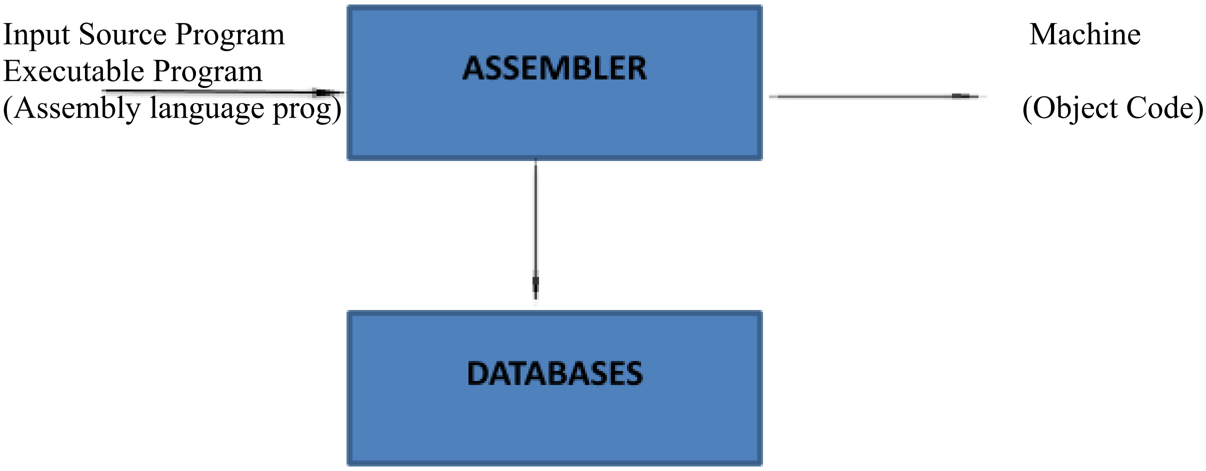
Experiment No.1
Design & Implementation of Pass-1 of Two Pass Assembler
Date of Performance: 20-01-2025
Date of Submission: 27-01-2025

Aim: Design & Implementation of Pass-1 of Two Pass Assembler.

Objective: To study and implement Pass 1 of two pass assembler.

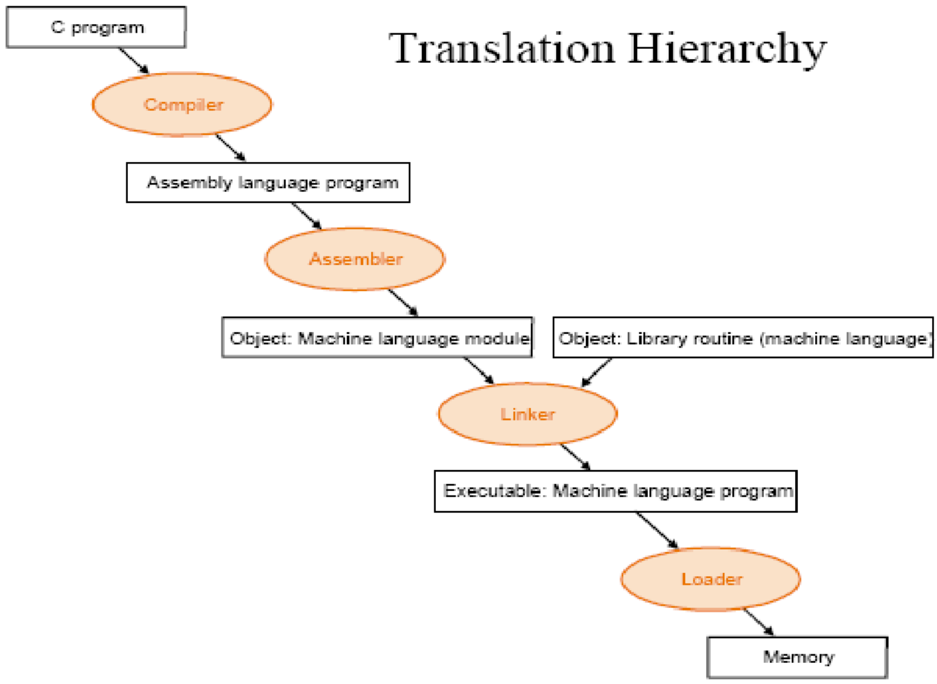
Theory:

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.



Relevance of Different system programs:

Translation Hierarchy



General Design Steps:

1. Specify the problem
2. Specify data structures
3. Define format of data structures
4. Specify algorithm
5. Look for modularity [capability of one program to be subdivided into independent programming units.]
6. Repeat 1 through 5 on modules

In following design IBM360 machine architecture is considered for reference.

Students should design 8086 / 8088 machine architecture.

1. Specify the problem

Pass1: Define symbols & literals.

1) Determine length of m/c instruction [MOTGET1]

2) Keep track of Location Counter [LC]

3) Remember values of symbols [STSTO]

4) Process some pseudo ops[EQU,DS etc] [POTGET1]

5) Remember Literals [LITSTO]

2. Data structure

Pass1: Databases

1. Input source program
2. "LC" location counter used to keep track of each instructions address.
3. M/c operation table (MOT) [Symbolic mnemonic & length]
4. Pseudo operation table [POT], [Symbolic mnemonic & action]
5. Symbol Table (ST) to store each lable & it's value.
6. Literal Table (LT), to store each literal (variable) & it's location.
7. Copy of input to used later by PASS-2.

3. Format of Data Structures

1. Machine Operation Table (MOT):

The op-code is the key and it's value is the binary op code equivalent, which is used for use in generating machine code.

The instruction length is stored for updating the location counter.

Instruction format is use in forming the m/c language equivalent.

6 bytes entry

Mnemonic Op-code (4 byte) character	Binary Op-code (1 byte) Hex	Instruction length (2 bits) (binary)	Instruction format (3 bits) (binary)	Not used in this design (3 bits)
"Abbb"	5A	10	001	-
"LOAD"	4A	10	001	-
"MOVb"	5E	01	000	-
"MVIb"	1E	11	100	-

b→ represents blank

Codes used-

Instruction length
01 = 1 half words
10 = 2 half words
11 = 3 half words

Instruction
000 - RR
001 – RX
010 – RS
011 – SI
100 – SS

2. Pseudo Operation Table (POT):

8 bytes / entry

Pseudo-op (5-bytes) (character)	Address of routine to process pseudo-op 3 bytes = 24 bit
"EQUbb"	P1EQG
"ENDbb"	P1END
"START"	P1START
"USING"	P1USING

These are presumably labels of routines in pass1

3. Symbol table & Literal table:-

14 bytes / entry

Symbol 8 byte character	Value (4 byte) Hex	Length 1 byte Hex	Relocation 1 byte character
"LOOPbbbb"	0000	01	"R"
"JOHNbbbb"	0010	04	"R"

Relation – R- Relative
A- Absolute (i.e. does not change)

4. Base table (BT):

4 bytes / entry

	Availability Indicator 1 byte Character	**Designated relative – address contents of base register (3 bytes = 24bit address) Hex**	
1	"N"	-	15 entries
2	"N"	-	
3	"N"	-	
.	:	-	
.	:	-	
15	"Y"	00 00 00	

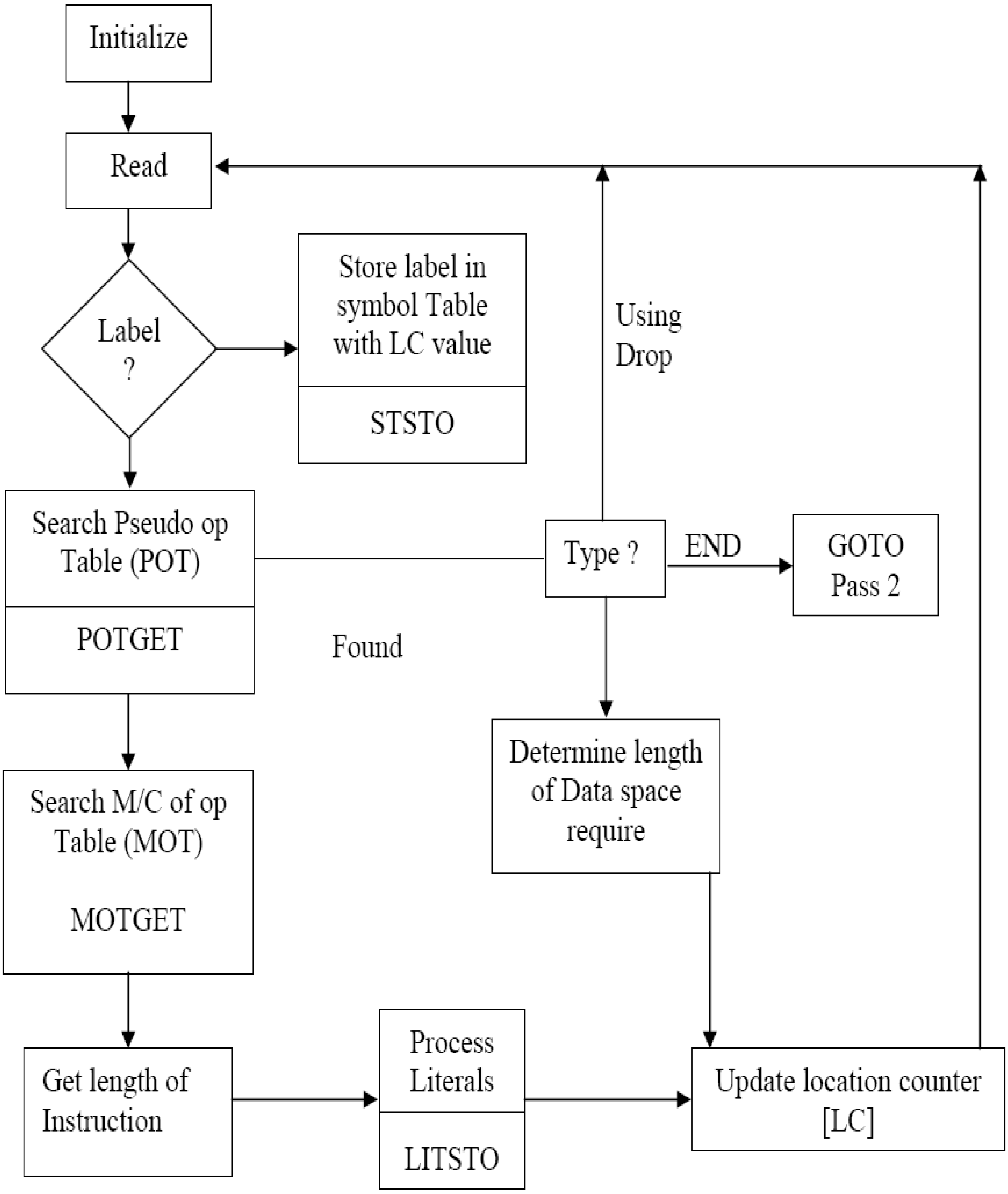
Codes:-

Availability:-

Y = register specified in USING pseudo-op

N = register never specified in USING pseudo-op or subsequently made unavailable by Drop pseudo-op.

Flow Chart of Pass 1 of Two Pass Assembler



Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAX_SYMBOLS 100
#define MAX_LITERALS 100
#define MAX_MNEMONICS 10
#define MAX_PSEUDO_OPS 5
#define MAX_LINE_LENGTH 100

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

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

struct Mnemonic {
   char mnemonic[10];
   int opcode;
```

```
};

struct PseudoOp {
    char mnemonic[10];
    int opcode;
};

const char *registers[] = {"AREG", "BREG", "CREG", "DREG"};

struct Mnemonic MOT[MAX_MNEMONICS] = {
    {"MOVER", 1}, {"MOVEM", 2}, {"ADD", 3}, {"SUB", 4}, {"MULT", 5},
    {"DIV", 6}, {"BC", 7}, {"COMP", 8}, {"PRINT", 9}, {"READ", 10}
};

struct PseudoOp POT[MAX_PSEUDO_OPS] = {
    {"START", 1}, {"END", 2}, {"EQU", 3}, {"ORIGIN", 4}, {"LTORG", 5}
};

struct Symbol symbolTable[MAX_SYMBOLS];
int symbolCount = 0;

struct Literal literalTable[MAX_LITERALS];
int literalCount = 0;

int searchMOT(char *mnemonic) {
    for (int i = 0; i < MAX_MNEMONICS; i++) {
```

```
        if (strcmp(MOT[i].mnemonic, mnemonic) == 0) {
            return MOT[i].opcode;
        }
    }
    return -1;
}

int searchPOT(char *mnemonic) {
    for (int i = 0; i < MAX_PSEUDO_OPS; i++) {
        if (strcmp(POT[i].mnemonic, mnemonic) == 0) {
            return POT[i].opcode;
        }
    }
    return -1;
}

int getRegisterOpcode(const char *reg) {
    if (strcmp(reg, "AREG") == 0) return 1;
    if (strcmp(reg, "BREG") == 0) return 2;
    if (strcmp(reg, "CREG") == 0) return 3;
    if (strcmp(reg, "DREG") == 0) return 4;
    return -1;
}

int getSymbolIndex(const char *symbol) {
    for (int i = 0; i < symbolCount; i++) {
```

```
        if (strcmp(symbolTable[i].name, symbol) == 0) {
            return i;
        }
    }
    return -1;
}

int getLiteralIndex(const char *literal) {
    for (int i = 0; i < literalCount; i++) {
        if (strcmp(literalTable[i].value, literal) == 0) {
            return i;
        }
    }
    return -1;
}

void addSymbol(const char *symbol, int address) {
    if (getSymbolIndex(symbol) == -1) {
        if (symbolCount < MAX_SYMBOLS) {
            strcpy(symbolTable[symbolCount].name, symbol);
            symbolTable[symbolCount].address = address;
            symbolCount++;
        }
    }
}
```

```
void addLiteral(const char *literal, int address) {
    if (getLiteralIndex(literal) == -1) {
        if (literalCount < MAX_LITERALS) {
            strcpy(literalTable[literalCount].value, literal);
            literalTable[literalCount].address = address;
            literalCount++;
        }
    }
}

void writeSymbolTable(FILE *fp) {
    fprintf(fp, "Index\tSymbol\tAddress\n");
    for (int i = 0; i < symbolCount; i++) {
        fprintf(fp, "%d\t%s\t%d\n", i + 1, symbolTable[i].name, symbolTable[i].address);
    }
}

void writeLiteralTable(FILE *fp) {
    fprintf(fp, "Index\tLiteral\tAddress\n");
    for (int i = 0; i < literalCount; i++) {
        fprintf(fp, "%d\t%s\t%d\n", i + 1, literalTable[i].value, literalTable[i].address);
    }
}

void generateIntermediateCode(FILE *output, int currentAddress, const char *mnemonic, int opcode, const char *operand1, const char *operand2, int isPseudoOp, int isLabel, char label[]) {
```

```
    if (isLabel) {
        addSymbol(label, currentAddress);
    }

    if (isPseudoOp) {
        fprintf(output, "(AD, %02d) ", opcode);
        if (strcmp(mnemonic, "START") == 0 || strcmp(mnemonic, "ORIGIN") == 0) {
            fprintf(output, "_ (C, %s)\n", operand1);
        } else {
            fprintf(output, "_\n");
        }
    } else {
        int regOpcode1 = getRegisterOpcode(operand1);
        int regOpcode2 = getRegisterOpcode(operand2);

        if (regOpcode1 != -1 && regOpcode2 != -1) {
                fprintf(output, "%03d) (IS, %02d) %02d %02d\n", currentAddress, opcode,
regOpcode1, regOpcode2);
        } else if (regOpcode1 != -1) {
            fprintf(output, "%03d) (IS, %02d) %02d ", currentAddress, opcode, regOpcode1);

            if (operand2 && operand2[0] == '#') {
                int literalIndex = getLiteralIndex(operand2 + 1);
                if (literalIndex == -1) {
                    addLiteral(operand2 + 1, -1);
                    literalIndex = literalCount - 1;
                }
```

```
            fprintf(output, "(L, %02d)\n", literalIndex + 1);
        } else {
            int symbolIndex = getSymbolIndex(operand2);
            if (symbolIndex == -1) {
                addSymbol(operand2, -1);
                symbolIndex = symbolCount - 1;
            }
            fprintf(output, "(S, %02d)\n", symbolIndex + 1);
        }
    } else {
        fprintf(output, "%03d) (IS, %02d) ", currentAddress, opcode);
        if (operand1 && operand1[0] == '#') {
            int literalIndex = getLiteralIndex(operand1 + 1);
            if (literalIndex == -1) {
                addLiteral(operand1 + 1, -1);
                literalIndex = literalCount - 1;
            }
            fprintf(output, "(L, %02d)\n", literalIndex + 1);
        } else {
            int symbolIndex = getSymbolIndex(operand1);
            if (symbolIndex == -1) {
                addSymbol(operand1, -1);
                symbolIndex = symbolCount - 1;
            }
            fprintf(output, "(S, %02d)\n", symbolIndex + 1);
        }
```

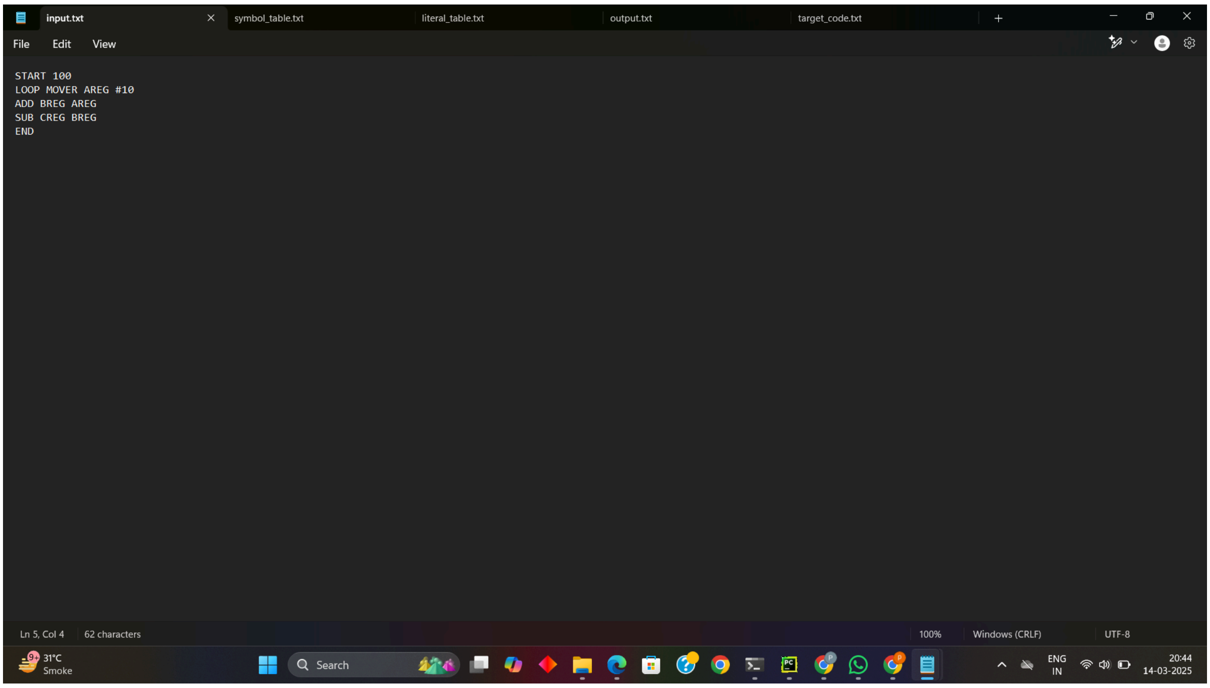
```
        }
    }
}

void pass1(FILE *source, FILE *output) {
    char line[MAX_LINE_LENGTH];
    int currentAddress = 0;

    while (fgets(line, sizeof(line), source)) {
        char label[20] = "", mnemonic[20] = "", operand1[20] = "", operand2[20] = "";
        int opcode = 0, isLabel = 0;

        int fields = sscanf(line, "%s %s %s %s", label, mnemonic, operand1, operand2);

        if (searchMOT(label) != -1 || searchPOT(label) != -1) {
            strcpy(operand2, operand1);
            strcpy(operand1, mnemonic);
            strcpy(mnemonic, label);
            label[0] = '\0';
        }

        opcode = searchMOT(mnemonic);
        int isPseudoOp = 0;
        if (opcode == -1) {
            opcode = searchPOT(mnemonic);
            isPseudoOp = 1;
```

```
        }

        if (opcode == 1 && strcmp(mnemonic, "START") == 0) {
            currentAddress = atoi(operand1);
            fprintf(output, "(AD, 01) _ (C, %d)\n", currentAddress);
            continue;
        }

        if (opcode == 2 && strcmp(mnemonic, "END") == 0) {
            for (int i = 0; i < literalCount; i++) {
                literalTable[i].address = currentAddress;
                fprintf(output, "%03d) (AD, 02) _ (L, %02d)\n", currentAddress, i + 1);
                currentAddress++;
            }
            break;
        }

        if (label[0] != '\0') isLabel = 1;
            generateIntermediateCode(output, currentAddress, mnemonic, opcode, operand1, operand2, isPseudoOp, isLabel, label);
        currentAddress++;
    }
}

int main() {
    FILE *source = fopen("C:/Users/student/Downloads/input.txt", "r");
    FILE *output = fopen("output.txt", "w");
```

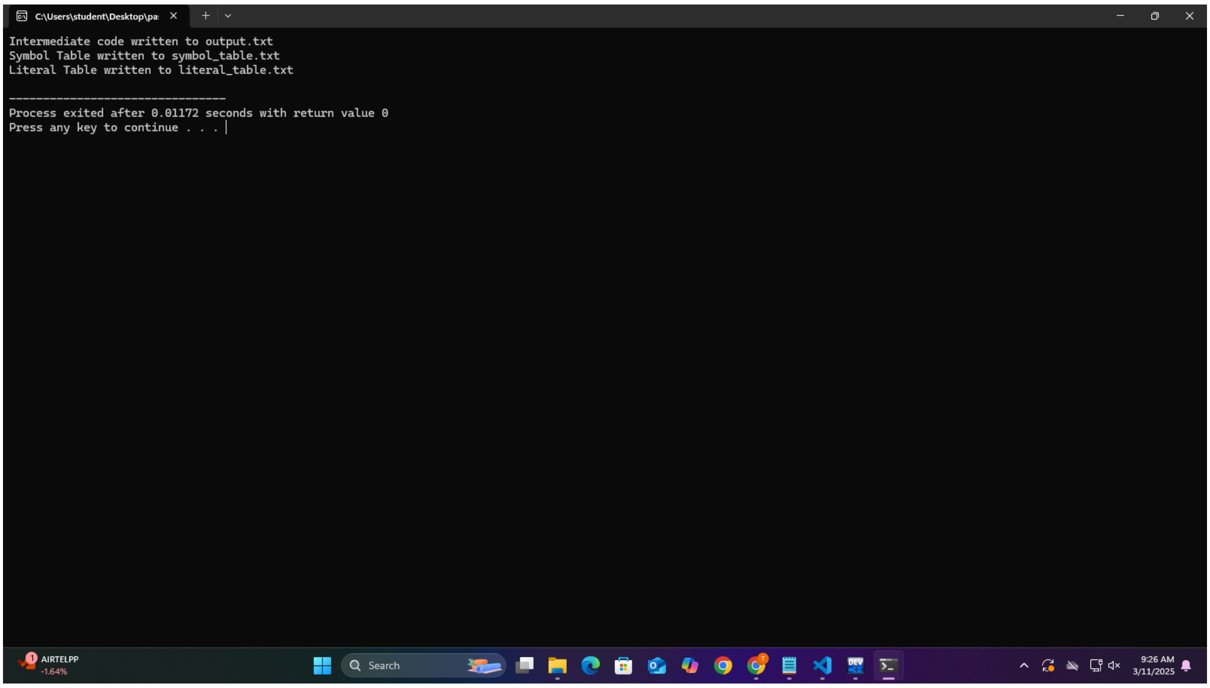
```
    FILE *symFile = fopen("symbol_table.txt", "w");
    FILE *litFile = fopen("literal_table.txt", "w");

    if (!source || !output || !symFile || !litFile) {
        perror("File open error");
        return 1;
    }

    pass1(source, output);

    writeSymbolTable(symFile);
    writeLiteralTable(litFile);

    fclose(source);
    fclose(output);
    fclose(symFile);
    fclose(litFile);

    printf("Intermediate code written to output.txt\n");
    printf("Symbol Table written to symbol_table.txt\n");
    printf("Literal Table written to literal_table.txt\n");

    return 0;
}
```

Output:

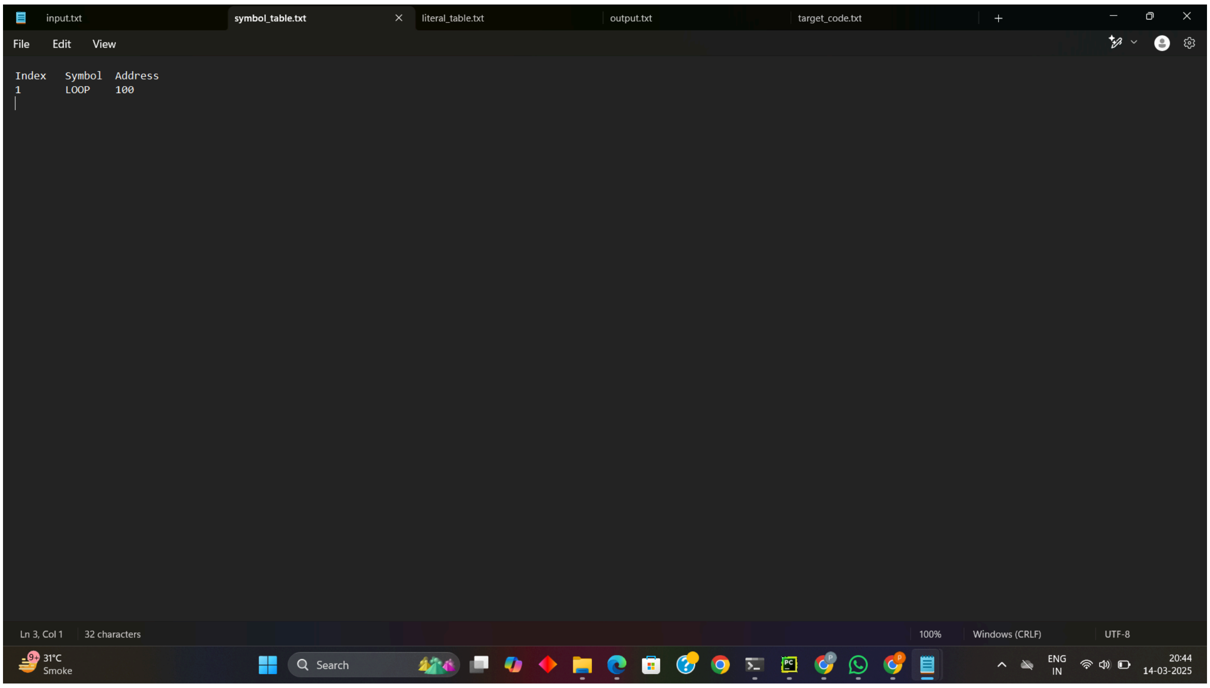
input.txt



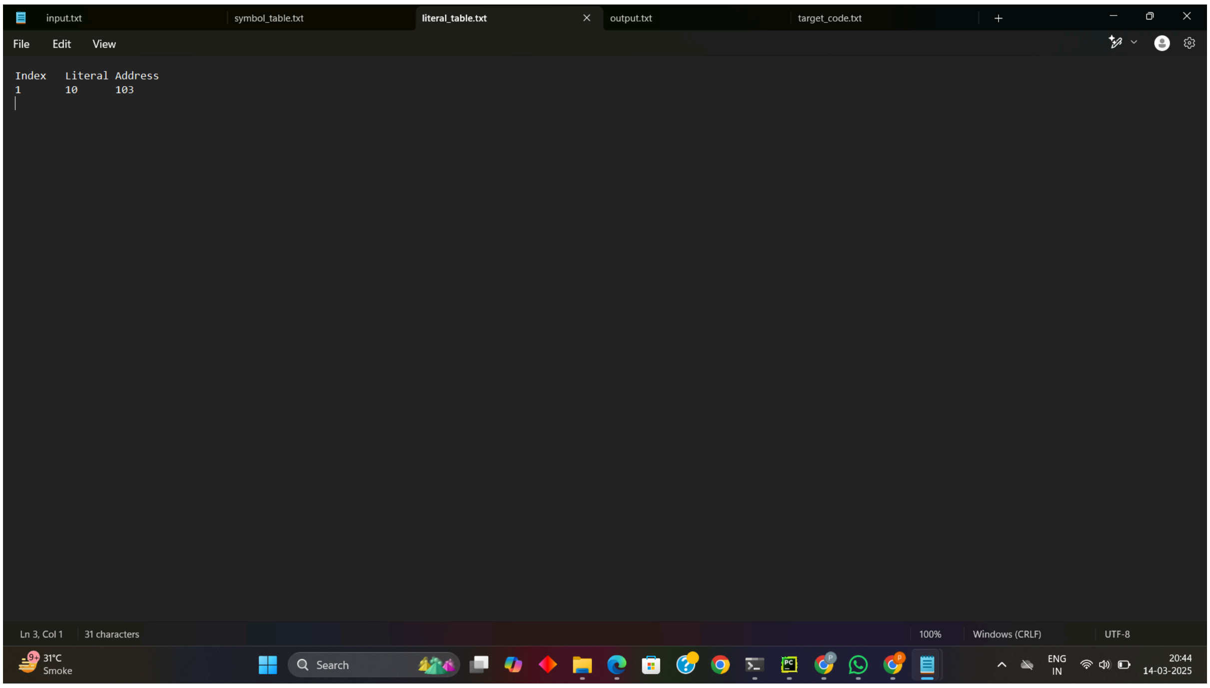
pass-1 output :



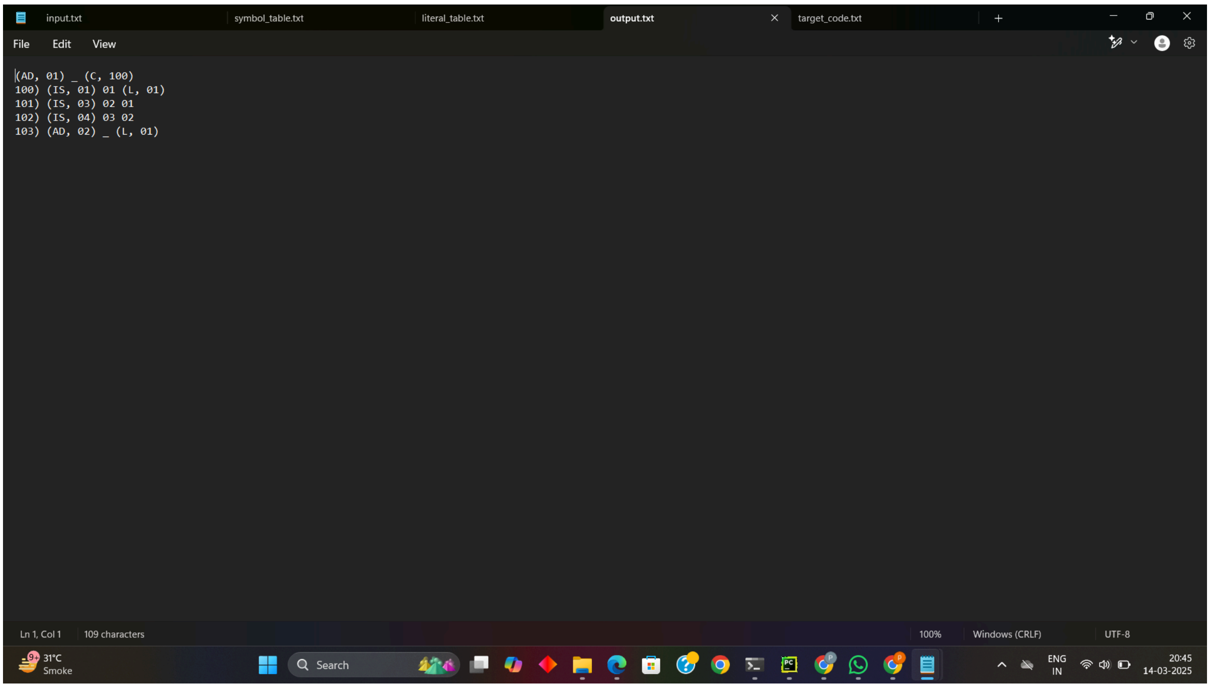
symbol_table.txt



literal_table.txt



output.txt



```
(AD, 01) _ (C, 100)
100) (IS, 01) 01 (L, 01)
101) (IS, 03) 02 01
102) (IS, 04) 03 02
103) (AD, 02) _ (L, 01)
```

Conclusion:

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

The Pass-1 of the Two Pass Assembler successfully processes the input source program by scanning it line by line to identify labels, mnemonics, and operands. It generates intermediate code by assigning addresses to instructions and capturing the program structure efficiently. The assembler builds a location counter and updates it based on instruction size, ensuring correct memory allocation. When an input program containing labels and directives is provided, the assembler correctly identifies and processes them. Overall, the working of the program demonstrates the logical flow of assembly processing and sets a solid foundation for the generation of final machine code in Pass-2.

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

During the execution of Pass-1 of the Two Pass Assembler, essential databases such as the Symbol Table and Literal Table are generated. The Symbol Table maintains the names and addresses of all symbols or labels encountered in the source program, ensuring proper memory reference. The Literal Table stores all constants and their assigned addresses. These tables act as critical references during Pass-2 for code translation. Additionally, an intermediate code file is created that includes address and operation information, simplifying the second pass. These structured databases enhance the modularity and clarity of the assembler's functionality and are vital to overall program execution.