**Module: SV for Verification**

**Section:** Coverage **Task:** Coverage Groups & Coverpoints

**Task 3**

Coverage Groups & Coverpoints

* **Q1: What are coverpoints and bins? Explain with an example.**

Coverpoints and bins are key concepts in SystemVerilog's functional coverage, which helps verify how thoroughly the design has been tested. They are used to capture and measure the occurrence of specific events, signal values, or conditions during simulation.

* **Coverpoints:** A coverpoint is a specific variable or expression whose value you want to monitor during simulation. It defines what you want to track for coverage.
* **Bins:** Bins are used to categorize or group the values or ranges of values that a coverpoint can take. They help quantify how often certain conditions or values are hit during simulation.

### **Example:**

Let's say you have a signal **addr** in a memory design that can take values between 0 and 3. You want to track how often each value of **addr** occurs during simulation.

covergroup addr\_coverage;

coverpoint addr {

bins addr\_0 = {0}; // A bin for addr = 0

bins addr\_1 = {1}; // A bin for addr = 1

bins addr\_2 = {2}; // A bin for addr = 2

bins addr\_3 = {3}; // A bin for addr = 3

}

endgroup

initial begin

addr\_coverage cov = new; // Create an instance of the coverage group

// Simulate and sample values

repeat(10) begin

addr = $random % 4; // Generate random values for addr (0 to 3)

cov.sample(); // Sample the covergroup

end

end

### **Explanation:**

* During the simulation, every time the value of **addr** changes, the **sample()** method is called. It checks the current value of **addr** and places it into the appropriate bin (e.g., if **addr = 2**, the **addr\_2** bin gets incremented).
* At the end of the simulation, you can see which bins were hit and which weren't, giving you insight into whether all values of **addr** were tested.
* **Q2: What is the difference between ignore bins and illegal bins?**

| **Aspect** | **Ignore bins** | **Illegal bins** |
| --- | --- | --- |
| **Purpose** | Exclude values from coverage analysis | Flag invalid values that should never occur |
| **Behavior** | Values are ignored and not counted | Errors or failures reported if values occur |
| **Use Case** | Values are valid but not relevant | Values are considered invalid in the design |
| **Example Use** | Exclude redundant or unimportant values | Catch invalid or illegal design states |

#### **Ignore Bins Example:**

For instance, we want to collect coverage on addresses but don’t care about address 3.

covergroup my\_cov;

coverpoint addr {

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

ignore\_bins ignore\_vals = {3}; // Ignore addr = 3

}

endgroup

* **Result:** If **addr = 3** occurs, it's ignored in the coverage report.

#### 

#### **Illegal Bins Example:**

You want to ensure that **addr = 3** should never happen.

covergroup my\_cov;

coverpoint addr {

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

illegal\_bins illegal\_vals = {3}; // addr = 3 is illegal

}

endgroup

* **Result:** If **addr = 3** occurs, it raises an error because it’s in the illegal bin.
* **Q3: How can we write a coverpoint to look for transition coverage on an expression?**

In SystemVerilog, we can write a coverpoint to look for transition coverage on an expression by defining a coverpoint for that expression and specifying transitions using the **bins** keyword with the **=>** operator. Here's how you can do it:

### **Example:**

Suppose we want to track transitions of an expression **a + b** over time. We can define a coverpoint that captures transitions between different values of **a + b**.

covergroup transition\_cov;

coverpoint (a + b) {

bins trans\_1\_to\_2 = (1 => 2); // Tracks transitions from (a + b) = 1 to (a + b) = 2

bins trans\_2\_to\_3 = (2 => 3); // Tracks transitions from (a + b) = 2 to (a + b) = 3

bins all\_transitions[] = (1 => 2, 2 => 3, 3 => 1); // Tracks multiple transitions

}

endgroup

This will capture and report transitions of the expression **(a + b)** during simulation.

* **Q4: What does the following bin try to cover?**

covergroup test\_cg @(posedge clk);

coverpoint var\_a {

bin hit\_bin = { 3[\*4]};

}

endgroup

The given bin tries to cover the case where the value of **var\_a** is **3** for exactly 4 consecutive occurrences. This bin checks if the value of **var\_a** equals **3** for 4 consecutive clock cycles. If this condition is met during simulation, the bin will be considered as hit.

* **Q5: Write a SystemVerilog snippet to create a wildcard bin that captures all values greater than or equal to 10 in a simulation. Explain how you would use this wildcard bin in a simulation environment to analyze the results efficiently.**

covergroup cg @(posedge clk);

coverpoint var\_a {

wildcard bins values\_bin = {10:default}; // Covers all values >= 10

}

endgroup

**Usage in a Simulation Environment:**

In testbench, we would instantiate the covergroup and sample it at every clock cycle or based on relevant events in the simulation. After the simulation, you can analyze the results using coverage reports.

### **Example:**

module testbench;

logic clk;

logic [7:0] var\_a;

// Instantiate the covergroup

value\_cg cov\_instance = new();

// Clock generation

always #5 clk = ~clk;

// Simulation of variable changes and sampling

initial begin

clk = 0;

var\_a = 8'h05; // Value less than 10, not captured by the bin

#10 var\_a = 8'h0A; // Value 10, captured by the bin

#10 var\_a = 8'h12; // Value greater than 10, captured by the bin

#10 var\_a = 8'h20; // Value much greater than 10, captured by default wildcard

#10 $finish;

end

always @(posedge clk) begin

cov\_instance.sample(); // Sample the covergroup at every clock cycle

end

endmodule

* **Q6: What is Cross Coverage? When is it useful in verification?**

1. **Definition:** Cross coverage is defined using the **cross** construct in SystemVerilog, which allows you to create coverage points that track combinations of multiple cover groups.
2. **Usage:** It's useful in verifying that all possible scenarios involving multiple signals are tested. For example, if you have two control signals, you might want to check the coverage of all possible combinations of their states (e.g., both signals high, one high and one low, etc.).
3. **Benefits:**
   * **Identifies Gaps**
   * **Enhances Test Quality**
   * **Guides Test Generation**

### **When It's Useful:**

* **Complex Designs:** In designs with multiple interacting signals, where the behavior is dependent on the combination of these signals.
* **Functional Safety:** In safety-critical applications where specific signal combinations could lead to unsafe states.
* **Protocol Verification:** When verifying communication protocols where multiple signal interactions determine the protocol's state.
* **Q7: What can be wrong with the following coverage code?**

int var\_a;

covergroup test\_cg @(posedge clk);

cp\_a: coverpoint var\_a {

bins low = {0,1};

bins other[] = default;

}

endgroup

This coverage code has a couple of potential issues:

**Variable Declaration**: The variable var\_a should be declared as **logic** or **bit** instead of **int**. SystemVerilog uses **logic** or **bit** for signals that are used in covergroups.

**Covergroup Instantiation**: You need to instantiate the covergroup and call its **sample()** method to collect coverage data. Without instantiation and sampling, the covergroup won’t function as expected.

Here’s a revised version:

logic var\_a;

covergroup test\_cg @(posedge clk);

cp\_a: coverpoint var\_a {

bins low = {0,1};

bins other[] = default;

}

endgroup

// Instantiate and sample the covergroup

test\_cg cg\_instance = new();

always @(posedge clk) begin

cg\_instance.sample();

end

* **Q8: What is the difference between coverage per instance and per type? How do we control the same using coverage options?**

| **Aspect** | **Per Instance** | **Per Type** |
| --- | --- | --- |
| **Definition** | Each instance of a covergroup maintains its own coverage data. This means that every time you create a new instance of the covergroup, it tracks coverage independently. | Coverage data is aggregated across all instances of the covergroup. In this case, the coverage results are combined into a single report that reflects the overall behavior across all instances. |
| **Use Case** | This is useful when you want to observe coverage behavior for specific instances of a module or a component, allowing for detailed analysis of individual behaviors. | This is beneficial for understanding the overall functionality of a design without focusing on individual instances. It provides a higher-level view of how the design behaves under different conditions. |

You can control whether coverage is collected per instance or per type using the **options** clause in the covergroup declaration:

covergroup my\_cg @(posedge clk) option.per\_instance;

coverpoint var\_a {

bins low = {0, 1};

bins other[] = default;

}

endgroup

// For coverage per type

covergroup my\_cg\_type @(posedge clk) option.per\_type;

coverpoint var\_a {

bins low = {0, 1};

bins other[] = default;

}

endgroup