과목 명: 시스템프로그래밍

담당 교수 명: 소 정 민

<<Assignment 1>>

**서강대학교 컴퓨터공학과**

**[20141515]**

**[김연후]**

목 차

1. 프로그램 개요

2. 프로그램 설명 3

2.1 프로그램 흐름도 3

3. 모듈 정의 3

3.1 모듈 이름 : main() 3

3.1.1 기능 3

3.1.2 사용 변수 3

3.2 모듈 이름 : getFormat() 4

3.2.1 기능 4

3.2.2 사용 변수 4

3.3 모듈 이름 : insertSym() 4

3.3.1 기능 4

3.3.2 사용변수 4

3.4 모듈 이름 : searchSym() 4

3.4.1 기능 4

3.4.2 사용변수 4

3.5 모듈 이름 : printSym() 4

3.5.1 기능 4

3.5.2 사용변수 4

3.6 모듈 이름 : freeSym() 4

3.6.1 기능 4

3.6.2 사용변수 4

3.7 모듈 이름 : findSym() 5

3.7.1 기능 5

3.7.2 사용변수 5

3.8 모듈 이름 : findSymAddr() 5

3.8.1 기능 5

3.8.2 사용변수 5

3.9 모듈 이름 : hexTostring() 5

3.9.1 기능 5

3.9.2 사용변수 5

3.10 모듈 이름 : makeObjectcode() 5

3.10.1 기능 5

3.10.2 사용변수 5

3.11 모듈 이름 : displayFile() 5

3.11.1 기능 5

3.11.2 사용변수 6

3.12 모듈 이름 : assembleFile() 6

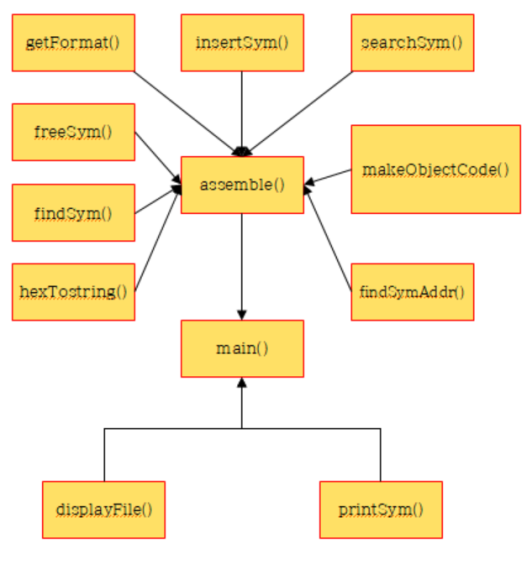
3.12.1 기능 6

3.12.2 사용변수 6

1. **프로그램 개요**

프로그램에서 assemble (filename) 을 이용하여 SIC/XE 어셈블리 프로그램 소스파일을 input으로 받아와 object file과 assembly listing file을 만드는 기능을 할 수 있도록 하였다. 또한 type (filename) 으로 해당 파일의 내용을 볼 수 있게 출력하는 기능을 만들었다.

1. **프로그램 설명**
   1. **프로그램 흐름도**

**그림 1> 프로그램 흐름도**

프로그램을 구성하는 모듈의 구조를 도식하고, 각 모듈간의 상호작용에 사용되는 데이터에 대해서 기술한다. 각각의 상호작용의 흐름은 번호를 부여하여 도식한다.

1. **모듈 정의**
   1. **모듈 이름 : main()**
      1. *기능*

Display\_instructions를 호출해 프로그램이 하는 일을 설명한 후, 사용자의 입력을 받아서 나머지 모듈을 호출하거나 끝낸다.

* + 1. *사용 변수*

SymTab \*\*symtab - symbol table을 담는 구조체 변수

int assembleFlag - 어셈블 된 상태인지 확인 (Yes : 1, No : 0)

* 1. **모듈 이름: getFormat()**
     1. *기능*

해당 opcode mnemonic이 어떤 Format의 object code로 저장되는지 Format에 대한 정보를 가져온다.

* + 1. *사용 변수*

HashTable \*\*mne - mnemonic 해쉬테이블

char \*str - mnemonic string

int index - 찾는 opcode mnemonic이 담겨있는 해쉬테이블 linked list

헤드의 index

* 1. **모듈이름: insertSym()**
     1. *기능*

SYMTAB(symbol table)에 해당 symbol이 없는 경우 pass1의 과정에서 symbol을 삽입해주는 기능을 한다.

* + 1. *사용변수*

SymTab \*\*symtab - symbol table 구조체 변수

char \*label - 삽입할 symbol

int locctr - 해당 label의 location address

* 1. **모듈이름: searchSym()**
     1. *기능*

SYMTAB(symbol table)에 symbol이 있는지 확인하는 기능을 한다.

* + 1. *사용변수*

SymTab \*\*symtab - symbol table 구조체 변수

char \*symbol - 찾고자 하는 symbol

* 1. **모듈이름: printSym()**
     1. *기능*

main()에서 명령어 symbol 호출 시, assemble 된 symbol 들을 출력해주는 기능을 한다.

* + 1. *사용변수*

SymTab \*\*symtab - symbol table 구조체 변수

* 1. **모듈이름: freeSym()**
     1. *기능*

assemble 된 symbol 들을 저장하고 있는 구조체 변수들의 메모리를 free 시켜준다.

* + 1. *사용변수*

SymTab \*\*symtab - symbol table 구조체 변수

* 1. **모듈이름: findSym()**
     1. *기능*

해당 instruction (inst) 내부에 있는 operand(symbol)을 걸러내는 기능을 한다.

* + 1. *사용변수*

SymTab \*\*symtab - symbol table 구조체 변수

char \*str - symbol을 받아갈 string 변수

char \*inst - insturctions of assembly file

* 1. **모듈이름: findSymAddr()**
     1. *기능*

찾고자 하는 symbol의 주소를 찾아주는 기능을 한다.

* + 1. *사용변수*

SymTab \*\*symtab - symbol table 구조체 변수

char \*symbol - 찾고자 하는 symbol

* 1. **모듈이름: hexTostring()**
     1. *기능*

16진수를 문자열로 변경(대문자로object code에 맞춰서)하는 기능을 한다.

* + 1. *사용변수*

int n - 변경하고자 하는 16진수

char \*str - 받고자 하는 16진수의 문자열 형태

int size - object code의 규격 (자리 수)

* 1. **모듈이름: makeObjectcode()**
     1. *기능*

각 Format 별로 해당 instruction을 objectcode로 변환하여 반환하는 기능을 한다.

* + 1. *사용변수*

SymTab \*\*symtab - symbol table 구조체 변수

int locctr - 현재 줄의 주소

int format - 현재 instruction의 object code format

int opcode - 현재 instruction의 mnemonic의 opcode

char \*inst - 현재 instruction string

char \*baseSym - base relative addressing에 사용될 base symbol

* 1. **모듈이름: displayFile()**
     1. *기능*

main에서 프로그램 명령어 type 기능을 위한 해당 파일을 출력하여 보여주는 기능을 한다.

* + 1. *사용변수*

char \*fname - display할 file name

* 1. **모듈이름: assembleFile()**
     1. *기능*

assembly program source를 받아와서 SIC/XE assembly program을 이용하여 읽고, lst, obj 파일을 만드는 기능을 한다.

* + 1. *사용변수*

char \*fname - assemble 할 file name

HashTable \*\*mne - opcode mnemonic들의 해쉬테이블

SymTab \*\*symtab - symbol table 구조체 변수

1. **코드**

/\*포함되는 파일\*/

/\*헤더파일\*/

#ifndef \_\_20141515\_\_

#define \_\_20141515\_\_

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<sys/stat.h>

#include"dirent.h"

typedef struct Node\_ {

char strcmd[500];

struct Node\_ \*link;

}Node

typedef struct HashTable\_ {

char mne[10];

int format;

int opcode;

struct HashTable\_ \*link;

}HashTable

typedef struct SymTab\_ {

char sym[10];

int value;

int flag;

struct SymTab\_ \*link;

}SymTab

Node \*head;

Node \*rear;

void addNode(char \*cmd);

void help();

void dir();

void history();

int charTohex(char c);

int TwoInputsString(char \*str, int \*first, int \*second);

int dump(unsigned char \*mem, int s, int e);

int ThreeInputsString(char \*str, int \*first, int \*second, int \*third);

int hashfunc(char \*str);

int getOpcode(HashTable \*\*mne, char \*str, int index);

int getFormat(HashTable \*\*mne, char \*str, int index);

void insertSym(SymTab \*\*symtab, char \*label, int locctr);

int searchSym(SymTab \*\*symtab, char \*symbol);

void printSym(SymTab \*\*symtab);

void freeSym(SymTab \*\*symtab);

int findSym(SymTab \*\*symtab, char \*str, char \*inst);

int findSymAddr(SymTab \*\*symtab, char \*symbol);

void hexTostring(int n, char \*str, int size);

int makeObjectcode(SymTab \*\*symtab, int locctr, int format, int opcode, char \*inst, char \*baseSym);

void displayFile(char \*fname);

int assembleFile(char \*fname, HashTable \*\*mne, SymTab \*\*symtab);

#endif

/\*소스파일\*/

#include "20141515.h"

void addNode(char \*cmd) { // when valid command is excuted add the command string to the linked list (for history)

Node \*n = (Node\*)malloc(sizeof(Node));

strcpy(n->strcmd, cmd);

n->link = NULL;

if (head == NULL)

head = n;

else

rear->link = n;

rear = n;

}

void help() { // display all commands supported by this program

printf("\n");

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

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

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

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

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

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

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

printf("reset\n");

printf("opcode mnemonic\n");

printf("opcodelist\n");

printf("assemble filename\n");

printf("type filename\n");

printf("symbol\n\n");

}

void dir() { // shows the files in the current directory

DIR \*dir;

struct dirent \*dire;

struct stat statbuf;

dir = opendir("./");

printf("\n");

if (dir != NULL) {

readdir(dir);

readdir(dir);

while ((dire = readdir(dir)) != NULL) {

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

if (stat(dire->d\_name, &statbuf) < 0) {

perror("stat error\n");

return;

}

if (S\_ISDIR(statbuf.st\_mode)) // if it's a directory

printf("/");

else if ((statbuf.st\_mode & S\_IXUSR) != 0 || (statbuf.st\_mode & S\_IXGRP) != 0 || (statbuf.st\_mode & S\_IXOTH) != 0) // if file has execute permission (which means it's executable file)

printf("\*");

printf("\n");

}

closedir(dir);

}

else

perror("\nerror on directory output...(d[ir])\n\n");

printf("\n");

}

void history() { // print out history of all valid commands excuted

Node \*temp = head;

int i = 1;

printf("\n");

while (temp != NULL) {

printf("%d\t%s\n", i, temp->strcmd);

temp = temp->link;

i++;

}

printf("\n");

}

int charTohex(char c) { // change character to hexadecimal

if (c >= '0'&&c <= '9')

return c - '0';

else if (c >= 'A'&&c <= 'F')

return c - 'A' + 10;

else if (c >= 'a'&&c <= 'f')

return c - 'a' + 10;

else

return -1;

}

int TwoInputsString(char \*str, int \*first, int \*second) { // get two inputs from str (used for du[mp] and e[dit])

int i = 0, startSize = 0, comaFlag = 0, endSize = 0, spaceFlag = 0, f = 0, s = 0;

while (str[i] != '\0') {

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

if ((str[i] >= '0'&&str[i] <= '9') || (str[i] >= 'A'&&str[i] <= 'F') || (str[i] >= 'a'&&str[i] <= 'f') || (str[i] == ',') || (str[i] == '0' && (str[i + 1] == 'X' || str[i + 1] == 'x'))) {

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

if (startSize == 0) { // command , ...

printf("\nWrong Command!!!\n\n");

return 0;

}

else { // command start, ...

if (str[i + 1] == '\0') { // command start, (error)

printf("\nWrong Command!!!\n\n");

return 0;

}

comaFlag++;

spaceFlag = 0;

if (comaFlag >= 2) { // when there are more inputs (error)

printf("\nWrong Command!!!\n\n");

return 0;

}

}

}

else if (str[i] == '0' && (str[i + 1] == 'X' || str[i + 1] == 'x') && str[i - 1] != '0') {

if (!((str[i + 2] >= '0' && str[i + 2] <= '9') || (str[i + 2] >= 'A' && str[i + 2] <= 'F') || (str[i + 2] >= 'a' && str[i + 2] <= 'f'))) { // command start, 0x (end error)

printf("\nWrong Command!!!\n\n");

return 0;

}

i++;

}

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

if (str[i + 1] == '\0' && comaFlag == 0) // command start

s = -1;

if (comaFlag == 0) {

f = 16 \* f + charTohex(str[i]);

startSize++;

}

if (comaFlag == 1) {

s = 16 \* s + charTohex(str[i]);

if (s > 0xFFFFF) {

s = 0xFFFFF;

break;

}

endSize++;

}

if (spaceFlag == 1 && (startSize != 0 || endSize != 0)) { // command s tart... or dump stard end (errors)

printf("\nWrong Command!!! (Input error)\n\n");

return 0;

}

}

}

else {

printf("\nWrong Command!!! (Input error)\n\n"); // other words in string (error) com mand (error)

return 0;

}

}

else {

if (startSize != 0 && comaFlag == 0)

spaceFlag = 1;

if (endSize != 0) { // command start, e nd (error)

printf("\nWrong Command!!! (Input error)\n\n");

return 0;

}

}

i++;

}

\*first = f;

if (s != -1)

\*second = s;

else

\*second = -1;

return 1;

}

int dump(unsigned char \*mem, int s, int e) { // outputs for command d[ump]

int eAddr, start, tmps;

printf("\n");

if (s <= e || e == -1) { // print address from start to end (when e == -1 there is no end address)

int i = 0;

if (e == -1)

e = s + 159;

if (e > 0xFFFFF)

e = 0xFFFFF;

eAddr = e;

start = s;

while (start >= 0) {

start -= 16;

i++;

}

i--;

start = 0;

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

start += 16;

tmps = start;

for (int j = 0; j < (e - tmps) / 16 + 1; j++) {

if (tmps <= s && s <= e && e <= tmps + 15) {

printf("%05X ", start);

for (int k = tmps; k < s; k++)

printf(" ");

for (int k = s; k <= e; k++)

printf("%02X ", mem[k]);

for (int k = e + 1; k <= tmps + 15; k++)

printf(" ");

printf("; ");

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

printf(".");

printf("\n");

break;

}

if (j < (e - tmps) / 16) {

printf("%05X ", start);

for (int k = 0; k < 16; k++) {

if (j == 0 && k < s - (s / 16) \* 16)

printf(" ");

else

printf("%02X ", mem[tmps + 16 \* j + k]);

}

printf("; ");

for (int k = 0; k < 16; k++) {

if (mem[tmps + 16 \* j + k] < 0x20 || mem[tmps + 16 \* j + k]>0x7E)

printf(".");

else

printf("%c", mem[tmps + 16 \* j + k]);

}

printf("\n");

start += 16;

}

if (j == (e - tmps) / 16 && e - (e / 16) \* 16 >= 0) {

printf("%05X ", start);

for (int k = 0; k < 16; k++) {

if (k <= e - (e / 16) \* 16)

printf("%02X ", mem[tmps + 16 \* j + k]);

else

printf(" ");

}

printf("; ");

for (int k = 0; k < 16; k++) {

if (k <= e - (e / 16) \* 16) {

if (mem[tmps + 16 \* j + k] < 0x20 || mem[tmps + 16 \* j + k]>0x7E)

printf(".");

else

printf("%c", mem[tmps + 16 \* j + k]);

}

else

printf(".");

}

printf("\n");

}

}

}

else {

eAddr = 0;

printf("Wrong Command!!!(start address must be smaller than end address...)\n");

}

printf("\n");

return eAddr;

}

int ThreeInputsString(char \*str, int \*first, int \*second, int \*third) { // get three inputs from str (used for f[ill])

int i = 0, firstSize = 0, secondSize = 0, thirdSize = 0, comaFlag = 0, spaceFlag = 0, f = 0, s = 0, t = 0;

while (str[i] != '\0') {

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

if ((str[i] >= '0'&&str[i] <= '9') || (str[i] >= 'A'&&str[i] <= 'F') || (str[i] >= 'a'&&str[i] <= 'f') || (str[i] == ',') || (str[i] == '0' && (str[i + 1] == 'X' || str[i + 1] == 'x'))) {

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

if (firstSize == 0) { // command , ...

printf("\nWrong Command!!!\n\n");

return 0;

}

else { // command start, ...

if (str[i + 1] == '\0') { // command start, (error)

printf("\nWrong Command!!!\n\n");

return 0;

}

comaFlag++;

spaceFlag = 0;

if (comaFlag >= 3) {

printf("\nWrong Command!!!\n\n");

return 0;

}

}

}

else if (str[i] == '0' && (str[i + 1] == 'X' || str[i + 1] == 'x') && str[i - 1] != '0') {

if (!((str[i + 2] >= '0' && str[i + 2] <= '9') || (str[i + 2] >= 'A' && str[i + 2] <= 'F') || (str[i + 2] >= 'a' && str[i + 2] <= 'f'))) { // command start, 0x (end error)

printf("\nWrong Command!!!\n\n");

return 0;

}

i++;

}

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

if (comaFlag == 0) {

f = 16 \* f + charTohex(str[i]);

firstSize++;

}

if (comaFlag == 1) {

s = 16 \* s + charTohex(str[i]);

secondSize++;

}

if (comaFlag == 2) {

t = 16 \* t + charTohex(str[i]);

thirdSize++;

}

if (spaceFlag == 1 && (firstSize != 0 || secondSize != 0 || thirdSize != 0)) { // command s tart... or dump stard end (errors)

printf("\nWrong Command!!! (Input error)\n\n");

return 0;

}

}

}

else {

printf("\nWrong Command!!! (Input error)\n\n"); // other words in string (error) com mand (error)

return 0;

}

}

else {

if ((firstSize != 0 && comaFlag == 0) || (secondSize != 0 && comaFlag == 1))

spaceFlag = 1;

if (thirdSize != 0) { // command start, e nd (error)

printf("\nWrong Command!!! (Input error)\n\n");

return 0;

}

}

i++;

}

if (firstSize == 0 || secondSize == 0 || thirdSize == 0) {

printf("\nWrong Command!!!(Need to write 3 inputs)\n\n");

return 0;

}

\*first = f;

\*second = s;

\*third = t;

if (f > s) {

printf("\nWrong Command!!!(start address must be smaller than end address...)\n\n");

return 0;

}

return 1;

}

int hashfunc(char \*str) { // hashfunction : Returns the remainder value of res(sum of values of each characters ASCII code in string) divided by 20

int i = 0, res = 0;

while (str[i] != '\0') {

res += (int)str[i];

i++;

}

return res % 20;

}

int getOpcode(HashTable \*\* mne, char \*str, int index) { // get (str)'s opcode from hashtable

HashTable \*temp = mne[index];

while (temp != NULL) {

if (strcmp(temp->mne, str) == 0)

return temp->opcode;

temp = temp->link;

}

return -1;

}

int getFormat(HashTable \*\*mne, char \*str, int index) {

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

return 0;

HashTable \*temp = mne[index];

while (temp != NULL) {

if (strcmp(temp->mne, str) == 0)

return temp->format;

temp = temp->link;

}

return -1;

}

void makeHashTable(HashTable \*\*mne) { // make hashtable of opcode

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

char m[10], tmp;

int code, i, j, format;

tmp = fgetc(fp);

while (tmp != EOF) {

i = 0; j = 0; code = 0;

for (int k = 0; k < 10; k++)

m[k] = 0;

while (tmp != '\n') {

if (i < 2) {

if (tmp >= '0'&&tmp <= '9')

code = code \* 16 + tmp - '0';

if (tmp >= 'A'&&tmp <= 'F')

code = code \* 16 + tmp - 'A' + 10;

i++;

}

else {

if (tmp >= 'A'&&tmp <= 'Z') {

m[j] = tmp;

j++;

}

if (tmp >= '1'&&tmp <= '3') {

format = tmp - '0';

}

}

tmp = fgetc(fp);

}

HashTable \*temp, \*node;

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

strcpy(temp->mne, m);

temp->opcode = code;

temp->format = format;

temp->link = NULL;

if (mne[hashfunc(m)] == NULL)

mne[hashfunc(m)] = temp;

else {

node = mne[hashfunc(m)];

while (node->link != NULL)

node = node->link;

node->link = temp;

}

tmp = fgetc(fp);

}

}

void insertSym(SymTab \*\*symtab, char \*label, int locctr) {

int i = label[0] - 'A', inputFlag = 0;

SymTab \*data = (SymTab\*)malloc(sizeof(SymTab));

SymTab \*head = symtab[i];

SymTab \*temp;

strcpy(data->sym, label);

data->value = locctr;

data->link = NULL;

if (symtab[i] == NULL)

symtab[i] = data;

else {

while (head->link != NULL) {

if (strcmp(symtab[i]->sym, data->sym) < 0) {

data->link = symtab[i];

symtab[i] = data;

inputFlag = 1;

break;

}

else if (strcmp(head->link->sym, data->sym) < 0) {

temp = head->link;

head->link = data;

data->link = temp;

break;

inputFlag = 1;

}

head = head->link;

}

if (inputFlag == 0)

head->link = data;

}

}

int searchSym(SymTab \*\*symtab, char \*symbol) {

SymTab \*temp;

if (symbol[0] - 'A' < 0 || symbol[0] - 'A'> 26)

return 0;

temp = symtab[symbol[0] - 'A'];

while (temp != NULL) {

if (strcmp(temp->sym, symbol) == 0)

return 1;

else

temp = temp->link;

}

return 0;

}

void printSym(SymTab\*\* symtab) {

SymTab \*temp;

for (int i = 25; i >= 0; i--) {

if (symtab[i] != NULL) {

temp = symtab[i];

while (temp != NULL) {

printf("\t%s\t%04X\n", temp->sym, temp->value);

temp = temp->link;

}

}

}

}

void freeSym(SymTab \*\*symtab) {

SymTab \*temp;

for (int i = 25; i >= 0; i--) {

while (symtab[i] != NULL) {

temp = symtab[i];

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

free(temp);

}

}

}

int findSym(SymTab \*\*symtab, char \*str, char \*inst) {

int j = 0, strFlag = 0;

for (int i = 0; i < strlen(inst); i++) {

if ((inst[i] >= 'A' && inst[i] <= 'Z') || (inst[i] >= '0' && inst[i] <= '9')) {

strFlag = 1;

str[j] = inst[i];

j++;

}

else {

if (strFlag == 1) {

if (searchSym(symtab, str) == 1)

return 1;

else

return 0;

}

}

}

if (strFlag == 1) {

if (searchSym(symtab, str) == 1)

return 1;

else

return 0;

}

else

return 0;

}

int findSymAddr(SymTab \*\*symtab, char \*symbol) {

SymTab \*temp = symtab[symbol[0] - 'A'];

while (temp != NULL) {

if (strcmp(temp->sym, symbol) == 0)

return temp->value;

else

temp = temp->link;

}

return 0;

}

void hexTostring(int n, char \*str, int size) {

int temp = n;

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

str[i] = 0;

for (int i = size-1; i >=0; i--) {

temp = n;

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

temp /= 16;

temp %= 16;

if (temp >= 0 || temp <= 9)

str[size - i - 1] = temp + '0';

if (temp == 10)

str[size - i - 1] = 'A';

if (temp == 11)

str[size - i - 1] = 'B';

if (temp == 12)

str[size - i - 1] = 'C';

if (temp == 13)

str[size - i - 1] = 'D';

if (temp == 14)

str[size - i - 1] = 'E';

if (temp == 15)

str[size - i - 1] = 'F';

}

}

int makeObjectcode(SymTab \*\*symtab, int locctr, int format, int opcode, char \*inst, char \*baseSym) {

int n = 1, i = 1, x = 0, b = 0, p = 1, e = 0;

int objectcode = -1, baseAddr, pcAddr;

char temp[10] = { 0 };

baseAddr = findSymAddr(symtab, baseSym);

if (inst[0] == '#')

n = 0;

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

i = 0;

if (inst[strlen(inst) - 1] == 'X')

x = 1;

if (format == 0) { // BYTE CHAR

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

int k = 2;

objectcode = 0;

while (inst[k] != '\'') {

objectcode \*= 256;

objectcode += (int)inst[k];

k++;

}

}

else if (inst[0] == 'X') { // BYTE HEX

temp[0] = inst[2];

temp[1] = inst[3];

objectcode = strtol(temp, NULL, 16);

}

else // WORD

objectcode = strtol(inst, NULL, 16);

}

if (format == 1)

return opcode;

if (format == 2) {

char reg[7] = { 'A','X','L','B','S','T','F' };

int j = 0;

objectcode = 0;

objectcode += opcode \* 256;

for (int k = 0; k < strlen(inst); k++) {

if (inst[k] != ' ' && inst[k] != '\t') {

temp[j] = inst[k];

j++;

}

}

if (strlen(temp) == 1) {

for (int k = 0; k < 7; k++) {

if (temp[0] == reg[k]) {

objectcode += k \* 16;

break;

}

}

}

if (strlen(temp) == 2) {

if (temp[0] == 'P' && temp[1] == 'C')

objectcode += 8 \* 16;

if (temp[0] == 'S' && temp[1] == 'W')

objectcode += 9 \* 16;

}

if (strlen(temp) == 3) {

for (int k = 0; k < 7; k++) {

if (temp[0] == reg[k])

objectcode += k \* 16;

if (temp[2] == reg[k])

objectcode += k;

}

}

if (strlen(temp) == 4) {

if (temp[1] == ',') {

for (int k = 0; k < 7; k++) {

if (temp[0] == reg[k]) {

objectcode += k \* 16;

break;

}

}

if (temp[3] == 'P' && temp[4] == 'C')

objectcode += 8;

if (temp[3] == 'S' && temp[4] == 'W')

objectcode += 9;

}

if (temp[2] == ',') {

if (temp[0] == 'P' && temp[1] == 'C')

objectcode += 8 \* 16;

if (temp[0] == 'S' && temp[1] == 'W')

objectcode += 9 \* 16;

for (int k = 0; k < 7; k++) {

if (temp[3] == reg[k]) {

objectcode += k;

break;

}

}

}

}

if (strlen(temp) == 5) {

if (temp[0] == 'P' && temp[1] == 'C')

objectcode += 8 \* 16;

if (temp[0] == 'S' && temp[1] == 'W')

objectcode += 9 \* 16;

if (temp[3] == 'P' && temp[4] == 'C')

objectcode += 8;

if (temp[3] == 'S' && temp[4] == 'W')

objectcode += 9;

}

}

if (format == 3) {

char operand[20] = { 0 };

int t;

t = findSym(symtab, operand, inst);

objectcode = 0;

if (operand[0] >= 'A' && operand[0] <= 'Z') {

pcAddr = findSymAddr(symtab, operand) - (locctr + 3);

if (pcAddr >= -2048 && pcAddr < 2047) {

if (pcAddr < 0) {

pcAddr += 4096;

objectcode += (opcode + 2 \* n + i) \* 65536;

objectcode += (8 \* x + 4 \* b + 2 \* p + e) \* 4096;

objectcode += pcAddr;

}

else {

objectcode += (opcode + 2 \* n + i) \* 65536;

objectcode += (8 \* x + 4 \* b + 2 \* p + e) \* 4096;

objectcode += pcAddr;

}

}

else {

int addr = findSymAddr(symtab, operand) - baseAddr;

b = 1; p = 0;

objectcode += (opcode + 2 \* n + i) \* 65536;

objectcode += (8 \* x + 4 \* b + 2 \* p + e) \* 4096;

objectcode += addr;

}

}

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

p = 0;

t = atoi(operand);

objectcode += (opcode + 2 \* n + i) \* 65536;

objectcode += (8 \* x + 4 \* b + 2 \* p + e) \* 4096;

objectcode += t;

}

else {

p = 0;

t = atoi(operand);

objectcode += (opcode + 2 \* n + i) \* 65536;

objectcode += (8 \* x + 4 \* b + 2 \* p + e) \* 4096;

objectcode += t;

}

}

if (format == 4) {

objectcode = 0;

b = 0; p = 0; e = 1;

objectcode += (opcode + 2 \* n + i) \* 16 + 8 \* x + 4 \* b + 2 \* p + e;

}

return objectcode;

}

void displayFile(char \* fname) {

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

char t;

if (fp == NULL) {

printf("There is no such name of file...\n\n");

return;

}

printf("\n");

while (1) {

t = fgetc(fp);

if (t == EOF)

break;

printf("%c", t);

}

printf("\n");

}

int assembleFile(char \*fname, HashTable \*\*mne, SymTab \*\*symtab) {

char str[200], symbol[20] = { 0 }, \*opcode, inst[20] = { 0 }, name[7] = { 0 }, baseSym[20] = { 0 };

int linenum = 5, size, symFlag, opFlag, insFlag, cmntFlag, errorFlag, errorCnt = 0, startFlag = 0, endFlag = 0, startAddr, locctr = 0, proglen, format4Flag;

char reg[7] = { 'A','X','L','B','S','T','F' };

////////////////////////////

// Pass1 of the Assembler //

////////////////////////////

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

if (fp == NULL) {

printf("There is no such name of file...\n\n");

return -1;

}

opcode = (char\*)malloc(20 \* sizeof(char));

while (fgets(str, 201, fp) != NULL) {

size = strlen(str); symFlag = 0; opFlag = 0; insFlag = 0; cmntFlag = 0; format4Flag = 0; errorFlag = 0;

int j = 0;

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

symbol[i] = 0;

opcode[i] = 0;

inst[i] = 0;

}

for (int i = 0; i < size; i++) {

if (str[i] == ' ' || str[i] == '\t' || str[i] == '\n') {

continue;

}

else if ((str[i] >= 'A' && str[i] <= 'Z') || (str[i] >= '0' && str[i] <= '9') || str[i] == '#' || str[i] == '@' || str[i] == '+' || str[i] == ',' || str[i] == '\'') {

if (str[i] == '#')

continue;

if (str[i] == '@')

continue;

if (str[i] == '+') {

format4Flag = 1;

continue;

}

if (symFlag == 0 && opFlag == 0)

symbol[j] = str[i];

if (symFlag == 1 && opFlag == 0)

opcode[j] = str[i];

if (opFlag == 1 && insFlag == 0)

inst[j] = str[i];

j++;

if (str[i + 1] == ' ' || str[i + 1] == '\t' || str[i + 1] == '\n') {

if (str[i] == ',')

continue;

if ((getOpcode(mne, symbol, hashfunc(symbol)) != -1 && opFlag == 0)

|| strcmp(symbol, "START") == 0 || strcmp(symbol, "END") == 0

|| strcmp(symbol, "BASE") == 0 || strcmp(symbol, "NOBASE") == 0) {

strcpy(opcode, symbol);

for (int k = 0; k < 10; k++)

symbol[k] = 0;

opFlag = 1;

}

else {

if (opFlag == 1) {

insFlag = 1;

}

else if (symFlag == 1) {

opFlag = 1;

}

else

symFlag = 1;

}

j = 0;

}

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

if (str[i + 1] == ',') {

errorFlag = -1;

errorCnt++;

break;

}

}

}

else if (str[i] == '.') {

cmntFlag = 1;

break;

}

else if (j >= 20) {

errorFlag = -1;

errorCnt++;

break;

}

else {

errorFlag = -1;

errorCnt++;

break;

}

}

for (int i = 0; i < strlen(symbol); i++) { // error check symbol

if (!((symbol[i] >= 'A' && symbol[i] <= 'Z') || (symbol[i] >= '0' && symbol[i] <= '9'))) {

errorFlag = -1;

errorCnt++;

break;

}

}

for (int i = 0; i < strlen(opcode); i++) { // error check opcode

if (opcode[i]<'A' || opcode[i]>'Z') {

errorFlag = -1;

errorCnt++;

break;

}

}

int quoteFlag = 0;

for (int i = 0; i < strlen(inst); i++) { // error check instruction

if (strcmp(opcode, "BYTE") == 0) {

if ((!(inst[i] == 'C' || inst[i] == 'X')) && i == 0) {

errorFlag = -1;

errorCnt++;

break;

}

if (inst[i] == '\'' && i == 1) {

quoteFlag++;

continue;

}

if (inst[i] == '\'' && i == strlen(inst) - 1) {

quoteFlag++;

continue;

}

if (!((inst[i] >= 'A' && inst[i] <= 'Z') || (inst[i] >= '0' && inst[i] <= '9'))) {

errorFlag = -1;

errorCnt++;

break;

}

}

if (inst[i] == ',' && (i == 0 || i == strlen(inst) - 1)) {

errorFlag = -1;

errorCnt++;

break;

}

if (!((inst[i] >= 'A' && inst[i] <= 'Z') || (inst[i] >= '0' && inst[i] <= '9') || inst[i] == ',')) {

errorFlag = -1;

errorCnt++;

break;

}

}

int nerror = 0, nerror2 = 0;

if (strcmp(opcode, "CLEAR") == 0 || strcmp(opcode, "TIXR") == 0) { // error check in using registers

for (int k = 0; k < 7; k++) {

if ((inst[0] == reg[k]) && inst[1] == '\0') {

nerror = 1;

break;

}

}

if (strcmp(inst, "PC") == 0 || strcmp(inst, "SW") == 0)

nerror = 1;

nerror2 = 1;

if (nerror == 0 || nerror == 0) {

errorFlag = -1;

errorCnt++;

}

}

if (strcmp(opcode, "ADDR") == 0 || strcmp(opcode, "COMPR") == 0 || strcmp(opcode, "DIVR") == 0 || strcmp(opcode, "MULR") == 0 || strcmp(opcode, "RMO") == 0 || strcmp(opcode, "SUBR") == 0) { // error check in using registers

char temp[2] = { 0 };

temp[0] = inst[0];

if (inst[1] != ',') {

temp[1] = inst[1];

if (strcmp(temp, "PC") == 0 || strcmp(temp, "SW") == 0)

nerror = 1;

if (strlen(inst) == 4) {

for (int k = 0; k < 7; k++) {

if (inst[3] == reg[k]) {

nerror2 = 1;

break;

}

}

}

if (strlen(inst) == 5) {

temp[0] = inst[3];

temp[1] = inst[4];

if (strcmp(temp, "PC") == 0 || strcmp(temp, "SW") == 0)

nerror2 = 1;

}

}

else {

for (int k = 0; k < 7; k++) {

if (inst[0] == reg[k]) {

nerror = 1;

break;

}

}

if (strlen(inst) == 3) {

for (int k = 0; k < 7; k++) {

if (inst[2] == reg[k]) {

nerror2 = 1;

break;

}

}

}

if (strlen(inst) == 4) {

temp[0] = inst[2];

temp[1] = inst[3];

if (strcmp(temp, "PC") == 0 || strcmp(temp, "SW") == 0)

nerror2 = 1;

}

}

if (nerror == 0 || nerror == 0) {

errorFlag = -1;

errorCnt++;

}

}

if (!(quoteFlag == 0 || quoteFlag == 2)) {

errorFlag = -1;

errorCnt++;

}

if (strcmp(opcode, "START") == 0) {

startFlag = 1;

if (strlen(symbol) <= 6) {

strcpy(name, symbol);

}

else {

errorFlag = -1;

errorCnt++;

}

if (locctr != 0) {

errorFlag = -1;

errorCnt++;

}

if (insFlag == 0) {

startAddr = 0;

locctr = 0;

}

else {

startAddr = strtol(inst, NULL, 16);

locctr = strtol(inst, NULL, 16);

}

}

if (strcmp(opcode, "END") != 0 && strcmp(opcode, "START") != 0 && endFlag == 0 && cmntFlag == 0) {

if (startFlag == 0) {

errorFlag = -1;

errorCnt++;

}

if (cmntFlag == 0) {

if (symFlag == 1) {

if (searchSym(symtab, symbol) == 1) {

errorFlag = -1;

errorCnt++;

}

else

insertSym(symtab, symbol, locctr);

}

int k = getFormat(mne, opcode, hashfunc(opcode));

if (k != -1) {

locctr += k;

if (format4Flag == 1)

locctr++;

}

else if (strcmp(opcode, "WORD") == 0) {

for (int i = 0; i < strlen(inst); i++) {

if (inst[i]<'0' || inst[i]>'9')

errorFlag = -1;

}

locctr += 3;

}

else if (strcmp(opcode, "RESW") == 0) {

for (int i = 0; i < strlen(inst); i++) {

if (inst[i]<'0' || inst[i]>'9')

errorFlag = -1;

}

locctr += (3 \* atoi(inst));

}

else if (strcmp(opcode, "RESB") == 0) {

for (int i = 0; i < strlen(inst); i++) {

if (inst[i]<'0' || inst[i]>'9')

errorFlag = -1;

}

locctr += atoi(inst);

}

else if (strcmp(opcode, "BYTE") == 0) {

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

locctr += (strlen(inst) - 3);

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

locctr += ((strlen(inst) - 3) / 2 + (strlen(inst) - 3) % 2);

else

errorFlag = -1;

}

else if (strcmp(opcode, "BASE") == 0)

strcpy(baseSym, inst);

else if (strcmp(opcode, "NOBASE") == 0) {}

else if (strcmp(opcode, "END") == 0) {}

else {

errorFlag = -1;

errorCnt++;

}

}

}

if (strcmp(opcode, "END") == 0) {

if (startFlag == 0) {

errorFlag = -1;

errorCnt++;

}

else

endFlag = 1;

}

if (errorFlag == -1)

printf("Error in line %d\n", linenum);

linenum += 5;

}

if (endFlag == 0) {

errorFlag = -1;

errorCnt++;

printf("Error in line %d\n", linenum);

}

if (errorCnt > 0) {

printf("\n");

return -1;

}

proglen = locctr - startAddr;

fclose(fp);

////////////////////////////

// Pass2 of the Assembler //

////////////////////////////

fp = fopen(fname, "r");

if (fp == NULL) {

printf("There is no such name of file...\n\n");

return -1;

}

char \*filelst = (char\*)malloc((strlen(fname) + 1) \* sizeof(char));

char \*fileobj = (char\*)malloc((strlen(fname) + 1) \* sizeof(char));

char \*textrecord = (char\*)malloc(70 \* sizeof(char));

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

for (int i = 0; i < 70; i++)

textrecord[i] = 0;

int textlen = 0, textAddr = startAddr, endAddr, endAddrFlag = 0;

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

strcpy(filelst, fname);

strcpy(fileobj, fname);

strcat(filelst, ".lst");

strcat(fileobj, ".obj");

fname[strlen(fname)] = '.';

FILE \*lst = fopen(filelst, "w");

if (lst == NULL) {

printf("File opening error...\n\n");

return -1;

}

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

if (obj == NULL) {

printf("File opening error...\n\n");

return -1;

}

linenum = 5; startFlag = 0; endFlag = 0;

while (fgets(str, 201, fp) != NULL) {

size = strlen(str); symFlag = 0; opFlag = 0; insFlag = 0; cmntFlag = 0; format4Flag = 0; errorFlag = 0;

int j = 0;

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

symbol[i] = 0;

opcode[i] = 0;

inst[i] = 0;

}

for (int i = 0; i < size; i++) {

if (str[i] == '\n' || str[i] == ' ' || str[i] == '\t') {

continue;

}

else if ((str[i] >= 'A' && str[i] <= 'Z') || (str[i] >= '0' && str[i] <= '9') || str[i] == '#' || str[i] == '@' || str[i] == '+' || str[i] == ',' || str[i] == '\'') {

if (str[i] == '+')

format4Flag = 1;

if (symFlag == 0 && opFlag == 0)

symbol[j] = str[i];

if (symFlag == 1 && opFlag == 0)

opcode[j] = str[i];

if (opFlag == 1 && insFlag == 0) {

inst[j] = str[i];

}

j++;

if (str[i + 1] == ' ' || str[i + 1] == '\t' || str[i + 1] == '\n') {

if (str[i] == ',')

continue;

if ((getOpcode(mne, symbol, hashfunc(symbol)) != -1 && opFlag == 0)

|| (getOpcode(mne, &symbol[1], hashfunc(&symbol[1])) != -1 && opFlag == 0)

|| strcmp(symbol, "START") == 0 || strcmp(symbol, "END") == 0

|| strcmp(symbol, "BASE") == 0 || strcmp(symbol, "NOBASE") == 0) {

strcpy(opcode, symbol);

for (int k = 0; k < 10; k++)

symbol[k] = 0;

opFlag = 1;

}

else {

if (opFlag == 1) {

insFlag = 1;

}

else if (symFlag == 1) {

opFlag = 1;

}

else

symFlag = 1;

}

j = 0;

}

}

else if (str[i] == '.') {

cmntFlag = 1;

break;

}

else if (j >= 10) {

errorFlag = -1;

break;

}

else {

errorFlag = -1;

break;

}

}

if (strcmp(opcode, "START") == 0) {

startFlag = 1;

if (strlen(symbol) <= 6) {

strcpy(name, symbol);

}

else

errorFlag = -1;

if (locctr != 0)

errorFlag = -1;

if (insFlag == 0) {

startAddr = 0;

locctr = 0;

}

else {

startAddr = strtol(inst, NULL, 16);

locctr = strtol(inst, NULL, 16);

}

int len = strlen(name);

for (int i = len; i < 6; i++)

name[i] = ' ';

fprintf(obj, "H%s%06X%06X\n", name, startAddr, proglen);

}

int objectcode;

if (endFlag == 0) {

if (cmntFlag == 0) {

int code, format, operAddr;

char operand[20] = { 0 };

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

code = getOpcode(mne, &opcode[1], hashfunc(&opcode[1]));

format = 4;

}

else {

code = getOpcode(mne, opcode, hashfunc(opcode));

format = getFormat(mne, opcode, hashfunc(opcode));

}

if (code != -1) {

if (endAddrFlag == 0) {

endAddr = locctr;

endAddrFlag = 1;

}

if (findSym(symtab, operand, inst) == 1)

operAddr = findSymAddr(symtab, operand);

else

operAddr = 0;

objectcode = makeObjectcode(symtab, locctr, format, code, inst, baseSym);

}

else if (strcmp(opcode, "BYTE") == 0 || strcmp(opcode, "WORD") == 0)

objectcode = makeObjectcode(symtab, locctr, 0, code, inst, baseSym);

else

objectcode = -1;

if (textlen == 0)

textAddr = locctr;

if (objectcode == -1) {

if (strcmp(opcode, "BASE") == 0 || strcmp(opcode, "NOBASE") == 0 || strcmp(opcode, "END") == 0) {

if (symFlag == 1)

fprintf(lst, "\t\t%s\t%s\t%s\n", symbol, opcode, inst);

else

fprintf(lst, "\t\t%s\t%s\n", opcode, inst);

if (strcmp(opcode, "END") == 0) {

if (textlen != 0) {

fprintf(obj, "T%06X%02X%s\n", textAddr, textlen, textrecord);

fprintf(obj, "E%06X\n", endAddr);

}

}

}

else {

if (symFlag == 1)

fprintf(lst, "%04X\t%s\t%s\t%s\n", locctr, symbol, opcode, inst);

else

fprintf(lst, "%04X\t\t%s\t%s\n", locctr, opcode, inst);

}

if (strcmp(opcode, "RESW") == 0 || strcmp(opcode, "RESB") == 0) {

if (textlen != 0) {

fprintf(obj, "T%06X%02X%s\n", textAddr, textlen, textrecord);

textlen = 0;

for (int i = 0; i < 70; i++)

textrecord[i] = 0;

}

}

}

else {

if (strcmp(opcode, "WORD") == 0) {

textlen += 3;

if (textlen > 30) {

textlen -= 3;

fprintf(obj, "T%06X%02X%s\n", textAddr, textlen, textrecord);

textlen = 0;

textAddr = locctr;

for (int i = 0; i < 70; i++)

textrecord[i] = 0;

textlen += 3;

}

hexTostring(objectcode, textobjcode, 6);

strcat(textrecord, textobjcode);

}

else if (strcmp(opcode, "BYTE") == 0) {

int tmp;

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

tmp = strlen(inst) - 3;

if (inst[0] == 'X')

tmp = (strlen(inst) - 3) / 2;

textlen += tmp;

if (textlen > 30) {

textlen -= tmp;

fprintf(obj, "T%06X%02X%s\n", textAddr, textlen, textrecord);

textlen = 0;

textAddr = locctr;

for (int i = 0; i < 70; i++)

textrecord[i] = 0;

textlen += tmp;

}

hexTostring(objectcode, textobjcode, tmp \* 2);

strcat(textrecord, textobjcode);

}

else {

textlen += format;

if (textlen > 30) {

textlen -= format;

fprintf(obj, "T%06X%02X%s\n", textAddr, textlen, textrecord);

textlen = 0;

textAddr = locctr;

for (int i = 0; i < 70; i++)

textrecord[i] = 0;

textlen += format;

}

if (format4Flag == 1) {

hexTostring(objectcode, textobjcode, 3);

strcat(textrecord, textobjcode);

int tempAddr;

if (operand[0] >= 'A' && operand[0] <= 'Z')

tempAddr = findSymAddr(symtab, operand);

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

tempAddr = atoi(operand);

hexTostring(tempAddr, textobjcode, 5);

strcat(textrecord, textobjcode);

}

else {

hexTostring(objectcode, textobjcode, format \* 2);

strcat(textrecord, textobjcode);

}

}

if (strlen(inst) < 8) {

if (strcmp(opcode, "BYTE") == 0 || strcmp(opcode, "WORD") == 0) {

if (symFlag == 1)

fprintf(lst, "%04X\t%s\t%s\t%s\t\t%02X\n", locctr, symbol, opcode, inst, objectcode);

else

fprintf(lst, "%04X\t\t%s\t%s\t\t%02X\n", locctr, opcode, inst, objectcode);

}

if (format == 1) {

if (symFlag == 1)

fprintf(lst, "%04X\t%s\t%s\t%s\t\t%02X\n", locctr, symbol, opcode, inst, objectcode);

else

fprintf(lst, "%04X\t\t%s\t%s\t\t%02X\n", locctr, opcode, inst, objectcode);

}

if (format == 2) {

if (symFlag == 1)

fprintf(lst, "%04X\t%s\t%s\t%s\t\t%04X\n", locctr, symbol, opcode, inst, objectcode);

else

fprintf(lst, "%04X\t\t%s\t%s\t\t%04X\n", locctr, opcode, inst, objectcode);

}

if (format == 3) {

if (symFlag == 1)

fprintf(lst, "%04X\t%s\t%s\t%s\t\t%06X\n", locctr, symbol, opcode, inst, objectcode);

else

fprintf(lst, "%04X\t\t%s\t%s\t\t%06X\n", locctr, opcode, inst, objectcode);

}

if (format == 4) {

int tempAddr;

if (operand[0] >= 'A' && operand[0] <= 'Z')

tempAddr = findSymAddr(symtab, operand);

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

tempAddr = atoi(operand);

if (symFlag == 1)

fprintf(lst, "%04X\t%s\t%s\t%s\t\t%03X%05X\n", locctr, symbol, opcode, inst, objectcode, tempAddr);

else

fprintf(lst, "%04X\t\t%s\t%s\t\t%03X%05X\n", locctr, opcode, inst, objectcode, tempAddr);

}

}

else {

if (strcmp(opcode, "BYTE") == 0 || strcmp(opcode, "WORD") == 0) {

if (symFlag == 1)

fprintf(lst, "%04X\t%s\t%s\t%s\t%02X\n", locctr, symbol, opcode, inst, objectcode);

else

fprintf(lst, "%04X\t\t%s\t%s\t%02X\n", locctr, opcode, inst, objectcode);

}

if (format == 1) {

if (symFlag == 1)

fprintf(lst, "%04X\t%s\t%s\t%s\t%02X\n", locctr, symbol, opcode, inst, objectcode);

else

fprintf(lst, "%04X\t\t%s\t%s\t%02X\n", locctr, opcode, inst, objectcode);

}

if (format == 2) {

if (symFlag == 1)

fprintf(lst, "%04X\t%s\t%s\t%s\t%04X\n", locctr, symbol, opcode, inst, objectcode);

else

fprintf(lst, "%04X\t\t%s\t%s\t%04X\n", locctr, opcode, inst, objectcode);

}

if (format == 3) {

if (symFlag == 1)

fprintf(lst, "%04X\t%s\t%s\t%s\t%06X\n", locctr, symbol, opcode, inst, objectcode);

else

fprintf(lst, "%04X\t\t%s\t%s\t%06X\n", locctr, opcode, inst, objectcode);

}

if (format == 4) {

int tempAddr;

if (operand[0] >= 'A' && operand[0] <= 'Z')

tempAddr = findSymAddr(symtab, operand);

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

tempAddr = atoi(operand);

if (symFlag == 1)

fprintf(lst, "%04X\t%s\t%s\t%s\t%03X%05X\n", locctr, symbol, opcode, inst, objectcode, tempAddr);

else

fprintf(lst, "%04X\t\t%s\t%s\t%03X%05X\n", locctr, opcode, inst, objectcode, tempAddr);

}

}

}

}

}

if (strcmp(opcode, "END") != 0 && strcmp(opcode, "START") != 0 && endFlag == 0 && cmntFlag == 0) {

if (startFlag == 0)

errorFlag = -1;

if (cmntFlag == 0) {

int k;

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

k = getFormat(mne, &opcode[1], hashfunc(&opcode[1]));

else

k = getFormat(mne, opcode, hashfunc(opcode));

if (k != -1) {

locctr += k;

if (format4Flag == 1)

locctr++;

}

else if (strcmp(opcode, "WORD") == 0)

locctr += 3;

else if (strcmp(opcode, "RESW") == 0)

locctr += (3 \* atoi(inst));

else if (strcmp(opcode, "RESB") == 0)

locctr += atoi(inst);

else if (strcmp(opcode, "BYTE") == 0) {

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

locctr += (strlen(inst) - 3);

if (inst[0] == 'X')

locctr += ((strlen(inst) - 3) / 2 + (strlen(inst) - 3) % 2);

}

else if (strcmp(opcode, "BASE") == 0) {}

else if (strcmp(opcode, "NOBASE") == 0) {}

else if (strcmp(opcode, "END") == 0) {}

else

errorFlag = -1;

}

}

if (strcmp(opcode, "END") == 0) {

if (startFlag == 0)

errorFlag = -1;

else

endFlag = 1;

}

linenum += 5;

}

printf("output file: [%s], [%s]\n\n", filelst, fileobj);

free(textrecord); // free

free(textobjcode);

free(opcode);

free(filelst);

free(fileobj);

fclose(fp);

fclose(lst);

fclose(obj);

return 1;

}

int main() {

char \*str;

unsigned char \*memory = (unsigned char\*)malloc(0x100000 \* sizeof(unsigned char));

int sAddr = 0, addr[3], assembleFlag = 0;

HashTable \*\*mnemonic = (HashTable\*\*)malloc(20 \* sizeof(HashTable\*));

for (int a = 0; a < 0x100000; a++)

memory[a] = 0x00;

for (int a = 0; a < 20; a++) {

mnemonic[a] = (HashTable\*)malloc(sizeof(HashTable));

mnemonic[a] = NULL;

}

SymTab \*\*symtab;

makeHashTable(mnemonic);

head = NULL;

while (1) { // program progress

printf("sicsim> ");

char \*command = (char\*)malloc(500 \* sizeof(char));

scanf("%[^\n]", command);

str = command;

//////////////////////////////////////////////////////////// erasing indent, space character from both left and right side of the string

if (command[0] == ' ' || command[0] == '\t') {

int i = 1;

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

i++;

command = &command[i];

}

if (command[strlen(command) - 1] == ' ' || command[strlen(command) - 1] == '\t') {

int i = (int)strlen(command) - 2;

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

i--;

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

}

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

if (strcmp(command, "h") == 0 || strcmp(command, "help") == 0) { // h[elp]

addNode(str);

help();

}

else if (strcmp(command, "q") == 0 || strcmp(command, "quit") == 0) // q[uit]

break;

else if (strcmp(command, "d") == 0 || strcmp(command, "dir") == 0) {

addNode(str);

dir();

}

else if (strcmp(command, "hi") == 0 || strcmp(command, "history") == 0) { // hi[story]

addNode(str);

history();

}

else if (command[0] == 'd' && command[1] == 'u') { // du[mp]

if (command[2] == 'm' && command[3] == 'p') {

if (command[4] == ' ' || command[4] == '\0' || command[4] == '\t') {

if (command[4] == ' ' || command[4] == '\t') { // dump A,B

if (TwoInputsString(&command[5], &addr[0], &addr[1]) == 1) {

if (addr[0] >= 0x00000 && addr[0] <= 0xFFFFF) {

if (addr[1] > 0xFFFFF)

addr[1] = 0xFFFFF;

addNode(str);

sAddr = dump(memory, addr[0], addr[1]) + 1;

}

else

printf("\nWrong Command!!!(input is outside the bounds)\n\n");

}

}

else { // just dump

if (sAddr < 0x100000) {

addNode(str);

sAddr = dump(memory, sAddr, -1) + 1;

}

else

printf("\nWrong Command!!! (start address is out of range(0 ~ FFFFF))\n\n");

}

}

else

printf("\nWrong Command!!!\n\n");

}

else if (command[2] == ' ' || command[2] == '\0' || command[2] == '\t') {

if (command[2] == ' ' || command[2] == '\t') { // du A,B

if (TwoInputsString(&command[3], &addr[0], &addr[1]) == 1) {

if (addr[0] >= 0x00000 && addr[0] <= 0xFFFFF) {

if (addr[1] > 0xFFFFF)

addr[1] = 0xFFFFF;

addNode(str);

sAddr = dump(memory, addr[0], addr[1]) + 1;

}

else

printf("\nWrong Command!!!(input is outside the bounds)\n\n");

}

}

else { // just du

if (sAddr < 0x100000) {

addNode(str);

sAddr = dump(memory, sAddr, -1) + 1;

}

else

printf("\nWrong Command!!! (start address is out of range(0 ~ FFFFF))\n\n");

}

}

else

printf("\nWrong Command!!!\n\n");

}

else if (command[0] == 'e') { // e[dit]

if (command[1] == 'd' && command[2] == 'i' && command[3] == 't') {

if (command[4] == ' ' || command[4] == '\t') { // edit A,B

if (TwoInputsString(&command[5], &addr[0], &addr[1]) == 1) {

if (addr[0] >= 0x00000 && addr[0] <= 0xFFFFF && addr[1] >= 0x00 && addr[1] <= 0xFF) {

addNode(str);

memory[addr[0]] = addr[1];

printf("\n");

}

else

printf("\nWrong Command!!!(input is outside the bounds)\n\n");

}

}

else

printf("\nWrong Command!!!\n\n");

}

else if (command[1] == ' ' || command[1] == '\0' || command[1] == '\t') {

if (command[1] == ' ' || command[1] == '\t') { // e A,B

if (TwoInputsString(&command[2], &addr[0], &addr[1]) == 1) {

if (addr[0] >= 0x00000 && addr[0] <= 0xFFFFF && addr[1] >= 0x00 && addr[1] <= 0xFF) {

addNode(str);

memory[addr[0]] = addr[1];

printf("\n");

}

else

printf("\nWrong Command!!!(input is outside the bounds)\n\n");

}

}

else

printf("\nWrong Command!!!\n\n");

}

else

printf("\nWrong Command!!!\n\n");

}

else if (command[0] == 'f') { // f[ill]

if (command[1] == 'i' && command[2] == 'l' && command[3] == 'l') {

if (command[4] == ' ' || command[4] == '\t') { // fill A,B,C

if (ThreeInputsString(&command[5], &addr[0], &addr[1], &addr[2]) == 1) {

if (addr[0] >= 0x00000 && addr[0] <= 0xFFFFF && addr[1] >= 0x00000 && addr[1] <= 0xFFFFF && addr[2] >= 0x00 && addr[2] <= 0xFF) {

addNode(str);

for (int j = addr[0]; j <= addr[1]; j++)

memory[j] = addr[2];

printf("\n");

}

else

printf("\nWrong Command!!!(input is outside the bounds)\n\n");

}

}

else

printf("\nWrong Command!!!\n\n");

}

else if (command[1] == ' ' || command[1] == '\0' || command[1] == '\t') {

if (command[1] == ' ' || command[1] == '\t') { // f A,B,C

if (ThreeInputsString(&command[2], &addr[0], &addr[1], &addr[2]) == 1) {

if (addr[0] >= 0x00000 && addr[0] <= 0xFFFFF && addr[1] >= 0x00000 && addr[1] <= 0xFFFFF && addr[2] >= 0x00 && addr[2] <= 0xFF) {

addNode(str);

for (int j = addr[0]; j <= addr[1]; j++)

memory[j] = addr[2];

printf("\n");

}

else

printf("\nWrong Command!!!(input is outside the bounds)\n\n");

}

}

else

printf("\nWrong Command!!!\n\n");

}

else

printf("\nWrong Command!!!\n\n");

}

else if (strcmp(command, "reset") == 0) { // reset (make all values in the memory 0)

addNode(str);

for (int j = 0x00000; j <= 0xFFFFF; j++)

memory[j] = 0;

printf("\n");

}

else if (strcmp(command, "opcodelist") == 0) { // print out opcodelist by hashtable

addNode(str);

printf("\n");

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

printf("%d : ", i);

HashTable \*temp;

temp = mnemonic[i];

while (temp != NULL) {

printf("[%s, %02X]", temp->mne, temp->opcode);

if (temp->link != NULL)

printf(" -> ");

temp = temp->link;

}

printf("\n");

}

printf("\n");

}

else if (command[0] == 'o' && command[1] == 'p' && command[2] == 'c' && command[3] == 'o' && command[4] == 'd' && command[5] == 'e' && (command[6] == ' ' || command[6] == '\t')) { // opcode command

int i = 6;

//////////////////////////////////////////////////////////// erasing indent, space character from both left and right side of the string

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

i++;

command = &command[i];

if (command[strlen(command) - 1] == ' ' || command[strlen(command) - 1] == '\t') {

int i = (int)strlen(command) - 2;

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

i--;

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

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

i = getOpcode(mnemonic, command, hashfunc(command));

if (i != -1) {

addNode(str);

printf("opcode is %02X.\n\n", i);

}

else

printf("Invalid mnemonic... opcode doesn't exist!\n\n");

}

else if (command[0] == 't' && command[1] == 'y' && command[2] == 'p' && command[3] == 'e' && (command[4] == ' ' || command[4] == '\t')) {

int i = 4;

//////////////////////////////////////////////////////////// erasing indent, space character from both left and right side of the string

addNode(str);

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

i++;

command = &command[i];

if (command[strlen(command) - 1] == ' ' || command[strlen(command) - 1] == '\t') {

int i = (int)strlen(command) - 2;

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

i--;

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

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

displayFile(command); // displays content of a file

}

else if (command[0] == 'a' && command[1] == 's' && command[2] == 's' && command[3] == 'e' && command[4] == 'm' && command[5] == 'b' && command[6] == 'l' && command[7] == 'e' && (command[8] == ' ' || command[8] == '\t')) {

int i = 8;

addNode(str);

if (assembleFlag == 1)

freeSym(symtab);

symtab = (SymTab\*\*)malloc(26 \* sizeof(SymTab\*));

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

symtab[i] = (SymTab\*)malloc(sizeof(SymTab));

symtab[i] = NULL;

}

//////////////////////////////////////////////////////////// erasing indent, space character from both left and right side of the string

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

i++;

command = &command[i];

if (command[strlen(command) - 1] == ' ' || command[strlen(command) - 1] == '\t') {

int i = (int)strlen(command) - 2;

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

i--;

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

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

assembleFlag = assembleFile(command, mnemonic, symtab);

if (assembleFlag != 1)

freeSym(symtab);

}

else if (strcmp(command, "symbol") == 0) {

addNode(str);

if (assembleFlag == 1)

printSym(symtab);

else

printf("Error: Not assembled properly (needs to be assembled first)\n\n");

}

else

printf("\nWrong Command!!!\n\n");

getchar();

free(str);

}

if (assembleFlag == 1)

freeSym(symtab);

free(memory); // free

for (int a = 0; a < 20; a++) {

HashTable \*temp;

while (mnemonic[a] != NULL) {

temp = mnemonic[a];

mnemonic[a] = mnemonic[a]->link;

free(temp);

}

free(mnemonic[a]);

}

free(mnemonic);

while (head != NULL) {

Node \*temp = head;

head = head->link;

free(temp);

}

return 0;

}