과목 명: 시스템프로그래밍

담당 교수 명: 박 운 상

<<Assignment 3>>

**서강대학교 컴퓨터학과**

**[20141543]**

**[안승일]**

목 차

1. 프로그램 개요 3

2. 프로그램 설명 3

2.1 프로그램 흐름도 3

3. 모듈 정의 3

3.1 모듈 이름 : pass1\_loader() 3

3.1.1 기능 3

3.1.2 사용 변수 3

3.2 모듈 이름: pass2\_loader() 3

3.2.1 기능 3

3.2.2 사용 변수 3

3.3 모듈이름: make\_EXSYM() 4

3.3.1 기능 4

3.3.2 사용변수 4

3.4 모듈이름: check\_EXSYM() 4

3.4.1 기능 4

3.4.2 사용변수 4

3.5 모듈이름: print\_Load\_map() 4

3.5.1 기능 4

3.5.2 사용변수

3.6 모듈이름: print\_bp()

3.6.1 기능

3.6.2 사용변수

3.7 모듈이름: make\_bplist()

3.7.1 기능

3.7.2 사용변수

3.8 모듈이름: Run()

3.8.1 기능

3.8.2 사용변수

3.9 모듈이름: free\_EXSYM()

3.9.1 기능

3.9.2 사용변수

3.10 모듈이름: free\_ bp()

3.10.1 기능

3.10.2 사용변수

4. 전역 변수 정의 4

4.1 symbol\_table \*bp\_list 4

4.2 exsym\_table \*exsym[MAX\_HASH] 4

4.3 int progaddr, run\_first\_try 4

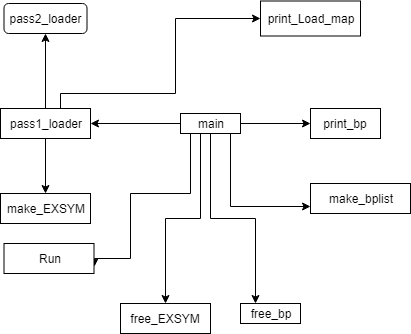
4.4 int run\_start\_addr, run\_end\_addr, run\_last\_addr 5

# 프로그램 개요

프로젝트 #1, #2 에서 구현한 셀(shell)에 linking과 loading 기능을 추가하는 프로그램이다. 프로젝트 #2 에서 구현된 assemble 명령을 통해서 생성된 object 파일을 link시켜 메모리에 올리는 일을 수행합니다.

# 프로그램 설명

## 프로그램 흐름도



# 모듈 정의

## 모듈 이름 : pass1\_loader()

### 기능

Load 기능을 위한 pass1을 수행한다. Estab을 만드는 역할이 주된역할이다.

### 사용 변수

FILE \*fp, \*fp1, \*fp2, \*fp3 : file open을 위한 변수이다.

int objfile\_cnt : object file이 몇개 들어왔는지 저장할 변수이다.

int cslth, csaddr : control section 의 길이, 주소를 저장할 변수이다.

char objfile[3][10] : object file을 저장할 배열이다.

char name\_tmp[10], addr\_tmp[10], len\_tmp[10] : 각각 object file의 이름, 주소, 길이를 저장할 배열이다.

## 모듈 이름: pass2\_loader()

### 기능

loader의 pass2기능을 수행한다. R record, M record의 처리를 한다. relocation이 주된 업무이다.

### 사용 변수

int csaddr, cslth : control section의 주소, 길이를 저장할 변수이다

int reg\_num, ref\_len, txt\_len : reference 의 숫자, 길이, text record의 길이를 저장할 변수이다.

int ref\_arr[100] : R record를 저장할 배열이다.

int mo\_type : M record에서 05, 06을 저장할 변수이다.

char mo\_cal : M record에서 +인지 –인지를 저장할 변수이다.

## 모듈이름: make\_EXSYM()

### 기능

external symbol table을 만든다.

### 사용변수

int key : hash function인 external symbol table의 인덱스를 얻기 위한 변수이다.

## 모듈이름: check\_EXSYM()

### 기능

external symbol table에 이미 있는지 없는지 검사하는 함수이다. 있다면 주소를 반환한다.

### 사용변수

int key : hash function의 인덱스를 알기 위해 저장하는 변수이다.

## 모듈이름: print\_Load\_map()

### 기능

relocation 후 control section의 주소, definition record의 주소를 출력하는 함수이다.

### 사용변수

exsym\_table \*tmp, sorted[300], swap : for문을 위한 counter와 swap을 위한 변수이다.

## 모듈이름: print\_bp()

### 기능

break point의 linked list를 출력하는 함수이다.

### 사용변수

symbol\_table \*ptr : break point linked list를 출력하기 위한 for문의 counter이다.

## 모듈이름: make\_bplist()

### 기능

break point linked list를 만들기 위한 함수이다.

### 사용변수

symbol\_table \*ptr, \*new : for문을 위한 counter와 새로운 노드를 위한 변수이다.

## Run()

### 기능

memory에 올라간 값들을 바탕으로 레지스터의 값들을 지정해주는 함수이다.

### 사용변수

없음

## free\_EXSYM()

### 기능

external symbol table을 free해주는 함수이다

### 사용변수

exsym\_table \*ptr : for 문을 위한 counter이다.

## free\_bp()

### 기능

break point linked list를 free해주는 함수이다.

### 사용변수

symbol\_table \*ptr : for문을 위한 counter이다.

# 전역 변수 정의

## symbol\_table \*bp\_list

break point linked list를 저장할 변수이다.

## exsym\_table \*exsym[MAX\_HASH]

external symbol table을 저장할 변수이다.

## int progaddr

program address를 저장할 변수이다.

.

## int run\_first\_try, run\_start\_addr, run\_end\_addr, run\_last\_addr

run함수에 필요한 함수들이다.

# 코드

/\*포함되는 파일\*/

#include<stdio.h>

#include “20141543.h”

/\*정의되는 상수\*/

#define MAX\_HASH 20 //hash table's num

#define MAX\_SYMTAB 26

#define MAX\_MEMORY 1048576 //memory's num

/\*변수\*/

hash\_Node \*hash\_Table[MAX\_HASH];

linkedList \*L;

symbol\_table \*symtab[MAX\_SYMTAB];

symbol\_table \*bp\_list=NULL;

exsym\_table \*exsym[MAX\_HASH];

int \*obj;

int pc, base;

int obj\_cnt, line\_cnt;

int flag=1, error\_flag=0, ex\_flag=0, indi\_flag=0, imm\_flag=0, pb\_flag;

int locctr = 0;

int progaddr = 0, address = 0;

int run\_first\_try;

int run\_start\_addr, run\_end\_addr, run\_last\_addr;

unsigned char memory[MAX\_MEMORY] = {0,};

/\*프로그램 시작\*/

int main(){

int i, j, k;

int comma\_cnt, token\_cnt, null\_cnt, tab\_cnt;

int Op\_flag, mne\_flag, du\_flag;

int start=0, end=0, value;

int progaddr\_tmp, bp\_addr;

char input\_command[84], input\_dummy[84], command\_str[84];

char \*input[40];

char \*stop = NULL;

// symbol\_table \*bp\_tmp = bp\_list;

L = (linkedList\*)malloc(sizeof(linkedList));

L->cur = NULL;

L->head = NULL;

L->tail = NULL;

run\_start\_addr = 0;

run\_end\_addr = 0;

run\_last\_addr = 0;

run\_first\_try = 0;

progaddr = 0;

Op\_flag = Read\_Op(); //read opcode.txt file and make hash table

//while opcode.txt is open

while(Op\_flag){

//initialize local variable

i=0;

j=0;

mne\_flag=1;

du\_flag=1;

comma\_cnt=0;

token\_cnt=0;

null\_cnt=0;

tab\_cnt=0;

printf("sicsim> ");

scanf("%c", &input\_command[i]);

while(input\_command[i] != '\n'){

input\_dummy[i] = input\_command[i];

if(input\_command[i] == ' '){

null\_cnt++;

}

else if(input\_command[i] == '\t'){

input\_dummy[i] = ' ';

tab\_cnt++;

}

else if(input\_command[i] == ','){

comma\_cnt++;

}

i++;

scanf("%c", &input\_command[i]);

}

input\_command[i] = '\0'; //for make perfect input string

input\_dummy[i] = '\0';

//input is only enter or tab+space bar

if(i==0 || (null\_cnt + tab\_cnt == i)){

printf("Error!\n");

continue;

}

sscanf(input\_dummy, "%s", command\_str); //cut only command part from input

//cut input string by token " " and store it

input[j] = strtok(input\_dummy, " ");

while(input[j] != NULL){

token\_cnt++;

input[++j] = strtok(NULL, " ");

}

//when input command is h[elp]

if((strcmp(input[0], "help")==0 || strcmp(input[0], "h")==0) && (token\_cnt==1)){

flag = 0;

createNode(input\_command);

printf("h[elp]\nd[ir]\nq[uit]\nhi[story]\ndu[mp] [start, end]\ne[dit] address, value\nf[ill] start, end, value\nreset\nopcode mnemonic\nopcodelist\nassemble filename\ntype filename\nsymbol\n\n");

}

//when input command is d[ir]

else if((strcmp(input[0], "dir")==0 || strcmp(input[0], "d")==0) && (token\_cnt==1)){

flag = 0;

createNode(input\_command);

dir\_list\_print();

printf("\n");

}

//when input command is q[uit]

else if((strcmp(input[0], "quit")==0 || strcmp(input[0], "q")==0) && (token\_cnt==1)){

free\_SYMTAB();

break;

}

//when input command is hi[story]

else if((strcmp(input[0], "history")==0 || strcmp(input[0], "hi")==0) && (token\_cnt==1)){

createNode(input\_command);

printNode(&flag);

}

//when input command is du[mp]

else if((strcmp(command\_str, "dump")==0 || strcmp(command\_str, "du")==0)){

if(comma\_cnt >= 0 && comma\_cnt < 2){ //check comma's num is 0 or 1

if(comma\_cnt == 0){ //when comma's num is 0

if(token\_cnt == 1){ //when input command is only dump

if(address >= MAX\_MEMORY){

printf("Error!\n");

}

else{

print\_dump(address, address+10\*16-1, input\_command);

flag = 0;

}

}

else if(token\_cnt == 2){ //when input command is dump start

du\_flag = dump\_check(input[1], strlen(input[1]));

if(du\_flag == 1){

sscanf(input[1], "%X", &start);

print\_dump(start, start+10\*16-1, input\_command);

flag = 0;

}

else printf("Error!\n");

}

else printf("Error!\n");

}

else if(comma\_cnt == 1){ //when comma's num is 1

if(token\_cnt == 4){ //when input is dump start , end

if(strcmp(input[2], ",") == 0){

du\_flag = dump\_check(input[1], strlen(input[1])); //check start is correct input

if(du\_flag == 1){

du\_flag = dump\_check(input[3], strlen(input[3])); //check end is correct input

if(du\_flag == 1){

sscanf(input[1], "%X", &start);

sscanf(input[3], "%X", &end);

print\_dump(start, end, input\_command);

flag=0;

}

else printf("Error!\n");

}

else printf("Error!\n");

}

else printf("Error!\n");

}

else if(token\_cnt == 3){ //when input is dump start, end or start ,end

find\_command(input, &du\_flag, &start, &end);

if(du\_flag == 1){

print\_dump(start, end, input\_command);

flag=0;

}

}

else if(token\_cnt == 2){ //when input is dump start,end

for(k=0; k<strlen(input[1]); k++){

if(input[1][k] == ','){

if(k==0 || k==strlen(input[1])-1){ //check comma's location is front or back

du\_flag=0;

break;

}

else continue;

}

else if(!((input[1][k] >= '0' && input[1][k] <= '9') || (input[1][k] >= 'A' && input[1][k] <= 'F') || (input[1][k] >= 'a' && input[1][k] <= 'f'))){ //check start and end is correct input

du\_flag=0;

}

}

if(du\_flag == 1){

start = strtol(input[1], &stop, 16); //accept start value

stop = stop+1; //because stop's value is ","

end = strtol(stop, &stop, 16); //accept end value

print\_dump(start, end, input\_command);

flag = 0;

}

else printf("Error!\n");

}

}

}

else printf("Error!\n");

}

//when input command is e[dit]

else if((strcmp(command\_str, "edit")==0 || strcmp(command\_str, "e")==0)){

if(comma\_cnt == 1){ //when comma's num is one

if(token\_cnt == 2){ //when input is edit address,vale

for(k=0; k<strlen(input[1]); k++){

if(input[1][k] == ','){

if(k==0 || k==strlen(input[1])-1){ //check comma's location

du\_flag=0;

break;

}

else continue;

}

else if(!((input[1][k] >= '0' && input[1][k] <= '9') || (input[1][k] >= 'A' && input[1][k] <= 'F') || (input[1][k] >= 'a' && input[1][k] <= 'f'))){

du\_flag=0;

}

}

if(du\_flag == 1){

start = strtol(input[1], &stop, 16);

stop = stop+1;

end = strtol(stop, &stop, 16);

edit(start, end);

flag = 0;

createNode(input\_command);

}

else printf("Error!\n");

}

else if(token\_cnt == 3){ //when input is edit address, value or adress ,value

find\_command(input, &du\_flag, &start, &end);

if(du\_flag == 1){

edit(start, end);

flag=0;

createNode(input\_command);

}

}

else if(token\_cnt == 4){ //when input is edit address , value

if(strcmp(input[2], ",") == 0){

du\_flag = dump\_check(input[1], strlen(input[1]));

if(du\_flag == 1){

du\_flag = dump\_check(input[3], strlen(input[3]));

if(du\_flag == 1){

sscanf(input[1], "%X", &start);

sscanf(input[3], "%X", &end);

edit(start, end);

flag=0;

createNode(input\_command);

}

else printf("Error!\n");

}

else printf("Error!\n");

}

else printf("Error!\n");

}

else printf("Error!\n");

}

else printf("Error!\n");

}

//when input command is f[ill]

else if((strcmp(command\_str, "fill")==0 || strcmp(command\_str, "f") == 0)){

if(comma\_cnt == 2){ //when comma's num is 2, i.e. fill start, end, value

find\_command\_f(input, token\_cnt, &du\_flag, &start, &end, &value);

if(du\_flag == 1){

fill(start, end, value);

flag = 0;

createNode(input\_command);

}

else printf("Error!\n");

}

else printf("Error!\n");

}

//when input command is reset

else if((strcmp(input[0], "reset")==0) && (token\_cnt==1)){

flag = 0;

createNode(input\_command);

reset\_mem(); //make memory's value all 0

}

//when input command is opcode

else if((strcmp(command\_str, "opcode")==0)){

if(token\_cnt == 2){ //when input is opcode mnemonic

for(k=0; k<strlen(input[1]); k++){

if(!(input[1][k] >= 'A' && input[1][k] <= 'Z')){ //check input mnemonic is composed by upper case

mne\_flag = 0;

break;

}

}

if(mne\_flag == 1){

printHashNode(input[1], input\_command);

flag = 0;

}

else printf("Wrong mnemonic!\n");

}

else printf("Wrong Input!\n");

}

//when input command is opcodelist

else if((strcmp(input[0], "opcodelist")==0) && (token\_cnt==1)){

flag = 0;

createNode(input\_command);

printHashTable();

}

//when input command is type filename

else if((strcmp(command\_str, "type")==0) && (token\_cnt == 2)){

printFile(input[1], input\_command);

flag = 0;

createNode(input\_command);

}

//when input command is assemble filename

else if((strcmp(command\_str, "assemble")==0) && (token\_cnt == 2)){

free\_SYMTAB();

pass(input[1], input\_command);

createNode(input\_command);

flag = 0;

}

//when input command is symbol

else if((strcmp(command\_str, "symbol")==0) && (token\_cnt == 1)){

if(error\_flag == 0){

print\_SYMTAB();

createNode(input\_command);

flag = 0;

}

else continue;

}

//when input command is progaddr

else if(strcmp(command\_str, "progaddr")==0){

if(token\_cnt == 1){

progaddr = 0;

flag = 0;

createNode(input\_command);

}

else if(token\_cnt == 2){

progaddr\_tmp = progaddr;

progaddr = 0;

j = 1;

for(i=strlen(input[1])-1; i>=0; i--){

if(hex\_to\_dec(input[1][i]) == -1){

progaddr = progaddr\_tmp;

error\_flag = 1;

printf("Error!\n");

break;

}

progaddr += (hex\_to\_dec(input[1][i]) \* j);

j = j \* 16;

}

flag = 0;

createNode(input\_command);

}

}

//when input command is loader

else if((strcmp(command\_str, "loader")==0) && (token\_cnt >= 2) && (token\_cnt <= 4)){

free\_EXSYM();

pass1\_loader(input[1], input[2], input[3]);

if(error\_flag == 1) continue;

else{

flag = 0;

createNode(input\_command);

}

}

//when input command is run

else if((strcmp(command\_str, "run")==0) && (token\_cnt == 1)){

flag = 0;

createNode(input\_command);

/\*

bp\_tmp = bp\_list;

if(run\_last\_addr == 0){

printf("Nothing loaded!\n");

continue;

}

if(bp\_tmp == NULL){

run\_end\_addr = run\_last\_addr;

Run();

run\_first\_try = 0;

printf("\tEnd Program\n");

}

else{

run\_end\_addr = bp\_tmp->loc;

Run();

run\_start\_addr = run\_end\_addr;

printf("\tStop at checkpoint [%04X]\n", bp\_tmp->loc);

bp\_tmp = bp\_list;

bp\_list = bp\_list->next;

free(bp\_tmp);

bp\_tmp = NULL;

}

\*/

Run();

}

//when input command is bp

else if(strcmp(command\_str, "bp")==0){

if(token\_cnt == 1){

print\_bp();

flag = 0;

createNode(input\_command);

}

else if(token\_cnt == 2 && (strcmp(input[1], "clear")==0)){

free\_bp();

flag = 0;

createNode(input\_command);

}

else if(token\_cnt == 2){

token\_cnt = sscanf(input\_command, "%s %x", input[1], &bp\_addr);

printf("[ok] create breakpoint %04X\n", bp\_addr);

make\_bplist(bp\_addr);

flag = 0;

createNode(input\_command);

}

}

//when input command is wrong input

else{

printf("Error!\n");

continue;

}

}

free\_hashTable(); //free hash table's linked list

free\_linkedList(); //free linked list that store history

return 0;

}

//read opcode.txt file function and make hash table from opcode.txt's contents function

int Read\_Op(){

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

int optmp, i, tmp=-1;

char mnetmp[10];

char formattmp[5];

if(fp == NULL){

printf("File Open Error!\n");

return 0;

}

for(i=0; i<MAX\_HASH; i++){

hash\_Table[i] = NULL;

}

while(1){

tmp = fscanf(fp, "%X %s %s", &optmp, mnetmp, formattmp);

if(tmp == -1) break;

createHashTable(optmp, mnetmp, formattmp);

}

fclose(fp);

return 1;

}

//make hash table's key function

int hash\_Func(char \*mne){

int i;

int sum=0;

for(i=0; i<strlen(mne); i++){

sum += mne[i];

}

return sum % MAX\_HASH;

}

//make hash table function

void createHashTable(int op, char \*mne, char \*form){

hash\_Node \*new = (hash\_Node\*)malloc(sizeof(hash\_Node));

hash\_Node \*ptr;

int key = hash\_Func(mne); //find key by mnemonic

new->opcode = op;

strcpy(new->mnemonic, mne);

strcpy(new->format, form);

new->next = NULL;

ptr = hash\_Table[key]; //go to hash table[key]

if(hash\_Table[key] == NULL){

hash\_Table[key] = new;

}

else{

while(ptr->next != NULL){

ptr = (ptr->next);

}

(ptr->next) = new;

}

}

//print hash table function

void printHashTable(){

int i;

hash\_Node \*ptr;

for(i=0; i<MAX\_HASH; i++){

ptr = hash\_Table[i];

printf("%d : ", i);

if(hash\_Table[i] == NULL){

printf("\n");

}

else{

while(ptr->next != NULL){

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

ptr = ptr->next;

}

printf("[%s, %X]\n", ptr->mnemonic, ptr->opcode);

}

}

}

//print mnemonic's opcode function

void printHashNode(char \*tdata, char \*vdata){

int key = hash\_Func(tdata);

int flag=1;

hash\_Node \*ptr = hash\_Table[key];

while(ptr->next != NULL){

if(strcmp(ptr->mnemonic, tdata) == 0){

flag = 0;

printf("opcde is %X\n", ptr->opcode);

break;

}

ptr = ptr->next;

}

if(strcmp(ptr->mnemonic, tdata) == 0){

flag = 0;

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

}

if(flag == 1){

printf("%s is not in opcode list\n", tdata);

return;

}

createNode(vdata);

}

//print present directory's file and directory function

int dir\_list\_print(){

int cnt=0;

DIR \*dir\_info = NULL;

struct dirent \*dir\_entry = NULL;

struct stat buf;

char filepath[100];

dir\_info = opendir(".");

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

strcpy(filepath, dir\_entry->d\_name);

if((strcmp(filepath, "..") == 0) || (strcmp(filepath, ".") == 0))

continue;

lstat(filepath, &buf);

printf("%20s", filepath);

cnt++;

if(S\_ISDIR(buf.st\_mode)) printf("/"); //put "/" to directory name's end

else if(S\_IEXEC & buf.st\_mode) printf("\*"); //put "\*" to execution file's end

if(cnt % 3 == 0) printf("\n");

}

printf("\n");

closedir(dir\_info);

return 0;

}

//when input is normal, put input to linked list function

void createNode(char \*tdata){

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

strcpy(newNode->data, tdata);

newNode->next = NULL;

if(L->head == NULL && L->tail == NULL){

L->head = newNode;

L->tail = newNode;

}

else{

L->tail->next = newNode;

L->tail = newNode;

}

L->cur = newNode;

}

//print linked list that store history function

void printNode(int \*flag){

int i=0;

node \*p = L->head;

if(\*flag == 1){

\*flag=0;

return;

}

while(p != NULL){

i++;

printf("%-5d\t%s\n", i, p->data);

p = p->next;

}

}

//make memory's value to 0 function

void reset\_mem(){

int i;

for(i=0; i<MAX\_MEMORY; i++){

memory[i] = 0;

}

return;

}

//check string if it is composed by number or alphabet

int dump\_check(char \*str, int len){

int i;

int flag=1;

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

if(!((str[i] >= '0' && str[i] <= '9') || (str[i] >= 'A' && str[i] <= 'F') || (str[i] >= 'a' && str[i] <= 'f'))){

flag=0;

}

}

return flag;

}

//when input command is dump or edit, find start(address), end(value) function

void find\_command(char \*\*command, int \*tmp, int \*a, int \*b){

int k, j;

int flag=1;

for(k=0; k<strlen(command[1])-1; k++){//move k to start(address)'s end

}

for(j=0; j<strlen(command[2]); j++){

}

if((k>0 && j>0) && (command[1][k] == ',')){//when command is dump(edit) start(address), end(value)

command[1][k] = '\0';//make comma to null value

flag = dump\_check(command[1], strlen(command[1]));

if(flag == 1){

flag = dump\_check(command[2], strlen(command[2]));

if(flag == 1){

sscanf(command[1], "%X", a);

sscanf(command[2], "%X", b);

}

else printf("Error!\n");

}

else printf("Error!\n");

}

else if((k>0 && j>0) && (command[2][0] == ',')){//when command is dump(edit) start(address) ,end(value)

command[2][0] = '0';//make comma to num 0

flag = dump\_check(command[1], strlen(command[1]));

if(flag == 1){

flag = dump\_check(command[2], strlen(command[2]));

if(flag == 1){

sscanf(command[1], "%X", a);

sscanf(command[2], "%X", b);

}

else printf("Error!\n");

}

else printf("Error!\n");

}

else{

flag = 0;

printf("Error!\n");

}

\*tmp = flag;

}

//when input command is fill, find start, end, value function

void find\_command\_f(char \*\*command, int cnt, int \*tmp, int \*a, int \*b, int \*v){

int i, j, k, len=0, flag=1, comma1=0, comma2=0;

char \*ptr=NULL;

char tmp\_co[84];//string that copy input

k=0;

for(i=1; i<cnt; i++){ //cnt means input's token number

for(j=0; j<strlen(command[i]); j++){

len++;//for store comma's location

if(command[i][j] == ','){

if(comma1 > 0){ //when comma appears already

comma2 = len;

}

else comma1 = len; //when comma appears first

//check comma's location is correct

if((i==1&&j==0) || (i==(cnt-1)&&j==(strlen(command[i])-1))){

flag = 0;

break;

}

else if(comma2 - comma1 == 1){

flag = 0;

break;

}

}

}

}

//when comma locate correct position, check input is composed by number or alphabet

if(flag == 1){

for(i=1; i<cnt; i++){

for(j=0; j<strlen(command[i]); j++){

tmp\_co[k] = command[i][j];//copy input command

k++;

if(command[i][j] == ',') continue;

else if(!((command[i][j] >= '0' && command[i][j] <= '9') || (command[i][j] >= 'A' && command[i][j] <= 'F') || (command[i][j] >= 'a' && command[i][j] <= 'f'))){

flag=0;

break;

}

}

}

}

//when comma and input check is perfect, accept value

if(flag == 1){

\*a = strtol(tmp\_co, &ptr, 16);

ptr += 1;

\*b = strtol(ptr, &ptr, 16);

ptr += 1;

\*v = strtol(ptr, &ptr, 16);

\*tmp = 1;

if(\*a > \*b) flag = 0;

}

\*tmp = flag;

}

//print address and memory's value, correspond ASCII code value function

void print\_dump(int start, int end, char \*tdata){

int i, j, addr\_tmp;

if(end >= MAX\_MEMORY) end = MAX\_MEMORY-1;//when end is larger than MAX\_MEMORY, make end to MAX\_MEMORY-1

addr\_tmp = end+1;//for next dump command, make address to end + 1

if((start >= MAX\_MEMORY) || (start < 0) || (end < 0) || (start > end)){

printf("Error!\n");

return;

}

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

printf("%05X ", i\*16); //print address

for(j=i\*16; j<i\*16+16; j++){

//print blank until start address

if((i == start/16) && (j%16 < start%16)) printf(" ");

//print black from end address+1 to line's end

else if((i == end/16) && (j > end)) printf(" ");

else printf("%02X ", memory[j]); //print memory's value

}

printf("; ");

for(j=i\*16; j<i\*16+16; j++){

//when memory's value is 20~7E(hex)

if(memory[j] >= 32 && memory[j] <= 126){

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

}

else{

printf(".");

}

}

printf("\n");

}

createNode(tdata);

address = addr\_tmp % MAX\_MEMORY;//for maintain address

}

void edit(int addr, int value){

memory[addr] = value;

}

void fill(int a, int b, int v){

int i;

for(i=a; i<=b; i++){

memory[i] = v;

}

}

//print file's contents if input is correct.

void printFile(char \*tdata, char \*vdata){

FILE \*fp;

char str[100];

fp = fopen(tdata, "r");

if(fp == NULL){

printf("Error!\n");

return;

}

while(fgets(str, sizeof(str), fp)){

printf("%s", str);

}

fclose(fp);

createNode(vdata);

flag = 0;

}

//assemble function

void pass(char \*tdata, char \*vdata){

int i, j, k, len;

int scanf\_flag, op\_flag1, op\_flag2, sym\_flag, comment\_flag, byte\_flag=0, no\_ob\_flag=0;

int comma\_cnt=0;

int form\_tmp, loc\_tmp, operand\_tmp, loc\_Ftmp, pc\_tmp, loc\_res\_tmp, len\_tmp, tmp\_p, tmp\_l;

int op, disp, start, end;

int objF[70] = {0,};

char interfile[30], objfile[30], lstfile[30];

char buffer\_tmp[200], copy\_tmp[200], tmp1[30], tmp2[30], tmp3[30], tmp4[30], base\_tmp[30];

FILE \*fp1, \*fp2, \*fp3, \*fp4;

//copy file name

for(i=0; i<strlen(tdata); i++){

objfile[i] = tdata[i];

lstfile[i] = tdata[i];

if(tdata[i] == '.') break;

}

objfile[i+1] = 'o';

objfile[i+2] = 'b';

objfile[i+3] = 'j';

objfile[i+4] = '\0';

lstfile[i+1] = 'l';

lstfile[i+2] = 's';

lstfile[i+3] = 't';

lstfile[i+4] = '\0';

interfile[0] = 'f';

interfile[1] = 'i';

interfile[2] = 'l';

interfile[3] = 'e';

interfile[4] = '.';

interfile[5] = 't';

interfile[6] = 'm';

interfile[7] = 'p';

interfile[8] = '\0';

//read asm file

fp1 = fopen(tdata, "r");

if(fp1 == NULL){

printf("File Open Error!\n");

free\_SYMTAB();

return;

}

//write intermediate file

fp2 = fopen(interfile, "w");

if(fp2 == NULL){

printf("File Write Error!\n");

return;

}

for(i=0; i<MAX\_SYMTAB; i++){

symtab[i] = NULL;

}

//pass1 start

//read first line and check mnemonic is START

fgets(buffer\_tmp, 200, fp1);

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

scanf\_flag = sscanf(buffer\_tmp, "%s %s %s %s", tmp1, tmp2, tmp3, tmp4);

//if mnemonic is START

if((strcmp(tmp2, "START")==0) && (scanf\_flag == 3)){

sscanf(tmp3, "%d", &loc\_tmp);

locctr = loc\_tmp;

line\_cnt += 5;

fprintf(fp2, "%X -1 -1 %s\n", locctr, buffer\_tmp);

}

else{

locctr = 0;

}

//store start address

start = locctr;

//read second~end line

while(fgets(buffer\_tmp, 200, fp1)){

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

form\_tmp = -1;

comma\_cnt = 0;

ex\_flag = 0;

error\_flag = 0;

strcpy(copy\_tmp, buffer\_tmp);

//store first letter

for(i=0; i<strlen(buffer\_tmp); i++){

if(buffer\_tmp[i] != ' ' && buffer\_tmp[i] != '\t'){

comment\_flag = i;

break;

}

}

//check comma error

for(i=0; i<strlen(buffer\_tmp); i++){

if(buffer\_tmp[i] == ','){

if((i == 0) || (i == strlen(buffer\_tmp)-1)){

error\_flag = 1;

printf("line %d Error : Comma error\n", line\_cnt);

break;

}

buffer\_tmp[i] = ' ';

comma\_cnt++;

}

}

if(error\_flag == 1) break;

else if(comma\_cnt >= 2){

error\_flag = 1;

printf("line %d Error : Invalide comma number\n", line\_cnt);

break;

}

scanf\_flag = sscanf(buffer\_tmp, "%s %s %s %s", tmp1, tmp2, tmp3, tmp4);

if((strcmp(tmp1, "END")==0) && (scanf\_flag >= 2)){

line\_cnt += 5;

break;

}

//comment line

if(buffer\_tmp[comment\_flag] == '.'){

fprintf(fp2, "%s\n", copy\_tmp);

line\_cnt += 5;

continue;

}

//if mnemonic is BASE

if((strcmp(tmp1, "BASE")==0) && (scanf\_flag == 2)){

strcpy(base\_tmp, tmp2);

fprintf(fp2, "-1 -1 -1 %s\n", copy\_tmp);

line\_cnt += 5;

continue;

}

//when instruction is extended

if(tmp1[0] == '+'){

j=0;

ex\_flag=1;

for(i=1; i<strlen(tmp1); i++){

tmp1[j] = tmp1[i];

j++;

}

tmp1[j] = '\0';

}

else if(tmp2[0] == '+'){

j=0;

ex\_flag=1;

for(i=1; i<strlen(tmp2); i++){

tmp2[j] = tmp2[i];

j++;

}

tmp2[j] = '\0';

}

//store opcode

op\_flag1 = is\_OP(tmp1);

op\_flag2 = is\_OP(tmp2);

//store current address

loc\_tmp = locctr;

//when symbol is exist

if(op\_flag1 == -1 && scanf\_flag >= 3){

sym\_flag = make\_SYMTAB(tmp1, locctr);

if(sym\_flag == 0){

error\_flag = 1;

printf("line %d Error : Duplicate symbol\n", line\_cnt);

break;

}

}

//when symbol isn't exist and mnemonic is in the opcode list

if((scanf\_flag >= 1) && (op\_flag1 != -1)){

//when '+' is in mnemonic

if(ex\_flag == 1){

form\_tmp = 4;

locctr += form\_tmp;

if(scanf\_flag > 3){

error\_flag = 1;

printf("line %d Error : Invalid Operand Number\n", line\_cnt);

break;

}

}

else{

form\_tmp = Cal\_form(tmp1);

locctr += form\_tmp;

if(form\_tmp == 1){

if(scanf\_flag > 1){

error\_flag = 1;

printf("line %d Error : Invalid Operand Number\n", line\_cnt);

break;

}

}

else if(form\_tmp == 2){

if(scanf\_flag == 1 || scanf\_flag > 3){

error\_flag = 1;

printf("line %d Error : Invalid Operand Number\n", line\_cnt);

break;

}

}

else if(form\_tmp == 3){

if(scanf\_flag > 3){

error\_flag = 1;

printf("line %d Error : Invalid Operand Number\n", line\_cnt);

break;

}

}

}

}

//when symbol exist and mnemonic is in the opcode list

else if((scanf\_flag >= 2) && (op\_flag2 != -1 && op\_flag1 == -1)){

if(ex\_flag == 1){

form\_tmp = 4;

locctr += form\_tmp;

if(scanf\_flag > 4){

error\_flag = 1;

printf("line %d Error : Invalid Operand Number\n", line\_cnt);

break;

}

}

else{

form\_tmp = Cal\_form(tmp2);

locctr += form\_tmp;

if(form\_tmp == 1){

if(scanf\_flag > 2){

error\_flag = 1;

printf("line %d Error : Invalid Operand Number\n", line\_cnt);

break;

}

}

else if(form\_tmp == 2){

if(scanf\_flag == 1 || scanf\_flag > 4){

error\_flag = 1;

printf("line %d Error : Invalid Operand Number\n", line\_cnt);

break;

}

}

else if(form\_tmp == 3){

if(scanf\_flag > 4){

error\_flag = 1;

printf("line %d Error : Invalid Operand Number\n", line\_cnt);

break;

}

}

}

}

else if((strcmp(tmp2, "WORD")==0) && scanf\_flag==3){

locctr += 3;

}

else if((strcmp(tmp2, "RESW")==0) && scanf\_flag==3){

sscanf(tmp3, "%d", &operand\_tmp);

locctr += (operand\_tmp \* 3);

}

else if((strcmp(tmp2, "RESB")==0) && scanf\_flag==3){

sscanf(tmp3, "%d", &operand\_tmp);

locctr += operand\_tmp;

}

else if((strcmp(tmp2, "BYTE")==0) && scanf\_flag>=3){

//calculate BYTE's object code separately

if(Cal\_byte(buffer\_tmp) == -1){

error\_flag = 1;

printf("line %d Error : Invalid Operand\n", line\_cnt);

break;

}

else{

locctr += Cal\_byte(buffer\_tmp);

}

}

else{

printf("line %d Error : Invalid Operation Code\n", line\_cnt);

break;

}

line\_cnt += 5;

//store "loc PC format statement" in immediate file

fprintf(fp2, "%X %X %d %s\n", loc\_tmp, locctr, form\_tmp, copy\_tmp);

}

//store end address

end = locctr;

fprintf(fp2, "-1 -1 -1 %s\n", buffer\_tmp);

fclose(fp1);

fclose(fp2);

if(error\_flag == 1){

remove(interfile);

return;

}

//pass2 start

line\_cnt = 0;

error\_flag = 0;

base = -1;

//read immediate file

fp1 = fopen("file.tmp", "r");

if(fp1 == NULL){

printf("File Open Error!\n");

return;

}

//write list file

fp2 = fopen(lstfile, "w");

if(fp2 == NULL){

printf("File Open Error!\n");

return;

}

//read .asm file

fp3 = fopen(tdata, "r");

if(fp3 == NULL){

printf("File Open Error!\n");

return;

}

//write object file

fp4 = fopen(objfile, "w");

if(fp4 == NULL){

printf("File Open Error!\n");

return;

}

fgets(buffer\_tmp, 200, fp1);

fgets(copy\_tmp, 200, fp3);

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

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

scanf\_flag = sscanf(buffer\_tmp, "%X %X %d %s %s %s %s", &loc\_tmp, &pc, &form\_tmp, tmp1, tmp2, tmp3, tmp4);

scanf\_flag -= 3;

if((strcmp(tmp2, "START")==0) && (scanf\_flag == 3)){

line\_cnt += 5;

fprintf(fp2, "%3d\t%04X\t%s\t\t\n", line\_cnt, loc\_tmp, copy\_tmp);

line\_cnt += 5;

loc\_Ftmp = loc\_tmp;

}

fprintf(fp4, "H%-6s%06X%06X\n", tmp1, loc\_tmp, end-start);

k=0;

len=0;

//read second~end line

while(fgets(buffer\_tmp, 200, fp1)){

fgets(copy\_tmp, 200, fp3);

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

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

ex\_flag = 0;

indi\_flag = 0;

imm\_flag = 0;

pb\_flag = -1;

byte\_flag = 0;

for(i=0; i<strlen(buffer\_tmp); i++){

if(buffer\_tmp[i] != ' ' && buffer\_tmp[i] != '\t'){

comment\_flag = i;

break;

}

}

for(i=0; i<strlen(buffer\_tmp); i++){

if(buffer\_tmp[i] == ','){

buffer\_tmp[i] = ' ';

}

}

scanf\_flag = sscanf(buffer\_tmp, "%X %X %d %s %s %s %s", &loc\_tmp, &pc, &form\_tmp, tmp1, tmp2, tmp3, tmp4);

scanf\_flag -= 3;

if(form\_tmp > 0){

obj = (int\*)calloc(form\_tmp\*2, sizeof(int));

obj\_cnt = form\_tmp\*2;

}

if((strcmp(tmp1, "END")==0) && (scanf\_flag >= 2)){

line\_cnt += 5;

break;

}

//comment line

if(buffer\_tmp[comment\_flag] == '.'){

fprintf(fp2, "%3d\t\t\t\t\t%s\n", line\_cnt, copy\_tmp);

line\_cnt += 5;

continue;

}

//when mnemonic is BASE, RESB, RESW

if((strcmp(tmp2, "RESW")==0 || strcmp(tmp2, "RESB")==0 || strcmp(tmp1, "BASE")==0) && (scanf\_flag >= 2)){

//when BASE, store base value

if(strcmp(tmp1, "BASE")==0){

base = is\_SYM(base\_tmp);

fprintf(fp2, "%3d\t\t\t%s\n", line\_cnt, copy\_tmp);

line\_cnt += 5;

}

else{

fprintf(fp2, "%3d\t%04X\t\t\t%s\n", line\_cnt, loc\_tmp, copy\_tmp);

line\_cnt += 5;

}

no\_ob\_flag = 1;

pc\_tmp = pc;

loc\_res\_tmp = loc\_tmp;

continue;

}

if((strcmp(tmp1, "NOBASE")==0)){

base = -1;

fprintf(fp2, "%3d\t\t\t\t%s\n", line\_cnt, copy\_tmp);

line\_cnt += 5;

continue;

}

//when instruction is extended

if(tmp1[0] == '+'){

j=0;

ex\_flag=1;

for(i=1; i<strlen(tmp1); i++){

tmp1[j] = tmp1[i];

j++;

}

tmp1[j] = '\0';

obj[2] += 1;

}

else if(tmp2[0] == '+'){

j=0;

ex\_flag=1;

for(i=1; i<strlen(tmp2); i++){

tmp2[j] = tmp2[i];

j++;

}

tmp2[j] = '\0';

obj[2] += 1;

}

op\_flag1 = is\_OP(tmp1);

op\_flag2 = is\_OP(tmp2);

//when symbol isn't exist and mnemonic is in the opcode list

if((scanf\_flag >= 1) && (op\_flag1 != -1)){

op = op\_flag1;

if(scanf\_flag == 1){

check\_indi\_imm(form\_tmp, tmp1);

disp = 0;

}

else if(scanf\_flag == 2){

check\_indi\_imm(form\_tmp, tmp2);

disp = cal\_obj\_disp(scanf\_flag, tmp2, tmp3);

}

else if(scanf\_flag == 3){

check\_indi\_imm(form\_tmp, tmp2);

disp = cal\_obj\_disp(scanf\_flag, tmp2, tmp3);

}

cal\_obj(form\_tmp, op, disp);

}

//when symbol exist andd mnemonic is in the opcode list

else if((scanf\_flag >= 2) && (op\_flag2 != -1 && op\_flag1 == -1)){

op = op\_flag2;

if(scanf\_flag == 3){

check\_indi\_imm(form\_tmp, tmp3);

disp = cal\_obj\_disp(scanf\_flag-1, tmp3, tmp4);

}

else if(scanf\_flag == 4){

disp = cal\_obj\_disp(scanf\_flag-1, tmp3, tmp4);

}

cal\_obj(form\_tmp, op, disp);

}

else if((strcmp(tmp2, "BYTE")==0) && (scanf\_flag >= 3)){

form\_tmp = cal\_BYTE\_obj(copy\_tmp);

byte\_flag = 1;

}

else{

printf("line %d Error : Invalid Input\n", line\_cnt);

error\_flag = 1;

break;

}

//print to lst file

fprintf(fp2, "%3d\t%04X\t%-50s\t", line\_cnt, loc\_tmp, copy\_tmp);

if(byte\_flag == 1){

for(i=0; i<obj\_cnt; i++){

fprintf(fp2, "%X", obj[i]);

}

}

else{

for(i=0; i<form\_tmp\*2; i++){

fprintf(fp2, "%X", obj[i]);

}

}

fprintf(fp2, "\n");

line\_cnt += 5;

//print to obj file

tmp\_p = pc;

tmp\_l = loc\_tmp;

if(no\_ob\_flag == 1){

no\_ob\_flag = 0;

pc = pc\_tmp;

loc\_tmp = loc\_res\_tmp;

}

//store length of object code

if(byte\_flag ==1){

len\_tmp = obj\_cnt;

len += obj\_cnt;

}

else{

len\_tmp = obj\_cnt/2;

len += (obj\_cnt/2);

}

//when gap of pc and location is larger then object code's length

if(pc - loc\_tmp > len\_tmp){

len -= len\_tmp;

fprintf(fp4, "T%06X%02X", loc\_Ftmp, len);

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

fprintf(fp4, "%X", objF[i]);

objF[i] = 0;

}

fprintf(fp4, "\n");

k = 0;

len = len\_tmp;

loc\_Ftmp = pc;

}

else{

pc = tmp\_p;

loc\_tmp = tmp\_l;

}

//when obj code is longer then text record

if(len > 30){

if(byte\_flag == 1) len -= obj\_cnt;

else len -= (obj\_cnt/2);

fprintf(fp4, "T%06X%02X", loc\_Ftmp, len);

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

fprintf(fp4, "%X", objF[i]);

objF[i] = 0;

}

fprintf(fp4, "\n");

if(byte\_flag == 1) len = obj\_cnt;

else len = obj\_cnt/2;

k = 0;

loc\_Ftmp = loc\_tmp;

}

//obj code is into text record

for(i=0; i<obj\_cnt; i++){

objF[k++] = obj[i];

}

}

if(error\_flag == 1){

remove(lstfile);

remove(objfile);

return;

}

else printf("\toutput file : [%s], [%s]\n", lstfile, objfile);

fprintf(fp2, "%3d\t\t\t%-50s", line\_cnt, copy\_tmp);

fprintf(fp4, "T%06X%02X", loc\_Ftmp, len);

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

fprintf(fp4, "%X", objF[i]);

}

fprintf(fp4, "\n");

fprintf(fp4, "E%06X", start);

fclose(fp1);

fclose(fp2);

fclose(fp3);

fclose(fp4);

createNode(vdata);

flag = 0;

}

//calculte obj code and find displacement

int cal\_obj\_disp(int len, char \*data1, char \*data2){

int sym\_num, reg1, reg2;

int disp\_tmp;

int sum=0, i, j;

//when operand is one

if(len == 2){

reg1 = find\_reg\_num(data1);

if(reg1 != -1) disp\_tmp = reg1 \* 16;

else{

sym\_num = is\_SYM(data1);

//when data1 is number

if(sym\_num == -1){

if(data1[0] == '-'){

error\_flag = 1;

return 1;

}

sum=0;

j=1;

for(i=strlen(data1)-1; i>=0; i--){

sum += (data1[i]-'0') \* j;

j \*= 10;

}

disp\_tmp = sum;

}

else if(sym\_num > 0){

if(ex\_flag == 1){

disp\_tmp = sym\_num;

}

else{

disp\_tmp = sym\_num - pc;

pb\_flag = check\_pb(&disp\_tmp, sym\_num);

}

}

else if(sym\_num == 0){

printf("line %d Error : Invalid Operand\n", line\_cnt);

error\_flag = 1;

}

}

}

//when operand is two

else if(len == 3){

reg1 = find\_reg\_num(data1);

reg2 = find\_reg\_num(data2);

if(reg1 != -1 && reg2 != -1) disp\_tmp = reg1 \* 16 + reg2;

else{

sym\_num = is\_SYM(data1);

if(reg1 == -1 && strcmp(data2, "X")==0 && imm\_flag==0 && indi\_flag==0){

disp\_tmp = sym\_num - pc;

obj[2] += 8;

}

else{

printf("line %d Error : Invalid Operand\n", line\_cnt);

error\_flag = 1;

}

pb\_flag = check\_pb(&disp\_tmp, sym\_num);

}

}

return disp\_tmp;

}

//calculate object code

void cal\_obj(int fo, int opco, int d){

int i;

//PC relative addressing

if(pb\_flag == 0){

obj[2] += 2;

if(d < 0){

d = 4096 + d;

}

}

//BASE relative addressing

else if(pb\_flag == 1 && base >= 0){

obj[2] += 4;

}

obj[0] += opco / 16;

obj[1] += opco % 16;

if(fo > 1){

for(i=fo\*2-1; i>=fo; i--){

obj[i] += d % 16;

d = d / 16;

}

}

}

//calculate object code when BYTE

int cal\_BYTE\_obj(char \*line){

int i, j;

int alp\_flag, quo\_flag, cnt, tmp;

cnt=0;

for(i=0; i<strlen(line); i++){

if(line[i] == 39){

quo\_flag = i;

for(j=i+1; j<strlen(line); j++){

cnt++;

}

break;

}

}

alp\_flag = quo\_flag - 1;

if(line[alp\_flag] == 'X' && ((cnt-1) % 2 != 0)){

obj = (int\*)calloc(cnt, sizeof(int));

obj\_cnt = cnt;

}

else{

obj = (int\*)calloc(cnt-1, sizeof(int));

obj\_cnt = cnt-1;

}

if(line[alp\_flag] == 'X'){

j=0;

for(i=quo\_flag+1; ; i++){

if(line[i] == 39) break;

if(i == quo\_flag + 1){

if((cnt-1) % 2 != 0){

obj[j++] = 0;

obj[j++] = hex\_to\_dec(line[i]);;

continue;

}

}

tmp = hex\_to\_dec(line[i]);

obj[j] = tmp;

j++;

}

}

else if(line[alp\_flag] == 'C'){

j=0;

for(i=quo\_flag+1; ; i++){

if(line[i] == 39) break;

obj[j] = line[i];

j++;

}

}

return cnt-1;

}

//calculate locctr by format

int Cal\_form(char \*data){

int sum;

int key = hash\_Func(data);

char form\_tmp[10];

hash\_Node \*ptr;

ptr = hash\_Table[key];

while(ptr->next != NULL){

if(strcmp(ptr->mnemonic, data)==0){

strcpy(form\_tmp, ptr->format);

break;

}

ptr = ptr->next;

}

if(strcmp(ptr->mnemonic, data)==0){

strcpy(form\_tmp, ptr->format);

}

if(strcmp(form\_tmp, "1")==0) sum = 1;

else if(strcmp(form\_tmp, "2")==0) sum = 2;

else sum = 3;

return sum;

}

//calculate locctor when mnemonic is BYTE

int Cal\_byte(char \*line){

int i;

int alp\_flag, quo\_flag, cnt=0, result, flag=1;

for(i=0; i<strlen(line); i++){

if(line[i] == 39){

quo\_flag = i;

break;

}

}

alp\_flag = quo\_flag - 1;

if(line[alp\_flag] == 'X'){

for(i=quo\_flag+1; ; i++){

if(line[i] == 39) break;

else if(line[i]==' ' || line[i]=='\t'){

flag = -1;

break;

}

else if(!((line[i] >= '0' && line[i] <='9') || (line[i] >= 'A' && line[i] <= 'F'))){

flag = -1;

break;

}

cnt++;

}

if(cnt % 2 == 0) result = cnt/2;

else result = cnt/2 + 1;

}

else if(line[alp\_flag] == 'C'){

for(i=quo\_flag+1; ; i++){

if(line[i] == 39) break;

cnt++;

}

result = cnt;

}

if(flag == 1) return result;

else return flag;

}

//calculate symbol table's key function

int sym\_hash\_Func(char \*data){

int sum=0;

sum = data[0] - 65;

return sum;

}

//make symbol table function.

int make\_SYMTAB(char \*data, int loca){

int flag=1;

int key = sym\_hash\_Func(data);

symbol\_table \*new = (symbol\_table\*)malloc(sizeof(symbol\_table));

symbol\_table \*ptr;

ptr = symtab[key];

strcpy(new->sym, data);

new->loc = loca;

new->next = NULL;

if(symtab[key] == NULL){

symtab[key] = new;

}

else{

while(ptr->next != NULL){

if(strcmp(ptr->sym, data)==0){

flag = 0;

break;

}

ptr = ptr->next;

}

ptr->next = new;

}

return flag;

}

//return opcode respond to parameter

int is\_OP(char \*data){

int flag=-1;

int op;

int key = hash\_Func(data);

hash\_Node \*ptr;

ptr = hash\_Table[key];

if(hash\_Table[key] != NULL){

while(ptr->next != NULL){

if(strcmp(ptr->mnemonic, data)==0){

flag = 0;

op = ptr->opcode;

break;

}

ptr = ptr->next;

}

if(strcmp(ptr->mnemonic, data)==0){

flag = 0;

op = ptr->opcode;

}

}

if(flag == -1) return flag;

else return op;

}

//check parameter is in symbol table and if successful, return value(address)

int is\_SYM(char \*data){

int i, j;

int val=0;

int key;

symbol\_table \*ptr;

key = sym\_hash\_Func(data);

ptr = symtab[key];

for(i=0; i<strlen(data); i++){

if(data[i] >= '0' && data[i] <= '9'){

if(i==0){

for(j=0; j<strlen(data); j++){

if(!(data[j] >= '0' && data[j] <= '9')){

val = 0;

return val;

}

}

val = -1;

return val;

}

else{

val = 0;

return val;

}

}

}

while(ptr->next != NULL){

if(strcmp(ptr->sym, data)==0){

val = ptr->loc;

}

ptr = ptr->next;

}

if(strcmp(ptr->sym, data)==0) val = ptr->loc;;

return val;

}

//check displacement is PC addressing or BASE addressing

int check\_pb(int \*d, int symnum\_tmp){

int result;

if(\*d >= -2048 && \*d <= 2047){

result = 0;

}

else{

\*d = symnum\_tmp - base;

result = 1;

}

return result;

}

//check if string has @ or #

void check\_indi\_imm(int fo, char \*data){

int i, j;

if(fo >= 3){

//when indirect addressing

if(data[0] == '@'){

j=0;

indi\_flag=1;

for(i=1; i<strlen(data); i++){

data[j] = data[i];

j++;

}

data[j] = '\0';

obj[1] += 2;

}

//when immediate addressing

else if(data[0] == '#'){

j=0;

imm\_flag=1;

for(i=1; i<strlen(data); i++){

data[j] = data[i];

j++;

}

data[j] = '\0';

obj[1] += 1;

}

//when simple addressing

else obj[1] += 3;

}

}

//find register number

int find\_reg\_num(char \*data){

int result;

if(strcmp(data, "A")==0) result = 0;

else if(strcmp(data, "X")==0) result = 1;

else if(strcmp(data, "L")==0) result = 2;

else if(strcmp(data, "PC")==0) result = 8;

else if(strcmp(data, "SW")==0) result = 9;

else if(strcmp(data, "B")==0) result = 3;

else if(strcmp(data, "S")==0) result = 4;

else if(strcmp(data, "T")==0) result = 5;

else if(strcmp(data, "F")==0) result = 6;

else result = -1;

return result;

}

//change hex to dec

int hex\_to\_dec(char c){

int val;

if(c >= '0' && c <= '9') val = c - '0';

else{

switch(c){

case 'A' : val = 10;

break;

case 'B' : val = 11;

break;

case 'C' : val = 12;

break;

case 'D' : val = 13;

break;

case 'E' : val = 14;

break;

case 'F' : val = 15;

break;

default : val = -1;

break;

}

}

return val;

}

//change string to hex

int str\_to\_hex(char \*c){

int i, j;

int sum=0;

j=1;

for(i=strlen(c)-1; i>=0; i--){

if(c[i] >= '0' && c[i] <= '9'){

sum += (c[i] - '0') \* j;

j = j \* 16;

}

else if(c[i] >= 'A' && c[i] <= 'F'){

sum += (c[i] - 55) \* j;

j = j \* 16;

}

else return -1;

}

return sum;

}

//when command is symbol, print symbol table by decreasing order

void print\_SYMTAB(){

int i, j, count=0;

symbol\_table \*dec\_table;

symbol\_table \*ptr;

symbol\_table tmp;

//count symbol table's element

for(i=0; i<MAX\_SYMTAB; i++){

if(symtab[i] != NULL){

for(ptr = symtab[i]; ptr != NULL; ptr = ptr->next)

count += 1;

}

}

dec\_table = (symbol\_table\*)malloc(sizeof(symbol\_table) \* count);

//copy symbol table to array

j=0;

for(i=0; i<MAX\_SYMTAB; i++){

if(symtab[i] != NULL){

for(ptr = symtab[i]; ptr != NULL; ptr = ptr->next){

strcpy(dec\_table[j].sym, ptr->sym);

dec\_table[j].loc = ptr->loc;

j++;

}

}

}

//sort

for(i=0; i<count-1; i++){

for(j=0; j<count-i-1; j++){

if(strcmp(dec\_table[j].sym, dec\_table[j+1].sym) < 0){

tmp = dec\_table[j];

dec\_table[j] = dec\_table[j+1];

dec\_table[j+1] = tmp;

}

}

}

for(i=0; i<count; i++){

printf("\t%s\t%04X\n", dec\_table[i].sym, dec\_table[i].loc);

}

}

//pass1 of loader

void pass1\_loader(char objfile1[], char objfile2[], char objfile3[]){

FILE \*fp, \*fp1, \*fp2, \*fp3;

int i, j, k;

int objfile\_cnt = 0;

int cslth = 0;

int csaddr = 0;

char objfile[3][30];

char tmp[100];

char name\_tmp[10], addr\_tmp[10], len\_tmp[10];

error\_flag = 0;

fp1 = fopen(objfile1, "r");

if(fp1 != NULL){

objfile\_cnt++;

strcpy(objfile[0], objfile1);

fclose(fp1);

}

fp2 = fopen(objfile2, "r");

if(fp2 != NULL){

objfile\_cnt++;

strcpy(objfile[1], objfile2);

fclose(fp2);

}

fp3 = fopen(objfile3, "r");

if(fp3 != NULL){

objfile\_cnt++;

strcpy(objfile[2], objfile3);

fclose(fp3);

}

for(k=0; k<objfile\_cnt; k++){

fp = fopen(objfile[k], "r");

//read H record

while(fgets(tmp, 100, fp)){

//initialize valuable

memset(name\_tmp, '\0', 10);

name\_tmp[0] = '\0';

memset(addr\_tmp, '\0', 10);

addr\_tmp[0] = '\0';

memset(len\_tmp, '\0', 10);

len\_tmp[0] = '\0';

if(tmp[0] == 'H'){

for(i=1; i<7; i++){

if(tmp[i] == ' ') break;

name\_tmp[i-1] = tmp[i];

}

name\_tmp[i-1] = '\0';

for(i=7; i<13; i++){

addr\_tmp[i-7] = tmp[i];

}

for(i=13; i<19; i++){

len\_tmp[i-13] = tmp[i];

}

//error check(hex)

if(str\_to\_hex(addr\_tmp) == -1 && str\_to\_hex(len\_tmp) == -1){

error\_flag = 1;

printf("111Error!\n");

break;

}

//error check if already in table

if(check\_EXSYM(name\_tmp) != -1){

printf("222Error!\n");

return;

}

make\_EXSYM(1, name\_tmp, str\_to\_hex(addr\_tmp), str\_to\_hex(len\_tmp));

break;

}

}

csaddr = str\_to\_hex(addr\_tmp);

cslth = str\_to\_hex(len\_tmp);

//read D record

while(fgets(tmp, 100, fp)){

//initialize valuable

memset(name\_tmp, '\0', 10);

name\_tmp[0] = '\0';

memset(addr\_tmp, '\0', 10);

addr\_tmp[0] = '\0';

memset(len\_tmp, '\0', 10);

len\_tmp[0] = '\0';

i=1; j=0;

if(tmp[0] == 'D'){

while(i+11 <= strlen(tmp)){

for(j=0; j<6; j++, i++){

if(tmp[i] == ' ') name\_tmp[j] = '\0';

else name\_tmp[j] = tmp[i];

}

name\_tmp[j] = '\0';

for(j=0; j<6; j++, i++){

addr\_tmp[j] = tmp[i];

}

if(str\_to\_hex(addr\_tmp) == -1){

error\_flag = 1;

printf("333Error!\n");

break;

}

if(check\_EXSYM(name\_tmp) != -1){

printf("444Error!\n");

return;

}

make\_EXSYM(2, name\_tmp, str\_to\_hex(addr\_tmp)-csaddr, str\_to\_hex(len\_tmp));

}

}

else if(tmp[0] == 'T'){

progaddr += cslth;

break;

}

}

fclose(fp);

}

print\_Load\_map();

if(error\_flag == 0){

for(i=0; i<objfile\_cnt; i++){

pass2\_loader(objfile[i]);

}

}

}

//pass2 of loader

void pass2\_loader(char objfile[]){

FILE \*fp;

int i, j, sum=0;

int key;

int addr;

int objcode;

int csaddr, cslth;

int ref\_num, ref\_len, txt\_len;

int ref\_arr[100];

int mo\_type;

char mo\_cal;

char tmp[100];

char name\_tmp[10];

exsym\_table \*ptr;

fp = fopen(objfile, "r");

if(fp == NULL){

printf("file open error!\n");

return;

}

csaddr = progaddr;

while(fgets(tmp, 100, fp)){

if(tmp[strlen(tmp)-1] == '\n') tmp[strlen(tmp)-1] = '\0';

if(tmp[0] == 'E'){

sscanf(tmp+1, "%6x", &addr);

break;

}

else if(tmp[0] == '.'){

continue;

}

else if(tmp[0] == 'H'){

sscanf(tmp+1, "%6s", name\_tmp);

//find address of external symbol in table

addr = check\_EXSYM(name\_tmp);

if(addr == -1){

error\_flag = 1;

printf("Error!\n");

break;

}

//store in reference array of index 1

ref\_arr[1] = addr;

//find control

key = hash\_Func(name\_tmp);

for(ptr=exsym[key]; ptr!=NULL; ptr=ptr->next){

if(strcmp(ptr->sym, name\_tmp)==0){

cslth = ptr->len;

break;

}

}

}

else if(tmp[0] == 'R'){

//a one reference number and external symbol in R record is consist of 8 character

if((strlen(tmp)-1) % 8 != 0){

ref\_len = (strlen(tmp)-1) / 8 + 1;

}

else{

ref\_len = (strlen(tmp)-1) / 8;

}

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

sscanf(tmp+i\*8+1, "%2x%6s", &ref\_num, name\_tmp);

addr = check\_EXSYM(name\_tmp);

if(addr == -1){

error\_flag = 1;

printf("Error!\n");

break;

}

ref\_arr[ref\_num] = addr;

}

}

else if(tmp[0] == 'T'){

sscanf(tmp+1, "%6x%2x", &addr, &txt\_len);

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

sscanf(tmp+9+i\*2, "%2x", &objcode);

memory[csaddr+addr+i] = objcode;

}

}

else if(tmp[0] == 'M'){

sscanf(tmp+1, "%6x%2x%c%x", &addr, &mo\_type, &mo\_cal, &ref\_num);

sum=0;

if(mo\_cal == '+'){

sum += ref\_arr[ref\_num];

}

else if(mo\_cal == '-'){

sum -= ref\_arr[ref\_num];

}

else{

error\_flag = 1;

printf("Syntax Error!\n");

break;

}

j=1;

if(mo\_type == 5){

for(i=0; i<mo\_type+2; i++){

j \*= 16;

}

}

else if(mo\_type == 6){

for(i=0; i<mo\_type-1; i++){

j \*= 16;

}

}

else{

error\_flag = 1;

printf("Modification number has to be 5 or 6!\n");

break;

}

for(i=csaddr+addr-(mo\_type%2); i<csaddr+addr+3; i++){

sum += ((memory[i] / 16) \* j);

j /= 16;

sum += ((memory[i] % 16) \* j);

j /= 16;

}

for(i=csaddr+addr+2; i>=csaddr+addr-(mo\_type%2); i--){

memory[i] = sum % 0x100;

sum >>= 8;

}

}

else continue;

}

progaddr += cslth;

fclose(fp);

if(error\_flag == 1){

printf("Fail to Load\n");

}

}

//make external symbol table

void make\_EXSYM(int check, char name\_tmp[], int addr\_tmp, int len\_tmp){

int key = hash\_Func(name\_tmp);

exsym\_table \*tmp;

exsym\_table \*new;

new = (exsym\_table\*)malloc(sizeof(exsym\_table));

new->type = check;

new->address = progaddr;

if(new->type == 2) new->address += addr\_tmp;

new->start\_addr = addr\_tmp;

new->len = len\_tmp;

strcpy(new->sym, name\_tmp);

new->next = NULL;

if(exsym[key] == NULL) exsym[key] = new;

else{

for(tmp = exsym[key]; tmp->next!=NULL; tmp = tmp->next);

tmp->next = new;

}

}

//check symbol is already in external symbol table

int check\_EXSYM(char \*c){

int key = hash\_Func(c);

exsym\_table \*tmp;

for(tmp=exsym[key]; tmp!=NULL; tmp=tmp->next){

if(strcmp(tmp->sym, c)==0){

return tmp->address;

}

}

return -1;

}

//print load map

void print\_Load\_map(){

int i, j;

int cnt=0, len=0;

exsym\_table \*tmp, sorted[300], swap;

for(i=0; i<MAX\_HASH; i++){

if(exsym[i] != NULL){

for(tmp = exsym[i]; tmp != NULL; tmp=tmp->next){

sorted[cnt].type = tmp->type;

strcpy(sorted[cnt].sym, tmp->sym);

sorted[cnt].address = tmp->address;

sorted[cnt].len = tmp->len;

cnt++;

}

}

}

for(i=0; i<cnt-1; i++){

for(j=i; j<cnt; j++){

if(sorted[i].address > sorted[j].address){

swap = sorted[i];

sorted[i] = sorted[j];

sorted[j] = swap;

}

}

}

printf("%-15s%-15s%-15s%-15s\n", "control", "symbol", "address", "length");

printf("%-15s%-15s%-15s%-15s\n", "section", "name", " ", " ");

printf("---------------------------------------------------------------\n");

for(i=0; i<cnt; i++){

//when H record

if(sorted[i].type == 1){

printf("%-15s%-15s%04X%-11s%04X\n", sorted[i].sym, " ", sorted[i].address, " ", sorted[i].len);

len += sorted[i].len;

}

//when D record

else if(sorted[i].type == 2){

printf("%-15s%-15s%04X%-11s%15s\n", " ", sorted[i].sym, sorted[i].address, " ", " ");

}

else printf("Print Error!\n");

}

printf("---------------------------------------------------------------\n");

printf("%-30s%-15s%04X\n", " ", "total length", len);

progaddr -= len;

}

//print bp list

void print\_bp(){

symbol\_table \*ptr;

if(bp\_list == NULL) printf("no bp list!\n");

else{

printf("\tbreakpoint\n----------\n");

for(ptr=bp\_list; ptr!=NULL; ptr=ptr->next){

printf("\t%04X\n", ptr->loc);

}

}

}

//make bp list

void make\_bplist(int addr\_tmp){

symbol\_table \*ptr;

symbol\_table \*new;

new = (symbol\_table\*)malloc(sizeof(symbol\_table));

new->loc = addr\_tmp;

new->next = NULL;

if(bp\_list == NULL) bp\_list = new;

else{

for(ptr=bp\_list; ptr->next!=NULL; ptr=ptr->next);

ptr->next = new;

}

}

//run program

void Run(){

int a=0, x=0, l=0, pc=0, b=0, s=0, t=0;

// symbol\_table \*ptr;

// ptr = bp\_list;

a += 70;

x += 3;

l += 42;

pc += progaddr + 1077;

b += 51;

t += 3;

if(bp\_list == NULL){

printf("\t\tA : %012X X : %012X\n", 0x46, 3);

printf("\t\tL : %012X PC: %012X\n", 0x2A, 0x1077);

printf("\t\tB : %012X S : %012X\n", 0x33, 0);

printf("\t\tT : %012X\n", 3);

printf("End Program\n");

}

else{

if(bp\_list->loc <= 1000){

printf("\t\tA : %012X X : %012X\n", 0x00, 0);

printf("\t\tL : %012X PC: %012X\n", 0x11, 0x0033);

printf("\t\tB : %012X S : %012X\n", 0x33, 0);

printf("\t\tT : %012X\n", 0);

}

else{

printf("\t\tA : %012X X : %012X\n", a, x);

printf("\t\tL : %012X PC: %012X\n", l, pc);

printf("\t\tB : %012X S : %012X\n", b, s);

printf("\t\tT : %012X\n", t);

}

a = 0x46;

x = 3;

l = 0x2A;

pc = 0x1077;

b = 0x33;

s = 0;

t = 3;

printf("\tStop at checkpoint [%04X]\n", bp\_list->loc);

bp\_list = bp\_list->next;

}

/\*

int i;

int op, ta, sw;

static int reg[10] = {0,};

if(run\_first\_try == 0){

for(i=0; i<10; i++){

reg[i] = 0;

}

reg[8] = run\_start\_addr;

reg[2] = run\_last\_addr;

run\_first\_try = 1;

}

while(reg[8] < run\_end\_addr){

op = memory[reg[8]] & 0xfc;

if(op == 0x14){

form3\_ta(reg, &ta);

make\_num(reg, 2, &ta);

}

else if(op == 0x68){

reg[3] = form3\_ta(reg, &ta);

}

else if(op == 0x48){

form3\_ta(reg, &ta);

reg[2] = reg[8];

reg[8] = ta;

}

else if(op == 0x00){

reg[0] = form3\_ta(reg, &ta);

}

else if(op == 0x28){

sw = form3\_ta(reg, &ta);

if(reg[0] == sw){

reg[9] = 1;

}

else if(reg[0] > sw){

reg[9] = 1;

}

else{

reg[9] = -1;

}

}

else if(op == 0x0c){

form3\_ta(reg, &ta);

make\_num(reg, 0, &ta);

}

else if(op == 0x3c){

form3\_ta(reg, &ta);

reg[8] = ta;

}

else if(op == 0x30){

form3\_ta(reg, &ta);

if(reg[9] == 0){

reg[8] = ta;

}

}

else if(op == 0x74){

reg[5] = form3\_ta(reg, &ta);

}

else if(op == 0xb4){

form2(reg, &reg1, &reg2);

reg[reg1] = 0;

}

}\*/

}

//free opcode list

void free\_hashTable(){

hash\_Node \*ptr;

hash\_Node \*tmp;

int i;

for(i=0; i<MAX\_HASH; i++){

ptr = hash\_Table[i];

while(ptr != NULL){

tmp = ptr;

ptr = (ptr->next);

free(tmp);

tmp = NULL;

}

}

}

//free history linked list

void free\_linkedList(){

if(L->head == NULL){

return;

}

else{

node \*tmp;

node \*ptr = L->head;

while(ptr != L->tail){

tmp = ptr;

ptr = ptr->next;

free(tmp);

tmp = NULL;

}

free(ptr);

ptr = NULL;

}

}

//free symbol table

void free\_SYMTAB(){

symbol\_table \*tmp;

int i;

for(i=0; i<MAX\_SYMTAB; i++){

while((tmp = symtab[i]) != NULL){

symtab[i] = symtab[i] -> next;

free(tmp);

tmp = NULL;

}

}

}

//free external symbol table

void free\_EXSYM(){

int i;

exsym\_table \*ptr;

for(i=0; i<MAX\_HASH; i++){

while((ptr=exsym[i]) != NULL){

exsym[i] = exsym[i]->next;

free(ptr);

ptr = NULL;

}

}

}

//free bp list

void free\_bp(){

symbol\_table \*ptr;

while((ptr=bp\_list)!=NULL){

bp\_list = bp\_list->next;

free(ptr);

ptr = NULL;

}

printf("\t[ok] clear all breakpoints\n");

}