**과목명: 시스템프로그래밍**

**1 분반**

**<<Project #2>>**

**서강대학교 컴퓨터공학과**

**[학번] 20161643**

**[이름] 정진원**

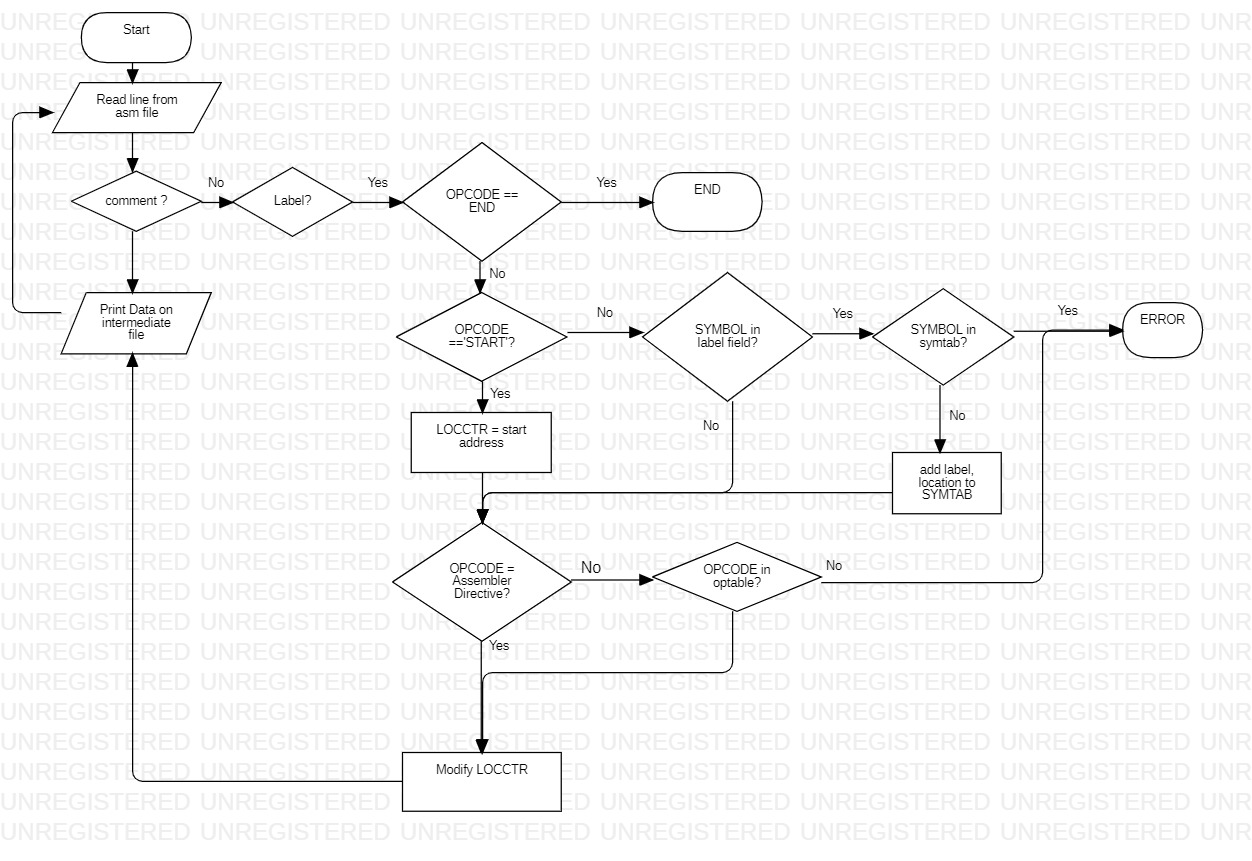
목 차

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
3. **모듈 정의**
   1. 모듈이름: int findName(char cmd[], char name[])
      1. 기능
      2. 사용변수
   2. 모듈이름: int createSHash(char symb[])
      1. 기능
      2. 사용변수
   3. 모듈이름: opNode\* getFormat(char instruction[])
      1. 기능
      2. 사용변수
   4. 모듈이름: Symbol\* getSymbol(char symb[])
      1. 기능
      2. 사용변수
   5. 모듈이름: int pass1(char cmd[],char filename[])
      1. 기능
      2. 사용변수
   6. 모듈이름: void printlst(FILE \* lst, int num, int loc, char label[], char opcode[], char operand[], char address[])
      1. 기능
      2. 사용변수
   7. 모듈이름: void createobj(char TA[], int loc, int mf)
      1. 기능
      2. 사용변수
   8. 모듈이름: int pass2(char filename[])
      1. 기능
      2. 사용변수
   9. 모듈이름: int compare(const void \*a, const void \*b)
      1. 기능
      2. 사용변수
   10. 모듈이름: void cmdAssemble(char cmd[])
       1. 기능
       2. 사용변수
   11. 모듈이름: void cmdType(char cmd[])
       1. 기능
       2. 사용변수
   12. 모듈이름: void cmdSymbol(char cmd[])
       1. 기능
       2. 사용변수
4. **전역 변수, 구조체, typedef 정의**
   1. typedef struct \_Symbol
   2. typedef struct \_Object
   3. int line\_num
   4. int line\_num2
   5. int start\_address
   6. Symbol\* symList[26]
   7. int LOCCTR
   8. int symNum
   9. int enterflag
   10. Object \*ohead
   11. int objfree
   12. Symbol \*saveSym
5. **코드 설명**
   1. 20161643.h
   2. 20161643.c
6. **프로그램 개요**

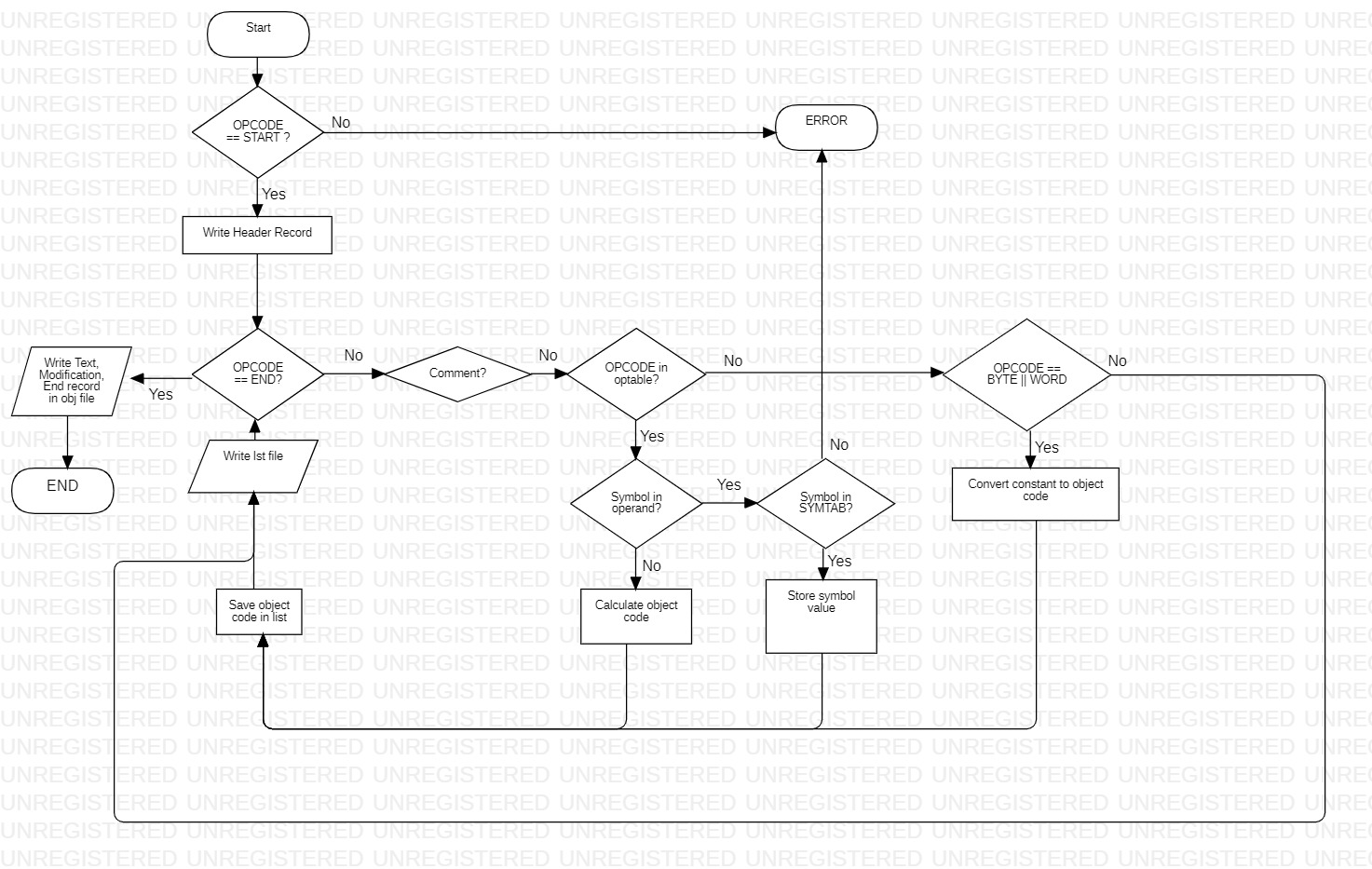
Project 1에서 구현한 SIC Machine을 바탕으로 \*.asm 파일을 입력 받아 \*.lst 파일과 \*.obj 파일을 생성하는 프로그램을 만들었습니다. 어셈블리 과정 중 pass 1과 pass 2 알고리즘을 이용하여 SYMTAB을 hash table 형태로 생성하였으며, 오류를 검출 할 수 있게 했습니다. 또한 type 명령어를 이용하여 파일을 볼 수 있게 해주었으며, symbol 명령어도 구현하여 가장 최근에 assemble에 성공한 파일의 SYMTAB을 확인 할 수 있게 해주었습니다.

1. **프로그램 설명**
   1. **프로그램 흐름도**

**PASS1**

****

**PASS2**

****

1. **모듈 정의**
   1. **모듈이름: int findName(char cmd[], char name[])**
      1. ***기능***

Type filename, assemble filename과 같이 space로 구분되는 명령어에서, space 뒤 부분을 string 에 저장해주는 함수. 자주 사용되는 기능이기도 하고. Project 1에서는 반복문으로 구현했었는데, space의 간격과 상관없이 명령어를 받으려다 글자 사이의 space도 무시하게 구현하여, 글자 사이의 space는 무시하지 않게 수정해주었다.

* + 1. ***사용변수***

int i, j - 반복문용 변수

int idx,idx2 = 첫 번째 space가 등장하는 위치, space가 끝나는 위치

cmdLen – 입력 받은 명령어의 길이

* 1. **모듈이름: int createSHash(char symb[])**
     1. ***기능***

SYMTAB을 hash table로 구성하기 위해 symbol에 따른 hash code를 반환해주는 함수. Symbol의 맨 앞 글자가 A로부터 얼마나 떨어져 있는지를 계산해준다.

* + 1. ***사용변수***

int result – 계산한 hash 값을 반환해주기 위한 변수

* 1. **모듈이름: opNode\* getFormat(char instruction[])**
     1. ***기능***

Opcode hash table에서 전달받은 opcode에 맞는 node를 찾아서 반환해주는 함수.

* + 1. ***사용변수***

int hash – 전달 받은 opcode의 hash number를 저장해주는 변수

opNode\* curr – opcode hash table에서 찾은 opcode를 가지고 있는 node를 반환해주기 위한 변수

* 1. **모듈이름: Symbol\* getSymbol(char symb[])**
     1. ***기능***

SYMTAB에서 전달받은 symbol의 정보를 담고 있는 node를 찾아서 반환해주는 함수

* + 1. ***사용변수***

int hash – 전달 받은 symbol의 hash number를 저장해주는 변수

Symbol \*find - SYMTAB에서 전달받은 symbol을 가지고 node를 반환해주기 위한 변수

* 1. **모듈이름: int pass1(char cmd[], char filename[])**
     1. ***기능***

Pass 1 알고리즘을 수행하는 함수. \*asm 파일로부터 한 줄씩 읽어서 intermediate 파일을 생성한다. 오류가 발생하면 -1을 반환한다. START이 처음이 아닌 곳에 등장할 때, symbol table에 같은 symbol이 존재할 때, opcode 영역에 적절하지 않은 입력이 들어 왔을 때 오류 처리를 해주었다. 함수의 flow는 순서도와 같다.

* + 1. ***사용변수***

FILE \*fp – asm 파일을 open하는 포인터

FILE \*midFile – intermediate 파일을 open하는 포인터

char temp1[50], temp2[50], temp3[50] – sscanf를 통해서 label, opcode, operand를 읽어오기 위한 배열

char line[150] – asm 파일로부터 한 줄씩 읽어서 저장하기 위한 배열

int scan\_num – sscanf를 통해 몇 개를 읽어왔는지 저장하는 변수. 읽어온 개수에 따라 형식이 달라진다. 예를 들어 3개를 읽어왔으면, label, opcode, operand 모두 읽어온 것이다

int i – loop용 변수

int hash – symbol이 존재하는 경우 symbol의 hash값을 저장하기 위한 변수

int format = 0, ret = 1 – opcode의 format을 읽어와서 저장해주는 변수. 함수의 return 값을 저장해주는 변수. 1인 경우 error가 없고 -1인 경우 에러이다.

int tempsnum – Symbol이 몇 개나 등장했는지 저장해주는 임시 변수. Assemble에 성공하면 이 값을 symNum에 저장해준다.

Symbol \*curr, \*next, \*temp – symbol을 찾거나 추가해주기 위해 사용되는 변수

* 1. **모듈이름: void printlst(FILE \* lst, int num, int loc, char label[], char opcode[], char operand[], char address[])**
     1. ***기능***

형식에 맞게 intermediate file에 출력해주는 함수. Pass2에서 읽어 들이는 string의 개수가 같도록, location이 존재하지 않는 line이면 -1을, symbol이 존재하지 않으면 ‘!’를 출력해주었다.

* + 1. ***사용변수***

없음

* 1. **모듈이름: void createobj(char TA[], int loc, int mf)**
     1. ***기능***

Pass2에서 .obj 파일을 만들기 위해 object code의 정보를 저장하는 함수. Object code, object code의 location, modify flag를 포함하는 node로 linked list를 생성해준다.

* + 1. ***사용변수***

Object\* obj – 정보를 저장하기 위한 node를 생성하기 위한 object 포인터

* 1. **모듈이름: int pass2(char filename[])**
     1. ***기능***

Pass2 알고리즘을 수행하기 위한 함수. Intermediate 파일로부터 한 줄씩 읽어서 object code를 생성하고 lst 파일과 obj 파일에 기록한다. start이나 end가 없거나, displacement가 범위를 벗어나는 경우, 잘못된 레지스터가 들어온 경우 등 에러 처리를 해주었으며, 에러가 발생하면 object file에 기록하기 위해 생성했던 list를 free하고 system call의 remove함수를 이용하여 생성한 .lst파일과 .obj파일을 제거한다. Flow는 순서도와 같다.

* + 1. ***사용변수***

FILE \*midFile, \*lst, \*obj – intermediate, .lst, .obj 파일을 열기 위한 포인터

int num, loc – linenum과 location을 저장하기 위한 변수

char label[50], opcode[50], operand[50] -lable, opcode, operand를 저장하기 위한 array

char line[150] – intermediate 파일에서 한 줄을 읽기 위한 변수

char rmoperand[50], rmopcode[50] – prefix 제거한 operand와 opcode

char TA[10], tempstr[100] = {'\0',} – target address와 obj 파일을 만들기 위해 object ocde를 나열하기 위해 만든 변수

char reg[][3] = {"A", "X", "L", "B", "S", "T", "F", "0", "PC", "SW"} – 레지스터들을 저장한 array

int N, I, X, B, P, E, xbpe, PC = 0 – NIXBPE 값을 저장하기 위한 변수와 program counter를 저장하기 위한 변수

char fname[50] = {'\0',}, fname2[50] = {'\0',} – file 이름을 저장하기 위한 변수

int i, cidx – loop 변수 comma의 위치를 저장하기 위한 변수

int mf – modify flag을 저장하기 위한 변수

int ret = 1 – return 값을 저장하기 위한 변수

int format = 0, BASE = 0, disp = 0 - format값과 base값 displacement 값을 저장하기 위한 변수

int startFlag = 0, ENDFLAG = 0 – START와 END가 있으면 flag를 set

Symbol\* curr – symbol을 찾기 위한 변수

opNode\* node – opcode를 찾기 위한 변수

Object\* obnode – object code list를 만들기 위한 변수

* 1. **모듈이름: int compare(const void \*a, const void \*b)**
     1. ***기능***

qsort 함수를 사용하기 위한 compare 함수

* + 1. ***사용변수***

없음

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

Command로 assmeble이 입력되면 pass1 함수와 pass2 함수를 호출해주는 함수. 시작 시 symList와 object code list를 free해준다. Assemble이 성공적이면 saveSym에 정렬된 SYMTAB을 저장한다. 이는 오류가 있는 경우 제일 마지막에 성공적이었던 symbol table을 저장해 symbol 명령에서 출력해주기 위함이다. 오류가 있는 경우 오류가 발생한 줄을 출력해준다. 에러가 발생하지 않았다면 이후 history에 명령어를 추가한다.

* + 1. ***사용변수***

char filename[100] – 입력받은 파일 이름을 저장하기 위한 변수

int i , j – loop 용 변수

Symbol \*curr – symList를 free해주기 위한 변수. saveSym에 sort된 SYMTAB을 저장하기 위한 변수

Object\* node – object code list를 free 해주기 위한 변수

int cmdLen; - 명령어의 길이를 저장하는 변수

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

Type 명령어가 입력된 경우 파일에 존재하는 내용을 화면에 보여주는 함수. 디렉토리 이름이 입력되는 경우 오류처리를 해주었다. 에러가 발생하지 않았다면 이후 history에 명령어를 추가한다.

* + 1. ***사용변수***

FILE \*fp – 파일을 열기 위한 파일 포인터

char c – 출력할 문자

char filename[100] = {'\0',} – 파일 이름을 저장하는 배열

DIR \*currDir – 디렉토리를 확인하기 위한 DIR\*

struct dirent \*currFile

struct stat currStat

int dirFlag = 0 – 입력된 파일명이 디렉토리이면 1

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

Symbol 명령어가 입력된 경우 가장 최근에 assemble에 성공한 파일의 SYMTAB을 출력해준다. SYMTAB에 symbol이 존재하지 않는 경우 존재하지 않는다고 해주었다.

에러가 발생하지 않았다면 이후 history에 명령어를 추가한다.

* + 1. *사용변수*

int i – loop용 변수

1. **전역 변수, 구조체, typedef 정의**
   1. typedef struct \_Symbol

symbol구조체로, symbol의 종류, 위치, 그리고 다음 node에 대한 link를 저장하고 있다.

* 1. typedef struct \_Object

object code들을 linked list 형태로 저장하기 위한 구조체. Object code, location, enterflag(RESW, RESB인 경우 .obj 파일에서 엔터가 필요하여 기록), 4형식 중에서 modification flag이 필요한 경우 modification flag을 저장하기 위한 변수를 가지고 있다.

* 1. int line\_num

pass1에서 얼만큼의 줄을 읽었는지 저장하는 변수

* 1. int line\_num2

pass2에서 얼만큼의 줄을 읽었는지 저장하는 변수

* 1. int start\_address

pass1에서 START이 지정해주는 start address를 저장해주는 변수

* 1. Symbol\* symList[26]

SYMTAB을 만들기 위해 만들 배열

* 1. int LOCCTR

LOCCTR를 저장해주는 변수

* 1. int symNum

SYMTAB에 입력된 symbol의 개수

* 1. int enterflag

RESW, RESB의 경우 .obj 파일을 만들 때, 줄이 바뀌어야 한다. 그것을 기록해주는 flag

* 1. Object \*ohead, \* otail

Object code list의 head와 tail을 가리키는 포인터

* 1. int objfree

오류가 나서 pass 2에서 object list를 free 해주었으면 다시 q를 입력 받아 프로그램이 종료될 때 다시 free 해주지 않기 위해 사용하는 flag.

* 1. Symbol \*saveSym

Assemble에 성공한 가장 최근의 SYMTAB을 저장하기 위한 포인터

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;

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]; // for memory space 1048756 = 16 ^ 5

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

// 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

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");

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));

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);

}

**5.2 2016143.c**

#include "20161643.h"

int main() {

char command[100] = {'\0',}; // array to save command

char trimCmd[100] = {'\0',}; // array to save trimmed command

int cmdNum; // variable to save command number

History\* temp; // variable to free history link

int i;

opNode \*curr, \*prev;

Symbol \*sym;

Object \*node;

funcPtr fPtr[] = { cmdHelp, cmdDir, cmdQuit, cmdHistory, cmdDump, cmdEdit, cmdFill, cmdReset, cmdOpcode, cmdOpcodelist, cmdAssemble, cmdType, cmdSymbol}; // function pointer to reduce the code

loadOpcode(); // load opcode list from opcode.txt

while (!endFlag) {

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

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

printf("sicsim>");

fgets(command, sizeof(command), stdin);

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

cmdNum = checkCmd(command, trimCmd); // check the command number

if (cmdNum == -1)

printf("Invalid Command\n");

else {

fPtr[cmdNum](trimCmd); // call the function that has the right command number

}

}

while(head){ // free history list

temp = head;

head = head->link;

free(temp);

}

for(i=0; i < 20; i++){ // free opcode list

curr = opList[i];

while(curr){

prev = curr;

curr = curr->link;

free(prev);

}

}

for (i = 0; i < 26; i++) { // free symbol list

while (symList[i]) {

sym = symList[i];

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

free(sym);

}

}

if(saveSym != NULL) // free saved symbol

free(saveSym);

if(objfree != 1){

while (ohead) { // free object list

node = ohead;

ohead = ohead->link;

free(node);

}

}

return 0;

}