시스템 프로그래밍

-Project1. Assembler-

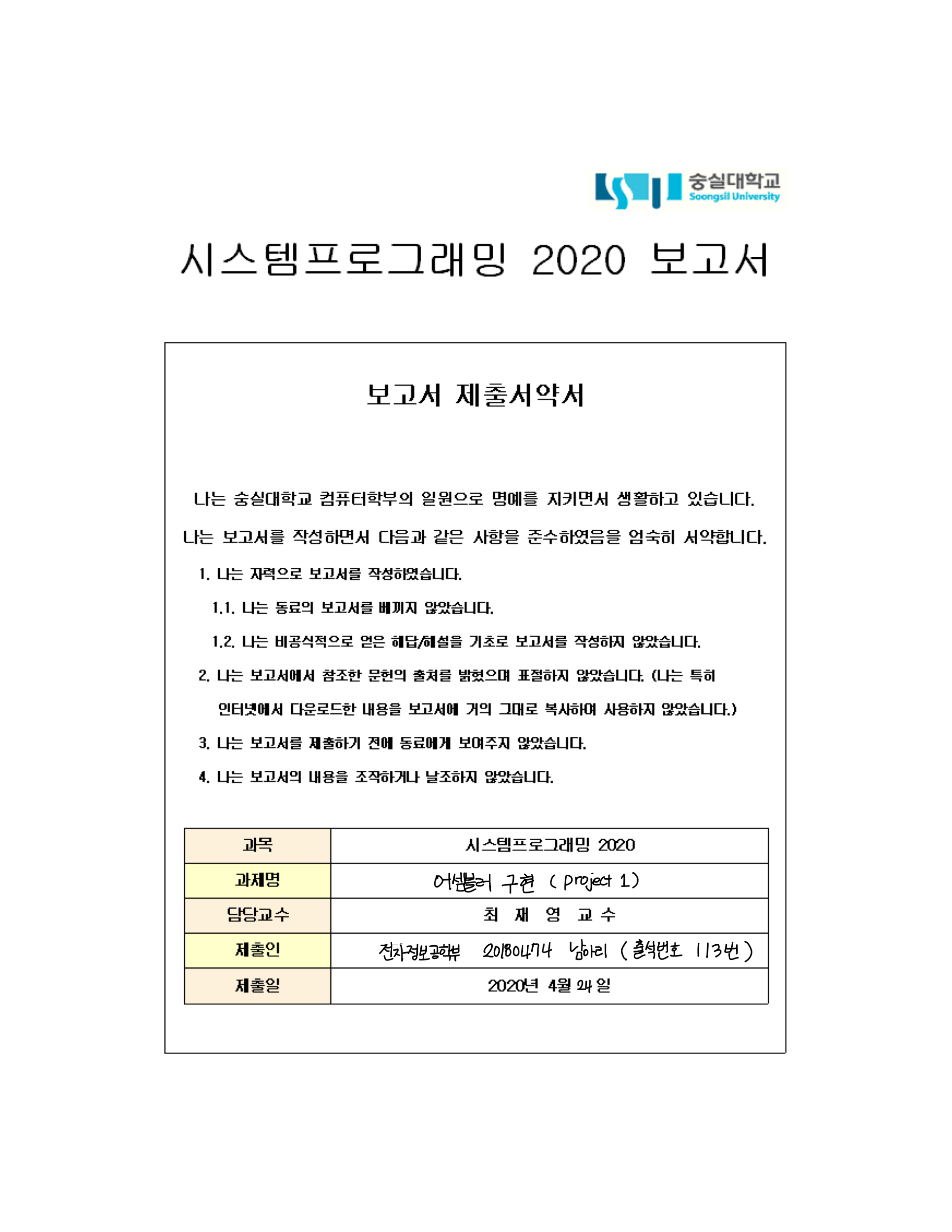
CS시스템프로그래밍 월수 12시반

최재영 교수님

전자정보공학부 전자전공

출석번호 113

20180474 남아리



차례

1장 프로젝트 개요

1. 개발 동기
2. 개발 목적

2장 배경지식

1. 주제에 관한 배경지식
2. 기술적 배경지식

3장 시스템 설계 내용

1. 설계 이미지화
2. 전체 시스템 설계 내용
3. 모듈별 설계 내용

4장 시스템 구현 내용

1. 전체 시스템 구현 내용
2. 모듈별 구현 내용
3. 구현 화면

5장 기대효과 및 결론, 고찰

+) 출처 링크

\*소스코드는 압축 파일로 첨부했습니다.

**1장 프로젝트 개요**

**(1) 개발 배경 및 목적**

프로그램 소스파일을 입력 받아, 오브젝트 코드를 생성하는 어셈블러를 구현한다. 먼저 명령어의 OPCODE가 적힌 DATA파일과 어셈블리 할 소스파일을 입력 받는 작업을 한다. 입력 받은 파일을 각각의 테이블로 만들고 PASS1에서 파싱을 통해 테이블을 만든다. 그리고 그 테이블을 이용해 PASS2에서 주소를 연산하며, 마지막으로 오브젝트코드, 리터럴 테이블, 심볼 테이블을 출력한다.

이를 통해 SIC/XE머신의 기본적인 구조를 이해할 수 있으며, 손으로 하던 주소 계산을 프로그래밍으로 바꾸기 위해 생각하면서, addressing 방식을 좀 더 체계적이고 정확하게 정립할 수 있다. 또한 PASS1, PASS2 등 각각의 구성들이 어셈블러 안에서 무슨 역할을 하며, 어떻게 동작하고 생성되는지 이해할 수 있다. 그리고 직접 구현해봄으로써 테이블이 만들어지는 과정, 소스코드를 오브젝트코드로 출력하는 과정을 이해하고 최종적으로 어셈블러의 동작과정을 이해할 수 있다.

**2장 시스템 설계 내용**

**(1) 주제에 대한 배경지식**

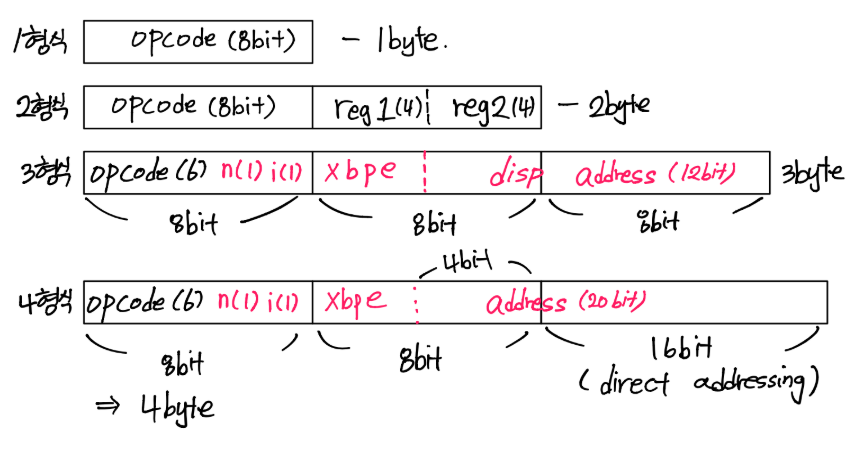
1. 레지스터

SIC/XE 머신은 SIC 머신을 호환해야하기 때문데 SIC머신에서 사용되던 A,X,L,PC,SW도 사용가능하며 XE 머신에서는 B,F,S,T가 추가되었다. 길이는 F레지스터를 제외하고는 모두 24-bit이며 F레지스터는 48-bit이다.

|  |  |  |
| --- | --- | --- |
| Register 이름 | Number | 용도 |
| A | 0 | Accumulator |
| X | 1 | Index Register |
| L | 2 | Link : 돌아올 주소를 저장한다. |
| PC | 8 | Program Counter : 다음 명령문의 주소를 저장한다. |
| SW | 9 | 플래그 |
| B | 3 | Base register : 주소 할당시 사용 |
| S | 4 | General working register |
| T | 5 | General working register |
| F | 6 | Floating – point accumulator (48bit) |

1. Format

Operator의 형식에 따라 프로그램 주소를 할당하는 것이 다르다. 형식은 다음과 같다.



1. Addressing mode

1. n= 1, i = 1, p = 1

PC relative : disp = Target address – pc counter

2. n= 1, i = 1, b = 1

Base relative이다. 떨어진 정도가 너무 멀어서 PC relative로 주소를 12bit에 표현하지 못할 때 사용한다.

disp = Target address – base address

3. n= 0, i = 1

Immediate addressing이다. (ex. LDA #3) 메모리 레퍼런스가 일어나지 않는다. 연산할 필요없이 해당 값을 16진수로 바꾸어 바로 사용한다.

4. n= 1, i = 0

Indirect addressing이다. (ex. J @RETADR) 메모리 레퍼런스가 2번 일어난다. Target address안에 있는 value를 가져온다.

5. x=1

x레지스터가 사용되었을 때이다. 예를 들면 STCH BUFFER,X같은 instruction에서 사용된다.

6. e=1

4형식이다. Direct addressing이므로 Target address를 그대로 가져와서 사용한다.

1. Pass

1. Pass1 (테이블 생성, 메모리 주소 할당)

- 모든 instruction에 주소를 할당한다. (locctr)

- 소스파일을 파싱하여 토큰별로 테이블을 만든다.

- 모든 레이블에 할당된 주소의 값을 저장한다.

2. Pass2 (Pass1의 테이블을 이용하여 instruction 어셈블)

- Pass1에서 만든 테이블을 바탕으로 명령어를 오피코드로 변환하고 주소를 찾아 연산하여 각 문장의 최종 프로그램 주소들을 생성한다.

- Pass1에서 처리가 안된 지시어나 심볼 등을 처리한다.

1. Literal

Constant operand의 값을 프로그램의 다른 곳에 따로 정의하거나 리터럴에 대한 라벨을 따로 만들지 않기 위해 리터럴을 사용한다.

LTORG를 만나면 앞에 나온 리터럴들이 한꺼번에 묶인다.

LTORG를 사용하지 않으면 프로그램 맨 밑에 묶이는데, 거리가 멀면 참조하면서 miss가 발생한다.

1. Control Section

Program Block은 program 부분과 data 부분이 섞여 있어서 그 부분들을 분리하여 재배치하는 것이다. Control Section도 마찬가지로 분리하는 것이지만 program block과는 달리, COPY, RDREC, WRREC를 세 개의 소스 파일로 구분하여 각각에 대해 오브젝트 코드를 생성하는 것이다. 그렇게 분리된 오브젝트 코드는 추후 링커에 의해 하나의 프로그램으로 합쳐지며, 각각의 소스파일은 로더에 의해 서로 참조할 수 있다. EXTDEF 과 EXTREF를 사용하며, EXTDEF는 해당 소스파일에 정의된 것을 외부에서 참조할 수 있다는 의미이고, EXTREF는 외부에 정의된 것을 참조할 것이라는 의미이다.

주소 계산 시, 주의해야할 점은 외부 참조로 BUFFER를 참조했을 때, 해당 파일에서는 주소를 알 수 없다는 점이다. 그러므로 주소 값에 ZERO를 넣어야 한다.

**(2) 기술적 배경지식**

1. Strtok

입력 받은 문장을 토큰(구분자)로 구분해준다.

Strtok(name,”token”)의 형식으로 사용하며 name에는 구분자로 나누고 싶은 문장이 들어간 변수의 이름을 적고, token에는 구분자를 (예를 들면 \t) 적으면 된다. Token\_parsing 함수나 입출력 등 여러 함수 안에서 유용하게 사용되었다. 이미 구분자로 잘라낸 뒷 문장을 그대로 사용하고 싶다면 변수의 이름을 적는 자리에 NULL을 사용하면 된다. 그러면 앞에서 자른 문장을 그대로 이어받아 사용할 수 있다. Strtok(NULL,”\t”);

1. File

FILE\* fp = NULL;

fp = fopen(file\_name,”w”);

. . .

fprintf(fp, "%s \t", symbol\_token[i].label);

파일을 입출력 할 때 사용하는 함수이다. FILE\*은 처리할 파일을 받아온다. 다른함수로는 fopen, fprintf, fclose 등이 있다. fopen은 위와 같이 사용했으며 첫 번째 인자에는 처리할 파일명, 두 번째 인자에는 파일을 처리할 모드를 썼다. 모드에는 w,wt,r등이 있다. Fprintf는 (처리할 파일명, 형식제어 문자, 출력할 값)의 순서로 작성한다.

1. Struct

struct 구조체이름 {

자료형 멤버이름;

};

typedef struct 구조체이름 변수이름;

구조체는 비슷한 속성을 가진 것들을 한 번에 모아 놓은 것이다. 위와 같은 형태로 정의한다.

포인터 구조체 변수는 token\_table[index]->operand; 처럼 ‘->’을 이용하여 멤버를 지시하고, 일반적인 구조체 변수는 symbol\_token[i].operator1 처럼 ‘.’을 이용한다.

struct token\_unit {

char \*label;

char \*operator1;

char \*operand[MAX\_OPERAND];

char \*comment;

char nixpbpe;

int extents; //구역

};

typedef struct token\_unit token;

token \*token\_table[MAX\_LINES];

token symbol\_token[MAX\_LINES];

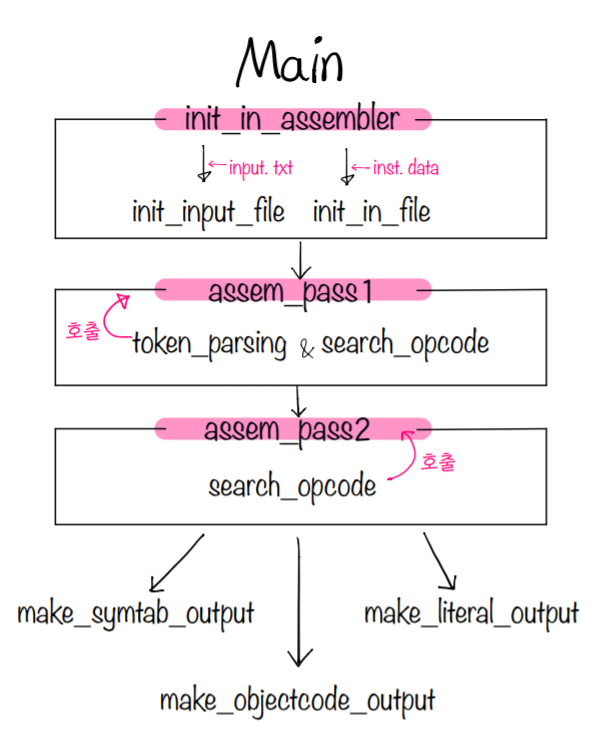
1. Shift 연산

Shift를 이용하면 비트연산을 할 수 있다. 변수 << 이동할 비트 수 의 형태로 사용한다. op\_tab[i] = op\_tab[i] << 4; 의 형태로 사용하였으며, Pass2에서 각 operator의 형식에 따라 xbpe 등이 들어갈 공간을 확보하여, 프로그램 주소를 계산하기 위해 사용되었다.

만약 0000 0011 의 값을 갖는 변수를 <<2 했다면, 0000 1100 이 된다.

**3장 시스템 설계 내용**

**(1) 설계의 이미지화**



**(2) 전체 시스템 설계 내용**

1. main함수가 실행된다.

2. init\_in\_assembler를 통해서 init\_input\_file과 init\_in\_file 함수가 호출된다. 그리고 각각 소스파일인 input.txt와 기계어 목록 파일인 inst.data 파일이 입력되어 테이블로 만들어진다.

3. assem\_pass1에 의해 token\_parsing 함수가 호출되면서 입력 받은 소스파일을 토큰으로 구분하는 토큰테이블(symbol\_token[ ])을 생성한다. 그리고 search\_opcode를 호출하여 오퍼레이터가 명령어인지 지시어 등인지 구분하고 각각에 맞게 location counter를 계산한다. 또한 Sym\_table을 생성하고, sym\_table과 liter\_table에 주소(addr)를 추가한다.

4. assem\_pass2에 의해, search\_opcode가 호출되고 해당 오퍼레이터의 inst.data파일에서의 인덱스를 반환한다. 그리고 해당 오퍼레이터의 형식을 가져와, 형식마다 switch 문으로 구분하고, 각각에 맞게 프로그램 주소를 연산한다.

5. 마지막으로, make\_objectcode\_output에 의해 object code가 파일 출력으로 생성되고, make\_symtab\_output과 make\_literal\_output에 의해 화면에 symbol table과 literal table이 출력된다.

**(3) 각 모듈별 설계 내용**

* **int init\_my\_assembler(void)**

프로그램을 초기화하고 파일을 읽는다.

* **int init\_inst\_file(char \*inst\_file)**

기계어 목록 파일인 inst.data 파일을 읽고, 기계어 목록 테이블을 생성한다.

* **int init\_input\_file(char \*input\_file)**

어셈블리 할 소스코드인 input.txt 파일을 읽고, 테이블을 생성한다.

* **int search\_opcode(char \*str)**

기계어인지 확인하고, 기계어라면 목록 테이블의 해당 인덱스를 반환한다. 기계어가 아니라면 -1을 반환하여 pass1에서 RESB 같은 지시어를 처리하기 위해 구분할 때 호출된다.

* **int token\_parsing(int index)**

소스코드를 토큰 단위로 나누고 토큰 테이블을 생성한다.

Literal을 처리하여 literal table(liter tab)을 생성한다.

* **static int assem\_pass1(void)**

어셈블러의 pass1과정을 수행한다. 파싱함수(token\_parsing)를 사용하여 토큰테이블을 만든다.

* **static int assem\_pass2(void)**

pass1에서 만들어진 테이블을 바탕으로 pc counter와 target address를 계산하여 프로그램 주소를 만든다.

* **void make\_objectcode\_output(char \*file\_name)**

최종 오브젝트 코드를 파일에 출력한다.

* **void make\_symtab\_output(char \*file\_name)**

최종 심볼테이블을 화면에 출력한다.

* **void make\_literal\_output(char \*file\_name)**

최종 리터럴 테이블을 화면에 출력한다.

**4장 시스템 구현 내용**

**(1) 전제 시스템 구현 내용**

* **기본 함수에 대한 구현 내용은 설계했던 내용에서 달라지지 않았다.**
* **search\_symbol**

해당 symbol이 이미 테이블에 저장되어 있는지를 확인하기 위해 새로 만든 함수이며 assem\_pass1에서 호출된다.

* **search\_placed\_literal**

리터럴이 테이블에 저장되어 있는지 확인하기 위해 새로 만든 함수이며 token\_parsing에서 호출된다.

* **Make\_literal\_output**

콘솔 화면에 리터럴 테이블을 출력하기 위해 새로 만든 함수이다. 메인함수에서 호출된다.

* **Make-symtab\_output**

콘솔 화면에 심볼 테이블을 출력하기 위해 새로 만든 함수이다. 메인함수에서 호출된다.

* **Assem\_pass2**

pass1에서 만들어진 테이블을 바탕으로 프로그램 주소를 만든다.

format에 따라 스위치문으로 구분한다. 그리고 각각의 format에 맞게 pc counter와 target address를 계산하여 프로그램 주소를 만든다.

특히, 2,3,4형식에서는 shift연산을 이용하여 해당 bit만큼 공간을 할당하고 값을 추가했다.

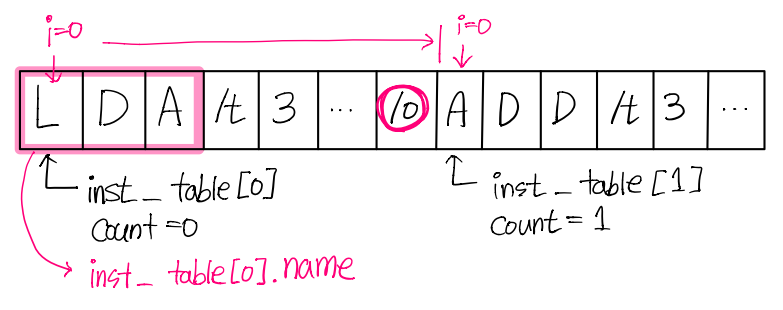
**(2) 각 모듈의 구현 내용**

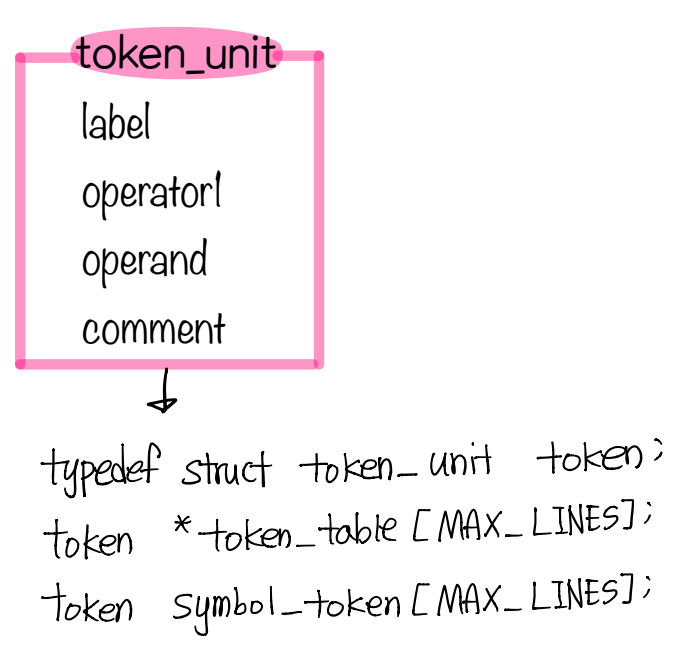
* inst\_table
* obj inst\_table[MAX\_INST];
* init\_in\_file에서 사용된다.
* 구조체 obj로 정의된다.
* 기계어 목록 파일인 inst.data 파일을 입력 받아 테이블을 만든다.
* 기계어 목록 파일로부터 다음 OPCODE값이 순차적으로 들어왔다고 가정하면,

LDA 3 00 /0

ADD 3 18 /0

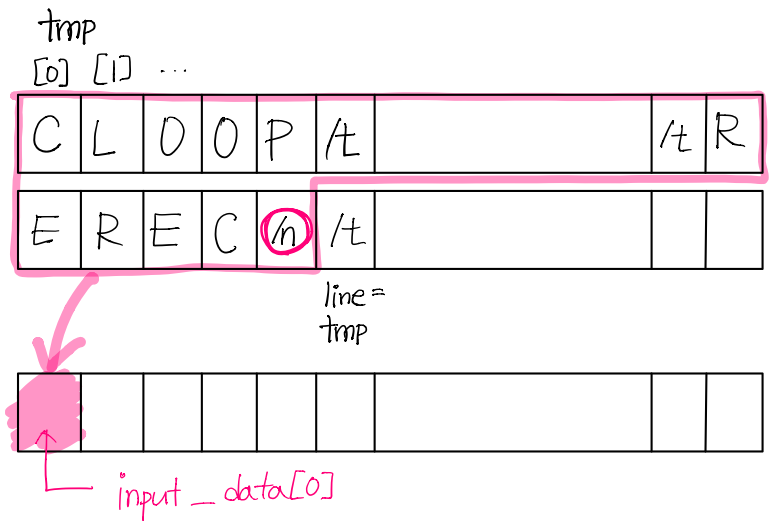
다음과 같이 저장되어 TABLE이 만들어질 것이라고 생각할 수 있다.

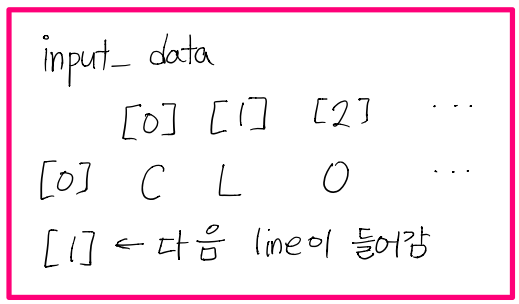


* input\_data
* char \*input\_data[MAX\_LINES];
* init\_input\_file에서 사용된다.
* 임시 저장용 tmp[100]에 개행 문자가 나오기 전까지 소스코드 input.txt 한 줄을 입력 받는다.
* tmp에 저장된 string을 input\_data[instruction\_cnt]에 저장한다.
* 라인단위로 관리된다.
* Input.txt의 일부로 저장방식을 이미지화 한다면, 다음과 같이 생각할 수 있다.

CLOOP +JSUB REREC

LDA LENGTH





* token\_unit
* struct token\_unit {

char \*label;

char \*operator1;

char \*operand[MAX\_OPERAND];

char \*comment;

int extents;

};

typedef struct token\_unit token;

token \*token\_table[MAX\_LINES];

token symbol\_token[MAX\_LINES];

* 어셈블리 할 소스코드를 토큰단위로 관리하기 위한 구조체 변수이다.
* assem\_pass1과 token\_parsing에서 사용된다.
* 먼저 line의 첫 번째 글자가 주석일 때를 처리한다.
* 라벨이 있는 경우와 없는 경우로 나눈 뒤, 라벨이 없다면 strcpy와 strtok를 사용하여 operator를 symbol\_token[].operator1에 저장한다.
* 그 뒤, 오퍼랜드가 있는지 없는지를 따져준다.
* 오퍼랜드는 없는데 뒤에 comment가 있을 때를 처리해준다.
* 오퍼랜드가 있다면, 구분자 탭과 쉼표를 통해 오퍼랜드 1,2,3을 저장한다.
* 이 과정을 라벨이 있는 경우도 동일하게 수행한다.
* literal\_unit
* typedef struct literal\_unit {

char\* literal;

int addr;

int extents;

}literal;

literal liter\_tab[MAX\_LINES];

* 리터럴을 관리하는 구조체이다
* Symbol\_unit
* struct symbol\_unit {

char symbol[10];

int extents; //구역 나눔

int addr;

};

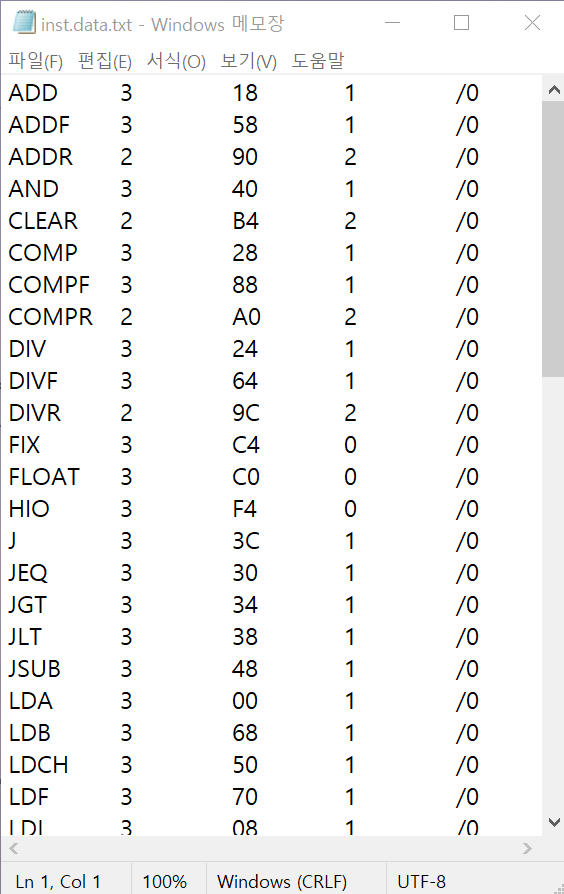
typedef struct symbol\_unit symbol;

symbol sym\_table[MAX\_LINES];

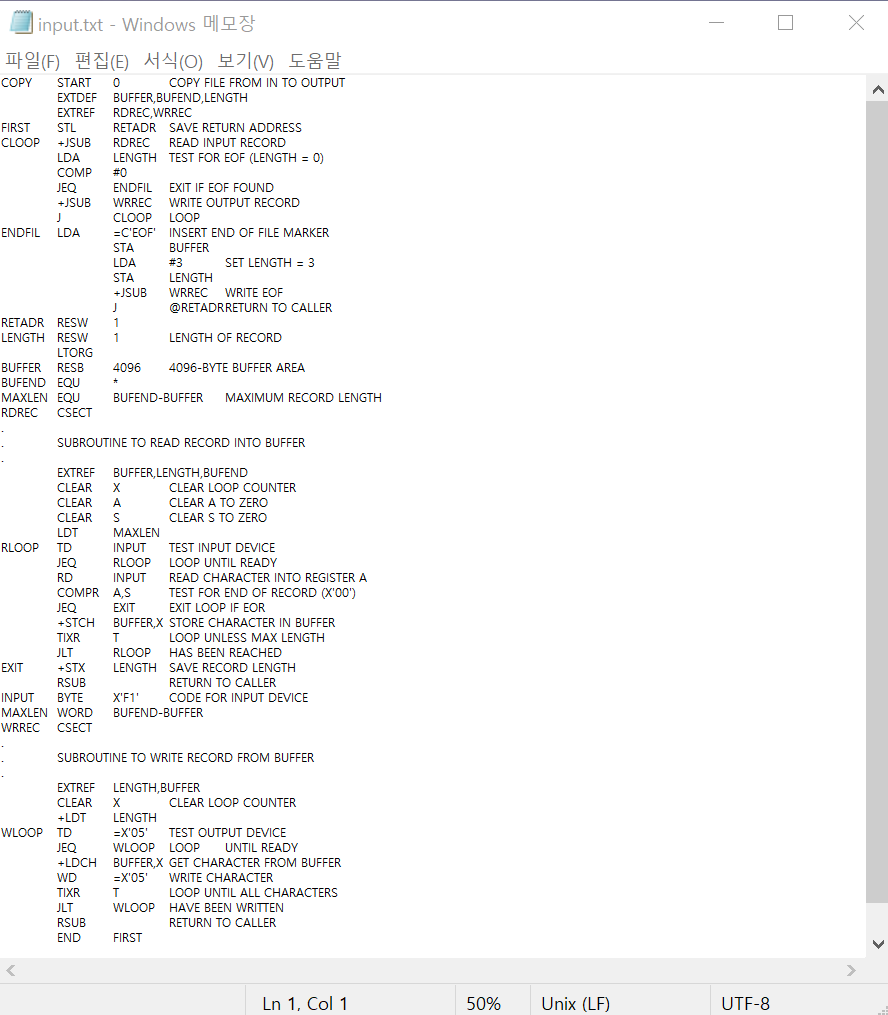
* 심볼을 관리하는 구조체이다.
* Assem\_pass1, assem\_pass2, search\_symbol, make\_objectcode\_output에서 사용된다.
* Location Counter index
* int location\_counter\_index[MAX\_LINES];
* 라인마다 location counter를 저장하는 배열 형태의 변수이다.

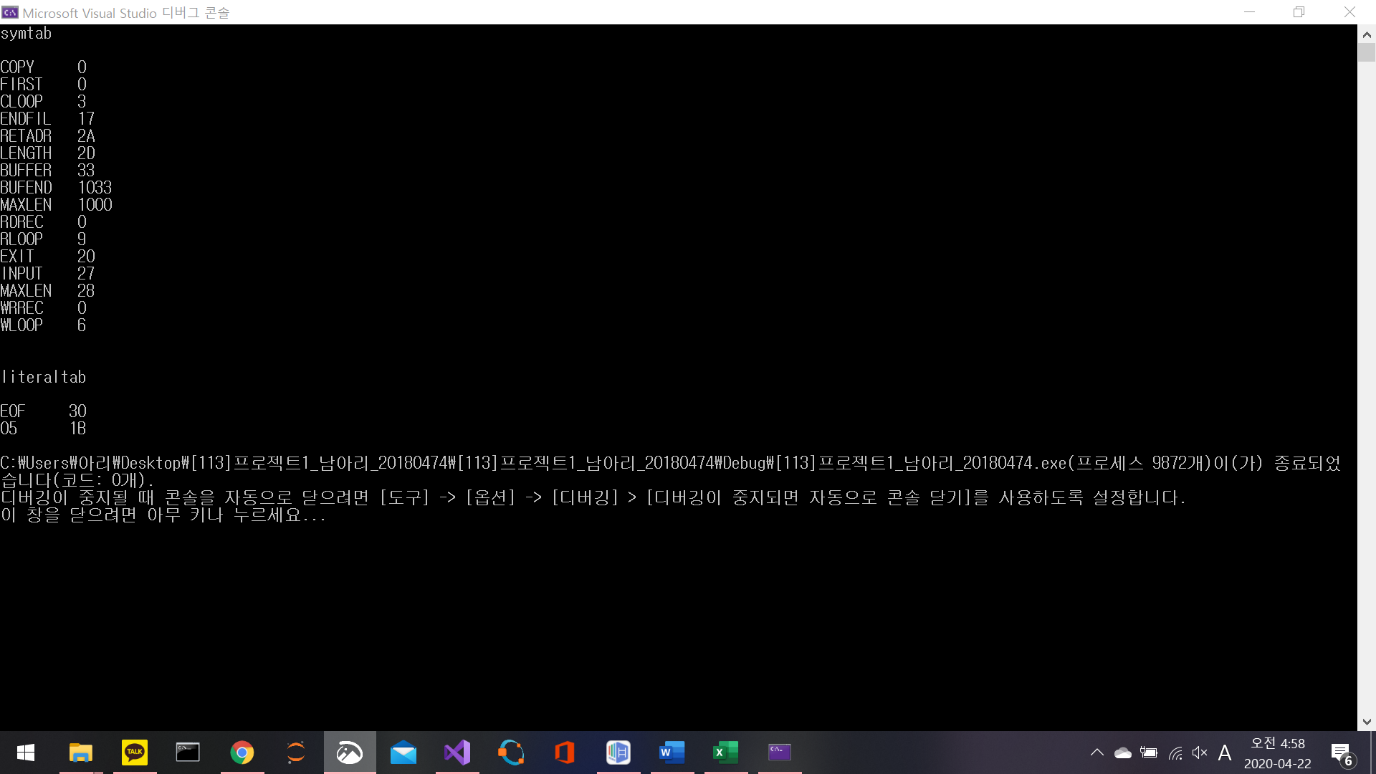
**(3) 구현 화면**

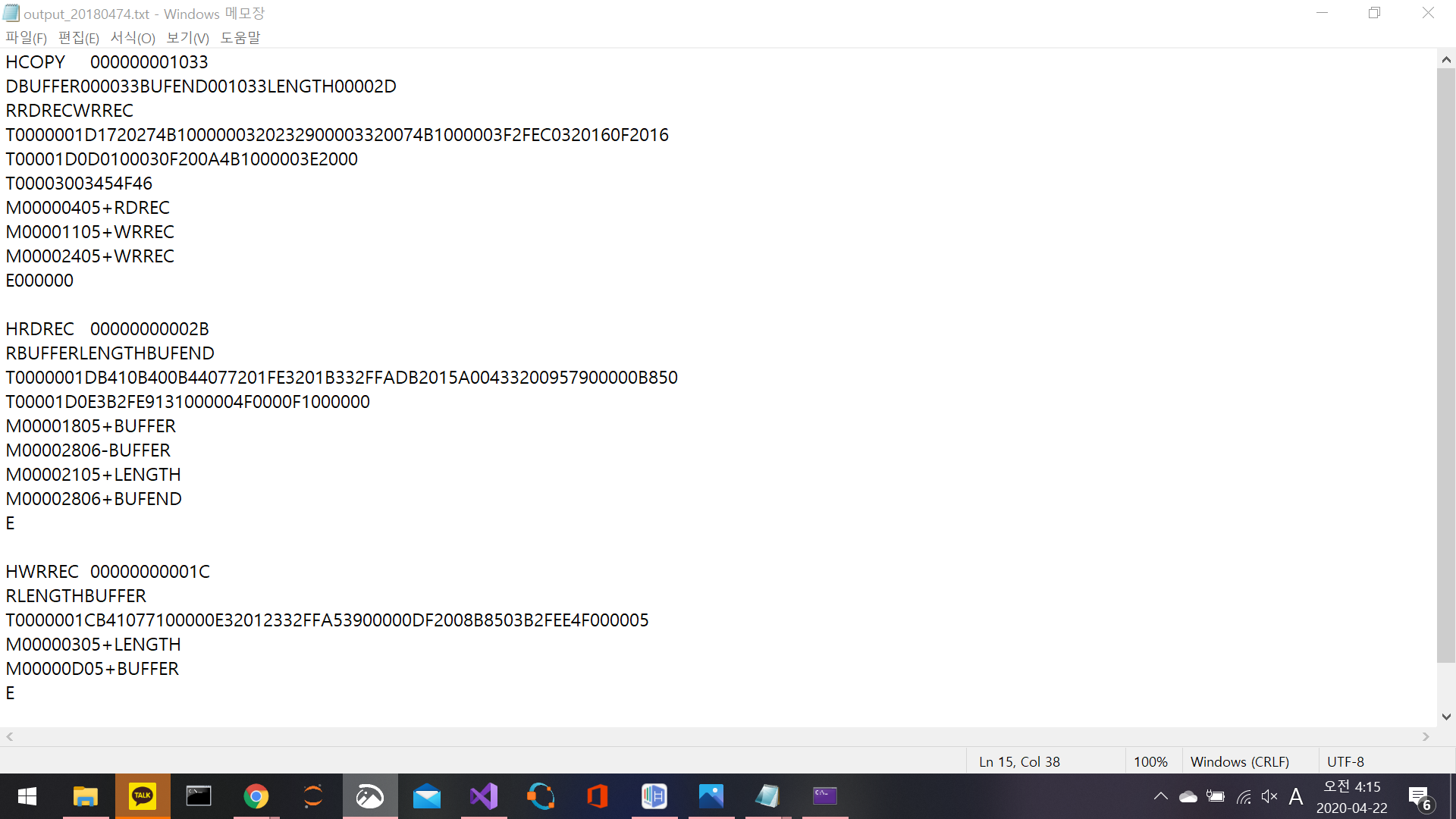
1. **data파일 일부**



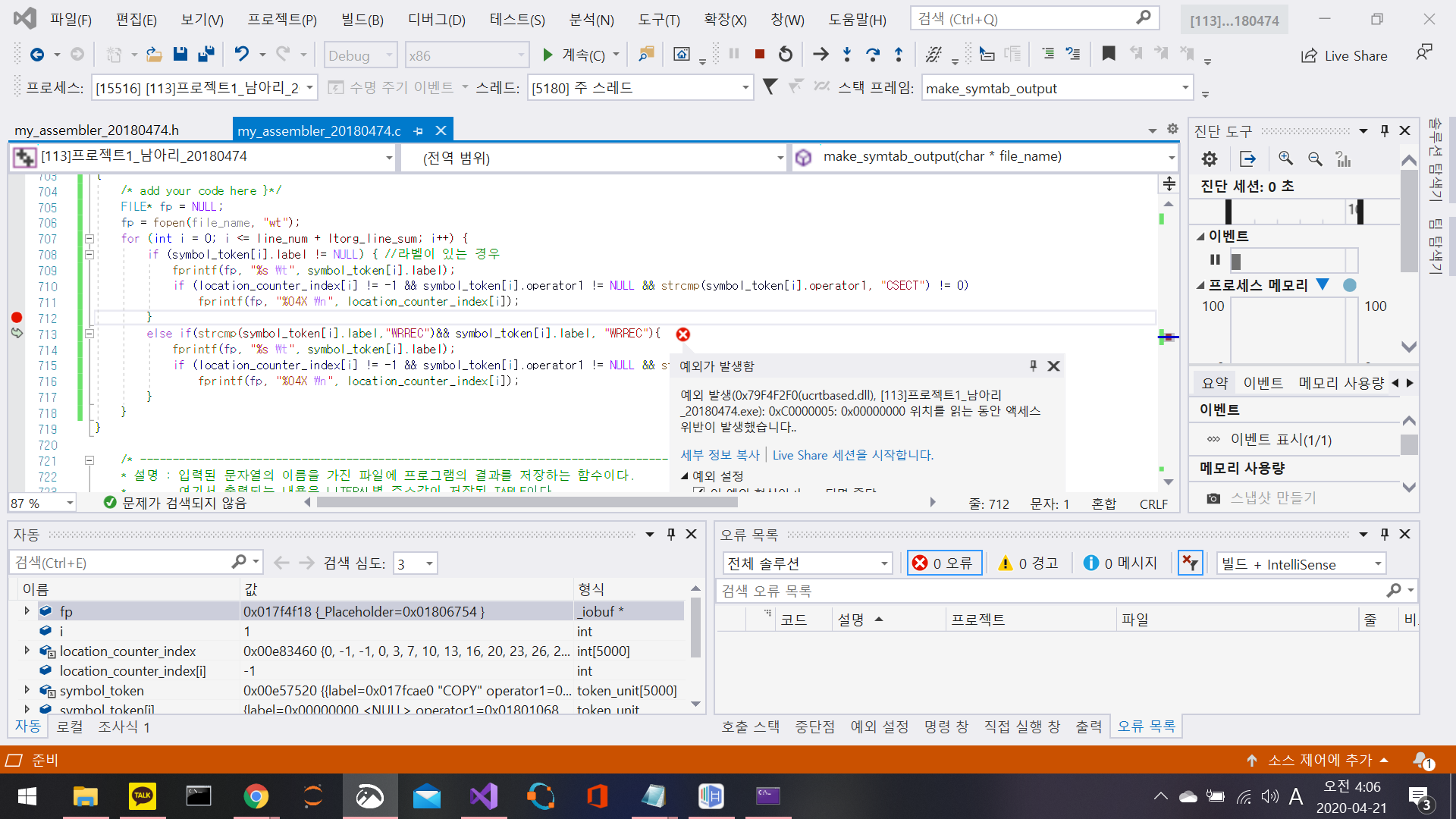
1. **3-2. 소스파일 (input)**



1. **콘솔화면 (output) – symtab, litertab**
2. **파일 화면(output) – object code**

****

1. **디버깅 화면**

****

**5장. 기대효과 및 결론, 고찰**

이번 프로젝트에서는 과제5에서 소스코드를 파싱하고 토큰 테이블을 만들었던 것에 이어서, 입력 받은 소스파일을 오브젝트 코드로 만드는 어셈블러를 구현하였다. 그리고 심볼 테이블, 리터럴 테이블을 만들어 화면에 출력하는 작업을 지난 과제에 이어 추가적으로 진행하였다.

이번 프로젝트는 기본적인 SIC/XE머신이 아니라 Control Section SIC/XE 머신을 구현하는 것이었기 때문에, 메모리 주소 연산, 프로그램 주소 연산에 조금 더 신경을 써야했다. 그래서 PASS2에서 각 명령어의 형식과 target address를 고려하여 프로그램 주소를 생성하는 것에 큰 어려움이 있었다. 물론 프로젝트의 모든 작업이 매우 힘들었지만, 1형식은 오퍼레이터만 처리하면 되었기 때문에 다른 작업에 비해 비교적 시간이 덜 걸렸다. 하지만 2,3,4형식 연산 부분을 구현할 때는 무척 하드했다.

특히 shift 연산을 떠올리는 것이 힘들었고, opcode+ni 와 pe + program address를 진행하는 과정에서 많은 시행착오를 겪었다. Addressing 방식에 따라 ni의 값을 구분하여 프로그램을 짜는 것이 머릿속에 잘 잡히지 않아 구현하는데 어려움을 겪었고, opcode에 ni를 어떻게 연산하여 첫번째 8bit를 생성해야 할지 많은 고민을 하였다. 그리고 처음에는 pe와 disp공간을 한번에 shift하여 공간을 할당하고 값을 넣으려 했는데, 연산이 복잡해지고 각각의 경우마다 비슷한 작업이 반복되어 프로그램 길이만 늘어나는 결과를 초래했다. 그래서 방향을 다시 잡는 시행착오를 겪었다.

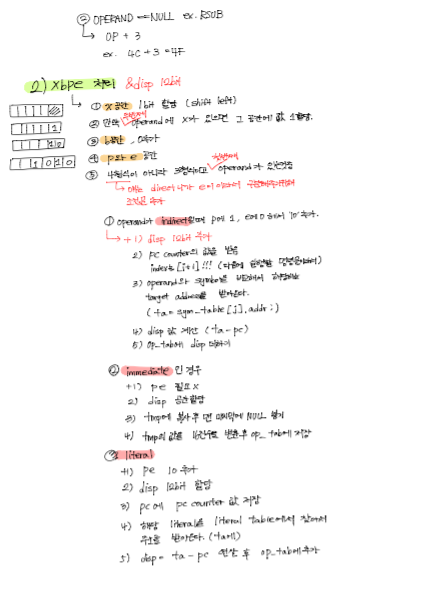
또한 기본 함수, 구조체, 변수 등의 이름들이 익숙하지 않아 소스파일과 헤더파일을 왔다갔다하면서 파악하는 것에도 시간이 한참 걸렸다. 무엇보다, c언어 포인터까지만 수박 겉핥기로 훑은 본인이 이 과제를 수행하기에는 너무 많은 시간이 들었다. 특히 처음 과제5를 받았을 때는 대체 무엇을 해야 하는지조차 파악하지 못했으며, fprinf, strtok 등의 사용에 익숙하지 않아서 프로그램을 구현하기까지 많은 어려움이 있었다

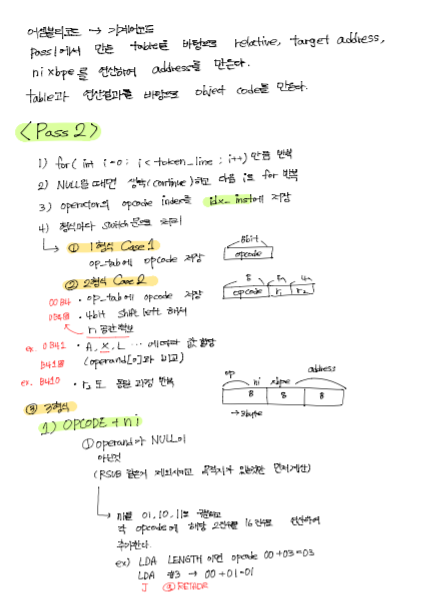
그리고 초반에는 불필요한 변수를 최대한 적게 사용하고 되도록 깔끔하게 프로그램을 작성하기 위해 노력했으나 계속되는 시행착오와 빌드 실패로 인해 ‘구현만이라도 성공하자’라는 마음으로 임했더니 쓸데없는 변수와 코드가 늘어나 프로그램이 헤비해졌다.

이번 프로젝트를 통해, C프로그래밍 능력을 함양할 수 있었으며, 내용이 섞여서 헷갈렸던 Program Block, Control Section 같은 내용들을 정리하는 시간을 갖을 수 있었다. 또한 개념이 잘 잡히지 않아서 정말 매번 헷갈렸던 LTORG, PASS1과 PASS2의 역할, 오브젝트 코드에 대한 개념들을 정립할 수 있었다.

사실 지난 과제 7을 할 때, Control Section의 주소 계산에 어려움을 겪었다. 그런데 스스로 계산하기도 힘들어 했던 것을 컴퓨터가 하도록 프로그래밍 하려니, 머리가 터지는 줄 알았다. 그러나 이 과정을 통해 addressing 방법에 대해서 확실하게 개념을 잡을 수 있었다. 또한 SIC/XE 머신의 구조를 좀 더 깊이 있게 생각해보는 시간을 갖을 수 있었다. 프로젝트 수행 전만해도, SIC/XE의 구조를 어느정도 이해했다고 생각했지만, 프로젝트를 진행하면서 구조의 이해는커녕, 과제를 위해 계산만 로봇처럼 해왔다는 것을 깨달을 수 있었다.

그래서 이 과제를 통해 스스로의 문제점을 파악하여 좀 더 깊이 있는 공부를 할 수 있었으며 머신의 구조를 이해할 수 있었다. 또한 소스코드를 오브젝트코드로 만드는 과정과 PASS1 PASS2로 이루어진 어셈블러의 동작과정을 이해할 수 있었다.

(아래의 사진은 프로젝트를 진행하며 동작과정에 대해 공부한 내용의 일부입니다.)



**+) 출처**

<구조체>

<https://dojang.io/mod/page/view.php?id=408>

<https://m.post.naver.com/viewer/postView.nhn?volumeNo=17986958&memberNo=21815&searchKeyword=%EA%B5%AC%EC%A1%B0%EC%B2%B4&searchRank=3>

<https://dpdpwl.tistory.com/65>

<shift 연산>

<https://dojang.io/mod/page/view.php?id=174>

<파일입출력>

<https://ra2kstar.tistory.com/53>