Mobile Processor

# Simple Calculator

project #1

32171550 박다은

2020 Spring

Freedays left: 4

# Contents

# 1.Project Overview

## (1) Project Goal

CPU 의 작동을 따르는 계산기를 만들어보며 CPU 내부의 연산 명령어 작동을 이해한다. 평소에 자주 사용하던 visual 환경을 벗어나 Linux 환경에서 프로그래밍을 해보며 작동 방법을 익히고 숙지한다.

## (2) Feature of Simple Calculator

CPU 내부 연산 명령어를 모방하고 10 개의 레지스터와 9 개의 명령어를 갖는 간단한 ISA 로 구성된다.

ISA(Instruction Set Architecture): 마이크로프로세서가 인식해서 기능을 이해하고 실행할 수 있는 기계어 명령어를 말한다.

## (3) Composition of Instruction

| OP code | Operand |
|---|---|

OP Code: 명령어에서 연산 동작을 지정하는 부분으로 명령어의 종류를 표현한다.

Operand: 연산을 실행하는 데 필요한 데이터 혹은 주소 값을 포함한다.

## (4) Program Structure

Open_file

+-----------------------

| Print_line

| Print_opcode

| distinguish_operand()

| Execution

+-----------------------

각 단계에서 각자의 역할은 결과를 출력하는 것이다.

Print_line 단계에서는 출력되는 단계의 번호를 [n]으로 나타낸다.

Print_opcode 단계에서는 입력된 Op Code 를 출력한다.

distinguish_operand() 단계에서는 Operand 의 형태를 구분하고(숫자 or 레지스터) 각각의 값을 출력한다.

마지막으로, Execution 단계에서는 실행된 연산 결과를 출력한다.

### (5) Build Environment

Compilation: Linux Assam, with GCC

# 2. Project Introduction

다음 연산들을 실행하는 간단한 계산기를 만든다.

### (1) 기본구현

각 연산들의 괄호'()' 속 기호는 Op Code 를 나타낸다.

- 기본 사칙연산: Plus(+), Minus(-), Multiply(*), Divide(/)

  ※결과값이 R0 에 저장된다.

- Move(M): Operand2 의 값을 Operand1 에 저장한다.
- Halt(H): 계산기 실행을 중단한다.

### (2) 추가구현

- Jump(J): input 파일 내의 Operand 값 line 으로 점프한다.
- Compare(C): Operand1 과 Operand2 의 크기를 비교한다.

Operand1 이 Operand2 보다 크거나 같으면 R0 에 0 이 저장되고, 그렇지 않으면 1 이 저장된다.

- Branch(B): R0 이 1 이면 input 파일 내의 Operand 값 line 으로 점프한다.
- GCD(G): Operand1 과 Operand2 의 최대공약수를 구한다.

# 3. Code & Consideration

## (1) 주요 함수

### ① getline() | sseize_t getline(char **lineptr, size_t *n, FILE *stream);

stream 의 전체 line 을 읽고, 버퍼의 주소를 *lineptr 에 저장한다. 버퍼는 NULL 로 종료된다. 만약 함수 호출 전에 *lineptr 이 NULL 이고, *n 이 0 이라면, getline()은 line 을 저장할 버퍼를 할당한다. getline()이 line 을 읽는 것을 실패하면 -1 을 반환한다. 사용한 후에 free()함수를 통해 동적으로 할당되었던 메모리를 시스템에 반납하여야한다.

### ② strtok() | char *strtok(char *str, const char *delim);

strtok()함수는 문자열 token 을 분리하는 함수이다. *delim 은 token 을 분리하는 분리자이다.

### ③ atoi() | int atoi(const char *nptr);

str 형을 int 형으로 변환시켜주는 함수이다.

### ④ strtol() | long int strtol(const char *nptr, char **endptr, int base);

받은 문자열을 여러 진수로 표현할 수 있다. endptr 이 NULL 이 아닌경우 strtol()함수는 유효하지 않은 첫 번째 문자의 주소를 *endptr 에 저장한다.

atoi(nptr)은 strtol(nptr, NULL, 10)과 같다.

## (2) 주요 코드

## ① operand Structure

```
20 struct operand{
21         char *opr;
22         int n;
23         int index;
24 }opr1, opr2;
```

- operand1 과 operand2 는 같은 형식으로 이루어져 있고 프로그램 내에서의 사용 방식도 동일하다. 따라서 opr1 과 opr2 를 한 구조체에서 각각 선언해주었다.

## ② Written Functions

### ➢ gcd Function

```
134 int gcd(int a, int b){
135         while( a != b){
136                 if(a> b)
137                         a -= b;
138                 else
139                         b -= a;
140                 }
141         return a;
142 }
```

- a 와 b 의 최대공약수를 구하는 함수이다. a 와 b 가 같은 값을 가질 때까지 뺄셈을 하면 최대공약수를 구할 수 있게 된다.

### ➢ jump Function

```
144 void jump(int num){
145         printf("\n->Jump to line%d\n", num);
146         if(num > p_index){
147                 for(; num >  p_index+1 ; p_index++)
148                 getline(&line, &n, fp);
149         }
150         else{
151                 free(line);
152                 line = NULL;//getline()두 번 째 사 용 부 터 는  str(line)->NULL이
    므 로
153                 fclose(fp);
154
155                 fp = fopen("input.txt", "r");
156                 for(int i = 1; i < num; i++){
157                         getline(&line, &n, fp);
158                 }
159         }
160 }
```

- Jump 와 Branch 의 실행 방식은 거의 동일하다. 따라서 main 문 내에서 불필요한 중복을 막기 위해 jump 함수를 만들어주었다. 함수가 실행되면 먼저 operand 의 값을 받아와 "₩n->Jump to line%d₩n"을 출력한다. 여기서 %d 에 들어갈 정수 값은 operand 의 값이다.

- operand 의 값과 출력되는 index 값을 비교해 operand 의 값이 index 값보다 클 때와 작거나 같을 때를 나눠주었다. operand 의 값이 index 값보다 클 때는 상관이 없는데, operand 값이 index 값보다 작으면 무한루프가 생길 수도 있기 때문이다.

- operand 의 값이 index 값보다 클 때는 for 문에서 operand 의 값이 index 의 값과 같아질 때까지 getline 함수를 통해 line 을 읽는다. 따라서, 반복문이 종료된 후에 원하는 위치로 점프할 수 있다.

- operand 의 값이 index 값보다 작거나 같다면 파일을 종료하고 다시 파일을 불러온다. 이 과정에서 getline 함수의 특성에 따라 line 을 NULL 로 초기화해야 한다. 파일을 불러온 후 for 문을 통해 파일의 맨처음부터 다시 line 을 읽어 원하는 위치에 도달했으면 반복문에서 벗어나게 해주었다.

### ➢ dis_operand Function

```
162 int dis_operand(char *oprnd, int a){
163         int num;
164                 //J or B
165                 if(*oprnd >= '1' && *oprnd <= '9'){
166                         num = atoi(oprnd);
167                         if(a == 1)  printf("n_opr1: %d ", num);
168                         //else if(a == 2) printf("n_opr2: %d\n", num);
169                         return num;

170                 }
171                 //operand의 prefix가 0x인 16진 수 일 때
172                 else if(*(oprnd+1) == 'x' || *oprnd == 'X'){
173                         num = (int)strtol(oprnd+2,NULL,16);
174                         if(a == 1)  printf("n_opr1: 0x%x | ", num);
175                         else if(a == 2) printf("n_opr2: 0x%x\n", num);
176                         return num;
177                 }
178                 //operand가 Register일 때
179                 else if(*oprnd == 'R' || *oprnd == 'r'){
180                                 if(a == 1){
181                                         opr1.index = atoi(oprnd+1);
182                                         num = Reg[opr1.index];
183                                          printf("R_opr1: R%d (val: %x) |
    " , opr1.index, num);
184                                         return Reg[opr1.index];
```

```
185                                 } else if(a == 2){
186                                         opr2.index = atoi(oprnd+1);
187                                         num = Reg[opr2.index];
188                                         printf("R_opr2: R%d (val: %x)\n",
    opr2.index, num);
189                                         return Reg[opr2.index];
190                                 }
191                         } else return 0; //error mal-formatted number
192 }
```

- operand 가 레지스터인지 혹은 10 진수나 16 진수인지 구별한 후에 각각의 operand 값을 출력하는 if-else-if 문을 원래 main 문에 뒀었다.(3-3-①에서 자세히 설명하도록 하겠다.) 하지만 내용이 길기도 하고, operand1 과 operand2 에서 비슷한 부분이 두 번 반복되길래 함수로 만들어주었다.

- 첫 if 문인 if(*oprnd >= '1' && *oprnd <= '9')는 오직 op code 가 J 나 B 일 때에만 쓰이도록 나눈 것이다. ‘oprnd == 0’이면 0x 로 시작하는 16 진수도 포함되므로 1 부터 9 까지로 범위를 나누었다. 0 번째 줄이란 존재하지 않으므로, Jump 와 Branch 의 성질과도 어긋나지 않아 0 을 제외해도 상관이 없다.

  Jump 와 Branch 는 operand1 만 가지므로, a 가 1 일 때의 경우만 생각했다. 여기서 a 값은 operand1 이면 1, operand2 면 2 를 갖도록 main 문에서 설정해주었다.

- 두번째 if 문인 else if(*(oprnd+1) == 'x' || *(oprnd+1) == 'X')를 통해 operand 의 prefix 가 0x 인 16 진수를 구별할 수 있게 했다.

  먼저 strtol 함수를 통해 문자열의 16 진수를 정수형의 16 진수로 바꿔주었다. 이후엔 첫 if 문과 같은 방법으로 출력하도록 하였다. 다만, 16 진수가 사용되는 기본 사칙연산들 (‘+’, ’_’, ’/’, ’*’)과, Compare, GCD 는 operand 를 두개 가지므로 또 한번의 if 문을 통해 각각의 출력을 다르게 형성했다.

- 마지막 if 문인 else if(*oprnd == 'R' || *oprnd == 'r')은 operand 가 레지스터 형식일 때 실행된다.

  먼저 레지스터의 index 값(예를 들어 R1 이면 index 값은 1 이다.)을 atoi 함수를 통해 정수 값으로 받아주었다. 그 후 그 index 에 맞는 배열의 값을 출력 값으로 지정하였다.

## ③ main 문

### a. argument

```
26 int  main(int argc, char* argv[]){
27
28         /*printf("argc: %d, argv[0]: %s, argv[1]: %s \n",
29                 argc, argv[0], argv[1]);*/
```

- argc 는 argument count 의 준말이고, argv 는 argument vextor 의 준말이다. 여기서 argument count 는 메인 함수에 전달되는 정보의 개수이고, argument vector 은 메인함수의 전달되는 실질적 정보(문자열의 배열)이다.

- 만약 input 이 ./hello 이면 argc: 1, argv[0]: ./hello 이고,
  input 이 ./hello 123 이면 argc: 2, argv[0]: ./hello, argv[1]: 123 이다.

  vector 의 인자는 띄어쓰기를 통해 나뉜다고 생각하면 된다.

- printf 를 통해 위의 사실을 확인할 수 있지만, 프로그램 내의 중요한 부분은 아니라 주석처리 해주었다.

```
40          if(argc == 2)
41                  fp = fopen(argv[1], "r");//"r" : 읽 기 모 드
42          else
43                  fp = fopen("input.txt", "r");
```

- 현재 계산기 프로그램에선 'input.txt'파일이 필요하다. 따라서 argc 가 2 일 때와 아닐 때로 나누었다. argc 가 2 라면 사용자가 './프로그램이름 input.txt'라고 입력한 경우일 테니 argv[1]인 'input.txt'를 불러온다. 하지만 argc 가 2 가 아니라면, 사용자가 잘못 입력한 경우이므로, 강제로 'input.txt'파일을 열도록 하였다.

### b. strtok 함수 이용해 line 분리

```
50                  //parse the line
51                  //operator
52                  op = strtok(line, " \t\n");//분 리 자 :space,tap,newline
53
54                  //Halt
55                  if(*op == 'H') {
56                          printf("\n\n>>>>>>>>>>>>>>>>>>Halt Execution<<<<<<
<<<<<<<<<<\n\n\n");
57                          return 0;
58                  }
59
60                  printf("opcode: %s | ", op);
61
62                  //operand1
63                  opr1.opr = strtok(NULL, " \t\n");//두 번 째 부 터 는  str->NULL
64                  opr1.n = dis_operand(opr1.opr,first);
65
66                  //operand2
67                  if(*op != 'J' && *op != 'B'){
68                          opr2.opr = strtok(NULL, " \t\n");
69                          opr2.n = dis_operand(opr2.opr,second);
70                  }
```

- 분리자를 각각 space, tap, newline 으로 설정해 line 을 분리하도록 하였다.
- Halt 는 operand 를 가지지 않으므로 operand 전에 위치하도록 관련 내용을 작성하였다.

  (원래 3-(2)-③-c 에 나오는 switch 문에 Halt 를 작성하였는데 'segmentation fault'라는 오류가 발생해 디코딩을 해본 후 현재의 자리로 옮겨주었다.)
- 앞서 말했듯이 Jump 와 Branch 는 operand 를 하나만 갖는다. 따라서 67 번째 줄처럼 Jump 와 Branch 가 아닐 경우에만 operand2 를 가지도록 했다.

  (이도 Halt 와 마찬가지로 처음엔 따로 고려하지 않았다가 'segmentation fault'를 겪고 디코딩 후 바꿔주었다.)

## c. execution

execution 은 switch 문을 사용했다. 원래는 if 문을 사용했지만 추가구현을 통해 opcode 의 수가 증가하면서 코드가 복잡해져 후에 알아보기 편하도록 switch 문으로 변경해주었다.

➢ **기본 사칙연산**

```
73                  switch(*op){
74                          case '+' :
75                                  result = opr1.n + opr2.n;
76                                  printf("Reslut) %x := %x + %x\n",
77                                          result, opr1.n, opr2.n);
78                                  Reg[0] = result;
79                                  break;

80                          case '-' :
81                                  result = opr1.n - opr2.n;
82                                  printf("Result) %x := %x - %x\n",
83                                          result, opr1.n, opr2.n);
84                                  Reg[0] = result;
85                                  break;
86                          case '*' :
87                                  result = opr1.n * opr2.n;
88                                  printf("Result) %x := %x * %x\n",
89                                          result, opr1.n, opr2.n);
90                                  Reg[0] = result;
91                                  break;
92                          case '/' :
93                                  result = opr1.n / opr2.n;
94                                  printf("Result) %x := %x / %x\n",
95                                          result, opr1.n, opr2.n);
96                                  Reg[0] = result;
97                                  break;
```

- %x 를 통해 16 진수로 출력했다. 각각의 연산이 끝나고 나면 결과값을 R0 에 넣는다.

➢ GCD, Compare, Move

```
 98                          case 'G' :
 99                                  result = gcd(opr1.n, opr2.n);
100                                 printf("Result) %x = gcd(%x, %x)\n"
101                                         , result, opr1.n, opr2.n);
102                                 Reg[0] = result;
103                                 break;
104                         case 'C' :
105                                 if(opr1.n >= opr2.n){
106                                         result = 0;
107                                         printf("Result(R0)) %d\n", result
    );
108                                 }else{
109                                         result = 1;
110                                         printf("Result(R0)) %d\n", result
    );
111                                 }
112                                 Reg[0] = result;
113                                 break;
114                         case 'M' :
115                                 Reg[opr1.index] = Reg[opr2.index];
116                                 printf("Result) R%d(%x) = R%d\n",
117                                 opr1.index, Reg[opr2.index], opr2.index);
118                                 break;
```

- GCD 는 앞서 설명한 gcd 함수를 통해 실행시켰다.
- GCD 와 Compare 의 경우, 사칙연산과 마찬가지로 결과값을 R0 에 넣어주었다.
- operand2 의 값(R0 의 값 = 직전에 실행한 line 의 결과값)을 operand1 로 옮기도록 case 'M'을 구성하였다. 레지스터의 index 값은 10 진수값이므로 %d 로 작성하였고, 반면에 16 진수로 나타내야 하는 괄호 안의 부분(옮겨진 값)은 %x 로 작성하였다.

➢ Jump, Branch

```
119                         case 'J' :
120                                 jump(opr1.n);
121                                 break;
122                         case 'B' :
123                                 if(Reg[0] == 0) printf("\n->Continue\n");
124                                 if(Reg[0] == 1) jump(opr1.n);
125                                 break;
126                 }
```

- Jump 와 Branch 는 비슷한 실행을 갖는다. 따라서 두 연산 모두 jump 함수를 이용해 원하는 결과를 가져올 수 있도록 해주었다.

## (3) 실행 화면

➢ Input.txt 파일

```
+ 0x7 0x4
M R1 R0
* 0x2 0x5
M R2 R0
C R1 R2
B 12
G 0x6 0x1a
M R3 R0
- R1 R3
M R1 R0
J 5
H
```

➢ 실행 결과

```
>>>>>>>>>>>>>>>>>Hello World!<<<<<<<<<<<<<<<<<

[1] + 0x7 0x4
opcode: + | n_opr1: 0x7 | n_opr2: 0x4
Reslut) b := 7 + 4

[2] M R1 R0
opcode: M | R_opr1: R1 (val: 0) | R_opr2: R0 (val: b)
Result) R1(b) = R0

[3] * 0x2 0x5
opcode: * | n_opr1: 0x2 | n_opr2: 0x5
Result) a := 2 * 5

[4] M R2 R0
opcode: M | R_opr1: R2 (val: 0) | R_opr2: R0 (val: a)
Result) R2(a) = R0

[5] C R1 R2
opcode: C | R_opr1: R1 (val: b) | R_opr2: R2 (val: a)
Result(R0)) 0

[6] B 12
opcode: B | n_opr1: 12
->Continue

[7] G 0x6 0x1a
opcode: G | n_opr1: 0x6 | n_opr2: 0x1a
Result) 2 = gcd(6, 1a)

[8] M R3 R0
opcode: M | R_opr1: R3 (val: 0) | R_opr2: R0 (val: 2)
Result) R3(2) = R0

[9] - R1 R3
opcode: - | R_opr1: R1 (val: b) | R_opr2: R3 (val: 2)
Result) 9 := b - 2

[10] M R1 R0
opcode: M | R_opr1: R1 (val: b) | R_opr2: R0 (val: 9)
Result) R1(9) = R0

[11] J 5
opcode: J | n_opr1: 5
->Jump to line5

[12] C R1 R2
opcode: C | R_opr1: R1 (val: 9) | R_opr2: R2 (val: a)
Result(R0)) 1

[13] B 12
opcode: B | n_opr1: 12
->Jump to line12

[14] H

>>>>>>>>>>>>>>>>>Halt Execution<<<<<<<<<<<<<<<<<
```

## (3) Consideration

① First Code

```
38          char *opr1;//operand1
39          char *opr2;//operand2
40
41          int result;
42          int n1, n2;
```

```
44          int index1 = 0;
45          int index2 = 0;
```

- 기존 코드는 main 문 내에서 너무 많은 변수를 가졌었다. 변수의 이름도 비슷하고 사용하는 목적도 같아 쓸 데마다 헷갈렸고 오류도 수차례 떴다. 따라서 3-(2)-①처럼 구조체를 만들어주었다. 현재 코드는 main 문 내의 변수가 단 4 개로 사용하기에도 편리하고, 코드도 깔끔하다.

```
81                  //operand1
82                  opr1 = strtok(NULL, " \t\n"); //두 번 째 부 터 는  str->NULL
83                  if(*opr1 >= '0' && *opr1 <= '9'){
84                            n1 = atoi(opr1);
85                            printf("n_opr1: %d | ", n1);
86                  } else if(*opr1 == 'R' || *opr1 == 'r'){
87                                    R.index1 = atoi(opr1+1);
88                                    n1 = Reg[R.index1];

90                                                  , R.index1, n1);
91                          } else n1 = 0; //error mal-formatted number
92                  }
93                  //opernad2
94                  if(*op != 'J' && *op != 'B'){
95                          opr2 = strtok(NULL, " \t\n");
96                          // printf("opr2: %s \n", opr2);
97
98                          if(*opr2 >= '0' && *opr2 <= '9'){
99                                    n2 = atoi(opr2);
100                                   printf("n_opr2: %d\n", n2);
101                         } else if(*opr2 == 'R' || *opr2 == 'r'){
102                                           R.index2 = atoi(opr2+1);
103                                           n2 = Reg[R.index2];
104                                           printf("R_opr2: R%d (val: %d)\n"
105                                                           , R.index2, n2);
106                                   } else n2 = 0; //error mal-formatted number
107                  }
108 }
```

- 보다시피 opr1 과 opr2 라는 변수를 제외하고는 위의 if 문과 아래의 if 문이 99% 같은 양상을 띈다. 애초에 operand 의 유형을 구분하는 함수를 만들고 사용하려 했으나, 여러 번의 실패 끝에 위와 같은 코드를 만들었었다.

- 하지만, main 문이 너무 길어졌고 보기에도 불편하고 거슬려서 실패를 거듭하며 반복한 수차례의 시도 끝에 앞서 선보였던 dis_operand 함수를 만들어냈다. 함수가 원래 생각했던 것보다 길기도 하고 복잡해서 원했던 모양은 아니지만, 노력 끝에 만들어냈다는 게 굉장히 의미가 있고 뿌듯했다. 실제로 main 문의 내용은 10 줄 이상 줄어들었다.

## ② Hexadecimal

```
197 int InputHex(char *opr){
198
199         unsigned char num = 0, i;
200
201         for(i=2;*(opr+i)==' '||*(opr+i)=='\n',i++){
202                 if(*(opr+i) >= 'A' && *(opr+i) <= 'F')
203                         num = num*16+atoi(opr+i)-'A'+10;
204                 else if(*(opr+i) >= 'a' && *(opr+i) <= 'f')
205                         num = num*16+atoi(opr+i)-'a'+10;
206                 else if((*(opr+i) >= '0' && *(opr+i) <= '9')
207                         num = num*16+atoi(opr+i)-'0';
208         }
209         return num;
210 }
```

- 모든 구현을 10 진수로 완료한 뒤에 16 진수로 바꾸려고 코드를 짰다. 그래서 위와 같은 InputHex 함수를 작성했었다. 실패한 코드이지만 간략하게 설명을 해보자면, 16 진수의 입력은 0x__로 받아지므로, opr+2 부터 포인터가 space 나 newline 을 만날 때까지의 반복문을 만들었다. 오류는 나지 않았지만 함수 적용 후 출력이 되는 모든 16 진수가 0 으로 나왔다. 함수를 고치려고 인터넷을 찾아보던 중에 strtol 함수를 발견해 이 함수를 이용하는 방법으로 방향을 바꿨다.

## (4) Review

처음엔 putty 를 통하여 c 언어 코드를 작성하는 데에 어려움을 느꼈었다. 그래서 visual studio 로 작성할까라는 생각도 했지만, 앞으로 계속 visual 로 작성할 수는 없을 테이고 이번 기회에 putty 와 친해져보자는 생각으로 강의를 반복해 들으며 작성해 나갔었다. 물론 초반엔 visual 을 이용해 프로그래밍을 하는 것보다 많은 시간과 노력이 필요했고, 명령어가 익숙하지 않아 찾아보는 데에도 많은 시간을 들였었다. 하지만 코드를 하나 하나씩 완성해 나갈 때마다, 요구사항들을 하나 하나씩 충족시킬 때마다 뿌듯함은 이루어 말할 수 없었다. 물론 이를 완성시켜 나가는 데에는 문제점도 많았다. 큼지막한 문제들은 위에 따로 게시해 놨지만, 자그마한 문제들도 굉장히 많았다. if 문의 조건식을 '>'을 '<'로 잘못 작성한다든가, '=='을 '-'로 잘못 작성하는 어이없는 실수들도 여러 차례 행하였다. 굉장히 기초적이고 황당한 실수이지만 코드를 작성하는 당시엔 무엇이 잘못된 것인지 몰라 엉뚱한 부분만 계속 고쳤었다. 실수를 인지한 후엔 허무했지만, 이후엔 더 꼼꼼히 확인해 다시는 같은 실수를 반복하지 않도록 했다. 이번 프로젝트를 통해 알지 못했던 여러 유용한 함수들도 알게 되었다. 이런 함수들은 책에 표시해두고 필기하며 완전히 내 것으로 만들려고 노력하고 있다. 며칠 간 프로젝트를 진행하면서 점점 성장했고, 스스로 모든 구현을 완성시켰다. 낮부터 밤까지 하루 종일 노트북만 붙잡고 있기도 했고, 같은 수업을 듣는 친구들과 밤새도록 연락하며 같이 문제점을 찾아 나가기도 했다. 다른 사람들의 코드가 어떨지는 모르지만, 적어도 내가 생각하기엔 내 코드에 자부심을 느낀다.

첫번째 프로젝트를 끝마치며 아쉬운 점이 많이 남는다. 인터넷으로 강의를 들으며 그날그날 배운 것을 모두 이해하지 못했었다. 이는 결국 나중에 결과로 나타났다. 프로젝트를 진행하면서 더 많이 찾아보고 수용해야 했으며, 이미 본 강의를 몇 번이고 다시 돌려봐야 했다. 이를

교훈삼아 다음 프로젝트는 더욱 능동적인 자세로 맞이해 아쉬움 없이 끝마치도록 시도할 것이다.
개인적으로 전공적인 지식 뿐만 아니라 여러가지를 배울 수 있던 첫 프로젝트였다.