과목 명: 시스템프로그래밍

담당 교수 명: 소 정 민

<<Assignment 1>>

서강대학교 컴퓨터공학과

[20161631]

[임동진]

목 차

1. 프로그램 개요

SIC/XE 머신 구현을 위한 첫 단계이다.
어셈블러, 링커, 로더들을 실행하게될 쉘과 가상 메모리 공간, opcode table 등의 기본적인 기능을 구현한 프로그램이다.

2. 프로그램 설명

2.1 프로그램 흐름도

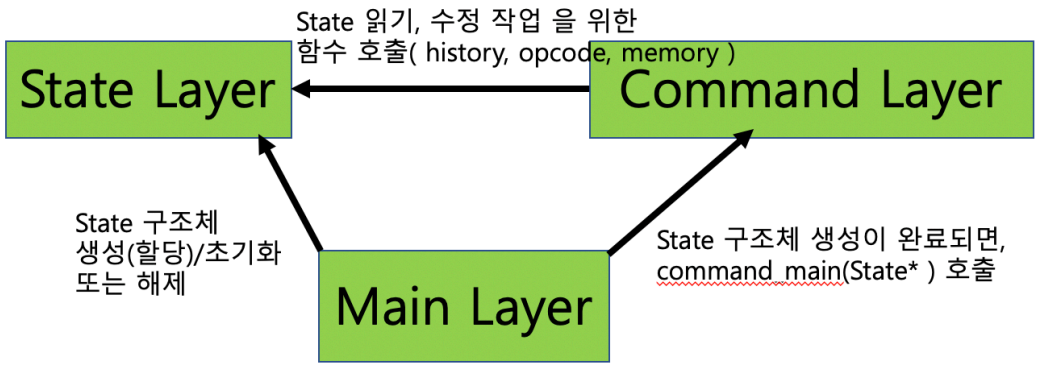

그림 1> 프로그램 전체 구조도

프로그램이 시작되면 main() 함수에서 construct_state()함수를 호출해서
history, opcode, memory 등이 저장되는 State 구조체를 생성(초기화)하여 state_store 라는 변수에 저장한다. 그다음 Command Layer 의 command_main()함수를 호출하여 쉘이 실행되도록 한다.
Command Layer 의 모듈들을 통해 사용자로 부터 입력을 받거나, 토크나이징하고, 명령어를 검증, 실행하는 과정을 수행한다. 또한 State 에 접근해야하는 명령어의 경우에는 State 모듈의 함수를 호출하여 실행한다.
사용자가 quit(q)명령을 입력하면 command_main() 함수를 탈출하고 할당했던 메모리들을 해제하면서 프로그램을 종료한다.

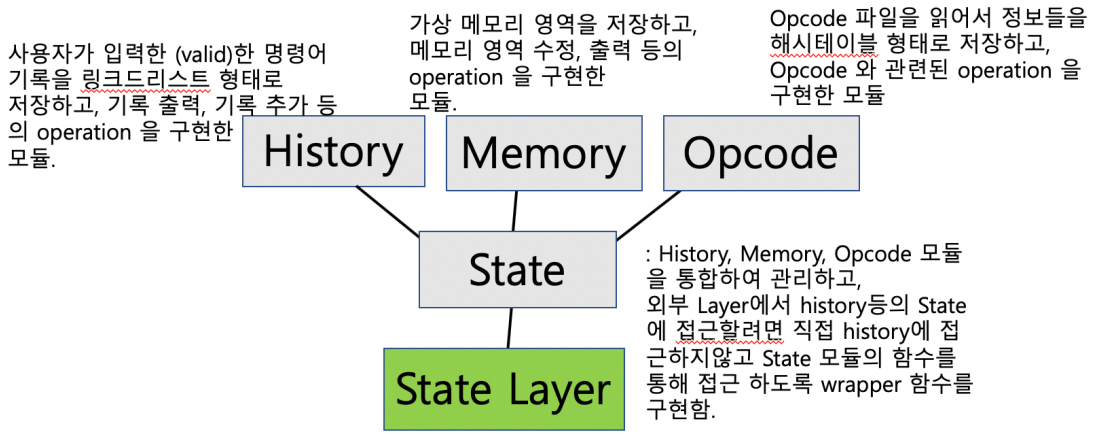

그림 2> State Layer 구조도

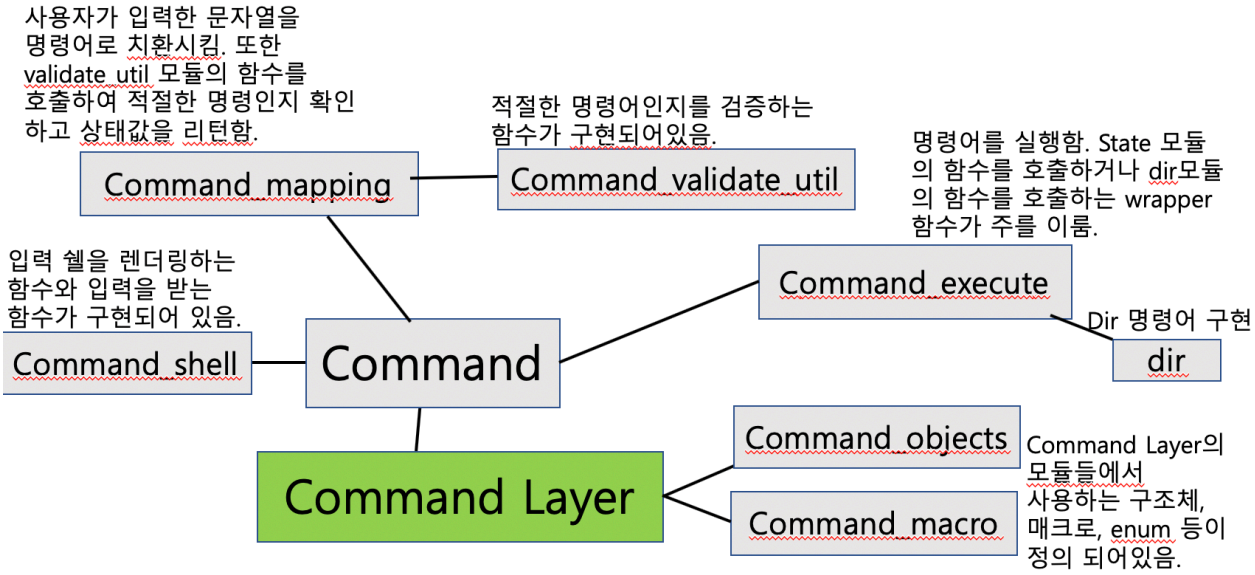

그림 3> Command Layer 구조도

3. 모듈 정의

3.1 모듈 이름 : Main Layer(main())

3.1.1 기능

construct_state() 함수를 호출하여 state_store 를 생성/초기화하고 command 모듈의 command_main() 함수에 state_store 를 파라미터로 전달하면서 실행시킴.
quit(q) 명령어로 인해 쉘이 끝나면 Main 으로 돌아와서 메모리를 전부 해제시키고 프로그램을 종료함

3.1.2 사용 변수

State* state_store	History, opcode, Memory 등의 정보를 저장함

3.2 모듈 이름: Command Layer

3.2.1 기능

사용자로 부터 입력을 받는 모듈, 입력을 검증하고 명령어로 매핑하는 모듈, 명령어를 실행하는 모듈 등으로 이루어졌다.

3.2.2 사용 변수

없음

3.3 모듈이름: Command Layer - command 모듈

3.3.1 기능

command layer 의 모듈들을 통합하는 역활을 하며 command 모듈의 함수들의 기능은 아래와 같다.

bool command_main(State* state_store)	종료 조건이 발생하기 이전까지 1~4 까지의 과정을 반복한다. 또한 1~4 과정을 순차적으로 실행하면서 만일 검증 함수에 의해서 에러 상태를 받게되면 이를 핸들링하고 continue 하여 다시 1 과정으로 돌아간다. 1. read_input() 함수를 호출하여 사용자로 부터 입력을 받아서 Command 구조체에 저장한다. / 검증 함수 호출 2. command_mapping 함수를 호출하여 입력 받은 문자열을 명령어로 매핑한다. / 검증 함수 호출 3. command_execute 함수를 호출하여 명령어를 수행한다. / 검증 함수 호출 4. add_history 함수를 호출하여 명령어 기록을 저장한다.
bool exception_check_and_handling(shell_status status);	Command Layer 의 몇몇 함수들은 입력 검증이 성공했는지 실패했는지 상태를 공유하기위해 리턴 값으로 shell_status 라는 enum을 리턴한다. 이 함수에서는 이 enum 값에 따라서 에러메세지를 출력하고
bool check_quit_condition(Command* user_command)	Command 로 quit(q) 명령이 들어왔다면 true 를 리턴하고, 아니라면 false 를 리턴한다.

3.3.2 사용변수

없음

3.4 모듈이름: Command Layer - command_mapping 모듈

3.4.1 기능

사용자가 입력한 문자열을 명령어로 치환시키기 위한 함수들로 구성되어있음

shell_status command_mapping(Command* user_command)	사용자의 raw_input 이 적절한 명령어인지 확인하고, 적절하다면 지정된 명령어로 매핑하고, 성공했다는 shell_status 를 리턴한다. 적절하지않다면, 실패했다는 shell_status 를 리턴한다. 프로그램은 아래와 같은 순서로 진행된다. 1. 토크나이징 2. 명령어 타입 체크 / 검증 3. 파라미터 검증 4. 성공/실패 여부 리턴
shell_status tokenizing(Command* user_command)	사용자의 raw_input 을 토크나이징하여 user_command->tokens 에 저장한다. ex) 입력이 dump 1, 2 가 들어왔다면 tokens[0] = "dump" tokens[1] = "1" tokens[2] = "2" 형태로 저장된다.
shell_status command_mapping_type(Command *user_command)	user_command->tokens[0]에 저장된 문자열이 적절한 명령어 타입인지 확인하고, user_command->type 에 명령어 타입을 설정해준다.

3.4.2 사용변수

없음

3.5 모듈이름: Command Layer - command_validate_util 모듈

3.5.1 기능

명령어 검증과 관련한 함수들로 이루어졌다.

bool validate_tokenizing(char *str, int token_cnt, int max_token_num);	토크나이징이 적절하게 되었는지 검증한다. 예를들어 “du , 1 1” 과 같이 파라미터 사이에 콤마가 없거나 이상한 위치에 콤마가 있는 등의 문제를 잡아낸다.
shell_status validate_parameters(Command *user_command)	사용자가 입력한 파라미터가 적절한 파라미터 값인지 검증한다. (파라미터 개수, 크기, 범위 등)
shell_status validate_dump_parameters(Command *user_command)	Dump 명령어의 파라미터를 검증한다. 파라미터가 적절한 주소값인지, 범위를 넘어가지는 않는지 등을 체크한다.

shell_status validate_opcode_parameters(Command *user_command)	Opcode 명령어의 파라미터를 검증한다.
shell_status validate_edit_parameters(Command *user_command)	Edit 명령어의 파라미터를 검증한다.
shell_status validate_fill_parameters(Command *user_command)	Fill 명령어의 파라미터를 검증한다.

3.5.2 사용변수

없음

3.6 모듈이름: Command Layer - command_objects 모듈

3.6.1. 기능

Command Layer 에서 사용하는 enum, 구조체 타입을 정의 해두었다.

enum shell_command_type {
TYPE_HELP, TYPE_DIR, TYPE_QUIT, TYPE_HISTORY,
TYPE_DUMP, TYPE_EDIT, TYPE_FILL, TYPE_RESET,
TYPE_OPCODE, TYPE_OPCODELIST
}
	명령어 타입을 enum 형태로 표현한다.

Struct command {
char* raw_command;
char* tokens[TOKEN_MAX_NUM + 5];
size_t token_cnt;
enum shell_command_type type;
}
	Command 정보를 저장한다.
	Typedef 를 통해 Command 타입으로 설정하였다.

enum SHELL_STATUS { ….. }	쉘의 상태를 enum 으로 표현한다.

3.6.2. 사용변수

없음

3.7 모듈이름: Command Layer - command_execute 모듈

3.7.1. 기능

Command 구조체에 저장된 명령어를 실행한다. 이 모듈의 함수를 실행하는 시점은 모든 입력 검증이 완료된 시점이다.

shell_status command_execute(Command *user_command, State *state_store);	사용자가 입력한 명령어에 따른 실행 함수(execute_***())를 실행한다.
void execute_help()	Help 명령어를 실행한다. 사용할수있는 명령어 목록을 출력해서 보여준다
shell_status execute_history(State* state_store, **char** *last_command)	History 명령어를 실행한다. 명령어 기록을 출력해서 보여준다. State 모듈의 print_histories_state 함수를 호출하는 wrapper function 형태이다.
shell_status execute_quit()	QUIT 이라는 shell_status enum 값을 리턴한다.
shell_status execute_dir()	Dir 명령어를 실행한다. 실행 파일이 위치한 폴더에 있는 파일과 폴더들을 출력해서 보여준다. Dir 모듈의 print_dir 함수를 호출하는 wrapper function 형태이다.
shell_status execute_dump(Command *user_command, Memories *memories_state)	Dump 명령어를 실행한다. 사용자가 입력한 파라미터에 따라서 출력할 영역을 지정하고 memory 모듈의 print_memories 함수를 호출하고, 마지막 주소값을 저장한다.
shell_status execute_edit(Command *user_command, Memories *memories_state)	Edit 명령어를 실행한다. 파라미터를 strtol 함수를 이용하여 숫자로 변환하고 edit_memoy 함수를 호출한다.
shell_status execute_fill(Command *user_command, Memories *memories_state)	Fill 명령어를 실행한다. 파라미터를 strtol 함수를 이용하여 숫자로 변환하고, 반복문을 돌면서 파라미터로 넘어온 영역의 메모리 주소들의 값을 수정한다.
shell_status execute_reset(Memories *memories_state)	Reset 명령어를 실행한다. 가상 메모리 영역의 모든 값들을 0 으로 바꾼다.
shell_status execute_opcode(Command *user_command, State* state_store)	Opcode 명령어를 실행한다. find_opcode_by_name 함수를 호출하고 NULL이라면 에러를 출력하고, 찾는데에 성공하였다면 opcode 값을 출력한다.
shell_status execute_opcodelist(State* state_store)	Opcodelist 명령어를 실행한다. Opcode 모듈의 print_opcodes 함수를 호출하는 wrapper function 형태이다.

3.7.2. 사용변수

없음

3.8 모듈이름: Command Layer - command_shell 모듈

3.8.1. 기능

Command 구조체에 저장된 명령어를 실행한다. 이 모듈의 함수를 실행하는 시점은 모든 입력 검증이 완료된 시점이다.

shell_status read_input(char** target)	사용자로 부터 입력을 받고 *target 에 저장한다. 너무 길 경우 에러 상태를 리턴해준다.
void render_shell()	쉘을 출력해서 보여준다.

3.8.2. 사용변수

없음

3.9 모듈이름: Command Layer - dir 모듈

3.9.1. 기능

Command 구조체에 저장된 명령어를 실행한다. 이 모듈의 함수를 실행하는 시점은 모든 입력 검증이 완료된 시점이다.

bool print_dir()	실행 파일이 위치한 폴더에 있는 파일들과 폴더들을 출력한다.

3.9.2. 사용변수

없음

3.10 모듈이름: Command Layer - command_macro 모듈

3.10.1. 기능

#define 매크로를 선언하였다.

```
#define COMPARE_STRING(T, S) (strcmp ((T), (S)) == 0)

#define COMMAND_MAX_LEN 510

#define TOKEN_MAX_NUM 30
```

3.9.2. 사용변수

없음

3.11 모듈이름: State Layer

3.11.1. 기능

명령어 기록, 메모리 , opcode 등의 정보를 관리하는 Layer 이다.

3.11.2. 사용변수

없음

3.12 모듈이름: State Layer – state 모듈

3.12.1. 기능

History, memory, opcode 모듈을 통합하는 State 구조체와 관련한 정의와 함수들로 이루어졌다.

State* construct_state()	History, 가상 Memory, Opcode 정보가 초기화(및 저장)된 State* 을 리턴한다
bool destroy_state(State **state_store)	*state_store 가 할당한 모든 메모리를 해제한다
bool add_history(State *state_store, char* history_str)	History_str 문자열을 명령어 히스토리에 기록한다.
void print_histories_state(State* state_store, **char*** last_command)	명령어 히스토리를 출력한다.

struct state{
Histories* histories_state;
Memories* memories_state;
OpcodeTable* opcode_table_state;
}
	History, memory, opcode 모듈을 통합하여 관리한다.
	또한 이 구조체는 State 라는 이름으로 typedef 되었다.

3.12.2. 사용변수

없음

3.13 모듈이름: State Layer – history 모듈

3.13.1. 기능

명령어 히스토리를 저장하기 위한 구조체와 관련된 함수들로 구성되었다.

struct history {
 	
  char value[HISTORY_MAX_LEN];
 	
};
	히스토리에 저장될 하나의 명령어 문자열을 wrapping 해서 저장함.
	(Typedef History)

struct history_node{
 	
  History* data;
 	
  struct history_node* prev;
 	
  struct history_node* next
;
};
	history_list 구조체의 Node 역활을 한다.
	(typedef HistoryNode)

struct history_list {
 	
  struct history_node* head;
 	
  struct history_node* tail;
 	
  int size;
 	
}
	링크드리스트 형태로 히스토리를 저장한다.
	(typedef HistoryList)

struct histories {
 	
  int size;
 	
  HistoryList* list;
 	
}
	History_list 구조체의 wrapper function
	(typedef Histories)

Histories* construct_histories()	Histories 구조체를 초기화/생성하여 리턴한다.
History* construct_history()	History 구조체를 초기화/생성하여 리턴한다.
History* construct_history_with_string(**char*** str)	Str 이라는 문자열로 초기화된 History 구조체를 리턴한다.
bool push_history(Histories* histories_store, History* target)	Histories_state의 링크드리스트에 target 히스토리를 추가한다.
void print_history(Histories *histories_store, **char** *last_command)	Histories 에 저장된 명령어 히스토리를 출력한다.

bool destroy_histories(Histories **histories_state)	Histories 구조체를 생성하면서 할당했던 모든 메모리를 해제한다.

3.13.2. 사용변수

없음

3.14 모듈이름: State Layer – Memory 모듈

3.14.1. 기능

가상 메모리 영역을 구현하기 위한 함수들과 구조체로 이루어짐

struct memory {
 	
short value;
 	
}
	메모리 값을 wrapping 한 구조체
	(typedef Memory)

struct memories {
 	
Memory data[MEMORIES_SIZE]; // (1024 * 1024)
 	
int last_idx;
 	
}
	1024*1024 만큼의 크기의 배열을 저장하고,
	Dump 명령어의 마지막 주소값을 저장할 변수로 이루어졌다.
	(typedef Memories)

Memories* construct_memories()	Memories 구조체 생성/초기화 한후 리턴
void print_memories(Memories* memories_state, int start, int end)	Start ~ End 메모리 영역을 출력한다. Start, end 는 10 진수로 변환한 값이다.
void edit_memory(Memories* memories_state, int address, short value)	Address 인덱스의 메모리 값을 value 로 수정한다.
bool destroy_memories(Memories** memories_state)	*memories_state 를 생성하면서 할당했던 메모리를 모두 해제한다.

3.14.2. 사용변수

없음

3.15 모듈이름: State Layer – Opcode 모듈

3.15.1. 기능

Opcode 관련 기능들을 구현한 모듈

struct opcode {
 	
enum op_format format;
 	
enum op_mnemonic mnemonic;
 	
char mnemonic_name[10];
 	
int value;
 	
}
	Opcode 를 구조화해서 저장한다.
	(typedef Opcode)

struct op_node{
 	
Opcode* data;
 	
struct op_node* prev;
 	
struct op_node* next;
 	
}
	op_linked_list 의 Node 역활을 한다.
	(typedef OpNode)

struct op_linked_list {
 	
struct op_node* head;
 	
struct op_node* tail;
 	
int size;
 	
}
	Opcode Table 구현을 위한 링크드 리스트 구조체.
	(typedef OpLinkedList)

struct opcode_table {
 	
OpLinkedList** list;
 	
int size;
 	
}
	해시테이블 형태로 구현된 Opcode Table 이다.
	(typedef OpcodeTable)

Opcode* construct_opcode()	Opcode 구조체를 생성/초기화 하여 리턴한다.
struct op_node* construct_opnode()	Op_node 구조체를 생성하여 리턴한다
OpcodeTable* construct_opcode_table()	OpcodeTable 구조체를 생성/초기화 하여 리턴한다.
bool build_opcode_table(OpcodeTable* table)	Opcode 파일을 읽어서 OpcodeTable 에 해시테이블 형태로 저장한다.

bool destroy_opcode_table(OpcodeTable** table)	*OpcodeTable을 생성하면서 할당했던 모든 메모리를 해제한다.
bool insert_opcode(OpcodeTable* table, Opcode* opc)	OpcodeTable 에 opc 라는 Opcode 값을 추가한다.
Opcode* find_opcode_by_name(OpcodeTable* table, **char*** name)	OpcodeTable에서 mnemonic 값을 기준으로 Opcode 를 검색한다.
void print_opcodes(OpcodeTable* table)	모든 Opcode 목록을 해시테이블에 저장된 형태로 출력한다.

3.15.2. 사용변수

없음

3.16 모듈이름: util 모듈

3.16.1. 기능

재사용성이 높은 함수/매크로들로 이루어졌음

#define COMPARE_STRING(T, S) (strcmp ((T), (S)) == 0)	문자열 비교 매크로
size_t hash_string (**char** *str, **int** hash_size);	해시테이블 구현을 위한, hash function
bool is_zero_str(**char*** str);	문자열이 0으로 변환될수있는지 확인한다
bool is_valid_hex(**char*** str);	문자열이 16진수로 변환될수있는지 확인한다.
bool is_valid_address(**char** *str, **int** max_size);	문자열이 주소값으로 표현될수있는지 확인한다

3.16.2. 사용변수

없음

4. 전역 변수 정의

전역변수 정의 하지않았음.

5. 코드

5.1 20161631.c

```
#include "20161631.h"

int main() {
   /*
    * state_store 에서는 명령어 히스토리, 가상 메모리 영역, Opcode 정보를 저장한다.
    */
   State* state_store = construct_state();

   /*
    * 사용자가 quit(q) 명령을 입력하기전까지
    * 쉘을 통해 명령어를 입력, 수행할수있도록 한다.
    */
   command_main(state_store);

   /*
    * 동적 할당 받은 메모리를 모두 해제한다.
    */
   destroy_state(&state_store);

   return 0;
}
```

5.2 20161631.h

```
#ifndef __20161631_H__
#define __20161631_H__

#include <stdio.h>
#include <dirent.h>
#include <sys/stat.h>
#include "command.h"
#include "state.h"
#include "dir.h"

#endif
```

5.3 command.c

```
#include "command.h"

/*
 * 사용자가 quit(q)를 명령을 입력하기 이전까지 쉘을 계속 수행한다.
 */
bool command_main(State* state_store){
   shell_status status;
   Command user_command;

   while (1){

      render_shell();

      // 사용자로 부터 입력을 받는다.
```

```
        status = read_input(&user_command.raw_command);

        // 잘못된 입력이라면 continue 한다.
        if(!exception_check_and_handling(status)) continue;

        // 입력을 지정된 명령어에 있는지 확인하고, 매핑한다.
        status = command_mapping(&user_command);
        if(!exception_check_and_handling(status)) continue;

        // 매핑된 명령어를 수행한다.
        status = command_execute(&user_command, state_store);
        if(check_quit_condition(&user_command)) break;
        if(!exception_check_and_handling(status)) continue;
        if(status == EXECUTE_FAIL) continue;

        // 실행이 완료된 명령어를 입력 그대로 히스토리에 추가한다.
        add_history(state_store, user_command.raw_command);
    }
    return true;
}

/*
 * status 파라미터에 넘어온 내용에 따라서
 * 에러에 해당한다면 적절한 에러문을 출력해주고 false 를 리턴한다.
 * 에러에 해당하지않는다면 true 를 리턴한다.
 *
 * 참고: 사용자에게 입력을 받거나, 토크나이징 하는 등의 함수들은
 *      성공, 실패 여부등에 따라서 shell_status ( enum )을 리턴한다.
 */
bool exception_check_and_handling(shell_status status){
    switch(status){
        case INPUT_READ_SUCCESS:
            break;
        case TOKENIZING_SUCCESS:
            break;
        case VALID_COMMAND_TYPE:
            break;
        case VALID_PARAMETERS:
            break;
        case EXECUTE_SUCCESS:
            break;
        case TOO_LONG_WRONG_INPUT:
            fprintf(stderr, "[ERROR] Too Long Input\n");
            return false;
        case TOO_MANY_TOKEN:
            fprintf (stderr, "[ERROR] Too Many Tokens\n");
            return false;
        case INVALID_COMMAND_TYPE:
            fprintf(stderr, "[ERROR] Invalid Command Type\n");
            return false;
        case INVALID_INPUT:
            fprintf(stderr, "[ERROR] Invalid Input\n");
```

```
            return false;
        case INVALID_PARAMETERS:
            fprintf(stderr, "[ERROR] Invalid Parameters\n");
            return false;
        case MISSING_REQUIRE_PARAMETER:
            fprintf(stderr, "[ERROR] Missing Required Parameter\n");
            return false;
        case EXECUTE_FAIL:
            fprintf(stderr, "[ERROR] Invalid Input\n");
            break;
        default:
            break;
    }
    return true;
}

/*
 * quit(q) 명령이 들어왔다면 true 를 리턴하고,
 * 아니라면 false 를 리턴한다.
 */
bool check_quit_condition(Command* user_command){
    if(user_command->type == TYPE_QUIT){
        assert(user_command->token_cnt == 1);
        return true;
    }
    else{
        return false;
    }
}
```

5.4 command.h

```
#ifndef __COMMAND_H__
#define __COMMAND_H__


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <stdbool.h>

#include "command_macro.h"
#include "command_objects.h"
#include "command_shell.h"
#include "command_mapping.h"
#include "command_execute.h"
#include "state.h"

/*
 * 사용자가 quit(q)를 명령을 입력하기 이전까지 쉘을 계속 수행한다.
 */
bool command_main(State* state_store);
```

```
/*
 * status 파라미터에 넘어온 내용에 따라서
 * 에러에 해당한다면 적절한 에러문을 출력해주고 false 를 리턴한다.
 * 에러에 해당하지않는다면 true 를 리턴한다.
 *
 * 참고: 사용자에게 입력을 받거나, 토크나이징 하는 등의 함수들은
 *      성공, 실패 여부등에 따라서 shell_status ( enum )을 리턴한다.
 */
bool exception_check_and_handling(shell_status status);

/*
 * quit(q) 명령이 들어왔다면 true 를 리턴하고,
 * 아니라면 false 를 리턴한다.
 */
bool check_quit_condition(Command* user_command);

#endif
```

5.5 command_execute.c

```
#include "command_execute.h"

/*
 * 사용자가 입력한 명령어에 따른 실행 함수(execute_***())를 실행한다.
 * 또한 실행된 결과를 리턴해준다.
 */
shell_status command_execute(Command *user_command, State *state_store) {
   assert(user_command);
   assert(state_store);

   switch (user_command->type){
      case TYPE_HELP:
         execute_help();
         return EXECUTE_SUCCESS;
      case TYPE_HISTORY:
         return execute_history(state_store, user_command->raw_command);
      case TYPE_QUIT:
         return execute_quit();
      case TYPE_DIR:
         return execute_dir();
      case TYPE_EDIT:
         return execute_edit(user_command, state_store->memories_state);
      case TYPE_FILL:
         return execute_fill(user_command, state_store->memories_state);
      case TYPE_RESET:
         return execute_reset(state_store->memories_state);
      case TYPE_OPCODE:
         return execute_opcode(user_command, state_store);
      case TYPE_OPCODELIST:
         return execute_opcodelist(state_store);
      case TYPE_DUMP:
         return execute_dump(user_command, state_store->memories_state);
      default:
```

```
            break;
    }
    return EXECUTE_SUCCESS;
}

/*
 * history 명령어
 * 실행되었던 명령어 히스토리를 출력한다
 */
shell_status execute_history(State* state_store, char *last_command) {
    assert(state_store);
    assert(last_command);

    print_histories_state(state_store, last_command);

    return EXECUTE_SUCCESS;
}

/*
 * help 명령어
 * 사용할수있는 명령어들을 화면에 출력해서 보여준다,
 */
void execute_help(){
    fprintf (stdout,"h[elp]\n"
            "d[ir]\n"
            "q[uit]\n"
            "hi[story]\n"
            "du[mp] [start, end]\n"
            "e[dit] address, value\n"
            "f[ill] start, end, value\n"
            "reset\n"
            "opcode mnemonic\n"
            "opcodelist\n");
}

/*
 * quit 명령어
 * QUIT 이라는 shell_status(enum)을 리턴한다.
 *
 * 참고: command_main 함수의 무한 루프는 QUIT 이라는 status 가 들어오면
 *      break 되도록 설계되었음.
 */
shell_status execute_quit(){
    fprintf(stdout, "Bye :)\n");

    return QUIT;
}

/*
 * dir 명령어
 * 실행 파일이 위치한 폴더에 있는 파일들과 폴더들을 출력한다.
 */
```

```
shell_status execute_dir(){
   if(!print_dir()) return EXECUTE_FAIL;
   return EXECUTE_SUCCESS;
}

/*
 * dump 명령어
 * dump start, end : start~end 까지의 가상 메모리영역을 출력한다.
 * dump start      : start 메모리 부터 10 라인의 영역을 출력한다.
 * dump            : 가장 마지막으로 실행되었던 메모리부터 10 라인의 영역 출력한다.
 */
shell_status execute_dump(Command *user_command, Memories *memories_state) {
   assert(memories_state);
   assert(user_command);
   assert(user_command->token_cnt < 4);

   size_t token_cnt = user_command->token_cnt;
   int start=0, end=0;

   // 출력할 메모리 영역의 범위를 초기화 한다.
   if(token_cnt == 1){
      // case1. dump

      start = memories_state->last_idx + 1;
      end = start + 159;
   } else if(token_cnt == 2){
      // case2. dump start

      start = (int)strtol(user_command->tokens[1], NULL, 16);
      if(start + 159 >= MEMORIES_SIZE) end = MEMORIES_SIZE - 1;
      else end = start + 159;
   } else if(token_cnt == 3){
      // case3. dump start, end

      start = (int)strtol(user_command->tokens[1], NULL, 16);
      end = (int)strtol(user_command->tokens[2], NULL, 16);
      if(end >= MEMORIES_SIZE) end = MEMORIES_SIZE - 1;
   }

   // start ~ end 범위의 메모리 영역을 출력한다.
   print_memories(memories_state, start, end);

   // 출력된 마지막 메모리 주소를 저장한다.
   // 만일 마지막 메모리 주소에 1 을 더한 주소가 범위를 벗어났다면 -1 을 저장한다.
   if(end + 1 >= MEMORIES_SIZE)
      memories_state->last_idx = -1;
   else
      memories_state->last_idx = end;

   return EXECUTE_SUCCESS;
}
```

```
/*
 * edit 명령어
 * edit addr, value: addr 주소의 값을 value 로 수정한다.
 */
shell_status execute_edit(Command *user_command, Memories *memories_state) {
   assert(user_command);
   assert(memories_state);
   assert(user_command->token_cnt == 3);

   int addr = (int)strtol(user_command->tokens[1], NULL, 16);
   short value = (short)strtol(user_command->tokens[2], NULL, 16);

   edit_memory(memories_state, addr, value);

   return EXECUTE_SUCCESS;
}

/*
 * fill 명령어
 * fill start, end, value: start~end 의 메모리 영역의 값들을 value 로 수정한다.
 */
shell_status execute_fill(Command *user_command, Memories *memories_state) {
   assert(user_command);
   assert(memories_state);
   assert(user_command->token_cnt == 4);
   int start = (int)strtol(user_command->tokens[1], NULL, 16);
   int end = (int)strtol(user_command->tokens[2], NULL, 16);
   short value = (short)strtol(user_command->tokens[3], NULL, 16);
   int addr = 0;
   assert(start <= end || start >= 0 || end >= 0 || value >= 0);

   for(addr = start; addr <= end; addr++)
      edit_memory(memories_state, addr, value);

   return EXECUTE_SUCCESS;
}

/*
 * reset 명령어
 * 가상 메모리 영역의 모든 값들을 0 으로 바꾼다.
 */
shell_status execute_reset(Memories *memories_state) {
   assert(memories_state);

   int addr = 0, end = MEMORIES_SIZE - 1;
   for(addr=0;addr<=end;addr++)
      edit_memory(memories_state, addr, 0);

   return EXECUTE_SUCCESS;
}

/*
```

```
 * opcode 명령어
 * opcode mnemonic : mnemonic 의 value 를 출력
 * ex) opcode LDF => opcode is 70
 */
shell_status execute_opcode(Command* user_command, State* state_store){
   assert(user_command->token_cnt == 2);
   Opcode* opc = find_opcode_by_name(state_store->opcode_table_state, user_command->tokens[1]);

   if(!opc) return EXECUTE_FAIL;

   fprintf(stdout, "opcode is %X\n", opc->value);

   return EXECUTE_SUCCESS;
}

/*
 * opcodelist 명령어
 * 해시테이블 형태로 저장된 opcode 목록을 출력해준다.
 */
shell_status execute_opcodelist(State* state_store){
   print_opcodes(state_store->opcode_table_state);
   return EXECUTE_SUCCESS;
}
```

5.6 command_execute.h

```
#ifndef __COMMAND_EXECUTE_H__
#define __COMMAND_EXECUTE_H__

#include "command_macro.h"
#include "command_objects.h"
#include "state.h"
#include "dir.h"

/*
 * 사용자가 입력한 명령어에 따른 실행 함수(execute_***())를 실행한다.
 * 또한 실행된 결과를 리턴해준다.
 */
shell_status command_execute(Command *user_command, State *state_store);

/*
 * help 명령어
 * 사용할수있는 명령어들을 화면에 출력해서 보여준다,
 */
void execute_help();

/*
 * history 명령어
 * 실행되었던 명령어 히스토리를 출력한다
 */
shell_status execute_history(State* state_store, char *last_command);

/*
```

```
 * quit 명령어
 * QUIT 이라는 shell_status(enum)을 리턴한다.
 *
 * 참고: command_main 함수의 무한 루프는 QUIT 이라는 status 가 들어오면
 *      break 되도록 설계되었음.
 */
shell_status execute_quit();

/*
 * dir 명령어
 * 실행 파일이 위치한 폴더에 있는 파일들과 폴더들을 출력한다.
 */
shell_status execute_dir();

/*
 * dump 명령어
 * dump start, end : start~end 까지의 가상 메모리영역을 출력한다.
 * dump start      : start 메모리 부터 10 라인의 영역을 출력한다.
 * dump            : 가장 마지막으로 실행되었던 메모리부터 10 라인의 영역 출력한다.
 */
shell_status execute_dump(Command *user_command, Memories *memories_state);

/*
 * edit 명령어
 * edit addr, value: addr 주소의 값을 value 로 수정한다.
 */
shell_status execute_edit(Command *user_command, Memories *memories_state);

/*
 * fill 명령어
 * fill start, end, value: start~end 의 메모리 영역의 값들을 value 로 수정한다.
 */
shell_status execute_fill(Command *user_command, Memories *memories_state);

/*
 * reset 명령어
 * 가상 메모리 영역의 모든 값들을 0 으로 바꾼다.
 */
shell_status execute_reset(Memories *memories_state);

/*
 * opcode 명령어
 * opcode mnemonic : mnemonic 의 value 를 출력
 * ex) opcode LDF => opcode is 70
 */
shell_status execute_opcode(Command *user_command, State* state_store);

/*
 * opcodelist 명령어
 * 해시테이블 형태로 저장된 opcode 목록을 출력해준다.
 */
```

```
shell_status execute_opcodelist(State* state_store);

#endif
```

5.7 command_macro.h

```
#ifndef __COMMAND_MACRO_H__
#define __COMMAND_MACRO_H__

#define COMPARE_STRING(T, S) (strcmp ((T), (S)) == 0)
#define COMMAND_MAX_LEN 510
#define TOKEN_MAX_NUM 30

#endif
```

5.8 command_mapping.c

```
#include "command_mapping.h"

/*
 * 사용자의 raw_input 이 적절한 명령어인지 확인하고,
 * 적절하다면 지정된 명령어로 매핑하고, 성공했다는 shell_status 를 리턴한다.
 * 적절하지않다면, 실패했다는 shell_status 를 리턴한다.
 */
shell_status command_mapping(Command* user_command){
   assert(user_command);
   shell_status status;

   status = tokenizing(user_command);
   if(status != TOKENIZING_SUCCESS) return status;

   status = command_mapping_type(user_command);
   if(!validate_tokenizing(user_command->raw_command,
                     (int) user_command->token_cnt,
                     TOKEN_MAX_NUM))
      return INVALID_INPUT;
   if(status != VALID_COMMAND_TYPE) return status;

   // 파라미터가 해당 명령어에 적절한지 확인한다.
   // 예를들어 dump -1 과 같은 경우를 잡아낸다.
   status = validate_parameters(user_command);
   if(status != VALID_PARAMETERS) return status;

   return status;
}

/*
 * 사용자의 raw_input 을 토크나이징하여 user_command->tokens 에 저장한다.
 * ex) 입력이 dump 1, 2 가 들어왔다면
 *     tokens[0] = "dump"
 *     tokens[1] = "1"
 *     tokens[2] = "2"
 *     형태로 저장된다.
 *
```

```
 * @return TOKENIZING_SUCCESS or INVALID_INPUT
 */
shell_status tokenizing(Command* user_command){
   assert(user_command);
   static char raw_command[COMMAND_MAX_LEN];

   strncpy (raw_command, user_command->raw_command, COMMAND_MAX_LEN);
   user_command->token_cnt = 0;
   user_command->tokens[user_command->token_cnt] = strtok(raw_command, " ,\t\n");
   while (user_command->token_cnt <= TOKEN_MAX_NUM && user_command->tokens[user_command-
>token_cnt])
      user_command->tokens[++user_command->token_cnt] = strtok (NULL, " ,\t\n");

   if(!user_command->tokens[0]) return INVALID_INPUT;

   return TOKENIZING_SUCCESS;
}

/*
 * user_command->tokens[0]에 저장된 문자열이
 * 적절한 명령어 타입인지 확인하고, user_command->type 에 명령어 타입을 설정해준다.
 *
 * @return VALID_COMMAND_TYPE or INVALID_COMMAND_TYPE
 */
shell_status command_mapping_type(Command *user_command){
   assert(user_command);
   char* first_token = user_command->tokens[0];
   if(COMPARE_STRING(first_token, "h") ||
     COMPARE_STRING(first_token, "help")) {
      user_command->type = TYPE_HELP;
   }else if(COMPARE_STRING(first_token, "d") ||
         COMPARE_STRING(first_token, "dir")) {
      user_command->type = TYPE_DIR;
   }else if(COMPARE_STRING(first_token, "q") ||
         COMPARE_STRING(first_token, "quit")) {
      user_command->type = TYPE_QUIT;
   } else if(COMPARE_STRING(first_token, "hi") ||
         COMPARE_STRING(first_token, "history")){
      user_command->type = TYPE_HISTORY;
   } else if(COMPARE_STRING(first_token, "du") ||
         COMPARE_STRING(first_token, "dump")){
      user_command->type = TYPE_DUMP;
   } else if(COMPARE_STRING(first_token, "e") ||
         COMPARE_STRING(first_token, "edit")){
      user_command->type = TYPE_EDIT;
   } else if(COMPARE_STRING(first_token, "f") ||
         COMPARE_STRING(first_token, "fill")){
      user_command->type = TYPE_FILL;
   } else if(COMPARE_STRING(first_token, "reset")){
      user_command->type = TYPE_RESET;
   } else if(COMPARE_STRING(first_token, "opcode")){
      user_command->type = TYPE_OPCODE;
```

```
    } else if(COMPARE_STRING(first_token, "opcodelist")){
        user_command->type = TYPE_OPCODELIST;
    } else {
        return INVALID_COMMAND_TYPE;
    }
    return VALID_COMMAND_TYPE;
}
```

5.9 command_mapping.h

```
#ifndef __COMMAND_MAPPING_H__
#define __COMMAND_MAPPING_H__

#include "command_macro.h"
#include "command_objects.h"
#include "command_validate_util.h"
#include "util.h"
#include <string.h>
#include <stdbool.h>
#include <assert.h>
#include <sys/types.h>
#include <stdlib.h>

/*
 * 사용자의 raw_input 이 적절한 명령어인지 확인하고,
 * 적절하다면 지정된 명령어로 매핑하고, 성공했다는 shell_status 를 리턴한다.
 * 적절하지않다면, 실패했다는 shell_status 를 리턴한다.
 */
shell_status command_mapping(Command* user_command);

/*
 * 사용자의 raw_input 을 토크나이징하여 user_command->tokens 에 저장한다.
 * ex) 입력이 dump 1, 2 가 들어왔다면
 *     tokens[0] = "dump"
 *     tokens[1] = "1"
 *     tokens[2] = "2"
 *     형태로 저장된다.
 *
 * @return TOKENIZING_SUCCESS or INVALID_INPUT
 */
shell_status tokenizing(Command* user_command);

/*
 * user_command->tokens[0]에 저장된 문자열이
 * 적절한 명령어 타입인지 확인하고, user_command->type 에 명령어 타입을 설정해준다.
 *
 * @return VALID_COMMAND_TYPE or INVALID_COMMAND_TYPE
 */
shell_status command_mapping_type(Command *user_command);

#endif
```

5.10 command_objects.h

```
#ifndef __COMMAND_OBJECTS_H__
#define __COMMAND_OBJECTS_H__
#include <sys/types.h>
#define TOKEN_MAX_NUM 30

// 명령어 타입을 enum 형태로 표현한다.
enum shell_command_type{
   TYPE_HELP, TYPE_DIR, TYPE_QUIT, TYPE_HISTORY,
   TYPE_DUMP, TYPE_EDIT, TYPE_FILL, TYPE_RESET,
   TYPE_OPCODE, TYPE_OPCODELIST
};

// command 를 구조체로 구조화 하여 표현.
typedef struct command {
   char* raw_command;
   char* tokens[TOKEN_MAX_NUM + 5];
   size_t token_cnt;
   enum shell_command_type type;
} Command;

// shell 의 상태를 표현함.
typedef enum SHELL_STATUS {
   INPUT_READ_SUCCESS, TOO_LONG_WRONG_INPUT,
   TOKENIZING_SUCCESS, TOO_MANY_TOKEN,
   INVALID_INPUT, VALID_COMMAND, INVALID_COMMAND,
   INVALID_COMMAND_TYPE, VALID_COMMAND_TYPE,
   INVALID_PARAMETERS, VALID_PARAMETERS,
   MISSING_REQUIRE_PARAMETER, EXECUTE_SUCCESS, QUIT, EXECUTE_FAIL
} shell_status;

#endif
```

5.11 command_shell.c

```
#include "command_shell.h"

/*
 * 사용자로 부터 입력을 받고 *target 에 저장한다.
 * 입력이 너무 길 경우 에러를 리턴해준다.
 *
 * @return INPUT_READ_SUCCESS or TOO_LONG_WRONG_INPUT
 */
shell_status read_input(char** target){
   static char input[COMMAND_MAX_LEN + 10];
   fgets(input, COMMAND_MAX_LEN + 10, stdin);
   if(strlen(input) >= COMMAND_MAX_LEN) return TOO_LONG_WRONG_INPUT;
   *target = input;
   return INPUT_READ_SUCCESS;
}

/*
 * shell 을 출력함.
```

```
 */
void render_shell(){
   printf("sicsim > ");
}
```

5.12 command_shell.h

```
#ifndef __COMMAND_SHELL_H__
#define __COMMAND_SHELL_H__

#include "command_macro.h"
#include "command_objects.h"
#include <string.h>
#include <stdio.h>

/*
 * 사용자로 부터 입력을 받고 *target 에 저장한다.
 * 입력이 너무 길 경우 에러를 리턴해준다.
 *
 * @return INPUT_READ_SUCCESS or TOO_LONG_WRONG_INPUT
 */
shell_status read_input(char** target);

/*
 * shell 을 출력함.
 */
void render_shell();

#endif
```

5.13 command_validate_util.c

```
#include "command_validate_util.h"

/*
 * 토크나이징이 적절하게 되었는지 검증한다.
 * 예를들어 du , 1 1 과 같이
 * 파라미터 사이에 콤마가 없거나 이상한 위치에 콤마가 있는 등의 문제를 잡아낸다.
 *
 * @return true or false
 */
bool validate_tokenizing(char *str, int token_cnt, int max_token_num) {
   assert(str);
   int length = (int)strlen(str);
   int i=0, comma_cnt = 0, flag = 0;
   char cm;

   // 토큰의 개수를 검증한다.
   if(token_cnt > max_token_num)
      return false;
   if(token_cnt <= 0)
      return false;

   // 콤마의 개수를 계산한다.
```

```
for(i=0;i<length; i++){
   if(str[i] == ',')
      comma_cnt++;
}

// 토큰의 개수가 두개이면서 콤마가 없는 경우는 적절한 경우다.
if(token_cnt <= 2 && comma_cnt == 0)
   return true;

// 예시와 같은 경우의 에러를 잡아낸다.
// ex)   du , 1 1 [x]
// ex)   , du
flag = 0;
for(i=0;i<length;i++){
   cm = str[i];
   if(flag == 0){
      // 첫번째 토큰 이전 문자열.
      if(cm == ',') return false;
      if(cm != ' ' && cm != '\t') flag = 1;
      continue;
   } else if(flag == 1){
      // 첫번째 토큰을 지나가는 중
      if(cm == ',') return false;
      if(cm == ' ' || cm == '\t') flag = 2;
      continue;
   }
   // 첫번째 토큰과 두번째 토큰 사이

   if(str[i] == ' ') continue;
   if(str[i] == '\t') continue;
   if(str[i] != ',') break;
   else return false;
}

// 예시와 같은 경우의 에러를 잡아낸다.
// ex) du 1 1 , [x]
for(i=length-2;i>=0;i--){
   if(str[i] == ' ') continue;
   if(str[i] == '\t') continue;
   if(str[i] != ',') break;
   else return false;
}

// 토큰의 개수와 콤마의 개수를 비교한다.
if(token_cnt != comma_cnt + 2)
   return false;


// ex) f 1 ,, 2 3 [x]
// , 다음에는 [ ] [\t]이 나오다가 [ ] [\t] [,]이 아닌 값이 나와야한다.
flag = 0; // 콤마가 나오면 flag 는 1 로 놓자.
for(i=0;i<length;i++) {
```

```
        cm = str[i];
        if (flag == 1) {
            if (cm == ',') return false;
            if (cm == ' ' || cm == '\t') continue;
            flag = 0;
            continue;
        }
        if (cm == ',') {
            flag = 1;
            continue;
        }
    }

    return true;
}

/*
 * 사용자가 입력한 파라미터가 적절한 파라미터 값인지 검증한다.
 * (명령어에 따른 파라미터 개수, 크기, 범위 등)
 *
 * @return VALID_PARAMETERS or INVALID_PARAMETERS
 */
shell_status validate_parameters(Command *user_command){
    assert(user_command);
    if((user_command->type == TYPE_QUIT ||
        user_command->type == TYPE_HELP ||
        user_command->type == TYPE_HISTORY ||
        user_command->type == TYPE_DIR ||
        user_command->type == TYPE_RESET ||
        user_command->type == TYPE_OPCODELIST) &&
       user_command->token_cnt > 1
          )
        return INVALID_PARAMETERS;
    if(user_command->type == TYPE_OPCODE)
        return validate_opcode_parameters(user_command);
    if(user_command->type == TYPE_EDIT)
        return validate_edit_parameters(user_command);

    if(user_command->type == TYPE_FILL)
        return validate_fill_parameters(user_command);
    if(user_command->type == TYPE_DUMP)
        return validate_dump_parameters(user_command);

    return VALID_PARAMETERS;
}

/*
 * dump 명령어의 파라미터를 검증한다.
 *
 * @return VALID_PARAMETERS or INVALID_PARAMETERS
 */
shell_status validate_dump_parameters(Command *user_command){
```

```
    assert(user_command);
    assert(user_command->type == TYPE_DUMP);
    int tok1, tok2;
    if(user_command->token_cnt > 1) {

        // 적절한 주소값인지 확인한다.
        if (!is_valid_address(user_command->tokens[1], MB))
            return INVALID_PARAMETERS;
        else if (user_command->token_cnt == 2)
            return VALID_PARAMETERS;

        // 적절한 주소값인지 확인한다.
        if (!is_valid_address(user_command->tokens[2], MB))
            return INVALID_PARAMETERS;

        tok1 = (int) strtol(user_command->tokens[1], NULL, 16);
        tok2 = (int) strtol(user_command->tokens[2], NULL, 16);

        // invalid area
        if (tok1 > tok2) return INVALID_PARAMETERS;
    }

    // dump 1, 2, 3 과 같이 파라미터가 세개 이상이 되는 경우는 에러이다.
    if(user_command->token_cnt > 3)
        return INVALID_PARAMETERS;

    return VALID_PARAMETERS;
}

/*
 * opcode 명령어의 파라미터를 검증한다.
 *
 * @return VALID_PARAMETERS or INVALID_PARAMETERS
 */
shell_status validate_opcode_parameters(Command *user_command){
    assert(user_command);
    assert(user_command->type == TYPE_OPCODE);
    if(user_command->token_cnt != 2) return INVALID_PARAMETERS;
    if(strlen(user_command->tokens[1]) > 14) return INVALID_PARAMETERS;

    return VALID_PARAMETERS;
}

/*
 * edit 명령어의 파라미터를 검증한다.
 *
 * @return VALID_PARAMETERS or INVALID_PARAMETERS
 */
shell_status validate_edit_parameters(Command *user_command){
    assert(user_command);
    assert(user_command->type == TYPE_EDIT);
    int value;
```

```
    if(user_command->token_cnt != 3)
        return INVALID_PARAMETERS;

    // 적절한 주소값인지 검증
    if(!is_valid_address(user_command->tokens[1], MB))
        return INVALID_PARAMETERS;

    // 적절한 16 진수 값인지 검증한다.
    if(!is_valid_hex(user_command->tokens[2]))
        return INVALID_PARAMETERS;

    value = (int) strtol(user_command->tokens[2], NULL, 16);
    // 범위 확인
    if(!(0 <= value && value <= 0xFF))
        return INVALID_PARAMETERS;

    return VALID_PARAMETERS;
}

/*
 * fill 명령어의 파라미터를 검증한다.
 *
 * @return VALID_PARAMETERS or INVALID_PARAMETERS
 */
shell_status validate_fill_parameters(Command *user_command){
    assert(user_command);
    assert(user_command->type == TYPE_FILL);
    int value, start, end;

    if(user_command->token_cnt != 4)
        return INVALID_PARAMETERS;
    if(!is_valid_address(user_command->tokens[1], MB))
        return INVALID_PARAMETERS;
    if(!is_valid_address(user_command->tokens[2], MB))
        return INVALID_PARAMETERS;

    start = (int) strtol(user_command->tokens[1], NULL, 16);
    end = (int) strtol(user_command->tokens[2], NULL, 16);
    if (start > end) return INVALID_PARAMETERS;

    value = (int) strtol(user_command->tokens[3], NULL, 16);
    if(!is_valid_hex(user_command->tokens[3]))
        return INVALID_PARAMETERS;
    if(!(0 <= value && value <= 0xFF)) return INVALID_PARAMETERS;

    return VALID_PARAMETERS;
}
```

5.14 command_validate_util.h

```
#ifndef __COMMAND_VALIDATE_UTIL_H__
#define __COMMAND_VALIDATE_UTIL_H__
```

```
#include <stdbool.h>
#include <string.h>
#include <assert.h>
#include "command_objects.h"
#include "util.h"
#define MB (1024*1024)

/*
 * 토크나이징이 적절하게 되었는지 검증한다.
 * 예를들어 du , 1 1 과 같이
 * 파라미터 사이에 콤마가 없거나 이상한 위치에 콤마가 있는 등의 문제를 잡아낸다.
 *
 * @return true or false
 */
bool validate_tokenizing(char *str, int token_cnt, int max_token_num);

/*
 * 사용자가 입력한 파라미터가 적절한 파라미터 값인지 검증한다.
 * (명령어에 따른 파라미터 개수, 크기, 범위 등)
 *
 * @return VALID_PARAMETERS or INVALID_PARAMETERS
 */
shell_status validate_parameters(Command *user_command);

/*
 * dump 명령어의 파라미터를 검증한다.
 *
 * @return VALID_PARAMETERS or INVALID_PARAMETERS
 */
shell_status validate_dump_parameters(Command *user_command);

/*
 * opcode 명령어의 파라미터를 검증한다.
 *
 * @return VALID_PARAMETERS or INVALID_PARAMETERS
 */
shell_status validate_opcode_parameters(Command *user_command);

/*
 * edit 명령어의 파라미터를 검증한다.
 *
 * @return VALID_PARAMETERS or INVALID_PARAMETERS
 */
shell_status validate_edit_parameters(Command *user_command);

/*
 * fill 명령어의 파라미터를 검증한다.
 *
 * @return VALID_PARAMETERS or INVALID_PARAMETERS
 */
shell_status validate_fill_parameters(Command *user_command);
```

```
#endif
```

5.15 dir.c

```
#include "dir.h"

/*
 * 실행 파일이 위치한 폴더에 있는 파일들과 폴더들을 출력한다.
 */
bool print_dir(){
   DIR* dir = opendir(".");
   struct dirent *ent;
   struct stat stat;
   char* ent_dname;
   char* format;
   char path[1025];
   int i = 0;

   if(!dir){
      fprintf(stderr, "[ERROR] Can't open directory");
      return false;
   }
   ent = readdir(dir);
   while (ent){
      ent_dname = ent->d_name;
      lstat(ent_dname, &stat);

      if(S_ISDIR(stat.st_mode)) format = "%s/";
      else if(S_IXUSR & stat.st_mode) format = "%s*";
      else format = "%s ";

      sprintf(path, format, ent->d_name);
      printf("%-20s", path);

      if(++i % 5 == 0) printf("\n");
      ent = readdir(dir);
   }
   if (i % 5 != 0) printf ("\n");
   closedir(dir);
   return true;
}
```

5.16 dir.h

```
#ifndef __DIR_H__
#define __DIR_H__

#include <stdio.h>
#include <dirent.h>
#include <sys/stat.h>
#include <stdbool.h>
#include <unistd.h>
#include <string.h>
```

```
/*
 * 실행 파일이 위치한 폴더에 있는 파일들과 폴더들을 출력한다.
 */
bool print_dir();

#endif
```

5.17 history.c

```
#include "history.h"

/*
 * Histories 구조체를 생성(할당)하고 HistoryList 도 생성(할당)한다.
 */
Histories* construct_histories(){
   Histories* hists = (Histories*)malloc(sizeof(*hists));

   hists->list = (HistoryList*)malloc(sizeof(HistoryList));
   hists->list->head = (HistoryNode*)malloc(sizeof(HistoryNode));
   hists->list->tail = (HistoryNode*)malloc(sizeof(HistoryNode));
   hists->list->head->prev = NULL;
   hists->list->head->next = hists->list->tail;
   hists->list->tail->prev = hists->list->head;
   hists->list->tail->next = NULL;

   hists->list->head->data = construct_history();
   hists->list->tail->data = construct_history();

   hists->size = 0;
   return hists;
}

/*
 * History 구조체를 생성(할당)한다.
 */
History* construct_history(){
   History* hist = (History*)malloc(sizeof(*hist));

   return hist;
}

/*
 * Histories 구조체를 생성하면서 할당했던 모든 메모리를 해제한다.
 */
bool destroy_histories(Histories **histories_state){
   HistoryNode *cur;
   int i;
   cur = ((*histories_state)->list->head);
   for(i=0;i<(*histories_state)->size + 1;i++){
      cur = (*histories_state)->list->head;
      (*histories_state)->list->head = (*histories_state)->list->head->next;
      free(cur->data);
      free(cur);
   }
```

```
    free((*histories_state)->list->tail->data);
    free((*histories_state)->list->tail);

    free((*histories_state)->list);

    free((*histories_state));

    return true;
}

/*
 * History 구조체를 생성(할당)하고 value 에 str 문자열을 저장한다.
 */
History* construct_history_with_string(char* str){
    History* hist = construct_history();
    strncpy(hist->value, str, HISTORY_MAX_LEN);

    return hist;
}

/*
 * Histories 에 target History 구조체를 저장한다.
 */
bool push_history(Histories* histories_store, History* target){
    assert(histories_store);
    assert(target);
    assert(histories_store->list);
    assert(histories_store->list->head);
    assert(histories_store->list->tail);

    HistoryNode* hist_node = (HistoryNode*)malloc(sizeof(HistoryNode));
    hist_node->data = target;

    hist_node->prev = histories_store->list->tail->prev;
    hist_node->next = histories_store->list->tail;
    histories_store->list->tail->prev->next = hist_node;
    histories_store->list->tail->prev = hist_node;
    histories_store->list->size += 1;

    histories_store->size += 1;
    return true;
}

/*
 * Histories 에 저장된 명령어 히스토리를 출력한다.
 */
void print_history(Histories *histories_store, char *last_command) {
    assert(histories_store);
    assert(last_command);
    assert(histories_store->list);
    assert(histories_store->list->head);
```

```
    assert(histories_store->list->tail);

    HistoryNode** cur = &histories_store->list->head;
    int i = 0;
    for(i=0;i<histories_store->size + 1;i++){
        if(i == 0){
            cur = &((*cur)->next);
            continue;
        }
        printf("%-4d %s", i, (char*)(*cur)->data);
        cur = &((*cur)->next);

    }
    printf("%-4d %s", i, last_command);
    printf("\n");
}
```

5.18 history.h

```
#ifndef __HISTORY_H__
#define __HISTORY_H__
#define HISTORY_MAX_LEN 501
#include "util.h"
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <stdio.h>

/*
 * 히스토리에 저장될 하나의 명령어 문자열을
 * history 구조체에 wrapping 해서 저장하게된다.
 */
typedef struct history {
    char value[HISTORY_MAX_LEN];
} History;

/*
 * history_list 의 Node 역활을 한다.
 */
typedef struct history_node{
    History* data;
    struct history_node* prev;
    struct history_node* next;
} HistoryNode;

/*
 * 명령어 히스토리를 링크드 리스트 형태로 저장하는 구조체.
 */
typedef struct history_list {
    struct history_node* head;
    struct history_node* tail;
    int size;
} HistoryList;
```

```
/*
 * 히스토리에 담긴 명령어의 개수를 저장 하고
 * 실질적으로 히스토리를 저장하는 history_list* 를 멤버로 갖고있다.
 * 일종의 wrapper struct 이다.
 */
typedef struct histories {
   int size;
   HistoryList* list;
} Histories;

/*
 * Histories 구조체를 생성(할당)하고 HistoryList 도 생성(할당)한다.
 */
Histories* construct_histories();

/*
 * History 구조체를 생성(할당)한다.
 */
History* construct_history();

/*
 * History 구조체를 생성(할당)하고 value 에 str 문자열을 저장한다.
 */
History* construct_history_with_string(char* str);

/*
 * Histories 에 target History 구조체를 저장한다.
 */
bool push_history(Histories* histories_store, History* target);

/*
 * Histories 에 저장된 명령어 히스토리를 출력한다.
 */
void print_history(Histories *histories_store, char *last_command);

/*
 * Histories 구조체를 생성하면서 할당했던 모든 메모리를 해제한다.
 */
bool destroy_histories(Histories **histories_state);

#endif
```

5.19 memory.c

```
#include "memory.h"

/*
 * Memories 구조체를 생성(할당)한다.
 */
Memories* construct_memories(){
   Memories* virtual_memories = (Memories*)malloc(sizeof(*virtual_memories));
```

```
    virtual_memories->last_idx = -1;

    return virtual_memories;
}

/*
 * Memories 구조체를 생성하면서 할당했던 메모리를 해제한다.
 */
bool destroy_memories(Memories** memories_state){
    free(*memories_state);
    return true;
}

/*
 * start ~ end 메모리 영역을 출력한다.
 * start 와 end 는 10 진수로 변환한 값이다.
 */
void print_memories(Memories* memories_state, int start, int end){
    assert(memories_state);
    assert(start >= 0);
    assert(end >= 0);
    assert(start <= end);
    assert(start < MEMORIES_SIZE);
    assert(end < MEMORIES_SIZE);

    int start_row = (start / 16)*16;
    // ex, [dump 11] 에서 start 는 17 이 되고, start_row 는 16 이 됨.
    // ex, [dump AA] 에서 start 는 170 이 되고, start_row 는 160 이 됨. ( 10 == 0xA0)

    int end_row = (end / 16)*16;
    int y, x, n;
    short value;
    for(y=start_row;y<=end_row;y+=16){
        printf("%05X ", y);
        for(x=0;x<16;x++){
            n = y + x;
            if(n < start || n > end) printf("   ");
            else printf("%02X ", memories_state->data[n].value);
        }
        printf("; ");
        for(x=0;x<16;x++){
            n = y + x;
            if(n < start || n > end) printf(".");
            else {
                value = memories_state->data[n].value;
                if(0x20 <= value && value <= 0x7E) printf("%c", value);
                else printf(".");
            }
        }
        printf("\n");
    }
}
```

```
/*
 * 메모리 영역의 address 주소의 값을 value 로 수정한다.
 */
void edit_memory(Memories* memories_state, int address, short value){
   assert(address < MEMORIES_SIZE && address >= 0);
   assert(value >= 0 && value <= 0xFF);

   memories_state->data[address].value = value;
}
```

5.20 memory.h

```
#ifndef __MEMORY_H__
#define __MEMORY_H__

#define MEMORIES_SIZE (1024 * 1024) // 1MB
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <stdbool.h>

/*
 * 메모리 값 하나를 의미하는 wrapper struct
 */
typedef struct memory {
   short value;
} Memory;

/*
 * 메모리 영역을 저장하는 구조체
 */
typedef struct memories {
   Memory data[MEMORIES_SIZE];
   int last_idx;
} Memories;

/*
 * Memories 구조체를 생성(할당)한다.
 */
Memories* construct_memories();

/*
 * start ~ end 메모리 영역을 출력한다.
 * start 와 end 는 10 진수로 변환한 값이다.
 */
void print_memories(Memories* memories_state, int start, int end);

/*
 * 메모리 영역의 address 주소의 값을 value 로 수정한다.
 */
void edit_memory(Memories* memories_state, int address, short value);
```

```
/*
 * Memories 구조체를 생성하면서 할당했던 메모리를 해제한다.
 */
bool destroy_memories(Memories** memories_state);

#endif
```

5.21 opcode.c

```
#include "opcode.h"

/*
 * Opcode 구조체를 생성(할당)한다.
 */
Opcode* construct_opcode(){
   Opcode* op = (Opcode*)malloc(sizeof(Opcode));
   op->value = -1;
   return op;
}

/*
 * op_node 를 생성(할당)한다.
 */
struct op_node* construct_opnode(){
   struct op_node* node = (struct op_node*)malloc(sizeof(struct op_node));

   return node;
}

/*
 * OpcodeTable 을 생성(할당)한다.
 */
OpcodeTable* construct_opcode_table(){
   OpcodeTable* table = (OpcodeTable*)malloc(sizeof(OpcodeTable));
   table->list = (OpLinkedList**)malloc(sizeof(OpLinkedList*)*20);
   for(int i = 0; i<20;i++) {
      table->list[i] = (OpLinkedList*)malloc(sizeof(OpLinkedList));
      table->list[i]->head = construct_opnode();
      table->list[i]->tail = construct_opnode();

      table->list[i]->head->prev = NULL;
      table->list[i]->head->next = table->list[i]->tail;
      table->list[i]->tail->prev = table->list[i]->head;
      table->list[i]->tail->next = NULL;

      table->list[i]->head->data = construct_opcode();
      table->list[i]->tail->data = construct_opcode();
      table->list[i]->size = 0;
   }
   table->size = 20;
   return table;
}
```

```
/*
 * opcode 파일을 읽어서 OpcodeTable 에 적절히 저장한다
 */
bool build_opcode_table(OpcodeTable* table){
   FILE* fp = fopen("opcode.txt", "rt");
   if(!fp) {
      fprintf(stderr, "[ERROR] opcode file not exists\n");
      return false;
   }
   char format_name[20];
   char name[20];
   unsigned int value;

   while (fscanf(fp, "%X %6s %5s",
         &value, name, format_name) != EOF){
      Opcode* opc = construct_opcode();

      strncpy(opc->mnemonic_name, name, 10);

      opc->value = value;
      if(opc->value > 0xFF) {
         fprintf(stderr, "[ERROR] %X %6s %5s is Invalid Input!\n", value, name, format_name);
         continue;
      }
//       printf("%s %d\n", opc->mnemonic_name,opc->value);
      if(COMPARE_STRING(format_name, "1")){
         opc->format = OP_FORMAT_1;
      }else if(COMPARE_STRING(format_name, "2")){
         if(COMPARE_STRING(opc->mnemonic_name, "CLEAR") ||
            COMPARE_STRING(opc->mnemonic_name, "TIXR"))
            opc->format = OP_FORMAT_2_ONE_REG;
         else if(COMPARE_STRING(opc->mnemonic_name, "SHIFTL") ||
               COMPARE_STRING(opc->mnemonic_name, "SHIFTR"))
            opc->format = OP_FORMAT_2_REG_N;
         else if(COMPARE_STRING(opc->mnemonic_name, "SVC"))
            opc->format = OP_FORMAT_2_ONE_N;
         else
            opc->format = OP_FORMAT_2_GEN;
      } else if(COMPARE_STRING(format_name, "3/4")){
         if(COMPARE_STRING(opc->mnemonic_name, "RSUB"))
            opc->format = OP_FORMAT_3_4_NO_OPERAND;
         else
            opc->format = OP_FORMAT_3_4_GEN;
      }
      insert_opcode(table, opc);
   }
   fclose(fp);

   return true;
}

/*
```

```
 * OpcodeTable 을 해제한다.
 */
bool destroy_opcode_table(OpcodeTable** table){
   assert(table);
   OpNode *cur;
   OpNode *next;
   OpLinkedList** list;
   int i, j;

   list = (*table)->list;
   for(i=0;i<(*table)->size;i++){
      cur = list[i]->head;

      for(j=0;j<list[i]->size+1;j++){
         next = cur->next;
         free(cur->data);
         free(cur);
         cur = next;
      }
      free(list[i]);
   }
   free(*table);

   return true;
}

/*
 * OpcodeTable 에 Opcode 를 추가한다.
 */
bool insert_opcode(OpcodeTable* table, Opcode* opc){
   OpNode* op_node = construct_opnode();

   op_node->data = opc;
   int hash = (int)hash_string(opc->mnemonic_name, 20);

   op_node->prev = table->list[hash]->tail->prev;
   op_node->next = table->list[hash]->tail;
   table->list[hash]->tail->prev->next = op_node;
   table->list[hash]->tail->prev = op_node;
   table->list[hash]->size += 1;

//    assert(opc->value >= 0);
//    assert(table->list[hash]->tail->prev->data->value >= 0);
   return true;
}

/*
 * OpcodeTable 에서 name 문자열 이름의 mnemonic 을 가진
 * Opcode 를 찾고 리턴한다.
 */
Opcode* find_opcode_by_name(OpcodeTable* table, char* name){
   assert(strlen(name) <= 14);
```

```
    assert(table);

    int hash = (int)hash_string(name, table->size);
    int i = 0;
    OpNode** cur = &(table->list[hash]->head);
    Opcode* opc;
//    printf("hash: %d\n", hash);
    for(i=0;i<(table->list[hash]->size)+1;i++){
        opc = (*cur)->data;
//        printf("op-name: %s\n", opc->mnemonic_name);
        if(opc->value == -1){
            cur = &((*cur)->next);
            continue;
        }
        if(COMPARE_STRING(name, opc->mnemonic_name)){
            return opc;
        }
        cur = &((*cur)->next);
    }
    return NULL;
}

/*
 * OpcodeTable 에 저장된 opcode 목록을 출력한다.
 */
void print_opcodes(OpcodeTable* table){
    int size = table->size;
    for(int i = 0;i < size;i++){
        printf("%2d : ", i);
        OpNode** cur = &(table->list[i]->head);
        Opcode* opc;
        for(int j=0;j<table->list[i]->size+1;j++){
            opc = (*cur)->data;
//            if(opc->value == -1) cnt -= 1;
            if(opc->value == -1){
                cur=&((*cur)->next);
                continue;
            }
            printf("[%s,%02X] ",opc->mnemonic_name,opc->value);
//            printf("gogo %s %d\n", opc->mnemonic_name, opc->value);
//            cnt += 1;
            if(j != table->list[i]->size)
                printf(" -> ");
            cur = &((*cur)->next);
        }
        printf("\n");
    }
}
```

5.22 opcode.h

```
#ifndef __OPCODE_H__
#define __OPCODE_H__
```

```
#include <stdbool.h>
#include "util.h"

/*
 * opcode 의 format 을 enum 으로 표현한다. (구현중)
 */
enum op_format {
   OP_FORMAT_1,

   OP_FORMAT_2_ONE_REG, OP_FORMAT_2_REG_N,
   OP_FORMAT_2_ONE_N, OP_FORMAT_2_GEN,

   OP_FORMAT_3_4_NO_OPERAND, OP_FORMAT_3_4_GEN
};

/*
 *  opcode 의 mnemonic 을 enum 으로 표현한다. (구현중)
 */
enum op_mnemonic {
   OP_ADD, OP_ADDR
};

/*
 * opcode 정보 하나를 나타낸다.
 */
typedef struct opcode {
   enum op_format format;
   enum op_mnemonic mnemonic;
   char mnemonic_name[10];
   int value;
} Opcode;

/*
 * op_linked_list 의 Node 역활
 */
typedef struct op_node{
   Opcode* data;
   struct op_node* prev;
   struct op_node* next;
} OpNode;

/*
 * 링크드 리스트
 */
typedef struct op_linked_list {
   struct op_node* head;
   struct op_node* tail;
   int size;
}OpLinkedList;

/*
 * opcode 정보들을 해시테이블 형태로 저장하기 위한 구조체.
```

```
 */
typedef struct opcode_table {
   OpLinkedList** list;
   int size;
} OpcodeTable;

/*
 * Opcode 구조체를 생성(할당)한다.
 */
Opcode* construct_opcode();

/*
 * op_node 를 생성(할당)한다.
 */
struct op_node* construct_opnode();

/*
 * OpcodeTable 을 생성(할당)한다.
 */
OpcodeTable* construct_opcode_table();

/*
 * opcode 파일을 읽어서 OpcodeTable 에 적절히 저장한다
 */
bool build_opcode_table(OpcodeTable* table);

/*
 * OpcodeTable 을 해제한다.
 */
bool destroy_opcode_table(OpcodeTable** table);

/*
 * OpcodeTable 에 Opcode 를 추가한다.
 */
bool insert_opcode(OpcodeTable* table, Opcode* opc);

/*
 * OpcodeTable 에서 name 문자열 이름의 mnemonic 을 가진
 * Opcode 를 찾고 리턴한다.
 */
Opcode* find_opcode_by_name(OpcodeTable* table, char* name);

/*
 * OpcodeTable 에 저장된 opcode 목록을 출력한다.
 */
void print_opcodes(OpcodeTable* table);

#endif
```

5.23 state.c

```
#include "state.h"
```

```
/*
 * History, 가상 Memory, Opcode 정보가 초기화(및 저장)된 State* 을 리턴한다.
 */
State* construct_state(){
   State* state_obj = (State*)malloc(sizeof(*state_obj));

   state_obj->histories_state = construct_histories();
   state_obj->memories_state = construct_memories();

   state_obj->opcode_table_state = construct_opcode_table();
   build_opcode_table(state_obj->opcode_table_state);

   return state_obj;
}

/*
 * state_store 가 동적 할당한 모든 메모리를 해제한다.
 */
bool destroy_state(State **state_store){

   destroy_histories(&((*state_store)->histories_state));
   destroy_memories(&((*state_store)->memories_state));
   destroy_opcode_table(&(*state_store)->opcode_table_state);

   free(*state_store);

   return true;
}

/*
 * history_str 문자열을 명령어 히스토리에 기록한다.
 */
bool add_history(State *state_store, char* history_str){
   return push_history(state_store->histories_state,
         construct_history_with_string(history_str));
}

/*
 * 명령어 히스토리를 출력한다.
 */
void print_histories_state(State* state_store, char* last_command){
   print_history(state_store->histories_state, last_command);
}
```

5.24 state.h

```
#ifndef __STATE_H__
#define __STATE_H__

#include "history.h"
#include "memory.h"
#include "opcode.h"
#include <stdlib.h>
```

```
/*
 * state 구조체에서는 명령어 히스토리, 가상 메모리 영역, Opcode 정보를 저장하는
 * 구조체 포인터들을 멤버 변수로 갖고있다.
 */
typedef struct state {
   // 명령어 히스토리
   Histories* histories_state;

   // 가상 메모리 영역
   Memories* memories_state;

   // opcode 파일을 읽어 들인 내용들
   OpcodeTable* opcode_table_state;
} State;

/*
 * History, 가상 Memory, Opcode 정보가 초기화(및 저장)된 State* 을 리턴한다.
 */
State* construct_state();

/*
 * state_store 가 동적 할당한 모든 메모리를 해제한다.
 */
bool destroy_state(State **state_store);

/*
 * history_str 문자열을 명령어 히스토리에 기록한다.
 */
bool add_history(State *state_store, char* history_str);

/*
 * 명령어 히스토리를 출력한다.
 */
void print_histories_state(State* state_store, char* last_command);

#endif
```

5.25 util.c

```
#include "util.h"

/*
 * hashtable 구현을 위한 hash function
 */
size_t hash_string (char *str, int hash_size){
   int32_t hash = 2829;
   int32_t c;
   size_t res;
   while((c = *str++)){
      hash = (hash * 615) + c;
   }
   res = (size_t)hash % hash_size;
```

```
    return res;
}

/*
 * 문자열 str 이 0 으로 변환될수있는지 확인한다.
 * ex, is_zero_str("0000") => return true
 * ex, is_zero_str("A00") => return false
 */
bool is_zero_str(char* str){
    assert(str);
    int len = (int)strlen(str);
    int i;
    for(i=0;i<len;i++)
        if(str[i] != '0')
            return false;
    return true;
}

/*
 * 문자열 str 이 16 진수인지 확인한다.
 * ex, is_valid_hex("00F1") => return true
 * ex2, is_valid_hex("FZ") => return false
 */
bool is_valid_hex(char* str){
    assert(str);
    int l = (int)strlen(str), i;
    for(i=0;i<l;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 false;
    }
    return true;
}


/*
 * 문자열 str 이 [0 ~ max_size-1] 범위의 적절한 주소값인지 확인한다.
 * ex, is_zero_str("00F", 100) => return true
 * ex, is_zero_str("FG", 100000) => return false
 */
bool is_valid_address(char *str, int max_size) {
    assert(str);
    assert(max_size);
    int target = (int)strtol(str, NULL, 16);
```

```
    if(target < 0) return false; // 0 보다 큰지 검증
    if(target == 0 && !is_zero_str(str)) return false; // 올바른 hex 값인지 검증
    if(target >= max_size) return false; // 범위 내에 있는지 검증
    if(!is_valid_hex(str)) return false; // 올바른 hex 값인지 검증

    return true;
}
```

5.26 util.h

```
#ifndef __UTIL_H__
#define __UTIL_H__
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

#define COMPARE_STRING(T, S) (strcmp ((T), (S)) == 0)

/*
 * hashtable 구현을 위한 hash function
 */
size_t hash_string (char *str, int hash_size);

/*
 * 문자열 str 이 0 으로 변환될수있는지 확인한다.
 * ex, is_zero_str("0000") => return true
 * ex, is_zero_str("A00") => return false
 */
bool is_zero_str(char* str);

/*
 * 문자열 str 이 16 진수인지 확인한다.
 * ex, is_valid_hex("00F1") => return true
 * ex2, is_valid_hex("FZ") => return false
 */
bool is_valid_hex(char* str);

/*
 * 문자열 str 이 [0 ~ max_size-1] 범위의 적절한 주소값인지 확인한다.
 * ex, is_zero_str("00F", 100) => return true
 * ex, is_zero_str("FG", 100000) => return false
 */
bool is_valid_address(char *str, int max_size);

#endif
```