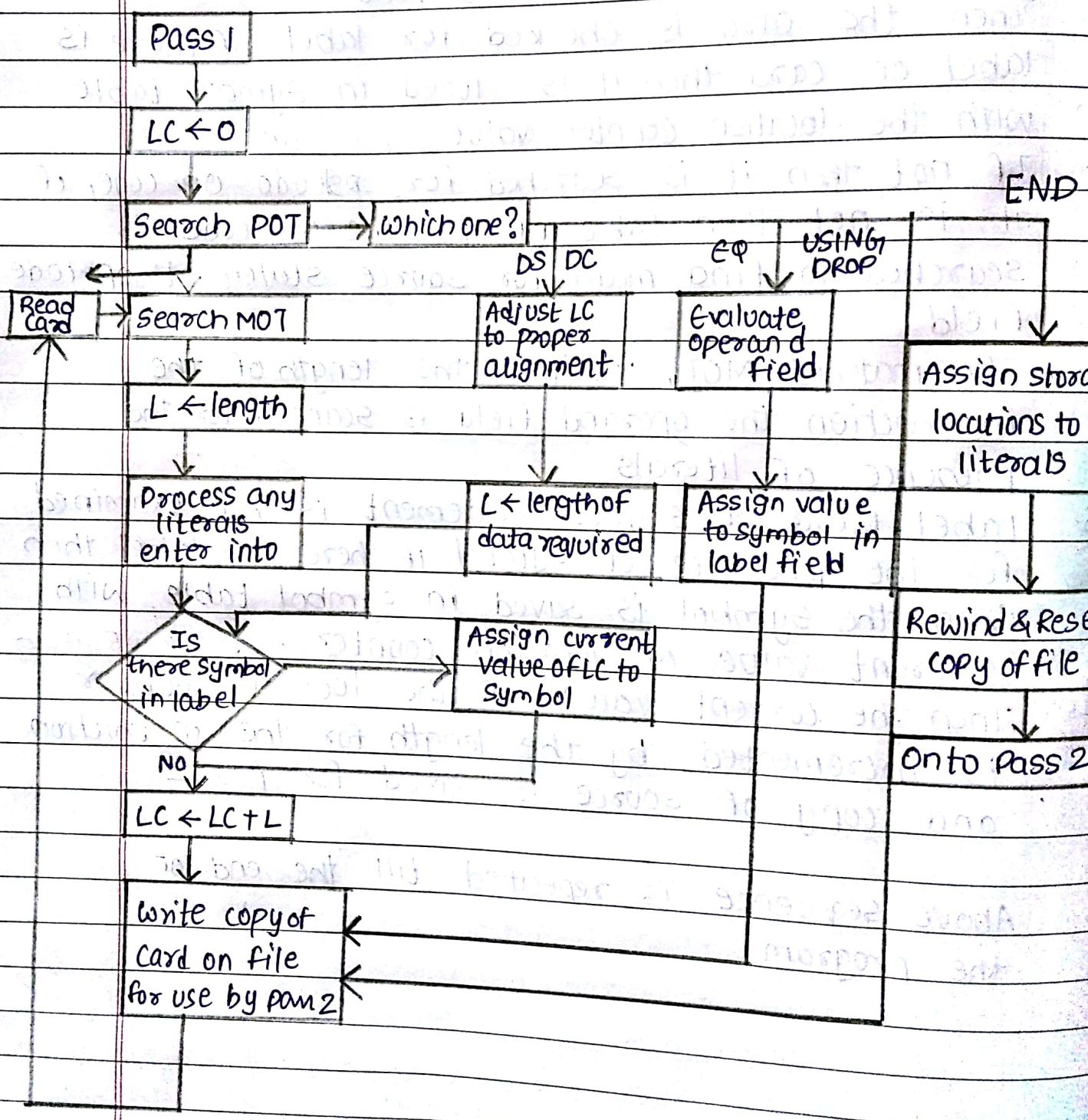
PROCESS

1. The Location counter is set to zero first location where relative address is zero
2. Then the source statement is read
3. Then the card is checked for label, if there is label on card then it is stored in symbol table with the location counter value
4. If not then it is searched for pseudo-op code, if it is not then table of machine-op code is searched to find match for source statement op-code field
5. The matched MOT, specifies the length of the instruction the operand field is scanned for the presence of literals
6. Label field of source statement is then examined for the presence of symbol if there is label then then the symbol is saved in symbol table with current value of location counter
7. Then the current value of the location counter is incremented by the length for the instruction and copy of source is saved for pass 2

Above sequence is repeated till the end of the program.

Scanned by CamScanner

DETAILED EXPLANATION OF ASSEMBLER PASS I



Scanned by CamScanner

Date
Page

PROCESS

1. The location counter is set to first location where relative address is zero
2. Then the source statement is read
3. Then it is searched for pseudo-op table if it is found then it is checked for it's type
4. If the POT is DS/DC then LC is adjusted to proper alignment or if it is EQ then using DROP evaluate operand and assign value to symbol in label field
5. If there is no pseudo-op then it is searched for machine-op code is searched to find match for source statement op-code field and it's length is determined
6. The matched MOT is then scanned for the presence of literals and if label is found then the symbol is saved in the symbol table with current value of location counter

And the above sequence is repeated till the end of program.

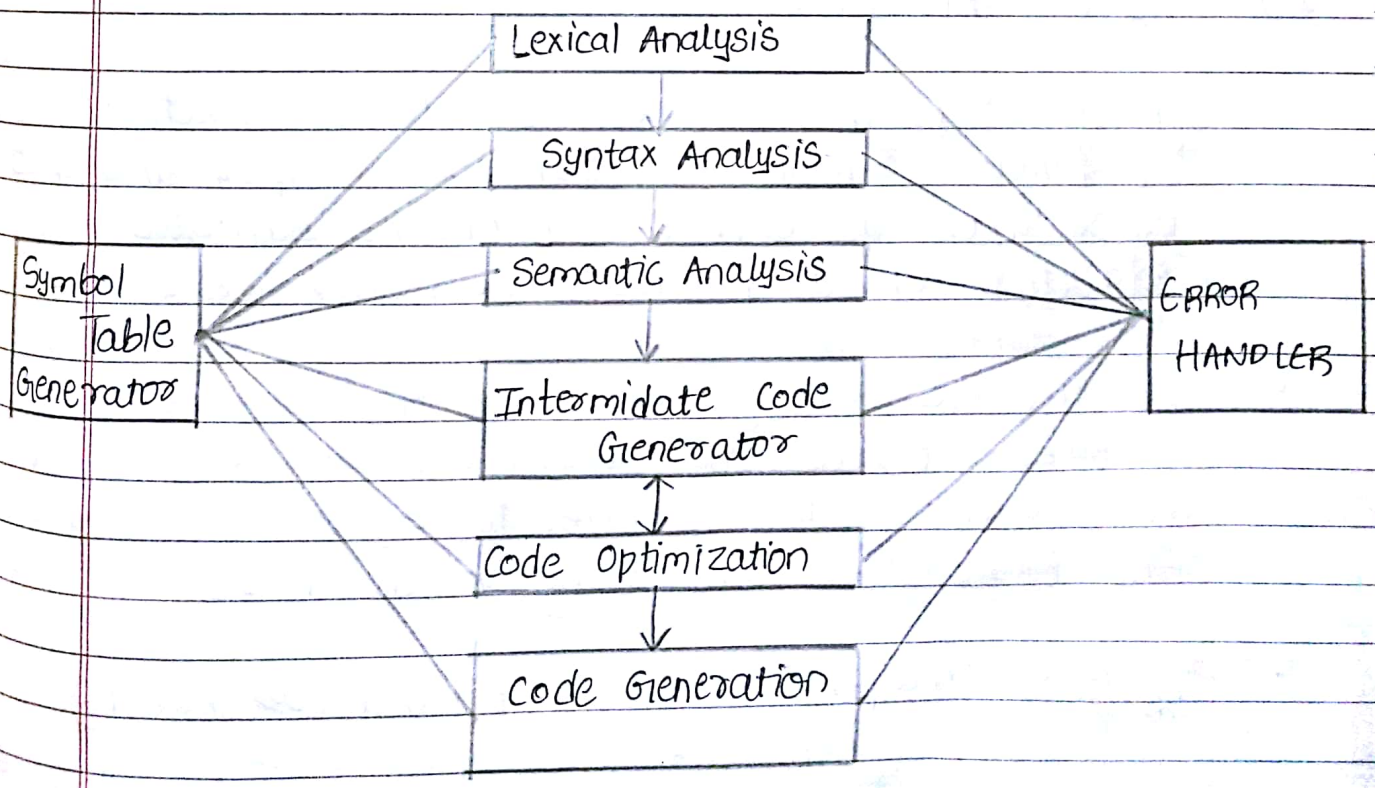
Scanned by CamScanner

COMPILERS

DEFINITION:

- A complier is a program that reads any source program and translates it into an equivalent program in another or target language.
- As an important part of translation process, the complier reports to its user the presence of the errors in source Program.

BLOCK DIAGRAM OF COMPILERS



Scanned by CamScanner

I) LEXICAL ANALYSIS

- The lexical analyzer is the first phase of complier. Its main task is to read the i/p characters and produces o/p a sequence of tokens that the praser uses for syntax analysis

- In this phase, stripping out of user comments, white spaces, tab, new line character and blanks from the program are being taken care of.

- Another task is correlating error message from the complier with the source program.

II) Syntax Analysis:

- In the second phase of Compiler Construction. It takes stream of tokens as input and gives parse tree as output. It checks syntatic structure of a program & checks for errors.

- It involves the grouping of the source program tokens into grammatical phases that are used by the complier to synthesize output. The Parse Trees are being made here.

- Syntax Analysis is also termed as Parsing

Scanned by CamScanner

III) SEMANTIC ANALYSIS

- It checks for the semantic errors and gathers type info, uses the hierarchical structure determined by the previous phase to identify the operators and the operands of expressions & statements

- Important component here is type checking, where the complier check that each operator has operands that are allowed by the source language specification.

- In other words, this phase adds meaning to various tokens, identifiers, operators, expressions etc.

IV) INTERMIDATE CODE GENERATOR

- In this many compliers generate an explicit low level or machine like intermidate representation

for eg:-

t_1 = int to float (60)
$t2$ = id3 * t3
$t3$ = id2 + t2
id1 = t3

this type of representation should have two important properties. ~~It~~ It should be easy to produce & easy to translate into target m/c

Types of IR are:

1) Structural 2) Linear 3) Hybrid

Scanned by CamScanner

V CODE OPTIMIZATION

- This phase attempts to improve the intermidate code so that faster running machine code will result
- The basic idea of this field is to improve the execution efficiency of the program.
  For eg:-

$$t_1 = id3 * 60.0$$
$$id1 = id2 + t1$$

VI CODE GENERATION

- The final phase of compilation process is the generation of target code, consisting normally of relesable machine code or assembly code.

- Memory locations are selected for each of the variable used by the program, then IR are translated into a sequence of m/c instn that perform the same task

- To avoid redundant codes, the code generator must keep track of the register contents at runtime

- A good code generator would attempt to use these registers as efficiently as possible. This aspect of code generation is called register allocation.

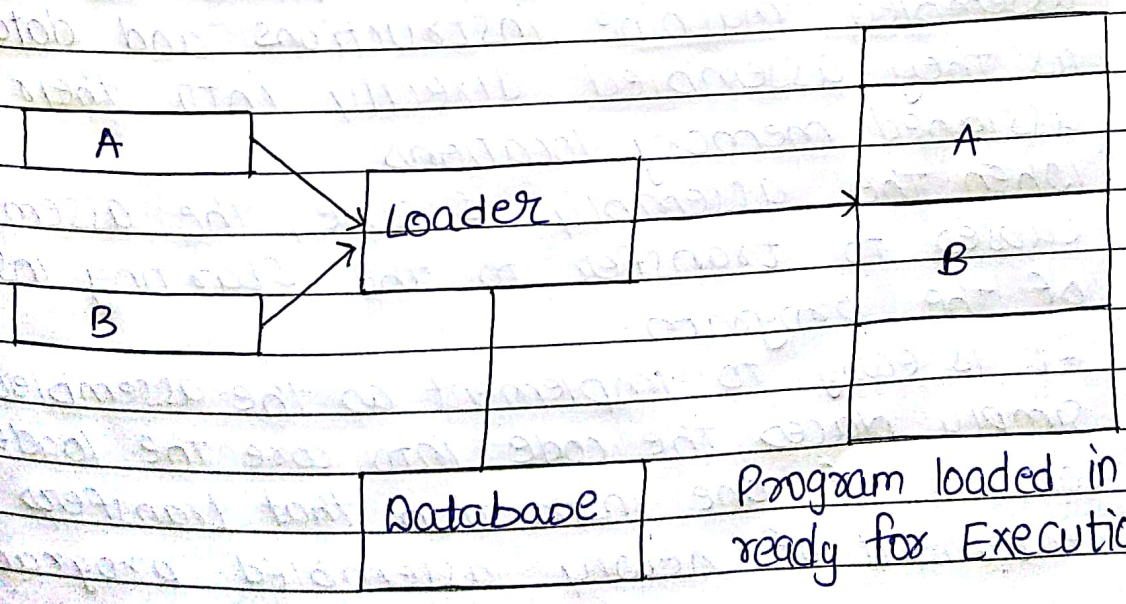
Scanned by CamScanner

CHAPTER 3 : LOADERS AND LINKERS

- A loader is a program/software which accepts the object / target code by assembler and compiler and intiates the execution

Four basic function of a Loader

- Allocate space in memory for the programs (Allocation)
- Resolve symbolic references between object program (Linking)
- Adjust all address dependent location such as address constants to correspond to the allocated space (Relocation)
- Physically place the machine instructions and data into the memory (Loading)

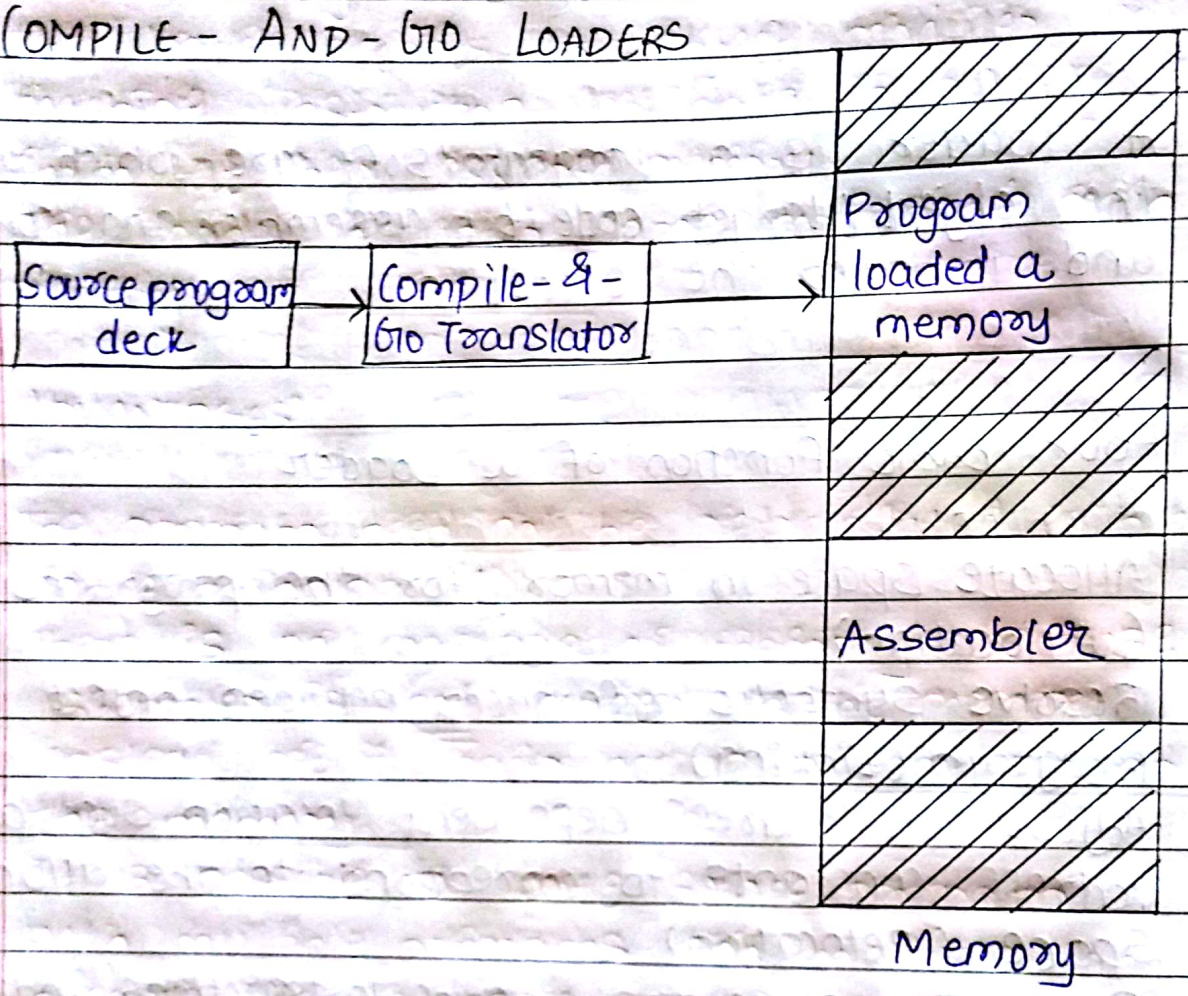


Program loaded in memory ready for Execution.

Scanned by CamScanner

LOADER SCHEMES

1] COMPILE - AND - GO LOADERS



- The assembler is allowed to run in one part of the memory and place the assembled machine instructions and data as they assembled directly into their assigned memory locations
- When the assembly is done, the assembler caused to transfer to the starting instruction of the program.
- It is easy to implement as the assembler simply places the code into core the loader consists of one instruction that transfers to the statement of newly assembled program.

Scanned by CamScanner

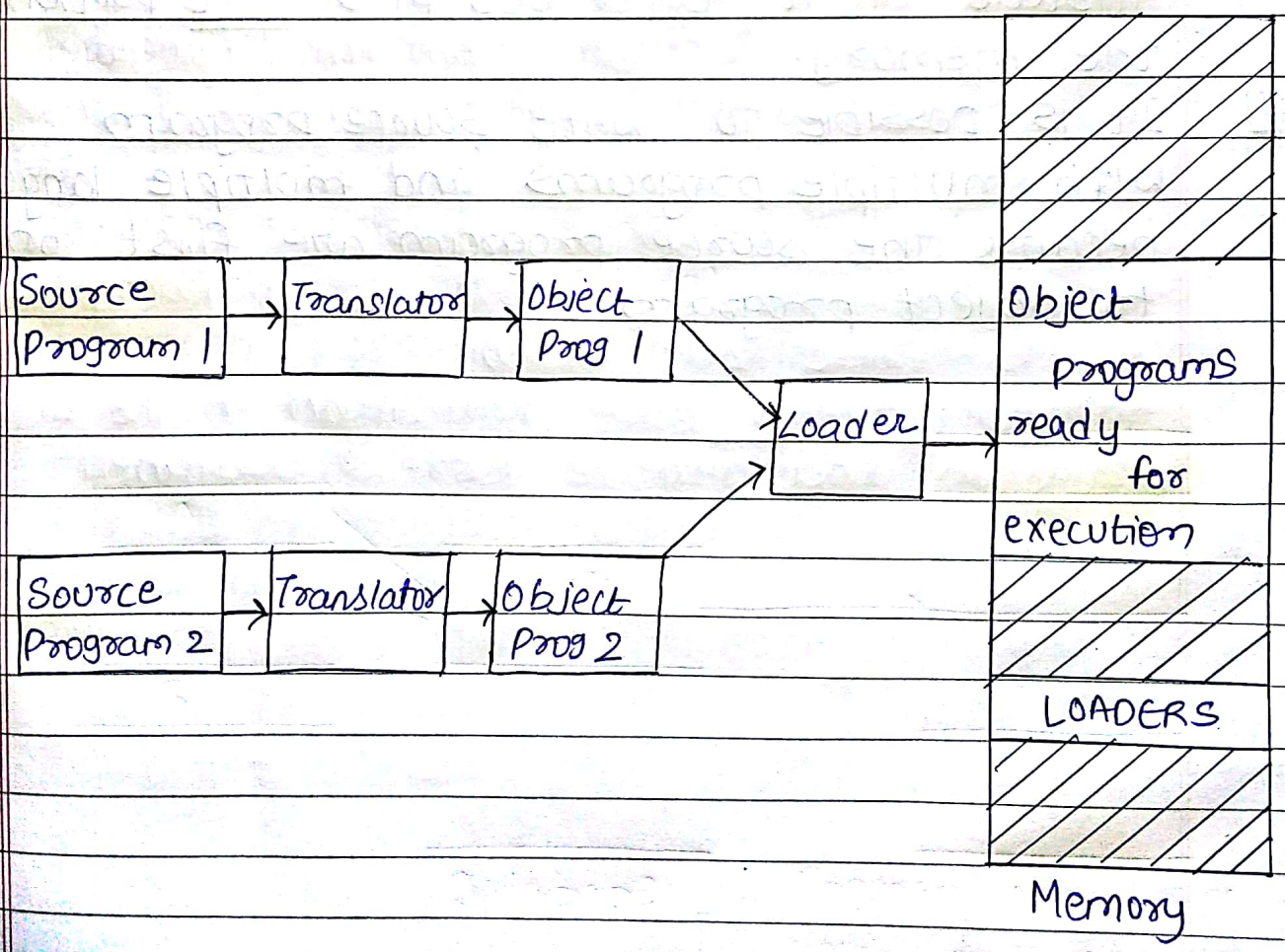
DISADVANTAGES

i) A portion of memory is always occupied by assembler. That portion is unavailable to any other object program

ii) User's program have to reassemble every time it runs.

iii) If source program contains subroutine written in different languages, then different assemblers have to be stored in memory (i.e wastage of memory).

2.] GENERAL LOADER SCHEMES



Scanned by CamScanner

- The source program is converted to object program by some translator (assembler)
- The loader accepts the object modules and puts machine instruction and data in an executable form at their assigned memory
- The loader occupies some portion of main memory.

ADVANTAGES

1] The program need not be retranslated each time while running it.
2] There is no wastage of memory, because assembler is not placed in the memory, instead of it, loader occupies some portion of the memory.
3] It is possible to write source program with multiple programs and multiple languages, because the source program are first converted to object program.

Scanned by CamScanner

SUBROUTINE LINKAGE

- The problem : A MAIN program A wishes to transfer to sub-program B. However the assembler does not know this symbol and declare it is an undefined symbol unless a special mechanism has been provided.
- This mechanism has been implemented with a direct-linking or relocating loader.
- The assembler pseudo-op EXTRN followed by a list of symbol indicates that these symbols are defined in other programs.
- Corresponding if a symbol is defined in one program & referenced in others, we insert it into symbol list following the ENTRY
- Assembler will also inform the loader that these symbols may be referenced by other programs.

RELOCATING LOADERS

- To avoid possible re-assembling of all subroutines when a single subroutine is changed and to perform the task of allocation and linking for the programmer, this class of loader was introduced
- The assembler assembles each procedure segment independently and passes on to the loader the text information as to relocation and their segment references.

Scanned by CamScanner

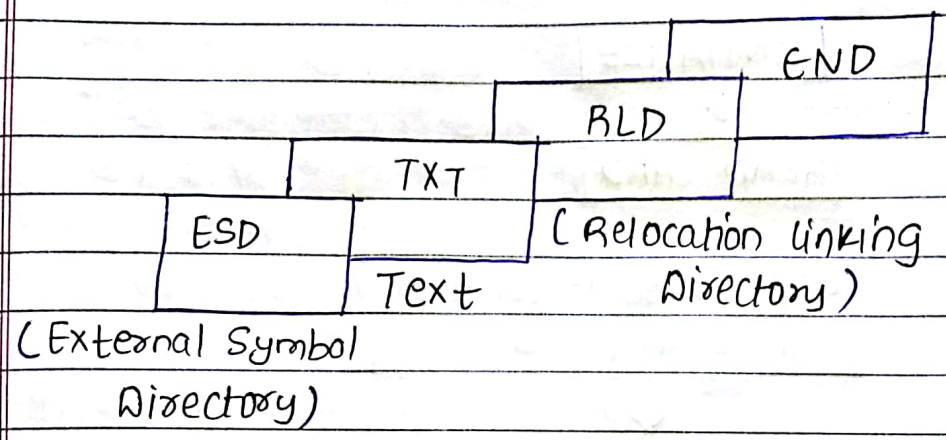
DIRECT LINKING LOADER

- It is a general relocatable loader it has has an advantage of allowing the programmer multiple procedure segments & multiple data segments and giving him complete freedom
- In referencing data or instructions content in other segments this provides flexible intersegment referencing and accessing ability, while at the same time allowing independent translation of programs
- The assembler must give the loader the following information with each procedure or data segment.
  i) The length of the segment
  ii) The list of all the symbols in the segment that may be referenced by other segments & their relative location within the segment.
  iii) A list of all symbol not defined in segment but referenced in the segment.
  iv) Information as to where address constants are located in the segment and a description of how to revise their values
  v) The machine code translation of the source program and relative address assigned.

Scanned by CamScanner

DESIGN OF DIRECT LINKING LOADER

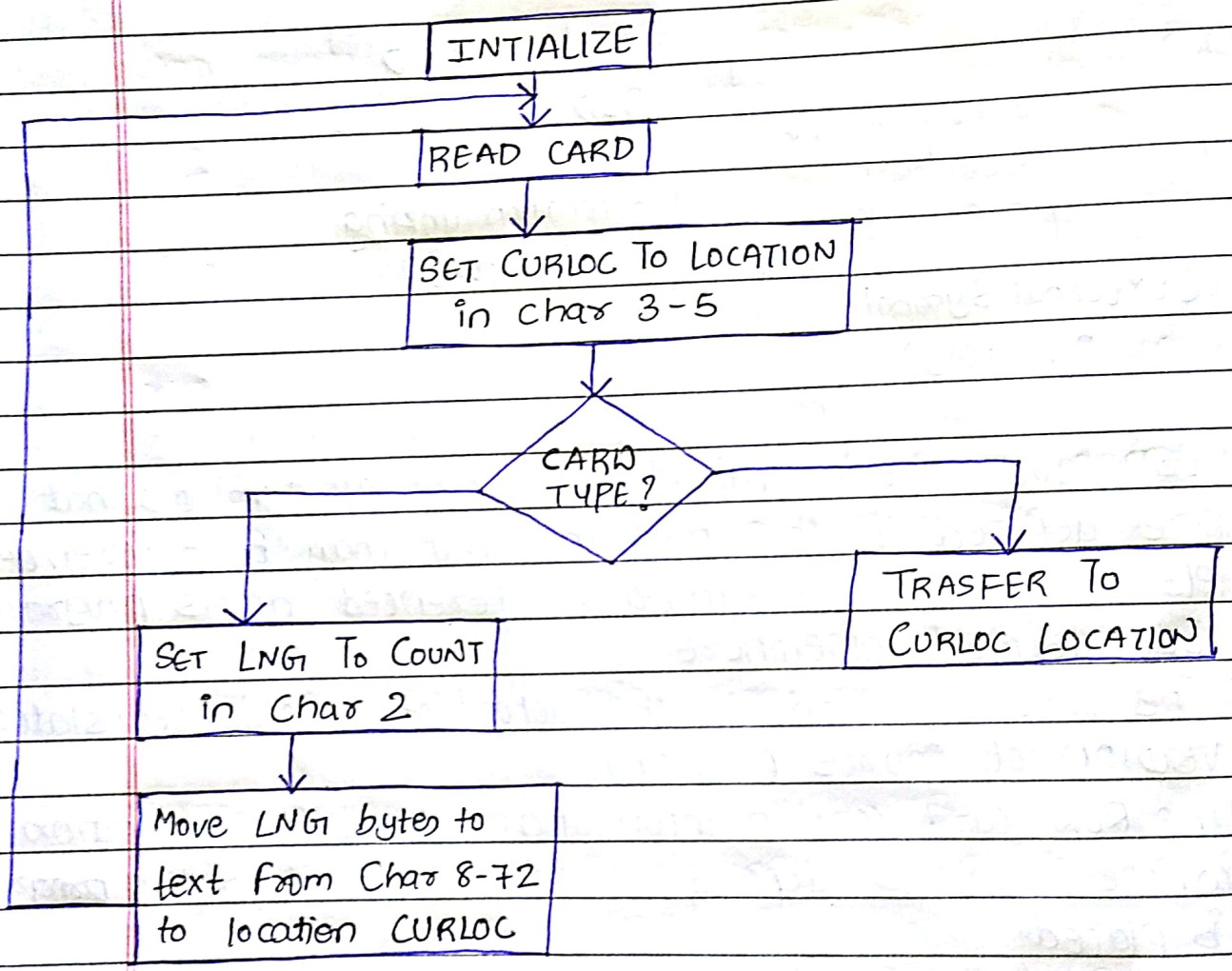
Object deck for DLL



- ESD card contain information about all symbols that are defined in this program that may be referenced elsewhere, and all symbols referenced in this program but defined elsewhere.
- The text card contain the actual object code translated version of source program.
- The RLD card contain information about those location whose contents depend on address at which the program is placed.
- The END card indicates the end of object deck & specifies the starting address for execution if the assembled routine is the "Main Program"
- RLD card also contains location of each constant that needs to be changed due to relocation
- By what it has to be changed & operation to be performed.

Scanned by CamScanner

ABSOLUTE LOADER



Scanned by CamScanner

DESIGN OF ABSOLUTE LOADER

- With an absolute loading scheme the programmer & assembler perform the task of allocation, relocation & linking
- The loader has to read cards of the object deck & move the text on the cards into the absolute locations specified by the assembler
- There are 2 types of information that the object deck must communicate from the assembler to the loader
  - It must convey the machine instruction along with assigned core locations.
  - It must convey the entry point of the program which is where the loader is to transfer control when all instructions are loaded
- Assuming that this information is transmitted on cards
- Instruction are stored on cards as one core byte per column. 8-bit means 356 possible contents
- Information is stored in core as 80 contiguous bytes.

Scanned by CamScanner

CARD FORMATS FOR AN ABSOLUTE LOADER

TEXT CARDS (for instruction and data)

CARD COLUMN	CONTENTS
1	card type=0 (for text identifier)
2	count of number of bytes (1 byte per column)
3-5	address at which data on card is to be put
6-7	empty (could be used for validating checking)
8-72	instruction & data to be loaded
73-80	card sequence number

TRANSFER CARDS (to hold entry point to program)

CARD COLUMN	CONTENTS
1	card type=1 (transfer card identifier)
2	count=0
3-5	address of entry point
6-72	empty
73-80	card sequence number.

Scanned by CamScanner