# Documentation

# CSCI\_6461\_12: Project P0

## Group 10:

### Alekya Kowta

### Nikhil Arethiya

### Peiyan Liu

### Aryan Saxena

# Overview and Purpose

This assembler transforms assembly language source code written for the specific instruction set architecture (ISA) into executable machine code and accompanying symbolic listing files. It employs a standard two-pass assembly process where the first pass collects symbols (labels) and addresses, and the second pass performs instruction encoding.  
The assembler is designed for educational and foundational purposes, emphasizing clear modularity and correctness.

# Installation and Usage

Installation:

* Ensure the target system has Java Runtime Environment (JRE) version 8 or higher installed.
* Download or obtain the packaged Assembler.jar file.
* Also obtain the input assembly source file, typically named sourceProgram.txt.
* Place Assembler.jar and sourceProgram.txt together in the same folder/directory.

Usage:

* Open a terminal or command prompt.
* Navigate to the directory containing Assembler.jar and sourceProgram.txt.
* Run the assembler with the command:

A black text on a white background

AI-generated content may be incorrect.

The assembler reads the source file sourceProgram.txt, performs the two-pass assembly, and generates the following output files in the same directory:

* **ListingFile.txt** — assembly listing combining source lines and generated machine code.
* **LoadFile.txt** — machine code output formatted for loading into the target CPU or simulator.

# Input Specification

Assembly source code consists of instructions following the ISA syntax:

* Instruction Lines: Optionally begin with a label followed by opcode and operands.
* Labels: Identifiers ending with a colon (:) marking a memory address.
* Operands: Comma-separated values representing registers, addresses, or flags.
* Comments: Text following a semicolon (;) is ignored.
* Directives: LOC sets the initial address for subsequent instructions or data.
* Data Declarations: Lines starting with Data specify constant values at a given address.
* Operands accept numeric constants or symbolic labels resolved by the symbol table.

# Output Specification

* Listing File: Each line pairs machine code (in octal) with its corresponding source code line, allowing for debugging and verification.
* Load File: Contains only the octal representations of machine code instructions, formatted for loading into memory by a simulator or hardware loader.
* Errors in assembly produce error messages inline in the output for easier identification.

# Design and Structure

The assembler follows the two-pass assembly methodology:

**First Pass:**

Reads source lines, strips comments, indexes labels, and tracks instruction addresses.

**Second Pass:**

Parses instructions and operands, encodes instructions using opcode tables and symbol addresses, and produces machine code.  
Key components include:

* + Symbol Table: HashMap mapping labels to address integers.
  + Opcode Tables: Centralized instruction-to-opcode mapping.
  + Instruction Handlers: Modular functions handling specific instruction categories.
  + File I/O Utilities: Methods for reading input and writing output.

# Instruction Parsing and Encoding

* Instructions are tokenized into opcode and operands.
* Supported formats include register-register-memory, index address, register-only, and immediate address forms.
* Operands parsed carefully to handle optional indirect addressing.
* Machine codes are computed using bit-shifting combining opcode and operand bits per ISA specification.
* Symbolic addresses replaced with numeric equivalents from the symbol table.

# Error Handling

* Unknown instructions or invalid operand counts generate explicit error outputs.
* Missing labels or illegal symbols cause assembly errors reported in output.
* The assembler halts on major exceptions with stack trace logged.

# Extensibility

* New instructions added by updating opcode tables and adding handler functions.
* New addressing modes integrated by extending operand parsing and encoding logic.
* Modularity facilitates future enhancements without major rewrites.

# Limitations

* Does not currently support macros, conditional assembly, or complex expressions.
* Error recovery is minimal; one error may halt assembly.
* Limited to the provided instruction set architecture subset.
* Designed primarily for educational or simulator use, not production environments.

# Testing

* The project includes a folder called “TestFiles”
* Input Program can be changed in the code to test/testing.txt
* Expected behavior can be seen.

# References

* Instruction Set Architecture Specification Document
* Java Development Kit Documentation
* Related Academic Publications on Assembly Language Programming