Pass One Two Assembler Documentation

Overview

The program implements a simple graphical assembler simulator using Java's Swing framework. It performs the two passes typically involved in the assembly process of translating assembly language into machine code:

- **Pass 1**: It scans the assembly code, builds a symbol table, and calculates memory addresses.
- **Pass 2**: It generates the object code using the symbol table and opcode mappings.

The program has a graphical user interface (GUI) that allows the user to input assembly language source code and view the results of both passes.

Components

1. **Class `AssemblerGUI`**:
   - Extends `JFrame` to create a graphical window.
   - Contains all GUI components (text areas for input/output, buttons) and logic for processing the assembler passes.
2. **GUI Components**:
   - **Text Areas**:
     - `sourceCodeArea`: Input area where the user writes the assembly source code.
     - `pass1OutputArea`: Displays the intermediate file and symbol table after Pass 1.
     - `pass2OutputArea`: Displays the generated object code after Pass 2.
   - **Buttons**:
     - `pass1Button`: Runs Pass 1 when clicked.
     - `pass2Button`: Runs Pass 2 when clicked.
3. **Opcode and Symbol Tables**:
   - **`symbolTable`**: A `HashMap` that stores labels and their memory addresses.
   - **`opcodeTable`**: A `HashMap` that stores opcodes and their corresponding machine code.
4. **Location Counter (`locctr`)**:
   - Keeps track of the current memory address during the assembly process.
5. **Start Address**:
   - The starting address of the program is set by the `START` directive in the source code.

Key Methods

1. `runPass1()`

This method handles the first pass of the assembler, where:

- The source code is split into lines and processed line-by-line.
- The location counter (`locctr`) is updated based on the assembly instructions.

- Labels are added to the symbol table with their memory addresses.
- The intermediate file is output in the `pass1OutputArea`.
- The program length is calculated and displayed.

Steps:

- If the `START` directive is present, the starting address is initialized.
- Each instruction is processed, updating the `locctr` based on the opcode and operands.
- The symbol table is built with any label definitions.
- The intermediate file format is displayed with the memory addresses and instruction details.
- The program length (end address minus start address) is displayed.

2. `runPass2()`

This method handles the second pass, where:

- It reads the intermediate file (output of Pass 1).
- Object code is generated using the symbol table and opcode table.
- The object code is displayed in the `pass2OutputArea`.

Steps:

- For each instruction in the intermediate file, the opcode is converted into machine code.
- Labels (symbols) are resolved using the symbol table.
- For `WORD`, `BYTE`, and other instructions, specific handling is applied to generate object code.
- The output shows the memory address and corresponding object code.

3. `initializeOpcodeTable()`

- Initializes a `HashMap` that stores opcodes and their corresponding machine code values.
- Opcodes like `LDA`, `STA`, `ADD`, etc., are mapped to hexadecimal machine code strings.

4. `main()`

- The entry point of the program.
- It uses `SwingUtilities.invokeLater()` to create and display the `AssemblerGUI` window on the Event Dispatch Thread.

User Requirements

1. **Simple GUI for Entering and Assembling Code**:
    - Users can input assembly source code into the `sourceCodeArea`.
    - Users can run Pass 1 or Pass 2 by clicking the respective buttons.
    - Results are shown in separate text areas for clarity.
2. **Clear Output for Intermediate File and Object Code**:

    - The user sees the intermediate file (memory addresses, labels, opcodes, and operands) in Pass 1 output.
    - Pass 2 shows the object code generated based on the assembly instructions.
3. **Support for Basic Assembly Language Features**:
    - The assembler supports basic instructions like `LDA`, `STA`, `ADD`, `WORD`, `RESW`, `RESB`, `BYTE`, and `START/END`.
4. **Scroll Support for Large Code**:
    - Both input and output areas are scrollable, allowing the user to work with large programs.
5. **Error Handling**:
    - The application gracefully skips invalid or empty lines without crashing, providing stable feedback on valid instructions.

Developer Requirements

1. **Java Swing Knowledge**:
    - Developers should understand Java Swing components, layouts (`BorderLayout`, `GridLayout`), and event handling (`ActionListener`).
2. **Two-Pass Assembler Implementation**:
    - Pass 1 is responsible for building the symbol table and generating intermediate code.
    - Pass 2 uses the intermediate code and symbol table to generate machine-level object code.
3. **Extensible Opcode Table**:
    - The `opcodeTable` can be easily modified or extended by adding new opcodes and their machine code mappings in the `initializeOpcodeTable()` method.
4. **Symbol Table Management**:
    - The symbol table is a `HashMap<String, Integer>` that holds labels and their corresponding memory addresses.
5. **Handling Special Cases**:
    - Instructions like `BYTE` and `WORD` require special handling in object code generation.
    - The program length is calculated after Pass 1 by subtracting the start address from the final location counter value.

Software and Hardware Requirements

Software Requirements

1. **Operating System**:
    - Windows 7 or higher
    - macOS 10.10 or higher
    - Linux (any modern distribution that supports Java)
2. **Java Development Kit (JDK)**:
    - **Version**: JDK 8 or higher
    - The program uses Java Swing for the graphical interface, so a JDK installation is required for compiling and running the program.
3. **IDE** (Integrated Development Environment) (Optional but recommended):
    - **Eclipse**, **IntelliJ IDEA**, **NetBeans**, or any Java IDE can be used to write, compile, and debug the code.

    - These IDEs provide features like syntax highlighting, error checking, and integrated debugging tools.
4. **Java Runtime Environment (JRE)**:
    - **Version**: Java SE 8 or higher
    - Required for running the compiled `.jar` file if the program is distributed without source code.
5. **GUI Framework**:
    - The program uses Java Swing for building the graphical user interface, which is part of the Java Standard Edition (Java SE), so no external libraries are required.
6. **Text Editor (Optional)**:
    - A basic text editor (like Notepad++, Visual Studio Code, or Sublime Text) can be used to write and edit the assembly source code outside the GUI if needed.

Hardware Requirements

1. **Processor**:
    - Minimum: **Dual-core processor**
    - Recommended: **Quad-core processor** or higher for faster performance, especially for compiling large assembly programs.
2. **RAM**:
    - Minimum: **2 GB RAM**
    - Recommended: **4 GB RAM** or higher for smooth performance, especially if running other programs in parallel (like an IDE or browser).
3. **Storage**:
    - Minimum: **500 MB of free disk space**
    - Recommended: **1 GB of free disk space** for additional files, Java SDK, IDE installations, and assembly source code storage.
4. **Display**:
    - Minimum: **1024 x 768 resolution**
    - Recommended: **1366 x 768 resolution** or higher for better visibility of the GUI and source code editor.
5. **Input Devices**:
    - A **keyboard** for entering assembly code.
    - A **mouse** or touchpad for interacting with the GUI components like buttons, text areas, and scrollbars.
6. **Internet Access** (Optional but recommended):
    - Required for downloading the JDK, IDE, and additional development tools.
    - Internet is not required to run the program itself but may be useful for troubleshooting, downloading dependencies, or updating Java.

Pass 1 and pass 2 logic explanation:

The logic portion of the **AssemblerGUI** code is where the actual assembly-like processing is simulated. This part focuses on two main operations: **Pass 1** and **Pass 2**. Each pass performs different tasks related to the assembly process, mimicking how a real assembler works to translate assembly code into machine code.

1. Pass 1 Logic (`runPass1()`)

The **Pass 1** function simulates the first step of the assembly process, which involves:

- Generating the **symbol table** (which holds the memory addresses for labels in the source code).
- Calculating the location counter (`locctr`).
- Formatting intermediate code.

```
private void runPass1() {
    symbolTable.clear(); // Clear symbol table from any previous run.
    locctr = 0;          // Initialize the location counter to 0.
    pass1OutputArea.setText("");  // Clear any previous output in the Pass
1 text area.

    String[] lines = sourceCodeArea.getText().split("\n"); // Split input
source code into lines.

    // Check the first line for a START directive
    String[] firstLine = lines[0].trim().split("\\s+");
    if (firstLine.length >= 3 && firstLine[1].equals("START")) {
        start = Integer.parseInt(firstLine[2]);  // Set starting address.
        locctr = start;                          // Initialize location
counter to start.
        pass1OutputArea.append("\t" + firstLine[0] + "\t" + firstLine[1] +
"\t" + firstLine[2] + "\n");
    } else {
        locctr = 0;  // If no START directive, start at location 0.
    }

    // Process remaining lines
    for (int i = 1; i < lines.length; i++) {
        String line = lines[i].trim();
        if (line.isEmpty()) continue;  // Skip empty lines.

        String[] parts = line.split("\\s+");
        String label = parts.length > 2 ? parts[0] : "**";  // If a label
exists, store it, otherwise "**".
        String opcode = parts.length > 2 ? parts[1] : parts[0]; // Get the
opcode.
        String operand = parts.length > 2 ? parts[2] : parts[1]; // Get the
operand.

        // Output intermediate code with location counter, label, opcode,
and operand.
        pass1OutputArea.append(locctr + "\t" + label + "\t" + opcode + "\t"
+ operand + "\n");

        // Add label to the symbol table if it exists.
        if (!label.equals("**")) {
            symbolTable.put(label, locctr);
        }

        // Update location counter based on opcode type
        if (opcode.equals("WORD")) {
            locctr += 3; // Each WORD instruction is 3 bytes.
        } else if (opcode.equals("RESW")) {
            locctr += (3 * Integer.parseInt(operand)); // RESW reserves
words (3 bytes each).
        } else if (opcode.equals("RESB")) {
            locctr += Integer.parseInt(operand); // RESB reserves bytes.
        } else if (opcode.equals("BYTE")) {
```

```
            locctr += 1; // BYTE reserves 1 byte.
        } else {
            locctr += 3; // Default size for an instruction is 3 bytes.
        }

        // Stop processing if END directive is encountered.
        if (opcode.equals("END")) {
            break;
        }
    }

    // Display symbol table
    pass1OutputArea.append("\nSymbol Table:\n");
    for (Map.Entry<String, Integer> entry : symbolTable.entrySet()) {
        pass1OutputArea.append(entry.getKey() + "\t" + entry.getValue() +
"\n");
    }

    // Calculate the length of the program.
    length = locctr - start;
    pass1OutputArea.append("\nLength of the program: " + length + "\n");
}
```

Logic Explanation:

- **Symbol Table Creation**: The function checks each line of the source code for labels (symbols), and if they exist, they are stored in a `symbolTable` with their associated memory address (`locctr`).
- **Location Counter Calculation**: Based on the type of opcode (e.g., `WORD`, `BYTE`, `RESW`), the location counter is incremented accordingly to reflect how much memory is being reserved.
- **Intermediate Code Generation**: Each line of the source code is processed, and an intermediate representation (line with location counter, label, opcode, and operand) is generated and displayed in the `pass1OutputArea`.
- **Program Length**: After processing all lines, the length of the program is calculated.

---

2. Pass 2 Logic (`runPass2()`)

The **Pass 2** function simulates the second step of the assembly process, where:

- The program reads the intermediate code generated by Pass 1.
- The object code is generated using the opcode and the symbol table.

```
private void runPass2() {
    pass2OutputArea.setText("");  // Clear previous Pass 2 output.

    String[] lines = pass1OutputArea.getText().split("\n");  // Read the
intermediate code generated by Pass 1.

    for (String line : lines) {
        if (line.trim().isEmpty() || line.startsWith("Symbol") ||
line.startsWith("Length")) {
            continue;  // Skip empty lines and non-code sections.
        }
```

```
        String[] parts = line.trim().split("\\s+");
        if (parts.length < 4) continue;

        String address = parts[0];  // Memory address from Pass 1.
        String label = parts[1];    // Label from Pass 1.
        String opcode = parts[2];   // Opcode from Pass 1.
        String operand = parts[3];  // Operand from Pass 1.

        // Generate object code if the opcode is in the opcode table.
        if (opcodeTable.containsKey(opcode)) {
            String objectCode = opcodeTable.get(opcode);

            // If the operand is in the symbol table, append the address.
            if (symbolTable.containsKey(operand)) {
                objectCode += String.format("%03d",
symbolTable.get(operand));
            } else if (opcode.equals("BYTE")) {
                // For BYTE operations, convert characters to object code.
                if (operand.startsWith("C'")) {
                    objectCode = operand.substring(2, operand.length() -
1);  // Extract characters from C'X'.
                }
            } else if (opcode.equals("WORD")) {
                // For WORD operations, convert to hexadecimal format.
                objectCode = String.format("%06X",
Integer.parseInt(operand));
            }

            // Output the object code and address to Pass 2 output area.
            pass2OutputArea.append(address + "\t" + objectCode + "\n");
        }
    }
}
```

Logic Explanation:

- **Object Code Generation**: For each line of intermediate code (from Pass 1), the function checks the opcode and generates corresponding object code using the `opcodeTable`. The symbol table is used to resolve operand addresses.
    - **Opcode Handling**: The opcode is mapped to a machine instruction using the `opcodeTable`.
    - **Operand Handling**: If the operand is a symbol (i.e., a label), its memory address is fetched from the `symbolTable`. If the operand is a literal (like `BYTE` or `WORD`), it is directly converted to an object code.
- **Output**: The generated object code, along with the address, is printed to the `pass2OutputArea`.

Conclusion

This program provides a simple assembler simulation with a user-friendly GUI. It allows the user to input assembly code, generate symbol tables, intermediate files, and object code in two passes. It is modular, easy to extend, and handles basic assembly instructions.

LINK: https://github.com/aksa-ann2/2-pass-assembler/blob/main/README.md

OUTPUT