MAKING AN ASSEMBLER IN JAVA

(DIGITAL ELECTRONICS)

A PROJECT SUBMITTED BY

YESHWANTH BALAJI

ASWIN RAVIDEV

SARVESH

KISHOR

As a part of the subject

ELEMENTS OF COMPUTING SYSTEMS – 02

Centre for Computational Engineering and Networking

AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112 (INDIA)

April – 2023

DECLARATION

We hereby declare that the project work entitled, “Elements of Computing “submitted to the Department of CEN is a record of the original work done by us under the guidance of Ms. Sree Lakshmi, Amrita Vishwa Vidyapeetham and that it has not been performed for the award of any Degree/Diploma/Associate Fellowship/Fellowship and similar titles if any. The results embodied in this thesis have not been submitted to any other University or Institute for the award of any degree or diploma.

Acknowledgement

This project has been possible due to the sincere and dedicated efforts of many. First, we would like to thank the dean of our college for giving us the opportunity to get involved in a project and express our skill. We thank our elements of computing teacher, Ms. Sreelakshmi for her guidance and support without which this project would have been impossible. Finally, we thank our parents and our classmates who encouraged us throughout the project.

YESHWANTH BALAJI AP – CB.EN.U4AIE22102

ASWIN RAVIDEV – CB.EN.U4AIE22108

KISHOR - CB.EN.U4AIE22128

SARVESH - CB.EN.U4AIE22153

Contents

1. Abstract:

This report is basically made for the code explanation and the stages we went through while building the assembler in Java and successfully finished it. The assembler converts a ".asm" file to a ".hack" file. Here ".asm" file consists of the code in the Assembly language whereas the ".hack" file consists of the code which is converted to binary language. Let's discuss this further in the upcoming pages.

2. Introduction:

This is a Java program that implements an assembler for the Hack computer architecture, which is a simple computer system designed for educational purposes. The program takes as input a file written in the Hack Assembly Language, which is a symbolic representation of the machine language instructions that the computer can execute. The assembler translates this file into binary machine code that can be executed by the computer.

The program uses a HashMap data structure to build a symbol table that maps symbolic names to memory addresses, allowing the assembler to resolve symbolic references to memory locations. It also uses three other HashMaps to map the various fields of the machine language instructions to their binary representations.

The assembler performs two passes over the input file. In the first pass, it builds the symbol table by assigning memory addresses to label definitions, which are enclosed in parentheses. In the second pass, it generates the binary machine code by translating each instruction to its corresponding binary representation.

The resulting binary code can be loaded into the computer's memory and executed by its CPU, allowing the computer to carry out the operations specified in the input file.

3. Methodology:

The main methodology or idea behind making this assembler is reading the lines from the inputted file line by line and evaluating each line. [1] [2]

1. First of all, this assembler has 2 passes. In the first pass, the code gets cleaned up and only the assembly code is left behind. Here clean up in the sense that each and every empty line, lines which are commented.
2. Here we also need to take care of the in-line comments where the inline comments have to be removed without making any changes to the code that is present there.

3. In the second pass the lines that are read are classified into A- instruction and C- instruction and accordingly it is converted to their respective binary lines.

4. Finally, the lines which are converted are written into an output file.

4. Building the code:

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

java.io imports the required functions and simple input-output functions.

java.util.* imports the lists, and hashmaps that are required.

```
public class Assembler {
    private static HashMap<String,Integer> symboltable;
```

Here in this line, we create a new class named Assembler. In that class, we initiate a HashMap in which the input parameters are String and Integer. [2] [3]

4.1 Table creation :

```
static {
    symboltable = new HashMap<String, Integer>();
    symboltable.put("SP",0);
    symboltable.put("LCL",1);
    symboltable.put("ARG",2);
    symboltable.put("THIS",3);
    symboltable.put("THAT",4);
    symboltable.put("R0",0);
    symboltable.put("R1",1);
    symboltable.put("R2",2);
    symboltable.put("R3",3);
    symboltable.put("R4",4);
    symboltable.put("R5",5);
    symboltable.put("R6",6);
    symboltable.put("R7",7);
    symboltable.put("R8",8);
    symboltable.put("R9",9);
    symboltable.put("R10",10);
    symboltable.put("R11",11);
    symboltable.put("R12",12);
    symboltable.put("R13",13);
    symboltable.put("R14",14);
    symboltable.put("R15",15);
    symboltable.put("SCREEN",16384);
    symboltable.put("KBD",24576);
}
```

Here we give the input of the predefined keywords and registers into the HashMap we created which is named as symboltable in this case. [3] [4] [5]

```
private static final Map<String, String> DEST_MAP = new HashMap<String,
String>();
private static final Map<String, String> COMP_MAP = new HashMap<String,
String>();
private static final Map<String, String> JUMP_MAP = new HashMap<String,
String>();



static {
    DEST_MAP.put("", "000");
    DEST_MAP.put("M", "001");
    DEST_MAP.put("D", "010");
    DEST_MAP.put("MD", "011");
    DEST_MAP.put("A", "100");
    DEST_MAP.put("AM", "101");
    DEST_MAP.put("AD", "110");
    DEST_MAP.put("AMD", "111");

    COMP_MAP.put("0", "0101010");
    COMP_MAP.put("1", "0111111");
    COMP_MAP.put("-1", "0111010");
    COMP_MAP.put("D", "0001100");
    COMP_MAP.put("A", "0110000");
    COMP_MAP.put("M","1110000");
    COMP_MAP.put("!D", "0001101");
    COMP_MAP.put("!A", "0110001");
    COMP_MAP.put("!M","1110001");
    COMP_MAP.put("-D", "0001111");
    COMP_MAP.put("-A", "0110011");
    COMP_MAP.put("D+1", "0011111");
    COMP_MAP.put("A+1", "0110111");
    COMP_MAP.put("M+1", "1110111");
    COMP_MAP.put("D-1", "0001110");
    COMP_MAP.put("A-1", "0110010");
    COMP_MAP.put("M-1","1110010");
    COMP_MAP.put("D+A", "0000010");
    COMP_MAP.put("A+D", "0000010");
    COMP_MAP.put("D+M","1000010");
    COMP_MAP.put("M+D","1000010");
    COMP_MAP.put("D-A", "0010011");
    COMP_MAP.put("D-M","1010011");
    COMP_MAP.put("A-D", "0000111");
    COMP_MAP.put("M-D","1000111");
    COMP_MAP.put("D&A", "0000000");
    COMP_MAP.put("D&M","1000000");
    COMP_MAP.put("D|A", "0010101");
    COMP_MAP.put("D|M","1010101");

    JUMP_MAP.put("", "000");
    JUMP_MAP.put("JGT", "001");
    JUMP_MAP.put("JEQ", "010");
    JUMP_MAP.put("JGE", "011");
    JUMP_MAP.put("JLT", "100");
    JUMP_MAP.put("JNE", "101");
    JUMP_MAP.put("JLE", "110");
    JUMP_MAP.put("JMP", "111");
}
```

Same as the previous case where we inserted the predefined variables, likewise here do the same thing by creating 3 more tables for DESTINATION, COMPUTATION, and JUMP and insert the values accordingly.

You can see that in both sections here we can see a common thing called “static”, this static is used to not let the table get edited when the code runs.

4.2 Preprocessing(First pass):

```
private static List<String> preprocess(String fileName) throws IOException
{
    List<String> lines = new ArrayList<String>();
    BufferedReader reader = new BufferedReader(new FileReader(fileName));
    String line;
    while ((line = reader.readLine()) != null) {
        line = line.trim();
        if (line.length() > 0 && !line.startsWith("//")) {
            if (line.contains("//")) {
                line = line.substring(0, line.indexOf("//")).trim();
            }
            lines.add(line);
        }
    }
    reader.close();
    return lines;
}
```

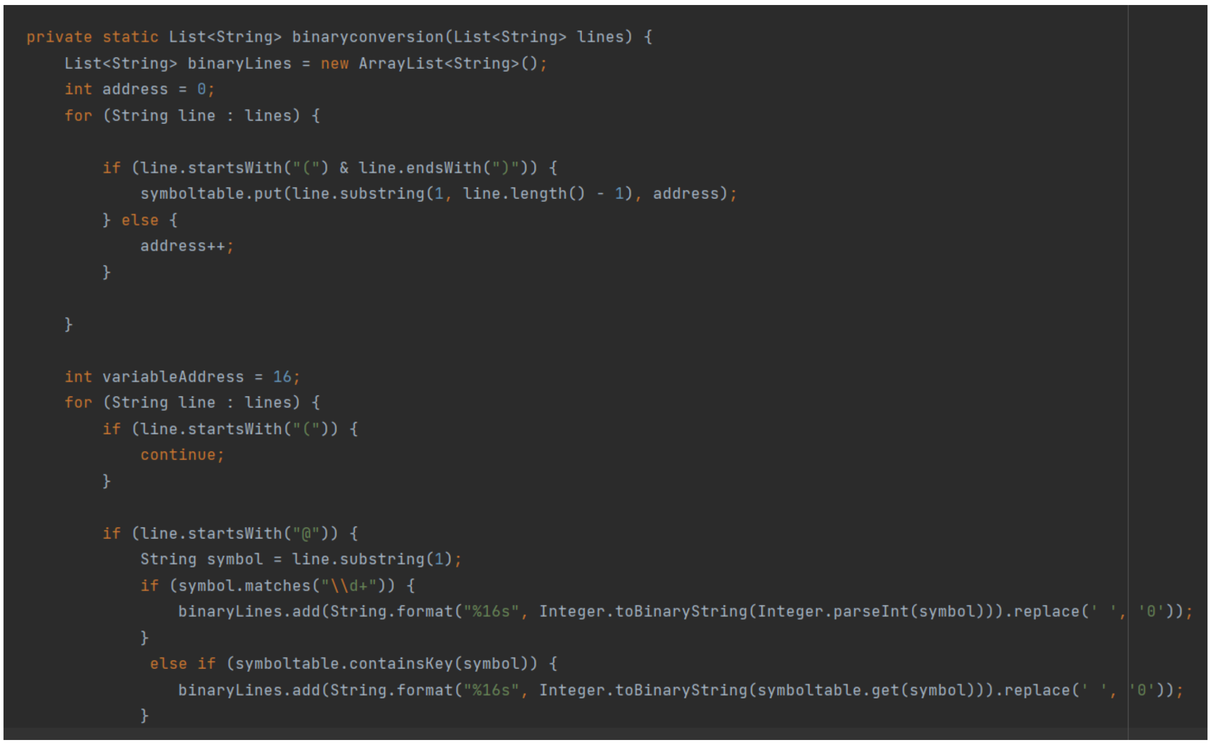
Here we create a method(function) named “preprocess”, this method takes the file as input.

In this method, we create a list which stores the lines that are read. Now we create a new object named reader in “Bufferreader” class which reads the lines and store in the list that we created earlier. [5]

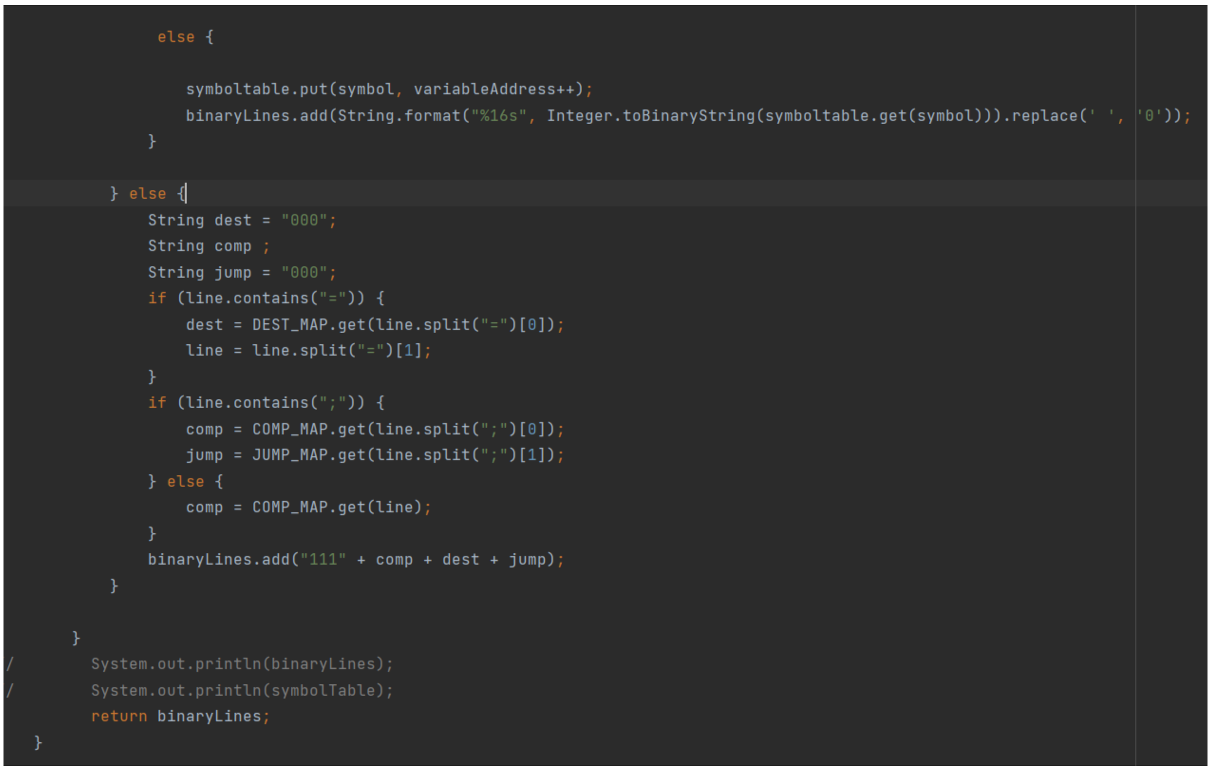
Now we create a new String named line where we access each line,

Nextly we initiate a loop with condition where there is no lines left for reading in the file. Now we give an if condition where if and only if the line has some value and if the line doesn’t start with “//” then the line gets added into the “lines” list we created. Here again if the line consists “//” then this line gets split till the part where the “//” is present and the other part is ignored.

Then we close the Bufferreader and return those “lines” list.



```java
private static List<String> binaryconversion(List<String> lines) {
    List<String> binaryLines = new ArrayList<String>();
    int address = 0;
    for (String line : lines) {

        if (line.startsWith("(") & line.endsWith(")")) {
            symboltable.put(line.substring(1, line.length() - 1), address);
        } else {
            address++;
        }

    }

    int variableAddress = 16;
    for (String line : lines) {
        if (line.startsWith("(")) {
            continue;
        }

        if (line.startsWith("@")) {
            String symbol = line.substring(1);
            if (symbol.matches("\\d+")) {
                binaryLines.add(String.format("%16s", Integer.toBinaryString(Integer.parseInt(symbol))).replace(' ', '0'));
            }
             else if (symboltable.containsKey(symbol)) {
                binaryLines.add(String.format("%16s", Integer.toBinaryString(symboltable.get(symbol))).replace(' ', '0'));
            }
```



```java
             else {

                symboltable.put(symbol, variableAddress++);
                binaryLines.add(String.format("%16s", Integer.toBinaryString(symboltable.get(symbol))).replace(' ', '0'));
            }

        } else {
            String dest = "000";
            String comp ;
            String jump = "000";
            if (line.contains("=")) {
                dest = DEST_MAP.get(line.split("=")[0]);
                line = line.split("=")[1];
            }
            if (line.contains(";")) {
                comp = COMP_MAP.get(line.split(";")[0]);
                jump = JUMP_MAP.get(line.split(";")[1]);
            } else {
                comp = COMP_MAP.get(line);
            }
            binaryLines.add("111" + comp + dest + jump);
        }

    }
/        System.out.println(binaryLines);
/        System.out.println(symbolTable);
        return binaryLines;
    }
```

Here from this code, we are heading to the main part of this assembler, which is identifying if the line is A- instruction or C- instruction.

Same as the last part we create a list for storing the converted binarylines. We initiate a for loop in which we identify if the line has a label. If the line has a label, then the label gets stored in the symboltable including the address. Or else the address count gets incremented.

Now we head to the main part, THE CONVERSION.

4.3 CONVERSION(SECOND PASS):

4.3.1 A - INSTRUCTION

We check if the line starts with “@”. If the line starts with “@” then the code considers it as an A-instruction and then operates on it. If the line has any decimal number then the number is directly converted to binary and the line is added to the “binaryLines” list

If the line has any symbol that matches in the symbol table then we take that corresponding value from that table and convert it into binary and then add it to the “binaryLines” list.

If the line doesn’t have any symbol that is referred in the symboltable and if the line doesn’t consist of any decimal number, then it is considered a variable. So, we add the variable with that variable address in the table we created. Then we immediately access that symbol’s value and convert it into binary. [1] [5]

After conversion the values with not be in 16-bit values, to make it 16-bit we add spaces and then replace those spaces with zeroes.

4.3.2 C – INSTRUCTION

Firstly the logic behind this is if the line consists of either “=” or “;” then the line is classified as C instruction, but here we already removed the unwanted lines and also we read the A – instructions so just we take every other line other than the line consists “@”.

Initially, we initiated the dest and jump as “000” since it can be null, but comp cannot be null.

We know that C instruction is basically “dest = comp;jmp”.

As said in the logic if the line consists of “=” then we split the line into 2 halves and take the 0th index and compare it with the table to get the corresponding value and store that value in dest since it is for the destination.

Then we take the 1st index of the split and then again we split it till “;” then we insert the 0th and 1st indexes since we know that “dest = comp;jmp” and compare it with the table to get the corresponding value and store that value into comp and jump respectively.

Then finally we just append the values as ("111" + comp + dest + jump) and then add it to the “binaryLines” list. Then we return the list. [1] [5] [4]

```
private static void writeBinaryFile(List<String> binaryLines, String
fileName) throws IOException {
    FileWriter writer = new FileWriter(fileName);
    for (String binaryLine : binaryLines) {
        writer.write(binaryLine + "\n");
    }
    writer.close();
}
```

This is a method which is used to write the binary file since all the lines are written in Bufferreader we read that lines and write it in the output file.

```
public static void main(String[] args) throws IOException {
    List<String> asmlines = preprocess("Pong.asm");
    List<String> asmbinarylines = binaryconversion(asmlines);

    writeBinaryFile(asmbinarylines, "Pong.hack");
    }
}
```

Finally, we create a main function and in that main function, we use the methods we created and then finally create an output file.

5. Observation & Results:

Let's take "Max.asm"

Input ASM file :

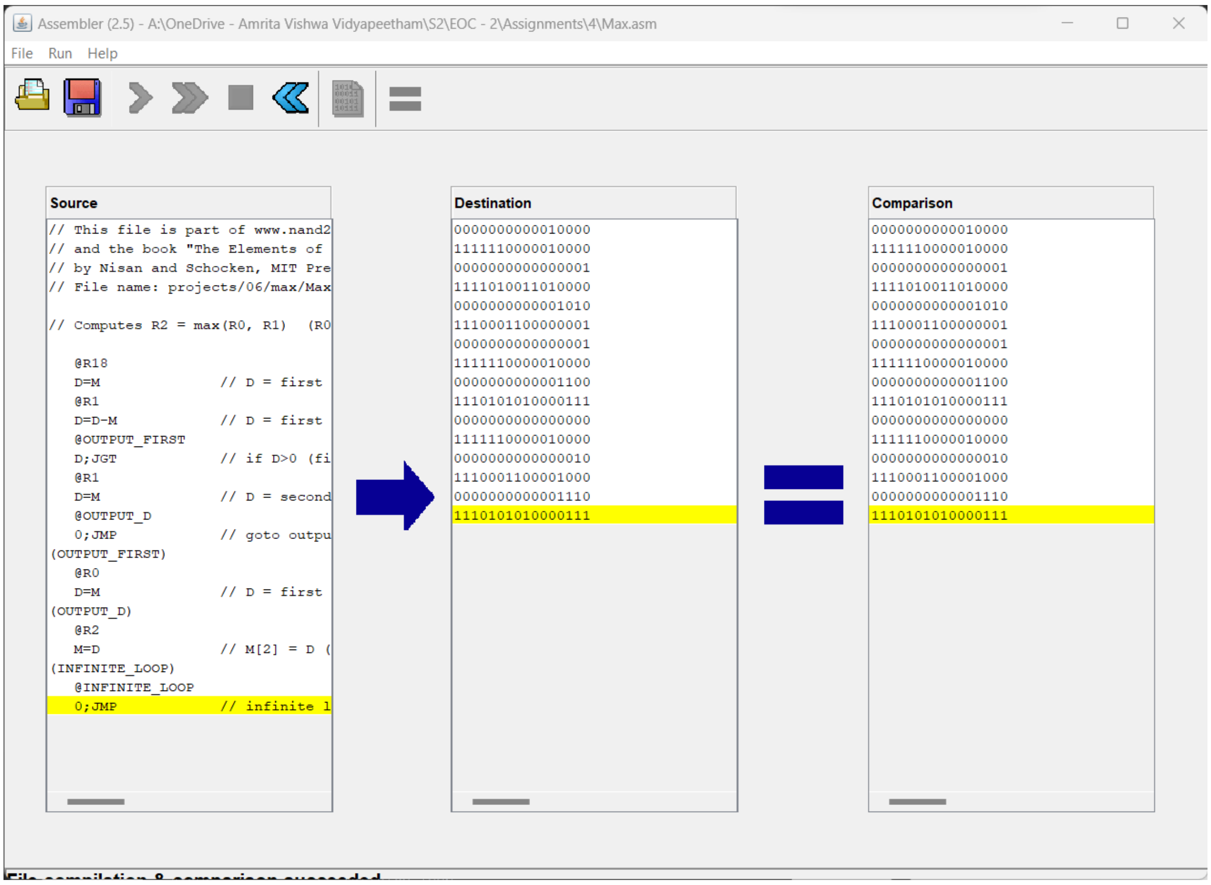
```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/06/max/Max.asm

// Computes R2 = max(R0, R1)  (R0,R1,R2 refer to RAM[0],RAM[1],RAM[2])

   @R18
   D=M              // D = first number
   @R1
   D=D-M            // D = first number - second number
   @OUTPUT_FIRST
   D;JGT            // if D>0 (first is greater) goto output_first
   @R1
   D=M              // D = second number
   @OUTPUT_D
   0;JMP            // goto output_d
(OUTPUT_FIRST)
   @R0
   D=M              // D = first number
(OUTPUT_D)
   @R2
   M=D              // M[2] = D (greatest number)
```

```
(INFINITE_LOOP)
   @INFINITE_LOOP
   0;JMP            // infinite loop
```

OUTPUT HACK FILE:

```
0000000000010000
1111110000010000
0000000000000001
1111010011010000
0000000000001010
1110001100000001
0000000000000001
1111110000010000
0000000000001100
1110101010000111
0000000000000000
1111110000010000
0000000000000010
1110001100001000
0000000000001110
1110101010000111
```

COMPARING:



Assembler (2.5) - A:\OneDrive - Amrita Vishwa Vidyapeetham\S2\EOC - 2\Assignments\4\Max.asm

Source

```
// This file is part of www.nand2
// and the book "The Elements of
// by Nisan and Schocken, MIT Pre
// File name: projects/06/max/Max

// Computes R2 = max(R0, R1)  (R0

   @R18
   D=M              // D = first
   @R1
   D=D-M            // D = first
   @OUTPUT_FIRST
   D;JGT            // if D>0 (fi
   @R1
   D=M              // D = second
   @OUTPUT_D
   0;JMP            // goto outpu
(OUTPUT_FIRST)
   @R0
   D=M              // D = first
(OUTPUT_D)
   @R2
   M=D              // M[2] = D (
(INFINITE_LOOP)
   @INFINITE_LOOP
   0;JMP            // infinite l
```

Destination

```
0000000000010000
1111110000010000
0000000000000001
1111010011010000
0000000000001010
1110001100000001
0000000000000001
1111110000010000
0000000000001100
1110101010000111
0000000000000000
1111110000010000
0000000000000010
1110001100001000
0000000000001110
1110101010000111
```

Comparison

```
0000000000010000
1111110000010000
0000000000000001
1111010011010000
0000000000001010
1110001100000001
0000000000000001
1111110000010000
0000000000001100
1110101010000111
0000000000000000
1111110000010000
0000000000000010
1110001100001000
0000000000001110
1110101010000111
```

File compilation & comparison succeeded

Here when we compare the two files it matches. Hence the assembler that we created is successfully running.

For the sake of example, we have taken the risk of running the largest code in the “nand2tetris” folder, which is “Pong.asm” with nearly 30,000 lines.

Click on this icon to access the Excel comparison file.

Click on this icon to access the Excel comparison file.

Successfully this code has done its job correctly by creating a hack file for the assembly language file.

6. Conclusion:

From this we can conclude that the assembler that we created is working perfectly and it can handle any assembly file and can convert it into a hack file. Through this project we learnt so many things in Java including creating HashMaps and assigning values to it, accessing the values from that HashMap and we also explored classes and the functions in that class, example: “Integer.tobinaryString”, in which converts an integer to binary. Also, we learnt how internally the assembler works and converts the file to binary file. [1]

7. References

[1] Assembler, "Geeks For Geeks," [Online]. Available: https://www.geeksforgeeks.org/introduction-of-assembler/.

[2] Modifiers, "Javatpoint," [Online]. Available: https://www.javatpoint.com/private-constructor-in-java#:~:text=Java%20allows%20us%20to%20declare%20a%20constructor%20as,use%20this%20private%20constructor%20in%20Singleton%20Design%20Pattern..

[3] Hashmaps, "Geeks for Geeks," [Online]. Available: https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/.

[4] F. Reading, "Geeks for geeks," [Online]. Available: https://www.geeksforgeeks.org/different-ways-reading-text-file-java/.

[5] BufferReader, "Javatpoint," [Online]. Available: https://www.javatpoint.com/java-bufferedreader-class.