DAY 53 -111 DAYS VERIFICATION CHALLENGE

Topic: Messaging, BFMs

Skill: Verilog, RTL Design

DAY 53 CHALLENGE:

1. Write a code to generate constrained random numbers in Verilog.

```
module random_number();

reg [7:0] random_number;  // 8-bit wide register to store the random
number

  integer seed;  // Seed for random number generation

  integer i;


  initial begin

    seed = 12345;  // Initialize the seed


    for (i = 0; i < 10; i = i + 1) begin

      random_number = get_constrained_random(10, 50);

      $display("Random Number: %d", random_number);

    end

  end


  // Function to generate constrained random number between min and
max (inclusive)

  function [7:0] get_constrained_random;

    input integer min;

    input integer max;

    integer range;

    begin

      range = max - min + 1;

      get_constrained_random = min + $random(seed) % range;

    end

  endfunction

endmodule
```

2. How can you be sure that the constrained random stimulus has covered all the values in the range without repetition in a cyclic random fashion? Explain with an example.

To ensure that constrained random stimulus covers all values in the range without repetition in a cyclic fashion, you can shuffle the values in the range and then use them one by one.

```
module constrained_random_no_repetition;
  reg [7:0] values [0:40];  // Array to hold values 10 to 50
  reg [7:0] shuffled_values [0:40];  // Array to hold shuffled values
  integer i, j, temp;
  initial begin
    // Initialize the array with values from 10 to 50
    for (i = 0; i < 41; i = i + 1) begin
      values[i] = i + 10;
    end
    // Copy values to shuffled_values
    for (i = 0; i < 41; i = i + 1) begin
      shuffled_values[i] = values[i];
    end
    // Shuffle the array
    for (i = 0; i < 41; i = i + 1) begin
      j = $urandom % 41;
      temp = shuffled_values[i];
      shuffled_values[i] = shuffled_values[j];
      shuffled_values[j] = temp;
    end
    // Display the shuffled values
    for (i = 0; i < 41; i = i + 1) begin
      $display("Value: %d", shuffled_values[i]);
    end
  end
endmodule
```

3. Write a code to change the sequence of constrained random stimulus?

```
module change_sequence_random_stimulus;
  reg [7:0] stimulus_values[0:9];
  reg [7:0] randomized_values[0:9];
  reg [7:0] temp;
  integer i, j;
  initial begin
    // Initialize the array with values from 0 to 9
    for (i = 0; i < 10; i = i + 1) begin
      stimulus_values[i] = i;
    end
    // Shuffle the array
    for (i = 0; i < 10; i = i + 1) begin
      j = $random % 10;
      temp = stimulus_values[i];
      stimulus_values[i] = stimulus_values[j];
      stimulus_values[j] = temp;
    end
    // Copy shuffled values to randomized_values
    for (i = 0; i < 10; i = i + 1) begin
      randomized_values[i] = stimulus_values[i];
    end
    // Display the shuffled values
    for (i = 0; i < 10; i = i + 1) begin
      $display("Shuffled Value: %0d", randomized_values[i]);
    end
  end
endmodule
```

4. What is weighted random stimulus? Explain with an example.

Verilog does not have built-in support for weighted random stimulus like SystemVerilog, so you need to manually implement a weighted selection. Here's an example approach using Verilog:

```
module weighted_random_stimulus;
  reg [7:0] value;
  integer random_number;
  integer i;
  // Procedure to generate weighted random numbers
  function [7:0] weighted_random;
    input [7:0] weights[0:3];  // Array to hold weights
    integer sum_weights;
    integer threshold;
```

```
      integer cumulative_weight;
      integer rand_value;
      begin
        // Calculate sum of weights
        sum_weights = 0;
        for (i = 0; i < 4; i = i + 1) begin
          sum_weights = sum_weights + weights[i];
        end

        // Generate random number within sum of weights
        rand_value = $random % sum_weights;
        // Determine value based on weights
        cumulative_weight = 0;
        for (i = 0; i < 4; i = i + 1) begin
          cumulative_weight = cumulative_weight + weights[i];
          if (rand_value < cumulative_weight) begin
            weighted_random = i * 10;  // Example values: 0, 10, 20, 30
            return;
          end
        end
        weighted_random = 0;  // Default value
      end
    endfunction
    initial begin
      integer weights[0:3];
      weights[0] = 10;  // Weight for 0
      weights[1] = 20;  // Weight for 10
      weights[2] = 30;  // Weight for 20
      weights[3] = 40;  // Weight for 30
      // Generate and display weighted random numbers
      repeat (10) begin
        value = weighted_random(weights);
        $display("Weighted Random Value: %0d", value);
      end
    end
  endmodule
```

5. What metrics help in defining the completeness of the random simulations?

Metrics for defining completeness of random simulations include:

- Functional Coverage: Measures which functional aspects of the design have been exercised.
- Code Coverage: Includes line, toggle, FSM, and condition coverage to see which parts of the code have been executed.
- Assertion Coverage: Ensures that assertions are checked during simulation.
- Cross Coverage: Measures combinations of different variables or conditions.

6. What are some stimulus generation techniques when the stimulus is not reproducible using BFMs? Explain using Verilog examples.

When stimulus is not reproducible using Bus Functional Models (BFMs), techniques include:

- Directed Tests: Manually crafted tests for specific scenarios.
- Constrained Random Tests: Using constraints to generate valid random inputs.
- Parameterized Tests: Generating tests based on parameter values.

```
module stimulus_generation_example;

  reg clk;

  reg reset;

  reg [7:0] data;

  initial begin

    clk = 0;

    reset = 1;

    #10 reset = 0;

  end


  always #5 clk = ~clk;


  initial begin

    #20 data = $random;

    #20 data = $random;

    // Add more stimulus as required

  end
```

```
endmodule
```

7. What is SDF back-annotation, and how is it implemented in Verilog testbench?

Standard Delay Format (SDF) back-annotation is used to include timing information in simulations.

```
module sdf_back_annotation_example;

  // Include the instance of the design under test

  dut uut (

    // Connect ports

  );

  initial begin

    $sdf_annotate("path/to/sdf_file.sdf", uut);

    // Run simulation

  end

endmodule
```

8. What is the difference between unit delay and full timing simulations?

- Unit Delay Simulations: All gates are assumed to have unit delay (e.g., 1ns), which helps in quick functional verification.
- Full Timing Simulations: Use actual delays from the design, extracted from synthesis or post-layout timing analysis, to check timing correctness.

9. My gate simulation is not passing, and some tests hang. What are the key points to look for?

Key points to look for when gate simulations fail or tests hang:

- Clock Issues: Ensure the clock is correctly generated and connected.
- Reset Issues: Check if the design is correctly reset.
- Timing Violations: Look for setup/hold time violations.
- X-propagation: Check for uninitialized signals causing 'X' states.
- Mismatch between RTL and Gate-level: Ensure the gate-level netlist matches the RTL functionality.

10. Using a synchronous FSM approach, design a circuit that takes a single bit stream as an input at the pin in.

```
module pattern_detector(

  input clk,

  input reset,

  input in,
  output reg match
);
  typedef enum logic [2:0] {
    S0, S1, S2, S3, S4
  } state_t;

  state_t state, next_state;

  always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
      state <= S0;
      match <= 0;
    end else begin
      state <= next_state;
    end
  end
  always_comb begin
    next_state = state;
    match = 0;
    case (state)
      S0: if (in) next_state = S1;
      S1: if (!in) next_state = S2; else next_state = S1;
      S2: if (in) next_state = S3; else next_state = S0;
      S3: if (!in) next_state = S4; else next_state = S1;
      S4: if (in) begin
             next_state = S1;
             match = 1;
          end else next_state = S0;
    endcase
  end
endmodule
```

An output pin match is asserted high each time the pattern 10101 is detected. A reset pin initializes the circuit synchronously. Input pin clk is used to clock the circuit. Apply constrained randomized stimulus to this design for verification.