UVM

INTERVIEW QUESTIONS

JAIRAJ MIRASHI
DESIGN VERIFICATION ENGINEER

1. **What is a UVM RAL model, and why is it required?**

A UVM RAL (Register Abstraction Layer) model is a fundamental component of the Universal Verification Methodology (UVM). It consists of a set of base classes used to create register models that emulate the register contents within a hardware design. The necessity for a UVM RAL model arises from the need to simplify interactions with the design's registers. Instead of sending individual bus transactions for every read and write operation, a register model provides a higher-level abstraction, making it easier to read from and write to the registers. Additionally, the register model maintains a mirrored value, representing the current state of the design, which enhances efficiency and convenience.

2. **What is p_sequencer, and where is it used in UVM?**

p_sequencer is a term used in UVM, referring to a handle that identifies the sequencer on which the current sequence should execute. It is typically accessed within sequences using the uvm_declare_p_sequencer macro. The primary use of p_sequencer is to initiate and execute other sequences, allowing for complex test scenarios and sequencing control within the UVM verification environment.

3. **What is an analysis port in UVM?**

An analysis port is a crucial Transaction-Level Modeling (TLM) mechanism in UVM. It serves as a communication channel that enables a component to broadcast a class object to multiple listeners. These listeners can implement various methods to perform different operations on the data they receive. Analysis ports facilitate flexible and scalable data communication between components in a UVM-based testbench.

4. **What is the difference between new() and create() in UVM?**

In UVM, new() and create() serve different purposes for object instantiation. The new() method is the traditional SystemVerilog way of creating an object instance. In contrast, the create() method is a UVM addition that leverages the factory mechanism. It allows the factory to return an object of the desired type, providing greater configurability within the UVM framework.

5. Is UVM independent of SystemVerilog?

No, UVM is not independent of SystemVerilog. UVM is built on top of the SystemVerilog language, meaning that it cannot be used with tools or environments that do not support SystemVerilog.

6. Why do we need to register a class with a factory in UVM?

Registering a class with a factory in UVM is not mandatory but highly beneficial. It enhances the reusability of components within the testbench and allows for the potential override of a registered class with a derivative component when necessary. This registration process provides flexibility and configurability in testbench construction.

7. What are Active and Passive modes in an agent in UVM?

In UVM, an agent typically comprises a driver, sequencer, and a monitor. Active and Passive modes define the behavior of the agent:

A Passive agent only monitors signals on the interface without driving data to the Design Under Test (DUT). In this mode, sequencer and driver components are not instantiated.

An Active agent can run sequences on its sequencer, drive signals to the DUT, and monitor the interface, enabling comprehensive testing capabilities.

8. What is a TLM FIFO in UVM?

A TLM FIFO (Transaction-Level Modeling FIFO) is a critical component in UVM used when two components with different clock domains need to operate independently and communicate. It acts as a buffer, allowing one component to send data at a faster rate while the other component receives data at a slower rate. TLM FIFOs ensure proper synchronization and data flow between components.

9. What are the advantages of using uvm_component_utils and uvm_object_utils in UVM?

Both uvm_component_utils and uvm_object_utils are macros used to register classes with the factory in UVM. The choice between them depends on whether the class is derived from uvm_component or uvm_object. There are no inherent advantages of one over the other; they offer separate ways to register classes with the factory, enhancing configurability and reuse.

10. How does a sequence start in UVM?

In UVM, a sequence can be initiated in two primary ways:

By calling its start() method directly.

By using the uvm_do macro. This allows sequences to be executed within the testbench, providing control over the verification environment's sequencing and flow.

11. What are the different phases in UVM?

The main phases in UVM, used for testbench development and synchronization, are as follows:

build_phase

connect_phase

end_of_elaboration_phase

start_of_simulation_phase

run_phase

extract_phase

check_phase

report_phase

final_phase

12. What is a virtual sequence, and what is a virtual sequencer in UVM?

A virtual sequence in UVM is a container designed to hold and execute multiple smaller sequences. It provides a means to manage and execute a group of sequences as a single entity.

A virtual sequencer is a similar concept, serving as a container to hold handles to other sequencers within an environment. This allows each sequence within a virtual sequence to be executed on the appropriate sequencer, enabling better control and organization of sequence execution.

13. What is the difference between uvm_do and uvm_send in UVM?

The uvm_do macro automatically creates a new object, randomizes it, and sends it to the specified sequencer in UVM.

On the other hand, uvm_send is used when the object is already created and randomized but needs to be executed on a sequencer. It is employed to send a pre-existing item to the sequencer for execution.

14. What is the difference between uvm_transaction and uvm_sequence_item in UVM?

uvm_transaction is the root base class for UVM transactions, featuring a timing and recording interface. It is deprecated for user-defined transactions, and its intended use is to record events to a vendor-specific transaction database using methods like accept_tr, begin_tr, and end_tr.

uvm_sequence_item is primarily used to define data objects and related methods, making it the preferred base class for user-defined transactions in UVM.

15. What are the benefits of using UVM?

Faster testbench prototyping due to provided base classes for drivers, monitors, sequencers, and more.

A well-defined reporting system supporting various verbosity levels.

Support for register model creation and maintenance.

Structured, plug-and-play components like agents for protocol support.

A factory mechanism for component overrides without modifying existing connections.

Configuration databases for sharing objects and data between components.

Transaction-Level Modeling (TLM) features for flexible data operations.

Promotes reusability, flexibility, uniformity, and robustness in testbench development.

16. Is it possible to have a user-defined phase in UVM?

A: Yes, it is possible to define custom phases in UVM. To achieve this, you need to create a new phase class inherited from uvm_task_phase, implement the exec_task or exec_func method, and then insert the custom phase into the existing schedule or domain object.

17. What is the difference between RAL backdoor and frontdoor accesses?

In RAL (Register Abstraction Layer) in UVM:

Backdoor access allows direct dumping of values onto DUT registers via a hard-coded RTL signal path. It does not consume simulation time.

Frontdoor access involves sending data as a transaction through an associated peripheral bus interface, consuming simulation time.

18. What is a phase objection in UVM?

A phase objection is a mechanism in UVM used for component synchronization during different phases. It allows a component to stall other components from proceeding to the next phase until it completes its own tasks. This is achieved using raise_objection() and drop_objection() methods from the uvm_phase class.

19. What is the difference between set_config_* and uvm_config_db in UVM?

The basic set_config_* methods in UVM are mapped to corresponding uvm_config_db operations. For example, set_config_int maps to uvm_config_db#(uvm_bitstream_t)::set, and set_config_string maps to uvm_config_db#(string)::set. These methods are used to configure components in the testbench.

20. What are the different factory override types in UVM?

Factory overrides in UVM can be done in four different ways:

Instance override by type of the component/object.

Instance override by name of the component/object.

Type override by type of the component/object.

Name override by type of the component/object. These overrides provide flexibility in customizing testbench behavior.

21. How can we access a DUT signal in a UVM component or sequence?

To access signals within a Design Under Test (DUT) in a UVM component or sequence:

Interface signals can be accessed via a virtual interface handle that points to the actual physical interface.

Signals within the DUT can be accessed directly by providing a hierarchical RTL path to uvm_hdl_* functions like uvm_hdl_force, uvm_hdl_deposit, and uvm_hdl_read.

For example:

```
uvm_hdl_force("top.eatable.fruits.apple.slice", 2);
uvm_hdl_deposit("top.eatable.fruits.apple.slice", 3);
uvm_hdl_read("top.eatable.fruits.apple.slice", rdata);
```

22. What is RALGEN, and how is it used?

RALGEN is a tool by Synopsys used to generate Register Abstraction Layer (RAL) models from an IPXACT specification file. To use RALGEN:

Specify the required options and select the block for which you need to generate class structures.

Provide these options to the tool.

RALGEN will generate the RAL model classes based on the IPXACT specification.

23. What are desired and mirrored values in RAL?

In RAL (Register Abstraction Layer):

Desired values are the values that are intended to be written to the design registers and can be updated later.

Mirrored values are the latest known values that reflect the actual values in the Design Under Test (DUT).

24. What are reg2bus and bus2reg functions used for in RAL?

reg2bus and bus2reg are RAL functions designed for converting between generic register contents and actual bus transactions:

reg2bus is used to convert register data into bus transactions.

bus2reg is used to convert bus transactions into register data. These functions are protocol-specific and are defined based on the protocol being used.

25. How would you debug a config database name or path mismatch problem in UVM?

To debug a configuration database name or path mismatch issue in UVM:

Use the command-line define +UVM_CONFIG_DB_TRACE to enable tracing for SET and GET calls made to the configuration database.

This trace will provide information related to the path, instance, and other details of the configuration database calls, helping identify any mismatches.

26. What are the different testbench components in UVM?

Some of the major testbench components in UVM include:

Driver

Monitor

Scoreboard

Sequencer

Agent

Environment

Test

Sequences

27. Which UVM phase typically takes more time and why?

Run-time phases in UVM typically consume more time as they are the major phases that consume simulation time. The duration of these phases can vary depending on the specific tests being executed, as they may test different aspects of the design.

28. How do you connect a monitor with a scoreboard in UVM?

To connect a monitor with a scoreboard in UVM, you can:

Declare the implementation of an analysis port within the scoreboard.

In the environment's connect method, connect the monitor's analysis port with the analysis port declared in the scoreboard.

29. How do you connect a driver and sequencer in UVM?

To connect a driver and sequencer in UVM:

The driver has a TLM (Transaction-Level Modeling) port called seq_item_port.

In an agent's connect method, you can connect the driver's seq_item_port with the sequencer's seq_item_export.

30. What are uvm_config_db and uvm_resource_db in UVM?

Both uvm_config_db and uvm_resource_db are mechanisms in UVM used for placing an object in a central lookup table under a specified name and path. This allows another component to retrieve the object using the same name and path.

uvm_config_db provides a convenient interface on top of uvm_resource_db for configuring uvm_component instances.

31. Can you provide pseudo-code for implementing an AHB-Lite driver in UVM?

Certainly! Below is pseudo-code for implementing an AHB-Lite driver in UVM. It's important to note that AHB-Lite is a pipelined protocol, so the address phase of the next transaction should be active while the data phase of the current transaction is ongoing. This is achieved by starting the same task twice in a fork-join.

```
class ahb_driver extends uvm_driver;

  semaphore sema4;

  virtual task run_phase(uvm_phase phase);
    fork
      drive_tx();
      drive_tx();
    join
```

```
  endtask

  virtual task drive_tx();
    // 1. Get hold of a semaphore
    // 2. Get transaction packet from sequencer
    // 3. Drive the address phase
    // 4. Release semaphore
    // 5. Drive data phase
  endtask
endclass
```

32. What does a sequence typically contain in UVM?

A: In UVM, a sequence typically contains a task called body() within which you can write the actual stimulus to test the design for a particular feature. The body() task defines the sequence of actions and transactions to be executed during the sequence's execution.

33. Can you explain DRIVER AND SEQUENCE HANDSHAKING? (LINK)

```
  sequence                          sequencer              driver
                                           ||
  task body();                             ||               task
run_phase(uvm_phase phase);
                                           ||                  forever
    begin                                  ||                    begin
        start_item(req);                   ||  seq_item_port.get_next_item(req);
                                           ||
        req.randomize();                   ||        drive(req);
                                           ||
        finish_item(req);                  ||     seq_item_port.item_done();
    end                                    ||                    end
  endtask                                  ||               endtask
```

 The sequence will be waiting for a request (get_next_item) from the driver.

 After receiving the request, the sequence will call "start_item(req)" and generate the sequence_item or transactions using "req.randomize()".

The generated transactions will be sent to the driver with the help of a sequencer.

The driver will receive the transactions and drive (req) them to DUT.

Once all the data is driven, the driver will send an acknowledgment as item_done.

The sequence will wait for the acknowledgment from the driver and then call finish_item(req).

📝 Here's an example of how to write a UVM sequence:

```
class mem_sequence extends uvm_sequence#(mem_seq_item);
 `uvm_object_utils(mem_sequence)

 // Constructor
 function new(string name = "mem_sequence");
  super.new(name);
 endfunction

 // Body method
 virtual task body();
  req = mem_seq_item::type_id::create("req"); // create the req (seq item)
  wait_for_grant(); // wait for grant
  assert(req.randomize()); // randomize the req
  send_request(req); // send req to driver
  wait_for_item_done(); // wait for item done from driver
  get_response(rsp); // get response from driver
 endtask
endclass
```

34. Is the build_phase() method executed top-down in the UVM component hierarchy?

Yes, in UVM, the build_phase() method is executed top-down in the testbench component hierarchy. This means that a parent component constructs its child components, which can further construct their own child components using the build_phase() method.