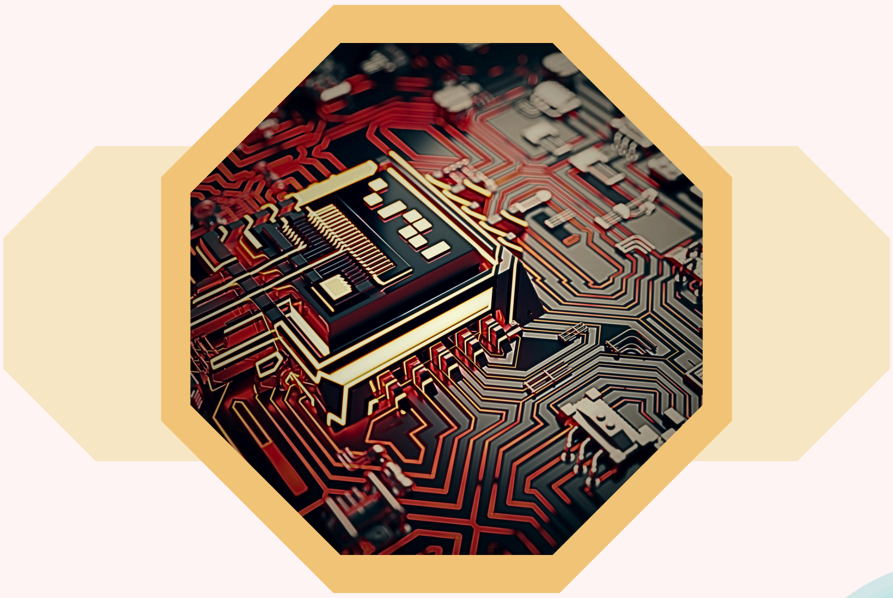
UVM INTERVIEW HANDBOOK



JAIRAJ MIRASHI
DESIGN VERIFICATION ENGINEER

Basic Level Questions

1. Why do we need UVM methodology?

The Universal Verification Methodology (UVM) is a standardized methodology for verifying digital designs that helps develop a reusable, modular, and well-structured verification test bench. UVM provides pre-defined base classes and allows users to extend and reuse pre-defined methods.

UVM uses a set of rules and approaches for verification that allows different teams to collaborate more effectively and understand each other's code. This helps to improve communication across teams.

UVM also provides standardized ways like verbosity control, phases, analysis ports for communication, pre-built components like uvm_driver, uvm_monitor, uvm_env, etc

2. Difference between uvm_object and uvm_component

1) The uvm_transaction class is inherited from uvm_object that adds additional information of a timing, notification events, and recording interface.
2) The uvm_sequence_item class is derived from the uvm_transaction class that adds basic functionality for sequence and sequence items like get_sequence_id, set_sequencer, get_sequence, etc.
3) It is important to note that uvm_transaction usage is deprecated as a base class for user-defined transactions. Hence, the uvm_sequence_item class shall be used as a base class for user-defined transactions.

3. Difference between uvm_transaction and uvm_sequence_item

1) The uvm_transaction class is inherited from uvm_object that adds additional information on timing, notification events, and recording interface.
2) The uvm_sequence_item class is derived from the uvm_transaction class that adds basic functionality for sequence and sequence items like get_sequence_id, set_sequencer, get_sequence, etc.
3) It is important to note that uvm_transaction usage is deprecated as a base class for user-defined transactions. Hence, the uvm_sequence_item class shall be used as a base class for user-defined transactions

.

4. Difference between create and new()

The new method is a constructor for SystemVerilog classes to create an object instance.

The create() method of the wrapper class is used to create objects for the uvm_object and uvm_component class which is commonly known as factory registration.

In UVM based testbench, it is valid to use a new() function to create class objects, but factory registration has its benefits. The UVM factory allows an object of one type to be overridden with an object of its derived type without changing the testbench structure. This is known as the UVM factory override mechanism. This is applicable for uvm objects and components.

5. Difference between copy and clone.

The copy method makes a copy of the mentioned object. It performs a deep copy.

The clone method is called the create() method followed by copy(). Thus, it creates and returns a copy of an object.

1. While using copy mtheod we will make sure object for destination handle already created

but for clone there is no need to create a destination object

2. clone does two things ---> create+copy

1. first it will create object for destination

2. then it call the copy method

after the copy the object which is created it will return that object reference through the parent handle.

3. The return type of clone is uvm_object

6. Difference between uvm_resource_db and uvm_config_db

1) Although both classes provide a convenience layer on top of the uvm resource facility, uvm_config_db is derived from uvm_resource_db. This adds up additional methods on top of uvm_resource_db methods.
2) With respect to component hierarchy, the uvm_config_db has an argument as a context that has the type of uvm_component and instance name as a string that provides more flexibility as compared to uvm_resource that has two strings scope and name. Since component hierarchy is mentioned in the context, it provides more control over the hierarchical path. The `context = this` argument provides a relative path from the component. The `context = null` provides an absolute path which means there is no hierarchical path above.

Thus, it is recommended to use uvm_config_db always.

7. What is severity and verbosity in UVM?

The severity and verbosity are parameters used for controlling the reporting and logging of messages during the verification process.

Severity: It represents the importance of a message generated during the simulation.

1. UVM_INFO: Provides general information about the progress of the simulation and also we can print variable values for ease in debugging
2. UVM_WARNING: Indicates non-critical errors or potential issues
3. UVM_ERROR: Indicates critical errors that require attention.
4. UVM_FATAL: Indicates fatal errors that lead to the termination of the simulation.

Verbosity: It controls the print messages generated by simulation and it helps to print messages in well well-structured way without flooding the log with all messages.

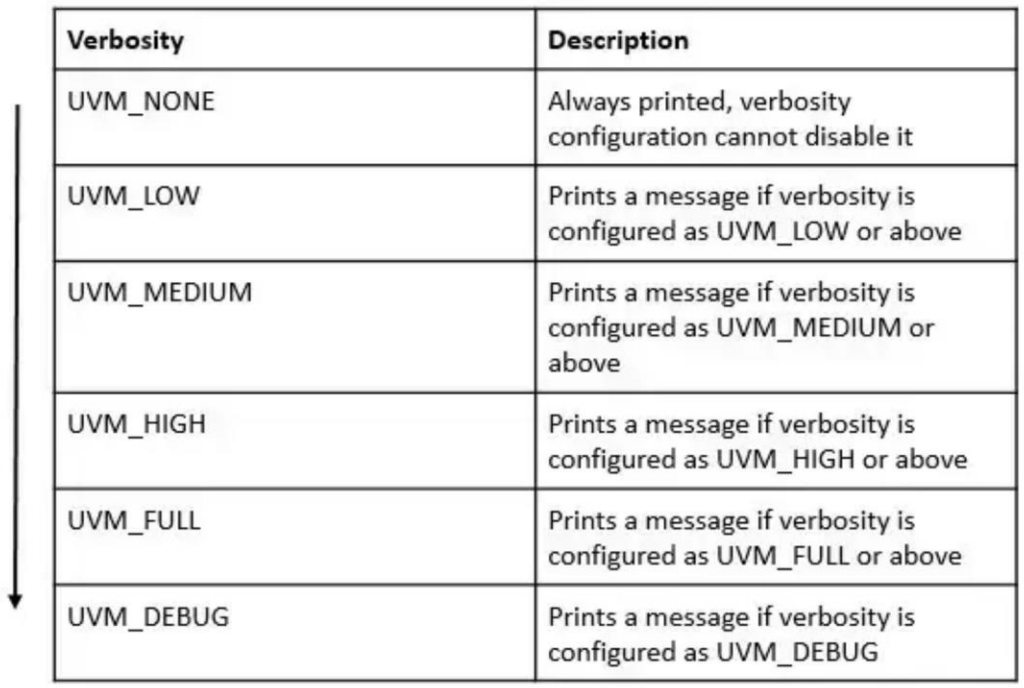
Levels:

UVM_NONE: No messages are displayed.

UVM_LOW: Minimal information is displayed.

UVM_MEDIUM: Moderate level of information, suitable for regular debugging.

UVM_HIGH: High level of information, providing detailed debugging messages.



Verbosity	Description
UVM_NONE	Always printed, verbosity configuration cannot disable it
UVM_LOW	Prints a message if verbosity is configured as UVM_LOW or above
UVM_MEDIUM	Prints a message if verbosity is configured as UVM_MEDIUM or above
UVM_HIGH	Prints a message if verbosity is configured as UVM_HIGH or above
UVM_FULL	Prints a message if verbosity is configured as UVM_FULL or above
UVM_DEBUG	Prints a message if verbosity is configured as UVM_DEBUG

8. Difference between sequence and sequencer.

The uvm_sequence defines a sequence of transactions or stimuli that will be applied to the DUT. Users can also develop complex sequences which consist of multiple sub-sequences.

The uvm_sequencer manages the execution of uvm_sequence that schedules sequences in an order. Uvm_sequencer ensures synchronization and coordination.

9. How to run any test case in UVM?

Execution of the test

The test execution is a time-consuming activity that runs sequence or sequences for DUT functionality. On executing the test, it builds a complete UVM testbench structure in the build_phase and time consuming activities are performed in the run_phase with the help of sequences. It is mandatory to register tests in the UVM factory otherwise, the simulation will terminate with UVM_FATAL (test not found).

run_test() task

The run_test task is a global task declared in the uvm_root class and responsible for running a test case.

Declaration:

```
virtual task run_test (string test_name = "")
```

1. The run_test task starts the phasing mechanism that executes phases in a pre-defined order.

2. It is called in the initial block of the testbench top and accepts test_name as a string. Example:

```
// initial block of tb_top
initial begin
  run_test("my_test");
end
```

3. Passing an argument to the run_test task causes recompilation while executing different tests. To avoid it, UVM provides an alternative way using the +UVM_TESTNAME command-line argument. In this case, the run_test task does not require passing any argument in the initial block of the testbench top.

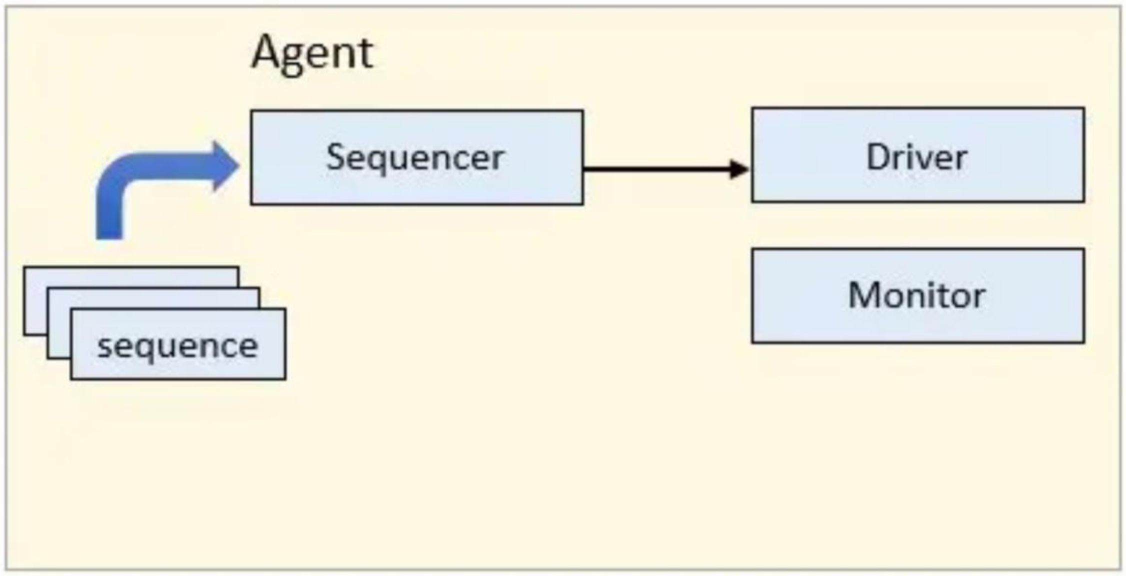
```
// initial block of tb_top
initial begin
  run_test();
end

// Passing command line argument to the simulator
<other options> +UVM_TESTNAME = my_test
```

4. After execution of all phases, run_test finally calls $finish task for simulator exit.

10. Write a simple uvm sequence template and explain each line.

UVM Sequence

UVM sequence is a container that holds data items (uvm_sequence_items) which are sent to the driver via the sequencer.



uvm_sequence class declaration:

```
virtual class uvm_seqence #( type REQ = uvm_sequence_item, type RSP = REQ)
extends uvm_sequence_base
```

uvm_sequence class structure

```
class my_sequence extends uvm_sequence #(my_seq_item);
  `uvm_object_utils(my_sequence)

  function new(string name = "my_sequnce");
    super.new(name);
  endfunction

  task body();
    ...
  endtask
endclass
```

Why `uvm_object_utils is used in sequence, why `uvm_sequence_utils are not used?

`uvm_sequence_utils is a string-based sequence library that is deprecated in UVM.

What is the body() method?

The operation which is intended to do by sequence is defined inside a body method.

Along with a body() method, pre_body, and post_body methods are called by default.

```
class my_sequence extends uvm_sequence #(my_seq_item);
  `uvm_object_utils(my_sequence)

  function new(string name = "my_sequnce");
    super.new(name);
  endfunction

  task pre_body();
    ...
  endtask

  task body();
    ...
  endtask

  task post_body();
    ...
  endtask
endclass
```

Note:

1. These pre_body and post_body tasks are additional (can be named as callbacks) which are useful to perform any operation before and after the execution of the body() method.
2. pre_body() and post_body() methods are optional.

How to write a sequence?

An intention is to create a seq_item, randomize it, and then send it to the driver. To perform this operation any one of the following approaches is followed in the sequence.

1. Using macros like `uvm_do , `uvm_create, `uvm_send etc
2. Using existing methods from the base class
   a. Using wait_for_grant(), send_request(), wait_for_item_done() etc
   b. Using start_item/finish_item methods.

11. How is scoreboard connected to different components?

The UVM scoreboard is a component that checks the functionality of the DUT. It receives transactions from the monitor using the analysis export for checking purposes.

The analysis port is used to connect the uvm monitor to the scoreboard to send the sequence items or transactions for comparison.

Intermediate level questions

1. What are the UVM factory and its use?

The UVM factory is used to create UVM objects and components. This is commonly known as factory registration. The factory registration for UVM objects and components are lightweight proxies of the actual objects and components.

In UVM based testbench, it is valid to use a new() function to create class objects, but factory registration has its benefits. The UVM factory allows an object of one type to be overridden with an object of its derived type without changing the testbench structure. This is known as the UVM factory override mechanism. This is applicable for uvm objects and components.

2. Why phases are introduced in UVM? What are all phases present in UVM?

All components are static in the Verilog-based testbench whereas System Verilog introduced the OOP (Object Oriented Programming) feature in the testbench. In Verilog, as modules are static, users don't have to care about their creation as they would have already created at the beginning of the simulation. In the case of UVM based System Verilog testbench, class objects can be created at any time during the simulation based on the requirement. Hence, it is required to have proper synchronization to avoid objects/components being called before they are created, The UVM phasing mechanism serves the purpose of synchronization.

We have the following phases present in UVM:

1. build_phase
2. connect_phase
3. end_of_elaboration_phase
4. start_of_simulation_phase
5. run_phase
6. pre_reset
7. reset
8. post_reset
9. pre_configure
10. configure

11. post_configure
12. pre_main
13. main
14. post_main
15. pre_shutdown
16. shutdown
17. post_shutdown
18. extract
19. check
20. report
21. final

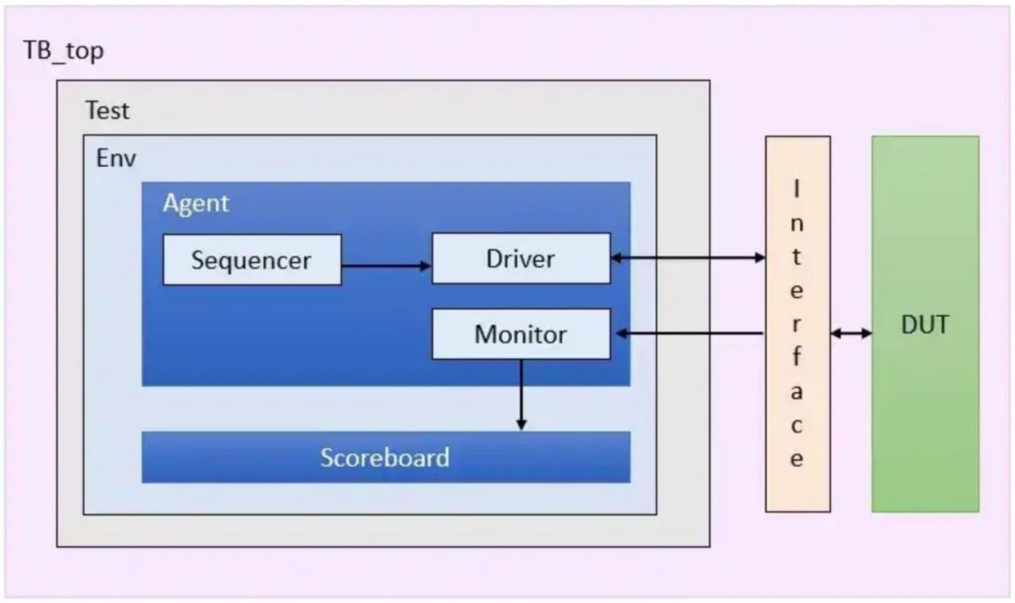
3. What approach does build_phase use and why?

The build_phase uses a top-down approach because build_phase creates the parent component first so that the parent component can be configured and then recursively calls their child components as per required customization. This ensures proper instantiation and hierarchical connectivity.

4. Draw and explain the verification environment in UVM.

UVM testbench Top

The testbench top is a static container that has an instantiation of DUT and interfaces. The interface instance connects with DUT signals in the testbench top. The clock is generated and initially reset is applied to the DUT. It is also passed to the interface handle. An interface is stored in the uvm_config_db using the set method and it can be retrieved down the hierarchy using the get method. UVM testbench top is also used to trigger a test using run_test() call.



Example of UVM Testbench Top **for** Adder **design**

```
`include "uvm_macros.svh"
import uvm_pkg::*;

module tb_top;
  bit clk;
  bit reset;
  always #5 clk = ~clk;

  initial begin
    clk = 0;
    reset = 1;
    #5;
    reset = 0;
  end
  add_if vif(clk, reset);

  // Instantiate design top
  adder DUT(.clk(vif.clk),
            .reset(vif.reset),
            .in1(vif.ip1),
            .in2(vif.ip2),
            .out(vif.out)
           );

  initial begin
    // set interface in config_db
    uvm_config_db#(virtual add_if)::set(uvm_root::get(), "*", "vif", vif);
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars;
  end
  initial begin
```

```
    run_test("base_test");
  end
endmodule
```

UVM testbench	**Description**
UVM Test	The test is at the top of the hierarchy that initiates the environment component construction. It is also responsible for the testbench configuration and stimulus generation process.
UVM Environment	An environment provides a well-mannered hierarchy and container for agents, scoreboards.
UVM Agent	An agent is a container that holds the driver, monitor, and sequencer. This is helpful to have a structured hierarchy based on the protocol or interface requirement
UVM Sequence Item	The transaction is a packet that is driven to the DUT or monitored by the monitor as a pin-level activity.
UVM Driver	The driver interacts with DUT. It receives randomized transactions or sequence items and drives them to the DUT as a pin-level activity.
UVM Sequence	The sequence is a container that holds data items (uvm_sequence_items) that are sent to or received from the driver via the sequencer.
UVM Sequencer	The sequencer is a mediator who establishes a connection between the sequence and the driver. Ultimately, it passes transactions or sequence items to the driver so that they can be driven to the DUT.
UVM Monitor	The monitor observes pin-level activity on the connected interface at the input and output of the design. This pin-level activity is converted into a transaction packet and sends to the scoreboard for checking purposes.
UVM Scoreboard	The scoreboard receives the transaction packet from the monitor and compares it with the reference model. The reference module is written based on design specification understanding and design behavior.

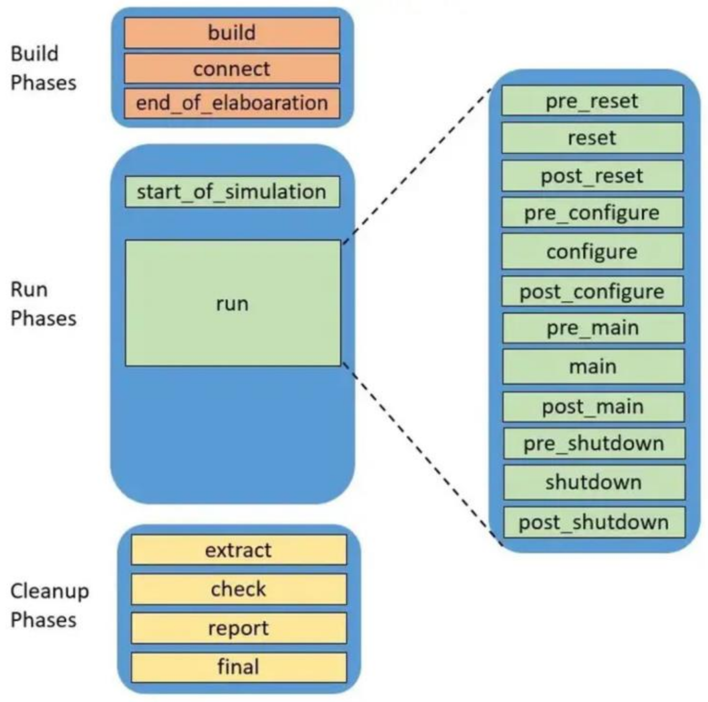
5. What all UVM phases take time?

All run phases (including its sub-phases) take time.

1. run_phase
2. pre_reset
3. reset
4. post_reset
5. pre_configure
6. configure
7. post_configure
8. pre_main
9. main
10. post_main
11. pre_shutdown
12. shutdown
13. post_shutdown

6. What all phases are functions and tasks in UVM?

Build phases



To construct a testbench, it is necessary to build component objects first and then are connected to form a hierarchy, The build phase category consists

a. build_phase

b. connect_phase

c. end_of_elaboration_phase

Phase Type	Phase Name	Description	Execution approach
function	build_phase	Build or create testbench component	Top to down
function	connect_phase	Connect different testbench component using the TLM port mechanism	Bottom to top
function	end_of_elaboration_phase	Before simulation starts, this phase is used to make any final adjustment to the structure, configuration, or connectivity of the testbench. It also displays UVM topology.	Bottom to top

Run-time phases

Once testbench components are created and connected in the testbench, it follows run phases where actual simulation time consumes. After completing, start_of_simulation phase, there are two paths for run-time phases. The run_phase and pre_reset phase both start at the same time.

a. run_phase

Phase Type	Phase Name	Description	Execution approach
function	start_of_simulation_phase	Used to display testbench topology or configuration.	Bottom to top

a. run_phase

Phase Type	Phase Name	Description
task	run_phase	Used for the stimulus generation, checking activities of the testbench, and consumes simulation time cycles. The run_phase for all components are executed in parallel.

Clean up phases

The clean-up phases are used to collect information from functional coverage monitors and scoreboards to see whether the coverage goal has been reached or the test case has passed. The cleanup phases will start once the run phases are completed. They are implemented as functions and work from the bottom to the top of the component hierarchy. The extract, check, and report phase may be used by analysis components.

Phase Type	Phase Name	Description	Execution approach
function	extract	Used to retrieve and process the information from functional coverage monitors and scoreboards. This phase may also calculate any statistical information that will be used by report_phase.	Bottom to top
function	check	Checks DUT behavior and identity for any error that occurred during the execution of the testbench.	Bottom to top
function	report	Used to display simulation results. It can also write results to the file.	Bottom to top
function	final	Used to complete any outstanding actions that are yet to be completed in the testbench.	Top to down

7. Why do we use the super keyword in phases likes super.build_phase, super.main_phase etc?

The super keyword is used to refer to class members (like build_phase, main_phase, run_phase, etc) of its immediate base class or uvm_component (if derived class is extended directly from uvm_component).

8. Difference between `uvm_do and `uvm_rand_send.

Both are uvm sequence macros that are used to generate the sequence or seq_item.

`uvm_do (seq/item) – On calling this macro, create, randomize and send to the driver will be executed
Example:

```
class my_sequence extends uvm_sequence #(seq_item);
  `uvm_object_utils(my_sequence)

   function new (string name = "my_sequence")
     super.new(name);
   endfunction

   task body();
     `uvm_do(req);
   endtask
endclass

class my_sequence extends uvm_sequence #(seq_item);
  `uvm_object_utils(my_sequence)

   function new (string name = "my_sequence")
     super.new(name);
   endfunction

   task body();
     `uvm_do(seq1); // calling seq1
     `uvm_do(seq2); // calling seq2
   endtask
endclass
```

`uvm_rand_send(seq/item) – It directly sends a randomized seq/item without creating it. So, make sure the seq/item is created first.
Example:

```
class my_sequence extends uvm_sequence #(seq_item);
  `uvm_object_utils(my_sequence)

   function new (string name = "my_sequence")
```

```
    super.new(name);
  endfunction

  task body();
    `uvm_create(req);
    `uvm_rand_send(req);
  endtask
endclass
```

9. What are bottom-up and top-down approaches in UVM phases?

These approaches are associated with the execution of various phases of UVM based verification environment.

1. Bottom-Up Approach: Lower-level components being built and configured before higher-level components.
   Example: connect_phase, start_of_simulation_phase, etc
2. Top-Down Approach: On the contrary, higher-level components are built and configured before lower-level components.
   Example: build_phase, final_phase

10. How do uvm_config_db set/get work?

uvm_config_db in UVM

The uvm_config_db class is derived from the uvm_resource_db class. It is another layer of convenience layer on the top of uvm_resource_db that simplifies the basic interface (access methods of resource database) used for uvm_component instances.

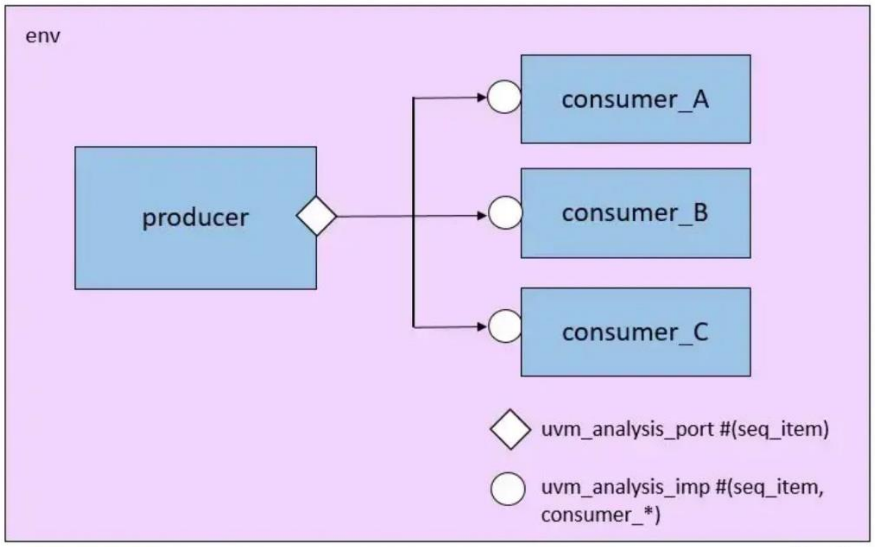
The below code snippet for the uvm_config_db class is taken from the uvm source code.

```
class uvm_config_db#(type T=int) extends uvm_resource_db#(T);

  static uvm_pool#(string,uvm_resource#(T)) m_rsc[uvm_component];
  static local uvm_queue#(m_uvm_waiter) m_waiters[string];

  static function bit get(uvm_component cntxt,
                          string inst_name,
                          string field_name,
                          inout T value);
    ...
  endfunction

  static function void set(uvm_component cntxt,
                           string inst_name,
                           string field_name,
                           T value);
    ...
  endfunction

...
...
endclass



Syntax for set method

void uvm_config_db#(type T = int)::set(uvm_component cntxt,
                                        string inst_name,
                                        string field_name,
                                        T value);
Syntax for get method

bit uvm_config_db#(type T=int)::get(uvm_component cntxt,
                                    string inst_name,
                                    string field_name,
                                    ref T value);
```

11. What is an analysis port?

The uvm_analysis_port is a TLM-based class that provides a write method for communication. TLM analysis port broadcasts transactions to one or multiple components.

Example:


env
producer
consumer_A
consumer_B
consumer_C
uvm_analysis_port #(seq_item)
uvm_analysis_imp #(seq_item, consumer_*)

TLM analysis port

seq_item.sv

```
class seq_item extends uvm_sequence_item;
  rand bit [3:0] value;
  `uvm_object_utils(seq_item)

  function new(string name = "seq_item");
    super.new(name);
  endfunction

  constraint val_c {value > 0;}
endclass
`include "uvm_macros.svh"
import uvm_pkg::*;

`include "seq_item.sv"

class producer extends uvm_component;
  seq_item req;
  uvm_analysis_port #(seq_item) a_put;

  `uvm_component_utils(producer)

  function new(string name = "producer", uvm_component parent = null);
    super.new(name, parent);
    a_put = new("a_put", this);
  endfunction

  task run_phase(uvm_phase phase);
    super.run_phase(phase);

    req = seq_item::type_id::create("req");
    assert(req.randomize());
    `uvm_info(get_type_name(), $sformatf("Send value = %0h", req.value), 
UVM_NONE);
    a_put.write(req);
  endtask
endclass

class consumer_A extends uvm_component;
  seq_item req;
  uvm_analysis_imp #(seq_item, consumer_A) analysis_imp_A;

  `uvm_component_utils(consumer_A)

  function new(string name = "consumer_A", uvm_component parent = null);
    super.new(name, parent);
    analysis_imp_A = new("analysis_imp_A", this);
  endfunction

  virtual function void write(seq_item req);
    `uvm_info(get_type_name(), $sformatf("Received value = %0h", req.value), 
UVM_NONE);
  endfunction
endclass
```

```
class consumer_B extends uvm_component;
  seq_item req;
  uvm_analysis_imp #(seq_item, consumer_B) analysis_imp_B;

  `uvm_component_utils(consumer_B)
  function new(string name = "consumer_B", uvm_component parent = null);
    super.new(name, parent);
    analysis_imp_B = new("analysis_imp_B", this);
  endfunction

  virtual function void write(seq_item req);
    `uvm_info(get_type_name(), $sformatf("Received value = %0h", req.value),
UVM_NONE);
  endfunction
endclass

class consumer_C extends uvm_subscriber #(seq_item);
  seq_item req;
  `uvm_component_utils(consumer_C)

  function new(string name = "consumer_C", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  virtual function void write (seq_item t);
    req = t;
    `uvm_info(get_type_name(), $sformatf("Received value = %0h", req.value),
UVM_NONE);
  endfunction
endclass

class env extends uvm_env;
  producer pro;
  consumer_A con_A;
  consumer_B con_B;
  consumer_C con_C;
  `uvm_component_utils(env)

  function new(string name = "env", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    pro = producer::type_id::create("pro", this);
    con_A = consumer_A::type_id::create("con_A", this);
    con_B = consumer_B::type_id::create("con_B", this);
    con_C = consumer_C::type_id::create("con_C", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    pro.a_put.connect(con_A.analysis_imp_A);
    pro.a_put.connect(con_B.analysis_imp_B);
    pro.a_put.connect(con_C.analysis_export);
  endfunction
endclass
```

```
class test extends uvm_test;
  env env_o;
  `uvm_component_utils(test)

  function new(string name = "test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env_o = env::type_id::create("env_o", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    #50;
    phase.drop_objection(this);
  endtask
endclass

module tb_top;
  initial begin
    run_test("test");
  end
endmodule
```

```
Output:

UVM_INFO testbench.sv(22) @ 0: uvm_test_top.env_o.pro [producer] Send value =
6
UVM_INFO testbench.sv(39) @ 0: uvm_test_top.env_o.con_A [consumer_A] Received
value = 6
UVM_INFO testbench.sv(55) @ 0: uvm_test_top.env_o.con_B [consumer_B] Received
value = 6
UVM_INFO testbench.sv(69) @ 0: uvm_test_top.env_o.con_C [consumer_C] Received
value = 6
```

12. What are pre_body and post_body used in sequence?

Both are user-defined callback tasks and will be called when the sequence is started.

pre_body: It is a user-definable callback that is called before the execution of body only when the sequence is started with a ‘start’ task.

post_body: It is a user-definable callback task that is called after the execution of the body only when the sequence is started with the start method.

13. What is an active and passive agent?

Active Agent: An Active agent drives stimulus to the DUT. It instantiates all three components driver, monitor, and sequencer.

Passive Agent: A passive agent does not drive stimulus to the DUT. It instantiates only a monitor component. It is used as a sample interface for coverage and checker purposes.

14. Difference between UVM subscriber and UVM scoreboard

The UVM scoreboard is a component that checks the functionality of the DUT. It receives transactions from the monitor using the analysis export for checking purposes whereas the uvm_subscriber class provides a built-in analysis export that connects with the analysis port. As the name suggests, it subscribes to the broadcaster i.e. analysis port to receive broadcasted transactions. It is used to implement a functional coverage monitor.

15. What is uvm objection and why do we need it?

The uvm_objection class provides a mechanism to coordinate status information between two or more components, objects. The uvm_objection class is extended from uvm_report_object.

Class Declaration

```
class uvm_objection extends uvm_report_object
```

The objection deals with the concept of raise and drop objection which means the internal counter is increment and decrement respectively. Each participating component and object may raise or drop objections asynchronously. When all objections are dropped, the counter value will become zero. The objection has to be raised before starting any process and drop it once it is completed.

UVM Objection Usage

1) UVM phasing mechanism uses objections to coordinate with each other and the phase should be ended when all objections are dropped. They can be used in all UVM phases.

2) It allows proceeding for the “End of test”. The simulation time-consuming activity happens in the run phase. If all objections dropped for run phases, it means simulation activity is completed. The test can be ended after executing clean up phases.

```
task reset_phase( uvm_phase phase);
  phase.raise_objection(this);
  ...
  phase.drop_objection(this);
endtask

task run_phase(uvm_phase phase);
  phase.raise_objection(this, "Raised Objection");
  ...
  phase.drop_objection(this, "Dropped Objection");
endtask
```

Note:

1. Objections are generally used in components and sequences.
2. Other objects can also use them but they must use a component or sequence object context.

Methods in uvm_objection

Methods	**Description**
raise_objection (uvm_object obj = null, string description = ” “, int count = 1)	Raises number of objections for corresponding object with default count = 1
drop_objection (uvm_object obj = null, string description = ” “, int count = 1)	Drops number of objections for corresponding object with default count = 1
set_drain_time (uvm_object obj=null, time drain)	Sets drain time for corresponding objects.

16. What are the different ways of exiting the code execution?

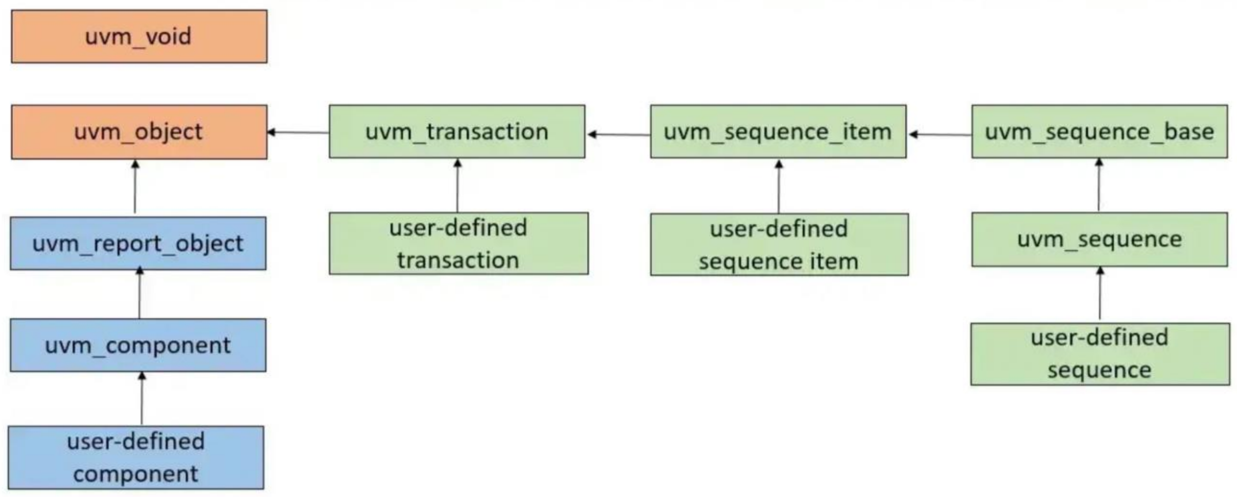
1) **Using UVM Objections (Recommended approach):** It allows proceeding for the "End of test". The simulation time-consuming activity happens in the run phase. If all objections are dropped for run phases, it means the simulation activity is completed. The test can be ended after executing clean-up phases.
2) **$finish:** It tells the simulator to exit the simulation. However, it is not a graceful simulation for a UVM-based environment.
3) **`uvm_fatal:** Based on certain conditions as per requirement, it can be used to terminate the simulation.
   Example: If class config class is being used during run_phase and if this config class is not available in config_db, then uvm_fatal is used to terminate simulation right away.

17. What is a sequence item in UVM?

Sequence Items in UVM

All user-defined sequence items are extended from the uvm_sequence_item class as it leverages generating stimulus and has control capabilities for the sequence-sequencer mechanism.

uvm_sequence_item class hierarchy



18. Explain how the sequence starts.

A sequence is started by calling the start method that accepts a pointer to the sequencer through which sequence_items are sent to the driver. A pointer to the sequencer is also commonly known as m_sequencer. The start method assigns a sequencer pointer to the m_sequencer and then calls the body() task. On completing the body task with the interaction with the driver, the start() method returns. As it requires interaction with the driver, the start is a blocking method.

19. How do sequence, driver, and sequencer communicate?

Refer sequence-driver-sequencer-communication-in-uvm

20. What are lock and grab methods?

Lock and Grab Methods in UVM sequencer

The UVM sequencer provides the facility to have exclusive access for the sequence to a driver via a sequencer using a locking mechanism. This locking mechanism is implemented using lock and grab methods.

Example: In controller or microprocessor, internal core services interrupt handling along with other operations. Sometimes, if a particular interrupt is raised by the device which needs immediate attention and stops ongoing process execution. Once this high-priority interrupt is serviced by the core, the previous process can be resumed.

Lock method

On calling the lock method from a sequence, the sequencer will grant the sequence exclusive access to the driver when the sequence gets the next available slot via a sequencer arbitration mechanism. Until the sequence calls the unlock method, no other sequence will have access to the driver. On calling unlock method, the lock will be released. The lock is a blocking method and returns once the lock has been granted.

```
task lock (uvm_sequencer_base sequencer = null)
```

grab method

The grab method is similar to the lock method except it takes immediate control to grab the next sequencer arbitration slot itself by overriding any other sequence priorities.

The pre-existing lock or grab condition on the sequence will stop from grabbing the sequence for the current sequence.

```
task grab (uvm_sequencer_base sequencer = null)
```

unlock method

The unlock method of the sequencer is called from the sequence to release its lock or grab. It is mandatory to call unlock method in order to avoid sequencer locked conditions.

```
function void unlock (uvm_sequencer_base sequencer = null)
```

ungrab method

The ungrab method is an alias of the unlock method

```
function void ungrab( uvm_sequencer_base sequencer = null )
```

```
Lock and unlock methods example
There are three sequences seq_A, seq_B, and seq_C are forked at the same
time. The only seq_B calls for lock and unlock methods.

class seq_A extends uvm_sequence #(seq_item);
  seq_item req;
  `uvm_object_utils(seq_A)

  function new (string name = "seq_A");
    super.new(name);
  endfunction

  task body();
    req = seq_item::type_id::create("req");
    wait_for_grant();
    assert(req.randomize());
    send_request(req);
    `uvm_info(get_type_name(), $sformatf("%0t: seq_item is sent", $time),
UVM_LOW);
    wait_for_item_done();
  endtask
endclass

class seq_B extends uvm_sequence #(seq_item);
  seq_item req;
  `uvm_object_utils(seq_B)

  function new (string name = "seq_B");
```

```
    super.new(name);
  endfunction

  task body();
      lock(m_sequencer);
      req = seq_item::type_id::create("req");
      wait_for_grant();
      assert(req.randomize());
      send_request(req);
      `uvm_info(get_type_name(), $sformatf("%0t: seq_item is sent", $time),
UVM_LOW);
      wait_for_item_done();
      unlock(m_sequencer);
  endtask
endclass

class seq_C extends uvm_sequence #(seq_item);
  seq_item req;
  `uvm_object_utils(seq_C)

  function new (string name = "seq_C");
    super.new(name);
  endfunction

  task body();
    req = seq_item::type_id::create("req");
    wait_for_grant();
    assert(req.randomize());
    send_request(req);
    `uvm_info(get_type_name(), $sformatf("%0t: seq_item is sent", $time),
UVM_LOW);
    wait_for_item_done();
  endtask
endclass

Output:

UVM_INFO sequences.sv(14) @ 0: uvm_test_top.env_o.agt.seqr@@seq_a [seq_A] 0:
seq_item is sent
UVM_INFO sequences.sv(33) @ 50: uvm_test_top.env_o.agt.seqr@@seq_b [seq_B]
50: seq_item is sent
UVM_INFO sequences.sv(52) @ 100: uvm_test_top.env_o.agt.seqr@@seq_c [seq_C]
100: seq_item is sent
UVM_INFO sequences.sv(14) @ 150: uvm_test_top.env_o.agt.seqr@@seq_a [seq_A]
150: seq_item is sent
UVM_INFO sequences.sv(33) @ 200: uvm_test_top.env_o.agt.seqr@@seq_b [seq_B]
200: seq_item is sent
UVM_INFO sequences.sv(52) @ 250: uvm_test_top.env_o.agt.seqr@@seq_c [seq_C]
250: seq_item is sent
UVM_INFO sequences.sv(14) @ 300: uvm_test_top.env_o.agt.seqr@@seq_a [seq_A]
300: seq_item is sent
UVM_INFO sequences.sv(33) @ 350: uvm_test_top.env_o.agt.seqr@@seq_b [seq_B]
350: seq_item is sent
UVM_INFO sequences.sv(52) @ 400: uvm_test_top.env_o.agt.seqr@@seq_c [seq_C]
400: seq_item is sent
```

Grab and ungrab methods example
There are three sequences seq_A, seq_B, and seq_C are forked at the same time. The only seq_B calls for grab and ungrab methods.

```
class seq_A extends uvm_sequence #(seq_item);
  seq_item req;
  `uvm_object_utils(seq_A)

  function new (string name = "seq_A");
    super.new(name);
  endfunction

  task body();
    req = seq_item::type_id::create("req");
    wait_for_grant();
    assert(req.randomize());
    send_request(req);
    `uvm_info(get_type_name(), $sformatf("%0t: seq_item is sent", $time),
UVM_LOW);
    wait_for_item_done();
  endtask
endclass

class seq_B extends uvm_sequence #(seq_item);
  seq_item req;
  `uvm_object_utils(seq_B)

  function new (string name = "seq_B");
    super.new(name);
  endfunction

  task body();
      grab(m_sequencer);
      req = seq_item::type_id::create("req");
      wait_for_grant();
      assert(req.randomize());
      send_request(req);
      `uvm_info(get_type_name(), $sformatf("%0t: seq_item is sent", $time),
UVM_LOW);
      wait_for_item_done();
      ungrab(m_sequencer);
  endtask
endclass

class seq_C extends uvm_sequence #(seq_item);
  seq_item req;
  `uvm_object_utils(seq_C)

  function new (string name = "seq_C");
    super.new(name);
  endfunction

  task body();
    req = seq_item::type_id::create("req");
    wait_for_grant();
    assert(req.randomize());
    send_request(req);
    `uvm_info(get_type_name(), $sformatf("%0t: seq_item is sent", $time),
UVM_LOW);
    wait_for_item_done();
  endtask
endclass
Output:

UVM_INFO sequences.sv(33) @ 0: uvm_test_top.env_o.agt.seqr@@seq_b [seq_B] 0:
seq_item is sent
UVM_INFO sequences.sv(33) @ 50: uvm_test_top.env_o.agt.seqr@@seq_b [seq_B]
50: seq_item is sent
UVM_INFO sequences.sv(33) @ 100: uvm_test_top.env_o.agt.seqr@@seq_b [seq_B]
100: seq_item is sent
UVM_INFO sequences.sv(14) @ 150: uvm_test_top.env_o.agt.seqr@@seq_a [seq_A]
150: seq_item is sent
UVM_INFO sequences.sv(52) @ 200: uvm_test_top.env_o.agt.seqr@@seq_c [seq_C]
200: seq_item is sent
UVM_INFO sequences.sv(14) @ 250: uvm_test_top.env_o.agt.seqr@@seq_a [seq_A]
250: seq_item is sent
UVM_INFO sequences.sv(52) @ 300: uvm_test_top.env_o.agt.seqr@@seq_c [seq_C]
300: seq_item is sent
UVM_INFO sequences.sv(14) @ 350: uvm_test_top.env_o.agt.seqr@@seq_a [seq_A]
350: seq_item is sent
UVM_INFO sequences.sv(52) @ 400: uvm_test_top.env_o.agt.seqr@@seq_c [seq_C]
400: seq_item is sent
```

21. What are the RAL model and its application?

"The RAL model is an abstract model for registers and memories in DUT."

The register abstraction layer (RAL) provides standard base class libraries. It is used to create a memory-mapped model for registers in DUT using an object-oriented model.

The UVM RAL is used to model DUT registers and memories as a set of classes. It generates stimulus to the DUT and covers some aspects of functional coverage.

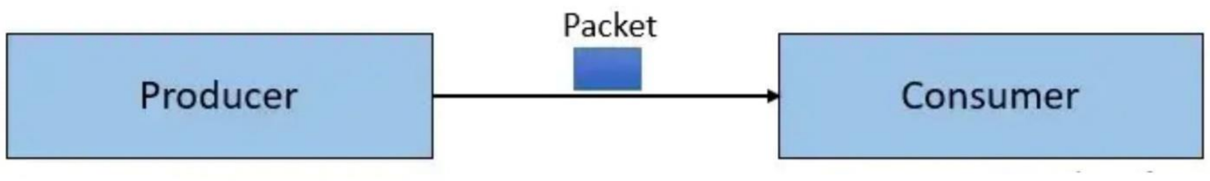
22. What do you mean by the front-door and back-door register access?

Front door Access: A register is said to be accessed as a front door if it involves a bus interface. It does consume time to access the register.

Back door Access: A register is said to be accessed as a back door if it uses a simulator database to directly access the DUT register using design signals.

23. What is TLM?

TLM establishes a connection between producer and consumer components through which transactions are sent.



TLM provides

1. unidirectional, bidirectional, or broadcasting manner communication between components
2. Broadcasting of information promptly.
3. One component to multiple components connection.
4. The mechanism uses task and function methods to establish a connection.
5. FIFO can hold transactions and get them based on the requirement.
6. A higher level of abstraction.
7. Connection to systemC.

24. What is TLM FIFO?

```
1. Both are initiator here and Generator sending the data so it should have
put_port
        2. Driver recieving the data so it should have get_port
        3. TLM_FIFO have put_export & get_export
        4. Methods like get(), put(), peek() etc are implemented inside
TLM_FIFO

        Producer puts the transaction into uvm_tlm_fifo, while consumer gets
the transaction from fifo.

        uvm_tlm_fifo#(write_xtn) fifoh;



//APB GENERATOR:(Initiator)
----------------------------

    class apb_gen extends uvm_component;
        uvm_blocking_put_port#(write_xtn) put_port;

        function new(string name,uvm_component parent);
            super.new(name,parent);
            put_port = new("put_port",this);
        endfunction
        virtual task run_phase(uvm_phase phase);
            write_xtn t;
                for(int i=0;i<N;i++);
                    begin
                       put_port.put(t);
                    end
        endtask
    endclass

//APB DRIVER:(Initiator)
-------------------------

    class apb_driver extends uvm_component;
        uvm_blocking_get_port#(write_xtn) get_port;

        function new(string name,uvm_component parent);
            super.new(name,parent);
            get_port = new("get_port",this);
        endfunction

        virtual task run_phase(uvm_phase phase);
            write_xtn t;
                for(int i=0;i<N;i++);
                    begin
                       //generate t
                       get_port.get(t);     //It invoke get method inside TLM
FIFO
                    end
        endtask
    endclass

//APB_AGENT:(CONNECTION)
--------------------------
class apb_agent extends uvm_component;

    apb_gen apb_genh;
    apb_driver apb_drvh;

    uvm_tlm_fifo#(write_xtn) fifoh;

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

        fifoh = new("fifoh",this);

    endfunction

    function void connect_phase(uvm_phase phase);

        apb_genh.put_port.connect(fifoh.put_export);
        apb_drvh.get_port.connect(fifoh.get_export);

    endfunction
endclass
INCASE WE DON'T WANT TO USE FIFO THEN:
```

```
//TARGET CLASS
---------------

class target extends uvm_component;

    uvm_blocking_get_imp#(write_xtn,target) get_imp;
    uvm_blocking_put_imp#(write_xtn,target) put_imp;

    virtual task put(input write_xtn t);

        case(t.kind)
            READ:    //Do read
            WRITE:   //Do write
        endcase

    endtask

    virtual task get(output write_xtn t);
        write_xtn tmp = new();
        //assign value to temp
        tmp.addr = 15;
        tmp.data = 16;

        t = tmp;
    endtask

endclass

**********WARNING*******
//These get and put method in target should implemented in such a way data
should be first in first out
//We declare handle of target class in a class in which driver and initiator
enclosed and do connection
//we do not need to write those methods It is already defined inside
uvm_tlm_fifo class
```

uvm_tlm_fifo Methods

Methods	**Description**
new	To create TLM FIFO. function new (string name, uvm_component parent=null, int size=1); Size – Represents the size of TLM FIFO. If a size is 0 then TLM FIFO is not bounded.
size	Returns the size of TLM FIFO.

	If a size is 0 then TLM FIFO is not bounded.
used	Return number of entries put into TLM FIFO.
is_empty	No entries in TLM FIFO returns 1 else returns 0
is_full	on TLM FIFO full, returns 1 else returns 0. **Full condition:** FIFO size == number of entries in FIFO.
flush	On calling, it removes all entries from TLM FIFO Thus, on calling used method, it returns 0 (<tlm_fifo>.used returns 0) On calling is_empty method, it returns 1 (<tlm_fifo>,is_empty return 1)

25. What is run_test?

Execution of the test

The test execution is a time-consuming activity that runs sequence or sequences for DUT functionality. On executing the test, it builds a complete UVM testbench structure in the build_phase and time consuming activities are performed in the run_phase with the help of sequences. It is mandatory to register tests in the UVM factory otherwise, the simulation will terminate with UVM_FATAL (test not found).

run_test() task

The run_test task is a global task declared in the uvm_root class and responsible for running a test case.

Declaration:

```
virtual task run_test (string test_name = "")
```

1. The run_test task starts the phasing mechanism that executes phases in a pre-defined order.

2. It is called in the initial block of the testbench top and accepts test_name as a string. Example:

```
// initial block of tb_top
initial begin
  run_test("my_test");
end
```

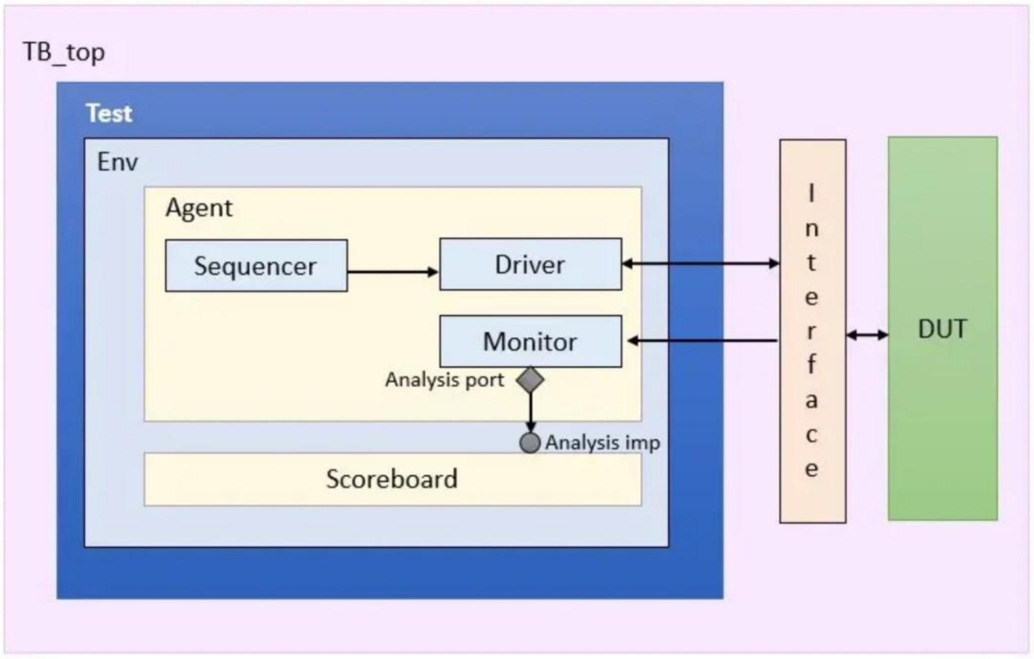
3. Passing an argument to the run_test task causes recompilation while executing different tests. To avoid it, UVM provides an alternative way using the +UVM_TESTNAME command-line argument. In this case, the run_test task does not require passing any argument in the initial block of the testbench top.

```
// initial block of tb_top
initial begin
  run_test();
end

// Passing command line argument to the simulator
<other options> +UVM_TESTNAME = my_test
```

4. After execution of all phases, run_test finally calls $finish task for simulator exit.

26. Explain generalized code structure for UVM monitor and scoreboard.

How to create a UVM monitor:



1) Create a user-defined monitor class extended from uvm_monitor and register it in the factory.
2) Declare virtual interface handle to retrieve actual interface handle using configuration database in the build_phase.
3) Declare analysis port to broadcast the sequence items or transactions.

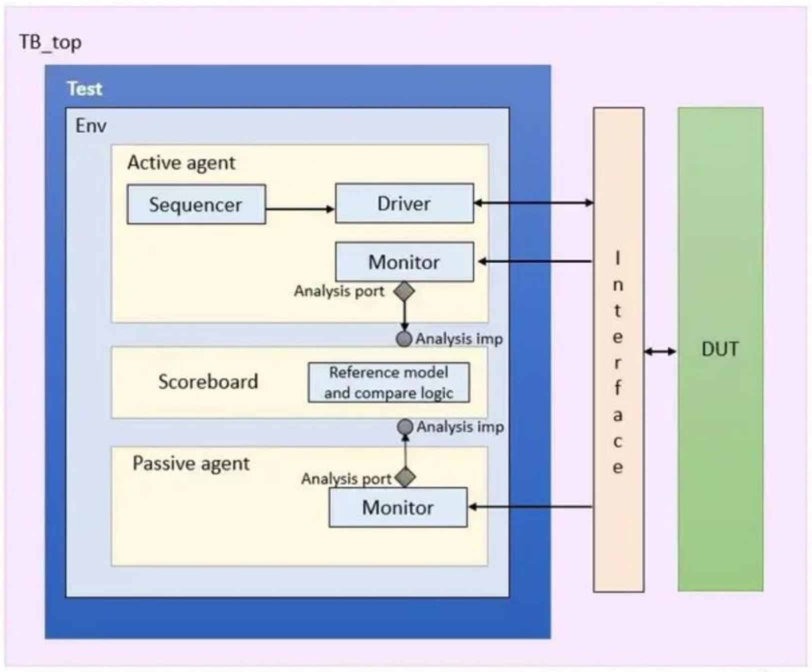
4) Write standard new() function. Since the monitor is a uvm_component. The new() function has two arguments as string name and uvm_component parent.
5) Implement build_phase and get interface handle from the configuration database.
6) Implement run_phase to sample DUT interface using a virtual interface handle and translate into transactions. The write() method sends transactions to the collector component.

UVM Monitor Example

```
class monitor extends uvm_monitor;
  // declaration for the virtual interface, analysis port, and monitor
sequence item.
  virtual add_if vif;
  uvm_analysis_port #(seq_item) item_collect_port;
  seq_item mon_item;
  `uvm_component_utils(monitor)

  // constructor
  function new(string name = "monitor", uvm_component parent = null);
    super.new(name, parent);
    item_collect_port = new("item_collect_port", this);
    mon_item = new();
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual add_if) :: get(this, "", "vif", vif))
      `uvm_fatal(get_type_name(), "Not set at top level");
  endfunction

  task run_phase (uvm_phase phase);
    forever begin
      // Sample DUT information and translate into transaction
      item_collect_port.write(mon_item);
    end
  endtask
endclass
```

Scoreboard Usage

1) Receive transactions from monitor using analysis export for checking purposes.
2) The scoreboard has a reference model to compare with design behavior. The reference model is also known as a predictor that implements design behavior so that the scoreboard can compare DUT outcome with reference model outcome for the same driven stimulus.



How to write scoreboard code in UVM?

1. Create a user-defined scoreboard class extended from uvm_scoreboard and register it in the factory.
2. Declare an analysis export to receive the sequence items or transactions from the monitor.
3. Write standard new() function. Since the scoreboard is a uvm_component. The new() function has two arguments as string name and uvm_component parent.
4. Implement build_phase and create a TLM analysis export instance.
5. Implement a write method to receive the transactions from the monitor.
6. Implement run_phase to check DUT functionality throughout simulation time.

Scoreboard Example

```
class scoreboard extends uvm_scoreboard;
  uvm_analysis_imp #(seq_item, scoreboard) item_collect_export;
  seq_item item_q[$];
  `uvm_component_utils(scoreboard)

  function new(string name = "scoreboard", uvm_component parent = null);
    super.new(name, parent);
    item_collect_export = new("item_collect_export", this);
  endfunction
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
  endfunction

  function void write(seq_item req);
    `uvm_info(get_type_name, $sformatf("Received transaction = %s", req),
UVM_LOW);
    item_q.push_back(req);
  endfunction

  task run_phase (uvm_phase phase);
    seq_item sb_item;
    forever begin
      wait(item_q.size > 0);

      if(item_q.size > 0) begin
        sb_item = item_q.pop_front();
        // Checking comparing logic
        ...
      end
    end
  endtask
endclass
```

27. What is an in-order and out-of-order scoreboard?

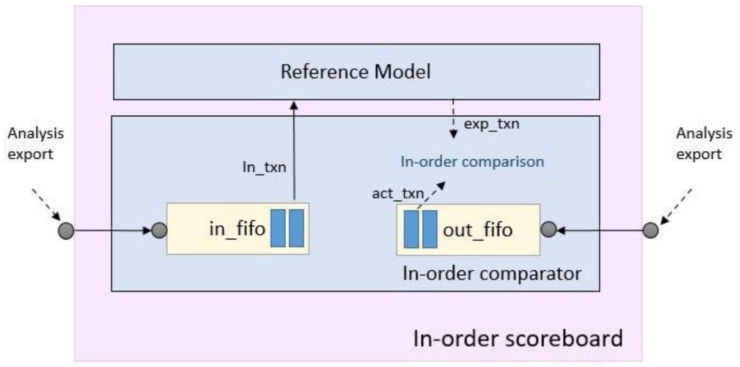
UVM Scoreboad types

Depends on design functionality scoreboards can be implemented in two ways.

1. In-order scoreboard
2. Out-of-order scoreboard

In-order scoreboard

The in-order scoreboard is useful for the design whose output order is the same as driven stimuli. The comparator will compare the expected and actual output streams in the same order. They will arrive independently. Hence, the evaluation must block until both expected and actual transactions are present.

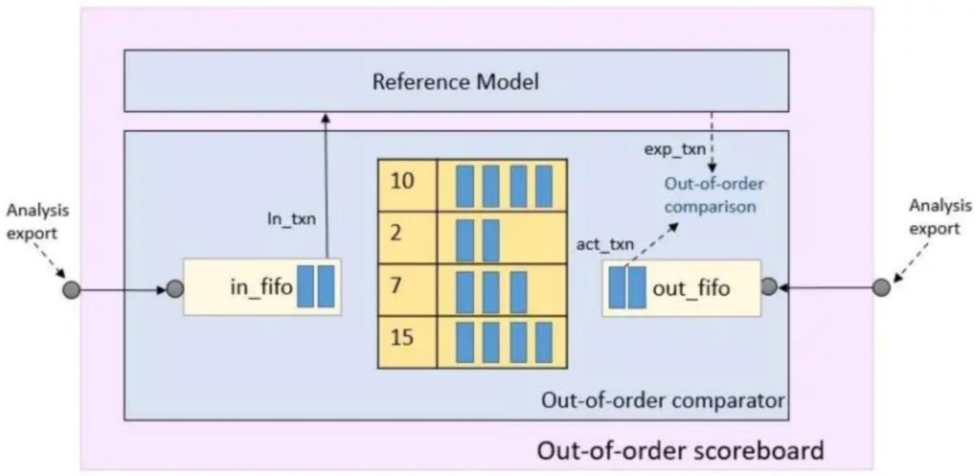


To implement such scoreboards, an easier way would be to implement TLM analysis FIFOs. In the below example, there are two monitors whose analysis port is connected to the scoreboard to provide input and output transactions.

```
class inorder_sb extends uvm_scoreboard;

  `uvm_component_utils(inorder_sb)
  uvm_analysis_export #(txn) in_export, out_export;
  uvm_tlm_analysis_fifo #(txn) in_fifo, out_fifo;

  function new (string name = "inorder_sb" , uvm_component parent = null) ;
    super.new(name, parent);
  endfunction


  function void build_phase (uvm_phase phase);
    in_fifo    = new("in_fifo", this);
    out_fifo   = new("out_fifo", this);
    in_export  = new("in_export", this);
    out_export = new("out_export", this);
  endfunction


  function void connect_phase (uvm_phase phase);
    in_export.connect(in_fifo.analysis_export);
    out_export.connect(out_fifo.analysis_export);
  endfunction


  task run_phase( uvm_phase phase);
    txn in_txn;
    txn exp_txn, act_txn;
    forever begin
      in_fifo.get(in_txn);
      process_data(in_txn, exp_txn);
      out_fifo.get(act_txn);
      if (!exp_txn.compare(act_txn)) begin
        `uvm_error(get_full_name(), $sformat("%s does not match %s",
exp_txn.sprint(), act_txn.sprint()), UVM_LOW);
      end
    end
  endtask

  // Reference model
  task process_data(input txn in_txn, output txn exp_txn);
    // Generate expected txn for driven stimulus
    ...
    ...
  endtask
endclass
```

Out-of-order scoreboard

The out-of-order scoreboard is useful for the design whose output order is different from driven input stimuli. Based on the input stimuli reference model will generate the expected outcome of DUT and the actual output is expected to come in any order. So, it is required to store such unmatched transactions generated from the input stimulus until the corresponding output has been received from the DUT to be compared. To store such transactions, an associative array is widely used. Based on index value, transactions are stored in the expected and actual associative arrays. The entries from associative arrays are deleted when comparison happens for the matched array index.



```
class txn extends uvm_sequence_item;
  int id;
  // other class properties
  //...
endclass

class out_of_order_sb extends uvm_scoreboard;
  `uvm_component_utils(out_of_order_sb)
  uvm_analysis_export #(txn) in_export, out_export;
  uvm_tlm_analysis_fifo #(txn) in_fifo, out_fifo;
  // associative array of class type as txn and indexed by int
  txn expected_out_array[int];
  txn actual_out_array[int];

  // Store idx in separate queues.
  int expected_out_q[$], actaul_out_q[$];

  function new (string name = "out_of_order_sb" , uvm_component parent =
null) ;
    super.new(name, parent);
  endfunction
  function void build_phase (uvm_phase phase);
    in_fifo    = new("in_fifo", this);
    out_fifo   = new("out_fifo", this);
    in_export  = new("in_export", this);
    out_export = new("out_export", this);
  endfunction


  function void connect_phase (uvm_phase phase);
    in_export.connect(in_fifo.analysis_export);
    out_export.connect(out_fifo.analysis_export);
  endfunction


  task run_phase( uvm_phase phase);
    txn in_txn, out_txn;
    forever begin
      fork
        begin
          in_fifo.get(in_txn);
          process_data(in_txn);
        end
        begin
          out_fifo.get(out_txn);
          actual_out_array[out_txn.id] = out_txn;
          actaul_out_q.push_back(out_txn.id);
        end

      join
      compare_data();
    end
  endtask

  // check_phase to check whether any entry is pending in queues.
  function void check_phase(uvm_phase phase);
    super. check_phase(phase);
    if(expected_out_q.size() != 0) `uvm_info (get_full_name(),
$sformatf("expected_out_q size = %0d", expected_out_q.size()), UVM_LOW);
    if(actaul_out_q.size() != 0) `uvm_info (get_full_name(),
$sformatf("actaul_out_q size = %0d", actaul_out_q.size()), UVM_LOW);
  endfunction

  task process_data(txn in_txn);
    txn exp_out_txn;
    // Using reference models, generate output for input stimulus.
    // store expected output (exp_out_txn) in expected_out_array
    ...
    ...
    expected_out_array[in_txn.id] = exp_out_txn;
    expected_out_q.push_back(in_txn.id);
  endtask

  task compare_data();
    int idx;
    txn exp_txn, act_txn;
    if(expected_out_q.size() > && actaul_out_q.size() > 0) begin
```

```
      idx = expected_out_q.pop_front();

      // Look for idx in actual_out_array to see whether the output has been 
received for a driven stimulus or not.
      if(actual_out_array.exists(idx)) begin
        exp_txn = expected_out_array[idx];
        act_txn = actual_out_array[idx];

        if(!exp_txn.compare(act_txn)) begin
          `uvm_error(get_full_name(), $sformat("%s does not match %s", 
exp_txn.sprint(), act_txn.sprint()), UVM_LOW);
        end
        else begin
          expected_out_array.delete(idx);
          actual_out_array.delete(idx);
        end
      end
      else expected_out_q.push_back(idx); // exp_idx is not found in 
actual_out_array.
    end
  endtask
endclass
```

Difficult level questions

1. Difference between p_sequencer and m_sequencer

m_sequencer	p_sequencer
m_sequencer is associated with a monitor and is responsible for driving sequences.	p_sequencer does not actively drive sequences or generate transactions, thus is also called a virtual sequencer
Used along with monitors to dynamically respond to the design behavior	Controls other sequencers and it is not attached to any driver.
m_sequencer is a handle available by default in a sequence.	All sequences have a m_sequencer handle but they do not have a p_sequencer handle.
No separate macro is needed for the declaration	It is defined using macro `uvm_declare_p_sequencer(sequencer_name).

2. What is the virtual sequence? How is it used? Is it necessary to have virtual sequencer for virtual sequence?

Virtual Sequence and Virtual Sequencers

A virtual sequence is nothing but a container that starts multiple sequences on different sequencers.

Virtual sequencer controls other sequencers and it is not attached to any driver.

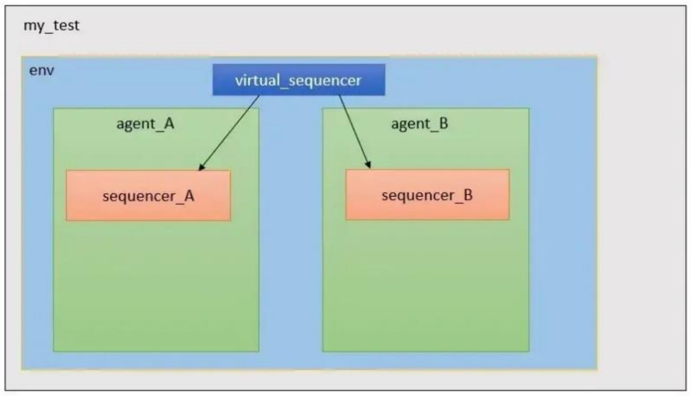
Virtual Sequence and Virtual Sequencer Usage

In SOC, there could be different modules that interact with different protocols So, we need different drivers to drive corresponding interfaces. So we usually keep separate agents to handle the different protocols. So, we need to execute sequences on corresponding sequencers.

Another example can be thought of as multiple cores in SOC. There can be multiple cores present in SOC that can handle different operations on input provided and respond to the device differently. In this case, as well, different sequence execution becomes important on different sequencers.

A virtual sequence is usually executed on the virtual sequencer. A virtual sequence gives control to start different sequences.

It is recommended to use a virtual sequencer if you have multiple agents and stimulus coordination is required.



Virtual Sequencer

Why are the virtual_sequence and virtual_sequencer named virtual?

System Verilog has virtual methods, virtual interfaces, and virtual classes. "virtual" keyword is common in all of them. But, virtual_sequence and virtual_sequencer do not require any virtual keyword. UVM does not have uvm_virtual_sequence and uvm_virtual_sequencer as base classes. A virtual sequence is derived from uvm_sequence. A virtual_sequencer is derived from uvm_sequencer as a base class.

Virtual sequencer controls other sequencers. It is not attached to any driver and can not process any sequence_items too. Hence, it is named virtual.

It is not necessary to have a virtual sequencer for a virtual sequence.

3. Explain the virtual sequencer and its use.

Examples

```
Without virtual sequence and virtual sequencer

// No Virtual Sequencer
class core_A_sequencer extends uvm_sequencer #(seq_item);
  `uvm_component_utils(core_A_sequencer)

  function new(string name = "core_A_sequencer", uvm_component parent =
null);
    super.new(name, parent);
  endfunction

endclass
```

```
class core_B_sequencer extends uvm_sequencer #(seq_item);
  `uvm_component_utils(core_B_sequencer)

  function new(string name = "core_B_sequencer", uvm_component parent =
null);
    super.new(name, parent);
  endfunction
endclass

// base_test
class base_test extends uvm_test;
  env env_o;

  core_A_seq Aseq;
  core_B_seq Bseq;

  `uvm_component_utils(base_test)

  function new(string name = "base_test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env_o = env::type_id::create("env_o", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    Aseq = core_A_seq::type_id::create("Aseq");
    Bseq = core_B_seq::type_id::create("Bseq");

    Aseq.start(env_o.agt_A.seqr_A);
    Bseq.start(env_o.agt_B.seqr_B);

    phase.drop_objection(this);
  endtask
endclass
```

Output:

```
UVM_INFO sequence.sv(10) @ 0: uvm_test_top.env_o.agt_A.seqr_A@@Aseq
[core_A_seq] core_A_seq: Inside Body
UVM_INFO driver.sv(38) @ 0: uvm_test_top.env_o.agt_A.drv_A [core_A_driver]
Driving from core A
UVM_INFO sequence.sv(30) @ 50: uvm_test_top.env_o.agt_B.seqr_B@@Bseq
[core_B_seq] core_B_seq: Inside Body
UVM_INFO driver.sv(55) @ 50: uvm_test_top.env_o.agt_B.drv_B [core_B_driver]
Driving from core B
```

With virtual sequence and without a virtual sequencer

```
// virtual sequence
class virtual_seq extends uvm_sequence #(seq_item);
  core_A_seq Aseq;
  core_B_seq Bseq;
  core_A_sequencer seqr_A;
  core_B_sequencer seqr_B;

  `uvm_object_utils(virtual_seq)

  function new (string name = "virtual_seq");
    super.new(name);
  endfunction

  task body();
    `uvm_info(get_type_name(), "virtual_seq: Inside Body", UVM_LOW);
    Aseq = core_A_seq::type_id::create("Aseq");
    Bseq = core_B_seq::type_id::create("Bseq");

    Aseq.start(seqr_A);
    Bseq.start(seqr_B);
  endtask
endclass

// No Virtual sequencer
class core_A_sequencer extends uvm_sequencer #(seq_item);
  `uvm_component_utils(core_A_sequencer)

  function new(string name = "core_A_sequencer", uvm_component parent =
null);
    super.new(name, parent);
  endfunction

endclass

class core_B_sequencer extends uvm_sequencer #(seq_item);
  `uvm_component_utils(core_B_sequencer)

  function new(string name = "core_B_sequencer", uvm_component parent =
null);
    super.new(name, parent);
  endfunction
endclass
```

Output:

```
UVM_INFO sequence.sv(56) @ 0: reporter@@v_seq [virtual_seq] virtual_seq:
Inside Body
UVM_INFO sequence.sv(10) @ 0: uvm_test_top.env_o.agt_A.seqr_A@@Aseq
[core_A_seq] core_A_seq: Inside Body
UVM_INFO driver.sv(38) @ 0: uvm_test_top.env_o.agt_A.drv_A [core_A_driver]
Driving from core A
UVM_INFO sequence.sv(30) @ 50: uvm_test_top.env_o.agt_B.seqr_B@@Bseq
[core_B_seq] core_B_seq: Inside Body
UVM_INFO driver.sv(55) @ 50: uvm_test_top.env_o.agt_B.drv_B [core_B_driver]
Driving from core B
```

With **virtual sequence and virtual** sequencer using p_senquencer handle

```
// Virtual sequence
class virtual_seq extends uvm_sequence #(seq_item);
  core_A_seq Aseq;
  core_B_seq Bseq;

  core_A_sequencer seqr_A;
  core_B_sequencer seqr_B;
  `uvm_object_utils(virtual_seq)
  `uvm_declare_p_sequencer(virtual_sequencer)

  function new (string name = "virtual_seq");
    super.new(name);
  endfunction

  task body();
    `uvm_info(get_type_name(), "virtual_seq: Inside Body", UVM_LOW);
    Aseq = core_A_seq::type_id::create("Aseq");
    Bseq = core_B_seq::type_id::create("Bseq");

    Aseq.start(p_sequencer.seqr_A);
    Bseq.start(p_sequencer.seqr_B);
  endtask
endclass

// Virtual p_sequencer
class virtual_sequencer extends uvm_sequencer;
  `uvm_component_utils(virtual_sequencer)
  core_A_sequencer seqr_A;
  core_B_sequencer seqr_B;

  function new(string name = "virtual_sequencer", uvm_component parent =
null);
    super.new(name, parent);
  endfunction
endclass
```

Output:

```
UVM_INFO sequence.sv(56) @ 0: uvm_test_top.env_o.v_seqr@@v_seq [virtual_seq]
virtual_seq: Inside Body
UVM_INFO sequence.sv(10) @ 0: uvm_test_top.env_o.agt_A.seqr_A@@Aseq
[core_A_seq] core_A_seq: Inside Body
UVM_INFO driver.sv(38) @ 0: uvm_test_top.env_o.agt_A.drv_A [core_A_driver]
Driving from core A
UVM_INFO sequence.sv(30) @ 50: uvm_test_top.env_o.agt_B.seqr_B@@Bseq
[core_B_seq] core_B_seq: Inside Body
UVM_INFO driver.sv(55) @ 50: uvm_test_top.env_o.agt_B.drv_B [core_B_driver]
Driving from core B
```

With **virtual sequence and virtual** sequencer but without using p_senquencer handle

```
// virtual sequence
class virtual_seq extends uvm_sequence #(seq_item);
  core_A_seq Aseq;
  core_B_seq Bseq;

  core_A_sequencer seqr_A;
  core_B_sequencer seqr_B;
  `uvm_object_utils(virtual_seq)

  function new (string name = "virtual_seq");
    super.new(name);
  endfunction

  task body();
    env env_s;
    `uvm_info(get_type_name(), "virtual_seq: Inside Body", UVM_LOW);
    Aseq = core_A_seq::type_id::create("Aseq");
    Bseq = core_B_seq::type_id::create("Bseq");

    // virtual_sequencer is created in env, so we need env handle to find
v_seqr.
    if(!$cast(env_s, uvm_top.find("uvm_test_top.env_o")))
`uvm_error(get_name(), "env_o is not found");

    Aseq.start(env_s.v_seqr.seqr_A);
    Bseq.start(env_s.v_seqr.seqr_B);
  endtask
endclass

// virtual_sequencer
class virtual_sequencer extends uvm_sequencer;
  `uvm_component_utils(virtual_sequencer)
  core_A_sequencer seqr_A;
  core_B_sequencer seqr_B;

  function new(string name = "virtual_sequencer", uvm_component parent =
null);
    super.new(name, parent);
  endfunction
endclass
```

Output:

```
UVM_INFO sequence.sv(55) @ 0: uvm_test_top.env_o.v_seqr@@v_seq [virtual_seq]
virtual_seq: Inside Body
UVM_INFO sequence.sv(10) @ 0: uvm_test_top.env_o.agt_A.seqr_A@@Aseq
[core_A_seq] core_A_seq: Inside Body
UVM_INFO driver.sv(38) @ 0: uvm_test_top.env_o.agt_A.drv_A [core_A_driver]
Driving from core A
UVM_INFO sequence.sv(30) @ 50: uvm_test_top.env_o.agt_B.seqr_B@@Bseq
[core_B_seq] core_B_seq: Inside Body
UVM_INFO driver.sv(55) @ 50: uvm_test_top.env_o.agt_B.drv_B [core_B_driver]
Driving from core B
```

4. What is the UVM barrier and its usage?

UVM Barrier

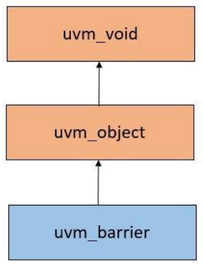
Similar to the UVM event, UVM provides another way to achieve synchronization with the UVM Barrier.

The UVM barrier provides multi-process synchronization that blocks a set of processes until the desired number of processes reaches a particular synchronizing point at which all the processes are released.

UVM Barrier Class Declaration:

```
class uvm_barrier extends uvm_object
```

UVM Barrier class hierarchy



UVM Barrier Usage

UVM Barrier is used to synchronize the number of processes based on the threshold value set by the set_threshold method. Once the threshold is reached, the processes are unblocked by the wait_for method.

UVM Barrier Examples

A basic uvm_barrier example with set_threshold method

In the below examples, the process task has two arguments string name and p_delay.

The name argument denotes process name and p_delay indicates delay time taken by the process to complete. Total 10 processes are forked with different execution delay time.

The processes will wait till the threshold being reached which is set by the set_threshold method to proceed further as demonstrated below.

```
module barrier_example();
  uvm_barrier br;
  task automatic process(string name, int p_delay);
    $display("@%0t: Process %s started", $time, name);
    #p_delay;
    $display("@%0t: Process %s completed", $time, name);
    br.wait_for();
    $display("@%0t: Process %s wait_for is unblocked", $time, name);
  endtask

  initial begin
    br = new("br");
    br.set_threshold(4);
    fork
      process("A", 5);
      process("B", 10);
      process("C", 20);
      process("D", 25);

      process("E", 30);
      process("F", 40);
      process("G", 50);
      process("H", 55);
      process("I", 60);
      process("J", 70);
    join
  end
endmodule
```

```
Output:

@0: Process A started
@0: Process B started
@0: Process C started
@0: Process D started
@0: Process E started
@0: Process F started
@0: Process G started
@0: Process H started
@0: Process I started
@0: Process J started
@5: Process A completed
@10: Process B completed
@20: Process C completed
@25: Process D completed
@25: Process A wait_for is unblocked
@25: Process B wait_for is unblocked
@25: Process C wait_for is unblocked
@25: Process D wait_for is unblocked
@30: Process E completed
@40: Process F completed
@50: Process G completed
@55: Process H completed
@55: Process E wait_for is unblocked
@55: Process F wait_for is unblocked
@55: Process G wait_for is unblocked
@55: Process H wait_for is unblocked
@60: Process I completed
@70: Process J completed
```

5. What is the UVM heartbeat and its usage?

UVM Heartbeat

The UVM Heartbeat acts as a watchdog timer that provides a flexible way for the environment to ensure that their descendants are alive. The uvm_heartbeat class is derived from uvm_object and it is associated with a specific objection object.

UVM Heartbeat Usage

The UVM heartbeat can detect simulation hang or lock-up conditions at an early stage rather than the expiry of the global simulation timeout. Thus, it can save the simulation time and terminate it at an early state. The lock-up may happen due to FSM issues, race around conditions between two communicating modules waiting for the response for transmitted packet or packet is not being transmitted, etc.

UVM Heartbeat Working

The uvm_heartbeat monitor activity for an objection object. The UVM Heartbeat detects testbench activity for registered components based on raises or drops objections on that objection object during the heartbeat monitor window. If the heartbeat monitor does not see any activity in that window, FATAL is issued and simulation is terminated.

Note: The UVM 1.1d has an objection object of type uvm_callbacks_objection which is derived from uvm_objection class and UVM 1.2 has an objection object of uvm_objection class.

6. What is SIngleton object and its usage?

Singleton Object in UVM

The singleton object is nothing but a single object for the class. The same object is returned even if a user tries to create multiple new objects. The class that allows creating a single object is called a singleton class.

In UVM, the uvm_root class has only one instance. Hence, it is said to be a singleton class.

Singleton Object Usage

A singleton object is useful wherever it is required to create a single object and want to restrict the user to create another object. For example, configuration classes can be written to behave as a singleton class.

Note: System Verilog and UVM do not have a separate construct to create a singleton object.

Singleton Object Example

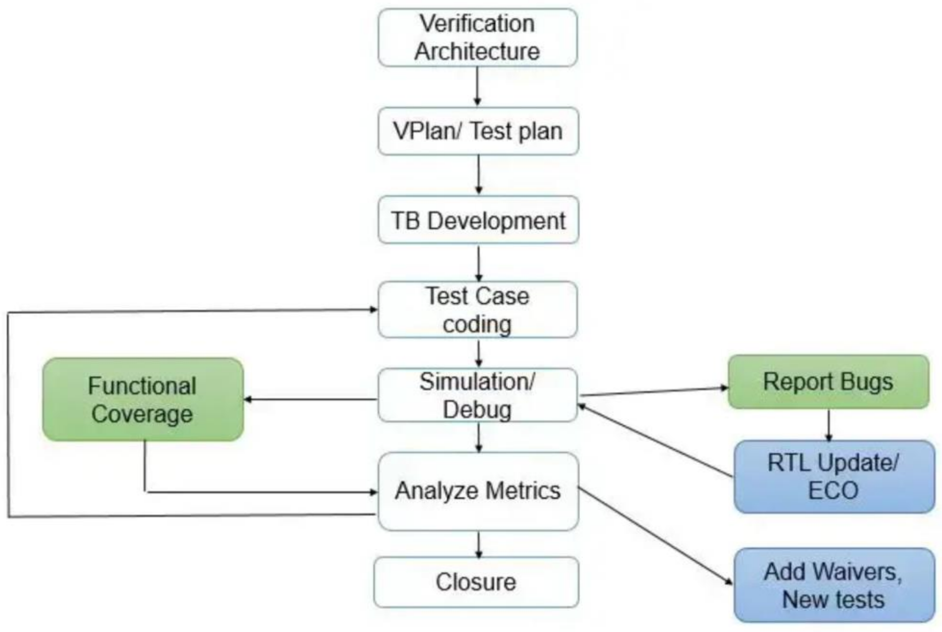
In the below example, there are two handles for the same singleton component sc1 and sc2 and objects have been tried to create twice. But on creating an object using handle sc2, separate memory will not be allotted and sc2 handle points to a memory allocated using handle sc1.

```
`include "uvm_macros.svh"
import uvm_pkg::*;

class singleton_comp extends uvm_component;
  static singleton_comp s_comp;
  rand bit [7:0] addr;
  rand bit [7:0] data;

  `uvm_component_utils_begin(singleton_comp)
    `uvm_field_int(addr, UVM_ALL_ON)
    `uvm_field_int(data, UVM_ALL_ON)
  `uvm_component_utils_end

  function new(string name = "singleton_comp", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  static function singleton_comp create_singleton();
    if(s_comp == null) begin
      $display("creating new object as it found null");
      s_comp = new();
    end
    else $display("object already exist, separate memory will not be
allocated.");
    return s_comp;
  endfunction

  task run_phase(uvm_phase phase);
    super.run_phase(phase);
  endtask
endclass

class base_test extends uvm_test;
  `uvm_component_utils(base_test)
  singleton_comp sc1, sc2;

  function new(string name = "base_test",uvm_component parent=null);
    super.new(name,parent);
  endfunction : new

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

    // create singleton object
    sc1 = singleton_comp::create_singleton();
    assert(sc1.randomize());
    `uvm_info(get_type_name, $sformatf("Printing sc1 = \n%s",sc1.sprint()),
UVM_LOW);
```

```
    // Trying to create another object but it won't be created
    sc2 = singleton_comp::create_singleton();
    `uvm_info(get_type_name, $sformatf("Printing sc2 = \n%s",sc2.sprint()),
UVM_LOW);
  endfunction : build_phase
endclass

module singleton_example;
  initial begin
    run_test("base_test");
  end
endmodule
```

```
Output:

creating new object as it found null
UVM_INFO testbench.sv(46) @ 0: uvm_test_top [base_test] Printing sc1 =
-------------------------------------------
Name            Type            Size  Value
-------------------------------------------
singleton_comp  singleton_comp  -     @349
  addr          integral        8     'h90
  data          integral        8     'ha6
-------------------------------------------

object already exist, separate memory will not be allocated.
UVM_INFO testbench.sv(50) @ 0: uvm_test_top [base_test] Printing sc2 =
-------------------------------------------
Name            Type            Size  Value
-------------------------------------------
singleton_comp  singleton_comp  -     @349
  addr          integral        8     'h90
  data          integral        8     'ha6
-------------------------------------------
```

7. Given an IP, you are the owner of the IP, what are the steps you start with and when will it be signed off?



ASIC Verification Flow

Verification Architecture

In the verification architectural phase, engineers decide what all verification components are required.

Verification Plan/ Testplan

The verification plan includes a test plan(list of test cases that target design features), functional coverage planning, module/block assignments to the verification engineers, checker, and assertion planning. The verification plan also involves planning for how verification components can be reused at system/ SOC level verification.

Testbench Development

As a part of testbench development, verification engineers develop testbench components, interface connections with the DUT, VIP integration with a testbench, inter-component connections within testbench (like monitor to scoreboard connection), etc.

Testcase Coding

A constraint-based random or dedicated test case is written for single or multiple features in the design. A test case also kicks off UVM-based sequences to generate required scenarios.

Simulation/ Debug

In this phase, engineers validate whether a specific feature is targetted or not, If not, again test case is modified to target the feature. With the help of a checker/ scoreboard, the error is reported if the desired design does not behave as expected. Using waveform analysis or log prints, the design or verification environment is judged and a bug is reported to the design team if it comes out to be a design issue otherwise, simulation is re-run after fixing the verification component.

Analyze Metrics

Assertions, code, and functional coverage are common metrics that are used as analysis metrics before we close the verification of the design.