과목 명: 시스템프로그래밍

담당 교수 명: 박 운 상

<<Assignment 2>>

서강대학교 컴퓨터학과

[학번] 20151623

[이름] 한상구

목 차

1. 프로그램 개요

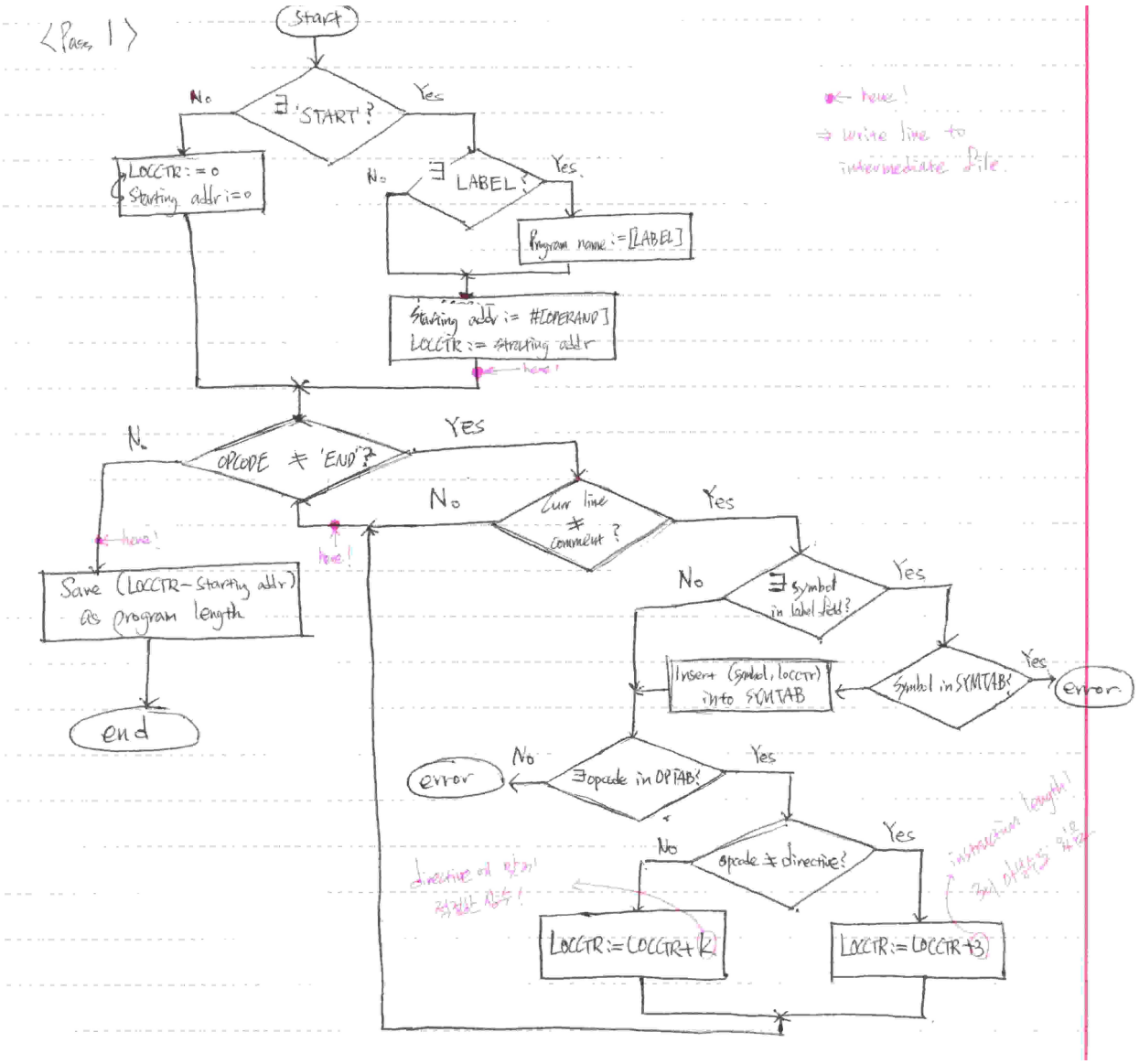
Assignment 1에서 구현한 SIC SIMULATOR를 토대로, 주어진 *.asm 파일을 assemble하는 assembler를 구현하였습니다. OPTAB과 SYMTAB을 hash-table을 이용하여 접근, 추가에 용이하도록 하였고, pass1과 pass2를 통해 Overflow, Invalid mnemonic, Invaild format등의 에러를 검출할 수 있도록 하였습니다.

성공적으로 assemble을 마치면 *.lst와 *.obj파일을 생성하며, 이들을 볼 수 있도록 type명령어를 구현하였습니다. 가장 최근에 assemble 되었던 *.asm파일의 SYMTAB을 출력하는 symbol명령어도 구현, 보다 상세한 파일 분석을 도왔습니다.

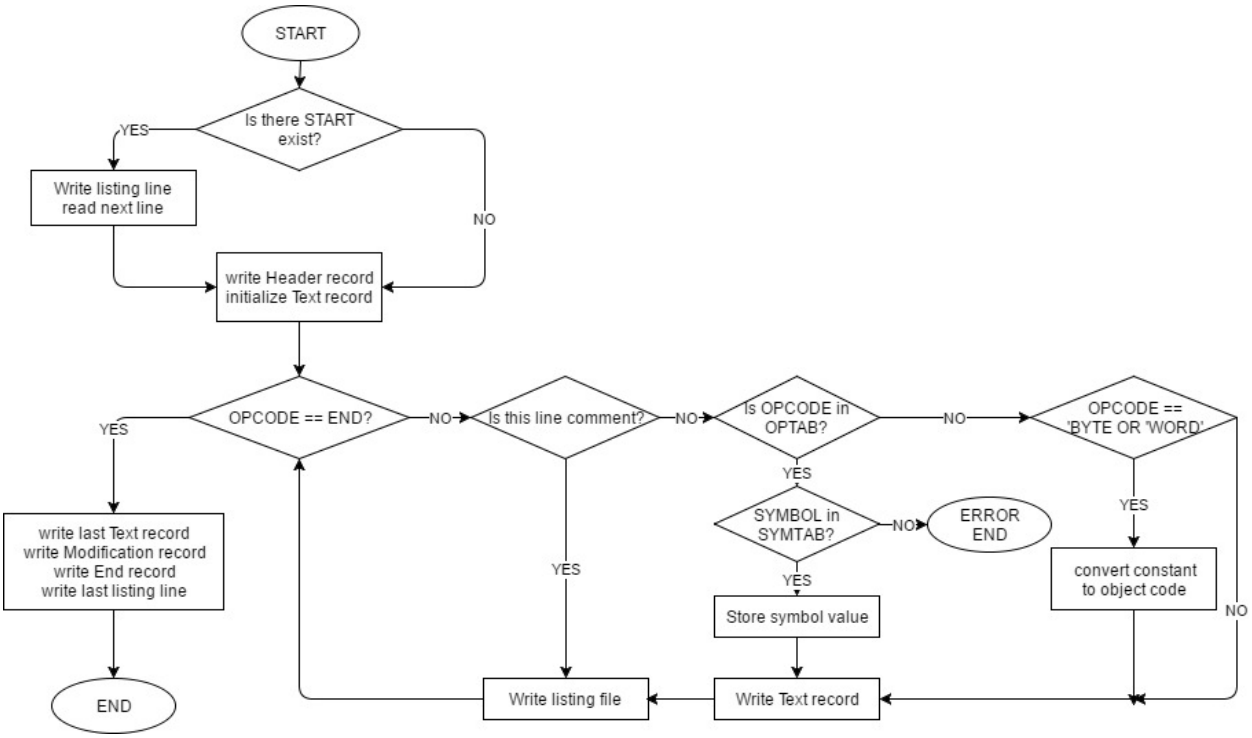
2. 프로그램 설명

2.1 프로그램 흐름도

2.1.1 Pass1 흐름도



2.1.2 Pass2 흐름도



3. 모듈 정의

3.1 모듈 이름 : cmp(const void *a, const void *b)

3.1.1 기능

commandSymbol 에서 qsort 를 사용할 때 사용되는 비교함수이다. Strcmp 값을 반환한다.

3.1.2 사용 변수

Const void *a, *b – qsort 시 비교를 위해 넘어오는 두 문자열의 주소를 가리킨다

3.2 모듈이름: commandAssemble(char *tok, char com[])

3.2.1 기능

입력 된 *.asm 파일을 assemble 해준다. 크게 pass1, pass2 로 구성되어있으며, 에러가 발생하지 않는다면 *.obj, *.lst 파일을 생성하고, 생성된 SYMTAB 을 보존한다.

3.2.2 사용변수

char com[] – Simulator 에 입력된 문장 전체를 담고 있는 문자열로, 본 함수에 인자로 들어온다.
char *tok – char type pointer 로 잘라낸 command 의 첫 글자를 가리킨다
char filename[] - tok 에서 파일 이름만을 떼어내어 저장하는 문자열.
int i – iterator. fileName 을 복사할 때 사용한다.
Int programLen – pass1 에 인자로 넘겨준 뒤, error 없이 pass1 이 끝난다면 프로그램의 길이를 저장하게 된다.

3.3 모듈이름: commandType(char *tok, char com[])

3.3.1 기능

주어진 파일이 현재 directory 내에 존재하면 그 파일의 내용을 화면상에 출력해준다.
cat 과 비슷하지만 1 개의 파일만을 출력한다는 것이 특징이다.

3.3.2 사용 변수

char *tok – char type pointer 로 잘라낸 command 의 첫 글자를 가리킨다
char com[] - Simulator 에 입력된 문장 전체를 담고 있는 문자열로, 본 함수에 인자로 들어온다.
FILE *fp – 주어진 파일에서 정보를 읽어오기 위한 file pointer.
int i, c– 각각 fileName 을 얻기 위해 사용하는 iterator, fp 로부터 한 글자씩 저장할 임시변수이다.
char fileName[] – 파일 이름만을 추출하여 저장하는 문자열

3.4 모듈 이름: commandSymbol(char *tok, char com[])

3.4.1 기능

SYMTAB 이 비어있지 않다면 (assemble 이 실행되지 않았거나 직전 assemble 이 error 를 뱉지 않았다면) 현재 저장되어 있는 SYMTAB 을 화면상에 출력한다. Symbol 은 내림차순으로 출력되며, 기준은 가장 먼저 등장하는 다른 character 의 ascii code 이다.

3.4.2 사용변수

char *tok – char type pointer 로 잘라낸 command 의 첫 글자를 가리킨다
char com[] - Simulator 에 입력된 문장 전체를 담고 있는 문자열로, 본 함수에 인자로 들어온다.
int symtanSize - SYMTAB 내에 존재하는 원소의 수를 저장하고 있다.
Symbol *it – SYMTAB 순회를 위한 포인터이다.
Symbol *arr – SYMTAB 내에 존재하는 모든 원소를 1 차원 배열 하나에 담기 위해 선언되었다. *(symtabSize * sizeof(Symbol))* 만큼 메모리를 동적으로 할당하며, 이후 qsort 를 통해 정렬된 원소를 갖는 배열이 된다.

3.5 모듈 이름: strToDecimal(char *val)

3.5.1 기능

val 이라는 character type array 를 decimal value 로 변환하여 반환한다. atoi()와 비슷한 역할이다. '-'가 붙어있을 시 음수처리도 해주며, 중간에 에러가 발생했을 시 inf(= 1<<30) 값을 반환한다.

3.5.2 사용변수

char *val – decimal 로 변환하려는 문자열.
int ret – return value 를 담기 위한 변수.

3.6 모듈 이름: symtabHashFunction(char *val)

3.6.1 기능

*.asm 에서 pass1 중 읽어온 symbol 을 bucket size 가 47 인 hash table 에 저장하기 위해 index 를 배정하는 hash function 이다. symbol 의 길이를 *l* 이라고 할 때, 주어진 symbol 을 *l* 자리의 19 진수로 간주한다. (이 때, 각 자리의 숫자는 '0'와의 거리로 한다. 예를 들어, '0' = 0, …, A = 10, B = 11, C =12, …, a = 36, …,)
이를 10 진수로 변환한 뒤, mod 47 의 int 형 값을 반환한다.
19 진수로 가정한 이유는, mnemonic 과 달리 symbol 은 길이가 다양하고 대문자, 소문자, 숫자가 들어올 수 있기 때문에 overflow 가 발생할 가능성이 크기 때문이다.

3.6.2 사용 변수

int ret – 주어진 19 진수를 10 진수로 변환할 때 값을 저장하는데 쓰인다. Return value.
char *val – parameter 로 넘어오는 문자열이다. symbol 을 저장하고 있으며 ret 에 담길 19 진수의 수를 담고있다.

3.7 모듈 이름: pass1(char fileName[], int *programLen)

3.7.1 기능

Pass1 algorithm 을 수행하는 함수이다. *.asm 파일에서 한 글자씩 읽어 중간파일은 만든다.
에러가 발생했을 시 '파일이름':'에러발생 라인':'에러발생 character': error: '에러메시지' 의 형식으로 출력하게 하여 어디서 어떠한 오류가 발생하였는지 명시하였다.
최대한 많은 오류를 pass1 에서 걸러내어 pass2 에서는 에러 처리에 보다 신경을 쓰지 않을 수 있도록 하였다. (operand 의 수, 범위, 형식, label 의 첫 글자는 숫자가 사용될 수 없는 것, directive 에 prefix 가 붙는 경우, START 가 중간에 오는 경우, END 뒤의 operand 가 first executable instruction 을 가리키지 않는 경우 등)
흐름도와 같은 흐름으로 작성하였으며, 흐름도에서 예외처리상황은 일부만 작성하였다.
에러가 발생하지 않는 경우, *.asm 파일에 존재하는 symbol 이 담긴 SYMTAB, "line number\tlabel\topcode\toperand" 의 형식으로 *.asm 을 변환한 intermediate 파일, "line\tLOCCTR\tPC" 의 형식으로 저장된 location 파일이 생성된다.
Return value 는 에러가 발생했을 시 -1, 아닌 경우 프로그램의 길이 (*programLen)이다

3.7.2 사용변수

Int *programLen – 정상적으로 종료 시 프로그램의 길이를 저장한다.
int locCtr – LOCCTR 을 저장한다.
int startAddr – start address 를 저장한다.
int insLen – locCtr 을 얼마나 증가시켜야 할 지를 저장한다.
int c – file 에서 한 글자씩 읽어오기 위해 사용하는 변수이다.
int cntLine, cntChar, cntGap – 각각 line, character, character 사이의 character 를 세어 저장하는 변수이다.
int i, idx – 인덱스, hash-table 의 인덱스를 저장한다.
int prefix, val – prefix, number 가 하나라도 존재할 시 이를 저장한다.
int constMode - constMode stores 1 if constant is string, 2 if constant is hexadecimal (in BYTE), stores 1 if "RESB", 3 if "RESW"
int neg – val 이 음수 값을 가질 때 -1 을, 아닌 경우 0 을 저장한다
int firstFlag, firstAddr - firstFlag stores (instruction line - 1), firstAddr stores first executable instruction's address
char filename[], label[33], opcode[33], operand[33] – 파일 이름을 저장하는 문자열, 각각 label, opcode, operand 를 저장하기 위한 character type array
FILE *asmPointer, *intPointer, *locPointer; - file pointer points aseembly file, intermediate file, location file each
bool errorFlag - when errorFlag stores 1 (TRUE), ksll the function
Instruction *opIt - iterate around OPTAB
Symbol *symPtr, *newSymbol – iterate around SYMTAB, allocating new node

3.8 모듈 이름: pass2(char fileName[], int programLen)

3.8.1 기능

Pass2 algorithm 을 수행하는 함수이다. Pass1() 이 에러 없이 수행 된 경우 실행되며, 실제로 object code 를 생성, object 파일 (*.obj)과 list 파일 (*.lst)를 생성하는 함수이다. 주어진 opcode 에 맞는 format 인지, 주어진 prefix 가 적절한지 등의 에러처리가 되어있다.

흐름도와 같이 작성하였으며, 예외처리 상황은 흐름도에 일부만 작성하였다.
에러가 발생하지 않는다면 *.obj, *.lst 가 생성되며 SYMTAB 이 유지되며, 1 을 반환한다.
에러가 발생한다면 SYMTAB 을 비우고, -1 을 반환한다.
Object code 를 생성하는 과정에서 많은 bitwise operation 을 사용했다.
1. set opcode

*opAddr = (opIt->opcode) << ((opIt->format[0] - '1') * 8 + (prefix & 4) * 2);*

format 1 일때에는 opcode 가 left 로 shift 될 필요가 없으며, format 2 일때에는 8bit, format 3 일때는 16bit, format 4 일 때는 24bit 만큼 shift 되어야 한다. OPTAB 내의 원소에서 format[]은 자신의 Foramt 을 문자열로 저장하고 있으므로 위와 같은 식을 세울 수 있었다.
마지막의 (prefix & 4) * 2 는 format 3, 4 를 구분하기 위함인데, format 4 는 prefix '+' 가 나타나야 하며
prefix 는 4 에 해당하는 bit 가 1 임을 알 수 있다. (1 – immediate, 2 – indirect, 3 – simple, 4 – extended) 곧 format 4 라면 8bit 만큼 추가로 left-shift 해주므로 올바르게 작동함을 알 수 있다.
2. set n, i, e bit

*opAddr |= (prefix & 3) << (16 + (prefix & 4) * 2); // set n, i bit*
opAddr |= (prefix & 4) << 18; // set e bit

prefix & 3 은 곧 n, i 의 상태만을 bit making 하는 것이고, 이들의 위치 또한 18,17 (26,25) bit 이므로 올바르게 작동함을 알 수 있다. (set e bit 에 해당하는 문장도 동일하다)
3. set x bit

*opAddr |= 1 << (15 + (prefix & 4) * 2);*

x bit 을 사용한다는 것은 format 3, 4 임을 나타내며, x 는 format 3 에서는 16 번째, format 4 에서는 24 번째 bit 를 사용한다. 곧 (prefix & 4) * 2 에 따라 x 가 결정되는데, 이는 format 3, 4 에 따라 결정되는 것이므로 올바르게 작동함을 알 수 있다.
4. set operand address

opAddr |= ((symPtr->locCtr - PC) & (0x1000 - 1)); // format 3, PC relative
opAddr |= ((symPtr->locCtr - B) & (0x1000 - 1)); // format 3, Base relative
opAddr |= ((symPtr->locCtr) & (0x100000 - 1)); // format 4

operand address 를 set 한다. Foramt3 는 12bit, format4 는 20bit 를 사용하므로 이에 대한 bit 만 살리는 bit masking 을 하였다. 항상 disp 혹은 address 는 lsb 쪽에 존재하므로 올바르게 작동함을 알 수 있다.

3.8.2 *사용변수*

int i, idx, locCtr, c - i for iterator, hash-index, LOCCTR, temporary variable
int intLine, locLine - line number from imediate, location file each
int prefix - store proper integer for prefix if any
int opAddr, constVal, cntObj, firstAddr - object code, constant values, count text record's length, first executable address
int modi[111], idxModi - stores location which needs to be relocated
bool errorFlag, idxFlag, resFlag - stores that error occured or not, flag that indicates this instruction is indexed addressing or not, flag that last recorded instruction is 'RESB' or 'RESW'
char fName[111], line[111], label[33], opcode[33], first[33], second[33], obj[77] - stores file name without file name extention, gets line from intermediate file, charater type array for store label, opcode, operands each
char *it - iterator
char regs[][3] – regsisters, 자신의 번호와 매칭되게 선언하였다.
FILE *intPointer, *locPointer, *objPointer, *lstPointer -ile pointers
reg24 B, PC - stores B, PC
Symbol *symPtr - SYMTAB iterator
Instruction *opIt - OPTAB iterator

4. 전역 변수 및 구조체, 매크로, typedef 정의

4.1 #define IS_COMMENT(x) (!((x)^'.'))

주어진 글자가 '.'인지 확인하는 매크로이다. 문장의 첫 글자에 대해 시행되며,
1(TRUE)를 반환할 시 본 문장은 주석임을 나타낸다..

4.2 #define IS_DELIMITER(x) ((x) == ' ' || (x) == ',' || (x) == '\r' || (x) == '\t' || (x) == '\n')

주어진 글자가 DELIMITER 인지 확인하는 매크로이다.
맞다면 1 을, 아니면 0 을 반환한다.

4.3 #define IS_ALPHABET(x) (((x)|32) >= 'a' && ((x)|32) <= 'z')

주어진 글자가 알파벳인지 확인하는 매크로이다.
'a' – 'A' = 32 이며, 이는 6 번째 bit 만의 차이이므로 이를 이용했다.
어떠한 character 에 32 를 bitwise OR 한 값이 ['a', 'z']에 속하지 않는다면, 이는 알파벳이 아니다.
알파벳이면 1, 아니면 0 을 반환한다.

4.4 #define IS_DECIMAL(x) ((x) >= '0' && (x) <= '9')

인자로 넘겨받은 x 가 가리키는 글자가 숫자인지 아닌지 확인하는 매크로이다.
숫자라면 1 을, 아니라면 0 을 반환한다.

4.5 #define IS_VALID_PREFIX(x) ((x) == '#' || (x) == '@' || (x) == '+')

주어진 x 가 유효한 prefix 인지 아닌지 확인하는 매크로이다.
유효한 prefix 라면 1 을, 아니라면 0 을 반환한다.

4.6 typedef int reg24

다른 일반 int 형 변수와 다르게 변수 선언부에 레지스터임을 명시하기 위해 선언했다.

4.7 typedef struct _Symbol

SYMTAB 을 구현하기 위해 hash table 구현이 필요했고, 이를 위해 선언한 structure 이다.
Elemnet 로는 symbol 을 저장하는 char type array 와 symbol 이 위치하는 location 을 담은 integer type variable,다음 node 를 연결할 link (struct _Instruction type pointer)가 있다.

4.8 Symbol *symbolHead[47]

SYMTAB 구현을 위해 hash-bucket 이 필요했고, 이를 위해 선언하였다. symbolHead 는 각 bucket 의 head 를 가리킨다. 프로그램 시작 시, 혹은 assemble 도중 error 가 발생할 시 NULL 로 초기화된다.

4.9 int symtabSize

SYMTAB 의 사이즈를 저장한다. SYMTAB 이 비어있는 것을 *O(1)*에 처리하기 위해 선언했다.

5. 코드

5.1 20151623.h

```
#ifndef _macros_h_
#define _macros_h_

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
```

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

#define FALSE 0
#define TRUE 1
#define IS_END_STRING(x) (*(x) == EOF || *(x) == '\0' || *(x) == '\n')
#define IS_INDENT(x) (*(x) == ' ' || *(x) == '\t')

#define FIRST_BYTE(x) ((x)&(0x0F00))
#define SECOND_BYTE(x) ((x)&(0X00F0))
#define THIRD_BYTE(x) ((x)&(0x000F))
#define IS_COMMENT(x) (!((x)^'.'))
#define IS_DELIMITER(x) ((x) == ' ' || (x) == ',' || (x) == '\r' || (x) == '\t' || (x) == '\n')
#define IS_ALPHABET(x) (((x)|32) >= 'a' && ((x)|32) <= 'z')
#define IS_DECIMAL(x) ((x) >= '0' && (x) <= '9')
#define IS_VALID_PREFIX(x) ((x) == '#' || (x) == '@' || (x) == '+')

typedef void (*comFuncPtr)(char *, char *);

typedef int reg24;

typedef struct _History {
	char command[111];
	struct _History *link;
} History;	// structure for history
History *historyHead, *last;	// empty linked list

typedef struct _Instruction {
	char mnemonic[7], format[4];
	int opcode;
	struct _Instruction *link;
} Instruction;  // structure for instructions
Instruction *instructionHead[20];	// empty hashtable

typedef struct _Symbol {
	char symbol[33];
	int locCtr;
	struct _Symbol *link;
} Symbol;	// structure for symbol
Symbol *symbolHead[47];

unsigned char virtualMem[1048576];	// for virtual memory, 1048576 = 16 ^ 5
int dumpLastAddr;	// last address of dump command
int symtabSize; // stores symbol table's size
bool quitFlag;  // when quitFlag stores 1 (TRUE), terminates this program

int cmp(const void *a, const void *b);
int hashFunction(char *val);
char* getCommand(char str[], int *commandNum);
int hexstrToInt(char *str);
void addHistory(char com[]);
void commandHelp(char *tok, char com[]);
```

```
void commandDir(char *tok, char com[]);
void commandQuit(char *tok, char com[]);
void commandHistory(char *tok, char com[]);
void commandDump(char *tok, char com[]);
void commandEdit(char *tok, char com[]);
void commandFill(char *tok, char com[]);
void commandReset(char *tok, char com[]);
void commandMnemonic(char *tok, char com[]);
void commandOplist(char *tok, char com[]);
void commandType(char *tok, char com[]);
void commandAssemble(char *tok, char com[]);
void commandSymbol(char *tok, char com[]);
void commandCat(char *tok, char com[]);
void commandCmp(char *tok, char com[]);
void commandCopy(char *tok, char com[]);
void commandTouch(char *tok, char com[]);
void commandHead(char *tok, char com[]);
void commandEcho(char *tok, char com[]);
void loadInstruction();
// define on commands.c
int strToDecimal(char *val);
int symtabHashFuction(char *val);
int pass1(char fileName[], int *programLen);
int pass2(char fileName[], int programLen);
// define on assemble.c

#endif
```

5.2 20151623.c

```
#include"20151623.h"

int main(){
	int commandNum = 0, i;  // to save the command
	char inputLine[111], *tok;	// inputLine for get input from user, tok for tokenizing
	History *curr;  // for deallocating the list
	Instruction *it;	// for deallocating the hashtable
	comFuncPtr comFunc[] = { commandHelp, commandDir,
		commandQuit, commandHistory, commandDump, commandEdit,
		commandFill, commandReset, commandMnemonic, commandOplist,
		commandCat, commandCmp, commandCopy, commandTouch,
		commandHead, commandEcho, commandType, commandAssemble,
		commandSymbol };
	// function pointer to reduce code cluster

	quitFlag = FALSE;
	dumpLastAddr = 0;
	historyHead = last = NULL;
	memset(virtualMem, 0, sizeof(virtualMem));
	for(i = 0; i < 20; i++) instructionHead[i] = NULL;
	for(i = 0; i < 47; i++) symbolHead[i] = NULL;
	//initialize part
```

```
	loadInstruction();
	// load instructions

	do{
		printf("sicsim>");	// shell
		fgets(inputLine, sizeof(inputLine), stdin);	// get input in inputLine
		inputLine[strlen(inputLine)-1] = '\0';  // set the last character of given line as NULL
		tok = getCommand(inputLine, &commandNum);	// tokenizing given line
		// now tok points first character of parameter, of end of string
		if(commandNum < 0) puts("Invaild command"); // if commandNum stores negative value, it means that
given command is invaild
		else comFunc[commandNum](tok, inputLine);	// else run function which mathces to given command
	}while(!quitFlag);

	while(historyHead){
		curr = historyHead;
		historyHead = historyHead->link;
		free(curr);
	}	// deallocating the list

	for(i = 0; i < 20; i++){
		while(instructionHead[i]){
			it = instructionHead[i];
			instructionHead[i] = instructionHead[i]->link;
			free(it);
		}
	}	// deallocating the table

	return 0;
}
```

5.3 commands.c

```
#include"20151623.h"

int cmp(const void *a, const void *b) { return -strcmp(((Symbol *)a)->symbol, ((Symbol *)b)->symbol); }

int hashFunction(char *val){
	int ret = 0;
	while(*val){
		ret *= 39;
		ret += *val - 'A';
		val++;
	}	// converts to hash
	//	let each character represents number which how far from 'A'
	//	and let that string is base39 digit
	return ret % 20;
}

char* getCommand(char str[], int *commandNum){
	char *com, tmp;
```

```
	while(!IS_END_STRING(str) && IS_INDENT(str)) str++;
	// when this loop ends, str points the first character which is not indent or somethin

	for(com = str; !IS_END_STRING(str) && !IS_INDENT(str); str++);
	tmp = *str, *str = '\0';
	// set the very first character after first word (command) as NULL, com now points only command string

	switch(*com){
		case 'a' : *commandNum = strcmp(com, "assemble") ? -1 : 17;
				break;
		case 'c' : *commandNum = strcmp(com, "cat") ? strcmp(com, "cmp") ? strcmp(com, "copy") ? -1 : 12 :
11 : 10;
				break;
		case 'd' : *commandNum = strcmp(com, "d") * strcmp(com, "dir") ? strcmp(com, "du") * strcmp(com,
"dump") ? -1 : 4 : 1;
				break;
		case 'e' : *commandNum = strcmp(com, "e") * strcmp(com, "edit") ?  strcmp(com, "echo") ? -1 : 15 : 5;
				break;
		case 'f' : *commandNum = strcmp(com, "f") * strcmp(com, "fill") ? -1 : 6;
				break;
		case 'h' : *commandNum = strcmp(com, "h") * strcmp(com, "help") ? strcmp(com, "hi") * strcmp(com,
"history") ? strcmp(com, "head") ? -1 : 14 : 3 : 0;
				break;
		case 'o' : *commandNum = strcmp(com, "opcode") ? strcmp(com, "opcodelist") ? -1 : 9 : 8;
				break;
		case 'q' : *commandNum = strcmp(com, "q") * strcmp(com, "quit") ? -1 : 2;
				break;
		case 'r' : *commandNum = strcmp(com, "reset") ? -1 : 7;
				break;
		case 's' : *commandNum = strcmp(com, "symbol") ? -1 : 18;
				break;
		case 't' : *commandNum = strcmp(com, "touch") ? strcmp(com, "type") ? -1 : 16 : 13;
				break;
		default : *commandNum = -1;
				break;
	}	// switch for given command is vaild or not

	*str = tmp;
	// restore given line

	while(!IS_END_STRING(str) && IS_INDENT(str)) str++;
	// when this loop ends, str points the first parameter or end of string

	return str;
}

int hexstrToInt(char *str){	// for convert hex string to deciaml integer
	int ret = 0, len = 0;	// integer to store the result, length of str
	bool errFlag = FALSE, negFlag = FALSE;  // flag to check whether given string is headecimal or not, negative
or not

	if(*str == '-') negFlag = TRUE, str++;
```

```
	while(!IS_END_STRING(str) && !IS_INDENT(str) && len < 6 && *str != ','){
		ret *= 16;
		if(*str < '0' || (*str > '9' && *str < 'A') || (*str > 'F' && *str < 'a') || *str > 'f'){
		  errFlag = TRUE;
		  break;
		}
		ret += *str <= '9' ? *str - '0' : *str <= 'Z' ? *str - 'A' + 10 : *str - 'a' + 10;
		str++; len++;
	}

	return len < 6 ? errFlag ? -1 : negFlag ? -3 : ret : -2;
	// -1 for invalid number, -2 for overflow ( over 0xFFFFF ), -3 for negative number
}

void addHistory(char com[]){
	History *newNode;
	newNode = (History *)malloc(sizeof(History));
	strcpy(newNode->command, com);
	newNode->link = NULL;   // set new node with given command

	if(!historyHead) historyHead = newNode, last = newNode;
	else last->link = newNode, last = newNode;	// add new node to list's last node
}

void commandHelp(char *tok, char com[]){
	if(!*tok){
		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\ncat [filename(s)]\ncmp
filename1 filename2\ncopy filename1 filename2\ntouch filename(s)\nhead lines filename\necho [message]\n");
// print the list
		addHistory(com);	// add command to list
	} else puts("Invalid command"); // if any character that is not indent followed by command, it's invalid
command
	return;
}

void commandDir(char *tok, char com[]){
	DIR *currDir;
	struct dirent *currFile;
	struct stat currStat;

	if(!*tok){
		if((currDir = opendir(".")){   // for read current directory. When opendir returns NULL, it means error
occured.
			while((currFile = readdir(currDir))){   // when readdir returns NULL, that means it reaches to the end
of directory
				stat(currFile->d_name, &currStat);
				printf("\t%s", currFile->d_name);
				if(S_ISDIR(currStat.st_mode)) putchar('/');	// when this item is directory
				else if(S_IXUSR & currStat.st_mode) putchar('*');	// when this item is executable
			}
			closedir(currDir);
```

```
            puts(""); // prints new line
        }
        addHistory(com);        // add command to list
    } else puts("Invalid command"); // if any character that is not indent followed by command, it's invalid
command
}

void commandQuit(char *tok, char com[]){
    if(!*tok) quitFlag = TRUE;
    else puts("Invalid command"); // if any character that is not indent followed by command, it's invalid command
    return;
}

void commandHistory(char *tok, char com[]){
    int i = 0;      // for history index
    History *curr;  // for loop

    if(!*tok){
        addHistory(com);        // add command to list
        curr = historyHead;
        while(curr){    // while the history remains
            printf("%-5d %s\n", ++i, curr->command);        // print the history
            curr = curr->link;      // jump to next node
        }
    } else puts("Invalid command"); // if any character that is not indent followed by command, it's invalid
command
    return;
}

void commandDump(char *tok, char com[]){
    int i, j, start, end; // loop variables, integer which stores start and end address each
    start = dumpLastAddr; end = 0;

    if(*tok){       // when parameters have entered after 'du' or 'dump'
        start = hexstrToInt(tok);

        if(start < 0 || *tok == ','){   // when error occurs
            if(start == -1 || *tok == ',') puts("Invalid start address has entered");
            else if(start == -2) puts("Please enter the address between 0x00000 through 0xFFFFF");
            else puts("Negative number has entered");
            return;
        }

        while(!IS_END_STRING(tok) && !IS_INDENT(tok) && *tok != ',') tok++;
        while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
        // now tok points the first character that is not indent right after first parameter, or end of string
        if(*tok == ','){        // when end exists
            tok++;  // for point character after ','
            while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
            // now tok points the first character of second parameter or end of string

            if(!*tok){      // when comma has entered but no parameter followed by
                puts("After comma need to enter end address");
```

```
                return;
            }

            end = hexstrToInt(tok);

            if(end < 0){    // when error occurs
                if(end == -1) puts("Invalid end address has entered");
                else if(end == -2) puts("Please enter the address between 0x00000 through 0xFFFFF");
                else puts("Negative number has entered");
                return;
            }

            if(start > end){        // when start is bigger than end
                puts("Start address value is bigger than end address value");
                return;
            }
        } else if(*tok){
            puts("Need to use comma to classify two addresses");    // when two parameters have entered without
comma
            return;
        } else dumpLastAddr = start;    // when only start exists
    }

    end = end ? end : start + 159 > 0xFFFFF ? 0xFFFFF : start + 159;        // set end address

    for(i = start / 16 * 16; i <= end / 16 * 16; i += 16){
        printf("%05X ", i);
        for(j = i; j < i + 16; j++){
            j >= start && j <= end ? printf("%02X ", virtualMem[j]) : printf("   ");
        }
        printf("; ");
        for(j = i; j < i + 16; j++){
            j >= start && j <= end && virtualMem[j] >= 0x20 && virtualMem[j] <= 0x7E ?
putchar(virtualMem[j]) : putchar('.');
        }
        puts("");
    }    // print

    dumpLastAddr = (end + 1) % (1<<20);

    addHistory(com);        // add command to list
}

void commandEdit(char *tok, char com[]){
    int addr, val;  // stores address, value each
    if(!*tok){      // when only 'e' or 'edit' were given
        puts("Parameters required : no parameters have entered");
        return;
    } else{
        addr = hexstrToInt(tok);

        if(addr < 0 || *tok == ','){    // when error occurs
            if(addr == -1 || *tok == ',') puts("Invalid address has entered");
```

```
            else if(addr == -2) puts("Please enter the address between 0x00000 through 0xFFFFF");
            else puts("Negative number has entered");
            return;
        }

        while(!IS_END_STRING(tok) && !IS_INDENT(tok) && *tok != ',') tok++;
        while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
        // now tok points the first character that is not indent right after first parameter, or end of string
        if(*tok == ','){        // when value exists
            tok++;  // for point character after ','
            while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
            // now tok points the first character of second parameter or end of string

            if(!*tok){      // when comma has entered but no parameter followed by
                puts("After comma need to enter value");
                return;
            }

            val = hexstrToInt(tok);

            if(val < 0 || val > 0xFF){      // when error occurs
                if(val == -1) puts("Invalid value has entered");
                else if(val == -2 || val > 0xFF) puts("Please enter the value between 0x00 through 0xFF");
                else puts("Negative number has entered");
                return;
            }
        } else if(*tok){
            puts("Need to use comma to classify two parameters");   // when two parameters have entered
without comma
            return;
        } else{ // when only address exists
            puts("Please enter the value");
            return;
        }
    }
    virtualMem[addr] = val;
    addHistory(com);        // add command to list
}

void commandFill(char *tok, char com[]){
    int i, start, end, val; // loop variable, stores start, end address and value each

    if(!*tok){      // when only 'f' or 'fill' were given
        puts("Parameters required : no parameters have entered");
        return;
    } else{
        start = hexstrToInt(tok);

        if(start < 0 || *tok == ','){   // when error occurs
            if(start == -1 || *tok == ',') puts("Invalid address has entered");
            else if(start == -2) puts("Please enter the address between 0x00000 through 0xFFFFF");
            else puts("Negative number has entered");
            return;
```

```
}

while(!IS_END_STRING(tok) && !IS_INDENT(tok) && *tok != ',') tok++;
while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
// now tok points the first character that is not indent right after first parameter, or end of string

if(*tok == ','){        // when end address exists
        tok++;  // for point character after ','
        while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
        // now tok points the first character of second parameter or end of string

        if(!*tok){      // when comma has entered but no parameter followed by
                puts("After comma need to enter end address");
                return;
        }

        end = hexstrToInt(tok);

        if(end < 0){    // when error occurs
                if(end == -1) puts("Invalid value has entered");
                else if(end == -2) puts("Please enter the value between 0x00 through 0xFF");
                else puts("Negative number has entered");
                return;
        }

        if(start > end){        // when start is bigger than end
                puts("Start address value is bigger than end address value");
                return;
        }

        while(!IS_END_STRING(tok) && !IS_INDENT(tok) && *tok != ',') tok++;
        while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
        // now tok points the first character that is not indent right after second parameter, or end of string

        if(*tok == ','){        // when value exists
                tok++;
                while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
                // now tok points the first character of third parameter or end of string

                if(!*tok){      // when comma has entered but no parameter followed by
                        puts("After comma need to enter value");
                        return;
                }

                val = hexstrToInt(tok);

                if(val < 0 || val > 0xFF){      // when error occurs
                        if(val == -1) puts("Invalid value has entered");
                        else if(val == -2 || val > 0xFF) puts("Please enter the value between 0x00 through 0xFF");
                        else puts("Negative number has entered");
                        return;
                }
```

```
                } else if(*tok){
                    puts("Need to use comma to classify end address and value");
                    return;
                } else{
                    puts("Please enter the value");
                    return;
                }

            } else if(*tok){
                puts("Need to use comma to classify three parameters"); // when two parameters have entered
without comma
                return;
            } else{ // when only start address exists
                puts("Please enter start address and value");
                return;
            }
        }
        for(i = start; i <= end; i++) virtualMem[i] = val;      // fill
        addHistory(com);        // add command to list
}

void commandReset(char *tok, char com[]){
        if(!*tok){
            memset(virtualMem, 0, sizeof(virtualMem));      // set every element in virtual memory 0
            addHistory(com);        // add command to list
        } else puts("Invalid command"); // if any character that is not indent followed by command, it's invalid
command
        return;
}

void commandMnemonic(char *tok, char com[]){
        int idx;        // stores index for given parameter
        char *it;       // iterator
        Instruction *curr;      // iterator
        if(!*tok) puts("No mnemonic has entered");
        else{
            for(it = tok; !IS_END_STRING(it) && !IS_INDENT(it); it++);
            *it = '\0';     // tokenize given mnemonic
            idx = hashFunction(tok);
            curr = instructionHead[idx];
            while(curr && strcmp(curr->mnemonic, tok)) curr = curr->link;
            // if given mnemonic exists in table, curr must point some node.
            if(!curr) puts("Invalid mnemonic");
            else{
                printf("opcode is %x\n", curr->opcode);
                addHistory(com);        // add command to list
            }
        }
}

void commandOplist(char *tok, char com[]){
        int i;  // loop variable
        Instruction *it;        // iterator
```

```
	if(!*tok){
		for(i = 0; i < 20; i+=puts("")){
			printf("%3d : ", i);
			it = instructionHead[i];
			while(it){
				printf("[%s,%x]", it->mnemonic, it->opcode);
				it = it->link;
				if(it) printf(" -> ");
			}
		}	// print the hash table
		addHistory(com);	// add command to list
	} else puts("Invalid command"); // if any character that is not indent followed by command, it's invalid
command
}

void commandType(char *tok, char com[]){
	FILE *fp;
	char fileName[111];
	int i, c;	// iterator, temporary character storage
	if(!*tok){
		puts("Filename required");
		return;
	} else{ // when file name has entered
		i = 0;
		while(!IS_END_STRING(tok) && !IS_INDENT(tok)) fileName[i++] = *tok++;
		fileName[i] = '\0';
		// copy file name

		while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
		if(*tok){
			puts("Too many arguments");
			return;
		}	// when more than one arguments exists

		if((fp = fopen(fileName, "r"))){	// when file exists
			while(~(c = fgetc(fp))) putchar(c);
			fclose(fp);
		} else{
		  printf("type: %s: no such file\n", fileName);
		  return;
		}
		// when file does not exists
	}
	addHistory(com);	// add command to list
}

void commandAssemble(char *tok, char com[]){
	char fileName[111];
	int i, programLen;
	if(!*tok){
		puts("Filename required");
		return;
	} else{
```

```
            i = 0;
            while(!IS_END_STRING(tok) && !IS_INDENT(tok)) fileName[i++] = *tok++;
            fileName[i] = '\0';
            //copy file name

            while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
            if(*tok){
                  puts("Too many arguments");
                  return;
            }      // when more than one arguments exists

            if((programLen = pass1(fileName, &programLen)) < 0 || pass2(fileName, programLen) < 0) return;
      }
      addHistory(com);
}

void commandSymbol(char *tok, char com[]){
      int i, idx;     // i for iterator, idx for array index
      Symbol *it, *arr;       // SYMTAB iterator, array
      if(*tok){       // when arguments exists
            puts("Command symbol does not need arguments");
            return;
      }
      if(!symtabSize){        // when SYMTAB is empty
            puts("SYMTAB has no symbol");
      } else{
            arr = (Symbol *)malloc(sizeof(Symbol) * symtabSize);    // array for sort symbols in lexicographical order
            for(i = idx = 0; i < 47; i++){
                  it = symbolHead[i];
                  while(it){      // copy from SYMTAB
                        strcpy(arr[idx].symbol, it->symbol);
                        arr[idx++].locCtr = it->locCtr;
                        it = it->link;
                  }
            }
            qsort(arr, symtabSize, sizeof(Symbol), cmp);    // sort
            for(i = 0; i < symtabSize; i++) printf("\t%s\t%X\n", arr[i].symbol, arr[i].locCtr);     // print
            free(arr);      // free
      }
      addHistory(com);        // add to history
}

void commandCat(char *tok, char com[]){
      FILE *fp;
      char fileName[111];
      int i, c, catCounter = 0;       // iterator, temporary character storage, counts successfully done file
      if(!*tok) while(~(c = getchar())) putchar(c);
      // when no file name has entered, cat command uses stdin
      else{   // when file name has entered
            do{
                  i = 0;
                  while(!IS_END_STRING(tok) && !IS_INDENT(tok)) fileName[i++] = *tok++;
                  fileName[i] = '\0';
```

```
                // copy file name
                if((fp = fopen(fileName, "r"))){        // when file exists
                    while(~(c = fgetc(fp))) putchar(c);
                    fclose(fp);
                    ++catCounter;
                } else printf("cat: %s: no such file\n", fileName);
                // when file does not exists
                while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
                // now tok points next file name's first character or end of string
            } while(*tok);
        }
        if(catCounter) addHistory(com); // if there is any of given file done successfully, add command to list
}

void commandCmp(char *tok, char com[]){
        FILE *sfp, *dfp;
        char sourceName[111], destName[111];
        int i = 0, line, s, d;  // iterator, stores line number, temporary storage
        bool diffFlag = 0;      // flag that shows whether two files are different or not
        if(!*tok){      // when no file name has entered
            puts("File name required");
            return;
        } else{
            while(!IS_END_STRING(tok) && !IS_INDENT(tok)) sourceName[i++] = *tok++;
            sourceName[i] = '\0';
            // copy first file name

            while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
            // now tok points second file name's first character of end of string

            if(!*tok){      // when there are no second file name
                puts("Second file name required");
                return;
            }

            i = 0;
            while(!IS_END_STRING(tok) && !IS_INDENT(tok)) destName[i++] = *tok++;
            destName[i] = '\0';
            // copy second file name

            if((sfp = fopen(sourceName, "r")) && (dfp = fopen(destName, "r"))){     // when both file exists
                s = d = 0;      // initiallize storage
                for(i = line = 1; ~s && ~d; i++){
                    s = fgetc(sfp); d = fgetc(dfp);
                    if(s ^ d){      // when diffrence have found
                        diffFlag = 1;
                        break;
                    }
                    if(s == '\n') ++line, i = 0;    // when line changes
                }
                if(diffFlag) printf("%s %s differ : byte %d, Line %d\n", sourceName, destName, i, line);
            } else{ // when some of files does not exist
                puts("Invalid file name");
```

```
                return;
            }
        }
        addHistory(com);        // add command to list
}

void commandCopy(char *tok, char com[]){
        FILE *sfp, *dfp;
        char sourceName[111], destName[111];
        int i = 0, c;   // iterator, temporary storage
        if(!*tok){      // when no file name has entered
            puts("File name required");
            return;
        } else{
            while(!IS_END_STRING(tok) && !IS_INDENT(tok)) sourceName[i++] = *tok++;
            sourceName[i] = '\0';
            // copy source file name

            while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
            // now tok points destiation file name's first character or end of string

            if(!*tok){      // when there are no destination file name
                puts("Destination file name required");
                return;
            }

            i = 0;
            while(!IS_END_STRING(tok) && !IS_INDENT(tok)) destName[i++] = *tok++;
            destName[i] = '\0';
            // copy destination file name

            if((sfp = fopen(sourceName, "r"))){     // when source file exists
                if(!(dfp = fopen(destName, "w"))){      // when failed to open destination file
                    puts("Failed to make / access file");
                    return;
                }
                while(~(c = fgetc(sfp))){
                  if(fputc(c, dfp) == EOF){
                      puts("File writing error occured");
                      return;
                  }
                }       // copy
                fclose(sfp); fclose(dfp);
            } else{ // when file does not exist
                printf("copy: %s: no such file\n", sourceName);
                return;
            }
        }
        addHistory(com);        // add command to list
}

void commandTouch(char *tok, char com[]){
        FILE *fp;
```

```
	char fileName[111];
	int i = 0;	// iterator
	if(!*tok){
		puts("File name required");
		return;
	} else{
		do{
			i = 0;
			while(!IS_END_STRING(tok) && !IS_INDENT(tok)) fileName[i++] = *tok++;
			fileName[i] = '\0';
			// copy given file name
			if((fp = fopen(fileName, "a"))){
				fclose(fp);
				// create 0-byte new file or touch already created file
			} else puts("File create / access error");
			while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
			// now tok points next file name's first character or end of string
		} while(*tok);
	}
	addHistory(com);
}

void commandHead(char *tok, char com[]){
	FILE *fp;
	char fileName[111];
	int line, i = 0, c;	// for store given line, iterator
	if(!*tok){	// when no parameters are entered
		puts("No parameters had entered");
		return;
	} else{
		line = hexstrToInt(tok);
		if(line < 0){	// given number is not hex
			puts("Please enter correct hexadecimal");
			return;
		}

		while(!IS_END_STRING(tok) && !IS_INDENT(tok)) tok++;
		while(!IS_END_STRING(tok) && IS_INDENT(tok)) tok++;
		// now tok points the first character of file name or end of string

		if(!*tok){	// when there are no more parameter left
			puts("Please enter file name");
			return;
		}

		while(!IS_END_STRING(tok) && !IS_INDENT(tok)) fileName[i++] = *tok++;
		fileName[i] = '\0';
		// copy file name

		if((fp = fopen(fileName, "r"))){
			while(~(c = fgetc(fp)) && line){
				putchar(c);
				if(c == '\n') --line;
```

```
                }
                fclose(fp);
            } else{
                printf("head: %s: no such file\n", fileName);
                return;
            }
        }
        addHistory(com);
}

void commandEcho(char *tok, char com[]){
        char *it;
        it = com;
        while(!IS_END_STRING(it) && IS_INDENT(it)) it++;
        while(!IS_END_STRING(it) && !IS_INDENT(it)) it++;
        // now it points right after the command echo

        if(*it) it++;
        printf("%s\n", it);
        addHistory(com);
}

void loadInstruction(){
        FILE *fp;       // for reading opcode.txt file
        char mnemonic[6], formats[4];   // for store mnemonic and given formats
        int opcode, idx;        // for store opcode, index
        Instruction *newNode, *it;      // for allocate new nodes, and iterator
        fp = fopen("opcode.txt", "r");
        while(~fscanf(fp, "%x %s %s", &opcode, mnemonic, formats)){     // until file pointer reaches EOF
            newNode = (Instruction *)malloc(sizeof(Instruction));
            newNode->opcode = opcode;
            strcpy(newNode->mnemonic, mnemonic);
            strcpy(newNode->format, formats);
            newNode->link = NULL;   // set new node with inputs from file
            idx = hashFunction(mnemonic);   // get hash through hash function
            if(instructionHead[idx]){
                it = instructionHead[idx];
                while(it->link) it = it->link;
                it->link = newNode;
            } else instructionHead[idx] = newNode;  // link to table
        }
        fclose(fp);
        return;
}
```

5.4 assemble.c

```
#include"20151623.h"

#define SET_ERRORFLAG errorFlag = TRUE; break;

int strToDecimal(char *val){
```

```
	int ret = 0, neg = 1;
	if(*val == '-'){
		neg = -1;
		val++;
	}
	while(*val){
		if(!IS_DECIMAL(*val)){
			ret = 1 << 30;
			neg = 1;
			break;
		}
		ret *= 10;
		ret += *val++ - '0';
	}
	return ret * neg;	// return 1 << 30 if given string is not decimal, or given decimal value if not
}

int symtabHashFuction(char *val){
	int ret = 0;
	while(*val){
		ret *= 19;
		ret += *val - (*val >= 'a' ? 'a'-36 : *val >= 'A' ? 'A'-10 : '0');
		ret = (ret + 47) % 47;
		val++;
	}	// converts to hash
	//	let each character represents number which how far from 'A' (let's define 'A'-'0' = 10, 'a'-'A' = 26)
	//	and let that string is base 19 digit, modulo 47
	return ret;
}

int pass1(char fileName[], int *programLen){
	int locCtr, startAddr, insLen, c, cntLine, cntChar, cntGap, i, idx, prefix, val, constMode, neg, firstFlag,
firstAddr;
	// for store LOCCTR, start address, bytes to increase LOCCTR, temporary variable
	// counter for line, characters, characters between strings, index, hash-index, prefix if any, value if any
	// constMode stores 1 if constant is string, 2 if constant is hexadecimal ( in BYTE ), stores 1 if "RESB", 3 if
"RESW"
	// neg stores 1 if val is nonnegative, -1 or not, firstFlag stores (instruction line - 1), firstAddr stores first
executable instruction's address
	char label[33], opcode[33], operand[33];
	FILE *asmPointer, *intPointer, *locPointer;	// file pointer points aseembly file, intermediate file, location
file each
	bool errorFlag; // when errorFlag stores 1 (TRUE), ksll the function
	Instruction *opIt;	// iterate around OPTAB
	Symbol *symPtr, *newSymbol;

	if(!(asmPointer = fopen(fileName, "r"))){	// when file does not exist
		printf("assemble: %s: no such file\n", fileName);
		return -1;	// return error value
	}
	if(!(intPointer = fopen("intermediate", "w"))){ // when failed to make intermediate file
		puts("assemble: Failed to create intermediate file");
		fclose(asmPointer);
```

```
        return -1;
    }
    if(!(locPointer = fopen("location", "w"))){     // when failed to make intermediate file
        puts("assemble: Faile to create location file");
        fclose(asmPointer); fclose(intPointer);
        return -1;
    }
    symtabSize = startAddr = locCtr = cntLine = cntChar = c = 0;
    errorFlag = FALSE; firstFlag = firstAddr = 0;
    for(i = 0; i < 47; i++){
        while(symbolHead[i]){
            symPtr = symbolHead[i];
            symbolHead[i] = symbolHead[i]->link;
            free(symPtr);
        }
    }
    // initialize part

    while(~c && ~(c = fgetc(asmPointer)) && !errorFlag){
        ++cntLine; cntChar = 1; neg = insLen = prefix = val = 0;
        *opcode = *label = *operand = '\0';

        fprintf(intPointer, "%4d\t", cntLine * 5);

        // check if this line is comment
        if(IS_COMMENT(c)){
            while(~c && !IS_END_STRING(&c)){
                fprintf(intPointer, "%c", c);
                c = fgetc(asmPointer);
            }
            fprintf(intPointer, "\n");
            continue;
        }     // if this line is comment, skip this line

        if(IS_DECIMAL(c)){      // when label's first character is number
            printf("%s:%d:%d: error: Label's first character cannot be number\n", fileName, cntLine, cntChar);
            SET_ERRORFLAG
        }

        // check whether label exists or not
        if(IS_ALPHABET(c)){     // when label exists
            i = cntGap = 0;
            while(~c && !IS_DELIMITER(c)){
                label[i++] = c;
                c = fgetc(asmPointer);
                ++cntGap;
                if(!IS_DELIMITER(c) && !IS_ALPHABET(c) && !IS_DECIMAL(c)){      // when there exists
non-alphabet and non-numeric character
                    printf("%s:%d:%d: error: Label cannot have character '%c' (non-alphabet, non-numeric)\n",
fileName, cntLine, cntChar+cntGap, c);
                    SET_ERRORFLAG
                }
            }
```

```
            label[i] = '\0';        // set NULL character at the end of string

            idx = symtabHashFuction(label);
            symPtr = symbolHead[idx];
            while(symPtr && strcmp(label, symPtr->symbol)) symPtr = symPtr->link;
            if(symPtr){
                printf("%s:%d:%d: error: Symbol '%s' has already used before\n", fileName, cntLine, cntChar,
label);
                SET_ERRORFLAG
            }       // if symbol already exists in SYMTAB

            cntChar += cntGap;
        }

        if(*label) fprintf(intPointer, "%s\t", label);
        else fprintf(intPointer, "!\t");

        while(!IS_END_STRING(&c) && IS_INDENT(&c)) c = fgetc(asmPointer), ++cntChar;    // skips indent
        if(!*label && IS_END_STRING(&c)) continue;      // when given line is blank

        firstFlag += 1;

        // check given format is valid
        if(!IS_ALPHABET(c) && !IS_VALID_PREFIX(c)){     // if something that is not alphabet shown before
opcode, it's error
            printf("%s:%d:%d: error: Unexpected character %c appeared\n", fileName, cntLine, cntChar, c);
            SET_ERRORFLAG
        }

        if(IS_VALID_PREFIX(c)){ // when prefix exists, save that prefix and let c store first character of opcode
            prefix = c == '+';
            if(!prefix){    // when '#', '@' comes before opcode
                printf("%s:%d:%d: error: Prefix '%c' cannot be used here\n", fileName, cntLine, cntChar, c);
                SET_ERRORFLAG
            }
            fprintf(intPointer, "%c", c);
            c = fgetc(asmPointer);
        }

        // check whether opcode is vaild or not
        i = 0; cntGap = prefix;
        while(IS_ALPHABET(c)){  // get given opcode
            opcode[i++] = c;
            c = fgetc(asmPointer);
            ++cntGap;
        }
        opcode[i] = '\0';       // set NULL character at the end of string

        while(!IS_END_STRING(&c) && IS_INDENT(&c)) c = fgetc(asmPointer), ++cntGap;     // skips indent

        fprintf(intPointer, "%s", opcode);

        if(!strcmp("START", opcode) || !strcmp("END", opcode) || !strcmp("BASE", opcode) || !strcmp("BYTE",
```

```
opcode) || !strcmp("WORD", opcode) || !strcmp("RESB", opcode) || !strcmp("RESW", opcode)){     // if opcode is
directive
                if(prefix){     // when directive has prefix
                    printf("%s:%d:%d: error: Directive do not support prefix '%c'\n", fileName, cntLine, cntChar,
'+');
                    SET_ERRORFLAG
                }
                --firstFlag;
                // do proper action for given directive
                if(!strcmp("START", opcode)){
                    if(firstFlag){  // when START appears after first instruction
                        printf("%s:%d:%d: error: 'START' directive must be used at very first instruction\n",
fileName, cntLine, cntChar);
                        SET_ERRORFLAG
                    }
                    if(IS_END_STRING(&c)){  // when no operands entered
                        printf("%s:%d:%d: error: No starting address\n", fileName, cntLine, cntChar+cntGap);
                        SET_ERRORFLAG
                    }
                    i = 0;
                    while(~c && !IS_DELIMITER(c)){
                        operand[i++] = c;
                        c = fgetc(asmPointer);
                    }
                    operand[i] = '\0';
                    startAddr = hexstrToInt(operand);
                    if(startAddr < 0){      // when given operand is not valid
                        printf("%s:%d:%d: error: ", fileName, cntLine, cntChar+cntGap);
                        puts(startAddr == -1 ? "Invalid value (not hexadecimal)" : startAddr == -2 ? "Operand has
too big value" : "Negative Address has entered");
                        SET_ERRORFLAG
                    }
                    locCtr = startAddr;     // set LOCCTR as startAddr
                    while(!IS_END_STRING(&c) && IS_INDENT(&c)) c = fgetc(asmPointer), ++cntGap;     //
skip indent
                    if(!IS_END_STRING(&c)){ // when there are any character after operand
                        printf("%s:%d:%d: error: Too many arguments\n", fileName, cntLine, cntChar+cntGap);
                        SET_ERRORFLAG
                    }
                    insLen = 0;     // START directive does not make LOCCTR increase
                    fprintf(intPointer, "\t%s", operand);
                }
                if(!strcmp("END", opcode)){
                    if(IS_ALPHABET(c)){
                        i = 0;
                        while(~c && !IS_DELIMITER(c)){
                            label[i++] = c;
                            c = fgetc(asmPointer);
                        }
                        label[i] = '\0';
                        while(!IS_END_STRING(&c) && IS_INDENT(&c)) c = fgetc(asmPointer), ++cntGap;//
skip indent
                        if(!IS_END_STRING(&c)){ // when there are any character after operand
```

```
                    printf("%s:%d:%d: error: Too many arguments\n", fileName, cntLine,
cntChar+cntGap);
                    SET_ERRORFLAG
                }
                idx = symtabHashFuction(label);
                symPtr = symbolHead[idx];
                while(symPtr && strcmp(label, symPtr->symbol)) symPtr = symPtr->link;
                if(!symPtr){    // when given symbol does not exist in SYMTAB
                    printf("%s:%d:%d: error: Symbol '%s' does not exist\n", fileName, cntLine,
cntChar+cntGap, label);
                    SET_ERRORFLAG
                }
                if(symPtr->locCtr != firstAddr){        // when given label is not point the first instruction
                    printf("%s:%d:%d: error: Label '%s' does not point first executable instruction \n",
fileName, cntLine, cntChar+cntGap, label);
                    SET_ERRORFLAG
                }
                fprintf(intPointer, "\t%s", label);
            } else if(!IS_END_STRING(&c)){  // when given label does not start with alphabet
                printf("%s:%d:%d: error: Label cannot start with '%c'\n", fileName, cntLine,
cntChar+cntGap, c);
                SET_ERRORFLAG
            }
            *programLen = locCtr - startAddr;
            *label = '\0';
        }
        if(!strcmp("BASE", opcode)){
            if(IS_END_STRING(&c)){  // when there is no operand
                printf("%s:%d:%d: error: Base directive needs operand\n", fileName, cntLine,
cntChar+cntGap);
                SET_ERRORFLAG
            }
            if(!IS_ALPHABET(c)){    // when given label does not start with alphabet
                printf("%s:%d:%d: error: Label cannot start with '%c'\n", fileName, cntLine,
cntChar+cntGap, c);
                SET_ERRORFLAG
            }

            fprintf(intPointer, "\t");
            i = 0;
            while(~c && !IS_DELIMITER(c)){  // get operand
                fprintf(intPointer, "%c", c);
                c = fgetc(asmPointer);
                ++cntGap;
            }

            while(~c && IS_INDENT(&c)) c = fgetc(asmPointer), ++cntGap;     // skips indent
            if(!IS_END_STRING(&c)){ // when there is any character after operand
                printf("%s:%d:%d: error: Unexpected character '%c' appreared\n", fileName, cntLine,
cntChar+cntGap, c);
                SET_ERRORFLAG
            }
        }
```

```
		if(!strcmp("BYTE", opcode)){
			if(c != 'C' && c != 'X'){	// when constant is not valid
				printf("%s:%d:%d: error: Invalid constant '%c' used\n", fileName, cntLine,
cntChar+cntGap, c);
				SET_ERRORFLAG
			}
			fprintf(intPointer, "\t%c", c);
			constMode = c == 'C' ? 1 : 2;
			if((c = fgetc(asmPointer)) != '\''){	// when format is invalid
				printf("%s:%d:%d: error: Invalid format for generating constant\n", fileName, cntLine,
cntChar+++cntGap);
				SET_ERRORFLAG
			}
			fprintf(intPointer, "%c", c);
			c = fgetc(asmPointer); cntGap += 2; i = 0;
			while(!IS_END_STRING(&c) && c != '\''){
				fprintf(intPointer, "%c", c);
				if(constMode & 2 && !IS_DECIMAL(c) && (c|32) < 'a' && (c|32) > 'f'){
					printf("%s:%d:%d: error: '%c' is not hexadecimal value\n", fileName, cntLine,
cntChar+cntGap, c);
					SET_ERRORFLAG
				}	// when constant is hexadecimal but non-hexadeciaml value has given
				++i; ++cntGap;
				c = fgetc(asmPointer);
			}
			if(c != '\''){  // when string ends before ''' appears
				printf("%s:%d:%d: error: Invalid match for '''\n", fileName, cntLine, cntChar+cntGap);
				SET_ERRORFLAG
			}
			fprintf(intPointer, "%c", c);
			insLen = constMode & 2 ? (i+1)/constMode : i;
		}
		if(!strcmp("WORD", opcode)){
			insLen = 3;	// word == 3bytes ( in SIC/XE )
			neg = 1;
			if(c == '-') neg = -1, c = fgetc(asmPointer);	// when negative value has entered
			while(!IS_END_STRING(&c) && IS_DECIMAL(c) && val < 1<<24){	// get operand
				val *= 10;
				val += c - '0';
				c = fgetc(asmPointer);
				++cntGap;
			}
			if(~c && !IS_DELIMITER(c)){	// when there exists non-decimal character in operand
				printf("%s:%d:%d: error: '%c' is not decimal\n", fileName, cntLine, cntChar+cntGap, c);
				SET_ERRORFLAG
			}
			val *= neg;	// give value proper domain
			if(val > (1<<23)-1 || val < -(1<<23)){  // when overflow occurs
				while(val) val /= 10, --cntGap;
				printf("%s:%d:%d: error: Operand is too big to store in a word\n", fileName, cntLine,
cntChar+cntGap);
				SET_ERRORFLAG
			}
```

```
                fprintf(intPointer, "\t%d", val);
            }
            if(!strcmp("RESB", opcode) || !strcmp("RESW", opcode)){
                constMode = strcmp("RESB", opcode) ? 3 : 1;     // stores 1 when opcode is "RESB", and stores
3 when opcode is "RESW"
                while(!IS_END_STRING(&c) && IS_DECIMAL(c) && val < 1<<20){      // get operand
                    val *= 10;
                    val += c - '0';
                    c = fgetc(asmPointer);
                    ++cntGap;
                }
                if(val >= 1<<20){       // when overflow occurs
                    while(val) val /= 10, --cntGap;
                    printf("%s:%d:%d: error: Operand is too big\n", fileName, cntLine, cntChar+cntGap);
                    SET_ERRORFLAG
                }
                if(~c && !IS_DELIMITER(c)){     // when there exists non-decimal character in operand
                    printf("%s:%d:%d: error: '%c' is not decimal\n", fileName, cntLine, cntChar+cntGap, c);
                    SET_ERRORFLAG
                }
                fprintf(intPointer, "\t%d", val);
                insLen = constMode * val;
            }
        } else{ // if opcode is not directive
            idx = hashFunction(opcode);
            opIt = instructionHead[idx];
            while(opIt && strcmp(opIt->mnemonic, opcode)) opIt = opIt->link;
            if(!opIt){      // when given opcode does not exist in OPTAB
                printf("%s:%d:%d: error: %s is invalid opcode\n", fileName, cntLine, cntChar+prefix, opcode);
                SET_ERRORFLAG
            }
            insLen = opIt->format[0] - '0';
            if(insLen < 3 && prefix){       // when there exist prefix where it should not
                printf("%s:%d:%d: error: Inapproapriate prefix '%c' exists\n", fileName, cntLine, cntChar, '+');
                SET_ERRORFLAG
            } else insLen += prefix;

            prefix = 0;
            if(IS_VALID_PREFIX(c)){ // when '#' or '@' exists
                if(c == '+'){   // when there exist prefix where it should not
                    printf("%s:%d:%d: error: Inapproapriate prefix '%c' exists\n", fileName, cntLine, cntChar,
'+');
                    SET_ERRORFLAG
                }
                prefix = c;
                c = fgetc(asmPointer);
                ++cntGap;
            }

            i = 0;
            while(!IS_END_STRING(&c) && !IS_DELIMITER(c)){
                if(!IS_ALPHABET(c) && !IS_DECIMAL(c)){  // when non-alphabet and non-decimal
character exists
```

```
                    printf("%s:%d:%d: error: '%c' is not allowed for operand\n", fileName, cntLine,
cntChar+cntGap, c);
                    SET_ERRORFLAG
                }
                operand[i++] = c;
                c = fgetc(asmPointer);
                ++cntGap;
            }
            operand[i] = '\0';

            if(prefix && !*operand){        // when only prefix exists
                printf("%s:%d:%d: error: Operand should be followed after prefix\n", fileName, cntLine,
cntChar+cntGap);
                SET_ERRORFLAG
            }

            fprintf(intPointer, "\t");
            if(prefix) fprintf(intPointer, "%c", prefix);
            fprintf(intPointer, "%s", operand);

            while(!IS_END_STRING(&c) && IS_INDENT(&c)) c = fgetc(asmPointer), ++cntGap;     // skip
indent

            if(!IS_END_STRING(&c)){
                if(c != ','){   // when other operand exists but no delimiter exists
                    printf("%s:%d:%d: error: operands should be classified by ','\n", fileName, cntLine,
cntChar+cntGap);
                    SET_ERRORFLAG
                }

                while(!IS_END_STRING(&c) && IS_DELIMITER(c)) c = fgetc(asmPointer), ++cntGap;   //
let c store first character of operand

                i = 0;
                while(!IS_END_STRING(&c) && !IS_DELIMITER(c)){
                    if(!IS_ALPHABET(c) && IS_DECIMAL(c)){   // when non-alphabet and non-decimal
character exists
                        printf("%s:%d:%d: error: '%c' is not allowed for operand\n", fileName, cntLine,
cntChar+cntGap, c);
                        SET_ERRORFLAG
                    }
                    operand[i++] = c;
                    c = fgetc(asmPointer);
                    ++cntGap;
                }
                operand[i] = '\0';

                fprintf(intPointer, "\t%s", operand);
            }
        }
        while(!IS_END_STRING(&c)) c = fgetc(asmPointer), ++cntGap;      // skip the last part of this line ( this
part will be handled by pass2 algorithm )
        cntChar += cntGap;
```

```
            fprintf(intPointer, "\n");

            // if all instruction is fine and symbol exists, put symbol into SYMTAB
            if(*label){
                // get hash-index
                idx = symtabHashFuction(label);

                // increase symtabSize
                ++symtabSize;

                //      allocate new node for SYMTAB
                newSymbol = (Symbol *)malloc(sizeof(Symbol));
                newSymbol->locCtr = locCtr;
                strcpy(newSymbol->symbol, label);
                newSymbol->link = NULL;

                // insert symbol in SYMTAB
                if(!symbolHead[idx]) symbolHead[idx] = newSymbol;
                else{
                    symPtr = symbolHead[idx];
                    while(symPtr->link) symPtr = symPtr->link;
                    symPtr->link = newSymbol;
                }
            }
            if(!firstAddr && firstFlag == 1) firstAddr = locCtr;
            fprintf(locPointer, "%4d\t%X\t", cntLine * 5, locCtr);  // save LOCCTR for current line
            locCtr += insLen;       // increase LOCCTR properly
            fprintf(locPointer, "%X\n", locCtr);    // save PC for current line
        }

        if(errorFlag){  // if any error occurs, SYMTAB need to be cleared
            for(i = 0; i < 47; i++){
                while(symbolHead[i]){
                    symPtr = symbolHead[i];
                    symbolHead[i] = symbolHead[i]->link;
                    --symtabSize;
                    free(symPtr);
                }
            }
            symtabSize = 0;
        }

        fclose(asmPointer), fclose(intPointer), fclose(locPointer);
        return errorFlag ? -1 : *programLen;
}

int pass2(char fileName[], int programLen){
        int i, idx, locCtr, c, intLine, locLine, prefix, opAddr, constVal, cntObj, firstAddr;
        // i for iterator, hash-index, LOCCTR, temporary variable, line number from imediate, location file each, store
proper integer for prefix if any, object code, constant values, count text record's length, first executable address
        int modi[111], idxModi; // stores location which needs to be relocated
        bool errorFlag, idxFlag, resFlag;       // stores that error occured or not, flag that indicates this instruction is
indexed addressing or not, flag that last recorded instruction is 'RESB' or 'RESW'
```

```
	char fName[111], line[111], label[33], opcode[33], first[33], second[33], obj[77], *it;
	// stores file name without file name extention, gets line from intermediate file, charater type array for store
label, opcode, operands each
	// it for iterator
	char regs[][3] = {"A", "X", "L", "B", "S", "T", "F", "", "PC", "SW"};   // regsisters
	FILE *intPointer, *locPointer, *objPointer, *lstPointer;        // file pointers
	reg24 B, PC;    // stores B, PC
	Symbol *symPtr; // SYMTAB iterator
	Instruction *opIt;      // OPTAB iterator

	if(!(intPointer = fopen("intermediate", "r"))){
		puts("Cannot find intermediate file");
		return -1;
	}
	if(!(locPointer = fopen("location", "r"))){
		puts("Cannot find location file");
		fclose(intPointer);
		return -1;
	}
	// open intermediate, location file

	strcpy(fName, fileName);
	for(i = 0; fName[i] != '.'; i++);
	fName[i] = '\0';
	// get file name without extention from file name

	if(!(objPointer = fopen("tmpobj", "w"))){
		puts("File open error");
		fclose(intPointer), fclose(locPointer);
		return -1;
	}
	if(!(lstPointer = fopen("tmplst", "w"))){
		puts("File open error");
		fclose(intPointer), fclose(locPointer), fclose(objPointer);
		return -1;
	}
	// open temporary list, object file

	errorFlag = resFlag = FALSE; cntObj = B = idxModi = 0; locLine = -1;

	while(~fscanf(intPointer, "%d\t", &intLine)){
		opAddr = prefix = 0;
		idxFlag = FALSE;
		*label = '\0';

		fscanf(intPointer, "%s", label);
		fgets(line, 100, intPointer);
		// read from intermediate file
		if(locLine < intLine) fscanf(locPointer, "%d%X%X", &locLine, &locCtr, &PC);
		// if it is excutable line, read LOCCTR and PC from location file

		if(!cntObj){
			*obj = 'T';
```

```
            sprintf(obj+1, "%06X", locCtr); // write starting address for object code in this record
            cntObj = 9;
            resFlag = FALSE;
        }

        fprintf(lstPointer, "%4d\t", intLine);
        if(*label == '.'){      // when this line is comment
            fprintf(lstPointer, "\t");
            if(!IS_END_STRING(&line[1])) fprintf(lstPointer, "  %s", line);
            else fprintf(lstPointer, "\n");
            continue;
        }

        it = line + 1; i = 0;
        if(IS_VALID_PREFIX(*it)){       // when prefix exists
            prefix |= *it == '+' ? 4 : 0;
            it++;
        }
        while(!IS_END_STRING(it) && !IS_INDENT(it)) opcode[i++] = *it++;
        opcode[i] = '\0'; it++;
        // get opcode from imediate  file
        //sarangsarangS2

        idx = hashFunction(opcode);
        opIt = instructionHead[idx];
        while(opIt && strcmp(opIt->mnemonic, opcode)) opIt = opIt->link;
        // find opcode in OPTAB, if it does not exist in OPTAB, it means opcode stores directive

        if(opIt){       // when opcode stores opcode
            // get symbol and fine in SYMTAB
            fprintf(lstPointer, "%04X\t", locCtr);
            if(*label == '!') fprintf(lstPointer, "\t");
            else fprintf(lstPointer, "%s\t", label);
            if(prefix & 4) fprintf(lstPointer, "+");
            fprintf(lstPointer, "%s\t", opcode);

            opAddr = (opIt->opcode) << ((opIt->format[0] - '1') * 8 + (prefix & 4) * 2);    // set opcode

            if(opIt->format[0] == '3'){     // when this instruction froms format 3
                if(IS_VALID_PREFIX(*it)) prefix |= (*it++ == '#') ? 1 : 2;
                else prefix |= 3;       // 1 for immediate, 2 for indirect, 3 for simple ( n, i bit as lsb )
                opAddr |= (prefix & 3) << (16 + (prefix & 4) * 2);      // set n, i bit
                opAddr |= (prefix & 4) << 18;   // set e bit
            } else{ // when format 1 or format 2 have prefix on operand
                if(IS_VALID_PREFIX(*it)){
                    printf("%s:%d: error: Format %c should not have prefix\n", fileName, intLine, opIt-
>format[0]);
                    SET_ERRORFLAG
                }
            }

            if(opIt->format[0] == '1'){
                if(!IS_END_STRING(it)){ // when this instruction is format 1 but operand exists
```

```
                    printf("%s:%d: error: Opcode '%s' does not need operand(s)\n", fileName, intLine, opIt-
>mnemonic);
                    SET_ERRORFLAG
                }
            } else{
                for(i = 0; !IS_END_STRING(it) && !IS_INDENT(it); first[i++] = *it++);
                first[i] = '\0';
                if(!IS_END_STRING(it)) it++;
                // get first operand if any

                idx = symtabHashFuction(first);
                symPtr = symbolHead[idx];
                while(symPtr && strcmp(symPtr->symbol, first)) symPtr = symPtr->link;
                if(*first && !symPtr){  // when given symbol does not exist in SYMTAB
                    for(i = 0; i < 10 && strcmp(first, regs[i]); i++);      // if first stores register, i must be less
than 10
                    constVal = strToDecimal(first);
                    if((prefix & 3) == 1 && constVal != (1<<30)){   // for immediate addressing ( '#' +
decimal )
                        if(constVal >= 1 << 20){        // when overflow occurs
                            printf("%s:%d: error: Operand has too big value\n", fileName, intLine);
                            SET_ERRORFLAG
                        } else if(constVal >= 1<<12){   // when instruction needs format 4
                            if(prefix & 4) opAddr |= constVal;
                            else{   // when prefix '+' is not uesd
                                printf("%s:%d: error: Prefix '+' need to be used\n", fileName, intLine);
                                SET_ERRORFLAG
                            }
                        } else if(constVal >= 0) opAddr |= constVal;
                        else{   // when negative constant has entered
                            printf("%s:%d: error: Negative decimal\n", fileName, intLine);
                            SET_ERRORFLAG
                        }
                        if(!IS_END_STRING(it)){ // when there exists any operand after ('#' + deciaml)
                            printf("%s:%d: error: Too many arguments\n", fileName, intLine);
                            SET_ERRORFLAG
                        }
                    } else if(i > 9){       // for undefined symbol
                        printf("%s:%d: error: Undefined symbol '%s' is used\n", fileName, intLine, first);
                        SET_ERRORFLAG
                    }
                }

                for(i = 0; !IS_END_STRING(it) && !IS_INDENT(it); second[i++] = *it++);
                second[i] = '\0'; it++;
                // get second operand if any

                if(opIt->format[0] == '2'){
                    if(!*first){    // when no operands has entered
                        printf("%s:%d: error: No operands\n", fileName, intLine);
                        SET_ERRORFLAG
                    }
                    for(i = 0; i < 10 && strcmp(regs[i], first); i++);
```

```
                    if(i > 9){      // when invalid register has entered
                      printf("%s:%d: error: '%s' is invalid register\n", fileName, intLine, first);
                      SET_ERRORFLAG
                    }
                    opAddr |= i << 4;       // set r1
                    fprintf(lstPointer, "%s", first);
                    if(*second){    // when second register has entered
                        for(i = 0; i < 10 && strcmp(regs[i], second); i++);
                        if(i > 9){      // when invalid register has entered
                            printf("%s:%d: error: '%s' is invalid register\n", fileName, intLine, second);
                            SET_ERRORFLAG
                        }
                        opAddr |= i;    // set r1
                        fprintf(lstPointer, ",%s\t\t", second);
                    } else fprintf(lstPointer, "\t\t");
                } else{
                    if(*first){
                        if((prefix & 3) < 3) fprintf(lstPointer, "%c", (prefix & 3) == 2 ? '@' : '#');
                        fprintf(lstPointer, "%s", first);
                        if(*second){
                            if(strcmp("X", second)){        // when second operand exist that is not 'X'
                                printf("%s:%d: error: Too many arguments\n", fileName, intLine);
                                SET_ERRORFLAG
                            } else opAddr |= 1 << (15 + (prefix & 4) * 2);  // set x bit
                            fprintf(lstPointer, ",%s\t", second);
                        } else fprintf(lstPointer, "\t\t");
                        if(symPtr){
                            if(prefix & 4){
                                opAddr |= ((symPtr->locCtr) & (0x100000 - 1));  // set operand address
                                if((prefix & 3) != 1) modi[idxModi++] = locCtr + 1; // when this instruction
needs to be relocated
                            }
                            else{
                                if(symPtr->locCtr - PC < 2048 && symPtr->locCtr - PC > -2049){     // PC
relative
                                    opAddr |= ((symPtr->locCtr - PC) & (0x1000 - 1));  // set operand
address
                                    opAddr |= 1 << 13;      // set p bit
                                } else if(symPtr->locCtr - B < 4096 && symPtr->locCtr - B > -1){   // Base
relative
                                    opAddr |= ((symPtr->locCtr - B) & (0x1000 - 1));   // set operand
address
                                    opAddr |= 1 << 14;      // set b bit
                                } else{ // overflow occured
                                    printf("%s:%d: error: Prefix '+' need to be used\n", fileName, intLine);
                                    SET_ERRORFLAG
                                }
                            }
                        }
                    } else if((prefix & 3) < 3){
                        printf("%s:%d: error: No operands\n", fileName, intLine);
                        SET_ERRORFLAG
                    } else fprintf(lstPointer, "\t\t");
```

```
                }
            }
        } else{ // when opcode stores directive
            if(!strcmp("BYTE", opcode) || !strcmp("WORD", opcode) || !strcmp("RESB", opcode)
|| !strcmp("RESW", opcode)) fprintf(lstPointer, "%04X", locCtr);
            fprintf(lstPointer, "\t");
            if(*label == '!') fprintf(lstPointer, "\t");
            else fprintf(lstPointer, "%s\t", label);
            fprintf(lstPointer, "%s\t", opcode);

            if(!strcmp("START", opcode)){
                fprintf(lstPointer, "%X", locCtr);
                fprintf(objPointer, "H");
                firstAddr = locCtr;
                if(*label) fprintf(objPointer, "%-6s", label);
                else fprintf(objPointer, "      ");
                fprintf(objPointer, "%06X%06X\n", locCtr, programLen);
                cntObj = 0;
            }
            if(!strcmp("END", opcode)){
                while(!IS_END_STRING(it)) fprintf(lstPointer, "%c", *it++);
                fprintf(objPointer, "%s%02X%s\n", obj, (cntObj - 9) / 2, obj + 9);
                for(i = 0; i < idxModi; i++) fprintf(objPointer, "M%06X05\n", modi[i]);
                fprintf(objPointer, "E%06X\n", firstAddr);
            }
            if(!strcmp("BASE", opcode)){
                i = 0;
                while(!IS_END_STRING(it)) first[i++] = *it++;
                first[i] = '\0';
                // get operand

                idx = symtabHashFuction(first);
                symPtr = symbolHead[idx];
                while(symPtr && strcmp(symPtr->symbol, first)) symPtr = symPtr->link;
                if(!symPtr){    // when symbol does not exist in SYMTAB
                    printf("%s:%d: error: Undefined symbol '%s' is used\n", fileName, intLine, first);
                    SET_ERRORFLAG
                }
                fprintf(lstPointer, "%s", first);
                B = symPtr->locCtr;     // set B register
            }
            if(!strcmp("BYTE", opcode)){
                i = 0;
                while(!IS_END_STRING(it)) first[i++] = *it++;
                first[i] = '\0';
                fprintf(lstPointer, "%s\t\t", first);

                for(i = 2; first[i] != '\''; i++) *first == 'X' ? fprintf(lstPointer, "%c", first[i]) : fprintf(lstPointer,
"%02X", first[i]);
                first[i--] = '\0';

                if(*first == 'C'){
                    if(cntObj + (i - 1) * 2 > 68 || resFlag){
```

```
                    fprintf(objPointer, "%s%02X%s\n", obj, (cntObj - 9) / 2, obj + 9);
                    sprintf(obj+1, "%06X", locCtr);
                    for(i = 2, cntObj = 9; first[i]; i++, cntObj+=2) sprintf(obj+cntObj, "%02X", first[i]);
                } else{
                    for(i = 2; first[i]; i++) sprintf(obj + cntObj + (i-2) * 2, "%02X", first[i]);
                    cntObj += (i-2) * 2;
                }
            } else{
                if(cntObj + i - 1 > 68|| resFlag){
                    fprintf(objPointer, "%s%02X%s\n", obj, (cntObj - 9) / 2, obj + 9);
                    sprintf(obj+1, "%06X", locCtr);
                    sprintf(obj+9, "%s", first);
                    cntObj = 9 + (i - 1);
                } else sprintf(obj + cntObj, "%s", first + 2), cntObj += i-1;
            }
            resFlag = FALSE;
        }
        if(!strcmp("WORD", opcode)){
            i = 0;
            while(!IS_END_STRING(it)) first[i++] = *it++;
            first[i] = '\0';
            constVal = strToDecimal(first) & 0xFFFFFF;
            fprintf(lstPointer, "%s\t%d", first, constVal);

            if(cntObj + 6 > 68 || resFlag){
                fprintf(objPointer, "%s%02X%s\n", obj, (cntObj - 9) / 2, obj + 9);
                sprintf(obj+1, "%06X", locCtr);
                sprintf(obj+9, "%06X", constVal);
                cntObj = 15;
            } else sprintf(obj + cntObj, "%06X", constVal), cntObj += 6;
            resFlag = FALSE;
        }
        if(!strcmp("RESB", opcode) || !strcmp("RESW", opcode)){
            while(!IS_END_STRING(it)) fprintf(lstPointer, "%c", *it++);
            resFlag = TRUE;
        }
        fprintf(lstPointer, "\n");
        continue;
    }

    if(opIt->format[0] == '1'){
        fprintf(lstPointer, "%02X\n", opAddr);
        if(cntObj + 2 > 68 || resFlag){
            fprintf(objPointer, "%s%02X%s\n", obj, (cntObj - 9) / 2, obj + 9);
            sprintf(obj+1, "%06X", locCtr);
            sprintf(obj+9, "%02X", opAddr);
            cntObj = 11;
        } else{
            sprintf(obj + cntObj, "%02X", opAddr);
            cntObj += 2;
        }
    }
    if(opIt->format[0] == '2'){
```

```
            fprintf(lstPointer, "%04X\n", opAddr);
            if(cntObj + 4 > 68 || resFlag){
                fprintf(objPointer, "%s%02X%s\n", obj, (cntObj - 9) / 2, obj + 9);
                sprintf(obj+1, "%06X", locCtr);
                sprintf(obj+9, "%04X", opAddr);
                cntObj = 13;
            } else{
                sprintf(obj + cntObj, "%04X", opAddr);
                cntObj += 4;
            }
        }
        if(opIt->format[0] == '3'){
            fprintf(lstPointer, prefix & 4 ? "%08X\n" : "%06X\n", opAddr);
            if(prefix & 4){
                if(cntObj + 8 > 68 || resFlag){
                    fprintf(objPointer, "%s%02X%s\n", obj, (cntObj - 9) / 2, obj + 9);
                    sprintf(obj+1, "%06X", locCtr);
                    sprintf(obj+9, "%08X", opAddr);
                    cntObj = 17;
                } else{
                    sprintf(obj + cntObj, "%08X", opAddr);
                    cntObj += 8;
                }
            }
            if(!(prefix & 4)){
                if(cntObj + 6 > 68 || resFlag){
                    fprintf(objPointer, "%s%02X%s\n", obj, (cntObj - 9) / 2, obj + 9);
                    sprintf(obj+1, "%06X", locCtr);
                    sprintf(obj+9, "%06X", opAddr);
                    cntObj = 15;
                } else{
                    sprintf(obj + cntObj, "%06X", opAddr);
                    cntObj += 6;
                }
            }
        }
        resFlag = FALSE;
    }

    fclose(intPointer), fclose(locPointer), fclose(objPointer), fclose(lstPointer);

    if(errorFlag){
        for(i = 0; i < 47; i++){        // cleaning SYMTAB
            while(symbolHead[i]){
                symPtr = symbolHead[i];
                symbolHead[i] = symbolHead[i]->link;
                --symtabSize;
                free(symPtr);
            }
        }
    } else{
        it = fName;
        while(*it) it++;
```

```
            intPointer = fopen("tmpobj", "r");
            locPointer = fopen("tmplst", "r");
            strcat(fName, ".obj");
            objPointer = fopen(fName, "w");
            *it = '\0';
            strcat(fName, ".lst");
            lstPointer = fopen(fName, "w");
            while(~(c = fgetc(intPointer))) fputc(c, objPointer);
            while(~(c = fgetc(locPointer))) fputc(c, lstPointer);
            fclose(intPointer), fclose(locPointer), fclose(objPointer), fclose(lstPointer);
        }

        return errorFlag ? -1 : 1;
}
```