문제 1: 다음 중 연산 코드(Opcode)와 오퍼랜드(Operand)의 관계에 대한 설명으로 틀린 것은?

① 연산 코드(Opcode)는 수행할 연산을 나타내며, 오퍼랜드(Operand)는 연산 대상 데이터를 지정한다.

② 일부 명령어는 오퍼랜드를 포함하지 않고 연산 코드만으로 동작할 수도 있다.

③ 오퍼랜드의 개수는 아키텍처에 따라 다르며, 0개에서 여러 개까지 존재할 수 있다.

④ 오퍼랜드로 레지스터, 메모리 주소, 즉시 값(Immediate Value) 등이 사용될 수 있다.

⑤ 모든 명령어는 반드시 하나 이상의 오퍼랜드를 가져야 한다.

문제 2: 다음 중 오퍼랜드의 종류에 대한 설명으로 틀린 것은?

① 즉시 값(Immediate Operand)은 연산에 사용할 데이터가 명령어 내부에 직접 포함된 형태이다.

② 메모리 오퍼랜드(Memory Operand)는 특정 주소에 저장된 값을 연산에 사용한다.

③ 레지스터 오퍼랜드(Register Operand)는 연산에 사용할 데이터를 CPU 내부의 레지스터에서 가져온다.

④ 연산 결과를 항상 메모리에 저장하는 방식이 가장 빠른 오퍼랜드 접근 방식이다.

⑤ 스택 오퍼랜드(Stack Operand)는 명령어 실행 중 스택을 활용하여 연산을 수행할 때 사용된다.

문제3: 다음은 파이썬으로 작성된 재귀 함수의 코드이다.

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

result = factorial(5)
print(result)
```

이 코드를 실행하면 factorial(5)가 호출되고, 이를 해결하기 위해 factorial(4), factorial(3), ..., factorial(0)까지 재귀적으로 호출되었다가 반환된다. PVM(Python Virtual Machine)은 이 과정을 실행하기 위해 바이트코드를 생성하고, 내부적으로 스택을 활용한다. 이를 간단한 어셈블리어(Pseudo x86-64)로 변환하면 아래와 같은 코드로 나타낼 수 있다.

```
factorial:
    cmp rdi, 0          ; if n == 0
    je base_case        ; return 1 if true

    push rdi            ; push n to stack
    dec rdi             ; n - 1
    call factorial      ; recursive call: factorial(n-1)

    pop rax             ; restore original n
    imul rax, rdi       ; multiply result with n
    ret                 ; return result

base_case:
    mov rax, 1          ; return 1
    ret
```

위 어셈블리 코드는 x86-64 기반의 의사 코드(Pseudo x86-64 Assembly) 로, 파이썬의 재귀 호출이 실제로 어떻게 스택을 활용하며 동작하는지를 나타낸다.

다음 중 PVM이 재귀 함수를 처리하는 과정에 대한 설명으로 틀린 것은?

① factorial(5)를 호출하면 PVM은 새로운 스택 프레임을 생성하여 함수의 매개변수 및 반환 주소를 저장한다.

② 각 재귀 호출 시, 새로운 스택 프레임이 생성되며 factorial(0)까지 도달한 후 반환되면서 프레임이 차례로 해제된다.

③ PVM은 바이트코드를 실행하는 동안 스택 기반 가상 머신(Stack-based VM) 방식으로 동작하며, 스택을 사용하여 연산을 수행한다.

④ PVM은 실행 중 불필요한 재귀 호출을 감지하여 자동으로 tail-call optimization (TCO)을 적용하여 최적화한다.

⑤ 바이트코드는 CPU가 직접 실행할 수 없으며, PVM이 이를 해석하여 실행하는 방식으로 동작한다.

문제4: 컴퓨터의 주소 지정 방식(Addressing Mode)에는 여러 가지 종류가 있으며, 즉시 주소 지정 방식(Immediate Addressing)과 직접 주소 지정 방식(Direct Addressing)은 자주 사용되는 방식 중 하나이다. 각 방식의 차이점과 특징을 고려할 때, 다음 중 틀린 설명을 고르시오.

① 즉시 주소 지정 방식에서는 연산에 사용할 값이 명령어 내부에 포함되며, 메모리 접근 없이 바로 사용될 수 있다.

② 직접 주소 지정 방식에서는 명령어가 지정한 주소의 메모리 위치에서 데이터를 가져와 연산을 수행한다.

③ 즉시 주소 지정 방식은 메모리 접근이 필요 없기 때문에 속도가 빠르지만, 저장할 수 있는 값의 크기가 명령어 크기에 제한을 받는다.

④ 직접 주소 지정 방식은 연산할 데이터를 메모리에서 가져와야 하므로 즉시 주소 지정 방식보다 실행 속도가 빠르다.

⑤ 즉시 주소 지정 방식은 주로 상수 값을 사용할 때 유용하며, 직접 주소 지정 방식은 변수에 저장된 값을 참조할 때 유용하다.

1.정답: ⑤

설명: ⑤번 문장이 틀린 이유는, 모든 명령어가 오퍼랜드를 필요로 하는 것은 아니기 때문이다. 예를 들어, NOP(No Operation) 같은 명령어나 RET(Return) 같은 명령어는 오퍼랜드 없이 동작한다. 또한, 명령어 구조에 따라 스택 기반 연산(Stack-based operation) 방식에서는 오퍼랜드 없이 스택에서 값을 가져와 연산하는 방식도 존재한다.

2.정답: ④

설명: ④번 문장이 틀린 이유는, 연산 결과를 메모리에 저장하는 방식이 가장 느린 방식 중 하나이기 때문이다. CPU가 데이터를 처리할 때, 레지스터 → 캐시 → 메모리(RAM) → 디스크 순으로 속도가 느려진다. 즉, 레지스터 오퍼랜드를 사용하는 방식이 가장 빠르며, 메모리에 접근하는 방식은 상대적으로 느리다.

3.정답: ④

설명: ④번 문항이 틀린 이유는, 파이썬의 PVM은 Tail Call Optimization(TCO)을 지원하지 않기 때문이다. 재귀 호출이 계속되면 스택 프레임이 증가하며 결국 "RecursionError" 가 발생할 수 있다. TCO는 특정 언어(C, Scheme 등)에서 재귀를 반복문으로 변환하여 최적화하는 기법이지만, 파이썬에서는 이를 적용하지 않고 재귀 호출마다 새로운 스택 프레임을 계속 생성한다.

4.정답: ④

설명: ④번 문항이 틀린 이유는 직접 주소 지정 방식(Direct Addressing)은 메모리 접근이 필요하므로 즉시 주소 지정 방식(Immediate Addressing)보다 느리다. 즉시 주소 지정 방식은 값이 명령어 내부에 포함되므로 메모리 접근이 필요하지 않아 속도가 빠르다. 반면, 직접 주소 지정 방식은 메모리에 있는 데이터를 가져와야 하므로 메모리 접근 오버헤드가 발생하여 속도가 상대적으로 느리다.