1. Assembler Pass 1 :

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX\_SYMBOLS 100

#define MAX\_LITERALS 100

#define MAX\_POOLS 10

#define MAX\_CODE\_LINES 100

#define MAX\_OPCODES 50

typedef struct {

char symbol[10];

int address;

int opcode;

} Symbol;

int symbol\_opcode\_counter = 1;

typedef struct {

char literal[10];

int address;

int opcode;

} Literal;

typedef struct {

int start\_idx;

} Pool;

typedef struct {

char mnemonic[10];

char type[3];

int code;

} Opcode;

typedef struct {

char reg[5];

int code;

} Register;

Register regtab[4] = {

{"AREG", 1},

{"BREG", 2},

{"CREG", 3},

{"DREG", 4}

};

Symbol symtab[MAX\_SYMBOLS];

int symtab\_count = 0;

Literal littab[MAX\_LITERALS];

int littab\_count = 0;

Pool pooltab[MAX\_POOLS];

int pooltab\_count = 1;

char intermediate\_code[MAX\_CODE\_LINES][50];

int intermediate\_count = 0;

Opcode optab[MAX\_OPCODES];

int optab\_count = 0;

int location\_counter = 0;

int search\_register(char \*reg) {

int i;

for (i = 0; i < 4; i++) {

if (strcmp(regtab[i].reg, reg) == 0) {

return regtab[i].code;

}

}

return -1;

}

void add\_symbol(char \*symbol, int address) {

int idx = search\_symbol(symbol);

if (idx == -1) {

strcpy(symtab[symtab\_count].symbol, symbol);

symtab[symtab\_count].address = address;

symtab[symtab\_count].opcode = symbol\_opcode\_counter++;

symtab\_count++;

} else {

symtab[idx].address = address;

}

}

void add\_literal(char \*literal) {

strcpy(littab[littab\_count].literal, literal);

littab[littab\_count].address = -1;

littab[littab\_count].opcode = 1;

littab\_count++;

}

int search\_symbol(char \*symbol) {

int i;

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

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

return i;

}

}

return -1;

}

void load\_opcode\_table(char \*filename) {

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

if (fp == NULL) {

printf("Error: Cannot open opcode table file\n");

exit(1);

}

char mnemonic[10], type[3];

int code;

while (fscanf(fp, "%s %s %d", mnemonic, type, &code) != EOF) {

strcpy(optab[optab\_count].mnemonic, mnemonic);

strcpy(optab[optab\_count].type, type);

optab[optab\_count].code = code;

optab\_count++;

}

fclose(fp);

}

Opcode\* search\_opcode(char \*mnemonic) {

int i;

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

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

return &optab[i];

}

}

return NULL;

}

void trim\_comma(char \*str) {

int len = strlen(str);

if (len > 0 && str[len - 1] == ',') {

str[len - 1] = '\0';

}

}

void to\_uppercase(char \*str) {

int i = 0;

while (str[i]) {

if (str[i] >= 'a' && str[i] <= 'z') {

str[i] = str[i] - ('a' - 'A');

}

i++;

}

}

void process\_line(char \*line) {

char token1[10], token2[10], token3[10];

sscanf(line, "%s %s %s", token1, token2, token3);

int i;

trim\_comma(token2);

to\_uppercase(token2);

token2[strcspn(token2, "\n")] = 0;

token3[strcspn(token3, "\n")] = 0;

if (strcmp(token1, "START") == 0) {

location\_counter = atoi(token2);

sprintf(intermediate\_code[intermediate\_count++], "(AD,01) (C,%02d)", location\_counter);

} else if (strcmp(token1, "END") == 0) {

for (i = pooltab[pooltab\_count - 1].start\_idx; i < littab\_count; i++) {

if (littab[i].address == -1) {

littab[i].address = location\_counter;

sprintf(intermediate\_code[intermediate\_count++], "%02d (DL,02) (L,%d)", location\_counter, i + 1);

location\_counter++;

}

}

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

if (symtab[i].address == -1) {

sprintf(intermediate\_code[intermediate\_count++], "%02d (DL,01) (S,%s)", location\_counter, symtab[i].symbol);

location\_counter++;

}

}

} else if (strcmp(token1, "LTORG") == 0) {

for (i = pooltab[pooltab\_count - 1].start\_idx; i < littab\_count; i++) {

if (littab[i].address == -1) {

littab[i].address = location\_counter;

sprintf(intermediate\_code[intermediate\_count++], "%02d (DL,02) (L,%d)", location\_counter, i + 1);

location\_counter++;

}

}

pooltab[pooltab\_count++].start\_idx = littab\_count;

} else if (strcmp(token2, "DC") == 0) {

add\_symbol(token1, location\_counter);

sprintf(intermediate\_code[intermediate\_count++], "%02d (DL,01) (C,%s)", location\_counter, token3);

location\_counter++;

} else if (strcmp(token2, "DS") == 0) {

add\_symbol(token1, location\_counter);

int size = atoi(token3);

sprintf(intermediate\_code[intermediate\_count++], "%02d (DL,02) (C,%d)", location\_counter, size);

location\_counter += size;

} else {

Opcode \*opcode = search\_opcode(token1);

if (opcode != NULL) {

int reg\_code = search\_register(token2);

if (reg\_code != -1) {

if (token3[0] == '=') {

add\_literal(token3);

sprintf(intermediate\_code[intermediate\_count++], "%02d (IS,%02d) (R,%d) (L,%d)", location\_counter, opcode->code, reg\_code, littab\_count);

} else {

int symbol\_idx = search\_symbol(token3);

if (symbol\_idx != -1) {

sprintf(intermediate\_code[intermediate\_count++], "%02d (IS,%02d) (R,%d) (S,%d)", location\_counter, opcode->code, reg\_code, symbol\_idx);

} else {

printf("Error: Undefined symbol %s\n", token3);

}

}

location\_counter++;

} else {

printf("Error: Undefined register %s\n", token2);

}

} else if (token1[0] == '=') {

add\_literal(token1);

sprintf(intermediate\_code[intermediate\_count++], "%02d (DL,01) (C,%s)", location\_counter, token1);

location\_counter++;

} else {

printf("Error: Undefined operation %s\n", token1);

}

}

}

void display\_tables(char \*symbol\_file, char \*literal\_file, char \*pool\_file, char \*intermediate\_file) {

FILE \*symbol\_output = fopen(symbol\_file, "w");

FILE \*literal\_output = fopen(literal\_file, "w");

FILE \*pool\_output = fopen(pool\_file, "w");

FILE \*intermediate\_output = fopen(intermediate\_file, "w");

int i;

if (symbol\_output == NULL || literal\_output == NULL || pool\_output == NULL || intermediate\_output == NULL) {

printf("Error: Cannot open one or more output files\n");

return;

}

fprintf(symbol\_output, "Symbol Table:\n");

fprintf(symbol\_output, "Opcode \tSymbol\tAddress\n");

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

fprintf(symbol\_output, "%02d\t%s\t%d\n", symtab[i].opcode, symtab[i].symbol, symtab[i].address);

}

fprintf(literal\_output, "Literal Table:\n");

fprintf(literal\_output, "Opcode \tLiteral\tAddress\n");

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

fprintf(literal\_output, "%02d\t%s\t%d\n", littab[i].opcode, littab[i].literal, littab[i].address); // Updated format

}

fprintf(pool\_output, "Pool Table:\n");

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

fprintf(pool\_output, "%d\n", pooltab[i].start\_idx);

}

fprintf(intermediate\_output, "Intermediate Code:\n");

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

fprintf(intermediate\_output, "%s\n", intermediate\_code[i]);

}

fclose(symbol\_output);

fclose(literal\_output);

fclose(pool\_output);

fclose(intermediate\_output);

}

int main(int argc, char \*argv[]) {

char input\_file[100] = "input.txt";

char opcode\_file[100] = "opcode\_table.txt";

char symbol\_file[100] = "symbol\_table.txt";

char literal\_file[100] = "literal\_table.txt";

char pool\_file[100] = "pool\_table.txt";

char intermediate\_file[100] = "intermediate\_code.txt";

if (argc >= 2) {

strcpy(input\_file, argv[1]);

}

if (argc >= 3) {

strcpy(opcode\_file, argv[2]);

}

if (argc >= 4) {

strcpy(symbol\_file, argv[3]);

}

if (argc >= 5) {

strcpy(literal\_file, argv[4]);

}

if (argc >= 6) {

strcpy(pool\_file, argv[5]);

}

if (argc >= 7) {

strcpy(intermediate\_file, argv[6]);

}

load\_opcode\_table(opcode\_file);

FILE \*input = fopen(input\_file, "r");

if (input == NULL) {

printf("Error: Cannot open input file\n");

return 1;

}

char line[50];

while (fgets(line, sizeof(line), input) != NULL) {

process\_line(line);

}

fclose(input);

display\_tables(symbol\_file, literal\_file, pool\_file, intermediate\_file);

return 0;

}

2. Assembler Pass 2 :

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct {

int opcode;

int address;

} TableEntry;

int load\_literal\_table(const char \*filename, TableEntry \*\*table) {

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

if (!file) {

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

return 0;

}

int count = 0;

char line[100];

while (fgets(line, sizeof(line), file) != NULL) {

count++;

}

rewind(file);

\*table = (TableEntry \*)malloc(count \* sizeof(TableEntry));

int index = 0;

while (fgets(line, sizeof(line), file) != NULL) {

int opcode, address;

char literal[10];

if (sscanf(line, "%d %\*s %d", &opcode, &address) == 2) {

(\*table)[index].opcode = opcode;

(\*table)[index].address = address;

index++;

}

}

fclose(file);

return count;

}

int load\_symbol\_table(const char \*filename, TableEntry \*\*table) {

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

if (!file) {

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

return 0;

}

int count = 0;

char line[100];

while (fgets(line, sizeof(line), file) != NULL) {

count++;

}

rewind(file);

\*table = (TableEntry \*)malloc(count \* sizeof(TableEntry));

int index = 0;

while (fgets(line, sizeof(line), file) != NULL) {

int opcode, address;

char symbol[20];

if (sscanf(line, "%d %\*s %d", &opcode, &address) == 2) {

(\*table)[index].opcode = opcode;

(\*table)[index].address = address;

index++;

}

}

fclose(file);

return count;

}

int get\_address(TableEntry \*table, int size, int opcode) {

int i;

for (i = 0; i < size; i++) {

if (table[i].opcode == opcode) {

return table[i].address;

}

}

return -1;

}

void generate\_machine\_code(char \*intermediate\_file, char \*machine\_code\_file, TableEntry \*literal\_table, int literal\_count, TableEntry \*symbol\_table, int symbol\_count) {

FILE \*intermediate\_input = fopen(intermediate\_file, "r");

FILE \*machine\_code\_output = fopen(machine\_code\_file, "w");

if (intermediate\_input == NULL || machine\_code\_output == NULL) {

printf("Error: Cannot open intermediate or machine code file\n");

return;

}

char line[100];

int lc = 0;

while (fgets(line, sizeof(line), intermediate\_input) != NULL) {

int opcode, reg = -1, operand = -1, address = -1;

char type1[5], type2[5], reg\_type[5];

if (sscanf(line, "(AD,%d) (C,%d)", &opcode, &lc) == 2) {

fprintf(machine\_code\_output, "01 - %d\n", lc);

continue;

}

if (sscanf(line, "%d (DL,%d) (C,'%d')", &lc, &opcode, &operand) == 3) {

fprintf(machine\_code\_output, "%03d %02d - %d\n", lc, opcode, operand);

continue;

}

if (sscanf(line, "%d (DL,%d) (C,%d)", &lc, &opcode, &operand) == 3) {

fprintf(machine\_code\_output, "%03d %02d - %d\n", lc, opcode, operand);

continue;

}

if (sscanf(line, "%d (DL,%d) (L,%d)", &lc, &opcode, &operand) == 3) {

address = get\_address(literal\_table, literal\_count, operand);

if (address != -1) {

fprintf(machine\_code\_output, "%03d %02d - %d\n", lc, opcode, address);

}

continue;

}

if (sscanf(line, "%d (IS,%d) (R,%d) (%[^,],%d)", &lc, &opcode, &reg, type2, &operand) == 5) {

if (strcmp(type2, "L") == 0) {

address = get\_address(literal\_table, literal\_count, operand);

} else if (strcmp(type2, "S") == 0) {

address = get\_address(symbol\_table, symbol\_count, operand);

}

if (address != -1) {

fprintf(machine\_code\_output, "%03d %02d %01d %02d\n", lc, opcode, reg, address);

}

continue;

}

if (sscanf(line, "%d (IS,%d) (%[^,],%d)", &lc, &opcode, type2, &operand) == 4) {

if (strcmp(type2, "L") == 0) {

address = get\_address(literal\_table, literal\_count, operand);

} else if (strcmp(type2, "S") == 0) {

address = get\_address(symbol\_table, symbol\_count, operand);

}

if (address != -1) {

fprintf(machine\_code\_output, "%03d %02d - %02d\n", lc, opcode, address);

}

continue;

}

if (sscanf(line, "%d (AD,%d) %d", &lc, &opcode, &operand) == 3) {

fprintf(machine\_code\_output, "%03d %02d %03d\n", lc, opcode, operand);

continue;

}

printf("Warning: Unable to parse line: %s", line);

}

fclose(intermediate\_input);

fclose(machine\_code\_output);

}

int main(int argc, char \*argv[]) {

char intermediate\_file[100] = "intermediate\_code.txt";

char machine\_code\_file[100] = " machine\_code.txt";

TableEntry \*literal\_table = NULL;

TableEntry \*symbol\_table = NULL;

int literal\_count = load\_literal\_table("literal\_table.txt", &literal\_table);

int symbol\_count = load\_symbol\_table("symbol\_table.txt", &symbol\_table);

if (argc >= 2) {

strcpy(intermediate\_file, argv[1]);

}

if (argc >= 3) {

strcpy(machine\_code\_file, argv[2]);

}

generate\_machine\_code(intermediate\_file, machine\_code\_file, literal\_table, literal\_count, symbol\_table, symbol\_count);

free(literal\_table);

free(symbol\_table);

printf("Machine code generated successfully and saved to %s\n", machine\_code\_file);

return 0;

}

3. Macro Pass 1 (MDT, MNT, ALT, Intermediate code) :

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_LINE\_LENGTH 256

#define MAX\_MDT\_LINES 100

#define MAX\_MNT\_LINES 100

void writeMDTFile(char mdt[MAX\_MDT\_LINES][MAX\_LINE\_LENGTH], int mdtCount) {

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

int i;

if (outFile == NULL) {

printf("Unable to open MDT file for writing.\n");

return;

}

for (i = 0; i < mdtCount; i++) {

fprintf(outFile, "%s", mdt[i]);

}

fclose(outFile);

printf("MDT written to 2MDT.txt\n");

}

void writeMNTFile(char mnt[MAX\_MNT\_LINES][MAX\_LINE\_LENGTH], int mntCount) {

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

int i;

if (outFile == NULL) {

printf("Unable to open MNT file for writing.\n");

return;

}

for (i = 0; i < mntCount; i++) {

fprintf(outFile, "%s", mnt[i]);

}

fclose(outFile);

printf("MNT written to 2MNT.txt\n");

}

void writeIntermediateFile(char lines[][MAX\_LINE\_LENGTH], int startLine, int totalLines) {

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

int i;

if (intermediateFile == NULL) {

printf("Unable to open Intermediate file for writing.\n");

return;

}

for (i = startLine; i < totalLines; i++) {

fprintf(intermediateFile, "%s", lines[i]);

}

fclose(intermediateFile);

printf("Intermediate code written to 2Intermediatecodemacro.txt\n");

}

void writeALTFile(char alt[MAX\_MNT\_LINES][MAX\_LINE\_LENGTH], int altCount) {

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

int i;

if (altFile == NULL) {

printf("Unable to open ALT file for writing.\n");

return;

}

for (i = 0; i < altCount; i++) {

fprintf(altFile, "%s", alt[i]);

}

fclose(altFile);

printf("ALT written to 2ALT.txt\n");

}

void processMacros(const char \*inputFileName) {

FILE \*inFile = fopen(inputFileName, "r");

if (inFile == NULL) {

printf("Unable to open input file.\n");

return;

}

char line[MAX\_LINE\_LENGTH];

char mdt[MAX\_MDT\_LINES][MAX\_LINE\_LENGTH];

char mnt[MAX\_MNT\_LINES][MAX\_LINE\_LENGTH];

char alt[MAX\_MNT\_LINES][MAX\_LINE\_LENGTH];

char allLines[MAX\_MNT\_LINES][MAX\_LINE\_LENGTH];

int mdtIndex = 1;

int mdtCount = 0;

int mntCount = 0;

int altCount = 0;

int lineCount = 0;

int lastMendLine = -1;

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

snprintf(allLines[lineCount++], MAX\_LINE\_LENGTH, "%s", line);

if (strstr(line, "MACRO") != NULL) {

int argumentIndex = 1;

if (fgets(line, sizeof(line), inFile)) {

snprintf(allLines[lineCount++], MAX\_LINE\_LENGTH, "%s", line);

char macroName[MAX\_LINE\_LENGTH];

sscanf(line, "%s", macroName);

snprintf(mnt[mntCount++], MAX\_LINE\_LENGTH, "%d %s\n", mdtIndex, macroName);

snprintf(mdt[mdtCount++], MAX\_LINE\_LENGTH, "%d %s", mdtIndex++, line);

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

snprintf(allLines[lineCount++], MAX\_LINE\_LENGTH, "%s", line);

if (strstr(line, "MEND") != NULL) {

snprintf(mdt[mdtCount++], MAX\_LINE\_LENGTH, "%d %s", mdtIndex++, line);

lastMendLine = lineCount;

break;

}

char tempLine[MAX\_LINE\_LENGTH];

strcpy(tempLine, line);

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

while (token != NULL) {

if (token[0] == '&') {

char \*equalPos = strchr(token, '=');

if (equalPos != NULL) {

\*equalPos = '\0';

char \*paramName = token;

char \*paramValue = equalPos + 1;

snprintf(alt[altCount++], MAX\_LINE\_LENGTH, "%d %s = %s\n", argumentIndex++, paramName, paramValue);

char \*pos = strstr(line, paramName);

if (pos != NULL) {

strncpy(pos, paramValue, strlen(paramValue));

}

} else {

snprintf(alt[altCount++], MAX\_LINE\_LENGTH, "%d %s\n", argumentIndex++, token);

char indexedParam[MAX\_LINE\_LENGTH];

snprintf(indexedParam, sizeof(indexedParam), "#%d", argumentIndex - 1);

char \*pos = strstr(line, token);

if (pos != NULL) {

strncpy(pos, indexedParam, strlen(indexedParam));

}

}

}

token = strtok(NULL, " ,");

}

snprintf(mdt[mdtCount++], MAX\_LINE\_LENGTH, "%d %s", mdtIndex++, line);

}

}

}

}

fclose(inFile);

writeMDTFile(mdt, mdtCount);

writeMNTFile(mnt, mntCount);

writeALTFile(alt, altCount);

if (lastMendLine != -1 && lastMendLine < lineCount) {

writeIntermediateFile(allLines, lastMendLine, lineCount);

} else {

printf("No intermediate code found after the last MEND.\n");

}

}

int main() {

const char \*inputFileName = "macro.txt";

processMacros(inputFileName);

return 0;

}

4. Macro Pass 2 (Expanded code) :

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_LINE\_LENGTH 256

#define MAX\_MDT\_LINES 100

#define MAX\_MNT\_LINES 100

#define MAX\_ALT\_LINES 10

#define MAX\_EXPANDED\_LINES 500

void readFileToArray(const char \*fileName, char array[][MAX\_LINE\_LENGTH], int \*count) {

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

if (file == NULL) {

printf("Unable to open file: %s\n", fileName);

return;

}

char line[MAX\_LINE\_LENGTH];

\*count = 0;

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

strcpy(array[(\*count)++], line);

}

fclose(file);

}

void writeExpandedCodeFile(char expanded[MAX\_EXPANDED\_LINES][MAX\_LINE\_LENGTH], int expandedCount) {

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

int i;

if (outFile == NULL) {

printf("Unable to open ExpandedCode file for writing.\n");

return;

}

for (i = 0; i < expandedCount; i++) {

fprintf(outFile, "%s", expanded[i]);

}

fclose(outFile);

printf("Expanded code written to 2ExpandedCode.txt\n");

}

int findMacroDefinition(char mnt[MAX\_MNT\_LINES][MAX\_LINE\_LENGTH], int mntCount, const char \*macroName) {

int i;

for (i = 0; i < mntCount; i++) {

char name[MAX\_LINE\_LENGTH];

int mdtIndex;

sscanf(mnt[i], "%d %s", &mdtIndex, name);

if (strcmp(name, macroName) == 0) {

return mdtIndex;

}

}

return -1;

}

void substituteArguments(char \*line, char actualArgs[MAX\_ALT\_LINES][MAX\_LINE\_LENGTH], int actualArgCount) {

char placeholder[MAX\_LINE\_LENGTH];

int i;

for (i = 0; i < actualArgCount; i++) {

sprintf(placeholder, "#%d", i + 1);

char \*pos = strstr(line, placeholder);

while (pos != NULL) {

char temp[MAX\_LINE\_LENGTH];

strncpy(temp, line, pos - line);

temp[pos - line] = '\0';

strcat(temp, actualArgs[i]);

strcat(temp, pos + strlen(placeholder));

strcpy(line, temp);

pos = strstr(line, placeholder);

}

}

}

void expandMacro(char mdt[MAX\_MDT\_LINES][MAX\_LINE\_LENGTH], int mdtIndex, char actualArgs[MAX\_ALT\_LINES][MAX\_LINE\_LENGTH], int actualArgCount, char expanded[MAX\_EXPANDED\_LINES][MAX\_LINE\_LENGTH], int \*expandedCount) {

while (strstr(mdt[mdtIndex], "MEND") == NULL) {

char line[MAX\_LINE\_LENGTH];

strcpy(line, mdt[mdtIndex]);

char \*instructionStart = strchr(line, ' ');

if (instructionStart != NULL) {

strcpy(line, instructionStart + 1);

}

substituteArguments(line, actualArgs, actualArgCount);

strcpy(expanded[(\*expandedCount)++], line);

mdtIndex++;

}

}

void processPass2(const char \*intermediateFileName, char mnt[MAX\_MNT\_LINES][MAX\_LINE\_LENGTH], int mntCount, char mdt[MAX\_MDT\_LINES][MAX\_LINE\_LENGTH]) {

FILE \*intermediateFile = fopen(intermediateFileName, "r");

if (intermediateFile == NULL) {

printf("Unable to open intermediate file.\n");

return;

}

char expanded[MAX\_EXPANDED\_LINES][MAX\_LINE\_LENGTH];

int expandedCount = 0;

char line[MAX\_LINE\_LENGTH];

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

char macroName[MAX\_LINE\_LENGTH];

int foundMacro = 0;

sscanf(line, "%s", macroName);

int mdtIndex = findMacroDefinition(mnt, mntCount, macroName);

if (mdtIndex != -1) {

foundMacro = 1;

char \*start = strchr(line, ' ');

char actualArgs[MAX\_ALT\_LINES][MAX\_LINE\_LENGTH];

int actualArgCount = 0;

if (start != NULL) {

start++;

char \*token = strtok(start, " ,\n");

while (token != NULL) {

strcpy(actualArgs[actualArgCount++], token);

token = strtok(NULL, " ,\n");

}

}

expandMacro(mdt, mdtIndex, actualArgs, actualArgCount, expanded, &expandedCount);

}

if (!foundMacro) {

strcpy(expanded[expandedCount++], line);

}

}

fclose(intermediateFile);

writeExpandedCodeFile(expanded, expandedCount);

}

int main() {

char mnt[MAX\_MNT\_LINES][MAX\_LINE\_LENGTH];

char mdt[MAX\_MDT\_LINES][MAX\_LINE\_LENGTH];

int mntCount, mdtCount;

readFileToArray("MNT.txt", mnt, &mntCount);

readFileToArray("MDT.txt", mdt, &mdtCount);

processPass2("Intermediatecodemacro.txt", mnt, mntCount, mdt);

return 0;

}

5. Lexical analyser

#include <ctype.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_LENGTH 100

bool isDelimiter(char chr)

{

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

|| chr == '\*' || chr == '/' || chr == ','

|| chr == ';' || chr == '%' || chr == '>'

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

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

|| chr == '{' || chr == '}');

}

bool isOperator(char chr)

{

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

|| chr == '/' || chr == '>' || chr == '<'

|| chr == '=');

}

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]));

}

bool isKeyword(char\* str)

{

int i;

const char\* keywords[]

= { "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" };

for (i = 0;

i < sizeof(keywords) / sizeof(keywords[0]); i++) {

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

return true;

}

}

return false;

}

bool isInteger(char\* str)

{

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

return false;

}

int i = 0;

while (isdigit(str[i])) {

i++;

}

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

}

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

{

int length = strlen(str);

int subLength = end - start + 1;

char\* subStr

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

strncpy(subStr, str + start, subLength);

subStr[subLength] = '\0';

return subStr;

}

int 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);

left = right;

}

}

return 0;

}

int main() {

char lex\_input[MAX\_LENGTH];

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

fgets(lex\_input, MAX\_LENGTH, stdin);

lex\_input[strcspn(lex\_input, "\n")] = '\0';

printf("For Expression \"%s\":\n", lex\_input);

lexicalAnalyzer(lex\_input);

return 0;

}

6. Relocation table, link table

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX\_LINES 100

#define MAX\_LINE\_LENGTH 100

#define MAX\_SYMBOLS 100

#define MAX\_ADDRESS\_SENSITIVE\_INSTRUCTIONS 100

typedef struct {

int original\_address;

int relocation\_factor;

int relocated\_address;

int is\_address\_sensitive;

char instruction[MAX\_LINE\_LENGTH];

} RelocationEntry;

typedef struct {

char name[MAX\_LINE\_LENGTH];

int address;

int linked\_address;

char type[3];

} Symbol;

void extract\_symbols(char assembly\_code[MAX\_LINES][MAX\_LINE\_LENGTH], int line\_count, Symbol symbol\_table[MAX\_SYMBOLS], int \*symbol\_count, char address\_sensitive\_instructions[MAX\_ADDRESS\_SENSITIVE\_INSTRUCTIONS][MAX\_LINE\_LENGTH], int \*address\_sensitive\_count) {

int address = 100;

int i;

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

char \*line = assembly\_code[i];

if (strlen(line) == 0 || line[0] == ';') {

continue;

}

if (strncmp(line, "START", 5) == 0) {

sscanf(line, "START %d", &address);

continue;

}

if (strncmp(line, "ENTRY", 5) == 0) {

char symbol\_name[MAX\_LINE\_LENGTH];

sscanf(line, "ENTRY %s", symbol\_name);

strncpy(symbol\_table[\*symbol\_count].name, symbol\_name, MAX\_LINE\_LENGTH);

symbol\_table[\*symbol\_count].address = address;

strncpy(symbol\_table[\*symbol\_count].type, "PD", 3);

(\*symbol\_count)++;

continue;

}

else if (strstr(line, "DS") || strstr(line, "DC")) {

char symbol\_name[MAX\_LINE\_LENGTH];

sscanf(line, "%s", symbol\_name);

strncpy(symbol\_table[\*symbol\_count].name, symbol\_name, MAX\_LINE\_LENGTH);

symbol\_table[\*symbol\_count].address = address;

strncpy(symbol\_table[\*symbol\_count].type, "PD", 3);

(\*symbol\_count)++;

}

if (strstr(line, "LOAD") || strstr(line, "MOVEM") || strstr(line, "SUB") || strstr(line, "CMP") || strstr(line, "ADD")) {

strncpy(address\_sensitive\_instructions[\*address\_sensitive\_count], line, MAX\_LINE\_LENGTH);

(\*address\_sensitive\_count)++;

}

address++;

}

}

void calculate\_relocation(int link\_origin, int start\_origin, char assembly\_code[MAX\_LINES][MAX\_LINE\_LENGTH], int line\_count, Symbol symbol\_table[MAX\_SYMBOLS], int symbol\_count, char address\_sensitive\_instructions[MAX\_ADDRESS\_SENSITIVE\_INSTRUCTIONS][MAX\_LINE\_LENGTH], int address\_sensitive\_count, FILE \*output\_file) {

int relocation\_factor = link\_origin - start\_origin;

int address = start\_origin;

RelocationEntry relocation\_table[MAX\_LINES];

int entry\_count = 0;

int i, j;

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

char \*line = assembly\_code[i];

if (strlen(line) == 0 || line[0] == ';') {

continue;

}

if (strncmp(line, "START", 5) == 0) {

sscanf(line, "START %d", &address);

continue;

}

int is\_address\_sensitive = 0;

for (j = 0; j < address\_sensitive\_count; j++) {

if (strstr(line, address\_sensitive\_instructions[j]) != NULL) {

is\_address\_sensitive = 1;

break;

}

}

if (is\_address\_sensitive) {

relocation\_table[entry\_count].original\_address = address;

relocation\_table[entry\_count].relocation\_factor = relocation\_factor;

relocation\_table[entry\_count].relocated\_address = address + relocation\_factor;

relocation\_table[entry\_count].is\_address\_sensitive = is\_address\_sensitive;

entry\_count++;

}

address++;

}

fprintf(output\_file, "%-20s\n", "Relocated Address");

fprintf(output\_file, "%s\n", "--------------------");

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

fprintf(output\_file, "%-20d\n", relocation\_table[j].relocated\_address);

}

}

int main() {

int link\_origin, start\_origin = 100;

char assembly\_code[MAX\_LINES][MAX\_LINE\_LENGTH];

Symbol symbol\_table[MAX\_SYMBOLS];

char address\_sensitive\_instructions[MAX\_ADDRESS\_SENSITIVE\_INSTRUCTIONS][MAX\_LINE\_LENGTH];

int line\_count = 0, symbol\_count = 0, address\_sensitive\_count = 0;

int j;

printf("Enter the link origin: ");

scanf("%d", &link\_origin);

getchar();

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

if (file == NULL) {

perror("Error opening file");

return EXIT\_FAILURE;

}

while (line\_count < MAX\_LINES && fgets(assembly\_code[line\_count], MAX\_LINE\_LENGTH, file)) {

assembly\_code[line\_count][strcspn(assembly\_code[line\_count], "\n")] = 0;

line\_count++;

}

fclose(file);

extract\_symbols(assembly\_code, line\_count, symbol\_table, &symbol\_count, address\_sensitive\_instructions, &address\_sensitive\_count);

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

symbol\_table[j].linked\_address = symbol\_table[j].address + (link\_origin - start\_origin);

}

FILE \*symbol\_file = fopen("linktable.txt", "w");

if (symbol\_file == NULL) {

perror("Error opening symbol output file");

return EXIT\_FAILURE;

}

fprintf(symbol\_file, "%-20s %-20s %-20s\n", "Symbol", "Address", "Type");

fprintf(symbol\_file, "%s\n", "----------------------------------------------");

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

fprintf(symbol\_file, "%-20s %-20d %-20s\n", symbol\_table[j].name, symbol\_table[j].address, symbol\_table[j].type);

}

fclose(symbol\_file);

FILE \*output\_file = fopen("relocationtable.txt", "w");

if (output\_file == NULL) {

perror("Error opening output file");

return EXIT\_FAILURE;

}

calculate\_relocation(link\_origin, start\_origin, assembly\_code, line\_count, symbol\_table, symbol\_count, address\_sensitive\_instructions, address\_sensitive\_count, output\_file);

fclose(output\_file);

return 0;

}

7. Object module

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_LINE\_LENGTH 100

int is\_empty\_line(const char \*line) {

int i;

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

if (!isspace(line[i])) {

return 0;

}

}

return 1;

}

int find\_start\_address(FILE \*input\_file) {

char line[MAX\_LINE\_LENGTH];

int start\_address = -1;

while (fgets(line, MAX\_LINE\_LENGTH, input\_file)) {

if (strstr(line, "START") != NULL) {

sscanf(line, "%\*s %d", &start\_address);

break;

}

}

return start\_address;

}

int count\_code\_lines(FILE \*input\_file) {

char line[MAX\_LINE\_LENGTH];

int count = 0;

while (fgets(line, MAX\_LINE\_LENGTH, input\_file)) {

if (!is\_empty\_line(line)) {

count++;

}

}

return count;

}

void append\_relocation\_table(const char \*relocation\_filename, FILE \*output\_file) {

FILE \*relocation\_file = fopen(relocation\_filename, "r");

if (!relocation\_file) {

perror("Error opening relocation table file");

exit(EXIT\_FAILURE);

}

char line[MAX\_LINE\_LENGTH];

fprintf(output\_file, "\nRelocation Table:\n");

while (fgets(line, MAX\_LINE\_LENGTH, relocation\_file)) {

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

}

fclose(relocation\_file);

}

void append\_link\_table(const char \*symbol\_table\_filename, FILE \*output\_file) {

FILE \*symbol\_table\_file = fopen(symbol\_table\_filename, "r");

if (!symbol\_table\_file) {

perror("Error opening symbol table file");

exit(EXIT\_FAILURE);

}

char line[MAX\_LINE\_LENGTH];

fprintf(output\_file, "\nLink Table:\n");

while (fgets(line, MAX\_LINE\_LENGTH, symbol\_table\_file)) {

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

}

fclose(symbol\_table\_file);

}

void append\_machine\_code(const char \*machine\_code\_filename, FILE \*output\_file) {

FILE \*machine\_code\_file = fopen(machine\_code\_filename, "r");

if (!machine\_code\_file) {

perror("Error opening machine code file");

exit(EXIT\_FAILURE);

}

char line[MAX\_LINE\_LENGTH];

fprintf(output\_file, "\nMachine Code:\n");

while (fgets(line, MAX\_LINE\_LENGTH, machine\_code\_file)) {

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

}

fclose(machine\_code\_file);

}

int main() {

FILE \*input\_file = fopen("inputfile.txt", "r");

FILE \*output\_file = fopen("object\_module.txt", "w");

if (!input\_file || !output\_file) {

perror("Error opening files");

exit(EXIT\_FAILURE);

}

int start\_address = find\_start\_address(input\_file);

if (start\_address == -1) {

fprintf(stderr, "START address not found in om.txt\n");

exit(EXIT\_FAILURE);

}

rewind(input\_file);

int code\_size = count\_code\_lines(input\_file);

int relocation\_factor = 100;

int adjusted\_start\_address = start\_address + relocation\_factor;

fprintf(output\_file, "Header:\n");

fprintf(output\_file, "Translated Address: %d\n", start\_address);

fprintf(output\_file, "Code Size: %d\n", code\_size);

fprintf(output\_file, "Start Address: %d\n", adjusted\_start\_address);

append\_machine\_code("machinecode.txt", output\_file);

append\_relocation\_table("relocation\_table.txt", output\_file);

append\_link\_table("link\_table.txt", output\_file);

fclose(input\_file);

fclose(output\_file);

printf("Object module with header, machine code, relocation table, and link table created successfully in object\_module.txt.\n");

return 0;

}