Language Processor and Compiler Construction (LPCC) – Introduction, Logistics, Assignment 1 Introduction

CSUA31201 : Language Processor and Compiler Construction

Teaching Scheme	**Examination Scheme**
Credits : 4	Continuous Evaluation(CE): 20 Marks
Lectures : 3 Hrs/week	In-Semester Examination(ISE): 30 Marks
Practical : 2 Hrs/week	Skills & Competency Exam(SCE): 20 Marks
	End Semester Examination(ESE): 30 Marks
	PR/OR: 25 Marks

Introduction

Prerequisites :

- Computer Organization and Architecture.
- Processor Architecture and Interfacing.
- Data Structures
- Theory of Computation: DFA, NFA, Regular expressions, Grammars.

Course Objectives :

- To introduce language processing fundamentals and assemblers.
- To explain design of macro processors.
- To introduce compiler design process
- To explain working of syntax analyser.
- To explain importance of semantic analysis and intermediate code representation
- To introduce different code optimization methods

...contd..Introduction

Course Outcomes :

After completion of the course, student will be able to

1. Learn language processing fundamentals with detail designing of assembler.
2. Design macro processors , linkers and loaders.
3. Implement lexical analyser using LEX tool.
4. Understand working of parser and use YACC tool for generation of syntax analyzer.
5. Construct the intermediate code representations
6. Demonstrate code optimization and code generation concept

Unit- I,II,III

Unit I : Introduction To Systems Programming And Assemblers

Introduction: Need of System Software, Components of System Software, Language Processing Activities, Fundamentals of Language Processing, Interpreter

Assemblers: Elements of Assembly Language Programming, A simple Assembly Scheme, Pass structure of Assemblers, Design of Two Pass Assembler.

Unit II : Macroprocessors, Loaders And Linkers

Macro Processor: Macro Definition and call, Macro Expansion, Nested Macro Calls and definition, Advanced Macro Facilities, Design of two-pass Macro Processor.

Loaders: Loader Schemes, Compile and Go, General Loader Scheme, Absolute Loader Scheme, Subroutine Linkages, Relocation and linking concepts, Self-relocating programs, Relocating Loaders, Direct Linking Loaders, Overlay Structure. Linkers.

Unit III : Introduction To Compilers

Phase structure of Compiler and entire compilation process. Lexical Analyzer: The Role of the Lexical Analyzer, Input Buffering. Specification of Tokens, Recognition Tokens, Design of Lexical Analyzer using Uniform Symbol Table, Lexical Errors.

LEX: LEX Specification, Generation of Lexical Analyzer by LEX.

Unit – IV,V,V

Unit IV : Parsers

Role of parsers, Classification of Parsers: Top down parsers- recursive descent parser and predictive parser (LL parser), Bottom up Parsers – Shift Reduce parser, LR parser.

YACC specification and Automatic construction of Parser (YACC).

Unit V : Semantic Analysis And Intermediate Code Generation

Need, Syntax Directed Translation, Syntax Directed Definitions, Translation of assignment Statements, iterative statements, Boolean expressions, conditional statements, Type Checking and Type conversion.

Intermediate Code Formats: Postfix notation, Parse and syntax trees, Three address code, Quadruples and triples.

Unit VI : Code Generation And Optimization

Code Generation: Code generation Issues. Basic blocks and flow graphs, A Simple Code Generator.

Code Optimization: Machine Independent: Peephole optimizations: Common Sub-expression elimination, Removing of loop invariants, Induction variables and Reduction in strengths, Use of machine idioms, Dynamic Programming Code Generation.

Machine dependent Issues: Assignment and use of registers

Books

Text Books :

1. D. M. Dhamdhere, Systems Programming and Operating Systems, Tata McGraw-Hill, ISBN 13:978-0-07-463579-7, Second Revised Edition
2. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers Principles, Techniques and Tools, Addison Wesley, ISBN:981–235–885 - 4, Low Price Edition
3. John R. Levine, Tony Mason & Doug Brown, "Lex & Yacc", O'Reilly

Reference Books :

1. J. J. Donovan, Systems Programming, McGraw-Hill, ISBN 13:978-0-07-460482-3, Indian Edition

List of Assignments

1. Generate Symbol table, Literal table, Pool table & Intermediate code along with error table for first pass of a two-pass Assembler for the given source code.
2. Implement second pass of a two-pass Assembler and generate machine language code for the given intermediate code.
3. Design suitable data structures & implement first pass of a two-pass Macro processor
4. Design suitable data structures & implement second pass of a two-pass Macro processor

5. Write a program to implement a lexical analyzer for parts of speech.
6. Write a program to evaluate arithmetic expression, built-in functions and variables using Yacc specification.
7. Write a program to generate three address code for simple expression.
8. Write a program to apply various code optimization techniques for given three address code.

Unit- I,II,III

Unit I : Introduction To Systems Programming And Assemblers

Introduction: Need of System Software, Components of System Software, Language Processing Activities, Fundamentals of Language Processing, Interpreter

Assemblers: Elements of Assembly Language Programming, A simple Assembly Scheme, Pass structure of Assemblers, Design of Two Pass Assembler.

Unit II : Macroprocessors, Loaders And Linkers

Macro Processor: Macro Definition and call, Macro Expansion, Nested Macro Calls and definition, Advanced Macro Facilities, Design of two-pass Macro Processor.

Loaders: Loader Schemes, Compile and Go, General Loader Scheme, Absolute Loader Scheme, Subroutine Linkages, Relocation and linking concepts, Self-relocating programs, Relocating Loaders, Direct Linking Loaders, Overlay Structure. Linkers.

Unit III : Introduction To Compilers

Phase structure of Compiler and entire compilation process. Lexical Analyzer: The Role of the Lexical Analyzer, Input Buffering. Specification of Tokens, Recognition Tokens, Design of Lexical Analyzer using Uniform Symbol Table, Lexical Errors.

LEX: LEX Specification, Generation of Lexical Analyzer by LEX.

What is an Assembler?

- General Definition: Program that takes input as **basic** computer instructions (Assembly Language instructions), and converts them into machine code (pattern of bits) that the computer's processor can use to perform its basic operations

- (The task of TRANSLATION -> from ALP code to Machine Code)

Elements of Assembly Language Programming

Mnemonics [Based on Underlying Architecture], Symbolic Data, Literals and Constants, Addressable memory, Instruction Lengths

Statement Types – Imperative, Declarative, Assembler Directives

Assembly Language

- **Low level language** - Coding of problem is at instruction level
- **Basic features**
  - Operation Codes **(Mnemonics) based on underlying architecture**
  - Symbolic Operands (**Labels, Variables**)
  - Assembler Directives
  - Data Declaration (**DC,DS**)
  - Registers - **based on underlying architecture**
  - Addressable memory - **based on underlying architecture**
  - Instruction Lengths- **based on underlying architecture**

- **Samples -**

```
LABEL   OPCODE      OPERAND
        MOVER       AREG, X
X       DS          1
L1      ADD         CREG,='80'
```

Assembly Language Statements

- **Imperative (executable)**
  - E.g.
  - MOVER BREG,X
  - STOP
  - READ X
  - PRINT Y
  - ADD AREG,Z
  - BC NE,L1
- **Declarative (Reserving Memory for variable)**
  - **DS : Declare Storage, DC: Declare Constant**

  E.g.
  - X DS 1
  - Y DS 5
  - TWO DC '2'
- **Assembler Directives**
  - Instruct the Assembler, to perform an action during assembling; directive gives direction to assembler; **Pseudo instructions-** no machine code is generated generally

  - START <constant>
  - END
  - LTORG
  - ORIGIN
  - EQU

A Simple Assembly Scheme

(Hypothetical Assembly Scheme)

Mnemonics, Registers, Assembler Directives, Data Declaration, Addressable Memory

Hypothetical Machine – For Simple Assembly Scheme

- **Functional Units** capable of : **Adding, Subtracting, Multiplying, Dividing, Comparing, Branch on conditon**
- 3 **Registers** : **AREG,BREG,CREG**
- **Instruction size** and Format = 1 memory word
  - Format

LABEL	MNEMONIC	REGISTER	REG/SYMBOL/Literal

- **Machine code form** : **Decimal**

Assembly Language Statements

- **Imperative : 11 Machine Operations (STOP, ADD, SUB, MULT, DIV, MOVER, MOVEM, READ, PRINT, COMP, BC)**
  - E.g.
  - MOVER BREG,X
  - STOP
  - READ X
  - PRINT Y
  - ADD AREG,Z
  - BC NE,L1
- **2 Declarative statements Types (DS, DC)**
  - **DS : Declare Storage, DC: Declare Constant**

  E.g.
  - X DS 1
  - Y DS 5
  - TWO DC '2'
- **5 (Basic) Assembler Directives**
  - Instruct the Assembler, to perform an action during assembling; directive gives direction to assembler; **Pseudo instructions-** no machine code is generated generally
  - START <constant>
  - END
  - LTORG
  - ORIGIN <expression>
  - <symbol > EQU <expression/numeric constant>

Literals and Constants in the Assembly Language

LITERALS in Assembly Language

ADD AREG, =‘6’

Handling literal

LABEL	OPCODE	OPERANDS
	ADD	AREG, assembLOC
assembLOC	DC	‘6’ *{Similar to DC }*

CONSTANTS in Assembly Language

LABEL	OPCODE	OPERANDS
TWO	DC	‘2’

Mnemonic operation code (Mnemonic opcode)

Instruction Opcode	Mnemonic
00	STOP
01	ADD
02	SUB
03	MULT
04	MOVER
05	MOVEM
06	COMP
07	BC
08	DIV
09	READ
10	PRINT

Code for declaration statements and directives

Declaration statements	
DC	01
DS	02

Assembler directives	
START	01
END	02
ORIGIN	03
EQU	04
LTORG	05

Sample assembly program and its Translation *[will be explained in the coming slides]*

```
 1          START   200
 2          MOVER   AREG, ='5'          200)  +04 1 211
 3          MOVEM   AREG, A             201)  +05 1 217
 4   LOOP   MOVER   AREG, A             202)  +04 1 217
 5          MOVER   CREG, B             203)  +05 3 218
 6          ADD     CREG, ='1'          204)  +01 3 212
 7          ...

12          BC      ANY, NEXT           210)  +07 6 214
13          LTORG
                    ='5'                211)  +00 0 005
                    ='1'                212)  +00 0 001
14          ...
15   NEXT   SUB     AREG, ='1'          214)  +02 1 219
16          BC      LT, BACK            215)  +07 1 202
17   LAST   STOP                        216)  +00 0 000
18          ORIGIN  LOOP+2
19          MULT    CREG, B             204)  +03 3 218
20          ORIGIN  LAST+1
21   A      DS      1                   217)
22   BACK   EQU     LOOP
23   B      DS      1                   218)
24          END
25                  ='1'                219)  +00 0 001
```

Meaning of Mnemonics

Imperative (executable) Statements

1. AREG 2. BREG 3. CREG

Our machine supports 11 different operations :

1. STOP – to stop execution
2. ADD – operand1 ← operand1 + operand2
3. SUB – operand1 ← operand1 – operand2
4. MULT – operand1 ← operand1 * operand2
5. MOVER – CPU-register ← memory operand
6. MOVEM – Memory operand ← CPU-register
7. COMP – Set condition code, these condition codes will be used by the conditional branch instruction **BC**.
8. BC – Branch on condition. Conditions to be used for branching are :
   a) EQ – equal
   b) NE – not equal
   c) LT – less than
   d) GT – greater than
   e) LE – less or equal
   f) GE – greater or equal
   g) ANY
9. DIV – operand1 ← operand1 / operand2
10. READ – operand2 ← input value
11. PRINT – output ← operand2

– First operand is always a CPU register

– Second operand is always a memory operand

– READ and PRINT instructions do not use the first operand

– The stop instruction has no operand.

Symbolic opcode (mnemonic)	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

EQ	01
NE	02
LT	03
GT	04
LE	05
GE	06
ANY	07

Declarative (Reserving Memory for variable) Statements

DS : Declare Storage: Reserve the specified amount of memory locations

DC : Declare Constant: Reserve 1 memory location and store the specified value in that location

E.g.

- X **DS** **1**
- Y **DS** **5**
- TWO **DC** **'2'**

Assembler Directives

- Instruct the Assembler, to perform an action during assembling; directive gives direction to assembler
- **Pseudo instructions-** no machine code is generated generally
  - START <constant> (Sample : START 500)
  - END { End of program, and Allocate storage for literals, if any }
  - LTORG {Allocate storage for literals }
  - ORIGIN { Update the location counter- LC}
  - EQU {Update the numeric value of LHS with that of RHS}

Important **Input** Data Structures

MOT, POT / **EMOT**

MOT, POT, EMOT

- MOT – Machine Operation Table
  - Contains symbolic mnemonic and its corresponding (Decimal) machine code, and the length (1 here) and format (same for all instructions)

- POT – Pseudo-Operation Table – Contains the pseudo mnemonic (Like assembler directive), and the corresponding action to be taken (*not used here*)
- EMOT= POT+MOT *(version of a simple scheme is in next slide)*

EMOT contains all (MOT,POT)

MNEMONICS, Registers, Assembler Directives, Data Declaration = Enhanced Machine Operation Table (EMOT)

Meaning of CLASS in EMOT

Type	SYMBOL	VAL. OF CLASS FIELD
Imperative Statements	IS	1
Declarative Statements	DL	2
Assembler Directive	AD	3
CPU Register	RG	4
Conditional codes	CC	5

EMOT

Mnemonic	Class	Opcode
STOP	1	00
ADD	1	01
SUB	1	02
MULT	1	03
MOVER	1	04
MOVEM	1	05
COMP	1	06
BC	1	07
DIV	1	08
READ	1	09
PRINT	1	10
START	3	01
END	3	02
ORIGIIN	3	03
EQU	3	04
LTORG	3	05
DS	2	01
DC	2	02
AREG	4	01
BREG	4	02
CREG	4	03
EQ	5	01
LT	5	02
GT	5	03
NE	5	04
LE	5	05
GT	5	06
ANY	5	07

Sample INPUT – OUTPUT in a SIMPLE ASSEMBLY SCHEME

Tasks for Converting Assembly Code to M/c code

I. REPLACE MNEMONICS with machine OPCODES
II. FIND ADDRESSES of SYMBOLS (VARIABLES and LABELS)
III. REPLACE SYMBOLS with the found ADDRESSES
IV. RESERVE STORAGE for DATA

```
          START   101
          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
```

Location counter (LC)	Assembly program		
		START	101
101		READ	X
102		READ	Y
103		MOVER	AREG, X
104		ADD	AREG, Y
105		MOVEM	AREG, RESULT
106		PRINT	RESULT
107		STOP	
108	X	DS	1
109	Y	DS	1
110	RESULT	DS	1
		END	

Variable	Address
X	108
Y	109
RESULT	110

Observations

- There are some **instructions for Assembler**
  - These need not be converted to machine code for execution (but assembler has to perform task)
- Instructions have reference to variables not declared yet (Forward Referencng)
- Tasks for conversion are as follows :

I. **REPLACE MNEMONICS with machine OPCODES**
II. **FIND ADDRESSES of SYMBOLS (VARIABLES and LABELS)**
III. **REPLACE SYMBOLS with ADDRESSES**
IV. **RESERVE STORAGE for DATA**

RECAP: Hypothetical Assembly Language

11 Machine Operations

1. AREG 2. BREG 3. CREG

Our machine supports 11 different operations :

1. STOP – to stop execution
2. ADD – operand1 ← operand1 + operand2
3. SUB – operand1 ← operand1 – operand2
4. MULT – operand1 ← operand1 * operand2
5. MOVER – CPU-register ← memory operand
6. MOVEM – Memory operand ← CPU-register
7. COMP – Set condition code, these condition codes will be used by the conditional branch instruction **BC**.
8. BC – Branch on condition. Conditions to be used for branching are :
   a) EQ – equal
   b) NE – not equal
   c) LT – less than
   d) GT – greater than
   e) LE – less or equal
   f) GE – greater or equal
   g) ANY
9. DIV – operand1 ← operand1 / operand2
10. READ – operand2 ← input value
11. PRINT – output ← operand2

– First operand is always a CPU register

– Second operand is always a memory operand

– READ and PRINT instructions do not use the first operand

– The stop instruction has no operand.

Sample Operations (with 3 registers AREG,BREG,CREG)

Statement	Meaning
ADD AREG, X	AREG ← AREG + X
MOVER BREG, X	BREG ← X
MOVEM CREG, X	X ← CREG
COMP AREG, Y	Compare AREG and the memory variable Y and set the necessary condition codes. These condition codes are required by **BC** instruction.

Machine Operation Table (MOT)

Symbolic opcode (mnemonic)	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

Translating (Assembling) a Sample Program

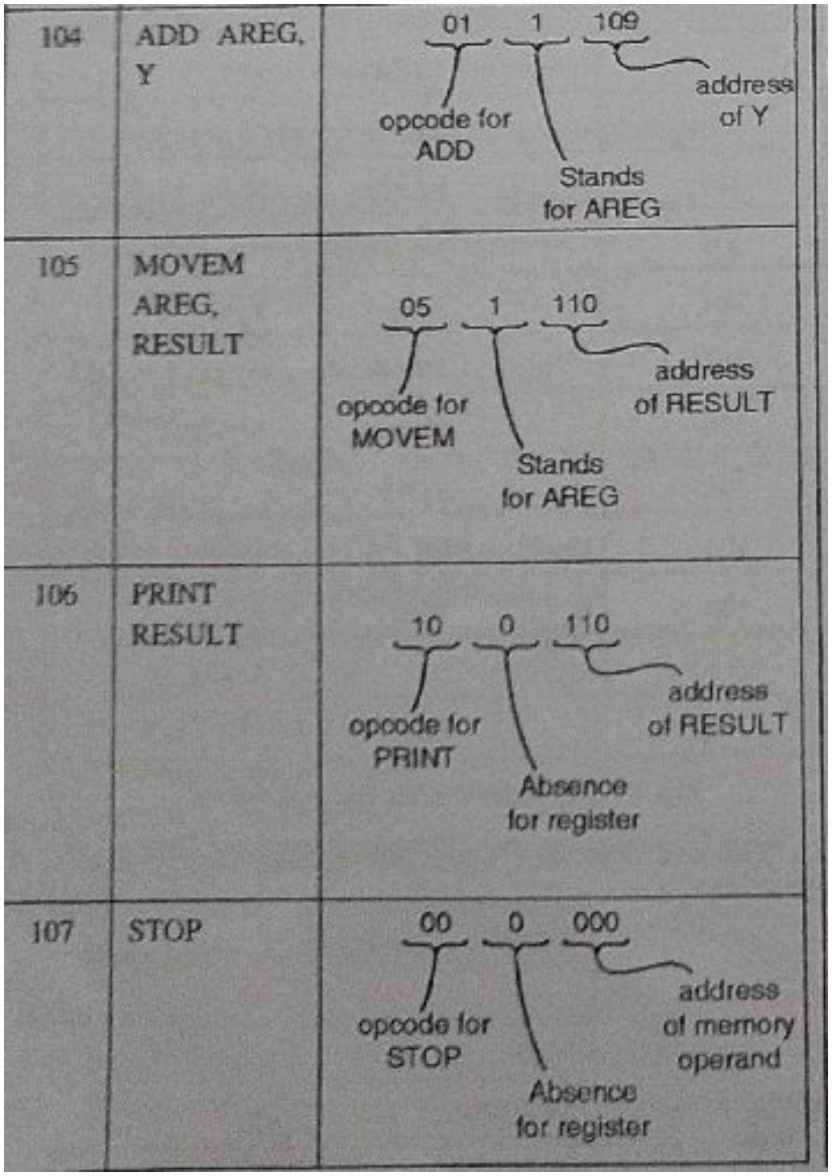
```
          START   101
          READ    X
          READ    Y
          MOVER   AREG, X
          ADD     AREG, Y
          MOVEM   AREG, RESULT
          PRINT   RESULT
          STOP
X         DS  1
Y         DS  1
RESULT    DS  1
          END
```

Step 2 : Generation of machine code

LC	Assembly Instruction	Machine code
101	READ X	09 0 108 — 09: opcode for READ as declared in table 2.1.1; 0: Absence of register operand is shown by 0; 108: Address of X
102	READ Y	09 0 109 — 09: opcode for READ; 0: Register operand is missing; 109: address of Y
103	MOVER AREG, X	04 1 108 — 04: opcode for MOVER; 1: Stands for AREG; 108: address of X

…contd…Translating (Assembling) a Sample Program

```
            START   101
            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
```



Completely translating (Assembling) a Sample Program

Step 2 : Generation of machine code

LC	Assembly Instruction	Machine code
101	READ X	09 0 108 — opcode for READ as declared in table 2.1.1; Absence of register operand is shown by 0; Address of X
102	READ Y	09 0 109 — opcode for READ; Register operand is missing; address of Y
103	MOVER AREG, X	04 1 108 — opcode for MOVER; Stands for AREG; address of X
104	ADD AREG, Y	01 1 109 — opcode for ADD; Stands for AREG; address of Y
105	MOVEM AREG, RESULT	05 1 110 — opcode for MOVEM; Stands for AREG; address of RESULT
106	PRINT RESULT	10 0 110 — opcode for PRINT; Absence for register; address of RESULT
107	STOP	00 0 000 — opcode for STOP; Absence for register; address of memory operand

LC	Assembly Instruction	Machine code
108	X DS 1	Memory is reserved but no-code is generated.
109	Y DS 1	
110	RESULT DS 1	

Thus the required machine code will be :

LC	Opcode	Register	Address
101	09	0	108
102	09	0	109
103	04	1	108
104	01	1	109
105	05	1	110
106	10	0	110
107	00	0	000
108			
109			
110			

Sample Input – Output of the Assembler

LC	Assembly Instruction	M/C code Opcode	Reg	Address
101	READ X	09	0	108
102	READ Y	09	0	109
103	MOVER AREG, X	04	1	108
104	ADD AREG, Y	01	1	109
105	MOVEM AREG, RESULT	05	1	110
106	PRINT RESUT	10	1	110
107	STOP	00	0	000
108	X DS 1			
109	Y DS 1			
110	RESULT DS 1			

Symbolic opcode (mnemonic)	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

Variable	Address
X	108
Y	109
RESULT	110

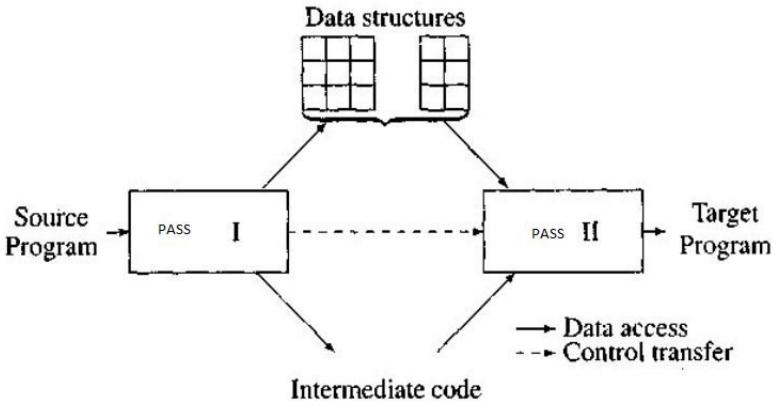
PASS STRUCTURE of ASSEMBLER

1 Pass VS 2 PASS assembler

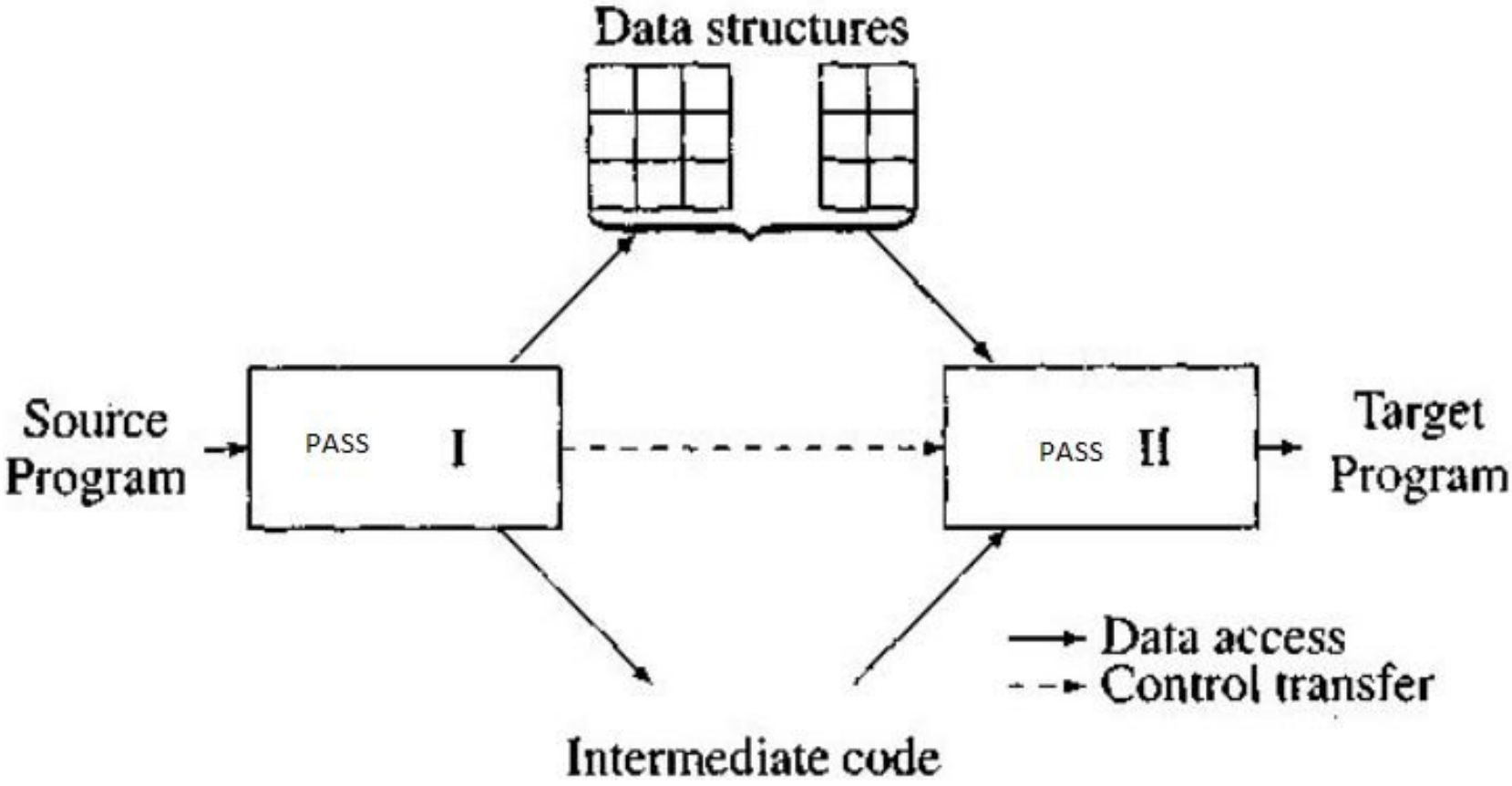
Meaning of PASS (wrt Assembler)

- Pass = going through **source code** (=**input program**), from first to last line, for processing the source code
- 1 pass assembler = Assembler which needs to go through source code **only one time**, to process the source code
- 2 pass assembler = Assembler which needs to go through source code two times, to correctly process the source code

Overview of two pass assembler



Overview of two pass assembler



Need for 2 Pass

- Need for 2 Pass is due to
  - **Forward Referencing**
    - Forward referencing = referencing a variable/label before it is defined
  - **Use of LITERALS and CONSTANTS**

Need for 2 Pass - Forward Referencing

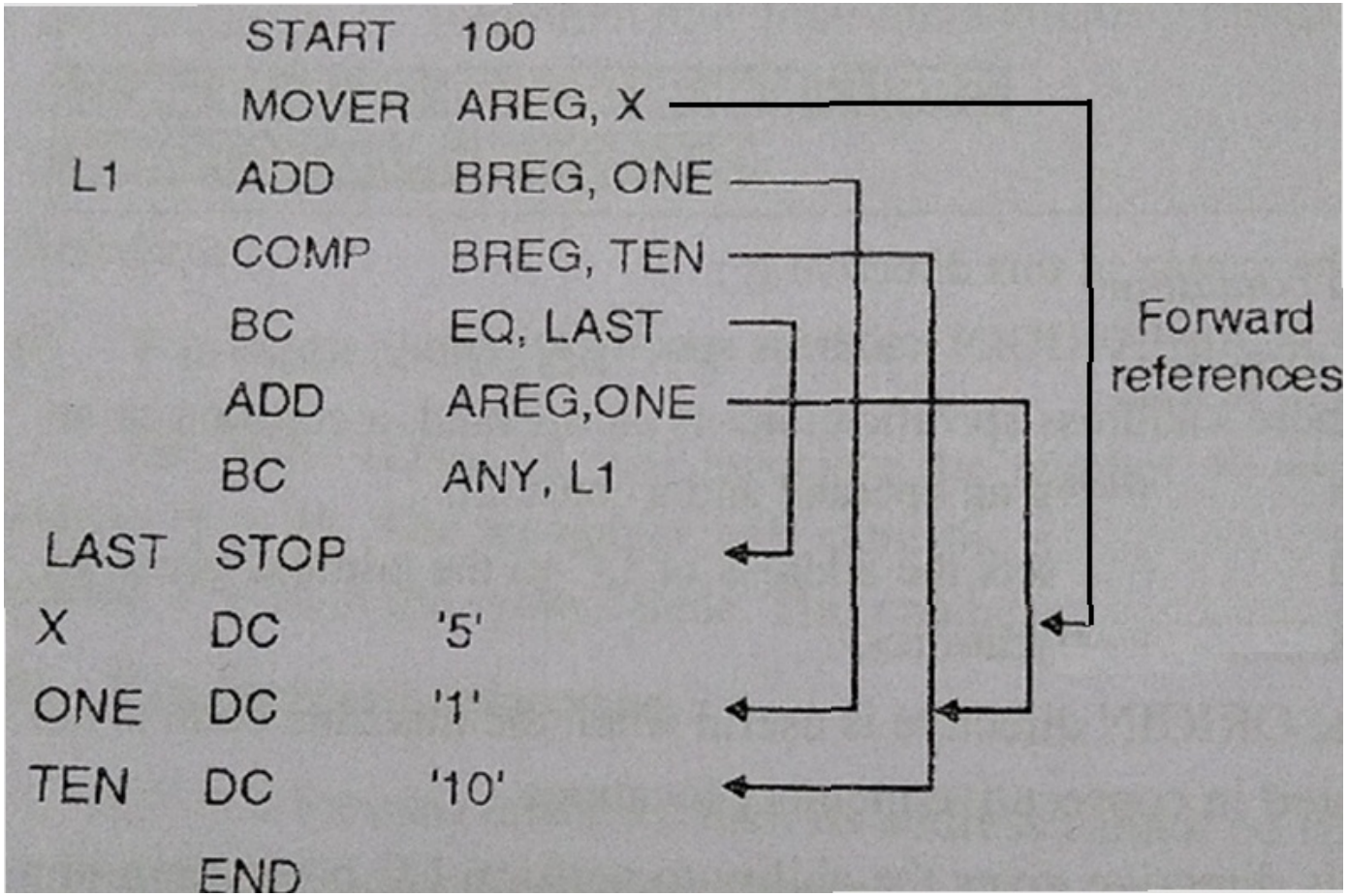
Forward Referencing (FR)

Forward Referencing (FR) = Using a variable before it is declared

```
LABEL   OPCODE     OPERANDS
        START   100
        MOVER     AREG, X
        ----------------------
X       DC        '1'
```

FR is solved with BACKPATCHING

Forward Referencing (Sample code)



Need for 2 Pass - Forward Referencing

Solution – Symbol Table (ST) in Pass1, and Address resolution in Pass 2

Need for 2 Pass - **Use of LITERALS and CONSTANTS in program**

Literals and Constants in the Assembly Language

LITERALS in Assembly Language

ADD AREG, =‘6’

Handling literal

LABEL	OPCODE	OPERANDS
	ADD	AREG, assembLOC
assembLOC	DC	‘6’ *{Similar to DC }*

CONSTANTS in Assembly Language

LABEL	OPCODE	OPERANDS
TWO	DC	‘2’

Need for 2 Pass - **Use of LITERALS and CONSTANTS in program**

SOLUTION = Literal and Pool Tables in Pass 1, and defining constants in Pass2

Design of a 2 Pass Assembler

Assembler Tasks in Pass1

- **Address Calculation**
  - Calculates the addresses of the instructions
    - Calculates the size of each instruction
  - Assigns addresses to symbols (labels and variables)
    - Calculates location of each symbol in memory
- **Generate Intermediate Code (IC)**
  - **Form** - Symbol table or an intermediate file
  - IC includes information about the address and size of each instruction and the location of symbols
- **Handle Directives**
  - Process assembler directives, such as defining constants, allocating storage space, or specifying the start and end of the program

Assembler TASKS in Pass 2

- **Code Generation**
  - Using the information obtained from the first pass, the assembler generates the actual machine code instructions.
  - It converts the assembly language instructions and addresses into binary machine code
- **Resolve Addresses**
  - The assembler resolves the addresses of symbolic references, such as labels, by substituting the actual addresses determined in the first pass.
- **Generate Output**
  - The final executable code or an object file is produced as output

Building a Simple Assembler

- Simple Assembler = Program (say written in C++/Java/Python)
- Input = Text file containing Assembly language Program
  - Fixed Hypothetical Assembly Language corresponding to a Hypothetical Machine
    - Hypothetical machine = Operations it can perform, number of registers, Instruction length, Memory
- Output= Text file containing assembled Input Program
  - Each line of input (instruction) converted to Numeric

- Note: Assumptions will have to be kept in mind

Build Assembler : 2 Pass Assembler

Requirements:

1) Select (Fix) a Hypothetical Assembly Language, for a hypothetical underlying machine

2) **Understand** the assembly language and program features (for translation/assembling purpose)

3) Understand tasks to be performed / Algorithm for Pass-I , II

4) Implementation using C++, Java, Python

1) Selecting (fixing) a Hypothetical Assembly Language

Hypothetical Assembly Language

Fixing the requirements (Hypothetical Assembly Language):

- Assembly Language m/c instructions –> **11**
- Assembler Directives -> **5** (minimum)
- Data handling/memory reserving instructions -> **2**
- Registers –> **3**
- Instruction Length -> **1 Word**
- Numeric **Machine Code** -> **Decimal**
- Memory Address -> **000** to **999**

Hypothetical Assembly Language Features : Operation Codes (Mnemonics)

11 Machine Operations

1. AREG 2. BREG 3. CREG

Our machine supports 11 different operations :

1. STOP – to stop execution
2. ADD – operand1 ← operand1 + operand2
3. SUB – operand1 ← operand1 – operand2
4. MULT – operand1 ← operand1 * operand2
5. MOVER – CPU-register ← memory operand
6. MOVEM – Memory operand ← CPU-register
7. COMP – Set condition code, these condition codes will be used by the conditional branch instruction **BC**.
8. BC – Branch on condition. Conditions to be used for branching are :
   a) EQ – equal
   b) NE – not equal
   c) LT – less than
   d) GT – greater than
   e) LE – less or equal
   f) GE – greater or equal
   g) ANY
9. DIV – operand1 ← operand1 / operand2
10. READ – operand2 ← input value
11. PRINT – output ← operand2

– First operand is always a CPU register
– Second operand is always a memory operand
– READ and PRINT instructions do not use the first operand
– The stop instruction has no operand.

Sample Operations (with 3 registers AREG,BREG,CREG)

Statement	Meaning
ADD AREG, X	AREG ← AREG + X
MOVER BREG, X	BREG ← X
MOVEM CREG, X	X ← CREG
COMP AREG, Y	Compare AREG and the memory variable Y and set the necessary condition codes. These condition codes are required by **BC** instruction.

Machine Operation Table (MOT)

Symbolic opcode (mnemonic)	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

Hypothetical Assembly Language Features : Assembler Directives **(For Assembler for help in Translation only)**

They Instruct Assembler to perform an action during assembling; directive gives direction to assembler; **Pseudo instruction-** no machine code is generated generally

- **START <constant>**
- **END**
- **LTORG**
- **ORIGIN**
- **EQU**

Hypothetical Assembly Language Features : Data Declaration

(For space allocation for variables and constants)

- Declarative (Reserving Memory for variable)
  - **DS : Declare Storage, DC: Declare Constant**

  E.g.
  - X DS 1
  - Y DS 5
  - TWO DC '2'

Build Assembler : 2 Pass Assembler

Requirements:

1) Select (Fix) a Hypothetical Assembly Language, for a hypothetical underlying machine

2) **Understand** the assembly language and program features (for translation/assembling purpose)

3) Understand tasks to be performed / Algorithm for Pass-I , II

4) Implementation using C++, Java, Python

2) **Understanding** assembly language program features (for purpose of translation/assembling)

Forward Referencing (FR)

Forward Referencing (FR) = Using a variable before it is declared

```
LABEL   OPCODE     OPERANDS
        START   100
        MOVER     AREG, X
        ----------------------
X       DC        '1'
```

FR is solved with BACKPATCHING

Literals and Constants in Assembly Language

LITERALS in Assembly Language

ADD AREG, =‘6’

Handling literal

```
LABEL   OPCODE   OPERANDS
        ADD      AREG, Laddr
        ----------
Laddr   000006
```

CONSTANTS in Assembly Language

```
LABEL   OPCODE   OPERANDS
TWO     DC       ‘2’
```

Build Assembler : 2 Pass Assembler

Requirements:

1) Select (Fix) a Hypothetical Assembly Language, for a hypothetical underlying machine

2) **Understand** the assembly language and program features (for translation/assembling purpose)

3) Understand tasks to be performed / Algorithm for Pass-I , II

4) Implementation using C++, Java, Python

3) Tasks in Translation of Assembly language program/code to machine code

For the Hypothetical Assembly Language

Simple Assembly Scheme – 2 Pass

MOT = Machine Op table; POT = Pseudo-op Table; ST= Symbol Table ; IC = Intermediate Code;

INPUT- Assembly Language Program , MOT, POT, AREG=1,BREG=2,CREG=3

I. Read each Instruction
II. Perform Action as per Opcode / Label to IC, ST
III. Loop to II. Till END

(Output = IC, ST)

V. Read IC Instructions
VI. Translate the Mnemonics, Symbols to OP
VII. Loop to V. Till EOF

Variable	Address
X	108
Y	109
RESULT	110

```
          START   101
          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
```

Location counter (LC)	Assembly program		
		START	101
101		READ	X
102		READ	Y
103		MOVER	AREG, X
104		ADD	AREG, Y
105		MOVEM	AREG, RESULT
106		PRINT	RESULT
107		STOP	
108	X	DS	1
109	Y	DS	1
110	RESULT	DS	1
		END	

LC	Assembly Instruction	M/C code Opcode	Reg	Addr
101	READ X	09	0	108
102	READ Y	09	0	109
103	MOVER AREG, X	04	1	108
104	ADD AREG, Y	01	1	109
105	MOVEM AREG, RESULT	05	1	110
106	PRINT RESUT	10	1	110
107	STOP	00	0	000
108	X DS 1			
109	Y DS 1			
110	RESULT DS 1			

Designing Pass-I , Pass-II of 2 Pass Assembler

Pass-I , Pass-II

- The primary purpose of Pass I is to **collect information** needed for the subsequent Pass II
- Pass-I also generates an intermediate code (called IC/IR) which contains information for Pass-II
- In Pass II, the actual machine code is generated

Pass- I

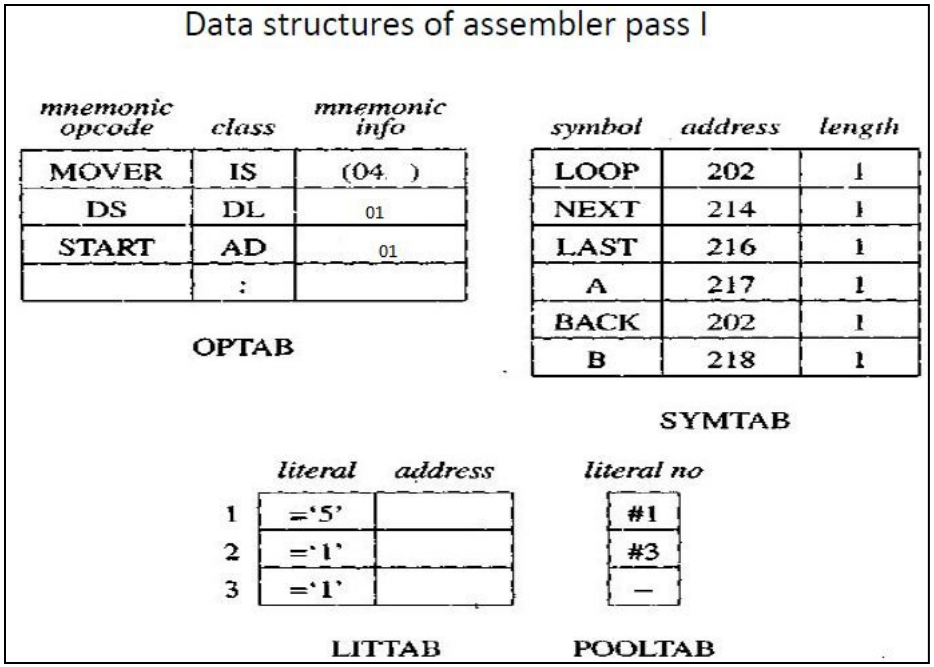
- **Record Symbol Information in Symbol Table (ST) - Generate Symbol Table Entries**
- **Determine Memory Requirements- DC , DS - Reserve Memory Space**
- **Record Literal Information**
- **Generate intermediate Code (IC)**
- Pass I is primarily concerned with collecting information about symbols, constants, and variables declared in the source code. For constants declared using the DC directive, Pass I records their attributes, determines their memory requirements, and updates the symbol table.

Pass-II

- In Pass-I, assembler has a complete understanding of the program's structure
- The actual memory allocation and assignment of addresses occur in Pass II

Data Structures for PASS-I of Assembler

- Location Counter (LC)
- Operation code Table (MOT, POT, EMOT)
- Symbol Table
- Literal Table
- Pool Table



Symbol Table (ST)

- Symbol table is used for keeping the track of symbol that are defined in the program
- It is used to give a location for a symbol specified
- Symbol is said to be defined when it appears in a label field
- In pass 1, whenever a symbol is defined, entry is made in symbol table
- In pass2, symbol table is used for generating address of a symbol

Literal Table

Programmer can directly specify a value - **literal ,** **assembler** is used to give a location

for the value

literal can start with an equal sign (=), which indicates to the assembler that a literal follows

Literals are always encountered in the operand field of an instruction

literals define *read-only* data, they must not be used in operands that represent the receiving field of an instruction that modifies storage

A **literal** both defines data and represents data

The address of the **literal** is assembled into the object code of the instruction in which it is used

In pass 1, whenever a **Literal** is defined an entry is made in Literal table

In pass2, Literal table is used for generating address of a Literal, and assigning the literal value at the address

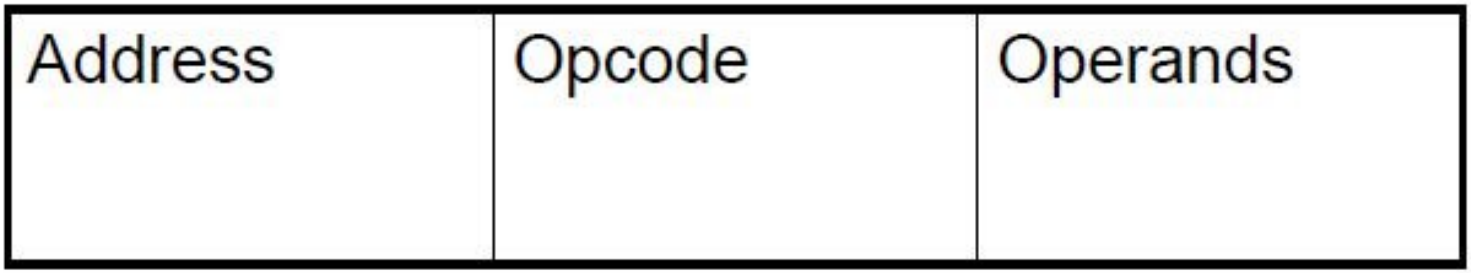
Intermediate Code (IC)

- IC generated in Pass I, is intermediate code between source code and the final machine code.
- It includes essential information about the program's structure, symbols, instructions, and memory requirements.
- Pass II then uses this intermediate code to resolve addresses, allocate memory, and generate the final machine code.

An Intermediate code unit

Address	Opcode	Operands

An Intermediate code unit



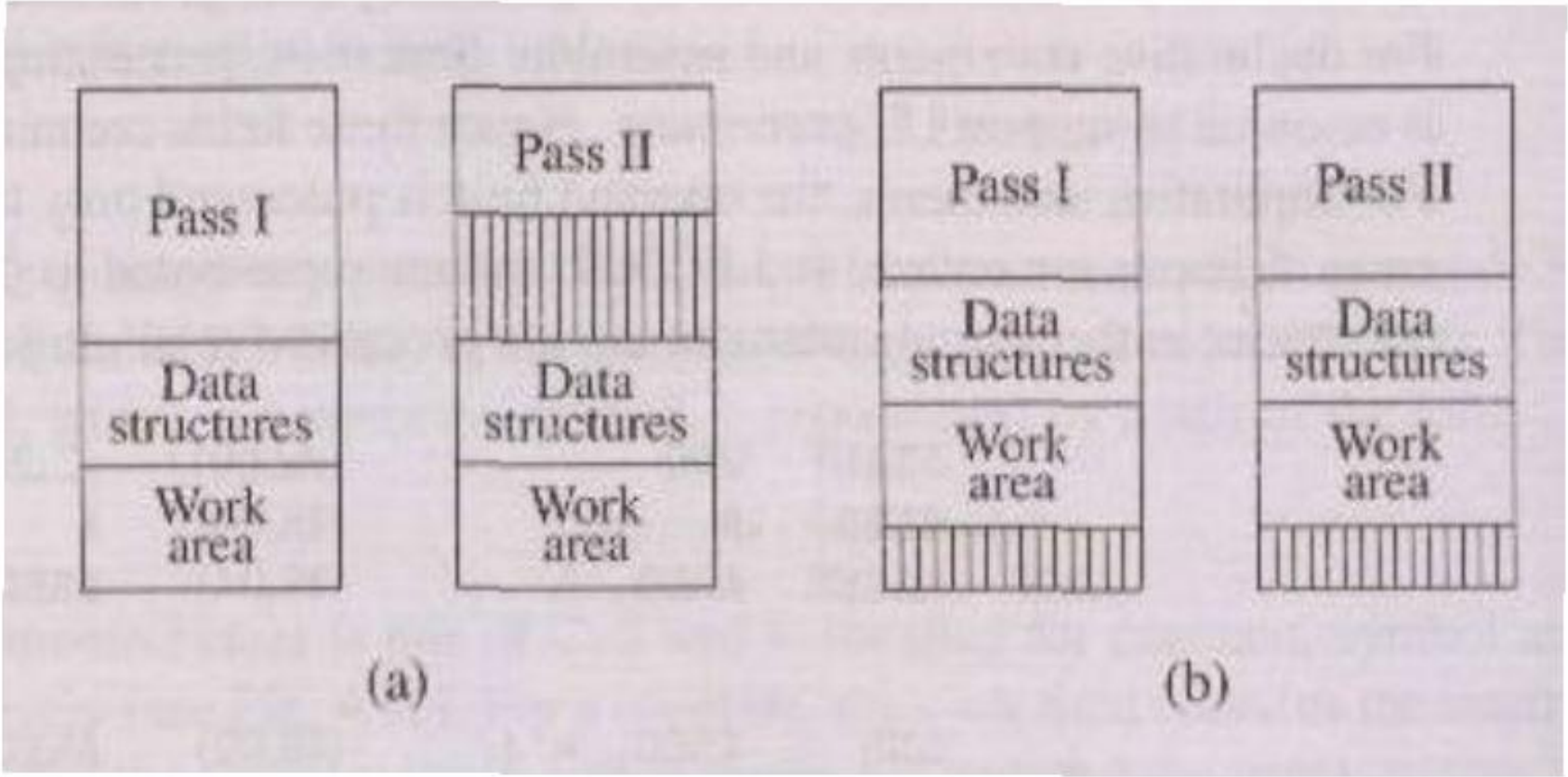
Intermediate code – variant I

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

Intermediate code – variant II

```
        START   200             (AD,01)   (C,200)
        READ    A               (IS,09)   A
LOOP    MOVER   AREG, A         (IS,04)   AREG, A
          :                       :
        SUB     AREG, ='1'      (IS,02)   AREG, (L,01)
        BC      GT, LOOP        (IS,07)   GT, LOOP
        STOP                    (IS,00)
A       DS      1               (DL,02)   (C,1)
        LTORG                   (DL,05)
        ...                     ...
```

Memory requirements using variant I and variant II



Building the Assembler

Understanding (Pass – I) with Advanced Assembler directives

Tasks to be Done(1) ST

```
        START    200                          ////LC=200
        MOVER    AREG,='5'     // LC=200              ////LC=LC+1
        MOVEM    AREG,X        ////LC=201       LC=LC+1
L1      MOVER    BREG,='3'     ////LC=202     LC=LC+1
NEXT        ADD      AREG,='1'     ////LC = 203    LC=LC+1
            SUB      BREG,='5'     ////LC=204    LC=LC+1
        BC       LT, NEXT      ////LC = 205   LC=LC+1
        MULT     CREG, X ////LC=206          LC=LC+1
        STOP                   ////207-----LC=LC+1
X       DS       5     (LC=208)   ////   LC=LC+5=213
            END                        (LC=209) ='5',        209,
            ='3',        210
                                  ='1',        211
```

SYMBOL TABLE

X	208
L1	202
NEXT	203

LITERAL TAB

LT	
'5'	21
'3'	21
'1'	21

Tasks to be Done(2) LT

```
        START          200
        MOVER   AREG,='5'   //lc=200
        MOVEM   AREG,X
L1      MOVER   BREG,='3'     //202    lc=lc+1 (203
        LTORG
                                       0000005 (203)
                                       0000003 (204

NEXT    ADD     AREG,='1'     LC(205)
        SUB     BREG,='2'     LC(206)
        BC      LT, BACK   LC(207)
        LTORG   (LC=208)
                0000001 (208)
                0000002 (209)

        MULT    CREG, X
        STOP
X       DS      1
        END
```

Literal Table

='5'	203
='3'	204
='1'	208
='2'	209

POOL TABLE

1

3

5

Assembler Directives

- START <mem address> 100
- END
- LTORG : Used so that assembler can collect and assemble literals into a literal pool
- ORIGIN : Instructs where to load instructions and data into memory
  - ORIGIN Y+2
  - ORIGIN 250
- EQU : Simply equates a symbolic name to a numeric value
  - Sunday **EQU** 1
  - Monday **EQU** 2
  - Tuesday **EQU** Sunday + 2
    - (What is assigned to Tuesday?)

Advance Assembler Directives

- **LTORG** - Permits programmer to specify where literals should be placed
  - When no LTORG, they are placed at END
- **ORIGIN**
- **EQU**

LTORG (Literal Origin) directive

- LTORG specifies the location of literals in the program
- Literals are constants, and LTORG is typically used in assembly languages that support literals.
- LTORG : Use the LTORG instruction so that the assembler can collect and assemble literals into a literal pool

USE of LTORG:

- LTORG asks the assembler to reserves space for the literals and assign them addresses in the program
- It helps organize the literals in memory by placing them at a specific location, often after the code that uses them
- Aims to improves the readability of the assembly code by separating the code instructions from the literals

LTORG (Example)

```
                START          200              LC=200
LOAD      MOVER        AREG,  A          ----------200
          MOVER        BREG, ='2'        ----------201
          ADD          AREG,  ='2'       ----------202
          SUB          BREG, ='3'        ----------203
          LTORG                          ----------204  (Add of ='2')0000002
                                         ----------205 (Add of ='3')0000003
          ORIGIN  LOAD+10                 -----210
     MOVER         AREG, ='4'        ----------210
          ADD          BREG, ='2'        ----------211
     LTORG                          ----212   ='4'
                                    ---213  ='2'
A         DC         '5'            ----------214
          END
```

Assembler Directives : ORIGIN, EQU

- **ORIGIN** <address specification>
  - **Set LC to address specified by address specification**
  - Use : when machine code is not in consecutive memory locations
    - E.g. **ORIGIN 208**
  - **LC** processing can be Relative (not Absolute)
    - E.g. **ORIGIN LOOP+5**

- <symbol> **EQU** <address specification>
  - **Associate symbol with address**
    - E.g. **X EQU Y+10**

- Equate: The EQU assembler directive simply equates a symbolic name to a numeric value. Consider: Sunday EQU 1
- Monday EQU 2
- You could also write Sunday EQU 1
- Monday EQU Sunday + 1
- In this case, the assembler evaluates "Sunday + 1" as 1 + 1 and assigns the value 2 to the symbolic name "Monday

- Origin : The origin directive tells the assembler where to load instructions and data into memory
- ORG 1024

LTORG, ORIGIN (Example)

```
    START     200                LC
        MOVER   AREG,='7'          200
        MOVER   BREG,X             201
L1  MOVER   BREG, ='1'         202
    ORIGIN    L1+4             206
    LTORG                          206…Addr of  ='7'
                                   207…Addr  of ='1'
NEXT  ADD AREG, ='2'                 208
X   DS    1                        209
    END                            210…..Addr of ='2'
```

Handling LTORG in Pass I

- On encountering the LTORG directive during PASS I, the assembler doesn't necessarily generate machine code or reserve memory for literals at that point.
- Instead, it marks the locations of literals and records them for processing in the second pass.

Sample for Assignment 3 and 4

```
        START     200
        MOVER     AREG,='5'
        MOVEM     AREG,X
L1      MOVER     BREG,='2'

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

        MULT      CREG, X
        SUB       BREG,='2'
        STOP
X       DS        1
        END
```

```
(LC =200)
200
201
202
(LC=203)
(LC=204)
205
206
207
(208,'1')
(209,'2')
210
LC=211
212('00')
213
214
```

SYMBOL TABLE (ST)

	SYMBOL	ADDRESS
0		
1	L1	202
2	NEXT	205
	X	213

LITERAL TABLE (LT)

	VALUE	ADDRESS
0	='5'	203
1	='2'	204
2	='1'	208
3	='2'	209
4	='2'	214

POOL TABLE

	LITERAL NUMBER
0	
1	0
	2

```
        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+9
        MULT      CREG, X
        STOP
X       DS        1
        END
```

(LC =200)
200
201
202
(LC=205)
(LC=205)
207
208
209
(210,'1')
(211,'2')
212
LC=216
216
217
218

SYMBOL TABLE (ST)

	SYMBOL	ADDRESS	
0			
1	L1	202	
2	NEXT	207	
3	BACK	212/202	<-STN
	X	218	

LITERAL TABLE (LT)

	VALUE	ADDRESS	
0	='5'	205	
1	='2'	206	
2	='1'	210	<-LTN
3	='2'	211	

POOL TABLE

	LITERAL NUMBER	
0		
1	0	<-PTN
	2	

For the given assembly code prepare and write down: a) Symbol Table, b) Literal Table and c) Pool Table. **[6 Marks]**

```
START 500
READ X
NEXT MOVER AREG,X
MOVER BREG, Y
ADD AREG, ='2'
MOVEM AREG, X
BACK MOVER CREG, ='2'
ADD AREG, CREG
CMP BREG,AREG
BC GT, NEXT
LTORG
ADD  AREG, ='1'
SUB  BREG,Y
MULT CREG, ='2'
CMP AREG,BREG
BC LT, BACK
LTORG
ADD  AREG,='1'
MULT CREG, X
STOP
X DS 2
Y DC 10
END
```

For the given assembly code prepare and write down: a) Symbol Table, b) Literal Table and c) Pool Table. **[5 Marks]**

Given Input in assembly language:

```
START 200
READ X
READ Y
NEXT MOVER AREG,=’5’
MOVEM AREG, X
MOVER CREG,=’1’
BACK MOVER BREG,=’1’
LTORG
ADD  AREG,=’1’
SUB  BREG,Y
MULT CREG,=’2’
CMP AREG,BREG
BC LT, BACK
LTORG
ADD  AREG,=’1’
MULT CREG, X
STOP
X DS 1
Y DS 1
END
```

An assembly program

```
 1          START   200
 2          MOVER   AREG, ='5'      200)  +04 1 211
 3          MOVEM   AREG, A         201)  +05 1 217
 4  LOOP    MOVER   AREG, A         202)  +04 1 217
 5          MOVER   CREG, B         203)  +05 3 218
 6          ADD     CREG, ='1'      204)  +01 3 212
 7          ...

12          BC      ANY, NEXT       210)  +07 6 214
13          LTORG
                    ='5'            211)  +00 0 005
                    ='1'            212)  +00 0 001
14          ...
15  NEXT    SUB     AREG, ='1'      214)  +02 1 219
16          BC      LT, BACK        215)  +07 1 202
17  LAST    STOP                    216)  +00 0 000
18          ORIGIN  LOOP+2
19          MULT    CREG, B         204)  +03 3 218
20          ORIGIN  LAST+1
21  A       DS      1               217)
22  BACK    EQU     LOOP
23  B       DS      1               218)
24          END
25                  ='1'            219)  +00 0 001
```

Assembler PASS-I

Algorithm : First pass of two pass assembler -1

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.

Algorithm : First pass of two pass assembler -2

(b) If an LTORG statement then

(i) Process literals LITTAB [POOLTAB [pooltab-ptr]]... LITTAB [lit- tab-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;

Algorithm : First pass of two pass assembler - 3

(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 1C '(DL, code) …..'.

Algorithm : First pass of two pass assembler - 4

(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 1C '(IS, code)(S, this-entry)';

Algorithm : First pass of two pass assembler - 5

3. (Processing of END statement)

(a) Perform step 2(b).

(b) Generate 1C '(AD.02)'.

(c) Go to Pass II.

Assembler PASS-II

Algorithm : Second pass of two pass assembler - 1

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;

Algorithm : Second pass of two pass assembler - 2

(b) If an LTORG statement

(i) Process literals in LITTAB[POOLTAB[pooltab-ptr]} ... LITTAB [POOLTAB [pooltab-ptr+l]]—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;

Algorithm : Second pass of two pass assembler - 3

(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;

Algorithm : Second pass of two pass assembler - 4

(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;

Algorithm : Second pass of two pass assembler - 5

3. (Processing of END statement)

(a) Perform steps 2(b) and 2(f).

(b) Write code-area into output file

Thank You