```
/*

 CODE WRITER - translates VM Code (foo.vm) into assembly language (foo.asm). It can
translate more than one .vm file into one .asm file.
             - it uses R13 to momentarily memorize the address of the memory segment to
pop to.
                       R14 to momentarily memorize returnAddress duning return.
                       R15 to momentarily memorize LCL during return.

*/

package vmtranslator;

import java.io.*;
import java.util.*;

public class CodeWriter {

    // In order to read from the .vm file(s) and write to the .asm file.
    private BufferedReader r;
    private PrintWriter w;

    // The other component of the CodeWriter
    private final VMParser parse = new VMParser(); // Decomposes each VM command and
retrieves its various parts.

    // Loaded file(s) internal representation.
    private final File directory;                                    // The directory
containing the .vm file(s).
    private final Vector<String> fileList = new Vector<>(); // List of .vm files to be
transcoded
    private String currentFileName;                                  // Name of the single .vm
file currently being processed (no extension).
    private Vector<String> program;                        // Contains line by line the
currently to be translated .vm file.
    private String line;                                             // Current line being
processed.

    private int lineNumber = 0;       // Used to create unique labels for A-commands that
need them. Gets incremented each new line being processed. NOT resetted for each file.
    private int nestedCallNumber = 0; // Used to create unique labels for same-function
nested calls.


    public CodeWriter(File directory) {
        this.directory = directory;

        populateProgramList(directory); // Creates list of .vm files contained in
'directory'.
        initializeDirW(directory);      // Creates PrintWriter w for the
directoryName.asm file.
        writeBootstrap();               // Writes BOOTSTRAP code to the directoryName.asm
file.
        translateDirectory();           // Actually adds the translation of all .vm files
to directoryName.asm.
        exit();                         // Closes writer and reader.
    }


    private void populateProgramList(File path) {
        String [] allFiles =
path.list();                                                     // All files contained in
the directory.
        for (String file : allFiles) if (file.endsWith(".vm"))
fileList.addElement(file); // Only .vm files get added to fileList.
    }
```

```
    private void initializeDirW(File path) {
        String pathFile = path.getAbsolutePath() + File.separator + path.getName() +
".asm";
        try {
            w = new PrintWriter(new FileWriter(new File(pathFile)));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void writeBootstrap() {
        w.println("@256"); // SP=256
        w.println("D=A");
        w.println("@SP");
        w.println("M=D");

        w.println("@SP"); // Leave space for return address.
        w.println("M=M+1");

        callBody("0");

        w.println("@Sys.init");
        w.println("0;JEQ");

        w.println("(Sys.init$returnAddress)"); // Writes label name
    }

    private void translateDirectory() {
        for(int i = 0; i< fileList.size(); i++) { // Cycles all files.
            currentFileName = fileList.elementAt(i).replaceAll(".vm", "");             //
Retrieves the name (no extension) of the .vm file currently being translated.
            initializeDirR(directory.getAbsolutePath() + File.separator + currentFileName
+ ".vm"); // Creates BufferedReader r to read the .vm file currently being translated.

loadFile();                                                                             //
Loads .vm file in program. Also removes spaces and comments.

translateFile();                                                                        //
Actually does the translating.
        }
    }

    private void initializeDirR(String file) {
        try {
            r = new BufferedReader(new FileReader(new File(file)));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void loadFile() {
        program = new Vector<>(); // Re-initializes each time the program vector.
        try {
            while(true) {
                line = r.readLine();
                if (line == null) break;                // File ended.
                clean();                                // Removes comments from each line.
                if(!line.isEmpty()) program.add(line); // Adds line to program, unless it
is an empty line.
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
    private void clean() {
        line = removeComments(line); // Removes comments.
        line = line.trim();
    }

    private String removeComments(String line) {
        if(line.contains("//")) line = line.substring(0, line.indexOf("//")); // In case
there is a commment i removes it.
        return line;
    }

    private void translateFile() {
        for(int i=0; i<program.size(); i++) { // Cycles all lines in program and
translate them.
            line = program.elementAt(i);
            translateLine();
            lineNumber++; // Used in order to create unique labels in A-commands that
require labels.
        }
    }

    private void translateLine() {
        w.println("//" + line); // Useful to inspect the output to see what VM command
produced what translation.
        switch(parse.commandType(line)) {
            case "A": writeA(); break; // A is ARITHMETIC/LOGIC command
            case "B": writeB(); break; // B is MEMORY SEGMENT command
            case "C": writeC(); break; // C is BRANCHING command
            case "D": writeD(); break; // D is FUNCTION command
            default: System.out.println("ERROR"); break;
        }
    }

    private void translateLine(String line) {
        this.line = line;
        translateLine();
    }

    private void writeA() { // ARITHMETIC/LOGIC command.
        switch(line) { // 'line' content coincides with the command.
          case "add": writeAadd(); break;
          case "sub": writeAsub(); break;
          case "neg": writeAneg(); break;
          case "eq": writeAeq(); break;
          case "gt": writeAgt(); break;
          case "lt": writeAlt(); break;
          case "and": writeAand(); break;
          case "or": writeAor(); break;
          case "not": writeAnot(); break;
          default: break;
        }
    }

    private void writeAprimer() {
      w.println("@SP");   // Stack: |...|x|y|SP|
      w.println("M=M-1"); // Decrement SP   ->   |...|x|ySP|
      w.println("A=M");   // Select address of last element of the stack (y) (SP which
has just been decremented).
      w.println("D=M");   // Memorize its content in D (D=y).
      w.println("A=A-1"); // Select address of second-to-last element of the stack (x).
    }

    private void writeApost() {
      w.println("@SP");                      // In case it doesn't jump, it has to
w.println false.
      w.println("A=M-1");                    // Last element of stack.
```

```
      w.println("M=0");                          // Sets it to FALSE.
      w.println("@END_" + lineNumber);           // Points to the end.
      w.println("0;JMP");                        // Goes to the end.
      w.println("(LABEL_" + lineNumber + ")"); // In case it jumps.
      w.println("@SP");
      w.println("A=M-1");                        // Last element of stack.
      w.println("M=-1");                         // Sets it to TRUE.
      w.println("(END_" + lineNumber + ")");
    }

    private void writeAadd() {
      writeAprimer();
      w.println("M=D+M"); // Sum its content to D (which contains the other factor y) and
replaces it (x = x+y).
    }

    private void writeAsub() {
      writeAprimer();
      w.println("M=M-D"); // Subtracts D (which contains the other factor y) to x and
replaces it (x = x-y).
    }

    private void writeAneg() {
      w.println("@SP");   // *(SP-1)   ->   -(SP-1)
      w.println("A=M-1"); // Select address of last element of the stack (y) (SP
decremented).
      w.println("M=-M");  // Substitute its content with the opposite of it (y=-y).
    }

    private void writeAeq() {
      writeAprimer();
      w.println("D=M-D");                // Subtracts D (which contains the other factor
y) to x and puts it in D (D = x-y).
      w.println("@LABEL_" + lineNumber); // Points LABEL.
      w.println("D;JEQ");                // if x==y, then x-y==0 and the program jumps to
(LABEL).
      writeApost();
    }

    private void writeAgt() {
      writeAprimer();
      w.println("D=M-D");                // Subtracts D (which contains the other factor
y) to x and puts it in D (D = x-y).
      w.println("@LABEL_" + lineNumber); // Points LABEL.
      w.println("D;JGT");                // if x>y, then x-y>0 and the program jumps to
(LABEL).
      writeApost();
    }

    private void writeAlt() {
      writeAprimer();
      w.println("D=M-D");                // Subtracts D (which contains the other factor
y) to x and puts it in D (D = x-y).
      w.println("@LABEL_" + lineNumber); // Points LABEL.
      w.println("D;JLT");                // if x==y, then x-y==0 and the program jumps to
(LABEL).
      writeApost();
    }

    private void writeAand() {
      writeAprimer();
      w.println("M=D&M"); // ANDs D (which contains the other factor y) and x and puts it
in x (x = x&y).
    }

    private void writeAor() {
```

```
      writeAprimer();
      w.println("M=D|M"); // ORs D (which contains the other factor y) and x and puts it
in x (x = x|y).
    }

    private void writeAnot() {
      w.println("@SP");
      w.println("A=M-1"); // Select address of last element of the stack (y) (SP
decremented).
      w.println("M=!M");  // Negates the content and substitutes it.
    }

    private void writeB() { // MEMORY SEGMENT command.
      switch(parse.arg1(line)) { // Whether is a PUSH or POP.
        case "push": writeBpush(); break;
        case "pop": writeBpop(); break;
        default: break;
      }
    }

    private void writeBpush() {
      switch(parse.arg2(line)) { // Whether is local, argument, this, that, constant,
static, pointer, temp.
        case "local": w.println("@LCL"); break;
        case "argument": w.println("@ARG"); break;
        case "this": w.println("@THIS"); break;
        case "that": w.println("@THAT"); break;
        case "constant": writeBpushConstant(); return;
        case "static": writeBpushStatic(); return;
        case "pointer": writeBpushPointer(); return;
        case "temp": writeBpushTemp(); return;
        default: break;
      }
      writeBfinal();
    }

    private void writeBpushConstant() {
        w.println("@" + parse.arg3(line));
        w.println("D=A");
        simplerPush();
    }

    private void writeBpushStatic() {
        simplePush(currentFileName + "." + parse.arg3(line));
    }

    private void simplePush(String toPush) {
        w.println("@" + toPush);
        w.println("D=M");
        simplerPush();
    }

    private void simplePush() {
        w.println("D=M");
        simplerPush();
    }

    private void simplerPush() {
        w.println("@SP");
        w.println("M=M+1");
        w.println("A=M-1");
        w.println("M=D");
    }

    private void writeBpushPointer() {
      switch (parse.arg3(line)) {
```

```
            case "0": w.println("@3"); break;
            case "1": w.println("@4"); break;
            default: break;
          }
          simplePush();
        }

        private void writeBpushTemp() {
            w.println("@5");
            w.println("D=A");
            writeBend();
        }

        private void writeBfinal() {
            w.println("D=M");
            writeBend();
        }

        private void writeBend() {
            w.println("@" + parse.arg3(line));
            w.println("D=D+A"); // D = address + offset.
            w.println("A=D");   // Point to address + offset.
            simplePush();
        }

        private void writeBpop() {
            switch(parse.arg2(line)) { // Whether is local, argument, this, that, constant,
static, pointer, temp.
                case "local": w.println("@LCL");  break;
                case "argument": w.println("@ARG"); break;
                case "this": w.println("@THIS"); break;
                case "that": w.println("@THAT"); break;
                case "static": writeBpopStatic(); return;
                case "pointer": writeBpopPointer(); return;
                case "temp": writeBpopTemp(); return;
                default: break;
            }
            writeBpopFinal();
        }

        private void simplePop() { // Puts last stack value (value to be popped) into D and
decreases SP.
            w.println("@SP");
            w.println("M=M-1");
            w.println("A=M");
            w.println("D=M");
        }

        private void writeBpopStatic() {
            simplePop();
            w.println("@" + currentFileName + "." + parse.arg3(line)); // Creates new static
variable. For name.asm creates name.1, name.2, name.3... as static variables.
            w.println("A=M");
            w.println("M=D");
        }

        private void writeBpopPointer() {
            simplePop();
            switch (parse.arg3(line)) {
                case "0": w.println("@THIS"); break;
                case "1": w.println("@THAT"); break;
                default: break;
            }
            w.println("M=D");
        }
```

```
    private void writeBpopTemp() {
        w.println("@5");  // 5..12 are the memory cells reserved to TEMP.
        w.println("D=A");
        writeBpopEnd();
    }

    private void writeBpopFinal() {
        w.println("D=M");  // Gets value inside LCL/ARG/THIS/THAT.
        writeBpopEnd();
    }

    private void writeBpopEnd() { // Operates with the starting address of the memory
segment to pop to in D.
        w.println("@" + parse.arg3(line)); // D already contains the starting address of
the memory segment to pop to. Arg3 is the offset.
        w.println("D=D+A");               // Points the actual address to which the
value will be popped.
        w.println("@R13");                // Memorize address of memory to pop to in R13.
        w.println("M=D");
        simplePop();
        w.println("@R13");                // Puts the content od D (value to pop) into
the address in R13.
        w.println("A=M");
        w.println("M=D");
    }

    private void writeC() {
        switch(parse.arg1(line)) { // Whether is a LABEL, IF-GOTO, GOTO.
            case "label": writeClabel(); break;
            case "if-goto": writeCifgoto(); break;
            case "goto": writeCgoto(); break;
            default: break;
        }
    }

    private void writeClabel() {
        w.println("(" + parse.arg2(line) + ")"); // Writes label name.
    }

    private void writeCifgoto() {
        w.println("@SP");                 // Puts last element of the stack in D.
        w.println("M=M-1");
        w.println("A=M");
        w.println("D=M");
        w.println("@" + parse.arg2(line)); // Points to the goto address.
        w.println("D;JNE");               // Jumps if D!=0 (if last element of the stack
was TRUE).
    }

    private void writeCgoto() {
        w.println("@" + parse.arg2(line));
        w.println("0;JEQ");
    }

    private void writeD() {
        switch(parse.arg1(line)) { // Whether is a FUNCTION, CALL, RETURN.
            case "function": writeDfunction(); break;
            case "call": writeDcall(); break;
            case "return": writeDreturn(); break;
            default: break;
        }
    }

    private void writeDcall() {
        w.println("@" + parse.arg2(line) + "_call." + nestedCallNumber + "_" +
"$returnAddress"); // Pushes returnAddress of the CALLER.
```

```
        w.println("D=A");
        simplerPush();

        callBody(parse.arg3(line));

        w.println("@" + parse.arg2(line)); // Jumps to the CALLEE, which is arg2.
        w.println("0;JEQ");

        w.println("(" +  parse.arg2(line) + "_call." + nestedCallNumber + "_" +
"$returnAddress" + ")"); // Writes label name of the returnAddress of the current
specific call of the CALLER function.
        nestedCallNumber+-
+;                                                                              //
Increases the call number so that if the same function gets called in the future the
return addresses are not confused.
    }

    private void callBody(String arg3) {
        simplePush("LCL");

        simplePush("ARG");

        simplePush("THIS");

        simplePush("THAT");

        w.println("@SP");                  // Repositions ARG to SP-5-nArgs.
        w.println("D=M");                  // D contains SP
        w.println("@5");
        w.println("D=D-A");                // D contains SP-5.
        w.println("@" + arg3);
        w.println("D=D-A");                // D contains (SP-5)-nArgs.
        w.println("@ARG");                 // Set new ARG value.
        w.println("M=D");

        w.println("@SP"); // LCL = SP
        w.println("D=M");
        w.println("@LCL");
        w.println("M=D");
    }

    private void writeDreturn() {
        w.println("@LCL");  // Save LCL value to R15.
        w.println("D=M");
        w.println("@R15");
        w.println("M=D");

        w.println("@5");    // Compute LCL-5, which contains the value of returnAddress.
        w.println("D=D-A");
        w.println("A=D");   // Points to LCL-5.
        w.println("D=M");   // Gets its content, which is returnAddress.
        w.println("@R14");  // Saves it to R14
        w.println("M=D");

        simplerReinstate("SP", "ARG"); // Gets the last element of stack (returnValue),
decreases SP and points to ARG.
        w.println("A=M");                              // Points to ARG[0].
        w.println("M=D");                              // Puts returnValue to ARG[0].

        w.println("@ARG"); // SP=ARG+1
        w.println("D=M");
        w.println("@SP");
        w.println("M=D+1");

        simpleReinstate("R15", "THAT"); // Reinstates THAT.
```

```
        simpleReinstate("R15", "THIS"); // Reinstates THIS.

        simpleReinstate("R15", "ARG");  // Reinstates ARG.

        simpleReinstate("R15", "LCL");  // Reinstates LCL.

        w.println("@R14");  // Points to R14 which contains returnAddress.
        w.println("A=M");   // Points to returnAddress.
        w.println("0;JEQ"); // Jumps to returnAddress.
    }

    private void simpleReinstate(String address, String destination) {
        simplerReinstate(address, destination); // Gets value to reinstate and points to
the destination.
        w.println("M=D");                       // Reinstates the value.
    }

    private void simplerReinstate(String address, String destination) {
        w.println("@" + address);     // Points to the cell containing the address of the
value to reinstate (usually R15 or SP).
        w.println("M=M-1");           // Decreases it.
        w.println("A=M");             // Points to the address.
        w.println("D=M");             // Gets the value to reinstate.
        w.println("@" + destination); // Points to the destination.
    }

    private void writeDfunction() {
        w.println("(" + parse.arg2(line) + ")");                    // Sets label for the
beginning of function. The name of the label is the same as the function.
        int args = Integer.parseInt(parse.arg3(line));              // Retrieves number
of arguments which is argument 3.
        for(int i=0; i<args; i++) translateLine("push constant 0"); // Pushes constant 0
as many times as there are arguments.
    }

    public void exit() {
        try{
            w.close();
            r.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```