1. What is the need for coverage in System Verilog?

Coverage in SystemVerilog is essential for ensuring that the testbench thoroughly exercises the design under test (DUT). It is a measure of how much of the design has been tested during simulation. The main reasons for using coverage are:

* Quality assurance: To determine if the testbenches are adequate and if all corner cases of the design are tested.
* Uncovered functionality: Coverage helps identify sections of the design that have not been tested, ensuring no part of the design is overlooked.
* Efficiency: Helps prioritize areas of the design that need more testing, saving time and effort by focusing on uncovered or under-tested areas.
* Bug detection: By monitoring coverage, designers can catch bugs that may only occur in certain untested conditions.

1. Explain:
   1. Functional coverage

Functional coverage measures whether all the possible scenarios for a design have been exercised during simulation. It tracks specific values, ranges, transitions, or combinations of signal values in the DUT and ensures they are covered by the test cases

Example: Tracking the number of times a 2-bit signal a takes the values 00, 01, 10, and 11 during simulation.

* 1. Code coverage

Code coverage measures how much of the code (i.e., RTL or design code) has been executed during simulation. It typically includes statement coverage, branch coverage, and toggle coverage.

Example: If a line of RTL code contains an if statement, code coverage ensures that both the if and else parts of the statement are executed at least once during simulation.

* 1. Assertion coverage

Assertion coverage tracks how often the assertions in the testbench are evaluated and checks if they pass or fail during simulation. It helps to monitor the effectiveness of the assertions in detecting design violations.

Example: If there is an assertion in the design that checks if a signal a is always high when signal b is low, assertion coverage will show how often this condition hs been checked during simulation.

1. Explain the use of auto\_bin\_mux

auto\_bin\_mux is a keyword used in coverpoints and covergroups to automatically generate coverage bins for all possible values or conditions of an expression, typically a variable or signal. This keyword is particularly useful for generating bins for variables that are randomly set and have many possible values.

Example:

covergroup cg;

bit [3:0] data; // 4-bit signal

coverpoint data {

option.auto\_bin\_mux = 1; // Automatically create bins for all 16 values (0 to 15)

}

endgroup

In this example, auto\_bin\_mux ensures that coverage bins are automatically created for all the 16 possible values that the data signal can take (0 to 15). Without auto\_bin\_mux, you would have to manually define each bin, which could be tedious.

1. Explain with an example how can you write transition coverage for a 2-bit input “a” to cover transitions for:
   1. 0 -> 1 & 1 -> 0 transition for bits “0” & “1”.

Transition coverage tracks state changes between signal values. For a 2-bit input a, we need to monitor transitions from 0 to 1 and 1 to 0 for both the 0th and 1st bits.

covergroup cg\_transition;

bit [1:0] a; // 2-bit signal

// Cover transitions for bit 0 (a[0])

coverpoint a[0] {

transition a0\_transition[2] = (a[0] == 0) -> a[0]; // 0 -> 1

transition a0\_transition[3] = (a[0] == 1) -> a[0]; // 1 -> 0

}

// Cover transitions for bit 1 (a[1])

coverpoint a[1] {

transition a1\_transition[2] = (a[1] == 0) -> a[1]; // 0 -> 1

transition a1\_transition[3] = (a[1] == 1) -> a[1]; // 1 -> 0

}

endgroup

In this code, the transitions from 0 to 1 and 1 to 0 are tracked for each bit of the 2-bit signal a. The transition coverage will record if both transitions are covered for a[0] and a[1].

* 1. Transition of 2-bit signal from 00 to other 3 states & from 11 to other 3 states.

covergroup cg\_transition;

bit [1:0] a; // 2-bit signal

// Transition from 00 to other states

coverpoint a {

transition a\_transition\_from\_00[3] = (a == 2'b00) -> a; // 00 -> 01, 00 -> 10, 00 -> 11

}

// Transition from 11 to other states

coverpoint a {

transition a\_transition\_from\_11[3] = (a == 2'b11) -> a; // 11 -> 00, 11 -> 01, 11 -> 10

}

endgroup

This code ensures that the transitions for the signal a from the state 00 to all other states (01, 10, 11) are tracked, as well as from the state 11 to all other states (00, 01, 10).

1. What are illegal bins? Explain with an example.

Illegal bins represent values or conditions that are invalid or out of range according to the design specification. You can use illegal bins to capture unexpected or erroneous states that should not occur in the simulation.

Example:

covergroup cg;

bit [2:0] a; // 3-bit signal

coverpoint a {

bins valid = { 0, 1, 2 }; // Valid values

bins illegal = default; // All other values are considered illegal

}

endgroup

In the example:

* he valid bin will cover the values 0, 1, and 2 for the signal a.
* The illegal bin will cover any value outside the set {0, 1, 2}, meaning all other values of a (3, 4, 5, ..., 7) are considered illegal.

1. What are ignore bins? Explain with an example.

Ignore bins are used when you want to ignore certain values during coverage collection. These values are still valid but will not be considered for coverage reporting.

Example:

covergroup cg;

bit [2:0] a; // 3-bit signal

coverpoint a {

bins valid = { 0, 1, 2 }; // Valid values

bins ignore = { 3 }; // Ignore value 3

}

endgroup

In the example:

* The valid bin will cover values 0, 1, and 2.
* The ignore bin will ensure that the value 3 is ignored during coverage collection, i.e., it will not be reported as covered, but it will not be considered for invalid state reporting.

1. Difference between ignore and illegal bins.

* Illegal bins capture values that are outside the valid or expected range and are typically used to identify unexpected or erroneous states.
* Ignore bins are used to exclude certain values from coverage reporting, even if they are within a valid range. They essentially say, "I don't want to track this value."

1. What is covergroup?

A covergroup in SystemVerilog is a construct used to specify coverage points for a particular signal or condition. It is used to track and collect coverage information for specific signals, states, or transitions during simulation.

1. What is cross coverage?

Cross coverage is a way to track combinations of values between two or more signals. It allows you to cover combinations of different signals or conditions to ensure all possible interactions between them are tested.

1. How can you ignore the bins in cross-coverage? Explain with an example.

You can ignore certain combinations of signals in cross-coverage by defining ignore bins. These bins will prevent certain cross-combinations from being tracked during coverage collection.

Example:

covergroup cg;

bit [1:0] a;

bit [1:0] b;

cross a, b {

bins valid = {0, 1, 2};

bins ignore = {3}; // Ignore cross-combinations where a or b is 3

}

endgroup

In this case, the cross-coverage will ignore any combinations where either a or b is 3.

1. Consider a 4-bit variable. Write binds to cover min value, max value and odd values. Then consider one more variable bit rd\_wr\_access. Write bins to cross cover above and in this case.

covergroup cg\_cross;

bit [3:0] a; // 4-bit variable

bit rd\_wr\_access; // Read/Write access

cross a, rd\_wr\_access; // Cross-coverage between `a` and `rd\_wr\_access`

endgroup