System Programming

Unit 1
Introduction

Introduction

- **What is System?**
  - System is the collection of various components
- Ex:- College is a system.
- College is a system because it consist of various components like various departments, classrooms, faculties and students.
- **What is Programming?**
  - Art of designing and implementing the programs.

- In college system, what is **program?**
- A LECTURE can be a program. Because it has input and output.
- **Input**-> The information that teacher is delivering.
- **Output**-> The knowledge student has been received.

- So **system programming** is an art of designing and implementing system Programs.

What is Software ?

- Software is collection of many programs
- Two types of software

– **System software**: These programs assist general user application programs

- Ex:- Operating System , Assembler etc.

– **Application software**

- These are the software developed for the specific goal. Ex. Media Player, Adobe Reader.

- **System Program:-**

"These are programs which are required for the effective execution of general user programs on computer system."

- **System Programming:-**

" It is an art of designing and implementing system programs."

Syllabus

Unit I	Introduction	09 Hours
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		

Text Editors

- Text editor's example is Notepad.
- Editor is a computer program that allows a user to create and revise a document.
- A text editor is a program in which the primary elements being edited are character strings.
- 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(e.g. C Program or Java Program).

Loaders

- **A loader is a program that takes object code as input and prepares them for execution.**
- It initiates the execution.
- **Functions:**

1. **Allocation**
2. **Linking**
3. **Relocation**
4. **Loading**

Allocation

- Loader allocates space for programs in main memory.

Linking

- If we have different modules of our program.
- Loader links object modules with each other.

Relocation

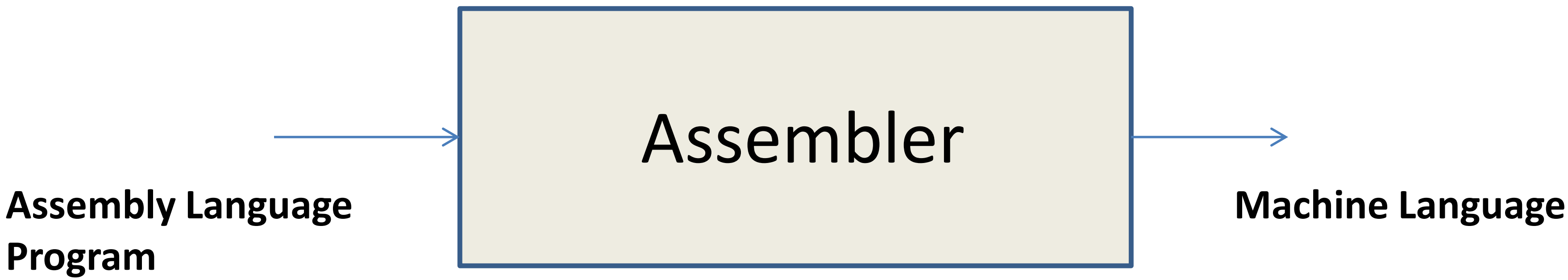
- Adjusting all address dependent location.
- E.g. If we have two Programs Program A and Program B.
- Program A is saved at location 100.
- And user wants to save Program B on same location. That is physically not possible.
- So loader relocates program B to some another free location.

Loading

- Physically loading the machine instructions and data into main memory.

Assembler

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



Macro Processor

- **Macro** allows a sequence of source language code to be defined once and then referred many times.
- Syntax:

Macro macro-name [set of parameters]

// Macro Body

MEND

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

Compiler

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

Debugger

- Debugging tool helps programmer for testing and debugging programs.
- It provides some facilities:
- **Setting breakpoints.**
- **Displaying values of variables.**

Assembly Language

- Assembly language is low level language.
- An assembly language is machine dependent.
- It differs from computer to computer.
- Writing programs in assembly language is very easy as compared to machine(binary) language.

Assembly language programming Terms:

- Location Counter: (LC) points to the next instruction.
- Literals: constant values
- Symbols: name of variables and labels
- Procedures: methods/ functions

Assembly language Statements:

- Imperative Statements:

- Declarative/Declaration Statements:

- Assembler Directive:

Imperative Statements

- Imperative means mnemonics
- These are executable statements.
- Each imperative statement indicates an action to be taken during execution of the program.
- E.g.   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:
- DS   // Declare Storage

- DC   // Declare Constant

DS(Declare Storage):

- Syntax:
- [Label]   DS <Constant specifying size>
- E.g.   X DS 1

DC (Declare Constant):

Syntax:

[Label]   DC   <constant specifying value>

E.g   Y   DC   ‘5’

Assembler Directive

- Assembler directive instruct the assembler to perform certain actions during assembly of a program.
- Some assembler directive are:
- START <address constant>
- END

Advanced Assembler Directives

- 1. ORIGIN
- 2. EQU
- 3. USING
- 4. DROP
- 5. LTORG

Sample Assembly Code

1. START 100 **It is an AD statement becoz it has Assembler directive START**

2. MOVER AREG, X **It is an IS because it starts with mnemonic.**
3. MOVER BREG, Y
4. ADD AREG, Y
5. MOVEM AREG, X

6. X DC ‘10’ **It is an DS/ DL statement because it has DC**
7. Y DS 1 **It is an DS/ DL statement because it has DS**
8. END

Identify the types of statements

State.No	IS	DS	AD
1			
2			
3			
4			
5			
6			
7			
8			

Identify the types of statements

State.No	IS	DS	AD
1			AD
2	IS		
3	IS		
4	IS		
5	IS		
6		DS	
7		DS	
8			AD

Advanced Assembler Directives

- ORIGIN

- EQU

- LTORG

Definitions

- LC:
- Symbol:
- Literals:
- Procedures:

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	

How LC Operates?

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	

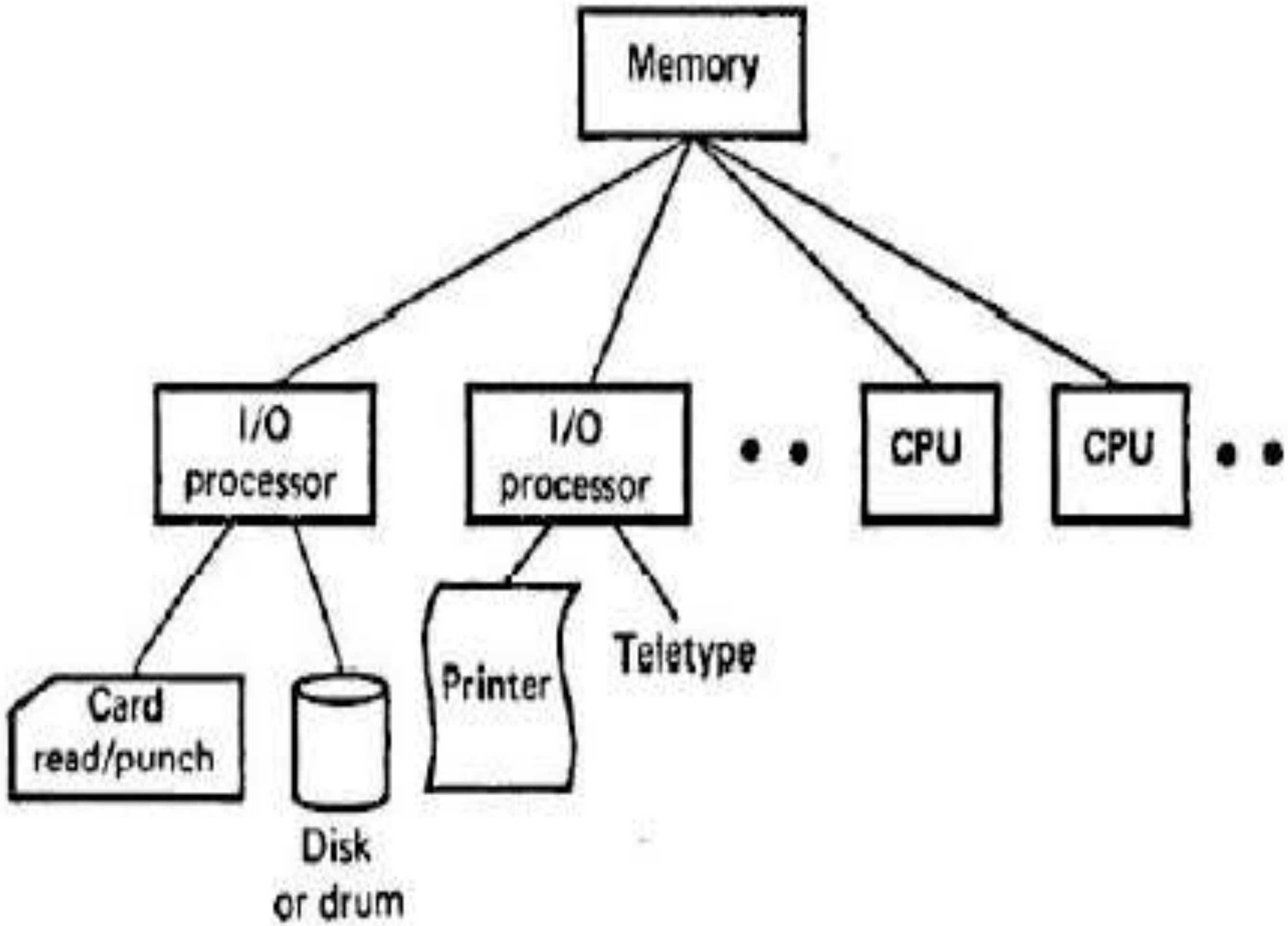
Identify symbol, literals, AD, IS, DS, Label

- 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

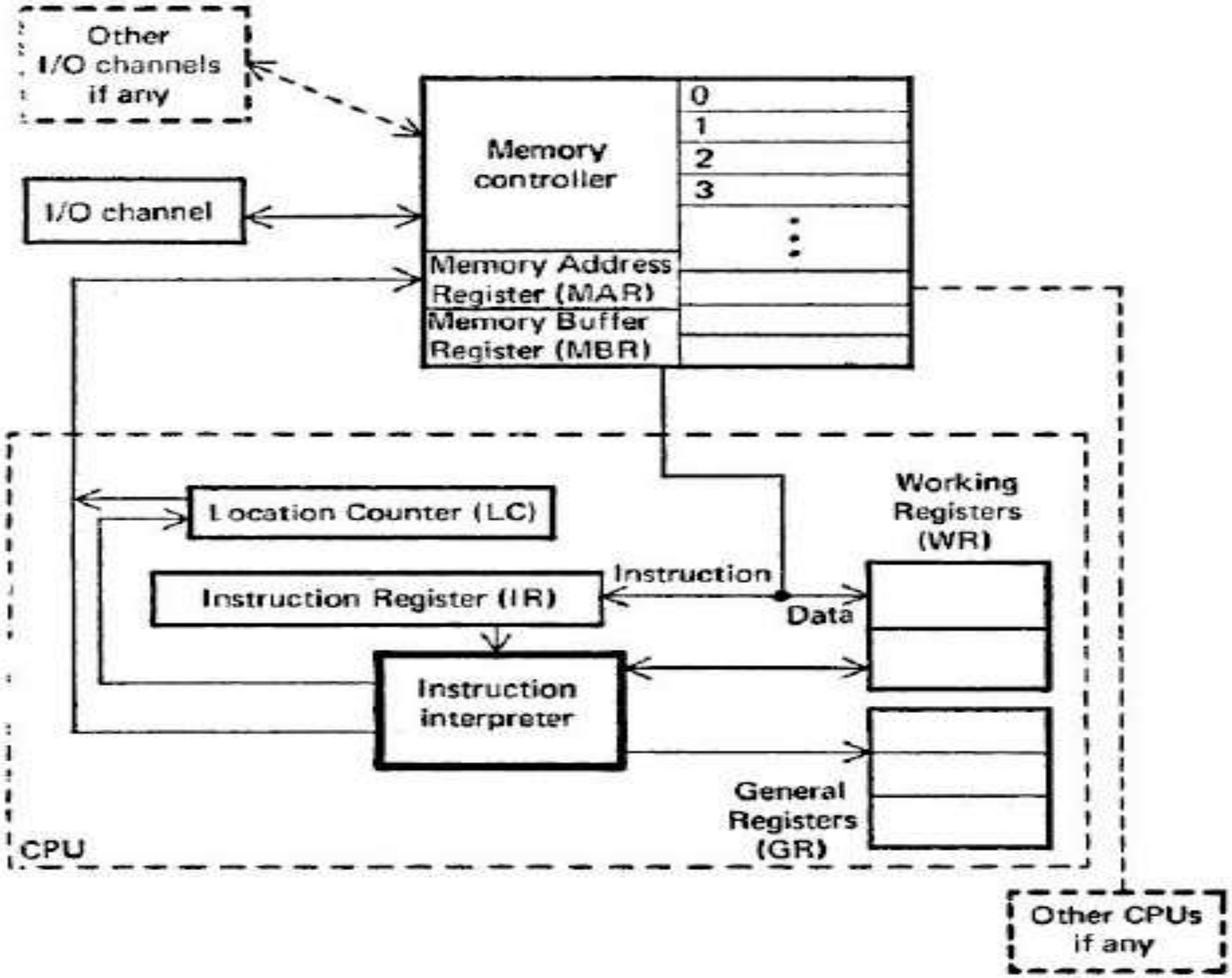
Solution (From Previous Example)

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 Structure

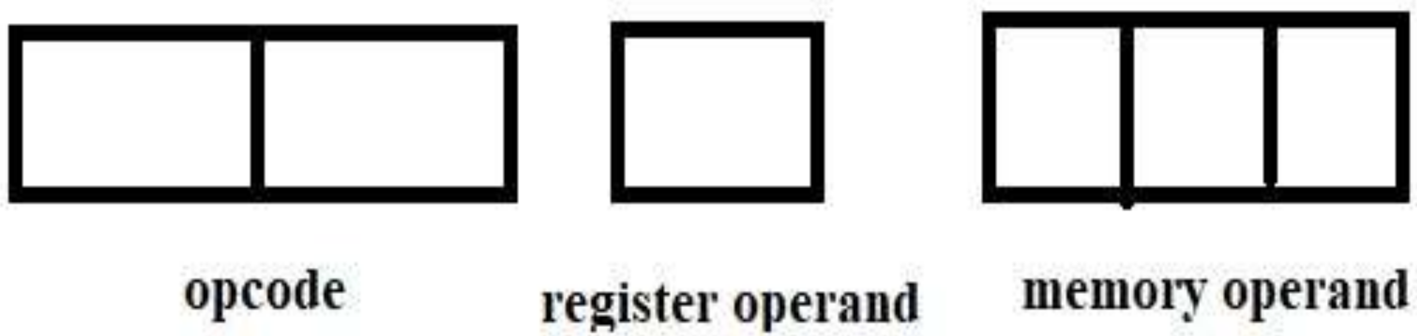


Machine Structure



- Consider any hypothetical assembly language.
- It supports three registers:
- **AREG**
- **BREG**
- **CREG**

- **Machine instruction Format:**



- It supports 11 different OPERATIONS.
- **STOP**
- **ADD**
- **SUB**
- **MULT**
- **MOVER**
- **MOVEM**
- **COMP**
- **BC**
- **DIV**
- **READ**
- **PRINT**

- In this hypothetical machine,
- First operand is always **a CPU register.**
- Second operand is always **memory operand**.
- READ and 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 the opcode.

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

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

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

ASSEMBLER

- An assembly language program can be translated into machine language.
- It involves following steps:
- 1. Find addresses of variable.
- 2. Replace symbolic addresses by numeric addresses.
- 3. Replace symbolic opcodes by machine operation codes.
- 4. Reserve storage for data.

Step 1

- We can find out addresses of variable using LC.
- First identify all variables in your program.
- **START 100**
- **MOVER AREG, X**
- **MOVER BREG, Y**
- **ADD AREG, X**
- **MOVEM AREG, X**
- **X DC ‘10’**
- **Y DC ‘15’**
- **END**

Step 1

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

Sr. No	Name of Variable(Symbol)	Address
1	X	104
2	Y	105

Step2: Replace all symbolic address with numeric address.

- START 100
- MOVER AREG, 104
- MOVER BREG, 105
- ADD AREG, 104
- MOVEM AREG, 104
- X DC ‘10’
- Y DC ‘15’
- END

Memory is reserved but no code is generated.

Step3: Replace symbolic opcodes by machine operation codes.

LC	Assembly Instruction	Machine Code
101	**MOVER AREG, 104**	**04 01 104**
102	**MOVER BREG, 105**	**04 02 105**
103	**ADD AREG, 104**	**01 01 104**
104	**MOVEM AREG, 104**	**05 01 104**
105		
106		
107		

Question For U

```
START 102
READ X
READ Y
MOVER AREG, X
ADD AREG, Y
MOVEM AREG, RESULT
PRINT RESULT
STOP
X DS 1
Y DS 1
RESULT DS 1
END
```

Question For u

```
        START 101
        READ N
        MOVER BREG, ONE
        MOVEM BREG, TERM
AGAIN   MULT BREG, TERM
        MOVER CREG, TERM
        ADD CREG, ONE
        MOVEM CREG, TERM
        COMP CREG, N
        BC LE, AGAIN
        MOVEM BREG, RESULT
        PRINT RESULT
        STOP
        N DS 1
        RESULT DS 1
        ONE DC '1'
        TERM DS 1
```

Assembler

- An Assembler is a translator which translates assembly language code into machine language with help of data structure.

- It has two types
- Pass 1 Assembler.
- Pass 2 Assembler.

General design procedure of assembler

- Statement of Problem
- Data Structure
- Format of databases
- Algorithms
- Look for modularity.

Statement of Problem

- We want to convert assembly language program into machine language.

Data Structure Used

- Data Structure used are as follows:
- Symbol table
- Literal Table
- Mnemonic Opcode Table
- Pool Table

Format of Databases

- Symbol Table:

Name of Symbol	address

- Literal Table:

Literal	address

- MOT:

Mnemonic	Machine Opcode	Class	Length

- Pool Table:

Literal Number

Look for Modularity

- If your program is too long…
- U can make modules of it.

Forward Reference Problem

- Using a variable before its definition is called as forward reference problem.
- E.g.
- START 100
- MOVEM AREG, X
- MOVER BREG, Y
- ADD AREG, Y
- X DC ‘4’
- Y DC ‘5’
- END

- In example variable X, Y are making forward reference.
- So, We can solve it by using back patching.

Consider another example

```
        START   100
        MOVER   AREG, X
L1      ADD     BREG, ONE
        ADD     CREG, TEN
        STOP
X       DC      '5'
ONE     DC      '1'
TEN     DC      '10'
        END
```

Apply LC

	START	100	
	MOVER	AREG, X	100
L1	ADD	BREG, ONE	101
	ADD	CREG, TEN	102
	STOP		103
X	DC	‘5’	104
ONE	DC	‘1’	105
TEN	DC	‘10’	106
	END		

Try to convert it into machine code

Try to convert into machine code

	START	100		
	MOVER	AREG, X	100	04 1 ___
L1	ADD	BREG, ONE	101	01 2 ___
	ADD	CREG, TEN	102	06 3 ___
	STOP		103	00 0 000
X	DC	'5'	104	
ONE	DC	'1'	105	
TEN	DC	'10'	106	
	END			

Backpatching

- The operand field of instruction containing a forward reference is left blank initially.
- Step 1: Construct TII(Table of incomplete instruction)

Instruction Address	Symbol Making a forward reference
100	X
101	ONE
102	TEN

- Step 2: After encountering END statement symbol table would contain the address of all symbols defined in the source program.

SYMBOL NAME	ADDRESS
X	104
ONE	105
TEN	106

- Now we can generate machine code…

04	**1**	**104**
01	2	105
06	2	106
00	0	000

Assembler Directive

- ORIGIN
- LTORG
- EQU

Pass 1 Assembler

- **Pass 1 assembler** separate the labels , mnemonic opcode table, and operand fields.
- Determine storage requirement for every assembly language statement and update the location counter.
- Build the symbol table. Symbol table is used to store each label and each variable and its corresponding address.
- **Pass 2 Assembler**: Generate the machine code

How pass 1 assembler works?

- Pass I uses following data structures.
- 1. Machine opcode table.(MOT)
- 2. Symbol Table(ST)
- 3. Literal Table(LT)
- 4. Pool Table(PT)

- Contents of MOT are fixed for an assembler.

Observe Following Program

```
        START 200
        MOVER AREG, ='5'
        MOVEM AREG, X
L1      MOVER BREG, ='2'
        ORIGIN L1+3
        LTORG


NEXT    ADD AREG, ='1'
        SUB BREG, ='2'
        BC LT, BACK
        LTORG

        BACK EQU L1
        ORIGIN NEXT+5
        MULT CREG, ='4'
        STOP
        X DS 1
        END
```

Apply LC

```
START 200
            MOVER AREG, ='5'          200
            MOVEM AREG, X             201
   L1       MOVER BREG, ='2'          202
            ORIGIN L1+3
            LTORG
                        ='5'          205
                        ='2'          206

 NEXT       ADD AREG, ='1'            207
            SUB BREG, ='2'            208
            BC LT, BACK               209
            LTORG
                        ='1'          210
                        ='2'          211

BACK        EQU    L1
            ORIGIN NEXT+5
            MULT CREG, ='4'           212
            STOP                      213
            X DS 1                    214
            END
                        ='4'          215
```

Construct Symbol table

index	Symbol Name	Address
0	X	214
1	L1	202
2	NEXT	207
3	BACK	202

Construct Literal Table

index	Literal	Address
0	5	205
1	2	206
2	1	210
3	2	211
4	4	215

Pool Table.

- Pool table contains starting literal(index) of each pool.

Literal number
0
2
4

NOW CONSTRUCT INTERMEDIATE CODE/MACHINE CODE

- For constructing intermediate code we need MOT.

Enhanced Machine opcode Table

Table 1.10.1 : An enhanced machine opcode table (MOT)

	Mnemonic opcode	Class	Opcode	Length
0	STOP	IS	00	1
1	ADD	IS	01	1
2	SUB	IS	02	1
3	MULT	IS	03	1
4	MOVER	IS	04	1
5	MOVEM	IS	05	1
6	COMP	IS	06	1
7	BC	IS	07	1
8	DIV	IS	08	1
9	READ	IS	09	1
10	PRINT	IS	10	1
11	START	AD	01	–
12	END	AD	02	–
13	ORIGIN	AD	03	–
14	EQU	AD	04	–
15	LTORG	AD	05	–
16	DS	DL	01	–
17	DC	DL	02	1
18	AREG	RG	01	–
19	BREG	RG	02	–
20	CREG	RG	03	–
21	EQ	CC	01	–

Mnemonic opcode	Class	Opcode	Length
LT	CC	02	–
GT	CC	03	–
LE	CC	04	–
GE	CC	05	–
NE	CC	06	–
ANY	CC	07	–

INTERMEDIATE CODE

- **Format for intermediate code:**
- For every line of assembly statement, one line of intermediate code is generated.

- Each mnemonic field is represented as

- **(statement class, and machine code)**

- Statement class can be:
- 1. IS
- 2. DL/DS
- 3. AD

- So, IC for mnemonic field of above line is,
- **(statement class, machine code)**
- (IS, 04) …………………from MOT

- Operand Field:
- Each operand field is represented as

(operand class, reference)

- The operand class can be:

- 1. C: Constant
- 2. S: Symbol
- 3. L: Literal
- 4. RG: Register
- 5. CC: Condition codes

- E.g. MOVER AREG, X
- **For a symbol or literal the reference field contains the index of the operands entry in symbol table or literal table.**
-
- So IC for above line is:
- **(IS, 04) (RG, 01) (S, 0)**

- For example…

- START 200
- IC: (AD, 01) (C, 200)

Intermediate Code

(AD, 01) (C, 200)

200 (IS, 04) (RG,01) (L, 0)

201 (IS, 05) (RG,01) (S,0)

202 (IS, 04) (RG,02) (L,1)

203 (AD, 03) (C, 205)

205 (DL, 02) (C,5)
206 (DL, 02) (C, 2)

207 (IS,01) (RG, 01) (L, 2)

208 (IS, 02) (RG, 02) (L,3)

209 (IS, 07) (CC, 02) (S, 3)

210 (DL,02) (C,1)

211 (DL,02) (C,2)

212 (AD, 04) (C, 202)

212 (AD, 03) (C, 212)

212 (IS, 03) (RG, 03)(L, 4)

213 (IS, 00)

214 (DL, 01, C, 1)

215 (AD, 02)

215 (DL, 02) (C,4)

Example No.2

```
          START 205
          MOVER AREG, ='6'
          MOVEM AREG, A
LOOP      MOVER AREG, A
          MOVER CREG, B
          ADD CREG, ='2'
          BC ANY , NEXT
          LTORG
          ADD BREG, B
NEXT      SUB AREG, ='1'
          BC LT, BACK
LAST      STOP
          ORIGIN  LOOP+2
          MULT CREG, B
          ORIGIN LAST+1
A         DS        1
BACK      EQU       LOOP
B         DS        1
END
```

- PASS 2 assembler requires two scans of program to generate machine code.
- It uses data structures defined by pass 1. like symbol table, MOT, LT.

Design of two pass assembler

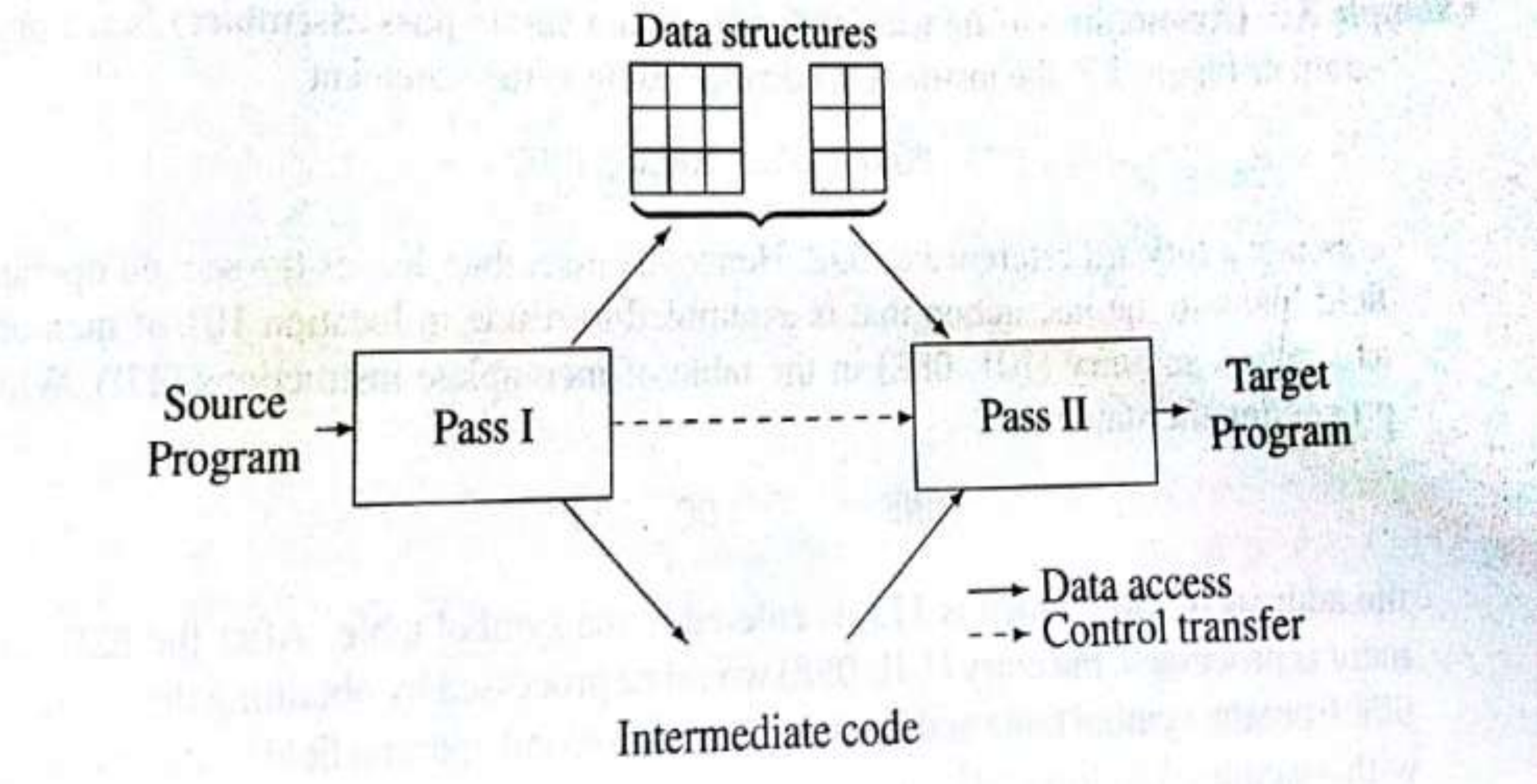
- Tasks performed by the passes of a two pass assembler are as follows:
- **Pass 1:**

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

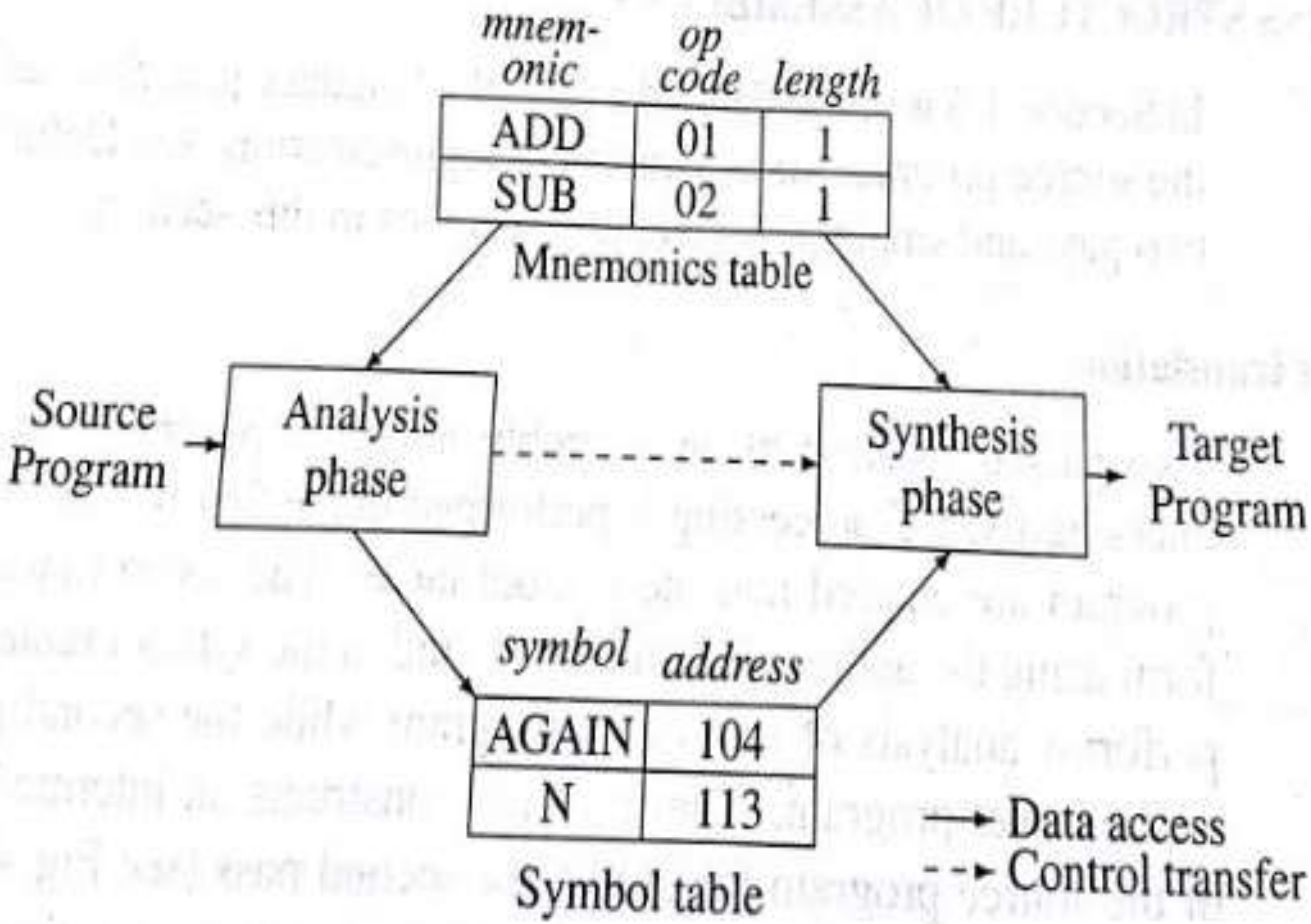
- **Pass 2:**

1. Synthesize the target program.

Two Pass Assembler



Analysis Phase Vs. Synthesis Phase



Pass 1 Algorithm

Algorithm 4.1 (Assembler First Pass)

1. *loc_cntr* := 0; (default value)
   pooltab_ptr := 1; POOLTAB [1] := 1;
   littab_ptr := 1;
2. While next statement is not an END statement
   (a) If label is present then
       this_label := symbol in label field;
       Enter (*this_label, loc_cntr*) in SYMTAB.
   (b) If an LTORG statement then
       (i) Process literals LITTAB [POOLTAB [*pooltab_ptr*]] ... LITTAB [*littab_ptr*−1] to allocate memory and put the address in the *address* field. Update *loc_cntr* accordingly.
       (ii) *pooltab_ptr* := *pooltab_ptr* + 1;
       (iii) POOLTAB [*pooltab_ptr*] := *littab_ptr*;
   (c) If a START or ORIGIN statement then
       loc_cntr := value specified in operand field;
   (d) If an EQU statement then
       (i) *this_addr* := value of <*address spec*>;
       (ii) Correct the symtab entry for *this_label* to (*this_label, this_addr*).
   (e) If a declaration statement then
       (i) *code* := code of the declaration statement;
       (ii) *size* := size of memory area required by DC/DS.
       (iii) *loc_cntr* := *loc_cntr* + *size*;
       (iv) Generate IC '(DL, *code*) ···'.
   (f) If an imperative statement then
       (i) *code* := machine opcode from OPTAB;
       (ii) *loc_cntr* := *loc_cntr* + instruction length from OPTAB;
       (iii) If operand is a literal then
           this_literal := literal in operand field;
           LITTAB [*littab_ptr*] := *this_literal*;
           littab_ptr := *littab_ptr* + 1;
       else (i.e. operand is a symbol)
           this_entry := SYMTAB entry number of operand;
           Generate IC '(IS, *code*)(S, *this_entry*)';
3. (Processing of END statement)
   (a) Perform step 2(b).
   (b) Generate IC '(AD,02)'.
   (c) Go to Pass II.

Pass 2 Algorithm

Algorithm 4.2 (Assembler Second Pass)

1. *code_area_address* := address of *code_area*;
   pooltab_ptr := 1;
   loc_cntr := 0;
2. While next statement is not an END statement
   - (a) Clear *machine_code_buffer*;
   - (b) If an LTORG statement
     - (i) Process literals in LITTAB [POOLTAB [*pooltab_ptr*]] ... LITTAB [POOLTAB [*pooltab_ptr*+1]] – 1 similar to processing of constants in a DC statement, i.e. assemble the literals in *machine_code_buffer*.
     - (ii) *size* := size of memory area required for literals;
     - (iii) *pooltab_ptr* := *pooltab_ptr* + 1;
   - (c) If a START or ORIGIN statement then
     - (i) *loc_cntr* := value specified in operand field;
     - (ii) *size* := 0;
   - (d) If a declaration statement
     - (i) If a DC statement then
       Assemble the constant in *machine_code_buffer*.
     - (ii) *size* := size of memory area required by DC/DS;
   - (e) If an imperative statement
     - (i) Get operand address from SYMTAB or LITTAB.
     - (ii) Assemble instruction in *machine_code_buffer*.
     - (iii) *size* := size of instruction;
   - (f) If *size* ≠ 0 then
     - (i) Move contents of *machine_code_buffer* to the address *code_area_address* + *loc_cntr*;
     - (ii) *loc_cntr* := *loc_cntr* + *size*;
3. (Processing of END statement)
   - (a) Perform steps 2(b) and 2(f).
   - (b) Write *code_area* into output file.

Comparison between Pass 1 and Pass2

Sr. No	Pass 1	Pass 2
01	It requires only one scan to generate machine code	It requires two scan to generate machine code.
02	It has forward reference problem.	It don’t have forward reference problem.
03	It performs analysis of source program and synthesis of the intermediate code.	It process the IC to synthesize the target program.
04	It is faster than pass 2.	It is slow as compared to pass 1.

Pass 1 output and pass 2 output

- Pass 1 assembler generates **Intermediate code.**
- Pass 2 assembler generates **Machine code.**

Consider following example

```
START 200
            MOVER AREG, ='5'        200
            MOVEM AREG, X           201
     L1     MOVER BREG, ='2'        202
            ORIGIN L1+3
            LTORG
                    ='5'              205
                    ='2'              206

   NEXT     ADD AREG, ='1'          207
            SUB BREG, ='2'          208
            BC LT, BACK             209
            LTORG
                     ='1'       210
                     ='2'       211

BACK       EQU    L1
            ORIGIN NEXT+5
            MULT CREG, ='4'         212
            STOP                    213
            X DS 1                  214
            END
                    ='4'              215
```

Symbol Table and Literal Table

index	Symbol Name	Address
0	X	214
1	L1	202
2	NEXT	207
3	BACK	202

index	Literal	Address
0	5	205
1	2	206
2	1	210
3	2	211
4	4	215

I.C	LC	Machine Code
(AD, 01) (C, 200)		
(IS, 04) (RG,01) (L, 0)	**200**	**04 01 205**
(IS, 05) (RG,01) (S,0)	**201**	**05 01 214**
(IS, 04) (RG,02) (L,1)	**202**	**04 02 206**
(AD, 03) (C, 205)	**203**	
(DL, 02) (C,5)	**205**	**00 00 005**
(DL, 02) (C, 2)	**206**	**00 00 002**
(IS,01) (RG, 01) (L, 2)	**207**	**01 01 210**

I.C	LC	Machine Code
(IS, 02) (RG, 02) (L,3)	208	02 02 211
(IS, 07) (CC, 02) (S, 3)	209	07 02 202
(DL,02) (C,1)	210	00 00 001
(DL,02) (C,2)	211	00 00 002
(AD, 04) (C, 202)	212	
(AD, 03) (C, 212)	212	
(IS, 03) (RG, 03)(L, 4)	212	03 03 215
(IS, 00)	213	00 00 000

I.C	LC	Machine Code
(DL, 01, C, 1)	214	
(AD, 02)	215	
(DL, 02) (C,4)	215	00 00 004

Variants of Intermediate Code.

- There are two variants of I.C.:
- Variant I
- Variant II.

Variant I

- In Variant I, each operand is represented by a pair of the form (operand class, code).
- The operand class is one of:

1. S for symbol 2. L for literal

3. C for constant 4. RG for register.

Variant I

	START	100	(AD, 01) (C, 100)
L1	READ	A	(IS, 09) (S, 1)
	SUB	AREG, = '5'	(IS, 02) (RG, 01) (L, 0)
	BC	GT, L1	(IS, 07) (CC, 03) (S, 0)
	STOP		(IS, 00)
A	DS	1	(DL, 01) (C, 1)
	–		–
	–		–
	–		–

Variant II

- In variant II, operands are processed selectively.
- Constants and literals are processed. Symbols, condition codes and CPU registers are not processed.

Variant II

A sample intermedi...

Fig. 1.10.6.

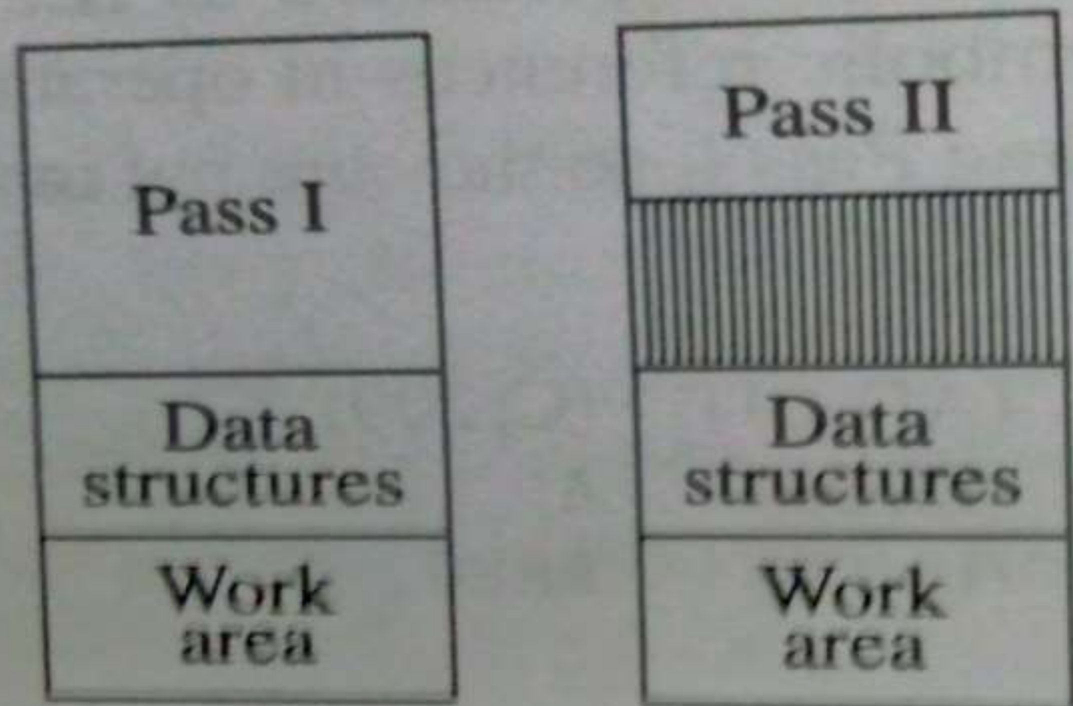
	START	100	(AD, 01) (C, 100)
L1	READ	A	(IS, 09) A
	SUB	AREG, = '5'	(SI, 02) AREG, (L, 0)
	BC	GT, L1	(IS, 07) GT, L1
	STOP		(SI, 00)
A	DS	1	(DL, 01) (C, 1)
	–		–
	–		–
	–		–

Fig. 1.10.6 : Intermediate code using variant II

Comparison

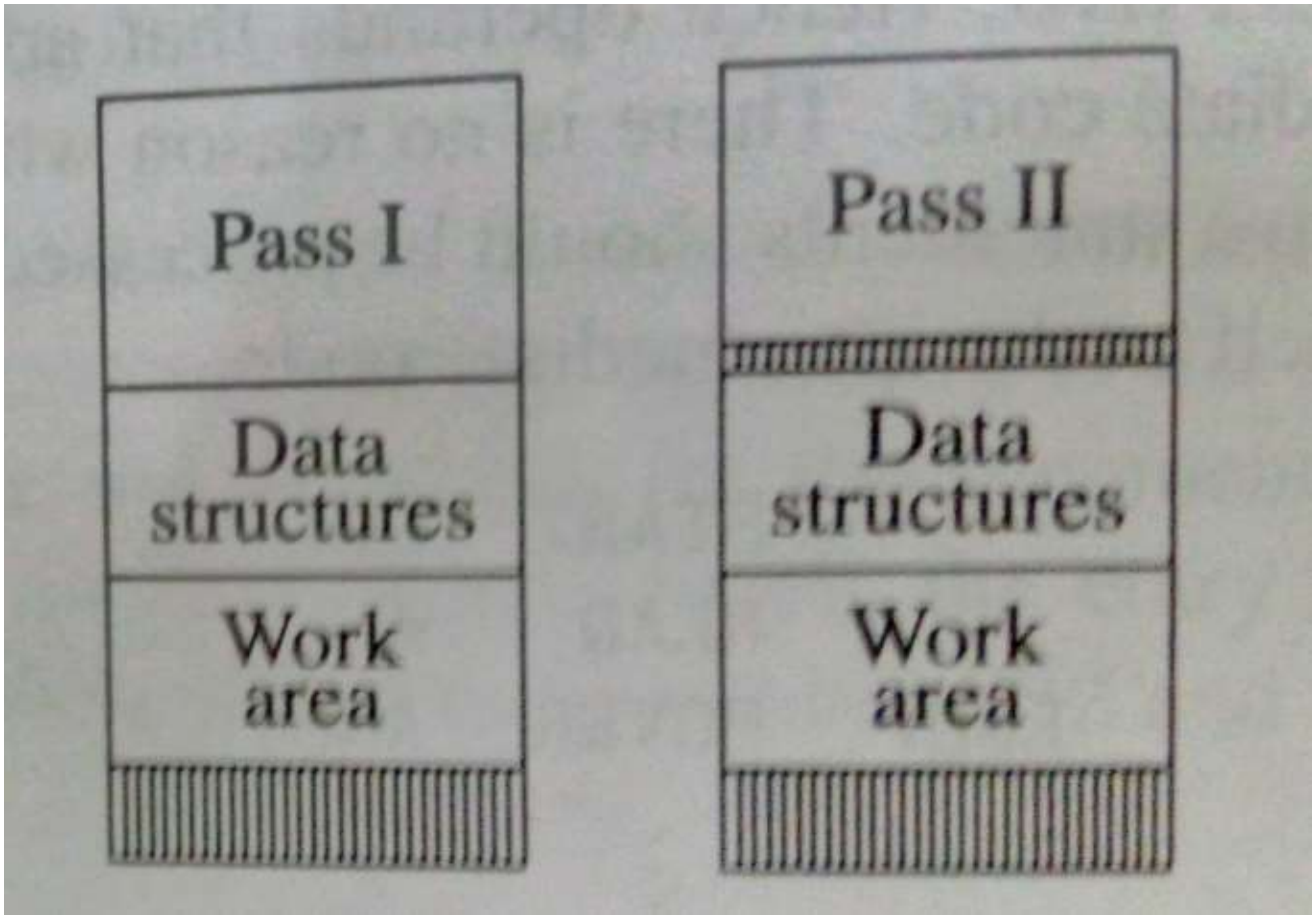
- Variant I does more work in Pass I. Operands fields are completely processed in Pass 1. Memory requirements are higher in Pass 1.
- Variant II, Pass 2 has to do more work. Here the processing requirement is evenly distributed over two passes.
- In Variant II over all memory requirement of the assembler is lower.

Memory requirement in Variant 1



(a)

Memory requirement in Variant 2



Error Reporting

- An assembly program may contain errors.
- It may be necessary to report these errors effectively.
- Some errors can be reported at the end of the source program.
- Some of the typical programs include:
- Syntax errors like missing commas…
- Invalid opcode
- Duplicate definition of a symbol.
- Undefined symbol
- Missing START statement.

Example

- START 100
- MOVER AREG, X
- ADDER BREG, X
- ADD AREG, Y
- X DC ‘2’
- X DC ‘3’
- Z DC ‘3’
- END

- START 100
- MOVER AREG, X
- **ADDER** BREG, X **Invalid opcode**
- ADD AREG, Y Undefined symbol Y
- X DC ‘2’
- X DC ‘3’ duplicate definition of Symbol X.
- Z DC ‘3’
- END

Assignment 1

3. Given the following source program:

```
A        START    100
         DS       3
L1       MOVER    AREG, B
         ADD      AREG, C
         MOVEM    AREG, D
D        EQU      A+1
L2       PRINT    D
         ORIGIN   A-1
C        DC       '5'
         ORIGIN   L2+1
         STOP
B        DC       '19'
         END      L1
```

(a) Show the contents of the symbol table at the end of Pass I.

(b) Explain the significance of EQU and ORIGIN statements in the program and explain how they are processed by the assembler.

(c) Show the intermediate code generated for the program.

2. Explain following Assembler Directives with example.

    1. ORIGIN
    2. LTORG
    3. EQU

3. Write a short note on Assembly language statements.

4. Explain the difference between Pass 1 and Pass2 assembler.

5. Draw and explain flowchart of pass 1 and pass2 assembler.

6. Write a short note on Variant 1 and Variant 2 of Intermediate code.

7. Refer the program from question no. 1 and write down the Intermediate code and machine code for the same.