

Mobile processor Programming assignment #1: Simple calculator

Document

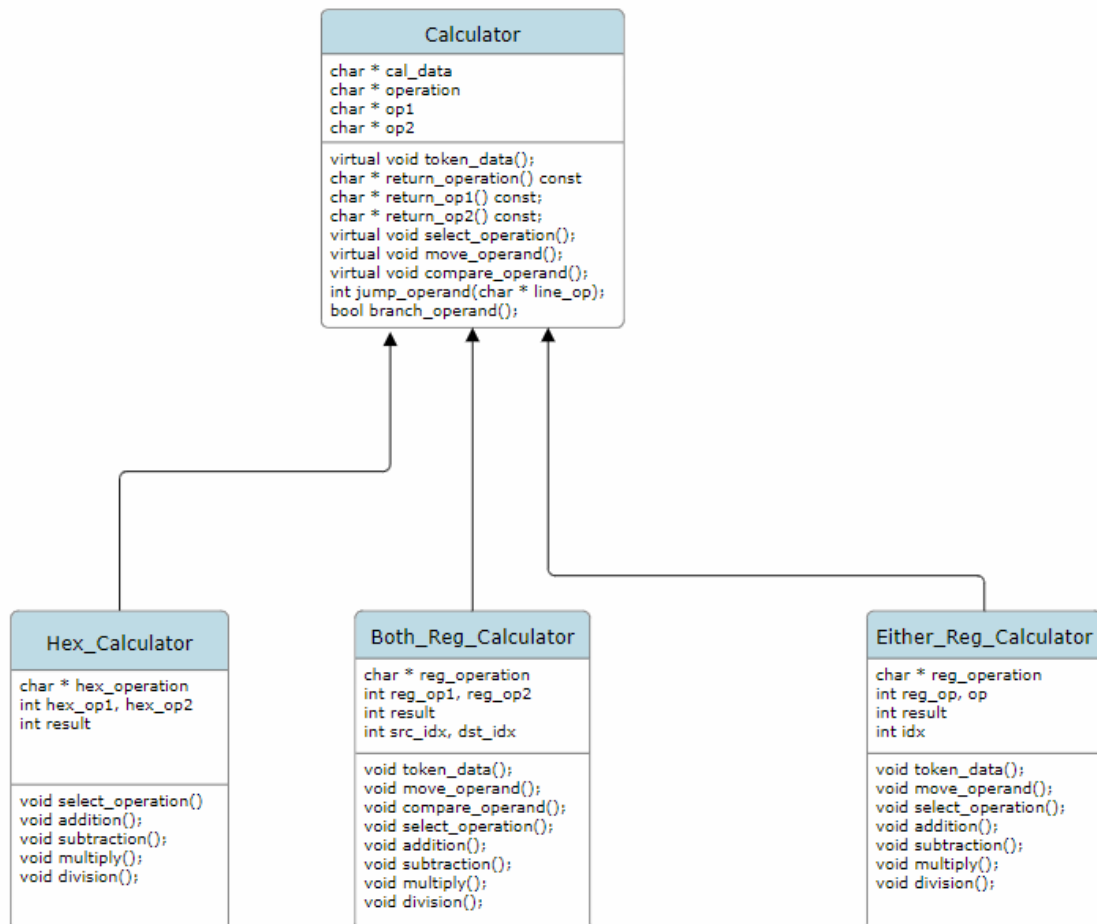
Left Freedays : 5 days

프로젝트 제출일

----- Simple Calculator -----

[0]. Introduction

- 개요 및 설명



Calculator Class를 기초 클래스로 설정함으로써 각 Case 별로 유도 클래스를 설정하였습니다. 'input.txt'로부터 operand를 입력 받을 때 발생하는 상황은 총 3가지가 발생합니다

다.

CASE 1) Operand1, Operand2 모두 Hexadecimal인 경우

CASE 2) Operand1, Operand2 모두 Register인 경우

CASE 3) Operand1, Operand2 하나는 Hexadecimal, 또 다른 하나는 Register인 경우

따라서, 각각 상황에 맞는 Class가 필요했고, 각 Class는 모두 Calculator 역할을 한다는 점에서 기초 Class로 Calculator를 두고 상황에 맞게 유도 Class가 생성되고 실행되도록 하였습니다.

또한, 'Hex_Calculator' Class의 경우에는 둘 다 Immediate Value라는 점에서 'MOV' 연산은 불가능하였고, 사칙연산 기능만 하도록 설정하였습니다. 'Both_Reg_Calculator' Class의 경우에는 Operand 모두 Register이기 때문에 Register Index를 식별하는 과정이 필요하였으므로 'src_idx' 와 'dst_idx'에 각각 index가 분리되도록 하였습니다. 그리고, 사칙연산 기능과 더불어 둘 다 Register Value이기에 'MOV' 연산, 'Compare' 연산이 모두 가능하기에 이러한 기능을 하도록 하였습니다.

마지막으로 'Either_Reg_Calculator' Class의 경우에는 하나의 값은 Immediate Value이고, 또 다른 하나는 Register Value이기에 Register Value의 경우에는 Index를 식별하는 과정이 필요하였고, 사칙연산 기능과 더불어 Immediate Value가 Register로 'MOV'가 가능하도록 'MOV' 연산을 추가하였습니다.

* 모든 Class들의 연산의 결과는 R[0]에 저장되도록 하였습니다.

[1]. Build Configuration/Environment 설명



- 작업 환경 : Visual Studio Code
- 사용한 프로그래밍 언어 : C++
- 사용한 Compiler :

```
{  
  "version": "2.0.0",
```

```

"tasks": [
  {
    "type": "cppbuild",
    "label": "C/C++: cpp build active file",
    "command": "/usr/bin/cpp",
    "args": [
      "-fdiagnostics-color=always",
      "-g",
      "${file}",
      "-o",
      "${fileDirname}/${fileBasenameNoExtension}"
    ],
    "options": {
      "cwd": "${fileDirname}"
    },
    "problemMatcher": [
      "$gcc"
    ],
    "group": "build",
    "detail": "compiler: /usr/bin/cpp"
  },
  {
    "type": "cppbuild",
    "label": "C/C++: g++ build active file",
    "command": "/usr/bin/g++",
    "args": [
      "-fdiagnostics-color=always",
      "-g",
      "${file}",
      "-o",
      "${fileDirname}/${fileBasenameNoExtension}"
    ],
    "options": {
      "cwd": "${fileDirname}"
    },
    "problemMatcher": [
      "$gcc"
    ],
    "group": {
      "kind": "build",
      "isDefault": true
    },
    "detail": "Task generated by Debugger."
  }
]

```

```
}
```

- 사용한 헤더 파일

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
```

[2]. 설명 (Important Part)

- Class의 생성

```
while(input_file.eof() == false){
    input_file.getline(my_data, 100, '\n');
    program_counter = &input_file;

    Calculator * calculator = new Calculator(my_data);

    calculator->token_data();
    char * operation = calculator->return_operation();
    char * op1 = calculator->return_op1();
    char * op2 = calculator->return_op2();

    Calculator * hex_calculator = new Hex_Calculator(my_data, operation,
op1, op2);
    Calculator * both_reg_calculator = new Both_Reg_Calculator(my_data,
operation, op1, op2);
    Calculator * either_reg_calculator = new
Either_Reg_Calculator(my_data, operation, op1, op2);

    // Operation 이 사칙연산일 경우 아래 연산 수행
    if(my_data[0] == '+' || my_data[0] == '-' || my_data[0] == '*' || my_data[0]
== '/'){
        if(strstr(op1, "0x") && strstr(op2, "0x")){
            hex_calculator->token_data();
            hex_calculator->select_operation();
        }
        else if(strstr(op1, "R") && strstr(op2, "R")){
            both_reg_calculator->token_data();
            both_reg_calculator->select_operation();
        }
        else{
```

```

        either_reg_calculator->token_data();
        either_reg_calculator->select_operation();
    }
}

```

FILE Pointer인 input_file은 'input.txt'를 읽어와서 File의 끝에 도달하기 전까지 반복되도록 하였습니다. getline() 함수를 통해서 한 줄 씩 읽어왔고, 읽어온 Data를 바탕으로 Calculator 객체를 생성하여 operation, op1, op2을 분리하여 OOP의 '다형성' 개념에 맞추어 각각의 객체를 생성하였고, 먼저 operation이 사칙연산인지를 판단하였고, op1과 op2가 Hexdecimal인지 Register인지 조건문을 통해 판단하여 각각에 맞는 객체에서 연산이 수행되도록 하였습니다. 또한 각 연산은 같은 이름의 함수로써 기초 클래스인 Calculator에 'Virtual'로 선언된 함수를 사용하여 '다형성'을 유지하도록하여 효율적인 코드가 작성되도록 하였습니다.

```

void Hex_Calculator::addition(){
    hex_op1 = stoul(op1, nullptr, 16);
    hex_op2 = stoul(op2, nullptr, 16);

    try{
        result = hex_op1 + hex_op1;
    }
    catch(exception& e){
        std::cout<<e.what()<<endl;
    }
    std::cout<<"<<count<<"> "<<hex_op1<<" + "<<hex_op2<<" = "<<result<<"
and 0x"<<result<<" is saved in R[0]"<<endl;
    R[0] = result;
}

```

Hex_Calculator에서 addition() 함수를 살펴보면, 먼저 op1과 op2가 문자열로 입력 받아왔기에 16진수로 변환해주는 작업을 진행하였고, 단순히 더해주는 작업을 수행하였습니다. 또한 이 과정 속에서 Error 또는 Exception이 발생하면 RunTime 중 어떤 Error가 발생했는지 출력되도록 하였습니다. 그리고, 연산의 결과는 R[0]에 저장되도록 하였습니다. 다른 연산들('-', '*', '/')도 마찬가지로 이런 매커니즘으로 진행되도록 하였습니다.

```

void Both_Reg_Calculator::token_data(){
    dst_idx = op1[1] - '0';
    src_idx = op2[1] - '0';

    // idx 범위가 정상적이지 않을 경우 예외처리
    if(dst_idx < 0 || dst_idx >9 || src_idx < 0 || src_idx >9){
        std::cout<<"Out of Range !!!"<<endl;
        exit(1);
    }
}

```

두 개의 operand가 모두 Register의 경우 dst_idx와 src_idx를 분리하여 식별하도록 하였고, 범위가 정상적이지 않을 경우에는 예외처리를 하였습니다.

```

void Both_Reg_Calculator::move_operand(){
    std::cout<<"-----The Both Move Operation-----"<<endl;
    R[dst_idx] = R[src_idx];
    std::cout<<"<<+count<<">"<<" Moved Value is "<<R[src_idx]<<" and
R["<<src_idx<<" moves to R["<<dst_idx<<"]"<<endl;
    std::cout<<"The Dst Register R["<<dst_idx<<"]'s value is now
"<<R[dst_idx]<<endl;
}

```

‘MOV’ 연산의 경우에는 단순히 Rs가 Rd로 대입되도록 하였고, 이에 대한 결과를 보여주는 출력문을 작성하였습니다. 그리고, 사칙연산의 경우에는 Hex_Calculator와 유사하게 작성하였습니다.

```

void Either_Reg_Calculator::token_data(){
    if(strstr(op1, "R")){
        idx = op1[1] - '0';
    }
    else{
        idx = op2[1] - '0';
    }

    if(idx < 0 || idx > 9){
        std::cout<<"Out of Range"<<endl;
        exit(1);
    }
}

```

두개의 operand 중 하나만 Register일 경우에는 operand1이 Register인지 operand2가 Register인지 조건문을 통해 판단하여 Index를 가져왔고, Index의 범위가 유효한 범위인지 판단하여 예외처리 하도록 하였습니다. 사칙연산과 'MOV'연산은 기존의 Class와 유사하게 작성하였습니다.

```
else if(my_data[0] == 'H'){
    // 프로그램 종료를 알림
    std::cout<<"====="<<endl;
    std::cout<<"Finish the Simple Calculation Program."<<endl;
    std::cout<<"====="<<endl;
    break;
}
```

그리고, operation이 'Halt'인 경우에는 프로그램이 종료하도록 하였습니다.

[3] Unique Features (Optional Implementation)

3-1) 'C' Operation

```
else if(my_data[0] == 'C'){
    both_reg_calculator->token_data();
    both_reg_calculator->compare_operand();
}
```

```
void Both_Reg_Calculator::compare_operand(){
    std::cout<<"-----The Register Compare Operation-----"<<endl;
    if(R[dst_idx] == R[src_idx]){
        R[0] = 0;
        std::cout<<"<<count<<"<<" Now the R[0] is "<<R[0]<<endl;
    }
    else if(R[dst_idx] > R[src_idx]){
        R[0] = 1;
        std::cout<<"<<count<<"<<" Now the R[0] is "<<R[0]<<endl;
    }
    else{
        R[0] = -1;
        std::cout<<"<<count<<"<<" Now the R[0] is "<<R[0]<<endl;
    }
}
```

```
}
```

'Compare' 연산의 경우에는 MIPS의 조건에 맞추어 Immediate Value가 존재하는 경우 연산이 불가능하다 판단하여 두개의 Operand가 모두 Register인 경우에만 연산을 수행하도록 하였습니다. Register의 값을 가져와 크기를 비교하도록 하였고, 그 결과가 R[0]에 저장되도록 하였습니다.

3-2) 'J' Operation

```
while(input_file.eof() == false){  
    input_file.getline(my_data, 100, '\n');  
    if(my_data[0] == 'J'){  
        total_jump_count++;  
    }  
}
```

'Jump' Operation의 경우에는 사전에 'input.txt' 에서 'J' Operation의 총 개수를 계산하여 결과가 total_jump_count에 저장되도록 하였습니다.

```
else if(my_data[0] == 'J' && jump_count != total_jump_count){  
    cout<<"-----Jump Instruction !!!-----"<<endl;  
    int file_idx = calculator->jump_operand(op1);  
    input_file.seekg(0);  
    for(int i = 0; i<file_idx-1; i++){  
        input_file.getline(my_data, 100, '\n');  
    }  
    jump_count++;  
}
```

따라서, operation이 'J'인 경우와 RunTime에서 계산되고 있는 jump_count와 total_jump_count가 일치하지 않는 경우 아직 수행되지 않은 'Jump' operation이 존재한다고 판단하여 프로그램이 무한 루프에 빠지지 않도록 하였습니다. 또한, 기존에 jump operation을 계산하는 과정에서 File Pointer의 위치가 변하였기에 처음 위치로 초기화하도록 하였습니다.

```
int Calculator::jump_operand(char * line_op){  
    int line_number = line_op[0] - '0';  
    return line_number;  
}
```

Jump 해야할 Line으로 가기 위해 jump_operand() 함수를 통해 문자열로 이루어진 line

의 value를 정수로 변환하여 return 하도록하여 file_idx에 저장하였습니다. 그 후, 반복문을 통해 해당 줄만큼 반복하여 해당 명령어가 실행되도록 하였습니다. 그리고 Jump 연산을 하였기에 jump_count를 증가시켰습니다.

3-3) 'B' Operation

```
// Operation 이 'B'일 경우 아래 연산 수행
else if(my_data[0] == 'B' && branch_count != total_branch_count){
    cout<<"-----Branch Instruction !!!-----"<<endl;
    bool branch = calculator->branch_operand();
    if(branch){
        int branch_idx = op1[0] - '0';
        input_file.seekg(0);
        for(int i = 0; i<branch_idx-1; i++){
            input_file.getline(my_data, 100, '\n');
        }
        branch_count++;
    }
}
```

Branch Operation 역시 Jump_operation과 마찬가지로 먼저 operation이 'B'인지를 판단하고, branch_count와 총 'B' 연산 개수가 동일한 지 판단하여 연산이 수행되도록 하였습니다.

```
bool Calculator::branch_operand(){
    if(R[0] == 0){
        return true;
    }
    else{
        return false;
    }
}
```

Branch_operand() 함수에서는 R[0]가 0인 경우에만 branch하도록 하는 전제가 있었기에 R[0] == 0 이라면 true를 반환하고, 그렇지 않다면 false를 반환하도록 하였습니다.

따라서, true라면 branch 연산을 하여 반복문을 통해 branch하고자 하는 줄로 이동하도록 하였습니다.

```
else{
```

```

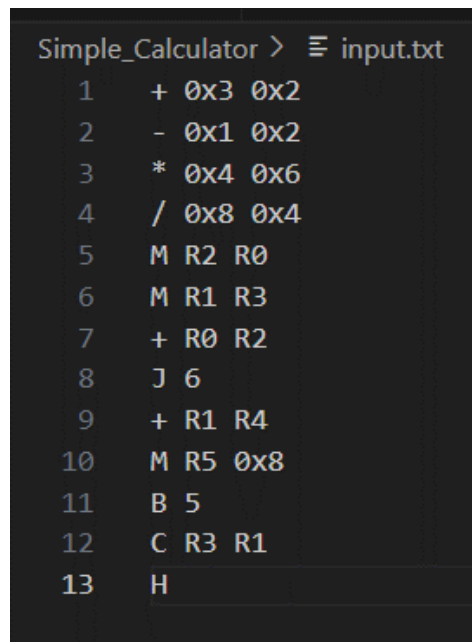
        if(total_jump_count != jump_count && total_branch_count !=
branch_count){
            std::cout<<"There is no Operation !!!"<<endl;
            exit(1);
        }
    }
}

```

마지막으로, jump_count가 일치하지 않고, branch_count가 일치하지 않는 경우에 else 까지 프로그램이 도달했다면 올바르지 않는 operation이라고 판단하고 해당 메시지를 출력하고 exit하도록 하였습니다.

[4] Screen capture/working proofs

4-1) Input.txt



```

Simple_Calculator > ≡ input.txt
1    + 0x3 0x2
2    - 0x1 0x2
3    * 0x4 0x6
4    / 0x8 0x4
5    M R2 R0
6    M R1 R3
7    + R0 R2
8    J 6
9    + R1 R4
10   M R5 0x8
11   B 5
12   C R3 R1
13   H

```

Input.txt가 위와 같을 경우를 가정하여 Simple_Calculator 프로그램을 실행시켰습니다.

4-2) Execution Result

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

[-----The HexaDecimal Calculation-----]
<1> 3 + 2 = 6 and 0x6 is saved in R[0]
[-----The HexaDecimal Calculation-----]
<2> 1 - 2 = -1 and 0x-1 is saved in R[0]
[-----The HexaDecimal Calculation-----]
<3> 4 * 6 = 24 and 0x24 is saved in R[0]
[-----The HexaDecimal Calculation-----]
<4> 8 / 4 = 2 and 0x2 is saved in R[0]
-----The Both Move Operation-----
<5> Moved Value is 2 and R[0] moves to R[2]
The Dst Register R[2]'s value is now 2
-----The Both Move Operation-----
<6> Moved Value is 0 and R[3] moves to R[1]
The Dst Register R[1]'s value is now 0
[-----The Both Reg Calculation-----]
<7> R[0] + R[2] = 0x4 and saved it R[0]
-----Jump Instruction !!!-----
-----The Both Move Operation-----
<8> Moved Value is 0 and R[3] moves to R[1]
The Dst Register R[1]'s value is now 0
[-----The Both Reg Calculation-----]
<9> R[0] + R[2] = 0x6 and saved it R[0]
[-----The Both Reg Calculation-----]
<10> R[1] + R[4] = 0x0 and saved it R[0]
[-----The Either Move Operation-----]
<11> Moved Value is 0x8 and 8 moves to R[5]
The Dst Register R[5]'s value is now 8
-----Branch Instruction !!!-----
-----The Both Move Operation-----
<12> Moved Value is 0 and R[0] moves to R[2]
The Dst Register R[2]'s value is now 0
-----The Both Move Operation-----
<13> Moved Value is 0 and R[3] moves to R[1]
The Dst Register R[1]'s value is now 0
[-----The Both Reg Calculation-----]
<14> R[0] + R[2] = 0x0 and saved it R[0]
[-----The Both Reg Calculation-----]
<15> R[1] + R[4] = 0x0 and saved it R[0]
[-----The Either Move Operation-----]
<16> Moved Value is 0x8 and 8 moves to R[5]
The Dst Register R[5]'s value is now 8
-----The Register Compare Operation-----
<17> Now the R[0] is 0
=====
Finish the Simple Calculation Program.
```

그 결과, 아래와 같이 작성한 명령어에 맞게 올바르게 작성하는 것을 알 수 있었습니다.

[5] Your personal feelings, wishes, suggestions, etc. on the project

프로그램을 작성하면서 600줄이 넘어가는 프로그램 제작 전 구상했던 코드 라인 수보다 훨씬 많은 코드가 발생하여 당황하였습니다. 이와 같은 경험을 하면서 단순한 프로그램이라도 수많은 양의 코드가 발생할 수 있다는 것을 느낄 수 있었고, 첫 프로젝트를 진행하면서 까먹었던 부분에 대해 복습할 수 있는 시간을 가질 수 있어 뿌듯했고, 모르는 부분을 찾아가며 공부를 하면서 프로그램을 제작하는 재미가 있었습니다.

다만 아쉬운 점은 모든 Option들을 제작하고 싶었지만 아직 능력이 부족해서인지, GCD Part는 제작하지 못하였습니다. 비록 프로젝트 과제로써 제출을 하겠지만, 제출 후에도 100% 완벽하지 않다고 생각하는 저의 Simple Calculator를 더 발전시킬 계획을 가지고자 합니다.