

# **Two-Pass Assembler**

## **GUI Documentation**

# Introduction

The **Two-Pass Assembler GUI** is a graphical user interface (GUI) application built in Java using Swing. This program simulates the two-pass assembly process, where input and opcode files are processed to generate object code, a symbol table, and intermediate results. The GUI allows users to load the input assembly code and the opcode table, run the assembler, and view the results in an easy-to-read format.

---

## Code Walkthrough

### 1. Main Components

- **TwoPassAssemblerGUI**: The main class responsible for setting up the GUI and running the assembler.
- **RoundedButton**: A custom JButton subclass used to create rounded buttons for the GUI, enhancing the aesthetics.

### 2. UI Components

- **JFrame**: The main window for the application.
- **TextArea**: These text areas are used to display various parts of the assembly process, including the input file, opcode table, intermediate file, symbol table, program length, output, and object code.
- **JFileChooser**: A dialog to choose input and opcode files.

### 3. Custom Button Class: RoundedButton

- This class extends JButton to create buttons with rounded corners. The `paintComponent()` method ensures the rounded shape, while `getPreferredSize()` determines its size.

### 4. GUI Setup

- The GUI's main frame is created here, with a default close operation and size. The background color is set using a hexadecimal color code.

## 5. Text Areas and Panels

- `createTextArea()` initializes multiple non-editable text areas, used to display different outputs of the assembly process.
- Two labeled text areas are placed side-by-side to show the input and opcode table files. The `GridLayout(1, 2)` divides the panel into two equal parts.

## 6. File Loading

- The "Load Input File" and "Load Optab File" buttons are assigned `ActionListener` events. These trigger the `loadFile()` method, which opens a file chooser and loads the selected file into the corresponding text area.

## 7. Two-Pass Assembler Functions

### Pass 1

- **Pass 1** reads the input file, separates each line into label, opcode, and operand, and calculates the location counter for each instruction. It builds the intermediate file content, updates the symbol table, and calculates the program length.

### Pass 2

- **Pass 2** reads the intermediate file, translates instructions to object code, and builds the object program, complete with header, text records, and an end record.
-

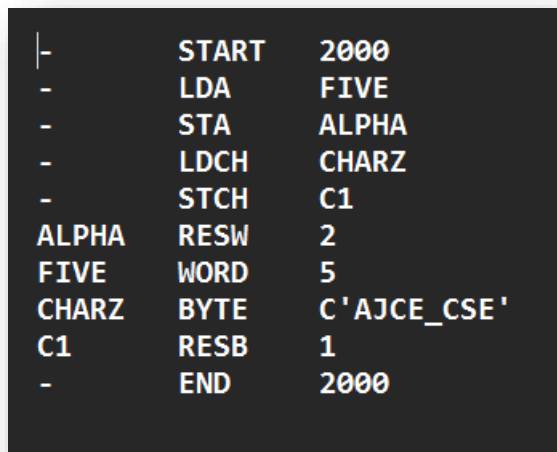
# User Manual

## 1. Starting the Application

- To start the Two-Pass Assembler, run the main() method of TwoPassAssemblerGUI.java. This will open the graphical interface with areas for input, opcode table, intermediate results, symbol table, program length, output, and object code.

## 2. Loading Files

- **Load Input File:** Click the "Load Input File" button to open a file chooser. (e.g: input.txt)
- .Select the assembly program file you want to assemble.
- The contents of the file will be displayed in the "Input File" area.



```
|-      START      2000
-      LDA        FIVE
-      STA        ALPHA
-      LDCH       CHARZ
-      STCH       C1
ALPHA   RESW       2
FIVE    WORD       5
CHARZ   BYTE      C'AJCE_CSE'
C1      RESB       1
-      END        2000
```

- **Load Optab File:** Click the "Load Optab File" button to open a file chooser. (e.g: optab.txt).
- Select the file containing the opcode table (opcode and corresponding machine code).
- The contents will be displayed in the "Optab File" area.

LDA	00
STA	0c
LDCH	50
STCH	54

### 3. Running the Two-Pass Assembler

- After loading both the input file and the opcode table, click the "Run Two-Pass Assembler" button. The assembler will execute, performing both passes.
- **Pass 1:** This will generate an intermediate file, symbol table, and compute the program length. The intermediate file will be displayed in the "Intermediate" area, and the symbol table in the "Symbol Table" area.
- **Pass 2:** This will generate the object code and output the final object program, including the header, text records, and end record.

### 4. Viewing Results

- **Intermediate Results:** The output from Pass 1, including location counters and original instructions.
- **Symbol Table:** The list of symbols (labels) encountered during Pass 1, along with their memory addresses.
- **Program Length:** The total length of the program.
- **Output:** The detailed output from Pass 2, including addresses, opcodes, and generated object code.
- **Object Code:** The final object code, ready for execution by a machine.

### 5. Error Handling

- If any errors occur, such as missing files or invalid input, the program will show an appropriate error message in a dialog box.
- 

## **Features**

- **Custom Buttons:** The buttons are aesthetically designed with rounded corners for a modern look.
  - **Comprehensive Output:** The assembler provides detailed outputs at every stage, making it easy to follow the assembly process.
  - **File Loading:** Users can easily load input and opcode files using file dialogs.
  - **Error Messages:** If any issues arise, the user is notified through message dialogs.
-

# User Interface

Two-Pass Assembler

Input File:

Optab File:

Pass 1

Intermediate:

Symbol Table:

Pass 2

Output:

Program Length:

Object Code:

Load Input File

Load Optab File

Run Two-Pass Assembler

# Input Interface

Two-Pass Assembler

Input File:

-

START

2000

-

LDA

FIVE

-

STA

ALPHA

-

LDCH

CHARZ

-

STCH

C1

ALPHA

RESW

2

FIVE

WORD

5

CHARZ

BYTE

C'AJCE\_CSE'

C1

RESB

1

-

END

2000

Optab File:

LDA 00

STA 0c

LDCH 50

STCH 54

Pass 1

Intermediate:

Symbol Table:

Pass 2

Output:

Program Length:

Object Code:

Load Input File

Load Optab File

Run Two-Pass Assembler



# Ouput Interface

Two-Pass Assembler

Input File:

```
- START 2000
- LDA FIVE
- STA ALPHA
- LDCH CHARZ
- STCH C1
ALPHA RESW 2
FIVE WORD 5
CHARZ BYTE C'AJCE_CSE'
C1 RESB 1
- END 2000
```

Optab File:

```
LDA 00
STA 0c
LDCH 50
STCH 54
```

Pass 1

Intermediate:

```
- START 2000
2000 - LDA FIVE
2003 - STA ALPHA
2006 - LDCH CHARZ
2009 - STCH C1
200C ALPHA RESW 2
2012 FIVE WORD 5
2015 CHARZ BYTE C'AJCE_CSE'
201D C1 RESB 1
201E - END 2000
```

Symbol Table:

```
FIVE 2012
CHARZ 2015
ALPHA 200C
C1 201D
```

Pass 2

Output:

```
- START 2000
2000 - LDA FIVE 002012
2003 - STA ALPHA 0c200C
2006 - LDCH CHARZ 502015
2009 - STCH C1 54201D
200C ALPHA RESW 2
2012 FIVE WORD 5 000005
2015 CHARZ BYTE C'AJCE_CSE' 414A43455F435345
201D C1 RESB 1
201E - END 2000
```

Object Code:

```
H^ ^002000^00001E
T^002000^17^002012^0c200C^502015^54201D^000005^414A43455F435345
E^2000
```

Program Length:

Program Length: 1E

Load Input File

Load Optab File

Run Two-Pass Assembler

## User's Perspective

As a user, the **Two-Pass Assembler GUI** is designed to offer an intuitive and smooth experience, even if you're not deeply familiar with assemblers. The graphical interface ensures that every step of the assembler process is straightforward, from loading input files to reviewing results. Here's what it feels like from a user's perspective:

1. **Loading Input Files:** The interface allows you to easily load your assembly code and opcode table files with the click of a button. There's no need to worry about file format specifics; just select the correct files, and the system handles the rest. You receive confirmation messages when the files are successfully loaded, making the process feel seamless.
2. **Understanding Passes:** The GUI is split into two clear sections, labeled "Pass 1" and "Pass 2," which helps you visualize the assembler's operations step-by-step. In Pass 1, you'll see how the intermediate file and symbol table are generated, and in Pass 2, you'll view the final object code and output. This layout gives you insight into how the assembler processes instructions, making it easy to follow along.
3. **Interactive Feedback:** The visual output, including the symbol table, intermediate code, and object code, is well-organized and readable. It's color-coded with clearly labeled sections, so you don't have to hunt for information. Each file loaded and action performed is followed by feedback, ensuring you always know what the system is doing.
4. **Error Handling:** The system ensures that you load both an input file and an opcode table before running the assembler, preventing mistakes and confusion. If something goes wrong, such as a file not being loaded, the GUI immediately informs you, so you can quickly fix the issue.

5. **Readable Output:** After the two passes, you get to see a detailed breakdown of the output. Whether you're looking for the object code or analyzing the symbol table, all the results are displayed in a clear and readable manner. The text boxes use a simple format to ensure you can review and analyze the assembler's work without distractions.
- 

## Programmer's Perspective

From a programmer's perspective, the **Two-Pass Assembler GUI** is a well-structured, Java-based application that combines the two-pass assembler logic with a user-friendly graphical interface built using Java Swing. Here's a breakdown of its key features and considerations from the developer's point of view:

1. **Modular Design:** The project follows a modular design that separates concerns between the assembler logic and the GUI. The assembler logic itself, comprising Pass 1 and Pass 2, is cleanly separated from the user interface, making the codebase more maintainable and easier to extend. Each component, such as loading files, reading input, handling the opcode table, and displaying output, is organized into logical methods.
2. **Custom GUI Components:** The GUI is built using Java Swing, with some customization to make the interface more visually appealing. For instance, the `RoundedButton` class is a custom implementation of a button with rounded corners and a specific color scheme, which enhances the look and feel of the interface. This shows thoughtful UI/UX design considerations while working within Swing's framework.

3. **File Handling:** The assembler reads both the input file and the opcode table file using standard Java file handling (via `BufferedReader`), which is reliable and efficient. The use of a `JFileChooser` makes file loading intuitive for users, while the file handling logic is abstracted into utility methods (`loadTextFile`, `loadFile`), simplifying code reuse.
4. **Two-Pass Assembler Logic:** The core of the assembler is implemented in two stages:
  - **Pass 1:** Builds the symbol table and generates intermediate code. It handles instructions like `WORD`, `RESW`, `RESB`, and `BYTE`, along with the usual opcodes. The location counter logic is crucial here, ensuring accurate tracking of memory addresses.
  - **Pass 2:** Reads the intermediate file to generate object code, which is then formatted in both text and hexadecimal format. It handles special cases like `WORD`, `BYTE`, and `START`, and ensures that the opcode and operand values are correctly translated into machine-readable format.

This split between passes is critical for maintaining clarity in the assembler's workflow, ensuring that Pass 1 handles address assignment and symbol resolution, while Pass 2 focuses on code generation.

5. **Error Handling:** The assembler includes basic error handling, such as ensuring both input and opcode files are loaded before attempting to run the assembler. Additionally, it checks for duplicate labels in Pass 1 and displays meaningful messages if errors occur. However, from a programmer's perspective, this aspect could be expanded with more sophisticated error handling (e.g., catching syntax errors or invalid opcodes).
6. **Use of Data Structures:**

- **Symbol Table:** A `HashMap<String, Integer>` is used to store labels and their corresponding memory addresses. This provides constant-time lookup for symbols in both passes.
- **Opcode Table:** Another `HashMap<String, String>` is used for storing opcodes and their corresponding machine code equivalents. This makes opcode lookup fast and efficient during both passes.

These data structures are well-suited to the task and ensure the assembler performs efficiently, even for larger programs.

7. **Performance Considerations:** The two-pass assembler implemented here is quite efficient for typical educational use cases, thanks to the efficient lookup times provided by hash maps and the simplicity of reading and processing the input line-by-line. For larger-scale assembly programs, performance optimizations could be considered, such as streamlining the file reading or optimizing the way object code is generated.
8. **Scalability and Extensibility:** The design of the assembler is scalable for basic SIC assembler programs but could be easily extended to handle more complex assembler tasks or different instruction sets. For example:
  - Adding support for more directives or additional addressing modes.
  - Enhancing the symbol table or opcode table to handle different machine architectures.
  - Extending error-checking capabilities, such as providing detailed warnings or suggestions to the user when errors are encountered during assembly.
9. **UI/UX Considerations:** The GUI is thoughtfully designed with clear sectioning into Pass 1 and Pass 2, allowing the user to see the assembler's work in stages. Text areas are used to display

intermediate and final outputs, symbol tables, and object code, providing immediate feedback. For a programmer, the UI code is straightforward and uses standard Swing components, allowing for quick updates or modifications if needed.

10.      **Readability and Maintenance:** The code is well-documented and easy to follow, with clear method names and logical separation of concerns. The user interface elements are grouped into panels with intuitive names, making it easy to understand how the GUI is laid out. Moreover, the methods for handling Pass 1, Pass 2, and file loading are clearly structured, making the codebase easy to maintain and extend.
-

# Conclusion

The **Two-Pass Assembler GUI** provides a user-friendly environment for assembling SIC assembly programs. The interface makes it easy to load assembly files and opcode tables, view intermediate and final outputs, and visualize the process of both passes in the assembler. The tool ensures clarity through its structured display of information such as symbol tables, object codes, and program length.

The use of Java Swing for the graphical interface enables cross-platform compatibility and flexibility. Features like the custom rounded buttons and organized panels improve usability and aesthetics. Furthermore, error handling mechanisms are in place to assist users in ensuring files are correctly loaded and that the assembler process runs smoothly.

This tool bridges the gap between the underlying complexity of a two-pass assembler and its practical use, making it highly useful for students, educators, and developers learning about assemblers or working on related projects.