과목명**:** 시스템프로그래밍

1분반 박운상 교수님

**<<Project #3>>**

서강대학교 **[**학부명:컴퓨터공학과**]**

**[**학번:20161619**]**

**[**이름:이세진**]**

Contents

1. Program Summary
2. Program Description

2.1. Flowchart for PROGADDR command

2.2. Flowchart for BP command

2.3. Flowchart for Load Command

2.4. Flowchart for Run Command

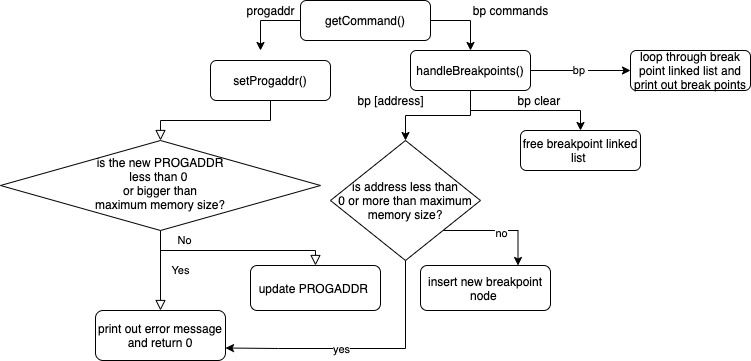
1. Global Variables
2. User Defined Modules
3. Code Description

**1. Program Summary**

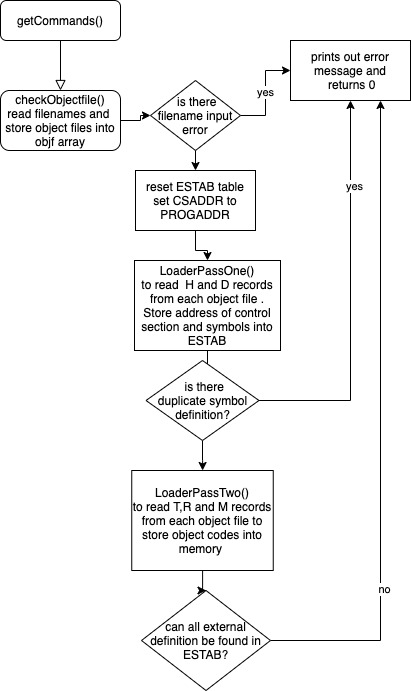
This project implements linker and loader, which links object files created by assemble command implemented in project 2. By linking multiple control section, the program loads values to the memory of SIC/XE machine.

**2. Program Description**

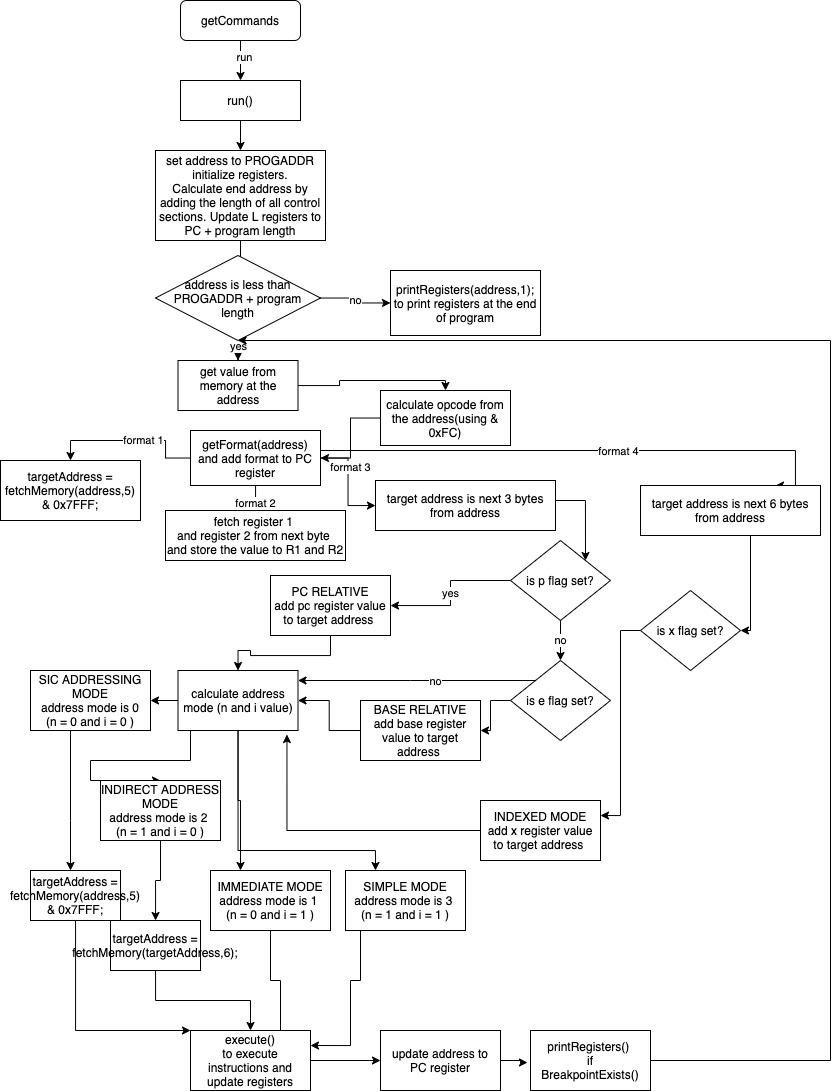
***2.1. Flowchart for PROGADDR and BP command***



***2.2. Flowchart for Load Command***

******

***2.3. Flowchart for Run Command***

******

**3. Global Variables**

Please note that previously defined variables in project 1 and 2 are excluded from this list. Only linker related variables will be described in this section.

**typedef struct ESTAB\_Table{ // node used to store external symbols**

**long address;**

**long length; // length only used for control section name**

**char symbol[100];**

**struct ESTAB\_Table \* next;**

**}ESTAB\_Table;**

**typedef struct Break\_Point{ // node used to store break points**

**int visited; // if break point was already visited**

**long address;**

**struct Break\_Point \* next;**

**}Break\_Point;**

**long PROGADDR = 0; //Program Load Address. Starting address in memory where the linked program is to be loaded.**

**long CSADDR = 0; // Control Section Address.Starting address assigned to the control section currently being scanner by the loader**

**long CSLTH = 0; // Length of Control Section**

**long EXECADDR = 0; // Execution control section address**

**int numOfFile; // num of files given in loader command**

**int currentFileNum; // current file number being loaded**

**FILE \* objf[3]; // store 3 object files**

**long last\_address = 0; // store address of last executed address**

**int bpNum = 0; // number of breakpoints**

**long REG[9]; // store register values**

**int R1, R2 = 0; // register 1 and 2 for format 2**

**char CC; // conditional flag**

**ESTAB\_Table \* ESTAB[3]; // ESTAB for each object file**

**Break\_Point \* bHead; // head node for break point linked list**

**4. User-defined Modules**

\* Note that functions that were defined in project 1 and 2 are excluded from this list.

**4.1 int setProgaddr();**

\* set program start address

\* returns 0 if failed to set PROGADDR(Out of range).

\* returns 1 if successful

**4.2 int handleBreakpoint();**

\* handles printing out breakpoints, clearing breakpoint and adding

\* new breakpoint. returns 0 if error 1 if successful

**4.3 void resetESTAB();**

\* initialize ESTAB. Called each time new program is loaded

**4.4 ESTAB\_Table \* findESTAB(char \* symbol);**

\* find external symbol from ESTAB table

\* returns null if not found

**4.5 int insertESTAB(char \* symbol, long address, long length);**

\* insert new external symbol to ESTAB

\* returns 0 if error(duplicate symbol definition)

\* returns 1 if successful

**4.6 int LoaderPassOne();**

\* pass one of loading and linking process.

\* reads H, E and D record to insert symbols into ESTAB, which will be used in pass 2

\* returns 1 if successful and 0 otherwise

**4.7 int LoaderPassTwo();**

\* uses values from ESTAB updated by load pass 1

\* to read T,M and R records and update memory

\*return 1 if successful and 0 otherwise

**4.8 int loader();**

\* uses LoaderPassOne and LoaderPassTwo to load program into memory

\* returns 1 if successful and 0 otherwise

**4.9 int checkObjectfile();**

\* reads object file name from command and

\* stores file into objf array. returns 0 if there is error in command

\* returns 1 if successful.

**4.10 void COMPMnemonic(long a, long b);**

\* used for COMP, COMPF, COMPR opcodes

\* compares a and b and stores the result into CC

**4.11 int fetchMemory(int address, int hBytes);**

\* fetch value from memory at the given address

\* according to number of half bytes

**4.12 void storeMemory(long address, int bytes, long value);**

\* store value to memory at given address

**4.13 int getFormat(long address);**

\* returns address format of the object code at the given address

**4.14 void execute(long opcode, long targetAddress, int addressMode, int format );**

\* execute instruction using the opcode, target address, address mode

\* and address format

**4.15 int breakpointExists(long address);**

\* returns 1 if breakpoint exists in the given address

\* returns 0 if not

**4.16 void printRegisters(long address, int option);**

\* prints out the values of registers at the given address

\* if option is 0, the current address is breakpoint

\* if option is 1, the current address is the end address of the program

**4.17 int run();**

\* run program, starting at address of PROGADDR

\* returns 1 if successful

\* returns 0 if error exists

**5. Code Description**

**5-1) 20161619.h (only parts related to project 3)**

**typedef struct ESTAB\_Table{ // node used to store external symbols**

**long address;**

**long length; // length only used for control section name**

**char symbol[100];**

**struct ESTAB\_Table \* next;**

**}ESTAB\_Table;**

**typedef struct Break\_Point{ // node used to store break points**

**int visited; // if break point was already visited**

**long address;**

**struct Break\_Point \* next;**

**}Break\_Point;**

**long PROGADDR = 0; //Program Load Address. Starting address in memory where the linked program is to be loaded.**

**long CSADDR = 0; // Control Section Address.Starting address assigned to the control section currently being scanner by the loader**

**long CSLTH = 0; // Length of Control Section**

**long EXECADDR = 0; // Execution control section address**

**int numOfFile; // num of files given in loader command**

**int currentFileNum; // current file number being loaded**

**FILE \* objf[3]; // store 3 object files**

**long last\_address = 0; // store address of last executed address**

**int bpNum = 0; // number of break points**

**long REG[9]; // store register values**

**int R1, R2 = 0; // register 1 and 2 for format 2**

**char CC; // conditional flag**

**ESTAB\_Table \* ESTAB[3]; // ESTAB for each object file**

**Break\_Point \* bHead; // head node for break point linked list**

**typedef enum{**

**A = 0,**

**X = 1,**

**L = 2,**

**B = 3,**

**S = 4,**

**T = 5,**

**F = 6,**

**PC = 7,**

**SW = 8**

**}Register; // Register number**

**typedef enum{ // opcode for mnemonics**

**ADD = 0x18,**

**ADDF = 0x58,**

**ADDR = 0x90,**

**AND = 0x40,**

**CLEAR = 0xB4,**

**COMP = 0x28,**

**COMPF = 0x88,**

**COMPR = 0xA0,**

**DIV = 0x24,**

**DIVF = 0x64,**

**DIVR = 0x9C,**

**FIX = 0xC4,**

**FLOAT = 0xC0,**

**HIO = 0xF4,**

**J = 0x3C,**

**JEQ = 0x30,**

**JGT = 0x34,**

**JLT = 0x38,**

**JSUB = 0x48,**

**LDA = 0x00,**

**LDB = 0x68,**

**LDCH = 0x50,**

**LDF = 0x70,**

**LDL = 0x08,**

**LDS = 0x6C,**

**LDT = 0x74,**

**LDX = 0x04,**

**LPS = 0xD0,**

**MUL = 0x20,**

**MULF = 0x60,**

**MULR = 0x98,**

**NORM = 0xC8,**

**OR = 0x44,**

**RD = 0xD8,**

**RMO = 0xAC,**

**RSUB = 0x4C,**

**SHIFTL = 0xA4,**

**SIO = 0xF0,**

**SSK = 0xEC,**

**STA = 0x0C,**

**STB = 0x78,**

**STCH = 0x54,**

**STF = 0x80,**

**STI = 0xD4,**

**STL = 0x14,**

**STS = 0x7C,**

**STSW = 0xE8,**

**STT = 0x84,**

**STX = 0x10,**

**SUB = 0x1C,**

**SUBF = 0x5C,**

**SUBR = 0x94,**

**SVC = 0xB0,**

**TD = 0xE0,**

**TIO = 0xF8,**

**TIX = 0x2C,**

**TIXR = 0xB8,**

**WD = 0xDC**

**}MNENOMIC;**

**//// USER-DEFINED FUNCTIONS**

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

**\* setProgaddr**

**\* set program start address**

**\* returns 0 if failed to set PROGADDR(Out of range).**

**\* returns 1 if successful**

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

**int setProgaddr();**

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

**\* handleBreakpoint**

**\* handles printing out breakpoints, clearing breakpoint and adding**

**\* new breakpoint. returns 0 if error 1 if successful**

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

**int handleBreakpoint();**

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

**\* resetESTAB**

**\* initalize ESTAB. Called each time new program is loaded**

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

**void resetESTAB();**

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

**\* findESTAB**

**\* find external symbol from ESTAB table**

**\* returns null if not found**

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

**ESTAB\_Table \* findESTAB(char \* symbol);**

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

**\* insertESTAB**

**\* insert new external symbol to ESTAB**

**\* returns 0 if error(duplicate symbol definition)**

**\* returns 1 if successful**

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

**int insertESTAB(char \* symbol, long address, long length);**

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

**\* LoaderPassOne**

**\* pass one of loading and linking process.**

**\* reads H, E and D record to insert symbols into ESTAB, which**

**\* will be used in pass 2**

**\* returns 1 if successful and 0 otherwise**

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

**int LoaderPassOne();**

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

**\* LoaderPassTwo**

**\* uses values from ESTAB updated by load pass 1**

**\* to read T,M and R records and update memory**

**\* return 1 if successful and 0 otherwise**

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

**int LoaderPassTwo();**

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

**\* loader**

**\* uses LoaderPassOne and LoaderPassTwo to load program into memory**

**\* returns 1 if successful and 0 otherwise**

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

**int loader();**

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

**\* checkObjectfile**

**\* reads object file name from command and**

**\* stores file into objf array. returns 0 if there is error in command**

**\* returns 1 if successful.**

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

**int checkObjectfile();**

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

**\* COMPMnemonic**

**\* used for COMP, COMPF, COMPR opcodes**

**\* compares a and b and stores the result into CC**

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

**void COMPMnemonic(long a, long b);**

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

**\* fetchMemory**

**\* fetch value from memory at the given address**

**\* according to number of half bytes**

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

**int fetchMemory(int address, int hBytes);**

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

**\* storeMemory**

**\* store value to memory at given address**

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

**void storeMemory(long address, int bytes, long value);**

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

**\* getFormat**

**\* returns address format of the object code at the given address**

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

**int getFormat(long address);**

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

**\* execute**

**\* execute instruction using the opcode, target address, address mode**

**\* and address format**

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

**void execute(long opcode, long targetAddress, int addressMode, int format );**

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

**\* breakpointExists**

**\* returns 1 if breakpoint exists in the given address**

**\* returns 0 if not**

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

**int breakpointExists(long address);**

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

**\* printRegisters**

**\* prints out the values of registers at the given address**

**\* if option is 0, the current address is breakpoint**

**\* if option is 1, the current address is the end address of the program**

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

**void printRegisters(long address, int option);**

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

**\* run**

**\* run program, starting at address of PROGADDR**

**\* returns 1 if successful**

**\* returns 0 if error exists**

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

**int run();**

**5-2) 20161619.c (excluded part of code from project 1 and 2)**

int setProgaddr(){ // set program start address

char \* err;

if(numOfParams!=1 || isEmpty(params[0][0])){

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

return ERROR;

}

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

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

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

PROGADDR = 0;

return ERROR;

}

PROGADDR = temp;

return SUCCESS;

}

int handleBreakpoint(){ // bp command

char p[100];

int idx = 0;

int check = 0;

memset(p,0,sizeof(p));

for(int i = instLength+1; i < MAX\_USER\_INPUT; i++){

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

p[idx++]=userInput[i];

}

if(isEmpty(userInput[i]) && idx!=0){

break;

}

}

if(idx==0){

// print break points;

printf("\t\tbreakpoint\n");

printf("\t\t--------\n");

for(Break\_Point \* temp = bHead; temp!=NULL; temp=temp->next){

printf("\t\t%lX\n",temp->address);

}

return SUCCESS;

}else if(strcmp("clear",p)==0){ // clear break points

for(Break\_Point \* temp = bHead; temp!=NULL;){

Break\_Point \* deleteNode = temp;

temp=temp->next;

free(deleteNode);

}

bHead = NULL;

printf("\t\t[ok] clear all breakpoints \n");

return SUCCESS;

}else{ // insert new break points

for(int i = 0 ; i < strlen(p); i++){ // check format

if(!isHexadecimal(p[i])){

printf("Address should be given in hexadecimal format.\n");

return SUCCESS;

}

}

long hexValue = strtol(p, NULL, 16); // convert each parameter to long type

if(hexValue > MAX\_MEMORY\_SIZE || hexValue < 0){ // SHOULD NOT EXCEED MAX LENGTH

printf("breakpoints cannot exceed max memory size..\n");

return ERROR;

}

Break\_Point \* temp;

Break\_Point \* newNode = (Break\_Point\*)malloc(sizeof(Break\_Point)); // create new node

newNode->address = hexValue;

newNode->visited = 0;

newNode->next = NULL;

if(bHead==NULL){ // if no previous node

bHead = newNode;

}else {

for(temp = bHead; temp->next!=NULL; temp=temp->next); // attach to end of previous node

temp->next = newNode;

}

printf("\t\t[ok] create breakpoint %s\n", p);

return SUCCESS;

}

}

void resetESTAB(){ // initialize estab

for(int i = 0 ; i < 3; i++)

ESTAB[i] = NULL;

}

ESTAB\_Table \* findESTAB(char \* symbol){ // find symbol in ESTAB table

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

for(ESTAB\_Table \* temp=ESTAB[i]; temp!=NULL; temp=temp->next){

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

return temp;

}

}

}

return NULL; // return null if not found

}

int insertESTAB(char \* symbol, long address, long length){ // insert symbol into ESTAB table

ESTAB\_Table \* newElem = (ESTAB\_Table \*) malloc(sizeof(ESTAB\_Table));

newElem->address = address;

newElem->length = length;

newElem->next=NULL;

memset(newElem->symbol,0,sizeof(newElem->symbol));

strcpy(newElem->symbol,symbol);

ESTAB\_Table \* temp;

if(ESTAB[currentFileNum]==NULL){ // if control section name is not inserted yet

for(int i = 0 ; i < 3; i++){

if(i==currentFileNum){

continue;

}

for(temp = ESTAB[i]; temp!=NULL; temp=temp->next){

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

printf("Duplicate External symbol %s\n", symbol);

return ERROR;

}

}

}

ESTAB[currentFileNum]=newElem;

}else{ // insert external symbol

for(temp = ESTAB[currentFileNum]; temp->next!=NULL; temp=temp->next){

if(strcmp(temp->symbol,symbol)==0){ // check for duplicates

printf("Duplicate External symbol\n");

return ERROR;

}

}

temp->next = newElem;

for(int i = 0 ; i < 3; i++){

if(i!=currentFileNum){

for(temp = ESTAB[i]; temp!=NULL; temp=temp->next){

if(strcmp(temp->symbol,symbol)==0){ // check for duplicates

printf("Duplicate External symbol\n");

return ERROR;

}

}

}

}

}

return SUCCESS;

}

int LoaderPassOne(){ // check for head and definition records

char line[1000];

char addr[100];

char symbol[8];

char \* err;

CSADDR = PROGADDR; // set csaddr to progaddr for first control section

long len = 0;

long address;

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

currentFileNum = i;

while(fgets(line,sizeof(line),objf[i])!=NULL){ // while end of input

memset(symbol,0,sizeof(symbol));

memset(addr,0,sizeof(address));

if(line[0]=='H'){ // HEADER record

strncpy(symbol,line+1,6);

CSLTH= strtol(line+13,&err,16); // set cslth to control section length

len += CSLTH;

if(!insertESTAB(symbol,CSADDR,CSLTH)){ // enter control section name into ESTAB with value CSADDR

printf("Control section named %s already exists\n",symbol);

return ERROR;

}

}

else if(line[0]=='D'){ // EXTERNAL DEFINITION

for(int j = 0; j < strlen(line)/2; j += 12) { // read each symbol

strncpy(symbol,line+1+j,6);

strncpy(addr,line+7+j,6); // indicated address

for(int k=0; k<6; k++) {

if(isEmpty(symbol[k]))

symbol[k] = '\0';

}

address = strtol(addr,&err,16);

if(!insertESTAB(symbol, CSADDR+address,0)) { // insert symbol into ESTAB with value(CSADDR + indicated address)

printf("Symbol already exists in the ESTAB.\n");

return ERROR;

}

}

}else if(line[0]=='E'){ // END Record

CSADDR += CSLTH; // update starting address for next control section

break;

}

}

}

return SUCCESS;

}

int LoaderPassTwo(){ // check for T, R, M records

char line[1000];

char addr[100];

char symbol[100];

char objectCode[100];

char length[10];

char onebyte[3];

char \* err;

CSADDR = PROGADDR;

EXECADDR = PROGADDR; // set CSADDR and EXECADDR to PROGADDR

long len = 0;

long address;

long value;

char ref[1000][10];

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

currentFileNum = i;

memset(ref,0,sizeof(ref));

strcpy(ref[1], ESTAB[currentFileNum]->symbol);

while(fgets(line,sizeof(line),objf[i])!=NULL){ // while end of input

memset(onebyte,0,sizeof(onebyte));

memset(addr,0,sizeof(addr));

memset(symbol,0,sizeof(symbol));

memset(objectCode, 0, sizeof(objectCode));

if(line[0]=='H'){ // HEADER record

CSLTH= strtol(line+13,&err,16); // set cslth to control section length

len += CSLTH;

}

else if(line[0]=='T'){ // Text Record

strncpy(addr,line+1,6);

strncpy(length,line+7,2);

address = strtol(addr,&err,16);

address += CSADDR; // move object code from record to location

len = strtol(length,&err,16);

for(int j = 0; j < len; j++){

strncpy(onebyte,line+9+j\*2,2);

onebyte[2]='\0';

value = strtol(onebyte,&err,16);

VMemory[address++] = value;

}

}else if(line[0]=='R'){ // External Reference

for(int rn = 2; rn < (strlen(line)-1)/8+3; rn++){

strncpy(symbol, line+(rn-2)\*8+3,6);

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

if(symbol[k]==' ' || symbol[k] == '\n')

symbol[k] = '\0';

}

if(ref[rn][0]!='\0'){

printf("Reference number %lX is assigned for multiple times.\n",value);

return ERROR;

}

strcpy(ref[rn],symbol);

}

}else if(line[0]=='M'){ // Modificcation Record

strncpy(addr,line+1,6);

address = strtol(addr,&err,16);

address += CSADDR; // move object code from record to location

strncpy(onebyte,line+10,2); // get reference number

value = strtol(onebyte,&err,16);

if(value==0){

printf("Reference number is out of range..\n");

return ERROR;

}

strcpy(symbol,ref[value]); // find external reference symbol

ESTAB\_Table \* refElem = findESTAB(symbol);

if(refElem==NULL){

printf("External symbol is not defined in ESTAB table\n");

return ERROR;

}

for(int j = 0; j < 3; j++){

sprintf(onebyte,"%02X", VMemory[address+j]);

strcat(objectCode,onebyte);

}

long objectValue = strtol(objectCode,&err,16);

if(('A' <= objectCode[0] && objectCode[0] <= 'F') || ('8'<= objectCode[0] && '9' >= objectCode[0])){

objectValue = 0xFFFFFF - objectValue + 1;

objectValue = -objectValue;

}

objectValue += (line[9] == '-' ? -1 : 1) \* refElem->address; // add or subtract symbol address

if(objectValue < 0) { // check for negative

objectValue -= 0xFFFFFFFFFF000000;

}

sprintf(objectCode,"%06lX",objectValue);

for(int j = 0; j < 6; j+=2){

strncpy(onebyte,objectCode+j,2);

onebyte[2]='\0';

value = strtol(onebyte,&err,16);

VMemory[address++] = value;

}

}

else if(line[0]=='E'){ // END Record

if(!isEmpty(line[1])) { // if end address is specified

strncpy(addr,line+1,6);

EXECADDR = CSADDR;

EXECADDR += strtol(addr,&err,16);

}

CSADDR += CSLTH; // update starting address for next control section

break;

}

}

}

return SUCCESS;

}

int loader(){

resetESTAB(); // initalize ESTAB

if(!LoaderPassOne()) // pass 1 for constructing ESTAB

return ERROR;

for(int i = 0 ; i < numOfFile;i++) // reset file pointer to start of file

rewind(objf[i]);

if(!LoaderPassTwo()) // pass 2 for loading object code into memory

return ERROR;

REG[L] = CSADDR; // update L register

// PRINT ESTAB INFO

printf("control symbol address length\n");

printf("section name\n");

printf("--------------------------------\n");

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

for(ESTAB\_Table \* temp = ESTAB[i]; temp!=NULL; temp= temp->next){

if(temp==ESTAB[i]){

printf("%s\t\t%04lX\t%04lX\n", temp->symbol, temp->address, temp->length);

}else{

printf("\t%s\t%04lX\n",temp->symbol,temp->address);

}

}

}

printf("--------------------------------\n");

printf("\ttotal length %04lX\n", CSADDR-PROGADDR);

return SUCCESS;

}

int checkObjectfile(){

char filename[3][100];

char temp[100];

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

numOfFile = sscanf(userInput,"%100s %100s %100s %100s\n", temp, filename[0],filename[1],filename[2])-1;

if(numOfFile==0){ // no filename is given

printf("At least one input file is required..\n");

return ERROR;

}

for(int i =0; i < numOfFile;i++){ // loop through all filename

char \* ext = strrchr(filename[i], '.');

if(strcmp(ext,".obj")!=0){ // if not object file

printf("Only object file can be loaded..\n");

return ERROR;

}

objf[i] = fopen(filename[i],"r");

if(objf[i]==NULL){ // if file cannot be found

printf("filename %s not found in directory..\n",filename[i]);

return ERROR;

}

}

return SUCCESS;

}

void COMPMnemonic(long a, long b){ // compare a and b and set CC

if(a < b){

CC = '<';

}else if(a > b){

CC = '>';

}else

CC = '=';

}

int fetchMemory(int address, int hBytes) {

int value = 0;

int limit = (hBytes - 1 )/2;

if(address>MAX\_MEMORY\_SIZE-1)

return value;

if(hBytes%2){ // set initial value via bit operation

value = VMemory[address] % 0x10;

}else{

value = VMemory[address] % 0x100;

}

for(int i = 1; i <= limit; i++) {

value \*= 0x100; // increase byte

value += VMemory[address + i];

}

return value;

}

void storeMemory(long address, int bytes, long value) { // store value into VMemory

int i = address + bytes - 1;

for(; i >= address && address < MAX\_MEMORY\_SIZE ; i--) {

VMemory[i] = value & 0xFF;

value /= 0x100;

}

}

void execute(long opcode, long targetAddress, int addressMode, int format ){ // execute instruction

long value;

value = targetAddress;

if(addressMode==3){ // if simple addressing mode

value = fetchMemory(targetAddress,6);

}

switch(opcode){ // execute approp

// ARITHMETICS

case ADD : REG[A] = REG[A] + value; break;

case ADDF : REG[F] = REG[F] + value; break;

case ADDR : REG[R2] = REG[R2] + REG[R1]; break;

case SUB : REG[A] = REG[A] - value; break;

case SUBF : REG[F] = REG[F] - value; break;

case SUBR : REG[R2] = REG[R2] - REG[R1]; break;

case DIV : if(value!=0)

REG[A] = REG[A] / value; break;

case DIVF : if(value!=0)

REG[F] = REG[F] / value; break;

case DIVR : if(REG[R1]!=0)

REG[R2] = REG[R2] / REG[R1];

break;

case MUL : REG[A] = REG[A] \* value; break;

case MULF : REG[F] = REG[F] \* value; break;

case MULR : REG[R2] = REG[R1] \* REG[R2]; break;

// LOAD AND STORE

case LDA : REG[A] = value; break;

case LDB : REG[B] = value; break;

case LDCH : REG[A] = REG[A] & 0xFFFFFF00; // lower 3 bytes

REG[A] = REG[A] + (value / 0x10000); break;

case LDF : REG[F] = value; break;

case LDL : REG[L] = value; break;

case LDS : REG[S] = value; break;

case LDT : REG[T] = value; break;

case LDX : REG[X] = value; break;

case LPS : REG[S] = value; break;

case STA : storeMemory(targetAddress,3,REG[A]); break; // store value to target address

case STB : storeMemory(targetAddress,3,REG[B]); break;

case STCH : VMemory[targetAddress] = REG[A] & 0xFF; break; // store lower byte value to target address

case STF : storeMemory(targetAddress,6,REG[F]); break; // store value(6 bytes) to target address

case STI : break;

case STL : storeMemory(targetAddress,3,REG[L]); break;

case STS : storeMemory(targetAddress,3,REG[S]); break;

case STSW : storeMemory(targetAddress,3,REG[SW]); break;

case STT : storeMemory(targetAddress,3,REG[T]); break;

case STX : storeMemory(targetAddress,3,REG[X]); break;

// CONDITIONALS

case COMP : COMPMnemonic(REG[A],value); break; // compare mnemonics and update CC

case COMPF : COMPMnemonic(REG[F],value); break;

case COMPR :COMPMnemonic(REG[R1], REG[R2]); break;

case J : REG[PC] = targetAddress; break; // update pc register to TA value

case JEQ : if(CC=='=') // update pc register to TA if condition is met

REG[PC] = targetAddress;

break;

case JGT : if(CC=='>')

REG[PC] = targetAddress;

break;

case JLT :

if(CC=='<')

REG[PC] = targetAddress;

break;

case JSUB :

REG[L] = REG[PC]; // update L register to PC value

REG[PC] = targetAddress; // update PC register to TA

break;

case CLEAR : REG[R1] = 0; break;

case FIX : REG[A] = REG[F]; break;

case FLOAT : REG[F] = REG[A]; break;

case HIO : break;

case NORM : break;

case AND : REG[A] = REG[A] & value; break;

case OR : REG[A] = REG[A] | value; break;

case RMO : break;

case RSUB : REG[PC] = REG[L]; break;

case SHIFTL : break;

case SIO : break;

case SSK : break;

case SVC : break;

case RD : CC='<'; break;

case TD : CC = '<'; break;

case TIO : REG[X]++; CC = '<'; break; // Device is always ready

case TIX : REG[X]++; CC='<'; break;

case TIXR : REG[X]++;

CC = '=';

if(REG[X] < REG[R1]){

CC = '<';

}else if(REG[X] > REG[R1]){

CC = '>';

}

break;

case WD : break;

}

}

int getFormat(long address){ // return format of the object code

long a = VMemory[address]/0x10;

if(a==0xC || a==0xF) // format 1

return 1;

if(a==9 || a == 0xA || a == 0xB) // format 2

return 2;

if(VMemory[address+1]&0x10) // check e bit

return 4; // format 4 if e bit is set

return 3; // format 3

}

int breakpointExists(long address){ // return 1 if break point exists

Break\_Point \* temp;

temp = bHead;

while(temp){ // loop through list

if(temp->address==address&&temp->visited==0&&last\_address!=address){

temp->visited = 1; // if breakpoint was not visited previously

return 1;

}

temp=temp->next;

}

return 0; // breakpoint not found

}

void printRegisters(long address, int option){ // print register values

printf("\tA : %06lX X : %06lX\n", REG[A], REG[X]);

printf("\tL : %06lX PC: %06lX\n", REG[L], REG[PC]);

printf("\tB : %06lX S : %06lX\n", REG[B], REG[S]);

printf("\tT : %06lX\n", REG[T]);

if(option == 0){ // if it's break point

printf("\tStop at checkpoint[%lX]\n", address);

last\_address = address;

}else{ // if end of program

for(Break\_Point \* temp = bHead; temp!=NULL; temp=temp->next){ // initalize break point

temp->visited = 0; // mark them as unvisited

}

printf("\t\tEnd Program\n");

last\_address = -1;

}

}

int run(){ // run program from progaddr

long address = PROGADDR;

long targetAddress = 0;

long value = 0;

int addressMode = 0;

long endADDR = 0;

int j = 0;

int extended = 0;

REG[L] = PROGADDR;

for(int i = 0 ; i < 8 ; i++) // initialize registers

REG[i] = 0;

for(int i = 0; i < numOfFile; i ++){ // get length of each control section

if(ESTAB[i]!=NULL)

endADDR += ESTAB[i]->length;

}

REG[L] = PROGADDR + endADDR; // update L register

while(address <= PROGADDR + endADDR){ // loop until end of program

long opcode = VMemory[address] & 0xFC; // get opcode using and bit operation

int addressMode = VMemory[address] - opcode;

int format = getFormat(address); // get opcode format

REG[PC] += format; // add PC

if(format==1){

targetAddress = fetchMemory(address,5) & 0x7FFF;

}else if(format==2){

R1 = VMemory[address+1]/0x10; // fetch register 1 and 2 from next byte

R2 = VMemory[address+1]%0x10;

}else if(format >=3){ // format 3 and 4

long xbpe = VMemory[address+1]/0x10; // fetch xbpe value

targetAddress = fetchMemory(address + 1, format == 3 ? 3 : 5); // calculate TA according to format type

if(format == 3 ) { // check PC relative or base relative if format 3

if(targetAddress >= 0xF00){ // check for overflow

targetAddress = -(0xFFF-targetAddress+1);

}

if(xbpe&0x2){ // PC Relative

targetAddress += REG[PC]; // add PC value to TA

}else if(xbpe&0x4){ // b flag is set. Base Relative

targetAddress += REG[B]; // add Base value to TA

}

} // simple address is already calulcated.

switch(addressMode){ // update TA according to address mode

case 0 : targetAddress = fetchMemory(address,5) & 0x7FFF; break; // SIC

case 1 : break; // Immediate address

case 2 : targetAddress = fetchMemory(targetAddress,6); break; // Indirect

case 3 : break; // simple address

}

if(xbpe&0x8){ // Indexed

targetAddress += REG[X];

}

}

execute(opcode,targetAddress,addressMode,format); // execute instruction

address = REG[PC]; // update address to PC register value

if(breakpointExists(address)){ // check if break point exists in the address

printRegisters(address,0);

return SUCCESS;

}

if(address>=PROGADDR+endADDR){ // end of program

printRegisters(address,1);

return SUCCESS;

}

}

printRegisters(address,1); // for handling run when no program is loaded yet

return SUCCESS;

}