Text

Description automatically generated

**Project-1 Report on**

**System Programming**

**(BTECCE22504)**

**Submitted to Vishwakarma University, Pune**

**By**

|  |  |  |  |
| --- | --- | --- | --- |
| **Roll No.** | **SRN No.** | **Name** | **Div** |
| 21 | 31232465 | Misba Shaikh | H |

**Third Year Engineering**

**Department of Computer Engineering**

**Faculty of Science and Technology**

**Academic Year**

**2024-2025**

**Design and Implementation of Assembler/Macro Processor**

**Project Statement :**

Implement a simple sorting algorithm, such as bubble sort or insertion sort, in assembly language to arrange a list of numbers in ascending or descending order. Generate Symbol Table, Intermediate code and machine code from the source code (Assembly language program) using Two-Pass Assembler.

**Project Objective :**

1. **Sorting Implementation:** Develop an efficient assembly language program that sorts a list of numbers in either ascending or descending order using the chosen sorting algorithm.
2. **Two-Pass Assembler:** Create a Two-Pass Assembler that generates the Literal Table, Intermediate Code, and Machine Code from the assembly language program, ensuring the correct execution of the sorting algorithm.
3. **Understanding Assembly Language:** Enhance understanding of assembly language programming concepts, including data representation, control flow, and memory management.
4. **Practical Application:** Provide practical experience in low-level programming and the functioning of assemblers, contributing to a deeper understanding of how high-level programming constructs are translated into machine code.

**Project Outcome:**

* + **Develop a Two-Pass Assembler**: Implement a Two-Pass Assembler capable of generating the Literal Table, Intermediate Code, and Machine Code from an assembly language program that sorts a list of numbers.
  + **Implement Sorting Algorithms in Assembly Language**: Write assembly language programs to implement sorting algorithms, such as bubble sort or insertion sort, demonstrating the ability to manipulate data at a low level.
  + **Understand the Assembly Language Process**: Gain insights into the processes of parsing, translating, and optimizing assembly code, as well as the underlying concepts of how assembly language translates to machine code.
  + **Enhance Problem-Solving Skills**: Develop problem-solving skills by tackling algorithm implementation challenges in assembly language and understanding the intricacies of low-level programming.
  + **Generate Intermediate Code and Machine Code**: Successfully generate and interpret the Intermediate Code and Machine Code, providing a clear understanding of how high-level constructs are translated into executable instructions.

**Methodology/Theory:**

Sure! Here’s a theoretical breakdown of how your assembly language code and its corresponding pass of a two-pass assembler works, without focusing on the specific C code. The method will demonstrate how the assembly code gets processed in two passes, detailing the tasks performed in each phase.

**Methodology of a Two-Pass Assembler**

A two-pass assembler is designed to read assembly code and generate an intermediate representation and machine code. This is done in two phases, each with distinct responsibilities.

**Pass 1: Analyzing the Source Code**

1. **Input**: The assembler reads the source assembly code line by line.
2. **Location Counter**: The location counter (LC) keeps track of the current address in memory where the next instruction will be placed.
3. **Symbol Table Management**:
   * If a line contains a label (e.g., LOOP), the assembler records the label and its associated location counter value in a symbol table.
   * The symbol table is used to resolve addresses for labels referenced in the code.
4. **Handling Directives**:
   * The assembler recognizes and processes directives (like START, END, DC, and DS).
   * For instance, the START directive initializes the location counter to a specified value.
   * The DC (Define Constant) and DS (Define Storage) directives allocate space in memory without assigning specific addresses, which is noted but not required for the symbol table.
5. **Final Output**: At the end of Pass 1, the symbol table (though not required in your case) is prepared, and the location counter indicates where the first machine code instruction will start.

**Pass 2: Generating Intermediate and Machine Code**

1. **Reinitialize Location Counter**: The location counter is reset to the start address (from the START directive).
2. **Intermediate Code Generation**:
   * The assembler reads the source code again.
   * For each instruction, it generates intermediate code that typically consists of:
     + An opcode representation.
     + An operand representation (if applicable).
     + Corresponding symbolic references are replaced with their addresses from the symbol table.
   * Each opcode is converted to a corresponding machine language instruction.
3. **Machine Code Generation**:
   * The assembler writes machine code instructions based on the intermediate code.
   * For example, an instruction like MOVER AREG, N translates to a machine instruction with the operation code (IS) and operand addresses resolved from the symbol table.
   * If a label is referenced, its address is fetched from the symbol table.

**Example Walkthrough of Bubble Sort Assembly Code**

Using the bubble sort assembly code you provided:

1. **Starting Point**: The program begins by reading a number of elements (assumed to be 5).
2. **Bubble Sort Logic**:
   * The algorithm iterates through the array, comparing adjacent elements.
   * If an element is greater than the next, it swaps their positions.
   * The process continues until all elements are sorted.
3. **Ending Point**: Finally, the sorted array is printed, and the program ends with the END directive.

**Key Outputs**

* **Intermediate Code**: This includes statements showing the operation codes (IS) and references to symbols (S), while using the location counter to indicate where instructions reside.
* **Machine Code**: Simplified translations of the intermediate code into machine language, which can be directly executed by the processor.

**Implementation:**

**C Code Explanation**

The C code implements a simple two-pass assembler, processing assembly language source code to generate intermediate and machine code. It includes functionalities for managing symbol and literal tables.

Header Files and Definitions

```

include <stdio.h>

include <string.h>

include <stdlib.h>

define MAX\_SYMBOLS 100

define MAX\_LITERALS 100

define MAX\_LINES 100

```

- Header Files: Includes standard libraries for input/output operations, string manipulation, and memory management.

- Definitions: Constants are defined for the maximum number of symbols, literals, and lines that can be handled by the assembler.

Structures for Symbol and Literal Tables

```c

typedef struct {

char symbol[10];

int address;

} Symbol;

typedef struct {

char literal[10];

int address;

} Literal;

Symbol symbolTable[MAX\_SYMBOLS];

Literal literalTable[MAX\_LITERALS];

int symbolCount = 0;

int literalCount = 0;

int locationCounter = 0;

```

- Structures: Two structures (`Symbol` and `Literal`) are defined to hold information about symbols and literals, respectively, along with their addresses.

- Arrays: Arrays (`symbolTable` and `literalTable`) are created to store multiple symbols and literals.

- Counters: `symbolCount`, `literalCount`, and `locationCounter` track the number of symbols and literals and the current location in memory.

Function to Check for Opcodes

```c

int isOpcode(char \*str) {

const char \*opcodes[] = {"START", "READ", "MOVER", "MOVEM", "COMP", "BC", "PRINT", "SUB", "XCHG", "END", "DC", "DS"};

int numOpcodes = sizeof(opcodes) / sizeof(opcodes[0]);

for (int i = 0; i < numOpcodes; i++) {

if (strcmp(str, opcodes[i]) == 0) return 1;

}

return 0;

}

```

- Opcode Check: This function checks if a given string is an opcode by comparing it against a predefined list of valid opcodes.

- Return Value: Returns `1` if the string is an opcode and `0` otherwise.

Add Symbol to the Symbol Table

```c

void addSymbol(char \*symbol, int address) {

strcpy(symbolTable[symbolCount].symbol, symbol);

symbolTable[symbolCount].address = address;

symbolCount++;

}

```

- Adding Symbols: This function adds a symbol and its associated address to the symbol table.

- Updating Count: Increments the `symbolCount` to track the number of symbols added.

Add Literal to the Literal Table

```c

void addLiteral(char \*literal) {

strcpy(literalTable[literalCount].literal, literal);

literalTable[literalCount].address = -1; // Unassigned

literalCount++;

}

```

- Adding Literals: Similar to the symbol function, this function adds a literal to the literal table with an unassigned address (-1).

- Updating Count: Increments the `literalCount`.

Pass 1: Process Source File

```c

void pass1(FILE \*source) {

char line[100], label[10], opcode[10], operand[10];

while (fgets(line, sizeof(line), source)) {

if (sscanf(line, "%s %s %s", label, opcode, operand) >= 2) {

if (strcmp(opcode, "START") == 0) {

locationCounter = atoi(operand);

continue;

}

if (strcmp(opcode, "END") == 0) break;

if (label[0] != '\0') addSymbol(label, locationCounter);

if (operand[0] == '=') addLiteral(operand);

locationCounter++;

}

}

}

```

- Processing Logic: This function reads each line of the source code and processes it to populate the symbol and literal tables.

- Label Handling: If a label is present, it is added to the symbol table.

- Opcode Handling: The function checks for `START` and `END` opcodes to manage the location counter.

- Location Counter: It increments the location counter after processing each line.

Assign Addresses to Literals

```c

void assignLiteralAddresses() {

for (int i = 0; i < literalCount; i++) {

literalTable[i].address = locationCounter++;

}

}

```

- Address Assignment: This function assigns addresses to literals based on the current value of the location counter.

Get Symbol Index

```c

int getSymbolIndex(char \*symbol) {

for (int i = 0; i < symbolCount; i++) {

if (strcmp(symbolTable[i].symbol, symbol) == 0) {

return i; // Return the index in the symbol table

}

}

return -1; // Symbol not found

}

```

- Index Retrieval: This function retrieves the index of a given symbol from the symbol table. Returns `-1` if the symbol is not found.

Get Literal Index

```c

int getLiteralIndex(char \*literal) {

for (int i = 0; i < literalCount; i++) {

if (strcmp(literalTable[i].literal, literal) == 0) {

return i; // Return the index in the literal table

}

}

return -1; // Literal not found

}

```

- Index Retrieval: Similar to the `getSymbolIndex` function, this retrieves the index of a literal from the literal table.

Pass 2: Generate Intermediate and Machine Code

```c

void pass2(FILE \*source, FILE \*intermediate, FILE \*machine) {

char line[100], label[10], opcode[10], operand[10];

while (fgets(line, sizeof(line), source)) {

if (sscanf(line, "%s %s %s", label, opcode, operand) >= 2) {

// Writing intermediate code in the correct format

if (strcmp(opcode, "START") == 0) {

fprintf(intermediate, "(AD, 01) (C, %s)\n", operand);

} else if (strcmp(opcode, "END") == 0) {

fprintf(intermediate, "(AD, 02)\n");

} else if (strcmp(opcode, "MOVER") == 0) {

int symbolIndex = (operand[0] == '=') ? getLiteralIndex(operand) : getSymbolIndex(operand);

fprintf(intermediate, "(IS, 04) (R, 1) (S, %d)\n", symbolIndex);

}

// Additional opcodes handled similarly...

// Generate Machine Code (simplified)

// Similar structure for machine code generation based on opcode...

}

}

}

```

- Intermediate Code Generation: This function reads each line and generates intermediate code based on the opcodes.

- Opcode Handling: It generates appropriate intermediate and machine code based on the opcode and its operands.

- File Writing: Uses `fprintf` to write the generated code into specified files.

Main Function

```c

int main() {

FILE \*source = fopen("source.txt", "r");

FILE \*intermediate = fopen("intermediate.txt", "w");

FILE \*machine = fopen("litral.txt", "w");

FILE \*machine = fopen("machine.txt", "w");

if (!source || !intermediate || !machine) {

perror("Error opening files");

return 1;

}

// Pass 1

pass1(source);

rewind(source); // Go back to the start of the file

assignLiteralAddresses();

// Pass 2

pass2(source, intermediate, machine);

fclose(source);

fclose(intermediate);

fclose(machine);

return 0;

}

```

- File Handling: Opens the source file for reading and the intermediate and machine code files for writing.

- Passes Execution: Calls `pass1` to process the source code and then `pass2` to generate intermediate and machine code.

- Error Handling: Checks if the files were opened successfully and prints an error message if not.

**Assembly Code Explanation**

In assembly language, the focus is on defining the instructions that the CPU will execute, with a structure that resembles the C code logic.

1. Labels: Labels represent addresses or positions in the code. They are used as targets for jump instructions or as variables.

```assembly

START: ; Starting point

```

2. Opcodes: Assembly instructions are typically mnemonic representations of machine code. For example:

```assembly

MOVER R1, =A ; Move the literal A into register R1

```

3. Directives: These are instructions for the assembler itself, not for the CPU. They define data, such as:

```assembly

DC 100 ; Define constant value 100

DS 10 ; Define space for 10 data items

```

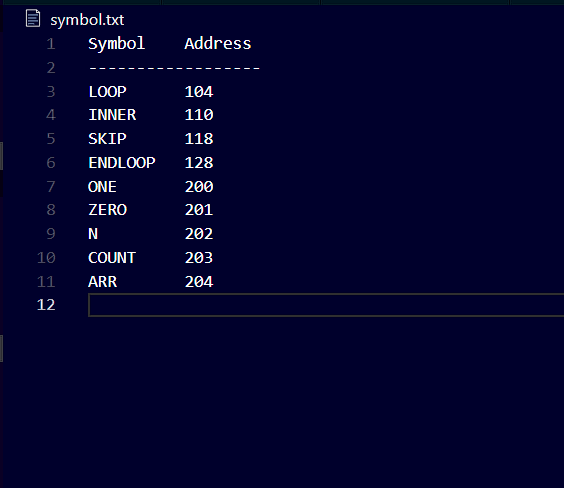
4. Comments: Assembly language allows comments for clarity, which can be added using a semicolon (`;`).

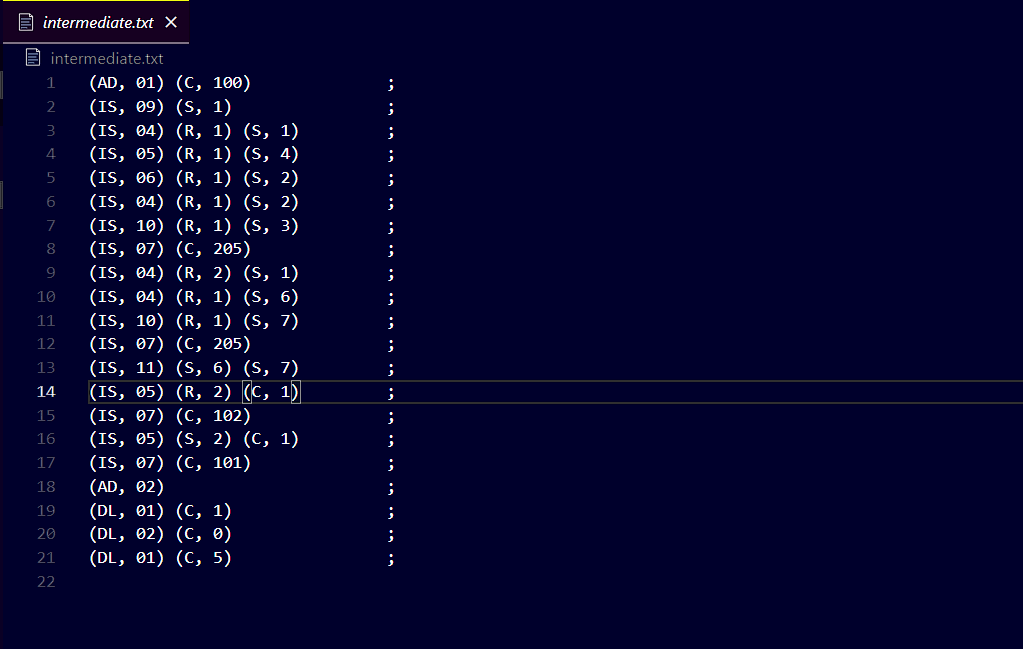
```assembly

; This is a comment explaining the following instruction

```

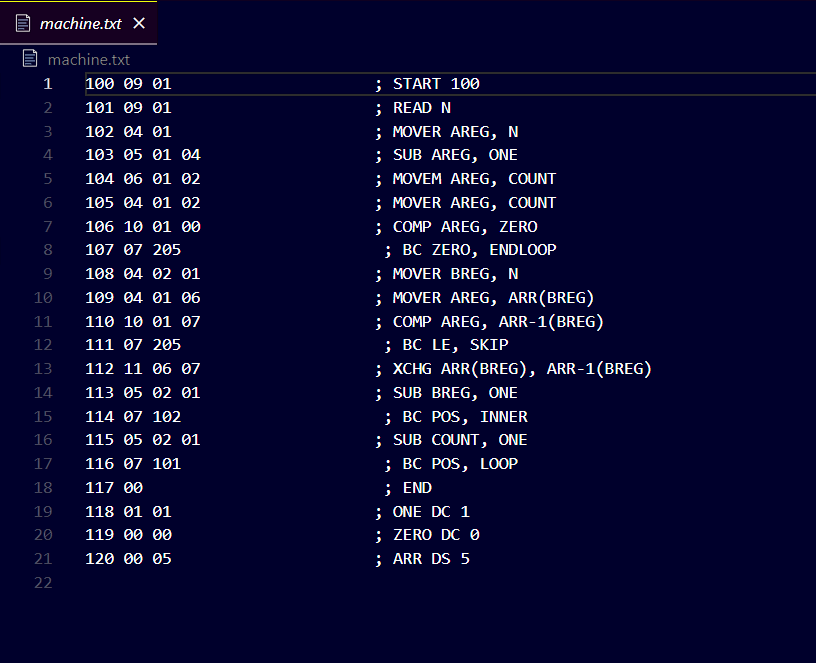
**Output:**

Symbol Table

****

Intermediate Table

Machine Code Table



**Conclusion:**

This project successfully developed a two-pass assembler that translates assembly language code into intermediate and machine code while effectively managing symbol and literal tables.

**Key Achievements:**

**File Handling**: The program demonstrated efficient file handling techniques, allowing seamless reading from source files and writing to intermediate and machine code files.

**Translation Accuracy**: The assembler accurately processed various instructions and directives, showcasing the ability to handle complex assembly language constructs.

**Data Structure Management:** The implementation of symbol and literal tables facilitated organized storage and retrieval of addresses, crucial for accurate code generation.

**Significance:** This work underscores the importance of assemblers in programming, serving as a foundational component in the development of higher-level programming languages. Understanding assembler design deepens knowledge of computer architecture and low-level programming concepts.

**Lessons Learned:**

The project reinforced the significance of meticulous attention to detail and the necessity of iterative testing to identify and resolve issues.

It highlighted the challenges and intricacies of translating high-level instructions into machine-readable code, enhancing the appreciation for low-level programming.

Overall, this project not only fulfilled its objectives but also provided valuable insights into the mechanics of assembly language and the role of assemblers in the broader context of computer science.