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;

}
```