1. What are the ways to avoid race conditions between testbench and RTL using System Verilog?

* Use of always\_ff blocks: Ensure that flip-flops are clocked properly to avoid unintended races.
* Synchronization using semaphore: Use semaphores to control access to shared resources between testbench and RTL, ensuring mutual exclusion.
* @ (at) operator for synchronization: Properly use event control (@) to synchronize between testbench stimuli and RTL responses.
* Use of fork and join for process synchronization: Ensure that parallel processes in the testbench are properly synchronized.
* Avoiding blocking assignments in RTL during simulation: In RTL, avoid blocking assignments (=) in places where they could cause race conditions.

1. How can you detect a deadlock condition in FSM?

* State transition analysis: Examine the FSM for unreachable or stuck states where transitions never occur.
* Formal Verification: Use formal methods to prove that all states are reachable or check that no deadlocks occur during simulation.
* Timeouts or Watches: Implement timeouts in the testbench to check if the FSM is stuck in a state for too long, indicating a deadlock.
* Logging: Use extensive logging in the testbench to track state transitions and identify cases where no progress is made.

1. What is circular dependency and how can we avoid this problem?

Circular Dependency: This occurs when two or more modules or components depend on each other in a loop, creating an interdependency that cannot resolve.

Avoidance:

* Refactor the design to break the cycle by removing direct dependencies between the modules.
* Introduce intermediate signals or buffers to decouple the components.
* Use a hierarchical design approach where dependencies are structured in a clear, non-circular manner.

1. In a simulation environment under what condition should the simulation end?

* When the testbench has completed all the scenarios or coverage points.
* When a predefined time or clock cycle limit is reached.
* When an error or assertion failure occurs.
* When all coverage goals are met.
* When the simulation reaches a specific point of completion as defined by the test plan (e.g., all transactions processed).
* Use the $finish or $stop system functions in SystemVerilog for controlled termination of the simulation.

1. What is DPI in System Verilog? Explain DPI export and import.

* DPI (Direct Programming Interface): DPI allows the interaction between SystemVerilog and external C/DPI-based code. It enables SystemVerilog to call C functions and vice versa.
* DPI Export: The export defines C functions that can be called from SystemVerilog. It’s used to declare a C function that will be accessible in the SystemVerilog code.

export "DPI" function void c\_function\_name();

* DPI Import: The import is used to bring SystemVerilog functions or tasks into C code. It allows external C code to call SystemVerilog code.

import "DPI" function void sv\_function\_name();

1. Difference between clocking block and modport? Are both required in an interface block?

* Clocking block is used when the interface signals are clocked, whereas modport is essential to control access direction for different modules. Both can be used together within an interface but are not strictly required for every interface.

Clocking block:

* Defines a clocking domain for synchronization of signals.
* Specifies which signals should be sampled on a given clock edge.
* Helps in handling timing differences between different clock domains.
* Example:

clocking cb @(posedge clk);

input data;

output ready;

endclocking

Modport:

* Defines direction and access permissions (input, output, etc.) for an interface.
* Used for specifying how signals in the interface should be accessed by different modules (master, slave, etc.).
* Example

interface my\_if;

logic data;

modport master (input data);

modport slave (output data);

endinterface

1. What is the need for the scope resolution operator?

The scope resolution operator (::) is used to:

* Access module parameters or functions from different hierarchical levels or namespaces.
* Qualify names within a specific module, interface, or class to avoid ambiguity.
* Example

module A;

parameter int PARAM = 10;

endmodule

module B;

A::PARAM my\_param; // Accesses PARAM from module A

endmodule

1. What are synthesizable constructs in SV that are used in RTL design?

* Registering Variables (reg, logic): Used for defining storage elements.
* Combinational Logic (assign): Used for continuous assignments for combinational logic.
* always\_ff and always\_comb blocks: For describing clocked processes (flip-flops) and combinational logic, respectively.
* if-else, case, for loops: For implementing decision-making logic and loops.
* Instantiation of modules and primitives: Synthesizable RTL designs often instantiate lower-level modules.
* No initial blocks: Since initial blocks are non-synthesizable, it’s essential to avoid them in RTL design.

1. What is the difference between a Directed Test and a Random Test?

Directed Test:

* Test cases are manually written with predefined stimulus and expected results.
* It targets specific functionality or bug scenarios, providing controlled testing.
* More predictable and easier to debug.

Random Test:

* Test cases are randomly generated (e.g., using constrained random generation).
* It covers a broad range of input conditions, including edge cases, but is less predictable.
* Ideal for discovering unknown bugs or corner cases.

1. How can you connect a Generator and a Driver with a Queue instead of Mailbox?

Queue: A queue is a dynamic data structure that can hold multiple items and provide FIFO access. It can be used to decouple the generator and driver in a testbench.

class generator;

rand bit [7:0] data;

// generate data

endclass

class driver;

queue<generator> gen\_queue;

// consume data from gen\_queue

endclass

This avoids the overhead of mailbox operations while allowing the driver to pull the data in a FIFO manner.

1. What is the difference between sequence and property in an SV assertion?

Sequence:

* Describes a temporal sequence of events or conditions.
* A sequence defines patterns of events over time, such as event sequences or specific conditions occurring in a specific order.
* Example:

sequence seq;

@(posedge clk) a ##1 b ##2 c;

endsequence

Property:

* Defines an assertion property, which is a condition that must always hold true during simulation.
* Properties can be used with sequences to assert that specific behaviors or conditions occur over time.
* Example:

property p;

@(posedge clk) a |-> b;

endproperty

assert property(p);