1. Different types of code coverage.

Code coverage measures how much of the design code is exercised during simulation. There are different types of code coverage in SystemVerilog, which help verify that the RTL design has been fully tested. These types include:

* Statement Coverage: Ensures that every statement in the code is executed at least once. It checks if each line of code has been visited.
* Branch Coverage: Ensures that every branch of a decision (like an if or case statement) is executed. It checks both the true and false branches of a decision point.
* Condition Coverage: Checks whether every individual condition in a compound statement (such as if or while) evaluates to both true and false.
* Path Coverage: Ensures that every possible path through the design's control flow is exercised. Path coverage ensures that combinations of decisions are also tested, including the possible ordering of conditions.
* Toggle Coverage: Tracks whether a signal changes its state (from 0 to 1 or 1 to 0) during simulation. It checks if all signal toggles are covered.
* FSM Coverage: Specifically used to cover all states and state transitions of a finite state machine (FSM) in the design.

1. What is the difference between code and functional coverage?

* Code Coverage: Measures how much of the design code (like RTL or Verilog/VHDL code) is executed during simulation. It does not provide insight into whether the design is correctly functioning, only that the code paths are being exercised.
* Functional Coverage: Measures how well the design’s functionality is being tested. It tracks whether all functional behaviours (such as certain values, transitions, or conditions) are being tested during the simulation, which provides more insight into the correctness of the design.

1. If the functional coverage is more than code coverage, what does it mean?

If functional coverage is greater than code coverage, it suggests that the testbenches are effectively verifying specific behaviours or conditions (e.g., value ranges, state transitions, or particular functional scenarios) without necessarily covering all the code paths. This could mean:

* The design might be functionally correct for the intended use cases, but there might still be portions of the code that aren't tested or exercised.
* The testbench may be well-targeted but not fully exhaustive in terms of code paths.

1. How will you test the functionality of interrupt using functional coverage?

To test the functionality of interrupts using functional coverage, you would typically track the following aspects:

* Interrupt assertion and de-assertion: Cover the different scenarios where the interrupt is triggered and cleared.
* Interrupt prioritization: If multiple interrupts are possible, track the order in which they occur.
* Interrupt handling logic: Check how the system reacts to interrupts (e.g., interrupt servicing, masking, etc.).

Example:

covergroup interrupt\_coverage;

bit irq\_asserted;

bit irq\_serviced;

bit irq\_priority;

// Cover when interrupt is asserted and serviced

coverpoint irq\_asserted { bins asserted = {1}; }

coverpoint irq\_serviced { bins serviced = {1}; }

coverpoint irq\_priority { bins high\_priority = {1}; }

endgroup

1. How will be your approach if code coverage is 100% but functional coverage is too low?

If code coverage is 100%, but functional coverage is low, it suggests that while all lines of code have been executed, not all possible scenarios, values, or state transitions are being tested. The following actions can be taken:

* Increase the diversity of test scenarios: Add test cases to cover a broader range of input combinations and edge cases that could trigger functional behaviors.
* Improve constraint-driven randomization: Refine the randomization constraints to increase the chances of testing all required functional paths.
* Focus on untested functionality: Use functional coverage reports to focus on areas of functionality that are not being exercised.
* Check for missing or incomplete coverage points: If certain important functionality has not been captured, extend the coverage model to track those functional behaviors.

1. During a project if we observe high functional coverage(close to 100%) and low code coverage (say<60%) what can be inferred?

If functional coverage is high and code coverage is low, it could mean:

* Not all code is relevant: The code that isn’t covered may not be functionally necessary or might not be exercised by the current tests.
* Inadequate test scenarios for code paths: While the functional requirements are being met, the testbenches might not be covering all the possible code paths, which means some code may remain untested.
* Dead code: There could be dead or unreachable code in the design that’s not necessary for functionality but still present in the design.

1. Write rand constraint on a 3-bit variable with distribution 60% for 0 to 5 and 40% for 6,7. Write a cover point for this.

Constraint

class my\_class;

rand bit [2:0] a; // 3-bit variable

constraint dist\_constraint {

a inside {[0:5]}; // Cover values 0 to 5 (60%)

a inside {[6,7]}; // Cover values 6 and 7 (40%)

}

endclass

Cover point

covergroup cg;

coverpoint a { bins low = {[0:5]}; bins high = {[6:7]}; }

endgroup

1. How will you make sure that address ranges from 0x2000 to 0x9000 are covered?

To ensure that the address range from 0x2000 to 0x9000 is covered, you can write a constraint for the address and create a coverpoint to track this range.

class my\_address\_class;

rand bit [31:0] addr;

constraint addr\_constraint {

addr inside {[32'h2000:32'h9000]}; // Ensure address range 0x2000 to 0x9000

}

endclass

covergroup addr\_covergroup;

coverpoint addr { bins addr\_range = {[32'h2000:32'h9000]}; }

endgroup

1. What are the different ways to increase SV code coverage?

* Add more tests: More test cases that cover different execution paths can increase code coverage.
* Refine test scenarios: Add specific test cases that target untested branches, paths, or conditions in the design.
* Increase randomization: Use constrained randomization to cover edge cases and uncommon paths.
* Use formal verification tools: Formal tools can mathematically prove that all possible paths and states are covered.
* Monitor and adjust coverage models: Use coverage reports to identify uncovered code paths and refine the test scenarios.

1. Can covergroups be defined and used inside classes?

Yes, covergroups can be defined and used inside classes. In fact, covergroups can be instantiated in a class to provide functional coverage for object instances of that class.

Example:

class my\_class;

rand bit [3:0] a;

covergroup cg;

coverpoint a { bins low = {0, 1}; bins high = {2, 3}; }

endgroup

// Class constructor to initialize covergroup

function new;

cg = new;

endfunction

endclass

1. How many bins are created in the following examples for coverpoint cp\_x?

bit[3:0] var\_x;

covergroup test\_cg @(posedge clk);

cp\_x: coverpoint car\_x{

bins low\_bins[] = {[0:3]};

bins med\_bins = {[4:12]};

}

endgroup

* low\_bins[]: This bin covers values from 0 to 3, so it will have 4 bins (0, 1, 2, 3).
* med\_bins: This bin covers values from 4 to 12, so it will have 9 bins (4, 5, 6, 7, 8, 9, 10, 11, 12).
* Thus, 13 buns are created in total.