CSCI6461 Section 10 Group Project Documentation: Part 0

Team Number: 09

Group Members:

1. Jugal Gajjar
2. Kamalasankari Subramaniakuppusamy
3. Karan Patel

Introduction:

The objective of Part 0 of the CS6461 Computer Architecture Project is to create an assembler that translates machine code instructions from assembly language. This assembler will follow the C6461 Instruction Set Architecture (ISA) and convert legible instructions into binary representations. To simulate the C6461's operation in later project phases, the assembler will create load files and listing files using a one-pass or two-pass assembly approach.

Objectives of Part 0:

1. Develop a Working Assembler: The primary goal is to build an assembler that correctly translates assembly language instructions into machine-readable code for the C6461 Computer, ensuring that the output is compatible with the system's Instruction Set Architecture (ISA).
2. Handle Instruction Encoding and Addressing Modes: The assembler needs to correctly handle a variety of addressing modes (direct, indirect, and indexed) and instruction formats, ensuring that memory and registers are properly addressed.
3. Generate Load and Listing Files: The assembler must produce two key outputs: a load file containing the machine code to be executed by the simulator, and a listing file that pairs the original assembly instructions with their corresponding machine code and memory locations.
4. Ensure Error Handling: The assembler should provide clear feedback when encountering invalid instructions, memory overflows, or syntax errors, helping the user identify and correct issues in the assembly code.

Design Structure:

1. Main Class (‘Main.java’):
   - The entry point of the assembler.
   - Responsible for managing the overall flow: reading input files, invoking the assembler, and writing output files (load and listing files).

- Key functions:
    - Extracts assembly instructions from the input file.
    - Invokes the assembler to process each instruction.
    - Handles file operations like generating the final output files.

2. Assembler Class ('Assembler.java'):

- Core of the assembler, responsible for translating assembly language into machine code.
- Key Components:
    - Instruction Processing: Decodes and encodes each instruction based on the opcode and addressing modes.
    - Two-Pass Assembly: The assembler first scans to resolve memory locations and labels, then generates the final machine code.
    - Addressing Modes: Handles direct, indirect, and indexed addressing along with register and memory encoding.
    - Instruction Encoding: Encodes assembly instructions into a 16-bit binary format and converts them to octal.
- Key Methods:
    - handleComments(): Removes comments from assembly code.
    - assembleInstruction(): Translates instructions based on opcode and parameters.
    - getOpcodeBinary(): Encodes opcodes into binary values.

3. FileHandler Class ('FileHandler.java'):

- Manages file input/output operations for the assembler.
- Key Functions:
    - Reads assembly instructions from the input file.
    - Writes the generated machine code to the load file.
    - Creates the listing file that includes original instructions, machine code, and memory addresses.

4. Input File ('input.txt'):

- Contains the assembly instructions to be processed.
- Typically structured with instructions, memory locations, and comments.

5. Output Files:

- Load File ('load.txt'): Contains machine code (in octal) that will be loaded into the C6461 simulator for execution.
- Listing File ('listing.txt'): Displays the assembly code alongside the corresponding machine code and memory locations for easier debugging.

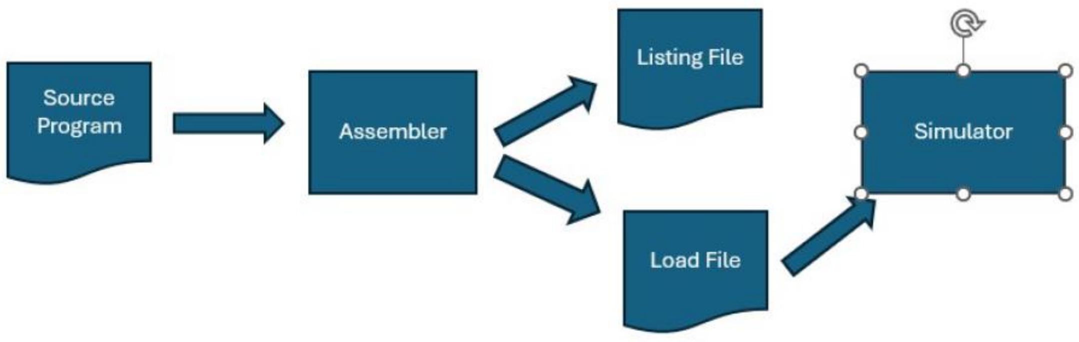

Fig. Overall Assembler Data Flow

GitHub Link:

https://github.com/JugalGajjar/CSCI6461-Computer-System-Architecture-Project/

Steps to Use:

1. Install OpenJDK 17.0.7
2. Place the test_input.txt and CSCI6461.jar files in the same directory
3. Execute .jar file. (java -jar CSCI6461.jar)

Test Cases:

Case 1: Data Handling

- Input: Data 15, Data End
- Expected Output:
  - At memory location 6: 000017 (Data 15)
  - At memory location 8: 002000 (End label, address 1024)
- Purpose: Ensure correct handling of Data directives and label resolution for the End label.

Case 2: Register Loading

- Input: LDR 1,0,6, LDR 2,0,7
- Expected Output:
  - LDR 1,0,6 = 002406 (Load register 1 with content from memory location 6)
  - LDR 2,0,7 = 003207 (Load register 2 with content from memory location 7)
- Purpose: Validate that the assembler correctly loads values into registers.

Case 3: Indexed Addressing

- Input: LDX 2,11
- Expected Output:

  - LDX 2,11 = 102411 (Load index register 2 with content from memory location 11)
- Purpose: Ensure correct handling of Data directives and label resolution for the End label.

<u>Case 4: Register-to-Memory Storing</u>

- Input: STR 1,0,13
- Expected Output:
  - STR 1,0,13 = 003413 (Store the content of register 1 to memory location 13)
- Purpose: Test if the assembler correctly handles storing register content into memory.

<u>Case 5: Conditional Branching</u>

- Input: JZ 1,0,20
- Expected Output:
  - JZ 1,0,20 = 020001 (Jump to location 20 if register 1 equals 0)
- Purpose: Ensure conditional branching works correctly.

<u>Case 6: Program Termination</u>

- Input: End: HLT
- Expected Output:
  - HLT = 000000 (Program halts at the End label)
- Purpose: Verify that the assembler encodes the HLT instruction correctly and stops the program.

Conclusion:

The assembler is the tool that converts human-readable assembly code into machine language that the C6461 computer architecture can execute. It handles instructions, memory locations, and addressing modes, ensuring everything is translated correctly. The assembler generates both a load file, which the simulator can run, and a listing file, which helps debug any issues. Overall, building this assembler lays the groundwork for running more complex simulations and provides a deeper understanding of how computer architectures operate at a low level.