과목 명: 시스템프로그래밍

담당 교수 명: 박 운 상

<<Assignment 3>>

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

```
Pass 1:
    begin
    get PROGADDR from operating system
    set CSADDR to PROGADDR {for first control section}
    while not end of input do
        begin
            read next input record {Header record for control section}
            set CSLTH to control section length
            search ESTAB for control section name
            if found then
                set error flag {duplicate external symbol}
            else
                enter control section name into ESTAB with value CSADDR
            while record type ≠ 'E' do
                begin
                    read next input record
                    if record type = 'D' then
                        for each symbol in the record do
                            begin
                                search ESTAB for symbol name
                                if found then
                                    set error flag (duplicate external symbol)
                                else
                                    enter symbol into ESTAB with value
                                        (CSADDR + indicated address)
                            end {for}
                end {while ≠ 'E'}
            add CSLTH to CSADDR {starting address for next control section}
        end {while not EOF}
    end {Pass 1}
```

2.1.2 Pass2 흐름도

```
Pass 2:

    begin
    set CSADDR to PROGADDR
    set EXECADDR to PROGADDR
    while not end of input do
        begin
            read next input record  {Header record}
            set CSLTH to control section length
            while record type ≠ 'E' do
               begin
                   read next input record
                   if record type = 'T' then
                       begin
                           {if object code is in character form, convert
                               into internal representation}
                           move object code from record to location
                               (CSADDR + specified address)
                       end {if 'T'}
                   else if record type = 'M' then
                       begin
                           search ESTAB for modifying symbol name
                           if found then
                               add or subtract symbol value at location
                                   (CSADDR + specified address)
                           else
                               set error flag (undefined external symbol)
                       end  {if 'M'}
               end {while ≠ 'E'}
            if an address is specified {in End record} then
               set EXECADDR to (CSADDR + specified address)
            add CSLTH to CSADDR
        end  {while not EOF}
    jump to location given by EXECADDR {to start execution of loaded program}
    end {Pass 2}
```

3. 모듈 정의

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

3.1.1 기능

externSymbol 에서 qsort 를 사용할 때 사용되는 비교함수이다. 각 struct 의 csAddr 을 뺀 값을 반환한다.

3.1.2 사용 변수

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

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

3.2.1 기능

총 3 개의 기능을 한다.

1. 인자가 없을 시 – 현재까지의 break point 들을 address 순으로 화면상에 출력한다
2. 인자가 'clear'일 시 – 이전에 set 된 모든 break point 를 삭제한다
3. 인자가 적절한 address 일 시 – address 에 break point 를 설정한다.

opcode 의 첫 부분 이외의 부분에 break point 를 설정하는 것은 고려하지 않았다.

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 arg[] - tok 에서 인자만을 떼어내어 저장하는 문자열.

int i – iterator. arg 를 복사할 때 사용한다.
int point– arg 에 address 가 왔을 때, 그 address 를 저장하기 위해 사용한다.

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

3.3.1 기능

주어진 파일들을 linking loader 를 통해 memory 상에 load 한다.
성공적으로 load 될 시, ESTAB 을 화면상에 출력한다 (Load map)

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.
int fileNum – 몇 개의 파일이 주어졌는지 체크하는 변수이다.
int i – fileName 을 얻기 위해 사용하는 iterator 이다.
char first[], second[], third[] – 파일 이름만을 추출하여 저장하는 문자열

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

3.4.1 기능

입력 받은 address 를 PROGADDR 로 설정한다.

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 addr –이후 progAddr 에 값을 넘기기 위해 주어진 address 를 저장한다.
int progAddr – 전역 변수로써, PROGADDR 을 저장한다.

3.5 모듈 이름: commandRun(char *tok, char com[])

3.5.1 기능

memory 에 올라간 program 을 실행하기 위해 run()을 호출한다.

3.5.2 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.

3.6 모듈 이름: externHashFunction(char *val)

3.6.1 기능

3 번째 hash function 이다. ESTAB 이라는 hash table 에 대한 index 를 구하기 위하여 사용한다.
이번엔 각 문자열에 39문자열의 위치의 가중치를 주고, overflow 를 고려하지 않고 ret 이라는 값에
더해준다. 이후 return 할 때 sign bit (MSB)을 0 으로 bit masking 한 뒤, modular 18 한 값을 return 한다.
의도적으로 overflow 를 형성하는 hash function 이 있다고 하여 구현해보았다.

3.6.2 사용변수

char *val – hash table 의 index 로 변환하려는 문자열.
int ret – return value 를 담기 위한 변수.

3.7 모듈 이름: linkingLoaderPass1(int numOfFile, …)

3.7.1 기능

Linking Loader 의 Pass1 algorithm 을 수행하는 함수이다.
파일의 수가 가변적이기 때문에 가변 인자 함수를 구현해 보았으며, 고정 인자인 numOfFile 을 통해

가장 큰 loop 인 파일단위의 loop 를 제어한다.
에러가 발생했을 시 -1 을 return 하여 곧 commandLoader 에서 오류가 발생했음을 인지할 수 있도록 하였다.
이미 assemble 과정을 거친 obj file 이기 때문에 별 다른 에러체크는 하지 않았으나, ESTAB 을 활용하여 소수의 오류는 확인하도록 하였다. (중복된 혹은 정의되지 않은 symbol)
흐름도와 같은 흐름으로 작성하였다.
에러가 발생하지 않는 경우, 각 control section 의 D 에서 define 한 external symbol 들이 담겨있는 ESTAB 이 생성된다.
Return value 는 에러가 발생했을 시 -1, 아닌 경우 1 이다

3.7.2 사용변수

unsigned int csAddr, csLength, symAddr – control section 의 시작 address, control section 의 길이, symbol 이 위치하는 address 를 저장한다.
int i, idx – iterator, hash-table 의 인덱스를 저장한다.
char filename[], line[], symbol[] – 파일 이름을 저장하는 문자열, 각각 line (object file 의 record 한 줄), symbol 을 저장하기 위한 character type array
FILE *fp - file pointer points object file
va_list names – 가변인자 함수를 사용 및 활용하기 위한 pointer
Extern *curr, *new – 각각 ESTAB 을 탐색하기 위한 iterator, 새로운 node 를 할당하기 위한 pointer

3.8 모듈 이름: linkingLoaderPass2(int numOfFile, …)

3.8.1 기능

Linking Loader 의 Pass2 algorithm 을 수행하는 함수이다. Pass1() 이 에러 없이 수행 된 경우 실행되며, memory 상에 object file 을 올리고, 이 과정에서 modification 을 수행한다. 주어진 symbol 이 중복되지 않는지, 정의 되지 않았는지 등의 linkingLoaderPass1 과 비슷한 에러처리를 하였다.
에러가 발생하지 않는다면 memory 상에 object file 들이 linking 과정을 마친 채로 올라가게 되며, 1 을 반환한다.
에러가 발생한다면 -1 을 반환한다.
이번 Pass2 에서도 address 를 수정하는 과정에서 bitwise operation 을 사용했다.
1. half byte 가 홀수일 때
*virtualMem[tmpAddr] = (virtualMem[tmpAddr] & ((1 << 8) - (1 << 4))) + (val >> --halfBytes * 4)*
첫 바이트의 상위 4bit 는 그대로 두고, 하위 4bit 에 modificated address 를 올려야 한다.
(virtualMem[tmpAddr] & ((1 << 8) - (1 << 4))) 에서 상위 4bit 만을 살린 bit masking 을 하며, *(val >> --halfBytes * 4)* 에서 modificated address 의 상위 4bit 를 lsb 쪽으로 밀어 하위 4bit 로 만든 뒤 더해주므로 정상적으로 작동함을 알 수 있다.
2. modify 한 address 를 메모리에 올릴 때
*virtualMem[tmpAddr + i] = (unsigned char)((val >> (halfBytes - (i + 1) * 2) * 4) & ((1 << 8) - 1));*
modificated address 의 상위 bit 부터 순차적으로 memory 상의 target address 에 8bit 씩 올린다.
i 번째 memory element 에는 modification address 에서 상위 8 * i 번째 bit 부터 8bit 가 들어가야하며 이는 *(val >> (halfBytes - (i + 1) * 2) * 4* 에서 해당 byte 를 떼어내고, *& ((1 << 8) - 1)* 를 통해 8bit 만을 bit masking 함을 알 수 있다. 곧 정상적으로 작동한다.

3.8.2 사용변수

unsigned int csAddr, csLength, symAddr – control section 의 시작 address, control section 의 길이, symbol 이 위치하는 address 를 저장한다.
unsigned int tmpAddr, modAddr, halfByte – address 임시 저장을 위해, 수정되어야 할 address 에 더하거나 뺄 address 를 저장하기 위해, 얼마나 많은 half-byte 에 수정할 지 저장하기 위해 선언.
unsigned int ref[], refNum – reference number 가 사용되었을 때 이를 기록, 사용하기 위해 선언.
unsigned int tLen – text record 한 줄의 길이를 저장한다.

int i, idx, val – iterator, hash-table 의 인덱스를 저장, text record 에서 혹은 modification record 에서 memory element 혹은 address 를 계산할 때 사용한다.
char filename[], line[], symbol[] – 파일 이름을 저장하는 문자열, 각각 line (object file 의 record 한 줄), symbol 을 저장하기 위한 character type array
char sign – modification record 에서 external symbol 의 csAddr 을 더할 것인지 뺄 것인지를 기억하기 위한 변수
FILE *fp - file pointer points object file
va_list names – 가변인자 함수를 사용 및 활용하기 위한 pointer
Extern *ptr – ESTAB 을 탐색하기 위한 iterator.
int lastExecAddr, progAddr – load 가 되었으므로 break point 를 활용하기 위해 선언한 변수를 초기화.

3.9 모듈 이름: printLoadmap()

3.9.1 기능

ESTAB 을 출력한다. SYMTAB 을 출력할 때와 동일하게, estabSize 로 ESTAB 에 들어가 있는 노드의 수를 기록하며 그 만큼을 동적 할당을 통해 1 차원 배열에 모든 노드를 삽입한다.
qsort 를 이용해 csAddr (symbol 이 메모리상에 위치한 address) 순으로 정렬하며, 형식에 맞게 화면상에 출력한다. (CSNAME 은 항상 모든 symbol 앞에 위치하며 (같은 CS 에 속해있다면), 0 이 아닌 length 값을 가지고 있으므로 이를 통해 CSNAME 과 SYMBOL 을 구분하였다.)

3.9.2 사용 변수

int i, idx – iterator, hash-table 의 인덱스를 저장
Extern *ptr, *arr – 각각 ESTAB iterator, 1 차원 배열에 노드를 정렬하기 위해 사용하는 동적 할당을 위한 pointer.
Extern *externHead[] - ESTAB 에서 각 head node 를 가지고 있는 배열.
int estabSize – ESTAB 에 얼마나 많은 노드가 할당되어 있는지 확인하기 위해 사용한다.

3.10 모듈 이름: getTargetAddr(int curr, int flags, int reg[])

3.10.1 기능

memory 상에 올려진 object code 는 relative address 이기 때문에, 실제로 그 값이 나타내는 address 를 계산해야 한다. 이 과정은 거의 대부분의 instruction 에 사용되기 때문에 함수로 정의하여 사용하였으며, curr 로 전해진 instruction 이 가리키는 TA 를 계산, 반환한다.
flags 의 e 가 1 이면 PC 에 4 를, 0 이면 3 을 더해주며 (format 3/4 일 경우만 이 함수를 사용하기 때문이다) 이후 TA 계산을 수행한다.
미리 define 한 macro ADDRESS 와 DISP 을 통해 address 와 disp 을 각각 계산하며, immediate 인 경우는 이 값이 곧 TA 이기 때문에 targetAddr 을 바로 이 값으로 설정한다.
PC 혹은 Base relative 인 경우에는, disp 이 음수인 경우 int 형을 사용하기 때문에 음수 계산에 오차가 생길 수 있는 점을 감안하여 계산한다.

if(disp & (1 << 11)) disp = -((~disp + 1) & ((1 << 12) - 1))

disp 의 MSB 가 1 인경우 2's complement 를 이용하여 양수 값으로 변환한 뒤, - (unary operator)를 사용하여 정상적인 음수 값을 가지도록 하였다.
이후는 케이스에 맞게 PC 혹은 B 와 더하여 TA 를 계산한다.

3.10.2 사용 변수

int curr, flags, reg[] – TA 를 구하기 위해 필요한 인자들이다. reg 는 register 값들을 담고 있으며, flags 는 nixbpe 를 담고있다 (e 가 lsb 에 위치, 차례로 1bit 씩). curr 은 현재 opcode 의 위치를 담고있다.
int targetAddr, disp – disp 은 format 3 일 때 disp 을 저장하기 위해, targetAddr 은 최종적인 TA 를 반환하기 위해 선언했다. targetAddr = Return value.

3.11 모듈 이름: getVal(int targetAddr, int flags)

3.11.1 기능

flags 에 setting 된 xbpe 를 확인하여 케이스에 맞게 value 를 memory 로부터 읽어온다.
단순한 읽어오는 기능 뿐이지만 반복적으로 사용되어 함수로 선언하여 사용하였다.

3.11.2 사용 변수

int targetAddr, flags – value 를 구하기 위해 필요한 인자들이다. flags 는 nixbpe 를 담고있으며 (e 가 lsb 에 위치, 차례로 1bit 씩), targetAddr 은 value 를 가져올 address 를 담고있다.
int ret – 메모리에 담겨있는 값을 저장한다. Return value.

3.12 모듈 이름: printReg(int reg[])

3.12.1 기능

register 에 담긴 값을 화면상에 출력한다.
"%2s: %06X %2s: %06X" 의 형식으로 총 4 줄에 걸쳐 7 개의 register 를 출력한다.

3.12.2 사용 변수

int reg[] – register 의 값을 담고 있어 참조할 때 사용한다.

3.13 모듈 이름: run()

3.13.1 기능

memory 상에 올라간 object file 을 실제로 읽어오며 실행한다.
progAddr + fileLen 까지 실행하면 종료하며, 크게 instruction 실행, break point 체크로 나눌 수 있다.
1. break point check
bptab[]이라는 Boolean array 를 통해 현재 address 에 break point 가 setting 되어있는지 확인한다. bpLast 는 마지막으로 멈춘 break point 의 주소를 저장하고 있는데, 현재 위치와 bpLast 가 같다면 이는 실행 도중 돌아온 address 가 아니라 실행 처음 위치한 address 이므로 bpLast 를 -1 로 초기화한 뒤 정상적으로 instruction 을 읽고 실행한다.
프로그램 실행 중 마주친 break point 라면, bpLast 를 현재 위치로 설정한 뒤 현재 위치와 함께 printReg()를 호출하고 함수를 종료한다.
2. instruction 실행
opcode 를 토대로 이 instruction 의 format 과 기능을 확인하고, 이를 수행한다.
code cluster 를 줄이기 위해 function pointer 를 여기서 사용하려 했으나, (opcode 는 4 씩 증가하므로 /4 로 압축하여 indexing 할 수 있을 것이다) 시간이 부족하여 구현하지 못했다. 아쉽다.
추가로 너무 늦게 생각난 enum 을 통한 reg[] indexing 또한 code 의 가독성을 대폭 높일 수 있었을 것이나 역시 시간이 부족하여 구현하지 못해 아쉬운 감이 있다.
단순한 if-else 가 중첩된 형태이고, opcode 에 맞는 if 문에 들어가 적절한 행동을 한 뒤, format 에 따라 PC 를 적절하게 증가시켜준다. ST- 라인이나 LD- 라인, J- 라인 등 수행하는 기능은 같으나 레지스터만 다른 경우 같은 if 문에 위치하게 하였고 결과값만 적절한 레지스터에 (혹은 memory 에) 기록될 수 있도록 했다.
I/O 함수 및 F register 를 사용하는 instruction 은 구현하지 않으며, 약 38 개 정도의 instruction 을 구현하였다.

3.13.2 사용 변수

int i, idx, val – iterator, hash-table 의 인덱스를 저장, memory 에서 읽어온 값을 저장한다.
int r1, r2 – format 2 instruction 을 수행할 때 두 레지스터를 저장한다.
int opcode, targetAddr, curr, flags – 현재 address 에서 읽은 opcode, 가리키는 TA, 현재 address, 현재 address 에 setting 되어 있는 flag 들을 저장한다.
int endAddr – 프로그램의 마지막 주소를 저장한다.

4. 전역 변수 및 구조체, 매크로, typedef 정의

4.1 #define DISP(x) (((*(x) & 0x0F) << 8) | (*(x+1)))

disp 을 구하는 매크로이다. disp 은 총 12bit 이며, 4 | 8 bit 로 볼 수 있기 때문에 주어진 위치에서 4bit, 다음 위치에서 8bit 를 읽어와 총 12bit 를 읽는다.

4.2 #define ADDRESS(x) ((((*(x) & 0x0F) << 16)) | (*(x+1) << 8) | (*(x+2)))

address 를 구하는 매크로이다. address 는 총 20bit 이며, 4 | 8 | 8 bit 로 볼 수 있기 때문에 주어진 위치에서 4bit, 다음 위치에서 8bit, 그 다음 위치에서 8bit 을 읽어와 총 20bit 를 읽는다.

4.3 typedef struct _Extern

ESTAB 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, CSNAME 이 저장될 경우 해당 control section 의 길이를 저장할 interger type variable,다음 node 를 연결할 link (struct _Extern type pointer)가 있다.

4.4 Extern *externHead[47]

ESTAB 구현을 위해 hash-bucket 이 필요했고, 이를 위해 선언하였다. externHead 는 각 bucket 의 head 를 가리킨다. 프로그램 시작 시, 혹은 linkingLoaderPass1() 시작 시 NULL 로 초기화된다.

4.5 int estabSize

ESTAB 의 사이즈를 저장한다. ESTAB 출력 시 노드의 개수 확인을 *O(1)*에 처리하기 위해 선언했다.

4.6 unsigned int progAddr

PROGADDR 을 저장한다. 프로그램 실행 시 0x00 으로 초기화 되며, progaddr command 를 통해 수정할 수 있다.

4.7 int fileLen

run() 을 실행할 시, 마지막 instruction 의 주소를 계산하기 위해 loader command 수행 시 memory 에 올라간 총 길이를 저장해둔다.

4.8 int lastExecAddr

마지막으로 실행되었던 address 다음 주소를 저장한다.

4.9 int bpLast

마지막으로 break 한 위치를 저장한다. -1 로 초기화하며, break 시 break 된 address 를 저장한다. break 이후 다시 run command 를 수행할 시 다음 break point 를 만나기 전까지 -1 의 값을 갖는다.

4.10 bool bptab[1048576]

memory 위 특정한 address 에 break point 가 설정되어있으면 1, 아니면 0

4.11 reg24 reg[10]

register 를 저장하는 배열이다. 차례로 A, X, L, B, S, T, (X), (X), PC, SW
((X)는 사용하지 않음을 나타낸다)

5. 코드

5.1 20151623.h

```
#ifndef _20151623_h_
```

```
#define _20151623_h_

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<stdbool.h>
#include<dirent.h>
#include<sys/stat.h>
#include<stdarg.h>
#include<unistd.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) == '+')
#define DISP(x) (((*(x) & 0x0F) << 8) | (*(x+1)))
#define ADDRESS(x) ((((*(x) & 0x0F) << 16)) | (*(x+1) << 8) | (*(x+2)))

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];

typedef struct _Extern {
	char symbol[7];
```

```
        int csAddr, length;
        struct _Extern *link;
} Extern;       // structure for external symbol
Extern *externHead[18];

unsigned char virtualMem[1048576];      // for virtual memory, 1048576 = 16 ^ 5
int dumpLastAddr;       // last address of dump command
int symtabSize; // stores symbol table's size
int estabSize;  // stores external symol table's size
unsigned int progAddr;  // starting address when excute 'run' or 'loader' command
int fileLen;    // store file length
int lastExecAddr;       // store last executed address (At opcode)
int bpLast;     // last address that stopped
bool quitFlag;  // when quitFlag stores 1 (TRUE), terminates this program
bool bptab[1048576];    // for checking break point
reg24 reg[10];
// 0: A, 1: X, 2: L, 3: B, 4: S, 5: T, 6: F, 8: PC, 9: SW

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 commandProgaddr(char *tok, char com[]);
void commandLoader(char *tok, char com[]);
void commandRun(char *tok, char com[]);
void commandBp(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 assemblerPass1(char fileName[], int *programLen);
int assemblerPass2(char fileName[], int programLen);
// define on assemble.c
```

```
int cmpE(const void *a, const void *b);
int externHashFunction(char *val);
int linkingLoaderPass1(int numOfFile, ...);
int linkingLoaderPass2(int numOfFile, ...);
void printLoadmap();
// define on linkingloader.c
int getTargetAddr(int curr, int flags, int reg[]);
int getVal(int targetAddr, int flags);
void printReg(int reg[]);
void run();
// define on run.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
	Symbol *symPtr; // for deallocating symbol table
	Extern *exPtr;  // for deallocating external symbol table
	comFuncPtr comFunc[] = { commandHelp, commandDir,
		commandQuit, commandHistory, commandDump, commandEdit,
		commandFill, commandReset, commandMnemonic, commandOplist,
		commandCat, commandCmp, commandCopy, commandTouch,
		commandHead, commandEcho, commandType, commandAssemble,
		commandSymbol, commandProgaddr, commandLoader, commandRun,
		commandBp	};
	// function pointer to reduce code cluster

	quitFlag = FALSE;
	dumpLastAddr = 0;
	progAddr = 0x00;
	historyHead = last = NULL;
	bpLast = -1;
	memset(virtualMem, 0, sizeof(virtualMem));
	memset(bptab, 0, sizeof(bptab));
	memset(reg, 0, sizeof(reg));
	for(i = 0; i < 20; i++) instructionHead[i] = NULL;
	for(i = 0; i < 47; i++) symbolHead[i] = NULL;
	for(i = 0; i < 18; i++) externHead[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

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

    for(i = 0; i < 18; i++){
        while(externHead[i]){
            exPtr = externHead[i];
            externHead[i] = externHead[i]->link;
            free(exPtr);
        }
    }     // 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 'b' : *commandNum = strcmp(com, "bp") ? -1 : 22;
				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 'l' : *commandNum = strcmp(com, "loader") ? -1 : 20;
				break;
		case 'o' : *commandNum = strcmp(com, "opcode") ? strcmp(com, "opcodelist") ? -1 : 9 : 8;
				break;
		case 'p' : *commandNum = strcmp(com, "progaddr") ? -1 : 19;
				break;
		case 'q' : *commandNum = strcmp(com, "q") * strcmp(com, "quit") ? -1 : 2;
				break;
		case 'r' : *commandNum = strcmp(com, "reset") ? strcmp(com, "run") ? -1 : 21 : 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 = assemblerPass1(fileName, &programLen)) < 0 || assemblerPass2(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 commandProgaddr(char *tok, char com[]){
	int addr;
	if(!*tok){	// when arguments does not exist
		puts("command progaddr needs argument");
		return;
	}

	addr = hexstrToInt(tok);

	if(addr < 0){	// when error occurs
	  if(addr == -1) puts("Invalid number has entered");
	  if(addr == -2) puts("Argument is too big for SIC ( over 0xFFFFF )");
	  if(addr == -3) puts("negative number has entered");
	  return;
	}

	progAddr = addr;	// set progaddr as given number

	addHistory(com);	// add to history
}

void commandLoader(char *tok, char com[]){
	int i, fileNum;
	char first[111], second[111], third[111];

	if(!*tok){	// when no file name has entered
		puts("command loader needs at least oneobject filename");
		return;
	}

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

	while(IS_INDENT(tok) && !IS_END_STRING(tok)) tok++;
	if(*tok){	// when second file has entered
		if(!IS_ALPHABET(*tok)){ // when file name starts with non-alphabet character
			puts("Invaild file name has entered");
			return;
		}

		i = 0;
		while(!IS_INDENT(tok) && !IS_END_STRING(tok)) second[i++] = *tok++;
		second[i] = '\0'; fileNum++;
	}

	while(IS_INDENT(tok) && !IS_END_STRING(tok)) tok++;
	if(*tok){	// when third file has entered
		if(!IS_ALPHABET(*tok)){ // when file name starts with non-alphabet character
			puts("Invaild file name has entered");
```

```
                return;
            }

            i = 0;
            while(!IS_INDENT(tok) && !IS_END_STRING(tok)) third[i++] = *tok++;
            third[i] = '\0'; fileNum++;
        }

        if(fileNum == 1){       // when number of file is 1
            if(linkingLoaderPass1(fileNum, first) < 0 || linkingLoaderPass2(fileNum, first) < 0) return;
        }else if(fileNum == 2){ // when number of file is 2
            if(linkingLoaderPass1(fileNum, first, second) < 0 || linkingLoaderPass2(fileNum, first, second) < 0)
return;
        }else{  // when number of file is 3
            if(linkingLoaderPass1(fileNum, first, second, third) < 0 || linkingLoaderPass2(fileNum, first, second,
third) < 0) return;
        }
        // if error occured, do not add this command to history

        printLoadmap();
        // when link & load is done successfully, print load map

        addHistory(com);        // add to history
}

void commandRun(char *tok, char com[]){
        if(*tok){       // when arguments does exist
            puts("command run does not need argument");
            return;
        }
        run();
        addHistory(com);        // add to history
}

void commandBp(char *tok, char com[]){
        int i, point;
        char arg[111];
        if(!*tok){      // print break point table
            printf("breakpoint\n");
            printf("----------\n");
            for(i = 0; i < 1048576; i++) bptab[i] ? printf("%04X\n", i) : 0;
        } else{ // add break point or clear break point table
            i = 0;
            while(!IS_INDENT(tok) && !IS_END_STRING(tok)) arg[i++] = *tok++;
            arg[i] = '\0';

            while(IS_INDENT(tok) && !IS_END_STRING(tok)) tok++;
            if(!IS_END_STRING(tok)){
                puts("Too many arguments");
                return;
            }

            if(!strcmp(arg, "clear")) memset(bptab, 0, sizeof(bptab));      // clear
```

```
            else{   // create
                point = hexstrToInt(arg);
                if(point < 0){
                    puts("Invalid argument");
                    return;
                }
                if(bptab[point]) printf("At %06X break point already exists\n", point);
                bptab[point] = TRUE;
            }
        }
        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;
}
```

5.5 linkingloader.c

```
#include "20151623.h"

int cmpE(const void *a, const void *b) { return ((Extern *)a)->csAddr - ((Extern *)b)->csAddr; }

int externHashFunction(char *val){
        int ret = 0;
        while(*val){
            ret *= 39;
            ret += *val++;
        }
        return (ret & 0x7fffffff) % 18;
}
```

```
int linkingLoaderPass1(int numOfFile, ...){
	int i, idx;
	unsigned int csAddr, csLength, symAddr;
	char fileName[111], line[111], symbol[7];
	FILE *fp;
	Extern *curr, *new;
	va_list names;
	va_start(names, numOfFile);

	for(i = estabSize = 0; i < 18; i++){
		while(externHead[i]){
			curr = externHead[i];
			externHead[i] = externHead[i]->link;
			free(curr);
		}
	}
	csAddr = progAddr;
	// initialize

	while(numOfFile--){
		vsprintf(fileName, "%s", names);
		if(!(fp = fopen(fileName, "r"))){
			printf("Failed to open file: %s\n", fileName);
			return -1;
		}
		// open file

		fgets(line, 110, fp);
		// get first line in object file

		sscanf(line, "%*c%6s%*6x%6x", symbol, &csLength);
		// get control section name and CSLTH

		idx = externHashFunction(symbol);
		curr = externHead[idx];
		while(curr && strcmp(curr->symbol, symbol)) curr = curr->link;
		if(curr){
			puts("Duplicated symbol");
			return -1;
		}
		// find duplicated symbol here

		new = (Extern *)malloc(sizeof(Extern));
		strcpy(new->symbol, symbol);
		new->csAddr = csAddr;
		new->length = csLength;
		new->link = NULL;
		if(!externHead[idx]) externHead[idx] = new;
		else{
			curr = externHead[idx];
			while(curr->link) curr = curr->link;
			curr->link = new;
		}
```

```
            ++estabSize;
            // put csname in ESTAB

            do{
                memset(line, 0, sizeof(line));
                fgets(line, 110, fp);
                if(*line != 'D') continue;
                // In pass 1, we only consider D record
                for(i = 1; !IS_END_STRING(line+i); i += 12){
                    memset(symbol, 0, sizeof(symbol));
                    sscanf(line + i, "%6s%6x", symbol, &symAddr);
                    idx = externHashFunction(symbol);
                    curr = externHead[idx];
                    while(curr && strcmp(curr->symbol, symbol)) curr = curr->link;
                    if(curr){
                        puts("Duplicated symbol");
                        return -1;
                    }
                    // find duplicated symbol here

                    new = (Extern *)malloc(sizeof(Extern));
                    strcpy(new->symbol, symbol);
                    new->csAddr = symAddr + csAddr;
                    new->length = 0;
                    new->link = NULL;
                    if(!externHead[idx]) externHead[idx] = new;
                    else{
                        curr = externHead[idx];
                        while(curr->link) curr = curr->link;
                        curr->link = new;
                    }
                    ++estabSize;
                    // put symbol in ESTAB here
                }
            } while(*line != 'E');
            // read file until last line, end record.

            csAddr += csLength;
            // starting address for next section

            fclose(fp);
        }

        fileLen = csAddr - progAddr;
        va_end(names);
        return 1;
}

int linkingLoaderPass2(int numOfFile, ...){
        int i, idx, val;
        unsigned int csAddr, execAddr, csLength, tmpAddr, modAddr, halfBytes, ref[22], refNum, tLen;
        char fileName[111], line[111], symbol[7], sign;
        FILE *fp;
```

```
Extern *ptr;
va_list names;
va_start(names, numOfFile);

csAddr = progAddr;
while(numOfFile--){
	vsprintf(fileName, "%s", names);
	if(!(fp = fopen(fileName, "r"))){
		printf("Failed to open file: %s\n", fileName);
		return -1;
	}

	*ref = 0;
	// *ref == 1 ? using reference number : does not use reference number

	fgets(line, 110, fp);
	// get first line in object file

	sscanf(line, "%*c%6s%*6x%6x", symbol, &csLength);
	ref[1] = csAddr;
	// get control section name and CSLTH
	// set reference number for section name

	do{
		val = 0;
		memset(line, 0, sizeof(line));
		fgets(line, 110, fp);
		if(*line == 'R'){	// R record
			if(!IS_ALPHABET(line[1])){	// when reference number is used
				*ref = 1;
				for(i = 1; !IS_END_STRING(line+i); i += 8){
					sscanf(line + i, "%2x%6s", &refNum, symbol);
					idx = externHashFunction(symbol);
					ptr = externHead[idx];
					while(ptr && strcmp(ptr->symbol, symbol)) ptr = ptr->link;
					if(!ptr){	// when such symbol does not defined external or does not defined at all
						printf("Undefined symbol %s\n", symbol);
						return -1;
					}
					ref[refNum] = ptr->csAddr;
				}
			}
		} else if(*line == 'T'){
			sscanf(line + 1, "%6x%2x", &tmpAddr, &tLen);
			tmpAddr += csAddr;
			for(i = 0; i < tLen * 2; i += 2){
				sscanf(line + 9 + i, "%2x", &val);
				val &= (1 << 8) - 1;
				virtualMem[tmpAddr + i/2] = (unsigned char)val;
			}
		} else if(*line == 'M'){
			sscanf(line, "%*c%6x%2x%c%6s", &tmpAddr, &halfBytes, &sign, symbol);
			tmpAddr += csAddr;
```

```
                        if(*ref){
                                refNum = hexstrToInt(symbol);
                                modAddr = ref[refNum];
                        } else{
                                idx = externHashFunction(symbol);
                                ptr = externHead[idx];
                                while(ptr && strcmp(ptr->symbol, symbol)) ptr = ptr->link;
                                if(!ptr){       // when such symbol does not defined external or does not defined at all
                                        printf("Undefined symbol %s\n", symbol);
                                        return -1;
                                }
                                modAddr = ptr->csAddr;
                        }
                        modAddr *= sign == '+' ? 1 : -1;
                        // get symbol value

                        for(i = tmpAddr; i < tmpAddr + (halfBytes + 1) / 2; val |= virtualMem[i++]) val <<= 8;
                        val += modAddr;
                        val &= ((1 << (halfBytes * 4)) - 1);
                        // add / subtract value with specific value

                        if(halfBytes & 1) virtualMem[tmpAddr] = (virtualMem[tmpAddr] & ((1 << 8) - (1 << 4))) +
(val >> --halfBytes * 4), ++tmpAddr;
                        for(i = 0; i < halfBytes / 2; i++) virtualMem[tmpAddr + i] = (unsigned char)((val >> (halfBytes -
(i + 1) * 2) * 4) & ((1 << 8) - 1));
                }
        }while(*line != 'E');

        csAddr += csLength;
        // starting address for next section

        fclose(fp);
    }

    lastExecAddr = progAddr;
    va_end(names);
}

void printLoadmap(){
    int i, idx;
    Extern *ptr, *arr;
    arr = (Extern *)malloc(sizeof(Extern) * estabSize);
    for(i = idx = 0; i < 18; i++){
        ptr = externHead[i];
        while(ptr){
            strcpy(arr[idx].symbol, ptr->symbol);
            arr[idx].csAddr = ptr->csAddr;
            arr[idx++].length = ptr->length;
            ptr = ptr->link;
        }
    }
    qsort(arr, estabSize, sizeof(Extern), cmpE);
```

```
	// sort by address where symbol located at

	printf("Control\t\tSymbol\n");
	printf("section\t\tname\t\tAddress\t\tLength\n");
	printf("-----------------------------------------------------------\n");
	for(i = 0; i < estabSize; i++){
		if(!arr[i].length) printf("\t\t%s\t\t%04X\n", arr[i].symbol, arr[i].csAddr);
		else printf("%s\t\t\t\t%04X\t\t%04X\n", arr[i].symbol, arr[i].csAddr, arr[i].length);
	}
	printf("-----------------------------------------------------------\n");

	free(arr);
}
```

5.6 run.c

```
#include "20151623.h"

int getTargetAddr(int curr, int flags, int reg[]){
	int targetAddr, disp;

	reg[8] += 3 + (flags & 1);

	if(flags & 1) targetAddr = ADDRESS(virtualMem + curr + 1);	// extended format
	else if(flags & 6){ // PC, Base relative
		disp = DISP(virtualMem + curr + 1);
		if(disp & (1 << 11)) disp = -((~disp + 1) & ((1 << 12) - 1));
		targetAddr = (flags & 2 ? reg[8] : reg[3]) + disp;
	}
	else targetAddr = DISP(virtualMem + curr + 1);
	// immediate addressing
	// calculate target address

	if(flags & 8) targetAddr += reg[1];
	// indexed addressing

	return targetAddr;
}

int getVal(int targetAddr, int flags){
	int ret = 0;
	flags >>= 4;

	if(flags == 1) ret = targetAddr;
	// immediate
	else if(flags == 2){	// indirect
		targetAddr = (virtualMem[targetAddr] << 16) | (virtualMem[targetAddr + 1] << 8) |
virtualMem[targetAddr + 2];
		ret = (virtualMem[targetAddr] << 16) | (virtualMem[targetAddr + 1] << 8) | virtualMem[targetAddr + 2];
	} else if(flags == 3) ret = (virtualMem[targetAddr] << 16) | (virtualMem[targetAddr + 1] << 8) |
virtualMem[targetAddr + 2];
	// simple

	return ret;
```

```
}

void printReg(int reg[]){
	printf("A : %06X X : %06X\nL : %06X PC: %06X\nB : %06X S : %06X\nT : %06X\n", reg[0], reg[1], reg[2],
reg[8], reg[3], reg[4], reg[5]);
	return;
}

void run(){
	int i, opcode, curr, endAddr, idx, flags, targetAddr, val, r1, r2;

	curr = reg[8] = lastExecAddr;
	endAddr = progAddr + fileLen;
	reg[2] = endAddr;
	while(reg[8] != endAddr){
		if(bptab[curr]){	// when break point is located at current location
			if(bpLast != -1) bpLast = -1;
			else{
				bpLast = lastExecAddr = curr;
				printReg(reg);
				printf("Stop at checkpoint[%04X]\n", curr);
				return;
			}
		}
		opcode = virtualMem[curr] & 0xFC;
		if(opcode == 0x00 || opcode == 0x68 || opcode == 0x74 || opcode == 0x08 || opcode == 0x6C || opcode ==
0x04 || opcode == 0x50){	// LDA, LDB, LDL, LDS, LDT, LDX, LDCH
			flags = ((virtualMem[curr] & 0x03) << 4) | ((virtualMem[curr + 1] & 0xF0) >> 4);

			targetAddr = getTargetAddr(curr, flags, reg);
			val = getVal(targetAddr, flags);

			r1 = opcode == 0x00 || opcode == 0x50 ? 0 : opcode == 0x68 ? 3 : opcode == 0x08 ? 2 : opcode ==
0x6C ? 4 : opcode == 0x74 ? 5 : 1;
			reg[r1] = opcode == 0x50 ? ((reg[0] & 0xFFFF00) | (val >> 16)) : val;
			// store value in reg
		} else if(opcode == 0x0C || opcode == 0x10 || opcode == 0x14 || opcode == 0x78 || opcode == 0x7C ||
opcode == 0x84){ // STA, STB, STL, STS, STT, STX, STL
			flags = ((virtualMem[curr] & 0x03) << 4) | ((virtualMem[curr + 1] & 0xF0) >> 4);

			targetAddr = getTargetAddr(curr, flags, reg);
			if((flags & 0x30) == 0x20) targetAddr = getVal(targetAddr, 0x30);

			r1 = opcode == 0x0C ? 0 : opcode == 0x10 ? 1 : opcode == 0x14 ? 2 : opcode == 0x78 ? 3 : opcode
== 0x7C ? 4 : 5;
			for(i = 0; i < 3; i++) virtualMem[targetAddr + i] = (unsigned char)((reg[r1] >> (2 - i) * 8) & ((1 << 8)
- 1));
			// set memory as reg
		} else if(opcode == 0x18 || opcode == 0x1C){	// ADD, SUB
			flags = ((virtualMem[curr] & 0x03) << 4) | ((virtualMem[curr + 1] & 0xF0) >> 4);

			targetAddr = getTargetAddr(curr, flags, reg);
			val = getVal(targetAddr, flags);
```

```
        reg[0] = reg[0] + (opcode == 0x18 ? val : -val);
        reg[0] &= (1 << 24) - 1;
        // add (or substract) (m..m+2) to reg A
    } else if(opcode == 0x28){      // COMP
        flags = ((virtualMem[curr] & 0x03) << 4) | ((virtualMem[curr + 1] & 0xF0) >> 4);

        targetAddr = getTargetAddr(curr, flags, reg);
        val = getVal(targetAddr, flags);

        reg[9] = reg[0] > val ? '>' : reg[0] < val ? '<' : '=';
        // save the result in SW (CC)
    } else if(opcode == 0x2C){      // TIX
        flags = ((virtualMem[curr] & 0x03) << 4) | ((virtualMem[curr + 1] & 0xF0) >> 4);

        targetAddr = getTargetAddr(curr, flags, reg);
        val = getVal(targetAddr, flags);

        ++reg[1];
        reg[9] = reg[1] > val ? '>' : reg[1] < val ? '<' : '=';
    } else if(opcode >= 0x30 && opcode <= 0x3C){    // JEQ, JGT, JLT, J
        flags = ((virtualMem[curr] & 0x03) << 4) | ((virtualMem[curr + 1] & 0xF0) >> 4);

        targetAddr = getTargetAddr(curr, flags, reg);
        if((flags & 0x30) == 0x20) targetAddr = getVal(targetAddr, 0x30);

        if(opcode == 0x30 && reg[9] == '=') reg[8] = targetAddr;
        if(opcode == 0x38 && reg[9] == '<') reg[8] = targetAddr;
        if(opcode == 0x34 && reg[9] == '>') reg[8] = targetAddr;
        if(opcode == 0x3C) reg[8] = targetAddr;
        // set PC as value when CC satisfies condition
    } else if(opcode == 0x40 || opcode == 0x44){    // AND, OR
        flags = ((virtualMem[curr] & 0x03) << 4) | ((virtualMem[curr + 1] & 0xF0) >> 4);

        targetAddr = getTargetAddr(curr, flags, reg);
        val = getVal(targetAddr, flags);

        if(opcode == 0x40) reg[0] &= val;
        else reg[0] |= val;
        // and (or or) reg A with value

        reg[0] &= (1 << 24) - 1;
    } else if(opcode == 0x48){      // JSUB
        flags = ((virtualMem[curr] & 0x03) << 4) | ((virtualMem[curr + 1] & 0xF0) >> 4);

        targetAddr = getTargetAddr(curr, flags, reg);
        if((flags & 0x30) == 0x20) targetAddr = getVal(targetAddr, 0x30);

        reg[2] = reg[8];
        reg[8] = targetAddr;
        // save return address in reg L, set PC as value just read
    } else if(opcode == 0x4C) reg[8] = reg[2];      // RSUB
    else if(opcode == 0x54){        // STCH
```

```
            flags = ((virtualMem[curr] & 0x03) << 4) | ((virtualMem[curr + 1] & 0xF0) >> 4);

            targetAddr = getTargetAddr(curr, flags, reg);
            if((flags & 0x30) == 0x20) targetAddr = getVal(targetAddr, 0x30);

            virtualMem[targetAddr] = (unsigned char)(reg[0] & 0xF);
        } else if(opcode == 0x90 || opcode == 0x94){    // ADDR, SUBR
            reg[8] += 2;    // format 2
            r1 = (virtualMem[curr + 1] & 0xF0) >> 4;
            r2 = virtualMem[curr + 1] & 0x0F;
            reg[r2] = reg[r2] + (opcode == 0x90 ? reg[r1] : -reg[r1]);
            reg[r2] &= (1 << 24) - 1;
            // add (or subtract) r1, r2 and store the result in r2
        } else if(opcode == 0xA0){      // COMPR
            reg[8] += 2;    // format 2
            r1 = (virtualMem[curr + 1] & 0xF0) >> 4;
            r2 = virtualMem[curr + 1] & 0x0F;
            reg[9] = reg[r1] > reg[r2] ? '>' : reg[r1] < reg[r2] ? '<' : '=';
            // save the result in SW (CC)
        } else if(opcode == 0xA4 || opcode == 0xA8){    // SHIFTR, SHIFTL
            reg[8] += 2;    // format 2
            r1 = (virtualMem[curr + 1] & 0xF0) >> 4;
            r2 = virtualMem[curr + 1] & 0x0F;
            while(r2--){
                if(opcode == 0xA4) reg[r1] = ((reg[r1] << 1) & ((1 << 24) - 1)) | (((reg[r1] << 1) & (1 << 24)) ?
1 : 0);
                // SHIFTR - circular shift
                else reg[r1] = (reg[r1] >> 1) | (reg[r1] & (1 << 23));
                // SHIFTL - copy left-most bit to vacated bit
            }
        } else if(opcode == 0xAC){      // RMO
            reg[8] += 2;    // format 2
            r1 = (virtualMem[curr + 1] & 0xF0) >> 4;
            r2 = virtualMem[curr + 1] & 0x0F;
            reg[r2] = reg[r1];
            // r2 <- (r1)
        } else if(opcode == 0xB4){      // CLEAR
            reg[8] += 2;    // format 2
            r1 = (virtualMem[curr + 1] & 0xF0) >> 4;
            reg[r1] = 0;    // clear
        } else if(opcode == 0xB8){      // TIXR
            reg[8] += 2;    // format 2
            r1 = (virtualMem[curr + 1] & 0xF0) >> 4;
            ++reg[1];
            // increase reg X
            reg[9] = reg[1] > reg[r1] ? '>' : reg[1] < reg[r1] ? '<' : '=';
            // set CC
        } else if(opcode == 0xE0 || opcode == 0xDC || opcode == 0xD8){  // TD, WD, RD
            reg[8] += 3 + ((virtualMem[curr + 1] & 16) >> 4);
            if(opcode == 0xE0) reg[9] = '<';
        }
        curr = reg[8];
        // move PC to LOCCTR
```

```
        }
        printReg(reg);
        puts("End program");
        memset(reg, 0, sizeof(reg));
        lastExecAddr = progAddr;        // when run ends, need to initialize lastExecAddr
        bpLast = -1;

        return;
}
```