1. What is a UVM scoreboard?

A UVM scoreboard is a component in a UVM (Universal Verification Methodology) testbench used to compare the actual outputs of the Design Under Test (DUT) with the expected results. It plays a key role in functional verification by tracking and checking the correctness of transactions generated by the DUT. A scoreboard usually receives transactions from the monitor or other components, compares the observed data against expected values, and determines whether the DUT behavior is correct.

1. What is the use of a UVM scoreboard?

* Functional Verification: The primary role of a UVM scoreboard is to validate the correctness of the DUT by comparing its outputs to expected results. It checks that the DUT's output is correct for a given input.
* Tracking Transactions: It keeps track of transactions (e.g., read or write operations) and ensures that they occur in the expected order.
* Error Detection: If a mismatch between expected and actual results is found, the scoreboard reports an error.
* Pass/Fail Evaluation: Based on the comparison of expected vs actual results, the scoreboard can determine whether the DUT passed or failed a test scenario.

1. Write a uvm\_scoreboard template & explain each line.

class my\_scoreboard extends uvm\_scoreboard;

`uvm\_component\_utils(my\_scoreboard)

// Declare a queue to hold expected transactions

uvm\_blocking\_transport\_if #(my\_transaction) my\_queue;

// Constructor

function new(string name = "my\_scoreboard", uvm\_component parent = null);

super.new(name, parent);

endfunction

// Build phase: Create necessary components

virtual function void build\_phase(uvm\_phase phase);

super.build\_phase(phase);

// Initialize the queue or transaction interfaces

my\_queue = new();

endfunction

// Connect phase: Connect to other components

virtual function void connect\_phase(uvm\_phase phase);

super.connect\_phase(phase);

// Connect the monitor's output (observed transactions) to the scoreboard

monitor.my\_export.connect(my\_queue);

endfunction

// Comparison and checking phase

virtual task check\_phase(uvm\_phase phase);

my\_transaction txn;

// Check for transactions in the queue and compare with expected

foreach (my\_queue) begin

txn = my\_queue.get\_next\_item();

// Compare txn with expected results

if (txn != expected\_txn) begin

`uvm\_error("SCOREBOARD", "Mismatch found")

end

end

endtask

endclass

* class my\_scoreboard extends uvm\_scoreboard;: The my\_scoreboard class extends uvm\_scoreboard, a base class for implementing a scoreboard in a UVM environment.
* uvm\_blocking\_transport\_if #(my\_transaction) my\_queue;: Declares a queue or interface to hold the transactions. In this case, my\_transaction is a custom transaction type. The queue holds the data received from the monitor or other components.
* function new(string name = "my\_scoreboard", uvm\_component parent = null);: Constructor to initialize the scoreboard and optionally set its name and parent component.
* virtual function void build\_phase(uvm\_phase phase);: The build\_phase is used to instantiate components and initialize any objects or queues required for the scoreboard to funct
* my\_queue = new();: Initializes the transaction queue in the build phase.
* virtual function void connect\_phase(uvm\_phase phase);: In the connect\_phase, the scoreboard is connected to other components (like the monitor). The monitor provides transactions, which are passed to the scoreboard for comparison.
* monitor.my\_export.connect(my\_queue);: Connects the monitor's transaction export (output) to the scoreboard's queue to receive observed transactions.
* virtual task check\_phase(uvm\_phase phase);: In the check\_phase, the scoreboard compares the received transactions with expected values. If there is a mismatch, an error is reported.
* foreach (my\_queue): Iterates through all the transactions in the queue and performs the comparison between observed (txn) and expected (expected\_txn) transactions.
* if (txn != expected\_txn): Compares each received transaction against an expected transaction. If they do not match, an error is triggered.

1. How is the scoreboard connected to different components?

* Monitor to Scoreboard: The monitor sends transactions to the scoreboard for checking. The monitor captures the signals from the DUT and generates transactions, which are passed to the scoreboard via a queue or transaction interface.

Example: monitor.my\_export.connect(my\_scoreboard.my\_queue);

* Scoreboard and DUT: The scoreboard does not directly interact with the DUT; rather, it compares the DUT's output (observed by the monitor) to expected values.
* Sequencer/Driver to Scoreboard: The sequencer may generate expected transactions that are sent to the scoreboard, or the driver can send expected data to the scoreboard during the stimulus generation phase.

1. What is an in-order and out-order scoreboard?

* In-order scoreboard: In an in-order scoreboard, transactions are expected to arrive in a specific order, and the scoreboard verifies that the received transactions match the expected order. It ensures that the sequence of operations is preserved.
* Out-order Scoreboard: In an out-of-order scoreboard, transactions can arrive in any order, and the scoreboard does not enforce any specific order. It compares each received transaction with the expected ones but allows for flexibility in the arrival sequence.

1. Explain below components in a UVM testbench with code:
   1. top

* This is the top-level container for the entire testbench.
* It typically instantiates and connects the environment, test, and other components.
* Example:

class top extends uvm\_top;

`uvm\_component\_utils(top)

// Declare environment and test

my\_env env;

my\_test test;

function new(string name = "top", uvm\_component parent = null);

super.new(name, parent);

endfunction

// Build phase

virtual function void build\_phase(uvm\_phase phase);

super.build\_phase(phase);

// Instantiate environment and test

env = my\_env::type\_id::create("env", this);

test = my\_test::type\_id::create("test", this);

endfunction

endclass

* 1. tes
* The test defines the scenario to be run, including configuring the environment and starting the simulation.
* Example:

class my\_test extends uvm\_test;

`uvm\_component\_utils(my\_test)

my\_env env;

function new(string name = "my\_test", uvm\_component parent = null);

super.new(name, parent);

endfunction

virtual function void build\_phase(uvm\_phase phase);

super.build\_phase(phase);

env = my\_env::type\_id::create("env", this);

endfunction

virtual task run\_phase(uvm\_phase phase);

super.run\_phase(phase);

// Stimulus generation or driving transactions

endtask

endclass

* 1. env
* The environment contains all the components necessary for verification (agents, scoreboard, monitor, driver, sequencer).
* Example:

class my\_env extends uvm\_env;

`uvm\_component\_utils(my\_env)

my\_agent agent;

my\_scoreboard scoreboard;

function new(string name = "my\_env", uvm\_component parent = null);

super.new(name, parent);

endfunction

virtual function void build\_phase(uvm\_phase phase);

super.build\_phase(phase);

agent = my\_agent::type\_id::create("agent", this);

scoreboard = my\_scoreboard::type\_id::create("scoreboard", this);

endfunction

endclass

1. How can you define custom types for use in your env.

Custom types in UVM can be defined using classes for transactions or other objects. These custom types are typically used for transactions in agents (driver/monitor) and scoreboards.

Example:

class my\_transaction extends uvm\_sequence\_item;

`uvm\_object\_utils(my\_transaction)

// Declare fields for the transaction

bit [7:0] data;

bit valid;

// Constructor

function new(string name = "my\_transaction");

super.new(name);

endfunction

// Print function for the transaction

virtual function void do\_print();

`uvm\_info("MY\_TRANSACTION", $sformatf("data = %0h, valid = %b", data, valid), UVM\_LOW);

endfunction

endclass