과목 명: 시스템프로그래밍

담당 교수 명: 박 운 상

<<Assignment 1>>

**서강대학교 컴퓨터학과**

**[학번] 20151607**

**[이름] 정재훈**

목 차

1. 프로그램 개요 3

2. 프로그램 설명 3

2.1 프로그램 흐름도 3

3. 모듈 정의 3

3.1 모듈 이름 : main() 3

3.1.1 기능 3

3.1.2 사용 변수 4

3.2 모듈 이름: read\_opcode() 4

3.2.1 기능 4

3.2.2 사용 변수 4

3.3 모듈이름: refine\_argument() 4

3.3.1 기능 4

3.3.2 사용변수 4

3.4 모듈이름: is\_command() 4

3.4.1 기능 4

3.4.2 사용변수 5

3.5 모듈이름: extract\_parameter() 5

3.5.1 기능 5

3.5.2 사용변수 5

3.6 모듈이름: command() 5

3.6.1 기능 5

3.6.2 사용변수 6

3.7 모듈이름: error\_print() 6

3.7.1 기능 6

3.7.2 사용변수 6

4. 전역 변수 정의 6

4.1 argument관련 전역변수 6

4.2 opcode hash 관련 전역변수 6

4.3 history 관련 전역변수 6

4.4 memory 관련 전역변수 7

4.5 error 관련 전역변수 7

4.6 flag 관련 전역변수 7

5. 코드 8

# 프로그램 개요

이 프로그램은 앞으로 구현하게 될 SIC/XE머신을 구현하기 위한 전 단계로서 어셈블러, 링크, 로더들을 실행하게 될 셸**(shell)**과 컴파일을 통해서 만들어진 object코드가 적재되고 실행될 **메모리공간**과 mnemonic (ADD, COMP, FLOAT, etc …)을 opcode값으로 변환하는 **OPCODE 테이블**과 **관련 명령어**들을 구현하는 프로그램입니다.

# 프로그램 설명

## 프로그램 흐름도

1.Opcode hash table

Read\_opcode()

2. refined argument

Refine\_

Argument

Main()

그림 1> 프로그램 흐름도

3. refined argument

Extract\_parameter()

4. parameters

Is\_command

()

6. Error code

Print\_error()

find\_

mnemonic()

5. Parsed parameters

Linkloader()

Parse\_

argument()

Simulator()

Run()

command()

linkloader()

assembler()

Hex\_to\_int()

Setbreak()

Progaddr()

# 모듈 정의

## 모듈 이름 : main() //20151607.c

### 기능

Read\_opcode() 를 호출하여 opcode hash function을 저장한다.

그 후 사용자로부터 명령어를 계속 받아 각 command에 따라 is\_command()를 수행시켜 맞는 command를 찾게된다.

### 사용 변수

Char arg[INPUT\_SIZE] : 사용자의 입력을 받아 arg에 저장한다.

Char tmp\_arg[INPUT\_SIZE] : 사용자의 입력을 임시 저장해두는곳, 이는 history command와 refine\_argument() 위해 존재한다.

## 모듈 이름: read\_opcode() //opcode.c

### 기능

opcode.txt 파일을 읽어들여서 이를 mnemonic에 따라 hash number을 구성하고

각 hash number에 맞는 hash table에 linked list 형식으로 추가한다

.

read\_opcode\_file() : txt파일을 읽어 새로운 node를 구성한다.

add\_hash() : 생성된 node를 hash table에 linked list로 추가시킨다.

hash\_function() : string에 따라 고유 hash number을 리턴한다.

### 사용변수

FILE\* fp : opcode.txt filepointer

## 모듈 이름: refine\_argument() //argument.c

### 기능

refine\_argument()

:사용자가 입력한 명령어를 프로그램이 읽기 좋게 정제하는 과정이다.

사용자가 입력한 \t, \n, spacebar 를 지우고 comma가 어떤 문자에 붙어서 존재한다면 떨어뜨려준다.

예를 들어 dump AA, BB \n 이 있다면 다음과 같은 형식으로 저장한다.

Dump’\0’AA’\0’,’\0’BB’\0’’\0’

끝을 구분하기 위해서 마지막에는 null 문자가 두개 이상 온다.

### 사용 변수

Char pre : 정제하기 위해 필요한 previous character. Current character와 비교하여 각 경우를 파악한다.

## 모듈이름: is\_command() // argument.c

### 기능

Is\_command 모듈에서는 각 명령어마다 함수가 존재한다.

Is\_help(), is\_history(), is\_quit(), is\_dir(), is\_type() //shell 관련 명령어

Is\_dump(), is\_fill(), is\_edit(), is\_reset() //memory 관련 명령어

Is\_opcodelist(), is\_opcode\_mnemonic() //opcode 관련 명령어

Is\_assemble(), is\_symbol() //assemble 관련 명령어

Is run(), is\_bp(), is\_progaddr() //linkloader 관련 명령어

정제된 Argument를받아 이 agument를parsing한다. Parsing된 값이 해당 command와 일치하는지 확인한다. 일치하지 않으면 0을 반환한다.

만약 일치한다면 해당 명령어의 문법에 따라 parsing과 hex\_to\_int를 진행하여 문법적 오류가 없는지 확인한다. 만약 있다면 error\_message를 출력한다.

문법적 오류가 없이 완벽하다면 이에 해당하는 command 모듈을 실행하고 history에 저장한다.

### 사용변수

Char \*parse : parsing된 문자열의 시작 주소

Int len : parsing 된 문자열의 길이. Over argument인지 Lack argument인지 판별하는데에 쓰인다.

Int start,end,addr : 메모리 주소를 저장하는 변수 //is\_dump(), is\_fill(),is\_edit()

Int value : 메모리 정보를 대체할 값으로 사용되는 변수 //is\_fill(), is\_edit()

## 모듈이름: extract\_parameter() //argument.c, opcode.c assemble.c

### 기능

### Extract\_parameter 모듈은 정제된 argument에서 command에 필요한 변수들을 추출하기위해 존재하는 모듈이다. 이러한 추출을 위해 이 모듈에는 3가지 함수가 존재한다.

### Parsing\_argument() : refined argument에서 문장을 하나씩 읽어온다. //argument.c

### Hex\_to\_int() : char형 hex code로 이루어져있는 것을 int형으로 바꿔준다. //argument.c

### Find\_mnemonic() : 입력한 mnemonic에 해당하는게 hash table에 있는지 확인해준다. 있으면 해당 노드의 pointer을 반환한다. //opcode.c

Find\_symbol() : 입력한 symbol에 해당하는게 hash table에 있는지 확인해준다 있으면 해당 노드의 포인터를 반환한다.

추출을 위한 포맷에 맞지 않는다면 에러코드를 리턴한다.

### 사용변수

Int len : 해당 string의 길이를 저장한다.

Char hex[] : char형으로 된 hex code를 저장한다.

Int hash\_number : 해당 string의 hash\_number을 저장한다.

## 

## 모듈이름: command() //shell.c, memory.c, opcode.c assemble.c

### 기능

각 command에 해당하는 기능을 수행한다.

Command는 3가지 종류로 나뉘는데

1. shell command //shell.c

cmd\_help() : 명령어 목록을 출력한다.

cmd\_history : 프로그램 시작부터 성공적으로 수행한 명령어 목록을 보여준다.

-add\_history() : history를 linked list에 저장

cmd\_dir() : 현재 위치의 파일과 디렉토리상황을 보여준

cmd\_quit() : 모든 할당된 메모리를 해제하고 프로그램을 종료한다.

-clear\_history() : linked list 를 해제

-clear\_hash() : hash table 해제

cmd\_type(): 해당 파일의 모든 텍스트를 화면에 출력

1. memory command //memory.c

cmd\_dump() : 메모리 상황을 parameter에 따라 출력한다.

cmd\_fill() : 메모리를 시작주소부터 끝주소까지 목표 값으로 채운다.

cmd\_edit() : 타겟주소의 메모리를 목표값으로 바꾼다.

1. opcode command //opcode.c

cmd\_opcodelist() : opcode의 hashtable을 출력한다.

cmd\_opcode\_mnemonic() : 해당 mnemonic의 opcode를 출력한다.

1. assemble command //assemble.c

cmd\_assemble() : asm file을 obj, lst 파일로 만들어준다.

cmd\_symbol(): symbol table을 출력한다.

### 사용변수

//cmd\_dir()

char currentPath[\_MAX\_PATH] : 현재 위치 디렉토리의 정보를 저장

DIR \*dir\_ptr : 현재 디렉토리의 주소를 저장하는 구조체

struct dirent \*file : 현재 디렉토리의 정보를 저장하는 구조체

struct stat buf : 실행파일인지 디렉토리인지 알아보기 위한 구조체

## 모듈이름: error\_print() //argument.c

### 기능

받은 error code에 따라 맞는 error message를 출력해준다.

.

### 사용변수

없음

3.8 모듈이름 : assembler() //assemble.c

*3.8.1 기능*

asm파일을 assemble 하여 obj파일과 lst 파일을 생성한다.

먼저 symbol table을 초기화 시키고 .asm 확장자가 맞는지 확인하고 그 asm 파일이 빈파일인지 확인한다. asm파일을 성공적으로 열면, pass1, pass2를 호출하여 assemble을 시행한다.

pass1 : 주소가 필요한 instruction에 주소를 배정해주고, 각 label 마다 symbol table에 add한다.

일단 1 line이 들어오면 이를 make\_instruction() 을 이용해 좀더 정규화되도록 한다.

이때 정규화란 이 line에 label이 있는지 comment가 있는지operand가 있는지 mnemonic이 있는지 directive가 있는지 일일히 검사하여 이 string을 읽기 쉬운 구조체로 만들어준다.

이는 pass1에서 각 line을 읽을 때 마다 실행해주며 프로그래머가 좀 더 instruction 파악이 쉽게 도움을 준다. 이렇게 구조체로 정규화 시키고 우선 comment가 아닌지 검사한다.

comment가 아니라면 label이 있는지 검사하고 label 이 있다면 이를 symbol table에 넣어준다. 이때 두개 이상의 레이블이 중복된다면 에러 처리한다.

또한 directive와 mnemonic에서 주소를 갖는 명령어를 판별하여 길이에 맞는 각 주소를 배정해준다.

이러한 모든 과정은 make instruction에서 만들었던 구조체 intermediate에 linked list로 저장되며 pass1이 끝나고 나면 pass2에서는 이러한 linked list를 이용한다.

pass2 : pass1에서 받은 중간데이터를 받아 이를 리스트파일과 오브젝트 파일로 변환한다.

일단 START directive를 판별하여 start address를 알아내고 첫줄의 list파일과 obj 파일을 작성한다. 이 이후로 매번 하나의 instruction을 읽어나가며 END directive가 나온다면 종료한다.

각 읽은 Instruction의 type은 2가지로 나뉘어져 있다. obj code를 생성하지 않는 directive들, object code를 생성하는 mnemonic directive들이 있다. obj code를 생성하지 않는 directive는 BASE UNBASE, RESW, RESB 정도가 있고 나머지는 obj code를 생성한다. 따라서 obj code를 생성하지 않는 directive에 대한 예외 처리를 해준다. 그리고 나머지 instruction은 make\_obj() 함수를 호출하는데 이는 object code를 생성해주는 함수이다. 각 format에 따라 분류하고 이에 맞게 object code를 생성한다. 이러한 obj code는 code만 전달해주는 것이 아닌 format도 같이 전달해준다 왜냐하면 obj file에서 format에 따라 전체길이가 달라지기 때문이다. 이러한 obj를 만들고 나면 이를 obj record에 점점 추가시키고 record가 너무 길어지면 기준에 따라 개행하고 record 를 obj file에 추가 시킨다. list file은 instruction 한 개당 한줄씩 추가시켜야한다.

end를 만나면 마지막 end list와 modification obj, end obj를 작성하며 종료된다.

이런 pass1 -> pass2의 과정을 무사히 마치게 되면 file stream을 닫고 intermediate data들을 해제 시키며 해당 command를 종료시킨다.

write\_

obj()

cmd\_assemble()

intermediate

data

PASS2

(make obj code,

write list file,

write obj file)

PASS1

(location alocate,

make symbol table)

asm file

pointer

write\_

list()

make\_

obj()

add\_

symbol()

make\_

instruction()

3.9 모듈이름 : Linkloader() //linkloader.c

*3.9.1 기능*

*여러 개의 object 파일을 하나로 묶고 이를 메모리에 올린다. 이때 progaddr이 지정되어 있으면 이를 기준으로 object 파일을 순차적으로 상대배치한다.*

*Linkloader는 2 pass로 이루어져 있다. Pass1에서는 D record type으로 지정되어 있는 extern symbol을 hash table에 저장한다. 만약 여러 개의 define symbol이 나온다면 duplicate error을 출력한다. Pass1을 거치고 나면 각 control section과 관련된 extern symbol이 나타나 있는 load map과 extern symbol table이 만들어 진다. 이러한 정보들을 가지고 pass2에서는 본격적으로 load시킨다. 먼저 reference record 에 저장되어 있는 symbol name과 index를 읽어 reference array를 생성하고 modification에 용의하게 한다. 이를 읽은 후에 Text record를 progaddr 부터 차례대로 load한다. Text record를 다 로드 시킨후 modification record를 읽는데 address, half byte length, sign, reference index를 차례대로 읽어 해당 메모리의 데이터에 sign에 따라 더하거나 뺀다.이를 마치고 나면 command 함수에 성공을 반환하며 동적 할당된 load map이나 extern symbol table의 동적메모리를 해제시킨다.*

3.10 모듈이름 : Simulator() //simulator.c

*3.10.1 기능*

*Simulator는 실제 load된 메모리에 따라 프로그램을 실행시키는 모듈이다. Run 명령어를 입력하면 각 메모리에 로드되어있는 object code 를 해석하여 각 포맷에 맞는 instruction을 수행시켜준다. Break point가 설정되어 있다면 break point에 종료하며 break point가 설정되어 있지 않다면 프로그램이 전체가 종료되면 모듈이 종료된다.*

# 전역 변수 정의

## argument 관련 전역 변수

1. INPUT\_SIZE 100 : 사용자가 입력할 수 있는 문자열 최대 길이
2. MNEMONIC\_SIZE 30 : opcode의 mnemonic 문자열 최대 길이
3. FORMAT\_SIZE 5 : opcode의 format 문자열 최대 길이
4. \_MAX\_PATH 260 : 디렉토리 PATH 문자열 최대 길이
5. FILE\_BUFFER\_SIZE 1000 : FILE 에서 읽어들일 수 있는 문자열 최대 길이

## opcode hash 관련 전역 변수

1)

typedef struct opcode\* onptr;

typedef struct opcode

{

char mnemonic[MNEMONIC\_SIZE];

int opcode;

char format[FORMAT\_SIZE];

onptr link;

}opcode\_node;

hnptr hash[HASH\_SIZE] = {NULL};

: hash table과 mnemonic에 따른 opcode, format을 저장하기 위한 node

2) HASH\_SIZE : hash table의 크기

## history 관련 전역 변수

1)

typedef struct history\_node\* hnptr;

typedef struct history\_node //linked list of history

{

char arg[INPUT\_SIZE];

hnptr link;

}history\_node;

hnptr hn\_head = NULL;

: history를 저장하기 위한 linked list node structure

2)int history\_cnt : history list에 저장되어 있는 histroy의 개수. 맨 처음 history 명령어가 입력되었을 때 아무것도 출력하지 않기 위한 변수

## memory 관련 전역 변수

1. MEM\_SIZE 65536\*16 : memory의byte 수
2. int last\_addr : dump 명령어에서 출력한 메모리의 마지막 주소를 저장하는 변수

## error 관련 전역 변수

1. ERROR\_HEX\_RANGE -1000 : 16진수의 범위를 벗어나는 에러
2. ERROR\_HEX\_FORMAT -1001 : 16진수의 형식이 아닐 때 발생하는 에러
3. ERROR\_MEM\_BOUND -1002 : 메모리 영역을 벗어날 때 발생하는 에러
4. ERROR\_OVER\_ARG -1003 : 요구하는 명령어보다 더 많은 명령어를 쳤을 때 발생하는 에러
5. ERROR\_LACK\_ARG -1004 : 요구하는 명령어보다 적은 명령어를 쳤을 때 발생하는 에러
6. ERROR\_SE\_RANGE -1005 : start address가 end address보다 더 클 때 발생하는 에러
7. ERROR\_VALUE\_RANGE -1006 : 1바이트에 저장될 value의 범위가 1바이트를 넘어갈 때 발생하는 에러
8. ERROR\_NOT\_FOUND -1007 : 해당하는 명령어를 찾을 수 없을 때 발생하는 에러
9. ERROR\_FILE\_OPEN -1008 : FILE 을 열 때 발생하는 에러
10. ERROR\_FILE\_CLOSE -1009 : FILE을 닫을 때 발생하는 에러
11. ERROR\_NOT\_MNEMONIC -1010 : 사용자 입력 mnemonic이 hash table에 없을 때 발생하는 에러
12. ERROR\_DIR\_OPEN -1011 : directory 열때 발생하는 에러
13. ERROR\_DIR\_CLOSE -1012 : directory 닫을 때 발생하는 에러
14. ERROR\_EMPTY\_ASM -1013 : asm file이 비었을 때 발생하는 에러
15. ERROR\_WRONG\_FORMAT -1014 : 올바른 format을 어겼을 때 발생하는 에러
16. ERROR\_DUP\_SYM -1015 : symbol이 중복되었을 때 발생하는 에러
17. ERROR\_NOT\_END -1016 : end directive가 나오지 않았을 때 발생하는 에러
18. ERROR\_DEC\_RANGE -1017 : 10진수 범위 에러
19. ERROR\_DEC\_FORMAT -1018 : 10진수 형식 에러
20. ERROR\_NOT\_ASM -1019 : asm 파일이 아닐 때 발생하는 에러
21. ERROR\_NOT\_SYM -1020 : symbol이 없을 때 발생하는 에러
22. ERROR\_ADDR\_MODE -1021 : 맞는 addressing mode가 없을 때 발생하는 에러

## flag 관련 전역변수

1. FLAG\_EMPTY -2000 : dump 명령어에서 start나 end 가 비어있을 수 도 있는데 이때 비어있음을 이 FLAG\_EMPTY로 표현한다.

**4.7 assemble 관련 전역변수**

1) INSTRUCTION\_SIZE 100 : 하나의 instruction의 버퍼 사이즈

2) SYMBOL\_SIZE 30 : 각 symbol마다 가질수있는 length

3) OBJCODE\_SIZE 30 : 각 object code마다 가질수 있는 length

4) RECODE\_SZIE 100 : 레코드가 가질수 있는 최대 길이

typedef struct symbol\_node\* snptr;

typedef struct symbol\_node

{

char symbol[SYMBOL\_SIZE];

int loc;

}symbol\_node;

심볼을 저장하기 위한 구조체

typedef struct instruction\* inptr;

typedef struct instruction

{

char \*comment;

char \*label;

char \*directive;

char \*mnemonic;

char \*operand;

int loc;

int line;

inptr link;

}instruction;

중간파일을 저장하기 위한 구조체

typedef struct object\_code

{

int code;

int format;

}object\_code;

오브젝트 코드를 저장하기위한 구조체

hnptr sym\_table[] : 심볼 테이블

inptr intermediate : 중간파일 링크드 리스트 헤더 파일

inptr rear : 중간파일 링크드 리스트의 가장 끝 포인터

int base\_flag : base relative addressing 사용가능 여부

int base : base register 값

char program\_name[] : program의 name

int program\_length : program의 전체 길이

int sym\_table\_num : symbol table에 들어있는 element의 숫자

int modificaition\_table[] : 재배치 테이블

int mdf\_num : 재배치 테이블에 들어있는 원소 개수

**4.8 hash 관련 전역변수**

typedef struct hash\_node\* hnptr;

typedef struct hash\_node

{

void\* data;

hnptr link;

}hash\_node;

hash table을 구성하기 위한 구조체

**4.9 Link loader 관련 전역변수**

typedef struct esymbol\_node\* esnptr;

typedef struct esymbol\_node

{

char symbol[SYMBOL\_SIZE];

int loc;

esnptr link;

}esymbol\_node;

Extern symbol table의 data node

typedef struct LOAD\_MAP

{

esnptr head[\_MAX\_LINK];

esnptr rear[\_MAX\_LINK];

int prog\_length[\_MAX\_LINK];

int link\_cnt;

}LOAD\_MAP;

Load map을 저장하기 위한 구조체

extern int prog\_addr;

//program의 첫 시작 주소를 지정해주는 전역 변수

**4.10 Simulator 관련 전역변수**

extern unsigned int reg[16];

레지스터를 저장하는변수

enum { A = 0, X = 1, L = 2, PC = 8, SW = 9, B = 3, S = 4, T = 5, F = 6 };

레지스터의 번호를 지정

enum { LT, GT, EQ };

조건 상태를 지정

typedef enum addressing { EMPTY, IMMEDIATE, INDIRECT, INDEXED, RELATIVE\_B, RELATIVE\_PC, DIRECT, EXTEND }addressing;

typedef struct ADDR\_MODE

{

addressing m[4];

}ADDR\_MODE;

주소 지정 모드를 저장하는 구조체

# 코드

20151607.c

main.h

argument.h

assemble.h

shell.h

opcode.h

exception.h

memory.h

hash.h

assemble.c

shell.c

opcode.c

memory.c

hash.c

exception.c

argument.c

Simulator.h

Debug.h

Linkloader.h

Simulator.c

Debug.c

Linkloader.c

**5.1 main.h**

#ifndef \_\_MAIN\_H\_\_

#define \_\_MAIN\_H\_\_

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <dirent.h>

#include <unistd.h>

#include <sys/stat.h>

#include <sys/types.h>

#define ERROR\_HEX\_RANGE -1000

#define ERROR\_HEX\_FORMAT -1001

#define ERROR\_MEM\_BOUND -1002

#define ERROR\_OVER\_ARG -1003

#define ERROR\_LACK\_ARG -1004

#define ERROR\_SE\_RANGE -1005

#define ERROR\_VALUE\_RANGE -1006

#define ERROR\_NOT\_FOUND -1007

#define ERROR\_FILE\_OPEN -1008

#define ERROR\_FILE\_CLOSE -1009

#define ERROR\_NOT\_MNEMONIC -1010

#define ERROR\_DIR\_OPEN -1011

#define ERROR\_DIR\_CLOSE -1012

#define ERROR\_EMPTY\_ASM -1013

#define ERROR\_WRONG\_FORMAT -1014

#define ERROR\_DUP\_SYM -1015

#define ERROR\_NOT\_END -1016

#define ERROR\_DEC\_RANGE -1017

#define ERROR\_DEC\_FORMAT -1018

#define ERROR\_NOT\_ASM -1019

#define ERROR\_NOT\_SYM -1020

#define ERROR\_ADDR\_MODE -1021

#define ERROR\_NOT\_OBJ -1022

#define ERROR\_FILE\_SEEK -1023

#define ERROR\_LINK\_CNT -1024

#define ERROR\_MDF\_OVERFLOW -1025

#define ERROR\_NOT\_INSTRUCT -1026

#define FLAG\_EMPTY -2000

#define FILE\_BUFFER\_SIZE 1000

#define INSTRUCTION\_BUFFER 100

#define INPUT\_SIZE 100

#define \_MAX\_PATH 260

#define HASH\_SIZE 20

#define FORMAT\_SIZE 5

#define MEM\_SIZE 65536\*16

#define MNEMONIC\_SIZE 30

#define SYMBOL\_SIZE 30

#define OBJCODE\_SIZE 30

#define RECODE\_SIZE 100

#define \_MAX\_LINK 3

#define \_MAX\_REFER 256

#define BP\_SIZE 1000

#endif

**5.2 20151607.c //main module 역할**

#include "main.h"

#include "argument.h"

int main()

{

char arg[INPUT\_SIZE];

char tmp\_arg[INPUT\_SIZE];

read\_opcode\_file();

while(1)

{

printf("sicsim> ");

fgets(arg,INPUT\_SIZE, stdin);

strcpy(tmp\_arg, arg);

refine\_argument(tmp\_arg, arg);

if (is\_help(arg, tmp\_arg))

;

else if (is\_dir(arg, tmp\_arg))

;

else if (is\_quit(arg))

;

else if (is\_history(arg, tmp\_arg))

;

else if (is\_dump(arg, tmp\_arg))

;

else if (is\_edit(arg, tmp\_arg))

;

else if (is\_fill(arg, tmp\_arg))

;

else if (is\_reset(arg, tmp\_arg))

;

else if (is\_opcodelist(arg, tmp\_arg))

;

else if (is\_opcode\_mnemonic(arg, tmp\_arg))

;

else if (is\_type(arg, tmp\_arg))

;

else if (is\_assemble(arg, tmp\_arg))

;

else if (is\_symbol(arg, tmp\_arg))

;

else if (is\_loader(arg, tmp\_arg))

;

else if (is\_progaddr(arg, tmp\_arg))

;

else if (is\_bp(arg, tmp\_arg))

;

else if (is\_run(arg, tmp\_arg))

;

else

error\_print(ERROR\_NOT\_FOUND);

}

return 0;

}

**5.3 argument.h , argument.c //extract\_parmeter, print\_error, is\_command module 실행**

#ifndef \_\_ARGUMENT\_H\_\_

#define \_\_ARGUMENT\_H\_\_

#include "main.h"

#include "exception.h"

#include "shell.h"

#include "memory.h"

#include "opcode.h"

#include "assemble.h"

#include "linkloader.h"

#include "debug.h"

#include "simulator.h"

void refine\_argument(char arg[], char result[]);

int parsing\_argument(char \*\*arg, char \*\*parse);

int hex\_to\_int(char hex[]);

int dec\_to\_int(char dec[]);

int int\_to\_hex(int num, char hex[], int length);

int is\_help(char arg[], char tmp\_arg[]);

int is\_history(char arg[], char tmp\_arg[]);

int is\_dir(char arg[], char tmp\_arg[]);

int is\_type(char arg[], char tmp\_arg[]);

int is\_quit(char arg[]);

int is\_dump(char arg[], char tmp\_arg[]);

int is\_edit(char arg[], char tmp\_arg[]);

int is\_fill(char arg[], char tmp\_arg[]);

int is\_reset(char arg[], char tmp\_arg[]);

int is\_opcodelist(char arg[], char tmp\_arg[]);

int is\_opcode\_mnemonic(char arg[], char tmp\_arg[]);

int is\_assemble(char arg[], char tmp\_arg[]);

int is\_symbol(char arg[], char tmp\_arg[]);

int is\_loader(char arg[], char tmp\_arg[]);

int is\_progaddr(char arg[], char tmp\_arg[]);

int is\_bp(char arg[], char tmp\_arg[]);

int is\_run(char arg[], char tmp\_arg[]);

#endif

#include "argument.h"

int symbol\_flag = 0;

//if argument have help command and set right format, execute help command and add history

//else print error message

//return 1 if argument have help command (even if wrong format) else return 0

int is\_help(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "help") == 0 || strcmp(parse, "h") == 0)

{

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

cmd\_help();

add\_history(tmp\_arg);

return 1;

}

return 0;

}

//if argument have dir command and set right format, execute dir command and add history

//else print error message

//return 1 if argument have dir command (even if wrong format) else return 0

int is\_dir(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "dir") == 0 || strcmp(parse, "d") == 0)

{

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

int error = cmd\_dir();

if (error == ERROR\_DIR\_OPEN || error == ERROR\_DIR\_CLOSE)

{

error\_print(error);

return 1;

}

add\_history(tmp\_arg);

return 1;

}

return 0;

}

int is\_type(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "type") == 0)

{

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

error\_print(ERROR\_LACK\_ARG);

return 1;

}

char \*filename = parse;

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

int error = cmd\_type(filename);

if (error == ERROR\_FILE\_OPEN || error == ERROR\_FILE\_CLOSE)

{

error\_print(error);

return 1;

}

add\_history(tmp\_arg);

return 1;

}

return 0;

}

//if argument have quit command and set right format, execute quit command

//else print error message

//return 1 if argument have quit command (even if wrong format) else return 0

int is\_quit(char arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "quit") == 0 || strcmp(parse, "q") == 0)

{

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

cmd\_quit();

return 1;

}

return 0;

}

//if argument have history command and set right format, execute history command and add history

//else print error message

//return 1 if argument have history command (even if wrong format) else return 0

int is\_history(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "history") == 0 || strcmp(parse, "hi") == 0)

{

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

add\_history(tmp\_arg);

cmd\_history();

return 1;

}

return 0;

}

//if argument have dump command and set right format, execute dump command and add history

//else print error message

//return 1 if argument have dump command (even if wrong format) else return 0

int is\_dump(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "dump") == 0 || strcmp(parse, "du") == 0)

{

int start = FLAG\_EMPTY, end = FLAG\_EMPTY;

len = parsing\_argument(&arg, &parse);

if (len)

{

start = hex\_to\_int(parse);

if (start == ERROR\_HEX\_FORMAT)

{

error\_print(ERROR\_HEX\_FORMAT);

return 1;

}

if (start == ERROR\_HEX\_RANGE)

{

error\_print(ERROR\_HEX\_RANGE);

return 1;

}

len = parsing\_argument(&arg, &parse);

if (len)

{

if (strcmp(parse, ",") != 0)

{

error\_print(ERROR\_LACK\_ARG);

return 1;

}

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

error\_print(ERROR\_LACK\_ARG);

return 1;

}

end = hex\_to\_int(parse);

if (end == ERROR\_HEX\_FORMAT)

{

error\_print(ERROR\_HEX\_FORMAT);

return 1;

}

if (end == ERROR\_HEX\_RANGE)

{

error\_print(ERROR\_HEX\_RANGE);

return 1;

}

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

}

}

int error = cmd\_dump(start, end);

if (error == ERROR\_SE\_RANGE)

{

error\_print(ERROR\_SE\_RANGE);

return 1;

}

else if (error == ERROR\_MEM\_BOUND)

{

error\_print(ERROR\_MEM\_BOUND);

return 1;

}

add\_history(tmp\_arg);

return 1;

}

return 0;

}

//if argument have edit command and set right format, execute edit command and add history

//else print error message

//return 1 if argument have edit command (even if wrong format) else return 0

int is\_edit(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "edit") == 0 || strcmp(parse, "e") == 0)

{

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

error\_print(ERROR\_LACK\_ARG);

return 1;

}

int addr = hex\_to\_int(parse);

if (addr == ERROR\_HEX\_FORMAT)

{

error\_print(ERROR\_HEX\_FORMAT);

return 1;

}

if (addr == ERROR\_HEX\_RANGE)

{

error\_print(ERROR\_HEX\_RANGE);

return 1;

}

len = parsing\_argument(&arg, &parse);

if (len == 0 || strcmp(parse, ",") != 0)

{

error\_print(ERROR\_LACK\_ARG);

return 1;

}

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

error\_print(ERROR\_LACK\_ARG);

return 1;

}

int value = hex\_to\_int(parse);

if (value == ERROR\_HEX\_FORMAT)

{

error\_print(ERROR\_HEX\_FORMAT);

return 1;

}

if (value == ERROR\_HEX\_RANGE)

{

error\_print(ERROR\_HEX\_RANGE);

return 1;

}

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

int error = cmd\_edit(addr, value);

if (error == ERROR\_VALUE\_RANGE)

{

error\_print(ERROR\_VALUE\_RANGE);

return 1;

}

add\_history(tmp\_arg);

return 1;

}

return 0;

}

//if argument have fill command and set right format, execute fill command and add history

//else print error message

//return 1 if argument have fill command (even if wrong format) else return 0

int is\_fill(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "fill") == 0 || strcmp(parse, "f") == 0)

{

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

error\_print(ERROR\_LACK\_ARG);

return 1;

}

int start = hex\_to\_int(parse);

if (start == ERROR\_HEX\_FORMAT || start == ERROR\_HEX\_RANGE)

{

error\_print(start);

return 1;

}

len = parsing\_argument(&arg, &parse);

if (len == 0 || strcmp(parse, ",") != 0)

{

error\_print(ERROR\_LACK\_ARG);

return 1;

}

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

error\_print(ERROR\_LACK\_ARG);

return 1;

}

int end = hex\_to\_int(parse);

if (end == ERROR\_HEX\_FORMAT || end == ERROR\_HEX\_RANGE)

{

error\_print(end);

return 1;

}

len = parsing\_argument(&arg, &parse);

if (len == 0 || strcmp(parse, ",") != 0)

{

error\_print(ERROR\_LACK\_ARG);

return 1;

}

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

error\_print(ERROR\_LACK\_ARG);

return 1;

}

int value = hex\_to\_int(parse);

if (value == ERROR\_HEX\_FORMAT || value == ERROR\_HEX\_RANGE)

{

error\_print(value);

return 1;

}

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

int error = cmd\_fill(start, end, value);

if (error == ERROR\_VALUE\_RANGE || error == ERROR\_SE\_RANGE)

{

error\_print(error);

return 1;

}

add\_history(tmp\_arg);

return 1;

}

return 0;

}

//if argument have reset command and set right format, execute reset command and add history

//else print error message

//return 1 if argument have reset command (even if wrong format) else return 0

int is\_reset(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "reset") == 0)

{

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

cmd\_reset();

add\_history(tmp\_arg);

return 1;

}

return 0;

}

//if argument have opcodelist command and set right format, execute opcodelist command and add history

//else print error message

//return 1 if argument have opcodelist command (even if wrong format) else return 0

int is\_opcodelist(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "opcodelist") == 0)

{

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

cmd\_opcodelist();

add\_history(tmp\_arg);

return 1;

}

return 0;

}

//if argument have opcode mnemonic command and set right format, execute opcode mnemonic command and add history

//else print error message

//return 1 if argument have opcode mnemonic command (even if wrong format) else return 0

int is\_opcode\_mnemonic(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "opcode") == 0)

{

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

error\_print(ERROR\_LACK\_ARG);

return 1;

}

char\* mnemonic = parse;

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

int error = cmd\_opcode\_mnemonic(mnemonic);

if (error == ERROR\_NOT\_MNEMONIC)

{

error\_print(error);

return 1;

}

add\_history(tmp\_arg);

return 1;

}

return 0;

}

//check if argument is assemble command

//if this is assemble command return 1 even if error exist else return 0

int is\_assemble(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "assemble") == 0)

{

symbol\_flag = 0;

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

error\_print(ERROR\_LACK\_ARG);

return 1;

}

char \*filename = parse;

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

if (cmd\_assemble(filename))

{

symbol\_flag = 1;

add\_history(tmp\_arg);

}

return 1;

}

return 0;

}

//check if argument is symbol command

//if this is symbol command return 1 even if error exist else return 0

int is\_symbol(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "symbol") == 0)

{

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

if(symbol\_flag)

cmd\_symbol();

else

printf("there is not recent success asm file\n");

add\_history(tmp\_arg);

return 1;

}

return 0;

}

int is\_loader(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "loader") == 0)

{

int cnt = 0;

char filename[\_MAX\_LINK][INPUT\_SIZE];

while (len = parsing\_argument(&arg, &parse))

{

if (cnt >= \_MAX\_LINK)

{

error\_print(ERROR\_LINK\_CNT);

return 1;

}

strcpy(filename[cnt++], parse);

}

if (cmd\_loader(filename,cnt))

{

add\_history(tmp\_arg);

}

return 1;

}

return 0;

}

int is\_progaddr(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "progaddr") == 0)

{

len = parsing\_argument(&arg, &parse);

int value = hex\_to\_int(parse);

if (value == ERROR\_HEX\_FORMAT || value == ERROR\_HEX\_RANGE)

{

error\_print(value);

return 1;

}

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

cmd\_progaddr(value);

add\_history(tmp\_arg);

return 1;

}

return 0;

}

int is\_bp(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "bp") == 0)

{

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

//show all break point

cmd\_bp\_show();

}

else if (strcmp(parse, "clear") == 0)

{

//clear all break point

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

cmd\_bp\_clear();

}

else

{

//set break point

int value = hex\_to\_int(parse);

if (value == ERROR\_HEX\_FORMAT || value == ERROR\_HEX\_RANGE)

{

error\_print(value);

return 1;

}

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

cmd\_bp\_set(value);

}

add\_history(tmp\_arg);

return 1;

}

return 0;

}

int is\_run(char arg[], char tmp\_arg[])

{

char \*parse;

int len = 0;

len = parsing\_argument(&arg, &parse);

if (strcmp(parse, "run") == 0)

{

len = parsing\_argument(&arg, &parse);

if (len)

{

error\_print(ERROR\_OVER\_ARG);

return 1;

}

cmd\_run();

add\_history(tmp\_arg);

return 1;

}

return 0;

}

//refine argument to proper format (ex refined = "arg1'\0'arg2'\0'arg3\0\0")

//remove tab and space bar and divde comma from argument

void refine\_argument(char arg[], char result[])

{

int i = 0;

while (arg[i] == ' ' || arg[i] == '\t')

i++;

char pre = 'a';

int j = 0;

for (; arg[i] != '\0'; i++)

{

if (arg[i] != ' ' && arg[i] != '\t')

{

if (arg[i] == ',' || pre == ',' || pre == ' ' || pre == '\t')

result[j++] = '\0';

result[j++] = arg[i];

}

pre = arg[i];

}

if (result[j - 1] == '\n')

result[j - 1] = '\0';

else

result[j++] = '\0';

result[j] = '\0';

}

//parsing construct from refined argument and set arg pointer to point next construct

//return length of construct and pointer of construct

int parsing\_argument(char \*\*arg, char \*\*parse)

{

int len = strlen(\*arg);

\*parse = \*arg;

\*arg += len + 1;

return len;

}

//change hex code to integer

//if hex code is not valid for given hex format return error number

//else return changed inter

int hex\_to\_int(char hex[])

{

while (hex[0] == '0')

hex++;

int len = strlen(hex);

if (len > 5)

return ERROR\_HEX\_RANGE; //overflow error

int ret = 0;

for (int i = 0; i < len; i++)

{

ret \*= 16;

if ('0' <= hex[i] && hex[i] <= '9')

ret += hex[i] - '0';

else if ('A' <= hex[i] && hex[i] <= 'F')

ret += hex[i] - 'A' + 10;

else if ('a' <= hex[i] && hex[i] <= 'f')

ret += hex[i] - 'a' + 10;

else

return ERROR\_HEX\_FORMAT; //It is not hex code

}

return ret;

}

//transform decimal string to integer

//if error exsit return error code else return 1

int dec\_to\_int(char dec[])

{

while (dec[0] == '0')

dec++;

int len = strlen(dec);

if (len > 9)

return ERROR\_DEC\_RANGE; //overflow error

int ret = 0;

for (int i = 0; i < len; i++)

{

ret \*= 10;

if ('0' <= dec[i] && dec[i] <= '9')

ret += dec[i] - '0';

else

return ERROR\_DEC\_FORMAT; //It is not dec code

}

return ret;

}

//transform integer to hexa code

//if error exist return error code else return 1

int int\_to\_hex(int num, char hex[], int length)

{

int i = 0;

while (num)

{

if (length - i - 1 < 0)

return ERROR\_VALUE\_RANGE;

int tmp = num % 16;

if (tmp < 10)

hex[length - i - 1] = tmp + '0';

else

hex[length - i - 1] = tmp - 10 + 'A';

i++;

num /= 16;

}

while (length - i - 1 >= 0)

{

hex[length - i - 1] = '0';

i++;

}

hex[length] = '\0';

return 1;

}

**5.4 shell.h shell.c // (shell) command module 실행**

#ifndef \_\_SHELL\_H\_\_

#define \_\_SHELL\_H\_\_

#include "main.h"

#include "hash.h"

#include "opcode.h"

#include "assemble.h"

typedef struct history\_node\* hinptr;

void add\_history(char arg[]);

void clear\_history();

void cmd\_help();

void cmd\_history();

int cmd\_dir();

int cmd\_type(char filename[]);

void cmd\_quit();

#endif#include "shell.h"

typedef struct history\_node

{

char arg[INPUT\_SIZE];

hinptr link;

}history\_node;

hinptr hn\_head = NULL;

int history\_cnt = 0;

//add argument to the linked-list

void add\_history(char arg[])

{

history\_cnt++;

hinptr pnew = (hinptr)malloc(sizeof(\*pnew));

for (int i = 0; arg[i] != '\0';)

{

strcpy(pnew->arg, arg);

i += strlen(arg + i) + 1;

}

pnew->link = NULL;

strcpy(pnew->arg, arg);

if (hn\_head == NULL)

{

hn\_head = pnew;

return;

}

hinptr p = hn\_head;

while (p->link)

p = p->link;

p->link = pnew;

}

//free all of dynamic memories in history linked list

void clear\_history()

{

hinptr p = hn\_head;

while (p)

{

hinptr next = p->link;

free(p);

p = next;

}

}

//execute history command

//show all history in its linked-list

void cmd\_history()

{

if(history\_cnt == 1)

return;

int i = 1;

for (hinptr p = hn\_head; p; p = p->link, i++)

{

printf("%d %s", i, p->arg);

}

}

//execute help command

//show all command name with each right format

void cmd\_help()

{

printf("h[elp]\n");

printf("d[ir]\n");

printf("q[uit]\n");

printf("hi[story]\n");

printf("du[mp] [start, end]\n");

printf("e[dit] address, value\n");

printf("f[ill] start, end, value\n");

printf("reset\n");

printf("opcode mnemonic\n");

printf("opcodelist\n");

printf("assemble filename\n");

printf("type filename\n");

printf("symbol\n");

printf("progaddr [address]\n");

printf("loader [object filename1] [...]\n");

printf("run\n");

printf("bp [address] (or) bp clear (or) bp\n");

return;

}

//execute dir command

//show all directories and executable files in current directory

//if fail to open current directory or close return error message else return 1

int cmd\_dir()

{

char currentPath[\_MAX\_PATH];

getcwd(currentPath, \_MAX\_PATH);

DIR \*dir\_ptr;

struct dirent \*file;

struct stat buf;

if ((dir\_ptr = opendir(currentPath)) == NULL)

return ERROR\_DIR\_OPEN;

while ((file = readdir(dir\_ptr)) != NULL)

{

if (file->d\_name[0] != '.')

{

lstat(file->d\_name, &buf);

if (S\_ISDIR(buf.st\_mode))

printf("%s/ ", file->d\_name);

else if (S\_ISREG(buf.st\_mode))

printf("%s\* ", file->d\_name);

}

}

printf("\n");

if(closedir(dir\_ptr) == 1)

return ERROR\_DIR\_CLOSE;

return 1;

}

//execute type command

//print all text in file

//if error exist return 0 else return 1

int cmd\_type(char filename[])

{

FILE\* fp = fopen(filename, "r");

if (fp == NULL)

{

error\_print(ERROR\_FILE\_OPEN);

return 0;

}

char buffer[FILE\_BUFFER\_SIZE];

while (fgets(buffer, FILE\_BUFFER\_SIZE, fp) != NULL)

{

printf("%s", buffer);

}

if (fclose(fp) == EOF)

{

error\_print(ERROR\_FILE\_CLOSE);

return 0;

}

return 1;

}

//execute quit command

//delete all of dynamic memmories which are assigend by the program (history linked-list, hash table)

//and return 0 and terminate process

void cmd\_quit()

{

hnptr\* op\_table = get\_op\_table\_addr();

hnptr\* sym\_table = get\_sym\_table\_addr();

clear\_history();

clear\_hash(op\_table);

clear\_hash(sym\_table);

exit(0);

}

**5.5 memory.h memory.c // (memory) command module 실행**

#ifndef \_\_MEMORY\_H\_\_

#define \_\_MEMORY\_H\_\_

#include "main.h"

extern unsigned char memory[MEM\_SIZE];

int cmd\_dump(int start, int end);

int cmd\_edit(int addr, int value);

int cmd\_fill(int start, int end, int value);

void cmd\_reset();

#endif

#include "memory.h"

unsigned char memory[MEM\_SIZE] = { 0, };

int last\_addr = -1;

//execute dump command

//show memory status from start address to end address

//print the first address in line and each value with ASCII code

//if start is greater than end, return error number else return 1

int cmd\_dump(int start, int end)

{

if (start == FLAG\_EMPTY)

start = last\_addr + 1;

if (start == MEM\_SIZE)

start = 0;

if (end == FLAG\_EMPTY)

end = start + 159;

if (start > end)

return ERROR\_SE\_RANGE;

if (end >= MEM\_SIZE)

end = MEM\_SIZE - 1;

for (int i = start / 16; i <= end / 16; i++)

{

printf("%05X ", i \* 0x10);

for (int j = 0; j<16; j++)

{

int cur = i \* 0x10 + j;

if (start <= cur && cur <= end)

printf("%02X ", memory[cur]);

else

printf(" ");

}

printf("; ");

for (int j = 0; j<16; j++)

{

int cur = i \* 0x10 + j;

if (start <= cur && cur <= end && 0x20 <= memory[cur] && memory[cur] <= 0x7E)

printf("%c", memory[cur]);

else

printf(".");

}

printf("\n");

}

last\_addr = end;

return 1;

}

//execute edit command

//edit original value in the address to the given value

//if value is out of range return error number else return 1

int cmd\_edit(int addr, int value)

{

if (value > 0xFF)

return ERROR\_VALUE\_RANGE;

memory[addr] = value;

return 1;

}

//execute fill command

//fill memory with given value from the start address to the end address

//if start is greater than end or value is out of range return error number else return 1

int cmd\_fill(int start, int end, int value)

{

if (value > 0xFF)

return ERROR\_VALUE\_RANGE;

if (start > end)

return ERROR\_SE\_RANGE;

for (int i = start; i <= end; i++)

memory[i] = value;

return 1;

}

//execute reset command

//reset all memory by zero

void cmd\_reset()

{

for (int i = 0; i<MEM\_SIZE; i++)

memory[i] = 0;

}

**5.6 opcode.h opcode.c // (opcode) command module , read\_opcode module 실행**

#ifndef \_\_OPCODE\_H\_\_

#define \_\_OPCODE\_H\_\_

#include "main.h"

#include "hash.h"

#include "exception.h"

#include "argument.h"

typedef struct opcode\_node\* onptr;

typedef struct opcode\_node

{

char mnemonic[MNEMONIC\_SIZE];

int opcode;

char format[FORMAT\_SIZE];

}opcode\_node;

int read\_opcode\_file();

onptr find\_mnemonic(char mnemonic[]);

void cmd\_opcodelist();

int cmd\_opcode\_mnemonic(char mnemonic[]);

hnptr\* get\_op\_table\_addr();

#endif

#include "opcode.h"

hnptr op\_table[HASH\_SIZE] = { NULL, };

//read opcode.txt and store in hash table

//if file open error or file close error return 0 and print error message

//else return 1

int read\_opcode\_file()

{

FILE\* fp = fopen("opcode.txt", "r");

if (fp == NULL)

{

error\_print(ERROR\_FILE\_OPEN);

return 0;

}

char buffer[FILE\_BUFFER\_SIZE];

char tmp\_buffer[FILE\_BUFFER\_SIZE];

while (fgets(buffer, FILE\_BUFFER\_SIZE, fp) != NULL)

{

strcpy(tmp\_buffer, buffer);

refine\_argument(tmp\_buffer, buffer);

char \*arg = buffer;

char \*parse;

onptr pnew = (onptr)malloc(sizeof(\*pnew));

parsing\_argument(&arg, &parse);

pnew->opcode = hex\_to\_int(parse);

parsing\_argument(&arg, &parse);

strcpy(pnew->mnemonic, parse);

parsing\_argument(&arg, &parse);

strcpy(pnew->format, parse);

int hash\_number = hash\_function(pnew->mnemonic);

add\_hash(op\_table, pnew, hash\_number);

}

if (fclose(fp) == EOF)

{

error\_print(ERROR\_FILE\_CLOSE);

return 0;

}

return 1;

}

//find precise mnemonic in hash table

//if success to find, return the mnemonic node else return null pointer

onptr find\_mnemonic(char mnemonic[])

{

int hash\_number = hash\_function(mnemonic);

for (hnptr p = op\_table[hash\_number]; p; p = p->link)

{

onptr op = p->data;

if (strcmp(op->mnemonic, mnemonic) == 0)

return op;

}

return NULL;

}

//execute opcodelist command

//show all opcode in each hash table number

void cmd\_opcodelist()

{

for (int i = 0; i < HASH\_SIZE; i++)

{

printf("%d : ", i);

for (hnptr p = op\_table[i]; p; p = p->link)

{

onptr data = p->data;

printf("[%s,%02X] ", data->mnemonic, data->opcode);

if (p->link)

printf("-> ");

}

printf("\n");

}

}

//execute opcode mnemonic command

//show opcode of the mnemonic

int cmd\_opcode\_mnemonic(char mnemonic[])

{

onptr op = find\_mnemonic(mnemonic);

if (op == NULL)

{

return ERROR\_NOT\_MNEMONIC;

}

printf("opcode is %02X\n", op->opcode);

return 1;

}

//get op\_table address

hnptr\* get\_op\_table\_addr()

{

return op\_table;

}

**5.7 hash.h hash.c //해쉬 테이블의 삽입,삭제, 찾기를 관리**

#ifndef \_\_HASH\_H\_\_

#define \_\_HASH\_H\_\_

#include "main.h"

typedef struct hash\_node\* hnptr;

typedef struct hash\_node

{

void\* data;

hnptr link;

}hash\_node;

void add\_hash(hnptr hash[], void\* data, int hash\_number);

void clear\_hash(hnptr hash[]);

int hash\_function(char str[]);

#endif

#include "hash.h"

void add\_hash(hnptr hash[], void\* data, int hash\_number)

{

hnptr pnew = (hnptr)malloc(sizeof(\*pnew));

pnew->data = data;

pnew->link = hash[hash\_number];

hash[hash\_number] = pnew;

}

//free all of dynamic memories in hash table

void clear\_hash(hnptr hash[])

{

for (int i = 0; i < HASH\_SIZE; i++)

{

hnptr p = hash[i];

while (p)

{

hnptr next = p->link;

free(p->data);

free(p);

p = next;

}

hash[i] = NULL;

}

}

//according to string, decide hash number

int hash\_function(char str[])

{

unsigned long hash = 5381;

int c;

while ((c = \*str++))

{

hash = ((hash << 5) + hash) + c;

}

return hash % HASH\_SIZE;

}

**5.8 exception.h exception.c //오류 코드 출력을 관리**

#ifndef \_\_EXCEPTION\_H\_\_

#define \_\_EXCEPTION\_H\_\_

#include "main.h"

#include "assemble.h"

#include "hash.h"

void error\_print(int error);

void asm\_error(int error, int line);

#endif

#include "exception.h"

//according to error parameter, print proper error message

//errors number are defined in header file

void error\_print(int error)

{

if (error == ERROR\_OVER\_ARG)

printf("ERROR\_OVER\_ARG : over argument\n");

if (error == ERROR\_HEX\_FORMAT)

printf("ERROR\_HEX\_FORMAT : please write right positive hex code\n");

if (error == ERROR\_HEX\_RANGE)

printf("ERROR\_HEX\_RANGE : hex code can be range of 0x00000 ~ 0xFFFFF\n");

if (error == ERROR\_MEM\_BOUND)

printf("ERROR\_MEM\_BOUND : memory can't over 0xFFFFF\n");

if (error == ERROR\_LACK\_ARG)

printf("ERROR\_LACK\_ARG : lack argument\n");

if (error == ERROR\_SE\_RANGE)

printf("ERROR\_SE\_RANGE : start can't be greater than end\n");

if (error == ERROR\_VALUE\_RANGE)

printf("ERROR\_VALUE\_RANGE : wrong range of value\n");

if (error == ERROR\_NOT\_FOUND)

printf("ERROR\_NOT\_FOUND : can not find command\n");

if (error == ERROR\_FILE\_OPEN)

printf("ERROR\_FILE\_OPEN : fail to open file\n");

if (error == ERROR\_FILE\_CLOSE)

printf("ERROR\_FILE\_CLOSE : fail to close file\n");

if (error == ERROR\_NOT\_MNEMONIC)

printf("ERROR\_NOT\_MNEMONIC : can not find the mnemonic\n");

if (error == ERROR\_DIR\_OPEN)

printf("ERROR\_DIR\_OPEN : can not read information of current directory\n");

if (error == ERROR\_DIR\_CLOSE)

printf("ERROR\_DIR\_CLOSE : can not close current directory\n");

if (error == ERROR\_EMPTY\_ASM)

printf("ERROR\_EMPTY\_ASM : It is empty asm file\n");

if (error == ERROR\_WRONG\_FORMAT)

printf("ERROR\_WRONG\_FORMAT : please set right format\n");

if (error == ERROR\_DUP\_SYM)

printf("ERROR\_DUP\_SYM : duplicate symbol error\n");

if (error == ERROR\_NOT\_END)

printf("ERROR\_NOT\_END : can not find program end directive\n");

if (error == ERROR\_DEC\_RANGE)

printf("ERROR\_DEC\_RANGE : wrong range of decimal\n");

if (error == ERROR\_DEC\_FORMAT)

printf("ERROR\_DEC\_FORMAT : wrong format of decimal\n");

if (error == ERROR\_NOT\_ASM)

printf("ERROR\_NOT\_ASM : it is not asm file\n");

if (error == ERROR\_NOT\_SYM)

printf("ERROR\_NOT\_SYM : there is not the symbol in symbol table\n");

if (error == ERROR\_ADDR\_MODE)

printf("ERROR\_ADDR\_MODE : can not find right addressing mode\n");

if (error == ERROR\_NOT\_OBJ)

printf("ERROR\_NOT\_OBJ : it is not obj file\n");

if (error == ERROR\_FILE\_SEEK)

printf("ERROR\_FILE\_SEEK : can not move buffer cursor to start of file\n");

if (error == ERROR\_LINK\_CNT)

printf("ERROR\_LINK\_CNT : over maximum link count %d\n", \_MAX\_LINK);

if (error == ERROR\_MDF\_OVERFLOW)

printf("ERROR\_MDF\_OVERFLOW : value overflow by modification code\n");

if (error == ERROR\_NOT\_INSTRUCT)

printf("ERROR\_NOT\_INSTRUCT : can not run the no exist instruction\n");

}

//error\_print with error line

void asm\_error(int error, int line)

{

printf("LINE %d : ", line);

error\_print(error);

}

**5.9 assemble.h assemble.c //어셈블 기능 담당**

#ifndef \_\_ASSEMBLE\_H\_\_

#define \_\_ASSEMBLE\_H\_\_

#include "main.h"

#include "hash.h"

#include "argument.h"

typedef struct symbol\_node\* snptr;

typedef struct symbol\_node

{

char symbol[SYMBOL\_SIZE];

int loc;

}symbol\_node;

typedef struct instruction\* inptr;

typedef struct instruction

{

char \*comment;

char \*label;

char \*directive;

char \*mnemonic;

char \*operand;

int loc;

int line;

inptr link;

}instruction;

typedef struct object\_code

{

int code;

int format;

}object\_code;

//in next project, create file io source code

int cmd\_assemble(char filename[]);

void cmd\_symbol();

int read\_line(FILE\* fp, char buffer[], char tmp\_buffer[], int \*line);

inptr make\_instruction(char arg[], int line);

void clear\_intermediate();

void add\_symbol(char symbol[], int loc);

snptr find\_symbol(char symbol[]);

void add\_instruction(inptr pnew);

int assemble\_pass1(FILE \*asm\_fp);

int assemble\_pass2(FILE\* list\_fp, FILE\* obj\_fp);

int make\_obj(inptr cur, object\_code \*obj);

int obj\_to\_str(object\_code obj, char str[]);

void write\_list\_line(FILE \*fp, inptr cur, char obj[]);

void write\_obj\_line(FILE \*fp, char type, int record\_start, char record[]);

int is\_directive(char directive[]);

int is\_mnemonic(char mnemonic[]);

int is\_register(char reg[]);

int cmp\_string(void\* a, void\* b);

hnptr\* get\_sym\_table\_addr();

#endif

#include "assemble.h"

hnptr sym\_table[HASH\_SIZE] = { 0, };

inptr intermediate = NULL;

inptr rear = NULL;

int base\_flag = 0;

int base = 0;

char program\_name[7];

int program\_length = 0;

int sym\_table\_num = 0;

int modification\_table[1000];

int mdf\_num = 0;

//assembler command

//assemble asm file to obj, lst file

//if error, print error and return 0

int cmd\_assemble(char filename[])

{

clear\_hash(sym\_table);

sym\_table\_num =0;

mdf\_num = 0;

base\_flag = 0;

base = 0;

intermediate = NULL;

rear = NULL;

strcpy(program\_name, "NONAME");

program\_length = 0;

int len = strlen(filename);

if (len <= 4 || strcmp(filename + len - 4, ".asm") != 0) //NOT asm file

{

error\_print(ERROR\_NOT\_ASM);

return 0;

}

FILE\* asm\_fp = fopen(filename, "r");

if (asm\_fp == NULL)

{

error\_print(ERROR\_FILE\_OPEN);

return 0;

}

filename[len - 4] = '\0';

strcat(filename, ".lst");

FILE \*list\_fp = fopen(filename, "w");

if (list\_fp == NULL)

{

error\_print(ERROR\_FILE\_OPEN);

return 0;

}

clear\_hash(sym\_table); //sym\_table initialize

sym\_table\_num = 0;

filename[len - 4] = '\0';

strcat(filename, ".obj");

FILE \*obj\_fp = fopen(filename, "w");

if (obj\_fp == NULL)

{

error\_print(ERROR\_FILE\_OPEN);

return 0;

}

add\_symbol("A", 0);

add\_symbol("X", 1);

add\_symbol("L", 2);

add\_symbol("B", 3);

add\_symbol("S", 4);

add\_symbol("T", 5);

add\_symbol("F", 6);

add\_symbol("PC", 8);

add\_symbol("SW", 9);

int success = 0;

if (assemble\_pass1(asm\_fp) && assemble\_pass2(list\_fp, obj\_fp))

success = 1;

else

{

clear\_hash(sym\_table);

char currentPath[\_MAX\_PATH];

getcwd(currentPath, \_MAX\_PATH);

strcat(currentPath,"/");

strcat(currentPath,filename);

remove(currentPath);

currentPath[strlen(currentPath) - 4] = '\0';

strcat(currentPath, ".lst");

remove(currentPath);

}

clear\_intermediate();

if (fclose(asm\_fp) == EOF)

{

error\_print(ERROR\_FILE\_CLOSE);

return 0;

}

if (fclose(list\_fp) == EOF)

{

error\_print(ERROR\_FILE\_CLOSE);

return 0;

}

if (fclose(obj\_fp) == EOF)

{

error\_print(ERROR\_FILE\_CLOSE);

return 0;

}

filename[len - 4] = '\0';

printf("output file : [%s.lst], [%s.obj]\n",filename,filename);

return success;

}

//symbol command

//print all symbol in symbol table

void cmd\_symbol()

{

if (sym\_table\_num == 0)

{

printf("symbol table is empty\n");

return;

}

snptr \*tmp = (snptr\*)malloc(sizeof(\*tmp) \* sym\_table\_num);

int k = 0;

for (int i = 0; i < HASH\_SIZE; i++)

{

for (hnptr p = sym\_table[i]; p; p = p->link)

{

snptr sym = p->data;

if (!is\_register(sym->symbol))

tmp[k++] = sym;

}

}

for (int i = 0; i < k; i++)

{

for (int j = 0; j < k - i-1; j++)

{

if (cmp\_string(tmp[j], tmp[j + 1]))

{

snptr t = tmp[j];

tmp[j] = tmp[j + 1];

tmp[j + 1] = t;

}

}

}

//qsort(tmp, k, sizeof(\*tmp), cmp\_string);

for (int i = 0; i < k; i++)

{

printf("\t%s\t%04X\n", tmp[i]->symbol, tmp[i]->loc);

}

free(tmp);

return;

}

//string comparison in increasing

int cmp\_string(void\* a,void\* b)

{

return strcmp(((snptr)a)->symbol , ((snptr)b)->symbol) < 0;

}

//assembler pass1 process

//allocate location address to each instruction

//add symbol to hash table

//create intermediate data

int assemble\_pass1(FILE \*asm\_fp)

{

char tmp\_buffer[FILE\_BUFFER\_SIZE];

char buffer[FILE\_BUFFER\_SIZE];

char \*arg;

char \*parse;

int len;

int line = 0;

if(read\_line(asm\_fp, buffer, tmp\_buffer,&line) == 0)

asm\_error(ERROR\_EMPTY\_ASM, line); //empty asm file

inptr instruct = make\_instruction(buffer, line);

if (instruct == NULL) //make\_instruction error

return 0;

int locctr = 0;

int start\_addr = 0;

if (instruct->directive != NULL && strcmp(instruct->directive , "START") == 0) //opcode == start

{

arg = instruct->operand;

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

asm\_error(ERROR\_LACK\_ARG, line);

return 0; //not start addr error

}

start\_addr = hex\_to\_int(parse);

if (start\_addr < 0)

{

asm\_error(start\_addr, line);

return 0;

}

if (instruct->label)

strcpy(program\_name, instruct->label);

locctr = start\_addr;

instruct->loc = start\_addr;

}

while (read\_line(asm\_fp, buffer, tmp\_buffer, &line))

{

instruct = make\_instruction(buffer, line);

if (instruct == NULL)

return 0;

if (instruct->directive != NULL && strcmp(instruct->directive, "END") == 0)

break;

if (instruct->comment == NULL) //not comment

{

int tmp = locctr;

if (instruct->label != NULL) //there is a symbol in the label field

{

if (find\_symbol(instruct->label) != NULL) //symbol duplicate

{

asm\_error(ERROR\_DUP\_SYM, line);

return 0;

}

add\_symbol(instruct->label, locctr);

}

if (instruct->mnemonic != NULL) //mnemonic

{

instruct->loc = tmp;

int extend\_flag = 0;

onptr op;

if (instruct->mnemonic[0] == '+')

{

extend\_flag = 1;

op = find\_mnemonic(instruct->mnemonic + 1);

}

else

op = find\_mnemonic(instruct->mnemonic);

if (op->format[0] == '1')

locctr += 1;

else if (op->format[0] == '2')

locctr += 2;

else if (op->format[0] == '3')

locctr += 3 + extend\_flag;

else

{

asm\_error(ERROR\_WRONG\_FORMAT, line);

return 0;

}

}

else if (instruct->directive != NULL) //directive

{

if (strcmp(instruct->directive, "WORD") == 0)

{

instruct->loc = tmp;

locctr += 3;

}

if (strcmp(instruct->directive, "RESW") == 0)

{

arg = instruct->operand;

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

asm\_error(ERROR\_LACK\_ARG,line);

return 0;

}

int num = dec\_to\_int(parse);

if (num < 0)

{

asm\_error(num, line);

return 0;

}

instruct->loc = tmp;

locctr += num \* 3;

}

if (strcmp(instruct->directive, "RESB") == 0)

{

arg = instruct->operand;

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

asm\_error(ERROR\_LACK\_ARG, line);

return 0;

}

int num = dec\_to\_int(parse);

if (num < 0)

{

asm\_error(num, line);

return 0;

}

instruct->loc = tmp;

locctr += num;

}

if (strcmp(instruct->directive, "BYTE") == 0)

{

arg = instruct->operand;

len = parsing\_argument(&arg, &parse);

if (len <= 3)

{

asm\_error(ERROR\_LACK\_ARG, line);

return 0;

}

instruct->loc = tmp;

if (parse[0] == 'C' && parse[1] == '\'' && parse[len - 1] == '\'')

locctr += len - 3;

else if (parse[0] == 'X' && parse[1] == '\'' && parse[len - 1] == '\'')

locctr += (len - 2) / 2;

else

{

asm\_error(ERROR\_WRONG\_FORMAT, line);

return 0;

}

}

}

}

}

if (instruct->directive != NULL && strcmp(instruct->directive, "END") != 0)

{

asm\_error(ERROR\_NOT\_END, line);

return 0;

}

program\_length = locctr - start\_addr;

return 1;

}

//assembler pass2 process

//write list file and obj file

int assemble\_pass2(FILE\* list\_fp, FILE\* obj\_fp)

{

inptr cur = intermediate;

char \*arg = cur->operand;

char \*parse;

int len;

int record\_start = 0;

int start\_addr = 0;

object\_code obj;

char obj\_str[OBJCODE\_SIZE];

char record[RECODE\_SIZE];

if (cur->directive && strcmp(cur->directive, "START") == 0)

{

arg = cur->operand;

len = parsing\_argument(&arg, &parse);

start\_addr = hex\_to\_int(parse);

len = parsing\_argument(&arg, &parse);

if (len)

{

asm\_error(ERROR\_OVER\_ARG, cur->line);

return 0;

}

write\_list\_line(list\_fp, cur, obj\_str);

cur = cur->link;

}

write\_obj\_line(obj\_fp,'H', start\_addr, NULL);

record[0] = '\0';

record\_start = start\_addr;

while (cur)

{

if (cur->directive && strcmp(cur->directive, "END") == 0)

break;

if (cur->comment == NULL)

{

if (cur->directive != NULL && strcmp(cur->directive, "BASE") == 0)

{

base\_flag = 1;

arg = cur->operand;

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

asm\_error(ERROR\_LACK\_ARG, cur->line);

return 0;

}

snptr symbol = find\_symbol(parse);

if (symbol == NULL)

{

asm\_error(ERROR\_NOT\_SYM, cur->line);

return 0;

}

base = symbol->loc;

len = parsing\_argument(&arg, &parse);

if (len)

{

asm\_error(ERROR\_OVER\_ARG, cur->line);

return 0;

}

}

else if (cur->directive != NULL && strcmp(cur->directive, "NOBASE") == 0)

{

base\_flag = 0;

arg = cur->operand;

len = parsing\_argument(&arg, &parse);

if (len)

{

asm\_error(ERROR\_OVER\_ARG, cur->line);

return 0;

}

}

else

{

if (make\_obj(cur, &obj) == 0)

return 0;

len = obj\_to\_str(obj, obj\_str);

if (len < 0)

{

asm\_error(len, cur->line);

return 0;

}

if (record\_start + 0x1D <= cur->loc) //new line

{

write\_obj\_line(obj\_fp, 'T', record\_start, record);

record[0] = '\0';

record\_start = cur->loc;

}

strcat(record, obj\_str); //add object code in text record

}

}

write\_list\_line(list\_fp, cur, obj\_str);

cur = cur->link;

}

write\_obj\_line(obj\_fp,'T', record\_start, record);

write\_obj\_line(obj\_fp,'M',0,NULL);

write\_obj\_line(obj\_fp,'E', start\_addr , NULL);

write\_list\_line(list\_fp, cur, obj\_str);

return 1;

}

//read line from file pointer fp

//and refine this line and store in buffer

//calculate line

int read\_line(FILE\* fp, char buffer[], char tmp\_buffer[], int \*line)

{

do

{

\*line += 5;

if (fgets(tmp\_buffer, FILE\_BUFFER\_SIZE, fp) == NULL)

return 0;

} while (tmp\_buffer[0] == '\n');

refine\_argument(tmp\_buffer, buffer);

return 1;

}

//make instruction from argument

//categorize instruction (ex. comment directive, mnemonic)

//create intermediate file

inptr make\_instruction(char arg[], int line)

{

inptr pnew = (inptr)malloc(sizeof(\*pnew));

pnew->comment = pnew->label = pnew->directive = pnew->mnemonic = pnew->operand = NULL;

pnew->link = NULL;

pnew->loc = 0;

pnew->line = line;

char \*parse;

int len = parsing\_argument(&arg, &parse);

if (parse[0] == '.') //comment

{

pnew->comment = (char\*)malloc(sizeof(char)\*INSTRUCTION\_BUFFER);

int start = 0;

strcpy((pnew->comment) + start, parse + 1);

start = len - 1;

while (len = parsing\_argument(&arg, &parse))

{

strcpy((pnew->comment) + start, parse);

start += len;

}

pnew->comment[strlen(pnew->comment) + 1] = '\0';

add\_instruction(pnew);

return pnew;

}

if (is\_mnemonic(parse) == 0 && is\_directive(parse) == 0) //label

{

pnew->label = (char\*)malloc(sizeof(char)\*INSTRUCTION\_BUFFER);

strcpy(pnew->label, parse);

len = parsing\_argument(&arg, &parse);

pnew->label[strlen(pnew->label) + 1] = '\0';

}

if (is\_mnemonic(parse)) //mnemonic

{

pnew->mnemonic = (char\*)malloc(sizeof(char)\*INSTRUCTION\_BUFFER);

strcpy(pnew->mnemonic, parse);

pnew->mnemonic[strlen(pnew->mnemonic) + 1] = '\0';

}

else if (is\_directive(parse))//directive

{

pnew->directive = (char\*)malloc(sizeof(char)\*INSTRUCTION\_BUFFER);

strcpy(pnew->directive, parse);

pnew->directive[strlen(pnew->directive) + 1] = '\0';

}

else

{

asm\_error(ERROR\_WRONG\_FORMAT, line);

return NULL;

}

pnew->operand = (char\*)malloc(sizeof(char)\*INSTRUCTION\_BUFFER);

pnew->operand[0] = '\0';

int start = 0;

while (len = parsing\_argument(&arg, &parse))

{

strcpy((pnew->operand) + start, parse);

start += len+1;

}

\*((pnew->operand) + start) = '\0';

add\_instruction(pnew);

return pnew;

}

//clear intermediate file

void clear\_intermediate()

{

inptr p = intermediate;

while (p)

{

inptr next = p->link;

if (p->comment)

free(p->comment);

if (p->label)

free(p->label);

if (p->mnemonic)

free(p->mnemonic);

if (p->operand)

free(p->operand);

free(p);

p = next;

}

intermediate = NULL;

rear = NULL;

}

//add instruction to intermediate file

void add\_instruction(inptr pnew)

{

if (intermediate == NULL)

{

intermediate = pnew;

rear = pnew;

return;

}

rear->link = pnew;

rear = pnew;

}

//return 1 if string == mnemonic else return 0

int is\_mnemonic(char mnemonic[])

{

if (find\_mnemonic(mnemonic) != NULL)

return 1;

if (mnemonic[0] == '+' && find\_mnemonic(mnemonic + 1) != NULL)

return 1;

return 0;

}

//return 1 if string == directive else return 0

int is\_directive(char directive[])

{

if (strcmp(directive, "START") == 0)

return 1;

if (strcmp(directive, "END") == 0)

return 1;

if (strcmp(directive, "BASE") == 0)

return 1;

if (strcmp(directive, "NOBASE") == 0)

return 1;

if (strcmp(directive, "BYTE") == 0)

return 1;

if (strcmp(directive, "WORD") == 0)

return 1;

if (strcmp(directive, "RESB") == 0)

return 1;

if (strcmp(directive, "RESW") == 0)

return 1;

return 0;

}

//return 1 if string == register else return 0

int is\_register(char reg[])

{

if (strcmp(reg, "A") == 0)

return 1;

if (strcmp(reg, "X") == 0)

return 1;

if (strcmp(reg, "L") == 0)

return 1;

if (strcmp(reg, "B") == 0)

return 1;

if (strcmp(reg, "S") == 0)

return 1;

if (strcmp(reg, "T") == 0)

return 1;

if (strcmp(reg, "F") == 0)

return 1;

if (strcmp(reg, "PC") == 0)

return 1;

if (strcmp(reg, "SW") == 0)

return 1;

return 0;

}

//add symbol to symbol table

void add\_symbol(char symbol[], int loc)

{

int hash\_number = hash\_function(symbol);

snptr pnew = (snptr)malloc(sizeof(\*pnew));

strcpy(pnew->symbol, symbol);

pnew->loc = loc;

add\_hash(sym\_table, pnew, hash\_number);

sym\_table\_num++;

}

//find symbol in symbol table if not found return NULL

snptr find\_symbol(char symbol[])

{

int hash\_number = hash\_function(symbol);

for (hnptr p = sym\_table[hash\_number]; p; p = p->link)

{

snptr sp = p->data;

if (strcmp(sp->symbol, symbol) == 0)

return sp;

}

return NULL;

}

//make object code from intermediate node

//return 1 if success else return 0

int make\_obj(inptr cur, object\_code \*obj)

{

int extend\_flag = 0;

int indexed\_flag = 1;

int immediate\_flag = 1;

int x\_flag = 0;

char \*arg = cur->operand;

char \*parse;

int len;

int disp;

obj->format = 0; //not object code

obj->code = 0;

if (cur->mnemonic != NULL)

{

char \*mnemonic = cur->mnemonic;

if (mnemonic[0] == '+')

{

extend\_flag = 1;

mnemonic++;

}

onptr op = find\_mnemonic(mnemonic);

snptr symbol = NULL;

int constant = 0;

if (op->format[0] != '3' && extend\_flag == 1) // extend format

{

asm\_error(ERROR\_WRONG\_FORMAT, cur->line);

return 0;

}

if (op->format[0] == '3') //format 3 or 4

{

if (extend\_flag)

obj->format = 4;

else

obj->format = 3;

len = parsing\_argument(&arg, &parse);

if (strcmp(op->mnemonic, "RSUB") == 0)

{

if (len)

{

asm\_error(ERROR\_OVER\_ARG, cur->line);

return 0;

}

obj->code = op->opcode + 3; //simple addressing mode

if (extend\_flag)

obj->code \*= 1 << 24;

else

obj->code \*= 1 << 16;

return 1;

}

if (len == 0)

{

asm\_error(ERROR\_LACK\_ARG, cur->line);

return 0;

}

if (parse[0] == '@')

{

parse++;

indexed\_flag = 1;

immediate\_flag = 0;

}

else if (parse[0] == '#')

{

parse++;

indexed\_flag = 0;

immediate\_flag = 1;

}

snptr symbol = find\_symbol(parse);

int constant = 0;

if (symbol == NULL) //not symbol

{

constant = dec\_to\_int(parse);

if (constant < 0) //not symbol not constant

{

asm\_error(constant, cur->line);

return 0;

}

}

len = parsing\_argument(&arg, &parse);

if (len && indexed\_flag == 1 && immediate\_flag == 1 && parse[0] == ',') //x flag only simple

{

len = parsing\_argument(&arg, &parse);

if (len && strcmp(parse, "X") == 0)

x\_flag = 1;

else

{

asm\_error(ERROR\_WRONG\_FORMAT, cur->line);

return 0;

}

len = parsing\_argument(&arg, &parse);

if (len)

{

asm\_error(ERROR\_OVER\_ARG, cur->line);

return 0;

}

}

else if (len)

{

asm\_error(ERROR\_OVER\_ARG, cur->line);

return 0;

}

obj->code = (op->opcode) >> 2;

obj->code = obj->code \* 2 + indexed\_flag;

obj->code = obj->code \* 2 + immediate\_flag;

obj->code = obj->code \* 2 + x\_flag;

if (extend\_flag)

{

obj->code = obj->code \* 8 + extend\_flag;

obj->code \*= 1 << 20;

if (immediate\_flag == 1 && indexed\_flag == 0)

{

if (symbol == NULL)

disp = constant;

else

disp = symbol->loc;

}

if (indexed\_flag == 1) //indexed addressing mode + simple addressing\_mode

{

if (symbol == NULL)

{

asm\_error(ERROR\_NOT\_SYM, cur->line);

return 0;

}

disp = symbol->loc;

modification\_table[mdf\_num++] = cur->loc;

}

obj->code += disp;

}

else //format 3

{

obj->code \*= 1 << 15;

if (immediate\_flag == 1 && indexed\_flag == 0)

{

if (symbol == NULL)

disp = constant;

else

disp = symbol->loc;

}

if (indexed\_flag == 1) //simple, indexed

{

if (symbol == NULL)

{

asm\_error(ERROR\_NOT\_SYM, cur->line);

return 0;

}

disp = symbol->loc;

}

if (symbol != NULL)

{

obj->code += (1 << 13);

disp -= (cur->loc + 3);

if (disp < -2048 || disp > 2047) //base relative

{

obj->code += (1 << 14) - (1 << 13); //base relative?

disp = symbol->loc - base;

if (disp < 0 || disp > 4095 || base\_flag == 0)

{

asm\_error(ERROR\_ADDR\_MODE, cur->line);

return 0;

}

}

else

{

disp = disp & 0x00000fff;

}

}

if (extend\_flag)

{

if (disp >= 1 << 20)

{

asm\_error(ERROR\_VALUE\_RANGE, cur->line);

return 0;

}

}

else

{

if (disp >= 1 << 12)

{

asm\_error(ERROR\_VALUE\_RANGE, cur->line);

return 0;

}

}

obj->code += disp;

}

}

else if (op->format[0] == '2')

{

snptr symbol = NULL;

int r1 = FLAG\_EMPTY;

int r2 = FLAG\_EMPTY;

arg = cur->operand;

obj->format = 2;

len = parsing\_argument(&arg, &parse);

if (len == 0)

{

asm\_error(ERROR\_LACK\_ARG, cur->line);

return 0;

}

symbol = find\_symbol(parse);

if (symbol == NULL)

{

r1 = hex\_to\_int(parse);

if (r1 < 0)

{

asm\_error(r1, cur->line);

return 0;

}

if (r1 >= 16)

{

asm\_error(ERROR\_VALUE\_RANGE, cur->line);

return 0;

}

}

else

r1 = symbol->loc;

len = parsing\_argument(&arg, &parse);

if (len)

{

if (parse[0] == ',')

{

len = parsing\_argument(&arg, &parse);

symbol = find\_symbol(parse);

if (symbol == NULL)

{

r2 = hex\_to\_int(parse);

if (r2 < 0)

{

asm\_error(r2, cur->line);

return 0;

}

if (r2 >= 16)

{

asm\_error(ERROR\_VALUE\_RANGE, cur->line);

return 0;

}

}

else

r2 = symbol->loc;

len = parsing\_argument(&arg, &parse);

if (len)

{

asm\_error(ERROR\_OVER\_ARG, cur->line);

return 0;

}

}

else

{

asm\_error(ERROR\_WRONG\_FORMAT, cur->line);

return 0;

}

}

if (strcmp(op->mnemonic, "CLEAR") == 0)

{

if (r1 == FLAG\_EMPTY || r2 != FLAG\_EMPTY)

{

asm\_error(ERROR\_WRONG\_FORMAT, cur->line);

return 0;

}

}

else if (strcmp(op->mnemonic, "SVC") == 0 )

{

if (r1 == FLAG\_EMPTY || r2 != FLAG\_EMPTY)

{

asm\_error(ERROR\_WRONG\_FORMAT, cur->line);

return 0;

}

}

else if (strcmp(op->mnemonic, "TIXR") == 0)

{

if(r1 == FLAG\_EMPTY || r2 != FLAG\_EMPTY)

{

asm\_error(ERROR\_WRONG\_FORMAT, cur->line);

return 0;

}

}

else if ((r1 == FLAG\_EMPTY || r2 == FLAG\_EMPTY))

{

if (r1 == FLAG\_EMPTY || r2 != FLAG\_EMPTY)

{

asm\_error(ERROR\_WRONG\_FORMAT, cur->line);

return 0;

}

}

if (r2 == FLAG\_EMPTY)

r2 = 0;

obj->code = (op->opcode);

obj->code = obj->code << 4;

obj->code += r1;

obj->code = obj->code << 4;

obj->code += r2;

}

else if (op->format[0] == '1')

{

obj->format = 1;

len = parsing\_argument(&arg, &parse);

if (len)

{

asm\_error(ERROR\_OVER\_ARG, cur->line);

return 0;

}

obj->code += op->opcode;

}

}

else if (cur->directive != NULL)

{

obj->code = 0;

if (strcmp(cur->directive, "BYTE") == 0)

{

len = parsing\_argument(&arg, &parse);

if (parse[0] == 'C')

{

for (int i = 2; i < len-1; i++)

{

obj->code \*= 1 << 8;

obj->code += parse[i];

}

obj->format = (len - 3) \* 2;

obj->format = -obj->format;

}

else if (parse[0] == 'X')

{

for (int i = 2; i < len - 1; i++)

{

obj->code \*= 1 << 4;

if ('0' <= parse[i] && parse[i] <= '9')

obj->code += parse[i] - '0';

else if ('A' <= parse[i] && parse[i] <= 'Z')

obj->code += parse[i] - 'A' + 10;

else

{

asm\_error(ERROR\_HEX\_FORMAT, cur->line);

return 0;

}

}

obj->format = (len - 2) / 2 \* 2;

obj->format = -obj->format;

}

len = parsing\_argument(&arg, &parse);

if (len)

{

asm\_error(ERROR\_OVER\_ARG, cur->line);

return 0;

}

}

else if (strcmp(cur->directive, "WORD") == 0)

{

len = parsing\_argument(&arg, &parse);

int num = dec\_to\_int(parse);

if (num < 0)

{

asm\_error(num, cur->line);

return 0;

}

if (num > 0xFFFFFF)

{

asm\_error(ERROR\_VALUE\_RANGE, cur->line);

return 0;

}

obj->code = num;

}

}

return 1;

}

//transform obj code to string

//if sucess return 1 else return 0

int obj\_to\_str(object\_code obj, char str[])

{

int len = 0;

int error = 0;

if (obj.format == 0)

{

str[0] = '\0';

return 0;

}

if (obj.format == 1)

{

error = int\_to\_hex(obj.code, str, 2);

if (error < 0)

{

return error;

}

return 2;

}

if (obj.format == 2)

{

error = int\_to\_hex(obj.code, str, 4);

if (error < 0)

{

return error;

}

return 4;

}

if (obj.format == 3)

{

error = int\_to\_hex(obj.code, str, 6);

if (error < 0)

{

return error;

}

return 6;

}

if (obj.format == 4)

{

error = int\_to\_hex(obj.code, str, 8);

if (error < 0)

{

return error;

}

return 8;

}

if (obj.format < 0)

{

error = int\_to\_hex(obj.code, str, -obj.format);

if (error < 0)

{

return error;

}

return -obj.format;

}

}

//write 1 line in obj file

void write\_obj\_line(FILE \*fp, char type, int record\_start, char record[])

{

if (type == 'H')

{

fprintf(fp, "H%-6s",program\_name);

fprintf(fp, "%06X", record\_start);

fprintf(fp, "%06X\n", program\_length);

}

if (type == 'T')

{

int len = strlen(record);

fprintf(fp, "T%06X", record\_start);

fprintf(fp, "%02X", len/2);

fprintf(fp, "%s\n", record);

}

if (type == 'E')

{

fprintf(fp, "E%06X\n", record\_start);

}

if(type == 'M')

{

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

{

fprintf(fp,"M%06X05\n",modification\_table[i] + 1);

}

}

}

//write 1 line to list file

void write\_list\_line(FILE \*fp, inptr cur, char obj[])

{

fprintf(fp, "%-5d\t", cur->line);

if (cur->comment)

{

fprintf(fp, ". %s\n", cur->comment);

return;

}

if (cur->directive != NULL && (strcmp(cur->directive, "BASE") == 0 || strcmp(cur->directive, "NOBASE") == 0 || strcmp(cur->directive, "END") == 0))

fprintf(fp, "%4s\t", " ");

else

fprintf(fp, "%04X\t", cur->loc);

if (cur->label)

fprintf(fp, "%-10s", cur->label);

else

fprintf(fp, "%-10s","");

if (cur->directive)

fprintf(fp, "%-10s\t", cur->directive);

if (cur->mnemonic)

fprintf(fp, "%-10s\t", cur->mnemonic);

if (cur->operand)

{

char \*arg = cur->operand;

char \*parse;

int len;

while (len = parsing\_argument(&arg, &parse))

{

parse[len] = ' ';

}

fprintf(fp, "%-20s", cur->operand);

}

if (cur->directive != NULL && (strcmp(cur->directive, "BASE") == 0 || strcmp(cur->directive, "NOBASE") == 0 || strcmp(cur->directive, "START") == 0 || strcmp(cur->directive, "END") == 0))

;

else

fprintf(fp, "%s", obj);

fprintf(fp,"\n");

}

//get sym\_table address

hnptr\* get\_sym\_table\_addr()

{

return sym\_table;

}

**5.9 linkloader.h linkloader.c //링크 로더 기능 담당**

#ifndef \_\_LINKLOADER\_H\_\_

#define \_\_LINKLOADER\_H\_\_

#include "main.h"

#include "hash.h"

#include "argument.h"

#include "memory.h"

typedef struct esymbol\_node\* esnptr;

typedef struct esymbol\_node

{

char symbol[SYMBOL\_SIZE];

int loc;

esnptr link;

}esymbol\_node;

typedef struct LOAD\_MAP

{

esnptr head[\_MAX\_LINK];

esnptr rear[\_MAX\_LINK];

int prog\_length[\_MAX\_LINK];

int link\_cnt;

}LOAD\_MAP;

extern int prog\_addr;

int cmd\_loader(char filename[\_MAX\_LINK][INPUT\_SIZE], int file\_cnt);

int loader\_pass1(FILE\* fp[], int file\_cnt);

int loader\_pass2(FILE\* fp[], int file\_cnt);

int obj\_read\_line(FILE\* fp, char buffer[]);

void add\_esymbol(char esymbol[], int loc, int idx);

void add\_load\_map(esnptr esym\_node, int idx);

esnptr find\_esymbol(char esymbol[]);

void print\_load\_map();

void cmd\_progaddr(int value);

#endif

#include "linkloader.h"

int prog\_addr = 0; //default 0x00

int exe\_addr = 0;

hnptr esym\_table[HASH\_SIZE] = { 0, };

LOAD\_MAP\* load\_map = NULL;

int cmd\_loader(char filename[\_MAX\_LINK][INPUT\_SIZE], int file\_cnt)

{

FILE\* fp[\_MAX\_LINK] = { 0, };

for (int i = 0; i < HASH\_SIZE; i++)

esym\_table[i] = NULL;

load\_map = (LOAD\_MAP\*)malloc(sizeof(\*load\_map));

for (int i = 0; i < \_MAX\_LINK; i++)

{

load\_map->head[i] = NULL;

load\_map->rear[i] = NULL;

load\_map->prog\_length[i] = 0;

}

load\_map->link\_cnt = file\_cnt;

for (int i = 0; i < file\_cnt; i++)

{

int len = strlen(filename[i]);

if (len <= 4 || strcmp(filename[i] + len - 4, ".obj") != 0) //NOT obj file

{

error\_print(ERROR\_NOT\_OBJ);

return 0;

}

fp[i] = fopen(filename[i], "r");

if (fp[i] == NULL)

{

error\_print(ERROR\_FILE\_OPEN);

return 0;

}

} //file open

int success = 1;

if (loader\_pass1(fp,file\_cnt))

{

//init file buf

for (int i = 0; i < file\_cnt; i++)

{

if (fseek(fp[i], 0L, SEEK\_SET) != 0)

{

error\_print(ERROR\_FILE\_SEEK);

success = 0;

break;

}

}

if (success && loader\_pass2(fp,file\_cnt))

{

print\_load\_map();

}

else

success = 0;

}

else

success = 0;

clear\_hash(esym\_table);

free(load\_map);

load\_map = NULL;

for (int i = 0; i < file\_cnt; i++)

{

if (fclose(fp[i]) == EOF)

{

error\_print(ERROR\_FILE\_CLOSE);

success = 0;

}

}

//file close

return success;

}

int loader\_pass1(FILE\* fp[], int file\_cnt)

{

int cs\_addr = prog\_addr; //control section addr

int cs\_length = 0;

char buffer[FILE\_BUFFER\_SIZE];

char parse[FILE\_BUFFER\_SIZE];

char record\_type = 0;

for (int i = 0; i < file\_cnt; i++)

{

record\_type = obj\_read\_line(fp[i], buffer); //read header record

strncpy(parse, buffer + 13, 6); //program length. HPROGA 000000(000063)

parse[6] = '\0';

cs\_length = hex\_to\_int(parse);

load\_map->prog\_length[i] = cs\_length;

strncpy(parse, buffer + 1, 6); //program name

parse[6] = '\0';

if (find\_esymbol(parse))

{

error\_print(ERROR\_DUP\_SYM);

return 0;

//prgram name duplicate external symbol error

}

add\_esymbol(parse, cs\_addr,i);

while (record\_type = obj\_read\_line(fp[i], buffer))

{

if (record\_type == 'E')

break;

if (record\_type == 'D')

{

int len = strlen(buffer);

int cursor = 1;

while (cursor < len)

{

strncpy(parse, buffer + cursor + 6, 6);

parse[6] = '\0';

int loc = hex\_to\_int(parse); //normal input assume

strncpy(parse, buffer + cursor, 6);

parse[6] = '\0';

if (find\_esymbol(parse))

{

error\_print(ERROR\_DUP\_SYM);

return 0;

}

add\_esymbol(parse,cs\_addr+loc, i);

cursor += 12;

}

}

}

cs\_addr += cs\_length;

}

if (cs\_addr > 0x100000)

{

error\_print(ERROR\_MEM\_BOUND);

return 0;

}

return 1;

}

int loader\_pass2(FILE\* fp[], int file\_cnt)

{

int cs\_addr = prog\_addr;

exe\_addr = prog\_addr; //exe\_addr default

int cs\_length = 0;

char buffer[FILE\_BUFFER\_SIZE];

char parse[FILE\_BUFFER\_SIZE];

char record\_type = 0;

for (int i = 0; i < file\_cnt; i++)

{

esnptr refer[\_MAX\_REFER + 1] = { 0, };

refer[1] = load\_map->head[i]; //refer[1] = program esymbol

cs\_length = load\_map->prog\_length[i];

while (record\_type = obj\_read\_line(fp[i], buffer))

{

if (record\_type == 'E')

break;

if (record\_type == 'R')

{

int len = strlen(buffer);

int cursor = 1;

while (cursor < len)

{

strncpy(parse, buffer + cursor, 2);

parse[2] = '\0';

int idx = hex\_to\_int(parse);

strncpy(parse, buffer + cursor + 2, 6);

int tmp = 0;

while((tmp = strlen(parse)) < 6)

{

parse[tmp] = ' ';

parse[tmp+1] = '\0';

}

parse[6] = '\0';

esnptr esymbol = find\_esymbol(parse);

if (esymbol == NULL)

{

error\_print(ERROR\_NOT\_SYM);

return 0;

}

refer[idx] = esymbol;

cursor += 8;

}

}

if (record\_type == 'T')

{

strncpy(parse, buffer+1, 6);

parse[6] = '\0';

int start\_addr = hex\_to\_int(parse) + cs\_addr; //set start\_addr

strncpy(parse, buffer + 7, 2);

parse[2] = '\0';

int record\_len = hex\_to\_int(parse); //set record length

int cursor = 9;

for (int j = 0; j < record\_len; j++,cursor += 2)

{

strncpy(parse, buffer + cursor, 2);

parse[2] = '\0';

int value = hex\_to\_int(parse);

memory[start\_addr + j] = value;

} //write memory

}

if (record\_type == 'M')

{

strncpy(parse, buffer + 1, 6);

parse[6] = '\0';

int tar\_addr = hex\_to\_int(parse) + cs\_addr;

strncpy(parse, buffer + 7, 2);

parse[2] = '\0';

int mdf\_len = hex\_to\_int(parse);

int sign = 0; //sign = 0 is plus

strncpy(parse, buffer + 9, 1);

if (parse[0] == '-')

sign = 1;

strncpy(parse, buffer + 10, 2);

parse[2] = '\0';

int idx = hex\_to\_int(parse);

int mdf\_value = 0;

for (int j = 0; j < (mdf\_len + 1) / 2; j++)

{

mdf\_value \*= 0x100;

mdf\_value += memory[tar\_addr + j];

}//memory load

if (mdf\_len % 2) //mdf\_len = 5

mdf\_value = mdf\_value & 0x0FFFFF;

if (sign)

mdf\_value -= refer[idx]->loc;

else

mdf\_value += refer[idx]->loc;

//do not check overflow

if (mdf\_len % 2)

{

memory[tar\_addr] = memory[tar\_addr] & 0xF0;

memory[tar\_addr] += (mdf\_value & 0x0F0000) / 0x10000;

tar\_addr++;

}

for (int j = mdf\_len/2 - 1; j >= 0 ; j--)

{

memory[tar\_addr + j] = mdf\_value % 0x100;

mdf\_value /= 0x100;

}

}

}

if (record\_type == 'E') //if normal exit, break by end record

{

int len = strlen(buffer);

if (len > 1) //set exe\_addr

{

strncpy(parse, buffer + 1, 6);

exe\_addr = hex\_to\_int(parse) + cs\_addr;

}

}

cs\_addr += cs\_length;

}

return 1;

}

void cmd\_progaddr(int value)

{

prog\_addr = value;

}

//read line from right obj file (assume that we read right obj file)

//return record type and interchange first character by spacebar from record line

//if read end of file, return 0

int obj\_read\_line(FILE\* fp, char buffer[])

{

if (fgets(buffer, FILE\_BUFFER\_SIZE, fp) == NULL)

return 0;

int ret = buffer[0];

buffer[0] = ' ';

int len = strlen(buffer);

if(buffer[len-1] == '\n')

buffer[len-1] = '\0';

return ret;

}

void add\_esymbol(char esymbol[], int loc, int idx)

{

esymbol[6] = '\0'; //extern symbol size always be 6

int hash\_number = hash\_function(esymbol);

esnptr pnew = (esnptr)malloc(sizeof(\*pnew));

strcpy(pnew->symbol, esymbol);

pnew->loc = loc;

pnew->link = NULL;

add\_hash(esym\_table, pnew, hash\_number);

add\_load\_map(pnew, idx);

}

void add\_load\_map(esnptr esym\_node, int idx)

{

if (load\_map->head[idx] == NULL)

{

load\_map->head[idx] = load\_map->rear[idx] = esym\_node;

return;

}

(load\_map->rear[idx])->link = esym\_node;

load\_map->rear[idx] = esym\_node;

}

esnptr find\_esymbol(char esymbol[])

{

int hash\_number = hash\_function(esymbol);

for (hnptr p = esym\_table[hash\_number]; p; p = p->link)

{

esnptr esp = p->data;

if (strcmp(esp->symbol, esymbol) == 0)

return esp;

}

return NULL;

}

void print\_load\_map()

{

printf("control symbol address length\n");

printf("section name\n");

printf("----------------------------------------------------\n");

for (int i = 0; i < load\_map->link\_cnt; i++)

{

esnptr p = load\_map->head[i];

printf("%-13s%-13s%-16X%-13X\n",p->symbol," ",p->loc,load\_map->prog\_length[i]);

for (p = p->link; p; p = p->link)

printf("%-13s%-13s%-16X\n"," ", p->symbol, p->loc);

}

printf("----------------------------------------------------\n");

}

**5.10 debug.h debug.c //디버그 기능(break point 지정) 담당**

#ifndef \_\_DEBUG\_H\_\_

#define \_\_DEBUG\_H\_\_

#include "main.h"

extern int break\_point[];

void cmd\_bp\_show();

void cmd\_bp\_clear();

void cmd\_bp\_set(int value);

int cmp\_increasing(const void\* a,const void\* b);

int check\_bp(int value);

#endif

#include "debug.h"

int break\_point[BP\_SIZE] = { 0, };

int bp\_cnt = 0;

//show all break point

void cmd\_bp\_show()

{

if (bp\_cnt == 0)

{

printf("There not exist break point\n");

return ;

}

printf("breakpoint\n");

printf("----------\n");

for (int i = 0; i < bp\_cnt; i++)

{

printf("%05X\n", break\_point[i]);

}

return;

}

//clear all break point

void cmd\_bp\_clear()

{

bp\_cnt = 0;

printf("[ok] clear all breakpoints\n");

}

//set break point

void cmd\_bp\_set(int value)

{

if (!check\_bp(value))

{

break\_point[bp\_cnt++] = value;

qsort(break\_point, bp\_cnt, sizeof(int), cmp\_increasing);

}

printf("[ok] create breakpoint %05X\n", value);

}

//a purpose of the function is to sort break point

int cmp\_increasing(const void\* a,const void\* b)

{

return \*(int\*)a > \*(int\*)b;

}

//check if value is break point

//if true, return 1 else return 0

int check\_bp(int value)

{

int left = 0, right = bp\_cnt - 1;

while (left <= right)

{

int mid = left + right / 2;

if (break\_point[mid] == value)

return 1;

else if (break\_point[mid] < value)

left = mid + 1;

else

right = mid - 1;

}

return 0;

}

**5.11 simulator.h simulator.c //프로그램 실행 기능 담당**

#ifndef \_\_SIMULATOR\_H\_\_

#define \_\_SIMULATOR\_H\_\_

#include "main.h"

#include "linkloader.h"

#include "debug.h"

#include "memory.h"

extern unsigned int reg[16];

enum { A = 0, X = 1, L = 2, PC = 8, SW = 9, B = 3, S = 4, T = 5, F = 6 };

enum { LT, GT, EQ };

typedef enum addressing { EMPTY, IMMEDIATE, INDIRECT, INDEXED, RELATIVE\_B, RELATIVE\_PC, DIRECT, EXTEND }addressing;

typedef struct ADDR\_MODE

{

addressing m[4];

}ADDR\_MODE;

int cmd\_run();

void show\_register();

int instruction\_f1(int cur\_addr);

int inst\_fix();

int inst\_float();

int inst\_hio();

int inst\_norm();

int inst\_sio();

int inst\_tio();

int instruction\_f2(int cur\_addr);

int inst\_addr(int cur\_addr);

int inst\_clear(int cur\_addr);

int inst\_compr(int cur\_addr);

int inst\_divr(int cur\_addr);

int inst\_mulr(int cur\_addr);

int inst\_rmo(int cur\_addr);

int inst\_shiftl(int cur\_addr);

int inst\_shiftr(int cur\_addr);

int inst\_subr(int cur\_addr);

int inst\_svc(int cur\_addr);

int inst\_tixr(int cur\_addr);

int instruction\_f34(int cur\_addr, int extend);

int load\_word(int addr);

int load\_byte(int addr);

int inst\_add(int tar\_value);

int inst\_addf(int tar\_value);

int inst\_and(int tar\_value);

int inst\_comp(int tar\_value);

int inst\_compf(int tar\_value);

int inst\_div(int tar\_value);

int inst\_divf(int tar\_value);

int inst\_j(int tar\_value);

int inst\_jeq(int tar\_value);

int inst\_jgt(int tar\_value);

int inst\_jlt(int tar\_value);

int inst\_jsub(int tar\_value);

int inst\_lda(int tar\_value);

int inst\_ldb(int tar\_value);

int inst\_ldch(int tar\_value);

int inst\_ldf(int tar\_value);

int inst\_ldl(int tar\_value);

int inst\_lds(int tar\_value);

int inst\_ldt(int tar\_value);

int inst\_ldx(int tar\_value);

int inst\_lps(int tar\_value);

int inst\_mul(int tar\_value);

int inst\_mulf(int tar\_value);

int inst\_or(int tar\_value);

int inst\_rd(int tar\_value);

int inst\_rsub(int tar\_value);

int inst\_ssk(int tar\_value);

int inst\_sta(int tar\_value);

int inst\_stb(int tar\_value);

int inst\_stch(int tar\_value);

int inst\_stf(int tar\_value);

int inst\_sti(int tar\_value);

int inst\_stl(int tar\_value);

int inst\_sts(int tar\_value);

int inst\_stsw(int tar\_value);

int inst\_stt(int tar\_value);

int inst\_stx(int tar\_value);

int inst\_sub(int tar\_value);

int inst\_subf(int tar\_value);

int inst\_td(int tar\_value);

int inst\_tix(int tar\_value);

int inst\_wd(int tar\_value);

#endif

#include "simulator.h"

unsigned int reg[16] = { 0, };

int immediate = 1;

int cmd\_run()

{

reg[PC] = prog\_addr;

unsigned int end = 0;

do

{

int cur\_addr = reg[PC];

unsigned char format = (memory[cur\_addr] & 0xF0) / 0x10;

int flag = 0;

if (check\_bp(reg[PC]))

flag = 1;

if (0xC == format && format == 0xF)

{

reg[PC] += 1;

instruction\_f1(cur\_addr);

}

else if (0x9 <= format && format <= 0xB)

{

reg[PC] += 2;

instruction\_f2(cur\_addr);

}

else

{

int extend = 0;

if (memory[cur\_addr + 1] & 0x10)

{

reg[PC] += 4;

extend = 1;

}

else

reg[PC] += 3;

instruction\_f34(cur\_addr, extend);

}

if (flag)

{

show\_register();

printf("End program to break point.\n");

return 1;

}

}while(reg[PC] != end);

show\_register();

printf("End totally program.\n");

printf("Initialize Registers\n");

for(int i = 0 ;i<16;i++)

reg[i] = 0;

return 0;

}

void show\_register()

{

for (int i = 0; i < 16; i++)

reg[i] = reg[i] & 0xFFFFFF;

printf("A : %06X X : %06X\n", reg[A], reg[X]);

printf("L : %06X PC: %06X\n", reg[L], reg[PC]);

printf("B : %06X S : %06X\n", reg[B], reg[S]);

printf("T : %06X\n", reg[T]);

}

int instruction\_f34(int cur\_addr, int extend)

{

unsigned int opcode = memory[cur\_addr] & 0xFC;

unsigned int flag = ((memory[cur\_addr] & 0x03) << 8) | (memory[cur\_addr + 1]);

unsigned int tar\_addr = ((memory[cur\_addr + 1] & 0x0F) << 8) | memory[cur\_addr + 2];

if (extend)

tar\_addr = (tar\_addr << 8) | memory[cur\_addr + 3]; //20bit

if (flag & 0x080) //indexed mode

tar\_addr += reg[X];

if ((flag & 0x040) && !(flag & 0x020)) //base relative mode

{

if(tar\_addr & 0x800)

tar\_addr = tar\_addr | 0xFFF000;

tar\_addr += reg[B];

}

if (!(flag & 0x040) && (flag & 0x020)) //pc relative mode

{

if (tar\_addr & 0x800)

tar\_addr = tar\_addr | 0xFFF000;

tar\_addr += reg[PC];

}

if (flag & 0x200 && !(flag & 0x100)) //indirect mode if not simple mode

{

tar\_addr = tar\_addr & 0xFFFFF; //rounding;

tar\_addr = (memory[tar\_addr] << 16) | (memory[tar\_addr + 1] << 8) | memory[tar\_addr + 2];

}

immediate = 0;

if (!(flag & 0x200) && flag & 0x100)

immediate = 1;

if (opcode == 0x18)

return inst\_add(tar\_addr);

else if (opcode == 0x58)

return inst\_addf(tar\_addr);

else if (opcode == 0x40)

return inst\_and(tar\_addr);

else if (opcode == 0x28)

return inst\_comp(tar\_addr);

else if (opcode == 0x88)

return inst\_compf(tar\_addr);

else if (opcode == 0x24)

return inst\_div(tar\_addr);

else if (opcode == 0x64)

return inst\_divf(tar\_addr);

else if (opcode == 0x3C)

return inst\_j(tar\_addr);

else if (opcode == 0x30)

return inst\_jeq(tar\_addr);

else if (opcode == 0x34)

return inst\_jgt(tar\_addr);

else if (opcode == 0x38)

return inst\_jlt(tar\_addr);

else if (opcode == 0x48)

return inst\_jsub(tar\_addr);

else if (opcode == 0x00)

return inst\_lda(tar\_addr);

else if (opcode == 0x68)

return inst\_ldb(tar\_addr);

else if (opcode == 0x50)

return inst\_ldch(tar\_addr);

else if (opcode == 0x70)

return inst\_ldf(tar\_addr);

else if (opcode == 0x08)

return inst\_ldl(tar\_addr);

else if (opcode == 0x6C)

return inst\_lds(tar\_addr);

else if (opcode == 0x74)

return inst\_ldt(tar\_addr);

else if (opcode == 0x04)

return inst\_ldx(tar\_addr);

else if (opcode == 0xD0)

return inst\_lps(tar\_addr);

else if (opcode == 0x20)

return inst\_mul(tar\_addr);

else if (opcode == 0x60)

return inst\_mulf(tar\_addr);

else if (opcode == 0x44)

return inst\_or(tar\_addr);

else if (opcode == 0xD8)

return inst\_rd(tar\_addr);

else if (opcode == 0x4C)

return inst\_rsub(tar\_addr);

else if (opcode == 0xEC)

return inst\_ssk(tar\_addr);

else if (opcode == 0x0C)

return inst\_sta(tar\_addr);

else if (opcode == 0x78)

return inst\_stb(tar\_addr);

else if (opcode == 0x54)

return inst\_stch(tar\_addr);

else if (opcode == 0x80)

return inst\_stf(tar\_addr);

else if (opcode == 0xD4)

return inst\_sti(tar\_addr);

else if (opcode == 0x14)

return inst\_stl(tar\_addr);

else if (opcode == 0x7C)

return inst\_sts(tar\_addr);

else if (opcode == 0xE8)

return inst\_stsw(tar\_addr);

else if (opcode == 0x84)

return inst\_stt(tar\_addr);

else if (opcode == 0x10)

return inst\_stx(tar\_addr);

else if (opcode == 0x1C)

return inst\_sub(tar\_addr);

else if (opcode == 0x5C)

return inst\_subf(tar\_addr);

else if (opcode == 0xE0)

return inst\_td(tar\_addr);

else if (opcode == 0x2C)

return inst\_tix(tar\_addr);

else if (opcode == 0xDC)

return inst\_wd(tar\_addr);

else

{

error\_print(ERROR\_NOT\_INSTRUCT);

return 0;

}

}

int load\_word(int addr)

{

if (immediate)

return addr;

addr = addr & 0xFFFFF;

return (memory[addr] << 16) + (memory[addr + 1] << 8) + (memory[addr + 2]);

}

int load\_byte(int addr)

{

if (immediate)

return addr;

addr = addr & 0xFFFFF;

return memory[addr];

}

int inst\_add(int tar\_addr)

{

//A <- A + m(0~2)

reg[A] += load\_word(tar\_addr);

return 1;

}

int inst\_addf(int tar\_addr) //float instruction need to 48bit memory

{

//F <- F + m(0~5)

return 1;

}

int inst\_and(int tar\_addr)

{

//A <- A & m(0~2)

reg[A] = reg[A] & load\_word(tar\_addr);

return 1;

}

int inst\_comp(int tar\_addr)

{

//A : m(0~2)

unsigned int value = load\_word(tar\_addr);

if (A < value)

reg[SW] = LT;

else if (A > value)

reg[SW] = GT;

else

reg[SW] = EQ;

return 1;

}

int inst\_compf(int tar\_addr)

{

//F : m(0~5)

return 1;

}

int inst\_div(int tar\_addr)

{

//A <- A / m(0~2)

reg[A] = reg[A] / load\_word(tar\_addr);

return 1;

}

int inst\_divf(int tar\_addr)

{

//F <- F / m(0~5)

return 1;

}

int inst\_j(int tar\_addr)

{

//PC <- m

reg[PC] = tar\_addr & 0xFFFFF;

return 1;

}

int inst\_jeq(int tar\_addr)

{

//if cc set to =, PC <- m

if (reg[SW] == EQ)

reg[PC] = tar\_addr & 0xFFFFF;

return 1;

}

int inst\_jgt(int tar\_addr)

{

//if cc set to >, PC <- m

if (reg[SW] == GT)

reg[PC] = tar\_addr & 0xFFFFF;

return 1;

}

int inst\_jlt(int tar\_addr)

{

//if cc set to <, PC <- m

if (reg[SW] == LT)

reg[PC] = tar\_addr & 0xFFFFF;

return 1;

}

int inst\_jsub(int tar\_addr)

{

//L <- PC ; PC <- m

reg[L] = reg[PC];

reg[PC] = tar\_addr & 0xFFFFF;

return 1;

}

//A <- m(0~2)

int inst\_lda(int tar\_addr)

{

reg[A] = load\_word(tar\_addr);

return 1;

}

//B <- m(0~2)

int inst\_ldb(int tar\_addr)

{

reg[B] = load\_word(tar\_addr);

return 1;

}

//A[rightmost byte] <- (m)

int inst\_ldch(int tar\_addr)

{

reg[A] = load\_byte(tar\_addr);

return 1;

}

//F <- m(0~5)

int inst\_ldf(int tar\_addr)

{

return 1;

}

//L <- m(0~2)

int inst\_ldl(int tar\_addr)

{

reg[L] = load\_word(tar\_addr);

return 1;

}

//S <- m(0~2)

int inst\_lds(int tar\_addr)

{

reg[S] = load\_word(tar\_addr);

return 1;

}

//T <- m(0~2)

int inst\_ldt(int tar\_addr)

{

reg[T] = load\_word(tar\_addr);

return 1;

}

//X <- m(0~2)

int inst\_ldx(int tar\_addr)

{

reg[X] = load\_word(tar\_addr);

return 1;

}

//Load processor status from information beginning at address m

int inst\_lps(int tar\_addr)

{

return 1;

}

//A <- A \* m(0~2)

int inst\_mul(int tar\_addr)

{

reg[A] \*= load\_word(tar\_addr);

return 1;

}

//F <- F \* m(0~2)

int inst\_mulf(int tar\_addr)

{

return 1;

}

//A <- A | m(0~2)

int inst\_or(int tar\_addr)

{

reg[A] = reg[A] | load\_word(tar\_addr);

return 1;

}

//A[rightmost byte] <- data from device specified by (m)

int inst\_rd(int tar\_addr)

{

reg[SW] = LT;

return 1;

}

//PC <- L

int inst\_rsub(int tar\_addr)

{

reg[PC] = reg[L];

return 1;

}

//protection key for address m <- A

int inst\_ssk(int tar\_addr)

{

return 1;

}

//m(0~2) <- A

int inst\_sta(int tar\_addr)

{

memory[tar\_addr] = (reg[A] & 0xFF0000) / 0x10000;

memory[tar\_addr + 1] = (reg[A] & 0x00FF00) / 0x100;

memory[tar\_addr + 2] = reg[A] & 0x0000FF;

return 1;

}

//m(0~2) <- B

int inst\_stb(int tar\_addr)

{

tar\_addr = tar\_addr & 0xFFFFF;

memory[tar\_addr] = (reg[B] & 0xFF0000) / 0x10000;

memory[tar\_addr + 1] = (reg[B] & 0x00FF00) / 0x100;

memory[tar\_addr + 2] = reg[B] & 0x0000FF;

return 1;

}

//m <- A [rightmost byte]

int inst\_stch(int tar\_addr)

{

tar\_addr = tar\_addr & 0xFFFFF;

memory[tar\_addr] = reg[A] & 0x0000FF;

return 1;

}

//m(0~5) <- F

int inst\_stf(int tar\_addr)

{

return 1;

}

//Interval timer value <- m(0~2)

int inst\_sti(int tar\_addr)

{

return 1;

}

//m(0~2) <- L

int inst\_stl(int tar\_addr)

{

tar\_addr = tar\_addr & 0xFFFFF;

memory[tar\_addr] = (reg[L] & 0xFF0000) / 0x10000;

memory[tar\_addr + 1] = (reg[L] & 0x00FF00) / 0x100;

memory[tar\_addr + 2] = reg[L] & 0x0000FF;

return 1;

}

//m(0~2) <- S

int inst\_sts(int tar\_addr)

{

tar\_addr = tar\_addr & 0xFFFFF;

memory[tar\_addr] = (reg[S] & 0xFF0000) / 0x10000;

memory[tar\_addr + 1] = (reg[S] & 0x00FF00) / 0x100;

memory[tar\_addr + 2] = reg[S] & 0x0000FF;

return 1;

}

//m(0~2) <- SW

int inst\_stsw(int tar\_addr)

{

tar\_addr = tar\_addr & 0xFFFFF;

memory[tar\_addr] = (reg[SW] & 0xFF0000) / 0x10000;

memory[tar\_addr + 1] = (reg[SW] & 0x00FF00) / 0x100;

memory[tar\_addr + 2] = reg[SW] & 0x0000FF;

return 1;

}

//m(0~2) <- T

int inst\_stt(int tar\_addr)

{

tar\_addr = tar\_addr & 0xFFFFF;

memory[tar\_addr] = (reg[T] & 0xFF0000) / 0x10000;

memory[tar\_addr + 1] = (reg[T] & 0x00FF00) / 0x100;

memory[tar\_addr + 2] = reg[T] & 0x0000FF;

return 1;

}

//m(0~2) <- X

int inst\_stx(int tar\_addr)

{

tar\_addr = tar\_addr & 0xFFFFF;

memory[tar\_addr] = (reg[X] & 0xFF0000) / 0x10000;

memory[tar\_addr + 1] = (reg[X] & 0x00FF00) / 0x100;

memory[tar\_addr + 2] = reg[X] & 0x0000FF;

return 1;

}

//A <- A - m(0~2)

int inst\_sub(int tar\_addr)

{

reg[A] -= load\_word(tar\_addr);

return 1;

}

//F <- F - m(0~5)

int inst\_subf(int tar\_addr)

{

return 1;

}

//Test device specified by m

int inst\_td(int tar\_addr)

{

reg[SW] = LT;

return 1;

}

//X <- X + 1 ; X : m(0~2)

int inst\_tix(int tar\_addr)

{

unsigned int value = load\_word(tar\_addr);

reg[X]++;

if (reg[X] < value)

reg[SW] = LT;

else if (reg[X] > value)

reg[SW] = GT;

else

reg[SW] = EQ;

return 1;

}

//Device specified by m <- A[rightmost byte]

int inst\_wd(int tar\_addr)

{

return 1;

}

int instruction\_f2(int cur\_addr)

{

if (memory[cur\_addr] == 0x90)

return inst\_addr(cur\_addr+1);

else if (memory[cur\_addr] == 0xB4)

return inst\_clear(cur\_addr + 1);

else if (memory[cur\_addr] == 0xA0)

return inst\_compr(cur\_addr + 1);

else if (memory[cur\_addr] == 0x9C)

return inst\_divr(cur\_addr + 1);

else if (memory[cur\_addr] == 0x98)

return inst\_mulr(cur\_addr + 1);

else if (memory[cur\_addr] == 0xAC)

return inst\_rmo(cur\_addr + 1);

else if (memory[cur\_addr] == 0xA4)

return inst\_shiftl(cur\_addr + 1);

else if (memory[cur\_addr] == 0xA8)

return inst\_shiftr(cur\_addr + 1);

else if (memory[cur\_addr] == 0x94)

return inst\_subr(cur\_addr + 1);

else if (memory[cur\_addr] == 0xB0)

return inst\_svc(cur\_addr + 1);

else if (memory[cur\_addr] == 0xB8)

return inst\_tixr(cur\_addr + 1);

else

{

error\_print(ERROR\_NOT\_INSTRUCT);

return 0;

}

}

int inst\_addr(int cur\_addr)

{

//r2 <- (r2) + (r1)

int r1 = memory[cur\_addr] / 0x10;

int r2 = memory[cur\_addr] % 0x10;

reg[r2] = reg[r2] + reg[r1];

return 1;

}

int inst\_clear(int cur\_addr)

{

//r1 <- 0

int r1 = memory[cur\_addr] / 0x10;

int r2 = memory[cur\_addr] % 0x10;

if (r2 != 0) //instruction error

;

reg[r1] = 0;

return 1;

}

int inst\_compr(int cur\_addr)

{

//(r1) : (r2)

int r1 = memory[cur\_addr] / 0x10;

int r2 = memory[cur\_addr] % 0x10;

if (reg[r1] < reg[r2])

reg[SW] = LT;

else if (reg[r1] > reg[r2])

reg[SW] = GT;

else

reg[SW] = EQ;

return 1;

}

int inst\_divr(int cur\_addr)

{

//r2 <- r2 / r1

int r1 = memory[cur\_addr] / 0x10;

int r2 = memory[cur\_addr] % 0x10;

reg[r2] = reg[r2] / reg[r1];

return 1;

}

int inst\_mulr(int cur\_addr)

{

//r2 <- r2 \* r1

int r1 = memory[cur\_addr] / 0x10;

int r2 = memory[cur\_addr] % 0x10;

reg[r2] = reg[r2] \* reg[r1];

return 1;

}

int inst\_rmo(int cur\_addr)

{

//r2 <- r1

int r1 = memory[cur\_addr] / 0x10;

int r2 = memory[cur\_addr] % 0x10;

reg[r2] = reg[r1];

return 1;

}

int inst\_shiftl(int cur\_addr)

{

//r1 <- (r1) << left circular shift n bits

int r1 = memory[cur\_addr] / 0x10;

int r2 = memory[cur\_addr] % 0x10;

r2++;

reg[r1] = (reg[r1] << r2) | (reg[r1] >> (32 - r2));

return 1;

}

int inst\_shiftr(int cur\_addr)

{

//r1 <- (r1) >> right arithmetic shift n bits

int r1 = memory[cur\_addr] / 0x10;

int r2 = memory[cur\_addr] % 0x10;

r2++;

int sign = 0;

if (reg[r1] & 0x800000)

sign = 1;

reg[r1] = (reg[r1] >> r2) | ((((0x1 << r2) - 1) << (32 - r2)) \* sign);

return 1;

}

int inst\_subr(int cur\_addr)

{

//r2 <- r2 - r1

int r1 = memory[cur\_addr] / 0x10;

int r2 = memory[cur\_addr] % 0x10;

reg[r2] = reg[r2] - reg[r1];

return 1;

}

int inst\_svc(int cur\_addr)

{

//generate svc interrupt

return 1;

}

//X <- (X) + 1; (X) : (r1)

int inst\_tixr(int cur\_addr)

{

int r1 = memory[cur\_addr] / 0x10;

reg[X]++;

if (reg[X] < reg[r1])

reg[SW] = LT;

else if (reg[X] > reg[r1])

reg[SW] = GT;

else

reg[SW] = EQ;

return 1;

}

int instruction\_f1(int cur\_addr)

{

if (memory[cur\_addr] == 0xC4)

return inst\_fix();

else if (memory[cur\_addr] == 0xC0)

return inst\_float();

else if (memory[cur\_addr] == 0xF4)

return inst\_hio();

else if (memory[cur\_addr] == 0xC8)

return inst\_norm();

else if (memory[cur\_addr] == 0xF0)

return inst\_sio();

else if (memory[cur\_addr] == 0xF8)

return inst\_tio();

else

{

error\_print(ERROR\_NOT\_INSTRUCT);

return 0;

}

}

int inst\_fix()

{

//A <- F(convert to integer)

return 1;

}

int inst\_float()

{

//F <- A(convert to float)

return 1;

}

int inst\_hio()

{

//Halt I/O channel number(A)

return 1;

}

int inst\_norm()

{

//F <- (F) [nomalized]

return 1;

}

int inst\_sio()

{

//start I/O channel number(A)

//address of channel program is given by (S)

return 1;

}

int inst\_tio()

{

//Test I/O channel number (A)

return 1;

}