**과목명: 시스템프로그래밍**

**분반: 1반(홀수)**

**<<Project #1>>**

**서강대학교 컴퓨터공학과**

**학번 : 20161641**

**이름 : 정성엽**

**목 차**

1. 프로그램 개요

2. 프로그램 설명

2.1 프로그램 흐름도

3. 모듈 정의

3.1 main() 함수

3.2 InstructionProcessing.c 내부함수

3.2.1 void instructionSetInit();

3.2.2 void loadOneInstruction(FILE\* fp, instructionNode \*\*instSet);

3.2.3 void parser(char str[], int\* argv, char argc[100][100], char sep[]);

3.2.4 int classifyInput(int argv, char argc[100][100]);

3.2.5 void storeHistory(char str[]);

3.2.6 int printHistory(int argv, char argc[100][100]);

3.2.7 void recurPrintOneHistory(historyNode\* history);

3.3 functions.c 내부함수

3.3.1 void printError(int errorCase);

3.3.2 int inappropriateInput(int argv, char argc[100][100]);

3.3.3 int help(int argv, char argc[100][100]);

3.3.4 int printDirectory(int argv, char argc[100][100]);

3.3.5 int quitProgram(int argv, char argc[100][100]);

3.4 MemoryInstructions.c 내부함수

3.4.1 int isHex(char str[]);

3.4.2 void printMemoryOneLine(int memoryIdx,int s,int e);

3.4.3 int dumpMemory(int argv, char argc[100][100]);

3.4.4 int editMemory(int argv, char argc[100][100]);

3.4.5 int fillMemory(int argv, char argc[100][100]);

3.4.6 int resetMemory(int argv, char argc[100][100]);

3.5 OpcodeTable.c 내부함수

3.5.1 int returnHash(char str[]);

3.5.2 void makeOpcodeTable();

3.5.3 int opcodeMnemonic(int argv, char argc[100][100]);

3.5.4 int recurFindOpcode(opcodeNode \*node, char str[]);

3.5.5 int opcodeList(int argv, char argc[100][100]);

3.5.6 void recurPrintOneHashTable(opcodeNode \*node);

4. 전역 변수 정의

4.1 구조체 정의

4.1.1 instructionNode

4.1.2 historyNode

4.1.3 opcodeNode

4.2 매크로 정의

4.2.1 MIN / MAX

4.2.2 입력관련 에러 상수 정의

4.2.3 메모리 관련 에러 상수 정의

4.2.4 opcode 관련 에러 상수 정의

4.2.5 MemoryInstructions.c 내부 상수 정의

4.2.6 OpcodeTable.c 내부 상수 정의

4.3 InstructionProcessing.c 전역변수

4.3.1 instructionSet

4.3.2 historySet

4.3.3 int (\*functionPointer[20])(int,char[100][100]);

4.4 MemoryInstructions.c 전역변수

4.4.1 unsigned char memory[65536][16] = {0,};

4.5 OpcodeTable.c 전역변수

4.5.1 opcodeNode\* hashTable[TABLE\_MAX];

5. 코드 설명

5.1 operations.txt 파일 설명

5.2 20161641.h 코드

5.3 20161641.c 코드

5.4 InstructionProcessing.c 코드

5.5 functions.c 코드

5.6 MemoryInstruction.c 코드

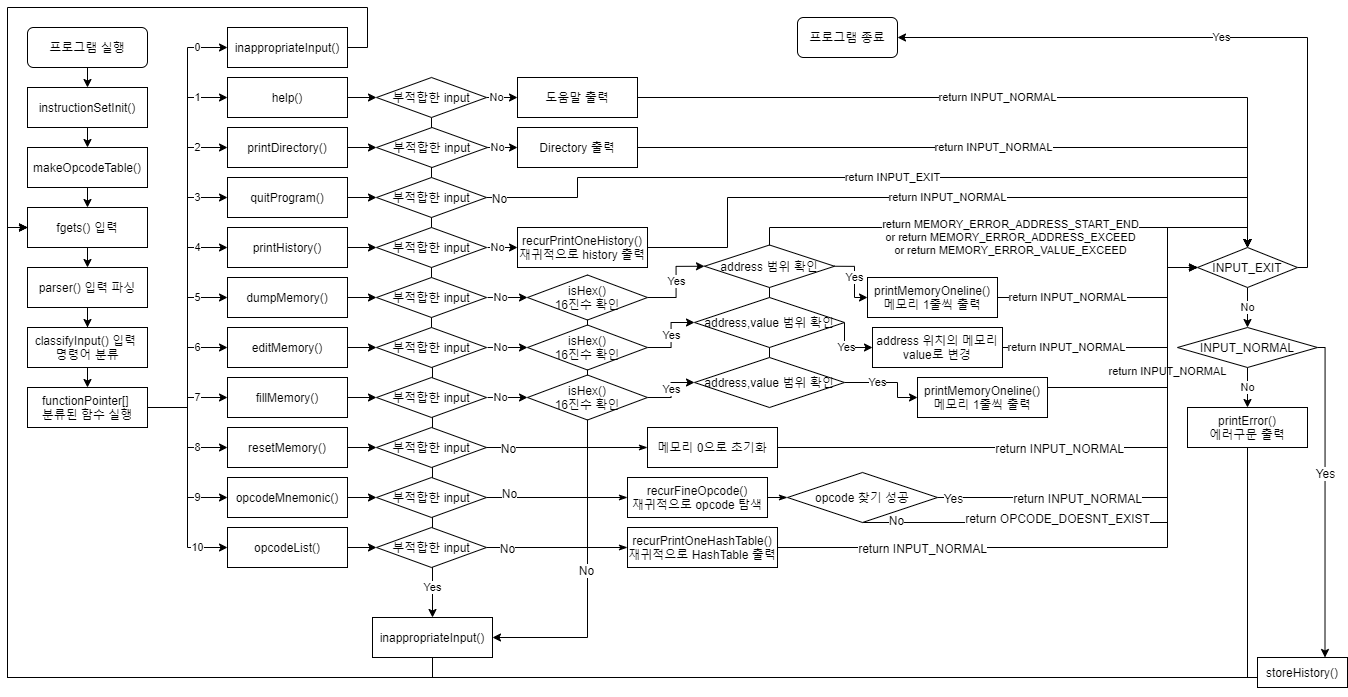
5.7 OpcodeTable.c 코드

1. 프로그램 개요

이 프로그램은 SIC/XE 머신을 구현하기 위한 전 단계로서 어셈블러, 링크 로더들을 실행시킬 Shell과 컴파일을 통해 만들어진 object 코드가 적재 되고 실행될 메모리공간을 구현하고, mnemonic을 opcode 값으로 반환하는 opcode 테이블 및 관련 명령어를 구현한 프로그램입니다.

2. 프로그램 설명

2.1 프로그램 흐름도



3. 모듈 정의

3.1 main() 함수

▶사용 변수

(1) char input[100];

: 들어온 입력을 저장한다.

(2) int argc;

: 파싱 후 분할되어 저장된 인자들의 개수 저장

(3) char argv[100][100];

: 파싱 후 분할된 인자들 저장

(4) char tmpHist[100];

: 파싱 과정에서 input 배열의 변경이 일어남으로 history 저장을 위해 기존 명령어를 저장

▶함수의 동작 및 기능

: 처음 프로그램이 실행되면 instructionSetInit(), makeOpcodeTable() 함수를 호출하여 명령어를 저장하는 instructionSet과 hashTable을 만든다. 이후 while문을 수행하면서 fgets()를 통해 명령어 입력을 받아들인다. Fgets는 '\n'까지 읽기 때문에 이 character를 '\0'으로 변경해준다. 이후 parser() 함수를 통해 명령어를 ", \t"를 기준으로 나누어준다. 나누어진 인자들은 argv[][] 배열에 저장되고 인자들의 개수는 argc 변수에 저장된다. 이후 호출된 classifyInput() 함수는 받은 인자를 이용해 명령어가 어떤 함수를 호출할지 return 받는다. return받은 인자를 이용해 functionPointer 배열에 함수 포인터를 접근하여 해당하는 함수를 호출한다. 각 함수의 수행이 끝나고 함수가 return하는 caseNum을 통해 명령어가 틀렸는지, 종료 명령어가 입력되었는지, 정상적인 명령어라서 history에 저장해야 하는지, ERROR 구문을 출력할지 판별한다. 이후 다시 입력을 기다린다.

3.2 InstructionProcessing.c 내부함수

3.2.1 void instructionSetInit();

▶사용 변수

(1) FILE \*fp;

: operations.txt를 열어 사용하기 위한 FILE pointer

▶함수의 동작 및 기능

: fopen()을 이용해 operations.txt 파일을 열고 loadOneInstruction() 함수를 호출하여 명령어 리스트를 만들도록 한다. 리스트 만들기를 마친 후 fclose()함수를 사용해 파일을 닫아준다.

3.2.2 void loadOneInstruction(FILE\* fp, instructionNode \*\*instSet);

▶사용 변수

(1) FILE \*fp;

: operations.txt를 열어 사용하기 위한 FILE pointer

(2) instructionNode \*\*instSet

: linked list를 저장할 리스트의 시작노드. 함수를 재귀적으로 실행하면서 다음 노드를 넘겨준다.

(3) char str[100];

: operations.txt파일을 읽기 위한 char 배열

▶함수의 동작 및 기능

: 이 함수는 명령어 linked list를 만들기 위해 재귀적으로 명령을 수행한다. linked list의 시작 노드 instructionSet (함수내부에서는 instSet으로 사용)은 비어있는 상태이다. 재귀적으로 파일의 끝까지 읽으면 거꾸로 저장을 시작한다. 마지막 문장부터 저장을 한 후 재귀함수가 종료되면 뒤에서 두 번째 문장을 저장하고 이를 반복한다.

3.2.3 void parser(char str[], int\* argv, char argc[100][100], char sep[]);

▶사용 변수

(1) char str[];

: 파싱을 진행할 문자열

(2) int\* argv;

: 파싱 이후 분할된 인자들의 개수를 저장한다.

(3) char argc[100][100];

: 파싱 이후 분할된 인자들을 저장한다.

(4) char sep[];

: 문자열을 자르는 기준 character들을 입력으로 받아들인다.

(5) char\* token;

:strtok() 함수에서 return 하는 char pointer를 저장하기 위한 변수

▶함수의 동작 및 기능

: 이 함수는 string.h 헤더파일에 선언된 strtok() 함수를 사용한다. strtok() 함수는 인자로 문자열 str과 자르는 기준이 되는 char sep[] 배열를 받아 잘린 인자 1개를 return 한다.

3.2.4 int classifyInput(int argv, char argc[100][100]);

▶사용 변수

(1) int\* argv;

: 파싱된 명령어 인자들의 개수를 저장한다.

(2) char argc[100][100];

: 파싱된 명령어 인자들을 저장한다.

▶함수의 동작 및 기능

: argv 가 0이라면 틀린 명령어임으로 INPUT\_ERROR를 return한다. argc[0]에 저장된 문자열과 instructionSet의 str에 저장된 명령어를 strcmp()를 사용해 비교한다. 만약 두 문자열이 같다면 instructionSet의 caseNum을 return 한다. 모든 리스트를 순회하여도 같은 문자열을 찾을 수 없다면 실행 불가능한 잘못된 명령어가 입력된 것 임으로 INPUT\_ERROR를 return 한다.

3.2.5 void storeHistory(char str[]);

▶사용 변수

(1) char str[];

: 입력으로 들어온 명령어 문장이 저장되어 있다.

(2) historyNode\* newNode;

: historySet에 추가될 새로운 노드이다. 명령어 문자열과 명령어가 몇 번째인지 저장한다.

▶함수의 동작 및 기능

: newNode에 malloc()을 이용 새 historyNode를 할당하고 strcpy()를 이용해 명령어를 저장한다. historySet이 비었는지 확인하고 비었다면 첫 count를 1로 이전 history가 있다면 그 history의 count+1을 저장한다. 이후 historyset에 newNode를 저장한다.

3.2.6 int printHistory(int argv, char argc[100][100]);

▶사용 변수

(1) int\* argv;

: 파싱된 명령어 인자들의 개수를 저장한다.

(2) char argc[100][100];

: 파싱된 명령어 인자들을 저장한다.

▶함수의 동작 및 기능

: 먼저 argc가 1이 아니라면 명령어가 잘못 입력된 것 임으로 INPUT\_ERROR를 return 하고, 아니라면 recurPrintOneHistory() 함수를 호출하여 history를 출력한다. History 저장하는 구조상 printHistory()를 호출하는 명령어는 저장되지 않았기 때문에 이 명령어를 마지막에 출력해준다. 함수가 정상적으로 종료되면 INPUT\_NORMAL을 return한다.

3.2.7 void recurPrintOneHistory(historyNode\* history);

▶사용 변수

(1) historyNode\* history;

: historyNode에 history 명령어와 count가 저장되어있다.

▶함수의 동작 및 기능

: 재귀적으로 histoeySet의 historyNode를 하나씩 탐색하여 count와 str을 출력한다.

3.3 functions.c 내부함수

3.3.1 void printError(int errorCase);

▶사용 변수

(1) int errorCase;

: 에러 종류를 저장하는 변수

▶함수의 동작 및 기능

: 에러의 종류별로 해당하는 에러구문을 출력한다. #define을 이용해 각 에러의 종류를 상수로 지정하였고, 이를 이용한다. 구현된 에러의 종류는 다음과 같다. MEMORY\_ERROR\_VALUE\_EXCEED, MEMORY\_ERROR\_ADDRESS\_EXCEED, MEMORY\_ERROR\_ADDRESS\_START\_END, OPCODE\_DOESNT\_EXIST. 각 에러의 종류는 아래 "4.2 매크로 정의"에 나와있다.

3.3.2 int inappropriateInput(int argv, char argc[100][100]);

▶사용 변수

(1) int\* argv;

: 파싱된 명령어 인자들의 개수를 저장한다.

(2) char argc[100][100];

: 파싱된 명령어 인자들을 저장한다.

▶함수의 동작 및 기능

: 이 함수는 적합하지 않은 명령어가 입력되었을 때 실행된다. 잘못 입력되었다는 경고 문구를 출력해주고 다시 함수가 호출되는 것을 막기 위해 ERROR\_CHECKED를 return 한다.

3.3.3 int help(int argv, char argc[100][100]);

▶사용 변수

(1) int\* argv;

: 파싱된 명령어 인자들의 개수를 저장한다.

(2) char argc[100][100];

: 파싱된 명령어 인자들을 저장한다.

▶함수의 동작 및 기능

: 만약 argc가 1이 아니라면 잘못된 명령어가 입력된 것임으로 INPUT\_ERROR를 return한다. 아니라면 각 줄에 어떤 명령어가 있는지 출력한다. 함수가 정상적으로 종료되면 INPUT\_NORMAL을 return한다.

3.3.4 int printDirectory(int argv, char argc[100][100]);

▶사용 변수

(1) int\* argv;

: 파싱된 명령어 인자들의 개수를 저장한다.

(2) char argc[100][100];

: 파싱된 명령어 인자들을 저장한다.

(3) DIR\* dir;

: opendir()함수를 이용해 현재 directory를 열고 그 포인터를 저장한다.

(4) struct dirent\* entry;

: readdir()함수를 이용해 읽어 들인 파일(혹은 directory) 이름을 저장한다.

(5) struct stat buf;

: 파일(혹은 directory) 정보를 저장한다.

(6) int count;

: 한 줄에 3개씩 출력하기 위해 몇 개를 출력했는지 저장한다.

(7) char str[35];

: 파일(혹은 directory) 이름에 문자열을 더하는 연산을 위해 이름을 저장한다.

▶함수의 동작 및 기능

: opendir()함수를 이용해 현재 directory를 열고, readdir()함수를 이용해 파일(혹은 directory)정보를 읽어 들인다. 만약 이름이 매우 긴 경우(29자 이상) "…"을 이용해 뒤쪽을 생략한다. S\_ISDIR(), S\_ISREG(), S\_IEXEC 매크로를 사용하여 일반 파일인지 실행파일인지, directory인지 판별하고 각 경우마다 정해진 양식으로 출력한다. 함수가 정상적으로 종료되면 INPUT\_NORMAL을 return한다.

<추가구현> directory인 경우 파란색으로 출력하고, 실행파일인 경우 초록색으로 출력한다.

3.3.5 int quitProgram(int argv, char argc[100][100]);

▶사용 변수

(1) int\* argv;

: 파싱된 명령어 인자들의 개수를 저장한다.

(2) char argc[100][100];

: 파싱된 명령어 인자들을 저장한다.

▶함수의 동작 및 기능

: 만약 argc가 1이 아니라면 잘못된 명령어가 입력된 것임으로 INPUT\_ERROR를 return한다. 아니라면 프로그램 종료를 위해 INPUT\_EXIT을 return한다.

3.4 MemoryInstructions.c 내부함수

3.4.1 int isHex(char str[]);

▶사용 변수

(1) char str[];

: 16진수인지 검사할 문자열이 주어진다.

▶함수의 동작 및 기능

: 입력으로 주어지는 문자열의 각 character가 16진수의 인자가 될 수 있는지 판별한다. (0~F) 16진수의 값이 아닌 경우 0을 return 하고 16진수가 될 수 있는경우 1을 return 한다.

3.4.2 void printMemoryOneLine(int memoryIdx,int s,int e);

▶사용 변수

(1) int memoryIdx;

: 몇 번째 주소의 메모리인지 메모리의 index를 저장하는 변수

(2) int s;

: 해당 메모리에서 0~15까지 범위 중 어디서 출력을 시작하는지 저장하는 변수

(3) int e;

: 해당 메모리에서 0~15까지 범위 중 어디서 출력을 끝내는지 저장하는 변수

▶함수의 동작 및 기능

: memory[memoryidx][s] 부터 memory[memosyIdx][e]까지 메모리의 인자들을 출력 양식에 맞추어 출력한다. character로 출력할 때 메모리의 값이 [VALUE\_MIN, VALUE\_MAX] 범위에 들어오지 않으면 '.'을 출력한다.

3.4.3 int dumpMemory(int argv, char argc[100][100]);

▶사용 변수

(1) int\* argv;

: 파싱된 명령어 인자들의 개수를 저장한다.

(2) char argc[100][100];

: 파싱된 명령어 인자들을 저장한다.

(3) static int prevIndex;

: static 변수를 사용해 마지막 실행주소를 계속 저장하는 변수

(4) int index[2];

: 문자열로 입력된 주소를 strtol() 함수를 이용해 변환한 후 integer로 address의 start와 end를 저장한다.

▶함수의 동작 및 기능

: argv가 3초과이거나, argc안의 address인자들을 isHex()함수를 사용해 검사하여 16진수가 아닌 경우 잘못된 명령어가 입력된 것임으로 INPUT\_ERROR를 return한다. strtol()함수를 사용해 문자열을 integer로 변환한 후 이 address 값이 INDEX\_MAX를 초과하는지 확인한다. 만약 범위를 초과한다면 MEMORY\_ERROR\_ADDRESS\_EXCEED를 return한다. 그리고 index[0]>index[1] 이라면 start address가 end address보다 크다는 뜻이고 MEMORY\_ERROR\_ADDRESS\_START\_END를 return한다. Index 예외처리가 완료된 후, 입력받은 address를 address/16을 통해 몇 번째 행인지 주소를 구하고 address%16을 통해 몇 번째 열까지 인지 주소를 구한다. 이후 printMemoryOneLine()함수를 호출하여 각 메모리를 한 줄씩 출력한다. 이후 함수가 정상적으로 종료되면 INPUT\_NORMAL을 return한다.

3.4.4 int editMemory(int argv, char argc[100][100]);

▶사용 변수

(1) int\* argv;

: 파싱된 명령어 인자들의 개수를 저장한다.

(2) char argc[100][100];

: 파싱된 명령어 인자들을 저장한다.

(3) int address;

: 문자열로 입력된 주소를 strtol() 함수를 이용해 변환한 후 integer로 address를 저장하는 변수

(4) int value;

: 문자열로 입력된 값을 strtol() 함수를 이용해 변환한 후 integer로 value를 저장하는 변수

▶함수의 동작 및 기능

: argv가 3이 아니거나, argc안의 address, value인자를 isHex()함수를 사용해 검사하여 16진수가 아닌 경우 잘못된 명령어가 입력된 것임으로 INPUT\_ERROR를 return한다. strtol()함수를 사용해 문자열을 integer로 변환한 후 address 값이 INDEX\_MAX를 초과하는지 확인한다. 만약 범위를 초과한다면 MEMORY\_ERROR\_ADDRESS\_EXCEED를 return한다. 그리고 value가 [VALUE\_MIN,VALUE\_MAX] 범위 안에 들어오는지 확인한다. 만약 범위를 초과한다면 MEMORY\_ERROR\_VALUE\_EXCEED를 return한다. Address, value 예외처리가 완료된 후, memory[address/16][address%16]을 value로 변경한다. 함수가 정상적으로 종료되면 INPUT\_NORMAL을 return한다.

3.4.5 int fillMemory(int argv, char argc[100][100]);

▶사용 변수

(1) int\* argv;

: 파싱된 명령어 인자들의 개수를 저장한다.

(2) char argc[100][100];

: 파싱된 명령어 인자들을 저장한다.

(3) int index[2];

: 문자열로 입력된 주소를 strtol() 함수를 이용해 변환한 후 integer로 address의 start와 end를 저장한다.

(4) int value;

: 문자열로 입력된 값을 strtol() 함수를 이용해 변환한 후 integer로 value를 저장하는 변수

▶함수의 동작 및 기능

: argv가 4가 아니거나, argc안의 인자들을 isHex()함수를 사용해 검사하여 16진수가 아닌 경우 잘못된 명령어가 입력된 것임으로 INPUT\_ERROR를 return한다. strtol()함수를 사용해 문자열을 integer로 변환한 후 이 address 값이 INDEX\_MAX를 초과하는지 확인한다. 만약 범위를 초과한다면 MEMORY\_ERROR\_ADDRESS\_EXCEED를 return한다. 그리고 index[0]>index[1] 이라면 start address가 end address보다 크다는 뜻이고 MEMORY\_ERROR\_ADDRESS\_START\_END를 return한다. 그리고 value가 [VALUE\_MIN,VALUE\_MAX] 범위 안에 들어오는지 확인한다. 만약 범위를 초과한다면 MEMORY\_ERROR\_VALUE\_EXCEED를 return한다. 모든 예외처리가 완료된 후, 반복문을 사용해 index[0]부터 index[1]까지 범위에 해당하는 mamory를 value로 변경한다. 함수가 정상적으로 종료되면 INPUT\_NORMAL을 return한다.

3.4.6 int resetMemory(int argv, char argc[100][100]);

▶사용 변수

(1) int\* argv;

: 파싱된 명령어 인자들의 개수를 저장한다.

(2) char argc[100][100];

: 파싱된 명령어 인자들을 저장한다.

▶함수의 동작 및 기능

: argv가 1이 아니라면 잘못된 명령어가 입력된 것임으로 INPUT\_ERROR를 return한다. 이후 0부터 INDEX\_MAX까지 반복문을 이용해 순회하면서 memory의 값을 0으로 초기화한다. 함수가 정상적으로 종료되면 INPUT\_NORMAL을 return한다.

3.5 OpcodeTable.c 내부함수

3.5.1 int returnHash(char str[]);

▶사용 변수

(1) char str[];

Hashcode를 만들 문자열을 저장하는 변수

(2) int hash;

Hashcode를 만드는 과정에서 사용할 소수를 저장

▶함수의 동작 및 기능

: 문자열에 character마다 hash변수를 shift연산을 수행한 값에 더하고 TABLE\_MAX로 modulo연산을 수행한다. 결과로 문자열에 대응하는 hash값을 return 한다.

3.5.2 void makeOpcodeTable();

▶사용 변수

(1) FILE \*fp;

: opcode.txt를 열어 사용하기 위한 FILE pointer

(2) int argc;

: 파싱된 명령어 인자들의 개수를 저장한다.

(3) int hashNum;

: returnHash()를 통해 전달받은 hash값을 저장할 변수

(4) char str[];

: opcode.txt에서 1줄씩 읽어 들여 저장할 배열

(5) char argv[100][100];

: 파싱된 명령어 인자들을 저장한다.

▶함수의 동작 및 기능

fopen()을 이용해 opcode.txt 파일을 열고 fgets() 함수를 사용하여 str에 한 줄을 입력받고, parser() 함수를 이용해 문자열을 파싱해준다. 각 인자들을 newNode에 저장하고 returnHash()를 통해 전달받은 hash값을 이용해 newNode를 hashTable[hashNum]에 linked list 형태로 저장한다. 이를 통해 opcode hash table을 만든다. 만들기를 마친 후 fclose()함수를 사용해 파일을 닫아준다.

3.5.3 int opcodeMnemonic(int argv, char argc[100][100]);

▶사용 변수

(1) int\* argv;

: 파싱된 명령어 인자들의 개수를 저장한다.

(2) char argc[100][100];

: 파싱된 명령어 인자들을 저장한다.

(3) int tmp;

: recurFindOpcode() 함수에서 반환한 opcode를 저장한다

▶함수의 동작 및 기능

argv가 2가 아니라면 잘못된 명령어가 입력된 것임으로 INPUT\_ERROR를 return한다. recurFindOpcode() 함수를 사용해 각 hashTable[i]를 순회하면서 문자열에 해당하는 opcode를 찾는다. 만약 opcode를 찾는다면 (tmp!=-1) 결과를 출력하고 INPUT\_NORMAL을 return한다. 만약 모든 hashTable을 탐색했지만 찾지 못한 경우, OPCODE\_DOESNT\_EXIST를 return 한다.

3.5.4 int recurFindOpcode(opcodeNode \*node, char str[]);

▶사용 변수

(1) opcodeNode \*node;

: 함수에서 문자열을 확인할 node를 가리키는 pointer

(2) char str[];

: 찾을 문자열이 저장된 배열

▶함수의 동작 및 기능

: 재귀적으로 (node==NULL)에 도달할 때까지 str의 문자열과 일치하는 node->str이 있는지 strcmp()를 사용해 탐색한다. 만약 일치하는 node를 잦는다면 node에 저장된 opcode를 return 한다. 만약 찾지 못한 경우 -1을 return한다.

3.5.5 int opcodeList(int argv, char argc[100][100]);

▶사용 변수

(1) int\* argv;

: 파싱된 명령어 인자들의 개수를 저장한다.

(2) char argc[100][100];

: 파싱된 명령어 인자들을 저장한다.

▶함수의 동작 및 기능

: argv가 1이 아니라면 잘못된 명령어가 입력된 것임으로 INPUT\_ERROR를 return한다. 각 index마다 recurPrintOneHashTable()을 호출하여 opcode hash table의 내용을 출력한다. 함수가 정상적으로 종료되면 INPUT\_NORMAL을 return한다.

3.5.6 void recurPrintOneHashTable(opcodeNode \*node);

▶사용 변수

(1) opcodeNode \*node;

: 함수에서 출력할 내용이 저장된 opcodeNode를 가리키는 pointer

▶함수의 동작 및 기능

: (node==NULL)이 아닐 때 까지 재귀적으로 node->link를 따라 hashTable의 mnemonic과 opcode를 출력한다.

4. 전역 변수 정의

4.1 구조체 정의

4.1.1 instructionNode

▶원소

(1) char str[100];

: instruction 문자열을 저장

(2) int caseNum;

: 각 명령어별로 실행할 함수가 저장된 함수 포인터 배열 functionPointer[]의 각 함수에 해당하는 index를 저장

(3) struct \_instructionNode\* link;

: linkedlist 구조를 위해 다음 노드를 가리키는 pointer

▶역할

: instruction을 저장하는 linked list의 노드

4.1.2 historyNode

▶원소

(1) int count;

: 몇 번째로 입력받은 명령어인지 저장

(2) char str[100];

: 입력받은 명령어 문자열을 저장

(3) struct \_historyNode\* link;

: linkedlist 구조를 위해 다음 노드를 가리키는 pointer

▶역할

: history을 저장하는 linked list의 노드

4.1.3 opcodeNode

▶원소

(1) int opcode;

: 해당 mnemonic의 opcode저장

(2) char str[100];

: mnemonic문자열을 저장

(3) int val[2];

: mnemonic이 몇 번mode로 동작하는지 저장

(4) struct \_opcodeNode\* link;

: linkedlist 구조를 위해 다음 노드를 가리키는 pointer

▶역할

: history을 저장하는 linked list의 노드

4.2 매크로 정의

4.2.1 MIN / MAX

: C에서는 min,max 함수가 정의되지 않았기 때문에 매크로를 사용해 정의해준다.

4.2.2 입력관련 에러 상수 정의

#define ERROR\_CHECKED (-1)

#define INPUT\_ERROR (0)

#define INPUT\_NORMAL (1)

#define INPUT\_EXIT (2)

4.2.3 메모리 관련 에러 상수 정의

#define MEMORY\_ERROR\_VALUE\_EXCEED (3)

#define MEMORY\_ERROR\_ADDRESS\_EXCEED (4)

#define MEMORY\_ERROR\_ADDRESS\_START\_END (5)

4.2.4 opcode 관련 에러 상수 정의

#define OPCODE\_DOESNT\_EXIST (6)

4.2.5 MemoryInstructions.c 내부 상수 정의

#define INDEX\_MAX 1048575 // 65536\*16-1

#define VALUE\_MIN 32 // 0x20

#define VALUE\_MAX 126 // 0x7E

4.2.6 OpcodeTable.c 내부 상수 정의

#define TABLE\_MAX (20)

4.3 InstructionProcessing.c 전역변수

4.3.1 instructionSet

: instructionSetInit() 함수를 통해 만들 명령어 linked list를 가리키는 pointer

4.3.2 historySet

: storeHistory() 함수를 통해 만들 history linked list를 가리키는 pointer

4.3.3 int (\*functionPointer[20])(int,char[100][100]);

: 각 함수를 가리키는 함수포인터를 저장하는 배열

4.4 MemoryInstructions.c 전역변수

4.4.1 unsigned char memory[65536][16] = {0,};

: memory 관련 명령어를 수행할 memory를 저장하는 배열

4.5 OpcodeTable.c 전역변수

4.5.1 opcodeNode\* hashTable[TABLE\_MAX];

: 각 mnemonic과 opcoded를 저장하는 hashTable. 만약 hash값이 충돌을 일으킨다면 linked list를 사용해 다음 노드를 저장한다.

5. 코드 설명

: 해당 프로젝트는 각 함수들을 모듈화하는 것을 중점으로 구성되었습니다. 때문에 각 기능별로 함수가 분할되어있고, c파일 또한 기능별로 분할되어있습니다. 전체적인 구동순서는 다음과 같습니다.

(1) 초기화

(2) 명령어 입력

(3) 명령어 파싱

(4) 명령어 분류

(5) 각 명령어에 해당하는 함수 호출

(6) 함수 기능 수행

(7) 함수 정상 종료 시 history저장

(8) 에러 발생시 에러처리함수에서 에러 출력

(9) (2) 반복

이러한 동작방식은 2.1의 순서도를 통해 더 자세히 확인할 수 있습니다.

5.1 operations.txt 파일 설명

: operations.txt는 프로그램의 유지보수를 용이하게 하기 위해 각 명령어에 대한 정보를 저장한 txt 파일입니다. 파일의 각 줄은 명령어와 그 명령어에 해당하는 함수를 가리키는 함수 포인터 배열의 index를 나타냅니다. 파일의 내용은 아래와 같습니다.

h 1

help 1

d 2

dir 2

q 3

quit 3

hi 4

history 4

du 5

dump 5

e 6

edit 6

f 7

fill 7

reset 8

opcode 9

opcodelist 10

5.2 20161641.h 코드

#ifndef MY20161641

#define MY20161641

#include <stdio.h>

#include <dirent.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#include <dirent.h>

#include <sys/stat.h>

////////

#define MAX(a,b) ({ \_\_typeof\_\_ (a) \_a = (a); \_\_typeof\_\_ (b) \_b = (b); \_a > \_b ? \_a : \_b; })

#define MIN(a,b) ({ \_\_typeof\_\_ (a) \_a = (a); \_\_typeof\_\_ (b) \_b = (b); \_a < \_b ? \_a : \_b; })

////////

// 입력 관련 에러

#define INPUT\_ERROR (0)

#define INPUT\_NORMAL (1)

#define INPUT\_EXIT (2)

////////

// 메모리 명령어 관련 에러

#define MEMORY\_ERROR\_VALUE\_EXCEED (3)

#define MEMORY\_ERROR\_ADDRESS\_EXCEED (4)

#define MEMORY\_ERROR\_ADDRESS\_START\_END (5)

////////

// opcode 관련 에러

#define OPCODE\_DOESNT\_EXIST (6)

////////

typedef struct \_instructionNode{

char str[100];

int caseNum;

struct \_instructionNode\* link;

}instructionNode; // instruction을 저장하는 linked list의 노드

typedef struct \_historyNode{

int count;

char str[100];

struct \_historyNode\* link;

}historyNode; // history를 저장하는 linked list의 노드

typedef struct \_opcodeNode{

int opcode;

char str[100];

int val[2];

struct \_opcodeNode\* link;

}opcodeNode; // opcode를 저장하는 linked list의 노드

////////

// functions in InstructionProcessing.c

void instructionSetInit(); // 명령어 종류를 읽어들여, 명령어 리스트를 만든다

void loadOneInstruction(FILE \*fp,instructionNode \*\*instSet); // 재귀적으로 명령어 리스트 생성

void parser(char str[], int\* argv, char argc[100][100], char sep[]); // 문자열을 파싱

int classifyInput(int argv, char argc[100][100]); // 명령어를 분류

void storeHistory(char str[]);// 규칙에 맞는 명령어 저장

int printHistory(int argv, char argc[100][100]); // 명령어 기록 출력 (함수 포인터 4)

void recurPrintOneHistory(historyNode\* history); // 재귀적으로 하니씩 출력

////////

////////

// functions in functions.c

void printError(int errorCase); // 에러 종류별 에러구문 출력

int inappropriateInput(int argv, char argc[100][100]); // 적합하지 않은 명령어 처리 (함수 포인터 0)

int help(int argv, char argc[100][100]); // 도움말 출력 (함수 포인터 1)

int printDirectory(int argv, char argc[100][100]); // 현재위치 디랙토리 파일 출력 (함수 포인터 2)

int quitProgram(int argv, char argc[100][100]); // 프로그램 종료 (함수 포인터 3)

////////

////////

// functions in MemoryInstructions.c

int isHex(char str[]); // 16진수인지 확인하기 위해 각각의 char을 검사한다

void printMemoryOneLine(int memoryIdx,int s,int e); // 메모리의 내용물을 1줄 출력

int dumpMemory(int argv, char argc[100][100]); // 메모리의 내용을 출력

int editMemory(int argv, char argc[100][100]); // 메모리의 address 번지의 값을 value로 변경

int fillMemory(int argv, char argc[100][100]); // start 번지부터 end 번지까지의 값을 value로 변경

int resetMemory(int argv, char argc[100][100]); // 메모리 전체를 전부 0으로 변경

////////

////////

// functions in OpcodeTable.c

int returnHash(char str[]); // string에 해당하는 hash값 반환

void makeOpcodeTable(); // opcode hash table을 만든다

int opcodeMnemonic(int argv, char argc[100][100]); // 명령어에 해당하는 opcode를 출력

int recurFindOpcode(opcodeNode \*node, char str[]); // 재귀적으로 opcode를 찾는다

int opcodeList(int argv, char argc[100][100]); // opcode hash table을 출력

void recurPrintOneHashTable(opcodeNode \*node); // 재귀적으로 하나씩 출력

////////

#endif

5.3 20161641.c 코드

#include "20161641.h"

extern int (\*functionPointer[20])(int,char[100][100]); // InstructionProcessing.c 에 선언됨

int main(){

char input[100]; // 명령어를 읽어들일 char 배열

int argc; // 파싱 후 분할된 인자들의 수

char argv[100][100]; // 파싱 후 분할되어 저장되는 인자들 저장

char tmpHist[100]; // 파싱 과정에서 input배열이 변경됨으로 history 저장을 위해 기존 명령어 저장

instructionSetInit(); // instruction 들을 읽어들여 저장

makeOpcodeTable(); // opcode hash table을 만든다

while(1){

printf("sicsim> ");

fgets(input,100,stdin);

input[strlen(input)-1] = '\0'; // fgets는 개행까지 읽어들임으로 개행을 '\0'으로 변경

strcpy(tmpHist,input);

parser(input,&argc,argv,", \t"); // 명령어 파싱

int caseNum = functionPointer[classifyInput(argc,argv)](argc,argv);

switch(caseNum){

case INPUT\_ERROR: // 입력이 잘못 들어온 경우

inappropriateInput(argc,argv);

break;

case ERROR\_CHECKED: // 에러 처리를 완료한 경우

break;

case INPUT\_EXIT: // quit, q 명령어가 입력된 경우

return 0; // 프로그램 종료 (정상적으로 종료 : retuen 0)

case INPUT\_NORMAL: // 적합한 명령어가 입력되어 정상적으로 처리가 완료된 경우

storeHistory(tmpHist); // 명령어 저장

break;

default : // 다른 경우는 모두 에러가 생긴 경우

printError(caseNum); //에러문 출력

}

}

return 1; // 비정상적으로 종료함 (return 1)

}

5.4 InstructionProcessing.c 코드

#include "20161641.h"

instructionNode\* instructionSet = NULL;

historyNode\* historySet = NULL;

int (\*functionPointer[20])(int,char[100][100])={\

inappropriateInput,\

help,\

printDirectory,\

quitProgram,\

printHistory,\

dumpMemory,\

editMemory,\

fillMemory,\

resetMemory,\

opcodeMnemonic,\

opcodeList}; // 명령어에 대응하는 함수들의 함수포인터를 저장하는 배열

void instructionSetInit(){ // 명령어 종류를 읽어들여, 명령어 리스트를 만든다

FILE \*fp = fopen("operations.txt","r");

loadOneInstruction(fp,&instructionSet);

fclose(fp);

}

void loadOneInstruction(FILE \*fp,instructionNode \*\*instSet){ // 재귀적으로 명령어 리스트 생성

char str[100];

if(fscanf(fp,"%s",str)==EOF)

return;

\*instSet=(instructionNode\*)malloc(sizeof(instructionNode));

strcpy((\*instSet)->str,str);

fscanf(fp,"%d",&(\*instSet)->caseNum);

(\*instSet)->link=NULL;

loadOneInstruction(fp,&(\*instSet)->link);

}

void parser(char str[], int\* argc, char argv[100][100],char sep[]){

char\* token;

\*argc = 0;

token = strtok(str,sep);

while(token != NULL){

strcpy(argv[(\*argc)++],token);

token = strtok(NULL, sep);

}

}

int classifyInput(int argv, char argc[100][100]){ // 명령어를 분류

if(argv == 0)

return INPUT\_ERROR;

for(instructionNode \*tempNode=instructionSet;tempNode != NULL;tempNode = tempNode->link){

if(strcmp(tempNode->str,\*argc))

continue;

return tempNode->caseNum;

}

return INPUT\_ERROR;

}

void storeHistory(char str[]){ // 규칙에 맞는 명령어 저장

historyNode\* newNode =(historyNode\*)malloc(sizeof(historyNode));

strcpy(newNode->str,str);

newNode->count = historySet == NULL ? 1 : historySet->count+1;

newNode->link = historySet;

historySet=newNode;

}

int printHistory(int argc, char argv[100][100]){ // 명령어 기록 출력 (함수 포인터 4)

if(argc != 1)

return INPUT\_ERROR;

recurPrintOneHistory(historySet);

printf("%-7d %s\n",historySet==NULL ? 1 : historySet->count+1,argv[0]); // 자신을 호출한 hist 명령어 출력해야 함

return INPUT\_NORMAL;

}

void recurPrintOneHistory(historyNode \*history){ // 재귀적으로 하나씩 출력

if(history==NULL)

return;

recurPrintOneHistory(history->link);

printf("%-7d %s\n",history->count,history->str);

}

5.5 functions.c 코드

#include "20161641.h"

void printError(int errorCase){ // 에러 종류별 에러구문 출력

switch(errorCase){

case MEMORY\_ERROR\_VALUE\_EXCEED:

printf("VALUE OUT OF RANGE\n");

break;

case MEMORY\_ERROR\_ADDRESS\_EXCEED:

printf("ADDRESS OUT OF RANGE\n");

break;

case MEMORY\_ERROR\_ADDRESS\_START\_END:

printf("START ADDRESS IS BIGGER THEN END ADDRESS\n");

break;

case OPCODE\_DOESNT\_EXIST:

printf("OPCODE DOESN'T EXIST\n");

}

}

int inappropriateInput(int argc,char argv[100][100]){ // 적합하지 않은 명령어 처리 (함수 포인터 0)

if(argc != 0)

printf("INAPPROPRIATE INPUT\n");

return INPUT\_ERROR;

}

int help(int argc,char argv[100][100]){ // 도움말 출력 (함수 포인터 1)

if(argc != 1)

return INPUT\_ERROR;

printf("h[elp]\n");

printf("d[ir]\n");

printf("q[uit]\n");

printf("hi[story]\n");

printf("du[mp] [start, end]\n");

printf("e[dit] address, value\n");

printf("f[ill] start, end, value\n");

printf("reset\n");

printf("opcode mnemonic\n");

printf("opcodelist\n");

return INPUT\_NORMAL;

}

int printDirectory(int argc, char argv[100][100]){ // 현재위치 디랙토리 파일 출력 (함수 포인터 2)

if(argc != 1)

return INPUT\_ERROR;

DIR\* dir = NULL;

struct dirent\* entry;

struct stat buf;

if((dir = opendir("./")) == NULL){

printf("DIRECTORY OPEN ERROR\n");

return INPUT\_NORMAL;

}

int count = 0;

char str[35];

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

lstat(entry->d\_name,&buf);

if(strlen(entry->d\_name)>=29){ // 이름이 매우 긴 경우 "..."으로 생략

strncpy(str,entry->d\_name,26);

str[26] = str[27] = str[28] = '.';

str[29] = '\0';

}

else{

strncpy(str,entry->d\_name,29);

str[(int)strlen(entry->d\_name)] = '\0';

}

if(S\_ISDIR(buf.st\_mode)){

printf("%c[1;34m",27); // 디랙토리인 경우 파란색으로 출력 (추가구현)

printf("%\*.\*s",31,(int)strlen(entry->d\_name)+1,strcat(str,"/"));

printf("%c[0m",27);

}

else if(S\_ISREG(buf.st\_mode) && (S\_IEXEC&buf.st\_mode)){

printf("%c[1;32m",27); // 실행파일인 경우 초록색으로 출력 (추가구현)

printf("%\*.\*s",31,(int)strlen(entry->d\_name)+1,strcat(str,"\*"));

printf("%c[0m",27);

}

else

printf("%\*.\*s",31,(int)strlen(entry->d\_name),str);

count++;

if(count%3 == 0)

printf("\n");

}

printf("\n");

closedir(dir);

return INPUT\_NORMAL;

}

int quitProgram(int argc, char argv[100][100]){ // 프로그램 종료 (함수 포인터 3)

if(argc != 1)

return INPUT\_ERROR;

return INPUT\_EXIT; // 프로그램 종료를 위해 INPUT\_EXIT return

}

5.6 MemoryInstruction.c 코드

#include "20161641.h"

#define INDEX\_MAX 1048575 // 65536\*16-1

#define VALUE\_MIN 32 // 0x20

#define VALUE\_MAX 126 // 0x7E

unsigned char memory[65536][16] = {0,};

int isHex(char str[]){ // 16진수인지 확인하기 위해 각각의 char을 검사한다

for(int i = 0;i<strlen(str);i++){

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

continue;

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

continue;

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

continue;

return 0; // 16진수의 값이 아닌 경우

}

return 1; // 16진수가 맞는 경우

}

void printMemoryOneLine(int memoryIdx,int s,int e){ // 메모리의 내용물을 1줄 출력

printf("%05X ",memoryIdx\*16);

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

printf(" ");

for(int i = s;i<=e;i++)

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

for(int i = e+1;i<16;i++)

printf(" ");

printf("; ");

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

printf(".");

for(int i = s;i<=e;i++)

printf("%c",(VALUE\_MIN<=memory[memoryIdx][i]&&memory[memoryIdx][i]<=VALUE\_MAX) ? \

memory[memoryIdx][i] : '.');

for(int i = e+1;i<16;i++)

printf(".");

printf("\n");

}

int dumpMemory(int argv, char argc[100][100]){ // 메모리의 내용을 출력

static int prevIndex = -1; // 마지막 실행주소를 static으로 저장

if(argv>3)

return INPUT\_ERROR;

for(int i = 1;i<argv;i++)

if(!isHex(argc[i]))

return INPUT\_ERROR;

int index[2];

index[0] = prevIndex == INDEX\_MAX ? 0 : prevIndex+1; // start index

index[1] = MIN(index[0]+159,INDEX\_MAX); // end index

for(int i = 1;i<argv;i++)

index[i-1] = strtol(argc[i],NULL,16);

if(argv == 2)

index[1] = MIN(index[0]+159,INDEX\_MAX);

if(index[0]>index[1])

return MEMORY\_ERROR\_ADDRESS\_START\_END;

if(index[1]>INDEX\_MAX)

return MEMORY\_ERROR\_ADDRESS\_EXCEED;

// index 예외처리 후. index에 따라 출력

if(index[0]/16 == index[1]/16)

printMemoryOneLine(index[0]/16,index[0]%16,index[1]%16);

else{

printMemoryOneLine(index[0]/16,index[0]%16,15);

for(int i = index[0]/16+1;i<index[1]/16;i++){

printMemoryOneLine(i,0,15);

}

printMemoryOneLine(index[1]/16,0,index[1]%16);

}

prevIndex = index[1];

return INPUT\_NORMAL;

}

int editMemory(int argv, char argc[100][100]){ // 메모리의 address 번지의 값을 value로 변경

if(argv!=3)

return INPUT\_ERROR;

for(int i = 1;i<argv;i++)

if(!isHex(argc[i]))

return INPUT\_ERROR;

int address = strtol(argc[1],NULL,16);

int value = strtol(argc[2],NULL,16);

if(address>INDEX\_MAX || address<0)

return MEMORY\_ERROR\_ADDRESS\_EXCEED;

if(value>255|| value < 0)

return MEMORY\_ERROR\_VALUE\_EXCEED;

// address, value 예외처리 후, memory의 address위치에 value 할당

memory[address/16][address%16] = value;

return INPUT\_NORMAL;

}

int fillMemory(int argv, char argc[100][100]){ // start 번지부터 end 번지까지의 값을 value로 변경

if(argv!=4)

return INPUT\_ERROR;

for(int i = 1;i<argv;i++)

if(!isHex(argc[i]))

return INPUT\_ERROR;

int index[2];

int value;

for(int i = 1;i<3;i++)

index[i-1] = strtol(argc[i],NULL,16);

value = strtol(argc[3],NULL,16);

if(index[0]>index[1])

return MEMORY\_ERROR\_ADDRESS\_START\_END;

if(index[1]>INDEX\_MAX || index[0]<0)

return MEMORY\_ERROR\_ADDRESS\_EXCEED;

if(value>255|| value < 0)

return MEMORY\_ERROR\_VALUE\_EXCEED;

// address, value 예외처리 후, memory의 address위치에 value 할당

for(int i = index[0];i<=index[1];i++){

memory[i/16][i%16] = value;

}

return INPUT\_NORMAL;

}

int resetMemory(int argv, char argc[100][100]){ // 메모리 전체를 전부 0으로 변경

if(argv!=1)

return INPUT\_ERROR;

for(int i = 0;i<65536;i++)

for(int j = 0;j<16;j++)

memory[i][j] = 0;

return INPUT\_NORMAL;

}

5.7 OpcodeTable.c 코드

#include "20161641.h"

#define TABLE\_MAX (20)

opcodeNode\* hashTable[TABLE\_MAX];

int returnHash(char str[]){ // string에 해당하는 hash값 반환

int hash = 401;

for(int i = 0;str[i] != '\0';i++)

hash = ((hash<<4)+(int)(str[i]))%TABLE\_MAX;

return hash%TABLE\_MAX;

}

void makeOpcodeTable(){ // opcode hash table을 만든다

FILE \*fp = fopen("opcode.txt","r");

int argc,hashNum;

char str[100],argv[100][100];

while(fgets(str,100,fp)!=NULL){

parser(str,&argc,argv," /\t\n");

opcodeNode \*newNode = (opcodeNode\*)malloc(sizeof(opcodeNode));

strcpy(newNode->str,argv[1]);

newNode->opcode = strtol(argv[0],NULL,16);

newNode->val[0] = atoi(argv[2]);

newNode->val[1] = argc == 4 ? atoi(argv[3]):0;

hashNum = returnHash(newNode->str);

newNode->link = hashTable[hashNum];

hashTable[hashNum] = newNode;

}

fclose(fp);

}

int opcodeMnemonic(int argv, char argc[100][100]){ // 명령어에 해당하는 opcode를 출력

if(argv != 2)

return INPUT\_ERROR;

int tmp;

for(int i = 0;i<TABLE\_MAX;i++){

tmp = recurFindOpcode(hashTable[i],argc[1]);

if(tmp!=-1){

printf("opcode is %X\n",tmp);

return INPUT\_NORMAL;

}

}

return OPCODE\_DOESNT\_EXIST;

}

int recurFindOpcode(opcodeNode \*node,char str[]){ // 재귀적으로 opcode를 찾는다

if(node == NULL)

return -1;

if(strcmp(node->str,str) == 0)

return node->opcode;

return recurFindOpcode(node->link,str);

}

int opcodeList(int argv, char argc[100][100]){ // opcode hash table을 출력

if(argv != 1)

return INPUT\_ERROR;

for(int i = 0;i<TABLE\_MAX;i++){

printf("%d: ",i);

recurPrintOneHashTable(hashTable[i]);

printf("\n");

}

return INPUT\_NORMAL;

}

void recurPrintOneHashTable(opcodeNode \*node){ // 재귀적으로 하나씩 출력

if(node==NULL)

return;

printf("[%s %X] ",node->str,node->opcode);

if(node->link != NULL)

printf("-> ");

recurPrintOneHashTable(node->link);

}