SYSTEM VERILOG INTERVIEW QUESTIONS

PART 4

JAIRAJ MIRASHI
DESIGN VERIFICATION ENGINEER

1. What is scope randomization?

Scope randomization refers to the process of randomizing data within the current scope without the need to define a class or instantiate an object. It is achieved using the std::randomize() function. The function allows for randomizing variables passed as arguments, and there is no limit on the number of arguments.

If a property does not have the rand or randc attribute, it can still be randomized by passing it as an argument to the randomize() method. For example: success = std::randomize(a, b); where std refers to the class name.

Inline constraints can also be added using the randomize() method.

For instance: `success = std::randomize(a, b) with {a > 10; b < 20; a < 20};`

The std::randomize() function behaves similarly to a class's randomize method, but instead operates on variables within the current scope rather than member variables of a class.

Upon successful randomization, the std::randomize() function returns 1, and it returns 0 on failure.

Here's an example usage of scope randomization within a module:

```
module top;
    int a;
    int b;
    bit success;

    initial begin
        success = std::randomize(a, b);
        $display("a=%d and b=%d", a, b);
    end
endmodule
```

2. What is the need of pre and post randomize methods?

The pre-randomize and post-randomize methods in SystemVerilog are user-defined methods within a class that are called before and after the randomization process, respectively. These methods provide additional control and customization over the randomization process. Here's a breakdown of their need:

❖ Pre-Randomize Method:

The pre-randomize method, also known as pre_randomize(), is called before the randomization process starts for an object. It allows you to perform any necessary initialization or setup tasks before the randomization begins. This method is typically used to set up default values, apply additional constraints, or perform any other preparatory actions.

For example, if you have certain variables that need to be initialized or constrained before randomization, you can use the pre-randomize method to ensure that those requirements are met. It helps in maintaining consistency and ensuring that the object is in a valid state before randomization takes place.

❖ Post-Randomize Method:

The post-randomize method, also known as post_randomize(), is called after the randomization process completes for an object. It allows you to perform any necessary post-processing or validation tasks on the randomized data. This method is typically used to verify the generated random values, perform additional calculations, or apply post-processing operations.

For example, if you need to validate the randomized values against specific criteria or perform calculations based on the randomized data, you can use the post-randomize method to implement such checks or operations. It helps in ensuring that the generated random values meet the desired requirements and can be further processed or used correctly.

Overall, the pre-randomize and post-randomize methods provide flexibility and control over the randomization process. They allow you to customize the behavior before and after randomization, enabling you to initialize, constrain, validate, or process the randomized data according to your specific needs and requirements.

3. What is the relationship between classes and objects?

A class can be thought of as a blueprint or template that defines the structure, properties, and behaviors of an object. It describes what an object of that class will look like and what it can do. It serves as a reusable design or specification.

On the other hand, an object is an instance or realization of a class. It is like an actual building constructed based on a blueprint. An object has its own unique set of data and can perform actions defined by the class. Multiple objects can be created from the same class, each having its own independent set of data.

To understand the relationship between classes and objects, think of a class as a general description or plan, and an object as a specific realization or implementation of that plan. The class defines the common properties and behaviors shared by all objects of that class, while each object has its own individual data and can perform actions based on the class's definition.

4. What are packages?

In SystemVerilog, packages are containers or namespaces that group related definitions, such as data types, functions, tasks, and constants, together. They provide a way to organize and modularize code by grouping related elements into a single unit.

Packages allow for better code organization, reusability, and avoid naming conflicts. They can be used to create libraries of reusable components that can be easily included in multiple designs or testbenches.

A package is defined using the package keyword, and its contents are enclosed within the package and endpackage keywords. The elements within a package, such as data types or functions, can be accessed using the package scope operator ::.

Here's a simple example of a package definition:

```
package myPackage;
   // Data types
   typedef logic [7:0] myType;

   // Constants
   localparam int MAX_VALUE = 100;

   // Functions
   function int add(int a, int b);
      return a + b;
   endfunction
endpackage
```

5. What is a constructor in a class?

A class constructor is a special member function that is automatically called when an object of that class is created using the new method. The constructor is responsible for allocating memory, initializing properties, and returning the address of the memory to handle.

Here's an example illustrating the explicit definition of a class constructor:

```
class Packet;
   bit [31:0] addr;
   bit [31:0] data;

   // Constructor
   function new(bit [31:0] data = 15);
      addr = 20;   // Initialize addr to 20
   endfunction
endclass

module tb;
```

```
   Packet pkt;

   initial begin
      pkt = new;  // Create an object of Packet class
      $display("addr=%d, data=%d", pkt.addr, pkt.data);
   end
endmodule
```

6. What is the difference between always_comb and always@(*)?

❖ always_comb:

always_comb is used to describe combinational logic that depends on signals within its sensitivity list.

It implies that the block should execute whenever any of the signals in its sensitivity list change, and it will produce a combinational output based on the current values of those signals.

It ensures that the block is automatically sensitive to changes in the signals without explicitly mentioning them in the sensitivity list.

It helps to avoid mistakes caused by missing signals in the sensitivity list.

Example:

```
always_comb
begin
   sum = a + b;
   carry = a & b;
end
```

In this example, the always_comb block computes the sum and carry signals based on the inputs a and b. The block will be executed whenever a or b changes, producing the updated sum and carry values.

❖ always @(*):

always @(*) is used to describe combinational logic that depends on any signal within the design, and it ensures that the block executes whenever any signal within the design changes.

It does not require an explicit sensitivity list and automatically detects changes in any signal within the design.

It is useful when you want to capture all signals that affect the block's behavior without manually specifying them.

Example:

```
always @(*)
begin
   if (enable)
      output = inputA + inputB;
   else
      output = 0;
end
```

In this example, the always @(*) block calculates the output based on the inputs inputA and inputB. It also considers the enable signal to determine whether to perform the addition or assign zero to the output. Whenever any of the signals (inputA, inputB, or enable) change, the block is triggered to update the output value.

7. What is the difference between a union and a structure?

Union	Structure
A union is a composite data type that allows different variables to share the same memory space. In other words, all variables within a union occupy the same memory location.	A structure is a composite data type that allows you to group multiple variables of different data types together
The size of a union is determined by the largest variable within it.	Each member within a structure has its own separate memory location.
Only one member of the union can be accessed at a time, and accessing one member may overwrite the values of other members.	The size of a structure is determined by the sum of the sizes of its members.
Unions are useful when you want to store different types of data in the same memory location and need to conserve memory space.	All members of a structure can be accessed simultaneously and independently.
	Structures are useful when you want to group related variables together to represent a complex data structure.

8. How to call a task defined in a parent object in derived classes?

In SystemVerilog, you can call a task defined in a parent object from a derived class by using the super keyword. The super keyword refers to the parent object within the derived class.

Here's an example that demonstrates how to call a task defined in a parent object from a derived class:

```
class Parent;
   task myTask();
      $display("Executing myTask() in Parent class");
   endtask
endclass

class Derived extends Parent;
   task myTask();
      super.myTask(); // Call myTask() from the Parent class
      $display("Executing myTask() in Derived class");
   endtask
endclass

module Testbench;
   Derived obj;

   initial begin
      obj = new;
      obj.myTask();  // Call myTask() from the Derived class
   end
endmodule
```

In the above code, we have a Parent class that defines a task myTask(). The Derived class extends the Parent class and overrides the myTask() task. Within the myTask() task of the Derived class, we use super.myTask() to call the myTask() task from the parent Parent class. This allows us to execute the task defined in the parent class before executing the code specific to the derived class.

In the Testbench module, we create an object obj of the Derived class and call the myTask() task using obj.myTask(). This triggers the execution of the myTask() task defined in both the parent and derived classes.

Using super.myTask() allows you to invoke and execute the task defined in the parent object from within a derived class, providing a way to reuse and extend the functionality defined in the parent class.

9. What is the difference between rand and randc?

❖ rand Keyword:

➢ Variables declared with the rand keyword are standard random variables.

➢ The values of these variables are uniformly distributed over their specified range.

➢ Random values assigned to rand variables can be repeated during the simulation.

➢ For example, if a rand variable is declared as rand bit[1:0] var1;, it can take on values such as 0, 1, 2, or 3, and these values can be repeated randomly.

❖ randc Keyword:

➢ Variables declared with the randc keyword are random cyclic variables.

➢ When randomizing a randc variable, the assigned values do not repeat until every possible value within the range has been assigned.

➢ This means that the sequence of values for a randc variable is deterministic within a simulation run, but it will iterate through all possible valid values before repeating the sequence.

➢ For example, if a randc variable is declared as randc bit[1:0] var1;, it will cycle through all possible combinations of 0, 1, 2, and 3 without repeating until all values have been assigned.

10. What is the purpose of solving constraints before randomization?

Solving constraints before randomization ensures that the generated random values meet specific requirements and are valid for the variables being randomized. It helps prevent invalid or unrealistic values from being assigned during the randomization process.

11. Explain pass by reference and pass by value.

❖ Pass by value:

➢ In pass by value, each argument is copied into the subroutine area.

➢ Any changes made to the argument within the subroutine will not affect the original value outside the subroutine.

➢ The original value remains unchanged.

➢ Example: In the provided code, the sum function uses pass by value for arguments x and y. Changes made to x within the function do not affect the value of x outside the function.

```
module argument_passing;
  int x,y,z;
  //function to add two integer numbers.
  function int sum(int x,y);
    x = x+y;
    return x+y;
  endfunction
  initial begin
    x = 20;
    y = 30;
    z = sum(x,y);
    $display("-------------------------------------------------------");
    $display("\tValue of x = %0d",x);
    $display("\tValue of y = %0d",y);
    $display("\tValue of z = %0d",z);
    $display("-------------------------------------------------------");
  end
endmodule
```

❖ Pass by reference:

➢ In pass by reference, a reference to the original argument is passed to the subroutine.

➢ Changes made to the argument within the subroutine will be reflected in the original value outside the subroutine.

➢ The original value can be modified.

➢ Example: In the provided code, the sum function uses pass by reference for argument x by using the keyword "ref". Changes made to x within the function affect the value of x outside the function.

```
module argument_passing;
  int x,y,z;

  //function to add two integer numbers.
  function automatic int sum(ref int x,y);
    x = x+y;
    return x+y;
  endfunction

  initial begin
    x = 20;
    y = 30;
    z = sum(x,y);
    $display("--------------------------------------------------------");
    $display("\tValue of x = %0d",x);
    $display("\tValue of y = %0d",y);
    $display("\tValue of z = %0d",z);
    $display("--------------------------------------------------------");
  end
endmodule
```

12. What is the difference between bit[7:0] sig_1; and byte sig_2;?

bit[7:0] represents an unsigned 8-bit variable that can count up to 255, while byte represents a signed 8-bit variable with a range of -128 to 127.

13. What is the difference between a program block and a module?

❖ Program Block:

- The program block provides a race-free interaction between the design and the testbench.
- Statements within the program block are scheduled in the Reactive region, ensuring synchronization with the design.
- It can be instantiated and its ports can be connected, similar to a module.
- It can contain one or more initial blocks.
- It cannot contain always blocks, modules, interfaces, or other programs.
- Variables within the program block can only be assigned using blocking assignments, and using non-blocking assignments will result in an error.

Example:

```
//-----------------------------------------
// Design code
//-----------------------------------------
module design_ex(output bit [7:0] addr);
  initial begin
    addr <= 10; // Assign the value 10 to the 'addr' output
  end
endmodule

//------------------------------------------
// Testbench
//------------------------------------------
module testbench(input bit [7:0] addr);
  initial begin
    $display("\t Addr = %0d",addr); // Display the value of 'addr' in the
testbench
  end
endmodule

//-----------------------------------------
// Testbench top
//-----------------------------------------
module tbench_top;
  wire [7:0] addr; // Declare a wire for 'addr' as an 8-bit signal

  // Design instance
  design_ex dut(addr); // Instantiate the design module and connect 'addr'
wire

  // Testbench instance
  testbench test(addr); // Instantiate the testbench module and connect
'addr' wire
endmodule
```

❖ Module:

➢ A module is used to define the behavior and structure of digital circuits.
➢ It encapsulates logic and data and provides a way to organize and reuse design components.
➢ Modules are synthesized into hardware and generate the corresponding gate-level netlist.
➢ It can contain input/output ports, internal signals, and procedural blocks such as always blocks or initial blocks.
➢ Modules follow an active region and reactive region scheduling, where non-blocking assignments are scheduled in the active region.
➢ Modules define the connectivity and behavior of the design components.

Example:

```
//----------------------------------------------------------------------------
// Design code
//----------------------------------------------------------------------------
module design_ex(output bit [7:0] addr);
  initial begin
    addr <= 10; // Assign the value 10 to the 'addr' output
  end
endmodule

//----------------------------------------------------------------------------
// Testbench
//----------------------------------------------------------------------------
program testbench(input bit [7:0] addr);
  initial begin
    $display("\t Addr = %0d",addr); // Display the value of 'addr' in the
testbench
  end
endprogram

//----------------------------------------------------------------------------
// Testbench top
//----------------------------------------------------------------------------
module tbench_top;
  wire [7:0] addr; // Declare a wire for 'addr' as an 8-bit signal

  // Design instance
  design_ex dut(addr); // Instantiate the design module and connect 'addr'
wire

  // Testbench instance
  testbench test(addr); // Instantiate the testbench module and connect
'addr' wire
endmodule
```

14. I have an object of a class. I would like to print the class name using the object handle.

```
class MyClass;
  function string getClassName();
    return $typename(this);
  endfunction
endclass

module top;
  initial begin
    MyClass myObj = new;
    $display("Class name: %s", myObj.getClassName());
  end
endmodule
```

In this code, we have a class named MyClass that contains a function getClassName(). Inside the function, the $typename system function is used to retrieve the name of the class. The function returns the class name as a string.

Inside the top module, an object myObj of MyClass is instantiated using the new keyword. The getClassName() function is called on myObj to obtain the class name. Finally, the class name is displayed using the $display statement.

15. What is the difference between for/join, fork/join_none, and fork/join_any? (LINK)

❖ Fork-Join: This construct starts all the processes inside it in parallel and waits for the completion of all the processes before moving to the next step. The fork block is blocking and will be blocked until all processes are finished.

Fork-Join example:

```
module fork_join;
 initial begin
  fork
   begin
    $display($time,"\tProcess-1 Started");
    #5;
    $display($time,"\tProcess-1 Finished");
   end
   begin
    $display($time,"\tProcess-2 Started");
    #20;
    $display($time,"\tProcess-2 Finished");
   end
  join
  $display($time,"\tOutside Fork-Join");
  $finish;
 end
endmodule
```

- Fork-Join_any: This construct starts all the processes inside it in parallel and waits for the completion of any one process before moving to the next step. The fork block is blocking and will be blocked until any one process is completed.

Fork-Join_any example:

```
module fork_join;
 initial begin
  fork
   begin
    $display($time,"\tProcess-1 Started");
    #5;
    $display($time,"\tProcess-1 Finished");
   end
   begin
    $display($time,"\tProcess-2 Started");
    #20;
    $display($time,"\tProcess-2 Finished");
   end
  join_any
  $display($time,"\tOutside Fork-Join");
 end
endmodule
```

- Fork-Join_none: This construct starts all the processes inside it in parallel, and the fork block is non-blocking. It does not wait for the completion of the processes and allows them to execute simultaneously.

Fork-Join_none example:

```
module fork_join_none;
 initial begin
  fork
   begin
    $display($time,"\tProcess-1 Started");
    #5;
    $display($time,"\tProcess-1 Finished");
   end
   begin
    $display($time,"\tProcess-2 Started");
    #20;
    $display($time,"\tProcess-2 Finished");
   end
  join_none
  $display($time,"\tOutside Fork-Join_none");
 end
endmodule
```

16. What is the use of modports?

Modport is a construct in SystemVerilog that is used to define signal directions within an interface. It allows different components to have different input-output directions for each module or entity they interact with.

Here's an example to illustrate the use of modports:

```
interface myBus (input clk);
  logic [7:0] data;
  logic enable;

  // From TestBench perspective, 'data' is input and 'enable' is output
  modport TB (input data, clk, output enable);

  // From DUT perspective, 'data' is output and 'enable' is input
  modport DUT (output data, input enable, clk);
endinterface
```

17. Write a clock generator without using always block.

```
module ClockGenerator;
  reg clk;

  initial begin
    clk = 0;
    forever begin
      #5 clk = ~clk; // Toggle the clock every 5 time units
    end
  end

endmodule
```

18. Difference between Associative array and Dynamic array?

❖ Dynamic Array:

➢ A dynamic array is declared with empty word subscripts [] or square brackets.

➢ The array size is not specified at compile time, but rather at runtime using the new[] constructor.

➢ The array is initially empty, and space is allocated using new[] to determine the number of entries.

. Example code for Dynamic Array:

```
module dynamic;
  int dyn[], d2[];

  initial begin
    dyn = new[5]; // Allocate 5 elements
    foreach (dyn[j])
      dyn[j] = j; // Initialize the elements
    d2 = dyn; // Copy a dynamic array
    d2[0] = 5; // Modify the copy

    $display(dyn[0], d2[0]); // See both values (0 & 5)

    dyn = new[20](dyn); // Expand and copy
    dyn = new[100]; // Allocate 100 new integers, old values are lost
    $display(dyn.size); // Size of dyn is 100
    dyn.delete; // Delete all elements
  end

endmodule
```

❖ Associative Array:

➢ An associative array is suitable when the size of the collection is unknown or the data space is sparse.

➢ Associative arrays do not have any storage allocated until they are used.

➢ Index expressions are not limited to integral expressions but can be of any type, such as strings.

. Example code for Associative Array:

```
module tb;
  int array1[int];
  int array2[string];
  string array3[string];

  initial begin
    // Initialize each associative array with some values
    array1 = '{1: 22, 6: 34};
    array2 = '{ "Ross": 100, "Joey": 60 };
    array3 = '{ "Apples": "Oranges", "Pears": "44" };

    // Print each associative array
    $display("array1 = %p", array1);
    $display("array2 = %p", array2);
    $display("array3 = %p", array3);
  end
endmodule
```

19. Why is an always block not allowed in a program block?

An always block is not allowed in a program block because they serve different purposes and have different execution mechanisms in SystemVerilog.

A program block is used to define a reusable collection of code that can be instantiated and connected to different components. It is primarily used for testbench organization and structure.

On the other hand, an always block is used to describe the behavior of hardware by specifying a set of conditions and associated statements. It is used to model the behavior of registers, combinational logic, and sequential processes.

The reason why an always block is not allowed in a program block is that the execution semantics of program blocks and always blocks are fundamentally different. Program blocks operate in the reactive region, where the execution is event-driven and synchronized with the design. Always blocks, on the other hand, operate in the active region and are scheduled based on the sensitivity list.

Combining the two constructs would lead to a potential race condition between the testbench and the design, as the program block's execution would be scheduled in the reactive region while the always block's execution would be scheduled in the active region. This could result in unpredictable and erroneous behavior.

20. Initial

wait_order(a, b, c);

Which from below initial process will cause the above wait order to pass.

a)
```
initial begin
 #1;
q->a;
q->b;
q->c;
 end
```

b)
```
initial begin
 #1;
q->a;
 end
always @a
 q->b;
always @b
 q->c;
```

c)
```
initial begin
 #1;
 q->a;
q#0re->b;
 q->>c;
 end
```

d)
```
initial begin
q#1re->a;
q#1re->b;
q#1re->c;
 end
```

Option "a":

```
initial begin
  #1;
  q->a;
  q->b;
  q->c;
end
```

In this option, the wait order (a, b, c) is being passed correctly. Here's the breakdown of the code:

The initial block is executed.

After a delay of 1 time unit (#1), the first transaction q->a; occurs. It means that a transaction is sent to the signal q with the value of a.

Following that, the next transaction q->b; sends the value of b to q.

Finally, the transaction q->c; sends the value of c to q.

The order of the transactions matches the wait order (a, b, c), ensuring that all three values are sent in the expected sequence.

By executing these statements within the initial block, the specified wait order is correctly passed, fulfilling the given requirement.