**과목명: 시스템프로그래밍**

**분반2**

**<<Project #1>>**

**컴퓨터공학과 [학부명]**

**20161580[학번]**

**김현규[이름]**

목 차

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
3. **모듈 정의**
   1. 정의한 모듈에 대해 간략적인 설명
4. **전역 변수 정의**
5. **코드 설명**
6. 프로그램 개요

|  |
| --- |
| 이 프로그램은 앞으로 구현하게 될 SIC/XE 머신을 구현하기 위한 전 단계로서 어셈블러, 링크, 로더들을 실행하게 될 셸(shell)과 컴파일을 통해서 만들어진 object 코드가 적재 되고 실행될 메모리공간과 mnemonic (ADD, COMP, FLOAT, etc …)을 opcode 값으로 변환하 는 OPCODE 테이블과 관련 명령어들을 구현하는 프로그램이다. |

1. 프로그램 설명

|  |
| --- |
| -이 프로그램을 실행시키면 아래와 같이 unix shell 과 유사한 입력 프롬프트상태가 된다. sicsim>  -이 상태에서 아래에 있는 명령어들을 입력할 때, 그에 해당되는 기능을 수행하여야 한다. -구현해야 할 사항들  ① 셸 ( sicsim> )  ② 셸 관련 명령어들 (help, dir, quit, history)  ③ 메모리공간 (1MB의 메모리를 할당해서 사용)  ④ 메모리공간 관련 명령어들 (dump, edit, fill, reset)  ⑤ opcode 테이블 (HashTable 로 만들어야 함) ⑥ opcode 관련 명령어들 (opcode, opcodelist) |

2-1. 프로그램 흐름도

|  |
| --- |
| 1. Int main (void)      1. Input\_Order      1. Activate |

1. 모듈 정의

3.1 모듈 이름: int main(void)

3.1.1 기능

-Input\_Data() 함수를 호출한다. stdin에서 명령어를 입력받는다. 명령어를 Input\_Order에 넘겨준다. 또한 명령어 실행 후 그것이 ERROR가 없는 적절한 명령어라 판단될 경우 이를 history가 다루는 링크드리스트(ActicatedCommands\_head or ActivatedCommands\_tail)에 저장한다. ERROR로 판단될 경우 마땅한 ERROR MESSAGE를 출력한다.

-char\* command: stdin에서 입력받는 명령어이고, int flag: 실행한 명령어가 ERROR인지 아닌지의 여부이다.

3.2 모듈 이름: int Input\_Data(void)

3.2.1 기능

-unsigned char\*\* memory, hashtable을 초기화한다.

-opcode.txt를 읽어 이를 hashtable에 저장한다.

-FILE\* ip: file pointer

-hash\* temphash: opcode.txt를 hashtable로 넘겨주는 데에 쓰이는 임시 공간

-int THISopcode: temphash의 opcode의 10진수

-int THISkey: temphash가 hashtable에서 저장될 위치

3.3 모듈 이름: int key(int opcode)

3.3.1 기능

-주어진 parameter을 바탕으로 일정한 기준에 따라 key를 계산한다.

3.4 모듈 이름: int Input\_Order(char\* order)

3.4.1 기능

-char\* order을 프로그램에서 쓸 수 있도록 바꾼다. int par1에 첫 번째 parameter, int par2에 두 번째 parameter, int par3에 세 번째 parameter, char\* whattodo에 명령어, int parmetercount에 총 parmeter의 개수, char\* opcode에 opcode(이는 ‘opcode’를 실행할 때 사용한다.)를 저장한다.

-이들을 parameter로 하는 함수 Activate를 실행한다. Activate 함수는 넘겨받는 parameter들로 error임을 판단할 경우 이에 맞는 error를 return해준다. 해당 모듈 역시 return받은 error의 정보를 main에 return한다.

-char \*\*num: order의 paramete(숫자)들이 저장

-char\* opcode: opcode(char\* whattodo가 ‘opcode’일 경우 쓰인다.) 저장

-char\* whattodo: order의 명령어 저장

3.5 모듈 이름: int Activate(char\* whattodo, int par1, int par2, int par3, int parametercount, char\* string)

3.5.1 기능

-char\* whattodo에 저장되어 있는 명령어에 해당하는 함수를 실행한다.

-dump를 실행할 경우 parametercount에 따라 그것이 어떻게 실행되는지가 다르다. 0일 경우 start = dump\_start(dump를 실행했을 경우 시작 주소), end = start + line\_size \* 10 - 1이다. 그리고 바뀐 dump\_start을 저장해줘야 한다. 1일 경우 start와 end는 0과 같다. 2일 경우 start는 par1, end는 par2이다.

-위 내용은,

*“dump*

*기본적으로 10 라인이 출력된다. (한 라인은 메모리의 16 개 바이트로 구성)*

*dump의 실행으로 출력된 마지막 address는 내부에 저장하고 있다.*

*다시 dump를 실행시키면 마지막 (address + 1) 번지부터 출력된다.*

*dump 명령어가 처음 시작될 때는 0 번지부터 출력된다*

*계속된 dump 출력 시 boundary check를 하여 주소의 끝(0xFFFFF)까지*

*- dump start*

*start 번지부터 10 라인(160 개)을 출력.*

*주소를 넘어간 경우 주소의 끝 (0xFFFFF)까지 출력.*

*Start의 주소값이 메모리 범위를 벗어나는 경우 적당한 에러 메시지를*

*출력합니다.*

*- dump start, end*

*start 부터 end 번지까지의 내용을 출력.*

*Start 주소가 end 주소보다 작은 값이 들어온 경우, 에러 처리.*

*Star, end의 주소값이 메모리 범위를 벗어나는 경우 적당한 에러 메시지를*

*출력합니다.”*

을 고려한 내용이다.

-edit을 실행할 경우 par1은 address, par2는 value이다.

-fill을 실행할 경우 par1은 start, par2는 end, par3는 value이다.

-opcode를 실행할 경우 찾고자 하는 opcode는 char\* string이다.

-ERROR가 생길 경우 Input\_Order 모듈에 해당 ERROR의 정보를 return한다.

3.6 모듈 이름: void help()

3.6.1 기능

-Shell에서 실행 가능한 모든 명령어들의 리스트를 화면에 출력한다.

3.7 모듈 이름: int dir()

3.7.1 기능

-현재 디렉토리에 존재하는 모든 파일과 디렉토리를 출력한다.

-실행 파일은 파일 이름 옆에 \*표시를, 디렉토리는 /표시를 해준다.

-S\_ISDIR(buf.st\_mode)가 1일 경우 주어진 parameter struct dirent\* dir\_entry가 디렉토리임을 뜻한다.

-S\_ISREG(buf.st\_mode)가 1일 경우 주어진 parameter struct dirent\* dir\_entry가 파일임을 뜻한다.

3.8 모듈 이름: void history()

3.8.1 기능

-현재까지 사용한 명령어들을 순서대로 번호(int index)와 함께 보여준다.

-node\* ActivatedCommands\_head를 이용한다.

3.9 모듈 이름: int dump(int start, int end)

3.9.1 기능

-가장 왼쪽 column은 출력하는 메모리의 주소, 가운데 column은 메모리 내용을 16진수로, 가장 오른쪽 column은 메모리 내용을 byte 별로 대응하는 ASCII code 형태로 보여준다.

-(int y\_line\_start, int x\_line\_start)와 (int y\_line\_end, int x\_line\_end)는 각각 start와 end의 2차원 배열 memory의 좌표다. 이를 활용하여 좀더 쉽게 코드를 작성할 수 있다.

3.10 모듈 이름: int edit(int address, int value)

3.10.1 기능

-메모리의 address번지의 값을 value에 지정된 값으로 변경한다.

-(int y, int x)는 address의 좌표다.

3.11 모듈 이름: int fill(int start, int end, int value)

3.11.1 기능

-메모리의 start번지부터 end번지까지의 값을 value에 지정된 값으로 변경한다.

-start번지부터 end번지까지에 edit 함수를 실행한다.

3.12 모듈 이름: int reset(void)

3.12.1 기능

-메모리 전체를 전부 0으로 초기화한다.

3.13 모듈 이름: void opcodelist(void)

3.13.1 기능

-hash\* hashtable[tablesize]에 저장되어있는 값들을 출력한다.

3.14 모듈 이름: int opcode(char\* Mnemonic)

3.14.1 기능

-char\* Mnemonic의 opcode를 출력한다.

-hash\* temphash는 hash\* hashtable[tablesize]를 임시로 지정하는 포인터 변수이다.

-올바른 opcode를 입력하지 않았을 경우 ERROR를 return한다.

3.15 모듈 이름: void AddCommand(char\* command)

3.15.1 기능

-main 모듈에서 실행한 명령어를 node\* ActivatedCommand\_head 혹은 ActivatedCommand\_tail에 저장한다.

3.16 모듈 이름: void AddHash(int key, char\* sym, char\* order, char\* format)

3.16.1 기능

-Input\_Data 모듈에서 Hashtable을 만들 때 사용되는 함수다. hashtable[key]에 해당 hash\*를 저장한다. 해당 hash\*에는 char\* sym, char\* order, char\* format을 저장한다.

1. 전역 변수 정의

4.1 node\* ActivatedCommands\_head, node\* ActivatedCommands\_tail

4.1.1 기능

-그동안 실행되었던 명령어들을 저장하는 링크드리스트의 head와 tail이다.

4.2 hash\* hashtable[tablesize] (tablesize: 20)

4.2.1 기능

-opcode.txt를 저장하는 해시 테이블이다.

4.3 int dump\_start

4.3.1 기능

-parameter가 0개인 dump를 실행할 때 시작 주소이다.

4.4 unsigned char\*\* mem

4.4.1 기능

-1MB 크기의 가상 메모리를 할당하는 2차원 배열이다.

4.5 int minus\_flag

4.5.1 기능

-parameter로 음수가 들어왔을 경우를 구분해주는 flag이다.

1. 코드 설명

5-1. 20161580.h

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h> //셸 명령어 dir에서 쓰이는 library

#include <sys/stat.h>//셸 명령어 dir에서 쓰이는 library

#define line\_size 16

#define line\_max 65536

#define tablesize 20

#define max\_command\_size 50

//ERROR 종류

#define ERROR\_OutofRange -1

#define ERROR\_StartIsLargerThanEnd -2

#define ERROR\_WrongCommand -3

#define ERROR\_WrongOpcode -4

//셸 명령어 history에 쓰이는 자료구조

typedef struct \_node{

char name[30];

struct \_node \*next;

}node;

node\* ActivatedCommands\_head;

node\* ActivatedCommands\_tail;

typedef struct \_hash {

char sym[10];

char order[6];

char format[6];

struct \_hash \*next;

}hash; //hashtable을 위한 구조체

hash\* hashtable[tablesize];

void AddHash(int key, char\* sym, char\* order, char\* format);

unsigned char\*\* mem;

void help();

int dir();

void history();

int dump(int start, int end);

int dump\_start = 0; //dump의 시작 주소

int edit(int address, int value);

int fill(int start, int end, int value);

int reset(void);

int opcode(char\* Mnemonic);

void opcodelist(void);

int key(int opcode);

int Input\_Data(void);

int Input\_Order(char\* order);

int Activate(char\* whattodo, int par1, int par2, int par3, int parametercount, char \*opcode);

void AddCommand(char\* command);

int minus\_flag = 0; //입력받은 parameter에 음수가 존재할 경우 1이 됨

5-2. 20161580.c

#include"20161580.h"

void help() { //Shell에서 실행 가능한 모든 명령어들의 리스트를 화면에 출력한다.

printf("\nh[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\n");

}

int dir(){//현재 Directory에 있는 파일들을 출력한다.

DIR\* dp = NULL;

struct dirent \*dir\_entry = NULL;

struct stat buf;

//현재 directory를 여는데 실패했을 경우

//"."은 현재 디렉토리를 지칭한다.

if((dp = opendir(".")) == NULL){

printf("Can't open present directory\n");

return -1;

}

//현재 directory에 있는 모든 파일과 directory를 출력한다.

while((dir\_entry = readdir(dp))!= NULL){

//주어진 parameter(dir\_entry)가 파일인지 Directory인지 판단한다.

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

//Dir의 결과를 출력할 때 실행 파일은 파일 이름 옆에 "\*"표시를 해야한다.

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

{

dir\_entry->d\_name[strlen(dir\_entry->d\_name)] = '\0';

if(!strcmp(".out", dir\_entry->d\_name + strlen(dir\_entry->d\_name) - 4)) printf("%s\*\t", dir\_entry->d\_name);

else printf("%s\t", dir\_entry->d\_name);

}

//Directory는 '/'표시를 해야한다.

else if(S\_ISDIR(buf.st\_mode))printf("%s/\t", dir\_entry->d\_name);

}

//이용한 DIR(현재 디렉토리)을 닫아준다.

closedir(dp);

printf("\n");

return 0;

}

void history(){//현재까지 사용한 명령어들을 순서대로 번호(index)와 함께 보여준다.

int index = 1;

node\* his = ActivatedCommands\_head;

while(his != NULL)

{

printf("%d %s\n", index++, his->name);

his = his->next;

}

free(his);

}

int Input\_Order(char\* order){//입력받은 명령어의 정보를 상황에 맞게 구분한다.

//char\* whattodo는 명령어를 저장하는 array이다.

//char\* num[0]은 명령에 쓰이는 1번째 parameter,

//char\* num[1]은 2번째 parameter이다. 이들은 우선 16진수로 저장한다.

//ex)1번째 parameter가 6A(16)일 경우, num[0][0] = '6', num[0][1] = 'A'로 저장한다.

char \*\*num = malloc(3\*sizeof(char\*));

for(int n=0; n<3; ++n){

num[n] = (char\*)malloc(10\*sizeof(char));

}

char \*opcode = malloc(10\*sizeof(char));

char \*whattodo = malloc(10\*sizeof(char));

int i; int order\_len = strlen(order); int NN = 0, nn = 0, wtd = 0, str = 0;

int IsParameterNum = 0; int IsParameterString = 0;

for(i=0; i<order\_len; ++i){

if(IsParameterNum){

if(order[i] == '-') minus\_flag = 1;

//parameter가 소문자로 입력되었을 때, 대문자로 바꿔준다.

//따라서 프로그램은 16진수가 소문자이든 대문자이든간에 인식할 수 있다.

if(order[i] >= 'a' && order[i] <= 'f') order[i] = 'A' + order[i] - 'a';

if((order[i] >= '0' && order[i] <= '9') || (order[i] >= 'A' && order[i] <= 'F')){ //parameter가 숫자일 경우

num[NN][nn++] = order[i];

}

else if(order[i] == ',') {//NN번째 parameter 정보를 저장했고, NN+1번째 parameter 정보를 저장해야 할 경우

NN++; nn=0;

}

else if(order[i] == ' ') continue;

}

else if(IsParameterString){//parameter가 string일 경우

opcode[str++] = order[i];

}

else{

if(order[i] >= 'a' && order[i] <= 'z'){ //입력받는 값이 소문자 알파벳일 경우

//즉, 명령어일 경우

whattodo[wtd++] = order[i];

}

if(order[i] == ' ') {

if(!strcmp(whattodo, "opcode")) IsParameterString = 1; //명령어 정보를 다 담아낸 후, paramter가 숫자인지 string인지 알아낸다

else IsParameterNum = 1;

}

}

}

//char\* num을 strtol 함수를 이용하여 10진수로 변환한다.

//예를 들어, (16진수)37 -> (10진수)55

int par1 = strtol(num[0], NULL, 16);

int par2 = strtol(num[1], NULL, 16);

int par3 = strtol(num[2], NULL, 16);

//whattodo의 마지막에 NULL값을 저장한다.

whattodo[wtd] = '\0'; opcode[str] = '\0';

int parametercount = NN + 1;

//Parameter가 string일 경우 parametercount는 0이다.

if(!IsParameterNum) parametercount = 0;

//flag에는 실행한 명령어에 ERROR가 있을 경우 해당하는 ERROR을 return받는 변수다.

//명령어에 ERROR이 없을 경우 아무 값도 return받지 않는다.

int flag = Activate(whattodo, par1, par2, par3, parametercount, opcode);

if(flag == ERROR\_WrongCommand) return flag;

else if(flag == ERROR\_StartIsLargerThanEnd) return flag;

else if(flag == ERROR\_OutofRange) return flag;

else if(flag == ERROR\_WrongOpcode) return flag;

}

int Activate(char\* whattodo, int par1, int par2, int par3, int parametercount, char \*string)

{

//char\* whattodo의 정보에 해당하는 함수를 실행한다.

int start, end, value;

int address;

if(!strcmp(whattodo, "h") || !strcmp(whattodo, "help")){help();}

else if(!strcmp(whattodo, "d") || !strcmp(whattodo, "dir")){dir();}

else if(!strcmp(whattodo, "hi") || !strcmp(whattodo, "history")){

//프로그램의 특성상 미리 명령어를 저장한다.

AddCommand(whattodo);

history();

}

else if(!strcmp(whattodo, "du") || !strcmp(whattodo, "dump"))

{

if(parametercount == 0) //parameter의 개수가 0일 경우. ex) dump

{

start = dump\_start; end = start + line\_size \* 10 - 1;

dump\_start = end + 1;

}

else if(parametercount == 1) //parameter의 개수가 1일 경우. ex) dump 10

{

start = par1; end = par1 + line\_size \* 10 - 1;

}

else //parameter의 개수가 2일 경우. ex)dump 10, 2A

{

start = par1; end = par2;

//parameter가 2개일 때, end가 memory 범위를 벗어날 경우 return ERROR\_OutofRange

if(end >= line\_size \* line\_max) return ERROR\_OutofRange;

}

//parameter가 0개 혹은 1개일 때, end가 memory 영역 밖으로 벗어날 경우

if(end >= line\_size \* line\_max){

//end는 memory 끝 값으로 바꿔준다.

end = line\_size \* line\_max - 1;

//parametercount가 0일 경우 dump를 실행할 때 처음부터 실행되어야 한다.

//따라서 dump\_start를 0으로 초기화한다.

if(parametercount == 0) dump\_start = 0;

}

//start 또는 end가 음수일 경우 minus\_flag = 1

if(minus\_flag){

minus\_flag = 0;

return ERROR\_OutofRange;

}

//start가 memory 영역 끝 값보다 클 경우

if(start >= line\_size \* line\_max) return ERROR\_OutofRange;

else {

//start가 end보다 클 경우

if(start > end) return ERROR\_StartIsLargerThanEnd;

else dump(start, end);

}

}

else if((!strcmp(whattodo, "e") || !strcmp(whattodo, "edit")) && parametercount == 2)

{

address = par1; value = par2;

//입력받은 parameter에 음수가 있을 경우

if(minus\_flag){

minus\_flag = 0;

return ERROR\_OutofRange;

}

else if(value >= 32 && value <= 126) edit(address, value);

//value가 영역에 해당하지 않을 경우 ERROR

else return ERROR\_OutofRange;

}

else if((!strcmp(whattodo, "f") || !strcmp(whattodo, "fill")) && parametercount == 3)

{

start = par1; end = par2; value = par3;

//입력받은 parameter에 음수가 있을 경우

if(minus\_flag){

minus\_flag = 0;

return ERROR\_OutofRange;

}

//start가 end보다 클 경우

else if(start > end) return ERROR\_StartIsLargerThanEnd;

//start나 end가 메모리 영역을 양의 방향으로 벗어날 경우

else if(start >= line\_size \* line\_max || end >= line\_size \* line\_max) return ERROR\_OutofRange;

//value가 영역에 해당하지 않을 경우

else if(value < 32 || value > 126) return ERROR\_OutofRange;

else fill(start, end, value);

}

else if(!strcmp(whattodo, "reset")){reset();}

else if(!strcmp(whattodo, "opcode")){

//opcode.txt에 존재하지 않는 opcode가 넘겨졌을 때

if(opcode(string) == ERROR\_WrongOpcode){return ERROR\_WrongOpcode;}}

else if(!strcmp(whattodo, "opcodelist")){opcodelist();}

else return ERROR\_WrongCommand;

}

int dump(int start, int end)

{

//start번지부터 end번지까지의 정보를 출력한다.

//첫번째 column은 출력하는 메모리의 주소, 두번째 column은 메모리 내용을 16진수로 보여준다.

//세번째 column은 메모리 내용을 byte 별로 대응하는 ASCII code 형태로 보여준다.

int y\_line\_start = start / line\_size; //시작점의 y좌표

int y\_line\_end = end / line\_size; //종점의 y좌표

int x\_line\_start = start % line\_size; //시작점의 x좌표

int x\_line\_end = end % line\_size; //종점의 x좌표

int x, y;

for(y = y\_line\_start; y <= y\_line\_end; ++y){

//첫번째 column

printf("%04X0 ", y);

//두번째 column

for(x = 0; x < line\_size; ++x){

if(y == y\_line\_start && x < x\_line\_start) printf(" ");

else if(y == y\_line\_end && x > x\_line\_end) printf(" ");

else printf("%02X ", (unsigned char)mem[y][x]);

}

printf("; ");

//세번째 column

for(x = 0; x < line\_size; ++x) {

if(y == y\_line\_start && x < x\_line\_start) printf(".");

else if(y == y\_line\_end && x > x\_line\_end) printf(".");

else{

//value 영역에 속할 경우

if(mem[y][x] >= 32 && mem[y][x] <= 126){

printf("%c", mem[y][x]);

}

//안 속할 경우

else printf(".");

}

}

printf("\n");

}

}

int edit(int address, int value)

{

//함수의 parameter은 10진수이다.

//메모리의 address번지 값을 value로 변경한다.

//address의 위치를 2차원 배열 char\*\* mem에 맞게 변환한다.

int y = address / line\_size;

int x = address % line\_size;

mem[y][x] = value;

}

int fill(int start, int end, int value){

//start부터 end까지 edit한다.

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

edit(address, value);

}

int reset(void){

//메모리의 전부를 0으로 초기화한다.

for(int y=0; y<line\_max; ++y)

memset(mem[y], 0, sizeof(char)\*line\_size);

}

void AddCommand(char\* command)

//실행한 명령어를 ActivatedCommands\_head 혹은 ActivatedCommands\_tail에 넣는다.

{

node\* newNode = (node\*)malloc(sizeof(node));

strcpy(newNode->name, command);

newNode->next = NULL;

if(ActivatedCommands\_head == NULL){

ActivatedCommands\_head = newNode;

ActivatedCommands\_tail = ActivatedCommands\_head;

}

else{

ActivatedCommands\_tail->next = newNode;

ActivatedCommands\_tail = newNode;

}

}

void AddHash(int key, char\* sym, char\* order, char\* format)

{

//hashtable[key]에 char\* order, char\* format을 저장한다.

hash\* newHash = (hash\*)malloc(sizeof(hash));

newHash->next = hashtable[key]->next;

strcpy(newHash->sym, sym);

strcpy(newHash->order, order);

strcpy(newHash->format, format);

hashtable[key]->next = newHash;

}

void opcodelist(void)

{

//opcode.txt에서 받아온 opcode의 정보들을 나열한다.

hash\* temphash = (hash\*)malloc(sizeof(hash));

for (int h = 0; h < tablesize; ++h) {

temphash = hashtable[h]->next;

printf("%d : ", h);

while (temphash != NULL) {

if(temphash->next!=NULL) printf("[%s,%s] ->", temphash->order, temphash->sym);

else printf("[%s,%s]", temphash->order, temphash->sym);

temphash = temphash->next;

}

printf("\n");

}

}

int opcode(char\* Mnemonic)

{

//opcode의 정보가 저장되어 있는 hashtable에서 char\* Mnemonic을 검색한다.

int Find = 0;

hash\* temphash = (hash\*)malloc(sizeof(hash));

//Mnemonic의 예로 Add가 있다.

for (int h = 0; h < tablesize; ++h) {

temphash = hashtable[h];

while(temphash != NULL){

//Mnemonic을 찾았을 경우

if (!strcmp(temphash->order, Mnemonic))

{

Find = 1;

printf("opcode is %s\n", temphash->sym);

}

if (Find) break;

temphash = temphash->next;

}

if (Find) break;

}

//해당하는 opcode를 찾지 못했을 경우 Find = 0으로 유지된다. 따라서 ERROR을 return한다.

if(!Find) return ERROR\_WrongOpcode;

}

int key(int opcode)

{

int key = opcode % tablesize;

//key를 return한다.

return key;

}

int Input\_Data(void)

{

FILE\* ip;

if (!(ip = fopen("./system1.c", "rb"))) {

fputs("ERROR: File NOT OPEN\n", stderr);

return 0;

}

//Memory 초기화

int i;

//1MB를 만들기 위해 mem의 data type을 1byte인 unsigned char로 하였다.

mem = (unsigned char\*\*)malloc(sizeof(unsigned char\*)\*line\_max);

for(i=0; i<line\_max; ++i){

mem[i] = (unsigned char\*)malloc(sizeof(unsigned char) \* line\_size);

}

for(i=0; i<line\_max; ++i){

fread(mem[i], sizeof(unsigned char), line\_size, ip);

}

fclose(ip);

//opcode.txt를 기반으로 Hash table 초기화

int h;

for(h = 0; h < tablesize; ++h){

hashtable[h] = malloc(sizeof(hash));

hashtable[h]->next = NULL;

}

if (!(ip = fopen("./opcode.txt", "r"))) {

fputs("ERROR: opcode.txt NOT OPEN\n", stderr);

return 0;

}

hash\* temphash = malloc(sizeof(hash));

while(1){

//opcode.txt의 한 line을 읽는다.

fscanf(ip, "%s %s %s", temphash->sym, temphash->order, temphash->format);

if(feof(ip) != 0) break;

//hash table에 값을 저장한다. 방법은 다음과 같다.

//1. sym에 알맞는 key를 배정한다.

int THISopcode = (int)strtol(temphash->sym, NULL, 16);

int THISkey = key(THISopcode);

//2. key에 해당하는 linked list에 값을 저장한다.

//저장해야 하는 값은 1. symbol 2. order 3. format

AddHash(THISkey, temphash->sym, temphash->order, temphash->format);

}

}

int main (void)

{

Input\_Data();

char\* command = malloc(sizeof(char) \* max\_command\_size);

while(1){

printf("sicsim>");

fgets(command, max\_command\_size, stdin);

//fgets로 string을 입력할 경우 string 마지막에 enter, 즉 '\n' 값이 저장된다.

//따라서 string의 마지막을 NULL로 저장해준다.

command[strlen(command) - 1] = '\0';

//q를 입력할 경우 프로그램 종료

if(!strcmp("q", command) || !strcmp("quit", command)) break;

else {

//입력받은 명령어(command)를 실행한다.

int flag = Input\_Order(command);

//history를 실행할 때 history 역시 링크드리스트에 저장이 되어야 한다.

//따라서 다른 명령어들과 달리 history는 main에서 AddCommand를 안한다.

//대신 Activate 함수에서 AddCommand를 한다.

if(!strcmp("hi", command) || !strcmp("history", command)){}

//입력한 명령어에 ERROR가 있을 경우 해당 ERROR의 정보를 출력한다.

else if(flag == ERROR\_OutofRange) printf("ERROR: Out of Range\n");

else if(flag == ERROR\_StartIsLargerThanEnd) printf("ERROR: Start is larger than End\n");

else if(flag == ERROR\_WrongCommand) printf("ERROR: Wrong Command\n");

else if(flag == ERROR\_WrongOpcode) printf("ERROR: Wrong Opcode\n");

//입력한 명령어가 ERROR가 없을 경우 해당 명령어를 저장한다.

else AddCommand(command);

}

}

}