1. Explain with an example how uvm\_config\_db can be used to pass a virtual interface from top to agent.

In UVM (Universal Verification Methodology), the uvm\_config\_db provides a mechanism to share configuration data between different components of the testbench, such as between a top-level module and a UVM agent. This is done in a way that is decoupled, allowing testbench components to be reusable and modular.

1. Write a code for virtual interface in uvm which has:
   1. Port declarations
   2. Signal & Variable declarations
   3. Modports
   4. Other interfaces
   5. Any other system Verilog code as need be

// File: my\_if.sv

interface my\_if #(parameter WIDTH = 8);

// Signal declarations

logic clk; // Clock signal

logic rst; // Reset signal

logic [WIDTH-1:0] data; // Data bus

logic valid; // Valid signal indicating data validity

// Modports to control signal directions

modport master(input clk, rst, valid, data); // Master interface modport

modport slave(output clk, rst, valid, data); // Slave interface modport

// Nested interface example

interface nested\_if;

logic [3:0] control; // Control signal (4 bits)

logic enable; // Enable signal

endinterface

// Nested interface instance

nested\_if nested; // Instance of nested\_if

endinterface

// File: my\_agent.sv

class my\_agent extends uvm\_agent;

// Declare a virtual interface of type my\_if

virtual my\_if m\_if;

// Constructor to initialize the agent

function new(string name = "my\_agent");

super.new(name); // Call parent constructor

endfunction

// Task to interact with the virtual interface

task run();

// Wait for the positive edge of the clock

@(posedge m\_if.clk);

// Display the data received when valid is high

if (m\_if.valid) begin

$display("Data received: %h", m\_if.data);

end

// Show nested interface values

$display("Control signal: %b, Enable signal: %b", m\_if.nested.control, m\_if.nested.enable);

endtask

endclass

// File: top.sv

module top;

// Declare the physical interface instance (WIDTH = 8)

my\_if #(8) my\_interface();

// Declare the UVM agent

my\_agent agnt;

// Initial block to set the virtual interface and start the test

initial begin

// Instantiate the UVM agent

agnt = new("agnt");

// Set the virtual interface in the uvm\_config\_db for the agent to use

uvm\_config\_db#(virtual my\_if)::set(null, "agnt.m\_if", my\_interface);

// Start the UVM test (this will call agnt.run)

run\_test();

end

// Generate clock, reset, and data signals

initial begin

// Initialize signals

my\_interface.clk = 0;

my\_interface.rst = 1;

my\_interface.valid = 0;

my\_interface.data = 8'h00;

my\_interface.nested.control = 4'b1010; // Nested control signal

my\_interface.nested.enable = 1'b1; // Nested enable signal

// Reset and clock generation

#5 my\_interface.rst = 0;

#10 my\_interface.clk = 1; // Rising edge of clock

#10 my\_interface.clk = 0; // Falling edge of clock

#10 my\_interface.valid = 1; my\_interface.data = 8'hAA; // Send data

#10 my\_interface.clk = 1; // Rising edge of clock

#10 my\_interface.clk = 0; // Falling edge of clock

#10 my\_interface.valid = 0; // Clear valid signal

end

endmodule

// File: test.sv

`include "uvm\_macros.svh"

// UVM test class

class test extends uvm\_test;

// Constructor

function new(string name = "test");

super.new(name); // Call the parent constructor

endfunction

// Run the test function

function void run();

// Print a simple message

$display("Running the test...");

endfunction

endclass

// Testbench top-level module to run the UVM test

module testbench;

test t;

initial begin

// Create the test instance

t = test::new();

// Run the UVM test

run\_test();

end

endmodule

1. Illustrate with an example how to use parameterized interfaces with the uvm\_config\_db

Parameterized interfaces allow you to pass different types of data widths or other parameters. We’ve already seen a simple example with my\_if #(8). To make it clearer, let's use another example where the interface's width is parameterized.

interface my\_if #(parameter WIDTH = 8);

logic clk;

logic rst;

logic [WIDTH-1:0] data;

modport m1(input clk, rst, data);

modport m2(output clk, rst, data);

endinterface

module top;

// Instantiate the interface with parameter WIDTH = 16

my\_if #(16) my\_interface();

my\_agent agnt;

initial begin

// Set the virtual interface for the agent

uvm\_config\_db#(virtual my\_if)::set(null, "agnt.m\_if", my\_interface);

// Start the UVM test

run\_test();

end

endmodule

This example shows how you can parameterize the width of the interface by specifying WIDTH = 16. This allows flexibility in terms of data width between different components, which is a common use case in verification environments.

1. How can you connect abstract class with a concrete class in UVM?

In UVM, abstract classes are often used to define interfaces that will be extended by concrete classes. This is especially useful for defining generic functionality that can be specialized for different types of agents or other UVM components.

* Define an Abstract Class

class base\_agent extends uvm\_agent;

// Abstract method

virtual function void run();

`uvm\_error("RUN", "Abstract run() function not implemented.")

endfunction

endclass

* Define a Concrete Class that Extends the Abstract Class

class concrete\_agent extends base\_agent;

// Concrete implementation of the abstract method

function void run();

$display("Concrete agent running.");

endfunction

endclass

* Instantiate the Concrete Class in the Testbench

module top;

base\_agent agnt;

initial begin

// Instantiate the concrete class

agnt = new("agnt");

// Call the run method

agnt.run();

end

endmodule

Here, in the top module, we instantiate the concrete class concrete\_agent via the abstract class pointer agnt. This is an example of connecting an abstract class (base\_agent) with a concrete class (concrete\_agent) in UVM.