INTERVIEW QUESTIONS

VERILOG

PART 2

JAIRAJ MIRASHI
DESIGN VERIFICATION ENGINEER

1. **How to generate a random number between 0.5 to 0.9?**

```
module Tb();
    real address;

    initial
    begin
        repeat (5)
        begin
            #1 address = 0.5 + ($urandom % 5) / 10.0;
            $display("address = %0.2f;", address);
        end
    end

endmodule
```

2. **How to get different random numbers in different simulations?**

To achieve different random numbers in different simulations in Verilog, you need to ensure that the random number generator is seeded differently for each simulation run. Here's a general approach to achieve this:

```
module Tb();
    real address;
    time seed;

    initial
    begin
        // Set a unique seed value for each simulation run
        //seed = $time;

        // Seed the random number generator
       // $random(seed);

        repeat (5)
        begin
          seed = $time;
           $random(seed);
            #1 address = 0.5 + ($random % 5) / 10.0;
            $display("address = %0.2f;", address);
        end
    end

endmodule
```

In this code, the $time system function gives the current simulation time in picoseconds. By using this value as the seed for the $random function, you're setting a different seed for each simulation run, which will result in a different sequence of random numbers.

Keep in mind that the effectiveness of this approach can depend on the simulation tool you are using and its support for these system functions. If you encounter any issues, consult your simulation tool's documentation or support resources to understand how to correctly set the seed for generating different random numbers in different simulations.

3. What is the difference between $sformat and $swrite?

The system task $sformat is similar to the system task $swrite, with one major difference. Unlike the display and write family of output system tasks, $sformat always interprets its second argument, and only its second argument, as a format string. This format argument can be a static string, such as "data is 0", or it can be a reg variable whose content is interpreted as the format string. No other arguments are interpreted as format strings. $sformat supports all the format specifiers supported by $display.

4. What is the difference between wire and reg?

Net types:

wire, tri: Physical connection between structural elements. Value assigned by continuous assignment or gate output.

Register type:

reg, integer, time, real, real time: Represents abstract data storage element. Assigned values only within an always statement or an initial statement.

The main difference between wire and reg is that wire cannot hold (store) the value when there's no connection between a and b like "a----------b". If there is no connection between a and b, a wire loses its value. On the other hand, a reg can hold the value even if there is no connection.

Default values: wire is 'Z', reg is 'x'.

5. What happens if a port is unconnected?

Unconnected input ports initialize to 'z' and if that value is fed into the component, it can cause problems. More commonly, redundant or unwanted outputs are left unconnected to be optimized away in synthesis.

6. What is the difference between === and ==?

The output of "==" can be 1, 0, or 'x'. The output of "===" can only be 0 or 1. When comparing two numbers using "==", if one or both of the numbers have 'x' bits, the output would be 'x'. But when using "===", the output would be 0 or 1.

For example:

A === B will give 0 as output.

A == B will give 'x' as output.

7. What is the difference between case, casex, and casez?

case treats only 0 or 1 values in case alternatives and does not deal with don't-care conditions.

casex treats all 'x' and 'z' values in case alternatives as don't-care conditions.

casez treats all 'z' values in case alternatives as don't-care conditions.

8. What is the difference between @(a or b) and @(a | b)?

In Verilog, @(a or b) and @(a | b) are two different ways to trigger a simulation event based on the values of signals a and b. However, there is no practical difference between them; they both achieve the same result.

The @(a or b) syntax uses the logical operator or to specify that the simulation event will be triggered when either signal a or signal b changes value. Similarly, the @(a | b) syntax uses the bitwise OR operator |, but in Verilog, the bitwise OR operation is the same as the logical OR operation for single-bit signals, so it's equivalent to @(a or b).

Here's an example to illustrate the equivalence:

```
module EventExample;
    reg a, b;
    integer time_counter;

    always @(a or b) begin
        $display("@(a or b) event triggered at time %0t", $time);
        time_counter = time_counter + 1;
    end

    always @(a | b) begin
        $display("@(a | b) event triggered at time %0t", $time);
        time_counter = time_counter + 1;
    end

    initial begin
        a = 0; b = 0;
        time_counter = 0;

        // Simulate changing values of a and b
        repeat (10) begin
            #10;
            a = !a;
            #15;
            b = !b;
        end

        // Display the total number of triggered events
        $display("Total triggered events: %d", time_counter);
        $finish;
    end
endmodule
```

9. What is a full case and a parallel case?

In Verilog, "full case" and "parallel case" are terms used to describe different ways of writing case statements to describe the behavior of a specific input condition. These terms pertain to the structure and style of the case statement.

Full Case:

A "full case" style of writing a case statement includes all possible input conditions explicitly. In other words, each possible value of the control expression is listed along with its corresponding set of actions. If any value is not explicitly listed, it is assumed to have no effect.

case (ctrl)

2'b00: action_00;

2'b01: action_01;

2'b10: action_10;

2'b11: action_11;

endcase

In this example, all four possible 2-bit values of ctrl (00, 01, 10, and 11) are listed along with their respective actions.

Parallel Case:

A "parallel case" style of writing a case statement groups together actions that share common behavior for certain input conditions. In other words, multiple input values may result in the same set of actions, and those actions are grouped together.

Here's an example of a parallel case statement:

case (ctrl)

2'b00, 2'b01: action_0x; // Both 00 and 01 cases trigger action_0x

2'b10: action_10;

2'b11: action_11;

endcase

In this example, both 2'b00 and 2'b01 cases trigger the action_0x behavior.

In summary:

"Full case" style includes all possible input conditions explicitly.

"Parallel case" style groups together input conditions that have the same behavior.

Example that demonstrates both "full case" and "parallel case" styles using a case statement:

```
module CaseExample;
    reg [1:0] ctrl;

    initial begin
        ctrl = 2'b01;

        // Full Case Style
        case (ctrl)
            2'b00: $display("Full Case: Control is 00");
            2'b01: $display("Full Case: Control is 01");
```

```
            2'b10: $display("Full Case: Control is 10");
            2'b11: $display("Full Case: Control is 11");
        endcase

        // Parallel Case Style
        case (ctrl)
            2'b00, 2'b01: $display("Parallel Case: Control is 00 or 01");
            2'b10: $display("Parallel Case: Control is 10");
            2'b11: $display("Parallel Case: Control is 11");
        endcase
    end
endmodule
```

Full Case Style:

Each possible value of ctrl is listed explicitly along with a corresponding display message.

When ctrl is 2'b00, it triggers the message "Full Case: Control is 00".

Similarly, for other values of ctrl.

Parallel Case Style:

Here, we group together values 2'b00 and 2'b01 under one case condition.

When ctrl is either 2'b00 or 2'b01, it triggers the message "Parallel Case: Control is 00 or 01".

Similarly, for other values of ctrl.

10. What is the difference between compiled, interpreted, event-based, and cycle-based simulators?

- Compiled Simulators:
- Compiled simulators convert the Verilog code into an optimized binary representation before simulation.
- The conversion process involves translating the code into machine code or an intermediate representation for efficient execution.
- Compiled simulators tend to be faster than interpreted simulators since they avoid interpreting the source code directly during simulation.
- They are well-suited for larger designs and extensive simulations.
- Examples include Synopsys VCS and Cadence Incisive.

- Interpreted Simulators:
  - Interpreted simulators execute the Verilog code directly without prior compilation.
  - They read the source code line by line during simulation and execute corresponding actions.
  - Interpreted simulators are generally slower than compiled simulators due to the lack of pre-compiled optimization.
  - They are useful for smaller designs and quick debugging since they don't require compilation.
  - Examples include ModelSim and QuestaSim.

- Event-Based Simulators:
  - Event-based simulators focus on simulating changes in signal values that trigger events.
  - They maintain an event queue and simulate hardware behavior based on the order of these events.
  - Event-based simulators are more accurate in modeling signal-level interactions and timing delays.
  - They handle events occurring at different times and in different orders.
  - Examples include ModelSim and QuestaSim.

- Cycle-Based Simulators:
  - Cycle-based simulators focus on simulating at a higher level of abstraction, considering signal values only at discrete time steps (cycles).
  - Each cycle represents a unit of simulation time during which signals can change.
  - They are faster than event-based simulators because they eliminate the need to track and process individual events.
  - Timing accuracy can be lower compared to event-based simulators.
  - Examples include NCSim and XSIM.

In summary:

Compiled simulators are optimized and fast but require an initial compilation step.

Interpreted simulators are flexible for small designs and debugging but can be slower.

Event-based simulators offer accurate signal-level simulation, handling events efficiently.

Cycle-based simulators provide higher-level abstraction for faster simulation but with potentially lower timing accuracy.

11. What data types can be used for input ports, output ports, and inout ports?

Input ports, output ports, and inout ports of a module can use various data types to represent the signals being passed between different modules. Here are the commonly used data types for each type of port:

Input Ports:

wire: Wires are used for unidirectional communication from the calling module to the called module. They can be used for any digital signal, such as control signals, data signals, or clock signals.

Output Ports:

wire: Just like with input ports, wires can be used for unidirectional communication from the called module to the calling module. They are commonly used for transmitting digital signals.

Inout Ports:

wire: Inout ports are used for bidirectional communication, allowing the calling and called modules to communicate in both directions. Wires are often used for inout ports when signals can flow in either direction.

reg: Registers can also be used for inout ports, typically when the inout signal needs to be latched or stored. This is common for bidirectional buses where data is read or written in specific clock cycles.

tri: Tri-state buffers are used in conjunction with wire or reg to create bidirectional communication. Tri-state signals can be actively driven low, driven high, or put in a high-impedance (Z) state.

Remember that the choice of data type depends on the nature of the signal being passed and the requirements of the design. For basic digital communication, wire is the most commonly used data type for input, output, and inout ports.

12. What is the functionality of a trireg?

A trireg in Verilog is a data type used to model the behavior of bidirectional buses. It represents a signal that can be actively driven low (0), actively driven high (1), or put into a high-impedance (Z) state. Tri-state signals are often used to connect multiple devices to a common bus, allowing only one device to drive the bus at a time while others are in a high-impedance state, preventing conflicts on the bus.

13. What is the functionality of tri1 and tri0?

- tri1 Functionality:
  - Represents a bidirectional signal that can be actively driven low (0) or put into a high-impedance (Z) state.
  - When actively driven low, it provides a low logic value.
  - Used for signals that are actively pulled low and can be shared among multiple devices using tri-state behavior.

- tri0 Functionality:
  - Represents a bidirectional signal that can be actively driven high (1) or put into a high-impedance (Z) state.
  - When actively driven high, it provides a high logic value.
  - Used for signals that are actively pulled high and can be shared among multiple devices using tri-state behavior.

14. Difference between conditional compilation and $plusargs?

- Conditional Compilation:
  - Conditional compilation involves including or excluding parts of the code during compilation based on predefined conditions.
  - It's controlled by preprocessor directives (ifdef, ifndef, else, elsif, endif) that evaluate compile-time constants or macros.
  - It allows you to create different versions of the code for different scenarios without affecting runtime behavior.
  - Used for enabling or disabling code sections based on compile-time conditions.

- $plusargs:
  - $plusargs is a system function in Verilog used to provide runtime configuration inputs to the simulation.
  - It allows users to pass command-line arguments to the simulation tool, which can then be read by the Verilog code.

- $plusargs provides flexibility to modify simulation behavior without recompiling the code.
- Used for passing runtime parameters to the simulation, such as test configurations or initial values.

15. What is the benefit of using Behavioral modeling style over RTL modeling?

Benefits of Using Behavioral Modeling Style over RTL Modeling in Verilog:

- Abstraction Level: Higher-level, abstract representation of functionality.
- Design Exploration: Enables quick iteration and exploration of design options.
- Ease of Modification: Easier to modify and adapt to changing requirements.
- Readability and Maintainability: More readable and manageable code.
- Simulation Efficiency: Faster simulation due to higher-level abstraction.
- Rapid Prototyping: Allows rapid prototyping and early functional verification.
- Portability and Reusability: More portable and reusable across implementations

16. What is the difference between a task and a function?

- Task:
  - Tasks are capable of enabling a function as well as enabling other versions of a task
  - Tasks also run with a zero simulation however they can if required be executed in a non zero simulation time.
  - Tasks are allowed to contain any of these statements.
  - A task is allowed to use zero or more arguments which are of type output, input or inout.
  - A Task is unable to return a value but has the facility to pass multiple values via the output and inout statements .

- Function:
  - A function is unable to enable a task however functions can enable other functions.
  - A function will carry out its required duty in zero simulation time. (The program time will not be incremented during the function routine)
  - Within a function, no event, delay or timing control statements are permitted
  - In the invocation of a function their must be at least one argument to be passed.
  - Functions will only return a single value and can not use either output or inout statements.

17. What is the difference between a static function and an automatic function?

Static function has module scope, can't access local variables outside its module. Can use $Monitor and $Strobe on local variables.
Automatic function has function scope, is more memory efficient, and cannot be used with $Monitor and $Strobe.

18. What is the advantage of wired AND and wired OR over wire?

Wired AND: This construct is used for open collector or open emitter designs. It allows multiple drivers to pull the line low, but the line is left floating (high-impedance state) when none of the drivers are active. Wired AND doesn't directly provide logical conflict resolution but is more about combining multiple pull-down paths.

Wired OR: Wired OR allows multiple drivers to pull the line high. When none of the drivers are active, the pull-up resistor connected to the wire ensures a defined high state. Wired OR is often used for open drain designs. Similarly, like Wired AND, Wired OR doesn't provide logical conflict resolution but enables multiple pull-up paths.

It's important to note that logical conflict resolution is generally not achieved through wired AND or wired OR constructs themselves, but through additional logic or design practices in more complex circuitry.

19. Identify the error in the following code.
a[7:0] = {4{'b10}};

The error in the provided Verilog code is the incorrect usage of the replication operator ({}). The replication operator is used to replicate a pattern a specified number of times. In this case, it seems like you want to create a 4-bit value where each 2-bit chunk is 2'b10.

Here's the corrected code:

```
a[7:0] = {4{2'b10}};
```

In this corrected version, the replication operator {4{2'b10}} replicates the 2-bit pattern 2'b10 four times to create an 8-bit value.

```
module ReplicationExample;
    reg [7:0] a;

    initial begin
        a = {4{2'b10}};
        $display("a = %b", a);
        $finish;
    end
endmodule
```

20. What is the difference between initial and always blocks?

- Initial Block:

- Functionality: An initial block is used to specify initial conditions and execute statements at the beginning of simulation.
- Execution Timing: The statements inside an initial block are executed only once at the start of the simulation.
- Usage: Typically used for initializing variables, displaying messages, or setting up initial testbench conditions.

- Always Block:

- Functionality: An always block is used to define continuous or sequential behavior based on sensitivity lists.
- Execution Timing: The statements inside an always block are executed repeatedly based on the conditions specified in the sensitivity list.
- Usage: Used for defining sequential logic, combinational logic, or modeling hardware behavior based on clock edges or signal changes.

In summary, initial blocks are used for simulation setup and one-time actions at the beginning of simulation, while always blocks are used to model ongoing behavior based on specific conditions or signal changes.