Pass One Two Pass Assembler Documentation

Overview

The **PassOneTwoAssembler** is a Java Swing application that simulates a two-pass assembler for a simplified assembly language. It enables users to input assembly code and an operation table (OPTAB) to generate intermediate code, a symbol table, object code, and an object program. The application features a user-friendly graphical interface for ease of use.

Features

- Input assembly code and OPTAB through text areas.
- Generate intermediate code and symbol tables.
- Generate object code and object programs based on the provided assembly code and OPTAB.
- User-friendly GUI built with Java Swing.

Requirements

User Requirements

- Users should have basic knowledge of assembly language concepts.
- Users should be familiar with the format of assembly instructions and operation tables (OPTAB).
- Users need access to a computer with Java installed to run the application.

Developer Requirements

- **Java Development Kit (JDK)**: Version 8 or higher.
- **IDE or Text Editor**: An Integrated Development Environment (IDE) or text editor for Java development (e.g., IntelliJ IDEA, Eclipse, or NetBeans).
- **Libraries**: Ensure necessary Java libraries for Swing components are available.

Hardware Requirements

- **Processor**: Intel i3 or equivalent (minimum).
- **RAM**: 4 GB or more (8 GB recommended for smoother performance).
- **Storage**: At least 100 MB of free disk space for application and related files.
- **Display**: Minimum resolution of 1366 x 768 pixels.

Software Requirements

- **Operating System**:
    - Windows 7 or higher
    - macOS 10.12 (Sierra) or higher
    - Linux (most distributions that support Java)
- **Java Development Kit (JDK)**: Version 8 or higher, installed and configured properly.
- **IDE or Text Editor**:
    - IntelliJ IDEA
    - Eclipse
    - NetBeans
    - Any text editor that supports Java development.

Usage Instructions

1. **Input Assembly Code**:
    - Type or paste your assembly code into the "Input Assembly Code" text area. Each line should follow the format: <Label> <Opcode> <Operand>.
2. **Input OPTAB**:
    - Type or paste the operation table (OPTAB) in the "OPTAB" text area. Each line should contain the opcode followed by its machine code representation.
3. **Process Pass One**:
    - Click the "Process Pass One" button to analyze the assembly code. This will generate intermediate code and a symbol table, which will be displayed in their respective text areas.
4. **Process Pass Two**:
    - Click the "Process Pass Two" button to generate the object code and object program based on the intermediate code and symbol table. The results will be displayed in the respective text areas.

GUI Components

1. **JTextArea Components**:

- assemblyArea: Text area for inputting assembly code.
- optabArea: Text area for inputting the operation table (OPTAB).
- intermediateArea: Displays the generated intermediate code.
- symtabArea: Displays the generated symbol table.
- objectCodeArea: Displays the generated object code.
- objectprogramArea: Displays the generated object program.

2. **JButton Components**:
   - passOneButton: Button to process Pass One of the assembler.
   - passTwoButton: Button to process Pass Two of the assembler, initially disabled until Pass One is completed.
3. **JPanel Components**:
   - inputPanel: Contains the input text areas for assembly code and OPTAB.
   - outputPanel: Contains the output text areas for intermediate code, symbol table, object code, and object program.
   - buttonPanel: Contains the buttons for processing Pass One and Pass Two.

Methods

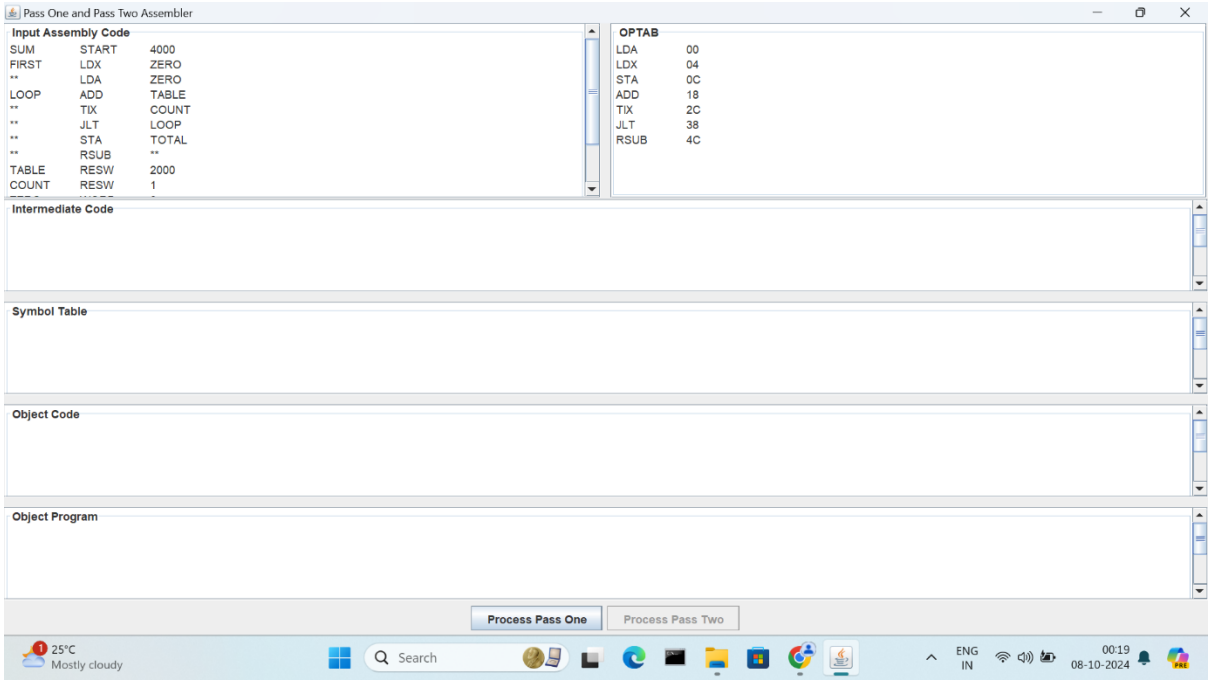
1. **public PassOneTwoAssembler()**
   - Constructor for initializing the GUI components and layout.
2. **public void actionPerformed(ActionEvent e)**
   - Handles button click events for processing Pass One and Pass Two.
3. **private void processPassOne()**
   - **Description**: Parses the assembly code and OPTAB to generate the intermediate code and symbol table.
   - **Details**: Initializes the location counter and processes the assembly code line by line. Handles START, END, and directives such as WORD, RESW, BYTE, and RESB. Updates the symbol table with labels and their corresponding addresses.
4. **private void processPassTwo()**

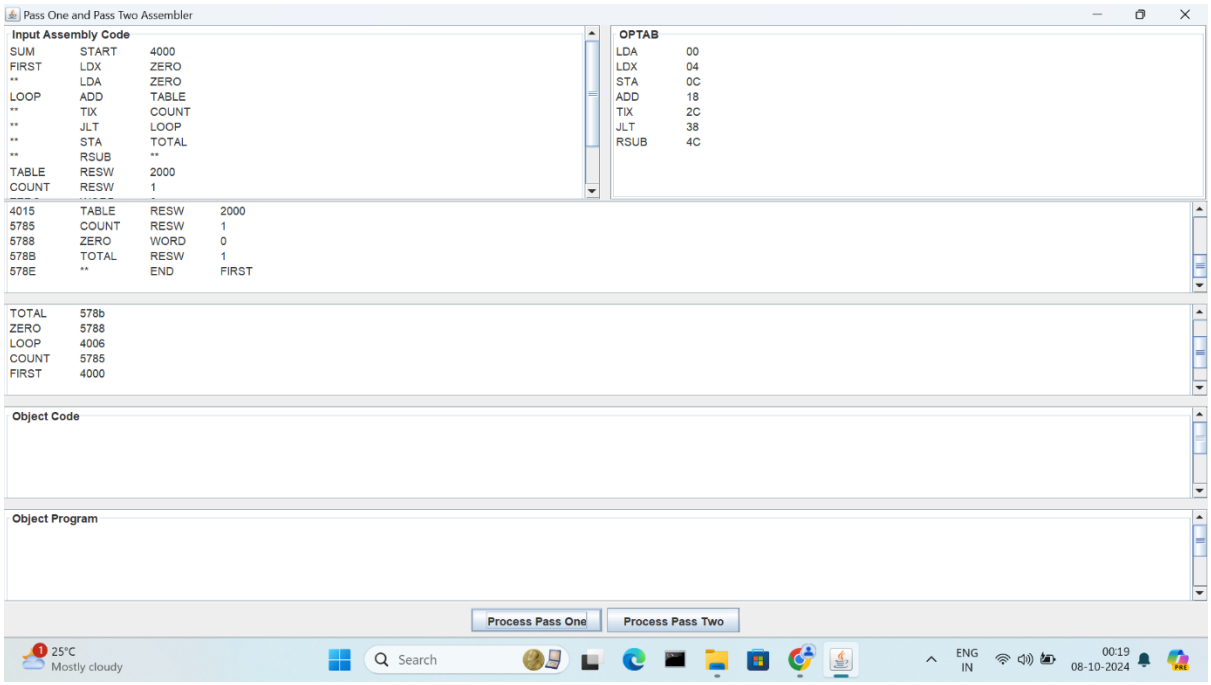
- **Description**: Processes the intermediate code to generate the object code and object program.
- **Details**: Creates header and text records. Handles the generation of object codes based on the operation table and symbol table. Ensures that text records do not exceed the maximum allowed length.

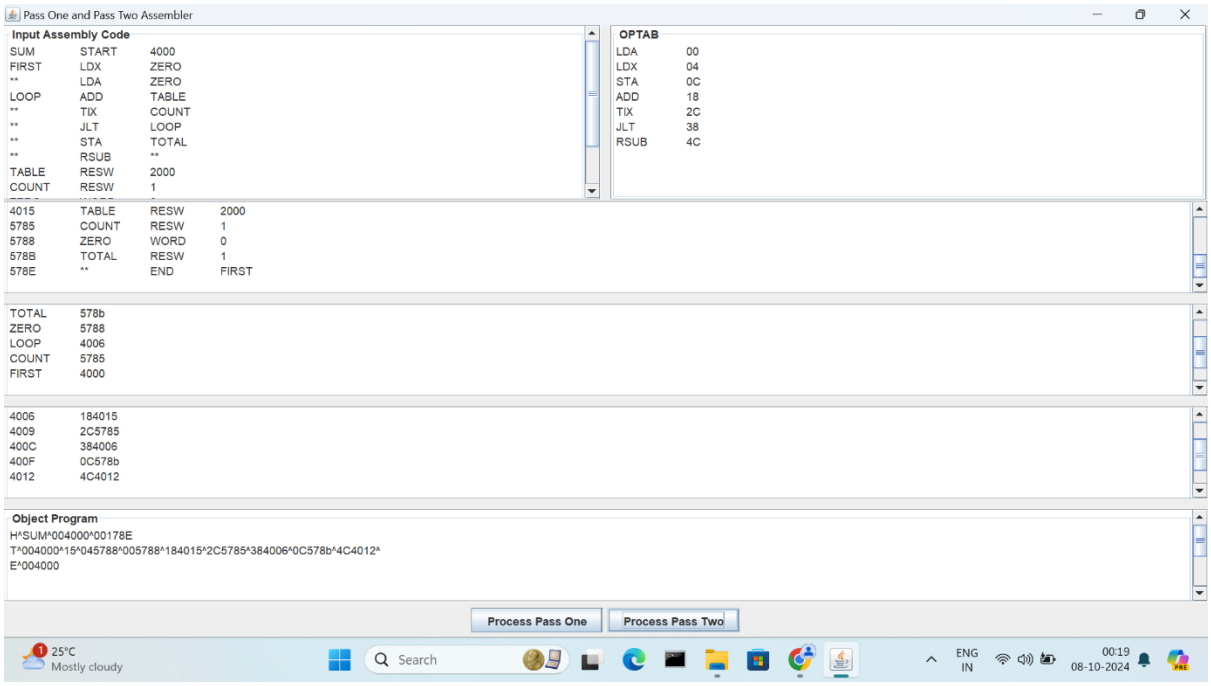
5. **private void parseOptab(String optabInput)**
   - **Description**: Parses the provided OPTAB input to populate the operation table.
   - **Details**: Reads each line, splitting the opcode and its corresponding machine code.
6. **public static void main(String[] args)**
   - **Description**: The entry point of the application. Initializes and displays the assembler GUI.

Output

- **Intermediate Code**: Displays location counter, labels, opcodes, and operands.
- **Symbol Table**: Displays labels with their addresses.
- **Object Code**: Displays generated object codes for the instructions.
- **Object Program**: Displays the formatted object program.


Pass One and Pass Two Assembler
Input Assembly Code
SUM START 4000
FIRST LDX ZERO
** LDA ZERO
LOOP ADD TABLE
** TIX COUNT
** JLT LOOP
** STA TOTAL
** RSUB **
TABLE RESW 2000
COUNT RESW 1
OPTAB
LDA 00
LDX 04
STA 0C
ADD 18
TIX 2C
JLT 38
RSUB 4C
Intermediate Code
Symbol Table
Object Code
Object Program
Process Pass One
Process Pass Two
25°C
Mostly cloudy
Search
ENG
IN
00:19
08-10-2024


Pass One and Pass Two Assembler
Input Assembly Code
SUM START 4000
FIRST LDX ZERO
** LDA ZERO
LOOP ADD TABLE
** TIX COUNT
** JLT LOOP
** STA TOTAL
** RSUB **
TABLE RESW 2000
COUNT RESW 1
OPTAB
LDA 00
LDX 04
STA 0C
ADD 18
TIX 2C
JLT 38
RSUB 4C
4015 TABLE RESW 2000
5785 COUNT RESW 1
5788 ZERO WORD 0
578B TOTAL RESW 1
578E ** END FIRST
TOTAL 578b
ZERO 5788
LOOP 4006
COUNT 5785
FIRST 4000
Object Code
Object Program
Process Pass One
Process Pass Two
25°C
Mostly cloudy
Search
ENG
IN
00:19
08-10-2024



github link:-

https://github.com/Alaina-cs/GUI-2-pass-assembler?tab=readme-ov-file

pass1 logic

```
private void processPassOne() {
   String assemblyCode = assemblyArea.getText().trim();
   String optabInput = optabArea.getText().trim();

   if (assemblyCode.isEmpty() || optabInput.isEmpty()) {
      JOptionPane.showMessageDialog(this, "Please provide both Assembly Code and OPTAB.");
      return;
   }

   parseOptab(optabInput); // Parse the OPTAB

   Scanner scanner = new Scanner(assemblyCode);
   StringBuilder intermediateBuilder = new StringBuilder();
```

```
    symtab.clear();
    intermediateCode = "";

    int locctr = 0, start = 0;
    boolean started = false;

    while (scanner.hasNextLine()) {
        String line = scanner.nextLine().trim();
        String[] tokens = line.split("\\s+");

        if (tokens.length == 0) continue;

        String label = tokens[0];
        String opcode = tokens.length > 1 ? tokens[1] : "";
        String operand = tokens.length > 2 ? tokens[2] : "";

        if (opcode.equals("START")) {
            programName = label;
            start = Integer.parseInt(operand, 16);
            locctr = start;
            programStartAddress = locctr;
            intermediateBuilder.append(String.format("%04X",
locctr)).append("\t").append(label).append("\t").append(opcode).append("\t").append(
operand).append("\n");
            started = true;
            continue;
        } else if (!started) {
            locctr = 0;
            started = true;
```

```
    }

    if (!opcode.equals("END")) {
        intermediateBuilder.append(String.format("%04X",
locctr)).append("\t").append(label).append("\t").append(opcode).append("\t").append(
operand).append("\n");

        if (!label.isEmpty()) {
            symtab.put(label, Integer.toHexString(locctr));
        }

        if (optab.containsKey(opcode)) {
            locctr += 3;
        } else {
            switch (opcode) {
                case "WORD": locctr += 3; break;
                case "RESW": locctr += 3 * Integer.parseInt(operand); break;
                case "BYTE": locctr += operand.length() - 3; break; // For handling constants
like BYTE C'EOF'
                case "RESB": locctr += Integer.parseInt(operand); break;
            }
        }
    } else {
        programLength = locctr - start;
        intermediateBuilder.append(String.format("%04X",
locctr)).append("\t").append(label).append("\t").append(opcode).append("\t").append(
operand).append("\n");
        break;
    }
```

```
    }

    intermediateCode = intermediateBuilder.toString();
    intermediateArea.setText(intermediateCode);
    updateSymbolTable(); // Helper method to update symbol table UI
    passTwoButton.setEnabled(true);
}

private void updateSymbolTable() {
    StringBuilder symtabBuilder = new StringBuilder();
    symtab.forEach((key, value) ->
symtabBuilder.append(key).append("\t").append(value).append("\n"));
    symtabArea.setText(symtabBuilder.toString());
}
```

Pass 2 logic

```
private void processPassTwo() {
    Scanner scanner = new Scanner(intermediateCode);
    StringBuilder programBuilder = new StringBuilder();
    StringBuilder textRecord = new StringBuilder();
    StringBuilder objectBuilder = new StringBuilder();
    int textStartAddress = 0;
    int textLength = 0;

    // Header record
    programBuilder.append("H").append("^")
```

```
        .append(programName).append("^")
        .append(String.format("%06X", programStartAddress)).append("^")
        .append(String.format("%06X", programLength)).append("\n");

while (scanner.hasNextLine()) {
    String line = scanner.nextLine().trim();
    String[] tokens = line.split("\\s+");
    if (tokens.length < 4) continue;

    String address = tokens[0];
    String opcode = tokens[2];
    String operand = tokens[3];

    if (optab.containsKey(opcode)) {
        String code = optab.get(opcode);
        String operandAddress = symtab.getOrDefault(operand, "0000");
        String objectCode = code + operandAddress;

        // New text record if current one is too long
        if (textLength + objectCode.length() > 60) {
            programBuilder.append("T").append("^")
                    .append(String.format("%06X", textStartAddress)).append("^")
                    .append(String.format("%02X", textLength / 2)).append("^")
                    .append(textRecord).append("\n");

            textRecord = new StringBuilder();
            textStartAddress = Integer.parseInt(address, 16);
            textLength = 0;
```

```
        }

        if (textLength == 0) {
            textStartAddress = Integer.parseInt(address, 16);
        }

        textRecord.append(objectCode).append("^");
        textLength += objectCode.length();
        objectBuilder.append(address).append("\t").append(objectCode).append("\n");
      }
    }

    // Write the last text record
    if (textLength > 0) {
      programBuilder.append("T").append("^")
            .append(String.format("%06X", textStartAddress)).append("^")
            .append(String.format("%02X", textLength / 2)).append("^")
            .append(textRecord).append("\n");
    }

    // End record
    programBuilder.append("E").append("^")
        .append(String.format("%06X", programStartAddress)).append("\n");

    objectCodeArea.setText(objectBuilder.toString());
    objectprogramArea.setText(programBuilder.toString());
}
```