Name: Rushikesh Kacbhaci Palve
Roll No.: 31258

: Date :	Page No :
/ /20	

Assignment No. 3

DOP: 19-08-2021 **DOS**: 24-08-2021

Problem Definition :

Design suitable data structures and implement pass-I of a two-pass assembler for pseudo-machine in Java. Implementation should consist of a few instructions from each category and few assembler directives.

Learning Objectives :-

① To understand Data structure of Pass-I assembler.
② To understand Pass-I assembler concept.
③ To understand Advanced Assembler Directives.

Outcomes :-

After completion of assignment, students WILL be able to -
① Implement Pass-I assembler.
② Implement Symbol Table, Literal Table and Pool Table.
③ Understand concept of Advanced Assembler Directive.

Theory Concepts :-

Introduction -

There are two main classes of programming languages: high level (e.g. C, pascal) and low level. Assembly language is a low level programming language. Programmers code symbolic instructions, each of which generates machine instructions.

An assembler is a program that accepts as input an assembly language program (source) and produces its machine language equivalent (object code) along with the information for the loader.

Advantages of coding in assembly language are :

① Provides more control over handling particular hardware components.
② May generate smaller, more compact executable modules.
③ Often results in faster execution.

Disadvantages :-

① Not portable
② More complex
③ Requires understanding of hardware details (interfaces).

Date: / /20 | Page No.

Pass-1 Assembler :-

An assembler does the following :

① Generate machine instructions
⇒ evaluate the mnemonics to produce their machine code.
⇒ evaluate the symbols, literals, addresses to produce their equivalent machine addresses.
⇒ convert the data constants into their machine representations.

② Process pseudo operations

Pass-2 Assembler :-

A two-pass assembler performs two sequential scans over the source code :

Pass 1 :- symbols and literals are defined.

Pass 2 :- object program is generated.

Parsing : moving in program lines to pull out op-codes and operands.

Data structures :-

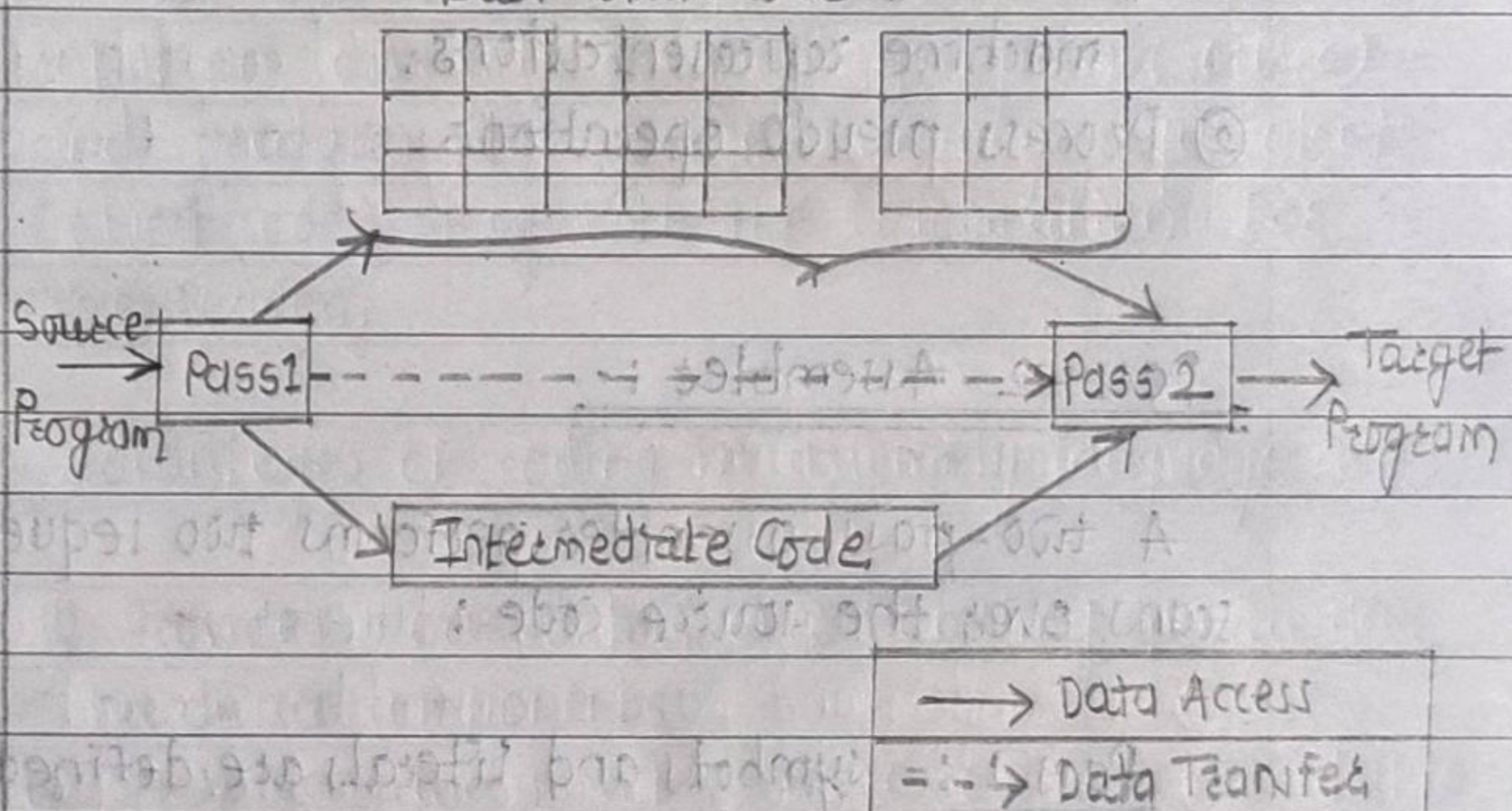
1.) Location counter (LC) : points to the next location where the code will be placed.

2.) Op-code translation table : contains symbolic instructions, their lengths and their op-codes (or subroutine to use for translation)

3.) Symbol table (ST) : contains lables and their values.

4.) string storage buffer (SSB) : contains ASCII characters for the strings.

5.) Forward references table (FRT) : contains pointer to the string in SSB and offset where its value will be inserted in the object code.



A simple two pass Assembler

Elements of Assembly Language :-

An assembly language programming provides three basic features which simplify programming 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 diagnostics. 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 statements. the assembler performs memory binding to these names; the programmer need not know any details of the memory bindings performed by the assembler.

3.) Data Declarations :-

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

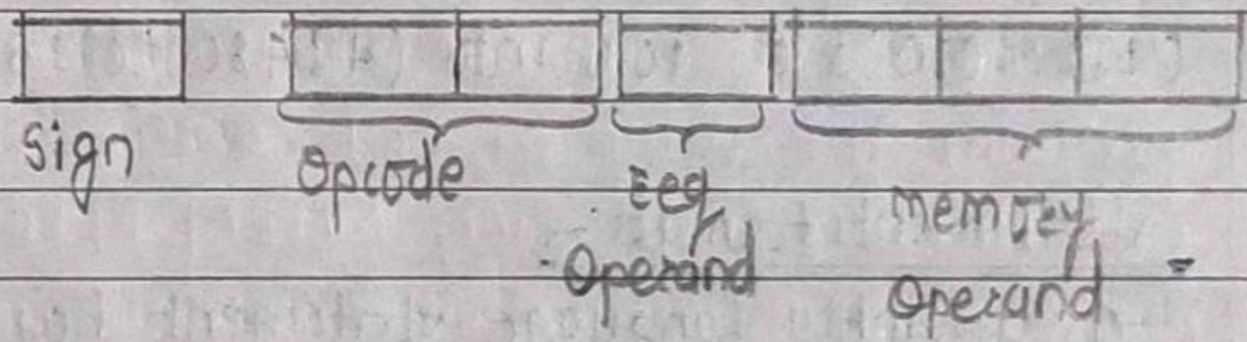
where the notation [...] indicates that the enclosed specification is optional.
Label associated as a symbolic name with the memory word(s) generated for the statement.

Mnemonic Operation Codes :

Instruction Opcode	Assembly mnemonic	Remarks
00	STOP	Stop execution.
01	ADD	First operand is modified Condition code is set

Instruction Opcode	Assembly mnemonic	Remarks
02	SUB	First operand is modified and 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	Analogous to SUB.
09	READ	First operand is not used
10	PRINT	

Instruction Format :-



Assembly Language Statements :-

Three kinds of statements -

1. Imperative statements
2. Declaration statements
3. Assembler Directives

a.) Imperative statements -

It indicates an action to be performed during the execution of the assembled program. Each imperative statement typically translates into one machine instruction.

b.) Declaration statements -

Two types of declaration statements are as follows -

[label] DS <constant>
[label] DC '<value>'

the DS (Declare Storage) statement reserves area of memory and associates names with them -

Eg.) A DS L
B DS 150

Eg.) ONE DC '1'

Associates the name ONE with a memory word containing the value '1'. The programmer can declare constants in decimal, binary, hexadecimal forms etc. These values are not protected by the assembler. In the above assembly language program the value of ONE can be changed by executing an instruction MOVEM BREG, ONE.

c.) Assembler Directives :-

Assembler directives instruct the assembler to perform certain actions during the assembly of a program. Some assembler directives are described in the following,

START <Constant>

Indicates that the first word of the target program generated by the assembler should be placed in the memory word with address <Constant>

END [<Operand spec>]

It indicates the end of the source program -

Pass structure of Assembler :-

Two pass Assembler -
can handle forward reference problem easily.

First phase : (Analysis)
Symbols are entered in the table called symbol table. Mnemonics and the corresponding opcodes are stored in a table called Mnemonic table.

LC processing

Second phase : (Synthesis)
Synthesis the target form using the address information found in symbol table.
First pass constructs an Intermediate Representation (IR) of the source program for use by the second pass.

Data Structure used during Synthesis phase :-

1.) Symbol Table 2.) Mnemonics table

Symbol address

AGAIN	104
N	113

Mnemonic opcode length

ADD	01	1
SUB	02	1

ADVANCED ASSEMBLER DIRECTIVES :-

1.) ORIGIN :
2.) EQU :
3.) LTORG :

ORIGIN :

Syntax: ORIGIN <address spec>

<address spec> can be an <operand spec> or constant

Indicates that location counter should be set to the address given by <address spec>.

This statement is useful when the target program does not consist of consecutive memory words.

E.g) ORIGIN loop+2

EQU :

Syntax: <symbol> EQU <address spec>

<address spec> operand spec (or) constant

Simply associates the name symbol with address specification. No location counter processing is implied.

E.g.) Back EQU loop

LTORG : (Literal Origin)

Where should the assembler place literals?

It should be placed such that the control never reaches it during the execution of a program. By default, the assembler places the literals after the END statement.

LTORG statement permits a programmer to specify where literals should be placed.

Example :-

Solve pass 1 of 2 pass assembler. Generate intermediate code and show the contents of symbol table, literal table, pool table.

Intermediate Code

START 100	(AD, 01) (C, 100)
L1 MOVER AREG, ='5'	100 (IS, 04) (01) (L, 0)
MOVEM BREG, X	101 (IS, 05) (02) (S, 1)
SUB AREG, ='2'	102 (IS, 02) (01) (L, 1)
LTORG	103 (DL, 02) (C, 5)
MOVER AREG, Y	104 (DL, 02) (C, 2)
BC any, L1	105 (IS, 04) (01) (S, 2)
ADD CREG, 4	106 (IS, 07) (CC, 07) (S, 0)
X DC 5	107 (IS, 01) (03) (L, 2)
Y DS 2	108 (DL, 02) (C, 5)
END	109 (DL, 01) (C, 2)
	111 (DL, 02) (C, 4)

Symbol Table –

L1	100	0
X	108	1
Y	109	2

Pool Table –

0
2

Literal Table –

5	103
2	104
4	111

Algorithm :-

Assembler pass 1 :

```
begin.
read first input line
    if OPCODE = 'START' then
        begin
        save #[OPERAND] as starting address
        initialize LOCCTR to starting address
        write line to intermediate file.
        read next input line
        end {if START}
    else
        initialize LOCCTR to 0
    while OPCODE != 'END' do
        begin
        if this is not a comment line then
            begin
                search SYMTAB for LABEL
                if found then
                    set error flag (duplicate symbol)
                else
                    insert (LABEL, LOCCTR) into SYMTAB
            end {if symbol}
        search OPTAB for OPCODE
        if found then
            add 3 to LOCCTR {instruction length} to LOCCTR
        else if OPCODE = 'WORD' then
            add 3 to LOCCTR
        else if OPCODE = 'RESW' then
            add 3* #[OPERAND] to LOCCTR
        else if OPCODE = 'BYTE' then
```

: Date : 1/20 | Page No :


```
begin
    find length of constant in bytes
    add length to LOCCTR
end {if BYTE}
else
    set error flag (invalid operation code)
end {if not a comment}
write line to intermediate file
read next input line
end {while not END}
write last line to intermediate file
save (LOCCTR - starting address) as program length
end {pass 1}
```

TEST CASES :-

Test case Id	steps to be exeruted	Expected Output	Actual Output	Test case Result
1.	'Input.txt' :			
	START 200	(AD,01)(C,200)	(AD,01)(C,200)	PASS
	MOVER AREG, ='4'	(IS,04)(1)(L,1)	(IS,04)(1)(L,1)	
	MOVEM AREG, A	(IS,05)(1)(S,1)	(IS,05)(1)(S,1)	
	MOVER BREG, ='1'	(IS,04)(2)(L,2)	(IS,04)(2)(L,2)	
	LOOP MOVER CREG, B	(IS,04)(3)(S,3)	(IS,04)(3)(S,3)	
	LTORG	(AD,05)	(AD,05)	
	ADD CREG, ='6'	(IS,01)(3)(L,3)	(IS,01)(3)(L,3)	
	STOP	(IS,00)	(IS,00)	
	A DS 1	(DL,02)(C,1)	(DL,02)(C,1)	
	B DS 1	(DL,02)(C,1)	(DL,02)(C,1)	
	END	(AD,02)	(AD,02)	

Test Case Id	Steps to be executed	Expected Result	Actual Result	Test case Result
		Symbol Table:	Symbol Table:	
		A 208	A 208	
		LOOP 203	LOOP 203	
		B 209	B 209	
		Literal Table:	Literal Table:	
		='4' 204	='4' 204	
		='6' 210	='6' 210	
		='1' 205	='1' 205	
		Pool Table:	Pool Table:	
		1	1	
		3	3	

Conclusion :-

We understood Data structure of pass I of assembler. Also, understood the advanced assembler directives. We have implemented pass I of assemble with symbol Table, Literal Table and Pool Table.

CODE :-

```
/*
 * Problem Statement :-
 *  Design suitable data structures and implement pass-I of a two-pass assembler for
 *  Pseudo-machine in Java.
 *  Implementation should consists of a few instructions from each category and few
 *  assembler directives.
 */


package assignmentNo_3;

import java.io.BufferedReader;
import java.io.*;
import java.io.IOException;
import java.util.*;

public class Assignment_No_3
{
    @SuppressWarnings({ "unchecked", "rawtypes" })

    public static void main(String[] args) {

        BufferedReader br = null;
        FileReader fr = null;

        FileWriter fw = null;
        BufferedWriter bw = null;

        try {
            String inputfilename = "F:\\TE Sem I\\LP - I\\LP-
I_Assignments\\Assignment No. 3\\src\\assignmentNo_3\\Input.txt";
            fr = new FileReader(inputfilename);
            br = new BufferedReader(fr);

            String OUTPUTFILENAME = "F:\\TE Sem I\\LP - I\\LP-
I_Assignments\\Assignment No. 3\\src\\assignmentNo_3\\Intermediate_Code.txt";
            fw = new FileWriter(OUTPUTFILENAME);
            bw = new BufferedWriter(fw);

            Hashtable<String, String> is = new Hashtable<String, String>();
            is.put("STOP", "00");
            is.put("ADD", "01");
            is.put("SUB", "02");
            is.put("MULT", "03");
            is.put("MOVER", "04");
```

```
            is.put("MOVEM", "05");
            is.put("COMP", "06");
            is.put("BC", "07");
            is.put("DIV", "08");
            is.put("READ", "09");
            is.put("PRINT", "10");

            Hashtable<String, String> dl = new Hashtable<String, String>();
            dl.put("DC", "01");
            dl.put("DS", "02");

            Hashtable<String, String> ad = new Hashtable<String, String>();

            ad.put("START", "01");
            ad.put("END", "02");
            ad.put("ORIGIN", "03");
            ad.put("EQU", "04");
            ad.put("LTORG", "05");

            Hashtable<String, String> symtab = new Hashtable<String, String>();
            Hashtable<String, String> littab = new Hashtable<String, String>();
            ArrayList<Integer> pooltab=new ArrayList<Integer>();

            String sCurrentLine;
            int locptr = 0;
            int litptr = 1;
            int symptr = 1;
            int pooltabptr = 1;

            sCurrentLine = br.readLine();

            String s1 = sCurrentLine.split(" ")[1];
            if (s1.equals("START")) {
                bw.write("AD \t 01 \t");
                String s2 = sCurrentLine.split(" ")[2];
                bw.write("C \t" + s2 + "\n");
                locptr = Integer.parseInt(s2);
            }

            while ((sCurrentLine = br.readLine()) != null) {
                int mind_the_LC = 0;
                String type = null;

                int flag2 = 0;      //checks whether address is assigned to current sy
mbol
```

```
                String s = sCurrentLine.split(" |\\,")[0];  //consider the first word
in the line

                for (Map.Entry m : symtab.entrySet()) //allocating address to arrived
symbols
                {
                    if (s.equals(m.getKey()))
                    {
                        m.setValue(locptr);
                        flag2 = 1;
                    }
                }
                if (s.length() != 0 && flag2 == 0) //if current string is not " " or a
ddress is not assigned,
                {                                   //then the current string must be
a new symbol

                    symtab.put(s, String.valueOf(locptr));
                    symptr++;
                }

                @SuppressWarnings("unused")
                int isOpcode = 0;       //checks whether current word is an opcode or
not

                s = sCurrentLine.split(" |\\,")[1]; //consider the second word in the
line

                for (Map.Entry m : is.entrySet())
                {
                    if (s.equals(m.getKey()))
                    {
                        bw.write("IS\t" + m.getValue() + "\t"); //if match found in im
perative statement
                        type = "is";
                        isOpcode = 1;
                    }
                }


                for (Map.Entry m : ad.entrySet())
                {
                    if (s.equals(m.getKey()))
                    {
                        bw.write("AD\t" + m.getValue() + "\t"); //if match found in As
sembler Directive
```

```
                    type = "ad";
                    isOpcode = 1;
                }
            }

            for (Map.Entry m : dl.entrySet())
            {
                if (s.equals(m.getKey()))
                {
                    bw.write("DL\t" + m.getValue() + "\t"); //if match found in de
clarative statement
                    type = "dl";
                    isOpcode = 1;
                }
            }


            if (s.equals("LTORG"))
            {
                pooltab.add(pooltabptr);
                for (Map.Entry m : littab.entrySet())
                {
                    if (m.getValue() == "") //if address is not assigned to the li
teral
                    {
                        m.setValue(locptr);
                        locptr++;
                        pooltabptr++;
                        mind_the_LC = 1;
                        isOpcode = 1;
                    }
                }
            }


            if (s.equals("END"))
            {
                pooltab.add(pooltabptr);
                for (Map.Entry m : littab.entrySet())
                {
                    if (m.getValue() == "")
                    {
                        m.setValue(locptr);
                        locptr++;
                        mind_the_LC = 1;
                    }
```

```
                    }
                }


                if(s.equals("EQU"))
                {
                    symtab.put("equ", String.valueOf(locptr));
                }


                if (sCurrentLine.split(" |\\,").length > 2) //if there are 3 words
                {
                    s = sCurrentLine.split(" |\\,")[2];             //consider the 3rd
 word

                                                                    //this is our firs
t operand.
                                                                    //it must be eithe
r a Register/Declaration/Symbol
                    if (s.equals("AREG"))
                    {
                        bw.write("1\t");
                        isOpcode = 1;
                    }
                    else if (s.equals("BREG"))
                    {
                        bw.write("2\t");
                        isOpcode = 1;
                    }
                    else if (s.equals("CREG"))
                    {
                        bw.write("3\t");
                        isOpcode = 1;
                    }
                    else if (s.equals("DREG"))
                    {
                        bw.write("4\t");
                        isOpcode = 1;
                    }
                    else if (type == "dl")
                    {
                        bw.write("C\t" + s + "\t");
                    }
                    else {
                        symtab.put(s, "");              //forward referenced symbol
                    }
```

```
                }


                if (sCurrentLine.split(" |\\,").length > 3) //if there are 4 words
                {

                    s = sCurrentLine.split(" |\\,")[3];     //consider 4th word.
                                                            //this is our 2nd operand
                                                            //it is either a literal,
or a symbol
                    if (s.contains("="))
                    {
                        littab.put(s, "");
                        bw.write("L\t" + litptr + "\t");
                        isOpcode = 1;
                        litptr++;
                    }
                    else
                    {
                        symtab.put(s, "");          //Doubt : what if the current symb
ol is already present in SYMTAB?
                                                    //Overwrite?
                        bw.write("S\t" + symptr + "\t");
                        symptr++;

                    }
                }

                bw.write("\n");     //done with a line.

                if (mind_the_LC == 0)
                    locptr++;
            }

            String f1 = "F:\\TE Sem I\\LP - I\\LP-
I_Assignments\\Assignment No. 3\\src\\assignmentNo_3\\Symbol_Table.txt";
            FileWriter fw1 = new FileWriter(f1);
            BufferedWriter bw1 = new BufferedWriter(fw1);
            System.out.println("\n\n\t === Symbol Table ===");
            for (Map.Entry m : symtab.entrySet())
            {
                bw1.write(m.getKey() + "\t" + m.getValue()+"\n");
                System.out.println("\t\t" + m.getKey() + "  " + m.getValue());
            }

            System.out.println("\n\n\t === Literal Table ===");
```

```
            String f2 = "F:\\TE Sem I\\LP - I\\LP-
I_Assignments\\Assignment No. 3\\src\\assignmentNo_3\\Literal_Table.txt";
            FileWriter fw2 = new FileWriter(f2);
            BufferedWriter bw2 = new BufferedWriter(fw2);
            for (Map.Entry m : littab.entrySet())
            {
                bw2.write(m.getKey() + "\t" + m.getValue()+"\n");
                System.out.println("\t\t" + m.getKey() + " " + m.getValue());
            }

            System.out.println("\n\n\t === Pool Table ===");
            String f3 = "F:\\TE Sem I\\LP - I\\LP-
I_Assignments\\Assignment No. 3\\src\\assignmentNo_3\\Pool_Table.txt";
            FileWriter fw3 = new FileWriter(f3);
            BufferedWriter bw3 = new BufferedWriter(fw3);
            for (Integer item : pooltab)
            {
                bw3.write(item+"\n");
                System.out.println("\t\t" + item);
            }

            bw.close();
            bw1.close();
            bw2.close();
            bw3.close();

        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

INPUT :-

Input.txt

```
 START 200
 MOVER AREG,='4'
 MOVEM AREG,A
 MOVER BREG,='1'
LOOP MOVER CREG,B
 LTORG
 ADD CREG,='6'
 STOP
A DS 1
B DS 1
 END
```

OUTPUT :-

Intermediate_Code.txt

```
AD     01   C     200
IS    04    1     L     1
IS    05    1     S     1
IS    04    2     L     2
IS    04    3     S     3
AD    05
IS    01    3     L     3
IS    00
DL    02    C     1
DL    02    C     1
AD    02
```

Symbol_Table.txt

```
A     208
LOOP  203
B     209
```

Literal_Table.txt

```
='4'  204
='6'  210
='1'  205
```

Pool_Table.txt

```
1
3
```