**과목명: 시스템프로그래밍**

**2분반**

**<<Project #2>>**

**서강대학교 [심리학과]**

**[20140424]**

**[문성혁]**

목 차

1. **프로그램 개요**
2. **프로그램 흐름도**
3. **모듈 정의**
   1. 정의한 모듈의 변수 및 함수 소개
4. **파싱 전략**
5. 프로그램 개요

본 프로그램은 시스템 프로그래밍 수업의 두 번째 프로젝트 결과물입니다. 이번 프로젝트의 목표는 프로젝트 1에서 구현한 Shell에 assembler의 기능을 추가하는 것이 핵심입니다. 즉, ‘assemble’ 커맨드와 assembly로 작성된 텍스트 파일을 이용해, pass1과 pass2를 거쳐 최종적으로 리스팅 파일과 오브젝트 파일을 만들어 내는 것입니다. 또한, 이 과정에서 생성되는 symbol table을 유지하여 화면에 출력하는 ‘symbol’ 커맨드와, 현재 폴더 내 특정 파일의 내용을 보여주는 ‘type’ 커맨드도 추가적으로 구현하였습니다.

최종적으로 추가 또는 수정된 커맨드들은 다음과 같습니다.

* h[elp] (수정됨)
* assemble filename (추가됨)
* type filename (추가됨)
* symbol (추가됨)

가급적이면 프로젝트 1에서 작성한 코드를 수정하지 않고 진행하려 했으나, 전체적인 구조를 고려하여 기존 코드를 일부 리팩토링 하였습니다. 이에 대해서는 아래에서 자세히 설명하겠습니다.

1. 프로그램 흐름도

전체적인 흐름도는 기존과 같으므로, 여기서는 assemble 커맨드가 입력되었을 때의 흐름을 상세히 전개하겠습니다. 참고로 모든 흐름도는 [이 링크](https://drive.google.com/file/d/1cgfVnlScFPZojPmojl7xPFtUvtpAl-_x/view?usp=sharing)에서 원 다이어그램을 확인할 수 있습니다.

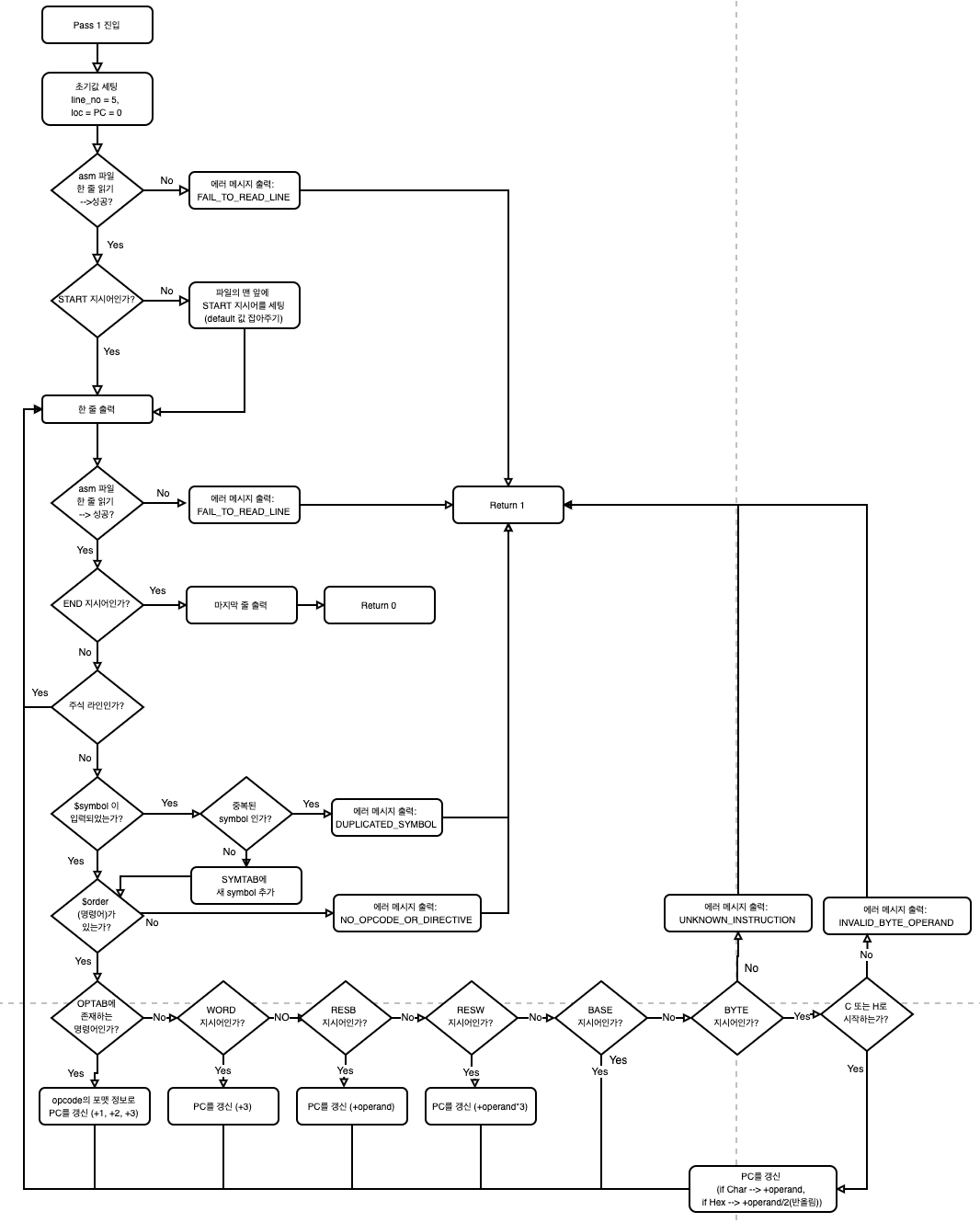
우선, assemble 메인 함수의 흐름도입니다.

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

assemble 관련 functions 중 가장 바깥단에 위치한 `assemble` 함수는 주로 현재 상황에 맞는 파일을 열거나(fopen) 닫는(fclose) 역할을 하고 있습니다. 이 함수에서는 두 개의 에러를 직접 탐지하는데, 이는 각각 (1) 파일 이름 파라미터가 전달되지 않았을 경우와 (2) 해당 이름의 파일이 존재하지 않았을 경우입니다. 나머지는 `pass1`, `pass2` 함수 내부에서 자체적으로 에러 메시지를 출력합니다. assemble에서는 각각의 함수에서 리턴받은 값으로 에러가 발생했는지를 판단하고, 에러가 발생했을 경우엔 최상위 main 함수로 1을 리턴합니다. 에러가 발생하였는가와 무관하게 열려있는 모든 FILE \* 포인터들을 fclose() 하여 Segmentation fault erro를 방지하였습니다.

두 번째 흐름도는 Pass1의 흐름도입니다. pass1()에서는 .asm 파일을 읽어 .itm (intermediate) 파일에 작성합니다.



지시어, 혹은 opcode에 해당하는 입력 부분에 어떤 변수명을 사용해야 할 지 많이 고민하였습니다. 처음에는 opcode로 설정하려 했으나, 이미 프로젝트 1에서 `opcode`라는 명령어를 사용하였기 때문에 사용할 수 없었습니다. 또한 이 자리에는 opcode라기보단 정확히는 opcode로 변환되어야 할 mnemonic이 입력되며 경우에 따라선 assembler directive가 입력되는 경우도 있으므로 이 모든 케이스를 고려하여 “order”이라는 변수명을 설정하였습니다. 따라서, asm 파일의 포맷은 [symbol,] order [, operand] 입니다.

몇 가지 각별히 유의하여 작성한 부분이 있습니다. 우선, 첫 번째 줄이 “START”가 아닐 경우 임의로 default 값을 설정해 START 지시어가 존재하는 것처럼 만들었습니다. 이는 gcc 컴파일 시 -o 옵션을 주지 않으면 a.c 이라는 이름으로 파일이 생성되는 부분에서 착안하였습니다. 또한, BYTE 지시어의 경우 첫 글자가 ‘H, C, h, c’ 중 하나가 아니라면 이해할 수 없는 operand로 판단하고 에러로 처리했습니다.

결과적으로 pass1()에서는 다음과 같은 오류들이 발생할 수 있습니다.

* FAIL\_TO\_READ\_LINE
* DUPLICATED\_SYMBOL
* NO\_OPCODE\_OR\_DIRECTIVE
* INVALID\_BYTE\_OPERAND
* UNKNOWN\_INSTRUCTION

.itm 파일에는 총 다섯 가지 칼럼이 한 줄에 입력되며, 각각 line\_no, loc, symbol, order, operan 입니다. 빈 칸에는 “~” 특수기호를 넣어 placeholder 역할을 하도록 했습니다.

마지막으로 Pass2의 흐름도를 보겠습니다. pass2()에서는 .itm 파일을 읽어 .lst 파일과 .obj 파일을 작성합니다. 멀리서 본 전체적인 그림은 다음과 같습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

명령어(instruction)가 OPTAB에 존재하는 경우를 제외한 나머지 케이스가 위 흐름도에 나와 있습니다. unsigned char objCode[255] 배열을 활용하여, 한 줄에 입력해야 하는 모든 Text record를 순차적으로 저장했습니다. 이때, 한 줄에 최대 FF(255)개 입력이 가능한 Text record와 달리 listing file은 한 번의 loop 동안 한 줄씩 작성해야 하므로, 매 loop가 시작할 때 ‘현재까지 입력된 objCode character의 갯수’를 저장하고, loop 끝에서 listing file에 저장할 때 이 값을 활용하여 해당 라인에 해당하는 만큼의 object code를 작성할 수 있었습니다. RESB, RESW 등의 변수 관련 지시어가 들어오면 Text record를 줄바꿈 해야 하므로, 이에 대해서는 별도로 boolean 값을 두어 force line break를 구현했습니다.

위 흐름도에서 잘린 부분이 아래 나머지 흐름도에 나와 있습니다.

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

opcode가 발견되었다면, 즉 정상적인 SIC/XE 명령어 중 하나라면, 위 흐름도와 같은 로직이 진행됩니다. 우선 현재 라인에 해당하는 objCode[i]의 첫 인덱스에 opcode의 8비트를 입력합니다. 다음 인덱스로 넘어가기 전에, operand가 있으며 operand[0]가 # 또는 @ 일 경우, order가 3/4포맷일 경우 n/i bit를 적절히 더해줍니다.

이후 objCode[i+1] 부터 objCode[i+2] (4포맷일 경우엔 objCode[i+3]) 까지 입력해야 합니다. 만약 1포맷일 경우엔 8 bit가 전부이므로 그대로 리스팅 파일 출력 파이프라인으로 보냅니다. 2포맷일 경우엔 r1, r2에 해당하는 각 4bit를 찾아야 하므로, operand를 ‘,’ 기준으로 분리해 register mnemonic에 맞는 number로 변환 후 이를 objCode[i+1]에 입력합니다.

3포맷일 경우엔 우선 operand가 어떤 값인지 확인합니다. SYMTAB에 없다면 이 값을 immediate value로 판단하고, number로 변환해 objCode의 나머지 인덱스에 입력합니다(3 또는 4포맷). 반면 operand가 등록된 symbol 중 하나라면 해당 symbol의 address를 displacement로 간주하고, 이에 대해 addressing을 시도합니다. addressing은 PC – Base 순으로 시도합니다. objCode에 입력이 완료되면 리스팅 파일을 출력하기 위해 해당 파이프라인으로 전달됩니다.

1. 모듈 정의
   1. 정의한 모듈의 변수 및 함수 소개
2. assemble.c : paas1, pass2 및 이를 수행하기 위한 몇 가지 helper function들이 포함되어있습니다.

아래는 assemble.c의 상단 선언부 입니다. (standard libraries 제외)

#include "optab.h" // OPTAB을 참조할 때 사용  
#include "symtab.h" // SYMTAB을 참조할 때 사용  
  
// asm 파일, itm 파일을 한 줄씩 읽는 array를 선언할 때 사  
#define READ\_LINE\_LENGTH 255  
// Modification record를 담는 array를 선언할 때 사용  
#define MAX\_MODIFICATION\_RECORD 100  
  
// 4개의 파일을 각각의 포인터로 관리  
static FILE \*fp\_asm, \*fp\_itm, \*fp\_lst, \*fp\_obj;  
// 파일 한 줄을 읽는 포인터와 세 칼럼을 보관하는 포인터  
static char line[READ\_LINE\_LENGTH], \*symbol, \*order, \*operand;  
// 좌측 라인 넘버, LOC, PC, (pass1에서 계산되는) 전체 프로그램 길이  
static int line\_no, loc, PC, program\_length;  
// B register 값. BASE 지시어가 없으면 사용할 수 없다.  
static int regB = -1;  
// format 2 --> 두 레지스터 값을 찾기 위해 사용  
static int r1, r2;  
// format 3/4 --> operand 1 [,2] 을 보관하는 포인터  
static char \*first\_operand, \*second\_operand;  
// OPTAB에서 mnemonic에 맞는 opcode 구조체를 가져올 때 사용  
static optab\_node op;  
// SYMTAB에서 symbol에 맞는 symbol 구조체를 가져올 때 사용  
static symtab\_node sym;  
// 파일 출력 시 빈 칸이 '(null)' 텍스트가 뜨는 것을 방지  
static char \*placeHolder = " ";  
// Modification Record를 보관  
static int m\_record\_node[MAX\_MODIFICATION\_RECORD];  
// Modification Record의 갯수를 저장  
static int m\_record\_cnt = 0;  
  
/\*\*  
 \* 공통 요소  
 \*/  
// 확장자만 바꿔서 파일 오픈  
static void \_open\_file\_with\_ext(FILE \*\*fp, char \*name, const char ext[4], char mode[2]);  
// 확장자만 바꿔서 파일 삭제 (에러 발생 시 사용)  
static void \_delete\_file\_with\_ext(char \*name, const char ext[4]);  
  
/\*\*  
 \* Pass 1 관련  
 \*/  
// asm 파일 1 줄을 읽는다. 성공 시 return 0  
static int \_parse\_asm\_line();  
// 성공 시 return 0  
static int \_pass1();  
  
/\*\*  
 \* Pass 2 관련  
 \*/  
// itm(intermediate) 파일 1 줄을 읽는다. 성공 시 return 0  
static int \_parse\_itm\_line();  
// 성공 시 return 0  
static int \_pass2();  
// register 이름을 그에 맞는 int로 변환한다.  
static int \_convertStrToRegId(const char \*target);  
// '0' ~ '9', 'A' ~ 'F'의 char를 int로 변환한다  
static int \_charToHex(char target);  
// format 2 --> operand에서 r1, r2다 (int) 를 찾아낸다  
static void \_find\_r1r2();  
// format 3/4 --> operand에서 first, second operand (char\*) 를 찾아낸다  
static void \_split\_operands();  
  
int assemble(char \*file) { ... }

1. symtab.c, symtab.h : symbol table을 구현하는 코드가 포함되어 있습니다.

아래는 symtab.h에 선언된 struct와 functions입니다.

#ifndef SYMTAB\_H  
#define SYMTAB\_H  
  
#define MAX\_SYMBOL\_SIZE 30  
  
/\*\*  
 \* SYMTAB (Symbol Table) 구조체 선  
 \*/  
typedef struct \_symtab\_node \*symtab\_node;  
struct \_symtab\_node {  
 char symbol[MAX\_SYMBOL\_SIZE];  
 int address;  
 symtab\_node next\_node;  
};  
  
/\*\*  
 \* SYMTAB을 초기화한다. 기존에 SYMTAB이 있다면 모든 노드를 free한다.  
 \*/  
void \_init\_symtab(void);  
  
/\*\*  
 \* 새로운 symbol을 해당 symbol의 address와 함께 SYMTAB에 추가한다.  
 \* 중복 검사가 별도로 이루어지지 않으므로 아래 find\_symbol()을 이용해  
 \* duplication을 검출해야 한다  
 \*/  
void \_add\_symbol(char \*symbol, int address);  
  
/\*\*  
 \* symbol 이름으로 symbol node를 탐색한다.  
 \* 만약 없다면 NULL을 리턴한다.  
 \*/  
  
symtab\_node find\_symbol(char \*symbol);  
  
/\*\*  
 \* 전체 symbol list를 각 symbol에 맞는 address와 함께 출력한다.  
 \*/  
int show\_symbol(void);  
  
#endif

SYMTAB은 단방향 링크드 리스트로 구현하였고, head node만 포인터를 유지했습니다. 따라서 새로운 symbol 입력은 O(n) 의 시간복잡도를 지닙니다. 또한 `\_add\_symbol()`을 사용할 때 사전 순서를 유지하도록 node를 추가했습니다. 이는 `show\_symbol` 때 사전 순으로 출력하기 위함입니다.

1. optab.c, optab.h : opcode table을 구현하는 코드가 포함되어 있습니다. 프로젝트 1에서 작성한 코드지만, 프로젝트 2를 위해 다음과 같은 함수가 새로 생겨났습니다. 내부 구현 (해쉬 테이블) 에는 변화가 없습니다.

아래는 optab.h에 추가된 public function입니다.

/\*\*  
 \* Get target which has a given mnemonic  
 \*/  
optab\_node get\_optab\_node(char \*mnemonic);

1. 파싱 전략

프로젝트 2에서 가장 힘들었던 코드 작업은 파일을 읽어서 적절하게 분류하는, 파싱 파트였습니다. 저는 .asm 파일을 읽는 부분을 굉장히 중요하게 생각했는데, 왜냐하면 .asm 파일은 제가 수정할 수가 없기 때문입니다. 반면 pass1의 결과물로서 만들어지는 .itm 파일은 온전히 제가 만드는 것이므로 원하는 만큼 임의의 값으로 빈칸을 채워줄 수 있습니다. 따라서 가장 중요한 것은 .asm 파일을 문제없이 읽는 작업이었습니다.

int \_parse\_asm\_line() {  
 const char \*WORD = " \t", \*LINE = "\0";  
  
 if (fgets(line, sizeof(line), fp\_asm) == NULL) return 1;  
 else if (line[0] == '\n') return 1; // consider as EOF  
  
 line[strlen(line) - 1] = '\0'; // delete '\n'  
  
 // Step 1 : Get symbol (1st slot)  
 if (line[0] == '.') {  
 // comment line  
 symbol = line;  
 return 0;  
 } else if (line[0] == ' ') {  
 // no symbol  
 symbol = NULL;  
 } else {  
 // symbol  
 symbol = strtok(line, WORD);  
 }  
  
 // Step 2 : Get order (opcode or directive) (2nd slot)  
 if (symbol) order = strtok(NULL, WORD);  
 else order = strtok(line, WORD);  
  
 // Step 3 : Get operand (3rd slot)  
 operand = strtok(NULL, LINE);  
 while (operand && \*operand == ' ' && \*(operand + 1) != '\0') operand += 1;  
 if (operand && operand[0] == ' ') operand = NULL;  
  
 return 0;  
}

그래서 다음과 같은 전략을 세웠습니다.

* 읽을 라인이 있는지 판단한다. 없으면 return 1
* 주석인지 판단한다. 주석일 경우엔 symbol 포인터를 해당 라인 텍스트 전체를 담는데에 사용하고 return 0
* 주석이 아닐 경우엔 해당 라인의 첫 칸이 공백인지 판단한다. 공백이 아니라면 symbol이 존재한다고 판단하여 공백 기준으로 라인을 잘라 symbol 포인터에 보관한다. symbol이 없을 경우 symbol = NULL
* instruction(저는 order라는 키워드를 사용했습니다)은 order에 보관한다. instruction은 주석이 아닌 라인에 한하여 반드시 존재한다고 말할 수 있다.
* 이후 operand가 존재하는지 판단하여 있을 경우 operand 포인터에 해당 내용을 담고 없다면 operand는 NULL 처리한다. return 0

위 과정을 거쳐 [symbol, ] order [, operand]의 칼럼 구분을 완벽하게 구현 했습니다.

이후엔 intermediate file을 작성할 때 NULL인 포인터를 물결(“~”) 캐릭터로 대체하여 placeholder 역할을 수행하도록 했습니다. 이 과정을 통해 pass2에서 읽어올 때 조금 더 수월하게 파싱을 진행할 수 있었습니다.

이상으로 프로젝트2 보고서를 마칩니다.