```
package ass1assemblerpass2;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.StringTokenizer;

//Desgining MOT
class Tuple {
 String mnemonic, bin_opcode, type;
 int length;

 Tuple() {}

 Tuple(String s1, String s2, String s3, String s4) {
  mnemonic = s1;
  bin_opcode = s2;
  length = Integer.parseInt(s3);
  type = s4;
 }
}
//Desgining ST
class SymTuple {
 String symbol, ra;
 int value, length;

 SymTuple(String s1, int i1, int i2, String s2) {
  symbol = s1;
  value = i1;
  length = i2;
  ra = s2;
 }
}
//Designing Literal
class LitTuple {
 String literal, ra;
 int value, length;

 LitTuple() {}

 LitTuple(String s1, int i1, int i2, String s2) {
  literal = s1;
  value = i1;
  length = i2;
  ra = s2;
 }
}
```

```
public class ass1assemblerpass2 {

    static int lc;
    static List<Tuple> mot; //required to read MOT
    static List<String> pot; //required to read POT
    static List<SymTuple> symtable; //generate symbol table
    static List<LitTuple> littable; //generate literal table
    static List<Integer> lclist;
    static Map<Integer, Integer> basetable; //base table
    static PrintWriter out_pass2; //output of pass 2
    static PrintWriter out_pass1; //output of pass 1
    static int line_no;

    public static void main(String[] args) throws Exception {
        initializeTables(); //initialize everything needed

        //initialize evrything as per output of pass 1
        //initialize symtable from out_symtable.txt
        String s;
        BufferedReader br;
        br = new BufferedReader(new InputStreamReader(new FileInputStream("/home/student/workspace/
SPOSL/src/out_symtable.txt")));//reading Symbol table
        while ((s = br.readLine()) != null) {
            StringTokenizer st = new StringTokenizer(s, " ", false); //convert line into tokens
            symtable.add(new SymTuple(st.nextToken(), Integer.parseInt(st.nextToken()), Integer.parseInt(st.
nextToken()), st.nextToken())); //adding token into list
        }

        //initialize littable from out_littable.txt

        br = new BufferedReader(new InputStreamReader(new FileInputStream("/home/student/workspace/
SPOSL/src/out_littable.txt")));//reading literal table
        while ((s = br.readLine()) != null) {
            StringTokenizer st = new StringTokenizer(s, " ", false); //convert line into tokens
            littable.add(new LitTuple(st.nextToken(), Integer.parseInt(st.nextToken()), Integer.parseInt(st.next
Token()), st.nextToken())); //adding token into list
        }

        //initialize lclist from lclist.txt
        br = new BufferedReader(new InputStreamReader(new FileInputStream("/home/student/workspace/
SPOSL/src/lclist.txt")));//reading lclist
        while ((s = br.readLine()) != null) {
            StringTokenizer st = new StringTokenizer(s, "\n", false); //convert line into tokens
            lclist.add(Integer.parseInt(st.nextToken())); //adding token into list
        }

        System.out.println("\n====== PASS 2 ======\n");
 pass2(); //Run Pass 2

    }
    static void initializeTables() throws Exception {
        symtable = new LinkedList<>();
        littable = new LinkedList<>();
        lclist = new ArrayList<>();
        basetable = new HashMap<>();
```

```
        mot = new LinkedList<>();
        pot = new LinkedList<>();
        String s;
        BufferedReader br;
        br = new BufferedReader(new InputStreamReader(new FileInputStream("/home/student/workspace/
SPOSL/src/mot.txt")));//reading MOT
        while ((s = br.readLine()) != null) {
            StringTokenizer st = new StringTokenizer(s, " ", false); //convert line into tokens
            mot.add(new Tuple(st.nextToken(), st.nextToken(), st.nextToken(), st.nextToken())); //adding toke
n into list
        }
        br = new BufferedReader(new InputStreamReader(new FileInputStream("/home/student/workspace/
SPOSL/src/pot.txt")));//reading POT
        while ((s = br.readLine()) != null) {
            pot.add(s); //adding token into POT list
        }
        Collections.sort(pot); //sorting all the POT as per their index
    }
    static void pass2() throws Exception {
  line_no = 0; //give line number as 0 for checking output pass 1 file
  out_pass2 = new PrintWriter(new FileWriter("/home/student/workspace/SPOSL/src/output_pass2.txt"), tr
ue);
  BufferedReader input = new BufferedReader(new InputStreamReader(new FileInputStream("/home/stud
ent/workspace/SPOSL/src/output_pass1.txt"))); //read output pass 1
  String s; //to be used
  System.out.println("Pass 2 input:");
  while((s = input.readLine()) != null) { //read the complete pass 2 input from pass1 output file
   System.out.println(s);
   StringTokenizer st = new StringTokenizer(s, " ", false); //dividing line into tokens
   String s_arr[] = new String[st.countTokens()]; //initialize the s_arr
   for(int i=0 ; i < s_arr.length ; i++) {
    s_arr[i] = st.nextToken(); //read all tokens
   }
   if(searchPot2(s_arr) == false) { //check if entry in POT
    searchMot2(s_arr); //if not, check in MOT
   }
   line_no++; //update line no
  }
  System.out.println("\nPass 2 output:");
  input = new BufferedReader(new InputStreamReader(new FileInputStream("/home/student/workspace/S
POSL/src/output_pass2.txt")));
  while((s = input.readLine()) != null) {
   System.out.println(s);
  }
 }
 static boolean searchPot2(String[] s) {
  int i = 0; //to be used

  if(s.length == 3) { //check if 3 tokens in a line
   i = 1;
  }
  if(Collections.binarySearch(pot, s[i]) >= 0) { //check all symbols and pseudo-ops in a file by using binary
search
   if(s[i].equalsIgnoreCase("USING")) { //if USING occurs
    s = tokenizeOperands(s); //tokenize operands
```

```
   if(s[i+1].equals("*")) { //if there is a pointer after USING
    s[i+1] = lclist.get(line_no) + ""; //get next value as location counter which is line_no
   } else {
    for(int j=i+1 ; j<s.length ; j++) {
     int value = getSymbolValue(s[j]); //get symbol value in value
     if(value != -1) {
      s[j] = value + ""; //get symbol value
     }
    }
   }
   basetable.put(new Integer(s[i+2].trim()), new Integer(s[i+1].trim())); //store base register and offset
  }
  return true; //got POT
 }
 return false; //go for MOT
}
static int getSymbolValue(String s) { //get the symbol value from symbol table
 for(SymTuple st : symtable) {
  if(s.equalsIgnoreCase(st.symbol)) {
   return st.value;
  }
 }
 return -1;
}
static void searchMot2(String[] s) {
 Tuple t = new Tuple(); //create new MOT object
 int i = 0;
 int j;

 if(s.length == 3) { //if three tokens in a line
  i = 1;
 }
 s = tokenizeOperands(s); //convert line into tokens

 for(Tuple x : mot) { //traverse through MOT entries
  if(s[i].equals(x.mnemonic)) { //get all mnemonics in t
   t = x;
   break;
  }
 }

 String output = new String();
 String mask = new String();
 if(s[i].equals("BNE")) { //mask BNE with 7
  mask = "7";
 } else if(s[i].equals("BR")) { //mask BR with 15
  mask = "15";
 } else {
  mask = "0";
 }
 if(s[i].startsWith("B")) { //check for BCR or BR instruction
  if(s[i].endsWith("R")) {
   s[i] = "BCR";
  } else {
```

```
  s[i] = "BC";
 }
 List<String> temp = new ArrayList<>();
 for(String x : s) {
  temp.add(x); //get all tokens into temp
 }
 temp.add(i+1, mask); //add masks to temp
 s = temp.toArray(new String[0]); //convert list into arrayList and store in x
}
if(t.type.equals("RR")) { //check for instruction type, if 'RR'
 output = s[i]; //write to output string
 for(j=s[i].length() ; j<6 ; j++) { //get symbol name in output
  output += " ";
 }
 for(j=i+1 ; j<s.length ; j++) { //get symbol value
  int value = getSymbolValue(s[j]);
  if(value != -1) {
   s[j] = value + "";
  }
 }
 output += s[i+1]; //append output
 for(j=i+2 ; j<s.length ; j++) {
  output += ", " + s[j]; //append the instruction length
 }
} else { //if RX instruction
 output = s[i]; //get s[i] in output
 for(j=s[i].length() ; j<6 ; j++) { //get name
  output += " ";
 }
 for(j=i+1 ; j<s.length-1 ; j++) { //get instruction value
  int value = getSymbolValue(s[j]);
  if(value != -1) {
   s[j] = value + "";
  }
 }
 s[j] = createOffset(s[j]); //create offset of RX type instructions
 output += s[i+1];
 for(j=i+2 ; j<s.length ; j++) {
  output += ", " + s[j]; //get length of instruction
 }
}
 out_pass2.println(output); //print output of pass 2
}
      static String[] tokenizeOperands(String[] s) {
 List<String> temp = new LinkedList<>(); //to be used
 for(int j=0 ; j<s.length-1 ; j++) { //adding all tokens into temp
  temp.add(s[j]);
 }
 StringTokenizer st = new StringTokenizer(s[s.length-1], " ,", false); //convert line into tokens
 while(st.hasMoreTokens()) {
  temp.add(st.nextToken()); //adding all tokens
 }
 s = temp.toArray(new String[0]); //convert linked list to array list
 return s;
}
```

```
        static String createOffset(String s) {
  String original = s; //get s in original
  Integer[] key = basetable.keySet().toArray(new Integer[0]); //get base register number in key
  int offset, new_offset; //to be used
  int index = 0; //to be used
  int value = -1; //to be used
  int index_reg = 0; //to be used
  if(s.startsWith("=")) { //check RX by checking '=' in an output pass 1 line
   value = getLiteralValue(s); //get literal value ahead of '='
  } else {
   int paranthesis = s.indexOf("("); //check '(' in line
   String index_string = new String(); //index_string
   if(paranthesis != -1) { //check index of paranthesis
    s = s.substring(0, s.indexOf("(")); //store substring in s
    index_string = original.substring(original.indexOf("(")+1, original.indexOf(")"));//get index_string '(offset)'

    index_reg = getSymbolValue(index_string); //get symbol value
   }
   value = getSymbolValue(s); //get symbol value here
  }
  offset = Math.abs(value - basetable.get(key[index])); //calculate offset by offset=value in ST - contents of
 Base Register
  for(int i=1 ; i<key.length ; i++) {
   new_offset = Math.abs(value - basetable.get(key[i])); //calculate offset by offset=value in ST - contents o
f Base Register
   if(new_offset < offset) { //check if new offset is in range
    offset = new_offset; //give new offset
    index = i; //update index position
   }
  }
  String result = offset + "(" + index_reg + ", " + key[index] + ")"; //represent index_register and base regist
er
  return result; //give in '(index_reg,Base_register)' format
 }
 static int getLiteralValue(String s) {
  s = s.substring(1, s.length());
  for(LitTuple lt : littable) { //traverse literal table and get literal value
   if(s.equalsIgnoreCase(lt.literal)) {
    return lt.value;
   }
  }
  return -1; //if not present then return -1
 }

}
```