FUNCTIONAL VERIFICATION

INTERVIEW QUESTIONS

PART 1

JAIRAJ MIRASHI
DESIGN VERIFICATION ENGINEER

1. Explain how to inject a CRC error in a packet which has just data and CRC fields.

To inject a CRC error in a packet that contains only data and CRC fields, follow the steps below:

- Modify the CRC value: The CRC error injection is achieved by altering the CRC value in the packet.
- Avoid data modification: It is important to note that modifying the data itself to inject a CRC error may lead to the new modified packet having the same CRC value as the original packet.
- Consider the packet structure: Let's consider a packet with a total length of 5 bits, consisting of 3 bits of data and 2 bits of CRC value.
- Analyze possible combinations: With this configuration, there will be a total of 8 different packets that can be formed by varying the data field. However, since there are only 4 possible CRC values (2 bits), multiple data values may result in the same correct CRC value.
- Change the CRC value: To inject a CRC error for a specific data field, modify the CRC value associated with that data field. By altering the CRC value, a CRC error will be introduced in the packet.

2. How do you know when verification is completed?

Verification is considered complete when all points in the test plan have been thoroughly verified and validated, ensuring the system meets the specified requirements.

3. How to avoid race condition between Testbench and DUT?

To avoid race conditions between the Testbench and DUT (Device Under Test), the following guidelines can be followed:

- In VHDL:
- Clock Phase Difference: Ensure that the clock provided to the DUT and Testbench has a phase difference. This helps in synchronizing their operations and avoids conflicts.
- Clock Edge Sensitivity: Design the DUT to operate on the positive edge of the clock, while the Testbench should operate on the negative edge. This separation of clock edges prevents simultaneous access and potential race conditions.
- Non-Blocking Statements: Drive the output pins of both the Testbench and DUT using non-blocking assignments. This ensures proper synchronization and prevents conflicts during signal assignments.

- In SV (SystemVerilog):
- Apply the techniques from VHDL.
- Utilize Clocking Blocks: Implement clocking blocks to define the timing and synchronization aspects of signals and interfaces. Clocking blocks provide a structured methodology for handling clock signals, minimizing race conditions.
- Program Blocks: Employ program blocks to encapsulate and manage complex verification sequences. Program blocks allow for efficient control flow and synchronization between the Testbench and DUT.

4. What is a semaphore?

A semaphore is a synchronization mechanism used in concurrent programming. Conceptually, a semaphore can be thought of as a bucket that holds a fixed number of keys. Processes or threads that need to proceed must acquire a key from the bucket before they can continue execution. If a process requires a key and none are available, it must wait until a key is returned to the bucket by another process. Semaphores are commonly used for mutual exclusion, controlling access to shared resources, and basic synchronization.

5. What is the need of regression?

Regression serves the following purposes:

- Handling changes: Changes in the RTL (redevelopment, enhancement, or bug fixes) may cause existing functionality to break. Regression testing ensures that previously verified features continue to function correctly after these changes.
- Scenario generation: Regression testing allows for the creation of new scenarios by providing different seeds to the randomization engine. This helps explore different test cases and uncover potential issues or corner cases that may not have been covered during initial testing.

By incorporating regression testing, developers and testers can ensure that the system remains stable, reliable, and functions as intended even after modifications or the introduction of new scenarios.

6. What is randomization?

Randomization is a technique used in verification to generate scenarios in a random fashion. Since it is not possible to list out every possible real-time scenario while verifying a Device Under Test (DUT), randomization helps generate scenarios based on the specifications provided. By introducing randomness, different test cases can be generated, enabling thorough testing of the DUT. For example, in SystemVerilog, the expression {$random()} % 10 can be used to generate packets of random lengths.

7. What is the significance of seed in randomization?

The seed is a crucial factor in randomization. It is used to initialize the random number generator and determines the sequence of random numbers generated during simulation. By providing the same seed, the same sequence of random numbers can be reproduced, ensuring consistency and repeatability in simulation results. In regression testing, using the simulation start time as the seed allows for the generation of different stimuli for each regression run, leading to diverse test scenarios and improved coverage

8. What is the difference between code coverage and functional coverage?

Code coverage and functional coverage are both important in verification. Code coverage provides information about how many lines, expressions, and branches have been executed in the code. It is collected by simulation tools and helps identify untested areas. Functional coverage, on the other hand, is defined by the user and focuses on verifying specific functionalities of the design. Users define coverage points for the functions to be covered in the DUT, and it is under user control.

9. If code coverage is 100% and functional coverage is not, what does it mean?

If code coverage is 100% while functional coverage is not, it indicates that every line, expression, and branch of the code has been executed during the verification process. However, despite achieving full code coverage, there may still be untested scenarios or functionalities in the design. This discrepancy highlights the importance of functional coverage, which focuses on verifying specific functionalities rather than just the code itself.

To address this gap, engineers and testers need to design and execute directed test cases that specifically target the untested scenarios or functionalities. By doing so, they can ensure comprehensive coverage of the design's intended behaviors and functionalities. The combination of achieving both full code coverage and comprehensive functional coverage is crucial for a thorough and reliable verification process.

10. If functional coverage is 100% and code coverage is not, what does it mean?

If functional coverage is 100% while code coverage is not, it suggests that all intended functionalities have been thoroughly verified, but certain parts of the code have not been fully exercised. This may be due to issues in the test environment, unused code blocks, or miscommunication. Achieving comprehensive code coverage is important to ensure thorough testing and minimize the risk of hidden bugs or errors.

11. What is the difference between a passive monitor and an active monitor?

In a simulation environment, monitors play a crucial role in reporting protocol violations and capturing transaction information. Monitors can be categorized into two types: passive and active.

A passive monitor is designed to observe and analyze signals without driving any of its own signals. It acts as an observer and records the inputs provided to it.

On the other hand, an active monitor, sometimes referred to as a receiver, not only observes the signals but also has the ability to drive specific signals in the Device Under Test (DUT). It actively participates in the simulation by driving signals and capturing the responses, enabling more comprehensive monitoring and analysis.

Both passive and active monitors are valuable in capturing and processing transaction-level information. They convert the design's state and outputs into a higher-level transaction abstraction, which can be stored in a scoreboard database for further analysis.

12. Under what condition should simulation end in a simulation environment?

Simulation can be terminated based on different conditions depending on the requirements and objectives of the simulation. Some common conditions for ending simulation include:

- Packet count match: The simulation may end when a specific number of packets have been processed or generated, achieving the desired coverage.
- Error occurrence: If an error is detected during the simulation, it can trigger the termination of the simulation process. This helps identify and investigate issues promptly.
- Error count threshold: Simulation can be set to end when the number of errors reaches a predefined threshold. This allows for control over the maximum number of errors tolerated during the verification process.
- Interface idle: If there is no activity on a specific interface or bus for a certain period, the simulation can be terminated. This helps capture scenarios where the interface is expected to be idle under certain conditions.
- Global timeout: A global timeout can be defined to limit the duration of the simulation. If the simulation exceeds the specified time limit, it is automatically terminated.

These termination conditions provide flexibility and control over the simulation process, ensuring efficient use of resources and timely completion of the verification tasks.

13. What is a scoreboard?

The term "scoreboard" is used in the verification domain but its exact definition can vary. It can refer to different aspects depending on the context.

In some cases, a scoreboard refers to a storage data structure used to store and track information during the verification process. It acts as a repository for storing results, status, or other relevant data related to the verification tasks.

In other cases, a scoreboard may encompass not only the storage data structure but also the transfer and comparison functions associated with the dynamic response-checking process. It includes mechanisms to transfer information between the design under test (DUT) and the verification environment, as well as perform comparisons and checks against expected values.

It's worth noting that within the Verification Methodology Manual (VMM) methodology, the term "scoreboard" specifically refers to the comprehensive dynamic response-checking structure that includes storage, transfer, and comparison functions.

14. How are test cases included in the simulation environment?

Test cases are included in the simulation environment to verify the functionality of a design. Different approaches can be used to incorporate test cases into the simulation environment.

- Separate Compilation:

In this method, each test case is compiled individually along with the testbench code. It requires separate compilation and simulation for each test case.

Example:

```
// Testbench File: tb.v
// Test Case Files: testcase_1.v, testcase_2.v
compile_command tb.v testcase_1.v
compile_command tb.v testcase_2.v
```

- Include Test Case Files:

Test case files are directly included in the testbench code using the include directive. This method avoids separate compilation but still requires individual simulation for each test case.

Example:

```
// Testbench File: tb.v
// Test Case Files: testcase_1.v, testcase_2.v
cp testcase_1.v test.v
compile_command tb.v test.v
```

or

```
// Testbench File: tb.v
// Test Case Files: testcase_1.v, testcase_2.v
cp testcase_2.v test.v
compile_command tb.v test.v
```

- Single Intermediate File:

Multiple test case files are combined into a single intermediate file. The testbench code is compiled once, and during simulation, specific test cases are selected using command-line arguments or other means.

Example:

```
// Testbench File: tb.v
// Test Case Files: testcase_1.v, testcase_2.v
cat testcase_1.v > test.v
compile_command tb.v test.v
```

or

```
// Testbench File: tb.v
// Test Case Files: testcase_1.v, testcase_2.v
cat testcase_2.v > test.v
compile_command tb.v test.v
```

Simulation Command:

```
run_command +testcase_1

or

run_command +testcase_2
```

15. What are the different ways test cases are included for simulations?

There are different ways to include test cases for simulations:

1) Compile once, simulate multiple times with different test cases:
   - ✓ Compile the testbench and all test cases together.
   - ✓ During simulation, use $plusargs or other means to select the specific test case to execute.
   - ✓ This approach is commonly used in methodologies like OVM and UVM to save compilation time in regressions.

2) Separate compilation:
   - ✓ Compile the testbench code once.
   - ✓ For each test case, compile the corresponding test case code separately.
   - ✓ Link the compiled testbench code with each test case and simulate them individually.

3) Compile once, simulate once:
   - ✓ Compile the testbench and all test cases with similar configuration settings.
   - ✓ Simulate them in a single run, applying a hard reset after each test case to ensure a fresh start.
   - ✓ This method is used in VMM 1.2 and saves time in regressions.

4) Compile once, simulate multiple times with different data:
   - ✓ In some verification environments, test case code may not need to be compiled.
   - ✓ Test case files contain data that is read by the testbench to generate different scenarios.
   - ✓ The test case file can be read using $fopen or $plusargs in Verilog.There may be other ways to accomplish this as well.

16. Write code for a clock generator.

```
reg clk;
initial clk = 0;
always #10 clk = ~clk;
```

17. What is a test plan? What it contains?

- Test Plan:

A test plan is a detailed document that outlines the strategy and objectives for functional verification in a project. It serves as a roadmap for the verification process and guides the verification engineers throughout the project. The test plan ensures that all features of the design are thoroughly verified, and the verification goals are achieved.

- Content of a Test Plan:

- Overview: Provides an introduction to the project, specification followed, and methodologies used.
- Feature Extraction: Contains a list of features to be verified, each associated with a unique ID, description, expected result, and priority.
- Resources, Budget, and Schedule: Details the manpower, tools, and schedule required for verification phases.
- Verification Environment: Describes the TestBench architecture, including component details, special techniques, and guidelines for component reuse.
- System Verilog Verification Flow: Explains the verification levels (block, sub-system, system) and phases (RTL, gate).
- Stimulus Generation Plan: Outlines the stimulus to be generated, transaction sequences, and various scenarios as per the specification.
- Checker Plan: Explains expected result checking in the TestBench using monitor/checker.
- Coverage Plan: Details functional coverage groups and assertions to be implemented in the verification environment.

The test plan ensures a comprehensive verification strategy, optimized resource usage, and effective communication within the verification team. It helps monitor progress and ensures the completion of the verification phase successfully.

18. Explain some coding guidelines that you followed in your environment?

Coding Guidelines in SystemVerilog:

In our environment, we follow a set of coding guidelines to ensure consistency, readability, and maintainability of our SystemVerilog code. These guidelines are essential for efficient collaboration among team members and improving the overall quality of the verification process.

1) Module and Interface Names:
   - Use descriptive names that convey the functionality of the module or interface.
   - Follow a consistent naming convention, such as using camelCase or snake_case.

2) Indentation and Formatting:
   - Use consistent indentation for better code readability (e.g., 2 or 4 spaces per level).
   - Format code blocks neatly to improve code comprehension.

3) Commenting:
   - Add clear and concise comments to explain the purpose and functionality of the code.
   - Document module ports, parameters, and other critical elements for easy understanding.

4) Avoiding Ambiguity:
   - Use explicit sensitivity lists in always_comb, always_ff, and always_latch blocks to avoid race conditions.
   - Avoid using blocking assignments in combinational logic for clear signal propagation.

5) Avoiding Blocking Assignments in Testbenches:
   - Use non-blocking assignments (<=) in testbenches to model concurrent behavior accurately.
   - Employ proper synchronization and event-driven mechanisms in testbench code.

6) Testbench Structure:
   - Organize testbench code in a modular structure to improve maintainability.
   - Use separate tasks/functions for repetitive tasks (e.g., stimulus generation, checking).

7) Assertions:
   - Include assertions to check the correctness of the DUT behavior and detect bugs early.
   - Use meaningful messages in assertions to identify the failing conditions quickly.

8) Parameterization:
   - Utilize parameters and define them at the module or package level to enhance code reusability.
   - Avoid hardcoding values and use parameters for configurable options.

9) Avoiding Undriven Signals:
   - Ensure all signals in the DUT are driven in testbench test cases to avoid floating values.

10) Using Enums and Enums' Methods:
   - Employ enums to define a set of related constants for better code clarity.
   - Utilize enums' methods (e.g., 'name, 'num) to manipulate and display enum values.

19. Explain about white/black box testing and gray-box testing.

- Black Box Testing:
  - Testbench interacts with the System Under Test (SUT) without knowledge of its internal structure.
  - Focuses on external behavior using inputs and examining outputs.
  - Ideal for validating functional requirements.
  - Doesn't require access to the DUT's source code or design details.

- White Box Testing:
  - Testbench has complete access to the internal structure of the DUT.
  - Allows direct observation and control of internal signals and states.
  - Suitable for in-depth verification of the DUT's design and logic paths.
  - May lead to less reusable testbench environments due to tight coupling with the DUT's implementation.

- Gray Box Testing:
  - Testbench has partial knowledge of the DUT's internal structure.
  - Accesses specific areas or signals while remaining unaware of the complete implementation.
  - Balances depth of White Box testing with independence of Black Box testing.
  - Useful for targeted verification of internal functionalities.

20. In a packet protocol, where is the packet comparison done?

The packet comparison is done in a scoreboard.

SYSTEMVERILOG QUETIONS

What would be the output of the following code, and how to avoid it?

```
for (int i = 0; i < N; i++) begin
fork
int j=i;
  begin
        #10 $display("J value is 0", j);
     end
join_none
   end
```

J is always N. By using the automatic keyword, this problem can be avoided.

```
for (int i = 0; i < N; i++) begin
fork
automatic int j=i;
  begin
        #10 $display("J value is 0", j);
     end
join_none
   end
```

The output of the provided code would display the value of "j" as 0 for each iteration of the loop. This is because the variable "j" is assigned the value of "i" inside the loop. However, it seems that the code has a typo, as the desired output is "J value is [i]" instead of "J value is 0".

To achieve the desired output and avoid the issue, you can modify the code as second option.

By using the "automatic" keyword and fixing the display statement to use the correct variable placeholder (%0d), the code will display the value of "j" as the corresponding value of "i" for each iteration of the loop.

Example:

```
module Testbench;
  parameter int N = 10; // Define the value of N as needed

  initial begin
```

```
    for (int i = 0; i < N; i++) begin
      fork
        automatic int j = i;
        begin
          #10 $display("J value is %0d", j);
        end
      join_none
    end
  end
endmodule
```

Do we need to call super.new() when extending a class? What happens if we don't call it?

In SystemVerilog (SV), when extending a class, the super.new() call is used to invoke the constructor of the base class. It is necessary to call super.new() within the derived class's constructor if the base class has defined one.

Calling super.new() ensures that the initialization and setup code in the base class constructor is executed before any additional code in the derived class constructor. It allows for proper initialization of inherited members and sets up the object's state correctly.

If you don't call super.new() when extending a class, the base class constructor will not be executed. This can lead to uninitialized or inconsistent states in the derived class object, potentially causing unexpected behavior or errors in your code.

Here's an example that demonstrates the usage of super.new() when extending a class in SystemVerilog:

```
lass BaseClass;

  int baseData;

  function new();
    baseData = 10; // Initialize baseData in the constructor
  endfunction

  function void display();
    $display("BaseData: %0d", baseData);
  endfunction
endclass

class DerivedClass extends BaseClass;
  int derivedData;

  function new();
    super.new(); // Call the base class constructor

    derivedData = 20; // Initialize derivedData in the derived class constructor
  endfunction
  function void display();
    super.display(); // Call the base class display function
    $display("DerivedData: %0d", derivedData);
  endfunction
endclass

module Testbench;
  initial begin
    DerivedClass obj = new();
    obj.display();
  end
endmodule
```

What is "this"?

The "this" keyword in SystemVerilog is used to refer to class properties, parameters, and methods of the current instance. It can only be used within non-static methods, constraints, and covergroups. The "this" keyword represents a pre-defined object handle that refers to the object used to invoke the method where "this" is used.

Here's an example that illustrates the usage of the "this" keyword:

```
class Packet;
  int addr;
  int data;

  // Constructor: assigns values to addr and data
  function new(int addr, int b);
    data = b;                // Assign the value of b to the data property
    this.addr = addr;       // Assign the value of addr to the addr property of
the current object
  endfunction
endclass

initial begin
  Packet p = new(10, 20);    // Create a new object of the Packet class with
addr = 10 and data = 20
end
```

What is the difference between

```
logic data_1;
var logic data_2;
wire logic data_3;
bit data_4;
```

```
var bit data_5;
```

`logic data_1;` declares a variable named data_1 of type logic. logic is a data type used to represent digital values with four-state logic (0, 1, X, Z).

`var logic data_2;` declares a variable named data_2 that can have different data types, including logic. The var keyword in SystemVerilog allows for flexibility in data types, meaning the variable can be assigned different types during runtime. In this case, data_2 is specifically declared as a variable of type logic.

`wire logic data_3;`declares a variable named data_3 of type logic and also specifies it as a wire type. Wires are commonly used for interconnecting components in a hardware design.

`bit data_4;` declares a variable named data_4 of type bit. bit is a data type used to represent a single binary value with two states (0 or 1).

`var bit data_5;` declares a variable named data_5 that can have different data types, including bit. Similar to var logic data_2, data_5 is declared as a variable of type bit in this case.