2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE OCTOBER 26-27, 2021

Using Dependency Injection Design Pattern in Power Aware Tests

Mehmet Tukel, QT Technologies Ireland Limited, Cork, Ireland (*mtukel@qualcomm.com*)

Luca Sasselli, QT Technologies Ireland Limited, Cork, Ireland (*lsassell@qualcomm.com*)

David Guthrie, QT Technologies Ireland Limited, Cork, Ireland (*dguthrie@qti.qualcomm.com*)

***Abstract*— One of the pillars of staying strong in the semiconductor market is managing the time-to-market of new products. This management of course includes many faceted, complicated tasks, one of which is successfully delivering the Design Verification as quickly and as complete as possible. Although reusing IPs in the product design solves some of the problems of time-to-market management, Design Verification should also be considered carefully as its NRE cost is usually higher. A good structured testbench finds the bugs in the design and increases reuse which consequently reduces the time-to-market. Another common feature of today's large SoCs is that they must have low-power consumption. Advanced low-power design techniques are used for obtaining such products. Verification of these low-power SoCs and IPs is a challenge, as making a UVM compatible testbench power aware introduces another level of complexity both in the testbench and the test structure. In this paper, we present how to use an Object-Oriented Design Pattern called Dependency Injection in a power aware testbench - a concept borrowed from the software world. We aim to show how it helps testbench reuse and constructing UVM compatible tests and as a result decreases the Verification NRE.**

Keywords*— *dependency injection; testbench reuse; power aware; time-to-market

I. Introduction

A well-structured testbench requires effort and attention to detail, especially during the development of the verification environment. This effort subsequently pays off when scaling and improving the reuse of the testbench between different projects and even in test development. Overall, developing a well-structured testbench becomes cheaper and faster than those without a structure that considers what kind of design it verifies. Universal Verification Methodology (UVM) provides a base methodology for how the testbench should be constructed. UVM separates the testbench into two domains: the static domain and the dynamic domain. The static domain consists of the Design under Test (DUT) and the wrapping testbench top module that connects the Verification IP (VIP) interfaces to the DUT. The dynamic domain is the class hierarchy that implements the test layer. UVM dictates a certain flow and design hierarchy for the dynamic domain, i.e., the test instantiates the environment and the environment instantiates the agents. There is not a lot of opportunity for enhancement in the static domain outside of using macros and generate blocks to improve reusability. On the other hand, for the dynamic part we can leverage Object Oriented Design patterns which were initially introduced into the software world. This is mostly due to the flexibility of the SystemVerilog (SV) *class* over SV *module*. UVM itself also leverages factory and singleton design patterns. UVM also encourages inheritance and polymorphism to a great degree. The built-in phases or methods of a UVM test can be overridden in extended tests to configure the testbench differently. If this is done as per the UVM, it provides significant reusability.

However, the UVM understandably does not provide any resources on how to implement power aware (PA) tests. This is too design oriented for a generic library like UVM. In this paper, our goal is to fill the gap by proposing a method for PA UVM tests that enhance the scalability and the reusability of the tests. In doing this, we aim to encapsulate the PA methods into one class and provide a base test class for concrete PA/non-PA tests. We also propose an inheritance hierarchy for the tests. In Section 2, we discuss problem statement and the possible solutions. In Section 3, we present our preferred solution and how to use it. In Section 4, we present a sample design to demonstrate the proposed technique in a use case. In Section 4, we conclude our remarks.

II. Problem Statement and Possible Solutions

A. *Problem Statement*

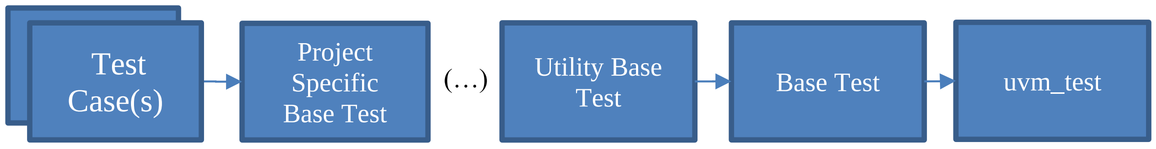

Figure 1. An example test hierarchy in a UVM compatible testbench

A concrete test (Test Case) extended from a base test, i.e., the tests that are simulated, should work seamlessly whether they are PA or non-PA.

The PA tests must be able to access PA related functionalities/dependencies such as power down and power up tasks. A PA test case must also be able to access all the dependencies that a non-PA test accesses. Additionally, some overlapping between PA and non-PA functions might exist, although with different behavior in each case, requiring the test to handle the same task in a different way.

B. *Possible Solutions*

In a PA verification environment, the test structure is not the only component that needs to be different. The DUT might be built differently for PA and non-PA tests. Furthermore, the Universal Power Format (UPF) is elaborated together with the DUT in a PA simulation. Therefore, non-PA and PA tests have essentially different builds. There is no easy way around it. Requiring different builds is out of scope of this paper which focuses on the test hierarchy and structure of a PA verification environment.

1) A Completely Different Testbench for PA tests

A completely different testbench can be developed for the PA tests. In this solution, the DUT is instantiated in a different top module with a limited number of VIP interfaces. In addition, the test hierarchy is structured with the sole consideration of the PA use cases. If the main focus of PA verification is integration testing and the tests do not utilize most of the VIPs, this would be a preferable solution. For instance, a software running on a CPU in the DUT might be initiating the power up and power down sequences. Therefore, the testbench might not need stimuli generation by the UVM compatible VIPs. This solution is obviously only suitable for a certain type of PA verification environment. The downside of this approach is that two different verification projects are required for the same DUT. However, it may improve simulation run times.

2) Make every test power aware

Based on the problem statement, a PA test needs to have all the features that a non-PA test has. Therefore, we can implement the "*Project Specific Base Test*" as a base test that has all the PA functionalities. For instance, this test runs the initial power up sequences in its run phase and calls UPF APIs as well as all other initialization tasks. This approach requires all tests to be run on a UPF annotated build despite it not being required for most of the tests. Given that a UPF annotated simulation runs slower than a regular simulation, this approach would be a misuse of compute resources. Using this approach, we could have simplified the test hierarchy, however the simulation times would be significantly increased.

3) Extending the test cases from different parent tests

In this approach, we have two separate base tests as the "*Project Specific Base Test*". The former implements all the PA functionalities, i.e., the PA base test. The latter, which is basically the parent of the PA base test, implements the functionalities that are required for a non-PA test case. This is the second-best approach among the solutions presented in this paper and does not have negative impact on the simulation performance. However, the downside is that the test cases tend to be tightly coupled with their parents and it is difficult to promote a non-PA test to a PA test. Besides this, the PA test base must override most of the initialization and bring-up tasks of its parent which makes the maintenance of these tests difficult, as the changes in the parent test must be adequately overridden in the child. Another important issue appears when we want to have a base test for certain group of test cases, e.g., a base test targeting a certain IP in the DUT. In this case, we would need to implement two base tests: one derived from the regular *Project Specific Base Test* and the other should be derived from the PA version of the *Project Specific Base Test*. Therefore, we may observe issues in scaling and reusing the base tests if we implement this solution.

4) Dependency Injection Design Pattern for the PA Dependency

The Dependency Injection Design Pattern [1] provides a loosely coupled relationship between the dependent and the client class. The client class depends on the injected class. In our proposed PA UVM compatible testbench solution, the client class is the *Project Specific Base Test*. The injected dependency is a power component that is aggregated into the *Project Specific Base Test*. The injector in our case is simply a simulation argument that distinguishes whether the test is a PA or non-PA test. This approach has multiple benefits: The *Project Specific Base Test* is the same for PA and non-PA tests. Therefore, we are not required to develop two base tests (PA and non-PA) if we decide to implement a base test for certain test cases. This solution does not have a negative impact on performance as the non-PA tests do not depend on the concrete power component. Any non-PA test can be promoted to a PA test seamlessly. Therefore, this approach improves reusability and scalability significantly.

III. PREFERED SOLUTION: DEPENDENCY INJECTION

The test structure shown in Fig. 2 presents the implementation of the proposed solution. The power component depicted in Fig. 2 is a child class of a power component base class. The *Project Specific Base Test* has a handle of the power component base type. The power component base class defines the methods that are planned to be used in PA use cases.

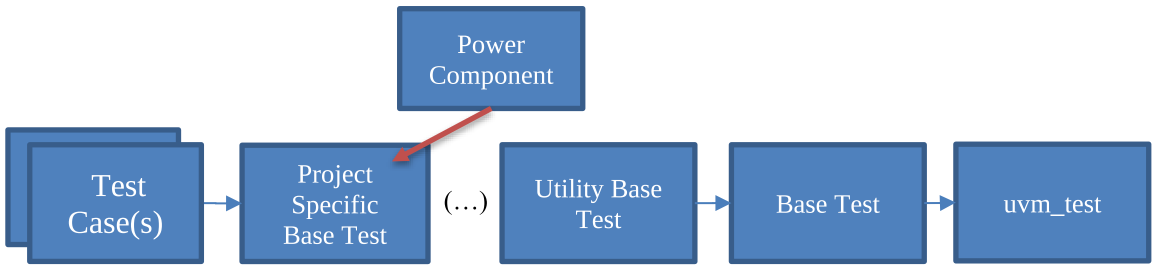

Figure 2. Power Component Dependency Injection into the Test Hierarchy.

```
class power_comp_base extends uvm_component;
(..)
virtual task power_up();
endtask

virtual task power_down();
endtask
(…)
endclass : power_comp_base

class power_comp extends power_comp_base;
(...)
task power_up();
//Do something
endtask

task power_down();
//Do something
endtask
(…)
endclass : power_comp
```

Figure 3. Concrete and Base Power Component Classes.

The methods of the power component base class can be implemented with minimal features or can be just blank depending on the requirements. However, the concrete component (*power component*) will have actual implementations of the PA methods. Due to polymorphism, when a method of the base handle is called, its behavior depends on the type of real instance assigned to that handle. The real instance is instantiated in the build phase of the *Project Specific Base Test*, depending on the type of the test, i.e., whether it is PA or non-PA or by a command line argument. A PA test instantiates the concrete component whereas the non-PA test instantiates the power component base. Thus, a PA method is always called in the *Project Specific Base Test* and the call either does nothing or execute the real PA functionality. In Fig. 3 and 4, we present a sample of UVM code that shows how to implement the proposed dependency injection design pattern in a PA testbench.

```
class prj_test_base extends util_test_base;
(…)
virtual function build_phase(uvm_phase phase);
   super.build_phase(phase);
   if (is_power_aware)
      m_pwr_comp = power_comp::type_id::create("m_pwr_comp", this);
   else
      m_pwr_comp = power_comp_base::type_id::create("m_pwr_comp", this);
endfunction

virtual task run_phase(uvm_phase phase);
   super.run_phase(phase);
   m_pwr_comp.power_up();
endtask

(…)
endclass : prj_test_base
```

Figure 4. Injecting the correct dependency in the test without impacting the test cases.

IV. CASE STUDY

The usage and benefits of the method presented in this paper will be explained over the use case depicted in Fig. 5. The case study design has two power domains such as PD1 and PD2. The green block is PD1, and the red is PD2. Both domains are subject to power gating: PD1 is controlled externally by a third-party and PD2 is controlled by an internal controller or MCU through a Power Switch (PSW) powered by the PD1 domain. In this paper, for completeness we assume both power domains have logic that are subject to retention.

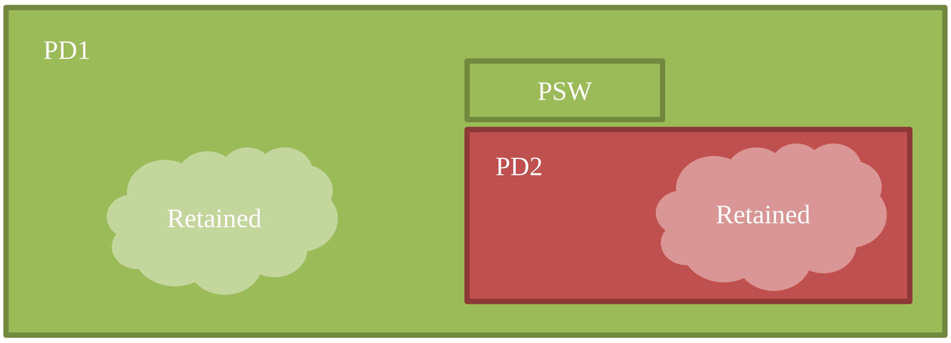

Figure 5. Power domain representation of the case study

The behavior of different logic elements must be validated in different power states and sequences. Therefore, the presence of power gating capability increases the complexity of the testbench, and the number of tests required to fully verify the DUT functionality. However, since those functions should have been individually tested in non-PA tests, a well-structured PA environment shouldn't affect the reusability of the existing tests. Suppose two functions, F1 and F2, residing in two different power domains, PD1 and PD2. The existing tests and functions developed to validate F1 and F2 in a non-PA setting should be reusable when the interactions with their respective power domains is verified. Table 1. presents a sample reference PA testplan for the reader.

Table I. A sample PA Testplan for the use case

# Test	**Description**	**Expectation**
1	Partial Power Gate: Validate F2 → Partial power down → Power restore using PSW → Validate F2 (after restore).	Expectation is that the DUT runs Function F2. The DUT is able to power gate and restore PD2 supply gracefully, i.e., no X propagation due to missing isolation, etc. Function F2 is still accessible after restoring the power.
2	Do partial power down → Power restore and check retention in PD2.	After power down and following power restore in PD2, the expectation is that the retained logic preserves the values it had before power down. The DUT must be able to resume from where it left off before power down.
3	Complete Power Gate: Validate F1 → Partial Power down → Power down PD1 → Power restore of PD1 → Partial power restore via PSW → Validate F1 (after complete restore).	Expectation is that the DUT runs Function F1 as expected. The DUT is able to power gate and restore completely (PD2 & PD1) gracefully, i.e., no X propagation due to missing clamps etc. Function F1 is still accessible after restoring the power.
4	Do complete power down (PD2 & PD1) → Power restore and check retention in PD1.	After power down and following power restore, the expectation is that the retained logic in PD1 preserves the values it had before power down. The DUT must be able to resume from where it left off before power down.

The functions required from a design are generally independent from whether the design is power collapsible or not. Usually, a low-power design is expected to go to low-power mode (power gated) after it performs the required function. The device remains in low-power mode for a certain period of time and then it is woken up by an internal or external agent when the function is required again. In short, a big chunk of the Functional PA verification is essentially validation of the core functions before and after power down to ensure nothing unexpected is happening due to power down and power restore sequences. Tests 1, 3 in Table 1 check the core functions of the sample design before and after powering down the power domains. Functions F1 and F2 can be individually verified in a non-PA test and can be reused in a PA test like we have in our use case scenario in Tests 1,3. Better yet, the non-PA test without any modification can be used as the PA test to verify the core functionality before-after power gating. See Fig. 6 for the reference test that checks the core functionality F1 (test 3).

Fig. 6 shows a reference test implementation for verification of a function that can seamlessly be used in a non-PA and PA environment. The implementation is merely presented to demonstrate the different possibilities available, although the non-PA test does not need power aware calls. In addition to the structural benefits mentioned in Section 2 and 3, the dependency injection technique allows us to implement the initial power up and the configuration of the DUT elegantly in the run phase of the base test (super.run_phase() call in Fig. 6.), and wraps up different sequences of the calls in the power component depending on whether the base test is PA or non-PA without needing nested if conditions or if/def macros. One can wonder what might happen if the testbench is developed gradually and the power up/down sequences are not ready when the core functions are to be tested. In

```
class testcase_f1_nonpa_pa extends prj_test_base;
(…)
virtual task run_phase(uvm_phase phase);
   super.run_phase(phase);//m_pwr_comp.init_pwr_up() call is in the base test

   repeat(N) begin //test multiple times for robustness
      check_function_F1();
      m_pwr_comp.power_down_PD2();
      m_pwr_comp.power_down_PD1();

      #1us; //wait some time in low power mode

      m_pwr_comp.power_restore_PD1();
      m_pwr_comp.power_restore_PD2();
    end
endtask
(…)
endclass : testcase_f1_nonpa_pa
```

Figure 6. Using a non-PA test as a PA test without any change.

other words, how can someone test the core functions without implementing power aware features? Note that, power aware features of the testbench are only needed in the actual PA tests. In a non-PA test, as explained in Section 3, the power component is taken from the base class, therefore power up/down tasks can be blank. The retention tests (tests 2,4) follow the same power down and restore sequence depicted in Fig 6, so we do not repeat them. Finally, if a low-level retention test is to be developed, e.g., one that checks what registers are retained and what registers are reset after power down and restore, implementing a PA test only is the valid option here, as there is no corresponding scenario in the non-PA environment.

V. Conclusion

The dependency injection technique is well suited to a PA testbench. It provides scalability such that there is no need to retain both PA and non-PA version of the base tests. Only the power component and the *Project Specific Base Test* need to be maintained. The encapsulation of the code is adequate in that no similar code is repeated throughout the components or tests. PA features are only implemented in the power component. By using the proposed approach, a non-PA test can be promoted to a PA test without needing to edit the code. If a subset of the test suit is to be run in a Gate Level Simulation (GLS), the same test suit can also be run in the Power Aware Gate Level Simulation (PA-GLS). This improves the maintainability and scalability of the environment dramatically, drastically reducing the NRE cost of the verification and thus contributes positively to the time-to-market.

Acknowledgment

We would like to thank Esha Hourihane Clancy for proofreading the manuscript.

References

[1] Wikipedia, "Dependency Injection," [Online]. Available: https://en.wikipedia.org/wiki/Dependency_injection.