1. How to check whether randomization is successful or not?

In SystemVerilog, you can use the $randomize method or randomize() on an object. To check if the randomization is successful, you can check the return value of the randomization function. The randomize() function returns a boolean value.

1. You have an array, int arr[6:10]. Randomize arr between 1 to 20 without using randc.

int arr[6:10]; // Array with indices 6 to 10

constraint arr\_constraint {

foreach (arr[i]) arr[i] inside {[1:20]};

}

// Randomize

if (!arr.randomize()) begin

$display("Randomization failed!");

end else begin

$display("Randomized arr: %p", arr);

end

1. Is it possible to call a function from constraint? If yes, explain with an example.

Yes, it is possible to call functions from a constraint in SystemVerilog. However, the function should be side-effect-free and must return a value that can be used within the constraints. This is useful for checking conditions dynamically during randomization.

Example:

class MyClass;

rand bit [7:0] a;

rand bit [7:0] b;

// Function to check if the sum is even

function bit is\_even(input int value);

return (value % 2 == 0);

endfunction

constraint c {

is\_even(a + b); // Using function within a constraint

}

endclass

MyClass obj;

if (!obj.randomize()) begin

$display("Randomization failed!");

end else begin

$display("Randomized a = %0d, b = %0d", obj.a, obj.b);

end

1. Write a constraint for an 8-bit variable that provides a distribution of 70% for the range 0-50 and the remaining 30% for the range 50-255.

bit [7:0] x;

constraint dist\_constraint {

x dist { [0:50] : 70, [51:255] : 30 };

}

// Randomize the variable

if (!x.randomize()) begin

$display("Randomization failed!");

end else begin

$display("Randomized x = %0d", x);

end

1. Derive odd numbers within the range of 10 to 30 using SV constraint.

bit [7:0] y;

constraint odd\_constraint {

y inside {11, 13, 15, 17, 19, 21, 23, 25, 27, 29};

}

// Randomize the variable

if (!y.randomize()) begin

$display("Randomization failed!");

end else begin

$display("Randomized odd y = %0d", y);

end

1. Write a constraint: divisible by 5.

bit [7:0] z;

constraint divisible\_by\_5 {

z % 5 == 0;

}

// Randomize the variable

if (!z.randomize()) begin

$display("Randomization failed!");

end else begin

$display("Randomized z = %0d", z);

end

1. Write a constraint to detect an odd number of 1s in an 8-bit sequence.

bit [7:0] w;

constraint odd\_ones\_constraint {

$countones(w) % 2 == 1; // Count the number of 1s and check if it's odd

}

// Randomize the variable

if (!w.randomize()) begin

$display("Randomization failed!");

end else begin

$display("Randomized w = %0b", w);

end

1. How to disable constraints?

To disable a constraint, you can use the disable keyword. This is typically used with constraints inside classes or structures.

disable constraint\_name; // Disables the named constraint

Or

disable this.constraint\_name; // Disable a constraint within the current scope

1. How to disable randomization?

To disable randomization, you can use the randomize() function and explicitly ensure that the randomization process is skipped or prevent it from running. Alternatively, you can bypass randomization by simply not calling the randomize() function.

class MyClass;

rand bit [7:0] a;

endclass

MyClass obj;

obj.a = 10; // Set to a fixed value, thus "disabling" randomization

1. Is it possible to generate random numbers without using rand or randc keywords?

Yes, you can generate random numbers without rand or randc by using $random or $urandom. These system functions generate random values and are often used in procedural code outside of constraints.

1. What is constraint solve-before?

The solve-before keyword is used in SystemVerilog to control the order in which constraints are solved. It specifies that one constraint should be solved before another, ensuring the constraints are applied in the desired sequence.

constraint c1 {

x == y;

}

constraint c2 solve-before c1 {

y == 10;

}

1. What are all the bins that are generated by the following code:

coverpoint data {bins b1={20,30,50};

bins b2[] = {[1:8],11};

bins b3[4] = {0:8};

* b1: A single bin containing values {20, 30, 50}.
* b2: A set of bins corresponding to the values in the range {1:8} (i.e., 1 to 8) and an additional bin for 11. This results in 9 bins.
* b3: 4 bins, each covering the range from 0 to 8 (i.e., 0, 1, 2, ..., 8), for a total of 4 bins.
* Thus, b1 -> 3 values
* b2 -> 9 values
* b3 -> 4 values
* A total of 16 bins