시스템 프로그래밍

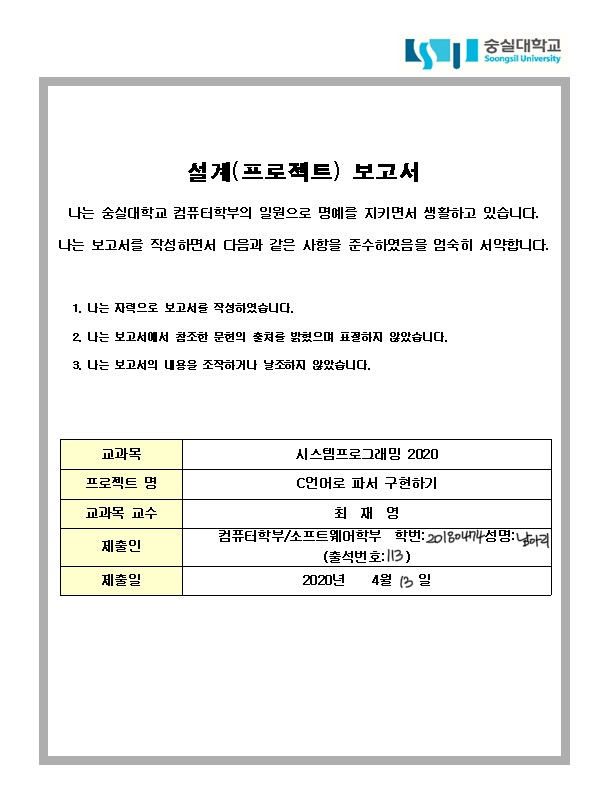
-과제 5번 OPCODE 출력하기-

CS시스템프로그래밍 월수 12시반

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

\*소스코드는 압축 파일로 첨부했습니다.

**1. 과제 진행 동기와 목적**

**1-1. 진행 동기**

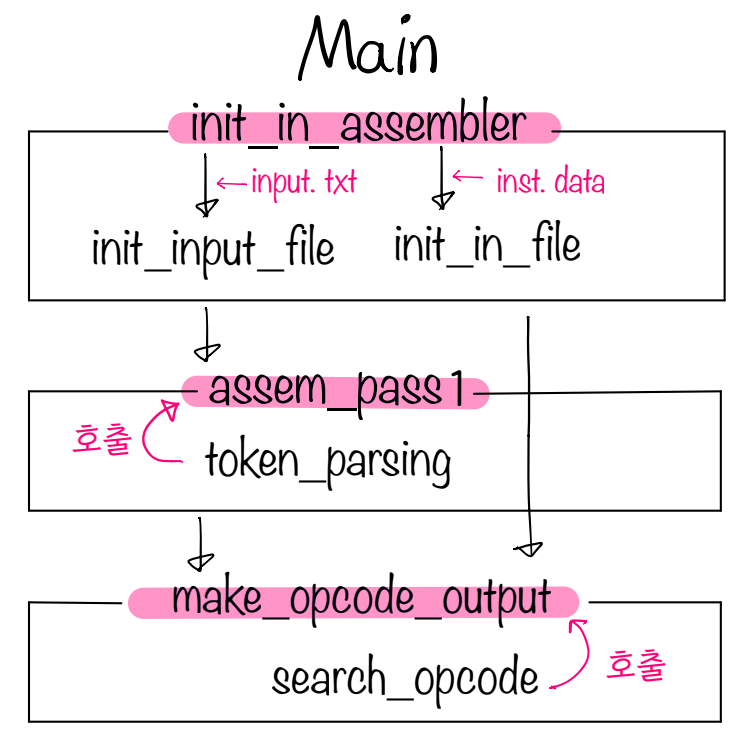
* 첫 번째 프로젝트인 SIC/XE 어셈블러를 개발하기 전, 프로젝트를 위한 기본 작업을 수행하면서 다가올 프로젝트를 준비한다.
* 과제를 수행함으로써 프로젝트의 방향을 이해할 수 있으며, SIC/XE머신의 기본적인 구조와 동작 과정을 이해한다.
* 과제를 진행하며 겪었던 여러 문제들을 통해 프로젝트를 진행하면서 겪게 될 시행착오를 줄인다.

**2-1. 진행 목적**

추후 진행하게 될 프로젝트를 위한 기본 작업을 준비하기 위함이다. 기본 작업으로는, 먼저 명령어의 OPCODE가 적힌 DATA파일과 어셈블리 할 소스파일을 입력 받는 작업을 한다. 입력 받은 파일을 각각의 테이블로 만들고 파싱하여 토큰테이블을 만든다. 또한 입력 받은 소스코드의 OPERATOR를 명령어 파일에서 찾아, 알맞은 OPCODE를 파일에 출력시킨다. 이를 통해 SIC/XE머신의 기본적인 구조와 입출력, 파싱, OPCODE 매핑 등의 동작 과정을 이해할 수 있다.

**2. 시스템 설계 내용**

**(1) 설계의 이미지화**



**(2) 전체적인 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 테이블과 search\_opcode에서 호출한 인덱스를 바탕으로 format에 따라 스위치문으로 구분한다. 그리고 각각의 format에 맞게 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.

**3. 시스템 구현**

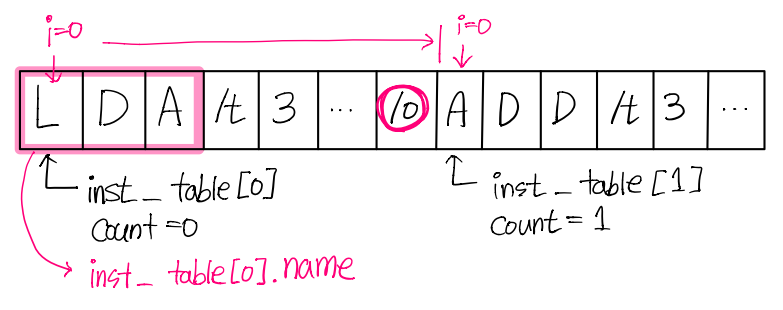
**(1) 각 모듈의 구현**

* 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

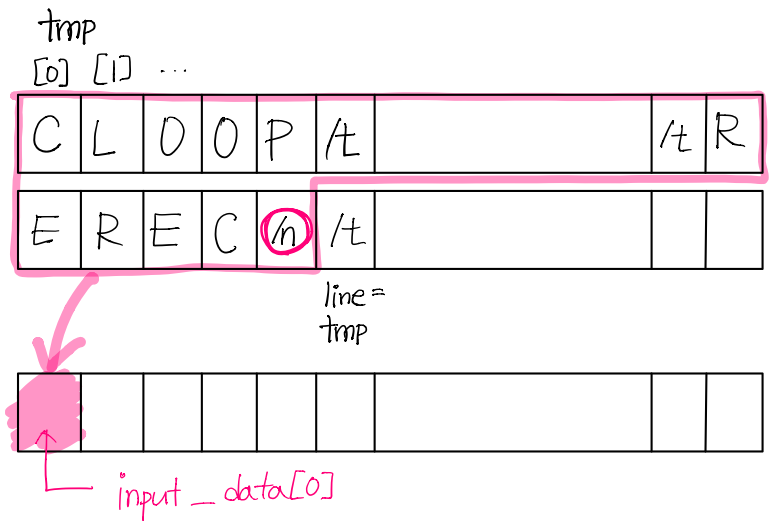
다음과 같이 저장되어 TABLE이 만들어질 것이라고 생각할 수 있다.

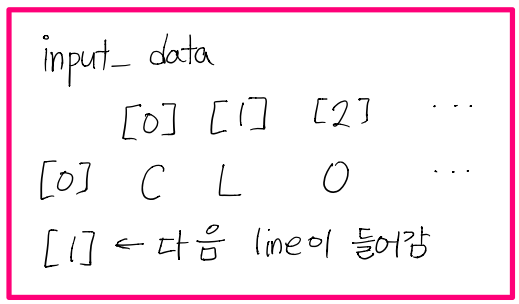


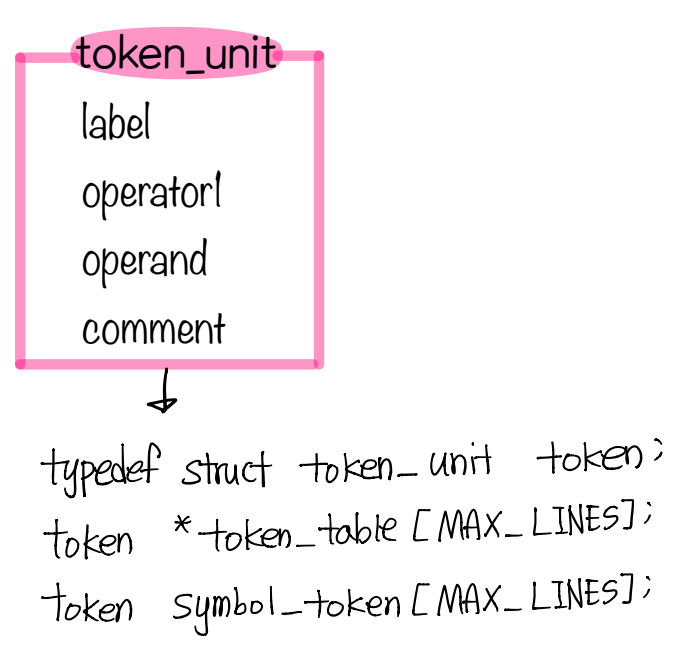
* input\_data
* char \*input\_data[MAX\_LINES];
* init\_input\_file에서 사용된다.
* 임시 저장용 tmp[100]에 개행 문자가 나오기 전까지 소스코드 input.txt 한 줄을 입력 받는다.
* tmp에 저장된 string을 input\_data[instruction\_cnt]에 저장한다.
* 라인단위로 관리된다.
* Input.txt의 일부로 저장방식을 이미지화 한다면, 다음과 같이 생각할 수 있다.

CLOOP +JSUB REREC

LDA LENGTH





* symbol\_token
* 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];

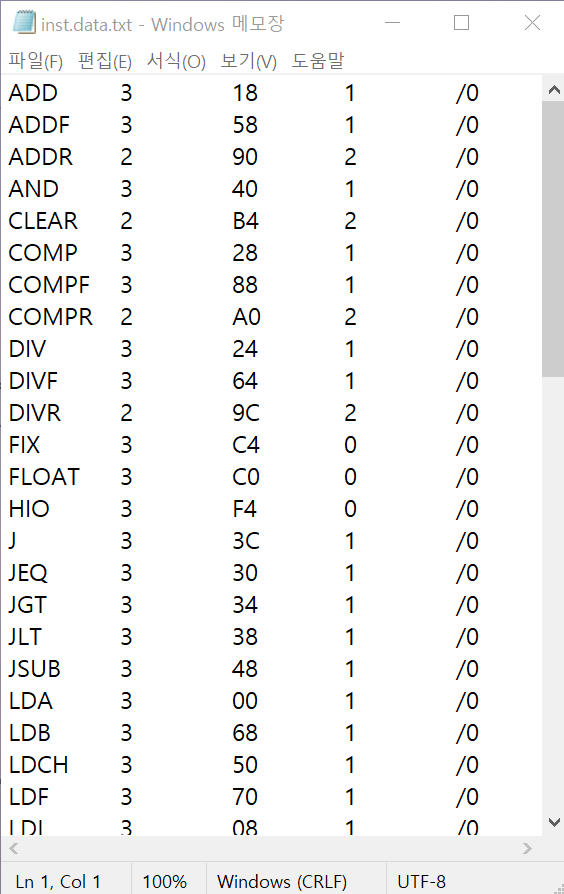
token symbol\_token[MAX\_LINES];

* 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.

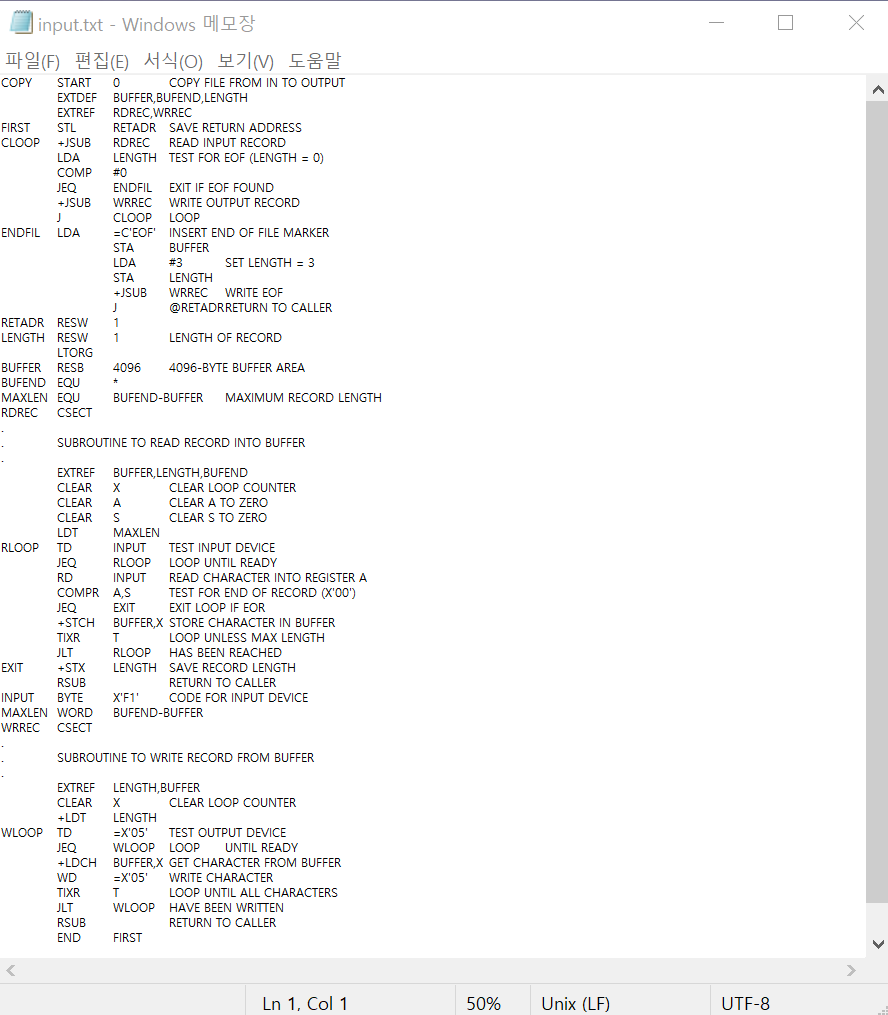
**(2) 전제구현 내용**

pass1에서 만들어진 테이블과 search\_opcode에서 호출한 인덱스를 바탕으로 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. Bit를 추가하여 프로그램 주소를 만드는 과정은 operator의 opcode,ni,xbpe,

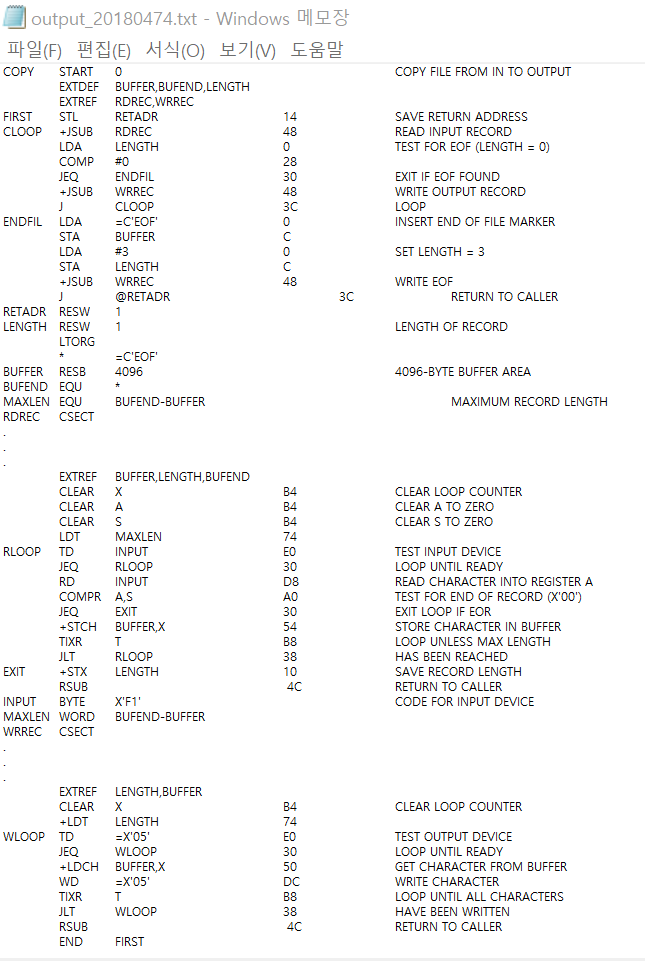
1. **data파일 일부**



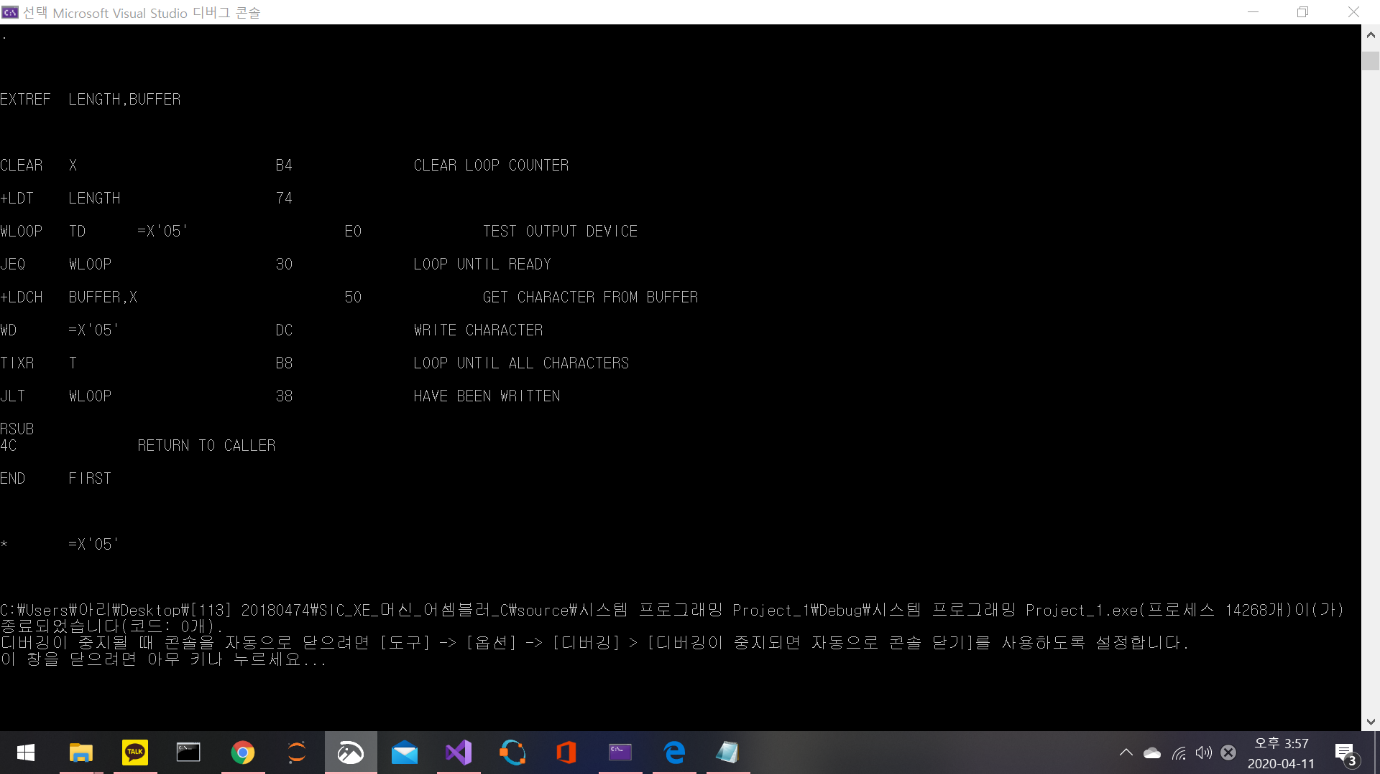
1. **3-2. 소스파일 (input)**



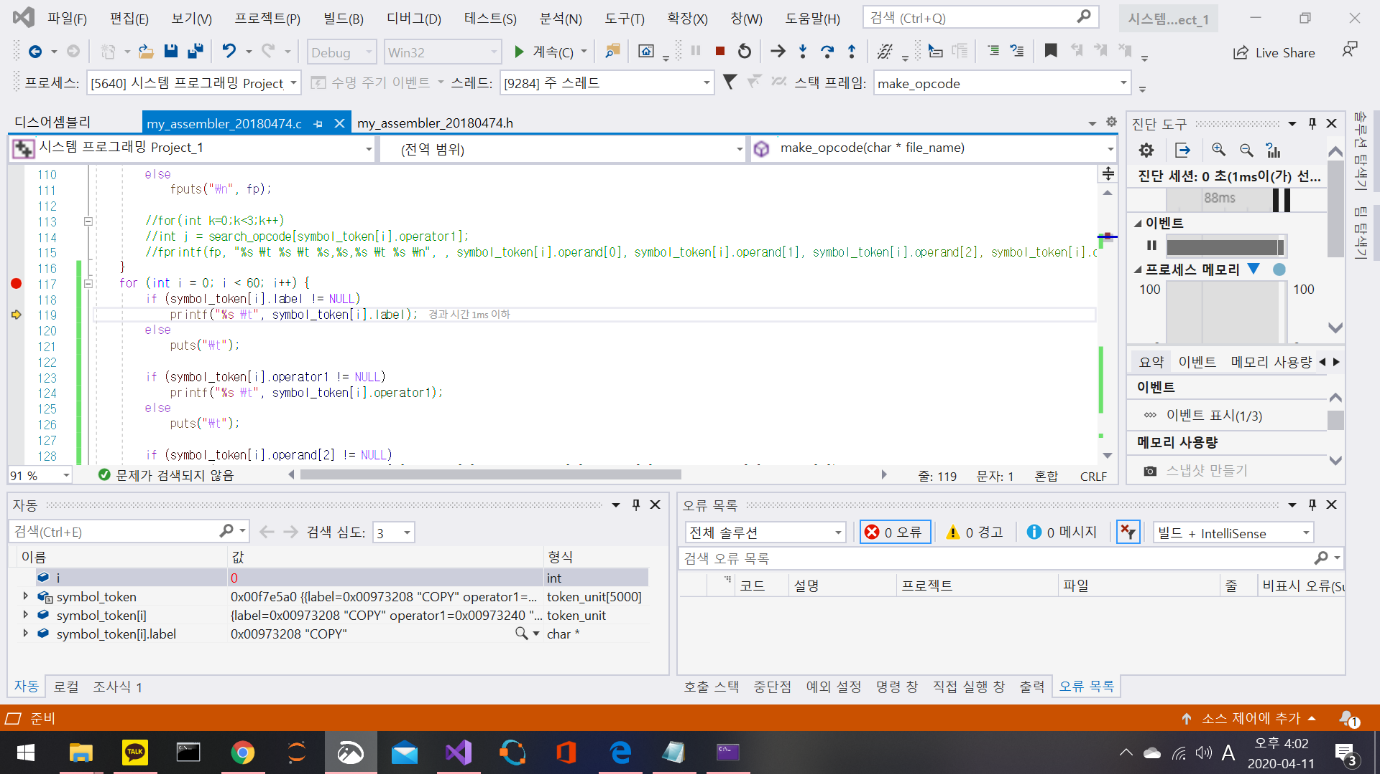
1. **구현화면 (output)**



메모장 생성 화면 캡쳐



콘솔 디버깅 화면



디버깅 화면 캡쳐

**4. 결과 및 고찰**

이번 과제5는 파일을 입력 받아 테이블을 만들고, 파싱 프로그램을 만들어서 토큰 테이블을 만든 뒤, opcode를 mapping 시키는 과제였다. 그래서 기본 소스코드에 있는 함수들을 바탕으로 입력 함수 두 개와, 파싱 프로그램, pass1, opcode 출력 함수를 만들었고, 결과는 다행히 제대로 출력되었다.

그러나 출력에서 opcode 탭을 몇 번 정도 해야 균일하게 일렬로 출력되는지 알아내지 못했다. 그래서 중간중간 열을 못 맞추고 튄 opcode 부분이 있다는 점이 아쉽다. 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 과제를 받았을 때는 대체 무엇을 해야 하는지조차 파악하지 못했으며, fprinf, strtok 등을 처음 사용해보아서 프로그램을 구현하기까지 많은 어려움이 있었다.

물론 이후의 프로젝트는 더 어려울 것이기에, 이 과제 이후에 pass2를 구현할 생각을 하면 앞이 캄캄하다. 그러나 이 과제를 하면서 강의 내용을 한 번 더 복습하고, 머신의 구조를 좀 더 깊이있게 생각해보는 시간을 갖을 수 있었다. 과제 수행 전만해도, SIC/XE의 구조를 어느정도 이해했다고 생각했지만, 과제를 진행하면서 구조의 이해는 커녕, 과제를 위해 계산만 로봇처럼 해왔다는 것을 깨달을 수 있었다.

이 과제를 통해 스스로의 문제점을 파악하고 다시 공부를 하니 좀 더 깊이 있는 공부를 할 수 있었고 SIC/XE 머신의 구조를 대략 이해할 수 있었다. 물론 아직 완벽하게는 아니지만 추후 프로젝트를 진행하며 이해도를 더 높일 수 있을 것 같다.