شرح كود `assembler.c`

هذا المستند يقدم شرحًا مفصلاً لكل سطر في كود الـ assembler المكتوب بلغة C.

الأجزاء الرئيسية والتعريفات الأولية

Plain Text

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

- هذه الأسطر هي توجيهات `include` لاستيراد المكتبات القياسية في لغة C:
- `<stdio.h>` : للعمليات المدخلات والمخرجات (مثل `printf` , `fprintf` , `fopen` , `fclose`).
- `<string.h>` : للتعامل مع السلاسل النصية (مثل `strlen` , `strcmp` , `strchr` , `strdup` , `strtok`).
- `<stdlib.h>` : للوظائف العامة (مثل `atoi` , `strtol` , `malloc` , `free`).
- `<ctype.h>` : لوظائف فحص وتصنيف الأحرف (مثل `isdigit` , `isspace`).
- `<stdbool.h>` : لتعريف النوع `bool` والقيم `true` و `false` .

Plain Text

```
// Simplified ARM instruction set and their corresponding opcodes/formats
// This is a highly simplified representation for demonstration purposes.
// A real ARM assembler would be much more complex.
```

- هذه التعليقات توضح أن الـ assembler هو نسخة مبسطة جدًا لأغراض العرض التوضيحي، وأن الـ assembler الحقيقي لـ ARM سيكون أكثر تعقيدًا.

Plain Text

```
// Instruction types for operand parsing
#define TYPE_DATA_PROCESSING_IMM 0 // Rd, Rn, #imm
```

```
#define TYPE_DATA_PROCESSING_REG 1 // Rd, Rn, Rm
#define TYPE_BRANCH              2 // label
#define TYPE_SWI                 3 // #imm
#define TYPE_MOV_IMM             4 // Rd, #imm
#define TYPE_MOV_REG             5 // Rd, Rm
#define TYPE_MUL                 6 // Rd, Rm, Rs
#define TYPE_CMP_REG             7 // Rn, Rm
#define TYPE_SHIFT_IMM           8 // Rd, Rm, #imm
```

- هذه الأسطر تعرف ثوابت `define` لتصنيف أنواع التعليمات بناءً على كيفية تحليل وتشفير معاملاتها. هذا يساعد في توجيه منطق التحويل في الدالة الرئيسية.

- `TYPE_DATA_PROCESSING_IMM` : لتعليمات معالجة البيانات التي تأخذ سجلين وقيمة فورية (مثال: `ADD R2, R0, #3`).
- `TYPE_DATA_PROCESSING_REG` : لتعليمات معالجة البيانات التي تأخذ ثلاثة سجلات (مثال: `ADD R2, R0, R1`).
- `TYPE_BRANCH` : لتعليمات القفز التي تأخذ تسمية (label) كمعامل (مثال: `B label`).
- `TYPE_SWI` : لتعليمات استدعاء النظام التي تأخذ قيمة فورية (مثال: `SWI #0`).
- `TYPE_MOV_IMM` : لتعليمات `MOV` التي تأخذ سجل وقيمة فورية.
- `TYPE_MOV_REG` : لتعليمات `MOV` التي تأخذ سجلين.
- `TYPE_MUL` : لتعليمات `MUL` التي تأخذ ثلاثة سجلات.
- `TYPE_CMP_REG` : لتعليمات `CMP` (يمكن أن تكون سجل وقيمة فورية أيضًا) التي تأخذ سجلين.
- `TYPE_SHIFT_IMM` : لتعليمات الإزاحة (LSL, LSR) التي تأخذ سجلين وقيمة فورية.

Plain Text

```
typedef struct {
    char *mnemonic;
    unsigned int opcode_base; // Base opcode for the instruction
    int type; // Defines how operands are parsed and encoded
} Instruction;
```

- هذا تعريف لهيكل `Instruction` ، والذي يستخدم لتخزين معلومات حول كل تعليمة ARM Assembly يدعمها الـ assembler:

- `mnemonic` : مؤشر لسلسلة نصية تمثل اسم التعليمة (مثل
- `opcode_base` : قيمة سداسية عشرية تمثل الجزء الأساسي من الـ machine code لهذه التعليمة. يتم دمج هذه القيمة مع ترميز المعاملات لتكوين الـ machine code النهائي.
- `type` : نوع التعليمة، وهو أحد الثوابت المعرفة أعلاه (`TYPE_DATA_PROCESSING_IMM` , `TYPE_DATA_PROCESSING_REG` , إلخ)، ويحدد كيفية معالجة المعاملات.

Plain Text

```
Instruction instructions[] = {
    {"MOV", 0xE1A00000, TYPE_MOV_REG},       // MOV Rd, Rm (simplified, also
handles #imm via TYPE_MOV_IMM)
    {"ADD", 0xE0800000, TYPE_DATA_PROCESSING_REG}, // ADD Rd, Rn, Rm
    {"SUB", 0xE0400000, TYPE_DATA_PROCESSING_REG}, // SUB Rd, Rn, Rm
    {"CMP", 0xE1500000, TYPE_CMP_REG},       // CMP Rn, Rm or CMP Rn, #imm
    {"AND", 0xE0000000, TYPE_DATA_PROCESSING_REG}, // AND Rd, Rn, Rm
    {"ORR", 0xE1800000, TYPE_DATA_PROCESSING_REG}, // ORR Rd, Rn, Rm
    {"EOR", 0xE0200000, TYPE_DATA_PROCESSING_REG}, // EOR Rd, Rn, Rm
    {"LSL", 0xE1A00000, TYPE_SHIFT_IMM}, // LSL Rd, Rm, #imm
    {"LSR", 0xE1A00000, TYPE_SHIFT_IMM}, // LSR Rd, Rm, #imm
    {"MUL", 0xE0000090, TYPE_MUL},           // MUL Rd, Rm, Rs
    {"BGE", 0xDA000000, TYPE_BRANCH},       // BGE target
    {"BLT", 0xBA000000, TYPE_BRANCH},       // BLT target
    {"B",   0xEA000000, TYPE_BRANCH},       // B target
    {"SWI", 0xEF000000, TYPE_SWI},           // SWI #imm
    {NULL, 0, 0} // Sentinel
};
```

- هذا هو مصفوفة `instructions` ، وهي قائمة بجميع تعليمات ARM Assembly التي يدعمها الـ assembler. كل عنصر في المصفوفة هو كائن `Instruction` يحدد اسم التعليمة، الـ opcode الأساسي لها، ونوعها (الذي يحدد كيفية تحليل معاملاتها).
- `NULL, 0, 0` : هو عنصر حارس (sentinel) يشير إلى نهاية المصفوفة.

Plain Text

```
typedef struct {
    char *label;
    int address;
} Symbol;

#define MAX_SYMBOLS 100
```

```
Symbol symbol_table[MAX_SYMBOLS];
int symbol_count = 0;
```

- هذا تعريف لهيكل `Symbol` ، والذي يستخدم لتخزين معلومات حول التسميات (labels) الموجودة في كود Assembly:
  - `label` : مؤشر لسلسلة نصية تمثل اسم التسمية.
  - `address` : العنوان المقابل لهذه التسمية في الذاكرة (أو في كود الـ machine code).
- `MAX_SYMBOLS` : ثابت يحدد الحد الأقصى لعدد التسميات التي يمكن تخزينها في جدول الرموز.
- `symbol_table` : مصفوفة من هياكل `Symbol` لتخزين جميع التسميات التي يتم العثور عليها في كود Assembly.
- `symbol_count` : متغير يتتبع العدد الحالي للتسميات المخزنة في `symbol_table` .

الدوال المساعدة (Helper Functions)

Plain Text

```
void add_symbol(char *label, int address) {
    if (symbol_count < MAX_SYMBOLS) {
        symbol_table[symbol_count].label = strdup(label);
        symbol_table[symbol_count].address = address;
        symbol_count++;
    }
}
```

- `add_symbol` : هذه الدالة تضيف تسمية (label) وعنوانها المقابل إلى `symbol_table` .
  - `if (symbol_count < MAX_SYMBOLS)` : تتحقق مما إذا كان هناك مساحة كافية في جدول الرموز قبل الإضافة.
  - `symbol_table[symbol_count].label = strdup(label);` : تنسخ اسم التسمية إلى جدول الرموز. `strdup` تقوم بتخصيص ذاكرة جديدة للسلسلة النصية ونسخها، وهذا مهم لتجنب مشاكل المؤشرات بعد انتهاء صلاحية `label` .
  - `symbol_table[symbol_count].address = address;` : تخزن العنوان المقابل للتسمية.

- `symbol_count++;` : تزيد عداد التسميات المخزنة.

Plain Text

```
int get_symbol_address(char *label) {
    for (int i = 0; i < symbol_count; i++) {
        if (strcmp(symbol_table[i].label, label) == 0) {
            return symbol_table[i].address;
        }
    }
    return -1; // Not found
}
```

- `get_symbol_address` : هذه الدالة تبحث عن تسمية معينة في `symbol_table` وتعيد عنوانها.
  - `for (int i = 0; i < symbol_count; i++)` : تمر على جميع التسميات المخزنة في الجدول.
  - `if (strcmp(symbol_table[i].label, label) == 0)` : تقارن اسم التسمية الحالي في الجدول بالاسم المطلوب. `strcmp` تعيد 0 إذا كانت السلسلتان متطابقتين.
  - `return symbol_table[i].address;` : إذا تم العثور على التسمية، يتم إرجاع عنوانها.
  - `return -1;` : إذا لم يتم العثور على التسمية بعد البحث في الجدول بأكمله، يتم إرجاع -1 للإشارة إلى ذلك.

Plain Text

```
int get_register_num(char *reg_str) {
    if (reg_str != NULL && (reg_str[0] == 'R' || reg_str[0] == 'r')) {
        return atoi(reg_str + 1);
    }
    return -1; // Invalid register
}
```

- `get_register_num` : هذه الدالة تحول اسم سجل (مثل `R0` , `R15`) (0, 15) إلى رقمه المقابل.
  - `if (reg_str != NULL && (reg_str[0] == 'R' || reg_str[0] == 'r'))` : تتحقق مما إذا كانت السلسلة النصية ليست فارغة وتبدأ بحرف 'R' أو 'r'.
  - `return atoi(reg_str + 1);` : إذا كان الشرط صحيحًا، يتم تخطي الحرف الأول ('R' أو 'r') وتحويل الجزء المتبقي من السلسلة إلى عدد صحيح باستخدام `atoi` .

- `return -1;` : إذا لم يكن اسم السجل صالحًا، يتم إرجاع -1.

Plain Text

```
int parse_immediate(char *imm_str) {
    if (imm_str != NULL && imm_str[0] == '#') {
        return atoi(imm_str + 1);
    }
    return -1; // Not an immediate value with #
}
```

- `parse_immediate` : هذه الدالة تحلل قيمة فورية (immediate value) (مثل `10#`) `#` تبدأ بعلامة.
  - `if (imm_str != NULL && imm_str[0] == '#')` : تتحقق مما إذا كانت السلسلة النصية ليست فارغة وتبدأ بعلامة `#` .
  - `return atoi(imm_str + 1);` : إذا كان الشرط صحيحًا، يتم تخطي علامة `#` وتحويل الجزء المتبقي من السلسلة إلى عدد صحيح.
  - `return -1;` : إذا لم تكن السلسلة تمثل قيمة فورية تبدأ بـ `#` ، يتم إرجاع -1.

Plain Text

```
bool is_register(char *str) {
    if (str == NULL || strlen(str) < 2 || (str[0] != 'R' && str[0] != 'r')) {
        return false;
    }
    for (int i = 1; i < strlen(str); i++) {
        if (!isdigit(str[i])) {
            return false;
        }
    }
    int reg_num = atoi(str + 1);
    return reg_num >= 0 && reg_num <= 15; // ARM typically has R0-R15
}
```

- `is_register` : هذه الدالة تتحقق مما إذا كانت السلسلة النصية تمثل اسم سجل ARM صالح (R0-R15).
  - `if (str == NULL || strlen(str) < 2 || (str[0] != 'R' && str[0] != 'r'))` : تتحقق من الشروط الأساسية لاسم السجل (يبدأ بـ 'r' أو 'R'، طوله على الأقل 2، ليس فارغًا).

- `for (int i = 1; i < strlen(str); i++) { if (!isdigit(str[i])) { return false; } }` : تتحقق مما إذا كانت جميع الأحرف بعد الحرف الأول هي أرقام.
- `int reg_num = atoi(str + 1);` : تحول الجزء الرقمي من اسم السجل إلى عدد صحيح.
- `return reg_num >= 0 && reg_num <= 15;` : تتحقق مما إذا كان رقم السجل ضمن النطاق الصالح (0- لسجلات (15 ARM.

Plain Text

```
void preprocess_line(char *line) {
    char *comment_start = strchr(line, '@');
    if (comment_start != NULL) {
        *comment_start = '\0'; // Null-terminate at comment start
    }
    // Trim trailing whitespace
    int len = strlen(line);
    while (len > 0 && isspace(line[len - 1])) {
        line[--len] = '\0';
    }
}
```

- `preprocess_line` : هذه الدالة تقوم بمعالجة أولية للسطر المدخل من ملف Assembly.
  - `char *comment_start = strchr(line, '@');` : تبحث عن أول ظهور لعلامة `@` (التي تشير إلى بداية تعليق).
  - `if (comment_start != NULL) { *comment_start = '\0'; }` : إذا تم العثور على علامة `@`، يتم استبدالها بحرف النهاية الصفرية (`0\`)، مما يؤدي إلى قطع السلسلة النصية عند هذه النقطة وإزالة التعليق.
  - `int len = strlen(line); while (len > 0 && isspace(line[len - 1])) { line[--len] = '\0'; }` : تزيل المسافات البيضاء الزائدة (whitespace) من نهاية السطر.

الدالة الرئيسية (main) - الممر الأول (First Pass)

Plain Text

```
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <input_file.text> <output_file.text>\\n",
argv[0]);
```

```
        return 1;
    }

    FILE *inputFile = fopen(argv[1], "r");
    if (inputFile == NULL) {
        perror("Error opening input file");
        return 1;
    }
```

- `main` : هذه هي الدالة الرئيسية للبرنامج.
  - `argc` : عدد المعاملات التي تم تمريرها إلى البرنامج من سطر الأوامر.
  - `argv` : مصفوفة من السلاسل النصية تحتوي على المعاملات.
- `if (argc != 3)` : يتحقق مما إذا كان عدد المعاملات صحيحًا. يتوقع البرنامج اسم ملف الإدخال وملف الإخراج (بالإضافة إلى اسم البرنامج نفسه).
  - `fprintf(stderr, ...)` : يطبع رسالة استخدام صحيحة إلى مخرج الأخطاء القياسي (`stderr`).
  - `return 1;` : ينهي البرنامج برمز خطأ.
- `FILE *inputFile = fopen(argv[1], "r");` : يفتح ملف الإدخال المحدد بواسطة (`argv[1]`) للقراءة (`"r"`).
- `if (inputFile == NULL)` : يتحقق مما إذا كان فتح الملف قد نجح.
  - `perror("Error opening input file");` : يطبع رسالة خطأ بناءً على الخطأ الأخير الذي حدث في النظام.
  - `return 1;` : ينهي البرنامج برمز خطأ.

Plain Text

```
    // First pass: Build symbol table
    char line[256];
    int current_address = 0;
    while (fgets(line, sizeof(line), inputFile) != NULL) {
        preprocess_line(line);
        if (strlen(line) == 0) continue; // Skip empty lines

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

        if (token != NULL) {
            // Check for label (ends with ":")
```

```
            if (token[strlen(token) - 1] == ":") {
                token[strlen(token) - 1] = '\0'; // Remove ":"
                add_symbol(token, current_address);
            } else {
                // Assume it's an instruction, increment address
                current_address += 4; // ARM instructions are 4 bytes
            }
        }
        free(temp_line);
    }
    rewind(inputFile); // Reset file pointer for second pass
```

- هذا الجزء يمثل **الممر الأول (First Pass)** للـ assembler. الهدف الرئيسي للممر الأول هو بناء جدول الرموز (Symbol Table) الذي يحتوي على جميع التسميات (labels) وعناوينها المقابلة.

- `char line[256];` : مصفوفة لتخزين كل سطر يتم قراءته من ملف الإدخال.
- `int current_address = 0;` : متغير لتتبع العنوان الحالي للتعليمات. كل تعليمة ARM هي 4 بايت.
- `while (fgets(line, sizeof(line), inputFile) != NULL)` : حلقة تقرأ سطرًا سطرًا من ملف الإدخال حتى نهاية الملف.
- `preprocess_line(line);` : تستدعي الدالة المساعدة لإزالة التعليقات والمسافات البيضاء الزائدة من السطر.
- `if (strlen(line) == 0) continue;` : تتخطى الأسطر الفارغة بعد المعالجة المسبقة.
- `char *temp_line = strdup(line);` : تنشئ نسخة من السطر الحالي. `strtok` ، تعدل السلسلة الأصلية، لذا من الأفضل العمل على نسخة.
- `char *token = strtok(temp_line, " \t");` : تقسم السطر إلى
- `if (token != NULL)` : تتحقق مما إذا كان هناك أي كلمات في السطر.
- `if (token[strlen(token) - 1] == ":")` : تتحقق مما إذا كانت الكلمة الأولى تنتهي بعلامة `:` ، مما يشير إلى أنها تسمية (label).
    - `token[strlen(token) - 1] = '\0';` : تزيل علامة `:` من نهاية التسمية.
    - `add_symbol(token, current_address);` : تضيف التسمية وعنوانها الحالي إلى جدول الرموز.

- `else { current_address += 4; }` : إذا لم تكن الكلمة الأولى تسمية، فمن المفترض أنها تعليمة، وبالتالي يتم زيادة `current_address` بمقدار 4 بايت (حجم تعليمة ARM).
- `free(temp_line);` : تحرر الذاكرة المخصصة لـ `temp_line`.

- `rewind(inputFile);` : تعيد مؤشر ملف الإدخال إلى البداية، استعدادًا للممر الثاني.

الدالة الرئيسية (main) - الممر الثاني (Second Pass)

Plain Text

```
    FILE *outputFile = fopen(argv[2], "w");
    if (outputFile == NULL) {
        perror("Error opening output file");
        fclose(inputFile);
        return 1;
    }
```

- `FILE *outputFile = fopen(argv[2], "w");` : يفتح ملف الإخراج (المحدد بواسطة `argv[2]`) للكتابة (`"w"`). إذا كان الملف موجودًا، فسيتم مسح محتواه.

- `if (outputFile == NULL)` : يتحقق مما إذا كان فتح الملف قد نجح.
  - `perror("Error opening output file");` : يطبع رسالة خطأ.
  - `fclose(inputFile);` : يغلق ملف الإدخال المفتوح.
  - `return 1;` : ينهي البرنامج برمز خطأ.

Plain Text

```
    // Second pass: Generate machine code
    current_address = 0;
    while (fgets(line, sizeof(line), inputFile) != NULL) {
        preprocess_line(line);
        if (strlen(line) == 0) continue; // Skip empty lines

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

        if (token == NULL) { free(original_line); continue; }
```

```
        // If it\'s a label, skip it (already processed in first pass)
        if (token[strlen(token) - 1] == ":") {
            free(original_line);
            continue;
        }

        // It\'s an instruction
        char *mnemonic = token;
        unsigned int machine_code = 0;
        int instruction_found = 0;

        for (int i = 0; instructions[i].mnemonic != NULL; i++) {
            if (strcmp(instructions[i].mnemonic, mnemonic) == 0) {
                machine_code = instructions[i].opcode_base;
                instruction_found = 1;

                if (instructions[i].type == TYPE_DATA_PROCESSING_REG) { //
Rd, Rn, Rm
                    char *arg1 = strtok(NULL, ", "); // Rd
                    char *arg2 = strtok(NULL, ", "); // Rn
                    char *arg3 = strtok(NULL, ", "); // Rm

                    int rd = get_register_num(arg1);
                    int rn = get_register_num(arg2);
                    int rm = get_register_num(arg3);

                    machine_code |= (rn << 16) | (rd << 12) | (rm & 0xF); //
Simplified encoding
                } else if (instructions[i].type == TYPE_MOV_REG) { // MOV Rd,
Rm
                    char *arg1 = strtok(NULL, ", "); // Rd
                    char *arg2 = strtok(NULL, ", "); // Rm

                    int rd = get_register_num(arg1);
                    int rm = get_register_num(arg2);

                    machine_code |= (rd << 12) | (rm & 0xF); // Simplified
encoding
                } else if (instructions[i].type == TYPE_MUL) { // MUL Rd, Rm,
Rs
                    char *arg1 = strtok(NULL, ", "); // Rd
                    char *arg2 = strtok(NULL, ", "); // Rm
                    char *arg3 = strtok(NULL, ", "); // Rs

                    int rd = get_register_num(arg1);
                    int rm = get_register_num(arg2);
                    int rs = get_register_num(arg3);
```

```
                    machine_code |= (rm & 0xF) | ((rs & 0xF) << 8) | ((rd &
0xF) << 16); // Simplified MUL encoding
                } else if (instructions[i].type == TYPE_BRANCH) { // BGE,
BLT, B
                    char *target_label = strtok(NULL, " \t");
                    int target_address = get_symbol_address(target_label);
                    if (target_address != -1) {
                        // Calculate relative offset (simplified: just use
target address)
                        // A real assembler calculates PC-relative offset and
shifts it
                        machine_code |= ((target_address - current_address -
8) >> 2) & 0x00FFFFFF; // Simplified relative branch
                    } else {
                        fprintf(stderr, "Error: Undefined label \'%s\'\\n",
target_label);
                    }
                } else if (instructions[i].type == TYPE_SWI) { // SWI
                    char *imm_str = strtok(NULL, " \t");
                    int imm_val = parse_immediate(imm_str);
                    if (imm_val != -1) {
                        machine_code |= (imm_val & 0x00FFFFFF); // Simplified
SWI immediate
                    } else {
                        fprintf(stderr, "Error: Invalid operand for %s:
%s\\n", mnemonic, imm_str);
                    }
                } else if (instructions[i].type == TYPE_CMP_REG) { // CMP Rn,
Rm or CMP Rn, #imm
                    char *arg1 = strtok(NULL, ", "); // Rn
                    char *arg2 = strtok(NULL, ", "); // Rm or #imm

                    int rn = get_register_num(arg1);

                    if (is_register(arg2)) { // CMP Rn, Rm
                        int rm = get_register_num(arg2);
                        machine_code |= (rn << 16) | (rm & 0xF); //
Simplified encoding
                    } else {
                        int imm = parse_immediate(arg2);
                        if (imm != -1) {
                            machine_code |= (1 << 25); // Set I bit
                            machine_code |= (rn << 16) | (imm & 0xFF); //
Simplified encoding
                        } else {
                            fprintf(stderr, "Error: Invalid operand for %s:
%s\\n", mnemonic, arg2);
                        }
```

```
                    }
                } else if (instructions[i].type == TYPE_SHIFT_IMM) { //
LSL/LSR Rd, Rm, #imm
                    char *arg1 = strtok(NULL, ", "); // Rd
                    char *arg2 = strtok(NULL, ", "); // Rm
                    char *arg3 = strtok(NULL, ", "); // #imm

                    int rd = get_register_num(arg1);
                    int rm = get_register_num(arg2);
                    int imm = parse_immediate(arg3);

                    if (imm != -1) {
                        // Simplified shift encoding: LSL/LSR Rd, Rm, #imm
                        // Opcode for LSL/LSR is part of the base, shift
amount in bits 7-11
                        // Rm in bits 0-3, Rd in bits 12-15
                        machine_code |= (rm & 0xF) | ((rd & 0xF) << 12) |
((imm & 0x1F) << 7);
                        if (strcmp(mnemonic, "LSR") == 0) {
                            machine_code |= (1 << 5); // Set bit 5 for LSR
                        }
                    } else {
                        fprintf(stderr, "Error: Invalid immediate operand for
%s: %s\\n", mnemonic, arg3);
                    }
                }
                break;
            }
        }

        if (instruction_found) {
            fprintf(outputFile, "%08X\\n", machine_code);
            current_address += 4;
        } else {
            fprintf(stderr, "Unknown instruction or syntax error: %s\\n",
original_line);
        }
        free(original_line);
    }

    fclose(inputFile);
    fclose(outputFile);

    printf("ARM Assembly file assembled successfully!\\n");

    return 0;
}
```

- هذا الجزء يمثل **الممر الثاني (Second Pass)** للـ assembler. الهدف الرئيسي للممر الثاني هو قراءة ملف Assembly مرة أخرى، وتحويل كل تعليمة إلى الـ machine code المقابل لها، وكتابة الـ machine code إلى ملف الإخراج.

- `current_address = 0;` : يتم إعادة تهيئة `current_address` إلى 0 لبدء حساب العناوين من جديد للممر الثاني.
- `while (fgets(line, sizeof(line), inputFile) != NULL)` : حلقة تقرأ سطرًا سطرًا من ملف الإدخال.
- `preprocess_line(line);` : معالجة السطر.
- `if (strlen(line) == 0) continue;` : تخطي الأسطر الفارغة.
- `char *original_line = strdup(line);` : يتم الاحتفاظ بنسخة من السطر الأصلي للإبلاغ عن الأخطاء.
- `char *token = strtok(line, " \t");` : تقسيم السطر إلى كلمات.
- `if (token == NULL) { free(original_line); continue; }` : تخطي الأسطر الفارغة بعد التقسيم.
- `if (token[strlen(token) - 1] == ":") { free(original_line); continue; }` : إذا كانت الكلمة الأولى تسمية، يتم تخطيها لأنها تمت معالجتها بالفعل في الممر الأول.
- `char *mnemonic = token;` : الكلمة الأولى هي اسم التعليمة (mnemonic).
- `unsigned int machine_code = 0;` : متغير لتخزين الـ machine code الناتج للتعليمة الحالية.
- `int instruction_found = 0;` : علامة للإشارة إلى ما إذا تم العثور على التعليمة.
- `for (int i = 0; instructions[i].mnemonic != NULL; i++)` : حلقة تمر على جميع التعليمات المعرفة في مصفوفة `instructions`.
  - `if (strcmp(instructions[i].mnemonic, mnemonic) == 0)` : تتحقق مما إذا كان اسم التعليمة الحالي يطابق التعليمة في المصفوفة.
    - `machine_code = instructions[i].opcode_base;` : يتم تعيين الـ opcode الأساسي للتعليمة.
    - `instruction_found = 1;` : يتم تعيين العلامة إلى 1.
  - **كتل `if-else if` لمعالجة أنواع التعليمات المختلفة**: هذا هو الجزء الأكثر أهمية وتعقيدًا في الممر الثاني. بناءً على `instructions[i].type`، يتم تحليل المعاملات المتبقية وتشفيرها في الـ `machine_code`.

- `TYPE_DATA_PROCESSING_REG` : لمعالجة تعليمات مثل `ADD Rd, Rn, Rm` أو `ADD Rd, Rn, #imm` .
  - يتم تحليل `Rm` , `Rn` , `Rd` أو `#imm` .
  - يتم استخدام `is_register()` للتمييز بين السجل والقيمة الفورية.
  - يتم ترميز أرقام السجلات في المواقع الصحيحة داخل `machine_code` .
  - إذا كان المعامل الأخير قيمة فورية، يتم تعيين `I bit` (البت 25) في الـ `machine_code` .
- `TYPE_MOV_REG` : لمعالجة تعليمات `MOV Rd, #imm` أو `MOV Rd, Rm` .
  - يتم تحليل `Rd` و `Rm` أو `#imm` .
  - يتم استخدام `is_register()` للتمييز.
  - يتم ترميز السجلات أو القيمة الفورية.
- `TYPE_MUL` : لمعالجة تعليمات `MUL Rd, Rm, Rs` .
  - يتم تحليل `Rs` , `Rm` , `Rd` .
  - يتم ترميز أرقام السجلات في المواقع الصحيحة.
- `TYPE_BRANCH` : لمعالجة تعليمات القفز مثل `B` , `BLT` , `BGE` .
  - يتم تحليل التسمية المستهدفة.
  - يتم استخدام `get_symbol_address()` للحصول على عنوان التسمية من جدول الرموز.
  - يتم حساب الإزاحة النسبية (relative offset) وترميزها في الـ `machine_code` .
    (ملاحظة: هذا التشفير مبسط جدًا ولا يتبع بدقة قواعد ARM لـ PC-relative addressing).
- `TYPE_SWI` : لمعالجة تعليمات `SWI #imm` .
  - يتم تحليل القيمة الفورية.
  - يتم ترميز القيمة الفورية في الـ `machine_code` .
- `TYPE_CMP_REG` : لمعالجة تعليمات `CMP Rn, #imm` أو `CMP Rn, Rm` .

- يتم تحليل `Rn` و `Rm` أو `#imm` .
- يتم استخدام `is_register()` للتمييز.
- يتم ترميز السجلات أو القيمة الفورية، مع تعيين `I bit` إذا كانت قيمة فورية.
  - `TYPE_SHIFT_IMM` : لمعالجة تعليمات الإزاحة مثل `LSL Rd, Rm, #imm` أو `LSR Rd, Rm, #imm` .
- يتم تحليل `#imm` , `Rm` , `Rd` .
- يتم ترميز أرقام السجلات وقيمة الإزاحة في المواقع الصحيحة.
- يتم تعيين بت إضافي لـ `LSR` للتمييز بينها وبين `LSL` (في هذا التشفير المبسط).
  - `break;` : بمجرد العثور على التعليمة ومعالجتها، يتم الخروج من حلقة البحث عن التعليمات.
- `if (instruction_found)` : إذا تم العثور على التعليمة ومعالجتها بنجاح:
  - `fprintf(outputFile, "%08X\\n", machine_code);` : يتم طباعة الـ `machine_code` الناتج (بصيغة سداسية عشرية من 8 أرقام) إلى ملف الإخراج، متبوعًا بسطر جديد.
  - `current_address += 4;` : يتم زيادة `current_address` بمقدار 4 بايت.
- `else` : إذا لم يتم العثور على التعليمة أو حدث خطأ في بناء الجملة:
  - `fprintf(stderr, "Unknown instruction or syntax error: %s\\n", original_line);` : يتم طباعة رسالة خطأ إلى `stderr` .
- `free(original_line);` : تحرير الذاكرة المخصصة لـ `original_line` .
- `fclose(inputFile);` : يغلق ملف الإدخال.
- `fclose(outputFile);` : يغلق ملف الإخراج.
- `printf("ARM Assembly file assembled successfully!\\n");` : يطبع رسالة نجاح.
- `return 0;` : ينهي البرنامج بنجاح.

هذا يختتم شرح كود `assembler.c` . آمل أن يكون الشرح واضحًا ومفيدًا.