BRENNAN GARTH CASTILLO FINAL PROJECT
CSCI 3000 FALL 2023

INTRODUCTION

Why Java?

Java is well-suited for implementing an assembler due to its platform independence, robust support for file handling, and strong exception handling capabilities. As an assembler needs to interact with the file system to read and write source code and generated object code, Java's rich set of I/O libraries simplifies these operations. Additionally, Java's platform independence ensures that the assembler can run on different operating systems without modification, providing flexibility and accessibility to a wide range of users. This is particularly advantageous for an assembler, as it may need to process assembly code developed on various platforms.

Furthermore, Java's strong exception handling mechanisms contribute to the reliability of the assembler. During the assembly process, various errors and exceptional situations may occur, such as syntax errors, file not found exceptions, or unexpected input. Java's exception handling allows for the graceful handling of these errors, improving the program's robustness and providing meaningful error messages to the user. The ability to catch and handle exceptions ensures that the assembler can gracefully recover from unexpected situations, enhancing the overall stability and user experience of the tool. Overall, Java's portability, extensive I/O capabilities, and robust exception handling make it a suitable and reliable choice for implementing an assembler.

Why Eclipse?

Eclipse is chosen for this project not necessarily because it is the "best" option but due to its widespread adoption and strong support for Java development. As a renowned Integrated Development Environment (IDE), Eclipse provides a robust set of tools specifically tailored for Java programming. Its project management features facilitate the organization of modular structures commonly used in Java projects, and the built-in tools for coding, debugging, and version control contribute to a streamlined development process.

Additionally, Eclipse's extensive plugin ecosystem allows developers to customize their environment based on project requirements. The IDE's debugging and profiling tools are particularly valuable for identifying and resolving issues in complex projects like an assembler.

Being open source, Eclipse aligns with the preferences of many Java developers who appreciate freely available and community-driven tools. Its cross-platform compatibility further ensures a consistent development experience across various operating systems, making it a versatile choice for Java development teams. Nevertheless, it's important to note that the "best" IDE can vary based on individual preferences, team familiarity, and specific project needs, with alternatives like IntelliJ IDEA and NetBeans also being popular choices for Java development.

FUNCTIONS

The provided Java program serves as a simple two-pass assembler for a hypothetical assembly language. It is split into several functions (methods) to handle different aspects of the assembly process. Here's an overview of the main functions in the program:

1. `main` Method:
    a. Purpose:
        i. The entry point of the program.
    b. Functionality:
        i. Creates an instance of the `Main` class.
        ii. Defines an array of input file names to process.
        iii. Iterates through each input file and performs pass 1 and pass 2 of the assembly process.
        iv. Displays tables after processing each input file.
2. `pass1` Method:
    a. Purpose:
        i. Perform the first pass of the assembly process.
    b. Functionality:
        i. Read the input source file line by line.
        ii. Processes labels, opcodes, and operands.
        iii. Builds the symbol table (`symTable`) and location table (`locTable`).
        iv. Determines the program length.
        v. Updates the location counter.
3. `pass2` Method
    a. Purpose:
        i. Perform the second pass of the assembly process to generate object code.
    b. Functionality:

        i. Defines the opcode table (`optab`) for SIC/XE instructions.
        ii. Read the input source file again.
        iii. Generates object code for instructions and writes them to an output file.

4. 4. `displayObjectPrograms` Method:
    a. Purpose:
        i. Display the object programs generated during pass 2.
    b. Functionality:
        i. Read the object code file.
        ii. Displays the line number, location counter, source statement, and object code.
5. `displayTables` Method:
    a. Purpose:
        i. Display symbol and location tables along with program length.

    b. Functionality:
        i. Displays the symbol table and location table.
6. `displayObjectPrograms` Method :
    a. Purpose:
        i. Display the object programs with extended details.
    b. Functionality:
        i. Read the object code file.
        ii. Displays the line number, location counter, source statement, and object code.

These methods encapsulate different stages of the assembly process, providing a modular and organized structure to the assembler program. Each method has a specific role in reading, processing, and displaying information related to the assembly of source code.

MODULES/OBJECTS IMPLEMENTED

There are several key modules or objects that contribute to the functionality of the assembler. These can be identified based on their roles and interactions within the code:

1. Main Class (`Main`):

- Responsibility: The main class orchestrates the entire assembler process. It calls the necessary methods for both pass 1 and pass 2, processes input files, and displays tables or object programs.
- Key Methods:
  - `pass1(String inputFileName)`: Performs the first pass of the assembly process, building the symbol and location tables.
  - `pass2(String inputFileName, String outputFileName)`: Executes the second pass to generate object code based on the information gathered in pass 1.
  - `displayTables(String inputFileName)`: Displays tables, including the symbol and location tables, after processing an input file.
  - `displayObjectPrograms(String inputFileName, String outputFileName)`: Displays object programs generated during pass 2.

2. Symbol Table (`symTable`):
- Responsibility:Stores symbols and their corresponding addresses for reference during the assembly process.
- Usage: Populated during pass 1 with the labels encountered in the source code.

3. Location Table (`locTable`):
- Responsibility:Records the addresses associated with specific opcodes for reference during the assembly process.
- Usage: Populated during pass 1 based on the location counter and opcodes encountered.

4. Operation Code Table (`optab`):
- Responsibility: Contains the operation codes (opcodes) and their corresponding values for SIC/XE instructions.
- Usage: Used during pass 2 to look up the opcode values when generating object code.

5. File I/O (BufferedReader, BufferedWriter):
- Responsibility: Handles reading from and writing to files, facilitating the input and output of assembly code and generated object programs.
- Usage: Used in various parts of the code, especially in pass 1 to read input files and in pass 2 to write object code to an output file.

These modules work together to perform the multi-pass assembly process, translating source code into machine-readable object code. The relationships involve data flow between the tables, file I/O operations, and the execution of methods to carry out specific tasks during each pass.

MAIN DATA STRUCTURES/OBJECTS

The main data structures/objects in the provided assembler program are as follows:

1. Symbol Table (`symTable`):
    - Description: A `HashMap` that stores symbols (labels) encountered in the source code along with their corresponding addresses.
    - Role: Used to keep track of symbols and their locations for reference during the assembly process.

2. Location Table (`locTable`):
    - DescriptionAnother `HashMap` that associates operation codes (opcodes) with their respective addresses.
    - Role: Helps in maintaining the addresses corresponding to specific opcodes, aiding in the determination of program addresses during pass 1.

3. Operation Code Table (`optab`):
    - Description: A `HashMap` containing SIC/XE operation codes and their corresponding numerical values.
    - Role: Used during pass 2 to look up the numerical values associated with operation codes when generating object code.

4. File I/O (BufferedReader, BufferedWriter):
    - Description:Instances of `BufferedReader` and `BufferedWriter` classes for reading from and writing to files.
    - Role: Facilitates input and output operations, allowing the program to read the source code from files and write the generated object code to an output file.

5. Primitive Data Types (int, boolean):

- Description: Variables of primitive data types, such as `int` and `boolean`, are used for storing and manipulating numerical values, flags, and counters.
- Role: Supports various arithmetic and logical operations, as well as tracking the state of the assembly process.

These data structures/objects collectively form the backbone of the assembler, providing the necessary tools for storing, manipulating, and retrieving information throughout the multi-pass assembly process. The `HashMap` data structures are particularly valuable for their ability to quickly look up values based on keys, which is crucial in the context of symbol and opcode processing.

PROCESSING LOGIC

The processing logic of the assembler is divided into two passes, and it involves reading and interpreting the source code to generate object code.

Pass 1:

1. Initialize Data Structures:
    - Initialize symbol table (`symTable`), location counter (`locCounter`), and other necessary variables.

2. Read Source Code:
   - Read the source code line by line from the input file.
   - Tokenize each line to extract labels, opcodes, and operands.

3. Process Labels and Symbols:
   - If a label is present, update the symbol table with the label and its corresponding location counter.
   - Adjust the label based on whether the assembler is in a control section or program block.

4. Determine Program Length:
   - Track the program length by updating the location counter when encountering the "END" opcode.

5. Update Location Counter:

- Update the location counter based on different opcodes and directives.
- Handle special cases like literals, macros, and program blocks.

Pass 2:

1. Initialize Data Structures:
   - Initialize the opcode table (`optab`) with machine code values for SIC/XE instructions.
2. Read Source Code:
   - Read the source code line by line from the input file.

3. Generate Object Code:
   - Tokenize each line to extract labels, opcodes, and operands.
   - Check if the opcode is in the opcode table (`optab`).
   - If the opcode is present, generate the object code based on the instruction format (format 3 or format 4).
   - Consider displacement, format flags, and opcode values.

4. Write Object Code to Output File:
   - Write the generated object code to the output file.

5. Generate Additional Records:
   - Implement logic to generate modification records, refer records, define records, and end records.
   - Modify the code according to the format of the object file you are working with.

Display Object Programs:

1. Display Head Records:
   - Display any head records generated during pass 2.

2. Display Text Records:
   - Display the source code along with its corresponding object code (mnemonic instruction with machine code).

3. Display Modification Records:
   - Display any modification records generated during pass 2.

4. Display Refer Records:
   - Display any refer records generated during pass 2.

5. Display Define Records:
   - Display any define records generated during pass 2.

6. Display End Records:
   - Display any end records generated during pass 2.

PROBLEMS ENCOUNTERED AND RESOLUTION

a. ***Implementing A Two-Pass Assembler***

Pass 2 involves generating object code for SIC/XE instructions, considering both format 3 and format 4 instructions. The handling of these different instruction formats requires careful calculation of displacement and the inclusion of specific bits in the resulting object code. The assembler needs to look up operand values from the Symbol Table, which introduces additional complexity. It involves checking whether the operand is a symbol, resolving its value, and calculating the displacement for format 3 instructions. The assembler needs to differentiate between standard (format 3) and extended (format 4) instructions. The addition of the '+' symbol before the opcode indicates a format 4 instruction, which affects the calculation of displacement and the structure of the resulting object code. The construction of the object code involves bit manipulation to combine opcode values, format flags, and displacement values into a single 32-bit integer. This requires a good understanding of bitwise operations.

b. ***Laying out the Data Structure***

The use of three separate maps (symTable, locTable, and optab) introduces some complexity. Each map serves a distinct purpose (Symbol Table, Location Table, and Operation Codes), but this design choice might be seen as more intricate compared to a single data structure. The inclusion of flags (inLiteral, inControlSection, inProgramBlock) and their nested handling introduces complexity. These flags are used to track the state of the assembler as it processes different sections of the input code (e.g., literals, control sections, program blocks). The separation of program counters (locCounter, programLength) and lengths may add to the perceived complexity. While having separate counters for location and program length can provide clarity, it also contributes to the overall structure's intricacy.

c. ***Understanding the Object Code***

Generating object code in an assembler, particularly for complex architectures like SIC/XE, is challenging due to diverse instruction formats, symbol resolution complexities, addressing mode variations, handling of literal values, bitwise operations for code packing, extended instruction formats, error handling requirements, intricate file I/O operations, considerations for control sections and program blocks, the necessity for multiple passes, and architecture-specific intricacies. The process demands careful handling of these factors to ensure accurate translation of high-level assembly language instructions into machine code.

LEARNING/INSIGHT

The final project offers a rich learning experience by delving into the complexities of designing a two-pass assembler for the SIC/XE architecture. The intricacies of managing symbol tables, location counters, and different instruction formats showcase the challenges inherent in translating high-level assembly code into machine-readable instructions. Notably, the code highlights the significance of bitwise operations for object code generation, emphasizing the essential role of low-level programming skills in this context. The two-pass approach underscores the need for meticulous state management, especially when handling control sections, program blocks, and various instruction formats.

Moreover, the code sheds light on practical programming aspects, including file I/O operations and error handling. It emphasizes the importance of architecture-specific considerations, such as opcode tables, providing insights into the meticulous attention required to ensure accurate translation into machine code. Overall, the code serves as a comprehensive lesson in the intricacies of assembler design, offering valuable insights into low-level programming challenges and best practices for efficient assembly code translation.