|  |
| --- |
|  |

|  |
| --- |
| **시스템 프로그래밍 Project #1B** |
| **Control Section방식 어셈블러**  **Java로 구현** |
|  |
|  |
|  |
| **교수 : 최재영 교수님** |
| **가반 114번**  **컴퓨터학부 20132361 박진수** |
|  |

**목 차**

1. **동기/목적**
2. **설계/구현 아이디어**
3. **수행결과**
4. **결론 및 보충할 점**
5. **소스코드**
6. **동기/목적**

**목적**

**Control Section방식의 어셈블러를 C언어로 구현한 프로젝트 1을 C언어가 아닌 Java나 다른**

**언어로 구현하여 프로그래밍의 언어에 대한 이해와 숙련도를 향상시키고 Java를 사용함으로써**

**객체지향 언어에 대한 이해와 활용방법을 익힐 수 있다.**

**동기**

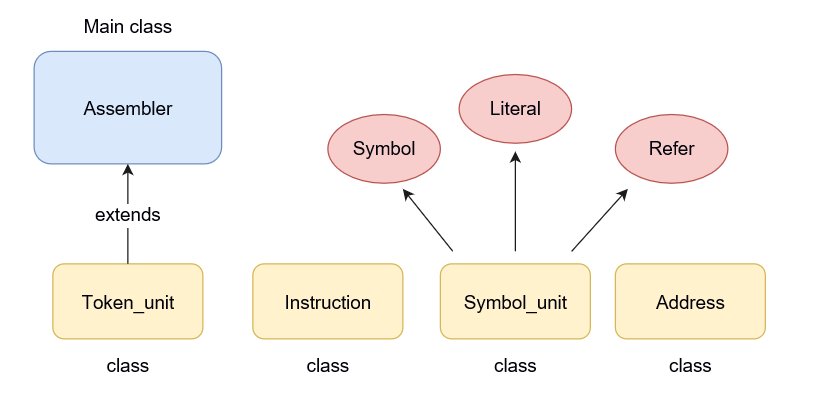
**C언어가 아닌 가장 많이 사용되는 언어인 Java를 이용하여 객체지향 언어의 활용도를 최대한**

**이용하여 객체의 사용이나 상속 등 C언어가 가지지 않는 특징을 최대한 활용하여 좀 더 효율**

**적인 프로그램을 만들 수 있음.**

**.**

1. **설계/구현 아이디어**



**Token, Instruction 저장**

**Inst.data파일 에서 명령어를 한 줄 씩 읽어 들어 Instruction 객체배열에 저장하고 input.txt파일**

**에서 한 줄씩 읽어 들어 Token\_unit 객체배열에 저장을 한다. 널 포인터가 생기는 것을 막기 위해**

**/t로 나눈 문자열의 개수에 따라 객체배열에 저장**

**SYMTAB 저장, Literal, EQU구현, LOCCTR 저장 – PASS 1**

**토큰테이블을 한 개씩 읽어 들어 symbol이 있을 경우 SYMTAB에 저장하고 location counter값도 계산하여 LOCCTR 배열에 저장함. EQU에는 절대값을 저장하고 연산이 있는 경우에는 Token\_unit객체의 Math함수로 연산이 있는지 검사하여 LOCCTR값을 계산하여 저장함. 명령어의 경우**

**형식에 따라 LOCCTR값을 더해서 LOCCTR배열에 저장하였음. LTORG을 만나거나 프로그램 끝에서**

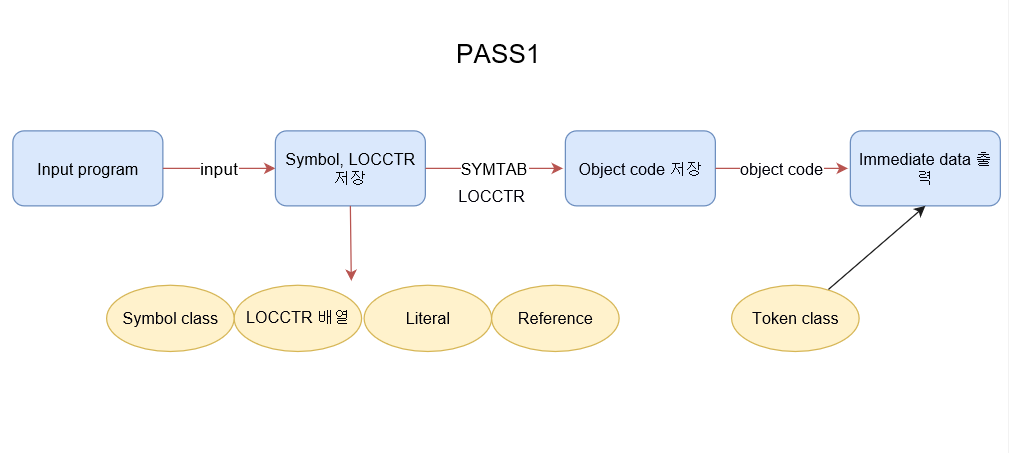
**Literal pool을 비워야 할 경우에는 리터럴을 사용하는 명령어에서 Literals Arraylist에 저장하고**

**LTORG나 프로그램 끝에서 비우고 리터럴 객체배열에 저장함. EXTREF가 나올 경우에는 refer 객체**

**배열에 저장함**

**Object Code 저장, Immediate Data 출력(Address 객체배열) – PASS 1**

**토큰테이블을 한 개씩 읽어 들어 명령어일 경우에는 opcode와 어드레싱 모드와 형식, x레지스터에 따라 nixbpe 값과 어드레싱 모드에 따른 주소값을 addr 구조체에 저장함. 참조값을 사용하는 경우에는 주소값에 0을 저장. 명령어가 아니고 WORD, BYTE나 리터럴 값일 경우에는 연산이 있는 경우 없는 경우, 참조값을 사용하는 경우를 고려하여 주소값을 저장함. 저장한 주소값, opcode, nixbpe 값을 사용하여 바로 화면에 출력함**



**Object Program Output(PASS 2)**

**H레코드에는 addr구조체의 처음 주소값과 pass1에서 section으로 나눈 각 루틴의 마지막 주소값을 저장하였던 값을 출력, T레코드일 경우 30바이트의 수 제한이나 LTORG, section에 의한 조건으로 개행하여 길이와 첫 주소값과 함께 object code를 출력하였음. M레코드의 경우 PASS1에서 참조값을 사용한 경우 addr구조체에 확인 값을 저장하여 명령어 코드의 주소와 수정할 길이, 수정할 operand를 출력, section의 루틴이 끝날 경우에는 E레코드를 출력**

**3. 수행결과**

**Input**

COPY START 0 COPY FILE FROM IN TO OUTPUT

EXTDEF BUFFER,BUFEND,LENGTH

EXTREF RDREC,WRREC

FIRST STL RETADR SAVE RETURN ADDRESS

CLOOP +JSUB RDREC READ INPUT RECORD

LDA LENGTH TEST FOR EOF (LENGTH = 0)

COMP #0

JEQ ENDFIL EXIT IF EOF FOUND

+JSUB WRREC WRITE OUTPUT RECORD

J CLOOP LOOP

ENDFIL LDA =C'EOF' INSERT END OF FILE MARKER

STA BUFFER

LDA #3 SET LENGTH = 3

STA LENGTH

+JSUB WRREC WRITE EOF

J @RETADR RETURN TO CALLER

RETADR RESW 1

LENGTH RESW 1 LENGTH OF RECORD

LTORG

BUFFER RESB 4096 4096-BYTE BUFFER AREA

BUFEND EQU \*

MAXLEN EQU BUFEND-BUFFER MAXIMUM RECORD LENGTH

RDREC CSECT

EXTREF BUFFER,LENGTH,BUFEND

CLEAR X CLEAR LOOP COUNTER

CLEAR A CLEAR A TO ZERO

CLEAR S CLEAR S TO ZERO

LDT MAXLEN

RLOOP TD INPUT TEST INPUT DEVICE

JEQ RLOOP LOOP UNTIL READY

RD INPUT READ CHARACTER INTO REGISTER A

COMPR A,S TEST FOR END OF RECORD (X'00')

JEQ EXIT EXIT LOOP IF EOR

+STCH BUFFER,X STORE CHARACTER IN BUFFER

TIXR T LOOP UNLESS MAX LENGTH

JLT RLOOP HAS BEEN REACHED

EXIT +STX LENGTH SAVE RECORD LENGTH

RSUB RETURN TO CALLER

INPUT BYTE X'F1' CODE FOR INPUT DEVICE

MAXLEN WORD BUFEND-BUFFER

WRREC CSECT

EXTREF LENGTH,BUFFER

CLEAR X CLEAR LOOP COUNTER

+LDT LENGTH

WLOOP TD =X'05' TEST OUTPUT DEVICE

JEQ WLOOP LOOP UNTIL READY

+LDCH BUFFER,X GET CHARACTER FROM BUFFER

WD =X'05' WRITE CHARACTER

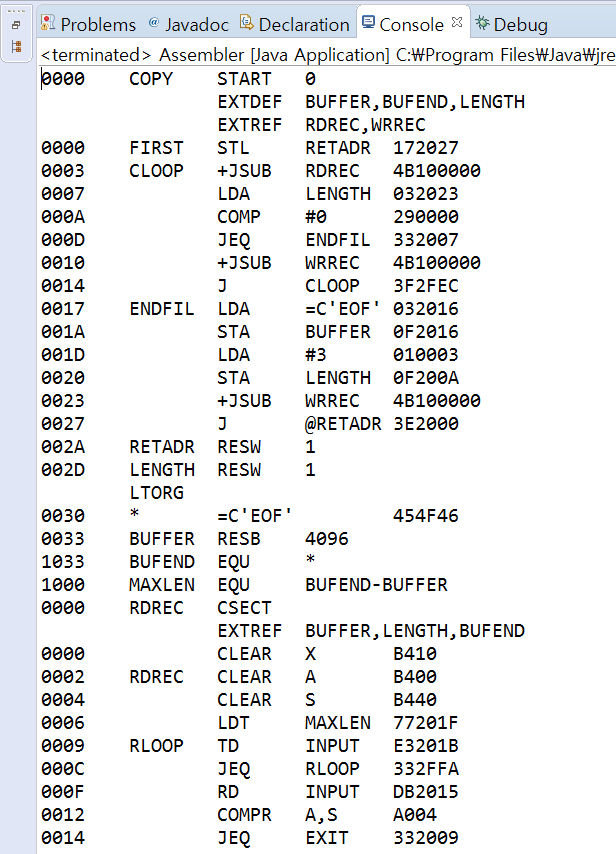
TIXR T LOOP UNTIL ALL CHARACTERS

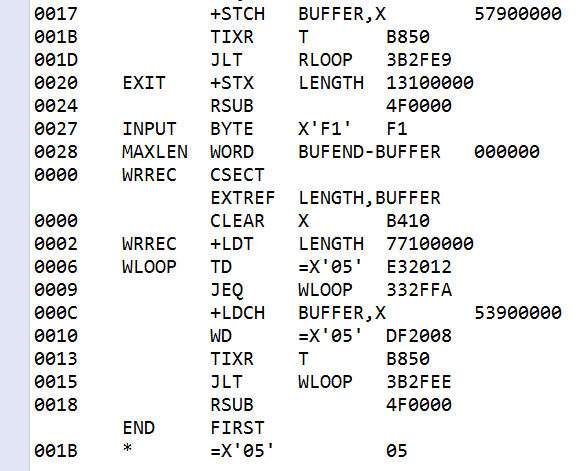
JLT WLOOP HAVE BEEN WRITTEN

RSUB RETURN TO CALLER

END FIRST

**Immediate Data(화면출력)**

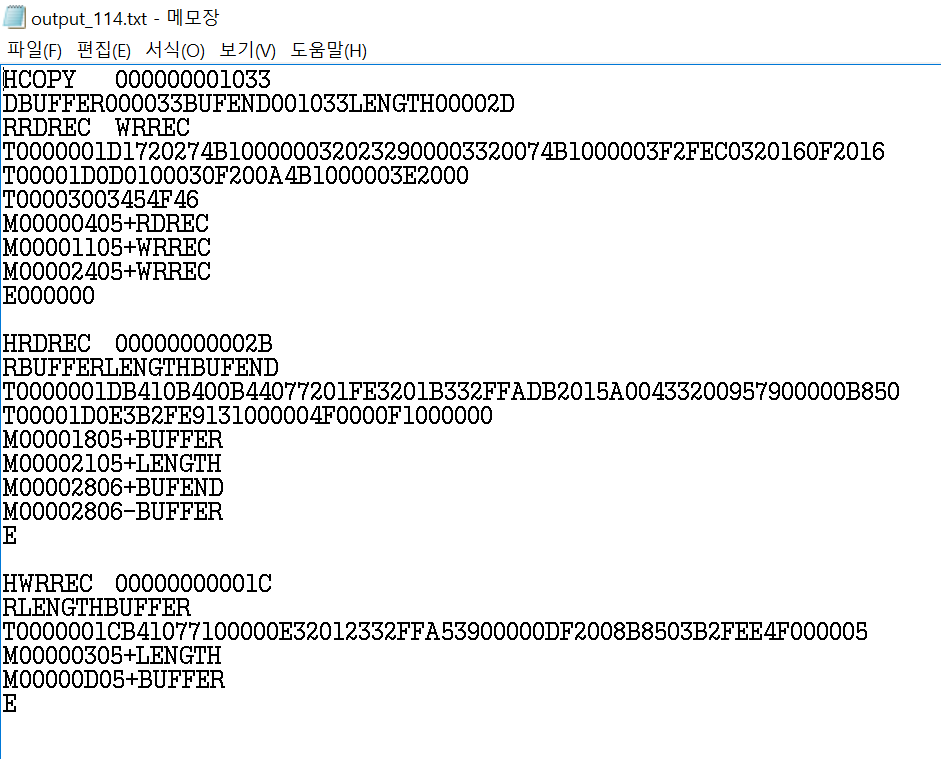




**주어진 입력파일에서 LOCCTR값과 Object code를 추가하고 입력파일에 없는 리터럴 폴과 섹션으로**

**나누어진 부프로그램의 이름을 추가하였음**

**Output(파일 출력)**



**어셈블리 코드를 기계어 코드로 바꾼 PASS 2과정이다. 섹션에 따라 나누었고, 참조값과 같은 모르는**

**주소값을 사용한 경우 수정레코드를 추가하였고, 나머지 T,R,D,H,E레코드도 모두 출력하였다.**

**4. 결론 및 보충할 점**

**명령어(기계어)의 목록파일과 주어진 입력파일을 파일로 입력 받아 Instruction과 Token 테이블을 만들고 그것을 통해 SYMTAB을 만들고 어셈블러 지시어를 처리하고 LOCCTR값과 주소값을 구한 PASS 1을 구현하고 PASS 1을 통하여 기계어 코드로 번역한 Object Program을 완성하였음.**

**자바를 사용함으로써 C언어와 달리 구조체가 없으므로 객체의 사용과 그 객체의 메소드 구현, 상속 등**

**을 이용하여 자바의 숙련도를 높였음. 그리고 char 배열을 사용하는 C언어와 달리 String을 사용하여**

**그에 따른 라이브러리를 사용하였음. 자바를 통한 이번 프로젝트를 통하여 객체지향 언어의 이점과**

**단점을 알게 되었음.**

**SIC/XE 어셈블러에서 Control Section을 방식으로 구현하여 각각의 루틴을 독립적인 부분으로 나누어 그에 따른 메모리 저장과 접근의 이점을 이해하고 외부 참조값과 정의값에 의한 모르는 주소값을 수정 레코드로 수정하여 로더 할 때의 문제점을 해결하고 편리함을 위한 Literal과 EQU를 사용함에 따라 편리함과 신뢰도를 향상 시킬 수 있음.**

**자바의 이점을 최대한 활용하여 하였으나, 상속과 객체에 대한 이해의 부족으로 활용을 제대로 하지**

**못하였음. 상속과 객체의 사용에 따른 데이터의 공유, 값의 참조 등에 어려움을 느낌. 자바의 지속적**

**인 사용을 통하여 좀 더 효율적인 어셈블러 구현과 또 다른 자바의 장점과 GUI 등 수많은 라이브러리**

**의 사용을 해볼 것 이다. 뿐만 아니라 C나 Java에 국한하지 않고 여러 프로그래밍 언어를 사용해보고**

**자신에 맞는 언어를 찾을 것이다.**

**5. 소스코드**

**package** project;

/\*

\* object code 클래스

\*/

**public** **class** Address {

String byte\_ob; //operator이 byte인 경우를 위한 문자열

**int** op; //opcode를 위한 변수(ni값도 포함)

**int** loc; //주소값을 위한 변수

**int** xbpe; //xbpe값을 위한 변수

**int** r\_cnt; //섹션의 순서를 위한 변수

**int** is\_r; //pass2에서 수정레코드에 쓰기 위해 참조값일 경우 1을 저장하는 변수 LTORG일 경우는 2 둘 다 아니 경우는 0

**int** format; //pass2에서 T레코드를 나누기 위해 쓰일 각 라인의 증가할 주소값

Address(String ob,

**int** o,

**int** l,

**int** x,

**int** r,

**int** i,

**int** f)

{

**this**.byte\_ob=ob;

**this**.op=0;

**this**.loc=l;

**this**.xbpe=x;

**this**.r\_cnt=r;

**this**.is\_r=i;

**this**.format=f;

}

}

**package** project;

/\*

\* 심볼을 관리하는 클래스이다.

\* 심볼 테이블은 심볼 이름, 심볼의 위치(주소), 섹션으로 구성된다.

\*/

**public** **class** Symbol\_unit {

**public** String name;

**public** **int** addr;

**public** **int** sub\_cnt;

Symbol\_unit(String n, **int** a, **int** s)

{

**this**.name=n;

**this**.addr=a;

**this**.sub\_cnt=s;

}

**public** **void** setName(String n)

{

**this**.name=n;

}

**public** String getName()

{

**return** name;

}

**public** **void** setAddr(**int** a)

{

**this**.addr=a;

}

**public** **int** getAddr()

{

**return** addr;

}

//주어진 문자열이 symbol인지 확인하는 함수이다.

//반환 : 있을 경우 : true 없을 경우 : false

**public** **boolean** isSymbol(String str,**int** r\_cnt)

{

**boolean** is=**false**;

**if**(r\_cnt==sub\_cnt)

{

**if**(name.equals(str))

{

**return** **true**;

}

}

**return** **false**;

}

}

**package** project;

//

//instruction 목록 파일로 부터 정보를 받아와서 생성하는 클래스

//

**public** **class** instruction {

**public** String name;

**public** **int** opcode;

**public** **int** format;

instruction(String n, **int** op,**int** form)

{

**this**.name=n;

**this**.opcode=op;

**this**.format=form;

}

**public** **void** setName(String n)

{

**this**.name=n;

}

**public** **void** setFormat(**int** f)

{

**this**.format=f;

}

**public** **int** getFormat()

{

**return** format;

}

}

**package** project;

//

// 어셈블리 할 소스코드를 토큰단위로 나눈 클래스

//

**public** **class** Token\_unit **extends** Assembler {

**public** String label;

**public** String operator;

**public** String[] operand={**null**,**null**,**null**,**null**};

**public** String comment;

Token\_unit(String l, String opr, String op, String c)

{

**this**.label=l;

**this**.operator=opr;

**this**.operand[0]=op;

**this**.operand[1]=op;

**this**.operand[2]=op;

**this**.operand[3]=op;

**this**.comment=c;

}

**public** **void** setLabel(String l)

{

**this**.label=l;

}

**public** String getLabel()

{

**return** label;

}

**public** **void** setOperator(String o)

{

**this**.operator=o;

}

**public** String getOperator()

{

**return** operator;

}

**public** **void** setOperand(String r,**int** k)

{

**this**.operand[k]=r;

}

**public** String getOperand(**int** k)

{

**return** operand[k];

}

**public** **void** setComment(String c)

{

**this**.comment=c;

}

**public** String getComment()

{

**return** comment;

}

**public** **void** print()

{

System.***out***.println(**this**.label+"\t"+**this**.operator+"\t"+**this**.operand[3]);

}

//토큰의 operand에 연산이 있을 경우 연산을 해주는 함수

**public** **int** Math(**int** r\_cnt)

{

**if**(operand[0].contains("+"))

{

String[] opline=operand[0].split("+");

String op1\_str=opline[0];

String op2\_str=opline[1];

**int** op1=-1,op2=-1;

**if**(*isNum*(op1\_str))

{

op1=Integer.*parseInt*(op1\_str);

}

**else**

{

**for**(**int** k=0;k<*sym\_num*;k++)

{

**if**(*symbol*[k].isSymbol(operand[0], r\_cnt))

{

op1=*symbol*[k].addr;

**break**;

}

}

}

**if**(*isNum*(op2\_str))

{

op2=Integer.*parseInt*(op2\_str);

}

**else**

{

**for**(**int** k=0;k<*sym\_num*;k++)

{

**if**(*symbol*[k].isSymbol(operand[0], r\_cnt))

{

op2=*symbol*[k].addr;

**break**;

}

}

}

**if**(op1==-1 || op2==-1)

{

**return** 0;

}

**else**

{

**return** op1+op2;

}

}

**if**(operand[0].contains("-"))

{

String[] opline=operand[0].split("-");

String op1\_str=opline[0];

String op2\_str=opline[1];

**int** op1=-1,op2=-1;

**if**(*isNum*(op1\_str))

{

op1=Integer.*parseInt*(op1\_str);

}

**else**

{

**for**(**int** k=0;k<*sym\_num*;k++)

{

**if**(*symbol*[k].isSymbol(op1\_str, r\_cnt))

{

op1=*symbol*[k].addr;

**break**;

}

}

}

**if**(*isNum*(op2\_str))

{

op2=Integer.*parseInt*(op2\_str);

}

**else**

{

**for**(**int** k=0;k<*sym\_num*;k++)

{

**if**(*symbol*[k].isSymbol(op2\_str, r\_cnt))

{

op2=*symbol*[k].addr;

**break**;

}

}

}

**if**(op1==-1 || op2==-1)

{

**return** 0;

}

**else**

{

**return** op1-op2;

}

}

**else**

{

**return** 0;

}

}

}

**package** project;

**import** java.io.BufferedReader;

**import** java.io.BufferedWriter;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.io.PrintStream;

**import** java.io.PrintWriter;

**import** java.util.ArrayList;

**public** **class** Assembler {

**public** **static** instruction[] *inst* = **new** instruction[256]; //명령어 객체 생성

**static** Token\_unit[] *token* = **new** Token\_unit[500]; //토큰 객체 생성

**static** Symbol\_unit[] *symbol* = **new** Symbol\_unit[50]; //심볼 객체 생성

**static** Symbol\_unit[] *literal* = **new** Symbol\_unit[50]; //literal테이블 객체 생성 (심볼과 형식이 같음)

**static** Symbol\_unit[] *refer* = **new** Symbol\_unit[50]; //참조값 객체 생성 (심볼과 형식이 같음)

**static** Address[] *addr* = **new** Address[500]; //object code 객체 생성

**static** **int** *sym\_num* = 0; //심볼의 갯수

**static** **int** *token\_num* = 0; //토큰 갯수

**static** **int** *refer\_num* = 0; //참조값 갯수

**static** **int** *lit\_num* = 0; //literal 총 갯수

**static** **int** *loc\_num* = 0; //output 명령어 라인 총 갯수

**static** **int** *inst\_num*=0; //명령어 갯수

**static** **int** *LOCCTR*[] = **new** **int**[5000]; //LOCCTR 배열

**static** **int**[] *end*=**new** **int**[4]; //프로그램의 길이를 재기 위한 배열

**static** **int**[] *m\_record* = **new** **int**[100]; //수정할 값의 주소

**static** **int**[] *is\_word* = **new** **int**[100]; //연산을 사용했을 경우 연산의 갯수

**static** String[] *m\_str*=**new** String[15]; //수정할 값의 이름

**static** ArrayList<String> *literals* = **new** ArrayList<>(); //literal pool을 사용하기 위한 Arraylist

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

{

//

// 객체 배열의 초기화

//

**for**(**int** i=0; i<256; i++)

{

*inst*[i]=**new** instruction(**null**,0,0);

}

**for**(**int** i=0; i<500; i++)

{

*token*[i]=**new** Token\_unit(**null**,**null**,**null**,**null**);

*addr*[i]=**new** Address(**null**,0,0,0,0,0,0);

}

**for**(**int** i=0; i<50; i++)

{

*symbol*[i]=**new** Symbol\_unit(**null**,0,0);

*literal*[i]=**new** Symbol\_unit(**null**,0,0);

*refer*[i]=**new** Symbol\_unit(**null**,0,0);

}

BufferedReader bw=**null**;

BufferedReader tw=**null**;

PrintWriter wr=**null**;

//inst.data에서 기계 코드목록 파일을 읽어 기계어 목록 테이블을 생성

**try** {

bw = **new** BufferedReader(**new** FileReader("C:/Users/jinsu/workspace/Assembler\_114/src/project/inst.data"));

**for**(**int** i=0; i<256; i++)

{

String li;

li=bw.readLine();

**if**(li==**null**)

{

*inst\_num*=i;

**break**;

}

String[] line=li.split(" ");

*inst*[i].setName(line[0]);

*inst*[i].opcode=Integer.*parseInt*(line[1],16);

*inst*[i].format=Integer.*parseInt*(line[2]);

}

bw.close();

} **catch** (FileNotFoundException e) {

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

e.printStackTrace();

} **catch** (IOException e) {

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

e.printStackTrace();

}

**try** {

//

//소스코드를 읽어와 토큰단위로 파싱

//

tw = **new** BufferedReader(**new** FileReader("C:/Users/jinsu/workspace/Assembler\_114/src/project/input.txt"));

**for**(**int** i=0; i<5000; i++)

{

String li;

li=tw.readLine();

**if**(li==**null**)

{

*token\_num*=i;

**break**;

}

String[] line=li.split("\t");

*token*[i].setLabel(line[0]);

*token*[i].setOperator(line[1]);

**if** (line.length > 2) //null포인터가 안생기기 위해 길이에 따라 토큰테이블에 추가

{

**if** (line[2].contains(","))

{

String[] op\_line = line[2].split(",");

*token*[i].setOperand(op\_line[0], 0);

*token*[i].setOperand(op\_line[1], 1);

**if** (op\_line.length == 3)

{

*token*[i].setOperand(op\_line[2], 2);

}

*token*[i].setOperand(line[2], 3);

}

**else**

{

*token*[i].setOperand(line[2], 0);

*token*[i].setOperand(line[2], 3);

}

}

**if**(line.length==4) //null포인터가 안생기기 위해 길이에 따라 토큰테이블에 추가

{

*token*[i].setComment(line[3]);

}

}

tw.close();

} **catch** (FileNotFoundException e) {

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

e.printStackTrace();

} **catch** (IOException e) {

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

e.printStackTrace();

}

//

//input symbol : 심볼 테이블을 저장하고 LOCCTR값도 저장

//

**int** r\_cnt=0; //각 섹션의 순서

**int** loc=0; //LOCCTR에 저장할 값

**int** k=0; //operator이 명령어인지 아닌지 구분하는 값

**for**(**int** i=0;i<*token\_num*;i++,*loc\_num*++)

{

k=*search\_opcode*(*token*[i].operator); //명령어의 index값을 k에 저장

**if**(*token*[i].operator.equals("START") || *token*[i].operator.equals("CSECT"))

{

//pass2에서 쓰일 각 섹션의 길이를 구하여 end배열에 저장

**if**(*token*[i].operator.equals("CSECT"))

{

**int** j=i-1;

**for**(; j>0;j--)

{

**if** (*token*[j].operator.equals("EQU"))

{

**continue**;

}

**else**

{

**break**;

}

}

j+=*lit\_num*;

**if**(*LOCCTR*[j+2]!=0)

{

*end*[r\_cnt]=*LOCCTR*[j+2];

}

**else**

{

*end*[r\_cnt]=loc;

}

//출력값에서 각 섹션의 첫 명령어 줄에 섹션의 이름을 label에 추가

**for** (**int** jk = 0; jk < 5; jk++)

{

**int** jjk = *search\_opcode*(*token*[i + jk + 1].operator);

**if** (jjk >= 0)

{

*token*[i + jk + 2].label= *token*[i].label;

**break**;

}

}

}

loc=0;

*LOCCTR*[*loc\_num*]=loc;

r\_cnt++;

**continue**;

}

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

{

*LOCCTR*[*loc\_num*]=-1;

**continue**;

}

//EXTREF일 경우 refer테이블에 저장

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

{

*LOCCTR*[*loc\_num*]=-1;

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

{

**if**(*token*[i].operand[j]!=**null**)

{

*refer*[*refer\_num*].sub\_cnt=r\_cnt;

*refer*[*refer\_num*++].setName(*token*[i].operand[j]);

}

}

**continue**;

}

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

{

//LTORG를 만날 경우 literal pool을 비우고 literal테이블에 저장

*LOCCTR*[*loc\_num*]=-1;

*loc\_num*++;

**for**(**int** t=0; t<*literals*.size(); t++)

{

*LOCCTR*[*loc\_num*]=loc;

**if**(*literals*.get(t).startsWith("=X")) //16진수일 경우

{

*literal*[*lit\_num*].setName(*literals*.get(t));

*literal*[*lit\_num*].setAddr(loc);

*literal*[*lit\_num*].sub\_cnt=r\_cnt;

*lit\_num*++;

*loc\_num*++;

loc+=1;

}

**else** //character일 경우

{

**int** cnum=0;

cnum=*literals*.get(t).indexOf("'",3);

*literal*[*lit\_num*].setName(*literals*.get(t));

*literal*[*lit\_num*].setAddr(loc);

*literal*[*lit\_num*].sub\_cnt=r\_cnt;

*lit\_num*++;

*loc\_num*++;

//character길이에 따라 LOCCTR값 증가

**if**(cnum==6)

{

loc+=3;

}

**else** **if**(cnum==5)

{

loc+=2;

}

**else**

{

loc+=1;

}

}

}

*literals*.clear(); //literal pool 비우기

**continue**;

}

//RESW일 경우 심볼 테이블에 저장

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

{

*LOCCTR*[*loc\_num*]=loc;

*symbol*[*sym\_num*].setName(*token*[i].label);

*symbol*[*sym\_num*].setAddr(loc);

*symbol*[*sym\_num*].sub\_cnt=r\_cnt;

*sym\_num*++;

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

loc+=num;

**continue**;

}

//RESB일 경우 심볼 테이블에 저장

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

{

*LOCCTR*[*loc\_num*]=loc;

*symbol*[*sym\_num*].setName(*token*[i].label);

*symbol*[*sym\_num*].setAddr(loc);

*symbol*[*sym\_num*].sub\_cnt=r\_cnt;

*sym\_num*++;

**int** num=(Integer.*parseInt*(*token*[i].operand[0]));

loc+=num;

**continue**;

}

// EQU일 경우 그에 맞는 값을 구하여 심볼 테이블에 저장

**else** **if**(*token*[i].operator.equals("EQU"))

{

**if**(*token*[i].operand[0].equals("\*")) //현재주소값(\*)을 사용하는 경우 현재 주소값 저장

{

*LOCCTR*[*loc\_num*]=loc;

*symbol*[*sym\_num*].setName(*token*[i].label);

*symbol*[*sym\_num*].setAddr(loc);

*symbol*[*sym\_num*].sub\_cnt=r\_cnt;

*sym\_num*++;

**continue**;

}

**else**

{

//연산이 있을 경우 연산한 값을 구하여 저장

**if**(*token*[i].operand[0].contains("+") || *token*[i].operand[0].contains("-"))

{

*symbol*[*sym\_num*].setName(*token*[i].label);

**int** c=*token*[i].Math(r\_cnt);

*LOCCTR*[*loc\_num*]=c;

*symbol*[*sym\_num*].setAddr(c);

*symbol*[*sym\_num*].sub\_cnt=r\_cnt;

*sym\_num*++;

**continue**;

}

//연산이 없을 경우

**else**

{

//operand가 숫자가 아닐 경우 심볼테이블에서 operand와 같은 값을 찾아 주소값을 저장하고

//심볼테이블에 저장

**if**(!*isNum*(*token*[i].operand[0]))

{

**for** (**int** j = 0; j < *sym\_num*; j++)

{

**if** (*symbol*[j].isSymbol(*token*[i].operand[0], r\_cnt))

{

*symbol*[*sym\_num*].setName(*token*[i].label);

*LOCCTR*[*loc\_num*] = *symbol*[j].addr;

*symbol*[*sym\_num*].setAddr(*symbol*[j].addr);

*symbol*[*sym\_num*].sub\_cnt = r\_cnt;

*sym\_num*++;

**break**;

}

}

}

//숫자일 경우

**else**

{

**int** a=Integer.*parseInt*(*token*[i].operand[0]);

*symbol*[*sym\_num*].setName(*token*[i].label);

*symbol*[*sym\_num*].sub\_cnt=r\_cnt;

*LOCCTR*[*loc\_num*]=a;

*symbol*[*sym\_num*].setAddr(a);

*sym\_num*++;

**continue**;

}

}

}

}

//WORD일 경우 심볼테이블에 저장

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

{

*symbol*[*sym\_num*].setAddr(loc);

*symbol*[*sym\_num*].setName(*token*[i].label);

*LOCCTR*[*loc\_num*] = loc;

*symbol*[*sym\_num*].sub\_cnt=r\_cnt;

loc += 3;

*sym\_num*++;

**continue**;

}

//BYTE일 경우 16진수와 character을 구분하여 심볼테이블에 저장

**else** **if**(*token*[i].operator.equals("BYTE"))

{

**int** cnum=0;

**if** (*token*[i].operand[0].startsWith("X'")) //16진수일 경우

{

*symbol*[*sym\_num*].setAddr(loc);

*symbol*[*sym\_num*].setName(*token*[i].label);

*LOCCTR*[*loc\_num*] = loc;

*symbol*[*sym\_num*].sub\_cnt=r\_cnt;

loc += 1;

*sym\_num*++;

**continue**;

}

**else** //character일 경우

{

cnum=*token*[i].operand[0].indexOf("'",2);

*symbol*[*sym\_num*].setAddr(loc);

*symbol*[*sym\_num*].setName(*token*[i].label);

*LOCCTR*[*loc\_num*] = loc;

*symbol*[*sym\_num*].sub\_cnt=r\_cnt;

**if**(cnum==5)

{

loc+=3;

}

**else** **if**(cnum==4)

{

loc+=2;

}

**else**

{

loc+=1;

}

*sym\_num*++;

**continue**;

}

}

//END일 경우 리터럴 폴에 값이 있을경우 비우고 input symbol 반복문을 끝냄

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

{

*LOCCTR*[*loc\_num*]=-1;

*loc\_num*++;

**for**(**int** t=0; t<*literals*.size(); t++)

{

*LOCCTR*[*loc\_num*]=loc;

**if**(*literals*.get(t).startsWith("=X"))

{

*literal*[*lit\_num*].setName(*literals*.get(t));

*literal*[*lit\_num*].setAddr(loc);

*literal*[*lit\_num*].sub\_cnt=r\_cnt;

*lit\_num*++;

*loc\_num*++;

loc+=1;

}

**else**

{

**int** cnum=0;

cnum=*literals*.get(t).indexOf("'",3);

*literal*[*lit\_num*].setName(*literals*.get(t));

*literal*[*lit\_num*].setAddr(loc);

*literal*[*lit\_num*].sub\_cnt=r\_cnt;

*lit\_num*++;

*loc\_num*++;

**if**(cnum==6)

{

loc+=3;

}

**else** **if**(cnum==5)

{

loc+=2;

}

**else**

{

loc+=1;

}

}

}

*literals*.clear();

*end*[r\_cnt]=loc;

**break**;

}

**else**

{

**if**(k>=0) //operator이 명령어일 경우

{

**if**(*token*[i].label!=**null**) //label이 있을경우 심볼테이블에 저장

{

*symbol*[*sym\_num*].setAddr(loc);

*symbol*[*sym\_num*].setName(*token*[i].label);

*symbol*[*sym\_num*].sub\_cnt=r\_cnt;

*sym\_num*++;

}

//literal값을 사용하는 경우 literal pool에 저장

**if**(*token*[i].operand[0].startsWith("="))

{

**if**(!*literals*.contains(*token*[i].operand[0]))

{

*literals*.add(*token*[i].operand[0]);

}

}

//명령어의 형식에 따라 switch문 실행

**switch**(*inst*[k].format)

{

**case** 2: //명령어가 2형식일 경우

{

*LOCCTR*[*loc\_num*]=loc;

loc+=2;

**break**;

}

**case** 3: //명령어가 3형식일 경우

{

*LOCCTR*[*loc\_num*]=loc;

loc+=3;

**break**;

}

**case** 4: //4형식일 경우

{

*LOCCTR*[*loc\_num*]=loc;

loc+=4;

**break**;

}

**default**:

{

**break**;

}

}

}

}

}

// PASS 1

// data output

// object code를 구하고 저장하고 immediate data를 화면에 출력

//

r\_cnt=0;

**int** a=0; //object code의 객체 순서를 위한 변수

k=0;

**for**(**int** i=0; i<*token\_num*; i++,a++)

{

k=*search\_opcode*(*token*[i].operator);

**if**(*token*[i].operator.equals("START") || *token*[i].operator.equals("CSECT"))

{

**if**(*token*[i].operand[0]!=**null**)

{

System.***out***.format("%04X\t%s\t%s\t%s\r\n", *LOCCTR*[a],*token*[i].label,*token*[i].operator,*token*[i].operand[0]);

}

**else**

{

System.***out***.format("%04X\t%s\t%s\r\n", *LOCCTR*[a],*token*[i].label,*token*[i].operator);

}

r\_cnt++;

**continue**;

}

//object code가 없는 operator일 경우 LOCCTR값과 함꼐 출력

**else** **if**(*token*[i].operator.equals("RESW") || *token*[i].operator.equals("RESB") || *token*[i].operator.equals("EXTDEF")

|| *token*[i].operator.equals("EXTREF") || *token*[i].operator.equals("EQU"))

{

**if**(*LOCCTR*[a]==-1)

{

System.***out***.format("\t%s\t%s\t%s\r\n", *token*[i].label,*token*[i].operator,*token*[i].operand[3]);

}

**else**

{

System.***out***.format("%04X\t%s\t%s\t%s\r\n", *LOCCTR*[a],*token*[i].label,*token*[i].operator,*token*[i].operand[3]);

}

**continue**;

}

//END일 경우 literal값과 함께 출력

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

{

*addr*[a].r\_cnt=r\_cnt;

System.***out***.format("%s\t%s\t%s\r\n", *token*[i].label,*token*[i].operator,*token*[i].operand[0]);

a++;

**int** li=0;

String li\_str;

//literal값이 있을 경우 출력

**for**(**int** t=0; t<*lit\_num*; t++)

{

**if**(*literal*[t].sub\_cnt==r\_cnt)

{

li=*literal*[t].name.indexOf("'",3); //데이터 끝의 길이

li\_str=*literal*[t].name.substring(3, li); //데이터를 추출

*addr*[a].byte\_ob=li\_str; //추출한 데이터를 저장

*addr*[a].r\_cnt=r\_cnt;

*addr*[a].is\_r=0;

//character일 경우 길이에 따라 출력

**if**(*literal*[t].name.startsWith("=C'"))

{

**if**(li==6)

{

System.***out***.format("%04X\t\*\t%s\t\t%02X%02X%02X\r\n",*LOCCTR*[a],*literal*[t].name,

(**int**)*addr*[a].byte\_ob.charAt(0),(**int**)*addr*[a].byte\_ob.charAt(1),(**int**)*addr*[a].byte\_ob.charAt(2));

*addr*[a].format=3;

}

**else** **if**(li==5)

{

System.***out***.format("%04X\t\*\t%s\t\t%02X%02X\r\n",*LOCCTR*[a],*literal*[t].name,

(**int**)*addr*[a].byte\_ob.charAt(0),(**int**)*addr*[a].byte\_ob.charAt(1));

*addr*[a].format=2;

}

**else**

{

System.***out***.format("%04X\t\*\t%s\t\t%02X\r\n",*LOCCTR*[a],*literal*[t].name,

(**int**)*addr*[a].byte\_ob.charAt(0));

*addr*[a].format=1;

}

}

**else** //16진수일 경우

{

System.***out***.format("%04X\t\*\t%s\t\t%s\r\n", *LOCCTR*[a],*literal*[t].name,

*addr*[a].byte\_ob);

*addr*[a].format=1;

}

a++;

}

}

a--;

**break**;

}

//WORD일 경우 연산을 사용하는지 심볼을 사용하는지 숫자를 사용한 값인지 판단하여 출력

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

{

*addr*[a].r\_cnt=r\_cnt;

*addr*[a].format=3;

//연산이 있을 경우 연산한 값을 object code의 주소값에 저장

**if**(*token*[i].operand[0].contains("+") || *token*[i].operand[0].contains("-"))

{

**int** c=*token*[i].Math(r\_cnt);

**if**(c==0) //참조값을 사용하는 경우

{

*addr*[a].is\_r=1;

}

*addr*[a].loc=c;

}

**else**

{

**if**(*isNum*(*token*[i].operand[0])) //숫자일 경우

{

*addr*[a].loc=Integer.*parseInt*(*token*[i].operand[0]);

*addr*[a].is\_r = 0;

}

**else** //숫자가 아닐 경우 심볼테이블에서 검색하여 저장

{

**for** (**int** j = 0; j < *sym\_num*; j++)

{

**if** (*symbol*[j].isSymbol(*token*[i].operand[0], r\_cnt))

{

*addr*[a].loc = *symbol*[j].addr;

*addr*[a].is\_r = 0;

**break**;

}

**else**

{

*addr*[a].loc = 0;

*addr*[a].is\_r = 1;

}

}

}

}

//저장한 값을 출력

System.***out***.format("%04X\t%s\t%s\t%s\t%06X\r\n",*LOCCTR*[a],*token*[i].label,*token*[i].operator,*token*[i].operand[0],*addr*[a].loc );

**continue**;

}

//BYTE일 경우 literal처럼 데이터를 추출하여 저장하고 출력

**else** **if**(*token*[i].operator.equals("BYTE"))

{

**int** by=0;

String byte\_str;

by=*token*[i].operand[0].indexOf("'",2);

byte\_str=*token*[i].operand[0].substring(2, by); //데이터 추출

*addr*[a].byte\_ob=byte\_str; //추출한 데이터를 저장

*addr*[a].r\_cnt=r\_cnt;

*addr*[a].is\_r = 0;

//character일 경우 길이에 따라 출력

**if**(*addr*[a].byte\_ob.startsWith("C'"))

{

**if**(by==5)

{

System.***out***.format("%04X\t%s\t%s\t%s\t%02X%02X%02X\r\n",*LOCCTR*[a],*token*[i].label,*token*[i].operator,*token*[i].operand[0],

(**int**)*addr*[a].byte\_ob.charAt(0),(**int**)*addr*[a].byte\_ob.charAt(1),(**int**)*addr*[a].byte\_ob.charAt(2));

*addr*[a].format=3;

}

**else** **if**(by==4)

{

System.***out***.format("%04X\t%s\t%s\t%s\t%02X%02X\r\n",*LOCCTR*[a],*token*[i].label,*token*[i].operator,*token*[i].operand[0],

(**int**)*addr*[a].byte\_ob.charAt(0),(**int**)*addr*[a].byte\_ob.charAt(1));

*addr*[a].format=2;

}

**else**

{

System.***out***.format("%04X\t%s\t%s\t%s\t%02X\r\n",*LOCCTR*[a],*token*[i].label,*token*[i].operator,*token*[i].operand[0],

(**int**)*addr*[a].byte\_ob.charAt(0));

*addr*[a].format=1;

}

}

**else** //16진수일 경우

{

System.***out***.format("%04X\t%s\t%s\t%s\t%s\r\n", *LOCCTR*[a],*token*[i].label,*token*[i].operator,*token*[i].operand[0],

*addr*[a].byte\_ob);

*addr*[a].format=1;

}

**continue**;

}

//LTORG일 경우 비워진 literal값을 불러와 object code를 구해 출력

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

{

System.***out***.format("%s\t%s\r\n", *token*[i].label,*token*[i].operator);

a++;

*addr*[a].is\_r=2;

*addr*[a].r\_cnt=r\_cnt;

**int** li=0;

String li\_str;

**for**(**int** t=0; t<*lit\_num*; t++)

{

**if**(*literal*[t].sub\_cnt==r\_cnt)

{

li=*literal*[t].name.indexOf("'",3);

li\_str=*literal*[t].name.substring(3, li); //데이터 추출

*addr*[a].byte\_ob=li\_str; //추출한 데이터 저장

*addr*[a].r\_cnt=r\_cnt;

*addr*[a].is\_r=0;

//character일 경우 길이에 따라 출력

**if**(*literal*[t].name.startsWith("=C'"))

{

**if**(li==6)

{

System.***out***.format("%04X\t\*\t%s\t\t%02X%02X%02X\r\n",*LOCCTR*[a],*literal*[t].name,

(**int**)*addr*[a].byte\_ob.charAt(0),(**int**)*addr*[a].byte\_ob.charAt(1),(**int**)*addr*[a].byte\_ob.charAt(2));

*addr*[a].format=3;

}

**else** **if**(li==5)

{

System.***out***.format("%04X\t\*\t%s\t\t%02X%02X\r\n",*LOCCTR*[a],*literal*[t].name,

(**int**)*addr*[a].byte\_ob.charAt(0),(**int**)*addr*[a].byte\_ob.charAt(1));

*addr*[a].format=2;

}

**else**

{

System.***out***.format("%04X\t\*\t%s\t\t%02X\r\n",*LOCCTR*[a],*literal*[t].name,

(**int**)*addr*[a].byte\_ob.charAt(0));

*addr*[a].format=1;

}

}

**else** //16진수일 경우

{

System.***out***.format("%04X\t\*\t%s\t\t%s\r\n", *LOCCTR*[a],*literal*[t].name,

*addr*[a].byte\_ob);

*addr*[a].format=1;

}

a++;

}

}

**continue**;

}

**else** //operator이 명령어일 경우 형식에 따라 object code를 구하고 화면출력

{

k=*search\_opcode*(*token*[i].operator); //명령어의 인덱스를 k에 저장

**if**(k>=0) //k가 -1일 경우를 제외

{

**switch**(*inst*[k].format)

{

**case** 2: //2형식일 경우 레지스터의 값을 구하여 opcode와 함께 저장

{

**int** r1=0;

**int** r2=0;

**if**(*token*[i].operand[3].contains(",")) //레지스터가 2개인 경우

{

r1 = *search\_register*(*token*[i].operand[0]); //무슨 레지스터인지 검사

r2 = *search\_register*(*token*[i].operand[1]); //무슨 레지스터인지 검사

*addr*[a].loc = r1 \* 10;

*addr*[a].loc += r2;

*addr*[a].format = 2;

*addr*[a].op=*inst*[k].opcode;

*addr*[a].r\_cnt=r\_cnt;

}

**else** //레지스터가 1개인 경우

{

r1 = *search\_register*(*token*[i].operand[0]); //무슨 레지스터인지 검사

*addr*[a].loc = r1 \* 10;

*addr*[a].format = 2;

*addr*[a].op=*inst*[k].opcode;

*addr*[a].r\_cnt=r\_cnt;

}

*addr*[a].is\_r=0;

//화면 출력

**if**(*token*[i].label==**null**)

{

System.***out***.format("%04X%s\t%s\t%s\t%02X%02d\r\n", *LOCCTR*[a],*token*[i].label,*token*[i].operator,*token*[i].operand[3],

*addr*[a].op,*addr*[a].loc);

}

**else**

{

System.***out***.format("%04X\t%s\t%s\t%s\t%02X%02d\r\n", *LOCCTR*[a],*token*[i].label,*token*[i].operator,*token*[i].operand[3],

*addr*[a].op,*addr*[a].loc);

}

**break**;

}

**case** 3: //3형식인 경우 주소값은 pc값을 이용한 상대주소값을 저장하고 주소지정방식에 따른 object code를 구함

{

**if**(*token*[i].operator.equals("RSUB")) //RSUB인 경우 주소는 0을 넣음

{

*addr*[a].loc=0;

*addr*[a].format=3;

*addr*[a].op=*inst*[k].opcode+3;

*addr*[a].xbpe=0;

*addr*[a].is\_r=0;

*addr*[a].r\_cnt=r\_cnt;

}

//리터럴 값을 사용한 경우 literal테이블에서 찾아서 저장

**else** **if**(*token*[i].operand[0].startsWith("="))

{

**for**(**int** t=0; t<*lit\_num*; t++)

{

**if** (*literal*[t].name.equals(*token*[i].operand[0]))

{

**if** (*literal*[t].sub\_cnt == r\_cnt)

{

*addr*[a].loc = *literal*[t].addr - *LOCCTR*[a + 1];

**if** (*addr*[a].loc < 0)

{

*addr*[a].loc = *addr*[a].loc & 0XFFF;

}

*addr*[a].is\_r=0;

**break**;

}

}

}

*addr*[a].format=3;

*addr*[a].op=*inst*[k].opcode+3;

*addr*[a].xbpe=2;

*addr*[a].r\_cnt=r\_cnt;

}

//즉시주소지정방식인 경우

**else** **if**(*token*[i].operand[0].startsWith("#"))

{

String im=*token*[i].operand[0].substring(1, *token*[i].operand[0].length());

**if**(*isNum*(im)) //숫자일 경우 값을 그대로 주소값에 저장

{

*addr*[a].loc=Integer.*parseInt*(im);

*addr*[a].is\_r=0;

*addr*[a].op=*inst*[k].opcode+1;

*addr*[a].xbpe=0;

*addr*[a].format=3;

*addr*[a].r\_cnt=r\_cnt;

}

**else** //숫자가 아닌 경우 심볼테이블을 검색하여 주소값 저장

{

**for**(**int** s=0; s<*sym\_num*; s++)

{

**if**(*symbol*[s].isSymbol(im, r\_cnt))

{

*addr*[a].loc=*symbol*[s].addr-*LOCCTR*[a+1];

**if**(*addr*[a].loc<0) //음수인 경우 비트연산하여 저장

{

*addr*[a].loc=*addr*[a].loc & 0XFFF;

}

*addr*[a].is\_r=0;

**break**;

}

**else**

{

*addr*[a].loc=0;

*addr*[a].is\_r=1;

}

}

*addr*[a].op=*inst*[k].opcode+1;

*addr*[a].xbpe=2;

*addr*[a].format=3;

*addr*[a].r\_cnt=r\_cnt;

}

}

//간접주소지정방식인 경우 PC값을 이용하여 저장

**else** **if**(*token*[i].operand[0].startsWith("@"))

{

String im=*token*[i].operand[0].substring(1, *token*[i].operand[0].length());

**for**(**int** s=0; s<*sym\_num*; s++)

{

**if**(*symbol*[s].isSymbol(im, r\_cnt))

{

*addr*[a].loc=*symbol*[s].addr-*LOCCTR*[a+1];

**if**(*addr*[a].loc<0)

{

*addr*[a].loc=*addr*[a].loc & 0XFFF;

}

*addr*[a].is\_r=0;

**break**;

}

**else**

{

*addr*[a].loc=0;

*addr*[a].is\_r=1;

}

}

*addr*[a].op=*inst*[k].opcode+2;

*addr*[a].xbpe=2;

*addr*[a].format=3;

*addr*[a].r\_cnt=r\_cnt;

}

//직접주소지정방식인 경우

**else**

{

**if**(*token*[i].operand[3].contains(",")) //X레지스터를 사용하는 경우

{

*addr*[a].xbpe=10;

}

**else**

{

*addr*[a].xbpe=2;

}

**for**(**int** s=0; s<*sym\_num*; s++) //operand를 심볼테이블에서 찾아 주소값을 구함

{

**if**(*symbol*[s].isSymbol(*token*[i].operand[0], r\_cnt))

{

*addr*[a].loc=*symbol*[s].addr-*LOCCTR*[a+1];

**if**(*addr*[a].loc<0) //음수인 경우 비트연산하여 저장

{

*addr*[a].loc=*addr*[a].loc & 0XFFF;

}

*addr*[a].is\_r=0;

**break**;

}

**else** //심볼테이블에 없는 경우 참조값인 경우 주소값은 0을 저장

{

*addr*[a].loc=0;

*addr*[a].is\_r=1;

}

}

*addr*[a].op=*inst*[k].opcode+3;

*addr*[a].r\_cnt=r\_cnt;

*addr*[a].format=3;

}

//저장한 값을 출력

**if**(*token*[i].label==**null**)

{

System.***out***.format("%04X%s\t%s\t%s\t%02%1X%03X\r\n", *LOCCTR*[a],*token*[i].label,*token*[i].operator,*token*[i].operand[3],

*addr*[a].op,*addr*[a].xbpe,*addr*[a].loc);

}

**else**

{

System.***out***.format("%04X\t%s\t%s\t%s\t%02X%1X%03X\r\n", *LOCCTR*[a],*token*[i].label,*token*[i].operator,*token*[i].operand[3],

*addr*[a].op,*addr*[a].xbpe,*addr*[a].loc);

}

**break**;

}

//4형식인 경우 3형식과 동일하되 주소값은 20비트를 사용(5자리로 출력)

**case** 4:

{

**if**(*token*[i].operand[0].startsWith("="))

{

**for**(**int** t=0; t<=*lit\_num*; t++)

{

**if** (*literal*[t].equals(*token*[i].operand[0]))

{

**if** (*literal*[t].sub\_cnt == r\_cnt)

{

*addr*[a].loc = *literal*[t].addr;

*addr*[a].is\_r=0;

**break**;

}

}

}

*addr*[a].format=4;

*addr*[a].op=*inst*[k].opcode+3;

*addr*[a].xbpe=1;

*addr*[a].r\_cnt=r\_cnt;

}

**else** **if**(*token*[i].operand[0].startsWith("#"))

{

String im=*token*[i].operand[0].substring(1, *token*[i].operand[0].length());

**if**(*isNum*(im))

{

*addr*[a].loc=Integer.*parseInt*(im);

*addr*[a].is\_r=0;

*addr*[a].op=*inst*[k].opcode+1;

*addr*[a].xbpe=1;

*addr*[a].format=4;

*addr*[a].r\_cnt=r\_cnt;

}

**else**

{

**for**(**int** s=0; s<*sym\_num*; s++)

{

**if**(*symbol*[s].isSymbol(im, r\_cnt))

{

*addr*[a].loc=*symbol*[s].addr;

*addr*[a].is\_r=0;

**break**;

}

**else**

{

*addr*[a].loc=0;

*addr*[a].is\_r=1;

}

}

*addr*[a].op=*inst*[k].opcode+1;

*addr*[a].xbpe=1;

*addr*[a].format=4;

*addr*[a].r\_cnt=r\_cnt;

}

}

**else** **if**(*token*[i].operand[0].startsWith("@"))

{

String im=*token*[i].operand[0].substring(1, *token*[i].operand[0].length());

**for**(**int** s=0; s<*sym\_num*; s++)

{

**if**(*symbol*[s].isSymbol(im, r\_cnt))

{

*addr*[a].loc=*symbol*[s].addr;

*addr*[a].is\_r=0;

**break**;

}

**else**

{

*addr*[a].loc=0;

*addr*[a].is\_r=1;

}

}

*addr*[a].op=*inst*[k].opcode+2;

*addr*[a].xbpe=1;

*addr*[a].format=4;

*addr*[a].r\_cnt=r\_cnt;

}

**else**

{

**if**(*token*[i].operand[3].contains(","))

{

*addr*[a].xbpe=9;

}

**else**

{

*addr*[a].xbpe=1;

}

**for**(**int** s=0; s<*sym\_num*; s++)

{

**if**(*symbol*[s].isSymbol(*token*[i].operand[0], r\_cnt))

{

*addr*[a].loc=*symbol*[s].addr;

*addr*[a].is\_r=0;

**break**;

}

**else**

{

*addr*[a].loc=0;

*addr*[a].is\_r=1;

}

}

*addr*[a].op=*inst*[k].opcode+3;

*addr*[a].r\_cnt=r\_cnt;

*addr*[a].format=4;

}

//위는 3형식과 거의 동일하나 출력할 경우 주소값에는 5자리로 출력

**if**(*token*[i].label==**null**)

{

System.***out***.format("%04X%s\t%s\t%s\t%02%1X%05X\r\n", *LOCCTR*[a],*token*[i].label,*token*[i].operator,*token*[i].operand[3],

*addr*[a].op,*addr*[a].xbpe,*addr*[a].loc);

}

**else**

{

System.***out***.format("%04X\t%s\t%s\t%s\t%02X%1X%05X\r\n", *LOCCTR*[a],*token*[i].label,*token*[i].operator,*token*[i].operand[3],

*addr*[a].op,*addr*[a].xbpe,*addr*[a].loc);

}

**break**;

}

**default** :

**break**;

}

}

}

}

//

// pass 2

// 어셈블리 코드를 기계어 코드로 바꾸기 위한 패스2 과정을 실행

// object program을 output\_114.txt에 저장

r\_cnt=0;

a=0;

**int** a2=0; //LTORG의 개행을 위한 또다른 카운트변수

k=0;

**int** max = 30; //T레코드의 길이 제한하기위한 변수

**int** real\_cnt = 0; //T레코드의 길이 변수

**int** count = 0; //T레코드의 개행을 위한 변수

**int** enter = 0; //LTROG만나면 T레코드의 개행을 위한 변수

**int** start\_cnt = 0; //메인섹션을 위한 변수

**int** m\_cnt = 0; //M레코드이 갯수

**try** {

wr = **new** PrintWriter("C:/Users/jinsu/workspace/Assembler\_114/src/project/output\_114.txt");

**for**(**int** i=0; i<*token\_num*; i++,a++)

{

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

{

r\_cnt++;

start\_cnt=1;

wr.printf("H%s\t%06X%06X\r\n", *token*[i].label, Integer.*parseInt*(*token*[i].operand[0]), *end*[r\_cnt]);

**continue**;

}

**else** **if**(*token*[i].operator.equals("CSECT"))

{

//섹션에 넘어갈 때 마다 M레코드를 추가하여 파일에 출력

**if** (m\_cnt != 0)

{

wr.println();

**for** (**int** m = 0; m < m\_cnt; m++) //수정 레코드를 넣어줌

{

**if** (*is\_word*[m] == 1)

{

wr.printf("M%06X06+%s\r\n", *m\_record*[m], *m\_str*[m]);

}

**else** **if** (*is\_word*[m] == 2)

{

wr.printf("M%06X06%s\r\n", *m\_record*[m], *m\_str*[m]);

}

**else** **if** (*is\_word*[m] == 3)

{

wr.printf("M%06X06%s\r\n", *m\_record*[m], *m\_str*[m]);

}

**else**

{

wr.printf("M%06X05+%s\r\n", *m\_record*[m], *m\_str*[m]);

}

}

}

**if** (r\_cnt == 1) //메인섹션이 끝난경우의 E레코드 출력

{

wr.printf("E%06d\r\n\r\n", *LOCCTR*[0]);

r\_cnt++;

}

**else** //메인섹션이 아닌경우 E레코드 출력

{

wr.printf("E\r\n\r\n");

r\_cnt++;

}

count = 0;

wr.printf("H%s\t%06X%06X\r\n", *token*[i].label, 0, *end*[r\_cnt]); //나눠진 섹션의 헤더레코드

a2 = *LOCCTR*[a];

start\_cnt = 1;

**for** (**int** j = 0; j < m\_cnt; j++)

{

*is\_word*[j] = 0;

}

m\_cnt = 0;

**continue**;

}

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

{

**int** loc1 = -1, loc2 = -1, loc3 = -1;

**for** (**int** p = 0; p < *sym\_num*; p++)

{

**if** (*token*[i].operand[0].equals(*symbol*[p].name))

{

loc1 = *symbol*[p].addr;

}

**if** (*token*[i].operand[1].equals(*symbol*[p].name))

{

loc2 = *symbol*[p].addr;

}

**if** (*token*[i].operand[2].equals(*symbol*[p].name))

{

loc3 = *symbol*[p].addr;

}

}

**if** (*token*[i].operand[2] != **null**)

{

wr.printf("D%s%06X%s%06X%s%06X\r\n", *token*[i].operand[0], loc1, *token*[i].operand[1], loc2,

*token*[i].operand[2], loc3);

}

**else** **if** (*token*[i].operand[1] != **null**)

{

wr.printf("D%s%06X%s%06X\r\n", *token*[i].operand[0], loc1, *token*[i].operand[1], loc2);

}

**else**

{

wr.printf("D%s%06X\r\n", *token*[i].operand[0], loc1);

}

**continue**;

}

//R레코드 추가

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

{

**if** (r\_cnt == 1)

{

wr.print("R");

**for** (**int** r = 0; r < *refer\_num*; r++)

{

**if**(*refer*[r].sub\_cnt==r\_cnt)

{

wr.print(*refer*[r].name+" ");

}

**else**

{

**break**;

}

}

wr.println();

}

**else**

{

wr.print("R");

**for** (**int** r = 0; r < *refer\_num*; r++)

{

**if**(*refer*[r].sub\_cnt==r\_cnt)

{

wr.print(*refer*[r].name);

}

**else**

{

;

}

}

wr.println();

}

**continue**;

}

**else** **if**(*token*[i].operator.equals("RESB") || *token*[i].operator.equals("RESW") || *token*[i].operator.equals("EQU"))

{

**continue**;

}

//나머지 operator들

**else**

{

**if** (*addr*[a].is\_r == 1) //수정할 object code일 경우

{

//수정할 object code의 명령어가 word일 경우

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

{

**if** (*token*[i].operand[0].contains("-")) //-일 경우 -와 같이 추가

{

String[] str=*token*[i].operand[0].split("-");

*m\_record*[m\_cnt] = *LOCCTR*[a];

*is\_word*[m\_cnt] = 1;

*m\_str*[m\_cnt]=str[0];

m\_cnt++;

*m\_record*[m\_cnt] = *LOCCTR*[a];

*is\_word*[m\_cnt] = 2;

*m\_str*[m\_cnt]="-".concat(str[1]);

m\_cnt++;

}

**else** **if** (*token*[i].operand[0].contains("+")) //+일 경우 +와 같이 추가

{

String[] str=*token*[i].operand[0].split("+");

*m\_record*[m\_cnt] = *LOCCTR*[a];

*is\_word*[m\_cnt] = 1;

*m\_str*[m\_cnt]=str[0];

m\_cnt++;

*m\_record*[m\_cnt] = *LOCCTR*[a];

*is\_word*[m\_cnt] = 2;

*m\_str*[m\_cnt]="+".concat(str[1]);

m\_cnt++;

}

**else** //연산이 아닐 경우

{

*m\_record*[m\_cnt] = *LOCCTR*[a];

*is\_word*[m\_cnt] = 1;

*m\_str*[m\_cnt]=*token*[i].operand[0];

m\_cnt++;

}

}

//수정할 object code 추가

**else**

{

*m\_record*[m\_cnt] = *LOCCTR*[a] + 1;

*m\_str*[m\_cnt]=*token*[i].operand[0];

m\_cnt++;

}

}

**if**(count==0) //개행할 떄

{

**for** (**int** c = a; c < a + 12; c++)

{

**if** (real\_cnt + 3 > max || *addr*[c].is\_r == 2 || *addr*[c].r\_cnt != r\_cnt)

{

**break**;

}

**else**

{

real\_cnt += *addr*[c].format; //T레코드의 길이를 계산

}

}

wr.print("T");

**if** (start\_cnt == 1)

{

wr.printf("%06X%02X",*LOCCTR*[a],real\_cnt); //T레코드의 첫 주소와 길이 출력

start\_cnt = 0;

}

**else**

{

wr.printf("%06X%02X",*LOCCTR*[a],real\_cnt);

}

real\_cnt = 0;

}

//OPERATOR가 BYTE일 경우

**if** (*token*[i].operator.equals("BYTE"))

{

//character일 경우 길이에 맞게 출력

**if** (*token*[i].operand[0].startsWith("C'"))

{

**int** len=*addr*[a].byte\_ob.length();

**if**(len==3)

{

wr.printf("%02X%02X%02X",(**int**)*addr*[a].byte\_ob.charAt(0),(**int**)*addr*[a].byte\_ob.charAt(1),

(**int**)*addr*[a].byte\_ob.charAt(2));

count += 3;

}

**else** **if**(len==2)

{

wr.printf("%02X%02X",(**int**)*addr*[a].byte\_ob.charAt(0),(**int**)*addr*[a].byte\_ob.charAt(1));

count += 2;

}

**else**

{

wr.printf("%02X",(**int**)*addr*[a].byte\_ob.charAt(0));

count += 1;

}

}

//16진수일 경우

**else** **if** (*token*[i].operand[0].startsWith("X'"))

{

wr.printf("%s", *addr*[a].byte\_ob);

count += 1;

}

}

//OPERATOR가 word일 경우

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

{

wr.printf("%06X", *addr*[a].loc);

count += 3;

}

//LTORG이 나올경우

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

{

a++;

enter = 1;

**int** lk = 0;

**for** (**int** j = 0; j < *lit\_num*; j++)

{

**if** (r\_cnt == *literal*[lk].sub\_cnt)

{

a2 = a; //개행을 위한 변수 저장

**if** (*literal*[lk].name.startsWith("=C"))

{

**if**(*addr*[a].byte\_ob.length()==3)

{

count += 3;

}

**else** **if**(*addr*[a].byte\_ob.length()==2)

{

count+=2;

}

**else**

{

count+=1;

}

lk++;

}

**else** **if** (*literal*[lk].name.startsWith("=X"))

{

count += 1;

lk++;

}

a++;

}

**else**

{

lk++;

**continue**;

}

}

}

**else**

{

;

}

k = *search\_opcode*(*token*[i].operator); //명령어 index 저장

**if**(k>=0)

{

**switch**(*inst*[k].format)

{

**case** 2: //2형식일 경우

{

wr.printf("%02X%02d",*addr*[a].op,*addr*[a].loc);

count += 2;

**break**;

}

**case** 3: //3형식일 경우

{

wr.printf("%02X%1X%03X",*addr*[a].op, *addr*[a].xbpe, *addr*[a].loc);

count += 3;

**break**;

}

**case** 4: //4형식일 경우

{

wr.printf("%02X%1X%05X",*addr*[a].op, *addr*[a].xbpe, *addr*[a].loc);

count += 4;

**break**;

}

**default**:

{

**break**;

}

}

}

//LTORG가 나올경우 T 레코드를 개행해줌

**if** (enter == 1)

{

enter = 0;

count = 0;

wr.println();

**for** (**int** c = a2; c < a2 + 12; c++)

{

**if** (real\_cnt + 3 > max || *addr*[c].r\_cnt != r\_cnt)

{

**break**;

}

**else**

{

real\_cnt += *addr*[c].format; //T레코드의 길이를 구해줌

}

}

wr.print("T");

wr.printf("%06X%02X", *LOCCTR*[a2], real\_cnt); //T레코드의 처음 값과 길이를 출력

real\_cnt = 0;

**int** jk = 0;

**for** (**int** j = 0; j < *lit\_num*; j++)

{

**if** (r\_cnt == *literal*[jk].sub\_cnt)

{

**if** (*literal*[jk].name.startsWith("=C"))

{

**if**(*addr*[a2].byte\_ob.length()==3)

{

wr.printf("%02X%02X%02X",(**int**)*addr*[a2].byte\_ob.charAt(0),(**int**)*addr*[a2].byte\_ob.charAt(1),

(**int**)*addr*[a2].byte\_ob.charAt(2));

count += 3;

}

**else** **if**(*addr*[a2].byte\_ob.length()==2)

{

wr.printf("%02X%02X",(**int**)*addr*[a2].byte\_ob.charAt(0),(**int**)*addr*[a2].byte\_ob.charAt(1));

count+=2;

}

**else**

{

wr.printf("%02X",(**int**)*addr*[a2].byte\_ob.charAt(0));

count+=1;

}

jk++;

}

**else** **if** (*literal*[jk].name.startsWith("=X"))

{

wr.printf("%s", *addr*[a2].byte\_ob);

count += 1;

jk++;

}

}

**else**

{

**break**;

}

}

}

//T레코드의 바이트 수가 30을 넘을경우

**else**

{

**if** (count + 3 > max)

{

a2 = a;

count = 0;

wr.println();

}

}

//프로그램의 마지막

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

{

a++;

//literal이 있을 경우 출력

**for** (**int** j = 0; j < *lit\_num*; j++)

{

**if** (r\_cnt == *literal*[j].sub\_cnt)

{

**if** (*literal*[j].name.startsWith("=C"))

{

**if**(*addr*[a].byte\_ob.length()==3)

{

wr.printf("%02X%02X%02X",(**int**)*addr*[a].byte\_ob.charAt(0),(**int**)*addr*[a].byte\_ob.charAt(1),

(**int**)*addr*[a].byte\_ob.charAt(2));

count += 3;

}

**else** **if**(*addr*[a].byte\_ob.length()==2)

{

wr.printf("%02X%02X",(**int**)*addr*[a].byte\_ob.charAt(0),(**int**)*addr*[a].byte\_ob.charAt(1));

count+=2;

}

**else**

{

wr.printf("%s",*addr*[a].byte\_ob);

count+=1;

}

}

**else** **if** (*literal*[j].name.startsWith("=X"))

{

wr.printf("%s", *addr*[a].byte\_ob);

count += 1;

}

}

**else**

{

**continue**;

}

}

//마지막 M레코드 추가

**if** (m\_cnt != 0)

{

wr.println();

**for** (**int** m = 0; m < m\_cnt; m++)

{

**if** (*is\_word*[m] == 1)

{

wr.printf("M%06X06+%s\r\n", *m\_record*[m], *m\_str*[m]);

}

**else**

{

wr.printf("M%06X05+%s\r\n", *m\_record*[m], *m\_str*[m]);

}

}

}

wr.print("E");

}

}

}

wr.close();

} **catch** (FileNotFoundException e) {

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

e.printStackTrace();

}

}

//문자열을 입력받아 레지스터의 number를 반환해주는 함수

//반환 : 레지스터의 number

**public** **static** **int** search\_register(String str)

{

**if**(str.equals("A"))

{

**return** 0;

}

**if**(str.equals("X"))

{

**return** 1;

}

**if**(str.equals("L"))

{

**return** 2;

}

**if**(str.equals("B"))

{

**return** 3;

}

**if**(str.equals("S"))

{

**return** 4;

}

**if**(str.equals("T"))

{

**return** 5;

}

**if**(str.equals("F"))

{

**return** 6;

}

**if**(str.equals("PC"))

{

**return** 8;

}

**if**(str.equals("SW"))

{

**return** 9;

}

**return** 0;

}

//문자열을 입력받아 문자열의 데이터가 숫자인지 판별하는 함수

//반환 : 숫자일 경우 : true 아닐 경우 : false

**public** **static** **boolean** isNum(String str)

{

**char** check;

**if** (str.equals(""))

{

**return** **false**;

}

**for** (**int** i = 0; i < str.length(); i++)

{

check = str.charAt(i);

**if** (check < 48 || check > 58)

{

**return** **false**;

}

}

**return** **true**;

}

//문자열을 입력받아 명령어인지 구분해주는 함수

//반환 : 명령어일 경우 : 명령어의 index 아닐 경우 : -1

**public** **static** **int** search\_opcode(String str)

{

**int** is=-1;

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

{

**if**(str.startsWith("+"))

{

String str2=str.substring(1,str.length());

**if**(*inst*[i].name.equals(str2))

{

*inst*[i].format=4;

**return** i;

}

}

**else** **if**(*inst*[i].name.equals(str))

{

**if**(*inst*[i].format==4)

{

*inst*[i].format=3;

}

**return** i;

}

}

**return** is;

}

}