**과목명: 시스템프로그래밍**

**1 분반**

**<<Project #3>**

**서강대학교 컴퓨터공학과**

**[학번] 20161643**

**[이름] 정진원**

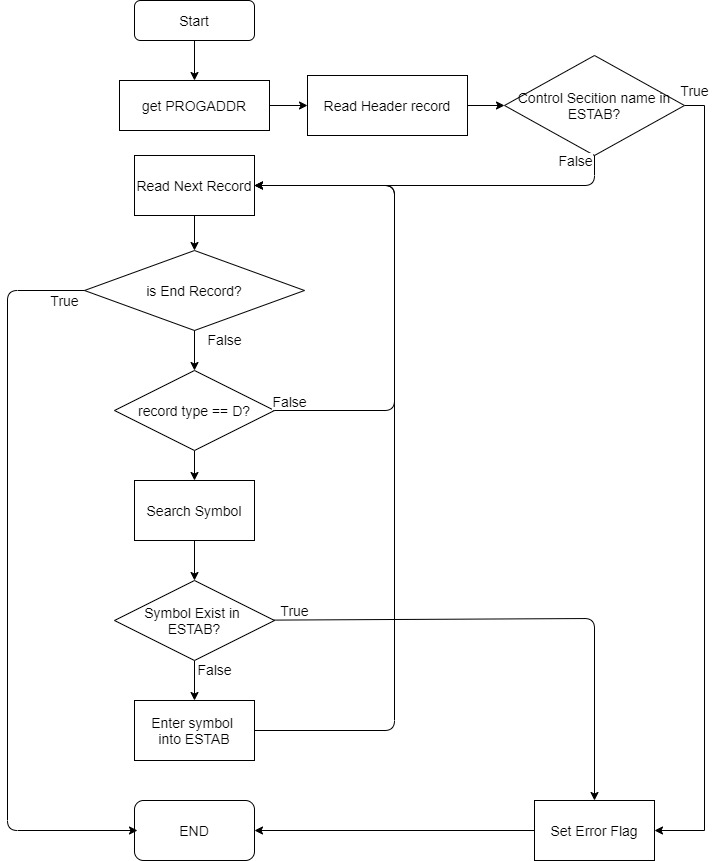
목 차

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
3. **모듈 정의**
   1. 모듈이름: void cmdProgaddr(char cmd[])
      1. 기능
      2. 사용변수
   2. 모듈이름: void printloadmap()
      1. 기능
      2. 사용변수
   3. 모듈이름: int loadpass1(int count, char f1[][30])
      1. 기능
      2. 사용변수
   4. 모듈이름: int loadpass2(int count, char f1[][30])
      1. 기능
      2. 사용변수
   5. 모듈이름: void cmdLoader(char cmd[])
      1. 기능
      2. 사용변수
   6. 모듈이름: void cmdBp(char cmd[])
      1. 기능
      2. 사용변수
   7. 모듈이름: void printReg(int reg[])
      1. 기능
      2. 사용변수
   8. 모듈이름: int getTarget(int curr, int reg[], int nixbpe)
      1. 기능
      2. 사용변수
   9. 모듈이름: int calcVal(int TA, int nixbpe){ int calcVal(int TA, int nixbpe)
      1. 기능
      2. 사용변수
   10. 모듈이름: void cmdRun(char cmd[])
       1. 기능
       2. 사용변수
4. **전역 변수, 구조체, typedef 정의**
   1. typedef struct \_Exsym
   2. unsigned int bptab[1048576] = {0,}
   3. unsigned int progaddr = 0;
   4. Exsym\* estab[20];
   5. int file\_len = 0;
   6. int end\_exec = 0;
   7. int bpset[1000] = {0,};
   8. int bpnum = 0;
   9. int bpflag = -1;
   10. int reg[10] = {0,};
   11. int loadflag = 0;
5. **코드 설명**
   1. 20161643.h
   2. 20161643.c
6. **프로그램 개요**

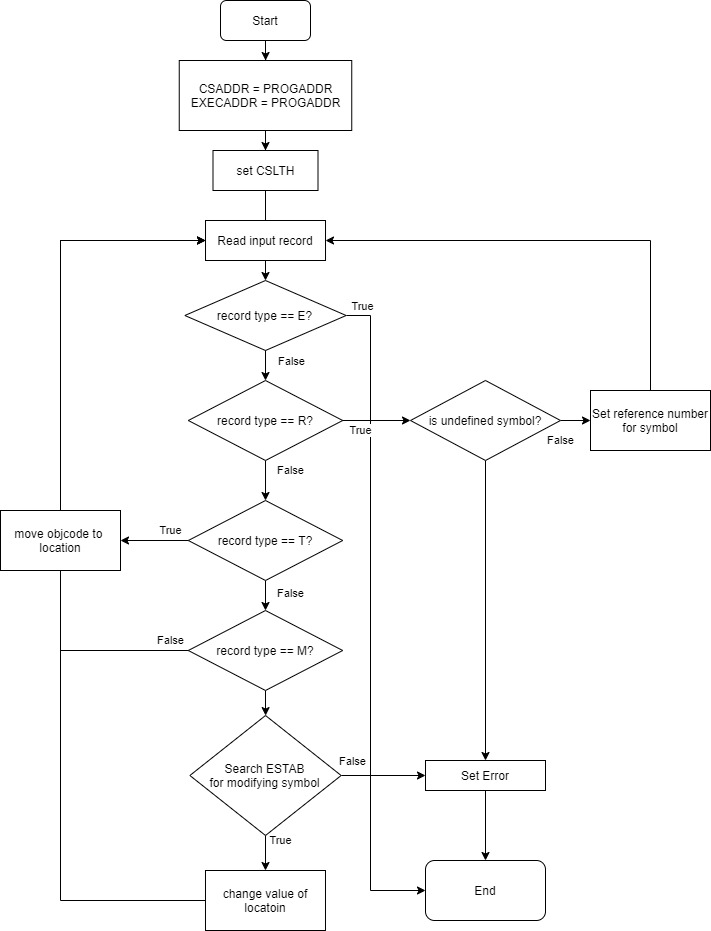
프로젝트 1과 2에서 구현한 shell에서 linking loading 기능을 추가한 프로그램이다. 프로젝트2 에서 생성된 obj 파일을 link하여 프로젝트 1에서 구현한 메모리에 올리는 역할을 수행한다. 작업을 수행하기 위해 2 pass를 이용하였다. 또한 load된 프로그램을 run할 수 있다.

1. **프로그램 설명**
   1. **프로그램 흐름도**

**PASS1**

****

**PASS2**

****

1. **모듈 정의**
   1. 모듈이름: void cmdProgaddr(char cmd[])
      1. 기능

Progaddr 명령어가 입력되면 수행되는 함수로, 입력된 address로 progaddr의 값을 설정해준다.

* + 1. 사용변수

int idx – address가 시작되는 위치를 확인하기 위한 index

int I – loop용 변수

int cmdLen – 입력된 command의 길이를 저장하는 변수

* 1. 모듈이름: void printloadmap()
     1. 기능

ESTAB에 저장된 symbol의 이름과 CSADDR를 형식에 맞게 출력해주는 함수. 만약 control section의 이름인 경우 length까지 출력해준다.

* + 1. 사용변수

int I – loop 용 변수

Exsym \*curr – estab를 출력해주기 위한 변수

int total\_length - cslen끼리 더해줘서 총 길이를 저장해주는 변수

* 1. 모듈이름: int loadpass1(int count, char f1[][30])
     1. 기능

Linking loader의 pass1 algorithm을 수행하는 함수이다. 전달된 파일들의 Header record를 확인하여 control section name과 cslth ESTAB에 저장해준다. 이미 존재하는 이름인 경우 에러 처리를 해주었다. 이후 D record를 확인하여 symbol들을 ESTAB에 저장해준다. 마찬가지로 중복된 symbol인 경우 에러처리를 해주었다.

* + 1. 사용변수

FILE\* fp – 파일을 열기 위한 파일 포인터

int i, j, hash, linenum – loop용 변수, symbol의 hash 값을 저장하기 위한 변수, 에러가 발생한 line을 확인하기 위해 line 번호를 저장하는 함수

unsigned int csaddr, cslen, loc – control section의 시작 위치, 길이를 저장하는 변수, symbol의 위치를 저장해주는 변수

char filename[30], sym[7] – file 이름과 symbol이름을 저장해주기 위한 변수

char line[150] – obj 파일로부터 한 줄씩 읽기 위한 변수

Exsym \*curr, \*node – ESTAB에 새로운 node를 추가하기 위한 변수

* 1. 모듈이름: int loadpass2(int count, char f1[][30])
     1. 기능

Linking loader의 pass2 algorithm을 수행하기 위한 함수. 전달된 파일들의 R record를 확인해 reference number에 symbol의 주소를 넣어준다. 만약 pass1에서 정의되지 않은 symbol이 등장할 경우 에러처리를 해주었다. T record의 경우 objcode를 읽어서 메모리에 저장해준다. M record의 경우 해당하는 location의 값을 수정하여 다시 저장해주었다.

* + 1. 사용변수

FILE\* fp – 파일을 열기 위한 변수

int i, j, linenum, hash – loop용 변수, 에러가 발생한 라인을 확인하기 위한 변수, symbol의 hash값을 저장하는 변수

int objcode – objcode를 읽어와서 저장하는 변수

unsigned int addr, tlen – text record에서 주소와 길이를 저장하기 위한 변수

unsigned int csaddr, cslen, rnum, refer[30] = {0,}, hb, maddr – cs addr, length를 저장하기 위한 변수, reference number를 저장하기 위한 변수, reference number에 해당하는 symbol의 주소를 저장하기 위한 배열, m record의 half byte의 개수, m record에 의해 수정된 값을 저장하기 위한 변수

char filename[30], sym[7] – 파일 이름과 symbol을 저장하기 위한 변수

char line[150], sign – 파일에서 한 줄씩 읽기 위한 변수, m record의 부호를 저장하기 위한 변수

* 1. 모듈이름: void cmdLoader(char cmd[])
     1. 기능

Loader 명령어가 입력되면 수행되는 함수. Filename이 주어지지 않으면 에러처리를 했다. 3개까지의 파일을 입력 받을 수 있다. Pass1과 pass2가 성공적으로 수행되면 estab에 저장된 값을 printloadmap() 함수를 이용해 출력해준다. 또 history에 명령어를 추가한다. 또 loadflag를 1로 세팅해 run 명령어가 수행 될 수 있게 해준다.

* + 1. 사용변수

int count – 파일의 개수를 저장하는 변수

int i, j – loop용 변수

int cmdLen – 입력받은 명령어의 길이를 저장하기 위한 변수

int pass1, pass2 – pass 1과, pass2 함수의 return 값을 저장하는 변수. 오류가 없으면 1이고 오류가 있으면 값이 -1이다. -1을 return 받은 경우 에러를 출력하였다.

char f1[3][30] = {'\0'} – 여러 개의 파일 이름들을 저장하기 위해 2차원 배열을 이용하였다.

* 1. 모듈이름: void cmdBp(char cmd[])
     1. 기능

Bp 명령어가 입력된 경우 수행되는 함수. Bp만 입력된 경우 저장된 break point들의 값을 출력해준다. Bp clear가 입력된 경우 저장된 bp들을 삭제해준다. Bp address가 입력된 경우 address가 제대로 입력된 경우 이를 bptab과 bpset에 저장해준다. 범위를 벗어나거나 잘못된 address가 입력된 경우 에러처리를 해주었다.

* + 1. 사용변수

int i, cmdlen – loop용 변수와 입력된 command의 길이를 저장하는 변수

int idx, bp – command에서 명령어 이외의 부분이 시작되는 위치를 저장하는 변수, break point를 저장하는 변수

* 1. 모듈이름: void printReg(int reg[])
     1. 기능

Run에 의해서 레지스터에 저장된 값들을 형식에 맞게 출력해주는 함수

* + 1. 사용변수
  1. 모듈이름: int getTarget(int curr, int reg[], int nixbpe)
     1. 기능

Object code를 해석해서 target address를 반환해주는 함수. Run 함수로부터 object code의 nixbpe 값을 받아서 format과 addressing 방식을 확인하고 그에 따른 target address 값을 계산해서 반환해준다.

* + 1. 사용변수

enum registers {A,X,L,B,S,T,F,NA,PC,SW} – 가독성을 올리기 위해 레지스터의 이름들을 enum으로 선언해주었다

int target, format, disp – 반환된 target address를 저장하는 변수, e flag를 확인해서 format 4인지 3인지 저장하는 변수, address의 변위를 계산해서 저장하는 변수

* 1. 모듈이름: int calcVal(int TA, int nixbpe){ int calcVal(int TA, int nixbpe)
     1. 기능

전달받은 target address에 저장된 값을 반환해주는 함수. Immediate addressing경우 target address를 다시 반환해주고, indirect나 simple addressing의 경우 주소에 저장된 값을 반환해준다.

* + 1. 사용변수

int result – 반환 값을 저장하기 위한 변수

* 1. 모듈이름: void cmdRun(char cmd[])
     1. 기능

Run 명령어가 입력되면 수행되는 함수. 메모리에 올라간 obj 파일을 읽어서 실행한다. Copy.obj에 나오는 명령어들의 기능을 구현하였다. 계산을 쉽게 하기 위해 bitwise operation들을 많이 사용하였다. PC가 가리키는 값에 break point가 존재한다면 수행을 종료하고, 수행되면서 레지스터에 저장된 값을 출력해준다. 프로그램이 끝까지 수행되어 종료된 경우, 레지스터들의 값을 초기화 해주고 pc를 다시 progaddr로 바꿔서 run을 실행한 경우 다시 처음부터 수행되도록 구현하였다.

* + 1. 사용변수

enum registers {A,X,L,B,S,T,F,NA,PC,SW} – 가독성을 올리기 위한 enum 변수

int nixbpe – objcect code의 nixbpe값을 저장하는 변수

int curr, end, target – 프로그램이 수행되는 위치, 프로그램이 종료되어야 할 위치와 objcode로부터 계산된 주소값을 저장하는 변수

int opcode, value – objcode의 opcode값을 저장하는 변수

int regnum, regnum2 – 레지스터 번호를 저장하는 변수

int I – loop용 변수

1. **전역변수, 구조체, typedef 정의**
   1. typedef struct \_Exsym

ESTAB를 hash table로 구성하기 위한 구조체이다. Symbol, csaddr, length(cs name을 저장), \_Exsym \*link로 구성되어 있다. Hash function은 앞선 프로젝트에서 구현한 createHash함수를 이용했다.

* 1. unsigned int bptab[1048576] = {0,}

breakpoint를 저장하기 위한 배열이다. 메모리의 크기와 같은 크기의 배열을 선언하였고, breakpoint가 존재하는 위치의 값은 1로 세팅해주어 run 함수에서 bptab[PC] == 1이면 breakpoint가 존재하는 것이므로 함수의 실행을 멈춰주었다.

* 1. unsigned int progaddr = 0;

progaddr를 저장하기 위한 변수이다. 프로그램 실행 시 0으로 초기화된다. Progaddr를 통해 수정할 수 있다.

* 1. Exsym\* estab[20];

Estab를 hash table로 구성하기 위한 배열이다. Create hash function을 재사용하기 위해 크기를 20으로 지정해주었다.

* 1. int file\_len = 0;

메모리에 올라간 총 길이를 저장하는 변수

* 1. int end\_exec = 0;

마지막으로 수행된 명령어가 가리키는 주소. 마지막으로 수행된 명령의 위치를 확인하기 위해 선언했다.

* 1. int bpset[1000] = {0,};

breakpoint들을 출력하기 위해, breakpoint를 순서대로 저장하는 배열

* 1. int bpnum = 0;

breakpoint를 출력하기 위해 입력된 breakpoint들의 개수를 저장하는 변수

* 1. int bpflag = -1;

bp == 0인경우를 확인하고 L register의 값을 유지하기 위해 설정해준 flag. Bp가 한번도 수행되지 않은 경우, bpflag == -1이고, 이때 bp[0]을 확인해준다. L register의 값 또한 bp가 수행되지 않은 경우에만 end값으로 만들어준다. 프로그램이 실행중인 경우에 l register의 값을 변경하면 안되기 때문에. Bp 가 실행되면 bpflag = 1로 설정해준다.

* 1. int reg[10] = {0,};

레지스터의 값들을 저장하는 전역변수

* 1. int loadflag = 0;

load가 되기 이전에 run을 실행하면 run되지 않도록 만들어주기 위해 선언한 전역변수. Load가 성공적으로 수행되면 loadflag = 1로 세팅해준다. 만약 reset을해서 메모리를 초기화하면 loadflag를 0으로 만들어주었다.

1. **코드 설명**
   1. 20161643.h

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <sys/stat.h>

#include <time.h>

#include <stdbool.h>

#include <ctype.h>

#include <math.h>

#define FALSE 0

#define TRUE 1

typedef void(\*funcPtr)(char \*);

typedef struct \_History {

char hCmd[100];

struct \_History \*link;

}History; // structure for saving history

typedef struct \_opNode {

int opcode;

char mnemonic[7], format[4];

struct \_opNode \*link;

}opNode; // structure for making opcode list

typedef struct \_Symbol{

char symbol[10];

int location;

struct \_Symbol \*link;

}Symbol;

typedef struct \_Object{

char objcode[10];

int loc;

int eflag;

int mflag;

struct \_Object\* link;

}Object;

typedef struct \_Exsym{

char symbol[7];

int csaddr;

int length;

struct \_Exsym \*link;

}Exsym; // structure for saving external symbols

bool endFlag = FALSE; // if input is q or quit, set the flag true

bool errFlag = FALSE; // when there is an invalid input in hex to decimal function

History \*head = NULL, \*tail = NULL; // pointer for the front and end of history list

unsigned char memory[1048576] = {0,}; // for memory space 1048756 = 16 ^ 5

unsigned int bptab[1048576] = {0,};

int dumpEnd = 0; // the last memory address which was printed

opNode\* opList[20]; // array for opcode list

int line\_num = 1; // int to check the line numbers of the asm code

int line\_num2 = 0; // int to check the numbers of intermediate file

int start\_address = 0; //the address where START points

Symbol\* symList[26] = {}; // array for SYMTAB

int LOCCTR = 0; // int for LOCCTR

int symNum = 0; // number of symbols

int enterflag = 0; // flag to check if we need an enter in obj file

Object \*ohead = NULL, \*otail = NULL; // pointer that points to the head and tail of obj list

int objfree = 0; //to check if free is already done;

Symbol \*saveSym = NULL; //save the symbol list from the last successful assemble

unsigned int progaddr = 0; //default progadder 0

Exsym\* estab[20]; // array for EXSYMTAB

int file\_len = 0;

int end\_exec = 0;

int bpset[1000] = {0,};

int bpnum = 0;

int bpflag = -1;

int reg[10] = {0,};

int loadflag = 0;

// function to change hexadecimal number to decimal number

int hextodec(char cmd[], int start, int end) {

int i; // loop iterator

int dec = 0; // converted decimal number

int num = 1;

bool startFlag = FALSE; // flag to find the starting point of the given address or value

for (i = end - 1; i >= start; i--) {

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

startFlag = TRUE; // set the start flag when the hexadecimal number starts

dec += (cmd[i] - '0') \* num;

num = num \* 16;

}

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

startFlag = TRUE;

dec += (cmd[i] - 'A' + 10) \* num;

num = num \* 16;

}

else if (cmd[i] >= 'a' && cmd[i] <= 'f') {

startFlag = TRUE;

dec += (cmd[i] - 'a' + 10) \* num;

num = num \* 16;

}

else if (cmd[i] == ' ' && startFlag == FALSE) // ignore spaces before the hexadecimal number starts

continue;

else { // if there is an input which is out of range, set errFlag /

errFlag = TRUE;

return 0;

}

}

return dec;

}

// function to create hash index for hash table

int createHash(char mnec[]) {

int i;

int result = 0;

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

result += (mnec[i] - 'A') \* 41; // creating a hash index

}

return result % 20;

}

// function to check the input command is an existing command

int checkCmd(char cmd[], char tcmd[]) {

int i, j, k, l = 0; // variables for loops

char temp[100]={'\0',}; //

for (i = 0; i < strlen(cmd); i++) { // trim spaces at front of the command

if (cmd[i] != ' ')

break;

}

for (j = strlen(cmd) - 1;; j--) { // trim spaces at the end of the command

if (cmd[j] != ' ')

break;

}

for (k = i; k <= j; k++) { // save the trimmed comamnd at tcmd

tcmd[l] = cmd[k];

l++;

}

//printf("%d %d\n",i,j);

tcmd[l + 1] = '\0';

for (i = 0; i < strlen(tcmd); i++) { //for commands with variables, find the command

if (tcmd[i] == ' ' || tcmd[i] == '\0')

break;

temp[i] = tcmd[i];

}

temp[i + 1] = '\0';

//printf("%s\n%s\n%s\n",cmd, tcmd, temp);

if (!strcmp(tcmd, "h") || !strcmp(tcmd, "help")) { // find the matching command and return a number

return 0;

}

else if (!strcmp(tcmd, "d") || !strcmp(tcmd, "dir")) {

return 1;

}

else if (!strcmp(tcmd, "q") || !strcmp(tcmd, "quit")) {

return 2;

}

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

return 3;

}

else if (!strcmp(temp, "du") || !strcmp(temp, "dump")) {

return 4;

}

else if (!strcmp(temp, "e") || !strcmp(temp, "edit")) {

return 5;

}

else if (!strcmp(temp, "f") || !strcmp(temp, "fill")) {

return 6;

}

else if (!strcmp(tcmd, "reset")) {

return 7;

}

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

return 8;

}

else if (!strcmp(tcmd, "opcodelist")) {

return 9;

}

else if (!strcmp(temp, "assemble")){

return 10;

}

else if (!strcmp(temp, "type")){

return 11;

}

else if (!strcmp(tcmd, "symbol")){

return 12;

}

else if(!strcmp(temp, "progaddr")){

return 13;

}

else if(!strcmp(temp, "loader")){

return 14;

}

else if(!strcmp(tcmd, "run")){

return 15;

}

else if(!strcmp(temp, "bp")){

return 16;

}

else

return -1;

}

// add command to the history list

void addHistory(char cmd[]) {

History \*new;

new = (History \*)malloc(sizeof(History));

strcpy(new->hCmd, cmd);

new->link = NULL; // set new node with passed command

if (!head) {

head = new;

tail = new;

}

else {

tail->link = new;

tail = new;

} // add the new node at the end of the list

}

// function to print the commnads

void cmdHelp(char cmd[]) {

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\n");

printf("assemble filename\ntype filename\nsymbol\n");

printf("progaddr\nloader\nbp\nbp clear\nbp address\nrun\n");

addHistory(cmd);

}

// function to print the files in the current directory

void cmdDir(char cmd[]) {

DIR \*dir = NULL;

struct dirent \*file = NULL;

struct stat cStat;

int prtNum = 0;

dir = opendir(".");

if (dir != NULL) { // point to the current directory

while ((file = readdir(dir))!= NULL) { // read the files in the directory

stat(file->d\_name, &cStat);

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

if (S\_ISDIR(cStat.st\_mode)) // if the read file is a directory print '/'

printf("/");

else if (S\_IXUSR & cStat.st\_mode) // if the file is a exe file print '\*'

printf("\*");

prtNum++;

if (prtNum % 4 == 0)

printf("\n");

}

closedir(dir);

if (prtNum % 4 != 0)

printf("\n");

}

addHistory(cmd);

}

// function to end the program

void cmdQuit(char cmd[]) {

endFlag = TRUE;

}

// function to read the history list and print it

void cmdHistory(char cmd[]) {

History\* temp;

int count = 1; // for the number of commands

addHistory(cmd);

for (temp = head; temp != NULL; temp = temp->link) { // follow the history list and print it in sequence

printf("%4d %s\n", count, temp->hCmd);

count++;

}

}

// function to print the allocated memory

void cmdDump(char cmd[]) {

int idx = 0, cidx = 0, sidx = 0, eidx = 0;// index which points to the needed location

int start = -1, end = -1; // starting and end point of the memory

int i, j;

int cmdLen; // length of the command

cmdLen = strlen(cmd);

while (1) {

if (cmd[idx] == ' ' || cmd[idx] == '\0')

break;

idx++;

} // find where the command part ends

if ((idx == 2 || idx == 4) && cmdLen == idx) { //if the input is just du or dump

start = dumpEnd;

end = start + 159;

if (end > 0xFFFFF)

end = 0xFFFFF;

dumpEnd = end + 1;

if (dumpEnd > 0xFFFFF)

dumpEnd = 0;

}

else {

for (i = idx; i < cmdLen; i++) {

if (cmd[i] == ',') {

cidx = i;

break;

} // find the index of the comma

}

for (i = idx; i < cmdLen; i++) {

if (cmd[i] != ' ') {

sidx = i;

break;

} // find the index where the starting address starts

}

if (cidx == cmdLen - 1) {

printf("Address needed after comma\n");

return;

} // exception : there isn't a address after the comma

else if (cidx == 0) { // if there isn't a comma, and just the starting address

start = hextodec(cmd, sidx, cmdLen); // convert the starting address

if (errFlag == TRUE) {

printf("Invalid Address\n");

errFlag = FALSE;

return;

} // exception : there was an invalid input

end = start + 159;

if (end > 0xFFFFF)

end = 0xFFFFF;

dumpEnd = end + 1;

if (dumpEnd > 0xFFFFF)

dumpEnd = 0;

}

else { // if there were two addresses given

for (i = cidx + 1; i < cmdLen; i++) {

if (cmd[i] != ' ') {

eidx = i;

break;

}

} // find the index where the end address starts

start = hextodec(cmd, sidx, cidx);

end = hextodec(cmd, eidx, cmdLen);

if (errFlag == TRUE) {

printf("Invalid Address\n");

errFlag = FALSE;

return;

} // exception : there was an invalid input

dumpEnd = end + 1;

if (dumpEnd > 0xFFFFF)

dumpEnd = 0;

}

}

if (start > end) {

printf("Start should be less than end\n");

return;

} // error when start adrress is bigger than end address

if (start > 0xFFFFF || end > 0xFFFFF) {

printf("Address should be less than 0xFFFFF\n");

return;

} // error when addresses are out of max range

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

printf("%05X ", i);

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

if (j >= start && j <= end)

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

else

printf(" ");

}

printf("; ");

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

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

putchar(memory[j]);

else

putchar('.');

}

printf("\n");

} // print saved memory into the given format

if (end == 0xFFFFF)

dumpEnd = 0;

addHistory(cmd);

}

// function to edit memory

void cmdEdit(char cmd[]) {

int idx = 0, cidx = 0, aidx = 0, vidx = 0; // index to save needed location

int address = 0, value = 0; // variable to save addresss and value

int i; // loop iterator

int cmdLen;

cmdLen = strlen(cmd);

while (1) {

if (cmd[idx] == ' ' || cmd[idx] == '\0')

break;

idx++;

} // find where the command ends

for (i = idx; i < cmdLen; i++) {

if (cmd[i] == ',') {

cidx = i;

break;

}

} // check the index of the comma

if (cidx == 0) {

printf("Need comma to identify parameters\n");

return;

} // if there is no commna, error

else if (cidx == cmdLen - 1) {

printf("Value is needed after comma\n");

return;

} // if there is no value after comma, error

for (i = idx; i < cmdLen; i++) {

if (cmd[i] != ' ') {

aidx = i;

break;

}

} // find the index of where the address starts

for (i = cidx + 1; i < cmdLen; i++) {

if (cmd[i] != ' ') {

vidx = i;

break;

}

} // find the index of where the value starts

address = hextodec(cmd, aidx, cidx); // change address to decimal number

value = hextodec(cmd, vidx, cmdLen); // change value to decimal number

if (errFlag == TRUE) {

printf("Invalid input\n");

errFlag = FALSE;

return;

} // if there was an invalid input error

if (address < 0x00000 || address > 0xFFFFF) {

printf("Address out of range\n");

return;

} // if address is out of range error

if (value < 0 || value > 0xFF)

{

printf("Value out of range\n");

return;

} // if value is out of range error

memory[address] = value;

addHistory(cmd);

}

// function to fill the address from start to end with value

void cmdFill(char cmd[]) {

int idx = 0, cidx1 = 0, cidx2 = 0, sidx = 0, eidx = 0, vidx = 0; // index to point to the needed location

int start = 0, end = 0, value = 0; // variables to save decimal values

int i;

int cmdLen;

cmdLen = strlen(cmd);

while (1) {

if (cmd[idx] == ' ' || cmd[idx] == '\0')

break;

idx++;

} // index where the command part ends

for (i = idx; i < cmdLen; i++) {

if (cmd[i] == ',') {

cidx1 = i;

break;

}

} // index of the first comma

for (i = cidx1 + 1; i < cmdLen; i++) {

if (cmd[i] == ',') {

cidx2 = i;

break;

}

} // index of the second comma

if (cidx1 == 0 || cidx2 == 0) {

printf("Need two commas to identify parameters\n");

return;

} // if there are less than two commas, error

else if (cidx1 == cmdLen - 1) {

printf("Invalid Command\n");

return;

} // if the command ends with a comma, error

else if (cidx2 == cmdLen - 1) {

printf("Need to input value\n");

return;

} // if there is no input value, error

for (i = idx; i < cmdLen; i++) {

if (cmd[i] != ' ') {

sidx = i;

break;

}

} // index of the starting point of the start address

for (i = cidx1 + 1; i < cmdLen; i++) {

if (cmd[i] != ' ') {

eidx = i;

break;

}

} // index of the starting point of the end address

for (i = cidx2 + 1; i < cmdLen; i++) {

if (cmd[i] != ' ') {

vidx = i;

break;

}

} // index of the starting point of the value

start = hextodec(cmd, sidx, cidx1);

end = hextodec(cmd, eidx, cidx2);

value = hextodec(cmd, vidx, cmdLen);

if (errFlag == TRUE) {

printf("Invalid Input\n");

errFlag = FALSE;

return;

} // if there was an invalid input, error

if (start < 0x00000 || start > 0xFFFFF || end < 0x00000 || end > 0xFFFFF) {

printf("Address out of range\n");

return;

} // if addresses are out of range, error

if (start > end) {

printf("Start should be less than end\n");

return;

} // if start address is bigger than end address error

if (value < 0 || value > 0xFF) {

printf("Value out of range\n");

return;

} // if value is out of range, error

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

memory[i] = value;

}

addHistory(cmd);

}

// function to reset memory

void cmdReset(char cmd[]) {

memset(memory, 0, sizeof(memory));

loadflag = 0;

addHistory(cmd);

}

// function to open opcode.txt and create opcode list

void loadOpcode() {

FILE\* fp;

char mnemonic[6], format[4];

int opcode, idx;

opNode \*new, \*temp;

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

if(fp == NULL){

printf("opcode.txt doesn't exist\n");

endFlag = TRUE;

return;

} // if opcode.txt doesn't exist error

while (fscanf(fp, "%x %s %s", &opcode, mnemonic, format) != EOF) {

new = (opNode \*)malloc(sizeof(opNode));

new->link = NULL;

new->opcode = opcode;

strcpy(new->format, format);

strcpy(new->mnemonic, mnemonic);

idx = createHash(mnemonic); // create hash with given mnemonic

if (opList[idx]) { // add opNode structure to opcode list

temp = opList[idx];

while (temp->link)

temp = temp->link;

temp->link = new;

}

else

opList[idx] = new;

}

fclose(fp);

return;

}

//function to find the instruction/filename from command

int findName(char cmd[], char name[]){

int i, j = 0;

int idx = 0, idx2 = 0, cmdLen = 0;

cmdLen = strlen(cmd);

while(1){

if(cmd[idx] == ' ' || cmd[idx] == '\0')

break;

idx++;

}

idx2 = idx;

while(1){

if(cmd[idx2] != ' ')

break;

idx2++;

}

if(idx == cmdLen)// if there is no instruction/filname

return -1;

for(i = idx2; i < cmdLen; i++){

name[j] = cmd[i];

j++;

}

return 0;

}

// function to show the number of the opcode mnemonic

void cmdOpcode(char cmd[]) {

opNode\* curr;

int hash;

char instruction[10] = {'\0', }; // to save the mnemonic

if(findName(cmd, instruction) == -1){

printf("Mnemonic is required\n");

return;

}

hash = createHash(instruction);

curr = opList[hash];

while (curr && strncmp(curr->mnemonic, instruction, strlen(instruction))) {

curr = curr->link;

if (!curr)

break;

} // find the node that has the same instruction

if (!curr) {

printf("Invalid Mnemonic\n");

return;

}

else {

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

}

addHistory(cmd);

}

// function that prints opcodelist

void cmdOpcodelist(char cmd[]) {

int i;

int count;

opNode\* curr;

for (i = 0; i < 20; i++) {

printf("%3d : ", i);

curr = opList[i];

count = 0;

while (curr) {

printf("[%s,%x]", curr->mnemonic, curr->opcode);

curr = curr->link;

if (curr)

printf(" -> ");

count++;

}

if (count == 0)

printf("empty");

printf("\n");

}

addHistory(cmd);

}

// function to create symbol hash

int createSHash(char symb[]){

int result;

result = symb[0] - 'A';

return result;

}

// function to get the node that has the matching instruction

opNode\* getFormat(char instruction[]){

int hash;

opNode\* curr;

hash = createHash(instruction);

curr = opList[hash];

while(curr && strncmp(curr->mnemonic, instruction, strlen(instruction))){

curr = curr->link;

if(!curr)

break;

}

return curr;

}

// function to get the Symbol\* that has the matching label

Symbol\* getSymbol(char symb[]){

int hash;

Symbol \*find;

hash = createSHash(symb);

if(hash<0 || hash>25)

hash = 0;

find = symList[hash];

while(find && strncmp(find->symbol, symb, strlen(find->symbol))){

find = find->link;

if(!find)

break;

}

return find;

}

//pass 1 algorithm to assemble file. creates intermediate file for pass 2

int pass1(char cmd[], char filename[]){

FILE \*fp;

FILE \*midFile;

char temp1[50], temp2[50], temp3[50];

char line[150];

int scan\_num;

int i;

int hash;

int format = 0, ret = 1;

int tempsnum = 0;

Symbol \*curr, \*next, \*temp;

start\_address = 0;

LOCCTR = 0;

if(!(fp = fopen(filename, "r"))){

printf("File Not Found\n");

return -1;

}

if(!(midFile = fopen("intermediate", "w"))){

printf("Failed creating intermediate file\n");

return -1;

}

while(fgets(line, 110, fp)){// read 1 line from asm file

memset(temp1, '\0', sizeof(temp1));

memset(temp2, '\0', sizeof(temp2));

memset(temp3, '\0', sizeof(temp3));

format = 0;

scan\_num = sscanf(line, "%s %s %s", temp1, temp2, temp3);

for(i=0;i<strlen(line);i++){ // if comma exists change scan\_num

if(line[i] == ','){

scan\_num--;

break;

}

}

fprintf(midFile, "%4d\t", line\_num \* 5); // print line number

line\_num++;

if(temp1[0] >= '0' && temp1[0] <='9'){ // error when first character is number

printf("Number can't be firt character\n");

ret = -1;

break;

}

// if it is a comment

if(temp1[0] == '.'){

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

fprintf(midFile, "%s", line);

continue;

}

else{ // if label exists

if(scan\_num == 3){

if(strcmp(temp2, "START")){ // search if label exists in SYMTAB

hash = createSHash(temp1);

curr = symList[hash];

while(curr && strcmp(temp1, curr->symbol)) curr = curr->link;

if(curr){ // when it is existing symbol error

printf("Duplicate Symbol\n");

ret = -1;

break;

}

else{

temp = (Symbol \*)malloc(sizeof(Symbol));

strcpy(temp->symbol, temp1);

temp->location = LOCCTR;

temp->link = NULL;

if(!symList[hash]) symList[hash] = temp;

else{

next = symList[hash];

while(next->link) next = next->link;

next->link = temp;

}

tempsnum++;

}

}

if(!strcmp(temp2, "START")){ // when label is START

if(line\_num != 2){ // when first line is not START, error

printf("Start should be the first instruction\n");

ret = -1;

break;

}

fprintf(midFile, "%4d\t", LOCCTR);

fprintf(midFile, "%s", line);

start\_address = hextodec(temp3,0,strlen(temp3));

if(errFlag == TRUE){ // when invaild starting address, error

printf("Invalid start address\n");

errFlag = FALSE;

ret = -1;

break;

}

LOCCTR = start\_address; // adjust LOCCTR to start address

continue;

}//adjust LOCCTR

if(!strcmp(temp2,"WORD")){ // when WORD, +3 to LOCCTR

fprintf(midFile, "%4d\t", LOCCTR);

fprintf(midFile, "%s", line);

LOCCTR += 3;

continue;

}

else if(!strcmp(temp2,"RESW")){ // when RESW +3 \* operand to LOCCTR

fprintf(midFile, "%4d\t", LOCCTR);

fprintf(midFile, "%s", line);

LOCCTR += 3 \* atoi(temp3);

continue;

}

else if(!strcmp(temp2,"RESB")){ // when RESB +operand to LOCCTR

fprintf(midFile, "%4d\t", LOCCTR);

fprintf(midFile, "%s", line);

LOCCTR += atoi(temp3);

continue;

}

else if(!strcmp(temp2,"BYTE")){ // when BYTE

fprintf(midFile, "%4d\t", LOCCTR);

fprintf(midFile, "%s", line);

if(temp3[0] == 'C')

LOCCTR += strlen(temp3) - 3;

else if (temp3[0] == 'X') {

if((int)((strlen(temp3) - 3) % 2) == 0)

LOCCTR += (int)(strlen(temp3) - 3) / 2;

else

LOCCTR += (int)(strlen(temp3) - 3) / 2 + 1;

}

continue;

}

else{

if(temp2[0] == '+'){ // format 4

if(!getFormat(temp2+1)){ // when opcode doesn't exist error

printf("Invalid Mnemonic\n");

ret = -1;

break;

}

fprintf(midFile, "%4d\t", LOCCTR);

fprintf(midFile, "%s", line);

LOCCTR += 4;

continue;

}

else{

//format = atoi(getFormat(temp2)->format);

if(!getFormat(temp2)){ // when opcode doesn't exist, error

printf("Invalid Mnemonic\n");

ret = -1;

break;

}

else{ // add format to LOCCTR

format = atoi(getFormat(temp2)->format);

fprintf(midFile, "%4d\t", LOCCTR);

fprintf(midFile, "%s", line);

LOCCTR += format;

continue;

}

}

}

}

else if(!strcmp(temp1, "BASE")){ //when lable doesn't exist, mark with '!', when loc isn't needed mark with -1

fprintf(midFile, "%4d\t",-1);

fprintf(midFile, "!");

fprintf(midFile, "%s",line);

continue;

}

else{//when lable doesn't exist, mark with '!', when loc isn't needed mark with -1

if(!strcmp(temp2, "START") && temp3[0] == '\0'){ // when START doesn't have an operand, ERROR

printf("No starting address\n");

ret = -1;

break;

}

else if(!strcmp(temp1, "END")){

fprintf(midFile, "%4d\t",-1);

fprintf(midFile, "!");

fprintf(midFile, "%s", line);

break;

}

else if(temp1[0] == '+'){ // format 4

if(!getFormat(temp1+1)){// when opcode doens't exist, error

printf("Invalid Mnemonic\n");

ret = -1;

break;

}

fprintf(midFile, "%4d\t", LOCCTR);

fprintf(midFile, "!");

fprintf(midFile, "%s",line);

LOCCTR += 4;

continue;

}

else{ // when opcode doesn't exist

//format = atoi(getFormat(temp1)->format);

if(!getFormat(temp1)){

printf("Invalid Mnemonic\n");

ret = -1;

break;

}

else{

format = atoi(getFormat(temp1)->format);

fprintf(midFile, "%4d\t", LOCCTR);

fprintf(midFile,"!%s", line);

LOCCTR += format;

continue;

}

}

}

}

}

fclose(fp);

fclose(midFile);

if(ret != -1) // when not error save Symnum

symNum = tempsnum;

return ret;

}

// function to print .lst file with given format

void printlst(FILE \* lst, int num, int loc, char label[], char opcode[], char operand[], char address[]){

fprintf(lst, "%4d\t", num);

if(loc != -1)

fprintf(lst, "%04X\t", loc);

else

fprintf(lst, "%-4s\t", "");

if(!strcmp(label, "!"))

fprintf(lst, "%-6s ", "");

else

fprintf(lst, "%-6s ", label);

fprintf(lst, "%-6s ", opcode);

fprintf(lst, "%-10s ", operand);

fprintf(lst, "%-10s\n", address);

}

// function to create object code list for printing .obj file

void createobj(char TA[], int loc, int mf){

Object\* obj;

obj = (Object \*)malloc(sizeof(Object));

strcpy(obj->objcode, TA);

obj->loc = loc;

obj->eflag = enterflag;

obj->mflag = mf;

if(!ohead){

ohead = obj;

otail = obj;

}

else{

otail->link = obj;

otail = otail->link;

}

enterflag = 0;

}

// function for pass 2 algorithm creates .lst .obj file with intermediate file

int pass2(char filename[]){

FILE \*midFile, \*lst, \*obj; // file pointer for creating obj, lst file

int num, loc; // to read line number, and location

char label[50], opcode[50], operand[50]; // array for saving lable, opcode operand

char line[150];

char rmoperand[50], rmopcode[50]; // array for saving operand, opcode without prefix

char TA[10], tempstr[100] = {'\0',}; //array for saving target address and object code

char reg[][3] = {"A", "X", "L", "B", "S", "T", "F", "0", "PC", "SW"}; // register array

int N, I, X, B, P, E, xbpe, PC = 0; // value of NIXBPE to check the addressing mode, and program counter

char fname[50] = {'\0',}, fname2[50] = {'\0',}; // array for filename

int i, cidx; // loop variable and index of comma

int mf;

int ret = 1;

int format = 0, BASE = 0, disp = 0;

int startFlag = 0, ENDFLAG = 0;

Symbol\* curr;

opNode\* node;

Object\* obnode;

for(i = 0; filename[i] != '.'; i++)

fname[i] = filename[i];

strcpy(fname2, fname);

strcat(fname, ".lst");

strcat(fname2, ".obj");

if(!(midFile = fopen("intermediate", "r"))){

printf("Failed opening intermediate file\n");

return -1;

}

if(!(lst = fopen(fname, "w"))){

printf("Failed creating lst file\n");

return -1;

}

if(!(obj = fopen(fname2, "w"))){

printf("Failed creating obj file\n");

return -1;

}

PC = start\_address;

while(fgets(line, 150, midFile)){

num = 0;

loc = 0;

N = I = X = B = E = mf = 0;

P = 1;

i = cidx = disp = xbpe = 0;

memset(label, '\0', sizeof(label));

memset(opcode, '\0', sizeof(opcode));

memset(operand, '\0', sizeof(operand));

memset(rmoperand, '\0', sizeof(rmoperand));

memset(rmopcode, '\0', sizeof(rmopcode));

memset(TA, '\0', sizeof(TA));

sscanf(line, "%d %d %s %s %[^\n]", &num, &loc, label, opcode, operand);

//printf("%d %d %s %s %s\n", num,loc,label,opcode,operand);

line\_num2 = num;

if(!strcmp(opcode, "END")){

//write last listing line

ENDFLAG = 1;

printlst(lst, num, loc, label, opcode, operand, "");

break;

}

if(!strcmp(opcode, "START")){

// write header record (H)

// initialize first text record

printlst(lst, num, loc, label, opcode, operand, "");

fprintf(obj,"H%-6s%06X%06X\n",label,loc,LOCCTR-loc);

startFlag = 1;

continue;

}

if(opcode[0] == '\0'){

//when comment

fprintf(lst, "%s", line);

continue;

}

else{

//not comment

//opcode should exist(checked in pass 1)

if(!strcmp(opcode, "BASE")){ // when BASE

if(!(curr=getSymbol(operand))){ // if it doesn't exist in SYMTAB error

printf("Undefiend symbol\n");

ret = -1;

break;

}

else{

BASE = curr->location; //set base

printlst(lst, num, loc, label, opcode, operand, "");

}

continue;

}

switch(operand[0]){// check if operand has prefix

case '#': // immediate addressing

N = 0;

I = 1;

B = 0;

if(operand[1] >= '0' && operand[1] <= '9')

P = 0;

strcpy(rmoperand, operand + 1);

break;

case '@': // indirect addressing

N = 1;

I = 0;

strcpy(rmoperand, operand + 1);

break;

default : // simple addressing

strcpy(rmoperand, operand);

N = 1;

I = 1;

break;

}

if(opcode[0] == '+'){ // when format 4

node = getFormat(opcode + 1);

format = atoi(node->format);

if(format != 3){ // when + is used for wrong opcode

printf("+ should be used for 3/4 format\n");

ret = -1;

break;

}

format = 4;

B = 0;

P = 0;

E = 1;

}

else {

node = getFormat(opcode);

if(node)

format = atoi(node->format);

}

//PC update

if(!strcmp(opcode, "BYTE")){

if(operand[0] == 'C'){

PC += strlen(operand) - 3;

for(i = 2; i <= strlen(operand) - 2; i++){

sprintf(TA + (2\*(i-2)), "%X",(int)operand[i]);

}

}

else if (operand[0] == 'X'){

if ((int)(strlen(operand) - 3) % 2 == 0) {

PC += (strlen(operand) - 3) / 2;

for (i = 2; i <= strlen(operand) - 2; i++)

TA[i - 2] = operand[i];

}

else {

PC += (int)(strlen(operand) - 3) / 2 + 1;

TA[0] = '0';

for (i = 2; i <= strlen(operand) - 2; i++)

TA[i - 1] = operand[i];

}

}

printlst(lst, num, loc, label, opcode, operand, TA);

createobj(TA, loc, 0);

continue;

}

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

PC += 3;

sprintf(TA, "%06X", atoi(operand));

printlst(lst, num, loc, label, opcode, operand, TA);

createobj(TA, loc, 0);

continue;

}

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

PC += atoi(rmoperand);

printlst(lst, num, loc, label, opcode, operand, "");

enterflag = 1; //there needs to be an enter in obj file

continue;

}

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

PC += atoi(rmoperand) \* 3;

printlst(lst, num, loc, label, opcode, operand, "");

enterflag = 1; //enter is needed in obj file

continue;

}

else

PC += format;

for(i = 0; i < strlen(operand); i++){ // check for comma in operand

if(operand[i] == ','){

cidx = i;

break;

}

}

if(format == 1){ // when format 1

if(operand[0] != '\0'){ // format 1 can't have operand

printf("Operand is not allowed for format 1\n");

ret = -1;

break;

}

sprintf(TA, "%X", node->opcode);

strcat(TA,"0000");

}

else if(format == 2){ // when format 2

sprintf(TA, "%X", node->opcode);

if(cidx!=0){ // if there are 2 register

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

if(!strncmp(reg[i], operand, cidx))

break;

}

if(i == 10 || i == 7){ // check if valid register

printf("Unidentified register %s\n",operand);

ret = -1;

break;

}

sprintf(TA+2, "%d",i);

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

if(!strncmp(reg[i], operand + cidx + 2, strlen(reg[i])))

break;

}

if(i == 10 || i == 7){ // check if valid register

printf("Unidentified register %s\n",operand+cidx+2);

ret = -1;

break;

}

sprintf(TA+3,"%d", i);

}

else{ // 1 register

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

if(!strncmp(reg[i], operand, strlen(reg[i])))

break;

}

if(i == 10 || i == 7){//check if valid register

printf("Unidentified register %s\n",operand);

ret = -1;

break;

}

sprintf(TA+2, "%d",i);

TA[3] = '0';

}

}

else if(format == 3){ // format 3

sprintf(TA, "%02X", (node->opcode) + (2 \* N + I));

if(cidx != 0){ // if there is a register

if(!strncmp(operand + cidx + 2, "X", strlen("X"))) // if register is X set X = 1

X = 1;

else{

printf("Invalid Register with BUFFER\n");

ret = -1;

break;

}

}

if(!(curr = getSymbol(rmoperand))){ //if there is a symbol match

sprintf(TA+2, "%04X", atoi(rmoperand));

printlst(lst, num, loc, label, opcode, operand, TA);

createobj(TA, loc, 0);

continue;

}

else{ // check if PC relative or base relative

disp = curr->location - PC;

if(disp < -2048 || disp > 2047){

disp = curr->location - BASE;

B = 1;

P = 0;

if(disp<0 || disp>4095){

printf("Displacement out of range\n");

ret = -1;

break;

}

}

if(disp < 0)

disp = disp & 4095;

}

xbpe = (int)(pow(2,3))\*X+(int)(pow(2,2))\*B+(int)(pow(2,1))\*P+E;

sprintf(TA+2,"%04X",(int)(pow(16,3))\*xbpe + disp);

}

else if(format ==4){ // format 4

sprintf(TA, "%02X", (node->opcode) + (2 \* N + I));

xbpe = (int)(pow(2,3))\*X+(int)(pow(2,2))\*B+(int)(pow(2,1))\*P+E;

curr = getSymbol(rmoperand);

if(curr){

sprintf(TA+2, "%06X", (int)(pow(16,5))\*xbpe + getSymbol(rmoperand)->location);

mf = 1;

}

else

sprintf(TA+2, "%06X", (int)(pow(16,5))\*xbpe + atoi(rmoperand));

}

printlst(lst, num, loc, label, opcode, operand, TA);

//printf("%d %X %s %s %s %s\n",num, loc, label, opcode ,operand, TA);

createobj(TA, loc, mf);

continue;

//if opcode exists

}

}

if((ENDFLAG == 0 || startFlag == 0) && ret != -1){// if there is no START or END error

printf("Code should have START/END\n");

ret = -1;

}

if(ret != -1){ //when not error write .obj file

//printf("ob");

obnode = ohead;

while(obnode){ // print text record

fprintf(obj,"T%06X", obnode->loc);

while(((int)(strlen(tempstr) + strlen(obnode->objcode))/2) <= 29){

strcat(tempstr,obnode->objcode);

obnode = obnode->link;

if(!obnode)

break;

if(obnode->eflag == 1)

break;

}

fprintf(obj, "%02X", (int)strlen(tempstr)/2);

fprintf(obj, "%s\n", tempstr);

memset(tempstr, '\0', sizeof(tempstr));

}

obnode = ohead;

while(obnode){ // print modification record

if(obnode->mflag == 1){

fprintf(obj,"M%06X05\n",obnode->loc+1);

}

obnode = obnode->link;

}

fprintf(obj, "E%06X\n", ohead->loc);

}

fclose(midFile);

fclose(lst);

fclose(obj);

if(ret == -1){

while(ohead){

obnode = ohead;

ohead = ohead->link;

free(obnode);

}

objfree = 1;

remove(fname);

remove(fname2);

}//remove .lst, .obj file and obj list when error

return ret;

}

// compare to make alphabetical order

int compare(const void \*a, const void \*b) {

return (strcmp(((Symbol \*)a)->symbol, ((Symbol \*)b)->symbol));

}

// call pass 1 and pass 2 and initialize

void cmdAssemble(char cmd[]){

char filename[100] = {'\0',};

int i , j = 0;

Symbol \*curr;

Object\* node;

line\_num = 1;

line\_num2 = 0;

//symNum = 0;

//freeFlag = 0;

objfree = 0;

for(i = 0; i < 26; i++){

while(symList[i]){

curr = symList[i];

symList[i] = symList[i] -> link;

free(curr);

}

}

while(ohead){

node = ohead;

ohead = ohead->link;

free(node);

}

if(findName(cmd,filename) == -1){

printf("Filename required\n");

return;

}

if(pass1(cmd,filename)!=1){

printf("PASS1 Error at line : %d\n",(line\_num - 1) \* 5);

return;

}

if(pass2(filename)==1){

printf("\x1b[32mSuccessfully ");

printf("\x1b[0massemble %s.\n",filename);

if (saveSym != NULL) {

free(saveSym);

}

//save the SYMTAB of the last successful assmeble

if (symNum != 0) { // create a list that saves the value of symbols

saveSym = (Symbol \*)malloc(sizeof(Symbol) \* symNum);

for (i = 0; i < 26; i++) {

curr = symList[i];

while (curr) {

strcpy(saveSym[j].symbol, curr->symbol);

saveSym[j].location = curr->location;

j++;

curr = curr->link;

}

}

qsort(saveSym, symNum, sizeof(Symbol), compare); // sort the contents

}

}

else{

printf("PASS2 Error at line : %d\n", line\_num2);

}

addHistory(cmd);

return;

}

// function for command Type

void cmdType(char cmd[]){

FILE \*fp;

char c;

char filename[100] = {'\0',};

DIR \*currDir;

struct dirent \*currFile;

struct stat currStat;

int dirFlag = 0;

if(findName(cmd, filename)==-1){

printf("Filename required\n");

return;

}

if((currDir = opendir("."))){ // error when directory is input

while((currFile = readdir(currDir))){

stat(currFile->d\_name, &currStat);

if(!strncmp(currFile->d\_name, filename, sizeof(filename))){

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

dirFlag = 1;

}

}

closedir(currDir);

}

if(dirFlag == 1){

printf("Directory name cannot be an input\n");

return;

}

if((fp = fopen(filename, "r"))!=NULL){ // print the contents of the file on screen

while(~(c = fgetc(fp)))

putchar(c);

fclose(fp);

}

else{

printf("File Not Found\n");

return;

}

addHistory(cmd);

}

// function for command symbol

void cmdSymbol(char cmd[]){

int i;

//print saved symbol

for(i=0;i<symNum;i++)

printf("\t%s\t%04X\n",saveSym[i].symbol, saveSym[i].location);

if(symNum == 0)

printf("No symbol in SYMTAB\n");

addHistory(cmd);

}

//function for command Progaddr

void cmdProgaddr(char cmd[]){

int idx;

int i;

int cmdLen;

cmdLen = strlen(cmd);

if(cmdLen == 8){ // if there is no address Error

printf("Address required for progaddr\n");

return;

}

for (i = 8; i < cmdLen; i++){

if(cmd[i] != ' '){

idx = i;

break;

}

}

progaddr = hextodec(cmd, idx, cmdLen);

if(errFlag == TRUE || progaddr > 0xFFFFF || progaddr < 0){ // calculate address to hex and set progaddr

printf("Invalid program address\n");

errFlag = FALSE;

progaddr = 0;

return;

}

addHistory(cmd);

}

//function for printing loadmap

void printloadmap(){

int i;

Exsym \*curr;

int total\_length = 0;

printf("control symbol address length\n");

printf("section name\n");

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

for(i = 0; i < 20; i++){

curr = estab[i];

while(curr){

if(curr->length != 0){

printf("%-6s%7s %04X %04X\n", curr->symbol, " ", curr->csaddr, curr->length);

total\_length += curr->length;

}

else

printf("%6s%7s %04X\n", " ", curr->symbol, curr->csaddr);

curr = curr->link;

}

}

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

printf("\t total length %04X\n", total\_length);

}

//pass1 -> assign address to external symbol

int loadpass1(int count, char f1[][30]){

FILE\* fp;

int i, j, hash, linenum;

unsigned int csaddr, cslen, loc;

char filename[30], sym[7];

char line[150];

Exsym \*curr, \*node;

csaddr = progaddr;

for(i = 0; i < 20; i++){

while(estab[i]){

curr = estab[i];

estab[i] = estab[i]->link;

free(curr);

}

}

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

linenum = 1;

strcpy(filename, f1[i]);

if(!(fp = fopen(filename, "r"))){

printf("File %s does not exist\n", filename);

return -1;

}

fgets(line, 150, fp);

sscanf(line, "%\*c%6s%\*6x%6x", sym, &cslen); //get the symbol and length of cs at the first line

hash = createHash(sym);

curr = estab[hash];

while(curr && strcmp(curr->symbol, sym)) //search for duplicate symbols

curr = curr->link;

if(curr){

printf("Duplicate symbol %s at line %d in %s\n", sym, linenum, filename);

return -1;

}

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

strcpy(node->symbol, sym);

node->csaddr = csaddr;

node->length = cslen;

node->link = NULL;

if(estab[hash]){ //if there is no duplicate symbol, add it to the external symbol tab

curr = estab[hash];

while(curr->link)

curr = curr->link;

curr->link = node;

}

else

estab[hash] = node;

while(line[0] != 'E'){

memset(line, '\0', sizeof(line));

fgets(line, 150, fp);

linenum++;

if(line[0] != 'D') //only read line with D record

continue;

for(j = 1; j < strlen(line)-1; j += 12){

sscanf(line + j, "%6s%6x", sym, &loc);

hash = createHash(sym);

curr = estab[hash];

while(curr && strcmp(curr->symbol, sym)) //check for duplicate symbols

curr = curr->link;

if(curr){

printf("Duplicated symbol %s at line %d in %s\n", sym, linenum, filename);

return -1;

}

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

strcpy(node->symbol, sym);

node->csaddr = loc + csaddr;

node->length = 0;

node->link = NULL;

if(estab[hash]){ //add the external symbol if there is no duplicate

curr = estab[hash];

while(curr->link)

curr = curr->link;

curr->link = node;

}

else

estab[hash] = node;

}

}

csaddr += cslen;

fclose(fp);

}

file\_len = csaddr - progaddr; //save the file length

return 1;

}

//pass2 -> link and load

int loadpass2(int count, char f1[][30]){

FILE\* fp;

int i, j, linenum, hash;

int objcode;

unsigned int addr, tlen;

unsigned int csaddr, cslen, rnum, refer[30] = {0,}, hb, maddr;

char filename[30], sym[7];

char line[150], sign;

Exsym \*curr;

csaddr = progaddr;

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

linenum = 1;

refer[0] = 0;

strcpy(filename, f1[i]);

if(!(fp = fopen(filename, "r"))){

printf("File %s does not exist\n", filename);

return -1;

}

fgets(line, 150, fp);

sscanf(line, "%\*c%6s%\*6x%6x", sym, &cslen);

refer[1] = csaddr;

while(line[0] != 'E'){ //until the end record

memset(line, '\0', sizeof(line));

fgets(line, 150, fp);

linenum++;

if(line[0] == 'R'){//check reference number

refer[0] = 1;

for(j = 1; j < strlen(line)-1; j+=8){

sscanf(line+j, "%2X%6s", &rnum, sym);

hash = createHash(sym);

curr = estab[hash];

while(curr && strcmp(curr->symbol, sym))

curr = curr->link;

if(!curr){ //if there is a symbol undefined, error

printf("In %s line %d undefined symbol %s\n", filename, linenum, sym);

return -1;

}

refer[rnum] = curr->csaddr; //give the reference number the actual address of the symbol

}

}

else if(line[0] == 'T'){ //read text record and save it in memory

sscanf(line+1, "%6x%2x", &addr, &tlen);

for(j = 0; j < tlen; j++){

sscanf(line+9+j\*2, "%2X", &objcode);

memory[j+csaddr+addr] = (unsigned char)objcode;

}

}

else if(line[0] == 'M'){//read modification record

sscanf(line+1, "%6x%2x%c%2x", &addr, &hb, &sign, &rnum);

maddr = memory[csaddr + addr] % (hb % 2 ? 0x10 : 0x100); //check if the length of the half byte is 5 / 6

for(j = 1; j < (hb + 1) / 2; j++){//merge the saved values in the memory to modify

maddr \*= 0x100;

maddr += memory[csaddr + addr + j];

}

maddr += (sign == '-' ? -1 : 1) \* refer[rnum];

for(j = (hb -1)/2; j > 0; j--){ //save the modified value back to memory

memory[csaddr + addr + j] = maddr % 0x100;

maddr /= 0x100;

}

if(hb % 2 == 1) //if half byte == 5, modify half byte

memory[csaddr + addr] = (memory[csaddr + addr]&((1<<8)-(1<<4))) + maddr;

else

memory[csaddr + addr] = maddr;

}

}

csaddr += cslen;

fclose(fp);

}

end\_exec = progaddr;

return 1;

}

//when comand loader is input

void cmdLoader(char cmd[]){

int count = 0;

int i, j = 0;

int cmdLen;

int pass1, pass2;

char f1[3][30] = {'\0'};

cmdLen = strlen(cmd);

if(cmdLen == 6){ //if there is no filename error

printf("Filename required for loader\n");

return;

}

//check the number of files and save each filename

for (i = 6; i < cmdLen; i++){

if(cmd[i] != ' '){

count++;

break;

}

}

for(j = 0; i < cmdLen; i++){

if(cmd[i] == ' ')

break;

f1[0][j++] = cmd[i];

}

for (; i < cmdLen;i++){

if(cmd[i] != ' '){

count++;

break;

}

}

for(j = 0; i < cmdLen; i++){

if(cmd[i] == ' ')

break;

f1[1][j++] = cmd[i];

}

for (; i < cmdLen; i++){

if(cmd[i] != ' '){

count++;

break;

}

}

for(j = 0; i < cmdLen; i++){

f1[2][j++] = cmd[i];

}

//if pass1 and pass2 is successful, print loadmap and add command to history

pass1 = loadpass1(count, f1);

if(pass1 < 0){

printf("[\x1b[31mError\x1b[0m] Pass1\n");

return;

}

pass2 = loadpass2(count ,f1);

if(pass2 < 0){

printf("[\x1b[31mError\x1b[0m] Pass2\n");

return;

}

loadflag = 1;

printloadmap();

addHistory(cmd);

}

//function for command bp

void cmdBp(char cmd[]){

int i, cmdlen;

int idx, bp;

cmdlen = strlen(cmd);

for(i=2;i<cmdlen;i++){

if(cmd[i] != ' '){

idx = i;

break;

}

}

if(strlen(cmd) == 2){ //when command == bp, print all the breakpoints

printf("\tbreakpoint\n");

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

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

printf("\t%X\n", bpset[i]);

}

}

else if(!strcmp(cmd+idx, "clear")){ //when command == bp clear, remove all the breakpoints

memset(bpset, 0, sizeof(bpset));

memset(bptab, 0, sizeof(bptab));

bpnum = 0;

bpflag = -1;

printf("\t[\x1b[32mok\x1b[0m] clear all breakpoints\n");

}

else{ //when address is input

bp = hextodec(cmd,idx,cmdlen);

if(errFlag == TRUE || bp > 0xFFFFF || bp < 0){//if address is out of range, error

printf("Invalid Breakpoint\n");

errFlag = FALSE;

return;

}

if(bptab[bp] == 1){ //when the breakpoint already exist, error

printf("Existing break point at %d\n", bp);

return;

}

//add bp to bpset, bptab

printf("\t[\x1b[32mok\x1b[0m] create breakpoint %X\n", bp);

bpset[bpnum] = bp;

bptab[bp] = 1;

bpnum++;

}

addHistory(cmd);

}

//function for printing values in the registers

void printReg(int reg[]){

printf("A : %06X X : %06X\n", reg[0], reg[1]);

printf("L : %06X PC : %06X\n", reg[2], reg[8]);

printf("B : %06X S : %06X\n", reg[3], reg[4]);

printf("T : %06X\n",reg[5]);

}

//function for getting the target address of from objcode in memory(only used in format 3,4)

int getTarget(int curr, int reg[], int nixbpe){

enum registers {A,X,L,B,S,T,F,NA,PC,SW};

int target, format, disp;

format = 3 + (nixbpe & 1);

reg[PC] += format; // get format

if(format == 4){ //when format 4

target = (((memory[curr+1] & 0x0F) << 16) | (memory[curr+2] << 8) | (memory[curr+3]));

}

else if(nixbpe & 6){ // base, pc relative

disp = (((memory[curr+1] & 0x0F) << 8) | memory[curr+2]);

if(disp & (1<<11)) // if negative number

disp = -((~disp + 1) & 0xFFF);

if(nixbpe & 4)

target = reg[B] + disp;

else

target = reg[PC] + disp;

}

else{

target = (((memory[curr+1] & 0x0F) << 8) | memory[curr+2]);

}

if(nixbpe & 8){ // index

target += reg[X];

}

if((nixbpe & 0x30) == 0x20) //when indirect addressing, return the address that the target address is pointing to

target = ((memory[target] << 16) | (memory[target+1] << 8) | memory[target+2]);

return target;

}

//function for calculating the value from the address

int calcVal(int TA, int nixbpe){

int result;

nixbpe = nixbpe & 0x30; //check for n,i flags

if(nixbpe == 0x10) // immediate addressing

result = TA;

else{//Indirect, simple addressing

result = ((memory[TA] << 16) | (memory[TA+1] << 8) | memory[TA+2]);

}

return result;

}

//function for command run

void cmdRun(char cmd[]){

enum registers {A,X,L,B,S,T,F,NA,PC,SW};

int nixbpe;

int curr, end, target;

int opcode, value;

int regnum, regnum2;

int i;

if(strlen(cmd) != 3){

printf("Invalid command\n");

return;

}

if(loadflag == 0){

printf("Loaded program doesn't exist\n");

return;

}

//set the initail values

reg[PC] = end\_exec;

curr = end\_exec;

end = progaddr + file\_len;

if(bpflag == -1)//L register shouldn't be modified when program is run

reg[L] = end;

if(bptab[0] == 1 && bpflag == -1){ //when pc points to the breakpoint print registers

bpflag = 1;

printReg(reg);

printf("\tStop at checkpoint [%X]\n",curr);

addHistory(cmd);

return;

}

while(reg[PC]!= end){

opcode = memory[curr] & 0xFC; //find the opcode from the first 6bits

nixbpe = ((memory[curr] & 0x03) << 4) | ((memory[curr+1] & 0xF0) >> 4); //find the nixbpe from the next 6bits

if(opcode == 0x0C || opcode == 0x78 || opcode == 0x54 || opcode == 0x14 || opcode == 0x10){ // STA, STB, STCH, STL, STX

target = getTarget(curr, reg, nixbpe);

if(opcode == 0x54){ //when STCH

memory[curr] = (unsigned char)(reg[A] & 0xF);

}

else{

if(opcode == 0x0C) //STA

regnum = A;

else if(opcode == 0x78) //STB

regnum = B;

else if(opcode == 0x14) //STL

regnum = L;

else if(opcode == 0x10) //STX

regnum = X;

for(i = 0; i < 3; i++) //store the value of the register to memory

memory[target + i] = (unsigned char)((reg[regnum]>>(2-i)\*8)&0xFF);

}

}

else if(opcode == 0x00 || opcode == 0x68 || opcode == 0x50 || opcode == 0x74 || opcode == 0x08 || opcode == 0x04 ){ //LDA, LDB, LDCH, LDT, LDL

target = getTarget(curr, reg, nixbpe);

value = calcVal(target, nixbpe);

if(opcode == 0x50){

reg[A] = ((reg[A] & 0xFFFF00) | (value >> 16));

} //LDCH

else{

if(opcode == 0x00) //LDA

regnum = A;

else if(opcode == 0x68) //LDB

regnum = B;

else if(opcode == 0x74) //LDT

regnum = T;

else if(opcode == 0x08) //LDL

regnum = L;

else if(opcode == 0x04) //LDX

regnum = X;

reg[regnum] = value; //save the value to the register

}

}

else if(opcode == 0x48){ // JSUB

target = getTarget(curr, reg, nixbpe);

reg[L] = reg[PC];

reg[PC] = target;

}

else if(opcode == 0x28){ // COMP

target = getTarget(curr, reg, nixbpe);

value = calcVal(target, nixbpe);

if( reg[A] > value)

reg[SW] = '>';

else if (reg[A] < value)

reg[SW] = '<';

else

reg[SW] = '=';

}

else if(opcode == 0x3C || opcode == 0x30 || opcode == 0X38 ){ // J, JEQ, JLT

target = getTarget(curr, reg, nixbpe);

if(opcode == 0x3C)

reg[PC] = target;

else if(opcode == 0x30 && reg[SW] == '=')

reg[PC] = target;

else if(opcode == 0x38 && reg[SW] == '<')

reg[PC] = target;

}

else if(opcode == 0xB4){ // CLEAR

reg[PC] += 2;

regnum = (memory[curr + 1] & 0xF0) >> 4;

reg[regnum] = 0;

}

else if(opcode == 0xE0 || opcode == 0xDC || opcode == 0xD8){ // TD, WD, RD

reg[PC] += (3 + ((memory[curr+1] & 0x10) >> 4));

if(opcode == 0xE0)

reg[SW] = '<';

}

else if(opcode == 0xA0){ // COMPR

reg[PC] += 2;

regnum = ((memory[curr + 1] & 0xF0) >> 4);

regnum2 = (memory[curr + 1] & 0x0F);

if(reg[regnum] > reg[regnum2])

reg[SW] = '>';

else if(reg[regnum] < reg[regnum2])

reg[SW] = '<';

else

reg[SW] = '=';

}

else if(opcode == 0x4C){ // RSUB

reg[PC] = reg[L];

}

else if(opcode == 0x2C){ // TIX

target = getTarget(curr, reg, nixbpe);

value = calcVal(target, nixbpe);

reg[X] += 1;

if(reg[X] > value)

reg[SW] = '>';

else if(reg[X] < value)

reg[SW] = '<';

else

reg[SW] = '=';

}

else if(opcode == 0xB8){ // TIXR

reg[PC] += 2;

regnum = ((memory[curr+1] & 0xF0) >> 4);

reg[X] += 1;

if(reg[X] > reg[regnum])

reg[SW] = '>';

else if(reg[X] < reg[regnum])

reg[SW] = '<';

else

reg[SW] = '=';

}

curr = reg[PC];

if(bptab[curr] == 1){ //when pc points to the breakpoint print registers

bpflag = 1;

end\_exec = curr;

printReg(reg);

printf("\tStop at checkpoint [%X]\n",curr);

addHistory(cmd);

return;

}

}

//program ends, reset values

end\_exec = progaddr;

bpflag = -1;

printReg(reg);

memset(reg, 0, sizeof(reg));

printf("\tEnd Program\n");

addHistory(cmd);

return;

}

* 1. 20161643.c

1. #include "20161643.h"
2. int main() {
3. char command[100] = {'\0',}; // array to save command
4. char trimCmd[100] = {'\0',}; // array to save trimmed command
5. int cmdNum; // variable to save command number
6. History\* temp; // variable to free history link
7. int i;
8. opNode \*curr, \*prev;
9. Symbol \*sym;
10. Object \*node;
11. Exsym \*tmp;
12. funcPtr fPtr[] = { cmdHelp, cmdDir, cmdQuit, cmdHistory, cmdDump, cmdEdit, cmdFill, cmdReset, cmdOpcode, cmdOpcodelist, cmdAssemble, cmdType, cmdSymbol, cmdProgaddr, cmdLoader, cmdRun, cmdBp}; // function pointer to reduce the code
13. loadOpcode(); // load opcode list from opcode.txt
14. while (!endFlag) {
15. memset(command, '\0', sizeof(command));
16. memset(trimCmd, '\0', sizeof(trimCmd));
17. printf("sicsim>");
18. fgets(command, sizeof(command), stdin);
19. command[strlen(command) - 1] = '\0';
20. cmdNum = checkCmd(command, trimCmd); // check the command number
21. if (cmdNum == -1)
22. printf("Invalid Command\n");
23. else {
24. fPtr[cmdNum](trimCmd); // call the function that has the right command number
25. }
26. }
27. while(head){ // free history list
28. temp = head;
29. head = head->link;
30. free(temp);
31. }
32. for(i=0; i < 20; i++){ // free opcode list
33. curr = opList[i];
34. while(curr){
35. prev = curr;
36. curr = curr->link;
37. free(prev);
38. }
39. }
40. for (i = 0; i < 26; i++) { // free symbol list
41. while (symList[i]) {
42. sym = symList[i];
43. symList[i] = symList[i]->link;
44. free(sym);
45. }
46. }
47. for (i = 0; i < 20; i++){
48. while(estab[i]){
49. tmp = estab[i];
50. estab[i] = estab[i]->link;
51. free(tmp);
52. }
53. }
54. if(saveSym != NULL) // free saved symbol
55. free(saveSym);
56. if(objfree != 1){
57. while (ohead) { // free object list
58. node = ohead;
59. ohead = ohead->link;
60. free(node);
61. }
62. }
63. return 0;
64. }