## PROGRAM

```
package Assembler.AssemblerPass1;
import java.io.*; //Import classes regarding file operations
import java.util.ArrayList;
//Define Data Structure for Symbol Table & Literal Table.
class Tables{
    String name;
    int address;
    Tables(String name, int address){
        this.name = name;
        this.address = address;
    }
}
//[OPTIONAL] Define Data Structure for Pooltable
class Pooltable{
    int first, total_literals;
    Pooltable(int f, int tl){
        first = f;
        total_literals = tl;
    }
}
public class Assembler_Pass1 {
    public static int search(String token, String[] list) {
        for(int i=0;i<list.length;i++)
            if(token.equalsIgnoreCase(list[i]))
                return i;
        return -1;
    }
    public static int search(String token, ArrayList<String> list) {
        for(int i=0;i<list.size();i++)
            if(token.equalsIgnoreCase(list.get(i)))
                return i;
        return -1;
    }
    public static int search(String token, Tables[] list, int cnt) {
        for(int i=0;i<cnt;i++)
            if(token.equalsIgnoreCase(list[i].name))
                return i;
        return -1;
    }
    public static void main(String[] args) throws IOException{
        //STEP 1 - Manually Define Arrays for Imperative, Declarative Statements and also
for Register symbols.
        String[] regs = {"AX","BX","CX","DX"};
        String[] impr =
{"STOP","ADD","SUB","MULT","MOVER","MOVEM","COMP","BC","DIV","READ","PRINT"};
        String[] decl = {"DS","DC"};
        //STEP 2 - Create Data Structures using above defined classes
        Tables[] op_table = new Tables[50];
        Tables[] symbol_table = new Tables[20];
        Tables[] literal_table = new Tables[20];
        Pooltable[] poolTab = new Pooltable[10];
        ArrayList<String > already_processed = new ArrayList<>();
        String line;
        try{
            BufferedReader br = new BufferedReader(new
FileReader("src\\Assembler\\AssemblerPass1\\sample.txt"));
            BufferedWriter bw = new BufferedWriter(new
FileWriter("src\\Assembler\\AssemblerPass1\\OutputTextTry.txt"));
            Boolean start = false, end = false, ltorg = false, fill_addr =
false,flag=false;
            int total_symb=0,total_ltr=0,optab_cnt=0,pooltab_cnt=0,loc=0,temp,pos,d;
```

```
            //Start reading ALP source code
            while((line=br.readLine())!=null && !end) {
                line=line.replaceAll(",", " ");
                System.out.println(line);
                //Spilt words in each line of source program.
                String[] words = line.split(" ");
                ltorg = fill_addr = false;
                //STEP 3 - Location Counter Processing.
                if (loc != 0 && !ltorg) {
                    //STEP 3.1 - As no locations are processed for Assembler
Directives,we just print that LC block as blank.
                    if(line.contains("START") || line.contains("END")
||line.contains("ORIGIN") ||line.contains("EQU") || line.contains("LTORG")){
                        bw.write("\n    ");
                    }
                    //STEP 3.2 - For Declarative statement processing of LC depends upon
memory word allocated so we process it while Symbol Processing
                    else if (line.contains("DS") || line.contains("DC")) {
                        flag = true;
                        bw.write("\n" + String.valueOf(loc));
                    }
                    //STEP 3.3 - For Imperative Statements simply increment LC by 1;
                    else bw.write("\n" + String.valueOf(loc++));
                }
                //Now we will process extracted word from line
                for (int i = 0; i < words.length; i++) {
                    pos = -1;
                    if (start == true) {
                        loc = Integer.parseInt(words[i]);
                        start = false;
                    }
                    //STEP 4 - Assembler Directives Processing.
                    switch (words[i]) {
                        //STEP 4.1 - For Start just print Intermediate Code and update
the start flag.
                        case "START":
                            start = true;
                            pos = 1;
                            bw.write("\t(AD," + pos + ")");
                            break;
                        //STEP 4.2 - For End just print Intermediate Code and update the
end flag.
                        case "END":
                            end = true;
                            pos = 2;
                            bw.write("\t(AD," + pos + ")");
                            //We need to process all literals
                            for (temp = 0; temp < total_ltr; temp++)
                                if (literal_table[temp].address == 0) {
                                    literal_table[temp].address = loc; //Assign the
ongoing line address to literals
                                    bw.write("\n\t(DL,2)\t(C," +
literal_table[temp].name.charAt(2) + ")"); //Print the Intermediate Code for literals
                                    loc++;
                                }
                            //PoolTable Processing [OPTIONAL]
                            if (pooltab_cnt == 0)
                                poolTab[pooltab_cnt++] = new Pooltable(0, temp);
                            else {
                                poolTab[pooltab_cnt] = new Pooltable(poolTab[pooltab_cnt
- 1].first + poolTab[pooltab_cnt - 1].total_literals, total_ltr - poolTab[pooltab_cnt -
1].first - 1);
                                pooltab_cnt++;
```

```
                            }
                            break;
                        //STEP 4.3 - When ORIGIN is encountered, LC is sent to provided
symbol's address in operand field.
                        case "ORIGIN":
                            pos = 3;
                            bw.write("\t(AD," + pos + ")");
                            //Search for given symbol in the operand field.
                            pos = search(words[++i], symbol_table, total_symb);
                            bw.write("\t(S," + (pos + 1) + ")");
                            //Update LC to given symbol's Address.
                            loc = symbol_table[pos].address;
                            break;
                        //STEP 4.4 - When EQU is encountered, LC is set to address of
symbol given in the operand field.
                        case "EQU":
                            pos = 4;
                            bw.write("\t(AD," + pos + ")");
                            String prev_word = words[i-1];  //Store new symbol (this
symbol is in the label field)
                            int pos1 = search(prev_word, symbol_table, total_symb);
                            //Get address of symbol provided in the operand field
                            pos = search(words[++i], symbol_table, total_symb);
                            //Set address of new symbol as same as address of Operand
Symbol
                            symbol_table[pos1].address = symbol_table[pos].address;
                            bw.write("\t(S," + (pos + 1) + ")");
                            break;
                        //STEP 4.5 - (IMP) Literals Processing.
                        case "LTORG":
                            ltorg = true;
                            pos = 5;
                            //We need to process all literals occured before LTORG
statement, so we use total_ltr to maintain count of literals
                            for (temp = 0; temp < total_ltr; temp++)
                                if (literal_table[temp].address == 0) {
                                    literal_table[temp].address = loc; //Assign the
ongoing line address to literals
                                    bw.write("\t(DL,2)\t(C," +
literal_table[temp].name.charAt(2) + ")\n"); //Print the Intermediate Code for literals
                                    loc++;
                                }
                            //PoolTable Processing [OPTIONAL]
                            if (pooltab_cnt == 0)
                                poolTab[pooltab_cnt++] = new Pooltable(0, temp);
                            else {
                                poolTab[pooltab_cnt] = new Pooltable(poolTab[pooltab_cnt
- 1].first + poolTab[pooltab_cnt - 1].total_literals, total_ltr - poolTab[pooltab_cnt -
1].first - 1);
                                pooltab_cnt++;
                            }
                            break;
                    }
                    //STEP 5 - Processing Imperative Statements.
                    if (pos == -1) {
                        //STEP 5.1 - Checke whether given word is an mnemonic by checking
OP Table.
                        pos = search(words[i], impr);
                        int r = search(words[i],regs);
                        //STEP 5.2 - If given word found in OP Table then it is an
Imperative Statement
                        if (pos != -1) {
                            bw.write("\t(IS," + pos + ")");//Print Intermediate Code for
```

```
Imperative Statement.
                        op_table[optab_cnt++] = new Tables(words[i], pos); //Upate
its entry in MOT
                    }
                    //STEP 6 - Declarative Statement Processing.
                    else {
                        //Check whether word is DS or DC
                        pos = search(words[i], decl);
                        //if word is DS or DC
                        if (pos != -1) {
                            bw.write("\t(DL," + (pos + 1) + ")");
                            op_table[optab_cnt++] = new Tables(words[i], pos+1);
                            fill_addr = true;
                        }
                        //STEP 7 - (IMP) SYMBOL PROCESSING.
                        //STEP 7.1 - Check label field of source code by  checking
i==0 or not and has any symbol.
                        else if (i==0 && words[i].matches("[a-zA-Z]+") && r==-1) {
                            //Check whether symbol is already present in Symbol Table
                            pos = search(words[i], symbol_table, total_symb);
                            System.out.println("TS" + total_symb);
                            //If new symbol encountered process it.
                            if (pos == -1) {
                                //Assign a temp variable for helping in assigning LC
to symbol without disturbing orginal LC.
                                if(flag==false){
                                    d = --loc;
                                    ++loc;
                                    flag = false;
                                }
                                else d = loc;
                                //STEP 7.2 - Update Entry in Symbol Table.
                                symbol_table[total_symb++] = new Tables(words[i],
(d));
                                //STEP 7.3 - Update LC as per the rules.
                                if(words[i+1].matches("DS")){
                                    loc += Integer.parseInt(words[2]); //For DS
increment LC by given operand value
                                }
                                else if(line.contains("DC")) loc++; //For DC
increment LC simply by 1.
                                pos = search(words[i], already_processed);
                                if(pos==-1){
                                    already_processed.add(words[i]);
                                    System.out.println(already_processed);
                                    pos = search(words[i], already_processed);
                                }
//                                  bw.write("\t(S," + total_symb + ")"); //Write its
intermediate code
//                                  pos = total_symb;
                            }
                        }
                        else if(words[i].matches("[a-zA-Z]+") && r==-1){
                            System.out.println("Words : " + words[i]);
                            pos = search(words[i], already_processed);
                            if(pos==-1){
                                already_processed.add(words[i]);
                                System.out.println(already_processed);
                                pos = search(words[i], already_processed);
                            }

                            bw.write("\t(S," + (pos+1) + ")"); //Write its
intermediate code
```

```
                                pos = total_symb;
                            }
                        }
                    }
                    //STEP 8 - Registers, Constants and Literal's IC Printing
                    if (pos == -1) {
                        pos = search(words[i], regs);
                        if(pos!=-1)
                            bw.write("\t("+(pos+1)+")");
                        else{
                            if(words[i].matches("='\\d+'")){
                                literal_table[total_ltr++] = new Tables(words[i],0);
                                bw.write("\t(L,"+total_ltr+")");
                            }
                            else if(words[i].matches("\\d+") || words[i].matches("\\d+H")
|| words[i].matches("\\d+h"))
                                bw.write("\t(C,"+words[i]+")");
                        }
                    }
                }
            }
            br.close();
            bw.close();
            BufferedWriter sw = new BufferedWriter(new
FileWriter("src\\Assembler\\AssemblerPass1\\symTab.txt"));
            sw.write("\nSYMBOL\tADDRESS\n");
            for(int i=0;i<total_symb;i++)
                sw.write(symbol_table[i].name+"\t\t"+symbol_table[i].address+"\n");
            sw.close();

            BufferedWriter lw = new BufferedWriter(new
FileWriter("src\\Assembler\\AssemblerPass1\\litTab.txt"));
            lw.write("\nIndex\t\tLITERAL\t\tADDRESS\n");
            for(int i=0;i<total_ltr;i++)
            {
                if(literal_table[i].address==0)
                    literal_table[i].address=loc++;
                lw.write((i+1)
+"\t\t\t"+literal_table[i].name+"\t\t\t"+literal_table[i].address+"\n");
            }
            lw.close();

            BufferedWriter pw = new BufferedWriter(new
FileWriter("src\\Assembler\\AssemblerPass1\\poolTab.txt"));
            pw.write("\nPOOL\tTOTAL LITERALS\n");
            for(int i=0;i<pooltab_cnt;i++)
                pw.write(poolTab[i].first+"\t\t\t"+poolTab[i].total_literals+"\n");
            pw.close();

            BufferedWriter mw = new BufferedWriter(new
FileWriter("src\\Assembler\\AssemblerPass1\\opTab.txt"));
            mw.write("\nMNEMONIC\tOPCODE\n");
            for(int i=0;i<optab_cnt;i++)
                mw.write(op_table[i].name+"\t\t"+op_table[i].address+"\n");
            mw.close();

        }catch(Exception e){
            System.out.println("Error occured while reading file\n");
            e.printStackTrace();
        }
    }
}
```

**INPUT**

sample.txt

```
1   START 100
2   MOVER AX,05
3   MOVER BX,10
4   UP ADD AX,BX
5   MOVEM A,='5'
6   MULT AX,A
7   ORIGIN UP
8   LTORG
9   MOVEM B,='9'
10  MOVEM C,='8'
11  LTORG
12  MOVEM B,='7'
13  MOVEM C,='8'
14  A DS 02
15  B DC 10
16  C DS 09
17  NEXT EQU UP
18  END
```

**Input – Source Program**

# OUTPUT

symTab.txt

```
1
2   SYMBOL   ADDRESS
3   UP       102
4   A        109
5   B        111
6   C        112
7   NEXT         102
```

Output 1 : - Symbol Table

litTab.txt

```
1
2   Index        LITERAL      ADDRESS
3   1            ='5'             102
4   2            ='9'             105
5   3            ='8'             106
6   4            ='7'             121
7   5            ='8'             122
```

Output 2 : - Literal Table

poolTab.txt

```
1
2   POOL     TOTAL LITERALS
3   0            1
4   1            2
5   3            3
```

Output 3 : - Pool Table

opTab.txt

```
1
2   MNEMONIC    OPCODE
3   MOVER       4
4   MOVER       4
5   ADD     1
6   MOVEM       5
7   MULT        3
8   MOVEM       5
9   MOVEM       5
10  MOVEM       5
11  MOVEM       5
12  DS      1
13  DC      2
14  DS      1
```

Output 4 : - MOT

OutputTextTry.txt

```
1       (AD,1)  (C,100)
2   100 (IS,4)  (1) (C,05)
3   101 (IS,4)  (2) (C,10)
4   102 (IS,1)  (1) (2)
5   103 (IS,5)  (S,2)   (L,1)
6   104 (IS,3)  (1) (S,2)
7       (AD,3)  (S,1)
8       (DL,2)  (C,5)
9
10  103 (IS,5)  (S,3)   (L,2)
11  104 (IS,5)  (S,4)   (L,3)
12      (DL,2)  (C,9)
13      (DL,2)  (C,8)
14
15  107 (IS,5)  (S,3)   (L,4)
16  108 (IS,5)  (S,4)   (L,5)
17  109 (DL,1)  (C,02)
18  111 (DL,2)  (C,10)
19  112 (DL,1)  (C,09)
20      (AD,4)  (S,1)
21      (AD,2)
22      (DL,2)  (C,7)
23      (DL,2)  (C,8)
```

Output 5 : - Intermediate Code