과목 명: 시스템프로그래밍

담당 교수 명: 소 정 민

<<Assignment 3>>

**서강대학교 컴퓨터공학과**

**[20151619]**

**[최준수]**

목 차

1. 프로그램 개요 4

2. 프로그램 설명 4

2.1 프로그램 흐름도 4

3. 모듈 정의 5

3.1 모듈 이름 : int cmd\_loader (char\*\*, int) 6

3.1.1 기능

3.1.2 사용 변수

3.2 모듈 이름: void run\_opcodes (int addr) 7

3.2.1 기능

3.3 모듈이름: void extsymtab\_push(extsymtab\*\* head,char\* symbol, int addr, int length) 8

3.3.1 기능

3.3.2 사용변수

3.4 모듈이름: extsymptr extsymtab\_search(extsymtab\* head, char\* symbol) 9

3.4.1 기능

3.4.2 사용변수

3.5 모듈이름: void extsymtab\_printAll(extsymtab\*) 9

3.5.1 기능

3.5.2 사용변수

3.6 모듈이름: int bplist\_push(int addr) 9

3.6.1 기능

3.7 모듈이름: void bplist\_printAll() 9

3.7.1 기능

3.7.2 사용변수

3.8 모듈이름: int bplist\_search(int) 10

3.8.1 기능

3.8.2 사용변수

3.9 모듈이름: void bplist\_clear() 10

3.9.1 기능

3.9.2 사용변수

3.10 모듈이름: void extractStr(char\* dest, char\* source, int start, int len) 11

3.10.1 기능

3.10.2 사용변수

3.11 모듈이름: int extractStrToHex(char\* source, int start, int len) 11

3.11.1 기능

3.11.2 사용변수

3.12 모듈이름: void charArrHexCal(unsigned char\* arr, int value, int len, char operation) 11

3.12.1 기능

3.12.2 사용변수

3.13 모듈이름: int bitFormat4(int addr) 11

3.13.1 기능

3.13.2 사용변수

3.14 모듈이름: int bitToHex(int addr,int start,int end) 11

3.14.1 기능

3.14.2 사용변수

3.15 모듈이름: void compareReg(int a, int b) 12

3.15.1 기능

3.16 모듈이름: void printReg() 12

3.16.1 기능

3.17 모듈이름: void writeToMem(int addr, int regNum) 12

3.17.1 기능

3.18 모듈이름: int getLastByte(int regNum) 12

3.18.1 기능

3.18.2 사용변수

3.19 모듈이름: void storeLastByte(int regNum, int value) 12

3.19.1 기능

3.19.2 사용변수

4. 전역 변수 정의 13

4.1

5. 코드 13

# 프로그램 개요

Assembly 코드로 이루어진 파일을 읽어와 강의시간에 학습한 PASS1, PASS2를 거쳐 Listing File과 Object File을 생성해주는 프로그램을 만든다.

# 프로그램 설명

## 프로그램 흐름도

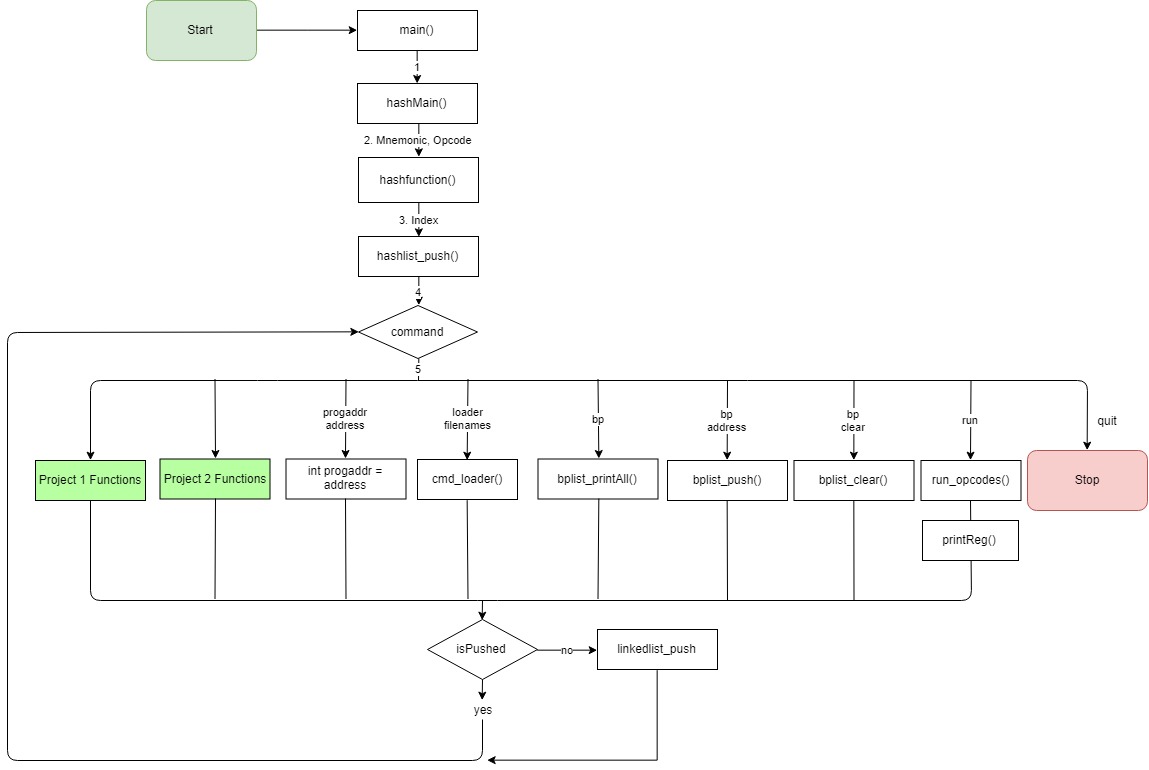
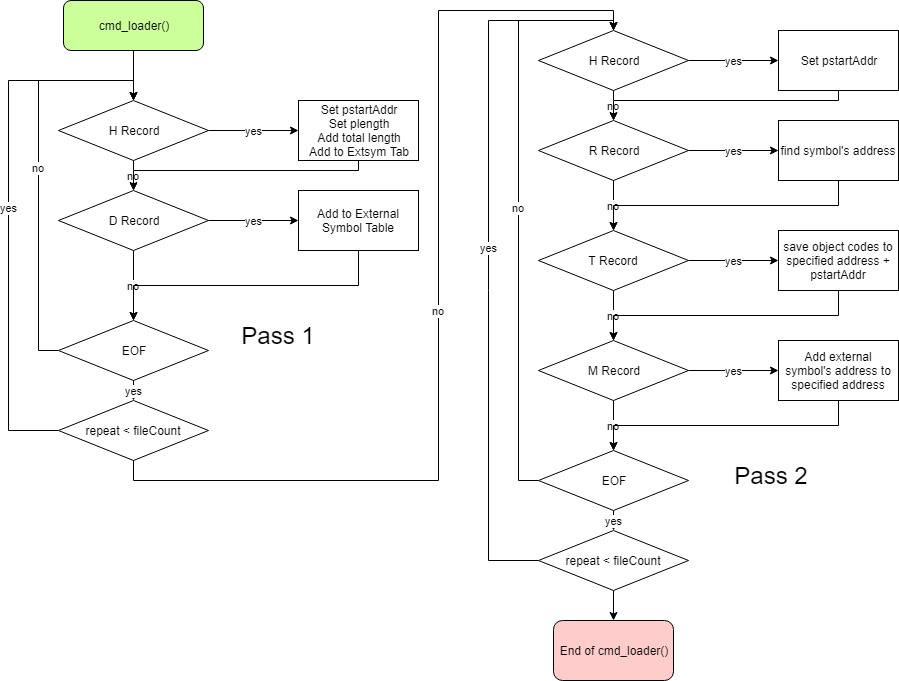
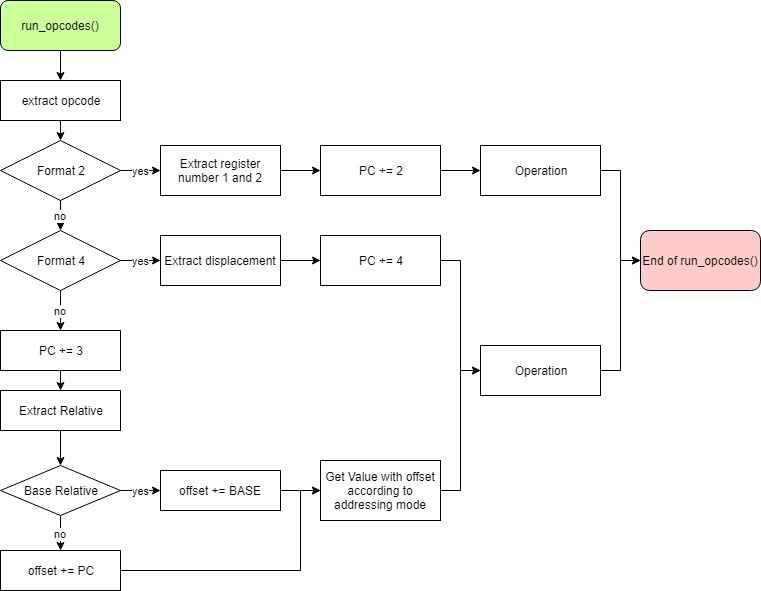


그림 1> 프로그램 흐름도



**그림 2>Loader의 흐름도**



**그림 3> run\_opcodes 흐름도**

# 모듈 정의

## 모듈 이름 : int cmd\_loader (char\*\* filename, int filenum)

### 기능

filename으로 들어온 filenum개수의 파일을 전역변수로 선언된 progaddr주소값에 로드해주는 함수이다. Assemble과 비슷하게 Pass1과 Pass2로 나뉘어 진행된다. Pass1에서는 모든 파일들을 돌며 각 프로그램의 길이와 External Definition을 확인하여 External Symbol Table에 삽입하여준다. Pass1이 끝났을 시점에는 Linking과 Loading을 진행할 모든 프로그램들이 로드될 주소와 길이, 다른 프로그램에서 참조될 external symbol들이 symbol table에 저장되어 있게 된다.

Pass1에서는 실제로 loading을 하지않고, loading을 하기 위한 사전작업을 하였다면, Pass2에서는 사전작업된 정보들을 바탕으로 실제로 메모리에 로드하게 된다. 각 레코드마다 다른 작업이 진행되는데, Header Record는 Pass1에서 Header Record에 담겨있는 정보를 모두 사용하였으므로 필요가 없다. Header Record에 있는 프로그램 이름을 가져와 External symbol table 에 검색하여 현재 프로그램이 어떤 메모리 주소에 로드되야되는지만 검사한다. R레코드에서는 (만약 있다면) Reference number를 사용하는지 확인하고, 사용한다면 배열에 Reference Number의 인덱스에 Symbol들의 주소를 External Symbol Table로부터 받아와 저장하여 준다. T레코드에서는 T레코드에 명시된 주소와 프로그램 시작주소를 더한 위치부터 Object Code를 넣어준다. Modification Record에서는 Record의 주소에 명시된 half byte수 만큼 external reference symbol의 주소값을 더하거나 프로그램의 시작주소를 더해준다.

### 사용 변수

|  |  |
| --- | --- |
| char line[300]  char temp[200]  FILE \*fp  int pstartAddr  int plength  int totallength  int isRefNum | 현재 처리하고 있는 object코드 한줄을 저장하고 있다.  extractStr을 호출할 때 line에서 부분적으로 추출한 문자열을 저장하고 있다.  현재 열고 있는 파일의 정보를 가지고 있는 파일 포인터  현재 loading하는 프로그램의 시작주소  현재 loading하는 프로그램의 길이  loading하는 모든 프로그램의 총 길이  현재 loading하는 프로그램이 reference number를 사용하여 linking을 하는지 TRUE or FALSE |

## 모듈 이름: void run\_opcodes(int addr)

### 기능

main함수에서 읽기 시작할 주소를 넘겨주면 그 주소부터 시작하게 되는데 첫 1바이트를 읽고 뒤 2비트를 버려 어떤 opcode인지를 알아내고, 그 opcode에 따라 어떻게 처리할지가 결정된다. Format2인 COMPR, CLEAR, TIXR은 뒤의 1바이트에 처리해야되는 register 2개의 number가 들어오고 각 명령에 맞는 처리를 하여준다.

만약 opcode가 Format 3/4일경우에는 6, 7비트(n, i)를 확인하여 addressing mode를 찾고, 8비트 (x)를 확인하여 indexed addressing인지를 확인하고, 9, 10비트를 확인하여 base relative 인지 pc relative인지를 확인한다. 마지막으로 11비트를 확인하여 3형식인지 4형식인지를 확인한다. 이 과정까지 마치면 이 object code에 대한 모든 분석이 끝나게 되므로 그 정보들을 가지고 뒤의 offset을 처리하면된다. PC나 B 레지스터의 값을 더해주고, indexed addressing이라면 X 레지스터의 값도 더해준다. 그리고 addressing mode에 따라 주소값에 접근하여 값을 받아오거나 그 주소값에 저장된 값 혹은 주소값 자체를 사용한다. 이 값에 실제로 접근하여 저장된 값을 받아올지 아니면 그 주소값자체를 사용할지는 opcode에 따라 다르다. 예를 들면 STA같은 명령어는 그 주소값 자체를 사용하여 A의 값을 Store해주지만 LDA같은 명령어는 주소값에 접근하여 주소값에 저장된 값을 A에 Load해준다.

이 일련의 과정이 끝난 후에는 처리했던 opcode의 형식 만큼 PC에 더해주고, 프로그램의 총길에 도달할 때까지 이 과정을 반복한다.

### 사용변수

|  |  |
| --- | --- |
| int mode  int format  int relative  unsigned int disp  int opcode | 현재 처리하고 있는 object code의 addressing mode  현재 처리하고 있는 object code의 format  현재 처리하고 있는 object code가 PC relative인지 Base relative인지 저장  현재 처리하고 있는 object code가 저장하고 있는 offset을 displacement로 바꾼 값을 저장하고 있다.  현재 처리하고 있는 object code의 opcode |

## 모듈이름: void extsymtab\_push(extsymtab\*\* head,char\* symbol, int addr, int length)

### 기능

External Symbol Table에는 순차적으로 symbol들이 삽입되기 때문에, push될 때마다 head에 연결된 linked list의 끝에 더해준다. 함수의 파라미터로 넘어온 값들은 newNode로 만들어져 추가된다. 만약 프로그램 이름이라면 그 프로그램의 길이가 length에 저장되어 있고, 그냥 external symbol이라면 length에 -1이 저장되어 추후에 검색할 때 구분이 가능하도록 했다.

### 사용변수

|  |  |
| --- | --- |
| Extsymtab\* temp  Extsymtab\* newNode | Head의 값을 보존하기 위해 Head에 저장되어 있는 값을 받아와 linkedlist를 탐색한다.  이 함수의 parameter로 넘어온 값들을 newNode에 입력하고 이 newNode를 head에 연결된 linked list의 끝에 삽입한다. |

## 모듈이름: extsymptr extsymtab\_search(extsymtab\* head, char\* symbol)

### 기능

head부터 시작하여 순차적으로 노드를 이동해가며 파라미터로 넘어온 symbol이 존재하는지 찾는다. Symbol이 존재한다면 symbol이 저장된 노드 자체의 주소값을 리턴하고, 만약 존재하지 않는다면 NULL을 반환한다.

## 모듈이름: void extsymtab\_printAll(extsymtab\*)

### 기능

head부터 시작하여 순차적으로 노드를 이동해가며 하나씩 출력한다. Program 이름인 경우는 length에 -1이 들어가 있으므로 구별이 가능하다. External symbol과 program은 출력하는 형식을 구분하여 매뉴얼에 명시된 것과 같도록 하였다.

## 모듈이름: int bplist\_push(int addr)

### 기능

Breakpoint의 검사를 위해 breakpoint를 오름차순으로 삽입한다. 전역변수인 breakpoints에 저장된 linked list에 삽입되게 된다. 만약 같은 주소값을 가진 breakpoint가 있으면 무시한다.

### 사용변수

|  |  |
| --- | --- |
| Bpptr temp  Bpptr newNode | 모든 breakpoint들을 저장하고 있는 전역변수인 breakpoints에 저장되어있는 linked list의 시작 노드의 주소값을 받아 대신 이동하여 전역변수에 저장된 노드의 주소값이 상실되지 않도록 한다.  새로운 breakpoint의 주소값을 받아 오름차순으로 맞는 위치에 들어가게 된다. |

## 모듈이름: void bplist\_printAll()

### 기능

전역변수인 breakpoints부터 시작하여 한칸씩 이동하며 지금까지 저장된 모든 breakpoint들의 주소값을 출력해준다.

### 사용변수

|  |  |
| --- | --- |
| Bpptr temp | 모든 breakpoint들을 저장하고 있는 전역변수인 breakpoints에 저장되어있는 linked list의 시작 노드의 주소값을 받아 대신 이동하여 전역변수에 저장된 노드의 주소값이 상실되지 않도록 한다. |

## 모듈이름: int bplist\_search(int addr)

### 기능

전역변수인 breakpoints부터 시작하여 한칸씩 이동하며 파라미터로 넘어온 addr와 동일한 주소값을 가진 breakpoint가 있는지 검사한다. breakpoint들은 오름차순으로 정렬되어 있기 때문에 만약 한 노드씩 이동하다가 addr보다 큰 주소값이 나왔다면 현재 breakpoint에는 addr와 같은 주소값을 가진 breakpoint가 없다는 뜻이므로 탐색을 종료하고 FALSE를 리턴한다. 만약 같은 값이 존재한다면 TRUE를 리턴한다.

### 사용변수

|  |  |
| --- | --- |
| Bpptr temp | 모든 breakpoint들을 저장하고 있는 전역변수인 breakpoints에 저장되어있는 linked list의 시작 노드의 주소값을 받아 대신 이동하여 전역변수에 저장된 노드의 주소값이 상실되지 않도록 한다. |

## 모듈이름: void bplist\_clear()

### 기능

Breakpoints 전역변수에 저장되어있는 linkedlist의 모든 노드들을 삭제한다. Temp와 temp2를 사용하여 temp2가 temp보다 한 노드 앞에서 linkedlist의 끝에 도달했는지 확인한다. 끝이 아니라면 temp를 free하고 temp에 temp2의 값을, 그리고 temp2는 다음 칸의 주소값을 갖는다.

### 사용변수

|  |  |
| --- | --- |
| Bpptr temp  Bpptr temp2 | 모든 breakpoint들을 저장하고 있는 전역변수인 breakpoints에 저장되어있는 linked list의 시작 노드의 주소값을 받아 대신 이동하여 전역변수에 저장된 노드의 주소값이 상실되지 않도록 한다.  temp보다 한 노드 앞에 위치하면서 null값인지를 확인한다. |

## 모듈이름: void extractStr(char\* dest, char\* source, int start, int len)

### 기능

Source의 start 인덱스부터 len의 길이만큼 dest에 저장하여준다. 그리고 혹시 dest에 띄어쓰기나 개행문자가 있으면 ‘\0’으로 바꿔준다.

## 모듈이름: int extractStrToHex(char\* source, int start, int len)

### 기능

source에서 start 인덱스부터 len길이만큼 dest에 저장을 한후 dest를 StrToHex()함수 호출을 통해 16진수로 변환한 후 리턴하여 준다.

### 사용변수

|  |  |
| --- | --- |
| Char dest[300] | source에서 추출된 문자열을 저장하고 있다. |

## 모듈이름: void charArrHexCal(unsigned char\* arr, int value, int len, char operation)

### 기능

16진수가 저장되어 있는 arr배열에서 len만큼 분리한 후 value를 operation에 따라 더하거나 빼준다. 여러 칸의 character배열에 저장되어 있는 16진수들을 하나의 숫자로 바꾸는 과정을 통해 converted에 합쳐진 16진수를 먼저 만들고 value와 계산한 후 다시 1바이트씩 분리하여 저장한다.

### 사용변수

|  |  |
| --- | --- |
| Int converted  Int multiplier  Int temp  Int rem  Int repeat  Unsigned int result | 여러칸에 나누어져 있는 16진수들을 하나의 숫자로 저장  Converted를 생성하는 과정에서 각 자리에 해당하는 multiplier를 곱한다  Arr에 저장된 1바이트의 값이 바뀌지않도록 temp에 저장후 사용  5half 바이트를 사용할 경우 앞의 1half 바이트를 rem에 저장한다.  Len은 half바이트의 개수이므로 이를 바이트 기준으로 몇 회 반복하여야 되는지를 저장하고 있다.  계산결과를 저장하고 있는 변수이다. |

## 모듈이름: int bitFormat4(int addr)

### 기능

addr부터 시작하는 object코드의 e 비트를 확인함으로써 4형식인지를 확인하고 맞다면 TRUE를 리턴하고 아니면 FALSE를 리턴한다.

## 모듈이름: int bitToHex(int addr,int start,int end)

### 기능

파라미터로 넘어온 addr부터 시작하여 +start번째 인덱스부터 end번째 인덱스까지의 값을 16진수로 변환하여준다. 여러 char배열에 나뉘어 있는 16진수들을 하나의 숫자로 합쳐주는 역할을 수행한다.

### 사용변수

|  |  |
| --- | --- |
| Int result | 16진수로 변환된 값을 저장하고 있다. |

## 모듈이름: void compareReg(int a, int b)

### 기능

파라미터로 넘어온 a와 b를 비교하여 만약 a가 더 크다면 SW 레지스터에 해당하는 int 배열에 -1을 저장하고, 같다면 0을, b가 더 크다면 1을 저장한다.

## 모듈이름: void printReg()

### 기능

A~T까지의 레지스터에 해당하는 int배열의 값을 매뉴얼의 형식에 맞게 출력하여준다.

## 모듈이름: void writeToMem(int addr, int regNum)

### 기능

레지스터에 저장된 3바이트의 값을 addr부터 시작하는 메모리에 저장한다. Mask (0xFF)를 사용해 1바이트씩 추출하고 temp를 8번 오른쪽으로 shift해 다음 byte를 추출한다.

### 사용변수

|  |  |
| --- | --- |
| Temp  mask | 파라미터로 넘어온 레지스터 넘버로 해당 레지스터의 값을 저장  레지스터의 값을 1바이트씩 추출하기 위해 mask를 사용한다(0xFF) |

## 모듈이름: int getLastByte(int regNum)

### 기능

regNum으로 전달받은 레지스터의 3바이트중 마지막 1바이트의 값을 리턴한다.

## 모듈이름: void storeLastByte(int regNum, int value)

### 기능

regNum으로 전달받은 레지스터의 3 바이트중 마지막 1바이트의 값을 value 로 세팅한다. Bit mask를 사용해 앞 2비트를 추출하고 value와 더하는 방식으로 수행하였다.

# 전역 변수 정의

## bpptr breakpoints

run에서 사용될 linked list의 시작 노드를 가지고 있다.

## int progaddr

progaddr 명령어를 통해 정해진 프로그램을 load & run할 주소값을 가지고 있다.

## int register[10]

프로젝트로 구현한 가상의 SIC/XE머신의 레지스터 역할을 하는 배열로 각 register number에 해당하는 index에 값이 저장되어 있다.

## int endaddr[20]

여러 개의 프로그램을 load하고 run하는 상황에 대비하여 load할때마다 그 프로그램이 끝나는 주소를 endaddr에 저장하고, 프로그램이 run될대 endaddr의 배열에 저장된 값들을 검사하여 프로그램이 끝에 도달하였는지를 확인한다.

## int endindex

endaddr이 몇번째 index까지 저장되어 있는지를 가지고 있는 변수이다.

# 코드

------------------------------------------------------------

20151619.h

------------------------------------------------------------

#ifndef \_20151619\_H\_

#define \_20151619\_H\_

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <sys/stat.h>

#include <limits.h>

#include "commands.h"

#define TRUE 1

#define FALSE 0

#define MEM\_SIZE 65536 \* 16

#define WORD 3

#define BYTE 1

#define MAX\_TEXT\_LINE 69

#include "ds.h"

unsigned char \*memory;

hashlist \*\*optable;

symptr \*symboltable;

bpptr breakpoints;

int newsymtable;

int progaddr;

// | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

// | A | X | L | B | S | T | F | | PC | SW |

int registers[10];

int endaddr[20];

int endindex;

#endif

------------------------------------------------------------

20151619.c

------------------------------------------------------------

#include "20151619.h"

#include "commands.h"

#include "util.h"

//DIR Executable

int main()

{

int isPushed = FALSE, opcode;

int argCount=0, bfrCount=0, comCount=0, realComCount=0;

char command[30],fullCmd[200];

char bfr[7][30];

unsigned int arg1,arg2,arg3;

int nextAdr = 0,cmaFlag=1;

int bpflag = FALSE, lastbp = -1;

//Initialization

lptr history = NULL;

newsymtable = TRUE;

progaddr = 0;

breakpoints = NULL;

for(int i=0;i<10;i++)

registers[i] = endaddr[i] = endaddr[10+i] = 0;

endindex = 0;

optable = (hptr\*)malloc(sizeof(hptr)\*HASH\_SIZE);

for (int i=0; i<HASH\_SIZE; i++)

optable[i] = NULL;

memory=(char\*)malloc(sizeof(char\*)\*MEM\_SIZE);

cmd\_reset();

for(int i=0; i<7; i++)

memset(bfr[i],0,sizeof(bfr[i]));

hashMain("opcode.txt");

while(TRUE)

{

printf("sicsim>");

//Get Input

fgets(fullCmd, sizeof fullCmd,stdin);

//save with the correct format

argCount = sscanf(fullCmd,"%s%x ,%x ,%x",command, &arg1, &arg2, &arg3);

//save everything as string for checking

bfrCount = sscanf(fullCmd,"%s%s%s%s%s%s%s",bfr[0],bfr[1],bfr[2],bfr[3],bfr[4],bfr[5],bfr[6]);

//For input with input value (space) , value (space) , value

if(!strcmp(",",bfr[2])){

for(int i=3; i<7; i++)

strcpy(bfr[i-1],bfr[i]);

comCount++;

}

if(!strcmp(",",bfr[3])){

for(int i=4; i<7; i++)

strcpy(bfr[i-1],bfr[i]);

comCount++;

}

for(int i=0; i<7; i++){

for(int j=0;j<strlen(bfr[i]);j++)

if(bfr[i][j] == ',')

realComCount++;

}

if (compareString(command, "opcode", NULL) && bfrCount == 2) {

//correct format for OPCODE [instruction] inserted

opcode = hashSearch\_opcode(bfr[1]);

if (opcode != -1){

printf("opcode is %X\n\n", opcode);

linkedlist\_push(&history,fullCmd);

}

//invalid [instruction]

else {

printf("Invalid Mnemonic!\n\n");

}

continue;

}

if (argCount != bfrCount - comCount) {

//cases such as dump 4, hello

cmaFlag = FALSE;

isPushed = TRUE;

}

// cases such as dump 1 , 10 ,

else if (comCount != 0 && argCount - 2 < comCount){

cmaFlag = FALSE;

isPushed = TRUE;

}

if (compareString(command, "h", "help") && argCount == 1){

cmd\_help();

isPushed = FALSE;

}

else if(compareString(command,"type",NULL) && argCount == 2){

cmd\_type(bfr[1]);

isPushed = FALSE;

}

else if(compareString(command,"symbol",NULL) && argCount == 1){

if(!newsymtable)

symtab\_printAll();

else {

printf("Not assembled yet!\n");

isPushed = TRUE;

}

}

//Maximum number of input is 3

else if (compareString(command, "du", "dump") && argCount <= 3 && cmaFlag) {

switch (argCount) {

//when only dump is inserted

case 1:

arg1 = INT\_MIN;

arg2 = INT\_MIN;

break;

//when dump start is inserted

case 2:

arg2 = INT\_MIN;

isPushed = checkHex(bfr[1]) && checkComma(bfr[1]);

isPushed = !isPushed;

break;

//when dump start, end is inserted

case 3:

isPushed = checkHex(bfr[1]) && checkHex(bfr[2]) && checkComma(bfr[2]);

isPushed = !isPushed;

break;

}

if (!isPushed)

isPushed = !(cmd\_dump(arg1, arg2, &nextAdr));

//Comma at the end

else

printf("Invalid command!\n");

}

else if (compareString(command, "e", "edit") && argCount == 3 && cmaFlag){

isPushed = checkComma(bfr[2]) && checkHex(bfr[1]) && checkHex(bfr[2]);

isPushed = !isPushed;

if (!isPushed)

isPushed = !(cmd\_edit(arg1, arg2));

else

printf("Invalid command!\n");

}

else if (compareString(command, "f", "fill") && argCount == 4 && cmaFlag){

//check if the command finished with a comma or if input has non-hex

isPushed = checkComma(bfr[3]) && checkHex(bfr[1]) && checkHex(bfr[2]) && checkHex(bfr[3]);

isPushed = !isPushed;

if (!isPushed)

isPushed = !(cmd\_fill(arg1,arg2, arg3));

else

printf("Invalid command!\n");

}

else if (compareString(command, "loader", NULL) && inRange(2,4,bfrCount) && realComCount == 0){

char \*\*filenames = (char\*\*)malloc(sizeof(char\*)\*3);

for(int j=0;j<3;j++){

filenames[j] = (char\*)malloc(sizeof(char)\*30);

strcpy(filenames[j],bfr[j+1]);

}

cmd\_loader(filenames,bfrCount-1);

}

else if(compareString(command,"assemble",NULL) && bfrCount == 2){

isPushed = cmd\_assemble(bfr[1]);

}

else if (compareString(command, "d", "dir") && argCount == 1){

cmd\_dir();

isPushed = FALSE;

}

else if (compareString(command, "q", "quit") && argCount == 1){

break;

}

else if (compareString(command,"progaddr",NULL) && argCount == 2 && cmaFlag){

progaddr = arg1;

printf("Progaddr set to %X\n",progaddr);

isPushed = FALSE;

}

else if (compareString(command, "hi", "history") && argCount == 1) {

linkedlist\_push(&history, fullCmd);

linkedlist\_print(history);

isPushed = TRUE;

}

else if (compareString(command, "reset", NULL) && argCount == 1){

cmd\_reset();

isPushed = FALSE;

}

else if (compareString(command, "opcodelist", NULL) && argCount == 1){

hashlist\_printAll(optable);

isPushed = FALSE;

}

else if (compareString(command, "bp", NULL) && bfrCount < 3 && cmaFlag){

//print all BPs

if (bfrCount == 1)

bplist\_printAll();

else if (bfrCount == 2){

//clear BPs

if(!strcmp(bfr[1],"clear")){

bplist\_clear();

printf("[ok] clear all breakpoints\n");

}

//add a new BP

else if(argCount == 2){

bplist\_push(arg1);

printf("[ok] create breakpoint %4X\n",arg1);

}

//invalid input

else

{

printf("Invalid input!\n");

isPushed = TRUE;

}

}

}

else if (compareString(command, "run", NULL) && bfrCount == 1){

int endflag = TRUE;

if(bpflag == FALSE)

registers[registerNum("PC")] = progaddr;

while(endflag){

int pcreg = registers[registerNum("PC")];

if(bplist\_search(pcreg) && pcreg != lastbp){

bpflag = TRUE;

printReg();

lastbp = pcreg;

printf("Stop at checkpoint[%4X]\n",lastbp);

break;

}//if

run\_opcodes(registers[registerNum("PC")]);

//check for end of program

for(int k=0;k < endindex;k++){

if (registers[registerNum("PC")] >= endaddr[k]){

endflag = FALSE;

bpflag = FALSE;

lastbp = -1;

printReg();

printf("End Program\n");

}

}//for

}//while

}

else {

isPushed = TRUE;

printf("Invalid Command!\n");

}

if (isPushed == TRUE)

isPushed = FALSE;

else

linkedlist\_push(&history,fullCmd);

//RESET

for(int i=0; i<7; i++)

memset(bfr[i],0,sizeof(bfr[i]));

argCount = bfrCount = comCount = realComCount = 0;

arg1 = arg2 = arg3 = INT\_MIN;

cmaFlag = TRUE;

printf("\n");

}

}

------------------------------------------------------------

commands.h

------------------------------------------------------------

#ifndef \_COMMANDS\_H\_

#define \_COMMANDS\_H\_

#define CHAR 0

#define HEX 1

#define DEC 2

#define SIMPLE 3

#define INDIRECT 2

#define IMMEDIATE 1

void cmd\_dir();

void cmd\_help();

int cmd\_loader(char\*\*,int);

void cmd\_type(char\*);

int cmd\_assemble(char\*);

void cmd\_reset();

int cmd\_dump(int, int, int\*);

int cmd\_edit(int, int);

int cmd\_fill(int, int, int);

int hashSearch\_opcode(char\*);

char\* hashSearch\_format(char\*);

int addressingMode(char\*);

void run\_opcodes(int addr);

#endif

------------------------------------------------------------

commands.c

------------------------------------------------------------

#include "commands.h"

#include "util.h"

#include "20151619.h"

#include <math.h>

#include "ds.h"

//Prints Help

void cmd\_help() {

printf("h[elp]\nd[ir]\nq[uit]\nhi[story]\ndu[mp] [start,end]\n");

printf("e[dit] address, value\nf[ill] start, end, value\nreset\n");

printf("opcode mnemonic\nopcodelist\nassemble filename\ntype filename\nsymbol\n");

printf("progaddr [address]\nloader [filename1] [filename2] [filename3]\nrun\nbp [address|clear]\n");

return;

}

int cmd\_loader(char \*\*file, int fileCount){

extsymtab \*head=NULL;

char line[300]={0,};

char temp[200]={0,};

FILE \*fp;

int pstartAddr=0,plength=0,totallength=0;

//check if invalid file name is included

for(int i=0; i<fileCount; i++){

fp = fopen(file[i],"r");

if(fp == NULL){

printf("invalid file name!\n");

return TRUE;

}

fclose(fp);

}

// INITIALIZATION

memset(temp, 0, sizeof temp);

pstartAddr = progaddr;

plength = 0;

//PASS1 : build external symbol table

for(int filenum=0;filenum < fileCount; filenum++){

fp = fopen(file[filenum],"r");

while(fgets(line,sizeof(line),fp)){

if(line[0] == 'H'){

extractStr(temp,line,1,6);

pstartAddr += extractStrToHex(line,7,6) + plength;

plength = extractStrToHex(line,13,6);

extsymtab\_push(&head,temp,pstartAddr,plength);

totallength += plength;

if(filenum + 1 == fileCount){

endaddr[endindex++] = progaddr + totallength;

registers[registerNum("L")] = progaddr + totallength;

}

}

else if(line[0] == 'D'){

int symvalue;

for(int i=0;;i++){

extractStr(temp,line,1+i\*6\*2,6);

symvalue = extractStrToHex(line,7+i\*6\*2,6);

//finished processing D record

if(temp[0] == 0 || temp[0] == '\n')

break;

if(extsymtab\_search(head,temp) != NULL){

printf("Same Symbol %s Defined Twice!\n",temp);

return TRUE;

}

else{

extsymtab\_push(&head, temp, symvalue+pstartAddr, -1);

}

}//for

}//else if

memset(temp, 0, sizeof temp);

memset(line, 0, sizeof line);

}//while

fclose(fp);

}//for

//PASS2 : load to memory

for(int filenum=0;filenum < fileCount; filenum++){

int refNum[50]={0,};

extsymptr searchResult;

fp = fopen(file[filenum],"r");

while(fgets(line,sizeof(line),fp)){

if(line[0] == 'T'){

int textAddr,textLen;

//address for text record to be loaded

textAddr = extractStrToHex(line,1,6) + pstartAddr;

//length of text record

textLen = extractStrToHex(line,7,2);

for(int j=0;j<textLen;j++)

memory[textAddr+j] = extractStrToHex(line,j\*2+9,2);

}//T record

else if(line[0] == 'R'){

for(int i=0;;i++){

//set external symbols to their reference number

int refIndex = extractStrToHex(line,1+8\*i,2);

extractStr(temp, line, 3+8\*i, 6);

searchResult = extsymtab\_search(head,temp);

if(temp[0] == 0 || temp[0] == '\n')

break;

if (searchResult == NULL){

printf("Symbol %s referenced without Definition\n",temp);

return TRUE;

}

refNum[refIndex] = searchResult->addr;

}

}//R record

else if(line[0] == 'H'){

//save prog name at the first index of reference Number

extractStr(temp,line,1,6);

searchResult = extsymtab\_search(head,temp);

refNum[1] = searchResult -> addr;

//set current program's start address

pstartAddr = searchResult -> addr;

}

else if(line[0] == 'M'){

int FixAddr = extractStrToHex(line,1,6) + pstartAddr;

int FixLen = extractStrToHex(line,7,2);

int FixRefIndex;

//relocatable program

if(strlen(line) < 12)

charArrHexCal(&(memory[FixAddr]),pstartAddr,FixLen,'+');

//uses reference number

else if(strlen(line) < 14){

FixRefIndex = extractStrToHex(line,10,2);

charArrHexCal(&(memory[FixAddr]),refNum[FixRefIndex],FixLen,line[9]);

}

else{

extractStr(temp,line,10,6);

searchResult = extsymtab\_search(head,temp);

charArrHexCal(&(memory[FixAddr]),searchResult->addr,FixLen,line[9]);

}

}//M record

else if(line[0] == 'E'){

break;

}//E record

memset(temp, 0, sizeof temp);

memset(line, 0, sizeof line);

}//while

}//for

extsymtab\_printAll(head);

//print total length!!

printf("------------------------------------------------------\n");

printf("\t\t\t\ttotal length\t%04X\n",totallength);

}

void cmd\_type(char\* filename){

FILE \*fp = fopen(filename,"r");

char line[100]={0,};

if (fp == NULL){

printf("File doesn't exist!\n");

return;

}

//prints 1 line at a time until FILE ends

while(fgets(line,100,fp) != NULL)

printf("%s",line);

fclose(fp);

return;

}

int cmd\_assemble(char\* filename){

FILE \*fp = fopen(filename, "r");;

char line[200]={0,}, line2[200]={0,};

char label[30], operand[30], operand2[30], operation[30], \*tempFormat;

char\* tempContent, \*fname, fname1[30], fname2[30];

int argCount, num, numCount, base, hashindex, totalLength;

int isFirst = TRUE, errorFlag = FALSE;

int opcode,format=-1, locctr=0,linecount;

intermptr intermediate=NULL;

intermptr newinterm;

symptr\* tempsymtab;

//FILE \*fp = fopen(bfr[1],"r");

if(fp == NULL){

printf("File doesn't exist!\n\n");

return TRUE;

}

//PASS 1

tempsymtab = (symptr\*)calloc(SYM\_SIZE,sizeof(symptr)\*SYM\_SIZE);

linecount = 0;

//Get 1 line at a time from file until NULL

while(fgets(line,200,fp) != NULL && errorFlag == FALSE){

int i=0;

//count line for error

linecount++;

//empty line

if(!strcmp(line,"") || !strcmp(line,"\n"))

continue;

//ignore comment line

i=0;

while(line[i] == ' ' || line[i] =='\t')

i++;

//if it's a comment line

if(line[i] == '.'){

newinterm = (intermptr)calloc(1,sizeof(interm));

strcpy(newinterm->line,line);

newinterm->addr = -1;

interm\_push(&intermediate,newinterm);

continue;

}

//seperates a line by label, directive and operands

argCount = asmSeparater(line,label,operation,operand,operand2);

//checks if the first line is START

if(isFirst){

//sets initial location counter with the operand

if(!strcmp(operation,"START"))

locctr = StrToHex(operand);

//if start address wasn't given, set to 0

else

locctr = 0;

isFirst = FALSE;

}

//if there is a symbol

if(strcmp(label,"")){

//get the index in the symbol table

int index = symfunction(label);

//insert into that index of symbol table

if(symtab\_push(&(tempsymtab[index]), label, locctr) == TRUE){

printf("ERROR at line %d\n",linecount);

errorFlag = TRUE;

}

}

//make new node for a line of intermediate file

newinterm = (intermptr)calloc(1,sizeof(interm));

newinterm->addr = locctr;

newinterm->next = NULL;

newinterm->argCount = argCount;

//Handle format 4 exception

if(operation[0] == '+'){

format = 4;

//extract from index 1

tempFormat = hashSearch\_format(&(operation[1]));

opcode = hashSearch\_opcode(&(operation[1]));

strcpy(newinterm->operation,&(operation[1]));

if(tempFormat == NULL){

printf("ERROR at line %d\n",linecount);

errorFlag = TRUE;

continue;

}

}

else{

tempFormat = hashSearch\_format(operation);

opcode = hashSearch\_opcode(operation);

strcpy(newinterm->operation,operation);

}

//Set format for 1,2,3

if(tempFormat != NULL && format != 4){

if(!strcmp(tempFormat,"3/4"))

format = 3;

else

format = (int)tempFormat[0] - (int)'0';

}

//make new node for a line of intermediate file

newinterm->format = format;

strcpy(newinterm->operand,operand);

strcpy(newinterm->operand2,operand2);

strcpy(newinterm->line,line);

strcpy(newinterm->label,label);

interm\_push(&intermediate,newinterm);

//increase LOCCTR

if(format != -1){

locctr += format;

}//if

//directives without opcode such as RESB, RESW

else{

if(!strcmp(operation,"RESW")){

num = StrToInt(operand);

locctr += num \* 3;

}

else if(!strcmp(operation,"RESB")){

num = StrToInt(operand);

locctr += num;

}

else if(!strcmp(operation, "BYTE")){

//remove indicator X and C

tempContent = extractContent(operand);

//if the input is in Hexadecimal

if(operand[0] == 'X'){

num = (int)strlen(tempContent);

locctr += ceil((float)num/2);

}

//if the input is in Character

else if(operand[0] == 'C'){

locctr += strlen(tempContent);

}

//other than these 2, error

else{

printf("ERROR at line %d\n",linecount);

errorFlag = TRUE;

}

}

else if(!strcmp(operation, "WORD")){

num = StrToInt(operand);

locctr += 3;

}

else if(!strcmp(operation,"END")){

totalLength = locctr;

}

else if(compareString(operation,"START","BASE")){

opcode = 0;

}

//unknown directive

else{

printf("ERROR at line %d\n",linecount);

errorFlag = TRUE;

continue;

}

}//else

//reset

format = -1;

memset(operation, 0, sizeof operation);

memset(label, 0, sizeof label);

memset(operand, 0, sizeof operand);

memset(operand2, 0, sizeof operand2);

}//while

fclose(fp);

//stop if there was an error

if(errorFlag)

return TRUE;

//--------------------------------------------

//PASS 2

//--------------------------------------------

if (intermediate == NULL){

printf("No lines to assemble\n");

return TRUE;

}

else{

int offset, pc, hashindex, newlineadr;

int colcount=0,colcount2=0, linenum=0, newlineFlag=FALSE;

int \*modification = (int\*)malloc(sizeof(int)\*300);

int modfIdx=0,modfSize=300;

unsigned int obj12=0,obj3=0,obj4=0;

char strcontent[30];

FILE \*lstfile, \*objfile;

intermptr curline;

curline = intermediate;

//open list file and object file

fname = strtok(filename,".");

strcpy(fname1,fname);

strcpy(fname2,fname);

strcat(fname1,".lst");

strcat(fname2,".obj");

lstfile = fopen(fname1,"w+");

objfile = fopen(fname2,"w+");

//reset values;

colcount = 9;

colcount2 = 0;

newlineadr = -1;

memset(line,0,sizeof(line));

//if first line isn't Start

if(strcmp(curline->operation,"START"))

fprintf(objfile,"H 000000%06X\n",totalLength);

linecount = 0;

while(curline != NULL && errorFlag == FALSE){

char objstr[3][20];

linecount++;

//lines that don't produce opcodes

curline->line[strlen(curline->line)-1] = '\0';

//print line number

fprintf(lstfile,"%04d\t",linenum);

linenum +=5;

//if current line of intermediate file is a comment line

if(curline->addr == -1){

fprintf(lstfile,"\t%s\n",curline->line);

curline = curline->next;

continue;

}

//print location counter if it's not END

if(strcmp(curline->operation,"END"))

fprintf(lstfile,"%04X\t",curline->addr);

else

fprintf(lstfile," \t");

//print assembly code

fprintf(lstfile,"%-30s",curline->line);

//if current line is RESW or RESB, start new line

if(compareString(curline->operation,"RESW","RESB")){

fprintf(lstfile,"\n");

newlineFlag = TRUE;

curline = curline->next;

continue;

}

//if current line is BASE, move to the next line

else if(!strcmp(curline->operation,"BASE")){

fprintf(lstfile,"\n");

curline = curline->next;

continue;

}

//if current line is START

else if(!strcmp(curline->operation,"START")){

fprintf(lstfile,"\n");

//Start Header record

fprintf(objfile,"H");

fprintf(objfile,"%-6s",curline->label);

fprintf(objfile,"%06X",curline->addr);

fprintf(objfile,"%06X\n",totalLength-curline->addr);

//start a new Text Record

fprintf(objfile,"T%06X",curline->addr);

curline = curline->next;

continue;

}

else if(!strcmp(curline->operation,"END")){

fprintf(lstfile,"\n");

//print last line TEXT LINE of obj file

if(strcmp(line,""))

fprintf(objfile,"%02X%s",(int)(strlen(line))/2,line);

if(strcmp(line2,""))

fprintf(objfile,"\nT%06X%s",newlineadr,line2);

//modification record

for(int k=0;k<modfIdx;k++)

fprintf(objfile,"\nM%06X05",modification[k]);

//get the address of the symbol that END points

hashindex = symfunction(curline->operand);

offset = symtab\_search(tempsymtab[hashindex],curline->operand);

fprintf(objfile,"\nE%06X\n",offset);

curline = curline->next;

continue;

}

//initialize variables

obj12 = obj3 = obj4 = 0;

for(int i=0;i<3;i++)

memset(objstr[i],0,sizeof(objstr[i]));

//intialize new line

if(newlineFlag == TRUE){

//if there's anything in line

if(strcmp(line,"")){

fprintf(objfile,"%02X%s\n",(int)(strlen(line))/2,line);

memset(line,0,sizeof(line));

}

//line2 is empty

if(!strcmp(line2,"")){

fprintf(objfile,"T%06X",curline->addr);

}

//if line2 isn't empty

else{

fprintf(objfile,"T%06X",newlineadr);

strcpy(line,line2);

memset(line2,0,sizeof(line2));

}

//reset values for new line

colcount = 9 + colcount2;

colcount2 = 0;

newlineadr = -1;

newlineFlag = FALSE;

}

//first 2 digits of object code

obj12 = hashSearch\_opcode(curline->operation);

//format 2

if(curline->format == 2){

//if command is SVC

if(!strcmp(curline->operand,"SVC")){

int temp = StrToHex(curline->operand);

obj3 = insertHexAt(obj3,temp,1);

}

else{

int temp = registerNum(curline->operand);

//if operand isn't a register

if(temp == -1){

printf("ERROR at line %d\n",linecount);

errorFlag = TRUE;

continue;

}

obj3 = insertHexAt(obj3,temp,1);

//If second operand exists

if(curline->argCount < 0){

//if directive is SHIFTL or SHIFTR, 2nd operand is an integer

if(compareString(curline->operand2,"SHIFTL","SHIFTR")){

temp = StrToHex(curline->operand2);

obj3 = insertHexAt(obj3,temp,0);

}

else{

temp = registerNum(curline->operand2);

//if operand isn't a register

if(temp == -1){

printf("ERROR at line %d\n",linecount);

errorFlag = TRUE;

continue;

}

obj3 = insertHexAt(obj3,temp,0);

}

}

}//else

}

else if(curline->format >= 3){

//set e flag for format 4

if(curline->format == 4)

obj3++;

if(abs(curline->argCount) > 1){

//set flag for addressing mode

obj12 += addressingMode(curline->operand);

//Set flag for indexed addressing

if(!strcmp(curline->operand2,"X"))

obj3 += 8;

//remove first character

if(curline->operand[0] == '#' || curline->operand[0] == '@')

strcpy(strcontent,curline->operand +1);

else

strcpy(strcontent,curline->operand);

hashindex = symfunction(strcontent);

offset = symtab\_search(tempsymtab[hashindex],strcontent);

//Check if operation is LDB and set base variable

if(!strcmp(curline->operation,"LDB"))

base = offset;

//operand doesn't exist in the symbol table

if(offset == -1){

int temp = StrToInt(strcontent);

if (temp != -1)

obj4 += temp;

else{

printf("ERROR at line %d\n",linecount);

errorFlag = TRUE;

continue;

}

}

else{

//if format4, direct addressing

if(curline->format == 4){

//if it is not immediate addressing

if(curline->operand[0] != '#'){

//add to modification record

modification[modfIdx++] = curline->addr + 1;

if(modfIdx == modfSize - 1){

modfSize \*= 2;

modification = (int\*)realloc(modification,sizeof(int)\*modfSize);

}

}

obj4 = offset;

}

else{

//check for PC relative

pc = curline->next->addr;

if(offset - pc >= -2048 && offset - pc <= 2047){

//use pc relative

obj4 = offset - pc;

obj3 += 2;

}

else{

//use base relative

obj4 = offset - base;

obj3 += 4;

}

}

}

}//if argCount > 1

else{

obj12 += 3;

}

}

//When BYTE directive is inserted

else if(!strcmp(curline->operation, "BYTE")){

//Extract content, removing C and X

char \*tempContent = extractContent(curline->operand);

if(curline->operand[0] == 'X'){

//write content to file

fprintf(lstfile,"%s\n",tempContent);

//if this line can fit into the current line of object file

if(colcount + strlen(tempContent) <= MAX\_TEXT\_LINE){

colcount += strlen(tempContent);

strcat(line,tempContent);

}

//if this line needs to be on the next line of object file

else{

newlineFlag = TRUE;

colcount2 = strlen(tempContent);

newlineadr = curline->addr;

strcat(line2,tempContent);

}

}

//Character inserted

else if(curline->operand[0] == 'C'){

char convstr[10];

char tempstr[10];

//convert each character to corresponding ASCII in hex and write to file

for(int i=0; i<strlen(tempContent); i++){

fprintf(lstfile,"%2X",(int)(tempContent[i]));

sprintf(convstr,"%2X",(int)(tempContent[i]));

strcat(tempstr,convstr);

}

fprintf(lstfile,"\n");

//if this line can fit into the current line of object file

if(colcount + strlen(tempstr) <= MAX\_TEXT\_LINE){

colcount += strlen(tempstr);

strcat(line,tempstr);

}

//if this line needs to be on the next line of object file

else{

newlineFlag = TRUE;

colcount2 = strlen(tempstr);

newlineadr = curline->addr;

strcat(line2,tempstr);

}

}

else

fprintf(lstfile,"%s\n",curline->operand);

curline = curline->next;

continue;

}//format3,4 if

else if(!strcmp(curline->operation, "WORD")){

char tempstr[10];

int temp = StrToInt(curline->operand);

sprintf(tempstr,"%06X",temp);

fprintf(lstfile,"%06X\n",temp);

//if this line can fit into the current line of object file

if(colcount + strlen(tempstr) <= MAX\_TEXT\_LINE){

colcount += strlen(tempstr);

strcat(line,tempstr);

}

//if this line needs to be on the next line of object file

else{

newlineFlag = TRUE;

colcount2 = strlen(tempstr);

newlineadr = curline->addr;

strcat(line2,tempstr);

}

curline = curline->next;

continue;

}

//make first 2 Hexadecimal

sprintf(objstr[0],"%02X",obj12);

//make the rest of object codes

switch(curline->format){

//format 1

case 1:

fprintf(lstfile,"%02X\n",obj12);

break;

//format 2

case 2:

fprintf(lstfile,"%02X%02X\n",obj12,obj3);

sprintf(objstr[1],"%02X",obj3);

break;

//format 3

case 3:

fprintf(lstfile,"%02X%01X%03X\n",obj12,obj3,obj4%(1<<12));

sprintf(objstr[1],"%01X",obj3);

sprintf(objstr[2],"%03X",obj4%(1<<12));

break;

//format 4

case 4:

fprintf(lstfile,"%02X%01X%05X\n",obj12,obj3,obj4%(1<<20));

sprintf(objstr[1],"%01X",obj3);

sprintf(objstr[2],"%05X",obj4%(1<<20));

break;

}

//if this line can fit into the current line of object file

if(colcount + curline->format\*2 <= MAX\_TEXT\_LINE){

colcount += curline->format\*2;

for(int k=0;k<curline->format && k<3;k++)

strcat(line,objstr[k]);

}

//if this line needs to be on the next line of object file

else{

newlineFlag = TRUE;

colcount2 = curline->format\*2;

newlineadr = curline->addr;

for(int k=0;k<curline->format && k<4;k++)

strcat(line2,objstr[k]);

}

curline = curline->next;

}//while

fclose(lstfile);

fclose(objfile);

}//else

if(errorFlag){

remove(fname1);

remove(fname2);

return TRUE;

}

//if symbol table is alreay set, clear

if(!newsymtable){

for(int i = SYM\_SIZE; i>=0; i--)

free(symboltable[i]);

free(symboltable);

symboltable = NULL;

}

printf("\noutput file: [%s], [%s]\n",fname1,fname2);

//successfully assembled. Set global symbol table to current file's symbol table

symboltable = tempsymtab;

newsymtable = FALSE;

return FALSE;

}

//Prints files in current directory

void cmd\_dir() {

DIR \*dir;

struct dirent \*file;

struct stat info;

if ((dir = opendir("./")) != NULL) {

/\* print all the files and directories within directory \*/

while ((file = readdir(dir)) != NULL) {

stat(file->d\_name, &info);

//If file is a directory

if(S\_ISDIR(info.st\_mode))

printf("%s/\n",file->d\_name);

//If file is executable

else if(info.st\_mode & S\_IXUSR)

printf("%s\*\n", file->d\_name);

//other files

else

printf("%s\n",file->d\_name);

}

closedir(dir);

}

else {

/\* could not open directory \*/

perror("");

printf("Error opening directory");

}

return;

}

//Prints memory

int cmd\_dump(int start, int end, int\* nextAdr) {

int row, col, over = FALSE;

int rowStart, rowEnd, curAdr;

//if ending addr is not inserted

if (end == INT\_MIN) {

//if starting addr is not inserted

if (start == INT\_MIN)

start = \*nextAdr;

end = start + 159;

}

else{

//if inserted range end is out of bound

if (checkRange(end) == FALSE || checkRange(start) == FALSE){

printf("Invalid Range\n");

return FALSE;

}

}

//if start addr is bigger than end addr

if (end < start) {

printf("Invalid Range\n");

return FALSE;

}

// if start is out of memory range

if (!checkRange(start)) {

printf("Invalid Range\n");

return FALSE;

}

rowStart = start / 16;

rowEnd = end / 16;

for (row = rowStart; row <= rowEnd; row++) {

// if row is out of range

if (row \* 16 > 0xFFFFF) {

over = TRUE;

break;

}

printf("%05X ", row \* 16);

//Content

for (col = 0; col < 16; col++) {

curAdr = row \* 16 + col;

if (curAdr > 0xFFFFF) {

over = TRUE;

break;

}

//Range before start

if (curAdr < start)

printf(" ");

//Range after end

else if (curAdr > end)

printf(" ");

else

printf("%02X ", memory[curAdr]);

}

printf("; ");

//Value

for (col = 0; col < 16; col++)

{

curAdr = row \* 16 + col;

if (curAdr > 0xFFFFF) {

over = TRUE;

break;

}

//Range before start

if (curAdr < start)

printf(". ");

//Range after end

else if (curAdr > end)

printf(". ");

else {

if ((int)memory[curAdr] >= 0x20 && (int)memory[curAdr] <= 0x7E)

printf("%c ", memory[curAdr]);

else

printf(". ");

}

}

printf("\n");

}

if (over)

\*nextAdr = 0;

else

\*nextAdr = end + 1;

return TRUE;

}

//change a memory's content

int cmd\_edit(int adr, int value) {

if (!checkRange(adr)) {

printf("Wrong Address!\n");

return FALSE;

}

memory[adr] = value;

return TRUE;

}

//changes memory to value from start adr to end adr

int cmd\_fill(int start, int end, int value) {

int rowStart, rowEnd, row, col, curAdr;

if (end < start) {

printf("Invalid Range!\n");

return FALSE;

}

if (!(checkRange(start) && checkRange(end))) {

printf("Invalid Range!\n");

return FALSE;

}

rowStart = start / 16;

rowEnd = end / 16;

for (row = rowStart; row <= rowEnd; row++) {

for (col = 0; col < 16; col++) {

curAdr = row \* 16 + col;

//change curAdr memory to value if curAdr is not out of bound

if (curAdr >= start && curAdr <= end)

memory[curAdr] = value;

}

}

return TRUE;

}

//resets entire memory

void cmd\_reset() {

memset(memory, 0, sizeof(char)\*MEM\_SIZE);

}

int hashSearch\_opcode(char\* mnem) {

//Convert lowercase to uppercase

LowerToUpper(mnem);

int index = hashfunction(mnem);

hptr temp = optable[index];

//search untill the end of one optable's index

while (temp != NULL) {

//if the mnemonic that i'm looking for is found

if (!strcmp(mnem, temp->mnem))

return temp->opcode;

//otherwise continue to the next node

temp = temp->next;

}

return -1;

}

char\* hashSearch\_format(char\* mnem) {

//Convert lowercase to uppercase

LowerToUpper(mnem);

int index = hashfunction(mnem);

hptr temp = optable[index];

//search untill the end of one optable's index

while (temp != NULL) {

//if the mnemonic that i'm looking for is found

if (!strcmp(mnem, temp->mnem))

return temp->format;

//otherwise continue to the next node

temp = temp->next;

}

return NULL;

}

int addressingMode(char\* str){

if(str[0] == '#')

return 1;

else if(str[0] == '@')

return 2;

else

return 3;

}

void run\_opcodes(int addr){

int mode, format, relative, value;

unsigned int disp;

int opcode;

int memoryValue = -1;

opcode = bitToHex(addr,0,7);

opcode -= bitToHex(addr,4,7) % 4;

//format 2

switch(opcode){

int reg1, reg2;

//COMPR FORMAT 2

case 0xA0:

reg1 = registers[bitToHex(addr,8,11)];

reg2 = registers[bitToHex(addr,12,15)];

compareReg(reg1,reg2);

registers[registerNum("PC")] += 2;

return;

//CLEAR FORMAT 2

case 0xB4:

registers[bitToHex(addr,8,11)] = 0;

registers[registerNum("PC")] += 2;

return;

//TIXR FORMAT2

case 0xB8:

registers[registerNum("X")]++;

reg1 = registers[registerNum("X")];

reg2 = registers[bitToHex(addr,8,11)];

compareReg(reg1,reg2);

registers[registerNum("PC")] += 2;

return;

}

mode = bitAddressMode(addr);

format = bitFormat4(addr);

//format 4

if(format){

registers[registerNum("PC")] += 4;

disp = bitToHex(addr,12,31);

}

//format 3

else{

registers[registerNum("PC")] += 3;

disp = bitToHex(addr,12,23);

//if negative

if(disp & 0x800)

disp = disp | 0xFFFFF000;

relative = bitToHex(addr,8,11);

//indexed addressing

if(relative & 8)

disp += registers[registerNum("X")];

//base relative

if(relative & 4)

relative = registers[registerNum("B")];

//pc relative

else if(relative & 2)

relative = registers[registerNum("PC")];

//direct addresing

else

relative = 0;

disp += relative;

//indirect addressing

if(mode == INDIRECT){

//retrieve addr to visit from the addr stored

disp = bitToHex(disp,0,23);

}

}

switch(opcode){

//J

case 0x3C:

registers[registerNum("PC")]= disp;

return;

//JSUB

case 0x48:

registers[registerNum("L")] = registers[registerNum("PC")];

registers[registerNum("PC")] = disp;

return;

//JLT >

case 0x38:

if(registers[registerNum("SW")] == 1){

registers[registerNum("PC")] = disp;

}

return;

//JEQ =

case 0x30:

if (registers[registerNum("SW")] == 0)

registers[registerNum("PC")] = disp;

return;

//STA

case 0x0C:

writeToMem(disp,registerNum("A"));

return;

//STX

case 0x10:

writeToMem(disp,registerNum("X"));

return;

//STL

case 0x14:

writeToMem(disp,registerNum("L"));

return;

//STCH

case 0x54:

memory[disp] = registers[registerNum(("A"))] & 0xFF;

return;

//RSUB

case 0x4C:

registers[registerNum("PC")] = registers[registerNum("L")];

return;

//LDCH

case 0x50:

disp = bitToHex(disp,0,7);

storeLastByte(registerNum("A"),disp);

return;

}

//simple addressing

if(mode == SIMPLE){

disp = bitToHex(disp,0,23);

}

switch(opcode){

//LDA

case 0x00:

registers[registerNum("A")] = disp;

return;

//LDB

case 0x68:

registers[registerNum("B")] = disp;

return;

//LDT

case 0x74:

registers[registerNum("T")] = disp;

return;

//COMP with A

case 0x28:

if(registers[registerNum("A")] > disp)

registers[registerNum("SW")] = -1;

else if(registers[registerNum("A")] == disp)

registers[registerNum("SW")] = 0;

else

registers[registerNum("SW")] = 1;

return;

//TD 1 if device is ready 0 if not

case 0xE0:

registers[registerNum("SW")] = 1;

return;

//RD set rightmost byte of A with content

case 0xD8:

storeLastByte(registerNum("A"),0);

return;

//WD

case 0xDC:

return;

}

return;

}

------------------------------------------------------------

ds.h

------------------------------------------------------------

#ifndef \_DS\_C\_

#define \_DS\_C\_

#define HASH\_SIZE 20

#define SYM\_SIZE 26

typedef struct interm\* intermptr;

typedef struct interm {

int addr;

int format;

int argCount;

char operation[30];

char operand[20];

char operand2[20];

char label[30];

char line[200];

intermptr next;

}interm;

typedef struct linkedlist\* lptr;

typedef struct linkedlist {

char command[100];

lptr next;

}linkedlist;

void linkedlist\_push(lptr\*, char\*);

void linkedlist\_print(lptr);

typedef struct symtable\* sptr;

typedef struct symtable {

char label[30];

unsigned int addr;

sptr next;

}symtable;

typedef struct hashlist\* hptr;

typedef struct hashlist {

char mnem[20];

int opcode;

char format[20];

hptr next;

}hashlist;

void hashMain(char\*);

int hashfunction(char\*);

void hashlist\_push(hptr\*, char\*, int, char\*);

void hashlist\_printAll(hptr\*);

typedef struct symtab\* symptr;

typedef struct symtab {

char label[30];

int addr;

symptr next;

}symtab;

int symfunction(char\*);

int symtab\_push(symtab\*\*,char\*,int);

void symtab\_print(symptr);

void symtab\_printAll();

int symtab\_search(symtab\*,char\*);

void interm\_push(intermptr\*, intermptr);

//if it's a control section, legnth is positive integer. otherwise negative

typedef struct extsymtab\* extsymptr;

typedef struct extsymtab {

char label[30];

int addr;

int length;

extsymptr next;

}extsymtab;

void extsymtab\_push(extsymtab\*\* head, char\* symbol, int addr, int length);

extsymptr extsymtab\_search(extsymtab\* head, char\* symbol);

void extsymtab\_printAll(extsymtab\*);

typedef struct bplist\* bpptr;

typedef struct bplist {

int addr;

bpptr next;

}bplist;

void bplist\_push(int addr);

void bplist\_printAll();

int bplist\_search(int);

void bplist\_clear();

#endif

------------------------------------------------------------

ds.c

------------------------------------------------------------

#include "ds.h"

#include "20151619.h"

#include <string.h>

#include <stdlib.h>

void extsymtab\_push(extsymtab\*\* head, char\* symbol, int addr, int length){

extsymtab\* temp;

extsymtab\* newNode = (extsymtab\*)malloc(sizeof(extsymtab));

strcpy(newNode->label,symbol);

newNode->addr = addr;

newNode->length = length;

newNode->next = NULL;

if(\*head == NULL)

\*head = newNode;

else{

temp = \*head;

while(temp->next != NULL)

temp = temp->next;

temp->next = newNode;

}

}

extsymptr extsymtab\_search(extsymtab\* head, char\* symbol){

if (head == NULL)

return NULL;

else{

while(head != NULL){

if (!strcmp(head->label,symbol))

return head;

head = head->next;

}

}

return NULL;

}

void extsymtab\_printAll(extsymtab\* head){

printf("control symbol address length\nsection name\n");

printf("------------------------------------------------------\n");

while(head != NULL){

if(head->length < 0){

printf("\t\t%-7s\t\t",head->label);

printf("%04X\n",head->addr);

}

else{

printf("%-7s\t\t\t\t",head->label);

printf("%04X\t\t",head->addr);

printf("%04X\n",head->length);

}

head = head -> next;

}

}

void bplist\_push(int addr){

bpptr temp;

bpptr newNode = (bpptr)malloc(sizeof(bplist));

newNode->next = NULL;

newNode->addr = addr;

if (breakpoints == NULL){

breakpoints = newNode;

return;

}

else if(breakpoints->addr > addr){

newNode->next = breakpoints;

breakpoints = newNode;

return;

}

else if(breakpoints->next == NULL){

breakpoints->next = newNode;

return;

}

else{

temp = breakpoints;

while(temp->next->addr < addr){

if(temp->next->next == NULL){

temp->next->next = newNode;

return;

}

temp = temp -> next;

}

newNode->next = temp->next;

temp->next = newNode;

}

}

int bplist\_search(int addr){

bpptr temp = breakpoints;

while( temp != NULL ){

if(temp->addr == addr)

return TRUE;

temp = temp->next;

}

return FALSE;

}

void bplist\_printAll(){

bpptr temp = breakpoints;

if(breakpoints == NULL){

printf("No breakpoints set.\n");

return;

}

printf("breakpoints\n-----------\n");

while(temp != NULL){

printf("%X\n",temp->addr);

temp = temp->next;

}

}

void bplist\_clear(){

if (breakpoints == NULL)

return;

bpptr temp = breakpoints;

bpptr temp2 = temp->next;

while(temp2 != NULL){

free(temp);

temp = temp2;

temp2 = temp2->next;

}

breakpoints = NULL;

}

//appends new node with command to the end of head linked list

void linkedlist\_push(lptr\* head,char\* command){

lptr temp = \*head;

lptr newNode = (lptr)malloc(sizeof(linkedlist));

newNode->next = NULL;

strcpy(newNode->command,command);

if(temp != NULL){

//goes to the last node

while(temp->next != NULL)

temp = temp->next;

temp->next = newNode;

}

else{

\*head = newNode;

}

return;

}

//prints all the nodes in the linked list my moving one node at a time

void linkedlist\_print(lptr head){

int count = 1;

lptr temp = head;

while(temp != NULL){

printf("%-4d %s",count++,temp->command);

temp = temp->next;

}

return;

}

//main function to create initial hashtable

void hashMain(char\* fname){

FILE \*fp = fopen(fname, "r");

int opcode,index;

char mnem[30], format[30];

if (fp != NULL){

while(fscanf(fp,"%X %s %s",&opcode,mnem,format) != EOF){

//finds the index for mnemonic to go in the hashtable

index = hashfunction(mnem);

//appends that mnemonic to the table

hashlist\_push(&(optable[index]),mnem,opcode,format);

}

}

else{

printf("Cannot open File\n!");

}

}

//returns index number for the mnemonic to go into in the hashtable

int hashfunction(char\* str){

int i,sum=0;

for(i=0; i<(int)strlen(str); i++)

sum += (int)str[i];

return sum % HASH\_SIZE;

}

//creates a new node with mnemonic, opcode and format. Then inserts the new node the end of head list

void hashlist\_push(hptr \*head,char\* mnem,int opcode, char\* format){

hptr temp = \*head;

//make new node

hptr newNode = (hptr)malloc(sizeof(hashlist));

newNode->next = NULL;

strcpy(newNode->mnem,mnem);

strcpy(newNode->format,format);

newNode->opcode = opcode;

if(\*head != NULL){

//go to the end of head

while(temp->next != NULL)

temp = temp->next;

//insert to new node

temp->next = newNode;

}else{

\*head = newNode;

}

return;

}

//prints whole hashtable

void hashlist\_printAll(hptr \*head){

int i, isFirst;

hptr temp;

for(i=0; i<HASH\_SIZE; i++){

temp = head[i];

isFirst = TRUE;

//if an index is not empty

if(temp != NULL){

printf("%d : ",i);

//iterate through the list

while(temp != NULL){

if(!isFirst)

printf(" -> ");

printf("[%s, %X]",temp->mnem,temp->opcode);

temp = temp->next;

isFirst = FALSE;

}

printf("\n");

}

}

}

int symtab\_push(symtab \*\*head,char\* label,int addr){

symtab \*temp;

symtab \*prev;

//make new node

symtab\* newNode = (symtab\*)malloc(sizeof(symtab));

strcpy(newNode->label,label);

newNode->addr = addr;

newNode->next = NULL;

if(\*head != NULL){

//head에 삽입하는 경우

if (strcmp((\*head)->label, label) < 0){

newNode->next = \*head;

\*head = newNode;

return FALSE;

}

else{

temp = (\*head)->next;

prev = \*head;

}

//Search for the right place to go

while(temp != NULL){

//duplicate symbol

if(!strcmp(temp->label,label))

return TRUE;

if(strcmp(temp->label,label) < 0){

prev->next = newNode;

newNode ->next = temp;

return FALSE;

}

prev = temp;

temp = temp->next;

}

//insert to new node

prev->next = newNode;

}else{

\*head = newNode;

}

return FALSE;

}

int symtab\_search(symtab\* head,char\* label){

while(head != NULL){

if(!strcmp(head->label,label))

return head->addr;

head = head->next;

}

return -1;

}

void symtab\_printAll(){

for(int i=SYM\_SIZE-1;i>=0;i--)

symtab\_print(symboltable[i]);

}

void symtab\_print(symtab \*head){

while(head != NULL){

printf("\t%s %X\n",head->label,head->addr);

head = head->next;

}

}

int symfunction(char\* str){

int index = (int)str[0] - (int)'A';

if(index < 0)

return 0;

else

return index;

}

void interm\_push(intermptr \*head,intermptr newNode){

intermptr temp = \*head;

if(\*head != NULL){

//go to the end of head

while(temp->next != NULL)

temp = temp->next;

//insert to new node

temp->next = newNode;

}else{

\*head = newNode;

}

return;

}

------------------------------------------------------------

util.h

------------------------------------------------------------

#ifndef \_UTIL\_H\_

#define \_UTIL\_H\_

//Checking Functions

int checkRange(int);

int compareString(char\*, char\*, char\*);

int isDec(char\*);

int isHex(char\*);

int inRange(int,int,int);

int HexBitCount(int);

int checkComma(char\*);

int checkHex(char\*);

//Parsing

int asmSeparater(char\*, char\*, char\*, char\*, char\*);

char\* extractContent(char\*);

void extractStr(char\* dest, char\* source, int start, int len);

int extractStrToHex(char\* source, int start, int len);

//Data manipulation

void LowerToUpper(char\*);

int StrToInt(char\*);

int StrToHex(char\*);

int insertHexAt(int,int,int);

void charArrHexCal(unsigned char\* arr, int value, int len, char operation);

//BIT functions

int bitFormat4(int addr);

int bitToHex(int addr,int start,int end);

int bitAddressMode(int addr);

//Register functions

int registerNum(char\*);

void compareReg(int a, int b);

void printReg();

void writeToMem(int addr, int regNum);

int getLastByte(int regNum);

void storeLastByte(int regNum, int value);

#endif

------------------------------------------------------------

util.c

------------------------------------------------------------

#include "util.h"

#include "20151619.h"

#include "math.h"

//compares command with shortcommand and longcommand and see if command matches with one of them

int compareString(char\* command, char\* shortcommand, char\* longcommand) {

if (!strcmp(command, shortcommand) || (longcommand != NULL && !strcmp(command, longcommand)))

return TRUE;

else

return FALSE;

}

//check if adr is within the valid range

int checkRange(int adr) {

if (adr < 0 || adr >= 0x100000)

return FALSE;

return TRUE;

}

//check if str has any non-hex char

int checkHex(char\* str){

for(int i=0; i<(int)strlen(str); i++){

if((str[i] >=48 && str[i] <=57) || (str[i] >= 65 && str[i] <=70) || (str[i] >= 97 && str[i] <= 102) || str[i] == ',')

continue ;

else if(str[i] == 'X' || str[i] == 'x'){

if(i != 0)

if(str[i-1] == '0')

continue;

}

else{

return FALSE;

}

}

return TRUE;

}

//if argCount < 0 , you need to check operand2

int asmSeparater(char\* str, char\* label, char\* operation, char\* operand, char\* operand2){

int argCount = sscanf(str,"%s %s %s %s",label,operation,operand,operand2);

//cases with , such as LDCH BUFFER, X

if(!checkComma(operation)){

argCount \*= -1;

//if not all 4 inputs come in, move back

if(argCount != -4){

operation[(int)strlen(operation)-1] = '\0';

strcpy(operand2,operand);

strcpy(operand,operation);

strcpy(operation,label);

memset(label,0,sizeof label);

}

}

else if(!checkComma(operand)){

argCount \*= -1;

operand[(int)strlen(operand)-1] = '\0';

}

//printf("[%d] ",argCount);

switch(argCount){

case 1:

strcpy(operation,label);

memset(label,0,sizeof label);

break;

case 2:

strcpy(operand,operation);

strcpy(operation,label);

memset(label,0,sizeof label);

break;

}

return argCount;

}

//changes word to uppercase

void LowerToUpper(char\* word) {

for (int i = 0; i < (int)strlen(word); i++)

if ((int)word[i] > 96 && (int)word[i] < 123)

word[i] -= 32;

}

//check if command finishes with a comma

int checkComma(char\* command){

int len = (int)strlen(command);

if(command[len-1] == ',')

return FALSE;

return TRUE;

}

int StrToHex(char\* str){

unsigned int num;

int cnt = sscanf(str,"%x",&num);

if(cnt == 0)

return -1;

return num;

}

int StrToInt(char\* str){

unsigned int num;

int cnt = sscanf(str,"%d",&num);

if(cnt == 0)

return -1;

return num;

}

char\* extractContent(char\* str){

char \*temp = (char\*)malloc(30);

strncpy(temp,&(str[2]),strlen(str)-3);

temp[strlen(str)-3] = 0;

return temp;

}

int isDec(char\* str){

int len = (int)strlen(str);

for(int i=0;i<len;i++)

if(!inRange((int)'0',(int)'9',str[i]))

return FALSE;

return TRUE;

}

int isHex(char\* str){

int len = (int)strlen(str);

for(int i=0;i<len;i++)

if(str[i] == '\n')

str[i] = '\0';

len = (int)strlen(str);

if(isDec(str) == TRUE)

return TRUE;

for(int i=0;i<len;i++)

if(inRange((int)'a',(int)'f',str[i]) || inRange((int)'A',(int)'F',str[i]))

return TRUE;

return FALSE;

}

int inRange(int min,int max,int value){

if(value >= min && value <=max)

return TRUE;

return FALSE;

}

int HexBitCount(int num){

int digit = 0;

while(num >=16){

num /= 16;

digit++;

}

return digit;

}

int HexByteCount(int num){

return (int)(ceil((float)num/2));

}

int insertHexAt(int orig, int value, int index){

for(int i=0;i<index;i++)

value \*= 16;

return orig + value;

}

int registerNum(char\* reg){

if(!strcmp(reg,"A"))

return 0;

else if(!strcmp(reg,"X"))

return 1;

else if(!strcmp(reg,"L"))

return 2;

else if(!strcmp(reg,"PC"))

return 8;

else if(!strcmp(reg,"SW"))

return 9;

else if(!strcmp(reg,"B"))

return 3;

else if(!strcmp(reg,"S"))

return 4;

else if(!strcmp(reg,"T"))

return 5;

else if(!strcmp(reg,"F"))

return 6;

else

return 7;

}

void extractStr(char\* dest, char\* source, int start, int len){

strncpy(dest,&(source[start]),len);

dest[len] = '\0';

for(int i=0;i<len;i++)

if (dest[i] == '\n' || dest[i] == ' ')

dest[i] = '\0';

}

int extractStrToHex(char\* source, int start, int len){

char dest[300];

strncpy(dest,&(source[start]),len);

dest[len] = '\0';

for(int i=0;i<len;i++)

if (dest[i] == '\n' || dest[i] == ' ')

dest[i] = '\0';

return StrToHex(dest);

}

void charArrHexCal(unsigned char\* arr, int value, int len, char operation){

int converted = 0;

int multiplier = 1;

int temp, rem = 0;

int repeat = ceil((float)len / 2.0);

unsigned int result;

for(int i = repeat-1; i>=0; i--){

temp = arr[i];

//last 4 bit

converted += (temp % 16)\*multiplier;

temp /= 16;

multiplier \*= 16;

if(len % 2 != 0 && i == 0){

rem = temp \* 16;

continue;

}

//first 4 bit

converted += (temp % 16)\*multiplier;

temp /= 16;

multiplier \*= 16;

}

if(operation == '+')

converted += value;

else

converted -= value;

result = (unsigned int)converted;

for(int i=repeat-1; i>=0; i--){

arr[i] = result & 255;

result /= 256;

}

arr[0] +=rem;

}

int bitAddressMode(int addr){

return memory[addr] & 3;

}

int bitFormat4(int addr){

addr += 1;

if(memory[addr] & 0x10)

return TRUE;

else

return FALSE;

}

int bitToHex(int addr,int start,int end){

int i=0,j=1, temp=1;

int result = 0;

temp = 0x100;

while(1){

for(; i<4\*j; i++){

temp = temp >> 1;

if(i<start)

continue;

else if(i>=end){

if(memory[addr] & temp)

result += 1;

/\*

while(i++ < 8\*j -1){

result \*= 2;

}

\*/

return result;

}

else

if(memory[addr] & temp)

result += 1;

result \*= 2;

}

if(j % 2 == 0){

temp = 0x100;

addr++;

}

j++;

}

return result;

}

void compareReg(int a, int b){

if(a > b)

registers[registerNum("SW")] = -1;

else if(a == b)

registers[registerNum("SW")] = 0;

else

registers[registerNum("SW")] = 1;

}

void printReg(){

printf("A : %06X X : %06X\n",registers[0],registers[1]);

printf("L : %06X PC: %06X\n",registers[2],registers[8]);

printf("B : %06X S : %06X\n",registers[3],registers[4]);

printf("T : %06X\n",registers[5]);

}

void writeToMem(int addr, int regNum){

int temp = registers[regNum];

int mask = 0xFF;

for(int i=2;i>=0;i--){

memory[addr+i] = temp & mask;

temp = temp >> 8;

}

}

int getLastByte(int regNum){

return registers[regNum] & 0xFF;

}

void storeLastByte(int regNum, int value){

registers[regNum] = (registers[regNum] & 0xFFFF00) + value;

}