Unit - I

Page No.
Date

8hrs
credit-3

Introduction

CO → CO1, CO2, CO3.

- Introduction to system Programming
- Need of system Programming
- Software Hierarchy
- Types of Software
- System Software and Application software
- Machine structure.

Evolution of Components of System Programming
- Text Editors
- Assemblers
- Macros
- Compiler
- Interpreter
- Loader
- Linker
- Debugger
- Device drivers
- Operating System.

Elements of Assembly Language Programming
- Assembly Language statements
- Benefits of Assembly Language
- A simple assembly schema.
- Pass structure of Assembler.

Design of two-Pass assembler.
- Processing of declaration statements
- Assembly directives and imperative statements
- Advanced assembly directives
- Intermediate code forms.
- Pass-I and Pass-II of two pass assembler.

Page No.
Date

Introduction to System programming :-

[Ref: SPOS by

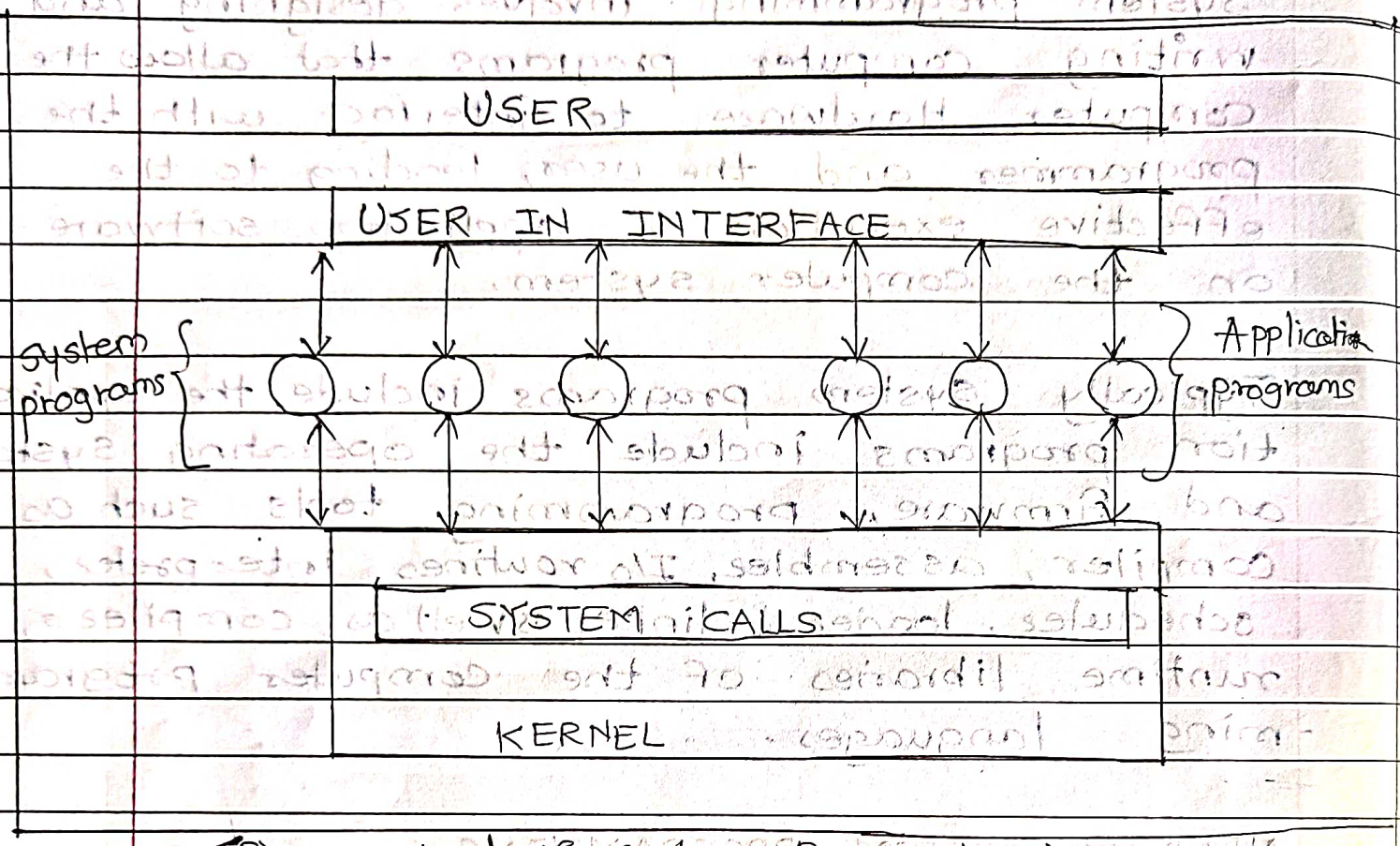
System programming involves designing and writing computer programs that allow the computer Hardware to interface with the programmer and the user, leading to the effective execution of application software on the computer system.

Typically system programs include the application programs include the operating system and firmware, programming tools such as compiler, assemblee, I/O routines, interpreter, schedulee, loader & linker as well as compilers, runtime libraries of the computer programming languages.

NEED OF SYSTEM PROGRAMMING :

- System programming provide environment where programs can be developed & executed
- In the simplest sense, system programs also provide a bridge between user Interface and system calls.
- In reality, they are much more complex, for example, a compiler is a complex system program.
- The system program serves as a part of the operating system. It traditionally lies between user Interface & system calls.
- The user view of the system is actually defined by system programs & not system calls

because that is what they interact with and System programs are closer to user Interface



<Figure: Need of System Programming>

In the above image, system programs as well as application programs form a bridge between the user Interface & system calls, so, from the user view the operating system observed is actually the system programs and not the system calls.

Need of System Programming

1. To acheive efficient performance of the system.
2. To make effective execution of general user programs.
3. To make effective utilization of human Resource
4. To make available new, better facilities.

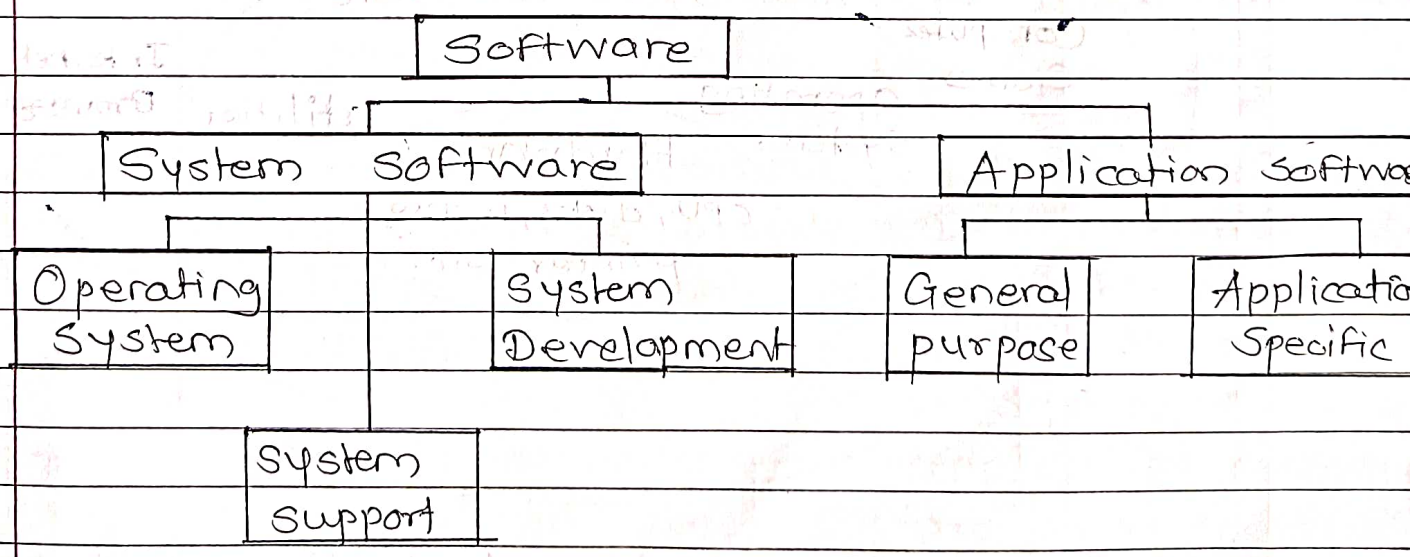
Page No.
Date

Software Hierarchy:

- A software Hierarchy is a combination of product, version and release (or Feature) that represent an item of software in a database or knowledge base. The product is the root of the hierarchy.

- software
1. System software
- Operating system
- System support
- System Development

2. Application software
- General purpose
- Application Specific



< Software Hierarchy >

Page No.
Date / /

What is Software?

- Software is collection of many programs.

Two types of software

System Software ; These programs assist general user application programs.

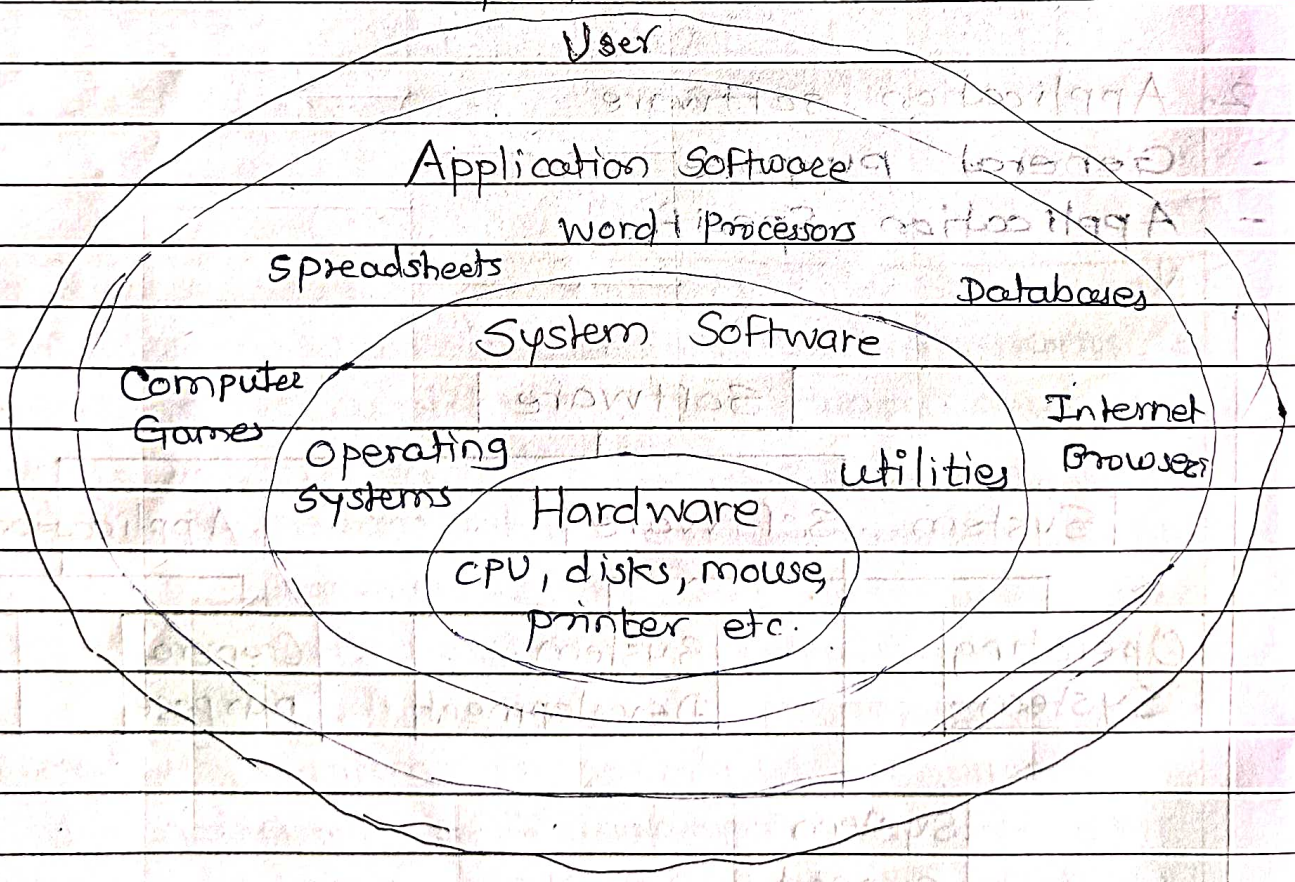
Eg:

Operating System, Assembler etc.

Application Software : These are the software developed for the specific goal.

Eg:

Media Player, Adobe Reader etc.



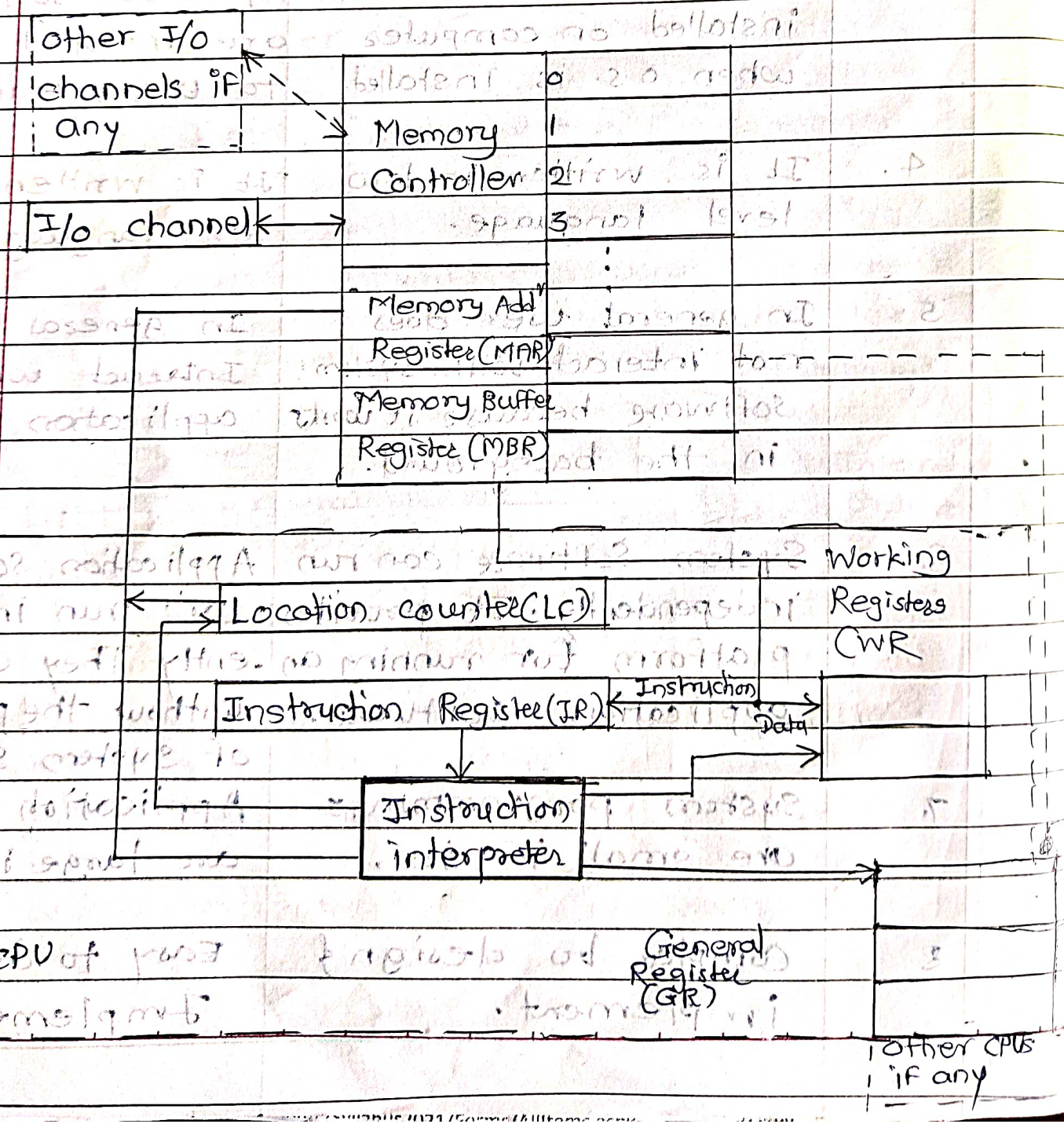
- System Software Vs Application Software

Page No.
Date

Sr. No	System Software	Application Software
1.	System software is used for operating computer Hardware.	Application software is used by user to perform specific task.
2.	It is general purpose software	It is specific purpose software.
3.	System softwares are installed on computer when O.S. is installed.	Application softwares are installed according to user's requirement
4.	It is written in low level language.	It is written in high level language
5.	In general user does not interact with system software because it works in the background.	In general user interact with the application software
6.	System software can run independently. It provides platform for running an application software.	Application software can't run independently. They can't run without the presence of system software
7.	System programmings are small in size.	Application programs are large in size.
8.	Complex to design & implement.	Easy to design & Implement.

9.	Some examples of System Software are compiler, assembler, debugger, drivers etc.	Some examples of Application software are word processor, web browser, media player etc.

- Machine Structure → The Basic model was proposed by John Von Neumann.



Page No.
Date

The above structure consist of

1. Instruction interpreter
2. Location counter
3. Instruction Registe
4. Working Register
5. General Registers

- The instruction interpreter Hardware is basically a group of circuits that performs the operation specified by the instruction fetched from the memory.
- The location counter can also be called as program Instruction counter simply points to the current instruction being executed.
- The working registee are often called the "scratch pads" because they are used to store temprary values while calculations is in proce
- The CPU interface with memory through MAR & MBR
- MAR (Memory Address Register): contains address of memory location (to be read or store into)
- MBR (memory Buffer Register)- contains a copy of address specified by MAR memory controller is used to transfer data between MBR & memory location specified by MAR.
- The Role of I/O channel is to input or output information from memory.
- Instruction Register - A copy of content of the L.C is stored in IR (Inst^n Reg).

Page No.
Date / /

Evolution of Component of System Programming

System program is a type of computer program that is designed to run a computer's hardware & application programs.
If we think of computer system as a layered model, the system software is the interface between the hardware & user applications.
components are -
- Text Editors
- Assembler
- Loader
- Macro processors
- Compilers
- Debuggers
- Device drivers
- Operating Systems
- Linker
- Interpreter

Text Editors -

- Editor is a computer program that allows a user to create & revise a document
- A Text Editor is a type of program used for editing plain text files.
- With the help of text editor you can write your program (eg. C program or Java program)
- Text Editor's example is Notepad.

Page No.
Date

Types of text Editors.

1. Line Editor:

This code editor edits the file line by line. You cannot work on a stream of lines using the line Editor.
eg - teleprinter.

2. Stream Editor:

In this type of editors, the file is treated as continuous flow or sequence of charecters instead of line numbers, which means here you can type paragraph.
eg - Sed editor in UNIX.

3. Screen Editors:

In this type of editors, the user is able to see the cursor on the screen and can make a copy, cut, paste operation easily. It is very easy to use mouse pointer.
eg - vi, emacs, Notepad.

4. Word Processor -

Overcoming the limitation of screen editors, it allows to use some format to insert images, files, videos, use Font size, style features. It majorly focus on Natural language.

Page No.
Date

Loaders -

- A loader is a program that takes object code as input and prepares them for execution.
- It initiate the execution.
- Functions:
  1. Allocation
  2. Linking
  3. Relocation
  4. Loading.

Functions of Loader

1. Allocation: loader allocates space for program in main memory.

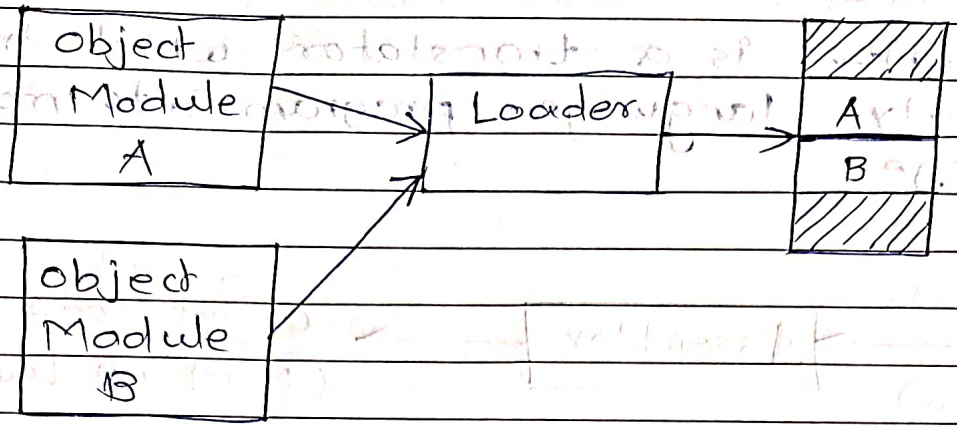
2. Linking: which combines two or more seperate object programs (by linker)

3. Relocation: which modifies the object program so that it can be loaded at an address different from the location originally specified - (Linking Loader) and data into main memory.

4. Loading: which allocates memory location and brings the object program into memory for execution - (Loader)

General Loading Scheme.

- Program combines A and B are leaded in memory after linking. It is ready for execution



General Loading scheme.

RELOCATION

Case 1: Address assigned to Prog. A & F1() when they translated to memory.

Drawback -

a lot of storage area is wased.

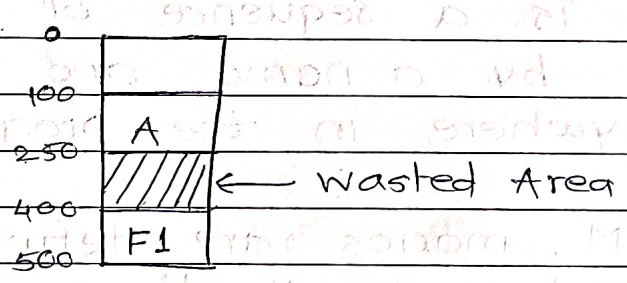


Fig: General Loading scheme.

RELOCATION

Case-II : These two module cannot co-exist at same storage locations.

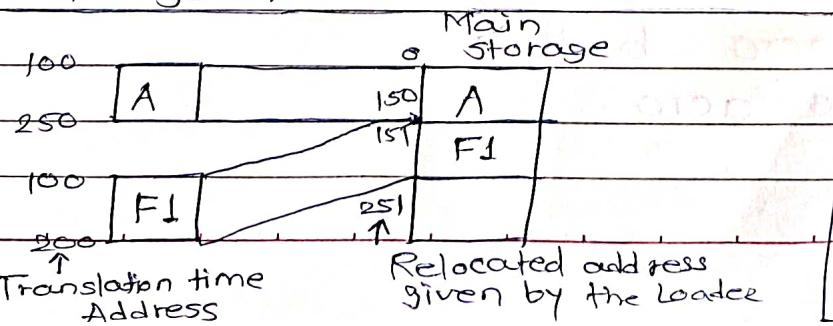
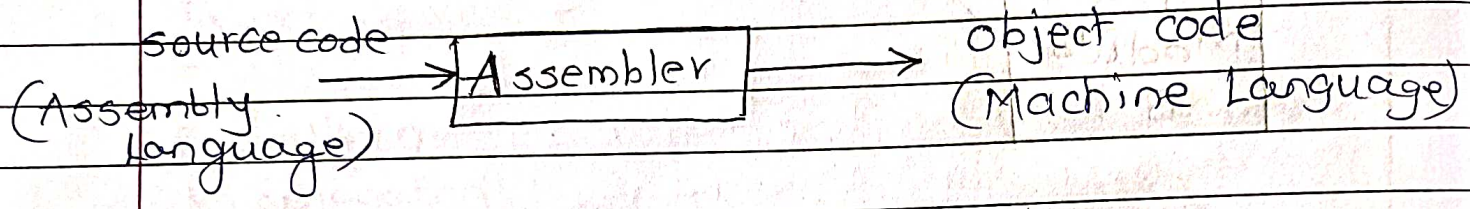


Fig:- Relocation to avoid address conflict or storage waste

Page No.
Date

Assembler -

- Assembler is a translator which translate assembly language program into machine language.



Macro

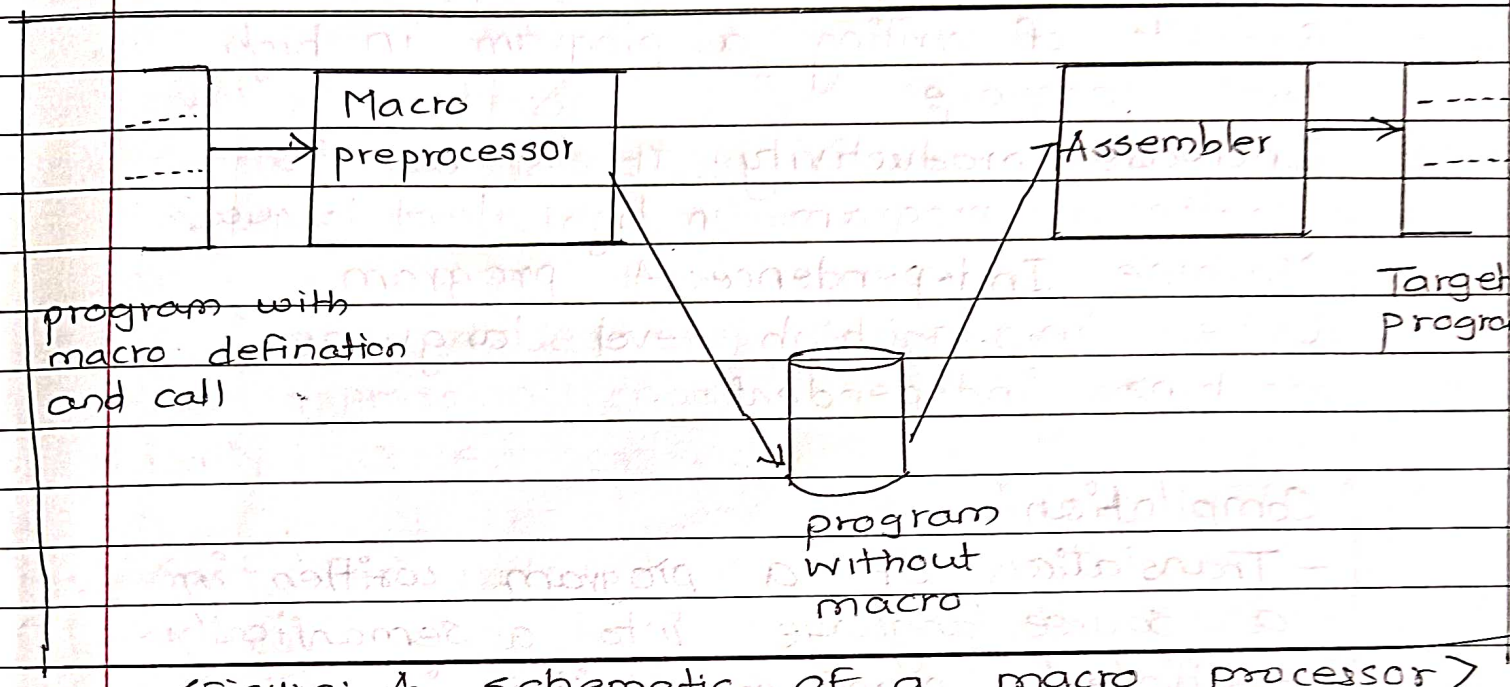
- Macro allows a sequence of source language code to be defined once & referred many times.

"A macro is a sequence of Instructions assigned by a name and could be used anywhere in the program".

- In NASM, macros are defined with %macro and %endmacro directives.
- The macro begins with %macro directives and ends with %endmacro directives
- Syntax :

```
%macro macro_name number_of_params.
<macro body>
%endmacro.
```

- A macro processor takes a source with macro defination and macro calls and replaces each macro call with its body.



<Figure: A schematic of a macro processor>

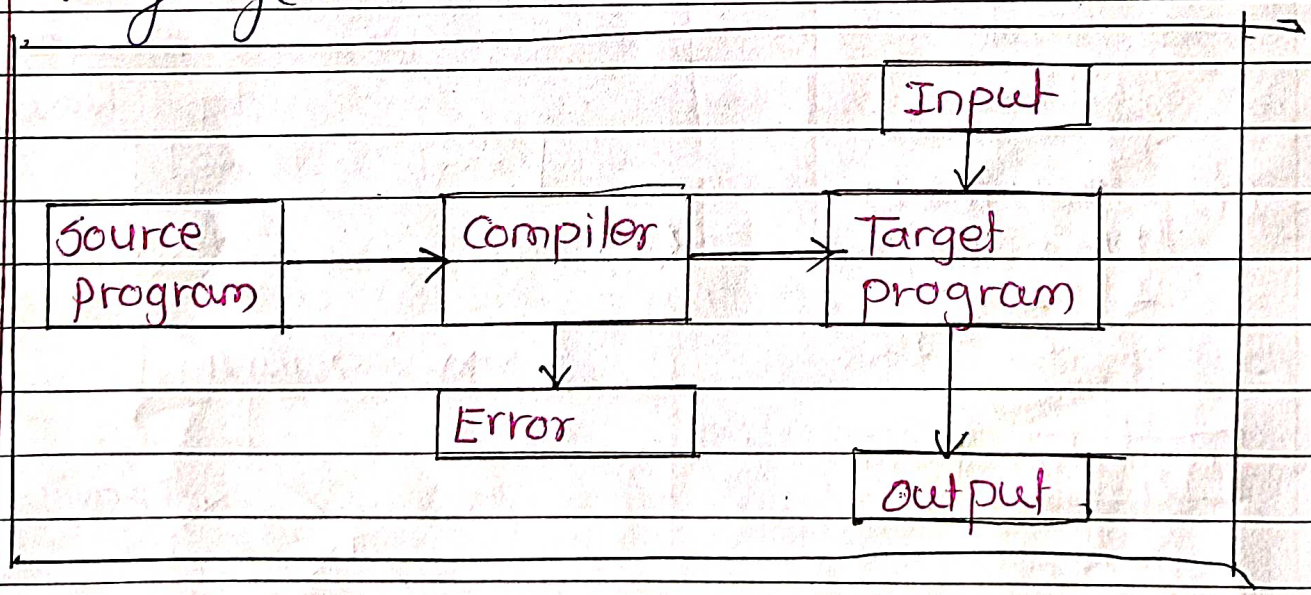
Page No.
Date / /

Compiler :-

- Compiler is a translator which converts the high level language into low level language.
- Benefits of writing a program in high level language:
- Increase productivity: It is easy to write a program in high level language
- Machine Independence: A program written in a high level language is machine independent.

"Compilation"

- Translation of a program written in a source language into a semantically equivalent program written in a target language.



phases of Compiler

- Lexical Analysis:

Lexical analyzer phase is a first phase of Compilation process. It take sourcecode

Page No.
Date

as input. It reads source program one char-
-ecter at time and converts it into meaning-
ful lexemes. Lexical analyzer represents these
Lexemes in the form of token.

- Syntax Analysis:
Syntax analysis is a second phase of compilation
process. It takes token as input and genera-
-tes a parse tree as output. In syntax
analysis phase, the parse ~~chekc~~ checks that
the expression made by the tokens is syntac
-ically correct or not.

- ~~Sym~~ Semantic Analysis:
Semantic analysis is third phase of
Compilation process. It checks whether the
parse tree follows the rule of language.
Semantic analyzer keeps track of identifiers,
their types & expressions. The output of semantic
analysis phase is the annotation tree syntax.

- Intermediate code Generation:-
In the intermediate code generation, compiler
generates the source code into the intermediat
code. Intermediate code is generated between
the high level language and the machine
language. The Intermediate code should be
generated in such a way that you can
easily translate it into the target machine
code.

Page No.
Date

- **Code Optimization -**
Code optimization is optional phase. It is used to improve the intermediate code so that the output of program could run faster and take less space. It removes unnecessary lines of the code & arranges the sequence of statement in order to speed up the program execution.

- **Code Generation -**
Code generation is final stage of compilation process. It takes intermediate / optimized intermediate code as input & maps it to the target machine language. Code generator translate the intermediate code into the machine code of specified computer.

eg - $c = a + b * 5;$

↓

Lexical analyzer

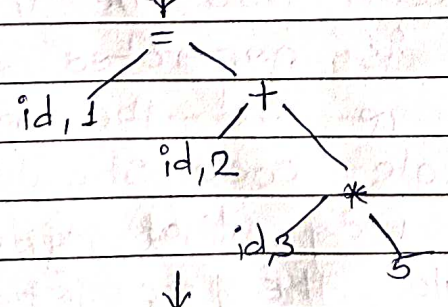
↓

<id,1><=><id,2><+><id,3><*><5>

↓

Syntax analyzer

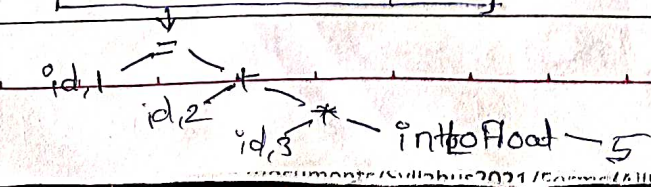
↓



↓

Semantic analyzer

↓



↓

Intermediate code Generator

↓

$t1 = \text{inttofloat}(5)$
$t2 = id_3 * t1$
$t3 = id_2 + t2$
$id1 = t3$

↓

Code Optimizer

↓

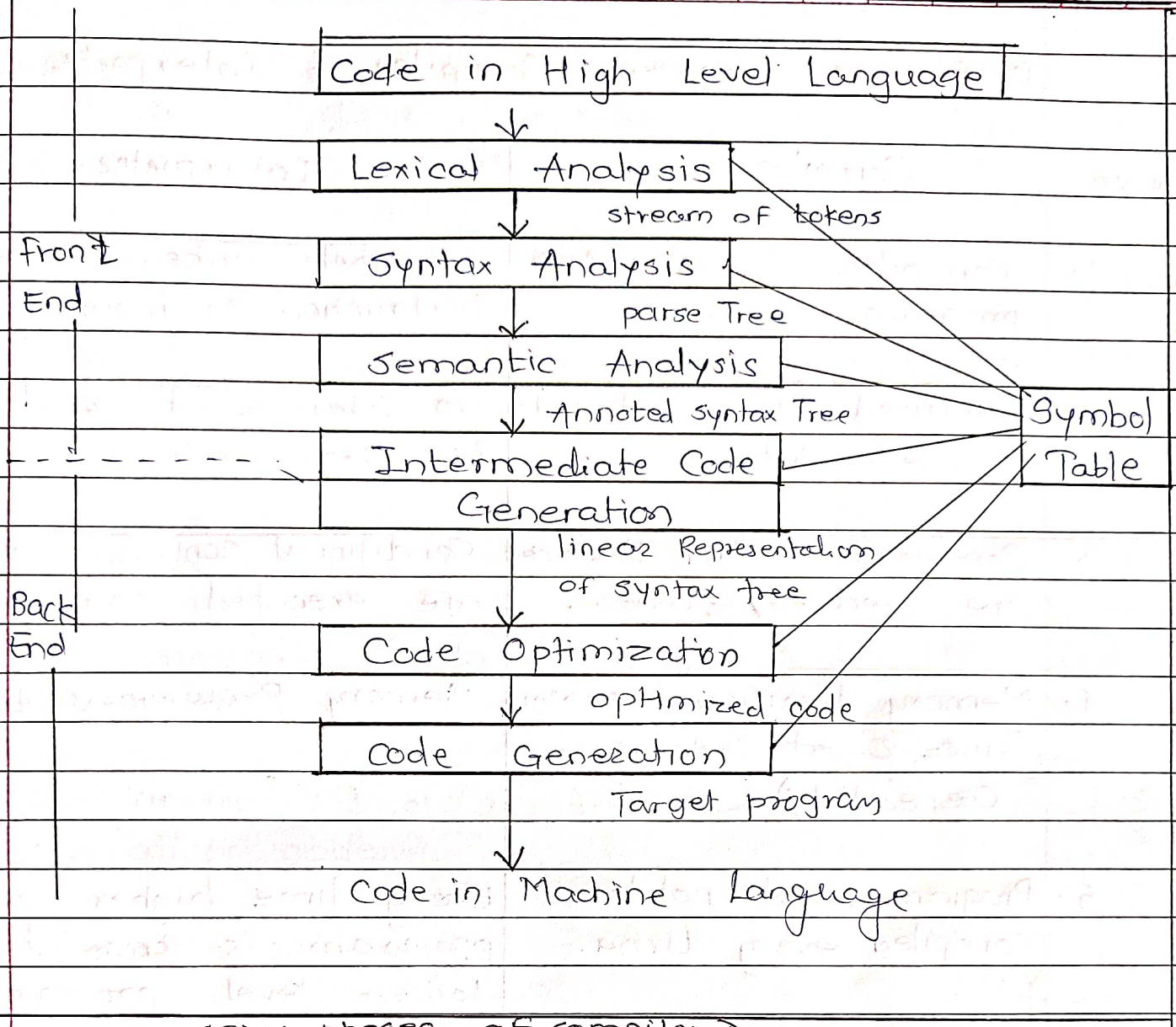
$t1 = id_3 * (5)$
$id1 = id_2 * t1$

↓

Code Generator

↓

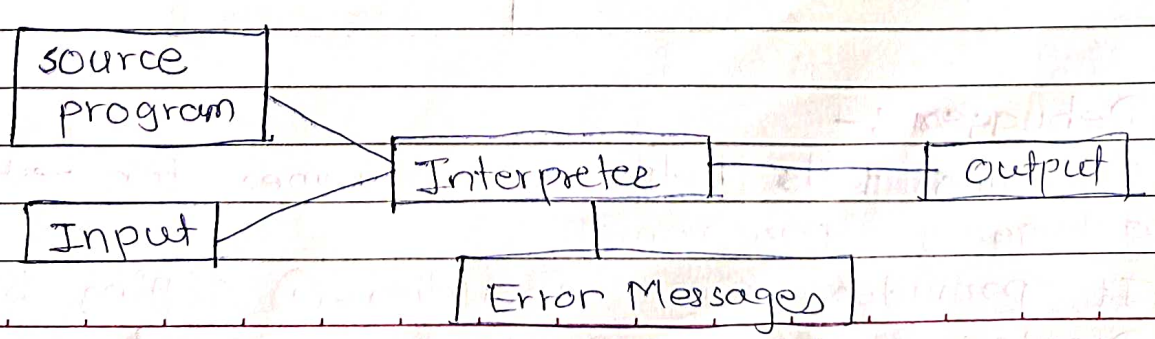
LDF R2, id_3
MULF R2, #5.0
LDF R1, id_2
ADDF R1, R2
STF id_1, R1



<Fig: phases of compiler>

Interpreters -

"Interpretation" - performing the operations ~~inter~~ implies by the source program.



Page No.
Date

Difference between Compiler & Interpreter -

SR. NO	Compiler	Interpreter
1.	Compiler Takes Entire program as input.	Interpreter takes single instruction as input.
2.	Intermediate Object code is Generated.	No Intermediate object code is Generated.
3.	Conditional Control statement are Executed faster.	Conditional control statements are executed slower.
4.	Memory Requirement: More (since Object Code is Generated)	Memory Requirement: Less
5.	Program need not be compiled every time.	Every time higher level program is converted into lower level program.
6.	Errors are displayed after entire program is checked.	Errors are displayed for every instruction interpreted (if any)
7.	Example: C Compiler	Example: BASIC

Debugger :-

- Debugging tools helps programmer for testing & debugging programs.
- It provides some facilities - (1) Setting breakpoints. (2) Displaying Values of Variables

Page No.
Date

Operating System:-

It is system software which provides interface between user and Hardware (comp. system)

Elements of Assembly Language programming:-

An assembly language programming provides three basic features which simplify progra- 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 diagnostic. 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 performances memory binding to these names; the programmer need not know any details of the memory binding performanced by the assembler.

3. Data declarations;

Data can be declared in a variety of notations, including the decimal notation. This avoids manual conversion of constant

Page No.
Date

into their internal machine representation, for example -5 into $(11111010)_2$ or 10.5 into $(41A80000)_{16}$.

Statement Format:

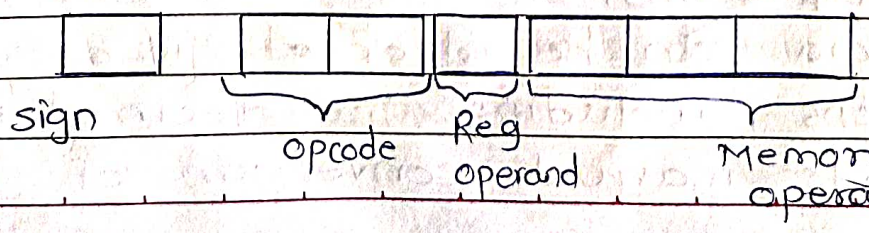
An assembly language statement has the following format:

[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	
09	READ	first operand is not used.
10	PRINT	

Instruction Format



Page No.
Date

- sign is not a part of Instruction.

An Assembly and equivalent machine Language pro

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

Assembly Language statements -

1) Imperative statements
2) Declarative / Declaration statement
3) Assembler directive

Page No.
Date / /

Imperative statements:

- Imperative means Mnemonics
- These are executable statements
- Each imperative statement indicates an action to be taken during execution of the program.

eg -
```
MOVER
BREG , X
STOP
READ X
ADD AREG ,Z
```

Declarative statements:

- Declaration statements are for reserving memory for Variables.
- We can specify the initial Value of a Variable.
- It has two types:
- 1) DS // Declare storage
- 2) DC // Declare Constant

DS (Declare Storage)

- Syntax

[Label] DS <Constant Specifing size>

eg - X DS 1

DC (Declare Constant)

Syntax

[Label] DC <constant Specifing Value>

eg - Y DC '5'

Page No.
Date

Assembler Directive

- Assembler directive instruct the assembler to perform certain actions during assembly of a program.
- Some assembler directives are:

```
START <address constant>
  :
END
```

Advanced Assembler Directives:

1. ORIGIN
2. EQU
3. LTORG

ORIGIN:

The origin directive tells the assembler where to load instructions and data into memory.

<u>syntax</u> –

ORIGIN

Equate:

The EQU directive is used to equate a name with an expression, symbolic address, or number. whenever these numbe name is used as symbol, it is replaced.

- we might do something such ~~that~~ as following, which makes the symbol, ~~it is replaced.~~ R12 to be equal to 12 and replaced by the value when the assembler is run.

eg – R12 EQU 12.

Page No.
Date / /

LTORG :

The LTORG directive instruct the assembler to assemblee to assemble the current literal pool immediately.

- The Literal Pool contains a collection of anonymous constant definations, which are generated by the assemblee.
- The LTORG directives defines the start of a literal pool.

syntax:

LTORG

How LC Operates ?

Sr. NO		LC
1	START 100	
2	MOVER AREG, X	
3	MOVER BREG, Y	
4	ADD AREG, BREG	
5	MOVEM, AREG, X	
6	X DC '10'	
7	Y DC '15'	
8	END	

Page No.
Date

Sr No		LC
1	START 100	
2	MOVER AREG, X	100
3	MOVER BREG, Y	101
4	ADD AREG, BREG	102
5	MOVEM AREG, X	103
6	X DC '10'	104
7	Y DC '15'	105
8	END	

Q. Identify Symbol, literals, AD, DS, IS, Symbol, Literal Label.

```
START 100
MOVER BREG ='2'
LOOP MOVER AREG, N
ADD BREG, ='1'
ORIGIN LOOP+5
LTORG
ORIGIN NEXT +2
LAST STOP
N DC '5'
END
```

Page No.
Date / /

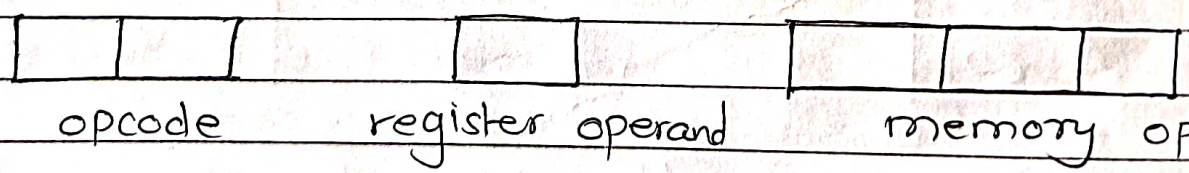
Ans:-

Sr. No	AD	DS	IS	Symbol	Literal	Label
1	AD					
2			IS		=2	
3			IS	N		LOOP
4			IS		=1	
5	AD					
6	AD					
7	AD					
8			IS			LAST
9		DS				
10	AD					

- Machine Instruction format:

In machine structure consider any hypothetical assembly language.

- It supports three registers:
  - AREG
  - BREG
  - CREG



<fig: Machine Instruction format>

Page No.
Date

It supports 11 different operations

1. STOP
2. ADD
3. SUB
4. MULT
5. MOVER
6. MOVEM
7. COMP
8. BC
9. DIV
10. READ
11. PRINT

- In this hypothetical machine, first operand is always a CPU Register.
- Second operand is always memory operand
- READ & PRINT instructions do not use first operand.
- The STOP instruction has no operand
- Each symbolic opcode is associated with machine opcode.
- These details are stored in Machine Opcode Table (MOT)
- MOT contains
  1. Opcode in Mneonic form
  2. Machine code associated with opcode.

Symbolic opcode	Machine code for Opcode
START	01
END	02
LTORG	03
ORIGIN	04
EQU	05

Page No.
Date / /

Symbolic opcode	Machine code for opcode	Size of Instruction (in number of words)
STOP	00	1
ADD	01	1
SUB	02	1
MULT	03	1
MOVER	04	1
MOVEM	05	1
COMP	06	1
BC	07	1
DIV	08	1
READ	09	1
PRINT	10	1

SR.NO	Declarative Statement	Machine Opcode
01	DS	01
02	DC	02

SR.NO	Symbolic opcode	Machine opcode
1	AREG	01
2	BREG	02
3	CREG	03

Advantages of Assembly Language

1) Reduced errors
2) Faster Translation times
3) changes could be made easier & faster.

Page No.
Date

Disadvantages of Assembly Language.

1. Many Instructions are required to acheive small task.
2. Source program tends to be large & difficult to follow.
3. Programmer requires knowledge of processor architecture & instruction set.
4. Programs are machine dependent, requires complete rewrites if the hardware is changed

A Simple Assembly scheme.

Following steps are used for the design specification of an assembler,

1. Find out the information necessary to perform a task.
2. Specify data structure and define format of data structure.
3. To obtain and maintain the information, determine required processing.
4. To perform the task, determine required processing.

Synthesis phase

- Let us consider the following assembly statement.

MOVER CREG, TWO

Corresponding to the above statement following information, synthesis the machine instruction.

1) The "Two" is name of memory word address.
2) MOVER mnemonic is machine operation code,

Page No.
Date / /

- Address of the memory word information depends on the source program. so it is available by the analysis phase.
- The second information i.e. machine operation Code does not depends on the source program. It only dependes on assembly language. so, the synthesis phase can determine this information for its own use

Data structures used in synthesis use

1. Symbol Table
2. Mnemonic table

- symbol table contains name & address Field
- symbol table is built by the analysis phase
- It also contains flags to indicate errors.
- Mnemonic table contains Mnemonic and opcode

Analysis phase

- primary function is to built symbol table for symbol table it must determine the addresses are directly determine.
- Consider the following program :-
- Memory allocation means the fixing the address of assembly language statement.
- Suppose we want to fix the memory address of M, then also fix the address of remainin instructions.
- Location counter is data structure used to implement the memory allocation, Location Counter (LC) is always made to contain the

address of next memory word in the target program.

LC is initialized to the constant specified in the START statement.

- The detail procedure is given below.

1. In the analysis phase, it reads the label of the assembly statement. The label and the contents of LC is entered as new entry in symbol tabel.
2. It finds the number of memory words required by assembly statements & updates the LC Contents.
3. Location counter points to next memory word in the target program
4. Analysis phase needs to know length of different instruction for updating the LC.
5. These information are provided and depends upon the assembly language. So, the Mnemonic table can extended to include this information in a new field called Length.

	Assembly language	Machine language			
	START 200				
	READ M	200)	+09	0	212
	MOVER BREG, TWO	201)	+04	2	214
		202)	+05	2	215
	MOVEM BREG, DATA	203)	+03	2	215
Here	MULT BREG, DATA	204)	+04	3	215
	MOVER CREG, DATA	205)	+01	3	214
	SUB CREG, TWO	206)	+05	3	215

Page No.
Date / /

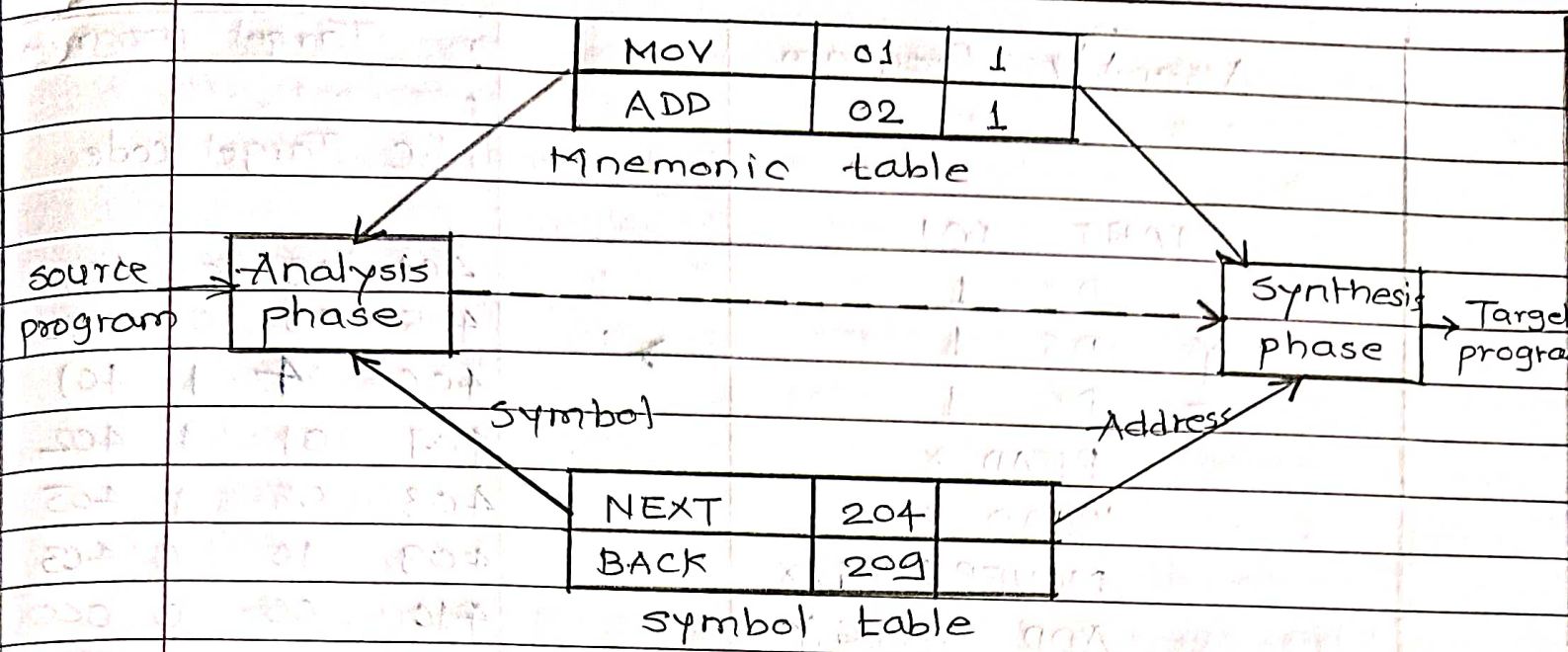
	Assembly language	Machine Language
	MOVEM CREG, DATA	207) +06 3 212
	COMP CREG, M	208) +07 2 203
	BC LE, Here	209) +05 2 203
	MOVEM BREG, OUTPUT	210) +10 0 213
	PRINT OUTPUT	211) +00 0 000
	STOP	212)
M	DS 1	213)
OUTPUT	DS 1	214) +00 0 001
TWO	DC '1'	215)
DATA	DS	
	END	

2 Synthesis phase task

- Synthesis phase is used to build the machine instruction for every assembly statement. It uses the symbol table which is generated by analysis phase during analysis of statements & mnemonic table.
- Synthesis phase includes:

1) search of machine opcode equivalent to the mnemonic table.
2) search of addresses of a memory operand from the symbol table.
3) Synthesize a machine instruction.

following fig. shows the phases of an assembler.



<Fig: Data structure used in analysis and synthesis phase>

Pass structure of Assembler

A whole scanning of a source program by assembler is called as a pass of assembler. There are two assembly schemes such as, single pass assembler & two pass assembler.

1. Single pass assembler

- In single pass assembler, one by one statement is given to first phase that is analysis phase. In this phase opcodes are validated by concentrating the Mnemonics table and symbol table created.
- After the analysis of this source statement, it can be forwarded to synthesis phase for further processing
- Consider the following program.

Page No.
Date / /

Assembly Program		Target program			
		LC	Target code		
START 401					
X DS 1		404	09	0	401
Y DS 1	→	405	09	0	402
Z DS 1		406	04	1	401
READ X		407	01	1	402
READ Y		408	05	1	403
MOVER AREG, X		409	10	0	403
ADD AREG, Y		410	00	0	000
MOVEM AREG, Z					
PRINT Z					
STOP					

- In the above program START statement direct the assembler to initialize the location counter (LC) to 401.
- The symbol X is placed in symbol table at memory location 401,
- The symbol Y is stored at memory location 402 and the symbol Z is stored at memory location 403.

Symbol Table

Symbol	Address
X	401
Y	402
Z	403

Page No.
Date

The instruction READ, ADD, MOVER, MOVEM, PRIN, STOP can be ~~display~~ analysed by ~~analyze~~ analysis phase and generates the machine code according to the opcode.

a) forward reference problem

Discuss the example what is forward reference problem (SPPU - mar-17 - 5 mks)

Assembly program			Target program			
			LC	Target code		
	START	400				
X	DS	1				
Y	DS	2				
Z	DS	1	403	09	0	400
	READ	X	404	09	0	402
	READ	Y	405	04	1	401
	MOVER	AREG, X	406	06	1	402
COMP		AREG, Y	407	0?		???
BC		GE, NST				
BC		ANY, AT NST				
PRINT		Y				
STOP						

(Target code will start from read X statement)

In the above program the instruction "BC GE NST" cannot be translated into machine code because the entry of symbol NST is not found in symbol table.

Page No.
Date

- If this is not translated into equivalent machine code ~~because~~ then subsequent statement are also not translated. So the source program is not executed.
- This problem is nothing but forward reference problem
- As we know that forward reference problem is nothing but the symbol used before its declaration
- forward reference problem can be resolved by using back-patching method.
- In back patching method forward-reference problem is handled by creating table called as Table of Incomplete Instruction (TII)
- The operand field of instruction having such problem is left blank. The entry (<instruction address>, <symbol>) is added into Table of Incomplete Instruction (TII)
- when the address of forward reference symbol is encountered then it can be added into symbol table.

Example -

For the following ~~pec~~ ~~pol~~ piece of assembly language code, show the contents of symbol table, literal table, pool table, Assume machine opcodes and size of instruction equal to one [6 MKS]

Page No.
Date / /

```
    START 100
A   DC 10
    MOVER AREG, B
    MOVEM BREG ='1'
    Add AREG ='2'
    SUB BREG,='1'
B   LTORG
    PRINT C
    MOVER CREG, LOOP
C   EQU B +10
    ORIGIN 10
    MOVER AREG, ='1'
    END
```

Ans :-

Symbol Table

Symbol	Address
A	100
B	111
C	112
LOOP	114

Literal Table

Literal	Address
='1'	105
='2'	106
='1'	107
='1'	111

Pool Table

Literal No
#3
#2

Design of Two-Pass Assembler

Q. Explain design of Two-pass Assembler

- In two pass assembler the forward reference problem can be resolved easily because in first pass all the LC processing is done & all the symbol that are used in the source program has the valid memory addresses associated with them.

Page No.
Date

- so the second pass only use the symbol table & generates the target code by using the addresses found in symbol table.
- following are the tasks that are performed during two pass of assemblee.

PASS-I

1. It seperate the labels, mnemonic opcode and operand fields of statement.
2. Validate the Mnemonic opcode through opcode table.
3. Build the symbol of literal table.
4. perform the LC processing.
5. Generate the intermediate code.

PASS-II

1. Takes the addresses of symbols from symbol table addresses of literals from literal table and addresses of opcodes from opcode table.
2. Synthesize the target program.

Advanced Assembler Directives

Q. Explain ORIGIN, EQU & LTORG with example. [SPPU March-19 March-18 3M]

There are three advanced assembler directives

1. ORIGIN
2. EQU
3. LTORG

Page No.
Date

1. ORIGIN

- The syntax of ORIGIN assembler directive is as follow:

ORIGIN <address speci>

where <address speci> is an operand specification or it may constant. The ORIGIN directive is used to set the location counter (LC) specified in the address specification.

Example -

1. ORIGIN 400

It set the LC to 400 memory location.

2. ORIGIN AGAIN +4

It set the location counter to the address of AGAIN+4 location, that means if the AGAIN is at the location 408 then LC will set to 412 memory location.

2. EQU

The syntax of EQU assembler directive is as follow:

<symbol> EQU <Address speci>

This directive can associate the symbol with the <address speci>. The new symbol gets the same address as specified in <address speci>

Example — RESULT EQU TOTAL

Here the symbol RESULT gets the same address as TOTAL.

Page No.
Date / /

3. LTORG

By default assembler allocates memory for literal at the end of source program

- The LTORG statement is used to specify where literals should be placed during the program execution
- This means when LTORG statement is found by assembler then addresses for literals ~~should~~ that still do not have address are allocated.
- Consider following example.

```
START    401
         READ  X
         READ  Y
         MOVER AREG,'=4'
         MOVER BREG,'=5'
AGAIN    SUB   AREG,Y
         MOVEM AREG,X
         COMP  AREG,X
         BC GE, AGAIN
         LTORG
         DIV   AREG,'=1'
BACK     EQU   PREV
         ADD   CREG,X
         X     DS 1
         Y     DS 1
         END
```

- In the above example LC initialize to 401 because of START 401 statement. The literals '=4' '=5' does not have any addresses. They are

Page No.
Date

kept in literal pool without addresses while processing statement number 4 and statement number 5 respectively.

- The symbol ~~again~~ AGAIN is entered into the symbol table while processing statement number ~~6~~. ~~and statement number 5 respectively~~. During processing statement number 10, the LTORG statement is encountered and it directs the assembler that assign address to all previous literals that does not have any address.
- So the literal '=4', '=5' gets the address where it is declared. Following table shows the address for symbols and literals in equivalent tables.

Literal Table

Literal	Address
'=4'	409
'=5'	410
'=1'	414

Symbol Table

Symbol	Address	Length
AGAIN	405	1
BACK	411	1
X	413	1
Y	414	1

Pass-I of the Assembler

Q Explain the data structure required for two pass assembler in details. [SPPU-Mar-19 6MKS]

- Pass-I of two-pass assembler uses some data structure. They are explained in brief.

Data structures used in pass I of two Assembler

1. OPTAB (opcode table)
2. SYMTAB (symbol table)
3. LITTAB (literal table)
4. POOLTAB (pool table)

Page No.
Date

1. OPTAB (Opcode table)

- It is used to see mnemonic operation codes and converts them to their equivalent machine language. In more difficult assembler the table also contains information about instruction format and length.
- In pass-I the OPTAB checks the operation code in the program and validate it.
- OPTAB is also called as Mnemonic table. The mnemonic table contains three fields:

1. Opcode
2. class
3. Info

where class indicates the type of statement of given opcode. The info field contains the pair ~~given~~ (Opcode, Length) if the class has the imperative statement otherwise it contains routine ID.

2. SYMTAB (symbol table)

- Symbol table includes the name & address for each label in the source program.
- In pass-1 all the symbols used in the source program are added in symbol table with the corresponding address & length.
- In Pass-2 address of all the symbol used in program are taken from symbol table and added into the assembly instruction to generate target program.

Page No.
Date

3. LITTAB (Literal table)
Literal table includes the name of the literals and there corresponding addresses from the source program.

4. POOLTAB (pool Table)
Pool table is used to record the pools in the literal table.

Q. write an algorithm for pass-I of two pass assembler.

Algorithm for pass-I Assembler

1. Initialize LC=0
Initialize Current_Pool=1
Initialize Lit_entry=1
2. Perform following steps
While next statement is not an END statement.
i) Isolate label, opcode, operand from statements.
ii) If label is present then add label into symbol table. Enter pair (label, LC) into SYMTAB.
iii) If opcode is LTORG statement then
a. Assign addresses to all literals from LITTAB [current_Pool] to LITTAB [Lit_entry-1]
b. Update the LC
current_Pool = current_Pool +1;
c. Generate Intermediate code.
iv) If opcode is START or ORIGIN statement then
a. LC = Value specified in the operand field
b. Generate Intermediate code.
v) If opcode is EQU statement then
a. Address of symbol is now the address specified in operand field.

Page No.
Date

b. Generate Intermediate code.

vii) If opcode is an imperative statement then get machine code from OPTAB

a. LC = LC + length from OPTAB

b. If operand is a literal then

viii) Add literal into LITTAB [Lit_entry]

ix) Lit_entry = Lit_entry + 1;

x) Generate Intermediate code

a. If operand is a symbol then

b. Add symbol into SYMTAB

c. Generate intermediate code (S, code) (S, symbolentryno)

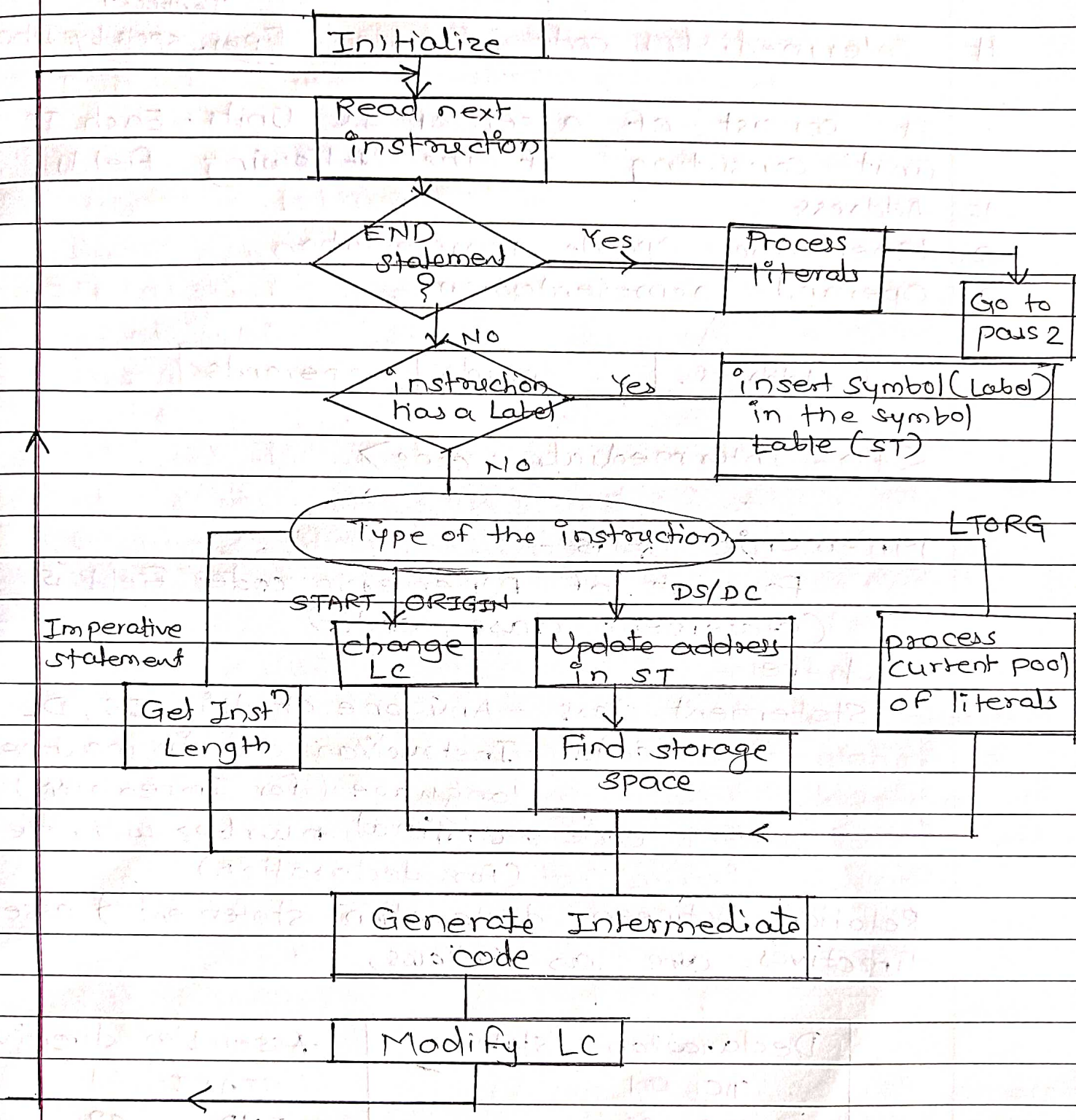
3. If END statement then

a. Generate intermediate code.

Flowchart of Pass-I of Two-Pass Assembler

Q. Draw a neat flowchart of Pass-I of two pass assembler [SPPU-Dec-18, May 19 SMKS]

Page No.
Date



<fig: Flowchart of Pass-I of two-pass Assembler>

Page No.
Date / /

Intermediate code forms

Textbook-1 [Ref. spos by Dhamdh

- It consist of a set of IC Units. Each IC unit consisting of the following fields

1. Address.
2. Mnemonic opcode representation
3. Operand representation

Address	opcode	operands

<fig- Intermediate code>

Mnemonic fields -

Format of mnemonic ~~code~~ field is (statement class, code)

where

Statement class = Any one of this IS, DL or AD

code = Instruction code in machine language (for Imperative)

code = Ordinal number within the class (for declaration)

Relation between declaration statement & assembler directives are as follows,

Declaration statement	Assembler directives	
DC 01	START	01
DS 02	END	02
	ORIGIN	03
	EQU	04
	LTORG	05

<fig: Codes for declaration statements & directives>

Page No.
Date

The mnemonic opcode field contains a pair of the form:

(statement class, code)

where statement class can be one of IS, DL and AD standing for imperative statement, declaration statement & assembler directive respectively. For an imperative statement, code is instruction opcode in the machine language. For declaration and assembler directives, code is an ordinal number within the class. Thus (AD, 01) stands for assembler directive number which is directive START.

Intermediate code for Imperative statements:

Here It consider 2 Varients for Intermediate code which ~~offe~~ differ in information contained in their operand fields. For simplicity, the address and mnemonic opcode fields are assumed to contain identical information in both variants.

Varient 1 :-

In the imperative statement first operand 1 is either register name or condition code. If it is a register then it can be represented ~~through~~ using numbers 1, 2, 3, 4 for AREG, BREG, CREG, DREG respectively. If the first operand is condition code then it can be represented using 1 to 6 for LT, LE, EQ, GT, GE, ANY respectively.

Page No.
Date / /

The operand 2 has following format:-
(operand_class, code)

where operand class is one of C, S & L standing for constant, symbol & literals respectively.
For a constant, the code field contains the representation of the constant itself.
For example, in ~~fig~~. following example the operand descriptor for the statement START 200 is (C, 200). For a symbol or literal, the code field contains the entry number of the operand in SYMTAB or LITTAB. Thus entries for a symbol XYZ and a literal='25' would be of the form (S, 17) and (L, 35) respectively.

Consider following example

Variant I for source code

START	200	(AD, 01)	(C, 200)
MOVER	AREG, ='5'	(IS, 04) (1)	(L, 01)
MOVER	BREG='1'	(IS, 04) (2)	(L, 02)
MOVEM	BREG, A	(IS, 05) (2)	(S, 01)
LTORG		(AD, 05)	
MOVER	CREG, ='3'	(IS, 04) (3)	(L, 03)
MOVER	DREG='1'	(IS, 04) (4)	(L, 04)
ADD	CREG, B	(IS, 01) (3)	(S, 02)
LTORG		(AD, 05)	
MOVER	BREG, ='5'	(IS, 04) (2)	(L, 05)
MULT	BREG, A	(IS, 03) (2)	(S, 01)
PRINT	A	(IS, 10)	(S, 01)
STOP		(IS, 00)	

A DS 1	(DL,02)	(C,1)
B DC '2'	(DL,01)	(C,2)
END	(AD,02)	

Varient II

- Varient II is different in working as compared to Varient I. In Varient II Operand fields are selectively processed.
- Operand fields are processed to carry out LC processesing. The main function of Varient II is to identify literals from the imperative statement.
- when it encounters the literals they must be entered into the LITTAB and it can be represented using a pair (L, en) where L is a Literal and 'en' is the entry number of Literals.
- Consider the following example which shows the Varient I & varient II for given source program.

Source program	Varient I	Varient II
START 400	(AD,00) (C,400)	(AD,00) (C,400)
READ X	(IS,09) (S,0)	(IS,09) X
READ Y	(IS,09) (S,1)	(IS,09) Y
MOVER AREG,X	(IS,04) 1 (S,0)	(IS,04) 1 X
ADD AREG,Y	(IS,01) 1 (S,1)	(IS,01) 1 Y
MOVEM AREG,Z	(IS,05) 1 (S,3)	(IS,05) 1 Z
MULT AREG,='D'	(IS,03) 1 (L,0)	(IS,03) 1 (L,0)
SUB AREG,A	(IS,02) 1 (S,4)	(IS,02) 1 A
COMP AREG,ONE	(IS,06) 1 (S,2)	(IS,06) 1 ONE
LTORG	(AD,05)	(AD,05)

Page No.
Date

Source program	Varient I	Varient II
PRINT Z	(IS,10) (S,3)	(IS,10) Z
STOP	(IS,00)	(IS,00)
X DS 100	(DL,01) (C,100)	(DL,01) (C,100)
Y DS 2	(DL,01) (C,2)	(DL,01) (C,2)
ONE DC '0'	(DL,02) (C,0)	(DL,02) (C,0)
Z DS 3	(DL,01) (C,3)	(DL,01) (C,3)
A DS 1	(DL,01) (C,2)	(DL,01) (C,2)
END	(AD,02)	(AD,02)

Comparision between Varient I and Varient II

parameter	Varient-I	Varient II
Processing	In Varient-I all the operands are processed	In Variant II main Focus is on processin literals.
IC	IC is in compact format	IC is in less compa format
Pass I	Varient-I performs extra work in PASS-I	Varient-II Simplifies task in PASS-I
Memory Utilization	In Varient-I memory utilization is not balanced	In Varient II memo utilization is balance in better way.
Memory Amount	Occupies more memory in PASS-II	It occupies less memory.

Page No.
Date

Processing of Declaration and Assembler Directives

- It is required to consider how fae declaratio and assemblee directives can be processed in Pass-I of an assembler.
- following two key Questions in this context are-

1. It is necessary to represent the address of each source statement in the intermediate code?
2. It is necessary to have a representation of DS statements & assembler directives in the intermediate code?

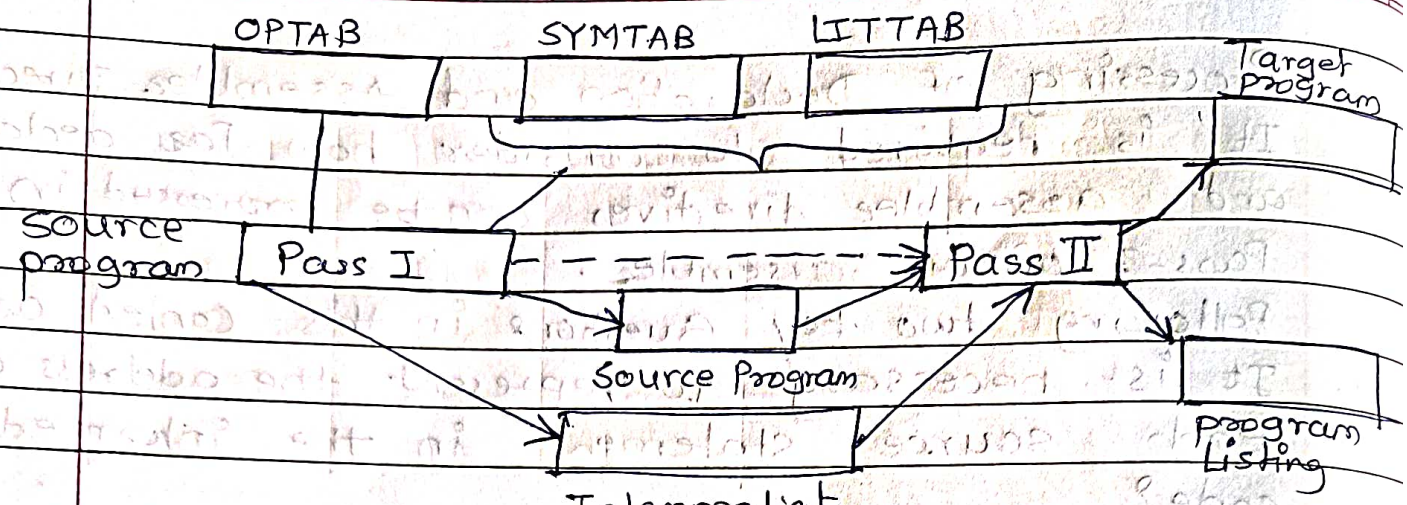
Consider the fragement of an assembly language program and its intermediate code.

	START 200	→	(AD,01)	(C,200)
AREA1	DS 20	⇒ 200)	(DL,02)	(C,20)
SIZE	DC 5	220)	(DL,01)	(C,5)

Here, it is redundant to have the representation of the START and DS statements in the intermediate code, since the effect of these statements is implied in the face target program. No processing advantage exists in either case since conversion of the constant into the machine representation is required anyway. If a DC statement defines many constants eg.

DC '5,3,-7'

a series of (DL,01). ~~The operand~~ units can be put in the intermediate code.



→ Data access

→ Control transfer

<Fig: Use of data structure and files in a two-pass assemblee>

Pass-II of Assembler

Pass I contains an Intermediate Code, SYMTAB and LITTAB. Pass II uses these data structures and converts the intermediate code into target code with the help of SYMTAB and LITTAB entries.

1. LC = 0
2. While next statement is not END statement
   (i) If START or ORIGIN statement
      (a) LC = value specified in operand.
   (ii) If LTORG statement
      Update LC as per the addresses assigned to literal in current pool.
   (iii) If declaration statement
      (a) If DC statement then assemble the constant.
      (b) LC = LC + size of operand.

Page No.
Date

(iv) If imperative statement
ⓐ Get operand address as per entry number from ~~SYMB~~ SYMTAB and LITTAB.
Gather instructure in target code.
ⓑ LC = LC + size of instruction.

3. Write target code into Input file.

- Consider the example which shows the target code generation from IC.

Source Program	Varient I	Target code
START 400	(AD,00) (C,400)	
READ X	(IS,09) (S,0)	09 0 412
READ Y	(IS,09) (S,1)	09 0 413
MOVER AREG, X	(IS,04) 1 (S,0)	09 1 413
ADD AREG, Y	(IS,01) 1 (S,1)	04 1 412
MOVEM AREG, Z	(IS,05) 1 (S,3)	05 1 416
MULT AREG, ='D'	(IS,03) 1 (L,0)	03 1 409
SUB AREG, A	(IS,02) 1 (S,4)	02 1 419
COMP AREG, ONE	(IS,06) 1 (S,2)	06 1 415
LTORG	(AD,05)	
PRINT Z	(IS,10) (S,3)	10 0 416
STOP	(IS,00)	00 0 000
X DS 1	(DL,01) (C,1)	
Y DS 2	(DL,01) (C,2)	
ONE DC '0'	(DL,02) (C,0)	
Z DS 3	(DL,01) (C,3)	
A DS 1	(DL,01) (C,2)	
END	(AD,02)	

Page No.
Date / /

Symbol Table

Symbol	Address	Length
X	412	1
Y	413	2
ONE	415	1
Z	416	3
A	419	1

Literal Table

Literal	Address
=D'1'	409

Pool Table

Literal No.
#1

Listing and Error Reporting [Ref. SP Pos-I D.M. Dhamdhere second edn]

- Design of an error indication scheme involves some decision which influence the effectiveness of error reporting and the speed and money requirement of the assembler.
- The basic decision is whether to produce program listing and error reports in Pass-I or delay these action until Pass-II. producing ~~and~~ the listing in the First Pass has the advantage that the source program need not to be preserved till pass-II.
- This conserves memory and avoid some amount of duplicate processing.
- The design decision also has very important implication from the programmer viewpoint.
- A listing produced in pass-I can report

Page No.
Date

only certain errors in the most relevant place, that is against source statement itself.

- Examples of such errors are syntax errors like missing commas or parentheses & semantic errors like duplicate definations of symbols. Other errors like references to undefine ~~error~~ variables like only be reported at the end of the source program.
- 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.

St. No	statement	Address
001	START 200	
002	MOVER AREG, A	200
003	⋮	
	MOVER BREG, A	207
	** error ** Invalid opcode	
010	ADD BREG, B	208
014	A DS 1	209
015	⋮	
021	A DC '5'	227
	** error ** Duplicate defination of Symbol A	
022	⋮	
035	END	
	** error ** Undefined symbol B in statement 10	

< error reporting in Pass-I >

Page No.
Date / /

For effective error reporting, it is necessary to report all errors against the erroneous statement itself.

- This can be achieved by delaying program listing and error reporting till Pass-II.
- The error reports as well as the target code can be printed against each source statement.
- Error specification at proper place affects the effectiveness of reporting and speed as well as memory requirement of the assembler.
- The most important decision is whether to report error in pass-I & pass-II
- The reporting of errors in pass-I have the following advantages.

1. source code statements need not be kept till pass-II
2. Minimizes the memory requirement
3. Duplicate processing is avoided
4. The errors can be reported at most significant position that is in opposition to the source statement itself.

—XOX— END —XOX—