**Instructions.txt-**

MOV A, 5

ADD B

SUB C

MUL D

DIV E

AND F

OR G

LOAD H

STORE I

DCR J

INC K

JMP 100

JNZ 200

LTORG

HALT

**1.LC,ST,LT,PT**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

// Define the maximum number of entries in the tables

#define MAX 100

// Structure for Mnemonic Operation Table (MOT)

struct MOT {

char mnemonic[10];

int opcode;

int size;

};

// Structure for Symbol Table (ST)

struct ST {

char symbol[10];

int address;

};

// Structure for Literal Table (LT)

struct LT {

int literal;

int address;

};

// Structure for Pool Table (PT)

struct PT {

int index;

};

// Function to initialize MOT (Mnemonic Operation Table)

void initializeMOT(struct MOT mot[]) {

strcpy(mot[0].mnemonic, "MOV");

mot[0].opcode = 1;

mot[0].size = 2;

strcpy(mot[1].mnemonic, "ADD");

mot[1].opcode = 2;

mot[1].size = 2;

strcpy(mot[2].mnemonic, "SUB");

mot[2].opcode = 3;

mot[2].size = 2;

strcpy(mot[3].mnemonic, "MUL");

mot[3].opcode = 4;

mot[3].size = 2;

strcpy(mot[4].mnemonic, "DIV");

mot[4].opcode = 5;

mot[4].size = 2;

strcpy(mot[5].mnemonic, "AND");

mot[5].opcode = 6;

mot[5].size = 2;

strcpy(mot[6].mnemonic, "OR");

mot[6].opcode = 7;

mot[6].size = 2;

strcpy(mot[7].mnemonic, "LOAD");

mot[7].opcode = 8;

mot[7].size = 2;

strcpy(mot[8].mnemonic, "STORE");

mot[8].opcode = 9;

mot[8].size = 2;

strcpy(mot[9].mnemonic, "DCR");

mot[9].opcode = 10;

mot[9].size = 1;

strcpy(mot[10].mnemonic, "INC");

mot[10].opcode = 11;

mot[10].size = 1;

strcpy(mot[11].mnemonic, "JMP");

mot[11].opcode = 12;

mot[11].size = 2;

strcpy(mot[12].mnemonic, "JNZ");

mot[12].opcode = 13;

mot[12].size = 2;

strcpy(mot[13].mnemonic, "HALT");

mot[13].opcode = 14;

mot[13].size = 1;

}

// Function to check if a string represents a number

int isNumber(char\* str) {

for (int i = 0; str[i] != '\0'; i++) {

if (!isdigit(str[i])) {

return 0; // Not a number

}

}

return 1; // Is a number

}

// Function to process instructions and fill tables

void processInstructions(char\* line, struct MOT mot[], struct ST st[], struct LT lt[], struct PT pt[], int\* lc, int\* stIndex, int\* ltIndex, int\* ptIndex) {

char mnemonic[10], operand[10];

sscanf(line, "%s %s", mnemonic, operand);

// Check for Mnemonic in MOT

int foundMnemonic = -1;

for (int i = 0; i < 14; i++) {

if (strcmp(mot[i].mnemonic, mnemonic) == 0) {

foundMnemonic = i;

break;

}

}

if (foundMnemonic == -1) {

printf("Error: Invalid mnemonic %s\n", mnemonic);

return;

}

// Process the instruction and generate machine code

printf("| %-4d | %-23s | %-17d |\n", \*lc, line, mot[foundMnemonic].opcode);

// Handle symbols and literals

if (isalpha(operand[0])) { // Symbol

int foundSymbol = -1;

for (int i = 0; i < \*stIndex; i++) {

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

foundSymbol = i;

break;

}

}

if (foundSymbol == -1) {

strcpy(st[\*stIndex].symbol, operand);

st[\*stIndex].address = \*lc + mot[foundMnemonic].size;

(\*stIndex)++;

}

} else if (isNumber(operand)) { // Literal

int foundLiteral = 0;

for (int i = 0; i < \*ltIndex; i++) {

if (lt[i].literal == atoi(operand)) {

foundLiteral = 1;

break;

}

}

if (!foundLiteral) {

lt[\*ltIndex].literal = atoi(operand);

lt[\*ltIndex].address = \*lc + mot[foundMnemonic].size;

(\*ltIndex)++;

pt[\*ptIndex].index = \*ltIndex;

(\*ptIndex)++;

}

}

\*lc += mot[foundMnemonic].size; // Increment LC based on the instruction size

}

int main() {

struct MOT mot[14];

struct ST st[MAX];

struct LT lt[MAX];

struct PT pt[MAX];

int lc = 0; // Line Counter

int stIndex = 0, ltIndex = 0, ptIndex = 0; // Table indexes

// Initialize Mnemonic Operation Table

initializeMOT(mot);

// Read instructions from file

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

if (fp == NULL) {

printf("Error: Could not open instructions.txt\n");

return 1;

}

char line[100];

printf("Line Counter [LC], Instruction Table\n");

printf("+------+--------------+--------------+\n");

printf("| LC | Instruction | Machine Code |\n");

printf("+------+--------------+--------------+\n");

// Process each instruction in the file

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

processInstructions(line, mot, st, lt, pt, &lc, &stIndex, &ltIndex, &ptIndex);

}

fclose(fp);

// Print the final tables

// Print Symbol Table (ST)

printf("\nSymbol Table (ST)\n");

printf("+------------+---------+\n");

printf("| Symbol | Address |\n");

printf("+------------+---------+\n");

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

printf("| %-10s | %-7d |\n", st[i].symbol, st[i].address);

}

printf("+------------+---------+\n");

// Print Literal Table (LT)

printf("\nLiteral Table (LT)\n");

printf("+------------+---------+\n");

printf("| Literal | Address |\n");

printf("+------------+---------+\n");

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

printf("| %-10d | %-7d |\n", lt[i].literal, lt[i].address);

}

printf("+------------+---------+\n");

// Print Pool Table (PT)

printf("\nPool Table (PT)\n");

printf("+------------+\n");

printf("| Literal Index |\n");

printf("+------------+\n");

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

printf("| %-12d |\n", pt[i].index);

}

printf("+------------+\n");

return 0;

}

**2.IC**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Mnemonic Operation Table (MOT)

typedef struct {

char mnemonic[10];

int opcode;

int size;

char type[3];

} MOTEntry;

typedef struct {

char symbol[10];

int address;

} SymbolTableEntry;

typedef struct {

int address;

char type[3];

char mnemonic[10];

char operand[20];

} IntermediateCodeEntry;

MOTEntry mot[] = {

{"MOV", 1, 2, "IS"}, {"ADD", 2, 2, "IS"}, {"SUB", 3, 2, "IS"}, {"MUL", 4, 2, "IS"},

{"DIV", 5, 2, "IS"}, {"AND", 6, 2, "IS"}, {"OR", 7, 2, "IS"}, {"LOAD", 8, 2, "IS"},

{"STORE", 9, 2, "IS"}, {"DCR", 10, 1, "IS"}, {"INC", 11, 1, "IS"}, {"JMP", 12, 2, "IS"},

{"JNZ", 13, 2, "IS"}, {"HALT", 14, 1, "IS"}, {"LTORG", 0, 0, "AD"}

};

int motSize = sizeof(mot) / sizeof(mot[0]);

// Function to search mnemonic in MOT

MOTEntry \*searchMOT(char \*mnemonic) {

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

if (strcmp(mot[i].mnemonic, mnemonic) == 0) {

return &mot[i];

}

}

return NULL;

}

void parseInstructions(const char \*filename) {

FILE \*file = fopen(filename, "r");

if (!file) {

printf("Error: Could not open file '%s'.\n", filename);

exit(1);

}

IntermediateCodeEntry ic[100];

SymbolTableEntry symbolTable[100];

int symbolCount = 0;

int icCount = 0;

int relativeAddress = 0;

char line[100];

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

char mnemonic[10], operand[10];

int numParts = sscanf(line, "%s %s", mnemonic, operand);

MOTEntry \*motEntry = searchMOT(mnemonic);

if (motEntry) {

IntermediateCodeEntry entry;

entry.address = relativeAddress;

strcpy(entry.type, motEntry->type);

strcpy(entry.mnemonic, mnemonic);

if (numParts == 2) { // Has an operand

if (operand[0] >= 'A' && operand[0] <= 'Z') { // Symbol

int found = 0;

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

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

found = 1;

break;

}

}

if (!found) {

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

symbolTable[symbolCount].address = -1; // To be resolved later

symbolCount++;

}

sprintf(entry.operand, "(symbol, %s)", operand);

} else { // Constant

sprintf(entry.operand, "(constant, %s)", operand);

}

} else {

strcpy(entry.operand, "");

}

ic[icCount++] = entry;

relativeAddress += motEntry->size;

}

}

fclose(file);

// Print Intermediate Code

printf("+------------------+------+----------+-----------------+\n");

printf("| Relative Address | Type | Mnemonic | Operand |\n");

printf("+------------------+------+----------+-----------------+\n");

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

printf("| %16d | %-4s | %-8s | %-15s |\n", ic[i].address, ic[i].type, ic[i].mnemonic, ic[i].operand);

}

printf("+------------------+------+----------+-----------------+\n");

}

int main() {

parseInstructions("instructions.txt");

return 0;

}

**3.MC**

#include <stdio.h>

#include <string.h>

// Define Mnemonic Operation Table (MOT) for IS (Imperative Statements)

struct MnemonicOpCode {

char mnemonic[10];

int opcode;

};

struct MnemonicOpCode motOpCode[] = {

{"MOV", 1}, {"ADD", 2}, {"SUB", 3}, {"MUL", 4}, {"DIV", 5},

{"AND", 6}, {"OR", 7}, {"LOAD", 8}, {"STORE", 9}, {"DCR", 10},

{"INC", 11}, {"JMP", 12}, {"JNZ", 13}, {"HALT", 14}

};

#define MOT\_SIZE (sizeof(motOpCode) / sizeof(motOpCode[0]))

// Define Registers and their opcodes

struct RegisterOpCode {

char register\_name[2];

int opcode;

};

struct RegisterOpCode registerOpCode[] = {

{"A", 1}, {"B", 2}, {"C", 3}, {"D", 4}

};

#define REG\_SIZE (sizeof(registerOpCode) / sizeof(registerOpCode[0]))

// Symbol Table

struct SymbolTable {

char symbol[2];

int address;

};

struct SymbolTable symbol\_table[] = {

{"B", 4}, {"C", 6}, {"D", 8}, {"E", 10}, {"F", 12}, {"G", 14},

{"H", 16}, {"I", 18}, {"J", 19}, {"K", 20}

};

#define SYMBOL\_SIZE (sizeof(symbol\_table) / sizeof(symbol\_table[0]))

// Literal Table

struct LiteralTable {

int literal\_index;

char literal\_value[5];

int address;

};

struct LiteralTable literal\_table[] = {

{0, "100", 25}, {1, "200", 26}

};

#define LITERAL\_SIZE (sizeof(literal\_table) / sizeof(literal\_table[0]))

// Intermediate code (as in Pass-1)

struct IntermediateCode {

int relative\_address;

char mnemonic[10];

char operand[10];

};

struct IntermediateCode intermediate\_code[] = {

{2, "ADD", "B"}, {4, "SUB", "C"}, {6, "MUL", "D"},

{8, "DIV", "E"}, {10, "AND", "F"}, {12, "OR", "G"},

{14, "LOAD", "H"}, {16, "STORE", "I"}, {18, "DCR", "J"},

{19, "INC", "K"}, {20, "JMP", "#100"}, {22, "JNZ", "#200"},

{24, "HALT", ""}

};

#define IC\_SIZE (sizeof(intermediate\_code) / sizeof(intermediate\_code[0]))

// Function to convert an integer to binary string

void toBinary(int num, int bits) {

for (int i = bits - 1; i >= 0; i--) {

printf("%d", (num >> i) & 1);

}

}

// Function to generate machine code from intermediate code

void generateMachineCode() {

printf("\nGenerated Machine Code:\n");

printf("+------------------+----------+---------+---------------------------+\n");

printf("| Relative Address | Mnemonic | Operand | Machine Code |\n");

printf("+------------------+----------+---------+---------------------------+\n");

for (int i = 0; i < IC\_SIZE; i++) {

int opcode = -1;

// Find the opcode for the mnemonic

for (int j = 0; j < MOT\_SIZE; j++) {

if (strcmp(intermediate\_code[i].mnemonic, motOpCode[j].mnemonic) == 0) {

opcode = motOpCode[j].opcode;

break;

}

}

if (opcode == -1) {

printf("| INVALID INSTRUCTION |\n");

continue;

}

// Print relative address, mnemonic and operand

printf("| %-16d | %-8s | %-7s | ", intermediate\_code[i].relative\_address,

intermediate\_code[i].mnemonic, intermediate\_code[i].operand);

// Convert and print opcode in binary (8 bits)

toBinary(opcode, 8);

printf(" ");

// Handle operand based on the type

if (strcmp(intermediate\_code[i].operand, "") == 0) {

// HALT instruction with no operand

printf("0000000000000000 ");

} else if (intermediate\_code[i].operand[0] == '#') {

// Operand is a literal

char \*literal\_value = intermediate\_code[i].operand + 1; // Remove the '#'

int address = -1;

// Search for the literal in the literal table

for (int j = 0; j < LITERAL\_SIZE; j++) {

if (strcmp(literal\_value, literal\_table[j].literal\_value) == 0) {

address = literal\_table[j].address;

break;

}

}

if (address == -1) {

printf("INVALID LITERAL ");

} else {

// Print operand address in binary (16 bits)

toBinary(address, 16);

}

} else {

// Operand is a symbol (register or variable)

int address = -1;

// Check if the operand is a symbol in the symbol table

for (int j = 0; j < SYMBOL\_SIZE; j++) {

if (strcmp(intermediate\_code[i].operand, symbol\_table[j].symbol) == 0) {

address = symbol\_table[j].address;

break;

}

}

if (address == -1) {

printf("INVALID SYMBOL ");

} else {

// Print operand address in binary (16 bits)

toBinary(address, 16);

}

}

printf("|\n");

}

printf("+------------------+----------+---------+---------------------------+\n");

}

int main() {

generateMachineCode();

return 0;

}

**4.MNT,MDT,PNTAB**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX\_LINES 100

#define MAX\_PARAMS 10

#define MAX\_NAME\_LEN 50

#define MAX\_CODE\_LEN 100

// Macro Definition Table (MDT)

char MDT[MAX\_LINES][MAX\_CODE\_LEN];

int mdt\_index = 0;

// Macro Name Table (MNT)

struct MNT {

char name[MAX\_NAME\_LEN];

int mdt\_index;

} MNT[MAX\_LINES];

int mnt\_count = 0;

// Parameter Name Table (PNTAB)

struct PNTAB {

char name[MAX\_PARAMS][MAX\_NAME\_LEN];

int count;

} PNTAB[MAX\_LINES];

int pntab\_count = 0;

void add\_macro(char macro\_name[], char params[][MAX\_NAME\_LEN], int param\_count) {

// Add to MNT

strcpy(MNT[mnt\_count].name, macro\_name);

MNT[mnt\_count].mdt\_index = mdt\_index;

mnt\_count++;

// Add parameters to PNTAB

for (int i = 0; i < param\_count; i++) {

strcpy(PNTAB[pntab\_count].name[i], params[i]);

}

PNTAB[pntab\_count].count = param\_count;

pntab\_count++;

// Add MACRO definition to MDT

sprintf(MDT[mdt\_index], "MACRO %s", macro\_name);

for (int i = 0; i < param\_count; i++) {

strcat(MDT[mdt\_index], " ");

strcat(MDT[mdt\_index], params[i]);

}

mdt\_index++;

}

void add\_macro\_line(char line[]) {

strcpy(MDT[mdt\_index], line);

mdt\_index++;

}

void end\_macro() {

strcpy(MDT[mdt\_index], "MEND");

mdt\_index++;

}

int find\_macro(char macro\_name[]) {

for (int i = 0; i < mnt\_count; i++) {

if (strcmp(MNT[i].name, macro\_name) == 0) {

return MNT[i].mdt\_index;

}

}

return -1;

}

void expand\_macro(char macro\_name[], char arguments[][MAX\_NAME\_LEN]) {

int macro\_index = find\_macro(macro\_name);

if (macro\_index == -1) return;

int param\_count = PNTAB[macro\_index].count;

// Replace parameters in MDT with arguments

for (int i = macro\_index + 1; i < MAX\_LINES; i++) {

if (strcmp(MDT[i], "MEND") == 0) break;

char line[MAX\_CODE\_LEN];

strcpy(line, MDT[i]);

for (int j = 0; j < param\_count; j++) {

char \*param\_position = strstr(line, PNTAB[macro\_index].name[j]);

if (param\_position != NULL) {

\*param\_position = '\0'; // Split line at parameter

strcat(line, arguments[j]);

strcat(line, param\_position + strlen(PNTAB[macro\_index].name[j]));

}

}

// Check if the line is non-empty before printing

if (strlen(line) > 0) {

printf("%s\n", line);

}

}

}

// Function to print tables in a tabular format

void print\_tables() {

printf("\n%-25s\n", "Macro Definition Table (MDT):");

printf("%-10s%-20s\n", "Index", "Definition");

for (int i = 0; i < mdt\_index; i++) {

printf("%-10d%-20s\n", i, MDT[i]);

}

printf("\n%-25s\n", "Macro Name Table (MNT):");

printf("%-10s%-15s\n", "Name", "MDT Index");

for (int i = 0; i < mnt\_count; i++) {

printf("%-10s%-15d\n", MNT[i].name, MNT[i].mdt\_index);

}

printf("\n%-25s\n", "Parameter Name Table (PNTAB):");

printf("%-10s%-20s\n", "Macro Name", "Parameters");

for (int i = 0; i < pntab\_count; i++) {

printf("%-10s", MNT[i].name);

for (int j = 0; j < PNTAB[i].count; j++) {

printf("%s ", PNTAB[i].name[j]);

}

printf("\n");

}

}

int main() {

// Example input

char code[][MAX\_CODE\_LEN] = {

"MACRO ADD &ARG1 &ARG2",

"LDA &ARG1",

"ADD &ARG2",

"STA RESULT",

"MEND",

"MACRO MULT &ARG1 &ARG2",

"MUL &ARG1",

"MUL &ARG2",

"STA RESULT",

"MEND",

"MACRO SUBTRACT &ARG1 &ARG2",

"LDA &ARG1",

"SUB &ARG2",

"STA RESULT",

"MEND",

"START",

"ADD NUM1 NUM2",

"MULT NUM3 NUM4",

"SUBTRACT NUM5 NUM6",

"END"

};

int line\_count = sizeof(code) / sizeof(code[0]);

// Process the input lines

for (int i = 0; i < line\_count; i++) {

char \*line = code[i];

char tokens[MAX\_PARAMS][MAX\_NAME\_LEN];

int token\_count = 0;

// Tokenize the input line

char \*token = strtok(line, " ");

while (token != NULL) {

strcpy(tokens[token\_count++], token);

token = strtok(NULL, " ");

}

if (token\_count == 0) continue;

// Macro definition start

if (strcmp(tokens[0], "MACRO") == 0) {

add\_macro(tokens[1], tokens + 2, token\_count - 2);

}

// Macro definition end

else if (strcmp(tokens[0], "MEND") == 0) {

end\_macro();

}

// Macro definition body

else if (find\_macro(tokens[0]) != -1) {

expand\_macro(tokens[0], tokens + 1);

} else {

printf("%s\n", line);

}

}

// Print tables

print\_tables();

return 0;

}

**Macros.txt-**

Macro Definition Table (MDT):

0 MACRO ADD &ARG1 &ARG2

1 LDA &ARG1

2 ADD &ARG2

3 STA RESULT

4 MEND

5 MACRO MULT &ARG1 &ARG2

6 MUL &ARG1

7 MUL &ARG2

8 STA RESULT

9 MEND

10 MACRO SUBTRACT &ARG1 &ARG2

11 LDA &ARG1

12 SUB &ARG2

13 STA RESULT

14 MEND

Macro Name Table (MNT):

Name MDT Index

ADD 0

MULT 5

SUBTRACT 10

Parameter Name Table (PNTAB):

Macro Name Parameters

ADD &ARG1 &ARG2

MULT &ARG1 &ARG2

SUBTRACT &ARG1 &ARG2

**5.EXPANSION**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX\_LINES 100

#define MAX\_PARAMS 10

#define MAX\_NAME\_LEN 50

#define MAX\_CODE\_LEN 100

// Structures for MDT, MNT, and PNTAB

struct MDT {

int index;

char definition[MAX\_CODE\_LEN];

} MDT[MAX\_LINES];

int mdt\_count = 0;

struct MNT {

char name[MAX\_NAME\_LEN];

int mdt\_index;

} MNT[MAX\_LINES];

int mnt\_count = 0;

struct PNTAB {

char name[MAX\_NAME\_LEN];

char parameters[MAX\_PARAMS][MAX\_NAME\_LEN];

int param\_count;

} PNTAB[MAX\_LINES];

int pntab\_count = 0;

// Function to read tables from file

void read\_tables() {

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

if (file == NULL) {

printf("Error: Could not open macros.txt\n");

return;

}

char line[MAX\_CODE\_LEN];

int section = 0;

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

if (strstr(line, "Macro Definition Table")) section = 1;

else if (strstr(line, "Macro Name Table")) section = 2;

else if (strstr(line, "Parameter Name Table")) section = 3;

else if (section == 1 && strstr(line, "MEND") == NULL) { // MDT

int index;

char definition[MAX\_CODE\_LEN];

sscanf(line, "%d %[^\n]s", &index, definition);

MDT[mdt\_count].index = index;

strcpy(MDT[mdt\_count++].definition, definition);

} else if (section == 2) { // MNT

struct MNT entry;

sscanf(line, "%s %d", entry.name, &entry.mdt\_index);

MNT[mnt\_count++] = entry;

} else if (section == 3) { // PNTAB

struct PNTAB entry;

sscanf(line, "%s %[^\n]s", entry.name, line);

entry.param\_count = 0;

char \*token = strtok(line, " ");

while (token != NULL) {

strcpy(entry.parameters[entry.param\_count++], token);

token = strtok(NULL, " ");

}

PNTAB[pntab\_count++] = entry;

}

}

fclose(file);

}

// Function to find macro index in MNT

int find\_macro(char \*name) {

for (int i = 0; i < mnt\_count; i++) {

if (strcmp(MNT[i].name, name) == 0) return MNT[i].mdt\_index;

}

return -1;

}

// Function to expand a macro and write to output file

void expand\_macro(FILE \*output, char \*name, char args[][MAX\_NAME\_LEN], int arg\_count) {

int mdt\_index = find\_macro(name);

if (mdt\_index == -1) return;

int param\_index = -1;

for (int i = 0; i < pntab\_count; i++) {

if (strcmp(PNTAB[i].name, name) == 0) {

param\_index = i;

break;

}

}

for (int i = mdt\_index + 1; i < mdt\_count && strstr(MDT[i].definition, "MEND") == NULL; i++) {

char line[MAX\_CODE\_LEN];

strcpy(line, MDT[i].definition);

for (int j = 0; j < PNTAB[param\_index].param\_count; j++) {

char \*pos = strstr(line, PNTAB[param\_index].parameters[j]);

if (pos != NULL) {

\*pos = '\0';

strcat(line, args[j]);

strcat(line, pos + strlen(PNTAB[param\_index].parameters[j]));

}

}

fprintf(output, "%s\n", line);

}

}

int main() {

read\_tables();

FILE \*output = fopen("expanded\_code.txt", "w");

if (output == NULL) {

printf("Error: Could not open output file\n");

return 1;

}

// Sample macro calls (for demonstration purposes)

char macro\_calls[][MAX\_CODE\_LEN] = {

"ADD NUM1 NUM2",

"MULT NUM3 NUM4",

"SUBTRACT NUM5 NUM6"

};

int num\_calls = sizeof(macro\_calls) / sizeof(macro\_calls[0]);

for (int i = 0; i < num\_calls; i++) {

char name[MAX\_NAME\_LEN], args[MAX\_PARAMS][MAX\_NAME\_LEN];

int arg\_count = 0;

char \*token = strtok(macro\_calls[i], " ");

strcpy(name, token);

token = strtok(NULL, " ");

while (token != NULL) {

strcpy(args[arg\_count++], token);

token = strtok(NULL, " ");

}

expand\_macro(output, name, args, arg\_count);

}

fclose(output);

printf("Macro expansion complete. Check 'expanded\_code.txt' for output.\n");

return 0;

}

**6.LEXICAL ANALYZER**

#include <ctype.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_LENGTH 100

#define MAX\_KEYWORDS 50

// Array to store keywords read from file

char keywords[MAX\_KEYWORDS][MAX\_LENGTH];

int keywordCount = 0;

// Function to load keywords from file

void loadKeywords(const char\* filename) {

FILE\* file = fopen(filename, "r");

if (file == NULL) {

printf("Error: Could not open keywords file.\n");

exit(1);

}

while (fgets(keywords[keywordCount], MAX\_LENGTH, file)) {

// Remove newline character from the end of each keyword

keywords[keywordCount][strcspn(keywords[keywordCount], "\n")] = 0;

keywordCount++;

}

fclose(file);

}

// Checks if a character is a delimiter

bool isDelimiter(char chr) {

return (chr == ' ' || chr == '+' || chr == '-' || chr == '\*'

|| chr == '/' || chr == ',' || chr == ';' || chr == '%'

|| chr == '>' || chr == '<' || chr == '=' || chr == '('

|| chr == ')' || chr == '[' || chr == ']' || chr == '{'

|| chr == '}');

}

// Checks if a character is an operator

bool isOperator(char chr) {

return (chr == '+' || chr == '-' || chr == '\*' || chr == '/'

|| chr == '>' || chr == '<' || chr == '=');

}

// Checks if a string is a valid identifier

bool isValidIdentifier(char\* str) {

return (str[0] != '0' && str[0] != '1' && str[0] != '2'

&& str[0] != '3' && str[0] != '4' && str[0] != '5'

&& str[0] != '6' && str[0] != '7' && str[0] != '8'

&& str[0] != '9' && !isDelimiter(str[0]));

}

// Checks if a string is a keyword

bool isKeyword(char\* str) {

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

if (strcmp(str, keywords[i]) == 0) {

return true;

}

}

return false;

}

// Checks if a string is an integer

bool isInteger(char\* str) {

if (str == NULL || \*str == '\0') {

return false;

}

int i = 0;

while (isdigit(str[i])) {

i++;

}

return str[i] == '\0';

}

// Trims a substring from a given string's start and end position

char\* getSubstring(char\* str, int start, int end) {

int subLength = end - start + 1;

char\* subStr = (char\*)malloc((subLength + 1) \* sizeof(char));

strncpy(subStr, str + start, subLength);

subStr[subLength] = '\0';

return subStr;

}

// Parses and analyzes the input

void lexicalAnalyzer(char\* input) {

int left = 0, right = 0;

int len = strlen(input);

while (right <= len && left <= right) {

if (!isDelimiter(input[right]))

right++;

if (isDelimiter(input[right]) && left == right) {

if (isOperator(input[right]))

printf("Token: Operator, Value: %c\n", input[right]);

right++;

left = right;

} else if (isDelimiter(input[right]) && left != right

|| (right == len && left != right)) {

char\* subStr = getSubstring(input, left, right - 1);

if (isKeyword(subStr))

printf("Token: Keyword, Value: %s\n", subStr);

else if (isInteger(subStr))

printf("Token: Integer, Value: %s\n", subStr);

else if (isValidIdentifier(subStr) && !isDelimiter(input[right - 1]))

printf("Token: Identifier, Value: %s\n", subStr);

else if (!isValidIdentifier(subStr) && !isDelimiter(input[right - 1]))

printf("Token: Unidentified, Value: %s\n", subStr);

free(subStr);

left = right;

}

}

}

// Main function

int main() {

// Load keywords from file

loadKeywords("keywords.txt");

// Get input expression from user

char input[MAX\_LENGTH];

printf("Enter an expression to analyze: ");

fgets(input, MAX\_LENGTH, stdin);

input[strcspn(input, "\n")] = 0; // Remove newline character from input

printf("\nAnalyzing Expression \"%s\":\n", input);

lexicalAnalyzer(input);

return 0;

}

**keywords.txt-**

auto

break

case

char

const

continue

default

do

double

else

enum

extern

float

for

goto

if

int

long

register

return

short

signed

sizeof

static

struct

switch

typedef

union

unsigned

void

volatile

while