

SYSTEM VERILOG

#1



JAI RAJ MIRASHI
DESIGN VERIFICATION
ENGINEER



1. What is the difference between an initial and final block of the SystemVerilog?

The initial block occurs at the start of simulation whereas the final block occurs at the end of the simulation without having any delays. So, it is the same as a function call that can execute in zero simulation time.

The main usage of the final block is to display statistical information about the simulation.

Syntax:

```
final begin
    ...
end
```

Example:

```
module final_example;
    initial begin
        $display("Inside initial block");
        #10;
        $display("Before calling $finish");
        $finish;
    end

    final begin
        $display("Inside final block at %0t", $time);
    end
endmodule
```

Output:

```
Inside initial block
Before calling $finish
$finish called from file "testbench.sv", line 8.
Inside final block at 10
$finish at simulation time 10
```

2. Explain the simulation phases of SystemVerilog verification?

1. Build Phase:

➤ Generate Configuration:

Randomize the configuration of the Device Under Test (DUT) and its surrounding environment.

➤ Build Environment:

Allocate and interconnect the testbench components based on the generated configuration. Testbench components are specific to the testbench and differ from physical components in the design that are built using Register-Transfer Level (RTL) logic.

➤ Reset the DUT:

Initialize the Device Under Test, bringing it to a predefined starting state.

➤ Configure the DUT:

Load the DUT command registers with values determined by the generated configuration in the first step.

2. Run Phase:

➤ Start Environment:

Activate the testbench components such as Bus Functional Models (BFMs) and stimulus generators.

➤ Run the Test:

Commence the test and await its completion. While it is straightforward to identify the completion of a directed test, determining the conclusion of a random test can be intricate. To facilitate this, utilize the testbench layers as a reference. Start from the topmost layer, ensuring input draining from the preceding layer (if any), verifying the current layer's idle status, and awaiting the readiness of the subsequent lower layer. Employ timeout checkers to prevent potential lockups in the DUT or testbench.

3. Wrap-up Phase:

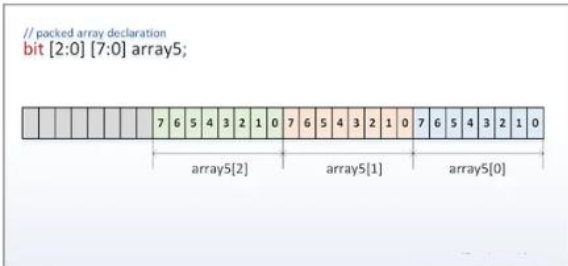
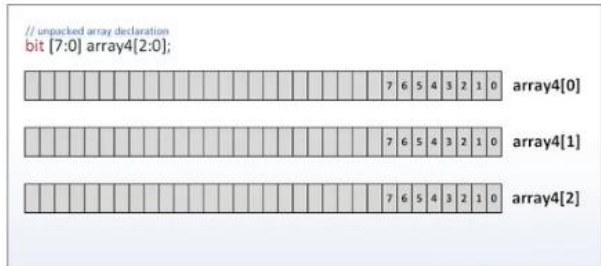
➤ Sweep:

Once the lowest layer completes its tasks, wait for the final transactions to be processed and drained out of the DUT.

➤ Report:

Once the DUT reaches an idle state, examine the testbench for any lost data. Occasionally, transactions may remain in the scoreboard without being processed, potentially due to them being dropped by the DUT. Armed with this information, compile the final report indicating the test's success or failure. In the event of a failure, ensure the deletion of any functional coverage results, as they might not accurately represent the test outcomes.

3. What is the Difference between SystemVerilog packed and unpacked array?

Packed Array	Unpacked Array
Packed arrays encompass single-bit data types (reg, logic, bit), enumerated types, and recursively packed arrays and structures.	Unpacked arrays can be of any data type and are declared by specifying element ranges after the identifier name.
A one-dimensional packed array is termed a vector, representing a multi-bit data object defined within a specified range.	They may or may not be represented as contiguous sets of bits, offering flexibility in data organization.
Packed arrays allow subdividing a vector into sub-fields, accessible as array elements.	
They are guaranteed to be represented as contiguous sets of bits.	
Example: <code>bit [2:0] [7:0] array5; //</code> Packed array declaration	Example: <code>bit [7:0] array4[2:0]; //</code> Unpacked array declaration
	

4. What is “This ” keyword in the systemverilog?

The this keyword in SystemVerilog is used to refer to class properties, parameters, and methods of the current instance. It can only be utilized within non-static methods, constraints, and covergroups. Essentially, this serves as a predefined object handle that points to the object used to invoke the method in which it is employed.

EXAMPLE:

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

    function new(int addr, int b);
        data = b;
        // addr = addr; // Ambiguity in assignment & compiler gets confused
        both in global and local same variable name
        this.addr = addr; // Explicitly assigns local variable addr to the
        addr variable in Packet class
    endfunction
endclass
```

```

        endfunction
    endclass

    initial
    begin
        Packet p = new(10, 20); // Creates an instance of the Packet class with
        addr=10 and data=20
    end

```

5. What is alias in SystemVerilog?

In SystemVerilog, an alias is a named reference to a variable, signal, or instance. It provides a way to refer to a variable using a different name. Aliases are valuable for enhancing code readability, reducing complexity, and improving simulation performance. They can be employed within modules, interfaces, and generate blocks, offering flexibility and clarity in code design.

Example:

```

logic [7:0] data;
alias mydata = data; // alias "mydata" for signal "data"

initial
begin
    mydata = 8'hFF; // assign the value to "data" using the alias "mydata"
end

```

In this example, the alias keyword establishes the alias "mydata" for the signal "data." By using the alias, you can manipulate the signal with a different name, enhancing code understandability and maintainability.

6. randomized in the systemverilog test bench?

In SystemVerilog testbenches, the randomize method is often used to generate random values for variables and class properties. It is a built-in method provided by SystemVerilog to simplify the randomization process during simulation.

When a variable or a class property is declared with the rand keyword, it indicates that the variable can be randomized. The randomize method can then be called on instances of these variables or class objects to assign them random values based on their constraints.

Example:

```

class MyClass;
    rand int data;
    rand bit [7:0] address;
endclass

```

```

module testbench;
    MyClass obj;

    initial begin
        // Creating an instance of MyClass
        obj = new();

        // Randomizing variables using randomize method
        if (obj.randomize()) begin
            // Randomization successful
            $display("Randomized data: %0d, address: %h", obj.data, obj.address);
        end else begin
            // Randomization failed due to constraints
            $display("Randomization failed! Constraints not met.");
        end
    end
endmodule

```

7. In SystemVerilog which array type is preferred for memory declaration and why?

In SystemVerilog, the preferred array type for memory declaration is an associative array. Associative arrays are favored because they offer efficiency in storing data at random address locations. Unlike dynamic arrays, associative arrays do not necessitate pre-allocation of all addresses in memory before usage. This flexibility allows for dynamic allocation of memory based on the specific addresses used during simulation, optimizing memory utilization and enhancing simulation performance.

Example:

```

module MemoryExample;
    // Declare an associative array for memory storage
    int memory[string];

    // Initialize and access the associative array
    initial begin
        // Store data at random addresses in the memory
        memory["Address1"] = 10;
        memory["Address2"] = 20;
        memory["Address3"] = 30;

        // Retrieve data from specific addresses
        $display("Data at Address1: %0d", memory["Address1"]);
        $display("Data at Address2: %0d", memory["Address2"]);
        $display("Data at Address3: %0d", memory["Address3"]);
    end
endmodule

```

8. How to avoid race round condition between DUT and test bench in SystemVerilog verification?

In SystemVerilog verification, race conditions between the Design Under Test (DUT) and the testbench can be avoided by using the program construct. The program construct ensures a race-free interaction between the design and the testbench. Here are the key points to consider:

Program Block Usage:

The program block provides a race-free interaction between the design and the testbench.

All elements declared within the program block execute in the Reactive region.

Non-blocking assignments within the module are scheduled in the active region.

Initial blocks within program blocks are scheduled in the Reactive region.

Syntax of Program Block:

```
program test(input clk, input [7:0] addr, output [7:0] wdata);  
    // ...  
endprogram  
  
//or//  
  
program test (interface mem_intf);  
    // ...  
endprogram
```

Example-1: Using Module Block (Race Condition Occurs)

```
module design_ex(output bit [7:0] addr);  
    initial begin  
        addr <= 10;  
    end  
endmodule  
  
module testbench(input bit [7:0] addr);  
    initial begin  
        $display("\t Addr = %0d", addr); // Displays Addr = 0 (Race  
Condition)  
    end  
endmodule  
  
module tbench_top;  
    wire [7:0] addr;  
    design_ex dut(addr);  
    testbench test(addr);  
endmodule
```

Example-2: Using Program Block (Race Condition Avoided)

```

module design_ex(output bit [7:0] addr);
    initial begin
        addr <= 10;
    end
endmodule

program testbench(input bit [7:0] addr);
    initial begin
        $display("\t Addr = %0d", addr); // Displays Addr = 10 (Race
Condition Avoided)
    end
endprogram

module tbench_top;
    wire [7:0] addr;
    design_ex dut(addr);
    testbench test(addr);
endmodule

```

By using the program construct as shown in Example-2, the race condition is avoided, ensuring reliable communication between the DUT and the testbench in SystemVerilog verification.

9. What are the advantages of the systemverilog program block?

- Race-Free Interaction: Ensures synchronized and race-free communication between testbench and Design Under Test (DUT).
- Explicit Timing Control: Code executes in the Reactive region, allowing precise timing control for operations.
- Enhanced Readability: Improves code organization and clarity, making the testbench structure more understandable.
- Simplified Connectivity: Facilitates easy and error-free port connections, similar to module instantiation.
- Efficient Debugging: Streamlines debugging efforts by isolating specific testbench functionalities for focused troubleshooting.
- Modular Code Organization: Encourages modular and hierarchical organization, enhancing code reusability and management.
- Improved Simulation Performance: Prevents unnecessary race conditions, contributing to efficient and accurate simulation runs.
- Prevents Design Pollution: Maintains a clear separation between testbench logic and the design, preventing unintended interference.

10. What is the difference between logic and bit in SystemVerilog?

❖ **logic Data Type:**

Introduced in SystemVerilog, logic is a versatile data type that can model both wires and state information (reg).

It can hold 0, 1, x (unknown), and z (high-impedance) values, making it suitable for multiple states.

logic can be used for modeling both combinational and sequential logic.

It allows multiple drivers, and the last assignment takes precedence.

❖ **bit Data Type:**

bit is a traditional 2-state data type in Verilog and SystemVerilog, representing binary values 0 and 1.

Unlike logic, bit does not handle unknown (x) or high-impedance (z) states.

bit is typically used for modeling basic binary values in digital logic circuits.

11. What is the difference between datatype logic and wire?

❖ **logic Data Type:**

Introduced in SystemVerilog, logic is a versatile 4-state data type that can represent binary values (0 and 1) as well as unknown (x) and high-impedance (z) states.

logic is used to model both combinational and sequential logic elements.

It allows multiple drivers, and the last assignment takes precedence.

Can be used to model both state elements (like reg) and wires, providing a unified data type for various purposes.

❖ **wire Data Type:**

wire is a 4-state data type in SystemVerilog used specifically for connecting different elements in a design.

Wires cannot hold values on their own; they are used to connect outputs and inputs, enabling data flow between modules and components.

Wires cannot be assigned values inside procedural blocks; they are driven by continuous assignments, module outputs, or other wire assignments.

Unlike logic, wire is primarily used for interconnecting signals rather than representing internal storage elements.

12. What is a virtual interface?

A virtual interface in SystemVerilog is a variable that represents an interface instance. It allows dynamic access to the interface signals within classes, providing flexibility in connecting and accessing interface elements.

Dynamic Access: Represents an interface instance, enabling dynamic access to interface signals.

Initialization: Must be initialized and connected to an actual interface instance before use.

Class Properties: Can be declared as properties within classes, facilitating dynamic connections.

Dynamic Connections: Can be passed as arguments to tasks, functions, or methods for flexible connections.

Access to Interface Elements: All interface variables and methods can be accessed using a virtual interface handle (e.g., `virtual_interface.variable`).

Multiple Instances: One virtual interface variable can represent different interface instances, allowing dynamic switching.

Allowed Operations: Permits assignment and equality operations with similar virtual interfaces and interface instances.

Example Usage:

```
virtual interface intf vif;

// Constructor in class or module
function new(virtual intf vif);
    this.vif = vif;
endfunction

// Accessing interface signals using virtual interface handle
vif.a = 6;
vif.b = 4;

$display("Value of a = %0d, b = %0d", vif.a, vif.b);
```

In summary, a virtual interface in SystemVerilog provides a dynamic and flexible mechanism for accessing interface signals within classes and modules, allowing runtime connectivity and interaction with different interface instances during simulation.

13. What is an abstract class?

An abstract class in SystemVerilog, declared with the `virtual` keyword, serves as a blueprint for sub-classes. It defines the structure and interface that derived classes must follow. Abstract classes have the following characteristics:

Prototype Definition: Sets the prototype for sub-classes, outlining methods and properties they should implement.

Cannot be Instantiated: Abstract classes cannot be instantiated directly; they only provide a template for derived classes.

Contains Unimplemented Methods: Can include methods with only prototypes and no implementations, defining the method signatures without specifying the functionality.

Example:

```
// Abstract class definition
virtual class packet;
    bit [31:0] addr;
    // Abstract method declaration with no implementation
    virtual task display();
endtask
endclass

// Derived class implementing the abstract class
class extended_packet extends packet;
    // Implementation of the abstract method
    task display();
        $display("Value of addr is %0d", addr);
    endtask
endclass
```

In the given example, `packet` is an abstract class defining the `addr` property and an abstract method `display()`. The `extended_packet` class extends `packet` and provides an implementation for the `display()` method. Abstract classes ensure a consistent interface across derived classes, enforcing a specific structure while allowing flexibility in implementations.

14. What is the difference between \$random and \$urandom?

\$urandom() returns an unsigned 32-bit random number.

\$urandom() accepts an optional seed argument that determines the sequence of random numbers generated.

\$random() returns a signed 32-bit random number.

\$random() does not accept a seed argument, making it purely pseudorandom without seed-based predictability.

Example:

```
module random_example;
    bit [31:0] random_unsigned1;
    bit [31:0] random_unsigned2;
    int random_signed1;
    int random_signed2;

    initial begin
        random_unsigned1 = $urandom(); // Generates a random unsigned 32-bit
number
        random_unsigned2 = $urandom(89); // Generates a random unsigned 32-bit
number with seed 89
        random_signed1 = $random(); // Generates a random signed 32-bit number
        random_signed2 = $random(); // Generates another random signed 32-bit
number

        $display("Random Unsigned 1: %0d", random_unsigned1);
        $display("Random Unsigned 2: %0d", random_unsigned2);
        $display("Random Signed 1: %0d", random_signed1);
        $display("Random Signed 2: %0d", random_signed2);
    end
endmodule
```

In this example, both \$urandom() and \$random() functions are used to generate random unsigned and signed 32-bit numbers respectively. The \$urandom() function demonstrates the usage of a seed (89) for predictable random number generation, while \$random() generates random signed numbers without a seed.

15. What is the expect statements in assertions?

The expect statement is a procedural blocking statement used in assertions. Similar to the assert statement, it blocks execution until the specified property or sequence is evaluated. It must be used within procedural blocks like always, initial, tasks, or functions.

Syntax:

```
expect (property or sequence) <statements>
```

Example:

```
initial begin
    #100;
    expect (@(posedge clk) req1 ##2 req2) else $error("Expect failure");
    // <statements>;
end
```

In this example, after 100 time units, the expect statement waits for a positive edge of the clock where req1 is true, followed by 2 clock cycles where req2 is true. If this sequence occurs, the expect statement passes; otherwise, it fails, and subsequent <statements> are not executed until the specified conditions are met.

16. What is DPI?

DPI allows seamless communication between foreign languages (like C) and SystemVerilog. It comprises a foreign language layer and a SystemVerilog layer, bridging the gap between languages and enabling heterogeneous systems with existing codebases.

1) Import Method:

- SystemVerilog calls functions/tasks implemented in a foreign language (import methods).
- ``import "DPI-C" function <return_type> <function_name> (<arguments if any>)``
- Example:

```
import "DPI-C" function void addition(int a, int b);
```

2) Export Method:

- Foreign languages call functions/tasks implemented in SystemVerilog (export methods).
- ``export "DPI-C" function <function_name>``
- Example:

```
export "DPI-C" function addition;
```

Step-by-Step Examples:

1. Import Method (SystemVerilog calls C **function**):

```
// C file (addition.c)
#include <stdio.h>
void addition(int a, int b) {
    printf("Addition of %0d and %0d is %0d\n", a, b, a+b);
}

// SystemVerilog file (tb.sv)
module tb;
    import "DPI-C" function void addition(int a, int b);
    initial begin
        $display("Before add function call");
        addition(4, 5);
        $display("After add function call");
    end
endmodule
```

2. Export Method (C calls SystemVerilog **function**):

```
// SystemVerilog file (tb.sv)
module tb;
    export "DPI-C" function addition; // Exporting SystemVerilog function
    import "DPI-C" context function void c_caller();

    function void addition(int a, b);
        $display("Addition of %0d and %0d is %0d\n", a, b, a+b);
    endfunction

    initial begin
        c_caller();
    end
endmodule

// C file (c_caller.c)
#include <stdio.h>
extern void addition(int a, int b);

void c_caller() {
    printf("Calling addition function from c_caller\n");
    addition(4, 5);
}
```

Output:

```
Before add function call
Addition of 4 and 5 is 9
After add function call
Calling addition function from c_caller
Addition of 4 and 5 is 9
```

In these examples, DPI enables communication between SystemVerilog and C, showcasing the import and export methods, allowing seamless integration of functionalities from different languages.

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

1) == (Logical Equality):

- `a == b` checks for logical equality between signals `a` and `b`.
- Evaluates to 1 (true) only if both `a` and `b` are equal to logic values of 0 or 1.
- If either `a` or `b` is X or high-Z, the expression evaluates to 0 (false).

2) === (Case Equality):

- `a === b` checks for case equality, considering all 4 logic states: 0, 1, X (unknown), and high-Z.
- Evaluates to 1 (true) even if either `a` or `b` are unknown values (X).

18. What are the system tasks?

1) Display System Tasks:

`$display`: Displays strings, variables, and expressions immediately in the active region.

`$monitor`: Monitors signal values upon changes and executes in the postpone region.

`$write`: Displays strings, variables, and expressions without appending a newline, in the active region.

`$strobe`: Displays strings, variables, and expressions at the end of the current time slot (postpone region).

Format Specifiers: Used to print values in different formats (%d, %b, %h, %o, %c, %s, %t, %f, %e).

2) Monitoring Control System Tasks:

`$monitoron`: Turns on monitoring during simulation.

`$monitoroff`: Turns off monitoring during simulation.

3) Simulation Controlling System Tasks:

`$stop`: Stops or suspends the running simulation, putting it in interactive mode for debugging.

`$finish`: Terminates the simulation.

4) Random Number Generator System Task:

`$random`: Returns a 32-bit signed integer (positive or negative).

Example:

```
module display_tb;
    reg [3:0] d1, d2;
    initial begin
        d1 = 4; d2 = 5;

        #5 d1 = 2; d2 = 3;
    end

    initial begin
        $display("At time %0t: {$display A} -> d1 = %0d, d2 = %0d", $time, d1,
d2);
        $monitor("At time %0t: {$monitor A} -> d1 = %0d, d2 = %0d", $time, d1,
d2);
        $write("At time %0t: {$write A} -> d1 = %0d, d2 = %0d", $time, d1, d2);
        $strobe("At time %0t: {$strobe A} -> d1 = %0d, d2 = %0d", $time, d1, d2);

        #5;

        $display("At time %0t: {$display B} -> d1 = %0d, d2 = %0d", $time, d1,
d2);
        // $monitor is missing -> Observe print for $monitor A
        $write("At time %0t: {$write B} -> d1 = %0d, d2 = %0d", $time, d1, d2);
        $strobe("At time %0t: {$strobe B} -> d1 = %0d, d2 = %0d", $time, d1, d2);
    end
endmodule
```

Let's break down the output based on the given code:

❖ At time 0:

\$display A: Displays the values of d1 and d2 using \$display. At this point, d1 = 4 and d2 = 5.

\$write A: Similar to \$display, but does not append a newline character. Displays d1= 4 and d2 = 5.

\$monitor A: Begins monitoring d1 and d2 using \$monitor. Initially, d1 = 4 and d2 = 5.

\$strobe A: Strobes (displays) the values of d1 and d2 at the end of the current time slot. Shows d1 = 4 and d2 = 5.

❖ At time 5:

\$display B: Displays the updated values of d1 and d2 after the time delay of 5 time units. Now, d1 = 2 and d2 = 3.

\$write B: Displays d1 = 2 and d2 = 3 without appending a newline character.

\$strobe B: Strobes the updated values of d1 and d2 at the end of the current time slot. Shows d1 = 2 and d2 = 3.

\$monitor A: Continues monitoring d1 and d2. Since they have been updated, the \$monitor statement displays the new values: d1 = 2 and d2 = 3.

19. What is SystemVerilog assertion binding and advantages of it?

SystemVerilog assertion binding allows assertions to be written in separate files, independent of the design code, and then bound to specific instances or all instances of the design module or interface. This approach provides modularity and flexibility in the verification process.

Syntax:

To Bind with Specific Instance:

```
bind <dut_specific_instance_path> <assertion_module> <instance>
```

To Bind with All Instances:

```
bind <dut_module> <assertion_module> <instance>
```

Example 1: Binding with Specific Instance:

```
// Assertion Module (assertion.sv)
module assertion_dff (input clk, rst_n, d, q);
    sequence seq1;
        d ##1 q;
    endsequence

    property prop;
        @(posedge clk) disable iff(rst_n)
            d |=> seq1;
    endproperty

    dff_assert: assert property (prop) else $display("Assertion failed at time
= %0t", $time);
endmodule

// Testbench Module (tb.sv)
module tb;
    reg clk, rst_n;
    reg d;
    wire q;

    D_flipflop dff1(clk, rst_n, d, q);

    // To Bind with Single Instance of DUT
    bind tb.dff1 assertion_dff single_inst(clk, rst_n, d, q);

    // Rest of the Testbench Code...

endmodule
```

Example 2: Binding with All Instances:

```
// Assertion Module (assertion.sv)
module assertion_dff (input clk, rst_n, d, q);
    // Assertion Logic...
endmodule

// Testbench Module (tb.sv)
module tb;
    reg clk, rst_n;
    reg d;
    wire q;

    D_flipflop dff1(clk, rst_n, d, q);
    D_flipflop dff2(clk, rst_n, d, q);

    // To Bind with All Instances of DUT
    bind D_flipflop assertion_dff all_inst(clk, rst_n, d, q);

    // Rest of the Testbench Code...

endmodule
```

Advantages:

Assertions organized in separate modules enhance code readability and organization.

Reusable assertions save development effort and promote efficient code reuse in diverse projects.

Reusable assertions save development effort and promote efficient code reuse in diverse projects.

Isolated debugging of assertion modules simplifies issue identification and speeds up bug resolution.

Design and verification tasks progress concurrently, accelerating the overall development process.

20. What are parameterized classes?

SystemVerilog Parameterized Classes allow you to create customizable class templates similar to parameterized modules in Verilog. Parameters act as local constants within the class, defining a set of attributes. Default values can be overridden during instantiation, a feature known as parameter overriding.

```
class packet #(parameter int ADDR_WIDTH = 32, DATA_WIDTH = 32);
    bit [ADDR_WIDTH-1:0] address;
    bit [DATA_WIDTH-1:0] data;

    function new();
        address = 10;
        data = 20;
    endfunction
endclass
```

In the above example, packet class is defined with default ADDR_WIDTH and DATA_WIDTH values. When instantiated without parameters, it uses the defaults:

```
packet pkt; // Creates pkt handle with default ADDR_WIDTH = 32 and
DATA_WIDTH = 32.
```

You can override default values during instantiation:

```
packet #(32, 64) pkt; // Creates pkt handle with ADDR_WIDTH = 32 and
DATA_WIDTH = 64.
```

Additionally, you can parameterize the data type itself:

```
class packet #(parameter type T = int);
    T address;
    T data;

    function new();
        address = 10;
        data = 20;
    endfunction
endclass
```

In this case, the data type can be specified during instantiation:

```
packet pkt; // Address and data type is int.
packet #(bit [31:0]) pkt; // Address and data type is bit [31:0].
```

21. How to generate array without randomization?

In SystemVerilog, you can generate an array without using randomization by manually assigning values to its elements. Here's a simple example demonstrating how to create an array and initialize its elements without relying on randomization:

```
module testbench;
    int unsigned data[10]; // Declare an integer array of size 10

    initial begin
        // Initialize the array elements using a loop
        for (int i = 0; i < 10; i++) begin
            data[i] = i; // Assign values to array elements (e.g., sequential
            integers)
            // You can also perform other computations or assign specific values as
            needed.
        end

        // Display the initialized array elements
        $display("Initialized Array:");
        for (int i = 0; i < 10; i++) begin
            $display("data[%0d] = %0d", i, data[i]);
        end
    end
endmodule
```

In this example, an integer array `data` of size 10 is declared. The initial block contains a loop that iterates from 0 to 9, assigning sequential integers to the array elements. You can modify the assignment statement within the loop to set specific values or perform computations to populate the array according to your requirements.

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

- `always_comb` is automatically triggered once at time zero, after all initial and `always` procedures have been started, whereas `always @ (*)` will be triggered first when any signal in the sensitivity list changes.
- `always_comb` is sensitive to changes of contents in a function where the latter is sensitive only to the arguments of the function when invoked inside it.
- `always_comb` cannot have statements that have delays or timing constructs in it.
- Variables used on the LHS inside an `always_comb` block cannot be assigned to in any other parallel process.

23. What is the difference between overriding and overloading?

1) Overriding:

Overriding occurs in the context of inheritance when a subclass provides a specific implementation for a method that is already defined in its superclass.

When a method in a subclass has the same name, return type, and input parameters as a method in its superclass, the subclass method overrides the superclass method.

Overriding allows a subclass to provide a specialized implementation of a method inherited from its superclass, providing a specific behavior tailored to the subclass.

Example:

```
class Superclass;
    virtual function void display();
    $display("Superclass display");
endfunction
endclass

class Subclass extends Superclass;
    virtual function void display();
    $display("Subclass display");
endfunction
endclass
```

In this example, the display method in the Subclass overrides the display method in the Superclass.

2) Overloading:

Overloading occurs when two or more methods or functions in the same class have the same name but different parameter lists (different number or types of parameters).

Methods or functions are overloaded within the same class, allowing them to perform different actions based on the number or types of parameters passed to them.

Overloading enables the class to provide multiple methods with the same name, offering flexibility in method usage.

Example:

```
class OverloadedMethods;  
    function void process(int a);  
        $display("Method with integer parameter: %0d", a);  
    endfunction  
  
    function void process(string b);  
        $display("Method with string parameter: %s", b);  
    endfunction  
endclass
```

In this example, the process method is overloaded within the OverloadedMethods class. One version takes an integer parameter, and the other takes a string parameter.

24. Explain the difference between deep copy and shallow copy?

Shallow Copy	Deep Copy
Shallow copy involves copying the handles (references) to objects, not the objects themselves. In other words, it duplicates the structure, but not the contents.	Deep copy involves creating new objects and recursively copying all the objects referred to by the original objects. It creates a complete, independent copy of the original structure and its contents.
In the given example, during a shallow copy, the handles to objects are copied from one instance to another. Changes made to the objects through one instance reflect in the other instance because they share the same underlying objects.	In the given example, a custom copy() method is implemented in Class B to perform a deep copy. This method creates new objects for properties, ensuring that changes made to one instance do not affect the other.
Shallow copy only copies the top-level properties of a class, not the objects referenced by those properties. In the example, changes to b2.a.j are reflected in b1.a.j since they point to the same object.	Deep copy creates entirely new instances of the objects, including all nested objects. Changes made to the properties of one instance do not affect the other instance, as they are independent copies.
<pre>class A; int j = 15; endclass class B; int i = 10; A a = new; endclass program main; B b1, b2;</pre>	<pre>class A; int j = 15; endclass class B; int i = 10; A a = new; function void copy(B obj_b); this.i = obj_b.i; this.a = new obj_b.a;</pre>

<pre> initial begin b1 = new; b2 = new b1; // Shallow copy of object b1 copies first level of properties to b2 b2.i = 20; b2.a.j = 30; // Assigns 30 to variable "j" shared by both b1 & b2 \$display("b1.i = %0d", b1.i); \$display("b2.i = %0d", b2.i); \$display("b1.a.j = %0d", b1.a.j); \$display("b2.a.j = %0d", b2.a.j); end endprogram </pre>	<pre> endfunction : copy endclass program main; B b1 = new, b2 = new; initial begin b2.copy(b1); b2.i = 20; b2.a.j = 30; \$display("b1.i = %0d", b1.i); \$display("b2.i = %0d", b2.i); \$display("b1.a.j = %0d", b1.a.j); \$display("b2.a.j = %0d", b2.a.j); end endprogram </pre>
<pre> //OUTPUT b1.i = 10 b2.i = 20 b1.a.j = 30 b2.a.j = 30 </pre>	<pre> //OUTPUT b1.i = 10 b2.i = 20 b1.a.j = 15 b2.a.j = 30 </pre>

25. What is interface and advantages over the normal way?

- Allows the number of signals to be grouped together and represented as a single port, the single port handle is passed instead of multiple signal/ports.
- Interface declaration is made once and the handle is passed across the modules/components.
- Addition and deletion of signals are easy.

26. What is modport and explain the usage of it?

Modport in SystemVerilog is used within an interface to declare port directions for signals, providing access restrictions and enhancing design clarity.

Syntax:

```
modport <name> (input <port_list>, output <port_list>);
```

Advantage of modport

Modport put access restriction by specifying port directions that avoid driving of the same signal by design and testbench.

Directions can also be specified inside the module.

Modport provide input, inout, output, and ref as port directions

Multiple modports can be declared for different directions for monitor and driver.

Examples:

```
modport TB (output a, b, en, input out, ack);  
modport RTL (input clk, reset, a, b, en, output out, ack);
```

27. What is a clocking block?

A clocking block in SystemVerilog is used to specify synchronization schemes and timing requirements for an interface. It defines a group of signals that are synchronized with a specific clock, allowing precise control over signal behavior concerning clock events.

Syntax:

```
clocking <clocking_name> (<clocking event>);  
    default input <delay> output <delay>;  
    <signals>  
endclocking
```

Advantages of clocking block

It provides a group of signals that are synchronous with a particular clock in DUT and testbench

Provides a facility to specify timing requirements between clock and signals.

Clocking event

To synchronize the clocking block, a clocking event is used. The clocking event governs the timing of all signals mentioned in that clocking block. All input or inout signals are sampled on the occurrence corresponding clock event. Similarly, output or inout are also driven by the occurrence of the corresponding clock event.

Clocking Event:

A clocking event determines the timing of all signals within the clocking block. Input and output signals are sampled on the occurrence of the corresponding clock event, while output or input signals are driven during the specified clock event.

Example:

```
@(posedge clk) or @(negedge clk)
```

Clocking Skew:

Clocking blocks allow setting input and output skews, indicating the time unit delay before sampling input signals and after driving output signals concerning the clock event.

```
default input #2 output #3;
```

Input skew: #2 (sampled 2 time units before the clocking event)

Output skew: #3 (driven 3 time units after the clocking event)

Note: Input and output skews can be parameters or constants, defaulting to 0 if not specified, and can be adjusted based on the timescale of the current scope.

28. What is the difference between the clocking block and modport?

Clocking Block:

Defines synchronization schemes and precise timing requirements for signals associated with a specific clock.

Specifies synchronization and timing control, ensuring signals align with specific clock edges.

Can be declared at the program, module, or interface level, providing flexibility in usage.

Focuses on signal synchronization and precise timing control.

Ensures synchronized behavior of signals concerning clock events.

Modport:

Declares port directions for signals within interfaces, providing access restrictions and control.

Defines different access modes (input, output, inout, ref) for signals within an interface.

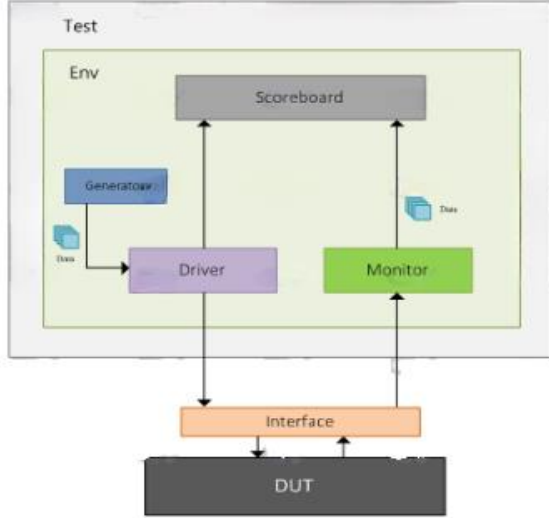
Specific to interfaces, used within interfaces to define port directions for signals.

Specifies access restrictions and directions for signals within interfaces.

Provides access control within interfaces, ensuring proper communication between modules.

29. What are the basic testbench components?

Component	Description
Generator	Generates diverse input stimuli that are applied to the Device Under Test (DUT) to assess its behavior under various conditions.
Interface	Description: Defines the design signals that can be either driven or monitored during the verification process. It acts as an interface between the testbench components and the DUT.
Driver	Responsible for transmitting the generated stimulus from the generator to the DUT. It applies the input signals based on the test scenarios.
Monitor	Observes the input and output ports of the DUT, capturing its activities and behavior during simulation. Monitors help in analyzing the responses of the DUT to the applied stimuli.
Scoreboard	Compares the output generated by the DUT with the expected behavior specified in the testbench. It checks whether the DUT's responses align with the anticipated results.
Environment	Houses all the verification components mentioned above, including the generator, interface, driver, monitor, and scoreboard. It orchestrates the interactions between these

	components, creating a cohesive environment for testing.
Test	Contains the entire environment and can be configured with various settings to conduct different test scenarios. It serves as the high-level entity that defines the overall testing procedure and conditions for the DUT.
 <pre> graph TD subgraph Test subgraph Env Generator[Generator] -- Data --> Driver[Driver] Driver --> Scoreboard[Scoreboard] Monitor[Monitor] -- Data --> Scoreboard end end Driver <--> Interface[Interface] Monitor <--> Interface Interface <--> DUT[DUT] </pre> <p>The diagram illustrates the architecture of a test environment. It is organized into three main layers. The top layer, labeled 'Test', contains an 'Env' (Environment) block. Inside 'Env', there is a 'Scoreboard' at the top, a 'Driver' on the left, and a 'Monitor' on the right. A 'Generator' provides 'Data' to the 'Driver'. The 'Driver' and 'Monitor' both send data to the 'Scoreboard'. The 'Driver' and 'Monitor' are connected to an 'Interface' block in the middle layer. The 'Interface' block is connected to the 'DUT' (Design Under Test) in the bottom layer. Arrows indicate the flow of data and control between these components.</p>	

30. What are the different layers of layered architecture?

In verification, a layered architecture is a methodology in which the verification environment is structured into multiple layers of abstraction, each building on the lower layers. This approach separates the functionality and responsibilities of different layers, making verification more manageable and scalable.

Layered architecture typically consists of three main layers:

1) Testbench layer

This is the top layer of the verification architecture, which contains test scenarios that stimulate the design under test (DUT) and verify its functionality. The testbench layer is responsible for test management, collecting and analyzing results, and reporting errors and warnings.

2) Verification IP (VIP) layer

The VIP layer provides hardware and software components that are reusable and pre-verified, such as memory models, bus functional models (BFMs), and protocol checkers. The VIP layer abstracts the DUT interface and behavior, allowing the testbench layer to focus on DUT functionality.

3) Functional block layer

This is the lowest layer of the verification architecture, which contains individual blocks of the design. The functional block layer specifies the behavior and function of the DUT at the RTL level. The functional block layer includes RTL code, gate-level models, and timing models.

The layered architecture approach promotes reusability and scalability of the verification environment. Each layer can be independently designed and tested, facilitating incremental verification of the DUT. It allows efficient and thorough testing for complex designs with multiple blocks, interfaces, and protocols. By separating concerns and providing abstraction, layered architecture makes verification more manageable and improves the quality of the design.