Team 3 - Design Document for C6461 Assembler

Introduction

This document provides a detailed overview of the design of an assembler for the C6461 architecture. An assembler translates assembly language source code into machine code, allowing execution on an emulator or on hardware. The assembler operates in the label resolution phase and instruction translation phase.

Purpose

The primary purpose is to convert human-readable assembly code into octal machine instructions. It handles label resolution, opcode translation and generates both listing and load files formatted in octal representation as specified by the C6461 architecture.

System Overview

The assembler is implemented in Java and operates in two main passes:

- Pass 1: Label Resolution: Identifies labels and their memory addresses.
- Pass2 : Instruction Translation: Converts instructions into machine code and generates output files.

This two-pass design enables forward label references and ensures that all symbols are resolved before machine code generation.

Key Components

- **Symbol Table:**
  - Description: Stores labels and their associated memory addresses.
  - Data Structure: Map<String, Integer>(Java HashMap)
  - Purpose: Resolves label references in pass 2, ensuring correct addressing for instructions and data.
  - Key feature: Supports forward references by building a complete symbol table in pass 1.

- **Source File Reader:**
  - Description: Reads source file from disk in both passes.
  - Data Structure: BufferedReader wrapping FileReader
  - Purpose: Provides line-by-line access to source.txt in both passes.
  - Key feature: File read twice from disk.

- **Location Counter:**
  - Description: Tracks the current memory address during assembly.
  - Data structure: int
  - Purpose: Ensures correct memory locations for instructions and data.
  - Key Features: Handles LOC direction adjustments, incremented by isWordAllocatingOP()

- **Inner Classes:**
    - LineParts - holds label + rest returned by peelLabel()
    - Token - holds op + tail returned by tokenizeHead()

- **Whitespace Pattern:**
    - Description : Pre-compiled regex pattern \s+
    - Data Structure: java.util.regex.Pattern
    - Purpose: splits operand strings efficiently in splitCommaArgs()
    - Key feature: Compiled once at class load and reused on every call.

- **Output File Writers**
    - Description: Writes output directly to files, no intermediate lists.
    - Data Structure: PrintWriter, BufferedWriter and FileWriter
    - Purpose: Writes octal address/code pairs to both output files.
    - Key feature: single emitWord() call writes both files at once.

Detailed Design

(I) Pass 1: Label Resolution

Algorithm:

1. Initialize the location counter to 0.
2. Read the source file line by line.
3. Source lines are not stored.
4. Remove comments and tokenize each line
5. Process labels(words ending with :)
    a. Extract label name
    b. Store in the symbol table with the current address
6. Process directives:
    a. LOC: Update location counter to specific address.
    b. DATA: Increment location counter (allocates one word)
7. Process instructions: Increment the location counter for each valid instruction.

Output:

- Populated symbol table with all the label addresses.
- Validation of source file syntax.

(II) Pass 2: Instruction Translation

Algorithm:

1. Reset the location counter to 0.
2. Re-read source.txt from disk using BufferReader.
3. Handle directives:
    a. LOC: Update location counter, add to listing without machine code.
    b. DATA: General data value or resolve label reference.
4. Handle instructions:
    a. Look up opcode hardcoded inside encode().
    b. Parse operands and resolve addresses/labels.
    c. Encode 16-bit machine code using bit field layout.
    d. Add to both listing and load file outputs.

5. Generate output files in octal format.
6. Machine Code Encoding:
    a. Bits 15-10: Opcode (6 bits)
    b. Bits 9-8: R field (register, 2 bits)
    c. Bits 7-6: IX field (index register, 2 bits)
    d. Bit 5: I field (indirect addressing, 1 bit)
    e. Bits 4-0: Address/Immediate field (5 bits)

Output:

1. **Listing File (listing.txt)**
    a. Content: Memory address (octal), machine code (octal) and original assembly line.
    b. Format: “ AAAAAA MMMMMM original_source_line”
    c. Purpose: Comprehensive view for debugging and verification.
2. **Load File (load.txt)**
    a. Memory address(octal) and machine code (octal) pairs only.
    b. Format: “AAAAAA MMMMMM”
    c. Purpose: Direct input for C6461 simulator loading.

Design Decisions:

- **Two-Pass Architecture:**
  Rationale: Chosen to handle forward label references elegantly. An alternative single-pass design would require complex backpatching.

- **HashMap for Symbol Table:**
  Rationale: O(1) lookup time for label resolution. An alternative linear search would be O(n) and slower for large programs.

- **Separate Listing and Load Files**
  Rationale: The listing file includes source for human readability and the load file is optimized for machine loading.

- **Octal Output Format**
  Rationale: Matches C6461 specification requirements and traditional minicomputer Conventions.

- **Inner classes for structured parsing**
  Rationale: LineParts and Token inner classes make parsing self-documenting and eliminate off-by-one errors when splitting labels from instructions and opcodes from Operands.

- **Hardcoded Opcodes Inside**
  Rationale: With only 5 instructions currently implemented, hardcoding opcodes directly inside encode() via if/else is simpler than maintaining a separate HashMap opcode table.

- **Static Compiled Regex Pattern**
  Rationale: The whitespace pattern WS is compiled once as a static field rather than

recompiling on every call to splitCommaArgs(), improving efficiency.

- **Direct File Writing via emitWord()**
  Rationale: Writing both output files simultaneously in a single emitWord() call guarantees listing.txt and load.txt always stay in sync.

Error Handling Strategy:

- File I/O Errors: Caught and reported with descriptive messages.
- Parse Errors: Invalid tokens identified and reported with line context.
- Label Errors: Undefined labels references cause a parse error, no dedicated label not found message exists.

Testing Strategy:

- **Test Program Components**
  - LOC directive testing with various addresses
  - Data directive testing with numeric and label values
  - Label definition and forward reference testing
  - Multiple instruction formats
  - Edge cases: label resolution

- **Verification Methods**
  - Manual verification of machine code encoding
  - Cross-check with C6461 instruction set specification
  - Output format validation against expected octal patterns

Performance Considerations:

- Time Complexity: O(n) for each pass, where n is the number of source lines
- Space Complexity: O(l) for labels
- Memory Usage: Minimal - suitable for large assembly programs

Future Enhancements:

- Indirect Addressing: Handling of the I-bit across all instructions.
- Macro Support: Expandable instruction sequences
- Error Recovery: Continue assembly after non-fatal errors
- Debug Symbols: Additional symbol table information for debugging

Implementation Notes:

- **Java Version Compatibility:**
  The assembler is designed for JDK 25 or later and uses the standard Java collections framework. No external dependencies are required, which keeps the implementation simple.

- **Cross-Platform Compatibility:**
  File I/O uses standard Java classes for portability. There's no platform-specific code or native libraries used, so the JAR file should work on Windows, Mac and Linux systems.

Conclusion:

This design document outlines the architecture and functionality of the C6461 Assembler implemented in Assembler.java. The two-pass design effectively handles label resolution while maintaining clear separation between symbol table construction in firstPass() and code generation in secondPass(). The implementation successfully translates C6461 assembly code format, providing a strong foundation for the full C6461 development cycle.