ASSIGNMENT NO.1

Problem Statement: Design suitable data structures and implement pass-I of a two-pass assembler. Implementation should consist of a few instructions from each category and few assembler directives.

Objective(s):

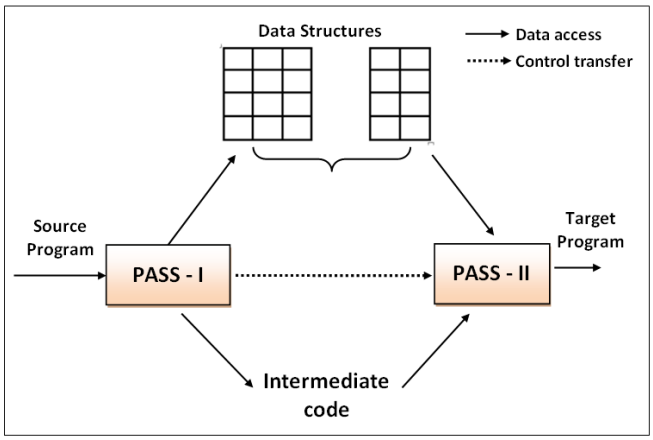
- To study basic translation process of assembly language to machine language.
- Data representation for symbols, literals, and pool table.
- Implement Pass-I to generate Intermediate code for all assembly language statements.

THEORY:

What is Language translator - Assembler?

- A language translator bridges an execution gap to machine language of computer system. An assembler is a language translator whose source language is assembly language.
- Language processing activity consists of two phases, Analysis phase and synthesis phase.
- Analysis of source program generates various data structures and intermediate code.
- Synthesis phase is concerned with construction of target language statements, which have the same meaning as source language statements. This consists of memory allocation and code generation.

Overview of 2 passes of Assembler:



Assembly Language Statements:

- Imperative Statements – Indicates an action to be performed.
- Assembler Directives- Instructs assemblers to perform certain actions. – START <constant>, END
- Declaration Statements - DS (Declare Storage) & DC (Declare Constant)
  - DS reserve areas of memory and associates names with them.

Design of Pass-1 Assembler

Tasks performed:

- Separate the symbol, mnemonic opcode and operand fields.
- Determine the storage-required for every assembly language statement and update the location counter.
- Build the symbol table and the literal table.
- Construct the intermediate code for every assembly language statement.

Data Structures:

- Source file containing assembly program.
- MOT: A table of mnemonic op-codes and related information.
- Symbol Table
- Literal table
- Pool table

Mnemonic Opcode Table (MOT)

Instruction Opcode	Assembly mnemonic	Remarks
00	STOP	stop execution
01	ADD	first operand modified condition code set
02	SUB	
03	MULT	
04	MOVER	register ← memory
05	MOVEM	memory ← register
06	COMP	sets condition code
07	BC	branch on condition code
08	DIV	analogous to SUB
09	READ	first operand is not used.
10	PRINT	

Symbol Table

<Symbol name, Address>.

Initialize all fields to 0.

If symbol gets added when it appears in label field or DS/DC/EQU replace address value with current LC.

SYMBOL_TABLE		
Ind	Sym	Addr
0		
1		
2		
3		

Literal Table and Pool Table

Literal table stores the literals used in the program and POOLTAB stores the pointers to the literals in the current literal pool.

LITERAL TABLE		
Ind	Lit	Addr
0		
1		

POOL TABLE	
Ind	Pool
0	
1	

ERRORS:

Forward reference (Symbol used but not defined): -	This error occurs when some symbol is used but it is not defined into the program.
Duplication of Symbol: -	This error occurs when some symbol is declared more than once in the program.
Mnemonic error:	If there is invalid instruction then this error will occur.
Register error: -	If there is invalid register then this error will occur.
Operand error: -	This error will occur when there is an error in the operand field

SAMPLE INPUT & OUTPUT:

```
    START 200
    MOVER AREG, A
L:  MOVEM BREG, = '2'
    ADD BREG, = '2'
    ADD CREG, = '3'
    ORIGIN L + 20
    LTORG
    MOVER AREG, C
    C EQU L + 15
    ADD AREG, = '2'
    ADD BREG, = '5'
    A DS 5
    END
```

```
(AD, 00) (C, 200)
(IS, 04) (0) (S, 00)
(IS, 05) (1) (L, 00)
(IS, 01) (1) (L, 00)
(IS, 01) (2) (L, 01)
(AD, 02)
(AD, 04)
(IS, 04) (0) (S, 02)
(AD, 03)
(IS, 01) (0) (L, 02)
(IS, 01) (1) (L, 03)
(DL, 01) (C, 05)
(AD, 01)
```

Intermediate Code

Symbol Table

	Symbol	Address
0	A	226
1	L	201
2	C	216

Literal Table

	Literal	Address
0	='2'	221
1	='3'	222
2	='2'	231
3	='5'	232
4		

Pool Table

	Pool
0	0
1	2
2	4

AIM :

Implement Assembler Pass-1 to generate respective data structures and intermediate code.

ALGORITHM(s): Main Module:

Algorithm passOne( )

1. Open .asm file in read mode and store the pointer in fp1
2. Open ic.txt file in write mode and store the pointer in fp2
3. **For** (every line in fp1)

```
        Tokenize the instruction for the respective tokens
        if tokCnt == 1: // STOP, START, LTORG, END
                processTok1(Tok1)
        if tokCnt -== 2: // READ, PRINT, START, ORIGIN
                processTok2(Tok1, Tok2)
        if tokCnt == 3: //ADD-DIV, DS/DC, EQU
                processIS(Tok1, Tok2, Tok3);
                processDL(Tok1, Tok2, Tok3);
                processEQU(Tok1, Tok2, Tok3);
        else tokCnt==4:// INSTRUCTION WITH LABEL
                Update Tok1 address with LC in the symbol table;
                processIS(Tok2, Tok3, Tok4));

        LC++;
```

End For

End passOne

Case 1:

Algorithm processTok1 (Tok1)

1. **SearchOp**(Tok1) in Mnemonics table for its index say i
2. Write Intermediate code for STOP in fp2 as (IS, i)
3. j = **SearchDirective**(Tok1) in the Directive table;
4. **If** (j == 0); Write Intermediate code for START in fp2 and set the LC = 0
5. **Else if**(j == 1 ||j == 4); Write Intermediate code for END or LTORG;
6. For the present pool; Update addresses for literals with LC;
7. Update POOLTAB with the new PoolValue;
8. **End if**

End processTok1

Case 2:

Algorithm processTok2 (Tok1, Tok2)

1. **SearchOp**(Tok1) in Mnemonics table for its index say i
2. Write Intermediate code for READ/PRINT in fp2 as (IS, i)
3. Search Tok2 in symtab for symbol processing;
   if Symbol is NOT present **then;** add new symbol with blank address value in the Symbol table;
   Fetch index of Tok2 and Write IC in fp2 as (S, index)
4. i = **SearchDirective**(Tok1) in the Directive table;
   Write IC in fp2 as (AD, i)
   Set LC = Tok2

End processTok2

Process IS:

Algorithm processIS (T1,T2, T3)

1. Search T1 in Mnemonics table and fetch index in variable i;
2. **if** T3 is **Literal then**
   Search T3 in Literal table for literal processing.
   if Literal is NOT present **then** add new literal with blank address value in the Literal table;
   Fetch index of Tok2 and Write IC as (L, index), in Tok3IC
3. **else**
   Search T3 in symtab for symbol processing;
   if Literal is NOT present **then** add new literal with blank address value in the Literal table;
   Fetch index of Tok2 and Write IC as (L, index) in Tok3IC
4. **end if**
5. Fetch the index of T2(Condition code or register) in variable j;
6. Write IC in fp2 as (IS, i) (j) Tok3IC;

End processIS

Process EQU:

Algorithm processEQU(T1,T2, T3)

1. Search T2 in Directive table to fetch its index, in a variable say i
2. Set the address of T1 and the address of T3
3. write IC as (AD, i)
4. LC-- //As no LC processing for EQU

End processEQU

Process DL:

Algorithm processDL(T1,T2, T3)

1. Set address of T1 as the present value of LC
2. **For DS**: LC += T3 and write IC in fp2 as (DL, 01) (C, T3);
3. **For DC:** write IC in fp2 as (DL, 00) (C, T3);

End processDL

Frequently Asked Questions:

a. Explain what is meant by pass of an assembler.
b. Explain the need for two pass assembler.
c. Explain terms such as Forward Reference and backward reference.
d. Explain various types of errors that are handled in two different passes.
e. Explain the need of Intermediate Code generation and the variants used.
f. How literals are handled in Pass 1 of 2 pass assembler