과목 명: 시스템프로그래밍

담당 교수 명: 김 지 환

<<Assignment 1>>

**서강대학교 컴퓨터학과**

**[학번] 20171667**

**[이름] 이용욱**

목 차

1. 프로그램 개요 3

2. 프로그램 설명 3

2.1 프로그램 흐름도 3

3. 모듈 정의 3

3.1 모듈 이름 : Main 3

3.1.1 기능 3

3.1.2 사용 변수 3

3.2 모듈 이름: Opcode 3

3.2.1 기능 3

3.2.2 사용 변수 3

3.3 모듈이름: Assembler 4

3.3.1 기능 4

3.3.2 사용변수 4

3.4 모듈이름: symbol 5

3.4.1 기능 5

3.4.2 사용변수 5

4. 전역 변수 정의 5

4.1 #define MAX\_OBJ\_PER\_LINE 60 5

4.2 #define MAX\_LINE\_LENGTH 200 5

4.3 #define NO\_REGISTRE -1 5

4.4 #define MAX\_LINE\_ASM 200 6

4.5 #define MAX\_LINE\_ASM 200 6

4.6 #define NOT\_FOUND -1 6

4.7 #define LENGTH\_SYMBOLTABLE 20 6

## 5 코드 7

5.1 20171667.c

5.2 opcode.c

5.3 assembler.c

5.4 assembler.h

5.5 symbol.c

5.6 symbol.h

# 프로그램 개요

프로젝트1에서 생성한 shell 환경에서 asm파일을 어셈블하는 기능을 만든다. 수업시간에 배운대로, SIC/XE명령을 Object code 로 변환한다. \*.asm파일이 주어지면, \*.lst파일과 \*.obj파일이 생성될 것 이며, 오류가 있는 경우 해당 라인 번호와 오류 메시지를 출력한다. 또한 파일의 내용을 출력하는 type명령과 , symbol들을 출력하는 symbol()명령어도 만든다.

# 프로그램 설명

## 프로그램 흐름도

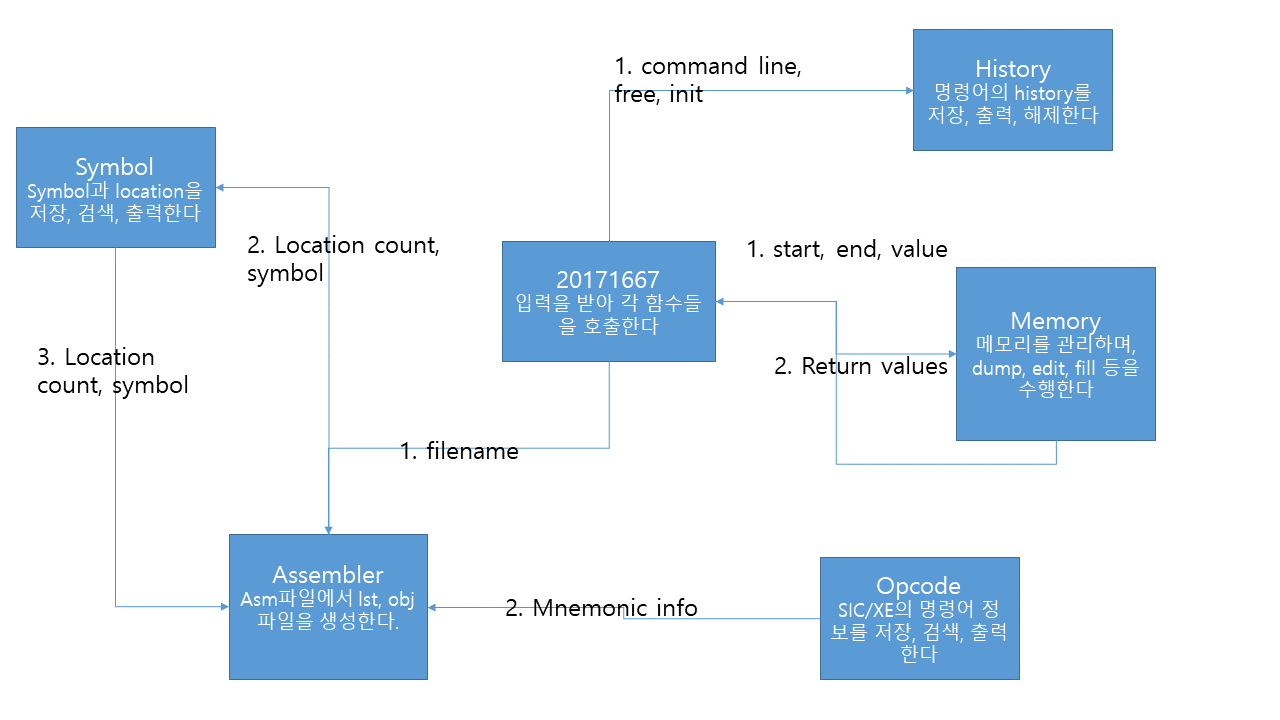


그림 1> 프로그램 흐름도

우선 main함수가 있는 Main모듈이 호출된다. Main모듈이지만 소스파일의 이름은 명세에서 요구하는 대로 20171667.c, 20171667.h로 하였다. 여기서는 명령을 입력받고, 이를 해석하여 해당하는 함수를 출력한다. 이 중 명령어의 history를 다루는 함수는 History모듈에, 메모리를 다루는 함수는 Memory모듈에, opcode를 다루는 함수는 Opcode 모듈에, 어셈블을 하는 함수는 Assembler 모듈에 속한다.

이번 과제의 핵심인 Assemble 모듈에서는, 파일의 이름을 Main으로부터 받아, 파일을 열고, pass1과 pass2를 진행하고, 이를 통해 lst파일과 obj 파일을 생성한다. 이 때 symbol을 저장해야 하는 경우에는, symbol을 다루는 Symbol모듈에 이를 넘겨준다. Opcode에 대해 정보가 필요한 경우에는 Opcode모듈에서 받아 온다.

Symbol모듈에서는 symbol의 이름과 location을 assembler로부터 받아 이를 헤시 테이블과 리스트에 저장하고 필요한 경우 symbol의 location값을 빠르게 반환한다.

또한 symbol명령시에는 저장된 symbol을 정렬하여 출력한다.

# 모듈 정의

## 모듈 이름 : Main (파일 이름은 20171667.c, 20171667.h)

### 기능

프로그램의 실행과 종료를 담당한다. 또한 명령어를 입력받고 parsing하여 해당하는 함수를 출력한다. 이 떄 명령어가 유효한지를 검사하여 syntax error를 출력할 수도 있다. Help나 quit, dir과 같은 일반적인 기능의 경우에는 Main모듈에 정의하였다.

이번 과제에서는 help함수에 명령어가 추가되었고, quit시에 symbol table을 해제하도록 수정되었다

### 사용 변수

프로젝트 1과 동일하다.

## 모듈 이름: Opcode

### 기능

Opcode.txt에서 SIC/XE에 관련한 정보를 읽어 이를 hashTable에 저장한다. 이후 다른 모듈에서 호출시마다. 해당하는 mnemonic의 정보를 반환한다. 또한 종료시에 hashTable의 free를 호출한다.

이번 과제에서는, getInst ()함수가 추가되었다. 이 함수는 mnemonic으로 추정되는 문자열을 받아서 이를 해당하는 instruction 구조체의 pointer를 반환한다. 해당 문자열이 opcode hashtable에 존재하지 않을 경우 NULL을 반환하여 이를 알리도록 하였다.

### 사용 변수

프로젝트 1과 동일

## 모듈이름: Assembler

### 기능

이번 프로젝트의 핵심이다. asm파일의 어셈블을 담당한다. Main모듈로부터 파일이름이 전달되면, 이 파일이 유효한 지 판단하고 어셈블을 시작한다.

어셈블 과정은 assemble함수에서 호출된 assemble\_pass1, assemble\_pass2, writeObj 함수들이 대체로 담당한다. pass1, pass2의 역할은 교제와 거의 같으나, 개발 편의상 obj코드를 쓰는 함수는 분리하였다.

Assemble\_pass1 함수에서는 asm파일로부터 코드를 읽어 pass1과정을 진행한다. 첫 줄에서START를 찾아 시작 location을 설정한다. 코드의 이름도 읽어온다. 그 뒤에는 asm파일을 한 줄씩 읽으며 각 줄을 분석하고, location값을 계산한다. 이때opcode모듈에서 format정보를 받아온다. label을 읽은 경우에는 symbol table에 해당 location과 함께 push한다. 또한 WORD, BYTE와 같은 경우에는 해당 데이터를 바로 16진수로 변환하여 저장한다. 이 결과는 모두 list에 저장된다. List는 list\_line구조체 배열로, 이 프로그램에서 중간 파일의 역할을 대신 한다.

Assemble\_pass2함수는 pass1에서 부분적으로 완성한 list를 넘겨 받아 lst파일을 만든다. 이 때에는 각 줄의 format을 파악하고addressing 방식, x레지스터 사용여부, nixbpe 플레그 등을 고려하여 최종적으로 obj 코드로 변환한다. 이 결과는 format{1, 2, 3, 4}구조체에서 이루어진다. Format 구조체는 bitfield와 int의 공용체이다. 각 비트를 변수처럼 접근할 수 있고, 동시에 최종 obj code도 계산할 수 있다. 이 결과를 바탕으로 lst파일을 작성한다.

writeObj 함수에서는 pass2에서 완성한 list파일에서 objcode를 추출하여 이를 양식에 맞추어 쓴다. 이 때에 printObj함수가 추출에 사용된다. 그 다음 list를 다시 읽으며 modify해야하는 주소값을 찾고 Modify code도 출력한다.

이 과정에서 error는 raiseError함수에 의해 출력되며, assemble과정을 종료하고 lst, obj파일은 삭제한다.

이 외에도 type명령을 수행한다. type함수에서는 해당 파일을 읽고 출력하는 역할을 한다.

### 사용변수

struct list\_line list[MAX\_LINE\_ASM]

asm파일의 각 줄의 정보를 저장하는 구조체의 배열이다. 전체 assemble 과정에서 공유된다. 해당 줄이 주석인지, obj code는 몇 바이트인지, location, lable, operand는 무엇인지, objcode는 무엇인지 저장한다.

enum assemble\_error 에러를 정의한 enum으로, raiseError에서 어떤 에러가 발생했는 지 알기 위해 사용한다.

Int line 해당 반복문이 몇번째 line을 다루고 있는 지 저장한다. 이때 1부터 1씩 증가한다. 따라서 list파일에서 출력할 때에는 5를 곱하여 출력한다.

FILE\* fp\_asm, fp\_lst, fp\_obj

각각 asm파일, lst파일, obj파일의 포인터이다.

Char filename\_asm [100], filename\_lst[100], filename\_obj[100]

파일의 이름을 저장할 문자열.

unsigned int loc = 0;

현제 location을 저장하는 변수

char buf[100];

입력을 받은 문자열을 저장할 문자열 버퍼

struct list\_line\* curr = &list[line++];

현제 list의 정보를 저장할 list\_line 구조체를 가르키는 포인터.

int num\_line

list파일의 줄 수를 저장하는 int변수.pass1에서 구한다.

Int B;

Base 레지스터의 값을 저장하는 변수. -1로 초기화하여 -1일때는 base relative에 사용되지 않는다.

struct format1(2, 3, 4) obj

pass2 에서는 해당하는 format의 구조체를 선언하고 이를 완성한다. 이 때에 format의 정보를 저장하는 변수.

### 사용 데이터 타입

enum assemble\_error

여러가지 assemble error를 표현하기 위해 사용

struct format1

format1을 표현하기 위해 사용한 bit field, opcode가 8비트를 사용한다.

struct format2

format2을 표현하기 위해 사용, opcode가 8비트, 두 레지스터가 5비트씩 사용한다.

struct format3

format3을 표현하기 위해 사용, opcode가 6비트, nixbpe flags가 1비트씩, disp가 12비트를 사용한다.

struct format4

format4을 표현하기 위해 사용, opcode가 6비트, nixbpe flags가 1비트씩, address가 20비트를 사용한다.

struct list\_line{

int isCommnet; // 주석 여부 저장

int obj\_size; // obj code 의 크기 저장

char\* str; // 주석인 경우 주석 저장

unsigned int loc; // location

char label[10]; // label

char mnemonic[10]; // mnemonic

char operand[10]; // operand

unsigned int obj; // objcode

char data[30+1]; // 상수 데이터인 경우 데이터 저장

}

Asm 파일의 한 라인을 표현하는 구조체이다

## 모듈이름: ****symbol****

### 기능

Symbol Table 을 관리한다. Symbol table을 초기화한다. assembler에서 symbol과 locatoin을 입력받아, list와 hashtable에 저장한다. Symbol을 찾아 location을 반환한다. symbol명령어 호출 시, list를 사전순의 역순으로 정렬하여 출력한다.

### 사용변수

node\_symbol\*\* symbolTable = NULL;

symbol을 저장할 hash table

int num\_symbol = 0;

입력받은 symbol의 총 개수

struct node\_symbol\* symbol\_list = NULL;

symbol을 저장할 list

int target\_idx = hash(symbol);

symbol이 저장될 table 상 위치

int\* indexes;

symbol을 정렬하는 대신, symbol을 기준으로 자연수 배열을 정렬한다.

이때 쓰일 정수배열

|  |
| --- |
| typedef |
|  | struct node\_symbol{ |
|  | int loc; |
|  | char symbol[100]; |
|  | struct node\_symbol\* next; |
|  | }node\_symbol; |
|  | // to store symbols |

Symbol을 저장하는 데 사용한 구조체

# 전역 변수 정의

## #define MAX\_OBJ\_PER\_LINE 60

한 줄에 올 최고 objcode의 개수를 지정한다. 처음의 주소는 제외하기 때문에 60이다,

## #define MAX\_LINE\_LENGTH 200

Type 시 읽어올 수 있는 한 줄의 최대 길이이다

## #define NO\_REGISTRE -1

해당하는 레지스터가 없음을 표현하는 상수

## #define MAX\_LINE\_ASM 200

asm파일의 최대 줄 수이다

## #define NOT\_FOUND -1

Symbol을 찾지 못한 경우 리턴값

## #define LENGTH\_SYMBOLTABLE 20

Symboltable 의 크기이다

.

# 코드 (프로젝트 1에서 변경된 파일만 작성)

# 5.1 20171667.c

#include "20171667.h"  
#include "opcode.h"  
#include "history.h"  
#include "memory.h"  
#include "assembler.h"  
#include "symbol.h"  
  
// for debug  
// if we define DEBUG, we can see debug message  
  
**int** min(**int** lhs, **int** rhs) {  
 // return minimun interger between lhs, rhs  
 **if** (lhs <= rhs)  
 **return** lhs;  
 **else  
 return** rhs;  
}  
  
**int** isHexString(**char** \*str) {  
 // check if str is right argument  
  
 **char** word[MAX\_COMMAND\_LENGTH];  
 memset(word, '\0', MAX\_COMMAND\_LENGTH);  
  
 **int** hex;  
  
 **int** num\_arg =  
 sscanf(str, "%x %[^ \n]", &hex, word);  
  
 // return false if str contain more than hexadecimal numder  
 **if** (num\_arg != 1)  
 **return** false;  
 **else  
 return** true;  
}  
  
**int** isOneWord(**char** \*str) {  
 // check if str is one word or not  
  
 **char** word[2][MAX\_COMMAND\_LENGTH];  
 memset(word, '\0', MAX\_COMMAND\_LENGTH);  
  
  
 **int** num\_arg =  
 sscanf(str, "%[^ \n\t] %[^ \n\t]", word[0], word[1]);  
  
 **return** num\_arg == 1;  
}  
  
**int** hash(**char**\* str){  
 // hash string to int [0, 19]  
 // add all characters and modular dy 20  
 **int** sum = 0;  
 **for** (**char**\* s = str; \*s != '\0'; s++)  
 {  
 // we ignore space when comparing  
 **if**(\*s != ' ')  
 sum += \*s;  
 }  
 **return** sum%20;  
}  
  
**void** help() {  
 // do help command, print all possible commands  
  
 **char** commandList[NUM\_COMMAND][50] = {  
 "h[elp]",  
 "d[ir]",  
 "q[uit]",  
 "hi[story]",  
 "du[mp][start, end]",  
 "e[dit] address, value",  
 "f[ill] start, end, value",  
 "reset",  
 "opcode mnemonic",  
 "opcodelist",  
 "assemble filename",  
 "type filename",  
 "symbol"  
 };  
  
 **for** (**int** i = 0; i < NUM\_COMMAND; i++) {  
 printf("%s\n", commandList[i]);  
 }  
}  
  
**void** quit() {  
 // do quit, after free all malloced memories  
 freeHistory();  
 freeHashTable();  
 freeSymbols();  
 isRunning = false;  
}  
  
  
**void** dir() {  
 // print information of current directory  
 DEBUG\_PRINT(("jumped into dir()\n"));  
  
 **struct** stat stat;  
 // strcuture represeting status of the information of directory  
  
 DIR \*dir;  
 **struct** dirent \*ent;  
 dir = opendir ("./");  
 // dir is currrent directory  
  
 **if** (dir == NULL) {  
 printf("Cannot open current directory!\n");  
 }  
 **else** {  
 **int** cnt = 0;  
 // count number of file or directory we printed  
  
 **while** ((ent = readdir (dir)) != NULL) {  
 lstat(ent->d\_name, &stat);  
  
 **if**(S\_ISDIR (stat.st\_mode)){  
 // this is directory  
 printf("%19s/", ent->d\_name);  
 }  
 **else if** (S\_IXUSR & stat.st\_mode) {  
 // this means executable  
 printf("%19s\*", ent->d\_name);  
 }  
 **else**{  
 printf("%20s", ent->d\_name);  
 }  
  
 cnt++;  
 **if**(cnt%4 == 0)  
 printf("\n");  
 // print newline for evey 4 files  
 }  
  
 **if**(cnt%4 != 0)  
 printf("\n");  
 // if not print newline just before, print newline  
  
  
 closedir (dir);  
 }  
}  
  
**void** parseCommand() {  
 // get input from stdio  
 // and seperate input into command, argments  
 // than decide what function should be called or raise error  
  
 printf("sicsim> ");  
  
 **char** commandLine[MAX\_COMMAND\_LENGTH];  
 fgets(commandLine, MAX\_COMMAND\_LENGTH, stdin);  
 // get whole line  
  
 **char** command[MAX\_COMMAND\_LENGTH];  
 memset(command, 0, MAX\_COMMAND\_LENGTH);  
 **char** word[3][MAX\_COMMAND\_LENGTH];  
 memset(word, 0, 3 \* MAX\_COMMAND\_LENGTH);  
 **char** sep[3][MAX\_COMMAND\_LENGTH];  
 memset(sep, 0, 2 \* MAX\_COMMAND\_LENGTH);  
  
 **int** frontSpace = sscanf(commandLine, "%[ \t]", sep[2]);  
 **int** num\_arg;  
  
 **if** (frontSpace == 0) {  
 num\_arg = sscanf(commandLine,  
 "%[^ \t\n] %[^,\n] %[,] %[^\n,] %[,] %[^\n]", command, word[0], sep[0], word[1], sep[1],  
 word[2]);  
 // seperate commandLine into command, word[0], sep[0], word[1], sep[1], word[2]  
 // command get the first word  
 // sep[] should be ','  
 // word[] get string between command and sep[]  
 // num\_arg get the number sscanf get from commandLine  
 } **else** {  
 num\_arg = sscanf(commandLine,  
 "%[ \t] %[^ \t\n] %[^,\n] %[,] %[^\n,] %[,] %[^\n]",  
 sep[2], command, word[0], sep[0], word[1], sep[1], word[2]);  
 num\_arg--;  
 }  
  
 DEBUG\_PRINT(("commandLine : %s\n", commandLine));  
 DEBUG\_PRINT(("num\_arg : %d\n", num\_arg));  
  
 DEBUG\_PRINT(("command : %s\n", command));  
 DEBUG\_PRINT(("word0 : %s\n", word[0]));  
 DEBUG\_PRINT(("sep0 : %s\n", sep[0]));  
 DEBUG\_PRINT(("word1 : %s\n", word[1]));  
 DEBUG\_PRINT(("sep1 : %s\n", sep[1]));  
 DEBUG\_PRINT(("word2 : %s\n", word[2]));  
 DEBUG\_PRINT(("sep2 : %s\n", sep[2]));  
 // check if we got right arguments  
  
 **switch** (num\_arg) {  
 **case** 1:  
 **if** (!strcmp(command, "h") || !strcmp(command, "help")) {  
 addHistory(commandLine);  
 help();  
 } **else if** (!strcmp(command, "d") || !strcmp(command, "dir")) {  
 addHistory(commandLine);  
 dir();  
  
 } **else if** (!strcmp(command, "q") || !strcmp(command, "quit")) {  
 addHistory(commandLine);  
 quit();  
  
 } **else if** (!strcmp(command, "hi") || !strcmp(command, "history")) {  
 addHistory(commandLine);  
 history();  
 } **else if** (!strcmp(command, "opcodelist")) {  
 opcodelist();  
 addHistory(commandLine);  
 } **else if** (!strcmp(command, "reset")) {  
 reset();  
 addHistory(commandLine);  
 } **else if** (!strcmp(command, "du") || !strcmp(command, "dump")) {  
 DEBUG\_PRINT(("call dump\n"));  
  
 // call dump [last\_dump+1, last\_dump+160]  
 // if last\_dump+160 > 0xFFFFF, call dump [last\_dump+1, 0xFFFFF]  
 // if last\_dump+1 > 0xFFFFF, error will be raised from dump()  
 **if** (dump(last\_dump + 1, min(0xFFFFF, last\_dump + 1 + 159)) == *SUCCESSFUL\_RETURN*)  
 addHistory(commandLine);  
 }  
 **else if** (!strcmp(command, "symbol")){  
 printSymbol();  
 addHistory(commandLine);  
 }  
 **else** {  
 printf("Syntax error!\n");  
 }  
 **break**;  
 **case** 2:  
 **if** (  
 (!strcmp(command, "du") || !strcmp(command, "dump"))  
 && isHexString(word[0])  
 ) {  
 // dump start  
 **int** start = (**int**) strtol(word[0], NULL, 16);  
 DEBUG\_PRINT(("call dump %02X\n", start));  
  
 **if** (dump(start, min(0xFFFFF, start + 159)) == *SUCCESSFUL\_RETURN*)  
 addHistory(commandLine);  
 } **else if** (!strcmp(command, "opcode") && isOneWord(word[0])) {  
 //opcode mnemonic  
 opcode(word[0]);  
 addHistory(commandLine);  
 } **else if** (!strcmp(command, "type")) {  
 **if** (type(word[0]) == *SUCCESSFUL\_RETURN*)  
 addHistory(commandLine);  
 } **else if** (!strcmp(command, "assemble")) {  
 **if** (assemble(word[0]) == *SUCCESSFUL\_RETURN*) {  
 addHistory(commandLine);  
 }  
 } **else** {  
 printf("Syntax error!\n");  
 }  
 **break**;  
 **case** 4:  
 **if** ((!strcmp(command, "du") || !strcmp(command, "dump"))  
 && isHexString(word[0])  
 && isHexString(word[1])) {  
  
 // for dump start, end  
  
 **int** start = (**int**)strtol(word[0], NULL, 16);  
 **int** end = (**int**)strtol(word[1], NULL, 16);  
  
 DEBUG\_PRINT(("call dump %02X %02X\n", start, end));  
 **if** (dump(start, end) == *SUCCESSFUL\_RETURN*)  
 addHistory(commandLine);  
 } **else if** ((!strcmp(command, "e") || !strcmp(command, "edit"))  
 && isHexString(word[0])  
 && !strcmp(sep[0], ",")  
 && isHexString(word[1])  
 ) {  
 // edit address, value  
 **int** address = (**int**) strtol(word[0], NULL, 16);  
 **int** value = (**int**) strtol(word[1], NULL, 16);  
  
 DEBUG\_PRINT(("address : %02X\n", address));  
 DEBUG\_PRINT(("value : %02X\n", value));  
  
 DEBUG\_PRINT(("call edit\n"));  
 **if** (edit(address, value) == *SUCCESSFUL\_RETURN*)  
 addHistory(commandLine);  
 } **else** {  
 printf("Syntax error!\n");  
 }  
 **break**;  
 **case** 6:  
 **if** (  
 (!strcmp(command, "f") || !strcmp(command, "fill"))  
 && isHexString(word[0])  
 && !strcmp(sep[0], ",")  
 && isHexString(word[1])  
 && !strcmp(sep[1], ",")  
 && isHexString(word[2])  
 ) {  
 // fill start, end, value  
 **int** start = (**int**) strtol(word[0], NULL, 16);  
 **int** end = (**int**) strtol(word[1], NULL, 16);  
 **int** value = (**int**) strtol(word[2], NULL, 16);  
  
 DEBUG\_PRINT(("call fill\n"));  
 **if** (fill(start, end, value) == *SUCCESSFUL\_RETURN*);  
 addHistory(commandLine);  
 }  
 **else** {  
 printf("Syntax error!\n");  
 }  
 **break**;  
 **default**:  
 printf("Syntax error!\n");  
 **break**;  
 }  
}  
  
**int** main() {  
 isRunning = true;  
 last\_dump = -1;  
 history\_head = NULL;  
  
 initHashTable();  
  
 **while** (isRunning) {  
 parseCommand();  
 }  
  
 **return** 0;  
}

# 5.2 opcode.c

#include "symbol.h"  
  
node\_symbol\*\* symbolTable = NULL;  
**int** num\_symbol = 0;  
**struct** node\_symbol\* symbol\_list = NULL;  
  
**void** initSymbolTable(){  
 **if**(symbolTable != NULL)  
 freeSymbols();  
 // if symbolTable used before, free them first  
  
  
 symbolTable = (node\_symbol\*\*)calloc(LENGTH\_SYMBOLTABLE, **sizeof**(node\_symbol\*));  
 symbol\_list = (node\_symbol\*)malloc(**sizeof**(node\_symbol)\*0);  
}  
  
**void** pushSymbol(**char**\* symbol, **int** loc){  
 // push symbols into symbolTable and symbolList  
  
 **int** target\_idx = hash(symbol);  
  
 node\_symbol\* new\_symbol = (node\_symbol\*)malloc(**sizeof**(node\_symbol));  
 new\_symbol->next = NULL;  
 strcpy(new\_symbol->symbol, symbol);  
 new\_symbol->loc = loc;  
 // make symbol structure  
  
 **if** (symbolTable[target\_idx] == NULL)  
 {  
 symbolTable[target\_idx] = new\_symbol;  
 }  
 **else** {  
 node\_symbol\* curr = symbolTable[target\_idx];  
 **while** (curr->next != NULL){  
 curr = curr->next;  
 }  
 curr->next = new\_symbol;  
 }  
 // push it into symbol table  
  
  
 num\_symbol++;  
 symbol\_list = (node\_symbol\*)realloc(symbol\_list, **sizeof**(node\_symbol)\*num\_symbol);  
 symbol\_list[num\_symbol- 1].loc = loc;  
 strcpy(symbol\_list[num\_symbol- 1].symbol, symbol);  
 // push it to symbol list  
}  
  
**int** getLoc(**char**\* symbol){  
 // return location of symbol  
 // if cannot find, return NOT\_FOUND(-1)  
  
 **int** target\_idx = hash(symbol);  
  
 node\_symbol\* curr = symbolTable[target\_idx];  
 **while** (curr != NULL){  
 **if** (!strcmp(curr->symbol, symbol)){  
 **return** curr->loc;  
 }  
 curr = curr->next;  
 }  
 **return** NOT\_FOUND;  
}  
  
**int** compareSymbol(**const void**\* lhs, **const void**\* rhs){  
 // sort indexes by lexicographical order of symbol\_list[\*lhs].symbol  
 **return** -1\*strcmp(symbol\_list[\*(**int**\*)lhs].symbol, symbol\_list[\*(**int**\*)rhs].symbol);  
}  
  
**int**\* sortedIndex(){  
 // sort array of indexes by lexicographical order of symbol\_list  
 // to avoid swap, copy of symbols  
  
 **int**\* indexes = (**int** \*)malloc(**sizeof**(**int**)\* num\_symbol);  
 **for** (**int** i = 0; i < num\_symbol; ++i) {  
 indexes[i] = i;  
 }  
  
 qsort(indexes, num\_symbol, **sizeof**(**int**), compareSymbol);  
  
 **return** indexes;  
}  
  
**void** printSymbol(){  
 // print symbols  
  
 **if**(symbolTable == NULL)  
 fprintf(stderr, "there is no symbol table yet\n");  
  
 **int**\* indexes = sortedIndex();  
  
 **for** (**int** i = 0; i < num\_symbol; ++i) {  
 printf("\t%-6s\t%04X\n", symbol\_list[indexes[i]].symbol, symbol\_list[indexes[i]].loc);  
 }  
}  
  
**void** freeSymbols(){  
 //free symbols to avoid memory leak  
  
 **for** (**int** i = 0; i < LENGTH\_SYMBOLTABLE; i++)  
 {  
 DEBUG\_PRINT(("i is %d\n", i));  
  
 **if** (symbolTable[i] != NULL)  
 {  
 node\_symbol\* curr = symbolTable[i]->next;  
 node\_symbol\* prev = symbolTable[i];  
  
 **while** (curr != NULL) {  
 DEBUG\_PRINT(("%s get free!\n", prev->symbol));  
 free(prev);  
  
 prev = curr;  
 curr = curr->next;  
 }  
 DEBUG\_PRINT(("%s get free!\n", prev->symbol));  
 free(prev);  
 }  
 }  
 free(symbolTable);  
 free(symbol\_list);  
  
 symbolTable = NULL;  
 num\_symbol = 0;  
 symbol\_list = NULL;  
}

# 5.2 assembler.c

#include "20171667.h"  
#include "assembler.h"  
#include "symbol.h"  
#include "opcode.h"  
  
**unsigned int** regToAddress(**char**\* reg){  
 // return address of register  
 // if ew cant find, return NO\_REGISTER  
  
 **if**(!strcmp(reg, "")){  
 **return** 0;  
 }  
 **if**(!strcmp(reg, "A")){  
 **return** 0;  
 }  
 **else if**(!strcmp(reg, "X")){  
 **return** 1;  
 }  
 **else if**(!strcmp(reg, "L")){  
 **return** 2;  
 }  
 **else if**(!strcmp(reg, "B")){  
 **return** 3;  
 }  
 **else if**(!strcmp(reg, "S")){  
 **return** 4;  
 }  
 **else if**(!strcmp(reg, "T")){  
 **return** 5;  
 }  
 **else if**(!strcmp(reg, "F")){  
 **return** 6;  
 }  
 **else if**(!strcmp(reg, "PC")){  
 **return** 8;  
 }  
 **else if**(!strcmp(reg, "SW")){  
 **return** 9;  
 }  
 **else** {  
 **return** NO\_REGISTRE;  
 }  
}  
  
**void** raiseError(**int** line, **enum** assemble\_error error) {  
 // handle error  
 // print error message  
  
 line \*= 5;  
  
  
 **switch** (error) {  
 **case** *twice\_declared*:  
 fprintf(stdout, "label was declared twice at line %d\n", line);  
 **break**;  
 **case** *undefined\_mnemonic*:  
 fprintf(stdout, "undefined mnemonic at line %d\n", line);  
 **break**;  
 **case** *undefined\_symbol*:  
 fprintf(stdout, "undefined symbol at line %d\n", line);  
 **break**;  
 **case** *undefined\_label*:  
 fprintf(stdout, "undefined label at line %d\n", line);  
 **break**;  
 **case** *memory\_boundary*:  
 fprintf(stdout, "out of memory boundary at line %d\n", line);  
 **break**;  
 **case** *not\_asm\_file*:  
 fprintf(stdout, "this is not .asm file\n");  
 **break**;  
 **case** *START\_not\_exist*:  
 fprintf(stdout, "START not exist at line %d\n", line);  
 **break**;  
 }  
}  
  
**char**\* checkX(**char**\* str){  
 // check it str have ",X"  
  
 **char** lable[10], comma[10], x[10];  
 **int** num\_arg = sscanf(str, "%[^ ,\n] %[,] %[X]", lable, comma, x);  
 **if** (num\_arg == 3){  
 **return** &str[strlen(lable)];  
 }  
 **else** {  
 **return** NULL;  
 }  
}  
  
**char**\* printObj(**struct** list\_line\* line){  
 // print object code of one line structure  
  
 **char**\* obj = malloc(**sizeof**(**char**)\* (MAX\_OBJ\_PER\_LINE+1));  
  
 **if** (line->isCommnet){  
 obj[0] = '\0';  
 }  
 **else if**(line->obj\_size == 0)  
 obj[0] = '\0';  
 **else if**(line->data[0] != '\0') {  
 strcpy(obj, line->data);  
 }  
 **else**{  
 **char** temp\_obj[10];  
 sprintf(temp\_obj, "%X", line->obj);  
 **if** (strlen(temp\_obj)%2)  
 sprintf(obj, "0");  
 sprintf(obj + strlen(temp\_obj)%2, "%X", line->obj);  
 }  
 **return** obj;  
}  
  
**void** printLine(FILE\* stream, **struct** list\_line\* line, **int** idx\_line){  
 // print one line of list code  
 // into FILE\* stream  
  
 fprintf(stream, "%-6d ", (idx\_line+1)\*5);  
  
 **if** (line->isCommnet){  
 fprintf(stream, " %s", line->str+1);  
 }  
 **else** {  
  
 fprintf(stream, "%04X %-6s %-6s %-30s ", line->loc, line->label, line->mnemonic, line->operand);  
  
 **if**(line->obj\_size == 0)  
 fprintf(stream, "\n");  
 **else if**(line->data[0] != '\0')  
 fprintf(stream, "%s\n", line->data);  
 **else**{  
 **char** obj[10];  
 sprintf(obj, "%X", line->obj);  
 **if** (strlen(obj)%2)  
 fprintf(stream, "0");  
 fprintf(stream, "%X\n", line->obj);  
 }  
 }  
}  
  
**int** type(**char** \*filename) {  
 // do type command  
 // print filename file until EOF come  
  
 FILE\* fp = fopen(filename, "r");  
  
 **if**(fp == NULL){  
 fprintf(stderr, "cannot open the file");  
 **return** UNSUCCESSFUL\_RETURN;  
 }  
  
 **char** line[MAX\_LINE\_LENGTH];  
 **while** (fgets(line, MAX\_LINE\_LENGTH, fp) != NULL){  
 printf("%s", line);  
 }  
  
 printf("\n");  
 fclose(fp);  
 **return** SUCCESSFUL\_RETURN;  
}  
  
**int** assemble\_pass1(FILE\* fp, **struct** list\_line list[MAX\_LINE\_ASM], **char** codename[10], **int**\* num\_line){  
 // do pass1  
 // push labels to hashTable of symbols  
 // count location and save location and information of each line into list[1000] array  
  
 **unsigned int** loc = 0;  
 **int** line = 0;  
 **char** buf[100];  
  
 **while** (fgets(buf, 100, fp) != NULL){  
 // read one line  
  
 **struct** list\_line\* curr = &list[line++];  
 // structure for current line  
  
 curr->isCommnet = false;  
 curr->loc = loc;  
 curr->data[0] = '\0';  
  
 **if**(buf[0] == '.'){  
 curr->isCommnet = true;  
 curr->str = (**char** \*)malloc(**sizeof**(**char**) \* (strlen(buf) + 1));  
 strcpy(curr->str, buf);  
 **continue**;  
 // if it is comment, save contents of comment  
 }  
 **else if**(buf[0] == ' ' || buf[0] == '\t'){  
 sscanf(buf, "%s %[^\n]", curr->mnemonic, curr->operand);  
 curr->label[0] = '\0';  
 // if it start with space or tab, scan mnemonic and operand  
 }  
 **else**{  
 sscanf(buf, "%s %s %[^\n]", curr->label, curr->mnemonic, curr->operand);  
  
 **if**(line == 1 && !strcmp(curr->mnemonic, "START")){  
 // if current mnemonic is START, do not push into symbol table  
 }  
 **else if** (getLoc(curr->label) == NOT\_FOUND)  
 pushSymbol(curr->label, loc);  
 **else** {  
 raiseError(line, twice\_declared);  
 **return** UNSUCCESSFUL\_RETURN;  
 }  
 }  
  
  
  
 **if**(line == 1){  
 // if there exist START and operand in line 1, set loc = mnemonic, else loc = 0  
 // if there is no START at first line, raise error  
  
 **if**(!strcmp(curr->mnemonic, "START")){  
 **if**(strcmp(curr->operand, "")){  
 loc = (**unsigned int**)strtol(curr->operand, NULL, 16);  
 }  
 **else**{  
 loc = 0;  
 }  
 list[0].loc = loc;  
 strcpy(codename, curr->label);  
 curr->obj\_size = 0;  
 }  
 **else**{  
 raiseError(line, START\_not\_exist);  
 **return** UNSUCCESSFUL\_RETURN;  
 }  
 }  
 **else if**(!strcmp(curr->mnemonic, "END")){  
  
 curr->isCommnet = true;  
  
 curr->str = (**char** \*)malloc(**sizeof**(**char**) \* (strlen(buf) + 1));  
 strcpy(curr->str, buf);  
 **break**;  
 }  
 **else if**(!strcmp(curr->mnemonic, "BYTE")){  
 // if BYTE copy data into curr-> data  
 // and set curr->obj\_size to right size  
  
  
 **if**(curr->operand[0] == 'X'){  
  
 **int** length = (**int**)strlen(curr->operand)-3;  
 **if**(length%2 == 1){  
 sprintf(curr->data, "0");  
 }  
 // if X'...' have odd number of characters  
 // add '0'  
 // example : X'E' -> X'0E'  
  
 **for** (**int** i = 0; curr->operand[i+2] != '\''; i++) {  
 sprintf(curr->data + i + length%2, "%c", curr->operand[i+2]);  
 }  
  
 curr->obj\_size = length + length%2;  
 loc += length/2;  
 }  
 **else if**(curr->operand[0] == 'C') {  
  
 **for** (**int** i = 2; curr->operand[i] != '\''; i++) {  
 **int** c = curr->operand[i];  
 sprintf(curr->data + (i - 2) \* 2, "%02X", c);  
 }  
  
 curr->obj\_size = (**int**)(2\*(strlen(curr->operand)-3));  
 loc += strlen(curr->operand)-3;  
 }  
  
 }  
 **else if**(!strcmp(curr->mnemonic, "RESB")){  
 **int** length;  
 sscanf(curr->operand, "%d", &length);  
 loc+=length;  
  
 curr->obj\_size = 0;  
 }  
 **else if**(!strcmp(curr->mnemonic, "WORD")){  
 loc+=3;  
  
 **int** data;  
 sscanf(curr->operand, "%d", &data);  
 sprintf(curr->data, "%06X", data);  
  
 curr->obj\_size = 3;  
 }  
 **else if**(!strcmp(curr->mnemonic, "RESW")){  
 **int** length;  
 sscanf(curr->operand, "%d", &length);  
 loc+=length\*3;  
  
 curr->obj\_size=0;  
 }  
 **else if**(!strcmp(curr->mnemonic, "BASE")){  
 curr->isCommnet = true;  
  
 curr->str = (**char** \*)malloc(**sizeof**(**char**) \* (strlen(buf) + 1));  
 strcpy(curr->str, buf);  
 }  
 **else**{  
  
 **if**(curr->mnemonic[0] == '+'){  
 loc += 4;  
 curr->obj\_size = 4;  
 }  
 **else if** (!strcmp(getInst(curr->mnemonic)->format, "1")){  
 loc += 1;  
 curr->obj\_size = 1;  
 }  
 **else if** (!strcmp(getInst(curr->mnemonic)->format, "2")){  
 loc += 2;  
 curr->obj\_size = 2;  
 }  
 **else if** (!strcmp(getInst(curr->mnemonic)->format, "3/4")){  
 loc += 3;  
 curr->obj\_size = 3;  
 }  
 }  
 }  
 \*num\_line = line;  
 **return** SUCCESSFUL\_RETURN;  
}  
  
**int** assemble\_pass2(**struct** list\_line list[MAX\_LINE\_ASM], FILE\* fp, **int** num\_line) {  
 // do pass2  
 // generate obj code of non-data, non-directive instruction  
  
 **int** B = -1;  
 // value of Base Register  
 // initialize to -1 so that we can know it is not initialized  
  
 **for** (**int** line = 0; line < num\_line; line++) {  
  
 **struct** list\_line\* curr = &list[line];  
 // structure represent curr line  
  
 **if**(!strcmp(curr->mnemonic, "BASE")){  
 B = getLoc(curr->operand);  
 // set Base;  
 }  
  
 **else if** (!curr->isCommnet && curr->data[0] == '\0') {  
 // for non-data, non-directive instruction  
  
 **int** format = curr->obj\_size;  
  
 **if** (format == 1) {  
 **struct** format1 obj = {0,};  
 obj.opcode = (**unsigned int**)(getInst(curr->mnemonic)->opcode);  
  
 curr->obj = obj.all;  
  
 } **else if** (format == 2) {  
 **struct** format2 obj = {0,};  
 obj.opcode = (**unsigned int**)(getInst(curr->mnemonic)->opcode);  
 format = 2;  
  
  
 **char** r1[10] = {0, }, r2[10] = {0, };  
 sscanf(curr->operand, "%[^,] , %s", r1, r2);  
  
 obj.r1 = regToAddress(r1);  
 obj.r2 = regToAddress(r2);  
  
 curr->obj = obj.all;  
  
 } **else if** (format == 3) {  
 format = 3;  
  
 **struct** format3 obj = {0,};  
 obj.opcode = (**unsigned int**)(getInst(curr->mnemonic)->opcode)/4;  
  
 **if** (!strcmp(curr->mnemonic, "RSUB")){  
 obj.i = 1;  
 obj.n = 1;  
 curr->obj = obj.all;  
 }**else** {  
  
 **char**\* keep\_end\_address = checkX(curr->operand);  
 **char** keep\_end\_char;  
  
 **if**(keep\_end\_address != NULL) {  
 keep\_end\_char = \*keep\_end\_address;  
 obj.x = 1;  
 \*keep\_end\_address='\0';  
 }  
  
 **int** target\_address;  
 **if** (curr->operand[0] == '@' || curr->operand[0] == '#') {  
 target\_address = getLoc(curr->operand + 1);  
 } **else** {  
 target\_address = getLoc(curr->operand);  
 }  
  
 **if** (curr->operand[0] == '#') {  
 obj.i = 1;  
 } **else if** (curr->operand[0] == '@') {  
 obj.n = 1;  
 } **else** {  
 obj.i = 1;  
 obj.n = 1;  
 }  
  
 **int** pc = curr->loc + 3;  
 **if** (curr->operand[0] == '#' && getLoc(curr->operand + 1) == NOT\_FOUND)  
 {  
 // for constant value  
 obj.disp = (**unsigned int**)strtol(curr->operand + 1, NULL, 10);  
  
 } **else if** (-2048 <= target\_address - pc && target\_address - pc <= 2047) {  
 // for PC relative  
 obj.b = 0;  
 obj.p = 1;  
 obj.disp = (target\_address - pc);  
  
 } **else if** (B != -1 && 0 <= target\_address - B && target\_address - B <= 4095) {  
 // for Base relative  
 obj.b = 1;  
 obj.p = 0;  
 obj.disp = (target\_address - B);  
 } **else** {  
 // cannot represent address in format3  
 raiseError(line, memory\_boundary);  
 **return** UNSUCCESSFUL\_RETURN;  
 }  
 curr->obj = obj.all;  
  
 **if**(keep\_end\_address != NULL) {  
 \*keep\_end\_address = keep\_end\_char;  
 }  
 }  
 }  
 **if** (format == 4) {  
  
 **struct** format4 obj = {0,};  
 obj.e = 1;  
 obj.opcode = getInst(curr->mnemonic + 1)->opcode/4;  
  
 **char**\* keep\_end\_address = checkX(curr->operand);  
 **char** keep\_end\_char;  
  
 **if**(keep\_end\_address != NULL) {  
 keep\_end\_char = \*keep\_end\_address;  
 obj.x = 1;  
 \*keep\_end\_address='\0';  
 }  
  
 **if** (curr->operand[0] == '#') {  
 **if**(getLoc(curr->operand + 1) == NOT\_FOUND)  
 obj.address = strtol(curr->operand + 1, NULL, 10);  
 **else** obj.address = getLoc(curr->operand + 1);  
 obj.i = 1;  
 } **else if** (curr->operand[0] == '@') {  
 obj.n = 1;  
 obj.address = getLoc(curr->operand + 1);  
 } **else** {  
 obj.i = 1;  
 obj.n = 1;  
 obj.address = getLoc(curr->operand);  
 }  
  
 **if**(obj.x == 1){  
 \*strchr(curr->operand, '\0') = ',';  
 }  
 curr->obj = obj.all;  
  
 **if**(keep\_end\_address != NULL) {  
 \*keep\_end\_address = keep\_end\_char;  
 }  
 }  
 }  
 printLine(fp, curr, line);  
 }  
 **return** SUCCESSFUL\_RETURN;  
}  
  
**void** writeObj(**struct** list\_line list[MAX\_LINE\_ASM], FILE\* fp, **int** num\_line, **char**\* codename) {  
 // write object code  
  
 fprintf(fp, "H%-6s%06X%06X\n", codename, list[0].loc, list[num\_line - 1].loc - list[0].loc);  
  
 **int** line = 1;  
  
 **do** {  
  
 **char** buf[70] = {0,};  
 **int** place\_print = 0;  
  
 **do** {  
 **char** \*obj = printObj(&list[line]);  
 **int** obj\_length = strlen(obj);  
 **if** (place\_print + obj\_length > 60)  
 **break**;  
  
 **if** (list[line].obj\_size == 0 && list[line].isCommnet == 0) {  
 // for variables or constants  
 line++;  
 **break**;  
 }  
  
 sprintf(buf + place\_print, "%s", obj);  
 place\_print += obj\_length;  
 line++;  
 } **while** (line < num\_line);  
  
 **if** (buf[0] != '\0')  
 fprintf(fp, "T%06X%02X%s\n", list[line].loc, (**int**)strlen(buf), buf);  
 } **while** (line < num\_line);  
  
 **for** (**int** i = 0; i < num\_line; ++i) {  
  
 **if** (!list[i].isCommnet && list[i].data[0] == '\0' && list[i].obj\_size == 4 &&  
 !(list[i].operand[0] == '#' && getLoc(list[i].operand + 1) == NOT\_FOUND)) {  
 // for format 4  
 // except #constants  
 fprintf(fp, "M%06X%02X\n", list[i].loc - list[0].loc + 1, 05);  
 }  
 }  
 fprintf(fp, "E%06X\n", list[0].loc);  
}  
  
**int** assemble(**char**\* filename\_asm){  
 **int** filename\_length = strlen(filename\_asm);  
 **if**(filename\_length < 4  
 || strcmp(&filename\_asm[filename\_length-4], ".asm")  
 ){  
  
 raiseError(-1, not\_asm\_file);  
 **return** UNSUCCESSFUL\_RETURN;  
 }  
  
  
 FILE\* fp\_asm = fopen(filename\_asm, "r");  
  
 **if**(fp\_asm == NULL){  
 fprintf(stderr, "cannot open the file");  
 **return** UNSUCCESSFUL\_RETURN;  
 }  
  
 initSymbolTable();  
 **char** codename[10];  
 **struct** list\_line list[MAX\_LINE\_ASM];  
  
 **int** num\_line = 0;  
 **int** return\_pass1 = assemble\_pass1(fp\_asm, list, codename, &num\_line);  
 fclose(fp\_asm);  
 **if** (return\_pass1 == UNSUCCESSFUL\_RETURN){  
 // if error raised during pass1, exit function  
 **return** UNSUCCESSFUL\_RETURN;  
 }  
  
  
 **char** filename\_lst[100];  
 strcpy(filename\_lst, filename\_asm);  
 strcpy(strchr(filename\_lst, '.'), ".lst");  
  
  
 **char** filename\_obj[100];  
 strcpy(filename\_obj, filename\_asm);  
 strcpy(strchr(filename\_obj, '.'), ".obj");  
  
  
 DEBUG\_PRINT(("loc label opcode format\n"));  
  
  
 FILE\* fp\_lst = fopen(filename\_lst, "w");  
 **int** return\_pass2 = assemble\_pass2(list, fp\_lst, num\_line);  
 fclose(fp\_lst);  
 **if** (return\_pass2 == UNSUCCESSFUL\_RETURN){  
 // if error raised during pass2, delete lst and exit function  
 remove(filename\_lst);  
 **return** UNSUCCESSFUL\_RETURN;  
 }  
  
 FILE\* fp\_obj = fopen(filename\_obj, "w");  
 writeObj(list, fp\_obj, num\_line, codename);  
 fclose(fp\_obj);  
 **return** SUCCESSFUL\_RETURN;  
}

**5.3 assembler.h**

#ifndef CSE4100\_ASSEMBLER\_H  
#define CSE4100\_ASSEMBLER\_H  
  
#define MAX\_OBJ\_PER\_LINE 60  
#define MAX\_LINE\_LENGTH 200  
#define NO\_REGISTRE -1  
#define MAX\_LINE\_ASM 10000  
  
**enum** assemble\_error{  
 *twice\_declared* = 0,  
 *undefined\_mnemonic*,  
 *undefined\_symbol*,  
 *undefined\_label*,  
 *memory\_boundary*,  
 *not\_asm\_file*,  
 *START\_not\_exist*};  
// enums for errors  
  
**struct** format1 {  
 **union** {  
 **struct** {  
 **unsigned int** opcode : 8;  
 };  
 **unsigned int** all;  
 };  
};  
// bit field for format 1  
  
**struct** format2 {  
 **union** {  
 **struct** {  
 **unsigned int** r2 : 4;  
 **unsigned int** r1 : 4;  
 **unsigned int** opcode : 8;  
 };  
 **unsigned int** all;  
 };  
};  
// bit field for format 2  
  
**struct** format3 {  
 **union** {  
 **struct** {  
 **unsigned int** disp : 12;  
 **unsigned int** e : 1;  
 **unsigned int** p : 1;  
 **unsigned int** b : 1;  
 **unsigned int** x : 1;  
 **unsigned int** i : 1;  
 **unsigned int** n : 1;  
 **unsigned int** opcode : 6;  
 };  
 **unsigned int** all;  
 };  
};  
// bit field for format 3  
  
**struct** format4 {  
 **union** { // 익명 공용체  
 **struct** { // 익명 구조체  
 **unsigned int** address : 20;  
 **unsigned int** e : 1;  
 **unsigned int** p : 1;  
 **unsigned int** b : 1;  
 **unsigned int** x : 1;  
 **unsigned int** i : 1;  
 **unsigned int** n : 1;  
 **unsigned int** opcode : 6;  
 };  
 **unsigned int** all;  
 };  
};  
// bit field for format 4  
  
**struct** list\_line{  
 **int** isCommnet;  
 **int** obj\_size;  
  
 **char**\* str;  
  
 **unsigned int** loc;  
 **char** label[10];  
 **char** mnemonic[10];  
 **char** operand[10];  
 **unsigned int** obj;  
 **char** data[30+1];  
};  
// to store information of each line of list file  
  
**int** type(**char**\* filename);  
**int** assemble(**char**\* filename\_asm);  
  
#endif //CSE4100\_ASSEMBLER\_H

# 5.4 symbol.c

#include "symbol.h"  
  
node\_symbol\*\* symbolTable = NULL;  
**int** num\_symbol = 0;  
**struct** node\_symbol\* symbol\_list = NULL;  
  
**void** initSymbolTable(){  
 **if**(symbolTable != NULL)  
 freeSymbols();  
 // if symbolTable used before, free them first  
  
  
 symbolTable = (node\_symbol\*\*)calloc(LENGTH\_SYMBOLTABLE, **sizeof**(node\_symbol\*));  
 symbol\_list = (node\_symbol\*)malloc(**sizeof**(node\_symbol)\*0);  
}  
  
**void** pushSymbol(**char**\* symbol, **int** loc){  
 // push symbols into symbolTable and symbolList  
  
 **int** target\_idx = hash(symbol);  
  
 node\_symbol\* new\_symbol = (node\_symbol\*)malloc(**sizeof**(node\_symbol));  
 new\_symbol->next = NULL;  
 strcpy(new\_symbol->symbol, symbol);  
 new\_symbol->loc = loc;  
 // make symbol structure  
  
 **if** (symbolTable[target\_idx] == NULL)  
 {  
 symbolTable[target\_idx] = new\_symbol;  
 }  
 **else** {  
 node\_symbol\* curr = symbolTable[target\_idx];  
 **while** (curr->next != NULL){  
 curr = curr->next;  
 }  
 curr->next = new\_symbol;  
 }  
 // push it into symbol table  
  
  
 num\_symbol++;  
 symbol\_list = (node\_symbol\*)realloc(symbol\_list, **sizeof**(node\_symbol)\*num\_symbol);  
 symbol\_list[num\_symbol- 1].loc = loc;  
 strcpy(symbol\_list[num\_symbol- 1].symbol, symbol);  
 // push it to symbol list  
}  
  
**int** getLoc(**char**\* symbol){  
 // return location of symbol  
 // if cannot find, return NOT\_FOUND(-1)  
  
 **int** target\_idx = hash(symbol);  
  
 node\_symbol\* curr = symbolTable[target\_idx];  
 **while** (curr != NULL){  
 **if** (!strcmp(curr->symbol, symbol)){  
 **return** curr->loc;  
 }  
 curr = curr->next;  
 }  
 **return** NOT\_FOUND;  
}  
  
**int** compareSymbol(**const void**\* lhs, **const void**\* rhs){  
 // sort indexes by lexicographical order of symbol\_list[\*lhs].symbol  
 **return** -1\*strcmp(symbol\_list[\*(**int**\*)lhs].symbol, symbol\_list[\*(**int**\*)rhs].symbol);  
}  
  
**int**\* sortedIndex(){  
 // sort array of indexes by lexicographical order of symbol\_list  
 // to avoid swap, copy of symbols  
  
 **int**\* indexes = (**int** \*)malloc(**sizeof**(**int**)\* num\_symbol);  
 **for** (**int** i = 0; i < num\_symbol; ++i) {  
 indexes[i] = i;  
 }  
  
 qsort(indexes, num\_symbol, **sizeof**(**int**), compareSymbol);  
  
 **return** indexes;  
}  
  
**void** printSymbol(){  
 // print symbols  
  
 **if**(symbolTable == NULL)  
 fprintf(stderr, "there is no symbol table yet\n");  
  
 **int**\* indexes = sortedIndex();  
  
 **for** (**int** i = 0; i < num\_symbol; ++i) {  
 printf("\t%-6s\t%04X\n", symbol\_list[indexes[i]].symbol, symbol\_list[indexes[i]].loc);  
 }  
}  
  
**void** freeSymbols(){  
 //free symbols to avoid memory leak  
  
 **for** (**int** i = 0; i < LENGTH\_SYMBOLTABLE; i++)  
 {  
 DEBUG\_PRINT(("i is %d\n", i));  
  
 **if** (symbolTable[i] != NULL)  
 {  
 node\_symbol\* curr = symbolTable[i]->next;  
 node\_symbol\* prev = symbolTable[i];  
  
 **while** (curr != NULL) {  
 DEBUG\_PRINT(("%s get free!\n", prev->symbol));  
 free(prev);  
  
 prev = curr;  
 curr = curr->next;  
 }  
 DEBUG\_PRINT(("%s get free!\n", prev->symbol));  
 free(prev);  
 }  
 }  
 free(symbolTable);  
 free(symbol\_list);  
  
 symbolTable = NULL;  
 num\_symbol = 0;  
 symbol\_list = NULL;  
}

# 5.5 symbol.h

#ifndef CSE4100\_SYMBOL\_H  
#define CSE4100\_SYMBOL\_H  
  
#include "20171667.h"  
  
#define NOT\_FOUND (-1)  
#define LENGTH\_SYMBOLTABLE 20  
  
**typedef  
struct** node\_symbol{  
 **int** loc;  
 **char** symbol[100];  
 **struct** node\_symbol\* next;  
}node\_symbol;  
// to store symbols  
  
**void** initSymbolTable();  
**void** pushSymbol(**char**\* symbol, **int** loc);  
**int** getLoc(**char**\* symbol);  
**void** printSymbol();  
**void** freeSymbols();  
  
  
#endif //CSE4100\_SYMBOL\_H