과목명**:** 시스템프로그래밍

1분반 박운상 교수님

**<<Project #2>>**

서강대학교 **[**학부명:컴퓨터공학과**]**

**[**학번:20161619**]**

**[**이름:이세진**]**

Contents

1. Program Summary
2. Program Description

2-1 Flowchart for shell

2-2 Algorithm for assemble command

1. Modules
2. Global Variables
3. Code Description

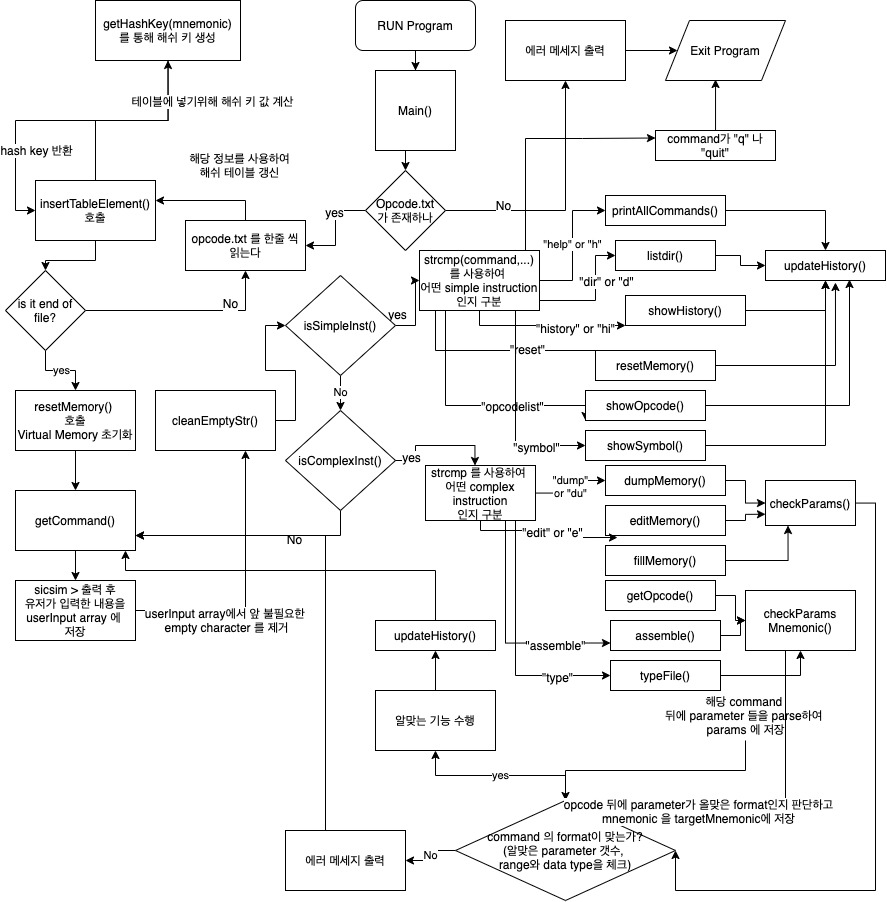
**1. Program Summary**

Using the shell created in project 1, project 2 implements assemble function to the program. With SIC/XE assembly program source as an input, the program generates object file, symbol table and immediate file.

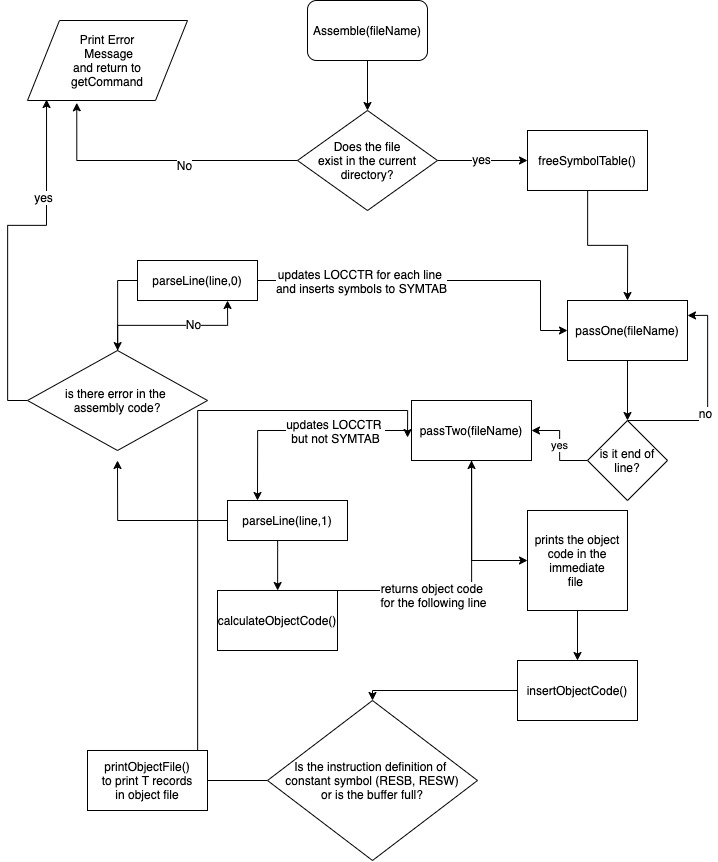
**2. Program Description**

***2.1. Program Flowchart for Shell command***

Flowchart from project 1 is slightly updated due to new commands, symbol, type and assemble.



***2.2. Flowchart for Assemble command***

******

***2.3 Algorithm for Calculating Object code***

Following algorithm was implemented in calculateObjectCode() function, where it calculates object code for each line.

|  |
| --- |
| ***Begin***  ***Initialize FORMAT to format of the OPCODE***  ***If FORMAT = 1 then***  ***Begin***  ***Set OBJECTCODE to OPCODE***  ***Shift OBJECTCODE to 4 bits to the left***  ***End***  ***Else if FORMAT = 2 then***  ***Begin***  ***Find Registers from operands***  ***If not found then***  ***Set error flag***  ***Else For each register given***  ***OBJECTCODE += registerNumber***  ***shift OBJECTCODE to 8 bits to the left***  ***END***  ***ELSE if FORMAT = “3 / 4” then***  ***Begin***  ***If OPCODE starts with “+” then***  ***Format = 4***  ***Else***  ***Format = 3***  ***If Format = 3 then***  ***Find operands***  ***Set Disp = LabelLoc-LOCCTR***  ***If not found then***  ***Set error flag***  ***Else if operand starts with “#” then***  ***Immediate addressing***  ***Set i = 1, n = 0***  ***Else if operands starts with ‘@’ then***  ***Indirect addressing***  ***Set i = 0, n = 1***  ***Else***  ***Simple addressing***  ***Set i = 1, n = 1***  ***If -2048 <= label - LOCCTR <= 2048 then***  ***Pc relative***  ***Set p = 1, b = 0***  ***Else if disp <= 4096 then***  ***Base relative***  ***Set p = 0 , b = 1***  ***Else***  ***Simple addressing***  ***Set p = 0, b = 0***  ***Else if Format = 4 then***  ***Begin***  ***Base relative addressing***  ***Set n = 0, i = 0, b = 1, p = 0 , e=1***  ***If Second Operand is ‘X’ then***  ***Set x = 1***  ***End***  ***OBJECTCODE = OPCODE;***  ***OBJECTCODE = OBJECTCODE + 32\*n + 16\*i + x\*8 + b \* 4 + p \* 2 + e;***  ***If Format = 3 then***  ***OBJECTCODE += DIsp***  ***Shift OBJECTCODE to 12 bits to the left***  ***Else if Format = 4 then***  ***OBJECTCODE += ADDRESS***  ***Shift OBJECTCODE to 20 bits to the left***  ***End***    ***End*** |

**3. User-defined Modules**

Note that functions that were defined in project 1 are excluded from this list.

*3.1.* int getHashKeySymbol(char \* symbol);

returns the hash key for the given symbol

*3.2.* long getAddress(char \* symbol);

returns the address of the given symbol

*3.2.* Table\_Element \* getElement(char \* mnemonic);

returns table element for the given mnemonic.

returns NULL if the given mnemonic is not found in the hash table.

3.3 int insertSymbolElement(char \* identifier, long address, char \* type, char \* value);

insert new symbol to the symbol table. returns 1 if successfully inserted and

returns 0 if there is already a duplicate in the table

3.4 void freeSymbolTable();

Delete previous symbol table. This is called before assemble function is called in order to erase the previous symbol table

3.5 long getByteSize(char \* str);

returns the byte size of the given string

3.6 int checkFilename();

separates the filename and extension and store them into filename and extension array

returns 0 if the file is not found in the directory and return 1 otherwise.

3.7 void showSymbol();

prints out the symbol table

3.8 int parseLine( char \* line, int option);

parse the given line to calculate the location of the mnemonic, addressing mode and seperate the operands.

if option is 0 : for the given line of assembly code, inserts symbols into SYMTAB if it exists

if option is 1 : does not insert symbol into SYMTAB

returns 1 if successful and returns 0 if there is an error

3.9 long getRegisterNumber(char \* r);

returns the register number. Return -1 if wrong register is given

3.10 void insertObjectCode(long address, char \* objcode);

insert new object code node into the object code linked list

3.12 void insertRelocationNode(long address);

insert new relocation node into the relocation linked list

3.13 long calculateObjectCode(int numLine,char \* line);

returns the object code of the given line

3.14 int typeFile(char \* fileName);

prints out the contents of the file

returns 0 if the file is not found in the directory

return 1 otherwise.

3.15 int assemble(char \* fileName);

create immediate file and object file from the assembly file

returns 0 if there is an error in the assembly code

returns 1 if successful

3.16 int passOne(char \* asmFileName);

Called when reading the assembly file for the first time

calculates LOCCTR for each line and insert symbols into the symbol table

returns 0 if there is an error in the assembly code

returns 1 if successful

3.17 int passTwo(char \* asmFileName);

method called when reading the assembly file for the second time

With the symbol table created by passOne(), it now calculates the object code for each line and generates immediate file and object file

returns 0 if there is an error in the assembly code

returns 1 successful

3.18 void printObjectFile();

prints the contents of the object code linked list into the object file and initialize the linked list

**4. Global Variables**

Please note that previously defined variables are excluded from this list. Only assembly related variables will be described in this section. Other variables are described in section 5 through comments in the code.

4.1 char filename[100]

stores the file name without the extension

4.2 char fullFileName[100]

stores the full filename

4.3 char extension[10]

stores extension of the filename

4.4 char operand[10][20]

used to store the operands in asm file

4.5 long LOCCTR = 0

stores the current LOCCTR

4.6 long previousLOCCTR = 0

stores previous LOCCTR

4.7 int line\_size = 50

size of line for reading asm file

4.8 char title[100]

stores the title of the assembly program

4.9 char base[100]

stores BASE indicated in the assembly code

4.10 int needToPrint

Indication of whether the object code needs to be printed

4.11 int endFound

flag to show that file reached END symbol

4.12 int numWord

the number of separate string in current line of assembly code

4.13 int locationOfMnemonic

index of Mnemonic in assembly code

4.14 char \* trueMnemonic

stores the mnemonic after removing +

4.15 int format

stores the format of the mnemonic

4.16 long baseLoc

address of the BASE register

4.17 int isConstant

flag to indicate the operand is a constant

4.18 int isVariable

flag to indicate the operand is a variable

4.19 int startFound

fag to indicate that the file reached START symbol

4. 20 long endLoc

location of END in the program

4. 21 long firstExecLoc

location of the first executable instruction

4.22 int isX

flag to indicate a hexadecimal constant

4.23 char TEMP\_BUFFER [100]

used to temporary store object code

**5. Code Description**

5-2) <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 <stdio.h>

#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 SYMBOL\_HASH\_SIZE 27 // max size for symbol 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{ // node for storing mnemonic information

char mnemonic[7];

char format[5];

long opcode;

struct Table\_Element\* next;

}Table\_Element;

typedef struct Symbol\_Element{ // node used for symbol table

char identifier[7];

char type[20];

long address;

char value[20];

struct Symbol\_Element\* next;

}Symbol\_Element;

typedef struct Obj\_Element{ // node used to store object code

long address;

char \* objcode;

struct Obj\_Element \* next;

}Obj\_Element;

typedef struct Relocation\_Element{ // node used to store address of relocation

long address;

long length;

struct Relocation\_Element\* next;

}Relocation\_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

Symbol\_Element \* SymbolTable [SYMBOL\_HASH\_SIZE]; // hash table used for symbol table

Obj\_Element \* T\_head;

Obj\_Element \* T\_last;

Relocation\_Element \* R\_head;

Relocation\_Element \* R\_last;

/// 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 [11][20] = {"help", "h", "dir", "d", "history","hi","quit","q", "reset","symbol","opcodelist"}; // list of instructions that does not require any parameter

char complexInsts[9][10] = {"dump","du","edit","e","fill","f","opcode","type","assemble"}; // 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

char filename[100]; // stores the file name without the extension

char fullFileName[100]; // stores the full filename

char extension[10]; // stores extension of the filename

char operand[10][20]; // used to store the operands in asm file

long LOCCTR = 0; // stores the current LOCCTR

long previousLOCCTR = 0; // stores previous LOCCTR

int line\_size = 50; // size of line for reading asm file

char title[100]; // stores the title of the assembly program

char base[100]; // stores BASE

int needToPrint; // decides whether the object code needs to be printed

int endFound; // flag to show that file reached END symbol

int numWord; // the number of seperate string in current line of assembly code

int locationOfMnemonic; // index of Mnemonic in assembly code

char \* trueMnemonic; // stores the mnemonic after removing +

int format; // stores the addressing mode of the mnemonic

long baseLoc; // address of the BASE register

int isConstant; // flag to indicate the operand is a constant

int isVariable; // flag to indicate the operand is a variable

int startFound; // flag to indicate that the file reached START symbol

long endLoc; // location of END in the program

long firstExecLoc; // location of the first executable instruction

int isX; // flag to indicate a hexadecimal constant

char TEMP\_BUFFER [100] = {0}; // used to temporary store object code

FILE \*objFile;

char objName [200]; // store file name for object file

char lstName [200]; // store file name for immediate file

//// 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

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

long getOpcode(char \* mnemonic, int option);

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

\* 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();

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

\* getHashKeySymbol

\* returns the hash key for the given symbol

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

int getHashKeySymbol(char \* symbol);

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

\* getAddress

\* returns the address of the given symbol

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

long getAddress(char \* symbol);

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

\* getElement

\* returns table element for the given mnemonic.

\* returns null if the given mnemonic is not found in the hash table.

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

Table\_Element \* getElement(char \* mnemonic);

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

\* insertSymbolElement

\* insert new symbol to the symbol table

\* returns 1 if successfully inserted

\* returns 0 if there is already a duplicate in the table

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

int insertSymbolElement(char \* identifier, long address, char \* type, char \* value);

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

\* freeSymbolTable

\* free symbol table. This is called before assemble function is called

\* in order to erase the previous symbol table

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

void freeSymbolTable();

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

\* getByteSize

\* returns the byte size of the given string

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

long getByteSize(char \* str);

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

\* checkFilename

\* seperates the file name and extension and store them

\* into filename and extension array

\* returns 0 if the file is not found in the directory

\* return 1 otherwise.

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

int checkFilename();

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

\* showSymbol

\* prints out the symbol table

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

void showSymbol();

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

\* parseLine

\* parse the given line to calculate the location of the mnemonic,

\* addressing mode and seperate the operands

\* if option is 0 : for the given line of assembly code, inserts symbols

\* into SYMTAB if it exists

\* if option is 1 : does not insert symbol into SYMTAB

\* returns 1 if successful

\* returns 0 if there is an error

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

int parseLine( char \* line, int option);

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

\* getRegisterNumber

\* returns the register number

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

long getRegisterNumber(char \* r);

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

\* insertObjectCode

\* insert new object code node into the object code linked list

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

void insertObjectCode(long address, char \* objcode);

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

\* insertRelocationNode

\* insert new relocation node into the relocation linked list

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

void insertRelocationNode(long address);

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

\* calculateObjectCode

\* returns the object code of the given line

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

long calculateObjectCode(int numLine,char \* line);

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

\* typeFile

\* prints out the contents of the file

\* returns 0 if the file is not found in the directory

\* return 1 otherwise.

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

int typeFile(char \* fileName);

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

\* assemble

\* create immediate file and object file from the assembly file

\* returns 0 if there is an error in the assembly code

\* returns 1 if successful

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

int assemble(char \* fileName);

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

\* passOne

\* method used to read the assembly file for the first time

\* and calculate LOCC for each line and insert symbols into the symbol table

\* returns 0 if there is an error in the assembly code

\* returns 1 if successful

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

int passOne(char \* asmFileName);

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

\* passTwo

\* method called when reading the assembly file for the second time

\* With the symbol table created by passOne(), it now calculates the object code

\* for each line and prints out immediate file and object file

\* returns 0 if there is an error in the assembly code

\* returns 1 successful

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

int passTwo(char \* asmFileName);

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

\* printObjectFile

\* prints the contents of the object code linked list into the

\* object file and free the linked list

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

void printObjectFile();

5-1) 20161619.c

#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

if(strlen(command)>0)

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

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 SUCCESS;

}

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

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

}

int isSimpleInst(){

int i,j,flag,found = 0;

for (i = 0; i < 11; 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 typeFile(char \* file){ // print out the contents of file

char c;

DIR \* dir;

struct dirent \*entry = NULL;

FILE \* in;

dir = opendir("."); //현재디렉토리를 연다.

if(dir != NULL) {

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

if(strcmp(entry->d\_name, file) == 0) {

in = fopen(file,"r");

if(in == NULL) {

printf("Failed to open file named %s \n",file);

return ERROR;

}

while ((c = getc(in)) != EOF) // print out each character

putchar(c);

return SUCCESS;

}

}

closedir(dir);

printf("File does not exist in the current directory\n");

return ERROR;

}

fclose(in);

return SUCCESS;

}

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

if(type==ERROR\_PARAMETER)

printf("Wrong command format given..\n");

else if(type== ERROR\_ADDRESS\_OUT\_OF\_BOUND)

printf("Address out of bound..\n");

else if(type==ERROR\_INPUT\_FORMAT)

printf("Wrong parameter given..\n");

else if(type== ERROR\_PARAMETER\_LENGTH)

printf("Parameter length cannot be bigger than 9..\n");

else if(type== ERROR\_PARAMETER\_OUT\_OF\_BOUND)

printf("All parameter should be in between 0~0XFF..\n");

else

printf("Wrong command.\n");

}

int isComplexInst(){

instLength = 0;

for (int i = 0; i < 9; 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 checkFilename(){ // seperate filename and extension

int i,j,lenF,lenE, lenT;

int numOfPeriod = 0;

int filenameLength = 0;

lenF = 0,lenE = 0,lenT = 0;

memset(fullFileName,0,sizeof(fullFileName)); // reset char arrays

memset(filename,0,sizeof(filename));

memset(extension,0,sizeof(extension));

for(i = instLength; i < MAX\_USER\_INPUT; i++){ // ignore empty space

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

break;

}

}

if(i==MAX\_USER\_INPUT) // no filename is given

return ERROR;

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

if(userInput[j]=='/'){

printf("Operand should not be a directory..\n");

return ERROR;

}

if(isEmpty(userInput[j])){ // break if empty

break;

}

if(userInput[j]=='.'){ //

fullFileName[lenT++] = userInput[j];

numOfPeriod++;

}

if(numOfPeriod>1){ // wrong file extension

printf("Wrong file format\n");

return ERROR;

}

if(numOfPeriod==0 && userInput[j]!= '.'){

filename[lenF++] = userInput[j]; // store the filename

fullFileName[lenT++] = userInput[j];

}else if(numOfPeriod==1 && userInput[j]!='.'){ // store extension

extension[lenE++] = userInput[j];

fullFileName[lenT++] = userInput[j];

}

}

return SUCCESS;

}

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

printf("there should not be more than 3 parameters..\n");

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

if(end > MAX\_MEMORY\_SIZE-1 ) // if end address is bigger than MAX memory size

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

}

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);

if(end > MAX\_MEMORY\_SIZE-1 )// if end address is bigger than MAX memory size

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

}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(start > MAX\_MEMORY\_SIZE || end > MAX\_MEMORY\_SIZE || start < 0 || end < 0){ // print error message if start is bigger than max memory or less than zero printErrorMessage(ERROR\_ADDRESS\_OUT\_OF\_BOUND);

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;

}

if(end > MAX\_MEMORY\_SIZE-1 ){ // if end address is bigger than MAX memory size

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

}

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|| value < 0 ){ // 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 ERROR;

}

if (value > 0xFF || value < 0){

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

return ERROR;

}

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

Table\_Element \*elem;

int i = 0;

while(i<MAX\_HASH\_SIZE){

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,%02lX]",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");

i++;

}

}

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

Symbol\_Element \*elem;

int i = 0;

int found = 0;

while(i<SYMBOL\_HASH\_SIZE){

elem = SymbolTable[i]; // get table element from hash table[i]

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

i++;

continue;

}

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

found = 1;

printf("\t%s\t%04lX\n",elem->identifier,elem->address); // print out the mnemonic and opcode of the table element

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

}

i++;

}

if(!found){

printf("Symbol Table is not found\n");

}

}

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

int key = 0;

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

return key%MAX\_HASH\_SIZE;

}

int getHashKeySymbol(char \* symbol) { // returns the hash key for the symbol

int key = 0;

key = symbol[0]-'A'; // use first letter of symbol

return key%SYMBOL\_HASH\_SIZE;

}

long getAddress(char \* symbol){ // returns the address of the symbol

int key = getHashKeySymbol(symbol); // get hash key

for(Symbol\_Element \* temp = SymbolTable[key]; temp != NULL; temp= temp -> next){ // loop through the linked list

if(strcmp(temp->identifier,symbol)==0){ // if symbol is found

return temp->address; // return the address

}

}

return -1; // symbol not found

}

Table\_Element \* getElement(char \* mnemonic){

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

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

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

return temp;

}

}

return NULL;

}

long getOpcode(char \* mnemonic, int option){ // return opcode for the target mnemonic

Table\_Element \* elem = getElement(mnemonic);

if(elem==NULL){

if(option==0) // if option is 0, print out the result in the terminal

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

return -1;

}else{

if(option==0)

printf("opcode for %s is %02lX\n", mnemonic, elem->opcode);

return elem->opcode;

}

}

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;

newElem->next = NULL;

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

}

}

int insertSymbolElement(char \* identifier, long address, char \* type, char \* value){

Symbol\_Element \*newElem = (Symbol\_Element\*) malloc(sizeof(Symbol\_Element)); // allocate new symbol element

Symbol\_Element \*temp, \*previous = NULL;

strcpy(newElem->identifier,identifier); // copy identifier,type,value

strcpy(newElem->type,type);

strcpy(newElem->value,value);

newElem->address= address; // copy address

newElem->next = NULL;

int key = getHashKeySymbol(identifier); // get hash key for the following symbol

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

if(SymbolTable[key]->next==NULL){ // if there is only one element in the list

if(strcmp(SymbolTable[key]->identifier,identifier)==0){ // if the new node comes first

printf("[Error] in Assembly code. Symbol, %s is already defined\n", identifier);

return ERROR;

}else if(strcmp(SymbolTable[key]->identifier,identifier)>0){ // if the new node comes first

newElem->next = SymbolTable[key];

SymbolTable[key] = newElem;

}else{

SymbolTable[key]->next = newElem; // else add it to end of the element

}

return SUCCESS;

}

for(temp = SymbolTable[key]; temp->next != NULL; temp=temp->next){ // loop through the linked list

if(strcmp(temp->identifier,identifier)==0){ // ERROR in the assembly code

printf("[Error] in Assembly code. Symbol, %s is already defined\n", identifier);

return ERROR;

}else if(strcmp(temp->identifier,identifier)>0){ // if the new node comes first

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

newElem->next = temp;

SymbolTable[key] = newElem;

return SUCCESS;

}else {

newElem->next = previous->next; // insert it after previous

previous->next = newElem;

return SUCCESS;

}

}

previous = temp;

}

temp->next = newElem;

return SUCCESS;

// update the head to new node

}else { // if it is the first element

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

return SUCCESS;

}

}

void freeSymbolTable(){ // erase symbol table

for(int i = 0 ; i <= SYMBOL\_HASH\_SIZE-1; i++) { // loop through every hash key

Symbol\_Element \*remove\_node = SymbolTable[i]; // get head of the table

while(remove\_node != NULL && SymbolTable[i] != NULL) { // free each node

free(remove\_node);

SymbolTable[i] = SymbolTable[i]->next;

remove\_node = SymbolTable[i];

}

}

}

long getByteSize(char \* str){ // return byte size of the string

long count = 0;

char \* err;

if(str[0]=='C'){ // if first character indicate it is char

for(int i = 1 ; i < strlen(str); i++){

if(str[i]!='\''){ // count the num

count++;

}

}

}else if(str[0]=='X'){ // if the first character is X, it is 1 byte

count = 1;

}else{

return strtol(str,&err,10); // convert the decimal number

}

return count;

}

int parseLine( char \* line, int option ){ // parse the given line to calculate the location of the mnemonic,\* addressing mode and seperate the operands

int variableOrConstant = 0;

int numChar = 0;

int needToIncrement = 0;

int requireTwoOperands = 0;

char \* err;

numWord = 0;

locationOfMnemonic = -1;

format = 0;

isVariable = 0;

memset(operand,0,sizeof(operand)); // reset content of operand

if(line[0]=='.'){ // no need to do any calculation for comments

needToPrint = 0; // indicate LOCCTR and object code doesnt need to be printed

return SUCCESS;

}

for(int i = 0; i < line\_size; i++){ // loop through the line and seperate the line to store each operands

if(line[i]=='\0' || line[i]=='\n'){ // break if end of line

break;

}

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

requireTwoOperands = 1;

}

if(!isEmpty(line[i])&&line[i]!=','){ // store content if not comma or empty space

operand[numWord][numChar++] = line[i];

needToIncrement = 1;

if(isEmpty(line[i+1])|| line[i+1]==','){ // if the next character is empty or comma

numWord++; // increment numWord to store the next operand

numChar = 0; // initialize numChar

}

}

}

if(getOpcode(operand[0],1)==-1 ){ // When first operand is not mnemonic

if(strcmp(operand[0],"BASE")==0){ // do not need to print LOCCTR and object code for BASE

strcpy(base,operand[1]); // copy the operand

needToPrint = 0;

}

else if(strcmp(operand[1],"START")==0){ // START indicates the name of program and starting address

strcpy(title,operand[0]); // copy the title

LOCCTR = strtol(operand[2], &err, 16); // set LOCCTR

needToPrint = 0;

startFound = 1;

} else if(strcmp(operand[0], "END")==0){ // END indicates the end of instruction

LOCCTR += 1; // increment LOCCTR by 1

endFound = 1;

needToPrint = 0;

}else if(strcmp(operand[1],"RESB")==0){ // RESB symbol

LOCCTR += getByteSize(operand[2]); // increment by byte size of operand 2

variableOrConstant = 1;

isVariable = 1;

}else if(strcmp(operand[1],"RESW")==0){ // RESW symbol

LOCCTR += 3\*getByteSize(operand[2]); // increment by 3 \* byte size of operand 2

variableOrConstant = 1;

isVariable = 1;

}else if(strcmp(operand[1],"WORD")==0){ // WORD symbol

LOCCTR += 3\*getByteSize(operand[2]); // increment by 3\* byte size of operand 2

variableOrConstant = 1;

}else if(strcmp(operand[1],"BYTE")==0){ // BYTE Symbol

LOCCTR += getByteSize(operand[2]); // increment by byte size of operand 2

variableOrConstant = 1;

}else { // insert new symbol, which are name of sub-routines

if(numWord == 3 ){

if(option==0){ // insert it to symbol table

if(!insertSymbolElement(operand[0],previousLOCCTR,"ROUTINE","ROUTINE")){

return ERROR;

}

}

locationOfMnemonic = 1; // set the index of opcode

}else

locationOfMnemonic = 0;

}

if(variableOrConstant){ // insert variable or constant symbol

if(option==0){

if(!insertSymbolElement(operand[0],previousLOCCTR,operand[1],operand[2])){

return ERROR;

}

}

locationOfMnemonic = -1; // mnemonic not found

}

}else{

locationOfMnemonic = 0; // mnemonic is found in first operand

}

if(locationOfMnemonic>=0){ // if there is mnemonic

trueMnemonic = (char\*)malloc(sizeof(operand[locationOfMnemonic])); // copy the contents to trueMnemonic

strcpy(trueMnemonic,operand[locationOfMnemonic]);

if(operand[locationOfMnemonic][0]=='+'){ // if there is + in the first letter

format = 4; // addressing mode is four

trueMnemonic = strtok(trueMnemonic ,"+"); // remove + from the string

}

Table\_Element \*elem = getElement(trueMnemonic); // get element from opcode table

if(elem!=NULL){

if(strcmp(elem->format,"1")==0){ // increment LOCCTR by 1 if format 1

LOCCTR++;

format = 1;

}else if(strcmp(elem->format,"2")==0){ // increment LOCCTR by 2 if format 2

LOCCTR+=2;

format = 2;

}else if(strcmp(elem->format,"3/4")==0){ // increment LOCCTR accordingly

if(format==4){

LOCCTR+=4;

}else{

format = 3;

LOCCTR+=3;

}

}

}else{

return ERROR;

}

}

if(requireTwoOperands && operand[locationOfMnemonic+2][0]=='\0'){

return ERROR;

}

return SUCCESS;

}

long getRegisterNumber(char \* r){ // returns the register number for the given character

long objcode = 0;

if(strcmp(r,"A") == 0) {

objcode += 0;

} else if(strcmp(r,"X") == 0) {

objcode += 1;

} else if(strcmp(r,"L") == 0) {

objcode += 2;

} else if(strcmp(r,"B") == 0) {

objcode += 3;

} else if(strcmp(r,"S") == 0) {

objcode += 4;

} else if(strcmp(r,"T") == 0) {

objcode += 5;

} else if(strcmp(r,"F") == 0) {

objcode += 6;

} else if(strcmp(r,"PC") == 0) {

objcode += 8;

} else if(strcmp(r,"SW") == 0) {

objcode += 9;

}else

return -1;

return objcode\*16; // return in hexadecimal

}

void insertObjectCode(long address, char \* objcode){ // insert new object code into the linked list

Obj\_Element \* newNode = (Obj\_Element\*)malloc(sizeof(Obj\_Element)); // allocate memory for element

newNode->objcode = (char\*)malloc(sizeof(char)\*strlen(objcode)); // allocate memory for char array

strcpy(newNode->objcode,objcode); // copy object code and address

newNode->address = address;

newNode->next = NULL;

if(T\_head==NULL){ // if list is empty

T\_head = newNode; // update head and last

T\_last = T\_head;

}else{

T\_last->next = newNode; // add next to last

T\_last = newNode; // update last

}

}

void insertRelocationNode(long address){ // insert new relocation node into the linked list

Relocation\_Element \* newNode = (Relocation\_Element\*)malloc(sizeof(Relocation\_Element)); // allocate memory

newNode->address = address;

newNode->next = NULL;

if(R\_head==NULL){ // if no element in the list

R\_head = newNode; // update head and last

R\_last = R\_head;

}else{

R\_last->next = newNode; // add at the end of list

R\_last = newNode;

}

}

long calculateObjectCode(int numLine, char \* line){ // returns the object for the given line

long objectCode = 0;

long reg = 0;

long opcode;

long disp;

long labelLoc;

char \* err;

int j=0;

int constantValue=0;

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

isConstant = 0;

isX = 0;

int ascii;

char \* trueOperand;

char v[100];

memset(v,0,sizeof(v));

if(locationOfMnemonic==-1&&numWord>=2){ // calculate object code for constants

if(strcmp(operand[numWord-2],"BYTE")==0|| strcmp(operand[numWord-2],"WORD")==0){

if(operand[numWord-1][0]=='X'){ // for hexadecimal

for(int i = 1; i < strlen(operand[numWord-1]);i++){

if(operand[numWord-1][i]!='\''){

v[j++] =operand[numWord-1][i]; // copy the contents to v

}

}

isX = 1; // set flag

objectCode = strtol(v,&err,16); // convert hexadecimal to long

}else if(operand[numWord-1][0]=='C'){ // for characters

for(int i = 1; i < strlen(operand[numWord-1]);i++){

if(operand[numWord-1][i]!='\''){

objectCode \*= 256; // convert ASCII code to hexadecimal

objectCode += operand[numWord-1][i];

}

}

isConstant = 1; // set flag

}

return objectCode;

}else{

return -1; // mnemonic is not found and it is not constant, then ERROR

}

}

opcode = getOpcode(trueMnemonic,1); // get Opcode from mnemonic

if(strcmp(trueMnemonic,"RSUB")==0){ // if mnemonic is RSUB

objectCode = opcode+3; // n = 1 i = 1 since it's simple addressing

objectCode = objectCode << 16; // format 3

return objectCode;

}

if(format==1){ // object code is the opcode if it is format 1

objectCode = opcode;

objectCode = objectCode << 8;

return objectCode;

}

if(format==2 && numWord - locationOfMnemonic >= 1 ){ // FORMAT 2

objectCode = opcode;

objectCode = objectCode << 8;

reg = getRegisterNumber(operand[locationOfMnemonic+1]);

if(reg < 0){ // wrong register given

printf("[ERROR in line %d] in assembly code. Wrong register input..\n", numLine);

return -2;

}

objectCode += reg; // add register code

objectCode = objectCode << 4; // shift 4

if(numWord - locationOfMnemonic >= 2){ // if second register is given

reg = getRegisterNumber(operand[locationOfMnemonic+2]);

if(reg > 0){

objectCode += reg; // add second register

}

}

objectCode /= 16;

return objectCode;

}else{ // FORMAT 3 OR 4

objectCode = opcode;

if(numWord-locationOfMnemonic > 2){

if(strcmp(operand[locationOfMnemonic+2],"X")==0) // X register used

x = 1;

else if(operand[locationOfMnemonic+2][0]!='\0'){

printf("[ERROR in line %d] in Assembly code. 3/4 format can only have X as 2nd operand\n",numLine);

return -2;

}

}

if(format==4){ // Extension if format 4

e = 1;

}

if(numWord - locationOfMnemonic == 0){ // if there is no operand

printf("[ERROR in line %d] in Assembly code. There should be an operand!\n", numLine);

return -2;

}

if(operand[locationOfMnemonic+1][0]=='#'){ // immediate addressing

trueOperand = (char\*)malloc(sizeof(operand[locationOfMnemonic+1]));

strcpy( trueOperand ,operand[locationOfMnemonic+1]);

trueOperand = strtok( trueOperand ,"#");

n = 0;

i = 1;

}else if(operand[locationOfMnemonic+1][0]=='@'){ // indirect addressing

trueOperand = (char\*)malloc(sizeof(operand[locationOfMnemonic+1]));

strcpy( trueOperand ,operand[locationOfMnemonic+1]);

trueOperand = strtok( trueOperand ,"@");

n = 1;

i = 0;

}else { // simple addressing

trueOperand = (char\*)malloc(sizeof(operand[locationOfMnemonic+1]));

strcpy( trueOperand ,operand[locationOfMnemonic+1]);

n = 1;

i = 1;

}

if(trueOperand==NULL){

printf("[Error in line %d] Wrong operand..\n", numLine);

return -2;

}

if(trueOperand[0]>='A' && trueOperand[0]<='Z'){ // if operand is string

labelLoc = getAddress(trueOperand); // get address

if(labelLoc==-1){

printf("[ERROR in line %d ] in assembly code. %s is not found in symbol table\n", numLine, trueOperand);

return -2;

}

}else{

if(i==1){

constantValue = 1;

labelLoc = strtol(trueOperand, &err, 10); // convert decimal to long

}

}

if(constantValue==1) // if operand was constant value

disp = labelLoc; // set disp to labelLoc

else

disp = labelLoc - LOCCTR;

if(format==3){ // FORMAT 3

if(-2048 <= disp && disp <= 2047 && constantValue != 1) { // PC relative

p = 1;

b = 0;

if(disp < 0)

disp = 0x1000 + disp;

}else if (disp <= 4095 && constantValue != 1) { // BASE relative

disp = labelLoc - baseLoc;

b = 1;

p = 0;

}else{ // direct addressing

b = 0;

p = 0;

}

}else{ // format 4

disp = labelLoc; // use absolute address instead of relative addressing

b = 0;

p = 0;

}

objectCode = opcode; // add n,i,x,b,p,e

objectCode = objectCode + n\*2 + i;

objectCode = objectCode << 4;

objectCode = objectCode + x\*8 + b\*4 + p\*2 + e;

if(format==4){

objectCode = objectCode << 20; // shift 20 bits if format 4

}else

objectCode = objectCode << 12; // shift 12 bits if format 3

objectCode += disp; // add displacement or address to object code

}

return objectCode;

}

void printObjectFile(){ // print T node to object file and initialize the linked list

Obj\_Element \* rem;

fprintf(objFile,"T%06lX%02lX", T\_head->address, previousLOCCTR-T\_head->address); // print the starting address

while(T\_head != NULL){ // loop until the end of list

rem = T\_head;

T\_head = T\_head->next;

fprintf(objFile,"%s", rem->objcode); // print object code

free(rem); // free the node

}

T\_head = NULL; // initialize head and last to null

T\_last = NULL;

fprintf(objFile,"\n"); // print new line

}

int passTwo(char \* asmFileName){ // read each line in assembly file and generates .lst and .obj file

memset(lstName,0,sizeof(lstName)); // initalize array

memset(objName,0,sizeof(objName));

strcpy(objName,filename); // copy the file name

strcpy(lstName,filename);

strcat(objName,".obj"); // add extensions

strcat(lstName,".lst");

objFile = fopen(objName,"w");

FILE \*asmFile = fopen(asmFileName,"r");

FILE \*lstFile = fopen(lstName,"w");

Obj\_Element \* rem;

LOCCTR = 0;

previousLOCCTR = 0;

endFound = 0;

firstExecLoc = -1;

startFound = 0;

char c;

int i = 0;

long objectCode;

int numOfLines = 0;

char line[line\_size];

if(asmFile==NULL){

printf("Failed to assemble. File named %s not found\n",asmFileName);

return ERROR;

}

while(i < line\_size){

c = fgetc(asmFile); // read each character in the file

if( feof(asmFile) ) { // break if reached end of file

break ;

}

if(c=='\n'){ // for every new line

numOfLines+=5; // increment line number

needToPrint = 1;

parseLine(line,1); // parse line to increment LOCCTR and seperate operands

memset(TEMP\_BUFFER,0,sizeof(TEMP\_BUFFER)); // intialize buffer

if(startFound==1){ // START is found

fprintf(objFile,"H%s\t%012lX\n",title,endLoc); // print H node

startFound = 2;

}

if(needToPrint){ // if the line was not a comment or variable

if(firstExecLoc==-1){

firstExecLoc = previousLOCCTR; // store location of first executable code

}

objectCode = calculateObjectCode(numOfLines,line); // calculate object code

if(objectCode >= 0){ // if object code was calculated

if(isX){ // for hexadecimal

fprintf(lstFile, "%-5d\t%04lX\t%-8s\t\t%02lX\n", numOfLines,previousLOCCTR,line,objectCode);

sprintf(TEMP\_BUFFER,"%02lX",objectCode);

}else if(isConstant){ // for constant

fprintf(lstFile, "%-5d\t%04lX\t%-8s\t\t%lX\n", numOfLines,previousLOCCTR,line,objectCode);

sprintf(TEMP\_BUFFER,"%lX",objectCode);

}else if(format==1|| format==2 || strcmp(trueMnemonic,"RSUB")==0 ){ // For format 1, 2 or RSUB

fprintf(lstFile, "%-5d\t%04lX\t%-8s\t\t",numOfLines,previousLOCCTR,line);

if(numWord==2 || (numWord==3 && strcmp(operand[locationOfMnemonic+1],"X")==0)||strcmp(trueMnemonic,"RSUB")==0)

fprintf(lstFile,"\t");

fprintf(lstFile,"%lX\n", objectCode);

sprintf(TEMP\_BUFFER,"%lX",objectCode);

}

else{

fprintf(lstFile, "%-5d\t%04lX\t%-8s\t\t%06lX\n", numOfLines,previousLOCCTR,line,objectCode);

sprintf(TEMP\_BUFFER,"%06lX",objectCode);

}

if(format==4&&operand[locationOfMnemonic+1][0]!='#'&&operand[locationOfMnemonic+1][0]!='@'){ // insert format 4 and simple addressing for relocation

insertRelocationNode(previousLOCCTR);

}

if(T\_head!=NULL&&LOCCTR-T\_head->address > 0x1E) { // print T records if buffer is full

printObjectFile();

}

insertObjectCode(previousLOCCTR,TEMP\_BUFFER); // insert new object code

}else{

if(objectCode==-2){ // ERROR in assembly code

return ERROR;

}

if(isVariable&&T\_head!=NULL&&T\_last!=NULL){ // for every variable statement

printObjectFile(); // empty the object linked list and print the content

}

fprintf(lstFile, "%-5d\t%04lX\t%-12s\n", numOfLines,previousLOCCTR,line); // print line # and locctr and content of line

}

}

else{

fprintf(lstFile, "%-5d\t\t%-12s\n", numOfLines,line); // print only line # and content of line if not executable code

}

previousLOCCTR = LOCCTR; // store previou locctr

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

i = 0;

if(endFound){

break; // stop calculating if reached END statement

}

}else {

line[i++] = c; // if not new line, insert it to line.

}

}

if(T\_head!=NULL&&T\_last!=NULL){ // print out the rest of the bject codes

printObjectFile();

}

if(R\_head!=NULL&&R\_last!=NULL){ // print out M records

while(R\_head != NULL){

Relocation\_Element \* temp = R\_head;

R\_head = R\_head->next;

fprintf(objFile,"M%06lX05\n", temp->address+1);

free(temp);

}

R\_head = NULL;

R\_last = NULL;

}

fprintf(objFile,"E%06lX\n",firstExecLoc); // print E node

fclose(lstFile);

fclose(asmFile); // close files

fclose(objFile);

return SUCCESS;

}

int passOne(char \* asmFileName){ // read each line in assembly file to update symbol table and LOCCTR

FILE \*asmFile = fopen(asmFileName,"r");

LOCCTR = 0;

previousLOCCTR = 0;

endFound = 0;

endLoc = 0;

int numOfLines = 0;

char line[line\_size];

char c;

int i = 0;

if(asmFile==NULL){ // cannot open file

printf("Failed to assemble. file named %s not found in this directory\n", asmFileName);

return ERROR;

}

while(i < line\_size ){

c = fgetc(asmFile); // read one letter at a time

if( feof(asmFile) ) { // break if reached the end of file

break ;

}

if(c=='\n'){ // for each new line

numOfLines+=5; // increment line number

needToPrint = 1;

if(!parseLine(line,0)) {// parse line to update LOCCTR and insert symbol to symbol table

printf("[ERROR in line : %d] in Assembly code. Failed to assemble.\n", numOfLines);

return ERROR;

}

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

i = 0;

if(endFound){ // if reached END statement, store address to endLoc

endLoc = previousLOCCTR;

break;

}

previousLOCCTR = LOCCTR;

}else {

line[i++] = c; // insert it to line

}

}

if(!startFound){ // print error if start is not found

printf("[ERROR] in assembly code. START is not found.\n");

return ERROR;

}

if(!endFound){ // print error if end is not found

printf("[ERROR] in assembly code. END is not found!\n");

return ERROR;

}

fclose(asmFile); // close file

return SUCCESS;

}

int assemble(char \* fileName){ // assemble assembly file

char \* err;

memset(base,0,sizeof(base));

if(strcmp(extension,"asm")!=0){ // return error if file extension is not .asm

printf("This machine can only assemble .asm file..\n");

return ERROR;

}

freeSymbolTable(); // initialize symbol table

if(!passOne(fileName)){ // pass 1 to update symbol table and LOCCTR

freeSymbolTable();

return ERROR;

}

if(base[0]!='\0'){ // update base register

baseLoc = getAddress(base);

if(baseLoc==-1){

printf("[ERROR in assembly code] BASE register is defined incorrectly\n");

return ERROR;

}

}else

baseLoc = 0;

if(!passTwo(fileName)){ // pass 2 to generate .obj and .lst files

remove(objName);

remove(lstName); // delete files if failed to assemble

freeSymbolTable();

return ERROR;

}

printf("successfully assembled %s\n", fileName);

return SUCCESS;

}

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(); // remove unnessary empty string

if (isSimpleInst()) { // user command is one of the simple insturctions

historyCount++;

updateHistory(userInput);

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

return QUIT;

}

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

printAllCommands();

}

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

listdir(".");

}

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

showHistory();

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

resetMemory();

}else if(strcmp(command,"opcodelist")==0){ // print opcode list

showOpcode();

}else if(strcmp(command,"symbol")==0){ // print symbol table

showSymbol();

}

}

else if (isComplexInst()){ // user command is one of the complex instructions

instLength = strlen(command);

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

successful = dumpMemory();

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

successful = editMemory();

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

successful = fillMemory();

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

successful = getOpcode(targetMnemonic,0);

}else if(strcmp(command,"type")==0 && checkFilename()){ // type

successful = typeFile(fullFileName);

}else if(strcmp(command,"assemble")==0 && checkFilename()){ // assemble

successful = assemble(fullFileName);

}

}

if(successful){ // insert to history list only if command was successful

historyCount++;

updateHistory(userInput);

}

return SUCCESS;

}

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;

}