SYSTEM VERILOG

INTERVIEW QUESTIONS

PART

3

JAIRAJ MIRASHI
DESIGN VERIFICATION ENGINEER

1. **What is the need for virtual interfaces?**

The need for virtual interfaces in SystemVerilog is primarily to facilitate modular and reusable design. Virtual interfaces provide a way to abstract the communication between different modules or blocks in a design, allowing for better encapsulation and separation of concerns.

Here are some key reasons for using virtual interfaces in SystemVerilog:

- Modularity: Virtual interfaces enable a modular design approach by decoupling the internal implementation details of a module from its external interfaces. This promotes code reuse and simplifies the integration of different modules into a larger system.
- Hierarchical Connectivity: Virtual interfaces provide a hierarchical connectivity mechanism, allowing modules to be interconnected in a structured manner. This helps in managing complex designs by providing a clear and organized representation of the interconnections between different components.
- Flexibility: Virtual interfaces offer flexibility in terms of connectivity options. They allow for different types of connections such as direct, hierarchical, or even dynamic connections, depending on the design requirements. This flexibility makes it easier to modify and extend the design without significant changes to the underlying modules.
- Verification and Testbench Development: Virtual interfaces are extensively used in testbench development for verifying designs. They allow the creation of virtual connections between the design under test (DUT) and the testbench components, facilitating stimulus generation, monitoring, and data transfer for effective verification.
- Reusability: By using virtual interfaces, modules can be easily reused in different designs or testbenches. The decoupling of interfaces from the internal implementation promotes portability and reusability, leading to more efficient and maintainable designs.

2. **What are the advantages of OOP?**

- Data hiding: OOP allows for data hiding or encapsulation, which means that the internal data and implementation details of an object can be hidden from other objects or modules. This promotes information security and reduces the risk of unauthorized access or manipulation of data.
- Code reusability: Inheritance, one of the key concepts of OOP, enables code reuse. With inheritance, a new class (derived class) can inherit properties and behaviors from an existing class (base class). This helps to avoid redundant code and promotes efficient code organization and maintenance.
- Modularity and flexibility: OOP facilitates modularity by breaking down complex designs into smaller, self-contained objects. Each object encapsulates its data and functionality, promoting a clear separation of concerns. This modularity makes it easier to develop, test, and modify individual objects without impacting the entire system.
- Object instances: OOP allows the creation of multiple instances or objects from a single class. Each object can have its own set of data and maintain its independent state. This enables concurrent or parallel processing, as well as representing multiple instances of real-world entities in a system.

- Inheritance and polymorphism: Inheritance not only promotes code reuse but also enables polymorphism. Polymorphism allows objects of different classes to be treated interchangeably based on their common base class. This flexibility simplifies system design, as objects can be manipulated uniformly, even if they have different behaviors.
- Productivity and maintainability: OOP methodologies, such as encapsulation, abstraction, and inheritance, contribute to improved productivity and maintainability. Encapsulation and abstraction help manage complexity by hiding internal details and providing simplified interfaces. Inheritance reduces redundant code, making codebase maintenance more efficient and less error-prone.

3. In what context are you using the "foreach" loop?

Here are a few common contexts where the "foreach" loop:

- Array Iteration: When you have an array and need to perform a particular action on each element, the "foreach" loop can be used. It simplifies the process by automatically iterating over each element of the array.
- Collection Iteration: SystemVerilog also provides built-in collection types like queues and lists. The "foreach" loop can be used to iterate over the elements in these collections, accessing and processing each item in a sequential manner.
- Testbench Stimulus: In a SystemVerilog testbench, the "foreach" loop can be used to generate stimulus for simulation. For example, if you have a set of test vectors stored in an array, you can use a "foreach" loop to iterate over each vector and apply it to the design under test.
- Data Validation: In verification scenarios, the "foreach" loop can be helpful for iterating over expected results or reference data. It allows you to compare the actual output of the design with the expected values, ensuring correctness.
- Signal Monitoring: When monitoring signals in a simulation, the "foreach" loop can be used to iterate over a set of signals or variables and capture their values at specific time points.

4. Write code to print the contents of the array_2d[][] using a foreach loop?

Here's the code to print the contents of a two-dimensional array, array_2d, using a foreach loop:

```
int array_2d [][];
// Assume the array_2d has been initialized with values

foreach (array_2d[row]) begin
  // Iterate over each row in the 2D array
  foreach (array_2d[row][col]) begin
    // Iterate over each column in the current row
    $display("array_2d[%0d][%0d] = %0d", row, col, array_2d[row][col]);
    // Print the value at the current row and column
  end
end
```

5. I want to delay the simulation by the smallest unit of time, i.e., the minimum of all the time precision. How can I do it?

In SystemVerilog, you can delay the simulation by the smallest unit of time, which is the minimum of all the time precision, by using the #0 delay.

The # operator is used to specify a delay in simulation time. In this case, #0 represents a zero time delay, indicating the smallest possible delay.

Here's an example usage:

```
initial begin
  #0; // Delay simulation by the smallest unit of time
  $display("Simulation delayed by the smallest unit of time");
end
```

By including #0 in the initial block, the simulation will pause for the smallest time unit before executing the subsequent statements. This can be useful when precise timing control is required, especially for synchronization or coordination purposes in your design or testbench.

6. Is this a valid syntax?

enum {a=0, b=7, c, d=8} alphabet;

No, the given syntax is not valid in SystemVerilog. The values assigned to the elements in an enumerated data type do not automatically increment based on the previous element's value if a value is not explicitly assigned.

If you want the enumerated elements to automatically increment based on the previous element's value, you can define the enumerated data type without explicitly assigning values. In this case, the elements will be assigned values automatically, starting from 0 and incrementing by 1 for each subsequent element.

Here's an example that follows the behavior you described:

```
enum {a=0, b=7, c, d=8} alphabet; // ERROR a=0, b=7, c and d both assigned 8
```

In this example, the enumerated data type alphabet is defined with four elements: a, b, c, and d. The value of a will be 0, b will be explicitly assigned 7, c will be automatically assigned 8 (incremented from the previous value).

7. What is the use of the "type" operator?

A custom data type can be created in SystemVerilog to enable the use of the same data type for declaring multiple variables. This allows for consistency and reusability in the code.

Let's consider an example:

```
module tb;
  typedef enum {TRUE, FALSE} e_true_false;
  // Custom data type "e_true_false" is defined with two valid values: TRUE
and FALSE
  initial begin
    // Declare a variable of type "e_true_false" that can store TRUE or FALSE
    e_true_false answer;
    // Assign TRUE or FALSE to the enumerated variable
    answer = TRUE;
    // Display the string value of the variable
    $display("answer = %s", answer.name);
  end
endmodule
```

In this example, a module named "tb" is defined. It showcases the creation of a custom data type named "e_true_false" using an enumeration. This data type has two valid values: TRUE and FALSE.

Inside the "initial" block, a variable named "answer" of type "e_true_false" is declared. This variable can store either TRUE or FALSE. In this case, the value TRUE is assigned to the "answer" variable.

The code then displays the string representation of the "answer" variable using the "name" attribute, which returns the name of the enumerated value as a string.

By utilizing custom data types, we can enhance the readability, maintainability, and flexibility of our SystemVerilog code.

8. What type is there for the index of an array_name[*]?

In SystemVerilog, the index of an associative array with the [*] dimension is typically of type string or int, depending on the data type specified for the keys used for indexing.

Since associative arrays can have various data types as keys, the index type will match the declared key type. For example, if the associative array has keys of type string, the index type would be string. Similarly, if the keys are of type int, the index type would be int.

In the provided example, the associative array assoc_array is declared with [*] dimension, indicating a dynamically sized associative array. The index type of assoc_array would depend on the data type specified for the keys of the packet class used for indexing.

Here's a simple example :

```
class packet;
  int a;
  int b;

  function void display();
    $display("\tValue of a = %0d", a);
    $display("\tValue of b = %0d", b);
  endfunction
endclass

module assoc_array;
  // Declaration of array
  packet assoc_array[*];
  packet pkt;
  int count, qu[$], tmp_var;

  initial begin
    // Initialize packet objects and assign values
    pkt = new();
    pkt.a = 8;
    pkt.b = 3;
    assoc_array[2] = pkt;

    pkt = new();
    pkt.a = 0;
    pkt.b = 6;
    assoc_array[5] = pkt;

    pkt = new();
    pkt.a = 2;
    pkt.b = 6;
    assoc_array[6] = pkt;

    pkt = new();
    pkt.a = 1;
    pkt.b = 6;
    assoc_array[9] = pkt;
    //-------------------------------------------------------------------
    //------- Find Index Method -------
    //-------------------------------------------------------------------
    // Type-1: Returning one matching index
    qu = assoc_array.find_index with (item.a == 2);
    count = qu.size();

    for (int i = 0; i < count; i++) begin
      tmp_var = qu.pop_front();
      $display("Index of Assoc Array for a == 2 is %0d", tmp_var);
    end
```

```
    // Type-2: Returning multiple matching indexes
    qu = assoc_array.find_index with (item.b == 6);
    count = qu.size();

    for (int i = 0; i < count; i++) begin
      tmp_var = qu.pop_front();
      $display("Index of Assoc Array for b == 6 is %0d", tmp_var);
    end

    // Type-3: With multiple conditions
    qu  = assoc_array.find_index with (item.a == 2 && item.b == 6);
    count = qu.size();

    for (int i = 0; i < count; i++) begin
      tmp_var = qu.pop_front();
      $display("Index of Assoc Array for a == 2,  b == 6 is %0d", tmp_var);
    end

    // Type-4: With multiple conditions
    qu  = assoc_array.find_index with (item.a < 2 && item.b > 5);
    count = qu.size();

    for (int i = 0; i < count; i++) begin
      tmp_var = qu.pop_front();
      $display("Index of Assoc Array for a < 2,  b > 5 is %0d", tmp_var);
    end
  end
endmodule
```

9. In an array, if the index is out of the address bounds, then what will be the return value?

In SystemVerilog, if the index is out of the address bounds of an array, the behavior is considered out-of-bounds access, which is generally undefined and can lead to unpredictable results.

When accessing an array with an index that is out of bounds, the behavior can vary depending on the specific circumstances. Some possible outcomes include:

- Reading or writing to memory locations outside the bounds of the array, which can overwrite or read unintended data and potentially cause memory corruption.
- Triggering an error or an assertion, depending on the implementation or tool used.
- Returning a garbage value or an uninitialized value if reading from the out-of-bounds memory location.

To avoid such issues, it is important to ensure that array indexes are within the valid range of the array's dimensions.

10. What is the return type of the Array locator method find_index?

The return type of the find_index method in SystemVerilog is of integers (int).

11. Write a program to access elements randomly from a Queue.

```
module queues_array;
  // Declaration
  int queue[$];          // Queue to store elements
  int index;             // Index variable
  int temp_var;          // Temporary variable for storing the selected
element
  initial begin
    // Queue Initialization:
    queue = {7, 3, 1, 0, 8};
    $display("----- Queue elements with index -----");
    foreach (queue[i])
      $display("\tqueue[%0d] = %0d", i, queue[i]);
    $display("-------------------------------------\n");

    $display("Before Queue size is %0d", queue.size());

    repeat (2) begin
      index = $urandom_range(0, 4); // Generate random index from 0 to 4
      temp_var = queue[index];      // Select element at the random index
      $display("Value of Index %0d in Queue is %0d", index, temp_var);
    end
    $display("After Queue size is %0d", queue.size());
  end
endmodule
```

12. Declare a queue of integers with a maximum number of elements set to 256.

```
module QueueExample;
  int queue[$:256];

  initial begin
    // Add elements to the queue
    for (int i = 0; i < 256; i++) begin
      queue.push_back(i);
    end
    // Access and display the elements in the queue
    for (int i = 0; i < queue.size(); i++) begin
      $display("Element at index %0d: %0d", i, queue[i]);
    end
  end
endmodule
```

13. Explain how you debugged randomization failure.

When debugging randomization failures with pkt.randomize() in SystemVerilog, you can follow these steps:

- Check Constraints: Verify that the constraints defined for randomizing the pkt object are correct and properly capture the desired range and distribution of values. Make sure the constraints are not too restrictive, preventing valid randomization.
- Add Debug Statements: Insert $display or $monitor statements before and after the pkt.randomize() call to print out the values of variables and expressions involved in the randomization process. This helps you identify any unexpected values or issues during randomization.
- Enable Randomization Tracing: Use the $srandom system function to enable randomization tracing. This allows you to track the random values generated during simulation. By examining the random values, you can identify any unexpected or incorrect values that may be causing the failure.
- Check Initialization: Ensure that all variables and fields of pkt are properly initialized before calling pkt.randomize(). Uninitialized variables or fields can lead to unpredictable behavior during randomization.
- Seed Control: Use the $random_seed system function to control the seed value for randomization. Setting a fixed seed value can help in reproducing the same randomization sequence across multiple runs, aiding in debugging and understanding the randomization failures consistently.

Example:

```
class MyPacket;
  int data;
  rand int length;

  constraint valid_length {
    length inside {4, 8, 16};
  }

  function void display();
    $display("Data: %0d, Length: %0d", data, length);
  endfunction
endclass

module Testbench;
  initial begin
    MyPacket pkt;
    pkt = new();

    // Randomization failure debugging
    repeat (10) begin
      if (!pkt.randomize()) begin
        $display("Randomization failed! Trying again...");
        continue;
      end
```

```
      pkt.display();
    end
  end
endmodule
```

In this example, we have a class MyPacket with a random variable length constrained to take values 4, 8, or 16. We want to randomize instances of MyPacket and display their values. If randomization fails, we retry up to 10 times.

To debug randomization failures with pkt.randomize(), we added a debug statement inside the repeat loop to display a message when randomization fails. This helps us identify any constraint violations or other issues preventing successful randomization. By examining the displayed values, we can track down the cause of the failure and make necessary adjustments to the constraints or randomization process.

14. Is the randomize() method virtual?

No, the randomize() method is not virtual in SystemVerilog. The randomize() method is a built-in method provided by SystemVerilog for randomization of objects based on their constraints and randomization methods.

15. Write code for the below specification:

- **Input variables a, b are declared in the module.**
- **Generate random numbers such that a > b.**
- **Do not use $random or $urandom.**

```
module RandomNumbers;
  // Declare input variables
  logic [7:0] a;
  logic [7:0] b;

  initial begin
    repeat (10) begin
      // Randomize variables with constraints
      randomize(a, b) with { a > b; };

      // Display the generated values
      $display("a = %0d, b = %0d", a, b);
    end
    $finish;
  end
endmodule
```

16. Is pre_randomize() virtual or not?

If "yes", did you use the keyword "virtual" in front of pre_randomize()?

If "no", then what about the pre_randomize() definition in the extended class?

In SystemVerilog, the pre_randomize() method is not inherently declared as virtual. However, it can be overridden and made virtual in an extended class by using the virtual keyword in the derived class.

If the pre_randomize() method is declared as virtual in the base class, it is recommended to also use the virtual keyword in the derived class when overriding the method. This ensures that the method is correctly identified as virtual and allows polymorphic behavior when invoking the method.

Here's an example to illustrate the usage of virtual keyword with pre_randomize() method:

```
class MyBaseClass;
  virtual task pre_randomize();
    // Pre-randomization behavior in the base class
  endtask
endclass

class MyDerivedClass extends MyBaseClass;
  virtual task pre_randomize();
    // Pre-randomization behavior in the derived class
  endtask
endclass
```

In the example above, the pre_randomize() method is declared as virtual in both the base class (MyBaseClass) and the derived class (MyDerivedClass). This ensures that the derived class can override the method and provide its own implementation.

17. How to generate random numbers within a range of values?

To generate random numbers within a range of values using the inside constraint in SystemVerilog, you can use the inside operator in the constraint block of a class.

Here's the modified code with an explanation:

```
class Packet;
  rand bit [3:0] number;        // Random number variable
  rand bit [3:0] startnumber;   // Start of the range
  rand bit [3:0] endnumber;     // End of the range

  constraint numberRange {
    number inside {[startnumber:endnumber]};  // Constraint to ensure number
falls within the specified range
  }
endclass
```

```
module RandomRangeExample;
  initial begin
    Packet pkt;
    pkt = new();
    $display("------------------------------------");
    repeat (3) begin
      pkt.randomize();    // Randomize the packet
      $display("\tstartnumber = %0d, endnumber = %0d", pkt.startnumber,
pkt.endnumber);
      $display("\tnumber = %0d", pkt.number);
      $display("------------------------------------");
    end
  end
endmodule
```

❖ Generating random numbers without using constraints

```
module Tb();
  integer add;

  initial
    begin
      repeat(5)
        begin
          #1; // Delay for simulation time
          add = 40 + {$random} % (50 - 40); //MIN + {$random} % (MAX - MIN )
// Generate random number between 40 and 50

          $display("add = %0d", add); // Display the generated random number
        end
    end
endmodule
```

18. What is the difference between mailbox and queue?

The difference between a mailbox and a queue in SystemVerilog can be summarized as follows:

❖ Mailbox:

- A mailbox is a synchronization primitive used for communication between different processes or threads.
- It is typically used for one-to-one communication, where one process sends a message to another process.
- The sender process blocks until the message is consumed by the receiver process.
- Multiple messages can be stored in the mailbox, but only one message can be read at a time.
- The order of message consumption is not guaranteed.

❖ Queue:

- A queue is a data structure used for storing and retrieving elements in a specific order.
- It is typically used for one-to-many communication or data buffering.
- Multiple processes or threads can write messages into the queue, and multiple processes can read messages from the queue simultaneously.
- The order of message retrieval is determined by the order of insertion into the queue (FIFO - First-In-First-Out).
- The size of the queue can be dynamic or fixed.

In summary, mailboxes are mainly used for point-to-point communication and synchronization, while queues are used for storing and retrieving elements in a specific order, supporting multiple readers and writers.

19. What is the difference between $random and $urandom?

The difference between $random and $urandom in SystemVerilog is in the type of numbers they generate.

$random generates signed pseudorandom numbers, while $urandom generates unsigned pseudorandom numbers. Both functions return a new random number each time they are called.

Here is an example of using $random and $urandom:

```
module RandomExample;
  integer signedNum;
  integer unsignedNum;
  initial begin
    signedNum = $random();
    unsignedNum = $urandom();

    $display("Signed Number: %d", signedNum);
    $display("Unsigned Number: %d", unsignedNum);
  end
endmodule
```

20. What is the scope of randomization?

The scope of randomization in SystemVerilog refers to the visibility and applicability of the randomization process. It determines which variables and objects are eligible for randomization and where the randomization process can be invoked.

The scope of randomization depends on where the randomization is defined and how it is used. Here are some common scopes of randomization:

- Class Scope: Randomization can be defined within a class or a structure using the rand keyword. Randomization constraints and methods can be specified within the class to control the random values generated for class variables.
- Module/Interface Scope: Randomization can be performed at the module or interface level by declaring variables with the rand keyword. Randomization can be controlled using constraints defined within the module or interface.
- Local Scope: Randomization can also be performed within procedural blocks, such as initial blocks or functions, by declaring variables with the rand keyword. Randomization can be controlled using local constraints or by invoking randomization methods.

The scope of randomization determines which variables are subject to randomization and where the randomization process can be triggered. It allows you to specify the level at which you want random values to be generated and provides flexibility in controlling the randomization process based on your specific requirements.

Here's an example:

```
class Packet;
  rand bit [7:0] data;

  constraint dataConstraint {
    data > 10;
    data < 100;
  }

  function void display();
    $display("Data: %0d", data);
  endfunction
endclass

module RandomizationExample;
  initial begin
    Packet pkt1; // Object at module scope

    pkt1 = new();
    pkt1.randomize(); // Randomization at module scope
    pkt1.display();

    repeat (3) begin
      automatic Packet pkt2; // Object at local scope

      pkt2 = new();
      pkt2.randomize(); // Randomization at local scope
      pkt2.display();
    end
  end
endmodule
```