Computer System Architecture_CSCI_6461_10

Project – 0

Write an assembler that converts each instruction into its 16-bit representation.

Team – 5

Jaiswal Nitish

Jadhav Yash

Jiang Xuechun

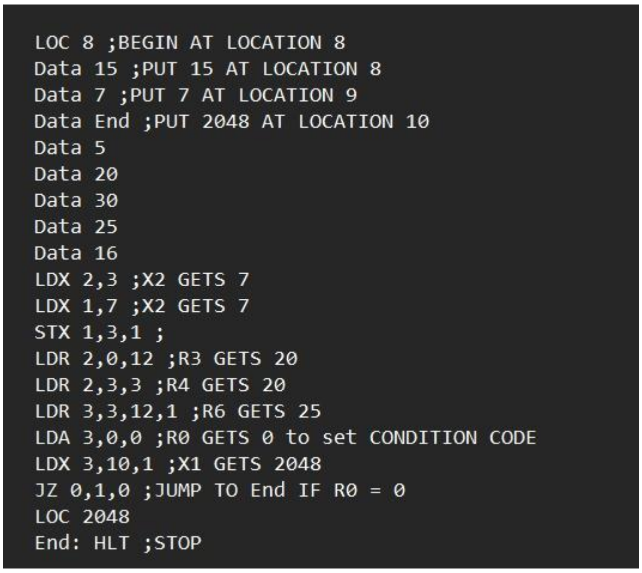
Problem Statement

The goal of this project is to create a 16-bit processor emulator that can carry out fundamental fixed-point arithmetic operations. The project's second section demonstrates a comprehension of the assembler's design, which would convert machine instructions into their 16-bit binary form and output them as octal values. In addition to addressing undefined instructions by producing a machine fault, the simulator will support the 64 distinct potential instructions specified by the ISA. Using the file system to load machine code into memory and a graphical user interface (GUI) to ask the user where the file is located, the CPU simulator will simulate a ROM loader. In some cases, this may necessitate the assembler translating machine-level instructions to their octal forms, which would then be printed down to a text file along with their memory addresses.

For example, Take the instruction `LDR 3,0,15`, for instance, which loads the data from memory address 15 into register 3. After being converted to its 16-bit counterpart, it will be expressed in octal. Deliverables will consist of a design document, test input files, one test JAR, source code, and simulator usage instructions. Throughout the project's implementation, Java is utilized.

ASSEMBLER SOURCE FILE

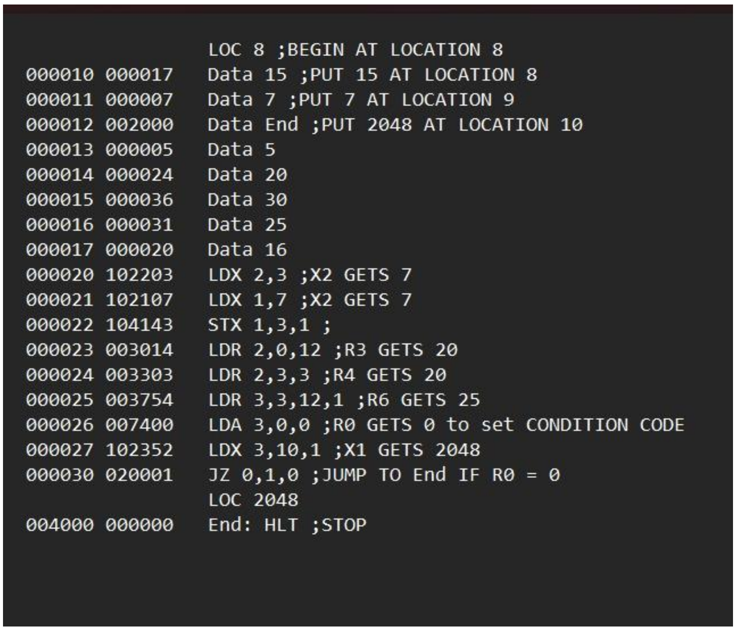
The following input operations are included in our simulator's source file, which should be saved in.asm format.



```
LOC 8 ;BEGIN AT LOCATION 8
Data 15 ;PUT 15 AT LOCATION 8
Data 7 ;PUT 7 AT LOCATION 9
Data End ;PUT 2048 AT LOCATION 10
Data 5
Data 20
Data 30
Data 25
Data 16
LDX 2,3 ;X2 GETS 7
LDX 1,7 ;X2 GETS 7
STX 1,3,1 ;
LDR 2,0,12 ;R3 GETS 20
LDR 2,3,3 ;R4 GETS 20
LDR 3,3,12,1 ;R6 GETS 25
LDA 3,0,0 ;R0 GETS 0 to set CONDITION CODE
LDX 3,10,1 ;X1 GETS 2048
JZ 0,1,0 ;JUMP TO End IF R0 = 0
LOC 2048
End: HLT ;STOP
```

LISTING-OUTPUT FILE

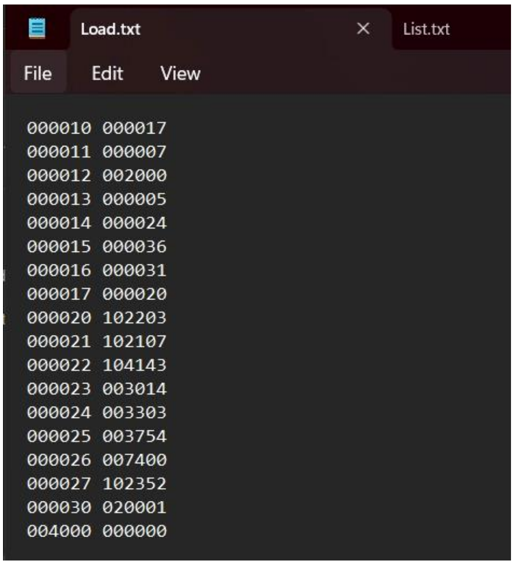
For the specified Assembler Source File, the Assembler Listing Output is a listing output file. Both of the assembler's columns are in OCTAL (decimal 10's 16-bit octal equivalent is 000012).



```
                 LOC 8 ;BEGIN AT LOCATION 8
000010 000017    Data 15 ;PUT 15 AT LOCATION 8
000011 000007    Data 7 ;PUT 7 AT LOCATION 9
000012 002000    Data End ;PUT 2048 AT LOCATION 10
000013 000005    Data 5
000014 000024    Data 20
000015 000036    Data 30
000016 000031    Data 25
000017 000020    Data 16
000020 102203    LDX 2,3 ;X2 GETS 7
000021 102107    LDX 1,7 ;X2 GETS 7
000022 104143    STX 1,3,1 ;
000023 003014    LDR 2,0,12 ;R3 GETS 20
000024 003303    LDR 2,3,3 ;R4 GETS 20
000025 003754    LDR 3,3,12,1 ;R6 GETS 25
000026 007400    LDA 3,0,0 ;R0 GETS 0 to set CONDITION CODE
000027 102352    LDX 3,10,1 ;X1 GETS 2048
000030 020001    JZ 0,1,0 ;JUMP TO End IF R0 = 0
                 LOC 2048
004000 000000    End: HLT ;STOP
```

LOADFILE

In the implementation of the load file, this file will be simulated as a text file rather than a binary file. Only non-blank lines should be loaded.

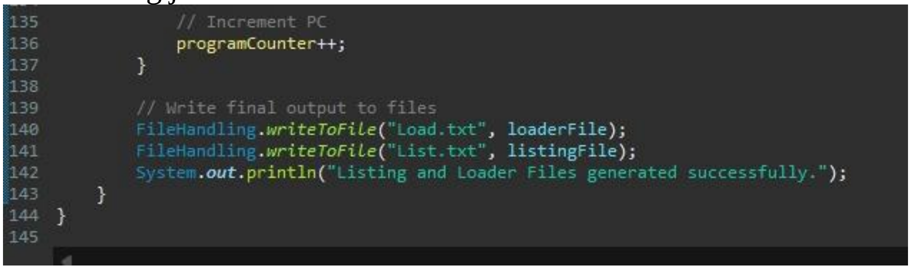


```
000010 000017
000011 000007
000012 002000
000013 000005
000014 000024
000015 000036
000016 000031
000017 000020
000020 102203
000021 102107
000022 104143
000023 003014
000024 003303
000025 003754
000026 007400
000027 102352
000030 020001
004000 000000
```

ASSEMBLER FILES

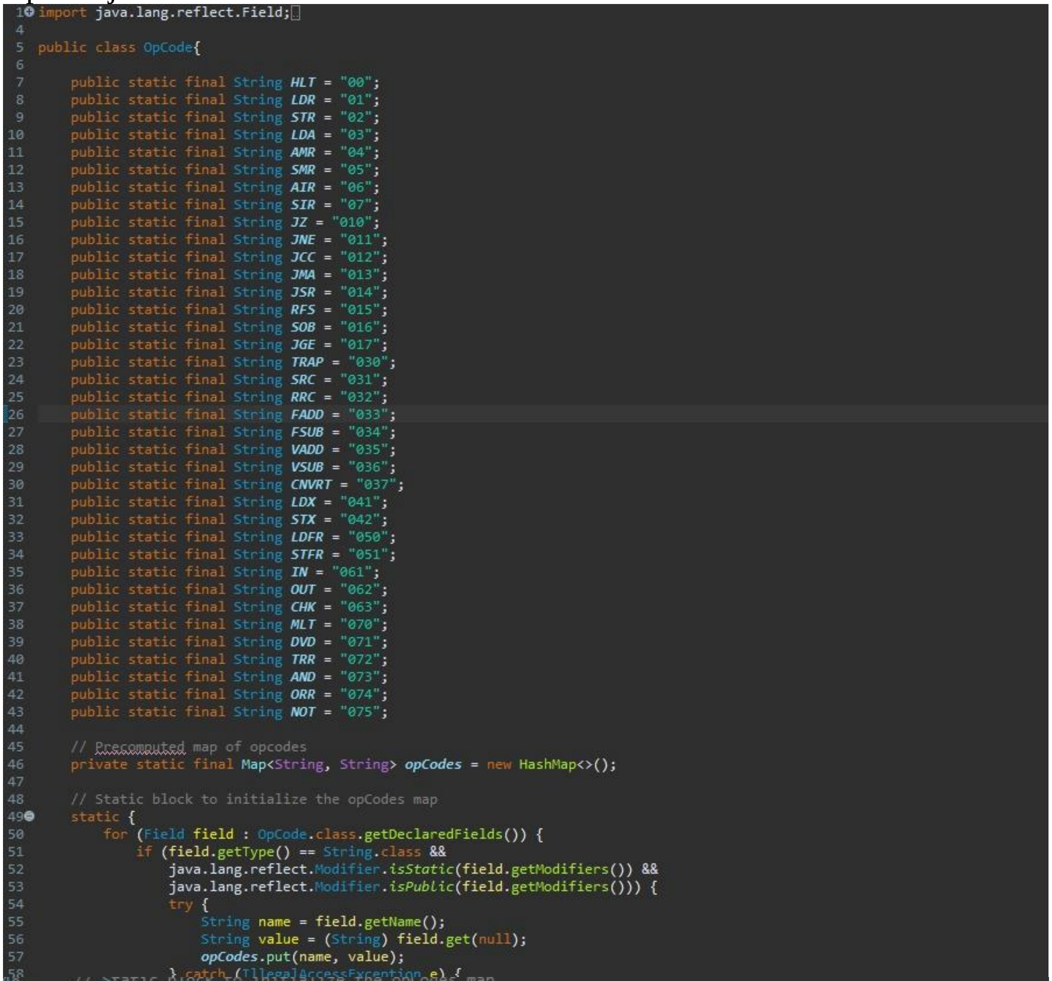
The Assembler code is written in JAVA. Below is the screenshot of the all code files—Opcode, Assembler, Format-string, and File-handling.

Filehandling.java

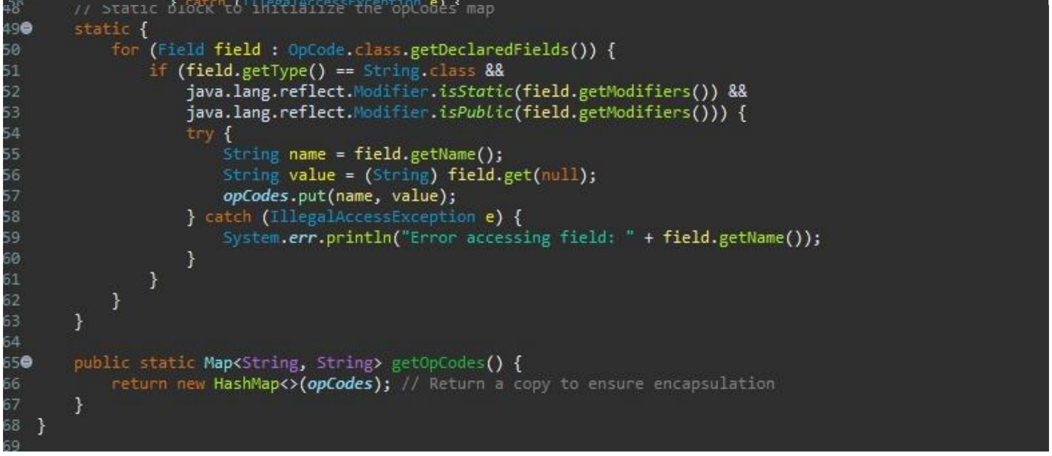


```
            // Increment PC
            programCounter++;
        }

        // Write final output to files
        FileHandling.writeToFile("Load.txt", loaderFile);
        FileHandling.writeToFile("List.txt", listingFile);
        System.out.println("Listing and Loader Files generated successfully.");
    }
}
```

Opcode.java

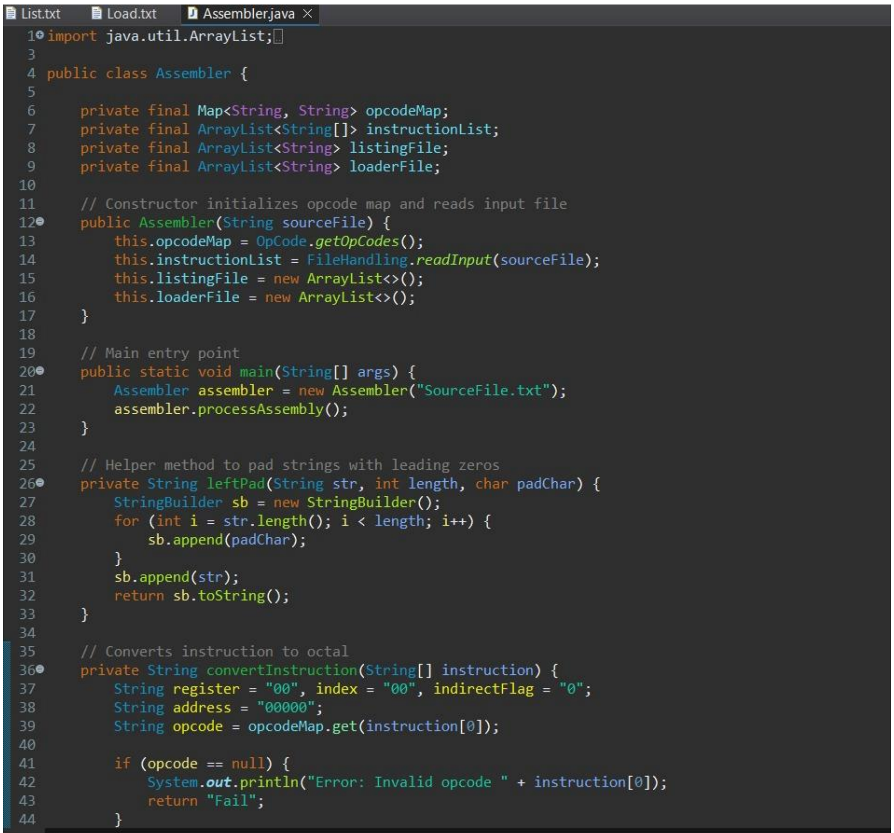


```
import java.lang.reflect.Field;

public class OpCode{

    public static final String HLT = "00";
    public static final String LDR = "01";
    public static final String STR = "02";
    public static final String LDA = "03";
    public static final String AMR = "04";
    public static final String SMR = "05";
    public static final String AIR = "06";
    public static final String SIR = "07";
    public static final String JZ = "010";
    public static final String JNE = "011";
    public static final String JCC = "012";
    public static final String JMA = "013";
    public static final String JSR = "014";
    public static final String RFS = "015";
    public static final String SOB = "016";
    public static final String JGE = "017";
    public static final String TRAP = "030";
    public static final String SRC = "031";
    public static final String RRC = "032";
    public static final String FADD = "033";
    public static final String FSUB = "034";
    public static final String VADD = "035";
    public static final String VSUB = "036";
    public static final String CNVRT = "037";
    public static final String LDX = "041";
    public static final String STX = "042";
    public static final String LDFR = "050";
    public static final String STFR = "051";
    public static final String IN = "061";
    public static final String OUT = "062";
    public static final String CHK = "063";
    public static final String MLT = "070";
    public static final String DVD = "071";
    public static final String TRR = "072";
    public static final String AND = "073";
    public static final String ORR = "074";
    public static final String NOT = "075";

    // Precomputed map of opcodes
    private static final Map<String, String> opCodes = new HashMap<>();

    // Static block to initialize the opCodes map
    static {
        for (Field field : OpCode.class.getDeclaredFields()) {
            if (field.getType() == String.class &&
                java.lang.reflect.Modifier.isStatic(field.getModifiers()) &&
                java.lang.reflect.Modifier.isPublic(field.getModifiers())) {
                try {
                    String name = field.getName();
                    String value = (String) field.get(null);
                    opCodes.put(name, value);
```

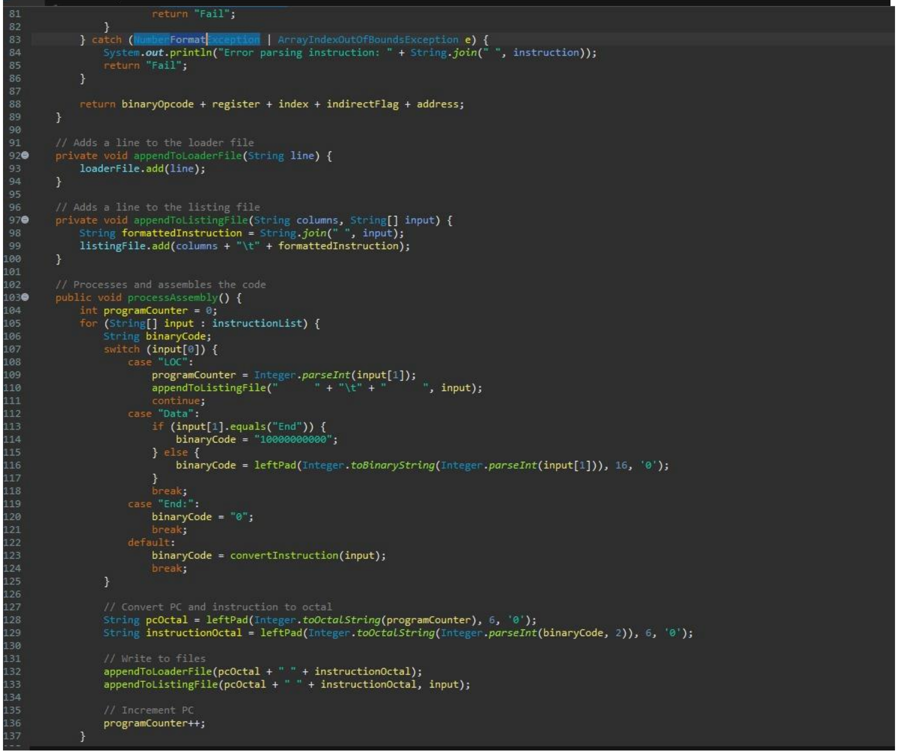


```
    // Static block to initialize the opCodes map
    static {
        for (Field field : OpCode.class.getDeclaredFields()) {
            if (field.getType() == String.class &&
                java.lang.reflect.Modifier.isStatic(field.getModifiers()) &&
                java.lang.reflect.Modifier.isPublic(field.getModifiers())) {
                try {
                    String name = field.getName();
                    String value = (String) field.get(null);
                    opCodes.put(name, value);
                } catch (IllegalAccessException e) {
                    System.err.println("Error accessing field: " + field.getName());
                }
            }
        }
    }

    public static Map<String, String> getOpCodes() {
        return new HashMap<>(opCodes); // Return a copy to ensure encapsulation
    }
}
```

Assembler.java



```java
import java.util.ArrayList;

public class Assembler {

    private final Map<String, String> opcodeMap;
    private final ArrayList<String[]> instructionList;
    private final ArrayList<String> listingFile;
    private final ArrayList<String> loaderFile;

    // Constructor initializes opcode map and reads input file
    public Assembler(String sourceFile) {
        this.opcodeMap = OpCode.getOpCodes();
        this.instructionList = FileHandling.readInput(sourceFile);
        this.listingFile = new ArrayList<>();
        this.loaderFile = new ArrayList<>();
    }

    // Main entry point
    public static void main(String[] args) {
        Assembler assembler = new Assembler("SourceFile.txt");
        assembler.processAssembly();
    }

    // Helper method to pad strings with leading zeros
    private String leftPad(String str, int length, char padChar) {
        StringBuilder sb = new StringBuilder();
        for (int i = str.length(); i < length; i++) {
            sb.append(padChar);
        }
        sb.append(str);
        return sb.toString();
    }

    // Converts instruction to octal
    private String convertInstruction(String[] instruction) {
        String register = "00", index = "00", indirectFlag = "0";
        String address = "00000";
        String opcode = opcodeMap.get(instruction[0]);

        if (opcode == null) {
            System.out.println("Error: Invalid opcode " + instruction[0]);
            return "Fail";
        }
```



```java
                return "Fail";
            }
        } catch (NumberFormatException | ArrayIndexOutOfBoundsException e) {
            System.out.println("Error parsing instruction: " + String.join(" ", instruction));
            return "Fail";
        }

        return binaryOpcode + register + index + indirectFlag + address;
    }

    // Adds a line to the loader file
    private void appendToLoaderFile(String line) {
        loaderFile.add(line);
    }

    // Adds a line to the listing file
    private void appendToListingFile(String columns, String[] input) {
        String formattedInstruction = String.join(" ", input);
        listingFile.add(columns + "\t" + formattedInstruction);
    }

    // Processes and assembles the code
    public void processAssembly() {
        int programCounter = 0;
        for (String[] input : instructionList) {
            String binaryCode;
            switch (input[0]) {
                case "LOC":
                    programCounter = Integer.parseInt(input[1]);
                    appendToListingFile("      " + "\t" + "      ", input);
                    continue;
                case "Data":
                    if (input[1].equals("End")) {
                        binaryCode = "10000000000";
                    } else {
                        binaryCode = leftPad(Integer.toBinaryString(Integer.parseInt(input[1])), 16, '0');
                    }
                    break;
                case "End:":
                    binaryCode = "0";
                    break;
                default:
                    binaryCode = convertInstruction(input);
                    break;
            }

            // Convert PC and instruction to octal
            String pcOctal = leftPad(Integer.toOctalString(programCounter), 6, '0');
            String instructionOctal = leftPad(Integer.toOctalString(Integer.parseInt(binaryCode, 2)), 6, '0');

            // Write to files
            appendToLoaderFile(pcOctal + " " + instructionOctal);
            appendToListingFile(pcOctal + " " + instructionOctal, input);

            // Increment PC
            programCounter++;
        }
```

CONCLUSION

To sum up, this project effectively illustrates how to use Java to create a CPU simulator and assembler for a 16-bit processor. Accurately translating machine-level instructions into their 16-bit binary counterparts, the assembler produces the results in octal format along with the memory addresses that correspond to them. The simulator can execute simple fixed-point and floating-point arithmetic instructions as specified by the Instruction Set Architecture (ISA) and load machine code from a file. Additionally, when undefined opcodes are encountered, it handles machine failures.

To demonstrate the development and execution process, screenshots of the source code and the completed file have been included. This documentation offers a thorough rundown of the project's capabilities, as do the source code, test files, and design paper that go with it.