CSCI 6461 COMPUTER SYSTEM ARCHITECTURE PROJECT
PART 0: DESIGN NOTES
TEAM NO: G
NAMES:

1. Sumairah Rahman
2. Sourabh Rajgole
3. Chiranjib Samantaray

---

1. Function and Purpose

The main function of this assembler is to translate assembly language, a form intelligible to humans, into a form of machine language, a binary language compatible with computer processing. An instruction consists of an operation code (opcode) and one or several operands. Assembler utilizes predefined bit values representing each operation and its respective arguments to generate the relevant machine language. With this mechanism, programmers can develop codes in a form easier for humans to understand and, at the same time, optimized for computational processing efficiency.

---

2. Components and Structure

Main Program: Assembler (main method in Assembler class)

- The assembler reads an input file containing assembly instructions.
- It processes each line according to predefined rules and translates it into machine code.
- The final output is written to a listing file and a load file.

Operation Metadata (opcodeMap)

- Defines details of each operation (e.g., ADD, HLT, LDR, etc.).
- Specifies bit lengths, required/optional arguments, and other attributes.

Argument Definition (ArgDefinition - handled in processInstruction method)

- Specifies the bit length and type of each argument (mandatory, optional, or unused).
- Ensures the assembler correctly handles arguments in the generated machine code.

Argument Types (ArgType)

- **MANDATORY:** Must always be included in the assembly code.
- **OPTIONAL:** May be supplied if necessary but not required.
- **UNUSED:** Takes up space in machine code (filled with zeros) even if not used.

Instruction Map (opcodeMap)

- Maps operation names (e.g., LDR, STR, HLT) to their corresponding metadata.
- Defines expected argument structure for each instruction.

---

3. Logic Flow

First Pass: Label Collection s Address Calculation (handled in main method)

- Reads input lines and extracts labels.

- Calculates memory locations for instructions and data.

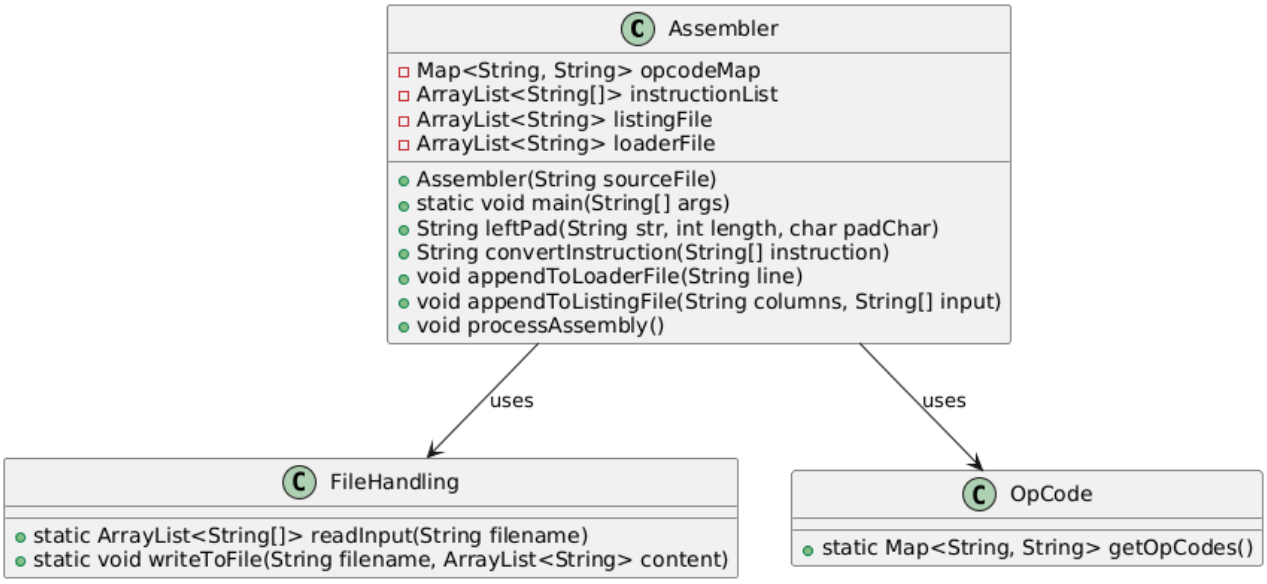
Second Pass: Machine Code Generation (handled in processInstruction method)

- Translate instructions into machine code using the predefined opcode map.
- Replaces label references with actual memory addresses.
- Generates binary representations of each instruction.

Processing Instructions

1. **Input Handling:** Reads input file line by line, ignoring comments (main method).
2. **Instruction Lookup:** Retrieves metadata from the opcodeMap (processInstruction method).
3. **Argument Handling:**
    - Ensures required arguments are present (processInstruction).
    - Assigns default values to unused arguments.
4. **Binary Conversion:**
    - Converts numeric values and labels to binary (getAddress method).
    - Assembles final machine code by concatenating binary strings.
5. **Output Generation:** Writes machine code to the load file and a formatted listing file (main method).

Object Model-



---

4. Important Elements

Detailed Operation Metadata Definition

Each instruction defines:

- **Number of Arguments:** How many operands are required?
- **Argument Properties:**
    - Bit lengths for each argument.
    - Whether they are required, optional, or unused.

Example: LDR Instruction

- Requires **four** arguments:
    - The first two (R, IX) are mandatory (2 bits each).

    - Third (I) is optional (1 bit).
    - Fourth (Address) is required (5 bits).

Argument Processing (processInstruction method)

- The processInstruction method defines the bit length and necessity of each argument.
- Ensure arguments fit within the expected bit constraints.

Instruction Assembly Process (processInstruction method)

1. Initializes an empty machine code string.
2. Iterates over argument definitions:

- Retrieves values from user input.
- Convert each argument to binary.
- Appends each binary segment to the final machine code string.

---

5. Handling Errors

- **Unknown Instructions:** Marks and skips unrecognized operations (main method error handling).
- **Missing Required Arguments:** Raises an error if a mandatory argument is missing (processInstruction).
- **Binary Conversion Errors:** Detects and reports invalid numeric values (isNumeric method).4
- **Error Messages:** Provide examples of error messages generated for different error conditions.

---

6. Flexibility and Extensibility

- **Easily expandable** by adding new operations to the opcode map.
- **Modular design** allows for modifications to argument handling and binary conversion.
- **Error handling system** provides clear debugging information.

---

7. Output Files

- **Listing File:** Human-readable format of the machine code with labels and original source lines.
- **Load File:** Machine code output ready for execution.
- **Examples:** Include examples of input assembly code and the corresponding output machine code.

---

8. Additional Considerations

- **Modularity**: The code separates diverse concerns, such as instruction lookup, argument handling, and binary conversion.
- **Scalability**: The assembler can handle complex instructions and different assembly languages by modifying opcodeMap and argument descriptions.

G. Overall Design Philosophy.

The assembler is intended to be:

- **Clear and simple:** Metadata defines each action and its arguments, making the program easy to understand and alter.

- **Error-Resistant:** The assembler provides unambiguous error handling for unfamiliar instructions and missing arguments, alerting users to input difficulties.
- **Extendable:** The software can handle more instructions, serving as a solid foundation for a more comprehensive assembler.

---

10. Summary

The Code Assembler converts human-readable assembly instructions to machine code in a systematic and flexible manner. It provides unambiguous metadata for each instruction, accepts many parameter types, and generates well-defined binary code. The design prioritises ease of use, scalability, and error management.