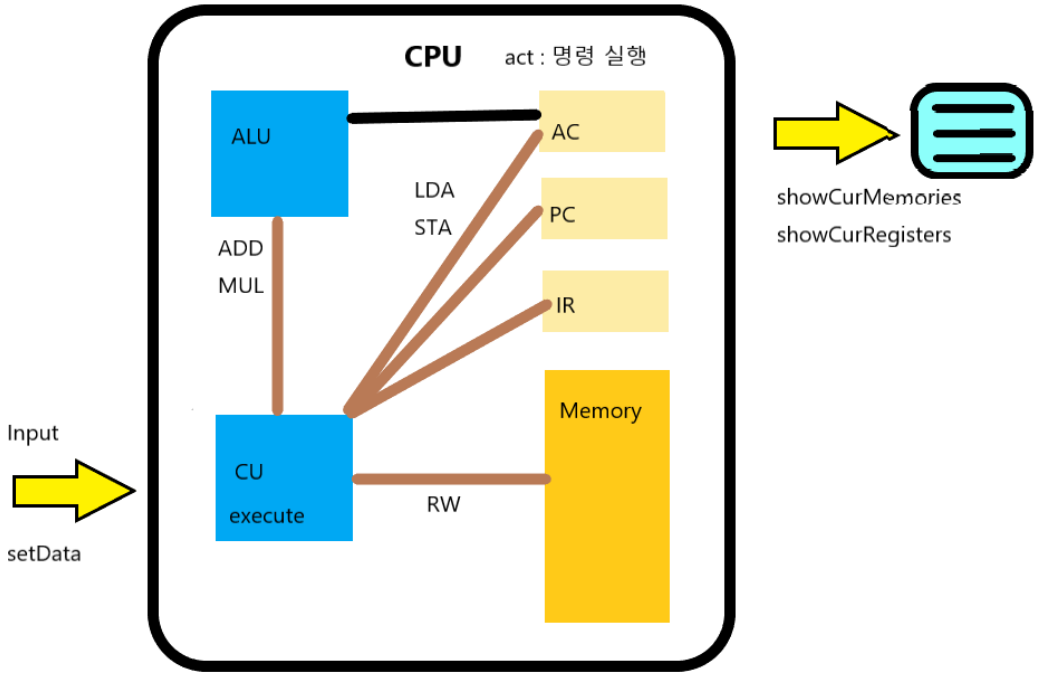
Report

CPU 구현

과목명	
담당 교수	
학과	
학번	
제출일	
이름	

[#1] CPU 프로그램의 전반적인 구성도



현재 프로그램의 구성은 개략적으로 위와 같다. 프로그램이 시작되면 CPU 클래스 내에서 ALU, CU, AC, PC, IR, Memory 에 대한 객체를 생성하고, 이들을 할당한다. 이때 생성된 객체들은 위와 같은 관계를 가지게 된다.

CPU 클래스에는 ALU, CU, AC, PC, IR, Memory 클래스 객체가 존재한다.

CU 클래스는 CPU 클래스에서 생성된 ALU, AC, PC, IR, Memory에 대한 포인터를 가진다.

ALU 클래스는 CPU 클래스에서 생성된 AC에 대한 포인터를 가진다.

CPU에는 CU, ALU 등 연산 및 처리에 사용되는 unit 과 AC, PC, IR 등의 레지스터가 존재한다. 본래 이들은 버스 등을 통해 신호를 전달하고 처리하도록 구현되기 때문에, 보통 특정 레지스터가 특정 유닛에 종속되지 않는 구조를 가진다. 이런 구조를 표현하기 위해 CPU 클래스의 자식으로 위 언급된 클래스에 대한 객체들을 두었다.

버스는 단순히 해당 객체의 포인터를 가지는 형태로 구현했다. 예를 들어 CU의 경우 CPU에서 명령어를 판단하고, 이를 실행하는 역할을 수행한다. 이 과정에서 현재 선언한 모든 클래스와의 접촉이 필요하다. 이를 위해 해당 클래스의 객체들을 가리키는 포인터를 CU의 생성자에 넣었다.

반면 ALU는 현재 주어진 명령어 하에서는 AC만 이용하여 연산 가능하므로, AC만 전달받는다.

[#2] 코드와 프로그램의 흐름

현재 프로젝트의 요구사항 및 프로그램의 흐름을 코드와 함께 설명한다.

전체 프로그램은 C++ 기반으로 구현되었다.

(01) 레지스터

현재 제시된 레지스터인 AC, PC, IR은 **Register** 클래스를 상속받는다.

```cpp
#include "MemMask.h"

using namespace std;

class Register {
protected:
    unsigned char memory{ 0 };
    // 초기 값은 항상 0
public:
    /**
    * 레지스터에 저장된 값을 반환한다.
    * @return 레지스터에 저장된 값
    */
    const unsigned char& get()
    {
        return memory;
    }
    /**
    * 메모리의 내용을 초기화
    * memory &= 0 으로 이루어진다
    */
    void clear()
    {
        // 그냥 0으로 초기화 해도 문제 없다.
        memory &= static_cast<unsigned char>(MemoryMask::CLEAR);
    }
    /**
    * 레지스터의 값을 value로 초기화한다.
    * @param value 초기화할 값
    */
    void set(const unsigned char& value)
    {
        memory = value;
    }
};
```

Register 클래스는 레지스터가 가진 저장 영역 memory를 관리하기 위한 클래스이다. 저장 공간은 unsigned char로 지정되어 있는데, 이는 프로젝트에서 제시된 레지스터의 크기인 8bit에 해당한다.

get : 레지스터에 저장된 내용을 반환한다.

set : 레지스터에 값을 쓴다.

clear : 레지스터의 내용을 초기화한다. 초기화할 때는 MemoryMask가 사용된다.

MemoryMask : 메모리와 관련된 마스크를 정의하는 enum 클래스.

```cpp
enum class MemoryMask {
    CLEAR = 0b00000000, // 0
    OPCODE = 0b11000000, // 위부터 2자리
    OPERAND = 0b00111111, // 아래부터 6자리
};
```

CLEAR : 메모리를 초기화하기 위한 마스크.

OPCODE : 저장된 unsigned char 데이터로부터 OPCODE에 해당하는 값을 추출하기 위한 마스크

OPERAND : 데이터로부터 OPERAND에 해당하는 값을 추출하기 위한 마스크.

기본적으로 현재 프로그램은 1 byte의 명령에서 첫 2비트를 OPCODE, 나머지 6비트를 OPERAND에 할당한다. 따라서 OPCODE는 0~3의 값을 가지고, OPERAND는 0~63의 값을 가지게 된다.

AC, IR

```
#pragma once
#include "Register.h"
class RegisterAC : public Register {};
```

```
#pragma once
#include "Register.h"
class RegisterIR : public Register {};
```

Register 클래스를 상속받는다. 별다른 코드는 존재하지 않는다.

IR 레지스터의 경우 단순히 현재 PC가 가리키는 메모리 주소로부터 값을 받아 오는 동작만 구현해도 충분하다. 따라서 기존의 set만으로도 충분하다.

AC 레지스터의 경우 ALU의 연산 결과를 "저장" 하는 용도의 레지스터로, 현재 프로젝트에 제시된 ADD, MUL 정도의 명령에 대해서는 set, get 만 있어도 충분하다.

위와 같은 이유로, 별다른 코드는 존재하지 않는다.

PC

```
class RegisterPC : public Register {
public:
    void increment()
    {
        memory = (memory + 1) % Memory::MEM_SIZE;
    }

    void add(const unsigned char& index)
    {
        memory = (memory + index) % Memory::MEM_SIZE;
    }
};
```

다음 실행할 명령어의 주소를 가진다. 이때 명령이 순차적으로 실행되는 경우 PC의 값은 1씩 증가하며, 실제 PC 레지스터 자체에도 increment 동작이 존재한다. 또한, 현재 프로젝트에서 해당 명령이 따로 존재하는게 좋을 것 같다는 생각이 들었다. 이런 이유로 현재 값을 1 증가시키는 increment, 값을 더하는 add 메서드를 추가했다.

PC 레지스터는 현재 6bit의 크기만 가져도 충분하다. 메모리의 크기가 64정도이기 때문이다. 현재 굳이 그 이상을 관측하지 않아도 되므로, 메모리 사이즈 내에서 PC가 순환하도록 구현했다.

(02) Memory

```
class Memory {
public:
    static constexpr int MEM_SIZE = 64; // 메모리 사이즈

private:
    int mpt; // 메모리 포인터
    std::array<unsigned char, Memory::MEM_SIZE> inner;// 저장공간

public:
    Memory()
        : mpt(0)
    {
        clearAll(0);
    }
```

Memory 클래스는 컴퓨터의 메모리와 대응되며, 현재 프로젝트에서는 조건에 따라 8bit * 64 의 크기를 가진다. 메모리는 내부적으로 64 크기의 unsigned char의 배열을 통해 구현된다. 메모리의 내부 위치를 가리키는 mpt 변수도 존재한다.

이 클래스에는 요구되는 명령 사항이 존재했다. 이를 각각 구분하여 설명한다.

기본적으로 메모리의 기본 단위인 unsigned char은 8bit 변수이다. 조건에 의해 mpt는 0~63의 범위를 가지는데, 8bit 정수를 4bit 정수 단위로 읽기 위해서는 mpt는 기존의 2배인 0~127의 범위를 가져야 한다. 이는 조건과 위배되며, 현재 프로그램에서도 4bit 단위로 분석하는 것은 큰 의미가 없다. 따라서 "4bit 정수로" 를 내부적으로 값을 어떻게 처리할 것인지의 의미로 해석한다. 현재 프로그램의 모든 값은 기본적으로 4bit 정수 2자리로 구성된다.

[01] READ 동작

```
#pragma region READ
    /**
     * idx에 해당하는 위치의 메모리의 값을 읽어온다.
     * @param idx   메모리에 해당하는 위치
     * @return      해당 메모리 주소에 저장된 값
     */
    const unsigned char& readFrom(const int& idx = 0) // 특정 주소
    {
        check(idx);
        return inner[idx];
    }

    /**
     * 현재 메모리 포인터가 가리키는 위치의 메모리의 값을 읽어온다
     * @return      현재 메모리 주소에 저장된 값
     */
    const unsigned char& readCur() // 현재 메모리 주소
    {
        check(mpt);
        return inner[mpt];
    }

    const unsigned char& readIterCur(const bool& from_0 = false) 
    {
        check(mpt);
        if (from_0)
        {
            clearPtr();
        }
        auto value = inner[mpt];
        incPtr();
        return value;
    }

#pragma endregion
```

읽기에는 readFrom, readCur, readIterCur이 있다.

readFrom 은 **특정 주소**에 저장된 값을 읽는 메서드로, 입력받은 주소의 값을 반환한다.

readCur 은 **현재 주소**에 저장된 값을 읽는 메서드로, mpt의 주소의 값을 반환한다.

readIterCur 은 iteration을 통해 값을 읽는 경우를 위한 메서드로, from_0에 true를 넣는 것으로 **메모리의 첫 주소부터** 값을 읽을 수 있다.

[02] Write 동작

```
#pragma region WRITE
    /**
    * idx에 해당하는 위치의 메모리에 value를 쓴다.
    * @param value 메모리에 쓰는 값
    * @param idx   메모리에 해당하는 위치
    */
    void writeAt(const unsigned char& value, const int& idx = 0) // 특정 주소 혹은 0
    {
        check(idx);
        inner[idx] = value;
    }

    /**
    * 현재 메모리 위치에 value를 쓴다.
    * @param value 현재 메모리에 쓰는 값
    */
    void writeCur(const unsigned char& value) // 현재 메모리 주소
    {
        check(mpt);
        inner[mpt] = value;
    }

    const unsigned char& writeIterCur(const unsigned char& value, const bool& from_0 = false)
    {
        check(mpt);
        if (from_0)
        {
            clearPtr();
        }
        inner[mpt] = value;
        incPtr();
    }
#pragma endregion
```

쓰기에는 writeAt, writeCur, writeIterCur이 있다. 이때 입력하고 값을 변환하는 과정 자체가 16진수에 대응되므로, 내부의 데이터는 정상적인 정수가 저장된다.

writeAt 은 **특정 주소**에 입력받은 값을 쓰는 메서드로, 입력받은 idx 위치에 value를 쓴다.

readCur 은 **현재 주소**에 입력받은 값을 쓰는 메서드로, mpt 위치에 value를 쓴다.

readIterCur 은 iteration을 통해 값을 쓰기 위한 메서드로, from_0에 true를 넣어 **메모리의 첫 주소부터** 값을 쓸 수 있다.

[03] CLEAR 동작

```
#pragma region CLEAR
    /**
    * idx에 해당하는 위치의 메모리를 0으로 초기화한다.
    * @param idx 값을 초기화할 위치
    */
    void clearAt(const int& idx) // idx에 해당하는 곳을 clear
    {
        check(idx);
        inner[idx] = 0;
    }

    /**
    * idx에 해당하는 위치 이후의 메모리를 모두 0으로 초기화한다.
    * @param idx 값을 초기화할 시작 위치
    */
    void clearFrom(const int& idx)
    {
        check(idx);
        for (int i = idx; i < Memory::MEM_SIZE; i++)
        {
            inner[idx] = 0;
        }
    }
    /**
    * 메모리를 defaultValue의 값으로 일괄적으로 초기화한다.
    * @param defaultValue 메모리를 초기화 하는 값
    */
    void clearAll(const unsigned char& defaultValue = 0)
    {
        inner.fill(defaultValue);
    }
#pragma endregion
```

메모리의 초기화를 위한 코드들이다. 메모리의 초기화는 기본적으로 8bit 단위로 수행된다.

clearAt 은 입력받은 **특정 주소**의 메모리를 0으로 초기화한다. 해당 위치에 0을 대입한다.

clearFrom은 **특정 주소부터** 메모리를 **0으로** 초기화하는 메서드이다. 입력받은 주소 이후의 값에 0을 대입한다.

clearAll은 메모리 전체를 입력받은 **defaultValue의 값으로 초기화**한다. 만약 **값을 입력하지 않으면, 모든 값이 0으로 초기화**된다.

[04] 메모리 포인터 설정

```
#pragma region POINTER
    /**
    * 현재 메모리 포인터가 가리키는 주소를 idx로 설정한다
    * @param idx   현재 메모리가 가리킬 주소
    */
    void setPtr(const int& idx)
    {
        mpt = idx;
    }

    /**
    * 메모리 포인터를 초기화한다
    */
    void clearPtr()
    {
        setPtr(0);
    }
    /**
    * 메모리가 가리키는 숫자를 증가시킨다.
    * 숫자가 끝까지 가면 0으로 돌아간다.
    */
    void incPtr()
    {
        mpt = (mpt + 1) % Memory::MEM_SIZE;
    }

    /**
    * 현재 메모리 포인터가 유효한 칸 범위인지 여부를 반환한다.
    * @return _check(mpt)
    */
    bool checkPtr()
    {
        return _check(mpt);
    }
#pragma endregion
```

setPtr은 현재 메모리 위치를 가리키는 mpt에 입력받은 idx을 대입하는 메서드로, **입력받은 주소로** 메모리가 가리키는 주소를 변경한다.

clearPtr은 **메모리 주소를 0으로 초기화**한다.

incPtr은 메모리 주소를 1 증가시킨다. 보통 메모리에 어떤 값을 직접적으로 써야 할 때 이 메서드를 사용하여 입력받은 값을 차례대로 메모리에 쓴다. 이때 잘못된 메모리를 참조하지 않도록 나머지 연산을 통해 범위를 제한한다.

checkPtr() 현재 메모리 포인터의 주소가 유효한지 반환한다.

checkPtr 메서드에는 _check메서드가 사용되고 있다. 해당 메서드는 아래와 같다.

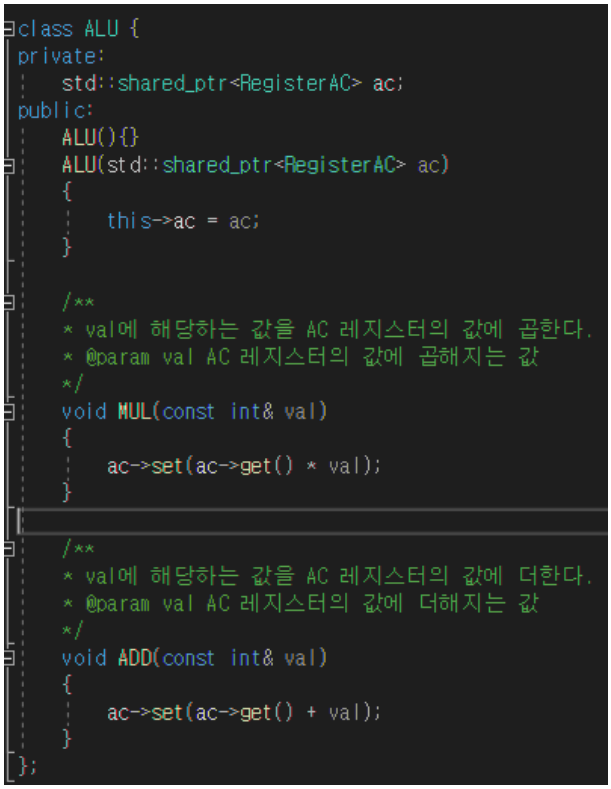
```
    /**
    * idx에 대해 올바른 인덱스인지 검사한다.
    * @param idx 검사할 인덱스
    */
    void check(const int& idx) // 인덱스를 검사한다.
    {
        if (!_check(idx)) // 잘못된 메모리 접근.
        {
            printError("Invalid index");
            exit(EXIT_FAILURE);
        }
    }
    /**
    * 내부적으로 메모리 체크에 사용되는 함수.
    * 해당 인덱스가 유효한지 여부를 bool 형으로 반환한다.
    * @param idx 체크할 메모리 위치
    * @return    현재 입력된 인덱스의 진위 여부
    */
    bool _check(const int& idx)
    {
        return idx >= 0 && idx < Memory::MEM_SIZE;
    }
```

_check : 인덱스가 0 이상, MEM_SIZE 미만인지를 bool형 변수로 알린다.

해당 메서드를 이용하여 Memory 클래스 내부에서는 자체적으로 입력받은 인덱스를 검사하고, 에러를 발생시킨다. 해당 메서드는 check이며, _check의 판단 결과가 false가 되면, 에러 메시지를 출력하고 프로그램을 종료시킨다. 잘못된 주소를 탐색하지 않기 위한 방법으로 사용되었다.

(03) ALU

ALU는 산술 / 논리 연산을 수행하기 위한 유닛이다. 현재 프로젝트에서는 CU에서 호출될 경우 ADD, MUL 메서드를 통해 연산을 수행한다.



```
class ALU {
private:
    std::shared_ptr<RegisterAC> ac;
public:
    ALU(){}
    ALU(std::shared_ptr<RegisterAC> ac)
    {
        this->ac = ac;
    }

    /**
    * val에 해당하는 값을 AC 레지스터의 값에 곱한다.
    * @param val AC 레지스터의 값에 곱해지는 값
    */
    void MUL(const int& val)
    {
        ac->set(ac->get() * val);
    }

    /**
    * val에 해당하는 값을 AC 레지스터의 값에 더한다.
    * @param val AC 레지스터의 값에 더해지는 값
    */
    void ADD(const int& val)
    {
        ac->set(ac->get() + val);
    }
};
```

ALU 클래스는 AC을 직접적으로 사용해야 한다. 따라서 생성자를 통해 AC 레지스터의 포인터를 전달받는다.

ADD : 기존 ADD 명령에 대응된다. AC에 저장된 값을 불러와서 해당 값에 val을 더하고, 이 값을 다시 AC에 저장한다.

MUL : 기존 MUL 명령에 대응된다. AC에 저장된 값을 불러와 해당 값에 MUL을 나누고, 이 값을 다시 AC에 저장한다.

위 명령들은 내부적으로 단순한 덧셈 및 곱셈이다. 프로그램 내부에서 레지스터의 값은 unsigned char으로 선언되어 있기 때문에, int와 연산을 수행하더라도 암시적 형변환에 의해 정상적인 연산이 가능하다.

(04) CU

CU는 해석된 OPCODE 및 OPERAND를 받고, OPCODE에 해당하는 명령을 수행한다. 이때 OPCODE가 대수 / 논리 연산과 관련되었다면, 해당 명령의 수행을 ALU에게 위임한다.

```
class CU{
private:
    std::shared_ptr<RegisterIR> ir;
    std::shared_ptr<RegisterPC> pc;
    std::shared_ptr<RegisterAC> ac;
    std::shared_ptr<Memory> memory;
    std::shared_ptr<ALU> alu;

    static enum class CODE // 내부에서 들어온 프로그램을 구분하기 위한 코드
    {
        LDA = 0b00,
        STA = 0b01,
        ADD = 0b10,
        MUL = 0b11
    };
public:
    CU(){}

    CU( std::shared_ptr<RegisterIR> ir,
        std::shared_ptr<RegisterPC> pc,
        std::shared_ptr<RegisterAC> ac,
        std::shared_ptr<Memory> memory,
        std::shared_ptr<ALU> alu    )
    {
        this->ir = ir;
        this->pc = pc;
        this->ac = ac;
        this->memory = memory;
        this->alu = alu;
    }
```

CU는 CPU의 각종 기기에 대한 연결을 가진다. 원래는 이 것이 버스 등을 통해 나타나지만, 현재 프로젝트에서는 포인터를 이용하여 표현했다.

CU 내부에는 입력받은 코드를 해석하기 위해 CODE enum class가 존재하는데, 현재는 LDA,STA,ADD,MUL에 대해서 판단이 가능하도록 구현되어 있다.

```
void execute(const int& opcode, const int& operand)
{
    switch (static_cast<CODE>(opcode))
    {
    case CODE::LDA:
        LDA(operand);
        break;
    case CODE::STA:
        STA(operand);
        break;
    case CODE::ADD:
        alu->ADD(operand);
        break;
    case CODE::MUL:
        alu->MUL(operand);
        break;
    default:
        printError("도달할 수 없는 코드 (CU, execute)");
        exit(EXIT_FAILURE);
    }
}
```

입력받은 명령어의 판단 및 실행 단계의 경우 execute 메서드가 수행한다.

외부에서 파싱된 opcode 및 operand 을 분석, 해당 값에 맞는 동작을 수행한다.

현재 프로젝트에서는 ADD, MUL 명령은 ALU에 정의되어 있고, 이런 대수 / 논리 명령이 아닌 경우 CU 내부에 정의된다.

```
/**
* operand에 해당하는 주소에서 데이터를 읽어 AC 레지스
* @param operand 읽어올 값의 메모리 주소
*/
void LDA(const int& operand)
{
    ac->set(memory->readFrom(operand));
}
/**
* operand에 해당하는 주소에 AC 레지스터에 저장된 값을
* @param operand AC 레지스터의 내용을 쓸 메모리 주소
*/
void STA(const int& operand)
{
    memory->writeAt(ac->get(), operand);
}
```

LDA : 단순히 operand에 대응되는 주소의 메모리를 가져와서 AC에 로드한다.

STA : operand에 대응되는 메모리에 ac에 저장된 값을 쓴다.

(05) CPU

```
class CPU {
private:
    std::shared_ptr <ALU> alu;
    CU cu;
    std::shared_ptr<RegisterAC> ac;
    std::shared_ptr<RegisterIR> ir;
    std::shared_ptr<RegisterPC> pc;

    std::shared_ptr<Memory> memory;

public:
    CPU()
    {
        ac = std::make_shared<RegisterAC>();
        ir = std::make_shared<RegisterIR>();
        pc = std::make_shared<RegisterPC>();
        memory = std::make_shared<Memory>();

        alu = std::make_shared<ALU>(ac);

        cu = CU(ir, pc, ac, memory, alu);
    }
```

CPU 클래스는 위 언급한 클래스들을 모아둔 클래스로, 데이터 입출력, 정보 출력, decode 등 여러 작업을 수행한다. 프로그램의 판단은 CU에서 수행하므로, 이런 작업을 제외한, 현재 프로젝트 상에서는 나타나지 않는 장치들의 역할을 수행한다.

생성자에서는 언급한 클래스들에 대한 포인터를 생성하거나 초기화 과정을 진행한다.

```
void setData(const char& data)
{
    memory->writeCur(data);
    memory->incPtr();
}
/**
* 데이터를 전부 읽어들인 후 사용하는 함수.
*/
void endData()
{
    memory->clearPtr();
}
```

setData : 메모리에 입력받은 데이터를 쓴다. 기본적으로 CPU가 프로그램을 인식하기 위해서는 obejct code / machine code 수준으로 변환을 거쳐야 하므로, 입력되는 데이터 역시 machine code 수준의 unsigned char 형으로 입력된다.

endData : 메모리의 포인터를 초기화한다.

이들은 일종의 input 역할을 수행한다.

```
void act(const int& from, const int& to)
{
    if (canAccess(from, to))
    {
        cout << "₩n[before]" << endl;
        showCurRegisters();
        showCurMemories(0);
        pc->set(from); // PC의 값을 from으로 초기화

        for (int i = 0; i <= to-from; i++)
        {
            auto [opcode, operand] = decode();
            cu.execute(opcode, operand);
            cout << "₩n[index : " << i << ", execute]" << endl;
            showCurRegisters();
            showCurMemories(0);
        }
    }
    else {
        printError("Cannot Access Memory... (method act)");
        exit(EXIT_FAILURE);
    }
}
```

act : 입력받은 코드를 실행하기 위한 메서드로, 시작 주소 및 끝 주소를 입력받는다. 해당 주소들을 canAccess을 통해 검사한 후, 유효하다면 현재 주소를 보여주고, 실행 과정에서 변하는 주소들을 보여준다. 이 과정에서 execute을 통해 실제 명령들을 수행한다.

```
void showCurRegisters()
{
    cout << "[Registers]" << endl;
    cout << "[PC] : " << std::dec << static_cast<int>(pc->get()) << endl;
    cout << "[IR] : " << static_cast<int>(ir->get()) << endl;
    cout << "[AC] : " << static_cast<int>(ac->get()) << std::dec << endl;
}

/**
* from에서 to까지 저장된 메모리 주소를 보여준다.
* @param from 보여주는 시작 주소
* @param to   보여주는 끝 주소
*/
void showCurMemories(int from, int to = Memory::MEM_SIZE - 1)
{
    constexpr int count = 10;
    if (canAccess(from, to))
    {
        cout << "[Memories]" << endl;
        int cond = 0;
        for (int i = from; i <= to; i++)
        {
            if (i % count == 0) // 구간 나눠서 보여주기
            {
                cout << "₩n[" << from << "]₩t";
            }
            int value = static_cast<int>(memory->readFrom(i));
            cout << std::hex << (value < 16 ? "0" : "") << value << std::dec << ' ';
            if ((i % count) == count - 1 || i == to)
            {
                cout << endl;
            }
        }
    }
    else
    {
        printError("올바르지 못한 메모리 지정입니다.(showMemories)");
    }
}
```

showCurRegisters / showCurMemories : 레지스터 / 메모리를 특정 포맷으로 보여준다.

showMemories의 경우 초기 주소 / 끝 주소를 입력받아, 해당 주소 영역의 값을 선택해서 볼 수 있다.

일종의 output 역할을 수행한다.

```
std::pair<int, int> decode()
{
    ir->set(memory->readFrom(pc->get()));
    //pc가 가리키는 메모리 주소의 값을 ir에 저장한다.
    pc->increment(); // pc를 1 증가시킨다.

    int opcode = (ir->get() & static_cast<int>(MemoryMask::OPCODE)) >> 6;
    int operand = ir->get() & static_cast<int>(MemoryMask::OPERAND);
    //ir에 저장된 값을 opcode와 operand로 나눈다.

    return make_pair(opcode, operand); //opcode와 operand을 반환한다.
}

/**
* 제시된 구간의 메모리에 접근할 수 있는지 여부를 반환한다.
*/
const bool& canAccess(const int& from, const int& to)
{
    return from >= 0 && from <= to && to < Memory::MEM_SIZE;
}
```

decode : 현재 PC가 가리키는 주소의 메모리 값을 IR에 저장한 후, IR에 저장된 데이터를 opcode / operand로 분석한 값을 반환한다. 현재 프로그램에는 없는 Instruction Decoder의 역할을 수행한다.

(06) 전반적인 흐름

현재 프로그램은 프로그램의 입력 -> 명령 수행할 영역의 작성 이라는 구조를 가진다.

```
int main()
{
    CPU cpu;

    std::string fileName;
    std::cout << "프로그램 이름을 입력하세요 : ";

    std::cin >> fileName;
    fileName = trim(fileName);

    fstream f(fileName, std::ios::in);
    if(f.good())
    {
        string temp;
        string opcode;
        string operand;

        OPTAB optab("Optab.txt");
        optab.operateOpCode();
```

프로그램이 시작되면 프로그램의 이름을 입력받는다. 해당 이름을 기반으로 파일을 열고, 파일이 있는 경우 프로그램이 정상적으로 실행된다. 만약 없다면 에러 메시지를 출력하고, 프로그램을 종료한다.

파일이 정상적으로 열리는 경우, 과거 과제에서 만들었던 OPTAB을 이용하여 CPU에 사용되는 명령어 set을 가져온다. 현재 명령어들은 프로젝트에 제시된 명령과 같다.

```
while(!f.eof())
{
    getline(f, temp); // 라인 읽어오기
    int int_opcode = stoi(optab.getOpCode(getOPCODE(temp)),nullptr, 2);
    int int_operand = stoi(getOPERAND(temp),nullptr, 16);
    unsigned char data = 0;
    data = (int_opcode<<6) | int_operand;
    cpu.setData(data);
}
cpu.endData();
cpu.showCurMemories(0, 9);
```

이후 프로그램을 끝까지 입력 받는다. 입력받는 프로그램은 반드시 sic 머신의 사양을 따라야 하며, 해당 규칙에 의해 프로그램 코드를 opcode와 operand로 구분한다. 해당 값으로 data를 생성하고, 해당 데이터를 setData를 통해 cpu 내부로 입력한다.

입력이 끝나면, endData을 통해 메모리 주소를 0으로 초기화하고, 메모리 정보를 보여준다.

```
while (true)
{
    try {
        std::cout << "수행할 명령 라인을 입력하세요!( END : negative value at from / to ) " << std::endl;
        std::cout << "From : ";
        std::cin >> from;
        std::cout << "To : ";
        std::cin >> to;
    }
    catch (exception e) {
        std::cout << "잘못된 입력입니다. 다시 입력하세요!" << std::endl;
        continue;
    }
    if (from < 0 || to < 0)
    {
        std::cout << "프로그램을 종료합니다" << std::endl;
        break;
    }
    cpu.act(from, to);
}
```

이후 입력된 프로그램을 실행하는 코드가 진행된다. 입력되는 값이 정수가 아닐 때를 대비하여 try / catch 구문을 이용했으며, from / to 가 제대로 작성되었을 경우 act가 실행된다.

```
auto [opcode, operand] = decode();
cu.execute(opcode, operand);
cout << "₩n[index : " << i << ", execute]" << endl;
showCurRegisters();
showCurMemories(0);
```

act에서는 입력된 각 명령을 CU에게 넘겨 이를 execute 한다.

```
switch (static_cast<CODE>(opcode))
{
case CODE::LDA:
    LDA(operand);
    break;
case CODE::STA:
    STA(operand);
    break;
case CODE::ADD:
    alu->ADD(operand);
    break;
case CODE::MUL:
    alu->MUL(operand);
    break;
default:
    printError("도달할 수 없는 코드 (CU, execute)");
    exit(EXIT_FAILURE);
}
```

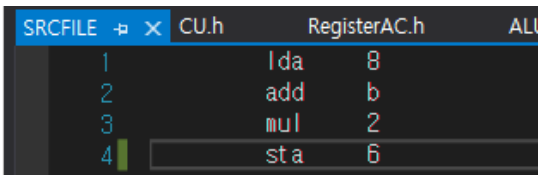
CU는 명령어를 opcode에 따라 분석하고, 해당 명령을 수행하되, 명령이 산술 / 논리 연산인 경우, ALU 에게 맡긴다. (alu->ADD, alu->MUL)

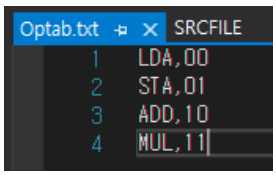
위와 같은 과정 이후 해당 명령의 결과를 보여준다.

[#3] 프로그램 실행 결과

주어진 명령을 수행한 결과는 다음과 같다.

입력 파일 : SRCFILE, Optab.txt





행동 : 0~3 범위에 대해 실행

```
프로그램 이름을 입력하세요 : SRCFILE
[Memories]

[0]     08 8b c2 46 00 00 00 00 00 00
수행할 명령 라인을 입력하세요!( END : negative value at from / to )
From : 0
To : 3
```

결과 :

```
[before]
[Registers]
[PC] : 0
[IR] : 0
[AC] : 0
[Memories]

[00]    03 8b c2 46 00 00 00 00 00 00

[0a]    00 00 00 00 00 00 00 00 00 00

[14]    00 00 00 00 00 00 00 00 00 00

[1e]    00 00 00 00 00 00 00 00 00 00

[28]    00 00 00 00 00 00 00 00 00 00

[32]    00 00 00 00 00 00 00 00 00 00

[3c]    00 00 00 00
```

프로그램 시작 전.

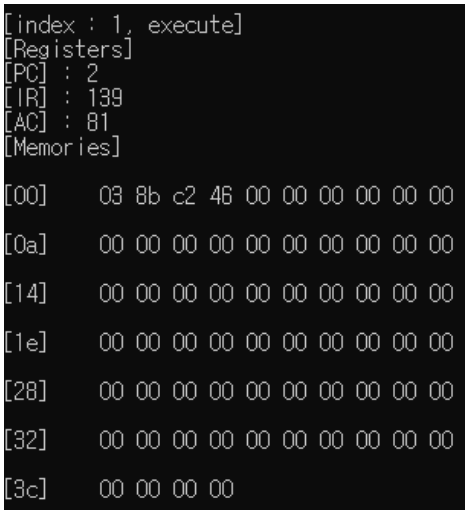
레지스터 및 메모리 정보를 볼 수 있다.

```
[index : 0, execute]
[Registers]
[PC] : 1
[IR] : 3
[AC] : 70
[Memories]

[00]    08 8b c2 46 00 00 00 00 00 00

[0a]    00 00 00 00 00 00 00 00 00 00

[14]    00 00 00 00 00 00 00 00 00 00

[1e]    00 00 00 00 00 00 00 00 00 00

[28]    00 00 00 00 00 00 00 00 00 00

[32]    00 00 00 00 00 00 00 00 00 00

[3c]    00 00 00 00
```

0번 코드를 실행한다. PC = 1로 증가했고, IR에는 해당 0번지 주소값인 08이 들어있다.

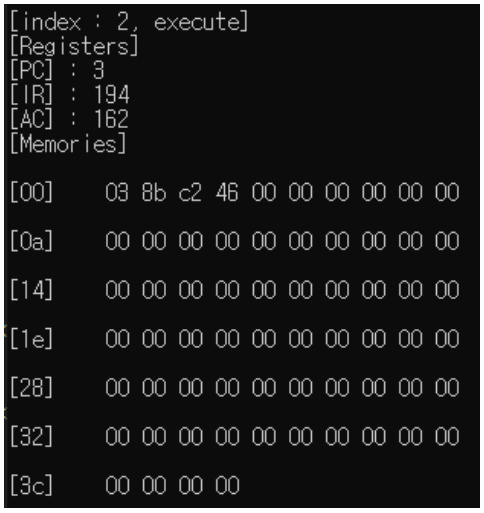
LDA 3 명령으로 인해 0x46 = 70이 AC에 들어갔다.

(현재 프로그램은 레지스터 정보를 10진수로 보여준다.)



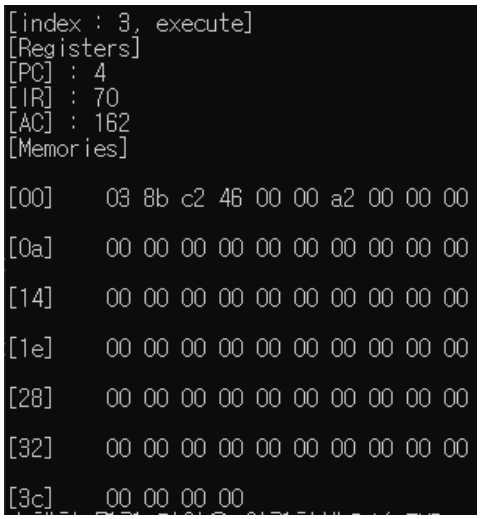
1번 코드를 실행한다. PC는 2를 가리키고, IR은 139 = 0x8b 을 담고 있다.

AC는 ADD B 명령에 의해 70 + 11 = 81 이 되었다.



2번 코드를 실행한다. PC = 3 이 되었으며, IR은 194 = 0xc2코드를 담고 있다.

AC는 MUL 2 명령에 의해 81 * 2 = 162 가 되었다.



3번 코드를 실행한다. PC = 4 가 되었으며, IR은 70 = 0x46 코드를 가지고 있다.

LDA 6에 의해 06 자리에 0xa2 = 162가 저장된 모습을 볼 수 있다.

위 결과에 의해, 프로그램이 제대로 수행되고 있음을 알 수 있다.

위 프로그램이 제대로 동작하는지 보기 위해 다음 명령도 한번 수행해보자.

```
1           lda     0
2           add     b
3           mul     2
4           sta     7
5           add     20
6           mul     3
7           sta     8
```

위 명령이 실행시 (0+11) * 2 = 22 이 7번 자리에, (22+32) * 3 = 162가 8번 자리에 저장된다.

결과를 살펴보자.

```
[Registers]
[PC] : 7
[IR] : 72
[AC] : 162
[Memories]

[00]    00 8b c2 47 a0 c3 48 16 a2 00

[0a]    00 00 00 00 00 00 00 00 00 00

[14]    00 00 00 00 00 00 00 00 00 00

[1e]    00 00 00 00 00 00 00 00 00 00

[28]    00 00 00 00 00 00 00 00 00 00

[32]    00 00 00 00 00 00 00 00 00 00

[3c]    00 00 00 00
```

이 경우에도 연산이 제대로 수행되고 있음을 알 수 있다.

[#4] 참고한 사이트

CPU 클래스를 설계할 때 참고한 사이트들이다,

https://irisxzy.wordpress.com/2016/10/01/cpu-alu-cu-and-registers/

https://velog.io/@underlier12/컴퓨터구조-09-CPU-내부-구조와-레지스터