Continue

Uvm_top and uvm_test_top

All verification components, interfaces and DUT are instantiated in a top level module called testbench. It is a static container to hold everything required to be simulated and becomes the root node in the hierarchy. This is usually named tb or tb_top although it can assume any other name. Simulators typically need to know the top level module so that it can analyze components within the top module and elaborate the design hierarchy. Click here to know more about top level modules !Testbench Top ExampleThe example below details the elements inside the top module tb_top. module tb_top; import uvm_pkg::*; // Complex testbenches will have multiple clocks and hence multiple clock // generator modules that will be instantiated elsewhere // For simple designs, it can be put into testbench top bit clk; always #10 clk Note the following :tb_top is a module and is a static container to hold everything elseIt is required to import uvm_pkg in order to use UVM constructs in this moduleClock is generated in the testbench and passed to the interface handle dut_if1The interface is set as an object in uvm_config_db via set and will be retrieved in the test class using get methodsThe test is invoked by run_test method which accepts name of the test class base_test as an argumentCall waveform dump tasks if requiredClock generationA real design may have digital blocks that operate on multiple clock frequencies and hence the testbench would need to generate multiple clocks and provide as an input to the design. Hence clock generation may not be as simple as an always block shown in the example above. In order to test different functionalities of the design, many clock parameters such as frequency, duty cycle and phase may need to be dynamically updated and the testbench would need infrastructure to support such dynamic operations. // Module level clock generation module clk_main (...); // Code for clock generation endmodule // Instantiated and connected to an interface module tb_top; bit clk_main_out; clk_main u_clk_main_0 (.out (clk_main_out), ...); dut_if dut_if0 (clk_main_out, ...); endmodule The approach shown above may not be scalable and need to be driven from the testbench using hierarchical signal paths since they are instantiated as modules. A better UVM alternative is to create an agent for the clock so that it can be easily controlled from sequence and tests using agent configuration objects. class clk_agent extends uvm_agent; clk_cfg m_clk_cfg; virtual function void build_phase(uvm_phase phase); super.build_phase(phase); // Get clk_cfg object and build this agent accordingly endfunction endclass class base_test extends uvm_test; clk_cfg m_clk_cfg; virtual function void build_phase(uvm_phase phase); super.build_phase(phase); m_clk_cfg.m_freq = 500; endfunction endclass Reset GenerationIn a similar way, a reset agent can be developed to handle all reset requests. In many systems, there are two kinds of reset - hardware and software. A software reset is typically done through a register model and would be separate from a reset agent. Hardware resets include assertion of the system reset pin for a given duration or follow a certain sequence of events before the actual reset is applied. All such scenarios can be handled separately using this reset agent which needs a handle to the reset interface. class reset_agent extends uvm_agent; reset_cfg m_reset_cfg; // Rest of the code endclass class my_sequence extends uvm_sequence; virtual task body(); hw_reset_seq m_hw_reset_seq; // Call the required kind of reset sequence in test scenario m_hw_reset_seq.start(p_sequencer); endtask endclass Creation of internal tap pointsSome testbench components may rely on tapping internal nets in the design to either force or sample values to test certain features. These internal nets may need to be assigned to a different value based on input stimuli and can be done so in the top level testbench module. Such signals can be tied to a generic interface and be driven from another agent. interface gen_if; logic [99:0] signals; // General 100-bit wide vector endinterface module tb_top; gen_if u_if0 (); des u_des (...); // Assign an internal net to a generic interface signal assign u_if0.signals[23] = u_des.u_xyz.u_abc.status; endmodule A testcase is a pattern to check and verify specific features and functionalities of a design. A verification plan lists all the features and other functional items that needs to be verified, and the tests neeeded to cover each of them.A lot of different tests, hundreds or even more, are typically required to verify complex designs.Instead of writing the same code for different testcases, we put the entire testbench into a container called an Environment, and use the same environment with a different configuration for each test. Each testcase can override, tweak knobs, enable/disable agents, change variable values in the configuration table and run different sequences on many sequencers in the verification environment. Class Hierarchy Steps to write a UVM Test 1. Create a custom class inherited from uvm_test, register it with factory and call function new // Step 1: Declare a new class that derives from "uvm_test" // my_test is user-given name for this class that has been derived from "uvm_test" class my_test extends uvm_test; // [Recommended] Makes this test more re-usable `uvm_component_utils (my_test) // This is standard code for all components function new (string name = "my_test", uvm_component parent = null); super.new (name, parent); endfunction // Code for rest of the steps come here endclass 2. Declare other environments and verification components and build them // Step 2: Declare other testbench components - my_env and my_cfg are assumed to be defined my_env m_top_env; // Testbench environment that contains other agents, register models, etc my_cfg m_cfg0; // Configuration object to tweak the environment for this test // Instantiate and build components declared above virtual function void build_phase (uvm_phase phase); super.build_phase (phase); // [Recommended] Instantiate components using "type_id::create()" method instead of new() m_top_env = my_env::type_id::create ("m_top_env", this); m_cfg0 = my_cfg::type_id::create ("m_cfg0", this); // [Optional] Configure testbench components if required, get virtual interface handles, etc set_cfg_params (); // [Recommended] Make the cfg object available to all components in environment/agent/etc uvm_config_db #(my_cfg) :: set (this, "m_top_env.my_agent", "m_cfg0", m_cfg0); endfunction 3. Print UVM topology if required // [Recommended] By this phase, the environment is all set up so its good to just print the topology for debug virtual function void end_of_elaboration_phase (uvm_phase phase); uvm_top.print_topology (); endfunction 4. Start a virtual sequence // Start a virtual sequence or a normal sequence for this particular test virtual task run_phase (uvm_phase phase); // Create and instantiate the sequence my_seq m_seq = my_seq::type_id::create ("m_seq"); // Raise objection - else this test will not consume simulation time* phase.raise_objection (this); // Start the sequence on a given sequencer m_seq.start (m_env.seqr); // Drop objection - else this test will not finish phase.drop_objection (this); endtask How to run a UVM testA test is usually started within testbench top by a task called run_test.This global task should be supplied with the name of user-defined UVM test that needs to be started. If the argument to run_test is blank, it is necessary to specify the testname via command-line options to the simulator using +UVM_TESTNAME. // Specify the testname as an argument to the run_test () task initial begin run_test ("base_test"); end Definition for run_test is given below. // This is a global task that gets the UVM root instance and // starts the test using its name. This task is called in tb_top task run_test (string test_name=""); uvm_root top; uvm_coreservice_t cs; cs = uvm_coreservice_t::get(); top = cs.get_root(); top.run_test(test_name); endtask How to run any UVM testThis method is preferred because it allows more flexibility to choose different tests without modifying testbench top every time you want to run a different test. It also avoids the need for recompilation since contents of the file is not updated.If +UVM_TESTNAME is specified, the UVM factory creates a component of the given test type and starts its phase mechanism. If the specified test is not found or not created by the factory, then a fatal error occurs. If no test is specified via command-line and the argument to the run_test() task is blank, then all the components constructed before the call to run_test() will be cycled through their simulation phases. // Pass the DEFAULT test to be run if nothing is provided through command-line initial begin run_test ("base_test"); // Or you can leave the argument as blank // run_test (); end // Command-line arguments for an EDA simulator $> [simulator] -f list +UVM_TESTNAME=base_test UVM Base Test ExampleIn the following example, a custom test called base_test that inherits from uvm_test is declared and registered with the factory.Testbench environment component called m_top_env and its configuration object is created during the build_phase and setup according to the needs of the test. It is then placed into the configuration database using uvm_config_db so that other testbench components within this environment can access the object and configure sub components accordingly. // Step 1: Declare a new class that derives from "uvm_test" class base_test extends uvm_test; // Step 2: Register this class with UVM Factory `uvm_component_utils (base_test) // Step 3: Define the "new" function function new (string name, uvm_component parent = null); super.new (name, parent); endfunction // Step 4: Declare other testbench components my_env m_top_env; // Testbench environment my_cfg m_cfg0; // Configuration object // Step 5: Instantiate and build components declared above virtual function void build_phase (uvm_phase phase); super.build_phase (phase); // [Recommended] Instantiate components using "type_id::create()" method instead of new() m_top_env = my_env::type_id::create ("m_top_env", this); m_cfg0 = my_cfg::type_id::create ("m_cfg0", this); // [Optional] Configure testbench components if required set_cfg_params (); // [Optional] Make the cfg object available to all components in environment/agent/etc uvm_config_db #(my_cfg) :: set (this, "m_top_env.my_agent", "m_cfg0", m_cfg0); endfunction // [Optional] Define testbench configuration parameters, if its applicable virtual function void set_cfg_params (); // Get DUT interface from top module into the cfg object if (! uvm_config_db #(virtual dut_if) :: get (this, "", "dut_if", m_cfg0.vif)) begin `uvm_error (get_type_name (), "DUT Interface not found !") end // Assign other parameters to the configuration object that has to be used in testbench m_cfg0.m_verbosity = UVM_HIGH; m_cfg0.active = UVM_ACTIVE; endfunction // [Recommended] By this phase, the environment is all set up so its good to just print the topology for debug virtual function void end_of_elaboration_phase (uvm_phase phase); uvm_top.print_topology (); endfunction function void start_of_simulation_phase (uvm_phase phase); super.start_of_simulation_phase (phase); // [Optional] Assign a default sequence to be executed by the sequencer or look at the run_phase ... uvm_config_db#(uvm_object_wrapper)::set(this,"m_top_env.my_agent.m_seqr0.main_phase", "default_sequence", base_sequence::type_id::get()); endfunction // or [Recommended] start a sequence for this particular test virtual task run_phase (uvm_phase phase); my_seq m_seq = my_seq::type_id::create ("m_seq"); super.run_phase(phase); phase.raise_objection (this); m_seq.start (m_env.seqr); phase.drop_objection (this); endtask endclass The UVM topology task print_topology displays all instantiated components in the environment and helps in debug and to identify if any component got left out.A test sequence object is built and started on the environment virtual sequencer using its start method.Derivative TestsA base test helps in the setup of all basic environment parameters and configurations that can be overridden by derivative tests. Since there is no definition for build_phase and other phases that are defined differently in dv_wr_rd_register , its object will inherently call its parent's build_phase and other phases because of inheritance. Function new is required in all cases and simulation will give a compilation error if its not found. // Build a derivative test that launches a different sequence // base_test In this case, only run_phase will be overriden with new definition in derived test and its super call will invoke the run_phase of the base_test .Assume that we now want to run the same sequence as in dv_wr_rd_register_test but instead want this test to be run on a different configuration of the environment. In this case, we can have another derivative test of the previous class and define its build_phase in a different way. // Build a derivative test that builds a different configuration // base_test (1) UVM使用双顶层的用法｜骏的世界 (lujun.org.cn) 1.UVM的根-uvm_root or uvm_test_top? (1) uvm_test_top实例化时的名字是uvm_test_top,这个名字是由UVM在run_test时自动指定的; (2) uvm_top的名字是_top_,但是在显示路径时,并不会显示这个名字,而只显示从uvm_test_top开始的路径; 2. uvm_root(类) (1) uvm_root本质是uvm_component; (2) uvm_root的存在是为了保证验证平台只有一棵树(uvm_root是单实例类,整个UVM验证平台中,有且只有uvm_root的一个实例存在),并且发挥着phase controller的作用,管理所有component的phase. (3) 任何没有指定parent的component,默认其parent是uvm_top(全局变量, uvm_root的一个实例). (4) uvm_component的两大机制(树形组织结构和phase机制)都离不开uvm_root; 3. uvm_top(uvm_root类的句柄)及常用函数 (1) UVM中真正的树根,任何组件实例都在它之下(如果组件的parent为null,那么该组件为uvm_top的子组件); (2) uvm_top是一个全局变量,是uvm_root的一个实例,也是uvm_root唯一的实例. (3) uvm_top的名字是_top_,但是在显示路径时,并不会显示这个名字,而只显示从uvm_test_top开始的路径. (4) uvm_top.find及示例; uvm_component comp; comp=uvm_top.find("*.axi_agt"); 注1:其中*表示匹配任意字符;?表示匹配一个字符;+表示匹配一个或多个字符; (5) uvm_top.find_all及示例;uvm_component comps[$]; uvm_top.find_all("*.axi_?",comps); foreach(comps[i]) begin comps[i].print(); end 4. uvm_test_top (1) 通过run_test语句创建的实例的名字为uvm_test_top.无论传递给run_test的参数是什么,创建的实例的名字都为uvm_test_top. (2) uvm_test_top的parent是uvm_top,而uvm_top的parent是null. (3) UVM中,支持uvm_top下有多个叶子节点,但是,多个叶子节点不能都叫uvm_test_top,如下所示. 5.uvm双顶层 详见UVM使用双顶层的用法｜骏的世界 (lujun.org.cn); 问题:双顶层中的另外一个顶层是怎么跑起来的？ Welcome to the world of Universal Verification Methodology (UVM), where efficient and effective verification processes come to life. If you're venturing into UVM, you may have stumbled across terms like UVM_Top and UVM_Test_Top. These components are pivotal in structuring your verification environment. But what exactly do they entail? Understanding these elements can significantly enhance your simulation experience. Whether you're a novice or an experienced engineer, grasping the ins and outs of UVM_Top and UVM_Test_Top will empower you with better control over your testing scenarios. Let's dive deep into this essential topic, unravel their differences, explore their benefits, and arm yourself with practical insights for successful implementation. The journey through the intricate maze of UVM starts here! Understanding UVM_Top and UVM_Test_Top UVM_Top and UVM_Test_Top play pivotal roles in the Universal Verification Methodology. They are essential for structuring your testbench effectively. UVM_Top acts as the top-level module in a verification environment. It orchestrates various components like agents, monitors, and scoreboards. This makes it a crucial element for managing complex designs. On the other hand, UVM_Test_Top is specifically designed to facilitate individual tests within that environment. It focuses on executing specific scenarios or test cases efficiently. While both serve important purposes, their functionalities differ significantly. Understanding these distinctions helps streamline your verification process, making it more efficient and organized. Using them correctly can enhance code readability and maintainability while reducing confusion during simulation runs. Grasping their unique roles empowers engineers to build robust verification environments tailored to their needs. Key Differences between UVM_Top and UVM_Test_Top UVM_Top serves as the primary container for your entire verification environment. It brings together all components and configurations required to execute a simulation. On the other hand, UVM_Test_Top acts as a specialized layer within UVM_Top focused on executing specific test scenarios. This distinction allows you to run targeted tests without altering the overall structure of the verification environment. While both serve critical roles, their purposes differ significantly. UVM_Top encompasses everything from agents to scoreboards, ensuring that they function cohesively. Conversely, UVM_Test_Top zeroes in on particular methodologies or use cases during testing phases. This separation helps streamline your workflow and enhances modularity in design. Understanding these differences can improve your approach to creating efficient testbenches and enable better resource management throughout various stages of development. Benefits of Using UVM_Top and UVM_Test_Top Utilizing UVM_Top and UVM_Test_Top brings several advantages to your verification process. First, they enhance modularity. This allows for more maintainable test environments, as components can be easily swapped or upgraded without disrupting the entire structure. Next, these constructs streamline the simulation process. By providing a clear hierarchy, they help organize tests effectively. This organization minimizes confusion when navigating complex systems. Another significant benefit is improved reusability. Once you establish a robust framework with UVM_Top and UVM_Test_Top, you can leverage it across multiple projects. These tools also promote better collaboration among team members. With defined roles and responsibilities in place, tasks become clearer, fostering an efficient working environment. Using these top-level constructs facilitates easier debugging by isolating issues within specific layers of your architecture. How to Implement UVM_Top and UVM_Test_Top in Your Verification Environment Implementing UVM_Top and UVM_Test_Top in your verification environment is a structured process that can enhance your testbench architecture. Start by defining the UVM_Top, which serves as the top-level module for your entire verification environment. This component orchestrates all sub-modules like agents, monitors, and drivers. Next, create the UVM_Test_Top that will manage individual tests within this framework. Each test should inherit from the base class provided by UVM to leverage its features effectively. Connect these components carefully to ensure smooth communication between them. Use configuration objects to pass parameters easily across different layers of your hierarchy. Don't forget to instantiate necessary sequences and scoreboards inside the test top for validation purposes. Properly setting up these components will lay a solid foundation for robust verification processes in any project you undertake. Troubleshooting Common Issues with UVM_Top and UVM_Test_Top When working with UVM_Top and UVM_Test_Top, it's common to encounter some hiccups. One frequent issue can be related to the configuration of your testbench. Ensure that all components are correctly initialized and connected. Another problem might arise from signal mismatches between modules. Make sure that the bus interfaces align properly, as any discrepancies can lead to simulation failures or incorrect results. Keep an eye on your environment's memory consumption too. If you're facing slow simulations, consider optimizing your code or breaking down large data sets into smaller chunks for easier management. Logging is crucial in troubleshooting. Utilize built-in reporting tools within UVM to identify error sources quickly. A well-structured log will guide you to potential issues faster than manual searches through the codebase. Don't underestimate the power of community forums and resources. Engaging with fellow users often uncovers solutions for problems you've yet to discover on your own. Best Practices for Using UVM_Top and UVM_Test_Top When using UVM_Top and UVM_Test_Top, organization is key. Maintain a clear hierarchy in your testbench to avoid confusion during complex simulations. A well-structured layout enhances readability and simplifies debugging. Utilize consistent naming conventions for your components. This practice helps team members quickly identify different parts of the verification environment, making collaboration more efficient. Leverage reusable components whenever possible. By designing modular tests, you can save time and reduce redundancy across projects. Integrate thorough documentation throughout your setup. Clear comments and guidelines will assist others in understanding your design choices while providing valuable insights for future enhancements. Regularly run regression tests to catch potential issues early on. Continuous validation ensures that changes do not introduce new bugs or disrupt existing functionalities within your UVM framework. Conclusion When it comes to verification in the UVM environment, understanding UVM_Top and UVM_Test_Top is crucial for achieving efficient and effective results. By grasping their distinct roles within your testbench architecture, you can enhance both performance and reliability. Utilizing these components properly allows for better organization of your testing framework while streamlining the development process. The benefits are clear: improved modularity, easier debugging, and a more structured approach to verification tasks. Implementing best practices ensures that you're not only using these elements effectively but also paving the way for future scalability in your projects. As challenges arise during implementation or execution, knowing how to troubleshoot common issues helps maintain productivity without unnecessary setbacks. By keeping these insights in mind, you'll be well-equipped to leverage UVM_Top and UVM_Test_Top successfully within your verification environments. Embracing these strategies will transform your workflow into one that's both robust and adaptable as technology continues to evolve. Hello All, I want to access driver or environment variables using $root or uvm_top or uvm_test_top? Thanks, Dipak Jatiya In reply to Dipak_jatiya: This is a bad idea, because you have to use hierarchical paths. This limits the reusability of your code. In reply to Dipak_jatiya: $root refers to your top module. Components like driver/env are created in test, so you can access these from anywhere (including from top module) using e.g., $root.uvm_test_top.env_h.agent_h.drv_h.var_name; $root.uvm_top.env_h.agent_h.drv_h.var_name; Or you can use uvm_top/uvm_test_top (refers to test instance) from tests or testbench environment below test hierarchy, uvm_test_top.env_h.agent_h.drv_h.var_name; uvm_top.env_h.agent_h.drv_h.var_name; In reply to MayurKubavat: Hi Mayur, I've tried the above things but can't get the test name instance from above. Get below Error while using $root.uvm_test_top.y and uvm_test_top.y Failed to find 'uvm_test_top' in hierarchical name '/$root/uvm_test_top/y uvm_top.uvm_test_top.y Field/method name (uvm_test_top) not in 'uvm_top' In short i can't get the test variable using uvm_test_top. I'm using this from driver run phase. Thanks, Dipak Jatiya Why not to use configDB in such case if you know which heir and which variable your driver need to know! not getting why you have to access TB variable using heir. If you could tell us more about your problem it would be more helpful. In reply to Dipak_jatiya: If you want to avoid using hierarchical paths. You can use uvm_utils#()::find(). For example, in the test, you know the component instance, you can do if you want to access a driver, $root.uvm_test_top.m_env.m_agent.m_driver You can do agent_driver m_agent_driver; m_agent_driver = uvm_utils#(agent_driver)::find(m_env) In reply to MayurKubavat: Hi, What is the difference between uvm_top and uvm_test_top? Can You Say in hierarchical order