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#1B SIC/XE Machine Assembler 구현** | | **담당교수** | **최 재 영 교 수** | | **제출인** | ***전자정보공학부 20160458 김지우***  **컴퓨터 학부수업 (출석번호 109번)** | | **제출일** | **2020년 5월 13일** | |

**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)**  29

4.1. 전체 시스템 구현 내용

4.2. 모듈별 구현 내용

1. **기대효과 및 결론**  62
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 자바 파일을 이용하여 SIC/XE 소스를 object program code로 변환하는 과정을 복습하고 C와 비교한다.

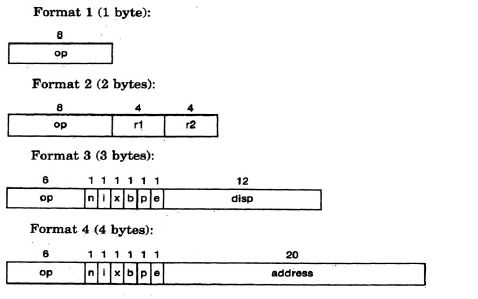
특히, 이번 프로젝트는 Control Section 내용이 추가되었다. 내용이 추가되어 동작이 복잡해진만큼 구현 또한 복잡할 것이다. 하지만 그만큼 더 고심해서 프로그램을 설계하고 구현하고 보면, 프로그램의 동작에 대한 이해를 확실히 할 수 있을 것이다. 또한 이번 프로젝트 1b는 C언어로 만든 것을 JAVA로 구현한다. 객체지향 언어인 자바를 사용함으로써 모듈별 설계의 감을 익히고, 두 언어간의 차이를 학습할 수 있다.

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**

java.io.File

파일을 열기 위해 사용되는 기본이다.

boolean exists()를 이용하여 파일이 존재하는지 여부를 묻고, 없다면 boolean createNewFile()를 이용하여 파일을 생성할 수 있다. 또한 append모드인지, 새로 쓰는 모드인지도 설정할 수 있다.

java.io.FileReader

java.io.BufferedReader

파일을 한 line씩 읽는 데 사용된다. 생성 시 인자로 Reader가 필요하기 때문에 FileReader를 인자로 넣어 주어야 한다. String readLine()을 이용해 읽는다.

java.io.FileWriter

void write(String str)를 이용하여 원하는 문장을 파일에 쓸 수 있다.

모든 File들은 close()를 이용하여 닫아 주어야 한다.

* **Token Parsing**

java.util.StringTokenizer

Token으로 자르는데 필요한 클래스로, 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. C와 달리 인자로 들어가는 문자열은 잘리지 않았다. StringTokenizer(String str, String delim)을 이용해 자를 문자열과 구분자를 지정한다. null error가 나지 않게 boolean hasMoreTokens()를 이용하여 확인하고, String nextToken()로 토큰을 불러온다.

* **Format에 맞추어 출력**

java.lang.String

static String format(String format, Object... args)

String class의 str에 데이터 형식에 맞추어 데이터를 쓸 수 있다. 원하는 format과 인자를 적으면 원하는 format으로 적힌 String을 반환한다. record 한 줄을 적을 때 포맷에 맞춰야 하기 때문에 주로 활용한다.

* **클래스 활용**

JAVA는 객체지향언어이기 때문에 Class를 사용한다. 모듈로 분리하는 데에 더 용이할 수 있다.

* **JAVA는 pointer를 개발자가 직접적으로 사용할 수 없다**
* **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.

* **Exception**

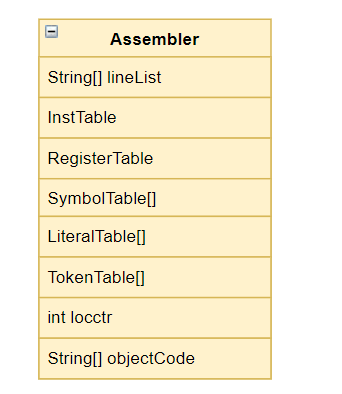
이번 프로젝트는 에러 처리를 Exception try-catch문을 이용하여 처리하였다.

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

이 프로젝트는 **소스 코드를 Objet Program으로 변환하는 Assembler** **프로그램을 JAVA로 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.

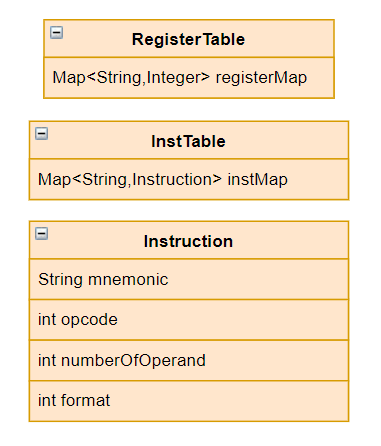
전체 시스템을 설계하기 위해 필요한 클래스부터 설계하였다.



[그림 3-1] 설계한 Class-1

1. **Assembler**

가장 주요 역할을 하는 Assembler Class이다. 전체 프로그램 코드를 한 줄씩 저장하는 lineList가 필요하다. 또, 명령어 set들을 저장하는 InstTable과 Register 정보를 가지는 Register Table을 멤버로 가진다. section별로 SymbolTable과 LiteralTable, TokenTable을 가지는데, section이 여러 개면 이것들을 List로 관리해야 할 것이다. 프로그램을 수행하는 데에 있어 locctr가 필요할 것이고 최종 object code들을 저장하는 objectCode를 멤버로 설계하였다.



[그림 3-2] 설계한 Class-2

1. **RegisterTable**

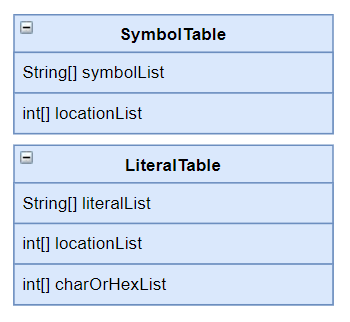
SIC/XE machine에는 여러 개의 register가 있고, 그 register에 고유의 번호가 부여된다. 해당 Register들을 한 곳에 보관하고 관리하는 Class를 설계하였다. Register들의 이름과 고유 번호를 매핑시키는 registerMap을 멤버로 가진다.

1. **InstTable**

처음 프로그램이 시작할 때, 명령어 set들을 모아놓은 파일을 읽어온다. 그 명령어들을 한 곳에 보관하고 관리하는 Class를 설계하였다. 문자 명령어와 그 명령어의 정보인 Instruction Instance를 매핑시키는 instMap을 멤버로 가진다.

1. **Instruction**

명령어를 저장하는 하나의 유닛이다. 문자 명령어, 명령어의 opcode, operand의 개수 그리고 명령어의 형식(형식이 곧 소요되는 byte와 같다)이 저장된다.



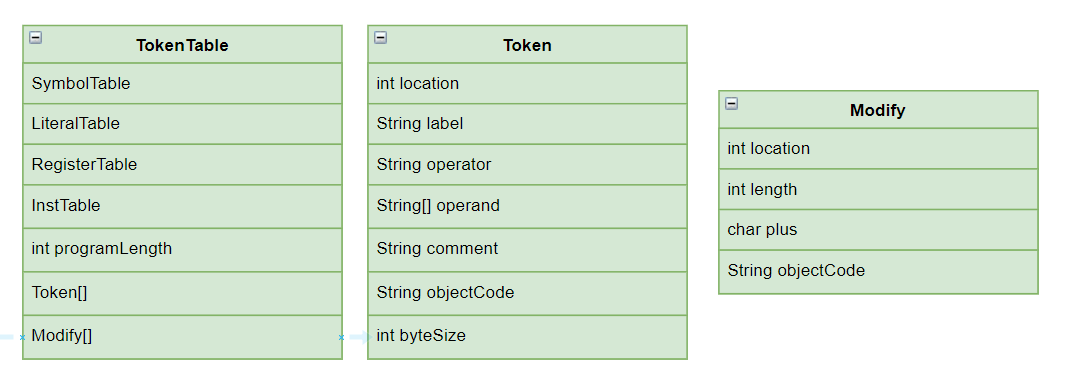
[그림 3-3] 설계한 Class-3

1. **SymbolTable**

SYMTAB의 역할을 하는 Class를 설계하였다. symbol들을 저장하는 symbolList와 symbol들의 주소를 저장하는 locationList를 멤버로 가진다. 이 Table은 section당 하나가 생성된다.

1. **LiteralTable**

LITTAB 역할을 하는 Class를 설계하였다. literal들을 저장하는 literalList와 literal들의 주소를 저장하는 locationList를 멤버로 가진다. 또한 추후에 char인지 hex인지 알기 위해 charOrHexList를 갖도록 설계하였다. 이 Table 또한 section당 하나가 생성된다.



[그림 3-4] 설계한 Class-4

1. **TokenTable**

parsing한 Token을 여러 개 저장하고 관리하는 Class를 설계하였다. Token Instance들을 여러 개 저장하고, 해당 섹션의 programLength를 저장한다. 또한 object code를 만들 때도 Token을 보며 만들어야 하므로, TokenTable에서 object code를 만들게 될 것인데, 그러기 위해서는 해당 섹션의 SymbolTable과 LiteralTable이 필요하고, Assembler로부터 InstTable과 RegisterTable을 링크해야 한다. 또한 object code를 만들다 보면 수정해야할 부분이 나온다. 그 정보를 저장하기 위해 Modify Instance를 여러 개 가진다.

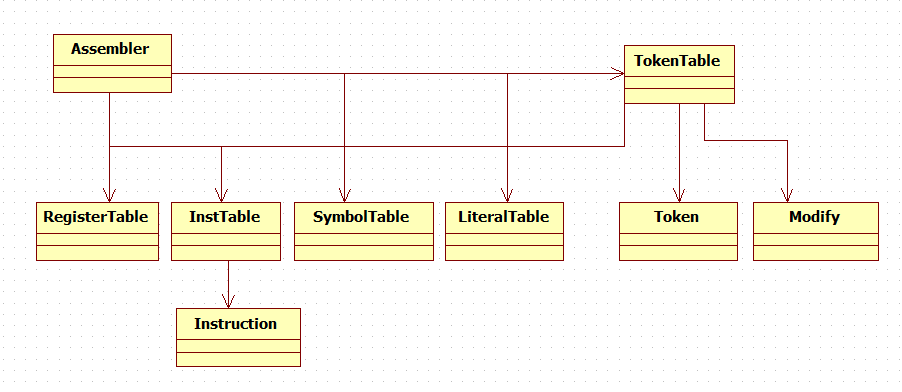
1. **Token**

Token별로 parsing한 정보를 저장하는 하나의 유닛이다. label, operator, operand, comment로 분리하여 저장하고, 해당 명령어의 byteSize를 저장한다. 또한 추후에 object code를 생성할 때 저장공간인 objectCode를 멤버로 가진다.

1. **Modify**

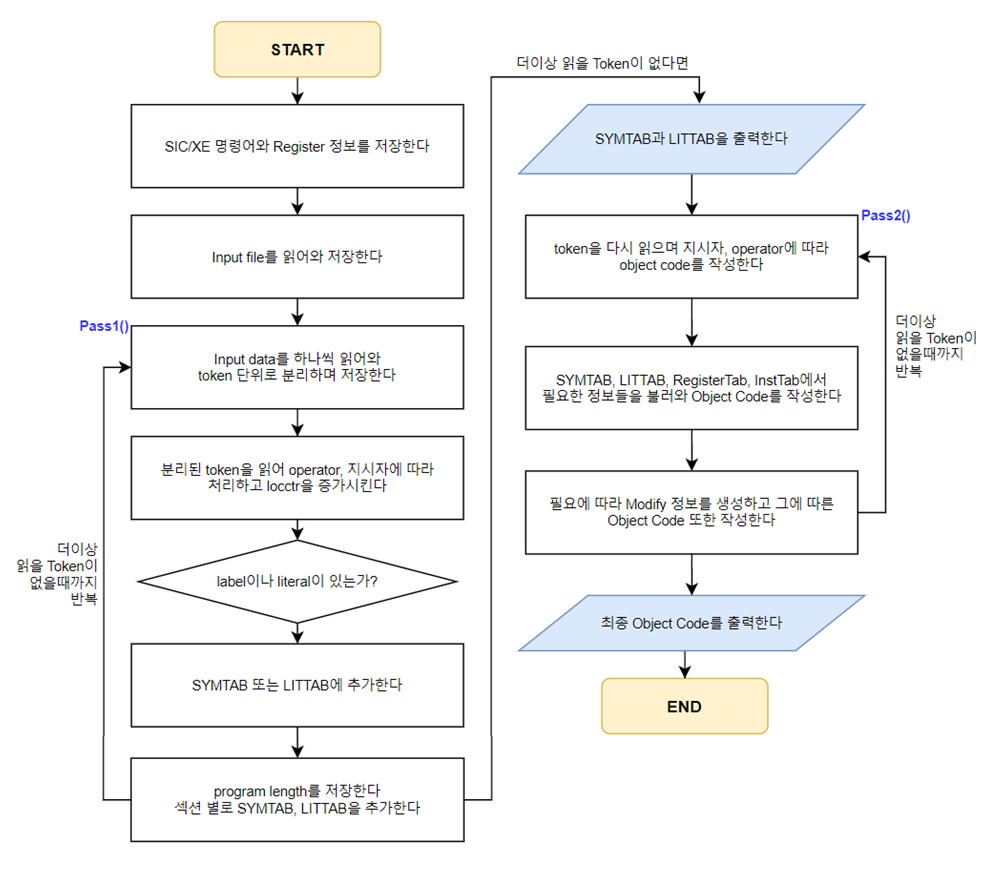
수정 레코드를 만들기 위해 저장되는 Modify 유닛이다. 수정 위치를 가리키는 location, 수정 길이를 가리키는 length, 더할지 뺄지를 결정하는 plus, 그리고 이 정보를 바탕으로 만들어진 Modification Record를 저장하는 objectCode를 멤버로 갖는다.

지금까지 설계한 Class들을 바탕으로 관계도를 그려보면 다음과 같다.



[그림 3-5] 설계한 Class들의 관계도

Assembler는 RegisterTable, InstTable과 여러 개의 섹션 별 SymbolTable, 여러 개의 LiteralTable, 여러 개의 TokenTable을 멤버로 가진다. TokenTable 또한 RegisterTable, InstTable을 멤버로 가지고, 같은 섹션의 SymbolTable, LiteralTable을 멤버로 가진다. 또, 해당 section에서 parsing한 Token들을 여러 개 갖고 있다. TokenTable은 해당 섹션 별 Modification Record를 갖기 위해 수정 레코드 유닛인 Modify들을 여러 개 갖고 있다. InstTable은 명령어 set을 저장하는데, 명령어를 간단하게 저장하기 위해 Instrcution이라는 명령어를 저장하는 유닛 클래스를 멤버로 가진다.



[그림 3-6] 전체적인 시스템 흐름도

다음은 간단히 나타낸 전체적인 시스템 설계 흐름도이다. 해당 흐름도에 따라 필요한 모듈의 기능들을 작성해 보았다.

**(1) RegisterTable**

(1-1) 초기에 register-number 정보를 저장하는 모듈

(1-2) register의 해당 number를 반환하는 모듈

**(2) Instruction**

(2-1) 명령어 set 하나를 parsing해 해당 멤버 변수들에 저장하는 모듈

**(3) InstTable**

(3-1) instMap에 명령어 정보들을 저장하는 모듈

(3-2) 명령어가 소요하는 byte 수를 반환하는 모듈

(3-3) 명령어의 operand 개수를 반환하는 모듈

(3-4) 명령어의 opcode를 반환하는 모듈

**(4)SymbolTable**

(4-1) 처음 만난 symbol을 저장하는 모듈

(4-2) symbol에게 주소를 할당하는 모듈

(4-3) symbol을 찾아 주소를 반환하는 모듈

**(5)LiteralTable**

(5-1) 처음 만난 literal을 저장하는 모듈

(5-2) literal에게 주소를 할당하는 모듈

(5-3) literal을 찾아 주소를 반환하는 모듈

(5-4) 해당 섹션의 모든 literal들에게 주소를 부여하는 모듈

**(6)Modify**

(6-1) Modification record를 생성하는 모듈

**(7)Token**

(7-1) token 단위로 parsing하여 해당 멤버에 값을 저장하는 모듈

(7-2) token의 nixbpe 중 원하는 flag를 세팅하는 모듈

(7-3) token의 nixbpe 중 원하는 flag를 반환하는 모듈

**(8)TokenTable**

(8-1) Token List에 Token을 생성해 저장하는 모듈

(8-2) Token List에서 원하는 Token을 반환하는 모듈

(8-3) 원하는 token의 object code를 만들어 저장하는 모듈

(8-4) 원하는 token의 object code를 반환하는 모듈

**(9)Assembler**

(9-1) Input file을 읽어 저장하는 모듈

(9-2) pass1 모듈(해당 Input file을 parsing하고, 섹션 별 symbol, literal table을 만든다)

(9-3) symbol tab 출력 모듈

(9-4) literal table 출력 모듈

(9-5) pass2 모듈(token들과 symbol, literal을 기준으로 object code를 만든다)

(9-6) Text record에 하나의 object code를 추가하는 모듈

(9-7) 해당 Text record를 적고 끝내는 모듈

(9-8) 해당 section의 모든 modification object code를 추가하는 모듈

(9-9) 해당 section의 모든 Literal을 object code로 변환해 추가하는 모듈

(9-10) 최종 변환된 object code들을 출력하는 모듈

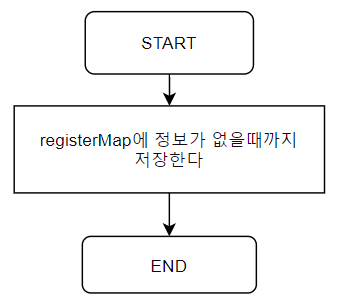
다음과 같은 함수들이 필요할 것이라 생각되었다. 세부적인 설계 흐름도는 아래에서 다룰 것이다.

* 1. **모듈별 설계 내용**

너무 자잘한 에러 처리는 보지 않고, 전체적인 흐름을 보기 위한 flow chart이다. (참고로 출력 모듈은 출력만 하는 모듈이기 때문에 flow chart를 따로 그리지 않았다.)

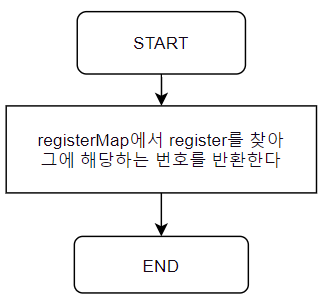
**(1) RegisterTable**

(1-1) 초기에 register-number 정보를 저장하는 모듈



[그림 3-7] 해당 모듈의 설계 흐름도

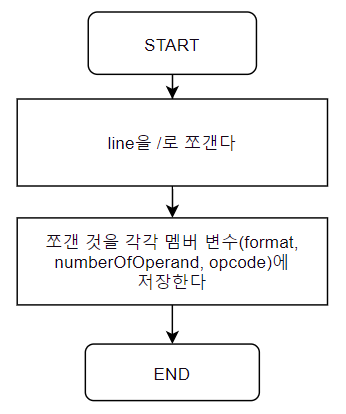
(1-2) register의 해당 number를 반환하는 모듈



[그림 3-8] 해당 모듈의 설계 흐름도

**(2) Instruction**

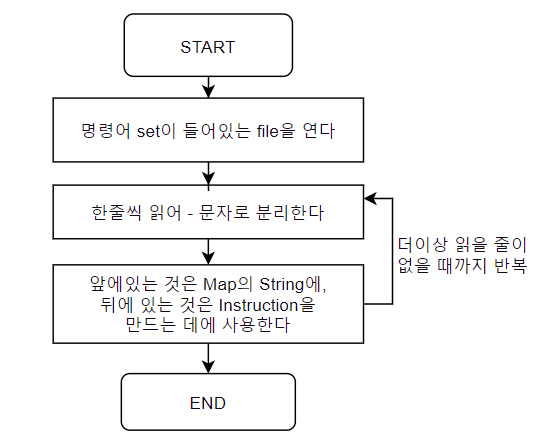
(2-1) 명령어 set 하나를 parsing해 해당 멤버 변수들에 저장하는 모듈



[그림 3-9] 해당 모듈의 설계 흐름도

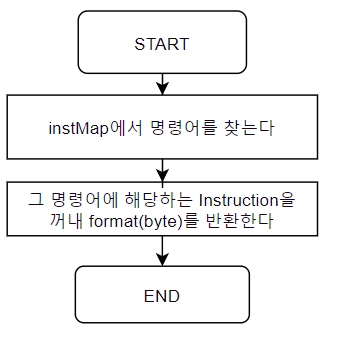
**(3) InstTable**

(3-1) instMap에 명령어 정보들을 저장하는 모듈



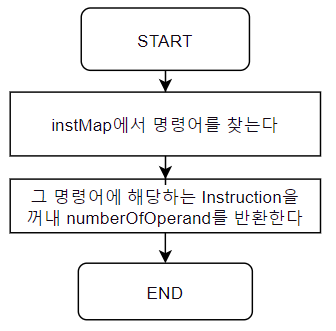
[그림 3-10] 해당 모듈의 설계 흐름도

(3-2) 명령어가 소요하는 byte 수를 반환하는 모듈



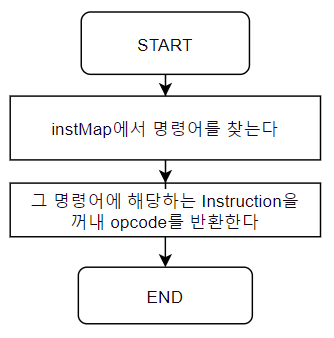
[그림 3-11] 해당 모듈의 설계 흐름도

(3-3) 명령어의 operand 개수를 반환하는 모듈



[그림 3-12] 해당 모듈의 설계 흐름도

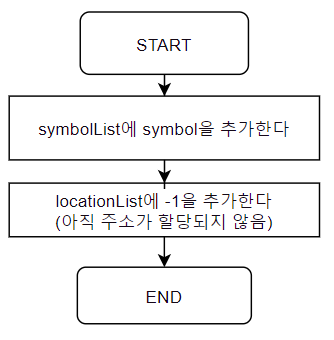
(3-4) 명령어의 opcode를 반환하는 모듈



[그림 3-13] 해당 모듈의 설계 흐름도

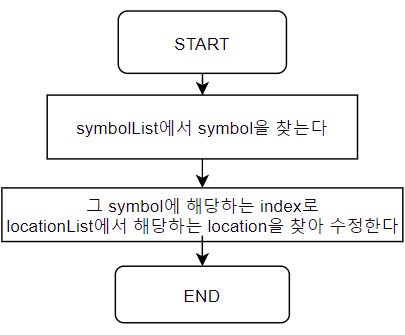
**(4)SymbolTable**

(4-1) 처음 만난 symbol을 저장하는 모듈



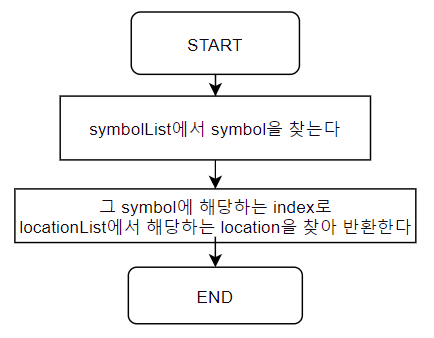
[그림 3-14] 해당 모듈의 설계 흐름도

(4-2) symbol에게 주소를 할당하는 모듈



[그림 3-15] 해당 모듈의 설계 흐름도

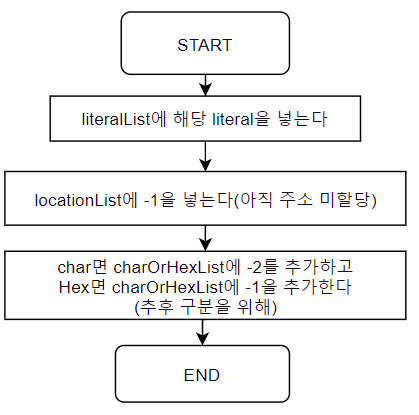
(4-3) symbol을 찾아 주소를 반환하는 모듈



[그림 3-16] 해당 모듈의 설계 흐름도

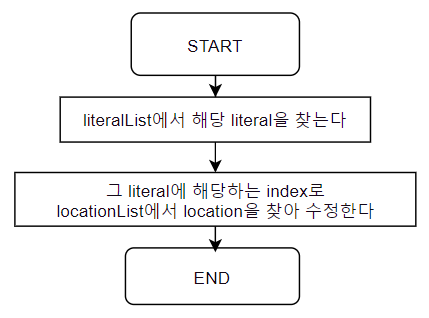
**(5)LiteralTable**

(5-1) 처음 만난 literal을 저장하는 모듈



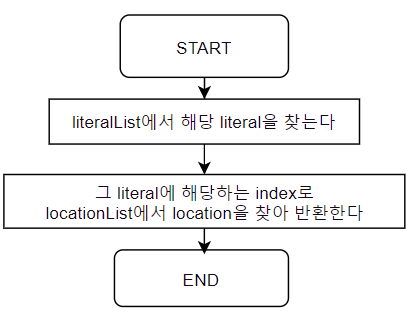
[그림 3-17] 해당 모듈의 설계 흐름도

(5-2) literal에게 주소를 할당하는 모듈



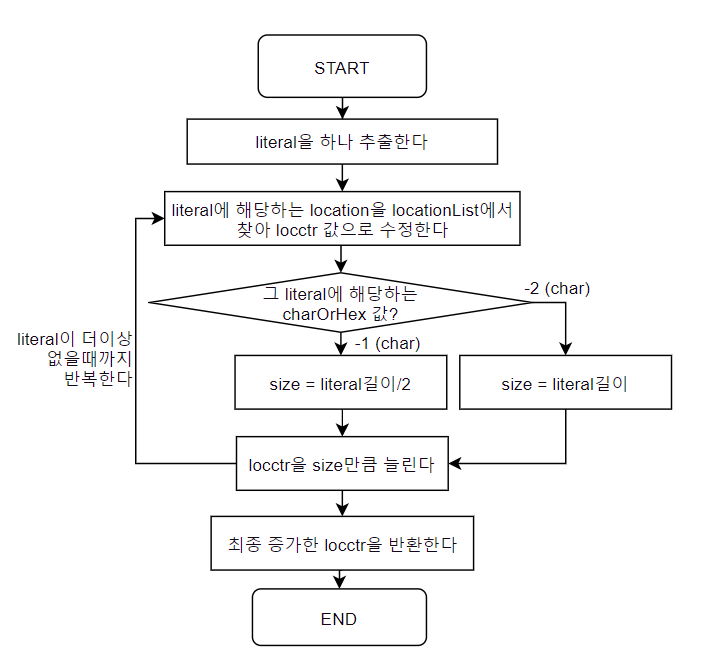
[그림 3-18] 해당 모듈의 설계 흐름도

(5-3) literal을 찾아 주소를 반환하는 모듈



[그림 3-19] 해당 모듈의 설계 흐름도

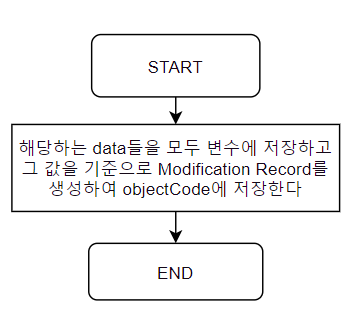
(5-4) 해당 섹션의 모든 literal들에게 주소를 부여하는 모듈



[그림 3-20] 해당 모듈의 설계 흐름도

**(6)Modify**

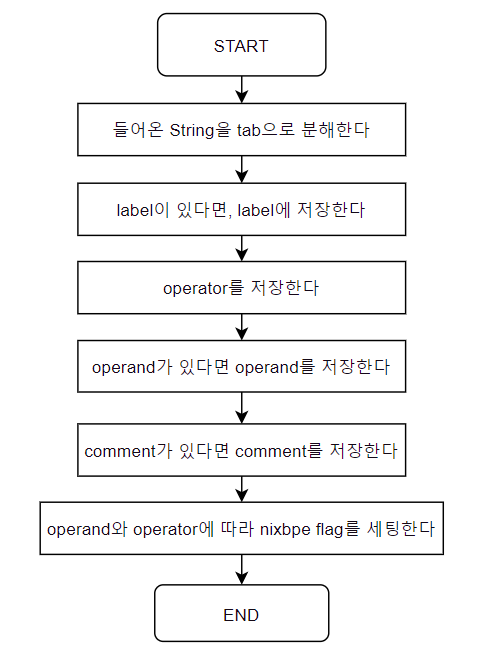
(6-1) Modification record를 생성하는 모듈



[그림 3-21] 해당 모듈의 설계 흐름도

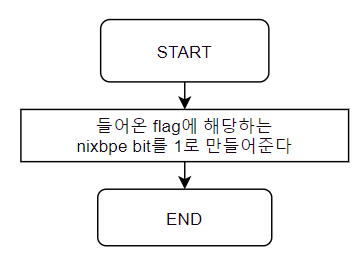
**(7)Token**

(7-1) token 단위로 parsing하여 해당 멤버에 값을 저장하는 모듈



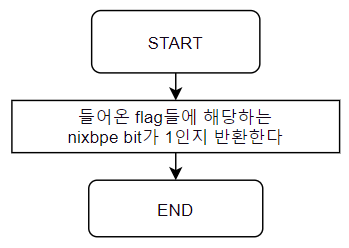
[그림 3-22] 해당 모듈의 설계 흐름도

(7-2) token의 nixbpe 중 원하는 flag를 세팅하는 모듈



[그림 3-23] 해당 모듈의 설계 흐름도

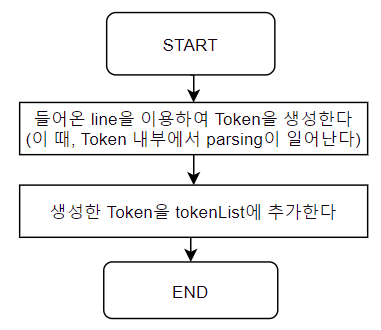
(7-3) token의 nixbpe 중 원하는 flag를 반환하는 모듈



[그림 3-24] 해당 모듈의 설계 흐름도

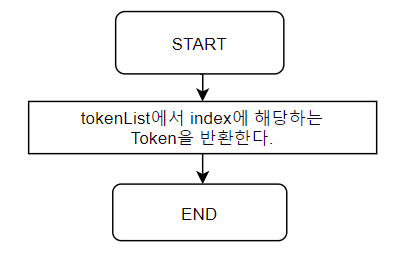
**(8)TokenTable**

(8-1) Token List에 Token을 생성해 저장하는 모듈



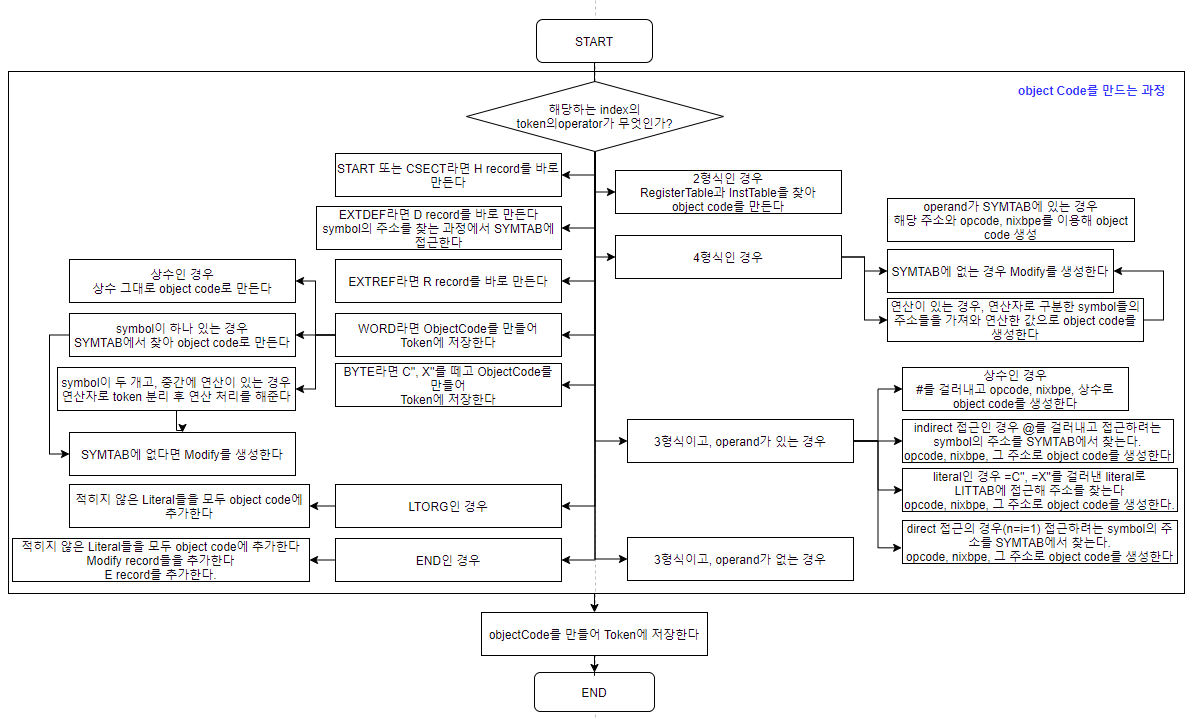
[그림 3-25] 해당 모듈의 설계 흐름도

(8-2) Token List에서 원하는 Token을 반환하는 모듈



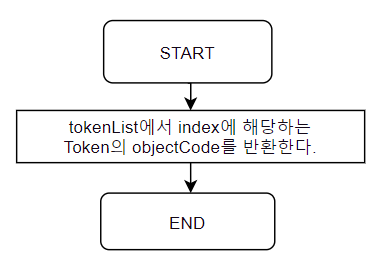
[그림 3-26] 해당 모듈의 설계 흐름도

(8-3) 원하는 token의 object code를 만들어 저장하는 모듈



[그림 3-27] 해당 모듈의 설계 흐름도

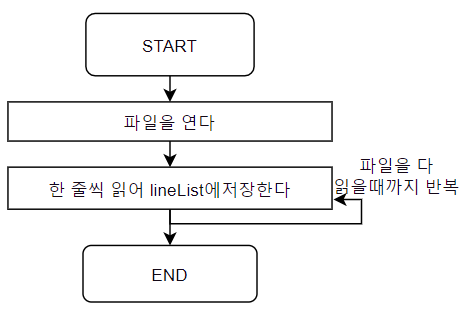
(8-4) 원하는 token의 object code를 반환하는 모듈



[그림 3-28] 해당 모듈의 설계 흐름도

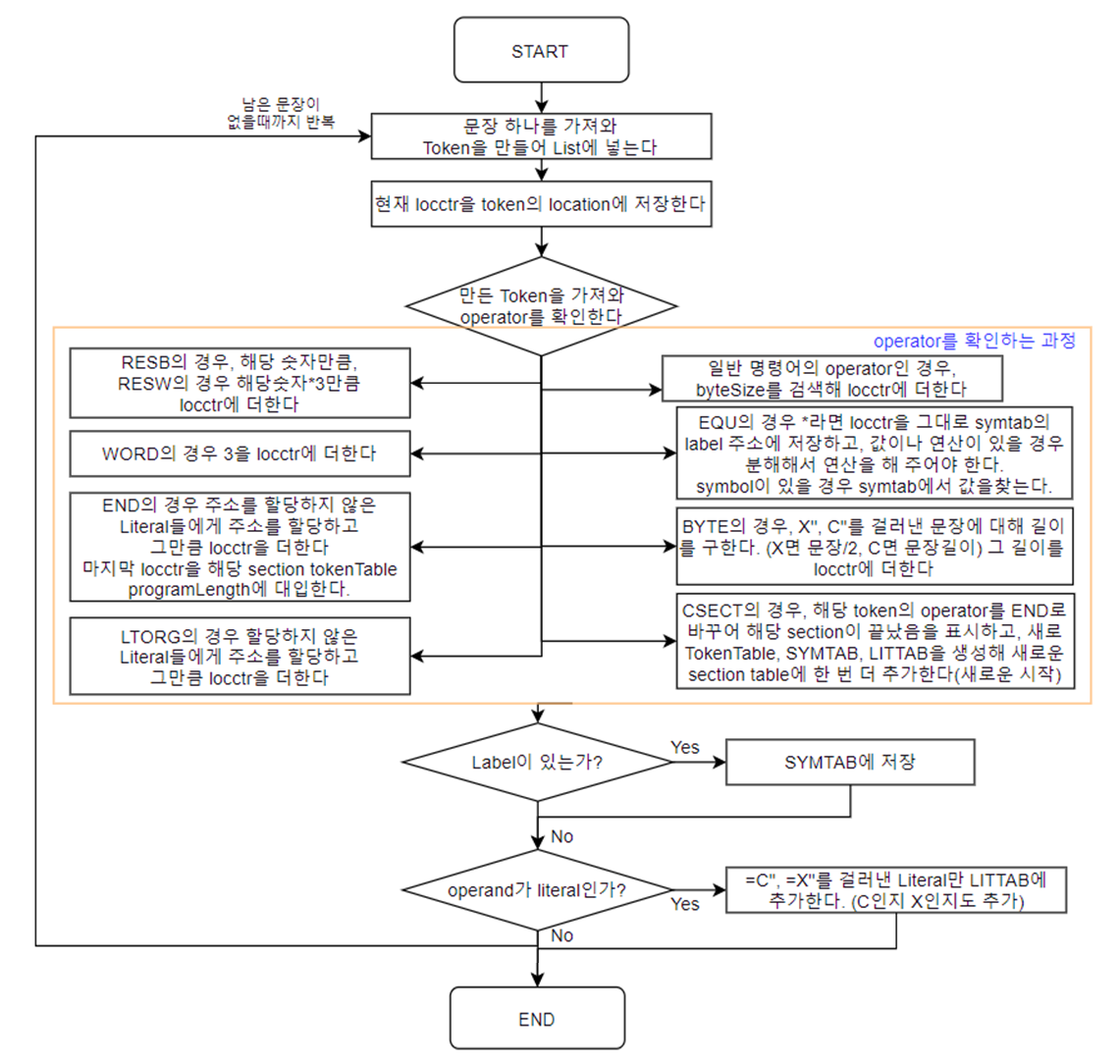
**(9)Assembler**

(9-1) Input file을 읽어 저장하는 모듈



[그림 3-29] 해당 모듈의 설계 흐름도

(9-2) pass1 모듈



[그림 3-30] 해당 모듈의 설계 흐름도

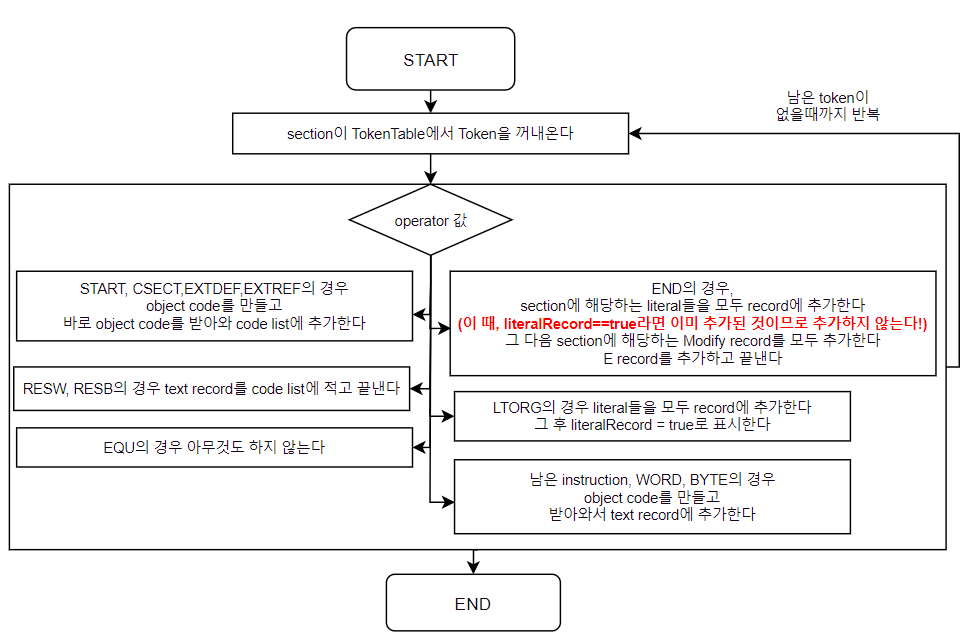
(9-3) symbol tab 출력 모듈

(출력 모듈이므로 생략)

(9-4) literal table 출력 모듈

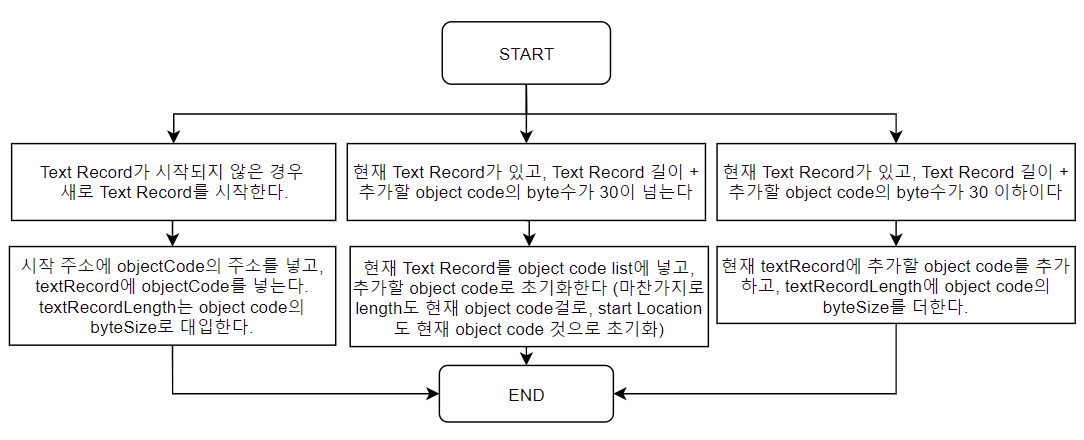
(출력 모듈이므로 생략)

(9-5) pass2 모듈



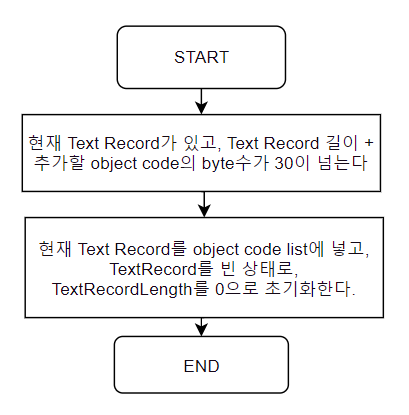
[그림 3-31] 해당 모듈의 설계 흐름도

(9-6) Text record에 하나의 object code를 추가하는 모듈



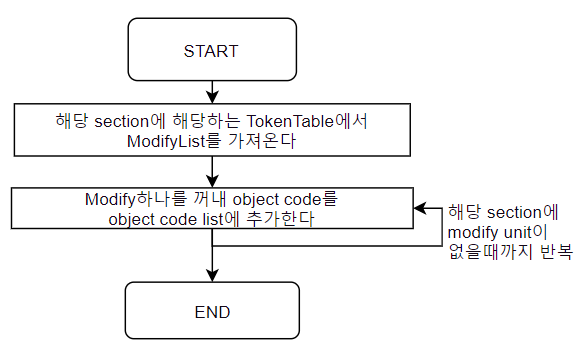
[그림 3-32] 해당 모듈의 설계 흐름도

(9-7) 해당 Text record를 적고 끝내는 모듈



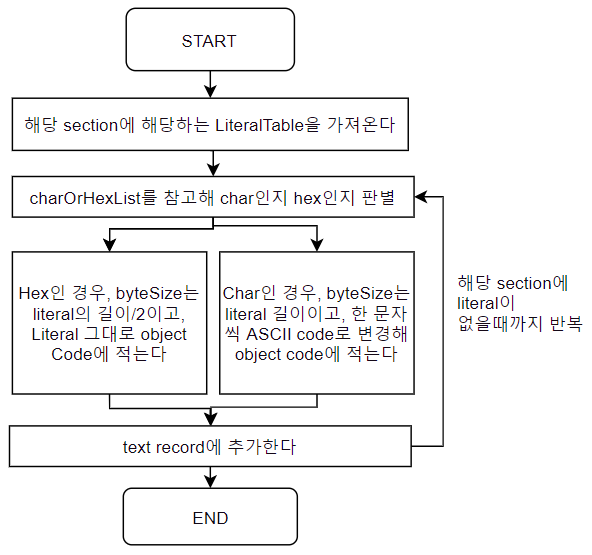
[그림 3-33] 해당 모듈의 설계 흐름도

(9-8) 해당 section의 모든 modification object code를 추가하는 모듈



[그림 3-34] 해당 모듈의 설계 흐름도

(9-9) 해당 section의 모든 Literal을 object code로 변환해 추가하는 모듈



[그림 3-35] 해당 모듈의 설계 흐름도

(9-10) 최종 변환된 object code들을 출력하는 모듈

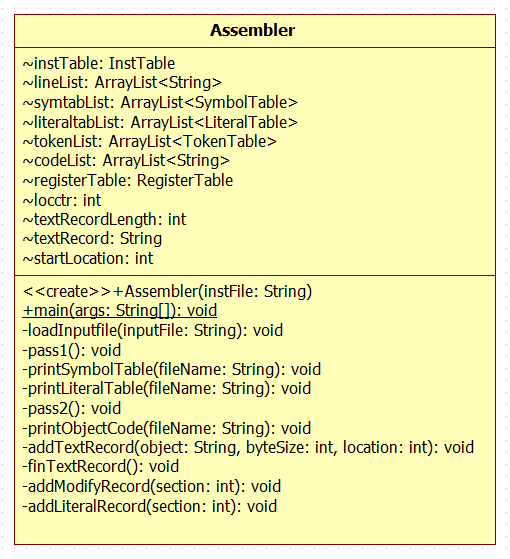
(출력 모듈이므로 생략)

이러한 flow chart들을 기반으로 처음 설계해본 함수는 다음과 같다.

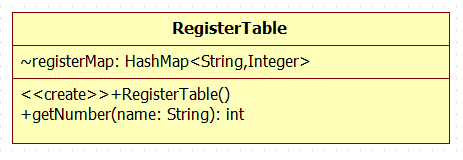
|  |  |
| --- | --- |
| 함수 기능 | 설계한 함수 |
| (1) RegisterTable |  |
| (1-1) 초기에 register-number 정보를 저장하는 모듈 | Register() 생성자로 가능하다 |
| (1-2) register의 해당 number를 반환하는 모듈 | int getNumber(String name) |
| (2) Instruction |  |
| (2-1) 명령어 set 하나를 parsing해 해당 멤버 변수들에 저장하는 모듈 | void parsing(String line) |
| (3) InstTable |  |
| (3-1) instMap에 명령어 정보들을 저장하는 모듈 | void openFile(String fileName) |
| (3-2) 명령어가 소요하는 byte 수를 반환하는 모듈 | int searchByte(String mnemonic) |
| (3-3) 명령어의 operand 개수를 반환하는 모듈 | int searchNumberOfOperand(String mnemonic) |
| (3-4) 명령어의 opcode를 반환하는 모듈 | int searchOpcode(String mnemonic) |
| (4) SymbolTable |  |
| (4-1) 처음 만난 symbol을 저장하는 모듈 | void putSymbol(String symbol, int location) |
| (4-2) symbol에게 주소를 할당하는 모듈 | void modifySymbol(String symbol, int newLocation) |
| (4-3) symbol을 찾아 주소를 반환하는 모듈 | int search(String symbol) |
| (5) LiteralTable |  |
| (5-1) 처음 만난 literal을 저장하는 모듈 | void putLiteral(String literal, int loctaion) |
| (5-2) literal에게 주소를 할당하는 모듈 | void modifyLiteral(String literal, int newLocation) |
| (5-3) literal을 찾아 주소를 반환하는 모듈 | int search(String literal) |
| (5-4) 해당 섹션의 모든 literal들에게 주소를 부여하는 모듈 | int addAllLiteral(int locctr) |
| (6) Modify |  |
| (6-1) Modification record를 생성하는 모듈 | Modify() 생성자로 가능하다 |
| (7) Token |  |
| (7-1) token 단위로 parsing하여 해당 멤버에 값을 저장하는 모듈 | void parsing(String line) |
| (7-2) token의 nixbpe 중 원하는 flag를 세팅하는 모듈 | void setFlag(int flag, int value) |
| (7-3) token의 nixbpe 중 원하는 flag를 반환하는 모듈 | int getFalg(int flags) |
| (8) TokenTable |  |
| (8-1) Token List에 Token을 생성해 저장하는 모듈 | void putToken(String line) |
| (8-2) Token List에서 원하는 Token을 반환하는 모듈 | Token getToken(int index) |
| (8-3) 원하는 token의 object code를 만들어 저장하는 모듈 | void makeObjectCode(int index) |
| (8-4) 원하는 token의 object code를 반환하는 모듈 | String getObjectCode(int index) |
| (9) Assembler |  |
| (9-1) Input file을 읽어 저장하는 모듈 | void loadInputFile(String inputFile) |
| (9-2) pass1 모듈 | void pass1() |
| (9-3) symbol tab 출력 모듈 | void printSymbolTable(String fileName) |
| (9-4) literal table 출력 모듈 | void printLiteralTable(String fileName) |
| (9-5) pass2 모듈 | void pass2() |
| (9-6) Text record에 하나의 object code를 추가하는 모듈 | void addTextRecord(String object, int byteSize, int location) |
| (9-7) 해당 Text record를 적고 끝내는 모듈 | void finTextRecord() |
| (9-8) 해당 section의 모든 modification object code를 추가하는 모듈 | void addModifyRecord(int section) |
| (9-9) 해당 section의 모든 Literal을 object code로 변환해 추가하는 모듈 | void addLiteralRecord(int section) |
| (9-10) 최종 변환된 object code들을 출력하는 모듈 | void printObjectCode(String fileName) |

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

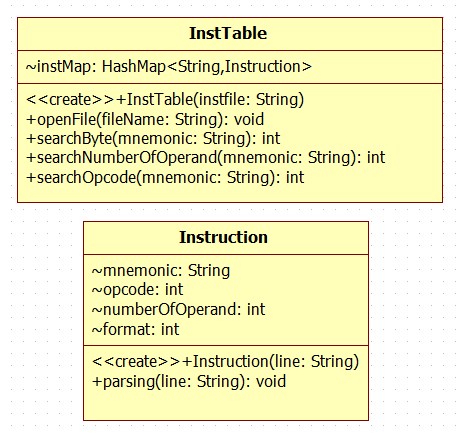
다음은 클래스 다이어그램이다. 클래스는 설계한대로 만들어졌고, 필요에 따라 함수와 static final 변수가 조금 추가되었다. 아래의 다이어그램은 StarUML을 사용하였다.



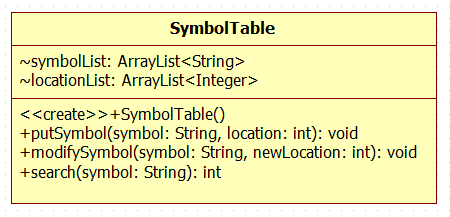
[그림 4-1] Assembler.java



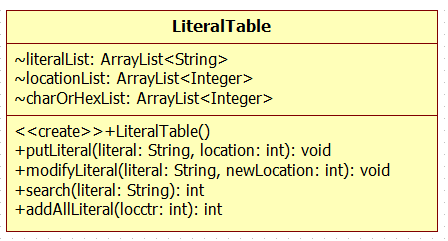
[그림 4-2] RegisterTable.java



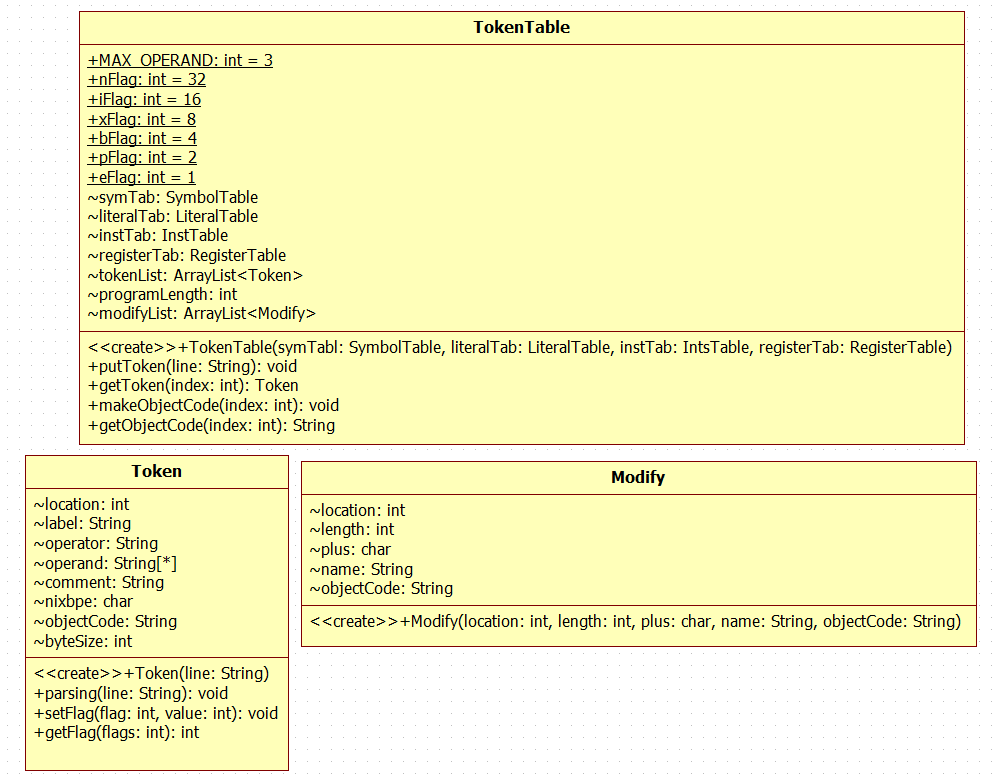
[그림 4-3] InstTable.java



[그림 4-4] SymbolTable.java



[그림 4-5] LiteralTable.java

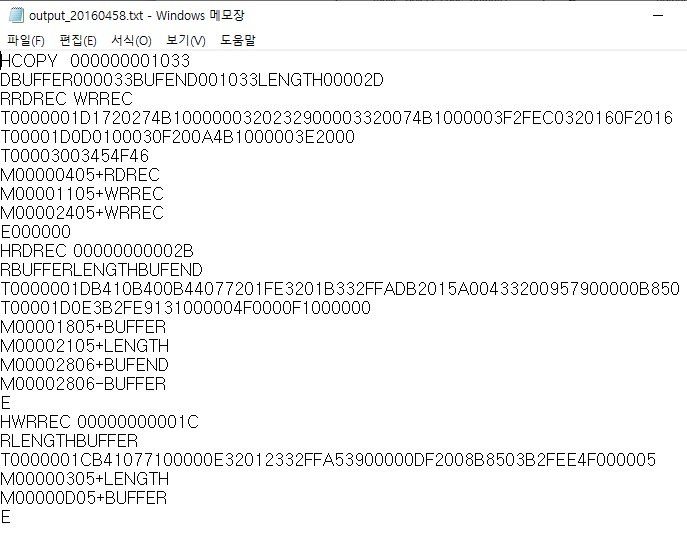


[그림 4-6] TokenTable.java

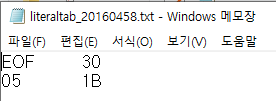
설계에서 조금 추가된 점은 배열들로 설계했던 멤버변수들이 ArrayList<>의 형태를 갖게 되었다. 또한 편의성을 위해 public static final int(flag들, MAX\_OPERAND 등)이 TokenTable, Assembler에 추가되었다. 그리고 원래 객체지향 설계를 위해서는 멤버변수들이 private인 것이 맞지만, 여러 번 접근하는 변수들이 많다 보니 모두 default(package) 범위로 두게 되었다. 사실 이 부분은 보안을 위해서 수정하는 것이 맞다고 생각한다.

Java는 Garbage Collector가 알아서 동적 할당을 해제해 주므로 data\_fre같은 동적 할당 해제 함수는 존재하지 않게 되었다.

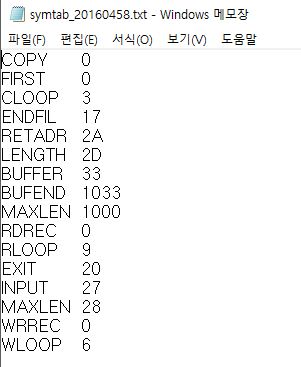
**프로그램 수행 결과**



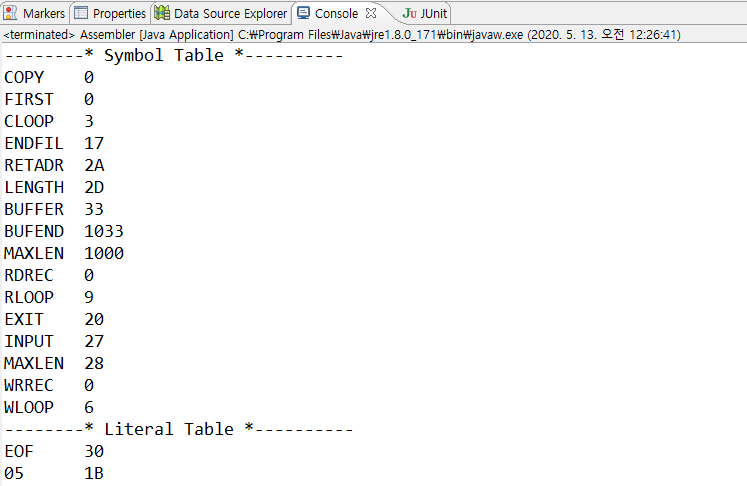
[그림 4-7] File에 출력된 최종 변환된 Object Code



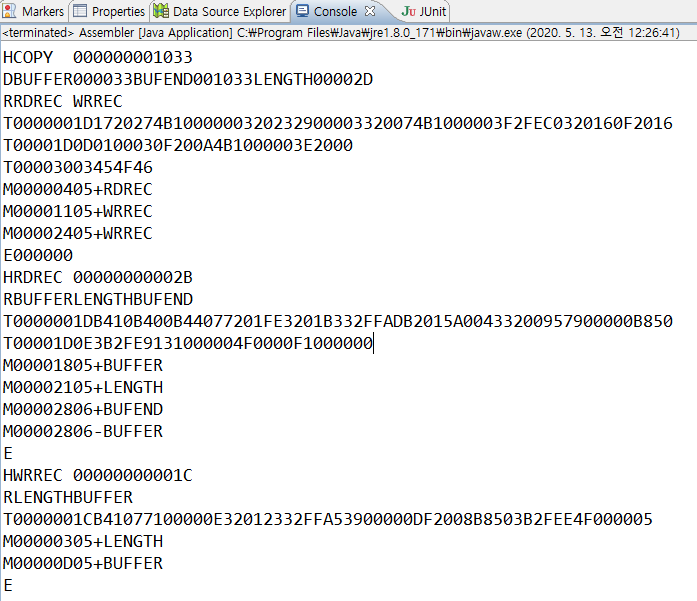
[그림 4-8] File에 출력된 LITTAB



[그림 4-9] File에 출력된 SYMTAB



[그림 4-10] Console에서 SYMTAB과 LITTAB의 출력



[그림 4-11] Console에서 최종 변환된 Object Code의 출력

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

* 1. **모듈별 구현 내용**

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

1. **Assembler**

|  |
| --- |
| /\*\*  \* Assembler :  \* 이 프로그램은 SIC/XE 머신을 위한 Assembler 프로그램의 메인 루틴이다.  \* 프로그램의 수행 작업은 다음과 같다.  \* 1) 처음 시작하면 Instruction 명세를 읽어들여서 assembler를 세팅한다.  \* 2) 사용자가 작성한 input 파일을 읽어들인 후 저장한다.  \* 3) input 파일의 문장들을 단어별로 분할하고 의미를 파악해서 정리한다. (pass1)  \* 4) 분석된 내용을 바탕으로 컴퓨터가 사용할 수 있는 object code를 생성한다. (pass2)  \*  \*  \* 작성중의 유의사항 :  \* 1) 새로운 클래스, 새로운 변수, 새로운 함수 선언은 얼마든지 허용됨. 단, 기존의 변수와 함수들을 삭제하거나 완전히 대체하는 것은 안된다.  \* 2) 마찬가지로 작성된 코드를 삭제하지 않으면 필요에 따라 예외처리, 인터페이스 또는 상속 사용 또한 허용됨.  \* 3) 모든 void 타입의 리턴값은 유저의 필요에 따라 다른 리턴 타입으로 변경 가능.  \* 4) 파일, 또는 콘솔창에 한글을 출력시키지 말 것. (채점상의 이유. 주석에 포함된 한글은 상관 없음)  \*  \*  \* + 제공하는 프로그램 구조의 개선방법을 제안하고 싶은 분들은 보고서의 결론 뒷부분에 첨부 바랍니다. 내용에 따라 가산점이 있을 수 있습니다.  \*/  **public** **class** Assembler {  **public** **static** **final** **int** ***MAX\_LENGTH***=30; //Text Record 최대    /\*\* instruction 명세를 저장한 공간 \*/  InstTable instTable;  /\*\* 읽어들인 input 파일의 내용을 한 줄 씩 저장하는 공간. \*/  ArrayList<String> lineList;  /\*\* 프로그램의 section별로 symbol table을 저장하는 공간\*/  ArrayList<SymbolTable> symtabList;  /\*\* 프로그램의 section별로 literal table을 저장하는 공간\*/  ArrayList<LiteralTable> literaltabList;  /\*\* 프로그램의 section별로 프로그램을 저장하는 공간\*/  ArrayList<TokenTable> TokenList;  /\*\*  \* Token, 또는 지시어에 따라 만들어진 오브젝트 코드들을 출력 형태로 저장하는 공간.  \* 필요한 경우 String 대신 별도의 클래스를 선언하여 ArrayList를 교체해도 무방함.  \*/  ArrayList<String> codeList;    /\*\* 레지스터 정보를 저장하는 공간 \*/  RegisterTable registerTable;    **int** locctr; //location counter    /\*\* pass2에서 쓰일 변수\*/  **int** textRecordLength; // textRecord 길이와  String textRecord; // object code를 누적하는 buffer  **int** startLocation; // text record 시작 location을 저장하는 데에 쓰인다    /\*\*  \* 클래스 초기화. instruction Table을 초기화와 동시에 세팅한다.  \*  \* **@param** instFile : instruction 명세를 작성한 파일 이름.  \*/  **public** Assembler(String instFile) {  instTable = **new** InstTable(instFile);  lineList = **new** ArrayList<String>();  symtabList = **new** ArrayList<SymbolTable>();  literaltabList = **new** ArrayList<LiteralTable>();  TokenList = **new** ArrayList<TokenTable>();  codeList = **new** ArrayList<String>();  registerTable = **new** RegisterTable();  }  /\*\*  \* 어셈블러의 메인 루틴  \*/  **public** **static** **void** main(String[] args) {    Assembler assembler = **new** Assembler("inst.data");    **try** {  assembler.loadInputFile("input.txt");  assembler.pass1();  assembler.printSymbolTable("symtab\_20160458.txt");  assembler.printLiteralTable("literaltab\_20160458.txt");  assembler.pass2();  assembler.printObjectCode("output\_20160458.txt");    } **catch** (Exception e) {  e.printStackTrace();  }    //메모리 해제는 garbage collector의 대상이 되어 해제됨  }  /\*\*  \* inputFile을 읽어들여서 lineList에 저장한다.  \* **@param** inputFile : input 파일 이름.  \*/  **private** **void** loadInputFile(String inputFile) **throws** IOException {  File file = **new** File("./"+inputFile);    **if**(!file.exists())  **throw** **new** IOException("there is no file");    BufferedReader br=**new** BufferedReader(**new** FileReader(file));  String line;  **while**((line = br.readLine())!=**null**){  lineList.add(**new** String(line));  }  br.close();    }  /\*\*  \* pass1 과정을 수행한다.  \* 1) 프로그램 소스를 스캔하여 토큰단위로 분리한 뒤 토큰테이블 생성  \* 2) label을 symbolTable에 정리  \*  \* 주의사항 : SymbolTable과 TokenTable은 프로그램의 section별로 하나씩 선언되어야 한다.  \*/  **private** **void** pass1() **throws** Exception {    locctr=0;  **int** section=0;  **int** tokenIndex=0;  Token token; //package 같으므로 사용 가능  **int** byteSize;    //먼저 main program COPY에 대한 section별 table을 만든다  symtabList.add(**new** SymbolTable());  literaltabList.add(**new** LiteralTable());  TokenList.add(**new** TokenTable(symtabList.get(section), literaltabList.get(section),instTable,registerTable));    **for** (String line : lineList){    **if**(line.charAt(0)=='.') //주석일 경우 건너뛴다.  **continue**;      TokenList.get(section).putToken(line);  token = TokenList.get(section).getToken(tokenIndex);    **if**(token.operator.equals("CSECT")){  //CSECT인 경우  // 원래는 방금 전 넣었던 Token을 삭제하려 했으나  // 나중에 section이 끝나는 point를 알기 위해 END로 operator를 고쳐준다.  // section별로 하나씩 END point가 생긴다.  // 새로운 section에 맞는 table들 생성 후 다시 CESCT를 Token으로 넣기  TokenList.get(section).getToken(tokenIndex).operator=**new** String("END");  TokenList.get(section).programLength=locctr; //program길이 추가    section ++; // section이 늘어남  tokenIndex=0; //초기화  locctr=0; //초기화    symtabList.add(**new** SymbolTable());  literaltabList.add(**new** LiteralTable());  TokenList.add(**new** TokenTable(symtabList.get(section), literaltabList.get(section),instTable,registerTable));  TokenList.get(section).putToken(line);  }    token = TokenList.get(section).getToken(tokenIndex);//최근 추가한 token에 대해  token.location=locctr; //현재 주소를 Token의 location에 할당(추후 pass2에서 쓰인다)    /\*\* label symtab에 추가 \*/  **if**(token.label!=**null**){  symtabList.get(section).putSymbol(token.label, locctr);  }    /\*\* operator인 경우 \*/  **if**((byteSize = instTable.searchByte(token.operator))>0){    token.byteSize=byteSize; //token byteSize 설정  locctr += byteSize; //location counter에 누적한다.    }**else**{  /\*\* directivies인 경우 \*/  **if**(token.operator.equals("EQU")){  // label에 할당된 주소를 고쳐줘야할 수도 있음.  // 이미 symtab의 해당 label은 location에 현재 주소가 잘 할당된 상태.  // operand가 \* 인 경우는 locctr을 넣으면 되지만, 아닐 경우 따로 값을 구해야 한다.  // - 연산에 대해서만 처리가 가능하다.  // EQU MAXLEN, EQU BUFEND-BUFFER는 가능하지만, EQU 4096이나, +의 경우에는 불가능하다  **if**(token.operand[0].charAt(0)!='\*'){  **int** EQUValue = 0; //symbol table에 들어갈 EQU 값  StringTokenizer str = **new** StringTokenizer(token.operand[0],"-");  EQUValue = symtabList.get(section).search(str.nextToken());  **if**(str.hasMoreTokens()){  EQUValue -= symtabList.get(section).search(str.nextToken());  }    symtabList.get(section).modifySymbol(token.label, EQUValue);  }    }**else** **if**(token.operator.equals("RESW")){  locctr += Integer.*parseInt*(token.operand[0]) \* 3; //할당 개수\*3byte 차지한다    }**else** **if**(token.operator.equals("RESB")){  locctr += Integer.*parseInt*(token.operand[0]); //할당 개수\*1byte 차지한다    }**else** **if**(token.operator.equals("WORD")){  token.byteSize=3; //byteSize설정  locctr += 3; //3byte만 사용한다    }**else** **if**(token.operator.equals("BYTE")){  **if**(token.operand[0].charAt(0)=='X'){  //X인 경우  StringTokenizer str = **new** StringTokenizer(token.operand[0],"'");  str.nextToken(); //X걸러내고  byteSize = str.nextToken().length()/2;  token.byteSize=byteSize; //byteSize설정  locctr += byteSize; //hex이므로 2로 나눈다  }**else**{  //C인 경우  StringTokenizer str = **new** StringTokenizer(token.operand[0],"'");  str.nextToken(); //C걸러내고  byteSize = str.nextToken().length();  token.byteSize=byteSize; //byteSize설정  locctr += byteSize; //char이므로 문자 길이 그대로 byte를 소요한다  }    }**else** **if**(token.operator.equals("LTORG")){  // literal들에 대해 주소 할당, locctr update  locctr = literaltabList.get(section).addAddLiteral(locctr);  }**else** **if**(token.operator.equals("END")){  // 할당하지 않은 literal들에 대해 주소 할당, locctr update  locctr = literaltabList.get(section).addAddLiteral(locctr);  TokenList.get(section).programLength=locctr; //program length추가  section++;  }    }  //pass1에서 EXTREF, EXTDEF는 넘어간다.    /\*\* Literal 처리 \*/  **if**(token.operand!=**null** && token.operand[0]!=**null** && token.operand[0].charAt(0)=='='){  StringTokenizer str = **new** StringTokenizer(token.operand[0], "'"); //=C'EOF'에서 EOF만 빼내기 위해  str.nextToken(); //=C' 혹은 =X' 분리    **if**(token.operand[0].charAt(1)=='X'){  //X인 경우  literaltabList.get(section).putLiteral(str.nextToken(),-1);  }**else**{  //C인 경우  literaltabList.get(section).putLiteral(str.nextToken(),-2);  }    }    tokenIndex++;    }    }    /\*\*  \* 작성된 SymbolTable들을 출력형태에 맞게 출력한다.  \* **@param** fileName : 저장되는 파일 이름  \* null인자로 들어올 경우 콘솔로 출력한다  \*/  **private** **void** printSymbolTable(String fileName) **throws** IOException {    **if**(fileName==**null**){  //콘솔 출력  System.***out***.println("--------\* Symbol Table \*----------");  **for**(SymbolTable st : symtabList){  **int** i=0;  **for**(String str : st.symbolList){    System.***out***.format("%s\t%X\n",str,st.locationList.get(i));  i++;  }  }  }**else**{  //File 출력  File file = **new** File("./"+fileName);  String buffer;    **if**(!file.exists())  file.createNewFile();    FileWriter fw = **new** FileWriter(file);    **for**(SymbolTable st : symtabList){    **int** i=0;    **for**(String str : st.symbolList){    buffer = str+"\t"+Integer.*toHexString*(st.locationList.get(i)).toUpperCase()+"\n";;  fw.write(buffer);  i++;  }  }    fw.flush(); //혹시 적히지 않은 것이 있다면 흘려보냄  fw.close();  }    }    /\*\*  \* 작성된 LiteralTable들을 출력형태에 맞게 출력한다.  \* **@param** fileName : 저장되는 파일 이름  \* null인자로 들어올 경우 콘솔로 출력한다  \*/  **private** **void** printLiteralTable(String fileName) **throws** IOException {    **if**(fileName==**null**){  //콘솔 출력  System.***out***.println("--------\* Literal Table \*----------");  **for**(LiteralTable lt : literaltabList){    **int** i=0;    **for**(String str : lt.literalList){    System.***out***.format("%s\t%X\n",str,lt.locationList.get(i));  i++;  }    }  }**else**{  //File 출력  File file = **new** File("./"+fileName);  String buffer;    **if**(!file.exists())  file.createNewFile();    FileWriter fw = **new** FileWriter(file);    **for**(LiteralTable lt : literaltabList){    **int** i=0;    **for**(String str : lt.literalList){    buffer = str+"\t"+Integer.*toHexString*(lt.locationList.get(i)).toUpperCase()+"\n";  fw.write(buffer);  //fw.flush();  i++;  }  }  fw.flush(); //혹시 적히지 않은 것이 있다면 흘려보냄  fw.close();        }  }  /\*\*  \* pass2 과정을 수행한다.  \* 1) 분석된 내용을 바탕으로 object code를 생성하여 codeList에 저장.  \*/  **private** **void** pass2() {  String objectCode; // 단 한 명령어의 object code 임시 저장소  **int** section = 0; // section Index  **boolean** literalRecord = **false**; // 해당 section의 literal이 이미 record 되었는지 확인하기 위한 변수  Token t;    **for**(TokenTable tt : TokenList){    textRecord = **new** String("");  literalRecord = **false**;  **for**(**int** index=0 ; index < tt.tokenList.size() ; index++){    t = tt.tokenList.get(index);    **if**(t.operator.equals("START") || t.operator.equals("CSECT") || t.operator.equals("EXTDEF") || t.operator.equals("EXTREF")){  // 이 경우 바로 object code를 받아 list에 저장하면 된다.  tt.makeObjectCode(index);  objectCode = tt.getObjectCode(index);  codeList.add(**new** String(objectCode));    }**else** **if**(t.operator.equals("END")){  // 만약 LTORG가 없어 literal이 적히지 않았다면, Literal record를 추가한다  **if**(!literalRecord)  addLiteralRecord(section);  // Text Record가 남아있을 수 있으므로 먼저 끝낸다  finTextRecord();  // section을 이동하기 전, Modify Record 추가  addModifyRecord(section);  // E record 추가  **if**(section==0){  //main program일경우  codeList.add(String.*format*("E%06X", 0));  // 원래는 START address를 저장했어야 하는데  // 어디에 loading될지 모르기 때문에 웬만하면 시작 주소가 0이므로 0으로 처리하였다.  }**else**{  codeList.add(**new** String("E")); //아닌 경우, E만 추가한다  }    }**else** **if**(t.operator.equals("RESW")){  // TextRecord 끝내기  finTextRecord();    }**else** **if**(t.operator.equals("RESB")){  // TextRecord 끝내기  finTextRecord();    }**else** **if**(t.operator.equals("LTORG")){  //Literal들을 추가한다  addLiteralRecord(section);  literalRecord = **true**;    }**else** **if**(t.operator.equals("EQU")){  //아무것도 하지 않는다    }**else** {  // 나머지 instruction, word, byte의 경우  // 바로 codeList에 추가하면 안된다. TextRecord이므로 더해야함  tt.makeObjectCode(index);  objectCode = tt.getObjectCode(index);  addTextRecord(objectCode, t.byteSize,t.location);  }    }  section++; //section 이동  }    }    /\*\*  \* 작성된 codeList를 출력형태에 맞게 출력한다.  \* **@param** fileName : 저장되는 파일 이름  \* fileName이 null일 경우, 콘솔에 출력한다  \*/  **private** **void** printObjectCode(String fileName) **throws** IOException {    **if**(fileName==**null**){  //console 출력  **for**(String str : codeList){  System.***out***.println(str);  }    }**else**{  //File 출력    File file = **new** File("./"+fileName);    **if**(!file.exists())  file.createNewFile();    FileWriter fw = **new** FileWriter(file);    **for**(String str : codeList){  fw.write(str+"\n");  }  fw.flush(); //혹시 적히지 않은 것이 있다면 흘려보냄  fw.close();  }    }    /\*\*  \* object code를 Text record에 누적시킨다  \* 만약 record 길이가 30을 넘어가면 현재 Text record는 끝내고, 새로운 record를 만들어 넣는다.  \* **@param** objectCode : 추가할 object code  \* **@param** byteSize : 추가할 object code의 길이  \*/  **private** **void** addTextRecord(String objectCode, **int** byteSize, **int** location){    **if**(textRecord.length()==0){  //만약 record가 시작되지 않은 경우, 새로 시작  startLocation = location; //시작 주소를 저장한다  textRecord += objectCode;  textRecordLength = byteSize;    }**else** **if**( (textRecordLength + byteSize) > ***MAX\_LENGTH***){  //길이가 30byte를 넘어가는 경우, 현재 record를 끝내고 새로 시작  finTextRecord();  startLocation = location;  textRecord += objectCode;  textRecordLength = byteSize;    }**else**{  //길이가 30byte를 넘어가지 않는 경우 누적만 시킨다  textRecord += objectCode;  textRecordLength += byteSize; //length도 누적시킨다    }      }    /\*\*  \* 현재 Text record를 기록하고 끝낸다.  \*/  **private** **void** finTextRecord(){  **if**(textRecord.length()==0){ //record가 시작되지 않은 경우 끝낼 것도 없으므로 그냥 돌아간다  **return**;  }**else**{ //끝낼 record가 있는 경우  textRecord = String.*format*("T%06X%02X",startLocation,textRecordLength) + textRecord;  //시작주소, 길이, 지금까지 누적한 Record들을 한 번에 Text Record로 만든다.  codeList.add(**new** String(textRecord));  textRecord = **new** String(""); //새로 시작하기 위해 만들어준다  textRecordLength = 0; //textRecord와 Length reset  }  }    /\*\*  \* 해당 section의 Modify Record를 모두 object code list에 추가한다  \* **@param** section : 해당 section index  \*/  **private** **void** addModifyRecord(**int** section){    **for**(Modify m : TokenList.get(section).modifyList){  codeList.add(**new** String(m.objectCode));  }    }    /\*\*  \* 해당 section의 Literal을 모두 text record에 추가한다  \* **@param** section : 해당 section index  \*/  **private** **void** addLiteralRecord(**int** section){  **int** index=0; //literal List index  **int** byteSize;  String str=**new** String("");    **for**(String literal : TokenList.get(section).literalTab.literalList){    **if**(TokenList.get(section).literalTab.charOrHexList.get(index)==-1){  //X인 경우  byteSize=literal.length()/2;  str = literal;  }**else**{  //C인 경우  byteSize=literal.length();    **for**(**int** i=0 ; i<byteSize ; i++){  //해당 literal 각각을 ASCII code로 변경한다  //Character.getNumericValue(c)는 unicode로 변경된다  //ASCII code로 바꾸기 위해 int로 casting 한다.  str += String.*format*("%X", (**int**)literal.charAt(i));  }  }  //Literal이 Text Record의 시작이 될 수도 있으므로 location이 필요하다.  addTextRecord(str,byteSize,TokenList.get(section).literalTab.locationList.get(index));  index++;  }  }    } |

pass1은, 코드가 긴만큼 아쉬운 점이 많은 함수이다. operator가 START거나 CSECT면 새로운 섹션이 시작되기 때문에 TokenTable을 추가해야 한다. 그러나 일단 Token이 분리되기 전까지는 operator에 무엇이 들어갈 지 알 수 없기 때문에 일단 확인 없이 tokenList에 넣게 된다. 일단 넣고 확인해야 하는 구조가 아쉬웠다. 나는 처음에는 CSECT를 넣은 것을 제거하고 새로 Table을 만들어 넣는 방법을 쓰다가, pass2 구현이 용이하도록 일단 넣어버렸기 때문에 section이 끝난다는 의미로 operator를 END로 고치는 방법을 택했다.

또한, 주석에서도 볼 수 있겠지만 여기서는 START인 경우를 처리해주지 않고 있다. 프로그램 길이만 구하면 되기 때문에 start address를 딱히 저장할 이유가 없다. 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.

pass2도 마찬가지이다. 대부분의 처리는 TokenTable의 makeObjectCode가 한다. 그러나 pass2에서도, makeOjectCode에서도 둘 다 operator를 이중으로 확인한다는 점이 아쉬웠다. 그러나 코드를 만드는 부분이 분리된다는 점은 굉장히 편리했고 코드의 가독성을 높였다.

1. **TokenTable**

|  |
| --- |
| /\*\*  \* 사용자가 작성한 프로그램 코드를 단어별로 분할 한 후, 의미를 분석하고, 최종 코드로 변환하는 과정을 총괄하는 클래스이다. <br>  \* pass2에서 object code로 변환하는 과정은 혼자 해결할 수 없고 symbolTable과 instTable의 정보가 필요하므로 이를 링크시킨다.<br>  \* section 마다 인스턴스가 하나씩 할당된다.  \*  \*/  **public** **class** TokenTable {  **public** **static** **final** **int** ***MAX\_OPERAND***=3;    /\* bit 조작의 가독성을 위한 선언 \*/  **public** **static** **final** **int** ***nFlag***=32;  **public** **static** **final** **int** ***iFlag***=16;  **public** **static** **final** **int** ***xFlag***=8;  **public** **static** **final** **int** ***bFlag***=4;  **public** **static** **final** **int** ***pFlag***=2;  **public** **static** **final** **int** ***eFlag***=1;    /\* Token을 다룰 때 필요한 테이블들을 링크시킨다. \*/  SymbolTable symTab;  LiteralTable literalTab;  InstTable instTab;  RegisterTable registerTab;    /\*\* 각 line을 의미별로 분할하고 분석하는 공간. \*/  ArrayList<Token> tokenList;    /\*\* 각 section별 길이 \*/  **int** programLength;    /\*각 section별 modify record를 저장하는 공간\*/  ArrayList<Modify> modifyList;    /\*\*  \* 초기화하면서 symTable과,literalTable, instTable을 링크시킨다.  \* **@param** symTab : 해당 section과 연결되어있는 symbol table  \* **@param** literalTab : 해당 section과 연결되어있는 literal table  \* **@param** instTab : instruction 명세가 정의된 instTable  \* **@param** registerTab : register name-번호가 정의된 register table  \*/  **public** TokenTable(SymbolTable symTab, LiteralTable literalTab, InstTable instTab, RegisterTable registerTab) {    **this**.symTab=symTab;  **this**.literalTab=literalTab;  **this**.instTab=instTab;  **this**.registerTab=registerTab;  tokenList=**new** ArrayList<Token>();  modifyList=**new** ArrayList<Modify>();  }  /\*\*  \* 일반 문자열을 받아서 Token단위로 분리시켜 tokenList에 추가한다.  \* **@param** line : 분리되지 않은 일반 문자열  \*/  **public** **void** putToken(String line) {  tokenList.add(**new** Token(line));  }    /\*\*  \* tokenList에서 index에 해당하는 Token을 리턴한다.  \* **@param** index  \* **@return** : index번호에 해당하는 코드를 분석한 Token 클래스  \*/  **public** Token getToken(**int** index) {  **return** tokenList.get(index);  }    /\*\*  \* Pass2 과정에서 사용한다.  \* instruction table, symbol table literal table 등을 참조하여 objectcode를 생성하고, 이를 저장한다.  \* **@param** index  \*/  **public** **void** makeObjectCode(**int** index){  // END, LTROG 등 몇몇 경우에는 이 함수를 call하지 않는다. pass2()에서 알아서 처리한다.  // Token의 object code에 생성한 object code를 저장한다.    Token t = tokenList.get(index);  //int byteSize = 0; //해당 object Code 길이  **int** pc=0; //pc값  **int** disp=0; //disp 값을 저장  **int** opcode=0; //opcode를 저장    **if**(t.operator.equals("START") || t.operator.equals("CSECT")){  //Header Record, 시작 주소는 CS program이고, 어디에 올라갈지 모르므로 0일 것이다.  t.objectCode = String.*format*("H%-6s%06X%06X", t.label, 0, programLength);    }**else** **if**(t.operator.equals("EXTDEF")){  //D record  t.objectCode=**new** String("D");  **int** i=0;  **while**(i<***MAX\_OPERAND*** && t.operand[i]!=**null**){  //Array Index Bound Exception을 막기 위해 i<MAX\_OPERAND 조건은 꼭 넣어줘야 한다.  //내보낼 symbol이름과 주소를 추가한다  t.objectCode += String.*format*("%s%06X", t.operand[i],symTab.search(t.operand[i]));  i++;  }    }**else** **if**(t.operator.equals("EXTREF")){  //R record  t.objectCode=**new** String("R");  **int** i=0;  **while**(i<***MAX\_OPERAND*** && t.operand[i]!=**null**){  t.objectCode += String.*format*("%-6s",t.operand[i]); //주소는 모르므로 symbol만 추가한다  i++;  }    }**else** **if**(t.operator.equals("WORD")){  // WORD인 경우  // 상수, 2개의 symbol 뺄셈 연산, 하나의 symbol만 가지는 경우 처리 가능하다.  // 덧셈 연산은 하지 못한다.    pc = t.location+t.byteSize;  StringTokenizer str = **new** StringTokenizer(t.operand[0],"-");  String searchStr = str.nextToken();  **try**{  disp = Integer.*parseInt*(searchStr); //상수인 경우    }**catch**(NumberFormatException e){  //symbol인 경우  disp = symTab.search(searchStr);    **if**(disp < 0){ //없는 symbol의 경우  //Modify unit추가  disp = 0;  modifyList.add(**new** Modify(t.location, 6, '+', searchStr)); //WORD이므로 WORD 쨰로 수정이 필요  }    **if**(str.hasMoreTokens()){  // "-"로 분리가되는, 뺄셈 연산이 있었을 경우  searchStr = str.nextToken();  **int** subDisp = symTab.search(searchStr);    **if**(subDisp < 0){ //없는 symbol의 경우  //Modify unit추가  subDisp=0;  modifyList.add(**new** Modify(t.location, 6, '-', searchStr)); //WORD이므로 WORD 쨰로 수정이 필요  }  disp -= subDisp;  //disp 설정 (만약 이 section에 없는것 - 있는것 일 경우  //'-있는것'이 disp로 남게된다.  }    **if**(disp < 0){  disp &= (**int**)Math.*pow*(2, 24) -1; //음수인 경우 WORD3byte모두에 대해 처리 해야한다  }    t.objectCode=String.*format*("%06X", disp);  }        }**else** **if**(t.operator.equals("BYTE")){  //BYTE인 경우 (따로 -,+에 대해 처리 해주지 않았다)  pc = t.location+t.byteSize;  //C와 X구분  **if**(t.operand[0].charAt(0)=='X'){  //X인 경우  StringTokenizer str = **new** StringTokenizer(t.operand[0],"'");  str.nextToken(); //X' 분리  t.objectCode = **new** String(str.nextToken());  }**else**{  //C인 경우, ASCII code로 변환한다  t.objectCode = **new** String("");  StringTokenizer str = **new** StringTokenizer(t.operand[0],"'");  str.nextToken(); //C' 분리  **for**(**int** i=0; i < t.byteSize ; i++){  t.objectCode += String.*format*("%02X", (**int**)str.nextToken().charAt(i));  }    }    }**else**{  //Instruction인 경우  pc = t.location+t.byteSize;  opcode = instTab.searchOpcode(t.operator);    **if**(t.getFlag(1)==1){  //4형식인 경우  // 사칙 연산에 대한 처리를 해주지 못했다  opcode <<= 4; //nixbpe 자리를 만들어주기 위해 shift left 4를 한다.  opcode |= t.nixbpe; //nixbpe bit or  //모든 nixbpe를 사용하는 경우에서, base relative에 대한 처리를 해주지 못함. pc relative만 사용한다.  disp = symTab.search(t.operand[0]); //operand의 주소를 찾는다    **if**(disp<0){  disp = 0; //없는 symbol인 경우(EXTREF로 사용한 symbol인 경우)  //modify record를 추가한다  modifyList.add(**new** Modify(t.location+1, 5, '+', t.operand[0])); //operand 부분만 수정이 필요하다  }**else**{  disp -= pc;  opcode |= 2; //pc relative, p=1 설정  }    **if**(disp<0){  // disp가 음수인 경우, int 크기 때문에 20bit로 format을 맞출 수 없다.  // 원하는 만큼 자르기 위해 bit and를 이용한다.  disp &= (**int**)Math.*pow*(2, 20) - 1;  }    t.objectCode=String.*format*("%03X%05X", opcode,disp);    }**else** **if**(t.byteSize == 2){  //2형식인 경우, nixbpe를 사용하지 않는다.  //register에 대해 계산한다.  t.objectCode=String.*format*("%02X%X%X", opcode, registerTab.getNumber(t.operand[0]),registerTab.getNumber(t.operand[1]));  }**else** **if**(instTab.searchNumberOfOperand(t.operator)==1){  //3형식이고 operand가 있는 경우  opcode <<= 4; //nixbpe 자리를 만들어주기 위해 shift left 4를 한다.  opcode |= t.nixbpe; //nixbpe bit or    **if**(t.getFlag(48)==48){  //n=i=1인 경우    **if**(t.operand[0].charAt(0)=='='){  //literal인 경우  StringTokenizer str = **new** StringTokenizer(t.operand[0],"'");  str.nextToken(); //=C', =X' 분리  disp=literalTab.search(str.nextToken()); //LITTAB에서 target address를 찾는다  disp-=pc; //disp를 구한다  opcode |=2; // p=1 설정  }**else**{  // 일반 access  // 4형식이 아니므로 operand가 symtab에 있어야 한다. (4형식처럼 없으면 안된다.)  disp = symTab.search(t.operand[0]);  disp -= pc;  opcode |=2; //disp를 구하고 p=1로 설정한다  }  }**else** **if**(t.getFlag(16)==16){  // 이 if문이 먼저 나오면 n=i=1인 경우도 들어가므로 후에 해주어야 한다.  // immediate인 경우  // # MAXLEN같은 경우는 처리해주지 않았다.  // 상대 주소가 아닌 진짜 값이므로 p=1처리 해주지 않는다.  StringTokenizer str = **new** StringTokenizer(t.operand[0],"#");  disp = Integer.*parseInt*(str.nextToken());  }**else** {  //indirect인 경우(1형식인 경우나 n=i=0인 경우도 여기 들어오지만, 처리하지 못한다.)  StringTokenizer str = **new** StringTokenizer(t.operand[0],"@");  disp = symTab.search(str.nextToken());  disp -= pc;  opcode |= 2; //상대 주소이므로 p=1로 처리해준다.  }    **if**(disp<0){  //disp가 음수인 경우, 하위 3column만 들어갈 수 있도록 bit and 해준다.  disp &= (**int**)Math.*pow*(2, 12) - 1;  }    t.objectCode=String.*format*("%03X%03X", opcode, disp);    }**else**{  //3형식이고 operand가 없는 경우  //1형식인 경우도 이곳에 포함되지만 처리하지 못함  opcode <<= 4; //nixbpe 자리를 만들어주기 위해 shift left 4를 한다.  opcode |= t.nixbpe; //nixbpe bit or, p값 설정할 필요 없음    t.objectCode=String.*format*("%03X%03X",opcode,0);    }  }  }    /\*\*  \* index번호에 해당하는 object code를 리턴한다.  \* **@param** index  \* **@return** : object code  \*/  **public** String getObjectCode(**int** index) {  **return** tokenList.get(index).objectCode;  }    } |

makeObjectCode는 마찬가지로 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.

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. **Token**

|  |
| --- |
| /\*\*  \* 각 라인별로 저장된 코드를 단어 단위로 분할한 후 의미를 해석하는 데에 사용되는 변수와 연산을 정의한다.  \* 의미 해석이 끝나면 pass2에서 object code로 변형되었을 때의 바이트 코드 역시 저장한다.  \*/  **class** Token{  //의미 분석 단계에서 사용되는 변수들  **int** location;  String label;  String operator;  String[] operand;  String comment;  **char** nixbpe;  // object code 생성 단계에서 사용되는 변수들  String objectCode;  **int** byteSize;    /\*\*  \* 클래스를 초기화 하면서 바로 line의 의미 분석을 수행한다.  \* **@param** line 문장단위로 저장된 프로그램 코드  \*/  **public** Token(String line) {  //initialize 추가  parsing(line);  }    /\*\*  \* line의 실질적인 분석을 수행하는 함수. Token의 각 변수에 분석한 결과를 저장한다.  \* **@param** line 문장단위로 저장된 프로그램 코드.  \*/  **public** **void** parsing(String line) {  /\*\* Token Parsing \*/  StringTokenizer str = **new** StringTokenizer(line,"\t");  StringTokenizer operandStr;  String operands;    **if**(line.charAt(0)!='\t'){ //label이 있는 경우  label = **new** String(str.nextToken().toString());  }    operator = **new** String(str.nextToken().toString());    **if**(str.hasMoreTokens()){    **int** i=0;    operand=**new** String[TokenTable.***MAX\_OPERAND***]; //MAX\_OPERAND개 생성, 자동으로 null 들어감  operands=**new** String(str.nextToken().toString());  operandStr=**new** StringTokenizer(operands, ",");    **while**(operandStr.hasMoreTokens() && i < TokenTable.***MAX\_OPERAND***){  operand[i]=**new** String(operandStr.nextToken().toString());  i++;  }  // 사실 이 때 operand에 comment가 들어가는 경우가 생긴다. (RSUB같이 operand 개수가 0인 경우)  // 이것에 대해 고민을 많이 했으나,  // 추후에 TokenTable에서 operand=0인 경우 operand를 들여다보지 않을 것이므로 괜찮다고 결론지었다.    /\*\*flag setting을 token parsing 때 해준다.\*/  **if**(operator.charAt(0)=='+'){  **this**.setFlag(1, 1); //e=1  }    **if**(operand[1]!=**null** && operand[1].charAt(0)=='X'){  **this**.setFlag(8, 1);  }  //nixbpe  **if**(operand[0]!=**null**){    **if**(operand[0].charAt(0)=='@'){  **this**.setFlag(32, 1); //n=1    }**else** **if**(operand[0].charAt(0)=='#'){  **this**.setFlag(16, 1); //i=1    }**else**{  **this**.setFlag(48, 1); //n=i=1  }    //b,p에 대한 값은 추후에 object code를 설정할때 pc relative가 가능한지 보고 결정한다.  }    }    **if**(str.hasMoreTokens()){  comment = **new** String(str.nextToken().toString());  }      }    /\*\*  \* n,i,x,b,p,e flag를 설정한다.  \*  \* 사용 예 : setFlag(nFlag, 1);  \* 또는 setFlag(TokenTable.nFlag, 1);  \*  \* **@param** flag : 원하는 비트 위치  \* **@param** value : 집어넣고자 하는 값. 1또는 0으로 선언한다.  \*/  **public** **void** setFlag(**int** flag, **int** value) {  **if**(value==1){  nixbpe |= flag; //bit or  }**else**{  //0인 경우  nixbpe &= (~flag);  }  }    /\*\*  \* 원하는 flag들의 값을 얻어올 수 있다. flag의 조합을 통해 동시에 여러개의 플래그를 얻는 것 역시 가능하다  \*  \* 사용 예 : getFlag(nFlag)  \* 또는 getFlag(nFlag|iFlag)  \*  \* **@param** flags : 값을 확인하고자 하는 비트 위치  \* **@return** : 비트위치에 들어가 있는 값. 플래그별로 각각 32, 16, 8, 4, 2, 1의 값을 리턴할 것임.  \*/  **public** **int** getFlag(**int** flags) {  **return** nixbpe & flags; //nixbpe & 32, 48 등 bit and값을 리턴  }  } |

Token을 만드는 과정에서, c에서는 operand 개수를 먼저 확인하고 그 개수만큼 저장을 하도록 했다. 그러나 이 class에서는 InstTable이 맵핑 되어있지 않기 때문에, 확인 할 수가 없어서 무작정 operand에 분리해 넣어야 했다. 그러다 보니 RSUB같이 operand 개수가 0개인 경우 comment가 종종 operand에 들어가는 일이 생겼다. 처음에는 해당하지 않는 정보가 들어가는 것에 대해 고민을 많이 했다. 추후에 TokenTable에서 operand 개수를 검사해 0인 경우 operand를 들여다 보지 않을 것이므로 괜찮다고 결론지었다.

1. **Modify**

|  |
| --- |
| /\*\*  \* Modify Record를 미리 저장해놓는 class  \* Modify Record가 하나씩 나올때마다 생성해놓았다가  \* 추후에 section이 끝날 때 실제로 Record를 적는다  \*/  **class** Modify{  **int** location; // 고칠 곳의 주소  **int** length; // 고칠 곳의 길이  **char** plus; // '+' or '-'  String name; // 더하거나 뺄 symbol 이름  String objectCode; //modify object code    **public** Modify(**int** location, **int** length, **char** plus, String name){  **this**.location=location;  **this**.length=length;  **this**.plus=plus;  **this**.name=name;  **this**.objectCode = String.*format*("M%06X%02X%c%s", location, length, plus, name);  }  } |

1. **InstTable**

|  |
| --- |
| /\*\*  \* 모든 instruction의 정보를 관리하는 클래스. instruction data들을 저장한다  \* 또한 instruction 관련 연산, 예를 들면 목록을 구축하는 함수, 관련 정보를 제공하는 함수 등을 제공 한다.  \*/  **public** **class** InstTable {  /\*\*  \* inst.data 파일을 불러와 저장하는 공간.  \* 명령어의 이름을 집어넣으면 해당하는 Instruction의 정보들을 리턴할 수 있다.  \*/  HashMap<String, Instruction> instMap;    /\*\*  \* 클래스 초기화. 파싱을 동시에 처리한다.  \* **@param** instFile : instuction에 대한 명세가 저장된 파일 이름  \*/  **public** InstTable(String instFile) {  instMap = **new** HashMap<String, Instruction>();    **try** {  openFile(instFile);  } **catch** (IOException e) {  e.printStackTrace();  }  }    /\*\*  \* 입력받은 이름의 파일을 열고 해당 내용을 파싱하여 instMap에 저장한다.  \* 기계어 목록파일 형식은 자유롭게 구현한다. 다음과 같이 inst.data에 저장되어있다.  \* **@param** fileName : instuction에 대한 명세가 저장된 파일 이름  \* ========================================================  \* 이름-형식/기계어 코드/오퍼랜드의 갯수/NULL  \* =========================================================  \* 이름이 key, 나머지는 Instruction으로 생성되어 value로 저장된다.  \*/  **public** **void** openFile(String fileName) **throws** IOException {    File file = **new** File("./"+fileName);    **if**(!file.exists())  **throw** **new** IOException("there is no file");    BufferedReader br=**new** BufferedReader(**new** FileReader(file));  String line;  StringTokenizer str;  **while**((line = br.readLine())!=**null**){  str = **new** StringTokenizer(line,"-");  instMap.put(str.nextToken(), **new** Instruction(str.nextToken()));  }    br.close();  }    /\*\*  \* 해당 명령어가 몇 byte짜리 명령어인지 byte수를 리턴하는 함수  \* **@param** mnemonic : byte를 찾을 명령어  \* **@return** 명령어가 소요하는 byte. 명령어가 아닐 경우 -1을 리턴한다.  \*/  **public** **int** searchByte(String mnemonic){    Instruction value;    **if**(mnemonic.charAt(0)=='+')  **return** 4;  **else**  value=instMap.get(mnemonic);    **if**(value==**null**)  **return** -1;    **return** value.format;  }    /\*\*  \* 해당 명령어가 몇개의 operand를 갖고 있는지 operand 수를 리턴하는 함수  \* **@param** mnemonic : operand 개수를 찾을 명령어  \* **@return** 명령어가 갖는 operand의 개수, 명령어가 아닐 경우 -1을 리턴한다.  \*/  **public** **int** searchNumberOfOperand(String mnemonic){    Instruction value;    **if**(mnemonic.charAt(0)=='+')  value=instMap.get(mnemonic.substring(1));  **else**  value=instMap.get(mnemonic);    **if**(value==**null**)  **return** -1;    **return** value.numberOfOperand;  }    /\*\*  \* 해당 명령어의 opcode를 찾는 함수  \* **@param** mnemonic : opcode를 찾을 명령어  \* **@return** 명령어의 opcode, 명령어가 아닐 경우 -1을 리턴한다.  \*/  **public** **int** searchOpcode(String mnemonic){    Instruction value;    **if**(mnemonic.charAt(0)=='+')  value=instMap.get(mnemonic.substring(1));  **else**  value=instMap.get(mnemonic);    **if**(value==**null**)  **return** -1;    **return** value.opcode;    }  } |

1. **Instruction**

|  |
| --- |
| /\*\*  \* 명령어 하나하나의 구체적인 정보는 Instruction클래스에 담긴다.  \* instruction과 관련된 정보들을 저장하고 기초적인 연산을 수행한다.  \*/  **class** Instruction {  /\*  \* 각자의 inst.data 파일에 맞게 저장하는 변수를 선언한다.  \*  \* ex)  \* String instruction;  \* int opcode;  \* int numberOfOperand;  \* String comment;  \*/    String mnemonic; //명령어 이름을 저장  **int** opcode; //명령어 opcode 숫자를 저장  **int** numberOfOperand; //명령어의 operand 개수를 저장    /\*\* instruction이 몇 바이트 명령어인지 저장. 이후 편의성을 위함 \*/  **int** format;    /\*\*  \* 클래스를 선언하면서 일반문자열을 즉시 구조에 맞게 파싱한다.  \* **@param** line : instruction 명세파일로부터 한줄씩 가져온 문자열  \*/  **public** Instruction(String line) {  parsing(line);  }    /\*\*  \* 일반 문자열을 파싱하여 instruction 정보를 파악하고 저장한다.  \* **@param** line : instruction 명세파일로부터 한줄씩 가져온 문자열  \*/  **public** **void** parsing(String line) {    StringTokenizer str = **new** StringTokenizer(line,"/");    format = Integer.*parseInt*(str.nextToken());  opcode = Integer.*parseInt*(str.nextToken(),16); // Hex로 받음  numberOfOperand = Integer.*parseInt*(str.nextToken());    }      } |

1. **SymbolTable**

|  |
| --- |
| /\*\*  \* symbol과 관련된 데이터와 연산을 소유한다.  \* section 별로 하나씩 인스턴스를 할당한다.  \*/  **public** **class** SymbolTable {  ArrayList<String> symbolList;  ArrayList<Integer> locationList;  // 기타 literal, external 선언 및 처리방법을 구현한다.    /\*\* 생성자 \*/  **public** SymbolTable(){  symbolList = **new** ArrayList<String>();  locationList = **new** ArrayList<Integer>();  }    /\*\*  \* 새로운 Symbol을 table에 추가한다.  \* **@param** symbol : 새로 추가되는 symbol의 label  \* **@param** location : 해당 symbol이 가지는 주소값  \* 주의 : 만약 중복된 symbol이 putSymbol을 통해서 입력된다면 이는 프로그램 코드에 문제가 있음을 나타낸다.  \* 매칭되는 주소값의 변경은 modifySymbol()을 통해서 이루어져야 한다.  \*/  **public** **void** putSymbol(String symbol, **int** location) **throws** Exception {    **if**(symbolList.indexOf(symbol)>=0){  **throw** **new** Exception("Same symbol already exists.");  }    symbolList.add(**new** String(symbol));  locationList.add(**new** Integer(location));    }    /\*\*  \* 기존에 존재하는 symbol 값에 대해서 가리키는 주소값을 변경한다.  \* **@param** symbol : 변경을 원하는 symbol의 label  \* **@param** newLocation : 새로 바꾸고자 하는 주소값  \*/  **public** **void** modifySymbol(String symbol, **int** newLocation) {    **int** index = symbolList.indexOf(symbol);  locationList.set(index, **new** Integer(newLocation));  }    /\*\*  \* 인자로 전달된 symbol이 어떤 주소를 지칭하는지 알려준다.  \* **@param** symbol : 검색을 원하는 symbol의 label  \* **@return** symbol이 가지고 있는 주소값. 해당 symbol이 없을 경우 -1 리턴  \*/  **public** **int** search(String symbol) {  **int** address = 0;    **int** index = symbolList.indexOf(symbol);  **if**(index < 0)  address = -1;  **else** address = locationList.get(index);    **return** address;  }        } |

1. **LiteralTable**

|  |
| --- |
| /\*\*  \* literal과 관련된 데이터와 연산을 소유한다.  \* section 별로 하나씩 인스턴스를 할당한다.  \*/  **public** **class** LiteralTable {  ArrayList<String> literalList;  ArrayList<Integer> locationList;  ArrayList<Integer> charOrHexList; //해당 literal이 char인지 hex인지 저장한다    **public** LiteralTable(){  literalList=**new** ArrayList<String>();  locationList=**new** ArrayList<Integer>();  charOrHexList=**new** ArrayList<Integer>();  }    /\*\*  \* 새로운 Literal을 table에 추가한다.  \* **@param** literal : 새로 추가되는 literal의 label  \* **@param** location : 해당 literal이 가지는 주소값  \* \*구분을 위해 X인 경우 location은 -1, C인 경우 location은 -2로 넣는다\*  \* charOrHex도 마찬가지로 구분을 위해 X는 -1, C는 -2로 넣는다  \* 주의 : 만약 중복된 literal이 putLiteral을 통해서 입력된다면 이는 프로그램 코드에 문제가 있음을 나타낸다.  \* 매칭되는 주소값의 변경은 modifyLiteral()을 통해서 이루어져야 한다.  \*/  **public** **void** putLiteral(String literal, **int** location) {    **if**(literalList.indexOf(literal)>=0) //이미 해당 literal이 추가되어 있다면 추가할 필요가 없음  **return**;    literalList.add(**new** String(literal));  locationList.add(**new** Integer(location));  charOrHexList.add(**new** Integer(location));  }    /\*\*  \* 기존에 존재하는 literal 값에 대해서 가리키는 주소값을 변경한다.  \* **@param** literal : 변경을 원하는 literal의 label  \* **@param** newLocation : 새로 바꾸고자 하는 주소값  \*/  **public** **void** modifyLiteral(String literal, **int** newLocation) {  **int** index = literalList.indexOf(literal);  locationList.set(index, **new** Integer(newLocation));  }    /\*\*  \* 인자로 전달된 literal이 어떤 주소를 지칭하는지 알려준다.  \* **@param** literal : 검색을 원하는 literal의 label  \* **@return** literal이 가지고 있는 주소값. 해당 literal이 없을 경우 -1 리턴  \*/  **public** **int** search(String literal) {  **int** address = 0;  **int** index = literalList.indexOf(literal);    **if**(index<0)  address = -1;  **else**  address = locationList.get(index);    **return** address;  }    /\*\*  \* 해당 섹션의 모든  \* **@param** locctr : literal에 주소를 추가하기 위해 locctr을 인자로 갖고온다  \* **@return** 모든 literal에 주소 할당이 끝난 후 pass1에서 locctr을  \* update 해주기 위해 누적된 locctr을 return한다  \*/  **public** **int** addAddLiteral(**int** locctr){    **int** index = 0;  **int** byteSize = 0;    **if**(locationList.get(index)>0){  **return** locctr;  // 만약 이미 주소가 할당되었다면 돌아가기  // LTORG, 혹은 END를 만나면 한번에 그 section의 모든 literal을 할당한다.  // 따라서 이미 하나라도 주소 할당이 되었다면, 모두 주소가 할당된 것이므로 돌아간다.  }    /\*\* 할당 안된 literal 모두 할당\*/  **for**(String literal : literalList){    **if**(charOrHexList.get(index)==-1){  //X인 경우  byteSize=literal.length()/2;  }**else**{  //C인 경우  byteSize=literal.length();  }  **this**.modifyLiteral(literal, locctr); //해당 literal 주소 수정  locctr+=byteSize;  index++;  }    **return** locctr; //locctr을 반환하여 assember에서의 locctr을 업데이트한다.    }      } |

1. **RegisterTable**

|  |
| --- |
| /\*\*  \* Register와 그에 해당하는 번호를 저장하는 Table이다.  \* 하나만 생성된다.  \*/  **public** **class** RegisterTable {    HashMap<String, Integer> registerMap;    **public** RegisterTable(){    registerMap = **new** HashMap<String,Integer>();    //registerMap에 register이름 - 매칭 번호를 차례로 넣는다.  registerMap.put("A", 0);  registerMap.put("X", 1);  registerMap.put("L", 2);  registerMap.put("B", 3);  registerMap.put("S", 4);  registerMap.put("T", 5);  registerMap.put("F", 6);  registerMap.put("PC", 8);  registerMap.put("SW", 9);    }    /\*\*  \* 인자에 해당하는 이름의 register의 번호를 알려준다.  \* **@param** name : 번호를 찾을 register의 이름  \* **@return** name이 null이면 0을 리턴한다.  \* name이 null이 아닌 경우, Map을 찾아 register의 번호를 리턴한다.  \* 없는 register의 경우 -1을 리턴한다.  \*/  **public** **int** getNumber(String name){    **if**(name==**null**)  **return** 0;    Integer number = registerMap.get(name);    **if**(number==**null**)  **return** -1;    **return** number;  }  } |

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.

C와 비교하자면, section별로 SYMTAB, LITTAB, TokenTable이 생성되는 것이 가장 큰 장점이라고 생각한다. 확실히 section별로 Table을 생성하고 그 안에서만 참조가 일어나니 section flag를 굳이 하나 더 넣지 않아도 되고, 관리가 잘 되어 편리했다. 또한 JAVA는 API가 잘 되어있는 언어이기 때문에 ArrayList와 HashMap를 쓰면서 C언어로 배열에 동적 할당을 할 때보다 편리하게 put과 search 등을 구현할 수 있었다.

그러나 pointer를 직접적으로 쓸 수 없어 Assembler에서 object code를 만들 때, 멤버 변수로 textRecord와 textLength, text start address를 잡은 것이 아쉽다. 이것은 Assembler의 변수라기 보다는 임시 버퍼이기 때문에 함수 변수로 잡는 게 맞을 것 같은데 말이다.

Assembler의 pass2와 TokenTable의 makeObjectCode 부분이 분리가 된 것은 역할이 분리가 되어 가독성을 좋게 했지만 ‘pass2에서도 operator를 확인하고, makeObjectCode에서도 operator를 확인하는 것은 이중으로 확인하는 것 아닌가?’라는 생각이 들기도 했다. makeObjectCode에 H,R,D,T record를 구분할 수 있는 flag를 넣어줬다면 이중 확인은 안 해도 됐을 것 같다.

또, 원래는 Class의 멤버 변수는 private으로 선언하는 것이 대부분인데, 접근을 쉽게 하기 위해 default로 선언했다는 점이 고쳐야 할 부분 같다.

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.