

**2025학년도 2학기**  
**객체지향설계-00분반**

**텀프로젝트**

## 1. 텁프로젝트 주제 – 가상 명령어 해석기 구현

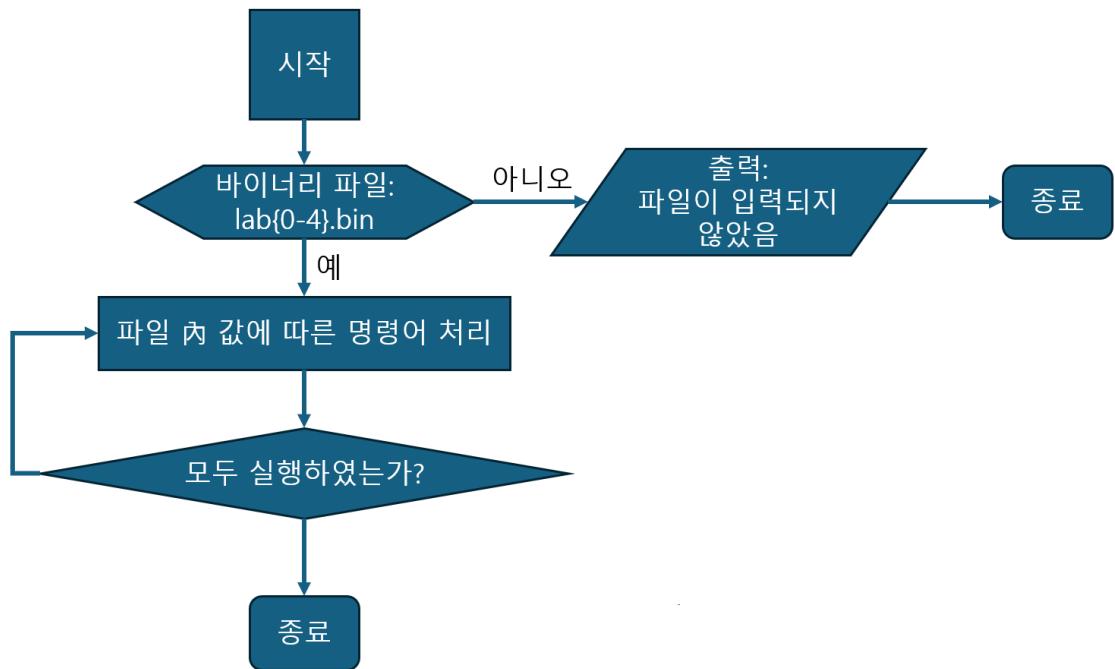
현대의 소프트웨어는 단순히 기능만 구현하는 것을 넘어, 코드 보안과 분석 저항성도 중요한 요소로 여겨지고 있다. 예를 들어, 악성코드 분석이나 소프트웨어 리버스 엔지니어링(reverse engineering)에서는 프로그램의 내부 동작을 이해하려는 시도가 이루어진다. 이를 어렵게 만들기 위한 기법 중 하나가 “가상 명령어 (Virtual Instruction)” 기반 코드 실행이다.

이 방식에서는 실제 CPU 명령어(예: x86, ARM 등)를 직접 사용하는 대신, 개발자가 임의로 정의한 가상의 명령어 집합을 사용한다. 프로그램은 이 가상 명령어로 구성된 바이트코드(bitstream) 형태로 작성되며, 실행 시에는 별도로 구현된 해석기(interpreter)가 이 명령어들을 읽고 의미를 해석하여 실제 동작을 수행한다. 즉, 프로그램의 실행 로직이 일반적인 어셈블리 코드나 기계어로 드러나지 않기 때문에, 외부에서 분석하거나 역공학하기가 매우 어려워진다.

본 프로젝트에서는 이러한 가상 명령어 해석기(Virtual Machine / Interpreter)를 직접 설계하고 구현해본다. 후술된 가상의 명령어 세트(예: MOV, ADD 등)를 정의하고, 입력으로 주어진 명령어 스트림(32비트 명령어 단위)을 해석하여 해당 연산을 수행하는 구조를 설계한다.

특히, 본 프로젝트에서는 객체지향 설계 원칙(OOP)과 디자인 패턴을 활용하여 확장 가능하고 유지보수가 용이한 해석기를 구현하는 것을 목표로 한다. 예를 들어, 새로운 명령어가 추가되거나 해석 방식이 바뀌더라도 기존 코드를 최소한만 수정하도록 설계해야 합니다.

아래는 제작할 프로그램을 간단히 표현한 순서도이다.



## 2. 조건

### 2-1. 구현을 위한 디자인 패턴

구현을 위해, 디자인 패턴은 **Command 패턴**을 사용한다.

### 2-2. 구현을 위한 클래스

구현을 위해, 다음과 같은 기능을 하는 class를 정의한다.

- **VMContext**: 이 클래스는 가상머신(가상 명령어의 집합)의 실행 상태를 관리한다. 레지스터의 값과, 코드를 직접 불러오는 역할을 수행한다.
- **Instruction**: 이 클래스는 명령어를 직접 실행하는 역할을 수행한다.

### 2-3. 입력 조건

입력으로는 Binary Representation으로 인코딩 된 **lab{0-4}.bin** 파일이 주어진다. 각 파일을 입력으로 받아 모든 명령을 정상적으로 실행해야 한다.

### 2-3. 구현해야 할 레지스터

Register	Binary Representation (8bit)	크기	기능
R0	0000 0001	8bit	일반 연산에 사용됨
R1	0000 0010	8bit	일반 연산에 사용됨
R2	0000 0011	8bit	일반 연산에 사용됨
PC	0000 0100	8bit	다음 번에 실행할 코드의 위치

			를 나타냄
SP	0000 0101	8bit	현재 스택의 Top 위치
BP	0000 0110	8bit	현재 스택의 Base 위치
ZF	0000 0111	8bit	연산의 결과가 0이면 설정된다.
CF	0000 1000	8bit	연산 중 자리 올림이 발생하면 설정된다.
OF	0000 1001	8bit	연산 중 오버플로우가 발생하면 설정된다.

#### 2-4. 구현해야 할 명령어

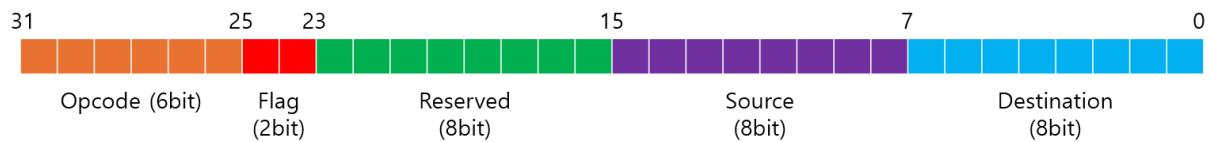
명령어는 다음을 구현하여야 한다. 표에서 REG는 레지스터를 의미하고, VAL은 정수이다.

명령어	Opcode (6bit)	Operand	결합 순서	설명
MOV	00 0001	[REG], [REG VAL]	<-	오른쪽 피연산자의 값을 왼쪽 피연산자로 옮긴다.
ADD	00 0010	[REG], [REG VAL]	<-	오른쪽 피연산자의 값과 왼쪽 피연산자의 값을 더해 왼쪽 피연산자에 저장한다.
SUB	00 0011	[REG], [REG VAL]	<-	왼쪽 피연산자의 값에 오른쪽 피연산자의 값을 뺀다. 결과를 왼쪽 피연산자에 저장한다.
MUL	00 0100	[REG], [REG VAL]	<-	왼쪽 피연산자의 값과 오른쪽 피연산자의 값을 곱하고, 그 값을 왼쪽 피연산자에 저장한다.
CMP	00 0101	[REG], [REG VAL]	<-	(내부적으로) SUB 연산을 수행한 뒤 값이 0이면 ZF를 1로 설정하고, 값이 1 이상이면 CF를 1로, -1 이하라면 OF를 1로 설정한다.
PUSH	00 0110	[REG]		피연산자의 값을 스택에 Push한다.
POP	00 0111	[REG]		SP가 가리키고 있는 곳의 값을 피연산자의 값으로 설정한다.
JMP	00 1000	[REG VAL]		피연산자의 값으로 PC를 설정한다.
BE	00 1001	[REG VAL]		ZF가 1이라면 피연산자의 값으로 점

				프한다.
BNE	00 1010	[REG VAL]		ZF가 1이 아니라면 피연산자의 값으로 점프한다.
PRINT	00 1011	[REG VAL]		피연산자의 값을 불러와 출력한다.

## 2-5 명령어 해석

각 명령어는 32바이트 단위로 해석된다. 아래 그림은 명령어의 구조를 나타낸다.



테스트 케이스로 주어질 파일은 Hex Data의 형식으로 주어진다. 비트 연산 등을 통해 명령어에서 각 부분을 추출해야 한다.

여기서 Flag는 2비트로서, 다음과 같은 경우의 수를 가진다.

- 00: Src, Dest 모두 Register인 경우
- 01: Dest는 Register, Src는 Value일 경우
- 10: 단일 Register만 Operand로 가지는 경우
- 11: 단일 Value만 Operand로 가지는 경우

Flag를 통해 명령어에서 받는 Operand가 Register인지, Value인지 구분하여 해석할 수 있도록 한다.

## 2-6. 명령어의 인코딩 예시

이 문단에서는 몇 가지 명령어의 인코딩 예시를 설명한다.

### mov r1, 10 의 경우

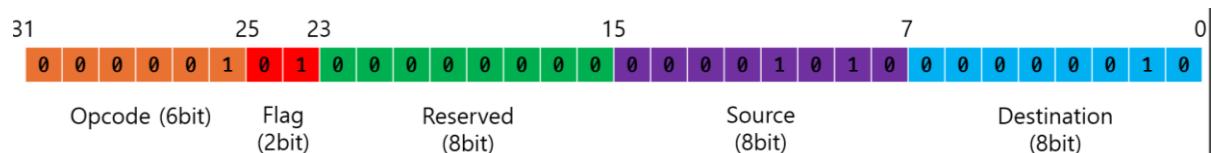
mov r1, 10은 10이라는 값을 r1 레지스터로 옮기는 동작을 의미한다. mov r1, 10은 다음 조건을 가지고 있다.

- Dest Operand는 Register, Src Operand는 Value이다.

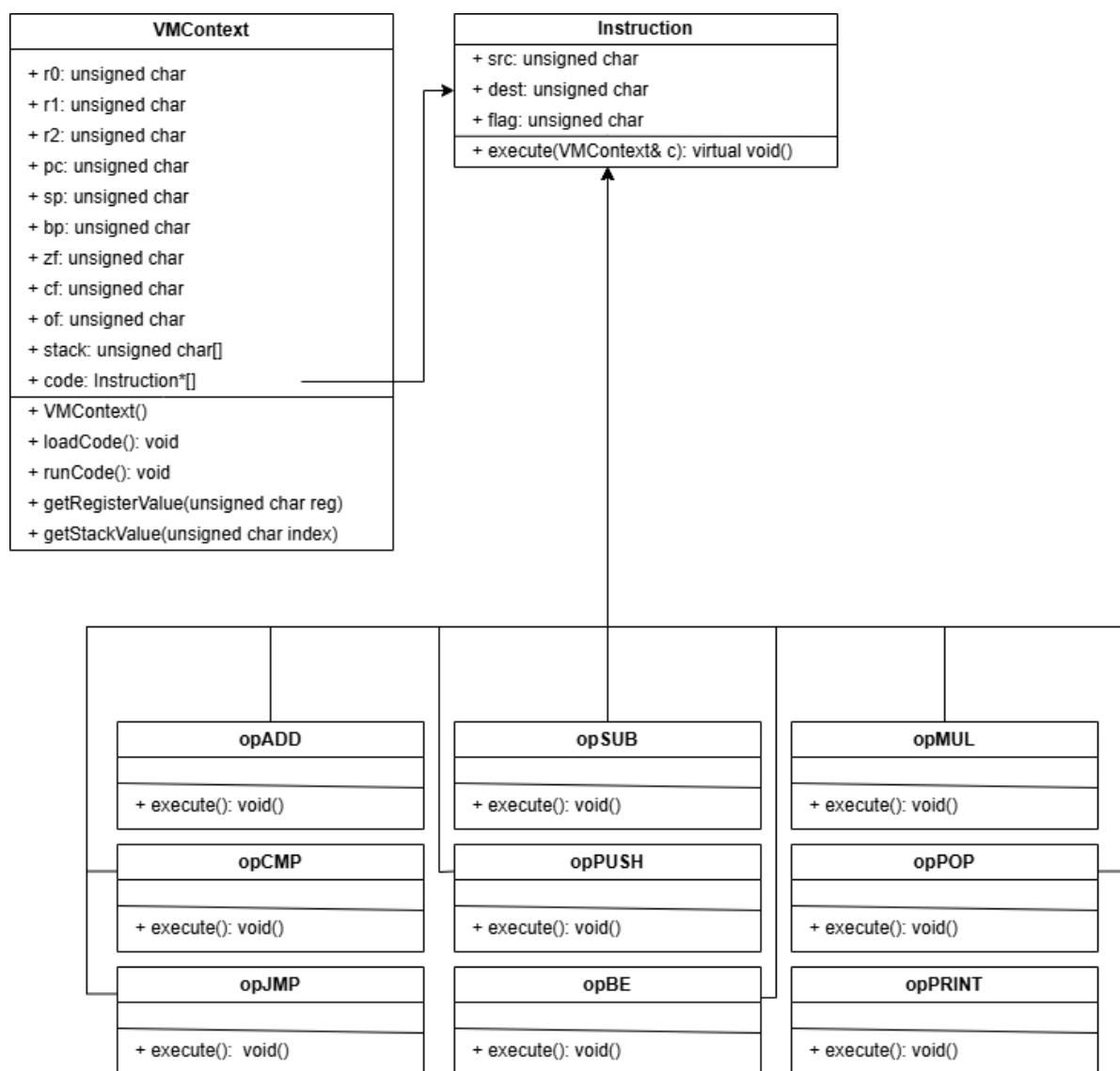
이에 따라, Flag (2bit)는 01 이 된다. 앞서 2-4. 와 2-3. 에서 기재한 표를 확인하면, MOV는 00 0001 이며, r1 레지스터는 0000 0010 이 된다. Reserved 공간은 예약된 공간이므로 무시한다. 여기서 정수 10은 값 그대로 명령어 한 줄에

들어간다.

즉 전체 인코딩을 진행한 그림은 다음과 같다.



## 2-7. 클래스 다이어그램



## 2-8. 테스트 케이스

테스트 케이스는 총 7가지로서, 다음과 같은 기능을 수행한다.

이 중 3가지 테스트 케이스만 공개되며, 4가지 테스트 케이스는 공개되지 않는다.

### 2-6-1. PRINT 테스트

PRINT 명령어를 구현해 값을 출력해야 하는 테스트 케이스이다. 성공하면, 다음과 같은 값을 출력한다.

<출력>

5
---

### 2-6-2. 덧셈 후 PRINT

ADD, PRINT 명령어를 구현하여 값을 출력해야 하는 테스트 케이스이다. 성공하면, 다음과 같은 값을 출력한다.

<출력>

10
10
20

### 2-6-3. PUSH, POP 테스트

PUSH, POP 명령어를 구현하여 오류없이 작동함을 확인해야 하는 테스트 케이스이다. 성공하면, 다음과 같은 값을 출력한다.

<출력>

1
2
3
4
5
4
3
2
1