1. Write a UVM driver template and explain each line.

The UVM driver is responsible for receiving sequence items from a sequencer and applying them as signals to the Design Under Test (DUT).

class my\_driver extends uvm\_driver #(uvm\_sequence\_item);

`uvm\_component\_utils(my\_driver)

// Declare input port and sequence item

virtual uvm\_sequencer #(uvm\_sequence\_item) sequencer;

uvm\_sequence\_item item;

// Constructor

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

super.new(name, parent);

endfunction

// Main run task that drives the signals

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

forever begin

// Get next item from sequencer

get\_next\_item(item);

// Apply item to DUT signals (simplified)

apply\_to\_dut(item);

// Indicate the item is done processing

item\_done();

end

endtask

// Task to apply the sequence item to DUT signals

task apply\_to\_dut(uvm\_sequence\_item item);

// Implement logic to drive signals to DUT

// e.g., dut\_signal <= item.some\_value;

endtask

endclass

* class my\_driver extends uvm\_driver #(uvm\_sequence\_item);: Defines a new class my\_driver that extends the uvm\_driver base class. This driver will work with uvm\_sequence\_item or any subclass of it.
* uvm\_sequencer #(uvm\_sequence\_item) sequencer;: Declares a reference to the sequencer object that will provide the sequence items to the driver. It specifies the type of sequence item it will work with.
* uvm\_sequence\_item item;: Declares a variable item of type uvm\_sequence\_item to store the sequence item retrieved from the sequencer.
* function new(string name = "my\_driver", uvm\_component parent = null);: Constructor for the driver. It initializes the name and parent component of the driver by calling the superclass constructor.
* super.new(name, parent);: Calls the constructor of the base class (uvm\_driver) to initialize the driver instance.
* virtual task run\_phase(uvm\_phase phase);: The run\_phase task is the main task executed in the UVM driver. The driver continuously requests sequence items and applies them to the DUT.
* get\_next\_item(item);: Retrieves the next sequence item from the sequencer. The driver then processes this item to apply it to the DUT.
* apply\_to\_dut(item);: A custom task to apply the item’s values to the DUT signals (e.g., driving values on a bus or setting control signals).
* item\_done();:Signals to the sequencer that the item has been fully processed and the driver is ready to receive the next item.

1. Explain the below methods in uvm\_driver:
   1. get\_next\_item:This method is used to retrieve the next item from the sequencer.

The driver calls get\_next\_item() when it is ready to receive the next sequence item to apply to the DUT. This method blocks until an item is available.

virtual task get\_next\_item(uvm\_sequence\_item item);

* 1. try\_next\_item: This method attempts to get the next item from the sequencer, but unlike get\_next\_item(), it does not block. It returns immediately, whether or not an item is available.

Useful when the driver wants to check if there is an item available but continue with other tasks if the sequencer does not have an item.

virtual function bit try\_next\_item(output uvm\_sequence\_item item);

* 1. item\_done: This method is used to signal that the current item has been completely processed and the driver is ready to receive the next item

After the driver finishes applying an item to the DUT, it calls item\_done() to notify the sequencer.

virtual task item\_done();

* 1. put: This method is used to send the sequence item to the sequencer.

Typically used when the driver has finished processing an item and the sequencer should be informed that it can proceed with the next item or operation.

virtual task put(uvm\_sequence\_item item);

1. Explain the protocol handshake between a sequencer and driver.

* Sequencer sends a sequence item to the Driver.
* Driver processes the item and applies it to the DUT.
* Once the Driver has finished processing the item, it calls item\_done() to notify the Sequencer.
* The Sequencer can then send the next item to the Driver.

1. What is the difference between a pipelined and non-pipelined driver?

Pipelined Driver:

* The pipelined driver allows the driver to process multiple items concurrently by using internal buffers. While one item is being applied to the DUT, the driver can retrieve the next item from the sequencer.
* This results in better throughput and efficiency as the driver can start processing the next item while the previous one is still being applied to the DUT.

Non-Pipelined Driver:

* In a non-pipelined driver, the driver processes one item at a time. It must finish applying the current item to the DUT before it retrieves the next item from the sequencer.
* This is simpler but might result in lower throughput as each item is processed sequentially.

1. Write a uvm monitor template and explain each line.

A UVM Monitor is used to observe the signals of the DUT and collect data (transactions) for further analysis, usually for checking correctness.

class my\_monitor extends uvm\_monitor;

`uvm\_component\_utils(my\_monitor)

// Declare DUT signals

bit dut\_signal;

// Constructor

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

super.new(name, parent);

endfunction

// Main observation task

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

forever begin

// Monitor DUT signals (simplified)

wait(dut\_signal); // wait for a signal change

collect\_data(); // Collect data when signal changes

end

endtask

// Task to collect data

task collect\_data();

// Implement data collection logic (e.g., storing transactions)

$display("Signal observed at time: %0t", $time);

endtask

endclass

* class my\_monitor extends uvm\_monitor;: Defines a custom monitor class (my\_monitor) extending the uvm\_monitor base class. It is responsible for monitoring signals from the DUT.
* bit dut\_signal;: Declares the DUT signal (dut\_signal) that the monitor will observe.
* function new(string name = "my\_monitor", uvm\_component parent = null);: Constructor for the monitor, where it initializes the monitor with the given name and parent component.
* super.new(name, parent);: Calls the constructor of the base class (uvm\_monitor) to initialize the monitor.
* virtual task run\_phase(uvm\_phase phase);: The main task that continuously monitors the DUT signals. It uses wait to observe changes in dut\_signal and then calls collect\_data() to collect the data.
* collect\_data();: A custom task to collect data whenever the signal of interest (dut\_signal) changes.

1. What does a uvm monitor do in UVM testbench?

In a UVM (Universal Verification Methodology) testbench, a monitor plays a critical role in observing the Design Under Test (DUT) signals or transactions and collecting data for further analysis, typically for debugging or checking correctness. The monitor does not interact directly with the DUT in terms of stimulus; instead, it listens to or "monitors" the DUT’s output signals, transactions, or interfaces to gather useful data, which can later be processed for verification.

1. What is the difference between a monitor and a scoreboard in UVM methodology?

* Monitor: A monitor is used to observe the DUT’s outputs and collect relevant signal data. It is passive and doesn’t directly interact with the DUT but captures signals or transactions for further analysis.
* Scoreboard: A scoreboard is used for comparing the expected results (from a reference model or golden outputs) against the actual outputs from the DUT. It helps check if the DUT behaves as expected by maintaining a record of expected and actual transactions

1. How do you connect a monitor with a scoreboard?

* The monitor generates or collects transactions based on the signals it observes from the DUT.
* The monitor then sends these transactions to the scoreboard for comparison with expected results.
* The scoreboard compares the actual data against the expected data and checks for any mismatches.

Example:

my\_monitor monitor;

my\_scoreboard scoreboard;

monitor.m\_sequence\_item\_port.connect(scoreboard.m\_sequence\_item\_export);