Experiment No. 2
Design & Implementation of Pass-2 of Two Pass Assembler
Date of Performance: 27-01-2025
Date of Submission: 03-02-2025

Aim: Design and Implementation of Pass 2 of Two Pass Assembler.

Objective: To study and implement Pass 2 of two pass assembler for IBM 360 Machine.

Theory:

An assembler is a program that accepts as input an assembly language program (source) and produces its machine language equivalent (object code) along with the information for the loader. The function of pass 2 of assembler is to generate machine program which is semantically equivalent to source program by taking input from pass 1. Databases generated are used to generate object code in pass 2.

Pass2: Generate object program

1. Look up value of symbols [STGET]
2. Generate instruction [MOTGET2]
3. Generate data (for DS, DC & literals)
4. Process pseudo ops [POTGET2]

Pass2: Databases

1. Copy of source program input to Pass1.
2. Location Counter (LC)
3. MOT [Mnemonic, length, binary m/c op code, etc.]
4. POT [Mnemonic & action to be taken in Pass2
5. ST [prepared by Pass1, label & value]
6. Base Table [or register table] indicates which registers are currently specified using 'USING' pseudo op & what are contents.
7. Literal table prepared by Pass1. [Lit name & value].

Flowchart of Pass 2:

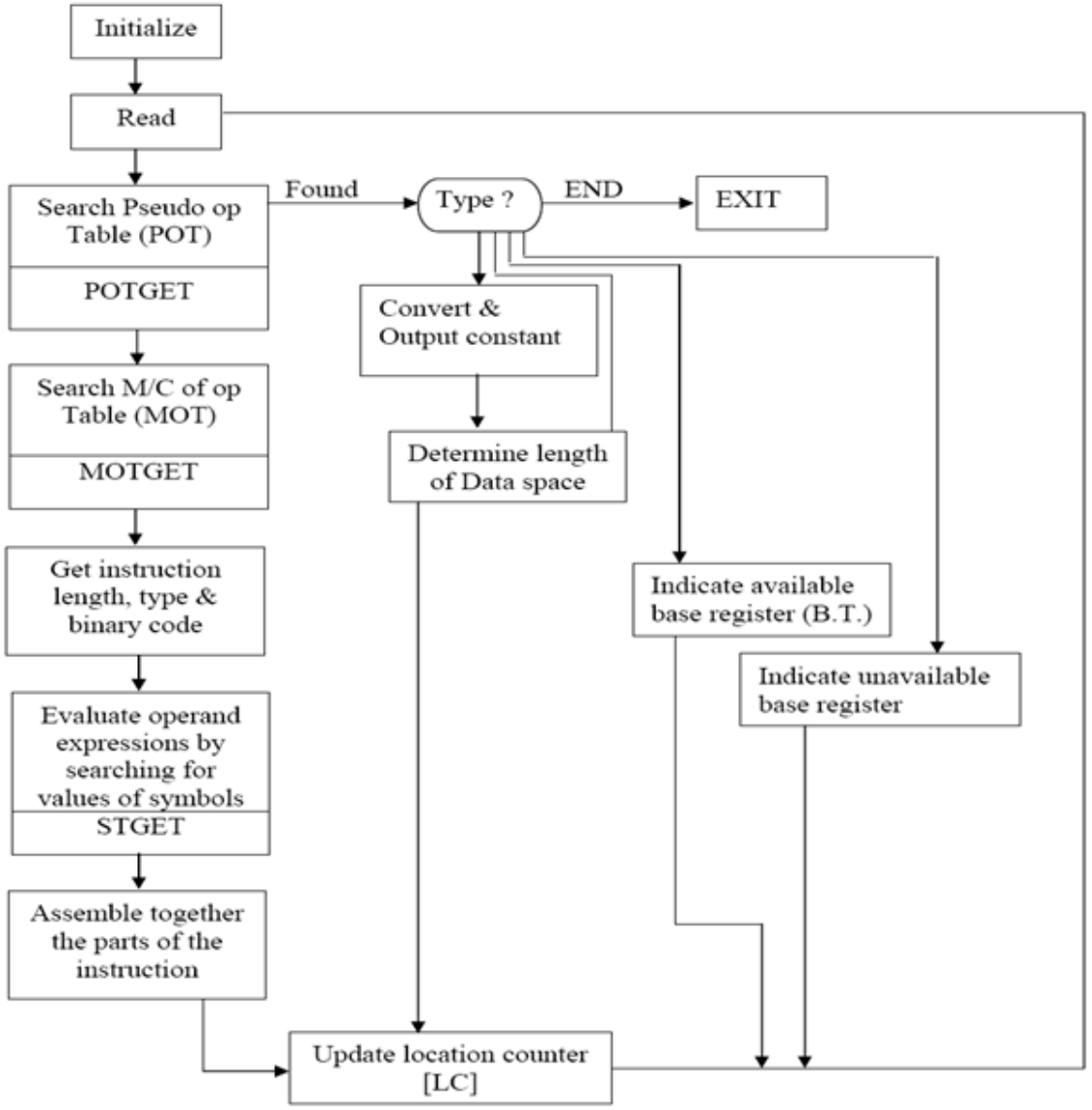


Fig. Flow chart of Pass -2

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_SYMBOLS 100
```

```
#define MAX_LITERALS 100

struct Symbol {
    char name[20];
    int address;
};

struct Literal {
    char value[20];
    int address;
};

struct Symbol symbolTable[MAX_SYMBOLS];
struct Literal literalTable[MAX_LITERALS];
int symbolCount = 0, literalCount = 0;

// Function to load Symbol Table from file
void loadSymbolTable() {
    FILE *fp = fopen("C:/Users/student/Desktop/symbol_table.txt", "r");
    if (!fp) {
        perror("Error opening symbol_table.txt");
        exit(1);
    }
    char line[100];
    fgets(line, sizeof(line), fp); // Skip header
    while (fgets(line, sizeof(line), fp)) {
```

```
        int index, address;
        char symbol[20];
        if (sscanf(line, "%d %s %d", &index, symbol, &address) == 3) {
            strcpy(symbolTable[symbolCount].name, symbol);
            symbolTable[symbolCount].address = address;
            symbolCount++;
        }
    }
    fclose(fp);
}

// Function to load Literal Table from file
void loadLiteralTable() {
    FILE *fp = fopen("C:/Users/student/Desktop/literal_table.txt", "r");
    if (!fp) {
        perror("Error opening literal_table.txt");
        exit(1);
    }
    char line[100];
    fgets(line, sizeof(line), fp); // Skip header
    while (fgets(line, sizeof(line), fp)) {
        int index, address;
        char literal[20];
        if (sscanf(line, "%d %s %d", &index, literal, &address) == 3) {
            strcpy(literalTable[literalCount].value, literal);
            literalTable[literalCount].address = address;
```

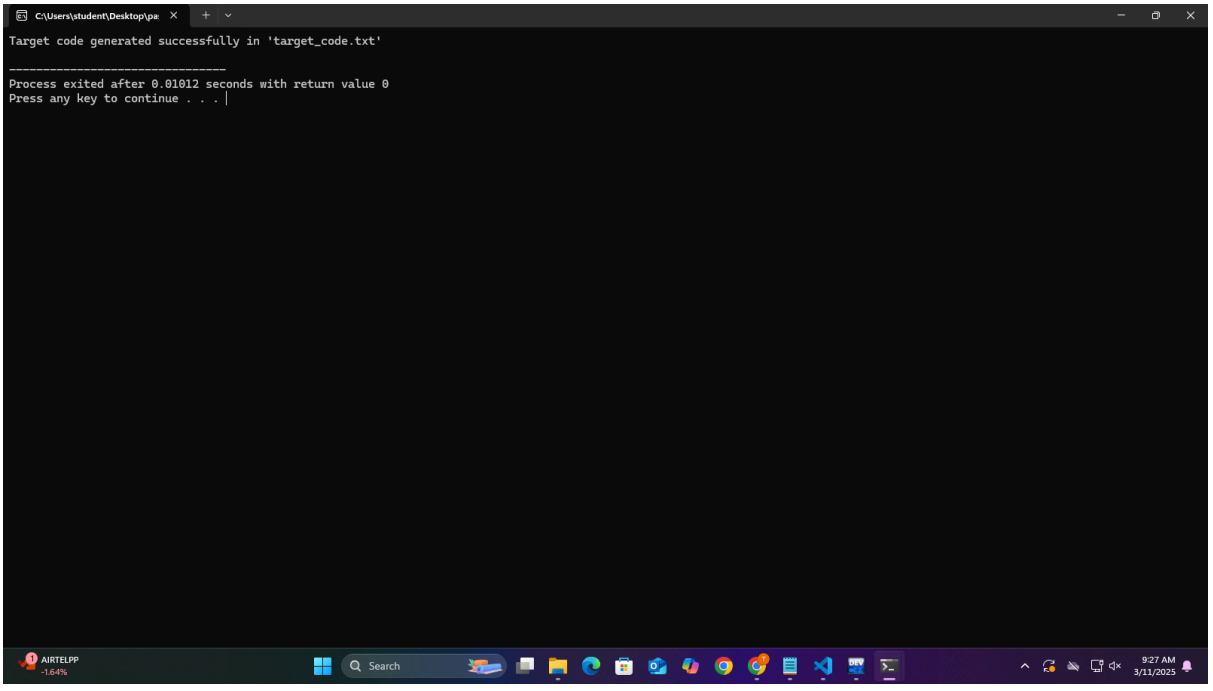
```
            literalCount++;
        }
    }
    fclose(fp);
}

int getLiteralAddress(int index) {
    if (index - 1 < literalCount)
        return literalTable[index - 1].address;
    return -1;
}

int getSymbolAddress(int index) {
    if (index - 1 < symbolCount)
        return symbolTable[index - 1].address;
    return -1;
}

void pass2() {
    FILE *input = fopen("C:/Users/student/Desktop/output.txt", "r");
    FILE *output = fopen("target_code.txt", "w");

    if (!input || !output) {
        perror("Error opening files");
        exit(1);
    }
```

```
char line[100];
int lc = 0;

while (fgets(line, sizeof(line), input)) {
    if (strstr(line, "(AD")) {
        continue;  // Skip assembler directives in target code
    }

    if (strstr(line, "(IS")) {
        int opcode, reg, index, addr;
        if (strstr(line, "(L,")) {
            sscanf(line, "%d) (IS, %d) %d (L, %d)", &lc, &opcode, &reg, &index);
            addr = getLiteralAddress(index);
        } else if (strstr(line, "(S,")) {
            sscanf(line, "%d) (IS, %d) %d (S, %d)", &lc, &opcode, &reg, &index);
            addr = getSymbolAddress(index);
        } else {
            sscanf(line, "%d) (IS, %d) (S, %d)", &lc, &opcode, &index);
            reg = 0;
            addr = getSymbolAddress(index);
        }
        fprintf(output, "%03d) %02d %02d %03d\n", lc, opcode, reg, addr);
    }
}
```

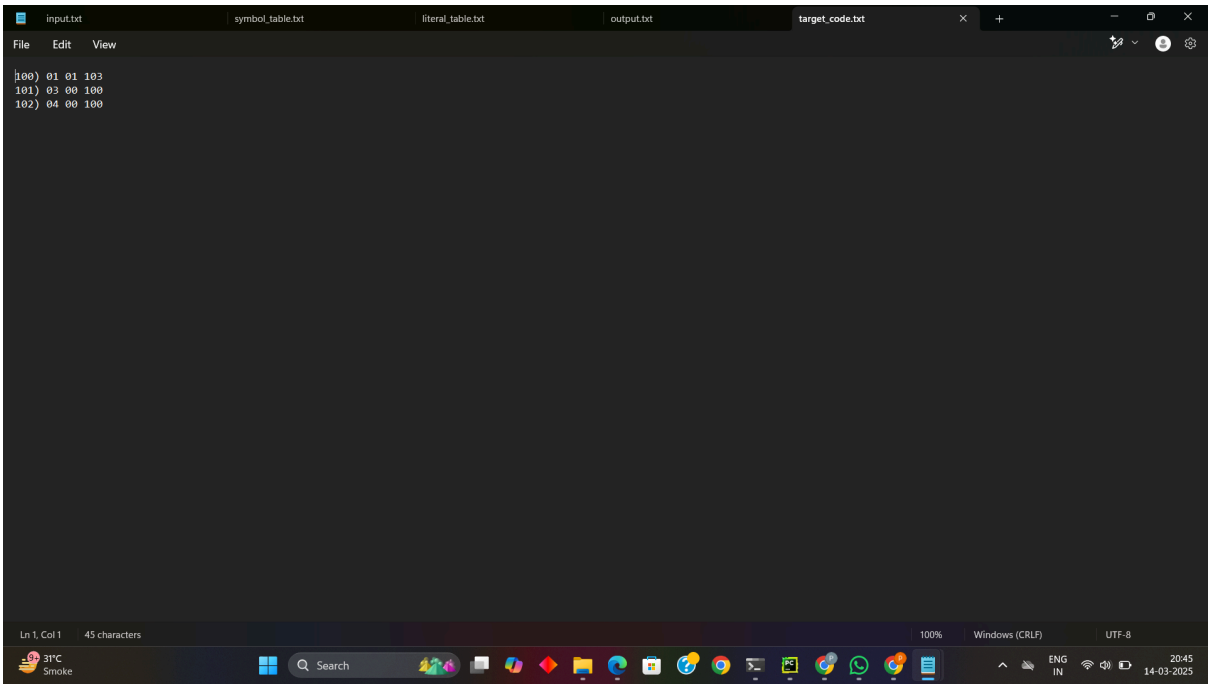
```
    fclose(input);
    fclose(output);
    printf("Target code generated successfully in 'target_code.txt'\n");
}

int main() {
    loadSymbolTable();
    loadLiteralTable();
    pass2();
    return 0;
}
```

Output:

Pass-2 Output:



target_code.txt



Conclusion:

→1. Working of Program by Providing Input Source Program :

The Pass-2 of the Two Pass Assembler accurately converts intermediate code into final machine code using the symbol and literal tables generated in Pass-1. When an input source program is provided, the assembler reads each instruction, resolves address references using the Symbol Table, and replaces symbolic operands with their actual memory addresses. It also processes assembler directives and generates the final object code in a structured format. The program demonstrates the correct working of Pass-2 by generating the required machine code output efficiently, ensuring proper instruction formatting and memory allocation as per the IBM 360 architecture.

→2. Databases Generated During Pass 1 of Two Pass Assembler :

The databases generated during Pass-1, such as the Symbol Table, Literal Table, and Intermediate Code, play a crucial role in the successful execution of Pass-2. The Symbol Table provides the actual memory addresses for labels and variables, while the Literal Table

stores constant values and their locations. These databases help resolve symbolic references into physical addresses during final code generation. The intermediate code acts as a bridge between the source program and machine code. The accurate creation and usage of these tables during Pass-2 ensures the correctness, consistency, and completeness of the final machine-level output for IBM 360 architecture.