숭실대학교 Soongsil University

프로그램 보고서

나는 숭실대학교 컴퓨터학부의 일원으로 명예를 지키면서 생활하고 있습니다.

나는 보고서를 작성하면서 다음과 같은 사항을 준수하였음을 엄숙히 서약합니다.

1. 나는 자력으로 보고서를 작성하였습니다.
2. 나는 보고서에서 참조한 문헌의 출처를 밝혔으며 표절하지 않았습니다.
3. 나는 보고서의 내용을 조작하거나 날조하지 않았습니다.

교과목	시스템프로그래밍<나> 2025
프로젝트 명	Hoework #6 - C언어로 파서 구현하기
교과목 교수	최 재 영
제출인	컴퓨터학부(학과) 학번:20211397 성명:강동민 (출석번호:201)
제출일	2025년 4월 11일

차 례

1장 프로젝트 개요

1.1) 개발 배경 및 목적

SIC/XE 머신은 대표적인 가상 어셈블리 시스템으로, 시스템 소프트웨어 및 컴파일러 과목에서 핵심 개념을 학습하는 데 사용된다.

본 프로젝트는 추후 개발할 어셈블러의 기능 구현을 위한 시작 단계로, SIC/XE 어셈블리 코드를 파싱하여 구조화된 형태로 변환하고, 이를 기반으로 opcdoe를 매칭하는 목적을 가진다.

2장 배경 지식

2.1) 주제에 관한 배경지식

어셈블리 언어는 기계어에 가까운 저수준 언어로, 명령어, 피연산자, 라벨 등으로 구성되어 있다. 어셈블러는 이를 해석하여 기계어로 번역하는 프로그램이며, 일반적으로 2-pass 방식으로 동작한다.

▶ Pass 1:

- 각 명령어에 주소를 할당 (LOCCTR)
- 라벨이 나타나는 위치를 기록하여 심볼 테이블(Symbol Table)을 구성
- 각 라인의 구조를 파싱하여 후처리용 토큰 테이블 생성

▶ Pass 2:

- 명령어와 피연산자를 참조하여 실제 기계어(Object code)로 변환
- 각 지시어(BYTE, WORD, RESW, LTORG 등)를 해석하여 데이터를 생성
- Object File (.obj) 생성 및 로더를 위한 정보 기록
- 이 프로젝트에서는 Pass 1의 핵심 로직 중 파싱에 집중하여,
  어셈블리 소스코드를 토큰 단위로 분석하고 각 요소(label, operator, operand, comment)를 구조체로 정리한다.

2.2) 기술적 배경지식

이 프로젝트는 C 언어 기반으로 구현되며, 다음과 같은 기술적 요소들이 포함된다.

가) 파일 입출력 (File I/O)

fopen_s(), fgets(), fclose() 등을 이용해 어셈블리 파일을 한 줄씩 읽어들이고, 각 라인을 메모리 배열에 저장한다.

나) 문자열 파싱 및 안전 함수 사용

- strtok_s() : 구분자('₩t', '₩n', ' ')를 기준으로 문자열을 안정적으로 분리
- strcpy_s(), strdup() : 문자열 복사 및 동적 할당

- 파싱 과정에서는 피연산자(operand)를 구분하는 쉼표(,)와 주석(comment) 내부에 사용된 쉼표를 구분할 필요가 있다. 이를 위해 다음과 같은 2단계 방식으로 파싱을 진행한다.

1. 먼저 한 줄을 label, operator, operands, comment로 구분한다. 이 단계에서는 쉼표(,)를 기준으로 피연산자를 나누지 않기 때문에, 피연산자가 2개 이상일 경우에도 operands 필드에 하나의 문자열로 저장된다.

2. operands 필드에 저장된 문자열을 쉼표(,) 기준으로 다시 분할하여 각 피연산자를 개별적으로 나눈다. 이렇게 하면 주석(comment) 부분에 쉼표가 포함되어 있어도 피연산자 구분 과정에 영향을 주지 않기 때문에 충돌을 방지할 수 있다.

다) 구조체 기반 데이터 관리

다-1) _token

```
typedef struct _token {
	char* label;
	char* oper;
	char* operand[MAX_OPERAND];
	char comment[100];
} token;
```

각 명령어 줄을 위와 같은 구조체로 파싱하여 **token* token_table[MAX_LINES]**에 저장한다.

※MAX_OPERAND는 3으로 정의되어있다.

다-2) _inst

```
typedef struct _inst
{
	char str[10];			// 명령어 이름
	unsigned char op;		// 명령어 코드
	int format;			// 명령어 형식
	int ops;			// 피연산자 갯수
} inst;
```

각 명령어에 대한 정보를 저장하는 구조체이다.

명령어별 이름, opcode, 형식, operand수를 저장하며 해당 정보는 “inst_table.txt”에서 읽어 들인다. 파싱된 _inst구조체는 **inst* inst_table[MAX_INST]**에 저장된다.

3장 시스템 설계 내용

3.1) 전체 시스템 설계 내용

본 시스템은 SIC/XE 어셈블리 소스코드를 읽고 각 줄을 분석하여, 명령어의 opcode를 찾아 출력하는 간단한 어셈블러 파서 구조로 설계되었다. 전체 설계는 다음 세 단계로 나뉜다

1. 명령어 테이블 로딩 (inst_table.txt):

- 각 명령어의 이름, 형식(format), opcode, 피연산자 수를 구조체(inst)에 저장하여 전역 배열에 로딩함.

2. 소스코드 입력 파일 로딩 (input.txt):

- 각 줄을 문자열로 읽어 input_data[]에 저장함.

3. 토큰 분석 및 출력:

- 각 줄을 파싱하여 token 구조체(label, operator, operand[], comment)로 분리하고 token_table[]에 저장
- operator에 해당하는 opcode를 검색하여 출력. 이 구조는 Pass1의 전처리 역할에 해당하며, 이후 Object code 생성을 위한 기반을 제공한다.

3.2) 모듈별 설계 내용

◆ 1) init_inst_file()

- 명령어 테이블(inst_table.txt)을 읽어 각 명령어를 구조체에 저장
- 형식이 "3/4"인 경우에는 format을 3으로 저장

◆ 2) init_input_file()

- 어셈블리 소스 파일을 한 줄씩 읽어 input_data[]에 저장
- 최대 줄 수(MAX_LINES)를 넘지 않도록 체크

◆ 3) token_parsing()

- 각 줄을 분석하여 label, operator, operand[], comment를 추출
- 피연산자 분리 시 ','를 기준으로 최대 3개까지 처리하며, 주석 내 쉼표는 무시
- 명령어가 RSUB처럼 피연산자가 없을 경우 주석과 operand를 혼동하지 않도록 inst_table에서 피연산자 수를 참조

◆ 4) search_opcode()

- 명령어(operator)로부터 opcode를 찾아 inst_table에 존재시 인덱스 반환
- '+' 접두사가 붙은 format 4 명령어도 대응함

◆ 5) make_opcode_output()

- 토큰 테이블을 순회하며 각 줄 앞에 opcode를 출력 파일로 출력

4장 시스템 구현 내용 (구현 화면 포함)

4.1) 전체 시스템 구현 내용

프로그램은 C 언어로 구현되며, 명령어 테이블 파일과 어셈블리 소스 파일을 입력으로 받아 실행된다.

1. `init_my_assembler()`호출:

init_my_assembler() init_inst_file과 init_input_file을 호출해 명령어 테이블(inst_table.txt)과 어셈블리 소스 파일(input.txt)을 읽고, 각각 inst_table[]과 input_data[] 배열에 저장한다.

2. assem_pass1()을 통해 파싱 및 토큰 테이블 작성 :

input_data[]의 각 줄을 token_parsing()을 통해 파싱하여 token_table[]에 구조화된 형태로 저장한다.

3. make_opcode_output()을 통해 각 줄 앞에 opcode를 붙여 출력:

출력 결과는 output_opcode.txt에 저장되며, 동시에 화면에도 출력된다.

4.2) 모듈별 구현 내용

가) init_inst_file()

기계어 명령어 테이블을 초기화하는 함수로, inst_table.txt 파일을 한 줄씩 읽어 각 명령어에 대한 정보를 구조체에 저장한다.

- 명령어 이름, 형식, opcode, 피연산자 수를 sscanf_s()를 사용해 파싱
- format 필드가 "3/4"일 경우, format은 3으로 저장(추후 + 접두사로 format 4 인식)
- 명령어는 inst_table[] 배열에 동적으로 할당하여 저장

inst_table.txt의 경우 "이름 형식 기계어 코드 오퍼랜드의개수 ₩n"의 형식을 가진다. 따라서 buffer에 파일을 한줄씩 읽고 아래와 같이 sscanf_s함수를 작성해 parsing한다.

```
if (sscanf_s(buffer, "%s\t%s\t%x\t%d",
    new_inst->str, (unsigned int)sizeof(new_inst->str),
    format, (unsigned int)sizeof(format),
    &op_tmp,
    &new_inst->ops) != 4) {
    printf("inst_table.txt 파일을 파싱하는데 실패했습니다.\n");
    return -1;
}
```

나) init_input_file()

어셈블리 소스 코드 파일(input.txt)을 한 줄씩 읽어 input_data[]에 저장하는 함수이다.

- 파일을 열고 fgets()로 한 줄씩 읽음
- 각 줄은 _strdup()을 통해 메모리에 복사한 후 저장
- 최대 줄 수는 MAX_LINES로 제한

```
// 파일을 라인 단위로 읽어 input_data에 저장
// MAXLINES을 넘을 경우 중단
while (fgets(buffer, sizeof(buffer), file) != NULL && line_num < MAX_LINES)
{
    char* temp = _strdup(buffer);
    if (temp == NULL) {
        printf("메모리 할당 실패 at line %d\n", line_num);
        break;
    }
    input_data[line_num++] = temp;
}
```

메모리 할당 실패 시 오류 메시지 출력, 파일 열기 실패 시 에러를 반환한다.

다) token_parsing()

어셈블리 코드 한 줄을 파싱하여 token 구조체로 분리하는 핵심 함수이다.

- '.'으로 시작할 경우 주석처리한다.

```
// 주석처리
if (str[0] == '.') {
    strcpy_s(tok->comment,sizeof(tok->comment), str);
    token_table[token_line] = tok;
    token_line++;
    return 0;
}
```

- 한 줄을 label, operator, operands, comment 4가지 요소로 분리

```
// label이 없는경우 : '\t'로 시작
if (str[0] == '\t') {
    tmp_str = tmp_str + 1;
    tmp_oper = strtok_s(tmp_str, " \t\n", &context);
}
//label이 있는 경우
else {
    tmp_label = strtok_s(tmp_str, " \t\n", &context);
    tmp_oper = strtok_s(NULL, " \t\n", &context);
}
```

입력받은 문자의 시작이 '₩t'인지 그렇지 않은지를 비교하여 label이 있는경우와 없는 경우로 나눈다.

- RSUB와 같이 operand가 없는 명령어의 경우, comment를 operand로 잘못 parsing 하는 경우가 생길 수 있다. 이를 방지하기 위해 inst_table->ops에 저장된 최대 operand수를 참고해 for문의 횟수를 제한한다.

```
if (tmp_oper != NULL) {
    tok->oper = _strdup(tmp_oper);

    // operator가 NULL이 아닌경우 최대 피연산자 개수를 가져온다
    for (int i = 0; i < inst_index; i++) {
        const char* opcheck = (tmp_oper[0] == '+') ? tmp_oper + 1 : tmp_oper;
        if (strcmp(inst_table[i]->str, opcheck) == 0) {
            num_operands = inst_table[i]->ops;
            break;
        }
    }
}
// operand 개수가 0 ex) RSUB인 경우에는 나머지 문자열을 모두 comment처리한다.
if (num_operands > 0)   tmp_operands = strtok_s(NULL, " \t\n", &context);
tmp_comment = strtok_s(NULL, "", &context); //남은 문자열을 모두 주석으로 처리
```

- 주석에 포함된 쉼표와 operand의 쉼표를 구분하기 위해 2단계 파싱을 수행

1. tmp_opers에 쉼표(,)를 기준으로 피연산자를 나누지 않고, 피연산자가 2개 이상일 경우에도 operands 필드에 하나의 문자열로 저장한다.
2. tmp_opers를 쉼표(,)을 기준으로 다시 parsing하여 저장한다.

```
/* operand parsing :
 * operand는 ','로 구분되어 있다.
 * tmp_operands로 따로 구분한 이유는 comment의 ','와 구분하기 위해서이다.
 */
 if (tmp_operands != NULL) {
    char* operand_context = NULL;   // strtok_s()의 context 변수
    char* tmp_operand = strtok_s(tmp_operands, ",", &operand_context);
    int i = 0;
    while (tmp_operand != NULL && i < MAX_OPERAND) {
        tok->operand[i++] = _strdup(tmp_operand);
        tmp_operand = strtok_s(NULL, ",", &operand_context);
    }
 }
```

라) search_opcode()

문자열로 주어진 operator를 inst_table[]에서 검색하여 해당 명령어의 inst_table 인덱스를 반환하는 함수이다.

- '+'로 시작하는 명령어 처리 (예: +JSUB → JSUB)

```
//4형식 처리 : +로 시작
if (str[0] == '+')
{
    str++;  // +를 제거


}
```

- inst_table에서 문자열 비교를 통해 opcode 검색하고, 일치하는 opcode를 찾으면 해당 opcode값을, 그렇지 못한 경우 -1을 반환한다.

```
for (int i = 0; i < inst_index; i++)
{
    if (strcmp(inst_table[i]->str, str) == 0)
    {
        return i;
    }
}
return -1;  // 찾지 못한 경우
```

마) make_opcode_output()

token_table[]을 기반으로 opcode와 어셈블리 소스코드를 한 줄씩 출력 파일(output_opcode.txt)에 기록하는 함수이다.

5장 기대효과 및 결론

본 프로젝트를 통해 SIC/XE 어셈블리 구조를 직접 다루며, 어셈블리어의 기본 구성 요소인 label, opcode, operand, comment의 의미와 사용법을 실제 코드 차원에서 명확히 이해할 수 있었다. 단순히 어셈블리 코드를 읽는 수준을 넘어, 어셈블러의 전체적인 처리 흐름을 체험할 수 있었다. 특히 파싱 과정에서 포맷4 명령어(+ 접두사) 처리, 피연산자 수에 따른 주석 구분, 쉼표(,) 처리 방식 등 실제 어셈블러가 고려해야 할 복잡한 요소들을 경험함으로써 시스템 소프트웨어의 기초 원리를 확실히 익힐 수 있었다.

첨부) 프로그램 소스파일

<my_assembler.c>

```
/*
 * 파일명 : my_assembler.c
 * 설  명 : 이 프로그램은 SIC/XE 머신을 위한 간단한 Assembler 프로그램의 메인루틴으로,
 * 입력된 파일의 코드 중, 명령어에 해당하는 OPCODE를 찾아 출력한다.
 *
 */
/*
  *
  * 프로그램의 헤더를 정의한다.
  *
  */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
  // 파일명의 "00000000"은 자신의 학번으로 변경할 것.
#include "my_assembler_20211397.h"
/*
------------------------------------------------------------------------------
------
 * 설명 : 사용자로 부터 어셈블리 파일을 받아서 명령어의 OPCODE를 찾아 출력한다.
 * 매계 : 실행 파일, 어셈블리 파일
 * 반환 : 성공 = 0, 실패 = < 0
 * 주의 : 현재 어셈블리 프로그램의 리스트 파일을 생성하는 루틴은 만들지 않았다.
 *                또한 중간파일을 생성하지 않는다.
 *
------------------------------------------------------------------------------
------
 */
int main(int args, char* arg[])
{
        if (init_my_assembler() <0)
        {
                printf("init_my_assembler: 프로그램 초기화에 실패 했습니다.₩n");
                return -1;
        }
        if (assem_pass1() <0)
        {
                printf("assem_pass1: 패스1 과정에서 실패하였습니다.  ₩n");
                return -1;
        }
        make_opcode_output(NULL);       // 기계어 코드 출력
        // make_symtab_output("output_symtab.txt");          //  추후 과제에 사용 예정
        // make_literaltab_output("output_littab.txt");      //  추후 과제에 사용 예정
        if (assem_pass2() <0)
        {
                printf(" assem_pass2: 패스2 과정에서 실패하였습니다.  ₩n");
                return -1;
        }
        // make_objectcode_output("output_objectcode.txt"); //  추후 과제에 사용 예정
}
/*
------------------------------------------------------------------------------
------
```

```
 * 설명 : 프로그램 초기화를 위한 자료구조 생성 및 파일을 읽는 함수이다.
 * 매계 : 없음
 * 반환 : 정상종료 = 0 , 에러 발생 = -1
 * 주의 : 각각의 명령어 테이블을 내부에 선언하지 않고 관리를 용이하게 하기
 *                위해서 파일 단위로 관리하여 프로그램 초기화를 통해 정보를 읽어 올 수 있도록
 *                구현하였다.
 *
-------------------------------------------------------------------------------
------
 */
int init_my_assembler(void)
{
        int result;
        if ((result = init_inst_file("inst_table.txt")) <0)
                return -1;
        if ((result = init_input_file("input.txt")) <0)
                return -1;
        return result;
}
/                                                                              *
-------------------------------------------------------------------------------
-----
 * 설명 : 머신을 위한 기계 코드목록 파일(inst_table.txt)을 읽어
 *       기계어 목록 테이블(inst_table)을 생성하는 함수이다.
 *
 *
 * 매계 : 기계어 목록 파일
 * 반환 : 정상종료 = 0 , 에러 < 0
 * 주의 : 기계어 목록파일 형식은 자유롭게 구현한다. 예시는 다음과 같다.
 *
 *
===============================================================================
==
 *                | 이름 | 형식 | 기계어 코드 | 오퍼랜드의 갯수 | ₩n |
 *
===============================================================================
==
 *
 *
-------------------------------------------------------------------------------
-----
 */
int init_inst_file(char* inst_file)
{
        FILE* file;
        int errno;
        char buffer[255];
        errno = fopen_s(&file, inst_file, "r");
        if (errno !=0)
        {
                printf("파일 열기 실패₩n");
                return -1;
        }
        fseek(file, 0, SEEK_SET);
        while (fgets(buffer, sizeof(buffer), file) !=NULL)
        {
                inst* new_inst = (inst*)malloc(sizeof(inst));
                if (new_inst ==NULL)
```

```
                {
                        printf("메모리 할당 실패₩n");
                        return -1;
                }
                unsigned int op_tmp;
                char format[10];
                if (sscanf_s(buffer, "%s₩t%s₩t%x₩t%d",
                        new_inst->str, (unsigned int)sizeof(new_inst->str),
                        format, (unsigned int)sizeof(format),
                        &op_tmp,
                        &new_inst->ops) !=4) {
                        printf("inst_table.txt 파일을 파싱하는데 실패했습니다.₩n");
                        return -1;
                }
                //opcode 저장
                new_inst->op = (unsigned char)op_tmp;
                /*
                 * format 저장 :
                 * format == "3/4" 이면 일단 ops == 3을 저장하고, Input_data에서 + 를 만나면
ops == 4로 업데이트 한다.
                 */
                if (format[1] =='/') {
                        new_inst->format =3;
                }
                else {
                        new_inst->format = format[0] -'0';
                }
                // inst_table에 저장
                inst_table[inst_index++] = new_inst;
        }
        fclose(file);
        return errno;
}
/                                                                                *
------------------------------------------------------------------------------
-----
 * 설명 : 어셈블리 할 소스코드 파일(input.txt)을 읽어 소스코드 테이블(input_data)를 생성하는 함수이
다.
 * 매계 : 어셈블리할 소스파일명
 * 반환 : 정상종료 = 0 , 에러 < 0
 * 주의 : 라인단위로 저장한다.
 *
 *
------------------------------------------------------------------------------
-----
 */
int init_input_file(char* input_file)
{
        FILE* file;
        int errno;
        char buffer[MAX_LINE_LENGTH];
        errno = fopen_s(&file, input_file, "r");
        if (errno !=0)
        {
                printf("파일 열기 실패₩n");
                return -1;
        }
        fseek(file, 0, SEEK_SET);
```

```
        // 파일을 라인 단위로 읽어 input_data에 저장
        // MAXLINES을 넘을 경우 중단
        while (fgets(buffer, sizeof(buffer), file) !=NULL && line_num < MAX_LINES)
        {
                char* temp = _strdup(buffer);
                if (temp ==NULL) {
                        printf("메모리 할당 실패 at line %d₩n", line_num);
                        break;
                }
                input_data[line_num++] = temp;
        }
        // MAXLINES을 넘을 경우 에러
        if (line_num >= MAX_LINES) {
                printf("최대 라인수를 초과하였습니다.₩n");
                return -1;
        }
        fclose(file);
        return errno;
}
/                                                                                                    *
-------------------------------------------------------------------------------
-----
 * 설명 : 소스 코드를 읽어와 토큰단위로 분석하고 토큰 테이블을 작성하는 함수이다.
 *        패스 1로 부터 호출된다.
 * 매계 : 파싱을 원하는 문자열
 * 반환 : 정상종료 = 0 , 에러 < 0
 * 주의 : my_assembler 프로그램에서는 라인단위로 토큰 및 오브젝트 관리를 하고 있다.
 *
-------------------------------------------------------------------------------
-----
 */
int token_parsing(char* str)
{
        // 임시토큰 초기화
        token* tok = (token*)malloc(sizeof(token));
        if (tok ==NULL) {
                printf("메모리 할당 실패₩n");
                return -1;
        }
        tok->label =NULL;
        tok->oper =NULL;
        for (int i =0; i < MAX_OPERAND; i++)    tok->operand[i] =NULL;
        tok->comment[0] ='₩0';
        // 주석처리
        if (str[0] =='.') {
                strcpy_s(tok->comment, sizeof(tok->comment), str);
                token_table[token_line] = tok;
                token_line++;
                return 0;
        }
        char* tmp_label =NULL;
        char* tmp_oper =NULL;
        char* tmp_str = _strdup(str);
        char* tmp_operands =NULL;
        char* tmp_comment =NULL;
        char* context =NULL;    // strtok_s()의 context 변수
        int num_operands = MAX_OPERAND;     // 피연산자 개수
        // label이 없는경우 : '₩t'로 시작
```

```
        if (str[0] =='₩t') {
                tmp_str = tmp_str +1;
                tmp_oper = strtok_s(tmp_str, " ₩t₩n", &context);
        }
        //label이 있는 경우
        else {
                tmp_label = strtok_s(tmp_str, " ₩t₩n", &context);
                tmp_oper = strtok_s(NULL, " ₩t₩n", &context);
        }
        if (tmp_oper !=NULL) {
                tok->oper = _strdup(tmp_oper);
                // operator가 NULL이 아닌경우 최대 피연산자 개수를 가져온다
                for (int i =0; i < inst_index; i++) {
                        const char* opcheck = (tmp_oper[0] =='+') ? tmp_oper +1 : tmp_oper;
                        if (strcmp(inst_table[i]->str, opcheck) ==0) {
                                num_operands = inst_table[i]->ops;
                                break;
                        }
                }
        }
        // operand 개수가 0 ex) RSUB인 경우에는 나머지 문자열을 모두 comment처리한다.
        if (num_operands >0)    tmp_operands = strtok_s(NULL, " ₩t₩n", &context);
        tmp_comment = strtok_s(NULL, "", &context);     //남은 문자열을 모두 주석으로 처리
        // label과 존재시 할당
        if (tmp_label !=NULL) {
                tok->label = _strdup(tmp_label);
        }
        /* operand parsing :
         * operand는 ','로 구분되어 있다.
         * tmp_operands로 따로 구분한 이유는 comment의 ','와 구분하기 위해서이다.
         */
        if (tmp_operands !=NULL) {
                char* operand_context =NULL;   // strtok_s()의 context 변수
                char* tmp_operand = strtok_s(tmp_operands, ",", &operand_context);
                int i =0;
                while (tmp_operand !=NULL && i < MAX_OPERAND) {
                        tok->operand[i++] = _strdup(tmp_operand);
                        tmp_operand = strtok_s(NULL, ",", &operand_context);
                }
        }
        // comment는 ','와 상관없이 통째로 저장
        if (tmp_comment !=NULL) {
                //comment 공백제거
                while (*tmp_comment ==' '||*tmp_comment =='₩t') tmp_comment++;
                strcpy_s(tok->comment, sizeof(tok->comment), tmp_comment);
        }
        token_table[token_line++] = tok;
        return 0;
}
/                                                                                *
------------------------------------------------------------------------------
-----
* 설명 : 어셈블리 코드를 위한 패스1과정을 수행하는 함수이다.
*                 패스1에서는..
*                 1. 프로그램 소스를 스캔하여 해당하는 토큰단위로 분리하여 프로그램 라인별 토큰
*                 테이블을 생성한다.
*         2. 토큰 테이블은 token_parsing()을 호출하여 설정한다.
*         3. assem_pass2 과정에서 사용하기 위한 심볼테이블 및 리터럴 테이블을 생성한다.
```

```
*
*
*
* 매계 : 없음
* 반환 : 정상 종료 = 0 , 에러 = < 0
* 주의 : 현재 초기 버전에서는 에러에 대한 검사를 하지 않고 넘어간 상태이다.
*            따라서 에러에 대한 검사 루틴을 추가해야 한다.
*
*        OPCODE 출력 프로그램에서는 심볼테이블, 리터럴테이블을 생성하지 않아도 된다.
*        그러나, 추후 프로젝트 1을 수행하기 위해서는 심볼테이블, 리터럴테이블이 필요하다.
*
*
------------------------------------------------------------------------------
------
*/
static int assem_pass1(void)
{
        /* input_data의 문자열을 한줄씩 입력 받아서
         * token_parsing()을 호출하여 _token에 저장
         */
        for (int i =0; i < line_num; i++) {
                if (token_parsing(input_data[i]) <0) {
                        printf("token_parsing: 토큰 파싱에 실패하였습니다.₩n");
                        return -1;
                }
        }
        return 0;
}
/                                                                              *
------------------------------------------------------------------------------
-----
 * 설명 : 입력 문자열이 기계어 코드인지를 검사하는 함수이다.
 * 매계 : 토큰 단위로 구분된 문자열
 * 반환 : 정상종료 = 기계어 테이블 인덱스, 에러 < 0
 * 주의 : 기계어 목록 테이블에서 특정 기계어를 검색하여, 해당 기계어가 위치한 인덱스를 반환한다.
 *        '+JSUB'과 같은 문자열에 대한 처리는 자유롭게 처리한다.
 *
 *
------------------------------------------------------------------------------
-----
 */
int search_opcode(char* str)
{
        // operator가 NULL인 경우
        if (str ==NULL)
        {
                return -1;
        }
        //4형식 처리 : +로 시작
        if (str[0] =='+')
        {
                str++;   // +를 제거
        }
        for (int i =0; i < inst_index; i++)
        {
                if (strcmp(inst_table[i]->str, str) ==0)
                {
                        return i;
```

```
                }
        }
        return -1;      // 찾지 못한 경우
}
/                                                                                *
-------------------------------------------------------------------------------
-----
* 설명 : 입력된 문자열의 이름을 가진 파일에 프로그램의 결과를 저장하는 함수이다.
*
* 매계 : 생성할 오브젝트 파일명
* 반환 : 없음
* 주의 : 소스코드 명령어 앞에 OPCODE가 기록된 코드를 파일에 출력한다.
*        파일이 NULL값이 들어온다면 프로그램의 결과를 stdout으로 보내어
*        화면에 출력해준다.
*
*        OPCODE 출력 프로그램의 최종 output 파일을 생성하는 함수이다.
*        (추후 프로젝트 1에서는 불필요)
*
*
-------------------------------------------------------------------------------
------
*/
void make_opcode_output(char* file_name)
{
        FILE* file;
        int errno;
        if (file_name ==NULL)
        {
                file = stdout;
        }
        else {
                errno = fopen_s(&file, file_name, "w");
                if (errno !=0)
                {
                        printf("파일 열기 실패₩n");
                        return;
                }
        }
        fseek(file, 0, SEEK_SET);
        for (int i =0; i < token_line; i++)
        {
                token* t = token_table[i];
                // 주석 라인
                if (t->oper ==NULL)
                {
                        fprintf(file, "%s₩n", t->comment);
                        continue;
                }
                // operand 문자열 만들기 (보안 버전)
                char operand_str[100] ="";
                for (int j =0; j < MAX_OPERAND; j++) {
                        if (t->operand[j] !=NULL) {
                                if (operand_str[0] !='₩0') {
                                        strcat_s(operand_str, sizeof(operand_str), ",");
                                }
                                strcat_s(operand_str, sizeof(operand_str), t->operand[j]);
                        }
                }
```

```
                // opcode 문자열 만들기 (보안 버전)
                char opcode_str[10] ="";
                int idx = search_opcode(t->oper);
                if (idx !=-1) {
                        sprintf_s(opcode_str, sizeof(opcode_str), "%02X", inst_table[idx]->op);
                }
                // 정렬된 출력
                fprintf(file, "%-8s%-8s%-16s%-8s₩n",
                        t->label ? t->label : "",
                        t->oper ? t->oper : "",
                        operand_str,
                        opcode_str);
        }
        if (file != stdout)
        {
                fclose(file);
        }
        return;
}
/*
-------------------------------------------------------------------------------
-----
* 설명 : 입력된 문자열의 이름을 가진 파일에 프로그램의 결과를 저장하는 함수이다.
*        여기서 출력되는 내용은 SYMBOL별 주소값이 저장된 TABLE이다.
* 매개 : 생성할 오브젝트 파일명 혹은 경로
* 반환 : 없음
* 주의 : 파일이 NULL값이 들어온다면 프로그램의 결과를 stdout으로 보내어
*        화면에 출력해준다.
*
*
-------------------------------------------------------------------------------
------
*/
void make_symtab_output(char* file_name)
{
        /* add your code here */
}
/*
-------------------------------------------------------------------------------
-----
* 설명 : 입력된 문자열의 이름을 가진 파일에 프로그램의 결과를 저장하는 함수이다.
*        여기서 출력되는 내용은 LITERAL별 주소값이 저장된 TABLE이다.
* 매개 : 생성할 오브젝트 파일명
* 반환 : 없음
* 주의 : 파일이 NULL값이 들어온다면 프로그램의 결과를 stdout으로 보내어
*        화면에 출력해준다.
*
*
-------------------------------------------------------------------------------
------
*/
void make_literaltab_output(char* filename)
{
        /* add your code here */
}
/*
-------------------------------------------------------------------------------
------
```

```
 * 설명 : 어셈블리 코드를 기계어 코드로 바꾸기 위한 패스2 과정을 수행하는 함수이다.
 *                패스 2에서는 프로그램을 기계어로 바꾸는 작업은 라인 단위로 수행된다.
 *                다음과 같은 작업이 수행되어 진다.
 *                1. 실제로 해당 어셈블리 명령어를 기계어로 바꾸는 작업을 수행한다.
 * 매개 : 없음
 * 반환 : 정상종료 = 0, 에러발생 = < 0
 * 주의 :
 *
-------------------------------------------------------------------------------
------
 */
static int assem_pass2(void)
{
        /* add your code here */
}
/                                                                              *
-------------------------------------------------------------------------------
-----
* 설명 : 입력된 문자열의 이름을 가진 파일에 프로그램의 결과를 저장하는 함수이다.
*        여기서 출력되는 내용은 object code이다.
* 매개 : 생성할 오브젝트 파일명
* 반환 : 없음
* 주의 : 파일이 NULL값이 들어온다면 프로그램의 결과를 stdout으로 보내어
*        화면에 출력해준다.
*        명세서의 주어진 출력 결과와 완전히 동일해야 한다.
*        예외적으로 각 라인 뒤쪽의 공백 문자 혹은 개행 문자의 차이는 허용한다.
*
*
-------------------------------------------------------------------------------
------
*/
void make_objectcode_output(char* file_name)
{
        /* add your code here */
}
```

<my_assembler_20211397.h>

```
/*
 * my_assembler 함수를 위한 변수 선언 및 매크로를 담고 있는 헤더 파일이다.
 *
 */
#define MAX_INST 256
#define MAX_LINES 5000
#define MAX_COLUMNS 4
#define MAX_OPERAND 3
#define MAX_LINE_LENGTH 1000
/*
 * instruction 목록을 저장하는 구조체이다.
 * instruction 목록 파일로부터 정보를 받아와서 생성한다.
 * instruction 목록 파일에는 라인별로 하나의 instruction을 저장한다.
 *
 */
typedef struct _inst
{
	char str[10];			// 명령어 이름
	unsigned char op;		// 명령어 코드
	int format;				// 명령어 형식
	int ops;				// 피연산자 갯수
} inst;
// 기계어를 관리하는 테이블
inst* inst_table[MAX_INST];
int inst_index;
// 어셈블리 할 소스코드를 파일로부터 불러와 라인별로 관리하는 테이블
char* input_data[MAX_LINES];
static int line_num;
int label_num;
/*
 * 어셈블리 할 소스코드를 토큰으로 변환하여 저장하는 구조체 변수이다.
 * operator 변수명은 renaming을 허용한다.
 */
typedef struct _token {
	char* label;
	char* oper;			//operator
	char* operand[MAX_OPERAND];
	char comment[100];
	// char nixbpe;    // 추후 프로젝트에서 사용
} token;
// 어셈블리 할 소스코드를 5000라인까지 관리하는 테이블
token* token_table[MAX_LINES];
static int token_line;
/*
 * 심볼을 관리하는 구조체이다.
 * 심볼 테이블은 심볼 이름, 심볼의 위치로 구성된다.
 * 추후 프로젝트에서 사용 예정
 */
typedef struct _symbol
{
	char symbol[10];
	int addr;
} symbol;
/*
* 리터럴을 관리하는 구조체이다.
* 리터럴 테이블은 리터럴의 이름, 리터럴의 위치로 구성된다.
```

```
* 추후 프로젝트에서 사용 예정
*/
typedef struct _literal {
        char* literal;
        int addr;
} literal;
symbol sym_table[MAX_LINES];
symbol literal_table[MAX_LINES];
/**
 * 오브젝트 코드 전체에 대한 정보를 담는 구조체이다.
 * Header Record, Define Recode,
 * Modification Record 등에 대한 정보를 모두 포함하고 있어야 한다. 이
 * 구조체 변수 하나만으로 object code를 충분히 작성할 수 있도록 구조체를 직접
 * 정의해야 한다.
 *
 * 추후 프로젝트에서 사용 예정
 */
typedef struct _object_code {
        /* add fields */
        char* obj_code; //에러방지 임시변수
} object_code;
static int locctr;
//---------------
static char* input_file;
static char* output_file;
int init_my_assembler(void);
int init_inst_file(char* inst_file);
int init_input_file(char* input_file);
int token_parsing(char* str);
int search_opcode(char* str);
static int assem_pass1(void);
void make_opcode_output(char* file_name);
void make_symtab_output(char* file_name);
void make_literaltab_output(char* filename);
static int assem_pass2(void);
void make_objectcode_output(char* file_name);
```

첨부2) 디버거 사용 확인

11번 라인에 “ DIVR k,l”입력시 oper=DIVR, operand[0]=k, operand[1]=l이 저장됨을 확인할 수 있다. label = NULL이다.