PW
13/1/25

Vansh Gusnani
PG-46
10382107490

SSCD Lab-1

Assignment Title - Design of Pass1 of Two Pass Assembler

Aim - Design suitable data structures & implement Pass I of 2 Pass Assembler for pseudo machine

Objective - Design suitable data structure & implement Pass I of 2 pass Assembler for pseudo machine. Subset should consist of a few instructions from each category & few assembler directives.

Theory -

① Design specification of an Assembler:
Analysis phase

→ lexical Analysis: Break source code into tokens (opcodes, operands, labels).

→ Syntax Analysis: check structural correctness

→ Semantic Analysis: Validate symbol definitions & references.

→ Intermediate Representation: Generate symbol table & intermediate code.

(2) Design of Two-Pass Assembler:

→ Initialize: Set LOCCTR & data structures (symbol table, literal table)

→ Process each line:
- If label exists, add it to the symbol table with LOCCTR
- Determine instruction length & update LOCCTR.

→ Handle directives:
Process START, END, EQU, etc.

→ Generate Intermediate File:
Include Address & operation information.

(3) Contents of OPTAB

→ OPTAB (operation Table) stores:

- Mnemonic: opcode of instruction name
- opcode value: Machine representation
- Format: Instruction size (e.g., 1, 2, or 3/4 bytes)
- operands Required: Number/type of operands

(4) ERROR listing & Error Handling

→ Types of ERRORS:

- Syntax Errors: Invalid assembly language structure
- Semantic Errors: Undefined symbols or invalid addresses.

→ Handling:

- Generate error messages with line numbers
- Stop translation for critical errors ; log non-critical ones
- update symbol table with error flags for undefined symbols.

Input :- Assembly language program for pseudo machine with symbol operands

Output - opcode table, symbol table, Intermediate code.

Conclusion -

FAQs

① what errors are handled by the 2 pass assembler in Pass 1 & Pass 2 ?

Ans – Pass 1:

- Undefined symbols in operand fields (not get handled; flagged for Pass 2)
- Syntax Errors (e.g., invalid labels or instructions)
- Invalid assembler directives (e.g., wrong arguments for START, EQU)

Pass 2:

- Undefined symbols not resolved in Pass 1
- Invalid addresses or memory violations
- Errors in generating machine code (e.g., incorrect operand formats).

(2) Explain SYMTAB and the Assembler directives: EQU, START and ORIGIN.

Ans:-

- SYMTAB (Symbol Table):
  - → stores symbols (labels) with their corresponding addresses.
  - → used to resolve references during code generation

- Assembler Directives:

  EQU: Assigns a constant value to a symbol

  START: Specifies the starting address of the program.

  ORIGIN: changes the value of the location counter (LOCCTR).

(7) Explain LITTAB, POOLTAB and the Assembler directives LTORG and END.

Ans:

→ LITTAB (Literal Table):

stores literals (constants prefixed with =, e.g., =5) & their assigned memory addresses.

→ POOLTAB (Pool Table):

Tracks groups of literals processed by LTORG or at program END.

→ Assembler directives:

- LTORG: Allocates memory for literals at the current LOCCTR & resets LITTAB.
- END: Marks the end of the source program; resolves pending references & generates machine code.

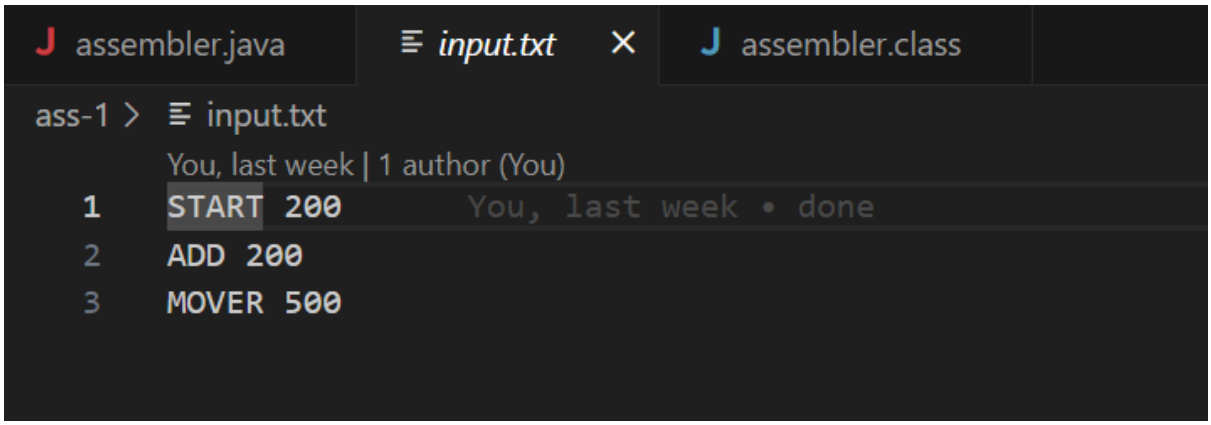
Code : -

```
import java.io.*;
// import java.util.ArrayList;
import java.util.HashMap;

public class assembler {
    public static void main(String[] args) {
        System.out.println("Hello, Assembler");

        HashMap<String, String[]> opcodeTab = new HashMap<>();
        opcodeTab.put("STOP", new String[]{"IS", "00"});
        opcodeTab.put("ADD", new String[]{"IS", "01"});
        opcodeTab.put("SUB", new String[]{"IS", "02"});
        opcodeTab.put("MULT", new String[]{"IS", "03"});
        opcodeTab.put("MOVER", new String[]{"IS", "04"});
        opcodeTab.put("MOVEM", new String[]{"IS", "05"});
        opcodeTab.put("COMP", new String[]{"IS", "06"});
        opcodeTab.put("BC", new String[]{"IS", "07"});
        opcodeTab.put("DIV", new String[]{"IS", "08"});
        opcodeTab.put("READ", new String[]{"IS", "09"});
        opcodeTab.put("PRINT", new String[]{"IS", "10"});
        opcodeTab.put("DC", new String[]{"DL", "01"});
        opcodeTab.put("DS", new String[]{"DL", "02"});
        opcodeTab.put("START", new String[]{"AD", "01"});
        opcodeTab.put("END", new String[]{"AD", "02"});

        String filepath = "input.txt";
        int locationCounter = 0;

        try (BufferedReader br = new BufferedReader(new FileReader(filepath))) {
```

```
        String line;
        System.out.println("Processing instructions from file:");

        while ((line = br.readLine()) != null){
            System.out.println(line);

            String[] tokens = line.split("\\s+");

            if (tokens.length != 2) {
                System.out.println("Invalid line format: " + line);
                continue;
            }

            String instruction = tokens[0];
            String value = tokens[1];

            if (instruction.equals("START")) {
                locationCounter = Integer.parseInt(value);
                System.out.println("LC initialized to: " + locationCounter);
                continue;
            }

            if (opcodeTab.containsKey(instruction)) {
                String[] details = opcodeTab.get(instruction);
                // Print the formatted output
                System.out.println( "LC=" + locationCounter + "(" + details[0] + ", " + details[1] + ") (C, " +
value + ")");

                locationCounter++;

            } else {
```

```
                System.out.println("Error: Unknown instruction '" + instruction + "' in line: " + line);
            }
        }
    }

    catch (IOException e) {
        System.err.println("Error reading the file: " + e.getMessage());
    }
  }
}
```

Input :-



Output: -