

20171969

박건호

과제명 : 시스템 프로그래밍 Project #1 SIC/XE assembler Programming

출석번호 : 110

수업 구분 : 가

1. 동기 / 목적

과제의 동기와 목적은 수업시간에 배운 control sections 방식의 어셈블리어 SIC/XE 프로그램을 Object Program Code로 변환해주는 어셈블러를 완성하는 것이다.

어셈블러를 구현하기 위해, 어셈블리어 프로그램 소스코드를 입력 받아 파싱하고 (파싱 과제), 소스코드의 명령어정보가 담긴 inst_table.txt를 토대로 Location Counter 할당, Symbol Table 작성, Literal Tabel 작성을 수행하여 결과적으로 소스코드의 한 라인마다 object code를 만들어낸다.

최종적으로 만들어진 어셈블러 오브젝트 코드는 어셈블리 언어로 작성된 소스 코드를 기계어로 번역한 실행 가능한 프로그램으로 변환된 코드이다.

본 과제 프로그램은 어셈블리어 소스코드와 기계어 목록을 읽어오고, 어셈블리 언어로 작성된 소스 코드를 토큰화하고, 각 소스코드 라인에 로케이션 카운터를 할당하고, 레이블을 심볼 테이블에 추가하고, 리터럴을 리터럴 테이블에 추가하고, nixbpe 값을 계산하고, 결과적으로 op코드의 값을 판단하고, 타겟 어드레스 계산을 수행하여 라인마다 알맞은 기계어 코드를 만들어내는 게 목적이다.

Pass1에서는 소스코드를 토큰으로 분할, 로케이션 카운터를 알맞게 증가. nixbpe 할당 레이블을 심볼 테이블에 추가 리터럴 풀, 리터럴 테이블 (본 과제 구현은 둘을 같은 리터럴 테이블에서 관리함)을 관리하는 기능을 수행한다.

Pass2 에서는 Pass1에서 만들었던 항목들을 통해 실제로 라인마다 기계어 코드를 계산한다.

주요 기능으로는 소스코드 파싱, 심볼 테이블의 관리, 리터럴의 처리, 오브젝트 코드의 생성 및 출력 등이 있다.

2. 설계 / 구현 아이디어

어셈블리 코드를 입력 받아 해당 코드를 기계어로 변환하는 이 과정은 Pass1, Pass2 두 단계로 나뉜다. 각 Pass 는 세부 함수들로 구성하였다.

데이터 구조 : 핵심 데이터 구조는 다음과 같다

기계어 명령어 테이블 (inst_table) : SIC/XE 기계어 명령어의 목록을 저장 - 인덱스로 해당 명령어에 접근하여 명령어의 정보를 가져올 수 있다

소스코드 테이블 (input) : 입력된 어셈블리 코드의 각 라인을 저장한다.

토큰 테이블 (tokens) : 파싱된 소스코드의 각 라인을 토큰 단위로 저장하고, nixbpe와 Location Counter 값을 저장한다.

심볼 테이블 (symbol_table) : 소스코드 내에서 사용된 모든 심볼과 그 위치를 저장하고, 소속 섹션과 타겟 어드레스 계산 시 섹션 시작주소를 더해야 하는지 의 여부 (Pass2에서 실제로 사용하는 않아서 출력상 하드코딩으로 구현하였음) , 섹션을 대표하는 심볼은 섹션의 총 크기도 저장한다.

리터럴 테이블(literal_table): 프로그램에서 사용된 리터럴과 그 위치, 소속 섹션을 저장한다.

오브젝트 코드(obj_code): 변환된 기계어 코드를 저장한다.

주요 함수

assem_pass1: 첫 번째 패스로 소스코드를 분석하며 tokens, symbol_table, literal_table을 생성한다

우선 패스원에선 토큰 할당 (파싱 과제) 이 먼저 일어났고 그 이후의 과정을 설명하겠다.

처음으로는 일단 오퍼레이터가 지시어인 경우에 대해서 판단한다. 스타트였다면 로케이션 카운터를 스타트 오퍼랜드 주소로 업데이트 하고 현재 섹션이 카피라는 것을 저장한다.

CSECT의 경우에는 작업 중이던 섹션이 끝났다는 뜻이다. 그렇기 때문에. 리터럴 테이블에 저장돼 있던 리터럴들 중 addr가 할당되지 않았던 리터럴에 대해서 모두 할당한다. 그리고 지금까지 계산된 로케이션 카운터의 값을 섹션의 총 크기로 심볼 테이블의 적절한 심볼에게 저장한 다음 현재 섹션이 CSECT의 레이블이라는 것을 알린다.

오퍼레이터가 EXTDEF이라면 현재 관리하는 EXTDEF operand들을 메모리 할당하여 기억해둔다.

LTORG가 나오면 실제로 리터럴 테이블에 있는 모든 리터럴들에 대해 메모리 할당을 해주면 된다.

END가 나오면 LORG를 또 해주고, 현재 섹션에 길이를 적절한 심볼에 저장한다.

지시어 처리가 끝났으면 다음으로는 심볼 테이블에 심볼들 추가하면 된다.

만약 오퍼레이터가 EQU였다면 심볼 테이블에 들어가는 값은 조금 처리가 필요한데, 만약 *가 나

왔다면 그때의 LC 값을 그대로 넣으면 된다. 만약 연산이 있다면 심볼 테이블을 Look up하여 심볼들의 어드레스를 구하고 그 어드레스들의 연산으로 값을 구한다.

다만 본 구현에서는 마이너스 연산에 대해 처리가 가능하도록 구현하였기에 만약 +연산이나 정수값의 연산 등이 나오면 오류가 있을 수 있다. EQU의 값을 다 계산했으면 심볼 테이블에 추가를 해주고 EQU 가 아닌 경우에도 심플하게 심볼 테이블에 추가해준다.

이렇게 심볼 테이블에 심볼 삽입이 끝나면 다음으로는 로케이션 카운터를 증가시키고, 할당을 해야 되는데, 증가시키기 전에 우선 해당 토큰에 대해 현재 LC 값을 할당한다.

그 다음으로 형식을 판단하여, 2형식이면 2증가 3형식이면 3증가 4형식이면 4를 증가시킨다. 그리고 RESB와 RESW, BYTE, WORD 지시어들은 몇 바이트, 몇 워드를 가지고 있는지 계산하여 증가시킨다. 동시에 NIXBPE도 세팅을 해주는데 이는 inst 테이블을 Look up 하여 기계어 인덱스를 찾고, 해당 인덱스를 이용하여, 명령어의 정보들을 가지고 와서 몇 형식인지 피연산자는 몇 개를 쓰는지를 판단하여 세팅한다. 그리고 operand를 보고 immediate 인지 indirect인지 NIXBPE 값을 조정한다. 만약 피연산자가 리터럴이라면 리터럴 테이블에 삽입을 하면서 어드레스 값을 초기화하여, 아직 할당이 되지 않은 리터럴임을 표시한다. 그러면은, 리터럴풀을 따로 관리할 필요가 없고 리터럴 테이블에 추가만 하여서 관리하여도 된다. 이 과정이 끝나면 Pass1에서 수행한 역할은 다한 것이다. 토큰마다 로케이션 카운터가 해당되었고, 토큰마다 NIXBPE 값도 할당되었고, 심볼도 모두 심볼테이블에 들어갔고, 리터럴도 리터럴 테이블로 만들어졌다. 그 외의 세부적인 함수들은 소스코드에 주석과 함께 기능 구현에 대한 설명을 자세히 달았다.

assem_pass2: 두 번째 패스로 tokens, symbol_table, literal_table을 사용해 최종 오브젝트 코드를 생성한다.

첫 시작은 END 지시어의 피연산자를 판단하는 일부터 시작한다. END의 피연산자는 프로그램이 끝나면 그 주소로 가라는 것을 지시한다. 만약 심볼이라면 그 심볼의 어드레스 값을 찾아가면 된다. 그런데 심볼이라면 그 심볼은 분명 소속된 섹션이 있을 것이다. 따라서 END의 오퍼랜드를 먼저 찾고, 그 오퍼랜드가 어느 섹션에 소속되어 있는지 심볼 테이블을 통해 찾고 해당하는 섹션을 변수에 저장하여 나중에 E 레코드를 작성하는데 사용한다.

이제 한 라인에 대해서 오브젝트 코드를 구하는 과정을 수행할 것인데, 기본적인 로직은 다음과 같다.

작업 버퍼를 두고 그 작업 버퍼에 오브젝트 코드에 대해 계산한 값을 계속 뒤쪽으로 붙여 나간다. 총 60 바이트까지 쓰여질 수 있는데 만약에 버퍼가 꽉 차서 비워내야 된다면 오브젝트 코드 라인을 작성하는 함수를 호출하여 버퍼에 있던 내용의 앞부분에 필요한 정보들을 기입하고 오브젝트 코드 라인에 쓰고 버퍼를 비우는 작업을 수행한다.

이것을 기본적인 과정으로 하되. 만약 다른 지시어들이 나오면 그 지시어에 따른 오브젝트 코드 라인생성을 수행할 것이다. 오브젝트를 구하는 과정 중에 modification 이 필요하다고 판단되는 것은 modification 레코드 변수를 따로 구조체로 관리하여 그곳에 계속 추가해 나간다.

만약 E라인을 작성해야 될 때, 그전에 modification 을 모두 쓰고 비우고 다시 다음 섹션의 modification Record 들을 채우고... 이런 식으로 진행을 한다.

먼저 맨 처음에는 작업버퍼를 초기화하고 시작을 한다.

지시어를 판단하여 스타트가 나오면 현재 작업 섹션을 설정하고 H 라인을. 작성하도록 함수를 호출한다. 이 함수는 버퍼의 내용을 받아서 적절한 문자를 버퍼에 추가하여 완전한 오브젝트 코드 라인을 만들고 그 라인을 오브젝트 코드 구조체에 쓰고. 버퍼는 초기화하는 과정을 거친다.

그럼 다음으로 EXTDEF 는 D로 시작하는. 라인을 작성하면 된다. EXTREF도 R로 시작하는 라인을 작성한다. CSECT는 조금 더 복잡한 과정이 필요하다.

CSECT는 이전 작업 섹션의 종료를 의미하기 때문에 LTORG 되지 않은 리터럴들을 메모리 할당해주는 프로세스 리터럴 함수를 호출한다. 그리고 해당 레이블을 작업 섹션으로 설정하고 버퍼에 뭔가 남아 있었다면 모두 오브젝트 구조체에 쓰고 지금까지 저장되었던 modification을 전부 쓰고, 마지막으로 E 레코드를 작성한다. E 레코드를 작성할 때는 반복문에 들어오기 전에 구했던 END CSECT는 이전 작업 섹션의 종료를 의미하기 때문에 LTORG 되지 않은 리터럴들을 메모리 할당해주는 프로세스 리터럴 함수를 호출한다. 그리고 해당 레이블을 작업 섹션으로 설정하고 버퍼에 뭔가 남아 있었다면 모두 오브젝트 구조체에 쓰고 지금까지 저장되었던 modification을 전부 쓰고, 마지막으로 E 레코드를 작성한다.

E 레코드를 작성할 때는 반복문에 들어오기 전에 구했던 END 피연산자의 판단에 근거하여 그냥 단순히 이에 한 글자만 쓸지 이해하고 시작 주소를 쓸지 결정하여 작성하면 된다.

그리고 CSECT 작업의 마지막에는 새로운 섹션이 시작되기도 하니까 H 라인까지 작성하는 함수를 호출한다. 이러면 총 T, M, E, H 네 개의 코드 라인을 작성하는 지시어인 것이다.

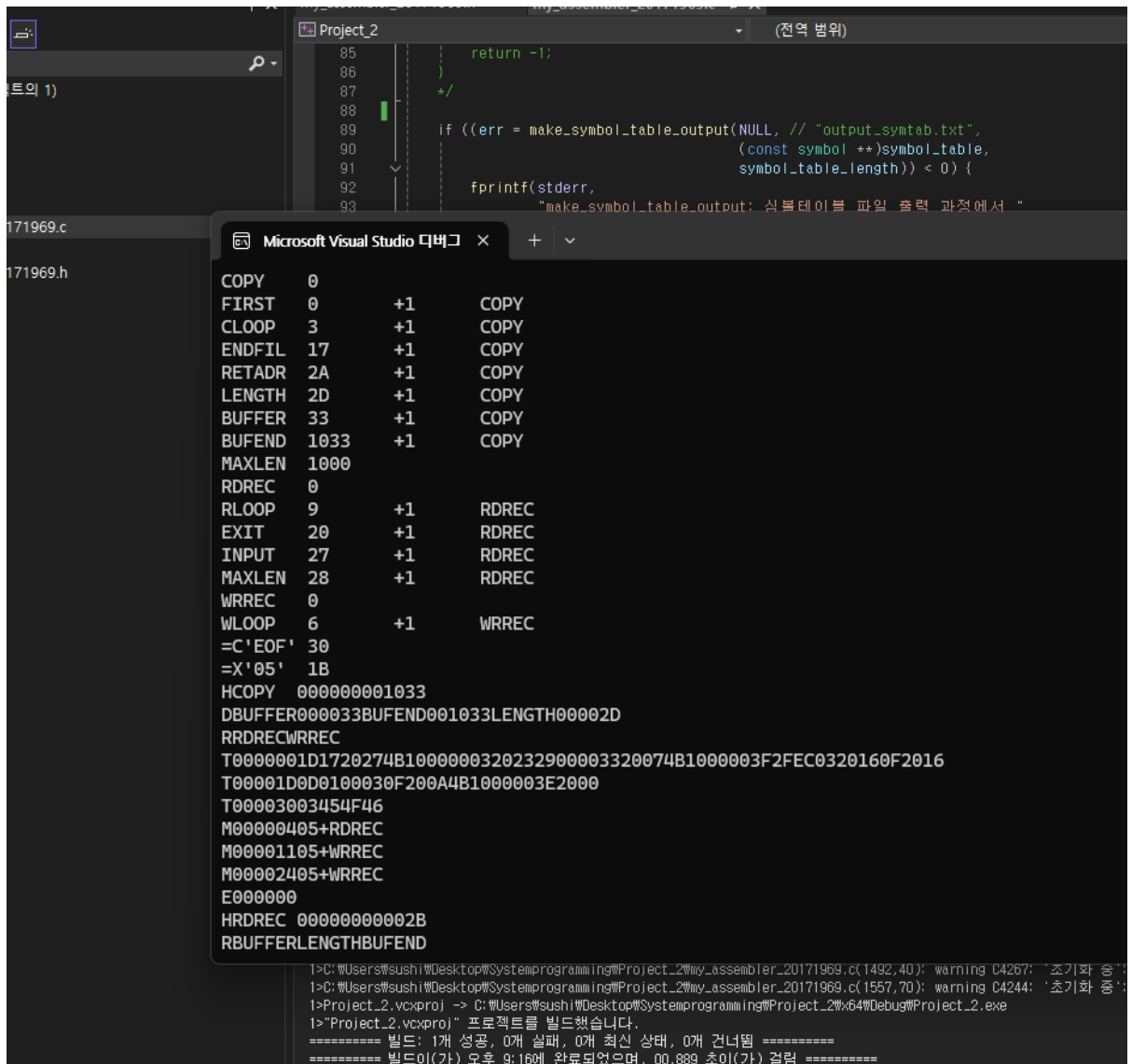
그리고 다음으로 가서 오퍼레이터를 판단하는데, 여기서 리저브 바이트와 리저브 워드는 메모리에 뭔가 특별한 내용을 쓰지 않고 공간만 잡아놓기 때문에 줄바꿈을 하면 된다. 만약 버퍼에 남아 있던 무언가 내용이 있다면 텍스트를 쓰고 다음 라인으로 넘어간다. LTORG는 assem pass 1에서 리터럴들에 대한 LC를 할당하고 소속 섹션까지 다 부여를 했기 때문에 조건문에 따라. LTORG 된 영역에서 assemble 된다. EQU는 아무 행위를 하지 않고. 바이트와 워드는 공간을 잡고 그 공간에 무슨 내용이 들어갈지 오퍼랜드를 보고 판단하여 계산한다.

오퍼레이터가 지시어가 아니었다면 이제 기본적으로 수행되는 T 라인을 위한 버퍼를 채워나가기 시작한다. 일단 pc 값을 계산을 해야 해서 형식에 따라 값을 조정한다. 그리고 op 코드는 inst 테이블을 Look up하여 찾아내서 왼쪽으로 몇 번 shift 할지는 형식에 따라 달려있고, 그 결과값에 추가적인 사항들 -. 몇 개의 레지스터를 쓰는지, nixbpe는 무엇인지 모두 구분하여 계산한다. 이런

게 오브젝트 결과로서 코드 결과를 알게 되었으면 버퍼에 추가해 준다. 3형식 4형식도 마찬가지로 다른 점 하나는 4형식을 처리할 때 displacement를 계산할 수 없기 때문에 modification 레코드에 정보를 추가하는 함수를 호출한다. 모든 과정이 끝나면, 버퍼가 꽉 찼다고 판단됐을 때만 텍스트 라인을 write하는 함수를 호출하고 그것이 아니라면 다음 라인, 또 다음 라인을 계속해서 계산, 버퍼에 추가해 나간다. 이 과정에서 쓰인 함수들은 소스코드에서 주석으로 자세히 설명하였다.

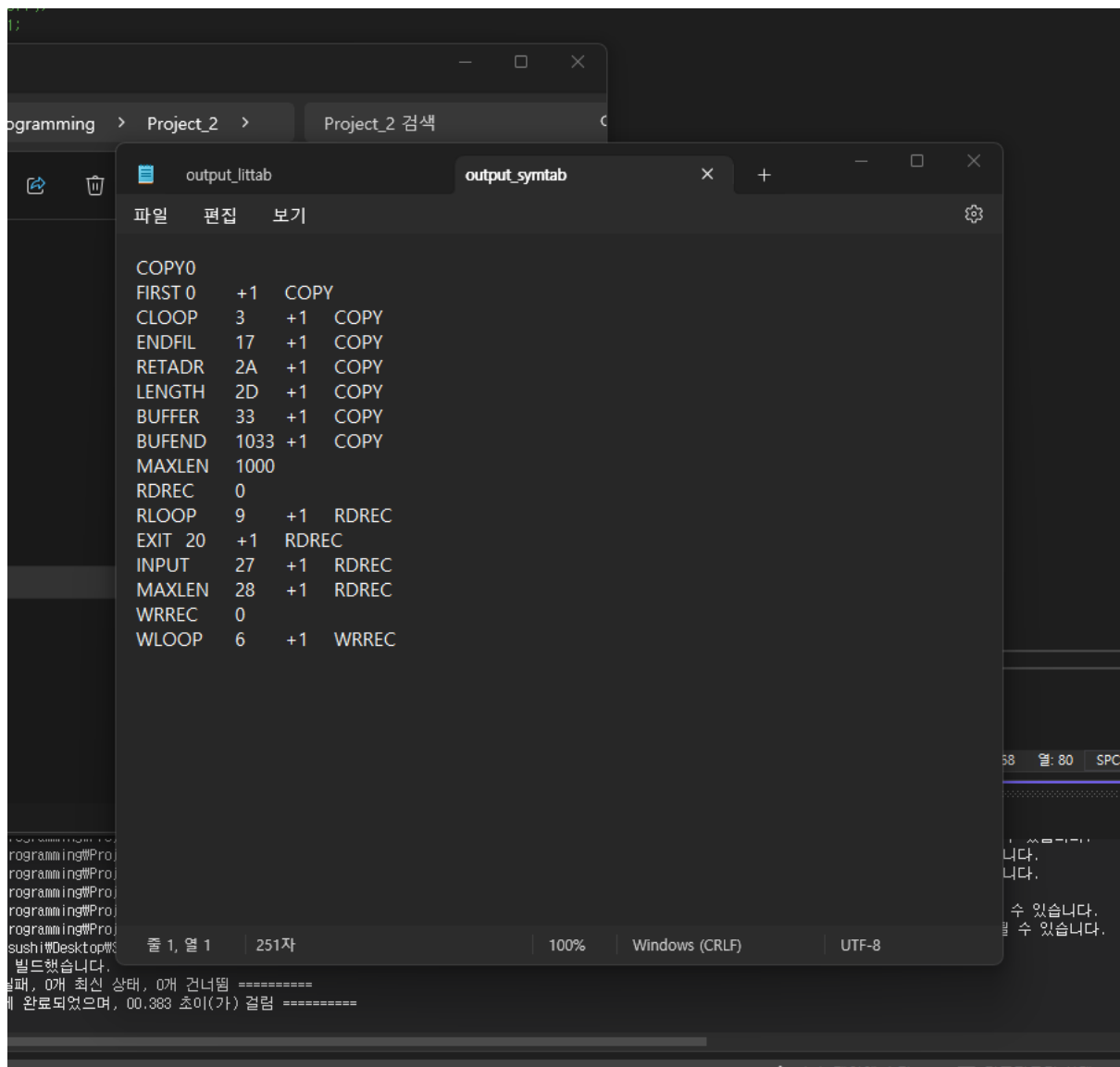
마지막으로 출력하는 함수들은 각 구조체에 있는 내용들을 간단하게 출력을 하는 함수들이다. 인자가 널로 들어오면 표준출력으로 출력하게 하고, 문자로 들어오면, 그 문자에 해당하는 파일을 만들어 거기에 출력한다.

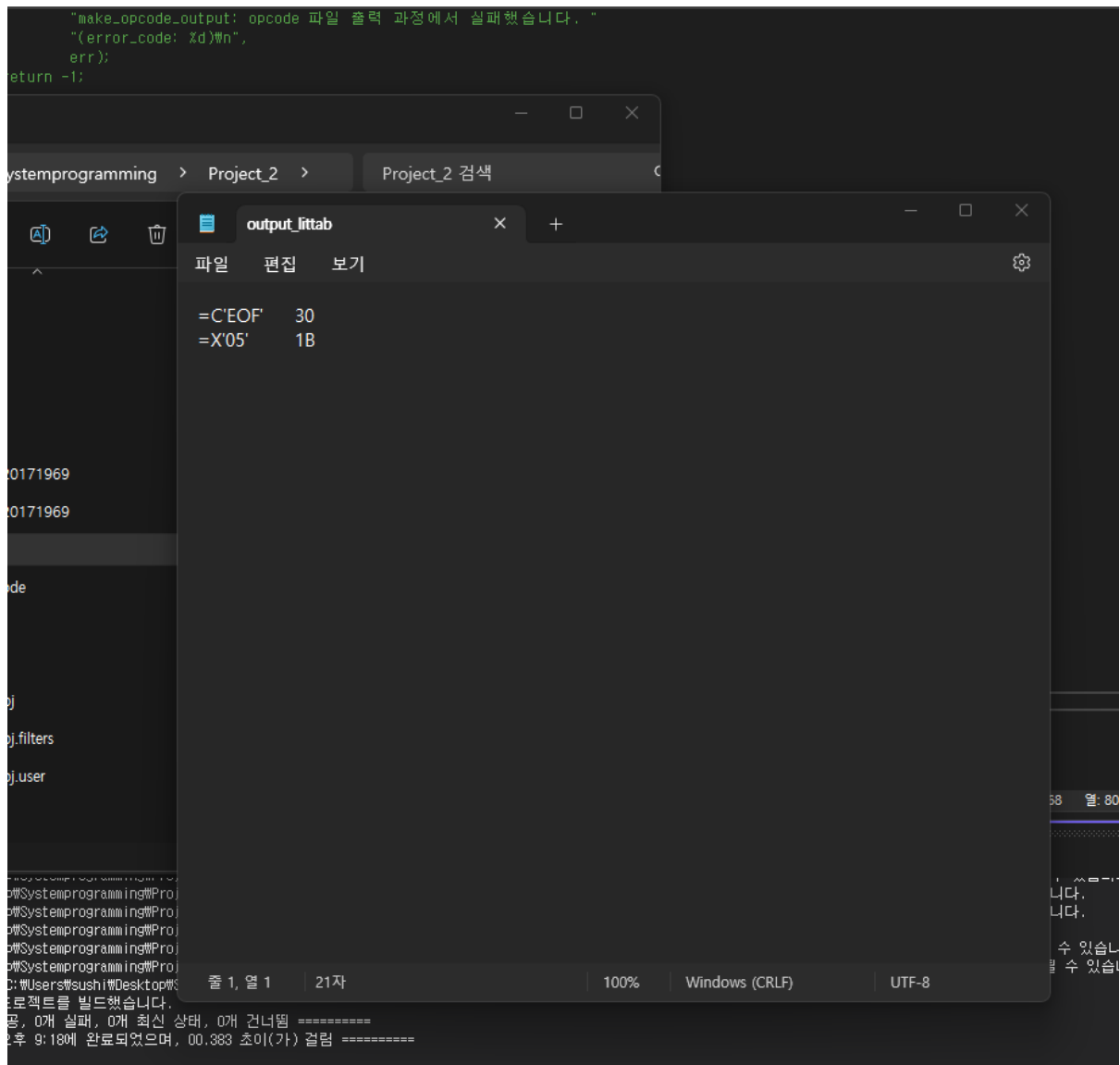
3. 수행결과

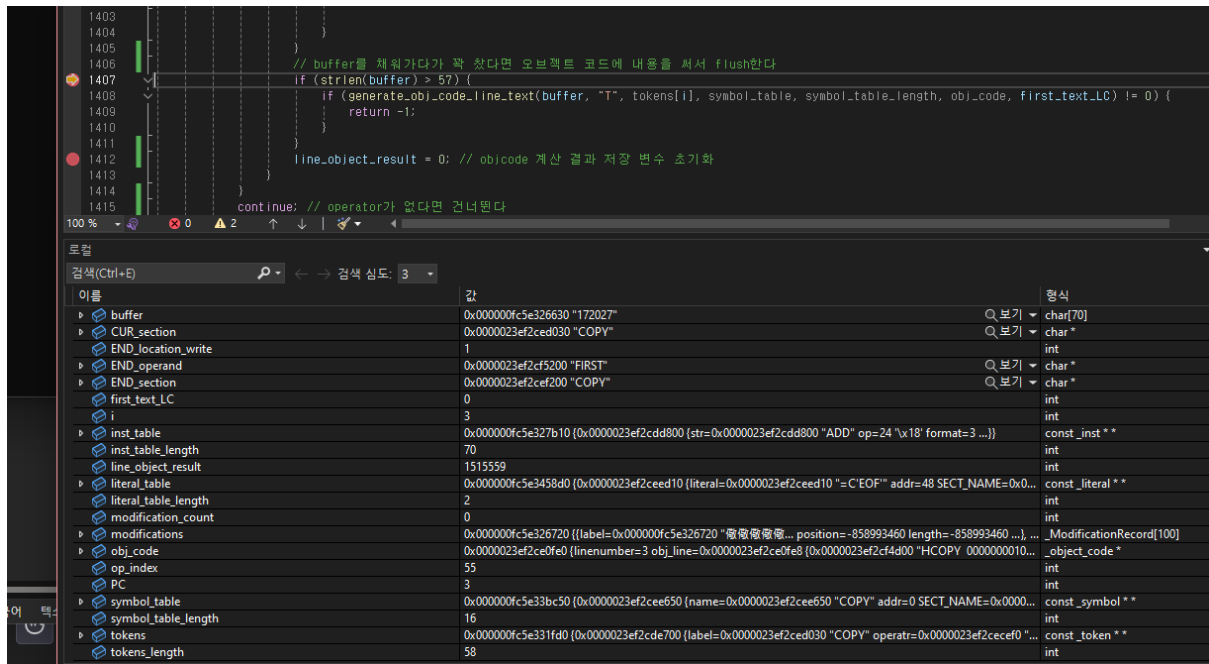


```
Project_2 (전역 범위)
85     return -1;
86   }
87   +/
88
89   if ((err = make_symbol_table_output(NULL, // "output_symtab.txt",
90                                     (const symbol **)symbol_table,
91                                     symbol_table_length)) < 0) {
92     fprintf(stderr,
93             "make_symbol_table_output: 심볼테이블 파일 출력 과정에서 "
```

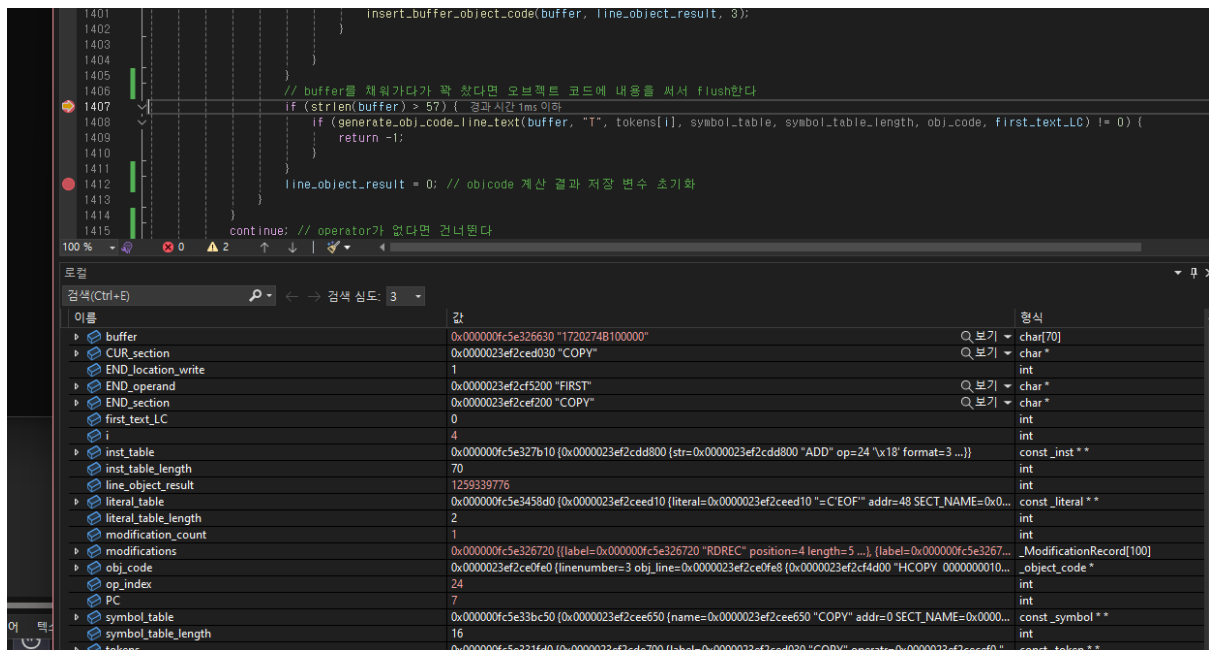
```
Microsoft Visual Studio 디버그
COPY      0
FIRST     0      +1    COPY
CLOOP     3      +1    COPY
ENDFIL    17     +1    COPY
RETADR    2A     +1    COPY
LENGTH    2D     +1    COPY
BUFFER     33     +1    COPY
BUFEND    1033   +1    COPY
MAXLEN    1000
RDREC     0
RLOOP     9      +1    RDREC
EXIT      20     +1    RDREC
INPUT     27     +1    RDREC
MAXLEN    28     +1    RDREC
WRREC     0
WLOOP     6      +1    WRREC
=C'EOF'   30
=X'05'    1B
HCOPY     000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDRECWRREC
T0000001D1720274B1000000320232900003320074B1000003F2FEC0320160F2016
T00001D0D0100030F200A4B1000003E2000
T00003003454F46
M00000405+RDREC
M00001105+WRREC
M00002405+WRREC
E0000000
HRDREC 00000000002B
RBUFFERLENGTHBUFEND
1>C:\Users\wsushi\Desktop\Systemprogramming\Project_2\my_assembler_20171969.c(1492,40): warning C4267: '초기화 중':
1>C:\Users\wsushi\Desktop\Systemprogramming\Project_2\my_assembler_20171969.c(1557,70): warning C4244: '초기화 중':
1>Project_2.vcxproj -> C:\Users\wsushi\Desktop\Systemprogramming\Project_2\x64\Debug\Project_2.exe
1>"Project_2.vcxproj" 프로젝트를 빌드했습니다.
===== 빌드: 1개 성공, 0개 실패, 0개 최신 상태, 0개 건너뛴 =====
===== 빌드이(가) 오후 9:16에 완료되었으며, 00.889 초이(가) 걸림 =====
```





중단점을 설정하고 buffer의 내용을 추적하고 있다.



중단점 설정으로 buffer에 명령어에 해당하는 기계어 코드가 하나씩 쌓여가는 것을 볼 수 있다.

실행환경 :

윈도우 11 / Visual Studio 2022

4. 결론 및 보충할 점

본 어셈블러 프로그램은 Contorls Sections로 작성된 SIC/XE 어셈블리어 소스코드 파일을 입력 받아, 기계어 목록파일을 통해 기계어 테이블을 만들고, 소스코드를 한 줄 씩 토큰나이징 하여 관리하고, 해당 라인마다 로케이션 카운터를 적용하여 저장하고, 심볼테이블과 리터럴테이블을 작성, 관리하여 결론적으로 실행가능한 기계어 코드로의 전환을 지원한다.

기본적인 SIC/XE 명령어 세트를 지원하며, 다양한 어셈블리어 코드에 대해 정확하게 오브젝트 코드를 생성할 수 있다. 그러나 현재 구현에서는 몇 가지 제한 사항이 있으며, 다음과 같은 개선이 필요하다.

확장성: 현재는 고정된 명령어 세트만을 지원한다. 사용자 정의 명령어나 형식의 확장을 위한 개선을 고려해 볼 수 있다.

assem pass2의 비효율성과 피연산자 파싱 : 이미 assem pass 1에서 판단한 operand 연산 파싱 (BUFEND-BUFEND)를 불필요하게 한번 더 수행한다. -> 토큰 파싱 과정에서 이 마저도 성공적으로 파싱하고, 무슨 연산자를 사용했는지 판단할 수 있는 디렉티브를 부여한다면 더 깔끔한 Pass1, Pass2의 디자인이 될 것이다.

사용자와의 상호작용 : 현재 작업경로에 있는 input.txt와 inst_table.txt를 사용하고 있는데, 이들이 같은 경로에 있지 않아도 괜찮도록 수정할 수 있을 듯하다. 또한, 출력을 파일로 하는 게 아니라, 간단히 보기위해 표준출력으로 하고자 하면, NULL이 인자로 들어가도록 직접 소스코드를 바꾸어야 하는데, 이를 간단하게 사용자와의 상호작용으로 바꾼다면 실행 결과를 확인하기 더 용이할 것이다.

5. 소스코드 (+주석)

my_assembler_20171969.c

```
#define _CRT_SECURE_NO_WARNINGS
```

```
#pragma warning(disable:4996)
```

```
#pragma warning(disable:6031)
```

```
#pragma warning(disable:6262)
```

```
#pragma warning(disable:6054)
```

```
#pragma warning(disable:4819)
```

```
// 윈도우 visual studio 오류 방지
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <fcntl.h>
```

```
#include <ctype.h>
```

```
// isdigit()
```

```
#include "my_assembler_20171969.h"
```

```
int main(int argc, char **argv) {
```

```
    // source by Enoch Jung(github.com / enochjung)
```

```
    /** SIC/XE 머신의 instruction 정보를 저장하는 테이블 */
```

```
    inst *inst_table[MAX_INST_TABLE_LENGTH];
```

```
    int inst_table_length = 0; // length 초기화
```

```

/** SIC/XE 소스코드를 저장하는 테이블 */

char *input[MAX_INPUT_LINES];

int input_length = 0; // length 초기화


/** 소스코드의 각 라인을 토큰 전환하여 저장하는 테이블 */

token *tokens[MAX_INPUT_LINES];

int tokens_length = 0; // length 초기화


/** 소스코드 내의 심볼을 저장하는 테이블 */

symbol *symbol_table[MAX_TABLE_LENGTH];

int symbol_table_length = 0; // length 초기화


/** 소스코드 내의 리터럴을 저장하는 테이블 */

literal *literal_table[MAX_TABLE_LENGTH];

int literal_table_length = 0; // length 초기화


/** 오브젝트 코드를 저장하는 변수 */

object_code *obj_code = (object_code *)malloc(sizeof(object_code));


int err = 0; // 0 : 에러 없음


if ((err = init_inst_table(inst_table, &inst_table_length,
                           "inst_table.txt")) < 0) {

    fprintf(stderr,
            "init_inst_table: 기계어 목록 초기화에 실패했습니다. "
            "(error_code: %d)\n",

```

```

        err);

    return -1;
}

// inst_table : 이름 / op / format / ops
// 주의 : ops는 개수가 아니어서 케이스별로 다뤄야함

if ((err = init_input(input, &input_length, "input.txt")) < 0) {

    fprintf(stderr,

        "init_input: 소스코드 입력에 실패했습니다. (error_code: %d)\n",

        err);

    return -1;
}

// input.txt 한줄씩 읽어서 input에 저장

if ((err = assem_pass1((const inst **)inst_table, inst_table_length,

    (const char **)input, input_length, tokens,

    &tokens_length, symbol_table, &symbol_table_length,

    literal_table, &literal_table_length)) < 0) {

    fprintf(stderr,

        "assem_pass1: 패스1 과정에서 실패했습니다. (error_code: %d)\n",

        err);

    return -1;
}

// token table / symbol table / literal table 생성하는 PASS1

// token table에 해당 라인의 Location Counter와 적절한 nixbpe를 저장한다.

```

```

/*
if ((err = make_opcode_output(NULL, (const token **)tokens, tokens_length,
                               (const inst **)inst_table,
                               inst_table_length)) < 0) {

    fprintf(stderr,
        "make_opcode_output: opcode 파일 출력 과정에서 실패했습니다. "
        "(error_code: %d)\n",
        err);

    return -1;
}
*/

```

```

if ((err = make_symbol_table_output("output_symtab.txt",
                                     (const symbol **)symbol_table,
                                     symbol_table_length)) < 0) {

    fprintf(stderr,
        "make_symbol_table_output: 심볼테이블 파일 출력 과정에서 "
        "실패했습니다. (error_code: %d)\n",
        err);

    return -1;
}

```

```

if ((err = make_literal_table_output("output_littab.txt",
                                      (const literal **)literal_table,
                                      literal_table_length)) < 0) {

    fprintf(stderr,

```

```

        "make_literal_table_output: 리터럴테이블 파일 출력 과정에서 "

        "실패했습니다. (error_code: %d)\n",

        err);

    return -1;

}

// 심볼과 리터럴 테이블 출력하여 결과 확인 - 인자 NULL : stdout

if ((err = assem_pass2((const token **)tokens, tokens_length,

                        (const inst **)inst_table, inst_table_length,

                        (const symbol **)symbol_table, symbol_table_length,

                        (const literal **)literal_table,

                        literal_table_length, obj_code)) < 0) {

    fprintf(stderr,

            "assem_pass2: 패스2 과정에서 실패했습니다. (error_code: %d)\n",

            err);

    return -1;

}

// PASS 2는 라인마다 실제 objcode를 작성한다.

if ((err = make_objectcode_output("output_objectcode.txt",

                                  (const object_code *)obj_code)) < 0) {

    fprintf(stderr,

            "make_objectcode_output: 오브젝트코드 파일 출력 과정에서 "

            "실패했습니다. (error_code: %d)\n",

            err);

```

```

        return -1;
    }

// 1. 명령어 테이블 내의 각 인스턴스 해제
for (int i = 0; i < inst_table_length; i++) {
    free(inst_table[i]); // 메모리 해제
    inst_table[i] = NULL; // 포인터 초기화
}

// 2. 입력 라인 해제
for (int i = 0; i < input_length; i++) {
    free(input[i]);
    input[i] = NULL;
}

// 3. 토큰 테이블 해제
for (int i = 0; i < tokens_length; i++) {
    free(tokens[i]->comment); // 메모리 해제
    tokens[i]->comment = NULL; // 포인터 초기화

    free(tokens[i]->label);
    tokens[i]->label = NULL;

    free(tokens[i]->operatr);
    tokens[i]->operatr = NULL;
}

```

```

for (int j = 0; j < 3; j++) {

    if (tokens[i]->operand[j]) { // operand 존재 시

        free(tokens[i]->operand[j]);

        tokens[i]->operand[j] = NULL;

    }

}

free(tokens[i]); // tokens[i] 전체를 마지막에 해제

tokens[i] = NULL; // 포인터 초기화

}

```

// 4. 심볼 테이블 해제

```

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

    free(symbol_table[i]);

    symbol_table[i] = NULL;

}

```

// 5. 리터럴 테이블 해제

```

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

    free(literal_table[i]);

    literal_table[i] = NULL;

}

```

// 6. 오브젝트 코드 변수 해제

```

for (int i = 0; i < obj_code->linenumber; i++) {

    free(obj_code->obj_line[i]);

}

```

```

        obj_code->obj_line[i] = NULL;

    }

    free(obj_code); // 자체 해제

    obj_code = NULL;

    return 0;

}

/**
 * @brief 기계어 목록 파일(inst_table.txt)을 읽어 기계어 목록
 * 테이블(inst_table)을 생성한다.
 *
 * @param inst_table 기계어 목록 테이블의 시작 주소
 * @param inst_table_length 기계어 목록 테이블의 길이를 저장하는 변수 주소
 * @param inst_table_dir 기계어 목록 파일 경로
 * @return 오류 코드 (정상 종료 = 0)
 *
 * @details
 * 기계어 목록 파일(inst_table.txt)을 읽어 기계어 목록 테이블(inst_table)을
 * 생성한다. 기계어 목록 파일 형식은 자유롭게 구현한다. 예시는 다음과 같다.
 *
 * =====
 *      | 이름 | 형식 | 기계어 코드 | 오퍼랜드의 갯수 | %n |
 *
 * =====
 */

int init_inst_table(inst *inst_table[], int *inst_table_length,

```

```

        const char *inst_table_dir) {

    /** 지시어도 inst_table.txt에 포함 */

    FILE *fp;

    int err;


    char buffer[20];


    if ((fp = fopen(inst_table_dir, "r")) == NULL) return ERR_FILE_IO_FAIL;


    while (!feof(fp)) {

        fgets(buffer, 20, fp);

        err = add_inst_to_table(inst_table, inst_table_length, buffer);

        if (err != 0) {

            fclose(fp);

            return err;

        }

    }


    if (fclose(fp) != 0) return ERR_FILE_IO_FAIL;

    err = 0;


    return 0;

}

/**
 * @brief inst_table.txt의 라인 하나를 입력으로 받아, 해당하는 instruction

```

* 정보를 inst_table에 저장함.

*/

```
static int add_inst_to_table(inst *inst_table[], int *inst_table_length,
```

```
    const char *buffer) {
```

```
    char name[10];
```

```
    char ops[10];
```

```
    int format;
```

```
    char op[10];
```

```
    if (*inst_table_length == MAX_INST_TABLE_LENGTH) return ERR_ARRAY_OVERFLOW;
```

```
    sscanf(buffer, "%s %s %d %s%Wn", name, ops, &format, op);
```

```
    if ((inst_table[*inst_table_length] = (inst *)malloc(sizeof(inst))) == NULL)
```

```
        return ERR_ALLOCATION_FAIL;
```

```
    memcpy(inst_table[*inst_table_length]->str, name, 9);
```

```
    inst_table[*inst_table_length]->str[9] = 'W0';
```

```
    /** !제 마음대로 저장했습니다. 여기선 ops가 operand 개수가 아닙니다! */
```

```
    if (ops[0] == '-')
```

```
        inst_table[*inst_table_length]->ops = 0; // ex : CSECT / LORG / RSUB -> operand 0개
```

```
    else if (ops[0] == 'M') // 일반적인 연산 ADD / EQU / START
```

```
        inst_table[*inst_table_length]->ops = 1; // 피연산자는 M -> operand 1개
```

```
    else if (ops[0] == 'N') // SVC
```

```
        inst_table[*inst_table_length]->ops = 3; // 피연산자는 N -> operand 1개
```

```

else if (ops[1] == 'R') // ADDR COMR DIVR RMO

    inst_table[*inst_table_length]->ops = 4; // 피연산자는 RR -> operand 2개

else if (ops[1] == 'N') // SHIFTL SHIFTR

    inst_table[*inst_table_length]->ops = 5; // 피연산자는 RN -> operand 2개

else

    inst_table[*inst_table_length]->ops = 2; // unknown, R -> operand 0개 / R하나 쓰는 2형
식

inst_table[*inst_table_length]->format = format;

inst_table[*inst_table_length]->op = (unsigned char)strtol(op, NULL, 16);

++(*inst_table_length);

return 0;
}

/**
 * @brief SIC/XE 소스코드 파일(input.txt)을 읽어 소스코드 테이블(input)을
 * 생성한다.
 *
 * @param input 소스코드 테이블의 시작 주소
 * @param input_length 소스코드 테이블의 길이를 저장하는 변수 주소
 * @param input_dir 소스코드 파일 경로
 * @return 오류 코드 (정상 종료 = 0)
 */

```

```

int init_input(char *input[], int *input_length, const char *input_dir) {

    FILE *fp;

    char buffer[250];

    int length;

    if ((fp = fopen(input_dir, "r")) == NULL) return ERR_FILE_IO_FAIL;

    while (!feof(fp)) {

        if (fgets(buffer, 249, fp) == NULL) break;

        buffer[249] = '\0';

        length = (int)strlen(buffer);

        if ((input[*input_length] = (char *)malloc(length + 1)) == NULL) {

            fclose(fp);

            return ERR_ALLOCATION_FAIL;

        }

        sscanf(buffer, "%[^\r\n]", input[*input_length]);

        ++(*input_length);

    }

    if (fclose(fp) != 0) return ERR_FILE_IO_FAIL;

    return 0;

}

```

/**

* @brief 어셈블리 코드를 위한 패스 1 과정을 수행한다.

*

* @param inst_table 기계어 목록 테이블의 주소

* @param inst_table_length 기계어 목록 테이블의 길이

* @param input 소스코드 테이블의 주소

* @param input_length 소스코드 테이블의 길이

* @param tokens 토큰 테이블의 시작 주소

* @param tokens_length 토큰 테이블의 길이를 저장하는 변수 주소

* @param symbol_table 심볼 테이블의 시작 주소

* @param symbol_table_length 심볼 테이블의 길이를 저장하는 변수 주소

* @param literal_table 리터럴 테이블의 시작 주소

* @param literal_table_length 리터럴 테이블의 길이를 저장하는 변수 주소

* @return 오류 코드 (정상 종료 = 0)

*

* @details

* 어셈블리 코드를 위한 패스1 과정을 수행하는 함수이다. 패스 1에서는 프로그램

* 소스를 스캔하여 해당하는 토큰 단위로 분리하여 프로그램 라인별 토큰 테이블을

* 생성한다. 토큰 테이블은 token_parsing 함수를 호출하여 설정하여야 한다. 또한,

* assem_pass2 과정에서 사용하기 위한 심볼 테이블 및 리터럴 테이블을 생성한다.

*/

/*

일단 첫 패스에서 한줄씩 읽을때 Location Counter부터 해야한다

CSECT 만나면 Location Counter 0 으로 초기화

문장의 Label을 심볼테이블에 저장

nixbpe의 세팅

n : indirect - @

i : immediate -

그 두개 확인 못하면 n, i = 1

x : operand 에 X가 있으면 1

b : BASE relative

p : PC relative

e : +만나면 1로 세팅

X X N I X B P E

32 16 8 4 2 1

target address 계산 : operand를 보고(SYMTAB) 2pass 에서

4형식은 00000 채워놓고 나중에 modification record

*/

```
int assem_pass1(const inst *inst_table[], int inst_table_length,
                const char *input[], int input_length, token *tokens[],
                int *tokens_length, symbol *symbol_table[],
                int *symbol_table_length, literal *literal_table[],
                int *literal_table_length) {
    int err;

    int op_index; // inst_table에서 얻은 명령어 인덱스
```

```

int LC_Value = 0; // 계속 더해져갈 Location counter

int starting_address = 0; // START 지시어의 operand값이 들어갈 변수

char CUR_SECT[10]; // 현재 섹션 이름 저장

char* CUR_EXTDEF[3] = {NULL}; // 현재 섹션이 EXTDEF하는 것들

for (int i = 0; i < input_length; ++i) { // 한 라인씩 수행

    if ((tokens[i] = (token *)malloc(sizeof(token))) == NULL) // 토큰 할당

        return ERR_ALLOCATION_FAIL;

    if ((err = token_parsing(input[i], tokens[i], inst_table, inst_table_length)) != 0) // 토큰 파싱
수행 by Enoch Jung(github.com / enochjung)

        return err;

    // 현재 토큰 할당 되었음

    // 라인마다 LC, nixbpe 할당 시작

    // 그리고 심볼, 리터럴 테이블을 생성한다.

    if (input[i][0] != '.') { // 주석 아니면 스타트 | 주석이면 생략

        if (tokens[i]->operatr != NULL) { // 현재 라인의 operatr가 존재함

            if (strcmp("START", tokens[i]->operatr) == 0) { // START의 경우 처리

                starting_address = strtol(tokens[i]->operand[0], NULL, 16); // strtol을 사용하
여 시작주소를 숫자로 저장 (16진수로)

                LC_Value = starting_address; // LC를 시작주소로 업데이트

                tokens[i]->LC = LC_Value; // token의 LC에 할당

                if (!tokens[i]->label) { // 심볼 없는 경우의 오류 처리

                    fprintf(stderr, "START operator have no LABEL (error_code: %d)\n",

```

```

ERR_MISSING_LABEL);

        return ERR_MISSING_LABEL;
    }

    strncpy(CUR_SECT, tokens[i]->label, sizeof(CUR_SECT) - 1); // 현재 섹션에
START의 LABEL 저장

    CUR_SECT[sizeof(CUR_SECT) - 1] = '\0';
}

else if (strcmp("CSECT", tokens[i]->operatr) == 0) { // CSECT의 경우 처리

    if ((err = ORG_LITTAB(CUR_SECT, literal_table, *literal_table_length,
&LC_Value)) != 0) {

        return err;
    }

    // 현재 섹션까지 나왔던 모든 리터럴을 테이블에 할당한다.
// 리터럴 - 각 섹션이 끝나고 할당 - 각 섹션은 하나의 독립적인 프로그램
으로 보기 때문에

    // 섹션이 끝날 때 리터럴 할당을 실시 하였음.

    //

    // 리터럴 테이블에는 - 현재 섹션 LABEL, LC값도 함께 포함한다.
    if (err = insert_SECTLEN(CUR_SECT, symbol_table, *symbol_table_length,
LC_Value)) {

        return err;
    }

    // 현재의 섹션의 총 길이를 심볼테이블에 삽입한다.
// ex) COPY의 총크기를 현재 LC로 저장 : 왜냐하면 이제 새로운 section으
로 끊기 때문

    LC_Value = 0; // LC 초기화 - 새로운 섹션임

    if (!tokens[i]->label) { // 심볼 없는 경우의 오류 처리

        fprintf(stderr, "CSECT operator have no LABEL (error_code: %d)\n",

```

ERR_MISSING_LABEL);

return ERR_MISSING_LABEL;

}

strncpy(CUR_SECT, tokens[i]->label, sizeof(CUR_SECT) - 1); // 현재 섹션의 현재 LABEL 저장

CUR_SECT[sizeof(CUR_SECT) - 1] = '\0';

for (int j = 0; j < 3; j++) { // !! EXTDEF 최대 3개로 제한된 구현법 -> EXTDEF 4개 이상 시 버그 가능성 있음

if (CUR_EXTDEF[j]) { // 이전 섹션에서 할당했던 EXTDEF가 있다면

free(CUR_EXTDEF[j]); // 메모리 해제

CUR_EXTDEF[j] = NULL; // 포인터를 NULL로 재설정

}

}

}

else if (strcmp("EXTDEF", tokens[i]->operatr) == 0) { // EXTDEF의 경우 처리

for (int j = 0; j < 3; j++) { // 현재 섹션에서 DEF한 LABEL들을 최대 3개 저장

CUR_EXTDEF[j] = malloc(strlen(tokens[i]->operand[j]) + 1); // 메모리 할

당

if (CUR_EXTDEF[j]) {

strcpy(CUR_EXTDEF[j], tokens[i]->operand[j]); // 복사

}

}

}

else if (strcmp("LTORG", tokens[i]->operatr) == 0) { // LTORG의 경우 처리

if ((err = ORG_LITTAB(CUR_SECT, literal_table, *literal_table_length, &LC_Value)) != 0) {

```

        return err;

    }

    // LITORG 기능 수행하는 함수를 호출

    // 리터럴 테이블에는 - 현재 섹션 LABEL, LC값도 함께 포함한다.

}

else if (strcmp("END", tokens[i]->operatr) == 0) { // END의 경우 처리

    if ((err = ORG_LITTAB(CUR_SECT, literal_table, *literal_table_length,
&LC_Value)) != 0) {

        return err;

    }

    // LITORG 기능 수행하는 함수를 호출

    if (insert_SECLen(CUR_SECT, symbol_table, *symbol_table_length, LC_Value) <
-1) {

        return -1;

    }

    // 현재 섹션의 길이를 심볼 테이블에 저장하는 함수 수행 - 나중에 H에 사
용할거임

}

} // 지시어 처리 끝

if (tokens[i]->label != NULL) { // LABEL 있는 경우 - SYMTAB에 추가

    // 중복 심볼 검사

    int SECT_flag = 1; // 1이면 소속 섹션 이름 적음 -> 해당 레이블은 소속 섹션의
시작 주소가 더해져야 구할 수 있다는 것임

    if (tokens[i]->operatr != NULL) { // operator가 있어서 검사

        if (strcmp("EQU", tokens[i]->operatr) == 0) { // EQU 인 경우

            for (int j = 0; j < 3; j++) {

```

```

    if (CUR_EXTDEF[j] && (strcmp(tokens[i]->label, CUR_EXTDEF[j]) ==
0)) {

        SECT_flag = 1; // EXTDEF 된 EQU라면 소속 영역 적어야 함
        // 왜냐하면 외부에서 위치 찾아오려면 이 심볼의 섹션 시작주
소를 더해야 하기 때문. -> +1

        break;

    }

    else SECT_flag = 0; // EXTDEF 안됐다면 안적어도 됨
}

int EQU_value = 0; // EQU에 들어갈 값

if (strcmp("**", tokens[i]->operand[0]) == 0) { // *면 LC 그대로 넣는다.
    EQU_value = LC_Value;
}

else { // tokens[i]->operand[0] : BUFFER-BUFEND 의 경우

    char* operand1; // BUFFER 담길 것

    char* operand2; // BUFEND 담길 것


    char* minus_pos = strchr(tokens[i]->operand[0], '-');

    // tokens[i]->operand[0]:BUFFER-BUFEND 를 BUFFER랑 BUFEND로
자른다.


    if (minus_pos == NULL) { // '-' 못찾으면 에러처리

        fprintf(stderr, "EQU : '-' not found in the operand
(error_code: %d)\n", ERR_MISSING_OPERAND);

        return ERR_MISSING_OPERAND;

    }

```

이 계산

BUFFER

시작

```
// 첫 번째 피연산자 처리

int length1 = minus_pos - tokens[i]->operand[0]; // 첫 피연산자 길

operand1 = (char*)malloc(length1 + 1); // 메모리 할당

if (operand1 == NULL) { // 할당 실패 에러 처리

    fprintf(stderr, "EQU : Memory allocation failed for operand1
(error_code: %d)\n", ERR_ALLOCATION_FAIL);

    return ERR_ALLOCATION_FAIL;

}

strncpy(operand1, tokens[i]->operand[0], length1); // operand1 :

operand1[length1] = '\0'; // 널 종료 문자 추가

// 두 번째 피연산자 처리

char* start_of_second_part = minus_pos + 1; // '-' 다음 문자부터

int length2 = strlen(start_of_second_part);

operand2 = (char*)malloc(length2 + 1);

if (operand2 == NULL) {

    fprintf(stderr, "EQU : Memory allocation failed for operand2
(error_code: %d)\n", ERR_ALLOCATION_FAIL);

    free(operand1); // 첫 번째 피연산자 메모리 해제

    return ERR_ALLOCATION_FAIL;

}

strcpy(operand2, start_of_second_part); // operand2 : BUFEND

// EQU_value 계산
```

```

        int    index1    =    search_SYMTAB(operand1,    symbol_table,
*symbol_table_length);

        int    index2    =    search_SYMTAB(operand2,    symbol_table,
*symbol_table_length);

        if (index1 == -1 || index2 == -1) { // 둘 중 하나라도 심볼 정의 안
되어 있다면 에러 처리

                fprintf(stderr, "EQU : SYMBOL not found (error_code: %d)\n",
ERR_SYMBOL_NOT_FOUND);

                free(operand1);

                free(operand2);

                return ERR_SYMBOL_NOT_FOUND;

        }

        EQU_value = symbol_table[index1]->addr - symbol_table[index2]-
>addr; // 저장될 EQU_VALUE

        // 사용 후 메모리 해제

        free(operand1);

        free(operand2);

    }

    insert_SYMTAB(tokens[i]->label,    symbol_table,    symbol_table_length,
EQU_value, CUR_SECT, SECT_flag);

    // SYMTAB에 추가

}

else { // EQU 아닌 경우 - 단순 삽입

    SECT_flag = 1;

    insert_SYMTAB(tokens[i]->label,    symbol_table,    symbol_table_length,

```

```
LC_Value, CUR_SECT, SECT_flag);
```

```
}
```

```
}
```

```
} // 심볼테이블 삽입 끝
```

```
// # 여기서 opcode search, LC의 크기를 판단한다
```

```
// # 오퍼레이터를 판정하면 됨.
```

```
// # 오퍼레이터 종류는??? tokens[i]->operatr 로 search opcode : 그럼 인덱스 나
```

옴

```
// # 그 인덱스 가지고 inst_table[index] 의 FORMAT 확인
```

```
// # 4형식은 tokens[i]->operatr[0] == '+' : 4만큼 증가
```

```
tokens[i]->LC = LC_Value; // LC 증가시키기 전에 token에 현재 LC 할당
```

```
if (tokens[i]->operatr != NULL) { // 오퍼레이터가 있다면
```

```
op_index = search_opcode(tokens[i]->operatr, inst_table, inst_table_length); //
```

opcode 찾아보기

```
if (op_index != -1) { // 존재하는 opcode라면 !! LC 계산 !! nixbpe 계산 !!
```

```
if (inst_table[op_index]->format == 2) { // 2형식
```

```
LC_Value += 2; // LC 2B 증가
```

```
tokens[i]->nixbpe = 0; // 2형식은 nixbpe 쓸모 없음
```

```
}
```

```
else if (inst_table[op_index]->format == 3) { // 3형식
```

```
tokens[i]->nixbpe = 0; // nixbpe 초기화
```

```
LC_Value += 3; // LC 3B 증가
```

```
tokens[i]->nixbpe += 2; // 기본값 : p 세팅
```

```

if (tokens[i]->operatr[0] == '+') { // 4형식

    LC_Value += 1; // 4형식은 1B 추가

    tokens[i]->nixbpe += 1; // 4형식 - e 세팅

    tokens[i]->nixbpe -= 2; // 4형식 - p 빼기 세팅

    // nix001

}

if ((inst_table[op_index]->ops == 1) || (inst_table[op_index]->ops == 3))
{ // ops 1,3은 operand가 1개

    if (tokens[i]->operand[1]) { // 그런데 operand가 하나 더 있다면 X
가 존재

        tokens[i]->nixbpe += 8; // X 세팅

    }

}

else if ((inst_table[op_index]->ops == 4) || (inst_table[op_index]->ops ==
5)) { // ops 4, 5는 operand가 2개

    if (tokens[i]->operand[2]) { // 그런데 operand가 3개라면 X 가 존재

        tokens[i]->nixbpe += 8; // 3형식 - x 세팅

    }

}

else { // RSUB / ops2 unknown -> operand가 없음

    tokens[i]->nixbpe += 48; // n, i 세팅 -> SIC/XE 머신

    tokens[i]->nixbpe -= 2; // p 빼기 세팅

}

}

else if (inst_table[op_index]->format == 0) { // 지시어들은 format 0

    // 지시어들의 LC 증가 폭을 계산

```

```

        if (strcmp("RESB", inst_table[op_index]->str) == 0) {

            LC_Value += (int)strtol(tokens[i]->operand[0], NULL, 10); // input.txt
입력은 10진수로 받았음

        } // 입력 값만큼 Byte 차지 -> 그만큼 LC 증가

        else if (strcmp("RESW", inst_table[op_index]->str) == 0) {

            LC_Value += 3 * (int)strtol(tokens[i]->operand[0], NULL, 10);

        } // WORD라서 입력값 * 3 만큼 차지 -> 그만큼 LC 증가

        else if (strcmp("BYTE", inst_table[op_index]->str) == 0) {

            LC_Value += 1;

        } // BYTE는 1B 증가

        else if (strcmp("WORD", inst_table[op_index]->str) == 0) {

            LC_Value += 3;

        } // WORD는 3B 증가

    }

}

else { // operator 못찾아서 인덱스가 -1이면 에러 처리

    fprintf(stderr, "Search opcode : operator not found (error_code: %d)\n",
ERR_ILLEGAL_OPERATOR);

    return ERR_ILLEGAL_OPERATOR;

}

}

if (tokens[i]->operand[0]) { // 첫 operand가 존재한다면

    if (tokens[i]->operand[0][0] == '#') { // immediate 라면

        tokens[i]->nixbpe += 16; // immediate 세팅 01XBPE

        tokens[i]->nixbpe -= 2; // immediate 는 p 빼줌

    }

```

```

else if (tokens[i]->operand[0][0] == '@') { // indirect라면

    tokens[i]->nixbpe += 32; // indirect 세팅 10XBPE

}

else {

    if (tokens[i]->operand[0][0] == '=') { // 리터럴이라면 리터럴 테이블 삽입

        if (search_LITTAB(tokens[i]->operand[0], literal_table, *literal_table_length)
== -1) { // 리터럴 테이블에서 못 찾았으면

            insert_LITTAB(tokens[i]->operand[0], literal_table, literal_table_length);
// 테이블에 삽입

        }

    }

    tokens[i]->nixbpe += 48; // 11XBPE 세팅

}

}

}

}

```

// 모든 라인 토큰 할당, LC 할당, nixbpe할당, SYMTAB, LITTAB 작성 완료

for (int j = 0; j < 3; j++) { // 배열의 크기만큼 반복

if (CUR_EXTDEF[j]) {

free(CUR_EXTDEF[j]); // 메모리 해제

CUR_EXTDEF[j] = NULL; // 포인터를 NULL로 재설정

}

} // 라인 체크 반복문 안에서 사용한 것 남아있다면 할당 해제

*tokens_length = input_length; // token length도 문장 수만큼 - 주석도 한 토큰으로 판단

```
    return 0;
}
```

```
static int search_SYMTAB(const char* str, symbol* symbol_table[], int symbol_table_length) {
    // 심볼테이블을 주어진 str과 비교해보고 인덱스를 리턴한다.
    for (int i = 0; i < symbol_table_length; ++i) {
        if (strcmp(str, symbol_table[i]->name) == 0) return i;
    }
    return -1;
}
```

```
static int search_SYMTAB_CUR_section(const char* str, symbol* symbol_table[], int
symbol_table_length, const char* CUR_section) {
    // 섹션마다 이름이 같은 심볼이 있을 수 있기 때문에 현재 섹션 정보도 확인하여, 현재 섹션
    // 에 있는 경우에만 인덱스를 리턴한다.
    for (int i = 0; i < symbol_table_length; ++i) {
        if ((strcmp(str, symbol_table[i]->name) == 0) && (strcmp(CUR_section, symbol_table[i]-
>SECT_NAME) == 0)) return i;
    }
    return -1;
}
```

```
static int insert_SYMTAB(const char* str, symbol* symbol_table[], int *symbol_table_length, int LC,
const char* CUR_SECT, int SECT_flag) {
    // 주어진 str과 LC, 현재 섹션이름, 섹션 플래그를 토대로 심볼을 심볼테이블에 삽입한다.
    if ((symbol_table[*symbol_table_length] = (symbol*)malloc(sizeof(symbol))) == NULL) return
ERR_ALLOCATION_FAIL;
    // 메모리 할당 오류시 에러값 리턴
```

```

size_t len = strlen(str);

if (len > 9) return ERR_NAME_LENGTH_OVER; // 최대 길이 제한 에러

memcpy(symbol_table[*symbol_table_length]->name, str, len);

symbol_table[*symbol_table_length]->name[len] = '\0'; // 심볼 이름 할당

symbol_table[*symbol_table_length]->addr = LC; // LC 할당

if (strcmp(str, CUR_SECT) == 0 || SECT_flag == 0) { // 섹션 심볼이거나 SECT_flag 통해 섹션 안
적어도 된다면

    symbol_table[*symbol_table_length]->SECT_NAME[0] = '\0';

    symbol_table[*symbol_table_length]->flag_add_address = 0;

}

else { // 섹션 이름 적어야 한다면

    strncpy(symbol_table[*symbol_table_length]->SECT_NAME, CUR_SECT, 9);

    symbol_table[*symbol_table_length]->SECT_NAME[9] = '\0'; // 섹션 이름 저장하고

    symbol_table[*symbol_table_length]->flag_add_address = 1; // +1 플래그도 설정

}

symbol_table[*symbol_table_length]->section_length = 0; // 섹션의 총 길이 변수 초기화 ->
insert_SECTLEN()으로 수정

++(*symbol_table_length); // 길이 1 증가

return 0;

}

```

```

static int insert_SECTLEN(const char* str, symbol* symbol_table[], int symbol_table_length, int
Section_length) {

    int index;

```

```

// 주어진 str과 동일한 인덱스를 먼저 찾는다.

if ((index = search_SYMTAB(str, symbol_table, symbol_table_length)) == -1) {

    return -1;

}

symbol_table[index]->section_length = Section_length;

// 인덱스에 해당되는 심볼에 섹션의 길이를 저장한다

return 0;

}

static int search_LITTAB(const char* str, literal* literal_table[], int literal_table_length) {

    // 주어진 str과 동일한 literal 테이블의 인덱스를 찾는다.

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

        if (strcmp(str, literal_table[i]->literal) == 0) return i;

    }

    return -1;

}

static int insert_LITTAB(const char* str, literal* literal_table[], int *literal_table_length) {

    // 리터럴 테이블에 주어진 str을 토대로 삽입한다.

    if ((literal_table[*literal_table_length] = (literal*)malloc(sizeof(literal))) == NULL) return
ERR_ALLOCATION_FAIL;

    // 메모리 할당 에러 처리

    size_t len = strlen(str);

    if (len > 19) return ERR_NAME_LENGTH_OVER; // 최대 길이 제한 에러 처리

    memcpy(literal_table[*literal_table_length]->literal, str, len); // 리터럴 이름을 저장한다

```

```
literal_table[*literal_table_length]->literal[len] = '\0';
```

literal_table[*literal_table_length]->addr = -1; // 리터럴의 주소값 addr은 -1로 초기화 해놓는다 -> ORG_LITTAB()으로 수정

```
++(*literal_table_length); // 길이 1 증가
```

```
return -1;
```

```
}
```

```
static int ORG_LITTAB(const char* str, literal* literal_table[], int literal_table_length, int *LC) {
```

```
// LITTAB의 있는 리터럴들을 메모리 할당한다.
```

```
for (int i = 0; i < literal_table_length; ++i) { // 테이블 길이 (저장된 갯수) 만큼 반복문을 돈다
```

```
int literal_size = 0; // 해당 리터럴의 사이즈 (Byte 기준)
```

if (literal_table[i]->addr == -1) { // 만약 해당 리터럴의 addr가 -1로 아직 할당이 안되어 있다면

```
literal_table[i]->addr = *LC; // 주어진 LC를 할당한다.
```

```
// 리터럴의 크기 계산
```

```
if (literal_table[i]->literal[1] == 'C') {
```

```
// 'C'로 시작하는 리터럴 ex : =C'EOF'
```

```
// 리터럴 값은 작은따옴표(') 사이에 있다
```

```
char* start = strchr(literal_table[i]->literal, '\0') + 1; // E의 위치
```

```
char* end = strchr(start, '\0'); // F 다음의 '의 위치
```

```
if (start && end) {
```

```
literal_size = end - start; // 문자 개수만큼 크기를 측정 (Byte)
```

```
}
```

```

        else return ERR_INVALID_LITERAL_FORMAT;

    }

    else if (literal_table[i]->literal[1] == 'X') {

        // 'X'로 시작하는 리터럴 ex : =X'05'

        // 리터럴 값은 작은따옴표(') 사이에 있다

        char* start = strchr(literal_table[i]->literal, 'W') + 1; // 0의 위치

        char* end = strchr(start, 'W'); // 5 다음 '의 위치

        if (start && end) {

            // 16진수는 두 문자가 1바이트를 나타낸다

            literal_size = (end - start) / 2; // 따라서 길이에 2를 나눈 값이 Byte 크기

        }

        else return ERR_INVALID_LITERAL_FORMAT;

    }

    *LC += literal_size; // 계산된 리터럴 크기만큼 LC를 증가시킨다

    strncpy(literal_table[i]->SECT_NAME, str, 9);

    // 해당 리터럴에 주어진 str로 섹션네임을 할당한다

    // 이는 꼭 필요하다기 보다 본 프로젝트 구현 편의를 위해 추가한 변수이다.

    literal_table[i]->SECT_NAME[9] = '\0';

}

else continue; // 만약 이미 addr 할당 된 리터럴이라면 생략하고 다음 리터럴 확인

}

return 0;

}

/**

```

* @brief 한 줄의 소스코드를 파싱하여 토큰에 저장한다.

*

* @param input 파싱할 소스코드 문자열

* @param tok 결과를 저장할 토큰 구조체 주소

* @param inst_table 기계어 목록 테이블의 주소

* @param inst_table_length 기계어 목록 테이블의 길이

* @return 오류 코드 (정상 종료 = 0)

*/

int token_parsing(const char *input, token *tok, const inst *inst_table[],

int inst_table_length) {

int input_length = (int)strlen(input);

// parser program by

// author Enoch Jung(github.com / enochjung)

tok->label = NULL;

tok->operatr= NULL;

tok->operand[0] = NULL;

tok->operand[1] = NULL;

tok->operand[2] = NULL;

tok->comment = NULL;

// token 구조체에 LC와 nixbpe를 추가하였다

tok->LC = 0;

tok->nixbpe = 0;

if (input[0] == '.') { // 수정 by Park Kunho 주석이라면 토큰 파싱 생략

if ((tok->comment = (char*)malloc(input_length + 1)) == NULL) // +1 : null terminator

```

        return ERR_ALLOCATION_FAIL;

// tok->comment에 . 제외하고 다 넣음
strncpy(tok->comment, input+1, input_length-1);

tok->comment[input_length-1] = '\0'; // Ensure null termination
}

else {

    int token_cnt = 0;

    for (int st = 0; st < input_length && token_cnt < 3; ++st) {

        int end = st;

        for (; input[end] != '\t' && input[end] != '\0'; ++end) ;

        // 토큰 개수 셤다

        switch (token_cnt) {

            case 0:

                if (st < end) {

                    if ((tok->label = (char *)malloc(end - st + 1)) == NULL)

                        return ERR_ALLOCATION_FAIL;

                    memcpy(tok->label, input + st, end - st);

                    tok->label[end - st] = '\0';

                }

                break;

            case 1:

                if (st < end) {

                    if ((tok->operatr=(char *) malloc(end - st + 1)) ==

                        NULL)

                        return ERR_ALLOCATION_FAIL;

```

```

        memcpy(tok->operatr, input + st, end - st);

        tok->operatr[end - st] = '\0';
    }

    break;

```

case 2: // operand 를 파싱한다.

```

    if (st < end) {

        int err;

        if ((err = token_operand_parsing(input + st, end - st,

                                          tok->operand)) != 0)

            return err;

    }

```

```

    st = end + 1;

    end = input_length;

    if (st < end) {

        if ((tok->comment = (char *)malloc(end - st + 1)) ==

            NULL)

            return ERR_ALLOCATION_FAIL;

        memcpy(tok->comment, input + st, end - st);

        tok->comment[end - st] = '\0';

    }

```

```

}

```

```

++token_cnt;

```

```

st = end;

```

```

    }

}

return 0;

}

/**
 * @brief 피연산자 문자열을 파싱하여 operand에 저장함. 문자열은 \0으로 끝나지
 * 않을 수 있기에, operand_input_length로 문자열의 길이를 전달해야 함.
 */

static int token_operand_parsing(const char *operand_input,
                                int operand_input_length, char *operand[]) {

    int operand_cnt = 0;

    for (int st = 0; st < operand_input_length; ++st) {

        int end = st;

        // 오퍼랜드는 ,기준으로 파싱한다.

        for (; operand_input[end] != ',' && operand_input[end] != '\t' &&
            operand_input[end] != '\0';
            ++end)

            ;

        switch (operand_cnt) {

            case 0:

                if ((operand[0] = (char *)malloc(end - st + 1)) == NULL)

                    return ERR_ALLOCATION_FAIL;

                memcpy(operand[0], operand_input + st, end - st);

```

```

operand[0][end - st] = '\0';

break;

case 1:

    if ((operand[1] = (char *)malloc(end - st + 1)) == NULL)

        return ERR_ALLOCATION_FAIL;

    memcpy(operand[1], operand_input + st, end - st);

    operand[1][end - st] = '\0';

    break;

case 2:

    if ((operand[2] = (char *)malloc(end - st + 1)) == NULL)

        return ERR_ALLOCATION_FAIL;

    memcpy(operand[2], operand_input + st, end - st);

    operand[2][end - st] = '\0';

    if (end != operand_input_length)

        return ERR_ILLEGAL_OPERAND_FORMAT;

    break;

}

++operand_cnt;

st = end;

}

// 결론적으로 중요한 점

// 현 파싱에서는 -기준 파싱은 수행하지 않았다

// ex) BUFEND-BUFFER 는 operand[0]에 다 담겨 있음을 주의

```

```

    return 0;
}

/**
 * @brief 기계어 목록 테이블에서 특정 기계어를 검색하여, 해당 기계어가 위치한
 * 인덱스를 반환한다.
 *
 * @param str 검색할 기계어 문자열
 * @param inst_table 기계어 목록 테이블 주소
 * @param inst_table_length 기계어 목록 테이블의 길이
 * @return 기계어의 인덱스 (해당 기계어가 없는 경우 -1)
 *
 * @details
 * 기계어 목록 테이블에서 특정 기계어를 검색하여, 해당 기계어가 위치한 인덱스를
 * 반환한다. '+JSUB'와 같은 문자열에 대한 처리는 자유롭게 처리한다.
 */
int search_opcode(const char *str, const inst *inst_table[],
                 int inst_table_length) {
    /** 함수명을 search_instruction으로 정했어야 했는데... 강을 많이 건넜네요 */

    // 만약 첫 글자가 +라서 4형식이라면 + 다음 글자부터 다시 재귀호출하여
    // 기계어 인덱스를 반환한다.
    if (str[0] == '+')
        return search_opcode(str + 1, inst_table, inst_table_length);

```

```

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

        if (strcmp(str, inst_table[i]->str) == 0) return i;

    }

    // 만약 존재하지 않는 기계어라면 -1를 리턴함에 주의

    return -1;

}

/**
 * @brief 소스코드 명령어 앞에 OPCODE가 기록된 코드를 파일에 출력한다.
 * `output_dir`이 NULL인 경우 결과를 stdout으로 출력한다. 프로젝트 1에서는
 * 불필요하다.
 *
 * @param output_dir 코드를 저장할 파일 경로, 혹은 NULL
 * @param tokens 토큰 테이블 주소
 * @param tokens_length 토큰 테이블의 길이
 * @param inst_table 기계어 목록 테이블 주소
 * @param inst_table_length 기계어 목록 테이블의 길이
 * @return 오류 코드 (정상 종료 = 0)
 *
 * @details
 * 소스코드 명령어 앞에 OPCODE가 기록된 코드를 파일에 출력한다. `output_dir`이
 * NULL인 경우 결과를 stdout으로 출력한다. 명세서에 주어진 출력 예시와 완전히
 * 동일할 필요는 없다. 프로젝트 1에서는 불필요하다.
 */
int make_opcode_output(const char* output_dir, const token* tokens[],

```

```

int tokens_length, const inst* inst_table[],

int inst_table_length) {

FILE* fp;

int err = 0;


if (output_dir == NULL)

    fp = stdout;

else if ((fp = fopen(output_dir, "w")) == NULL)

    return ERR_FILE_IO_FAIL;


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

    if ((err = write_opcode_output(fp, tokens[i], inst_table,

        inst_table_length)) != 0) {

        break;

    }

}


if (fp != stdout) {

    if (fclose(fp) != 0) return ERR_FILE_IO_FAIL;

}


return 0;

}

/**

* @brief 토큰 하나에 담긴 정보를 fp에 출력함.

```

```
*/
```

```
static int write_opcode_output(FILE* fp, const token* tok,
```

```
    const inst* inst_table[],
```

```
    int inst_table_length) {
```

```
    if (tok->label == NULL && tok->operatr == NULL) {
```

```
        fprintf(fp, ".Wt%sWn", tok->comment == NULL ? "" : tok->comment);
```

```
        return 0;
```

```
    }
```

```
        fprintf(fp, "%-8s%-8s", (tok->label != NULL ? tok->label : ""), (tok->operatr != NULL ? tok->operatr : ""));
```

```
    char buffer[50] = { 0 };
```

```
    if (tok->operand[2] != NULL)
```

```
        sprintf(buffer, "%s,%s,%s", tok->operand[0], tok->operand[1],
```

```
            tok->operand[2]);
```

```
    else if (tok->operand[1] != NULL)
```

```
        sprintf(buffer, "%s,%s", tok->operand[0], tok->operand[1]);
```

```
    else if (tok->operand[0] != NULL)
```

```
        sprintf(buffer, "%s", tok->operand[0]);
```

```
    fprintf(fp, "%-26s", buffer);
```

```
    if (tok->operatr != NULL) {
```

```
        int inst_idx =
```

```
            search_opcode(tok->operatr, inst_table, inst_table_length);
```

```

    if (inst_idx < 0) return ERR_ILLEGAL_OPERATOR;

    if (inst_table[inst_idx]->format > 0) {

        unsigned char op = inst_table[inst_idx]->op;

        fprintf(fp, "%02X", op);

    }

    else

        fprintf(fp, " ");

}

fprintf(fp, "    %s\n", tok->comment != NULL ? tok->comment : "");

return 0;

}

/**
 * @brief 어셈블리 코드를 위한 패스 2 과정을 수행한다.
 *
 * @param tokens 토큰 테이블 주소
 * @param tokens_length 토큰 테이블 길이
 * @param inst_table 기계어 목록 테이블 주소
 * @param inst_table_length 기계어 목록 테이블 길이
 * @param symbol_table 심볼 테이블 주소
 * @param symbol_table_length 심볼 테이블 길이
 * @param literal_table 리터럴 테이블 주소
 * @param literal_table_length 리터럴 테이블 길이

```

* @param obj_code 오브젝트 코드에 대한 정보를 저장하는 구조체 주소

* @return 오류 코드 (정상 종료 = 0)

*

* @details

* 어셈블리 코드를 기계어 코드로 바꾸기 위한 패스2 과정을 수행한다. 패스 2의

* 프로그램을 기계어로 바꾸는 작업은 라인 단위로 수행된다.

*/

/* 라인단위로 수행한다.

*

* 일단, 첫 라인 수행

* char buffer[70] : 오브젝트 코드가 쓰여질 버퍼이다

*

1. START / CSECT 나오면 헤더 H라인 작성

2. EXTDEF 보고 D라인 작성

3. EXTREF 보고 R라인 작성

4. T라인 작성

-> T 시작주소 (첫 라인)

버퍼를 두고-> 콇 차거나 / 중간에 공간이 생겨버리면 -> 줄바꾸기

5. E라인 작성 -> CSECT 나오면 그 이전까지를 토대로 작성

그런데, E이후에 나오는 06X는 END FIRST 라인을 보고나서

FIRST가 속해있는 SECTION (지금의 예제는 COPY 섹션)만

FIRST의 address를 출력하도록 한다.

그러므로, 반복문 시작 이전에 모든 라인의 operatr를 보고 END의 operand를 확인한다

그리고 영역의 끝 (CSECT나 END)를 만난 경우

현재 영역이 그 영역인지 확인 후 E000000출력하도록 구현

6. M라인 작성 -> 지금까지 나온 모든 4형식에 operand 기준으로 작성

그런데 이녀석은 E 이전에 나와야하기 때문에

E작성 이전에 수행한다.

*/

```
int assem_pass2(const token* tokens[], int tokens_length,
               const inst* inst_table[], int inst_table_length,
               const symbol* symbol_table[], int symbol_table_length,
               const literal* literal_table[], int literal_table_length,
               object_code* obj_code) {

    int line_object_result = 0; // obj code 비트연산의 계산 결과
    obj_code->linenumber = 0; // 오브젝트 코드 라인 수 초기화
    char buffer[70]; // 작업 버퍼 MAX 70B
    memset(buffer, 'W0', 70); // 버퍼 초기화

    char* END_operand = NULL; // END 의 operand
    char* END_section = NULL; // END의 operand 가 존재한 섹션 : 본 과제는 COPY 섹션
    char* CUR_section = NULL; // 현재 섹션

    int first_text_LC = 0; // T 라인 작성시 사용되는 가장 먼저 나온 objcode 시작주소를 저장한다.
    // T 시작주소 크기 objcode
    //  ^^^^^^^^
```

```

Modification_Record modifications[MAX_MODIFICATIONS];

int modification_count = 0;

// M 라인 작성을 위한 구조체 할당이다.
// 각 섹션의 E를 적기 이전에 사용된다.

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

    // 전체 라인을 먼저 확인하여 END의 operand 를 확인한다

    // E 프로그램시작주소 <- 를 찾는 과정이다

    if (tokens[i]->operatr) { // operator 있으면

        if (strcmp("END", tokens[i]->operatr) == 0) { // END인지 확인

            END_operand = tokens[i]->operand[0]; // END의 operand

            int index = search_SYMTAB(END_operand, symbol_table, symbol_table_length);

            // 해당 operand가 SYMTAB에 있는지 확인한다.

            if (index == -1) return ERR_MISSING_END_DIRECTIVE; // 못 찾았다면 에러 처리

            END_section = symbol_table[index]->SECT_NAME;

            // operand가 속한 섹션을 END_section으로 지정한다.

            break;

        }

    }

}

if (END_section == NULL) {

    return ERR_MISSING_END_DIRECTIVE;

} // 만약 END의 operator가 SYMTAB에 없었거나 모종의 이유로 못 찾은 경우 에러 처리

```

```

int END_location_write = 0; // 만약 1이 되면 E라인 작성 시

// END의 operand를 SYMTAB에서 찾고 그 addr값을 E뒤에 붙인다

// 나머지는 서브루틴 섹션들이기 때문에 E이후에 프로그램 시작 주소를 추가하지 않는다


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

    // 라인 횟수만큼 반복하면서 각 라인에 대해 objcode를 구하는 과정을 수행한다

    if (tokens[i]->label) {

        if (strcmp(END_section, tokens[i]->label) == 0) END_location_write = 1;

    } // 만약 레이블이 존재한다면 프로그램 시작주소 적는 섹션인지 확인, 맞다면 1로 세팅


    if (tokens[i]->operatr) { // operator가 존재한다면 objcode 작성 과정을 수행한다

        if (strcmp("START", tokens[i]->operatr) == 0) {

            // START : 현재 작업 섹션(START의 Label)을 CUR_section에 저장

            if (tokens[i]->label) { // 레이블이 존재하면 현재 작업 섹션으로 설정

                CUR_section = tokens[i]->label;

            }

            if (generate_obj_code_line(buffer, "H", tokens[i], symbol_table, symbol_table_length,
obj_code) != 0) {

                return -1;

            } // H라인 작성 함수 호출

        }

        else if (strcmp("EXTDEF", tokens[i]->operatr) == 0) {

            // EXTDEF : 현재 섹션이 EXTernal DEFinition한 LABEL들을 작성

            if (generate_obj_code_line(buffer, "D", tokens[i], symbol_table, symbol_table_length,
obj_code) != 0) {

                return -1;

            }

        }

    }

}

```

```

        } // D라인 작성 함수 호출
    }

    else if (strcmp("EXTREF", tokens[i]->operatr) == 0) {

        // EXTDEF : 현재 섹션이 EXTERNAL Reference한 LABEL들을 작성

        if (generate_obj_code_line(buffer, "R", tokens[i], symbol_table, symbol_table_length,
obj_code) != 0) {

            return -1;

        } // R라인 작성 함수 호출

    }

    else if ((strcmp("CSECT", tokens[i]->operatr) == 0) || (strcmp("END", tokens[i]->operatr)
== 0)) {

        // CSECT는 의미가 좀 있다.

        // 1. 이전 작업 섹션의 종료를 의미한다. 따라서 LTORG 되지 않은 리터럴들을
메모리 할당해준다

        if (process_literals(CUR_section, literal_table, literal_table_length, buffer, tokens[i],
symbol_table, symbol_table_length, obj_code, &first_text_LC) != 0) {

            return -1;

        } // 리터럴들을 할당하는 함수 호출


        if (tokens[i]->label) { // 레이블이 존재하면 현재 작업 섹션으로 설정

            CUR_section = tokens[i]->label;

        }


        if (strlen(buffer) > 0) { // 만약 버퍼에 뭔가 objcode내용이 남아있다면

            if (generate_obj_code_line_text(buffer, "T", tokens[i], symbol_table,
symbol_table_length, obj_code, first_text_LC) != 0) {

                return -1;

            }

```

```

    } // T코드 작성 함으로써 flush 해준다

// 2. Modification record

// 현재까지 추가되었던 모든 Modification record objcode를 작성한다
for (int m = 0; m < modification_count; m++) {

    char mod_line[70];

    snprintf(mod_line, sizeof(mod_line), "M%06X%02X%c%s", // M 시작위치 변경
길이 연산 해당SYMBOL

                modifications[m].position,                modifications[m].length,
modifications[m].operation, modifications[m].label);

    if (write_obj_code(mod_line, obj_code) != 0) {

        return -1;

    } // mod_line을 objcode구조체에 저장하는 함수를 호출
}

modification_count = 0; // 모든 modification record 작성 후 카운트 초기화

// 3. E

if (END_location_write == 0) { // 일반적인 E 라인 작성

    if (generate_obj_code_line(buffer, "E", tokens[i], symbol_table,
symbol_table_length, obj_code) != 0) {

        return -1;

    } // 일반적인 E라인 작성하는 함수 호출

}

else { // END_location_write == 1 -> END 의 operand를 SYMTAB에서 찾아보고
해당 addr를 E 뒤에 붙인다

    if (generate_obj_code_line_end(buffer, "E", tokens[i], symbol_table,
symbol_table_length, obj_code, END_operand) != 0) {

        return -1;

```

```

    } // 해당 작업을 수행하는 조금 변형된 함수를 호출

    END_location_write = 0;

    // 출력 후 다시 0으로 바꾸어 일반적인 E라인 작성하도록 만듦
}

// 4. H는 CSECT로 잘랐을때 마지막에 만든다.
if (strcmp("CSECT", tokens[i]->operatr) == 0) {
    if (generate_obj_code_line(buffer, "H", tokens[i], symbol_table,
symbol_table_length, obj_code) != 0) {
        return -1;
    } // H라인 작성하는 함수 호출
}

}

else { // 모든 T 라인 생성은 이 영역에서 수행한다

    // 일단 특별한 operator(지시어들) 에 대한 수행을 한다

    if ((strcmp("RESB", tokens[i]->operatr) == 0) || (strcmp("RESW", tokens[i]->operatr)
== 0)) {

        // RESB 나 RESW는 줄바꿈 수행한다

        // 왜냐하면 공간을 차지하지만 objcode로는 작성하지 않아서

        // 결론 : 만약 버퍼에 내용있었으면 출력한다

        if (strlen(buffer) > 0) {

            if (generate_obj_code_line_text(buffer, "T", tokens[i], symbol_table,
symbol_table_length, obj_code, first_text_LC) != 0) {

                return -1;
            }
        }
    }
}
}

```

```

else if (strcmp("LTORG", tokens[i]->operatr) == 0) {

    // -> CSECT 에서 처리로 옮김

    // 그렇게 수행해도 이미 assem1에서 소속 섹션을 부여했기 때문에

    // 조건문에 따라 LTORG된 영역에서만 assemble됨

}

else if (strcmp("EQU", tokens[i]->operatr) == 0) {

    // 아무 행동 안한다 -> 메모리 안잡음

}

else if (strcmp("BYTE", tokens[i]->operatr) == 0) {

    if (strlen(buffer) == 0) { // 만약 버퍼가 비어있었으면 현재 LC를 시작 LC로

지정한다

        first_text_LC = tokens[i]->LC;

    }

    // 바이트이기에 %02X만 공간 잡음

    // BYTE 의 오퍼랜드를 해석한다

    if (tokens[i]->operand[0][0] == 'X') { // 예: X'F1'

        char* start = strchr(tokens[i]->operand[0], 'W') + 1;

        char* end = strchr(start, 'W') - 1;

        strncpy(buffer + strlen(buffer), start, end - start + 1);

        // 버퍼에 F1을 붙인다

    }

    else if (tokens[i]->operand[0][0] == 'C') { // 예: C'EOF'

        for (char* p = strchr(tokens[i]->operand[0], 'W') + 1; *p != 'W'; ++p) {

            sprintf(buffer + strlen(buffer), "%02X", *p);

        } // p 포인터 ' 한칸 뒤부터 1씩 증가시키며 buffer에 16진수로 쓰기작

업한다

    }

}

```

```

// 이 위치에서 buffer를 체크하고 필요하면 오브젝트 코드 라인을 생성
(flush)

if (strlen(buffer) > 57) {

    if (generate_obj_code_line_text(buffer, "T", tokens[i], symbol_table,
symbol_table_length, obj_code, first_text_LC) != 0) {

        return -1;

    } // 버퍼에는 obj코드만 담기고 있는데 총 60 B만 담길 수 있어서 길이
가 58이면 flush 해야 함

}

}

else if (strcmp("WORD", tokens[i]->operatr) == 0) {

    // 워드이기에 %06X 공간 잡음 = 3Byte

    int value = 0; // 기본적으로 0으로 시작

    if (isdigit(tokens[i]->operand[0][0]) || tokens[i]->operand[0][0] == '-') {

        value = atoi(tokens[i]->operand[0]); // 양수나 음수 정수는 값으로 저장
한다

    }

    else { // 식일 경우 ex) BUFEND-BUFFER

        char part_one[10] = {0}; // BUFEND

        char part_two[10] = {0}; // BUFFER

        char* minus_pos = strchr(tokens[i]->operand[0], '-');

        if (minus_pos == NULL) {

            return ERR_INVALID_EXPRESSION; // 에러 처리

        }

        size_t part_one_len = minus_pos - tokens[i]->operand[0]; // BUFEND길이

```

```

        strncpy(part_one, tokens[i]->operand[0], part_one_len);

        part_one[part_one_len] = '\0'; // BUFEND를 part_one에 저장

        strcpy(part_two, minus_pos + 1); // BUFFER를 part_two에 저장


        int     idx_one     =     search_SYMTAB(part_one,     symbol_table,
symbol_table_length);

        int     idx_two     =     search_SYMTAB(part_two,     symbol_table,
symbol_table_length);

        // 각각을 SYMBOL index에서 확인

        if (idx_one == -1 || idx_two == -1) {

            return ERR_SYMBOL_NOT_FOUND; // 에러 처리

        }

        if (strcmp(CUR_section, symbol_table[idx_one]->SECT_NAME) != 0) { // 만
약 현재 작업 섹션에 없는 심볼이라면

            add_modification_record(modifications,            &modification_count,
tokens[i]->LC, 6, part_one);

            // modification 레코드로 정보를 추가하는 함수를 호출한다

            if (strcmp(CUR_section, symbol_table[idx_two]->SECT_NAME) != 0) {

                sub_modification_record(modifications,        &modification_count,
tokens[i]->LC, 6, part_two);

                // 두번째 것은 - 연산으로 들어와야하기 때문에 조금 수정된
modification record 추가 함수를 호출한다

            }

            else {

                // Relative - Relative 아니라서 에러

                printf("Error: Relative - Relative only possible.\n");

                return ERR_INVALID_EXPRESSION;

            }

```

다

```
// 수행 후 value는 그대로 0이고, 나중에 modification으로 수정한
```

```
}
```

```
else { // 만약 동일 섹션에 있는 symbol 들이라면
```

```
value = symbol_table[idx_one]->addr - symbol_table[idx_two]->addr;
```

```
} // value를 간단하게 SYMTAB에서 가져와서 계산한다
```

```
}
```

```
if (strlen(buffer) == 0) {
```

```
first_text_LC = tokens[i]->LC;
```

```
}
```

```
sprintf(buffer + strlen(buffer), "%06X", value); // buffer에 계산한 값을 WORD
```

값을 추가

```
// 이 위치에서 buffer를 체크하고 필요하면 오브젝트 코드 라인을 생성
```

```
if (strlen(buffer) > 57) {
```

```
if (generate_obj_code_line_text(buffer, "T", tokens[i], symbol_table,  
symbol_table_length, obj_code, first_text_LC) != 0) {
```

```
return -1;
```

```
}
```

```
}
```

```
}
```

```
else { // 가장 중요한 일단 operator에 대한 작업 수행
```

```
// T 라인을 위한 buffer 채워가기
```

```
// T 시작LC 크기 opcode
```

```
int PC = tokens[i]->LC; // Target Address 계산 위한 다음 LC 저장 값
```

```
int op_index = search_opcode(tokens[i]->operatr, inst_table, inst_table_length);
```

```

if (op_index < 0) {
    continue; // 유효하지 않은 opcode는 건너 뛴 (inst table에 없음)
}

```

```

if (inst_table[op_index]->format == 2) { // 찾은 인덱스로 형식을 판단

```

```

    // 2형식 : opcode << 8 + operand

```

```

    PC += 2; // PC 값 조정 : 현재 2B짜리 2형식 코드 -> 다음 명령어는 2

```

바이트 증가

```

if (inst_table[op_index]->ops == 2) { // ex) CLEAR r1 op(8)r1(4)(4)

```

```

    line_object_result += (inst_table[op_index]->op << 8); // opcode값

```

은 8칸 왼쪽 시프트한다

```

    int reg_value = get_register_code(tokens[i]->operand[0]);

```

```

    // r1값을 알기 위해 register 번호를 가져오는 함수를 호출한다

```

```

    if (reg_value == 30) { // 임의의 오류 number이다.

```

```

        fprintf(stderr, "Unknown register '%s'\n", tokens[i]->operand[0]);

```

```

        return ERR_INVALID_REGISTER_NAME; // 오류 처리

```

```

    }

```

```

    line_object_result += (reg_value << 4); // r1은 4칸 왼쪽 시프트한다.

```

```

if (strlen(buffer) == 0) { // 만약 버퍼가 비어있었다면 현재 LC 로 첫

```

LC 세팅

```

    first_text_LC = tokens[i]->LC;

```

```

}

```

```

insert_buffer_object_code(buffer, line_object_result, 2);

```

```

// buffer에 구한 object code를 삽입하는 함수를 호출한다.

```

```

}

```

```

else if (inst_table[op_index]->ops == 3) { // SVC n

```

값으로 취급한다

다

터면

```
line_object_result += (inst_table[op_index]->op << 8); // 위와 동일
int reg_value = atoi(tokens[i]->operand[0]); // 피연산자를 그냥 정수

line_object_result += (reg_value << 4); // 4칸 왼쪽 시프트해서 더한

if (strlen(buffer) == 0) { // 버퍼 비어있으면 첫 LC 세팅

    first_text_LC = tokens[i]->LC;

}

insert_buffer_object_code(buffer, line_object_result, 2);

// 버퍼에 추가
}

else if (inst_table[op_index]->ops == 4) { // op(8)r1(4)r2(4)

    line_object_result += (inst_table[op_index]->op << 8); // 위와 동일

    int reg_value1 = get_register_code(tokens[i]->operand[0]); // r1

    int reg_value2 = get_register_code(tokens[i]->operand[1]); // r2

    if (reg_value1 == 30 || reg_value2 == 30) { // 유효하지 않은 레지스

        fprintf(stderr, "Unknown register\n");

        return ERR_INVALID_REGISTER_NAME; // 오류 처리

    }

    line_object_result += (reg_value1 << 4); // r1 은 왼쪽 4칸 시프트

    line_object_result += (reg_value2); // r2는 그대로 더하기

    if (strlen(buffer) == 0) { // 첫 LC 세팅

        first_text_LC = tokens[i]->LC;

    }

    insert_buffer_object_code(buffer, line_object_result, 2);

    // 버퍼에 추가
```

```

    }

    else {

        fprintf(stderr, "Error: Unknown format\n");

        return ERR_ILLEGAL_OPERAND_FORMAT; // 오류 처리

    }

}

else if (inst_table[op_index]->format == 3) { // 3형식, 4형식

    PC += 3; // TA 구하기위해 PC값 구함

    if (tokens[i]->operatr[0] == '+') { // 4 형식

        PC += 1; // 4B라 크기 1 증가

        line_object_result += (inst_table[op_index]->op << 24); // opcode 24

        line_object_result += (tokens[i]->nixbpe << 20); // nixbpe는 20칸 시프트

        // 4형식은 Modification record 정보 저장

        if (modification_count < MAX_MODIFICATIONS) {

            // modification record를 추가하는 함수를 호출한다

            add_modification_record(modifications,

                                    &modification_count,

                                    tokens[i]->LC + 1, 5,

                                    tokens[i]->operand[0]);

        }

        if (strlen(buffer) == 0) { // 버퍼 비었으면 첫 LC 할당

            first_text_LC = tokens[i]->LC;

        }

```

칸 시프트

프트

```

insert_buffer_object_code(buffer, line_object_result, 3);

// 버퍼에 objcode 쓰기
}

else { // 3 형식

    int displacement; // TA - PC 값 저장

    line_object_result += (inst_table[op_index]->op << 16); // opcode 16

    if (inst_table[op_index]->ops == 1) { // M 존재 - ex) ADD

        if (tokens[i]->operand[0][0] == '#') { // # immediate

            displacement = atoi(tokens[i]->operand[0]+1); // 값은 그냥

        }

        else if (tokens[i]->operand[0][0] == '@') { // @ 계산

            int sym_idx = search_SYMTAB(tokens[i]->operand[0]+1,
symbol_table, symbol_table_length);

            // @ 한칸 뒤부터 SYMTAB를 찾아본다

            displacement = symbol_table[sym_idx]->addr - PC;

            // 찾은 symbol의 addr가 TA, - PC를 하여 값을 구한다

        }

        else if (tokens[i]->operand[0][0] == '=') { // literal 계산

            int lit_idx = search_LITAB(tokens[i]->operand[0],
literal_table, literal_table_length);

            // 리터럴은 리터럴 테이블을 서치한다

            displacement = literal_table[lit_idx]->addr - PC;

            // 찾은 리터럴의 addr가 TA, -PC를 하여 값을 구한다

        }

        else {

```

칸 시프트

정수로 씀

```

        // 심볼테이블에서 찾아본다

int sym_idx = search_SYMTAB(tokens[i]->operand[0],
symbol_table, symbol_table_length);

if (sym_idx == -1) {

    fprintf(stderr, "Symbol not found.\n");

    return ERR_SYMBOL_NOT_FOUND; // 심볼 테이블에
서 찾지 못했을 경우 오류 처리

}

if (strcmp(symbol_table[sym_idx]->SECT_NAME,
CUR_section) != 0) {

    // 이 경우에는 다른섹션의 중복 심볼을 가져온 것임

    // 다시 계산 해야함

    sym_idx = search_SYMTAB_CUR_section(tokens[i]-
>operand[0], symbol_table, symbol_table_length, CUR_section);

    // 동일 섹션에 있는 심볼을 가져오도록 현재 작업 섹
션을 인자로 넘겨주는 함수를 호출한다

    if (sym_idx == -1) {

        fprintf(stderr, "Symbol not found.\n");

        return ERR_SYMBOL_NOT_FOUND; // 심볼 테이
블에서 찾지 못했을 경우 오류 처리

    }

}

// 구한 심볼 인덱스로 심볼의 addr를 구하여 dis를 계산
한다

displacement = (symbol_table[sym_idx]->addr) - PC;

}

line_object_result += (tokens[i]->nixbpe << 12); // nixbpe는 12
칸 시프트

```

```

        line_object_result += displacement; // dis만큼 추가

        if (displacement < 0) { // !!! 만약 계산 결과가 음수 -> TA가 PC
보다 작은 값이었다

            line_object_result += 4096; // 만약 계산 했는데 음수이면
자릿수 하나 빌려온것이다

        } // 그래서 결과에 0x1000 추가해준다

        if (strlen(buffer) == 0) {

            first_text_LC = tokens[i]->LC;

        } // 첫 LC 설정

        insert_buffer_object_code(buffer, line_object_result, 3);

        // 버퍼에 쓰기

    }

    else { // M 없음 - RSUB

        line_object_result += (tokens[i]->nixbpe << 12);

        // opcode와 nixbpe만 시프트 합하고 결과 도출

        if (strlen(buffer) == 0) {

            first_text_LC = tokens[i]->LC;

        }

        insert_buffer_object_code(buffer, line_object_result, 3);

    }

}

}

// buffer를 채워가다가 꼭 찾다면 오브젝트 코드에 내용을 써서 flush한다

if (strlen(buffer) > 57) {

    if (generate_obj_code_line_text(buffer, "T", tokens[i], symbol_table,

```

```

symbol_table_length, obj_code, first_text_LC) != 0) {

    return -1;

}

}

line_object_result = 0; // objcode 계산 결과 저장 변수 초기화

}

}

continue; // operator가 없다면 건너뛰다

}

}

return 0;

}

```

```

static void safe_strcat(char* dest, const char* src, size_t buf_size) {

    // 버퍼끼리 안전하게 OVERFLOW를 검사하며 붙이는함수

    // from GPT4.0 OPENAI

    size_t cur_len = strlen(dest);

    size_t src_len = strlen(src);

    if (cur_len + src_len < buf_size) {

        strcat(dest, src);

    }

    else {

        fprintf(stderr, "Error: Buffer overflow prevented.\n");

    }

}

```

```

static int generate_obj_code_line(char* buffer, const char* type, const token* tok,

    const symbol* symbol_table[], int symbol_table_length, object_code* obj_code) {

    if (strcmp(type, "H") == 0) {

        // H라인의 경우 받은 레이블의 addr를 구하고, 심볼테이블에 저장된 섹션길어도 같이
        buffer에 저장한다

        int index = search_SYMTAB(tok->label, symbol_table, symbol_table_length);

        if (index == -1) return -1;

        snprintf(buffer, 70, "H%-6s%06X%06X", tok->label, tok->LC, symbol_table[index]-
>section_length);

    }

    else if (strcmp(type, "D") == 0) {

        strcpy(buffer, "D");

        for (int j = 0; j < 3 && tok->operand[j] != NULL; j++) {

            // 최대 3개의 EXTDEF operand들과 그들의 addr를 buffer에 추가한다

            int index = search_SYMTAB(tok->operand[j], symbol_table, symbol_table_length);

            if (index == -1) return -1;

            char temp[20];

            snprintf(temp, sizeof(temp), "%s%06X", tok->operand[j], symbol_table[index]->addr);

            safe_strcat(buffer, temp, 70);

        }

    }

    else if (strcmp(type, "R") == 0) {

        strcpy(buffer, "R");

        for (int j = 0; j < 3 && tok->operand[j] != NULL; j++) {

            // 최대 3개의 EXTREF operand들을 buffer에 추가한다

            char temp[20];

            snprintf(temp, sizeof(temp), "%s", tok->operand[j]);

```

```

        safe_strcat(buffer, temp, 70);

    }

}

else if (strcmp(type, "E") == 0) {

    // 일반 E라인은 E반 버퍼에 추가

    strcpy(buffer, "E");

}

// 완성된 버퍼를 obj_code 에 적어 넣는 함수를 호출한다.

return write_obj_code(buffer, obj_code);

}

static int generate_obj_code_line_end(char* buffer, const char* type, const token* tok,

    const symbol* symbol_table[], int symbol_table_length, object_code* obj_code, char*

    END_operand) {

    // END_operand가 추가된 약간의 E라인 변형 버전이다.

    if (strcmp(type, "E") == 0) {

        strcpy(buffer, "E");

        char temp[20];

        int index = search_SYMTAB(END_operand, symbol_table, symbol_table_length);

        if (index == -1) return -1;

        snprintf(temp, sizeof(temp), "%06X", symbol_table[index]->addr);

        // 해당 operand를 심볼테이블에서 탐색해 addr를 구하고 buffer에 추가한다

        safe_strcat(buffer, temp, 70);

    }

    // buffer를 obj_code에 쓴다

    return write_obj_code(buffer, obj_code);

```

```
}
```

```
static int generate_obj_code_line_text(char* buffer, const char* type, const token* tok,  
    const symbol* symbol_table[], int symbol_table_length, object_code* obj_code, int first_text_LC)  
{  
    // T라인을 위한 flush 함수이다  
  
    if (strcmp(type, "T") == 0) {  
        char temp[70];  
  
        strcpy(temp, buffer);  
  
        int code_size = strlen(buffer) / 2;  
  
        // 사이즈를 붙이기 위해 현재 버퍼의 사이즈를 구한다.  
        // 16진수 2글자당 1바이트이기 때문에 2로 나누어준다  
  
        memset(buffer, 'W0', 70); // 버퍼 초기화  
  
        snprintf(buffer, 70, "T%06X%02X", first_text_LC, code_size);  
  
        // buffer에 T시작주소 사이즈  
  
        // 가 추가되었고 그 뒤로 원래 버퍼의 내용이 붙여진다  
  
        safe_strcat(buffer, temp, 70);  
    }  
  
    // buffer를 obj_code에 쓴다  
  
    return write_obj_code(buffer, obj_code);  
}
```

```
// buffer 내용을 obj_code에 작성하고 buffer 초기화
```

```
static int write_obj_code(char* buffer, object_code* obj_code) {  
    if (obj_code->linenumber >= MAX_OBJECT_CODE_LENGTH) {  
        fprintf(stderr, "Maximum number of object code lines exceeded.₩n");  
    }  
}
```

```

        return ERR_OBJECTCODE_LIMIT_EXCEEDED;
    }

    obj_code->obj_line[obj_code->linenumber] = malloc(strlen(buffer) + 1);

    // 새롭게 할당

    if (obj_code->obj_line[obj_code->linenumber] == NULL) {

        fprintf(stderr, "Memory allocation failed.\n");

        return ERR_ALLOCATION_FAIL;

    }

    strcpy(obj_code->obj_line[obj_code->linenumber], buffer);

    // 버퍼의 내용을 objline으로 작성

    obj_code->linenumber++; // 개수 하나 추가


    memset(buffer, 'W0', 70); // 버퍼 초기화

    return 0;
}

```

```

static int get_register_code(const char* operand) {

    // 레지스터 넘버를 리턴하는 함수

    if (strcmp(operand, "A") == 0) return 0;

    if (strcmp(operand, "X") == 0) return 1;

    if (strcmp(operand, "L") == 0) return 2;

    if (strcmp(operand, "B") == 0) return 3;

    if (strcmp(operand, "S") == 0) return 4;

    if (strcmp(operand, "T") == 0) return 5;

    if (strcmp(operand, "F") == 0) return 6;

    if (strcmp(operand, "SW") == 0) return 8;
}

```

```

    if (strcmp(operand, "PC") == 0) return 9;

    return 30; // 레지스터 이름이 일치하지 않을 경우 오류 코드 반환
}

```

```

static int insert_buffer_object_code(char* buffer, int object_code, int num_bytes) {

    // 버퍼에 objectcode를 이어붙이는 함수이다

    // from GPT4.0 OPENAI

    char temp[20];

    snprintf(temp, sizeof(temp), "%0*X", num_bytes * 2, object_code);

    safe_strcat(buffer, temp, 70);

    return 0;

}

```

```

int process_literals(const char* section, literal* literal_table[], int literal_table_length, char* buffer,
const token* tok, const symbol* symbol_table[], int symbol_table_length, object_code* obj_code,
unsigned int* first_text_LC) {

```

```

    // 리터럴은 특별하게 할당을 해주기 위한 함수이다

    // 리터럴 테이블을 돌면서 리터럴을 확인하고, 그 내용을 buffer에 써주는 함수이다

    for (int j = 0; j < literal_table_length; j++) { // 리터럴 테이블을 돌면서

        if (strcmp(section, literal_table[j]->SECT_NAME) == 0) { // 섹션이 맞는 리터럴에 대해서만
수행한다

```

```

            char* start = strchr(literal_table[j]->literal, 'W') + 1;

```

```

            char* end = strchr(start, 'W');

```

```

            if (!start || !end) continue; // 시작이나 끝을 못 찾으면 넘어간다

```

```

            int literal_size = (literal_table[j]->literal[1] == 'C') ? (end - start) * 2 : (end - start) / 2;

```

```

            // 사이즈를 계산한다

```

```

    if (strlen(buffer) + literal_size > 60) {

        if (generate_obj_code_line_text(buffer, "T", tok, symbol_table, symbol_table_length,
obj_code, *first_text_LC) != 0) {

            return -1;

        }

        // 만약 리터럴을 추가해서 bufferoverflow라면 먼저 flush한다

    }

    if (strlen(buffer) == 0) { // 만약 버퍼가 비어있었다면 첫 LC 할당

        *first_text_LC = literal_table[j]->addr;

    }


    if (literal_table[j]->literal[1] == 'C') {

        // C였다면 내용을 아스키코드로 저장

        for (; start < end; ++start) {

            sprintf(buffer + strlen(buffer), "%02X", *start);

        }

    }

    else if (literal_table[j]->literal[1] == 'X') {

        // X였다면 그 내용물을 전부 저장

        strncpy(buffer + strlen(buffer), start, end - start);

    }


    if (strlen(buffer) > 57) { // 임계값을 넘으면 버퍼를 오브젝트 코드로 변환

        if (generate_obj_code_line_text(buffer, "T", tok, symbol_table, symbol_table_length,
obj_code, *first_text_LC) != 0) {

            return -1;

        }

    }

```

```

    }

    }

}

return 0;

}

```

```

void add_modification_record(Modification_Record *modifications,int *modification_count, int
address, int length, const char* symbolName) {

```

```

    // 새로운 수정 레코드를 배열에 추가

    modifications[*modification_count].position = address; // 수정될 위치

    modifications[*modification_count].length = length; // 수정 길이

    strcpy(modifications[*modification_count].label,symbolName); // 수정할 심볼의 이름

    modifications[*modification_count].operation = '+'; // 오퍼레이션을 +로 저장

    (*modification_count)++;

}

```

```

void sub_modification_record(Modification_Record* modifications, int* modification_count, int
address, int length, const char* symbolName) {

```

```

    // 새로운 수정 레코드를 배열에 추가

    modifications[*modification_count].position = address;

    modifications[*modification_count].length = length;

    strcpy(modifications[*modification_count].label, symbolName);

    modifications[*modification_count].operation = '-'; // 오퍼레이션을 -로 저장한다

    (*modification_count)++;

}

```

```

/**

```

```

* @brief 심볼 테이블을 파일로 출력한다. `symbol_table_dir`이 NULL인 경우 결과를

```

- * stdout으로 출력한다.
- *
- * @param symbol_table_dir 심볼 테이블을 저장할 파일 경로, 혹은 NULL
- * @param symbol_table 심볼 테이블 주소
- * @param symbol_table_length 심볼 테이블 길이
- * @return 오류 코드 (정상 종료 = 0)
- *
- * @details
- * 심볼 테이블을 파일로 출력한다. `symbol_table_dir`이 NULL인 경우 결과를
- * stdout으로 출력한다. 명세서에 주어진 출력 예시와 완전히 동일할 필요는 없다.

symbol_name / Location Counter / +1? / 소속 영역

*/

```
int make_symbol_table_output(const char *symbol_table_dir,
                             const symbol *symbol_table[],
                             int symbol_table_length) {

    FILE* fp;

    int err = 0;

    if (symbol_table_dir == NULL)
        fp = stdout;

    else if ((fp = fopen(symbol_table_dir, "w")) == NULL)
        return ERR_FILE_IO_FAIL;

    // 간단하게 심볼테이블의 모든 요소를 출력한다
    // NULL이 들어오면 stdout으로 출력

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

        fprintf(fp, "%s\t%X\t", symbol_table[i]->name, symbol_table[i]->addr);
```

```

        if (symbol_table[i]->flag_add_address) {

            fprintf(fp, "+1Wt");

        }

        fprintf(fp, "%sWn", symbol_table[i]->SECT_NAME);

    }

    if (fp != stdout) {

        if (fclose(fp) != 0) return ERR_FILE_IO_FAIL;

    }

    return 0;

}

/**
 * @brief 리터럴 테이블을 파일로 출력한다. `literal_table_dir`이 NULL인 경우
 * 결과를 stdout으로 출력한다.
 *
 * @param literal_table_dir 리터럴 테이블을 저장할 파일 경로, 혹은 NULL
 * @param literal_table 리터럴 테이블 주소
 * @param literal_table_length 리터럴 테이블 길이
 * @return 오류 코드 (정상 종료 = 0)
 *
 * @details
 * 리터럴 테이블을 파일로 출력한다. `literal_table_dir`이 NULL인 경우 결과를
 * stdout으로 출력한다. 명세서에 주어진 출력 예시와 완전히 동일할 필요는 없다.

```

literal_name / Location Counter (LTORG만나면 Location Counter 할당)

*/

```
int make_literal_table_output(const char *literal_table_dir,
                             const literal *literal_table[],
                             int literal_table_length) {

    /* add your code */

    FILE* fp;

    int err = 0;

    if (literal_table_dir == NULL)

        fp = stdout;

    else if ((fp = fopen(literal_table_dir, "w")) == NULL)

        return ERR_FILE_IO_FAIL;

    // 간단하게 리터럴테이블의 모든 요소를 출력한다

    // NULL이 들어오면 stdout으로 출력

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

        fprintf(fp, "%s\t%X\n", literal_table[i]->literal, literal_table[i]->addr);

    }

    if (fp != stdout) {

        if (fclose(fp) != 0) return ERR_FILE_IO_FAIL;

    }

    return 0;

}
```

```

/**
 * @brief 오브젝트 코드를 파일로 출력한다. `objectcode_dir`이 NULL인 경우 결과를
 * stdout으로 출력한다.
 *
 * @param objectcode_dir 오브젝트 코드를 저장할 파일 경로, 혹은 NULL
 * @param obj_code 오브젝트 코드에 대한 정보를 담고 있는 구조체 주소
 * @return 오류 코드 (정상 종료 = 0)
 *
 * @details
 * 오브젝트 코드를 파일로 출력한다. `objectcode_dir`이 NULL인 경우 결과를
 * stdout으로 출력한다. 명세서의 주어진 출력 결과와 완전히 동일해야 한다.
 * 예외적으로 각 라인의 뒤쪽 공백 문자 혹은 개행 문자의 차이는 허용한다.
 */
int make_objectcode_output(const char *objectcode_dir,
                           const object_code *obj_code) {

    FILE* fp;

    int err = 0;

    if (objectcode_dir == NULL)
        fp = stdout;

    else if ((fp = fopen(objectcode_dir, "w")) == NULL)
        return ERR_FILE_IO_FAIL;

    // 간단하게 오브젝트 코드의 모든 요소를 출력한다

    // NULL이 들어오면 stdout으로 출력

```

```
for (int i = 0; i < obj_code->linenumber; ++i) {  
    fprintf(fp, "%s\n", obj_code->obj_line[i]);  
} // 라인 하나당 objcode 한줄이다  
  
if (fp != stdout) {  
    if (fclose(fp) != 0) return ERR_FILE_IO_FAIL;  
}  
  
return 0;  
}
```

my_assembler_20171969.h

/*

* parser program by

* author Enoch Jung(github.com / enochjung)

*

* assembler program by

* Park KunHo 20171969

*/

#ifndef __MY_ASSEMBLER_PARSING_H__

#define __MY_ASSEMBLER_PARSING_H__

#define MAX_INST_TABLE_LENGTH 256

#define MAX_INPUT_LINES 5000

#define MAX_TABLE_LENGTH 5000

#define MAX_OPERAND_PER_INST 3

#define MAX_OBJECT_CODE_STRING 74

#define MAX_OBJECT_CODE_LENGTH 5000

#define MAX_CONTROL_SECTION_NUM 10

// 새로운 define

#define MAX_MODIFICATIONS 100

#define MAX_COMMENT_SIZE 100

#define ERR_FILE_IO_FAIL -100

```

#define ERR_ALLOCATION_FAIL -200

#define ERR_ARRAY_OVERFLOW -1000

#define ERR_ILLEGAL_OPERATOR -10200

#define ERR_ILLEGAL_OPERAND_FORMAT -10300


// 새로운 error define

#define ERR_NAME_LENGTH_OVER -20000

#define ERR_DUPLICATE_SYMBOL -30000

#define ERR_INVALID_REGISTER_NAME -40000

#define ERR_INVALID_EXPRESSION -50000

#define ERR_INVALID_LITERAL_FORMAT -55000

#define ERR_SYMBOL_NOT_FOUND -60000

#define ERR_OBJECTCODE_LIMIT_EXCEEDED -70000

#define ERR_MISSING_END_DIRECTIVE -80000

#define ERR_MISSING_LABEL -90000

#define ERR_MISSING_OPERAND -95000


/**
 * @brief 한 개의 SIC/XE instruction을 저장하는 구조체
 *
 *
 * @details
 * 기계어 목록 파일(inst_table.txt)에 명시된 SIC/XE instruction 하나를
 * 저장하는 구조체. 라인별로 하나의 instruction을 저장하고 있는 instruction 목록
 * 파일로부터 정보를 받아와서 생성한다.
 */

typedef struct _inst {

```

```

char str[10];    /** instruction 이름 */

unsigned char op; /** instruction의 opcode */

int format;      /** instruction의 format */

int ops;         /** instruction이 가지는 operator 구분 */

} inst;

/**
 * @brief 소스코드 한 줄을 분해하여 저장하는 구조체
 *
 * @details
 * 원할한 assem을 위해 소스코드 한 줄을 label, operator, operand, comment로
 * 파싱한 후 이를 저장하는 구조체. 필드의 `operator`는 renaming을 허용한다.
 */

typedef struct _token {

    char *label;    /** label을 가리키는 포인터 */

    char *operatr; /** operator를 가리키는 포인터 */

    char *operand[MAX_OPERAND_PER_INST]; /** operand들을
                                           가리키는 포인터 배열 */

    char *comment; /** comment를 가리키는 포인터 */

    char nixbpe;    /** 특수 bit 정보 */

    int LC; /**Location Counter*/

} token;

/**

```

* @brief 하나의 심볼에 대한 정보를 저장하는 구조체

*

* @details

* SIC/XE 소스코드에서 얻은 심볼을 저장하는 구조체이다. 기존에 정의된 `name` 및

* `addr`는 필수로 사용해야 한다. 필드가 더 필요한 경우 구조체 내에 필드를

* 추가하는 것을 허용한다.

*/

```
typedef struct _symbol {
```

```
    char name[10]; /** 심볼의 이름 */
```

```
    int addr;      /** 심볼의 주소 */
```

```
    char SECT_NAME[10]; /* 소속 섹션 이름 */
```

```
    int section_length; /* 섹션의 크기 - 첫 레이블만... */
```

```
    int flag_add_address; /* 1이면 +1 출력*/
```

```
} symbol;
```

/**

* @brief 하나의 리터럴에 대한 정보를 저장하는 구조체

*

* @details

* SIC/XE 소스코드에서 얻은 리터럴을 저장하는 구조체이다. 기존에 정의된 literal

* 및 addr는 필수로 사용하고, field가 더 필요한 경우 구조체 내에 field를

* 추가하는 것을 허용한다. addr 필드는 리터럴의 값을 저장하는 것이 아닌 리터럴의

* 주소를 저장하는 필드임을 유의하라.

*/

```

typedef struct _literal {

    char literal[20]; /** 리터럴의 표현식 */

    int addr;          /** 리터럴의 주소 */


    char SECT_NAME[10]; /* 소속 섹션 이름 */


} literal;


/**
 * @brief 오브젝트 코드 전체에 대한 정보를 담는 구조체
 *
 *
 * @details
 * 오브젝트 코드 전체에 대한 정보를 담는 구조체이다. Header Record, Define
 * Record, Modification Record 등에 대한 정보를 모두 포함하고 있어야 한다. 이
 * 구조체 하나만으로 object code를 충분히 작성할 수 있도록 구조체를 직접
 * 정의해야 한다.
 */

typedef struct _object_code {

    int linenumber; // 라인의 갯수

    char* obj_line[MAX_OBJECT_CODE_LENGTH]; // 각 라인은 하나의 objcode 라인


} object_code;


typedef struct _ModificationRecord {

    // assem2 에서 쓰이는 Modification Record 저장 구조체

    char label[10]; // 변경할 심볼의 이름

```

```

    int position;    // 변경할 위치

    int length;      // 변경할 길이

    char operation;  // 수행 연산 + / -
} Modification_Record;


int init_inst_table(inst *inst_table[], int *inst_table_length,
                   const char *inst_table_dir);

int init_input(char *input[], int *input_length, const char *input_dir);

int assem_pass1(const inst *inst_table[], int inst_table_length,
               const char *input[], int input_length, token *tokens[],
               int *tokens_length, symbol *symbol_table[],
               int *symbol_table_length, literal *literal_table[],
               int *literal_table_length);

int token_parsing(const char *input, token *tok, const inst *inst_table[],
                 int inst_table_length);

int search_opcode(const char *str, const inst *inst_table[],
                 int inst_table_length);

int make_opcode_output(const char *output_dir, const token *tokens[],
                     int tokens_length, const inst *inst_table[],
                     int inst_table_length);

int assem_pass2(const token *tokens[], int tokens_length,
               const inst *inst_table[], int inst_table_length,
               const symbol *symbol_table[], int symbol_table_length,
               const literal *literal_table[], int literal_table_length,
               object_code *obj_code);

```

```
int make_symbol_table_output(const char *symbol_table_dir,

                             const symbol *symbol_table[],

                             int symbol_table_length);
```

```
int make_literal_table_output(const char *literal_table_dir,

                              const literal *literal_table[],

                              int literal_table_length);
```

```
int make_objectcode_output(const char *objectcode_dir,

                           const object_code *obj_code);
```

```
static int add_inst_to_table(inst* inst_table[], int* inst_table_length, const char* buffer);
```

```
static int token_operand_parsing(const char* operand_input, int operand_input_length, char* operand[]);
```

```
static int write_opcode_output(FILE* fp, const token* tok, const inst* inst_table[], int inst_table_length);
```

```
// Modification Record 관리 함수
```

```
void add_modification_record(Modification_Record* modifications, int* modification_count, int address, int length, const char* symbolName);
```

```
void sub_modification_record(Modification_Record* modifications, int* modification_count, int address, int length, const char* symbolName);
```

```
// Symbol / Literal Table 관리 함수
```

```
static int search_SYMTAB(const char* str, symbol* symbol_table[], int symbol_table_length);
```

```
static int search_SYMTAB_CUR_section(const char* str, symbol* symbol_table[], int symbol_table_length, const char* CUR_section);
```

```
static int insert_SYMTAB(const char* str, symbol* symbol_table[], int* symbol_table_length, int LC, const char* CUR_SECT, int SECT_flag);
```

```
static int insert_SECLen(const char* str, symbol* symbol_table[], int symbol_table_length, int
Section_length);
```

```
static int search_LITAB(const char* str, literal* literal_table[], int literal_table_length);
```

```
static int insert_LITAB(const char* str, literal* literal_table[], int* literal_table_length);
```

```
static int ORG_LITAB(const char* str, literal* literal_table[], int literal_table_length, int* LC);
```

```
// 유틸리티 함수
```

```
static void safe_strcat(char* dest, const char* src, size_t buf_size);
```

```
static int generate_obj_code_line(char* buffer, const char* type, const token* tok, const symbol*
symbol_table[], int symbol_table_length, object_code* obj_code);
```

```
static int generate_obj_code_line_end(char* buffer, const char* type, const token* tok, const symbol*
symbol_table[], int symbol_table_length, object_code* obj_code, char* END_operand);
```

```
static int generate_obj_code_line_text(char* buffer, const char* type, const token* tok, const symbol*
symbol_table[], int symbol_table_length, object_code* obj_code, int first_text_LC);
```

```
static int write_obj_code(char* buffer, object_code* obj_code);
```

```
static int get_register_code(const char* operand);
```

```
static int insert_buffer_object_code(char* buffer, int object_code, int num_bytes);
```

```
// assem2 리터럴 할당 함수
```

```
int process_literals(const char* section, literal* literal_table[], int literal_table_length, char* buffer,
const token* tok, const symbol* symbol_table[], int symbol_table_length, object_code* obj_code,
unsigned int* first_text_LC);
```

```
#endif
```