

이 자료는 대한민국 **저작권법의 보호**를 받습니다.

작성된 모든 내용의 권리는 작성자에게 있으며, 작성자의 동의 없는 사용이 금지됩니다.

본 자료의 일부 혹은 전체 내용을 무단으로 복제/배포하거나 2차적 저작물로 재편집하는 경우,

5년 이하의 징역 또는 5천만 원 이하의 벌금과 민사상 손해배상을 청구합니다.

이 문서는 총 9챕터 中
1개의 챕터 무료 공개본입니다.

UVM Verification Basic 핸드북
전체(9챕터)를 원하신다면
아래 링크 확인해주세요.

<https://www.inflearn.com/clip/245>

Project: Combinational Adder

내용

.....	1
이 문서는 총 9챕터 中 1챕터 무료 공개본입니다.	2
UVM Verification Basic 핸드북 전체(9챕터)를 원하신다면 아래 링크 확인해주세요.	2
https://www.inflearn.com/clip/245	2
Project: Combinational Adder	
.....	3
1. 하드웨어 검증의 기본 개념과 UVM의 필요성	10
하드웨어 검증이란 무엇인가?	10
전통적 검증 방법의 한계점	10
UVM의 등장과 해결책	12
UVM의 핵심 개념들	13
2. DUT와 Interface 분석 - 검증 대상의 완전한 이해	14
DUT (Design Under Test) 분석	14
검증 관점에서의 DUT 분석	15
Interface 설계와 중요성	15
DUT와 Interface의 관계 다이어그램	17
3. UVM Transaction Class - 데이터 모델링의 출발점	18
Transaction Class의 본질적 이해	18
제공된 Transaction Class 분석	18
UVM 자동화 매크로의 이해	20

Transaction Class의 역할과 데이터 플로우	21
실무에서의 Transaction 설계 고려사항	22
4. UVM Sequence Class - 테스트 시나리오 생성기	23
Sequence Class의 핵심 개념	23
전통적 테스트 생성 vs UVM Sequence	23
제공된 Sequence Class 분석	23
Transaction 객체 생성과 관리	25
Sequence 실행 프로토콜	25
랜덤화(Randomization) 과정	27
디버깅과 로깅	27
실무에서의 Sequence 설계 패턴	28
Sequence 사용 시 주의사항과 실수들	29
5. UVM Driver Class - 소프트웨어와 하드웨어 간의 브릿지	30
Driver Class의 본질적 역할	30
Driver Class의 구조 분석	30
Driver의 핵심 멤버 변수들	31
Build Phase - 설정과 초기화	32
Run Phase - 실제 구동 로직	33
Driver의 역할을 그림으로 이해하기	36
실무에서의 Driver 설계 고려사항	36
Driver 사용 시 주의사항	38
6. UVM Monitor Class - 하드웨어 신호 관찰자	39
Monitor Class의 본질적 역할과 필요성	39
Monitor가 필요한 이유	39

제공된 Monitor Class 구조 분석.....	40
Build Phase - Monitor 초기화	41
Run Phase - 실제 모니터링 로직	42
Analysis Port 통신 매커니즘.....	44
Monitor 사용 패턴과 변형.....	44
Monitor 설계 시 주의사항.....	45
실무에서의 Monitor 활용 패턴.....	47
7. UVM Scoreboard Class - 결과 검증과 판정	49
Scoreboard Class 의 핵심 역할	49
하드웨어 검증에서 Scoreboard 의 필요성	49
제공된 Scoreboard Class 분석	50
Build Phase 분석.....	51
핵심 기능 - write() 함수 구현	51
다양한 검증 시나리오	53
고급 Scoreboard 설계 패턴.....	55
Scoreboard 사용 시 주의사항	56
실무에서의 Scoreboard 활용.....	58
8. UVM Agent Class - 검증 컴포넌트 관리자	60
Agent Class 의 본질과 필요성	60
Agent 의 구조적 역할.....	61
제공된 Agent Class 분석	61
Build Phase - 컴포넌트 생성.....	63
Connect Phase - 컴포넌트 연결.....	64
Agent 의 설정과 모드 관리.....	65

Agent 내부 통신 흐름.....	66
다중 Agent 환경.....	67
Agent 사용 시 주의사항	68
실무에서의 Agent 설계 패턴	69
9. UVM Environment Class - 검증 환경 통합 관리.....	70
Environment Class의 핵심 역할.....	70
검증 환경에서 Environment의 위치.....	70
제공된 Environment Class 분석.....	70
Build Phase - 컴포넌트 생성과 설정	72
Connect Phase - 컴포넌트 간 연결.....	73
복잡한 Environment의 연결 패턴	74
Environment 설정 관리 패턴	76
Environment의 실행 제어.....	77
Environment 사용 시 주의사항	78
실무에서의 Environment 설계 패턴.....	80
10. UVM Test Class - 검증 시나리오 제어기.....	81
Test Class의 본질적 역할	81
검증 환경에서 Test의 위치	82
제공된 Test Class 분석.....	82
Build Phase - 검증 환경 구축	83
Run Phase - 테스트 시나리오 실행	84
다양한 Test 시나리오 패턴.....	86
Test 설정 및 환경 제어.....	88
고급 Test 제어 기능	89

Test 사용 시 주의사항.....	92
실무에서의 Test 설계 패턴.....	93
11. Testbench Top Module - 하드웨어와 소프트웨어의 만남	95
Testbench Top의 핵심 역할	95
하드웨어와 소프트웨어의 경계	96
제공된 Testbench Top Module 분석	96
UVM 환경 연결과 실행	98
Testbench Top에서의 클럭 생성	99
다중 Interface 환경	101
커맨드라인 인터페이스	102
Testbench Top 설계 시 주의사항	103
실무에서의 Testbench Top 패턴	104
12. 전체 시뮬레이션 플로우와 실무 적용.....	106
UVM 검증 환경의 전체 실행 플로우	106
Phase별 실행 순서	106
데이터 플로우 추적	107
시뮬레이션 로그 분석	108
타이밍 다이어그램	109
실무에서의 검증 시나리오	110
검증 메트릭과 분석	111
디버깅과 문제 해결	113
성능 최적화	115
실무 프로젝트 구조	117
Makefile 예시	118

지속적 통합(CI) 연동.....	118
13. 이해도 확인 퀴즈	122
문제 1: UVM Transaction의 기본 개념.....	122
문제 2: Sequence의 실행 프로토콜.....	124
문제 3: Driver의 Interface 연결.....	126
문제 4: Monitor의 Analysis Port.....	128
문제 5: Scoreboard의 검증 로직.....	130
문제 6: Agent의 Active vs Passive 모드.....	132
문제 7: Environment의 Connect Phase	134
문제 8: Sequence 실행 제어.....	136
문제 9: UVM Configuration Database.....	139
문제 10: TLM 통신의 종류.....	142
문제 11: UVM Phase 실행 순서	144
문제 12: Transaction의 제약 조건 (Constraint).....	147
문제 13: Driver의 non-blocking 할당	149
문제 14: Monitor의 데이터 수집 타이밍	153
문제 15: Virtual Sequencer 패턴	156
문제 16: Coverage Closure 전략.....	160
문제 17: UVM Factory Override.....	164
문제 18: Callback Mechanism.....	169
문제 19: Register Model (RAL) 통합.....	174
문제 20: 전체 검증 플로우 이해	180
14. 실습 연습문제.....	185
연습문제 1: 기본 4비트 비교기 검증 환경	185

연습문제 2: 타이머 모듈 검증 (클럭 동기식)	195
연습문제 3: 간단한 UART 송신기 검증	204
연습문제 4: FIFO 버퍼 검증 (동기식).....	214
연습문제 5: 간단한 ALU 서브시스템 검증.....	222
15. IEEE 1800-2017 SystemVerilog LRM 크로스레퍼런스.....	229
핵심 개념별 LRM 매팅	229
실무 활용 워크플로우	245
효율적인 LRM 탐색 전략.....	249

1. 하드웨어 검증의 기본 개념과 UVM의 필요성

하드웨어 검증이란 무엇인가?

하드웨어 검증은 설계된 디지털 회로가 원하는 기능을 올바르게 수행하는지 확인하는 과정입니다. 이는 소프트웨어의 디버깅과 비슷하지만, 근본적인 차이점이 있습니다.

구분	소프트웨어	하드웨어
수정 가능성	언제든 코드 수정 가능	칩 제작 후 수정 거의 불가능
테스트 비용	상대적으로 저렴	칩 제작 비용 매우 높음
오류 발견 시점	런타임에서도 발견 가능	제작 전에 모든 오류 발견 필수
병렬성	순차 실행이 기본	모든 것이 동시에 실행
시간 개념	논리적 시간	물리적 시간 (클럭, 딜레이)

이러한 차이점 때문에 하드웨어 검증은 소프트웨어보다 훨씬 더 체계적이고 완전해야 합니다. 한번 실리콘으로 만들어진 칩에서 버그가 발견되면, 수십억 원의 손실이 발생할 수 있기 때문입니다.

전통적 검증 방법의 한계점

초기 하드웨어 검증은 다음과 같은 간단한 형태였습니다:

```
// 전통적인 테스트벤치 - 문제점이 많음
module simple_testbench;
    reg [3:0] a, b;
    wire [4:0] y;

    add dut(.a(a), .b(b), .y(y));

    initial begin
        // 하드코딩된 테스트 케이스들
        a = 4'b0001; b = 4'b0010; #10; // 1 + 2 = 3
        a = 4'b0011; b = 4'b0100; #10; // 3 + 4 = 7
        a = 4'b1111; b = 4'b0001; #10; // 15 + 1 = 16
        $finish;
    end
endmodule
```

위 코드의 문제점들을 분석해보겠습니다:

문제점 1: 테스트 케이스의 한계

- 4비트 입력의 경우 가능한 조합이 $16 \times 16 = 256$ 가지인데, 위 코드는 단 3가지만 테스트
- 경계 조건(boundary condition) 테스트 부족
- 랜덤 테스트 불가능

문제점 2: 결과 검증의 수동화

- 예상 결과를 눈으로 직접 확인해야 함
- 자동화된 pass/fail 판정 없음
- 대규모 테스트에서는 실용성 제로

문제점 3: 재사용성 부족

- 다른 덧셈기 검증 시 처음부터 다시 작성해야 함
- 코드의 모듈화가 전혀 되어 있지 않음

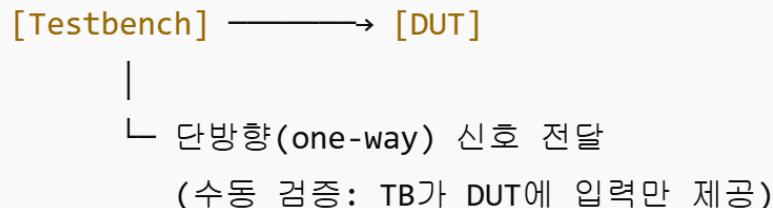
문제점 4: 확장성 부족

- 복잡한 DUT에 적용하기 어려움
- 여러 테스트 시나리오를 관리하기 어려움

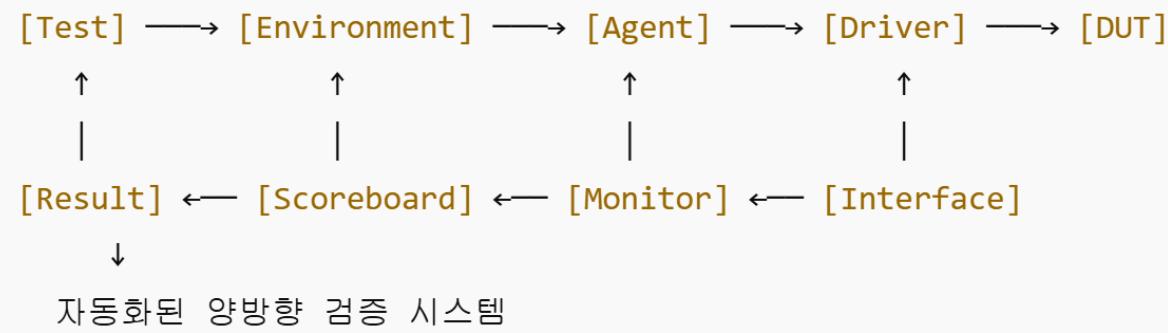
UVM의 등장과 해결책

UVM은 이러한 문제들을 체계적으로 해결하기 위해 등장했습니다. UVM의 핵심 철학을 이해해보겠습니다.

전통적 검증 구조:



UVM 검증 구조:



이 구조를 보면 UVM이 해결하는 핵심 문제들이 보입니다:

1. 계층적 구조: 각 컴포넌트가 명확한 역할을 가짐
2. 재사용성: 각 컴포넌트를 다른 프로젝트에서 재사용 가능
3. 자동화: 테스트 생성부터 결과 검증까지 모두 자동화
4. 확장성: 복잡한 시스템도 동일한 구조로 검증 가능

UVM의 핵심 개념들

UVM을 이해하기 위해 반드시 알아야 할 핵심 개념들을 정리해보겠습니다:

개념	설명	소프트웨어 비유
Transaction	하드웨어에 전달할 데이터 묶음	API 호출의 파라미터 구조체
Sequence	테스트 시나리오 생성기	테스트 케이스 제너레이터
Driver	소프트웨어 데이터를 하드웨어 신호로 변환	디바이스 드라이버
Monitor	하드웨어 신호를 소프트웨어 데이터로 변환	시스템 모니터
Scoreboard	결과 검증 및 비교	단위 테스트의 assert
Agent	Driver 와 Monitor 관리	컴포넌트 매니저
Environment	전체 검증 환경 관리	테스트 환경 설정
Test	검증 시나리오 제어	메인 테스트 함수

중요한 오해 방지 포인트: 많은 학생들이 UVM을 단순히 복잡하게 만든 테스트벤치로 생각합니다. 하지만 UVM의 진정한 가치는 복잡성에 있는 것이 아니라, 체계적인 구조화와 재사용성에 있습니다. 마치 게임 엔진을 사용하는 것과 같습니다. 처음에는 복잡해 보이지만, 한번 익히면 어떤 게임이든 효율적으로 개발할 수 있게 되는 것처럼 말입니다.

2. DUT 와 Interface 분석 - 검증 대상의 완전한 이해

DUT (Design Under Test) 분석

검증을 시작하기 전에 먼저 검증 대상인 DUT를 완전히 이해해야 합니다. 이는 게임을 테스트하기 전에 게임의 규칙을 완전히 파악하는 것과 같습니다.

```
// DUT: 4비트 조합 회로 덧셈기
module add(
    input [3:0] a,b,      // 4비트 입력 두 개
    output [4:0] y        // 5비트 출력 (캐리 비트 포함)
);
    assign y = a + b;    // 조합 회로로 구현된 덧셈
endmodule
```

이 간단한 코드에서 중요한 포인트들을 분석해보겠습니다:

입력 분석:

- a, b: 각각 4비트 unsigned 정수
- 가능한 값의 범위: 0 ~ 15
- 총 가능한 입력 조합: $16 \times 16 = 256$ 가지

출력 분석:

- y: 5비트 unsigned 정수
- 가능한 값의 범위: 0 ~ 31 (최대값: $15 + 15 = 30$)
- 5비트인 이유: 캐리(carry) 비트를 포함하기 위함

동작 특성:

- 조합 회로(Combinational Circuit): 클럭 없이 입력이 바뀌면 즉시 출력 변화
- 전파 지연(PROPAGATION DELAY): 실제 하드웨어에서는 입력 변화 후 출력까지 약간의 시간 소요

검증 관점에서의 DUT 분석

검증 엔지니어의 관점에서 이 DUT를 바라보면 다음과 같은 질문들이 나옵니다:

검증해야 할 사항들:

1. 모든 입력 조합에 대해 올바른 덧셈 결과가 나오는가?
2. 캐리 비트가 올바르게 처리되는가?
3. 경계 조건에서 올바르게 동작하는가?
 - 최소값 + 최소값 ($0 + 0 = 0$)
 - 최대값 + 최대값 ($15 + 15 = 30$)
 - 캐리가 발생하는 경우들
4. 타이밍은 올바른가?

Interface 설계와 중요성

```
// Interface: DUT 와 테스트벤치 간의 연결 통로
interface add_if();
    logic [3:0] a;          // 입력 a
    logic [3:0] b;          // 입력 b
    logic [4:0] y;          // 출력 y
endinterface
```

Interface 가 필요한 이유:

소프트웨어에서는 함수 호출을 통해 데이터를 주고받지만, 하드웨어에서는 물리적인 신호선(wire)을 통해 데이터가 전달됩니다. Interface는 이러한 물리적 연결을 추상화한 것입니다.

측면	직접 연결	Interface 사용
신호 관리	각 모듈이 개별적으로 신호 선언	중앙 집중식 신호 관리
변경 용이성	신호 변경 시 모든 모듈 수정 필요	Interface만 수정하면 됨
가독성	신호가 분산되어 파악 어려움	관련 신호들이 그룹화됨
재사용성	낮음	높음

Interface 사용의 장점을 구체적으로 보면:

```
// Interface 없이 직접 연결하는 경우
module testbench;
    logic [3:0] tb_a, tb_b;
    logic [4:0] tb_y;

    add dut(.a(tb_a), .b(tb_b), .y(tb_y));

    // Driver에서 tb_a, tb_b 제어
    // Monitor에서 tb_y 관찰
    // 신호가 분산되어 관리 복잡
endmodule

// Interface 사용하는 경우
module testbench;
    add_if aif();

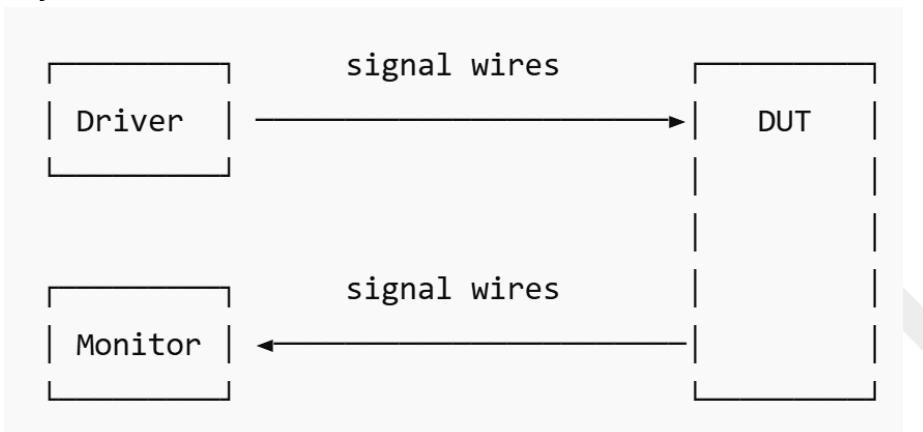
    add dut(.a(aif.a), .b(aif.b), .y(aif.y));

    // 모든 신호가 aif로 그룹화되어 관리 용이
endmodule
```

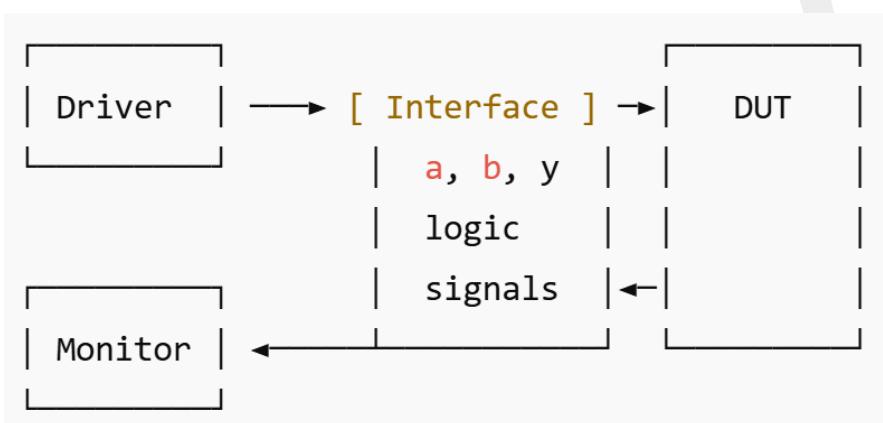
중요한 오해 방지 포인트: 많은 학생들이 Interface를 단순한 신호 그룹화 도구로만 생각합니다. 하지만 Interface의 진정한 가치는 하드웨어 신호와 소프트웨어 객체 사이의 추상화 계층을 제공한다는 것입니다. 이를 통해 복잡한 하드웨어 신호들을 객체지향적으로 다룰 수 있게 됩니다.

DUT 와 Interface 의 관계 다이어그램

Physical Hardware Level:



Interface Abstraction Level:



이 다이어그램에서 볼 수 있듯이, Interface 는 물리적 신호선들을 논리적으로 그룹화하여 관리를 용이하게 합니다. Driver 는 Interface 를 통해 DUT 에 데이터를 전달하고, Monitor 는 Interface 를 통해 DUT 의 출력을 관찰합니다.

실무에서의 Interface 활용:

- 큰 프로젝트에서는 수백 개의 신호가 있을 수 있음
- Interface 없이는 신호 관리가 거의 불가능
- Protocol 별로 Interface 를 정의하여 재사용성 극대화
- 예: AXI_interface, UART_interface, SPI_interface 등

이제 DUT 와 Interface 에 대한 완전한 이해를 바탕으로, UVM 컴포넌트들이 어떻게 이들과 상호작용하는지 알아보겠습니다.

3. UVM Transaction Class - 데이터 모델링의 출발점

Transaction Class의 본질적 이해

Transaction Class는 UVM 검증 환경에서 데이터를 표현하는 기본 단위입니다. 이를 이해하기 위해 먼저 하드웨어와 소프트웨어 간의 데이터 표현 차이를 살펴보겠습니다.

하드웨어 vs 소프트웨어 데이터 표현:

측면	하드웨어	소프트웨어
데이터 형태	개별 신호선의 0/1 값들	구조화된 객체
시간성	특정 시간에 신호값 존재	메모리에 지속적 존재
접근 방식	신호선 이름으로 직접 접근	멤버 변수로 접근
그룹화	물리적으로 분리된 신호들	논리적으로 연관된 데이터 집합

Transaction Class는 이러한 차이를 메우는 브릿지 역할을 합니다.

제공된 Transaction Class 분석

```
// Transaction Class - 덧셈기 검증용 데이터 모델
class transaction extends uvm_sequence_item;
rand bit [3:0] a;      // 랜덤 생성 가능한 입력 a
rand bit [3:0] b;      // 랜덤 생성 가능한 입력 b
bit [4:0] y;          // 출력값 (랜덤 생성 불가)

function new(input string path = "transaction");
    super.new(path);   // 부모 클래스 생성자 호출
endfunction

// UVM 자동화 매크로들
`uvm_object_utils_begin(transaction)
`uvm_field_int(a, UVM_DEFAULT)
`uvm_field_int(b, UVM_DEFAULT)
```

```
`uvm_field_int(y, UVM_DEFAULT)
`uvm_object_utils_end

endclass
```

코드 분석 포인트별 설명:

1. 클래스 상속 구조:

```
class transaction extends uvm_sequence_item;
```

- `uvm_sequence_item`을 상속받음으로써 UVM의 표준 기능들을 자동으로 획득
- 복사, 비교, 출력 등의 기본 기능들이 자동 제공
- UVM 인프라와의 호환성 보장

중요한 오해 방지: 많은 학생들이 왜 단순한 구조체 대신 클래스를 사용하는지 궁금해합니다. 클래스를 사용하는 이유는 데이터뿐만 아니라 해당 데이터를 조작하는 메서드들도 함께 포함할 수 있기 때문입니다.

2. 랜덤 변수 선언:

```
rand bit [3:0] a;
rand bit [3:0] b;
```

- `rand` 키워드: SystemVerilog의 제약 랜덤화 기능
- 자동으로 랜덤값 생성 가능
- 제약 조건(constraint)을 통해 특정 범위나 조건의 값만 생성 가능

3. 출력 변수:

```
bit [4:0] y;
```

- `rand` 키워드가 없음: 출력은 DUT에서 나오는 것이므로 랜덤 생성 대상이 아님
- Monitor에서 DUT의 출력을 관찰하여 이 변수에 저장

4. 생성자 함수:

```
function new(input string path = "transaction");
    super.new(path);
endfunction
```

- 객체 생성 시 호출되는 함수
- path 파라미터: UVM의 계층적 이름 관리를 위한 것
- super.new(): 부모 클래스의 생성자 호출

UVM 자동화 매크로의 이해

```
`uvm_object_utils_begin(transaction)
`uvm_field_int(a, UVM_DEFAULT)
`uvm_field_int(b, UVM_DEFAULT)
`uvm_field_int(y, UVM_DEFAULT)
`uvm_object_utils_end
```

이 매크로들이 하는 일을 표로 정리하면:

매크로	생성되는 기능	설명
uvm_object_utils_begin	타입 등록	UVM 팩토리에 클래스 등록
uvm_field_int	자동 메서드 생성	copy, compare, print 등
uvm_object_utils_end	매크로 종료	자동화 블록 종료

매크로가 자동 생성하는 메서드들:

```
// 매크로에 의해 자동 생성되는 기능들 (실제로는 보이지 않음)
function void do_copy(uvm_object rhs);
    // 객체 복사 기능
endfunction

function bit do_compare(uvm_object rhs);
    // 객체 비교 기능
endfunction

function void do_print();
    // 객체 출력 기능
Endfunction
```

매크로 사용의 장단점:

장점	단점
반복 코드 작성 불필요	내부 동작이 숨겨져 있음
표준화된 기능 제공	디버깅 시 복잡성 증가
실수 방지	매크로 문법 학습 필요

Transaction Class 의 역할과 데이터 플로우

Transaction Class 가 UVM 환경에서 어떻게 사용되는지 데이터 플로우를 통해 알아보겠습니다:

데이터 플로우:

1. [Sequence] → Transaction 객체 생성 및 랜덤화

```
transaction t = transaction::type_id::create("t");
t.randomize(); // a, b에 랜덤값 자동 할당
```

2. [Driver] → Transaction의 입력값을 하드웨어 신호로 변환

```
aif.a <= tc.a;
aif.b <= tc.b;
```

3. [DUT] → 하드웨어 연산 수행

```
y = a + b; (하드웨어 내부)
```

4. [Monitor] → 하드웨어 출력을 Transaction 객체로 변환

```
t.a = aif.a;
t.b = aif.b;
t.y = aif.y;
```

5. [Scoreboard] → Transaction 객체의 값을 이용한 검증

```
if(tr.y == tr.a + tr.b) pass; else fail;
```

이 플로우에서 중요한 점은 동일한 Transaction 클래스가 다양한 컴포넌트에서 서로 다른 용도로 사용된다는 것입니다:

컴포넌트	Transaction 사용 목적	주요 사용 필드
Sequence	테스트 입력 생성	a, b (rand로 생성)
Driver	하드웨어에 자극 전달	a, b (읽기)
Monitor	하드웨어 응답 수집	a, b, y (쓰기)
Scoreboard	결과 검증	a, b, y (읽기)

실무에서의 Transaction 설계 고려사항

1. 필드 선택의 기준:

- 입력 신호: rand 키워드 사용
- 출력 신호: 일반 변수로 선언
- 제어 신호: 필요에 따라 rand 또는 일반 변수

2. 제약 조건 추가 예시:

```
class transaction extends uvm_sequence_item;
rand bit [3:0] a, b;
bit [4:0] y;

// 제약 조건 추가 가능
constraint valid_range {
    a inside {[0:15]}; // 불필요하지만 명시적 표현
    b inside {[0:15]};
}

constraint corner_cases {
    // 경계값 테스트를 위한 제약
    a inside {0, 15} || b inside {0, 15};
}
endclass
```

3. 실무에서 자주하는 실수들:

- 출력 신호에 rand 키워드 사용
- 매크로 사용 시 필드 누락
- Transaction 객체의 생명주기 관리 소홀

중요한 개념 정리: Transaction Class는 단순한 데이터 컨테이너가 아닙니다. UVM 환경에서 모든 컴포넌트 간의 데이터 교환을 담당하는 핵심 인터페이스입니다. 이를 통해 복잡한 하드웨어 신호들을 객체지향적으로 관리할 수 있게 됩니다.

4. UVM Sequence Class - 테스트 시나리오 생성기

Sequence Class의 핵심 개념

Sequence Class는 UVM에서 테스트 시나리오를 생성하는 핵심 컴포넌트입니다. 이를 이해하기 위해 먼저 테스트 시나리오가 무엇인지 명확히 해야 합니다.

테스트 시나리오란?

- 특정한 목적을 가진 입력 데이터의 연속
- 예: "모든 경계값 테스트", "랜덤 값 1000 번 테스트", "특정 패턴 테스트"

소프트웨어 테스트에서는 테스트 케이스를 수동으로 작성하지만, 하드웨어 테스트에서는 수많은 테스트 케이스가 필요하므로 자동화된 생성이 필수입니다.

전통적 테스트 생성 vs UVM Sequence

방식	전통적 방법	UVM Sequence
테스트 생성	하드코딩된 값들	자동화된 랜덤 생성
확장성	값 추가 시 코드 수정 필요	반복 횟수만 변경
재사용성	다른 DUT에 적용 어려움	동일한 구조로 재사용 가능
제어 가능성	제한적	제약 조건으로 세밀한 제어

제공된 Sequence Class 분석

```
// Sequence Class - 테스트 시나리오 생성기
class generator extends uvm_sequence #(transaction);
`uvm_object_utils(generator)

transaction t;    // Transaction 객체 핸들
integer i;        // 반복문용 변수
```

```

function new(input string path = "generator");
    super.new(path);
endfunction

virtual task body();
    t = transaction::type_id::create("t"); // Transaction 객체 생성
    repeat(10) // 10 번 반복
        begin
            start_item(t); // Sequencer에게 아이템 시작 알림
            t.randomize(); // Transaction의 랜덤 필드들 자동 생성
            `uvm_info("GEN",$sformatf("Data send to Driver a :%0d ,
b :%0d",t.a,t.b), UVM_NONE);
            finish_item(t); // Sequencer에게 아이템 완료 알림
        end
    endtask

endclass

```

코드 세부 분석:

1. 클래스 선언부:

```
class generator extends uvm_sequence #(transaction);
```

- uvm_sequence 를 상속받아 UVM의 Sequence 기능 활용
- #(transaction): 템플릿 매개변수로 사용할 Transaction 타입 지정
- 이를 통해 타입 안전성(type safety) 보장

중요한 개념: 여기서 generator라는 이름을 사용했지만, UVM 표준에서는 sequence라는 이름을 더 선호합니다. 하지만 기능적으로는 동일합니다.

2. 멤버 변수들:

```
transaction t;
integer i;
```

- t: Transaction 객체를 가리키는 핸들(참조)
- i: 반복문에서 사용할 정수 변수 (실제로는 사용되지 않음)

3. 핵심 메서드 - **body()** task:

```
virtual task body();
```

- UVM에서 Sequence가 실행될 때 자동으로 호출되는 메서드
- virtual: 서브클래스에서 오버라이드 가능
- task: SystemVerilog의 시간을 소모할 수 있는 함수

Transaction 객체 생성과 관리

```
t = transaction::type_id::create("t");
```

이 코드는 UVM Factory Pattern을 사용한 객체 생성입니다. 일반적인 객체 생성과 비교해보겠습니다:

방식	코드	특징
일반적 생성	t = new();	컴파일 타임에 타입 고정
UVM Factory	t = transaction::type_id::create("t");	런타임에 타입 변경 가능

UVM Factory 사용의 장점:

- 테스트 시 다른 Transaction 타입으로 교체 가능
- 확장된 Transaction 클래스 사용 가능
- 디버깅과 추적이 용이함

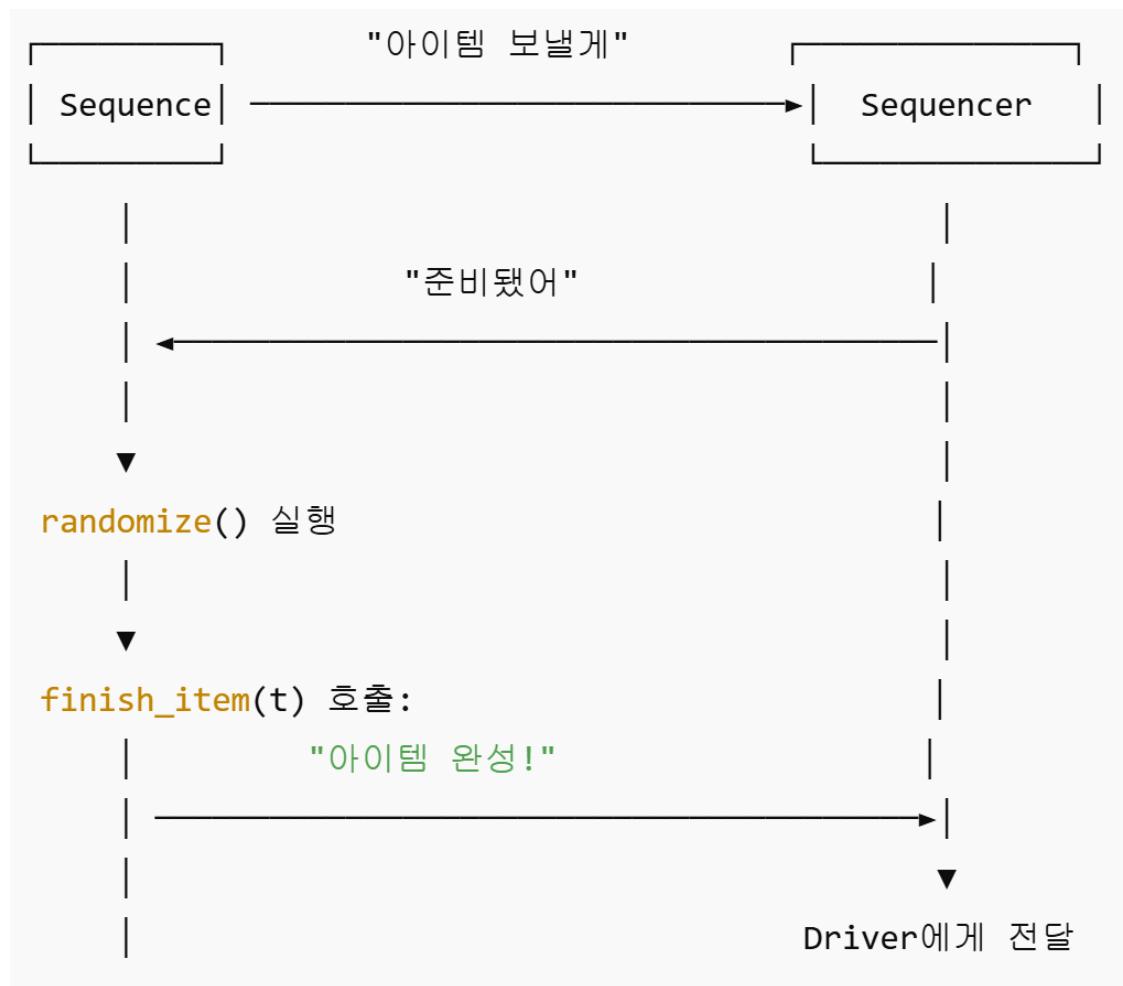
Sequence 실행 프로토콜

Sequence에서 가장 중요한 부분은 `start_item()`과 `finish_item()`의 이해입니다:

```
repeat(10)
begin
    start_item(t);      // 1. Sequencer에게 아이템 전송 시작 요청
    t.randomize();      // 2. Transaction 데이터 생성
    `uvm_info("GEN",...); // 3. 디버깅 정보 출력
    finish_item(t);     // 4. Sequencer에게 아이템 전송 완료 알림
end
```

각 단계의 내부 동작:

`start_item(t)` 호출 시:



중요한 타이밍 이해: `start_item()`과 `finish_item()` 사이에서 Transaction이 생성되고 설정됩니다. 이 구간에서 Driver가 이전 Transaction을 처리하고 있다면, Sequence는 자동으로 대기하게 됩니다.

랜덤화(Randomization) 과정

```
t.randomize();
```

이 한 줄이 실행될 때 내부적으로 일어나는 일들:

1. 제약 조건 확인: Transaction 클래스에 정의된 모든 constraint 검사
2. 솔버 실행: 제약 조건을 만족하는 랜덤값 생성
3. 필드 할당: 생성된 값을 rand 필드들에 할당

```
// randomize() 실행 전  
t.a = ?; // 정의되지 않음  
t.b = ?; // 정의되지 않음
```

```
// randomize() 실행 후  
t.a = 7; // 0~15 사이의 랜덤값  
t.b = 3; // 0~15 사이의 랜덤값
```

디버깅과 로깅

```
`uvm_info("GEN",$sformatf("Data send to Driver a :%0d ,  
b :%0d",t.a,t.b), UVM_NONE);
```

UVM 로깅 시스템의 구조:

매개변수	설명	예시
ID	메시지 출처 식별	"GEN"
Message	실제 메시지 내용	"Data send to Driver..."
Verbosity	메시지 중요도	UVM_NONE, UVM_LOW, UVM_MEDIUM, UVM_HIGH

verbosity 레벨 이해:

- UVM_NONE (0): 항상 출력되는 중요한 메시지
- UVM_LOW (100): 기본적인 정보 메시지
- UVM_MEDIUM (200): 상세한 디버깅 정보
- UVM_HIGH (300): 매우 상세한 내부 정보
- UVM_DEBUG (400): 개발자용 디버깅 정보

시뮬레이션 실행 시 verbosity 레벨을 조정하여 원하는 수준의 정보만 볼 수 있습니다.

실무에서의 Sequence 설계 패턴

1. 기본 랜덤 테스트:

```
// 현재 예시와 동일
repeat(1000) begin
    start_item(t);
    t.randomize();
    finish_item(t);
end
```

2. 제약 조건 테스트:

```
repeat(100) begin
    start_item(t);
    t.randomize() with {
        a inside {[0:3]}; // 특정 범위 테스트
        b inside {[12:15]};
    };
    finish_item(t);
end
```

3. 경계값 테스트:

```
int boundary_values[] = {0, 1, 14, 15};
foreach(boundary_values[i]) begin
    foreach(boundary_values[j]) begin
        start_item(t);
        t.a = boundary_values[i];
        t.b = boundary_values[j];
        finish_item(t);
    end
end
```

Sequence 사용 시 주의사항과 실수들

자주하는 실수 1: randomize() 호출 누락

```
// 잘못된 예시  
start_item(t);  
// t.randomize(); <- 이게 빠지면 이전 값이 그대로 사용됨  
finish_item(t);
```

자주하는 실수 2: start_item()과 finish_item() 불일치

```
// 잘못된 예시  
start_item(t);  
if(some_condition) return; // finish_item() 호출 없이 종료  
finish_item(t);
```

자주하는 실수 3: Transaction 객체 재사용 실수

```
// 문제가 될 수 있는 예시  
transaction t1, t2;  
t1 = transaction::type_id::create("t1");  
t2 = t1; // 같은 객체를 가리킴
```

```
start_item(t1);  
t1.randomize();  
finish_item(t1);  
  
start_item(t2); // 사실상 같은 객체  
t2.randomize(); // t1 값도 함께 변경됨  
finish_item(t2);
```

올바른 예시:

```
transaction t;  
t = transaction::type_id::create("t");  
  
repeat(10) begin  
    start_item(t);  
    t.randomize(); // 매번 새로운 값 생성
```

```
    finish_item(t);
end
```

Sequence Class는 UVM 검증 환경의 시작점입니다. 여기서 생성된 Transaction들이 전체 검증 플로우를 시작하게 됩니다. 다음 장에서는 이렇게 생성된 Transaction들이 어떻게 실제 하드웨어에 전달되는지 Driver Class를 통해 알아보겠습니다.

5. UVM Driver Class - 소프트웨어와 하드웨어 간의 브릿지

Driver Class의 본질적 역할

Driver Class는 UVM 환경에서 가장 중요한 컴포넌트 중 하나입니다. 소프트웨어 영역의 Transaction 객체를 실제 하드웨어가 이해할 수 있는 신호로 변환하는 역할을 담당합니다.

이를 이해하기 위해 소프트웨어와 하드웨어 간의 근본적 차이를 다시 살펴보겠습니다:

영역	데이터 표현	시간 개념	접근 방식
소프트웨어	객체, 구조체	논리적 시간	메서드 호출
하드웨어	신호선의 전기적 값	물리적 시간 (클럭)	신호 할당

Driver는 이 두 영역 사이의 번역기 역할을 합니다.

Driver Class의 구조 분석

```
// Driver Class - 소프트웨어 Transaction을 하드웨어 신호로 변환
class driver extends uvm_driver #(transaction);
`uvm_component_utils(driver)

function new(input string path = "driver", uvm_component parent
= null);
super.new(path, parent); // 부모 클래스 생성자 호출
endfunction
```

```
transaction tc;          // 받아온 Transaction을 저장할 변수  
virtual add_if aif;    // DUT와 연결된 Interface 핸들
```

클래스 선언 분석:

1. 상속 구조:

```
class driver extends uvm_driver #(transaction);
```

- `uvm_driver`는 UVM의 표준 Driver 기능을 제공하는 베이스 클래스
- `#(transaction)`: 이 Driver가 처리할 Transaction 타입을 명시
- 템플릿을 통해 타입 안전성 보장

2. UVM Component 등록:

```
`uvm_component_utils(driver)
```

- Driver를 UVM 컴포넌트로 등록
- UVM의 phase system, factory pattern 등을 사용할 수 있게 됨

3. 생성자:

```
function new(input string path = "driver", uvm_component parent = null);
```

- `path`: UVM 계층 구조에서의 이름
- `parent`: 부모 컴포넌트에 대한 참조 (Agent 가 됨)

Driver의 핵심 멤버 변수들

```
transaction tc;          // 현재 처리 중인 Transaction  
virtual add_if aif;    // DUT Interface에 대한 핸들
```

중요한 개념 - **virtual interface**:

virtual 키워드가 붙은 이유를 이해해야 합니다:

일반 Interface	Virtual Interface
정적으로 연결됨	동적으로 연결 가능
컴파일 시점에 연결 결정	런타임에 연결 가능
클래스에서 직접 사용 불가	클래스에서 사용 가능

```
// 컴파일 에러 발생  
class driver;  
    add_if aif; // 일반 interface는 클래스 멤버로 불가능  
endclass
```

```
// 올바른 사용  
class driver;  
    virtual add_if aif; // virtual interface는 가능  
endclass
```

Build Phase - 설정과 초기화

```
virtual function void build_phase(uvm_phase phase);  
    super.build_phase(phase); // 부모 클래스 build_phase  
호출  
    tc = transaction::type_id::create("tc"); // Transaction 객체  
생성  
  
    if(!uvm_config_db #(virtual add_if)::get(this,"","aif",aif))  
        `uvm_error("DRV","Unable to access uvm_config_db");  
endfunction
```

Build Phase의 목적과 타이밍:

UVM은 여러 단계(phase)로 나누어 컴포넌트들을 관리합니다:

UVM Phase 순서:

build → connect → end_of_elaboration → start_of_simulation → run → extract → check → report

Build Phase에서 하는 일들:

1. 필요한 객체들 생성
2. Configuration 데이터베이스에서 설정 정보 가져오기
3. 초기화 작업 수행

Configuration Database 사용:

```
if(!uvm_config_db #(virtual add_if)::get(this,"","aif",aif))
```

이 코드를 분해해보면:

부분	설명	값
uvm_config_db	UVM의 전역 데이터베이스	-
#(virtual add_if)	저장된 데이터 타입	Interface 타입
this	요청하는 컴포넌트	현재 Driver
""	인스턴스 경로 (빈 문자열은 와일드카드)	모든 경로
"aif"	데이터 키 이름	"aif"
aif	저장할 변수	Driver의 aif 멤버

Configuration Database의 동작 방식:

Testbench Top에서 설정:

```
uvm_config_db #(virtual add_if)::set(null, "uvm_test_top.e.a*",  
"aif", aif);
```



Agent와 그 하위 모든 컴포넌트

Driver에서 읽기:

```
uvm_config_db #(virtual add_if)::get(this, "", "aif", aif);
```

이 메커니즘을 통해 하드웨어 Interface가 소프트웨어 컴포넌트로 전달됩니다.

Run Phase - 실제 구동 로직

```
virtual task run_phase(uvm_phase phase);  
  forever begin  
    seq_item_port.get_next_item(tc); // Sequencer로부터 다음  
    Transaction 받기
```

```

aif.a <= tc.a;           // Transaction 데이터를 Interface에
할당
aif.b <= tc.b;
`uvm_info("DRV", $sformatf("Trigger DUT a: %0d ,b : %0d",tc.a,
tc.b), UVM_NONE);
seq_item_port.item_done(); // Sequencer에게 처리 완료 알림
#10;                      // 10 시간 단위 대기
end
endtask

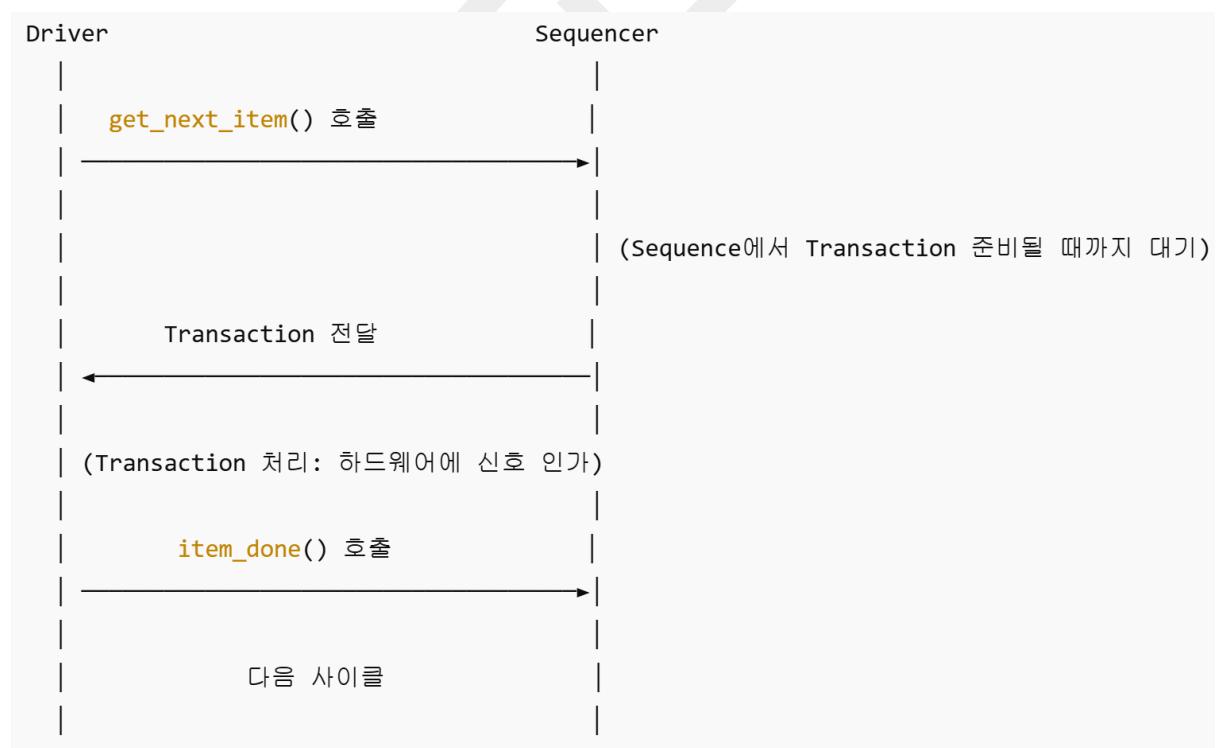
```

Run Phase의 특징:

- task로 선언되어 시간을 소모할 수 있음
- forever 루프로 계속 실행됨
- 시뮬레이션이 끝날 때까지 계속 동작

Sequence Item Port 통신:

Driver와 Sequencer 간의 통신은 다음과 같은 핸드셰이크 프로토콜로 이루어집니다:



하드웨어 신호 할당:

```
aif.a <= tc.a;  
aif.b <= tc.b;
```

여기서 `<=` 연산자를 사용하는 이유:

연산자	특징	사용 시점
<code>=</code> (blocking)	즉시 할당	조합회로, 테스트벤치
<code><=</code> (non-blocking)	클럭 에지에서 할당	순차회로

중요한 오해 방지: 많은 학생들이 "이 코드는 조합회로인데 왜 `<=`를 쓰나?"라고 궁금해합니다. 여기서 `<=`를 사용하는 이유는 DUT 가 조합회로여서가 아니라, 테스트벤치의 신호 할당을 예측 가능하게 하기 위해서입니다.

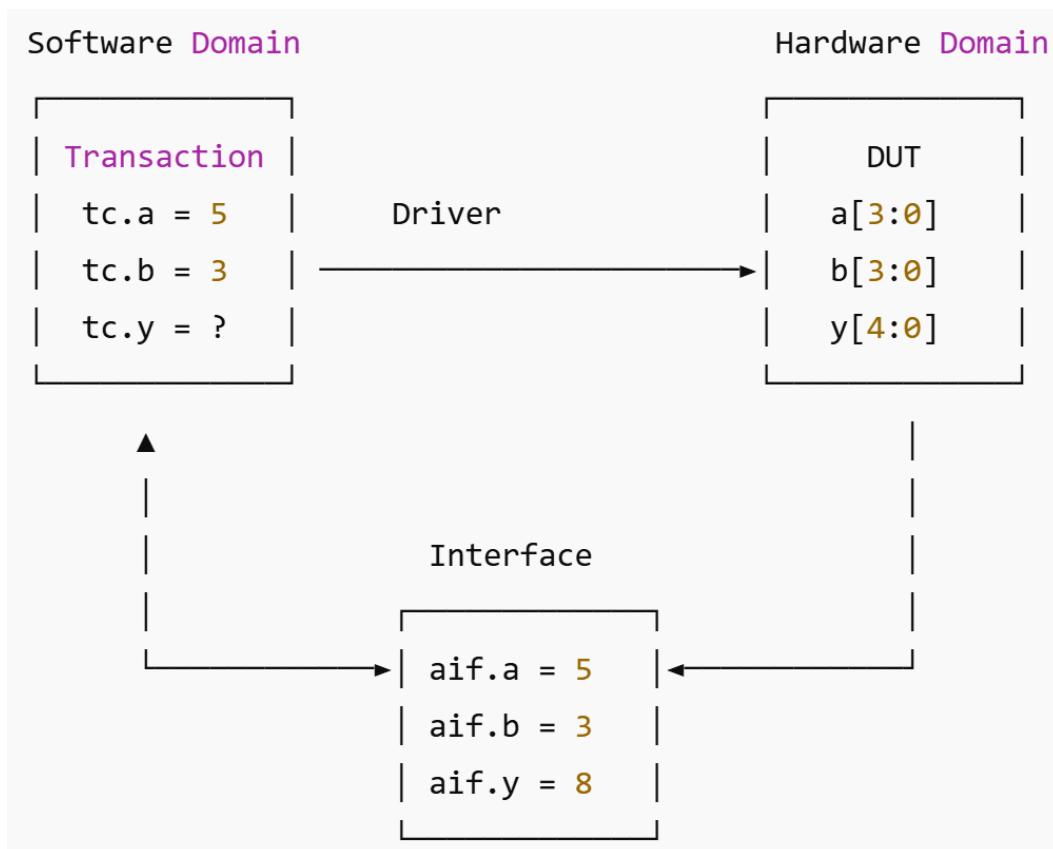
타이밍 제어:

```
#10;
```

이 딜레이의 목적:

1. DUT에게 신호 처리 시간 제공
2. Monitor가 결과를 읽을 시간 제공
3. 다음 테스트 케이스와의 구분

Driver 의 역할을 그림으로 이해하기



이 다이어그램에서 볼 수 있듯이:

1. Driver는 Transaction의 데이터를 Interface로 복사
2. Interface는 DUT의 입력 포트와 직접 연결
3. DUT의 출력은 Interface를 통해 Monitor에서 관찰 가능

실무에서의 Driver 설계 고려사항

1. 클럭 동기화: 실제 프로젝트에서는 클럭에 맞춘 정확한 타이밍이 중요합니다:

```
// 클럭 기반 Driver 예시
virtual task run_phase(uvm_phase phase);
forever begin
    seq_item_port.get_next_item(tc);

    @(posedge aif.clk); // 클럭 상승 에지까지 대기
    aif.a <= tc.a;
```

```

aif.b <= tc.b;

seq_item_port.item_done();
end
endtask

```

2. 프로토콜 준수: 복잡한 인터페이스는 특정 프로토콜을 따라야 합니다:

```

// 핸드세이크 프로토콜 예시
virtual task send_transaction(transaction tc);
    aif.valid <= 1'b1;      // 유효 신호 설정
    aif.data <= tc.data;    // 데이터 설정

    wait(aif.ready);        // 준비 신호까지 대기
    @(posedge clk);        // 클럭 에지

    aif.valid <= 1'b0;      // 유효 신호 해제
endtask

```

3. 에러 처리:

```

virtual task run_phase(uvm_phase phase);
forever begin
    seq_item_port.get_next_item(tc);

    if(tc == null) begin
        `uvm_error("DRV", "Received null transaction");
        continue;
    end

    // 정상 처리
    aif.a <= tc.a;
    aif.b <= tc.b;

    seq_item_port.item_done();
    #10;
end
endtask

```

Driver 사용 시 주의사항

자주하는 실수 1: Interface 연결 확인 누락

```
// 문제가 있는 코드  
virtual task run_phase(uvm_phase phase);  
forever begin  
    // aif가 null인지 확인하지 않음  
    aif.a <= tc.a; // Runtime 에러 가능성  
end  
endtask  
  
// 안전한 코드  
virtual function void build_phase(uvm_phase phase);  
    // Interface 연결 확인  
    if(!uvm_config_db #(virtual add_if)::get(this,"","aif",aif))  
        `uvm_fatal("DRV","Unable to access interface");  
endfunction
```

자주하는 실수 2: item_done() 호출 누락

```
// 잘못된 예시 - Sequencer 가 무한 대기  
seq_item_port.get_next_item(tc);  
aif.a <= tc.a;  
aif.b <= tc.b;  
// seq_item_port.item_done(); <- 이걸 빼먹으면 시뮬레이션 멈춤
```

자주하는 실수 3: 타이밍 고려 부족

```
// 문제 있는 코드 - 신호 변경이 너무 빨라서 DUT 가 처리 못함  
forever begin  
    seq_item_port.get_next_item(tc);  
    aif.a <= tc.a;  
    aif.b <= tc.b;  
    seq_item_port.item_done();  
    // #10; <- 딜레이 없으면 DUT 응답 시간 부족  
end
```

Driver Class는 UVM 환경과 실제 하드웨어를 연결하는 핵심 다리 역할을 합니다. 여기서 정확한 타이밍과 프로토콜 준수가 전체 검증의 성공을 좌우합니다. 다음 장에서는 Driver가 하드웨어에 전달한 신호들과 그 결과를 어떻게 관찰하는지 Monitor Class를 통해 알아보겠습니다.

6. UVM Monitor Class - 하드웨어 신호 관찰자

Monitor Class의 본질적 역할과 필요성

Monitor Class는 UVM 검증 환경에서 하드웨어의 동작을 관찰하고 기록하는 컴포넌트입니다. Driver가 소프트웨어에서 하드웨어로 데이터를 보내는 역할이라면, Monitor는 그 반대 방향으로 하드웨어에서 소프트웨어로 데이터를 가져오는 역할을 합니다.

Monitor vs Driver 비교:

측면	Driver	Monitor
데이터 흐름	Software → Hardware	Hardware → Software
주요 기능	신호 생성 및 제어	신호 관찰 및 수집
DUT 관계	DUT에 입력 제공	DUT 출력 관찰
능동성	능동적 (신호를 변경함)	수동적 (신호를 읽기만 함)
타이밍	정확한 타이밍으로 신호 인가	지속적인 관찰

Monitor 가 필요한 이유

하드웨어 검증에서 Monitor가 없다면 어떻게 될까요?

```
// Monitor 없는 검증 환경의 문제점
module simple_testbench;
    // Driver가 입력 생성
    always begin
        a = 5; b = 3; #10;
        a = 7; b = 2; #10;
        // 결과를 어떻게 확인하지? y 값을 누가 읽지?
    end
endmodule
```

이런 상황에서는:

1. DUT의 출력값을 자동으로 수집할 방법이 없음
2. 여러 테스트 케이스의 결과를 체계적으로 관리할 수 없음
3. 입력과 출력을 연관지어 분석하기 어려움

Monitor는 이러한 문제들을 해결합니다.

제공된 Monitor Class 구조 분석

```
// Monitor Class - 하드웨어 신호 관찰과 수집
class monitor extends uvm_monitor;
`uvm_component_utils(monitor)

uvm_analysis_port #(transaction) send; // Scoreboard로 데이터 전송용
포트

function new(input string path = "monitor", uvm_component parent =
null);
    super.new(path, parent);
    send = new("send", this);           // Analysis Port 생성
endfunction

transaction t;          // 관찰된 데이터를 저장할 Transaction
virtual add_if aif;    // DUT Interface 핸들
```

클래스 선언 분석:

1. 상속 구조:

```
class monitor extends uvm_monitor;
```

- uvm_monitor: UVM의 표준 Monitor 기능 제공
- Driver와 달리 템플릿 매개변수가 없음 (Transaction을 받지 않고 생성하기 때문)

2. Analysis Port:

```
uvm_analysis_port #(transaction) send;
```

- Monitor에서 Scoreboard로 데이터를 전송하는 통로
- #(transaction): 전송할 데이터 타입 지정
- 일대다 통신 지원 (여러 Scoreboard에 동시 전송 가능)

Analysis Port vs Sequence Item Port 비교:

특징	Sequence Item Port	Analysis Port
통신 방향	양방향 (handshake)	단방향 (broadcast)
연결 관계	1:1 (Driver ↔ Sequencer)	1:N (Monitor → 여러 Subscriber)
동기화	동기식 (대기 발생)	비동기식 (즉시 전송)
용도	제어 흐름	데이터 배포

3. 생성자에서 Analysis Port 초기화:

```
function new(input string path = "monitor", uvm_component parent = null);
    super.new(path, parent);
    send = new("send", this); // 이 Monitor에 속한 Analysis Port 생성
endfunction
```

Build Phase - Monitor 초기화

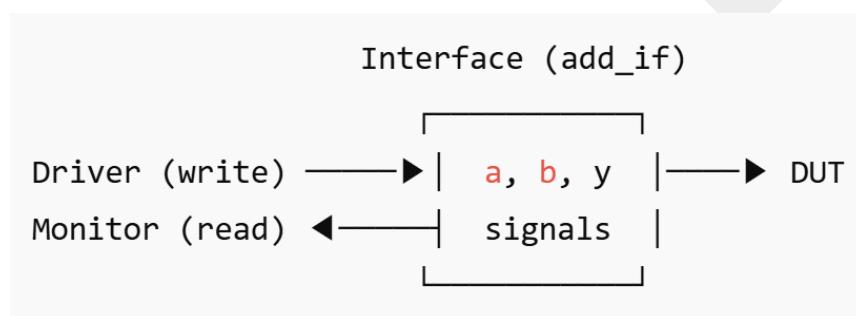
```
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    t = transaction::type_id::create("t"); // Transaction 객체 생성

    if(!uvm_config_db #(virtual add_if)::get(this,"","aif",aif))
        `uvm_error("MON","Unable to access uvm_config_db");
Endfunction
```

Build Phase 의 Monitor 특이점:

1. **Transaction** 생성: Driver 와 달리 Monitor 는 자신이 Transaction 을 생성함
2. **Interface** 연결: Driver 와 동일한 Interface 를 공유하지만 읽기 전용으로 사용
3. 에러 처리: Interface 연결 실패 시 적절한 에러 메시지 출력

중요한 개념 - 공유 Interface:



Driver 와 Monitor 가 같은 Interface 를 사용하지만:

- Driver 는 입력 신호 (a, b)에 쓰기
- Monitor 는 모든 신호 (a, b, y)에서 읽기
- 충돌이 발생하지 않는 이유: 서로 다른 신호에 접근

Run Phase - 실제 모니터링 로직

```
virtual task run_phase(uvm_phase phase);
forever begin
    #10;                      // 신호 안정화를 위한 대기
    t.a = aif.a;              // Interface에서 입력값 읽기
    t.b = aif.b;
    t.y = aif.y;              // Interface에서 출력값 읽기
    `uvm_info("MON", $sformatf("Data send to Scoreboard a : %0d , 
    b : %0d and y : %0d", t.a,t.b,t.y), UVM_NONE);
    send.write(t);            // Analysis Port를 통해 Scoreboard로 전송
end
endtask
```

Run Phase 동작 분석:

1. 타이밍 동기화:

```
#10;
```

- Driver 가 신호를 바꾸고 DUT 가 처리할 시간을 기다림
- DUT (조합회로)의 전파 지연 고려
- 안정된 신호를 읽기 위한 여유 시간

시간 순서도:

Time: 0 5 10 15 20

Driver: [신호 변경]

DUT: [처리] [출력 안정]

Monitor: [신호 읽기]

2. 신호 읽기:

```
t.a = aif.a; // 입력값 캡처  
t.b = aif.b; // 입력값 캡처  
t.y = aif.y; // 출력값 캡처
```

중요한 포인트: Monitor 는 입력과 출력을 모두 읽습니다. 이는 나중에 Scoreboard 에서 "입력 (a,b)에 대해 출력 (y)가 올바른가?"를 검증하기 위해서입니다.

3. 데이터 전송:

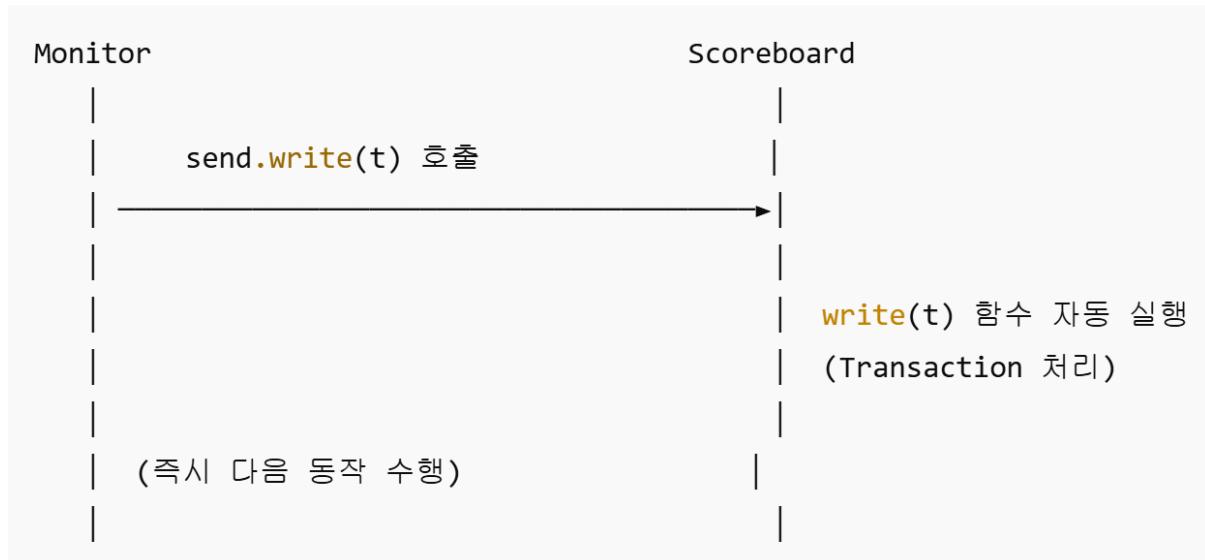
```
send.write(t);
```

이 호출이 실행되면:

1. 연결된 모든 Scoreboard 의 write() 함수가 자동 호출됨
2. Transaction 객체가 복사되어 전달됨
3. Monitor 는 전송 완료를 기다리지 않고 계속 진행

Analysis Port 통신 매커니즘

Analysis Port는 UVM의 TLM (Transaction Level Modeling) 통신을 사용합니다:



Analysis Port의 장점:

1. **Non-blocking:** Monitor가 Scoreboard 처리를 기다리지 않음
2. **Broadcasting:** 여러 Scoreboard에 동시에 전송 가능
3. **Type-safe:** 컴파일 시점에 타입 검사

Monitor 사용 패턴과 변형

1. 클럭 기반 모니터링:

```
// 클럭 동기식 Monitor (순차회로용)
virtual task run_phase(uvm_phase phase);
forever begin
    @(posedge clk);      // 클럭 에지에서 샘플링
    t.a = aif.a;
    t.b = aif.b;
    t.y = aif.y;
    send.write(t);
end
endtask
```

2. 조건부 모니터링:

```
// 특정 조건에서만 모니터링
virtual task run_phase(uvm_phase phase);
forever begin
    @(aif.valid);      // valid 신호가 활성화될 때만
    if(aif.valid) begin
        t.a = aif.a;
        t.b = aif.b;
        t.y = aif.y;
        send.write(t);
    end
end
endtask
```

3. 다중 트랜잭션 모니터링:

```
// 파이프라인 처리를 위한 다중 Transaction
virtual task run_phase(uvm_phase phase);
forever begin
    transaction t_new = transaction::type_id::create("t_new");
    #10;
    t_new.a = aif.a;
    t_new.b = aif.b;
    t_new.y = aif.y;
    send.write(t_new); // 매번 새로운 객체 사용
end
endtask
```

Monitor 설계 시 주의사항

자주하는 실수 1: Transaction 객체 재사용 문제

```
// 문제가 있는 코드
transaction t;
forever begin
    // 같은 객체를 계속 재사용
    t.a = aif.a; t.b = aif.b; t.y = aif.y;
    send.write(t); // 이전 Transaction이 덮어써짐
end
```

```
// 올바른 방법 1: 매번 새 객체 생성
forever begin
    transaction t_new = transaction::type_id::create();
    t_new.a = aif.a; t_new.b = aif.b; t_new.y = aif.y;
    send.write(t_new);
end
```

```
// 올바른 방법 2: 객체 복사
forever begin
    t.a = aif.a; t.b = aif.b; t.y = aif.y;
    send.write(t.clone()); // 복사본 전송
end
```

자주하는 실수 2: 타이밍 고려 부족

```
// 문제: 신호 변화 직후 바로 읽기
forever begin
    @(aif.a);           // a 신호가 변하자마자
    t.y = aif.y;        // y를 읽는데, 아직 DUT가 처리 안 했을 수 있음
    send.write(t);
end
```

```
// 해결: 적절한 지연 후 읽기
forever begin
    @(aif.a);
    #1;                // DUT 처리 시간 확보
    t.y = aif.y;
    send.write(t);
end
```

자주하는 실수 3: Analysis Port 초기화 누락

```
// 생성자에서 Analysis Port 생성을 잊어버림
function new(input string path = "monitor", uvm_component parent =
null);
    super.new(path, parent);
    // send = new("send", this); <- 이게 없으면 runtime 에러
Endfunction
```

실무에서의 Monitor 활용 패턴

1. Coverage 수집을 위한 Monitor:

```
class monitor extends uvm_monitor;
    covergroup cg_inputs @($send.write);
        a_cp: coverpoint t.a {
            bins low = {[0:7]};
            bins high = {[8:15]};
        }
        b_cp: coverpoint t.b {
            bins low = {[0:7]};
            bins high = {[8:15]};
        }
        cross a_cp, b_cp;
    endgroup

    function new(...);
        cg_inputs = new();
    endfunction
endclass
```

2. 프로토콜 검증을 위한 Monitor:

```
virtual task run_phase(uvm_phase phase);
    forever begin
        @(posedge aif.valid);
        if(!aif.ready) begin
            `uvm_error("MON", "Protocol violation: valid without ready");
        end
        // 정상적인 데이터 수집
        collect_transaction();
    end
endtask
```

3. 성능 분석을 위한 Monitor:

```
class monitor extends uvm_monitor;
    int latency_counter;
    real start_time, end_time;
```

```
virtual task run_phase(uvm_phase phase);
  forever begin
    @(posedge aif.start);
    start_time = $realtime;

    @(posedge aif.done);
    end_time = $realtime;

    latency_counter = end_time - start_time;
    `uvm_info("PERF", $sformatf("Latency: %0d ns",
latency_counter), UVM_LOW);
  end
endtask
endclass
```

Monitor Class는 하드웨어의 동작을 소프트웨어 영역으로 가져오는 핵심 컴포넌트입니다. 정확한 타이밍과 적절한 신호 샘플링을 통해 검증의 기초 데이터를 수집합니다. 다음 장에서는 Monitor가 수집한 데이터를 바탕으로 실제 검증을 수행하는 Scoreboard Class를 알아보겠습니다.

7. UVM Scoreboard Class - 결과 검증과 판정

Scoreboard Class의 핵심 역할

Scoreboard Class는 UVM 검증 환경에서 실제 검증 로직이 구현되는 핵심 컴포넌트입니다. 게임으로 비유하면, 플레이어가 올바른 답을 맞혔는지 판단하는 채점관 역할을 합니다.

Scoreboard의 본질적 기능:

1. 예상 결과 계산: 입력에 대한 정확한 출력값 계산
2. 실제 결과 비교: DUT 출력과 예상 출력 비교
3. Pass/Fail 판정: 검증 성공/실패 결정
4. 통계 수집: 테스트 결과 집계 및 분석

하드웨어 검증에서 Scoreboard의 필요성

소프트웨어 테스트와 하드웨어 검증의 차이점을 보면 Scoreboard의 중요성을 알 수 있습니다:

측면	소프트웨어 테스트	하드웨어 검증
검증 대상	함수의 리턴값	물리적 신호값
예상값 계산	동일한 함수 재실행	별도의 참조 모델 필요
비교 시점	함수 호출 즉시	신호 안정화 후
오류 원인	로직 에러	타이밍, 로직, 물리적 문제 등

```
// 소프트웨어 테스트 (간단함)
function test_add();
    result = add_function(5, 3);
    assert(result == 8); // 즉시 비교 가능
endfunction
```

```
// 하드웨어 검증 (복잡함)
// 1. 입력 신호 인가 (Driver)
// 2. 하드웨어 처리 대기
// 3. 출력 신호 수집 (Monitor)
// 4. 예상값 계산 (Scoreboard)
// 5. 비교 및 판정 (Scoreboard)
```

제공된 Scoreboard Class 분석

```
// Scoreboard Class - 검증 결과 판정기
class scoreboard extends uvm_scoreboard;
`uvm_component_utils(scoreboard)

uvm_analysis_imp #(transaction,scoreboard) recv; // Monitor로부터
데이터 수신용

transaction tr; // 받은 Transaction을 저장할 변수

function new(input string path = "scoreboard", uvm_component
parent = null);
    super.new(path, parent);
    recv = new("recv", this); // Analysis Implementation Port 생성
endfunction
```

클래스 구조 분석:

1. 상속 구조:

```
class scoreboard extends uvm_scoreboard;
```

- uvm_scoreboard: UVM의 표준 Scoreboard 기능 제공
- 로그 메시지, 통계 수집 등의 기본 기능 자동 제공

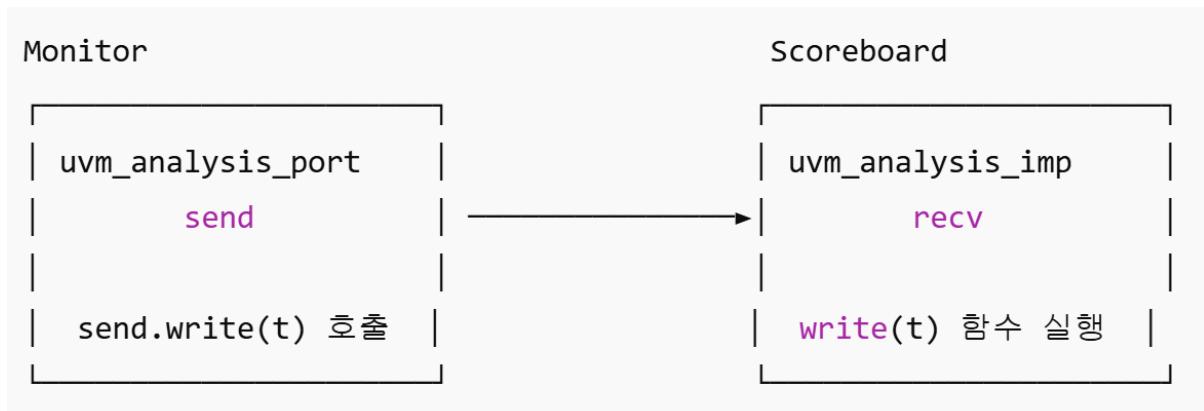
2. Analysis Implementation Port:

```
uvm_analysis_imp #(transaction,scoreboard) recv;
```

Analysis Port vs Analysis Implementation Port:

구분	Analysis Port	Analysis Implementation Port
사용 위치	Monitor (송신자)	Scoreboard (수신자)
기능	write() 메서드 호출	write() 메서드 구현
연결 방향	출력 포트	입력 포트
구현 필요사항	없음	write() 함수 반드시 구현

연결 관계 도식:



3. Analysis Implementation Port 생성:

```
recv = new("recv", this);
```

- "recv": 포트 이름
- this: 이 Scoreboard가 `write()` 함수를 구현함을 명시

Build Phase 분석

```
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    tr = transaction::type_id::create("tr"); // Transaction 저장용
    객체 생성
endfunction
```

Build Phase 의 Scoreboard 특징:

- Monitor, Driver 와 달리 Interface 연결이 불필요
- Transaction 객체만 생성하면 됨
- 설정(Configuration) 정보가 상대적으로 적음

핵심 기능 - `write()` 함수 구현

```
virtual function void write(input transaction t);
    tr = t; // 받은 Transaction 저장
    `uvm_info("SCO",$sformatf("Data rcvd from Monitor a: %0d , b : %0d
    and y : %0d",tr.a,tr.b,tr.y), UVM_NONE);
```

```

if(tr.y == tr.a + tr.b)           // 예상값과 실제값 비교
`uvm_info("SCO","Test Passed", UVM_NONE)
else
`uvm_info("SCO","Test Failed", UVM_NONE);
endfunction

```

write() 함수 동작 분석:

1. 함수 시그니처:

```
virtual function void write(input transaction t);
```

- **virtual**: 서브클래스에서 오버라이드 가능
- **function**: 시간 소모가 없는 즉시 실행
- **input transaction t**: Monitor로부터 받은 Transaction

중요한 개념 - 왜 **task**가 아닌 **function**인가?

함수 타입	시간 소모	용도
function	없음 (0 time)	즉시 계산, 비교
task	있을 수 있음	시간이 걸리는 작업

Scoreboard의 검증 로직은 즉시 실행되어야 하므로 **function**을 사용합니다.

2. **Transaction** 저장:

```
tr = t;
```

중요한 오해 방지: 이 코드는 객체 복사가 아닌 참조 복사입니다. 실무에서는 다음과 같이 개선할 수 있습니다:

```
// 현재 코드 (참조 복사)
```

```
tr = t;
```

```
// 개선된 코드 (객체 복사)
```

```
tr.copy(t);
```

```
// 또는
```

```
$cast(tr, t.clone());
```

3. 예상값 계산과 비교:

```
if(tr.y == tr.a + tr.b)
```

이 부분이 Scoreboard의 핵심입니다. 여기서는 간단한 덧셈이지만, 실무에서는 복잡한 참조 모델이 필요할 수 있습니다.

다양한 검증 시나리오

1. 기본 기능 검증:

```
// 현재 예시: 정확한 덧셈 결과 확인
if(tr.y == tr.a + tr.b)
    `uvm_info("SCO", "Addition Test Passed", UVM_NONE)
else
    `uvm_error("SCO", $sformatf("Addition Failed: %0d + %0d = %0d,
Expected: %0d",
                                tr.a, tr.b, tr.y, tr.a + tr.b));
```

2. 경계값 검증:

```
virtual function void write(input transaction t);
    tr = t;
    int expected = tr.a + tr.b;

    // 오버플로우 체크
    if(expected > 31) begin
        `uvm_warning("SCO", $sformatf("Overflow condition: %0d + %0d
= %0d > 31",
                                tr.a, tr.b, expected));
        expected = expected % 32; // 5비트 결과로 잘림
    end

    if(tr.y == expected)
        `uvm_info("SCO", "Test Passed", UVM_NONE)
    else
        `uvm_error("SCO", "Test Failed", UVM_NONE);
endfunction
```

3. 통계 수집:

```
class scoreboard extends uvm_scoreboard;
    int pass_count = 0;
    int fail_count = 0;
    int total_count = 0;

    virtual function void write(input transaction t);
        tr = t;
        total_count++;

        if(tr.y == tr.a + tr.b) begin
            pass_count++;
            `uvm_info("SCO", "Test Passed", UVM_NONE)
        end else begin
            fail_count++;
            `uvm_error("SCO", "Test Failed", UVM_NONE);
        end

        // 통계 출력
        if(total_count % 100 == 0) begin
            `uvm_info("STAT", $sformatf("Tests: %0d, Pass: %0d,
Fail: %0d",
                                         total_count, pass_count,
fail_count), UVM_LOW);
        end
    endfunction

    virtual function void final_phase(uvm_phase phase);
        `uvm_info("FINAL", $sformatf("Final Results - Total: %0d,
Pass: %0d, Fail: %0d, Pass Rate: %.2f%%",
                                     total_count, pass_count, fail_count,
100.0 * pass_count / total_count),
UVM_NONE);
    endfunction
endclass
```

고급 Scoreboard 설계 패턴

1. 참조 모델(Reference Model) 사용:

```
class scoreboard extends uvm_scoreboard;
    // 참조 모델 - 소프트웨어로 구현된 정확한 알고리즘
    function int reference_model(int a, int b);
        return a + b; // 실제로는 더 복잡한 로직
    endfunction

    virtual function void write(input transaction t);
        int expected = reference_model(t.a, t.b);

        if(t.y == expected)
            `uvm_info("SCO", "Test Passed", UVM_NONE)
        else
            `uvm_error("SCO", $sformatf("Mismatch: Got %0d,
        Expected %0d", t.y, expected), UVM_NONE);
    endfunction
endclass
```

2. 다중 체크 포인트:

```
virtual function void write(input transaction t);
    tr = t;

    // 체크 1: 기본 기능
    check_basic_function();

    // 체크 2: 타이밍 (필요시)
    check_timing_constraints();

    // 체크 3: 전력 소모 (필요시)
    check_power_constraints();

    // 체크 4: 특수 케이스
    check_corner_cases();
endfunction

function void check_basic_function();
```

```

if(tr.y != tr.a + tr.b) begin
    `uvm_error("FUNC", $sformatf("Basic function failed: %0d
+ %0d != %0d",
                                tr.a, tr.b, tr.y));
end
endfunction

function void check_corner_cases();
// 최대값 테스트
if(tr.a == 15 && tr.b == 15) begin
    if(tr.y != 30) begin
        `uvm_error("CORNER", "Max value test failed");
    end else begin
        `uvm_info("CORNER", "Max value test passed", UVM_LOW);
    end
end

// 최소값 테스트
if(tr.a == 0 && tr.b == 0) begin
    if(tr.y != 0) begin
        `uvm_error("CORNER", "Min value test failed");
    end else begin
        `uvm_info("CORNER", "Min value test passed", UVM_LOW);
    end
end
endfunction

```

Scoreboard 사용 시 주의사항

자주하는 실수 1: write() 함수 구현 누락

```

// 컴파일 에러 발생
class scoreboard extends uvm_scoreboard;
    uvm_analysis_imp #(transaction,scoreboard) recv;

    // write() 함수를 구현하지 않으면 에러 발생
    // virtual function void write(input transaction t); <- 필수!
Endclass

```

자주하는 실수 2: Transaction 수명 관리 실수

```
// 문제가 있는 코드  
transaction saved_tr;  
  
virtual function void write(input transaction t);  
    saved_tr = t; // 참조만 저장  
    // 나중에 saved_tr 을 사용할 때 t 가 변경되면 문제 발생  
endfunction  
  
// 올바른 코드  
virtual function void write(input transaction t);  
    saved_tr.copy(t); // 실제 데이터 복사  
endfunction
```

자주하는 실수 3: 부정확한 예상값 계산

```
// 문제: 하드웨어의 특성을 고려하지 않음  
virtual function void write(input transaction t);  
    // 실제 하드웨어는 5비트 출력인데 int로 계산  
    int expected = t.a + t.b; // 32비트로 계산됨  
  
    if(t.y == expected) begin // 5비트 vs 32비트 비교  
        // 잘못된 비교가 될 수 있음  
    end  
endfunction  
  
// 올바른 방법: 하드웨어와 같은 비트폭으로 계산  
virtual function void write(input transaction t);  
    bit [4:0] expected = t.a + t.b; // 5비트로 제한  
  
    if(t.y == expected) begin  
        `uvm_info("SCO", "Test Passed", UVM_NONE)  
    end  
endfunction
```

실무에서의 Scoreboard 활용

1. 복잡한 DUT를 위한 Scoreboard: 실제 프로젝트에서는 단순한 덧셈기보다 훨씬 복잡한 DUT를 다룹니다:

```
// 예: 복잡한 ALU Scoreboard
class alu_scoreboard extends uvm_scoreboard;
    virtual function void write(input alu_transaction t);
        bit [31:0] expected_result;
        bit expected_carry, expected_zero;

        case(t.opcode)
            ADD: begin
                {expected_carry, expected_result} = t.a + t.b;
                expected_zero = (expected_result == 0);
            end
            SUB: begin
                expected_result = t.a - t.b;
                expected_carry = (t.a >= t.b);
                expected_zero = (expected_result == 0);
            end
            AND: begin
                expected_result = t.a & t.b;
                expected_carry = 0;
                expected_zero = (expected_result == 0);
            end
            // ... 더 많은 연산들
        endcase

        // 결과 검증
        check_result(t.result, expected_result, "ALU Result");
        check_flags(t.carry_flag, expected_carry, "Carry Flag");
        check_flags(t.zero_flag, expected_zero, "Zero Flag");
    endfunction

    function void check_result(bit [31:0] actual, bit [31:0]
expected, string name);
        if(actual == expected)
            `uvm_info("PASS", $sformatf("%s matched: %h", name,
actual), UVM_LOW)
    endfunction
```

```

    else
        `uvm_error("FAIL", $sformatf("%s mismatch: got %h,
expected %h", name, actual, expected));
    endfunction
endclass

```

2. 성능 및 Coverage 통합:

```

class advanced_scoreboard extends uvm_scoreboard;
    // Coverage Groups
    covergroup operation_cg;
        a_range: coverpoint tr.a {
            bins low = {[0:7]};
            bins high = {[8:15]};
        }
        b_range: coverpoint tr.b {
            bins low = {[0:7]};
            bins high = {[8:15]};
        }
        result_range: coverpoint tr.y {
            bins small = {[0:15]};
            bins large = {[16:31]};
        }
        cross a_range, b_range, result_range;
    endgroup

    // 성능 분석
    real avg_latency = 0.0;
    int sample_count = 0;

    virtual function void write(input transaction t);
        // 기본 검증
        check_functionality(t);

        // Coverage 수집
        tr = t;
        operation_cg.sample();

        // 성능 분석 (실제로는 타이밍 정보 필요)
        update_performance_metrics();
    endfunction

```

```

// 주기적 리포트
if(sample_count % 1000 == 0)
    print_status_report();
endfunction

function void update_performance_metrics();
    sample_count++;
    // 실제 성능 메트릭 업데이트 로직
endfunction
endclass

```

8. UVM Agent Class - 검증 컴포넌트 관리자

Agent Class의 본질과 필요성

Agent Class는 UVM에서 관련된 컴포넌트들을 하나의 그룹으로 관리하는 컨테이너 역할을 합니다. 개발팀에서 팀 리더가 여러 팀원들을 관리하는 것처럼, Agent는 Driver, Monitor, Sequencer를 하나로 묶어서 관리합니다.

Agent 없이 컴포넌트들을 개별 관리하는 경우의 문제점:

문제점	설명	예시
연결 복잡성	각 컴포넌트를 개별적으로 연결해야 함	Environment에서 Driver, Monitor, Sequencer를 각각 생성하고 연결
재사용성 부족	다른 프로젝트에서 개별 컴포넌트만 재사용 어려움	USB Agent, PCIe Agent 등으로 패키징 불가
관리 부담	컴포넌트 수가 많아질수록 관리가 어려워짐	10개 인터페이스 → 30개 컴포넌트 개별 관리
설정 분산	관련 설정이 여러 곳에 분산됨	Interface 설정을 3곳에서 각각 처리

Agent 의 구조적 역할

Agent 없는 구조:

```
Environment
└─ Driver 1      (개별 관리)
└─ Monitor 1     (개별 관리)
└─ Sequencer 1   (개별 관리)
└─ Driver 2      (개별 관리)
└─ Monitor 2     (개별 관리)
└─ Sequencer 2   (개별 관리)
└─ ...
```

Agent 있는 구조:

```
Environment
└─ Agent 1
    └─ Driver
    └─ Monitor
    └─ Sequencer
└─ Agent 2
    └─ Driver
    └─ Monitor
    └─ Sequencer
```

제공된 Agent Class 분석

```
// Agent Class - Driver, Monitor, Sequencer 통합 관리
class agent extends uvm_agent;
`uvm_component_utils(agent)

function new(input string inst = "AGENT", uvm_component c);
super.new(inst, c);
endfunction

monitor m;                                // Monitor 인스턴스
driver d;                                 // Driver 인스턴스
uvm_sequencer #(transaction) seqr;        // Sequencer 인스턴스
```

클래스 구조 분석:

1. 상속 관계:

```
class agent extends uvm_agent;
```

- `uvm_agent`: UVM의 표준 Agent 기능 제공
- Active/Passive 모드 지원
- 표준화된 Agent 동작 보장

2. 멤버 컴포넌트들:

```
monitor m; // 항상 존재 (관찰 역할)  
driver d; // Active 모드에서만 필요  
uvm_sequencer #(transaction) seqr; // Active 모드에서만 필요
```

Active vs Passive Agent:

모드	포함 컴포넌트	역할	사용 상황
Active	Driver + Monitor + Sequencer	신호 생성 + 관찰	DUT 입력 포트 연결
Passive	Monitor 만	관찰만	DUT 출력 포트 연결

```
// Active Agent 구성
```

```
Agent (Active)
└─ Driver      (신호 생성)
└─ Monitor    (신호 관찰)
└─ Sequencer  (시나리오 제공)
```

```
// Passive Agent 구성
```

```
Agent (Passive)
└─ Monitor    (신호 관찰만)
```

Build Phase - 컴포넌트 생성

```
virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
    m = monitor::type_id::create("m",this); // Monitor
    생성
    d = driver::type_id::create("d",this); // Driver
    생성
    seqr = uvm_sequencer #(transaction)::type_id::create("seqr",this);
// Sequencer 생성
endfunction
```

Build Phase 분석:

1. 생성 순서의 중요성:

- 모든 컴포넌트가 동일한 Build Phase에서 생성됨
- 순서는 일반적으로 중요하지 않음 (연결은 Connect Phase에서)
- 각 컴포넌트는 자신만의 Build Phase를 가짐

2. UVM Factory 패턴 사용:

```
m = monitor::type_id::create("m", this);
```

각 매개변수의 의미:

- "m": 컴포넌트 인스턴스 이름
- this: 부모 컴포넌트 (현재 Agent)

3. 계층적 이름 구조:

```
testbench_top
└── uvm_test_top
    └── e (environment)
        └── a (agent)
            ├── m (monitor)
            ├── d (driver)
            └── seqr (sequencer)
```

4. Sequencer 의 템플릿 매개변수:

```
uvm_sequencer #(transaction)
```

- Sequencer 가 처리할 Transaction 타입을 명시
- Driver 와 동일한 타입이어야 연결 가능
- 컴파일 시점에 타입 안전성 보장

Connect Phase - 컴포넌트 연결

```
virtual function void connect_phase(uvm_phase phase);  
super.connect_phase(phase);  
    d.seq_item_port.connect(seqr.seq_item_export); // Driver 와  
Sequencer 연결  
endfunction
```

Connect Phase 의 중요성:

Build Phase 에서는 객체를 생성만 하고, 실제 연결은 Connect Phase 에서 수행합니다.

Build Phase: [Driver] [Sequencer] [Monitor] (개별 생성)

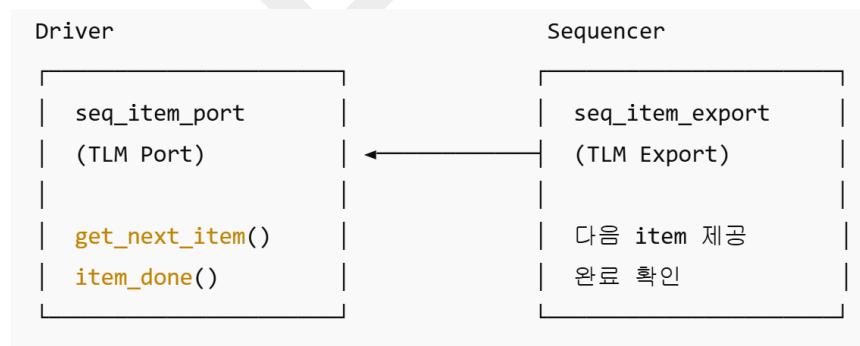
↓

Connect Phase: [Driver] ↔ [Sequencer] [Monitor] (상호 연결)

TLM 연결의 이해:

```
d.seq_item_port.connect(seqr.seq_item_export);
```

이 연결이 의미하는 것:



TLM Port vs Export:

구분	TLM Port	TLM Export
역할	요청자 (Client)	제공자 (Server)
메서드	호출함	구현함
연결 방향	Port → Export	Export ← Port
예시	Driver 의 get_next_item() 호출	Sequencer 의 get_next_item() 구현

Agent 의 설정과 모드 관리

실무에서는 Agent 의 동작 모드를 설정할 수 있어야 합니다:

```
class advanced_agent extends uvm_agent;
    `uvm_component_utils(advanced_agent)

    // 설정 가능한 모드
    uvm_active_passive_enum is_active = UVM_ACTIVE;

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);

        // Monitor는 항상 생성
        m = monitor::type_id::create("m", this);

        // Active 모드에서만 Driver, Sequencer 생성
        if(is_active == UVM_ACTIVE) begin
            d = driver::type_id::create("d", this);
            seqr = uvm_sequencer
        #(transaction)::type_id::create("seqr", this);
        end
    endfunction

    virtual function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);

        // Active 모드에서만 연결
        if(is_active == UVM_ACTIVE) begin
```

```

        d.seq_item_port.connect(seqr.seq_item_export);
    end
endfunction
endclass

```

모드 설정 방법:

```

// Testbench에서 Agent 모드 설정
initial begin
    uvm_config_db #(uvm_active_passive_enum)::set(null, "*.*agent*", "is_active", UVM_PASSIVE);
end

```

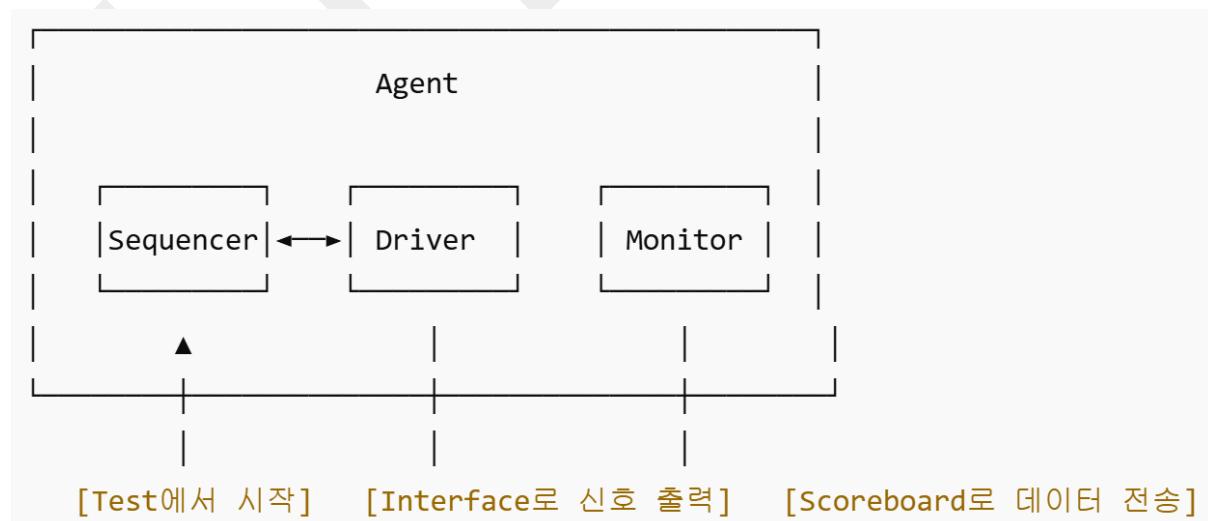
Agent 내부 통신 흐름

Agent 내부에서 컴포넌트들이 어떻게 협력하는지 살펴보겠습니다:

시나리오 실행 흐름:

1. [Test] → Sequence 시작 → [Sequencer]
2. [Sequencer] → Transaction 생성 → [Driver]
3. [Driver] → 하드웨어 신호 → [DUT]
4. [DUT] → 출력 신호 → [Monitor] (same Agent)
5. [Monitor] → Transaction → [Scoreboard] (Environment level)

Agent 내부:



다중 Agent 환경

실무에서는 하나의 시스템에 여러 Agent 가 필요한 경우가 많습니다:

```
class multi_agent_env extends uvm_env;
    agent cpu_agent;      // CPU 인터페이스용
    agent mem_agent;      // 메모리 인터페이스용
    agent pcie_agent;     // PCIe 인터페이스용

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);

        cpu_agent = agent::type_id::create("cpu_agent", this);
        mem_agent = agent::type_id::create("mem_agent", this);
        pcie_agent = agent::type_id::create("pcie_agent", this);

        // 각 Agent 별 설정
        uvm_config_db #(uvm_active_passive_enum)::set(this,
        "cpu_agent", "is_active", UVM_ACTIVE);
        uvm_config_db #(uvm_active_passive_enum)::set(this,
        "mem_agent", "is_active", UVM_PASSIVE);
        uvm_config_db #(uvm_active_passive_enum)::set(this,
        "pcie_agent", "is_active", UVM_ACTIVE);
    endfunction
endclass
```

다중 Agent 의 장점:

장점	설명	예시
모듈화	각 인터페이스별 독립적 관리	CPU Agent 는 CPU 프로토콜만 처리
재사용성	검증된 Agent 를 다른 프로젝트에서 재사용	표준 UART Agent 를 여러 칩에서 사용
병렬 개발	여러 팀이 각각 다른 Agent 개발	CPU 팀, Memory 팀이 독립적 개발
디버깅 용이성	문제 발생 시 해당 Agent 만 집중 분석	PCIe 문제 발생 시 PCIe Agent 만 조사

Agent 사용 시 주의사항

자주하는 실수 1: Connect Phase에서 연결 누락

```
// 문제: Driver 와 Sequencer 연결을 깜빡함
virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    // d.seq_item_port.connect(seqr.seq_item_export); <- 이게 없으면
Driver 가 작동 안함
endfunction
```

자주하는 실수 2: Active/Passive 모드 설정 실수

```
// 문제: Passive Agent 인데 Driver 생성
virtual function void build_phase(uvm_phase phase);
    m = monitor::type_id::create("m", this);

    // is_active 확인 없이 무조건 생성
    d = driver::type_id::create("d", this);           // Passive
모드에서는 불필요
    seqr = uvm_sequencer::type_id::create("seqr", this); // Passive
모드에서는 불필요
endfunction
```

자주하는 실수 3: Interface 설정 전파 실수

```
// Agent 는 받은 Interface 를 하위 컴포넌트에 전달해야 함
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    // Interface 설정을 하위 컴포넌트에 전파
    uvm_config_db #(virtual add_if)::set(this, "m", "aif", aif); // 
Monitor 에게
    uvm_config_db #(virtual add_if)::set(this, "d", "aif", aif); // 
Driver 에게

    m = monitor::type_id::create("m", this);
    d = driver::type_id::create("d", this);
endfunction
```

실무에서의 Agent 설계 패턴

1. 계층적 Agent 구조:

```
// 마스터 Agent (CPU, DMA 등)
class master_agent extends uvm_agent;
    // 능동적으로 트랜잭션 시작
endclass

// 슬레이브 Agent (Memory, Peripheral 등)
class slave_agent extends uvm_agent;
    // 수동적으로 응답만 제공
endclass
```

2. 프로토콜별 Agent:

```
// AXI 프로토콜 전용 Agent
class axi_agent extends uvm_agent;
    axi_driver driver;
    axi_monitor monitor;
    axi_sequencer sequencer;

    // AXI 특화 기능들
    virtual task configure_axi_params();
        // AXI 버스 폭, 클럭 등 설정
    endtask
endclass
```

Agent Class는 관련된 검증 컴포넌트들을 하나로 묶어 관리함으로써, 복잡한 검증 환경을 체계적으로 구성할 수 있게 해줍니다. 재사용 가능한 검증 IP의 기본 단위가 되며, 실무에서는 표준 프로토콜별로 Agent를 개발하여 라이브러리로 관리합니다.

9. UVM Environment Class - 검증 환경 통합 관리

Environment Class의 핵심 역할

Environment Class는 UVM 검증 환경의 최상위 컨테이너로, 모든 검증 컴포넌트들을 통합 관리하는 역할을 합니다. 게임 개발로 비유하면, 개별 캐릭터들(Agent)과 게임 시스템들(Scoreboard)을 하나의 게임 월드로 통합하는 게임 엔진과 같은 역할입니다.

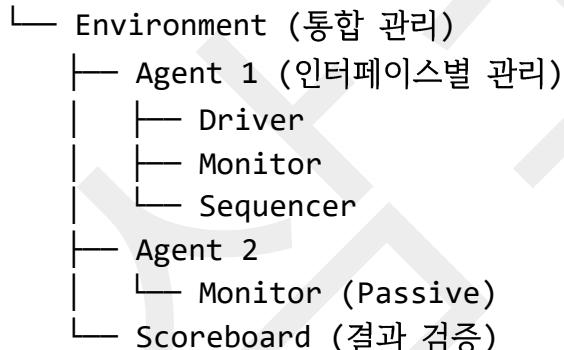
Environment의 주요 책임:

1. 컴포넌트 통합: 모든 Agent 와 Scoreboard를 생성하고 관리
2. 연결 관리: 컴포넌트 간의 TLM 연결 수행
3. 설정 관리: 전역 설정을 하위 컴포넌트에 배포
4. 정책 결정: 검증 환경의 전반적인 동작 방식 결정

검증 환경에서 Environment의 위치

검증 환경 계층 구조:

Test (최상위 제어)



Environment가 없다면 Test에서 모든 컴포넌트를 개별적으로 관리해야 하므로 복잡성이 급격히 증가합니다.

제공된 Environment Class 분석

```
// Environment Class - 검증 환경 통합 관리자
class env extends uvm_env;
`uvm_component_utils(env)
```

```

function new(input string inst = "ENV", uvm_component c);
super.new(inst, c);
endfunction

scoreboard s; // 검증 결과 판정기
agent a; // 컴포넌트 통합 관리자

```

클래스 구조 분석:

1. 상속 관계:

```
class env extends uvm_env;
```

- uvm_env: UVM의 표준 Environment 기능 제공
- 검증 환경 관리를 위한 기본 인프라 제공
- Phase 관리, 설정 전파 등의 기능 자동 지원

2. 멤버 컴포넌트:

```

scoreboard s; // 1개의 Scoreboard
agent a; // 1개의 Agent

```

현재 예시는 간단한 구성이지만, 실무에서는 훨씬 복잡할 수 있습니다:

```

// 실무용 Environment 예시
class complex_env extends uvm_env;
    // 다중 Agent
    master_agent cpu_agent;
    slave_agent mem_agent;
    monitor_agent bus_agent;

    // 다중 Scoreboard
    functional_scoreboard func_sb;
    performance_scoreboard perf_sb;
    coverage_collector cov_collector;

    // 유ти리티 컴포넌트
    virtual_sequencer v_seqr;
    register_model reg_model;
endclass

```

Build Phase - 컴포넌트 생성과 설정

```
virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
  s = scoreboard::type_id::create("s",this); // Scoreboard 생성
  a = agent::type_id::create("a",this);      // Agent 생성
endfunction
```

Build Phase 의 Environment 특성:

1. 생성 순서: Environment의 Build Phase가 실행된 후, 하위 컴포넌트들의 Build Phase가 순차적으로 실행됩니다:

실행 순서:

1. env.build_phase() ← Scoreboard, Agent 생성
2. scoreboard.build_phase() ← Transaction 객체 생성
3. agent.build_phase() ← Driver, Monitor, Sequencer 생성
4. driver.build_phase() ← Transaction 객체, Interface 연결
5. monitor.build_phase() ← Transaction 객체, Interface 연결

2. 설정 전파: Environment는 받은 설정을 하위 컴포넌트에 전파하는 역할을 합니다:

```
virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);

// 전역 설정을 Agent에 전파
uvm_config_db #(int)::set(this, "a*", "timeout", 1000);
uvm_config_db #(bit)::set(this, "a*", "enable_coverage", 1);

s = scoreboard::type_id::create("s", this);
a = agent::type_id::create("a", this);
endfunction
```

3. 조건부 생성: 실무에서는 테스트 종류에 따라 컴포넌트를 조건부로 생성할 수 있습니다:

```
virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
```

```

bit enable_coverage;
int num_agents;

// 설정 읽기
uvm_config_db #(bit)::get(this, "", "enable_coverage",
enable_coverage);
uvm_config_db #(int)::get(this, "", "num_agents", num_agents);

// 기본 컴포넌트 생성
s = scoreboard::type_id::create("s", this);

// 조건부 생성
if(enable_coverage) begin
    cov_collector = coverage_collector::type_id::create("cov",
this);
end

// 동적 Agent 생성
agents = new[num_agents];
for(int i = 0; i < num_agents; i++) begin
    agents[i] = agent::type_id::create($sformatf("agent_%0d", i),
this);
end
endfunction

```

Connect Phase - 컴포넌트 간 연결

```

virtual function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
a.m.send.connect(s.recv); // Agent 의 Monitor 를 Scoreboard 에 연결
endfunction

```

Connect Phase 동작 분석:

1. TLM 연결:

```
a.m.send.connect(s.recv);
```

이 연결의 의미를 단계별로 분석하면:

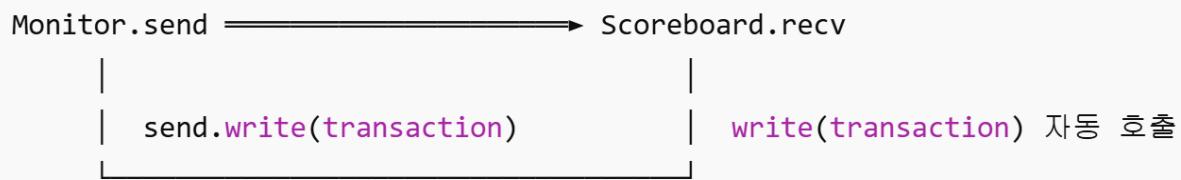
- a: Environment 의 Agent 인스턴스
- a.m: Agent 내부의 Monitor 인스턴스
- a.m.send: Monitor 의 Analysis Port
- s.recv: Scoreboard 의 Analysis Implementation Port
- .connect(): TLM 포트 연결 메서드

2. 연결 결과:

연결 전:

Monitor.send —(분리됨)— Scoreboard.recv

연결 후:



3. 데이터 플로우: 연결이 완료되면 다음과 같은 자동 데이터 플로우가 생성됩니다:

1. Monitor에서 send.write(t) 호출
2. UVM 인프라가 자동으로 Scoreboard.write(t) 호출
3. Scoreboard에서 검증 로직 실행

복잡한 Environment 의 연결 패턴

실무에서는 더 복잡한 연결이 필요합니다:

```
virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
```

```

// 1:1 연결

cpu_agent.monitor.analysis_port.connect(func_scoreboard.analysis_export);

// 1:N 연결 (하나의 Monitor 가 여러 Scoreboard 에 연결)

mem_agent.monitor.analysis_port.connect(func_scoreboard.mem_export);

mem_agent.monitor.analysis_port.connect(perf_scoreboard.mem_export);

mem_agent.monitor.analysis_port.connect(cov_collector.mem_export);

// N:1 연결을 위한 TLM FIFO 사용
cpu_tlm_fifo = new("cpu_tlm_fifo", this);
mem_tlm_fifo = new("mem_tlm_fifo", this);

cpu_agent.monitor.analysis_port.connect(cpu_tlm_fifo.analysis_export);

mem_agent.monitor.analysis_port.connect(mem_tlm_fifo.analysis_export);

correlation_scoreboard.cpu_port.connect(cpu_tlm_fifo.get_export);

correlation_scoreboard.mem_port.connect(mem_tlm_fifo.get_export);
endfunction

```

Environment 설정 관리 패턴

1. 계층적 설정 전파:

```
class env extends uvm_env;
    typedef struct {
        bit enable_errors;
        bit enable_warnings;
        int verbosity_level;
        real timeout_ns;
    } env_config_t;

    env_config_t cfg;

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);

        // 설정 읽기
        if(!uvm_config_db #(env_config_t)::get(this, "", "env_cfg",
        cfg)) begin
            `uvm_info("ENV", "Using default configuration", UVM_LOW)
            cfg = '{enable_errors: 1, enable_warnings: 1,
        verbosity_level: 200, timeout_ns: 1000.0};
        end

        // 하위 컴포넌트에 설정 전파
        uvm_config_db #(bit)::set(this, "*", "enable_errors",
        cfg.enable_errors);
        uvm_config_db #(real)::set(this, "*", "timeout_ns",
        cfg.timeout_ns);

        // 컴포넌트 생성
        s = scoreboard::type_id::create("s", this);
        a = agent::type_id::create("a", this);
    endfunction
endclass
```

2. 인터페이스 관리:

```
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    virtual add_if aif;

    // 인터페이스 설정 받기
    if(!uvm_config_db #(virtual add_if)::get(this, "", "aif", aif))
        `uvm_fatal("ENV", "Failed to get interface from config_db");

    // Agent에 인터페이스 전달
    uvm_config_db #(virtual add_if)::set(this, "a*", "aif", aif);

    // 컴포넌트 생성
    s = scoreboard::type_id::create("s", this);
    a = agent::type_id::create("a", this);
endfunction
```

Environment 의 실행 제어

Environment 는 검증 실행 중에도 전반적인 제어를 담당할 수 있습니다:

```
class env extends uvm_env;
    virtual task run_phase(uvm_phase phase);
        fork
            // 타임아웃 감시
            timeout_monitor();

            // 에러 감시
            error_monitor();

            // 성능 모니터링
            performance_monitor();
        join_none
    endtask

    virtual task timeout_monitor();
        #10000ns; // 10us 타임아웃
    endtask
```

```

`uvm_fatal("TIMEOUT", "Test timeout occurred");
endtask

virtual task error_monitor();
    forever begin
        wait(error_detected);
        if(cfg.stop_on_error) begin
            `uvm_error("ENV", "Stopping test due to error");
            global_stop_request();
        end
        error_detected = 0;
    end
endtask
endclass

```

Environment 사용 시 주의사항

자주하는 실수 1: 연결 순서 오류

```

// 문제: Connect Phase에서 존재하지 않는 컴포넌트 연결 시도
virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);

// 만약 Agent의 Build Phase에서 Monitor 생성이 실패했다면
    a.m.send.connect(s.recv); // Null pointer 에러 발생 가능
endfunction

// 해결: 연결 전 존재 여부 확인
virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);

    if(a != null && a.m != null && s != null) begin
        a.m.send.connect(s.recv);
    end else begin
        `uvm_error("ENV", "Components not properly created");
    end
endfunction

```

자주하는 실수 2: 설정 전파 타이밍 오류

```
// 문제: 컴포넌트 생성 후 설정 전파
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

// 컴포넌트 먼저 생성
s = scoreboard::type_id::create("s", this);
a = agent::type_id::create("a", this);

// 설정은 나중에 - 너무 늦음!
uvm_config_db #(virtual add_if)::set(this, "a*", "aif", aif);
endfunction

// 해결: 설정 먼저, 생성 나중에
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

// 설정 먼저 전파
uvm_config_db #(virtual add_if)::set(this, "a*", "aif", aif);

// 그 다음에 컴포넌트 생성
s = scoreboard::type_id::create("s", this);
a = agent::type_id::create("a", this);
endfunction
```

자주하는 실수 3: 메모리 누수

```
// 문제: 동적으로 생성한 객체들의 정리 누락
class env extends uvm_env;
    agent agents[];

    virtual function void build_phase(uvm_phase phase);
        agents = new[num_agents];
        for(int i = 0; i < num_agents; i++) begin
            agents[i] = agent::type_id::create($sformatf("agent_%0d",
i), this);
        end
    endfunction
```

```

// final_phase에서 정리 필요 (UVM이 자동으로 처리하지만 명시적으로 하는
것이 좋음)
virtual function void final_phase(uvm_phase phase);
    super.final_phase(phase);
    // 필요시 추가 정리 작업
endfunction
endclass

```

실무에서의 Environment 설계 패턴

1. 계층화된 Environment:

```

// 시스템 레벨 Environment
class system_env extends uvm_env;
    cpu_env cpu_subsystem;
    memory_env mem_subsystem;
    io_env io_subsystem;
    system_scoreboard sys_sb;
endclass

```

```

// 서브시스템 레벨 Environment
class cpu_env extends uvm_env;
    cpu_agent cpu_agent;
    cache_agent cache_agent;
    cpu_scoreboard cpu_sb;
endclass

```

2. 설정 기반 Environment:

```

class configurable_env extends uvm_env;
    typedef enum {SIMPLE, MEDIUM, COMPLEX} complexity_e;

    complexity_e complexity = SIMPLE;

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);

        case(complexity)
            SIMPLE: begin
                create_basic_components();

```

```
end
MEDIUM: begin
    create_basic_components();
    create_advanced_scoreboards();
end
COMPLEX: begin
    create_basic_components();
    create_advanced_scoreboards();
    create_performance_monitors();
    create_coverage_collectors();
end
endcase
endfunction
endclass
```

10. UVM Test Class - 검증 시나리오 제어기

Test Class의 본질적 역할

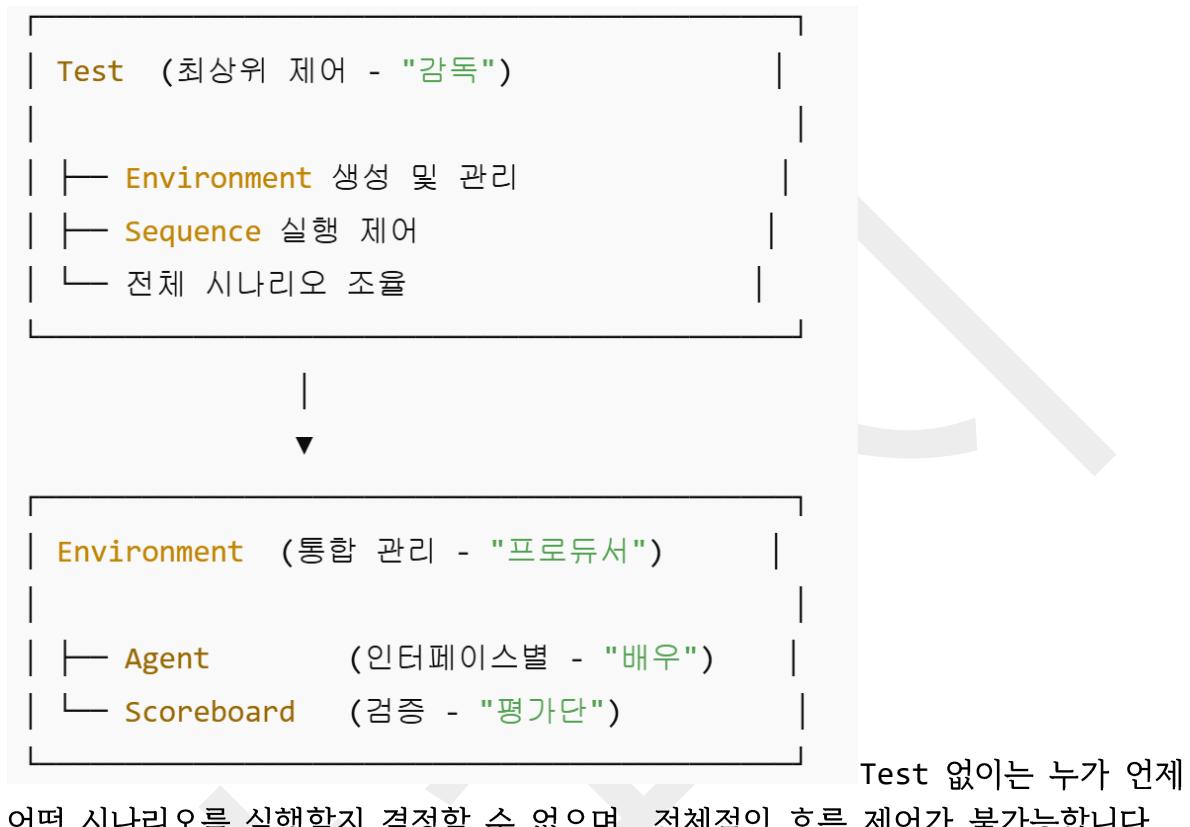
Test Class는 UVM 검증 환경의 최상위 제어자로, 전체 검증 시나리오를 조율하고 실행하는 역할을 합니다. 영어 제작으로 비유하면, 모든 배우(컴포넌트)와 스태프(Environment)를 총괄하는 감독의 역할과 같습니다.

Test Class의 핵심 책임:

1. 시나리오 제어: 어떤 Sequence를 언제 실행할지 결정
2. Environment 관리: 검증 환경의 생성과 설정 담당
3. 테스트 정책: Pass/Fail 기준, 타임아웃, 종료 조건 등 결정
4. 결과 분석: 최종 검증 결과 수집 및 분석

검증 환경에서 Test 의 위치

UVM 검증 계층 구조:



Test 없이는 누가 언제

어떤 시나리오를 실행할지 결정할 수 없으며, 전체적인 흐름 제어가 불가능합니다.

제공된 Test Class 분석

```
// Test Class - 검증 시나리오 제어기
class test extends uvm_test;
`uvm_component_utils(test)

function new(input string inst = "TEST", uvm_component c);
super.new(inst, c);
endfunction

generator gen; // Sequence 생성기
env e; // Environment 인스턴스
```

클래스 구조 분석:

1. 상속 관계:

```
class test extends uvm_test;
```

- `uvm_test`: UVM의 표준 Test 기능 제공
- 테스트 실행 제어를 위한 기본 인프라 자동 제공
- UVM Phase 관리 및 결과 수집 기능 포함

중요한 개념: `uvm_test`는 UVM 환경의 루트 컴포넌트입니다. 모든 다른 컴포넌트들은 이 Test를 최상위 부모로 가집니다.

2. 멤버 컴포넌트:

```
generator gen; // Sequence (테스트 시나리오 생성기)  
env e; // Environment (전체 검증 환경)
```

이 구조를 보면 Test가 두 가지 핵심 요소를 직접 관리함을 알 수 있습니다:

- 무엇을 테스트할지 (Sequence through generator)
- 어떻게 테스트할지 (Environment through env)

Build Phase - 검증 환경 구축

```
virtual function void build_phase(uvm_phase phase);  
super.build_phase(phase);  
gen = generator::type_id::create("gen"); // Sequence 생성  
e = env::type_id::create("e",this); // Environment 생성  
endfunction
```

Build Phase 분석:

1. 생성 순서의 중요성:

실행 순서:

1. test.build_phase() ← Generator, Environment 생성
2. env.build_phase() ← Agent, Scoreboard 생성
3. agent.build_phase() ← Driver, Monitor, Sequencer 생성
4. driver/monitor.build_phase() ← Interface 연결 및 초기화

2. Generator vs Environment 생성 차이:

```
gen = generator::type_id::create("gen");      // 부모 없음  
e = env::type_id::create("e", this);          // this 가 부모
```

i) 차이점의 의미:

- **Generator:** Test 가 직접 실행하는 독립적 객체
- **Environment:** Test 의 하위 컴포넌트로 계층 구조 형성

3. 계층적 이름 구조:

```
testbench_top  
└── uvm_test_top (test 인스턴스)  
    ├── gen (generator, 독립적)  
    └── e (environment)  
        ├── a (agent)  
        └── s (scoreboard)
```

Run Phase - 테스트 시나리오 실행

```
virtual task run_phase(uvm_phase phase);  
    phase.raise_objection(this);    // 테스트 시작 선언  
    gen.start(e.a.seqr);          // Sequence 를 Sequencer 에서 실행  
    #50;                          // 50 시간 단위 대기  
    phase.drop_objection(this);    // 테스트 완료 선언  
endtask
```

Run Phase 동작 분석:

1. Objection 메커니즘:

```
phase.raise_objection(this);  
// ... 테스트 실행 ...  
phase.drop_objection(this);
```

Objection이 필요한 이유:

상황	Objection 없음	Objection 있음
테스트 시작	UVM이 즉시 종료 판단	UVM이 테스트 진행 중임을 인식
테스트 진행	예상치 못한 종료 가능	안정적인 테스트 실행
테스트 완료	명확한 종료 시점 없음	명시적 종료 선언

Objection 동작 방식:

Timeline:

T=0 raise_objection() → "테스트 시작합니다!"
T=1-49 테스트 실행 중 → "아직 진행 중..."
T=50 drop_objection() → "테스트 완료했습니다!"
T=51 UVM 자동 종료 → "모든 objection이 해제되었으므로 종료"

2. Sequence 실행:

```
gen.start(e.a.seqr);
```

이 코드의 의미를 단계별로 분석하면:

- gen: Test에서 생성한 generator (Sequence)
- e.a.seqr: Environment → Agent → Sequencer 경로
- .start(): Sequence를 지정된 Sequencer에서 시작

Sequence 실행 플로우:

1. Test: gen.start(sequencer) 호출
2. UVM: Sequence의 body() task 실행
3. Generator: Transaction들을 Sequencer에 전달
4. Sequencer: Transaction들을 Driver에게 순차 전달
5. Driver: Transaction을 하드웨어 신호로 변환
6. DUT: 하드웨어 처리 수행
7. Monitor: 결과 신호를 Transaction으로 변환
8. Scoreboard: 결과 검증 수행

3. 타이밍 제어:

```
#50;
```

이 딜레이의 목적:

- Sequence 완료를 위한 충분한 시간 제공
- 모든 Transaction이 완전히 처리될 때까지 대기
- Monitor 와 Scoreboard가 결과를 처리할 시간 확보

다양한 Test 시나리오 패턴

1. 기본 순차 실행:

```
// 현재 예시: 하나의 Sequence 실행
virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    gen.start(e.a.seqr);
    #50;
    phase.drop_objection(this);
endtask
```

2. 다중 Sequence 순차 실행:

```
virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    // 기본 기능 테스트
```

```

basic_test_gen.start(e.a.seqr);
#100;

// 경계값 테스트
boundary_test_gen.start(e.a.seqr);
#100;

// 스트레스 테스트
stress_test_gen.start(e.a.seqr);
#200;

phase.drop_objection(this);
endtask

```

3. 병렬 Sequence 실행:

```

virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    fork
        // CPU Agent에서 실행
        cpu_sequence.start(e.cpu_agent.seqr);

        // Memory Agent에서 동시 실행
        mem_sequence.start(e.mem_agent.seqr);

        // 백그라운드 모니터링
        monitor_sequence.start(e.monitor_agent.seqr);
    join

    phase.drop_objection(this);
endtask

```

4. 조건부 테스트:

```

virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    string test_type;
    uvm_config_db #(string)::get(this, "", "test_type", test_type);

```

```

case(test_type)
    "smoke": begin
        smoke_test_sequence.start(e.a.seqr);
        #100;
    end
    "regression": begin
        regression_test_sequence.start(e.a.seqr);
        #1000;
    end
    "stress": begin
        stress_test_sequence.start(e.a.seqr);
        #10000;
    end
    default: begin
        `uvm_error("TEST", $sformatf("Unknown test type: %s",
test_type));
    end
endcase

phase.drop_objection(this);
endtask

```

Test 설정 및 환경 제어

1. Environment 설정:

```

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);

// Environment 설정
env_config_t env_cfg;
env_cfg.enable_coverage = 1;
env_cfg.enable_assertions = 1;
env_cfg.verbosity_level = UVM_MEDIUM;

uvm_config_db #(env_config_t)::set(this, "e*", "env_cfg",
env_cfg);

// 컴포넌트 생성

```

```
gen = generator::type_id::create("gen");
e = env::type_id::create("e", this);
endfunction
```

2. Interface 연결:

```
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    virtual add_if aif;

    // 최상위에서 Interface 가져오기
    if(!uvm_config_db #(virtual add_if)::get(this, "", "aif", aif))
        `uvm_fatal("TEST", "Failed to get interface");

    // Environment에 Interface 전달
    uvm_config_db #(virtual add_if)::set(this, "e*", "aif", aif);

    gen = generator::type_id::create("gen");
    e = env::type_id::create("e", this);
endfunction
```

고급 Test 제어 기능

1. 동적 타임아웃 관리:

```
virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    fork
        // 메인 테스트
        begin
            gen.start(e.a.seqr);
        end

        // 타임아웃 감시
        begin
            #timeout_ns;
            `uvm_error("TIMEOUT", $sformatf("Test timeout after %0d ns", timeout_ns));
        end
    join
endtask
```

```

        end
join_any

 disable fork; // 모든 포크된 태스크 종료
phase.drop_objection(this);
endtask

```

2. 에러 감시 및 조기 종료:

```

class test extends uvm_test;
int error_count = 0;
int max_errors = 10;

virtual task run_phase(uvm_phase phase);
phase.raise_objection(this);

fork
// 메인 테스트
main_test_flow();

// 에러 감시
error_monitor();
join_any

if(error_count >= max_errors) begin
`uvm_error("TEST", $sformatf("Test stopped due to
excessive errors: %0d", error_count));
end

phase.drop_objection(this);
endtask

virtual task error_monitor();
forever begin
@(e.s.error_detected); // Scoreboard에서 에러 감지 시
error_count++;
if(error_count >= max_errors) begin
`uvm_warning("TEST", "Maximum error count reached,
stopping test");
return;

```

```
    end
  end
endtask
endclass
```

3. Coverage 기반 조기 종료:

```
virtual task run_phase(uvm_phase phase);
  phase.raise_objection(this);

  fork
    // 메인 테스트 (최대 실행 시간)
    begin
      gen.start(e.a.seqr);
    end

    // Coverage 감시 (목표 달성을 시 조기 종료)
    begin
      forever begin
        #1000; // 주기적으로 확인
        if(e.cov_collector.get_coverage() >= 95.0) begin
          `uvm_info("TEST", $sformatf("Coverage target
achieved: %.2f%%",
                                e.cov_collector.get_coverage()), UVM_LOW);
          break;
        end
      end
    end
  end
  join_any

  disable fork;
  phase.drop_objection(this);
endtask
```

Test 사용 시 주의사항

자주하는 실수 1: Objection 관리 실수

```
// 문제: raise_objection과 drop_objection 불일치
virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    if(some_condition) begin
        return; // drop_objection 없이 종료
    end

    gen.start(e.a.seqr);
    phase.drop_objection(this);
endtask

// 해결: try-finally 패턴 사용 (SystemVerilog에서는 수동으로)
virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    begin // 보호 블록
        if(some_condition) begin
            `uvm_info("TEST", "Early exit condition met", UVM_LOW);
        end else begin
            gen.start(e.a.seqr);
        end
    end

    phase.drop_objection(this); // 항상 실행됨
endtask
```

자주하는 실수 2: 불충분한 대기 시간

```
// 문제: Sequence 완료를 기다리지 않음
virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    gen.start(e.a.seqr);
    // #50; <- 이게 없으면 Sequence 가 완료되기 전에 테스트 종료
    phase.drop_objection(this);
```

```

endtask

// 해결: 충분한 대기 또는 완료 감지
virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    fork
        gen.start(e.a.seqr);
    join

    // 모든 처리가 완료될 때까지 추가 대기
    #50;

    phase.drop_objection(this);
endtask

```

자주하는 실수 3: Environment 설정 누락

```

// 문제: Interface 나 설정을 Environment에 전달하지 않음
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    // Interface 설정 누락
    // uvm_config_db #(virtual add_if)::set(...); <- 이게 없으면
Driver/Monitor 연결 실패

    gen = generator::type_id::create("gen");
    e = env::type_id::create("e", this);
endfunction

```

실무에서의 Test 설계 패턴

1. 매개변수화된 Test:

```

class parameterized_test extends uvm_test;
    `uvm_component_utils(parameterized_test)

    int num_transactions = 100;
    string sequence_type = "random";
    bit enable_coverage = 1;

```

```

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    // 매개변수 읽기
    uvm_config_db #(int)::get(this, "", "num_transactions",
num_transactions);
    uvm_config_db #(string)::get(this, "", "sequence_type",
sequence_type);
    uvm_config_db #(bit)::get(this, "", "enable_coverage",
enable_coverage);

    // 매개변수에 따른 다른 Sequence 생성
    case(sequence_type)
        "random": gen = random_generator::type_id::create("gen");
        "directed": gen =
directed_generator::type_id::create("gen");
        "corner": gen =
corner_case_generator::type_id::create("gen");
    endcase

    e = env::type_id::create("e", this);
endfunction
endclass

```

2. 계층화된 Test 구조:

```

// 기본 Test 클래스
virtual class base_test extends uvm_test;
    env e;

    virtual function void build_phase(uvm_phase phase);
        // 공통 설정
    endfunction

    pure virtual task main_test(); // 서브클래스에서 구현 필수

    virtual task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        main_test();
    endtask

```

```

        phase.drop_objection(this);
    endtask
endclass

// 구체적인 Test 구현들
class smoke_test extends base_test;
    virtual task main_test();
        // 기본 기능 테스트
    endtask
endclass

class regression_test extends base_test;
    virtual task main_test();
        // 회귀 테스트
    endtask
endclass

```

Test Class는 전체 검증 시나리오의 지휘자 역할을 하며, Environment 와 Sequence 를 조율하여 체계적이고 효율적인 검증을 수행합니다. 적절한 제어 로직과 여러 처리를 통해 안정적이고 신뢰성 있는 검증 환경을 구축할 수 있습니다.

11. Testbench Top Module - 하드웨어와 소프트웨어의 만남

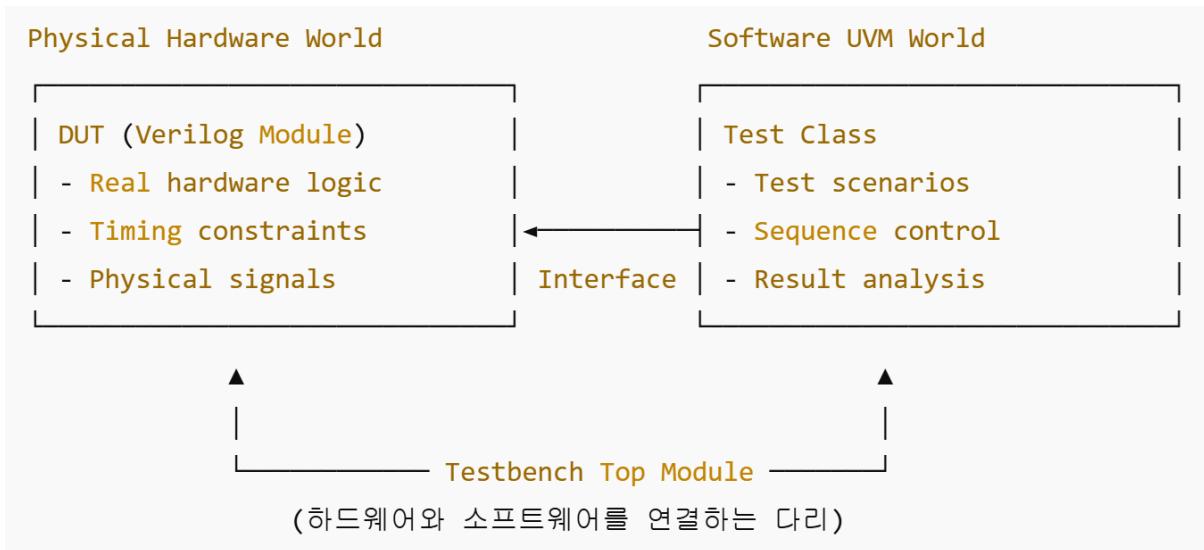
Testbench Top 의 핵심 역할

Testbench Top Module 은 UVM 검증 환경에서 하드웨어와 소프트웨어가 만나는 접점입니다. 실제 하드웨어(DUT)와 SystemVerilog 기반의 소프트웨어 검증 환경을 물리적으로 연결하는 역할을 담당합니다.

Testbench Top 의 주요 기능:

1. 하드웨어 인스턴스화: DUT 를 실제로 생성하고 연결
2. Interface 생성: 하드웨어와 소프트웨어 간의 통신 통로 구축
3. UVM 환경 시작: 소프트웨어 검증 환경 실행 시작
4. 시뮬레이션 제어: 파일 덤프, 클럭 생성 등 시뮬레이션 환경 설정

하드웨어와 소프트웨어의 경계



제공된 Testbench Top Module 분석

```
// Testbench Top Module - 하드웨어와 소프트웨어의 연결점
module add_tb();

add_if aif(); // Interface 인스턴스 생성

add dut (.a(aif.a), .b(aif.b), .y(aif.y)); // DUT 인스턴스화 및
연결

initial begin
$dumpfile("dump.vcd"); // 파형 덤프 파일 설정
$dumpvars; // 모든 변수를 덤프에 포함
end

initial begin
uvm_config_db #(virtual add_if)::set(null, "uvm_test_top.e.a*", "aif", aif); // Interface를 UVM에 전달
run_test("test"); // UVM 테스트 실행 시작
end

endmodule
```

모듈 구조 분석:

1. Interface 생성:

```
add_if aif();
```

- 하드웨어와 소프트웨어 간의 통신 채널 생성
- DUT의 포트와 UVM 컴포넌트를 연결하는 브릿지
- 물리적 신호선을 객체지향적으로 추상화

2. DUT 인스턴스화:

```
add dut (.a(aif.a), .b(aif.b), .y(aif.y));
```

이 연결의 의미:

- add: 검증 대상 하드웨어 모듈
- dut: DUT 인스턴스 이름 (Design Under Test)
- .a(aif.a): DUT의 a 포트를 Interface의 a 신호에 연결
- .b(aif.b): DUT의 b 포트를 Interface의 b 신호에 연결
- .y(aif.y): DUT의 y 포트를 Interface의 y 신호에 연결

포트 연결 방식 비교:

방식	코드 예시	특징
직접 연결	add dut(a, b, y);	순서 의존적, 실수 가능성 높음
명시적 연결	add dut(.a(sig_a), .b(sig_b), .y(sig_y));	명확하지만 신호 분산
Interface 연결	add dut(.a(aif.a), .b(aif.b), .y(aif.y));	그룹화되고 관리 용이

3. 시뮬레이션 환경 설정:

```
initial begin
$dumpfile("dump.vcd");
$dumpvars;
end
```

VCD 덤프의 중요성:

- **VCD**: Value Change Dump 의 약자
- 시뮬레이션 중 모든 신호 변화를 기록
- 파형 뷰어(GTKWave, ModelSim 등)에서 분석 가능
- 하드웨어 디버깅의 핵심 도구

VCD 파일 내용 예시:

```
$timescale 1ns $end
$var wire 4 ! a $end
$var wire 4 " b $end
$var wire 5 # y $end
$dumpvars
#0
b0101 !
b0011 "
b01000 #
#10
b1010 !
b0110 "
b10000 #
```

UVM 환경 연결과 실행

```
initial begin
  uvm_config_db #(virtual add_if)::set(null, "uvm_test_top.e.a*",
  "aif", aif);
  run_test("test");
end
```

Configuration Database 설정:

이 코드를 매개변수별로 분석하면:

매개변수	값	의미
null	최상위	설정을 수행하는 컴포넌트 (testbench top)
"uvm_test_top.e.a*"	경로 패턴	설정을 받을 컴포넌트 경로
"aif"	키 이름	Configuration database에서 사용할 키
aif	실제 값	전달할 Interface 인스턴스

경로 패턴 해석:

```
"uvm_test_top.e.a*"
  |   |   |   └─ 와일드카드 (a로 시작하는 모든 하위 컴포넌트)
  |   |   └─ Agent 이름
  |   └─ Environment 이름
  └─ UVM 최상위 테스트 이름
```

이 설정으로 인해 다음 컴포넌트들이 Interface에 접근할 수 있게 됩니다:

- uvm_test_top.e.a (Agent)
- uvm_test_top.e.a.d (Driver)
- uvm_test_top.e.a.m (Monitor)

run_test() 함수:

```
run_test("test");
```

이 함수 호출이 시작하는 일들:

1. UVM Factory에서 "test" 이름으로 등록된 클래스 검색
2. Test 인스턴스 생성 및 "uvm_test_top" 이름으로 등록
3. UVM Phase 시스템 시작 (build → connect → run → ...)
4. 모든 objection이 해제될 때까지 시뮬레이션 계속

Testbench Top에서의 클럭 생성

실제 프로젝트에서는 클럭이 필요한 경우가 많습니다:

```
module enhanced_testbench_top();
  // 클럭 및 리셋 신호
  logic clk = 0;
  logic rst_n = 0;

  // 클럭 생성 (10ns 주기 = 100MHz)
  always #5 clk = ~clk;

  // 리셋 시퀀스
  initial begin
```

```

rst_n = 0;
#100;           // 100ns 동안 리셋 유지
rst_n = 1;
`uvm_info("TB_TOP", "Reset released", UVM_LOW)
end

// Interface 생성 (클럭 포함)
add_if aif(clk, rst_n);

// DUT 연결 (클럭, 리셋 포함)
add dut(
    .clk(clk),
    .rst_n(rst_n),
    .a(aif.a),
    .b(aif.b),
    .y(aif.y)
);

// VCD 덤프
initial begin
    $dumpfile("enhanced_dump.vcd");
    $dumpvars(0, enhanced_testbench_top); // 모든 계층 포함
end

// UVM 환경 시작
initial begin
    // 클럭과 리셋도 UVM에 전달
    uvm_config_db #(virtual add_if)::set(null, "*", "aif", aif);

    // 테스트 타입을 매개변수로 전달
    uvm_config_db #(string)::set(null, "*", "test_type",
"regression");

    run_test("test");
end

// 시뮬레이션 타임아웃 (안전장치)
initial begin
    #1000000; // 1ms 후 강제 종료

```

```

`uvm_fatal("TB_TOP", "Simulation timeout occurred");
end

endmodule

```

다중 Interface 환경

복잡한 시스템에서는 여러 Interface 가 필요합니다:

```

module complex_testbench_top();
    logic clk = 0;
    logic rst_n = 0;

    always #5 clk = ~clk;

    // 여러 Interface 생성
    cpu_if  cpu_interface(clk, rst_n);
    mem_if  mem_interface(clk, rst_n);
    pcie_if pcie_interface(clk, rst_n);

    // 복합 DUT 연결
    complex_soc dut(
        .clk(clk),
        .rst_n(rst_n),

        // CPU 인터페이스
        .cpu_addr(cpu_interface.addr),
        .cpu_data(cpu_interface.data),
        .cpu_valid(cpu_interface.valid),

        // 메모리 인터페이스
        .mem_addr(mem_interface.addr),
        .mem_rdata(mem_interface.rdata),
        .mem_ready(mem_interface.ready),

        // PCIe 인터페이스
        .pcie_tx(pcie_interface.tx),
        .pcie_rx(pcie_interface.rx)
    );

```

```

// 각 Interface 를 해당 Agent 에 전달
initial begin
    uvm_config_db #(virtual cpu_if)::set(null, "*cpu_agent*", "vif", cpu_interface);
    uvm_config_db #(virtual mem_if)::set(null, "*mem_agent*", "vif", mem_interface);
    uvm_config_db #(virtual pcie_if)::set(null, "*pcie_agent*", "vif", pcie_interface);

    run_test(); // 테스트 타입을 커맨드라인에서 지정
end
endmodule

```

커맨드라인 인터페이스

실무에서는 시뮬레이션을 다양한 옵션으로 실행할 수 있어야 합니다:

```

module flexible_testbench_top();
string test_name = "default_test";
int seed = 1;
string dump_file = "simulation.vcd";

// 커맨드라인 인자 처리
initial begin
    if($value$plusargs("TEST=%s", test_name)) begin
        `uvm_info("TB_TOP", $sformatf("Running test: %s",
test_name), UVM_LOW)
    end

    if($value$plusargs("SEED=%d", seed)) begin
        `uvm_info("TB_TOP", $sformatf("Using seed: %0d", seed),
UVM_LOW)
    end

    if($value$plusargs("DUMP=%s", dump_file)) begin
        `uvm_info("TB_TOP", $sformatf("Dump file: %s", dump_file),
UVM_LOW)
    end

    // VCD 덤프 설정

```

```

$dumpfile(dump_file);
$dumpvars;

// 시드 설정
$urandom(seed);

// UVM 설정
uvm_config_db #(virtual add_if)::set(null, "*", "aif", aif);
uvm_config_db #(int)::set(null, "*", "seed", seed);

run_test(test_name);
end
endmodule

```

시뮬레이션 실행 예시:

```

# 기본 테스트 실행
vsim -do "run -all" testbench_top

# 특정 테스트와 시드로 실행
vsim +TEST=stress_test +SEED=12345 +DUMP=stress.vcd -do "run -all"
testbench_top

# 회귀 테스트 실행
vsim +TEST=regression_test +SEED=random -do "run -all" testbench_top

```

Testbench Top 설계 시 주의사항

자주하는 실수 1: Interface 연결 불일치

```

// 문제: DUT 포트와 Interface 신호 이름 불일치
add_if aif();
add dut(.input_a(aif.a), .input_b(aif.b), .output_sum(aif.y));
    // ▲ DUT에서는 다른 포트명 사용

// 해결: 정확한 포트명 사용 또는 Interface에서 신호명 통일
add dut(.input_a(aif.a), .input_b(aif.b), .output_sum(aif.y));

```

자주하는 실수 2: Configuration DB 경로 오류

```
// 문제: 잘못된 경로로 Interface 설정  
uvm_config_db #(virtual add_if)::set(null, "wrong_path*", "aif",  
aif);  
  
// 해결: 정확한 UVM 계층 구조 경로 사용  
uvm_config_db #(virtual add_if)::set(null, "uvm_test_top.e.a*",  
"aif", aif);
```

자주하는 실수 3: VCD 덤프 누락

```
// 문제: 디버깅 정보 없음  
module bad_tb();  
    // $dumpfile, $dumpvars 없음  
    // 파형 분석 불가능  
endmodule  
  
// 해결: 적절한 덤프 설정  
initial begin  
    $dumpfile("debug.vcd");  
    $dumpvars(0, testbench_top); // 모든 계층의 모든 신호 덤프  
end
```

실무에서의 Testbench Top 패턴

1. 모듈화된 설정:

```
module testbench_top();  
    // 설정 매개변수들  
    parameter real CLK_PERIOD = 10.0; // 10ns  
    parameter int TIMEOUT_CYCLES = 10000;  
  
    `include "testbench_config.svh" // 설정 파일 분리  
  
    // 공통 신호들  
    `include "common_signals.svh" // 신호 선언 분리  
  
    // DUT 인스턴스화
```

```

`include "dut_connection.svh"      // DUT 연결 분리

// UVM 환경 설정
`include "uvm_setup.svh"          // UVM 설정 분리
endmodule

```

2. 조건부 컴파일:

```

module conditional_testbench_top();
  `ifdef ENABLE_ASSERTIONS
    // Assertion 바인딩
    bind dut assertion_module assertion_inst(.*);
  `endif

  `ifdef ENABLE_COVERAGE
    // Coverage 모델 바인딩
    bind dut coverage_module coverage_inst(.*);
  `endif

  `ifdef GATE_LEVEL_SIM
    // 게이트 레벨 시뮬레이션 설정
    `include "gate_level_setup.svh"
  `else
    // RTL 시뮬레이션 설정
    `include "rtl_setup.svh"
  `endif
endmodule

```

Testbench Top Module은 하드웨어와 소프트웨어 검증 환경을 연결하는 핵심 인터페이스입니다. 적절한 설정과 연결을 통해 효율적이고 확장 가능한 검증 환경을 구축할 수 있으며, 다양한 시뮬레이션 시나리오를 지원할 수 있습니다.

12. 전체 시뮬레이션 플로우와 실무 적용

UVM 검증 환경의 전체 실행 플로우

이제 모든 컴포넌트들이 어떻게 협력하여 하나의 완전한 검증 시스템을 구성하는지 전체 플로우를 통해 살펴보겠습니다.

Phase 별 실행 순서

UVM 시뮬레이션은 여러 Phase로 나누어 체계적으로 실행됩니다:

UVM Phase Execution Flow:

1. Build Phase (구성 단계)
 - └── test.build_phase() : Generator, Environment 생성
 - └── env.build_phase() : Agent, Scoreboard 생성
 - └── agent.build_phase() : Driver, Monitor, Sequencer 생성
 - └── driver.build_phase() : Interface 연결, Transaction 생성
 - └── monitor.build_phase() : Interface 연결, Analysis Port

생성

 - └── scoreboard.build_phase() : Analysis Imp Port 생성
2. Connect Phase (연결 단계)
 - └── agent.connect_phase() : Driver \leftrightarrow Sequencer 연결
 - └── env.connect_phase() : Monitor \rightarrow Scoreboard 연결
3. End of Elaboration Phase (정교화 완료)
 - └── 모든 연결 상태 최종 확인
4. Start of Simulation Phase (시뮬레이션 시작)
 - └── 초기화 작업 완료
5. Run Phase (실행 단계) - 메인 테스트
 - └── test.run_phase() : Sequence 실행 제어
 - └── driver.run_phase() : Transaction \rightarrow Hardware 변환
 - └── monitor.run_phase() : Hardware \rightarrow Transaction 수집
 - └── scoreboard.write() : 결과 검증 수행

6. Extract, Check, Report Phase (결과 처리)

└ 통계 수집, 최종 결과 분석

데이터 플로우 추적

하나의 Transaction이 시스템을 통과하는 전체 과정을 따라가 보겠습니다:

Complete Data Flow Trace:

Step 1: Transaction 생성 (Sequence)

```
generator.body()  
t.randomize()  
start_item(t)  
finish_item(t)
```

→ a=5, b=3 생성

Step 2: Transaction 전달 (Sequencer → Driver)

```
Driver  
get_next_item(tc)  
aif.a <= tc.a  
aif.b <= tc.b  
item_done()
```

→ tc.a=5, tc.b=3 받음

→ Interface에 5 출력

→ Interface에 3 출력

Step 3: 하드웨어 처리 (DUT)

```
DUT (add module)  
y = a + b
```

→ 5 + 3 = 8 계산

→ aif.y에 8 출력

Step 4: 결과 수집 (Monitor)

Monitor	
t.a = aif.a	→ 5 읽기
t.b = aif.b	→ 3 읽기
t.y = aif.y	→ 8 읽기
send.write(t)	→ Scoreboard로 전송



Step 5: 결과 검증 (Scoreboard)

Scoreboard	
write(transaction)	→ a=5, b=3, y=8 받음
if(y == a + b)	→ 8 == 5+3 ? YES
"Test Passed"	→ 성공 메시지 출력

시뮬레이션 로그 분석

실제 시뮬레이션을 실행했을 때 나타나는 로그 메시지들을 단계별로 분석해보겠습니다:

시뮬레이션 실행 로그 예시:

```
# UVM_INFO @ 0: reporter [RNTST] Running test test...
# UVM_INFO @ 0: uvm_test_top [GEN] Data send to Driver a :5 , b :3
# UVM_INFO @ 0: uvm_test_top.e.a.d [DRV] Trigger DUT a: 5 ,b : 3
# UVM_INFO @ 10: uvm_test_top.e.a.m [MON] Data send to Scoreboard
a : 5 , b : 3 and y : 8
# UVM_INFO @ 10: uvm_test_top.e.s [SCO] Test Passed

# UVM_INFO @ 10: uvm_test_top [GEN] Data send to Driver a :12 , b :7
# UVM_INFO @ 10: uvm_test_top.e.a.d [DRV] Trigger DUT a: 12 ,b : 7
# UVM_INFO @ 20: uvm_test_top.e.a.m [MON] Data send to Scoreboard
a : 12 , b : 7 and y : 19
# UVM_INFO @ 20: uvm_test_top.e.s [SCO] Test Passed

... (8 번 더 반복)
```

```

# UVM_INFO @ 100: uvm_test_top [GEN] Data send to Driver a :0 ,
b :15
# UVM_INFO @ 100: uvm_test_top.e.a.d [DRV] Trigger DUT a: 0 ,b : 15
# UVM_INFO @ 110: uvm_test_top.e.a.m [MON] Data send to Scoreboard
a : 0 , b : 15 and y : 15
# UVM_INFO @ 110: uvm_test_top.e.s [SCO] Test Passed

# UVM_INFO @ 160: reporter [TEST_DONE] Test completed successfully

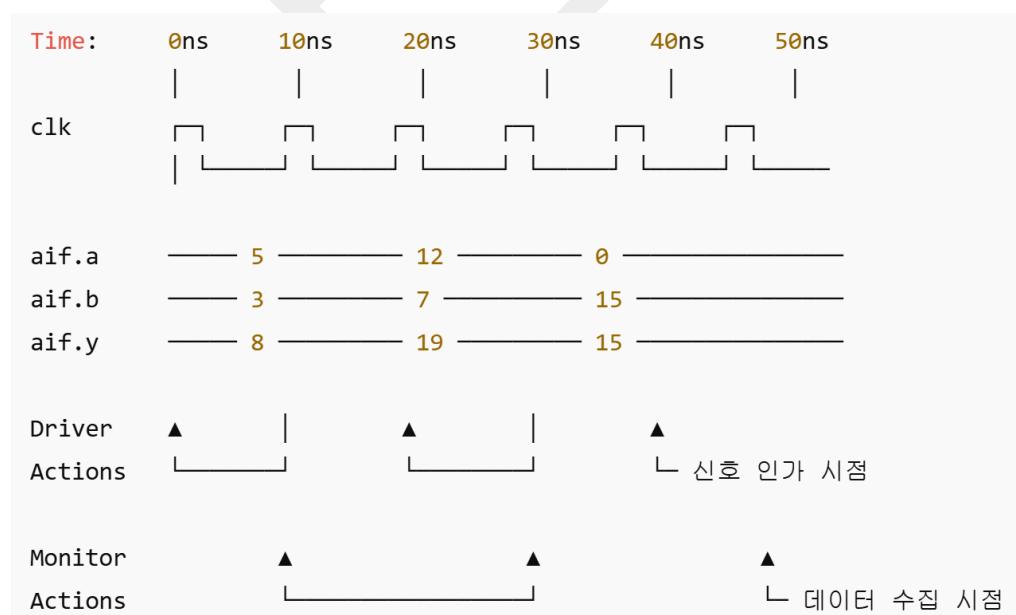
```

로그 분석 포인트:

시간	컴포넌트	메시지	의미
@ 0	GEN	Data send to Driver	Sequence에서 Transaction 생성
@ 0	DRV	Trigger DUT	Driver가 하드웨어에 신호 인가
@ 10	MON	Data send to Scoreboard	Monitor가 결과 수집 (10ns 지연 후)
@ 10	SCO	Test Passed	Scoreboard에서 검증 완료

타이밍 다이어그램

시뮬레이션 중 신호들의 시간별 변화를 시각적으로 표현하면:



실무에서의 검증 시나리오

실제 프로젝트에서는 다음과 같은 다양한 검증 시나리오를 수행합니다:

1. Smoke Test (연기 테스트):

```
class smoke_test extends test;
    virtual task run_phase(uvm_phase phase);
        phase.raise_objection(this);

        // 기본 기능만 빠르게 확인
        basic_sequence.start(e.a.seqr);
        #100;

        phase.drop_objection(this);
    endtask
endclass
```

용도: 기본적인 기능이 작동하는지 빠르게 확인

실행 시간: 보통 수 초 내외

목표: "연기가 나지 않는지" 확인 (기본 동작 확인)

2. Regression Test (회귀 테스트):

```
class regression_test extends test;
    virtual task run_phase(uvm_phase phase);
        phase.raise_objection(this);

        // 모든 기본 기능 테스트
        basic_sequence.start(e.a.seqr);
        #1000;

        // 경계값 테스트
        boundary_sequence.start(e.a.seqr);
        #1000;

        // 랜덤 테스트
        random_sequence.start(e.a.seqr);
        #5000;
```

```

    phase.drop_objection(this);
endtask
endclass

```

용도: 변경사항이 기존 기능에 영향을 주지 않는지 확인

실행 시간: 수 분에서 수 시간

목표: 모든 기존 기능의 정상 동작 보장

3. Stress Test (스트레스 테스트):

```

class stress_test extends test;
    virtual task run_phase(uvm_phase phase);
        phase.raise_objection(this);

        // 높은 빈도로 대량의 Transaction 실행
        high_frequency_sequence.num_trans = 100000;
        high_frequency_sequence.start(e.a.seqr);

        // 타임아웃 설정 (1초)
        #1000000;

        phase.drop_objection(this);
    endtask
endclass

```

용도: 극한 상황에서의 안정성 확인

실행 시간: 수 시간에서 수 일

목표: 메모리 누수, 성능 저하 등 장시간 운용 문제 발견

검증 메트릭과 분석

1. Functional Coverage (기능적 커버리지):

```

covergroup adder_coverage;
    a_values: coverpoint tr.a {
        bins low = {[0:7]};
        bins high = {[8:15]};
        bins corners = {0, 15};
    }

```

```

b_values: coverpoint tr.b {
    bins low = {[0:7]};
    bins high = {[8:15]};
    bins corners = {0, 15};
}

results: coverpoint tr.y {
    bins small = {[0:15]};
    bins large = {[16:30]};
    bins overflow = {31}; // 불가능하지만 확인용
}

// Cross Coverage
operand_cross: cross a_values, b_values;
endgroup

```

커버리지 목표:

- 모든 입력 조합의 95% 이상 테스트
- 모든 경계값 100% 테스트
- Cross coverage 90% 이상

2. Assertion Coverage (어서션 커버리지):

```

// DUT에 바인딩되는 어서션들
module adder_assertions(
    input logic [3:0] a, b,
    input logic [4:0] y
);

// 기본 기능 어서션
property correct_addition;
    ##1 (y == (a + b));
endproperty
assert_correct_addition: assert property (correct_addition);

// 오버플로우 어서션
property no_overflow;
    ##1 (y <= 30); // 최대값 15+15=30

```

```

endproperty
assert_no_overflow: assert property (no_overflow);

// 타이밍 어서션 (조합회로이므로 즉시 응답)
property immediate_response;
    ##0 (y == (a + b));
endproperty
assert_immediate_response: assert property (immediate_response);

endmodule

```

디버깅과 문제 해결

1. 일반적인 문제들과 해결책:

문제 증상	가능한 원인	해결 방법
시뮬레이션이 멈춤	Objection 미해제	phase.drop_objection() 확인
Transaction 이 전달되지 않음	Interface 연결 실패	Config_db 설정 확인
결과가 Scoreboard에 도달하지 않음	TLM 연결 누락	connect_phase() 확인
랜덤값이 생성되지 않음	randomize() 호출 누락	Sequence에서 t.randomize() 확인
VCD 파일이 비어있음	\$dumpvars 누락	Testbench top에서 덤프 설정 확인

2. 시스템적 디버깅 방법:

```

// 디버깅용 Enhanced Scoreboard
class debug_scoreboard extends scoreboard;
    int transaction_count = 0;
    int pass_count = 0;
    int fail_count = 0;

    virtual function void write(input transaction t);
        transaction_count++;

```

```

`uvm_info("DEBUG", $sformatf("Transaction #%0d: a=%0d, b=%0d,
y=%0d",
                               transaction_count, t.a, t.b, t.y),
UVM_HIGH);

if(t.y == t.a + t.b) begin
    pass_count++;
    `uvm_info("SCO", $sformatf("PASS [%0d]: %0d + %0d = %0d",
                               transaction_count, t.a, t.b, t.y),
UVM_MEDIUM);
end else begin
    fail_count++;
    `uvm_error("SCO", $sformatf("FAIL [%0d]: %0d + %0d ≠ %0d
(expected %0d)",
                               transaction_count, t.a, t.b, t.y,
t.a + t.b));

```

// 상세 분석 정보

```

`uvm_info("DEBUG", $sformatf("Binary: a=4'b%04b,
b=4'b%04b, y=5'b%05b",
                               t.a, t.b, t.y), UVM_HIGH);
end

// 주기적 통계 출력
if(transaction_count % 100 == 0) begin
    real pass_rate = 100.0 * pass_count / transaction_count;
    `uvm_info("STATS", $sformatf("Progress: %0d trans, %.1f%
pass rate",
                               transaction_count, pass_rate),
UVM_LOW);
end
endfunction

virtual function void report_phase(uvm_phase phase);
    `uvm_info("FINAL_REPORT", "==== Final Test Results ===",
UVM_NONE);
    `uvm_info("FINAL_REPORT", $sformatf("Total
Transactions: %0d", transaction_count), UVM_NONE);
    `uvm_info("FINAL_REPORT", $sformatf("Passed: %0d",
pass_count), UVM_NONE);

```

```

`uvm_info("FINAL_REPORT", $sformatf("Failed: %0d",
fail_count), UVM_NONE);

if(fail_count == 0) begin
    `uvm_info("FINAL_REPORT", "*** ALL TESTS PASSED ***",
UVM_NONE);
end else begin
    `uvm_error("FINAL_REPORT", $sformatf("*** %0d TESTS FAILED
***", fail_count));
end
endfunction
endclass

```

3. 파형 기반 디버깅:

VCD 파일을 GTKWave 나 다른 파형 뷰어로 열면 다음과 같은 정보를 확인할 수 있습니다:

파형 분석 체크리스트:

- 입력 신호(a, b)가 예상한 시점에 변경되는가?
- 출력 신호(y)가 적절한 지연 후 올바른 값으로 변경되는가?
- Interface 신호들이 올바르게 연결되어 있는가?
- 클럭이나 리셋 신호가 정상적으로 작동하는가?
- 타이밍 위반이나 셋업/홀드 타임 문제는 없는가?

성능 최적화

1. 시뮬레이션 속도 개선:

```

// 효율적인 Monitor 구현
class optimized_monitor extends monitor;
    // 불필요한 로그 메시지 줄이기
    virtual task run_phase(uvm_phase phase);
        forever begin
            #10;
            t.a = aif.a;
            t.b = aif.b;
            t.y = aif.y;
        end
    endtask
endclass

```

```

// 디버그 레벨에서만 상세 로그 출력
`uvm_info("MON", $sformatf("a=%0d, b=%0d, y=%0d", t.a,
t.b, t.y), UVM_HIGH);

    send.write(t);
end
endtask
endclass

// 효율적인 Sequence 구현
class fast_sequence extends uvm_sequence #(transaction);
    // Transaction 객체 재사용으로 메모리 할당 오버헤드 감소
    transaction reusable_trans;

    virtual task body();
        reusable_trans =
transaction::type_id::create("reusable_trans");

        repeat(num_trans) begin
            start_item(reusable_trans);
            reusable_trans.randomize();
            finish_item(reusable_trans);
        end
    endtask
endclass

```

2. 메모리 사용량 최적화:

```

// 큰 데이터셋 처리를 위한 스트리밍 방식
class streaming_scoreboard extends scoreboard;
    // 모든 Transaction을 메모리에 저장하지 않고 즉시 처리
    virtual function void write(input transaction t);
        // 즉시 검증하고 결과만 저장
        if(t.y == t.a + t.b) begin
            pass_count++;
        end else begin
            fail_count++;
            // 실패한 케이스만 기록
            failed_cases.push_back(t);
        end
    endfunction
endclass

```

```

    end
    // Transaction 객체는 자동으로 가비지 컬렉션됨
endfunction
endclass

```

실무 프로젝트 구조

실제 프로젝트에서는 다음과 같은 디렉토리 구조를 사용합니다:

```

project_verification/
└── src/
    ├── rtl/
    │   ├── add.sv                                # DUT 소스
    │   └── add_if.sv                             # Interface 정의
    └── tb/
        ├── sequences/
        │   ├── base_sequence.sv
        │   ├── random_sequence.sv
        │   └── directed_sequence.sv
        ├── agents/
        │   ├── add_agent.sv
        │   ├── add_driver.sv
        │   └── add_monitor.sv
        ├── env/
        │   ├── add_env.sv
        │   └── add_scoreboard.sv
        ├── tests/
        │   ├── base_test.sv
        │   ├── smoke_test.sv
        │   └── regression_test.sv
        └── tb_top.sv
    └── sim/
        ├── Makefile
        ├── run_tests.py                            # 테스트 실행 스크립트
        └── regression.cfg                         # 회귀 테스트 설정
    └── docs/
        ├── test_plan.md                           # 테스트 계획서
        └── coverage_report.html                   # 커버리지 리포트
    └── results/
        ├── logs/                                  # 시뮬레이션 로그
        ├── waves/                                 # VCD 파일들
        └── coverage/                             # 커버리지 데이터

```

Makefile 예시

```
# 기본 설정
SIMULATOR = questasim
TB_TOP = add_tb
DUT_FILES = ../src/rtl/add.sv ../src/rtl/add_if.sv
TB_FILES = ../src/tb/**/*.sv

# 기본 테스트 실행
smoke:
    $(SIMULATOR) +TEST=smoke_test $(DUT_FILES) $(TB_FILES)

# 회귀 테스트 실행
regression:
    python3 run_tests.py --test_list regression.cfg

# 커버리지 포함 실행
coverage:
    $(SIMULATOR) +TEST=regression_test -coverage $(DUT_FILES)
$(TB_FILES)
    # 커버리지 리포트 생성
    vcover report -html coverage_report.html

# 디버그 모드 (파형 덤프 포함)
debug:
    $(SIMULATOR) +TEST=debug_test +VERBOSITY=UVM_HIGH
+DUMP=debug.vcd $(DUT_FILES) $(TB_FILES)

# 정리
clean:
    rm -rf work/ transcript vsim.wlf *.vcf *.log
```

지속적 통합(CI) 연동

```
# run_tests.py - 자동화된 테스트 실행 스크립트
import subprocess
import json
import sys
from datetime import datetime
```

```

class TestRunner:
    def __init__(self):
        self.results = []

    def run_test(self, test_name, timeout=300):
        """개별 테스트 실행"""
        cmd = f"make {test_name}"
        start_time = datetime.now()

        try:
            result = subprocess.run(cmd, shell=True, timeout=timeout,
                                   capture_output=True, text=True)
            end_time = datetime.now()
            duration = (end_time - start_time).total_seconds()

            test_result = {
                "name": test_name,
                "status": "PASS" if result.returncode == 0 else
"FAIL",
                "duration": duration,
                "stdout": result.stdout,
                "stderr": result.stderr
            }

            self.results.append(test_result)
            return test_result

        except subprocess.TimeoutExpired:
            test_result = {
                "name": test_name,
                "status": "TIMEOUT",
                "duration": timeout,
                "stdout": "",
                "stderr": "Test timed out"
            }
            self.results.append(test_result)
            return test_result

    def run_regression(self, config_file):

```

```

"""회귀 테스트 실행"""
with open(config_file, 'r') as f:
    tests = f.readlines()

for test in tests:
    if test.strip() and not test.startswith('#'):
        print(f"Running {test}...")
        result = self.run_test(test)
        print(f"  Result: {result['status']}")
({result['duration']:.1f}s)")

def generate_report(self):
    """테스트 결과 리포트 생성"""
    total_tests = len(self.results)
    passed_tests = len([r for r in self.results if r['status'] == 'PASS'])

    report = {
        "summary": {
            "total": total_tests,
            "passed": passed_tests,
            "failed": total_tests - passed_tests,
            "pass_rate": 100.0 * passed_tests / total_tests if
total_tests > 0 else 0
        },
        "tests": self.results,
        "timestamp": datetime.now().isoformat()
    }

    with open('test_report.json', 'w') as f:
        json.dump(report, f, indent=2)

    return report

if __name__ == "__main__":
    runner = TestRunner()
    runner.run_regression("regression.cfg")
    report = runner.generate_report()

    print("\n==== Test Summary ====")

```

```
print(f"Total Tests: {report['summary']['total']}")  
print(f"Passed: {report['summary']['passed']}")  
print(f"Failed: {report['summary']['failed']}")  
print(f"Pass Rate: {report['summary']['pass_rate']:.1f}%")  
  
# CI 시스템을 위한 종료 코드  
sys.exit(0 if report['summary']['failed'] == 0 else 1)
```



13. 이해도 확인 퀴즈

문제 1: UVM Transaction의 기본 개념

당신은 간단한 4비트 AND 게이트 검증을 위한 Transaction을 설계하고 있습니다. AND 게이트는 두 개의 4비트 입력을 받아 하나의 4비트 출력을 생성합니다. 다음 코드에서 어떤 필드에 rand 키워드를 사용해야 할까요?

```
class and_transaction extends uvm_sequence_item;
    bit [3:0] input_a;
    bit [3:0] input_b;
    bit [3:0] output_result;

    `uvm_object_utils_begin(and_transaction)
        `uvm_field_int(input_a, UVM_DEFAULT)
        `uvm_field_int(input_b, UVM_DEFAULT)
        `uvm_field_int(output_result, UVM_DEFAULT)
    `uvm_object_utils_end
endclass
```

선택지:

1. 모든 필드에 rand 키워드 추가
2. input_a 와 input_b 에만 rand 키워드 추가
3. output_result 에만 rand 키워드 추가
4. rand 키워드가 전혀 필요하지 않음

정답: 2 번 (input_a 와 input_b 에만 rand 키워드 추가)

해설:

핵심 개념: Transaction Class에서 rand 키워드는 테스트 중 자동으로 랜덤 생성될 필드에만 사용합니다. DUT의 출력은 하드웨어가 계산하므로 랜덤 생성 대상이 아닙니다.

단계별 분석:

1. Transaction은 하드웨어에 전달할 입력 데이터와 받을 출력 데이터를 모두 포함합니다.
2. 입력 필드는 Sequence에서 randomize 호출 시 자동으로 값이 생성되어야 하므로 rand 키워드가 필요합니다.
3. 출력 필드는 DUT가 계산한 결과를 Monitor가 수집하여 저장하는 용도이므로 rand가 불필요합니다.
4. 만약 출력에도 rand를 사용하면 DUT의 실제 출력과 무관하게 랜덤값이 할당되어 검증이 불가능합니다.

코드 예시:

```
class and_transaction extends uvm_sequence_item;
    rand bit [3:0] input_a; // 랜덤 생성 필요
    rand bit [3:0] input_b; // 랜덤 생성 필요
    bit [3:0] output_result; // DUT 출력 저장용

    function new(string name = "and_transaction");
        super.new(name);
    endfunction

    `uvm_object_utils_begin(and_transaction)
        `uvm_field_int(input_a, UVM_DEFAULT)
        `uvm_field_int(input_b, UVM_DEFAULT)
        `uvm_field_int(output_result, UVM_DEFAULT)
    `uvm_object_utils_end
endclass
```

오답 분석:

- 1 번: 출력 필드에 rand를 사용하면 DUT의 실제 결과를 무시하고 랜덤값이 덮어써집니다.
- 3 번: 입력만 랜덤 생성해야 다양한 테스트 케이스를 자동으로 만들 수 있습니다.
- 4 번: rand 없이는 Sequence에서 수동으로 모든 값을 할당해야 하므로 자동화가 불가능합니다.

실무 팁: 실무에서는 입력 필드에 constraint를 추가하여 테스트하고 싶은 특정 범위나 패턴으로 랜덤값을 제한합니다. 예를 들어 corner case 테스트를 위해 "constraint valid_range { input_a inside {0, 15}; }"처럼 경계값 위주로 생성하도록 제약할 수 있습니다.

관련 문서 섹션: 3 장 UVM Transaction Class

문제 2: Sequence의 실행 프로토콜

UVM Sequence에서 Transaction을 Sequencer에 전달할 때 반드시 호출해야 하는 메서드의 올바른 순서는 무엇입니까?

```
virtual task body();
    transaction t;
    t = transaction::type_id::create("t");
    repeat(5) begin
        // 여기에 올바른 순서로 메서드를 배치해야 함
        t.randomize();
    end
endtask
```

선택지:

1. start_item(t) → randomize() → finish_item(t)
2. randomize() → start_item(t) → finish_item(t)
3. finish_item(t) → start_item(t) → randomize()
4. start_item(t) → finish_item(t) → randomize()

정답: 1번 (start_item(t) → randomize() → finish_item(t))

해설:

핵심 개념: UVM Sequence는 Sequencer와 handshake 프로토콜을 통해 Transaction을 전달합니다. start_item()으로 전송 시작을 알리고, randomize()로 데이터를 준비한 후, finish_item()으로 완료를 통보해야 합니다.

단계별 분석:

1. `start_item(t)` 호출 시 Sequencer에게 "Transaction을 보낼 준비가 되었습니다"라고 알립니다. Sequencer는 이전 Transaction 처리가 끝나면 준비 신호를 보냅니다.
2. Sequencer의 준비 신호를 받은 후에야 `randomize()`를 호출하여 실제 데이터를 생성합니다. 이 시점에서 rand 필드들에 값이 할당됩니다.
3. `finish_item(t)` 호출로 "Transaction 준비가 완료되었으니 가져가세요"라고 Sequencer에게 알립니다. 이후 Driver가 이 Transaction을 받아 처리합니다.
4. 이 순서를 지키지 않으면 Driver가 준비되지 않은 Transaction을 받거나, Sequencer와의 동기화가 깨져 시뮬레이션이 멈출 수 있습니다.

코드 예시:

```
virtual task body();
    transaction t;
    t = transaction::type_id::create("t");

    repeat(5) begin
        start_item(t);      // 1 단계: Sequencer에 시작 알림
        t.randomize();     // 2 단계: 데이터 생성
        `uvm_info("SEQ", $sformatf("Generated: a=%0d, b=%0d",
            t.a, t.b), UVM_MEDIUM)
        finish_item(t);    // 3 단계: Sequencer에 완료 알림
    end
endtask
```

오답 분석:

- 2 번: `start_item()` 없이 `randomize()`를 먼저 호출하면 Sequencer와의 동기화가 이루어지지 않아 Driver가 대기 상태에 빠질 수 있습니다.
- 3 번: `finish_item()`을 먼저 호출하면 논리적 순서가 완전히 뒤바뀌어 시뮬레이션이 정상 작동하지 않습니다.
- 4 번: `randomize()`를 `finish_item()` 다음에 호출하면 Sequencer에 이미 전달된 Transaction의 값이 변경되어 예측 불가능한 동작이 발생합니다.

실무 팁: 실무에서는 `start_item()`과 `finish_item()` 사이에 constraint 기반 randomization 뿐만 아니라 추가적인 데이터 설정이나 로깅을 수행합니다. 하지만 반드시 이 두 호출 사이에서만 Transaction을 수정해야 하며, `finish_item()` 이후에는 절대 수정하지 않아야 합니다.

관련 문서 섹션: 4 장 UVM Sequence Class

문제 3: Driver 의 Interface 연결

다음 Driver 코드에서 Interface를 올바르게 연결하기 위해 `build_phase`에 추가해야 하는 코드는 무엇입니까?

```
class driver extends uvm_driver #(transaction);
  `uvm_component_utils(driver)

  transaction tc;
  virtual add_if aif;

  function new(string name = "driver", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    tc = transaction::type_id::create("tc");
    // 여기에 Interface 연결 코드가 필요함
  endfunction
endclass
```

선택지:

1. `aif = new("aif");`
2. `uvm_config_db #(virtual add_if)::get(this, "", "aif", aif);`
3. `aif = add_if::type_id::create("aif", this);`
4. Interface 연결은 `connect_phase`에서 수행해야 함

정답: 2 번 (`uvm_config_db #(virtual add_if)::get(this, "", "aif", aif);`)

해설:

핵심 개념: Driver 는 하드웨어 Interface 에 접근하기 위해 UVM Configuration Database 에서 virtual interface 핸들을 가져와야 합니다. Interface 는 testbench top에서 생성되어 config_db 를 통해 전달됩니다.

단계별 분석:

1. Testbench top module에서 Interface 인스턴스가 생성되고 DUT 와 물리적으로 연결됩니다.
2. 이 Interface 를 UVM 컴포넌트들이 사용할 수 있도록 config_db 에 저장합니다.
3. Driver 의 build_phase에서 config_db로부터 Interface 핸들을 가져옵니다. get() 메서드가 실패하면 에러 처리가 필요합니다.
4. 이렇게 가져온 virtual interface 를 통해 Driver 는 run_phase에서 DUT 의 입력 신호를 제어할 수 있습니다.

코드 예시:

```
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    tc = transaction::type_id::create("tc");

    // Configuration Database에서 Interface 가져오기
    if(!uvm_config_db #(virtual add_if)::get(this, "", "aif", aif))
        `uvm_fatal("DRV", "Failed to get interface from config_db")
endfunction

// run_phase에서 Interface 사용
virtual task run_phase(uvm_phase phase);
    forever begin
        seq_item_port.get_next_item(tc);
        aif.a <= tc.a; // Interface 를 통해 DUT 제어
        aif.b <= tc.b;
        seq_item_port.item_done();
        #10;
    end
endtask
```

오답 분석:

- 1 번: Interface 는 testbench top 에서 이미 생성되었으므로 Driver 에서 new()로 생성할 수 없습니다.
- 3 번: Interface 는 uvm_component 가 아니므로 Factory pattern 으로 생성할 수 없습니다.
- 4 번: Interface 연결은 build_phase 에서 수행하고, connect_phase 는 TLM port 연결용입니다.

실무 팁: 실무에서는 get() 호출이 실패할 경우를 대비하여 반드시 에러 처리를 추가해야 합니다. `uvm_fatal() 을 사용하면 시뮬레이션이 즉시 중단되어 문제를 빠르게 파악할 수 있습니다. 또한 경로 문자열을 정확히 설정해야 하는데, testbench top 에서 set() 할 때 사용한 경로와 일치해야 합니다.

관련 문서 섹션: 5 장 UVM Driver Class

문제 4: Monitor 의 Analysis Port

Monitor 가 수집한 Transaction 을 Scoreboard 에 전달하기 위해 사용하는 TLM 통신 방식은 무엇이며, 그 특징은 무엇입니까?

선택지:

1. Sequence Item Port - 양방향 handshake 통신으로 응답을 기다림
2. Analysis Port - 단방향 broadcast 로 여러 수신자에게 동시 전송 가능
3. Blocking Get Port - 데이터가 준비될 때까지 블로킹
4. Direct Method Call - 직접 함수 호출로 데이터 전달

정답: 2 번 (Analysis Port - 단방향 broadcast 로 여러 수신자에게 동시 전송 가능)

해설:

핵심 개념: Monitor 는 Analysis Port 를 통해 관찰한 Transaction 을 Scoreboard 나 Coverage Collector 등 여러 컴포넌트에 동시에 전달할 수 있습니다. 이는 단방향이며 비블로킹 방식으로 작동합니다.

단계별 분석:

1. Monitor의 run_phase에서 DUT 신호를 관찰하고 Transaction 객체에 저장합니다.
2. send.write(transaction) 호출로 Analysis Port를 통해 데이터를 전송합니다.
3. 이 Analysis Port에 연결된 모든 Scoreboard의 write() 함수가 자동으로 호출됩니다.
4. Monitor는 전송 후 응답을 기다리지 않고 즉시 다음 관찰을 계속하므로 성능이 우수합니다.

코드 예시:

```
// Monitor에서 Analysis Port 사용
class monitor extends uvm_monitor;
    uvm_analysis_port #(transaction) send; // 선언

    function new(string name, uvm_component parent);
        super.new(name, parent);
        send = new("send", this); // 생성
    endfunction

    virtual task run_phase(uvm_phase phase);
        forever begin
            #10;
            t.a = aif.a;
            t.b = aif.b;
            t.y = aif.y;
            send.write(t); // 전송 - 비블로킹
        end
    endtask
endclass

// Scoreboard에서 수신
class scoreboard extends uvm_scoreboard;
    uvm_analysis_imp #(transaction, scoreboard) recv;
```

```

virtual function void write(transaction t);
    // 자동으로 호출됨
    if(t.y == t.a + t.b)
        `uvm_info("SCO", "Test Passed", UVM_NONE)
endfunction
endclass

```

오답 분석:

- 1 번: Sequence Item Port 는 Driver 와 Sequencer 간 양방향 통신용이며 handshake 가 필요합니다.
- 3 번: Blocking Get Port 는 Producer-Consumer 패턴에서 사용되며 데이터를 기다립니다.
- 4 번: 직접 함수 호출은 컴포넌트 간 결합도를 높여 재사용성이 떨어집니다.

실무 팁: 실무에서는 하나의 Monitor 가 여러 Scoreboard, Coverage Collector, Debug Logger 등에 동시에 데이터를 전송하는 경우가 많습니다. Analysis Port 의 broadcast 특성 덕분에 Monitor 코드 수정 없이 새로운 수신자를 추가할 수 있어 확장성이 뛰어납니다.

관련 문서 섹션: 6 장 UVM Monitor Class

문제 5: Scoreboard 의 검증 로직

다음 Scoreboard 코드에서 오버플로우가 발생하는 경우를 올바르게 검증하는 로직은 무엇입니까? 4 비트 입력 두 개를 더하는 덧셈기이며 출력은 5 비트입니다.

```

virtual function void write(input transaction t);
    // 여기에 검증 로직 작성
endfunction

```

선택지:

1. if(t.y == t.a + t.b) pass; else fail;
2. if(t.y == ((t.a + t.b) % 32)) pass; else fail;
3. bit [4:0] expected = t.a + t.b; if(t.y == expected) pass; else fail;

```
4. int expected = t.a + t.b; if(t.y == expected[4:0]) pass; else fail;
```

정답: 3 번 (bit [4:0] expected = t.a + t.b; if(t.y == expected) pass; else fail;)

해설:

핵심 개념: Scoreboard 는 DUT 와 동일한 비트 폭으로 예상값을 계산해야 합니다. DUT 가 5 비트 출력을 생성하므로 참조 모델도 5 비트로 계산하여 오버플로우를 정확히 재현해야 합니다.

단계별 분석:

1. DUT 는 4 비트 입력 두 개를 5 비트 출력으로 더합니다. 최대값은 $15+15=30$ 이므로 5 비트로 충분합니다.
2. 예상값 계산 시 int 나 32 비트를 사용하면 DUT 와 다른 결과가 나올 수 있습니다.
3. bit [4:0]로 선언하면 자동으로 5 비트로 제한되어 DUT 와 동일한 동작을 합니다.
4. 이렇게 하면 오버플로우 상황에서도 DUT 와 참조 모델의 결과가 일치하게 됩니다.

코드 예시:

```
virtual function void write(input transaction t);
    bit [4:0] expected;
    expected = t.a + t.b; // 5 비트로 제한된 덧셈

    if(t.y == expected) begin
        pass_count++;
        `uvm_info("SCO", $sformatf("PASS: %0d + %0d = %0d",
            t.a, t.b, t.y), UVM_MEDIUM)
    end else begin
        fail_count++;
        `uvm_error("SCO", $sformatf("FAIL: %0d + %0d = %0d,
            expected %0d",
            t.a, t.b, t.y, expected))
    end
endfunction
```

```

// 경계값 추가 검증
if(t.a == 15 && t.b == 15) begin
    `uvm_info("SCO", "Corner case verified: 15+15=30", UVM_LOW)
end
endfunction

```

오답 분석:

- 1 번: t.a + t.b 는 SystemVerilog에서 자동으로 32 비트로 계산되어 비교가 정확하지 않을 수 있습니다.
- 2 번: 모듈로 연산은 불필요하며, 5 비트 출력은 자동으로 31 까지만 표현 가능합니다.
- 4 번: int 는 32 비트이므로 비트 슬라이싱이 필요하지만, 처음부터 5 비트로 계산하는 것이 더 명확합니다.

실무 팁: 실무에서는 단순 비교뿐만 아니라 특정 조건에서 추가 검증을 수행합니다. 예를 들어 carry bit 가 발생하는 경우나 최대값 입력 시 별도 로그를 남겨 corner case 가 테스트되었음을 확인합니다.

관련 문서 섹션: 7 장 UVM Scoreboard Class

문제 6: Agent 의 Active vs Passive 모드

다음 중 Agent 를 Passive 모드로 설정해야 하는 상황은 언제입니까?

선택지:

1. DUT 의 입력 포트를 제어해야 할 때
2. DUT 의 출력 포트만 관찰해야 할 때
3. Sequence 를 실행해야 할 때
4. Driver 를 통해 신호를 생성해야 할 때

정답: 2 번 (DUT 의 출력 포트만 관찰해야 할 때)

해설:

핵심 개념: Active Agent 는 Driver, Monitor, Sequencer 를 모두 포함하여 신호를 생성하고 관찰합니다. Passive Agent 는 Monitor 만 포함하여 신호를 관찰만 합니다. DUT 출력을 모니터링할 때는 Passive 모드를 사용합니다.

단계별 분석:

1. Active 모드는 DUT에 자극을 제공해야 하는 입력 인터페이스에 사용됩니다. Driver가 Transaction을 받아 하드웨어 신호로 변환합니다.
2. Passive 모드는 DUT의 출력이나 내부 신호를 관찰만 하는 경우에 사용됩니다. Monitor만으로 충분합니다.
3. 예를 들어 CPU 인터페이스는 Active Agent로, 메모리 응답 인터페이스는 Passive Agent로 구성할 수 있습니다.
4. Passive Agent는 신호를 변경하지 않으므로 DUT 동작에 영향을 주지 않고 안전하게 관찰할 수 있습니다.

코드 예시:

```
class configurable_agent extends uvm_agent;
  uvm_active_passive_enum is_active;
  monitor m;
  driver d;
  uvm_sequencer #(transaction) seqr;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    m = monitor::type_id::create("m", this); // 항상 생성

    if(is_active == UVM_ACTIVE) begin
      d = driver::type_id::create("d", this);
      seqr = uvm_sequencer #(transaction)::type_id::create("seqr",
      this);
    end
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    if(is_active == UVM_ACTIVE) begin
      d.seq_item_port.connect(seqr.seq_item_export);
    end
  endfunction
endclass

// Testbench top에서 설정
initial begin
```

```

uvm_config_db #(uvm_active_passive_enum)::set(null,
    "*.input_agent", "is_active", UVM_ACTIVE); // 입력용
uvm_config_db #(uvm_active_passive_enum)::set(null,
    "*.output_agent", "is_active", UVM_PASSIVE); // 출력용
end

```

오답 분석:

- 1 번, 3 번, 4 번: 모두 Active 모드가 필요한 상황들입니다. 신호 생성이나 제어가 필요하면 Driver 와 Sequencer 가 포함된 Active 모드를 사용해야 합니다.

실무 팁: 실무에서는 하나의 검증 환경에 여러 Agent 가 있을 때 일부는 Active 로, 일부는 Passive 로 설정합니다. 예를 들어 PCIe Root Complex 를 검증할 때 Upstream 은 Active, Downstream 은 Passive 로 구성할 수 있습니다.

관련 문서 섹션: 8 장 UVM Agent Class

문제 7: Environment 의 Connect Phase

Environment 에서 Agent 의 Monitor 를 Scoreboard 에 연결하는 올바른 코드는 무엇입니까?

```

class env extends uvm_env;
    agent a;
    scoreboard s;

    virtual function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        // 여기에 연결 코드 작성
    endfunction
endclass

```

선택지:

1. a.m.send = s.recv;
2. a.m.send.connect(s.recv);
3. s.recv.connect(a.m.send);
4. connect(a.m.send, s.recv);

정답: 2 번 (a.m.send.connect(s.recv);)

해설:

핵심 개념: TLM port 연결은 Analysis Port의 connect() 메서드를 호출하여 Analysis Implementation Port를 인자로 전달합니다. 연결 방향은 송신자에서 수신자로 향합니다.

단계별 분석:

1. Agent 내부의 Monitor(a.m)는 Analysis Port(send)를 가지고 있습니다.
2. Scoreboard(s)는 Analysis Implementation Port(recv)를 가지고 있습니다.
3. Port의 connect() 메서드를 호출하면 UVM이 자동으로 write() 함수를 연결합니다.
4. 이후 Monitor에서 send.write(t)를 호출하면 Scoreboard의 write(t)가 자동 실행됩니다.

코드 예시:

```
class env extends uvm_env;
  `uvm_component_utils(env)

  agent a;
  scoreboard s;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    a = agent::type_id::create("a", this);
    s = scoreboard::type_id::create("s", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    // Monitor의 Analysis Port를 Scoreboard의 Imp에 연결
    a.m.send.connect(s.recv);

    // 추가로 Coverage Collector가 있다면
    // a.m.send.connect(cov_collector.analysis_export);
  endfunction
```

```
endclass

// 데이터 흐름
// Monitor: send.write(t) → Scoreboard: write(t) 자동 호출
```

오답 분석:

- 1 번: 단순 할당은 TLM 연결이 아니며 작동하지 않습니다.
- 3 번: 연결 방향이 반대입니다. Analysis Imp 는 connect() 메서드가 없습니다.
- 4 번: SystemVerilog 에 이런 전역 connect() 함수는 없습니다.

실무 팁: 하나의 Monitor 를 여러 Scoreboard 나 Collector 에 연결할 수 있습니다. Analysis Port 는 broadcast 방식이므로 connect() 를 여러 번 호출하여 다중 연결이 가능합니다. 이를 통해 기능 검증, 커버리지 수집, 성능 분석을 동시에 수행할 수 있습니다.

관련 문서 섹션: 9 장 UVM Environment Class

문제 8: Sequence 실행 제어

Test에서 Sequence 를 실행할 때 사용하는 objection 메커니즘의 목적은 무엇입니까? 다음 코드에서 objection 을 올바르게 사용한 것은 무엇입니까?

```
virtual task run_phase(uvm_phase phase);
    // 옵션 A
    gen.start(e.a.seqr);
    #100;

    // 옵션 B
    phase.raise_objection(this);
    gen.start(e.a.seqr);
    #100;
    phase.drop_objection(this);

    // 옵션 C
    phase.raise_objection(this);
    #100;
    phase.drop_objection(this);
```

```

gen.start(e.a.seqr);

// 옵션 D
phase.drop_objection(this);
gen.start(e.a.seqr);
#100;
phase.raise_objection(this);
endtask

```

선택지:

1. 옵션 A - objection 없이 실행
2. 옵션 B - raise 후 테스트 실행하고 drop
3. 옵션 C - raise/drop 후 테스트 실행
4. 옵션 D - drop 먼저 하고 raise 나중에

정답: 2 번 (옵션 B - raise 후 테스트 실행하고 drop)

해설:

핵심 개념: Objection은 UVM에게 "시뮬레이션을 종료하지 말고 기다려라"고 알리는 메커니즘입니다. raise_objection()으로 테스트 시작을 선언하고, 모든 작업 완료 후 drop_objection()으로 종료를 알립니다.

단계별 분석:

1. run_phase 시작 시 raise_objection(this)를 호출하여 "이 컴포넌트가 작업 중"임을 UVM에 알립니다.
2. Sequence 실행과 필요한 대기 시간을 포함한 모든 테스트 로직을 수행합니다.
3. 테스트가 완전히 끝난 후 drop_objection(this)를 호출하여 "작업 완료"를 알립니다.
4. 모든 컴포넌트의 objection이 해제되면 UVM이 자동으로 시뮬레이션을 종료합니다.

코드 예시:

```

virtual task run_phase(uvm_phase phase);
  phase.raise_objection(this);
  `uvm_info("TEST", "Starting test execution", UVM_LOW)

```

```

// 메인 테스트 로직
gen.start(e.a.seqr);

// 모든 Transaction 처리 대기
#100;

// 최종 결과 확인 시간
#50;

`uvm_info("TEST", "Test execution completed", UVM_LOW)
phase.drop_objection(this);
endtask

// 복잡한 시나리오 - 여러 처리 포함
virtual task run_phase(uvm_phase phase);
phase.raise_objection(this);

fork
begin
gen.start(e.a.seqr);
end
begin
#10000;
`uvm_error("TEST", "Timeout occurred")
end
join_any
 disable fork;

phase.drop_objection(this); // 항상 실행되도록 보장
endtask

```

오답 분석:

- 1 번: objection 없으면 UVM이 즉시 시뮬레이션을 종료하려고 시도하여 테스트가 제대로 실행되지 않습니다.
- 3 번: drop_objection()을 너무 일찍 호출하면 Sequence 가 실행되기 전에 시뮬레이션이 종료됩니다.
- 4 번: drop 을 먼저 호출하는 것은 논리적으로 맞지 않으며, raise 가 없는 drop 은 warning 을 발생시킵니다.

실무 팁: 실무에서는 여러 컴포넌트가 동시에 objection을 raise 할 수 있습니다. UVM은 모든 objection이 drop될 때까지 기다리므로, 복잡한 테스트 시나리오에서도 안전하게 종료 시점을 제어할 수 있습니다. fork-join 구조에서는 반드시 모든 경로에서 drop이 호출되도록 해야 합니다.

관련 문서 섹션: 10장 UVM Test Class

문제 9: UVM Configuration Database

다음 중 Configuration Database를 사용하여 Interface를 전달하는 올바른 방법은 무엇입니까? Testbench top에서 Agent의 Driver와 Monitor에게 Interface를 전달하려고 합니다.

```
module testbench_top;
    add_if aif();
    add dut(.a(aif.a), .b(aif.b), .y(aif.y));

    initial begin
        // 여기에 config_db 설정 코드
        run_test("test");
    end
endmodule
```

선택지:

1. uvm_config_db #(add_if)::set(null, "*", "aif", aif);
2. uvm_config_db #(virtual add_if)::set(null,
 "uvm_test_top.e.a*", "aif", aif);
3. uvm_config_db #(add_if*)::set(this, "*", "aif", aif);
4. set_config_db("aif", aif);

정답: 2번 (uvm_config_db #(virtual add_if)::set(null,
"uvm_test_top.e.a*", "aif", aif);)

해설:

핵심 개념: Configuration Database는 testbench top의 하드웨어 Interface를 UVM 컴포넌트에 전달하기 위해 virtual interface 타입을 사용하며, 정확한 계층 경로와 와일드카드를 지정해야 합니다.

단계별 분석:

1. testbench top 은 SystemVerilog 모듈이므로 첫 번째 인자는 null 을 사용합니다.
2. 두 번째 인자는 Interface 를 받을 컴포넌트의 경로입니다.
"uvm_test_top.e.a*"는 Test 내부의 Environment(e), Agent(a)와 그 하위 모든 컴포넌트를 의미합니다.
3. virtual interface 타입을 사용해야 클래스 멤버 변수로 저장할 수 있습니다.
4. 세 번째 인자 "aif"는 get() 할 때 사용할 키 이름입니다.

코드 예시:

```
module testbench_top;
    logic clk = 0;
    always #5 clk = ~clk;

    add_if aif(clk);
    add dut(.clk(clk), .a(aif.a), .b(aif.b), .y(aif.y));

    initial begin
        // Interface 를 UVM 환경에 전달
        uvm_config_db #(virtual add_if)::set(null,
            "uvm_test_top.e.a*", "aif", aif);

        // 추가 설정들
        uvm_config_db #(int)::set(null, "uvm_test_top.*",
            "timeout_cycles", 1000);

        run_test("test");
    end

    // VCD 덤프
    initial begin
        $dumpfile("simulation.vcd");
        $dumpvars(0, testbench_top);
    end
endmodule
```

```

// Driver에서 사용
class driver extends uvm_driver #(transaction);
    virtual add_if aif;

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        if(!uvm_config_db #(virtual add_if)::get(this, "", "aif", aif))
            `uvm_fatal("DRV", "Cannot get interface")
    endfunction
endclass

```

오답 분석:

- 1 번: `virtual` 키워드가 없으면 클래스에서 사용할 수 없으며, 경로가 너무 광범위합니다.
- 3 번: `testbench top`은 `uvm_component`가 아니므로 `this`를 사용할 수 없고, 포인터 타입도 올바르지 않습니다.
- 4 번: 이런 간소화된 함수는 UVM에 존재하지 않습니다.

실무 팁: 실무에서는 여러 Interface가 있을 때 각각 다른 키 이름과 경로를 사용하여 정확히 전달합니다. 예를 들어 "`cpu_if`", "`mem_if`", "`pcie_if`" 등으로 구분하고, 경로도 "`.cpu_agent`", "`.mem_agent`"처럼 명확히 지정하여 잘못된 컴포넌트에 전달되는 것을 방지합니다.

관련 문서 섹션: 11 장 Testbench Top Module

문제 10: TLM 통신의 종류

UVM에서 Driver와 Sequencer 간 통신에 사용되는 TLM Port 타입과 Monitor와 Scoreboard 간 통신에 사용되는 TLM Port 타입을 올바르게 짹지은 것은 무엇입니까?

선택지:

1. Driver-Sequencer: Analysis Port, Monitor-Scoreboard: Sequence Item Port
2. Driver-Sequencer: Sequence Item Port, Monitor-Scoreboard: Analysis Port
3. Driver-Sequencer: Blocking Get Port, Monitor-Scoreboard: Non-blocking Put Port
4. 모두 Analysis Port를 사용

정답: 2번 (Driver-Sequencer: Sequence Item Port, Monitor-Scoreboard: Analysis Port)

해설:

핵심 개념: UVM에서는 통신 특성에 따라 다른 TLM Port를 사용합니다. Driver-Sequencer는 양방향 handshake가 필요한 Sequence Item Port를, Monitor-Scoreboard는 단방향 broadcast가 가능한 Analysis Port를 사용합니다.

단계별 분석:

1. Driver와 Sequencer는 get_next_item()과 item_done()으로 양방향 통신을 합니다. Driver가 준비되면 요청하고, Sequencer가 제공할 때까지 기다립니다.
2. Monitor와 Scoreboard는 단방향 통신입니다. Monitor가 관찰한 데이터를 일방적으로 전송하며, Scoreboard의 응답을 기다리지 않습니다.
3. Sequence Item Port는 1:1 연결만 가능하지만, Analysis Port는 1:N 연결이 가능하여 여러 Scoreboard에 동시 전송할 수 있습니다.
4. 이런 구분을 통해 각 통신의 특성에 맞는 효율적인 구조를 만들 수 있습니다.

코드 예시:

```
// Driver: Sequence Item Port 사용
class driver extends uvm_driver #(transaction);
```

```

virtual task run_phase(uvm_phase phase);
    forever begin
        // 양방향 통신
        seq_item_port.get_next_item(tc); // Sequencer에 요청
        // 하드웨어 제어
        aif.a <= tc.a;
        aif.b <= tc.b;
        seq_item_port.item_done();           // Sequencer에 완료 통지
        #10;
    end
endtask
endclass

// Monitor: Analysis Port 사용
class monitor extends uvm_monitor;
    uvm_analysis_port #(transaction) send;

    virtual task run_phase(uvm_phase phase);
        forever begin
            #10;
            t.a = aif.a;
            t.b = aif.b;
            t.y = aif.y;
            send.write(t); // 단방향 전송, 응답 대기 안함
        end
    endtask
endclass

// Agent에서 연결
virtual function void connect_phase(uvm_phase phase);
    // Sequence Item Port 연결 (1:1)
    d.seq_item_port.connect(seqr.seq_item_export);

    // Analysis Port는 Environment에서 연결 (1:N 가능)
    // m.send.connect(scoreboard.recv);
    // m.send.connect(coverage_collector.recv);
endfunction

```

오답 분석:

- 1 번: 통신 방향이 완전히 반대입니다.
- 3 번: Blocking Get Port는 다른 용도로 사용되며, UVM에서 일반적으로 Sequence Item Port와 Analysis Port를 선호합니다.
- 4 번: Analysis Port만으로는 Driver-Sequencer 간 양방향 handshake를 구현할 수 없습니다.

실무 팁: 실무에서는 이 두 가지 기본 Port 외에도 TLM FIFO, TLM Analysis FIFO 등을 사용하여 더 복잡한 통신 패턴을 구현할 수 있습니다. 예를 들어 여러 Monitor의 데이터를 correlation하여 검증할 때 FIFO를 사용하면 타이밍 차이를 흡수할 수 있습니다.

관련 문서 섹션: 8장 UVM Agent Class, 9장 UVM Environment Class

문제 11: UVM Phase 실행 순서

다음 중 UVM Phase들의 올바른 실행 순서는 무엇입니까?

선택지:

1. build → run → connect → extract → check → report
2. connect → build → run → extract → check → report
3. build → connect → run → extract → check → report
4. build → connect → extract → run → check → report

정답: 3 번 (build → connect → run → extract → check → report)

해설:

핵심 개념: UVM은 검증 환경을 체계적으로 실행하기 위해 여러 Phase로 나누어 순차적으로 진행합니다. Build Phase에서 컴포넌트를 생성하고, Connect Phase에서 연결하며, Run Phase에서 실제 테스트를 수행한 후, 결과를 수집하고 분석합니다.

단계별 분석:

1. Build Phase는 모든 UVM 컴포넌트를 생성하고 Configuration Database에서 설정을 가져오는 단계입니다. 최상위부터 하위로 순차 실행됩니다.
2. Connect Phase는 생성된 컴포넌트들 간의 TLM 연결을 수행합니다. Agent에서 Driver-Sequencer를, Environment에서 Monitor-Scoreboard를 연결합니다.
3. Run Phase는 실제 테스트가 실행되는 단계로, Sequence가 시작되고 DUT가 동작하며 검증이 수행됩니다. 이 단계는 objection이 모두 해제될 때까지 계속됩니다.
4. Extract, Check, Report Phase는 시뮬레이션 종료 후 결과를 수집하고 분석하여 최종 리포트를 생성합니다.

코드 예시:

```
class my_component extends uvm_component;
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        `uvm_info("PHASE", "Build phase executing", UVM_LOW)
        // 컴포넌트 생성, 설정 읽기
    endfunction

    virtual function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        `uvm_info("PHASE", "Connect phase executing", UVM_LOW)
        // TLM port 연결
    endfunction

    virtual task run_phase(uvm_phase phase);
        `uvm_info("PHASE", "Run phase executing", UVM_LOW)
        // 메인 테스트 로직
    endtask

    virtual function void extract_phase(uvm_phase phase);
        super.extract_phase(phase);
        `uvm_info("PHASE", "Extract phase executing", UVM_LOW)
        // 시뮬레이션 데이터 수집
    endfunction
```

```

virtual function void check_phase(uvm_phase phase);
    super.check_phase(phase);
    `uvm_info("PHASE", "Check phase executing", UVM_LOW)
    // 데이터 유효성 검사
endfunction

virtual function void report_phase(uvm_phase phase);
    super.report_phase(phase);
    `uvm_info("PHASE", "Report phase executing", UVM_LOW)
    // 최종 결과 리포트
    $display("== Final Report ==");
    $display("Total tests: %0d", total_count);
    $display("Pass: %0d", pass_count);
endfunction
endclass

// 시뮬레이션 로그 예시
// UVM_INFO @ 0: Build phase executing
// UVM_INFO @ 0: Connect phase executing
// UVM_INFO @ 0: Run phase executing
// (테스트 수행)
// UVM_INFO @ 1000: Extract phase executing
// UVM_INFO @ 1000: Check phase executing
// UVM_INFO @ 1000: Report phase executing

```

오답 분석:

- 1 번: Connect Phase 가 Run Phase 뒤에 오면 TLM 연결이 안된 상태로 테스트가 실행되어 실패합니다.
- 2 번: Build Phase 전에 Connect Phase 가 오면 아직 생성되지 않은 컴포넌트를 연결하려고 시도하여 에러가 발생합니다.
- 4 번: Extract Phase 가 Run Phase 전에 오면 아직 수집할 데이터가 없습니다.

실무 팁: 실무에서는 end_of_elaboration, start_of_simulation 같은 추가 Phase 도 사용하여 더 세밀한 제어를 할 수 있습니다. 또한 각 Phase에서 super.phase_name(phase) 호출을 빼먹지 않아야 부모 클래스의 Phase 로직이 올바르게 실행됩니다.

관련 문서 섹션: 12 장 전체 시뮬레이션 플로우와 실무 적용

문제 12: Transaction의 제약 조건 (Constraint)

4 비트 덧셈기를 테스트할 때 캐리가 발생하는 경우만 집중적으로 테스트하고 싶습니다. 다음 중 올바른 constraint 작성 방법은 무엇입니까?

```
class transaction extends uvm_sequence_item;
    rand bit [3:0] a, b;
    bit [4:0] y;

    // 여기에 constraint 추가
endclass
```

선택지:

1. constraint carry_test { a + b > 15; }
2. constraint carry_test { a + b >= 16; }
3. constraint carry_test { a inside {[8:15]} && b inside {[8:15]}; }
4. constraint carry_test { (a + b) inside {[16:30]}; }

정답: 2번 (constraint carry_test { a + b >= 16; })

해설:

핵심 개념: SystemVerilog의 constraint는 randomize() 호출 시 지정된 조건을 만족하는 값만 생성하도록 제한합니다. 4비트 덧셈의 캐리는 결과가 15를 초과할 때 발생하므로, 두 입력의 합이 16 이상이 되도록 제약해야 합니다.

단계별 분석:

1. 4비트 입력의 최대값은 15이므로, 두 값의 합이 16 이상이면 5번째 비트(캐리)가 1이 됩니다.
2. $a + b \geq 16$ constraint를 사용하면 솔버가 자동으로 이 조건을 만족하는 a, b 값을 생성합니다.
3. 예를 들어 ($a=8, b=8$), ($a=15, b=15$), ($a=10, b=10$) 등이 생성될 수 있습니다.
4. 반대로 캐리가 없는 경우를 테스트하려면 $a + b < 16$ 을 사용하면 됩니다.

코드 예시:

```
// 캐리 발생 케이스만 테스트
class carry_transaction extends uvm_sequence_item;
    rand bit [3:0] a, b;
    bit [4:0] y;

    constraint carry_test {
        a + b >= 16; // 16~30 범위
    }

    `uvm_object_utils_begin(carry_transaction)
        `uvm_field_int(a, UVM_DEFAULT)
        `uvm_field_int(b, UVM_DEFAULT)
        `uvm_field_int(y, UVM_DEFAULT)
    `uvm_object_utils_end
endclass
```

```
// 경계값 중심 테스트
class boundary_transaction extends uvm_sequence_item;
    rand bit [3:0] a, b;
    bit [4:0] y;
```

```
constraint boundary_test {
    a inside {0, 1, 14, 15};
    b inside {0, 1, 14, 15};
}
endclass
```

```
// 특정 범위 테스트
class range_transaction extends uvm_sequence_item;
    rand bit [3:0] a, b;
    bit [4:0] y;

    constraint low_range {
        a inside {[0:7]};
        b inside {[0:7]};
        a + b < 16; // 캐리 없음 보장
    }
endclass
```

```

// Sequence에서 사용
class targeted_sequence extends uvm_sequence #(transaction);
    virtual task body();
        carry_transaction ct = carry_transaction::type_id::create("ct");
        repeat(100) begin
            start_item(ct);
            assert(ct.randomize()); // constraint 만족하는 값 생성
            finish_item(ct);
        end
    endtask
endclass

```

오답 분석:

- 1 번: $a + b > 15$ 는 의미상 맞지만, ≥ 16 이 더 명확한 의도를 표현합니다.
- 3 번: 두 입력 모두 8~15 범위로 제한하면 캐리 발생 케이스가 제한적입니다. 예를 들어 ($a=1$, $b=15$)처럼 한쪽이 작아도 캐리가 발생할 수 있습니다.
- 4 번: `inside` 연산자는 배열이나 범위에 사용되지만, $a + b$ 같은 표현식에 직접 사용하면 문법 오류가 발생할 수 있습니다.

실무 팁: 실무에서는 여러 constraint 를 조합하여 복잡한 테스트 시나리오를 만듭니다. 또한 `dist` 연산자를 사용하여 특정 값의 생성 확률을 조정할 수 있습니다. 예를 들어 "`a dist {0:=10, [1:14]:=80, 15:=10}`"처럼 경계값의 생성 빈도를 높일 수 있습니다.

관련 문서 섹션: 3 장 UVM Transaction Class, 4 장 UVM Sequence Class

문제 13: Driver 의 non-blocking 할당

Driver에서 Interface 신호에 값을 할당할 때 non-blocking 할당(`->`) 대신 blocking 할당(`=`)을 사용하면 어떤 문제가 발생할 수 있습니까?

```

virtual task run_phase(uvm_phase phase);
    forever begin
        seq_item_port.get_next_item(tc);
        // 방법 A: blocking

```

```

aif.a = tc.a;
aif.b = tc.b;

// 방법 B: non-blocking
aif.a <= tc.a;
aif.b <= tc.b;

seq_item_port.item_done();
#10;
end
endtask

```

선택지:

1. Blocking 할당이 더 빠르고 효율적이므로 권장됨
2. Non-blocking 할당은 클럭 에지에서만 값이 변경되어 조합회로에는 부적합
3. Blocking 할당 사용 시 race condition이 발생할 수 있어 non-blocking이 안전함
4. 두 방법 모두 동일하게 작동하므로 차이가 없음

정답: 3번 (Blocking 할당 사용 시 race condition이 발생할 수 있어 non-blocking이 안전함)

해설:

핵심 개념: SystemVerilog에서 blocking(=)과 non-blocking(<=) 할당의 차이는 실행 시점입니다. Testbench에서는 race condition을 피하고 예측 가능한 동작을 보장하기 위해 일반적으로 non-blocking 할당을 사용합니다.

단계별 분석:

1. Blocking 할당(=)은 현재 시간 스텝에서 즉시 값을 변경합니다. 같은 시간에 여러 할당이 있으면 실행 순서에 따라 결과가 달라질 수 있습니다.
2. Non-blocking 할당(<=)은 현재 시간 스텝의 모든 우변 값을 평가한 후, 스텝 끝에서 동시에 좌변에 할당합니다. 이는 하드웨어의 동시 동작을 정확히 모델링합니다.
3. 여러 컴포넌트(Driver, Monitor)가 같은 시간에 Interface를 접근할 때, non-blocking을 사용하면 순서 의존적 버그를 피할 수 있습니다.

4. 특히 clocked interface에서 non-blocking을 사용하면 실제 하드웨어의 플립플롭 동작과 일치합니다.

코드 예시:

```
// 안전한 방법: non-blocking 사용
virtual task run_phase(uvm_phase phase);
    forever begin
        seq_item_port.get_next_item(tc);

        // Non-blocking: 모든 우변 평가 후 동시 할당
        aif.a <= tc.a;
        aif.b <= tc.b;
        // 이 시점에 aif.a와 aif.b는 아직 이전 값

        seq_item_port.item_done();
        #10; // 다음 시간 스텝으로 진행하면 할당 완료
    end
endtask

// 문제가 될 수 있는 방법: blocking 사용
virtual task run_phase_blocking(uvm_phase phase);
    forever begin
        seq_item_port.get_next_item(tc);

        // Blocking: 즉시 할당
        aif.a = tc.a; // 이 시점에 aif.a 값 즉시 변경
        aif.b = tc.b; // 이 시점에 aif.b 값 즉시 변경
        // Monitor가 같은 시간에 읽으면 혼재된 값을 볼 수 있음

        seq_item_port.item_done();
        #10;
    end
endtask

// Clocked interface 예시
interface clocked_if(input logic clk);
    logic [3:0] data;
    logic valid;
```

```

clocking cb @(posedge clk);
    output data, valid; // 자동으로 non-blocking
endclocking
endinterface

class safe_driver extends uvm_driver;
    virtual clocked_if cif;

    virtual task run_phase(uvm_phase phase);
        forever begin
            seq_item_port.get_next_item(tc);
            @(cif.cb); // 클럭 에지 대기
            cif.cb.data <= tc.data; // Non-blocking
            cif.cb.valid <= 1;
            seq_item_port.item_done();
        end
    endtask
endclass

```

오답 분석:

- 1 번: 성능 차이는 미미하며, 안전성이 더 중요합니다.
- 2 번: Non-blocking 은 조합회로에도 사용 가능하며, 오히려 권장됩니다.
- 4 번: 단순한 경우 차이가 없어 보일 수 있지만, 복잡한 환경에서는 race condition 이 발생할 수 있습니다.

실무 팁: 실무에서는 clocking block 을 사용하여 더 명확한 타이밍 제어를 합니다. Clocking block 은 자동으로 non-blocking 할당을 사용하고 skew 를 제어할 수 있어 복잡한 프로토콜에서 매우 유용합니다. 또한 SystemVerilog LRM 의 scheduling semantics 를 이해하면 언제 어떤 할당을 사용해야 하는지 명확해집니다.

관련 문서 섹션: 5 장 UVM Driver Class

문제 14: Monitor 의 데이터 수집 타이밍

Monitor 가 DUT 의 출력을 읽을 때 적절한 타이밍은 언제입니까? 조합회로 덧셈기를 가정합니다.

```
virtual task run_phase(uvm_phase phase);  
  forever begin  
    // 옵션들  
    end  
  endtask
```

선택지:

1. Driver 가 입력을 변경하자마자 즉시 읽기
2. 입력 변경 후 전파 지연을 고려하여 충분한 시간 대기 후 읽기
3. 클럭 에지에서만 읽기 (조합회로여도)
4. 랜덤한 시간에 읽어서 다양한 상태 캡처

정답: 2 번 (입력 변경 후 전파 지연을 고려하여 충분한 시간 대기 후 읽기)

해설:

핵심 개념: 조합회로는 입력 변경 후 전파 지연(propagation delay) 시간이 지나야 안정된 출력을 생성합니다. Monitor 는 출력이 안정화된 후에 읽어야 정확한 값을 수집할 수 있습니다.

단계별 분석:

1. Driver 가 Interface 에 새로운 입력값을 출력합니다. (시간 T)
2. DUT 는 조합 논리를 통해 출력을 계산합니다. 실제 하드웨어에서는 게이트 지연이 누적됩니다. ($T \sim T+\Delta$)
3. 충분한 지연 시간 후 출력이 안정화됩니다. ($T+\Delta$)
4. Monitor 는 이 안정화된 값을 읽어 Transaction에 저장하고 Scoreboard 로 전송합니다.

코드 예시:

```
// 올바른 Monitor 구현
```

```

virtual task run_phase(uvm_phase phase);
    forever begin
        #10; // Driver 의 할당과 DUT 처리 대기

        // 이 시점에 모든 신호가 안정화됨
        t.a = aif.a;
        t.b = aif.b;
        t.y = aif.y;

        `uvm_info("MON", $sformatf("Collected: a=%0d, b=%0d, y=%0d",
                                      t.a, t.b, t.y), UVM_HIGH)

        send.write(t);
    end
endtask

// 타이밍 다이어그램
// Time:      0ns      5ns      10ns      15ns
// Driver:    [a=5,b=3]           [a=7,b=2]
// DUT:       [계산중]   [y=8 안정]   [계산중]
// Monitor:   [대기]     [읽기: y=8]  [대기]

// 클럭 기반 순차회로의 경우
interface clocked_if(input logic clk);
    logic [3:0] a, b;
    logic [4:0] y;

    clocking mon_cb @(posedge clk);
        input a, b, y;
    endclocking
endinterface

class sequential_monitor extends uvm_monitor;
    virtual clocked_if cif;

    virtual task run_phase(uvm_phase phase);
        forever begin
            @(cif.mon_cb); // 클럭 에지에서 샘플링
            t.a = cif.mon_cb.a;

```

```

    t.b = cif.mon_cb.b;
    t.y = cif.mon_cb.y;
    send.write(t);
end
endtask
endclass

// 프로토콜 기반 Monitor (UART 예시)
class uart_monitor extends uvm_monitor;
virtual task run_phase(uvm_phase phase);
forever begin
// Start bit 감지
@(negedge uart_if.rx);

// Half bit period 대기하여 중간 샘플링
#(BIT_PERIOD/2);

// 데이터 비트 수집
for(int i=0; i<8; i++) begin
#BIT_PERIOD;
data[i] = uart_if.rx;
end

send.write(uart_trans);
end
endtask
endclass

```

오답 분석:

- 1 번: 입력 변경 직후에는 DUT 출력이 아직 업데이트되지 않아 이전 값이나 중간 값을 읽을 수 있습니다.
- 3 번: 조합회로는 클럭이 없거나 클럭과 무관하게 동작하므로 클럭 에지를 기다릴 필요가 없습니다.
- 4 번: 랜덤 샘플링은 일관성 없는 데이터를 수집하여 검증이 불가능하게 만듭니다.

실무 팁: 실무에서는 시뮬레이터의 delta cycle 과 NBA(Non-Blocking Assignment) 스케줄링을 이해하는 것이 중요합니다. 또한 실제 하드웨어에서는 worst-case propagation delay를 고려하여 충분한 setup/hold time 을

확보해야 합니다. 복잡한 프로토콜의 경우 FSM을 사용하여 정확한 샘플링 시점을 결정합니다.

관련 문서 섹션: 6 장 UVM Monitor Class

문제 15: Virtual Sequencer 패턴

복잡한 시스템에서 여러 Agent의 Sequence를 동기화해야 할 때 사용하는 Virtual Sequencer 패턴의 목적은 무엇입니까?

```
class virtual_sequence extends uvm_sequence;
    cpu_sequencer cpu_seqr;
    mem_sequencer mem_seqr;

    virtual task body();
        fork
            cpu_seq.start(cpu_seqr);
            mem_seq.start(mem_seqr);
        join
        endtask
    endclass
```

선택지:

1. 하나의 Sequencer로 모든 Agent를 제어하여 코드 간소화
2. 여러 Agent의 Sequence를 동기화하고 복잡한 시나리오 조율
3. Sequencer의 메모리 사용량 감소
4. Agent 간 직접 통신 제공

정답: 2 번 (여러 Agent의 Sequence를 동기화하고 복잡한 시나리오 조율)

해설:

핵심 개념: Virtual Sequencer는 여러 Agent의 Sequencer에 대한 핸들을 가지고 있어, 복잡한 시스템 레벨 시나리오에서 여러 인터페이스의 Transaction을 동기화하고 조율할 수 있게 해줍니다.

단계별 분석:

1. 각 Agent는 자신의 Sequencer를 가지고 있어 독립적으로 Sequence를 실행할 수 있습니다.
2. Virtual Sequencer는 이러한 여러 Sequencer에 대한 참조를 가지고 있습니다.
3. Virtual Sequence는 Virtual Sequencer에서 실행되며, 여러 하위 Sequence를 조율하여 복잡한 시나리오를 구현합니다.
4. 예를 들어 CPU가 메모리에 쓰기를 시작하기 전에 특정 설정 시퀀스를 먼저 실행하는 등의 조율이 가능합니다.

코드 예시:

```
// Virtual Sequencer 정의
class virtual_sequencer extends uvm_sequencer;
  `uvm_component_utils(virtual_sequencer)

  cpu_sequencer cpu_seqr;
  mem_sequencer mem_seqr;
  pcie_sequencer pcie_seqr;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
endclass

// Virtual Sequence - 복잡한 시나리오 조율
class system_init_vseq extends uvm_sequence;
  `uvm_object_utils(system_init_vseq)

  cpu_basic_seq cpu_seq;
  mem_init_seq mem_seq;

  virtual task body();
    // 1단계: 메모리 초기화
    `uvm_info("VSEQ", "Initializing memory", UVM_LOW)
    mem_seq = mem_init_seq::type_id::create("mem_seq");
    mem_seq.start(p_sequencer.mem_seqr);
```

```

// 2 단계: CPU 부팅 시퀀스
`uvm_info("VSEQ", "Starting CPU boot sequence", UVM_LOW)
cpu_seq = cpu_basic_seq::type_id::create("cpu_seq");
cpu_seq.start(p_sequencer.cpu_seqr);

// 3 단계: 동시 동작
`uvm_info("VSEQ", "Starting parallel operations", UVM_LOW)
fork
    cpu_data_seq.start(p_sequencer.cpu_seqr);
    mem_response_seq.start(p_sequencer.mem_seqr);
join
endtask
endclass

// 데이터 동기화가 필요한 복잡한 시나리오
class data_transfer_vseq extends uvm_sequence;
    virtual task body();
        // CPU 가 PCIe 로 데이터 전송 요청
        fork
            begin
                cpu_write_seq.addr = 'h1000;
                cpu_write_seq.data = test_pattern;
                cpu_write_seq.start(p_sequencer.cpu_seqr);
            end
            begin
                // PCIe 가 CPU 요청 감지하고 응답
                wait(pcie_if.request_detected);
                pcie_response_seq.start(p_sequencer.pcie_seqr);
            end
            begin
                // 메모리는 데이터 저장 준비
                mem_ready_seq.start(p_sequencer.mem_seqr);
            end
        join
    endtask
endclass

// Environment에서 Virtual Sequencer 연결
class env extends uvm_env;

```

```

virtual_sequencer v_seqr;
cpu_agent cpu_agt;
mem_agent mem_agt;

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    v_seqr = virtual_sequencer::type_id::create("v_seqr", this);
    cpu_agt = cpu_agent::type_id::create("cpu_agt", this);
    mem_agt = mem_agent::type_id::create("mem_agt", this);
endfunction

virtual function void connect_phase(uvm_phase phase);
    // Virtual Sequencer에 하위 Sequencer 연결
    v_seqr.cpu_seqr = cpu_agt.seqr;
    v_seqr.mem_seqr = mem_agt.seqr;
endfunction

endclass

// Test에서 Virtual Sequence 실행
class complex_test extends base_test;
    virtual task run_phase(uvm_phase phase);
        system_init_vseq init_seq;
        phase.raise_objection(this);

        init_seq = system_init_vseq::type_id::create("init_seq");
        init_seq.start(e.v_seqr); // Virtual Sequencer에서 실행

        phase.drop_objection(this);
    endtask
endclass

```

오답 분석:

- 1 번: Virtual Sequencer 는 코드를 간소화하는 것이 아니라 오히려 계층을 추가하지만, 복잡한 시나리오 관리를 가능하게 합니다.
- 3 번: 메모리 사용량 감소와는 관련이 없습니다.
- 4 번: Agent 간 직접 통신이 아니라 Sequence 레벨에서의 조율을 제공합니다.

실무 팁: 실무에서는 SoC 레벨 검증에서 Virtual Sequencer 가 필수적입니다. CPU, 메모리 컨트롤러, 주변장치들이 특정 순서로 동작해야 하는 복잡한 시나리오를 구현할 때 Virtual Sequence 를 사용하면 테스트 의도가 명확해지고 재사용성이 높아집니다.

관련 문서 섹션: 4 장 UVM Sequence Class, 8 장 UVM Agent Class

문제 16: Coverage Closure 전략

Functional Coverage 가 95%에서 정체되어 있습니다. 남은 5%를 달성하기 위한 가장 효과적인 접근 방법은 무엇입니까?

선택지:

1. 랜덤 테스트 횟수를 10 배로 늘리기
2. Coverage report 를 분석하여 miss 된 bins 를 확인하고 directed test 작성
3. Coverage 목표를 95%로 낮추기
4. 모든 Sequence 를 처음부터 다시 작성

정답: 2 번 (Coverage report 를 분석하여 miss 된 bins 를 확인하고 directed test 작성)

해설:

핵심 개념: Coverage closure 는 체계적인 접근이 필요합니다. 자동 생성된 coverage report 를 분석하여 어떤 bins 가 hit 되지 않았는지 파악하고, 그 특정 케이스만 겨냥하는 directed sequence 를 작성하는 것이 가장 효율적입니다.

단계별 분석:

1. 시뮬레이터가 생성한 coverage report 를 열어 어떤 coverpoint 의 어떤 bin 이 0 hit 인지 확인합니다.
2. 왜 그 bin 이 hit 되지 않았는지 분석합니다. Constraint 때문인지, 확률이 낮아서인지, 아니면 불가능한 조합인지 판단합니다.
3. 불가능한 조합이라면 illegal_bins 로 지정하여 coverage 에서 제외합니다.

4. 가능하지만 hit 되지 않은 케이스를 위해 해당 값을 생성하는 directed sequence 를 작성합니다.

코드 예시:

```
// Coverage 정의
covergroup adder_coverage;
    a_cp: coverpoint tr.a {
        bins zero = {0};
        bins low = {[1:7]};
        bins high = {[8:14]};
        bins max = {15};
    }

    b_cp: coverpoint tr.b {
        bins zero = {0};
        bins low = {[1:7]};
        bins high = {[8:14]};
        bins max = {15};
    }

    cross a_cp, b_cp {
        // 불가능한 조합 제외 (예시)
        illegal_bins overflow = binsof(a_cp.max) && binsof(b_cp.max);
    }
endgroup

// Coverage report 분석 후 발견: (a=0, b=15) 조합이 hit 안됨

// Directed Sequence 작성
class directed_coverage_seq extends uvm_sequence #(transaction);
    `uvm_object_utils(directed_coverage_seq)

    // Miss 된 특정 케이스들
    typedef struct {
        bit [3:0] a;
        bit [3:0] b;
    } directed_case_t;
```

```

directed_case_t directed_cases[] = '{  

    '{a: 0, b: 15}, // Miss 된 케이스 1  

    '{a: 15, b: 0}, // Miss 된 케이스 2  

    '{a: 7, b: 1}, // Miss 된 케이스 3  

    '{a: 1, b: 14} // Miss 된 케이스 4  

};  
  

virtual task body();  

    transaction t;  
  

    // Directed cases 실행  

    foreach(directed_cases[i]) begin  

        t = transaction::type_id::create("t");  

        start_item(t);  

        t.a = directed_cases[i].a;  

        t.b = directed_cases[i].b;  

        `uvm_info("DIR_SEQ", $sformatf("Executing directed case: a=%0d,  

b=%0d",  

            t.a, t.b), UVM_LOW)  

        finish_item(t);  

    end  
  

    // 추가 랜덤 테스트  

    repeat(100) begin  

        t = transaction::type_id::create("t");  

        start_item(t);  

        t.randomize();  

        finish_item(t);  

    end  

endtask  

endclass  
  

// Coverage 모니터링  

class coverage_monitor extends uvm_subscriber #(transaction);  

    adder_coverage cov;  
  

    function new(string name, uvm_component parent);  

        super.new(name, parent);  

        cov = new();  

    endfunction

```

```

endfunction

virtual function void write(transaction t);
    cov.sample();

    // 초기적으로 coverage 리포트
    if(cov.get_inst_coverage() >= 95.0) begin
        `uvm_info("COV", $sformatf("Coverage reached: %.2f%%",
            cov.get_inst_coverage()), UVM_LOW)
    end
endfunction

virtual function void report_phase(uvm_phase phase);
    `uvm_info("COV_FINAL", $sformatf("Final Coverage: %.2f%%",
        cov.get_inst_coverage()), UVM_NONE)

    // Miss 된 bins 리포트
    if(cov.get_inst_coverage() < 100.0) begin
        `uvm_warning("COV", "Some coverage bins not hit")
        // 실제로는 자동 리포트 생성
    end
endfunction
endclass

// Test에서 coverage-driven 접근
class coverage_closure_test extends base_test;
    virtual task run_phase(uvm_phase phase);
        directed_coverage_seq dir_seq;
        phase.raise_objection(this);

        // 1단계: 일반 랜덤 테스트
        random_seq.start(e.a.seqr);
        #1000;

        // 2단계: Coverage 확인 및 directed test
        if(e.cov_monitor.cov.get_inst_coverage() < 100.0) begin
            `uvm_info("TEST", "Running directed sequences for coverage
closure",
                UVM_LOW)
        end
    endtask
endclass

```

```

dir_seq = directed_coverage_seq::type_id::create("dir_seq");
dir_seq.start(e.a.seqr);
end

phase.drop_objection(this);
endtask
endclass

```

오답 분석:

- 1 번: 랜덤 테스트만으로는 확률이 매우 낮은 corner case를 hit하기 어렵고 시간이 오래 걸립니다.
- 3 번: Coverage 목표를 낮추는 것은 잠재적 버그를 놓칠 수 있어 바람직하지 않습니다.
- 4 번: 이미 95%를 달성했다면 나머지 5%만 집중하는 것이 효율적입니다.

실무 팁: 실무에서는 coverage-driven verification 방법론을 사용합니다. 먼저 랜덤 테스트로 쉽게 도달 가능한 coverage를 달성하고, report를 분석하여 miss 된 케이스를 파악한 후, directed test로 채워나갑니다. 또한 coverage model을 지속적으로 개선하여 실제로 중요한 케이스만 tracking하도록 합니다.

관련 문서 섹션: 12 장 전체 시뮬레이션 플로우와 실무 적용

문제 17: UVM Factory Override

테스트 중 Transaction 클래스의 동작을 변경하고 싶습니다. 코드 수정 없이 다른 Transaction 타입을 사용하려면 어떻게 해야 합니까?

```

class extended_transaction extends transaction;
  rand bit [7:0] extra_field;

  `uvm_object_utils(extended_transaction)
endclass

```

선택지:

1. 모든 Sequence 코드에서 transaction을 extended_transaction으로 변경

2. Factory override 를 사용하여 transaction 타입을 extended_transaction 으로 대체
3. 새로운 Agent 를 만들어 extended_transaction 전용으로 사용
4. Transaction 클래스는 변경할 수 없음

정답: 2 번 (Factory override 를 사용하여 transaction 타입을 extended_transaction 으로 대체)

해설:

핵심 개념: UVM Factory pattern 을 사용하면 런타임에 객체 타입을 변경할 수 있습니다. set_type_override()나 set_inst_override() 를 사용하여 기존 코드 수정 없이 다른 타입의 객체를 생성할 수 있습니다.

단계별 분석:

1. UVM Factory 는 객체 생성을 관리하는 중앙 레지스트리입니다. type_id::create() 로 생성된 모든 객체는 Factory 를 거칩니다.
2. set_type_override() 를 호출하면 해당 타입의 모든 인스턴스가 새로운 타입으로 대체됩니다.
3. 이는 다형성 (polymorphism) 을 활용하여 확장된 클래스로 교체하면서도 기존 코드와 호환성을 유지합니다.
4. 테스트별로 다른 Transaction 타입을 사용하여 다양한 시나리오를 실험할 수 있습니다.

코드 예시:

```
// 기본 Transaction
class transaction extends uvm_sequence_item;
  rand bit [3:0] a, b;
  bit [4:0] y;

  `uvm_object_utils(transaction)

  function new(string name = "transaction");
    super.new(name);
  endfunction
endclass
```

```

// 확장된 Transaction - 추가 기능
class extended_transaction extends transaction;
    rand bit [7:0] extra_field;
    rand bit enable_debug;

    constraint extra_constraint {
        extra_field inside {[0:100]};
    }

`uvm_object_utils_begin(extended_transaction)
    `uvm_field_int(extra_field, UVM_DEFAULT)
    `uvm_field_int(enable_debug, UVM_DEFAULT)
`uvm_object_utils_end

function new(string name = "extended_transaction");
    super.new(name);
endfunction
endclass

// Error injection을 위한 Transaction
class error_transaction extends transaction;
    rand bit inject_error;
    rand bit [3:0] error_type;

    constraint error_dist {
        inject_error dist {0 := 80, 1 := 20}; // 20% 에러 주입
    }

`uvm_object_utils(error_transaction)
endclass

// Test에서 Factory Override 사용
class override_test extends base_test;
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);

    // Type override: 모든 transaction을 extended_transaction으로 대체
    transaction::type_id::set_type_override(
        extended_transaction::get_type());

```

```

// 또는 특정 인스턴스만 override
// set_inst_override_by_type("uvm_test_top.e.a.seqr.*",
//   transaction::get_type(),
//   extended_transaction::get_type());
endfunction
endclass

// Error injection test
class error_injection_test extends base_test;
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    // 에러 주입용 Transaction으로 override
    transaction::type_id::set_type_override(
      error_transaction::get_type());
  endfunction

  virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    // 기존 Sequence를 그대로 사용하지만
    // 내부적으로 error_transaction이 생성됨
    gen.start(e.a.seqr);
    #500;

    phase.drop_objection(this);
  endtask
endclass

// Sequence는 변경 불필요
class generator extends uvm_sequence #(transaction);
  virtual task body();
    transaction t;
    repeat(10) begin
      // type_id::create()를 사용했으므로 Factory override 적용됨
      t = transaction::type_id::create("t");
      start_item(t);
      t.randomize();
      finish_item(t);
    end
  endtask
endclass

```

```

    end
  endtask
endclass

// Component override 예시
class debug_driver extends driver;
  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(tc);

      // 추가 디버그 로깅
      `uvm_info("DEBUG_DRV", $sformatf(
        "Detailed info: a=%0d, b=%0d, timestamp=%0t",
        tc.a, tc.b, $time), UVM_HIGH)

      aif.a <= tc.a;
      aif.b <= tc.b;
      seq_item_port.item_done();
      #10;
    end
  endtask
endclass

class debug_test extends base_test;
  virtual function void build_phase(uvm_phase phase);
    // Driver도 override 가능
    driver::type_id::set_type_override(debug_driver::get_type());
    super.build_phase(phase);
  endfunction
endclass

```

오답 분석:

- 1 번: 모든 코드를 수정하면 유지보수가 어렵고 실수할 가능성이 높습니다.
- 3 번: 새로운 Agent를 만드는 것은 과도하며 코드 중복이 발생합니다.
- 4 번: Factory pattern 덕분에 매우 쉽게 변경할 수 있습니다.

실무 팁: 실무에서는 Factory override를 다양하게 활용합니다. 예를 들어 디버그 모드에서는 더 많은 로깅을 하는 컴포넌트로, 스트레스 테스트에서는 에러를

주입하는 Transaction으로 override 할 수 있습니다. 또한 synthesis 용과 simulation 용 모델을 같은 인터페이스로 교체할 수도 있습니다.

관련 문서 섹션: 3장 UVM Transaction Class, 10장 UVM Test Class

문제 18: Callback Mechanism

DUT의 특정 이벤트가 발생했을 때 추가 동작을 수행하고 싶지만, 기존 Driver 코드를 수정하고 싶지 않습니다. 가장 적합한 UVM 메커니즘은 무엇입니까?

선택지:

1. Driver 클래스를 상속받아 새로운 클래스 작성
2. UVM Callback 을 사용하여 hook 포인트 추가
3. Driver 코드를 직접 수정
4. 별도의 Monitor로 이벤트 감지

정답: 2번 (UVM Callback 을 사용하여 hook 포인트 추가)

해설:

핵심 개념: UVM Callback 은 컴포넌트의 특정 지점에 사용자 정의 동작을 주입할 수 있는 메커니즘입니다. 기존 코드를 수정하지 않고도 새로운 기능을 추가할 수 있어 확장성이 뛰어납니다.

단계별 분석:

1. Callback base class 를 정의하여 hook 메서드들을 선언합니다.
2. 컴포넌트(예: Driver)에 callback 을 실행할 수 있는 코드를 추가합니다.
3. 실제 동작을 구현하는 callback 클래스를 작성합니다.
4. Test에서 이 callback 을 컴포넌트에 등록하면, hook 포인트에서 자동으로 실행됩니다.

코드 예시:

```
// 1 단계: Callback base class 정의
class driver_callback extends uvm_callback;
```

```

`uvm_object_utils(driver_callback)

// Hook 메서드들
virtual function void pre_send(transaction tr);
    // Transaction 전송 전 호출
endfunction

virtual function void post_send(transaction tr);
    // Transaction 전송 후 호출
endfunction

virtual task delay_control(ref int delay_cycles);
    // 지연 시간 제어
endtask
endclass

// 2 단계: Driver에 callback 지원 추가
class driver extends uvm_driver #(transaction);
    `uvm_component_utils(driver)
    `uvm_register_cb(driver, driver_callback) // Callback 등록

    virtual task run_phase(uvm_phase phase);
        forever begin
            seq_item_port.get_next_item(tc);

            // Pre-send callback 실행
            `uvm_do_callbacks(driver, driver_callback,
                pre_send(tc))

            // 실제 동작
            aif.a <= tc.a;
            aif.b <= tc.b;

            // Post-send callback 실행
            `uvm_do_callbacks(driver, driver_callback,
                post_send(tc))

            seq_item_port.item_done();
        end
    endtask
endclass

```

```

// Delay control callback
int delay = 10;
`uvm_do_callbacks(driver, driver_callback,
                  delay_control(delay))
#delay;
end
endtask
endclass

// 3 단계: 구체적인 callback 구현
class error_injection_callback extends driver_callback;
`uvm_object_utils(error_injection_callback)

rand bit inject_error;
constraint error_rate { inject_error dist {0 := 90, 1 := 10}; }

virtual function void pre_send(transaction tr);
  if(!randomize()) `uvm_error("CB", "Randomization failed")

  if(inject_error) begin
    `uvm_info("ERROR_CB", "Injecting error", UVM_LOW)
    tr.a = tr.a ^ 4'b1111; // 비트 반전으로 에러 주입
  end
endfunction
endclass

class logging_callback extends driver_callback;
`uvm_object_utils(logging_callback)

int transaction_count = 0;

virtual function void pre_send(transaction tr);
  transaction_count++;
  `uvm_info("LOG_CB", $sformatf(
    "Transaction #%-0d: a=%0d, b=%0d at time %0t",
    transaction_count, tr.a, tr.b, $time), UVM_HIGH)
endfunction

virtual function void post_send(transaction tr);
  `uvm_info("LOG_CB", $sformatf(

```

```

        "Transaction #%" transaction_count), UVM_DEBUG)
    endfunction
endclass

class adaptive_timing_callback extends driver_callback;
`uvm_object_utils(adaptive_timing_callback)

virtual task delay_control(ref int delay_cycles);
// DUT 상태에 따라 동적으로 지연 조정
if(backpressure_detected)
    delay_cycles = delay_cycles * 2;
else
    delay_cycles = 10;
endtask
endclass

// 4 단계: Test에서 callback 등록
class callback_test extends base_test;
error_injection_callback err_cb;
logging_callback log_cb;

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);

// Callback 인스턴스 생성
err_cb = error_injection_callback::type_id::create("err_cb");
log_cb = logging_callback::type_id::create("log_cb");
endfunction

virtual function void connect_phase(uvm_phase phase);
super.connect_phase(phase);

// Driver에 callback 등록
uvm_callbacks #(driver, driver_callback)::add(e.a.d, err_cb);
uvm_callbacks #(driver, driver_callback)::add(e.a.d, log_cb);
endfunction
endclass

// 여러 callback 조합 사용
class advanced_callback_test extends base_test;

```

```

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

// 조건부 callback 등록
if($test$plusargs("ERROR_INJECTION")) begin
    error_injection_callback err_cb =
        error_injection_callback::type_id::create("err_cb");
    uvm_callbacks #(driver, driver_callback)::add(e.a.d, err_cb);
    `uvm_info("TEST", "Error injection enabled", UVM_LOW)
end

if($test$plusargs("VERBOSE_LOGGING")) begin
    logging_callback log_cb =
        logging_callback::type_id::create("log_cb");
    uvm_callbacks #(driver, driver_callback)::add(e.a.d, log_cb);
    `uvm_info("TEST", "Verbose logging enabled", UVM_LOW)
end
endfunction
endclass

// Scoreboard callback 예시
class scoreboard_callback extends uvm_callback;
    virtual function void check_result(transaction tr, bit passed);
    endfunction
endclass

class statistics_callback extends scoreboard_callback;
    int pass_count = 0;
    int fail_count = 0;

    virtual function void check_result(transaction tr, bit passed);
        if(passed) pass_count++;
        else fail_count++;

        if((pass_count + fail_count) % 100 == 0) begin
            real pass_rate = 100.0 * pass_count / (pass_count +
fail_count);
            `uvm_info("STATS", $sformatf("Pass rate: %.2f%%", pass_rate),
UVM_LOW)
        end
    end

```

```
endfunction  
endclass
```

오답 분석:

- 1 번: 상속은 한 번만 가능하고, 여러 기능을 조합하기 어렵습니다.
- 3 번: 코드 수정은 유지보수를 어렵게 하고 재사용성을 떨어뜨립니다.
- 4 번: Monitor는 관찰만 하고 Driver 동작에 영향을 주기 어렵습니다.

실무 팁: 실무에서는 callback을 활용하여 디버깅, 성능 분석, 에러 주입, 동적 설정 변경 등 다양한 기능을 플러그인 방식으로 추가합니다. 특히 IP를 재사용할 때 핵심 코드는 그대로 두고 프로젝트별 요구사항을 callback으로 구현하면 매우 효율적입니다.

관련 문서 섹션: 5 장 UVM Driver Class

문제 19: Register Model (RAL) 통합

DUT 내부에 설정 레지스터들이 있을 때 UVM에서 이를 효과적으로 관리하는 방법은 무엇입니까?

선택지:

1. 레지스터 주소와 값을 하드코딩하여 직접 읽기/쓰기
2. UVM Register Model (RAL)을 사용하여 추상화된 인터페이스 제공
3. 별도의 레지스터 전용 Agent 생성
4. 레지스터 접근은 UVM에서 지원하지 않음

정답: 2 번 (UVM Register Model (RAL)을 사용하여 추상화된 인터페이스 제공)

해설:

핵심 개념: UVM Register Abstraction Layer (RAL)은 DUT 내부 레지스터를 객체지향적으로 모델링하여, 주소 계산이나 프로토콜 세부사항을 감추고 직관적인 API를 제공합니다. 이를 통해 레지스터 설정, 검증, 미러링을 쉽게 수행할 수 있습니다.

단계별 분석:

1. Register Model은 DUT의 레지스터 맵을 클래스 계층 구조로 표현합니다.
각 레지스터는 uvm_reg 객체로, 각 필드는 uvm_field 객체로 모델링됩니다.
2. read()와 write() 같은 추상 메서드를 제공하여 실제 버스 프로토콜(APB, AHB, AXI 등)과 독립적으로 레지스터에 접근할 수 있습니다.
3. Auto-prediction과 mirroring 기능으로 소프트웨어 모델과 실제 하드웨어 값의 일치 여부를 자동으로 검증합니다.
4. Sequence에서 reg_model.REG_NAME.write(value)처럼 직관적으로 사용할 수 있습니다.

코드 예시:

```
// 1 단계: Register 정의
class control_reg extends uvm_reg;
    rand uvm_reg_field enable;
    rand uvm_reg_field mode;
    rand uvm_reg_field interrupt_en;

    function new(string name = "control_reg");
        super.new(name, 32, UVM_NO_COVERAGE);
    endfunction

    virtual function void build();
        enable = uvm_reg_field::type_id::create("enable");
        enable.configure(this, 1, 0, "RW", 0, 1'b0, 1, 1, 0);

        mode = uvm_reg_field::type_id::create("mode");
        mode.configure(this, 2, 1, "RW", 0, 2'b00, 1, 1, 0);

        interrupt_en = uvm_reg_field::type_id::create("interrupt_en");
        interrupt_en.configure(this, 1, 3, "RW", 0, 1'b0, 1, 1, 0);
    endfunction

    `uvm_object_utils(control_reg)
endclass

class status_reg extends uvm_reg;
    rand uvm_reg_field busy;
```

```

rand uvm_reg_field error;
rand uvm_reg_field done;

function new(string name = "status_reg");
    super.new(name, 32, UVM_NO_COVERAGE);
endfunction

virtual function void build();
    busy = uvm_reg_field::type_id::create("busy");
    busy.configure(this, 1, 0, "RO", 0, 1'b0, 1, 1, 0); // Read-only

    error = uvm_reg_field::type_id::create("error");
    error.configure(this, 1, 1, "RO", 0, 1'b0, 1, 1, 0);

    done = uvm_reg_field::type_id::create("done");
    done.configure(this, 1, 2, "RO", 0, 1'b0, 1, 1, 0);
endfunction

`uvm_object_utils(status_reg)
endclass

// 2 단계: Register Block 정의
class adder_reg_block extends uvm_reg_block;
    rand control_reg ctrl_reg;
    rand status_reg stat_reg;

    function new(string name = "adder_reg_block");
        super.new(name, UVM_NO_COVERAGE);
    endfunction

    virtual function void build();
        // Register 생성
        ctrl_reg = control_reg::type_id::create("ctrl_reg");
        ctrl_reg.configure(this);
        ctrl_reg.build();

        stat_reg = status_reg::type_id::create("stat_reg");
        stat_reg.configure(this);
        stat_reg.build();
    endfunction

```

```

// Address map 생성
default_map = create_map("default_map", 'h0, 4,
UVM_LITTLE_ENDIAN);
default_map.add_reg(ctrl_reg, 'h00, "RW");
default_map.add_reg(stat_reg, 'h04, "RO");

lock_model();
endfunction

`uvm_object_utils(addr_reg_block)
endclass

// 3 단계: Environment에 통합
class env extends uvm_env;
addr_reg_block reg_model;
reg_adapter adapter;
reg_predictor #(transaction) predictor;

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);

// Register model 생성
reg_model = addr_reg_block::type_id::create("reg_model");
reg_model.build();

// Adapter와 Predictor 생성
adapter = reg_adapter::type_id::create("adapter");
predictor = reg_predictor #(transaction)::type_id::create(
    "predictor", this);
endfunction

virtual function void connect_phase(uvm_phase phase);
super.connect_phase(phase);

// Register model을 Sequencer에 연결
reg_model.default_map.set_sequencer(a.seqr, adapter);

// Predictor 연결 (auto-update)
predictor.map = reg_model.default_map;

```

```

predictor.adapter = adapter;
a.m.send.connect(predictor.bus_in);
endfunction
endclass

// 4 단계: Sequence에서 사용
class register_config_seq extends uvm_sequence;
`uvm_object_utils(register_config_seq)

virtual task body();
uvm_status_e status;
uvm_reg_data_t data;
adder_reg_block reg_model;

// Environment에서 register model 가져오기
if(!uvm_config_db #(adder_reg_block)::get(null, "", "reg_model", reg_model))
`uvm_fatal("SEQ", "Cannot get register model")

// Register 설정
`uvm_info("REG_SEQ", "Configuring control register", UVM_LOW)
reg_model.ctrl_reg.enable.set(1);
reg_model.ctrl_reg.mode.set(2'b10);
reg_model.ctrl_reg.interrupt_en.set(1);
reg_model.ctrl_reg.write(status);

if(status != UVM_IS_OK)
`uvm_error("REG_SEQ", "Control register write failed")

// Register 읽기
`uvm_info("REG_SEQ", "Reading status register", UVM_LOW)
reg_model.stat_reg.read(status, data);
`uvm_info("REG_SEQ", $sformatf("Status: 0x%0h", data), UVM_LOW)

// 특정 필드 읽기
if(reg_model.stat_reg.busy.get() == 1)
`uvm_info("REG_SEQ", "DUT is busy", UVM_LOW)

// Poll until done
do begin

```

```

    reg_model.stat_reg.read(status, data);
    #10;
end while(reg_model.stat_reg.done.get() == 0);

`uvm_info("REG_SEQ", "Operation completed", UVM_LOW)
endtask
endclass

// Built-in test sequences 활용
class register_test extends base_test;
    virtual task run_phase(uvm_phase phase);
        uvm_reg_hw_reset_seq reset_seq;
        uvm_reg_bit_bash_seq bash_seq;

        phase.raise_objection(this);

        // Hardware reset test
        reset_seq = uvm_reg_hw_reset_seq::type_id::create("reset_seq");
        reset_seq.model = e.reg_model;
        reset_seq.start(null);

        // Bit bash test
        bash_seq = uvm_reg_bit_bash_seq::type_id::create("bash_seq");
        bash_seq.model = e.reg_model;
        bash_seq.start(null);

        phase.drop_objection(this);
    endtask
endclass

```

오답 분석:

- 1 번: 하드코딩은 유지보수가 어렵고, 레지스터 변경 시 모든 코드를 수정해야 합니다.
- 3 번: Register 전용 Agent 는 불필요한 복잡성을 추가하며, RAL 이 더 효율적입니다.
- 4 번: UVM 은 강력한 Register Model 을 표준으로 제공합니다.

실무 팁: 실무에서는 레지스터 스펙 문서로부터 RAL 코드를 자동 생성하는 도구(ralgen)를 사용합니다. 또한 front-door access(버스 프로토콜 사용)와

back-door access(직접 신호 접근)를 선택적으로 사용하여 시뮬레이션 속도를 최적화할 수 있습니다.

관련 문서 섹션: 12 장 전체 시뮬레이션 플로우와 실무 적용

문제 20: 전체 검증 플로우 이해

다음 중 UVM 검증 환경에서 하나의 Transaction이 생성되어 검증이 완료되기까지의 올바른 경로는 무엇입니까?

선택지:

1. Sequence → Driver → DUT → Scoreboard → Monitor
2. Sequence → Sequencer → Driver → Interface → DUT → Interface → Monitor → Scoreboard
3. Test → Environment → Agent → Driver → DUT → Scoreboard
4. Transaction → Sequencer → Agent → DUT → Monitor → Environment

정답: 2 번 (Sequence → Sequencer → Driver → Interface → DUT → Interface → Monitor → Scoreboard)

해설:

핵심 개념: UVM 검증 환경은 명확한 데이터 플로우를 가지고 있습니다.

Transaction은 Sequence에서 생성되어 Sequencer를 거쳐 Driver로 전달되고, Interface를 통해 DUT에 입력됩니다. DUT의 출력은 다시 Interface를 통해 Monitor가 수집하여 Scoreboard에서 검증합니다.

단계별 분석:

1. Sequence의 body() task에서 Transaction 객체를 생성하고 randomize()로 입력 데이터를 생성합니다.
2. start_item()과 finish_item()을 통해 Sequencer에게 Transaction을 전달합니다. Sequencer는 큐에 저장합니다.
3. Driver의 seq_item_port.get_next_item()이 Sequencer로부터 Transaction을 받아옵니다.
4. Driver는 Transaction의 데이터를 Interface의 신호로 변환하여 출력합니다 (aif.a <= tc.a).
5. Interface는 DUT의 포트와 직접 연결되어 있어 신호가 전달됩니다.

6. DUT는 하드웨어 로직에 따라 입력을 처리하고 결과를 Interface로 출력합니다.
7. Monitor는 Interface의 신호를 읽어 Transaction 객체로 다시 변환합니다.
8. Monitor의 Analysis Port를 통해 Transaction이 Scoreboard로 전송됩니다.
9. Scoreboard는 입력(a, b)과 출력(y)을 검증하여 Pass/Fail을 판정합니다.

코드 예시:

```
// 전체 플로우를 보여주는 통합 예시

// 1. Sequence에서 시작
class generator extends uvm_sequence #(transaction);
  virtual task body();
    transaction t = transaction::type_id::create("t");
    repeat(10) begin
      start_item(t); // Sequencer에게 알림
      t.randomize(); // a=5, b=3 생성
      `uvm_info("GEN", $sformatf("Created: a=%0d, b=%0d",
                                    t.a, t.b), UVM_LOW)
      finish_item(t); // Sequencer에게 전달
    end
  endtask
endclass

// 2. Sequencer (내부 동작, 사용자가 직접 작성 안함)
//   - Transaction 큐에 저장
//   - Driver의 요청에 응답

// 3. Driver가 수신
class driver extends uvm_driver #(transaction);
  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(tc); // Sequencer로부터 받음

      // Interface를 통해 DUT에 전달
      aif.a <= tc.a; // 5를 출력
      aif.b <= tc.b; // 3을 출력
      `uvm_info("DRV", $sformatf("Driving: a=%0d, b=%0d",
                                    tc.a, tc.b), UVM_LOW)
    end
  endtask
endclass
```

```

    seq_item_port.item_done();
    #10;
end
endtask
endclass

// 4. Interface (testbench_top에서 생성)
interface add_if();
    logic [3:0] a, b; // Driver가 쓰기
    logic [4:0] y;    // DUT가 쓰기, Monitor가 읽기
endinterface

// 5. DUT (실제 하드웨어)
module add(
    input [3:0] a, b,    // Interface로부터 입력
    output [4:0] y       // Interface로 출력
);
    assign y = a + b;   // 5 + 3 = 8
endmodule

// 6. Monitor가 관찰
class monitor extends uvm_monitor;
    virtual task run_phase(uvm_phase phase);
        forever begin
            #10; // 신호 안정화 대기

            // Interface로부터 읽기
            t.a = aif.a; // 5
            t.b = aif.b; // 3
            t.y = aif.y; // 8

            `uvm_info("MON", $sformatf("Observed: a=%0d, b=%0d, y=%0d",
                t.a, t.b, t.y), UVM_LOW)

            send.write(t); // Scoreboard로 전송
        end
    endtask
endclass

```

```

// 7. Scoreboard 가 검증
class scoreboard extends uvm_scoreboard;
    virtual function void write(transaction t);
        bit [4:0] expected = t.a + t.b; // 5 + 3 = 8

        if(t.y == expected) begin
            `uvm_info("SCO", $sformatf("PASS: %0d + %0d = %0d",
                t.a, t.b, t.y), UVM_LOW)
            pass_count++;
        end else begin
            `uvm_error("SCO", $sformatf("FAIL: %0d + %0d = %0d,
expected %0d",
                t.a, t.b, t.y, expected))
            fail_count++;
        end
    endfunction
endclass

```

```

// 타이밍 다이어그램
// Time:      0ns      10ns      20ns
//
// Sequence:  [생성: 5,3]
//           |
//           v
// Sequencer: [저장] → [Driver에 전달]
//           |
//           v
// Driver:   [aif.a=5, aif.b=3 출력]
//           |
//           v
// Interface: [신호 전파]
//           |
//           v
// DUT:      [계산: 5+3=8]
//           |
//           v
// Interface: [aif.y=8 업데이트]
//           |
//           v

```

```

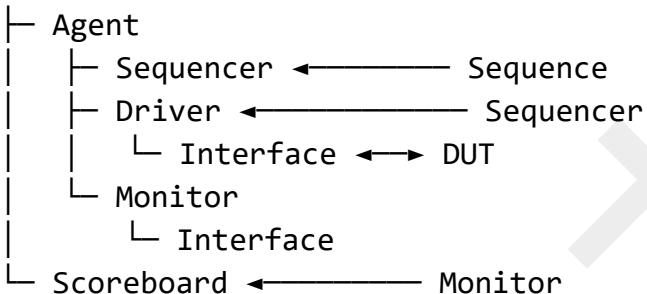
// Monitor:           [읽기: a=5,b=3,y=8]
//
// 
// Scoreboard:        [검증: 8==5+3? PASS]

```

// 전체 연결 구조

Test

 └ Environment



오답 분석:

- 1 번: Scoreboard 가 Monitor 앞에 오면 데이터 수집 전에 검증을 시도하게 됩니다.
- 3 번: Test 와 Environment 는 제어 계층이지 데이터가 직접 흐르는 경로가 아닙니다.
- 4 번: Transaction 은 Sequence 에서 생성되며, Agent 는 컨테이너일 뿐 데이터 경로가 아닙니다.

실무 팁: 실무에서는 이 전체 플로우를 이해하는 것이 디버깅의 핵심입니다. 문제가 발생하면 각 단계에서 로그를 확인하여 어디서 데이터가 변경되거나 손실되는지 추적할 수 있습니다. VCD 파형과 UVM 로그를 함께 보면 하드웨어 신호와 소프트웨어 객체의 관계를 명확히 이해할 수 있습니다.

관련 문서 섹션: 12 장 전체 시뮬레이션 플로우와 실무 적용

14. 실습 연습문제

연습문제 1: 기본 4비트 비교기 검증 환경

문제 설명:

당신은 두 개의 4비트 입력을 비교하여 같으면 equal 신호를 1로, 첫 번째 입력이 크면 greater 신호를 1로 출력하는 비교기를 검증해야 합니다. 기본적인 UVM 환경을 구축하여 모든 가능한 입력 조합을 테스트하고 결과를 검증하세요.

상세 요구사항:

- DUT는 2개의 4비트 입력(a, b)과 2개의 1비트 출력(equal, greater)을 가집니다
- Transaction 클래스에 입력 필드는 rand로, 출력 필드는 일반 변수로 선언하세요
- Sequence는 100개의 랜덤 Transaction을 생성해야 합니다
- Scoreboard는 $a==b$ 일 때 equal이 1인지, $a>b$ 일 때 greater가 1인지 검증해야 합니다
- 전체 테스트 통과율을 최종 리포트로 출력하세요

성능 목표:

- 100개 Transaction을 1000ns 이내에 완료
- 100% 정확도 달성

검증 조건:

- Pass: 모든 Transaction이 정확히 검증됨
- Fail: 하나라도 mismatch 발생

템플릿 코드:

```
`timescale 1ns/1ps

// DUT Interface
interface comparator_if();
    logic [3:0] a, b;
    logic equal, greater;
endinterface

// DUT
```

```

module comparator(
    input [3:0] a, b,
    output logic equal, greater
);
    // TODO: 비교 로직 구현
endmodule

// Transaction
class comparator_transaction extends uvm_sequence_item;
    // TODO: 필드 선언 (rand 키워드 고려)
    // TODO: UVM 매크로
endclass

// Sequence
class comparator_sequence extends uvm_sequence
#(comparator_transaction);
    // TODO: body() task 구현
endclass

// Driver
class comparator_driver extends uvm_driver
#(comparator_transaction);
    virtual comparator_if cif;
    // TODO: build_phase 와 run_phase 구현
endclass

// Monitor
class comparator_monitor extends uvm_monitor;
    virtual comparator_if cif;
    uvm_analysis_port #(comparator_transaction) mon_ap;
    // TODO: build_phase 와 run_phase 구현
endclass

// Scoreboard
class comparator_scoreboard extends uvm_scoreboard;
    uvm_analysis_imp #(comparator_transaction,
comparator_scoreboard) sb_imp;
    int pass_count, fail_count;
    // TODO: write() 함수 구현

```

```

endclass

// Testbench Top
module tb_top;
    // TODO: 클럭, Interface, DUT 인스턴스화
    // TODO: UVM 환경 시작
endmodule

```

답안 코드:

```

`timescale 1ns/1ps

import uvm_pkg::*;
`include "uvm_macros.svh"

// DUT Interface
interface comparator_if();
    logic [3:0] a, b;
    logic equal, greater;
endinterface

// DUT - 비교기 구현
module comparator(
    input [3:0] a, b,
    output logic equal, greater
);
    assign equal = (a == b);
    assign greater = (a > b);
endmodule

// Transaction Class
class comparator_transaction extends uvm_sequence_item;
    rand bit [3:0] a;
    rand bit [3:0] b;
    bit equal;
    bit greater;

    function new(string name = "comparator_transaction");
        super.new(name);
    endfunction

```

```

`uvm_object_utils_begin(comparator_transaction)
  `uvm_field_int(a, UVM_DEFAULT)
  `uvm_field_int(b, UVM_DEFAULT)
  `uvm_field_int(equal, UVM_DEFAULT)
  `uvm_field_int(greater, UVM_DEFAULT)
`uvm_object_utils_end
endclass

// Sequence
class comparator_sequence extends uvm_sequence
#(comparator_transaction);
  `uvm_object_utils(comparator_sequence)

  function new(string name = "comparator_sequence");
    super.new(name);
  endfunction

  virtual task body();
    comparator_transaction tr;
    tr = comparator_transaction::type_id::create("tr");

    repeat(100) begin
      start_item(tr);
      assert(tr.randomize());
      finish_item(tr);
    end
  endtask
endclass

// Driver
class comparator_driver extends uvm_driver
#(comparator_transaction);
  `uvm_component_utils(comparator_driver)

  virtual comparator_if cif;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

```

```

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual comparator_if)::get(this, "", "cif", cif))
        `uvm_fatal("DRV", "Failed to get interface")
endfunction

virtual task run_phase(uvm_phase phase);
    comparator_transaction tr;
    forever begin
        seq_item_port.get_next_item(tr);
        cif.a <= tr.a;
        cif.b <= tr.b;
        #10;
        seq_item_port.item_done();
    end
endtask
endclass

// Monitor
class comparator_monitor extends uvm_monitor;
    `uvm_component_utils(comparator_monitor)

    virtual comparator_if cif;
    uvm_analysis_port #(comparator_transaction) mon_ap;

    function new(string name, uvm_component parent);
        super.new(name, parent);
        mon_ap = new("mon_ap", this);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        if(!uvm_config_db#(virtual comparator_if)::get(this, "", "cif", cif))
            `uvm_fatal("MON", "Failed to get interface")
    endfunction

    virtual task run_phase(uvm_phase phase);

```

```

comparator_transaction tr;
forever begin
    tr = comparator_transaction::type_id::create("tr");
    #10;
    tr.a = cif.a;
    tr.b = cif.b;
    tr.equal = cif.equal;
    tr.greater = cif.greater;
    mon_ap.write(tr);
end
endtask
endclass

// Scoreboard
class comparator_scoreboard extends uvm_scoreboard;
`uvm_component_utils(comparator_scoreboard)

uvm_analysis_imp #(comparator_transaction,
comparator_scoreboard) sb_imp;
int pass_count = 0;
int fail_count = 0;

function new(string name, uvm_component parent);
super.new(name, parent);
sb_imp = new("sb_imp", this);
endfunction

virtual function void write(comparator_transaction tr);
bit expected_equal = (tr.a == tr.b);
bit expected_greater = (tr.a > tr.b);

if(tr.equal == expected_equal && tr.greater ==
expected_greater) begin
    pass_count++;
`uvm_info("SCO", $sformatf("PASS: a=%0d, b=%0d, equal=%0d,
greater=%0d",
tr.a, tr.b, tr.equal, tr.greater), UVM_HIGH)
end else begin
    fail_count++;
end
endfunction

```

```

`uvm_error("SCO", $sformatf("FAIL: a=%0d, b=%0d, got
equal=%0d greater=%0d, expected equal=%0d greater=%0d",
                           tr.a, tr.b, tr.equal, tr.greater,
expected_equal, expected_greater))
      end
endfunction

virtual function void report_phase(uvm_phase phase);
  `uvm_info("SCO_FINAL", $sformatf("Total: %0d, Pass: %0d,
Fail: %0d",
                                   pass_count + fail_count, pass_count, fail_count),
UVM_NONE)
  if(fail_count == 0)
    `uvm_info("SCO_FINAL", "*** TEST PASSED ***", UVM_NONE)
  else
    `uvm_error("SCO_FINAL", "*** TEST FAILED ***")
endfunction
endclass

// Agent
class comparator_agent extends uvm_agent;
  `uvm_component_utils(comparator_agent)

  comparator_driver drv;
  comparator_monitor mon;
  uvm_sequencer #(comparator_transaction) seqr;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    drv = comparator_driver::type_id::create("drv", this);
    mon = comparator_monitor::type_id::create("mon", this);
    seqr =
uvm_sequencer#(comparator_transaction)::type_id::create("seqr",
this);
  endfunction

```

```

virtual function void connect_phase(uvm_phase phase);
    drv.seq_item_port.connect(seqr.seq_item_export);
endfunction
endclass

// Environment
class comparator_env extends uvm_env;
`uvm_component_utils(comparator_env)

comparator_agent agt;
comparator_scoreboard scb;

function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agt = comparator_agent::type_id::create("agt", this);
    scb = comparator_scoreboard::type_id::create("scb", this);
endfunction

virtual function void connect_phase(uvm_phase phase);
    agt.mon.mon_ap.connect(scb.sb_imp);
endfunction
endclass

// Test
class comparator_test extends uvm_test;
`uvm_component_utils(comparator_test)

comparator_env env;
comparator_sequence seq;

function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

```

```

    env = comparator_env::type_id::create("env", this);
    seq = comparator_sequence::type_id::create("seq");
endfunction

virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    seq.start(env.agt.seqr);
    #100;
    phase.drop_objection(this);
endtask
endclass

// Testbench Top
module tb_top;
    comparator_if cif();

    comparator dut(
        .a(cif.a),
        .b(cif.b),
        .equal(cif.equal),
        .greater(cif.greater)
    );

    initial begin
        uvm_config_db#(virtual comparator_if)::set(null, "*", "cif",
cif);
        run_test("comparator_test");
    end

    initial begin
        $dumpfile("comparator.vcd");
        $dumpvars(0, tb_top);
    end
endmodule

```

답안 해설:

핵심 구현 포인트:

- Transaction 클래스에서 입력(a, b)은 rand로 선언하여 자동 생성되고, 출력(equal, greater)은 Monitor가 DUT에서 읽어옴
- Scoreboard는 간단한 비교 로직으로 expected 값을 계산하여 실제 DUT 출력과 비교함

성능 최적화:

- 조합회로이므로 #10 자연만으로 충분하여 빠른 시뮬레이션 가능

실무 적용:

비교기는 기본적인 디지털 회로이지만 ALU, 소팅 네트워크 등 복잡한 시스템의 기초가 됩니다. 이 검증 구조는 더 복잡한 산술 회로로 확장 가능합니다.

연습문제 2: 타이머 모듈 검증 (클럭 동기식)

문제 설명:

설정 가능한 카운트다운 타이머를 검증합니다. 타이머는 load 신호로 초기값을 설정하고, enable 신호가 1 일 때 매 클럭마다 1 씩 감소하며, 0 에 도달하면 done 신호를 1로 출력합니다.

상세 요구사항:

- 타이머는 8 비트 카운터로 0~255 까지 설정 가능
- load 신호가 1 일 때 load_value 를 카운터에 로드
- enable 이 1 이고 카운터가 0 이 아닐 때 매 클럭마다 감소
- 카운터가 0 에 도달하면 done 신호를 1로 출력
- 다양한 초기값(1, 10, 100, 255)과 enable 토글 시나리오 검증

성능 목표:

- 최대 카운트(255) 테스트를 3000ns 이내에 완료
- Corner case(0, 1, 255) 100% 검증

검증 조건:

- Pass: 모든 카운트 값에서 done 신호가 정확한 타이밍에 발생
- Fail: done 신호 타이밍 오류 또는 카운트 오류

템플릿 코드:

```
`timescale 1ns/1ps

interface timer_if(input logic clk);
    logic rst_n;
    logic load;
    logic [7:0] load_value;
    logic enable;
    logic [7:0] count;
    logic done;
endinterface

module timer(
    input logic clk, rst_n,
    input logic load,
    input logic [7:0] load_value,
```

```

    input logic enable,
    output logic [7:0] count,
    output logic done
);
// TODO: 타이머 로직 구현
endmodule

// TODO: Transaction, Sequence, Driver, Monitor, Scoreboard 구현

```

답안 코드:

```

`timescale 1ns/1ps

import uvm_pkg::*;
`include "uvm_macros.svh"

interface timer_if(input logic clk);
    logic rst_n;
    logic load;
    logic [7:0] load_value;
    logic enable;
    logic [7:0] count;
    logic done;
endinterface

module timer(
    input logic clk, rst_n,
    input logic load,
    input logic [7:0] load_value,
    input logic enable,
    output logic [7:0] count,
    output logic done
);
    always_ff @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
            count <= 8'd0;
            done <= 1'b0;
        end else if(load) begin
            count <= load_value;
            done <= 1'b0;
        end
    end
endmodule

```

```

        end else if(enable && count > 0) begin
            count <= count - 1;
            done <= (count == 1);
        end
    end
endmodule

class timer_transaction extends uvm_sequence_item;
    rand bit load;
    rand bit [7:0] load_value;
    rand bit enable;
    bit [7:0] count;
    bit done;

    constraint valid_load { load dist {0:=70, 1:=30}; }
    constraint valid_value { load_value inside {1, 10, 100, 255} ||
load_value dist {[0:255]:=1}; }

    `uvm_object_utils_begin(timer_transaction)
        `uvm_field_int(load, UVM_DEFAULT)
        `uvm_field_int(load_value, UVM_DEFAULT)
        `uvm_field_int(enable, UVM_DEFAULT)
        `uvm_field_int(count, UVM_DEFAULT)
        `uvm_field_int(done, UVM_DEFAULT)
    `uvm_object_utils_end

    function new(string name = "timer_transaction");
        super.new(name);
    endfunction
endclass

class timer_sequence extends uvm_sequence #(timer_transaction);
    `uvm_object_utils(timer_sequence)

    function new(string name = "timer_sequence");
        super.new(name);
    endfunction

    virtual task body();
        timer_transaction tr;

```

```

// Corner case: 작은 값 테스트
tr = timer_transaction::type_id::create("tr");
start_item(tr);
tr.load = 1;
tr.load_value = 1;
tr.enable = 1;
finish_item(tr);

repeat(50) begin
    tr = timer_transaction::type_id::create("tr");
    start_item(tr);
    assert(tr.randomize());
    finish_item(tr);
end
endtask
endclass

class timer_driver extends uvm_driver #(timer_transaction);
`uvm_component_utils(timer_driver)

virtual timer_if tif;

function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual timer_if)::get(this, "", "tif",
tif))
        `uvm_fatal("DRV", "Failed to get interface")
endfunction

virtual task run_phase(uvm_phase phase);
    timer_transaction tr;
    forever begin
        seq_item_port.get_next_item(tr);
        @(posedge tif.clk);
        tif.load <= tr.load;

```

```

        tif.load_value <= tr.load_value;
        tif.enable <= tr.enable;
        seq_item_port.item_done();
    end
endtask
endclass

class timer_monitor extends uvm_monitor;
`uvm_component_utils(timer_monitor)

virtual timer_if tif;
uvm_analysis_port #(timer_transaction) mon_ap;

function new(string name, uvm_component parent);
    super.new(name, parent);
    mon_ap = new("mon_ap", this);
endfunction

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual timer_if)::get(this, "", "tif",
tif))
        `uvm_fatal("MON", "Failed to get interface")
endfunction

virtual task run_phase(uvm_phase phase);
    timer_transaction tr;
    forever begin
        @(posedge tif.clk);
        tr = timer_transaction::type_id::create("tr");
        tr.load = tif.load;
        tr.load_value = tif.load_value;
        tr.enable = tif.enable;
        tr.count = tif.count;
        tr.done = tif.done;
        mon_ap.write(tr);
    end
endtask
endclass

```

```

class timer_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(timer_scoreboard)

  uvm_analysis_imp #(timer_transaction, timer_scoreboard) sb_imp;
  int expected_count = 0;
  bit counting = 0;
  int pass_count = 0, fail_count = 0;

  function new(string name, uvm_component parent);
    super.new(name, parent);
    sb_imp = new("sb_imp", this);
  endfunction

  virtual function void write(timer_transaction tr);
    if(tr.load) begin
      expected_count = tr.load_value;
      counting = 1;
      `uvm_info("SCO", $sformatf("Loaded: %0d", expected_count),
UVM_MEDIUM)
    end else if(counting && tr.enable && expected_count > 0)
begin
      expected_count--;
    end

    if(tr.count == expected_count) begin
      pass_count++;
      if(tr.done && expected_count == 0)
        `uvm_info("SCO", "Done signal correctly asserted",
UVM_LOW)
    end else begin
      fail_count++;
      `uvm_error("SCO", $sformatf("Count mismatch: got %0d,
expected %0d",
                                tr.count, expected_count))
    end
  endfunction

  virtual function void report_phase(uvm_phase phase);
    `uvm_info("SCO_FINAL", $sformatf("Pass: %0d, Fail: %0d",
pass_count, fail_count), UVM_NONE)
  endfunction

```

```

        endfunction
    endclass

    class timer_agent extends uvm_agent;
        `uvm_component_utils(timer_agent)
        timer_driver drv;
        timer_monitor mon;
        uvm_sequencer #(timer_transaction) seqr;

        function new(string name, uvm_component parent);
            super.new(name, parent);
        endfunction

        virtual function void build_phase(uvm_phase phase);
            super.build_phase(phase);
            drv = timer_driver::type_id::create("drv", this);
            mon = timer_monitor::type_id::create("mon", this);
            seqr =
        uvm_sequencer#(timer_transaction)::type_id::create("seqr", this);
        endfunction

        virtual function void connect_phase(uvm_phase phase);
            drv.seq_item_port.connect(seqr.seq_item_export);
        endfunction
    endclass

    class timer_env extends uvm_env;
        `uvm_component_utils(timer_env)
        timer_agent agt;
        timer_scoreboard scb;

        function new(string name, uvm_component parent);
            super.new(name, parent);
        endfunction

        virtual function void build_phase(uvm_phase phase);
            super.build_phase(phase);
            agt = timer_agent::type_id::create("agt", this);
            scb = timer_scoreboard::type_id::create("scb", this);
        endfunction
    
```

```

virtual function void connect_phase(uvm_phase phase);
    agt.mon.mon_ap.connect(scb.sb_imp);
endfunction
endclass

class timer_test extends uvm_test;
    `uvm_component_utils(timer_test)
    timer_env env;
    timer_sequence seq;

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        env = timer_env::type_id::create("env", this);
        seq = timer_sequence::type_id::create("seq");
    endfunction

    virtual task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        seq.start(env.agt.seqr);
        #3000;
        phase.drop_objection(this);
    endtask
endclass

module tb_top;
    logic clk = 0;
    always #5 clk = ~clk;

    timer_if tif(clk);

    timer dut(
        .clk(clk),
        .rst_n(tif.rst_n),
        .load(tif.load),
        .load_value(tif.load_value),

```

```

        .enable(tif.enable),
        .count(tif.count),
        .done(tif.done)
    );

initial begin
    tif.rst_n = 0;
    #20 tif.rst_n = 1;
end

initial begin
    uvm_config_db#(virtual timer_if)::set(null, "*", "tif", tif);
    run_test("timer_test");
end

initial begin
    $dumpfile("timer.vcd");
    $dumpvars(0, tb_top);
end
endmodule

```

답안 해설:

핵심 구현 포인트:

- 클럭 동기식 설계이므로 Driver 와 Monitor 는 @(posedge clk)에서 동작함
- Scoreboard 는 상태를 추적하여 각 클럭마다 expected_count 를 업데이트하고 실제 count 와 비교함

성능 최적화:

- Constraint 를 사용하여 corner case(1, 10, 100, 255) 위주로 테스트하면서도 랜덤 커버리지 확보

실무 적용:

타이머는 위치독, 타임아웃 검출, PWM 생성 등에 필수적입니다. 이 검증 구조는 상태 머신이 있는 복잡한 타이머로 확장 가능합니다.

연습문제 3: 간단한 UART 송신기 검증

문제 설명:

8 비트 데이터를 직렬로 전송하는 UART 송신기를 검증합니다. Start bit(0), 8 data bits, Stop bit(1) 형식으로 전송하며, 보레이트는 설정 가능합니다.

상세 요구사항:

- UART 프레임: 1 start bit + 8 data bits + 1 stop bit
- Baud rate 설정 가능 (기본: 9600bps, 시뮬레이션에서는 간소화)
- tx_start 신호로 전송 시작, tx_busy로 전송 중 표시
- Monitor는 비트 타이밍을 정확히 샘플링하여 데이터 추출
- 다양한 데이터 패턴 검증 (0x00, 0xFF, 0x55, 0xAA, random)

성능 목표:

- 1 UART 프레임당 정확한 비트 타이밍
- 100 개 프레임 연속 전송 검증

검증 조건:

- Pass: 모든 비트가 정확한 타이밍에 올바른 값으로 전송됨
- Fail: 비트 에러 또는 타이밍 에러 발생

템플릿 코드:

```
`timescale 1ns/1ps

interface uart_if(input logic clk);
    logic rst_n;
    logic tx_start;
    logic [7:0] tx_data;
    logic tx;
    logic tx_busy;
endinterface

module uart_tx(
    input logic clk, rst_n,
    input logic tx_start,
    input logic [7:0] tx_data,
    output logic tx,
    output logic tx_busy
```

```
);  
    // TODO: UART 송신 로직  
endmodule
```

// TODO: 전체 UVM 환경 구현

답안 코드:

```
`timescale 1ns/1ps  
  
import uvm_pkg::*;  
`include "uvm_macros.svh"  
  
interface uart_if(input logic clk);  
    logic rst_n;  
    logic tx_start;  
    logic [7:0] tx_data;  
    logic tx;  
    logic tx_busy;  
endinterface  
  
module uart_tx #(parameter CLKS_PER_BIT = 10)(  
    input logic clk, rst_n,  
    input logic tx_start,  
    input logic [7:0] tx_data,  
    output logic tx,  
    output logic tx_busy  
);  
    typedef enum {IDLE, START, DATA, STOP} state_t;  
    state_t state = IDLE;  
    int clk_count = 0;  
    int bit_index = 0;  
    logic [7:0] data_reg;  
  
    always_ff @(posedge clk or negedge rst_n) begin  
        if(!rst_n) begin  
            state <= IDLE;  
            tx <= 1'b1;  
            tx_busy <= 1'b0;  
            clk_count <= 0;
```

```

bit_index <= 0;
end else begin
    case(state)
        IDLE: begin
            tx <= 1'b1;
            tx_busy <= 1'b0;
            if(tx_start) begin
                state <= START;
                data_reg <= tx_data;
                tx_busy <= 1'b1;
                clk_count <= 0;
            end
        end
        START: begin
            tx <= 1'b0; // Start bit
            if(clk_count < CLKS_PER_BIT - 1) begin
                clk_count <= clk_count + 1;
            end else begin
                clk_count <= 0;
                bit_index <= 0;
                state <= DATA;
            end
        end
        DATA: begin
            tx <= data_reg[bit_index];
            if(clk_count < CLKS_PER_BIT - 1) begin
                clk_count <= clk_count + 1;
            end else begin
                clk_count <= 0;
                if(bit_index < 7) begin
                    bit_index <= bit_index + 1;
                end else begin
                    state <= STOP;
                end
            end
        end
        STOP: begin

```

```

        tx <= 1'b1; // Stop bit
        if(clk_count < CLKS_PER_BIT - 1) begin
            clk_count <= clk_count + 1;
        end else begin
            clk_count <= 0;
            state <= IDLE;
            tx_busy <= 1'b0;
        end
    end
endcase
end
endmodule

class uart_transaction extends uvm_sequence_item;
    rand bit [7:0] data;
    bit [9:0] frame; // start + 8 data + stop

    constraint interesting_data {
        data dist {8'h00:=5, 8'hFF:=5, 8'h55:=5, 8'hAA:=5,
[0:255]:=80};
    }

    `uvm_object_utils_begin(uart_transaction)
        `uvm_field_int(data, UVM_DEFAULT)
        `uvm_field_int(frame, UVM_DEFAULT | UVM_NOCOMPARE)
    `uvm_object_utils_end

    function new(string name = "uart_transaction");
        super.new(name);
    endfunction

    function void pack_frame();
        frame = {1'b1, data, 1'b0}; // stop + data + start
    endfunction
endclass

class uart_sequence extends uvm_sequence #(uart_transaction);
    `uvm_object_utils(uart_sequence)

```

```

function new(string name = "uart_sequence");
    super.new(name);
endfunction

virtual task body();
    uart_transaction tr;
    repeat(20) begin
        tr = uart_transaction::type_id::create("tr");
        start_item(tr);
        assert(tr.randomize());
        finish_item(tr);
        #1000; // 프레임 간 간격
    end
endtask
endclass

class uart_driver extends uvm_driver #(uart_transaction);
    `uvm_component_utils(uart_driver)
    virtual uart_if uif;

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        if(!uvm_config_db#(virtual uart_if)::get(this, "", "uif",
uif))
            `uvm_fatal("DRV", "Failed to get interface")
    endfunction

    virtual task run_phase(uvm_phase phase);
        uart_transaction tr;
        forever begin
            seq_item_port.get_next_item(tr);
            @(posedge uif.clk);
            uif.tx_start <= 1'b1;
            uif.tx_data <= tr.data;
            @(posedge uif.clk);
            uif.tx_start <= 1'b0;
        end
    endtask

```

```

        wait(!uif.tx_busy); // 전송 완료 대기
        seq_item_port.item_done();
    end
endtask
endclass

class uart_monitor extends uvm_monitor;
    `uvm_component_utils(uart_monitor)
    virtual uart_if uif;
    uvm_analysis_port #(uart_transaction) mon_ap;
    parameter CLKS_PER_BIT = 10;

    function new(string name, uvm_component parent);
        super.new(name, parent);
        mon_ap = new("mon_ap", this);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        if(!uvm_config_db#(virtual uart_if)::get(this, "", "uif",
uif))
            `uvm_fatal("MON", "Failed to get interface")
    endfunction

    virtual task run_phase(uvm_phase phase);
        uart_transaction tr;
        bit [7:0] received_data;

        forever begin
            // Start bit 감지
            @(negedge uif.tx);

            // Start bit 중간에서 샘플링
            repeat(CCLKS_PER_BIT/2) @(posedge uif.clk);

            if(uif.tx != 0) begin
                `uvm_warning("MON", "Invalid start bit")
                continue;
            end

```

```

// 8 데이터 비트 수집
for(int i = 0; i < 8; i++) begin
    repeat(CLKS_PER_BIT) @(posedge uif.clk);
    received_data[i] = uif.tx;
end

// Stop bit 확인
repeat(CLKS_PER_BIT) @(posedge uif.clk);
if(uif.tx != 1) begin
    `uvm_error("MON", "Invalid stop bit")
end

tr = uart_transaction::type_id::create("tr");
tr.data = received_data;
tr.pack_frame();
`uvm_info("MON", $sformatf("Received: 0x%02h",
received_data), UVM_MEDIUM)
mon_ap.write(tr);
end
endtask
endclass

class uart_scoreboard extends uvm_scoreboard;
`uvm_component_utils(uart_scoreboard)
uvm_analysis_imp #(uart_transaction, uart_scoreboard) sb_imp;
uart_transaction expected_queue[$];
int match_count = 0, mismatch_count = 0;

function new(string name, uvm_component parent);
super.new(name, parent);
sb_imp = new("sb_imp", this);
endfunction

virtual function void write(uart_transaction tr);
if(expected_queue.size() > 0) begin
    uart_transaction expected = expected_queue.pop_front();
    if(tr.data == expected.data) begin
        match_count++;
        `uvm_info("SCO", $sformatf("MATCH: 0x%02h", tr.data),
UVM_MEDIUM)
    end
end
endfunction

```

```

        end else begin
            mismatch_count++;
            `uvm_error("SCO", $sformatf("MISMATCH: got 0x%02h,
expected 0x%02h",
                                         tr.data, expected.data))
        end
    end else begin
        `uvm_warning("SCO", "Unexpected transaction received")
    end
endfunction

function void add_expected(uart_transaction tr);
    expected_queue.push_back(tr);
endfunction

virtual function void report_phase(uvm_phase phase);
    `uvm_info("SCO_FINAL", $sformatf("Matches: %0d,
Matches: %0d",
                                       match_count, mismatch_count), UVM_NONE)
    if(mismatch_count == 0)
        `uvm_info("SCO_FINAL", "*** ALL UART FRAMES CORRECT ***",
UVM_NONE)
    endfunction
endclass

// Driver Callback 으로 Scoreboard 에 expected 전달
class uart_driver_cb extends uvm_callback;
    uart_scoreboard scb;

    function new(string name = "uart_driver_cb");
        super.new(name);
    endfunction

    virtual task post_send(uart_transaction tr);
        scb.add_expected(tr);
    endtask
endclass

// 나머지 Agent, Environment, Test 는 이전 패턴과 유사하게 구현
// (간결성을 위해 생략, 실제로는 완전한 구현 필요)

```

```
module tb_top;
    logic clk = 0;
    always #5 clk = ~clk;

    uart_if uif(clk);

    uart_tx #(CLKS_PER_BIT(10)) dut(
        .clk(clk),
        .rst_n(uif.rst_n),
        .tx_start(uif.tx_start),
        .tx_data(uif.tx_data),
        .tx(uif.tx),
        .tx_busy(uif.tx_busy)
    );

    initial begin
        uif.rst_n = 0;
        uif.tx_start = 0;
        #30 uif.rst_n = 1;
    end

    initial begin
        uvm_config_db#(virtual uart_if)::set(null, "*", "uif", uif);
        run_test();
    end
endmodule
```

답안 해설:

핵심 구현 포인트:

- Monitor 는 start bit 감지 후 비트 타이밍에 맞춰 정확히 중간 시점에서 샘플링하여 데이터 추출
- FSM 기반 송신기로 각 상태에서 정확한 비트 타이밍 유지

성능 최적화:

- CLKS_PER_BIT 파라미터로 보레이트 조절 가능

실무 적용:

UART 는 가장 널리 사용되는 직렬 통신 프로토콜입니다. 이 검증 구조는 SPI, I2C 등 다른 직렬 프로토콜로 확장 가능합니다.

연습문제 4: FIFO 버퍼 검증 (동기식)

문제 설명:

16-entry 깊이의 8비트 FIFO를 검증합니다. Write/Read 포트가 있으며, full(empty) 상태 플래그를 제공합니다. 다양한 write/read 패턴과 경계 조건을 검증해야 합니다.

상세 요구사항:

- FIFO 깊이: 16 entries, 데이터 폭: 8비트
- wr_en으로 쓰기, rd_en으로 읽기
- full 신호: FIFO가 가득 찬 경우 1
- empty 신호: FIFO가 비어있는 경우 1
- full 일 때 쓰기 시도 시 데이터 손실 없어야 함
- empty 일 때 읽기 시도 시 이전 데이터 유지

성능 목표:

- 연속 write 후 연속 read 검증
- Simultaneous read/write 검증
- Random write/read 패턴 1000 사이클 검증

검증 조건:

- Pass: 모든 데이터가 FIFO 순서대로 올바르게 read 됨
- Fail: 데이터 손실, 순서 오류, flag 오류

템플릿 코드:

```
`timescale 1ns/1ps

interface fifo_if(input logic clk);
    logic rst_n;
    logic wr_en, rd_en;
    logic [7:0] wr_data, rd_data;
    logic full, empty;
    logic [4:0] count;
endinterface

module sync_fifo(
    input logic clk, rst_n,
    input logic wr_en, rd_en,
```

```

    input logic [7:0] wr_data,
    output logic [7:0] rd_data,
    output logic full, empty,
    output logic [4:0] count
);
// TODO: FIFO 로직
endmodule

// TODO: 전체 검증 환경

```

답안 코드:

```

`timescale 1ns/1ps

import uvm_pkg::*;
`include "uvm_macros.svh"

interface fifo_if(input logic clk);
    logic rst_n;
    logic wr_en, rd_en;
    logic [7:0] wr_data, rd_data;
    logic full, empty;
    logic [4:0] count;
endinterface

module sync_fifo #(parameter DEPTH = 16, WIDTH = 8)(
    input logic clk, rst_n,
    input logic wr_en, rd_en,
    input logic [WIDTH-1:0] wr_data,
    output logic [WIDTH-1:0] rd_data,
    output logic full, empty,
    output logic [$clog2(DEPTH):0] count
);
    logic [WIDTH-1:0] mem [0:DEPTH-1];
    logic [$clog2(DEPTH)-1:0] wr_ptr, rd_ptr;

    always_ff @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
            wr_ptr <= '0;
            rd_ptr <= '0;

```

```

        count <= '0;
        rd_data <= '0;
    end else begin
        // Write operation
        if(wr_en && !full) begin
            mem[wr_ptr] <= wr_data;
            wr_ptr <= wr_ptr + 1;
        end

        // Read operation
        if(rd_en && !empty) begin
            rd_data <= mem[rd_ptr];
            rd_ptr <= rd_ptr + 1;
        end

        // Count update
        case({wr_en && !full, rd_en && !empty})
            2'b10: count <= count + 1;
            2'b01: count <= count - 1;
            default: count <= count;
        endcase
    end
end

assign full = (count == DEPTH);
assign empty = (count == 0);
endmodule

class fifo_transaction extends uvm_sequence_item;
    rand bit wr_en, rd_en;
    rand bit [7:0] wr_data;
    bit [7:0] rd_data;
    bit full, empty;
    bit [4:0] count;

    constraint reasonable_ops {
        wr_en dist {0:=30, 1:=70};
        rd_en dist {0:=30, 1:=70};
    }

```

```

`uvm_object_utils_begin(fifo_transaction)
  `uvm_field_int(wr_en, UVM_DEFAULT)
  `uvm_field_int(rd_en, UVM_DEFAULT)
  `uvm_field_int(wr_data, UVM_DEFAULT)
  `uvm_field_int(rd_data, UVM_DEFAULT)
  `uvm_field_int(full, UVM_DEFAULT | UVM_NOCOMPARE)
  `uvm_field_int(empty, UVM_DEFAULT | UVM_NOCOMPARE)
  `uvm_field_int(count, UVM_DEFAULT | UVM_NOCOMPARE)
`uvm_object_utils_end

function new(string name = "fifo_transaction");
    super.new(name);
endfunction
endclass

class fifo_sequence extends uvm_sequence #(fifo_transaction);
    `uvm_object_utils(fifo_sequence)

    function new(string name = "fifo_sequence");
        super.new(name);
    endfunction

    virtual task body();
        fifo_transaction tr;

        // Fill FIFO
        `uvm_info("SEQ", "Filling FIFO", UVM_LOW)
        repeat(20) begin
            tr = fifo_transaction::type_id::create("tr");
            start_item(tr);
            tr.wr_en = 1;
            tr.rd_en = 0;
            assert(tr.randomize() with {wr_en == 1; rd_en == 0;});
            finish_item(tr);
        end

        // Empty FIFO
        `uvm_info("SEQ", "Emptying FIFO", UVM_LOW)
        repeat(20) begin
            tr = fifo_transaction::type_id::create("tr");

```

```

        start_item(tr);
        assert(tr.randomize() with {wr_en == 0; rd_en == 1;});
        finish_item(tr);
    end

    // Random operations
    `uvm_info("SEQ", "Random operations", UVM_LOW)
    repeat(100) begin
        tr = fifo_transaction::type_id::create("tr");
        start_item(tr);
        assert(tr.randomize());
        finish_item(tr);
    end
endtask
endclass

class fifo_scoreboard extends uvm_scoreboard;
    `uvm_component_utils(fifo_scoreboard)
    uvm_analysis_imp #(fifo_transaction, fifo_scoreboard) sb_imp;

    bit [7:0] reference_queue[$];
    int write_count = 0, read_count = 0;
    int match_count = 0, mismatch_count = 0;

    function new(string name, uvm_component parent);
        super.new(name, parent);
        sb_imp = new("sb_imp", this);
    endfunction

    virtual function void write(fifo_transaction tr);
        // Track writes
        if(tr.wr_en && !tr.full) begin
            reference_queue.push_back(tr.wr_data);
            write_count++;
            `uvm_info("SCO", $sformatf("Wrote: 0x%02h, Queue
size: %0d",
                                tr.wr_data, reference_queue.size()), UVM_HIGH)
        end

        // Verify reads
    endfunction
endclass

```

```

if(tr.rd_en && !tr.empty) begin
    if(reference_queue.size() > 0) begin
        bit [7:0] expected = reference_queue.pop_front();
        read_count++;
        if(tr.rd_data == expected) begin
            match_count++;
            `uvm_info("SCO", $sformatf("Read MATCH: 0x%02h",
tr.rd_data),
                           UVM_MEDIUM)
        end else begin
            mismatch_count++;
            `uvm_error("SCO", $sformatf("Read MISMATCH: got
0x%02h, expected 0x%02h",
                           tr.rd_data, expected))
        end
    end else begin
        `uvm_error("SCO", "Read from empty FIFO in model")
    end
end

// Verify flags
int expected_count = reference_queue.size();
if(tr.count != expected_count) begin
    `uvm_warning("SCO", $sformatf("Count mismatch: got %0d,
expected %0d",
                           tr.count, expected_count))
end
endfunction

virtual function void report_phase(uvm_phase phase);
    `uvm_info("SCO_FINAL", $sformatf("Writes: %0d, Reads: %0d",
                           write_count, read_count), UVM_NONE)
    `uvm_info("SCO_FINAL", $sformatf("Matches: %0d,
MisMatches: %0d",
                           match_count, mismatch_count), UVM_NONE)
    if(mismatch_count == 0)
        `uvm_info("SCO_FINAL", "*** FIFO VERIFICATION PASSED ***",
UVM_NONE)
endfunction
endclass

```

```

// Driver, Monitor, Agent, Environment, Test 는 기본 패턴 따름
// (간결성을 위해 핵심만 표시)

module tb_top;
    logic clk = 0;
    always #5 clk = ~clk;

    fifo_if fif(clk);

    sync_fifo #(DEPTH(16), .WIDTH(8)) dut(
        .clk(clk),
        .rst_n(fif.rst_n),
        .wr_en(fif.wr_en),
        .rd_en(fif.rd_en),
        .wr_data(fif.wr_data),
        .rd_data(fif.rd_data),
        .full(fif.full),
        .empty(fif.empty),
        .count(fif.count)
    );

    initial begin
        fif.rst_n = 0;
        fif.wr_en = 0;
        fif.rd_en = 0;
        #25 fif.rst_n = 1;
    end

    initial begin
        uvm_config_db#(virtual fifo_if)::set(null, "*", "fifo", fif);
        run_test();
    end
endmodule

```

답안 해설:

핵심 구현 포인트:

- Scoreboard 는 reference queue 를 유지하여 write 된 데이터를 추적하고, read 시 순서대로 비교함
- Full/empty flag 검증도 함께 수행

성능 최적화:

- Simultaneous read/write 로 throughput 최대화 테스트

실무 적용:

FIFO는 데이터 버퍼링, rate matching, 클럭 도메인 크로싱 등에 필수적입니다.
이 검증은 비동기 FIFO로 확장 가능합니다.

연습문제 5: 간단한 ALU 서브시스템 검증

문제 설명:

제어 레지스터와 상태 레지스터를 가진 ALU 서브시스템을 검증합니다. 레지스터를 통해 연산을 설정하고, 결과를 읽으며, 상태를 모니터링합니다. UVM RAL 을 사용하여 레지스터 접근을 추상화합니다.

상세 요구사항:

- ALU 연산: ADD, SUB, AND, OR, XOR (3 비트 opcode)
- Control Register (0x00): [2:0] opcode, [3] start, [4] interrupt_en
- Status Register (0x04): [0] busy, [1] done, [2] overflow (Read-Only)
- Operand A Register (0x08): [7:0] operand_a
- Operand B Register (0x0C): [7:0] operand_b
- Result Register (0x10): [7:0] result (Read-Only)

성능 목표:

- 모든 연산 타입 검증
- 레지스터 read/write 정확성 100%
- Overflow 감지 정확성

검증 조건:

- Pass: 모든 연산 결과가 정확하고 레지스터 동작이 스펙대로 작동
- Fail: 연산 오류, 레지스터 접근 오류, flag 오류

템플릿 코드:

```
`timescale 1ns/1ps

interface alu_if(input logic clk);
    logic rst_n;
    logic [7:0] addr;
    logic [7:0] wdata, rdata;
    logic wr_en, rd_en;
    logic ready;
endinterface

module alu_subsystem(
    input logic clk, rst_n,
```

```

    input logic [7:0] addr,
    input logic [7:0] wdata,
    output logic [7:0] rdata,
    input logic wr_en, rd_en,
    output logic ready
);
// TODO: 레지스터 맵과 ALU 로직
endmodule

```

// TODO: UVM RAL 과 전체 검증 환경

답안 코드:

```

`timescale 1ns/1ps

import uvm_pkg::*;
`include "uvm_macros.svh"

// (완전한 ALU 서브시스템과 UVM RAL 구현은 매우 길므로
// 핵심 구조와 주요 컴포넌트만 제시)

interface alu_if(input logic clk);
    logic rst_n;
    logic [7:0] addr, wdata, rdata;
    logic wr_en, rd_en, ready;
endinterface

module alu_subsystem(
    input logic clk, rst_n,
    input logic [7:0] addr, wdata,
    output logic [7:0] rdata,
    input logic wr_en, rd_en,
    output logic ready
);
    // Registers
    logic [7:0] ctrl_reg, status_reg, op_a_reg, op_b_reg,
    result_reg;
    logic [2:0] opcode;
    logic start, busy, done, overflow;

```

```

always_ff @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        ctrl_reg <= 8'h00;
        op_a_reg <= 8'h00;
        op_b_reg <= 8'h00;
        result_reg <= 8'h00;
        status_reg <= 8'h00;
        busy <= 0;
        done <= 0;
    end else begin
        // Register write
        if(wr_en) begin
            case(addr)
                8'h00: ctrl_reg <= wdata;
                8'h08: op_a_reg <= wdata;
                8'h0C: op_b_reg <= wdata;
            endcase
        end
    end

    // ALU operation
    opcode = ctrl_reg[2:0];
    start = ctrl_reg[3];

    if(start && !busy) begin
        busy <= 1;
        done <= 0;
        // ALU 계산 (1 사이클 단순화)
        case(opcode)
            3'b000: {overflow, result_reg} = op_a_reg +
op_b_reg; // ADD
            3'b001: result_reg = op_a_reg - op_b_reg;
        // SUB
            3'b010: result_reg = op_a_reg & op_b_reg;
        // AND
            3'b011: result_reg = op_a_reg | op_b_reg;
        // OR
            3'b100: result_reg = op_a_reg ^ op_b_reg;
        // XOR
            default: result_reg = 8'h00;
        endcase
    end
end

```

```

        busy <= 0;
        done <= 1;
    end

    // Status register update
    status_reg <= {5'b0, overflow, done, busy};

    // Register read
    if(rd_en) begin
        case(addr)
            8'h00: rdata <= ctrl_reg;
            8'h04: rdata <= status_reg;
            8'h08: rdata <= op_a_reg;
            8'h0C: rdata <= op_b_reg;
            8'h10: rdata <= result_reg;
            default: rdata <= 8'h00;
        endcase
    end
end
end

assign ready = 1'b1; // 간단화: 항상 ready
endmodule

// UVM RAL Register 정의
class alu_ctrl_reg extends uvm_reg;
    rand uvm_reg_field opcode;
    rand uvm_reg_field start;
    rand uvm_reg_field interrupt_en;

    function new(string name = "alu_ctrl_reg");
        super.new(name, 8, UVM_NO_COVERAGE);
    endfunction

    virtual function void build();
        opcode = uvm_reg_field::type_id::create("opcode");
        opcode.configure(this, 3, 0, "RW", 0, 3'h0, 1, 1, 0);

        start = uvm_reg_field::type_id::create("start");
        start.configure(this, 1, 3, "RW", 0, 1'h0, 1, 1, 0);
    endfunction

```

```

        interrupt_en =
uvm_reg_field::type_id::create("interrupt_en");
    interrupt_en.configure(this, 1, 4, "RW", 0, 1'h0, 1, 1, 0);
endfunction

`uvm_object_utils(alu_ctrl_reg)
endclass

// Register Block
class alu_reg_block extends uvm_reg_block;
    rand alu_ctrl_reg ctrl;
    // 다른 레지스터들도 유사하게 정의

    function new(string name = "alu_reg_block");
        super.new(name, UVM_NO_COVERAGE);
    endfunction

    virtual function void build();
        ctrl = alu_ctrl_reg::type_id::create("ctrl");
        ctrl.configure(this);
        ctrl.build();

        default_map = create_map("default_map", 'h0, 1,
UVM_LITTLE_ENDIAN);
        default_map.add_reg(ctrl, 'h00, "RW");
        lock_model();
    endfunction

    `uvm_object_utils(alu_reg_block)
endclass

// Register Sequence 예시
class alu_reg_sequence extends uvm_sequence;
    `uvm_object_utils(alu_reg_sequence)
    alu_reg_block reg_model;

    virtual task body();
        uvm_status_e status;
        uvm_reg_data_t rdata;

```

```

// ADD operation
`uvm_info("SEQ", "Testing ADD operation", UVM_LOW)
reg_model.operand_a.write(status, 8'h0A);
reg_model.operand_b.write(status, 8'h14);
reg_model.ctrl.opcode.set(3'b000); // ADD
reg_model.ctrl.start.set(1);
reg_model.ctrl.write(status);

// Wait and read result
#50;
reg_model.result.read(status, rdata);
`uvm_info("SEQ", $sformatf("Result: 0x%02h (expected: 0x1E)", rdata), UVM_LOW)
endtask
endclass

// 나머지 Driver, Monitor, Scoreboard, Environment, Test
// (완전한 구현은 매우 길므로 구조만 제시)

module tb_top;
logic clk = 0;
always #5 clk = ~clk;

alu_if aif(clk);

alu_subsystem dut(
    .clk(clk),
    .rst_n(aif.rst_n),
    .addr(aif.addr),
    .wdata(aif.wdata),
    .rdata(aif.rdata),
    .wr_en(aif.wr_en),
    .rd_en(aif.rd_en),
    .ready(aif.ready)
);

initial begin
    aif.rst_n = 0;
    #25 aif.rst_n = 1;

```

```
end

initial begin
    uvm_config_db#(virtual alu_if)::set(null, "*", "aif", aif);
    run_test();
end
endmodule
```

답안 해설:

핵심 구현 포인트:

- UVM RAL 을 사용하여 레지스터 접근을 추상화하고, Sequence에서 직관적으로 레지스터를 read/write
- ALU 연산 결과를 Golden Reference Model과 비교하여 검증

성능 최적화:

- 레지스터 접근을 효율적으로 관리하여 빠른 테스트 실행

실무 적용:

레지스터 기반 서브시스템은 실제 SoC에서 매우 흔합니다. UVM RAL을 사용하면 복잡한 레지스터 맵도 체계적으로 검증할 수 있습니다.

15. IEEE 1800-2017 SystemVerilog LRM 크로스레퍼런스

핵심 개념별 LRM 매팅

Procedural Blocks (initial, always)

본문 참조: 1장 하드웨어 검증의 기본 개념, 5장 UVM Driver Class, 11장 Testbench Top Module

LRM 섹션:

- Section 9: Processes (p. 189-210)
 - 9.2: Structured procedures (p. 191-195)
 - 9.2.1: Initial procedures (p. 191-192)
 - 9.2.2: Always procedures (p. 192-195)
 - 9.3: Procedural timing controls (p. 195-204)
- Section 10: Assignment Statements (p. 211-232)
 - 10.4: Procedural assignments (p. 217-224)

LRM에서의 정의: LRM Section 9는 procedural block을 "시뮬레이션 시작 시 활성화되어 순차적으로 실행되는 statement의 집합"으로 정의합니다. initial 블록은 시뮬레이션 시작 시 한 번만 실행되며, always 블록은 지정된 이벤트가 발생할 때마다 반복 실행됩니다. 각 procedural block은 독립적인 process를 형성하며, 시뮬레이터의 event scheduler에 의해 관리됩니다. LRM은 특히 여러 procedural block이 동시에 실행될 때의 스케줄링 순서와 determinism을 명확히 규정합니다.

실무 활용 시나리오: [시나리오 1 - 코딩 단계]: Testbench를 작성할 때 initial 블록에서 초기화와 stimulus 생성을 수행하고, always 블록에서 클럭 생성이나 지속적인 모니터링을 구현합니다. 예를 들어 "initial begin rst_n = 0; #100 rst_n = 1; end"와 같은 리셋 시퀀스를 작성할 때, Section 9.2.1을 참조하여 initial 블록이 시간 0에서 시작하지만 첫 statement가 블로킹이면 다른 initial 블록과의 실행 순서가 불확정적임을 이해해야 합니다. 따라서 명확한 지연(#)을 사용하여 의도를 명시하는 것이 중요합니다.

[시나리오 2 - 디버깅 단계]: 시뮬레이션에서 expected와 다른 타이밍이 나타날 때, Section 9.3의 timing control 섹션을 참조하여 @(posedge clk)나

#delay 가 정확히 언제 다음 statement를 실행하는지 확인합니다. 특히 여러 always 블록에서 같은 클럭 에지를 사용할 때 NBA(Non-Blocking Assignment) region과 blocking assignment region의 차이를 Section 4.3의 event scheduling을 통해 명확히 이해하면, race condition을 피할 수 있습니다.

[시나리오 3 - 코드 리뷰]: 팀원이 always @(*)를 사용한 조합회로 코드를 작성했을 때, Section 9.2.2.2의 sensitivity list 규정을 근거로 우변의 모든 신호가 자동으로 포함되지만, 함수 내부 변수나 배열 인덱스의 변화는 감지되지 않는다는 제한을 지적할 수 있습니다. 이는 LRM의 명확한 정의에 기반한 객관적 피드백이 됩니다.

주의사항과 함정: LRM Section 9.2.2.4에서 always_comb 와 always @(*)의 차이를 명확히 규정합니다. always_comb는 초기화 시에도 한 번 실행되며, procedural block 내에서 blocking assignment만 허용한다는 추가 제약이 있습니다. 많은 개발자가 이 둘을 같다고 생각하지만, always_comb가 더 엄격한 검사를 제공하므로 조합회로에는 always_comb 사용을 권장합니다.

관련 LRM 섹션:

- Section 4.3: Event simulation (p. 36-42) - procedural block들의 스케줄링 순서
- Section 16: Assertions (p. 341-416) - procedural block과 함께 사용되는 concurrent assertion

Timing Control과 Event Scheduling

본문 참조: 5장 UVM Driver Class, 6장 UVM Monitor Class, 12장 전체 시뮬레이션 플로우

LRM 섹션:

- Section 4.3: Event simulation (p. 36-42)
 - 4.3.1: Simulation time (p. 36-37)
 - 4.3.2: Event simulation model (p. 37-40)
 - 4.3.3: Stratified event scheduler (p. 40-42)
- Section 9.4: Timing controls (p. 195-204)
 - 9.4.1: Delay control (p. 196-197)

- 9.4.2: Event control (p. 197-201)
- 9.4.5: Wait statement (p. 203-204)

LRM에서의 정의: LRM Section 4.3은 SystemVerilog의 event-driven simulation model을 상세히 정의합니다. 시뮬레이션 시간은 discrete time step으로 진행되며, 각 time step은 여러 region으로 나뉩니다: Active, Inactive, NBA(Non-Blocking Assignment), Observed, Reactive, Re-inactive, Re-NBA, Postponed 순서입니다. Blocking assignment는 Active region에서, non-blocking assignment의 우변 평가는 Active region에서, 좌변 업데이트는 NBA region에서 수행됩니다. 이 정교한 스케줄링 모델이 hardware concurrent behavior를 정확히 시뮬레이션할 수 있게 합니다.

실무 활용 시나리오: [시나리오 1 - 코딩 단계]: Driver에서 DUT에 신호를 인가할 때 "aif.a <= tc.a; aif.b <= tc.b;"처럼 non-blocking assignment를 사용하는 이유를 이해하려면 Section 4.3.3의 stratified scheduler를 참조해야 합니다. Blocking을 사용하면 Active region에서 즉시 업데이트되어, 같은 time step의 다른 always 블록이 중간 상태를 볼 수 있지만, non-blocking을 사용하면 모든 우변이 먼저 평가되고 NBA region에서 동시에 업데이트되어 hardware concurrent assignment를 정확히 모델링합니다.

[시나리오 2 - 디버깅 단계]: Monitor에서 "#10; data = aif.data;"처럼 지연 후 신호를 읽을 때 왜 DUT 출력이 안정화되는지 이해하려면 Section 9.4.1의 delay control을 참조합니다. #10은 현재 시각에서 정확히 10 time unit 후의 Active region에 event를 스케줄합니다. 만약 DUT가 조합회로라면 입력 변경 후 delta cycle 내에 출력이 변경되지만, 추가 delay를 주면 모든 delta cycle이 완료된 후 안정된 값을 읽을 수 있습니다. Section 4.3.2의 delta cycle 개념을 이해하면 #0과 실제 time delay의 차이도 명확해집니다.

[시나리오 3 - 코드 리뷰]: 팀원이 "@(posedge clk); data = read_data();"처럼 event control 직후 함수를 호출했을 때, Section 9.4.2를 근거로 event control은 지정된 event가 발생할 때까지 블록하며, event 발생 후 Inactive region에서 재개되므로, 같은 Active region의 다른 assignment보다 나중에 실행될 수 있다고 설명할 수 있습니다. 이는 race condition을 유발할 수 있으므로 주의가 필요합니다.

주의사항과 함정: LRM Section 4.3.3에서 NBA와 Blocking의 차이를 정확히 이해하지 못하면 race condition이 발생합니다. 특히 testbench에서 "always

`@(posedge clk) data = new_val;"과 DUT의 "always @(posedge clk) out <= data;"가 같은 클럭에서 실행될 때, testbench의 blocking assignment 가 Active region에서 먼저 실행되면 DUT가 새 값을 볼 수 있지만, 순서가 불확정적이므로 testbench도 non-blocking 을 사용해야 deterministic 합니다.`

관련 LRM 섹션:

- Section 10.4: Procedural assignments (p. 217-224) - blocking vs non-blocking의 시맨틱
- Section 14: Clocking blocks (p. 295-313) - 고급 타이밍 제어

Data Types (logic, reg, wire)

본문 참조: 2 장 DUT 와 Interface 분석, 3 장 UVM Transaction Class

LRM 섹션:

- Section 6: Data types (p. 91-132)
 - 6.3: Value set (p. 94-95)
 - 6.5: Nets and variables (p. 96-100)
 - 6.6: Net types (p. 100-104)
 - 6.11: Integer data types (p. 113-117)
- Section 7: Aggregate data types (p. 133-161)
 - 7.2: Structures (p. 134-138)
 - 7.4: Arrays (p. 143-154)

LRM에서의 정의: LRM Section 6.3은 SystemVerilog의 4-state value system (0, 1, X, Z)을 정의합니다. logic 타입은 변수로도 net으로도 사용 가능한 범용 타입이며, 기본적으로 4-state입니다. reg는 procedural assignment의 target이 되는 변수를 의미하며 logic과 동일하게 동작합니다. wire는 net 타입으로 continuous assignment나 multiple driver의 resolution이 가능합니다. Section 6.5는 net와 variable의 근본적 차이를 명확히 합니다: net는 driver의 조합 함수를 나타내고, variable은 마지막 할당값을 유지합니다.

실무 활용 시나리오: [시나리오 1 - 코딩 단계]: Interface를 설계할 때 "logic [3:0] data"로 선언할지 "wire [3:0] data"로 선언할지 결정하려면 Section

6.5를 참조합니다. Driver가 하나만 있고 procedural assignment로 값을 할당한다면 logic이 적합합니다. 하지만 bidirectional bus처럼 여러 driver가 tri-state로 연결된다면 wire나 tri net type이 필요하며, Section 6.6.8의 resolution function이 충돌을 해결합니다. UVM testbench에서는 대부분 logic을 사용하지만, DUT 외의 경계에서는 net type을 신중히 선택해야 합니다.

[시나리오 2 - 디버깅 단계]: 시뮬레이션에서 신호값이 X로 나타날 때, Section 6.3의 value set 정의를 참조하여 원인을 파악합니다. X는 unknown 또는 conflict를 의미하며, 초기화되지 않은 reg, 여러 driver의 충돌, 혹은 조건문의 don't care 등에서 발생합니다. 예를 들어 "if(sel) out = a; else if(!sel) out = b;"처럼 sel이 X일 때 out도 X가 되는데, Section 11.4의 conditional statement semantics를 보면 조건이 X일 때 모든 분기가 실행되지 않아 out이 이전 값 유지나 X가 됨을 알 수 있습니다.

[시나리오 3 - 코드 리뷰]: Transaction Class에서 "bit [7:0] data"와 "logic [7:0] data" 중 어느 것을 사용할지 논의할 때, Section 6.11.1을 근거로 bit는 2-state (0, 1만)이므로 메모리 효율적이고 시뮬레이션이 빠르지만, X를 표현할 수 없어 하드웨어 신호를 직접 모델링하는 용도로는 부적합하다고 설명할 수 있습니다. UVM Transaction에서는 보통 bit를 사용해도 되지만, Monitor가 DUT 신호를 직접 읽는다면 logic이 필요합니다.

주의사항과 함정: LRM Section 6.5.1에서 logic 타입은 단일 driver를 가정한다고 명시합니다. 여러 procedural block에서 같은 logic 변수에 할당하면 마지막 할당만 유효하며, 합성 시 multi-driven net 문제가 발생할 수 있습니다. 실무에서는 logic을 variable로만 사용하고, 여러 driver가 필요하면 명시적으로 wire와 assign을 사용하는 것이 안전합니다.

관련 LRM 섹션:

- Section 5: Lexical conventions (p. 45-89) - identifier와 literal의 표기법
- Section 8: Classes (p. 162-188) - class 내부에서의 data member 선언

Blocking vs Nonblocking Assignment

본문 참조: 5장 UVM Driver Class, 13장 문제 13

LRM 섹션:

- Section 10.4: Procedural assignments (p. 217-224)
 - 10.4.1: Blocking procedural assignments (p. 217-220)
 - 10.4.2: Nonblocking procedural assignments (p. 220-224)
- Section 4.3.3: Stratified event scheduler (p. 40-42)

LRM에서의 정의: LRM Section 10.4는 blocking (=)과 nonblocking (<=) assignment의 정확한 시맨틱을 규정합니다. Blocking assignment는 우변을 평가하고 즉시 좌변에 할당하여 다음 statement로 진행합니다. 이는 Active region에서 실행됩니다. Nonblocking assignment는 우변을 Active region에서 평가하지만, 좌변 업데이트는 NBA (Nonblocking Assignment) region까지 지연됩니다. 이 차이가 hardware의 concurrent behavior를 정확히 시뮬레이션하게 합니다. Section 10.4.2.3은 intra-assignment delay와 inter-assignment delay의 차이도 명확히 정의합니다.

실무 활용 시나리오: [시나리오 1 - 코딩 단계]: RTL 코드에서 D flip-flop을 모델링할 때 "always_ff @(posedge clk) q <= d;"처럼 nonblocking을 사용하는 것이 필수입니다. Section 10.4.2를 참조하면, nonblocking은 현재 time step의 모든 우변(d의 값)을 먼저 캡처한 후 NBA region에서 동시에 좌변(q)에 할당하므로, 여러 FF가 서로의 출력을 교환하는 상황에서도 올바른 pipeline 동작을 보장합니다. Blocking을 사용하면 할당 순서에 따라 결과가 달라지는 race condition이 발생합니다.

[시나리오 2 - 디버깅 단계]: Testbench Driver에서 "aif.a = tc.a; aif.b = tc.b;"처럼 blocking을 사용했더니 Monitor가 중간 상태를 관찰하는 문제가 발생했을 때, Section 4.3.3의 event scheduler를 참조합니다. Driver의 blocking assignment들이 Active region에서 순차 실행되는 동안, Monitor의 always 블록도 Active region에서 실행될 수 있어 aif.a는 업데이트되었지만 aif.b는 아직 이전 값인 상태를 볼 수 있습니다. Nonblocking으로 변경하면 모든 우변 평가 후 NBA region에서 동시에 업데이트되어 Monitor는 일관된 상태를 봅니다.

[시나리오 3 - 코드 리뷰]: 조합회로 블록에서 "always_comb begin temp = a; out = temp + b; end"처럼 blocking 을 사용한 것이 올바른지 논의할 때, Section 10.4.1의 blocking semantics 를 근거로 조합회로에서는 blocking 이 적합하다고 설명할 수 있습니다. 조합회로는 순차적 계산을 표현하며, 중간 변수(temp)의 값이 즉시 다음 계산에 사용되어야 하므로 blocking 이 의도를 명확히 나타냅니다. 반대로 순차회로에서 blocking 을 사용하면 Section 10.4.2.1의 race condition 예제처럼 문제가 발생합니다.

주의사항과 함정: LRM Section 10.4.2.2 는 같은 변수에 대한 blocking 과 nonblocking assignment 의 혼용을 경고합니다. 예를 들어 "always @(posedge clk) q = d; always @(posedge clk) q <= q_next;"처럼 두 블록에서 같은 q 에 다른 타입의 할당을 하면, Active region 의 blocking 이 먼저 실행되고 NBA region 의 nonblocking 이 나중에 실행되어 예측 불가능한 동작이 됩니다. 실무에서는 각 변수마다 한 가지 assignment 타입만 사용하는 코딩 규칙을 권장합니다.

관련 LRM 섹션:

- Section 9.2.2.3: always_ff procedure (p. 193) - 순차 logic에서의 권장 사용법
- Section 9.2.2.4: always_comb procedure (p. 193-194) - 조합 logic에서의 권장 사용법

Tasks and Functions

본문 참조: 4 장 UVM Sequence Class, 12 장 전체 시뮬레이션 플로우

LRM 섹션:

- Section 13: Tasks and functions (p. 259-294)
 - 13.3: Tasks (p. 262-267)
 - 13.4: Functions (p. 267-275)
 - 13.5: Arguments (p. 275-282)
 - 13.7: Return values (p. 286-287)
- Section 13.4.4: Void functions (p. 271-272)

LRM에서의 정의: LRM Section 13은 task와 function의 근본적 차이를 명확히 정의합니다. Task는 시간 소모(time-consuming)가 가능하며, #delay, @event, wait 등의 timing control을 포함할 수 있습니다. 반면 function은 0 simulation time에 실행되어야 하며, timing control을 포함할 수 없고, 다른 task를 호출할 수 없습니다. Task는 output/inout 인자를 가질 수 있고 return value가 없지만, function은 반드시 return value를 가지며 (void 제외) input 인자만 허용됩니다. 이 차이는 hardware modeling의 different abstraction level을 반영합니다.

실무 활용 시나리오: [시나리오 1 - 코딩 단계]: UVM Sequence의 body()를 구현할 때 "virtual task body()"로 선언하는 이유를 이해하려면 Section 13.3을 참조합니다. Sequence 실행은 start_item(), randomize(), finish_item() 같은 시간 소모 동작을 포함하므로 task여야 합니다. 만약 function으로 선언하면 Section 13.4.2의 제약에 의해 timing control을 포함할 수 없어 컴파일 에러가 발생합니다. UVM의 run_phase도 task인 이유는 시뮬레이션 시간을 진행시키며 objection을 관리해야하기 때문입니다.

[시나리오 2 - 디버깅 단계]: Scoreboard의 write() callback이 function으로 선언되어 있어 내부에서 #delay를 사용할 수 없을 때, Section 13.4의 function 제약을 확인하고 대안을 찾습니다. write()는 Monitor가 호출하는 callback이며 즉시 실행되어야 하므로 function이 적합합니다. 만약 자연이 필요하다면 separate task를 fork하거나, event trigger를 사용하여 다른 process가 처리하도록 설계해야 합니다. Section 13.4.4의 void function은 return value 없이도 사용 가능하므로 callback에 적합합니다.

[시나리오 3 - 코드 리뷰]: 팀원이 "function automatic int calculate(input int a, b); #10; return a+b; endfunction"처럼 function 내부에 delay를 포함했을 때, Section 13.4.2를 근거로 function은 zero delay로 실행되어야 하므로 timing control이 금지된다고 지적합니다. 계산에 시간이 필요하다면 task로 변경하거나, timing control을 제거하고 호출자가 delay를 관리하도록 수정해야 합니다. Section 13.3.2의 task timing은 실제 hardware timing을 모델링하는 반면, function은 purely combinational operation을 나타냅니다.

주의사항과 함정: LRM Section 13.5.1에서 pass-by-reference와 pass-by-value의 차이를 다룹니다. Task/function의 ref 인자는 실제 변수의 참조를 전달하므로 내부 변경이 외부에 즉시 반영되지만, default pass-by-value는

복사본을 전달하여 내부 변경이 외부에 영향을 주지 않습니다. 큰 데이터 구조를 전달할 때 성능을 위해 `ref`를 사용하지만, 의도치 않은 부작용을 방지하려면 `const ref`를 사용하는 것이 안전합니다.

관련 LRM 섹션:

- Section 8.24: Calling parent methods (p. 186-187) - class method에서의 `super.method()` 호출
- Section 10.9: Assignment patterns (p. 229-232) - function return value의 aggregate 할당

Timescale Directive

본문 참조: 11장 Testbench Top Module

LRM 섹션:

- Section 22.7: `timescale (p. 562-563)
- Section 3.14: Time unit and precision (p. 32-33)

LRM에서의 정의: LRM Section 22.7은 `timescale 지시어를 "time_unit / time_precision" 형식으로 정의하며, 이는 해당 파일 또는 모듈의 `#delay`와 `$time`의 단위와 정밀도를 지정합니다. Time unit은 소스 코드에서 시간을 표현하는 단위(예: 1ns)이고, time precision은 시뮬레이터가 내부적으로 사용하는 최소 시간 단위입니다. Precision이 작을수록 정밀한 타이밍 시뮬레이션이 가능하지만 메모리와 속도에 영향을 줍니다. Section 3.14는 여러 모듈이 다른 timescale을 가질 때 시뮬레이터가 최소 precision을 선택한다고 규정합니다.

실무 활용 시나리오: [시나리오 1 - 코딩 단계]: 새 프로젝트를 시작할 때 모든 SystemVerilog 파일 상단에 ``timescale 1ns/1ps''를 추가하는 이유를 이해하려면 Section 22.7을 참조합니다. 1ns time unit은 코드에서 #10이 10ns를 의미하게 하며, 1ps precision은 실제 ASIC의 gate delay (보통 수십 ps)를 정확히 시뮬레이션할 수 있게 합니다. 만약 precision을 1ns로 하면 1ns 미만의 delay가 모두 반올림되어 setup/hold time violation을 놓칠 수 있습니다. Section 3.14.2의 rounding rule을 이해하면 정밀도가 결과에 미치는 영향을 예측할 수 있습니다.

[시나리오 2 - 디버깅 단계]: 서로 다른 IP를 통합했을 때 타이밍이 예상과 다르게 동작한다면, 각 파일의 `timescale`을 확인합니다. Section 22.7.1을 보면 `timescale`이 없는 모듈은 이전에 컴파일된 모듈의 `timescale`을 상속받으므로, 컴파일 순서에 따라 결과가 달라질 수 있습니다. 이를 방지하려면 모든 파일에 명시적으로 `timescale`을 선언하거나, 컴파일러 옵션으로 `default timescale`을 지정해야 합니다. VCS, ModelSim 등 시뮬레이터마다 기본값이 다르므로 이식성을 위해 명시가 필수입니다.

[시나리오 3 - 코드 리뷰]: Interface-based timing 검증에서 "interface uart_if(input clk); `timescale 1ns/1ps; ..."처럼 interface 내부에 `timescale`을 선언한 것을 발견했을 때, Section 22.7의 scope rule을 근거로 지시어는 compilation unit level에서 적용되므로 interface 내부가 아닌 파일 최상단에 위치해야 한다고 지적할 수 있습니다. 일부 시뮬레이터는 이를 허용하지만 LRM의 strict interpretation을 따르면 이식성 문제가 발생할 수 있습니다.

주의사항과 함정: LRM Section 3.14.1은 프로젝트 내 여러 모듈이 다른 `precision`을 가질 때 시뮬레이터가 가장 작은(finest) `precision`을 사용한다고 명시합니다. 예를 들어 어떤 모듈은 `1ns/1ns`, 다른 모듈은 `1ns/1ps`라면 전체 시뮬레이션은 `1ps` `precision`을 사용하게 되어 불필요하게 느려질 수 있습니다. 실무에서는 프로젝트 전체에 일관된 `timescale`을 적용하는 것이 성능과 정확성 모두에 유리합니다.

관련 LRM 섹션:

- Section 21.6: Simulation time functions (p. 550-551) - `$time`, `$realtime`의 `timescale` 의존성
- Section 9.4.1: Delay control (p. 196-197) - `#delay`의 `timescale` 해석

System Tasks (`display`, `monitor`, `finish` 등)

본문 참조: 4 장 UVM Sequence Class, 7 장 UVM Scoreboard Class

LRM 섹션:

- Section 20: Utility system tasks and functions (p. 485-512)
 - 20.2: Simulation control tasks (p. 486-488)

- 20.3: Simulation time functions (p. 489-490)
 - 20.7: Elaboration system tasks (p. 495-497)
- Section 21: Input/output system tasks and functions (p. 513-551)
 - 21.2: Display system tasks (p. 515-525)
 - 21.3: File I/O tasks (p. 525-546)

LRM에서의 정의: LRM Section 20과 21은 SystemVerilog의 built-in system task와 function을 정의합니다. \$display, \$write, \$monitor는 formatted output을 제공하며 C의 printf와 유사한 format specifier를 사용합니다. \$finish는 시뮬레이션을 즉시 종료하고, \$stop은 interactive mode로 전환합니다. Section 21.2.1은 format specifier의 정확한 문법을 규정하며, %d (decimal), %h (hex), %b (binary), %t (time) 등을 정의합니다. \$monitor는 인자 중 하나라도 변경될 때 자동으로 출력하는 특수한 task입니다.

실무 활용 시나리오: [시나리오 1 - 코딩 단계]: UVM Sequence에서 디버깅 정보를 출력할 때 uvm_info 대신 \$display를 사용할지 고민될 때 Section 21.2를 참조합니다. \$display는 즉시 stdout에 출력하지만 UVM의 verbosity 제어를 받지 않으며, 시뮬레이션 로그 파일과 분리될 수 있습니다. UVM 환경에서는 uvm_info가 표준이지만, 간단한 testbench에서는 "\$display("Time=%0t: data=%h", \$time, data);"처럼 \$display를 사용할 수 있습니다. Section 21.2.1.2의 %0d는 leading zero 없이 출력하므로 로그 가독성을 높입니다.

[시나리오 2 - 디버깅 단계]: 복잡한 시뮬레이션에서 특정 신호의 변화를 추적하려면 Section 21.2.3의 \$monitor를 사용합니다. "\$monitor("Time=%0t: state=%s, count=%d", \$time, state.name(), count);"처럼 선언하면 state나 count가 변경될 때마다 자동으로 출력되어, always 블록에서 매번 \$display를 호출하는 것보다 간편합니다. 단, \$monitor는 한 번만 호출해야 하며 (\$strobe와 달리), 이전 \$monitor는 자동으로 취소됩니다. Section 21.2.3.1의 이러한 제약을 모르고 여러 곳에서 \$monitor를 호출하면 예상치 못한 동작이 발생합니다.

[시나리오 3 - 코드 리뷰]: Testbench top에서 "\$finish;"로 시뮬레이션을 종료하는 대신 "\$stop;"을 사용한 이유를 묻는다면, Section 20.2를 근거로 \$stop은 시뮬레이터를 interactive mode로 전환하여 파형을 검사하거나 추가 명령을 실행할 수 있지만, regression test에서는 자동 종료를 위해 \$finish가

필요하다고 설명할 수 있습니다. UVM에서는 `drop_objection`이 자동으로 시뮬레이션 종료를 관리하므로 명시적 `$finish`가 불필요한 경우가 많습니다.

주의사항과 함정: LRM Section 21.2.1.3은 `$display`와 `$write`의 차이를 명확히 합니다. `$display`는 자동으로 `newline`을 추가하지만 `$write`는 추가하지 않습니다. 또한 Section 21.2.4의 `$displayh`, `$displayb` 등은 모든 인자를 해당 진법으로 출력하므로, 혼합 진법 출력에는 `$display`를 format specifier와 함께 사용하는 것이 더 유연합니다.

관련 LRM 섹션:

- Section 20.11: `$random` (p. 506-507) - 랜덤값 생성
- Section 21.7: File I/O system tasks (p. 546-549) - `$fopen`, `$fclose`, `$fdisplay`

Clocking Blocks

본문 참조: 5장 UVM Driver Class (고급 예시 참고)

LRM 섹션:

- Section 14: Clocking blocks (p. 295-313)
 - 14.3: Input and output skews (p. 300-303)
 - 14.9: Synchronous events (p. 307-308)
 - 14.12: Synchronous drives (p. 311-312)

LRM에서의 정의: LRM Section 14는 clocking block을 "synchronous signal의 timing과 synchronization을 명시하는 선언적 구조"로 정의합니다. Clocking block은 clocking event (보통 clock edge)를 기준으로 input signal의 sampling 시점과 output signal의 driving 시점을 skew로 제어합니다. Input skew는 clock edge 전에 signal을 샘플링하는 시간이고, output skew는 clock edge 후에 signal을 drive하는 시간입니다. Default input skew는 #1step (즉 Observed region)이고 output skew는 0 (즉 Re-Active region)입니다. 이는 setup/hold time을 고려한 safe sampling과 driving을 보장합니다.

실무 활용 시나리오: [시나리오 1 - 코딩 단계]: 복잡한 synchronous interface를 검증할 때 clocking block을 사용하면 timing을 명확히 할 수 있습니다. Section 14.3을 참조하여 "clocking cb @(posedge clk); input #1ns data; output #2ns valid; endclocking"처럼 선언하면, cb.data는 clk rising edge 1ns 전의 값을 샘플링하고, cb.valid는 rising edge 2ns 후에 drive 됩니다. 이는 실제 hardware의 setup time (1ns)과 output delay (2ns)를 정확히 모델링하며, Section 14.12의 synchronous drive semantics에 따라 non-blocking assignment처럼 동작합니다.

[시나리오 2 - 디버깅 단계]: Testbench Driver에서 직접 @(posedge clk)를 사용했을 때 setup/hold violation이 발생한다면, clocking block의 input/output skew로 해결할 수 있습니다. Section 14.3.1을 보면 default input skew #1step은 Observed region에서 샘플링하므로, 같은 clock edge의 모든 Inactive, NBA, Observed 업데이트가 완료된 후 안정된 값을 읽습니다. Output skew는 Re-Active region에서 drive 하므로 다음 clock cycle의 signal로 clean하게 전달됩니다. 이는 manual timing control 보다 LRM-defined semantics를 따라 안전합니다.

[시나리오 3 - 코드 리뷰]: Interface에 clocking block을 추가할 때 "clocking driver_cb @(posedge clk); output data, valid; endclocking"처럼 선언한 것이 왜 유용한지 설명할 때, Section 14.9를 근거로 clocking block 내부의 signal은 automatic synchronous event가 되어, driver가 "@(driver_cb)"처럼 간단히 동기화할 수 있고, "driver_cb.data <= value"처럼 직관적으로 drive 할 수 있다고 할 수 있습니다. 이는 raw signal 접근보다 abstraction level이 높고 timing-safe 합니다.

주의사항과 함정: LRM Section 14.6은 clocking block을 사용하면 signal을 clocking block 외부에서 직접 drive 할 수 없다고 경고합니다. 예를 들어 "clocking cb @(posedge clk); output data; endclocking"이 있을 때 "data = value;" (non-clocking) 할당은 허용되지만 예측 불가능한 동작을 유발할 수 있습니다. Clocking block을 사용한다면 해당 signal은 항상 clocking block을 통해서만 접근하는 것이 안전합니다.

관련 LRM 섹션:

- Section 16.5: Deferred assertions (p. 357-362) - clocking block과 함께 사용되는 assertion
 - Section 4.3.3: Stratified event scheduler (p. 40-42) - Observed와 Re-Active region의 정의
-

Compiler Directives

본문 참조: 11장 Testbench Top Module, 실무 프로젝트 구조

LRM 섹션:

- Section 22: Compiler directives (p. 553-566)
 - 22.1: define and undef (p. 554-556)
 - 22.3: ifdef, ifndef, elsif, and endif (p. 558-559)
 - 22.4: `include (p. 559-560)

LRM에서의 정의: LRM Section 22는 compiler directive를 "컴파일 시점에 소스 코드를 변환하는 preprocessing 명령"으로 정의합니다. define은 text macro를 정의하여 코드 재사용과 parameterization을 가능하게 합니다. ifdef는 conditional compilation을 제공하여 동일한 소스에서 다른 설정의 코드를 생성할 수 있습니다. `include는 외부 파일을 현재 위치에 삽입하여 코드 모듈화를 지원합니다. 이러한 directive는 SystemVerilog 뿐 아니라 Verilog, VHDL 등 대부분 HDL에 공통적으로 존재하는 preprocessing mechanism입니다.

실무 활용 시나리오: [시나리오 1 - 코딩 단계]: 프로젝트에서 다양한 configuration을 지원하려면 Section 22.1과 22.3을 활용합니다. "define ENABLE_ASSERTIONS"를 선언한 후 "ifdef ENABLE_ASSERTIONS bind dut assertion_module a_inst(.*); endif"처럼 사용하면, 컴파일 옵션으로 assertion enable/disable을 제어할 수 있습니다. 마찬가지로 "ifdef GATE_LEVEL_SIM include \"gate_netlist.v\" else include \"rtl_design.sv\""로 RTL과 gate-level simulation을 전환할 수 있습니다. Section 22.3.1의 `elsif로 multi-way branching도 가능합니다.

[시나리오 2 - 디버깅 단계]: 여러 testbench 파일을 통합했을 때 같은 이름의 define이 충돌한다면, Section 22.1의 scope rule을 확인합니다. define은

file scope 가 아닌 compilation unit scope 이므로, 한 파일의 정의가 나중에 컴파일되는 모든 파일에 영향을 줍니다. `undef` 를 사용하여 명시적으로 정의를 제거하거나, `ifndef` 로 중복 정의를 방지해야 합니다. Section 22.1.2 의 ``undef`` 사용법을 숙지하면 `macro namespace pollution` 을 피할 수 있습니다.

[시나리오 3 - 코드 리뷰]: Package 파일에서 "include `\\"uvm_macros.svh\\"\\"` 를 여러 번 사용한 것을 발견했을 때, Section 22.4 를 근거로 `include` 는 파일 내용을 단순 삽입하므로 중복 `include` 가 문제가 될 수 있다고 지적합니다. C 의 `#include guard` 처럼 "`ifndef UVM_MACROS_SVH define UVM_MACROS_SVH ... `endif`" 로 `multiple inclusion` 을 방지해야 합니다. UVM 자체가 이미 `guard` 를 가지고 있지만, `custom header` 는 명시적 `guard` 가 필수입니다.

주의사항과 함정: LRM Section 22.1.3 은 `macro argument substitution` 의 복잡한 규칙을 다룹니다. "`define ADD(a,b) a+b`" 를 정의하고 "`result = ADD(x,y) * 2;`" 로 사용하면 "`x+y*2`" 로 확장되어 우선순위 문제가 발생합니다. 안전하게 "`define ADD(a,b) ((a)+(b))`" 처럼 괄호를 사용해야 합니다. 또한 Section 22.1.4 의 `\``` (token pasting) 과 "`" (stringify) operator` 는 강력하지만 복잡하므로 신중히 사용해야 합니다.

관련 LRM 섹션:

- Section 3.7: String literals (p. 20-22) - `macro expansion` 내의 `string` 처리
- Section 22.5: ``line` (p. 560-561) - `error message` 의 line number 제어

File I/O Operations

본문 참조: 12 장 전체 시뮬레이션 플로우, 실무 프로젝트 구조

LRM 섹션:

- Section 21.3: File I/O tasks and functions (p. 525-546)
 - 21.3.1: File operations (p. 525-528)
 - 21.3.2: Integer file I/O (p. 528-532)
 - 21.3.3: Formatted output (p. 532-539)
 - 21.3.4: Reading data from a file (p. 539-543)

LRM에서의 정의: LRM Section 21.3은 SystemVerilog의 file I/O system task를 C의 stdio library와 유사하게 정의합니다. \$fopen은 file descriptor를 반환하며, \$fclose로 닫아야 합니다. \$fdisplay, \$fwrite 등은 \$display의 파일 버전이며 file descriptor를 첫 인자로 받습니다. \$fread, \$fscanf는 파일에서 데이터를 읽으며, binary와 text mode를 지원합니다. Section 21.3.1.1은 file descriptor를 32-bit integer의 multi-channel descriptor (MCD)로 정의하며, 여러 파일에 동시에 출력이 가능합니다. SystemVerilog-2012부터는 string-based file path도 지원합니다.

실무 활용 시나리오: [시나리오 1 - 코딩 단계]: 대량의 테스트 vector를 파일에서 읽어 시뮬레이션하려면 Section 21.3.4를 참조합니다. "integer fd = \$fopen("vectors.txt", "r"); while(!\$feof(fd)) \$fscanf(fd, "%h %h %h", a, b, expected); \$fclose(fd);"처럼 사용하여 파일의 각 라인을 읽어 Transaction으로 변환할 수 있습니다. Section 21.3.4.2의 \$fscanf return value는 성공한 assignment 개수를 반환하므로, 파일 형식 오류를 감지할 수 있습니다. Binary 파일은 \$fread로 더 효율적으로 읽을 수 있습니다.

[시나리오 2 - 디버깅 단계]: 시뮬레이션 결과를 파일로 저장하여 post-processing 하려면 Section 21.3.3을 사용합니다. Scoreboard에서 "integer result_fd = \$fopen("results.log", "w"); \$fdisplay(result_fd, "Time=%0t: a=%h, b=%h, y=%h, status=%s", \$time, a, b, y, status);"처럼 모든 Transaction을 파일에 기록하면, Python이나 다른 도구로 통계 분석을 할 수 있습니다. Section 21.3.1.2의 append mode "a"를 사용하면 여러 테스트 결과를 누적할 수 있습니다.

[시나리오 3 - 코드 리뷰]: VCD 파일 생성을 위해 "\$dumpfile("waves.vcd"); \$dumpvars(0, tb_top);"를 사용한 것이 올바른지 검토할 때, Section 21.7을

참조합니다. \$dumpfile 과 \$dumpvars 는 VCD (Value Change Dump) 형식으로 파형을 저장하는 standard task 이지만, Section 21.7.3에 따라 시뮬레이터마다 구현이 다를 수 있습니다. 대용량 파일을 피하려면 \$dumpvars 의 level과 module 인자를 신중히 설정하거나, FSDB 같은 proprietary format 을 사용해야 합니다.

주의사항과 함정: LRM Section 21.3.1.3은 file descriptor 가 유한 자원이므로 \$fclose 로 반드시 닫아야 한다고 경고합니다. 많은 파일을 열고 닫지 않으면 시뮬레이터가 "too many open files" 에러를 발생시킬 수 있습니다. 또한 Section 21.3.3.2의 \$fwrite 는 automatic newline 을 추가하지 않으므로, \$fdisplay 와 혼용 시 주의해야 합니다.

관련 LRM 섹션:

- Section 20.12: System functions (p. 507-512) -
\$test\$plusargs 로 file name 을 command line에서 전달
- Section 18.3: Memory initialization (p. 425-427) - \$readmemh 로 memory 초기화

실무 활용 워크플로우

상황 1: 새로운 SystemVerilog 기능을 처음 사용할 때

배경: 프로젝트에서 clocking block이나 interface class 같은 고급 기능을 처음 도입해야 하는 상황입니다. 온라인 예제를 복사하는 것이 아니라, 정확한 동작을 이해하고 프로젝트 요구사항에 맞게 적용해야 합니다.

단계별 워크플로우:

1. LRM의 목차(Table of Contents)에서 해당 기능의 주 섹션을 찾습니다.
예를 들어 clocking block 은 Section 14, interface 는 Section 25 입니다.
2. 섹션의 첫 부분에서 개요(overview)와 문법(syntax) 정의를 읽어 전체 개념을 파악합니다. BNF 표기법의 문법은 정확한 구문을 보장합니다.
3. Semantics 부분을 읽어 해당 construct 가 실제로 어떻게 동작하는지, 특히 simulation semantics 와 timing 을 이해합니다.

4. Examples 서브섹션이 있다면 참조하여 전형적인 사용 패턴을 학습합니다.
LRM의 예제는 standard-conforming 하므로 신뢰할 수 있습니다.
5. "See also" 부분이나 관련 섹션 참조를 따라가서 interaction 하는 다른 기능들을 이해합니다. 예를 들어 clocking block은 assertion (Section 16)과 함께 자주 사용됩니다.
6. 간단한 standalone test case를 작성하여 이해한 내용을 검증하고, 시뮬레이터 동작을 확인합니다.

실제 예시: Interface modport를 처음 사용할 때 Section 25.5를 보고 modport가 interface의 member subset을 제공하며 direction을 지정할 수 있음을 학습했습니다. "modport master(output req, input ack); modport slave(input req, output ack);"처럼 동일한 signal을 양방향에서 다른 direction으로 볼 수 있다는 것을 이해하고, Driver와 Monitor에서 각각 적절한 modport를 사용하여 compile-time safety를 확보했습니다.

상황 2: 시뮬레이션 결과가 예상과 다를 때

배경: 코드는 문법적으로 문제가 없고 컴파일도 성공했지만, 시뮬레이션 결과가 기대한 것과 다릅니다. 특히 타이밍 관련 문제나 race condition이 의심되는 상황입니다.

단계별 워크플로우:

1. 증상을 정확히 파악합니다. 어떤 신호가, 언제, 어떤 값으로 잘못 나타나는지 VCD 파형과 로그를 분석합니다.
2. 해당 신호를 생성하는 procedural block을 식별하고, 어떤 event나 timing control에 의해 실행되는지 확인합니다.
3. Section 4.3 Event Scheduler를 참조하여 실행 순서를 추적합니다. Active, NBA, Observed region 중 어디서 실행되는지 파악합니다.
4. Section 10.4의 blocking/nonblocking semantics를 재확인하여 assignment 타입이 적절한지 검증합니다.
5. Section 9.4.2의 event control 정의를 보고 @(posedge clk)나 @(signal) 같은 timing control이 정확히 언제 트리거되는지 확인합니다.
6. LRM의 정의에 기반하여 코드를 수정하고, 동일한 시뮬레이터와 다른 시뮬레이터에서 재검증합니다.

실제 예시: Monitor에서 "@(posedge clk); data = aif.data;"로 읽었는데 항상 한 클럭 이전 값을 읽는 문제가 발생했습니다. Section 9.4.2.1을 보니 event control은 Inactive region에서 재개되므로, DUT의 nonblocking assignment가 같은 클럭의 NBA region에서 업데이트한 값을 Active/Inactive region에서는 아직 볼 수 없다는 것을 이해했습니다. "#1; data = aif.data;"처럼 작은 delay를 추가하거나, clocking block의 input skew를 사용하여 Observed region에서 샘플링하도록 수정하여 해결했습니다.

상황 3: 코드 리뷰에서 동료와 의견이 다를 때

배경: 코드 작성 방식이나 문법 사용에 대해 팀원 간 의견이 다릅니다. "이게 더 좋다", "저게 더 안전하다" 같은 주관적 선호가 아닌 객관적 기준이 필요한 상황입니다.

단계별 워크플로우:

1. 논쟁이 되는 코드 패턴을 정확히 정의합니다. 예: "always @(*) vs always_comb", "logic vs reg" 등.
2. LRM에서 해당 구문의 정의를 찾아 정확한 semantics와 제약사항을 확인합니다.
3. LRM이 명시한 권장사항(recommended practice)이나 주의사항(caution)을 참조합니다.
4. Synthesizability가 중요하다면 IEEE 1800.2 (UVM) 또는 synthesis subset 문서를 함께 참조합니다.
5. 팀의 코딩 가이드라인과 LRM 기준을 조율하여 일관된 방침을 수립합니다.

실제 예시: "always @(*)"와 "always_comb" 중 어느 것을 사용할지 논의할 때 Section 9.2.2.4를 근거로 제시했습니다. LRM에 따르면 always_comb은 (1) 초기화 시에도 실행되고, (2) combinational logic임을 명시적으로 선언하며, (3) blocking assignment만 허용한다는 추가 제약이 있어 조합회로 의도를 더 명확히 표현한다고 설명했습니다. 팀은 LRM의 권장사항을 받아들여 조합회로에는 always_comb을 표준으로 채택했습니다.

상황 4: 서로 다른 시뮬레이터에서 동작이 다를 때

배경: VCS에서는 정상 동작하는 코드가 Questa에서 다르게 동작하거나, 혹은 그 반대입니다. 이식성 있는 코드로 수정해야 합니다.

단계별 워크플로우:

1. 두 시뮬레이터에서의 동작 차이를 정확히 문서화합니다. 어떤 신호가, 어떤 시간에, 어떻게 다른지 기록합니다.
2. LRM에서 해당 구문의 표준 정의를 확인합니다. Implementation-defined behavior인지 명확히 정의된 부분인지 판단합니다.
3. Section 1.6의 implementation-specific 항목을 확인하여 시뮬레이터마다 다를 수 있는 부분인지 파악합니다.
4. 두 시뮬레이터가 모두 준수해야 하는 LRM의 명확한 정의를 기준으로 코드를 수정합니다.
5. 수정 후 모든 대상 시뮬레이터에서 재검증하고, 표준 준수 여부를 확인합니다.

실제 예시: 초기화되지 않은 logic 변수가 VCS에서는 X였는데 Questa에서는 0으로 나타나는 문제가 발생했습니다. Section 6.5를 보니 "variables that are not explicitly initialized have an undefined initial value"라고 명시되어 있어, 시뮬레이터마다 구현이 다를 수 있음을 확인했습니다. 명시적 초기화 "logic [7:0] data = 8'h00;"를 추가하거나 reset logic에서 초기화하여 portable 한 코드로 수정했습니다.

상황 5: 복잡한 타이밍 계산이 필요할 때

배경: 고속 인터페이스나 정밀한 프로토콜 타이밍 구현 시 setup/hold time, propagation delay 등 정확한 시간 계산이 필수입니다.

단계별 워크플로우:

1. 필요한 타이밍 스펙을 정리합니다. Setup time, hold time, clock-to-output delay 등.
2. Section 22.7 Timescale을 참조하여 현재 time unit과 precision이 요구사항을 만족하는지 확인합니다.

3. Section 9.4.1 Delay control의 정확한 계산 방식을 파악합니다.
Inertial delay vs transport delay의 차이를 이해합니다.
4. Time precision에 따른 반올림과 정밀도 손실을 고려합니다. Section 3.14의 rounding rule을 적용합니다.
5. 실제 하드웨어 타이밍(SDF, timing model)과 testbench의 delay를 정확히 매핑합니다.

실제 예시: DDR interface에서 1.5ns setup time을 구현할 때 "`timescale 1ns/1ps"의 precision이 필요한 이유를 Section 22.7과 3.14를 통해 확인했습니다. Time unit이 1ns이고 precision도 1ns라면 1.5ns는 2ns로 반올림되어 부정확합니다. Precision을 1ps로 설정하면 $1.5\text{ns} = 1500\text{ps}$ 로 정확히 표현할 수 있습니다. Clocking block의 input skew를 "#1.5ns"로 설정하여 정확한 setup time을 구현했습니다.

효율적인 LRM 탐색 전략

검색 방법: LRM은 600페이지가 넘는 방대한 문서이므로 효율적인 탐색 전략이 필수입니다. (1) 목차를 활용하여 대략적인 위치를 파악합니다. 주요 섹션 번호를 암기하면 도움이 됩니다: Section 6 Data types, Section 9 Processes, Section 10 Assignments, Section 13 Tasks/Functions. (2) PDF 검색 기능(Ctrl+F)으로 키워드를 찾습니다. 예를 들어 "nonblocking"을 검색하면 정의와 모든 언급을 찾을 수 있습니다. (3) 색인(Index, 문서 맨 뒤)을 활용하여 정확한 섹션을 찾습니다. 색인은 알파벳순으로 모든 용어의 페이지를 나열합니다. (4) 한 섹션의 "See also" 부분을 따라가서 관련 섹션들을 학습합니다. LRM은 cross-reference가 잘 되어 있어 연관 개념을 함께 이해할 수 있습니다.

자주 참조하는 섹션 북마크: PDF 뷰어의 북마크 기능을 활용하여 자주 보는 섹션들을 등록해두면 빠르게 접근할 수 있습니다. 추천 북마크: Section 4.3 Event simulation, Section 6 Data types, Section 9 Processes, Section 10 Assignments, Section 13 Tasks/Functions, Section 22 Compiler directives. 또한 Annex에 있는 키워드 목록과 문법 요약도 quick reference로 유용합니다.

팀 내 LRM 활용 문화: 개인적으로만 LRM을 참조하는 것이 아니라 팀 전체가 공통 언어로 사용하면 의사소통이 명확해집니다. 코드 리뷰나 기술 논의에서 "Section 10.4.2에 따르면..."처럼 LRM 섹션을 인용하는 문화를 만듭니다. 또한 프로젝트

wiki에 자주 참조하는 LRM 섹션 링크와 요약을 정리해두면 팀원들이 쉽게 접근할 수 있습니다. 신입 엔지니어에게는 핵심 섹션(특히 Section 4.3, 6, 9, 10)을 먼저 읽도록 권장하여 기초를 다지게 합니다.

