시스템프로그래밍 \_ 소정민 교수

ASSIGNMENT 3

**서강대학교 경영학/컴퓨터공학**

**20150959정서원**

목 차

1. 프로그램 개요 5

2. 프로그램 설명 5

2.1 프로그램 흐름도 5

3. 모듈 정의 6

3.1 모듈 이름 : int main(void) 6

3.1.1 기능 6

3.1.2 사용 변수 6

3.2 모듈 이름: void help\_func(void) 6

3.2.1 기능 6

3.2.2 사용 변수 6

3.3 모듈 이름: int dir\_func(void) 6

3.3.1 기능 6

3.3.2 사용 변수 6

3.4 모듈 이름: void check\_history\_num\_func(void) 6

3.4.1 기능 6

3.4.2 사용 변수 7

3.5 모듈 이름: void add\_history\_func(char\* fullcmd) 7

3.5.1 기능 7

3.5.2 사용 변수 7

3.6 모듈 이름: void show\_history\_func(void) 7

3.6.1 기능 7

3.6.2 사용 변수 7

3.7 모듈 이름: void dump\_func(int start, int end) 7

3.7.1 기능 7

3.7.2 사용 변수 7

3.8 모듈 이름: void edit\_func(int address, int value) 7

3.8.1 기능 7

3.8.2 사용 변수 7

3.9 모듈 이름: void fill\_func(int start, int end, int value) 8

3.9.1 기능 8

3.9.2 사용 변수 8

3.10 모듈 이름: void reset\_func(void) 8

3.10.1 기능 8

3.10.2 사용 변수 8

3.11 모듈 이름: int read\_opcode\_func(void) 8

3.11.1 기능 8

3.11.2 사용 변수 8

3.12 모듈 이름: void add\_hash\_func(int op, char \*mn, char \*fm) 8

3.12.1 기능 8

3.12.2 사용 변수 8

3.13 모듈 이름: void opcode\_list\_func(void) 9

3.13.1 기능 9

3.13.2 사용 변수 9

3.14 모듈 이름: int find\_opcode\_func(char \*mn, int option) 9

3.14.1 기능 9

3.14.2 사용 변수 9

3.15 모듈 이름: int type\_func(char \*filename) 9

3.15.1 기능 9

3.15.2 사용 변수 9

3.16 모듈 이름: int key\_for\_symbol\_func(char \*sym) 9

3.16.1 기능 9

3.16.2 사용 변수 9

3.17 모듈 이름: void add\_symbol\_func(char \*sym, int addr) 10

3.17.1 기능 10

3.17.2 사용 변수 10

3.18 모듈 이름: int find\_label\_func(char \*label, int option) 10

3.18.1 기능 10

3.18.2 사용 변수 10

3.19 모듈 이름: char\* format\_for\_symbol\_func(char \*mn) 10

3.19.1 기능 10

3.19.2 사용 변수 10

3.20 모듈 이름: void symbol\_print\_func(void) 10

3.20.1 기능 10

3.20.2 사용 변수 10

3.21 모듈 이름: int assemble\_func(char \*filename) 10

3.21.1 기능 10

3.21.2 사용 변수 11

3.22 모듈 이름: void progaddr\_func(int address); 11

3.22.1 기능 11

3.22.2 사용 변수 11

3.23 모듈 이름: int loader\_func(char \*fname1, char \*fname2, char \*fname3) 11

3.23.1 기능 11

3.23.2 사용 변수 11

3.24 모듈 이름: int extdef\_func(FILE \*fp, int startaddr, char \*progname, int drflag) 11

3.24.1 기능 11

3.24.2 사용 변수 11

3.25 모듈 이름: void extref\_func(FILE \*fp, char \*progname, int drflag) 11

3.25.1 기능 11

3.25.2 사용 변수 11

3.26 모듈 이름: void tmrecord\_func(FILE \*fp, char \*progname, int drflag) 12

3.26.1 기능 12

3.26.2 사용 변수 12

3.27 모듈 이름: void add\_bp\_func(int address) 12

3.27.1 기능 12

3.27.2 사용 변수 12

3.28 모듈 이름: void bp\_print\_func(void) 12

3.28.1 기능 12

3.28.2 사용 변수 12

3.29 모듈 이름: void bp\_clear\_func(void) 12

3.29.1 기능 12

3.29.2 사용 변수 12

3.30 모듈 이름: int find\_mnemonic\_func(int opcode, char \*mnemonic) 13

3.30.1 기능 13

3.30.2 사용 변수 13

3.31 모듈 이름: void run\_func(int totallength, int \*currbp, int \*nextbp) 13

3.31.1 기능 13

3.31.2 사용 변수 13

4. 전역 변수 정의 13

4.1 int historynum 13

4.2 unsigned char mymemory[MEGABYTE] 13

4.3 HISTORYLIST \*historyhead, \*historytemp, \*historynewnode 13

4.4 HASH \*hashtable[HASHTABLENUM], \*hashhead, \*hashtemp, \*hashnewnode 13

4.5 SYMBOL \*symboltable[SYMBOLTABLENUM], \*symhead, \*symtemp, \*symnewnode 14

4.6 int A, X, L, PC, B, S, T 14

4.7 short int breakpoint[MEGABYTE], int bpnum 14

4.8 int progaddr, curraddr, execaddr 14

5. 코드 14

5.1 20150959.h 14

5.2 20150959.c 16

5.3 first\_project.c 25

5.4 second\_project.c 31

5.5 third\_project.c 41

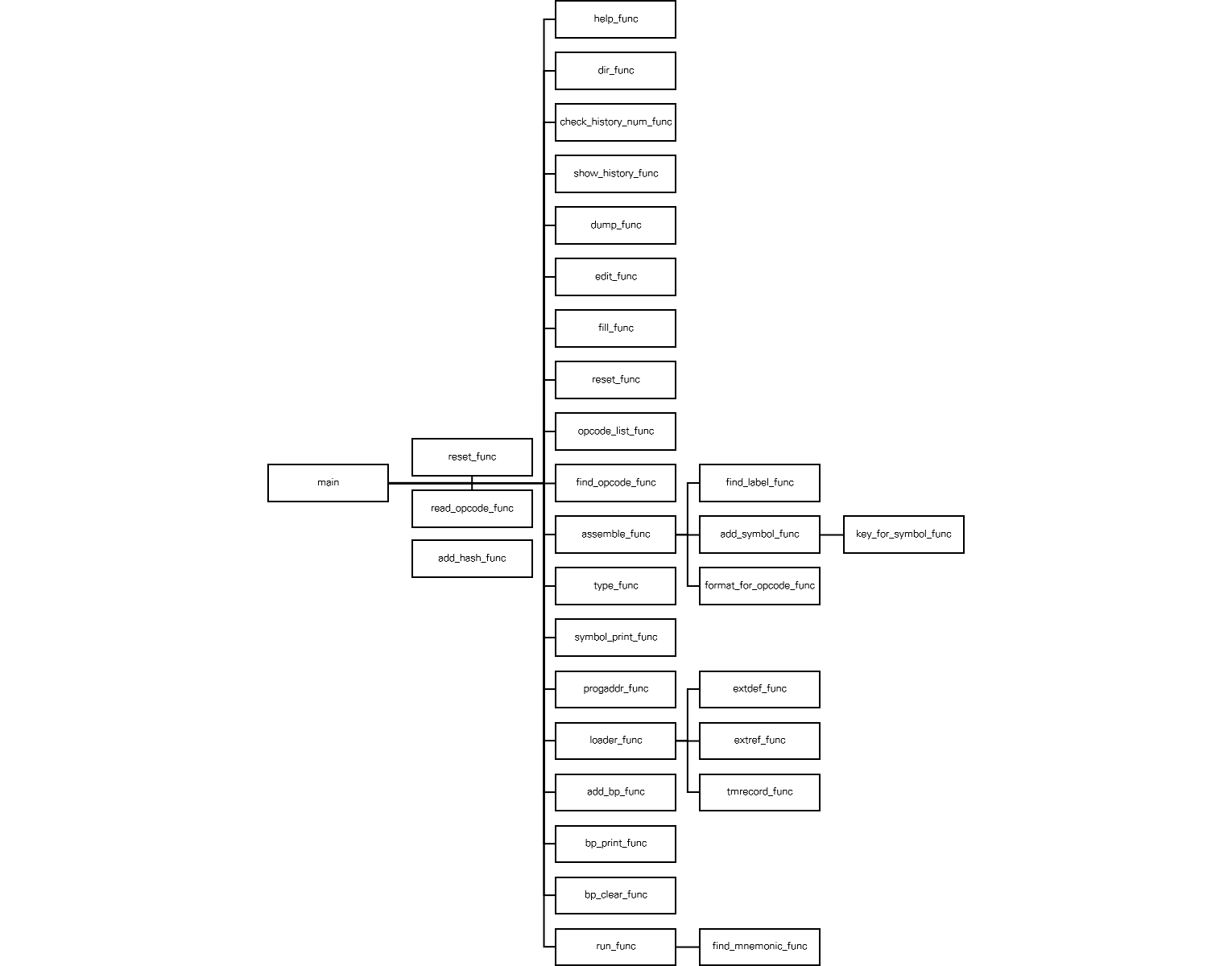
5.5 Makefile 52

# 프로그램 개요

3차 프로젝트에서는 .obj 파일을 메모리에 로드하고 실행하는 기능을 구현하였다. 이에 더해 프로그램의 시작점인 메모리 주소를 설정하는 기능과, 프로그램을 실행할 때 특정 메모리 주소에서 멈출 수 있도록 중단점을 설정하는 기능을 함께 구현하였다.

# 프로그램 설명

## 프로그램 흐름도



# 모듈 정의

## 모듈 이름 : int main(void)

### 기능

프로그램이 시작되면 reset\_func()을 호출해 가상 메모리를 초기화하고, read\_opcode\_func()을 호출해 opcode 해시테이블을 채운 다음, 변수들을 모두 초기화한다. command line을 입력받아 command와 argument(s)로 분리한 뒤, command에 따라 적절한 함수를 호출해 기능을 수행한다.

### 사용 변수

int argnum – 주요 명령어 뒤의 인자들의 개수를 저장

int wrongcmdflag – 입력된 명령어의 형식이 옳은지 틀린지 구분

int hexproblemflag – 인자들이 16진수로 입력됐는지 구분

int startpoint, endpoint – dump 명령어에서 값을 표시할 메모리의 시작 주소와 끝 주소

## 모듈 이름: void help\_func(void)

### 기능

프로그램에서 사용할 수 있는 명령어의 목록을 출력한다.

### 사용 변수

없음

## 모듈 이름: void dir\_func(void)

### 기능

현재 디렉토리에 있는 파일과 하위 디렉토리의 목록을 출력한다. 현재 실행 중인 파일이면 파일 이름 뒤에 ‘\*’을 붙이고, 디렉토리이면 ‘/’을 붙인다.

### 사용 변수

DIR \*dir

struct dirent \*entry

struct stat status

## 모듈 이름: void check\_history\_num\_func(void)

### 기능

history가 몇 개인지 확인한다.

### 사용 변수

없음

## 모듈 이름: void add\_history\_func(char\* fullcmd)

### 기능

historylist에 명령줄을 추가한다.

### 사용 변수

char\* fullcmd – 사용자가 입력한 명령줄을 전달하는 매개 변수

## 모듈 이름: void show\_history\_func(void)

### 기능

history를 출력한다.

### 사용 변수

없음

## 모듈 이름: void dump\_func(int start, int end)

### 기능

start 주소의 메모리부터 end 주소의 메모리까지 내용을 출력한다.

### 사용 변수

int start, end – 메모리와 시작 주소와 끝 주소를 전달하는 매개 변수

int linenum – 출력될 줄 개수

int startaddr, endaddr – 좌측에 표시하는 16진수 메모리 주소

## 모듈 이름: void edit\_func(int address, int value)

### 기능

해당 주소 메모리에 해당 값을 저장한다.

### 사용 변수

int address – 메모리 주소를 전달하는 매개 변수

int value – 저장할 값을 전달하는 매개 변수

## 모듈 이름: void fill\_func(int start, int end, int value)

### 기능

start 주소의 메모리부터 end 주소의 메모리까지 해당 값을 저장한다.

### 사용 변수

int start, end – 메모리의 시작 주소와 끝 주소를 전달하는 매개 변수

int value – 저장할 값을 전달하는 매개 변수

## 모듈 이름: reset\_func(void)

### 기능

가상 메모리를 초기화한다.

### 사용 변수

없음

## 모듈 이름: int read\_opcode\_func(void)

### 기능

opcode.txt를 읽어들여서 opcode 해시테이블을 채운다. opcode.txt가 없으면 -1을 반환한다.

### 사용 변수

char \*temp1, \*temp2, \*temp3 – opcode.txt에서 opcode, mnemonic, format을 추출하여 임시 저장

## 모듈 이름: void add\_hash\_func(int op, char \*mn, char \*fm)

### 기능

read\_opcode\_func() 내부에서 호출되는 함수로, 읽어들인 opcode, mnemonic, format을 해시테이블에 저장한다. key는 (int)sqrt(opcode)이다.

### 사용 변수

int key – 해시테이블의 키값

int op – 저장하려는 opcode를 전달하는 매개 변수

char \*mn – 저장하려는 mnemonic을 전달하는 매개 변수

char \*fm – 저장하려는 format을 전달하는 매개 변수

## 모듈 이름: void opcode\_list\_func(void)

### 기능

opcode 해시테이블을 출력한다.

### 사용 변수

없음

## 모듈 이름: int find\_opcode\_func(char \*mn, int option)

### 기능

option = 1일 때, 해시테이블에 mnemonic이 존재하면 1을 반환하고, 그렇지 않으면 0을 반환한다. opcode.txt가 존재하지 않으면 -1을 반환한다. option = 2일 때, 해시테이블에 mnemonic이 존재하면 그 값을 반환하고, 그렇지 않으면 -1을 반환한다.

### 사용 변수

char \*mn - 찾으려는 mnemonic을 전달하는 매개 변수

int option – 1 : mnemonic의 존재 여부 확인, 2 : mnemonic의 값을 전달

## 모듈 이름: int type\_func(char \*filename)

### 기능

전달된 문자열을 이름으로 가지는 파일의 내용을 출력한다. 파일이 존재하면 0을, 없으면 -1을 반환한다.

### 사용 변수

char buffer[1000] – 파일의 내용을 담는 버퍼

## 모듈 이름: int key\_for\_symbol\_func(char \*sym)

### 기능

전달된 symbol(mnemonic)을 해시테이블에 집어넣기 위해 key값을 지정하여 반환한다. key값은 문자열 안의 모든 문자의 ASCII값을 더한 뒤 해시테이블 사이즈(20)으로 나눈 나머지이다.

### 사용 변수

int sum – 문자열 안의 모든 문자의 ASCII 값의 총합

## 모듈 이름: int add\_symbol\_func(char \*sym, int addr)

### 기능

asm 파일 안의 label과 그 주소값을 심볼테이블에 집어넣는다.

### 사용 변수

int key – label의 key값, key\_for\_symbol\_func 함수를 이용해서 저장

## 모듈 이름: int find\_label\_func(char \*mn)

### 기능

option = 1일 때, 심볼테이블에 label이 이미 있으면 1을 반환하고, 없으면 0을 반환한다. option = 2일 때, label이 있으면 그 주소값을 반환하고, 없으면 -1을 반환한다.

### 사용 변수

int option – 1 : label의 존재 여부 확인, 2 : label의 주소값 반환

## 모듈 이름: char \*format\_for\_symbol\_func(char \*mn)

### 기능

입력된 symbol(mnemonic)의 format을 문자열로 반환한다.

### 사용 변수

없음

## 모듈 이름: void symbol\_print\_func(void)

### 기능

심볼테이블에 저장된 label들을 내림차순으로 정렬하여 주소값과 함께 출력한다. 내림차순으로 정렬하기 위해 selection sort와 strcpy 함수를 사용하였다.

### 사용 변수

char symbolname[100][10] – 심볼테이블에 저장된 label들을 쉽게 정렬하기 위한 문자열 배열

int symboladdr[100] - label들의 주소값을 저장하는 배열

## 모듈 이름: int assemble\_func(char \*filename)

### 기능

주어진 asm 파일을 한 줄씩 읽으면서 locctr를 할당하여 intermediate 파일을 생성하고, 이 intermediate 파일을 읽으면서 코드 format과 addressing 방식에 따라 적절하게 object code를 생성하여 lst 파일과 obj 파일을 생성한다.

### 사용 변수

(너무 많아서 생략)

## 모듈 이름: void progaddr\_func(int address)

### 기능

프로그램을 로드할 때의 메모리의 시작 주소를 지정하는 함수이다.

### 사용 변수

int address – 프로그램의 시작 주소

## 모듈 이름: int loader\_func(char \*fname1, char \*fname2, char \*fname3)

### 기능

최대 세 개의 프로그램을 메모리에 로드하고, external symbol table을 출력하는 함수이다.

### 사용 변수

char \*fname1, \*fname2, \*fname3 – 로드되는 프로그램의 파일 이름

int totallength – 로드되는 프로그램의 총 길이

int drflag\_1, drflag\_2, drflag\_3 – 각 프로그램에 D, R 레코드가 있는지 확인하는 변수

## 모듈 이름: int extdef\_func(FILE \*fp, int startaddr, char \*progname, int drflag)

### 기능

주어진 obj 파일에서 D 레코드를 읽어 extdef symbol과 그 주소를 external symbol table에 저장하는 함수이다.

### 사용 변수

char objline[MAX\_CMD\_LEN] – 각 레코드를 문자열로 저장하는 데 사용하는 변수

char hexstring[10] – 16진수 주소가 적힌 문자열을 숫자로 변환할 때 사용하는 변수

## 모듈 이름: void extref\_func(FILE \*fp, char \*progname, int drflag)

### 기능

주어진 obj 파일에서 R 레코드를 읽어 extref symbol의 번호를 저장한다.

### 사용 변수

char objline[MAX\_CMD\_LEN] – 각 레코드를 문자열로 저장하는 데 사용하는 변수

char symbolname[7] – extref symbol 이름을 저장하는 변수

char number[3] – extref symbol의 번호를 저장하는 변수

## 모듈 이름: void tmrecord\_func(FILE \*fp, char \*progname, int drflag)

### 기능

주어진 obj 파일에서 T 레코드를 읽어 메모리에 로드하고, M 레코드를 읽어 적절한 메모리 위치에서 그 정보를 수정한다.

### 사용 변수

char objline[MAX\_CMD\_LEN] – 각 레코드를 문자열로 저장하는 데 사용하는 변수

char hexstring[7], temphexstr[9] - 16진수 주소가 적힌 문자열을 숫자로 변환할 때 사용하는 변수

int address – 메모리에 프로그램을 로드할 때 주소값으로 사용하는 변수

int recpos – M 레코드에서 half byte의 개수를 저장하는 변수

## 모듈 이름: void add\_bp\_func(int address)

### 기능

중단점을 추가하는 함수이다.

### 사용 변수

int address – 추가될 중단점이 저장된 매개변수

## 모듈 이름: void bp\_print\_func(void)

### 기능

추가된 중단점들을 출력하는 함수이다.

### 사용 변수

int i – for문을 사용해서 중단점이 있는지 확인할 때 사용하는 변수

## 모듈 이름: void bp\_clear\_func(void)

### 기능

추가된 중단점들을 모두 삭제하는 함수이다.

### 사용 변수

(없음)

## 모듈 이름: int find\_mnemonic\_func(int opcode, char \*mnemonic)

### 기능

메모리에 로드된 instruction에서 opcode를 추출해서 mnemonic을 찾는 함수이다.

### 사용 변수

int i – for문으로 hashtable을 탐색할 때 사용하는 변수

int opcode – 메모리에 로드된 instruction에서 추출된 opcode를 저장하는 변수

char \*mnemonic – 찾은 mnemonic을 문자열로 저장하는 변수

## 모듈 이름: void run\_func(int totallength, int \*currbp, int \*nextbp)

### 기능

메모리에 로드된 프로그램에 저장된 각 instruction에 대해 적절히 처리하여 프로그램을 실행하고, 그 중 중단점을 만나면 실행을 중지하는 함수이다.

### 사용 변수

int \*currbp, \*nextbp – 다음 중단점과 다다음 중단점을 저장하는 변수

int bytenum – instruction의 format을 저장하는 함수

int firstreg, secondreg – format 2일 때 필요하면 레지스터의 값을 저장하는 함수

int endaddr – 프로그램의 끝주소, 프로그램이 종료되어야 하는지 확인할 때 사용

int comp – COMP, COMPR 등의 instruction에서의 결과값을 저장하는 변수

int meetbpflag – 중단점을 만났는지 확인하는 변수

# 전역 변수 정의

## int historynum

history의 개수를 저장한다.

## unsigned char mymemory[MEGABYTE]

가상 메모리의 역할을 한다.

## HISTORYLIST \*historyhead, \*historytemp, \*historynewnode

history를 기록하고 출력하는 데 사용한다.

## HASH \*hastable[20], \*hashhead, \*hashtemp, \*hashnewnode

해시테이블을 구성하고 mnemonic을 찾는 데 사용한다.

## SYMBOL \*symboltable[SYMBOLTABLENUM], \*symhead, \*symtemp, \*symnewnode

심볼테이블을 구성하고 label을 찾는 데 사용한다.

## int A, X, L, PC, B, S, T

각 레지스터의 값을 저장하는 변수이다.

## short int breakpoint[MEGABYTE], int bpnum

breakpoint는 중단점을 저장하는 메모리 크기의 배열이고, bpnum은 중단점의 개수를 저장하는 변수이다.

## int progaddr, curraddr, execaddr

progaddr는 로드된 메모리의 시작 주소를 가리킨다. execaddr는 프로그램에서 가장 처음으로 실행될 instruction의 주소를 가리킨다. curraddr는 현재 실행되는 instruction의 주소를 가리킨다.

# 코드

## 20150959.h

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <dirent.h>  #include <sys/stat.h>  #include <math.h>  #define MAX\_CMD\_LEN 100  #define DISPLAYSTART 0  #define DISPLAYEND 159  #define MEGABYTE 1048576  #define HASHTABLENUM 20  #define SYMBOLTABLENUM 30  /\* error flags \*/  #define DUPLICATESYMBOL 11111  #define INVALIDOPCODE 22222  #define INAPPROPRIATEOPERAND 33333  #define UNDEFINEDSYMBOL 44444  int historynum;  int progaddr, curraddr, execaddr;  unsigned char memory[MEGABYTE];  int A, X, L, PC, B, S, T; // registers  int PCtemp;  short int breakpoint[MEGABYTE];  int bpnum;  typedef struct HISTORYLIST\_ {  int idx;  char cmdline[MAX\_CMD\_LEN];  struct HISTORYLIST\_ \*next;  } HISTORYLIST;  HISTORYLIST \*historyhead, \*historytemp, \*historynewnode;  typedef struct HASH\_ {  int opcode;  char mnemonic[10];  char format[4];  struct HASH\_ \*next;  } HASH;  HASH \*hashtable[HASHTABLENUM], \*hashhead, \*hashtemp, \*hashnewnode;  typedef struct SYMBOL\_ {  char symbol[10];  int address;  struct SYMBOL\_ \*next;  } SYMBOL;  SYMBOL \*symboltable[SYMBOLTABLENUM], \*symhead, \*symtemp, \*symnewnode;  typedef struct EXTSYM\_ {  char controlsection[7];  char symbolname[7];  char number[3];  int address;  int length;  struct EXTSYM\_ \*next;  } EXTSYM;  EXTSYM \*exthead, \*exttemp, \*extnewnode;  /\*============== FIRST PROJECT ==============\*/  void help\_func(void); // print out the list of commands  int dir\_func(void); //print out the list of directories and files in the current directory  void check\_history\_num\_func(void); // check the number of history list which will be used in add\_history\_func  void add\_history\_func(char\* fullcmd); // add the correct command to history list  void show\_history\_func(void); // print out the history list  void dump\_func(int start, int end); // show the state of memory  void edit\_func(int address, int value); // edit the memory of given address with given value  void fill\_func(int start, int end, int value); // fill the memory within given address with given value  void reset\_func(void); // reset the memory  int read\_opcode\_func(void); // generate the opcode table  void add\_hash\_func(int opcode, char \*mnemonic, char \*fm); // add each mnemonic to the opcode table  void opcode\_list\_func(void); // print out the list of opcodes  int find\_opcode\_func(char \*mnemonic, int option); // (option = 1) return the existence of mnemonic in opcode table  // (option = 2) return the opcode of mnemonic  /\*============= SECOND PROJECT =============\*/  int type\_func(char \*filename); // print out the content of the file  int key\_for\_symbol\_func(char \*sym); // generate key to the symbol for the symbol table  void add\_symbol\_func(char \*sym, int addr); // add each symbol to the symbol table  int find\_label\_func(char \*label, int option); // (option = 1) return the existence of label in symbol table  // (option = 2) return the address of label in symbol table  char\* format\_for\_opcode\_func(char \*mnemonic); // return the format of opcode in opcode table  void symbol\_print\_func(void); // print out the list of symbols  int assemble\_func(char \*filename); // read the .asm file and generate .lst file and .obj file  /\*============== THIRD PROJECT =============\*/  void progaddr\_func(int address); // assign the program start address  int loader\_func(char \*fname1, char \*fname2, char \*fname3); // load up to three (sub)programs  int extdef\_func(FILE \*fp, int startaddr, char \*progname, int drflag); // save addresses for each extdef symbols  void extref\_func(FILE \*fp, char \*progname, int drflag); // save numbers of each extref symbols  void tm\_record\_func(FILE \*fp, char \*progname, int drflag); // read T / M records and process them  void add\_bp\_func(int address); // add breakpoints  void bp\_print\_func(void); // print list of breakpoints  void bp\_clear\_func(void); // clear out all the breakpoints  int find\_mnemonic\_func(int opcode, char \*mnemonic); // find mnemonic from an opcode  void run\_func(int totallength, int \*currbp, int \*nextbp); // run loaded program |

## 20150959.c

|  |
| --- |
| #include "20150959.h"  int main(void) {  int i, length, argnum = 0, totallength = 0, currbp = -1, nextbp = -1;  int wrongcmdflag = 0, hexproblemflag = 0, dumptrialnum = 0;  int hex1 = MEGABYTE, hex2 = MEGABYTE, hex3 = MEGABYTE;  int startpoint, endpoint;  char fullcmd[MAX\_CMD\_LEN], temp[MAX\_CMD\_LEN], \*maincmd, \*arg1, \*arg2, \*arg3, \*arg4;  //--------------------------------INITIALIZE MEMORY SPACE AND VARIABLES----------------------------------------  reset\_func();  read\_opcode\_func();  add\_symbol\_func("A", 0); add\_symbol\_func("X", 1); add\_symbol\_func("L", 2);  add\_symbol\_func("PC", 8); add\_symbol\_func("SW", 9); add\_symbol\_func("B", 3);  add\_symbol\_func("S", 4); add\_symbol\_func("T", 5); add\_symbol\_func("F", 6);  A = 0; X = 0; L = 0xFFFFFF; PC = 0; B = 0; S = 0; T = 0; PCtemp = -1;  progaddr = 0; curraddr = 0; execaddr = 0;  memset(breakpoint, 0, MEGABYTE); bpnum = 0;  exthead = exttemp = extnewnode = NULL;  while(1) {  argnum = 0;  wrongcmdflag = 0;  hexproblemflag = 0;  while(1) {  printf("sicsim> ");  /\* initialize string variables \*/  memset(fullcmd, 0, MAX\_CMD\_LEN);  memset(temp, 0, MAX\_CMD\_LEN);  arg1 = arg2 = arg3 = arg4 = NULL;  //------------------------------------------READ COMMAND LINE--------------------------------------------------  fgets(fullcmd, MAX\_CMD\_LEN, stdin);  length = strlen(fullcmd);  fullcmd[--length] = '\0';  strcpy(temp, fullcmd);  /\* if the user wrote nothing \*/  if(length == 0) break;  //----------------------------------SEPARATE THE COMMAND LINE INTO PARTS--------------------------------------  maincmd = strtok(temp, " \t");  arg1 = strtok(NULL, " \t,");  arg2 = strtok(NULL, " \t,");  arg3 = strtok(NULL, " \t,");  arg4 = strtok(NULL, " \t,");  /\* decide if the main command is an existing one \*/  if(strcmp(maincmd, "h") && strcmp(maincmd, "help")  && strcmp(maincmd, "f") && strcmp(maincmd, "fill")  && strcmp(maincmd, "d") && strcmp(maincmd, "dir")  && strcmp(maincmd, "q") && strcmp(maincmd, "quit")  && strcmp(maincmd, "hi") && strcmp(maincmd, "history")  && strcmp(maincmd, "du") && strcmp(maincmd, "dump")  && strcmp(maincmd, "e") && strcmp(maincmd, "edit")  && strcmp(maincmd, "f") && strcmp(maincmd, "fill")  && strcmp(maincmd, "reset")  && strcmp(maincmd, "opcode")  && strcmp(maincmd, "opcodelist")  && strcmp(maincmd, "assemble")  && strcmp(maincmd, "type")  && strcmp(maincmd, "symbol")  && strcmp(maincmd, "progaddr")  && strcmp(maincmd, "loader")  && strcmp(maincmd, "run")  && strcmp(maincmd, "bp")) {  wrongcmdflag = 1;  break;  }  if(!maincmd) break;  else if(!arg1) argnum = 0;  else if(!arg2) argnum = 1;  else if(!arg3) argnum = 2;  else if(!arg4) argnum = 3;  else /\* argnum == 4 \*/ { wrongcmdflag = 1; break; }  //----------------------------------HANDLE EXCEPTIONS CONCERNING COMMAS---------------------------------------  /\* if there is a space before the main command \*/  if(!strcmp(maincmd, "loader")) {  for(i = 0; i < length; i++)  if(fullcmd[i] == ',') { wrongcmdflag = 1; break; }  if(wrongcmdflag) break;  }  else {  for(i = 0; i < length; i++)  if(fullcmd[i] != ' ' && fullcmd[i] != '\t')  break;  /\* there should not be a comma between maincmd and arg1 \*/  if(argnum >= 0) {  for(i += strlen(maincmd); i < length; i++) {  if(fullcmd[i] != ' ' && fullcmd[i] != '\t') {  if(fullcmd[i] == ',') {  wrongcmdflag = 1;  break;  }  else break; // start of arg1  }  }  if(wrongcmdflag) break;  }  /\* there should be a single comma between arg1 and arg2 \*/  if(argnum >= 1) {  if((i += strlen(arg1)) < length) wrongcmdflag = 1;  for( ; i < length; i++) {  if(fullcmd[i] != ' ' && fullcmd[i] != '\t') {  if(fullcmd[i] == ',') {  if(wrongcmdflag)  wrongcmdflag = 0;  else { // more than one comma  wrongcmdflag = 1;  break;  }  }  else break; // start of arg2  }  }  if(wrongcmdflag) break;  }  /\* there should be a single comma between arg2 and arg3 \*/  if(argnum >= 2) {  if((i += strlen(arg2)) < length) wrongcmdflag = 1;  for( ; i < length; i++) {  if(fullcmd[i] != ' ' && fullcmd[i] != '\t') {  if(fullcmd[i] == ',') {  if(wrongcmdflag)  wrongcmdflag = 0;  else { // more than one comma  wrongcmdflag = 1;  break;  }  }  else break; // start of arg3  }  }  if(wrongcmdflag) break;  }  /\* fullcmd should not end with comma \*/  for(i = length - 1; i >= 0; i--) {  if(fullcmd[i] != ' ' && fullcmd[i] != '\t') {  if(fullcmd[i] == ',') {  wrongcmdflag = 1;  break;  }  else break;  }  }  if(wrongcmdflag) break;  }  /\* decide if the adequate number of arguments is given for the main command \*/  if(argnum == 0) {  if(!strcmp(maincmd, "e") || !strcmp(maincmd, "edit")  || !strcmp(maincmd, "f") || !strcmp(maincmd, "fill")  || !strcmp(maincmd, "opcode")  || !strcmp(maincmd, "assemble")  || !strcmp(maincmd, "type")  || !strcmp(maincmd, "loader")) {  wrongcmdflag = 1;  break;  }  }  else { // argnum = 1, 2, 3  /\* these commands are not supposed to have arguments \*/  if(!strcmp(maincmd, "h") || !strcmp(maincmd, "help")  || !strcmp(maincmd, "d") || !strcmp(maincmd, "dir")  || !strcmp(maincmd, "q") || !strcmp(maincmd, "quit")  || !strcmp(maincmd, "hi") || !strcmp(maincmd, "history")  || !strcmp(maincmd, "reset")  || !strcmp(maincmd, "opcodelist")  || !strcmp(maincmd, "symbol")  || !strcmp(maincmd, "run")) {  wrongcmdflag = 1;  break;  }  else if(((!strcmp(maincmd, "du") || !strcmp(maincmd, "dump")) && argnum > 2)  || ((!strcmp(maincmd, "e") || !strcmp(maincmd, "edit")) && argnum != 2)  || ((!strcmp(maincmd, "f") || !strcmp(maincmd, "fill")) && argnum != 3)  || (!strcmp(maincmd, "opcode") && argnum != 1)  || (!strcmp(maincmd, "assemble") && argnum != 1)  || (!strcmp(maincmd, "type") && argnum != 1)  || (!strcmp(maincmd, "progaddr") && argnum != 1)  || (!strcmp(maincmd, "bp") && argnum > 1)) {  wrongcmdflag = 1;  break;  }  if(!strcmp(maincmd, "du") || !strcmp(maincmd, "dump")  || !strcmp(maincmd, "e") || !strcmp(maincmd, "edit")  || !strcmp(maincmd, "f") || !strcmp(maincmd, "fill")  || !strcmp(maincmd, "progaddr")  || (!strcmp(maincmd, "bp") && (strcmp(arg1, "clear")))) { // command is not opcode  /\* arg1 and arg2 are addresses; hence they are written in hexadecimal only  arg3 is hexadecimal value \*/  if(argnum >= 1) {  for(i = 0; i < strlen(arg1); i++) {  if((arg1[i] >= 65 && arg1[i] <= 70) // A ~ F  || (arg1[i] >= 97 && arg1[i] <= 102) // a ~ f  || (arg1[i] >= 48 && arg1[i] <= 57)) // 0 ~ 9  continue;  else if(i == 1 && arg1[i - 1] == '0'  && (arg1[i] == 'X' || arg1[i] == 'x'))  continue;  else {  hexproblemflag = 1;  wrongcmdflag = 1;  break;  }  }  if(wrongcmdflag == 1) break;  }  if(argnum >= 2) {  for(i = 0; i < strlen(arg2); i++) {  if((arg2[i] >= 65 && arg2[i] <= 70) // A ~ F  || (arg2[i] >= 97 && arg2[i] <= 102) // a ~ f  || (arg2[i] >= 48 && arg2[i] <= 57)) // 0 ~ 9  continue;  else if(i == 1 && arg2[i - 1] == '0'  && (arg2[i] == 'X' || arg2[i] == 'x'))  continue;  else {  hexproblemflag = 1;  wrongcmdflag = 1;  break;  }  }  if(wrongcmdflag == 1) break;  }  if(argnum == 3) {  for(i = 0; i < strlen(arg3); i++) {  if((arg3[i] >= 65 && arg3[i] <= 70) // A ~ F  || (arg3[i] >= 97 && arg3[i] <= 102) // a ~ f  || (arg3[i] >= 48 && arg3[i] <= 57)) // 0 ~ 9  continue;  else if(i == 1 && arg3[i - 1] == '0'  && (arg3[i] == 'X' || arg3[i] == 'x'))  continue;  else {  hexproblemflag = 1;  wrongcmdflag = 1;  break;  }  }  if(wrongcmdflag == 1) break;  }  //---------------------------------CONVERT ARGUMENTS INTO HEXADECIMAL------------------------------------------  if(argnum >= 1) hex1 = (int)strtol(arg1, NULL, 16);  if(argnum >= 2) hex2 = (int)strtol(arg2, NULL, 16);  if(argnum == 3) hex3 = (int)strtol(arg3, NULL, 16);  }  if(wrongcmdflag == 1) break;  }  check\_history\_num\_func();  //----------------------------------------------HELP----------------------------------------------  if(!strcmp(maincmd, "h") || !strcmp(maincmd, "help")) {  add\_history\_func(fullcmd);  help\_func();  break;  }  //--------------------------------------------DIRECTORY-------------------------------------------  else if(!strcmp(maincmd, "d") || !strcmp(maincmd, "dir")) {  add\_history\_func(fullcmd);  dir\_func();  }  //----------------------------------------------QUIT----------------------------------------------  else if(!strcmp(maincmd, "q") || !strcmp(maincmd, "quit"))  return 0;  //--------------------------------------------HISTORY---------------------------------------------  else if(!strcmp(maincmd, "hi") || !strcmp(maincmd, "history")) {  add\_history\_func(fullcmd);  show\_history\_func();  }  //----------------------------------------------DUMP----------------------------------------------  else if(!strcmp(maincmd, "du") || !strcmp(maincmd, "dump")) {  if(argnum == 0) {  if(dumptrialnum == 0) {  startpoint = DISPLAYSTART;  endpoint = DISPLAYEND;  }  else {  if(endpoint == MEGABYTE - 1)  startpoint = 0;  else startpoint = endpoint + 1;  endpoint = startpoint + DISPLAYEND;  }  }  else if(argnum == 1) {  startpoint = hex1;  endpoint = startpoint + DISPLAYEND;  if(endpoint >= MEGABYTE)  endpoint = MEGABYTE - 1;  if(startpoint < 0 || startpoint >= MEGABYTE) {  printf("The address should be within boundary (0x00000 - 0xFFFFF).\n");  break;  }  }  else { // argnum == 2  startpoint = hex1;  endpoint = hex2;  if(startpoint > endpoint) {  printf("The start address should be smaller than the end address.\n");  break;  }  if(startpoint < 0 || startpoint >= MEGABYTE  || endpoint < 0 ||endpoint >= MEGABYTE) {  printf("The address should be within boundary (0x00000 - 0xFFFFF).\n");  break;  }  }  dump\_func(startpoint, endpoint);  dumptrialnum++;  add\_history\_func(fullcmd);  }  //----------------------------------------------EDIT----------------------------------------------  else if(!strcmp(maincmd, "e") || !strcmp(maincmd, "edit")) {  edit\_func(hex1, hex2);  add\_history\_func(fullcmd);  }  //----------------------------------------------FILL----------------------------------------------  else if(!strcmp(maincmd, "f") || !strcmp(maincmd, "fill")) {  if(hex1 > hex2) {  printf("The start address should be smaller than the end address.\n");  break;  }  fill\_func(hex1, hex2, hex3);  add\_history\_func(fullcmd);  }  //---------------------------------------------RESET----------------------------------------------  else if(!strcmp(maincmd, "reset")) {  reset\_func();  add\_history\_func(fullcmd);  }  //--------------------------------------------OPCODE----------------------------------------------  else if(!strcmp(maincmd, "opcode")) {  int found = find\_opcode\_func(arg1, 1);  if(!found) break;  add\_history\_func(fullcmd);  }  //------------------------------------------OPCODELIST--------------------------------------------  else if(!strcmp(maincmd, "opcodelist")) {  opcode\_list\_func();  add\_history\_func(fullcmd);  }  //-------------------------------------------ASSEMBLE---------------------------------------------  else if(!strcmp(maincmd, "assemble")) {  assemble\_func(arg1);  add\_history\_func(fullcmd);  }  //---------------------------------------------TYPE-----------------------------------------------  else if(!strcmp(maincmd, "type")) {  type\_func(arg1);  add\_history\_func(fullcmd);  }  //--------------------------------------------SYMBOL----------------------------------------------  else if(!strcmp(maincmd, "symbol")) {  symbol\_print\_func();  add\_history\_func(fullcmd);  }  //--------------------------------------------SYMBOL----------------------------------------------  else if(!strcmp(maincmd, "progaddr")) {  if(argnum == 0) progaddr\_func(0);  else progaddr\_func(hex1);  add\_history\_func(fullcmd);  }  else if(!strcmp(maincmd, "loader")) {  totallength = loader\_func(arg1, arg2, arg3);  add\_history\_func(fullcmd);  }  else if(!strcmp(maincmd, "bp")) {  if(argnum == 0)  bp\_print\_func();  else if(argnum == 1) {  if(!strcmp(arg1, "clear")) bp\_clear\_func();  else add\_bp\_func(hex1);  }  add\_history\_func(fullcmd);  }  else if(!strcmp(maincmd, "run")) {  run\_func(totallength, &currbp, &nextbp);  add\_history\_func(fullcmd);  }  }  if(wrongcmdflag) {  if(hexproblemflag)  printf("The arguments should be given in hexadecimal form.\n");  else  printf("Enter the proper command. Enter \"help\" to learn more.\n");  }  }  return 0;  } |

## first\_project.c

|  |
| --- |
| #include "20150959.h"  void help\_func(void) {  printf("h[elp]\nd[ir]\nq[uit]\nhi[story]\ndu[mp] [start, end] \  \ne[dit] address, value\nf[ill] start, end, value\nreset \  \nopcode mnemonic\nopcodelist\nassemble filename\ntype filename\nsymbol\  \nprogaddr [address]\nloader [filename1] [filename2] [filename3]\nrun\nbp [address|clear]\n");  return;  }  int dir\_func(void) {  DIR \*dir = NULL;  struct dirent \*entry = NULL;  struct stat status;  if((dir = opendir("."))) {  while((entry = readdir(dir))) {  if(stat(entry->d\_name, &status) < 0)  perror("stat() : ");  if(S\_ISDIR(status.st\_mode))  printf("%s/\t", entry->d\_name);  else if(S\_ISREG(status.st\_mode)) {  if((status.st\_mode & S\_IEXEC) != 0)  printf("%s\*\t", entry->d\_name);  else  printf("%s\t", entry->d\_name);  }  }  }  printf("\n");  closedir(dir);  return 0;  }  void check\_history\_num\_func(void) {  if (!historyhead) historynum = 0;  else {  historytemp = historyhead;  while(historytemp->next) {  historytemp = historytemp->next;  historynum = historytemp->idx;  }  }  return;  }  void add\_history\_func(char\* fullcmd) {  if(!historyhead) {  historyhead = (HISTORYLIST \*)malloc(sizeof(HISTORYLIST));  historyhead->idx = ++historynum;  strcpy(historyhead->cmdline, fullcmd);  historyhead->next = NULL;  }  else {  historytemp = historyhead;  while(historytemp->next)  historytemp = historytemp->next;  historynewnode = (HISTORYLIST \*)malloc(sizeof(HISTORYLIST));  historynewnode->idx = historynum + 1;  strcpy(historynewnode->cmdline, fullcmd);  historynewnode->next = NULL;  historytemp->next = historynewnode;  }  return;  }  void show\_history\_func(void) {  if(historyhead) {  historytemp = historyhead;  while(historytemp->next) {  printf("%d\t%s\n", historytemp->idx, historytemp->cmdline);  historytemp = historytemp->next;  }  printf("%d\t%s\n", historytemp->idx, historytemp->cmdline);  }  return;  }  void dump\_func(int start, int end) {  int linenum = 0, i, j, startblank, startaddr, endaddr;  startblank = start % 16;  startaddr = (start / 16) \* 16;  endaddr = (end / 16) \* 16;  for(i = startaddr; i <= endaddr; i += 16)  linenum++;  for(i = 0; i < linenum; i++) {  if(startaddr != endaddr) { // multiple lines to print  if (i == 0) { // first line  printf("%.5X ", startaddr); // address  for(j = 0; j < startblank; j++) // blank  printf(" ");  for(j = start; j < startaddr + 16; j++) // hex values  printf("%02X ", memory[j]);  printf(" ; ");  for(j = 0; j < startblank; j++) // blank values  printf(".");  for(j = start; j < startaddr + 16; j++) { // ASCII values  if (memory[j] >= 0x20 && memory[j] <= 0x7E)  printf("%c", memory[j]);  else  printf(".");  }  printf("\n");  }  else if (i == linenum - 1) { // last line  printf("%.5X ", startaddr + i \* 16); // address  for(j = startaddr + i \* 16; j <= end; j++) // hex values  printf("%02X ", memory[j]);  for(j = end + 1; j < startaddr + (i + 1) \* 16; j++)  printf(" "); // blank  printf(" ; ");  for(j = startaddr + i \* 16; j <= end; j++) {  if (memory[j] >= 0x20 && memory[j] <= 0x7E)  printf("%c", memory[j]); // ASCII values  else  printf(".");  }  for(j = end + 1; j < startaddr + (i + 1) \* 16; j++)  printf("."); // blank values  printf("\n");  }  else {  printf("%.5X ", startaddr + i \* 16); // address  for(j = startaddr + i \* 16; j < startaddr + (i + 1) \* 16; j++)  printf("%02X ", memory[j]); // hex values  printf(" ; ");  for(j = startaddr + i \* 16; j < startaddr + (i + 1) \* 16; j++) {  if (memory[j] >= 0x20 && memory[j] <= 0x7E)  printf("%c", memory[j]); // ASCII values  else  printf(".");  }  printf("\n");  }  }  else { // single line to print  printf("%.5X ", startaddr); // address  for(j = 0; j < startblank; j++)  printf(" "); // blank  for(j = start; j <= end; j++) // hex values  printf("%02X ", memory[j]);  for(j = end + 1; j < startaddr + (i + 1) \* 16; j++)  printf(" "); // blank  printf(" ; ");  for(j = 0; j < startblank; j++)  printf(".");  for(j = start; j <= end; j++) {  if (memory[j] >= 0x20 && memory[j] <= 0x7E)  printf("%c", memory[j]); // ASCII values  else  printf(".");  }  for(j = end + 1; j < startaddr + (i + 1) \* 16; j++)  printf("."); // blank values  printf("\n");  }  }  return;  }  void edit\_func(int address, int value) {  memory[address] = value;  return;  }  void fill\_func(int start, int end, int value) {  memset(&memory[start], value, end - start + 1);  return;  }  void reset\_func(void) {  memset(memory, 0, MEGABYTE);  return;  }  int read\_opcode\_func(void) {  int opcode;  char \*temp1, \*temp2, \*temp3;  char oneline[30], \*result;  FILE \*fp = fopen("opcode.txt", "r");  if(!fp) {  printf("Failed to open opcode.txt.\n");  return -1;  }  while((result = fgets(oneline, 30, fp))) {  temp1 = strtok(oneline, " \t");  temp2 = strtok(NULL, " \t");  temp3 = strtok(NULL, " \t");  opcode = (int)strtol(temp1, NULL, 16);  add\_hash\_func(opcode, temp2, temp3);  }  fclose(fp);  return 0;  }  void add\_hash\_func(int opcode, char \*mnemonic, char \*format) {  int key = 0, i;  for(i = 0; i < strlen(mnemonic); i++)  key += (int)mnemonic[i];  key %= HASHTABLENUM;  while(1) {  if(!hashtable[key]) {  hashtable[key] = (HASH\*)malloc(sizeof(HASH));  hashtable[key]->opcode = opcode;  strcpy(hashtable[key]->mnemonic, mnemonic);  strcpy(hashtable[key]->format, format);  hashtable[key]->next = NULL;  }  else {  hashtemp = hashtable[key];  while(hashtemp->next)  hashtemp = hashtemp->next;  hashnewnode = (HASH\*)malloc(sizeof(HASH));  hashnewnode->opcode = opcode;  strcpy(hashnewnode->mnemonic, mnemonic);  strcpy(hashnewnode->format, format);  hashnewnode->next = NULL;  hashtemp->next = hashnewnode;  }  break;  }  return;  }  void opcode\_list\_func(void) {  int i;  hashtemp = NULL;  for(i = 0; i < HASHTABLENUM; i++) {  hashtemp = hashtable[i];  printf("%d : ", i);  while(hashtemp) {  if(!hashtemp->next)  printf("[%s,%X]", hashtemp->mnemonic, hashtemp->opcode);  else  printf("[%s,%X] -> ", hashtemp->mnemonic, hashtemp->opcode);  hashtemp = hashtemp->next;  }  printf("\n");  }  return;  }  int find\_opcode\_func(char \*mnemonic, int option) {  int i, key = 0, found = 0;  hashtemp = NULL;  for(i = 0; i < strlen(mnemonic); i++)  key += (int)mnemonic[i];  key %= HASHTABLENUM;  hashtemp = hashtable[key];  while(hashtemp) {  if(!strcmp(mnemonic, hashtemp->mnemonic)) {  found = 1;  if(option == 1)  printf("Opcode is %X.\n", hashtemp->opcode);  else if(option == 2)  return hashtemp->opcode;  break;  }  hashtemp = hashtemp->next;  }  if(option == 1 && found == 0)  printf("Enter the correct mnemonic.\n");  if(option == 1) return found;  else return -1;  } |

## second\_project.c

|  |
| --- |
| #include "20150959.h"  int type\_func(char \*filename) {  char \*filestr, buffer[1000];  FILE \*fp = fopen(filename, "r");  if(!fp) {  printf("Failed to open %s.\n", filename);  return -1;  }  while (1) {  filestr = fgets(buffer, 1000, fp);  if (filestr == NULL) break;  printf("%s", filestr);  }  printf("\n");  return 0;  }  int key\_for\_symbol\_func(char \*sym) {  int i, sum = 0;  for(i = 0; i < strlen(sym); i++)  sum += sym[i];  return sum % SYMBOLTABLENUM;  }  void add\_symbol\_func(char \*sym, int addr) {  int key = key\_for\_symbol\_func(sym);  if(!symboltable[key]) {  symboltable[key] = (SYMBOL \*)malloc(sizeof(SYMBOL));  strcpy(symboltable[key]->symbol, sym);  symboltable[key]->address = addr;  symboltable[key]->next = NULL;  }  else {  symtemp = symboltable[key];  while(symtemp->next && strcmp(symtemp->symbol, sym))  symtemp = symtemp->next;  symnewnode = (SYMBOL \*)malloc(sizeof(SYMBOL));  strcpy(symnewnode->symbol, sym);  symnewnode->address = addr;  symnewnode->next = NULL;  symtemp->next = symnewnode;  }  }  int find\_label\_func(char \*label, int option) {  int i;  for(i = 0; i < SYMBOLTABLENUM; i++) {  symtemp = symboltable[i];  while(symtemp) {  if(!strcmp(symtemp->symbol, label)) {  if(option == 1) return 1;  if(option == 2) return symtemp->address;  }  symtemp = symtemp->next;  }  }  if(option == 1) return 0;  else return -1;  }  char \*format\_for\_opcode\_func(char \*mn) {  int i;  for(i = 0; i < HASHTABLENUM; i++) {  hashtemp = hashtable[i];  while(hashtemp) {  if(!strcmp(hashtemp->mnemonic, mn))  return hashtemp->format;  hashtemp = hashtemp->next;  }  }  return "";  }  void symbol\_print\_func(void) {  int i, j, n, maxidx;  char symbolname[100][10] = {0}, tempstr[10] = "";  int symboladdr[100], tempaddr = 0;  symtemp = symhead = symnewnode = NULL;  n = 0;  for(i = 0; i < SYMBOLTABLENUM; i++) {  symtemp = symboltable[i];  while(symtemp) {  if(strcmp(symtemp->symbol, "A") && strcmp(symtemp->symbol, "X") && strcmp(symtemp->symbol, "L")  && strcmp(symtemp->symbol, "PC") && strcmp(symtemp->symbol, "SW") && strcmp(symtemp->symbol, "B")  && strcmp(symtemp->symbol, "S") && strcmp(symtemp->symbol, "T") && strcmp(symtemp->symbol, "F")) {  strcpy(symbolname[n], symtemp->symbol);  symboladdr[n++] = symtemp->address;  }  symtemp = symtemp->next;  }  }  for(i = 0; i < n - 1; i++) {  maxidx = i;  for(j = i + 1; j < n; j++)  if(strcmp(symbolname[j], symbolname[maxidx]) > 0)  maxidx = j;  strcpy(tempstr, symbolname[maxidx]); strcpy(symbolname[maxidx], symbolname[i]); strcpy(symbolname[i], tempstr);  tempaddr = symboladdr[maxidx]; symboladdr[maxidx] = symboladdr[i]; symboladdr[i] = tempaddr;  }  for(i = 0; i < n; i++)  printf("\t%s\t%04X\n", symbolname[i], symboladdr[i]);  }  int assemble\_func(char \*filename) {  int i, line = 0, startaddr, locctr, errorflag = 0, bytenum = 0, pglength = 0, constflag = 0, errorcount = 0;  int opcodenum, targetaddr, disp;  char \*reg1, \*reg2;  char lstfname[10] = "", objfname[10] = "", record[70] = "";  char asmline[MAX\_CMD\_LEN] = "", templine[MAX\_CMD\_LEN] = "";  int N = 0, I = 0, X = 0, B = 0, P = 0, E = 0, pgctr = 0, base = 0;  char \*temp1, \*temp2, \*temp3, \*temp4, \*temp5;  char label[10] = "", opcode[10] = "", operand[10] = "", format[5] = "", comment[50] = "";  char objcode[9] = "", hexstring[9] = "", tempoperand[10] = "";  FILE \*fp, \*itmfp, \*lstfp, \*objfp, \*errfp;  for(i = 0; i < strlen(filename); i++)  if(filename[i] == '.')  break;  strncpy(lstfname, filename, i); strcat(lstfname, ".lst");  strncpy(objfname, filename, i); strcat(objfname, ".obj");  lstfname[strlen(lstfname)] = objfname[strlen(objfname)] = '\0';  if(filename[strlen(filename) - 3] != 'a'  || filename[strlen(filename) - 2] != 's'  || filename[strlen(filename) - 1] != 'm') {  printf("The input file is not .asm file.\n"); return -1; }  fp = fopen(filename, "r");  if(!fp) { printf("Failed to open %s.\n", filename); return -1; }  itmfp = fopen("intermediate", "w");  if(!itmfp) { printf("Failed to open intermediate file.\n"); return -1; }  errfp = fopen("errorflagfile", "w");  if(!errfp) { printf("Failed to open error flag file.\n"); }  fprintf(errfp, "-----------PASS 1 ERRORS-----------\n");  /\*------------------ READ ASM FILE LINE ----------------------\*/  while(fgets(asmline, MAX\_CMD\_LEN, fp)) {  asmline[strlen(asmline) - 1] = '\0';  temp1 = temp2 = temp3 = temp4 = temp5 = NULL;  memset(label, 0, 10); memset(opcode, 0, 10); memset(operand, 0, 10); memset(format, 0, 5);  strcpy(templine, asmline);  temp1 = strtok(templine, " \t");  temp2 = strtok(NULL, " \t");  temp3 = strtok(NULL, " \t");  temp4 = strtok(NULL, " \t");  temp5 = strtok(NULL, " \t");  /\*----------------- PREPROCESSING INPUT FOR PASS 1 -------------------\*/  // nothing written  if(!temp1) continue;  line += 5;  // if comment then pass  if(temp1[0] == '.')  fprintf(itmfp, "%3d\t%04X\t\t%s\n", line, locctr, asmline);  // only opcode exists  else if(!temp2) strcpy(opcode, temp1);  // no label  else if(!temp3) { strcpy(opcode, temp1); strcpy(operand, temp2); }  else if(!temp4) {  // no label, two operands  if(temp2[strlen(temp2) - 1] == ',' || temp3[0] == ',') {  strcpy(opcode, temp1);  strcpy(operand, temp2); strcat(operand, temp3);  }  // no label, one operand  else { strcpy(label, temp1); strcpy(opcode, temp2); strcpy(operand, temp3);}  }  // no label, two operands with separate comma  else if(!strcmp(temp3, ",")) {  strcpy(opcode, temp1);  strcpy(operand, temp2); strcat(operand, temp3); strcat(operand, temp4);  }  // label exists and two operands  else if(!strcmp(temp4, ",")) {  strcpy(label, temp1); strcpy(opcode, temp2);  strcpy(operand, temp3); strcat(operand, temp4); strcat(operand, temp5);  }  /\*---------------------------- PASS 1 ----------------------------\*/  if(line == 5) {  if(!strcmp(opcode, "START")) {  startaddr = locctr = (int)strtol(operand, NULL, 16);  fprintf(itmfp, "%3d\t%04X\t%s\t%s\t%s\n", line, locctr, label, opcode, operand);  continue;  }  else {  printf("WARNING: The program does not start with START directive.\n");  startaddr = locctr = 0;  }  }  if(temp1[0] != '.') { // while OPCODE != "END"  if(strcmp(opcode, "END")) { // if this is not a comment line  if(strcmp(label, "")) { // if there is a symbol in the LABEL field then  errorflag = find\_label\_func(label, 1); // search SYMTAB for LABEL, if found then set error flag  if(errorflag) {  errorcount++; errorflag = DUPLICATESYMBOL;  fprintf(errfp, "line %d : duplicate symbol (%s)\n", line, label);  }  else add\_symbol\_func(label, locctr);  }  // finding out the format  if(opcode[0] == '+') bytenum = 4;  else if(find\_opcode\_func(opcode, 2) != -1) {  strcpy(format, format\_for\_opcode\_func(opcode));  bytenum = (int)(format[0] - '0');  }  else if(!strcmp(opcode, "WORD")) bytenum = 3;  else if(!strcmp(opcode, "RESW")) bytenum = 3 \* (int)strtol(operand, NULL, 10);  else if(!strcmp(opcode, "RESB")) bytenum = (int)strtol(operand, NULL, 10);  else if(!strcmp(opcode, "BYTE")) {  if(operand[0] == 'C') bytenum = (strlen(operand)- 3);  if(operand[0] == 'X' && operand[1] == '\'') bytenum = ((strlen(operand) - 3) / 2);  }  else if(!strcmp(opcode, "BASE")) bytenum = 0;  else {  errorcount++; errorflag = INVALIDOPCODE; bytenum = 0;  fprintf(errfp, "line %d : invalid opcode (%s)\n", line, opcode);  }  fprintf(itmfp, "%3d\t%04X\t%s\t%s\t%s\n", line, locctr, label, opcode, operand);  }  else { // END  fprintf(itmfp, "%3d\t%04X\t%s\t%s\t%s\n", line, locctr, label, opcode, operand);  pglength = locctr - startaddr;  }  locctr += bytenum; // lext locctr = program counter  }  }  fclose(fp); fclose(itmfp);  /\*----------------- PREPROCESSING INPUT FOR PASS 2 -------------------\*/  itmfp = fopen("intermediate", "r");  objfp = fopen(objfname, "w");  lstfp = fopen(lstfname, "w");  memset(record, 0, 70);  fprintf(errfp, "\n-----------PASS 2 ERRORS-----------\n");  while(fgets(asmline, MAX\_CMD\_LEN, fp)) {  asmline[strlen(asmline) - 1] = '\0';  temp1 = temp2 = temp3 = temp4 = temp5 = NULL;  memset(label, 0, 10); memset(opcode, 0, 10); memset(operand, 0, 10);  memset(format, 0, 5); memset(objcode, 0, 9); memset(hexstring, 0, 9);  memset(comment, 0, 50); memset(tempoperand, 0, 10);  N = I = X = B = P = E = 0;  strcpy(templine, asmline);  temp1 = strtok(asmline, " \t"); // line number  temp2 = strtok(NULL, " \t"); // locctr  temp3 = strtok(NULL, " \t"); // label or opcode or '.'(comment)  if(temp3[0] != '.') {  temp4 = strtok(NULL, " \t"); // opcode or operand(s)  temp5 = strtok(NULL, " \t"); // operand(s) or NULL  }  else {  temp4 = strtok(NULL, "");  if(temp4) strcpy(comment, temp4);  }  line = (int)strtol(temp1, NULL, 10);  locctr = (int)strtol(temp2, NULL, 16);  if(temp3[0] == '.') {  if(!temp4) fprintf(lstfp, "\t.\n");  else fprintf(lstfp, "\t. %s\n", comment);  continue;  }  else if(!temp4) strcpy(opcode, temp3);  else if(!temp5) { strcpy(opcode, temp3); strcpy(operand, temp4); }  else { strcpy(label, temp3); strcpy(opcode, temp4); strcpy(operand, temp5); }  if(opcode[0] == '+') bytenum = 4;  else if(find\_opcode\_func(opcode, 2) != -1) {  strcpy(format, format\_for\_opcode\_func(opcode));  bytenum = (int)(format[0] - '0');  }  else if(!strcmp(opcode, "WORD")) bytenum = 3;  else if(!strcmp(opcode, "RESW")) bytenum = 3 \* (int)strtol(operand, NULL, 10);  else if(!strcmp(opcode, "RESB")) bytenum = (int)strtol(operand, NULL, 10);  else if(!strcmp(opcode, "BYTE")) {  if(operand[0] == 'C') bytenum = (strlen(operand) - 3);  if(operand[0] == 'X' && operand[1] == '\'') bytenum = ((strlen(operand) - 3) / 2);  }  else ;  pgctr = locctr + bytenum;  /\*---------------------------- PASS 2 ----------------------------\*/  if(line == 5) {  record[0] = 'H';  strcat(record, label);  if(strlen(label) < 6)  for(i = 0; i < (6 - strlen(label)); i++)  strcat(record, " ");  sprintf(objcode, "%06X", startaddr); strcat(record, objcode);  sprintf(objcode, "%06X", pglength); strcat(record, objcode);  record[19] = '\0';  fprintf(objfp, "%s\n", record);  memset(record, 0, 70);  if(!strcmp(opcode, "START")) {  fprintf(lstfp, "%04X\t%s\t%s\t%s\n", locctr, label, opcode, operand);  continue;  }  }  if(strcmp(opcode, "END")) { // opcode != END  // search OPTAB for OPCODE  if(((opcodenum = find\_opcode\_func(opcode, 2)) != -1)  || (bytenum == 4 && (opcodenum = find\_opcode\_func(&opcode[1], 2)) != -1)) {  if(constflag == 1) constflag = 2;  if(bytenum == 1) {  if(strcmp(operand, "")) {  errorcount++; errorflag = INAPPROPRIATEOPERAND; targetaddr = 0;  fprintf(errfp, "line %d : inappropriate operand for format 1 (%s)\n", line, operand);  }  else sprintf(objcode, "%02X", opcodenum);  }  else if(bytenum == 2) {  strcpy(tempoperand, operand);  sprintf(objcode, "%02X", opcodenum);  reg1 = strtok(tempoperand, ",");  reg2 = strtok(NULL, ",");  // reg1  if(!reg1) {  errorcount++; errorflag = UNDEFINEDSYMBOL; targetaddr = 0;  fprintf(errfp, "line %d : inappropriate operand for format 2 (%s)\n", line, operand);  }  else {  if(find\_label\_func(reg1, 2) != -1) {  if(strcmp(reg1, "A") && strcmp(reg1, "X") && strcmp(reg1, "L")  && strcmp(reg1, "PC") && strcmp(reg1, "SW") && strcmp(reg1, "B")  && strcmp(reg1, "S") && strcmp(reg1, "T") && strcmp(reg1, "F")) {  errorcount++; errorflag = INAPPROPRIATEOPERAND; targetaddr = 0;  fprintf(errfp, "line %d : inappropriate operand for format 2 (%s)\n", line, operand);  }  else objcode[2] = (char)(find\_label\_func(reg1, 2) + '0');  }  else {  if((int)strtol(reg1, NULL, 10) < 0 || (int)strtol(reg1, NULL, 10) >= 16) {  errorcount++; errorflag = INAPPROPRIATEOPERAND; targetaddr = 0;  fprintf(errfp, "line %d : inappropriate operand for format 2 (%s)\n", line, operand);  }  else objcode[2] = (char)((int)strtol(reg1, NULL, 16) + '0');  }  }  // reg2  if(reg2) {  if(find\_label\_func(reg2, 2) != -1) {  if(strcmp(reg2, "A") && strcmp(reg2, "X") && strcmp(reg2, "L")  && strcmp(reg2, "PC") && strcmp(reg2, "SW") && strcmp(reg2, "B")  && strcmp(reg2, "S") && strcmp(reg2, "T") && strcmp(reg2, "F")) {  errorcount++; errorflag = INAPPROPRIATEOPERAND; targetaddr = 0;  fprintf(errfp, "line %d : inappropriate operand for format 2 (%s)\n", line, operand);  }  else objcode[3] = (char)(find\_label\_func(reg2, 2) + '0');  }  else {  if((int)strtol(reg2, NULL, 10) < 0 || (int)strtol(reg2, NULL, 10) >= 16) {  errorcount++; errorflag = INAPPROPRIATEOPERAND; targetaddr = 0;  fprintf(errfp, "line %d : inappropriate operand for format 2 (%s)\n", line, operand);  }  else objcode[3] = (char)((int)strtol(reg2, NULL, 16) + '0');  }  }  else objcode[3] = '0';  }  else if(bytenum == 3 || bytenum == 4) {  if(strcmp(operand, "")) { // if there is a symbol in OPERAND field  if(operand[0] == '@' || operand[0] == '#') {  if(operand[0] == '@') { N = 1; I = 0; } // indirect addressing  else if(operand[0] == '#') { N = 0; I = 1; } // immediate addressing  targetaddr = find\_label\_func(&operand[1], 2);  if(targetaddr == -1) { // target address is written in number  if(operand[1] >= '0' && operand[1] <= '9') {  targetaddr = (int)strtol(&operand[1], NULL, 10);  disp = targetaddr;  }  else {  errorcount++; errorflag = UNDEFINEDSYMBOL; targetaddr = 0;  fprintf(errfp, "line %d : undefined symbol (%s)\n", line, operand);  }  }  }  else {// simple addressing  // indexed addressing  if(operand[strlen(operand) - 2] == ',' && operand[strlen(operand) - 1] == 'X') {  X = 1; operand[strlen(operand) - 1] = '\0'; operand[strlen(operand) - 1] = '\0';  targetaddr = find\_label\_func(operand, 2);  }  N = 1; I = 1;  targetaddr = find\_label\_func(operand, 2);  if(targetaddr == -1) { // target address is written in number  if(operand[0] >= '0' && operand[0] <= '9')  targetaddr = (int)strtol(operand, NULL, 10);  else {  errorcount++; errorflag = UNDEFINEDSYMBOL; targetaddr = 0;  fprintf(errfp, "line %d : undefined symbol (%s)\n", line, operand);  }  }  }  } // if symbol  else { N = 1; I = 1; disp = targetaddr = 0; } // example. RSUB  if(bytenum == 3) {  if(disp == targetaddr) { B = 0; P = 0; }  else {  disp = targetaddr - pgctr;  // PC relative address  if(disp >= -2048 && disp <= 2047) { B = 0; P = 1; }  // Base relative address  else {  B = 1; P = 0;  disp = targetaddr - base;  }  }  }  else if(bytenum == 4) { B = 0; P = 0; E = 1; disp = targetaddr; }  sprintf(objcode, "%02X", (opcodenum + N \* 2 + I \* 1));  sprintf(&objcode[2], "%01X", (X \* 8 + B \* 4 + P \* 2 + E \* 1));  if(bytenum == 3) {  sprintf(hexstring, "%08X", disp);  strcat(objcode, &hexstring[5]);  }  else if(bytenum == 4) {  sprintf(hexstring, "%08X", disp);  strcat(objcode, &hexstring[3]);  }  }  }// if opcode found  else if(!strcmp(opcode, "BASE")) {  if(operand[0] == '@' || operand[0] == '#')  base = find\_label\_func(&operand[1], 2);  else base = find\_label\_func(operand, 2);  }  else if(!strcmp(opcode, "BYTE")) {  if(operand[0] == 'C')  for(i = 2; i < strlen(operand) - 1; i++) {  sprintf(hexstring, "%02X", operand[i]);  strcat(objcode, hexstring);  }  else if(operand[0] == 'X')  strncpy(objcode, &operand[2], strlen(operand) - 3);  }  else if(!strcmp(opcode, "WORD"))  sprintf(objcode, "%06X", (int)strtol(operand, NULL, 10));  else if(!strcmp(opcode, "RESW") || !strcmp(opcode, "RESB"))  constflag = 1;  if(strlen(record) + strlen(objcode) >= 70 || constflag == 2) {  constflag = 0;  sprintf(hexstring, "%02X", (int)((strlen(record) - 9) / 2));  record[7] = hexstring[0]; record[8] = hexstring[1]; // write length of object code  fprintf(objfp, "%s\n", record);  memset(record, 0, 70);  }  if(strlen(record) == 0) {  record[0] = 'T';  sprintf(hexstring, "%06X", locctr); strcat(record, hexstring);  strcat(record, "00"); // temporarily assign length of object code  }  strcat(record, objcode);  }  else { // opcode = END  if(strlen(record) > 0) {  sprintf(hexstring, "%02X", (int)((strlen(record) - 9) / 2));  record[7] = hexstring[0]; record[8] = hexstring[1]; // write length of object code  fprintf(objfp, "%s\n", record);  memset(record, 0, 70);  record[0] = 'E';  sprintf(&record[1], "%06X", startaddr);  fprintf(objfp, "%s", record);  }  }  if(!strcmp(label, "BASE") || !strcmp(label, "END"))  fprintf(lstfp, "\t\t%s\t%s\t%s\t%s\n", label, opcode, operand, objcode);  else if(X == 1) fprintf(lstfp, "%04X\t%s\t%s\t%s,X\t%s\n", locctr, label, opcode, operand, objcode);  else fprintf(lstfp, "%04X\t%s\t%s\t%s\t\t%s\n", locctr, label, opcode, operand, objcode);  }  fclose(itmfp); fclose(objfp); fclose(errfp); fclose(lstfp);  if(errorcount) {  type\_func("errorflagfile");  printf("\nFailed to create %s and %s.\n", lstfname, objfname);  remove(lstfname); remove(objfname);  }  else printf("output file: [%s], [%s]\n", lstfname, objfname);  remove("intermediate"); remove("errorflagfile");  return 0;  } |

## third\_project.c

|  |
| --- |
| #include "20150959.h"  #include <math.h>  void progaddr\_func(int address) {  progaddr = address;  printf("Program starting address set to 0x%04X.\n", address);  return;  }  int loader\_func(char \*fname1, char \*fname2, char \*fname3) {  int totallength = progaddr, drflag\_1 = 0, drflag\_2 = 0, drflag\_3 = 0;  char progname1[7] = "", progname2[7] = "", progname3[7] = "", objline[MAX\_CMD\_LEN] = "";  FILE \*fp1, \*fp2, \*fp3;  if(exthead) {  exttemp = exthead;  while(exthead->next) {  exthead = exthead->next;  free(exttemp);  exttemp = exthead;  }  free(exttemp);  }  exthead = exttemp = extnewnode = NULL;  fp1 = fopen(fname1, "r");  if(!fp1) { printf("Failed to open %s.\n", fname1); return -1; }  while(fgets(objline, MAX\_CMD\_LEN, fp1))  if(objline[0] == 'D') { drflag\_1 = 1; break; }  fclose(fp1);  if(fname2) {  fp2 = fopen(fname2, "r");  if(!fp2) { printf("Failed to open %s.\n", fname2); return -1; }  while(fgets(objline, MAX\_CMD\_LEN, fp2))  if(objline[0] == 'D') { drflag\_2 = 1; break; }  fclose(fp2);  }  if(fname3) {  fp3 = fopen(fname3, "r");  if(!fp3) { printf("Failed to open %s.\n", fname3); return -1; }  while(fgets(objline, MAX\_CMD\_LEN, fp3))  if(objline[0] == 'D') { drflag\_3 = 1; break; }  fclose(fp3);  }  fp1 = fopen(fname1, "r");  totallength = extdef\_func(fp1, totallength, progname1, drflag\_1);  if(fname2) {  fp2 = fopen(fname2, "r");  totallength = extdef\_func(fp2, totallength, progname2, drflag\_2);  }  if(fname3) {  fp3 = fopen(fname3, "r");  totallength = extdef\_func(fp3, totallength, progname3, drflag\_3);  }  totallength -= progaddr;  extref\_func(fp1, progname1, drflag\_1); tm\_record\_func(fp1, progname1, drflag\_1);  if(fname2) { extref\_func(fp2, progname2, drflag\_2); tm\_record\_func(fp2, progname2, drflag\_2); }  if(fname3) { extref\_func(fp3, progname3, drflag\_3); tm\_record\_func(fp3, progname3, drflag\_3); }  printf("control\t\tsymbol\t\taddress\t\tlength\n");  printf("section\t\tname\n");  printf("----------------------------------------------------------\n");  exttemp = exthead;  while(exttemp) {  if(strcmp(exttemp->controlsection, ""))  printf("%s\t\t\t\t%04X\t\t%04X\n", exttemp->controlsection, exttemp->address, exttemp->length);  else printf("\t\t%s\t\t%04X\n", exttemp->symbolname, exttemp->address);  exttemp = exttemp->next;  }  printf("----------------------------------------------------------\n");  printf("\t\t\t\ttotal length\t%04X\n", totallength);  return totallength;  }  int extdef\_func(FILE \*fp, int startaddr, char \*progname, int drflag) {  int i, j, length = 0;  char objline[MAX\_CMD\_LEN] = "";  char hexstring[10] = "";  fgets(objline, MAX\_CMD\_LEN, fp);  if(objline[0] == 'H') {  extnewnode = (EXTSYM \*)malloc(sizeof(EXTSYM));  strncpy(extnewnode->controlsection, &objline[1], 6);  strcpy(progname, extnewnode->controlsection);  for(j = 0; j < 7; j++)  if(extnewnode->controlsection[j] == ' ' || extnewnode->controlsection[j] == '\t' || extnewnode->controlsection[j] == '\n') {  extnewnode->controlsection[j] = '\0'; progname[j] = '\0';  }  strcpy(extnewnode->symbolname, "");  strncpy(hexstring, &objline[7], 6);  extnewnode->address = startaddr + (int)strtol(hexstring, NULL, 16);  strncpy(hexstring, &objline[13], 6);  extnewnode->length = (int)strtol(hexstring, NULL, 16);  extnewnode->next = NULL;  length = extnewnode->length;  if(!exthead) exthead = exttemp = extnewnode;  else { exttemp->next = extnewnode; exttemp = exttemp->next; }  }  else { printf("This .obj file is not valid.\n"); return -1; }  if(drflag == 0) return startaddr + length;  fgets(objline, MAX\_CMD\_LEN, fp);  objline[strlen(objline) - 1] = '\0';  if(objline[0] == 'D') {  for(i = 1; i < strlen(objline); i += 12) {  extnewnode = (EXTSYM \*)malloc(sizeof(EXTSYM));  strcpy(extnewnode->controlsection, "");  strncpy(extnewnode->symbolname, &objline[i], 6);  for(j = 0; j < 7; j++)  if(extnewnode->symbolname[j] == ' ' || extnewnode->symbolname[j] == '\t' || extnewnode->symbolname[j] == '\n')  extnewnode->symbolname[j] = '\0';  strncpy(hexstring, &objline[i + 6], 6);  extnewnode->address = startaddr + (int)strtol(hexstring, NULL, 16);  extnewnode->length = 0;  extnewnode->next = NULL;  exttemp->next = extnewnode;  exttemp = exttemp->next;  }  }  return startaddr + length;  }  void extref\_func(FILE \*fp, char \*progname, int drflag) {  int i, j;  char objline[MAX\_CMD\_LEN] = "", symbolname[7] = "", number[3] = "";  if(drflag == 0) return;  exttemp = exthead;  while(exttemp) { memset(exttemp->number, 0, 3); exttemp = exttemp->next; }  fgets(objline, MAX\_CMD\_LEN, fp);  objline[strlen(objline) - 1] = '\0';  if(objline[0] == 'R') {  if(objline[1] >= '0' && objline[1] <= '9') {  for(i = 1; i < strlen(objline); i += 8) {  memset(symbolname, 0, 7); memset(number, 0, 3);  strncpy(number, &objline[i], 2); strncpy(symbolname, &objline[i + 2], 6);  for(j = 0; j < 7; j++)  if(symbolname[j] == ' ' || symbolname[j] == '\t' || symbolname[j] == '\n')  symbolname[j] = '\0';  exttemp = exthead;  while(exttemp) {  if(!strcmp(exttemp->controlsection, progname)) strcpy(exttemp->number, "01");  else if(!strcmp(exttemp->symbolname, symbolname)) strcpy(exttemp->number, number);  exttemp = exttemp->next;  }  }  }  }  return;  }  void tm\_record\_func(FILE \*fp, char \*progname, int drflag) {  char objline[MAX\_CMD\_LEN] = "", hexstring[7] = "", temphexstr[9] = "";  int i, address = 0, value = 0, recpos, startaddr = 0, firstnum, tempvalue = 0;  exttemp = exthead;  while(exttemp) {  if(drflag) {  if(!strcmp(exttemp->number, "01")) { startaddr = exttemp->address; break; }  }  else {  if(!strcmp(exttemp->controlsection, progname)) { startaddr = exttemp->address; break; }  }  exttemp = exttemp->next;  }  while(fgets(objline, MAX\_CMD\_LEN, fp)) {  objline[strlen(objline) - 1] = '\0';  memset(hexstring, 0, 7);  if(objline[0] == '.') continue;  else if(objline[0] == 'T') {  strncpy(hexstring, &objline[1], 6);  address = (int)strtol(hexstring, NULL, 16);  for(i = 9; i < strlen(objline); i += 2) {  memset(hexstring, 0, 7);  strncpy(hexstring, &objline[i], 2);  value = (int)strtol(hexstring, NULL, 16);  edit\_func(startaddr + address, value);  address += 1;  }  }  else if(objline[0] == 'M') {  memset(hexstring, 0, 7);  strncpy(hexstring, &objline[1], 6);  address = (int)strtol(hexstring, NULL, 16);  memset(hexstring, 0, 7);  strncpy(hexstring, &objline[7], 2);  recpos = (int)strtol(hexstring, NULL, 16);  for(i = 0; i < (recpos + 1) / 2; i++)  sprintf(&hexstring[i \* 2], "%02X", memory[startaddr + address + i]);  tempvalue = (int)strtol(hexstring, NULL, 16);  if(recpos % 2 == 1) {  hexstring[1] = '\0';  firstnum = (int)strtol(hexstring, NULL, 16);  tempvalue = tempvalue % ((int)pow((double)16, (double)recpos));  }  else firstnum = 0;  if(strlen(objline) >= 10) {  memset(hexstring, 0, 7);  if(objline[10] >= '0' || objline[10] <= '9') {  strncpy(hexstring, &objline[10], 2);  exttemp = exthead;  while(exttemp) {  if(!strcmp(exttemp->number, hexstring)) {  if(objline[9] == '+') tempvalue += exttemp->address;  else if(objline[9] == '-') tempvalue -= exttemp->address;  break;  }  exttemp = exttemp->next;  }  }  else {  strcpy(hexstring, &objline[10]);  exttemp = exthead;  while(exttemp) {  if(!strcmp(exttemp->symbolname, hexstring)) {  if(objline[9] == '+') tempvalue += exttemp->address;  else if(objline[9] == '-') tempvalue -= exttemp->address;  break;  }  exttemp = exttemp->next;  }  }  }  else tempvalue += progaddr;  memset(hexstring, 0, 7); memset(temphexstr, 0, 9);  sprintf(temphexstr, "%06X", tempvalue);  if(strlen(temphexstr) > 6) strcpy(hexstring, &temphexstr[2]);  else strcpy(hexstring, temphexstr);  tempvalue = (int)strtol(hexstring, NULL, 16);  tempvalue += firstnum \* ((int)pow((double)16, (double)recpos));  memset(hexstring, 0, 7);  sprintf(hexstring, "%06X", tempvalue);  for(i = 0; i < (recpos + 1) / 2; i++) {  memset(temphexstr, 0, 9);  strncpy(temphexstr, &hexstring[i \* 2], 2);  memory[startaddr + address + i] = (int)strtol(temphexstr, NULL, 16);  }  }  else if(objline[0] == 'E') {  if(strlen(objline) > 1) {  memset(hexstring, 0, 7);  strncpy(hexstring, &objline[1], 6);  execaddr = startaddr + (int)strtol(hexstring, NULL, 16);  }  }  }  return;  }  void add\_bp\_func(int address) {  breakpoint[address] = 1; bpnum++;  printf("[ok] create breakpoint %04X\n", address);  return;  }  void bp\_print\_func(void) {  int i;  if(bpnum == 0) { printf("No breakpoints set.\n"); return; }  printf("breakpoints\n");  printf("-----------\n");  for(i = 0; i < MEGABYTE; i++)  if(breakpoint[i] == 1) printf("%04X\n", i);  return;  }  void bp\_clear\_func(void) {  memset(breakpoint, 0, MAX\_CMD\_LEN);  bpnum = 0;  printf("[ok] clear all breakpoints\n");  }  int find\_mnemonic\_func(int opcode, char \*mnemonic) {  int i;  for(i = 0; i < HASHTABLENUM; i++) {  hashtemp = hashtable[i];  while(hashtemp) {  if((hashtemp->opcode >= (opcode - 3)) && (hashtemp->opcode <= opcode)) {  strcpy(mnemonic, hashtemp->mnemonic);  return hashtemp->opcode;  }  hashtemp = hashtemp->next;  }  }  return -1;  }  void run\_func(int totallength, int \*currbp, int \*nextbp) { //curraddr : current address  int opcode, endflag = 0, j, bytenum, temp, firstreg, secondreg;  int endaddr = progaddr + totallength;  int n = 0, i = 0, x = 0, b = 0, p = 0, e = 0;  int targetaddr, disp, comp = 0, meetbpflag = 0;  char mnemonic[10], format[5] = "";  if(PCtemp == -1) { PCtemp = PC; L = endaddr; }  /\* program execution from beginning \*/  if(\*currbp == -1) {  curraddr = execaddr;  if(bpnum == 0) \*currbp = \*nextbp = endaddr;  else  for(j = curraddr; j <= endaddr; j++)  if(breakpoint[j] == 1) { \*currbp = j; break; }  }  while(1) {  x = 0; b = 0; p = 0; e = 0; bytenum = 0;  /\* process of finding opcode and addressing mode \*/  temp = memory[curraddr];  opcode = find\_mnemonic\_func(temp, mnemonic);  if(bpnum > 0) {  for(j = \*currbp + 1; j <= endaddr; j++)  if(breakpoint[j] == 1) { \*nextbp = j; break; }  if(j == endaddr + 1) \*nextbp = endaddr;  }  if(strcmp(mnemonic, "")) {  strcpy(format, format\_for\_opcode\_func(mnemonic));  bytenum = (int)(format[0] - '0');  if((bytenum == 3 ) && ((memory[curraddr + 1] & 0x10) == 0x10)) { e = 1; bytenum = 4; }  PC = curraddr + bytenum;  // if PC is bigger than program length, then the loop should be stopped  if(PC > endaddr) { PC = endaddr; endflag = 1; break; }  for(j = 0; j < bytenum; j++)  if(curraddr + j == \*currbp) { meetbpflag = 1; break; }  if(meetbpflag) { PCtemp = PC; PC = \*currbp; break; }  /\* format 3 / 4 \*/  if(bytenum == 3 || bytenum == 4) {  switch(temp - opcode) {  case 1 : n = 0; i = 1; break; // 1 : immediate addressing  case 2 : n = 1; i = 0; break; // 2 : indirect addressing  case 3 : n = 1; i = 1; break; // 3 : simple addressing  }  if((memory[curraddr + 1] & 0x80) == 0x80) x = 1;  if((memory[curraddr + 1] & 0x40) == 0x40) b = 1;  if((memory[curraddr + 1] & 0x20) == 0x20) p = 1;  /\* format 3 \*/  if(e == 0) {  disp = ((memory[curraddr + 1] & 0x0F) << 8) + memory[curraddr + 2];  if((disp & 0x800) == 0x800) disp |= 0xF000;  if(b == 1 || p == 1) {  if(p == 1) targetaddr = disp + PC;  else if(b == 1) targetaddr = disp + B;  targetaddr &= 0xFFFF;  /\* indirect addressing \*/  if(n == 1 && i == 0) {  temp = (memory[targetaddr] << 16);  temp += (memory[targetaddr + 1] << 8);  temp += memory[targetaddr + 2];  targetaddr = temp;  }  }  /\* immediate / simple addressing \*/  else targetaddr = disp;  }  /\* format 4 \*/  else {  disp = ((memory[curraddr + 1] & 0x0F) << 16);  disp += (memory[curraddr + 2] << 8);  disp += memory[curraddr + 3];  targetaddr = disp;  }  /\* indexed addressing \*/  if(x == 1) targetaddr += X;  }  else if(bytenum == 2) disp = memory[curraddr + 1];  else if(bytenum == 1) ;  /\* function for each instruction \*/  if(!strcmp(mnemonic, "LDA")) {  if(n == 0 && i == 1) A = targetaddr;  else {  A = (memory[targetaddr] << 16);  A += (memory[targetaddr + 1] << 8);  A += memory[targetaddr + 2];  }  }  else if(!strcmp(mnemonic, "LDB")) {  if(n == 0 && i == 1) B = targetaddr;  else {  B = (memory[targetaddr] << 16);  B += (memory[targetaddr + 1] << 8);  B += memory[targetaddr + 2];  }  }  else if(!strcmp(mnemonic, "LDT")) {  if(n == 0 && i == 1) T = targetaddr;  else {  T = (memory[targetaddr] << 16);  T += (memory[targetaddr + 1] << 8);  T += memory[targetaddr + 2];  }  }  else if(!strcmp(mnemonic, "LDCH")) A = (A & 0xFFFF00) + memory[targetaddr];  else if(!strcmp(mnemonic, "STA")) {  temp = A; memory[targetaddr + 2] = temp % 256;  temp = temp >> 8; memory[targetaddr + 1] = temp % 256;  temp = temp >> 8; memory[targetaddr] = temp % 256;  }  else if(!strcmp(mnemonic, "STX")) {  temp = X; memory[targetaddr + 2] = temp % 256;  temp = temp >> 8; memory[targetaddr + 1] = temp % 256;  temp = temp >> 8; memory[targetaddr] = temp % 256;  }  else if(!strcmp(mnemonic, "STL")) {  temp = L; memory[targetaddr + 2] = temp % 256;  temp = temp >> 8; memory[targetaddr + 1] = temp % 256;  temp = temp >> 8; memory[targetaddr] = temp % 256;  }  else if(!strcmp(mnemonic, "STCH")) memory[targetaddr] = (A & 0xFF);  else if(!strcmp(mnemonic, "J")) PC = targetaddr;  else if(!strcmp(mnemonic, "JSUB")) { L = PC; PC = targetaddr; }  else if(!strcmp(mnemonic, "JLT")) { if(comp == -1) PC = targetaddr; }  else if(!strcmp(mnemonic, "JEQ")) { if(comp == 0) PC = targetaddr; }  else if(!strcmp(mnemonic, "RSUB")) PC = L;  else if(!strcmp(mnemonic, "COMP")) {  if(A == targetaddr) comp = 0;  else if(A > targetaddr) comp = 1;  else comp = -1;  }  else if(!strcmp(mnemonic, "COMPR")) {  switch((disp & 0xF0) >> 4) {  case 0 : firstreg = A; break;  case 1 : firstreg = X; break;  case 2 : firstreg = L; break;  case 3 : firstreg = B; break;  case 4 : firstreg = S; break;  case 5 : firstreg = T; break;  }  switch((disp & 0x0F) >> 4) {  case 0 : secondreg = A; break;  case 1 : secondreg = X; break;  case 2 : secondreg = L; break;  case 3 : secondreg = B; break;  case 4 : secondreg = S; break;  case 5 : secondreg = T; break;  }  if(firstreg == secondreg) comp = 0;  else if(firstreg > secondreg) comp = 1;  else comp = -1;  }  else if(!strcmp(mnemonic, "CLEAR")) {  switch((disp & 0xF0) >> 4) {  case 0 : A = 0; break;  case 1 : X = 0; break;  case 2 : L = 0; break;  case 3 : B = 0; break;  case 4 : S = 0; break;  case 5 : T = 0; break;  }  }  else if(!strcmp(mnemonic, "TIXR")) {  switch((disp & 0xF0) >> 4) {  case 0 : firstreg = A; break;  case 1 : firstreg = X; break;  case 2 : firstreg = L; break;  case 3 : firstreg = B; break;  case 4 : firstreg = S; break;  case 5 : firstreg = T; break;  }  X++;  if(X == firstreg) comp = 0;  else if(X > firstreg) comp = 1;  else comp = -1;  }  else if(!strcmp(mnemonic, "TD")) comp = 1;  else if(!strcmp(mnemonic, "RD")) A &= 0xFFFF00;  else if(!strcmp(mnemonic, "WD")) ;  }  if(PC >= MEGABYTE - 1) { PC = endaddr; endflag = 1; break; }  if(!endflag) curraddr = PC;  else curraddr = PCtemp;  }  printf("A : %06X X : %06X\n", A, X);  printf("L : %06X PC: %06X\n", L, PC);  printf("B : %06X S : %06X\n", B, S);  printf("T : %06X\n", T);  if(meetbpflag) {  printf("Stop at checkpoint[%04X]\n", \*currbp);  \*currbp = \*nextbp;  meetbpflag = 0;  }  if(PC >= 0xFFFFF || endflag) {  printf("End the program.\n");  PC = execaddr;  L = endaddr;  \*currbp = \*nextbp = -1;  }  } |

## Makefile

|  |
| --- |
| OBJECTS = 20150959.o first\_project.o second\_project.o third\_project.o  SRC = 20150959.c first\_project.c second\_project.c third\_project.c  CC = gcc  CFLAGS = -g -Wall  TARGET = 20150959.out  all : $(TARGET)  $(TARGET) : $(OBJECTS) 20150959.h  $(CC) $(CFLAGS) -o $(TARGET) $(OBJECTS) -lm  clean :  rm -rf $(OBJECTS) $(TARGET)  $(OBJECTS) : $(SRC) 20150959.h |