**과목명: 시스템프로그래밍**

**분반: CSE4100 - 01**

**Project #3**

담당교수: 서강대학교 컴퓨터공학과 박운상

서강대학교

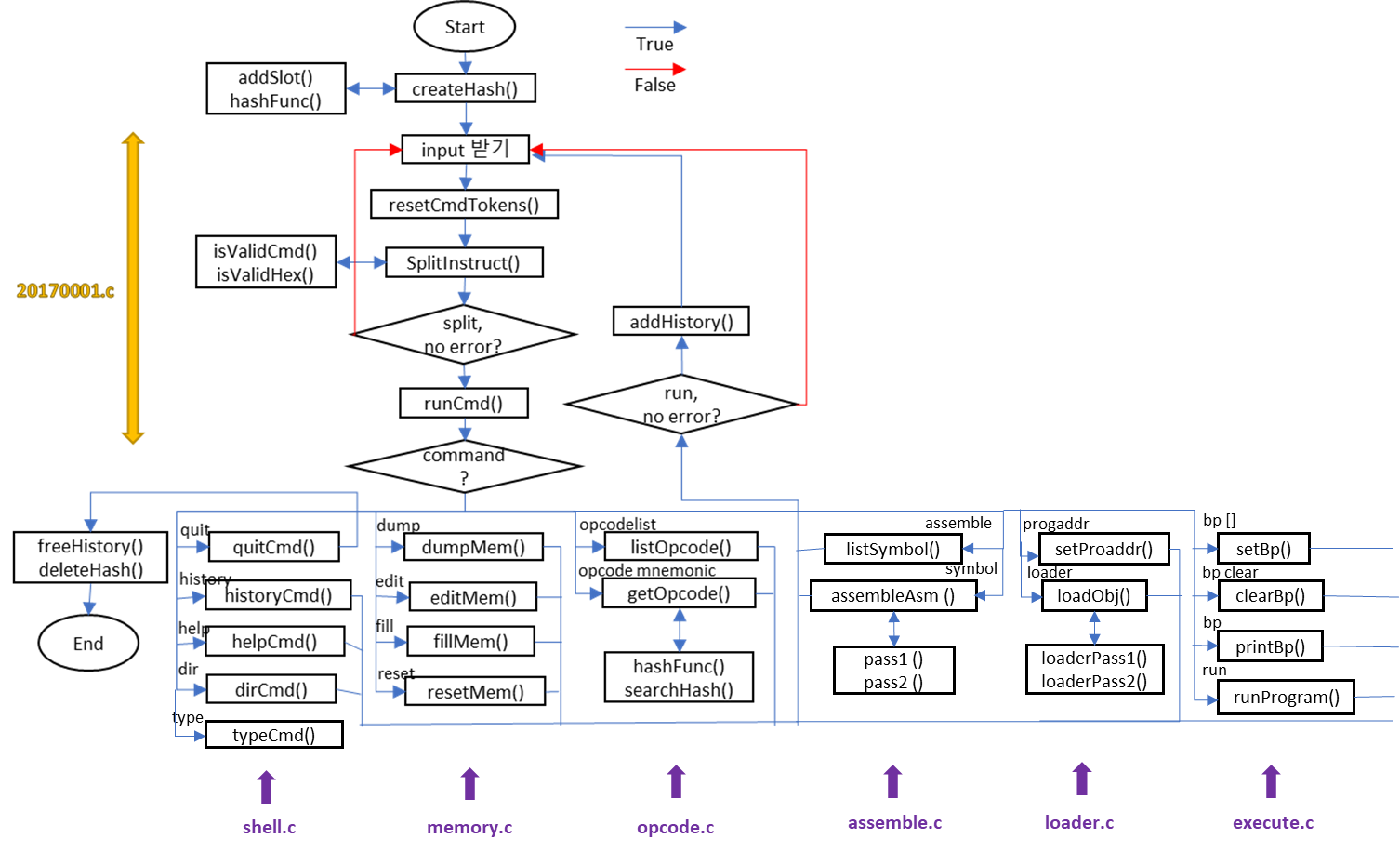
학번: 20170001

이름: 강서현

**목차**

1. **프로그램 개요**
2. **프로그램 설명**
   1. 전체 프로그램 순서도
   2. 기능별 자료구조
   3. 기능별 순서도
3. **모듈 정의**
   1. 주소 지정
   2. Linking loader
   3. Debug
   4. 프로그램 실행
4. **전역 변수 정의**
5. **코드 설명**
6. **프로그램 개요**

SIC/XE 머신의 Linking Loader와 실행 기능을 구현하는 것이 이번 과제의 목표이다. object 파일을 link시켜 메모리에 load하는 기능을 두 개의 pass로 나누어 구현한다. 또한 시작 주소 지정 기능을 추가한다. 그리고 프로그램 실행과 더불어 디버깅을 할 수 있는 기능을 구현한다.

1. **프로그램 설명**
   1. 전체 프로그램 순서도
   2. 기능별 자료구조
      1. 주소 지정 명령어

loader 또는 run 명령어를 수행할 때 시작하는 주소를 지정하는 명령어이다. ProgAddr: 시작 주소를 지정하는 전역 변수.

* + 1. Linking loader

프로그램을 연결하고 로드하는 기능이다.

CsAddr: Control section address를 하는 전역변수이다.

pass1에서 모든 프로그램의 external symbol을 저장하고 pass2에 넘겨주어야 하는데 이 작업을 위해 ExSymTab라는 external symbol table를 만들었다. 이 테이블은 hash table형태로 구현이 되었다.

typedef struct ExSymSlot {//node in external symbol table

char symb[10];//symbol name or name of control section

long addr;//address

long len;//length of control section

int csFlag; //flag for control section

struct ExSymSlot \*next;

} ExSymSlot;

ExSymSlot\* ExSymTab[EXSYMTAB\_LEN]; //external symbol table

* + 1. debug 명령어

중단점을 설정하고, 삭제하고 그 리스트를 보여주는 기능을 하는 명령어이다.

중단점을 저장하고 있어야 하므로 배열 형태의 중단점 리스트 BpList를 만들었다. 또한 이 배열의 길이를 저장하는 전역변수 BpLen를 설정하였다.

* + 1. 프로그램 실행

메모리에 로드된 프로그램을 실제로 실행하는 기능이다.

실행을 하면서 변화하는 레지스터의 값을 저장해야 하므로 레지스터 배열인 Reg[]를 만들고 인덱싱을 쉽게 하기 위해 각 enum을 사용하여 각 레지스터에 번호를 부여하였다.

typedef enum { A, X, L, B, S, T, F, PC = 8, SW } reg;

long Reg[10];

또한 여러 가지의 함수로 instruction의 operand를 쉽게 넘기기 위해 Operand라는 구조체를 만들었다.

typedef struct {

int r1;

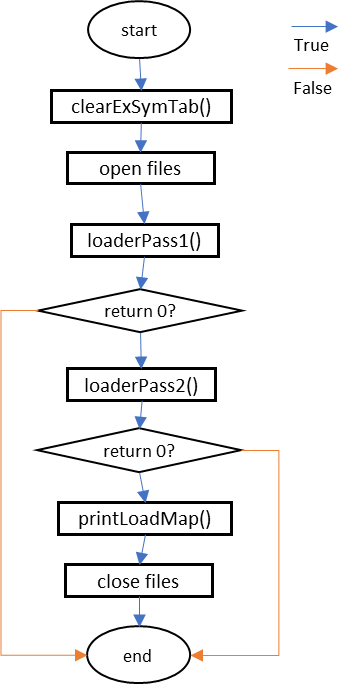
int r2;

long addr;

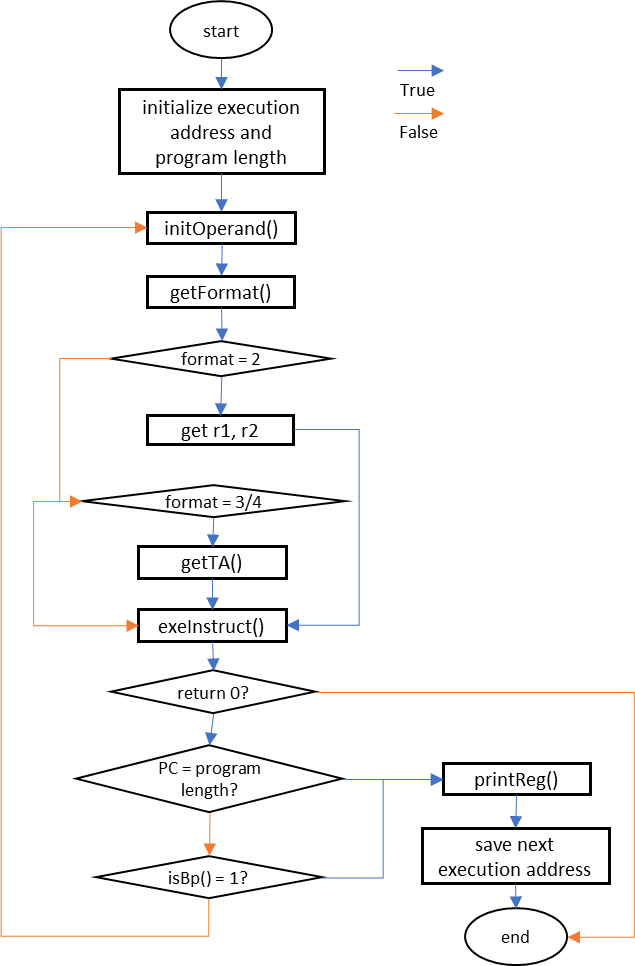
long val;

} Operand;

* 1. 기능별 순서도
     1. Linking loader(loadObj())



* + 1. Execution(runProgram())

1. **모듈 정의**
   1. 주소 지정
      1. int setProaddr(CmdTokens \* tokens)

Linking loader나 run 기능이 시작되는 주소를 설정하는 모듈이다.

<Parameters>

CmdTokens \* tokens: 사용자 입력

<Return Value>

0: 정상종료

-1: 메모리 범위 에러

<Variables>

long addr: 사용자가 입력한 주소를 숫자로 변경한 것 저장

* 1. Linking loader
     1. ExSymSlot \* newExSymSlot(void)

새로운 ExSymSlot을 만들어서 초기화한 후 반환한다.

<Parameters>

없음

<Return Value>

ExSymSlot \*: 새로 생성한 슬롯 반환. 슬롯이 만들어지지 않았다면 NULL반환

<Variables>

ExSymSlot \*newSlot: 새로운 슬롯을 참조하는 포인터

* + 1. int ExSymTabFunc(char \* symb)

symb에 해당하는 hash value를 만들어서 반환한다.

<Parameters>

char \* symb: hash table의 key

<Return Value>

int: hash value를 반환

<Variables>

unsigned long val: hash value 저장

* + 1. void insertExSymTab(ExSymSlot \* newSlot)

슬롯을 ExSymTab에 삽입한다.

<Parameters>

ExSymSlot \*newSlot: ExSymTabl에 새로 삽입하고자 하는 슬롯

<Return Value>

없음

<Variables>

ExSymSlot \*cur, \*prev: 슬롯을 삽입할 위치를 찾기 위해 사용되는 포인터

int index: hash value를 저장

* + 1. ExSymSlot \* getExSym(char \* symb)

ExSymTab에서 symb을 찾아서 반환한다.

<Parameters>

char \* symb: 찾고자 하는 symbol

<Return Value>

ExSymSlot \*≠NULL : 찾은 symbol 슬롯을 반환

=NULL :symbol을 찾지 못한 경우

<Variables>

ExSymSlot \*cur, \*prev: symbol을 찾기 위해 ExStymTab를 traverse할 때 사용되는 포인터

int index: hash value를 저장

* + 1. void clearExSymTab(void)

ExSymTab를 reset한다.

<Parameters>

없음

<Return Value>

없음

<Variables>

ExSymSlot \*cur, \*prev: 모든 슬롯을 traverse하기 위해 사용하는 포인터

* + 1. int loadObj(CmdTokens \* tokens)

사용자가 입력한 파일들을 열고 linking과 loading을 한 후 종료한다.

<Parameters>

CmdTokens \* tokens: 사용자 입력

<Return Value>

=0: 정상종료

≠0: 에러

<Variables>

FILE\* fp[4]: linking과 loading을 할 파일 포인터들의 배열

* + 1. int loaderPass1(FILE \*\* fp)

Linking loader의 pass1 과정을 수행하는 함수. 즉, define record의 심볼을 ExSymTab에 저장하는 과정.

<Parameters>

FILE \*\* fp: linking과 loading을 할 파일 포인터들의 배열

<Return Value>

=0: 정상 종료

≠0: 에러 발생(중복된 심볼)

<Variables>

char line[OBJLINE\_LEN]: object code를 한 줄씩 읽을 때 사용하는 버퍼

long CsLth: control section의 길이

char \*strp: object code에서 읽은 줄을 토큰으로 나눈 것

char addr[10]: 토큰화 된 object code에서 address 부분만 분리하기 위한 변수

int labelIdx: 현재 토큰(strp)에서 label부분이 시작되는 index

ExSymSlot\* symSlot: 새로운 슬롯을 생성하여 ExSymTab에 추가하기 위해 사용되는 포인터

* + 1. int loaderPass2(FILE \*\* fp)

Linking loader의 pass2 과정을 수행하는 함수. object code를 실제로 메모리에 로드하는 과정을 수행하며 modification record에 명시된 대로 object code를 변환한다.

<Parameters>

FILE \*\* fp: linking과 loading을 할 파일 포인터들의 배열

<Return Value>

=0: 정상 종료

≠0: 에러 발생(정의되지 않은 심볼 또는 메모리 범위를 벗어나는 위치에 메모리 로드하는 문제)

<Variables>

char line[OBJLINE\_LEN]: object code를 한 줄씩 읽을 때 사용하는 버퍼

long CsLth: control section의 길이

char \*strp: object code에서 읽은 줄을 토큰으로 나눈 것

long addr: Modification을 할 주소

long val: Modify되는 값

long len: 현재 Text record의 길이

ExSymSlot\* symSlot: ExSymTab에서 참조해야 하는 외부 심볼을 찾아서 그 내용물을 저장

long symList[EXSYM\_NUM]: Refer record에 명시된 외부 심볼의 address를 저장하는 리스트.

* + 1. void printLoadMap(void)

External symbol table의 모든 슬롯을 출력한다.

<Parameters>

없음

<Return Value>

없음

<Variables>

ExSymSlot \*cur: ExSymTab의 모든 슬롯을 traverse하기 위해 사용하는 포인터

* + 1. long toHex(char\* str, int len);

인자로 받은 스트링에서 len 길이 만큼을 hex로 변환하여 반환한다.

<Parameters>

char\* str: 숫자로 변환하고자 하는 스트링

int len: 숫자로 변환하고자 하는 범위(스트링의 길이)

<Return Value>

long: 스트링 값을 hex로 변환한 것을 반환.

<Variables>

int i: for loop에서 사용하는 index 값

int digit: str의 현재 index가 나타내는 자리수

long res: 스트링 값을 hex로 변환한 값

* 1. Debug
     1. int setBp(CmdTokens \* tokens)

사용자가 입력한 address에 breakpoint를 설정한다. 설정이 불가한 경우 -1를 반환한다.

<Parameters>

CmdTokens\* tokens: 사용자 입력

<Return Value>

=0: 정상 종료

≠0: 에러 발생(프로그램 길이를 벗어나는 중단점을 설정하는 경우 또는 아직 로드된 프로그램이 존재하지 않을 경우)

<Variables>

long addr: 사용자가 입력한 주소를 숫자로 변환한 것

* + 1. int isBp(long addr)

addr가 중단점 리스트에 있는지 확인하여 있다면 1을, 없다면 0을 반환한다.

<Parameters>

long addr: 중단점인지 확인하고자 하는 주소

<Return Value>

=0: 중단점이 아님

=1: 중단점 리스트에 존재함.

<Variables>

int i: 리스트의 인덱스

* + 1. void clearBp(void)

모든 중단점을 삭제. 즉, 중단점 리스트의 메모리를 해제한다.

<Parameters>

없음

<Return Value>

없음

<Variables>

없음

* + 1. void printBp(void)

모든 중단점을 출력한다.

<Parameters>

없음

<Return Value>

없음

<Variables>

int i: 중단점 리스트의 인덱스

* 1. 프로그램 실행
     1. int runProgram(void)

사용자가 ‘run’을 입력했을 때 호출되는 함수로, 중단점이나 프로그램 마지막 부분에 도달할 때까지 프로그램을 실행시킨다.

<Parameters>

없음

<Return Value>

=0: 정상 종료

≠0: 에러 발생

<Variables>

int mnem: 현재 instruction의 opcode

int format: 현재 instruction의 instruction format

long cur: 현재 instruction의 address

Operand operand: 현재 instruction의 operand, 즉 format2의 경우 r1과 r2, format3/4의 경우 address나 target value.

* + 1. void initOperand(Operand \* operand);

operand 변수를 0으로 초기화한다.

<Parameters>

Operand \* operand: 초기화하고자 하는 operand

<Return Value>

없음

<Variables>

없음

* + 1. int getFormat(int \* mnem, unsigned char \* mem)

mnem의 format을 반환하는 함수이다.

<Parameters>

int\* mnem: format을 구하고자 하는 대상 mnemonic.

unsigned char\* mem: 현재 instruction이 로드되어 있는 메모리

<Return Value>

=1,2,3,4: mnem에 해당하는 format을 반환

=-1: 에러 발생. mnem가 부적절함.

<Variables>

int flag = 1: 첫 번째 loop에서 format을 찾지 못한 경우 현재 instruction에서 6bit만 mnem로 보고 다시 loop를 실행하기 위한 flag.

* + 1. void getTA(int mnem, unsigned char \* mem, Operand \* op)

현재 instruction의 target address나 target value를 구하는 함수.

<Parameters>

int mnem: 현재 instruction opcode

unsigned char \* mem: 현재 instruction이 로드되어 있는 메모리

Operand \* op: target address나 target value를 저장할 공간

<Return Value>

없음

<Variables>

int ni: instruction 중 ni에 해당하는 2bit 값을 저장

int xbpe: instruction 중 xbpe에 해당하는 4bit 값을 저장

int addr: target address

* + 1. int exeInstruct(int mnem, Operand \* op)

현재 instruction을 실행한다.

<Parameters>

int mnem: 현재 instruction opcode

Operand \* op: 현재 instruction의 target register나 target address, 또는 target value.

<Return Value>

=0: 정상 종료

≠0: 에러 발생(0으로 나눠줄 때)

<Variables>

int r1, r2: 현재 instruction의 target register

int i: for loop을 위한 index

long addr, val: target address, target value.

long m3, m6; target address만을 인자로 받은 경우 그 주소로 가서 3byte 길이의 target value와 6byte 길이의 target value를 구함. 이때 m3가 3byte 길이의 target value, m6 가 6byte 길이의 target value.

* + 1. void storeMem(long val, long addr, int len)

레지스터의 value를 메모리에 저장하는 함수.

<Parameters>

long val: 레지스터의 value

long addr: value를 저장할 처음 위치

int len: value를 저장할 위치의 길이

<Return Value>

없음

<Variables>

없음

* + 1. void printReg(void)

register값을 출력하는 함수.

<Parameters>

없음

<Return Value>

없음

<Variables>

없음

1. **전역변수 정의**
   1. long CsAddr

현재 control section의 주소를 나타냄.

* 1. long ProgAddr

progaddr명령어로 변경할 수 있는 값으로 프로그램 링킹 로딩이나 실행시 시작 주소를 나타냄.

* 1. long ProgLen

로드 된 프로그램의 total length.

* 1. long ExeProgLen

실행중인 프로그램의 total length.

* 1. long ExeAddr

현재 실행중인 메모리 주소값.

* 1. long ExeProgAddr

현재 실행 중에 있는 프로그램의 Progaddr. 사용자가 지정하는 Progaddr와 달리 ‘run’명령어로 프로그램이 시작되는 때에 자동으로 값이 변경됨. 프로그램 실행시 이 값과 progaddr값이 다르면 progaddr값으로 업데이트한 후 실행하게 된다.

* 1. ExSymSlot\* ExSymTab[EXSYMTAB\_LEN]

hash table로 구현한 External symbol table. 2.2.2에서 다룸.

* 1. long\* BpList

breakpoint 배열. 2.2.3에서 다룸.

* 1. int BpLen

breakpoint 배열의 길이. 2.2.3에서 다룸.

* 1. long Reg[10]

레지스터의 값을 담는 배열. 2.2.4에서 다룸.

1. **코드 설명**
   1. i nt setProaddr(CmdTokens \* tokens)

입력받은 스트링을 숫자로 변환하고 메모리 주소 범위를 벗어나지 않는지 확인한다. 문제가 없다면 ProgAddr를 변경한다.

int setProaddr(CmdTokens\* tokens) {

long addr;

addr = strtol(tokens->strpar, NULL, 16);

if (addr < 0 || addr >= 0x100000) {//input bound error

printf("Error: Address Out of Memory Bound. Address range: [0x0, 0xFFFFF]\n");

return -1;

}

ProgAddr = addr;

return 0;

}

* 1. ExSymSlot \* newExSymSlot(void)

새로운 ExSymSlot에 메모리를 할당하고 초기화하여 반환한다.

ExSymSlot\* newExSymSlot(void) {

ExSymSlot \*newSlot = (ExSymSlot\*)malloc(sizeof(ExSymSlot));//allocate memory

//initialize

newSlot->addr = 0;

newSlot->len = 0;

newSlot->csFlag = 0;

newSlot->next = NULL;

return newSlot;

}

* 1. int ExSymTabFunc(char \* symb)

인자로 받은 심볼의 각 character의 값을 바탕으로 hash value를 생성하는 함수이다.

int ExSymTabFunc(char\* symb) {

unsigned long val = 5381;

while (\*symb != '\0') {//visit each character

val = 33 \* val + \*symb;

symb++;

}

return (int)(val % EXSYMTAB\_LEN);

}

* 1. void insertExSymTab(ExSymSlot \* newSlot)

주석에 쓴 것과 같이 해시 값을 구하여 그 해당 인덱스로 이동해 마지막으로 연결되어 있는 노드를 찾은 후 연결한다.

void insertExSymTab(ExSymSlot \*newSlot) {

ExSymSlot \*cur, \*prev;

int index;

index = ExSymTabFunc(newSlot->symb); //get hash value

cur = prev = ExSymTab[index]; //point to the first slot

while (cur) { //get the last slot in the current index of hash table

prev = cur;

cur = cur->next;

}

//link new slot to appropriate place

if (prev)

prev->next = newSlot;

else

ExSymTab[index] = newSlot;

return;

}

* 1. ExSymSlot \* getExSym(char \* symb)

Hash value를 찾은 후 해당 인덱스로 이동하여 연결된 노드를 돌아가면서 심볼을 찾는다.

ExSymSlot\* getExSym(char \* symb) {

ExSymSlot \*cur;

int index;

index = ExSymTabFunc(symb);//get hash value

cur = ExSymTab[index];//first slot of the index

while (cur) {//visit each slot in the same index

if (!strcmp(cur->symb, symb)) {//symbol found in the table

return cur;

}

cur = cur->next;

}

return NULL;//symbol not found in the table

}

* 1. void clearExSymTab(void)

모든 인덱스를 한 번씩 돌아가면서 연결된 노드의 메모리를 해제한다.

void clearExSymTab(void) {

ExSymSlot \*cur, \*next;

for (int i = 0; i < EXSYMTAB\_LEN; i++) {

cur = ExSymTab[i];

while (cur) {

next = cur->next;

free(cur);

cur = next;

}

ExSymTab[i] = 0;

}

return;

}

* 1. int loadObj(CmdTokens \* tokens)

파일 포인터를 저장하는 fp[]를 통해 파일을 열고 닫는다. pass1, pass2에 인자로 fp만 넘겨주어 파일을 읽을 수 있도록 구현했다. pass1과 pass2의 반환 값을 받아서 0이 아닌 경우 프로그램을 종료시키도록 한다. linking과 loading이 끝나면 로드맵을 출력한 후 register값과 실행 주소를 리셋한다.

int loadObj(CmdTokens\* tokens) {

FILE\* fp[4] = { NULL, NULL, NULL, NULL };

//initialize

clearExSymTab();

//open files

if (tokens->param\_num >= 1) {

fp[0] = fopen(tokens->strpar, "r");

}

if (tokens->param\_num >= 2) {

fp[1] = fopen(tokens->strpar1, "r");

}

if (tokens->param\_num >= 3) {

fp[2] = fopen(tokens->strpar2, "r");

}

//run pass1

if (loaderPass1(fp)) {

return -1;

}

//run pass2

if (loaderPass2(fp)) {

return -1;

}

//close files

for (int i = 0; i < tokens->param\_num; i++) {

fclose(fp[i]);

}

printLoadMap();

memset(Reg, 0, 10 \* sizeof(long));//reset registers

Reg[L] = ProgLen;

ExeAddr = ProgAddr;//set execution address

return 0;

}

* 1. int loaderPass1(FILE \*\* fp)

노란색 for loop으로 각 파일을 이동하고 파란색 while loop으로 같은 프로그램 내 섹션들을 돌아가며 pass1을 과정을 반복한다.

회색 부분은 헤더 레코드를 읽고 섹션 이름과 시작 주소, 길이 정보를 얻고 그 내용을 ExSymTab에 넣는 부분이다. 나머지 라인은 분홍색 while loop에서 읽어지는데, 이때 Define record만 주황색 부분에서 프로세싱된다. Define record의 각 심볼을 초록색 while loop으로 하나씩 돌아가며 심볼명과 주소값을 얻어 ExSymTab에 저장한다. 심볼을 추가하기 전에 ExSymTab를 확인하여 이미 존재하는 심볼이라면 프로그램을 종료한다.

int loaderPass1(FILE\*\* fp) {

char line[OBJLINE\_LEN];//buffer to read a line from object code

long CsLth;//length of control section

char \*strp;

char addr[10];

char progName[20];

int labelIdx;

ExSymSlot\* symSlot;

CsAddr = ProgAddr;//initialize address of control section

ProgLen = 0;//initialize total length

clearExSymTab();

for (int i = 0; i < 3 && (\*fp); i++) {//read each file

while (!feof(\*fp)) {//go through multiple control sections in a file

fgets(line, OBJLINE\_LEN, \*fp);//read header line

if (line[strlen(line) - 1] == '\n') {

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

}

if (line[0] == '.')//check if it is comment

continue;

strp = strtok(line, " ");//get control section name

strcpy(progName, strp + 1);//save program name

symSlot = getExSym(strp + 1);//find symbol from ExSymTab

if (symSlot) {//found cs name from ExSymTab

printf("Error from pass1: Duplicate program name '%s'\n", symSlot->symb);

return -1;

}

//not found, create new ExSym slot

symSlot = newExSymSlot();

//set ExSymSlot

strcpy(symSlot->symb, strp + 1);

symSlot->csFlag = 1;

strp = strtok(NULL, " ");//get control section address and length

symSlot->len = strtol(strp + 6, NULL, 16);//save CS length

CsLth = symSlot->len;//remember CS length

strp[6] = '\0';

symSlot->addr = CsAddr + strtol(strp, NULL, 16);//save CS address

insertExSymTab(symSlot);//insert the slot into ExSymTab

while (1) {

fgets(line, OBJLINE\_LEN, \*fp); //read next line

if (line[strlen(line) - 1] == '\n') {

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

}

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

break;

}

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

strp = strtok(line, " ");//get first token

labelIdx = 1;//symbol name starts from index 1

while (strp[6] != '\0' || labelIdx == 1) {//while next symbol exists or it is first symbol

//search ExSymTab for symbol name

symSlot = getExSym(strp + labelIdx);

if (symSlot) {//found

printf("Error from pass1: Duplicate external symbol '%s' in program '%s'\n", symSlot->symb, progName);

return -1;

}

//not found, create new ExSym slot

symSlot = newExSymSlot();

strcpy(symSlot->symb, strp + labelIdx);//save symbol name

//get next token

strp = strtok(NULL, " ");

//extract address

strncpy(addr, strp, 6);

addr[6] = '\0';

symSlot->addr = CsAddr + strtol(addr, NULL, 16);//save address

insertExSymTab(symSlot);//insert into ExSymTab

if (labelIdx == 1) {

labelIdx = 6;//symbol name starts from index 6

}

}

}

}

CsAddr += CsLth;//set starting address for next control section

ProgLen += CsLth;

}

fseek((\*fp), 0, SEEK\_SET);//move file pointer to the front

fp++;

}

return 0;

}

* 1. int loaderPass2(FILE \*\* fp)

loaderPass1과 마찬가지로 노란색 for loop로 각 파일을, 파란 색 while loop으로 각 컨트롤 섹션을 돌아가면서 pass2를 수행한다. 회색 박스는 head record를 읽고 컨트롤 세션의 길이와 주소값을 얻는 부분이다. 나머지 라인은 분홍색 while loop에서 읽힌다. pass1에서는 define record를 프로세싱했지만 pass2에서는 나머지 record를 프로세싱한다. refer record의 심볼은 노란색 박스에서 symList에 저장된다. 이때 명시한 index에 저장이 된다. Text record는 명시된 주소값과 CsAddr를 바탕으로 실제 주소값을 도출한 후 메모리에 object code를 저장한다. Modification record는 명시된 주소를 실제 주소로 변환하고 symList를 참조하여 주소값을 변경한다.

int loaderPass2(FILE\*\* fp) {

char line[OBJLINE\_LEN];//buffer to read a line from object code

long CsLth;//lenth of control section

char \*strp;

long addr, val, len;

char progName[20];

ExSymSlot\* symSlot;

long symList[EXSYM\_NUM];

CsAddr = ProgAddr;

for (int i = 0; i < 3 && (\*fp); i++) {//read each file

while (!feof(\*fp)) {//go through multiple control sections in a file

memset(symList, 0, EXSYM\_NUM \* sizeof(long));//initialize

fgets(line, OBJLINE\_LEN, \*fp);//read header line

if (line[strlen(line) - 1] == '\n') {

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

}

if (line[0] == '.')//check if it is comment

continue;

strp = strtok(line, " ");

strcpy(progName, strp + 1);

symSlot = getExSym(strp + 1);//search ExSymTab for program address

if (!symSlot) {//symbol not found

printf("Error from pass2: Undefined Program '%s'\n", strp + 1);

memset(Mem + ProgAddr, 0, sizeof(Mem[0])\*ProgLen);//unload program

return -1;

}

symList[1] = symSlot->addr;//save address in symbol list

CsLth = symSlot->len;//save control section length

while (1) {

fgets(line, OBJLINE\_LEN, \*fp); //read next line

if (line[strlen(line) - 1] == '\n') {

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

}

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

break;

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

strp = strtok(line + 1, " ");//get first token

while (strp) {

symSlot = getExSym(strp + 2);//search ExSymTab for symbol name

if (!symSlot) {//symbol not found

printf("Error from pass2: Undefined External Symbol '%s' in program '%s'\n", strp + 2, progName);

memset(Mem + ProgAddr, 0, sizeof(Mem[0])\*ProgLen);//unload program

return -1;

}

//save address in the reference number

symList[atoi(strp)] = symSlot->addr;

strp = strtok(NULL, " ");//get next token

}

}

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

strp = line + 9;//get starting point of object code

addr = CsAddr + toHex(line + 1, 6);//get real starting address of the line

len = toHex(line + 7, 2);

for (i = 0; i < len; i++) {//load each byte of object code

if (addr + i >= MEMORY\_SIZE) {//memory bound error

printf("Error from pass2: No more memory can be loaded because the memory address exceeded FFFFF. Address range: [0, FFFFF]\n");

return -1;

}

Mem[addr + i] = (char)toHex(strp + 2 \* i, 2);

}

}

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

addr = CsAddr + toHex(line + 1, 6);//get real starting address of modify code

if (addr < 0 || addr + 2 >= MEMORY\_SIZE) {//memory bound error;

printf("Error from pass2: Modfication code address is out of memory bound. Address range: [0, FFFFF]\n");

return -1;

}

//get the current value in the address

val = Mem[addr] \* 0X10000 + Mem[addr + 1] \* 0X100 + Mem[addr + 2];

if (line[9] == '+') {//add referenced address

val += symList[atoi(line + 10)];

}

else if (line[9] == '-') {//subtract referenced address

val -= symList[atoi(line + 10)];

}

for (i = 2; i >= 0; i--) {//load each byte of new address

Mem[addr + i] = val & 0XFF;

val = val >> 8;

}

}

}

CsAddr += CsLth;//set starting address for next control section

}

fseek((\*fp), 0, SEEK\_SET);//move file pointer to the front

fp++;

}

return 0;

}

* 1. void printLoadMap(void)

모든 인덱스를 한 번씩 돌아가면서 연결된 노드의 심볼명과 주소값을 출력한다. 이때 control section name을 출력할 때는 길이도 함께 출력한다. csFlag를 통해서 control section name인지 symbol인지 구분할 수 있다.

void printLoadMap(void) {

ExSymSlot \*cur;

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

for (int i = 0; i < EXSYMTAB\_LEN; i++) {

cur = ExSymTab[i];

while (cur) {

if (cur->csFlag == 1) {//control section

printf("%-10s %04X %04X\n", cur->symb, (unsigned int)(cur->addr & 0XFFFF), (unsigned int)(cur->len & 0XFFFF));

}

else {//symbol name

printf(" %6s %04X\n", cur->symb, (unsigned int)(cur->addr & 0XFFFF));

}

cur = cur->next;

}

}

printf("-----------------------------------\n total length %04X\n", (unsigned int)(ProgLen & 0XFFFF));

return;

* 1. long toHex(char\* str, int len)]

인자로 받은 스트링의 각 character를 돌아가며 hex숫자로 변환한다. 이때 가장 작은 digit부터 값을 변환해준다. ‘digit’은 현재 캐릭터의 자리수를 의미한다.

long toHex(char\* str, int len) {

int i, digit;

long res = 0;

for (i = len - 1; i >= 0; i--) {//go through each index from the smallest digit

digit = len - i - 1;

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

res += (str[i] - '0')\*(1 << (digit \* 4));

}

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

res += (str[i] - 'A' + 10)\*(1 << (digit \* 4));

}

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

res += (str[i] - 'a' + 10)\*(1 << (digit \* 4));

}

else {

return -1;

}

}

return res;

}

* 1. int setBp(CmdTokens \* tokens)

먼저 입력받은 중단점 주소값이 유효한지를 확인한다. 또한 기존에 존재하는 중단점인지 확인한다. 중단점 배열인 BpList의 자리가 찬 경우 메모리를 재할당한다.

int setBp(CmdTokens\* tokens) {

long addr;

addr = strtol(tokens->strpar, NULL, 16);//get breakpoint address

if (addr < ProgAddr || addr >= ProgAddr + ProgLen || addr < 0 || addr >= MEMORY\_SIZE) {//bp out of boundary

if (ProgLen == 0) {

printf("Error: Unable to set breakpoint. No loaded program on the memory.\n");

}

else {

printf("Error: The breakpoint is out of the program address range."

" Current Program range is [0x%X, 0x%X]\n",

(unsigned int)ProgAddr, (unsigned int)(ProgAddr + ProgLen - 1));

}

return -1;

}

if (isBp(addr)) {

printf(" The breakpoint already exists\n");

return 0;

}

if (BpLen == 0) {//allocate memory

BpList = (long\*)malloc(BPLIST\_LEN \* sizeof(long));

}

else if (sizeof(BpList) / sizeof(long) <= BpLen + 1) {//re allocate memory

BpList = (long\*)realloc(BpList, sizeof(BpList) + BPLIST\_LEN \* sizeof(long));

}

BpList[BpLen] = addr;//save breakbpoint

printf(" [ok] create breakpoint %04X\n", (unsigned int)(BpList[BpLen] & 0XFFFF));

BpLen++;

return 0;

}

* 1. int isBp(long addr)

BpList의 각 인덱스를 돌아가며 인자로 받은 주소 값이 존재하는지 확인한다.

int isBp(long addr) {

int i;

for (i = 0; i < BpLen; i++) {//go through each index

if (BpList[i] == addr) {//find addr in breakpoing list

return 1;

}

}

return 0;

}

* 1. void clearBp(void)

BpList메모리를 해제한다.

void clearBp(void) {

if (BpList == NULL) {

return;

}

free(BpList);

BpList = NULL;

BpLen = 0;

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

return;

}

* 1. void printBp(void)

BpList의 인덱스를 하나씩 돌아가며 중단점을 출력한다

void printBp(void) {

int i;

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

for (i = 0; i < BpLen; i++) {//move through BpList

printf(" %X\n", (unsigned int)BpList[i]);

}

return;

}

* 1. int runProgram(void)

분홍색 while loop을 돌아가면서 instruction을 하나씩 읽는다. 첫 번째 회색 부분은 프로그램을 새로 실행하는 경우 프로그램 길이와 주소값, 레지스터 값을 reset하는 부분이다.

while loop안의 주요 모듈은 format을 얻어 PC에 더하는 부분(노란색) 현재 instruction의 operand를 구하는 부분(초록색), 그리고 instruction을 실행하는 부분(파란색)으로 나뉜다. while loop 안의 회색 박스는 while loop을 벗어나는 base case로, 프로그램의 마지막 부분에 도달하였거나, 중단점에 도달했을 때에 해당하는 부분이다.

int runProgram(void) {

int mnem, format;

long cur;

Operand operand;

if (ExeAddr == ExeProgAddr || ExeProgAddr != ProgAddr) {//execution started

ExeProgLen = ProgLen;

Reg[L] = ProgLen;

ExeProgAddr = ProgAddr;//set ExeAddr as new ProgAddr

ExeAddr = ProgAddr;

}

//initialize registers

Reg[PC] = ExeAddr;

while (1) {

cur = Reg[PC];//save processing address

initOperand(&operand);//initialize operand

mnem = Mem[cur];//get mnemonic

format = getFormat(&mnem, Mem + cur);//get format

Reg[PC] += format;//update program counter

switch (format) {

case(2):

//get r1 and r2 from object code

operand.r1 = Mem[cur + 1];

operand.r2 = operand.r1 % 0X10;

operand.r1 = operand.r1 / 0X10;

break;

case(3):

case(4):

getTA(mnem, Mem + cur, &operand);//get target address

break;

default:

break;

}

//run instruction

if (exeInstruct(mnem, &operand)) {

return -1;

}

if (Reg[PC] == ExeProgLen) {//reached last code in the program

printReg();//print registers

memset(Reg, 0, 10 \* sizeof(long));//reset registers

ExeAddr = ProgAddr;//reset execution address

ExeProgAddr = ProgAddr;

ExeProgLen = ProgLen;

break;

}

if (isBp(Reg[PC])) {//reached breakpoint

printReg();//print registers

ExeAddr = Reg[PC];//save breakpoint as next execution address

break;

}

}

return 0;

}

* 1. void initOperand(Operand \* operand)

매 instruction마다 operand 변수를 reset해야 하는데, 그 기능을 수행한다.

void initOperand(Operand\* operand) {

operand->r1 = 0;

operand->r2 = 0;

operand->addr = 0;

operand->val = 0;

return;

}

* 1. int getFormat(int \* mnem, unsigned char \* mem)

기본적으로 switch문을 사용하여 mnem이 해당하는 case를 찾아 foramt을 반환한다. 그런데 인자로 받은 mnem은 opcode길이를 8bit로 가정하고 구한 것이기 때문에 3/4 format의 경우 mnem이 실제 opcode와 다를 수 있다. 이를 정정하기 위해 첫 번째 switch에서 default가 나온 mnem을 6bit opcode로 변경하여 다시 switch를 실행하게 한다.

int getFormat(int\* mnem, unsigned char\* mem) {

int flag = 1;

for (int i = 0; i < 2 && flag; i++) {

if (flag) {//reset flag

flag = 0;

}

switch (\*mnem) {

mnemonic이 format 1에 해당하는 case

return 1;//format 1

format 2에 해당하는 case

return 2;//format 2

format 3에 해당하는 case

if ((mem[1] & 0X10) == 0X10)//check e bit from xbpe

return 4;

else

return 3;

default://by default menm is 8bit, but it may be 6bit(format3/4)

(\*mnem) = (\*mnem) & 0XFC;//change to 6bit

flag = 1; //go through switch once again with new mnemonic

}

}

return -1;//error

}

* 1. void getTA(int mnem, unsigned char \* mem, Operand \* op)

Target address와 value를 얻기 위해서 먼저 메모리에 저장되어 있는 disp값을 얻는다(노란 박스). 이때 fomat을 고려해야 하며, disp가 음수인 경우 따로 처리해주어야 한다. 그리고 xbp의 값을 바탕으로 초록색 박스 부분에서 addr를 적절히 변경한다. 파란색 박스에서는 ni값으로 addressing mode를 찾아 indirect addressing과 simple addressing의 경우 op->addr에 마지막으로 도출한 주소 값을 넣고, immediate adderssing은 op->val에 value자체를 저장한다.

void getTA(int mnem, unsigned char\* mem, Operand\* op) {

int ni, xbpe, addr;

ni = mem[0] % 4;//get ni

xbpe = mem[1] >> 4;//get xbpe

//get specified address

if ((xbpe & 1) == 1) {//format 4

addr = (mem[1] & 0x0F) \* 0X10000 + mem[2] \* 0X100 + mem[3];

//2's complement

if (addr > 0x7FFFF) {

addr = (0xFFFFF - addr) + 1;

addr = (~addr) + 1;

}

}

else {//format 3

addr = (mem[1] & 0x0F) \* 0X100 + mem[2];

//2's complement

if (addr > 0x7FF) {

addr = (0xFFF - addr) + 1;

addr = (~addr) + 1;

}

}

//pc relative

if ((xbpe & 2) == 2) {

addr += Reg[PC];

}

//base relative

if ((xbpe & 4) == 4) {

addr += Reg[B];

}

//indexing

if ((xbpe & 8) == 8) {

addr += Reg[X];

}

//addressing mode

switch (ni) {

case(1)://immediate addressing

op->val = addr;

break;

case(2)://indirect addressing

if ((xbpe & 1) == 1) {//format 4

op->addr = Mem[addr] \* 0X1000000 + Mem[addr + 1] \* 0X10000 + Mem[addr + 2] \* 0X100 + Mem[addr + 3];

}

else {

op->addr = Mem[addr] \* 0X10000 + Mem[addr + 1] \* 0X100 + Mem[addr + 2];

}

break;

case(3)://simple addressing

op->addr = addr;

break;

default:

break;

}

return;

}

* 1. int exeInstruct(int mnem, Operand \* op)

이 모듈은 크게 target value를 구하는 부분과 실제 instruction을 실행하는 부분으로 나뉜다. 노란색 부분에서는 3byte target value인 m3와 6byte floating point target value인 m6를 모두 구한다. 이 값은 초록색 부분에서 target value가 필요한 경우 사용하게 된다. 일반적으로 integer value를 이용하는 연산인 경우 case ADD처럼 m3를 사용하게 된다. Floating point value를 사용하는 연산일 경우에는 case ADDF처럼 m6를 사용한다. 레지스터간 연산일 경우 ADDR처럼 r1의 값을 얻기 위해 Reg[r1]을, r2의 값을 얻기 위해 Reg[r2]를 사용한다. 비교한 값을 저장하는 것은 case(COMP)에서 알 수 있듯이 SW register에 ‘>’, ‘=’, ‘<’값을 넣는 방식으로 구현하였다.

int exeInstruct(int mnem, Operand\* op) {

int r1, r2, i;

long addr, val, m3, m6;

//subsitution

r1 = op->r1;

r2 = op->r2;

addr = op->addr;

val = op->val;

//calculate target value

if (addr) {

m3 = 0;

//calculate target value(3byte)

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

m3 += Mem[addr + i];

if (i == 2)

break;

m3 = m3 << 8;

}

m6 = m3 << 8;

//calculate floating point value

for (i = 3;; i++) {

m6 += Mem[addr + i];

if (i == 5)

break;

m6 = m6 << 8;

}

}

else if (val) {

m3 = m6 = val;

}

else {

m3 = m6 = 0;

}

//execute instruction

switch (mnem) {

case ADD:

Reg[A] += m3;

break;

case ADDF:

Reg[F] += m6;

break;

case ADDR:

Reg[r2] += Reg[r1];

break;

case(COMP):

if (Reg[A] > m3) {

Reg[SW] = '>';

}

else if (Reg[A] == m3) {

Reg[SW] = '=';

}

else {

Reg[SW] = '<';

}

break;

…생략

default:

break;

}

return 0;

}

* 1. void storeMem(long val, long addr, int len)

인자로 넘겨진 addr 위치부터 val의 값을 len만큼 메모리에 저장한다. 이때 마지막 byte부터 돌아가면서 저장해준다.

void storeMem(long val, long addr, int len) {

//store each byte starting from the last byte

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

Mem[addr + i] = (char)(val % 0X100);

val = val / 0X100;

}

return;

}

* 1. void printReg(void)

각 레지스터를 출력한다.

void printReg(void) {

printf("A : %06X X : %06X\n", (unsigned int)(Reg[A] & 0XFFFFFF), (unsigned int)(Reg[X] & 0XFFFFFF));

printf("L : %06X PC : %06X\n", (unsigned int)(Reg[L] & 0XFFFFFF), (unsigned int)(Reg[PC] & 0XFFFFFF));

printf("B : %06X S : %06X\n", (unsigned int)(Reg[B] & 0XFFFFFF), (unsigned int)(Reg[S] & 0XFFFFFF));

printf("T : %06X \n", (unsigned int)(Reg[T] & 0XFFFFFF));

if (Reg[PC] == ExeProgLen) {

printf(" End Program\n");

}

else {

printf(" Stop at checkpoint[%X]\n", (unsigned int)Reg[PC]);

}

return;

}