1. Explain:
   1. Block level testbench.

A block-level testbench focuses on testing a specific block or unit of the Design Under Test (DUT), typically a single module or functional block. This type of testbench includes only the necessary components to verify the functionality of the DUT in isolation, without the full integration with the rest of the design. It usually has the following components:

* DUT: The unit or block that is being tested.
* Driver: Drives the signals to the DUT.
* Monitor: Observes the DUT's outputs and generates transactions.
* Sequencer: Coordinates the sequence of transactions and provides data to the driver.
* Scoreboard: Compares the DUT's outputs with the expected results.
  1. Integration Level Testbench

An integration-level testbench verifies the interaction between multiple components or modules within a system. It focuses on the integration of several blocks or units to check that they work together as expected. This type of testbench is typically more complex and includes the following components:

* Multiple DUTs: Integrates multiple functional blocks or sub-systems.
* Interconnects: Models the communication between various blocks of the design.
* Complex Stimulus Generation: Stimuli might come from multiple sources or be generated in response to multiple inputs.
* Scoreboard: Verifies that the integration of multiple blocks works correctly by comparing actual and expected results across all interconnected components.
* Monitors: Monitors the outputs of each DUT or sub-system and ensures the system behaves correctly in various interaction scenarios.

1. How is coverage collection done in a UVM testbench?

Coverage is a measure of how much of the design has been exercised during simulation. UVM provides coverage collection capabilities to help ensure that the testbench covers all possible functional scenarios.

In UVM, coverage collection is done using UVM coverage mechanisms like:

* Functional Coverage: Measures whether all important states or scenarios in the design have been reached during the test. For example, checking if all combinations of input stimulus are exercised.
* Code Coverage: Ensures that all lines, branches, and conditions in the design’s RTL are exercised. This is typically done through simulation tools and is a separate activity from functional coverage.
* Toggle Coverage: Monitors how many times a signal changes state (from 0 to 1 or 1 to 0). This is useful to ensure that the signal paths are being driven correctly.
* Assertion Coverage: Tracks the number of times assertions are triggered, helping to determine if critical conditions in the design are being checked.

1. What is factory override in UVM?

Factory Override in UVM is a mechanism that allows for replacing components in the testbench dynamically. This is particularly useful when you need to replace or customize components (such as agents, drivers, monitors, etc.) without changing the testbench code. Factory override is useful in situations where you want to create different versions of components (for example, a default driver and a mock driver for specific testing). The factory allows a user to register a new class type to replace an existing one. When the testbench calls create(), the factory can instantiate the overridden version of the class.

Example:

// Default class

class my\_driver extends uvm\_driver;

// Driver implementation

endclass

// Overridden class

class my\_mock\_driver extends uvm\_driver;

// Mock driver implementation

endclass

// In the test

uvm\_factory::set\_type\_override("my\_driver", "my\_mock\_driver");

1. What is BFM?

A BFM (Bus Functional Model) is a model used in verification to simulate the behavior of a bus interface or communication protocol. It abstracts away low-level details and provides a higher-level interface to interact with the DUT during simulation. The BFM simplifies the interaction between the testbench and the DUT by modeling the protocol, providing an easy-to-use interface to generate and receive transactions. The BFM usually connects to the driver in UVM, where it generates and monitors the transactions based on the protocol (e.g., AXI, APB, SPI).

Example:

class my\_bfm extends uvm\_driver;

// BFM implementation for a protocol

// Includes transaction generation, handshake, and protocol control

endclass

1. How do you assign a Virtual Interface in UVM?

In UVM, a virtual interface is used to connect the UVM components (driver, sequencer, monitor) to the DUT's interface. It allows the UVM components to interact with the DUT at the interface level.

virtual interface my\_dut\_if;

// Interface definition

endinterface

class my\_driver extends uvm\_driver;

// Declare a virtual interface

virtual my\_dut\_if vif;

// Assign virtual interface in build phase

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

super.build\_phase(phase);

vif = my\_dut\_if::get\_virtual\_interface();

endfunction

endclass

1. Write a code to connect sequencer &driver in the connect phase.

In the connect phase of UVM, the sequencer is connected to the driver so that the sequencer can provide the transactions to the driver. This connection allows the driver to receive the sequence items from the sequencer.

class my\_agent extends uvm\_agent;

`uvm\_component\_utils(my\_agent)

my\_sequencer sequencer;

my\_driver driver;

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

super.new(name, parent);

endfunction

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

super.build\_phase(phase);

sequencer = my\_sequencer::type\_id::create("sequencer", this);

driver = my\_driver::type\_id::create("driver", this);

endfunction

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

super.connect\_phase(phase);

driver.seq\_item\_port.connect(sequencer.seq\_item\_export);

endfunction

endclass

1. What are the sub components in uvm env? Explain in detail.

A UVM environment (uvm\_env) typically contains several sub-components that form the structure of the testbench and enable stimulus generation, transaction observation, and verification. Common sub-components in a UVM environment include:

* Agent:
  + An agent contains the sequencer, driver, and monitor for interacting with a specific DUT or interface.
  + Driver: Generates signals and interacts with the DUT.
  + Sequencer: Generates sequences of transactions.
  + Monitor: Observes the DUT’s behavior and generates transactions for comparison
* Scoreboard:
  + Compares the DUT's actual behavior with expected results. It is used to validate functional correctness.
* Virtual Sequence:
  + A virtual sequence allows sequences from multiple agents to be coordinated in a higher-level sequence. This is especially useful when testing multiple interfaces.
* Coverage Collectors:
  + Components used to track and collect coverage data (functional, code, toggle, etc.).
* Configurator
  + :Handles configuration of sub-components in the environment, often using UVM’s uvm\_config\_db or uvm\_resource\_db.