

UVM_theory

Contents

Part 1. Introduction

- Important of Verification
- UVM의 설계 의도

Part 2. Structure

- UVM Component
- UVM RAL
- TLM

Part 3. Flexibility

- Factory
- Config DB
- Macro & Method

Part 4. Control (Orchestration)

- UVM Sequence
- Virtual Sequence
- Virtual Sequencer
- UVM Phase

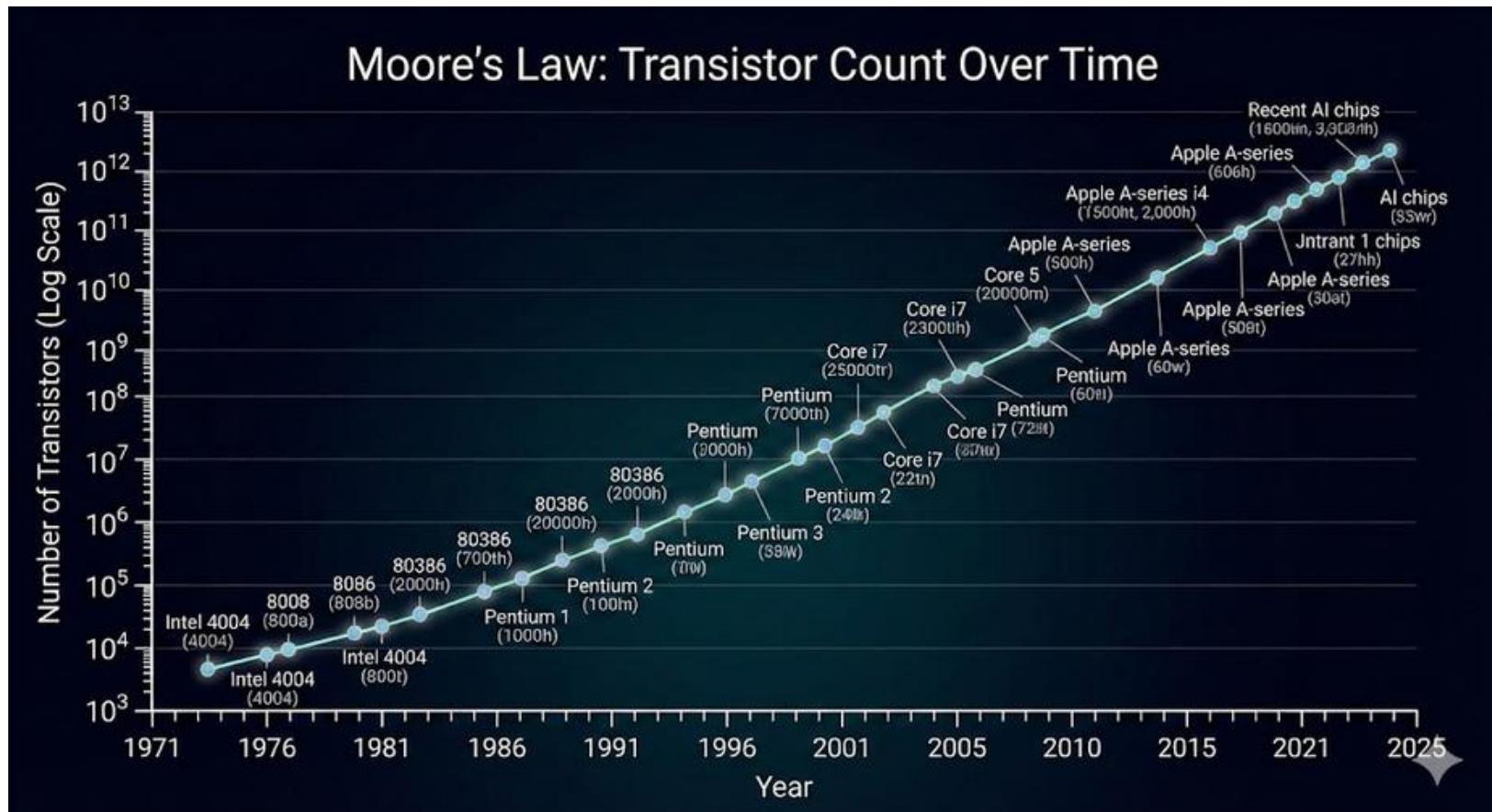
Part 5. Conclusion

- 배운 점
- Reference

01 Introduction

Introduction

✓ 검증이 왜 중요해졌을까?



무어의 법칙 : 반도체의 집적도는 기하급수적으로 증가

✓ 현재(As-Is)

- 현재 칩의 복잡도는 기하급수적으로 증가해왔고 현재 상용칩의 경우에 IP블록, 인터페이스까지 생각하면 복잡도가 엄청나다.

✓ 한계(Pain-Point)

- SystemVerilog 기반의 검증은 비효율로 인해서 현재 칩 발전 속도를 따라잡지 못한다.

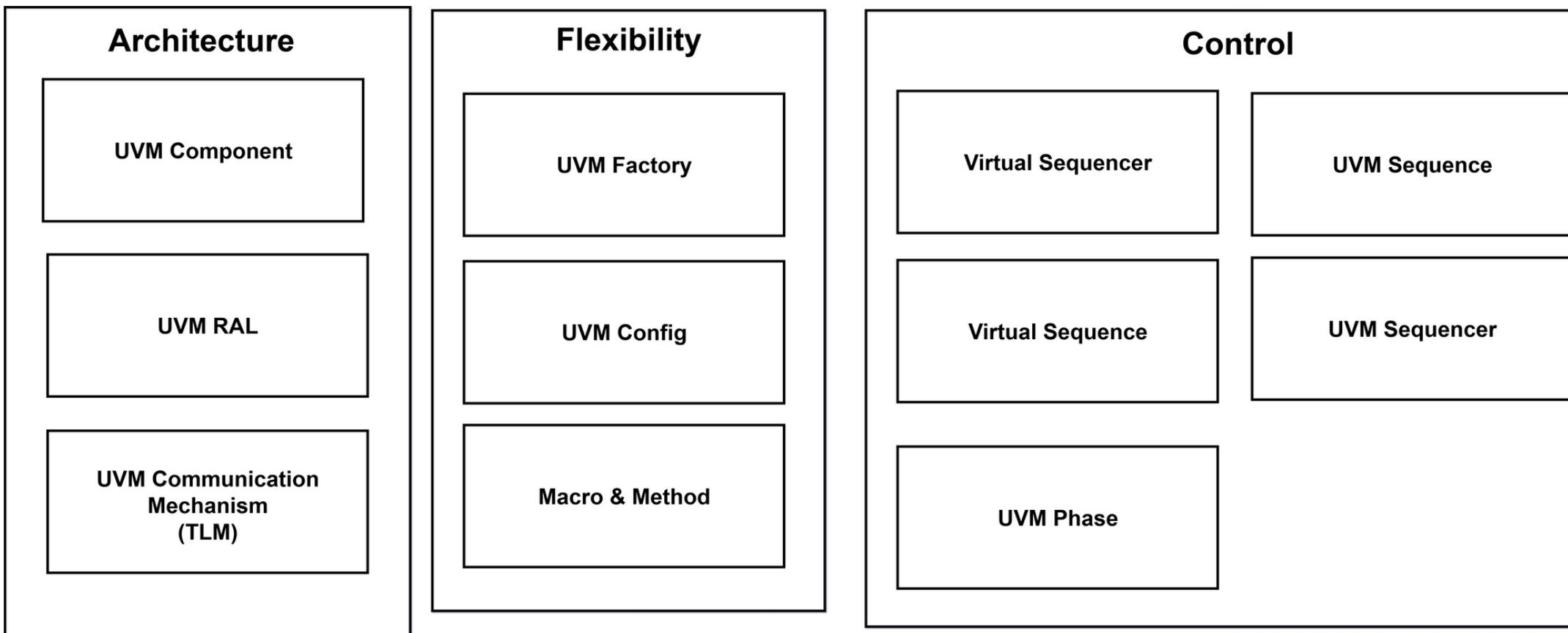
✓ 가야할 방향 (To-Be)

- 구조화 : 엔지니어마다 다른 코딩스타일 문제
- 유연성과 재사용성 : 테스트마다 작성해야하는 코드
- 제어와 동기화 : 복잡한 타이밍 제어

Introduction

✓ 그렇다면 왜 UVM을 사용하는가?

✓ UVM의 구조 (3개의 구조)



✓ UVM의 설계 철학

“우리는 검증 코드를 작성하는게 아니라
UVM을 통해서 ‘검증 플랫폼을 구축하는 것’이 핵심 목표이다.”

1. Architecture

- 계층성(Hierarchy)
- 모듈화(Modularity)
- 연결(T데이터 전송)

2. Flexibility

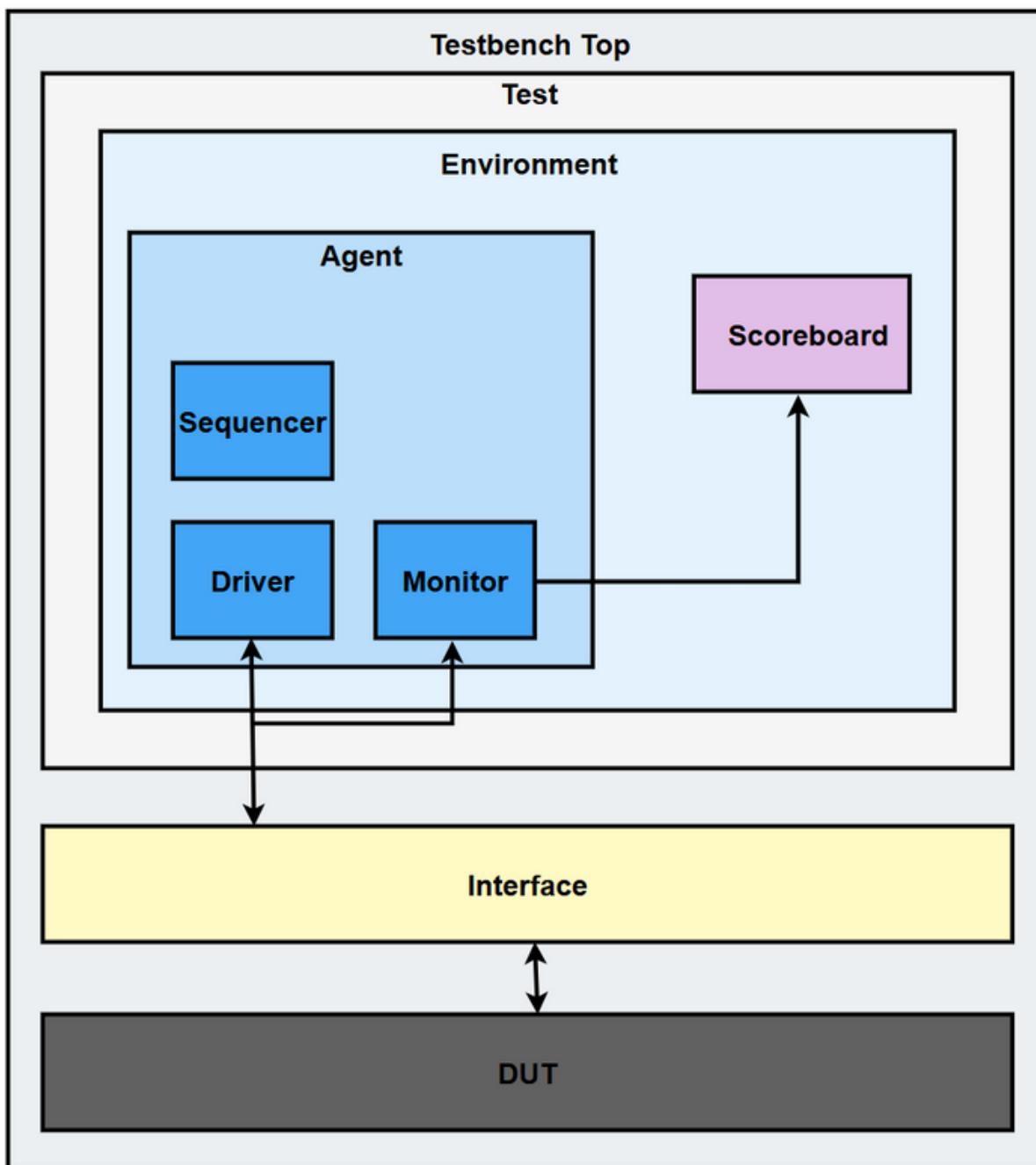
- UVM Factory (다형성)
- UVM Config DB (데이터 저장)
- Macro & Method(효율화)

3. Control

- Sequence, sequencer 제어
- UVM Sequence(생성 및 전송)
- UVM Sequencer(중재)
- UVM phase(스케줄러)

02 Structure

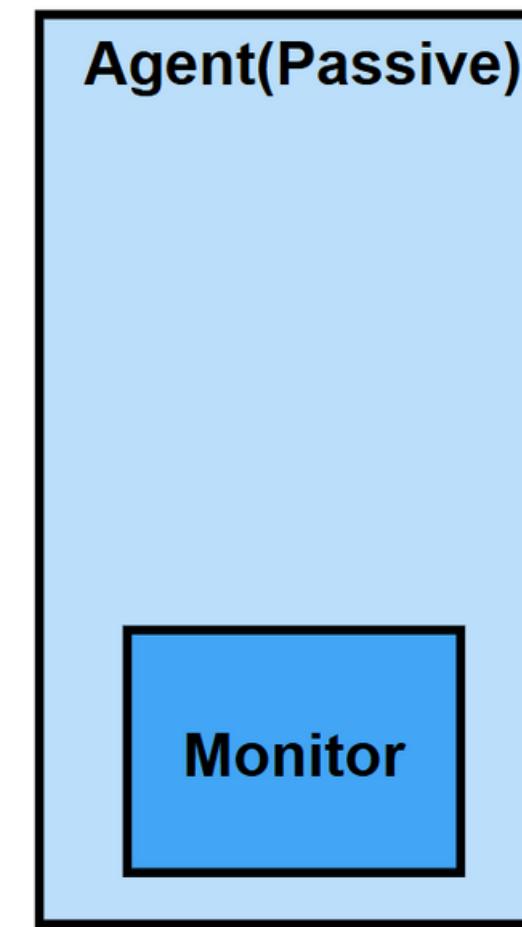
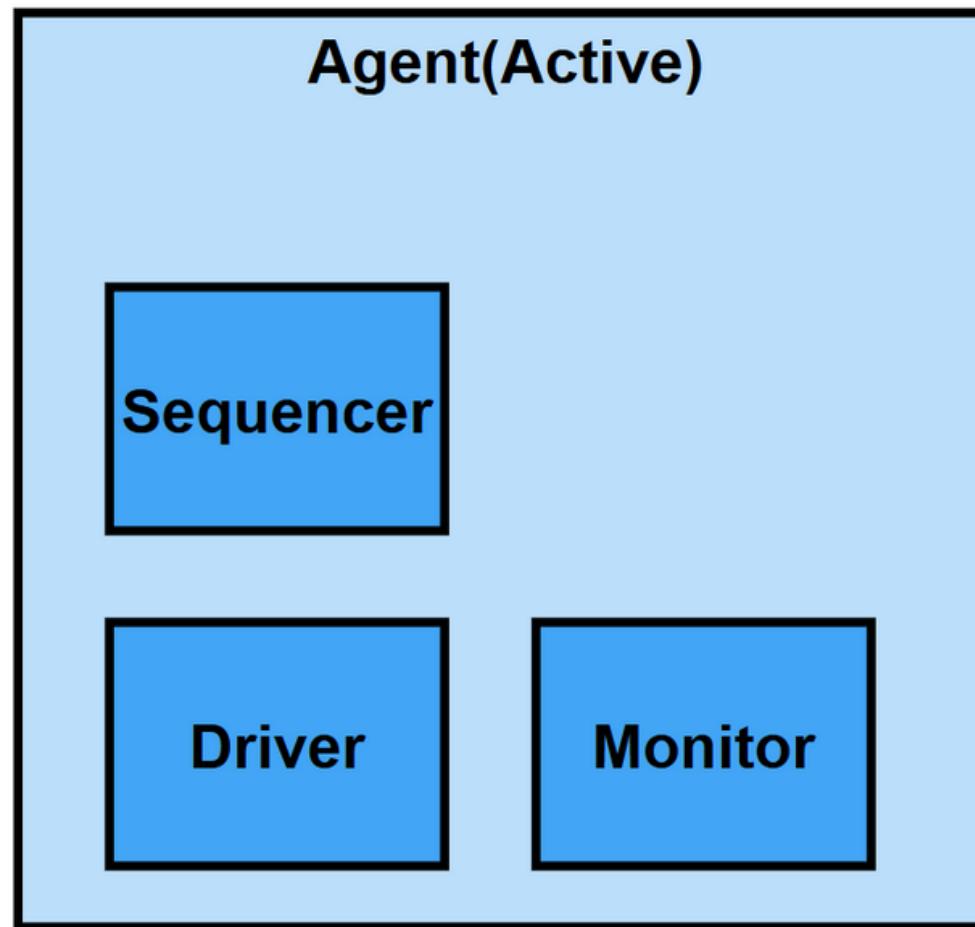
Component Architecture & Hierarchy



- **Test**
 - 검증 시나리오를 정의하고 Environment를 구성/실행
- **Environment**
 - Agent, Scoreboard 등 검증 컴포넌트들을 통합하는 컨테이너
- **Agent**
 - 특정 프로토콜을 처리하는 Sequencer, Driver, Monitor의 집합
- **Sequencer**
 - Transaction을 생성하고 Driver에게 전달하는 중재자
- **Driver**
 - Transaction을 pin-level signal로 변환하여 DUT에 인가
- **Monitor**
 - DUT의 pin-level signal을 관찰하여 Transaction으로 복원
- **Scoreboard**
 - Expected와 Actual 결과를 비교하여 Pass/Fail 판정

✓ 역할 기반 모듈화. 각 Component의 역할을 분리하여 컴포넌트의 재사용성과 확장성을 극대화

Agent

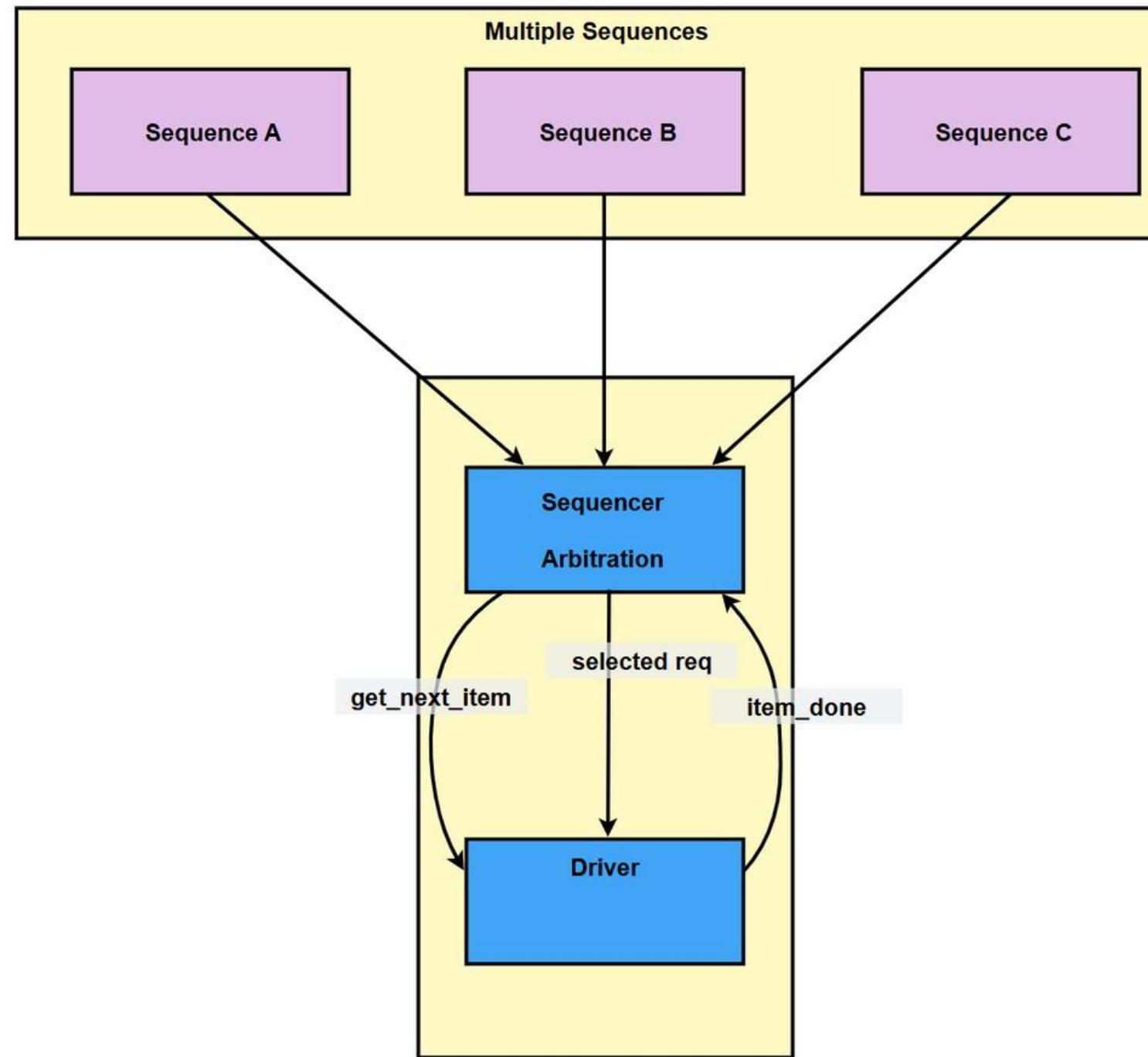


```
if (hget_is_active() == UVM_ACTIVE) begin  
    // Active일 때만 생성  
    sqr = simple_sequencer::type_id::create("sqr", this);  
    drv = simple_driver::type_id::create("drv", this);  
end  
  
// monitor는 조건 없이 항상 생성됨  
mon = sw_monitor::type_id::create("mon", this);
```

- Active Mode :
 - 구성: Sequencer + Driver + Monitor
 - 역할: DUT에 Stimulus를 주입하고, 응답을 관찰하여 Protocol Compliance를 검증함.

- Passive Mode :
 - 구성: Monitor Only
 - 역할: 트랜잭션 관찰 및 데이터 수집만 수행
 - 설계 의도: 수십 개의 Agent가 동시에 구동될 때 발생하는 시뮬레이션 오버헤드 제거. is_active 변경만으로 Block Level VIP를 SoC 환경에 수정 없이 통합 가능.

Sequencer



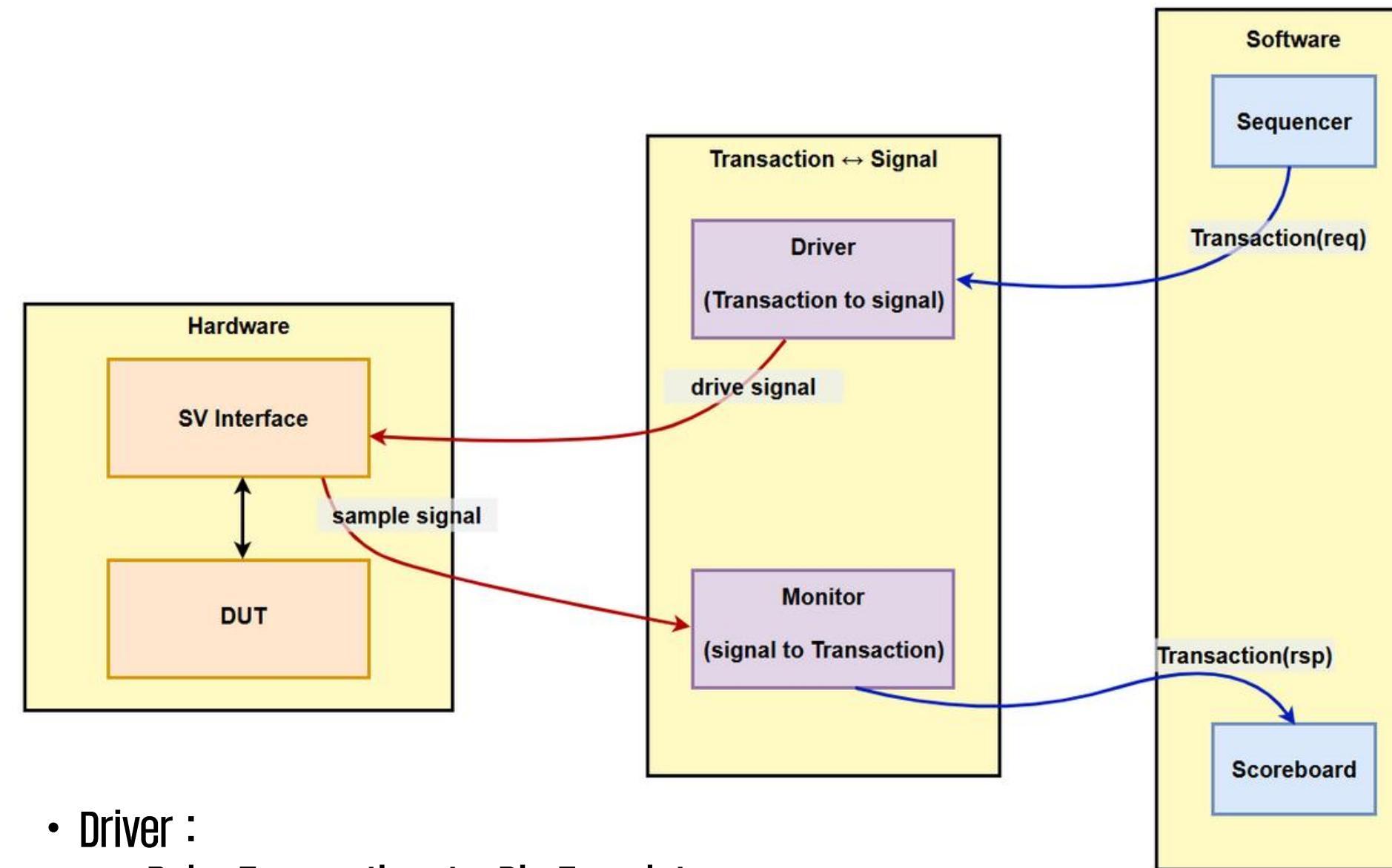
동작 원리: Handshake

- Pull Mode 통신. Driver가 준비되었을 때만 데이터를 전달함.
- 메커니즘:
 - Driver → Sequencer: `get_next_item()` 호출 - "Transaction 요청"
 - Sequencer: Arbitration 수행 후 선택된 `seq_item` 전달
 - Driver: DUT에 신호 인가 완료
 - Driver → Sequencer: `item_done()` 호출 - "처리 완료 통보"

핵심 기능: Arbitration

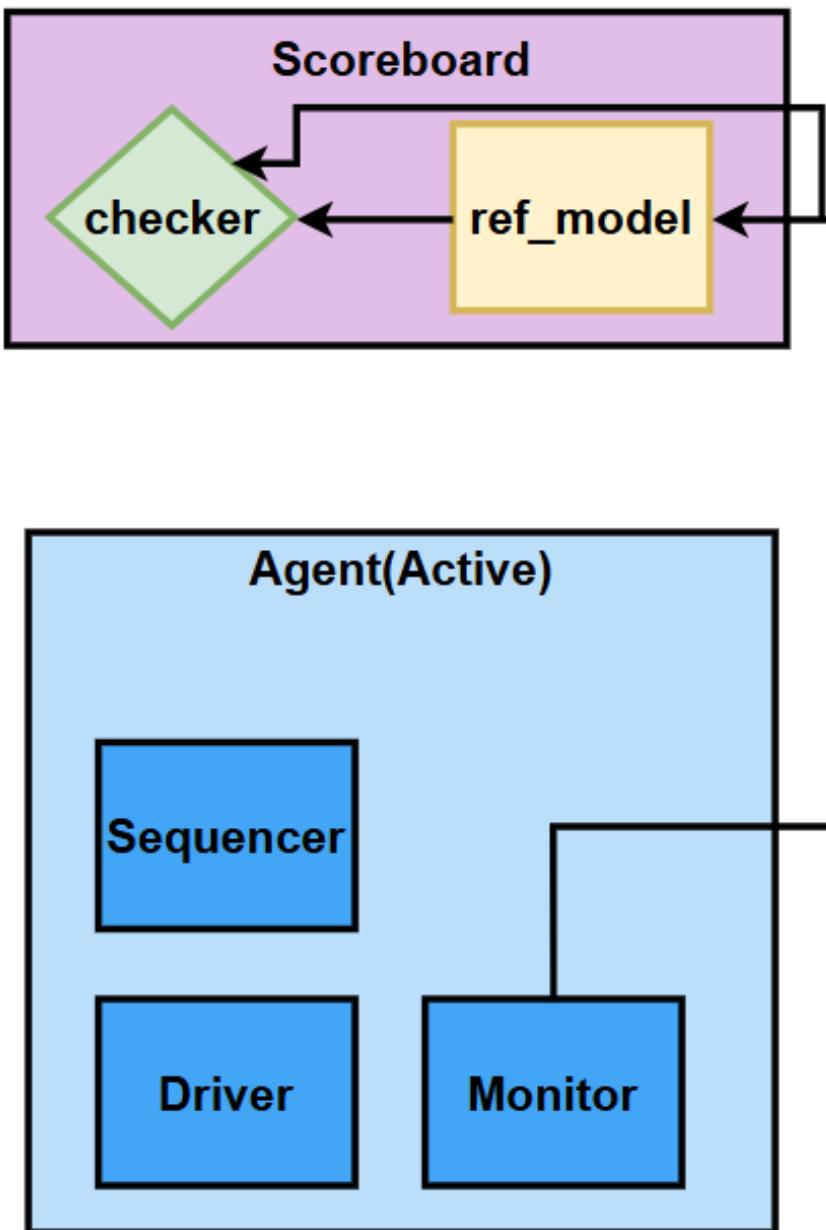
- 다수의 Sequence가 동시에 트랜잭션을 요청할 때, 우선순위나 알고리즘에 따라 실행 순서를 결정.
 - 알고리즘 예시: SEQ_ARB_FIFO, SEQ_ARB_RANDOM, SEQ_ARB_STRICT_FIFO...

Driver & Monitor



- **Driver :**
 - Role: Transaction-to-Pin Translator
 - Action: Sequencer의 Transaction을 pin-level Signal로 변환하여 DUT에 인가.
- **Monitor :**
 - Role: Pin-to-Transaction Reconstructor
 - Action: DUT의 pin-level signal 변화를 감지하여 의미 있는 Transaction으로 복원

Scoreboard



- Automated Self-Checking
 - 육안 검사를 배제하고, Reference Model을 기반으로 수만 개의 트랜잭션을 실시간으로 자동 판정하여 검증 신뢰성을 확보

```
class line_buf_scoreboard extends uvm_scoreboard;
function void write_input(line_buf_seq_item item);
    input_queue.push_back(item);
    total_input_pixels++;
endfunction

class line_buf_scoreboard extends uvm_scoreboard;
function void write_output(line_buf_seq_item item);
    foreach (input_queue[i]) begin
        if(input_queue[i].frame_num == item.frame_num)
            expected = input_queue[i];
        found = 1;
        break;
    end
endfunction

class line_buf_scoreboard extends uvm_scoreboard;
function void write_output(line_buf_seq_item item);
    if (found) begin
        if (expected.r_data == item.r_data &&
            expected.g_data == item.g_data &&
            expected.b_data == item.b_data) begin
            pass_count++;
        end
    end
endfunction
```



Scoreboard 구현 방식

1. Internal Reference Model

- Scoreboard 내부에 ref_model 포함
- 입력 Transaction으로 expected 값을 직접 계산
- 단순한 DUT에 적합

2. External Reference Model

- Reference Model이 Environment에 별도 존재
- Scoreboard는 순수 비교기 역할만 수행
- 복잡한 설계나 재사용성이 중요할 때 선호

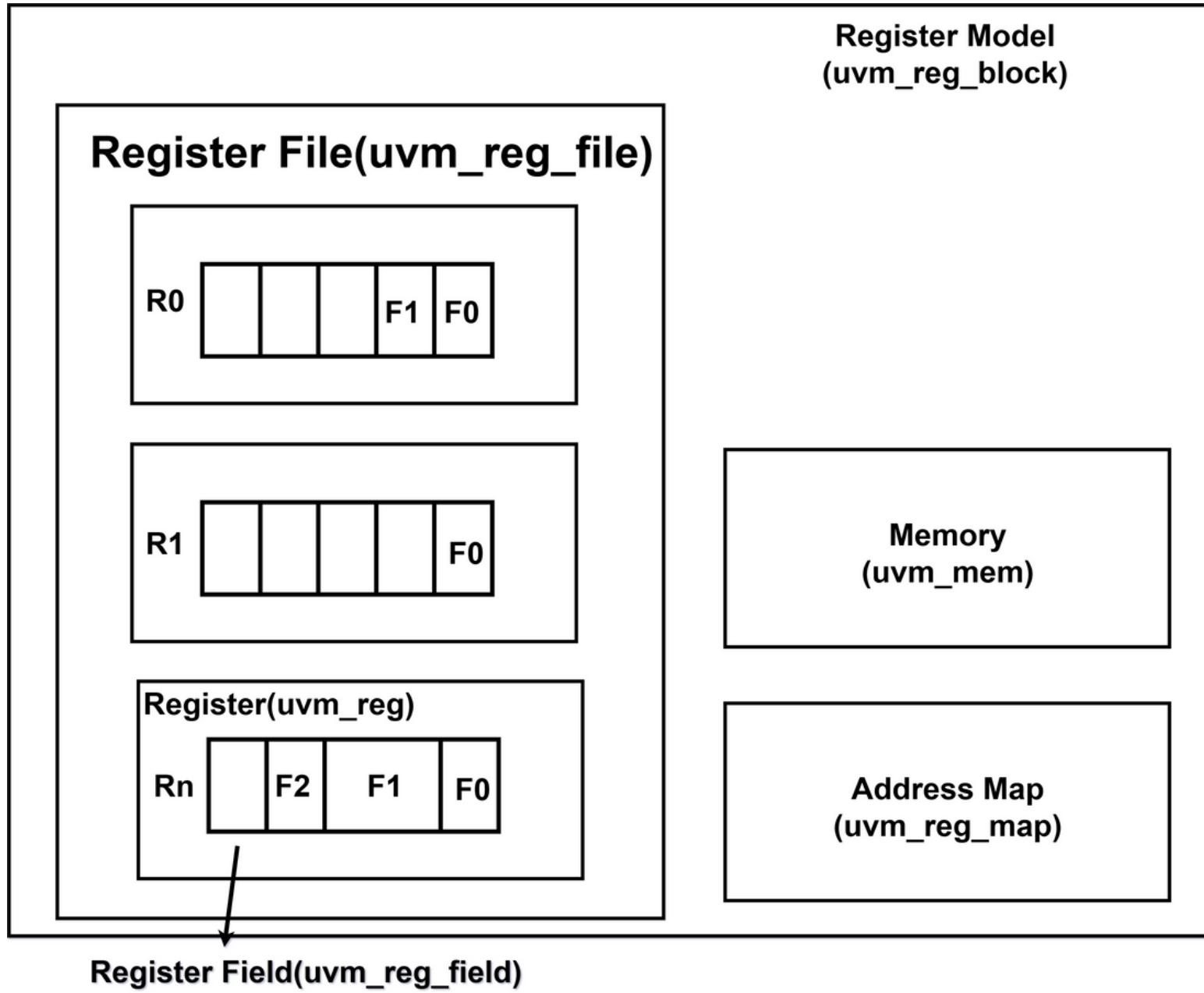
3. Golden File 기반

- 미리 생성된 expected 결과 파일과 비교
- Simulation 속도 최적화에 유리

RAL



RAL



RAL이란 Register 검증을 위한 객체지향 모델이고 이를 바탕으로 추상화하여 레지스터를 다룰 수 있게 해준다.

uvm_reg_block

정의 : 레지스터와 메모리 그리고 다른 블록들을 담는 최상위 컨테이너
역할 : 디자인의 계층 구조를 그대로 반영

uvm_reg

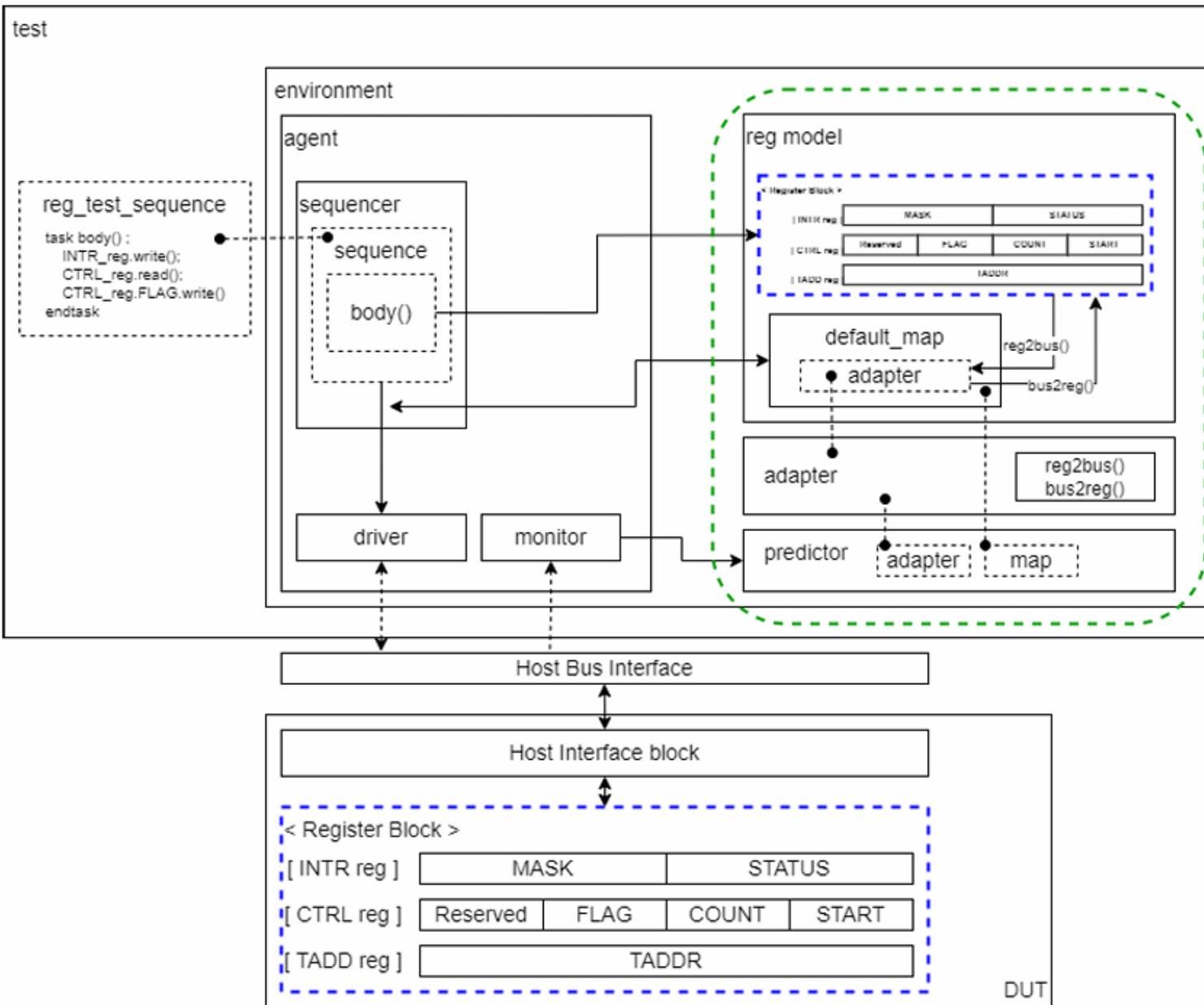
정의 : 물리적인 레지스터 하나를 추상화한 객체
역할 : 데이터 값을 저장하는 필드(uvm_reg_field)를 포함하며 읽고 쓰는 행위의 주체

uvm_reg_map

정의 : 레지스터들의 주소(Offset), 엔디안(Endian), 접근 권한 등을 정의한 지도
역할 : 이 레지스터가 버스에서 어디에 위치하는 가를 정의

RAL

Ral 기능



1. 연결 : Adapter (Translator)

reg2bus : register transaction을 bus transaction을 변환

bus2reg : bus transaction을 register transaction을 변환

2. 상태의 동기화 : Mirroring & Prediction

RAL은 단순히 명령만 내리는 게 아니고 현재 하드웨어의 상태를 기억한다.

- Mirror Value : RAL 모델이 생각하는 현재 DUT 레지스터의 값
- Prediction(예측)

Implicit Prediction는 사용자가 RAL을 통해 쓰고 읽은 값으로 Mirror를 자동 업데이트

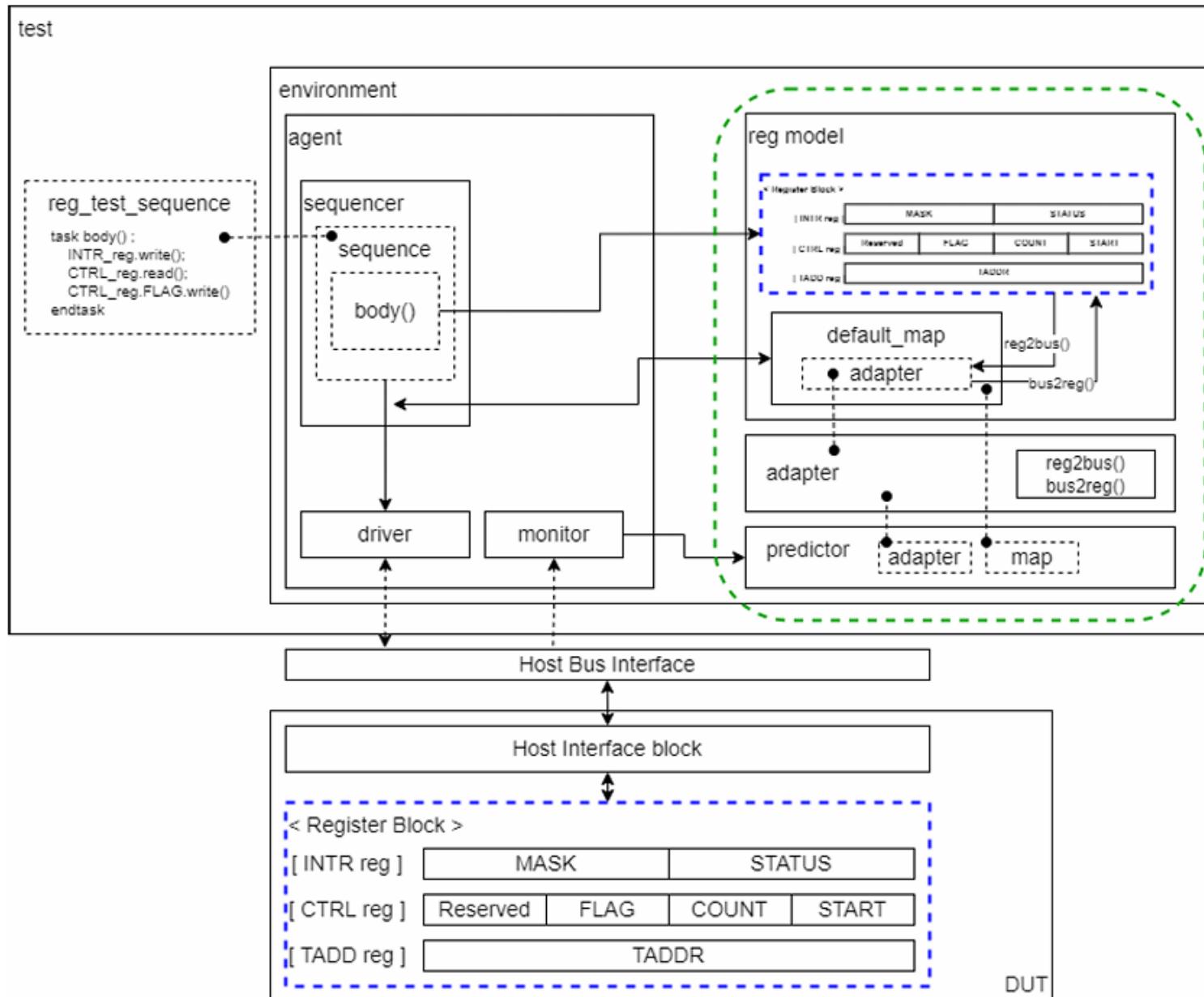
Explicit Prediction는 Bus Monitor가 관찰한 트랜잭션을 Predictor가 받아채서 Mirror를 업데이트 한다.

3. 접근 방식의 유연성 : Front-door Vs Back-door

- Front-door Access: 실제 버스 프로토콜(Agent)을 타고 물리적 시간(time)을 소모하며 접근
- Back-door Access: 시뮬레이션의 뒷문(HDL Path)을 통해 시간 소모 없이(Zero-Time) 즉시 값을 write하거나 read한다

RAL

Ral Data Flow



상황 : write

1. sequence : write() 호출

2. Register model : map 주소 계산

3. Adapter : reg2bus() 실행

4. Sequencer : 변환된 transaction() sequencer에게 전달

5. Driver : dut에 전달

상황 : Mirror

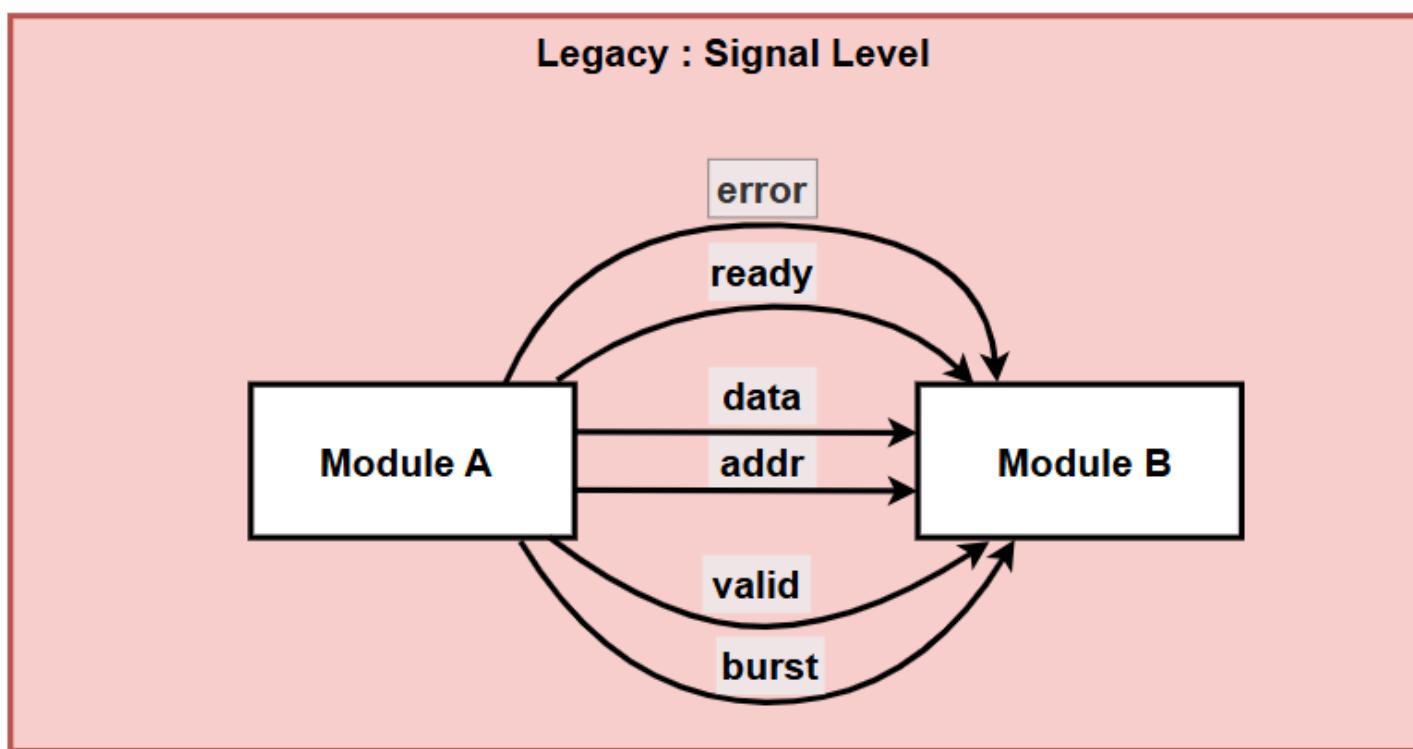
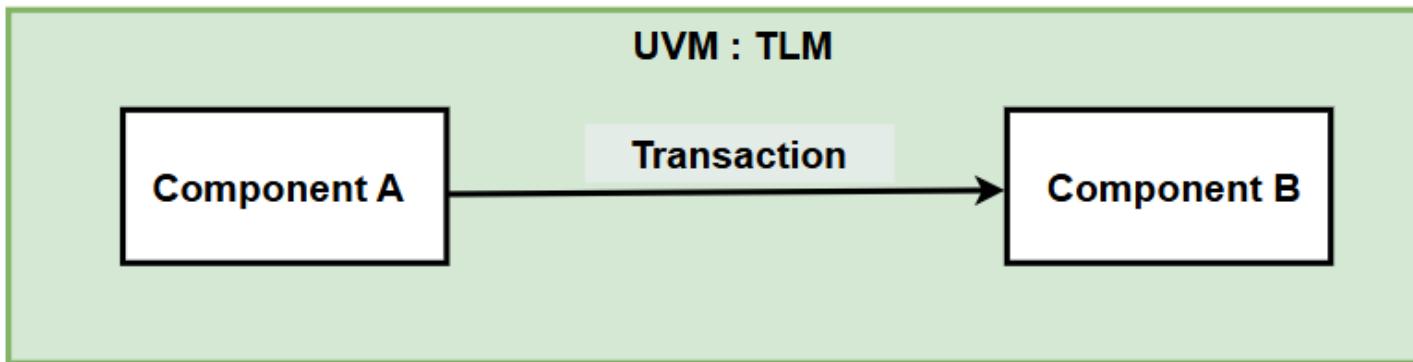
1. Monitor : Driver가 데이터 보낼때 신호 감지 후 트랜잭션 객체로 복원 후 broadcasting

2. predictor : Monitor broadcasting 받아서 adapter 호출

3. Adapter : bus2reg() 실행

4. register Model : 받아서 Mirror update

TLM



항목	Signal Level (Legacy)	TLM Level (TLM)
통신 단위	개별 신호 (addr, data, valid...)	Transaction
연결 복잡도	신호마다 개별 연결	단일 Port
추상화 수준	Low (Pin-level)	High (기능)
재사용성	낮음 (프로토콜 변경 시 전체 수정)	높음 (Transaction)

정의: 컴포넌트 간에 Transaction이라는 추상화된 데이터 패킷을 주고받는 표준 통신 규격.

목적: 컴포넌트 간의 결합도를 낮추고, 인터페이스 변경에 유연하게 대처하며, 재사용성을 높이기 위함

TLM

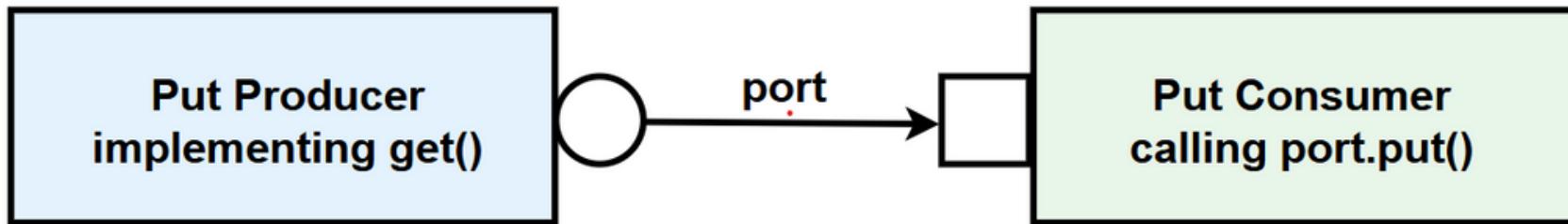
TLM 3요소

요소	이름	역할
Port	uvm_*_port	통신 시작자. 함수를 호출하여 데이터를 보내는 역할
Export	uvm_*_export	중계자. Port의 요청을 받아서 내부의 Imp를 통해 처리하는 역할

 모든 TLM 연결의 끝은 반드시 Imp여야 한다. Port끼리 연결하거나 Export끼리 연결할 수는 있어도, 종착지는 Imp여야 실제 함수가 실행된다.

TLM

PULL MODE



```
class my_driver extends uvm_driver #(my_transaction);
`uvm_component_utils(my_driver)

virtual task run_phase(uvm_phase phase);
forever begin
    // PULL 요청
    // Driver는 여기서 데이터가 올 때까지 대기(Initiator)
    seq_item_port.get_next_item(req);

    // drive : 받은 데이터(req) 구동
    drive_signal(req);

    // 완료 보고
    seq_item_port.item_done();
end
endtask
```

```
// Sequence (Producer 역할)
virtual task body();
    req = my_transaction::type_id::create("req");

    start_item(req);
    req.randomize();
    finish_item(req);
endtask
```

- 개념: "필요할 때 가져간다."
- 주체: Consumer가 Initiator가 되어 get()을 호출.
- Use Case: Sequencer ↔ Driver.
- Driver는 DUT의 timing에 맞춰 동작해야 하므로, Driver가 준비된 시점에 직접 Transaction을 요청하는 Pull Mode를 사용하여 안정적인 데이터 전달을 보장한다.

TLM

FIFO MODE



```
class my_scoreboard extends uvm_scoreboard;
    // 1. FIFO 선언
    uvm_tlm_analysis_fifo #(my_transaction) tlm_fifo;

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

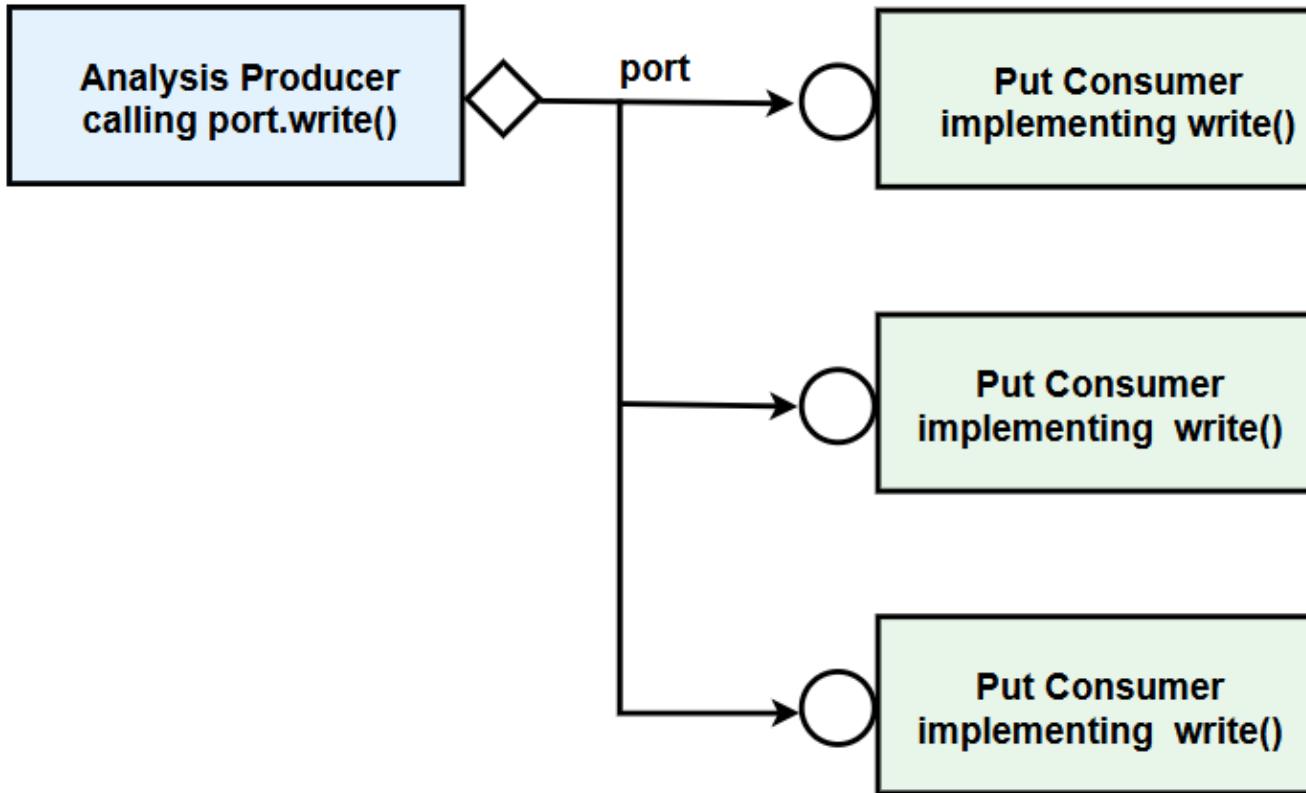
    task run_phase(uvm_phase phase);
        my_transaction tr;
        forever begin
            // 2. 데이터 꺼내기 (Blocking Get)
            // FIFO에 데이터가 없으면 대기, 있으면 가져와서 처리
            tlm_fifo.get(tr);

            // 3. 검증 로직 수행
            compare_data(tr);
        end
    endtask
endclass
```

- 개념: "밀어 넣되, 버퍼를 통해 안전하게"
- 주체: Producer가 put() 호출, Consumer가 get() 호출, 중간에 TLM FIFO가 버퍼 역할
- Use Case: 송신 속도와 수신 속도 차이가 있는 컴포넌트 간 통신
- Why FIFO? Producer와 Consumer의 처리 속도가 다를 경우, 직접 연결하면 Producer가 blocking되거나 데이터 유실이 발생할 수 있다. TLM FIFO를 중간에 배치하면 양쪽이 서로 독립적으로 동작하며(Decoupling), 속도 차이를 험수하여 안정적인 데이터 전달을 보장한다.

TLM

✓ Broadcast Mode(1:N)



```
// Monitor (Producer)
uvm_analysis_port #(my_tx) ap; // Analysis Port 선언
ap.write(tx); // 모든 Subscriber에게 전달

// Scoreboard (Subscriber)
uvm_analysis_imp #(my_tx, scoreboard) ai;
function void write(my_tx tx);
    // 수신 처리
endfunction
```

- 개념: "한 번에 여러 곳으로 전달한다"
- 주체: Producer(Monitor)가 write() 호출, 다수의 Subscriber가 동시 수신
- Use Case: Monitor → Scoreboard, Coverage
- Monitor는 DUT의 신호를 관찰하여 Transaction으로 변환하며, 이를 Scoreboard와 Coverage 등 여러 컴포넌트에 동시에 전달해야 한다. Analysis Port를 사용하면 1:N Broadcast가 가능하고, Subscriber가 없어도 예러 없이 동작한다.

03 Flexibility

Flexibility

✓ Flexibility란?

- 기존에 만들어진 Testbench 구조(Architecture)를 수정하지 않고, 외부 설정만으로 동작이나 구성을 마음대로 변경할 수 있는 능력

✓ 왜 필요한가?

1. 하드코딩 방지

테스트 케이스가 바뀔 때마다 검증 컴포넌트 (Agent 내부 Driver 등)의 소스 코드를 직접 수정하면 기존 테스트가 망가지거나 관리가 불가능해짐

2. 다양한 시나리오 대응

동일한 구조 안에서 정상 동작뿐만 아니라 에러 주입이나 예외 상황을 테스트해야 함

3. 계층 간 소통의 벽

상위(Test)에서 하위(Driver)로 데이터를 전달할 때 (Test → Env → Agent → Driver)
계층을 뚫고 데이터를 전달할 표준화된 방법이 필요

Flexibility

✓ Flexibility란?

- 기존에 만들어진 Testbench 구조(Architecture)를 수정하지 않고, 외부 설정만으로 동작이나 구성을 마음대로 변경할 수 있는 능력

✓ 해결책

1. 하드코딩 방지 → Factory로 부품 교체 (Override)
2. 다양한 시나리오 대응 → Factory + Config DB로 유연하게 대응
3. 계층 간 소통의 벽 → Config DB로 벽 뚫기
4. 복잡한 절차 → Macro & Method로 표준화

⇒ 재사용 가능한 ‘검증 플랫폼’ 구축 가능!!!

Factory

✓ Factory란?

- UVM 객체(Component, Object)를 생성(Create)하고 관리하는 중앙 시스템
- Testbench 소스 코드(Env, Agent 등)를 수정하지 않고, 외부(Test)에서 객체의 종류를 쉽게 교체(Override)하기 위해 사용
- 이로써 Testbench 설계를 더 유연하고 확장 가능하게 만듦

✓ Step-by-Step

1. Factory 등록 (Registration) → Macro
2. Factory 생성 (Create) → create()
3. Factory 교체 (Override) → Override

Factory

✓ Step 1: Factory 등록 (Registration)

- UVM factory가 어떤 class들을 생성할 수 있는지 알 수 있도록 미리 정보를 등록하는 과정
- uvm_object 및 uvm_component 파생 class 선언 부에 Macro 선언
 - object: `uvm_object_utils(Type)
 - component: `uvm_component_utils(Type)

```
// 1. 등록 (Registration)
// Component
class my_driver extends uvm_driver;
  `uvm_component_utils(my_driver)
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
endclass

// Object
class my_transaction extends uvm_sequence_item;
  `uvm_object_utils(my_transaction)
  function new(string name = "my_transaction");
    super.new(name);
  endfunction
endclass
```

→ Factory 등록 완료!

→ Factory 등록 완료!

Factory

✓ Step 2: Factory 생성 (Create)

- Factory에 등록된 class 객체를 생성
- 직접 new로 생성하는 방식이 아닌 create 메서드를 호출하는 방식
 - object: handle = 클래스명::type_id::create("이름");
 - component: handle = 클래스명::type_id::create("이름", this);

✓ Create()의 과정

1. 객체 생성 요청
2. 오버라이드 규칙 확인
 - Factory가 내부 오버라이드 테이블을 검색
 - 요청된 타입과 인스턴스 경로에 대해 Override 규칙([Type or Instance](#))이 등록되어 있는지 확인
3. 객체 생성 및 반환
 - 오버라이드 규칙이 없으면 → 요청된 원본 타입의 new() 호출
 - 오버라이드 규칙이 있으면 → 오버라이드된 타입의 new() 호출

Factory

✓ Step 3: Factory 교체 (Override)

- 특정 타입의 객체가 생성될 때, 항상 다른 타입의 객체를 생성하도록 지시 가능
- Testbench 코드 수정 없이 테스트 환경의 동작 변경
- 객체 생성 전에 Override 설정 (보통 Test의 build_phase)

✓ Override 종류

1. Type Override

- Testbench 전체에서 해당 타입이 생성될 때마다 적용
- **광범위한 변경**이 필요할 때 사용
- `set_type_override_by_type(original_type, override_type);`

2. Instance Override

- 특정 경로의 객체에만 Override 적용
- **지역적인 변경**이 필요할 때 사용
- `set_inst_override_by_type(inst_path, original_type, override_type);`

Factory

✓ Factory 전체 흐름

```
// 1. 등록 (Registration) - Driver 클래스
class my_driver extends uvm_driver;
  `uvm_component_utils(my_driver)
  ...
endclass

class error_driver extends my_driver;
  `uvm_component_utils(error_driver)
  ...
endclass

// 2. 생성 (Creation) - Env
function void build_phase(uvm_phase phase);
  drv = my_driver::type_id::create("drv", this);
endfunction

// 3. 교체 (Override) - Test
function void build_phase(uvm_phase phase);
  my_driver::type_id::set_type_override(error_driver::get_type());
  env = my_env::type_id::create("env", this);
endfunction
```

→ Step 1: Factory 등록

→ Step 1: Factory 등록

→ Step 2: Factory 생성, Override 규칙 확인

→ Step 3: Factory 교체

⇒ Override 규칙에 의해 my_driver가 아닌 error_driver 객체 생성

Config DB

✓ Config DB란?

- 계층 간 파라미터나 객체를 전달하는 중앙 집중식 데이터베이스
- 상위(Test, Env)에서 하위(Driver, Monitor)로 설정을 전달
- 장점: 계층적 제어 가능, 유연성 및 재사용성 향상
- 누구나 접근 가능한 공유 데이터센터!

✓ 왜 필요한가?

문제점

- UVM은 UVM은 Test → Env → Agent → Driver처럼 층층이 쌓인 구조 → 계층이 너무 깊고 복잡!
- 만약 Test에서 Driver로 변수 하나를 전달하려면? → 번거로움!

해결책

- uvm_config_db라는 전역(Global) 저장소를 만들자!
- Test는 데이터를 저장소에 올려두기만 하고(Set),
- Driver는 저장소에서 직접 꺼내 쓰기만 하면(Get) 됨!

Config DB

✓ uvm_config_db

- Instance 기반 설정 가능 (특정 경로의 컴포넌트에만 설정 전달), `uvm_resource_db`를 확장한 것
- 주로 `build_phase`에서 사용

✓ 핵심 동작

1. set() - 설정값 저장

- `uvm_config_db#(타입)::set(context, inst_name, field, value);`

2. get() - 설정값 가져오기

- `if (!uvm_config_db#(타입)::get(context, inst_name, field, variable))`
 '`uvm_fatal("GET_FAIL", "Configuration을 찾을 수 없습니다")
- 리턴값: 성공 시 1(true), 실패 시 0(false)

Config DB

✓ uvm_config_db

- Instance 기반 설정 가능 (특정 경로의 컴포넌트에만 설정 전달), [uvm_resource_db](#)를 확장한 것
- 주로 build_phase에서 사용

✓ 핵심 활용

1. Virtual Interface 연결

- Static(H/W, Module)과 Dynamic(S/W, Class) 세계를 연결하는 유일한 통로
 - Top: ::set(null, "*", "vif", vif_bus);
 - Driver: ::get(this, "", "vif", vif);

2. Component 동작 제어

- Test에서 Agent 내부 설정 변경 (예: is_active = UVM_PASSIVE)

3. Configuration Object의 계층적 전달

- 설정값을 객체로 묶어서 통째로 전달 → 변수 하나하나 보내는 번거로움 해결

Config DB

✓ uvm_resource_db

- Global 설정에 주로 사용
- 모든 Component가 공유하는 값을 설정할 때 편리
- Instance 경로 없이 이름만으로 접근

✓ 핵심 동작

1. `set()` - 설정값 저장

- `uvm_resource_db#(타입)::set(scope, name, value, accessor);`

2. `read_by_name()` - 이름으로 읽기

- `uvm_resource_db#(타입)::read_by_name(scope, name, value, accessor);`

3. `read_by_type()` - 타입으로 읽기

- `uvm_resource_db#(타입)::read_by_type(scope, value, accessor);`

Config DB

✓ uvm_config_db 활용 예시

1. Virtual Interface 연결 (H/W ↔ S/W)

```
// 1. 보내기 (Set) - tb_top.sv (Module)
initial begin
    uvm_config_db#(virtual bus_if)::set(null, "*", "vif", vif_bus);
end

// 2. 받기 (Get) - my_driver.sv (Class)
function void build_phase(uvm_phase phase);
    if(!uvm_config_db#(virtual bus_if)::get(this, "", "vif", vif))
        `uvm_fatal("NOVIF", "Virtual interface not found")
endfunction
```

→ null, *: 전역(Global)

→ this, “”: 자신

⇒ tb_top(H/W)의 물리적 인터페이스가 Driver(S/W)의 가상 인터페이스로 연결됨

Config DB

✓ uvm_config_db 활용 예시

2. 동작 제어 (Control)

```
// 1. 설정하기 (Set) - my_test.sv
function void build_phase(uvm_phase phase);
    uvm_config_db#(uvm_active_passive_enum)::set(this, "env.agent",
"is_active", UVM_PASSIVE);
endfunction

// 2. 적용하기 (Get) - my_agent.sv (이미 UVM 내부에 구현되어 있음)
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
endfunction
```

→ passive: 모니터링만, “env.agent”: 저 경로만

→ 부모 클래스(uvm_agent) 내부에 get 코드가
내장되어 있어서, 자동으로 설정을 받아옴

⇒ 설정(PASSIVE)에 따라 Driver와 Sequencer는 생성되지 않고, Monitor만 생성되어 동작함

Macro & Method

Macro & Method란?

- Macro([utils](#)): 만든 클래스를 Factory 명부(Lookup Table)에 등록하는 도구
- Method([create](#)): new() 대신 Factory를 통해 객체를 생성하는 표준 함수
- 관계: Macro로 등록하고 create()로 생성해야만, Factory가 개입하여 객체를 교체(Override)할 수 있음

✓ 주요 매크로

1. [Report Macros](#) (메시지 출력)
2. [Sequence-Related Macros](#) (트랜잭션 전송)
3. [Utility and Field Macros](#) (자동화)

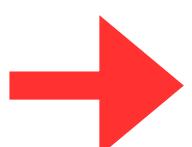
Macro & Method

✓ Report Macros (메시지 출력)

- **'uvm_info(ID, MSG, VERBOSITY)**
 - 로그 메시지를 출력할 때 사용
 - verbosity_level보다 이하인 verbosity를 갖는 메시지만 출력됨
- **'uvm_warning(ID, MSG)**: 경고 메시지 출력, 시뮬레이션은 계속됨
- **'uvm_error(ID, MSG)**: 오류 발생 메시지 출력, 시뮬레이션은 계속됨
- **'uvm_fatal(ID, MSG)**: 치명적 오류 발생 메시지 출력, 시뮬레이션을 즉시 종료

```
`uvm_info(  
    "IN_MON",  
    $sformatf(  
        "INPUT: Frame[%0d], Line[%0d], Pixel[%0d], R=%0d,  
        current_frame, current_line, current_pixel,  
        item.r_data, item.g_data, item.b_data), UVM_HIGH)  
  
function void build_phase(uvm_phase phase);  
    super.build_phase(phase);  
    if (!uvm_config_db#(virtual line_buf_intf)::get(  
        this, "", "line_buf_if", vif  
    ))  
        `uvm_fatal("DRV", "Failed to get virtual interface")  
endfunction
```

```
function void report_phase(uvm_phase phase);  
    super.report_phase(phase);  
    `uvm_info("SCB", "=====")  
    `uvm_info("SCB", "      SCOREBOARD FINAL REPORT")  
    `uvm_info("SCB", "=====")  
    `uvm_info("SCB", $sformatf(  
        "Total Input Pixels: %0d", total_input_...  
    `uvm_info("SCB", $sformatf(  
        "Total Output Pixels: %0d", total_output_...  
    `uvm_info("SCB", $sformatf("PASS Count: %0d", pass...  
    `uvm_info("SCB", $sformatf("FAIL Count: %0d", fail...  
  
    if (fail_count == 0 && pass_count > 0)  
        `uvm_info("SCB", "*** TEST PASSED ***", UVM_NO...  
    else `uvm_error("SCB", "*** TEST FAILED ***")  
  
    `uvm_info("SCB", "=====")  
endfunction
```



```
=====  
SCOREBOARD FINAL REPORT  
=====  
Total Input Pixels: 120  
Total Output Pixels: 120  
PASS Count: 120  
FAIL Count: 0  
*** TEST PASSED ***  
=====
```

Verbosity Level

- UVM_NONE(0): 무조건 출력
- UVM_LOW(100): 중요한 정보만
- UVM_MEDIUM(200): 보통 정보 (기본값)
- UVM_HIGH(300): 상세 정보
- UVM_FULL(400): 모든 정보
- UVM_DEBUG(500): 디버그 정보

Macro & Method

✓ Sequence-Related Macros

- `'uvm_declare_p_sequencer(SEQUENCER)`
 - `m_sequencer(Generic)`를 `p_sequencer(Specific)`로 변환하여 선언
 - Sequence 내부에서 Sequencer가 가진 변수나 핸들(Virtual Interface 등)에 직접 접근할 때 필수
- `'uvm_create(SEQ_OR_SEQ_ITEM)`
 - Factory를 통해 Transaction 객체 생성만 수행함 (Randomize 및 Send는 안 함)
- `'uvm_send(SEQ_OR_SEQ_ITEM)`
 - 이미 생성(및 처리)된 객체를 Driver로 전송
 - `uvm_create`와 짹을 이루어 사용하거나, 이미 만들어진 객체를 재전송할 때 사용
- `'uvm_do(SEQ_OR_SEQ_ITEM)`
 - 생성 + 랜덤화 + 드라이버 전송까지 전 과정을 한 번에 처리
 - 가장 보편적으로 사용됨
- `'uvm_do_with(SEQ_OR_SEQ_ITEM, CONSTRAINTS)`
 - `uvm_do`와 같으나, In-line Constraint를 추가하여 랜덤화 조건을 즉석에서 부여함

Macro & Method

Utility and Field Macros

- `'uvm_object_utils`
 - `Transaction`, `Sequence`, `Config` 등 `Dynamic Object` 클래스를 `Factory`에 등록
 - `Factory` 등록 + `copy()`, `compare()`, `print()` 등 기본 메서드 인프라 제공
- `'uvm_component_utils`
 - `Driver`, `Env`, `Agent` 등 `Static Component` 클래스를 `Factory`에 등록
 - `Factory` 등록 + 계층 구조(Hierarchy) 및 Phase 관리 기능 지원
- `'uvm_field_*`
 - 클래스 내부 변수들을 시스템에 알려주어, 주요 메서드(`print`, `copy`, `pack` 등)를 자동으로 구현해줌
 - 반드시 `_begin`과 `_end` 사이에 작성해야 함

Macro & Method

Method (주요 메서드)

```
module tb_line_buffer_uvm;
    initial begin
        // Register interface with config_db
        uvm_config_db#(virtual line_buf_intf)::set(null, "", "line_buf_if", lb_if);
        // Start UVM test
        run_test("line_buf_test");
    end
    class line_buf_output_monitor extends uvm_monitor;
        function void build_phase(uvm_phase phase);
            super.build_phase(phase);
            if (!uvm_config_db#(virtual line_buf_intf)::get(
                this, "", "line_buf_if", vif
            ))
                `uvm_fatal("OUT_MON", "Failed to get virtual interface")
        endfunction
    endclass
endmodule
```

- `uvm_config_db#(타입)::set(context, inst_name, field_name, value)`
 - 타입: 저장할 데이터 타입 (`virtual interface, int, string, object` 등)
 - context: 설정을 하는 컴포넌트 (보통 `this`)
 - `this`: UVM 클래스 내부, `null`: UVM 외부
 - `inst_name`: 대상 컴포넌트의 상대 경로 (와일드카드 * 사용 가능)
 - `field_name`: 설정 이름 (문자열)
 - `value`: 전달할 값
- `uvm_config_db#(타입)::get(cntxt, inst_name, field_name, variable)`
 - 타입: 가져올 데이터 타입
 - context: 가져오는 컴포넌트 (보통 `this`)
 - `inst_name`: 자신의 상대 경로 (보통 "" 빈 문자열)
 - `field_name`: 설정 이름
 - `variable`: 값을 저장할 변수

Macro & Method

Method (주요 메서드)

```
class line_buf_environment extends uvm_env;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        agt = line_buf_agent::type_id::create("agt", this);
        scb = line_buf_scoreboard::type_id::create("scb", this);
    endfunction
```

```
class line_buf_test extends uvm_test;
    task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        uvm_info("TEST", "Starting Line Buffer UVM Test", UVM_NONE)

        seq.start(env.agt.sqr);

        #1000;

        `uvm_info("TEST", "Line Buffer UVM Test Complete", UVM_NONE)
        phase.drop_objection(this);
    endtask
endclass
```

- `type_id::create("name", parent)`
 - Factory를 통한 객체 생성
- `raise_objection / drop_objection`
 - Phase 종료 시점 제어
 - raise: +1, drop: -1으로 모든 Component의 objection이 0이 될 때 종료
- `uvm_top` (최상위 루트)
 - UVM hierarchy의 최상위 노드
 - 모든 UVM component의 root

04 Control (Orchestration)

Sequence & Driver

✓ Sequence & Sequencer

- **Sequence** : DUT에 입력할 데이터(Transaction)을 생성하고 순서를 결정하는 객체
- **Sequencer** : Sequence item(Transaction)의 중재(Arbitration), 만약 여러 **sequence**에서 `Start_item()` 호출했을 때 순서 정렬

✓ 왜 필요한가?

SystemVerilog로 직접 TB를 작성하면 테스트를 할 때마다 Driver코드를 바꾸어야하는 문제가 있다. 그래서 역할을 나누어서 Driver는 보내기만 하고 데이터 생성 및 제어는 Sequence가 하면 하나의 Driver(재사용성)로 여러 테스트(Random, Error, Corner case)를 진행할 수 있다.

✓ Work Flow

- **Sequence** : 데이터 생성 및 순서 결정
- **Sequencer** : 중재자 역할
- **Driver** : 데이터를 신호로 바꾸어서 DUT에 발송(실행)

Sequence & Driver

✓ 예시 코드 : Line Buffer Verification

[Sequence]

```
class line_buf_sequence extends uvm_sequence #(line_buf_seq_item);
    task generate_line(int frame_num, int line_num, bit is_active);
        for (i = 0; i < HBP; i++) begin
            item = line_buf_seq_item::type_id::create("item");
            start_item(item);
            // Control Signal
            item.vsync = (line_num < VSW) ? 1'b1 : 1'b0;
            item.hsync = 1'b0;
            item.de = 1'b0;
            // RGB Data
            item.r_data = 0;
            item.g_data = 0;
            item.b_data = 0;
            // Counter Value
            item.frame_num = frame_num;
            item.line_num = line_num;
            item.pixel_num = HSW + i;
            finish_item(item);
        end
    end
```

[Sequencer]

```
class line_buf_agent extends uvm_agent;
    function void build_phase(uvm_phase phase);
        drv = line_buf_driver::type_id::create("drv", this);
        in_mon = line_buf_input_monitor::type_id::create("in_mon", this);
        out_mon = line_buf_output_monitor::type_id::create("out_mon", this);
        sqr = uvm_sequencer #(line_buf_seq_item)::type_id::create("sqr", this);
    endfunction

    // driver와 sequencer의 연결
    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        drv.seq_item_port.connect(sqr.seq_item_export);
    endfunction

endclass
```

[Driver]

```
class line_buf_driver extends uvm_driver #(line_buf_seq_item);
    task run_phase(uvm_phase phase);
        forever begin
            seq_item_port.get_next_item(item);
            @(posedge vif.clk);
            // Control Signals
            vif.i_vsync <= item.vsync;
            vif.i_hsync <= item.hsync;
            vif.i_de <= item.de;
            // RGB Data
            vif.i_r_data <= item.r_data;
            vif.i_g_data <= item.g_data;
            vif.i_b_data <= item.b_data;
        end
        seq_item_port.item_done();
    endtask
endclass
```

1. Start_item() 코드가 실행되면 Driver에서 get_next_item() 코드가 실행될 때까지 대기(Sequencer 중재)
2. Driver에서 get_next_item() 실행되면 Sequence의 다음 코드 진행해서 Transaction 생성(아직 전송대기)
3. finish_item(item) 코드 실행하면 sequencer가 driver에 데이터 전송, 현재 sequence는 finish_item 코드에서 대기
4. Driver에서 item_done() 코드 호출한 후에 sequence는 다음 루프를 돌 수 있다.

Virtual Sequencer & Virtual Sequence

✓ Virtual Sequencer란

? Driver에 연결되지 않고, 다른 Sequencer들을 참조만 하는 Sequencer

- 여러 개의 Sequencer를 한 곳에서 제어하기 위한 컨트롤 타워

✓ 왜 필요한가?

문제점

- 단일 Agent 테스트: Sequence → Sequencer → Driver 흐름이 간단함
- 하지만 여러 Agent를 동시에 제어해야 한다면?
 - Ethernet Agent와 CPU Agent를 동시에 동작
 - 두 Agent 간 타이밍 조율이 필요함

해결책: Virtual Sequencer

- 여러 Sequencer에 대한 참조(Reference)를 가짐
- 각 Sequencer에 Sequence를 실행시킬 수 있음
- 자신은 Driver를 갖지 않음 (Only 제어 역할)

Virtual Sequencer & Virtual Sequence

✓ Virtual Sequence란?

- Virtual Sequencer에서 실행되는 Sequence
- 여러 Sub-sequencer의 Sequence들을 조율함

✓ 일반 Sequence vs Virtual Sequence

구분	일반 Sequence	Virtual Sequence
실행 위치	일반 Sequencer	Virtual Sequencer
Item 생성	함 (Transaction)	안 함
역할	Transaction 생성/전송	다른 Sequence들 실행 제어
Macro	uvm_do	uvm_do_on 또는 start()

Virtual Sequencer & Virtual Sequence

✓ 동작 원리

1. Virtual Sequencer 생성 (핸들 선언)

```
class simple_virtual_sequencer extends uvm_sequencer;
  `uvm_component_utils(simple_virtual_sequencer)

  eth_sequencer eth_seqr;
  cpu_sequencer cpu_seqr;

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

→ 실제 Sequencer들에 대한 핸들 선언

2. Env의 Connect Phase (핸들 연결)

```
class comm_env extends uvm_env;
  ethernet_agent eth_agent;
  cpu_agent cpu_agent;
  simple_virtual_sequencer virt_seqr;

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

    virt_seqr.eth_seqr = eth_agent.sequencer;
    virt_seqr.cpu_seqr = cpu_agent.sequencer;
  endfunction
endclass
```

→ Virtual Sequencer의 핸들에 실제 Sequencer 연결

Virtual Sequencer & Virtual Sequence

✓ 동작 원리

3. Virtual Sequence 작성 및 실행

```
class simple_virt_seq extends uvm_sequence;
  `uvm_object_utils(simple_virt_seq)
  `uvm_declare_p_sequencer(simple_virtual_sequencer)

  cpu_config_seq conf_seq;
  eth_frame_seq frame_seq;

  virtual task body();
    `uvm_do_on(conf_seq, p_sequencer.cpu_seqr)
    `uvm_do_on(frame_seq, p_sequencer.eth_seqr)
  endtask
endclass
```

→ p_sequencer로 각 Sequencer 접근

⇒ Virtual Sequencer는 여러 Sequencer에 대한 참조를 관리하고,
Virtual Sequence는 이들을 통해 다중 Agent의 동작을 조율함

Virtual Sequencer & Virtual Sequence

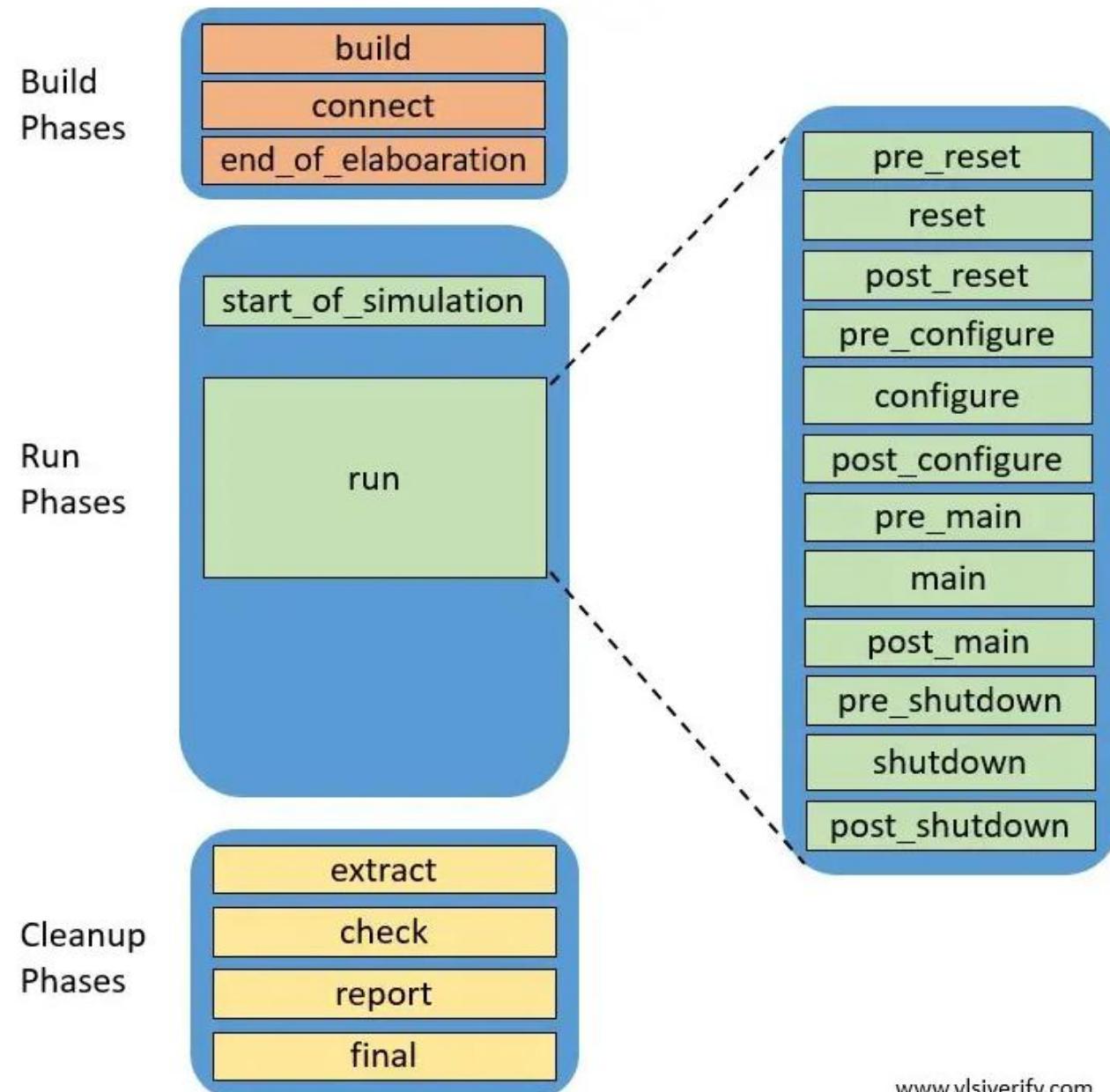
✓ p_sequencer

- m_sequencer를 구체적인 타입으로 형변환하여, Virtual Sequencer의 멤버에 접근 가능
- 여러 Sequencer를 한 번에 제어하여 Virtual Sequence의 핵심 동작을 가능하게 함
- 일반 Sequence에서는 불필요하지만, Virtual Sequence를 작성할 때는 필수

구분	m_sequencer	p_sequencer
타입	uvm_sequencer_base	사용자 정의 Sequencer 타입
선언	자동 제공	uvm_declare_p_sequencer 필요
멤버 접근	불가	가능
사용	일반 Sequence	Virtual Sequence

Phase

✓ Phase



UVM Phase: 왜 필요한가?

Problem

- **Race Conditions:** 컴포넌트 간 생성 및 연결 순서가 보장되지 않아 초기화 오류 발생.
- **Unpredictable Behavior:** 시뮬레이터마다, 혹은 실행할 때마다 동작 순서가 달라지는 문제 발생.
- **Premature Termination:** 테스트가 완료되기 전에 시뮬레이션이 조기 종료되는 문제.

Solution

- UVM Phase가 라이프사이클을 단계별로 강제하여 동기화 보장

Phase

Build Phase

```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // 1. 설정 가져오기 (Get Config)
    if(!uvm_config_db#(virtual my_if)::get(this, "", "vif", vif))
        `uvm_fatal("NO_VIF", "Virtual Interface not set!")

    // 2. 자식 생성 (Create Child) - Top-Down
    driver = my_driver::type_id::create("driver", this);
endfunction
```

1. 주요 기능: 컴포넌트 생성 및 인스턴스화

- uvm_component 객체를 메모리에 할당하고 생성.
- type_id::create() 사용 (Factory Pattern).

2. 실행 순서: Top-Down

- uvm_root → uvm_test → uvm_env → uvm_agent → Driver
- 부모가 먼저 존재해야 자식을 그 안에 생성할 수 있기 때문. (UVM Phase 중 유일한 Top-Down).

3. 추가 역할: 설정값 적용 (Configuration)

- 컴포넌트 생성 시점에 uvm_config_db::get()을 호출하여, 동작 모드(Active/Passive)나 파라미터를 확정 짓는 단계

.

Phase

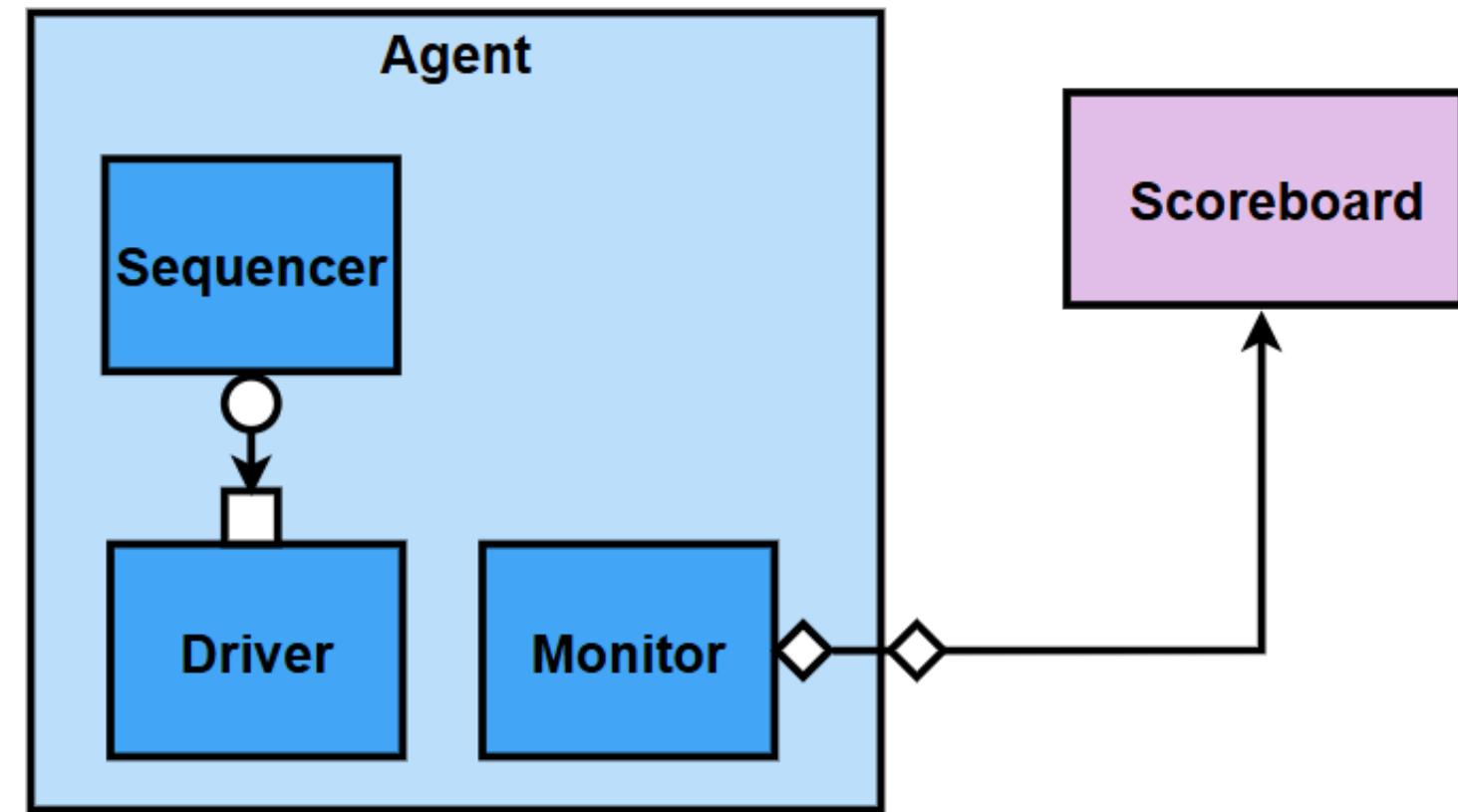
✓ Connect Phase

```
class my_agent extends uvm_agent;
    // ... (component declarations) ...

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

        // Driver(Consumer)와 Sequencer(Producer) 연결
        if(get_is_active() == UVM_ACTIVE) begin
            driver.seq_item_port.connect(sequencer.seq_item_export);
        end

        // 하위(Monitor)의 포트를 상위(Agent)의 포트로 연결하여 외부 노출
        monitor.ap.connect(this.ap);
    endfunction
endclass
```



1. 주요 기능: TLM Network 구성

- build_phase에서 생성된 독립적인 컴포넌트들의 Port와 Export를 연결.
- 모든 컴포넌트의 생성이 완료된 후 실행됨.

2. 실행 순서: Bottom-Up

- 하위 컴포넌트부터 연결을 확장하고 상위로 올라옴.
- 하위 계층의 포트 연결이 먼저 완료되어야, 상위 계층에서 이를 끌어서 외부로 노출할 수 있기 때문
- 에

Phase

Run Phase

```
class line_buf_driver extends uvm_driver #(line_buf_s
    task run_phase(uvm_phase phase);
        //initialize signals
        vif.i_vsync <= 0;
        vif.i_hsync <= 0;
        vif.i_de <= 0;
    class line_buf_input_monitor extends uvm_monitor;
        task run_phase(uvm_phase phase);
            line_buf_seq_item item;
            @(posedge vif.rst_n);

            forever begin
    class line_buf_output_monitor extends uvm_monitor;
        task run_phase(uvm_phase phase);
            @(posedge vif.rst_n);
```



```
class line_buf_test extends uvm_test;
    task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        `uvm_info("TEST", "Starting Line Buffer UVM Test", UVM_NONE)

        seq.start(env.agt.sqr);
        #1000;
        `uvm_info("TEST", "Line Buffer UVM Test Complete", UVM_NONE)
        phase.drop_objection(this);
    endtask
endclass
```

1. 주요 기능: Time Consuming

- 유일하게 task로 구현된 Phase. (나머지는 function).
- Delay, Wait, Drive 등 실제 물리적 시간이 소모되는 시뮬레이션 동작 수행.

2. 실행 특성: Parallel Execution

- Build(Top-down) & Connect(Bottom-up)와 달리 계층 순서가 없음.
- run_phase 진입 시, uvm_top부터 말단 Driver까지 모든 컴포넌트가 동시에 시작됨.

3. 종료 메커니즘: Objection System

- "단 한 명이라도 손을 들고 있으면(raise) 끝나지 않는다."
- 모든 컴포넌트의 Objection Count가 0이 되는 순간, run_phase가 강제 종료되고 extract_phase로 이동

Phase

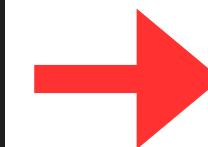
✓ Clean-Up Phase

```
class line_buf_scoreboard extends uvm_scoreboard;
    function void report_phase(uvm_phase phase);
        super.report_phase(phase);
        `uvm_info("SCB", "=====", UVM_NONE)
        `uvm_info("SCB", "      SCOREBOARD FINAL REPORT      ", UVM_NONE)
        `uvm_info("SCB", "=====", UVM_NONE)
        `uvm_info("SCB", $sformatf(
            "Total Input Pixels: %0d", total_input_pixels), UVM_NONE)
        `uvm_info("SCB", $sformatf(
            "Total Output Pixels: %0d", total_output_pixels), UVM_NONE)
        `uvm_info("SCB", $sformatf("PASS Count: %0d", pass_count), UVM_NONE)
        `uvm_info("SCB", $sformatf("FAIL Count: %0d", fail_count), UVM_NONE)

        if (fail_count == 0 && pass_count > 0)
            `uvm_info("SCB", "*** TEST PASSED ***", UVM_NONE)
        else `uvm_error("SCB", "*** TEST FAILED ***")

        `uvm_info("SCB", "=====", UVM_NONE)
    endfunction

endclass
```



```
uvm_test_top.env.scb [SCB] =====
uvm_test_top.env.scb [SCB]      SCOREBOARD FINAL REPORT
uvm_test_top.env.scb [SCB] =====
uvm_test_top.env.scb [SCB] Total Input Pixels: 120
uvm_test_top.env.scb [SCB] Total Output Pixels: 120
uvm_test_top.env.scb [SCB] PASS Count: 120
uvm_test_top.env.scb [SCB] FAIL Count: 0
uvm_test_top.env.scb [SCB] *** TEST PASSED ***
uvm_test_top.env.scb [SCB] =====
```

Report_phase

- Final Output : 앞서 판정된 결과를 보기 편한 형태로 가공하여 출력한다.
- 콘솔/로그 출력: "TEST PASSED" 또는 "TEST FAILED"와 같은 최종 문구를 터미널과 로그 파일에 출력한다.
- 통계 정보 표시: 시뮬레이션 동안 발생한 UVM_INFO, UVM_WARNING, UVM_ERROR, UVM_FATAL의 총개수를 집계하여 보여준다.
- 파일 저장: 필요한 경우, 커버리지 리포트나 별도의 요약 파일을 디스크에 쓴다.

06 Conclusion

Conclusion

✓ 배운 점

- 구조적 분리 : 정적 환경(env, agent, driver, monitor)와 동적 시나리오(Sequence)의 분리를 통해 UVM 환경에서 구조화를 통해 재사용성과 유연성을 확보하고, 다양한 시나리오에 확장에 용이한 구조를 습득했습니다.
- 하드웨어 추상화 : RAL을 도입하여 물리적인 주소가 아닌 객체(Object) 중심의 추상화된 하드웨어 검증 기법을 학습을 할 수 있었습니다.
- Line Buffer 검증 : 학습한 이론을 바탕으로 실제 Line Buffer 검증을 하면서 Transaction의 흐름과 component들의 상호작용을 실제로 적용하며 이해할 수 있습니다.

Reference

- [1] Accellera, "UVM (Universal Verification Methodology) User's Guide", accellera_uvm_userguide.pdf
- [2] "UVM Testbench 작성", WikiDocs, <https://wikidocs.net/book/8302>
- [3] ChipVerify, <https://www.chipverify.com/>
- [4] VLSI Verify, <https://vlsiverify.com/>

감사합니다