**과목명: 시스템프로그래밍**

**1분반**

**<<Project #3>>**

**서강대학교 [경제학과]**

**[20160563]**

**[송진아]**

목 차

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
   2. 프로그램 동작 방식
3. **모듈 정의**
   1. 20160563.h 및 20160563.c의 모듈
   2. loader.h 및 loader.c의 모듈
4. **전역 변수 정의**
   1. loader.c의 전역변수
5. **코드 설명**
   1. 20160563.h 및 20160563.c
   2. loader.h 및 loader.c

**……………… page 3**

**……………… page 4**

**……………… page 5**

**……………… page 7**

**……………… page 9**

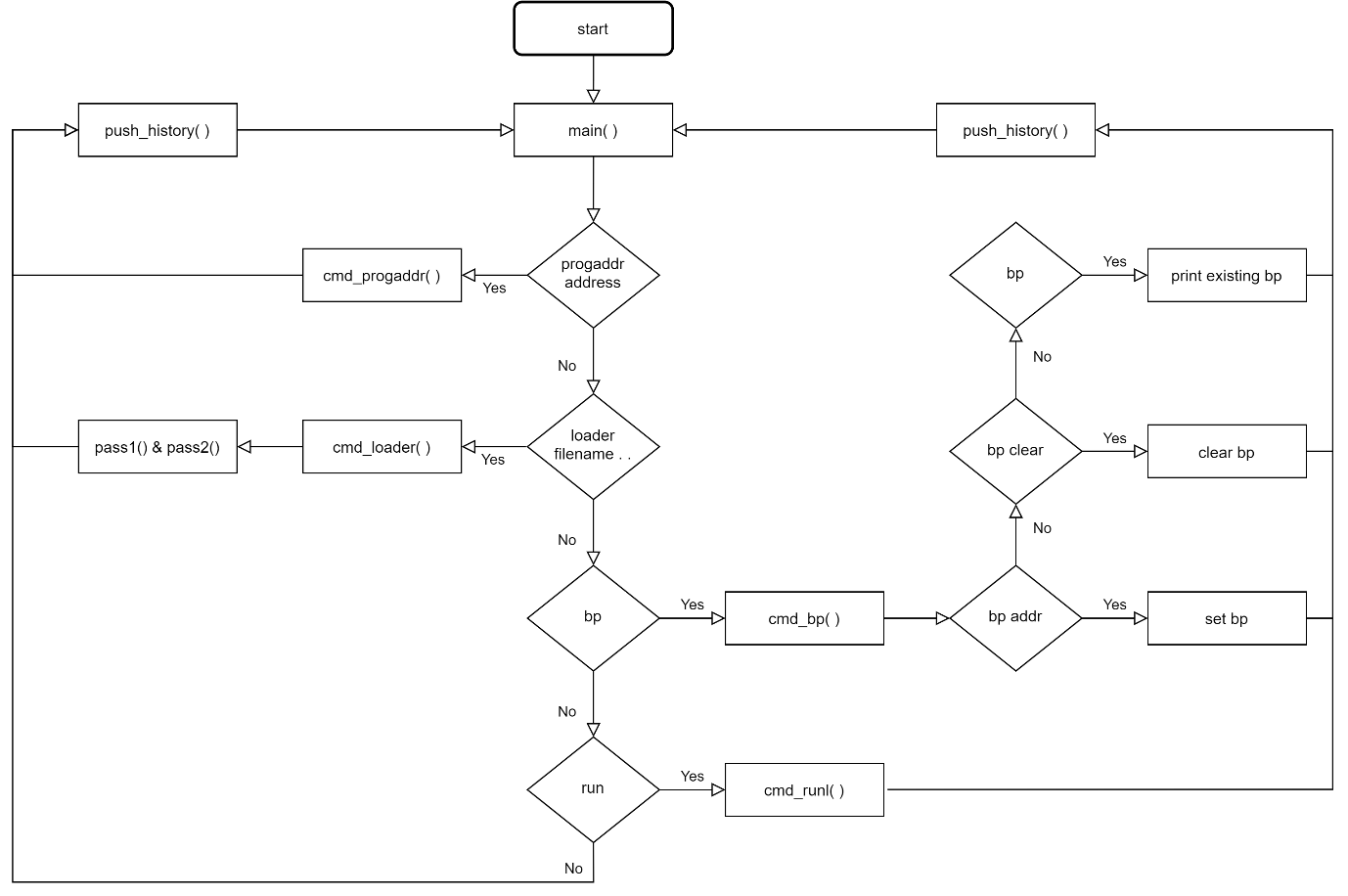
1. **프로그램 개요**

이 프로그램은 SIC/XE 머신을 단계적으로 구현하는 세 개의 프로젝트 중 마지막 프로젝트로서 제작되었으며, SIC/XE 머신의 linking loader와 run에 해당한다. 추가된 명령어는 “progaddr address”, “loader filename1 [filename2, filename3]“, “bp address”, “bp clear”, “bp”, “run” 이다. 아울러, 이전 프로젝트에서 구현한 h[elp] 명령어를 입력하면 이번 프로젝트에서 새롭게 추가된 명령어들이 함께 출력될 수 있도록 수정하였다. 또한 h[istory] 명령어에서도 위 명령어를 입력한 결과를 확인할 수 있도록 하였다.

모든 프로그램은 C언어로 작성되었다 main 함수가 있는 20160563.c 파일과 20160563.h. 파일에서 명령어를 받아 적절한 함수로 연결해준다. “progaddr”, “loader”, “bp”, “run”이 명령어의 첫 부분에 나오는 경우를 추가하였다. loader.c 파일과 loader.h 파일을 추가하여 이번 프로젝트의 명령어들을 처리할 수 있도록 하였다. Makefile은 loader.o를 생성하고 다른 object 파일들과 함께 컴파일 할 수 있도록 수정하였다. 이 외에도 프로그램의 전반적인 설명을 담은 README 파일과 linking loader로 동작시킬 proga.obj, progb.obj, progc.obj, copy.obj 파일 등이 있다.

각 명령어의 기능은 다음과 같다. “progaddr address” 명령어를 입력하면 loader 또는 run 명령어를 수행할 때 시작할 주소를 ‘address’로 설정한다. 이 명령어가 최초로 입력되기 전에 ‘progaddr’은 0으로 지정되어 있다. “loader filename1 [filename2, filename3]“ 명령어를 입력하면 인자로 받은 ‘filename’ 파일들을 link하여 메모리에 load한다. 이것이 정상적으로 실행되는지 확인하기 위한 external symbol들의 load map이 shell에 출력된다. external symbol은 linked list로 구현된 hash table에 저장하였다. 인자로서 들어오는 파일은 하나 이상이어야 하며, 최대 3개까지 받을 수 있다. 존재하지 않는 파일명을 입력하였을 경우 혹은 교재의 Pass1과 Pass2에 명시된 문제들이 발생하는 경우 에러 메시지가 출력된다. “bp address” 명령어를 입력하면 ‘address’ 주소에 breakpoint를 설정한다. 그 다음 break point가 지정되었다는 메시지가 출력된다. “bp clear” 명령어를 입력하면 설정된 모든 break points를 삭제하고 삭제하였다는 메시지를 출력한다. “bp” 명령어를 입력하면 설정되어 있는 모든 break points를 보여준다. “run” 명령어를 실행하면 progaddr에서 설정된 주소부터 메모리를 읽어 프로그램을 실행한다.

1. **프로그램 설명** 
   1. 프로그램 흐름도



* 1. 프로그램 동작 방식

프로그램을 실행시키면 ‘sicsim>’ 프롬프트가 출력된다. 사용자는 프롬프트에 명령을 입력한다. 프로그램은 명령이 유효한지 검사하고 유효하지 않다면 새로운 명령을 받는다. 유효한 명령어에 대해서는 ‘2.2.1 프로그램 흐름도’와 같이 각 명령에 대응되는 함수를 호출하고 성공적으로 수행하였다면 history에 저장한다. 각 명령에 대해 호출되는 함수는 ‘1 프로그램 개요’에서 서술한 바와 같은 기능을 수행한다. 명령이 수행된 후에는 계속해서 새로운 명령을 받고 위의 과정을 반복한다.

1. **모듈 정의**
   1. 20160563.h 및 20160563.c의 모듈

* int main()

입력 프롬프트 ‘sicsim>’ 을 출력하고 사용자로부터 명령을 받는다. 명령이 유효한 명령의 형식을 따르는지 검사하고 유효하다면 각 명령에 해당하는 기능을 수행하는 함수를 호출한다. 또한 유효한 명령이라면 push\_history 함수를 호출하여 history 명령에 추가한다. 명령 “q[uit]”을 받을 때까지 이를 반복한다. 프로젝트2에서 새롭게 구현한 “progaddr”, “loader”, “bp”, “run”에 대한 처리가 추가되었다.

* 1. loader.h 및 loader.c의 모듈

* void cmd\_progaddr(char\* address)  
  int 타입 전역변수 progaddr에 ‘address’ 값을 할당한다. ‘address’는 16진수로 해석한다.
* cmd\_loader(char\* filename1, char\* filename2, char\* filename3)  
  레지스터 값을 0으로 초기화하며 기존의 external symbol table을 삭제한다. 그 다음 ‘filename’들을 load\_pass1( )과 load\_pass2( )의 인자로 넘겨준다.
* void push\_es\_table(int hash, char\* sym, int addr, int length)  
  external symbol의 hash 값을 계산하여 hash table에 저장하는 함수이다. hash 함수는 “opcode.h”에 선언된 것을 사용하였다.
* void delete\_es\_table( )  
  external symbol table을 삭제한다.
* int load\_pass1(char\* filenames[3])  
  교재의 pass1을 구현한다. .obj 파일에서 H record와 D record를 읽어 external symbol table을 생성한다. 파일이 존재하지 않거나 중복되는 symbol을 저장하려고 하면 에러가 발생한다.
* int load\_pass2(char\* filenames[3])  
  교재의 pass2를 구현한다. H record에서는 현재 control section의 주소를 첫번째 reference number로 저장한다. R record에서는 다른 control section의 external symbol을 주어진 refererce number로 저장한다. T record에서는 내용을 읽어 메모리에 load한다. M record에서는 pass1에서 생성된 external symbol table을 통해 해당 reference number에 저장된 symbol의 주소를 찾는다. 그 주소의 메모리 값을 지정된 주소의 메모리 값에 더한다.
* void cmd\_bp(char\* option)  
  “bp address”, “bp clear”, “bp” 명령어를 처리한다. bp는 메모리 크기와 동일한 배열로 선언하였으며 bp가 설정된 주소를 인덱싱하면 1이, 아니면 0이 나온다. bp를 설정하고 나서 runtime에 확인되었다면 -1이다.
* void cmd\_run()  
  copy.obj에 등장한 명령어들을 처리하여 register의 값을 업데이트 하는 등 동작한다. 먼저 format2 명령어들을 처리하였다. 그 다음 format3과 4 명령어들을 처리하는데, n,i 값을 통해 target address를 해석하는 방식에 따라 3가지 종류로 나누었다. Jump Instruction은 ni=10이면 simple addressing, ni=11이면 immediate addressing이다. 이 외에 register 전체(3 bytes)의 값을 r-value로서 할당하지 않는 instruction들은 ni=01이면 immediate addressing, ni=10이면 indirect addressing, ni=11이면 simple addressing이다. 마지막으로 register 전체의 값을 r-value로 사용하는 명령어의 경우, 두 번째 경우와 같이 해석하되 메모리에서 연속된 3개의 bytes를 한 번에 읽어오게 하였다.

1. **전역 변수 정의**
   1. loader.c의 전역변수

* extern unsigned char mem[ ]  
  memory.c에서 선언된 mem[ ] 배열을 가져와 사용하였다. load를 수행하는 pass2와 run에서 사용된다.
* eptr es\_table[HASH\_SIZE]  
  external symbol을 담기 위한 hash table이다. loader의 pass1과 pass2에서 사용된다. 각 hash값에 대하여 linked list로 구현되었다.
* int progaddr  
  프로그램의 시작 주소를 저장한다. 디폴트 값은 0이며 “progaddr” 명령어로 값을 지정할 수 있다.
* int csaddr  
  control section의 시작 주소를 저장한다.
* int cslth  
  .obj 파일을 읽어 해당 control section의 길이를 저장한다.
* int execaddr  
  프로그램이 시작할 주소를 저장한다.
* int prev\_bp  
  이전에 breaking point를 만나 멈추었다면 해당 위치를 저장하여 다음 실행 시 이 곳부터 실행할 수 있도록 한다.
* int bp[MEM\_SIZE]  
  해당 주소에 breaking point가 지정되었는지 여부를 저장하는 배열이다.
* int reg\_val[10]  
  레지스터의 번호에 해당하는 index에 그 레지스터의 값을 저장하는 변수이다.
* typedef enum { A, X, L, B, S, T, F, None, PC, SW } reg\_name  
  reg\_val[ ]에 쉽게 접근하기 위하여 레지스터 이름을 저장하는 enum을 선언하였다.

1. **코드 설명**

각 코드 파일에 포함된 주석을 참고한다.

* 1. 20160563.h 및 20160563.c

#ifndef MAIN

#define MAIN

#include <stdio.h>

#include <string.h>

#include "utility.h"

#include "shell.h"

#include "memory.h"

#include "opcode.h"

#include "assemble.h"

#include "loader.h"

#endif

#include "20160563.h"

int main() {

char cmd[CMD\_SIZE]; // user command

char cmd\_cpy[CMD\_SIZE]; // copied user command

int dump\_cnt = -1; // last printed address

while(1) {

printf("sicsim> ");

fgets(cmd, CMD\_SIZE, stdin); // get user command from stdin

cmd[strlen(cmd)-1] = '\0'; // change newline to null

strcpy(cmd\_cpy, cmd); //copy user command cause strtok change the string

char \*cmd\_ptr = strtok(cmd\_cpy, DELIMITER); // get splitted string by space

if (!cmd\_ptr) continue; // user didn't give any cmd

char \*eos; // end of string : to check whether cmd has appropriate number of arguments

if (!strcmp(cmd\_ptr,"h") || !strcmp(cmd\_ptr,"help")) { // when cmd is h[elp]

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

cmd\_help();

push\_history(cmd); // since cmd is valid, push it to the history

}

else if (!strcmp(cmd\_ptr,"d") || !strcmp(cmd\_ptr,"dir")) { // when cmd is d[ir]

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

cmd\_dir();

push\_history(cmd); // since cmd is valid, push it to the history

}

else if (!strcmp(cmd\_ptr,"q") || !strcmp(cmd\_ptr,"quit")) { // when cmd is q[uit]

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

break; // stop getting user command

}

else if (!strcmp(cmd\_ptr,"hi") || !strcmp(cmd\_ptr,"history")) { // when cmd is hi[story]

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

push\_history(cmd); // since cmd is valid, push it to the history

cmd\_history();

}

else if (!strcmp(cmd\_ptr, "du") || !strcmp(cmd\_ptr, "dump")) { // when cmd is du[mp] [start [end]]

char\* start = strtok(NULL, DELIMITER); // get start addr if exist

char\* end = strtok(NULL, DELIMITER); // get end addr if exist

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

int start\_num = 0, end\_num = 0; // store integer value of address

if (!start) { // when cmd is du[mp]

push\_history(cmd); // since cmd is valid, push it to the history

cmd\_dump(dump\_cnt+1, (dump\_cnt+160 < MAX\_MEM ? dump\_cnt+160 : MAX\_MEM)); // boundary managing

dump\_cnt = dump\_cnt+160 < MAX\_MEM ? dump\_cnt+160 : -1; // boundary managing

continue;

}

if (end && start[strlen(start)-1] != ',') { // error occurred ; there's no ',' between start and end

printf("Invalid Arguments!\n");

continue;

}

start\_num = hexa\_str\_to\_int(start, end ? strlen(start)-2 : strlen(start)-1); // remove ',' from start string if end string exist

if (start\_num > MAX\_MEM || start\_num == -1) { // error occurred ; invalid boundary || invalid hexadecimal number

printf("Invalid Arguments!\n");

continue;

}

if (!end) { // when cmd is du[mp] start

push\_history(cmd); // since cmd is valid, push it to the history

cmd\_dump(start\_num, (start\_num+159 < MAX\_MEM ? start\_num+159 : MAX\_MEM)); // boundary managing

dump\_cnt = start\_num+159 < MAX\_MEM ? start\_num+159 : -1; // boundary managing

continue;

}

end\_num = hexa\_str\_to\_int(end, strlen(end)-1);

if (end\_num > MAX\_MEM || end\_num < start\_num || end\_num == -1) { // error occurred ; invalid boundary || start is bigger then end || invalid hexadecimal number

printf("Invalid Arguments!\n");

continue;

}

else { // when cmd is du[mp] start end

push\_history(cmd); // since cmd is valid, push it to the history

cmd\_dump(start\_num, end\_num);

dump\_cnt = end\_num != MAX\_MEM ? end\_num : -1; // boundary managing

}

}

else if (!strcmp(cmd\_ptr, "e") || !strcmp(cmd\_ptr, "edit")) { // when cmd is e[dit]

char\* address = strtok(NULL, DELIMITER);

char\* value = strtok(NULL, DELIMITER);

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

int address\_num = 0, value\_num = 0;

if (!address || address[strlen(address)-1] != ',' || !value) { // error occurred : missing required argument || no ','

printf("Invalid Arguments!\n");

continue;

}

address\_num = hexa\_str\_to\_int(address, strlen(address)-2);

value\_num = hexa\_str\_to\_int(value, strlen(value)-1);

if (address\_num > MAX\_MEM || address\_num == -1 || value\_num > MAX\_VAL || value\_num == -1) { // error occurred ; invalid boundary || invalid hexadecimal number

printf("Invalid Arguments!\n");

continue;

}

push\_history(cmd); // since cmd is valid, push it to the history

cmd\_edit(address\_num, value\_num);

}

else if (!strcmp(cmd\_ptr, "f") || !strcmp(cmd\_ptr, "fill")) { // when cmd is f[ill]

char\* start = strtok(NULL, DELIMITER);

char\* end = strtok(NULL, DELIMITER);

char\* value = strtok(NULL, DELIMITER);

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

int start\_num = 0, end\_num = 0, value\_num = 0;

if (!start || start[strlen(start)-1] != ',' || !end || end[strlen(end)-1] != ',' || !value) { // error occurred : missing required argument || no ','

printf("Invalid Arguments!\n");

continue;

}

start\_num = hexa\_str\_to\_int(start, strlen(start)-2);

end\_num = hexa\_str\_to\_int(end, strlen(end)-2);

value\_num = hexa\_str\_to\_int(value, strlen(value)-1);

if (start\_num > MAX\_MEM || start\_num == -1

|| end\_num > MAX\_MEM || end\_num == -1

|| end\_num < start\_num

|| value\_num > MAX\_VAL || value\_num == -1) { // error occurred ; invalid boundary || invalid hexadecimal number || start is bigger then end

printf("Invalid Arguments!\n");

continue;

}

push\_history(cmd); // since cmd is valid, push it to the history

cmd\_fill(start\_num, end\_num, value\_num);

}

else if (!strcmp(cmd\_ptr, "reset")) { // when cmd is reset

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

push\_history(cmd); // since cmd is valid, push it to the history

cmd\_reset();

}

else if (!strcmp(cmd\_ptr, "opcode")) { // when cmd is opcode mnemonic

char\* mnemonic = strtok(NULL, DELIMITER);

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

if (cmd\_opcode(mnemonic)) push\_history(cmd); // since cmd is valid, push it to the history

}

else if (!strcmp(cmd\_ptr, "opcodelist")) { // when cmd is opcodelist

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

if (cmd\_opcodelist()) push\_history(cmd); // since cmd is valid, push it to the history

}

else if (!strcmp(cmd\_ptr, "type")) { // when cmd is type filename

char\* filename = strtok(NULL, DELIMITER);

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

if (cmd\_type(filename)) push\_history(cmd); // since cmd is valid, push it to the history

}

else if (!strcmp(cmd\_ptr, "assemble")) { // when cmd is assemble filename

char\* filename = strtok(NULL, DELIMITER);

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

if ((cmd\_assemble(filename))) {

push\_history(cmd); // since cmd is valid, push it to the history

printf("\x1b[32mSuccessfully\x1b[0m %s\n", cmd);

}

}

else if (!strcmp(cmd\_ptr, "symbol")) { // when cmd is symbol

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

cmd\_symbol();

push\_history(cmd);

}

else if (!strcmp(cmd\_ptr, "progaddr")) { // when cmd is progaddr

char\* address = strtok(NULL, DELIMITER);

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

cmd\_progaddr(address);

push\_history(cmd);

}

else if (!strcmp(cmd\_ptr, "loader")) { // when cmd is loader

char\* filename1 = strtok(NULL, DELIMITER);

char\* filename2 = strtok(NULL, DELIMITER);

char\* filename3 = strtok(NULL, DELIMITER);

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

if (cmd\_loader(filename1, filename2, filename3)) {

push\_history(cmd); // since cmd is valid, push it to the history

}

}

else if (!strcmp(cmd\_ptr, "bp")) { // when cmd is bp

char\* option = strtok(NULL, DELIMITER);

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

cmd\_bp(option);

push\_history(cmd);

}

else if (!strcmp(cmd\_ptr, "run")) { // when cmd is run

if ((eos = strtok(NULL, DELIMITER))) { // cmd is invalid

printf("Too Many Arguments!\n");

continue;

}

cmd\_run();

push\_history(cmd); // assume that there's no error

}

else printf("Invalid Command!\n"); // Invalid cmd

}

return 0;

}

* 1. loader.h 및 loader.c

#ifndef LOAD

#define LOAD

#include <stdio.h>

#include <string.h>

#include "utility.h"

#include "opcode.h"

typedef struct \_ES\_NODE\* eptr;

typedef struct \_ES\_NODE { // external symbol table

char sym[SYM\_SIZE+1];

int addr;

int length;

eptr link;

} ES\_NODE;

void cmd\_progaddr(char\*);

int cmd\_loader(char\*, char\*, char\*);

void push\_es\_table(int, char\*, int, int);

void delete\_es\_table();

void loadmap();

int load\_pass1(char\*[3]);

int load\_pass2(char\*[3]);

void cmd\_bp(char\*);

void cmd\_run();

#endif

#include "loader.h"

extern unsigned char mem[]; // external array declared in memory.c

eptr es\_table[HASH\_SIZE]; // hash table for external symbols

int progaddr, csaddr, cslth, execaddr, prev\_bp; // prev\_bp : to restart the program from the previous breakpoint

int bp[MEM\_SIZE]; // bp[i]=1 if breakpoint on i-th address is set

int reg\_val[10]; // to store the value of the registers

typedef enum { A, X, L, B, S, T, F, None, PC, SW } reg\_name; // to make indexing reg\_val easier

void cmd\_progaddr(char\* address) { // when cmd is progaddr

progaddr = hexa\_str\_to\_int(address, strlen(address)-1);

return;

}

int cmd\_loader(char\* filename1, char\* filename2, char\* filename3) { // when cmd is loader

char \*files[3] = {filename1, filename2, filename3}; // array of files

memset(reg\_val, 0, sizeof(reg\_val)); // initialize the values of registers

delete\_es\_table(); // initialize external symbol table

if (!load\_pass1(files)) return 0; // if error occurred from pass1

if (!load\_pass2(files)) return 0; // if error occurred from pass2

return 1; // program succeed

}

void push\_es\_table(int hash, char\* sym, int addr, int length) { // push a node to the hash table

eptr tmp = (eptr)malloc(sizeof(ES\_NODE)); // create new node

strcpy(tmp->sym, sym);

tmp->addr = addr;

tmp->length = length;

tmp->link = NULL;

if (!es\_table[hash]) es\_table[hash] = tmp;

else {

eptr w;

for (w = es\_table[hash]; w->link; w = w->link); // go to the last node

w->link = tmp; // push current node to the end of the linked list

}

return;

}

void delete\_es\_table() { // remove external symbol table

for (int i = 0; i < HASH\_SIZE; i++) {

eptr next;

for (eptr curr = es\_table[i]; curr; curr = next) {

next = curr->link;

free(curr);

}

es\_table[i] = NULL; // initializing

}

return;

}

int load\_pass1(char\* filenames[3]) { // pass1

char line[LINE\_SIZE];

char sym[SYM\_SIZE+1];

char len[SYM\_SIZE+1];

char\* tok;

int hash;

FILE\* fp;

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

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

csaddr = progaddr;

reg\_val[PC] = progaddr; // intialize PC reg for run

for (int file\_num = 0; file\_num < 3 && filenames[file\_num]; file\_num++) { // interate files

if (!(fp = fopen(filenames[file\_num], "r"))) {

printf("File Doesn't Exist!\n"); // error occurred ; no file

return 0;

}

while (fgets(line, LINE\_SIZE, fp)) { // read a line from the file

line[strlen(line)-1] = '\0';

if (line[0] == 'E') break;

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

strncpy(sym, &line[1], SYM\_SIZE); // get program name

tok = strtok(sym, DELIMITER); // remove whitespace

cslth = hexa\_str\_to\_int(&line[SYM\_SIZE\*2+1], SYM\_SIZE-1); // get program size

if ((hash = hash\_function(tok)) == -1) {

printf("Invalid Control Section!\n"); // error occurred ; calculating hash value failed

return 0;

}

for (eptr w = es\_table[hash]; w; w = w->link) { // search

if (!strcmp(w->sym, tok)) { // if found

printf("Duplicated External Symbol!\n"); // error occurred

return 0;

}

}

printf("%-6s %04X %04X\n", tok, csaddr, cslth);

push\_es\_table(hash, tok, csaddr, cslth); // push it to the estab since it's valid

}

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

for (int i = 1; i < strlen(line); i += SYM\_SIZE\*2) { // read one tuple of (symbol name, address) at a time

strncpy(sym, &line[i], SYM\_SIZE); // get symbol name

tok = strtok(sym, DELIMITER); // remove whilespace

strncpy(len, &line[i+SYM\_SIZE], SYM\_SIZE); // get address

if ((hash = hash\_function(tok)) == -1) {

printf("Invalid Control Section!\n"); // error occurred ; calculating hash value failed

return 0;

}

for (eptr w = es\_table[hash]; w; w = w->link) { // search

if (!strcmp(w->sym, tok)) { // if found

printf("Duplicated External Symbol!\n"); // error occurred

return 0;

}

}

printf(" %-6s %04X\n", tok, csaddr+hexa\_str\_to\_int(len, SYM\_SIZE-1));

push\_es\_table(hash, tok, csaddr+hexa\_str\_to\_int(len, SYM\_SIZE-1), -1); // push it to the estab since it's valid

}

}

}

csaddr += cslth;

}

reg\_val[L] = csaddr - progaddr; // L reg for run

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

printf(" total length %04X\n", reg\_val[L]);

return 1;

}

int load\_pass2(char\* filenames[3]) { // pass2

char line[LINE\_SIZE];

char sym[SYM\_SIZE+1];

char addr[SYM\_SIZE+1];

char\* tok;

int len, idx, n, x, y, hash;

FILE\* fp;

csaddr = progaddr;

execaddr = progaddr;

for (int file\_num = 0; file\_num < 3 && filenames[file\_num]; file\_num++) {

char ref\_num[MAX\_REF\_NUM][SYM\_SIZE+1]; // to store reference number of R record and M record

if (!(fp = fopen(filenames[file\_num], "r"))) {

printf("File Doesn't Exist!\n"); // error occurred ; no file

return 0;

}

while (fgets(line, LINE\_SIZE, fp)) {

line[strlen(line)-1] = '\0';

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

execaddr = hexa\_str\_to\_int(&line[1], SYM\_SIZE-1);

break;

}

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

strncpy(sym, &line[1], SYM\_SIZE); // get program name

tok = strtok(sym, DELIMITER); // remove whitespace

strcpy(ref\_num[1], tok); // reference number of current program is 1

cslth = hexa\_str\_to\_int(&line[SYM\_SIZE\*2+1], SYM\_SIZE-1);

}

else if (line[0] == 'R') {

for (int i = 1; i < strlen(line); i += 8) { // read one tuple of (reference number, symbol name) at a time

idx = dec\_str\_to\_int(&line[i], 1); // get reference number

strncpy(sym, &line[i+2], SYM\_SIZE); // get symbol name

tok = strtok(sym, DELIMITER); // remove whilespace

strcpy(ref\_num[idx], tok); // store the info

}

}

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

strncpy(addr, &line[1], SYM\_SIZE); // start address of current record

idx = csaddr + hexa\_str\_to\_int(addr, SYM\_SIZE-1); // add control section addr to the addr

for (int i = SYM\_SIZE+3; i < strlen(line); i += 2)

mem[idx++] = hexa\_str\_to\_int(&line[i], 1); // load the info to the memory

}

else if (line[0] == 'M') { // M record

strncpy(addr, &line[1], SYM\_SIZE); // start address of current record

idx = csaddr + hexa\_str\_to\_int(addr, SYM\_SIZE-1); // add control section addr to the addr

len = hexa\_str\_to\_int(&line[SYM\_SIZE+1], 1); // how many half byte shold be modified

n = dec\_str\_to\_int(&line[SYM\_SIZE+4], 1); // get reference number

if ((hash = hash\_function(ref\_num[n])) == -1) {

printf("Invalid Control Section!\n"); // error occurred ; calculating hash value failed

return 0;

}

y = -1; // to store the addr of external symbol

for (eptr w = es\_table[hash]; w; w = w->link) { // search

if (!strcmp(w->sym, ref\_num[n])) { // if found

y = w->addr;

break;

}

}

if (y == -1){ // not found

printf("Undefined External Symbol!\n"); // error occured

return 0;

}

x = 0; // to store existing value that will be modified

for (int i = 0; i < len/2 + (len%2); i++) { // get the value

x \*= 0x100;

if ((len%2) && !i) x += mem[idx+i] % 0x10;

else x += mem[idx+i];

}

if (line[9] == '+') x += y; // calculate modified value ; add

else x -= y; // calculate modified value ; subtract

unsigned int ux = x; // to represent negative numbers as 2'complement form

for (int i = len/2 - 1 + (len%2); i >= 0; i--) { // load the modified value

if ((len%2) && !i) mem[idx+i] = (mem[idx+i]/0x10) \* 0x10 + ux % 0x10;

else mem[idx+i] = ux % 0x100;

ux /= 0x100;

}

}

}

csaddr += cslth;

}

return 1;

}

void cmd\_bp(char\* option) { // when cmd is bp

if (!option) { // no operand ; print existing break points

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

for (int i = 0; i < MEM\_SIZE; i++)

if (bp[i]) printf("\t%0X\n", i);

}

else if (!strcmp(option, "clear")) { // clear

memset(bp, 0, sizeof(bp));

printf("\t[\x1b[32mok\x1b[0m] clear all breakpoints\n");

}

else { // set

int idx = hexa\_str\_to\_int(option, strlen(option)-1);

bp[idx] = 1;

printf("\t[\x1b[32mok\x1b[0m] create breakpoint %04X\n", idx);

}

}

void cmd\_run() { // when cmd is run

reg\_val[PC] = prev\_bp;

for (int i = prev\_bp; (i < csaddr) && (i == prev\_bp || bp[i] != 1); i = reg\_val[PC]) {

// format2

reg\_val[PC] += 2; // increase program counter

if (mem[i] == 0xB4) { // CLEAR

reg\_val[mem[i+1]/0x10] = 0;

continue;

}

if (mem[i] == 0xA0) { // COMPR

reg\_val[SW] = reg\_val[mem[i+1]/0x10] - reg\_val[mem[i+1]%0x10];

continue;

}

if (mem[i] == 0xB8) { // TIXR

reg\_val[X]++;

reg\_val[SW] = reg\_val[X] - reg\_val[mem[i+1]/0x10];

continue;

}

// format3,4

int target, val = 0, e = (mem[i+1] & 16)/16, ni = mem[i] % 4;

reg\_val[PC] += 1 + e; // inc 3 if format3, inc 4 if format4

if (e) target = (mem[i+1] % 0x10) \* 0x10000 + mem[i+2] \* 0x100 + mem[i+3]; // e ; format4

else target = (mem[i+1] % 0x10) \* 0x100 + mem[i+2]; // e ; format3

if (mem[i+1] & 32) target = reg\_val[PC] + (target & 0x800 ? ((target-1)^0xFFF)\*(-1) : target); // p ; if negative use 2's complement

if (mem[i+1] & 64) target += reg\_val[B]; // b

if (mem[i+1] & 128) target += reg\_val[X]; // x

// J instructions

if (ni == 2) val = mem[target]\*0x10000 + mem[target+1]\*0x100 + mem[target+2]; // simple addressing

else if (ni == 3) val = target; // immediate addressing

if ((mem[i] - ni) == 0x3C) { // J

reg\_val[PC] = val;

continue;

}

else if ((mem[i] - ni) == 0x30) { // JEQ

if (!reg\_val[SW]) {

reg\_val[PC] = val;

continue;

}

}

else if ((mem[i] - ni) == 0x38) { // JLT

if (reg\_val[SW] < 0) {

reg\_val[PC] = val;

continue;

}

}

else if ((mem[i] - ni) == 0x48) { // JSUB

reg\_val[L] = reg\_val[PC];

reg\_val[PC] = val;

continue;

}

// normal instructions

if (ni == 1) val = target; // simple addressing

else if (ni == 2) val = mem[mem[target]]; // indirect addressing

else if (ni == 3) val = mem[target]; // simple addressing

if ((mem[i] - ni) == 0x50) { // LDCH

reg\_val[A] = (reg\_val[A]/0x100)\*0x100;

reg\_val[A] += val; // no overflow?

}

else if ((mem[i] - ni) == 0xD8) { // RD

reg\_val[A] = (reg\_val[A]/0x100)\*0x100;

}

else if ((mem[i] - ni) == 0x4C) { // RSUB

reg\_val[PC] = reg\_val[L];

continue;

}

else if ((mem[i] - ni) == 0x0C) { // STA

mem[target] = reg\_val[A]/0x10000;

mem[target+1] = (reg\_val[A]%0x10000)/0x100;

mem[target+2] = reg\_val[A]%0x100;

}

else if ((mem[i] - ni) == 0x54) { // STCH

mem[target] = reg\_val[A]%0x100;

}

else if ((mem[i] - ni) == 0x14) { // STL

mem[target] = reg\_val[L]/0x10000;

mem[target+1] = (reg\_val[L]%0x10000)/0x100;

mem[target+2] = reg\_val[L]%0x100;

}

else if ((mem[i] - ni) == 0x10) { // STX

mem[target] = reg\_val[X]/0x10000;

mem[target+1] = (reg\_val[X]%0x10000)/0x100;

mem[target+2] = reg\_val[X]%0x100;

}

else if ((mem[i] - ni) == 0xE0) { // TD

reg\_val[SW] = -1;

}

// using 3 bytes value instructions

if (ni == 2) val = mem[mem[target]] \* 0x10000 + mem[mem[target]+1] \* 0x100 + mem[mem[target]+2]; // indirect addressing

else if (ni == 3) val = mem[target] \* 0x10000 + mem[target+1] \* 0x100 + mem[target+2]; // simple addressing

if ((mem[i] - ni) == 0x28) { // COMP

reg\_val[SW] = reg\_val[A] - val;

}

else if ((mem[i] - ni) == 0x00) { // LDA

reg\_val[A] = val;

}

else if ((mem[i] - ni) == 0x68) { // LDB

reg\_val[B] = val;

}

else if ((mem[i] - ni) == 0x74) { //LDT

reg\_val[T] = val;

}

}

printf("A : %06X X : %06X\n", reg\_val[A], reg\_val[X]);

printf("L : %06X PC : %06X\n", reg\_val[L], reg\_val[PC]);

printf("B : %06X S : %06X\n", reg\_val[B], reg\_val[S]);

printf("T : %06X\n", reg\_val[T]);

if (bp[reg\_val[PC]]) {

prev\_bp = reg\_val[PC]; // set prev\_bp

bp[reg\_val[PC]] = -1; // to void eternal break at loop..

printf("\t\tStop at checkpoint[%X]\n", reg\_val[PC]);

}

else {

prev\_bp = progaddr; // reset prev\_bp cause program has done

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

}

}