Two-Pass Assembler in Swift: Documentation

Overview

The Two-Pass Assembler is a program that converts assembly language code into machine code. This assembler follows a two-pass process:

First Pass: It builds a symbol table that correlates all user-defined symbols (addresses) with their binary equivalent value.

Second Pass: It translates the assembly instructions into binary machine code using the information gathered during the first pass.

This document describes the structure and functionality of the Two-Pass Assembler implemented in Swift.

Key Components

Enumerations

InstructionType: This enumeration defines the different types of instructions the assembler can process:
MRI: Memory Reference Instruction.
nonMRI: Non-Memory Reference Instruction.
IO: Input/Output instruction.
pseudo: Pseudo-instructions (like `ORG`, `END`, etc.).

Structures

Instruction: Represents an assembly instruction, containing:
label: The label associated with the instruction, if any.
- **opcode**: The operation code (e.g., `LDA`, `STA`).
- **operand**: The operand for the instruction, if applicable.
- **type**: The type of instruction (`MRI`, `nonMRI`, `IO`, or `pseudo`).

SymbolTable: Maintains a dictionary mapping labels to their corresponding memory addresses. It provides functions to:

addSymbol(label: String, address: Int)`: Add a label and its address to the symbol table.
getAddress(label: String) -> Int?`: Retrieve the address associated with a label.

Two-Pass Assembler Class

Properties

instructions: [Instruction]: An array to store the parsed instructions.
symbolTable: SymbolTable: An instance of `SymbolTable` to store label-address mappings.
machineCode: [String]: An array to store the generated binary machine code.

Methods

determineType(opcode: String) -> InstructionType

Purpose: Determines the type of an instruction based on its opcode.
Functionality: Uses a `switch` statement to match the opcode with known instruction types (e.g., `MRI`, `nonMRI`, `IO`, `pseudo`). If the opcode is unknown, it triggers a fatal error.

readInstructions(_ assemblyCode: [String])

Purpose: Parses the assembly code to extract instructions and categorize them.
Functionality:
- Splits each line of the assembly code into parts (label, opcode, operand).
- Determines the instruction type using `determineType`.
- Stores the parsed instructions in the `instructions` array.

firstPass()

Purpose: The first pass of the assembler, which builds the symbol table.
Functionality:
- Initializes the location counter (`LC`) to 0.
- Scans each instruction to check for labels.
- If a label is found, it is added to the symbol table with the current location counter value.
- Handles pseudo-instructions like `ORG` (which sets the `LC`) and `END` (which terminates the first pass).

secondPass()

Purpose: The second pass of the assembler, which generates machine code.
Functionality:
- Resets the location counter to 0.
- Processes each instruction to generate corresponding binary machine code.
- Handles different instruction types:

Pseudo-Instructions: Like `ORG`, `DEC`, `HEX`, and `END`.
MRI Instructions: Uses the `convertMRIInstruction` function to combine the opcode and operand address.
Non-MRI Instructions**: Converts them to binary using `getNonMRIInstructionBinary`.
IO Instructions: Converts them to binary using `getIOInstructionBinary`.
Stores the generated machine code in the `machineCode` array.

getOpcodeBinary(_ opcode: String, indirect: Bool) -> String

Purpose: Converts an opcode into its corresponding binary string, considering whether the instruction is indirect.
Functionality: Uses a `switch` statement to return the binary representation of the opcode.

getNonMRIInstructionBinary(_ opcode: String) -> String
Purpose: Converts a non-MRI opcode into its corresponding binary string.
Functionality: Uses a `switch` statement to return the binary representation of the non-MRI opcode.

getIOInstructionBinary(_ opcode: String) -> String`**

Purpose: Converts an IO opcode into its corresponding binary string.
Functionality: Uses a `switch` statement to return the binary representation of the IO opcode.

convertMRIInstruction(opcode: String, operandAddress: Int, indirect: Bool) -> String`**

Purpose: Combines the opcode and operand address to form the full binary machine instruction for MRI instructions.
Functionality: Converts the opcode and operand address into their respective binary representations and combines them.

storeMachineCode(_ binaryCode: String, at location: Int)

Purpose: Stores the generated binary machine code with its corresponding location counter value.
Functionality: Appends the formatted binary code and location counter to the `machineCode` array.

runTests()

Purpose: Runs predefined test cases to verify the functionality of the assembler.
Functionality:
- Defines a set of test cases, each represented as an array of strings (assembly code).
- For each test case:
  - Resets the `instructions`, `symbolTable`, and `machineCode` arrays.
  - Runs the `readInstructions`, `firstPass`, and `secondPass` functions.
  - Prints the symbol table and the generated machine code to the console.

Example Test Cases

Test Case 1: Basic Functionality

```
[
    "START ORG 1000",
    "LABEL1 LDA VALUE",
    "ADD VALUE",
    "STA RESULT",
    "HLT",
    "VALUE DEC 5",
    "RESULT HEX 0",
    "END",
]
```

Description: This test case tests basic assembly functionality, including memory reference instructions (`LDA`, `ADD`, `STA`), pseudo-instructions (`ORG`, `DEC`, `HEX`), and non-MRI instructions (`HLT`).

Test Case 2: I/O Instructions

```
[
    "START ORG 2000",
    "INP",
    "OUT",
    "HLT",
    "END",
]
```

Description: This test case tests the handling of IO instructions (`INP`, `OUT`) and the `HLT` non-MRI instruction.

Test Case 3: Multiple ORG Instructions

```
[
    "START ORG 5000",
    "LDA VALUE",
    "ORG 6000",
    "STA RESULT",
    "ORG 7000",
    "VALUE DEC 15",
    "RESULT HEX 1234",
    "END",
]
```

Description: This test case tests the assembler's ability to handle multiple `ORG` pseudo-instructions that change the location counter.

Faulty Test Cases (Commented Out)

Test Case 4: Indirect Addressing (Commented Out)
Test Case 5: Undefined Label and Invalid HEX (Commented Out)

These test cases were commented out due to issues related to indirect addressing and handling undefined labels or invalid hexadecimal values. They can be uncommented and used for debugging and improving the assembler.

Running the Assembler

To run the assembler, simply instantiate the `TwoPassAssembler` class and call the `runTests()` method. The results of each test case, including the symbol table and generated machine code, will be printed to the console.

Conclusion

This document provides a detailed explanation of the Two-Pass Assembler implemented in Swift. The assembler successfully parses and translates assembly code into binary machine code, using a symbol table constructed during the first pass. It handles various instruction types, including MRI, non-MRI, IO, and pseudo-instructions. The test cases included in the `runTests()` method demonstrate the assembler's functionality and provide a framework for further testing and development.