CSCI 6461 Computer Architecture - Project Part 0

Design Notes and User Guide for C6461 Assembler

Submitted By: Sribalaji Annamalai Senthilkumar, Abenezer Golda, Razia Noor, Christabell Rego

1. Overview

Part 0 of this project focuses on the foundational first step in building a computer simulator: creating an **assembler**. The primary objective is to develop a Java program that translates assembly language source code, written for the custom C6461 Instruction Set Architecture (ISA), into machine-readable octal code.

George Washington University – Dept. of Computer Science
CS6461: Computer Architectures

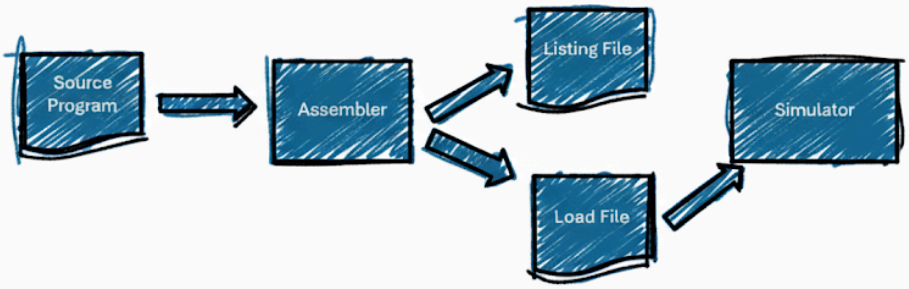

Figure – Overall assemble data flow

This assembler implements a **two-pass design** to handle forward references to labels efficiently. It processes a simple text file containing C6461 assembly instructions and produces two key outputs:

1. **A Listing File:** A human-readable file that shows the original source code alongside the memory address and the generated octal machine code for each line.

2. **A Load File:** A clean, two-column file containing only the memory addresses and the corresponding octal code, designed to be loaded directly into the simulator in future project parts.

This document provides a guide to the assembler's design, instructions for setting up the necessary environment, and a step-by-step walkthrough of how to compile and run the program.

2. Design Justification

Core Component: `Assembler.java`

The entire logic for the assembler is contained within `Assembler.java`. This centralizes the functionality and simplifies the project structure for Part 0. The design is based on the highly recommended two-pass approach.

Pass 1: Symbol Table Construction

The first pass is dedicated to identifying all labels in the source code and mapping them to their memory addresses.

- **Process:** The assembler reads the source file line by line, keeping track of the current memory location with a "location counter." When it encounters a label (e.g., `MyLabel:`), it stores the label and the current counter value in a `HashMap` that serves as the **Symbol Table**.
- **Justification:** This approach resolves the "forward reference" problem. Instructions can use labels that are defined later in the code (e.g., `JMP EndLoop` where `EndLoop:` is further down). By building a complete map of all labels first, Pass 2 knows the address of every label before it begins translation.

Pass 2: Machine Code Generation

The second pass performs the actual translation of assembly instructions into octal machine code.

- **Process:** The assembler reads the source file again, resetting the location counter. For each line, it parses the instruction and its operands. Using the Symbol Table from Pass 1 to resolve label addresses, it constructs the 16-bit binary machine code for the instruction. This binary string is then converted to its octal representation.
- **Justification:** Separating code generation from symbol discovery makes the logic cleaner and easier to debug. Each pass has a single, well-defined responsibility.

3. User Guide: Setting Up the Environment

Before compiling and running the assembler, you must ensure your system has the necessary development tools installed.

macOS Setup

For macOS users, a setup script is provided (`setup.sh`). To run it:

1. Open the Terminal.
2. Navigate to the project directory.
3. Make the script executable: `chmod +x setup.sh`
4. Run the script: `./setup.sh`

The script will check for **Homebrew**, **Java JDK 8+**, and **IntelliJ IDEA**.

Windows Setup

For Windows users, a setup script is provided (`windows_setup.bat`).

1. Navigate to the project directory in File Explorer.
2. **Double-click** the `windows_setup.bat` file.

A Command Prompt window will open and check for the **Java JDK 8+** and **IntelliJ IDEA**.

4. User Guide: How to Compile and Run the Assembler

The assembler is designed to be built into a runnable `.jar` file using IntelliJ IDEA.

Step 1: Build the JAR File

1. Open the project in IntelliJ IDEA.
2. In the menu bar, navigate to **File -> Project Structure**.
3. Select **Artifacts**, click the **+** icon, choose **JAR**, and then **From modules with dependencies**.
4. In the dialog, select `Assembler` as the **Main Class**.
5. Click **OK**.
6. Finally, go to the main menu and select **Build -> Build Artifacts... -> Build**.

This will create the `.jar` file in the `out/artifacts/YourProjectName_jar/` directory.

Step 2: Prepare the Test File

Ensure your assembly source code file (e.g., `test_program.txt`) is placed in the same directory as the newly created `.jar` file.

3. User Guide: How to Compile and Run the Assembler

The assembler is designed to be built into a runnable `.jar` file using IntelliJ IDEA.

Step 1: Build the JAR File

1. Open the project in IntelliJ IDEA.
2. In the menu bar, navigate to **File -> Project Structure**.
3. Select **Artifacts**, click the **+** icon, choose **JAR**, and then **From modules with dependencies**.
4. In the dialog, select `Assembler` as the **Main Class**.
5. Click **OK**.
6. Finally, go to the main menu and select **Build -> Build Artifacts... -> Build**.

Step 2: Prepare the Test File

Ensure your assembly source code file (e.g., `test_program.txt`) is placed in the same directory as the newly created `.jar` file.

Step 3: Run the Assembler via Command Line

1. **Open a terminal** (Terminal on macOS, Command Prompt on Windows).
2. **Navigate to the directory** containing your `.jar` and `.txt` files.
   ```
   # Example path
   cd path/to/your/project/out/artifacts/YourProjectName_jar/
   ```
3. **Execute the assembler** with the following command, passing the test file as an argument:
   ```
   java -jar YourProjectName.jar test_program.txt
   ```

Step 4: Verify the Output

After running the command, the assembler will print its progress to the console.

Two new files will be created in the directory:

- `test_program_listing.txt`
- `test_program_load.txt`

You can open these files to verify that the machine code was generated correctly.