REPORT

제목 : 2025-2 객체지향설계 텀프로젝트

과 목 : 객체지향설계

담 당 교 수 : 장진수 교수님

팀 명 : 1조

팀 원 : 신해솔 202401833

윤찬영 202401838

최승제 202401852

최준섭 202401853

충남대학교
Chungnam National University

목차

1 서론 : 가상 명령어 해석기의 개요

1.1 프로젝트 배경: 가상화 기반 코드 실행과 보안

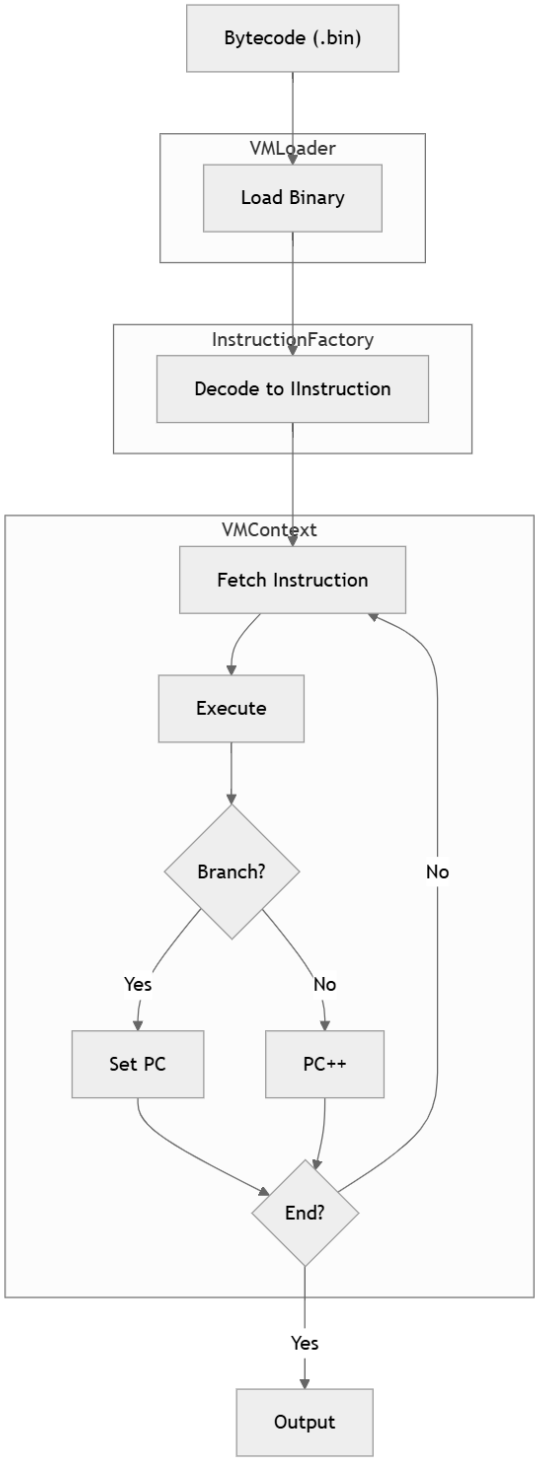
현대의 소프트웨어 환경에서는 단순한 기능 구현을 넘어, 코드 보안과 분석 저항성(Anti-Reverse Engineering)이 중요한 요소로 대두되고 있다. 악성코드 분석이나 리버스 엔지니어링 공격에 대응하기 위해 프로그램의 내부 동작을 숨기는 기법이 필요하며, 그중 하나가 가상 명령어 기반 코드 실행 방식이다. 이 방식은 실제 CPU 명령어 대신 개발자가 정의한 가상 명령어를 사용하여 로직을 난독화 함으로써 외부의 분석을 어렵게 만든다.

1.2 프로젝트 목표: OOP 기반의 확장 가능한 인터프리터 설계

본 프로젝트의 목표는 32비트 구조의 가상 명령어를 해석하고 실행하는 가상 머신(VM)을 설계하고 구현하는 것이다. 특히 단순 작동하는 해석기를 넘어, 객체지향 설계 원칙(OOP)과 디자인 패턴을 적극 활용하여 확장 가능하고 유지보수가 용이한 시스템을 구축하는 데 중점을 두었다. 또한, 새로운 명령어가 추가되거나 해석 방식이 변경되더라도 기존 코드의 수정을 최소화하는 개방-폐쇄 원칙(OCP)을 실현하고자 한다.

1.3 전체 시스템 실행 흐름 (Flowchart Analysis)

시스템은 바이너리 파일을 입력받아 명령어를 디코딩하고, VMContext 내의 가상 CPU 상태(레지스터, 스택)를 갱신하며 프로그램을 실행한다. 전체적인 흐름은
[입력 파일 로드] -> [명령어 파싱] -> [객체 생성] -> [실행 루프]
의 순서를 따른다.



2 핵심 설계 철학 및 아키텍처

2.1 Command 패턴의 적용: Instruction 추상 클래스 설계

본 프로젝트는 조건문으로 가득 찬 절차적 실행 방식 대신, Command 패턴을 적용하여 각 명령어를 독립된 객체로 캡슐화하였다. 모든 명령어는 **IInstruction**이라는 추상 클래스를 상속받으며, 공통된 실행 인터페이스인 **execute()** 메서드를 구현한다. 이를 통해 호출부(Invoker)인 VM은 구체적인 명령어의 종류를 알 필요 없이 다형성을 통해 일관된 방식으로 명령을 실행할 수 있다.

```cpp
class IInstruction {
public:
    IInstruction(uint8_t flag, uint8_t src, uint8_t dest)
        : m_flag(flag), m_src(src), m_dest(dest) {}
    virtual ~IInstruction() = default;
    virtual ExecutionResult execute(VMContext& context) = 0;

protected:
    [[nodiscard]] uint8_t resolveValue(const VMContext& context, uint8_t operand) const;

    uint8_t m_flag;
    uint8_t m_src;
    uint8_t m_dest;
};
```

2.2 Enum Class를 활용한 타입 시스템 강화 및 안전성 확보

기존 C++의 `enum`은 정수형으로 암시적 변환이 가능하여, 의미적으로 전혀 다른 상수끼리 연산되거나 비교되는 실수를 범하기 쉽다. 예를 들어 `RegisterID::R0`와 `OpCode::MOV`가 우연히 같은 정수 값을 가질 때, 컴파일러는 이를 오류로 잡지 못한다. 본 프로젝트에서는 이러한 논리적 오류를 차단하기 위해 `enum class`를 도입하였다. 이를 통해 서로 다른 열거형 타입 간의 혼용을 컴파일 타임에 방지하고, 네임스페이스 분리를 통해 변수명 충돌 문제를 해결하였다. 또한, 열거형의 기반 타입(Underlying Type)을 `uint8_t`로 명시하였다. 이를 통해 VM의 메모리와 명령어가 바이트 단위로 동작하는 하드웨어적 특성을 코드에 반영하고 불필요한 메모리 낭비를 최소화했다.

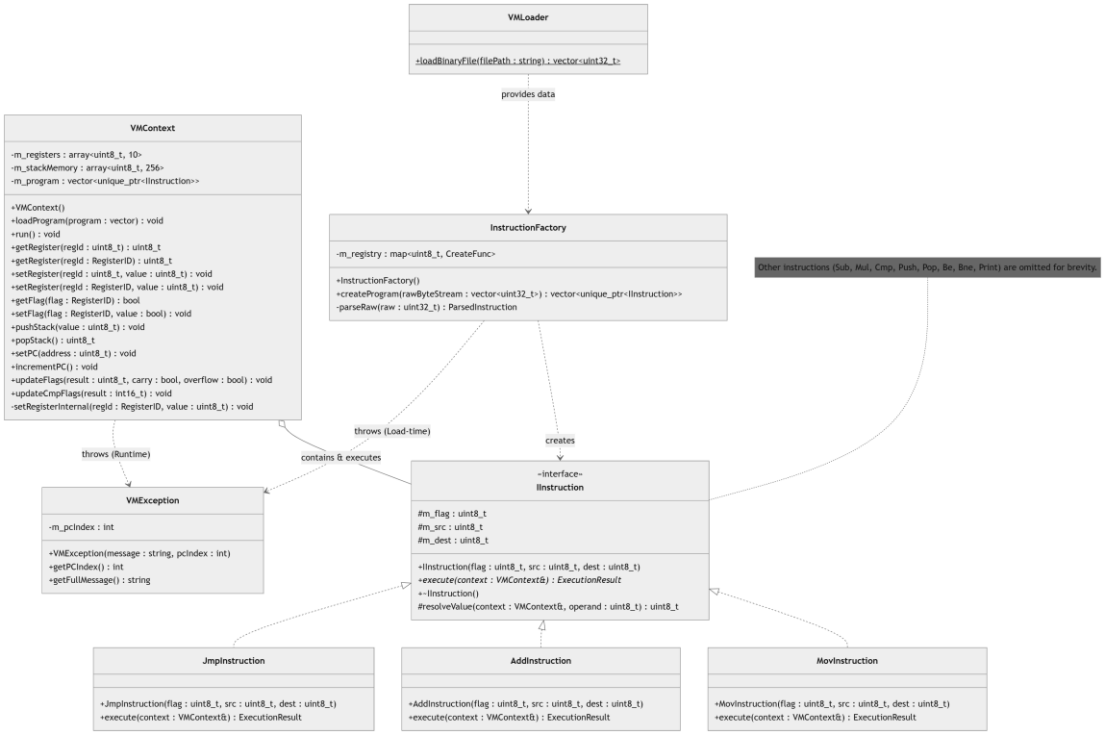
```cpp
enum class RegisterID : uint8_t {
    R0 = 0x01,
    R1 = 0x02,
    R2 = 0x03,
    PC = 0x04,
    SP = 0x05,
    BP = 0x06,
    ZF = 0x07,
    CF = 0x08,
    OF = 0x09
};

enum class OpCode : uint8_t {
    MOV = 0x01,
    ADD = 0x02,
    SUB = 0x03,
    MUL = 0x04,
    CMP = 0x05,
    PUSH = 0x06,
    POP = 0x07,
    JMP = 0x08,
    BE = 0x09,
    BNE = 0x0A,
    PRINT = 0x0B
};
```

2.3 Template Method 패턴을 통한 중복 코드 제거 (DRY 원칙)

`MOV`, `ADD`, `SUB` 등 대부분의 명령어는 피연산자가 레지스터인지 상수 값인지 판별하여 값을 가져오는 공통 로직이 필요하다. 이 공통 로직을 각 자식 클래스에서 반복하는 것을 피하기 위해, 부모 클래스(`IInstruction`)에 `resolveValue` 메서드를 구현하여 하위 클래스가 이를 재사용하도록 설계하였다. 이는 Template Method 패턴의 변형으로, 각 명령어 클래스는 고유한 연산 로직에만 집중할 수 있게 되어 단일 책임 원칙(SRP)을 준수한다.

```cpp
uint8_t IInstruction::resolveValue(const VMContext& context, uint8_t operand) const {
    auto flag = static_cast<FlagType>(m_flag);
    switch (flag) {
        case FlagType::REG_REG:
        case FlagType::SINGLE_REG:
            return context.getRegister(operand);
        case FlagType::REG_VAL:
        case FlagType::SINGLE_VAL:
            [[fallthrough]];
        default:
            return operand;
    }
}
```

2.4 전체 클래스 구조 (UML Diagram)



1. 핵심 시스템 클래스

VMContext

- 역할: 가상 머신의 실행 상태(State)를 총괄 관리하는 핵심 클래스.
- 구성:
  - `m_registers`: 범용 레지스터 및 특수 레지스터(`PC`, `SP`, `BP` 등)의 값을 저장하는 컨테이너.
  - `m_stackMemory`: 스택 연산을 위한 메모리 공간을 관리.
  - `m_program`: 실행될 명령어 객체들의 리스트 (`vector<unique_ptr<IInstruction>>`)를 보유.
- 특징: 정보 은닉 원칙에 따라 모든 데이터 멤버는 `private`으로 선언되었으며, `getRegister`, `pushStack` 등의 공개 인터페이스를 통해서만 상태에 접근할 수 있다. `run()` 메서드는 메인 실행 루프를 담당하며, 각 명령어의 `execute()`를 호출한다.

InstructionFactory

- 역할: 원시 바이너리 데이터를 해석하여 실행 가능한 명령어 객체(`IInstruction`)로 변환하는 팩토리 클래스.
- 구성: m_`registry`는 Opcode와 그에 대응하는 객체 생성 함수(Creator Function)를 매핑하는 테이블.
- 특징: 팩토리 메서드 패턴을 적용하여, 새로운 명령어가 추가되더라도 `createProgram`의 파싱 로직을 수정할 필요 없이 레지스트리에 등록만 하면 되는 확장성(OCP)을 제공한다. 또한, 객체 생성 시점에 잘못된 `Opcode`나 `Flag`를 검증하는 정적 분석을 수행한다.

VMLoader

- 역할: 외부 바이너리 파일을 읽어 시스템이 처리 가능한 `uint32_t` 스트림으로 변환하는 유틸리티 클래스.
- 특징: 파일 I/O의 효율성을 위해 대용량 데이터를 한 번에 메모리로 적재(Bulk Read)하며, 파일 크기 및 정합성을 검증한다.

VMException

- 역할: 시스템 전반에서 발생하는 예외를 체계적으로 처리하기 위한 사용자 정의 예외 클래스.
- 특징: 단순한 에러 메시지뿐만 아니라, 오류가 발생한 명령어의 인덱스(PC) 정보를 함께 캡슐화하여 `getFullMessage()`를 통해 디버깅에 필요한 문맥 정보를 제공한다.

2. 명령어 계층 구조

IInstruction

- 역할: 모든 명령어 클래스가 상속받아야 하는 Abstract Base Class이자 인터페이스이다.

- 구성:
  - `execute(VMContext&)`: 순수 가상 함수로 정의되어 있으며, 모든 하위 클래스는 이를 구현하여 자신만의 고유한 동작을 수행해야 한다. (Command/Strategy Pattern)
  - `m_flag`, `m_src`, `m_dest`: 모든 명령어에서 공통적으로 사용되는 피연산자 정보를 상위 클래스로 끌어올려 코드 중복을 제거.
  - `resolveValue()`: 피연산자가 레지스터인지 상수값인지 판별하는 공통 로직을 템플릿 메서드 형태로 제공.

Concrete Instructions (MovInstruction, AddInstruction, JmpInstruction 등)

- 역할: `IInstruction`을 상속받아 실제 연산 로직을 구현한 구체적인 클래스.
- 특징: 각 클래스는 단일 책임 원칙(SRP)에 따라 하나의 명령어 동작만을 구현하며, `VMContext`를 통해 상태를 변경하거나 흐름을 제어한다.

3 VMContext: 가상 머신 상태 관리

3.1 레지스터 및 메모리 구조 설계와 정보 은닉

`VMContext` 클래스는 가상 머신의 상태(State)를 관리하는 핵심 컴포넌트이다. 내부적으로 8비트 범용 레지스터(`R0`, `R1`, `R2`)와 프로그램 카운터(`PC`), 스택 포인터(`SP`), 베이스 포인터(`BP`), 그리고 상태 플래그(`ZF`, `CF`, `OF`)를 멤버 변수로 보유한다. 객체지향의 정보 은닉(Information Hiding) 원칙을 준수하기 위해 모든 레지스터와 스택 메모리 배열은 `private` 접근 제어자로 선언하여 외부의 직접 접근을 차단하였다. 대신 `getRegister`, `setRegister`와 같은 공개 인터페이스를 제공하여 제어된 방식으로만 상태를 변경할 수 있도록 설계하였다. 이는 외부 객체(Instruction)가 VM의 내부 상태를 무분별하게 수정하여 발생할 수 있는 오류를 방지한다.

```cpp
class VMContext {
public:
    VMContext();
    void loadProgram(std::vector<std::unique_ptr<IInstruction>> program);
    void run();

    [[nodiscard]] uint8_t getRegister(uint8_t regId) const;
    [[nodiscard]] uint8_t getRegister(RegisterID regId) const;
    void setRegister(uint8_t regId, uint8_t value);
    void setRegister(RegisterID regId, uint8_t value);

    [[nodiscard]] bool getFlag(RegisterID flag) const;
    void setFlag(RegisterID flag, bool value);

    void pushStack(uint8_t value);
    uint8_t popStack();

    void incrementPC();
    void setPC(uint8_t address);

    void updateFlags(uint8_t result, bool carry, bool overflow);

    void updateCmpFlags(int16_t result);

    static constexpr size_t STACK_SIZE = 256;

private:
    void setRegisterInternal(RegisterID regId, uint8_t value);
    std::array<uint8_t, REGISTER_COUNT> m_registers;
    std::array<uint8_t, STACK_SIZE> m_stackMemory;
    std::vector<std::unique_ptr<IInstruction>> m_program;
};
```

특히 레지스터 배열 관리에는 편의성과 확장성을 고려한 설계를 적용하였다. 실제 가상 머신의 레지스터는 9개(R0~0F)이지만, 내부적으로는 그림과 같이

```
1   constexpr size_t REGISTER_COUNT = 10;
```

정의하여 배열 크기를 관리하였다. 이는 0번 인덱스를 의도적으로 비워두고, 1부터 시작하는 레지스터 ID와 배열 인덱스를 직관적으로 1:1 매핑하기 위함이다. 이러한 설계는 인덱스 변환 과정에서의 실수를 줄여주며, 추후 레지스터 개수가 확장되더라도 유연하게 대응할 수 있는 구조를 제공한다.

3.2 스택(Stack) 메모리 관리와 경계 검사(Bounds Check)

스택 메모리는 `std::vector<uint8_t>` 또는 고정 배열을 사용하여 관리되며, LIFO(Last-In First-Out) 구조로 동작한다. 스택 조작 시 가장 중요한 위험 요소인 **스택 오버플로우(Overflow)**와 **언더플로우(Underflow)**를 방지하기 위해, `push`와 `pop` 메서드 내부에서 `SP` 인덱스의 유효성을 검사하는 경계 검사(Bounds Check) 로직을 구현하였다. 유효 범위를 벗어난 접근이 감지될 경우, 즉시 런타임 에러를 발생시켜 시스템이 비정상적인 상태로 계속 실행되는 것을 막는다.

```cpp
void VMContext::pushStack(uint8_t value) {
    uint8_t sp = getRegister(RegisterID::SP);
    if (sp == 0) {
        throw std::runtime_error("Error: Stack Overflow");
    }
    sp--;
    m_stackMemory[sp] = value;
    setRegisterInternal(RegisterID::SP, sp);
}

uint8_t VMContext::popStack() {
    uint8_t sp = getRegister(RegisterID::SP);
    if (sp == STACK_SIZE - 1) {
        throw std::runtime_error("Error: Stack Underflow");
    }
    uint8_t value = m_stackMemory[sp];
    sp++;
    setRegisterInternal(RegisterID::SP, sp);
    return value;
}
```

3.3 반환 타입의 구조화와 제어 흐름의 가시성 확보

명령어 실행 후의 반환 값 처리에서도 구조적인 설계를 적용하였다. 불명확한 `void`나 `int` 반환 방식 대신, `ExecutionResult` 라는 `enum class`를 정의하여 실행 상태를 명확히 구분하였다.

- `ExecutionResult::Next`: 일반적인 명령어 실행 후, `PC`를 증가시켜 다음 명령어로 진행함.
- `ExecutionResult::Jumped`: 분기 명령어(`JMP`, `BE` 등)에 의해 `PC`가 이미 변경되었으므로, 자동 증가를 수행하지 않음.

이를 통해 `VMContext`의 `run()` 루프 내에서 복잡한 조건문 없이 반환 값만으로 흐름을 제어할 수 있게 되어, 코드의 가독성과 유지보수성을 향상시켰다.

4 명령어 디코딩 시스템 (Instruction Decoding)

4.1 32비트 명령어 포맷 분석 및 비트와이즈 연산

본 시스템은 32비트(4바이트) 정수 하나에 명령어의 모든 정보가 담긴 구조를 채택하고 있다. `VMLoader`는 바이너리 파일에서 읽어 들인 32비트 데이터를 **비트와이즈 연산 (Bitwise Operation)** 과 비트 마스킹(Bit Masking) 기법을 사용하여 해석한다. 구체적으로 (instr >> 26) & 0x3F 연산을 통해 상위 6비트의 `Opcode`를 추출하고, 이어지는 비트 시프트 연산으로 `Flag`(2bit), `Src`(8bit), `Dest`(8bit) 필드를 각각 분리해낸다. 이 과정에서 리틀 엔디안(Little-Endian) 아키텍처에서의 바이트 순서 문제를 고려하여 데이터를 올바르게 정렬하는 전처리 과정을 거친다.

```
ParsedInstruction InstructionFactory::parseRaw(uint32_t raw) {
    ParsedInstruction p{};
    uint8_t byte0 = (raw >> 0) & 0xFF;
    p.opcode = byte0 >> 2;
    p.flag = byte0 & 0x03;
    p.src = (raw >> 16) & 0xFF;
    p.dest = (raw >> 24) & 0xFF;
    return p;
}
```

4.2 Flag 비트(2bit) 해석과 주소 지정 모드

명령어 내의 2비트 `Flag` 필드는 피연산자(Operand)가 **레지스터 주소**인지 **상수 값** (Immediate Value) 인지를 결정하는 주소 지정 모드(Addressing Mode)를 나타낸다. 예를 들어 `Flag`가 "00"일 경우 두 피연산자 모두 레지스터로 해석하고, 01일 경우 목적지(`Dest`)는 레지스터, 원본(`Src`)은 값으로 해석한다. 이러한 Flag 해석 로직은 각 Instruction 객체가 생성될 때 전달되어, 실행 시점에 동적으로 피연산자의 성격을 결정짓는다.

4.3 Factory Method 패턴 기반의 객체 생성과 OCP 준수

일반적인 구현에서는 `switch-case` 문을 사용하여 `Opcode`에 따른 객체를 생성하지만, 본 프로젝트에서는 확장성을 극대화하기 위해 Factory Method 패턴을 고도화하여 적용하였다. `std::map<OpCode, CreatorFunc>` 형태의 레지스트리를 구성하고, 람다(Lambda) 함수를 통해 객체 생성 로직을 매핑하였다. 이 설계를 통해 새로운 명령어가 추가되더라도 기존의 디코딩 루프나 `InstructionFactory`의 핵심 코드를 수정할 필요 없이, 단순히 `map`에 새로운 항목을 등록하기만 하면 되는 **개방-폐쇄 원칙(OCP)**을 준수하였다.

```cpp
InstructionFactory::InstructionFactory() {

    m_registry[static_cast<uint8_t>(OpCode::MOV)] =
        [](uint8_t f, uint8_t s, uint8_t d) {
            return std::make_unique<MovInstruction>(f, s, d);
        };

    m_registry[static_cast<uint8_t>(OpCode::ADD)] =
        [](uint8_t f, uint8_t s, uint8_t d) {
            return std::make_unique<AddInstruction>(f, s, d);
        };

    m_registry[static_cast<uint8_t>(OpCode::SUB)] =
        [](uint8_t f, uint8_t s, uint8_t d) {
            return std::make_unique<SubInstruction>(f, s, d);
        };

    m_registry[static_cast<uint8_t>(OpCode::MUL)] =
        [](uint8_t f, uint8_t s, uint8_t d) {
            return std::make_unique<MulInstruction>(f, s, d);
        };

    m_registry[static_cast<uint8_t>(OpCode::CMP)] =
        [](uint8_t f, uint8_t s, uint8_t d) {
            return std::make_unique<CmpInstruction>(f, s, d);
        };

    m_registry[static_cast<uint8_t>(OpCode::PUSH)] =
        [](uint8_t f, uint8_t s, uint8_t d) {
            return std::make_unique<PushInstruction>(f, s, d);
        };

    m_registry[static_cast<uint8_t>(OpCode::POP)] =
        [](uint8_t f, uint8_t s, uint8_t d) {
            return std::make_unique<PopInstruction>(f, s, d);
        };

    m_registry[static_cast<uint8_t>(OpCode::JMP)] =
        [](uint8_t f, uint8_t s, uint8_t d) {
            return std::make_unique<JmpInstruction>(f, s, d);
        };

    m_registry[static_cast<uint8_t>(OpCode::BE)] =
        [](uint8_t f, uint8_t s, uint8_t d) {
            return std::make_unique<BeInstruction>(f, s, d);
        };

    m_registry[static_cast<uint8_t>(OpCode::BNE)] =
        [](uint8_t f, uint8_t s, uint8_t d) {
            return std::make_unique<BneInstruction>(f, s, d);
        };

    m_registry[static_cast<uint8_t>(OpCode::PRINT)] =
        [](uint8_t f, uint8_t s, uint8_t d) {
            return std::make_unique<PrintInstruction>(f, s, d);
        };
}
```

5 명령어 구현 세부 사항

5.1 데이터 이동 및 산술 연산 (MOV, ADD, SUB, MUL)

데이터 이동(MOV)과 산술 연산(ADD, SUB, MUL) 명령어는 [Dest] <- [Src] 형태의 공통된 패턴을 가진다. 이 명령어들의 핵심은 피연산자 해석의 추상화이다. 명령어 객체는 flag 값에 따라 Src가 레지스터 ID인지 상수 값인지를 일일이 분기 처리하는 대신, 부모 클래스의 resolveValue() 메서드를 호출하여 값을 가져온다. 이를 통해 각 명령어의 execute 메서드는 피연산자의 타입 해석 로직에서 해방되어, 순수한 연산 로직(덧셈, 뺄셈 등)에만 집중할 수 있게 구현되었다.

```cpp
AddInstruction::AddInstruction(uint8_t flag, uint8_t src, uint8_t dest)
    : IInstruction(flag, src, dest) {}

ExecutionResult AddInstruction::execute(VMContext& context) {
    uint8_t val1 = context.getRegister(m_dest);
    uint8_t val2 = resolveValue(context, m_src);

    uint16_t result = static_cast<uint16_t>(val1) + static_cast<uint16_t>(val2);
    int16_t signedResult = static_cast<int16_t>(static_cast<int8_t>(val1)) + static_cast<int16_t>(static_cast<int8_t>(val2));

    auto finalResult = static_cast<uint8_t>(result);
    bool carry = (result > 0xFF);
    bool overflow = (signedResult > 127 || signedResult < -128);

    context.setRegister(m_dest, finalResult);
    context.updateFlags(finalResult, carry, overflow);
    return ExecutionResult::Next;
}
```

5.2 비교 및 플래그 제어 (CMP)

CMP 명령어는 두 피연산자의 크기를 비교하여 상태 플래그(ZF, CF, OF)를 갱신한다. 내부적으로는 SUB 연산과 유사하게 뺄셈을 수행하지만, 그 결과를 레지스터에 저장하지 않고 오직 플래그 레지스터만 업데이트한다는 점이 특징이다. 이 명령어는 연산(Calculation)과 흐름 제어(Control Flow)의 책임을 분리하는 역할을 한다. CMP는 오직 상태만을 변경하며, 실제 분기는 이어지는 BE나 BNE 명령어가 이 상태를 읽어 수행한다. 이를 통해 명령어 간의 결합도를 낮추고 로직의 단위를 명확히 하였다.

```cpp
#include "instructions/CmpInstruction.h"
#include "core/VMContext.h"

CmpInstruction::CmpInstruction(uint8_t flag, uint8_t src, uint8_t dest)
    : IInstruction(flag, src, dest) {}

ExecutionResult CmpInstruction::execute(VMContext& context) {
    uint8_t val1_unsigned = context.getRegister(m_dest);
    uint8_t val2_unsigned = resolveValue(context, m_src);

    int16_t result = static_cast<int16_t>(static_cast<int8_t>(val1_unsigned)) -
                     static_cast<int16_t>(static_cast<int8_t>(val2_unsigned));

    context.updateCmpFlags(result);
    return ExecutionResult::Next;
}

```

5.3 스택 조작 (PUSH, POP)

스택 명령어는 VMContext가 제공하는 push()와 pop() 인터페이스를 통해 스택 메모리를 조작한다.

- PUSH: resolveValue를 통해 레지스터 값이나 상수를 가져와 스택의 Top(SP)에 저장한다.
- POP: 스택의 Top에서 값을 꺼내어 목적지 레지스터(Dest)에 저장한다. 이 과정에서 스택 포인터(SP)의 변경과 메모리 경계 검사는 VMContext 내부에서 캡슐화되어 처리되므로, 명령어 클래스는 안전하게 스택 기능을 활용할 수 있다.

5.4 분기 제어와 흐름 변경 (JMP, BE, BNE)

프로그램 카운터(PC)를 제어하여 실행 흐름을 변경하는 명령어들이다. BE와 BNE는 VMContext의 ZF(Zero Flag) 상태를 확인하여 점프 여부를 결정한다. 특히 본 구현에서는 앞서 언급한 ExecutionResult 열거형을 적극 활용하였다. 분기가 발생한 경우 ExecutionResult::Jumped를 반환하여 메인 루프가 PC를 중복으로 증가시키는 것을 방지한다. 이는 "PC 자동 증가"와 "강제 점프" 사이의 충돌을 해결한 설계 패턴이다.

```cpp
BeInstruction::BeInstruction(uint8_t flag, uint8_t src, uint8_t dest)
    : IInstruction(flag, src, dest) {}

ExecutionResult BeInstruction::execute(VMContext& context) {
    if (context.getFlag(RegisterID::ZF)) {
        uint8_t jumpAddress = resolveValue(context, m_dest);
        context.setPC(jumpAddress);
        return ExecutionResult::Jumped;
    }
    return ExecutionResult::Next;
}
```

5.5 입출력 (PRINT)

PRINT 명령어는 디버깅과 결과 확인을 위한 I/O 기능을 수행한다. 다른 명령어와 마찬가지로 resolveValue를 사용하여 레지스터에 저장된 값이나 상수 값을 유연하게 출력할 수 있도록 구현되었으며, 테스트 케이스의 검증 도구로 활용된다.

6 시스템 안정성 및 예외 처리

6.1 파일 입출력(I/O) 최적화: VMLoader의 Bulk Read

대용량 바이너리 파일을 처리할 때, 4바이트 단위로 반복해서 디스크를 읽는 방식은 I/O 오버헤드를 발생시켜 성능 저하의 원인이 된다. 본 프로젝트의 **VMLoader**는 이를 개선하기 위해 Bulk Read(일괄 읽기) 방식을 채택하였다. **file.read** 메서드를 사용하여 파일의 끝까지 모든 데이터를 메모리 버퍼(**Vector**)로 한 번에 로드한 뒤, 메모리 상에서 고속으로 파싱을 수행한다. 이는 C++ **vector**가 보장하는 연속된 메모리 공간의 이점을 활용한 최적화 기법이다.

```cpp
std::vector<uint32_t> VMLoader::loadBinaryFile(const std::string& filePath) {
    std::ifstream file(filePath, std::ios::binary | std::ios::ate);

    if (!file.is_open()) {
        throw std::runtime_error("Error: Cannot open file " + filePath);
    }

    std::streamsize size = file.tellg();
    file.seekg(0, std::ios::beg);

    if (size % 4 != 0) {
        file.close();
        throw std::runtime_error("Error: File size is not a multiple of 4 bytes.");
    }

    if (size == 0) {
        file.close();
        return {};
    }

    std::vector<uint32_t> rawProgram(size / 4);

    if (!file.read(reinterpret_cast<char*>(rawProgram.data()), size)) {
        file.close();
        throw std::runtime_error("Error: Failed to read file " + filePath);
    }

    file.close();

    return rawProgram;
}
```

6.2 Load-time과 Runtime의 이원화된 예외 처리 전략

오류가 발생하는 시점에 따라 예외 처리 전략을 이원화하여 시스템의 견고함을 높였다.

- Load-time Error: `InstructionFactory`에서 객체를 생성할 때, 존재하지 않는 Opcode나 유효하지 않은 `Flag` 비트(`isValidFlag`)를 즉시 감지하여 차단한다. 이는 실행 불가능한 코드가 VM에 진입하는 것을 방지한다.
- Runtime Error: 실행 중 발생하는 스택 오버플로우나 잘못된 메모리 접근은 `run()` 루프 내에서 감지되어 안전하게 종료된다.

6.3 사용자 정의 예외 클래스(VMException) 활용

표준 예외(`std::runtime_error`) 대신, 도메인에 특화된 `VMException` 클래스를 정의하여 사용하였다. 이 클래스는 단순한 에러 메시지뿐만 아니라, 오류가 발생한 `PC` 위치와 명령어 인덱스 정보를 함께 저장한다. 이를 통해 사용자는 에러 발생 시점의 정확한 문맥을 파악할 수 있어 디버깅 효율성이 향상되었다.

```cpp
class VMException : public std::runtime_error {
public:
    explicit VMException(const std::string& message, int pcIndex = -1)
        : std::runtime_error(message), m_pcIndex(pcIndex) {}

    [[nodiscard]] int getPCIndex() const {
        return m_pcIndex;
    }

    [[nodiscard]] std::string getFullMessage() const {
        if (m_pcIndex != -1) {
            return "Runtime Error at instruction [" + std::to_string(m_pcIndex) + "]: " + what();
        }
        return "Load/Parse Error: " + std::string(what());
    }

private:
    int m_pcIndex;
};

```

6.4 방어적 프로그래밍: Null Check 및 자원 한계 관리

가상 머신 실행 도중 발생할 수 있는 치명적 오류를 방지하기 위해 방어적 프로그래밍(Defensive Programming) 기법을 적용하였다. `run()` 루프에서는 명령어 객체에 접근하기 전 항상 포인터의 유효성(`nullptr` 여부)을 검사하며 , `VMContext`는 8비트 `PC` 레지스터의 한계(0~255)를 초과하는 프로그램 로드를 사전에 차단하여 버퍼 오버런 공격 가능성을 배제하였다.

7 테스트 결과 및 검증

7.1 테스트 환경 및 시나리오 구성

구현된 가상 머신(VM)의 정합성을 검증하기 위해, 직접 작성한 바이너리 파일을 대상으로 테스트를 수행하였다. 테스트 환경은 Windows 11 환경의 CLion에서 진행되었으며, 각 명령어이 의도한 대로 동작하는지를 중점적으로 확인하였다.

7.2 Case 1: MOV, PRINT 명령어 검증

가장 기초적인 명령어인 `PRINT`의 동작을 검증하였다. move_print.bin 파일은 레지스터에 값을 저장하고 이를 출력하는 단순한 로직을 담고 있다. 테스트 결과, 프로그램은 입력된 정수 값을 정확하게 표시하였다. 이는 `Instruction` 객체가 `VMContext`의 레지스터 값에 올바르게 접근하고 있음을 보여준다.

- 명령어 (test/bin/move_print.bin)

```
MOV R0, 10
PRINT R0
```

- 실행 결과

```
10
```

7.3 Case 2: 산술 연산 및 레지스터 검증

`ADD`, `SUB`, `MUL` 등 산술 연산 명령어의 정확성을 검증하였다. 레지스터 간의 덧셈과 뺄셈이 누적되어 결과 레지스터(`Dest`)에 반영되는지 확인하였다.

- 명령어 (test/bin/add.bin)

```
MOV R0, 10
PRINT R0
MOV R1, 10
ADD R0, R1
PRINT R0
```

- 실행 결과

```
10
20
```

7.4 Case 3: 스택 연산 및 분기 검증 (스택 cnp 둘 중 하나)

PUSH/POP을 통한 스택 메모리 조작과 BE/BNE를 이용한 흐름 제어를 검증하였다.

- LIFO 구조 확인: 데이터를 순서대로 Push한 뒤 Pop했을 때 역순으로 출력되는 LIFO(Last-In First-Out) 구조가 정확히 유지됨을 확인하였다.

  - 명령어 (test/bin/stack.bin)

```
PUSH 1
PUSH 2
PUSH 3
PUSH 4
PUSH 5
POP R0
PRINT R0
POP R0
PRINT R0
POP R0
PRINT R0
POP R0
PRINT R0
POP R0
PRINT R0
```

  - 실행결과

```
5
4
3
2
1
```

- 분기 제어: 비교 연산(CMP) 후 조건(ZF)에 따라 PC가 점프하여, 반복문이 정상적으로 수행되는 것을 확인하였다.
  - 명령어 (test/bin/cmp_be.bin)

```
MOV R0, 10
CMP R0, 10
BE 4
PRINT 99
PRINT 1
```

  - 실행 결과

8 결론 및 고찰

8.1 프로젝트 요약 및 요구사항 충족 여부

본 프로젝트를 통해 32비트 가상 명령어 포맷을 해석하고 실행하는 가상 머신(Virtual Machine)을 구현하였다. 과제 명세서에서 요구한 Command 패턴을 적용하여 명령어를 객체화하였으며, 비트 연산을 통한 디코딩, 레지스터 및 스택 시뮬레이션 등 핵심 요구사항을 충족하였다.

8.2 객체지향 설계를 통해 얻은 이점

단순한 기능 구현을 넘어, 심도 있는 객체지향 설계를 적용함으로써 다음과 같은 이점을 얻을 수 있었다.

- 확장성(Extensibility): Factory Method 패턴과 **`std::map`** 레지스트리를 활용하여, 기존 코드를 수정하지 않고도 새로운 명령어를 손쉽게 추가할 수 있는 OCP(개방-폐쇄 원칙) 구조를 완성하였다.
- 안정성(Safety): **`Enum Class`**와 강력한 타입 시스템을 도입하여 컴파일 타임에 논리적 오류를 차단하였으며, Load-time/Runtime 예외 처리를 이원화하여 시스템의 견고함을 확보하였다.
- 유지보수성(Maintainability) : Template Method 패턴(**`resolveValue`**)을 통해 중복 코드를 제거(DRY)하고, 각 클래스가 단일 책임(SRP)만을 가지도록 설계하여 코드의 가독성과 관리 용이성을 높였다.

8.3 향후 개선 방향

현재 시스템은 콘솔 기반의 입출력만을 지원하지만, 향후에는 **`VMContext`**의 상태를 시각적으로 보여주는 GUI 디버거를 연동하거나, 더 복잡한 명령어(함수 호출을 위한 CALL/RET 등)를 지원하도록 확장한다면 실제 CPU 아키텍처에 더 근접한 시뮬레이터로 발전시킬 수 있을 것이다.