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 2025
프로젝트 명	Project #1 – ControlSection 방식의 어셈블러 구현하기
교과목 교수	최 재 영
제출인	컴퓨터학부(학과) 학번: 20232872 성명: 김도원(출석번호:102)
제출일	2025년 4월 24일

차 례

1장 동기/목적

2장 설계/구현 아이디어

3장 수행결과

4장 결론 및 보충할 점

5장 추가 함수 구현

6장 소스코드(+주석)

1장 동기/목적

SIC/XE 어셈블러를 본격적으로 개발한다. ControlSection 방식의 SIC/XE 소스를 분석해서 심볼 테이블과 리터럴 테이블을 출력하고 소스를 Object Code로 변환하는 프로그램을 구현한다. 이 프로젝트를 통해서 SIC/XE 어셈블러의 동작을 이해한다.

2장 설계/구현 아이디어

이전 과제에서 소스를 파싱해서 연산자의 opcode를 얻는 작업을 수행했었다. 해당 파싱과정에서 명령어 외의 심볼, 피연산자, 주석, nixbpe 도 추출해서 token 구조체에 저장한다. 이 프로세스가 실행되는 pass1에서는 추가로 Section을 구분하고 리터럴 테이블과 심볼 테이블의 작성도 수행한다. 그다음 pass2를 실행하면서 pass1에서 얻은 token, literal, symbol 정보들을 바탕으로 token을 순회하면서 Section 별 object code를 생성한다. 피연산자 값을 object code로 변환할 때 심볼 테이블에서 심볼을 찾지 못했으면 관련 정보를 modification record로 추가한다. object code가 만들어졌으면 output file에 Section 별로 object code를 출력한다.

3장 수행결과

작업 경로에 있는 input-1.txt를 읽어서 리터럴 테이블과 심볼 테이블을 표준출력하고 object code는 파일 출력한다. 테이블은 심볼(리터럴)과 주소로 이루어져 있다.

input-1 - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
COPY    START   0       COPY FILE FROM IN TO OUTPUT
        EXTDEF  BUFFER,BUFEND,LENGTH
        EXTREF  RDREC,WRREC
FIRST   STL     RETADR  SAVE RETURN ADDRESS
CLOOP   +JSUB   RDREC   READ INPUT RECORD
        LDA     LENGTH  TEST FOR EOF (LENGTH = 0)
        COMP    #0
        JEQ     ENDFIL  EXIT IF EOF FOUND
        +JSUB   WRREC   WRITE OUTPUT RECORD
        J       CLOOP   LOOP
ENDFIL  LDA     =C'EOF' INSERT END OF FILE MARKER
        STA     BUFFER
        LDA     #3      SET LENGTH = 3
        STA     LENGTH
        +JSUB   WRREC   WRITE EOF
        J       @RETADR RETURN TO CALLER
RETADR  RESW    1
LENGTH  RESW    1       LENGTH OF RECORD
        LTORG
BUFFER  RESB    4096    4096-BYTE BUFFER AREA
BUFEND  EQU     *
MAXLEN  EQU     BUFEND-BUFFER   MAXIMUM RECORD LENGTH
RDREC   CSECT
.
.       SUBROUTINE TO READ RECORD INTO BUFFER
.
        EXTREF  BUFFER,LENGTH,BUFEND
        CLEAR   X       CLEAR LOOP COUNTER
        CLEAR   A       CLEAR A TO ZERO
        CLEAR   S       CLEAR S TO ZERO
        LDT     MAXLEN
RLOOP   TD      INPUT   TEST INPUT DEVICE
        JEQ     RLOOP   LOOP UNTIL READY
        RD      INPUT   READ CHARACTER INTO REGISTER A
        COMPR   A,S     TEST FOR END OF RECORD (X'00')
        JEQ     EXIT    EXIT LOOP IF EOR
        +STCH   BUFFER,X        STORE CHARACTER IN BUFFER
        TIXR    T       LOOP UNLESS MAX LENGTH
        JLT     RLOOP   HAS BEEN REACHED
EXIT    +STX    LENGTH  SAVE RECORD LENGTH
        RSUB            RETURN TO CALLER
INPUT   BYTE    X'F1'   CODE FOR INPUT DEVICE
MAXLEN  WORD    BUFEND-BUFFER
WRREC   CSECT
.
.       SUBROUTINE TO WRITE RECORD FROM BUFFER
.
        EXTREF  LENGTH,BUFFER
        CLEAR   X       CLEAR LOOP COUNTER
        +LDT    LENGTH
WLOOP   TD      =X'05'  TEST OUTPUT DEVICE
        JEQ     WLOOP   LOOP UNTIL READY
        +LDCH   BUFFER,X        GET CHARACTER FROM BUFFER
        WD      =X'05'  WRITE CHARACTER
        TIXR    T       LOOP UNTIL ALL CHARACTERS
        JLT     WLOOP   HAVE BEEN WRITTEN
        RSUB            RETURN TO CALLER
        END     FIRST
```

<input-1.txt>

Microsoft Visual Studio 디버그 콘솔

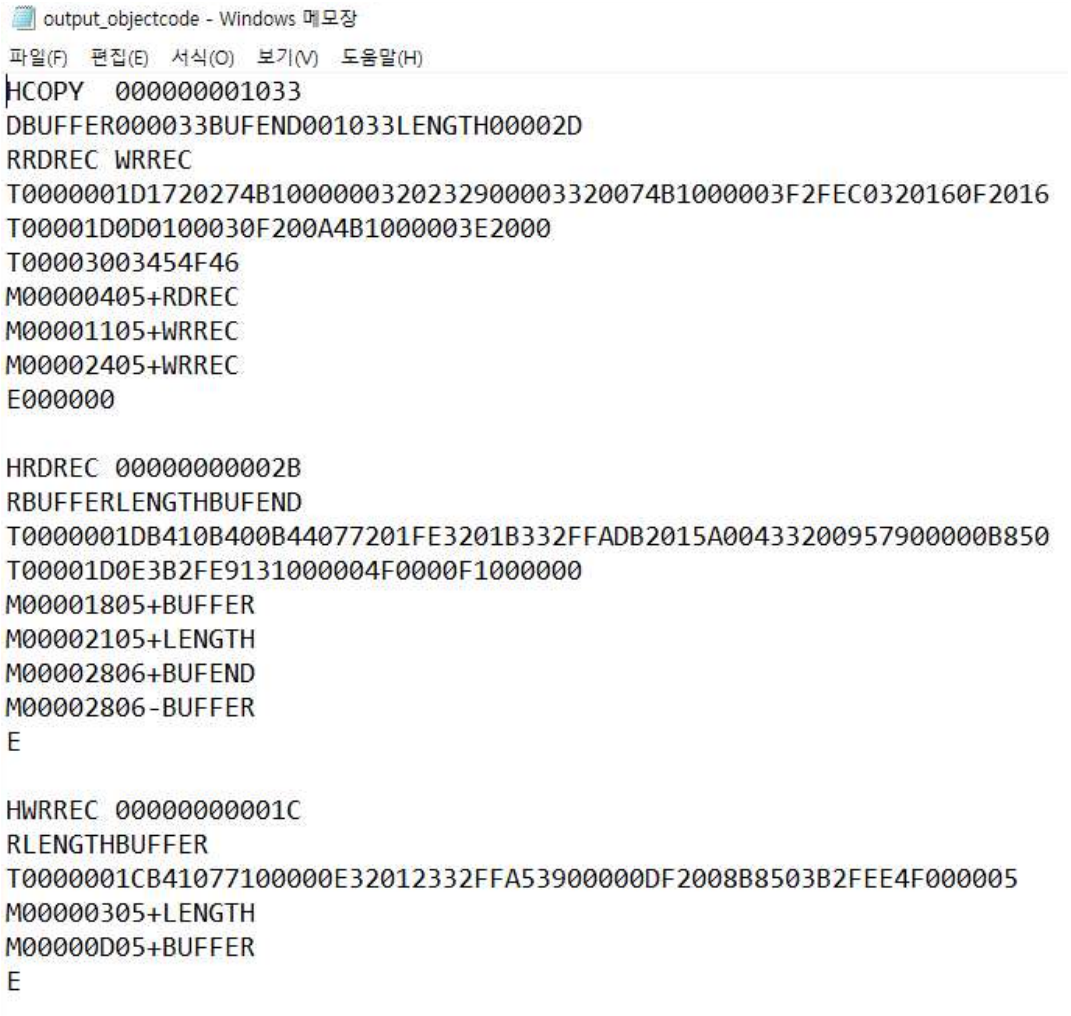
```
COPY            0
FIRST           0
CLOOP           3
ENDFIL          17
RETADR          2A
LENGTH          2D
BUFFER          33
BUFEND          1033
MAXLEN          1000

RDREC           0
RLOOP           9
EXIT            20
INPUT           27
MAXLEN          28

WRREC           0
WLOOP           6


EOF             30
05              1B
```

<stdout>


output_objectcode - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
HCOPY  000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC WRREC
T0000001D1720274B10000003202329000033200074B1000003F2FEC0320160F2016
T00001D0D0100030F200A4B1000003E2000
T00003003454F46
M00000405+RDREC
M00001105+WRREC
M00002405+WRREC
E000000

HRDREC 00000000002B
RBUFFERLENGTHBUFEND
T0000001DB410B400B44077201FE3201B332FFADB2015A0043320095790000B850
T00001D0E3B2FE9131000004F0000F1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFEND
M00002806-BUFFER
E

HWRREC 00000000001C
RLENGTHBUFFER
T0000001CB41077100000E32012332FFA53900000DF2008B8503B2FEE4F000005
M00000305+LENGTH
M00000D05+BUFFER
E
```

<output_objectcode.txt>

4장 결론 및 보충할 점

inst, token, symbol, literal, object_code 등의 구조체들을 모두 활용해서 pass1과 pass2를 거치면서 필요한 데이터들을 저장하고 최종적으로는 object code로의 변환을 마쳤다. 이 프로젝트를 완성하고 나서 자료구조 측면에서 보충할 점을 느꼈다.

4.1. token 구조체

우선 token 구조체에서 opt 변수 대신에 inst 구조체 변수를 이용하면 연산자 말고도 opcode, format, ops 정보도 token 파싱할 때 저장할 수 있게 된다. 또한, token 구조체에 int locctr과 char* section까지 추가한다면 pass2 과정을 더 쉽게 구현할 수 있다. 이 프로젝트를 구현할 때는 token 인덱스를 사용하는 locctr 배열, format 배열, 연산자 문자열 배열을 따로 추가했지만 구현에 불편했다. 이 정보들을 token 구조체 한 번에 묶으면 더 좋았을 것이다.

4.2. symbol 구조체

ControlSection 방식의 SIC/XE는 프로그램 Section별로 심볼 테이블을 관리하는 방식이 더 편할 것 같다. symbol 구조체에 해당 symbol을 가진 Section 이름인 char* section 변수를 추가했

으면 pass2의 object code 변환이 더 쉬웠을 것이다. 같은 Section 내의 심볼만 접근하고 그 외에는 modification record로 추가하는 과정을 효율적으로 처리할 수 있다. 이 프로젝트에서는 Section 이름과 심볼 테이블을 갖는 구조체를 추가 선언하고 배열로 여러 Section의 심볼 테이블을 관리하는 방식으로 구현했다.

5장 추가 함수 구현

```
//프로그램 이름으로 프로그램의 오브젝트 코드 구조체를 찾고 반환
object_code* get_objinfo(const char* block);
```

Section별 object_code 구조체가 저장된 obj_infos를 순회하면서 Section 이름에 맞는 object_code 구조체 포인터를 찾아서 반환한다. 해당 구조체를 수정해야되기 때문에 구조체 변수가 아닌 구조체 포인터 변수를 반환한다. object_code 찾지 못하면 NULL을 반환한다.

```
//START, CSECT 만났을때 프로그램 Section 레코드를 추가
void add_sector(const char* sector_name);
```

어셈블리 코드에서 새로운 Section을 감지하면 해당 함수를 호출해서 obj_infos에 해당 Section 이름의 레코드를 추가한다.

```
//특정 프로그램 Section의 심볼 테이블에 심볼 추가하기
void add_symbol(const char* sector_name, const char* label);
```

코드 분석 중에 심볼을 발견하면 현재 Section의 심볼 테이블에 해당 심볼과 주소를 추가한다.

```
//프로그램 Section 이름으로 Section의 심볼 테이블을 찾고 반환
sector_symbol* get_sectorsymbol(const char* block);
```

특정 Section의 심볼 테이블을 수정하거나 읽어야 할 때 해당 Section의 심볼 테이블을 반환하는 이 함수를 호출한다.

```
//문자열의 뒤에서부터 서브스트링을 찾기, strstr 함수 응용
char* strrstr(const char* str, const char* substr);
```

문자열의 맨 뒤에서부터 부분 문자열을 찾는 함수이다. 문자열의 앞에서부터 부분 문자열 위치를 찾는 c 라이브러리 함수인 strstr을 응용한 함수이다.

```
//char* 문자열을 int 정수로 변환, 실패하면 -1 반환
int string_to_int(const char* str);
```

문자열을 정수로 변환하는 함수이다. 코드에 숫자가 있을 때 정수형으로 변환하기 위해 주로 쓰인다.

```
//디버그용, 파싱한 토큰의 정보를 출력
```

```
void print_token();
```
token 구조체를 파싱하고 나서 올바르게 파싱됐는지를 확인하기 위한 디버그용 함수이다.

```
//delimiter를 기준으로 문자열 쪼개고 쪼개진 서브 스트링의 배열을 반환
char** split_str(const char* str, size_t* length, const char* delim);
```
구분자 문자열을 기준으로 문자열을 나눠서 문자열 배열에 담아 반환한다. 공백이나 탭으로 나뉜 명령어 줄을 분석하거나 쉼표나 +-로 나뉜 피연산자들을 분석할 때 사용된다.

```
//문자열 배열을 전체 메모리 해제
void free_strarray(char** strarr, size_t len);
```
문자열 배열의 각 원소들과 배열 자체를 다 해제하는 함수이다. 주로 split_str로 얻은 문자열 배열을 해제하는데 사용된다.

```
//C'EOF', X'05', =C'EOF' 같은 상수들을 파싱
char* parse_literal(const char* lit, size_t* len, int* equalflag);
```
피연산자 중에 C'EOF', X'EOF', 리터럴 등을 발견했을 때 이 함수를 호출해서 따옴표로 둘러싸인 실질적인 값을 추출한다.

```
//명령어의 피연산자가 리터럴인지 확인하고 리터럴이면 임시 테이블에 저장
void check_litoperand(const char* operand);
```
token 파싱할 때, 피연산자가 =로 시작하는 리터럴인지 확인하고 리터럴이면 테이블에 추가하는 작업을 한다.

```
//레이블의 주소를 찾고 전체 피연산자 값에 +-연산을 수행, 없는 레이블이면 modification record로 추가
void accumlate_label(const char* block, const char* label, int* addr, int neg, int offset, int modsize);
```
+-로 구성된 피연산자가 있을 때, 심볼의 주소를 검색해서 피연산자의 전체값을 계산하기 위한 함수이다. 해당 심볼을 못찾으면 modification record에 추가한다..

```
//레이블로 이루어진 피연산자의 전체 값을 계산
int get_labelvalue(const char* block, const char* operand, int offset, int modsize);
```
피연산자가 리터럴이나 정수가 아니고 심볼로 구성되어 있을 때 사용한다. +-가 포함된 심볼 피연산자면은 내부적으로 accumlate_label을 호출해서 피연산자 값을 구한다.

```
//명령어와 토큰을 분석해서 증가시킬 locctr 값을 계산, 몇몇 연산자의 경우에는 추가작업 실행
int get_locctr_add(inst instruction, token assemtok);
```
token 분석 후, 다음 token을 분석할 때 locctr을 얼마나 증가시킬지를 계산한다. 일반적으로 명령어의 형식만큼 증가하지만 RESB, RESW 같은 명령어의 경우에는 피연산자 값에 맞게 증

가시킨다. BYTE, WORD, EQU의 경우에는 문자열, 정수, 심볼 등의 여부를 따지면서 피연산자 값을 구하기도 한다.

```
//object code 구조체 초기화
void init_objcode();
```

object_code 구조체를 초기화한다, 주로 추가된 Section의 object_code를 반값으로 초기화하기 위해 사용된다.

```
//EOF나 05 등의 리터럴을 16진수 코드로 변환
unsigned int convert_literal(const char* literal, size_t* len);
```

문자열로 저장된 리터를 값을 길이에 맞게 16진수로 코드로 변환한다. “05”같은 리터럴은 그대로 16진수 변환을 시도해보고 안 되면 문자의 아스키코드를 16진수로 차례로 변환한다.

```
//토큰과 연산자를 분석해서 필요하면 새로운 프로그램의 object code를 추가하고 레이블이 존재한다면 추가
void add_sector_symbol(const char* op);
```

token을 파싱한 직후에 연산자와 토큰을 분석해서 START, CSECT가 있으면 새로운 Section을 추가하고 토큰에 레이블이 있다면 Section 심볼 테이블에 추가한다.

```
//임시 리터럴 테이블에 저장된 리터럴을 리터럴 테이블에 저장
void ltorg(const char* op, int checkop);
```

임시 리터럴 테이블에 저장된 리터럴들을 리터럴 테이블에 저장한다. LTORG, END를 만났을 때 호출한다.

```
//피연산자를 분석해서 immediate, indirect 등의 주소 계산 방식을 지정
void check_ni(const char* operand);
```

token을 파싱할 때, 피연산자의 첫 문자가 #, @인지 확인하고 이에 맞게 nixbpe의 ni 비트를 바꾼다.

```
//레이블 파싱하고 토큰에 저장, 성공 -> 1, 실패 -> 0
int parse_label(const char** lexes, size_t* idx, size_t lexlen);
```

명령어 줄의 레이블 파싱을 시도한다. 명령어 한 줄을 분석할 때 레이블, 연산자, 피연산자, 주석을 파싱하는 함수를 차례로 호출한다.

```
//연산자 파싱하고 토큰에 저장, 성공 -> 1, 실패 -> 0
int parse_operator(int opidx, size_t* idx, size_t lexlen);
```

레이블 파싱 후에 명령어 줄의 연산자 파싱을 시도한다. 명령어 한 줄을 분석할 때 레이블, 연산자, 피연산자, 주석을 파싱하는 함수를 차례로 호출한다.

```
//피연산자 파싱하고 토큰에 저장, 성공 -> 1, 실패 -> 0
```

```
int parse_operands(const char** lexes, size_t* idx, size_t lexlen, int opds);
```
연산자 파싱 후에 명령어 줄의 피연산자 파싱을 시도한다. 명령어 한 줄을 분석할 때 레이블, 연산자, 피연산자, 주석을 파싱하는 함수를 차례로 호출한다.

```
//주석 파싱하고 토큰에 저장, 성공 -> 1, 실패 -> 0
int parse_comment(const char** lexes, size_t* idx, size_t lexlen);
```
피연산자 파싱 후에 명령어 줄의 주석 파싱을 시도한다. 명령어 한 줄을 분석할 때 레이블, 연산자, 피연산자, 주석을 파싱하는 함수를 차례로 호출한다.

```
//리터럴 테이블을 탐색해서 해당 리터럴 구조체 반환
literal search_literal(const char* lit);
```
리터럴 테이블을 탐색해서 값과 주소가 담긴 literal 구조체를 반환한다. pass2 과정에서 피연산자를 opcode로 변환하는 과정에 주로 쓰인다.

```
//심볼 테이블을 탐색해서 해당 심볼 구조체 반환
symbol search_symbol(const char* block, const char* sym);
```
심볼 테이블을 탐색해서 이름과 주소가 담긴 symbol 구조체를 반환한다. pass2 과정에서 피연산자를 opcode로 변환하는 과정에 주로 쓰인다.
```
//연산자와 피연산자를 분석해서 EXTDEF EXTREF 레이블 저장
void check_ext(const char* op, const char* opd);
```

```
//token 구조체 초기화
void init_token();
```
token 구조체를 초기화한다, 새로운 어셈블리 한 줄을 분석하고 정보를 담기 전에 token의 변수들을 모두 초기화한다.

```
//레지스터별 번호 얻기
unsigned char get_registernum(const char* rg);
```
X, A, S, T 등의 레지스터에 맞는 레지스터 번호를 반환한다. 레지스터를 대상으로 연산하는 명령어를 opcode로 변환할 때 주로 쓰인다.

```
//연산자와 nixbpe까지만 반영된 16진수 코드에 피연산자 값도 채우기
unsigned int fill_operand(unsigned int binary, int idx);
```
pass2에서 opcode로 변환할 때, 연산자와 nixbpe를 까지 변환된 16진수 코드를 매개변수로 받아서 피연산자에 맞게 나머지 비트를 채운다.

```
//locctr를 기준으로 프로그램의 16진수 코드 정렬하는 비교 함수
int compare_locctr(const void* a, const void* b);
```
pass2 과정에서 만들어진 16진수 코드 배열을 locctr을 기준으로 정렬하기 위한 함수이다. c 라이브러리 함수인 qsort의 비교 함수에 쓰인다.

```
//modification record의 시작위치를 기준으로 프로그램의 mod record를 정렬하는 비교 함수
int compare_start(const void* a, const void* b);
```

pass1, pass2 과정에서 만들어진 modification record 배열을 start 위치를 기준으로 정렬하기 위한 함수이다. c 라이브러리 함수인 qsort의 비교 함수에 쓰인다.

6장 소스코드(+주석)

```
/*
 * 파일명 : my_assembler_00000000.c
 * 설  명 : 이 프로그램은 SIC/XE 머신을 위한 간단한 Assembler 프로그램의 메인루틴으로,
 * 입력된 파일의 코드 중, 명령어에 해당하는 OPCODE를 찾아 출력한다.
 * 파일 내에서 사용되는 문자열 "00000000"에는 자신의 학번을 기입한다.
 */

 /*
  *
  * 프로그램의 헤더를 정의한다.
  *
  */

#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <direct.h>
#include <ctype.h>
#pragma warning(disable:4996)
#define LINE_LENGTH 1024
#define MAX_BLOCKS  10
#define MAX_OPJECTS 100
//#define DEBUG

  // 파일명의 "00000000"은 자신의 학번으로 변경할 것.
#include "my_assembler_20232872.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.
 *-----------------------------------------------------------------------------
 */

//------ 리터럴 테이블에 넣기 전에 저장할 임시 리터럴 테이블 자료구조
literal tmpliteral_table[MAX_LINES];     //LTORG나 파일 끝에 도달하기 전에 저장할 임시 리터럴 배열
size_t tmplitlen_table[MAX_LINES];              //임시 리터럴의 바이트 수의 배열
int tmplit_num = 0;                                     //임시 리터럴 인덱스
int tmplit_comp = -1;                                 //LTORG나 파일 끝을 만나서 리터럴 테이블에 쓰여진 원소의 인덱스
//------

int literal_num = 0;     //리러털 테이블 인덱스

const char* block_name = NULL;       //현재 프로그램 이름

char* opstrs[MAX_LINES];              //토큰의 연산자 문자열, token 구조체와 동기화
int formats[MAX_LINES];                       //토큰의 포맷, token 구조체와 동기화
int locctrs[MAX_LINES];                 //토큰의 locctr, token 구조체와 동기화

//------ 프로그램별 심볼 테이블을 저장할 자료구조
typedef struct _sector_symbol {
        char* name;                                          //프로그램 이름
        symbol sym_table[MAX_LINES]; //프로그램의 심볼 테이블
        size_t length;                                    //심볼 테이블의 길이
} sector_symbol;

sector_symbol symbols[MAX_BLOCKS];  //프로그램들의 심볼 테이블
int symbol_num = 0;                                  //심볼 테이블의 프로그램 인덱스
//------

//------ 프로그램별 오브젝트 코드를 저장할 자료구조
object_code obj_infos[MAX_BLOCKS];   //프로그램들의 오브젝트 코드
int obj_num = -1;                                    //오브젝트 코드 배열의 프로그램 인덱스
//------
```

```
//프로그램 이름으로 프로그램의 오브젝트 코드 구조체를 찾고 반환
object_code* get_objinfo(const char* block) {
	if (block == NULL) return NULL;
	for (int i = 0; i < obj_num + 1; ++i) {
		if (strcmp(obj_infos[i].name, block) == 0) {
			return &obj_infos[i];
		}
	}
	return NULL;
}



//START, CSECT 만났을때 프로그램 Section 레코드를 추가
void add_sector(const char* sector_name) {
	if (sector_name == NULL) return;
	symbols[symbol_num].name = strdup(sector_name);
	symbols[symbol_num].length = 0;
	++symbol_num;
}



//특정 프로그램 Section의 심볼 테이블에 심볼 추가하기
void add_symbol(const char* sector_name, const char* label) {
	if (sector_name == NULL || label == NULL) return;
	for (int i = 0; i < symbol_num; ++i) {
		if (strcmp(symbols[i].name, sector_name) != 0)
			continue;
		symbol* sym_entry = &(symbols[i].sym_table[symbols[i].length]);
		strcpy(sym_entry->symbol, label);
		sym_entry->addr = locctr;
		++symbols[i].length;
		return;
	}
	add_sector(sector_name);
	symbol* sym_entry = &(symbols[symbol_num - 1].sym_table[symbols[symbol_num -
```

```
1].length]);      //추가할 symbol 구조체 포인터
        strcpy(sym_entry->symbol, label);
        sym_entry->addr = locctr;
        ++symbols[symbol_num - 1].length;
}



//프로그램 Section 이름으로 Section의 심볼 테이블을 찾고 반환
sector_symbol* get_sectorsymbol(const char* block) {
        if (block == NULL) return NULL;
        for (int i = 0; i < symbol_num; ++i) {
                if (strcmp(symbols[i].name, block) == 0) {
                        return &symbols[i];
                }
        }
        return NULL;
}

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_symtab_output("output_symtab.txt");
        //make_literaltab_output("output_littab.txt");
        make_symtab_output(NULL);              //표준출력
        make_literaltab_output(NULL);  //표준출력
        if (assem_pass2() < 0)
        {
                printf(" assem_pass2: 패스2 과정에서 실패하였습니다.  \n");
```

```
		return -1;
	}

	make_objectcode_output("output_objectcode.txt");	//파일출력
	//make_objectcode_output(NULL);

	return 0;
}

/*------------------------------------------------------------------------------
 * 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.
 *------------------------------------------------------------------------------
 */



//문자열의 뒤에서부터 서브스트링을 찾기, strstr 함수 응용
char* strrstr(const char* str, const char* substr) {
	char* result = NULL;
	char* current = (char*)str;

	if (*substr == '\0') return (char*)(str + strlen(str));

	while ((current = strstr(current, substr)) != NULL) {
		result = current;
		current++;
	}

	return result;
}
```

```
//char* 문자열을 int 정수로 변환, 실패하면 -1 반환
int string_to_int(const char* str) {
	if (str == NULL) return NULL;
	char* endptr = NULL;
	int num = (int)strtol(str, &endptr, 10);
	if (*endptr != 0) {
		return -1;
	}
	return num;
}



int init_my_assembler(void)
{
	int result;

	if ((result = init_inst_file("inst_table.txt")) < 0)
		return -1;
	if ((result = init_input_file("input-1.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;

	/* add your code here */
	file = fopen(inst_file, "r");
	if (file == NULL)
		return -1;

	inst_index = 0;
	char buffer[LINE_LENGTH] = { 0 };
	while (fgets(buffer, sizeof(buffer), file)) {	//inst_table.txt를 한 줄씩 읽어서
		inst* _inst = (inst*)malloc(sizeof(inst));
		unsigned int temp_op;
		sscanf(buffer, "%s %x %d %d", &(_inst->str), &(temp_op), &(_inst->format),
&(_inst->ops));
		//(이름\t기계어 코드\t형식\t오퍼랜드의 갯수) 형태로 저장된 명령어 정보를 inst
구조체 변수에 대입한다.
		_inst->op = (unsigned char)temp_op;
		inst_table[inst_index++] = _inst;
	}

	fclose(file);
	return 0;

	return errno;
}

/*-----------------------------------------------------------------------
 * 설명 : 어셈블리 할 소스코드를 읽어 소스코드 테이블(input_data)를 생성하는 함수이다.
 * 매계 : 어셈블리할 소스파일명
 * 반환 : 정상종료 = 0 , 에러 < 0
 * 주의 : 라인단위로 저장한다.
 *
 *-----------------------------------------------------------------------
 */
int init_input_file(char* input_file)
{
	FILE* file;
	int errno;
```

```
	/* add your code here */
	char path[1024];  // 경로를 저장할 버퍼

	file = fopen(input_file, "r");
	if (file == NULL)
		return -1;

	line_num = 0;
	char buffer[LINE_LENGTH] = { 0 };
	while (fgets(buffer, sizeof(buffer), file)) {		//input.txt를 한 줄씩 읽어서
		buffer[strlen(buffer) - 1] = 0;
		input_data[line_num++] = strdup(buffer);
		//문자열 배열인 input_data의 원소로 저장한다
	}

	fclose(file);
	return 0;

	return errno;
}



//디버그용, 파싱한 토큰의 정보를 출력
void print_token() {
	token* tok = token_table[token_line];
	if (tok == NULL) return;

	printf("locctr : %X\n", locctr);
	if(tok->label)
		printf("label : %s\n", tok->label);
	if (tok->opt)
		printf("operator : %s\n", tok->opt);
	if (tok->comment)
		printf("comment : %s\n", tok->comment);

	printf("operands : ");
	for (int i = 0; i < MAX_OPERAND; ++i) {
		if(tok->operand[i])
```

```
                printf("%s ", tok->operand[i]);
        }
        printf("\n");
        printf("nixbpe : ");
        printf("%d\n", tok->nixbpe);
        for (int i = 5; i >= 0; i--) {
                printf("%d", (tok->nixbpe >> i) & 1);
                if (i == 4)
                        printf(" ");
        }
        printf("\n");
        printf("\n\n");
}



//delimiter를 기준으로 문자열 쪼개고 쪼개진 서브 스트링의 배열을 반환
char** split_str(const char* str, size_t* length, const char* delim) {
        if (str == NULL) return NULL;
        char* tmpstr = strdup(str);     //strtok로 문자열을 쪼개면 기존 문자열이 훼손될 수
있으므로 임시 문자열로 strtok 실행
        size_t len = 0;
        char** splits = (char**)malloc(256 * sizeof(char*));

        char* split = strtok(tmpstr, delim);      //delim으로 어셈블리어 각 줄을 토큰화
        while (split != NULL && len < 256) {  //최대 1024개의 토큰받기
                splits[len++] = strdup(split);
                split = strtok(NULL, delim);
        }
        free(tmpstr);
        *length = len;
        return splits;
}



//문자열 배열을 전체 메모리 해제
void free_strarray(char** strarr, size_t len) {
        if (strarr == NULL) return;
        for (int i = 0; i < len; ++i) {
```

```
            if (strarr[i] == NULL)
                continue;
            free(strarr[i]);
        }
        free(strarr);
}



//C'EOF', X'05', =C'EOF' 같은 상수들을 파싱
char* parse_literal(const char* lit, size_t* len, int* equalflag) {
        if (lit == NULL) return NULL;
        char* current = lit;
        *equalflag = 0;
        if (*current == '=') {    //=으로 시작하는 리터럴도 처리
                *equalflag = 1;
                ++current;
        }
        char* left = strstr(lit, "'");
        char* right = strrstr(lit, "'");
        if (left == NULL) return NULL;
        size_t litlen = right - left - 1;
        char* literal = (char*)malloc(litlen + 1);
        strncpy(literal, left + 1, litlen); //따옴표 사이의 값을 추출
        literal[litlen] = 0;
        *len = (*current == 'X') ? litlen / 2 : litlen;
        return literal;
}



//명령어의 피연산자가 리터럴인지 확인하고 리터럴이면 임시 테이블에 저장
void check_litoperand(const char* operand) {
        if (operand == NULL) return;
        size_t litlen = 0;
        int equalflag = 0;
        char* literal = parse_literal(operand, &litlen, &equalflag);
        if (literal && equalflag) {              //=으로 시작하는 리터럴이면 임시 리터럴 테이
블에 추가
                tmpliteral_table[tmplit_num].literal = strdup(literal);
```

```
                tmpliteral_table[tmplit_num].addr = locctr;
                tmplitlen_table[tmplit_num] = litlen;
                ++tmplit_num;
        }
}
```

```
//함수 전방 선언
symbol search_symbol(const char* block, const char* sym);
```

```
//레이블의 주소를 찾고 전체 피연산자 값에 +-연산을 수행, 없는 레이블이면 modification record로 추가
void accumlate_label(const char* block, const char* label, int* addr, int neg, int offset, int modsize) {
        if (block == NULL || label == NULL || addr == NULL) return;
        symbol sym = search_symbol(block, label);
        if (sym.addr >= 0) {
                *addr += neg ? -(sym.addr) : sym.addr;//neg 부호에 따라 주소값에다가 더하기, 빼기 연산 수행
                return;
        }
        if (offset < 0)
                return;
        object_code* objcode = get_objinfo(block_name);
        objcode->mods[objcode->modcnt++] = (mod_record){ locctr + offset, modsize, neg, strdup(label), objcode->modcnt };
        //심볼 테이블에서 못찾으면 오프셋, 수정길이 등을 매개변수로 받아서 modification record로 추가
}
```

```
//레이블로 이루어진 피연산자의 전체 값을 계산
int get_labelvalue(const char* block, const char* operand, int offset, int modsize) {
        if (block == NULL || operand == NULL) return -1;
        if (strcmp(operand, "*") == 0) return locctr;
        char* label = strdup(operand);
```

```
	int addr = 0;

	int negflag = 0;
	char* start = label;
	for (int i = 0; i < strlen(label); ++i) {
		if (label[i] == '+' || label[i] == '-') {
			char opt = label[i];
			label[i] = 0;	//+, - 자리에 널 대입해서 문자열 분리
			accumlate_label(block, start, &addr, negflag, offset, modsize);
			negflag = (opt == '-') ? 1 : 0; //연산자가 -면 neg 에 1대입
			start = label + i + 1;			//다음 심볼 문자열 시작점
		}
	}
	if (start != NULL) {
		accumlate_label(block, start, &addr, negflag, offset, modsize);
	}
	return addr;
}



//명령어와 토큰을 분석해서 증가시킬 locctr 값을 계산, 몇몇 연산자의 경우에는 추가작업 실행
int get_locctr_add(inst instruction, token assemtok) {
	if ((assemtok.nixbpe >> 0) & 1) {
		return 4;
	}
	if (assemtok.operand[0] == NULL) {
		return instruction.format;
	}

	if (strcmp(instruction.str, "RESB") == 0) {	//RESB면 피연산자를 정수 파싱해서 정수 * 1 반환
		obj_infos[obj_num].bins[obj_infos[obj_num].bincnt++] = (binary){ 0, 0, locctr };
		int opdnum = string_to_int(assemtok.operand[0]);
		return opdnum * 1;
	}
	else if (strcmp(instruction.str, "RESW") == 0) { //RESW면 피연산자를 정수 파싱해서 정수 * 3 반환
		obj_infos[obj_num].bins[obj_infos[obj_num].bincnt++] = (binary){ 0, 0, locctr };
```

```
            int opdnum = string_to_int(assemtok.operand[0]);
            return opdnum * 3;
        }
        else if (strcmp(instruction.str, "BYTE") == 0) { //BYTE면 피연산자를 문자열 파싱해서 16진수의 길이 반환
            size_t litlen = 0;
            int equalflag = 0;
            char* literal = parse_literal(assemtok.operand[0], &litlen, &equalflag);
            unsigned int litcode = convert_literal(literal, &litlen);
            obj_infos[obj_num].bins[obj_infos[obj_num].bincnt++] = (binary){ litcode, litlen, locctr };

            return litlen;
        }
        else if (strcmp(instruction.str, "WORD") == 0) {//WORD면 피연산자를 정수 파싱 시도하고 안되면 심볼 파싱 실행
            size_t litlen = 0;
            int opdnum = string_to_int(assemtok.operand[0]);
            if (opdnum >= 0) {
                obj_infos[obj_num].bins[obj_infos[obj_num].bincnt++] = (binary){ opdnum, 3, locctr };
            }
            else {
                int value = get_labelvalue(block_name, assemtok.operand[0], 0, 6);
                obj_infos[obj_num].bins[obj_infos[obj_num].bincnt++] = (binary){ value, 3, locctr };
            }
            return 3;
        }
        else if (strcmp(instruction.str, "EQU") == 0) {  //EQU면 피연산자를 정수 파싱 시도하고 안되면 심볼 파싱 실행
            size_t litlen = 0;
            int opdnum = string_to_int(assemtok.operand[0]);
            opdnum = (opdnum >= 0) ? opdnum : get_labelvalue(block_name, assemtok.operand[0], -1, 0);
            sector_symbol* blocksym = get_sectorsymbol(block_name);
            blocksym->sym_table[blocksym->length - 1].addr = opdnum;
            return 0;
        }
        return instruction.format;
```

```
}


//object code 구조체 초기화
void init_objcode() {
	obj_infos[obj_num].name = NULL;
	obj_infos[obj_num].start = 0;
	obj_infos[obj_num].length = 0;
	obj_infos[obj_num].defs = (char**)malloc(MAX_OPERAND * sizeof(char*));
	obj_infos[obj_num].deflen = 0;
	obj_infos[obj_num].refs = (char**)malloc(MAX_OPERAND * sizeof(char*));
	obj_infos[obj_num].reflen = 0;
	obj_infos[obj_num].bincnt = 0;
	obj_infos[obj_num].modcnt = 0;
}


//EOF나 05 등의 리터럴을 16진수 코드로 변환
unsigned int convert_literal(const char* literal, size_t* len) {
	if (literal == NULL || len == NULL) return NULL;

	char* endptr = NULL;
	unsigned int bincode = (unsigned int)strtoul(literal, &endptr, 16);		//문자열 그대로
16진수 변환 시도(ex. "05" -> 0x05)
	if (*endptr == 0) {
		*len = strlen(literal) / 2;
		return bincode;
	}

	*len = 0;
	int shift = 0;
	bincode = 0;
	for (int i = strlen(literal) - 1; i >= 0; --i) {	//16진수 변환 실패하면 각 문자의 16진
수 아스키코드 추가
		if (shift >= 4) break;
		bincode |= (literal[i] << (shift * 8));
		++(*len);
		++shift;
```

```
	}
	return bincode;
}



//토큰과 연산자를 분석해서 필요하면 새로운 프로그램의 object code를 추가하고 레이블이 존재한다면 추가
void add_sector_symbol(const char* op) {
	if (op == NULL) return;
	if (strcmp(op, "START") == 0 || strcmp(op, "CSECT") == 0) {	//새로운 Section 진입
		if (block_name != NULL) {
			obj_infos[obj_num].length = locctr;
		}
		block_name = token_table[token_line]->label;

		int start = 0;
		if (strcmp(op, "START") == 0) {
			start = string_to_int(token_table[token_line]->operand[0]);		//START면 시작위치 파싱
		}
		++obj_num;
		init_objcode();
		obj_infos[obj_num].start = start;
		obj_infos[obj_num].name = strdup(block_name);
		locctr = 0;
	}
	add_symbol(block_name, token_table[token_line]->label);//레이블 존재하면 Section에 심볼 추가
}



//임시 리터럴 테이블에 저장된 리터럴을 리터럴 테이블에 저장
void ltorg(const char* op, int checkop) {
	if (checkop && op == NULL) return;
	if (checkop && strcmp(op, "LTORG") != 0) return;
	while (tmplit_comp < tmplit_num - 1) {//임시 리터럴 테이블의 마지막 원소까지 접근
		literal entry = tmpliteral_table[++tmplit_comp];
		int haslit = 0;
```

```
                for (int i = 0; i < literal_num; ++i) {
                        if (strcmp(entry.literal, literal_table[i].literal) == 0) {      //리터럴 테이블
에 추가할 리터럴에 중복되는지 검사
                                haslit = 1;
                                break;
                        }
                }
                if (haslit) continue;      //중복되지 않은 리터럴이며 리터럴 테이블에 추가
                literal_table[literal_num].literal = strdup(entry.literal);
                literal_table[literal_num].addr = locctr;
                size_t litlen = 0;
                unsigned int litcode = convert_literal(entry.literal, &litlen);       //리터럴 16진수
코드 변환하고 Section의 object code에 추가
                obj_infos[obj_num].bins[obj_infos[obj_num].bincnt++] = (binary){ litcode, litlen,
locctr };
                locctr += tmplitlen_table[tmplit_comp];
                ++literal_num;
        }
}



//피연산자를 분석해서 immediate, indirect 등의 주소 계산 방식을 지정
void check_ni(const char* operand) {
        if (operand == NULL) return NULL;
        if (operand[0] == '#') { //immediate
                token_table[token_line]->nixbpe &= ~(1 << 5);
        }
        else if (operand[0] == '@') {     //indirect
                token_table[token_line]->nixbpe &= ~(1 << 4);
        }
}



//레이블 파싱하고 토큰에 저장, 성공 -> 1, 실패 -> 0
int parse_label(const char** lexes, size_t* idx, size_t lexlen) {
        if (lexes == NULL) return 0;
        if (*idx >= lexlen) return 0;
        token_table[token_line]->label = strdup(lexes[*idx]);
```

```
        ++(*idx);
        return 1;
}



//연산자 파싱하고 토큰에 저장, 성공 -> 1, 실패 -> 0
int parse_operator(int opidx, size_t* idx, size_t lexlen) {
        if (opidx < 0) return 0;
        char opcodestr[3] = { 0 };
        snprintf(opcodestr, 3, "%02X", inst_table[opidx]->op);    //두 자리 16진수로 문자열로 저
장
        token_table[token_line]->opt = strdup(opcodestr);
        ++(*idx);
        return 1;
}



//피연산자 파싱하고 토큰에 저장, 성공 -> 1, 실패 -> 0
int parse_operands(const char** lexes, size_t* idx, size_t lexlen, int opds) {
        if (lexes == NULL) return 0;
        if (opds <= 0) return 1;
        if (*idx >= lexlen && opds > 0) return 0;
        if (strcmp(lexes[*idx], "EXTDEF") == 0 || strcmp(lexes[*idx], "EXTREF") == 0) {
                ++(*idx);
                return 1;
        }

        check_litoperand(lexes[*idx]);    //리터럴 피연산자인지 검사
        int flag = 1;
        char* tmpstr = strdup(lexes[*idx]);

        char* operand = strtok(tmpstr, ",");
        check_ni(operand);      //immdiate, indirect 검사
        for (int i = 0; i < opds; ++i) {  //명령어의 ops 만큼 피연산자 파싱
                if (operand == NULL) {
                        flag = 0;
                        break;
                }
```

```
            token_table[token_line]->operand[i] = strdup(operand);
            operand = strtok(NULL, ",");   //쉼표를 기준으로 피연산자 분리
        }

        if (operand && strcmp(operand, "X") == 0) {   //나머지 피연산자에 X가 있으면 index addressing
            token_table[token_line]->nixbpe |= (1 << 3);
            flag = 1;
        }

        free(tmpstr);
        ++(*idx);
        return flag;
}



//주석 파싱하고 토큰에 저장, 성공 -> 1, 실패 -> 0
int parse_comment(const char** lexes, size_t* idx, size_t lexlen) {
        if (lexes == NULL) return 0;
        if (*idx >= lexlen) return 1;

        char comment[100] = { 0 };
        strcpy(comment, lexes[*idx]);
        ++(*idx);
        while (*idx < lexlen) {  //명령어 끝까지 주석으로 추가
            strcat(comment, " ");
            strcat(comment, lexes[*idx]);
            ++(*idx);
        }

        strcpy(token_table[token_line]->comment, comment);
        ++(*idx);
        return 1;
}



//리터럴 테이블을 탐색해서 해당 리터럴 구조체 반환
literal search_literal(const char* lit) {
```

```
	if (lit == NULL) return (literal) { "", -1 };
	for (int i = 0; i < literal_num; ++i) {
		if (strcmp(lit, literal_table[i].literal) == 0) {
			return literal_table[i];
		}
	}
	return (literal) { "", -1 };
}



//심볼 테이블을 탐색해서 해당 심볼 구조체 반환
symbol search_symbol(const char* block, const char* sym) {
	if (block == NULL || sym == NULL) return (symbol) { "", -1 };
	sector_symbol* blocksym = get_sectorsymbol(block);
	for (int i = 0; i < blocksym->length; ++i) {
		if (strcmp(blocksym->sym_table[i].symbol, sym) == 0) {
			return blocksym->sym_table[i];
		}
	}
	return (symbol) { "", -1 };
}



//연산자와 피연산자를 분석해서 EXTDEF EXTREF 레이블 저장
void check_ext(const char* op, const char* opd) {
	if (op == NULL || opd == NULL) return;
	if (strcmp(op, "EXTDEF") && strcmp(op, "EXTREF")) return;
	size_t len = 0;
	char** opds = split_str(opd, &len, ",");
	if (strcmp(op, "EXTDEF") == 0) {	//EXTDEF 대상 심볼 추가
		for (int i = 0; i < len; ++i) {
			const char* def = opds[i];
			obj_infos[obj_num].defs[obj_infos[obj_num].deflen++] = strdup(def);
		}
	}
	else if (strcmp(op, "EXTREF") == 0) { //EXTREF 대상 심볼 추가
		for (int i = 0; i < len; ++i) {
			const char* ref = opds[i];
```

```
                        obj_infos[obj_num].refs[obj_infos[obj_num].reflen++] = strdup(ref);
                }
        }
        free_strarray(opds, len);
}



//token 구조체 초기화
void init_token() {
        token_table[token_line] = (token*)malloc(sizeof(token));
        token_table[token_line]->label = NULL;
        token_table[token_line]->opt = NULL;
        for (int i = 0; i < MAX_OPERAND; ++i) {
                token_table[token_line]->operand[i] = NULL;
        }
        strcpy(token_table[token_line]->comment, "");
        token_table[token_line]->nixbpe = 0x32; //0011 0010 으로 초기화
}

/*-----------------------------------------------------------------
 * 설명 : 소스 코드를 읽어와 토큰단위로 분석하고 토큰 테이블을 작성하는 함수이다.
 *        패스 1로 부터 호출된다.
 * 매개 : 파싱을 원하는 문자열
 * 반환 : 정상종료 = 0 , 에러 < 0
 * 주의 : my_assembler 프로그램에서는 라인단위로 토큰 및 오브젝트 관리를 하고 있다.
 *-----------------------------------------------------------------
 */
int token_parsing(char* str)
{
        /* add your code here */
        if (str == NULL) return -1;
        size_t lexlen = 0;
        char** lexemes = split_str(str, &lexlen, " \t");

#ifdef DEBUG
        for (int i = 0; i < lexlen; ++i) {
                printf("%d : %s\n", i, lexemes[i]);
        }
        printf("\n");
```

```
#endif // DEBUG

	if (lexemes[0][0] == '.') {
		free_strarray(lexemes, lexlen);
		return 0;
	}

	int labeladdr = 0;
	int flag = 1;
	size_t lexidx = 0;
	locctrs[token_line] = locctr;
	init_token();
	int opidx = search_opcode(lexemes[lexidx]);
	if (opidx < 0) { //명령어의 첫 단어가 연산자가 아니면 레이블부터 파싱
		flag = flag && parse_label(lexemes, &lexidx, lexlen);
	}
	//이후부터 차례대로 연산자, 피연산자, 주석 파싱 실행
	opidx = search_opcode(lexemes[lexidx]);
	const char* opstr = lexemes[lexidx];
	flag = flag && parse_operator(opidx, &lexidx, lexlen);
	const char* opd = lexemes[lexidx];
	flag = flag && parse_operands(lexemes, &lexidx, lexlen, inst_table[opidx]->ops);
	flag = flag && parse_comment(lexemes, &lexidx, lexlen);
	ltorg(opstr, 1);  //LTORG 명령어인지 확인
	add_sector_symbol(opstr);	//새로운 Section이나 새로운 심볼을 추가해야 하는지
확인
	check_ext(opstr, opd);  //EXTREF, EXTDEF 확인
	opstrs[token_line] = strdup(opstr);
	char nixbpe = token_table[token_line]->nixbpe;
	formats[token_line] = (nixbpe >> 0) & 1 ? 4 : inst_table[opidx]->format;	//
검사해서 format 지정

#ifdef DEBUG
	print_token();
#endif // DEBUG

	locctr += get_locctr_add(*inst_table[opidx], *token_table[token_line]);	//다음 명령어
분석을 위한 locctr 증가
	++token_line;
```

```
	free_strarray(lexemes, lexlen);
	return flag - 1;
}



/*-----------------------------------------------------------------------------
 * 설명 : 입력 문자열이 기계어 코드인지를 검사하는 함수이다.
 * 매개 : 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)
{
	/* add your code here */
	if (str == NULL) return -1;
	const char* targetop = str;
	if (str[0] == '+') {		//4형식 명령어일 경우 '+'뒷 문자열이 탐색 대상, e bit는 1로 수정, pc bit는 0으로 수정
		token_table[token_line]->nixbpe |= (1 << 0);
		token_table[token_line]->nixbpe &= ~(1 << 1);
		++targetop;
	}

	for (int i = 0; i < inst_index; ++i) {
		if (strcmp(targetop, inst_table[i]->str) != 0)		//명령어 구조체 배열을 순회하면서 토큰이랑 같은지 확인
			continue;

		if (str[0] == '+' && inst_table[i]->format != 3) {//4형식 명령어인데 형식이 3인지 확인, 아니면 에러
			return -1;
		}
		return i;
	}
	return -1;	//끝까지 명령어 못찾으면 에러
```

```
}

/*----------------------------------------------------------------------------------
* 설명 : 어셈블리 코드를 위한 패스1과정을 수행하는 함수이다.
*                 패스1에서는..
*                 1. 프로그램 소스를 스캔하여 해당하는 토큰단위로 분리하여 프로그램 라인
별 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.
*
*----------------------------------------------------------------------------------
*/
static int assem_pass1(void)
{
        /* add your code here */

        /* input_data의 문자열을 한줄씩 입력 받아서
         * token_parsing()을 호출하여 _token에 저장
         */
        token_line = 0;
        locctr = 0;
        for (int i = 0; i < line_num; ++i) {
                if (token_parsing(input_data[i]) < 0) {
                        return -1;
                }
        }
        ltorg(NULL, 0);
        if (block_name != NULL) {
                obj_infos[obj_num++].length = locctr;
        }

        make_opcode_output("output.txt");       //opcode가 같이 적혀져 있는 어셈블리어 파일
        return 0;
}
```

```
/*----------------------------------------------------------------------------
* 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.
*        프로젝트 1에서는 불필요하다.
 *
*----------------------------------------------------------------------------
*/
void make_opcode_output(char* file_name)
{
        /* add your code here */
        //FILE* fp = fopen(file_name, "w");
        //if (fp == NULL)
        //        return;

        //for (int i = 0; i < line_num; ++i) {
        //        fprintf(fp, "%s\n", input_data[i]);          //opcode가 적혀진 어셈블리어 줄 단위
로 output.txt에 입력
        //}
        //fclose(fp);
        return;
}

/*----------------------------------------------------------------------------
* 설명 : 입력된 문자열의 이름을 가진 파일에 프로그램의 결과를 저장하는 함수이다.
*        여기서 출력되는 내용은 SYMBOL별 주소값이 저장된 TABLE이다.
* 매계 : 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 */
	FILE* fp = file_name ? fopen(file_name, "w") : stdout;
	if (fp == NULL) return;

	for (int i = 0; i < symbol_num; ++i) {
		for (int j = 0; j < symbols[i].length; ++j) {
			symbol entry = symbols[i].sym_table[j];
			fprintf(fp, "%s\t\t\t%X\n", entry.symbol, entry.addr);
		}
		fprintf(fp, "\n");
	}
	fprintf(fp, "\n");
}

/*----------------------------------------------------------------------
* 설명 : 입력된 문자열의 이름을 가진 파일에 프로그램의 결과를 저장하는 함수이다.
*        여기서 출력되는 내용은 LITERAL별 주소값이 저장된 TABLE이다.
* 매계 : 생성할 오브젝트 파일명
* 반환 : 없음
* 주의 : 파일이 NULL값이 들어온다면 프로그램의 결과를 stdout으로 보내어
*        화면에 출력해준다.
*
*----------------------------------------------------------------------
*/
void make_literaltab_output(char* file_name)
{
	/* add your code here */
	FILE* fp = file_name ? fopen(file_name, "w") : stdout;
	if (fp == NULL) return;

	for (int i = 0; i < literal_num; ++i) {
		fprintf(stdout, "%s\t\t\t%X\n", literal_table[i].literal, literal_table[i].addr);
	}
	fprintf(fp, "\n");
}



//레지스터별 번호 얻기
unsigned char get_registernum(const char* rg) {
```

```
	if (rg == NULL) return -1;
	if (strcmp(rg, "A") == 0) {
		return 0;
	}
	if (strcmp(rg, "X") == 0) {
		return 1;
	}
	if (strcmp(rg, "L") == 0) {
		return 2;
	}
	if (strcmp(rg, "B") == 0) {
		return 3;
	}
	if (strcmp(rg, "S") == 0) {
		return 4;
	}
	if (strcmp(rg, "T") == 0) {
		return 5;
	}
	if (strcmp(rg, "F") == 0) {
		return 6;
	}
	if (strcmp(rg, "PC") == 0) {
		return 8;
	}
	if (strcmp(rg, "SW") == 0) {
		return 9;
	}
	return -1;
}



//연산자와 nixbpe까지만 반영된 16진수 코드에 피연산자 값도 채우기
unsigned int fill_operand(unsigned int binary, int idx) {
	if (formats[idx] == 2) { //2형식 명령어면 피연산자를 레지스터 번호로 변환
		int insert_idx = 1;
		for (int i = 0; i < MAX_OPERAND; ++i) {
			if (insert_idx < 0)
				return binary;
```

```
                if (token_table[idx]->operand[i] == NULL)
                        continue;
                unsigned char rgnum = get_registernum((token_table[idx]->operand[i]));
                binary |= (rgnum << (insert_idx * 4));
                --insert_idx;
        }
    }
    else if (formats[idx] == 3 || formats[idx] == 4) { //3, 4형식이면 심볼 탐색, 정수 변환, 리터럴 탐색을 실행
        if (token_table[idx]->operand[0] == NULL) {
                token_table[idx]->nixbpe &= ~(1 << 1);
                return binary;
        }
        char* start = token_table[idx]->operand[0];
        int num = -1;
        if (*start == '@' || *start == '#') {
                ++start;
        }
        symbol sym = search_symbol(block_name, start);        //단일 심볼 탐색 시도
        if (sym.addr >= 0) {
                num = sym.addr - (locctrs[idx] + formats[idx]);
                binary |= ((num & 0xFFF) << 0);
        }
        else if ((num = string_to_int(start)) >= 0) {      //정수 변환 시도
                token_table[idx]->nixbpe &= ~(1 << 1);
                binary |= (num << 0);
        }
        else {
                size_t litlen = 0;
                int equalflag = 0;
                char* litstr = parse_literal(start, &litlen, &equalflag);
                literal lit = search_literal(litstr); //리터럴 탐색 시도
                free(litstr);
                if (lit.addr >= 0) {
                        num = lit.addr - (locctrs[idx] + formats[idx]);
                        binary |= ((num & 0xFFF) << 0);
                }
                else {   //+, -로 이루어진 심볼값 계산 시도, 없는 심볼이면 0으로 채우고 modification record 추가
                        num = get_labelvalue(block_name, start, 1, formats[idx] * 2 -
```

```
3);
                        binary += num;
                    }
                }
            }
        return binary;
}



/*-------------------------------------------------------------------------
* 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.
* 매계 : 없음
* 반환 : 정상종료 = 0, 에러발생 = < 0
* 주의 :
*-------------------------------------------------------------------------
*/
static int assem_pass2(void)
{
        /* add your code here */
        object_code* objinfo = NULL;
        for (int i = 0; i < token_line; ++i) {
                locctr = locctrs[i];
                token* tokent = token_table[i];
                if (strcmp(opstrs[i], "START") == 0 || strcmp(opstrs[i], "CSECT") == 0) {
//현재 pass 중인 Section 이름 설정
                        block_name = tokent->label;
                        objinfo = get_objinfo(block_name);      //현재 pass 중인 Section object
code 얻기
                        continue;
                }

                if (strcmp(tokent->opt, "FF") == 0) continue;

                unsigned int code = (unsigned int)strtoul(tokent->opt, NULL, 16);      ■■
코드 변환
                code = code << (formats[i] - 1) * 8;   //format 만큼 비트 확장하기
```

```
                code = fill_operand(code, i);    //피연산자 비트 채우기
                if (formats[i] >= 3) {    //3형식 이상이면 nixbpe 채우기
                        unsigned char ni = (tokent->nixbpe >> 4) & 0xF;
                        code |= (ni << (formats[i] - 1) * 8);
                        unsigned char xbpe = (tokent->nixbpe >> 0) & 0xF;
                        code |= (xbpe << (formats[i] - 1) * 8 - 4);
                }
                objinfo->bins[objinfo->bincnt++] = (binary){ code, formats[i], locctrs[i] };//
16진수 코드 저장
        }
}



//locctr를 기준으로 프로그램의 16진수 코드 정렬하는 비교 함수
int compare_locctr(const void* a, const void* b) {
        binary* bin1 = (binary*)a;
        binary* bin2 = (binary*)b;
        if(bin1->start != bin2->start)
                return bin1->start - bin2->start;
        return bin1->len - bin2->len;
}



//modification record의 시작위치를 기준으로 프로그램의 mod record를 정렬하는 비교 함수
int compare_start(const void* a, const void* b) {
        mod_record* mod1 = (mod_record*)a;
        mod_record* mod2 = (mod_record*)b;
        if (mod1->pos == mod2->pos)
                return mod1->idx - mod2->idx;
        return mod1->pos - mod2->pos;
}

/*-----------------------------------------------------------------------------
* 설명 : 입력된 문자열의 이름을 가진 파일에 프로그램의 결과를 저장하는 함수이다.
*        여기서 출력되는 내용은 object code이다.
* 매계 : 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 */
        FILE* fp = file_name ? fopen(file_name, "w") : stdout;
        if (fp == NULL) return;

        char buffer[MAX_OPJECTS] = { 0 };
        for (int i = 0; i < obj_num; ++i) {
                qsort(obj_infos[i].bins, obj_infos[i].bincnt, sizeof(binary), compare_locctr);

                //헤더 정보 출력
                memset(buffer, 0, MAX_OPJECTS);
                snprintf(buffer, MAX_OPJECTS, "H%-6s%06X%06X\n", obj_infos[i].name,
obj_infos[i].start, obj_infos[i].length);
                fprintf(fp, buffer);

                //EXTDEF 정보 출력
                memset(buffer, 0, MAX_OPJECTS);
                for (int j = 0; j < obj_infos[i].deflen; ++j) {
                        symbol sym = search_symbol(obj_infos[i].name, obj_infos[i].defs[j]);
                        if (sym.addr >= 0) {
                                snprintf(buffer + strlen(buffer), MAX_OPJECTS, "%-6s%06X",
sym.symbol, sym.addr);
                        }
                }
                if (strcmp(buffer, "") != 0) {
                        fprintf(fp, "D%s\n", buffer);
                }

                //EXTREF 정보 출력
                memset(buffer, 0, MAX_OPJECTS);
                for (int j = 0; j < obj_infos[i].reflen; ++j) {
                        snprintf(buffer + strlen(buffer), MAX_OPJECTS, "%-6s",
obj_infos[i].refs[j]);
                }
```

```
		if (strcmp(buffer, "") != 0) {
			fprintf(fp, "R%s\n", buffer);
		}

		//TEXT 정보 출력
		memset(buffer, 0, MAX_OPJECTS);
		int total_len = 0;
		int start_loc = 0;
		for (int j = 0; j < obj_infos[i].bincnt; ++j) {
			if (total_len + obj_infos[i].bins[j].len > 30 || obj_infos[i].bins[j].len == 0)
{
				//길이가 30 바이트 초과거나 순차적인 주소로 연결되지 않으면
줄바꿈
				if (strcmp(buffer, "") == 0) continue;
				fprintf(fp, "T%06X%02X%s\n", start_loc, total_len, buffer);
				memset(buffer, 0, MAX_OPJECTS);
				total_len = 0;
				if (obj_infos[i].bins[j].len == 0) continue;
			}
			if (strcmp(buffer, "") == 0) {
				start_loc = obj_infos[i].bins[j].start;
			}
			snprintf(buffer + strlen(buffer), MAX_OPJECTS, "%0*X",
obj_infos[i].bins[j].len * 2, obj_infos[i].bins[j].code);
			//미리 저장한 코드 바이트 길이만큼 출력 형식 지정
			total_len += obj_infos[i].bins[j].len;
		}
		if (strcmp(buffer, "") != 0) {
			fprintf(fp, "T%06X%02X%s\n", start_loc, total_len, buffer);
		}

		//modification record 출력
		memset(buffer, 0, MAX_OPJECTS);
		for (int j = 0; j < obj_infos[i].modcnt; ++j) {
			qsort(obj_infos[i].mods, obj_infos[i].modcnt, sizeof(mod_record),
compare_start);
			char opt = obj_infos[i].mods[j].neg ? '-' : '+';
			snprintf(buffer, MAX_OPJECTS, "%06X%02X%c%s",
obj_infos[i].mods[j].pos, obj_infos[i].mods[j].bytes, opt, obj_infos[i].mods[j].label);
			fprintf(fp, "M%s\n", buffer);
```

```
                }

                //END 정보 출력
                memset(buffer, 0, MAX_OPJECTS);
                if (i == 0) {
                        snprintf(buffer, MAX_OPJECTS, "%06X", 0);
                }
                fprintf(fp, "E%s\n\n", buffer);
        }
}
```