과목 명: 시스템프로그래밍

담당 교수 명: 박 운 상

<<Assignment 1>>

**서강대학교 컴퓨터공학과**

**[20162004]**

**[한센토미]**

목 차

[1. 프로그램 개요 4](#_Toc509823615)

[2. 프로그램 설명 4](#_Toc509823616)

[2.1 프로그램 흐름도 4](#_Toc509823617)

[3. 모듈 정의 4](#_Toc509823618)

[3.1 모듈 이름 : int main(void) 4](#_Toc509823619)

[3.1.1 기능 4](#_Toc509823620)

[3.1.2 사용 변수 5](#_Toc509823621)

[3.2 모듈이름: void getInput(char\* command) 5](#_Toc509823622)

[3.2.1 기능 5](#_Toc509823623)

[3.2.2 사용 변수 5](#_Toc509823624)

[3.3 모듈이름: char\*\* parseInput(char\* command, int\* n) 5](#_Toc509823625)

[3.3.1 기능 5](#_Toc509823626)

[3.3.2 사용변수 5](#_Toc509823627)

[3.4 모듈이름: int removeCommas(char\* string) 6](#_Toc509823628)

[3.4.1 기능 6](#_Toc509823629)

[3.4.2 사용변수 6](#_Toc509823630)

[3.5 모듈이름: int runCommands(char\*\* args, int n) 6](#_Toc509823631)

[3.5.1 기능 6](#_Toc509823632)

[3.5.2 사용변수 6](#_Toc509823633)

[3.6 모듈이름: int funcHelp(void) 6](#_Toc509823634)

[3.6.1 기능 6](#_Toc509823635)

[3.6.2 사용변수 6](#_Toc509823636)

[3.7 모듈이름: int funcDir(void) 6](#_Toc509823637)

[3.7.1 기능 6](#_Toc509823638)

[3.7.2 사용변수 7](#_Toc509823639)

[3.8 모듈이름: int funcQuit(void) 7](#_Toc509823640)

[3.8.1 기능 7](#_Toc509823641)

[3.8.2 사용변수 7](#_Toc509823642)

[3.9 모듈이름: int funcHistory(void) 7](#_Toc509823643)

[3.9.1 기능 7](#_Toc509823644)

[3.9.2 사용변수 7](#_Toc509823645)

[3.10 모듈이름: int memDump(char\*\* args, int n) 7](#_Toc509823646)

[3.10.1 기능 7](#_Toc509823647)

[3.10.2 사용변수 8](#_Toc509823648)

[3.11 모듈이름: int memEdit(char\*\* args, int n) 8](#_Toc509823649)

[3.11.1 기능 8](#_Toc509823650)

[3.11.2 사용변수 8](#_Toc509823651)

[3.12 모듈이름: int memFill(char\*\* args, int n) 9](#_Toc509823652)

[3.12.1 기능 9](#_Toc509823653)

[3.12.2 사용변수 9](#_Toc509823654)

[3.13 모듈이름: int memReset(void) 9](#_Toc509823655)

[3.13.1 기능 9](#_Toc509823656)

[3.13.2 사용변수 9](#_Toc509823657)

[3.14 모듈이름: int opInsert(void) 9](#_Toc509823658)

[3.14.1 기능 9](#_Toc509823659)

[3.14.2 사용변수 9](#_Toc509823660)

[3.15 모듈이름: struct opcodeNode\* opSearch(char\* key, int hashcode) 10](#_Toc509823661)

[3.15.1 기능 10](#_Toc509823662)

[3.15.2 사용변수 10](#_Toc509823663)

[3.16 모듈이름: int opMnem(char\*\* args) 10](#_Toc509823664)

[3.16.1 기능 10](#_Toc509823665)

[3.16.2 사용변수 10](#_Toc509823666)

[3.17 모듈이름: int opPrintOpcodelist(void) 10](#_Toc509823667)

[3.17.1 기능 10](#_Toc509823668)

[3.17.2 사용변수 10](#_Toc509823669)

[3.18 모듈이름: int hashcode(char\* string) 10](#_Toc509823670)

[3.18.1 기능 10](#_Toc509823671)

[3.18.2 사용변수 11](#_Toc509823672)

[3.19 모듈이름: void addHistory(char\* string) 11](#_Toc509823673)

[3.19.1 기능 11](#_Toc509823674)

[3.19.2 사용변수 11](#_Toc509823675)

[3.20 모듈이름: int hextoInt(char\* string) 11](#_Toc509823676)

[3.20.1 기능 11](#_Toc509823677)

[3.20.2 사용변수 11](#_Toc509823678)

[3.21 모듈이름: int power(int base, int exp) 11](#_Toc509823679)

[3.21.1 기능 11](#_Toc509823680)

[3.21.2 사용변수 11](#_Toc509823681)

[3.22 모듈이름: void freeHistory(void) 12](#_Toc509823682)

[3.22.1 기능 12](#_Toc509823683)

[3.22.2 사용변수 12](#_Toc509823684)

[3.23 모듈이름: void freeHashtable(void) 12](#_Toc509823685)

[3.23.1 기능 12](#_Toc509823686)

[3.23.2 사용변수 12](#_Toc509823687)

[3.24 모듈이름: void freeArguments(char\*\* args) 12](#_Toc509823688)

[3.24.1 기능 12](#_Toc509823689)

[3.24.2 사용변수 12](#_Toc509823690)

[3.25 모듈이름: int readline(char\* buffer, FILE\* fp) 12](#_Toc509823691)

[3.25.1 기능 12](#_Toc509823692)

[3.25.2 사용변수 13](#_Toc509823693)

[3.26 모듈이름: struct historyNode\* getSetHistHead(void) 13](#_Toc509823694)

[3.26.1 기능 13](#_Toc509823695)

[3.26.2 사용변수 13](#_Toc509823696)

[4. 전역 변수 정의 13](#_Toc509823697)

[4.1 The Defined Constants: 13](#_Toc509823698)

[4.2 Unsigned char MEMORY[MEGA] = “”; 13](#_Toc509823699)

[4.3 struct opcodeNode\* HASHTABLE[HASHSIZE]; 13](#_Toc509823700)

[5. 코드 13](#_Toc509823701)

# 프로그램 개요

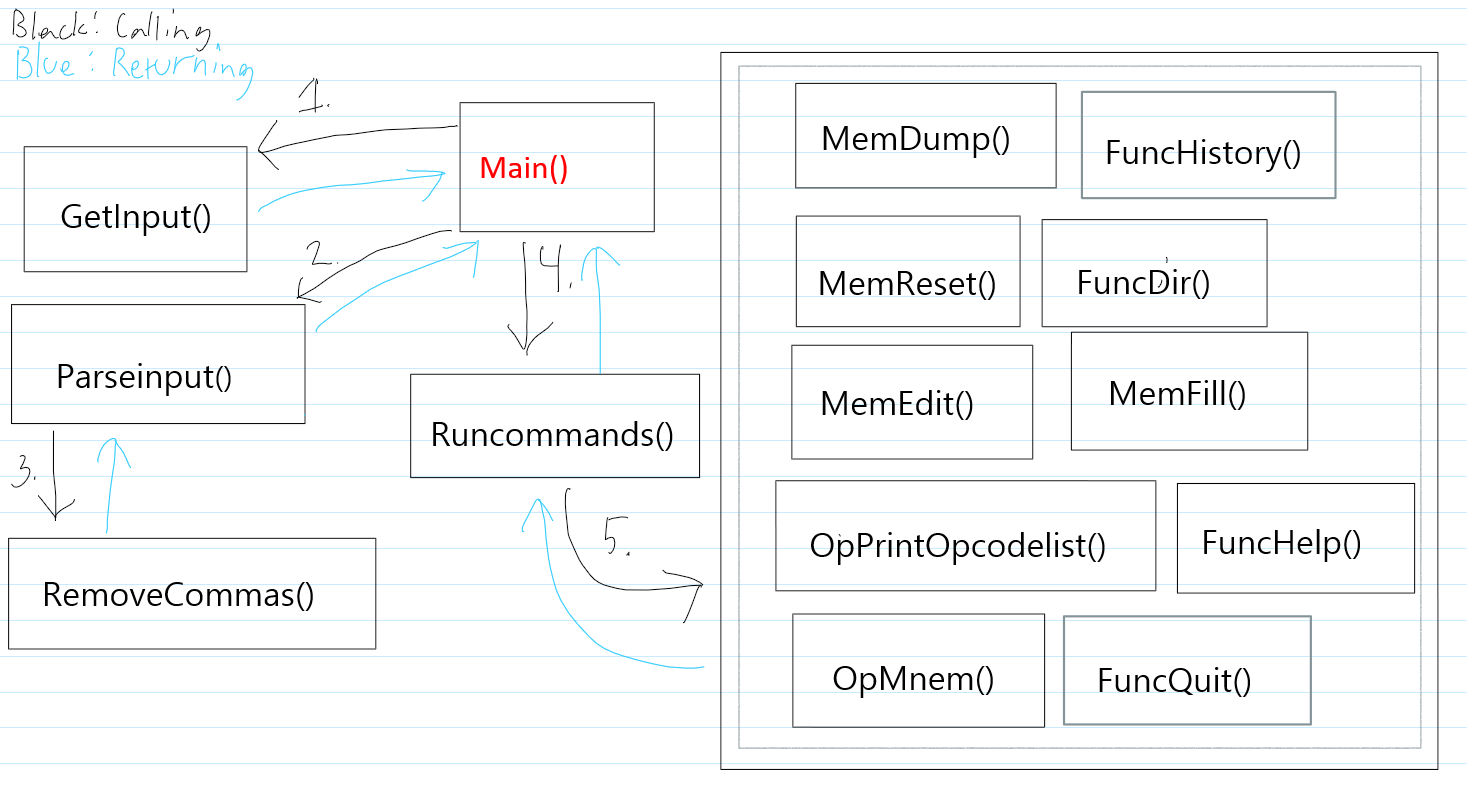
This program is a shell simulation for the SIC(Simplified Instruction Computer). It can run some basic shell commands and write/read to/from a virtual memory. It also stores mnemonic instructions for an assembly language and can provide a list of the stored mnemonics.

The program consists of a few source files and one header file:

20162004.c, shellCommands.c, generalFunctions.c, memoryCommands.c, opcodeCommands.c and 20162004.h

# 프로그램 설명

## 프로그램 흐름도

The above picture is the flowchart of the program. The black arrows indicates a function call and the blue arrows indicates a return statement. As indicated by the flowchart, the program first accepts an input, parses said input and then runs one of the programs in the box to the right depending on the input. If there was an error with the input, an error will be printed. This procedure is performed in a loop until funcQuit() is called, which terminates the program.

# 모듈 정의

## 모듈 이름 : int main(void)

### 기능

Main() is located in the file “20162004.c”. Main() first calls the function opInsert() to load up the opcode.txt into the hashtable.

In main() the main loop of the program is also located. In the loop it first prints the shell prompt “sicsim> “. The it accepts an input from user by calling the function getInput(). It the parses this input by calling the function parseInput(). If parseInput() returns at least 1 argument the function runCommand() is called. Next, depending of the status returned by runCommand(), the input will be added to the history list. At the end of the loop the memory allocated to hold the input is freed using free().

When the loop is broken main() frees all the allocated memory and the program is terminated.

### 사용 변수

char \*command: this is the array that holds the user input string.

char \*\*args: this stores the tokens created from the input in char \*command.

int status: used to keep track of the status of the program, if there have been an error etc.

## 모듈이름: void getInput(char\* command)

### 기능

This function takes a char pointer as a parameter in which it will store the users input. It uses a for loop which stores character by character, by calling getchar(), into the buffer and terminates when reaching either ‘\n’ or the MAXBUF(see global variables) is reached.

If MAXBUF was reached before ‘\n’, then the function flushes stdin before returning to the calling function.

### 사용 변수

char c: used to temporarily store the input from getchar().  
int i: this is the index offset, to keep track of the length of input.

char\* command: received as a parameter, used as a buffer to store input.

## 모듈이름: char\*\* parseInput(char\* command, int\* n)

### 기능

In this function the input is tokenized using the library function strtok().

It receives a character string and an integer value as parameters. The string holds the input from the user and the integer is used to save the amount of arguments in the input or alternatively an error message.

First the given string is copied to tmpString and strtok() is initialized once using tmpString. Then the function removeCommas() is called on tmpString to make it easier to fully tokenize the string, it also returns the amount of commas removed. If removeCommas() returns an error the function stores the error in \*n and returns NULL. Else it continues to tokenize the rest of tmpString. If there were too many arguments in the input or if there were too few commas in relation to amount of arguments, the function gives an error.

If input is correct it returns the given arguments tokenized and in \*n it stores the amount of arguments given.

### 사용변수

char\*\* arguments: used to store the pointers to the tokens.

char\* token: used to temporary store the memory address of a token.

char\* tmpString: used to store the given string, so we can edit it without losing its original.

char delim[]: initializes the delimiters used when calling strtok().

int len: stores the length of a string.

int i: used to store the index offset.

int commas: stores the amount of commas removed by removeCommas().

## 모듈이름: int removeCommas(char\* string)

### 기능

Receives a character string as a parameter. The function uses a for loop to replace all the commas in the string with blank spaces(‘ ‘) and counts how many commas it finds and also how many consecutive commas it finds.

If there are consecutive commas it aborts and returns an error. Else it returns the amount of commas it have found and erased.

### 사용변수

int i: used as index offset in the for loop.

int commaCount: used to keep count of how many commas were removed.

int conComma: used to store the amount of consecutive commas.

## 모듈이름: int runCommands(char\*\* args, int n)

### 기능

Receives the tokenized input arguments and amount of arguments as parameters.

The function uses strcmp() to compare the first argument in \*\*args with a list of valid commands. If it finds a match it calls the function related to the input argument. If there was no match the function calls printError(status) to print out the related error message.

The function returns the status, containing information whether there was an error with the input or if it was successful.

The functions it can launch are the given shell commands: h[elp], d[ir], q[uit], hi[story], du[mp] [start, end], e[dit] address, value, f[ill] start, end, value, reset, opcode mnemonic, opcodelist

### 사용변수

int status: stores the status of the program, if an error have occurred or if it is running successfully.

## 모듈이름: int funcHelp(void)

### 기능

Prints a list of available commands for the SIC shell.

### 사용변수

None

## 모듈이름: int funcDir(void)

### 기능

Prints a list of all the files and folders in the current working directory.

The function makes use of the dirent.h and sys/stat.h libraries to get information about the files in the folder. It runs a while loop to go through all of the files/folders in the folder. It uses the library function readdir() to read the files in the directory and checks if it is a folder. If it is a folder it prints the folder name followed by a slash(‘/’). If it is not a folder it makes use of the library function stat() to check if the file is an executable. If it is an executable it prints the file name followed by an asterisk(‘\*’) else it prints the filename followed by a space(‘ ‘).

### 사용변수

DIR \*dirp: this a directory pointer used to read information about the files in the current folder.

struct dirent \*dir: used to store the information returned from the library function readdir().

struct stat sb: used to store the information returned from the library function stat().

The function returns an error if there was an issue reading the directory else it returns success.

## 모듈이름: int funcQuit(void)

### 기능

Returns 0 to indicate that the loop in main() should terminate the program.

### 사용변수

None

## 모듈이름: int funcHistory(void)

### 기능

This function prints all the entered commands(valid commands) by the user, which are stored in a linked list format.

First the function makes a call to the function getSetHistHead() to get the head node of the linked list. If the list is empty the function returns 1, indicating that this history call should be added to the list in the main() loop.

If the list is not empty, it adds this history call to the list and then proceeds to print out the full history of valid commands input by the user during this session. It then returns the value 2 to indicate that this history call should not be saved by the main() loop.

### 사용변수

struct historyNode\* ptr: used to traverse the linked list that stores the history list.

## 모듈이름: int memDump(char\*\* args, int n)

### 기능

This function takes the tokenized arguments and the amount of arguments as parameters. The function’s purpose is to print out the contents from the virtual memory of the shell. On each row it indicates the start address of the memory, the content of 16 bytes of memory in both hexadecimal and ASCII format.

If there is only one argument given, the function prints out 160 bytes of memory from the current memory position. If two arguments is given the function calls hexToInt() to translate the argument from a string of hexadecimals to integer in order to know the start position, and from that start position it prints out 160 bytes of memory. If 3 arguments are given it call hexToInt() twice, to get the start and end position from input. It then continues to dump the memory in the given range.

The function will return errors if the hexadecimal values are too large, or in an invalid format. Or if the start position > end position.

I designed the function to store the hexadecimal values for addresses as integers to make it easy to use them as indices for the virtual memory as well as print them as hexadecimals.

Before entering the loop to print the information, it is calculated how many entries are to be skipped on the first row, since each row is designed to print 16 bytes from hexadecimal ranges such as 0~10 and 10~20. This calculation is done by taking the start position % 16, and the remainder is how many entries are to be skipped.

I made a nested for loop design to print all the information. In the outer loop it prints the row address in hexadecimal, ex. ABCF0, and the loop terminates either when the end position is reached or if the memory range have been breached (1MB). The inner loop terminates whenever 16 bytes have be printed or if the memory range have been breached. If there are entries to be skipped at the beginning of the first row it first makes sure to print those spaces. Then it prints the contents of the current memory address as hexadecimal values, byte by byte. While doing so it stores the values of the memory in a string as ASCII characters and when the 16 iterations for the row is done it prints the ASCII string at the end. As specified in the manual, only values between 20~7E (hexadecimal) are printed as ASCII, the other values are printed as “.”.

At the end of the function it stores the end position + 1 in a static integer to know where to start next time dump is called. If the end of range have been reached it will be reset to 0.

### 사용변수

static int memPos: used to “remember” the function’s end position + 1 from last time.

int startPos, endPos: used to store the start and end positions for the range to be dumped.

int i: used as a counter for the outer for loop.

int count: used as a counter for the inner for loop.

char buffer[]: used to store the ASCII string to be printed at the end of each row.

int begIn: short for beginning indentations, and stores how many entries to skip on first row.

## 모듈이름: int memEdit(char\*\* args, int n)

### 기능

This function accepts tokenized strings and their amount as parameters. The purpose of this function is to edit one byte of memory at a given location.

In args[1] is the memory position and in args[2] is the hexadecimal value to be stored in mentioned memory position.

The function makes use of hexToInt() function to understand the hexadecimal values given in the parameters. If the arguments are not valid or there are too few/many the function returns an error. Else it assigns the given value to the declared memory position.

### 사용변수

int address: here it stores the memory address, translated from string to integer form.

int value: here it stores the value to be assigned to the specified address.

## 모듈이름: int memFill(char\*\* args, int n)

### 기능

Function receives the tokenized input and the amount of tokens. It gets the start and end memory positions for the range to fill with the specified value in the last argument.

The function returns an error if not enough arguments are given or if the given arguments are invalid(i.e out of the designated range).

It uses hexToInt() to convert from a string of hexadecimals to integer form for the start, end positions as well as the values to fill the range with.

It uses the library function memset() to fill the specified memory range with the specified value.

### 사용변수

int startPos, endPos: used to store the start and end positions for the range to be filled.

int value: here it stores the value to be assigned to the specified address range.

## 모듈이름: int memReset(void)

### 기능

This function makes a call to the library function memset() to set all the values to ‘\0’ in the range 0x00000~0xFFFFF.

### 사용변수

None

## 모듈이름: int opInsert(void)

### 기능

The purpose of this function is to read the opcode.txt file and store its contents in a hashtable for ease of access.

It first uses fopen() to open the file in read-only mode.

Then in a while loop it reads the file one line at a time by making use of the readline() function, until it reaches end of file. For every line read it allocates memory for a new node to store the information read from the file. It uses the library function strtok() to tokenize the data before storing it in the new node.

Before adding the new node to the hashtable it finds its hashcode by calling the function hashcode(). If a collision occurs when adding the new node to the hashtable, the new node is added to the end of a linked list for specified hashtable index.

### 사용변수

FILE \*fp: the file pointer for opcode.txt.

char buffer[]: the buffer to store the line read from file.

struct opcodeNode\* newNode: pointer used to allocate memory for a new node.

struct opcodeNode\* ptr: pointer used to traverse the linked list when there are collisions in the hash table.

char\* token: used to temporarily point to a token given by strtok().

char delim[]: the delimeters passed to strtok().

int index: used for indices related to hashtable.

int status: used to store information on whether EOF have been reached while reading the file.

## 모듈이름: struct opcodeNode\* opSearch(char\* key, int hashcode)

### 기능

This function searches for an entry in the hashtable at the index offset “hashcode” with char key (given as parameters). If found the function returns a pointer to the entry, else it returns NULL.

### 사용변수

Struct opcodeNode\* ptr: Used to traverse the linked list in the hashtable.

## 모듈이름: int opMnem(char\*\* args)

### 기능

This function calls opSearch() function to locate the mnemonic command given as a parameter, it also used the function hashcode() to pass the hashcode of the mnemonic as a parameter to opSearch(). If found it prints out the mnemonic’s related opcode as a hexadecimal. Else the function returns an error.

### 사용변수

struct opcodeNode\* node: used to point to the node found by opSearch().

## 모듈이름: int opPrintOpcodelist(void)

### 기능

Makes use of an outer for loop and inner while loop to print out all of the entries in the hashtable storing the opcode list. The function prints out the mnemonic and the related opcode. The for loop iterates the index offset for the hashtable, and the while loop traverses the linked list at that hashtable index.

Example of output:

5 : [COMPR,A0] -> [TIX,2C]

6 : [STCH,54]

### 사용변수

int i: used as a counter and index offset in the for loop for the hashtable.

struct opcodeNode\* ptr: this pointer is used to traverse the linked lists.

## 모듈이름: int hashcode(char\* string)

### 기능

Takes a string as a parameter and calculates an integer value from the sum of each character in the given string. It then takes the calculated sum % HASHSIZE to find the remainder, which is then returned as the assigned hashcode to the received input.

### 사용변수

int result: stores the sum of the characters in the given string

int i: used as index offset to traverse the string.

## 모듈이름: void addHistory(char\* string)

### 기능

The purpose of this function is to store the parameter string to the linked list where all the previously used commands are stored as history.

It gets the head node of the linked list from the getSetHistHead() function. It then allocates memory for a new node, stores the parameter string in the new node using library function strcpy() and then adds the new node as the last entry to the linked list.

### 사용변수

Struct historyNode\* head: used to point to the head node of the linked list.

struct historyNode\* ptr: used to traverse the linked list to the last node.

struct historyNode\* newNode: used to allocate memory for a new node to add to list.

## 모듈이름: int hextoInt(char\* string)

### 기능

This function receives a character string as a parameter and returns the integer value representation of the hexadecimal value stored in the string.

It calculates the integer value by going through the string from the least significant value to the most significant value. To give the hexadecimal its correct value (Dn \* 16n) it makes use of a function to raise the base (16) to the appropriate power for each iteration.

This function returns an error if the string length is too long (>5), indicating that the hexadecimal is of too large value. Or if the input is not a valid hexadecimal number (i.e. 0~F).

### 사용변수

int result: stores the sum of the decimal representation.

int i: used as index offset for the character string.

int len: used to store the length of the given string.

## 모듈이름: int power(int base, int exp)

### 기능

This function raises the given base to the given exp (exponent) and returns it as an integer.

It does so by the use of a for loop doing the following multiplication: result = result \* base;

### 사용변수

int i: used as a counter for the iteration of the for loop.

int result: the accumulated integer value from raising the base to its exponent.

## 모듈이름: void freeHistory(void)

### 기능

This function frees the memory of the linked list holding the command history. It does this by traversing the linked list using 2 pointers and a while loop. It receives the head of the linked list by calling the function getSetHistHead().

### 사용변수

struct historyNode\* head: used to point to the head of the linked list.

struct historyNode\* ptr: used to traverse the linked list.

struct historyNode\* delete: points to the node that is to be freed.

## 모듈이름: void freeHashtable(void)

### 기능

This function frees the memory of the hashtable containing the opcode list.

It does so by using an outer for loop and an inner while loop. The outer for loop iterates the hashtable’s indices and the inner while loop traverses the linked lists that are due to collisions.

### 사용변수

struct opcodeNode\* ptr: this pointer is used to traverse the linked list.

struct opcodeNode\* delete: this pointer points to the node that is to be freed.

int i: used as the index offset for the hashtable.

## 모듈이름: void freeArguments(char\*\* args)

### 기능

This function frees the memory that stores the tokenized arguments that are created by the user input. The function is given the arguments as a parameter.

The library function strtok() that tokenized the input doesn’t actually allocate new memory for the tokens, but merely splits up the original string and gives pointers to different locations in that string. Thus to free the memory of all the tokens we only need to free the memory of the string in which the tokens exist. So this function only makes two calls to free() to free all the memory:

free(args[0]) and free(args).

### 사용변수

None

## 모듈이름: int readline(char\* buffer, FILE\* fp)

### 기능

This function reads a line from the file pointer given as a parameter and stores it in the buffer also given as a parameter.

It uses the library function fgetc() to read one character at a time using a for loop that terminates when reaching end of line or EOF.

It returns 0 to indicate EOF and returns 1 to indicate that there are more lines to read.

### 사용변수

int i: used as a counter and index offset for the for loop.

char c: used to temporarily store the read input from the file using fgetc().

## 모듈이름: struct historyNode\* getSetHistHead(void)

### 기능

This function when called the first time allocates memory and initializes a head node for the history’s linked list and returns a pointer to this head node. Next time the function is called it simply returns the previously initialized head node. This is achieved by using a static pointer.

### 사용변수

static struct historyNode\* head: a static pointer to the head node of the history linked list.

# 전역 변수 정의

## The Defined Constants:

#define MAXBUF 100

#define TOKENBUF 4

#define MEGA 1048576 // 220

#define HASHSIZE 20

## Unsigned char MEMORY[MEGA] = “”;

Used as virtual memory for the SIC.

## struct opcodeNode\* HASHTABLE[HASHSIZE];

The hashtable used to store the opcode list.

# 코드

**20162004.h**

#ifndef \_\_MYHEADER\_\_

#define \_\_MYHEADER\_\_

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

#include <string.h>

#include <sys/stat.h>

// Constansts

#define MAXBUF 100

#define TOKENBUF 4

#define MEGA 1048576 // 2^20

#define HASHSIZE 20

// Data Structures

struct historyNode{

int n;

char name[MAXBUF];

struct historyNode \*next;

struct historyNode \*last; // only used by head node to quickly find last node in list

};

struct opcodeNode{

char op[10];

unsigned char hex;

char format[4];

struct opcodeNode\* next;

struct opcodeNode\* last;

};

// Virtual Memory

extern unsigned char MEMORY[MEGA];

// Hash table

extern struct opcodeNode\* HASHTABLE[HASHSIZE];

/\* Function Declerations \*/

int runCommand(char\*\*, int);

// general functions

void getInput(char\*);

char\*\* parseInput(char\*, int\*);

int removeCommas(char\*);

void addHistory(char\*);

int hexToInt(char\*);

int power(int, int);

void printError(int);

void freeHistory(void);

void freeHashtable(void);

void freeArguments(char\*\*);

int readline(char\*, FILE\*);

struct historyNode\* getSetHistHead(void);

// functions related to memory

int memDump(char\*\*, int);

int memEdit(char\*\*, int);

int memFill(char\*\*, int);

int memReset(void);

// opcode functions

int opInsert(void);

int opMnem(char\*\*);

struct opcodeNode\* opSearch(char\*, int);

int opPrintOpcodelist(void);

int hashcode(char\*);

// shell command functions

int funcHelp(void);

int funcDir(void);

int funcQuit(void);

int funcHistory(char\*);

#endif

**20162004.c**

#include "20162004.h" // my header

// global variables

unsigned char MEMORY[MEGA] = "";

struct opcodeNode\* HASHTABLE[HASHSIZE];

int main(void){

char command[MAXBUF] = ""; // input buffer

char \*\*args; // will point to input arguments

int status = 0; // keeps track of correct/incorrect arguments from input

if(!(status = opInsert())) printf("failed to read opcode.txt");

// main loop

do{

printf("sicsim> ");

getInput(command);

args = parseInput(command, &status);

if(args) status = runCommand(args, status);

// only adds to history if the command is correct

if(status == 1){

addHistory(command);

}

if(args) freeArguments(args);

}while(status);

freeHistory();

freeHashtable();

return 1;

}

int runCommand(char \*\*args, int n){

int status = -2;

// n is amount of arguments in args

if(n > 0){

if(!(strcmp(args[0], "dump")) || !(strcmp(args[0], "du"))){

status = memDump(args, n);

}

else if(!(strcmp(args[0], "e")) || !(strcmp(args[0], "edit"))){

status = memEdit(args, n);

}

else if(!(strcmp(args[0], "f")) || !(strcmp(args[0], "fill"))){

status = memFill(args, n);

}

else if(n == 2 && !(strcmp(args[0], "opcode"))){

status = opMnem(args);

}

// commands that should only take 1 argument

else if(args[1] == NULL){

if(!(strcmp(args[0], "help")) || !(strcmp(args[0], "h")))

status = funcHelp();

else if(!(strcmp(args[0], "dir")) || !(strcmp(args[0], "d")))

status = funcDir();

else if(!(strcmp(args[0], "quit")) || !(strcmp(args[0], "q")))

status = funcQuit();

else if(!(strcmp(args[0], "history")) || !(strcmp(args[0], "hi"))){

status = funcHistory(args[0]);

}

else if(!(strcmp(args[0], "opcodelist"))) status = opPrintOpcodelist();

else if(!(strcmp(args[0], "reset"))) status = memReset();

else status = -1;

}

// if no match

else status = -1;

}

if(n < 0){

printError(n);

status = n;

}

else if(status < 0) printError(status);

return status;

}

**generalFunctions.c**

#include "20162004.h"

void getInput(char\* command){

/\* this function gets user input until newline is enter \*/

char c;

int i = 0;

while((c = getchar()) != '\n' && i < MAXBUF - 1){

command[i++] = c;

}

command[i] = '\0';

// flushes stdin

if(i >= MAXBUF - 1)

while((c = getchar()) != '\n');

}

char\*\* parseInput(char\* command, int\* n){

/\* this function tokenizes the input and does some initial validation on the input \*/

char\*\* arguments = (char\*\*) malloc(sizeof(char\*) \* TOKENBUF+1);

char\* token;

char\* tmpString = (char\*) calloc(MAXBUF, sizeof(char));

char delim[] = " \t\b\n";

int len = 0;

int i = 0;

int commas = 0;

// if no input was given

len = strlen(command);

if(len < 1){

\*n = -2;

arguments = NULL;

return arguments;

}

// tokenizes the recieved string

strcpy(tmpString, command);

token = strtok(tmpString, delim);

len = strlen(token);

// removes the commas from the string

commas = removeCommas(tmpString+len+2);

if(commas < 0){ // if < 0 it means the string is in an invalid format

\*n = commas;

return arguments;

}

while(token != NULL){

arguments[i] = token;

i++;

if(i > TOKENBUF){

// too many arguments

\*n = -1;

return arguments;

}

token = strtok(NULL, delim);

}

arguments[i] = NULL;

\*n = i;

if(\*n > 2 && commas != \*n - 2){

// not enough commas for amount of arguments

\*n = -1;

}

return arguments;

}

int removeCommas(char\* string){

/\* this function replaces commas with spaces

\* and also checks for consecutive commas

\*/

int i;

int commaCount = 0, conComma = 0;

for(i = 0; string[i] != '\0'; i++){

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

commaCount++; conComma++;

string[i] = ' ';

if(conComma > 1) return -1;

}

else if(string[i] != ' ' && string[i] != '\n'){

conComma = 0;

}

}

return commaCount;

}

void addHistory(char\* string){

/\* this function adds a new node to the history list \*/

struct historyNode\* head = getSetHistHead();

struct historyNode\* ptr = head->last;

struct historyNode\* newNode;

// allocate memory for new node

newNode = malloc(sizeof(struct historyNode));

if(!newNode){

printf("Error allocating memory for history node.\n");

return;

}

head->n++;

head->last = newNode;

newNode->n = head->n;

newNode->next = NULL;

strcpy(newNode->name, string);

// not the first entry in linked list

if(ptr){

newNode->last = ptr;

ptr->next = newNode;

}

// if it is the first entry

else{

head->next = newNode;

newNode->last = NULL;

}

}

int hexToInt(char\* string){

/\* returns integer representation of a hexadecimal number

\* given as a character string

\*/

int result = 0;

int i;

int len = strlen(string);

if(len > 5) return -5;

for(i = len-1; i >= 0; i--){

// 0 <= x <= 9

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

result += (string[i] - '0') \* power(16, len - i - 1);

}

// A <= x <= F

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

result += (string[i] - 'A' + 10) \* power(16, len - i - 1);

}

// a <= x <= f

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

result += (string[i] - 'a' + 10) \* power(16, len - i - 1);

}

// not a valid hexadecimal

else{

return -4;

}

}

return result;

}

int power(int base, int exp){

/\* raises base to the exp \*/

int i = 1;

int result = base;

if(exp == 0) return 1;

if(exp == 1) return base;

for(i = 1; i < exp; i++){

result = result \* base;

}

return result;

}

void printError(int n){

switch(n){

case -1: printf("Error: Not a valid command, please try 'help'.\n"); break;

case -3: printf("Error: End address < Start address.\n"); break;

case -4: printf("Error: Not a valid hexadecimal.\n"); break;

case -5: printf("Error: Hexadecimals have to be in the range [0x0, 0xFFFFF].\n"); break;

case -6: printf("Error: Value to be stored exceeds the range [0x0,0xFF].\n"); break;

}

}

void freeHistory(void){

/\* this function free's the memory allocated for history list \*/

struct historyNode\* head = getSetHistHead();

struct historyNode\* ptr = head->next;

struct historyNode\* delete;

while(ptr){

delete = ptr;

ptr = ptr->next;

free(delete);

}

free(head);

}

void freeHashtable(void){

/\* this function free's the memory for the opcodelist \*/

struct opcodeNode\* ptr;

struct opcodeNode\* delete;

int i;

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

if(HASHTABLE[i]){

ptr = HASHTABLE[i];

while(ptr){

delete = ptr;

ptr = ptr->next;

free(delete);

}

}

}

}

void freeArguments(char\*\* args){

/\* frees the tokenized arguments

\* since strtok() only returns pointers pointing to the given string + offset

\* free(args[0]) implicitly free's all the tokens

\*/

if(args[0]) free(args[0]);

if(args) free(args);

}

int readline(char\* buffer, FILE\* fp){

int i;

char c;

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

c = fgetc(fp);

if(!feof(fp)){

// reached end of line

if(c == '\n'){

buffer[i] = '\0';

return 1;

}

else{

buffer[i] = c;

}

}

// reached end of file

else return 0;

}

return 0;

}

struct historyNode\* getSetHistHead(void){

/\* This function returns the head node for the history list

\* and if the head node have not been initialized, it initializes it

\*/

static struct historyNode\* head;

if(!head){

head = malloc(sizeof(struct historyNode));

if(!head){

printf("Error allocating memory for history list's head node.\n");

return NULL;

}

head->n = 0;

head->next = NULL;

head->last = NULL;

}

return head;

}

**shellCommands.c**

#include "20162004.h"

/\* This file holds all the basic shell functions:

\* Help

\* Dir

\* History

\* Quit

\*/

int funcHelp(void){

// prints the commands

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");

return 1;

}

int funcDir(void){

DIR \*dirp;

struct dirent \*dir;

struct stat sb;

dirp = opendir(".");

if(dirp){

// reading current directory

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

// does not print current(.) and the previous(..) directory

if(strcmp(dir->d\_name, "..") && strcmp(dir->d\_name, ".")){

// if directory

if(dir->d\_type == DT\_DIR){

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

}

// if not a directory

// print a '\*' for executables and a space for nonexecutables.

// stat() used the sb structure to return information about the

// file and then compares it with the library constant to see if

// it is an executable or not. It is compared using the bitwise

// operator AND to see if there is a bit matching, meaning the

// file have executable rights

else{

printf("%s%c ", dir->d\_name, stat(dir->d\_name, &sb) == 0 && sb.st\_mode & S\_IXUSR ? '\*' : ' ');

}

}

}

closedir(dirp);

}

else{

printf("error occurred while opening folder\n");

//exit(EXIT\_FAILURE);

return -2;

}

printf("\n");

return 1;

}

int funcQuit(void){

return 0;

}

int funcHistory(char \*command){

struct historyNode\* ptr = getSetHistHead();

// only print history when there are at least 1 command saved

// if there is no command saved, then this history command

// should not be printed

if(ptr->n < 1) return 1;

addHistory(command);

ptr = ptr->next;

while(ptr){

printf("%d\t%s\n", ptr->n, ptr->name);

ptr = ptr->next;

}

return 2;

}

**memoryCommands.c**

#include "20162004.h"

/\* This file holds all the function for the different memory commands:

\* memDump

\* memEdit

\* memFill

\* memReset

\*/

int memDump(char\*\* args, int n){

static int memPos = 0;

int startPos, endPos;

int i, count;

char buffer[17] = ""; // holds the string to be printed at the end of line

int begIn = 0; // this variable keeps track on how many indentations we need

// one argument given, i.e du or dump

if(n == 1){

startPos = memPos;

begIn = (startPos % 16); // this calculates how many indentations are needed at the beginning of the first line

endPos = startPos + 16 \* 10;

}

// if start address is given

else if(n == 2){

startPos = hexToInt(args[1]);

if(startPos < 0) return startPos; // if startPos < 0 it means there was an error

begIn = (startPos % 16);

endPos = startPos + 16 \* 10;

}

// start and end address given

else if(n == 3){

startPos = hexToInt(args[1]);

if(startPos < 0) return startPos; // if startPos < 0 it means there was an error

begIn = (startPos % 16);

endPos = hexToInt(args[2]);

if(endPos < 0) return endPos; // same as with startPos

if(endPos < startPos) return -3;

endPos++;

}

// too many arguments

else return -1;

// loop for printing the memory dump

for(i = startPos; i < endPos && i < MEGA;){

printf("%05X ", i - (i % 16));

for(count = 0; count < 16 && i < MEGA; count++){

if(begIn){ // begIn = beginning indentation

// prints 3 spaces and adds '.' to buffer

printf(" ");

buffer[count] = '.';

begIn--;

}

else{

printf("%02X ", MEMORY[i]);

// if 20 <= x <= 7E it is added to buffer

if(MEMORY[i] >= (char)0x20 && MEMORY[i] <= (char)0x7E){

buffer[count] = MEMORY[i];

}

else{

buffer[count] = '.';

}

i++;

if(i == endPos){

while(++count < 16){

// prints 3 spaces and adds '.' to buffer

printf(" ");

buffer[count] = '.';

}

}

}

}

// adds terminating null to buffer and then prints buffer as a string

buffer[count] = '\0';

printf("; %s\n", buffer);

}

memPos = i;

if(!(memPos < MEGA)) memPos = 0; // if end of memory is reached, reset position

return 1;

}

int memEdit(char\*\* args, int n){

int address;

int value;

// invalid amount of arguments

if(n != 3) return -1;

// convert the given values and check their validity

if((address = hexToInt(args[1])) < 0){

return address;

}

if((value = hexToInt(args[2])) < 0){

return value;

}

if(value > 0xFF) return -6;

// store in memory

MEMORY[address] = (char)value;

return 1;

}

int memFill(char\*\* args, int n){

int startPos, endPos;

int value;

// invalid amount of arguments

if(n != 4) return -1;

// convert the given values and check their validity

if((startPos = hexToInt(args[1])) < 0) return startPos;

if((endPos = hexToInt(args[2])) < 0) return endPos;

if((value = hexToInt(args[3])) < 0) return value;

if(value > 0xFF) return -6;

if(endPos < startPos) return -3;

// use memset to fill the memory in the selected range

memset(MEMORY+startPos, (char)value, endPos-startPos+1);

return 1;

}

int memReset(void){

// memset resets all of memory to 0

memset(MEMORY, '\0', MEGA);

return 1;

}

**opcodeCommands.c**

#include "20162004.h"

/\* This file hold all the functions regarding the opcodelist:

\* opInsert

\* opMnem

\* opSearch

\* opPrintOpcodelist

\* hashcode

\*/

int opInsert(void){

// reads file and inserts information about opcodes to hashtable

FILE \*fp;

char buffer[MAXBUF] = "";

struct opcodeNode\* newNode;

char \*token;

char delim[] = " \n\t\b";

int index;

struct opcodeNode \*ptr;

int status = 1;

if(!(fp = fopen("opcode.txt", "r"))){

return 0; // if file was not found

}

while(!feof(fp) && status){

status = readline(buffer, fp);

if(status == 0) break;

if(status < 0) return 0;

if(!(newNode = malloc(sizeof(struct opcodeNode)))){

printf("Error allocating memory for opcodelist node.\n");

return -2;

}

// initialize a new node for the hashtable

// using data read from file

token = strtok(buffer, delim);

newNode->hex = hexToInt(token);

strcpy(newNode->op, strtok(NULL, delim));

strcpy(newNode->format, strtok(NULL, delim));

newNode->next = NULL; newNode->last = NULL;

index = hashcode(newNode->op);

// if there was a collision we add the new node to the end of the list

if(HASHTABLE[index]){

ptr = HASHTABLE[index];

if(ptr->last) ptr = ptr->last;

ptr->next = newNode;

newNode->last = ptr;

HASHTABLE[index]->last = newNode;

}

else HASHTABLE[index] = newNode;

}

return 1;

}

struct opcodeNode\* opSearch(char \*key, int hashcode){

struct opcodeNode\* ptr = HASHTABLE[hashcode];

// searches for the key in HASHTABLE[hashcode]

// if it finds the key it returns a pointer to its location

while(ptr){

if(!(strcmp(ptr->op, key))){

return ptr;

}

if(!(ptr = ptr->next)) break;

}

return NULL;

}

int opMnem(char\*\* args){

// searches for & prints mnemonics from hashtable

struct opcodeNode\* node;

node = opSearch(args[1], hashcode(args[1]));

if(!node){

printf("Error: mnemonic not found.\n");

return -2;

}

printf("opcode is %02X\n", node->hex);

return 1;

}

int opPrintOpcodelist(void){

// prints all nodes in the hashtable

int i;

struct opcodeNode\* ptr;

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

ptr = HASHTABLE[i];

if(ptr){

printf("%d : ", i);

while(ptr){

printf("[%s,%02X]", ptr->op, ptr->hex);

ptr = ptr->next;

if(ptr) printf(" -> ");

else printf("\n");

}

}

else printf("%d :\n", i);

}

return 1;

}

int hashcode(char\* string){

// this function takes a string,

// converts it to an integer and determines its hashcode

int result = 0;

int i = 0;

while(\*(string + i)){

result += (unsigned int)\*(string + i++);

}

return result % HASHSIZE;

}