**과목명: 시스템프로그래밍**

**1 분반**

**<<Project #1>>**

**서강대학교 컴퓨터공학과**

**[학번] 20161643**

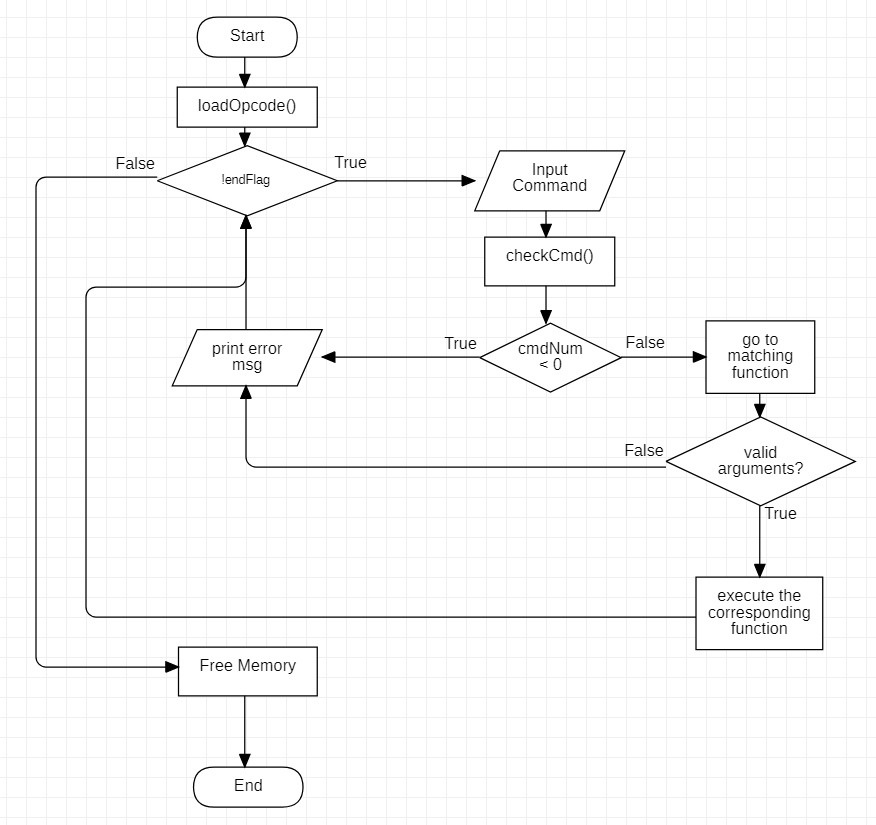
**[이름] 정진원**

목 차

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
3. **모듈 정의**
   1. 모듈이름: main()
      1. 기능
      2. 사용변수
   2. 모듈이름: hextodec(char cmd[], int start, int end)
      1. 기능
      2. 사용변수
   3. 모듈이름: createHash(char mnec[])
      1. 기능
      2. 사용변수
   4. 모듈이름: checkCmd(char cmd[], char tcmd[])
      1. 기능
      2. 사용변수
   5. 모듈이름: addHistory(char cmd[])
      1. 기능
      2. 사용변수
   6. 모듈이름: cmdHelp(char cmd[])
      1. 기능
      2. 사용변수
   7. 모듈이름: cmdDir(char cmd[])
      1. 기능
      2. 사용변수
   8. 모듈이름: cmdQuit(char cmd[])
      1. 기능
      2. 사용변수
   9. 모듈이름: cmdHistory(char cmd[])
      1. 기능
      2. 사용변수
   10. 모듈이름: cmdDump(char cmd[])
       1. 기능
       2. 사용변수
   11. 모듈이름: cmdEdit(char cmd[])
       1. 기능
       2. 사용변수
   12. 모듈이름: cmdFill(char cmd[])
       1. 기능
       2. 사용변수
   13. 모듈이름: cmdReset(char cmd[])
       1. 기능
       2. 사용변수
   14. 모듈이름: loadOpcode()
       1. 기능
       2. 사용변수
   15. 모듈이름: cmdOpcode(char cmd[])
       1. 기능
       2. 사용변수
   16. 모듈이름: cmdOpcodelist(char cmd[])
       1. 기능
       2. 사용변수
4. **전역 변수, 구조체, typedef 정의**
   1. #define FALSE 0
   2. #define TRUE 1
   3. typedef void(\*funcPtr)(char \*)
   4. typedef struct \_History
   5. typedef struct \_opNode
   6. bool endFlag
   7. bool errFlag
   8. History \*head, \*tail
   9. unsigned char memory[1048576]
   10. int dumpEnd = 0
   11. opNode\* opList[20]
5. **코드 설명**
   1. 20161643.h
   2. 20161643.c
6. **프로그램 개요**

SIC/XE 머신을 구현하기 위한 전 단계로 이 프로그램을 작성하였습니다. Shell의 기본적인 명령어들을 수행할 수 있으며, 이후 어셈블러, 링커, 로더를 실행 할 것입니다. 컴파일을 통해 만들어진 object 코드가 적재되고, 실행될 메모리 공간과 mnemonic을 opcode 값으로 변환하는 opcode table을 구현하였습니다.

1. **프로그램 설명**
   1. **프로그램 흐름도**



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

Shell을 실행하고, endFlag 가 True로 설정되기 전까지 사용자에게 입력을 받아 입력이 유효한 경우, 그에 따른 기능을 실행한다. Shell이 종료되면 동적 할당한 메모리를 전부 free 해준다.

* + 1. ***사용변수***

char command[100] – 사용자에게 입력 받은 command를 저장하는 배열

char trimCmd[100] – 사용자에게 입력 받은 command에서 앞과 뒤의 공백을 제거한 문자열을 저장하는 배열

int cmdNum – 유효한 command가 입력되었을 경우 그에 해당하는 숫자를 저장하는 변수. 유효하지 않은 입력의 경우 -1을 값으로 가진다.

History\* temp – liked list를 deallocate할 때 이용되는 포인터

int i – loop용 변수

opNode \*curr, \*prev – opcode table을 deallocate할 때 이용되는 포인터

funcPtr fPtr[] – 함수 포인터로 command를 실행하는 함수들을 묶어서 가리킨다. cmdNum에 해당하는 함수를 실행시킨다

* 1. **모듈이름: hextodec(char cmd[], int start, int end)**
     1. ***기능***

Command 및 command에서 16진수가 존재하는 부분을 전달받은 뒤, 16진수를10진수로 변환하여 준다. 16진수 표기를 벗어나는 문자가 있는 경우 errFlag를 True로 set하여 error message를 출력할 수 있게 해준다. 또한, 16진수 수가 시작되기 이전의 공백을 무시해준다.

* + 1. ***사용변수***

int i – 반복문을 위한 변수

int dec – 16진수가 변환된 10진수 수

int num - 16진수를 10진수로 변환하기 위해 16^n을 나타내는 수

bool startFlag – 16진수가 시작되기 이전의 공백들을 무시하기 위한 flag이다. 초기값은 False이며 16진수가 시작되면 True로 바뀌어 16진수 숫자 사이의 공백은 무시하지 않게 해준다. 공백을 \_로 표현한다고 가정하고 dump\_40\_\_\_,\_60이라는 입력이 들어온다고 치자. 변환될 첫 번째 주소는 40\_\_\_인데, 16진수의 1의 자릿수부터 확인하기 위해 뒤 index부터 앞으로 확인을 한다. 이때 0이전의 \_\_\_는 startFlag가 False이기 때문에 무시된다. 만약 4\_0\_\_\_같은 주소 값이 전달되면, 0에서 startFlag가 True로 set되어 4와 0사이의 공백을 무시하지 않게 되어 주소에 오류가 있다고 출력된다.

char cmd[]- 전달된 명령어

int start – 인수가 시작되는 index

int end – 인수의 끝을 찾기 위해 문자열 끝, 혹은 comma를 가리키는 index

* 1. **모듈이름: createHash(char mnec[])**
     1. ***기능***

opcode.txt에서 읽어온 instruction을 size가 20인 hash table에 저장하기 위한 index를 생성하는 hash function. 전해진 인자의 각 원소가 ‘A’로부터 얼마나 떨어져 있는지 계산한 뒤 소수 41을 곱해 result에 더해준다. Result에 table size인 20으로 나눈 나머지를 반환하였다.

* + 1. ***사용변수***

int i – 반복문 용 변수

int result – 반환할 결과를 저장하는 변수

char mnec[] – mnemonic을 저장하고 있는 문자열

* 1. **모듈이름: checkCmd(char cmd[], char tcmd[])**
     1. ***기능***

입력된 command에서 앞과 뒤의 공백을 제거해준 뒤 tcmd 배열에 저장하고, 명령어 이외에 추가적인 인자가 있으면 명령어만 temp 배열에 저장해준다. 이 배열을 비교하여, 유효한 명령어가 입력된 경우 이에 맞는 숫자를 반환해준다. 잘못된 입력이 들어온 경우 -1을 반환한다.

* + 1. ***사용변수***

int i, j, k, l – loop 용 변수

char temp[100] – command와 인수가 입력되면, command 부분만 저장하는 배열

char cmd[] – 입력된 문장 전체

char tcmd[] – 앞과 뒤의 공백이 지워진 문장

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

유효한 command인 경우에 이 함수를 호출하여, cmd 문자열에 담겨서 넘어온 문장을 history linked list에 추가한다.

* + 1. ***사용변수***

History \*new – linked list에 추가할 node를 동적 할당하기 위한 변수

char cmd[] – 전달된 명령어

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

getCmd에서 명령어가 h/help인 경우에 호출되는 함수이다. 사용할 수 있는 명령어 목록을 출력한다. 수행된 이후 history에 명령어를 추가한다.

* + 1. ***사용변수***

char cmd[] – 전달된 명령어

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

getCmd에서 명령어가 d/dir인 경우에 호출되는 함수이다. 본 프로그램 실행 파일이 위치한 directory의 파일들을 출력한다. opendir(), readdir() 함수를 이용해 파일에 접근하고, structure stat내의 st\_mode와 S\_ISDIR와 S\_IXSUSR로 파일에 대한 정보를 알아내어 실행파일 뒤에는 \*을, directory 뒤에는 /을 붙여준다. 수행된 이후 history에 명령어를 추가한다.

* + 1. ***사용변수***

DIR \*dir – 현재 directory의 정보를 담은 DIR 포인터

struct dirent \*file - 현재 directory에 존재하는 파일의 정보를 담은 struct dirent의 포인터.

struct stat cStat - 현재 directory에 존재하는 파일의 정보를 담은 struct stat형 변수. \*file이 가리키는 것이 무엇인지 알려주는 정보를 담은 element가 존재한다.

int prtNum – 출력 형식을 맞추기 위한 변수

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

getCmd에서 명령어가 q/quit인 경우 호출되는 함수. endFlag을 True로 set 하여 프로그램을 종료시킨다.

* + 1. ***사용변수***

없음

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

getCmd에서 명령어가 hi/history인 경우 호출되는 함수. History linked list를 head부터 순차적으로 따라가며 저장된 history에 저장된 유효한 명령어들을 순서대로 출력해주는 함수. 수행된 이후 history에 명령어를 추가한다.

* + 1. ***사용변수***

History\* temp – linked list을 따라가기 위한 포인터

int count – command 좌측에 몇 번째 history인지를 표현해주는 숫자를 저장하기 위한 변수

char cmd[] – 전달된 명령어

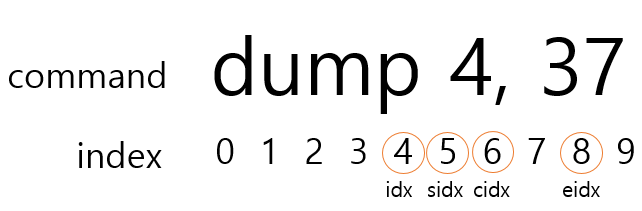
* 1. **모듈이름: cmdDump(char cmd[])**
     1. ***기능***

getCmd에서 명령어가 du/dump인 경우 호출되는 함수. 구현된 가상 메모리 공간에서 주어진 범위만큼의, 주어진 시작 주소 + 160만큼의, 혹은 마지막으로 출력한 주소 + 1 ~ 마지막으로 출력한 주소 + 160까지의 메모리 공간만큼을 화면상에 출력하는 함수. 좌측에는 시작 주소를, 중간에는 16개의 메모리에 저장된 값을, 우측에는 그에 해당하는 ASCII 코드 값을 출력한다. 쉼표가 있는데, 뒤에 주소가 입력되지 않은 경우, 주소 값으로 16진수 범위를 벗어나는 문자가 입력된 경우, 시작 주소가 끝 주소보다 큰 경우, 주소가 범위 밖을 벗어나는 경우 에러 처리를 하였다. 에러가 발생하지 않았다면 이후 history에 명령어를 추가한다.

* + 1. ***사용변수***

char cmd[] – 전달된 명령어

int idx, cidx, sidx, eidx – 전달된 명령어에서 각각 명령어 부분이 끝나는 index, 쉼표의 위치를 저장하는 index, 시작 주소가 시작되는 index, 끝 주소가 시작되는 index. 예시는 다음과 같다.



int start, end – 메모리의 시작과 끝을 나타내는 변수

int i, j – loop용 변수

int cmdLen; - 명령어의 길이를 저장하는 변수

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

getCmd에서 명령어가 e/edit인 경우 호출되는 함수. 구현한 가상메모리상의 공간에서 주어진 메모리의 값을 특정 값(0x00~0xFF)로 수정해주는 함수. 쉼표가 없어서 주어진 인자가 2개가 아닌 경우, 쉼표는 있지만 이후에 값이 존재하지 않는 경우, 16진수 범위를 벗어나는 문자가 입력된 경우, 주소와 값이 범위를 벗어나는 경우 에러 처리를 하였다. 에러가 발생하지 않았다면 이후 history에 명령어를 추가한다.

* + 1. ***사용변수***

char cmd[] – 전달된 명령어

int idx, cidx, aidx, vidx – 전달된 명령어에서 각각 명령어 부분이 끝나는 index, 쉼표의 위치를 저장하는 index, 수정될 주소 값이 시작되는 index, value가 시작되는 index

int address, value – 수정 될 주소 값과, 수정 될 값을 저장하는 변수

int i – loop용 변수

int cmdLen – 명령어의 길이를 저장하는 변수

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

getCmd에서 명령어가 f/fill인 경우 호출되는 함수. 구현된 가상메모리에서 주어진 범위 안의 값을 특정한 값으로 설정하는 함수. 주어진 인자가 세 개가 아닌 경우(쉼표가 2개가 아닌 경우), 명령어가 쉼표로 끝날 경우(인자가 부족한 경우), 16진수 범위를 벗어나는 문자가 입력으로 들어온 경우, 주소 값과 value가 범위를 벗어나는 경우, 시작 주소가 끝 주소보다 큰 경우 에러 처리를 하였다. 에러가 발생하지 않았다면 이후 history에 명령어를 추가한다.

* + 1. *사용변수*

char cmd[] – 전달된 명령어

int idx, cidx1, cidx2, sidx, eidx, vidx – 전달된 명령어에서 각각 명령어 부분이 끝나는 index, 첫 번째 쉼표의 위치를 저장하는 index, 두 번째 쉼표의 위치를 저장하는 index, 시작 주소가 시작되는 index, 범위의 끝 주소가 시작되는 index, value가 시작되는 index이다.

int start, end, value – 시작 주소, 끝 주소, value 값을 저장하는 변수

int i – loop용 변수

int cmdLen – 명령어의 길이를 저장하는 변수

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

getCmd에서 명령어가 reset인 경우 호출되는 함수. 구현된 가상 메모리 내의 모든 값을 0으로 초기화 시킨다. 이후 history에 명령어를 추가한다.

* + 1. ***사용변수***

char cmd[] – 전달된 명령어

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

프로그램이 실행되면 opcode table을 생성하기 위해 실행되는 함수이다. opcode.txt에서 opcode, mnemonic, format의 정보를 읽어서 opNode 구조체에 저장하고, 이를 hash table 형태로 저장한다. opcode.txt 파일이 존재하지 않는 경우 에러 처리를 하였다.

* + 1. ***사용변수***

FILE\* fp – opcode.txt를 읽어오기 위한 file 포인터

char mnemonic[6], format[4] – opcode.txt에 저장된 mnemonic과 format을 저장하기 위한 배열

int opcode, idx – opcode를 저장하기 위한 변수, createHash를 통해 얻은 index를 저장하기 위한 변수

opNode \*new, \*temp – 새로운 opNode를 할당하기 위한 포인터, opcode table 안의 node를 가리키는 포인터

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

getCmd에서 명령어가 opcode인 경우 호출되는 함수. Mnemonic이 존재하는 경우 해당 opcode를 화면에 출력한다. Mnemonic이 전달되지 않은 경우나, 없는 mnemonic이 전달된 경우 에러 처리를 해주었다. 이후 history에 명령어를 추가한다.

* + 1. ***사용변수***

char cmd[] – 전달된 명령어

int i, j, idx, cmdLen – loop용 변수, 명령어가 끝나는 index를 저장하는 변수, 명령어의 길이를 저장하는 변수d

opNode\* curr – opcode table안의 node를 가리키는 포인터

int hash – 전달된 mnemonic의 hash code를 저장하는 변수

char instruction[10] – 전달된 mnemonic을 저장하는 배열

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

getCmd에서 명령어가 opcodelist인 경우 호출되는 함수. Opcode table을 출력한다. 이후 history에 명령어를 추가한다.

* + 1. ***사용변수***

char cmd[] – 전달된 명령어

int i – loop용 변수

int count – opcode table 한 줄에 몇 개의 원소를 출력했는지 저장하는 변수

opNode\* curr – opcode table안의 node를 가리키는 포인터

1. **전역 변수, 구조체, typedef 정의**
   1. #define FALSE 0

FALSE를 0으로 정의한다.

* 1. #define TRUE 1

TRUE를 1로 정의한다.

* 1. typedef void(\*funcPtr)(char \*)

위의 함수들을 function pointer로 묶어서 main()함수를 정리하기 위해 정의

* 1. typedef struct \_History

history를 linked list로 구현하기 위해 선언한 구조체. Element로는 command를 저장하는 char type array와 다음 node를 연결할 link가 있다.

* 1. typedef struct \_opNode

opcode list를 구현하기 위한 구조체이다. Element로는 mnemonic과 format을 저장하는 char type array와 다음 node를 연결할 link가 있다.

* 1. bool endFlag

q/quit이 입력되어 프로그램이 종료되어야 할 때를 알려주기 위해 선언한 flag. 초기값은 False이며 cmdQuit()함수가 실행되면 True로 set되어 프로그램이 종료된다.

* 1. bool errFlag

16진수를 10진수로 변환할 때 16진수 범위를 벗어나는 문자가 입력되었는지 확인해주는 flag. 범위 밖의 입력이 들어온 경우 True로 set 된다.

* 1. History \*head, \*tail

History command 구현을 위해 선언한 포인터. Head는 history linked list의 처음을 가리키며, tail은 마지막 node를 가리킨다. 프로그램 시작 시 NULL로 초기화된다.

* 1. unsigned char memory[1048576]

가상 메모리를 위해 1MB의 배열을 만들었다. 한 cell당 표현할 수 있는 범위는 0x00~0xFF이기 때문에 unsigned char형 변수 하나로 표현될 수 있으며, 이를 이용하여 2^20개의 unsigned char를 가지는 배열을 선언하였다.

* 1. int dumpEnd

dump command를 구현 할 때, 마지막으로 출력된 address + 1의 값을 저장하기 위한 전역 변수

* 1. opNode\* opList[20]

hash table 구현을 위해 선언한 배열.

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>

#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

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

// 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]; //

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++;

}

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';

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

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");

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) { // 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 show the number of the opcode mnemonic

void cmdOpcode(char cmd[]) {

int i, j = 0, idx = 0, cmdLen = 0;

opNode\* curr;

int hash;

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

cmdLen = strlen(cmd);

while (1) {

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

break;

idx++;

} // where the command ends

if (idx == cmdLen) {

printf("Mnemonic is required\n");

return;

} // if there isn't a mnemonic, error

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

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

instruction[j] = cmd[i];

j++;

}

} // copy the mnemonic to array instruction

hash = createHash(instruction);

curr = opList[hash];

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

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);

}

* 1. **2016143.c**

1. #include "20161643.h"
2. int main() {
3. char command[100]; // array to save command
4. char trimCmd[100]; // 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. funcPtr fPtr[] = { cmdHelp, cmdDir, cmdQuit, cmdHistory, cmdDump, cmdEdit, cmdFill, cmdReset, cmdOpcode, cmdOpcodelist}; // function pointer to reduce the code
10. loadOpcode(); // load opcode list from opcode.txt
11. while (!endFlag) {
12. memset(command, '\0', sizeof(command));
13. memset(trimCmd, '\0', sizeof(trimCmd));
14. printf("sicsim>");
15. fgets(command, sizeof(command), stdin);
16. command[strlen(command) - 1] = '\0';
17. cmdNum = checkCmd(command, trimCmd); // check the command number
18. if (cmdNum == -1)
19. printf("Invalid Command\n");
20. else {
21. fPtr[cmdNum](trimCmd); // call the function that has the right command number
22. }
23. }
24. while(head){ // free history list
25. temp = head;
26. head = head->link;
27. free(temp);
28. }
29. for(i=0; i < 20; i++){ // free opcode list
30. curr = opList[i];
31. while(curr){
32. prev = curr;
33. curr = curr->link;
34. free(prev);
35. }
36. }
37. return 0;
38. }