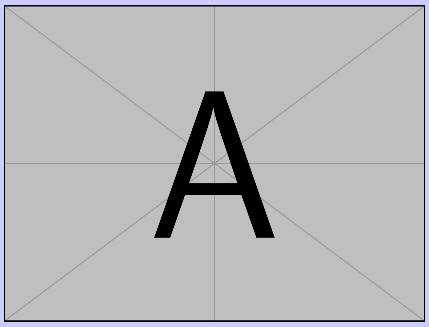
SystemVerilog Interview Mastery Guide

1000 Questions with Detailed Explanations



For VLSI Aspirants Targeting Top Product Companies

Includes Real Questions from:
Intel, AMD, Apple, Qualcomm, Synopsys, NVIDIA, and More

Prepared by:
KITTU K PATEL
ASIC Design & Verification Mentor — VERICORE

May 2025

Introduction

The SystemVerilog Interview Mastery Guide is designed to be a one-stop solution for aspirants preparing for digital verification and front-end roles in the VLSI industry. With increasing demand for ASIC and SoC professionals, top companies such as Intel, AMD, Synopsys, and Qualcomm are seeking engineers with a solid grasp on SystemVerilog and UVM fundamentals.

This guide compiles **1000 curated interview questions**, thoroughly explained with code, theory, and use-case context. The questions are organized by key topics such as:

- Randomization and Constraints
- Covergroups and Functional Coverage
- OOP in SV
- UVM Architecture and Phases
- TLM, Factory Overrides, Virtual Sequences
- Assertions and Formal Verification
- Scoreboards, Monitors, Drivers
- RAL (Register Abstraction Layer)
- Simulation Debug and Optimization

Whether you are a fresh graduate or an industry professional switching to DV roles, this handbook will help you confidently tackle both technical rounds and discussions that demand hands-on coding expertise.

How to Get into the VLSI Industry

Getting into the VLSI (Very Large Scale Integration) industry is both rewarding and challenging. Here's a step-by-step roadmap to guide your entry:

1. Build Strong Fundamentals

Ensure you have a good command over:

- Digital Electronics
- Computer Architecture
- CMOS Design
- Verilog/SystemVerilog

2. Choose a Path: Front-End or Back-End

- **Front-End:** RTL Design, Verification, UVM, SVA
- **Back-End:** Physical Design, STA, Layout, DRC/LVS

3. Get Skilled

Take up hands-on projects. Use open-source platforms like:

- EDA Playground, Makerchip (Front-End)
- OpenROAD, SKY130 PDK (Back-End)

4. Resume and LinkedIn Optimization

Highlight:

- Projects and Internships
- GitHub Repositories
- Certifications (Doulos, Maven Silicon, VLSI Academy)

5. Apply Strategically

Target product companies, EDA vendors, and VLSI service startups. Use referral networks and hackathons for visibility.

Topic 1: Randomization and Constraints

Q1. How do you generate a random variable between 10 and 50, restricted to even values only?

Answer: You can use SystemVerilog constraints inside a class with the ‘rand‘ keyword. Here is the solution:

```
class even_rand;
  rand bit [7:0] val;

  constraint even_c {
    val inside {[10:50]};
    val % 2 == 0;
  }
endclass

module tb;
  even_rand obj = new();
  initial begin
    repeat(5) begin
      void'(obj.randomize());
      $display("Even Value = %0d", obj.val);
    end
  end
endmodule
```

Explanation:

- ‘rand‘ declares the variable as randomizable.
- The ‘inside‘ operator constrains the range.
- ‘

Q2. How can you generate a unique array of 5 elements between 1 and 20?

Answer:

```
class unique_array;
  rand bit [4:0] arr[5];

  constraint c_range { foreach (arr[i]) arr[i] inside {[1:20]}; }
  constraint c_unique { unique {arr}; }
endclass

module tb;
  unique_array u = new();
  initial begin
    if (u.randomize())
      $display("Unique Array = %p", u.arr);
  end
endmodule
```

Explanation:

- ‘unique‘ keyword ensures all elements are different.
- ‘foreach‘ applies the range constraint to each element.

Q3. How to constrain a variable to only odd numbers from 5 to 15 but exclude consecutive odd values?

Answer:

```
class my_rand;
  rand bit [3:0] a;

  constraint c1 {
    a inside {[5:15]};
    a % 2 == 1;
    !(a == 7 || a == 9 || a == 11 || a == 13); // no consecutive odds
  }
endclass

module tb;
  my_rand obj = new();
  initial begin
```

```
    repeat(10) begin
      void'(obj.randomize());
      $display("a = %0d", obj.a);
    end
  end
endmodule
```

Explanation: The constraint excludes specific consecutive values to comply with the rule. This is a common interview challenge to test your understanding of exclusion logic in constraints.

Q4. Constrain two variables such that one is twice the other and their sum is less than 50.

Answer:

```
class rel_constraint;
  rand bit [7:0] a, b;

  constraint twice {
    a == 2 * b;
    a + b < 50;
  }
endclass

module tb;
  rel_constraint obj = new();
  initial begin
    repeat (5) begin
      void'(obj.randomize());
      $display("a=%0d, b=%0d, sum=%0d", obj.a, obj.b, obj.a + obj.b);
    end
  end
endmodule
```

Explanation: This type of relation constraint is crucial in protocol-level modeling when fields are mathematically dependent.

Q5. How do you write a constraint for sorting a 4-element array in ascending order?

Answer:

```
class sort_array;
  rand bit [7:0] arr[4];

  constraint c_sorted {
    arr[0] <= arr[1];
    arr[1] <= arr[2];
    arr[2] <= arr[3];
  }
endclass

module tb;
  sort_array s = new();
  initial begin
    if (s.randomize())
      $display("Sorted Array = %p", s.arr);
  end
endmodule
```

Explanation: Constraints are applied pairwise to enforce the sorting order. This is often used in test scenarios to ensure stimulus validity.

Q6. Generate a queue of size 5 containing even numbers only.

Answer:

```
class even_q;
  rand bit [7:0] q[$];

  constraint c1 {
    q.size() == 5;
    foreach (q[i]) q[i] % 2 == 0;
  }
endclass
```

```
module tb;
  even_q obj = new();
  initial begin
    if (obj.randomize())
      $display("Even Queue: %p", obj.q);
  end
endmodule
```

Explanation: Queues are dynamic arrays. The ‘foreach‘ loop ensures that each element satisfies the modulus constraint.

Q7. Randomize a 2D array in SystemVerilog such that each row is sorted.

Answer:

```
class array2D;
  rand bit [7:0] arr[3][4];

  constraint row_sorted {
    foreach (arr[i]) {
      foreach (arr[i,j]) if (j < 3)
        arr[i][j] <= arr[i][j+1];
    }
  }
endclass

module tb;
  array2D obj = new();
  initial begin
    if (obj.randomize())
      $display("2D Array = %p", obj.arr);
  end
endmodule
```

Explanation: Nested ‘foreach‘ constraints can be used to control the properties of multidimensional arrays. This ensures logical or protocol-based ordering.

Q8. Write a constraint such that the sum of array elements equals 100.

Answer:

```
class array_sum;
  rand bit [7:0] arr[5];

  constraint sum_c {
    arr.sum() == 100;
  }
endclass

module tb;
  array_sum obj = new();
  initial begin
    repeat (5) begin
      void'(obj.randomize());
      $display("Array = %p, Sum = %0d", obj.arr, obj.arr.sum());
    end
  end
endmodule
```

Explanation: '.sum()' is a built-in method used to calculate the total value of all elements in an array. This is especially useful for checksum generation and protocol timing scenarios.

Q9. Write a constraint such that each queue element is double the previous one.

Answer:

```
class exp_queue;
  rand bit [7:0] q[5];

  constraint exp {
    q[0] == 1;
    foreach(q[i]) if (i > 0) q[i] == 2 * q[i-1];
  }
```

```
endclass

module tb;
  exp_queue obj = new();
  initial begin
    if (obj.randomize())
      $display("Exponential Queue: %p", obj.q);
  end
endmodule
```

Explanation: This models geometric progressions useful in timing burst patterns or protocol test sequences.

Q10. Generate a queue from 0 to 9 with all unique values.

Answer:

```
class unique_q;
  rand bit [3:0] q[$];

  constraint c1 {
    q.size() == 10;
    foreach (q[i], q[j]) i != j -> q[i] != q[j];
    foreach (q[i]) q[i] inside {[0:9]};
  }
endclass

module tb;
  unique_q obj = new();
  initial begin
    if (obj.randomize())
      $display("Unique Queue: %p", obj.q);
  end
endmodule
```

Explanation: Constraints are used to ensure range and uniqueness, commonly used in generating randomized IDs or token sets.

Q11. Write a constraint to ensure sum of two variables is always even.

Answer:

```
class sum_even;
  rand bit [3:0] a, b;

  constraint even_sum { (a + b) % 2 == 0; }
endclass

module tb;
  sum_even obj = new();
  initial begin
    repeat (5) begin
      void'(obj.randomize());
      $display("a=%0d, b=%0d, sum=%0d", obj.a, obj.b, obj.a + obj.b);
    end
  end
endmodule
```

Explanation: Simple mathematical relations are a staple of DV testbench modeling. You'll often see parity or checksum conditions represented like this.

Q12. Create a constraint where one variable is always less than another, both in a given range.

Answer:

```
class less_than;
  rand bit [7:0] a, b;

  constraint limits { a inside {[10:50]}; b inside {[20:60]}; }
  constraint order { a < b; }
endclass

module tb;
  less_than obj = new();
  initial begin
```

```
    repeat (5) begin
      void'(obj.randomize());
      $display("a = %0d, b = %0d", obj.a, obj.b);
    end
  end
endmodule
```

Explanation: This is commonly used in timer delay settings or validation ranges where a dependent relationship exists.

Q13. How do you create a simple covergroup that captures a 4-bit data value into two bins: 0–7 (low) and 8–15 (high)?

Answer:

```
class transaction;
  rand bit [3:0] data;

  covergroup cg;
    option.per_instance = 1;
    DATA: coverpoint data {
      bins low = {[0:7]};
      bins high = {[8:15]};
    }
  endgroup

  function new();
    cg = new();
  endfunction

  function void sample_data();
    cg.sample();
  endfunction
endclass

module tb;
  transaction t = new();
  initial begin
```

```
    repeat(10) begin
      void'(t.randomize());
      t.sample_data();
      $display("Data = %0d", t.data);
    end
  end
endmodule
```

Explanation: Functional coverage uses covergroups and coverpoints to measure how thoroughly a stimulus space is exercised.

Q14. Create a covergroup to track transitions from IDLE to ACTIVE to DONE.

Answer:

```
typedef enum {IDLE, ACTIVE, DONE} state_t;
state_t state;

covergroup cg @(posedge clk);
  coverpoint state {
    bins full_path = (IDLE => ACTIVE => DONE);
  }
endgroup
```

Explanation: Transition bins are extremely useful in verifying protocol sequences and FSM transitions.

Q15. How do you sample a covergroup inside a class for protocol coverage?

Answer:

```
class bus_tx;
  rand bit [3:0] burst_len;

  covergroup burst_cg;
    coverpoint burst_len {
      bins single = {1};
      bins short  = {[2:4]};
      bins long   = {[5:15]};
    }
  endgroup

  function new();
    burst_cg = new();
  endfunction
endclass

module tb;
  bus_tx tx = new();
  initial begin
    repeat (10) begin
      void'(tx.randomize());
      tx.burst_cg.sample();
      $display("Burst Length = %0d", tx.burst_len);
    end
  end
endmodule
```

Explanation: This models burst length patterns, commonly required in protocol-level coverage models such as AXI or AHB.

Q16. How do you model cross coverage between two coverpoints in SystemVerilog?

Answer:

```
class cross_cov;
  rand bit [3:0] addr;
  rand bit [1:0] mode;

  covergroup cg;
    ADDR: coverpoint addr {
      bins low  = {[0:7]};
      bins high = {[8:15]};
    }
    MODE: coverpoint mode {
      bins rw = {0, 1};
      bins burst = {2, 3};
    }
    ADDR_MODE: cross ADDR, MODE;
  endgroup

  function new();
    cg = new();
  endfunction
endclass
```

Explanation: Cross coverage helps ensure that all combinations of two signals are exercised, which is vital in protocol verification.

Q17. What is ‘option.per$_{i}nstance‘andwhyisitimportant$?

Answer:
‘option.per_instance = 1;‘ enables each instance of a class to have its own copy of the coverage data. Without it, all instances share a single covergroup result.

Use-case Example:

```
covergroup cg;
  option.per_instance = 1;
  coverpoint opcode;
endgroup
```

When to Use:

- In class-based models
- When collecting coverage from multiple environments or testcases

Q18. How do you ignore specific bins in a coverpoint?

Answer:
Use the ‘ignore_bins‘ construct.

```
covergroup cg;
  coverpoint opcode {
    bins valid = {[0:15]};
    ignore_bins reserved = {6, 7};
  }
endgroup
```

Explanation: Ignored bins are excluded from coverage goals and are often used to filter out illegal or unused values.

Q19. How do you create automatic sampling on an event in a covergroup?

Answer:

```
covergroup auto_cg @(posedge clk);
  coverpoint state;
endgroup
```

Explanation: This allows coverage to be sampled automatically on clock edges or signal transitions, removing the need to call '.sample()' manually.

Q20. Write a constraint to randomize values divisible by 3 and in range 30 to 90.

Answer:

```
class div_by_3;
  rand bit [7:0] val;

  constraint c1 {
    val inside {[30:90]};
    val % 3 == 0;
  }
endclass

module tb;
  div_by_3 obj = new();
  initial begin
    repeat (10) begin
      void'(obj.randomize());
      $display("Value divisible by 3: %0d", obj.val);
    end
  end
endmodule
```

Explanation: The modulus operator is used to check divisibility. Combined with the ‘inside‘ operator, this restricts values to a specific range and multiple.

Q21. Generate a 3-element array such that the sum is even and elements are between 1 and 10.

Answer:

```
class sum_even_array;
  rand bit [3:0] arr[3];

  constraint range_c { foreach(arr[i]) arr[i] inside {[1:10]}; }
  constraint even_sum_c { arr.sum() % 2 == 0; }
endclass

module tb;
  sum_even_array obj = new();
  initial begin
    repeat (5) begin
      void'(obj.randomize());
      $display("Array = %p, Sum = %0d", obj.arr, obj.arr.sum());
    end
  end
endmodule
```

Explanation: The ‘.sum()‘ method calculates the array sum, and ‘

Q22. Constrain a variable to avoid a set of specific values.

Answer:

```
class avoid_vals;
  rand bit [7:0] val;

  constraint c1 {
    !(val inside {13, 27, 55});
    val inside {[10:60]};
  }
endclass

module tb;
  avoid_vals obj = new();
  initial begin
    repeat (5) begin
      void'(obj.randomize());
      $display("Value = %0d", obj.val);
    end
  end
endmodule
```

Explanation: Use '!inside' to exclude specific illegal values. Common in testbench scenarios where illegal or reserved values must be skipped.

Q31. What are the types of coverage in SystemVerilog?

Answer: SystemVerilog provides three main types of coverage:

- **Code Coverage:** Measures which parts of the code were executed (e.g., line, toggle, FSM).
- **Functional Coverage:** Measures whether all required functionality has been tested using coverage points.
- **Assertions Coverage:** Tracks whether assertions were triggered and whether pass/fail scenarios were covered.

Q32. What is functional coverage and why is it important?

Answer: Functional coverage is a user-defined metric that checks whether specific functional scenarios have been exercised during simulation. It is important because:

- It validates whether the design is tested thoroughly.
- It helps identify missing corner cases.
- It is used in coverage-driven verification.

Q33. What are covergroups in SystemVerilog?

Answer: A `covergroup` is a construct used to define functional coverage points. It includes:

- **coverpoints:** Variables or expressions to be monitored.
- **cross coverage:** Combination of two or more coverpoints.

Example:

```
covergroup cg;
  coverpoint op;
  coverpoint a;
  cross op, a;
endgroup
```

Q34. Explain the difference between coverpoint and cross coverage.

Answer:

- **Coverpoint:** Tracks the value distribution of a single variable.
- **Cross Coverage:** Measures combinations of values from two or more coverpoints to verify interaction between variables.

Q35. What is the significance of `bins` in coverage?

Answer: Bins group values into ranges or specific values for coverage tracking.

- **Implicit bins:** Created automatically.
- **Explicit bins:** User-defined for precise control.

Example:

```
coverpoint val {
  bins low = {[0:3]};
  bins mid = {[4:7]};
  bins high = {[8:15]};
}
```

Q36. What are coverage options in SystemVerilog?

Answer: Coverage options control how coverage data is collected or reported. Examples include:

- `option.per_instance` – Collects coverage per object instance.
- `option.weight` – Assigns weight to influence total coverage score.
- `option.comment` – Provides documentation.
- `option.name` – Customizes instance name in reports.

Q37. How to disable a coverpoint or covergroup dynamically?

Answer: You can disable a coverpoint or covergroup by:

- Setting `cg.option.weight = 0;` to ignore it in coverage.
- Not calling the `sample()` method when coverage is not required.

Q38. What is the use of `sample()` method in covergroups?

Answer: The `sample()` method captures current values of the coverpoints. It must be called explicitly if the covergroup is not declared with an `@` event. **Example:**

```
cg_inst.sample();
```

Q39. How do you use `illegal_bins` and `ignore_bins`?

Answer:

- `illegal_bins` – Flags undesired or invalid values.
- `ignore_bins` – Skips values in coverage calculation.

Example:

```
coverpoint mode {
  bins valid = {[0:3]};
  ignore_bins skip = {[4:5]};
  illegal_bins err = {[6:7]};
}
```

Q40. How can cross coverage be customized?

Answer: Cross coverage can be customized using:

- `bins` – To define valid combinations.
- `ignore_bins` or `illegal_bins` – To skip or flag combinations.
- `intersect` and `with` – To filter combinations with expressions.

Example:

```
cross a, b {
  ignore_bins ignore_ab = binsof(a) intersect {1} && binsof(b) intersect {3};
}
```

Q41. What is transition coverage in SystemVerilog?

Answer: Transition coverage tracks sequences of values a variable goes through. It helps ensure that legal state transitions occur. **Syntax Example:**

```
coverpoint state {
  bins s_trans[] = (S0 => S1 => S2);
}
```

Q42. How do you use wildcard bins in transition coverage?

Answer: Wildcards allow flexibility in defining transitions without exact values. **Example:**

```
bins wild[] = (1 => [*] => 3);
```

This matches a transition from 1 to 3 with any number of intermediate states.

Q43. What is the difference between `bins` and `bins[]` in transitions?

Answer:

- `bins name = (a => b);` defines a single transition.
- `bins name[] = (a => b => c);` creates a series of bins to capture each possible transition in the chain.

Q44. How can you collect coverage conditionally?

Answer: You can use `iff` clauses to conditionally enable coverage sampling. **Example:**

```
coverpoint addr iff (enable == 1);
```

This means `addr` will be sampled only if `enable` is 1.

Q45. How do you use coverage callbacks for advanced customization?

Answer: SystemVerilog does not directly support callbacks, but you can simulate similar behavior using class-based infrastructure and override `sample()` with custom logic before calling the actual sample.

Q46. What is `covergroup with` construct and how is it used?

Answer: `with` allows filtering or constraints on bins and cross coverage. **Example:**

```
cross a, b {
  bins cross_ab = binsof(a) intersect {1,2} &&
                  binsof(b) with (item < 5);
}
```

Q47. How do you merge coverage from multiple simulations?

Answer: Most simulators allow merging of multiple coverage databases using commands like:

- `vcover merge` (ModelSim/Questa)
- `urg -merge` (VCS)

This helps accumulate coverage over multiple test runs.

Q48. How to exclude certain bins or values from coverage reports?

Answer: Use `ignore_bins` to exclude values from affecting coverage percentage. **Example:**

```
ignore_bins idle_ignore = {[0:1]};
```

Q49. What are coverage directives and how are they used?

Answer: Directives are tool-specific pragmas used to control code coverage metrics. **Examples:**

- `// coverage off` – Turns off coverage for a block.
- `// coverage on` – Resumes coverage.

Q50. Can covergroups be parameterized or templated?

Answer: No, SystemVerilog does not support parameterized covergroups directly. However, you can achieve similar behavior by using arguments in the constructor to initialize different configurations. **Example:**

```
class cov;
  covergroup cg(int limit);
    coverpoint val {
      bins range[] = {[0:limit]};
    }
  endgroup
endclass
```

Q51. What is coverage-driven verification (CDV)?

Answer: CDV is a methodology where stimulus generation and verification effort are guided by coverage metrics. The aim is to maximize coverage efficiently, reducing redundant tests and improving quality.

Q52. How is functional coverage integrated with UVM?

Answer: In UVM, functional coverage is typically integrated using covergroups defined in sequence items, monitors, or scoreboards. The `uvm_subscriber` and `analysis_port` mechanism is commonly used to trigger coverage sampling.

Q53. What is the purpose of using `uvm_coverage` classes in UVM?

Answer: These classes help organize and encapsulate covergroups. They improve modularity and promote reuse by associating covergroups with the UVM testbench architecture.

Q54. How do you sample covergroups in UVM monitor or subscriber?

Answer: You call the `sample()` method after extracting or observing data. **Example:**

```
function void write(my_transaction tr);
  cov_inst.cp_inst.sample();
endfunction
```

Q55. Can coverpoints be triggered manually or automatically?

Answer: Coverpoints are always triggered manually using the `sample()` method. Unlike code coverage, functional coverage does not trigger automatically.

Q56. What are automatic bins and when are they useful?

Answer: Automatic bins are generated implicitly when a coverpoint range is specified without user-defined bins. They're useful for quickly covering wide ranges during early verification. **Example:**

```
coverpoint addr; // automatic bins based on type width
```

Q57. How do you measure cross coverage between signals from different interfaces?

Answer: You can create a central covergroup that includes coverpoints from both interfaces or pass values from both interfaces to a transaction class and cover them there.

Q58. What is the difference between `ignore_bins` and `illegal_bins`?

Answer:

- `ignore_bins` excludes values from coverage without raising any errors.
- `illegal_bins` raises a runtime error if those values are encountered, and they count as test failures.

Q59. How can you reduce simulation time while still collecting meaningful coverage?

Answer:

- Use targeted constraints to hit specific coverage goals.
- Use `ignore_bins` to skip irrelevant cases.
- Prioritize corner cases using weighted randomization.

Q60. How do you interpret functional coverage reports?

Answer: Coverage reports list:

- Covergroups and coverpoints with hit counts.
- Percentage covered.
- Missing bins and transitions.

Analyzing this helps guide regression planning and identify unverified behavior.

Q61. What is an assertion in SystemVerilog and why is it used?

Answer: An assertion in SystemVerilog is a construct used to validate the behavior of a design by checking for expected conditions at runtime. Assertions help catch design bugs early by ensuring that certain conditions hold true during simulation. They improve design reliability and simplify debugging by localizing violations of intended behavior.

Q62. What are the two types of assertions in SystemVerilog? Explain each.

Answer: SystemVerilog supports two main types of assertions:

- **Immediate Assertions**: These are evaluated immediately when the statement is executed. They are typically used in procedural code like initial or always blocks.
- **Concurrent Assertions**: These evaluate properties over time and are used to monitor behavior across clock cycles. They are written using temporal expressions.

Q63. What are the primary methods used with concurrent assertions?

Answer: Concurrent assertions use the following keywords/methods:

- **assert property**: Checks if a property holds.
- **assume property**: Used in formal verification; assumes the property holds.
- **cover property**: Used to ensure that a property occurs during simulation for functional coverage.

Q64. What is the significance of the "disable iff" construct in assertions?

Answer: The `disable iff` clause is used in concurrent assertions to disable or ignore the evaluation of the assertion when a certain condition is true. It's typically used to avoid false failures during reset or when specific invalid conditions are present.

```
assert property (@(posedge clk) disable iff (reset) (req |=> ack));
```

In this example, the assertion is not evaluated when `reset` is high.

Q65. What are the benefits of using assertions in a verification environment?

Answer: Assertions provide several benefits:

- Early detection of design bugs.
- Better documentation of design intent.
- Reduced debugging time.
- Enhanced readability and maintainability.
- Enables formal verification and simulation coverage tracking.

Q66. Write a concurrent assertion to check that whenever 'req' goes high, 'ack' should go high in the next cycle.

Answer:

```
property req_ack;
  @(posedge clk) req |=> ack;
endproperty

assert property (req_ack);
```

This assertion checks that if 'req' is high on one clock cycle, 'ack' must be high on the next clock cycle.

Q67. How do you assert that a signal 'data_valid' remains high for 3 consecutive clock cycles once it becomes high?

Answer:

```
property data_valid_stable;
  @(posedge clk) data_valid |=> data_valid [*2];
endproperty

assert property (data_valid_stable);
```

The repetition operator ‘[*2]‘ checks that ‘data_valid‘ remains high for 2 more cycles after the initial high (total of 3).

Q68. Write an assertion to ensure that a signal ‘write‘ is not active during reset.

Answer:

```
property no_write_on_reset;
  @(posedge clk) disable iff (reset) !write;
endproperty

assert property (no_write_on_reset);
```

The assertion ensures that ‘write‘ is not asserted when reset is active.

Q69. How can you use the ‘cover property‘ to check that a transaction of ‘start ¿ done‘ happens at least once during simulation?

Answer:

```
property start_done_transaction;
  @(posedge clk) start |=> done;
endproperty

cover property (start_done_transaction);
```

The ‘cover property‘ is used for functional coverage to ensure the ‘start -¿ done‘ scenario occurs during simulation.

Q70. Write an assertion that checks a burst transfer of 4 cycles starting from ‘burst_start‘ with ‘data_valid‘ high in all cycles.

Answer:

```
property burst_transfer;
  @(posedge clk) burst_start |=> data_valid [*4];
endproperty

assert property (burst_transfer);
```

This checks that after ‘burst_start‘, ‘data_valid‘ remains high for 4 consecutive cycles, indicating a proper burst transfer.

Q76. Explain how the ‘disable iff‘ clause works in assertions and provide an example.

Answer:

The ‘disable iff‘ clause disables the assertion when its condition is true, effectively masking the assertion temporarily.

Example:

```
property p;
  @(posedge clk) disable iff (reset) (a |=> b);
endproperty

assert property (p);
```

Here, the assertion is disabled (ignored) whenever ‘reset‘ is high.

Q77. What is the difference between immediate and concurrent assertions? Provide examples.

Answer:

- **Immediate assertions** execute instantly at the point in procedural code. - **Concurrent assertions** run continuously alongside simulation time, checking temporal properties.

Examples:

Immediate assertion:

```
always @(posedge clk) begin
  assert (a == b) else $error("Mismatch");
end
```

Concurrent assertion:

```
property p;
  @(posedge clk) a |=> b;
endproperty

assert property (p);
```

Q78. How can you parameterize assertions in SystemVerilog? Provide a simple example.

Answer:

Assertions can be parameterized using ‘parameter‘ to make them reusable with different conditions.

Example:

```
parameter int WIDTH = 8;

property p #(parameter int W = WIDTH);
  @(posedge clk) (data == W) |=> (valid == 1);
endproperty

assert property (p#(16));
```

Q79. Describe the use of ‘cover property‘ in assertions. How is it different from ‘assert property‘?

Answer:

- ‘assert property‘ checks that a property holds true; simulation reports failure on violation. - ‘cover property‘ tracks if a property ever becomes true during simulation, useful for coverage analysis.

Example:

```
cover property (@(posedge clk) a ##1 b);
```

This records coverage when ‘a‘ is followed by ‘b‘ after 1 cycle.

Q80. What is a temporal implication (‘—=¿‘) in SVA, and how does it differ from logical implication (‘-¿‘)?

Answer:

- Temporal implication (‘—=¿‘) means ”if antecedent sequence happens now, then consequent sequence must happen starting next clock cycle.” - Logical implication (‘-¿‘) is a combinational operator without timing semantics.

Example:

```
property p;
  @(posedge clk) (a ##1 b) |=> (c ##1 d);
endproperty
```

Here, if ‘a 1 b‘ occurs, then ‘c 1 d‘ must follow in the subsequent cycles.

Logical implication example:

```
assert (a -> b); // immediate logic check
```

No timing is involved in logical implication.

Q81: What is Object-Oriented Programming (OOP) in SystemVerilog?

Answer:

OOP in SystemVerilog is a programming paradigm that uses objects and classes to model real-world entities and concepts. It supports encapsulation, inheritance, and polymorphism to improve code reuse, modularity, and maintainability in verification environments.

Q82: What are the main features of OOP in SystemVerilog?

Answer:
The main features are:
- Encapsulation: Bundling data and methods in classes.
- Inheritance: Deriving new classes from existing ones to reuse and extend functionality.
- Polymorphism: Ability to call methods of derived classes through base class handles (dynamic dispatch).
- Abstraction: Hiding implementation details behind interfaces and classes.

Q83: What is a class in SystemVerilog? How do you define it?

Answer:
A class is a blueprint for creating objects containing data (variables) and behavior (methods). It is defined using the `class` keyword.
Example:

```
class Packet;
  int data;
  function void display();
    $display("Data = %0d", data);
  endfunction
endclass
```

Q84: How do you create an object from a class in SystemVerilog?

Answer:
You create an object using the `new` operator on a class handle.
Example:

```
Packet pkt;
pkt = new();
pkt.data = 10;
pkt.display();
```

Q85: What is a constructor in SystemVerilog classes?

Answer:
A constructor is a special function named `new` that initializes the object at creation. It can take arguments to set initial values.
Example:

```
class Packet;
  int data;
  function new(int d);
    data = d;
  endfunction
endclass
```

Q86: What is inheritance in SystemVerilog? How is it implemented?

Answer:
Inheritance allows a class (derived class) to acquire properties and methods of another class (base class). Implemented using the `extends` keyword.
Example:

```
class BaseClass;
  function void show();
    $display("Base class");
  endfunction
endclass

class DerivedClass extends BaseClass;
  function void show();
    $display("Derived class");
  endfunction
endclass
```

Q87: What is polymorphism in SystemVerilog?

Answer:
Polymorphism allows methods to behave differently depending on the object's actual derived type, even if accessed through a base class handle. Achieved using virtual methods and dynamic dispatch.

Q88: How do you declare and use virtual methods?

Answer:
Declare a method with the `virtual` keyword in the base class. Derived classes override it. When called via a base class handle, the derived version executes at runtime.
Example:

```
class BaseClass;
  virtual function void display();
    $display("Base");
  endfunction
endclass

class DerivedClass extends BaseClass;
  function void display();
    $display("Derived");
  endfunction
endclass

BaseClass obj;
obj = new DerivedClass();
obj.display(); // Prints "Derived"
```

Q89: What are class handles and how do they differ from variables?

Answer:
Class handles are pointers/references to objects. Variables hold actual data. Class handles allow dynamic memory allocation, sharing, and polymorphism.

Q90: How is memory managed in SystemVerilog OOP?

Answer:
Objects are created dynamically using `new`. The simulator automatically garbage collects unreferenced objects, but explicit `delete` can be used to free objects sooner.