Project 7

- Link to github repo: https://github.com/kdub8/CS3650-Project-7-NAND2Tetris.git
- Source code:

```
/*
 * Author: Kevin Wong
 * Assignment: NAND2TETRIS Project 7
 * Date: 11/2/2023
 * Professor: Nima Davarpanah
 * Course: CS3650-01
 * File: CodeWriter.java
 * File Description: This class is designed for translating VM (Virtual Machine)
 * commands into HACK assembly code. Its constructor initializes
 * an output file and a print writer, while a method named `setFileName` informs the class of a new VM file translation. The core
 * functionality lies in methods like `writeArithmetic` and `writePushPop`. The former generates assembly code for various arithmetic
 * operations, such as addition, subtraction, logical operations, and comparisons, employing specific templates for each. The
 * latter handles push and pop operations for memory segments, including local, argument, this, that, temp, pointer, and static.
 * The generated assembly code is written to the output file. The class also includes private helper methods with assembly templates
 * for code generation.
 */
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;

/**
 * Translates VM commands into HACK assembly code
 */
public class CodeWriter {

    private int arthJumpFlag;
    private PrintWriter outPrinter;

    /**
     * Open an output file and be ready to write content
     *
     * @param fileOut can be a directory!
     */
    public CodeWriter(File fileOut) {

        try {

            outPrinter = new PrintWriter(fileOut);
            arthJumpFlag = 0;
```

```
    } catch (FileNotFoundException e) {

        e.printStackTrace();

    }

}

/**
 * If the program's argument is a directory name rather than a file name,
 * the main program should process all the .vm files in this directory.
 * In doing so, it should use a separate Parser for handling each input file and
 * a single CodeWriter for handling the output.
 *
 * Inform the CodeWrither that the translation of a new VM file is started
 */
public void setFileName(File fileOut) {

}

/**
 * Write the assembly code that is the translation of the given arithmetic
 * command
 *
 * @param command
 */
public void writeArithmetic(String command) {

    if (command.equals("add")) {

        outPrinter.print(arithmeticTemplate1() + "M=M+D\n");

    } else if (command.equals("sub")) {

        outPrinter.print(arithmeticTemplate1() + "M=M-D\n");

    } else if (command.equals("and")) {

        outPrinter.print(arithmeticTemplate1() + "M=M&D\n");

    } else if (command.equals("or")) {

        outPrinter.print(arithmeticTemplate1() + "M=M|D\n");

    } else if (command.equals("gt")) {

        outPrinter.print(arithmeticTemplate2("JLE"));// not <=
        arthJumpFlag++;
```

```
    } else if (command.equals("lt")) {

        outPrinter.print(arithmeticTemplate2("JGE"));// not >=
        arthJumpFlag++;

    } else if (command.equals("eq")) {

        outPrinter.print(arithmeticTemplate2("JNE"));// not <>
        arthJumpFlag++;

    } else if (command.equals("not")) {

        outPrinter.print("@SP\nA=M-1\nM=!M\n");

    } else if (command.equals("neg")) {

        outPrinter.print("D=0\n@SP\nA=M-1\nM=D-M\n");

    } else {

        throw new IllegalArgumentException("Call writeArithmetic() for a non-arithmetic command");

    }

}

/**
 * Write the assembly code that is the translation of the given command
 * where the command is either PUSH or POP
 *
 * @param command PUSH or POP
 * @param segment
 * @param index
 */
public void writePushPop(int command, String segment, int index) {

    if (command == Parser.PUSH) {

        if (segment.equals("constant")) {

            outPrinter.print("@" + index + "\n" + "D=A\n@SP\nA=M\nM=D\n@SP\nM=M+1\n");

        } else if (segment.equals("local")) {

            outPrinter.print(pushTemplate1("LCL", index, false));

        } else if (segment.equals("argument")) {
```

```
            outPrinter.print(pushTemplate1("ARG", index, false));
        } else if (segment.equals("this")) {
            outPrinter.print(pushTemplate1("THIS", index, false));
        } else if (segment.equals("that")) {
            outPrinter.print(pushTemplate1("THAT", index, false));
        } else if (segment.equals("temp")) {
            outPrinter.print(pushTemplate1("R5", index + 5, false));
        } else if (segment.equals("pointer") && index == 0) {
            outPrinter.print(pushTemplate1("THIS", index, true));
        } else if (segment.equals("pointer") && index == 1) {
            outPrinter.print(pushTemplate1("THAT", index, true));
        } else if (segment.equals("static")) {
            outPrinter.print(pushTemplate1(String.valueOf(16 + index), index, true));
        }
    } else if (command == Parser.POP) {
        if (segment.equals("local")) {
            outPrinter.print(popTemplate1("LCL", index, false));
        } else if (segment.equals("argument")) {
            outPrinter.print(popTemplate1("ARG", index, false));
        } else if (segment.equals("this")) {
            outPrinter.print(popTemplate1("THIS", index, false));
        } else if (segment.equals("that")) {
            outPrinter.print(popTemplate1("THAT", index, false));
        } else if (segment.equals("temp")) {
            outPrinter.print(popTemplate1("R5", index + 5, false));
```

```
        } else if (segment.equals("pointer") && index == 0) {
            outPrinter.print(popTemplate1("THIS", index, true));
        } else if (segment.equals("pointer") && index == 1) {
            outPrinter.print(popTemplate1("THAT", index, true));
        } else if (segment.equals("static")) {
            outPrinter.print(popTemplate1(String.valueOf(16 + index), index, true));
        }
    } else {
        throw new IllegalArgumentException("Call writePushPop() for a non-pushpop command");
    }
}

/**
 * Close the output file
 */
public void close() {
    outPrinter.close();
}

/**
 * Template for add sub and or
 *
 * @return
 */
private String arithmeticTemplate1() {
    return "@SP\n" +
            "AM=M-1\n" +
            "D=M\n" +
            "A=A-1\n";
}

/**
 * Template for gt lt eq
 *
```

```
 * @param type JLE JGT JEQ
 * @return
 */
private String arithmeticTemplate2(String type) {

    return "@SP\n" +
            "AM=M-1\n" +
            "D=M\n" +
            "A=A-1\n" +
            "D=M-D\n" +
            "@FALSE" + arthJumpFlag + "\n" +
            "D;" + type + "\n" +
            "@SP\n" +
            "A=M-1\n" +
            "M=-1\n" +
            "@CONTINUE" + arthJumpFlag + "\n" +
            "0;JMP\n" +
            "(FALSE" + arthJumpFlag + ")\n" +
            "@SP\n" +
            "A=M-1\n" +
            "M=0\n" +
            "(CONTINUE" + arthJumpFlag + ")\n";

}

/**
 * Template for push local,this,that,argument,temp,pointer,static
 *
 * @param segment
 * @param index
 * @param isDirect Is this command a direct addressing?
 * @return
 */
private String pushTemplate1(String segment, int index, boolean isDirect) {

    // When it is a pointer, just read the data stored in THIS or THAT
    // When it is static, just read the data stored in that address
    String noPointerCode = (isDirect) ? "" : "@" + index + "\n" + "A=D+A\nD=M\n";

    return "@" + segment + "\n" +
            "D=M\n" +
            noPointerCode +
            "@SP\n" +
            "A=M\n" +
            "M=D\n" +
            "@SP\n" +
            "M=M+1\n";

}
```

```
    /**
     * Template for pop local,this,that,argument,temp,pointer,static
     *
     * @param segment
     * @param index
     * @param isDirect Is this command a direct addressing?
     * @return
     */
    private String popTemplate1(String segment, int index, boolean isDirect) {

        // When it is a pointer R13 will store the address of THIS or THAT
        // When it is a static R13 will store the index address
        String noPointerCode = (isDirect) ? "D=A\n" : "D=M\n@" + index + "\nD=D+A\n";

        return "@" + segment + "\n" +
                noPointerCode +
                "@R13\n" +
                "M=D\n" +
                "@SP\n" +
                "AM=M-1\n" +
                "D=M\n" +
                "@R13\n" +
                "A=M\n" +
                "M=D\n";

    }

}


/*
 * Author: Kevin Wong
 * Assignment: NAND2TETRIS Project 7
 * Date: 11/2/2023
 * Professor: Nima Davarpanah
 * Course: CS3650-01
 * File: CodeWriter.java
 * File Description: The `VMTranslator` class functions as the main entry point for a Virtual Machine (VM) to
HACK assembly
 * translator program. It offers a `getVMFiles` method to gather VM files within a directory and a `main` method for
translation.
 * In the main method, it identifies whether the input is a single VM file or a directory containing VM files. For a
single file,
 * it checks its extension and adds it to a processing list, determining the output file's path. In the case of a directory,
it
 * compiles all the VM files and sets the output file path accordingly. A `CodeWriter` object is used to generate
assembly code,
```

```
 * with a `Parser` handling the parsing of VM commands. The translated assembly code is saved to the output file,
and a message
 * confirms the file's creation.
 */
import java.io.File;
import java.util.ArrayList;

/**
 *
 */
public class VMTranslator {

    /**
     * Return all the .vm files in a directory
     *
     * @param dir
     * @return
     */
    public static ArrayList<File> getVMFiles(File dir) {
        File[] files = dir.listFiles();
        ArrayList<File> result = new ArrayList<File>();
        for (File f : files) {
            if (f.getName().endsWith(".vm")) {
                result.add(f);
            }
        }
        return result;
    }

    public static void main(String[] args) {

        if (args.length != 1) {
            System.out.println("Usage:java VMtranslator [filename|directory]");
        } else {
            File fileIn = new File(args[0]);
            String fileOutPath = "";
            File fileOut;
            CodeWriter writer;
            ArrayList<File> vmFiles = new ArrayList<File>();
            if (fileIn.isFile()) {
                String path = fileIn.getAbsolutePath();
                if (!Parser.getExt(path).equals(".vm")) {
                    throw new IllegalArgumentException(".vm file is required!");
                }
                vmFiles.add(fileIn);
                fileOutPath = fileIn.getAbsolutePath().substring(0, fileIn.getAbsolutePath().lastIndexOf(".")) + ".asm";
            } else if (fileIn.isDirectory()) {
                // if it is a directory get all vm files under this directory
                vmFiles = getVMFiles(fileIn);
```

```
            // if no vm file in this directory
            if (vmFiles.size() == 0) {
                throw new IllegalArgumentException("No vm file in this directory");
            }
            fileOutPath = fileIn.getAbsolutePath() + "/" + fileIn.getName() + ".asm";
        }
        fileOut = new File(fileOutPath);
        writer = new CodeWriter(fileOut);
        for (File f : vmFiles) {
            Parser parser = new Parser(f);
            int type = -1;
            // start parsing
            while (parser.hasMoreCommands()) {
                parser.advance();
                type = parser.commandType();
                if (type == Parser.ARITHMETIC) {
                    writer.writeArithmetic(parser.arg1());
                } else if (type == Parser.POP || type == Parser.PUSH) {

                    writer.writePushPop(type, parser.arg1(), parser.arg2());
                }
            }
        }
        // save file
        writer.close();
        System.out.println("File created : " + fileOutPath);
    }
}

}


/*
 * Author: Kevin Wong
 * Assignment: NAND2TETRIS Project 7
 * Date: 11/2/2023
 * Professor: Nima Davarpanah
 * Course: CS3650-01
 * File: CodeWriter.java
 * File Description: The `Parser` class handles the parsing of VM (Virtual Machine) commands from a `.vm` file.
 * It covers access to the input code and provides methods to extract information about the commands. The class
preprocesses
 * the input by removing comments and white spaces, simplifying parsing. It offers methods to check for more
commands, advance to
 * the next command, determine the command type (e.g., ARITHMETIC, PUSH, POP), and retrieve the command's
arguments. The class
 * also maintains constants for different command types and a list of valid arithmetic commands for categorization.
 */
import java.io.File;
```

```
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * Handles the parsing of a single .vm file, and encapsulates access to the
 * input code.
 * It reads VM commands, parses them, and provides convenient access to their
 * components.
 * In addition, it removes all white space and comments.
 */
public class Parser {
    private Scanner cmds;
    private String currentCmd;
    public static final int ARITHMETIC = 0;
    public static final int PUSH = 1;
    public static final int POP = 2;
    public static final int LABEL = 3;
    public static final int GOTO = 4;
    public static final int IF = 5;
    public static final int FUNCTION = 6;
    public static final int RETURN = 7;
    public static final int CALL = 8;
    public static final ArrayList<String> arithmeticCmds = new ArrayList<String>();
    private int argType;
    private String argument1;
    private int argument2;

    static {

        arithmeticCmds.add("add");
        arithmeticCmds.add("sub");
        arithmeticCmds.add("neg");
        arithmeticCmds.add("eq");
        arithmeticCmds.add("gt");
        arithmeticCmds.add("lt");
        arithmeticCmds.add("and");
        arithmeticCmds.add("or");
        arithmeticCmds.add("not");

    }

    /**
     * Opens the input file and get ready to parse it
     *
     * @param fileIn
     */
    public Parser(File fileIn) {
```

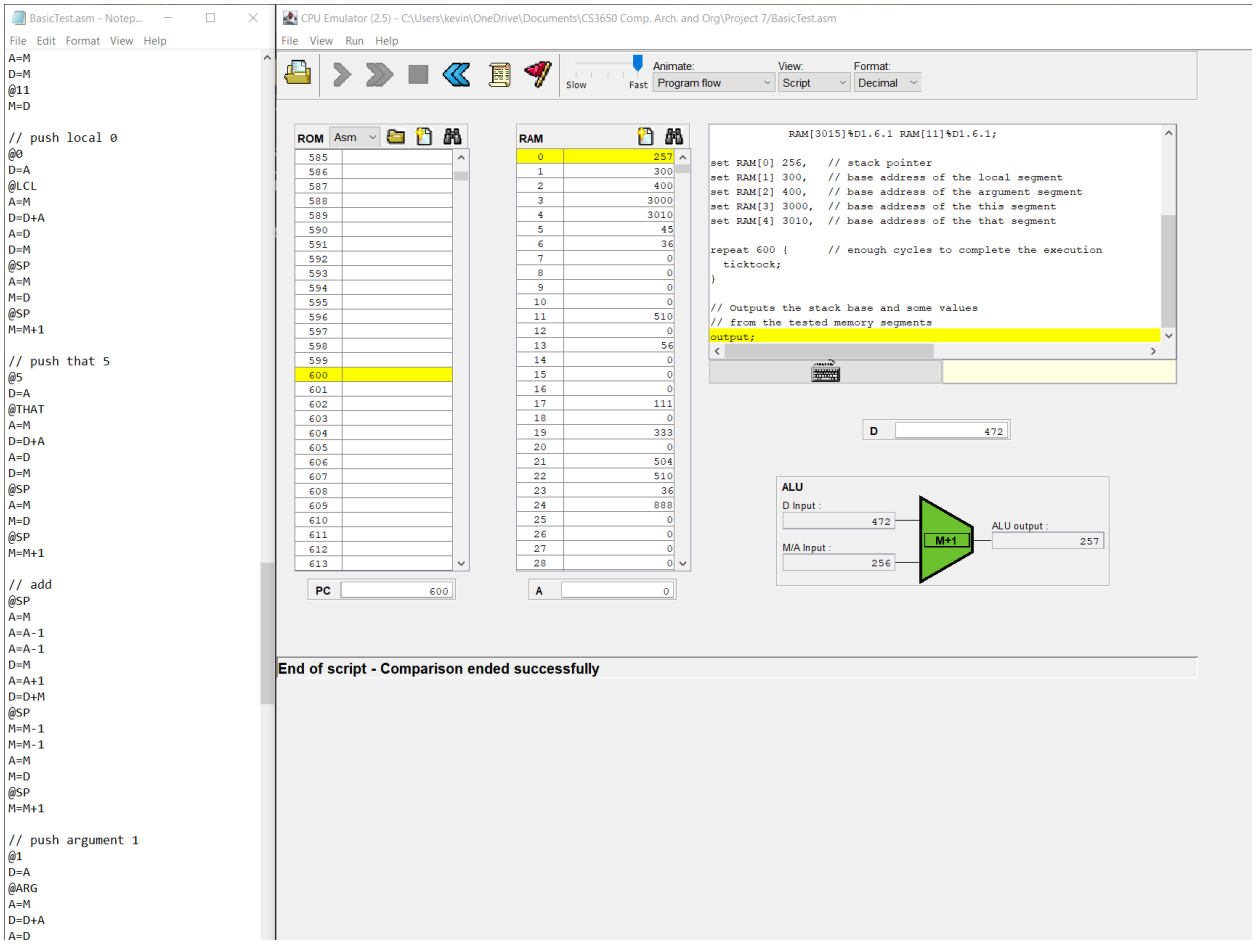
```
    argType = -1;
    argument1 = "";
    argument2 = -1;

    try {
        cmds = new Scanner(fileIn);

        String preprocessed = "";
        String line = "";

        while (cmds.hasNext()) {

            line = noComments(cmds.nextLine()).trim();

            if (line.length() > 0) {
                preprocessed += line + "\n";
            }
        }
        cmds = new Scanner(preprocessed.trim());
    } catch (FileNotFoundException e) {
        System.out.println("File not found!");
    }

}

/**
 * Are there more command to read
 *
 * @return
 */
public boolean hasMoreCommands() {

    return cmds.hasNextLine();
}

/**
 * Reads next command from the input and makes it current command
 * Be called only when hasMoreCommands() returns true
 */
public void advance() {

    currentCmd = cmds.nextLine();
    argument1 = "";// initialize arg1
    argument2 = -1;// initialize arg2

    String[] segs = currentCmd.split(" ");

    if (segs.length > 3) {
```

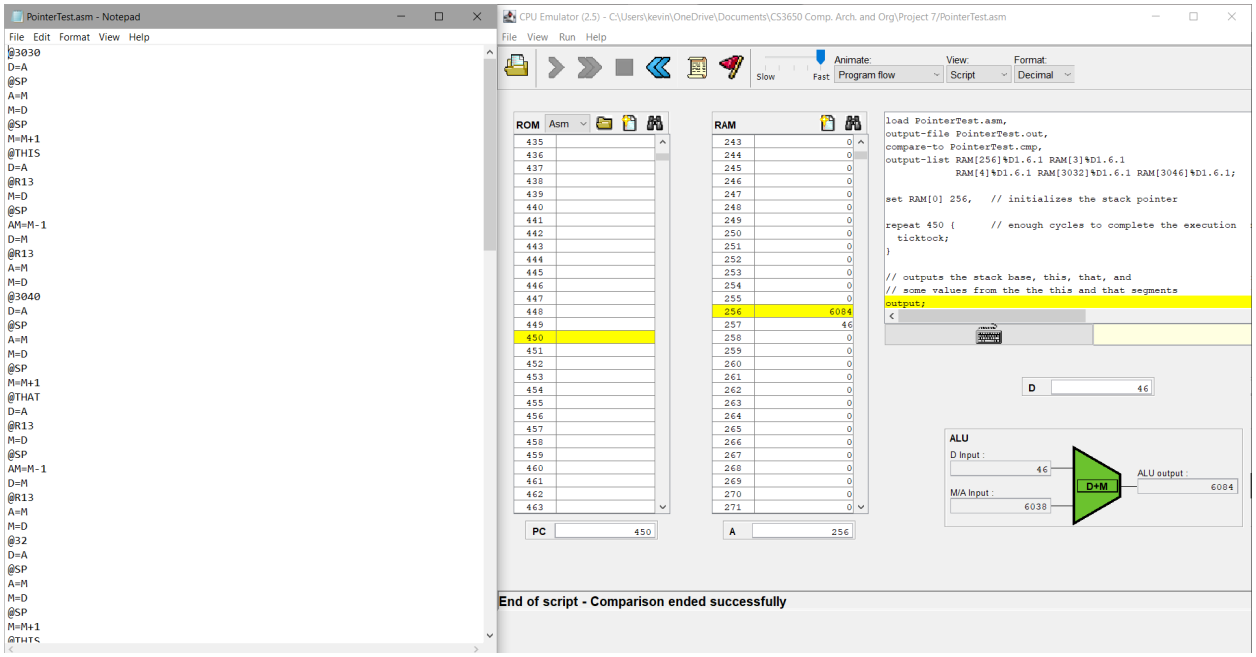
```
            throw new IllegalArgumentException("Too much arguments!");

        }

        if (arithmeticCmds.contains(segs[0])) {
            argType = ARITHMETIC;
            argument1 = segs[0];
        } else if (segs[0].equals("return")) {
            argType = RETURN;
            argument1 = segs[0];
        } else {
            argument1 = segs[1];
            if (segs[0].equals("push")) {
                argType = PUSH;
            } else if (segs[0].equals("pop")) {
                argType = POP;
            } else if (segs[0].equals("label")) {
                argType = LABEL;
            } else if (segs[0].equals("if")) {
                argType = IF;
            } else if (segs[0].equals("goto")) {
                argType = GOTO;
            } else if (segs[0].equals("function")) {
                argType = FUNCTION;
            } else if (segs[0].equals("call")) {
                argType = CALL;
            } else {
                throw new IllegalArgumentException("Unknown Command Type!");
            }

            if (argType == PUSH || argType == POP || argType == FUNCTION || argType == CALL) {
                try {
                    argument2 = Integer.parseInt(segs[2]);
                } catch (Exception e) {
                    throw new IllegalArgumentException("Argument2 is not an integer!");
                }
            }
        }
    }

    /**
     * Return the type of current command
     * ARITHMETIC is returned for all ARITHMETIC type command
     *
     * @return
     */
    public int commandType() {

        if (argType != -1) {
```

```
            return argType;
        } else {
            throw new IllegalStateException("No command!");
        }
    }

    /**
     * Return the first argument of current command
     * When it is ARITHMETIC, return it self
     * When it is RETURN, should not to be called
     *
     * @return
     */
    public String arg1() {

        if (commandType() != RETURN) {
            return argument1;
        } else {
            throw new IllegalStateException("Can not get arg1 from a RETURN type command!");
        }

    }

    /**
     * Return the second argument of current command
     * Be called when it is PUSH, POP, FUNCTION or CALL
     *
     * @return
     */
    public int arg2() {

        if (commandType() == PUSH || commandType() == POP || commandType() == FUNCTION || commandType()
== CALL) {
            return argument2;
        } else {
            throw new IllegalStateException("Can not get arg2!");
        }
    }

    /**
     * Delete comments(String after "//") from a String
     *
     * @param strIn
     * @return
     */
    public static String noComments(String strIn) {

        int position = strIn.indexOf("//");
```

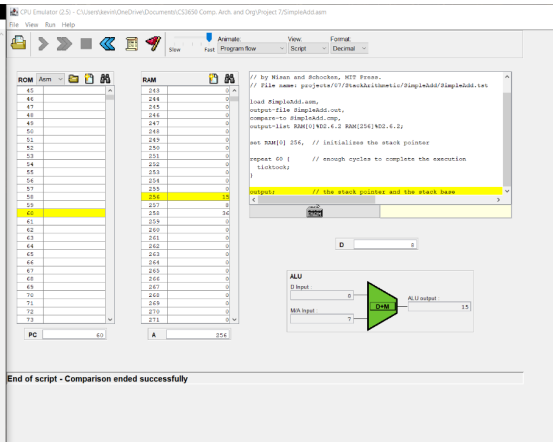
```
        if (position != -1) {
            strIn = strIn.substring(0, position);
        }
        return strIn;
    }

    /**
     * Delete spaces from a String
     *
     * @param strIn
     * @return
     */
    public static String noSpaces(String strIn) {
        String result = "";

        if (strIn.length() != 0) {

            String[] segs = strIn.split(" ");

            for (String s : segs) {
                result += s;
            }
        }

        return result;
    }

    /**
     * Get extension from a filename
     *
     * @param fileName
     * @return
     */
    public static String getExt(String fileName) {
        int index = fileName.lastIndexOf('.');
        if (index != -1) {
            return fileName.substring(index);
        } else {
            return "";
        }
    }
}
```

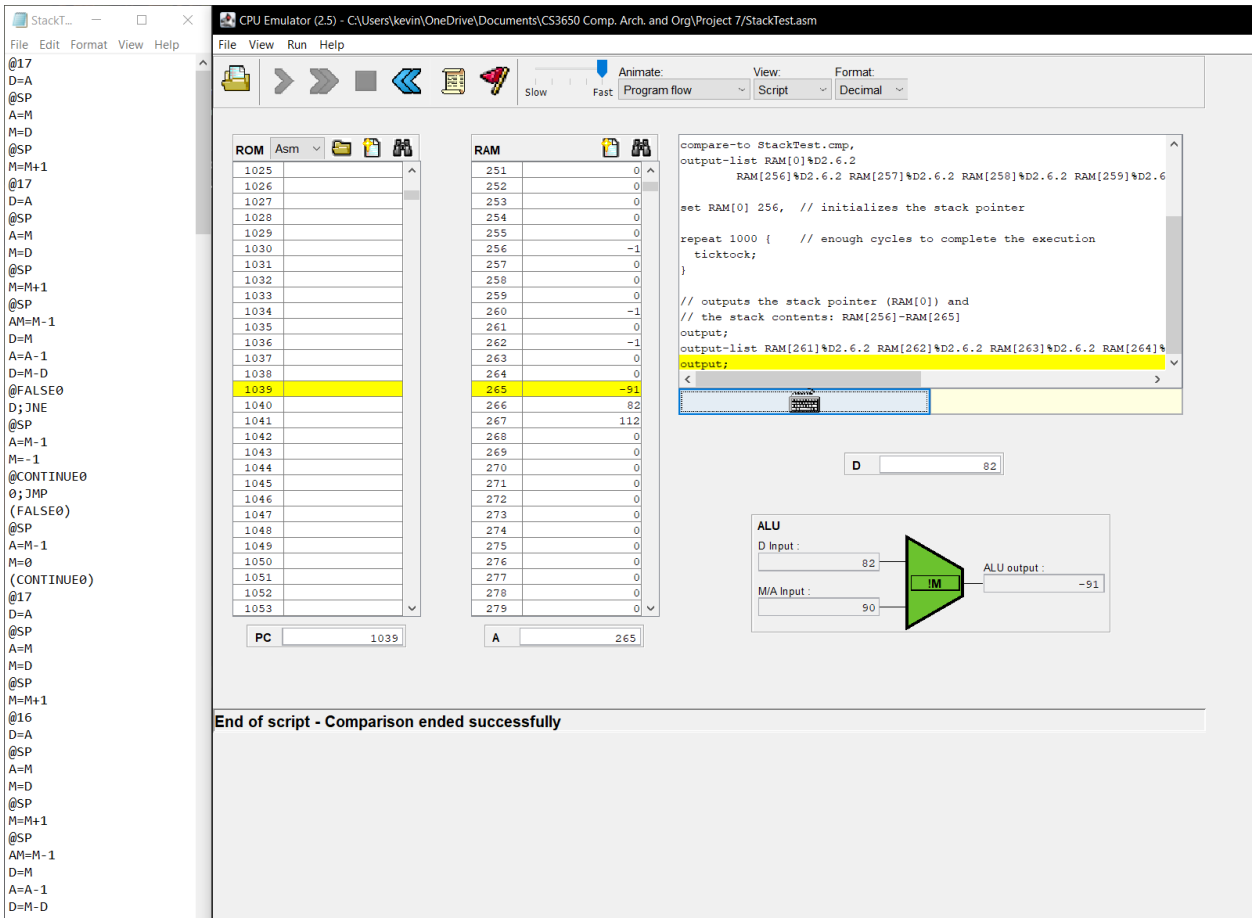
-Screenshots of tests

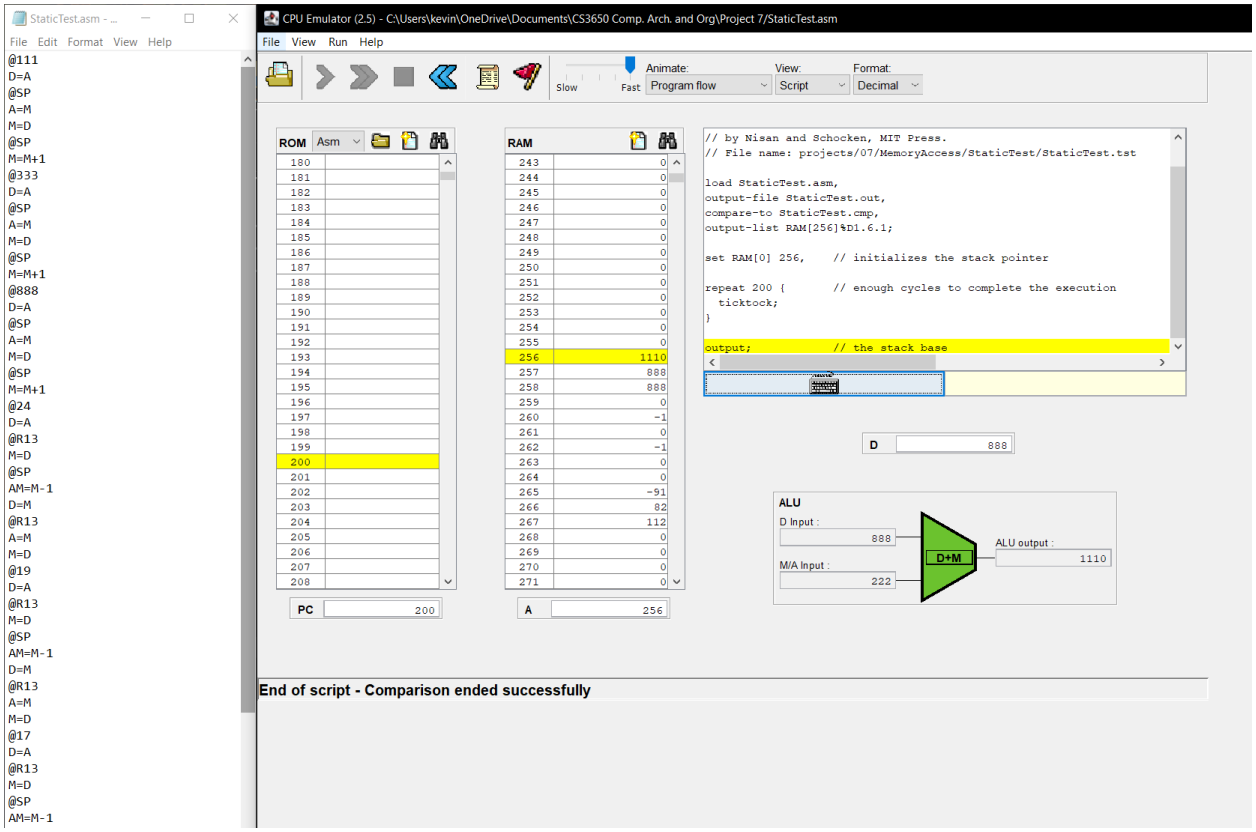

BasicTest.asm - Notep...
File Edit Format View Help
A=M
D=M
@11
M=D
// push local 0
@0
D=A
@LCL
A=M
D=D+A
A=D
D=M
@SP
A=M
M=D
@SP
M=M+1
// push that 5
@5
D=A
@THAT
A=M
D=D+A
A=D
D=M
@SP
A=M
M=D
@SP
M=M+1
// add
@SP
A=M
A=A-1
A=A-1
D=M
A=A+1
D=D+M
@SP
M=M-1
M=M-1
A=M
M=D
@SP
M=M+1
// push argument 1
@1
D=A
@ARG
A=M
D=D+A
A=D
CPU Emulator (2.5) - C:\Users\kevin\OneDrive\Documents\CS3650 Comp. Arch. and Org\Project 7/BasicTest.asm
File View Run Help
Slow
Fast
Animate:
Program flow
View:
Script
Format:
Decimal
ROM
Asm
RAM
RAM[3015]%D1.6.1 RAM[11]%D1.6.1;
set RAM[0] 256, // stack pointer
set RAM[1] 300, // base address of the local segment
set RAM[2] 400, // base address of the argument segment
set RAM[3] 3000, // base address of the this segment
set RAM[4] 3010, // base address of the that segment
repeat 600 { // enough cycles to complete the execution
ticktock;
}
// Outputs the stack base and some values
// from the tested memory segments
output;
D
472
ALU
D Input :
472
M/A Input :
256
M+1
ALU output :
257
PC
600
A
0
End of script - Comparison ended successfully


PointerTest.asm - Notepad
File Edit Format View Help
@3030
D=A
@SP
A=M
M=D
@SP
M=M+1
@THIS
D=A
@R13
M=D
@SP
AM=M-1
D=M
@R13
A=M
M=D
@3040
D=A
@SP
A=M
M=D
@SP
M=M+1
@THAT
D=A
@R13
M=D
@SP
AM=M-1
D=M
@R13
A=M
M=D
@32
D=A
@SP
A=M
M=D
@SP
M=M+1
CPU Emulator (2.5) - C:\Users\kevin\OneDrive\Documents\CS3650 Comp. Arch. and Org\Project 7/PointerTest.asm
File View Run Help
Slow
Fast
Animate:
Program flow
View:
Script
Format:
Decimal
ROM
Asm
RAM
load PointerTest.asm,
output-file PointerTest.out,
compare-to PointerTest.cmp,
output-list RAM[256]%D1.6.1 RAM[3]%D1.6.1
RAM[4]%D1.6.1 RAM[3032]%D1.6.1 RAM[3046]%D1.6.1;
set RAM[0] 256, // initializes the stack pointer
repeat 450 { // enough cycles to complete the execution
ticktock;
}
// outputs the stack base, this, that, and
// some values from the the this and that segments
output;
D
46
ALU
D Input :
46
M/A Input :
6038
D+M
ALU output :
6084
PC
450
A
256
End of script - Comparison ended successfully


SimpleAdd.asm - Notepad
File Edit Format View Help
@7
D=A
@SP
A=M
M=D
@SP
M=M+1
@8
D=A
@SP
A=M
M=D
@SP
M=M+1
@SP
AM=M-1
D=M
A=A-1
M=M+D


CPU Emulator (2.5) - C:\Users\kevin\OneDrive\Documents\CS3650 Comp. Arch. and Org\Project 7\SimpleAdd.asm
File View Run Help
Animate:
Slow
Fast
Program flow
View:
Script
Format:
Decimal
ROM
Asm
RAM
// by Nisan and Schocken, MIT Press.
// File name: projects/07/StackArithmetic/SimpleAdd/SimpleAdd.tst
load SimpleAdd.asm,
output-file SimpleAdd.out,
compare-to SimpleAdd.cmp,
output-list RAM[0]%D2.6.2 RAM[256]%D2.6.2;
set RAM[0] 256, // initialises the stack pointer
repeat 60 { // enough cycles to complete the execution
ticktock;
}
output; // the stack pointer and the stack base
Select
D
ALU
D Input
M/A Input
D+M
ALU output
PC
A
End of script - Comparison ended successfully


StackT...
File Edit Format View Help
@17
D=A
@SP
A=M
M=D
@SP
M=M+1
@17
D=A
@SP
A=M
M=D
@SP
M=M+1
@SP
AM=M-1
D=M
A=A-1
D=M-D
@FALSE0
D;JNE
@SP
A=M-1
M=-1
@CONTINUE0
0;JMP
(FALSE0)
@SP
A=M-1
M=0
(CONTINUE0)
@17
D=A
@SP
A=M
M=D
@SP
M=M+1
@16
D=A
@SP
A=M
M=D
@SP
M=M+1
@SP
AM=M-1
D=M
A=A-1
D=M-D
CPU Emulator (2.5) - C:\Users\kevin\OneDrive\Documents\CS3650 Comp. Arch. and Org\Project 7/StackTest.asm
File View Run Help
Slow
Fast
Animate:
Program flow
View:
Script
Format:
Decimal
ROM
Asm
RAM
PC
1039
A
265
compare-to StackTest.cmp,
output-list RAM[0]%D2.6.2
RAM[256]%D2.6.2 RAM[257]%D2.6.2 RAM[258]%D2.6.2 RAM[259]%D2.6
set RAM[0] 256, // initializes the stack pointer
repeat 1000 { // enough cycles to complete the execution
ticktock;
}
// outputs the stack pointer (RAM[0]) and
// the stack contents: RAM[256]-RAM[265]
output;
output-list RAM[261]%D2.6.2 RAM[262]%D2.6.2 RAM[263]%D2.6.2 RAM[264]%
output;
D
82
ALU
D Input :
82
M/A Input :
90
!M
ALU output :
-91
End of script - Comparison ended successfully



Project 8

- Link to github repo: https://github.com/kdub8/CS3650-Project-8-NAND2Tetris.git
- Source code:

```
/*
 * Author: Kevin Wong
 * Assignment: NAND2TETRIS Project 8
 * Date: 11/2/2023
 * Professor: Nima Davarpanah
 * Course: CS3650-01
 * File: CodeWriter.java
 * File Description: The `CodeWriter` class is responsible for translating VM commands into HACK assembly code.
 * It opens an output file for writing the assembly code, and it offers methods for writing various types of VM commands,
 * including arithmetic, push, pop, label, goto, if-goto, function, return, and call commands. The class maintains internal
 * variables for label management, jump flags, and the current file name being processed. It also provides templates for
 * generating assembly code for various command types, handling memory access and control flow.
 */
import java.io.File;
import java.io.FileNotFoundException;
```

```
import java.io.PrintWriter;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * Translates VM commands into HACK assembly code
 */
public class CodeWriter {

    private int arthJumpFlag;
    private PrintWriter outPrinter;
    private static final Pattern labelReg = Pattern.compile("^[^0-9][0-9A-Za-z\\_\\:\\.\\$]+");
    private static int labelCnt = 0;
    private static String fileName = "";

    /**
     * Open an output file and be ready to write content
     *
     * @param fileOut can be a directory
     */
    public CodeWriter(File fileOut) {

        try {

            fileName = fileOut.getName();
            outPrinter = new PrintWriter(fileOut);
            arthJumpFlag = 0;

        } catch (FileNotFoundException e) {

            e.printStackTrace();

        }

    }

    /**
     * If the program's argument is a directory name rather than a file name,
     * the main program should process all the .vm files in this directory.
     * In doing so, it should use a separate Parser for handling each input file and
     * a single CodeWriter for handling the output.
     *
     * Inform the CodeWrither that the translation of a new VM file is started
     */
    public void setFileName(File fileOut) {

        fileName = fileOut.getName();

    }
```

```
/**
 * Write the assembly code that is the translation of the given arithmetic
 * command
 *
 * @param command
 */
public void writeArithmetic(String command) {

    if (command.equals("add")) {

        outPrinter.print(arithmeticTemplate1() + "M=M+D\n");

    } else if (command.equals("sub")) {

        outPrinter.print(arithmeticTemplate1() + "M=M-D\n");

    } else if (command.equals("and")) {

        outPrinter.print(arithmeticTemplate1() + "M=M&D\n");

    } else if (command.equals("or")) {

        outPrinter.print(arithmeticTemplate1() + "M=M|D\n");

    } else if (command.equals("gt")) {

        outPrinter.print(arithmeticTemplate2("JLE"));// not <=
        arthJumpFlag++;

    } else if (command.equals("lt")) {

        outPrinter.print(arithmeticTemplate2("JGE"));// not >=
        arthJumpFlag++;

    } else if (command.equals("eq")) {

        outPrinter.print(arithmeticTemplate2("JNE"));// not <>
        arthJumpFlag++;

    } else if (command.equals("not")) {

        outPrinter.print("@SP\nA=M-1\nM=!M\n");

    } else if (command.equals("neg")) {

        outPrinter.print("D=0\n@SP\nA=M-1\nM=D-M\n");

    } else {
```

```
        throw new IllegalArgumentException("Call writeArithmetic() for a non-arithmetic command");

    }

}

/**
 * Write the assembly code that is the translation of the given command
 * where the command is either PUSH or POP
 *
 * @param command PUSH or POP
 * @param segment
 * @param index
 */
public void writePushPop(int command, String segment, int index) {

    if (command == Parser.PUSH) {

        if (segment.equals("constant")) {

            outPrinter.print("@" + index + "\n" + "D=A\n@SP\nA=M\nM=D\n@SP\nM=M+1\n");

        } else if (segment.equals("local")) {

            outPrinter.print(pushTemplate1("LCL", index, false));

        } else if (segment.equals("argument")) {

            outPrinter.print(pushTemplate1("ARG", index, false));

        } else if (segment.equals("this")) {

            outPrinter.print(pushTemplate1("THIS", index, false));

        } else if (segment.equals("that")) {

            outPrinter.print(pushTemplate1("THAT", index, false));

        } else if (segment.equals("temp")) {

            outPrinter.print(pushTemplate1("R5", index + 5, false));

        } else if (segment.equals("pointer") && index == 0) {

            outPrinter.print(pushTemplate1("THIS", index, true));

        } else if (segment.equals("pointer") && index == 1) {
```

```
            outPrinter.print(pushTemplate1("THAT", index, true));

        } else if (segment.equals("static")) {
            // every file has its static space
            outPrinter.print("@" + fileName + index + "\n" + "D=M\n@SP\nA=M\nM=D\n@SP\nM=M+1\n");

        }

    } else if (command == Parser.POP) {

        if (segment.equals("local")) {

            outPrinter.print(popTemplate1("LCL", index, false));

        } else if (segment.equals("argument")) {

            outPrinter.print(popTemplate1("ARG", index, false));

        } else if (segment.equals("this")) {

            outPrinter.print(popTemplate1("THIS", index, false));

        } else if (segment.equals("that")) {

            outPrinter.print(popTemplate1("THAT", index, false));

        } else if (segment.equals("temp")) {

            outPrinter.print(popTemplate1("R5", index + 5, false));

        } else if (segment.equals("pointer") && index == 0) {

            outPrinter.print(popTemplate1("THIS", index, true));

        } else if (segment.equals("pointer") && index == 1) {

            outPrinter.print(popTemplate1("THAT", index, true));

        } else if (segment.equals("static")) {
            // every file has its static space
            outPrinter.print("@" + fileName + index +
"\nD=A\n@R13\nM=D\n@SP\nAM=M-1\nD=M\n@R13\nA=M\nM=D\n");

        }

    } else {

        throw new IllegalArgumentException("Call writePushPop() for a non-pushpop command");
```

```
    }

}

/**
 * Write assembly code that effects the label command
 *
 * @param label
 */
public void writeLabel(String label) {

    Matcher m = labelReg.matcher(label);

    if (m.find()) {

        outPrinter.print("(" + label + ")\n");

    } else {

        throw new IllegalArgumentException("Wrong label format!");

    }

}

/**
 * Write assembly code that effects the goto command
 *
 * @param label
 */
public void writeGoto(String label) {

    Matcher m = labelReg.matcher(label);

    if (m.find()) {

        outPrinter.print("@" + label + "\n0;JMP\n");

    } else {

        throw new IllegalArgumentException("Wrong label format!");

    }

}

/**
 * Write assembly code that effects the if-goto command
 *
```

```
     * @param label
     */
    public void writeIf(String label) {

        Matcher m = labelReg.matcher(label);

        if (m.find()) {

            outPrinter.print(arithmeticTemplate1() + "@" + label + "\nD;JNE\n");

        } else {

            throw new IllegalArgumentException("Wrong label format!");

        }

    }

    /**
     * Write assembly code that effects the VM initialization
     * also called BOOTSTRAP CODE.
     * This code must be placed at the beginning of the output file
     */
    public void writeInit() {

        outPrinter.print("@256\n" +
                "D=A\n" +
                "@SP\n" +
                "M=D\n");
        writeCall("Sys.init", 0);

    }

    /**
     * Write assembly code that effects the call command
     *
     * @param functionName
     * @param numArgs
     */
    public void writeCall(String functionName, int numArgs) {

        String newLabel = "RETURN_LABEL" + (labelCnt++);

        outPrinter.print("@" + newLabel + "\n" + "D=A\n@SP\nA=M\nM=D\n@SP\nM=M+1\n");// push return
address
        outPrinter.print(pushTemplate1("LCL", 0, true));// push LCL
        outPrinter.print(pushTemplate1("ARG", 0, true));// push ARG
        outPrinter.print(pushTemplate1("THIS", 0, true));// push THIS
        outPrinter.print(pushTemplate1("THAT", 0, true));// push THAT
```

```
    outPrinter.print("@SP\n" +
            "D=M\n" +
            "@5\n" +
            "D=D-A\n" +
            "@" + numArgs + "\n" +
            "D=D-A\n" +
            "@ARG\n" +
            "M=D\n" +
            "@SP\n" +
            "D=M\n" +
            "@LCL\n" +
            "M=D\n" +
            "@" + functionName + "\n" +
            "0;JMP\n" +
            "(" + newLabel + ")\n");

}

/**
 * Write assembly code that effects the return command
 */
public void writeReturn() {

    outPrinter.print(returnTemplate());

}

/**
 * Write assembly code that effects the function command
 *
 * @param functionName
 * @param numLocals
 */
public void writeFunction(String functionName, int numLocals) {

    outPrinter.print("(" + functionName + ")\n");

    for (int i = 0; i < numLocals; i++) {

        writePushPop(Parser.PUSH, "constant", 0);

    }

}

/**
 * save value of pre frame to given position
 *
```

```
 * @param position
 * @return
 */
public String preFrameTemplate(String position) {

   return "@R11\n" +
          "D=M-1\n" +
          "AM=D\n" +
          "D=M\n" +
          "@" + position + "\n" +
          "M=D\n";

}

/**
 * assembly code template for return command
 * use R13 for FRAME R14 for RET
 *
 * @return
 */
public String returnTemplate() {

   return "@LCL\n" +
          "D=M\n" +
          "@R11\n" +
          "M=D\n" +
          "@5\n" +
          "A=D-A\n" +
          "D=M\n" +
          "@R12\n" +
          "M=D\n" +
          popTemplate1("ARG", 0, false) +
          "@ARG\n" +
          "D=M\n" +
          "@SP\n" +
          "M=D+1\n" +
          preFrameTemplate("THAT") +
          preFrameTemplate("THIS") +
          preFrameTemplate("ARG") +
          preFrameTemplate("LCL") +
          "@R12\n" +
          "A=M\n" +
          "0;JMP\n";
}

/**
 * Close the output file
 */
public void close() {
```

```
    outPrinter.close();

}

/**
 * Template for add sub and or
 *
 * @return
 */
private String arithmeticTemplate1() {

    return "@SP\n" +
            "AM=M-1\n" +
            "D=M\n" +
            "A=A-1\n";

}

/**
 * Template for gt lt eq
 *
 * @param type JLE JGT JEQ
 * @return
 */
private String arithmeticTemplate2(String type) {

    return "@SP\n" +
            "AM=M-1\n" +
            "D=M\n" +
            "A=A-1\n" +
            "D=M-D\n" +
            "@FALSE" + arthJumpFlag + "\n" +
            "D;" + type + "\n" +
            "@SP\n" +
            "A=M-1\n" +
            "M=-1\n" +
            "@CONTINUE" + arthJumpFlag + "\n" +
            "0;JMP\n" +
            "(FALSE" + arthJumpFlag + ")\n" +
            "@SP\n" +
            "A=M-1\n" +
            "M=0\n" +
            "(CONTINUE" + arthJumpFlag + ")\n";

}

/**
 * Template for push local,this,that,argument,temp,pointer,static
```

```
     *
     * @param segment
     * @param index
     * @param isDirect Is this command a direct addressing?
     * @return
     */
    private String pushTemplate1(String segment, int index, boolean isDirect) {

        // When it is a pointer, just read the data stored in THIS or THAT
        String noPointerCode = (isDirect) ? "" : "@" + index + "\n" + "A=D+A\nD=M\n";

        return "@" + segment + "\n" +
                "D=M\n" +
                noPointerCode +
                "@SP\n" +
                "A=M\n" +
                "M=D\n" +
                "@SP\n" +
                "M=M+1\n";

    }

    /**
     * Template for pop local,this,that,argument,temp,pointer,static
     *
     * @param segment
     * @param index
     * @param isDirect Is this command a direct addressing?
     * @return
     */
    private String popTemplate1(String segment, int index, boolean isDirect) {

        // When it is a pointer R13 will store the address of THIS or THAT
        String noPointerCode = (isDirect) ? "D=A\n" : "D=M\n@" + index + "\nD=D+A\n";

        return "@" + segment + "\n" +
                noPointerCode +
                "@R13\n" +
                "M=D\n" +
                "@SP\n" +
                "AM=M-1\n" +
                "D=M\n" +
                "@R13\n" +
                "A=M\n" +
                "M=D\n";

    }

}
```

```
/*
 * Author: Kevin Wong
 * Assignment: NAND2TETRIS Project 8
 * Date: 11/2/2023
 * Professor: Nima Davarpanah
 * Course: CS3650-01
 * File: Parser.java
 * File Description: The `Parser` class is responsible for handling the parsing of a single `.vm` file and
 * encapsulating access to the input code. It reads VM commands, parses them, and provides convenient access to
their components
 * while removing all white space and comments. The class defines constants for various VM command types,
including arithmetic,
 * push, pop, label, goto, if-goto, function, return, and call commands, with methods to retrieve their details.
 * The class's constructor opens the input file for parsing, and the `advance` method reads the next command from
the input and
 * sets it as the current command. The class provides methods to determine the command type, retrieve the first and
second
 * arguments, and handle different types of VM commands. It also includes utility methods for removing comments,
spaces, and
 *  extracting file extensions.
 */
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * Handles the parsing of a single .vm file, and encapsulates access to the
 * input code.
 * It reads VM commands, parses them, and provides convenient access to their
 * components.
 * In addition, it removes all white space and comments.
 */
public class Parser {
    private Scanner cmds;
    private String currentCmd;
    public static final int ARITHMETIC = 0;
    public static final int PUSH = 1;
    public static final int POP = 2;
    public static final int LABEL = 3;
    public static final int GOTO = 4;
    public static final int IF = 5;
    public static final int FUNCTION = 6;
    public static final int RETURN = 7;
    public static final int CALL = 8;
    public static final ArrayList<String> arithmeticCmds = new ArrayList<String>();
    private int argType;
    private String argument1;
```

```
private int argument2;

static {

   arithmeticCmds.add("add");
   arithmeticCmds.add("sub");
   arithmeticCmds.add("neg");
   arithmeticCmds.add("eq");
   arithmeticCmds.add("gt");
   arithmeticCmds.add("lt");
   arithmeticCmds.add("and");
   arithmeticCmds.add("or");
   arithmeticCmds.add("not");

}

/**
 * Opens the input file and get ready to parse it
 *
 * @param fileIn
 */
public Parser(File fileIn) {

   argType = -1;
   argument1 = "";
   argument2 = -1;

   try {
      cmds = new Scanner(fileIn);

      String preprocessed = "";
      String line = "";

      while (cmds.hasNext()) {

         line = noComments(cmds.nextLine()).trim();

         if (line.length() > 0) {
            preprocessed += line + "\n";
         }
      }

      cmds = new Scanner(preprocessed.trim());

   } catch (FileNotFoundException e) {
      System.out.println("File not found!");
   }

}
```

```
/**
 * Are there more command to read
 *
 * @return
 */
public boolean hasMoreCommands() {

   return cmds.hasNextLine();
}

/**
 * Reads next command from the input and makes it current command
 * Be called only when hasMoreCommands() returns true
 */
public void advance() {

   currentCmd = cmds.nextLine();
   argument1 = "";// initialize arg1
   argument2 = -1;// initialize arg2

   String[] segs = currentCmd.split(" ");

   if (segs.length > 3) {

      throw new IllegalArgumentException("Too much arguments!");

   }

   if (arithmeticCmds.contains(segs[0])) {

      argType = ARITHMETIC;
      argument1 = segs[0];

   } else if (segs[0].equals("return")) {

      argType = RETURN;
      argument1 = segs[0];

   } else {

      argument1 = segs[1];

      if (segs[0].equals("push")) {

         argType = PUSH;

      } else if (segs[0].equals("pop")) {
```

```
            argType = POP;
        } else if (segs[0].equals("label")) {
            argType = LABEL;
        } else if (segs[0].equals("if-goto")) {
            argType = IF;
        } else if (segs[0].equals("goto")) {
            argType = GOTO;
        } else if (segs[0].equals("function")) {
            argType = FUNCTION;
        } else if (segs[0].equals("call")) {
            argType = CALL;
        } else {
            throw new IllegalArgumentException("Unknown Command Type!");
        }
        if (argType == PUSH || argType == POP || argType == FUNCTION || argType == CALL) {
            try {
                argument2 = Integer.parseInt(segs[2]);
            } catch (Exception e) {
                throw new IllegalArgumentException("Argument2 is not an integer!");
            }
        }
    }
}

/**
 * Return the type of current command
 * ARITHMETIC is returned for all ARITHMETIC type command
 *
```

```
     * @return
     */
    public int commandType() {

        if (argType != -1) {

            return argType;

        } else {

            throw new IllegalStateException("No command!");

        }

    }

    /**
     * Return the first argument of current command
     * When it is ARITHMETIC, return it self
     * When it is RETURN, should not to be called
     *
     * @return
     */
    public String arg1() {

        if (commandType() != RETURN) {

            return argument1;

        } else {

            throw new IllegalStateException("Can not get arg1 from a RETURN type command!");

        }

    }

    /**
     * Return the second argument of current command
     * Be called when it is PUSH, POP, FUNCTION or CALL
     *
     * @return
     */
    public int arg2() {

        if (commandType() == PUSH || commandType() == POP || commandType() == FUNCTION || commandType()
== CALL) {

            return argument2;
```

```
    } else {

        throw new IllegalStateException("Can not get arg2!");

    }

}

/**
 * Delete comments(String after "//") from a String
 *
 * @param strIn
 * @return
 */
public static String noComments(String strIn) {

    int position = strIn.indexOf("//");

    if (position != -1) {

        strIn = strIn.substring(0, position);

    }

    return strIn;
}

/**
 * Delete spaces from a String
 *
 * @param strIn
 * @return
 */
public static String noSpaces(String strIn) {
    String result = "";

    if (strIn.length() != 0) {

        String[] segs = strIn.split(" ");

        for (String s : segs) {
            result += s;
        }
    }

    return result;
}
```

```
    /**
     * Get extension from a filename
     *
     * @param fileName
     * @return
     */
    public static String getExt(String fileName) {

        int index = fileName.lastIndexOf('.');

        if (index != -1) {

            return fileName.substring(index);

        } else {

            return "";

        }
    }
}
```

```
/*
 * Author: Kevin Wong
 * Assignment: NAND2TETRIS Project 8
 * Date: 11/2/2023
 * Professor: Nima Davarpanah
 * Course: CS3650-01
 * File: VMTranslator.java
 * File Description: The `VMTranslator` class is the main entry point for translating VM code into assembly
language code.
 * It provides functionality for handling both single VM files and entire directories containing VM files.The
`getVMFiles`
 * method takes a directory and returns all `.vm` files within it, storing them in an `ArrayList`.The `main` method
serves as
 * the application's entry point. It processes command-line arguments to determine whether to translate a single VM
file or
 * all VM files in a directory. It initializes the `CodeWriter` for writing the resulting assembly code and manages the
 * translation process for each VM file.If the input is a single VM file, it checks whether the file has a `.vm`
extension
 * and, if so, translates it to an assembly language file (`.asm`). If the input is a directory, it locates and translates
 * all `.vm` files within the directory.The program also takes care of initializing the output file, writing initialization
 * code, and appropriately handling different types of VM commands like arithmetic, push, pop, label, goto, if-goto,
return,
 * function, and call, by invoking methods of the `CodeWriter` class.The resulting assembly code is saved to an
output file,
 * and the program prints a message indicating the file's creation.
```

```
 */
import java.io.File;
import java.util.ArrayList;

/**
 *
 */
public class VMTranslator {

    /**
     * Return all the .vm files in a directory
     *
     * @param dir
     * @return
     */
    public static ArrayList<File> getVMFiles(File dir) {

        File[] files = dir.listFiles();

        ArrayList<File> result = new ArrayList<File>();

        for (File f : files) {

            if (f.getName().endsWith(".vm")) {

                result.add(f);

            }

        }

        return result;

    }

    public static void main(String[] args) {

        // String fileInName =
        // "/Users/xuchen/Documents/IntroToComputerSystem/nand2tetris/projects/07/MemoryAccess/StaticTest/";

        if (args.length != 1) {

            System.out.println("Usage:java VMtranslator [filename|directory]");

        } else {

            String fileInName = args[0];

            File fileIn = new File(fileInName);
```

```
String fileOutPath = "";

File fileOut;

CodeWriter writer;

ArrayList<File> vmFiles = new ArrayList<File>();

if (fileIn.isFile()) {

   // if it is a single file, see whether it is a vm file
   String path = fileIn.getAbsolutePath();

   if (!Parser.getExt(path).equals(".vm")) {

      throw new IllegalArgumentException(".vm file is required!");

   }

   vmFiles.add(fileIn);

   fileOutPath = fileIn.getAbsolutePath().substring(0, fileIn.getAbsolutePath().lastIndexOf(".")) + ".asm";

} else if (fileIn.isDirectory()) {

   // if it is a directory get all vm files under this directory
   vmFiles = getVMFiles(fileIn);

   // if no vn file in this directory
   if (vmFiles.size() == 0) {

      throw new IllegalArgumentException("No vm file in this directory");

   }

   fileOutPath = fileIn.getAbsolutePath() + "/" + fileIn.getName() + ".asm";
}

fileOut = new File(fileOutPath);
writer = new CodeWriter(fileOut);

writer.writeInit();

for (File f : vmFiles) {

   writer.setFileName(f);

   Parser parser = new Parser(f);
```

```
    int type = -1;

    // start parsing
    while (parser.hasMoreCommands()) {

        parser.advance();

        type = parser.commandType();

        if (type == Parser.ARITHMETIC) {

            writer.writeArithmetic(parser.arg1());

        } else if (type == Parser.POP || type == Parser.PUSH) {

            writer.writePushPop(type, parser.arg1(), parser.arg2());

        } else if (type == Parser.LABEL) {

            writer.writeLabel(parser.arg1());

        } else if (type == Parser.GOTO) {

            writer.writeGoto(parser.arg1());

        } else if (type == Parser.IF) {

            writer.writeIf(parser.arg1());

        } else if (type == Parser.RETURN) {

            writer.writeReturn();

        } else if (type == Parser.FUNCTION) {

            writer.writeFunction(parser.arg1(), parser.arg2());

        } else if (type == Parser.CALL) {

            writer.writeCall(parser.arg1(), parser.arg2());

        }

    }

}

// save file
```

```
            writer.close();

            System.out.println("File created : " + fileOutPath);
        }
    }

}
```

-Screenshots from tests:

FibonacciElement.asm - Notepad

File Edit Format View Help

```
// call Main.fibonacci 1
@RETURN23
D=A
@SP
A=M
M=D // push return-address
@SP
M=M+1
@LCL
D=M
@SP
A=M
M=D // push LCL
@SP
M=M+1
@ARG
D=M
@SP
A=M
M=D // push ARG
@SP
M=M+1
@THIS
D=M
@SP
A=M
M=D // push THIS
@SP
M=M+1
@THAT
D=M
@SP
A=M
M=D // push THAT
@SP
M=M+1
D=M
@1
D=D-A
@5
D=D-A
@ARG
M=D // ARG = SP-n-5
@SP
D=M
@LCL
M=D // LCL = SP
@Main.fibonacci
0;JMP // goto Main.fibonacci
(RETURN23)
```

CPU Emulator (2.5) - C:\Users\kevin\OneDrive\Documents\CS3650 Comp. Arch. and Org\Project 8\FibonacciElement/FibonacciElement.asm

File View Run Help

Animate: Program flow | View: Script | Format: Decimal

ROM (Asm)

442	M=M+1
443	D=M
444	@1
445	D=D-A
446	@5
447	D=D-A
448	@2
449	M=D
450	@0
451	D=M
452	@1
453	M=D
454	@52
455	0;JMP
456	@456
457	0;JMP
458	
459	
460	
461	
462	
463	
464	
465	
466	
467	
468	
469	
470	

PC 457

RAM

0	262
1	261
2	256
3	3010
4	4010
5	45
6	36
7	0
8	0
9	0
10	0
11	263
12	410
13	261
14	0
15	0
16	267
17	456
18	0
19	333
20	0
21	504
22	510
23	36
24	888
25	0
26	0
27	0
28	0

A 456

```
// File name: projects/08/FunctionCalls/FibonacciElement/FibonacciEle

// FibonacciElement.asm results from translating both Main.vm and Sys
// a single assembly program, stored in the file FibonacciElement.asm

load FibonacciElement.asm,
output-file FibonacciElement.out,
compare-to FibonacciElement.cmp,
output-list RAM[0]%D1.6.1 RAM[261]%D1.6.1;

repeat 6000 {
  ticktock;
}

output;
```

D 261

ALU

D Input : 261

M/A Input : 456

0

ALU output : 0

End of script - Comparison ended successfully

SimpleFunction.asm - Notepad

File Edit Format View Help

```
// return
@LCL
D=M
@frame
M=D // FRAME = LCL
@5
D=D-A
A=D
D=M
@ret
M=D // RET = *(FRAME-5)
@SP
M=M-1
A=M
D=M
@ARG
A=M
M=D // *ARG = pop
@ARG
D=M+1
@SP
M=D // SP = ARG+1
@frame
D=M
@1
D=D-A
A=D
D=M
@THAT
M=D // THAT = *(FRAME-1)
@frame
D=M
@2
D=D-A
A=D
D=M
@THIS
M=D // THIS = *(FRAME-2)
@frame
D=M
@3
D=D-A
A=D
D=M
@ARG
M=D // ARG = *(FRAME-3)
@frame
D=M
```

CPU Emulator (2.5) - C:\Users\kevin\OneDrive\Documents\CS3650 Comp. Arch. and Org\Project 8/SimpleFunction.asm

File View Run Help

Animate: Program flow | View: Script | Format: Decimal

ROM (Asm)

1121	
1122	
1123	
1124	
1125	
1126	
1127	
1128	
1129	
1130	
1131	
1132	
1133	
1134	
1135	
1136	
1137	
1138	
1139	
1140	
1141	
1142	
1143	
1144	
1145	
1146	
1147	
1148	
1149	

PC 1138

RAM

0	311
1	305
2	300
3	3010
4	4010
5	45
6	36
7	0
8	0
9	0
10	0
11	510
12	0
13	56
14	0
15	0
16	317
17	1000
18	0
19	333
20	0
21	504
22	510
23	36
24	888
25	0
26	0
27	0
28	0

A 1000

```
set RAM[3] 3000,
set RAM[4] 4000,
set RAM[310] 1234,
set RAM[311] 37,
set RAM[312] 1000,
set RAM[313] 305,
set RAM[314] 300,
set RAM[315] 3010,
set RAM[316] 4010,

repeat 300 {
  ticktock;
}

output;
```

D 305

ALU

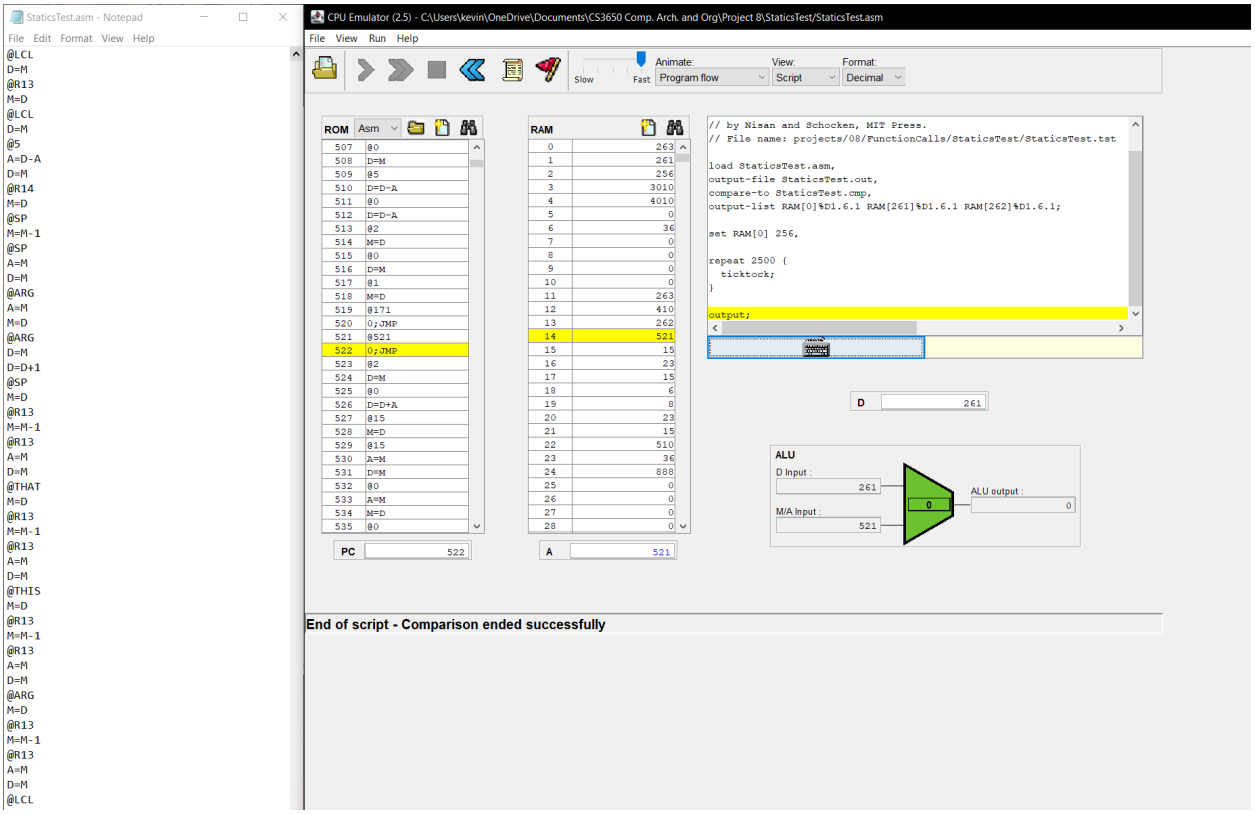
D Input : 305

M/A Input : 1000

0

ALU output : 0

End of script - Comparison ended successfully


StaticsTest.asm - Notepad
File Edit Format View Help
@LCL
D=M
@R13
M=D
@LCL
D=M
@5
A=D-A
D=M
@R14
M=D
@SP
M=M-1
@SP
A=M
D=M
@ARG
A=M
M=D
@ARG
D=M
D=D+1
@SP
M=D
@R13
M=M-1
@R13
A=M
D=M
@THAT
M=D
@R13
M=M-1
@R13
A=M
D=M
@THIS
M=D
@R13
M=M-1
@R13
A=M
D=M
@ARG
M=D
@R13
M=M-1
@R13
A=M
D=M
@LCL
CPU Emulator (2.5) - C:\Users\kevin\OneDrive\Documents\CS3650 Comp. Arch. and Org\Project 8\StaticsTest\StaticsTest.asm
File View Run Help
Slow
Fast
Animate:
Program flow
View:
Script
Format:
Decimal
ROM
Asm
507 @0
508 D=M
509 @5
510 D=D-A
511 @0
512 D=D-A
513 @2
514 M=D
515 @0
516 D=M
517 @1
518 M=D
519 @171
520 0;JMP
521 @521
522 0;JMP
523 @2
524 D=M
525 @0
526 D=D+A
527 @15
528 M=D
529 @15
530 A=M
531 D=M
532 @0
533 A=M
534 M=D
535 @0
PC 522
RAM
0 263
1 261
2 256
3 3010
4 4010
5 0
6 36
7 0
8 0
9 0
10 0
11 263
12 410
13 262
14 521
15 15
16 23
17 15
18 6
19 8
20 23
21 15
22 510
23 36
24 888
25 0
26 0
27 0
28 0
A 521
// by Nisan and Schocken, MIT Press.
// File name: projects/08/FunctionCalls/StaticsTest/StaticsTest.tst
load StaticsTest.asm,
output-file StaticsTest.out,
compare-to StaticsTest.cmp,
output-list RAM[0]%D1.6.1 RAM[261]%D1.6.1 RAM[262]%D1.6.1;
set RAM[0] 256,
repeat 2500 {
ticktock;
}
output;
D 261
ALU
D Input :
261
M/A Input :
521
0
ALU output :
0
End of script - Comparison ended successfully