과목명**:** 시스템프로그래밍

1분반 박운상 교수님

**<<Project #1>>**

서강대학교 **[**학부명:컴퓨터공학과**]**

**[**학번:20161619**]**

**[**이름:이세진**]**

Contents

1. Program Summary
2. Program Description

2-1 Flowchart

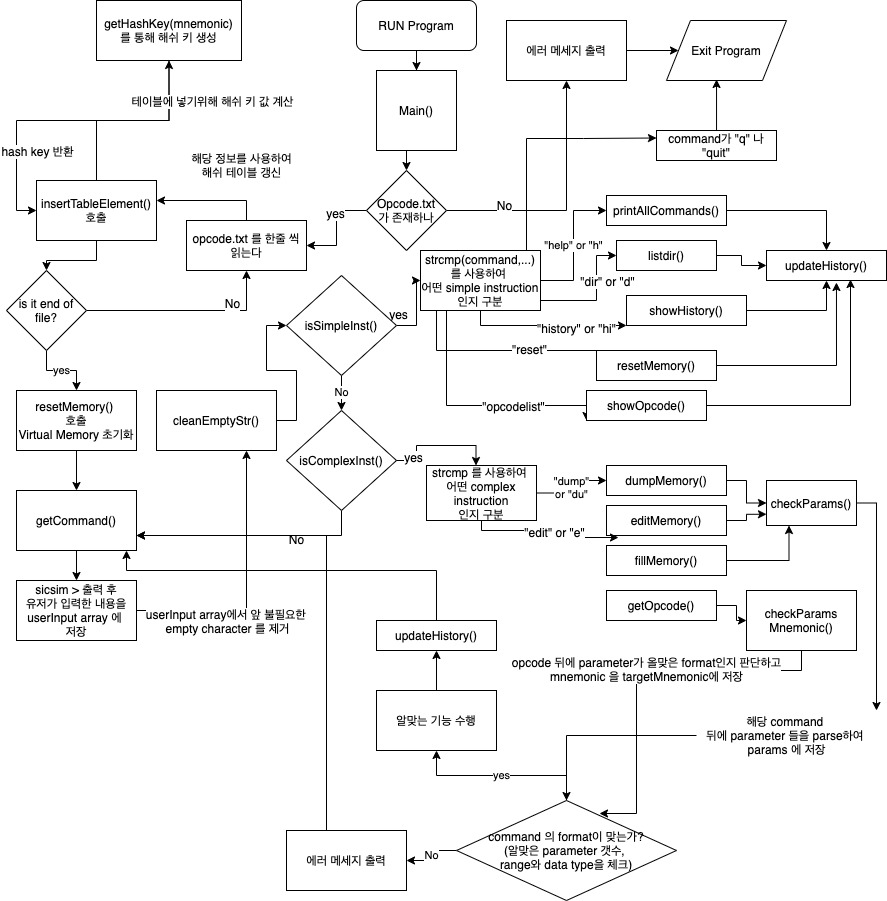
1. Modules
2. Global Variables
3. Code Description

**1. Program Summary**

This program is a step towards creating the SIC/XE machine that will be implemented in the future. This program is a shell prompt that will be later used by assembler, linker and loader. The task was to implement associated commands that will display and alter the contents of the memory, return the opcode of specific mnemonics and etc.

**2. Program Description**

***2.1. Program Flowchart***



**3. User-defined Modules**

***3.1 void updateHistory(char \*command)***

This function is called after a successful command input from the user. It inserts a new history node into the history linked list.

***3.2 void showHistory();***

displays the history linked list, showing all the previous working commands from the user

***3.3 int listdir(const char \*path);***

prints out the files in the current path. Folders and output files can be identified with '/' and '\*' added at the end of the filename.

***3.4 void printAllCommands();***

prints out all the commands working in the shell prompt

***3.5 void cleanEmptyStr();***

removes any empty characters in the userInput array before the first non-empty character. This function is used to ignore empty spaces in the command

***3.6 int isSimpleInst();***

Compares the userInput with the list of simple instructions(help,dir,history,reset,opcodelist and quit). Simple instructions are instructions that require no extra parameters. Returns 1 if the user input is one of the single instructions. ), returns 0 otherwise.

***3.7 int isComplexInst();***

Compares the userInput with the list of complex instructions(dump,edit,fill,opcode). Complex instructions are instructions that require extra parameters. Returns 1 if the user input is one of complex instructions and returns 0 otherwise.

***3.8 int isEmpty(char c);***

returns 1 if the character c is empty (space, tab or null), returns 0 otherwise

***3.9 int isHexadecimal(char c);***

returns 1 if the character is in hexadecimal, returns 0 otherwise

***3.10 int checkParams();***

checks the number of parameters in the user input and stores each into params array. Stores the number of parameters in the global variable numOfParams, which is later used to identify wrong commands in the program. Returns 0 if parameter is in the wrong format for the specific command. Returns 1 if parameter is in the right format

***3.12 int checkParamsMnemonic();***

checks the format of the parameter for opcode command. Parses mnemonic from user input and stores it to global variable, targetMnemonic. Returns 0 if mnemonic is not in capital letter or exceeds maximum length.

***3.13 void printMem(long start, long end);***

prints out the contents of virtual memory from start address to end address

***3.14 int dumpMemory();***

prints out the contents of virtual memory. If no extra parameter is given, it prints out 160 from defaultStartAddr. If one parameter is given(start address), it prints out 160 from the start address. If two parameters are given(start address and end address), it prints out from start to end address. Returns 0 if command is given in the wrong format. Returns 1 if the command worked successfully

***3.15 int editMemory();***

changes the memory at the given address to the given value. Returns 1 if successful and returns 0 if unsuccessful

due to wrong format

***3.16 void resetMemory();***

resets the entire virtual memory with the value 0

***3.17 int fillMemory();***

fills the virtual memory from start to end address with the given value. Returns 1 if successful and returns 0 if unsuccessful due to wrong format

***3.18 void showOpcode();***

Displays the list of mnemonic and associated opcodes stored in the hash table

***3.19 int getOpcode();***

prints out the opcode of the target mnemonic. Returns 1 if successful. Prints error message and returns 0 if failed to find the mnemonic

***3.20 void insertTableElement(int opcode, char \* mnemonic, char \* format );***

Insert a new hash table element into the hash table

***int getHashKey(char \* mnemonic);***

returns the hash key for the given mnemonic

***int getCommand();***

gets user input from user and stores it to global variable, userInput. By comparing userInput with the list of commands, it finds out which command should be called. Returns 0 if user puts quit, returns 1 otherwise

**4. Global Variables**

***4.1 h\_node head***

The head of linked list of history node

***4.2 h\_node current***

The current node of history node

***4.3 Table\_Element \* HashTable [MAX\_HASH\_SIZE]***

hash table used to store opcode,mnemonic and format from opcode.txt file

***4.4 unsigned char VMemory [MAX\_MEMORY\_SIZE]***

Virtual memory of SIC/XE machine implemented in this program, with size of 1048576

***4.5 char params[3][10]***

stores parsed parameters from user input. First parameter is stored in params[0], second in params[1] and third in params[2].

***4.6 int numOfParams***

store number of parameters in the user input

***4.7 long defaultStartAddr***

starting address of memory for dump command.

***4.8 int instLength***

The length of the instruction (excluding unnecessary inputs from user like empty space and parameters)

***4.9 int historyCount***

The total number of history stored

***4.10 char simpleInsts [10][10]***

{"help", "h", "dir", "d", "history","hi","quit","q", "reset","opcodelist"}, which is the list of instructions that does not require any parameter

***4.12 char complexInsts[7][10]***

{"dump","du","edit","e","fill","f","opcode"}, which is the list of instructions that require extra parameter input

***4.13 char userInput[100]***

stores user input from shell

***4.14 char command[11]***

stores command from user input (excludes empty string)

***4.15 char targetMnemonic[7]***

stores mnemonic from user input when opcode command is executed

**5. Code Description**

<20161619.h > defines global variables and user-defined functions that are used in 20161619.c.

Please read the comments in green to understand the code.

#include <limits.h>

#include <string.h>

#include <dirent.h>

#include <stdlib.h>

#include <sys/stat.h>

#define QUIT 0

#define MAX\_MEMORY\_SIZE 1048576 // Arraysize to store 1 MB memory

#define MAX\_HASH\_SIZE 20 // max size for hash table

#define MAX\_USER\_INPUT 100 // max user input

#define ERROR 0; // used for return value of function

#define SUCCESS 1;

/////////// DATA STRUCTURE ////////////

typedef struct History\_Node \* h\_node;

typedef struct History\_Node{ // node for storing history information

char \* s;

int n;

struct History\_Node \*next;

}History\_Node;

typedef struct Table\_Element{

char mnemonic[7];

char format[5];

int opcode;

struct Table\_Element\* next;

}Table\_Element;

h\_node head; // points to the head of linked list of history node

h\_node current; // points to the current node of history node

Table\_Element \* HashTable [MAX\_HASH\_SIZE]; // hash table used to store opcode info

/// CONSTANTS used for printing out errors in printErrorMessage function

const int ERROR\_PARAMETER = -1;

const int ERROR\_ADDRESS\_OUT\_OF\_BOUND = -2;

const int ERROR\_INPUT\_FORMAT = -3;

const int ERROR\_PARAMETER\_LENGTH = -4 ;

const int ERROR\_PARAMETER\_OUT\_OF\_BOUND = -5;

const int ERROR\_PARAMETER\_NOT\_HEX = -6 ;

///// VARIABLES

unsigned char VMemory [MAX\_MEMORY\_SIZE]; // Virtual memory

char params[3][10]; // used to parse parameters for commands

int numOfParams=0; // global variable to store number of parameters

long defaultStartAddr = -1; // default starting address for memory for dump command

int instLength = 0; // length of the instruction (excluding unnessary inputs from user like empty space and parameters)

int historyCount = 0; // total number of history stored

char simpleInsts [10][10] = {"help", "h", "dir", "d", "history","hi","quit","q", "reset","opcodelist"}; // list of instructions that does not require any parameter

char complexInsts[7][10] = {"dump","du","edit","e","fill","f","opcode"}; // list of instructions that require extra parameter input

char userInput[100]; // stores user input

char command[11]; // stores command from user input (excludes empty string)

char targetMnemonic[7]; // stores mnemonic from user input

//// USER-DEFINED FUNCTIONS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* updateHistory

\* adds new history node, storing user command, into history linked list.

\* this function is only called when user puts in the properly working command

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void updateHistory(char \*command);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* showHistory

\* prints out the history linked list, showing all the previous commands

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void showHistory();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* listdir

\* prints out the files in the path

\* folders and output files can be identified with '/' and '\*' added at the end of the filename

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int listdir(const char \*path);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* printAllCommands

\* prints out all the commands working for this SIC machine

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void printAllCommands();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* cleanEmptyStr

\* removes any empty characters in the userInput before the first character

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void cleanEmptyStr();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* isSimpleInst

\* returns 1 if the user input is a single instruction ( instruction requiring no extra parameters)

\* returns 0 otherwise

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int isSimpleInst();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* isComplexInst

\* returns 1 if the user input is a multiple instruction ( instruction requiring no extra parameters)

\* returns 0 otherwise

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int isComplexInst();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* isEmpty

\* returns 1 if the character is empty, 0 otherwise

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int isEmpty(char c);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* isHexadecimal

\* returns 1 if the character is hexadecimal, 0 otherwise

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int isHexadecimal(char c);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* cleanEmptyStr

\* removes any empty string in front of the first non-empty letter

\* in the command.

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void cleanEmptyStr();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* checkParams

\* checks the number of parameters in the user input

\* and stores them into params array. the number of parameter is stored in the global variable, numOfParams

\* returns 0(ERROR\_CODE) if parameter is in the wrong format for the specific command

\* returns 1(SUCCESS\_CODE) if parameter is in the right format

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int checkParams();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* checkParamsMnemonic

\* checks the format of parameter for opcode command

\* returns 0(ERROR\_CODE) if parameter isn't in capital letter or exceeds maximum length

\* parses mnemonic from user input and stores it to global variable, targetMnemonic

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int checkParamsMnemonic();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* printMem

\* prints out virtual memory from start address to end address

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void printMem(long start, long end);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* dumpMemory

\* prints out virtual memory

\* if no parameter is given, it prints out 160 from defaultStartAddr

\* if one parameter is given(start), it prints out 160 from start address

\* if two parameters are given(start,end), it prints out from start to end address

\* returns 0 if command is given in the wrong format

\* returns 1 if the command worked successfully

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int dumpMemory();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* editMemory

\* changes the memory at the given address to the value given.

\* returns 1 if successful

\* returns 0 if unsuccessful (wrong format)

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int editMemory();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* resetMemory

\* resets the entire virtual memory with value 0

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void resetMemory();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* fillMemory

\* fills the virtual memory from start to end address with the given value

\* returns 1 if successful

\* returns 0 if unsuccessful (wrong format)

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int fillMemory();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* showOpcode

\* prints out the list of opcodes stored in the hash table

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void showOpcode();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* getOpcode

\* prints out the opcode of given mnemonic

\* returns 1 if successful

\* returns 0 if failed to find the mnemonic

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int getOpcode();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* insertTableElement

\* insert new hash table element into the hash table

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void insertTableElement(int opcode, char \* mnemonic, char \* format );

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* getHashKey

\* returns the hash key for the given mnemonic

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int getHashKey(char \* mnemonic);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* getCommand

\* get user input from user and store it into global variable, userInput

\* by parsing the userInput, it finds out which command should be called

\* returns 0 if user puts quit

\* returns 1 otherwise

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int getCommand();

<20161619.c>

#include <stdio.h>

#include "20161619.h"

void updateHistory(char \*command){

h\_node temp = (History\_Node \*)malloc(sizeof(History\_Node) \* 1); // allocate history node

temp->n = historyCount; // store current number of history stored into n

temp->next = NULL;

temp->s = (char \*)malloc(sizeof(char) \* strlen(command)); // dynamically allocate char array

command[strlen(command)-1]='\0'; // store null value at the end of the array

strcpy(temp->s, command); // copy command to s

if (head == NULL){ // if there is no previous history node stored

head = temp; // update head and current

current = head;

}

else{

current->next = temp; // otherwise add the new node to the linked list

current = temp;

}

}

void showHistory(){

h\_node temp;

temp = head;

while (temp != NULL){ // starting from head node, until it reaches the end

printf("%d %s\n", temp->n, temp->s); // print out the history info

temp = temp->next;

}

}

int listdir(const char \*path){

struct dirent \*ent;

struct stat buf;

DIR \*dp = opendir(path);

if (dp == NULL) { // print error if directory cannot be opened

printf("directory cannot be opened\n");

return ERROR;

}

while ((ent = readdir(dp))) { // read the directory

printf("%s", ent->d\_name); // print out the name of the file

lstat(ent->d\_name, &buf);

if (S\_ISDIR(buf.st\_mode)){ // if the file is a folder

printf("/");

}else if (buf.st\_mode & S\_IEXEC ){ // if the file is output file

printf("\*");

}

printf(" ");

}

printf("\n");

closedir(dp); // close the directory

return 0;

}

void printAllCommands(){ // print out all the commands

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

}

int isSimpleInst(){

int i,j,flag,found = 0;

for (i = 0; i < 10; i++) { // loop through simple instruction list to find out it matches the user input

flag = 0; // initialize flag value

instLength = strlen(simpleInsts[i]); // get the length of the instruction

if (strncmp(userInput, simpleInsts[i], instLength) == 0){ // if user input contains the following instruction

for (j = instLength; j < MAX\_USER\_INPUT; j++) { // loop through the rest of user input

if (!isEmpty(userInput[j])){ // if other non empty character exists

flag = 1; // flag is set to show that the user input is not the following instruction

break;

}

}

if(flag == 0){ // if the user input contains only the instruction

strcpy(command, simpleInsts[i]); // copy the instruction to command

found = 1; // set found to 1

flag = 0;

}

}

}

if(found){ // if the user input is simple instruction in the right format

return SUCCESS; // return 1

}else

return ERROR; // return 0 otherwise

}

int isEmpty(char c){

return (c== ' ' || c == '\t' || c == '\0' || c == '\n'); // if the character is one of the following, it is identified to be empty

}

int isHexadecimal(char c){

return ('0' <= c && c <= '9' ) || ('A' <= c && c <= 'F') || ('a' <= c && c <= 'f'); // returns 1 if the character is in hexadecimal

}

void printErrorMessage(int type){ // print out error message for specific type

switch(type){

case ERROR\_PARAMETER : printf("Wrong parameter given..\n"); break;

case ERROR\_ADDRESS\_OUT\_OF\_BOUND : printf("Address out of bound..\n"); break;

case ERROR\_INPUT\_FORMAT : printf("Wrong parameter given..\n"); break;

case ERROR\_PARAMETER\_LENGTH : printf("Parameter length cannot be bigger than 9..\n"); break;

case ERROR\_PARAMETER\_OUT\_OF\_BOUND : printf("All parameter should be in between 0~0XFF..\n");break;

default : printf("Wrong command.\n");

}

}

int isComplexInst(){

instLength = 0;

for (int i = 0; i < 7; i++){ // loop through complex instruction list

instLength = strlen(complexInsts[i]); // get length of the instruction

if (strncmp(userInput, complexInsts[i], instLength) == 0) { // if the user input includes following instruction

if (isEmpty(userInput[instLength])) // check if the next character is empty

{

strcpy(command, complexInsts[i]); // copy the instruction to command

return SUCCESS;

}

}

}

return 0;

}

int checkParams(){

int comma = 0;

char input[10] = "";

int paramLength = 0 ;

int last = 0;

int idx1,idx2,idx3,i;

for(i = instLength; i < MAX\_USER\_INPUT; i++){ // search the string after the instruction

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

comma++; // keep track of number of comma in user input

if(comma==1){

idx1=i; // store the location of the first comma in idx1 eg) 1,2

} else if(comma==2){ // if there is 2 commas

idx2=i; // store the location of second comma in idx2 eg) 1,2,3

}else { // returns 0 because there should not be more than 3 commas

return ERROR; // eg) 1,2,3,4

}

}

}

if(comma == 0 && (strncmp(userInput,"du",2)==0 || strncmp(userInput,"dump",4)==0)){ // for dump without any parameter

for( i = instLength; i < MAX\_USER\_INPUT; i++){

if(!isEmpty(userInput[i])){ // check the rest of string

break;

}

}

if(i==MAX\_USER\_INPUT){ // if the string only contains dump

numOfParams = 0; // update numOfParams and return success code

return SUCCESS;

}

}

if(comma>=0){ // initial start

last = 0;

for(i = instLength; i<MAX\_USER\_INPUT; i++){

if(userInput[i]=='\0' || userInput[i]==','){ // skip if there is second comma or it's end of string

break;

}

if(!(isEmpty(userInput[i]))){

if(isHexadecimal(userInput[i])){

input[last++] += userInput[i]; // store the parameter into input

if(last>9){ // if parameter is too long print out error message

printErrorMessage(ERROR\_PARAMETER\_LENGTH);

return ERROR;

}

}else{ // if non-hexadecimal is given as parameter, print out error

printErrorMessage(ERROR\_INPUT\_FORMAT);

return ERROR;

}

}

}

numOfParams = 1; // update numOfParams

strcpy(params[0],input); // store first parameter to params

}

if(comma>=1){ // if there is at least one comma

memset(input,0,sizeof(input)); // initialize input array and last

last = 0;

for(i = idx1+1; i<MAX\_USER\_INPUT; i++){ // search from character next to the first comma

if(userInput[i]=='\0' || userInput[i]==','){ // skip if end of string or second comma

break;

}

if(!(isEmpty(userInput[i]))){

if(isHexadecimal(userInput[i])){ // store the hexadecimal value into input

input[last++] += userInput[i];

if(last>9){ // if parameter is too long, print out error message

printErrorMessage(ERROR\_PARAMETER\_LENGTH);

return ERROR;

}

}else{ // print error for non-hexadecimal value

printErrorMessage(ERROR\_INPUT\_FORMAT);

return ERROR;

}

}

}

numOfParams = 2;

strcpy(params[1],input); // store second parameter into params

}

if(comma==2){ // if there are two commas

memset(input,0,sizeof(input)); // initialize input and last

last = 0;

for(i = idx2+1; i<MAX\_USER\_INPUT; i++){ // search from character next to second comma

if(userInput[i]=='\0'){ // break if end of string

break;

}

if(!(isEmpty(userInput[i]))){

if(isHexadecimal(userInput[i])){ // store the hexadecimal value into input

input[last++] += userInput[i];

if(last>9){ // if parameter is too long, print out error message

printErrorMessage(ERROR\_PARAMETER\_LENGTH);

return ERROR;

}

}else{ // print error for non-hexadecimal value

printErrorMessage(ERROR\_INPUT\_FORMAT);

return ERROR;

}

}

}

numOfParams = 3;

strcpy(params[2],input); // store third parameter into params

}

return 1;

}

void printMem(long start, long end){

long rowStart = start - start % 16; // calculate the first row address

long rowEnd = end - end % 16; // calulcate the last row address

int value;

long i,j;

for (i = rowStart; i <= rowEnd; i += 16) { // loop through rowStart until rowEnd, each incrementing by 16

printf("%05lX ", i);

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

if (j >= start && j <= end) { // if the address is within the start and end address

value = VMemory[j]; // print out the memory

printf("%02X", value);

}

else{

printf(" "); // print out empty space otherwise

}

printf(" ");

}

printf("; ");

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

if (VMemory[j] >= 20 && VMemory[j] <= 0x7E) // if the value is within the range

printf("%c", VMemory[j]); // print out the memory in character value

else

printf("."); // print dot otherwise

}

printf("\n");

}

}

void cleanEmptyStr(){ // converts userInput so it starts with non-empty character

char temp[MAX\_USER\_INPUT]; // create temporary array

int i,j;

for(i = 0 ; i < MAX\_USER\_INPUT; i++) // initialize temp with empty character

temp[i] = ' ';

for(i = 0 ; i < MAX\_USER\_INPUT; i++){ // loop through user input

if(!(isEmpty(userInput[i]))){ // break if non-empty character

break;

}

}

if(i==MAX\_USER\_INPUT) // if the entire string is empty, return

return;

for(j = 0; j < MAX\_USER\_INPUT; j++){

if(j+i<MAX\_USER\_INPUT)

temp[j] = userInput[j+i]; // copy the content of userInput to temp, starting index of the first non-empty character

}

for(i = 0 ; i < MAX\_USER\_INPUT; i++) // erase the previous userInput

userInput[i] = ' ';

strcpy(userInput,temp); // copy the temp content to userInput

}

int dumpMemory(){

long start, end, rowStart, rowEnd;

char \*err;

if (numOfParams == 0) { // if no parameter is given

if (defaultStartAddr >= MAX\_MEMORY\_SIZE-1) // set defaultStartAddr to -1 if it exceeds MAX\_MEMORY\_SIZE

defaultStartAddr = -1;

start = defaultStartAddr + 1; // increment start value

end = start + (16 \* 10 - 1); // set the end address to be 159 bigger than start addresss

}

else if (numOfParams == 1){ // if one parameter is given

start = strtol(params[0], &err, 16); // set start address to the given parameter

end = start + (16 \* 10 - 1);

}

else if (numOfParams == 2) { // if two parameters are given

start = strtol(params[0], &err, 16); // set start

end = strtol(params[1], &err, 16); // set end address

}

else{

printErrorMessage(ERROR\_INPUT\_FORMAT); // if more than two parameters are given, print out error message

return ERROR;

}

if(end >= MAX\_MEMORY\_SIZE){ // if end address is bigger than MAX memory size

end = MAX\_MEMORY\_SIZE-1; // set end address to the end memory address

}

if(start > MAX\_MEMORY\_SIZE || start < 0 ){ // print error message if start is bigger than max memory or less than zero

printErrorMessage(ERROR\_ADDRESS\_OUT\_OF\_BOUND);

return ERROR;

}

if(start > end ){ // print error message if start is bigger than end

printErrorMessage(ERROR\_ADDRESS\_OUT\_OF\_BOUND);

return ERROR;

}

defaultStartAddr = end; // update global variable

printMem(start, end); // print out memory

return SUCCESS;

}

int editMemory(){

char \*err;

if (numOfParams == 2) { // if two parameter values are given

long address = strtol(params[0], &err, 16); // convert each parameter to long type

long value = strtol(params[1], &err, 16);

if (address < 0 || address > MAX\_MEMORY\_SIZE-1){ // print error message if address is out of bound

printErrorMessage(ERROR\_ADDRESS\_OUT\_OF\_BOUND);

return ERROR;

}else if(value > 0xFF){ // print out error message if value is out of bound

printErrorMessage(ERROR\_PARAMETER\_OUT\_OF\_BOUND);

return ERROR;

}

VMemory[address] = value; // change memory to the value

return SUCCESS;

}else{

printErrorMessage(ERROR\_PARAMETER); // if wrong number of parameter is given

return ERROR;

}

}

void resetMemory(){

memset(VMemory, 0, sizeof(VMemory)); // reset all virtual memory to zero

}

int fillMemory(){

char \*err;

if (numOfParams == 3){ // if three parameters are given

long start = strtol(params[0], &err, 16);

long end = strtol(params[1], &err, 16); // convert three parameters into long type

long value = strtol(params[2], &err, 16);

if (start < 0 || end < 0 || start >= MAX\_MEMORY\_SIZE || end >= MAX\_MEMORY\_SIZE) { // print out error if address is not in range

printErrorMessage(ERROR\_ADDRESS\_OUT\_OF\_BOUND);

return ERROR;

}

else if (start > end){

printf("start address should be bigger than end address\n");

return 0;

}

if (value > 0xFF){

printErrorMessage(ERROR\_PARAMETER\_OUT\_OF\_BOUND);

return 0;

}

for (long i = start; i <= end; i++){ // update the memory value from start to end address

VMemory[i] = value;

}

return SUCCESS;

}

else{

printErrorMessage(ERROR\_PARAMETER); // if wrong number of parameter is given

return ERROR;

}

}

void showOpcode(){ // print out all opcodes from hash table

for(int i = 0; i < MAX\_HASH\_SIZE ; i++) { // loop through hash table

Table\_Element \*elem = HashTable[i]; // get table element from hash table[i]

printf("%d :",i);

if(elem == NULL) { // if the following table is empty, skip

printf("\n");

continue;

}

while(elem != NULL) { // loop through the element until it points to null

printf(" [%s,%02X]",elem->mnemonic,elem->opcode); // print out the mnemonic and opcode of the table element

if(elem->next != NULL) { // print arrow if next element exists

printf(" ->");

}

elem = elem->next; // continue to next pointer

}

printf("\n");

}

}

int getHashKey(char \* mnemonic) { // returns the hash key for the following mnemonic

int key = mnemonic[0] + mnemonic[strlen(mnemonic)-1];

return key%MAX\_HASH\_SIZE;

}

int getOpcode(){ // prints out opcode for the target mnemonic

int key = getHashKey(targetMnemonic); // finds hashkey

for(Table\_Element \*temp = HashTable[key]; temp != NULL; temp = temp -> next) { // loop through the following hash table

if(strcmp(temp->mnemonic,targetMnemonic) == 0){ // if target mnemonic is found

printf("opcode is %X\n", temp->opcode); // print out the opcode

return SUCCESS;

}

}

printf("opcode is not found for %s\n", targetMnemonic); // if not found

return ERROR;

}

int checkParamsMnemonic(){

char tempM[10];

memset(tempM,0,sizeof(tempM)); // intialize

int i,idx = 0;

int lastIndex = 0;

int flag = 0;

for(i = 6; i < MAX\_USER\_INPUT; i++){ // loop through the rest of user input after 'opcode'

if(idx==10){

printf("Opcode should not be longer than 9 letter\n");

return ERROR;

}

if(userInput[i]>='A'&&userInput[i]<='Z'){ // if the character is a capital letter

tempM[idx++] = userInput[i]; // copy the letter to tempM

flag = 1;

}else if(!(isEmpty(userInput[i]))){ // if the character is not in capital letter

printf("opcode should be in capital letter\n"); // print error message

return ERROR;

}

if(flag){

if(isEmpty(userInput[i])) // if there has been a letter and currently it is empty, break

break;

}

}

for(int j = i; j < MAX\_USER\_INPUT; j++) { // loop through to make sure the rest of the string is empty after the mnemonic

if(!(isEmpty(userInput[j]))){ // if other characters exist after mnemonic

printErrorMessage(ERROR\_INPUT\_FORMAT);

return ERROR;

}

}

strcpy(targetMnemonic, tempM); // copy tempM to targetMnemonic

return SUCCESS;

}

void insertTableElement(int opcode, char \* mnemonic, char \* format ){

Table\_Element \*newElem = (Table\_Element\*) malloc(sizeof(Table\_Element)); // allocate new table element

strcpy(newElem->mnemonic,mnemonic); // copy mnemonic and format string

strcpy(newElem->format,format);

newElem->opcode = opcode;

int key = getHashKey(mnemonic); // get hash key for the following mnemonic

if(HashTable[key]!=NULL){ // if the element is not the first element to be inserted into the table with the specific hash key

newElem->next = HashTable[key]; // point to the current head

HashTable[key] = newElem; // update the head to new node

}else { // if it is the first element

HashTable[key] = newElem; // update head node

newElem->next = NULL; // point to null

}

}

int getCommand()

{

int i;

int successful = 0;

for (i = 0; i < MAX\_USER\_INPUT; i++)

userInput[i] = '\0';

printf("sicsim> ");

fgets(userInput, MAX\_USER\_INPUT, stdin);

cleanEmptyStr();

if (isSimpleInst()) {

historyCount++;

updateHistory(userInput);

if (strcmp(command, "quit") == 0 || strcmp(command, "q") == 0){

return QUIT;

}

else if (strcmp(command, "help") == 0 || strcmp(command, "h") == 0){

printAllCommands();

}

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

listdir(".");

}

else if (strcmp(command, "history") == 0 || strcmp(command, "hi") == 0){

showHistory();

}else if(strcmp(command,"reset")==0){

resetMemory();

}else if(strcmp(command,"opcodelist")==0){

showOpcode();

}

}

else if (isComplexInst()){

instLength = strlen(command);

if ((strcmp(command, "dump") == 0 || strcmp(command, "du") == 0) && checkParams()){

successful = dumpMemory();

}else if((strcmp(command, "edit") == 0 || strcmp(command, "e") == 0) && checkParams()){

successful = editMemory();

}else if((strcmp(command, "fill") == 0 || strcmp(command, "f") == 0) && checkParams()){

successful = fillMemory();

}else if(strcmp(command, "opcode") == 0 && checkParamsMnemonic()){

successful = getOpcode();

}

}

if(successful){

historyCount++;

updateHistory(userInput);

}

return 1;

}

int main()

{

FILE \* ip;

int opcode;

char mnemonic[7];

char format[5]; // temporary storage for storing opcode info

char \*err;

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

if(ip==NULL){ // print error message if opcode text file is not found

printf("opcode file does not exist\n");

return 0;

}

while(1){

fscanf(ip,"%X%s%s", &opcode, mnemonic, format); // read each line from opcode text file

if(feof(ip)) // break if reached end of file

break;

insertTableElement(opcode,mnemonic,format); // insert opcode info into hash table

}

resetMemory(); // initialize virtual memory

while (getCommand() != QUIT) { // get command from user and process, until quit command is given

}

return 0;

}