EMB00000da07941

**시스템프로그래밍 2020 보고서**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **보고서 제출서약서**  **나는 숭실대학교 컴퓨터학부의 일원으로 명예를 지키면서 생활하고 있습니다.**  **나는 보고서를 작성하면서 다음과 같은 사항을 준수하였음을 엄숙히 서약합니다.**  **1. 나는 자력으로 보고서를 작성하였습니다.**  **1.1. 나는 동료의 보고서를 베끼지 않았습니다.**  **1.2. 나는 비공식적으로 얻은 해답/해설을 기초로 보고서를 작성하지 않았습니다.**  **2. 나는 보고서에서 참조한 문헌의 출처를 밝혔으며 표절하지 않았습니다. (나는 특히**  **인터넷에서 다운로드한 내용을 보고서에 거의 그대로 복사하여 사용하지 않았습니다.)**  **3. 나는 보고서를 제출하기 전에 동료에게 보여주지 않았습니다.**  **4. 나는 보고서의 내용을 조작하거나 날조하지 않았습니다.**   |  |  | | --- | --- | | **과목** | **시스템프로그래밍 2020** | | **과제명** | **Project#1 SIC/XE Machine Assembler 구현** | | **담당교수** | **최 재 영 교 수** | | **제출인** | ***전자정보공학부 20160458 김지우***  **컴퓨터 학부수업 (출석번호 109번)** | | **제출일** | **2020년 4월 29일** | |

**INDEX**

1. **프로젝트 개요** 3
   1. 개발 배경 및 목적
2. **배경 지식**  4

2.1 주제에 관한 배경지식

2.2 기술적 배경지식

1. **시스템 설계 내용**  9

3.1 전체 시스템 설계 내용

3.2 모듈별 설계 내용

1. **시스템 구현 내용(구현 화면 포함)**  23

4.1. 전체 시스템 구현 내용

4.2. 모듈별 구현 내용

1. **기대효과 및 결론**  60
2. **프로젝트 개요** 
   1. **개발 배경 및 목적**

Assembler는 어셈블리어로 작성된 프로그램을 기계어로 된 Object 프로그램으로 만들어 준다. 현재 시스템 프로그래밍을 수강하며 Assembler의 동작 및 Object program이 어떻게 만들어지는지, 원리는 무엇인지를 배우는 중이다. 개발자라면 이러한 코드들이 어떠한 의미를 가지는지, 잘못 되었다면 어떤 부분이 잘못된 것인지 알 필요가 있다. 개발자의 꿈을 가진 사람으로써, Assembler를 설계하고 구현하는 것은 큰 의미가 있는 일일 것이다. Assembler의 동작에 더 나은 이해를 갖기 위해, 이 프로젝트를 시작하게 되었다.

이 프로젝트의 목적은 Assembler의 동작에 대해 더 높은 이해를 갖는 것이다. SIC/XE 소스를 object program code로 변환해봄으로써 SIC/XE 어셈블러의 동작을 이해한다. 또, 주어진 소스코드 외 헤더파일을 이용하여 SIC/XE 소스를 object program code로 변환하는 과정을 이해하고 이후 확장되는 과제 내용에 맞추어 프로그램의 확장성을 효과적으로 증진시키기 위한 기본지식을 학습한다.

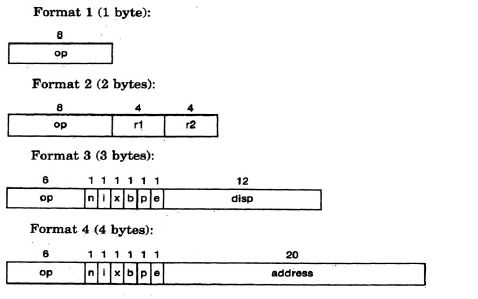
특히, 이번 프로젝트는 Control Section 내용이 추가되었다. 내용이 추가되어 동작이 복잡해진만큼 구현 또한 복잡할 것이다. 하지만 그만큼 더 고심해서 프로그램을 설계하고 구현하고 보면, 프로그램의 동작에 대한 이해를 확실히 할 수 있을 것이다.

1. **배경 지식** 
   1. **주제에 관한 배경지식**

* **SIC/XE Machine**

SIC는 Simplified Instructional Computer의 줄임말로 일반적인 하드웨어의 특성은 다 있는 간단한 컴퓨터이다. SIC Machine의 확장형이다. SIC 머신은 Addressing Mode가 Direct mode밖에 없는 반면, SIC/XE 머신은 Relative mode를 주로 쓴다.

* **Instruction Format**



[그림 2-1] Instruction Format

SIC/XE Machine은 명령어의 형식이 4가지가 존재하고, 각각의 길이가 다르다.

**1형식:** opcode 1byte만 존재한다.

**2형식:** opcode 1byte와 레지스터 두 개 각각 4bit씩 총 2byte이다.

**3형식:** opcode 6bit와 nixbpe 6bit, disp(상대주소) 12bit 총 3byte이다.

**4형식:** opcode 6bit와 nixbpe 6bit, address(보통 절대주소) 20bit 총 4byte이다. 4형식은 다른 형식들과 다르게 코드 operator앞에 +를 붙여 4형식임을 표현한다.

* **Directives**

**START:** 시작 주소와 프로그램 이름을 알려주며, 프로그램의 시작을 알리는 지시자이다.

**END:** 프로그램이 끝났음을 알려주며 처음으로 실행할 명령어의 주소를 알려준다.

**BYTE:** 16진수의 상수가 저장된다. 문자일 경우 16진수 아스키 코드가 저장이 되며, 숫자일 경우 16진수 숫자로 저장이 된다.

**WORD:** 1word(=3byte) 정수 상수가 저장이 된다.

**RESB:** 뒤에 나오는 수만큼 BYTE 데이터 공간을 예약한다.

**RESW:** 뒤에 나오는 수만큼 WORD 데이터 공간을 예약한다.

**EQU:** 해당 Symbol과 뒤에 나오는 값을 동일시한다는 지시자이다. 메모리를 잡지는 않고, 해당 Symbol은 가독성을 위해 사용된다.

**ORG:** Symbol에 간접적으로 값을 할당하는 지시자이다.

**LTORG:** Literal Organization의 줄임말로 LTORG를 만나면 지금까지 LITTAB에 입력된 Literal Pool을 생성한다. (LTORG가 없다면 Program 맨 끝에서 Literal Pool을 생성한다)

**CSECT:** 새로운 Control Section의 시작임을 알리는 지시자이다.

**EXTDEF:** External Definition의 줄임말로, 해당 Control Section에서 다른 Section에서 쓰일 수 있는 Symbol들을 정의하는 지시자이다.

**EXTREF:** External Reference의 줄임말로, 다른 Section에서 정의된 Symbol을 해당 Control Section에서 쓰겠다고 알리는 지시자이다.

* **Table**

**OPTAB(Operation Code Table):** Mnemonic operation code와 Machine language를 연결한다. (예를 들자면, STL-14) 또한 Operation마다 format과 길이가 다른데, 그 정보를 OPTAB이 갖고 있다. Pass1에서는 소스코드의 Mnemonic operation을 OPTAB을 보며 길이 정보를 확인하고, Pass2에서는 OPTAB을 보며 format 정보를 확인하고 소스코드를 Object Program으로 변환한다. 보통은 구현에 Hash table을 쓴다.

**SYMTAB(Symbol Table):** Label과 Label에 할당된 주소를 저장하는 데에 쓰인다. Pass1에서는 LOCCTR로부터 주소를 할당하여 SYMTAB에 element를 쌓아가고, Pass2에서는 SYMTAB을 보고 해당 Label과 연결된 주소를 Object Program을 만드는 데에 사용하게 된다. 이것도 보통 구현에 Hash Table을 쓴다.

**LITTAB(Literal Table):** Literal과 Literal에 할당된 주소를 저장하는 데에 쓰인다. Pass1에서는 Literal이 나온다면 LITTAB에 저장하고, LTORG나 END를 만나면 Literal들에 주소를 할당하고, Pass2에서는 LITTAB을 보며 해당 Literal의 주소를 Object Program을 쓰는 데에 사용한다.

* **Addressing Mode와 nixbpe**

**Direct mode:** 보통 가장 많이 쓰는 형식으로 해당 주소로 가는 것을 뜻한다. n=1, i=1로 설정한다. 만약, n=0, i=0일 경우 SIC 머신 명령어임을 표현한다. operand가 MAXLEN일 경우, MAXLEN에 해당하는 주소가 target address인데 SIC/XE machine은 절대 주소보다 상대 주소가 기반이기 때문에 먼저 PC relative로 계산하고, 안 된다면 Base relative로, 그것도 안 된다면 4형식으로 확장한다. 4형식의 경우 e=1로 설정한다.

**@(Indirect Addressing mode):** n=1, i=0으로 설정한다. @뒤의 값의 주소로 가서, 그 주소에 있는 메모리 값을 주소로 하여 또 한 번 이동한다. 두 번 건너뛰게 된다. @뒤의 값이 Label이라면 상대 주소를 이용해서 값을 계산한다.

**#(Immediate Addressing mode):** n=0, i=1로 설정한다. #뒤의 값이 상수로 들어가게 된다. 이 때도 #뒤에 4096같은 값이 아니라 MAXLEN같은 Label이 들어온다면 상대 주소를 이용해서 값을 계산한다. (b 또는 p가 1, relative로 안 될 경우 마찬가지로 4형식으로 확장 후, e=1로 설정한다)

**BUFFER, X(Index mode):** Index mode는 Operand에서 뒤에 X가 붙게 된다. 이 경우 x=1로 설정한다.

* **Control Section**

Program의 일부이지만, 어셈블리 이후에도 각각의 독립성을 유지하는 섹션이다. 각각의 Control Section은 어느 메모리 주소에 올라가는지 알 수 없고, 서로 독립적이다. 그러므로 각각 분리된 location counter를 사용해야 한다. 서로 다른 섹션에 있는 Symbol을 쓰고 싶을 때가 있는데, 이를 위해 EXTDEF, EXTREF라는 지시자가 생겨났다. (자세한 설명은 지시자 부분 참조) 외부 섹션의 Symbol을 쓸 때는 그 Symbol의 주소를 모르므로 상대적인 주소(disp)를 구할 수 없다. 그러므로 4형식으로 확장하여 모른다는 의미로 0을 넣어두고, 추후 Object program을 쓸 때 Modify Record를 추가하여 해당 Symbol의 주소를 넣도록 한다.

* **Pass1, Pass2에서 하는 일**

SIC/XE 머신은 Two Pass Assembler이다. 각각의 Pass에서 하는 일은 다음과 같다.

**Pass 1:** 소스 프로그램을 스캔하며 모든 문장에 주소를 할당하고, 프로그램의 크기를 계산한다. 또 Pass2에서 사용할 label들을 정의하고, 주소를 할당한다. Literal을 정의하고, 주소를 할당한다. 지시자가 나오면 처리한다.

**Pass 2:** 이제 모든 주소가 할당이 되었으므로, 실제로 소스코드를 Object Program으로 변환한다. (이것을 Assemble한다고 한다.) BYTE나 WORD로 정의된 상수들을 또한 Object Program에 생성한다. 또, Pass1에서 처리되지 않은 지시어를 처리하고, 마지막으로 Object Program을 쓴다.

* **Location Counter(LOCCTR)**

주소 값 할당을 도와주는 변수이다. 소스코드의 START 문장에서 시작주소로 초기화된다. 그 이후 명령어들을 읽으면서 해당하는 Byte를 LOCCTR에 누적해나간다. 현재 LOCCTR을 Label에 할당하면 그것이 Label의 주소가 된다.

* **Object Program에서의 Record**

Object Program에 적힌 Record의 뜻은 다음과 같다.

**Header record:** H, 프로그램(Section) 이름, 시작 주소, Section 길이

**Text record:** T, 해당 record의 시작주소, 해당 record의 길이를 적은 이후, 뒤에 Instruction을 object code로 변환한 것을 나열하는데 이 때 나열한 object code가 30byte를 넘어가게 되면 해당 record를 마무리하고 새로 Text record를 시작한다.

**End record:** E, 처음으로 실행할 명령어의 주소

**Define record:** D, 외부로 나갈 symbol 이름, 그 symbol의 주소(현재 Section에 기준한 주소)를 계속해서 나열한다.

**Refer record:** R, 외부에서 쓸 symbol 이름을 나열한다. 주소는 외부에 있어 모르므로 적지 않는다.

**Modification record:** M, 현재 Section에서 수정할 부분의 주소, 수정할 길이, Modification flag(+,-), 그 주소에 더하거나 혹은 뺄 Symbol 이름(=주소)

* 1. **기술적 배경지식**
* **파일 입출력**

FILE\* fopen(const char\* filename, const char\* mode);

file open 함수이다. mode는 “r”(읽기 전용) “w+t”(text로 쓰기)를 주로 사용한다.

int fprintf(FILE \* const \_Stream, char const \* const \_Format, ...);

서식을 지정하여 파일에 문자열을 쓸 수 있다

char\* fgets(char\* str, int num, FILE\* stream);

데이터를 한 줄씩 받기 위해 쓰는 함수. 개행 문자에 의해 입력이 끝나게 된다. stream에서 문자열을 받아 str이라는 buffer에 저장한다. 최대 num-1 바이트만큼 읽을 수 있다.

size\_t fwrite(const void\* ptr, size\_t size, size\_t count, FILE\* stream);

stream에 데이터 블록을 쓴다. size는 하나의 원소의 크기를 말하고, count는 배열의 원소 수이다. size \* count만큼 쓰게된다.

* **Token Parsing**

char \*strtok(char \*\_String, char const \*\_Delimiter);

Token으로 자르는데 필요한 string 함수로, 자른 문자열을 반환한다. 더 이상 자를 문자열이 없으면 NULL을 반환한다. 문자열 맨 앞에 토큰이라면 빈 문자열이 처음 반환되는 것이 아니라, 바로 토큰 뒤 문자열이 반환된다. 주의할 점은, 인자로 들어가는 문자열 또한 함께 잘린다는 것이다. strtok로 String을 자른 후에 String을 따로 쓰게 되면 잘린 상태로 나오게 된다. 자르고 남은 문자열을 남기고 싶을 경우 strtok\_s를 사용하도록 한다.

* **Format에 맞추어 출력**

int sprintf(char\* str, const char\* format, ...);

str에 데이터 형식에 맞추어 데이터를 쓸 수 있다. object code에 int를 hex string으로 바꾸어 적고 싶을 때, 또 record 한 줄을 적을 때 포맷에 맞춰야 하기 때문에 주로 활용한다.

* **구조체 활용**

여러 개의 정보를 담고 있는 하나의 단위가 있다면 구조체로 만들어 관리한다. 모든 구조체별로 Table들을 만들고, 얼마나 Table이 채워졌는지를 저장하는 Table index를 만들어 만든 구조체들을 관리한다.

* **포인터 활용**
* **bit처리**

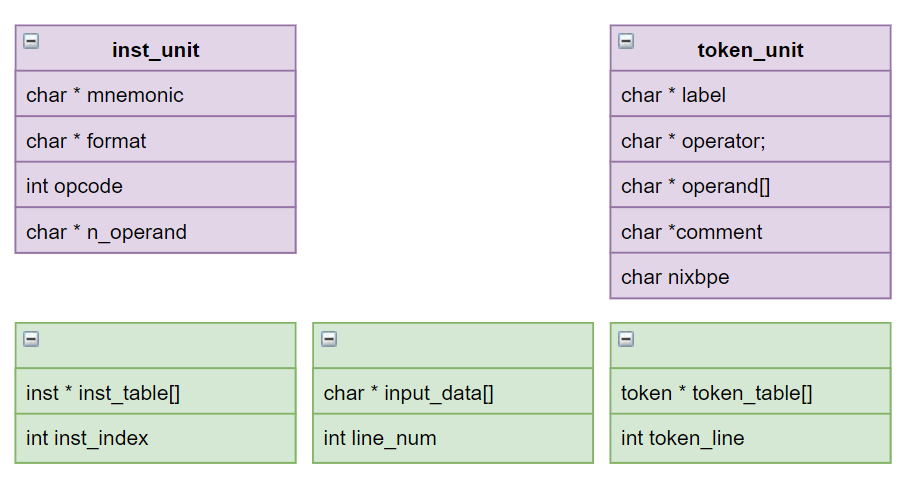
이번 프로젝트에서는 nixbpe를 bit 단위로 관리해야 했기 때문에 bit or ‘|’과 bit and ‘&’를 주로 사용하였다. (int) opcode는 맨 뒤 2bit가 00이기 때문에 opcode를 4만큼 left shift하고 nixbpe를 bit or 하게 되면 opcode에 opcode + nixbpe를 포함한 정보가 모두 들어있게 된다. 나중에 bit단위로 보고 처리할 때 &를 이용하여 원하는 비트가 1인지 아닌지를 판별하게 된다.

1. **시스템 설계 내용** 
   1. **전체 시스템 설계 내용**

이 프로젝트는 **소스 코드를 Objet Program으로 변환하는 Assembler** 프로그램을 구현하는 프로젝트이다. 위의 2.1 배경지식에서 읽을 수 있듯이, 프로그램의 전체적인 틀은 다음과 같다. SIC/XE machine의 명령어 set을 불러온 후, 이를 명령어 테이블에 저장한다. 그 후 Input text로부터 Assembly code를 불러오고, 이를 token단위로 분리하여 저장한다. 그 후 저장한 Token에 대해 Pass1과 Pass2를 실행한다.

Pass1에서는 소스 프로그램을 스캔하며 프로그램의 크기를 계산하고, SYMTAB을 만들어 Label에 주소를 할당한다. 또한 LITTAB을 만들어 Literal에 주소를 할당하고, 지시자를 처리한다. Pass2에서는 모든 주소가 할당되어 있으므로 소스코드를 Object Program으로 변환한다. BYTE와 WORD로 정의된 상수들도 HEX값으로 Object Program에 적는다.

전체 시스템을 설계하기 위해 필요한 구조체부터 설계하였다.



[그림 3-1] 설계한 구조체 및 Table - 1

1. **inst\_unit**

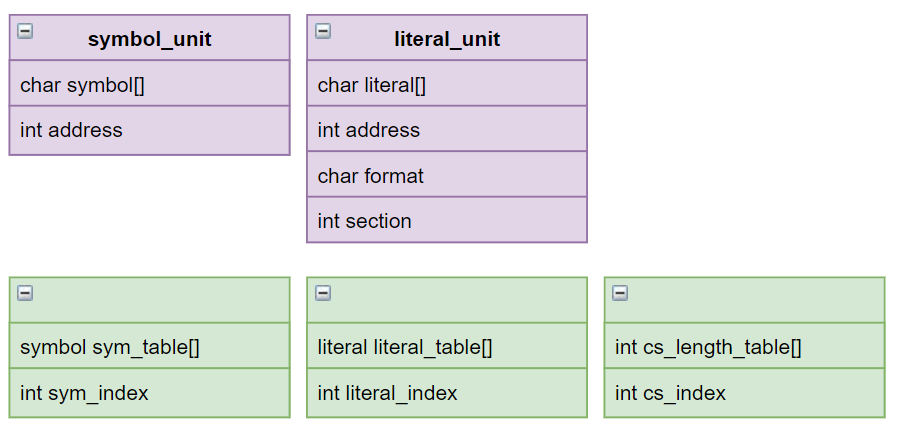
먼저 Inst data 하나하나를 저장하기 위해 inst\_unit이라는 구조체를 설계했다. 구조체에는 이름, 형식, 기계어 코드, operand의 개수를 저장하는 변수를 두었다. 해당 unit들 여러 개를 관리하기 위해 inst\_unit을 저장하는 inst\_table과 table 안에 있는 유닛들 개수를 저장하는 inst\_index 또한 설계하였다.

1. **input\_data[]**

파일을 계속 열어둘 수 없기 때문에 assembly code를 한 줄씩 모두 저장해놓을 input\_data 배열도 필요할 것이다. 몇 줄을 저장했는지 알기 위한 line\_num 변수도 함께 설계하였다.

1. **token\_unit**

처음에 코드를 한 줄씩 읽어 token으로 분리해야 하기 때문에 토큰으로 분리하여 만들어진 data를 저장하는 token\_unit 구조체를 설계하였다. 우리가 어셈블리 코드에서 볼 수 있는 label, operator, operand들, comment들 분리하여 저장할 수 있도록 했다. 해당 unit들 여러 개를 관리하기 위해 token\_unit을 저장하는 token\_table과 table 안에 있는 유닛들 개수를 저장하는 token\_line 또한 함께 설계하였다.



[그림 3-2] 설계한 구조체 및 Table - 2

1. **symbol\_unit**

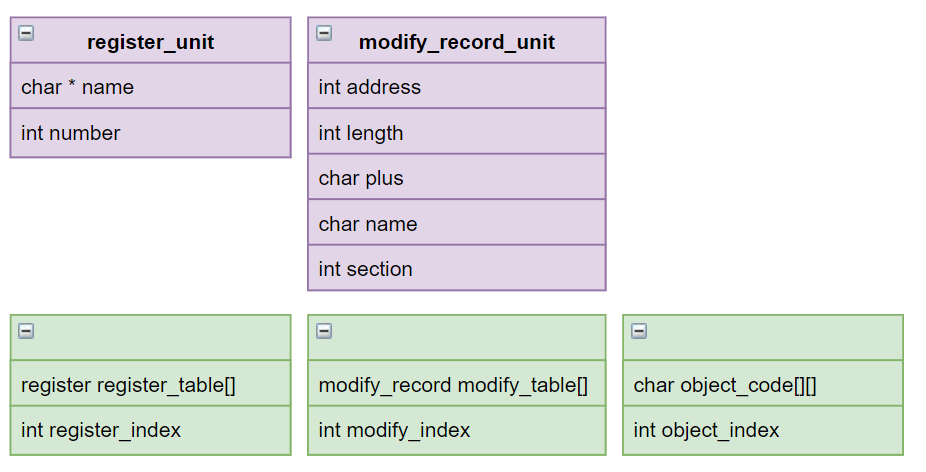
다음으로 분리된 토큰들을 읽으며 SYMTAB을 만들어야 한다. SYMTAB안에 Label과 할당된 주소를 각각 저장할 수 있는 symbol과 address 변수를 설계하였고, 이를 하나로 묶은 구조체 symbol\_unit을 설계하였다. symbol\_unit들을 저장하는, SYMTAB 역할을 하는 sym\_table[]과 몇 개의 symbol이 있는지 저장하는 sym\_index도 함께 설계했다.

1. **literal\_unit**

LITTAB 역할을 하는 literal\_table[]이 필요할 것이고, 저장된 literal 개수를 저장하는 literal\_index도 필요하다고 생각했다. LITTAB 안에 들어가는 unit으로 literal\_unit이라는 구조체를 설계했다. literal에는 literal이름과 할당 주소, 문자인지 숫자인지 구분을 위한 format과 해당 literal이 쓰이는 section을 저장하는 변수를 설계했다.

1. **cs\_length\_table[]**

이 Program은 Control Section이 나누어진 소스 코드에서도 잘 동작해야 하는 프로그램이다. 그러므로 각각 Control Section의 길이를 어딘가에 저장해 놓아야 한다고 생각했다. CS의 길이를 저장하는 cs\_length\_table[]과 현재 길이를 저장한 control section이 몇 개인지 저장하는 cs\_index를 설계하였다.



[그림 3-3] 설계한 구조체 및 Table - 3

1. **register\_unit**

SIC/XE 머신에서는 각각의 register마다(X,A,T 등) 고유한 번호가 있다. Object Program을 만들 때 Register가 쓰이는 경우, 이 번호가 반드시 필요하기 때문에 이 정보를 저장하고 있는 register\_unit이 필요하다고 생각하여 설계하였다. register 이름과 고유번호를 담고있다. Register가 여러 개이기 때문에 이를 저장하는 register\_table[]과 몇 개의 레지스터 정보가 채워졌는지 저장하는 register\_index도 함께 설계하였다.

1. **modify\_record\_unit**

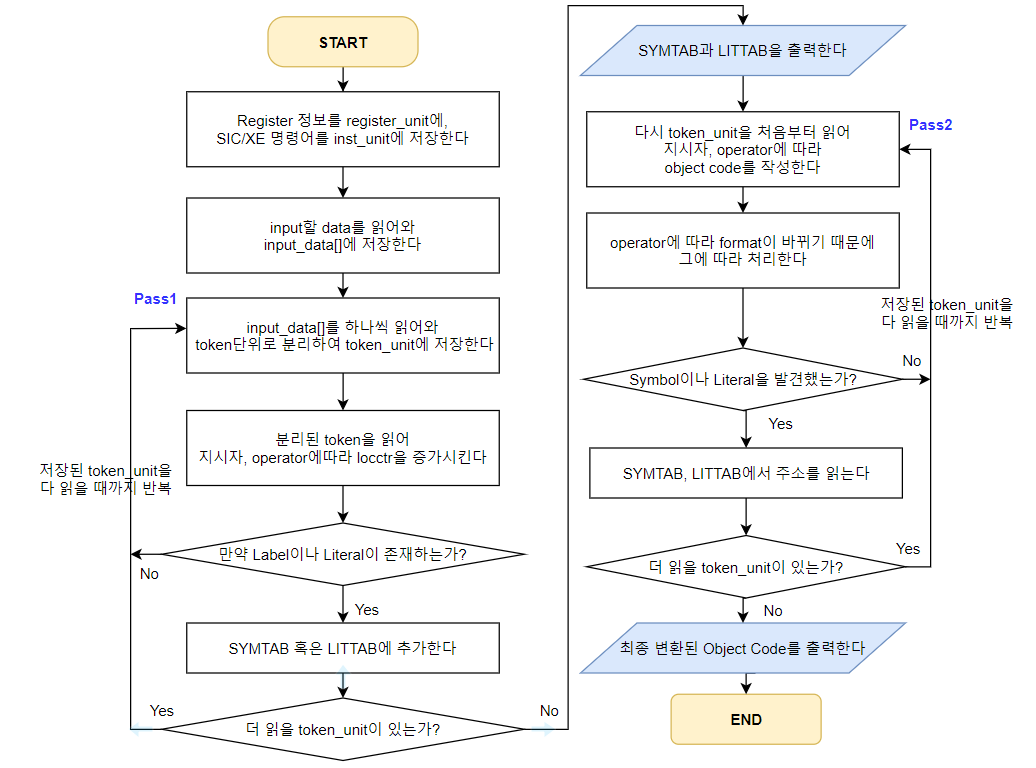
Pass2에서 수정해야 할 부분을 찾으면 바로 M레코드를 쓰는 것이 아니라, 저장해놓았다가, End 레코드를 쓰기 전에 M레코드를 쓴다. 그러므로 수정할 주소 정보를 미리 저장을 해 놓아야 하는데, 그래서 설계한 구조체가 modify\_record\_unit이다. 수정해야 할 주소, 수정할 길이, +,- flag와 더하거나 뺄 Symbol 이름을 저장한다. 또한 이 수정할 정보가 존재하는 Section도 저장한다. modify record들을 저장하는 modify\_table[]과 몇 개가 저장되어있는지 저장하는 modify\_index도 함께 설계하였다.

1. **object\_code[][]**

output file에 적으면서 그때 그때 Object 코드로 변환하는 것보다, 미리 변환하여 저장해 놓고 그것을 file에 쓰는 것이 낫다고 생각했다. 그래서 변환된 Object 코드를 저장하는 object\_code[][]를 설계했다. Object code가 몇 줄 채워졌는지 저장하는 object\_index도 함께 설계하였다.

1. **locctr**

위의 그림에는 존재하지 않지만 프로그램 길이를 계산하고 Symbol과 Literal에 주소를 할당하기 위해서는 Location Counter가 반드시 존재해야 한다.



[그림 3-4] 전체적인 시스템 흐름도

다음은 간단히 나타낸 전체적인 시스템 설계 흐름도이다.

이 시스템을 정말 큼직큼직한 모듈로 나눠본다면 먼저 (1) Register, Instruction set, Input data를 읽어와 초기화하는 모듈 (2) Pass1을 실행하는 모듈 (3) Pass2를 실행하는 모듈 (4) 출력 모듈로 크게 나눌 수 있겠다. 여기서 조금 더 세분화하여 모듈을 나누어 본다면,

**(1)은 초기화를 위해**

(1-1) register 정보를 읽어와 register\_table을 만드는 모듈

(1-2) SIC/XE 명령어 set을 읽어 inst\_table(OPTAB)을 초기화하는 모듈

(1-3) input data를 읽어 저장하는 모듈.

**(2)는 Pass1을 시행하기 위해**

(2-1) input data를 token으로 분리하여 token unit으로 저장하는 모듈

(2-2) locctr 계산을 위해 operator의 byte 수를 반환하는 모듈

(2-3) literal에 주소를 할당하는 모듈

**(3)은 Pass2를 시행하기 위해**

(3-1) opcode를 찾는 모듈

(3-2) symbol의 address를 찾는 모듈

(3-3) literal의 address를 찾는 모듈

(3-4) register number를 찾는 모듈

(3-5) modify record를 미리 만들어 저장하는 모듈

(3-6) 레코드를 추가하는 모듈

**(4)는 출력을 위해**

(4-1) symbol tab 출력 모듈

(4-2) literal table 출력 모듈

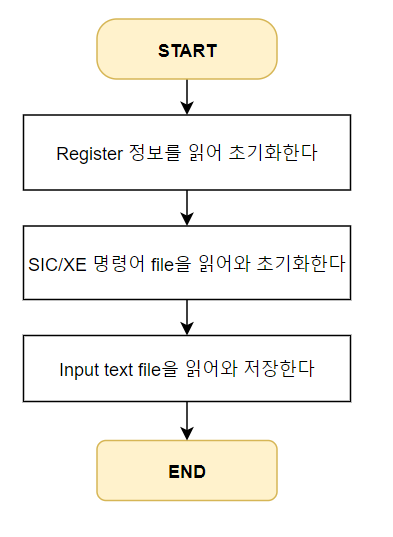
(4-3) 최종 변환된 object code 출력 모듈

다음과 같은 모듈들이 필요할 것이라 생각되었다. 세부적인 모듈에 대한 설계 흐름도는 아래에서 다룰 것이다.

* 1. **모듈별 설계 내용**

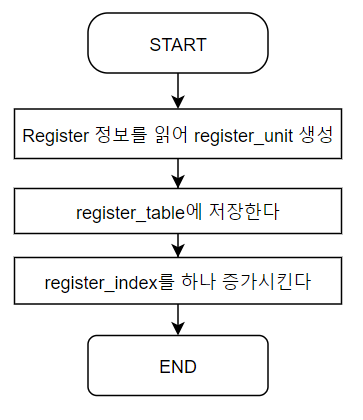
너무 자잘한 에러 처리는 보지 않고, 전체적인 흐름을 보기 위한 flow chart이다.

**(1) 초기화 모듈**



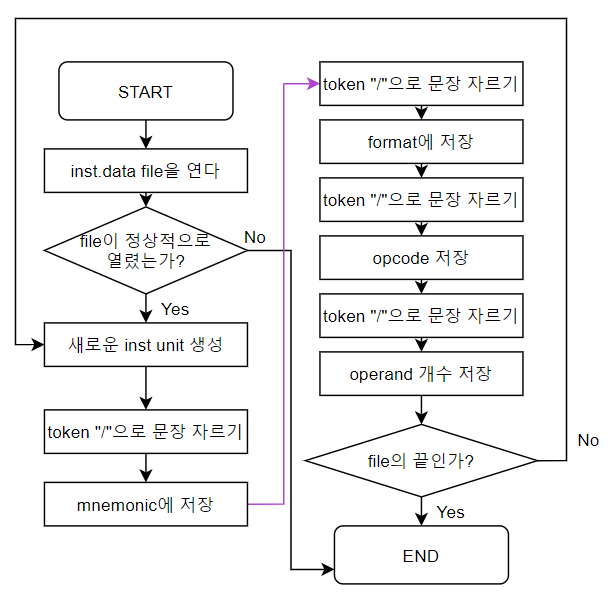
[그림 3-5] 해당 모듈의 flow chart

(1-1) register 정보를 읽어와 register\_table을 만드는 모듈



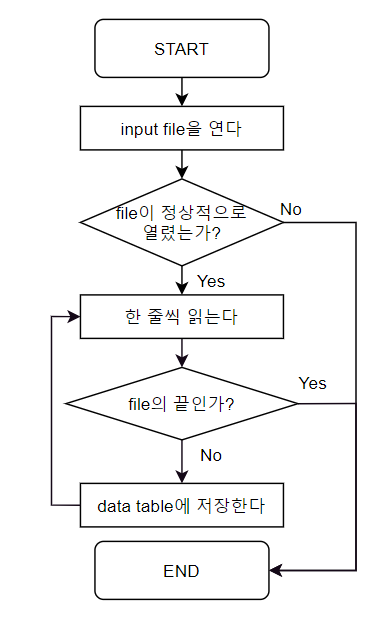
[그림 3-6] 해당 모듈의 flow chart

(1-2) SIC/XE 명령어 set을 읽어 inst\_table(OPTAB)을 초기화하는 모듈



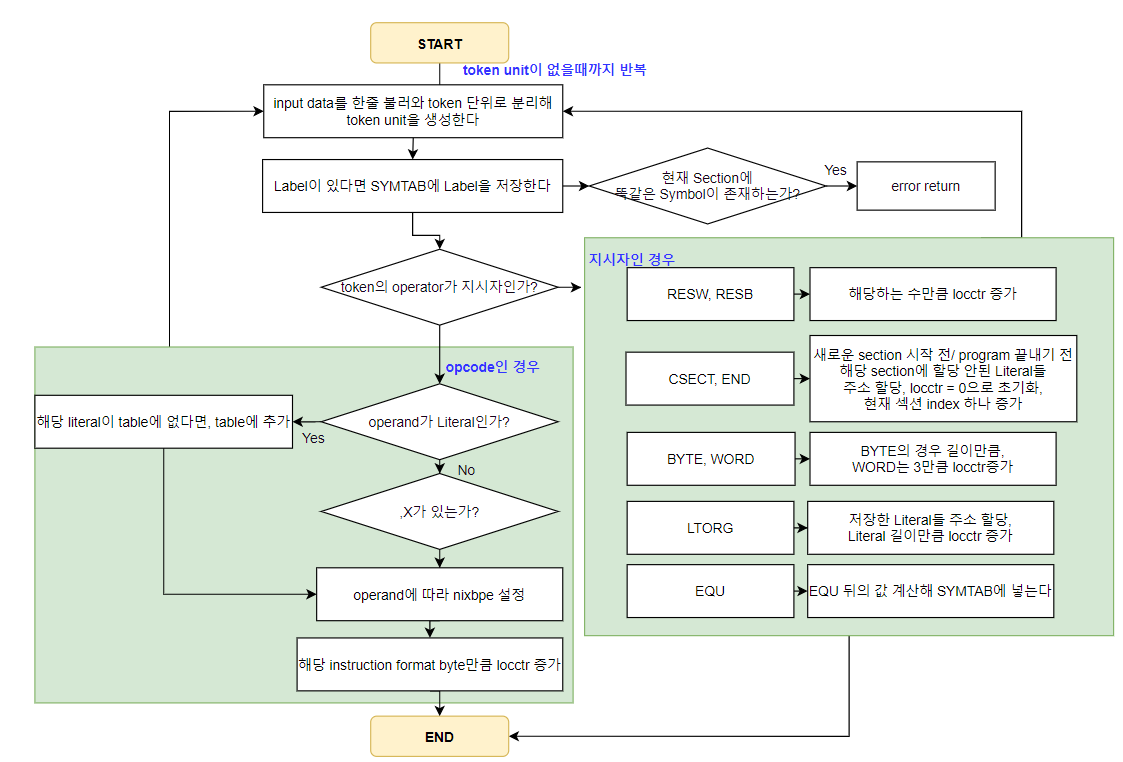
[그림 3-7] 해당 모듈의 flow chart

(1-3) input data를 읽어 저장하는 모듈.



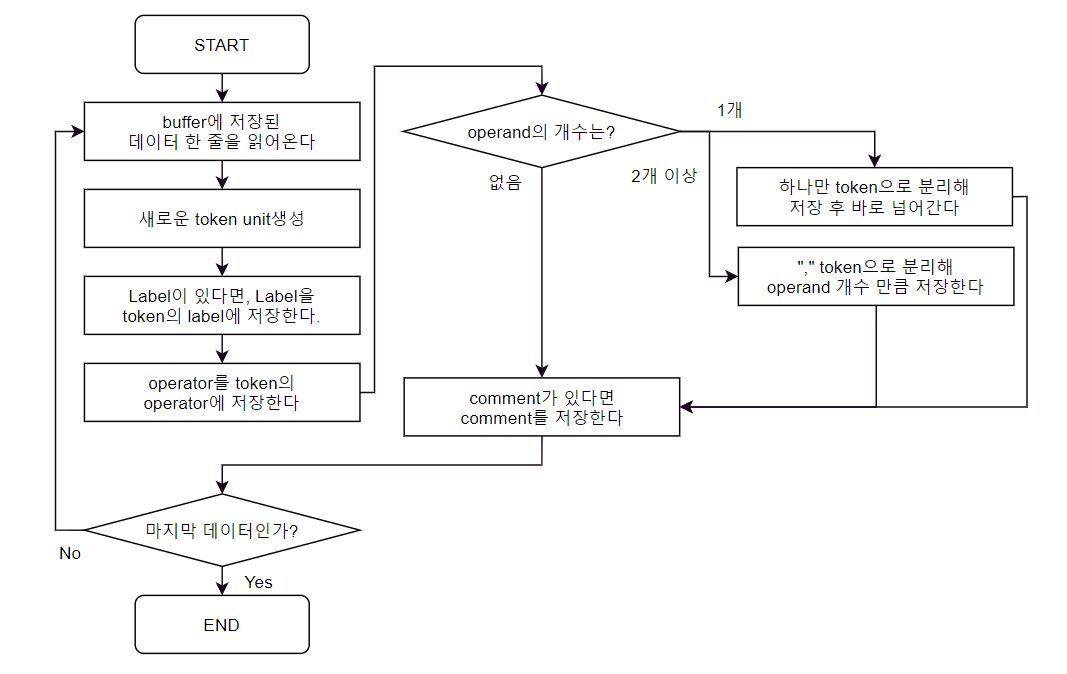
[그림 3-8] 해당 모듈의 flow chart

**(2) Pass1 모듈**



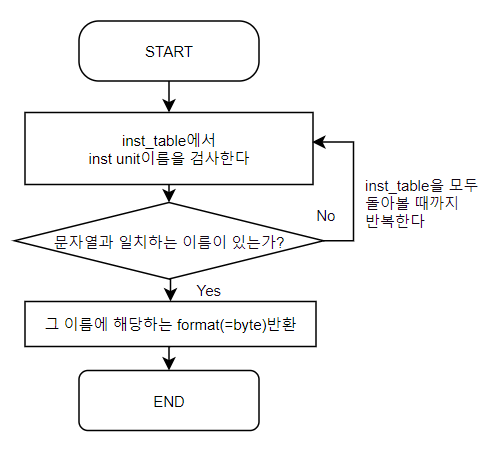
[그림 3-9] 해당 모듈의 flow chart

(2-1) input data를 token으로 분리하여 token unit으로 저장하는 모듈



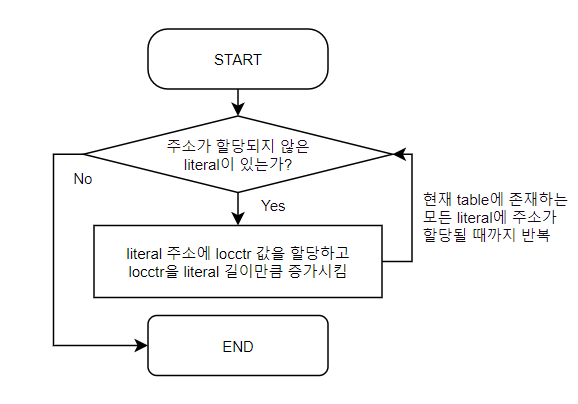
[그림 3-10] 해당 모듈의 flow chart

(2-2) operator의 byte 수를 반환하는 모듈



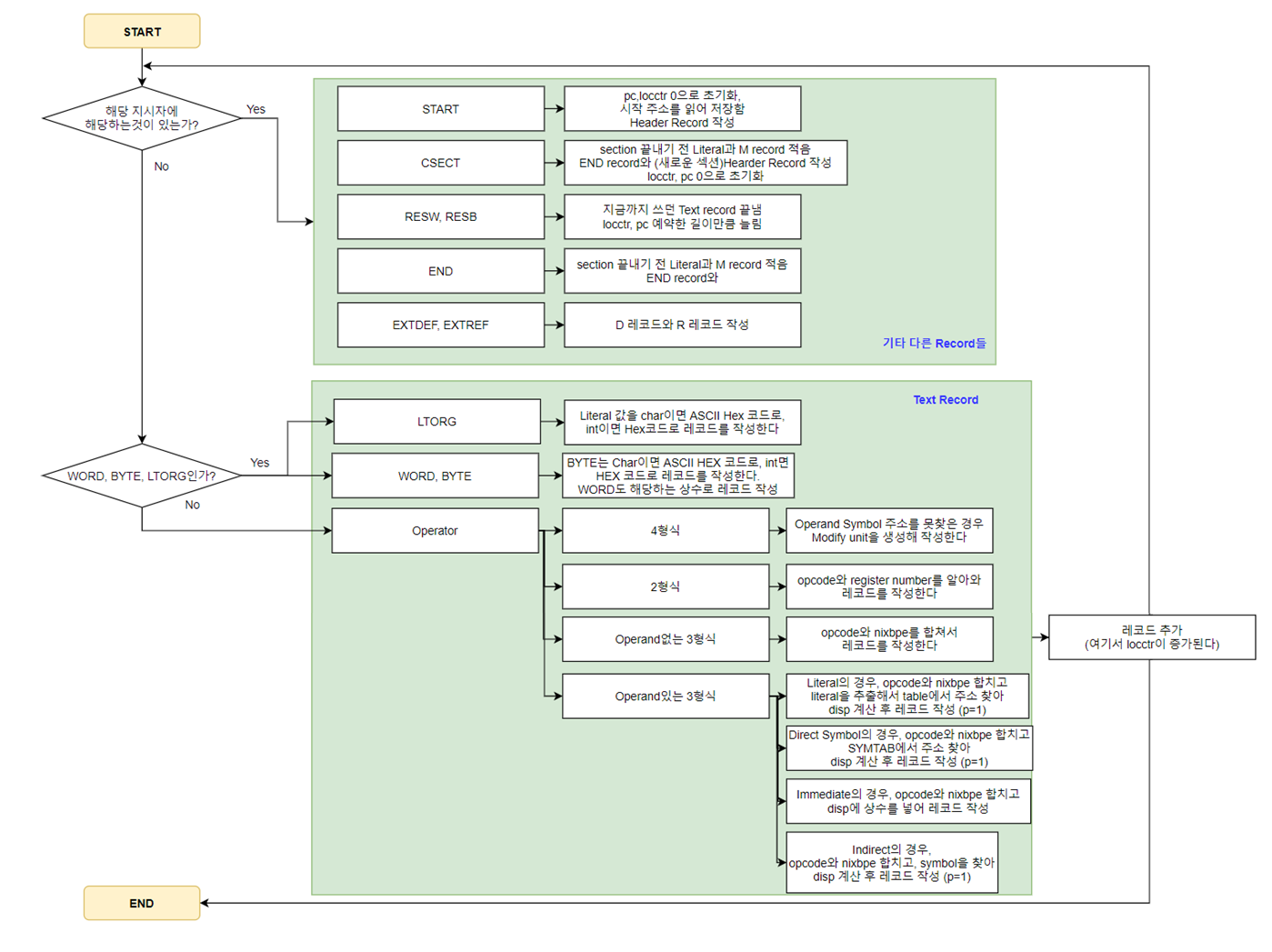
[그림 3-11] 해당 모듈의 flow chart

(2-3) literal에 주소를 할당하는 모듈



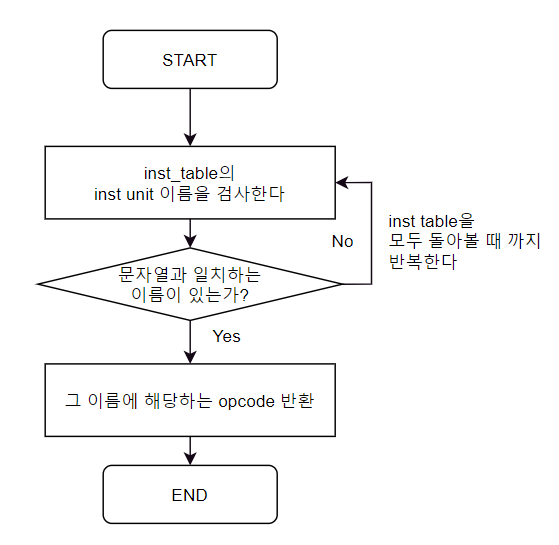
[그림 3-12] 해당 모듈의 flow chart

**(3) Pass2 모듈**



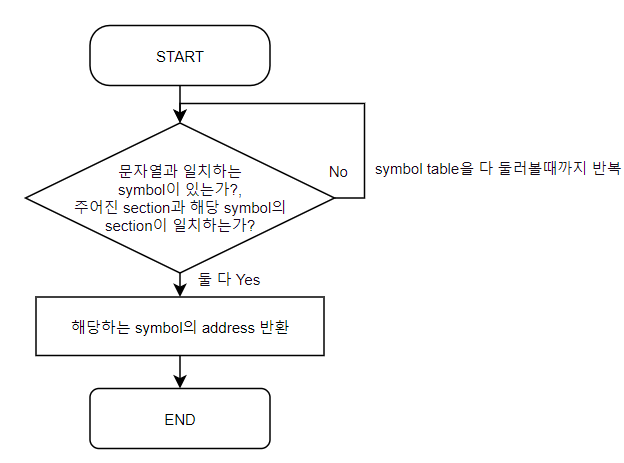
[그림 3-13] 해당 모듈의 flow chart

(3-1) opcode를 찾는 모듈



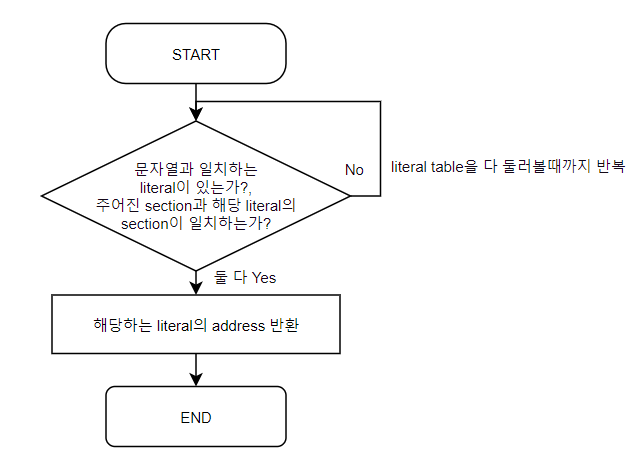
[그림 3-14] 해당 모듈의 flow chart

(3-2) symbol의 address를 찾는 모듈



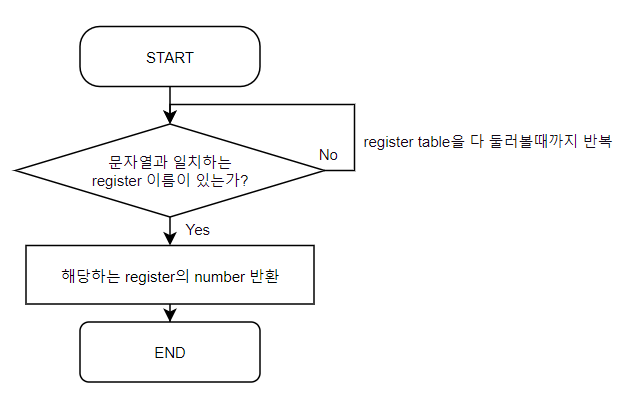
[그림 3-15] 해당 모듈의 flow chart

(3-3) literal의 address를 찾는 모듈



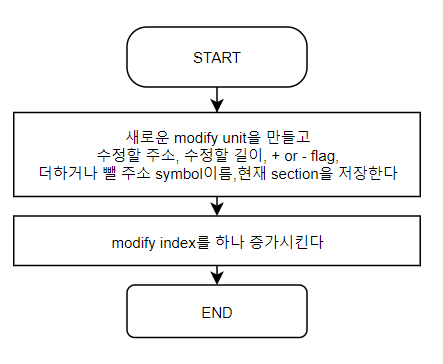
[그림 3-16] 해당 모듈의 flow chart

(3-4) register number를 찾는 모듈



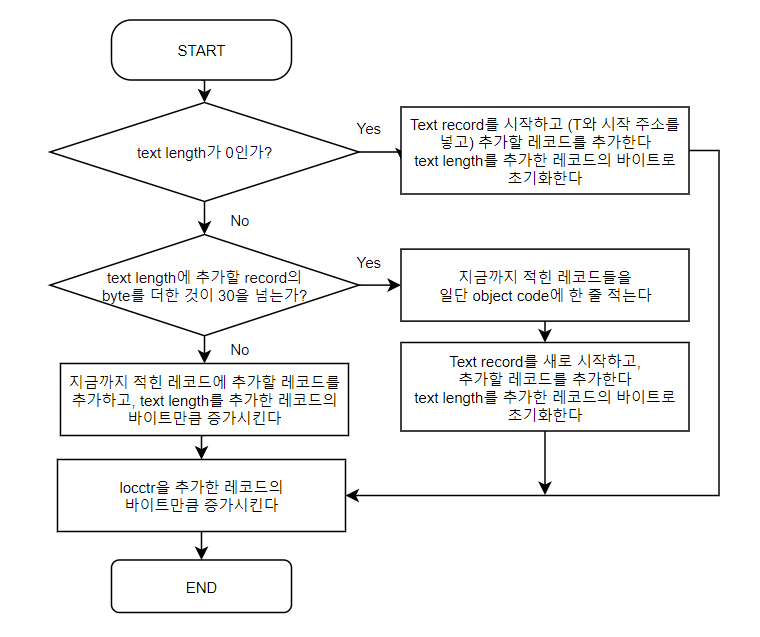
[그림 3-17] 해당 모듈의 flow chart

(3-5) modify record를 미리 만들어 저장하는 모듈



[그림 3-18] 해당 모듈의 flow chart

(3-6) 레코드를 추가하는 모듈



[그림 3-19] 해당 모듈의 flow chart

**(4) 출력 모듈은 table을 읽어 출력만 하므로 따로 flow chart를 그리지 않았다.**

(4-1) symbol tab 출력 모듈

(4-2) literal table 출력 모듈

(4-3) 최종 변환된 object code 출력 모듈

이러한 flow chart들을 기반으로 처음 설계해본 함수는 다음과 같다.

|  |  |
| --- | --- |
| 함수 기능 | 설계한 함수 |
| (1) 초기화 모듈 | int init\_my\_assembler() |
| (1-1) register 정보를 읽어와 register\_table을 만드는 모듈 | void init\_register\_number() |
| (1-2) SIC/XE 명령어 set을 읽어 inst\_table (OPTAB)을 초기화하는 모듈 | int init\_inst\_file(char \*inst\_file) |
| (1-3) input data를 읽어 저장하는 모듈. | int init\_input\_file(char \*input\_file) |
| (2) Pass1 모듈 | int assem\_pass1() |
| (2-1) input data를 token으로 분리하여 token unit으로 저장하는 모듈 | int token\_parsing(char \*str) |
| (2-2) operator의 byte 수를 반환하는 모듈 | int search\_n\_byte(char \*str) |
| (2-3) literal에 주소를 할당하는 모듈 | void add\_literal\_addr() |
| (3) Pass2 모듈 | int assem\_pass2() |
| (3-1) opcode를 찾는 모듈 | int search\_opcode(char \*str) |
| (3-2) symbol의 address를 찾는 모듈 | int symbol\_address(char \*str, int cs) |
| (3-3) literal의 address를 찾는 모듈 | int search\_literal\_table(char \*str) |
| (3-4) Register number를 찾는 모듈 | int search\_register(char \*x) |
| (3-5) modify record를 미리 만들어 저장하는 모듈 | void add\_modify\_unit(int addr, int length, char plus, char \*name, int section) |
| (3-6) 레코드를 추가하는 모듈 | int add\_text\_record(char \* buf, char \*record, char \* object, int \* tot\_length\_ptr, int byte) |
| (4) 출력 모듈 | - |
| (4-1) symbol tab 출력 모듈 | void make\_symtab\_output(char \*file\_name) |
| (4-2) literal table 출력 모듈 | void make\_literaltab\_output(char \*file\_name) |
| (4-3) 최종 object code 출력 모듈 | void make\_objectcode\_output(char \*file\_name) |

1. **시스템 구현 내용(구현 화면 포함)** 
   1. **전체 시스템 구현 내용**

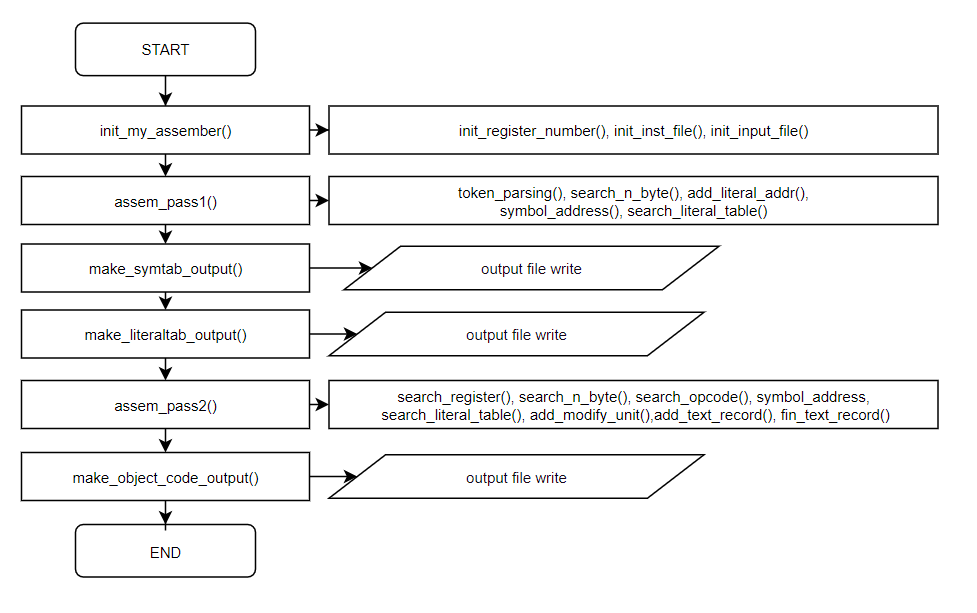
다음은 my\_assembler\_20160458.h안의 내용이다. 구조체는 설계한대로 만들어졌고, 필요에 따라 함수가 조금 추가되었다.

|  |
| --- |
| /\*  \* my\_assembler 함수를 위한 변수 선언 및 매크로를 담고 있는 헤더 파일이다.  \*/  #define MAX\_INST 256  #define MAX\_LINES 5000  #define MAX\_OPERAND 3  #define MAX\_SECTIONS 5 //최대 control section 개수  #define MAX\_NAME 6 //최대 변수 이름 길이  #define MAX\_COUNT 4096 //input data 한줄 읽을 때 최대 문자 수  /\*  \* instruction 목록 파일로 부터 정보를 받아와서 생성하는 구조체 변수이다.  \* 구조는 각자의 instruction set의 양식에 맞춰 직접 구현하되  \* 라인 별로 하나의 instruction을 저장한다.  \*/  struct inst\_unit  {  char \*mnemonic; //문자 명령어(LDA, STA 등)  char \*format; // 명령어 형식 1, 2, 3(=3/4)  int opcode; // 해당 문자 명령어의 opcode (+) int로 수정하였다  char \*n\_operand; // 오퍼랜드의 개수 (0,1,2)  };  // instruction의 정보를 가진 구조체를 관리하는 테이블 생성(opcode)  typedef struct inst\_unit inst;  inst \*inst\_table[MAX\_INST];  int inst\_index; // 지금까지 몇 개가 채워졌는지 알려주는 index  /\*  \* 어셈블리 할 소스코드를 입력받는 테이블이다. 라인 단위로 관리할 수 있다.  \*/  char \*input\_data[MAX\_LINES];  static int line\_num; // 지금까지 몇 라인 채워졌는지 저장하는 index  /\*  \* 어셈블리 할 소스코드를 토큰단위로 관리하기 위한 구조체 변수이다.  \* operator는 renaming을 허용한다.  \* nixbpe는 8bit 중 하위 6개의 bit를 이용하여 n,i,x,b,p,e를 표시한다.  \*/  struct token\_unit  {  char \*label; //명령어 라인 중 label  char \*operator\_sym; //명령어 라인 중 operator => operator symbol로 renaming했다.  char \*operand[MAX\_OPERAND]; //명령어 라인 중 operand, 최대 MAX\_OPERAND(=3)개  char \*comment; //명령어 라인 중 comment  char nixbpe; //하위 6bit 사용: \_ \_ n i x b p e  };  // token들을 관리할 수 있는 table  typedef struct token\_unit token;  token \*token\_table[MAX\_LINES];  static int token\_line; // 지금까지 몇개의 token\_unit이 채워졌는지 알려주는 index  /\*  \* 심볼을 관리하는 구조체이다.  \* 심볼 테이블은 심볼 이름, 심볼의 위치로 구성된다.  \*/  struct symbol\_unit  {  char symbol[10];  int addr;  };  typedef struct symbol\_unit symbol;  symbol sym\_table[MAX\_LINES];  static int sym\_index; //sym\_table에 얼만큼 채워졌는지 저장하는 index  /\*  \* 리터럴을 관리하는 구조체이다.  \* 리터럴 테이블은 리터럴의 이름, 리터럴의 위치로 구성된다.  \*/  struct literal\_unit  {  char literal[10];  int addr;  char format; //(+)Char(C), Hex(X)인지 구분하기 위해 추가하였다.  int section; //(+)이 literal이 어느 section에 있는지  };  typedef struct literal\_unit literal;  literal literal\_table[MAX\_LINES];  static int literal\_index; //sym\_table에 얼만큼 채워졌는지 저장하는 index  static int locctr;  static int cs\_length\_table[MAX\_SECTIONS]; //각 control section의 길이를 저장하는 table  int cs\_index; //cs\_length\_table이 얼마나 채워졌는지 관리하는 index  char object\_code[MAX\_LINES][MAX\_COUNT]; //Object code를 저장하는 buffer  static int object\_index; // 지금까지 object code가 몇 라인 채워졌는지 저장하는 index  /\*  \* 레지스터-번호를 저장하는 구조체이다.  \* 레지스터 이름과 그 번호로 구성된다.  \*/  struct register\_unit {  char \* name;  int number;  };  typedef struct register\_unit regi;  regi register\_table[MAX\_LINES]; //레지스터 정보들을 담고있는 table  static int regi\_index;  /\*  \* modify record를 저장하기 위한 구조체  \*/  struct m\_record\_unit {  int addr; //수정해야할 주소  int length; //수정해야할 (column)길이  char plus; //주소를 더해야하는지, 빼야하는지 기호  char name[MAX\_NAME + 1]; //더해야할 주소의 변수 이름  int section; //해당 record가 존재하는 section  };  typedef struct m\_record\_unit m\_record;  m\_record modify\_table[MAX\_LINES]; //M record 정보를 담고 있는 table  static int m\_index;  //------ 기본 함수 --------  static char \*input\_file;  static char \*output\_file;  int init\_my\_assembler(void);  int init\_inst\_file(char \*inst\_file);  int init\_input\_file(char \*input\_file);  int token\_parsing(char \*str);  int search\_opcode(char \*str);  static int assem\_pass1(void);  void make\_opcode\_output(char \*file\_name); //현재 프로젝트에서는 쓰이지 않음  void make\_symtab\_output(char \*file\_name);  void make\_literaltab\_output(char \*file\_name);  static int assem\_pass2(void);  void make\_objectcode\_output(char \*file\_name);    //------ 추가 함수 --------  int search\_n\_operand(char \*str); //명령어(mnemonic)를 주면 operand 개수를 리턴하는 함수  void init\_token(int index); //token을 처음 NULL들로 초기화해주는 함수  void data\_free(); //동적할당한 데이터들을 모두 해제하는 함수  int search\_n\_byte(char \*str); //명령어(mnemonic)를 주면 해당 명령어의 byte를 리턴하는 함수  int search\_literal\_table(char \*str); //literal을 주면 해당 literal의 주소를 반환하는 함수  void add\_literal\_addr(); //literal\_table의 literal에 address를 할당하는 함수  int symbol\_address(char \*str, int cs); //symbol 이름을 주면 address를 반환하는 함수  void init\_register\_number(); //register table을 초기화하는 함수  int search\_register(char \*x); //register의 number를 알려주는 함수  int add\_text\_record(char \* buf, char \*record, char \* object, int \* tot\_length\_ptr, int byte);  //현재 instruction을 text record에 추가하는 함수  void fin\_text\_record(char \* buf, char \*record, int \* tot\_length\_ptr);  //text record를 object program에 적고 끝내는 함수  void add\_modify\_unit(int addr, int length, char plus, char \*name, int section);  //modify table에 수정할 유닛을 추가하는 함수 |

설계와 다른 점은 token\_unit에서 operator라는 변수 이름을 쓸 수 없어서 operator symbol이라는 뜻으로 operator\_sym으로 이름을 수정하였다. 또한 input data file과 output data file 이름을 저장하는 변수가 각각 만들어졌다.

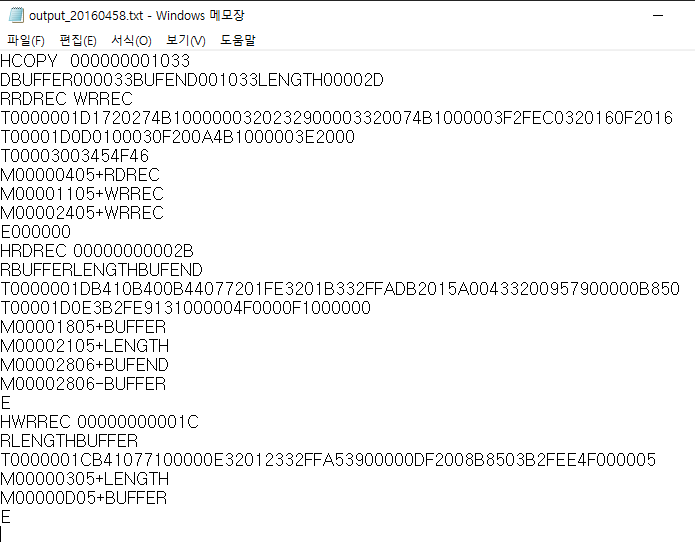
함수 몇 개가 추가되었다. token\_parsing에서 명령어의 operand 개수를 알아야 개수만큼 operand를 token으로 분리할 수 있기 때문에, operand 개수를 return해주는 search\_n\_operand라는 함수를 새로 만들게 되었다. 또한, token\_unit을 처음 만들었을 때, 주소에 쓰레기 값이 들어가 있는 것을 막기 위해 모든 멤버를 NULL로 초기화하는 init\_token이 추가되었다. text\_record를 끝내기만 하는 부분도 필요해서 fin\_text\_record라는 함수가 추가되었다. 마지막으로, 동적 할당한 메모리를 모두 해제하기 위해 data\_free라는 함수가 추가되었다.

아래는 메인 함수 흐름도에서, 각각의 함수들이 어떤 함수들을 호출하는지 나타낸 것이다.

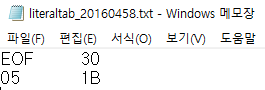


[그림 4-1] main함수의 흐름도와 각각 함수가 호출하는 함수

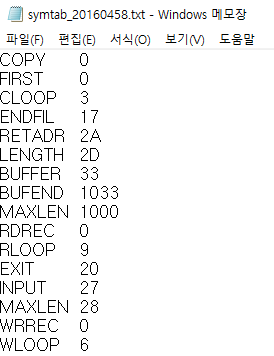
**프로그램 수행 결과**



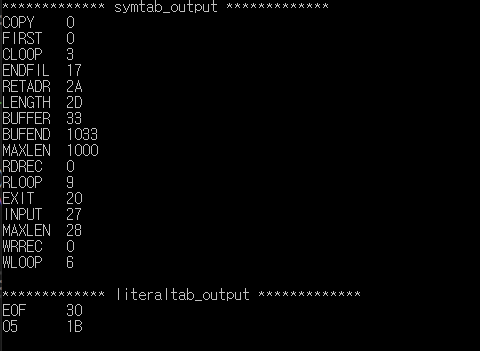
[그림 4-2] File에 출력된 최종 변환된 Object Code



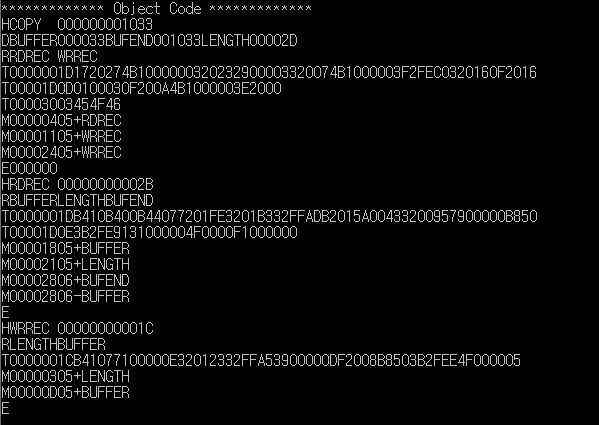
[그림 4-3] File에 출력된 LITTAB



[그림 4-4] File에 출력된 SYMTAB



[그림 4-5] Console에서 SYMTAB과 LITTAB의 출력



[그림 4-6] Console에서 최종 변환된 Object Code의 출력

위의 그림들은 실행 후 각각 콘솔에서, output file에서의 화면이다. 참고로 콘솔에 출력될 때는 output file 이름이 NULL일때 콘솔 창에 출력된다.

* 1. **모듈별 구현 내용**

함수 구현 내용이다. 주석으로 동작 과정들을 모두 설명하였다. 내용을 덧붙이는 경우, 몰랐던 사실이나 아쉬운 점에 대해 적었다.

1. **main**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 사용자로 부터 어셈블리 파일을 받아서 명령어의 OPCODE를 찾아 출력한다.  \* 매계 : 실행 파일, 어셈블리 파일  \* 반환 : 성공 = 0, 실패 = < 0  \* 주의 : 현재 어셈블리 프로그램의 리스트 파일을 생성하는 루틴은 만들지 않았다.  \* 또한 중간파일을 생성하지 않는다.  \* ----------------------------------------------------------------------------------  \*/  int main(int args, char \*arg[])  {    if (init\_my\_assembler() < 0)  {  printf("init\_my\_assembler: 프로그램 초기화에 실패 했습니다.\n");  return -1;  }    if (assem\_pass1() < 0)  {  printf("assem\_pass1: 패스1 과정에서 실패하였습니다. \n");  return -1;  }  //make\_opcode\_output("output\_20160458.txt");  make\_symtab\_output("symtab\_20160458.txt");  make\_literaltab\_output("literaltab\_20160458.txt");  if (assem\_pass2() < 0)  {  printf("assem\_pass2: 패스2 과정에서 실패하였습니다. \n");  return -1;  }  make\_objectcode\_output("output\_20160458.txt");    data\_free();  return 0;  } |

1. **init\_my\_assembler**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 프로그램 초기화를 위한 자료구조 생성 및 파일을 읽는 함수이다.  \* 매계 : 없음  \* 반환 : 정상종료 = 0 , 에러 발생 = -1  \* 주의 : 각각의 명령어 테이블을 내부에 선언하지 않고 관리를 용이하게 하기  \* 위해서 파일 단위로 관리하여 프로그램 초기화를 통해 정보를 읽어 올 수 있도록  \* 구현하였다.  \* ----------------------------------------------------------------------------------  \*/  int init\_my\_assembler(void)  {  int result;  init\_register\_number(); //register table도 초기화 시킨다.  if ((result = init\_inst\_file("inst.data")) < 0)  return -1;  if ((result = init\_input\_file("input.txt")) < 0)  return -1;  return result;  } |

1. **init\_inst\_file**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 머신을 위한 기계 코드목록 파일을 읽어 기계어 목록 테이블(inst\_table)을  \* 생성하는 함수이다.  \* 매계 : 기계어 목록 파일  \* 반환 : 정상종료 = 0 , 에러 < 0  \* 주의 : 기계어 목록파일 형식은 자유롭게 구현한다. 다음과 같이 inst.data에 저장되어있다.  \* ===============================================================================  \* 이름/형식/기계어 코드/오퍼랜드의 갯수/NULL  \* ===============================================================================  \* ----------------------------------------------------------------------------------  \*/  int init\_inst\_file(char \*inst\_file)  {  FILE \*file;  int errno;  char line\_buf[MAX\_COUNT]; // 한줄씩 읽을 때 저장하는 버퍼  char \* split; // 분리한 문장을 가리키는 ptr  if ((file = fopen(inst\_file, "rt")) == NULL) //file open이 되지 않을 경우 error  errno = -1;  else {  while (fgets(line\_buf, MAX\_COUNT-1, file) > 0 && inst\_index < MAX\_INST && line\_buf != NULL) {  //정상적으로 file open이 되었을 경우. 한줄씩 읽어들인다.  //buffer에 아무것도 없지 않고, index가 MAX를 넘지 않을때까지 읽어들인다.  inst\_table[inst\_index] = malloc(sizeof(inst));  // '/'가 token이므로 해당 문자를 이용하여 문자열을 분리한다.  // 분리 후, 해당하는 정보에 저장한다.  split=strtok(line\_buf, "/");  inst\_table[inst\_index]->mnemonic = malloc(strlen(split) + 1); //'\0'저장을 위해 +1  strcpy(inst\_table[inst\_index]->mnemonic, split);  //깊은 복사를 해야한다  split = strtok(NULL, "/");  inst\_table[inst\_index]->format = malloc(strlen(split) + 1);  strcpy(inst\_table[inst\_index]->format, split);  split = strtok(NULL, "/");  sscanf(split, "%X", &(inst\_table[inst\_index]->opcode)); //opcode(str)를 hex(int)로 고쳐 저장한다  split = strtok(NULL, "/");  inst\_table[inst\_index]->n\_operand= malloc(strlen(split) + 1);  strcpy(inst\_table[inst\_index]->n\_operand, split);  inst\_index++; //하나의 index가 꽉 찼으므로 올려준다.  }  fclose(file); //file을 꼭 닫아준다.  errno = 0;  }  return errno;  } |

1. **init\_input\_file**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 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;  char line\_buf[MAX\_COUNT]; // 한줄씩 읽은 문장을 저장하는 버퍼  if ((file = fopen(input\_file, "rt")) == NULL)  errno = -1; // file open error시 error flag 설정  else {  while ( fgets(line\_buf, MAX\_COUNT-1, file) > 0 && line\_num < MAX\_LINES && line\_buf != NULL) {  // input data에서 한줄씩 읽어들인다.    line\_buf[strlen(line\_buf) - 1] = '\0';  //개행문자를 제거한다. 추후 token에 개행이 들어가는 것을 막기 위함이다.  input\_data[line\_num] = malloc(strlen(line\_buf) + 1);  strcpy(input\_data[line\_num], line\_buf);  //해당 문자열만큼 공간을 잡고 깊은 복사를 한다.  line\_num++;  }  fclose(file);  errno = 0; //정상 종료시 error flag가 setting되지 않는다.  }  return errno;  } |

1. **token\_parsing**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 소스 코드를 읽어와 토큰단위로 분석하고 토큰 테이블을 작성하는 함수이다.  \* 패스 1로 부터 호출된다.  \* 매계 : 파싱을 원하는 문자열  \* 반환 : 정상종료 = 0 , 에러 < 0  \* 주의 : my\_assembler 프로그램에서는 라인단위로 토큰 및 오브젝트 관리를 하고 있다.  \* (+)operand가 3개인 경우를 고려하지 못했다.  \* ----------------------------------------------------------------------------------  \*/  int token\_parsing(char \*str)  {  if (str == NULL)  return -1;  int i=0, n\_operand;  char \* split = strtok(str, "\t"); // input data에서는 tab이 token이다.  token\_table[token\_line] = malloc(sizeof(token)); //먼저 token만큼 동적 할당한다.  init\_token(token\_line); //해당 token의 멤버들이 쓰레기 값을 갖지 않도록 초기화한다.  if (str[0] != '\t'){  token\_table[token\_line]->label = malloc(strlen(split) + 1);  strcpy(token\_table[token\_line]->label, split);  // Label이 있는 경우 Label을 저장한다.  split = strtok(NULL, "\t");  }    token\_table[token\_line]->operator\_sym = malloc(strlen(split) + 1);  strcpy(token\_table[token\_line]->operator\_sym, split); //opeartor를 저장한다.  n\_operand = search\_n\_operand(token\_table[token\_line]->operator\_sym);    //operand의 개수를 검색한다. 아래의 처리에서 이용된다.  if (split != NULL) {  if (n\_operand > 0) {  if (n\_operand == 1) { //operand가 하나인 경우  split = strtok(NULL, "\t");  //뒤에 ,가 없을 것이므로 tab앞까지가 operand이다.  token\_table[token\_line]->operand[i] = malloc(strlen(split) + 1);  strcpy(token\_table[token\_line]->operand[i], split);  //깊은 복사를 한다.  }  else { //operand가 2개 이상일 때  split = strtok(NULL, ",");  // ","으로 문장을 분리한다.    while (split != NULL && i < n\_operand) {  //operand 개수만큼 돈다.  token\_table[token\_line]->operand[i] = malloc(strlen(split) + 1);  strcpy(token\_table[token\_line]->operand[i], split); // operand에 저장한다.  i++;  if (i >= n\_operand)  break; //token을 또 분리하기 전에 나간다.  //operand2인 상태에서 token으로 분리해버리면 comment로 가기 때문이다.  split = strtok(NULL, "\t");  //comment 앞에서 operand를 읽기 위해 tab으로 분리한다.  //이렇게 하지 않으면 comment가 같이 딸려온다.  //대신 operand가 3개일 때는 처리하지 못했다.  }  }  }  else if (n\_operand < 0) {  // operand 개수가 -1인 경우 : 일단 지시자라고 생각을 한다.  // 지시자도 뒤에 (START, EXTDEF 등) 인수가 있을 수 있다.  // 없는 경우도 있기 때문에 꼭 NULL인지 체크가 필요하다.  split = strtok(NULL, "\t");  if (split != NULL) { //있는 경우, operand에다 넣어준다.  token\_table[token\_line]->operand[i] = malloc(strlen(split) + 1);  strcpy(token\_table[token\_line]->operand[i], split);  }  }  }  // comment 저장하는 부분  // operand가 없는 경우, (=0인 경우) 바로 comment로 넘어가게 된다.  split = strtok(NULL, "\t");  if (split != NULL) {  token\_table[token\_line]->comment = malloc(strlen(split)+1);  strcpy(token\_table[token\_line]->comment, split);  //comment가 있다면, 저장한다.  }    token\_line++; // 마찬가지로 index를 올려준다.  return 0;  } |

operand의 개수에 따라 처리를 해주어야 하는데, 0개와 1개일 때는 token 분리가 잘 되어 수월하게 진행했고, 2개일 때도 괜찮게 넘길 수 있었다. 그러나 MAX\_OPERAND가 3이라서 혹시 모를 경우를 대비해 operand가 3인 경우에도 처리를 하도록 해야 하는데, token을 분리하는 데 있어 어려움이 많아 3개일 때는 구현을 하지 못한 점이 아쉽다.

1. **search\_opcode**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 입력 문자열이 기계어 코드인지를 검사하는 함수이다.  \* 매계 : 토큰 단위로 구분된 문자열  \* 반환 : 정상종료 = 기계어 테이블 인덱스, 에러 < 0  \* ----------------------------------------------------------------------------------  \*/  int search\_opcode(char \*str)  {  int i = 0;  if (str[0] == '+') { // 4형식인 경우  str++; // str이 + 다음 것을 가리키도록 한다.  }    while (i < inst\_index) {  if (strcmp(str, inst\_table[i]->mnemonic) == 0) {  return i; // 알맞은 문자가 있을 경우, index를 리턴한다.  }  i++;  }  return -1;  } |

1. **assem\_pass1**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 어셈블리 코드를 위한 패스1과정을 수행하는 함수이다.  \* 1. 프로그램 소스를 스캔하여 해당하는 토큰단위로 분리하여 프로그램 라인별 토큰  \* 테이블을 생성한다.  \* 2. 프로그램 및 섹션의 크기를 구하고 모든 문장에 address를 할당한다.  \* 3. 지시자가 나오면 처리한다.  \* 4. SYMTAB과 LITTAB을 만든다.  \* 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)  {  int i = 0;  int byte = 0;  while (i < line\_num) { //input\_data line\_num까지  if (sizeof(input\_data[i]) == 0 || input\_data[i][0] == '.' ) {  i++;  continue; //혹시나 문장이 빈 경우, 주석인 경우 넘어간다.  }    if (token\_parsing(input\_data[i]) < 0) { //token\_parsing 호출  return -1; //error인 경우  }  token\_table[token\_line - 1]->nixbpe = 0; //쓰레기값 방지  if (token\_table[token\_line - 1]->label != NULL) { //label(symbol)이 있는 경우  //parsing을 하고 왔기 때문에 token line-1을 해줘야 방금 추가한 token으로 간다  if (symbol\_address(token\_table[token\_line - 1]->label,cs\_index)>=0) {  //해당 symbol이 있다면 중복된 symbol이 있는 것이므로  return -1;  }  //symbol table 만듦  strcpy(sym\_table[sym\_index].symbol, token\_table[token\_line - 1]->label);  sym\_table[sym\_index].addr = locctr;  //CSECT인 경우 밑에서 따로 처리를 또 해준다.  sym\_index++;  }  if ((byte = search\_n\_byte(token\_table[token\_line - 1]->operator\_sym)) > 0) {  //opcode일 경우  locctr += byte; //locctr 증가  //nixbpe에서 nix,e는 설정 가능 (bp는 나중에 pass2에서 설정)  if (token\_table[token\_line - 1]->operator\_sym[0] == '+') {  //4형식인 경우, e=1 설정(bit or)  token\_table[token\_line - 1]->nixbpe |= 1;  }  if (token\_table[token\_line - 1]->operand[0] != NULL && token\_table[token\_line - 1]->operand[0][0] == '@') {  // indirect, n=1 설정  token\_table[token\_line - 1]->nixbpe |= 32;  }  else if(token\_table[token\_line - 1]->operand[0] != NULL && token\_table[token\_line - 1]->operand[0][0] == '#') {  // immidiate, i=1 설정  token\_table[token\_line - 1]->nixbpe |= 16;  }  else {  //direct address이거나, operand가 없더라도 SIC/XE이므로 n, i 둘다 1 설정  //2형식 data도 nixbpe를 갖게 되긴 한다.  token\_table[token\_line - 1]->nixbpe |= 48;  }  if (token\_table[token\_line - 1]->operand[0] != NULL) {  char \* str = strtok(token\_table[token\_line - 1]->operand[0], ",");  //X가 있는 경우를 위해 한 번 strtok를 한다. 만약 ,이 없다면 operand가 그대로 들어감  str = strtok(NULL, ",");  // 다시 한 번 token 분리. ,이 없다면 NULL이 나온다  //strtok는 잘리는 문자가 잘린 상태 그대로 있기 때문에  //이후 opearnd[0]는 ,X없는 operand를 가지게 된다  if (str != NULL && str[0]=='X') { // x=1로 만들어준다  token\_table[token\_line - 1]->nixbpe |= 8;  }  }  }  else { //directives인 경우  if (strcmp(token\_table[token\_line - 1]->operator\_sym, "CSECT") == 0) { //CSECT인 경우  cs\_length\_table[cs\_index] = locctr;  //program length를 저장한다. (현재 locctr)  //여기서는 start address를 따로 저장하지 않는다. 항상 0이라고 가정.  cs\_index++; //control section이 새로 생긴다  locctr = 0; //locctr 초기화  sym\_table[sym\_index-1].addr = locctr;  //sym\_tab의 label의 address 값을 다시 고친다.  }  else if (strcmp(token\_table[token\_line - 1]->operator\_sym, "EQU") == 0) {  if (token\_table[token\_line - 1]->operand[0][0] != '\*') {  // \* 이면 그냥 현재 locctr을 넣으면 되지만 아닐 경우, 따로 값을 구해 저장해야한다  //일단 연산을 해야할 경우, -에 대해서만 구분하였다  // EQU MAXLEN, EQU BUFEND-BUFFER는 가능하지만, EQU 4096이나, +인 경우는 불가능하다.  char \* str = malloc(strlen(token\_table[token\_line - 1]->operand[0]) + 1);  strcpy(str, token\_table[token\_line - 1]->operand[0]);  //strtok를 쓰면 operand[0] 자체도 잘린 채로 저장되기 때문에  //나중에 pass2에서도 전체가 쓰이는 것들은 따로 복사해 사용한다.  char \* sub\_str= strtok(str, "-"); // -가 있는 경우  sym\_table[sym\_index-1].addr = symbol\_address(sub\_str,cs\_index);  //현재 section에 대해서만 찾는다 변수가 겹치더라도 가능하다    sub\_str = strtok(NULL, "-");  if(str!=NULL) //만약 - 뒤에 변수가 있다면  sym\_table[sym\_index - 1].addr -= symbol\_address(sub\_str, 0); // 마이너스를 계산해 저장한다  free(str); //동적할당한 str은 다시 메모리 free한다  }    }  else if (strcmp(token\_table[token\_line - 1]->operator\_sym, "RESW") == 0) {  locctr += atoi(token\_table[token\_line - 1]->operand[0]) \* 3; //operand\*3만큼 할당  }  else if (strcmp(token\_table[token\_line - 1]->operator\_sym, "RESB") == 0) {  locctr += atoi(token\_table[token\_line - 1]->operand[0]); //operand만큼 할당  }  else if (strcmp(token\_table[token\_line - 1]->operator\_sym, "WORD") == 0) {  locctr += 3; //word이므로 3만 더한다  }  else if (strcmp(token\_table[token\_line - 1]->operator\_sym, "BYTE") == 0) {  if (token\_table[token\_line - 1]->operand[0][0] == 'X') {  //HEX인 경우  //X''를 빼고 2column당 1byte이므로 2로 나눔 (BYTE X'F1')  locctr += (strlen(token\_table[token\_line - 1]->operand[0])-3)/2;  }  else if(token\_table[token\_line - 1]->operand[0][0] == 'C'){ //Char인경우  //operand length만큼 할당 (C''를 뺌) (BYTE C'EOF')  locctr += strlen(token\_table[token\_line - 1]->operand[0])-3;  }  }  else if (strcmp(token\_table[token\_line - 1]->operator\_sym, "LTORG") == 0) {  //LTORG를 만났으니 저장만 해놨던 literal들에 대해 진짜 주소를 할당한다.  add\_literal\_addr();  }  else if (strcmp(token\_table[token\_line - 1]->operator\_sym, "END") == 0) {  add\_literal\_addr();  //끝내기 전 주소가 할당 안된 literal들에 대해 주소 할당  cs\_length\_table[cs\_index] = locctr;// program length 저장  cs\_index++;  }  //pass1에서 EXTDEF, EXTREF는 넘어간다.  }  if (token\_table[token\_line - 1]->operand[0] != NULL && token\_table[token\_line - 1]->operand[0][0]=='=') {  //operand가 literal인 경우  char \* str = malloc(strlen(token\_table[token\_line - 1]->operand[0])+1);  strcpy(str, token\_table[token\_line - 1]->operand[0]);  //나중에 또 쓰기위한 깊은 복사  //이렇게 하지 않으면 operand[0]에 =C나 =X만 남아있게 된다.  char \* sub\_str = strtok(str, "'");// '로 구분  sub\_str = strtok(NULL, "'"); //X'F1'에서 F1을 가리킴  if (search\_literal\_table(sub\_str) < 0) {  //literal table에 있는지 검사하고 없다면 추가  strcpy(literal\_table[literal\_index].literal, sub\_str);  literal\_table[literal\_index].format= token\_table[token\_line - 1]->operand[0][1];  //literal이 C인지 X인지 형식을 추가한다  literal\_table[literal\_index].section = cs\_index;  //현재 control section index를 넣는다.  literal\_table[literal\_index].addr = 0;  //주소는 모르므로 0으로 할당한다  literal\_index++;  }    free(str);  }  i++;    }  return 0;  } |

코드가 긴만큼 아쉬운 점이 많은 함수이다. 에러 처리는 다음과 같다. (1) token\_parsing이 error인 경우 -1(에러)를 리턴 (2) Label을 SYMTAB에 저장하기 위해 해당 symbol이 table에 있는지 검사하는데, 만약 있다면 symbol이 중복된 것이므로 에러를 리턴한다. 이것 말고도 다른 에러 체크사항이 있는지 알지 못하여 아쉬웠다.

또한, 주석에서도 볼 수 있겠지만 여기서는 START인 경우를 처리해주지 않고 있다. 프로그램 길이만 구하면 되기 때문에 start address를 딱히 저장할 이유가 없다. Start address는 assem\_pass2에서 저장한다. 또, SIC Machine 명령어는 n=0, i=0인데, 이에 대한 처리도 없다. #나 @가 없으면 모두 direct 모드로 보고 n=1, i=1로 설정한 것이 아쉽다. 마지막으로 EQU 값을 계산할 때 EQU \*, EQU BUFFER, EQU BUFEND-BUFFER까지는 가능하지만, EQU 4096(상수) EQU BUFEND+BUFFER(더하기)는 처리해주지 못했다.

1. **search\_n\_operand**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 명령어의 operand 개수를 찾는 데 쓰이는 함수이다.  \* 매계 : operator name  \* 반환 : operand 개수, 없는 operator name일 경우 -1 반환  \* -----------------------------------------------------------------------------------  \*/  int search\_n\_operand(char \*str) {  int index=search\_opcode(str);  if (index < 0)  return -1;  else  return atoi(inst\_table[index]->n\_operand);  } |

1. **data\_free**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 동적 할당한 메모리를 해제하는 작업을 하는 함수이다.  \* 매계 : 없음  \* 반환 : 없음  \* -----------------------------------------------------------------------------------  \*/  void data\_free() {  // input\_data 해제  for (int i = 0; i < line\_num; i++) {  free(input\_data[i]);  }  // token\_table 해제  for (int i = 0; i < token\_line; i++) {  if (token\_table[i]->label != NULL)  free(token\_table[i]->label);  for (int j = 0; j < MAX\_OPERAND; j++) {  if(token\_table[i]->operand[j]!=NULL)  free(token\_table[i]->operand[j]);  }  if (token\_table[i]->comment != NULL)  free(token\_table[i]->comment);  free(token\_table[i]->operator\_sym);  free(token\_table[i]);  }  //inst\_table 해제  //token\_table이 search\_opcode를 쓰기 때문에 나중에 해제해주어야 한다.  for (int i = 0; i < inst\_index; i++) {  free(inst\_table[i]->format);  free(inst\_table[i]->mnemonic);  free(inst\_table[i]->n\_operand);  //opcode는 int이므로 따로 해제할 필요 없다.  free(inst\_table[i]);  }  // symtab, literal tab, object\_code는 동적 할당이 아니므로 해제할 필요 없다.    } |

1. **init\_token**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 처음 만든 token의 멤버들을 NULL로 초기화해주는 함수  \* 매계 : 초기화할 token을 가리키는 token\_table의 index  \* 반환 : 없음  \* -----------------------------------------------------------------------------------  \*/  void init\_token(int index) { // 쓰레기 값이 들어가는 것을 모두 NULL로 초기화 해준다.  token\_table[index]->label = NULL;  token\_table[index]->operator\_sym = NULL;  token\_table[index]->comment = NULL;  for (int i = 0; i < MAX\_OPERAND; i++) {  token\_table[index]->operand[i] = NULL;  }  } |

1. **make\_symtab\_output**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 입력된 문자열의 이름을 가진 파일에 프로그램의 결과를 저장하는 함수이다.  \* 여기서 출력되는 내용은 SYMBOL별 주소값이 저장된 TABLE이다.  \* 매계 : 생성할 오브젝트 파일명  \* 반환 : 없음  \* 주의 : 만약 인자로 NULL값이 들어온다면 프로그램의 결과를 표준출력으로 보내어  \* 화면에 출력해준다.  \* -----------------------------------------------------------------------------------  \*/  void make\_symtab\_output(char \*file\_name)  {  FILE\* file;  char line[MAX\_COUNT]; //출력 line을 저장하는 버퍼  char hexaddr[MAX\_COL]; //hex 주소를 저장할 버퍼  if (file\_name == NULL) {  // 인자로 NULL값이 들어올 경우.  // 콘솔에 결과를 출력한다.  printf("\*\*\*\*\*\*\*\*\*\*\*\*\* symtab\_output \*\*\*\*\*\*\*\*\*\*\*\*\*\n");  for (int i = 0; i < sym\_index; i++) {  printf("%s\t%X\n", sym\_table[i].symbol, sym\_table[i].addr); //16진수로 출력  }  return;  }  // filename이 NULL이 아닐경우  if ((file = fopen(file\_name, "w+t")) == NULL)  return;  else { // file이 정상적으로 열렸을 경우  for (int i = 0; i < sym\_index; i++) {  line[0] = '\0';  strcat(line, sym\_table[i].symbol);  strcat(line, "\t");  sprintf(hexaddr, "%X", sym\_table[i].addr);  strcat(line, hexaddr); //address hex로 변환  strcat(line, "\n");  fwrite(line, strlen(line), 1, file);  // file에 한 줄씩 적는다.  }  fclose(file);  }  } |

1. **make\_literaltab\_output**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 입력된 문자열의 이름을 가진 파일에 프로그램의 결과를 저장하는 함수이다.  \* 여기서 출력되는 내용은 LITERAL별 주소값이 저장된 TABLE이다.  \* 매계 : 생성할 오브젝트 파일명  \* 반환 : 없음  \* 주의 : 만약 인자로 NULL값이 들어온다면 프로그램의 결과를 표준출력으로 보내어  \* 화면에 출력해준다.  \* -----------------------------------------------------------------------------------  \*/  void make\_literaltab\_output(char \*file\_name)  {  FILE\* file;  char line[MAX\_COUNT]; //출력 line을 저장하는 버퍼  char hexaddr[MAX\_COL]; //hex 주소를 저장할 버퍼  if (file\_name == NULL) {  // 인자로 NULL값이 들어올 경우.  // 콘솔에 결과를 출력한다.  printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\* literaltab\_output \*\*\*\*\*\*\*\*\*\*\*\*\*\n");  for (int i = 0; i < literal\_index ; i++) {  printf("%s\t%X\n", literal\_table[i].literal, literal\_table[i].addr); //16진수로 출력  }  return;  }  // filename이 NULL이 아닐경우  if ((file = fopen(file\_name, "w+t")) == NULL)  return;  else { // file이 정상적으로 열렸을 경우  for (int i = 0; i < literal\_index; i++) {  line[0] = '\0';  strcat(line, literal\_table[i].literal);  strcat(line, "\t");  sprintf(hexaddr, "%X", literal\_table[i].addr);  strcat(line, hexaddr); //hex로 변환해야함;;  strcat(line, "\n");  fwrite(line, strlen(line), 1, file);  // file에 한 줄씩 적는다.  }  fclose(file);  }  } |

1. **assem\_pass2**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 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  \* -----------------------------------------------------------------------------------  \*/  static int assem\_pass2(void)  {  int i = 0;  int byte = 0; //현재 명령어의 byte를 저장하는 변수  int pc = 0; //pc값을 저장하는 변수  int text\_length = 0; //현재 text record에 얼만큼 썼는지 나타내는 변수  int current\_cs = 0; //현재 current section index 변수  char buf[MAX\_COUNT]; //최종 text record를 저장하는 변수  char record[100]; //T, length 뒤 object code들만을 위한 buffer  char object[MAX\_COL + 1];  //2~4byte의 object code 하나(opcode+nixbpe+disp)만을 담기 위한 buffer  int opcode; //opcode를 저장하는 변수  int disp = 0; //object code에서 disp부분  int op\_index = 0; //해당 opcode가 존재하는 table index를 저장하는 변수  int cur\_literal = 0; //현재 object code에 record한 literal table index  int cur\_modify = 0; //현재 object code에 record한 modify table index  int start\_addr = 0; //프로그램 시작 주소를 저장하는 변수  while (i < token\_line) { //token 단위로 읽어 object code를 만든다.  //Header 씀(CESCT, START인 경우)  if (strcmp(token\_table[i]->operator\_sym, "START") == 0) {  locctr = 0; //locctr 초기화  pc = 0; //pc 초기화  start\_addr = atoi(token\_table[i]->operand[0]); //start address 저장  sprintf(buf,"H%-6s%06X%06X",token\_table[i]->label,start\_addr,cs\_length\_table[current\_cs]);  //H record를 쓴다. (program name왼쪽정렬, 시작주소와 section length를 적음)  strcpy(object\_code[object\_index], buf);  object\_index++;  }  else if(strcmp(token\_table[i]->operator\_sym, "CSECT") == 0){  //끝나지 않은 record가 있을 수 있으므로 record를 먼저 끝낸다  fin\_text\_record(buf, record, &text\_length);  //section을 끝내기 전에 Modify record 쓰기  while (cur\_modify < m\_index && modify\_table[cur\_modify].section == current\_cs) {  sprintf(buf, "M%06X%02X%c%s", modify\_table[cur\_modify].addr, modify\_table[cur\_modify].length, modify\_table[cur\_modify].plus, modify\_table[cur\_modify].name);  strcat(object\_code[object\_index], buf);  object\_index++;  cur\_modify++;  }  //section을끝내기 전에 END record를 작성한다  if (current\_cs == 0) {  //주 프로그램일 경우, END record에 첫 시작주소를 적어야한다  sprintf(buf, "E%06X", start\_addr);  strcpy(object\_code[object\_index], buf);  }  else {  strcpy(object\_code[object\_index], "E");  //subprogram일 경우, E만 쓰고 끝낸다  }  object\_index++;  locctr = 0; //CSECT시작, locctr 초기화  pc = 0; //pc 초기화  current\_cs++;  //새 section이 시작되었으므로 current\_section도 하나 증가시킨다.  sprintf(buf, "H%-6s%06X%06X", token\_table[i]->label, 0, cs\_length\_table[current\_cs]);  //CSECT는 0이 시작주소이므로 0을 시작주소로 적는다  strcpy(object\_code[object\_index], buf);  object\_index++;  }  else if (strcmp(token\_table[i]->operator\_sym, "EXTDEF") == 0) {  //D record 작성  object\_code[object\_index][0] = 'D';  char \* str = strtok(token\_table[i]->operand[0],","); // , 로 구분한다  while (str != NULL) {  sprintf(buf, "%s%06X",str, symbol\_address(str,current\_cs));  strcat(object\_code[object\_index], buf);  //밖으로 내보낼 symbol들과 그 주소를 차례로 적는다.  str = strtok(NULL, ",");  }  object\_index++;  }  else if (strcmp(token\_table[i]->operator\_sym, "EXTREF") == 0) {  //R record 작성  object\_code[object\_index][0] = 'R';  char \* str = strtok(token\_table[i]->operand[0], ",");// , 로 구분한다  while (str != NULL) {  sprintf(buf, "%-6s",str);  strcat(object\_code[object\_index], buf);  //외부 주소는 모르므로 이름만 적는다  str = strtok(NULL, ",");  }  object\_index++;  }  else if (strcmp(token\_table[i]->operator\_sym, "END") == 0) {  //program이 끝이난 경우  //끝나기 전 적지 않은 literal record들을 적는다  while (cur\_literal < literal\_index && literal\_table[cur\_literal].section == current\_cs) {  // 현재 section에 대해서만  if (literal\_table[cur\_literal].format == 'C') {  //Char인 경우  char \* object\_ptr = object;  int size = strlen(literal\_table[cur\_literal].literal);  for (int j = 0; j < size; j++) {  //길이만큼 문자 하나하나의 ASCII code를 HEX로 적는다.  sprintf(object\_ptr, "%X", literal\_table[cur\_literal].literal[j]);  object\_ptr += 2;  }  byte = strlen(literal\_table[cur\_literal].literal); //Char byte  }  else { //X인 경우  sprintf(object, "%s", literal\_table[cur\_literal].literal); //HEX 문자 그대로 저장  byte = strlen(literal\_table[cur\_literal].literal) / 2; //HEX byte  }  add\_text\_record(buf, record, object, &text\_length, byte);  cur\_literal++;  }//end of while  fin\_text\_record(buf, record, &text\_length); //literal record 종료    //Modify record 쓰기  while (cur\_modify < m\_index && modify\_table[cur\_modify].section == current\_cs) {  //index 비교해주지 않으면 계속 참조하게 된다.  //현재 section에 있는 modify record만 적는다  sprintf(buf, "M%06X%02X%c%s", modify\_table[cur\_modify].addr, modify\_table[cur\_modify].length, modify\_table[cur\_modify].plus, modify\_table[cur\_modify].name);  strcat(object\_code[object\_index], buf);  object\_index++;  cur\_modify++;  }  //그리고 E record를 쓴다.  strcpy(object\_code[object\_index], "E");  object\_index++;  }  else {  //Word,Byte,나머지 operator인 경우에는 Text record 씀  //text record의 경우 text\_length를 이용해 길이를 저장해 놓는다  if ((byte = search\_n\_byte(token\_table[i]->operator\_sym)) > 0) {  // opcode일 경우  //nixbpe에서 b,p에 대한 값을 설정해주어야 한다  //이 예제에서 b인 경우가 없어서 b=1이되는 경우는 없다  if (token\_table[i]->nixbpe & 1) {  //4형식인 경우  pc += 4; //pc추가  op\_index = search\_opcode(token\_table[i]->operator\_sym);  opcode = inst\_table[op\_index]->opcode;  opcode = opcode << 4;  //nixbpe들어갈 자리 만들기 위해 left shift한다  opcode |= token\_table[i]->nixbpe;  //bit or로 nixbpe를 추가한다  //strtok로 ,X가 있는 경우 이미 분리시켰으므로 처리해줄 필요가 없다.  disp = symbol\_address(token\_table[i]->operand[0], current\_cs); //operand symbol 찾기  if (disp < 0) {  // symtab에 없는 경우  disp = 0;  //modify table에 unit을 추가한다  add\_modify\_unit(locctr+1,5,'+', token\_table[i]->operand[0],current\_cs);  //고칠 곳의 주소는 locctr+1 고칠 길이는 5로 설정한다  }  else {  // symtab에 있는 경우  opcode |= 2;  //nixbpe에서 p=1로 만들어줌(사실 이부분에서 오류 검사를 해야한다)  disp -= pc; //pc를 빼서 disp를 설정한다  }  if (disp < 0) {  //disp가 음수일 경우! 하위 5column만 필요하므로 나머지는 0으로 만들어줘야 에러가 나지 않는다.  //5byte 빼고는다 0으로 (bit and)  disp &= (int)pow(2,20) -1 ;  }  sprintf(object,"%03X%05X", opcode,disp);  add\_text\_record(buf, record, object, &text\_length, 4); //해당 record를 추가한다  //여기서 locctr은 위의 함수에서 증가한다  }  else if (byte==2) {  //2형식인 경우  pc += 2;  op\_index = search\_opcode(token\_table[i]->operator\_sym);  opcode = inst\_table[op\_index]->opcode;  //2형식은 nixbpe를 사용하지 않는다.  sprintf(object, "%02X%X%X", opcode, search\_register(token\_table[i]->operand[0]), search\_register(token\_table[i]->operand[1]));  //register 번호를 찾아와 넣는다  add\_text\_record(buf, record, object, &text\_length, 2); //record 추가  }  else if (search\_n\_operand(token\_table[i]->operator\_sym) == 1) {  //3형식이지만 operand가 하나인 경우  pc += 3;  op\_index = search\_opcode(token\_table[i]->operator\_sym);  opcode = inst\_table[op\_index]->opcode;  opcode = opcode << 4;  opcode |= token\_table[i]->nixbpe;  char \* str = token\_table[i]->operand[0];  if ((token\_table[i]->nixbpe & 48) == 48) {  //16 & 48 하면 16으로 출력됨. ==로 n,i 둘다 1인지 확인해야 한다  //n=1, i=1인 경우  if (token\_table[i]->operand[0][0] == '=') {  //operand가 literal인 경우  char \* sub\_str = strtok(str, "'");  // '로 구분  sub\_str=strtok(NULL, "'");  //X'', C''을 뗀다    int index = search\_literal\_table(sub\_str);  disp = literal\_table[index].addr; //literal의 address를 받아온다  disp -= pc; //disp를 계산한다  opcode |= 2;  //pc를 빼서 disp를 계산했으므로 p=1로 만들어준다.  }  else {  //평범한 symbol의 경우  //4형식이 아니므로 operand가 symtab에 있어야 한다  disp = symbol\_address(str, current\_cs); //현재 section의 symbol 주소를 찾아온다  disp -= pc; //disp 계산  opcode |= 2; //p=1로 만들어준다.  }    }  else if (token\_table[i]->nixbpe & 16) {  //i만1, immediate  // # 분해  char \* sub\_str = strtok(str, "#");  //바로 #뒤의 숫자들이 들어간다.  disp = atoi(sub\_str); //string to int  // p를 1로 만들어주지 않는다. 진짜 상수이기 때문에.  // \*\* #MAXLEN 같은 경우는 사용할 수 없다. \*\*  }  else {  // n만 1, indirect  // n=0, i=0인 경우도 여기 포함되지만 처리되지는 않음  // @뒤의 값을 symtab에서 찾아 object 만듦  char \* sub\_str = strtok(str, "@");  disp = symbol\_address(sub\_str, current\_cs);  disp -= pc;  opcode |= 2; //상대주소이므로 p=1로 만들어준다  }  if (disp < 0) {  //만약 disp가 음수인 경우, 하위 3column만 들어갈 수 있도록 bit and를 한다  disp &= (int)pow(2, 12) - 1;  }  sprintf(object, "%03X%03X", opcode, disp);  //추가할 record 만들기  add\_text\_record(buf, record, object, &text\_length, 3);  //record 추가  }  else {  //3형식이지만 operand가 없는 경우 (1형식인 경우도 여기 들어가지만 처리는 못한다)  pc += 3;  op\_index = search\_opcode(token\_table[i]->operator\_sym);  opcode = inst\_table[op\_index]->opcode;  opcode = opcode << 4;  opcode |= token\_table[i]->nixbpe; //pc는 여전히 0이다  sprintf(object, "%03X%03X", opcode, 0);  //disp 부분은 0으로 처리해준다  add\_text\_record(buf, record, object, &text\_length, 3);  //record 추가  }    }  else { //지시자인 경우    if (strcmp(token\_table[i]->operator\_sym, "RESW") == 0) {  //text record가 끝나지 않았다면 끝내야한다  fin\_text\_record(buf, record, &text\_length);  locctr += atoi(token\_table[i]->operand[0]) \* 3;  //locctr만 증가시킨다  }  else if (strcmp(token\_table[i]->operator\_sym, "RESB") == 0) {  //text record가 끝나지 않았다면 끝내한다  fin\_text\_record(buf, record, &text\_length);  locctr += atoi(token\_table[i]->operand[0]);  }  else if (strcmp(token\_table[i]->operator\_sym, "WORD") == 0) {  // 상수,symbol하나,"-"가 있는 경우에서만 처리할 수 있다.  // +가 있다면 처리하지 못한다  byte = 3;  pc += byte;  char \* str = strtok(token\_table[i]->operand[0], "-");  disp = atoi(str); //WORD 뒤에 상수가 들어간 경우  if (disp == 0) {  disp = symbol\_address(str, current\_cs);  //상수가 아닌 경우, symbol이므로 주소를 찾아온다  }  if (disp < 0) {  //symtab에 없는 경우  //modify unit을 추가  add\_modify\_unit(locctr, 6, '+', str, current\_cs);  disp = 0;  //WORD이므로 고칠 곳의 주소는 locctr그 자체, 고칠 곳의 길이는 6이다.  }    str = strtok(NULL, "-");  if (str != NULL) {  int sub\_disp = symbol\_address(str, current\_cs);  if (sub\_disp < 0) {  //symtab에 없는 경우, 똑같이 modify unit을 추가한다  add\_modify\_unit(locctr, 6, '-', str, current\_cs);  //마이너스 뒤이므로 '-'를 넣어주어야 한다  }  else {  disp -= sub\_disp;  //빼는 곳의 주소가 나왔을 경우 target address를 만들기 위해 뺀다!  }  }  if (disp < 0) {  //disp가 음수인경우 3column만 빼고 다 0으로 만든다  disp &= (int)pow(2, 12) - 1;  }  sprintf(object, "%06X", disp);  add\_text\_record(buf, record, object, &text\_length, 3);  }  else if (strcmp(token\_table[i]->operator\_sym, "BYTE") == 0) {  //BYTE는 -, +에 대한 처리를 따로 해주지 않았다  char \* str = strtok(token\_table[i]->operand[0], "'");  str = strtok(NULL, "'"); //C'' 또는 X''에서 분리    if (token\_table[i]->operand[0][0] == 'C') {  char \* object\_ptr = object;  int size = strlen(str);  for (int j = 0; j < size; j++) {  object\_ptr += sprintf(object\_ptr, "%X", str[j]);  //한 byte마다 ascii code(hex)로 적음  }  byte = size;  }  else { //X인 경우  sprintf(object, "%02s",str);  //byte이므로 그대로 적음  byte = strlen(str) / 2;  }  add\_text\_record(buf, record, object, &text\_length, byte);  }  else if (strcmp(token\_table[i]->operator\_sym, "LTORG") == 0) {  //저장한 literal들을 text record에 적어야한다  //현재 section에 대한 literal들만 적는다    while(literal\_table[cur\_literal].section == current\_cs){  if (cur\_literal <literal\_index && literal\_table[cur\_literal].format == 'C') {  char \* object\_ptr = object;  int size = strlen(literal\_table[cur\_literal].literal);  for (int j = 0; j < size; j++) {  sprintf(object\_ptr, "%2X", literal\_table[cur\_literal].literal[j]);  object\_ptr += 2;  //문자를 하나하나 ASCII code(HEX)로 적는다  }  byte = strlen(literal\_table[cur\_literal].literal);  }  else { //X인 경우  sprintf(object, "%s", literal\_table[cur\_literal].literal);  byte = strlen(literal\_table[cur\_literal].literal) / 2;  }  add\_text\_record(buf, record, object, &text\_length, byte);  cur\_literal++;  }//end of while  fin\_text\_record(buf, record, &text\_length);  // literal record를 object code에 적고 끝낸다  }  }    }  i++;  }  return 0;  } |

마찬가지로 가장 긴 코드인만큼 아쉬움이 많은 함수이다. COPY program에서는 모두 PC relative 주소라서 Base relative 주소를 사용하는 부분이 없었다. SIC/XE 머신에서 주소를 정할 때, 우선 순위가 (1) PC relative (2) Base relative (3) Extend foramt 순이다. 하지만 이 코드에서는 오로지 PC relative만 계산한다. (4형식으로 따로 표시되어 있는 부분 제외)

또한 Immediate Addressing Mode에서 #3같은 상수는 가능하지만, #MAXLEN같이 뒤에 symbol이 들어가있는 경우는 처리해주지 못했다. n=0, i=0인 경우와 1형식 명령어가 들어왔을 때에도 마찬가지로 처리해주지 못했다.

WORD 지시자를 처리할 때, WORD 3, WORD Symbol, WORD BUFEND-BUFFER처럼 상수, 심볼 하나, 마이너스까지는 처리가 가능하다. 그러나 +인 경우는 따로 처리하지 못했다. BYTE 지시자도 +와 –연산으로 적는 것이 가능할 것이다. 그러나 처리해주지 못했다.

1. **make\_objectcode\_output**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 입력된 문자열의 이름을 가진 파일에 프로그램의 결과를 저장하는 함수이다.  \* 여기서 출력되는 내용은 object code (프로젝트 1번) 이다.  \* 매계 : 생성할 오브젝트 파일명  \* 반환 : 없음  \* 주의 : 만약 인자로 NULL값이 들어온다면 프로그램의 결과를 표준출력으로 보내어  \* 화면에 출력해준다.  \* -----------------------------------------------------------------------------------  \*/  void make\_objectcode\_output(char \*file\_name)  {  FILE\* file;  if (file\_name == NULL) {  // 인자로 NULL값이 들어올 경우.  // 콘솔에 결과를 출력한다.  printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\* Object Code \*\*\*\*\*\*\*\*\*\*\*\*\*\n");  for (int i = 0; i < object\_index; i++) {  printf("%s\n", object\_code[i]);  }  return;  }  // filename이 NULL이 아닐경우  if ((file = fopen(file\_name, "w+t")) == NULL)  return;  else { // file이 정상적으로 열렸을 경우  for (int i = 0; i < object\_index; i++) {  fwrite(object\_code[i], strlen(object\_code[i]), 1, file);  fwrite("\n", strlen("\n"), 1, file);  // file에 한 줄씩 적는다.  }  fclose(file);  }  } |

1. **search\_n\_byte**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 명령어의 형식을 찾아 몇 바이트가 소요되는지 반환하는 함수이다  \* 매계 : operator name  \* 반환 : 몇 byte를 쓰는지 return, opcode가 아닐 경우 0 반환  \* -----------------------------------------------------------------------------------  \*/  int search\_n\_byte(char \*str) {  if (str[0] == '+') { // 4형식인 경우  return 4; //4byte를 return  }    //아닌 경우 opcode를 찾아 format(byte수)를 리턴  int index = search\_opcode(str);  if (index < 0)  return 0; //directives일 경우 0을 반환  else  return atoi(inst\_table[index]->format);  } |

1. **search\_literal\_table**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : literal이 LITTAB에 존재하는지 검사해주는 함수  \* 매계 : literal string  \* 반환 : literal이 존재하면 index를 반환, 없을 경우 -1 반환  \* -----------------------------------------------------------------------------------  \*/  int search\_literal\_table(char \*str) {  for (int i = 0; i < literal\_index; i++) {  if (strcmp(literal\_table[i].literal, str) == 0) {  return i;  }  }  return -1; //해당 literal이 존재하지 않는 경우  } |

1. **init\_register\_number**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : register정보를 담고있는 register\_table을 초기화한다.  \* 매계 : 없음  \* 반환 : 없음  \* -----------------------------------------------------------------------------------  \*/  void init\_register\_number() {  register\_table[regi\_index].name = "A";  register\_table[regi\_index].number = 0;  regi\_index++;  register\_table[regi\_index].name = "X";  register\_table[regi\_index].number = 1;  regi\_index++;  register\_table[regi\_index].name = "L";  register\_table[regi\_index].number = 2;  regi\_index++;  register\_table[regi\_index].name = "B";  register\_table[regi\_index].number = 3;  regi\_index++;  register\_table[regi\_index].name = "S";  register\_table[regi\_index].number = 4;  regi\_index++;  register\_table[regi\_index].name = "T";  register\_table[regi\_index].number = 5;  regi\_index++;  register\_table[regi\_index].name = "F";  register\_table[regi\_index].number = 6;  regi\_index++;  register\_table[regi\_index].name = "PC";  register\_table[regi\_index].number = 8;  regi\_index++;  register\_table[regi\_index].name = "SW";  register\_table[regi\_index].number = 9;  regi\_index++;  } |

1. **search\_register**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : register에 해당하는 숫자를 반환하는 함수  \* 매계 : register(X,A.. 등)  \* 반환 : 찾으면 register에 해당하는 숫자를 반환한다. 없는 register이면 -1을 반환한다  \* 주의 : NULL이면 0을 반환한다  \* -----------------------------------------------------------------------------------  \*/  int search\_register(char\* x) {  if (x == NULL)  return 0;  for (int i = 0; i < regi\_index; i++) {  if (strcmp(register\_table[i].name,x)==0) {  return register\_table[i].number;  }  }  return -1;  } |

1. **add\_literal\_addr**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : literal이 table의 원소들 중 주소 할당이 안된 리터럴에 주소를 할당  또한 literal의 길이를 검사해서 locctr을 자동으로 증가시킨다  \* 매계 : 없음  \* 반환 : 없음  \* -----------------------------------------------------------------------------------  \*/  void add\_literal\_addr() {  for (int i = 0; i < literal\_index; i++) {  if (literal\_table[i].addr == 0) {  //주소 할당이 안된 literal일 경우 주소를 할당한다  literal\_table[i].addr = locctr;  if (literal\_table[i].format == 'C') { //'EOF'->strlen만큼 추가  locctr += strlen(literal\_table[i].literal);  }  else { //HEX(X)인 경우  // 'F1' -> strlen/2만큼 추가  locctr += strlen(literal\_table[i].literal) / 2;  }  }  }  } |

1. **symbol\_address**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : symbol 이름을 주면 해당하는 control section에서의 address를 반환하는 함수  \* control section이 다르면 symbol이 겹쳐도 SYMTAB에 들어가기 때문에  \* 그것을 구분해줄 수 있는 변수가 필요하다.  \* 매계 : symbol이름, symbol을 찾을 control section  \* 반환 : control section에서의 address, 해당 symbol이 없을 경우 -1 반환  \* -----------------------------------------------------------------------------------  \*/  int symbol\_address(char \*str, int cs) {  int section=-2;  //addr가 0이 나올때마다 section은 하나씩 증가함.  //초반에 program 이름(COPY) 말하고, FIRST가 있기 때문에 처음 section을 -2로 잡았다.  //symbol\_unit에 section을 추가하는 것이 나았을 것 같다.  for (int i = 0; i < sym\_index; i++) {  if (sym\_table[i].addr == 0) {  section++;  }  if (strcmp(str, sym\_table[i].symbol) == 0 && section==cs) {  return sym\_table[i].addr;  //현재 section에서의 symbol을 찾을 경우 주소 반환  }  }  return -1;  } |

symbol\_unit구조체에 section 변수를 넣지 않아 아쉬움이 있었다. 처음에 int section=-2로 시작하는 이유는 COPY START, FISRT에서 label의 address가 모두 0이었기 때문이다. 그러나 이것은 너무 COPY program에 국한된 것이고, subprogram인 WRRED, RDREC에서도 충분이 두 번 address가 0인 label이 나올 수 있는 것이다. 이 점을 고려해서 구조체에 변수를 추가하는 것이 더 확장성 있는 프로그램을 구현하는 방법이었다고 생각한다.

1. **add\_text\_record**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : instruction code를 text record에 추가하는 함수  \* 추가하려는 레코드가 MAX\_TEXT를 넘을 경우, 이전 텍스트는 object code에 적고  \* 새 레코드를 만든다. 만약 넘지 않을 경우, 기존 레코드에 추가하려는 레코드를 덧붙이는  \* 작업만 한다. 또한 아직 record 작성을 시작하지 않았을 경우, 새 레코드를 만들어준다.  \* locctr 또한 알아서 더한다.  \* 매계 : buf(최종 text record 한 줄이 될 buffer)  \* record(현재 record의 instruction code들이 들어있는 buffer),  \* object(추가할 instruction code가 들어있는 buffer),  \* tot\_length\_ptr(현재 text record의 길이를 가진 변수의 pointer),  \* byte(추가하려는 instruction code의 byte)  \* 반환 : 성공시 1을 반환한다.  \* -----------------------------------------------------------------------------------  \*/  int add\_text\_record(char \*buf, char \*record, char \*object, int \*tot\_length\_ptr, int byte) {  if (\*tot\_length\_ptr==0) {  //이번에 새로운 text record를 적기 시작하는 경우  object\_code[object\_index][0] = 'T';  sprintf(buf, "%06X", locctr);  strcat(object\_code[object\_index], buf); //시작주소(현재 locctr)을 추가한다  strcpy(record, object); //record에 현재 instruction code를 넣어준다  //strcat하면 쓰레기값 뒤에 추가되므로 먼저 초기화 해야함!  \*tot\_length\_ptr = byte; //text\_length를 변경해준다  locctr += byte; //locctr 또한 증가시킨다  return 1;  }  if ((\*tot\_length\_ptr + byte) > MAX\_TEXT) {  //만약 현재 코드를 추가하면 text record 최대 허용 길이를 넘기는 경우  fin\_text\_record(buf, record, tot\_length\_ptr);  //지금까지의 text record는 object code에 적는다  object\_code[object\_index][0] = 'T'; //새로운 record를 만든다  sprintf(buf, "%06X", locctr);  strcat(object\_code[object\_index], buf);  \*tot\_length\_ptr = byte; //text\_length를 변경해준다  strcpy(record, object);  //record에 추가할 record를 새로 쓴다(record 새로 시작)  locctr += byte;  }  else {  //추가해도 이상 없는 경우  strcat(record, object); //현재 record에 추가할 레코드를 추가한다.  \*tot\_length\_ptr = \*tot\_length\_ptr + byte;  //text\_length도 추가할 record의 byte만큼 증가한다  locctr += byte; //locctr도 증가시킨다.  }    return 1;  } |

1. **fin\_text\_record**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : 지금까지 적은 record를 object\_code에 기록하고 끝내는 함수  \* 만약 현재 record가 시작되지 않은 상태라면 그냥 돌아간다.  \* 매계 : buf(최종 text record 한 줄이 될 buffer),  \* record(현재 record의 instruction code들이 들어있는 buffer),  \* tot\_length\_ptr(현재 text record의 길이를 가진 변수의 pointer)  \* -----------------------------------------------------------------------------------  \*/  void fin\_text\_record(char \*buf, char \*record, int \*tot\_length\_ptr) {  if (\*tot\_length\_ptr == 0) {  //record가 시작되지 않은 경우 끝낼 것이 없으므로 돌아간다  return;  }  //끝낼 record가 있는 경우  sprintf(buf, "%02X%s", \*tot\_length\_ptr,record);  //buffer에 length와 최종 record들을 추가한다  strcat(object\_code[object\_index], buf); //length + records를 object\_code에 적는다  //시작주소는 이미 들어가있었다  object\_index++; //index를 증가시킨다  \*tot\_length\_ptr = 0; //text length를 reset한다  record[0] = '\0'; //record도 reset시킨다  } |

1. **add\_modify\_unit**

|  |
| --- |
| /\* ----------------------------------------------------------------------------------  \* 설명 : modify table에 modify unit을 하나 추가하는 함수  \* 매계 : addr(수정할 주소), length(수정할 길이), plus(해당 주소를 더하는지 빼는지 기호),  \* name(더하거나 뺄 symbol의 이름), section(수정하는 record가 어느 section인지)  \* 반환 : 없음  \* -----------------------------------------------------------------------------------  \*/  void add\_modify\_unit(int addr, int length, char plus, char \* name,int section) {  modify\_table[m\_index].addr=addr;  modify\_table[m\_index].length=length;  modify\_table[m\_index].plus=plus;  strcpy(modify\_table[m\_index].name,name);  modify\_table[m\_index].section = section;  m\_index++;  } |

1. **기대효과 및 결론**

프로그램을 짜면서, 두루뭉실했던 SIC/XE 머신의 동작을 잘 알게 되었다. 어떻게 소스 코드를 읽고, token 단위로 parsing하고, 기계어 코드를 만드는지 읽는 과정부터 출력하는 과정까지 구현해볼 수 있는 좋은 기회였다. 머릿속으로 어렴풋이 짐작만하고 있었던 동작 과정이 생각보다 더 복잡하단 것을 알았다.

아쉬웠던 점은, Input text로 주어진 COPY 프로그램만 보고 설계를 하고 구현을 하다 보니 COPY 프로그램에만 잘 돌아가는 프로그램을 짰다는 것이다. 모듈 구현에서 아쉬운 점을 나열했듯이, ‘생각해보니 이 부분도 구현해야 했구나’ 하는 부분들이 많았다. 만약 COPY가 아닌 다른 프로그램을 넣으면 에러가 날 곳들이 보인다. 분명 내가 미처 생각하지 못한 점도 있을 것이다.

다음에는 아쉬웠던 점을 기반으로 어느 코드에도 잘 돌아갈 수 있는 프로그램을 만들 수 있도록 보완할 것이다. 그렇게 보완함으로써 더욱 SIC/XE 머신의 처리 동작의 이해를 높일 수 있을 것이다.