시스템 프로그래밍

Project #1-B

Java version

|  |  |
| --- | --- |
| 소속학과 | 컴퓨터학부 |
| 학번 | 20122366 |
| 성명 | 김홍석 |
| 출석번호 | 139 |
| 담당교수명 | 최재영 교수님 |
| 일자 | 2017-05-23 |

목차

1. 동기 / 목적
2. 설계 / 구현 아이디어
3. 소스 코드
4. 수행 결과
5. 결론 및 보충할 점
6. 동기 / 목적

어셈블리어로 작성된 소스코드를 object 코드로 변환하기 위해선 일련의 과정이 필요하다. 그 중 소스코드를 입력으로 받아 파싱하고 OPCODE를 매핑시키는 부분의 구현은 저번 과제에서 수행하였고, 본 과제에서는 2 Pass assembler 의 전체적인 구현이 목적이다. 이를 통해 SIC/XE 머신의 동작과정에 대해 이해한다. 그리고 자바로 구현을 함으로써 자바 언어에 대한 이해와 동작과정에 대해서도 추가로 습득한다.

1. 설계 / 구현 아이디어

Pass 1과 Pass 2 에서 생성한 정보를 통해서 최종 output인 오브젝트 프로그램 생성

Pass 1에서 생성한 정보를 통해 각 토큰의 object code 생성,

Locctr을 이용해 SYMTAB , LITTAB 생성 및 주소 할당.

* 1. Pass 1

Pass 1 에서 각 라인의 locctr 을 구하며, 이를 이용해 그 라인에서 라벨이 존재하면 SYMTAB을 만든다. 또한 LITTAB 구조체를 하나 생성해서 리터럴에 대한 관리도 또한 해준다.

* 1. Pass 2

다시 한번 각 라인을 돌면서 각 라인에 대한 object code를 구한다. 이 때, Pass1에서 생성했던 각 테이블을 이용해서 object code를 생성한다. Object code 계산을 할 때 nixbpe라는 1바이트 짜리 변수를 사용하며 상위 2비트(n,i) 와 하위 4비트(xbpe)를 비트연산으로 사용한다. 이렇게 만들어진 object code는 각 라인에 저장되며 , 추후 object program을 만들 때 필요한 M레코드에 관한 정보도 미리 저장해둔다. 그리고 과제의 명세에서 요구한 화면 출력을 pass2 에서 하도록 구현하였다.

* 1. Make\_objectcode\_output

Pass 2에서 만들어놓은 object code 정보들을 파일에 쓴다. H, T, M 레코드 등을 작성하도록 구현했다. 특히 T 레코드의 경우 읽는 object code가 30바이트를 넘으면 안되므로 30 바이트를 넘어가면 줄을 바꾼 후 출력하도록 구현하였고, M 레코드의 경우 pass2 에서 저장한 M레코드가 필요한 리인의 인덱스를 이용해 구현하였다.

1. 소스 코드
   1. Header.java

**class** inst\_unit{

String operator;

String type;

String opcode;

**int** operandAmount;

}

**class** token\_unit{

String label="";

String operator="";

String operand[] = **new** String[3];

String comment="";

**char** nixbpe=0; // 추후 프로젝트에서 사용된다. 1 2 4 8 16 32

//추가

**int** current\_locctr;

**int** pcValue;

String total\_operand="";

**int** object=0;

**int** hasobjectcode=0;

}

**class** symbol\_unit {

String symbol;

**int** addr;

**int** section;

**char** sign;

}

**class** literal\_unit {

String literal;

**int** addr;

**int** section;

**int** object;

}

**class** modification\_unit {

**int** index;

**int** section;

}

2. SICAssemblerTester.java

**public** **class** SICAssemblerTester {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

SICAssembler sic = **new** SICAssembler();

}

}

3. SICAssembler.java

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.io.PrintWriter;

**import** java.util.Scanner;

**import** java.util.StringTokenizer;

**public** **class** SICAssembler {

**final** **int** MAX\_INST = 256;

**final** **int** MAX\_LINES = 5000;

**final** **int** MAX\_OPERAND = 3;

**final** **int** N = 32;

**final** **int** I = 16;

**final** **int** X = 8;

**final** **int** B = 4;

**final** **int** P = 2;

**final** **int** E = 1;

//////////////////////////////////////////////

inst\_unit inst\_table[] = **new** inst\_unit[MAX\_INST];

**int** inst\_index = 0;

String input\_data[] = **new** String[MAX\_LINES];

**int** line\_num;

**int** label\_num;

token\_unit token\_table[] = **new** token\_unit[MAX\_LINES];

**int** token\_line = 0;

symbol\_unit sym\_table[] = **new** symbol\_unit[MAX\_LINES];

literal\_unit lit\_table[] = **new** literal\_unit[100];

modification\_unit mod\_table[] = **new** modification\_unit[100];

**int** hasliteral[] = **new** **int**[10];

**int** modification\_index[] = **new** **int**[100];

**int** modification\_line = 0;

**int** symbol\_line = -1;

**int** literal\_line = -1;

**int** section\_num = 0;

**int** locctr;

**int** section\_length[] = **new** **int**[10];

String input\_file;

String output\_file;

////////////////////////////////////////

**public** SICAssembler()

{

**for**(**int** i = 0 ; i<100;i++) //modification unit 인스턴스 할당

mod\_table[i] = **new** modification\_unit();

**if**(init\_my\_assembler() < 0 )

System.***out***.println("Error Occured in init\_my\_assembler");

**if**(assem\_pass1() < 0 )

System.***out***.println("Error Occured in pass1");

**if**(assem\_pass2() < 0 )

System.***out***.println("Error Occured in pass2");

make\_objectcode\_output("output\_139.txt");

}

/\* ----------------------------------------------------------------------------------

\* 설명 : 프로그램 초기화를 위한 자료구조 생성 및 파일을 읽는 함수이다.

\* 매개 : 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()

{

**int** result=0;

**if** ((result = init\_inst\_file("inst.data")) < 0)

**return** -1;

**if** ((result = init\_input\_file("input.txt")) < 0)

**return** -1;

**return** result;

}

/\* ----------------------------------------------------------------------------------

\* 설명 : 머신을 위한 기계 코드목록 파일을 읽어 기계어 목록 테이블(inst\_table)을

\* 생성하는 함수이다.

\* 매개 : 기계어 목록 파일

\* 반환 : 정상종료 = 0 , 에러 < 0

\* 주의 : 기계어 목록파일 형식은 자유롭게 구현한다. 예시는 다음과 같다.

\*

\* ===============================================================================

\* | 이름 | 형식 | 기계어 코드 | 오퍼랜드의 갯수 | NULL|

\* ===============================================================================

\*

\* ----------------------------------------------------------------------------------

\*/

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

{

**int** errno = 0;

**int** i = 0;

**try**

{

Scanner scanner = **new** Scanner(**new** File(inst\_file));

**while**(scanner.hasNext()){

StringTokenizer st = **new** StringTokenizer(scanner.nextLine(),"\t\n"); //스캐너 클래스를 이용해 한줄씩 받아옴 + 탭,개행으로 토크나이즈

inst\_table[i] = **new** inst\_unit();

inst\_table[i].operator = st.nextToken();

inst\_table[i].type = st.nextToken();

inst\_table[i].opcode = st.nextToken();

inst\_table[i].operandAmount = Integer.*parseInt*(st.nextToken());

inst\_index = i; //0부터 갯수 시작

i++;

}

scanner.close();

}

**catch** (FileNotFoundException e) { //파일이 없으면 try catch 로 예외처리

System.***out***.println("The inst\_file doesn't exist");

e.printStackTrace();

**return** errno = -1;

}

**return** errno;

}

/\* ----------------------------------------------------------------------------------

\* 설명 : 어셈블리 할 소스코드를 읽어오는 함수이다. 읽어오면서 바로 token 을 분리하여 테이블에 넣어준다.

\* 매개 : 어셈블리할 소스파일명

\* 반환 : 정상종료 = 0 , 에러 < 0

\* 주의 :

\*

\* ----------------------------------------------------------------------------------

\*/

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

{

**int** errno = 0;

**int** i = 0;

**try**

{

Scanner scanner = **new** Scanner(**new** File(input\_file));

**while**(scanner.hasNext()){

String line = scanner.nextLine();

**char** first = line.charAt(0);

StringTokenizer st = **new** StringTokenizer(line,"\t\n");

line\_num = i;

token\_table[i] = **new** token\_unit();

token\_table[i].label ="";

token\_table[i].operator ="";

token\_table[i].operand[0] =" ";

token\_table[i].comment ="";

**if**(first == '\t')

token\_table[i].label ="";

**else**

token\_table[i].label =st.nextToken();

**if**(st.hasMoreTokens())

token\_table[i].operator =st.nextToken();

**if**(st.hasMoreTokens())

token\_table[i].operand[0] =st.nextToken();

token\_table[i].total\_operand = token\_table[i].operand[0];

token\_table[i].operand[1] = " ";

token\_table[i].operand[2] = " ";

StringTokenizer st2 = **new** StringTokenizer(token\_table[i].operand[0],",\t");

**for**(**int** j=0;j<3;j++)

{

**if**(st2.hasMoreTokens())

token\_table[i].operand[j] = st2.nextToken();

}

**if**(st.hasMoreTokens())

token\_table[i].comment =st.nextToken();

token\_line = i; //0부터 시작

i++;

}

scanner.close();

}

**catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

System.***out***.println("The input\_file doesn't exist");

e.printStackTrace();

**return** errno = -1;

}

**return** errno;

}

/\* ----------------------------------------------------------------------------------

\* 설명 : 입력 문자열이 기계어 코드인지를 검사하는 함수이다.

\* 매개 : 토큰 단위로 구분된 문자열

\* 반환 : 정상종료 = 기계어 테이블 인덱스, 에러 < 0

\* 주의 :

\*

\* ----------------------------------------------------------------------------------

\*/

**int** search\_opcode(String str) //str을 비교값으로 해서 inst\_table을 모두 뒤져서 index를 찾음. 없으면 -1 리턴

{

String compare;

**if** (str.charAt(0) == '+') //4형식 처리 +를 무시

compare = str.substring(1);

**else**

compare = str;

**for** (**int** i = 0; i <= inst\_index; i++)

{

**if** (compare.equals(inst\_table[i].operator))

**return** i;

}

**return** -1;

}

/\* ----------------------------------------------------------------------------------

\* 설명 : 심볼들을 비교하여 없으면 테이블에 추가하고 있으면 에러발생

\* 매개 : 비교할 symbol

\* 반환 : symbol이 없어서 추가한 후 종료 = 0, 이미 symbol이 존재하면 -1

\* -----------------------------------------------------------------------------------

\*/

**int** manage\_symbol(String str)

{

**for** (**int** i = 0; i <= symbol\_line; i++)

{

**if** ((str.equals(sym\_table[i].symbol)) && section\_num == sym\_table[i].section) //symbol이 있으면서 그 symbol이 같은섹션일때

**return** -1;

}

//symbol이 없으면 symtab 에 추가, line 수 증가

sym\_table[symbol\_line + 1] = **new** symbol\_unit(); //C와 달리 인스턴스를 생성해줘야 하므로 생성

sym\_table[symbol\_line + 1].symbol= str;

sym\_table[symbol\_line + 1].addr = locctr;

sym\_table[symbol\_line + 1].section = section\_num;

sym\_table[symbol\_line + 1].sign = '+';

symbol\_line++;

**return** 0;

}

/\* ----------------------------------------------------------------------------------

\* 설명 : 리터럴들을 비교하여 없으면 테이블에 추가하고 있으면 에러발생

\* 매개 : 비교할 literal

\* 반환 : literal이 없어서 추가한 후 종료 = 0, 이미 literal이 존재하면 -1

\* -----------------------------------------------------------------------------------

\*/

**int** manage\_literal(String str)

{

**for** (**int** i = 0; i <= literal\_line; i++)

{

**if** (str.equals(lit\_table[i].literal))// && section\_num == lit\_table[i].section) //literal이 있으면서 그 literal이 같은섹션일때

**return** -1;

}

//literal이 없으면 littab 에 추가, line 수 증가

lit\_table[literal\_line+1] = **new** literal\_unit(); //C와 달리 인스턴스를 생성해줘야 하므로 생성

lit\_table[literal\_line + 1].literal = str;

lit\_table[literal\_line + 1].addr = locctr;

lit\_table[literal\_line + 1].object = 0;

lit\_table[literal\_line + 1].section = section\_num;

hasliteral[section\_num] = 1; // 각 섹션에서 리터럴이 쓰이는 지 없는지 판별에 도움을 주는 배열 1-> 리터럴이 섹션에 있음, 2-> 섹션에 없음

literal\_line++;

**return** 0;

}

/\* ----------------------------------------------------------------------------------

\* 설명 : 심볼들을 비교하여 있으면 주소 반환, 다른 경우 에러 숫자 반환

\* 매개 : 비교할 문자열

\* 반환 : symbol이 존재하고, 같은섹션에 있을 때 = 심볼에 대응되는 주소값 반환,

symbol이 존재하나 다른 섹션에 있을때 = -2 반환,

symbol이 존재 하지않다면 -1로 에러 반환

\* -----------------------------------------------------------------------------------

\*/

**int** search\_symbol(String str)

{

**int** addr = -1;

**for** (**int** i = 0; i <= symbol\_line; i++)

{

**if** (str.equals(sym\_table[i].symbol))

{

**if** (section\_num == sym\_table[i].section) //symbol이 있으면서 그 symbol이 같은섹션일때

**return** sym\_table[i].addr;

**else** //symbol이 있으나 같은 섹션이 아닐때는 -2 반환

addr = -2;

}

}

**return** addr;

}

/\* ----------------------------------------------------------------------------------

\* 설명 : 문자열과 리터럴 테이블의 값들 비교

\* 매개 : 비교할 문자열

\* 반환 : literal이 존재함 = 그 리터럴의 주소값 반환,

이미 literal이 존재하지 않으면 -1로 에러 반환

\* -----------------------------------------------------------------------------------

\*/

**int** search\_literal(String str)

{

**for** (**int** i = 0; i <= literal\_line; i++)

{

**if** (str.equals(lit\_table[i].literal))

**return** lit\_table[i].addr;

}

**return** -1;

}

/\* ----------------------------------------------------------------------------------

\* 설명 : 입력 문자열의 리터럴 인덱스를 찾는 함수이다.

\* 매개 : 비교할 문자열

\* 반환 : 정상종료 = 리터럴 테이블 인덱스, 에러 < 0

\* ----------------------------------------------------------------------------------

\*/

**int** search\_literal\_index(String str) //str을 비교값으로 해서 inst\_table을 모두 뒤져서 index를 찾음. 없으면 -1 리턴

{

**for** (**int** i = 0; i <= literal\_line; i++)

{

**if** (str.equals(lit\_table[i].literal))

**return** i;

}

**return** -1;

}

/\* ----------------------------------------------------------------------------------

\* 설명 : 입력 문자열의 심볼 인덱스를 찾는 함수이다.

\* 매개 : 비교할 문자열

\* 반환 : 정상종료 = 심볼 테이블 인덱스, 에러 < 0

\* ----------------------------------------------------------------------------------

\*/

**int** search\_symbol\_index(String str) //str을 비교값으로 해서 inst\_table을 모두 뒤져서 index를 찾음. 없으면 -1 리턴

{

**for** (**int** i = 0; i <= symbol\_line; i++)

{

**if** (str.equals(sym\_table[i].symbol))

**return** i;

}

**return** -1;

}

/\* ----------------------------------------------------------------------------------

\* 설명 : 어셈블리 코드를 위한 패스1과정을 수행하는 함수이다.

\* 패스1에서는..

\* 1. 프로그램 소스를 스캔하여 해당하는 토큰단위로 분리하여 프로그램 라인별 토큰

\* 테이블을 생성한다.

\*

\* 매개 : 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.

\*

\* -----------------------------------------------------------------------------------

\*/

**int** assem\_pass1()

{

**int** index = 0;

**int** ltorg\_line = 0;

**for** (**int** i = 0; i <= line\_num; i++)

{

**if** (token\_table[i].operator.equals("START")) //START 면 LOCCTR 을 그 인자로 초기화

{

locctr = Integer.*valueOf*(token\_table[i].operand[0],16);

}

**else** **if** (token\_table[i].label.equals( ".")) //주석 처리, 주석은 opcode를 찾을 필요가 없음.

**continue**;

token\_table[i].current\_locctr = locctr; //각 토큰에 locctr 기록

**if** (token\_table[i].operator.equals("CSECT")) //Control section이 바뀜 -> locctr 초기화, section\_num으로 배열 내에서 구분

{

section\_length[section\_num] = locctr; //각 섹션의 길이 저장해둠

locctr = 0;

section\_num++;

**continue**;

}

**if** (!token\_table[i].label.equals("")) // 라벨이 있으면 심볼 처리

**if** (manage\_symbol(token\_table[i].label) < 0)

**return** -1; //같은 섹션 내에 같은 이름의 라벨이 존재하면 에러 출력

**if** ((index = search\_opcode(token\_table[i].operator)) > -1) //opcode가 있을때, 이때만 pc값 기록, index로 테이블에 접근한다

{

token\_table[i].hasobjectcode = 1; //object코드를 생성하는 토큰임을 기록

**if** (inst\_table[index].type.equals("1")) // format 1 일 때

{

locctr = locctr + 1;

token\_table[i].pcValue = locctr;

}

**else** **if** (inst\_table[index].type.equals("2")) // format 2 일 때

{

locctr = locctr + 2;

token\_table[i].pcValue = locctr;

}

**else** **if** (inst\_table[index].type.equals("3/4")) // format 3/4 일 때 +여부로 locctr 증가되는 수 정해줌

{

**if** (token\_table[i].operator.charAt(0) == '+')

locctr = locctr + 4;

**else**

locctr = locctr + 3;

token\_table[i].pcValue = locctr;

}

}

**else** // opcode가 없는 directive 일 때

{

**if** (token\_table[i].operator.equals("WORD"))

{

token\_table[i].hasobjectcode = 1;

locctr = locctr + 3;

}

**else** **if** (token\_table[i].operator.equals("RESW"))

{

locctr = locctr + 3 \* Integer.*parseInt*(token\_table[i].total\_operand);

}

**else** **if** (token\_table[i].operator.equals("RESB"))

{

locctr = locctr + Integer.*parseInt*(token\_table[i].total\_operand);

}

**else** **if** (token\_table[i].operator.equals("BYTE")) //첫 글자가 X(16진수, 문자 2개당 1바이트), C(문자 1개당 1바이트) 일때 처리

{

token\_table[i].hasobjectcode = 1;

StringTokenizer st = **new** StringTokenizer(token\_table[i].total\_operand,"'");

String tok = st.nextToken();

**if** (tok.equals("C"))

{

tok = st.nextToken();

locctr = locctr + tok.length();

}

**else** **if** (tok.equals("X"))

{

tok = st.nextToken();

locctr = locctr + tok.length()/2;

}

}

**else** **if** (token\_table[i].operator.equals("LTORG"))

{

**for** (**int** j = ltorg\_line; j < i; j++)

{

**if** (token\_table[j].operand[0].charAt(0) == '=')

{

**if** (manage\_literal(token\_table[j].operand[0]) < 0) //이미 있는 리터럴을 무시

**continue**;

StringTokenizer st = **new** StringTokenizer(token\_table[j].operand[0],"'");

String tok = st.nextToken();

**if** (tok.equals("=C"))

{

tok = st.nextToken();

locctr = locctr + tok.length();

}

**else** **if** (tok.equals("=X"))

{

tok = st.nextToken();

locctr = locctr + tok.length()/2;

}

}

}

ltorg\_line = i; //ltorg 라인 관리. 이전 ltorg에서 나온 리터럴은 무시하도록 해야하기때문

}

**else** **if** (token\_table[i].operator.equals("END"))

{

**for** (**int** j = ltorg\_line; j < i; j++)

{

**if** (token\_table[j].operand[0].charAt(0) == '=')

{

**if** (manage\_literal(token\_table[j].operand[0]) < 0) //이미 있는 리터럴을 무시

**continue**;

StringTokenizer st = **new** StringTokenizer(token\_table[j].operand[0],"'");

String tok = st.nextToken();

**if** (tok.equals("=C"))

{

tok = st.nextToken();

locctr = locctr + tok.length();

}

**else** **if** (tok.equals("=X"))

{

tok = st.nextToken();

locctr = locctr + tok.length()/2;

}

}

}

section\_length[section\_num] = locctr;

ltorg\_line = i; //ltorg 라인 관리. 이전 ltorg에서 나온 리터럴은 무시하도록 해야하기때문

}

}

}

**return** 0;

}

/\* ----------------------------------------------------------------------------------

\* 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.

\* 매개 : 없음

\* 반환 : 정상종료 = 0, 에러발생 = < 0

\* 주의 :

\* -----------------------------------------------------------------------------------

\*/

**int** assem\_pass2()

{

**int** index = 0;

section\_num = 0;

**int** ltorg\_line = 0;

**for** (**int** i = 0; i <= line\_num; i++)

{

**if** (token\_table[i].operator.equals("START")) //START 면 LOCCTR 을 그 인자로 초기화

{

locctr = Integer.*valueOf*(token\_table[i].operand[0],16);

System.***out***.printf("%04X %s %s %s\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand);

**continue**;

}

**else** **if** (token\_table[i].label.equals(".")) //주석 처리, 주석은 opcode를 찾을 필요가 없음.

{

System.***out***.printf("%s %s %s %s\n", token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].comment);

**continue**;

}

**if** (token\_table[i].operator.equals("CSECT")) //Control section이 . -> locctr 초기화, section\_num으로 배열 내에서 구분

{

locctr = 0;

section\_num++;

}

**if** ((index = search\_opcode(token\_table[i].operator)) > -1) //opcode가 있을때, 이때만 pc값 기록

{

**if** (inst\_table[index].type.equals("1")) // format 1 일 때

{

token\_table[i].object = Integer.*valueOf*(inst\_table[index].opcode,16);

System.***out***.printf("%04X %s %s %s %06X\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].object);

locctr = locctr + 1;

}

**else** **if** (inst\_table[index].type.equals("2")) // format 2 일 때 operand로 오는 레지스터에 따라 구분

{

token\_table[i].object = Integer.*valueOf*(inst\_table[index].opcode,16);

**for** (**int** j = 0; j < 2; j++)

{

token\_table[i].object = token\_table[i].object << 4; //주소계산. opcode를 쓴 후 4비트씩 밀면서 operand들 기록

**if** (token\_table[i].operand[j].equals(""))

token\_table[i].object += 0;

**else**

{

**if** (token\_table[i].operand[j].equals("A"))

token\_table[i].object += 0;

**else** **if** (token\_table[i].operand[j].equals("X"))

token\_table[i].object += 1;

**else** **if** (token\_table[i].operand[j].equals("L"))

token\_table[i].object += 2;

**else** **if** (token\_table[i].operand[j].equals("B"))

token\_table[i].object += 3;

**else** **if** (token\_table[i].operand[j].equals("S"))

token\_table[i].object += 4;

**else** **if** (token\_table[i].operand[j].equals("T"))

token\_table[i].object += 5;

**else** **if** (token\_table[i].operand[j].equals("F"))

token\_table[i].object += 6;

**else** **if** (token\_table[i].operand[j].equals("PC"))

token\_table[i].object += 8;

**else** **if** (token\_table[i].operand[j].equals("SW"))

token\_table[i].object += 9;

}

}

System.***out***.printf("%04X %s %s %s %04X\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].object);

locctr = locctr + 2;

}

**else** **if** (inst\_table[index].type.equals("3/4"))

{

**int** target\_addr = -1;

**int** dest = 0;

**if** (token\_table[i].operator.charAt(0) == '+') //4형식

{

token\_table[i].nixbpe += N + I + E; //NIXBPE중에 4형식은 NIE 기본, X가 추가될수 있으므로 값 할당

**if** (token\_table[i].operand[1].equals("X"))

token\_table[i].nixbpe += X;

target\_addr = search\_symbol(token\_table[i].operand[0]);

**if** (target\_addr == -2) //라벨이 있으나 다른 섹션에 있음. M레코드에 써야하므로 구조체에 기록

{

mod\_table[modification\_line].index = i;

mod\_table[modification\_line].section = section\_num;

modification\_line++;

target\_addr = 0;

}

token\_table[i].object = Integer.*valueOf*(inst\_table[index].opcode,16) + (token\_table[i].nixbpe >> 4); //nixpbe의 상위 2비트만 사용

token\_table[i].object = (token\_table[i].object << 4) + (token\_table[i].nixbpe & 15); //15와 & 연산 .하위 4비트 만 사용

token\_table[i].object = (token\_table[i].object << 20) + target\_addr; //앞의 (16진수로)3자리 계산후 5자리 뒤로 밈 그후 타겟주소 계산

System.***out***.printf("%04X %s %s %s %06X\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].object);

locctr = locctr + 4;

}

**else** //3형식

{

**if** (token\_table[i].operand[0].charAt(0) == '#') //immediate addressing

{

String temp = token\_table[i].operand[0].substring(1);

target\_addr = Integer.*valueOf*(temp,10); //immediate은 10진수 이므로

token\_table[i].nixbpe += I; // I 비트만 사용

token\_table[i].object = Integer.*valueOf*(inst\_table[index].opcode,16) + (token\_table[i].nixbpe >> 4); //nixpbe의 상위 2비트만 사용

token\_table[i].object = (token\_table[i].object << 4) + (token\_table[i].nixbpe & 15); //15와 & 연산 .하위 4비트 만 사용

token\_table[i].object = (token\_table[i].object << 12) + target\_addr; //앞의 (16진수로)3자리 계산후 3자리 뒤로 밈 그후 타겟주소 계산

System.***out***.printf("%04X %s %s %s %06X\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].object);

locctr = locctr + 3;

**continue**;

}

**else** **if** (token\_table[i].operand[0].charAt(0) == '@') //indirect addressing

{

String temp = token\_table[i].operand[0].substring(1);

target\_addr = search\_symbol(temp);

dest = target\_addr - token\_table[i].pcValue; //target addr - pc 값

token\_table[i].nixbpe += N + P;

token\_table[i].object = Integer.*valueOf*(inst\_table[index].opcode,16) + (token\_table[i].nixbpe >> 4); //nixpbe의 상위 2비트만 사용

token\_table[i].object = (token\_table[i].object << 4) + (token\_table[i].nixbpe & 15); //15와 & 연산 .하위 4비트 만 사용

token\_table[i].object = (token\_table[i].object << 12) + dest; //앞의 (16진수로)3자리 계산후 3자리 뒤로 밈 그후 dest 주소 계산

System.***out***.printf("%04X %s %s %s %06X\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].object);

locctr = locctr + 3;

}

**else** //direct addressing

{

**if** (token\_table[i].operand[0].charAt(0) == '=') //리터럴

{

target\_addr = search\_literal(token\_table[i].operand[0]);

}

**else**

{

**if** (inst\_table[index].operandAmount == 0) //operand가 없을 때

{

target\_addr = 0; //RSUB 같은 operand가 없는 라인은 주소를 0으로 해줌

token\_table[i].nixbpe += N + I;

token\_table[i].object = Integer.*valueOf*(inst\_table[index].opcode,16) + (token\_table[i].nixbpe >> 4);

token\_table[i].object = (token\_table[i].object << 4) + (token\_table[i].nixbpe & 15); //15와 & 연산 .하위 4비트 만 사용

token\_table[i].object = (token\_table[i].object << 12) + target\_addr;

//System.out.println(""+locctr+" "+token\_table[i].label + " " + token\_table[i].operator + " "+token\_table[i].total\_operand + " "+token\_table[i].object);

System.***out***.printf("%04X %s %s %s %06X\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].object);

locctr = locctr + 3;

**continue**;

}

**else** //operand가 있을 때

target\_addr = search\_symbol(token\_table[i].operand[0]);

}

**if** (target\_addr == -1) //에러 발생 symbol을 쓰려는데 없음.

**return** -1;

**else** //타겟 주소를 받아옴.

{

dest = target\_addr - token\_table[i].pcValue;

**if** (-2048 <= dest && dest <= 2047) //PC relative 방식 범위 내에 존재해야함

{

token\_table[i].nixbpe += N + I + P;

token\_table[i].object = Integer.*valueOf*(inst\_table[index].opcode,16) + (token\_table[i].nixbpe >> 4);

token\_table[i].object = (token\_table[i].object << 4) + (token\_table[i].nixbpe & 15); //15와 & 연산 ->하위 4비트 만 사용

**if** (dest < 0)

{

token\_table[i].object = (token\_table[i].object << 12) + (dest & 0x00000FFF); //음수일 경우 하위 3바이트만 주소로 사용해야하므로 비트연산함

}

**else**

token\_table[i].object = (token\_table[i].object << 12) + dest;

System.***out***.printf("%04X %s %s %s %06X\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].object);

locctr = locctr + 3;

}

**else** //error

**return** -1;

}

}

}

}

}

**else** // opcode가 없는 directive 일 때

{

**if** (token\_table[i].operator.equals("WORD"))

{

**int** check = 0;

**for** (**int** j = 0; j < token\_table[i].operand[0].length(); j++)

{

**if** (token\_table[i].operand[0].charAt(j) == '-') //- 가 있을떄,

{

StringTokenizer st = **new** StringTokenizer(token\_table[i].operand[0],"-\n");

String tok = st.nextToken();

**int** sym\_index = search\_symbol\_index(tok);

tok = st.nextToken();

**int** sym\_index2 = search\_symbol\_index(tok);

sym\_table[sym\_index2].sign = '-';

**if** (sym\_table[sym\_index].section != section\_num || sym\_table[sym\_index2].section != section\_num)

{

mod\_table[modification\_line].index = i;

mod\_table[modification\_line].section = section\_num;

modification\_line++;

token\_table[i].object = 0;

System.***out***.printf("%04X %s %s %s %06X\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].object);

locctr = locctr + 3;

check = 1;

**break**;

}

}

**else** **if** (token\_table[i].operand[0].charAt(j) == '+') // + 가 있을때

{

StringTokenizer st = **new** StringTokenizer(token\_table[i].operand[0],"+\n");

String tok = st.nextToken();

**int** sym\_index = search\_symbol\_index(tok);

tok = st.nextToken();

**int** sym\_index2 = search\_symbol\_index(tok);

**if** (sym\_table[sym\_index].section != section\_num || sym\_table[sym\_index2].section != section\_num)

{

mod\_table[modification\_line].index = i;

mod\_table[modification\_line].section = section\_num;

modification\_line++;

token\_table[i].object = 0;

System.***out***.printf("%04X %s %s %s %06X\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].object);

locctr = locctr + 3;

check = 1;

**break**;

}

}

}

**if** (check == 0) // 두 부호가 존재하지 않을때, 즉 계산하지 않아도될때

{

token\_table[i].object = 3 \* Integer.*parseInt*(token\_table[i].operand[0]);

System.***out***.printf("%04X %s %s %s %06X\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].object);

locctr = locctr + 3;

}

}

**else** **if** (token\_table[i].operator.equals("BYTE")) //첫 글자가 X(16진수, 문자 2개당 1바이트), C(문자 1개당 1바이트) 일때 처리

{

StringTokenizer st = **new** StringTokenizer(token\_table[i].total\_operand,"'");

String tok = st.nextToken();

**if** (tok.equals("C"))

{

tok = st.nextToken();

**for** (**int** j = 0; j <tok.length(); j++)

{

token\_table[i].object += tok.charAt(j);

**if** (j == tok.length() - 1)

**continue**;

token\_table[i].object = token\_table[i].object << 8; //바이트 수 만큼 object 코드 쉬프트함

}

System.***out***.printf("%04X %s %s %s %X\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].object);

locctr = locctr + tok.length();

}

**else** **if** (tok.equals("X"))

{

tok = st.nextToken();

token\_table[i].object = Integer.*valueOf*(tok,16);

System.***out***.printf("%04X %s %s %s %02X\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].object);

locctr = locctr + tok.length() / 2;

}

}

**else** **if** (token\_table[i].operator.equals("RESW"))

{

System.***out***.printf("%04X %s %s %s\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand);

locctr = locctr + 3 \* Integer.*parseInt*(token\_table[i].total\_operand);

}

**else** **if** (token\_table[i].operator.equals("RESB"))

{

System.***out***.printf("%04X %s %s %s\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand);

locctr = locctr + Integer.*parseInt*(token\_table[i].total\_operand);

}

**else** **if** (token\_table[i].operator.equals("LTORG") || token\_table[i].operator.equals("END"))

{

System.***out***.printf(" %s %s %s\n", token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand);

**for** (**int** j = ltorg\_line; j < i; j++)

{

**if** (token\_table[j].operand[0].charAt(0) == '=')

{

**int** lit\_index = search\_literal\_index(token\_table[j].operand[0]);

**if** (lit\_table[lit\_index].object != 0) //이미 기록되어 있다면. 무시

;

**else**

{

StringTokenizer st = **new** StringTokenizer(token\_table[j].operand[0],"'");

String tok =st.nextToken();

**if** (tok.equals("=C"))

{

tok = st.nextToken();

**for** (**int** k = 0; k < tok.length(); k++)

{

lit\_table[lit\_index].object += tok.charAt(k);

**if** (k == tok.length() - 1)

**continue**;

lit\_table[lit\_index].object = lit\_table[lit\_index].object << 8;

}

System.***out***.printf("%04X %s %s %X\n", locctr, "\*", lit\_table[lit\_index].literal, lit\_table[lit\_index].object);

locctr = locctr + tok.length();

}

**else** **if** (tok.equals("=X"))

{

tok = st.nextToken();

lit\_table[lit\_index].object = Integer.*valueOf*(tok,16);

System.***out***.printf("%04X %s %s %02X\n", locctr, "\*", lit\_table[lit\_index].literal, lit\_table[lit\_index].object);

locctr = locctr + tok.length() / 2;

}

}

}

}

ltorg\_line = i; //ltorg 라인 관리. 이전 ltorg에서 나온 리터럴은 무시하도록 해야하기때문

}

**else**

System.***out***.printf(" %s %s %s\n", token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand);

}

}

**return** 0;

}

/\* ----------------------------------------------------------------------------------

\* 설명 : 입력된 문자열의 이름을 가진 파일에 프로그램의 결과를 저장하는 함수이다.

\* 여기서 출력되는 내용은 object code (프로젝트 1번) 이다.

\* 매개 : 생성할 오브젝트 파일명

\* 반환 : 없음

\* 주의 : 만약 인자로 NULL값이 들어온다면 프로그램의 결과를 표준출력으로 보내어

\* 화면에 출력해준다.

\*

\* -----------------------------------------------------------------------------------

\*/

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

{

PrintWriter pw = **null**;

**try** {

pw = **new** PrintWriter(file\_name);

//PrintWriter 클래스를 이용해 파일 출력을 함.(Printwriter 클래스의 경우 printf를 지원함으로써 포맷에 따른 출력을 지원한다는게 장점이다)

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**int** index = 0;

section\_num = 0;

**int** ltorg\_line = 0;

**int** change\_line\_index = 0;

**int** current\_addr = 0;

**int** totalbyte = 0;

**int** checkindex = 0;

**int** count = 0;

**int** firstaddr = 0;

**for** (**int** i = 0; i <= line\_num; i++)

{

**if** (token\_table[i].operator.equals("START"))

{

locctr = Integer.*valueOf*(token\_table[i].operand[0],16);

firstaddr = locctr; //H레코드 에서 시작주소를 저장하기 위해 저장

totalbyte = Integer.*valueOf*(token\_table[i].operand[0],16);

pw.printf("H%-6s%06X%06X", token\_table[i].label, totalbyte, section\_length[section\_num]); //H레코드 기록

pw.println("");

}

**else** **if** (token\_table[i].label.equals(".")) //주석 처리, 주석은 opcode를 찾을 필요가 없음.

{

**continue**;

}

**if** ((index = search\_opcode(token\_table[i].operator)) > -1) //opcode가 있을때 각 토큰들에선 현재 얼마까지 읽었는지만 기록

{

**if** (inst\_table[index].type.equals("1")) // format 1 일 때

{

count = 1;

totalbyte = totalbyte + count;

locctr = locctr + 1;

}

**else** **if** (inst\_table[index].type.equals("2")) // format 2 일 때 operand로 오는 레지스터에 따라 구분

{

count = 2;

totalbyte = totalbyte + count;

locctr = locctr + 2;

}

**else** **if** (inst\_table[index].type.equals("3/4"))

{

**if** (token\_table[i].operator.charAt(0) == '+')

{

count = 4;

totalbyte = totalbyte + count;

locctr = locctr + 4;

}

**else**

{

count = 3;

totalbyte = totalbyte + count;

locctr = locctr + 3;

}

}

}

**else** // opcode가 없는 directive 일 때

{

**if** (token\_table[i].operator.equals("WORD"))

{

count = 3;

totalbyte = totalbyte + count;

locctr = locctr + 3;

}

**else** **if** (token\_table[i].operator.equals("BYTE")) //첫 글자가 X(16진수, 문자 2개당 1바이트), C(문자 1개당 1바이트) 일때 처리

{

StringTokenizer st = **new** StringTokenizer(token\_table[i].total\_operand,"'");

String tok = st.nextToken();

**if** (tok.equals("C"))

{

tok = st.nextToken();

count = tok.length();

totalbyte = totalbyte + count;

locctr = locctr + count;

}

**else** **if** (tok.equals("X"))

{

tok = st.nextToken();

count = tok.length()/2;

totalbyte = totalbyte + count;

locctr = locctr + count;

}

}

**else** **if** (token\_table[i].operator.equals("RESW"))

{

count = 3 \* Integer.*parseInt*(token\_table[i].total\_operand);

locctr = locctr + count;

}

**else** **if** (token\_table[i].operator.equals("RESB"))

{

count = Integer.*parseInt*(token\_table[i].total\_operand);

locctr = locctr + count;

}

**else** **if** (token\_table[i].operator.equals("EXTDEF"))

{

count = 0;

pw.printf("D");

**for** (**int** j = 0; j < MAX\_OPERAND; j++)

{

**if** (token\_table[i].operand[j].equals(""))

**break**;

**else**

{

**int** addr = search\_symbol(token\_table[i].operand[j]); //각 주소를 비교 문자열로 받아옴

pw.printf("%-6s%06X", token\_table[i].operand[j], addr);

}

}

pw.println("");

}

**else** **if** (token\_table[i].operator.equals("EXTREF"))

{

count = 0;

pw.printf("R");

**for** (**int** j = 0; j < MAX\_OPERAND; j++)

{

**if** (token\_table[i].operand[j].equals(""))

**break**;

**else**

pw.printf("%-6s", token\_table[i].operand[j]);

}

pw.println("");

}

**else** **if** (token\_table[i].operator.equals("LTORG")) //LTORG 위 라인 까지 object 코드로 출력, 그 후 리터럴에 대한 코드 정보 처리(리터럴은 CSECT,END에서 출력)

{

count = 0;

pw.printf("T%06X%02X", current\_addr, totalbyte - count);

**for** (**int** j = checkindex; j < i; j++)

{

**int** temp\_index = search\_opcode(token\_table[j].operator);

**if** (token\_table[j].hasobjectcode == 1)

{

**if** (temp\_index == -1)

{

**if** (token\_table[j].operator.equals("WORD"))

pw.printf("%06X", token\_table[j].object);

**else**

pw.printf("%02X", token\_table[j].object);

}

**else** **if** (inst\_table[temp\_index].type.equals("2"))

pw.printf("%04X", token\_table[j].object);

**else**

pw.printf("%06X", token\_table[j].object);

}

}

pw.println("");

current\_addr += totalbyte - count;

totalbyte = 0;

checkindex = i; //어디까지 읽은지 저장

/////////////////////리터럴 코드

current\_addr = locctr;

**for** (**int** j = 0; j <= literal\_line; j++)

{

**if** (lit\_table[j].section == section\_num)

{

StringTokenizer st = **new** StringTokenizer(lit\_table[j].literal,"'");

String tok = st.nextToken();

**if** (tok.equals("=C"))

{

tok = st.nextToken();

count = tok.length();

totalbyte = totalbyte + count;

locctr = locctr + count;

}

**else** **if** (tok.equals("=X"))

{

tok = st.nextToken();

count = tok.length()/2;

totalbyte = totalbyte + count;

locctr = locctr + count;

}

}

}

}

**else** **if** (token\_table[i].operator.equals("END"))

{

**for** (**int** j = 0; j <= literal\_line; j++)

{

**if** (lit\_table[j].section == section\_num)

{

StringTokenizer st = **new** StringTokenizer(lit\_table[j].literal,"'");

String tok = st.nextToken();

**if** (tok.equals("=C"))

{

tok = st.nextToken();

count = tok.length();

totalbyte = totalbyte + count;

locctr = locctr + count;

}

**else** **if** (tok.equals("=X"))

{

tok = st.nextToken();

count = tok.length()/2;

totalbyte = totalbyte + count;

locctr = locctr + count;

}

}

}

pw.printf("T%06X%02X", current\_addr, totalbyte);

**for** (**int** j = checkindex; j < i; j++)

{

**if** (token\_table[j].label.equals(".")) //주석 처리, 주석은 opcode를 찾을 필요가 없음.

{

**continue**;

}

**int** temp\_index = search\_opcode(token\_table[j].operator);

**if** (token\_table[j].hasobjectcode == 1)

{

**if** (temp\_index == -1)

{

**if** (token\_table[j].operator.equals("WORD"))

pw.printf("%06X", token\_table[j].object);

**else**

pw.printf("%02X", token\_table[j].object);

}

**else** **if** (inst\_table[temp\_index].type.equals("2"))

pw.printf("%04X", token\_table[j].object);

**else**

pw.printf("%06X", token\_table[j].object);

}

}

/////////////////////리터럴 코드 출력

**for** (**int** j = 0; j <= literal\_line; j++)

**if** (lit\_table[j].section == section\_num)

{

pw.printf("%02X", lit\_table[j].object);

pw.println("");

}

**for** (**int** j = 0; j < modification\_line; j++) //modification record 가 있을때 출력

{

**int** x = mod\_table[j].index;

**if** (mod\_table[j].section == section\_num)

{

**if** (token\_table[x].operator.equals("WORD"))

{

**int** check = 0;

**for** (**int** k = 0; k < token\_table[x].total\_operand.length(); k++)

{

**if** (token\_table[x].total\_operand.charAt(k) == '-') //- 가 있을떄,

{

StringTokenizer st = **new** StringTokenizer(token\_table[x].total\_operand,"-\n");

String tok = st.nextToken();

**int** sym\_index = search\_symbol\_index(tok);

tok = st.nextToken();

**int** sym\_index2 = search\_symbol\_index(tok);

pw.printf("M%06X06%c%s", token\_table[x].current\_locctr, sym\_table[sym\_index].sign, sym\_table[sym\_index].symbol);

pw.println("");

pw.printf("M%06X06%c%s", token\_table[x].current\_locctr, sym\_table[sym\_index2].sign, sym\_table[sym\_index2].symbol);

pw.println("");

**break**;

}

}

}

**else**

{

pw.printf("M%06X05+%s", token\_table[x].current\_locctr + 1, token\_table[x].operand[0]);

pw.println("");

}

}

}

**if** (section\_num == 0)

{

pw.printf("E%06X", firstaddr);

pw.println("");

}

**else**

{

pw.printf("E");

pw.println("");

}

}

**else** **if** (token\_table[i].operator.equals("CSECT")) //Control section이 바뀜 . locctr 초기화, section\_num으로 배열 내에서 구분

{

**if** (hasliteral[section\_num]==1) //리터럴이 있을때 출력

{

pw.printf("T%06X%02X", current\_addr, totalbyte);

**for** (**int** j = 0; j <= literal\_line; j++)

{

**if** (lit\_table[j].section == section\_num)

{

pw.printf("%02X", lit\_table[j].object);

}

}

pw.println("");

}

**else**

{

count = 0;

pw.printf("T%06X%02X", current\_addr, totalbyte - count); //섹션이 바뀔때 다 출력하지 못한것들 출력

**for** (**int** j = checkindex; j < i; j++)

{

**int** temp\_index = search\_opcode(token\_table[j].operator);

**if** (token\_table[j].hasobjectcode == 1)

{

**if** (temp\_index == -1)

{

**if** (token\_table[j].operator.equals("WORD"))

pw.printf("%06X", token\_table[j].object);

**else**

pw.printf("%02X", token\_table[j].object);

}

**else** **if** (inst\_table[temp\_index].type.equals("2"))

pw.printf("%04X", token\_table[j].object);

**else**

pw.printf("%06X", token\_table[j].object);

}

}

pw.println("");

}

**for** (**int** j = 0; j < modification\_line; j++)

{

**int** x = mod\_table[j].index;

**if** (mod\_table[j].section == section\_num)

{

**if** (token\_table[x].operator.equals("WORD"))

{

**int** check = 0;

**for** (**int** k = 0; k < token\_table[x].total\_operand.length(); k++)

{

**if** (token\_table[x].total\_operand.charAt(k) == '-') //- 가 있을떄,

{

StringTokenizer st = **new** StringTokenizer(token\_table[x].total\_operand,"-\n");

String tok = st.nextToken();

**int** sym\_index = search\_symbol\_index(tok);

tok = st.nextToken();

**int** sym\_index2 = search\_symbol\_index(tok);

pw.printf("M%06X06%c%s", token\_table[x].current\_locctr, sym\_table[sym\_index].sign, sym\_table[sym\_index].symbol);

pw.println("");

pw.printf("M%06X06%c%s", token\_table[x].current\_locctr, sym\_table[sym\_index2].sign, sym\_table[sym\_index2].symbol);

pw.println("");

**break**;

}

}

}

**else**

{

pw.printf("M%06X05+%s", token\_table[x].current\_locctr + 1, token\_table[x].operand[0]);

pw.println("");

}

}

}

**if** (section\_num == 0)

{

pw.printf("E%06X", firstaddr);

pw.println("");

}

**else**

{

pw.printf("E");

pw.println("");

}

locctr = 0;

section\_num++;

totalbyte = 0;

checkindex = i;

current\_addr = 0;

pw.println("");

pw.printf("H%-6s%06X%06X", token\_table[i].label, locctr, section\_length[section\_num]);

pw.println("");

}

}

**if** (totalbyte >= 30)

{

pw.printf("T%06X%02X", current\_addr, totalbyte - count); //시작주소, 현재까지 파일에 쓴 바이트

**for** (**int** j = checkindex; j < i; j++) //30바이트가 넘기전까지 출력

{

**if** (token\_table[j].label.equals(".")) //주석 처리, 주석은 opcode를 찾을 필요가 없음.

{

**continue**;

}

**int** temp\_index = search\_opcode(token\_table[j].operator);

**if** (token\_table[j].hasobjectcode == 1)

{

**if** (temp\_index == -1)

{

**if** (token\_table[j].operator.equals("WORD"))

pw.printf("%06X", token\_table[j].object);

**else**

pw.printf("%02X", token\_table[j].object);

}

**else** **if** (inst\_table[temp\_index].type.equals("2"))

pw.printf("%04X", token\_table[j].object);

**else**

pw.printf("%06X", token\_table[j].object);

}

}

pw.println("");

current\_addr += totalbyte - count;

totalbyte = count;

checkindex = i;

}

}

pw.flush(); // 자바의 스트림은 flush를 해줘야 출력을 해줌

pw.close(); // 열었던 스트림을 닫아줌.

}

}

주요 소스코드 부분 설명

dest = target\_addr - token\_table[i].pcValue;

**if** (-2048 <= dest && dest <= 2047) //PC relative 방식 범위 내에 존재해야함

{

token\_table[i].nixbpe += N + I + P;

token\_table[i].object = Integer.*valueOf*(inst\_table[index].opcode,16) + (token\_table[i].nixbpe >> 4);

token\_table[i].object = (token\_table[i].object << 4) + (token\_table[i].nixbpe & 15); //15와 & 연산 ->하위 4비트 만 사용

**if** (dest < 0)

{

token\_table[i].object = (token\_table[i].object << 12) + (dest & 0x00000FFF); //음수일 경우 하위 3바이트만 주소로 사용해야하므로 비트연산함

}

**else**

token\_table[i].object = (token\_table[i].object << 12) + dest;

System.***out***.printf("%04X %s %s %s %06X\n", locctr, token\_table[i].label, token\_table[i].operator,token\_table[i].total\_operand, token\_table[i].object);

locctr = locctr + 3;

}

* Pass2 에서 주소계산을 해주는 부분.

PC relative방식으로 처리하는 3형식 명령어이다. 3형식이며 PC 방식이므로 nixbpe에 N I P 자리만 비트를 1로 올려줬고 이를 이용해 값을 구한다. 먼저 문자열형태로 저장된 opcode를 16진수 형태로 바꾼 후 nixbpe의 상위 2비트를 더해주고 저장한다. 그 후 왼쪽으로 4비트 밀어주고 xixbpe의 하위 4비트를 더해주고 저장, 다시 왼쪽으로 12비트를 밀어준 후 target\_addr - pc값을 빼준 값을 더해주면 object code의 계산이 완료된다.

**if** (totalbyte >= 30)

{

pw.printf("T%06X%02X", current\_addr, totalbyte - count); //시작주소, 현재까지 파일에 쓴 바이트

**for** (**int** j = checkindex; j < i; j++) //30바이트가 넘기전까지 출력

{

**if** (token\_table[j].label.equals(".")) //주석 처리, 주석은 opcode를 찾을 필요가 없음.

{

**continue**;

}

**int** temp\_index = search\_opcode(token\_table[j].operator);

**if** (token\_table[j].hasobjectcode == 1)

{

**if** (temp\_index == -1)

{

**if** (token\_table[j].operator.equals("WORD"))

pw.printf("%06X", token\_table[j].object);

**else**

pw.printf("%02X", token\_table[j].object);

}

**else** **if** (inst\_table[temp\_index].type.equals("2"))

pw.printf("%04X", token\_table[j].object);

**else**

pw.printf("%06X", token\_table[j].object);

}

}

pw.println("");

current\_addr += totalbyte - count;

totalbyte = count;

checkindex = i;

}

}

* Make\_objectcode\_output에서 T 레코드를 쓰는 부분

for문을 돌면서 체크. totalbyte = 현재까지 읽은 바이트. 즉 30이 넘어가면 현재 token\_table에 접근하는 i 인덱스까지의 object 코드와 각 T 레코드 정보를 출력한다.

PrintWriter pw = **null**;

**try** {

pw = **new** PrintWriter(file\_name);

//PrintWriter 클래스를 이용해 파일 출력을 함.(Printwriter 클래스의 경우 printf를 지원함으로써 포맷에 따른 출력을 지원한다는게 장점이다)

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

* Make\_objectcode\_output 함수 내에서 파일 출력 부분

C언어와 다르게 자바는 각종 입출력 스트림을 지원하며, 각종 입출력을 도와주는 필터스트림 또한 지원한다. 필터스트림 중 하나인 PrinterWriter 클래스를 이용해 입출력을 수행한다. PrinterWriter 클래스에는 printf함수가 존재하여 포맷에 따른 출력을 지원한다. 이 printf 함수를 이용해 출력을 하였다.

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

{

**int** errno = 0;

**int** i = 0;

**try**

{

Scanner scanner = **new** Scanner(**new** File(inst\_file));

**while**(scanner.hasNext()){

StringTokenizer st = **new** StringTokenizer(scanner.nextLine(),"\t\n"); //스캐너 클래스를 이용해 한줄씩 받아옴 + 탭,개행으로 토크나이즈

inst\_table[i] = **new** inst\_unit();

inst\_table[i].operator = st.nextToken();

inst\_table[i].type = st.nextToken();

inst\_table[i].opcode = st.nextToken();

inst\_table[i].operandAmount = Integer.*parseInt*(st.nextToken());

inst\_index = i; //0부터 갯수 시작

i++;

}

scanner.close();

}

**catch** (FileNotFoundException e) { //파일이 없으면 try catch 로 예외처리

System.***out***.println("The inst\_file doesn't exist");

e.printStackTrace();

**return** errno = -1;

}

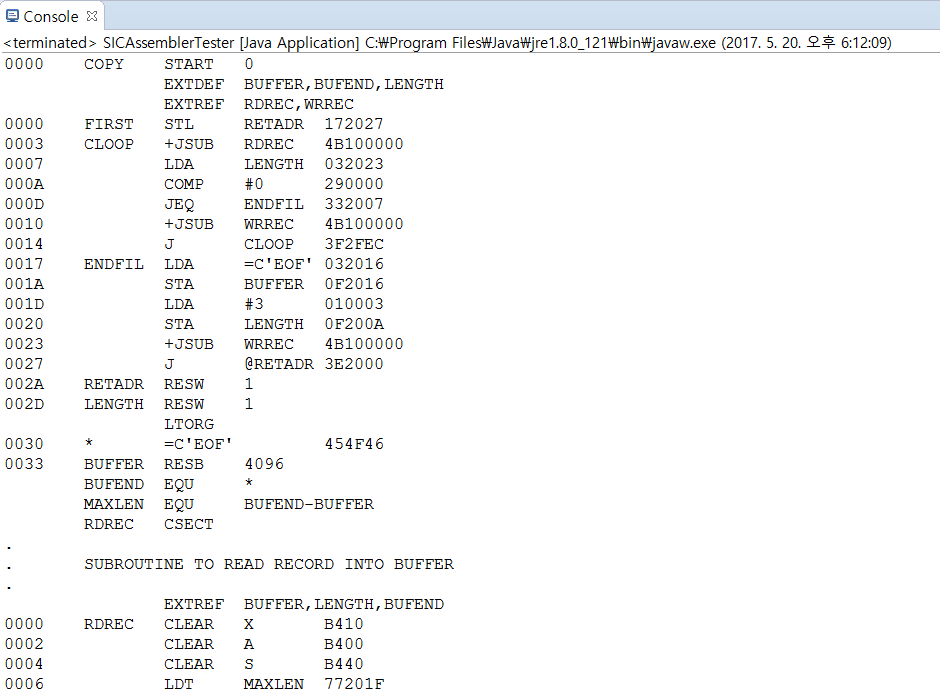
**return** errno;

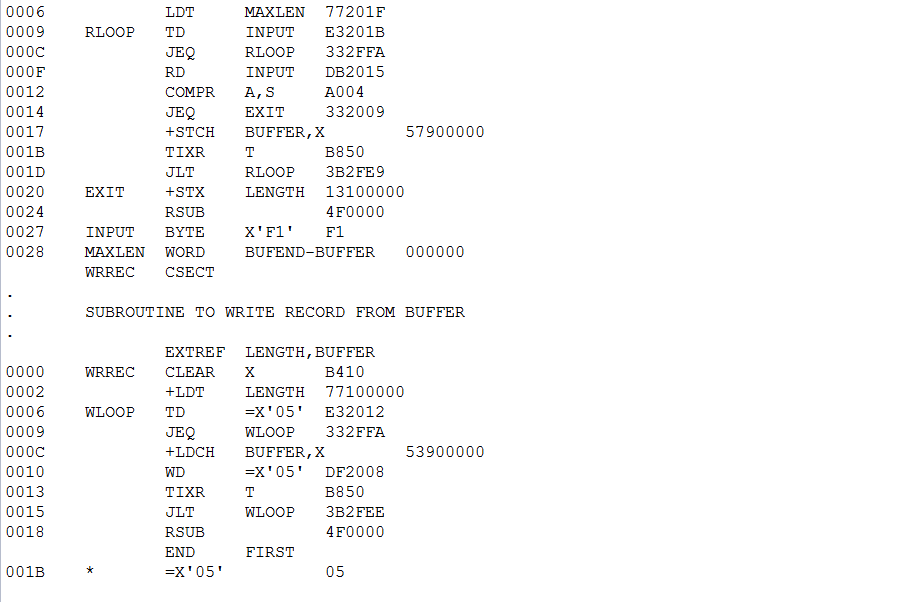
}

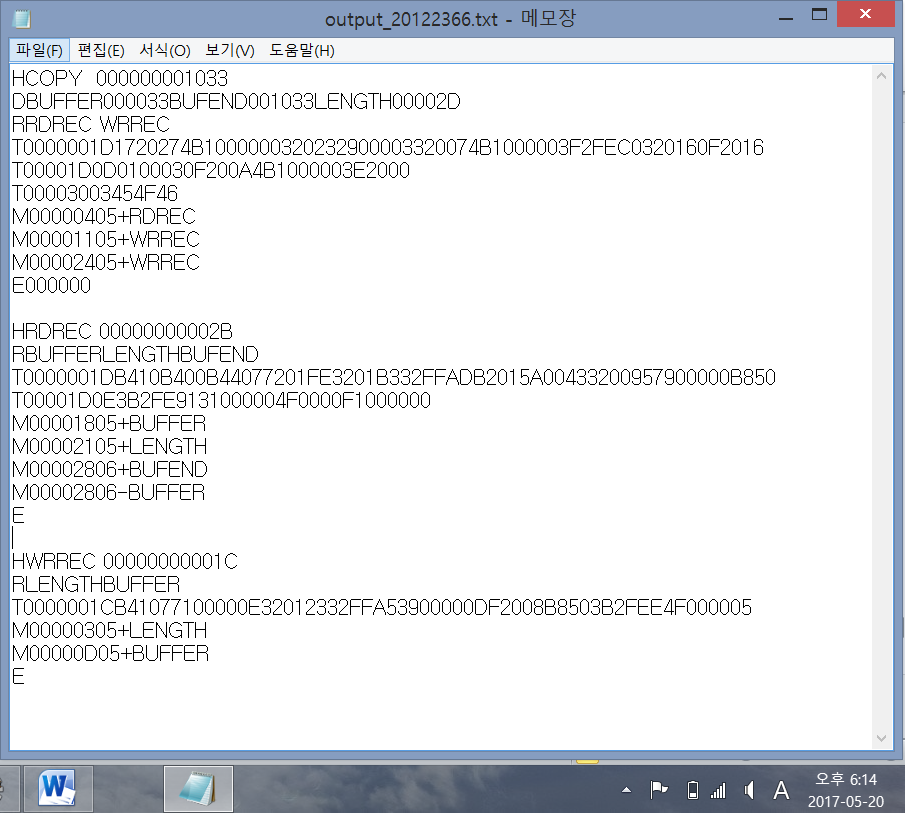
* Init\_inst\_file() 함수

기존 C언어와 다르게 입력을 Scanner 클래스를 이용해 받는다. Scanner 클래스는 한 줄씩 입력을 받아들이며, C언어의 strtok와 같은 StringTokenizer 클래스를 이용하여 토큰을 구분하며 , nextToken() 메소드로 각 값을 할당해준다.

1. 수행 결과







1. 결론 및 보충할 점

프로젝트를 진행하면서 어셈블러의 동작과정에 대해 심도있게 파악하게 되었다. 비록 파일 입출력을 통한 눈으로 보여지기만 하는 어셈블러지만 어셈블러를 구현했다는 것에 큰 보람을 느꼈다. 이 과제를 통해 그동안 잘 쓰지 않았던 씨언어의 각종 문법들을 다시 공부할 수 있었으며, 프로그래밍적으로 조금은 로직 짜는것에 익숙해지고 있음이 느껴진다. 그러나 여러 보충할 점이 있었다. 일단 예제 인풋에 기반하여 만들어서 예외의 경우가 다수 예상된다는 것이다. 또, 피상적으로 파일 입출력이 아닌 커널이나 시스템적으로 구현하는 어셈블러에 도전해보는 것이 더 의미있을 것 같았다. 물론 난이도가 엄청나게 올라가겠지만 추후에 시간이 된다면 시스템에 들어가는 어셈블러에 도전해보고 싶은 마음이다.