시스템 프로그래밍 프로젝트 #2

- 과 목 명 : 시스템 프로그래밍
- 담당 교수님 : 조용윤
- 학 과 : 인공지능공학부
- 학 번 : 20224254
- 이 름 : 장가연

목차

I. 설계 구현 아이디어

i. 프로젝트 목표

본 프로젝트는 SIC 어셈블러 개발로, 소스 프로그램을 입력받은 후 토큰 단위로 분리(Parsing)한 후 명령어에 해당하는 OPCODE 매핑

ii. 데이터 구조 설계

- Instruction Table(명령어 테이블) : inst_unit 구조체와 inst_table 배열을 사용하여 inst.data.txt 파일의 정보를 메모리에 로드
- Token Table(소스코드 토큰 테이블) : token_unit 구조체와 token_table 배열을 사용하여 입력 파일의 각 라인을 Label, Operator, Operand 등으로 분리하여 관리
  -> Location Counter 와 기계어 코드 생성을 위한 기반

iii. parsing 로직의 핵심 아이디어 : 레이블 유무 판별

Label 필드는 생략될 수 있다는 점을 처리하는 것이 핵심이었고, 이 과정에서 오류를 많이 겪었다. 처음에는 Label이 있는 라인은 잘 처리하지만, Label이 없는 라인을 unkown으로 처리해버렸다. 이를 해결하기 위해 단순한 토큰 지정 대신, Operation Table을 활용한 조건부 피싱 로직을 도입했다.

1) 토큰 분리

- strtok() 함수를 사용하여 라인을 공백/탭 rlwnss으로 분리하여 최대 4개 토큰으로 분리

2) 레이블 유무 판단

- case 1(Label X)

: 첫 번째 토큰이 명령어 테이블에 존재하거나 지시문이라면, 해당 토큰을 Operator로 확정하고, 이후 토큰을 Operand로 처리

- case 2(Label O)

: 첫 번째 토큰이 Operator이 아니라면 첫 번째 토큰을 Label로, 두 번째 토큰을 Operator로 처리한 후 이후 토큰을 Operand로 처리

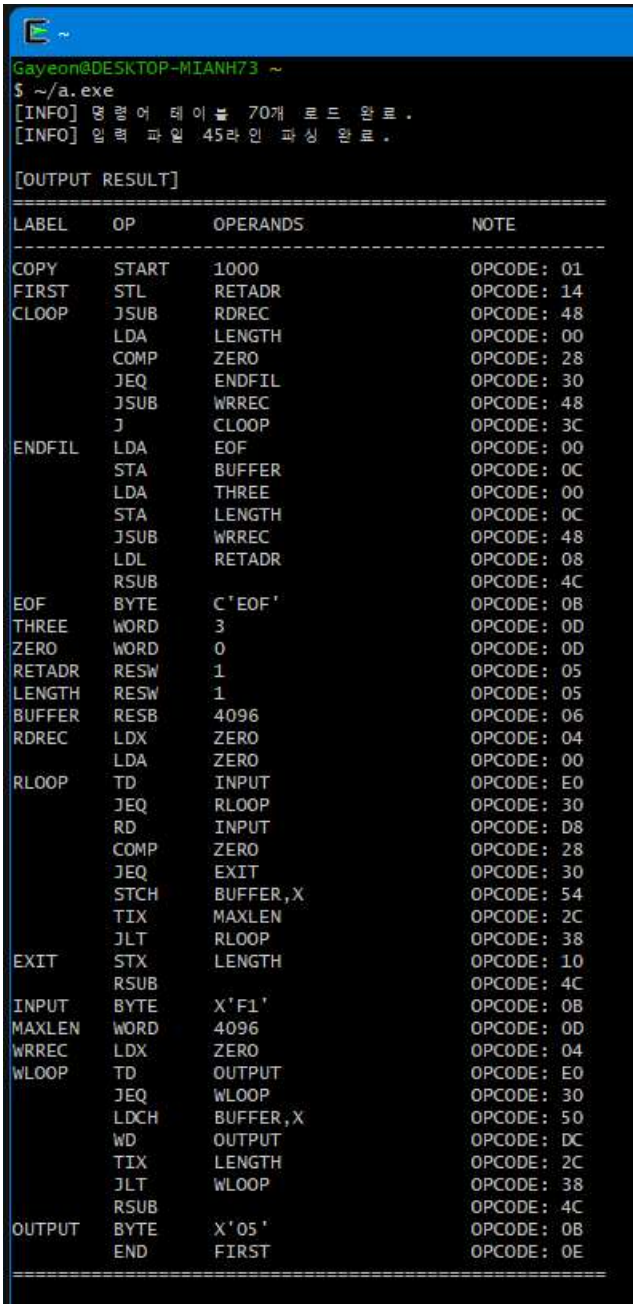
iv. operand 처리

- 색인 주소 지정 처리

: Operand 필드 내에 (,)가 포함된 경우(예: BUFFER, X)에 strchr()함수를 사용하여 콤마를 식별하였다. 이를 통해 BUFFER을 Operand1, X를 Operand2로 분리하여 token_unit의 operand[0]과 operand[1]에 저장할 수 있도록 하였다.

II. 수행 결과

input.asm 파일을 성공적으로 처리하고, 각 명령어에 대한 OPCODE 매핑 결과를 출력



```
~
Gayeon@DESKTOP-MIANH73 ~
$ ~/a.exe
[INFO] 명령어 테이블 70개 로드 완료.
[INFO] 입력 파일 45라인 파싱 완료.

[OUTPUT RESULT]
================================================
LABEL    OP       OPERANDS              NOTE
------------------------------------------------
COPY     START    1000                  OPCODE: 01
FIRST    STL      RETADR                OPCODE: 14
CLOOP    JSUB     RDREC                 OPCODE: 48
         LDA      LENGTH                OPCODE: 00
         COMP     ZERO                  OPCODE: 28
         JEQ      ENDFIL                OPCODE: 30
         JSUB     WRREC                 OPCODE: 48
         J        CLOOP                 OPCODE: 3C
ENDFIL   LDA      EOF                   OPCODE: 00
         STA      BUFFER                OPCODE: 0C
         LDA      THREE                 OPCODE: 00
         STA      LENGTH                OPCODE: 0C
         JSUB     WRREC                 OPCODE: 48
         LDL      RETADR                OPCODE: 08
         RSUB                           OPCODE: 4C
EOF      BYTE     C'EOF'                OPCODE: 0B
THREE    WORD     3                     OPCODE: 0D
ZERO     WORD     0                     OPCODE: 0D
RETADR   RESW     1                     OPCODE: 05
LENGTH   RESW     1                     OPCODE: 05
BUFFER   RESB     4096                  OPCODE: 06
RDREC    LDX      ZERO                  OPCODE: 04
         LDA      ZERO                  OPCODE: 00
RLOOP    TD       INPUT                 OPCODE: E0
         JEQ      RLOOP                 OPCODE: 30
         RD       INPUT                 OPCODE: D8
         COMP     ZERO                  OPCODE: 28
         JEQ      EXIT                  OPCODE: 30
         STCH     BUFFER,X              OPCODE: 54
         TIX      MAXLEN                OPCODE: 2C
         JLT      RLOOP                 OPCODE: 38
EXIT     STX      LENGTH                OPCODE: 10
         RSUB                           OPCODE: 4C
INPUT    BYTE     X'F1'                 OPCODE: 0B
MAXLEN   WORD     4096                  OPCODE: 0D
WRREC    LDX      ZERO                  OPCODE: 04
WLOOP    TD       OUTPUT                OPCODE: E0
         JEQ      WLOOP                 OPCODE: 30
         LDCH     BUFFER,X              OPCODE: 50
         WD       OUTPUT                OPCODE: DC
         TIX      LENGTH                OPCODE: 2C
         JLT      WLOOP                 OPCODE: 38
         RSUB                           OPCODE: 4C
OUTPUT   BYTE     X'05'                 OPCODE: 0B
         END      FIRST                 OPCODE: 0E
================================================
```

초기 구현에서 어려움을 겪었던 부분은 앞서 말한 것과 마찬가지로 Label이 없는 라인에서 Operator과 Label의 역할이 뒤바껴 제대로 처리하지 못하는 점을 다음과 같이 해결하였다.

오류 유형	라인	초기 오류 결과	해결 방법
파싱 오류	LDA LENGTH	LDA가 Label로, Length가 Operator로 인식됨.	find_opcode함수를 활용하여 LDA가 Operator인지 먼저 검사하는 로직 도입
결과 오류	LENGTH가 Operator	LENGTH가 명령어 테이블에 없어 Unknown Instruction으로 처리	find_opcode 로직 도입 후, LDA가 Operator로 인식되어 정상 출력

III. 결론 및 보충할 점

i. 결론

본 프로젝트를 통해 SIC 소스 코드를 입력으로 받아 토큰 단위로 정확하게 파싱하고, 명령어를 OPCODE와 OPERAND로 매핑하는 과정을 구현하며 이해도를 높일 수 있었다. 특히 Operation Table 기반의 동적 필드 식별 로직은 어셈블러 파싱의 핵심 원리를 깨닫고 학습하는 데에 있어 큰 도움이 되었다.

수업시간 및 개인 공부를 하며 학습했던 핵심적인 개념들을 프로젝트를 통해 확고하게 다질 수 있었다.

첫째로, 기계적 문자열 처리로 여겼던 토큰 분리(Parsing)가 실제로는 OPCODE Table을 참조하여 명령어의 필드(Label, Operator, Operand)를 식별하고 정의하는 과정임이, 이론 공부를 할 땐 잘 와닿지 않았는데 구체적인 코드 실습을 통해 매커니즘을 체득할 수 있었다.

둘째로, SIC 명령어 형식(Label 유무에 따른)의 처리를 통해 SIC 어셈블리 언어의 문법적 특성을 다시 한 번 학습할 수 있었다.

ii. 보충할 점

1) Operand 콤마 분리 로직 정교화

[BUFFER, X]와 같은 인덱스 주소 지정 Operand의 분리된 결과가 최종 출력 단계에서는 결합되어 출력되고 있다. 이를 분리 저장하여 인덱스 비트를 설정하는 데 활용할 수 있도록 수정이 필요하다.

2) 에러 구체화

현재는 파일 오류만 처리하고 있고, 그 외의 오류는 Unknown으로 처리하고 있는데 이를 구체화하여 '존재하지 않는 명령어 사용, 잘못된 Operand 개수 등' 어셈블리 문법적 오류를 탐지하여 사용자에게 보고하는 로직을 추가하면 오류를 직관적으로 확인 및 수정에 용이할 것이다.

IV. 소스코드(+주석)

my_assembler.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifdef _WIN32
  #define STRICMP _stricmp
#else
  #include <strings.h>
  #define STRICMP strcasecmp
#endif

#define MAX_LINES 5000
#define MAX_OPERAND 3 // 한 명령어 당 최대 3개의 operand
#define MAX_INST 256

// ---------------- 구조체 정의 ----------------
struct token_unit {
    char *label;                          // 명령어 라인 중 label(symbol(예 : CLOOP, ENDFIL))
    char *operator;                       // 명령어 라인 중 operator(LDA, JSUB 등0)
    char operand[MAX_OPERAND][50];        // 명령어 라인 중 operand(피연산자 목록(LENGTH,
ZERO 등))
    char comment[200];                    // 명령어 라인 중 comment(주석)
};
typedef struct token_unit token;  // 파싱된 토큰 정보를 라인별로 관리하는 테이블
token *token_table[MAX_LINES];  // 원본 소스 코드를 라인별로 저장하는 테이블
char *input_data[MAX_LINES]; //현재까지 읽기/파싱한 라인의 수
static int line_num = 0;

// 명령어 테이블 구조체
struct inst_unit {
    char str[16];           // instruction의 이름
    unsigned char op;       // 명령어의 OPCODE(16진수)
    int format;             // instruction의 형식(FORMAT 1,2,3)
    int ops;                // operand 개수
};
typedef struct inst_unit inst;
inst *inst_table[MAX_INST];  // 명령어 정보를 관리하는 테이블
int inst_index = 0; // 현재 로드된 명령어의 개수

// ---------------- 함수 선언 ----------------
void load_inst_table(const char *filename);  // inst.data.txt 파일 로드
```

```
void parse_input(const char *filename);  // input.asm 파일 파싱
unsigned char find_opcode(const char *mnemonic); // 주어진 명령어 이름으로 OPCODE 찾기
int is_directive(const char *mnemonic);  // 주어진 이름이 어셈블러 지시문인지 판별
void print_result(void);  // 파싱 결과 출력하는 함수
void free_all(void);  // 프로그램 종료 시 동적 할당된 메모리 해제

// ---------------- main ----------------
int main(void) {
    load_inst_table("inst.data.txt"); // inst.data.txt 파일 로드->명령어 테이블 =생성
    parse_input("input.asm"); // input.asm 파일 파싱->토큰 테이블 생성
    print_result();
    free_all();
    return 0;
}

// ---------------- 구현부 ----------------

// inst.data.txt 로드
void load_inst_table(const char *filename) {
    FILE *fp = fopen(filename, "r");
    if (!fp) {
        fprintf(stderr, "Error: %s 파일을 열 수 없습니다.\n", filename);
        exit(1);
    }

    char name[16], type[8], opcode_str[16];
    int format;
    inst_index = 0;

    //명령어 테이블 로드(이름, 타입, 포맷, OPCODE(16진수 문자열))
    while (fscanf(fp, "%15s %7s %d %15s", name, type, &format, opcode_str) == 4) {
        inst *new_inst = (inst *)malloc(sizeof(inst));
        if (!new_inst) { fprintf(stderr, "malloc failed\n"); exit(1); }

        // 명령어 이름 저장
        strncpy(new_inst->str, name, sizeof(new_inst->str)-1);
        new_inst->str[sizeof(new_inst->str)-1] = '\0';
        new_inst->op = (unsigned char)strtol(opcode_str, NULL, 16);

        // 문자열 형태의 16진수를 숫자로 변환해서 저장
        new_inst->format = format;
        new_inst->ops = 1;
        inst_table[inst_index++] = new_inst;
```

```
        if (inst_index >= MAX_INST) break;
    }
    fclose(fp);
    printf("[INFO] 명령어 테이블 %d개 로드 완료.\n", inst_index);
}

// opcode 찾기 (대소문자 무시)
// 찾고자 하는 명령어 이름
// 해당 명령어의 OPECODE 반환, 없으면 0xFF 반환
unsigned char find_opcode(const char *mnemonic) {
    if (!mnemonic) return 0xFF;
    // inst_table 순회하며 mnemonic과 일치하는지 확인
    for (int i = 0; i < inst_index; i++) {
        if (STRICMP(inst_table[i]->str, mnemonic) == 0)
            return inst_table[i]->op;
    }
    return 0xFF;
}

// pseudo directive 판별
// 주어진 이름이 SIC 어셈블러 지시문인지 판별(지시문이면 1, 아니면 0 반환)
int is_directive(const char *mnemonic) {
    if (!mnemonic) return 0;
    const char *dirs[] = {"START","END","WORD","RESW","RESB","BYTE",
                          "EQU","LTORG","CSECT","EXTDEF","EXTREF","BASE", NULL};
    for (int i = 0; dirs[i] != NULL; i++) {
        if (STRICMP(dirs[i], mnemonic) == 0) return 1;
    }
    return 0;
}

// input.asm 토큰 단위로 파싱
void parse_input(const char *filename) {
    FILE *fp = fopen(filename, "r");
    if (!fp) {
        fprintf(stderr, "Error: %s 파일을 열 수 없습니다.\n", filename);
        exit(1);
    }

    char linebuf[1024];
    line_num = 0;

    while (fgets(linebuf, sizeof(linebuf), fp)) {
```

```
// 1. 개행문자 제거(\n, \r\n)
linebuf[strcspn(linebuf, "\r\n")] = 0;

// 2. 라인의 시작 부분에서 공백/탭을 건너뛰고 주석이나 빈 줄인지 확인
char *p = linebuf;
while (*p == ' ' || *p == '\t') p++;
if (*p == '.' || *p == '\0') continue;

// 3. 토큰화 및 구조체에 저장
token *tk = (token *)malloc(sizeof(token));
if (!tk) { fprintf(stderr, "malloc failed\n"); exit(1); }
memset(tk, 0, sizeof(token));

// 4. strtok 사용 위해 라인 복사(in 버퍼)
char temp[1024];
strncpy(temp, linebuf, sizeof(temp)-1);
temp[sizeof(temp)-1] = '\0';

// 5. 라인을 공백/탭을 구분자로 사용하여 토큰화(최대 4개)
char *t1 = NULL, *t2 = NULL, *t3 = NULL, *t4 = NULL;
t1 = strtok(temp, " \t");
if (t1) t2 = strtok(NULL, " \t");
if (t2) t3 = strtok(NULL, " \t");
// 남은 부분은 주석일 수 있으므로 개행문자까지 포함
if (t3) t4 = strtok(NULL, "\n");

// 6. 토큰의 의미(Label, Operator, Operand) 판별 및 저장
// case 1: Label이 없는 경우(첫 번째 토큰이 명령어(operation) 또는 지시문(directive))
if (t1 && (find_opcode(t1) != 0xFF || is_directive(t1))) {
    tk->label = NULL;
    tk->operator = strdup(t1);
    if (t2) strncpy(tk->operand[0], t2, sizeof(tk->operand[0])-1);
    if (t3) strncpy(tk->operand[1], t3, sizeof(tk->operand[1])-1);
    if (t4) {
        while (*t4 == ' ' || *t4 == '\t') t4++;
        strncpy(tk->operand[2], t4, sizeof(tk->operand[2])-1);
    }
}
// case 2 : Label이 있는 경우(첫 번째 토큰이 심볼(label))
else {
    tk->label = t1 ? strdup(t1) : NULL;
    tk->operator = t2 ? strdup(t2) : NULL;
    if (t3) strncpy(tk->operand[0], t3, sizeof(tk->operand[0])-1);
```

```
            if (t4) {
                while (*t4 == ' ' || *t4 == '\t') t4++;
                strncpy(tk->operand[1], t4, sizeof(tk->operand[1])-1);
            }
        }

        // 7. 원본 라인 및 파싱 결과 저장
        input_data[line_num] = strdup(linebuf);
        token_table[line_num] = tk;
        line_num++;
        if (line_num >= MAX_LINES) break;
    }

    fclose(fp);
    printf("[INFO] 입력 파일 %d라인 파싱 완료.\n", line_num);
}

// 결과 출력
// OPCODE가 있는 명령어와 지시문을 구분하여 보여줌
void print_result(void) {
    printf("\n[OUTPUT RESULT]\n");
    printf("=================================================\n");
    printf("%-8s %-8s %-22s %-16s\n", "LABEL", "OP", "OPERANDS", "NOTE");
    printf("-------------------------------------------------\n");

    for (int i = 0; i < line_num; i++) {
        token *tk = token_table[i];
        if (!tk) continue;
        const char *lbl = tk->label ? tk->label : "";
        const char *op = tk->operator ? tk->operator : "";


        // 모든 operand들을 하나의 문자열로 결합(,로 구분)
        char ops_combined[200] = "";
        int printed = 0;
        for (int k = 0; k < MAX_OPERAND; k++) {
            if (tk->operand[k][0]) {
                if(printed)strncat(ops_combined,                    ",                    ",
sizeof(ops_combined)-strlen(ops_combined)-1);
                strncat(ops_combined,                                   tk->operand[k],
sizeof(ops_combined)-strlen(ops_combined)-1);
                printed = 1;
            }
```

```
        }

        // operator가 없는 경우(Label만 있는 경우)
        if (op[0] == '\0') {
            printf("%-8s %-8s %-22s %-16s\n", lbl, "", "", "");
            continue;
        }

        // OPCODE 찾기
        unsigned char opcode = find_opcode(op);
        // OPCODE가 있는 명령어인 경우(일반적인 SIC/XE 명령어)
        if (opcode != 0xFF) {
            printf("%-8s %-8s %-22s OPCODE: %02X\n", lbl, op, ops_combined, opcode);
        }
        // OPCODE가 없지만 지시문 목록에 존재(START, END 등)
        else if (is_directive(op)) {
            printf("%-8s %-8s %-22s %-16s\n", lbl, op, ops_combined, "DIRECTIVE");
        }
        // 알 수 없는 명령어인 경우(오타/지원하지 않는 명령어)
        else {
            printf("%-8s %-8s %-22s %-16s\n", lbl, op, ops_combined, "Unknown
Instruction");
        }
    }
    printf("=========================================================\n");
}

// 메모리 해제
void free_all(void) {
    for (int i = 0; i < line_num; i++) {
        if (token_table[i]) {
            if (token_table[i]->label) free(token_table[i]->label);
            if (token_table[i]->operator) free(token_table[i]->operator);
            free(token_table[i]);
            token_table[i] = NULL;
        }
        if (input_data[i]) {
            free(input_data[i]);
            input_data[i] = NULL;
        }
    }
    for (int i = 0; i < inst_index; i++) {
        if (inst_table[i]) {
```

```
                free(inst_table[i]);
                inst_table[i] = NULL;
            }
        }
}
```

V. 깃허브 url

https://github.com/JangGayeon/system_programming.git