과목 명: 시스템프로그래밍

담당 교수 명: 박 운 상

<<Project 2>>

**서강대학교 컴퓨터학과**

**[20161634]**

**[장수길]**

목 차

1. **프로그램 개요**
2. **프로그램 설명**

**2.1** 프로그램 흐름도

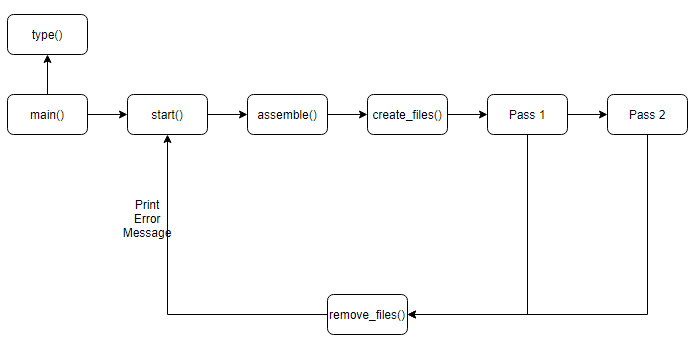
1. **구조체 및 헤더 정의**
2. **모듈 정의**
   1. 모듈 이름: int assemble(const char \*filename, struct NODE \*\* head)
      1. 기능
      2. 사용 변수
   2. 모듈 이름: struct INST myparser(const char \*line, int line\_number)
      1. 기능
      2. 사용 변수
   3. 모듈 이름: copy\_strings(struct INST \*target, const char \*label, const char \*opcode, const char \*oper, const char \*idx)
      1. 기능
      2. 사용 변수
   4. 모듈 이름: void remove\_files(const char \*filename)
      1. 기능
      2. 사용 변수
   5. 모듈 이름: void create\_files(const char \*filename)
      1. 기능
      2. 사용 변수
   6. 모듈 이름: void symtab\_add(const char \*label, int locctr, struct NODE \*\*head)
      1. 기능
      2. 사용 변수
   7. 모듈 이름: bool symtab\_exist(const char \*label, struct NODE \*\*head)
      1. 기능
      2. 사용 변수
   8. 모듈 이름: int symtab\_retval(const char \*label, struct NODE \*\*head)
      1. 기능
      2. 사용 변수
   9. 모듈 이름: bool onlynumber(const char \*s)
      1. 기능
      2. 사용 변수
   10. 모듈 이름: void appendString(char \*target, int xbpe, int temp, int mode, bool indexed)
       1. 기능
       2. 사용 변수
   11. 모듈 이름: void symbol\_clear(struct NODE \*\*head)
       1. 기능
       2. 사용 변수
   12. 모듈 이름: LSTfprinter(int line\_number, int LOCCTR, const char \*label, const char \*opcode, const char \*oper, const char \*idx, bool index\_exists)
       1. 기능
       2. 사용 변수
   13. 모듈 이름: void file\_ext\_appender(char\*filename, const char\*extension)
       1. 기능
       2. 사용 변수
   14. 모듈 이름: type\_filename(const char \*filename, struct NODE \*\*head)
       1. 기능
       2. 사용 변수
   15. 모듈 이름: void symtab\_print(struct NODE \*\*head)
       1. 기능
       2. 사용 변수
3. **전역 변수 정의**
4. **코드**
   1. **defines.h**
   2. **20161634.h**
   3. **20161634.c**
   4. **mem\_cmd.h**
   5. **mem\_cmd.c**
   6. **shell\_cmd.h**
   7. **shell\_cmd.c**
   8. **opcode.h**
   9. **opcode.c**
   10. **assembler.c**
   11. **assembler.h**
5. **프로그램 개요**

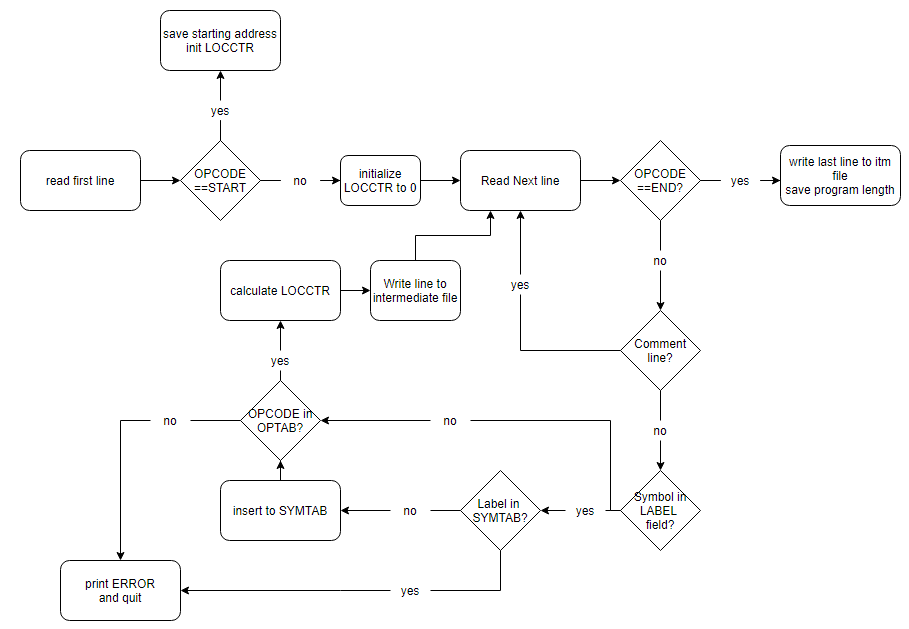
SIC/XE 머신 구현을 위한 두번째 단계이다.

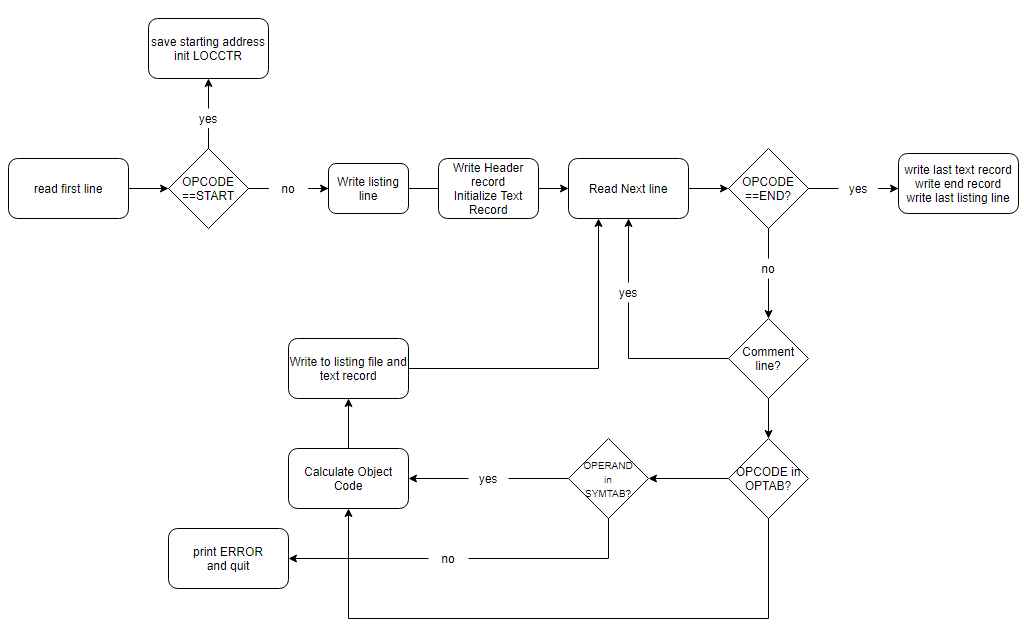
몇가지 명령어를 추가 구현하고  
주어진 .asm파일을 기반으로 object code를 생성하는 assembler를 구현한다.

1. **프로그램 설명**
   1. **프로그램 흐름도**

이번 프로젝트의 주요 흐름도와, Two-pass assembler의 단순화된 흐름도를 첨부하였다.

  
<그림 1: 전체적인 구조도>

  
<그림 2: Pass 1의 흐름도>

  
<그림 3: Pass 2 의 흐름도>

1. **구조체 및 헤더 정의**

이번 프로젝트 2에 새롭게 추가된 구조체는 다음과 같다.  
헤더는 추가되지 않았다.

|  |  |
| --- | --- |
| struct INST | 파일의 한 줄을 tokenize 하고 그 줄의 format과 label, opcode, operand, 그리고 index를 저장한다. |

1. **모듈 정의**
   1. **모듈 이름 int assemble(const char\*filename, struct NODE \*\*head)**
      1. *기능*이번 프로젝트에서 가장 중요한 부분을 담당하는 함수이다.  
         주어진 .asm 파일을 읽어내서 listing file 과 object file을 생성해준다.   
         기본적으로 two-pass어셈블러의 알고리즘을 사용하여 구현한다.  
         먼저 pass1에서는 myparser함수를 이용해 asm파일의 레이블과 opcode, operand 등을 읽어오고, symbol table을 채워넣음과 동시에 다시 읽기 좋은 상태로 intermediate file에 저장해준다.  
         Pass 2에서는 intermediate file을 다시 읽어들이면서 object code를 계산하고, listing file을 작성함과 동시에 object file의 내용에 해당하는 text record, header record, modification record 등을 작성해준다.  
         Object Code를 계산하는 과정은 먼저 PC relative를 적용하고, 나타낼 수 없는 범위인 경우 Base를 시도한 뒤, Base또한 불가능한 경우는 4형식의 format으로 나타내준다..
      2. *사용 변수*

|  |  |
| --- | --- |
| int line\_number | 현재 line number를 저장한다. |
| char line[120] | 현재 줄을 저장한다 |
| struct INST cur | 현재 줄의 tokenize된 정보를 저장한다 |
| int LOCCTR | 현재 location counter를 저장한다 |
| int start\_adr | 프로그램의 시작 주소를 저장한다 |
| int program\_length | 프로그램의 길이를 저장한다 |
| int modif\_record\_num | modification record의 개수를 저장한다 |
| int operand\_address | 현재줄의 operand address를 저장한다 |
| int BASE | base relative address를 위한 base 값을 저장한다 |
| int recordlen | record의 길이를 저장해준다 |
| char record[100] | record를 저장시켜준다 |
| int xbpe | x,b,p,e로 이루어진 1byte의 값을 저장한다 |
| char OBJ\_str | 해당 줄의 object code를 저장한다 |
| bool immediate | addressing 모드를 저장해준다 |
| bool indirect | addressing 모드를 저장해준다 |
| bool extended | 해당줄의 명령 format을 저장해준다 |
| bool index\_exists | index가 존재하는 명령인지 여부를 저장한다 |
| int format | 해당줄의 format을 저장한다 |

* 1. **모듈 이름 struct INSTt myparser(const char \*line, int line\_number)**
     1. *기능*파일에서 한 줄을 읽어오고, 어셈블리 파일의 형식을 고려하여 적절히 tokenize 해준 결과를 반환한다. 여러 개의 정보를 반환해야 하기 때문에 반환 타입으로 구조체를 선택하였으며, tokenize하는 과정에서 instruction의 format과 validity를 검증한다.   
        sscanf함수를 이용하여 몇 개의 토큰이 들어와있는 지 확인하고, 정보를 기반으로 어떤 타입의 명령어인지 확인해주고, 적절한 값을 적절한 문자 배열에 넣어준다.
     2. *사용 변수*

|  |  |
| --- | --- |
| char label[30] | 해당줄의 label을 저장한다 |
| char opcode[30] | 해당줄의 opcode를 저장한다 |
| char idx[30] | 해당줄의 index를 저장한다 |
| char operand[30] | 해당줄의 operand를 저장한다 |
| int words | 해당줄의 토큰 개수를 저장한다 |
| struct INST cur | 해당줄의 정보를 저장해서 반환한다 |

* 1. **모듈 이름 void copy\_strings(struct INST \*target, const char \*label, const char \*opcode, const char \*oper, const char \*idx)**
     1. *기능*특정한 구조체에 인자로 받은 문자열들을 복사시켜준다
     2. *사용 변수*없음
  2. **모듈 이름 void create\_files(const char \*filename)**
     1. *기능*파일이름을 받아 .obj, .lst, .itm 파일을 생성해준다
     2. *사용 변수*char fname[100] : 파일의 이름을 임시로 저장해준다
  3. **모듈 이름 void remove\_files(const char \*filename)**
     1. *기능*파일이름을 받아 .obj, .lst, .itm 파일을 삭제시켜준다
     2. *사용 변수*char fname[100]: 파일의 이름을 임시로 저장해준다.
  4. **모듈 이름 void symtab\_add(const char \*label, int locctr, struct NODE \*\*head)**
     1. *기능*symbol table에 주어진 label과 value를 받아서 추가해준다.  
        이때 사전순으로 출력한다는 symbol 명령어를 고려하여 사전순으로 linked list에 저장시켜준다.  
        head가 null이거나 새로운 노드가 헤드 노드의 정보보다 사전순으로 더 앞에 있을 경우 헤드와 노드의 위치를 바꿔주면서 추가해주고, 그렇지 않을 경우 사전순으로 적절한 위치를 찾아서 노드를 링크드 리스트에 추가해준다.
     2. *사용 변수*없음
  5. **모듈 이름 bool symtab\_exist(const char \*label, struct NODE \*\*head)**
     1. *기능*symbol table을 탐색하여 주어진 label이 존재하는지 확인해준다.  
        단순히 symbol table을 선형탐색하여 label과 일치할 경우 반환해준다
     2. *사용 변수*없음
  6. **모듈 이름 int symtab\_retval(const char \*label, struct NODE \*\*head)**
     1. *기능*symbol table을 탐색하여 주어진 label에 해당하는 value를 반환시켜준다
     2. *사용 변수*없음
  7. **모듈 이름 bool onlynumber(const char \*s)**
     1. *기능*주어진 문자열이 숫자로만 이루어져있는지 확인한다
     2. *사용 변수*
  8. **모듈 이름 int is\_register(const char \*operand)**
     1. *기능*주어진 operand가 레지스터인지 확인한다  
        A, X, L , B, S, T, F, PC, SW 중 같은 값이 존재하는지 확인시켜주고, 해당하는 반환값을 돌려준다.
     2. *사용 변수*
  9. **모듈 이름 void appendString(char \*target, int xbpe, int temp, int mode, bool indexed)**
     1. *기능*xbpe값과 instruction mode, 그리고 index의 유무를 인자로 받아 적절한 형태의 opcode값을 target string에 추가해준다.
     2. *사용 변수*

|  |  |
| --- | --- |
| char xbpe\_str[5] | xbpe값을 문자열에 추가한다 |
| char OPADR\_str[10] | operand address를 문자열에 추가한다 |

* 1. **모듈 이름 void symbol\_clear(struct NODE \*\*head)**
     1. *기능*symbol table에 있는 모든 symbol들을 제거해준다.
     2. *사용 변수*
  2. **모듈 이름 void LSTprinter(int line\_number, int LOCCTR, const char \*label, const char \*opcode, const char \*oper, const char \*idx, bool index\_exists)**
     1. *기능*listing file에 출력 포맷을 맞추고 object code 또한 추가하여 출력시켜준다.
     2. *사용 변수*없음
  3. **모듈 이름 file\_ext\_appender(char \*filename, const char \*extension)**
     1. *기능*파일의 확장자를 변경시켜준다.
     2. *사용 변수*없음
  4. **모듈 이름 void type\_filename(const char \*filename, struct NODE \*\*head)**
     1. *기능*주어진 파일의 내용을 모두 출력시켜준다
     2. *사용 변수*없음
  5. **모듈 이름 void symtab\_print(struct NODE \*\*head)**
     1. *기능*마지막으로 assemble이 성공한 symbol table을 모두 출력시켜준다.
     2. *사용 변수*없음

1. **전역변수 정의**
   1. struct NODE \*symbol
      1. symbol table을 저장하는 linked list의 헤드 노드를 저장한다.
   2. struct NODE \*last\_success\_symtab
      1. 마지막으로 assemble 성공한 파일의 symbol table linked list의 헤드 노드를 저장한다.
2. **코드**
   1. **defines.h**

#ifndef \_\_CHECK\_\_

#define \_\_CHECK\_\_

#include <stdarg.h>

#include <stdio.h>

#include <string.h>

#include <stdbool.h>

#include <dirent.h>

#include <sys/stat.h>

#include <ctype.h>

#include <stdlib.h>

#define MAX\_HASH 20

struct NODE{

    struct NODE \*next;

    char format[10];

    char cmd[100];

    int num;

};

#endif

* 1. **20161634.c**

#include "20161634.h"

#include "shell\_cmd.h"

#include "mem\_cmd.h"

#include "opcode.h"

#include "defines.h"

#include "assembler.h"

//head of linked list for history cmd

extern struct NODE \*head;

extern struct NODE \*symbol;

extern struct NODE \*last\_success\_symtab;

//hash table pointer

extern struct NODE \*hash[MAX\_HASH];

//works as 1MB virtual memory 1byte \* 2^20

unsigned char m[2<<20];

int dump\_adr=0;

int main(void){

    head=NULL;

    //resets virtual memory with value=

    memset(m, 0, sizeof(m));

    //reads from opcode.txt and initiates hash table

    init\_hash(hash);

    while(1){

        start();

    }

    return 0;

}

/\*---------------------------------------------------------------------\*/

/\*Function: hex\*/

/\*Purpose: reads a hexadecimal string and returns positive decimal value\*/

/\*Return value: if invalid -1 / if valid positive decimal value\*/

/\*---------------------------------------------------------------------\*/

int hex(const char\* num){

    if(strlen(num)==0) return -1;

    for(int i=0;i<strlen(num);i++){

        if(!((num[i]>='0' && num[i]<='9') || (num[i]>='a' && num[i]<='f') || (num[i]>='A' && num[i]<='F'))){

            return -1;

        }

    }

    int ret;

    sscanf(num, "%x", &ret);

    return ret;

}

/\*---------------------------------------------------------------------\*/

/\*Function: argnum\*/

/\*Purpose: parses multi-word commands and calls appropriate functions depending on input\*/

/\*Return value: if invalid -1 / if valid 0 \*/

/\*---------------------------------------------------------------------\*/

int argnum(char\* cmd){

    char what[100];

    //variables for start end value address

    int s,e,val, adr;

    //variable to check arguments

    int args=1;

    //checks validity of input

    int space=0, comma=0;

    //sets flag according to command

    bool editflag=false, dumpflag=false, fillflag=false;

    bool spaceflag=false;

    s=e=val=adr=-1;

    //string to add to history list

    char history\_string[100];

    char op[100], mne[100], trash[100]; trash[0]='\0';

    int words=sscanf(cmd, "%s %s %s", op, mne, trash);

    //TWO WORDED INPUTS

    if(words==2 && trash[0]=='\0'){

        if(!strcmp(op, "opcode")){

            //ex: opcode add

            op\_cmd(mne, hash, &head); return 0;

        }

        if(!strcmp(op, "type")){

            type\_filename(mne, &head); return 0;

        }

        if(!strcmp(op, "assemble")){

            int ret=assemble(mne, &head);

            if(ret!=-1){

                sprintf(history\_string, "assemble %s", mne);

                history\_add(history\_string, &head);

            }

            return 0;

        }

    }

    //checks comma numbers

    for(int i=0;i<strlen(cmd);i++){

        if(!spaceflag && isspace(cmd[i])){

            space=i; spaceflag=true;

        }

        if(cmd[i]==',') comma++;

    }

    sscanf(cmd, "%s", what);

    char \*tokenize=cmd;

    tokenize+=space;

    char \*ptr;

    //decides command flags

    if(!strcmp("e", what)||!strcmp("edit", what)) editflag=true;

    if(!strcmp("du", what)||!strcmp("dump", what)) dumpflag=true;

    if(!strcmp("f", what)||!strcmp("fill", what)) fillflag=true;

    char hexStart[10], hexEnd[10],hexAddress[10], hexValue[10];

    //splits command by comma, and does boundary check and argument checks

    while((ptr=strsep(&tokenize,","))){

        int tempVal=hex(trim(ptr));

        //if invalid hexadecimal error

        if(tempVal==-1) return -1;

        if(editflag){

            //edit : check second argument range 00~FF

            //edit : first argument should be 00~FFFFF

            if(args==1 && 0<=tempVal && tempVal<=0xFFFFF){

                adr=tempVal; args++;

                strcpy(hexAddress, trim(ptr));

            }

            else if(args==2 && 0<=tempVal && tempVal<=0xFF){

                val=tempVal; args++;

                strcpy(hexValue, trim(ptr));

            }

            else return -1;

        }

        else if(dumpflag){

            //dump : all arguments should be 00~FFFFF

            if(0<=tempVal && tempVal<=0xFFFFF){

                if(args==1){

                    s=tempVal; args++;

                    strcpy(hexStart, trim(ptr));

                }

                else if(args==2){

                    e=tempVal; args++;

                    strcpy(hexEnd, trim(ptr));

                }

            }

            //if else invalid input ex: du 123A

            else return -1;

        }

        else if(fillflag){

            //fill : first and second argument (00~FFFFF) third argument (00~FF)

            //printf("tempval:%d\n", tempVal);

            if(args==1 && 0<=tempVal && tempVal<=0xFFFFF){

                s=tempVal; args++;

                strcpy(hexStart, trim(ptr));

            }

            else if(args==2 && 0<=tempVal && tempVal<=0xFFFFF){

                e=tempVal; args++;

                strcpy(hexEnd, trim(ptr));

            }

            else if(args==3 && 0<=tempVal && tempVal<=0xFF){

                val=tempVal; args++;

                strcpy(hexValue, trim(ptr));

            }

            else return -1;

        }

    }

    //command should be checked

    //all s,e,val should never be -1

    //comma number and argument number should match

    if(comma+2!=args) return -1;

    //call functions based on command and append to history list

    if(args==2 && (s!=-1) && (!strcmp(what, "du")||!strcmp(what,"dump")) ){

        sprintf(history\_string, "%s %s", what, hexStart);

        history\_add(history\_string, &head);

        dump(m,1,s); return 0;

    }

    if(args==3 && (s!=-1 && e!=-1) &&  (!strcmp(what, "du")||!strcmp(what,"dump"))){

        sprintf(history\_string, "%s %s, %s", what, hexStart,hexEnd);

        history\_add(history\_string, &head);

        dump(m,2,s,e); return 0;

    }

    if(args==4 && (s!=-1 && e!=-1 && val!=-1) && (!strcmp(what, "f")||!strcmp(what,"fill"))){

        sprintf(history\_string, "%s %s, %s, %s", what, hexStart,hexEnd,hexValue);

        history\_add(history\_string, &head);

        fill(m,s,e,val); return 0;

    }

    if(editflag && args==3 && (adr!=-1 && val!=-1)){

        sprintf(history\_string, "%s %s, %s", what, hexAddress, hexValue);

        history\_add(history\_string, &head);

        edit(m,adr, val); return 0;

    }

    return -1;

}

/\*---------------------------------------------------------------------\*/

/\*Function: start\*/

/\*Purpose: reads user input and handles single word inputs\*/

/\*Return: void\*/

/\*---------------------------------------------------------------------\*/

void start(void){

    printf("sicsim>");

    char buffer[MAX\_INPUT];

    if(fgets(buffer, sizeof(buffer), stdin)){

        size\_t len=strlen(buffer);

        if(len>0 && buffer[len-1]=='\n') buffer[len-1]='\0';

    }

    char \*cmd=trim(buffer);

    if(strlen(cmd)==0) return;

    //one word inputs (trivial) - call appropriate functions

    if(!strcmp(cmd, "help") || !strcmp(cmd, "h")){

        shell\_help();

        history\_add(cmd, &head); return;

    }

    if(!strcmp(cmd, "dir") || !strcmp(cmd, "d")){

        shell\_dir();

        history\_add(cmd, &head); return;

    }

    if(!strcmp(cmd, "quit") || !strcmp(cmd, "q")){

        exit(0);

    }

    if(!strcmp(cmd, "history") || !strcmp(cmd, "hi")){

        history\_add(cmd, &head);

        history\_print(&head); return;

    }

    if(!strcmp(cmd, "du")||!strcmp(cmd,"dump")){

        dump(m,1, dump\_adr); dump\_adr+=160;

        history\_add(cmd, &head); return;

    }

    if(!strcmp(cmd, "opcodelist")){

        history\_add(cmd, &head);

        op\_list(hash); return;

    }

    if(!strcmp(cmd, "reset")){

        history\_add(cmd, &head);

        reset(m); return;

    }

    if(!strcmp(cmd, "symbol")){

        history\_add(cmd, &head);

        symtab\_print(&last\_success\_symtab); return;

    }

    //memory commands with multi word inputs

    if(argnum(cmd)==-1){

        printf("invalid input\n"); return;

    }

}

**20161634.h**

#pragma once

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_INPUT 100

void start(void);

int argnum(char\* cmd);

int hex(const char\* num);

**assembler.c**

#include "assembler.h"

#include "opcode.h"

#include <inttypes.h>

#define MAX\_FILENAME 100

#define ERROR -1

#define MEMSET(string) memset(string, '\0', sizeof(string));

//head of linked list for history cmd

struct NODE \*head;

//head of linked list for last success symbol table

struct NODE \*last\_success\_symtab;

//hash table pointer

struct NODE \*hash[MAX\_HASH];

//head of linked list for current symbol table

struct NODE \*symbol;

FILE \*lst, \*obj, \*itm;

/\*---------------------------------------------------------------------\*/

/\*Function: file\_ext\_appender\*/

/\*Purpose: Get a file name and change the extension format with parameter\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void file\_ext\_appender(char\* filename, const char\* extension) {

    int flen = strlen(filename);

    for (int i = 0; i < 3; i++) {

        filename[flen - 1 - i] = '\0';

    }

    strcat(filename, extension);

}

/\*---------------------------------------------------------------------\*/

/\*Function: remove\_files\*/

/\*Purpose: remove filename.obj, filename.lst, filename.itm\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void remove\_files(const char \*filename){

    char fname[MAX\_FILENAME];

    int rem=0;

    strcpy(fname, filename);

    file\_ext\_appender(fname, "lst");

    rem=remove(fname);

    if(rem==-1) printf("deletion failed\n");

    file\_ext\_appender(fname, "obj");

    rem=remove(fname);

    if(rem==-1) printf("deletion failed\n");

    file\_ext\_appender(fname, "itm");

    rem=remove(fname);

    if(rem==-1) printf("deletion failed\n");

}

/\*---------------------------------------------------------------------\*/

/\*Function: creates\_files\*/

/\*Purpose: creates filename.obj, filename.lst, filename.itm\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void create\_files(const char \*filename){

    //make files

    char fname[MAX\_FILENAME];

    strcpy(fname, filename);

    file\_ext\_appender(fname, "lst");

    lst=fopen(fname, "w+");

    file\_ext\_appender(fname, "obj");

    obj=fopen(fname, "w+");

    file\_ext\_appender(fname, "itm");

    itm=fopen(fname, "w+");

}

/\*---------------------------------------------------------------------\*/

/\*Function: symbol\_clear\*/

/\*Purpose: removes symbol table\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void symbol\_clear(struct NODE \*\*head){

    struct NODE \*cur=(\*head);

    struct NODE\* next;

    while(cur!=NULL){

        next=cur->next;

        free(cur);

        cur=next;

    }

    \*head=NULL;

}

/\*---------------------------------------------------------------------\*/

/\*Function: symtab\_retval\*/

/\*Purpose: retrieves value from corresponding symbol from symbol table\*/

/\*Return: symbol data if exists, -1 if doesn't exists\*/

/\*---------------------------------------------------------------------\*/

int symtab\_retval(const char \*label, struct NODE \*\*head){

    for(struct NODE \*cur=(\*head);cur!=NULL;cur=cur->next){

        if(!strcmp(label, cur->cmd)){

            return cur->num;

        }

    }

    return -1;

}

/\*---------------------------------------------------------------------\*/

/\*Function: symtab\_exist\*/

/\*Purpose: determine whether label exists in symbol table\*/

/\*Return: true if exists, false if doesn't exists\*/

/\*---------------------------------------------------------------------\*/

bool symtab\_exist(const char \*label, struct NODE \*\*head){

    for(struct NODE \*cur=(\*head);cur!=NULL;cur=cur->next){

        if(!strcmp(label, cur->cmd)){

            return true;

        }

    }

    return false;

}

/\*---------------------------------------------------------------------\*/

/\*Function: symtab\_add\*/

/\*Purpose: add data to symbol table\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void symtab\_add(const char \*label, int locctr, struct NODE \*\*head){

    struct NODE \*newNode=(struct NODE\*)malloc(sizeof(struct NODE));

    strcpy(newNode->cmd, label);

    newNode->num=locctr;

    newNode->next=NULL;

    struct NODE \*cur=(\*head);

    if(\*head==NULL || strcmp((\*head)->cmd, newNode->cmd)>=0){

        newNode->next=(\*head);

        \*head=newNode;

    }

    else{

        while(cur->next!=NULL && strcmp(cur->next->cmd,newNode->cmd)<0){

            cur=cur->next;

        }

        newNode->next=cur->next;

        cur->next=newNode;

    }

}

/\*---------------------------------------------------------------------\*/

/\*Function: is\_register\*/

/\*Purpose: checks if given string is a register\*/

/\*Return: corresponding register number, -1 if not a register\*/

/\*---------------------------------------------------------------------\*/

int is\_register(const char \*operand){

    if(!strcmp(operand,"A")) return 0;

    if(!strcmp(operand,"X")) return 1;

    if(!strcmp(operand,"L")) return 2;

    if(!strcmp(operand,"B")) return 3;

    if(!strcmp(operand,"S")) return 4;

    if(!strcmp(operand,"T")) return 5;

    if(!strcmp(operand,"F")) return 6;

    if(!strcmp(operand,"PC")) return 8;

    if(!strcmp(operand,"SW")) return 9;

    return -1;

}

/\*---------------------------------------------------------------------\*/

/\*Function: appendString\*/

/\*Purpose: Makes object code and prints into corresponding files\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void appendString(char \*target, int xbpe, int temp, int mode, bool indexed){

    char xbpe\_str[5];

    char OPADR\_str[10];

    char t[10];

    sprintf(xbpe\_str, "%X", xbpe);

    if(mode==0){

        strcat(target, xbpe\_str);

        if(temp>=0){

        sprintf(OPADR\_str, "%03X", temp);

        strcat(target, OPADR\_str);

        }

        else{

            sprintf(t, "%X", temp);

            int tlen=strlen(t);

            for(int i=0;i<4;i++){

                t[i]=t[i+tlen-3];

            }

            strcat(target, t);

        }

    }

    if(mode==1){

        if(temp>=0){

            if(!indexed) sprintf(OPADR\_str, "1%05X", temp);

            else sprintf(OPADR\_str, "9%05X", temp);

            strcat(target, OPADR\_str);

        }

        else{

            sprintf(t, "%X", temp);

            int tlen=strlen(t);

            for(int i=0;i<6;i++){

                t[i]=t[i+tlen-5];

            }

            strcat(target, t);

        }

    }

}

/\*---------------------------------------------------------------------\*/

/\*Function: LSTprinter\*/

/\*Purpose: prints to console with format (for debugging)\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void LSTprinter(int line\_number, int LOCCTR, const char \*label, const char \*opcode, const char \*oper, const char\*idx, bool index\_exists){

        printf("%d\t", line\_number);

        if(LOCCTR!=-1) printf("%04X\t", LOCCTR);

        else printf("\t");

        if(label[0]!='\*') printf("%s\t", label);

        else printf("\t");

        if(opcode[0]!='\*') printf("%s\t", opcode);

        else printf("\t");

        if(index\_exists){

            //BUFFER, X

            if(strlen(oper)+strlen(idx)>5) printf("%s, %s ", oper, idx);

            else printf("%s, %s\t", oper, idx);

        }

        else{

            if(oper[0]!='\*') printf("%s\t", oper);

            else printf("\t");

        }

        printf("\t");

}

/\*---------------------------------------------------------------------\*/

/\*Function: LSTfprinter\*/

/\*Purpose: prints to listing file with format\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void LSTfprinter(FILE \*fp, int line\_number, int LOCCTR, const char \*label, const char \*opcode, const char \*oper, const char\*idx, bool index\_exists){

        fprintf(fp, "%d\t", line\_number);

        if(LOCCTR!=-1) fprintf(fp, "%04X\t", LOCCTR);

        else fprintf(fp, "\t");

        if(label[0]!='\*') fprintf(fp, "%s\t", label);

        else fprintf(fp, "\t");

        if(opcode[0]!='\*') fprintf(fp, "%s\t", opcode);

        else fprintf(fp, "\t");

        if(index\_exists){

            //BUFFER, X

            if(strlen(oper)+strlen(idx)>5) fprintf(fp, "%s, %s ", oper, idx);

            else fprintf(fp, "%s, %s\t", oper, idx);

        }

        else{

            if(oper[0]!='\*') fprintf(fp, "%s\t", oper);

            else fprintf(fp, "\t");

        }

        fprintf(fp, "\t");

}

/\*---------------------------------------------------------------------\*/

/\*Function: getINSTtype\*/

/\*Purpose: gets instruction type\*/

/\*Return: instruction type, and -1 if invalid\*/

/\*---------------------------------------------------------------------\*/

int getINSTtype(const char \*label, const char \*opcode, const char \*oper, const char\*idx){

    if(label[0]!='\*' && opcode[0]!='\*' && oper[0]!='\*' && idx[0]!='\*' ) return 1;

    if(label[0]=='\*' && opcode[0]!='\*' && oper[0]!='\*' && idx[0]!='\*' ) return 2;

    if(label[0]!='\*' && opcode[0]!='\*' && oper[0]!='\*' && idx[0]=='\*' ) return 3;

    if(label[0]=='\*' && opcode[0]!='\*' && oper[0]!='\*' && idx[0]=='\*' ) return 4;

    if(label[0]!='\*' && opcode[0]!='\*' && oper[0]=='\*' && idx[0]=='\*' ) return 5;

    if(label[0]=='\*' && opcode[0]!='\*' && oper[0]=='\*' && idx[0]=='\*' ) return 6;

    return -1;

}

/\*---------------------------------------------------------------------\*/

/\*Function: onlynumber\*/

/\*Purpose: checks if string contains only digits\*/

/\*Return: true/false\*/

/\*---------------------------------------------------------------------\*/

bool onlynumber(const char \*s){

    for(int i=0;i<strlen(s);i++){

        if(!(s[i]>='0' && s[i]<='9')) return false;

    }

    return true;

}

/\*---------------------------------------------------------------------\*/

/\*Function: assemble\*/

/\*Purpose: executes assembling process\*/

/\*Return: -1 if error occurs, 1 if successful\*/

/\*---------------------------------------------------------------------\*/

int assemble(const char \*filename, struct NODE \*\*head){

    FILE \*fp;

    int len=strlen(filename);

    //if no file

    if(!(fp=fopen(filename,"r"))){

        printf("no file\n");

        return -1;

    }

    //if invalid extension

    if(!(filename[len-1]=='m' && filename[len-2]=='s' && filename[len-3]=='a' && filename[len-4]=='.')){

        fclose(fp);

        printf("invalid extension\n");

         return -1;

    }

    int line\_number=5;      //current line number

    char line[120];         //current line string

    struct INST cur;        //tokenized data of current line

    cur.comment=false;

    lst=obj=NULL;

    bool ERR\_FLAG=false;    //ERROR FLAG

    int LOCCTR=0;           //Location counter of current line

    int start\_adr=0;        //Starting address of program

    int program\_length=0;   //Program length

    int num;                //multipurpose

    char hexa[2];           //to convert digits to hexadecimal

    int modif\_record\_num=0; //number of modification records

    int \*modif;             //modification records dynamically allocated

    create\_files(filename);

    //========================================PASS 1========================================

    //read first input line

    if(fgets(line, sizeof(line), fp)!=NULL){

        cur=myparser(line, line\_number);

        if(!strcmp(cur.opcode, "START")){

            //SAVE  #[OPERAND] AS STARTING ADDRESS

            sscanf(cur.oper, "%X", &start\_adr);

            LOCCTR=start\_adr;

            fprintf(itm, "%d\t%04X\t%s\t%s\t%s \*\n", line\_number,LOCCTR,  cur.label, cur.opcode, cur.oper);

            //INITIALIZE LOCCTR TO STARTING ADDRESS

        }

        else{

            //INITIALIZE LOCCTR TO 0

            LOCCTR=0;

        }

        line\_number+=5;

    }

    while(fgets(line, sizeof(line), fp)!=NULL){

        int dx=0;

        cur=myparser(line, line\_number);

        if(!strcmp(cur.opcode,"END")){

            fprintf(itm,"%d\t", line\_number);

            fprintf(itm, "-1\t\*\t%s\t%s \*\n", cur.opcode, cur.oper);

            break;

        }

        //if this is not a comment line

        if(!cur.comment){

            //if there is a symbol in the label field

            if(cur.label[0]!='\0'){

                if(symtab\_exist(cur.label, &symbol)){

                    //set error flag

                    ERR\_FLAG=true;

                    printf("Error at line %d : duplicate symbol\n", line\_number);

                    remove\_files(filename);

                    symbol\_clear(&symbol);

                    return -1;

                }

                else{

                    symtab\_add(cur.label, LOCCTR, &symbol);

                }

            }

            sscanf(cur.oper, "%d", &num);

            //if opcode is found in optab

            if(op\_return(cur.opcode, hash)!=-1){

                if(cur.format==4) dx+=4;

                else{

                    dx+=op\_format(cur.opcode,hash);

                }

            }

            else if(!strcmp(cur.opcode,"WORD")){

                //if OPCODE is WORD add 3 to LOCCTR

                dx+=3;

            }

            else if(!strcmp(cur.opcode,"RESW")){

                //if OPCODE is RESW add 3 \* #[OPERAND] to LOCCTR

                dx+=3\*num;

            }

            else if(!strcmp(cur.opcode, "RESB")){

                //if OPCODE is RESB add #[OPERAND] to LOCCTR

                dx+=num;

            }

            else if(!strcmp(cur.opcode, "BYTE")){

                //if OPCODE is BYTE find length of constant in bytes and add to LOCCTR

                if(cur.oper[0]=='C'){

                    dx+=(int)strlen(cur.oper)-3;

                }

                else if(cur.oper[0]=='X'){

                    dx+=(int)(strlen(cur.oper)-3)/2;

                }

            }

            else{

                //if OPCODE is not BASE, return ERROR

                if(strcmp(cur.opcode, "BASE")){

                    printf("Error at line %d : invalid opcode\n", line\_number);

                    remove\_files(filename);

                    symbol\_clear(&symbol);

                    return -1;

                }

                //if invalid format, return ERROR

                if(cur.type==-1){

                    ERR\_FLAG=true;

                    printf("Error at line %d : invalid format instruction\n", line\_number);

                    remove\_files(filename);

                    symbol\_clear(&symbol);

                    return -1;

                }

            }

            //if format 4

            if(cur.format==4){

                int templen=strlen(cur.opcode);

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

                    cur.opcode[i]=cur.opcode[i-1];

                }

                cur.opcode[0]='+';

                modif\_record\_num++;

            }

            //print to intermediate file in a more accessible format

            fprintf(itm, "%d\t", line\_number);

            if(!strcmp(cur.opcode, "BASE")) fprintf(itm, "-1\t");

            else fprintf(itm, "%04X\t", LOCCTR);

            if(cur.label[0]=='\0') fprintf(itm, "\*\t");

            else fprintf(itm, "%s\t", cur.label);

            if(cur.opcode[0]=='\0') fprintf(itm, "\*\t");

            else fprintf(itm, "%s\t", cur.opcode);

            if(cur.oper[0]=='\0') fprintf(itm, "\*\t");

            else fprintf(itm, "%s\t", cur.oper);

            if(cur.idx[0]=='\0') fprintf(itm, "\*\n");

            else fprintf(itm, "%s\n", cur.idx);

            LOCCTR+=dx;

        }

        else{

            fprintf(itm, "%s", line);

        }

        line\_number+=5;

    }

    program\_length=LOCCTR-start\_adr;

    //===================================END OF PASS 1========================================

    //REWIND FILE POINTER BACK

    rewind(itm);

    int operand\_address=0;      //operand address

    int BASE=0;                 //BASE value for base relative instructions

    int recordlen=0;            //length of record

    bool newline=false;         //for RESB, RESW instructions which makes newline in obj files

    char record[100];           //for record in obj file

    char appendOBJ[100];        //for appending

    char append[10];            //for appending

    //Set modification dynamic array

    modif=(int\*)malloc(sizeof(int)\*modif\_record\_num);

    int modif\_index=0;

    int first\_executable=-1;

    int record\_global=0;

    //Reset record associated char arrays

    memset(record, '\0', sizeof(record));

    memset(append,'\0', sizeof(append));

    record[0]='T';

    //===================================PASS 2 START=========================================

    if(fgets(line, sizeof(line), itm)!=NULL){

        sscanf(line, "%d %d %s %s %s %s", &line\_number, &LOCCTR, cur.label, cur.opcode, cur.oper, cur.idx);

        if(!strcmp(cur.opcode, "START")){

            fprintf(lst, "%d\t%04X\t%s\t%s\t%s\t\n", line\_number, LOCCTR, cur.label, cur.opcode, cur.oper);

        }

        fprintf(obj, "H%s %06X%06X\n", cur.label, start\_adr, program\_length);

    }

    while(fgets(line, sizeof(line), itm)!=NULL){

        sscanf(line, "%d %X %s %s %s %s", &line\_number, &LOCCTR, cur.label, cur.opcode, cur.oper, cur.idx);

        int xbpe;                   //xbpe value

        int OP\_val;                 //Opcode value

        char OBJ\_str[100];          //Object code

        ERR\_FLAG=ERR\_FLAG;

        bool immediate=false;       //checks immediate

        bool indirect=false;        //checks indirect

        int temp;

        bool extended=false;        //checks extended

        bool index\_exists=false;    //checks index

        int format=getINSTtype(cur.label, cur.opcode, cur.oper, cur.idx);

        if(format==1 || format==2) index\_exists=true;

        if(line[0]=='.'){

            fprintf(lst, "%d\t\t%s", line\_number, line);

            continue;

        }

        LSTfprinter(lst, line\_number, LOCCTR, cur.label, cur.opcode, cur.oper, cur.idx, index\_exists);

        //search OPTAB for OPCODE

        if(cur.opcode[0]=='+'){

            temp=strlen(cur.opcode); extended=true;

            for(int i=0;i<temp;i++) cur.opcode[i]=cur.opcode[i+1];

        }

        //CHECKED FOR FORMAT 4 OPERATIONS

        //if opcode == BASE, set base value

        if(!strcmp(cur.opcode, "BASE")){

            BASE=symtab\_retval(cur.oper, &symbol);

        }

        if(op\_return(cur.opcode, hash)!=-1){

            //if there is a symbol in OPERAND field

            if(cur.oper[0]!='\*'){

                if(format==1 || format==2){

                    //IF INDEX EXISTS, REMOVE COMMA AT BACK

                    index\_exists=true;

                    temp=strlen(cur.oper);

                }

                //if immediate addressing

                if(cur.oper[0]=='#'){

                    temp=strlen(cur.oper); immediate=true;

                    for(int i=0;i<temp;i++) cur.oper[i]=cur.oper[i+1];

                    //+LDT #4096

                    if(onlynumber(cur.oper)) sscanf(cur.oper, "%d", &operand\_address);

                    //LDB #LENGTH

                    else{

                        if(symtab\_exist(cur.oper, &symbol)){

                            operand\_address=symtab\_retval(cur.oper, &symbol);

                        }

                    }

                }

                //if indirect addressing

                else if(cur.oper[0]=='@'){

                    temp=strlen(cur.oper); indirect=true;

                    for(int i=0;i<temp;i++) cur.oper[i]=cur.oper[i+1];

                }

                //IF NORMAL OPCODE STL RETADR

                if(symtab\_exist(cur.oper, &symbol)){

                    operand\_address=symtab\_retval(cur.oper, &symbol);

                }

                //IF IMMEDIATE CONSTANT : LDA #3

                else if(onlynumber(cur.oper)){

                    sscanf(cur.oper,"%d", &operand\_address);

                }

                //IF REGISTER ex : CLEAR X

                else if(is\_register(cur.oper)!=-1){

                    operand\_address=is\_register(cur.oper);

                }

                else{

                    operand\_address=0;

                    printf("Error at line %d : undefined symbol\n", line\_number);

                    remove\_files(filename);

                    symbol\_clear(&symbol);

                    return -1;

                }

                //ASSEMBLE OBJECT CODE INSTRUCTION

                //if operand is a register

                OP\_val=op\_return(cur.opcode, hash);

                if(is\_register(cur.oper)!=-1){

                    sprintf(OBJ\_str, "%02X", OP\_val);

                    int first=is\_register(cur.oper);

                    int second=is\_register(cur.idx);

                    if(second==-1) second=0;

                    char f[2], s[2];

                    f[0]=first+'0'; s[0]=second+'0'; s[1]=f[1]='\0';

                    strcat(OBJ\_str, f);

                    strcat(OBJ\_str, s);

                    line\_number+=5;

                    fprintf(lst, "%s\n", OBJ\_str);

                    goto TEXTRECORD;

                }

                else if(immediate){

                    sprintf(OBJ\_str, "%02X", OP\_val+1);

                }

                else if(indirect){

                    sprintf(OBJ\_str, "%02X", OP\_val+2);

                }

                else{

                    sprintf(OBJ\_str, "%02X", OP\_val+3);

                }

                //CALCULATE DISPLACEMENT

                //TRY PC FIRST

                temp=operand\_address-LOCCTR-3;

                if(immediate && onlynumber(cur.oper)){

                    sscanf(cur.oper, "%d", &temp);

                }

                if(extended){

                    if(!onlynumber(cur.oper) && !immediate && !indirect){

                        modif[modif\_index++]=record\_global+1;

                    }

                    appendString(OBJ\_str, xbpe, operand\_address, 1, index\_exists);

                }

                else if(-2048<=temp && temp<=2047){

                    //calculate displacement PC relative

                    xbpe=0;

                    if(extended) xbpe+=1;

                    if(index\_exists) xbpe+=8;

                    xbpe+=2;

                    if(onlynumber(cur.oper)) xbpe=0;

                    appendString(OBJ\_str, xbpe, temp,0, index\_exists);

                }

                else{

                    //TRY BASE

                    temp=operand\_address-BASE;

                    if(immediate && onlynumber(cur.oper)){

                        sscanf(cur.oper, "%d", &temp);

                    }

                    if(0<=temp && temp<=4095){

                        xbpe=0;

                        if(index\_exists)xbpe+=8;

                        xbpe+=4;

                        appendString(OBJ\_str, xbpe, temp,0, index\_exists);

                    }

                    //if nothing works, print error

                    else{

                        printf("Error at line %d: unreachable address\n", line\_number);

                        remove\_files(filename);

                    }

                }

            }

            else{

                OP\_val=op\_return(cur.opcode, hash);

                sprintf(OBJ\_str, "%X0000", OP\_val+3);

            }

        }

        //if OPCODE = BYTE

        else if(!strcmp(cur.opcode, "BYTE")){

            memset(OBJ\_str, '\0', sizeof(OBJ\_str));

            //calculates constant length in bytes

            if(cur.oper[0]=='C'){

                for(int i=2;i<strlen(cur.oper)-1;i++){

                    sprintf(hexa, "%02X", (int)cur.oper[i]);

                    strcat(OBJ\_str, hexa);

                }

            }

            else if(cur.oper[0]=='X'){

                for(int i=2;i<strlen(cur.oper)-1;i++){

                    OBJ\_str[i-2]=cur.oper[i];

                }

                OBJ\_str[strlen(cur.oper)-3]='\0';

            }

        }

        else if(!strcmp(cur.opcode, "WORD")){

            //generate object code with WORD directive

            memset(OBJ\_str, '\0', sizeof(OBJ\_str));

            int word\_directive;

            sscanf(cur.oper, "%d", &word\_directive);

            sprintf(OBJ\_str, "%06d", word\_directive);

        }

        else{

            memset(OBJ\_str, '\0', sizeof(OBJ\_str));

        }

        fprintf(lst, "%s\n", OBJ\_str);

        line\_number+=5;

        //Generates OBJECT CODE to write to .obj file, and write necessary newlines(RESW, RESB or full column)

        TEXTRECORD:

        if(first\_executable==-1) first\_executable=LOCCTR;

        //if record is full, or opcode is RESB or RESW, write a newline

        if(recordlen+strlen(OBJ\_str)/2>=30 || ((!strcmp(cur.opcode, "RESB") || !strcmp(cur.opcode, "RESW")))){

            if(!newline){

                sprintf(append, "%02X", recordlen);

                strcat(record, append);

                strcat(record, appendOBJ);

                fprintf(obj, "%s\n", record);

                memset(append, '\0', sizeof(append));

                memset(record, '\0', sizeof(record));

                memset(appendOBJ, '\0', sizeof(appendOBJ));

                recordlen=0;

                record[0]='T';

            }

            else{

                memset(append, '\0', sizeof(append));

                memset(record, '\0', sizeof(record));

                memset(appendOBJ, '\0', sizeof(appendOBJ));

                record[0]='T';

                continue;

            }

        }

        //Initialize new record with current LOCCTR

        if(recordlen==0){

            if(LOCCTR!=-1){

            sprintf(append, "%06X", LOCCTR);

            strcat(record, append);

            }

        }

        //Append OBJECT STRINGS to record

        if(recordlen+strlen(OBJ\_str)/2<30){

            strcat(appendOBJ, OBJ\_str);

            record\_global+=strlen(OBJ\_str)/2;

            recordlen+=strlen(OBJ\_str)/2;

        }

        //If RESB OR RESW, indicate newline

        if(!strcmp(cur.opcode, "RESB") || !strcmp(cur.opcode, "RESW"))  newline=true;

        else newline=false;

        //if opcode is END, write remaining object codes and write Modification records and End record

        if(!strcmp(cur.opcode, "END")){

            if(strlen(record)>6){

                sprintf(append, "%02X", recordlen);

                strcat(record, append);

                strcat(record, appendOBJ);

                fprintf(obj, "%s\n", record);

            }

            for(int i=0;i<modif\_index;i++){

                fprintf(obj, "M%06X%02X\n", modif[i], 5);

            }

            memset(record, '\0', sizeof(record));

            sprintf(append,"%06X", first\_executable);

            record[0]='E';

            strcat(record, append);

            fprintf(obj, "%s\n", record);

            break;

        }

    }

    //===================================END OF PASS 2=========================================

    //Close files

    fclose(itm);

    fclose(lst);

    fclose(obj);

    //Assemble success, so move symbol table to last success

    last\_success\_symtab=symbol;

    symbol=NULL;

    //remove intermediate file

    char fname[MAX\_FILENAME];

    strcpy(fname, filename);

    file\_ext\_appender(fname, "itm");

    remove(fname);

    file\_ext\_appender(fname, "lst");

    printf("[%s],", fname);

    file\_ext\_appender(fname, "obj");

    printf(" [%s]\n", fname);

    return 1;

}

/\*---------------------------------------------------------------------\*/

/\*Function: myparser\*/

/\*Purpose: tokenizes given string\*/

/\*Return: a struct containing tokenized information\*/

/\*---------------------------------------------------------------------\*/

struct INST myparser(const char \*line, int line\_number){

    struct INST cur;

    cur.comment=false;

    cur.line\_number=line\_number;

    cur.format=-1;

    int words=0;

    char label[30], opcode[30], oper[30], idx[30], trash[30];

    //INITIALIZE VALUES

    cur.label[0]=cur.opcode[0]=cur.oper[0]=cur.idx[0]=cur.trash[0]='\0';

    label[0]=opcode[0]=oper[0]=idx[0]=trash[0]='\0';

    words=sscanf(line, "%s %s %s %s %s", label, opcode, oper, idx, trash);

    //IF + is in front of OPCODE format 4

    if(label[0]=='+'){

        for(int i=0;i<strlen(label);i++) label[i]=label[i+1]; cur.format=4;

    }

    if(opcode[0]=='+'){

        for(int i=0;i<strlen(opcode);i++) opcode[i]=opcode[i+1]; cur.format=4;

    }

    int oper\_len=strlen(oper), opcode\_len=strlen(opcode);

    //assembler directives without labels

    if(words==2 && !strcmp(label, "END")){

        cur.type=4;

        strcpy(oper, opcode);

        strcpy(opcode, label);

        label[0]='\0';

        copy\_strings(&cur, label, opcode, oper, idx);

        return cur;

    }

    if(words==2 && !strcmp(label, "BASE")){

        cur.type=4;

        strcpy(oper, opcode);

        strcpy(opcode, label);

        label[0]='\0';

        copy\_strings(&cur, label, opcode, oper, idx);

        return cur;

    }

    if(label[0]=='.'){

        //LINE IS COMMENT

        cur.comment=true;

        return cur;

    }

    if(words==4 && oper[oper\_len-1]==','){

        //LABEL OPCODE OPERAND, INDEX

        cur.type=1;

        oper[oper\_len-1]='\0';

        copy\_strings(&cur, label, opcode, oper, idx);

        return cur;

    }

    else if(words==3 && idx[0]=='\0'){

        if(opcode[opcode\_len-1]==','){

            //  OPCODE OPERAND, INDEX

            strcpy(idx, oper); strcpy(oper, opcode); strcpy(opcode, label);

            oper\_len=strlen(oper);

            oper[oper\_len-1]='\0';

            label[0]='\0';

            cur.type=2;

            copy\_strings(&cur, label, opcode, oper, idx);

            return cur;

        }

        else{

            // LABEL OPCODE OPERAND

            cur.type=3;

            copy\_strings(&cur, label, opcode, oper, idx);

            return cur;

        }

    }

    else if(words==2 && idx[0]=='\0' && oper[0]=='\0'){

        if(op\_return(label, hash)!=-1){

            //OPCODE OPERAND

            cur.type=4;

            strcpy(oper, opcode); strcpy(opcode, label);

            label[0]='\0';

        }

        else if(op\_return(opcode, hash)!=-1){

            //LABEL OPCODE

            cur.type=5;

        }

        else{

            //NO OPCODE PRESENT (set errorflag)

            cur.type=-1;

        }

        copy\_strings(&cur, label, opcode, oper, idx);

        return cur;

    }

    else if(words==1){

        cur.type=6;

        strcpy(opcode, label);

        label[0]='\0';

        copy\_strings(&cur, label, opcode, oper, idx);

        return cur;

    }

    else{

        //CALL DEBUGGER

        cur.type=-1;

        printf("ERROR AT LINE %d\n", line\_number);

        return cur;

    }

}

/\*---------------------------------------------------------------------\*/

/\*Function: copy\_strings\*/

/\*Purpose: copies from string to string (multiple)\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void copy\_strings(struct INST \*target, const char \*label, const char \*opcode, const char \*oper, const char \*idx){

    strcpy(target->label, label); strcpy(target->oper, oper);

    strcpy(target->opcode, opcode); strcpy(target->idx, idx);

}

**assembler.h**

#include "defines.h"

struct INST{

    bool comment;

    int line\_number;

    int type;

    int format;

    /\*

    LABEL OPCODE OPERAND, INDEX  - type 1

          OPCODE OPERAND, INDEX  - type 2

    LABEL OPCODE OPERAND         - type 3

          OPCODE OPERAND         - type 4

    LABEL OPCODE                 - type 5

          OPCODE                 - type 6

    \*/

    char trash[30];

    char label[30];

    char oper[30];

    char opcode[30];

    char idx[30];

};

void file\_ext\_appender(char\* filename, const char\* extension);

void LSTfprinter(FILE \*fp, int line\_number, int LOCCTR, const char \*label, const char \*opcode, const char \*oper, const char\*idx, bool index\_exists);

void LSTprinter(int line\_number, int LOCCTR, const char \*label, const char \*opcode, const char \*oper, const char\*idx, bool index\_exists);

void symbol\_clear(struct NODE \*\*head);

void appendString(char \*target, int xbpe, int temp, int mode, bool indexed);

int is\_register(const char \*operand);

bool onlynumber(const char \*s);

int symtab\_retval(const char \*label, struct NODE \*\*head);

bool symtab\_exist(const char \*label, struct NODE \*\*head);

void symtab\_add(const char \*label, int locctr, struct NODE \*\*head);

void create\_files( const char \*filename);

void remove\_files(const char \*filename);

void copy\_strings(struct INST \*target, const char \*label, const char \*opcode, const char \*oper, const char \*idx);

struct INST myparser(const char \*line, int line\_number);

int assemble(const char \*filename, struct NODE \*\*head);

**mem\_cmd.c**

#include "mem\_cmd.h"

/\*---------------------------------------------------------------------\*/

/\*Function: edit\*/

/\*Purpose: edits memory based on address and value\*/

/\*Return: void\*/

/\*---------------------------------------------------------------------\*/

void edit(unsigned char mem[], int address, int value){

    mem[address]=(char)value;

}

/\*---------------------------------------------------------------------\*/

/\*Function: reset\*/

/\*Purpose: resets whole memory to 0\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void reset(unsigned char mem[]){

    for(int i=0;i<=0xFFFFF;i++){

        mem[i]=0;

    }

}

/\*---------------------------------------------------------------------\*/

/\*Function: fill\*/

/\*Purpose: assigns value to memory inside given address range\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void fill(unsigned char mem[], int start, int end, int value){

    //range overflow exception

    if(start>end) {

        printf("invalid range\n");

        return;

    }

    for(int i=start;i<=end;i++){

        mem[i]=value;

    }

}

/\*---------------------------------------------------------------------\*/

/\*Function: print\*/

/\*Purpose: prints formatted contents of virtual memory\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void print(unsigned char mem[], int start, int end){

    //ex: du 15 57 (decimal)  -->  0 ~ 59

    int s=(start/16)\*16;

    int e=(end/16+1)\*16-1;

    //range overflow exception

    if(e>0xFFFFF) e=0xFFFFF;

    for(int i=s;i<=e;i+=16){

        //address | memory contents | ASCII CODE

        printf("%05X ", i);

        for(int j=i;j<i+16;j++){

            if(j<start || j>end) printf("   ");

            else{

                printf("%02X ", (int)mem[j]);

            }

        }

        printf("; ");

        for(int j=i;j<i+16;j++){

            if(j<start || j>end) printf(".");

            else if(mem[j]<0x20 || mem[j]>0x7E) printf(".");

            else printf("%c", mem[j]);

        }

        printf("\n");

    }

}

/\*---------------------------------------------------------------------\*/

/\*Function: dump\*/

/\*Purpose: variadic function - checks argument size and calls corresponding function\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void dump(unsigned char mem[], int args, ...){

    va\_list ap;

    va\_start(ap, args);

    int num;

    if(args==1){

        //if only dump or dump start

        int start,end;

        for(int i=0;i<args;i++){

            num=va\_arg(ap,int);

            if(i==0) start=num;

        }

        end=start+159;

        va\_end(ap);

        print(mem, start, end);

    }

    else if(args==2){

        //dump start end

        int start, end;

        for(int i=0;i<args;i++){

            num=va\_arg(ap, int);

            if(i==0) start=num;

            if(i==1) end=num;

        }

        va\_end(ap);

        //range exception when dump 4,3

        if(start>end){

            printf("range error!\n"); return;

        }

        print(mem, start, end);

    }

}

**mem\_cmd.h**

#include "defines.h"

void dump(unsigned char mem[], int args, ...);

void print(unsigned char mem[], int start, int end);

char \*intToHex(int n, int len);

void fill(unsigned char mem[], int start, int end, int value);

void edit(unsigned char mem[], int address, int value);

void reset(unsigned char mem[]);

**opcode.c**

#include "opcode.h"

#define MAX\_HASH 20

void history\_add(char\* command, struct NODE \*\*head);

/\*---------------------------------------------------------------------\*/

/\*Function: calc\_key\*/

/\*Purpose: calculates hash function\*/

/\*Return: integer value of hash key\*/

/\*---------------------------------------------------------------------\*/

int calc\_key(const char\*cmd){

    int key=0;

    for(int i=0;i<strlen(cmd);i++){

        key+=(int)cmd[i];

    }

    return key%MAX\_HASH;

}

/\*---------------------------------------------------------------------\*/

/\*Function: addHash\*/

/\*Purpose: adds a node to hash table\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void addHash(struct NODE \*node, struct NODE \*hash[]){

    int key=calc\_key(node->cmd);

    if(hash[key]==NULL){

        hash[key]=node;

    }

    else{

        node->next=hash[key];

        hash[key]=node;

    }

}

/\*---------------------------------------------------------------------\*/

/\*Function: init\_hash\*/

/\*Purpose: reads opcode.txt and makes hash table\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void init\_hash(struct NODE \*hash[]){

    FILE \*fp = fopen("opcode.txt", "r");

    int cmd\_num;

    char op[10];

    char format[10];

    while(EOF!=fscanf(fp,"%x %s %s", &cmd\_num, op, format)){

        struct NODE \*newNode=(struct NODE\*)malloc(sizeof(struct NODE));

        strcpy(newNode->cmd, op);

        strcpy(newNode->format, format);

        newNode->num=cmd\_num;

        newNode->next=NULL;

        addHash(newNode, hash);

    }

}

/\*---------------------------------------------------------------------\*/

/\*Function: op\_cmd\*/

/\*Purpose: takes mnemonic input and prints corresponding opcode\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void op\_cmd(char \*mnemonic, struct NODE \*hash[], struct NODE \*\*head){

    int key=calc\_key(mnemonic);

    char history\_string[100];

    struct NODE\*cur=hash[key];

    while(cur!=NULL){

        if(!strcmp(cur->cmd, mnemonic)){

            sprintf(history\_string,"opcode %s", mnemonic);

            history\_add(history\_string, head);

            printf("opcode is %X\n", cur->num);

            return;

        }

        cur=cur->next;

    }

    printf("no such opcode\n"); return;

}

/\*---------------------------------------------------------------------\*/

/\*Function: op\_ret\*/

/\*Purpose: get corresponding opcode\*/

/\*Return: opcode for mnemonic\*/

/\*---------------------------------------------------------------------\*/

int op\_return(const char\* mnemonic, struct NODE\* hash[]){

    int key=calc\_key(mnemonic);

    struct NODE\*cur=hash[key];

    while(cur!=NULL){

        if(!strcmp(cur->cmd, mnemonic)) return cur->num;

        cur=cur->next;

    }

    return -1;

}

int op\_format(const char\* mnemonic, struct NODE\* hash[]){

    int key=calc\_key(mnemonic);

    struct NODE\*cur=hash[key];

    while(cur!=NULL){

        if(!strcmp(cur->cmd, mnemonic)){

            if(!strcmp(cur->format, "1")) return 1;

            else if(!strcmp(cur->format, "2")) return 2;

            else if(!strcmp(cur->format, "3/4")) return 3;

        }

        cur=cur->next;

    }

    return -1;

}

/\*---------------------------------------------------------------------\*/

/\*Function: op\_list\*/

/\*Purpose: Prints all opcodes in format\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void op\_list(struct NODE \*hash[]){

    for(int i=0;i<MAX\_HASH;i++){

        printf("%2d : ", i);

        if(hash[i]!=NULL){

            struct NODE \*cur=hash[i];

            while(cur!=NULL){

                printf("[%s, %X] ", cur->cmd, cur->num);

                cur=cur->next;

                if(cur!=NULL){

                    printf(" -> ");

                }

            }

        }

        printf("\n");

    }

}

**opcode.h**

#include "defines.h"

int op\_format(const char\* mnemonic, struct NODE\* hash[]);

int op\_return(const char\* mnemonic, struct NODE\* hash[]);

void init\_hash(struct NODE\*hash[]);

void op\_cmd(char \*mnemonic, struct NODE\*hash[], struct NODE \*\*head);

void op\_list(struct NODE\*hash[]);

**shell\_cmd.c**

#include "shell\_cmd.h"

/\*---------------------------------------------------------------------\*/

/\*Function: trim\*/

/\*Purpose: removes leading and trailing spaces in string\*/

/\*Return: char pointer of trimmed string\*/

/\*---------------------------------------------------------------------\*/

char\* trim(char \*cmd){

    //trims leading and trailing spaces

    while(isspace(\*cmd))cmd++;

    if(\*cmd==0) return cmd;

    int end=strlen(cmd)-1;

    while(isspace(cmd[end])){

        end--;

    }

    cmd[end+1]='\0';

    return cmd;

}

/\*---------------------------------------------------------------------\*/

/\*Function: checkExe\*/

/\*Purpose: checks if given file name is executable\*/

/\*Return: true if executable, false if not executable \*/

/\*---------------------------------------------------------------------\*/

bool checkExe(const char \*name){

    struct stat sb;

    if (stat(name, &sb) == 0 && sb.st\_mode & S\_IXUSR){

        /\* executable \*/

        return true;

    }

    else return false;

}

/\*---------------------------------------------------------------------\*/

/\*Function: checkDir\*/

/\*Purpose: checks if given file name is directory\*/

/\*Return: true if directory / false if not directory \*/

/\*---------------------------------------------------------------------\*/

bool checkDir(const char\* name){

    //checks if given file name is a directory

    struct stat status;

    stat(name,&status);

    return S\_ISDIR(status.st\_mode);

}

/\*---------------------------------------------------------------------\*/

/\*Function: shell\_help\*/

/\*Purpose: Print available commands\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void shell\_help(void){

    printf("h[elp]\n");

    printf("d[ir]\n");

    printf("q[uit]\n");

    printf("du[mp] [start, end]\n");

    printf("e[dit] address, value\n");

    printf("f[ill] start, end, value\n");

    printf("reset\n");

    printf("opcode mnemonic\n");

    printf("opcodelist\n");

    printf("assemble filename\n");

    printf("type filename\n");

    printf("symbol\n");

}

/\*---------------------------------------------------------------------\*/

/\*Function: shell\_dir\*/

/\*Purpose: prints files/directories in current directory\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void shell\_dir(void){

    DIR \*dir\_i=opendir(".");

    struct dirent \*dir\_e;

    if(dir\_i!=NULL){

        while((dir\_e=readdir(dir\_i))){

            char filename[100];

            //copy file name

            strcpy(filename, dir\_e->d\_name);

            int len=strlen(filename);

            //check if execution file (.out)

            if(checkExe(dir\_e->d\_name)){

                filename[len]='\*';

                filename[len+1]='\0';

            }

            //check if directory

            if(checkDir(dir\_e->d\_name)){

                filename[len]='/';

                filename[len+1]='\0';

            }

            printf("%s\n", filename);

        }

    }

    closedir(dir\_i);

}

/\*---------------------------------------------------------------------\*/

/\*Function: history\_add\*/

/\*Purpose: adds command to history linked list\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void history\_add(char\* command, struct NODE \*\*head){

    struct NODE \*newNode=(struct NODE\*)malloc(sizeof(struct NODE));

    strcpy(newNode->cmd, command);

    newNode->next=NULL;

    struct NODE \*cur;

    if(\*head==NULL){

        \*head=newNode; return;

    }

    else{

        for(cur=(\*head);cur->next!=NULL;cur=cur->next){}

            cur->next=newNode;

    }

}

/\*---------------------------------------------------------------------\*/

/\*Function: history\_print\*/

/\*Purpose: prints contents of linked list \*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void history\_print(struct NODE \*\*head){

    if(\*head==NULL) return;

    //prints contents of linked list

    int cnt=1;

    for(struct NODE \*cur=(\*head);cur!=NULL;cur=cur->next, cnt++){

        printf("%d %s\n", cnt, cur->cmd);

    }

}

/\*---------------------------------------------------------------------\*/

/\*Function: type\_filename\*/

/\*Purpose: prints contents of file\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void type\_filename(const char \*filename, struct NODE \*\*head){

    if(checkDir(filename)) {

        printf("%s is a directory. Abort\n", filename); return;

    }

    FILE \*fp;

    if((fp=fopen(filename, "r"))){

        char content=fgetc(fp);

        while(content!=EOF){

            printf("%c", content);

            content=fgetc(fp);

        }

        printf("\n");

        fclose(fp);

        char history\_string[100];

        sprintf(history\_string, "type %s", filename);

        history\_add(history\_string, head);

        return;

    }

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

}

/\*---------------------------------------------------------------------\*/

/\*Function: symbol\_print\*/

/\*Purpose: prints symbol table in lexicographical order\*/

/\*Return: none\*/

/\*---------------------------------------------------------------------\*/

void symtab\_print(struct NODE \*\*head){

    if(\*head==NULL) return;

    for(struct NODE \*cur=(\*head);cur!=NULL;cur=cur->next){

        printf("\t%s\t%04X\n", cur->cmd, cur->num);

    }

}

**shell\_cmd.h**

#include "defines.h"

void symtab\_print(struct NODE \*\*head);

void history\_add(char command[], struct NODE \*\*head);

void history\_print(struct NODE \*\*head);

void shell\_dir(void);

void shell\_help(void);

void type\_filename(const char \*filename, struct NODE \*\*head);

char\* trim(char \*cmd);

**Makefile**

CC=gcc

CFLAGS=-g -Wall

OBJS=20161634.o shell\_cmd.o mem\_cmd.o opcode.o assembler.o

TARGET=20161634.out

$(TARGET): $(OBJS)

    $(CC) -o $@ $(OBJS)

20161634.o: 20161634.c shell\_cmd.h mem\_cmd.h opcode.h assembler.h defines.h

shell\_cmd.o: shell\_cmd.c shell\_cmd.h

mem\_cmd.o: mem\_cmd.c mem\_cmd.h

opcode.o: opcode.c opcode.h

assembler.o: assembler.c assembler.h

clean:

    rm -f \*.o

    rm -f $(TARGET)