**과목명: 시스템프로그래밍**

**2분반**

**<<Project #1>>**

**서강대학교 수학/컴퓨터공학**

**[학번] 20141303**

**[이름] 이윤제**

목 차

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
3. **모듈 정의**
   1. Main(20141303) 모듈
      1. 기능
      2. 사용 변수
   2. Shell 모듈
      1. 기능
      2. 사용 변수
      3. 함수 목록
   3. Memory 모듈
      1. 기능
      2. 사용 변수
      3. 함수 목록
   4. OPCODE 모듈
      1. 기능
      2. 사용 변수
      3. 함수 목록
   5. Error 모듈
      1. 기능
      2. 사용 변수
      3. 함수 목록
4. **전역 변수 정의**
5. **코드 설명**
   1. 20141303.h
   2. 20141303.c
   3. Shell.h
   4. Shell.c
   5. Memory.h
   6. Memory.c
   7. Opcode.h
   8. Opcode.c
   9. Error.h
   10. Error.c
6. **프로그램 개요**

**SIX/XE 머신**을 구현하기 위한 전 단계로서 어셈블러, 링커, 로더를 실행하게 될 **shell**과 컴파일을 통해 만들어진 object 코드가 적재되고 실행될 **메모리 공간**, 그리고 mnemonic(ADD, COMP, FLOAR, etc..)을 opcode 값으로 변환하는 **OPCODE 테이블** 및 **관련 명령어**들을 구현하였습니다.

1. **프로그램 설명**

**2-1. 프로그램 흐름도**

Error 모듈

Opcode 모듈

Main 모듈

Shell 모듈

Memory 모듈

OPCODE 관련 구조체 및 명령어를 처리하는 모듈

메모리 관련 명령어를 처리하는 모듈

사용자로부터 입력받은 argument를 Shell 명령어 종류에 따라 분류하고 각각 기능에 맞게 수행한다.

Invalid command,

undefined parameter, range error 등 여러 상황에 맞는 에러를 정의한 모듈

각 모듈의 전처리 함수를 호출하고, Command loop로 진입한다. 프로그램이 끝나기 전 메모리 해제 등의 후처리를 수행한다.

프로그램의 흐름도는 이전 장의 그림과 같다.

먼저 Main 모듈에서는 SIC/XE 머신을 구동하기 전 각 모듈에서 필요한 초기화나 전처리를 수행한다. 예를 들어 opcode table을 구성해야 하고, 메모리가 쓰레기 값을 가지지 않도록 0으로 초기화해야 한다. 이렇게 전처리가 끝나고 command loop에 진입하면, 사용자가 quit 명령어를 입력하기 전까지 입력받은 명령어에 대응하는 작업을 수행한다. Command loop를 빠져나오면 프로그램을 종료하기 전에 해제해줘야 할 메모리(history, opcode table 등)을 해제해 주고 프로그램을 종료한다.

Shell 모듈은 main모듈에서 command loop를 돌며 사용자가 입력한 명령어들을 처리한다. 입력받은 명령어를 tokenize하고, tokenize한 token들을 통해 수행해야 할 연산을 확인한다. 그리고 그 연산들을 수행하게 되는데, 기본 shell 명령어인 [help, dir, history, quit] 은 shell 모듈에서 직접 수행하고, 메모리 관련 명령어와 opcode 관련 명령어는 해당 모듈로 위임한다.

Memory 모듈은 메모리 관련 명령어인 [dump, edit, fill, reset]을 수행한다. 메모리 관련 명령은 address와 value가 valid한지 여부가 중요하기 때문에 이에 대한 검사를 매 함수에서 함께 수행한다.

Opcode 모듈은 opcode table 관련 명령어인 [opcode, opcodelist] 를 수행한다. 프로그램이 시작하고 command loop에 들어가기 전 “opcode.txt”에서 받아온 data를 통해 opcode hash table을 구성하고, 이를 바탕으로 opcode 명령어를 처리한다.

마지막으로 Error 모듈은 proj1 에서 발생할 수 있는 여러 오류들에 대해 그에 대응하는 메시지를 출력한다. 어느 모듈에서나 접근할 수 있어야 하기 때문에 가장 기본 단계에 위치시킨 모듈이다.

1. **모듈 정의**

**3.1. Main(20141303) 모듈**

3.1.1 기능

각 모듈에서 필요한 초기화나 전처리를 수행한다. 예를 들어 opcode table을 구성해야 하고, 메모리가 쓰레기 값을 가지지 않도록 0으로 초기화해야 한다. 이렇게 전처리가 끝나고 command loop에 진입하면, 사용자가 quit 명령어를 입력하기 전까지 입력받은 명령어에 대응하는 작업을 수행한다. Command loop를 빠져나오면 프로그램을 종료하기 전에 해제해줘야 할 메모리(history, opcode table 등)을 해제해 주고 프로그램을 종료한다.

3.1.2 사용 변수

|  |
| --- |
| 사용자로부터 수행을 위해 받은 Argument를 저장하기 위한 char 배열 arg  char arg[ARG\_LEN]; |

**3.2. Shell 모듈**

3.2.1 기능

main모듈에서 command loop를 돌며 사용자가 입력한 명령어들을 처리한다. 입력받은 명령어를 tokenize하고, tokenize한 token들을 통해 수행해야 할 연산을 확인한다. 그리고 그 연산들을 수행하게 되는데, 기본 shell 명령어인 [help, dir, history, quit] 은 shell 모듈에서 직접 수행하고, 메모리 관련 명령어와 opcode 관련 명령어는 해당 모듈로 위임한다.

3.2.2 사용 변수

|  |
| --- |
| 입력받은 command를 tokenize하고 대응하는 함수로 넘기는 과정을 자연스럽게 하기 위해, 각 command를 const char\* type으로 저장하기 위한 배열 valid\_commands와, enum type을 define하여 switch문으로 command를 분류할 수 있게 하기 위한 command\_list 사용.  const char\* valid\_commands[COMMAND\_NUM] = {  "h",  "help",  "d",  "dir",  "q",  "quit",  "hi",  "history",  "du",  "dump",  "e",  "edit",  "f",  "fill",  "reset",  "opcode",  "opcodelist"  };  typedef enum {  h\_,  help\_,  d\_,  dir\_,  q\_,  quit\_,  hi\_,  history\_,  du\_,  dump\_,  e\_,  edit\_,  f\_,  fill\_,  reset\_,  opcode\_,  opcodelist\_  } command\_list; |

|  |
| --- |
| 프로그램 실행 중 입력받은 valid commands의 history를 linked list 형태로 유지/관리하기 위한 node type 선언.  typedef struct history\_node {  char name[101];  struct history\_node\* next;  }history\_node; |

3.2.3 함수 목록

|  |  |
| --- | --- |
| void help(); | print all valid commands with their correct format |
| void dir(); | print list of directory / files on current directory |
| void quit(); | quit program |
| void add\_history(char\* arg); | add command history to history head |
| void remove\_history\_tail(); | remove last history node. This is for exceptions(invalid commands) |
| void clear\_history(); | clear all allocated history memories before exit program |
| void history(); | print valid commands history with indices |
| int char\_to\_hex(char c); | convert each character with corresponding hexadecimal value |
| int str\_to\_hex(char\* str); | convert string with corresponding hexadecimal value |
| int make\_command(char\* str); | match string with corresponding command number. This is for switch statement in run() |
| bool run(char\* arg); | Tokenize whole argument with command and parameters, then execute corresponding job. |

**3.3. Memory 모듈**

3.3.1 기능

Memory 모듈은 메모리 관련 명령어인 [dump, edit, fill, reset]을 수행한다. 메모리 관련 명령은 address와 value가 valid한지 여부가 중요하기 때문에 이에 대한 검사를 매 함수에서 함께 수행한다.

3.3.2 사용 변수

|  |
| --- |
| //declare 1MB memory and last address for dump function  unsigned char mem[MAX\_ADDR + 1] = {0};  int last\_addr = -1; |

3.3.3 함수 목록

|  |  |
| --- | --- |
| bool is\_valid\_val(int val); | return if value is in range 0x0 ~ 0xFF |
| bool is\_valid\_addr(int addr); | return if address is in range 0x0 ~ 0xFFFFF |
| int dump(int start, int end); | dump memory from start to end, if not valid address range, return error |
| int edit(int idx, int val); | edit memory in idx to val, if not valid address or val, return error |
| int fill(int start, int end, int val); | fill memory from start to end with val, if not valid argument/address or val, return error |
| void reset(); | reset all memory to zero |

**3.4. OPCODE 모듈**

3.4.1 기능

Opcode 모듈은 opcode table 관련 명령어인 [opcode, opcodelist] 를 수행한다. 프로그램이 시작하고 command loop에 들어가기 전 “opcode.txt”에서 받아온 data를 통해 opcode hash table을 구성하고, 이를 바탕으로 opcode 명령어를 처리한다.

3.4.2 사용 변수

|  |
| --- |
| //struct node structure for opcode hash table  typedef struct hash\_node {  char code[10];  char name[10];  char data[10];  struct hash\_node\* next;  }hash\_node;  //declare 20-sized hash table  struct hash\_node\* hash\_head[20] = {NULL}; |

3.4.3 함수 목록

|  |  |
| --- | --- |
| void add\_hash\_node(struct hash\_node\* new\_); | insert new hash node to matched hash index |
| void init\_optable(); | initialize opcode table with given text file "opcode.txt" |
| int opcode(char\* str); | output matching opcode with input opcode name |
| void opcodelist(); | print all opcode data in hash table separated with index |
| void clear\_optable(); | before exit program, free all allocated memories used in opcode table |

**3.5. Error 모듈**

3.5.1 기능

Error 모듈은 proj1 에서 발생할 수 있는 여러 오류들에 대해 그에 대응하는 메시지를 출력한다..

3.5.2 사용 변수

|  |
| --- |
| #define ERROR -1  #define EMPTY -1  #define SUCCESS 1  #define IMPOSSIBLE 2000000000  ERROR : command가 정상적으로 처리되지 않은, 즉 history에 포함하면 안된다는 뜻  EMPTY : 해당 인자가 비어 있다는 뜻. 필수인자가 EMPTY라면 오류를 내보내야 한다.  SUCCESS : command가 문제없이 정상적으로 수행을 마쳤다는 뜻.  IMPOSSIBLE : 입력 받은 변수 중 16진수로 표현이 불가능 하다는 뜻. Ex) YJ |

3.5.3 함수 목록

|  |  |
| --- | --- |
| void undefined\_command(); | Error occurred by invalid commands |
| void undefined\_argument(); | Error occurred by Absence or invalid arguments. ex) 0xHZ |
| void out\_of\_range(); | Error occurred by out of range on memory address or value |
| void start\_bigger\_than\_end(); | Error occurred when start memory address is bigger than end |

1. **전역 변수 정의**

**전역 변수를 포함한 사용 변수들에 대해서 각 모듈의 [사용 변수]에서 설명했습니다.**

1. **코드 및 구현 설명**

**5.1 20141303.h**

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include "shell.h"  //sicsim> argument  char arg[ARG\_LEN]; |

**5.2 20141303.c**

|  |
| --- |
| #include "20141303.h"  int main() {  //initialize optable before start  init\_optable();  //initialize memory to zero before start  reset();  while(1) {  memset(arg, 0, ARG\_LEN);  printf("sicsim> ");  fgets(arg, ARG\_LEN, stdin);  arg[(int)strlen(arg) - 1] = 0; //erase '\n'  if(!run(arg)) break;  printf("\n");  }  //clear history node and opcode table before exit  clear\_history();  clear\_optable();  return 0;  } |

**5.3 shell.h**

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <dirent.h>  #include <sys/stat.h>  #include "memory.h"  #include "opcode.h"  #define ARG\_LEN 101  #define COMMAND\_NUM 17  //declare commands list to classify commands easier so that struct run() function clearly  typedef enum {  h\_,  help\_,  d\_,  dir\_,  q\_,  quit\_,  hi\_,  history\_,  du\_,  dump\_,  e\_,  edit\_,  f\_,  fill\_,  reset\_,  opcode\_,  opcodelist\_  } command\_list;  //node structure and head pointer for history functions  typedef struct history\_node {  char name[101];  struct history\_node\* next;  }history\_node;  //shell functions  void help();  void dir();  void quit();  void add\_history(char\* arg);  void remove\_history\_tail();  void clear\_history();  void history();  int str\_to\_hex(char\* str);  int make\_command(char\* str);  bool run(char\* arg); |

**5.4 shell.c**

|  |
| --- |
| #include "shell.h"  //constant string array for valid shell commands to match in run()  const char\* valid\_commands[COMMAND\_NUM] = {  "h",  "help",  "d",  "dir",  "q",  "quit",  "hi",  "history",  "du",  "dump",  "e",  "edit",  "f",  "fill",  "reset",  "opcode",  "opcodelist"  };  //head node pointer for history list  struct history\_node\* history\_head = NULL;  //print all valid commands with their correct format  void help() {  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;  }  //print list of directory / files on current directory  void dir() {  struct dirent \*\*namelist;  int count, idx, len;  struct stat buf;    //scan all files in directory  if((count = scandir("./", &namelist, NULL, alphasort)) != -1) {  for(idx = 0; idx < count; idx++) {  char fname[101];  strcpy(fname, namelist[idx]->d\_name);  len = strlen(fname);  lstat(namelist[idx]->d\_name, &buf);    //exe file(a.out\*)  if(fname[len-4] == '.' && fname[len-3] == 'o' && fname[len-2] == 'u' && fname[len-1] == 't') {  printf("%s\*\n", namelist[idx]->d\_name);  }  //directory(a/)  else if(S\_ISDIR(buf.st\_mode)) {  printf("%s/\n", namelist[idx]->d\_name);  }  //regular files  else if(S\_ISREG(buf.st\_mode)) {  printf("%s\n", namelist[idx]->d\_name);  }  }  for(idx = 0; idx < count; idx++) {  free(namelist[idx]);  }  free(namelist);  }  return;  }  //nothing to do in here to quit program  void quit() {  return;  }  //add command history to history head  void add\_history(char\* arg) {  struct history\_node\* new\_ = (struct history\_node\*)malloc(sizeof(history\_node));  new\_->next = NULL;  strcpy(new\_->name, arg);  if(history\_head==NULL) history\_head = new\_;  else {  struct history\_node\* tail = history\_head;  while(tail->next != NULL) {  tail = tail->next;  }  tail->next = new\_;  }  return;  }  //remove last history node. This is for exceptions(invalid commands)  void remove\_history\_tail() {  if(history\_head==NULL) return;  if(history\_head->next == NULL) {  free(history\_head);  history\_head = NULL;  return;  }  struct history\_node\* tmp;  tmp = history\_head;  while((tmp->next)->next != NULL) {  tmp = tmp->next;  }  free(tmp->next);  tmp->next = NULL;  return;  }  //clear all allocated history memories before exit program  void clear\_history() {  struct history\_node\* tmp;  while(history\_head != NULL) {  tmp = history\_head;  history\_head = history\_head->next;  free(tmp);  }  return;  }  //print valid commands history with indices  void history() {  int idx = 1;  struct history\_node\* tmp = history\_head;  while(tmp) {  printf("%3d ", idx);  printf("%s\n", tmp->name);  idx++;  tmp = tmp->next;  }  return;  }  //convert each character with corresponding hexadecimal value  int char\_to\_hex(char c) {  if('0' <= c && c <= '9') return c - '0';  if('A' <= c && c <= 'F') return c - 'A' + 10;  if('a' <= c && c <= 'f') return c - 'a' + 10;  return IMPOSSIBLE;  }  //convert string with corresponding hexadecimal value  int str\_to\_hex(char\* str) {  //EMPTY means there is no parameter  if(str==NULL) return EMPTY;  int res = 0, tmp;  int i;  for(i = 0; i < (int)strlen(str); i++) {  res \*= 16;  tmp = char\_to\_hex(str[i]);  //IMPOSSIVLE means incorrect format hexadecimal like YJ  if(tmp == IMPOSSIBLE) return IMPOSSIBLE;  res += tmp;  }  return res;  }  //math string wuth corresponding command number. This is for switch statement in run()  int make\_command(char\* str) {  int i;  for(i=0;i<COMMAND\_NUM;i++) {  if(!strcmp(str, valid\_commands[i])) return i;  }  return IMPOSSIBLE;  }  //Tokenize whole argument with command and parameters, then execute corresponding job.  bool run(char\* arg) {  char \*command, \*param1, \*param2, \*param3;  char arg\_cpy[ARG\_LEN];  int p1, p2, p3, err, len1 = 0, len2 = 0;  strcpy(arg\_cpy, arg);  add\_history(arg\_cpy);  //split argument to command, param1~3, param1~3 can be NULL  command = strtok(arg, " ");  param1 = strtok(NULL, " ");  param2 = strtok(NULL, " ");  param3 = strtok(NULL, " ");  //for make str\_to\_hexa easier, erase ','  if(param1 != NULL) len1 = strlen(param1);  if(param2 != NULL) len2 = strlen(param2);  if(len1>0 && param1[len1-1]==',') param1[len1-1] = 0;  if(len2>0 && param2[len2-1]==',') param2[len2-1] = 0;  //p1~3 == integer version of param1~3  p1 = str\_to\_hex(param1);  p2 = str\_to\_hex(param2);  p3 = str\_to\_hex(param3);  int com = make\_command(command);  //in switch statement, if some cases return ERROR, we will remove history of its command  switch(com) {  case h\_:  case help\_:  help();  return true;  case d\_:  case dir\_:  dir();  return true;  case q\_:  case quit\_:  quit();  return false;  case hi\_:  case history\_:  history();  return true;  case du\_:  case dump\_:  err = dump(p1, p2);  if(err == ERROR) remove\_history\_tail();  return true;  case e\_:  case edit\_:  err = edit(p1, p2);  if(err == ERROR) remove\_history\_tail();  return true;  case f\_:  case fill\_:  err = fill(p1, p2, p3);  if(err == ERROR) remove\_history\_tail();  return true;  case reset\_:  reset();  return true;  case opcode\_:  err = opcode(param1);  if(err == ERROR) remove\_history\_tail();  return true;  case opcodelist\_:  opcodelist();  return true;  default:  remove\_history\_tail();  undefined\_command();  return true;  }  return true;  } |

**5.5 memory.h**

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include "error.h"  #define MAX\_VAL 255  #define MAX\_ADDR 1048575  //memory functions  bool is\_valid\_val(int val);  bool is\_valid\_addr(int addr);  int dump(int start, int end);  int edit(int idx, int val);  int fill(int start, int end, int val);  void reset(); |

**5.6 memory.c**

|  |
| --- |
| #include "memory.h"  //declare 1MB memory and last address for dump function  unsigned char mem[MAX\_ADDR + 1] = {0};  int last\_addr = -1;  //return if value is in range 0x0 ~ 0xFF  bool is\_valid\_val(int val) {  return (val >= 0 && val <= MAX\_VAL);  }  //return if address is in range 0x0 ~ 0xFFFFF  bool is\_valid\_addr(int addr) {  return (addr >= 0 && addr <= MAX\_ADDR);  }  //dump memory from start to end, if not valid address range, return error  int dump(int start, int end){  //classify three formats  //dump  if(start == EMPTY && end == EMPTY) {  start = (last\_addr + 1) % (MAX\_ADDR + 1);  end = start + 159;  }  //dump start  else if(end == EMPTY) {  if(!is\_valid\_addr(start)) {  out\_of\_range();  return ERROR;  }  end = start + 159;  }  //dump start, end  else {  if(!is\_valid\_addr(start) || !is\_valid\_addr(end)) {  out\_of\_range();  return ERROR;  }  else if (start > end) {  start\_bigger\_than\_end();  return ERROR;  }  }  //now we have start and end, will print with format  //end = min(end, MAC\_ADDR) is required by policy  if(end > MAX\_ADDR) end = MAX\_ADDR;  int r\_start = start - start%16; //first row  int r\_end = end - end%16; //last row  int i, j;  for(i = r\_start; i <= r\_end; i += 16) {  printf("%05X ", i);  for(j = 0; j < 16; j++){  if(i + j < start || i + j > end) printf(" ");  else {  printf("%02X ", mem[i + j]);  }  }  printf("; ");  for(j = 0; j < 16; j++) {  if(i + j < start || i + j > end) printf(".");  else {  if(mem[i + j] >= 0x20 && mem[i+j] <= 0x7E) printf("%c", mem[i+j]);  else printf(".");  }  }  printf("\n");  }  last\_addr = end;  return SUCCESS;  }  //edit memory in idx to val, if not valid address or val, return error  int edit(int idx, int val) {  if(idx == EMPTY || val == EMPTY) {  undefined\_argument();  return ERROR;  }  else if(!is\_valid\_addr(idx) || !is\_valid\_val(val)) {  out\_of\_range();  return ERROR;  }  mem[idx] = val;  return SUCCESS;  }  //fill memory from start to end with val, if not valid argument/address or val, return error  int fill(int start, int end, int val) {  if(start == EMPTY || end == EMPTY || val == EMPTY) {  undefined\_argument();  return ERROR;  }  else if(!is\_valid\_addr(start) || !is\_valid\_addr(end) || !is\_valid\_val(val)) {  out\_of\_range();  return ERROR;  }  else if (start > end) {  start\_bigger\_than\_end();  return ERROR;  }  int i;  for(i = start; i <= end; i++) {  mem[i] = val;  }  return SUCCESS;  }  //reset all memory to zero  void reset(){  int i;  for(i = 0; i <= MAX\_ADDR; i++) mem[i] = 0;  return;  } |

**5.7 opcode.h**

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include "error.h"  //node structure for opcode hash table  typedef struct hash\_node {  char code[10];  char name[10];  char data[10];  struct hash\_node\* next;  }hash\_node;  //opcode functions  void add\_hash\_node(struct hash\_node\* new\_);  void init\_optable();  int opcode(char\* str);  void opcodelist();  void clear\_optable(); |

**5.8 opcode.c**

|  |
| --- |
| #include "opcode.h"  //declare 20-sized hash table  struct hash\_node\* hash\_head[20] = {NULL};  //insert new hash node to matched hash index  void add\_hash\_node(struct hash\_node\* new\_) {  int idx = (new\_->name[0] - 'A') % 20; //given hash index depends on 1st Character of name  //add new node to tail  if(hash\_head[idx] == NULL) hash\_head[idx] = new\_;  else {  struct hash\_node\* tail = hash\_head[idx];  while(tail->next != NULL) {  tail = tail->next;  }  tail->next = new\_;  }  return;  }  //initialize opcode table with given text file "opcode.txt"  void init\_optable() {  int i;  for(i = 0; i < 20; i++) hash\_head[i] = NULL;  char tmp[10];  FILE\* fp;  fp = fopen("./opcode.txt", "r");  if(fp == NULL) {  printf("Error : opcode.txt does'nt exist.\n");  return;  }  //read opcode, mnemonic, format of each opcode node  while(fscanf(fp, "%s", tmp) != EOF) {  struct hash\_node\* new\_ = (struct hash\_node\*)malloc(sizeof(hash\_node));  new\_->next = NULL;  strcpy(new\_->code, tmp);  fscanf(fp, "%s", tmp);  strcpy(new\_->name, tmp);  fscanf(fp, "%s", tmp);  strcpy(new\_->data, tmp);  add\_hash\_node(new\_);  }  fclose(fp);  return;  }  //output matching opcode with input opcode name  int opcode(char\* str) {  if(str == NULL){  undefined\_argument();  return ERROR;  }  //get hash index  int idx = (str[0] - 'A') % 20;  struct hash\_node\* tmp = hash\_head[idx];  //find corresponding node while iterating  while(tmp != NULL) {  if(!strcmp(tmp->name, str)) {  printf("opcode is %s\n", tmp->code);  return SUCCESS;  }  tmp = tmp->next;  }  undefined\_argument();  return ERROR;  }  //print all opcode data in hash table separated with index  void opcodelist(){  int i;  for(i = 0; i < 20; i++) {  printf("%3d : ", i);  struct hash\_node\* tmp = hash\_head[i];  while(tmp != NULL) {  if(tmp->next != NULL) printf("[%s, %s] -> ", tmp->name, tmp->code);  else printf("[%s, %s]", tmp->name, tmp->code);  tmp = tmp->next;  }  printf("\n");  }  return;  }  //before exit program, free all allocated memories used in opcode table  void clear\_optable() {  int i;  for(i = 0; i < 20; i++) {  struct hash\_node\* tmp;  while(hash\_head[i] != NULL) {  tmp = hash\_head[i];  hash\_head[i] = hash\_head[i]->next;  free(tmp);  }  }  return;  } |

**5.9 error.h**

|  |
| --- |
| #include <stdio.h>  #include <stdbool.h>  #define ERROR -1  #define EMPTY -1  #define SUCCESS 1  #define IMPOSSIBLE 2000000000  //declaration for boolean type  //extern typedef enum {false, true} bool;  //error functions  void undefined\_command();  void undefined\_argument();  void out\_of\_range();  void start\_bigger\_than\_end(); |

**5.10 error.c**

|  |
| --- |
| #include "error.h"  //Error occurred by invalid commands  void undefined\_command() {  printf("Error : Undefined Command!\n");  return;  }  //Error occurred by Absence or invalid arguments. ex) 0xHZ  void undefined\_argument() {  printf("Error : Undefined argument!\n");  return;  }  //Error occurred by out of range on memory address or value  void out\_of\_range() {  printf("Error : Value Out of Range!\n");  return;  }  //Error occurred when start memory address is bigger than end  void start\_bigger\_than\_end() {  printf("Error : start addr is bigger than end addr!\n");  return;  } |