SYSTEM VERILOG INTERVIEW QUESTIONS

PART

#1

JAIRAJ MIRASHI
DESIGN VERIFICATION ENGINEER

1. When import is used and when `include is used?

In SystemVerilog, both import and include are used for including external files, but they serve different purposes.

1. `include directive:
   - The include directive is used to insert the contents of an external file directly into the current file at the location where the directive is placed.
   - It is mainly used for including common definitions, declarations, or macros that are required across multiple files.
   - The included file is treated as if its content was directly written in place of the include directive.
   - The syntax for include directive is: include "filename"

2. `import statement:
   - The import statement is used to bring in names (such as modules, packages, or individual identifiers) from an external file or a namespace into the current scope.
   - It is primarily used for modularizing designs and organizing code hierarchically.
   - The imported items can be used directly in the current file without any prefix.
   - The syntax for import statement depends on what you are importing. For example, to import a module, the syntax is: import module_name::*;

2. What is the output of the below code?

```
class myClass #(type T = int);

static int a = static_function();

static function int static_func();

$display("In class myClass#()");

return 10;

endfunction : static_func

endclass : myClass

module mod_def();

initial begin

end
```

```
endmodule : mod_def
```

To answer the question about the output of the code, it's important to analyze the code and understand its structure and behavior. Let's break it down:

- The code defines a SystemVerilog class called myClass with a type parameter T defaulting to int.
- Within the class, there is a static variable a that is assigned the result of calling a static_function().
- The class also contains a static function static_func() that returns an integer value of 10 and displays the message "In class myClass#()" using $display.
- The code then defines a module called mod_def without any specific functionality inside its initial block.

Now, since the code does not include any instantiation of the myClass or any usage of the mod_def module, there is no direct output produced by the code itself. Therefore, the output would be empty.

3. What will be the output if the following line is added to the above code?

```
typedef cls_def#(int) cl_int;
```

In the provided code, the line typedef cls_def#(int) cl_int; is attempting to create a typedef of a class called cls_def instantiated with the int type parameter, and the typedef name cl_int. However, there is no definition or instantiation of the cls_def class in the code snippet you provided. Therefore, if you add this line to the code, it will result in a compilation error.

Without the definition of the cls_def class, it is not possible to determine the output of the code.

4. In the below example, "a1 = b1" is not allowed. Why is it not allowed, and what is the solution?

```
class MyBase #(type T = int);

virtual function void func1();

$display("Inside MyBase");

endfunction : func1

endclass


class MyDerived extends MyBase #(real);
```

```
virtual function void func1();
$display("Inside MyDerived");
endfunction : func1
endclass

module top;
MyBase a1;
MyDerived b1 = new();

initial begin
a1 = b1; // This assignment is not allowed
a1.func1();
end
endmodule
```

The assignment 'a1 = b1' is not allowed because SystemVerilog requires explicit casting when assigning objects of derived classes to objects of base classes. To resolve this, we can use explicit casting by adding the line '$cast(a1, b1); // Correct explicit casting syntax. $cast(a1, b1) casts the object b1 of type MyDerived to type MyBase and assigns it to a1. This casting ensures type safety and allows 'a1' to refer to the 'MyDerived' object.

5. If a virtual method is defined in a class using extended class handle/object, how can the virtual method defined in the base class be called?

To call the virtual method defined in the base class when using an extended class handle/object, we can use the super keyword. By declaring an object or handle of the base class type and assigning an instance of the derived class to it, we can then use super.myVirtualMethod() inside the derived class to explicitly call the virtual method defined in the base class. This way, we ensure that the base class implementation of the virtual method is executed.

Here's an example code snippet to illustrate this:

```
class MyBase;
```

```
  virtual function void myVirtualMethod();
    $display("Inside MyBase");
  endfunction
endclass

class MyDerived extends MyBase;
  virtual function void myVirtualMethod();
    $display("Inside MyDerived");
    super.myVirtualMethod(); // Calling the base class virtual method
  endfunction
endclass

module top;
  MyBase obj = new;
  MyDerived derivedObj = new;

  initial begin
    obj.myVirtualMethod(); // Calls the virtual method defined in MyBase
    derivedObj.myVirtualMethod(); // Calls the virtual method defined in MyDerived
  end
endmodule
```

6. What is the difference between a Verilog function and an SystemVerilog function?

❖ Verilog Function:

- In Verilog, functions are used for behavioral modeling and can only be declared inside a module or an initial/always block.
- Verilog functions can only operate on scalar and vector inputs and outputs.
- Verilog functions do not support hierarchical references to module instances.

- Verilog functions cannot have input/output ports and cannot be directly instantiated or called from other modules.

❖ SystemVerilog Function:

- SystemVerilog introduces a more advanced and enhanced version of functions.
- SystemVerilog functions can be declared inside modules, classes, or interfaces.
- SystemVerilog functions can operate on complex data types, including arrays, structures, and user-defined types.
- SystemVerilog functions support hierarchical references to module instances and can be called from other modules or classes.

SystemVerilog functions can have input/output ports, allowing them to be instantiated and used like modules.

SystemVerilog functions can have additional features like automatic tasks, which allow you to include procedural statements along with the function.

7. What is the output of the following program?

```
program main();

initial begin

fork

#25 $display("time = %0d at T25", $time);

join_none


fork

#20 $display("time = %0d at T20", $time);

#10 $display("time = %0d at T10", $time);

#5  $display("time = %0d at T10", $time);

join_any

disable fork;

end

initial
```

```
#100 $finish;

endprogram
```

The output of the program would be as follows:

```
time = 5 at T10

time = 10 at T10

time = 20 at T20

time = 25 at T25
```

Explanation:

At time 5, the $display statement inside the second fork block is executed and displays "time = 5 at T10".

At time 10, the $display statement inside the second fork block is executed again and displays "time = 10 at T10".

At time 20, the $display statement inside the second fork block is executed once more and displays "time = 20 at T20".

At time 25, the $display statement outside the fork blocks is executed and displays "time = 25 at T25".

After the execution reaches time 100, the program finishes and terminates.

8. Why Constructor is not virtual?

In SystemVerilog, constructors are not allowed to be declared as virtual. The main reason for this is that constructors are responsible for initializing the object and setting up its initial state. The construction process typically occurs before the object is fully formed and before any inheritance relationships are established.

Virtual functions, on the other hand, are used to enable polymorphism, which allows objects of different derived classes to be treated as objects of their common base class. Virtual functions are invoked based on the actual runtime type of the object rather than the declared type of the handle or reference to the object.

9. Write a $display statement to print the value of an enumerated variable.

```
typedef enum logic [1:0] {RED, GREEN, BLUE} Color;

module Testbench;
  Color myColor;

  initial begin
    myColor = RED;

    $display("The value of myColor is: %s", myColor);
    $finish;
  end
endmodule
```

In this example, the enumerated variable myColor is defined with three possible values: RED, GREEN, and BLUE. Inside the initial block, the value of myColor is set to RED, and then it is displayed using $display statement with the %s format specifier, which is used for displaying string values. The output will show the string representation of the enumerated value RED.

10. What is the output of the following code?

```
The value of myColor is: RED
```

What is the output of the following code?

```
module myModule();

  string myString[7];
  int i, j, k, file;
  initial begin
    string s;
    file = $fopen("file.txt", "r");

    while (!$feof(file)) begin
      k = $fscanf(file, "", s);
      myString[i] = s;
      i++;
    end

    $fclose(file);

    foreach (myString[j]) begin
      $display("index j = %0d, string = %s", j, myString[j]);
    end

    $finish;
  end
endmodule
```

The contents of the "file.txt" file are as follows:

```
================
aa
bb
cc
================
```

The expected output is:

```
Index j = 0, string = aa

Index j = 1, string = bb

Index j = 2, string = cc

Index j = 3, string = cc
```

Please note that "cc" is read twice because of the $feof

11. How to deallocate an object?

When an object is no longer needed in SystemVerilog, the automatic memory management system automatically reclaims the memory, making it available for reuse. The automatic memory management system is an integral part of SystemVerilog. If you want to deallocate an object explicitly, you can simply assign null to the object.

Here's an example:

```
testclass obj; // Declare a handle obj for testclass

obj = new; // Construct a testclass object and store the address in obj. "new" allocates space for testclass

obj = null; // Deallocate the object. This frees up the memory space for the object.
```

12. What is a callback?

A callback in SystemVerilog refers to a mechanism that allows a testbench to provide a "hook" or entry point for injecting new code without directly modifying the original classes or code. Callbacks are used in testbenches to add or modify functionality in certain stages of simulation or test execution.

For example, let's consider a scenario where you want to inject new functionality into a driver without modifying its code. You can achieve this by adding the desired functionality in a pre-callback task or a post-callback task.

Here's an example:

```
task Driver::run;

  forever begin
```

```
    ...
    <pre_callback> // Calls the pre_callback function.
    transmit(tr);
  end
endtask


task pre_callback;
  // Code for the pre-callback functionality goes here.
endtask
```

By utilizing the pre_callback task as a callback, you can add the desired functionality without directly modifying the Driver task. This allows for flexibility and customization within the testbench.

13. What is the Factory pattern?

```
class Generator;
Transaction tr;
mailbox mbx;
tr = new;
task run;
repeat (10)
begin
assert(tr.randomize);
mbx.put(tr);
end
endtask
endclass
```

Bug: Here the object "tr" is constructed once outside the loop. Then "tr" is randomized and put into the mailbox "mbx". But the mailbox "mbx" holds only handles, not objects. Therefore, the mailbox contains multiple handles pointing to a single object. Here, the code gets the last set of random values.

Solution: The loop should contain the following steps:

1) Constructing an object.
2) Randomizing the object.
3) Putting it into the mailbox.

```
task run;
  repeat (10)
  begin
    Transaction tr = new (); // 1. Constructing
    assert (tr.randomize ()); // 2. Randomize
    mbx.put (tr); // 3. Putting into mailbox
  end
endtask
```

Bug: The run task constructs a transaction object and immediately randomizes it. This means the transaction "tr" uses whatever constraints are turned on by default.

Solution: Separate the construction of "tr" from its randomization by using a method called the "Factory Pattern".

Factory Pattern:

1) Construct a blueprint object.

2) Randomize this blueprint (It has correct random values).

3) Make a copy of this object.

4) Put it into the mailbox.

```
class Generator;
  mailbox mbx;
  Transaction blueprint = new (); // 1. Constructing the Blueprint
  task run;
    Transaction tr;
    repeat (10)
    begin
      assert (blueprint.randomize); // 2. Randomizing the Blueprint
      tr = blueprint.copy; // 3. Copying the blueprint
      mbx.put (tr); // 4. Putting into mailbox
    end
  endtask
endclass
```

14. Explain the difference between data types logic, reg, and wire.

WIRE:

1) Wire is just an interconnection between two elements which does not have any driving strength.
2) It is used for modeling combinational circuits as it cannot store a value.
3) Wire has a default value of "z" and gets values continuously from the outputs of devices it is connected to.

Example:

```
wire A;
assign A = b & c;
```

Note: Wire A is evaluated for every simulation delta time, so there is no need to store the value.

REG:

1) Reg is a 4-state unsigned variable that can hold a value and retains it until a new value is assigned.
2) Register data type can be used for modeling both combinational and sequential logic.
3) The default value for a register is "x" and it can be driven from initial and always blocks. Values of the register can be changed anytime in the simulation by assigning a new value to it.

Example:

```
reg A;

always @*

begin

 A = b & c;

end
```

Note: A is declared as a reg which can be evaluated only when there is a change in any of the signals in the sensitivity list. Reg needs to store the value until there is a change in the sensitivity list.

LOGIC:

1) Logic is a 4-state unsigned data type introduced in SystemVerilog.
2) It allows continuous assignments (e.g., assign crc = ~crc;) and can be used with gates and modules.
3) It is a variable-like declaration and does not look like a register declaration.
4) If you only make procedural assignments to logic, then it is semantically equivalent to reg.
5) A logic signal can be used anywhere a net is used, except when a logic variable is driven by multiple structural drivers (e.g., modeling a bidirectional bus).

Example:

```
module sample1;

  logic crc, d, q_out;

  logic clk, rst;


  initial
  begin
    clk = 1'b0; // procedural assignment
    #10 clk = 1'b1;
  end

  assign crc = ~crc; // continuous assignment
  and g1 (q_out, d); // q_out is driven by gate
  Flp_flop f1 (q, q_out, clk, rst); // q is driven by module
endmodule
```

15. What is the need for clocking blocks?

Clocking blocks ensure that signals are driven or sampled synchronously, allowing the testbench to interact with the signals at the appropriate timing. They help avoid race conditions between the testbench and the Design Under Test (DUT).

Example:

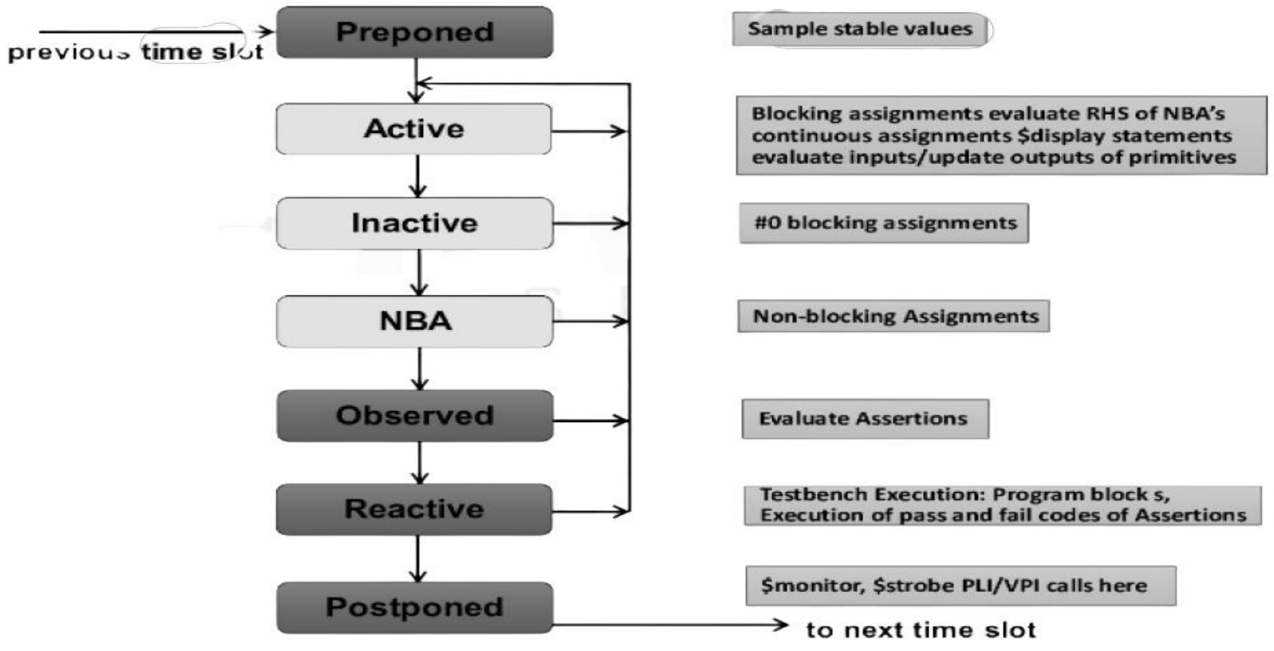
```
clocking cb @(posedge clk);
default input #1ns output #2ns; // Input skew and output skew
input grant; // Input from testbench to DUT
output request; // Output from DUT to testbench
endclocking
```

Note: Input signals (grant) are sampled at 1ns before the clock event, and output signals (request) are driven 2ns after the corresponding clock event. If skew is not specified, the default input skew is 1 step and the output skew is 0.

16. What are the ways to avoid race conditions between Testbench and RTL using SystemVerilog?

- The clock given to the DUT and the Testbench should have a phase difference.
- The DUT should work on the posedge of the clock while the Testbench should work on the negedge of the clock.
- Testbench output and DUT output pins should always be driven using non-blocking statements.
- Clocking blocks.
- Program blocks.

17. Explain event regions in SystemVerilog.



previous time slot
Preponed
Active
Inactive
NBA
Observed
Reactive
Postponed
Sample stable values
Blocking assignments evaluate RHS of NBA's continuous assignments $display statements evaluate inputs/update outputs of primitives
#0 blocking assignments
Non-blocking Assignments
Evaluate Assertions
Testbench Execution: Program block s, Execution of pass and fail codes of Assertions
$monitor, $strobe PLI/VPI calls here
to next time slot

18. What are the types of coverages?

The types of coverages can be categorized into Code Coverage and Functional Coverage.

- Code Coverage:

i. Code Coverage measures the quality of the testbench and inputs.
ii. It helps identify whether the code or the testbench is incorrect.
iii. Code Coverage measures how much of the code has been executed.
iv. Types of Code Coverage include Statement, Branch, Condition, Expression, Toggle, and FSM Coverage.

- Functional Coverage:

i. Functional Coverage is user-specified and aims to align the verification environment with the design's intent or functionality.
ii. Goals for functional coverage are derived from the functional specification.
iii. Unlike code coverage, functional coverage is not automatically inferred from the design.
iv. Functional Coverage is based on user-defined models.

In summary, code coverage focuses on assessing the quality and execution of the code, while functional coverage aims to capture the fulfillment of design intent and functional requirements. Both types of coverage are used to evaluate the effectiveness and progress of a verification project.

19. Can a constructor be qualified as protected or local in SV?

A constructor in SV cannot be qualified as protected or local. Constructors are always public by default in SystemVerilog. This means they can be accessed and called from anywhere in the program. It is not possible to restrict the visibility of a constructor to only certain subclasses (protected) or within the same module (local).

20. How to have a #delay statement that is independent of timescale? In Verilog, the #delay is dependent on the timescale.

To have a #delay statement that is independent of timescale in Verilog, you can use scaled time values. Here's an example:

Let's say you want a delay of 1 microsecond (1us) regardless of the timescale.

1) Determine the timescale in your Verilog code. For example, let's assume the timescale is set to 1ns/1ns.
2) Calculate the scaling factor by dividing the desired delay (1us) by the timescale (1ns):
   Scaling Factor = Desired Delay / Timescale
   = 1us / 1ns
   = 1000
3) Use the scaled time value in the #delay statement:
   #1000; // Delay of 1 microsecond (1us) independent of the timescale

By using the scaled time value, you can achieve a consistent delay of 1 microsecond regardless of the timescale setting.