**과목명: 시스템프로그래밍**

**2분반**

**<<Project #2>>**

**서강대학교 수학/컴퓨터공학**

**[학번] 20141303**

**[이름] 이윤제**

목 차

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
3. **모듈 정의**
   1. base 모듈
      1. 기능
      2. 사용 변수
   2. assembler 모듈
      1. 기능
      2. 사용 변수
      3. 함수 목록
   3. Opcode 모듈
      1. 추가 함수 목록
4. **전역 변수 정의**
5. **알고리즘 / 구현 방식**
6. **코드 설명**
   1. base.h
   2. base.c
   3. assembler.h
   4. assembler.c
7. **프로그램 개요**

**프로젝트1에서 구현한 셸(shell)에 assemble 기능을 추가하는 프로그램. SIC/XE의 assembly program source 파일을 입력받아서 object파일을 생성하고, 어셈블리 과정 중 생성된 symbol table과 결과물인 object 파일을 볼 수 있는 기능을 제공할 수 있도록 구현했습니다.**

1. **프로그램 설명**

**2-1. 프로그램 흐름도**

Base 모듈

Opcode 모듈

Main 모듈

Shell 모듈

Memory 모듈

Assembler 모듈

Error 모듈

프로그램의 흐름도는 이전 장의 그림과 같다.

새로 추가된 두 개의 모듈인 Assembly, Base 모듈을 제외하면 기존에 사용하던 모듈들이기 때문에 추가적인 설명은 불필요하다고 생각했습니다.

Opcode에서는 새로운 함수들이 추가되었기 때문에 함께 언급하였습니다.

Assembler에서 opcode 모듈의 함수인 opname\_to\_hex, get\_format, 등을 이용하여 opcode를 분별하기 때문에 opcode 모듈을 include하도록 구조를 만들었고, shell.c에 있던 char\_to\_hex 및 str\_to\_hex 함수들이 어셈블러의 동작에 매우 중요하기 때문에 char\_to\_int, str\_to\_int까지 함께 구현하여 base 모듈에 포함시켜 큰 모듈들에서 문제없이 사용할 수 있도록 하였습니다.

1. **모듈 정의**

**3.1. Base 모듈**

3.1.1 기능

각 모듈에서 필요한 기본 함수인 char\_to\_hex, str\_to\_hex, char\_to\_int, str\_to\_int 함수를 구현함으로써 주어진 입력을 자유로운 형태로 다룰 수 있게 하는 기능을 한다. 또한 중요한 부분인 예외처리를 쉽게 할 수 있게 돕는 기능을 한다.

3.1.2 사용 변수

|  |
| --- |
| 사용 변수 없음 |

3.1.3 함수 목록

|  |  |
| --- | --- |
| int char\_to\_hex(char c); | convert a character to hexadecimal number |
| int str\_to\_hex(char\* str); | convert a string to hexadecimal number |
| int char\_to\_int(char c); | convert a character to decimal number |
| int str\_to\_int(char\* str); | convert a string to decimal number |

**3.2. Assembly 모듈**

3.2.1 기능

Assembly 모듈의 기능은 .asm파일로 받아온 어셈블리 코드를 .obj 파일로 변환하는 기능을 하도록 구현하였다. Assembly 모듈은 크게 두개의 phase로 나뉘는데, 먼저 .asm파일에서 어셈블리 코드를 불러와 사용하기 용이한 형태로 파싱하는 작업과, 이 데이터를 이용하여 object 코드로 변환하는 형태이다.

3.2.2 사용 변수

|  |
| --- |
| symbol table과 assembly 코드 각 라인의 데이터를 효과적으로 분류/저장하기 위한 구조체를 구현하였다.  //node structure for symbol table  typedef struct symbol\_node {  char name[20]; //symbol name  int LOC; //symbol location  struct symbol\_node\* next; //pointer for struct linked list  }symbol\_node;  //node structure for each assembly code line  typedef struct line\_node {  int idx; //line number.  int LOC; //Location counter  int PC; //Program counter  int format; //1, 2, 3, 4 format  int object\_len; //object\_code length  char raw[100]; //raw string of assembly code  bool is\_symbol; //flag if node has a symbol  bool is\_start; //flag if node is start node  bool is\_end; //flag if node is end node  bool is\_base; //flag if node is base node  bool is\_const; //flag if node is "BYTE" or "WORD"  bool is\_var; //flag if node is "RESW" or "RESB"  bool is\_opcode; //flag if node has an instruction  char param1[10]; //parameter1  char param2[10]; //parameter2  char param3[10]; //parameter3  char param4[10]; //parameter4  char object\_code[8]; //object code. maximum length = 8  struct line\_node\* next; //pointer pointing next node for struct linked list.  }line\_node; |

3.2.3 함수 목록

|  |  |
| --- | --- |
| bool in\_range(int disp); | disp가 12bit 범위 내에 있는지 판단하는 함수 |
| int register\_num(char\*c ); | 9개의 각 register에 맞는 번호를 return하는 함수 |
| void clear\_symbol(struct symbol\_node\* root); | 프로그램 종료 전 symbol table의 메모리를 해제하는 함수 |
| void clear\_line(struct line\_node\* root); | 역시 프로그램 종료 전 line node들의 메모리를 해제하는 함수 |
| void clear\_assembler(); | clear symbol과 clear line 을 함께 처리하는 함수 |
| void insert\_symbol(char\* name, int LOC); | 새로운 symbol이 발견되었을 때 오름차순으로 정렬된 linked list 안에 새롭게 insert하는 함수 |
| int find\_symbol(char\* name); | symbol의 LOC를 찾아 return하는 함수 |
| int symbol(void); | 마지막에 어셈블된 어셈블리 코드의 symbol table을 출력하는 함수 |
| void insert\_line(struct line\_node\* new\_); | 어셈블리 코드를 한 줄씩 읽어가며 |
| int pass1(char\* filename); | 처음 한 번 전체 어셈블리 코드를 읽으며 각 symbol들을 저장하고 line의 중요한 정보들을 파싱하는 함수 |
| int pass2(char\* filename); | 파싱한 데이터를 기반으로 각 assembly code에 대응하는 object code를 생성하는 함수 |
| void make\_lst(char\* filename); | 생성된 object code를 포함한 전체 translation list 파일을 생성하는 함수 |
| void make\_obj(char\* filename); | 헤더 형식에 맞게 object 파일을 생성하는 함수 |
| int assemble(char\* filename); | assembler 에서 수행하는 모든 작업에 대한 전체 flow를 담당하는 함수 |

1. **전역 변수 정의**

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

1. **알고리즘 / 구현 설명**

기본적으로 교재와 강의자료에 설명된 내용에 따라 진행했습니다. 1차 프로젝트 때 미처 생각하지 못한 부분인 shell 모듈에 기본 함수들을 포함시켜 놓은 것이 첫 번째 난관이었고, 상위 모듈에서 str\_to\_hex 등의 함수를 이용하지 못하는 상황이었습니다. 이를 해결하기 위해 먼저 전체 코드 구조를 2.1 flow chart와 같이 변경하였고, 그 뒤로 순조롭게 진행할 수 있었습니다.

각 라인에서 에러가 발생하여 assemble 함수가 종료되면 list\_root나 symbol\_root에 그동안 담았던 데이터도 전부 삭제해야 하기 때문에 pass 1, pass 2가 문제없이 종료된 경우에만 실제 list\_root 및 symbol\_root에 코드 데이터를 저장할 수 있게 했습니다.

Assemble 과정은 먼저 어셈블리 코드를 전체적으로 훑으며 각 line의 LOC와 PC 정보를 모으고, 나아가 현재 라인의 format, 그에 따른 object code 길이, symbol 포함 여부, 전체 코드 내용 등 활용할 수 있는 데이터는 전부 저장하도록 구현했습니다. 이 덕에 pass2 에서 object code를 얻기가 훨씬 수월했다고 생각합니다.

실제 object code를 얻는 과정은 까다로운 case 처리들이 필요했는데, 먼저 n, I, x, b, p, e 레지스터 값들을 전부 형식에 맞게 특정해야 했습니다. 이에 따라 먼저 symbol을 포함하고 있는지 여부를 기준으로 1차 분류를 했습니다. Symbol이 있는 경우와 없는 경우의 차이점은 parameter가 1개 더 있냐 없냐만의 차이였기 때문에 같은 코드 구조로 진행할 수 있어 이후에 수월하게 case 분류를 할 수 있었습니다.

두번째로는 opcode와 아닌 것, opcode 중에서는 각각의 형식에 따라 분류하여 처리했습니다. 마지막으로 addressing mode를 기준으로 분류하고, disp 로 들어온 값과 PC, Base register와의 관계를 검사해 b, p, bit 값을 결정했습니다. X, E 레지스터는 다른 조건들과 독립적인 조건을 가지기 때문에 가장 마지막에 처리했습니다.

이렇게 얻은 object 코드를 가지고 두 파일을 생성했는데, list 파일은 데이터를 보여주는 작업뿐이라 크게 고려할 부분은 없었고, obj 파일은 어느 정도 encoding을 하는 작업이기 때문에 구현 방식에 조금 어려움을 겪었습니다. 특히 Text header는 각 row의 byte수를 실제 값보다 먼저 출력해야 하기 때문에 처리가 어려웠는데, 한 줄에 들어갈 수 있는 최대 byte 수가 255 byte로 비교적 넉넉했기 때문에, object code가 없는 node를 만날 때만 줄바꿈을 하는 식으로 구현했습니다.

1. **코드 및 구현 설명**

**5.1 Base.h**

|  |
| --- |
| #include "error.h"  #include <string.h>  #include <stdlib.h>  int char\_to\_hex(char c);  int str\_to\_hex(char\* str);  int char\_to\_int(char c);  int str\_to\_int(char\* str); |

**5.2 Base.c**

|  |
| --- |
| #include "base.h"  //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;  }  //convert each character with corresponding decimal value  int char\_to\_int(char c) {  if('0' <= c && c <= '9') return c - '0';  return IMPOSSIBLE;  }  //convert string with corresponding decimal value  int str\_to\_int(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 \*= 10;  tmp = char\_to\_int(str[i]);  //IMPOSSIVLE means incorrect format decimal like AA  if(tmp == IMPOSSIBLE) return IMPOSSIBLE;  res += tmp;  }  return res;  } |

**5.3 Assembler.h**

|  |
| --- |
| #include "memory.h"  #include "opcode.h"  //node structure for symbol table  typedef struct symbol\_node {  char name[20]; //symbol name  int LOC; //symbol location  struct symbol\_node\* next; //pointer for struct linked list  }symbol\_node;  //node structure for each assembly code line  typedef struct line\_node {  int idx; //line number.  int LOC; //Location counter  int PC; //Program counter  int format; //1, 2, 3, 4 format  int object\_len; //object\_code length  char raw[100]; //raw string of assembly code  bool is\_symbol; //flag if node has a symbol  bool is\_start; //flag if node is start node  bool is\_end; //flag if node is end node  bool is\_base; //flag if node is base node  bool is\_const; //flag if node is "BYTE" or "WORD"  bool is\_var; //flag if node is "RESW" or "RESB"  bool is\_opcode; //flag if node has an instruction  char param1[10]; //parameter1  char param2[10]; //parameter2  char param3[10]; //parameter3  char param4[10]; //parameter4  char object\_code[8]; //object code. maximum length = 8  struct line\_node\* next; //pointer pointing next node for struct linked list.  }line\_node;  //assembler functions. Will explain at assembler.c  bool in\_range(int disp);  int register\_num(char\*c );  void clear\_symbol(struct symbol\_node\* root);  void clear\_line(struct line\_node\* root);  void clear\_assembler();  void insert\_symbol(char\* name, int LOC);  int find\_symbol(char\* name);  int symbol(void);  void insert\_line(struct line\_node\* new\_);  int pass1(char\* filename);  int pass2(char\* filename);  void make\_lst(char\* filename);  void make\_obj(char\* filename);  int assemble(char\* filename); |

**5.4 Assembler.c**

|  |
| --- |
| #include "assembler.h"  //declare symbol table, line list roots  struct symbol\_node\* sym\_root = NULL;  struct line\_node\* line\_root = NULL;  //declare symbol table, line list temporary roots  struct symbol\_node\* sym\_tmp = NULL;  struct line\_node\* line\_tmp = NULL;  int start\_addr = 0; //start address  int base\_LOC = 0; //base register  int LOCCTR = 0; //caculating current LOC  int line = 5; //line index  int prog\_len = 0; //program length  //return whether disp can be represented with 12 bits  bool in\_range(int disp) {  return disp >= -2048 && disp <= 2047;  }  //return corresponding register number  int register\_num(char\* c) {  if(c == NULL) return ERROR;  if(!strcmp(c, "A")) return 0;  if(!strcmp(c, "X")) return 1;  if(!strcmp(c, "L")) return 2;  if(!strcmp(c, "PC")) return 8;  if(!strcmp(c, "SW")) return 9;  if(!strcmp(c, "B")) return 3;  if(!strcmp(c, "S")) return 4;  if(!strcmp(c, "T")) return 5;  if(!strcmp(c, "F")) return 6;  return ERROR;  }  //clear allocated memories for symbol table  void clear\_symbol(struct symbol\_node\* root) {  struct symbol\_node\* tmp;  while(root != NULL) {  tmp = root;  root = root->next;  free(tmp);  }  }  //clear allocated memories for line\_data  void clear\_line(struct line\_node\* root) {  struct line\_node\* tmp;  while(root != NULL) {  tmp = root;  root = root->next;  free(tmp);  }  }  //clear all allocated memories for assembler  void clear\_assembler() {  clear\_symbol(sym\_tmp);  clear\_line(line\_tmp);  clear\_symbol(sym\_root);  clear\_line(line\_root);  return;  }  //insert symbol data by increasing order  void insert\_symbol(char\* name, int LOC) {  struct symbol\_node\* new\_ = (struct symbol\_node\*)malloc(sizeof(symbol\_node));  new\_->next = NULL;  strcpy(new\_->name, name);  new\_->LOC = LOC;    struct symbol\_node\* pos = sym\_tmp;  int cnt = 0;  while(pos != NULL && strcmp(pos->name, name) < 0) {  pos = pos->next;  cnt++;  }  //if new\_ be root  if(cnt == 0) {  new\_->next = sym\_tmp;  sym\_tmp = new\_;  return;  }  //else  cnt--;  pos = sym\_tmp;  while(cnt--) pos = pos->next;  new\_->next = pos->next;  pos->next = new\_;  return;  }  //find LOC of symbol  int find\_symbol(char\* name) {  if(name == NULL) return ERROR;  struct symbol\_node\* tmp = sym\_tmp;  while(tmp != NULL) {  if(!strcmp(tmp->name, name)) {  return tmp->LOC;  }  tmp = tmp->next;  }  return ERROR;  }  //symbol function : print symbol name and LOC by order  int symbol() {  if(sym\_root == NULL) {  printf("Error : no symbol!\n");  return ERROR;  }  struct symbol\_node\* tmp = sym\_root;  while(tmp != NULL) {  printf("\t%s\t%04X\n", tmp->name, tmp->LOC);  tmp = tmp->next;  }  return SUCCESS;  }  //insert new line node  void insert\_line(struct line\_node\* new\_) {  if(line\_tmp == NULL) {  line\_tmp = new\_;  return;  }  struct line\_node\* tmp = line\_tmp;  while(tmp->next != NULL) tmp = tmp->next;  tmp->next = new\_;  return;  }  //pass1 : parse each line of asm file and save symbol table / line data  int pass1(char\* filename) {  int len = strlen(filename);  FILE\* fp = NULL;  fp = fopen(filename, "r");  //if not in directory or not asm file, return ERROR  if(fp == NULL || len <= 3 || !(filename[len-4] == '.' && filename[len-3] == 'a' && filename[len-2] == 's' && filename[len-1] == 'm')) {  printf("Error : Invalid filename\n");  return ERROR;  }  //input current line  char cur\_line[100];  char raw\_line[100];  while(1) {  fgets(cur\_line, 100, fp);  cur\_line[(int)strlen(cur\_line)-1] = 0;  char \*param1, \*param2, \*param3, \*param4;  strcpy(raw\_line, cur\_line);  //parse data  param1 = strtok(cur\_line, " ");  param2 = strtok(NULL, " ");  param3 = strtok(NULL, " ");  param4 = strtok(NULL, " ");    //initialize new line node  struct line\_node\* new\_ = (struct line\_node\*)malloc(sizeof(line\_node));    int j;  new\_->next = NULL;  new\_->idx = line;  new\_->LOC = 0;  new\_->PC = 0;  new\_->format = 0;  new\_->object\_len = 0;  for(j=0;j<100;j++) new\_->raw[j] = 0;  for(j=0;j<10;j++) new\_->param1[j] = 0;  for(j=0;j<10;j++) new\_->param2[j] = 0;  for(j=0;j<10;j++) new\_->param3[j] = 0;  for(j=0;j<10;j++) new\_->param4[j] = 0;  for(j=0;j<8;j++) new\_->object\_code[j] = 0;  //may have to debug    if(raw\_line != NULL) strcpy(new\_->raw, raw\_line);  if(param1 != NULL) strcpy(new\_->param1, param1);  if(param2 != NULL) strcpy(new\_->param2, param2);  if(param3 != NULL) strcpy(new\_->param3, param3);  if(param4 != NULL) strcpy(new\_->param4, param4);  new\_->is\_symbol = false;  new\_->is\_start = false;  new\_->is\_end = false;  new\_->is\_base = false;  new\_->is\_opcode = false;  new\_->is\_const = false;  new\_->is\_var = false;    if(!strcmp(param1, "END")) {  new\_->is\_end = true;  prog\_len = LOCCTR - start\_addr;  insert\_line(new\_);  break;  }  //comment line  else if(cur\_line[0] == '.') {  continue;  }  //if start node, init  else if(param2 != NULL && !strcmp(param2, "START")) {  new\_->is\_start = true;  start\_addr = str\_to\_hex(param3);  line = 5;  base\_LOC = 0;  new\_->idx = line;  LOCCTR = start\_addr;  insert\_line(new\_);  line += 5;  continue;  }  //if base node, update base register value  else if(!strcmp(param1, "BASE")) {  new\_->is\_base = true;  new\_->idx = line;  insert\_line(new\_);  line += 5;  continue;  }    else {  //have symbol  if(raw\_line[0] != ' '){  new\_->is\_symbol = true;  int loc = find\_symbol(param1);  //if there exists same symbol  if(loc != ERROR) {  printf("line %d Error : Duplicate Symbol!\n", line);  return ERROR;  }  //new symbol  else {  insert\_symbol(param1, LOCCTR);  }  //opcode  if(get\_format(param2) != ERROR ) {  new\_->is\_opcode = true;  new\_->LOC = LOCCTR;  LOCCTR += get\_format(param2);  new\_->PC = LOCCTR;  new\_->format = get\_format(param2);  insert\_line(new\_);  }  else if(!strcmp(param2, "WORD")) {  new\_->is\_const = true;  new\_->LOC = LOCCTR;  LOCCTR += 3;  new\_->PC = LOCCTR;  insert\_line(new\_);  }  else if(!strcmp(param2, "RESW")) {  new\_->is\_var = true;  new\_->LOC = LOCCTR;  LOCCTR += 3 \* str\_to\_int(param3);  new\_->PC = LOCCTR;  insert\_line(new\_);  }  else if(!strcmp(param2, "RESB")) {  new\_->is\_var = true;  new\_->LOC = LOCCTR;  LOCCTR += str\_to\_int(param3);  new\_->PC = LOCCTR;  insert\_line(new\_);  }  else if(!strcmp(param2, "BYTE")) {  new\_->is\_const = true;  new\_->LOC = LOCCTR;  if(param3[0] == 'C') LOCCTR += (int)strlen(param3) - 3;  else LOCCTR += ((int)strlen(param3) - 3) / 2;  new\_->PC = LOCCTR;  insert\_line(new\_);  }  else {  printf("line %d Error : invalid operation code!\n", line);  return ERROR;  }  }  //no symbol  else {  if(get\_format(param1) != ERROR) {  new\_->is\_opcode = true;  new\_->LOC = LOCCTR;  LOCCTR += get\_format(param1);  new\_->PC = LOCCTR;  new\_->format = get\_format(param1);  insert\_line(new\_);  }  else if(!strcmp(param1, "WORD")) {  new\_->is\_const = true;  new\_->LOC = LOCCTR;  LOCCTR += 3;  new\_->PC = LOCCTR;  insert\_line(new\_);  }  else if(!strcmp(param1, "RESW")) {  new\_->is\_var = true;  new\_->LOC = LOCCTR;  LOCCTR += 3 \* str\_to\_int(param2);  new\_->PC = LOCCTR;  insert\_line(new\_);  }  else if(!strcmp(param1, "RESB")) {  new\_->is\_var = true;  new\_->LOC = LOCCTR;  LOCCTR += str\_to\_int(param2);  new\_->PC = LOCCTR;  insert\_line(new\_);  }  else if(!strcmp(param1, "BYTE")) {  new\_->is\_const = true;  new\_->LOC = LOCCTR;  if(param2[0] == 'C') LOCCTR += (int)strlen(param2) - 3;  else LOCCTR += ((int)strlen(param2) - 3) / 2;  new\_->PC = LOCCTR;  insert\_line(new\_);  }  else {  printf("line %d Error : invalid operation code!\n", line);  return ERROR;  }  }  }  line += 5;  }  fclose(fp);  return SUCCESS;  }  //pass2 : make opcode for each line with corresponding line data and symbol table  int pass2(char\* filename) {  struct line\_node\* tmp = line\_tmp;  while(tmp != NULL) {  //have symbol  if(tmp->is\_symbol) {  //no need to make object code  if(tmp->is\_start || tmp->is\_base || tmp->is\_end || tmp->is\_var) {  if(tmp->is\_base) {  base\_LOC = find\_symbol(tmp->param3);  }  tmp->object\_len = 0;  tmp = tmp->next;  continue;  }  //opcode  if(tmp->is\_opcode) {  tmp->object\_len = (tmp->format) \* 2;  //separate cases by format  //format 1: 1byte  if(tmp->format == 1) {  int opval = opname\_to\_hex(tmp->param2);  tmp->object\_code[0] = opval / 16;  tmp->object\_code[1] = opval % 16;  }  //format 2: 2bytes  else if(tmp->format == 2) {  int opval = opname\_to\_hex(tmp->param2);  int r1 = register\_num(tmp->param3);  int r2 = register\_num(tmp->param4);  tmp->object\_code[0] = opval / 16;  tmp->object\_code[1] = opval % 16;  tmp->object\_code[2] = r1 == ERROR ? 0 : r1;  tmp->object\_code[3] = r2 == ERROR ? 0 : r2;  }  //format 3: 3bytes  else if(tmp->format == 3) {  int len = strlen(tmp->param2);  if(tmp->param2[len-1] == ',') tmp->param2[len-1] = 0;  len = strlen(tmp->param3);  if(tmp->param3[len-1] == ',') tmp->param3[len-1] = 0;  int opval = opname\_to\_hex(tmp->param2);  int disp;  tmp->object\_code[0] = opval / 16;  tmp->object\_code[1] = opval % 16;    int n = 0, i = 0, x = 0, b = 0, p = 0, e = 0;  bool flag = false;  //check addressing mode  //immediate addressing  if(tmp->param3[0] == '#') {  n = 0, i = 1;  disp = find\_symbol(tmp->param3+1);  if(disp == ERROR) {  disp = str\_to\_int(tmp->param3+1);  flag = true;  }  }  //indirect addressing  else if(tmp->param3[0] == '@') {  n = 1, i = 0;  disp = find\_symbol(tmp->param3+1);  if(disp == ERROR) {  disp = str\_to\_int(tmp->param3+1);  flag = true;  }  }  //simple addressing  else {  n = 1, i = 1;  disp = find\_symbol(tmp->param3);  if(disp == ERROR){  disp = str\_to\_int(tmp->param3);  flag = true;  }  }    //if disp is constant, no need PC or base relative mode  if(!flag) {  disp -= tmp->PC;  if(in\_range(disp)) {  p = 1;  }  else {  disp += tmp->PC;  disp -= base\_LOC;  b = 1;  }  }    if(tmp->param4[0] == 'X') {  x = 1;  }  //2's complement  if(disp < 0 ) disp += 4096;  tmp->object\_code[1] += n \* 2;  tmp->object\_code[1] += i;  tmp->object\_code[2] = 8\*x + 4\*b + 2\*p + e;  tmp->object\_code[3] = disp / 256;  disp -= tmp->object\_code[3] \* 256;  tmp->object\_code[4] = disp / 16;  disp -= tmp->object\_code[4] \* 16;  tmp->object\_code[5] = disp;  }  else if(tmp->format == 4) {  int len = strlen(tmp->param2);  if(tmp->param2[len-1] == ',') tmp->param2[len-1] = 0;  len = strlen(tmp->param3);  if(tmp->param3[len-1] == ',') tmp->param3[len-1] = 0;  int opval = opname\_to\_hex(tmp->param2+1);  int disp;  tmp->object\_code[0] = opval / 16;  tmp->object\_code[1] = opval % 16;    int n = 0, i = 0, x = 0, b = 0, p = 0, e = 0;  //check addressing mode  //immediate addressing  if(tmp->param3[0] == '#') {  n = 0, i = 1;  disp = find\_symbol(tmp->param3+1);  if(disp == ERROR) disp = str\_to\_int(tmp->param3+1);  }  //indirect addressing  else if(tmp->param3[0] == '@') {  n = 1, i = 0;  disp = find\_symbol(tmp->param3+1);  if(disp == ERROR) disp = str\_to\_int(tmp->param3+1);  }  //simple addressing  else {  n = 1, i = 1;  disp = find\_symbol(tmp->param3);  if(disp == ERROR) disp = str\_to\_int(tmp->param3);  }  if(tmp->param4[0] == 'X') {  x = 1;  }    e = 1;  tmp->object\_code[1] += n \* 2;  tmp->object\_code[1] += i;  tmp->object\_code[2] = 8\*x + 4\*b + 2\*p + e;    tmp->object\_code[3] = disp / 65536;  disp -= tmp->object\_code[3] \* 65536;  tmp->object\_code[4] = disp / 4096;  disp -= tmp->object\_code[4] \* 4096;  tmp->object\_code[5] = disp / 256;  disp -= tmp->object\_code[5] \* 256;  tmp->object\_code[6] = disp / 16;  disp -= tmp->object\_code[6] \* 16;  tmp->object\_code[7] = disp;  }  }  //constants  else {  int disp;  //BYTE  if(!strcmp(tmp->param2, "BYTE")) {  if(tmp->param3[0] == 'C') {  tmp->object\_len = 6;  int c1 = tmp->param3[2];  int c2 = tmp->param3[3];  int c3 = tmp->param3[4];  tmp->object\_code[0] = c1/16;  tmp->object\_code[1] = c1%16;  tmp->object\_code[2] = c2/16;  tmp->object\_code[3] = c2%16;  tmp->object\_code[4] = c3/16;  tmp->object\_code[5] = c3%16;    }  else if(tmp->param3[0] == 'X') {  tmp->object\_len = 2;  tmp->object\_code[0] = char\_to\_hex(tmp->param3[2]);  tmp->object\_code[1] = char\_to\_hex(tmp->param3[3]);  }  }  //WORD  else {  tmp->object\_len = 6;  disp = str\_to\_hex(tmp->param3);  tmp->object\_code[0] = disp / 1048576;  disp -= tmp->object\_code[0] \* 1048576;  tmp->object\_code[1] = disp / 65536;  disp -= tmp->object\_code[1] \* 65536;  tmp->object\_code[2] = disp / 4096;  disp -= tmp->object\_code[2] \* 4096;  tmp->object\_code[3] = disp / 256;  disp -= tmp->object\_code[3] \* 256;  tmp->object\_code[4] = disp / 16;  disp -= tmp->object\_code[4] \* 16;  tmp->object\_code[5] = disp;  }  }  }  //no symbol  else {  //no need to make object code  if(tmp->is\_start || tmp->is\_base || tmp->is\_end || tmp->is\_var) {  if(tmp->is\_base) {  base\_LOC = find\_symbol(tmp->param2);  }  tmp->object\_len = 0;  tmp = tmp->next;  continue;  }  //first 6 bits will be filled with opcode number  //next bits will be filled with n, i, x, b, p, e register value  if(tmp->is\_opcode) {  tmp->object\_len = (tmp->format) \* 2;  if(tmp->format == 1) {  int opval = opname\_to\_hex(tmp->param1);  tmp->object\_code[0] = opval / 16;  tmp->object\_code[1] = opval % 16;  }  else if(tmp->format == 2) {  int opval = opname\_to\_hex(tmp->param1);  int r1 = register\_num(tmp->param2);  int r2 = register\_num(tmp->param3);  tmp->object\_code[0] = opval / 16;  tmp->object\_code[1] = opval % 16;  tmp->object\_code[2] = r1 == ERROR ? 0 : r1;  tmp->object\_code[3] = r2 == ERROR ? 0 : r2;  }  else if(tmp->format == 3) {  int len = strlen(tmp->param1);  if(tmp->param1[len-1] == ',') tmp->param1[len-1] = 0;  len = strlen(tmp->param2);  if(tmp->param2[len-1] == ',') tmp->param2[len-1] = 0;    int opval = opname\_to\_hex(tmp->param1);  int disp;  tmp->object\_code[0] = opval / 16;  tmp->object\_code[1] = opval % 16;    int n = 0, i = 0, x = 0, b = 0, p = 0, e = 0;  bool flag = false;  //check addressing mode  //immediate addressing  if(tmp->param2[0] == '#') {  n = 0, i = 1;  disp = find\_symbol(tmp->param2+1);  if(disp == ERROR) {  disp = str\_to\_int(tmp->param2+1);  flag = true;  }  }  //indirect addressing  else if(tmp->param2[0] == '@') {  n = 1, i = 0;  disp = find\_symbol(tmp->param2+1);  if(disp == ERROR) {  disp = str\_to\_int(tmp->param2+1);  flag = true;  }  }  //simple addressing  else {  n = 1, i = 1;  disp = find\_symbol(tmp->param2);  if(disp == ERROR) {  disp = str\_to\_int(tmp->param2);  flag = true;  }  }    if(!flag) {  disp -= tmp->PC;  if(in\_range(disp)) {  p = 1;  }  else {  disp += tmp->PC;  disp -= base\_LOC;  b = 1;  }  }  if(tmp->param3[0] == 'X') {  x = 1;  }    if(disp < 0 ) disp += 4096;  tmp->object\_code[1] += n \* 2;  tmp->object\_code[1] += i;  tmp->object\_code[2] = 8\*x + 4\*b + 2\*p + e;  tmp->object\_code[3] = disp / 256;  disp -= tmp->object\_code[3] \* 256;  tmp->object\_code[4] = disp / 16;  disp -= tmp->object\_code[4] \* 16;  tmp->object\_code[5] = disp;  }  else if(tmp->format == 4) {  int len = strlen(tmp->param1);  if(tmp->param1[len-1] == ',') tmp->param1[len-1] = 0;  len = strlen(tmp->param2);  if(tmp->param2[len-1] == ',') tmp->param2[len-1] = 0;    int opval = opname\_to\_hex(tmp->param1+1);  int disp;  tmp->object\_code[0] = opval / 16;  tmp->object\_code[1] = opval % 16;    int n = 0, i = 0, x = 0, b = 0, p = 0, e = 0;  //check addressing mode  //immediate addressing  if(tmp->param2[0] == '#') {  n = 0, i = 1;  disp = find\_symbol(tmp->param2+1);  if(disp == ERROR) disp = str\_to\_int(tmp->param2+1);  }  //indirect addressing  else if(tmp->param2[0] == '@') {  n = 1, i = 0;  disp = find\_symbol(tmp->param2+1);  if(disp == ERROR) disp = str\_to\_int(tmp->param2+1);  }  //simple addressing  else {  n = 1, i = 1;  disp = find\_symbol(tmp->param2);  if(disp == ERROR) disp = str\_to\_int(tmp->param2);  }  if(tmp->param3[0] == 'X') {  x = 1;  }    e = 1;  tmp->object\_code[1] += n \* 2;  tmp->object\_code[1] += i;  tmp->object\_code[2] = 8\*x + 4\*b + 2\*p + e;    tmp->object\_code[3] = disp / 65536;  disp -= tmp->object\_code[3] \* 65536;  tmp->object\_code[4] = disp / 4096;  disp -= tmp->object\_code[4] \* 4096;  tmp->object\_code[5] = disp / 256;  disp -= tmp->object\_code[5] \* 256;  tmp->object\_code[6] = disp / 16;  disp -= tmp->object\_code[6] \* 16;  tmp->object\_code[7] = disp;  }  }  //constants  else {  int disp;  //BYTE  if(!strcmp(tmp->param1, "BYTE")) {  if(tmp->param2[0] == 'C') {  tmp->object\_len = 6;  int c1 = tmp->param2[2];  int c2 = tmp->param2[3];  int c3 = tmp->param2[4];  tmp->object\_code[0] = c1/16;  tmp->object\_code[1] = c1%16;  tmp->object\_code[2] = c2/16;  tmp->object\_code[3] = c2%16;  tmp->object\_code[4] = c3/16;  tmp->object\_code[5] = c3%16;    }  else if(tmp->param2[0] == 'X') {  tmp->object\_len = 2;  tmp->object\_code[0] = char\_to\_hex(tmp->param2[2]);  tmp->object\_code[1] = char\_to\_hex(tmp->param2[3]);  }  }  //WORD  else {  tmp->object\_len = 6;  disp = str\_to\_hex(tmp->param2);  tmp->object\_code[0] = disp / 1048576;  disp -= tmp->object\_code[0] \* 1048576;  tmp->object\_code[1] = disp / 65536;  disp -= tmp->object\_code[1] \* 65536;  tmp->object\_code[2] = disp / 4096;  disp -= tmp->object\_code[2] \* 4096;  tmp->object\_code[3] = disp / 256;  disp -= tmp->object\_code[3] \* 256;  tmp->object\_code[4] = disp / 16;  disp -= tmp->object\_code[4] \* 16;  tmp->object\_code[5] = disp;  }  }  }  tmp = tmp->next;  }  return SUCCESS;  }  //after parse assembly code and translate to object code, we'll write list file  void make\_lst(char\* filename) {  FILE\* fp;  int len = strlen(filename);  filename[len-3] = 'l';  filename[len-2] = 's';  filename[len-1] = 't';  fp = fopen(filename, "w");  //nothing to explain. Just format  struct line\_node\* tmp1 = line\_root;  while(tmp1 != NULL) {  if(tmp1->is\_base || tmp1->is\_end) fprintf(fp, "%d\t\t%s\n", tmp1->idx, tmp1->raw);  else if(tmp1->is\_start || tmp1->is\_var) fprintf(fp, "%d\t%04X\t%s\n", tmp1->idx, tmp1->LOC, tmp1->raw);  else {  fprintf(fp, "%d\t%04X\t%-25s\t", tmp1->idx, tmp1->LOC, tmp1->raw);  int i;  for(i=0;i<tmp1->object\_len;i++) fprintf(fp, "%X", tmp1->object\_code[i]);  fprintf(fp, "\n");  }  tmp1 = tmp1->next;  }  fclose(fp);  }  //finally, write obj file.  void make\_obj(char\* filename){  FILE\* fp;  int len = strlen(filename);  filename[len-3] = 'o';  filename[len-2] = 'b';  filename[len-1] = 'j';  fp = fopen(filename, "w");    //H header  struct line\_node\* tmp = line\_root;  fprintf(fp, "H%-6s%06X%06X\n", tmp->param1, start\_addr, prog\_len);    //T header  tmp = line\_root;  int arr[500], i;    while(tmp != NULL) {  while(tmp != NULL && tmp->object\_len == 0) tmp = tmp->next;  if(tmp == NULL) break;  int sLOC = tmp->LOC;  fprintf(fp, "T%06X", sLOC);  int cnt = 0;  while(tmp != NULL && tmp->object\_len > 0) {  for(i = 0; i< tmp->object\_len; i++) {  arr[cnt++] = tmp->object\_code[i];  }  tmp = tmp->next;  }  fprintf(fp, "%02X", cnt/2);  for(i = 0; i < cnt; i++) fprintf(fp, "%X", arr[i]);  fprintf(fp, "\n");  }  //M header  tmp = line\_root;  while(tmp!=NULL) {  if(tmp->format == 4 && !(tmp->is\_symbol && tmp->param3[0] == '#') && !(!tmp->is\_symbol && tmp->param2[0] == '#')) {  fprintf(fp, "M%06X05\n", tmp->LOC + 1);  }  tmp = tmp->next;  }  //E header  tmp = line\_root;  int exec\_LOC;  while(tmp->object\_len == 0) tmp = tmp->next;  exec\_LOC = tmp->LOC;  fprintf(fp, "E%06X", exec\_LOC);  fclose(fp);  filename[len-3] = 'a';  filename[len-2] = 's';  filename[len-1] = 'm';  }  //assemble .asm file  //shorten total flow of assembler  int assemble(char\* filename) {  //pass 1  int err = pass1(filename);  if(err == ERROR) {  clear\_symbol(sym\_tmp);  clear\_line(line\_tmp);  return ERROR;  }  //pass2  err = pass2(filename);  if(err == ERROR) {  clear\_symbol(sym\_tmp);  clear\_line(line\_tmp);  return ERROR;  }  //no error occurs  else {  clear\_symbol(sym\_root);  clear\_line(line\_root);  sym\_root = sym\_tmp;  line\_root = line\_tmp;  sym\_tmp = NULL;  line\_tmp = NULL;    //make lst, obj files  make\_lst(filename);  make\_obj(filename);  printf("Successfully assemble %s!\n", filename);  return SUCCESS;  }  return ERROR;  } |

* 1. **Opcode.c 추가 함수**

|  |
| --- |
| 1. //find matching opcode with input opcode\_name 2. int opname\_to\_hex(char\* str) { 3. if(str == NULL){ 4. return ERROR; 5. } 6. //get hash index 7. int idx = (str[0] - 'A') % 20; 8. struct hash\_node\* tmp = hash\_head[idx]; 9. //find corresponding node while iterating 10. while(tmp != NULL) { 11. if(!strcmp(tmp->name, str)) { 12. return str\_to\_hex(tmp->code); 13. } 14. tmp = tmp->next; 15. } 16. return ERROR; 17. } 18. //find matching opcode format with input opcode\_name 19. int get\_format(char\* str) { 20. if(str == NULL){ 21. return ERROR; 22. } 23. if(str[0] == '+') { 24. //get hash index 25. int idx = (str[1] - 'A') % 20; 26. struct hash\_node\* tmp = hash\_head[idx]; 27. //find corresponding node while iterating 28. while(tmp != NULL) { 29. if(!strcmp(tmp->name, str + 1)) { 30. if(strcmp(tmp->data, "3/4") == 0) return 4; 31. else return ERROR; 32. } 33. tmp = tmp->next; 34. } 35. return ERROR; 36. } 37. //get hash index 38. int idx = (str[0] - 'A') % 20; 39. struct hash\_node\* tmp = hash\_head[idx]; 40. //find corresponding node while iterating 41. while(tmp != NULL) { 42. if(!strcmp(tmp->name, str)) { 43. if(strcmp(tmp->data, "1") == 0) return 1; 44. else if(strcmp(tmp->data, "2") == 0) return 2; 45. else return 3; 46. } 47. tmp = tmp->next; 48. } 49. return ERROR; 50. } |