과목 명: 시스템프로그래밍

담당 교수 명: 박 운 상

<<Assignment 2>>

**서강대학교 컴퓨터학과**

**[학번] 20121625**

**[이름] 이정명**

목 차

2. 프로그램 설명

2.1 프로그램 흐름도

3. 모듈 정의

3.1 모듈 이름 : main()

3.1.0 함수 원형

3.1.1 기능

3.1.2 사용 변수

3.2 모듈 이름:strToDec(char\* str)

3.2.0 함수 원형

3.2.1 기능

3.2.2 사용 변수

3.3 모듈이름: type(char \*filename)

3.3.0 함수 원형

3.3.1 기능

3.3.2 사용변수

3.4 모듈이름: isValidSymb(char\* label)

symbol(void)

freeSymbTabl(void)

3.4.0 함수 원형

3.4.1 기능

3.4.2 사용변수

3.5 모듈이름: rearrangeAsm(char\* asm\_line, char\*\* label, char\*\* mnemonic, char\*\* symbol, char\*\* para)

3.5.0 함수 원형

3.5.1 기능

3.5.2 사용변수

3.6 모듈이름: operationRules(int format, char\* mnemonic, char\* operand, char\* para, int NI,int \*TA, int line\_num,int\* MEM\_REF)

3.6.0 함수 원형

3.6.1 기능

3.6.2 사용변수

3.7 모듈이름: printObj(int EXT, int OBJ, char\* operand,char\*para, int temp\_int,FILE\* fp)

3.7.0 함수 원형

3.7.1 기능

3.7.2 사용변수

3.8 모듈이름: createSymbNode(char\* symbol, int adr)

createSymbTable(SYMB\_NODE\* newNode, int idx)

createAdrNode(int adr)

createAdrList(ADR\_NODE\* newNode)

createModNode(int adr,int obj,int format)

createModList(M\_NODE\* newNode)

3.8.0 함수 원형

3.8.1 기능

3.8.2 사용변수

3.9 모듈이름: assemble(FILE\* fp\_r, FILE\* fp\_lst, FILE\* fp\_obj)

3.9.0 함수 원형

3.9.1 기능

3.9.2 사용변수

4. 구조체 및 전역 변수 정의

4.1 SYMB\_NODE \*\*symb\_head, \*\*symb\_tail;

4.2 ADR\_NODE \*adr\_head, \*adr\_tail;

4.3 M\_NODE \*m\_head,\*m\_tail; HIS\_NODE/ his\_head, his\_tail

5. 코드

1. 프로그램 개요

프로젝트 #1에서 구현한 셀(shell)에 assemble 기능을 추가한다. SIC/XE의

assembly program source 파일을 입력 받아서 object파일을 생성하고, 어셈블리 과정 중 생

성된 symbol table과 결과물인 object 파일을 볼 수 있는 기능을 제공해야 함. 교재의 2.2

까지 설명된 SIC/XE 어셈블러의 기능을 구현함을 원칙으로 한다.

assemble 과정에서의 에러, 입력 에러등을 적절히 처리함으로써 프로젝트 설계를 한다

1. 프로그램 설명
   1. 프로그램 흐름도

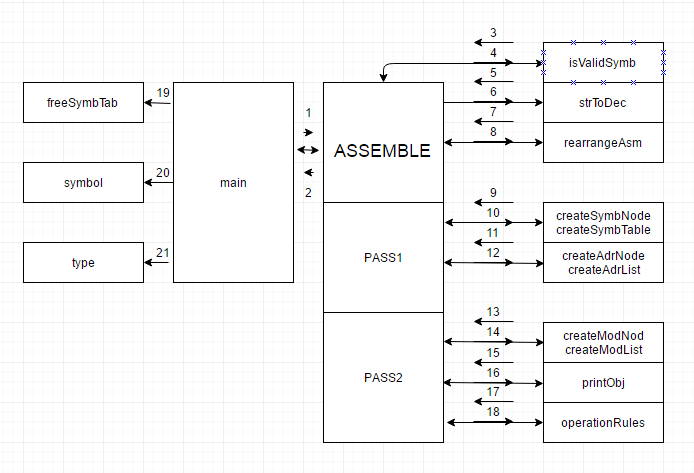


그림 > 프로그램 흐름도

1.file pointers / 2.flag

3.symbol node / 4. operands

5.Decimal / 6. Hexadecimal

7.flag / 8. instruction line(label, mnemonic, ect..)

9.symbol&address / 10. symbol table

11. address for enter / 12. address list

13. object code,address / 14. modification lis

15. object code / 17. instruction line(label, mnemonic, ect..) / 18. flag

16,19-21 function call

1. 모듈 정의
   1. 모듈 이름 : main()
   2. *3.1.0. 함수원형*int main(void)
   3. *3.1.1. 기능*

loadInstSet()을 호출해 SIC/XE instruction set을 저장한다. isValidInst() 함수를 호출해 사용자의

입력이 옳은 포맷인지 판단하고 그러할 경우 정의된 shell 함수를 호출한다. 그 외로 간단한

shell 명령어 기능구현.

또 assemble 함수를 호출해 asm file을 assemble 한다. assemble이 중간에 중단될 경우

symbol table과 lst, obj파일을 삭제한다.

* 1. *3.1.2. 사용변수*

char\*sh\_inst[17] = {"q", "quit","h", ...} : const values

char\*input, buf[100],full\_inst[100] : tokenizing pointer / input buffer / full instruction buffer

char \*inst\_tok[3], \*comma\_pos[3] : storage for string token / storage for comma's address

int j,j : indexer

int tok\_cnt, comma\_cnt, inst\_flag : token counter / comma counter

HIS\_NODE \*curr : ADT pointer for history linked list

FILE \*fp\_asm,\*fp\_lst, \*fp\_obj : filepointer

* 1. 모듈 이름: strToDec(char\* str)
  2. *3.2.0. 함수원형*
  3. int strToDec(char\* str)
  4. *3.2.1. 기능*

string을 10진수로 변환한다.

* 1. *3.2.2. 사용변수/parameters*

int dec : return value

int i : indexer

int pow : int value for multiplying power of 10

char\* str : file name

* 1. 모듈 이름: type(char\* filename)
  2. *3.3.0. 함수원형*
  3. int type(char\* filename)
  4. *3.3.1. 기능*

실행파일의 이름을 받아 실행파일일 경우 내용을 출력한다

* 1. *3.3.2. 사용변수/parameters*

FILE\* fp\_r : file pointer

char buf[100] :

char \* temp;

char\* filename : file name

* 1. 모듈 이름: isValidSymb(char\* label)

**symbol(void)**

**freeSymbTabl(void)**

* 1. *3.4.0. 함수원형*
  2. SYMB\_NODE\* isValidSymb(char\* label)

void symbol(void)

freeSymbTabl(void)

* 1. *3.4.1. 기능*

1. label 필드의 string을 받아 symbol table을 탐색한다. 만약 label이 존재할 경우 node pointer를

반환한다.

2. symbol table에 저장되어 있는 심볼들을 selection sort로 정렬해 출력한다.

3. assemble 함수 호출전, assemble 함수가 중단될 경우 symbol table을 삭제한다.

* 1. *3.4.2. 사용변수*

1. int sum = 0 : accumulator for indexing

SYMB\_NODE \*curr : node pointer for linear search

char\* label : string in label field

2. SYMB\_NODE \*curr\_i,\*arr,temp : node pointer for linear search

int i, j,curr,cnt=0 : indexer, counter

3. int i : indexer

SYMB\_NODE\* curr : node pointer for forwarding nodes

* 1. 모듈이름: rearrangeAsm(char\* asm\_line, char\*\* label, char\*\* mnemonic, char\*\* symbol, char\*\* para)
  2. *3.5.0. 함수원형*
  3. int rearrangeAsm(char\* asm\_line, char\*\* label, char\*\* mnemonic, char\*\* symbol, char\*\* para)
  4. *3.5.1. 기능*

assemble 함수에서 asm line string과 label, mnemonic, ..etc를 call by reference로 받아

tokenize한다. 만약 instruction 형식에 오류가 있을 경우 0, 성공적으로 token을 생성할 경우

1을 반환한다.

* 1. *3.5.2. 사용변수/parameter*

int len : variable for asm line length

int i : indexer

int c\_cnt = 0, a\_cnt = 0, a\_pos[2] = { 0 } : counting comma, apostrophe, array for apostrophe’s index

char\* temp : variable for 5th parameter

char \*a\_p[2],\*c\_p : pointers for addess of comma and apostrophes

char\* asm\_line, char\*\* label, char\*\* mnemonic, char\*\* symbol, char\*\* para : parameters

* 1. 모듈이름: operationRules(int format, char\* mnemonic, char\* operand, char\* para, int NI,int \*TA, int line\_num,int\* MEM\_REF)
  2. *3.6.0. 함수원형*
  3. int operationRules(int format, char\* mnemonic, char\* operand, char\* para, int NI, int \*TA, int
  4. line\_num, int\* MEM\_REF)
  5. *3.6.1. 기능*

rearranged 된 mnemonic, operand, para들과 addressing mode를 결정하는 NI, mnemonic의 format을

받아 옳은 instruction 인지 판별한다. 만약 에러가 있을 경우 line number를 출력한 후 –1을

반환한다.

또 operand가 상수일 경우 direct addressing mode 임을 MEM\_REF에 저장한다. instruction에

오류가 없을 경우 1을 반환한다.

* 1. *3.6.2. 사용변수/parameter*
  2. int format, char\* mnemonic, char\* operand, char\* para, int NI, int \*TA, int
  3. line\_num, int\* MEM\_REF : parameters

char \*FORMAT2[11] = { "ADDR","COMPR", "DIVR","MULR", "RMO","SUBR",

"SHIFTL","SHIFTR","TIXR","CLEAR","SVC"} : string array for format 2 mnemonics

char \*REG\_NUM[10] = { "A","X","L","B","S","T","F","overhead","PC","SW" } : string array for register names

SYMB\_NODE\* temp : node pointer for calling isValidsymb() function

int i,j : indexer

int REG[2] : variables for register

* 2. 모듈이름: printObj(int EXT, int OBJ, char\* operand, char\* para, int temp\_int,FILE\* fp)
  3. *3.7.0. 함수원형*
  4. void printObj(int EXT, int OBJ, char\* operand, char\* para, int temp\_int, FILE\* fp)
  5. *3.7.1. 기능*

assemble 과정에서 생성된 object code를 받아 format에 따라 다르게 file pointer에 출력해준다.

만약 directive BYTE인 경우 para에 저장된 string을 16진수, character 형식에 따라 출력한다.

* 1. *3.7.2. 사용변수/parameter*
  2. int EXT, int OBJ, char\* operand, char\* para, int temp\_int, FILE\* fp : paramters
  3. int i : indexer
  5. 모듈이름: createSymbNode(char\* symbol, int adr)
  6. **createSymbTable(SYMB\_NODE\* newNode, int idx)**
  7. **createAdrNode(int adr)**
  8. **createAdrList(ADR\_NODE\* newNode)**
  9. **createModNode(int adr,int obj,int format)**
  10. **createModList(M\_NODE\* newNode)**
  11. *3.8.0. 함수원형*
  12. SYMB\_NODE\* createSymbNode(char\* symbol, int adr)

void createSymbTable(SYMB\_NODE\* newNode, int idx)

ADR\_NODE\* createAdrNode(int adr)

void createAdrList(ADR\_NODE\* newNode)

M\_NODE\* createModNode(int adr,int obj,int format)

void createModList(M\_NODE\* newNode)

* 1. *3.8.1. 기능*
  2. 필요한 parameter를 받아 각각의 노드를 생성 반환한다.
  3. SYMB\_NODE\* ,ADR\_NODE\* ,M\_NODE\*를 반환받아 symbol table, address list, modification list를
  4. 저장한다.
  5. address list 에는 obj file에서 line feed가 필요한 location을 저장한다.
  6. modification list는 relocation이 필요한 object code 의 object code , address, format을 저장한다.
  7. *3.8.2. 사용변수/parameter*
  8. char\* symbol, int adr : symbol ,address
  9. SYMB\_NODE\* newNode, int idx : new node for addition / index of table
  10. int adr : address for line feed
  11. ADR\_NODE\* newNode : new node for addition
  12. int adr,int obj,int format : address needed relocation / object code / format of mnemonic
  13. M\_NODE\* newNode : new node for addition
  14. 모듈이름: assemble(FILE\* fp\_r, FILE\* fp\_lst, FILE\* fp\_obj)
  15. *3.9.0. 함수원형*
  16. int assemble(FILE\* fp\_r, FILE\* fp\_lst, FILE\* fp\_obj)
  17. *3.9.1. 기능*

asm file read, lst file write, obj file write을 위한 filepointer를 받아 assemble을 한다.

assemble 도중 오류가 있을 경우 line number를 출력한후 0을 반환한다.

정상적으로 lst, obj file을 생성할 경우 1을 반환한다.

* 1. *3.9.2. 사용변수/parameter*
  2. char prog\_name[7] : buffer for storing program name

char buf[200], full[120] : buffer for asm file input/ buffer for storing unmodified asm line

char \*label, \*mnemonic, \*operand, \*para : character pointers for tokenizing

int start\_adr, prog\_len, , rec\_len : start address / program length / record length

int loc\_ctr = -1, line\_ctr = 0 : location counter / line number counter

int i : indexer

int temp\_int; : temporary variable

char \*DIRECTIVES[8] = { "WORD","RESW","RESB","BYTE","BASE","NOBASE","END","START" }

: string array for directives

int TA, EXT, OP, NI, XBPE, MEM\_REF : Target address / format / opcode value / addressing mode /

value for XBPE bits / flag for memory relocation

int PC, BASE : storing PC value / storing BASE value

unsigned int OBJ : variable for object code

INST\_NODE \*temp : variable for returning node from opcode table

ADR\_NODE \*adr\_temp :variable for returning node from address list

SYMB\_NODE\* symb\_temp : variable for returning node from opcode table

M\_NODE\* m\_temp;

1. 구조체 전역 변수 정의

#define CONST 5

#define EXTENDED 4

#define WORD 3

#define DOUBLE 2

#define BYTE 1

#define DIRECTIVE 0

#define MAX\_TRECORD 30

typedef struct \_SYMB\_NODE {

char symbol[10];

int adr;

struct \_SYMB\_NODE \*next;

}SYMB\_NODE;

SYMB\_NODE \*\*symb\_head, \*\*symb\_tail : pointers for symbol table pointing first and end of the table of each index

typedef struct \_ADR\_NODE {

int adr;

struct \_ADR\_NODE \*next;

}ADR\_NODE;

ADR\_NODE \*adr\_head, \*adr\_tail : pointer for list pointing first and end of the list

typedef struct \_M\_NODE {

int adr;

int obj;

int format;

struct \_M\_NODE \*next;

}M\_NODE;

M\_NODE \*m\_head,\*m\_tail : pointer for list pointing first and end of the list

1. 코드

**5.1 assemble.c**

int strToDec(char\* str) {

int i,dec=0,pow=1;

if(!str) return -1;

for (i = strlen(str) - 1; i >= 0; i--) {

if ('0' <= str[i] && str[i] <= '9') { dec += (str[i] - '0')\*pow; }

else return -1;

pow \*= 10;

}

return dec;

}

SYMB\_NODE\* isValidSymb(char\* label) {

SYMB\_NODE \*curr;

if(!label) return NULL; // NULL string

if (!('A' <= label[0] && label[0] <= 'Z')) return NULL;

for (curr = symb\_head[label[0]-'A']; curr; curr = curr->next) {

if (!strcmp(curr->symbol, label)) {

return curr;

}

}

return NULL; // symbol not found

}

int rearrangeAsm(char\* asm\_line, char\*\* label, char\*\* mnemonic, char\*\* symbol, char\*\* para) {

int len = strlen(asm\_line);

int i, c\_cnt = 0, a\_cnt = 0, a\_pos[2] = { 0 };

char\* temp=NULL, \*a\_p[2],\*c\_p;

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

if (asm\_line[i] == ',' ){

if(a\_cnt == 1 && a\_pos[0] < i && a\_pos[1] == 0) continue; // case of ' .. , ..'

if(c\_cnt>=1) return 0; // else if several ','

c\_p=&asm\_line[i];

asm\_line[i] = ' ';

c\_cnt++;

}

else if (asm\_line[i] == 39) {

if(a\_cnt>=2) return 0;

a\_p[a\_cnt]=&asm\_line[i];

asm\_line[i] = ' ';

a\_pos[a\_cnt] = i;

a\_cnt++;

if(a\_cnt==2) temp=&asm\_line[i+1]; // 5th token

}

}

\*label = strtok(asm\_line, " \t\r\n");

\*mnemonic = strtok(NULL, " \t\r\n");

\*symbol = strtok(NULL, " \t\r\n");

if (a\_pos[0]) {

if(a\_cnt!=2) return 0;

asm\_line[a\_pos[1]] = '\0';

//if (!(\*symbol + strlen(\*symbol))[0] == asm\_line[a\_pos[0]-1]) return 0;

\*para = strtok(NULL, "\r\n");

}

else \*para = strtok(NULL, " \t\r\n");

if((temp=strtok(temp," \t\r\n"))) return 0;

/\*rearrangement\*//////////////////////////////////////////////////

if (!(\*symbol)) {

\*para = \*symbol; \*symbol = \*mnemonic; \*mnemonic = \*label; \*label = NULL;

}

else if ((a\_cnt||c\_cnt) && !(\*para)) {

\*para = \*symbol; \*symbol = \*mnemonic; \*mnemonic = \*label; \*label = NULL;

}

if(\*label){

for(i=0;i<strlen(\*label);i++){ // constraints for label name

if(i==0){

if(!(('A'<=(\*label)[i] && (\*label)[i]<='Z')

||('a'<=(\*label)[i] && (\*label)[i]<='z'))) return 0;

}

else if(!(('A'<=(\*label)[i] && (\*label)[i]<='Z')

||('a'<=(\*label)[i] && (\*label)[i]<='z')

||('0'<=(\*label)[i] && (\*label)[i]<='9'))) return 0;

}

}

if(c\_cnt){

if(!(\*symbol||\*para)) return 0;

if(!(&(\*symbol)[strlen(\*symbol)-1]<=c\_p && c\_p<=&(\*para)[0])) return 0; // location of ,

}

if(a\_cnt){

if(!(\*symbol||\*para)) return 0;

if(!(&(\*symbol)[strlen(\*symbol)-1]<=a\_p[0] && a\_p[0]<=&(\*para)[0]&& &(\*para)[strlen(\*para)-1]<=a\_p[1])) return 0; // location of '

}

return 1;

}

void symbol() {

SYMB\_NODE \*curr\_i,\*arr,temp;

int i,j,curr,cnt=0;

for (i = 0; i < 26; i++) { // count for memory allocation

for (curr\_i = symb\_head[i]; curr\_i; curr\_i = curr\_i->next) cnt++;

}

if(!cnt){

printf("empty symbol table\n");

}

arr = (SYMB\_NODE\*)malloc(sizeof(SYMB\_NODE)\*cnt);

cnt = 0;

for (i = 0; i < 26; i++) { // assign values from symbol table to array

for (curr\_i = symb\_head[i]; curr\_i; curr\_i = curr\_i->next) {

arr[cnt].adr=curr\_i->adr;

strcpy(arr[cnt++].symbol, curr\_i->symbol);

}

}

for (i = 0; i < cnt; i++) { // selection sort

curr = i;

for (j = i + 1; j < cnt; j++) {

if (strcmp(arr[curr].symbol, arr[j].symbol)<0) {

curr = j;

}

temp.adr = arr[curr].adr; strcpy(temp.symbol, arr[curr].symbol);

arr[curr].adr=arr[i].adr; strcpy(arr[curr].symbol,arr[i].symbol);

arr[i].adr = temp.adr; strcpy(arr[i].symbol, temp.symbol);

}

printf("\t%s\t%04X\n", arr[i].symbol, arr[i].adr);

}

free(arr);

}

int operationRules(int format, char\* mnemonic, char\* operand, char\* para, int NI,int \*TA, int line\_num,int\* MEM\_REF) {

char \*FORMAT2[11] = { "ADDR","COMPR", "DIVR","MULR", "RMO","SUBR", // 0~5 : r1 r2

"SHIFTL","SHIFTR","TIXR","CLEAR","SVC"}; // 6~7 : r1 n / 8 ~ 9: r1 / 9 : n

char \*REG\_NUM[10] = { "A","X","L","B","S","T","F","overhead","PC","SW" };

SYMB\_NODE\* temp;

int i,j,REG[2];

switch (format) {

case BYTE: if (operand) { printf("unnecessary operand for format 1 on line %d\n",line\_num); return -1; }

break;

case DOUBLE:for (i = 0; i < 10; i++) {

if (!strcmp(mnemonic, FORMAT2[i])) break;

}

if(!operand) { printf("not enough operands on line %d\n", line\_num); return -1; }

for (j = 0; j < 10; j++) {

if (!strcmp(operand, REG\_NUM[j])) { REG[0] = j; break; }

}

if (j == 7 || j > 9) { printf("not available on operand line %d\n", line\_num); return -1; }

if (0 <= i && i <= 5) {

if (!para) { printf("not enough operands for on line %d\n", line\_num); return -1; }

for (j = 0; j < 10; j++) {

if (!strcmp(para, REG\_NUM[j])) { REG[1] = j; break; }

}

if (j == 7 || j > 9) { printf("not available on line %d\n", line\_num); return -1; }

}

else if ( 6 == i || i == 7) {

if (!para) { printf("not enough operands on line %d\n", line\_num); return -1; }

REG[1]=strToDec(para);

if(!(1<=REG[1]&&REG[1]<=16)) { printf("unavailable value on line %d\n", line\_num); return -1; }

REG[1]--;

}

else if (i == 8 || i == 9) {

if (para) { printf("too much operands on line %d\n", line\_num); return -1; }

REG[1] = 0;

}

else if (i == 10) {

if (para) { printf("too much operands on line %d\n", line\_num); return -1; }

if (!(0<=REG[0]&&REG[0] <= 15)) { printf("unavailable value on line %d\n", line\_num); return -1; }

}

if(REG[0]>15||REG[1]>15) { printf("unavailable value on line %d\n", line\_num); return -1; }

\*TA = REG[0] \* 16 + REG[1] \* 1;

break;

case WORD:

case EXTENDED: if (!strcmp(mnemonic, "RSUB")) {

if (operand) { printf("unnecessary operand on line %d\n", line\_num); return -1; }

else break;

}

else if (!operand) { printf("not enough operand on line %d\n", line\_num); return -1; }

else { // immediate addressing mode

if((NI==1 || NI==2)&&para) { printf("unnecessary operand on line %d\n", line\_num); return -1; }

if (!(temp=isValidSymb(operand))) { // constant operands

if ((\*TA=strToDec(operand)) < 0) { printf("unavailable value on line %d\n", line\_num); return -1; }

if (format == WORD) { // op #c / op @c

if(!(0<=\*TA&& \*TA<=4095)) { printf("unavailable value on line %d\n", line\_num); return -1; } // contant value 0~4095

}

if (format == EXTENDED) { // +op #m / +op @m

if (!( 4096<= \*TA && \*TA <= 1048575)) { printf("unavailable value on line %d\n",line\_num); return -1; } // contant value 4096~1048575

}

\*MEM\_REF = CONST;

if(NI==1) \*MEM\_REF = -1;

}

else { // no need for relocation but relative

\*TA = temp->adr;

if(NI==1) \*MEM\_REF = 1;

}

}

break;

}

return 1;

}

void printObj(int EXT, int OBJ, char\* operand,char\*para, int temp\_int,FILE\* fp) {

int i;

if (EXT == EXTENDED) fprintf(fp, "%08X",(unsigned int) OBJ);

else if (EXT == WORD) {

if (!operand) fprintf(fp, "%06X", OBJ);

else fprintf(fp, "%06X", OBJ);

}

else if (EXT == DOUBLE) fprintf(fp, "%04X", OBJ);

else if (EXT == BYTE) fprintf(fp, "%02X", OBJ);

else if (EXT == CONST) { // if DIRECTIVE BYTE

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

if (temp\_int % 2) fprintf(fp, "0");

for(i=0;i<temp\_int;i++){

if('a'<=para[i]&&para[i]<='z') para[i] +='A' - 'a';

}

fprintf(fp, "%s", para);

}

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

for (i = 0; i< temp\_int; i++) {

fprintf(fp, "%02X", para[i]);

}

}

}

}

int assemble(FILE\* fp\_r, FILE\* fp\_lst, FILE\* fp\_obj) {

char prog\_name[7];

char buf[200], full[120], \*label, \*mnemonic, \*operand, \*para;

int start\_adr, prog\_len, loc\_ctr = -1, line\_ctr = 0, rec\_len;

int i, temp\_int;

char \*DIRECTIVES[8] = { "WORD","RESW","RESB","BYTE","BASE","NOBASE","END","START" };

/////////////////////////////////////////////////////////////////////////////

int TA, EXT, OP, PC, BASE, NI, XBPE, MEM\_REF;

unsigned int OBJ;

INST\_NODE \*temp;

ADR\_NODE \*adr\_temp;

SYMB\_NODE\* symb\_temp;

M\_NODE\* m\_temp;

if (!fp\_r) { printf("file open error\n"); return 0; }

/\* pass 1 \*//////////////////////////////////////////////////////

while (fgets(buf, 200, fp\_r)) {

/\* initailize variables \*////////////////////////////////////

para = label = mnemonic = operand = NULL;

EXT = 0;

line\_ctr += 5;

/////////////////////////////////////////////////////////////

if (buf[0] == '.') continue; // continue if comment

if (!rearrangeAsm(buf, &label, &mnemonic, &operand, &para)) { //rearrange instruction in proper form

printf("file format error on line %d\n", line\_ctr); return 0;

}

if (!(label || mnemonic || operand || para)) continue; // continue if empty line

if (mnemonic[0] == '+') { mnemonic++; EXT = EXTENDED; } // '+'prefix for format4

if (loc\_ctr == -1) { // start of the program

if (!strcmp(mnemonic, DIRECTIVES[7])) { // prog starts with DIRECTIVES "START"

strcpy(prog\_name, label);

loc\_ctr = PC = start\_adr = hexToDec(operand, 'a');

}

else { // prog starts with no DIRECTIVES "START"

strcpy(prog\_name, "");

loc\_ctr = PC = start\_adr = 0;

}

rec\_len = 0;

}

if (label) {

if (isValidSymb(label)) { printf("duplicate symbol on line %d\n", line\_ctr); return 0; } // duplicate symbol error

if((temp\_int = label[0] - 'A')<0){printf("unavailable instruction on line %d\n",line\_ctr); return 0;}; // unauthorized index

createSymbTable(createSymbNode(label, PC), temp\_int); // create symbol table

}

/\* asm directives \*//////////////////////////////////////////

for (i = 0; i < 8; i++) {

if (!strcmp(mnemonic, DIRECTIVES[i]))break;

}

if(0<=i&&i<=7){

switch (i) {

case 0:PC += 3; break;

case 1: if ((temp\_int = strToDec(operand)) < 0) { printf("unavailable value on line %d\n", line\_ctr); return 0; }

PC += temp\_int \* WORD; EXT = CONST; break;

case 2: if ((temp\_int = strToDec(operand)) < 0) { printf("unavailable value on line %d\n", line\_ctr); return 0; }

PC += temp\_int \* BYTE; EXT = CONST; break;

case 3: if (!strcmp(operand, "X")) {

temp\_int = strlen(para);

for (i = 0; i < temp\_int; i++) {

if (para[i] != '0') break;

}

if (i == temp\_int) para += (i - 1);

else para += (i);

if (60 < (temp\_int = strlen(para))) { printf("unavailable operand on line %d\n", line\_ctr); return 0; }

for (i = 0; i < temp\_int; i++) {

if (!(('A' <= para[i] && para[i] <= 'F') ||

('a' <= para[i] && para[i] <= 'f') ||

('0' <= para[i] && para[i] <= '9'))) {

printf("unavailable operand on line %d\n", line\_ctr); return 0;

}

}

PC += temp\_int / 2 + temp\_int % 2;

}

else if (!strcmp(operand, "C")) {

if ((temp\_int = strlen(para))>30) { printf("unavailable value on line %d\n", line\_ctr); return 0; }

PC += temp\_int \* 1;

}

else { printf("unavailable operand on line %d\n", line\_ctr); return 0; }

EXT = BYTE;

break;

case 4:

case 5:continue;

}

if (!strcmp(mnemonic, DIRECTIVES[6])) {

if((symb\_temp=isValidSymb(operand))) {

if(symb\_temp->adr!=start\_adr){

printf("END should specify the start of the program... line %d\n", line\_ctr); // if END operand do not indicate start of prog

return 0;

}

}

else{

printf("no operand for END on line %d",line\_ctr); // if NO operand for END

return 0;

}

}

if (!strcmp(mnemonic, DIRECTIVES[7]) && loc\_ctr!=start\_adr) { // if START middle of the program

printf("unavailable directive on line %d\n", line\_ctr);

return 0;

}

}

/\* mnemonic found in opcode table \*//////////////////////////

else if ((temp = opcodeMnemonic(mnemonic, 'a'))) {

if (!strcmp(temp->inst\_type, "1")) { PC += BYTE; EXT = BYTE; }

else if (!strcmp(temp->inst\_type, "2")) { PC += DOUBLE; EXT = DOUBLE; }

else if (!strcmp(temp->inst\_type, "3/4")) {

PC += 3;

if (EXT == EXTENDED)PC++;

else EXT = WORD;

}

}

/\* Not a directive nor mnemonic \*/////////////////////////////

else{

printf("unavailable instruction on line %d\n ",line\_ctr);

return 0;

}

/\* save record length \*//////////////////////////////////////

if (rec\_len + (PC - loc\_ctr) > MAX\_TRECORD) {

createAdrList(createAdrNode(rec\_len));

if ((rec\_len = PC - loc\_ctr)>MAX\_TRECORD) rec\_len = 0;

}

else if(rec\_len + (PC-loc\_ctr)==MAX\_TRECORD){

rec\_len += PC-loc\_ctr;

createAdrList(createAdrNode(rec\_len));

rec\_len = 0;

}

else if (DIRECTIVE<EXT&&EXT<CONST) rec\_len += PC - loc\_ctr;

loc\_ctr = PC;

}

createAdrList(createAdrNode(rec\_len)); // create Address node list

adr\_temp = adr\_head; // point first of the list

prog\_len = PC - start\_adr;

fseek(fp\_r, 0, 0); // reset file pointer

BASE = PC = loc\_ctr = start\_adr;

rec\_len = line\_ctr = 0;

/\* pass 2 \*/////////////////////////////////////////////////////////

fprintf(fp\_obj, "H%-6s%06X%06X\n", prog\_name, start\_adr, prog\_len); // write Header Record

while (fgets(buf, 90, fp\_r)&&EXT!=-1) {

/\*initialize variables\*/////////////////////////////////////////

para = label = mnemonic = operand = NULL;

temp = NULL;

MEM\_REF = temp\_int = XBPE = OP = TA = EXT = OBJ = 0;

line\_ctr += 5;

////////////////////////////////////////////////////////////////

if (buf[0] == '.') { fprintf(fp\_lst, "%3d\t%s", line\_ctr, buf); continue; } // continue if comment

strcpy(full, buf);

if(strchr(full, '\n')){

\*strchr(full,'\0')=-52;

\*strchr(full,'\n')= '\0';

}

else if(strchr(full,'\r')) \*strchr(full,'\r') = '\0'; // exchange '\n' to '\0'

if (!rearrangeAsm(buf, &label, &mnemonic, &operand, &para)) { // rearrange instruction to proper format

printf("file format error!\n"); return 0;

}

if (!(label || mnemonic || operand || para)) continue;

if (mnemonic[0] == '+') { mnemonic++; EXT = EXTENDED; XBPE = 1; } // '+' prefix for format4

if (operand) {

if (operand[0] == '@') { operand++; NI = 2; }

else if (operand[0] == '#') { operand++; NI = 1; }

else NI = 3;

}

/\* directives \*/////////////////////////////////////////////////

for (i = 0; i < 8; i++) {

if (!strcmp(mnemonic, DIRECTIVES[i]))break;

}

if(0<=i&&i<=7) {

switch (i) {

case 0:

case 1:

case 2: if ((TA = strToDec(operand)) < 0||TA>MAX\_MEM) { printf("unavailable value on line %d\n",line\_ctr); return 0; }

if (i == 0) { PC += WORD; EXT = WORD; }

else if (i == 1) { PC += TA\*WORD; EXT = DIRECTIVE; }

else if (i == 2) { PC += TA\*BYTE; EXT = DIRECTIVE; }

break;

case 3: temp\_int = strlen(para);

if (!strcmp(operand, "X")) {

for (i = 0; i < temp\_int; i++) {

if (para[i] != '0') break;

}

if (i == temp\_int) para += (i - 1);

else para += (i);

if (60<(temp\_int = strlen(para))) { printf("unavailable operand on line %d\n", line\_ctr); return 0; }

for (i = 0; i < temp\_int; i++) {

if (!(('A' <= para[i] && para[i] <= 'F') ||

('a' <= para[i] && para[i] <= 'f') ||

('0' <= para[i] && para[i] <= '9'))) {

printf("unavailable operand on line %d\n", line\_ctr); return 0;

}

}

PC += temp\_int / 2 + temp\_int % 2;

}

else if (!strcmp(operand, "C")) {

if (temp\_int>30) { printf("unavailable operand on line %d\n", line\_ctr); return 0; }

for (i = 0; i< temp\_int; i++) {

if (!(0 <= para[i] && para[i] <= 255)) { printf("unavailable operand on line %d\n", line\_ctr); return 0; }

}

PC += strlen(para);

}

else { printf("unavailable indicater on line %d\n", line\_ctr); return 0; }

EXT = CONST;

break;

case 4: if((symb\_temp=isValidSymb(operand))) BASE =symb\_temp->adr;

break;

case 5: BASE = -1; // base register can no longer be relied upon for addressing

break;

case 6: EXT = -1;

break;

case 7: if(strToDec(operand)==-1){printf("unavailable operand on line %d\n",line\_ctr); return 0;}

EXT = 0;

break;

}

}

/\* mnemonic found in opcode Table \*///////////////////////////////

else if ((temp = opcodeMnemonic(mnemonic, 'a'))) {

if (!strcmp(temp->inst\_type, "1")) { PC += BYTE; EXT = BYTE; NI = 0; }

else if (!strcmp(temp->inst\_type, "2")) { PC += DOUBLE; EXT = DOUBLE; NI = 0; }

else if (!strcmp(temp->inst\_type, "3/4")) {

PC += WORD;

if (EXT == EXTENDED) PC++;

else EXT = WORD;

OP = NI;

if (para) {

if (!strcmp(para, "X")) {

if (NI == 1 || NI == 2) { printf("wrong combinations of addressing mode on line %d\n",line\_ctr); return 0; } // indexing mode after @# prefix

XBPE += 8;

}

else{

printf("unavailable parameter on line %d\n",line\_ctr);

}

}

}

if (operationRules(EXT, mnemonic, operand, para, NI, &TA, line\_ctr, &MEM\_REF) == -1) { printf("error occured while assembling on line %d...\n",line\_ctr); return 0; }

OP += hexToDec(temp->opcode, 'a');

}

else{

printf("unavilable instruction on line %d\n",line\_ctr); return 0;

}

if (!(MEM\_REF == CONST || MEM\_REF==-1)) {

if ((WORD == EXT) && (-2048 <= TA - PC && TA - PC <= 2047)) { TA -= PC; XBPE += 2; } // PC relative

else if ((WORD == EXT) && (0 <= TA - BASE && TA - BASE <= 4095)&&BASE!=-1) { TA -= BASE; XBPE += 4; } // BASE relative

}

if (TA < 0) TA += 4096;

/\* generate obj code \*//////////////////////////////////////////////

if (EXT == EXTENDED) { OBJ = OP \* 16777216 + XBPE \* 1048576 + TA; }

else if (EXT == WORD) { OBJ = OP \* 65536 + XBPE \* 4096 + TA; }

else if (EXT == DOUBLE) { OBJ = OP \* 256 + TA; }

else if (EXT == BYTE) { OBJ = OP; }

else if (EXT == CONST) { OBJ = OP; }

/\*create Modification list\*/////////////////////////////////////////

if((EXT==EXTENDED && NI!=1) || MEM\_REF==CONST){

createModList(createModNode(loc\_ctr-start\_adr+1,OBJ,EXT));

}

/\* write lst code \*/////////////////////////////////////////////////

fprintf(fp\_lst, "%3d\t", line\_ctr);

if(EXT!=-1)fprintf(fp\_lst, "%04X",loc\_ctr);

fprintf(fp\_lst,"\t%-40s", full);

printObj(EXT, OBJ, operand, para, temp\_int,fp\_lst);

fprintf(fp\_lst, "\n");

/\*write obj code \*//////////////////////////////////////////////////

if (loc\_ctr == start\_adr&&EXT) {

fprintf(fp\_obj, "T%06X%2X", loc\_ctr, adr\_temp->adr);

adr\_temp = adr\_temp->next;

}

if (rec\_len + (PC - loc\_ctr) > MAX\_TRECORD) {

if (EXT) {

fprintf(fp\_obj, "\nT%06X%02X", loc\_ctr, adr\_temp->adr);

adr\_temp = adr\_temp->next;

}

printObj(EXT, OBJ, operand, para, temp\_int,fp\_obj);

rec\_len = PC - loc\_ctr;

}

else if (rec\_len + (PC - loc\_ctr) == MAX\_TRECORD) {

printObj(EXT, OBJ, operand, para, temp\_int, fp\_obj);

if (EXT) {

fprintf(fp\_obj, "\nT%06X%02X", loc\_ctr, adr\_temp->adr);

adr\_temp = adr\_temp->next;

}

rec\_len = 0;

}

else {

printObj(EXT, OBJ, operand, para, temp\_int,fp\_obj);

if (EXT > DIRECTIVE) {

rec\_len += PC - loc\_ctr;

}

}

loc\_ctr = PC;

}

/\* write modification record \*//////////////////////////////////////////

m\_temp = m\_head;

while(m\_temp){

if(m\_temp->format==EXTENDED)

fprintf(fp\_obj,"\nM%06X%08X05",m\_temp->adr,m\_temp->obj);

else if (m\_temp->format==WORD)

fprintf(fp\_obj,"\nM%06X%06X03",m\_temp->adr,m\_temp->obj);

m\_temp=m\_temp->next;

}

fprintf(fp\_obj,"\nE%06X",start\_adr);

/\* free allocated memory \*//////////////////////////////////////////////

while(adr\_head){

adr\_temp=adr\_head;

adr\_head=adr\_head->next;

free(adr\_temp);

}

while(m\_head){

m\_temp=m\_head;

m\_head=m\_head->next;

free(m\_temp);

}

return 1;

}

void freeSymbTab(){

int i;

SYMB\_NODE\* curr;

for(i=0;i<26;i++){

while(symb\_head[i]){

curr=symb\_head[i];

symb\_head[i]=symb\_head[i]->next;

free(curr);

}

}

}

**5.2 createTable.c**

SYMB\_NODE\* createSymbNode(char\* symbol, int adr) {

SYMB\_NODE \*newNode = (SYMB\_NODE\*)malloc(sizeof(SYMB\_NODE));

strcpy(newNode->symbol, symbol);

newNode->adr = adr;

newNode->next = NULL;

return newNode;

}

void createSymbTable(SYMB\_NODE\* newNode, int idx) {

if (!symb\_head[idx]) { symb\_head[idx] = newNode; symb\_tail[idx] = newNode; } // if hashtable's row is empty

else {

symb\_tail[idx]->next = newNode;

symb\_tail[idx] = newNode;

}

}

ADR\_NODE\* createAdrNode(int adr) {

ADR\_NODE \*newNode = (ADR\_NODE\*)malloc(sizeof(ADR\_NODE));

newNode->adr = adr;

newNode->next = NULL;

return newNode;

}

void createAdrList(ADR\_NODE\* newNode) {

if (!adr\_head) { adr\_head = newNode; adr\_tail = newNode; } // if hashtable's row is empty

else {

adr\_tail->next = newNode;

adr\_tail = newNode;

}

}

M\_NODE\* createModNode(int adr,int obj,int format){

M\_NODE\* newNode = (M\_NODE\*)malloc(sizeof(M\_NODE));

newNode->adr = adr;

newNode->obj=obj;

newNode->format = format;

newNode->next= NULL;

return newNode;

}

void createModList(M\_NODE\* newNode){

if (!m\_head) { m\_head = newNode; m\_tail = newNode; } // if hashtable's row is empty

else {

m\_tail->next = newNode;

m\_tail = newNode;

}

}

**5.3 shell.c**

int type(char\* filename) {

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

char buf[100];

char \* temp;

if (!strchr(filename, '.')){ printf("not an excutable file\n"); return 0;}

if (!fp\_r) { printf("file open error\n");return 0; }

while (fgets(buf,100,fp\_r)){

if((temp=strchr(buf,'\r'))) \*temp= ' ';

printf("%s", buf);

}

printf("\n");

fclose(fp\_r);

return 1;

}