과목 명: 시스템프로그래밍

담당 교수 명: 박 운 상

<과제1 프로젝트 보고서>

**컴퓨터공학과**

**20161665**

**황세현**

[목 차]

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
3. **모듈 정의**
   1. int main()
   2. void init\_queue(Queue \*)
   3. void init\_memory(char a[][16])
   4. void create\_hashTable()
   5. void func\_help()
   6. void func\_history(Queue \*)
   7. int func\_dir()
   8. void func\_dump(char a[][16],int start, int end, int \*, int \*)
   9. void func\_fill(char a[][16], int start, int end, int val)
   10. void func\_quit (Queue \*)
   11. void func\_opcodelist()
   12. void func\_opcode\_mnemonic(char \*)
   13. void add\_Node\_To\_Queue (char \*, Queue \*)
   14. void divide\_str(char \*, char \*, int \*, int \*, int \*, char \*)
   15. int correct\_Addr(char a)
   16. int find\_group(char \*)
   17. int get\_string(char \*, char\*, int \*, int )
   18. int test\_tail(char \*,int )
   19. int string\_to\_number(char \*, int )
   20. int func\_hash(char \*)
   21. int IS\_VALID(char a)
   22. int IS\_NUMBER(char a)
   23. int IS\_UPPER(chra a)
   24. int dec\_to\_hexa\_digit(int a);
4. **전역 변수 정의**

4.1 static int Pow16

4.2 hashNode\* hashTable[20]

1. **코드**

5.1 main 함수

5.2 dump 함수

5.3 문자열 분리 함수

1. **기타**

6.1 예외처리

6.2 기타 구현

# 프로그램 개요

프로그램 shell과 메모리공간, 그리고 opcode 테이블에 대한 기본 지식을 갖고 , 이를 linux 개발 환경을 이용하여 가상으로 설계해보는 프로그램이다.

# 프로그램 설명

## 프로그램 흐름도

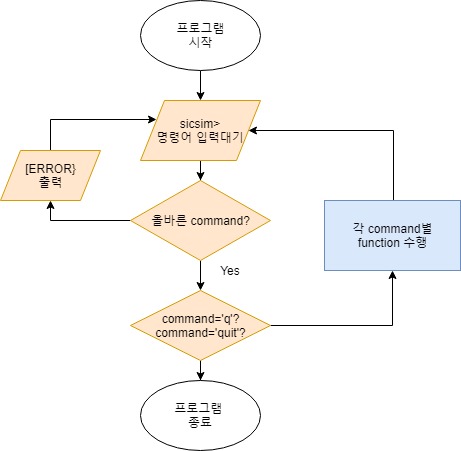


Figure 1 ) 프로그램의 전체적인 흐름

프로그램이 시작되면 sicsim>이 화면에 나타나면서 사용자의 입력을 대기하는 상태가 된다. 사용자가 입력한 명령이 올바르다면 그에 맞는 명령어 기능처리를 한 뒤, 그 입력 문자열이 history에 저장된다. 그 후, 다시 입력을 대기하는 상태로 돌아간다. 만약 올바르지 않은 입력일 경우에는 에러코드를 출력한 뒤, 사용자의 입력을 대기하는 상태가 된다. 입력으로 ‘q’ 또는 ‘quit’이 들어올 경우 프로그램은 정상적으로 종료된다.

# 모듈 정의

# [main 함수]

## 모듈 이름 : int main()

### 기능

While loop 내에서 명령어를 입력받고, 올바른 명령어라면 각각의 모듈을 호출하여 기능을 실행시킨다.

### 사용 변수

int memory[66536][16] 1MB의 가상 메모리 공간을 형성하는 배열

int tmpOrder[81], order[30] tmpOrder에 초기 문자열 저장. Order에 올바른 명령어 저장

int valid True/False 값 만을 가지며 올바른 명령어일 때 valid=True

int lastAddr, lastLine dump 명령 수행 시, 마지막 주소와 마지막 라인 정보를 저장

int a,b,c 올바른 명령어가 입력됐을 때, 주소 및 value 값을 저장

Queue h\_queue 명령어로 입력된 history정보를 담고 있을 queue 자료구조

char mnemo[8] mnemonic 정보를 담아낼 문자열

## [초기화 함수]

## 모듈 이름: void init\_queue(Queue \*)

### 기능

History 정보 저장에 사용될 queue 자료구조를 초기화한다.

### 사용 변수

없음

## 모듈이름: void init\_memory(char a[][16])

### 기능

1MB의 가상 메모리 공간에 저장된 정보들을 모두 0으로 초기화한다. 프로그램을 처음 시작했을 때 함수가 호출되며, command로 ‘reset’이 입력될 경우에도 함수가 호출된다.

### 사용변수

없음

## 모듈이름: void create\_hashTable()

### 기능

프로그램이 시작될 때 opcode.txt에서 읽어온 mnemonic 정보들을 바탕으로 opcodelist 명령어 기능에 사용될 hashtable 을 생성한다.

### 사용변수

FILE \*fp opcode.txt 텍스트 파일의 정보를 읽어올 파일 포인터

char str[8] mnemonic 문자열이 저장될 변수

unsigned int num mnemonic의 16진수 정보를 담을 정수

hashNode\* newNode, \*tmpPtr hashTable 노드 형성에 필요한 노드 포인터

## [각 명령어 기능별 함수]

## 모듈이름: void func\_help()

### 기능

가능한 명령어들을 화면상에 출력해주는 help기능을 수행한다.

### 사용변수

없음

## 모듈이름: void func\_history(Queue \*)

### 기능

Queue 에 저장된 history 정보가 존재할 경우 처음부터 끝까지 출력해준다.

### 사용변수

Int i 인덱스 변수

Node \*tmpNode 자료 출력에 사용될 임시 노드 포인터

## 모듈이름: int func\_dir()

### 기능

현재 디렉토리에 존재하는 모든 파일과 디렉토리 정보를 출력해준다.

### 사용변수

DIR \*dir 현재 디렉토리 정보를 담아올 DIR포인터

char str[10] 파일이름을 담을 문자 배열

struct dirent \*tmp 디렉토리 내의 파일 정보를 하나씩 담아올 dirent 포인터

struct dirent state 각 파일의 속성을 조사할 struct dirent변수

## 모듈이름: void func\_dump(char a[][16], int start, int end, int \*,int \*)

### 기능

사용자가 입력한 시작 주소와 끝 주소를 바탕으로 1MB 가상 메모리 정보를 화면에 출력해주는 함수이다. 1. dump 2. dump start 3. dump start, end 의 3가지 명령어를 처리해야한다. 기능이 종료된 후에는 마지막 주소정보를 포인터를 이용하여 새롭게 갱신한다.

### 사용변수

int start, end 메모리 시작, 끝 주소를 저장

int return\_to\_start True/False값을 갖는 변수. 만약 메모리 끝 주소가 허용되는 범위를 초과했을 경우 return\_to\_start=True가 된다.

int i, j, tmp for loop 연산에 쓰이는 변수들

## 모듈이름: void func\_fill(char a[][16], int start, int end, int val)

### 기능

처음 주소와 끝 주소를 인자로 넘겨받아 그 사이에 위치한 메모리들을 인자로 받은 value로 채우는 기능을 수행한다.

### 사용변수

int i, j, tmp for loop 연산에 쓰이는 변수들

## 모듈이름: void func\_quit(Queue \*)

### 기능

명령어로 ‘quit’또는 ‘q’가 들어오게 될 경우, 모든 동적으로 할당받은 메모리들을 해지한뒤 프로그램을 종료한다.

### 사용변수

Node \*del Queue구조로 이뤄진 history목록 삭제에 필요한 노드 포인터

hashNode \*del2 hashTable로 구성된 opcodelist 삭제에 필요한 노드 포인터

## 모듈이름: void func\_opcodelist()

### 기능

hashTable내에 저장된 opcode들을 hashTable 형태 그대로 출력한다.

### 사용변수

hashNode\* tmp opcode출력에 필요한 노드 포인터

## 모듈이름: void func\_opcode\_mnemonic(char \*)

### 기능

mnemonic이 입력되면 해당 mnemonic에 해당하는 opcode를 16진수의 형태로 출력한다

### 사용변수

int idx func\_hash를 호출하여 mnemonic에 해당하는 인덱스 값을 담아온다.

int find True/False값을 가지며 mnemonic에 대한 인덱스 값이 존재할 경우 find=True가 된다.

**[부가적인 함수]**

## 모듈이름: void add\_Node\_To\_Queue(char \*, Queue \*)

### 기능

올바른 명령어라면 입력된 명령어에 대한 기능을 수행한 뒤, history목록을 구성하는 queue에 노드를 추가하는 역할을 한다. 동적할당을 통하여 새로운 노드를 형성한다.

### 사용변수

Node \*tmpNode 동적할당으로 생성될 노드를 담고있다.

## 모듈이름: void divide\_str(char \*,char \*, int\*, int\*, int \*,int \*,char \*)

### 기능

명령어 처리의 가장 핵심적인 역할을 하는 함수이다. 처음 상태의 문자열을 인자로 넘겨받아 입력된 명령어가 정확히 무엇인지 분리하는 역할을 한다. 정상적인 명령어가 들어올 경우 order이라는 문자배열에 담아 전달하고, 그렇지 않을 경우에는 에러코드를 담아 전달한다. 명령어들을 효율적으로 분리하기 위해 get\_string과 test\_tail의 이름을 가진 반복적으로 호출된다. 함수의 초반에서는 주소 값이나 value를 필요로 하는 명령어와 그렇지 않은 나머지 명령어들을 따로 분리한 뒤, 각각의 처리 방법을 달리하여 분리 기능을 수행한다.

### 사용변수

int start 명령어 분리과정에서 다음 명령어 분리를 위해 시작 인덱스의 위치 정보를 담고있는 변수

char tmp[20] 분리된 command 나 address, value 및 mnemonic의 정보를 임시로 담고있을 문자열

## 모듈이름: int correct\_Addr(char a)

### 기능

인자로 넘겨받은 16진수 표현의 문자를 숫자로 알맞게 조정하여 반환하는 함수이다.

### 사용변수

없음

## 모듈이름: int find\_group(char \*)

### 기능

명령어에 따른 처리방법을 달리해야 하므로 처리방법이 같은 명령어들을 한 그룹으로 묶어서 해당하는 그룹의 정보를 반환하는 함수이다.

### 사용변수

없음

## 모듈이름: int get\_string(char \*,char \*,int \*, int )

### 기능

문자열 분리 기능의 핵심적인 역할을 하는 함수로, 처음 입력 받은 명령어에서 문자열 단위로 하나씩 추출하는 역할을 한다. 다음 문자열 추출을 위해 시작 인덱스 값을 새롭게 갱신해주고, 만약 추출된 문자열이 올바르지 않다면 에러코드를 반환한다.

### 사용변수

int no\_string True/False 값을 갖는 변수로 올바르지 않은 명령어가 입력될 경우 no\_string=True가 된다.

int valid 문자열 내에서 유효한 문자의 개수를 저장하는 변수이다.

int count 함수 내에서 콤마의 개수에 따른 예외처리가 필요하기 때문에 콤마의 개수를 저장하는 변수이다.

## 모듈이름: int test\_tail(char \*,int )

### 기능

위에서 언급한 get\_string 함수와 더불어 문자열 분리기능의 핵심적인 역할을 담당하는 함수이다. 문자열 추출이 모두 완료된 후, 문자열이 완전히 종료 (‘\0’)될 때까지 탭, 공백 문자 외에 다른 문자가 나올 경우 0을 반환한다. 정상종료시 1을 반환한다.

### 사용변수

없음

## 모듈이름: int string\_to\_number(char \*, int )

### 기능

correct\_Addr 함수가 1개의 문자를 숫자로 변환하는 함수라면 이 함수는 문자열 전체를 숫자로 변환하는 함수이다. 주소나 value 값을 필요로 하는 명령어가 입력될 시 문자열 형태로 입력된 16진수를 알맞은 숫자로 변환하는 역할을 한다. 문자열로 들어온 값이 주소를 뜻하는지 value값을 뜻하는지 알 수 없기 때문에 flag를 인자로 받아 함수 기능을 실행한다.

(flag = 1 : 주소, flag = 0 : value)

### 사용변수

int len 인자로 전달받은 문자열의 길이를 저장하는 변수

int sum 정수로 변환된 주소 또는 value 값을 저장하는 변수

## 모듈이름: int func\_hash(char \*)

### 기능

mnemonic이 입력되면 일정한 규칙에 의해 생성된 해당 mnemonic의 인덱스를 반환한다. 프로그램 초반 hashTable이 형성될 때 반복적으로 호출된다.

### 사용변수

int len 인자로 받은 mnemonic의 길이를 저장하는 변수

int sum hash 함수의 규칙에 따른 인덱스 반환을 위해 사용되는 변수

## 모듈이름: int IS\_VALID(char a)

### 기능

인자로 전달받은 문자가 16진수 범위내에 있다면 True, 그렇지 않으면 False를 반환한다.

### 사용변수

없음

## 모듈이름: int IS\_NUMBER(char a)

### 기능

인자로 전달받은 문자가 ‘0’과 ‘9’ 사이에 있다면 True, 그렇지 않으면 False 를 반환한다.

### 사용변수

없음

## 모듈이름: int IS\_UPPER(char a)

### 기능

인자로 전달받은 문자가 알파벳 대문자라면 True, 그렇지 않으면 False를 반환한다.

### 사용변수

없음

## 모듈이름: int dec\_to\_hexa\_digit(int a)

### 기능

인자로 전달받은 10진수 정수가 16진수로 변환 시 몇 자리 인지 알려주는 함수.

### 사용변수

없음

# 전역 변수 정의

## static int Pow16[5]

160, 161, 162, 163, 164 값을 각각 배열에 저장한다.

## hashNode \*hashTable[20]

각각의 hashTable[i] 는 노드를 가리키는 포인터로 opcode와 mnemonic의 정보를 담은 노드와 연결된다.

# 5. 코드(핵심 기능 위주)

# 5.1 main함수

int main(){

char tmpOrder[81]; //처음 입력되는 명령어 덩어리

char order[30]; //입력받은 명령어

char memory[66536][16]; //1MB의 메모리할당

int valid=False; //명령어의 유효성을 나타내는 변수

int lastAddr=0,lastLine=0;

int a=-1,b=-1,c=-1; //각각의 변수에 주소 또는 value가 저장됨

int col,row;

Queue h\_queue; //history 기능에 쓰일 자료구조

char mnemo[8];

/\* initialization start \*/

init\_queue(&h\_queue);

init\_memory(memory);

create\_hashTable();

/\* initialization end \*/

while(1){

valid=False; //valid 값 초기화

/\* 문자열 입력 받기 \*/

printf("sicsim> ");

fgets(tmpOrder,sizeof(tmpOrder),stdin);

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

/\* 입력받은 문자열을 분리 \*/

divide\_str(tmpOrder,order,&a,&b,&c,mnemo);

/\* 종료 조건 \*/

if(!strcmp(order,"q\0") || !strcmp(order,"quit\0")){

func\_quit(&h\_queue);

break;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*명령어 처리 시작!\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if(!strcmp(order,"h\0") || !strcmp(order,"help\0")){

func\_help(); //help function 기능 수행

valid=True; //유효한 명령어라면 valid=True

}

else if(!strcmp(order,"d\0") || !strcmp(order,"dir\0")){

func\_dir();

valid=True;

}

else if(!strcmp(order,"hi\0") || !strcmp(order,"history\0")){

if(h\_queue.num==0)

add\_Node\_To\_Queue(tmpOrder,&h\_queue);

else{

add\_Node\_To\_Queue(tmpOrder,&h\_queue);

func\_history(&h\_queue);

}

}

else if(!strcmp(order,"du\0") || !strcmp(order,"dump\0")){

func\_dump(memory,a,b,&lastAddr,&lastLine);

a=b=c=-1; //사용 후 주소, value 값 초기화

valid=True;

}

else if(!strcmp(order,"reset\0")){

init\_memory(memory);

valid=True;

}

else if(!strcmp(order,"edit\0") || !strcmp(order,"e\0")){

col=a/16;row=a%16;

memory[col][row]=b;

a=b=c=-1;

valid=True;

}

else if(!strcmp(order,"fill\0") || !strcmp(order,"f\0")){

func\_fill(memory,a,b,c);

a=b=c=-1;

valid=True;

}

else if(!strcmp(order,"opcodelist\0")){

func\_opcodelist();

valid=True;

}

else if(!strcmp(order,"opcode\0")){

func\_opcode\_mnemonic(mnemo);

valid=True;

}

else{ //에러코드 정리(invalid 명령어들)

if(!strcmp(order,"incorrect\0"))

printf("[ERROR]Invalid command\n");

else if(!strcmp(order,"incorrectAddr\0"))

printf("[ERROR]Invalid address\n");

else if(!strcmp(order,"incorrectVal\0"))

printf("[ERROR]Invalid value\n");

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*명령어 처리 종료!\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if(valid==True) //유효한 명령어라면 history에 저장한다.

add\_Node\_To\_Queue(tmpOrder,&h\_queue);

}

return 0;

}

# 5.2 dump 함수

void func\_dump(char memory[][16],int a,int b,int \*lastAddr,int \*lastLine){

int i,j,tmp,return\_to\_start=False;

int start=a,end=b;

if(start>end && end!=-1){ //start>end일경우 예외처리

printf("[ERROR]Invalid address\n");

return;

}

else{

if(start!=-1){

\*lastLine=start/16;

}

/\* start, end 주소값 보정 \*/

if(start==-1) //start값이 설정돼있지 않을 경우(dump 단독 입력시)

start=\*lastAddr;

if(end==-1){ //end값이 설정돼있지 않을 경우 (dump 단독 or dump start)

end=start+159;

}

for(i=\*lastLine;!return\_to\_start;i++){

/\* memory 주소 출력 부분\*/

switch(dec\_to\_hexa\_digit(i)){

case 1 : printf("000%X0 ",i);break;

case 2 : printf("00%X0 ",i);break;

case 3 : printf("0%X0 ",i);break;

case 4 : printf("%X0 ",i);break;

case 5 : \*lastLine=0;\*lastAddr=0;return\_to\_start=True;

continue;

default : printf("[ERROR]Invalid memory access\n");break;

}

/\* 각 메모리에 저장된 값을 hexa로 출력 \*/

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

if((tmp=i\*16+j)>=start && tmp<=end){

if((unsigned char)memory[i][j]<0x10)

printf("0%X ",(unsigned char)memory[i][j]);

else

printf("%2X ",(unsigned char)memory[i][j]);

}

else

printf(" ");

}

printf("; ");

/\* 각 메모리에 저장된 값을 char로 출력 \*/

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

if((tmp=i\*16+j)>=start && tmp<=end && memory[i][j]>=0x20 && memory[i][j]<=0x7E)

printf("%c",memory[i][j]);

else

printf(".");

}

/\* 반복문 탈출 조건\*/

tmp++;

if(tmp>end){

if(end%16==0xF)

(\*lastLine)++;

break;

}

printf("\n");

(\*lastLine)++;

}

/\*dump 기능 종료 후 새로운 값으로 갱신\*/

if(return\_to\_start) //만약 주소값이 0xfffff를 초과했다면

;

else{

\*lastAddr=end+1; //정상 범위 내의 주소값일때

printf("\n");

}

}

}

# 5.3 문자열 분리 함수

/\* 처음 입력받은 문자열에서 명령어, 주소, value, mnemonic 등을 분리하고

이를 각각의 속성에 알맞게 변환하는 함수. 올바르지 않은 문자열일 때는

문자열 부분에 에러코드를 저장한다. \*/

void divide\_str(char\* origin,char\* order,int \*a,int \*b,int \*c,char \*mnemonic){

int start=0;

char tmp[20];

/\* 문자열 덩어리에서 명령어를 뽑아냄 \*/

if(get\_string(origin,order,&start,0)){

strcpy(order,"incorrect\0");

return;

}

/\* 어떠한 명령어인지 식별하는 과정 \*/

switch(find\_group(order)){

case 1 : //case 1 : 명령어만 단독으로 존재하는 것(no address,value)

if(!test\_tail(origin,start))

strcpy(order,"incorrect\0");

return;

case 2 : //case 2 : opcode mnemonic

if(get\_string(origin,tmp,&start,0)){ //mnemonic문자열 get

strcpy(order,"incorrect\0");

return;

}

if(!test\_tail(origin,start)){

strcpy(order,"incorrect\0");return;

}

strcpy(mnemonic,tmp);

case 3 : //case 3 : dump 명령어

if(test\_tail(origin,start)) return; //dump 명령어만 입력된경우

if(get\_string(origin,tmp,&start,0)){

strcpy(order,"incorrect\0");return; //오류 처리ex) dump ,

}

if((\*a=string\_to\_number(tmp,1))==-1){ //첫번째 주소 저장

strcpy(order,"incorrectAddr\0");return; //올바르지 못한 주소 에러처리

}

if(test\_tail(origin,start)) return; //dump start 명령어 정상처리

if(get\_string(origin,tmp,&start,1)){

strcpy(order,"incorrect\0");return; //오류 처리ex) dump B2 ,

}

if((\*b=string\_to\_number(tmp,1))==-1){ //두번째 주소 저장

strcpy(order,"incorrectAddr\0");return; //올바르지 못한 주소 에러처리

}

if(test\_tail(origin,start)) return;

case 4 : //case 4 : edit 명령어

if(test\_tail(origin,start) || get\_string(origin,tmp,&start,0)){

strcpy(order,"incorrect\0");return;

}

if((\*a=string\_to\_number(tmp,1))==-1){ //첫번째 주소 저장

strcpy(order,"incorrectAddr\0");return;

}

if(test\_tail(origin,start) || get\_string(origin,tmp,&start,1)){

printf("3\n");

strcpy(order,"incorrect\0");return;

}

if((\*b=string\_to\_number(tmp,0))==-1){ //두번째 value 저장

strcpy(order,"incorrectVal\0");return; }

if(!test\_tail(origin,start))

strcpy(order,"incorrect\0");

break;

case 5 : //case 5 : fill 명령

if(test\_tail(origin,start) || get\_string(origin,tmp,&start,0)){

strcpy(order,"incorrect\0");return;

}

if((\*a=string\_to\_number(tmp,1))==-1){ //첫번째 주소 저장

strcpy(order,"incorrectAddr\0");return;

}

if(test\_tail(origin,start) || get\_string(origin,tmp,&start,1)){

strcpy(order,"incorrect\0");return;

}

if((\*b=string\_to\_number(tmp,1))==-1){ //두번째 주소 저장

strcpy(order,"incorrectAddr\0");return;

}

if(test\_tail(origin,start) || get\_string(origin,tmp,&start,1)){

strcpy(order,"incorrect\0");return;

}

if((\*c=string\_to\_number(tmp,0))==-1){ //세번째 value 저장

strcpy(order,"incorrectVal\0");return;

}

if(!test\_tail(origin,start))

strcpy(order,"incorrect\0");

break;

default : strcpy(order,"incorrect\0");return;

}

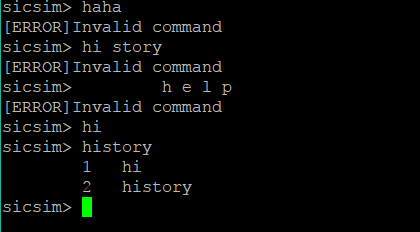
}

# 6 . 기타

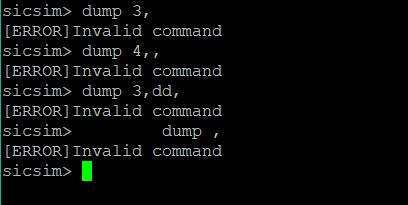
* 1. 예외처리

1. **올바르지 않은 명령어가 입력될 경우** : 존재하지 않는 명령어, 명령어 중간에 여러 개의 공백 문자나 tab, 그리고 올바르지 않은 콤마 위치와 개수가 입력되었을 때 모두 예외처리를 하였다.

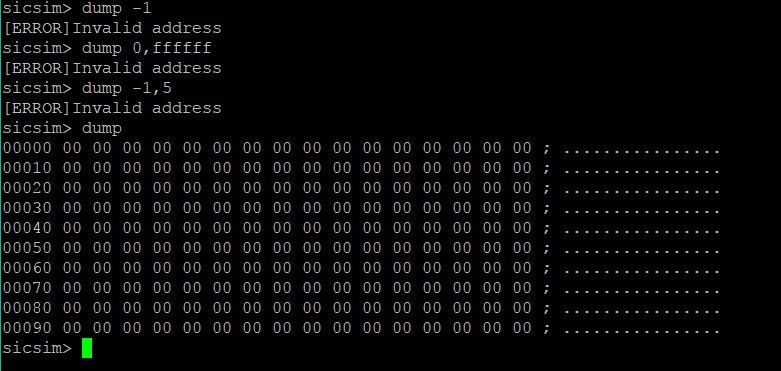
* 허용되지 않는 명령어, 명령어 사이 띄어쓰기



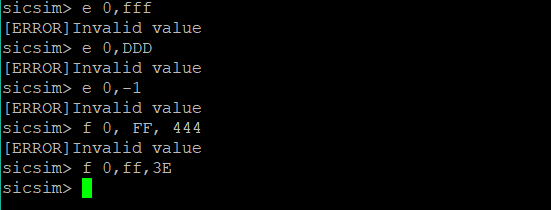
* 콤마 개수 오류



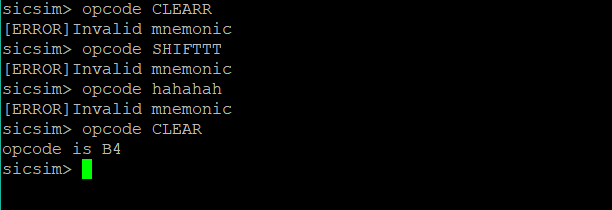
1. **올바르지 않은 주소 값이 입력될 경우** : 16진수 주소의 경우 0~FFFFF 이외의 값을 가질 때 예외처리를 하였다.



1. **올바르지 않은 value 값이 입력될 경우 :** 16진수 value의 경우 0~FF 이외의 값을 가질 때 예외처리를 하였다.

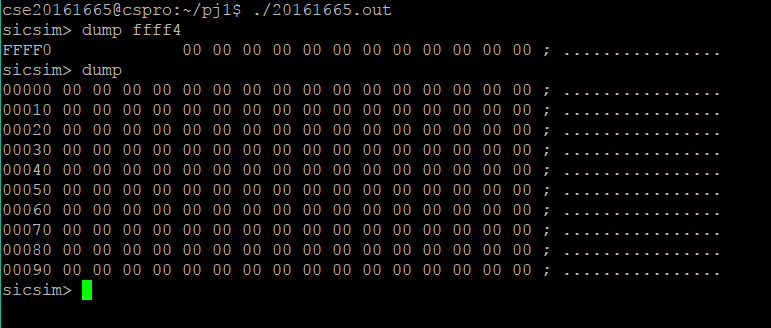


1. **올바르지 않은 mnemonic이 입력될 경우** : opcodelist 에 존재하지 않는 mnemonic 이 입력되었을 경우 예외처리를 하였다.

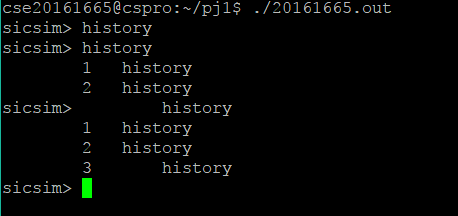


* 1. 기타 구현

1. **dump 명령시 메모리 끝에 도달할 경우, 0부터 다시 시작** : dump 명령을 수행하면서 메모리의 끝에 도달하였을 경우 FFFFF의 주소를 가진 메모리까지만 출력한다. 이 상태에서 다시 dump 명령을 수행하게 될 경우, 메모리는 다시 0 부터 출력된다.

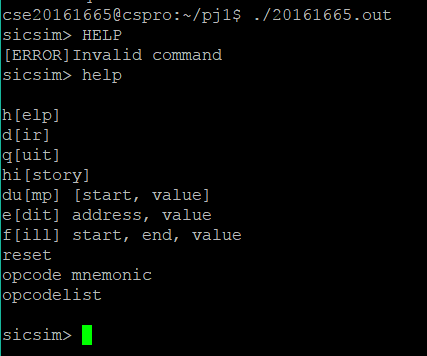


1. **첫번째 history 명령 입력시 출력 제외, 두번째부터 출력 :** history 목록에 아무것도 존재하지 않을 때 history 명령이 입력되면 아무것도 출력하지 않는다. 하지만 두번째 history 명령부터는 바로 출력하도록 한다.

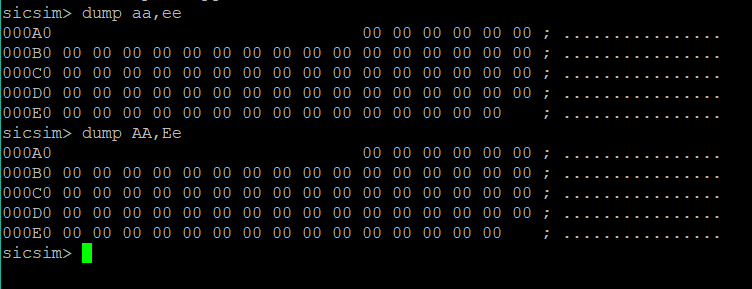


1. **명령어는 소문자만, 16진수는 대/소문자 허용 :** 명령어는 소문자로만 허용되므로 대문자 명령어가 입력될 시, 올바르지 않은 명령어로 인식한다. 하지만 16진수 주소와 value는 대/소문자를 모두 허용하므로 올바른 값으로 받아들인다.

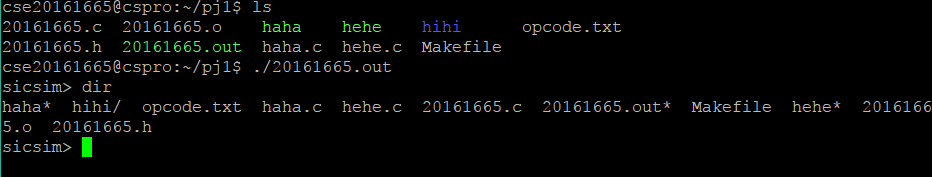
* 명령어 대문자 오류, 소문자 정상작동



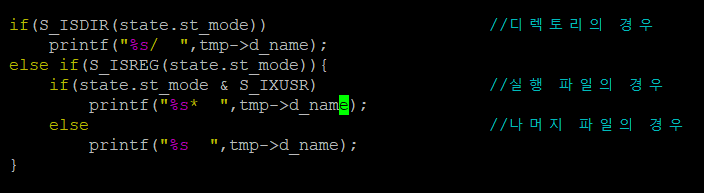
* 16진수 대/소문자 정상작동



1. **&연산을 통한 실행파일 찾기** : 실행 파일 명에 “.out” 이 없더라도 실행 권한을 나타내는 매크로와의 & 연산을 통하여 모든 실행파일을 찾아낼 수 있다.

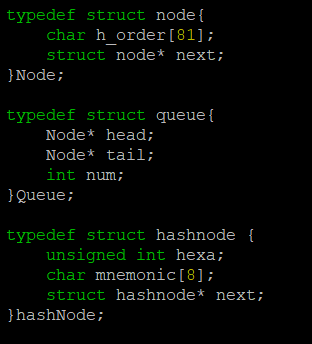


* “dir” 기능 코드의 일부 (& 연산)

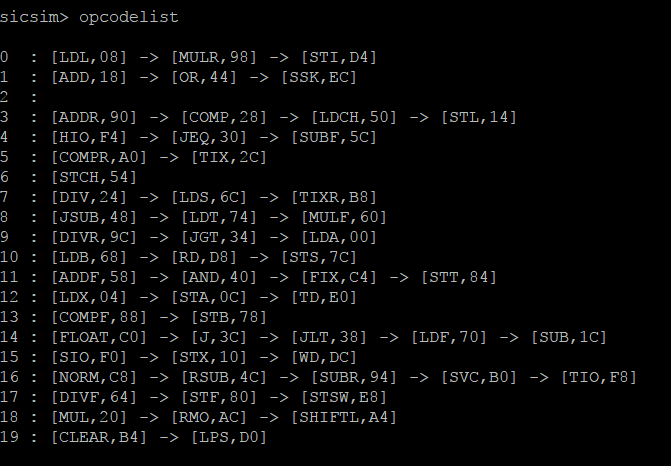


1. **linked list로 구현한 history와 opcodelist :** history는 linked list queue의 형태로, opcodelist는 linked list hashtable의 형태로 구현하였다.

* 헤더 파일에 정의된 linked list 노드



* linked list로 구현한 hashTable



1. **동적으로 할당된 메모리 해제** : history와 opcodelist는 동적으로 생성된 메모리이기 때문에 반드시 종료하기 전에 free해줘야 한다.

* 메모리 free함수는 quit 명령어가 입력될 때 호출된다.

