1. Which are three 3 main types of UVM phases & explain each phase in detail with syntax:
   1. Build Phases
      1. Build

* This phase is used to create the components (e.g., driver, monitor, agent) of the testbench. The testbench hierarchy is established in this phase, but components are not yet connected.
* Execution: It runs after the reset phase and before the connect phase.
* Syntax:

class my\_test extends uvm\_test;

`uvm\_component\_utils(my\_test)

my\_env env;

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

super.build\_phase(phase);

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

endfunction

endclass

* + 1. Connect
* Purpose: During this phase, all the components created in the Build phase are connected. This means establishing communication between components (e.g., connecting a driver to a sequencer or a monitor to a scoreboard).
* Execution: After all components have been created and before simulation starts. It is also responsible for resolving any configuration parameters between components.
* Syntax:

class my\_test extends uvm\_test;

`uvm\_component\_utils(my\_test)

my\_env env;

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

super.connect\_phase(phase);

// Connect components, such as driver to sequencer

endfunction

endclass

* + 1. end\_of\_elaboration
* Purpose: The end\_of\_elaboration phase ensures that all the components are fully elaborated and any final setup required for the simulation is completed. This phase typically does not involve creation or connection of components but is used to finalize the setup.
* Execution: This phase comes after the connect phase and before the start\_of\_simulation phase
* Syntax:

class my\_test extends uvm\_test;

`uvm\_component\_utils(my\_test)

function void end\_of\_elaboration\_phase(uvm\_phase phase);

super.end\_of\_elaboration\_phase(phase);

// Perform any last-minute configuration or checks

endfunction

endclass

* 1. Run-time Phases
     1. start\_of\_simulation
* Purpose: This phase marks the beginning of the simulation. Initialization tasks are performed here, such as starting threads or setting up time-sensitive operations.
* Execution: This phase is executed right before the run phase starts.
* Syntax:

class my\_test extends uvm\_test;

`uvm\_component\_utils(my\_test)

function void start\_of\_simulation\_phase(uvm\_phase phase);

super.start\_of\_simulation\_phase(phase);

// Perform any initializations, such as setting simulation time

endfunction

endclass

* + 1. run
* Purpose: This is the main phase where the actual testing occurs. It controls the execution of the simulation, including running the test sequences, driving stimulus to the DUT, and checking the responses.
* Execution: After the start\_of\_simulation phase. This phase is typically where you see the test executing.
* Syntax:

class my\_test extends uvm\_test;

`uvm\_component\_utils(my\_test)

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

super.run\_phase(phase);

// The test logic: Run sequences, check results, etc.

endfunction

endclass

Parallel Run-Time Phases

* pre\_reset, reset, post\_rest
  + Purpose: These phases are used to handle any reset actions on the DUT (Device Under Test).
    - Pre-reset: Prepare for the reset (e.g., stop active processes).
    - Reset: Perform the actual reset.
    - Post-reset: Perform any actions after the reset (e.g., reinitialize states).
    - Syntax:

class my\_test extends uvm\_test;

`uvm\_component\_utils(my\_test)

function void pre\_reset\_phase(uvm\_phase phase);

super.pre\_reset\_phase(phase);

// Tasks before reset

endfunction

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

super.reset\_phase(phase);

// Perform reset

endfunction

function void post\_reset\_phase(uvm\_phase phase);

super.post\_reset\_phase(phase);

// Tasks after reset

endfunction

endclass

* pre\_configure, configure, post\_configure
  + Purpose: These phases are used to handle configuration actions in the testbench.
    - Pre-configure: Prepare for configuration.
    - Configure: Apply the configurations.
    - Post-configure: Finalize configurations.
    - Syntax:

class my\_test extends uvm\_test;

`uvm\_component\_utils(my\_test)

function void pre\_configure\_phase(uvm\_phase phase);

super.pre\_configure\_phase(phase);

// Perform pre-configuration tasks

endfunction

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

super.configure\_phase(phase);

// Perform configuration

endfunction

function void post\_configure\_phase(uvm\_phase phase);

super.post\_configure\_phase(phase);

// Finalize configuration

endfunction

endclass

* pre\_main, main, post\_main
  + Purpose: These phases organize the main testing logic. The pre-main phase sets up the test environment, the main phase executes the test itself, and the post-main phase performs any wrap-up tasks.
  + Syntax:

class my\_test extends uvm\_test;

`uvm\_component\_utils(my\_test)

function void pre\_main\_phase(uvm\_phase phase);

super.pre\_main\_phase(phase);

// Set up the environment before the main phase

endfunction

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

super.main\_phase(phase);

// Execute the main test sequence

endfunction

function void post\_main\_phase(uvm\_phase phase);

super.post\_main\_phase(phase);

// Clean-up tasks after main test

endfunction

endclass

* pre\_shut\_down, shut\_down, post\_shut\_down
  + Purpose: These phases handle the shutting down of the simulation.
    - Pre-shutdown: Prepare the testbench for shutdown.
    - Shutdown: Perform the actual shutdown, finalize simulation results.
    - Post-shutdown: Handle any actions after the simulation has ended.
    - Syntax:

class my\_test extends uvm\_test;

`uvm\_component\_utils(my\_test)

function void pre\_shutdown\_phase(uvm\_phase phase);

super.pre\_shutdown\_phase(phase);

// Prepare to shut down

endfunction

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

super.shutdown\_phase(phase);

// Perform the shutdown

endfunction

function void post\_shutdown\_phase(uvm\_phase phase);

super.post\_shutdown\_phase(phase);

// Clean up after shutdown

endfunction

endclass

* 1. Clean-up phases(extract, check, repost,final)

The Clean-up Phases are responsible for performing any final verification, extracting results, checking correctness, and finalizing the simulation.

* Extract:The extract phase is used to gather final results or data from the testbench, such as extracting coverage information or performance metrics.
* Check: This phase is used to perform any final checks, including comparing the expected results with the actual results and validating the correctness of the DUT.
* Repost: Repost is used to manage any final updates or reposting of results to external components or scoreboards.
* Final: This is the final clean-up phase. It is where the last tasks are executed, and simulation results are finalized. This phase is typically used for overall statistics and reporting.

1. Which UVM phases refer to the function and which phases are a task?

Phases that are functions: Functions in UVM typically do not contain simulation time delays (#) and are meant for operations that are executed immediately without waiting.

* build\_phase: Used to create and configure components.
* connect\_phase: Used to connect the components and resolve connections.
* end\_of\_elaboration\_phase: Ensures the complete elaboration of the testbench hierarchy.
* start\_of\_simulation\_phase: Marks the start of the simulation.
* run\_phase: Used for driving the test, running sequences, and checking the responses.
* pre\_reset\_phase, reset\_phase, post\_reset\_phase: Used for reset-related activities.
* pre\_configure\_phase, configure\_phase, post\_configure\_phase: Used for configuring and initializing components.
* pre\_main\_phase, main\_phase, post\_main\_phase: The main test execution phases.
* pre\_shutdown\_phase, shutdown\_phase, post\_shutdown\_phase: Handles testbench shutdown operations.
* Example:

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

super.build\_phase(phase);

// Component creation and setup

Endfunction

Phases that are tasks: Tasks in UVM typically involve actions that can consume simulation time, such as waiting for events or generating delays.

* run\_phase: Although it appears as a function, it is often implemented as a task in user code. This is because it often involves sequence execution, which includes delays (#), event synchronization, and other operations that consume time.
* extract\_phase, check\_phase, repost\_phase, final\_phase: These are the clean-up and final validation phases. These often include checks and collecting results, which may involve delays.
* Example:

task run\_phase(uvm\_phase phase);

super.run\_phase(phase);

// Test logic, with simulation time delays

endtask

1. How can you start a UVM phase?

In UVM, phases are automatically triggered by the UVM Scheduler, which orchestrates the execution of all phases in the correct order. The UVM framework itself manages the phase transitions from one to another, but you can explicitly start and control these phases in some cases.

* Starting a UVM phase manually:Typically, you don't need to manually start a UVM phase, because UVM phases are scheduled by the UVM scheduler, which follows a specific order defined by the UVM methodology. However, you can initiate a phase manually or explicitly synchronize with phases if necessary.

Example:

uvm\_phase my\_phase;

my\_phase.start();

* Phase Scheduling and Manual Control:UVM uses the uvm\_scheduler to automatically schedule the different phases of the testbench. You typically don't manually start phases unless you are implementing your own custom scheduling logic.However, if you need to invoke or schedule phases manually, you can use the uvm\_scheduler API or the phase object:

Example:

// Manually triggering a phase

uvm\_scheduler::get().run\_phase(start\_phase, phase);

Notes:

* UVM Phases are Ordered: UVM phases are executed in a predefined order (Build → Connect → Start → Run → Reset → Shutdown → Final), and manual intervention is typically unnecessary for this ordering.
* Automated Scheduling: The UVM Scheduler manages the transitions between phases, and phases are typically executed in a sequential manner unless parallelism is explicitly requested.
* Testbench Control: If you want to start specific phases earlier or out-of-order, you can manage synchronization points or use explicit fork/join constructs for parallel execution within phases.

