Parameters and OVM — Can't They Just Get Along?

Bryan Ramirez
Xilinx, Inc.
3100 Logic Dr.
Longmont, CO 80503
720-652-3561
bryan.ramirez@xilinx.com

Michael Horn
Mentor Graphics, Corp.
1811 Pike Rd.
Longmont, CO 80501
303-974-0115
mike_horn@mentor.com

ABSTRACT
Verifying a highly parameterized design under test (DUT) that can be used in one of many different configurations requires additional effort over verifying an un-parameterized design or a parameterized DUT used in a single configuration. When using a single configuration, the parameters can be treated as static for the entire process, and the DUT can be verified without worrying about the parameters. However, the verification space grows exponentially if the DUT functionality must be verified over all possible configurations of each parameter. Strategies must be developed to ensure the verification process not only tests all parameterizations but also is as efficient as possible to maximize the number of different parameterizations that can be tested.

This paper will discuss the methods that have proven useful for verifying a highly parameterized DUT within an OVM testbench. This includes enhancing the default OVM functionality to create parameterized OVM tests and, consequently, testbenches that still allow the use of +OVM_TESTNAME. Techniques will be discussed for dealing with parameterized virtual interfaces, efficiently passing parameters down through the testbench hierarchy from the OVM test, and determining when to use parameters versus alternative methods. The paper will discuss optimizations for improving simulation throughput when using parameters, including actual numbers illustrating the efficiency improvements. Finally, the paper will describe options for ensuring that a parameterizable DUT is fully verified.

Categories and Subject Descriptors
B.5.2 [REGISTER-TRANSFER-LEVEL IMPLEMENTATION]: Reliability and Testing – built-in tests, error-checking, redundant design, test generation, testability.

General Terms
Measurement, Performance, Standardization, Languages, Design, Verification

Keywords
OVM, parameters, performance optimization, verification methodology, testbench

1. INTRODUCTION
A highly efficient testbench methodology and architecture that promote reuse are prerequisites to tackling multiple configurations of a parameterized DUT. To that end, the testbench should be built using the Open Verification Methodology (OVM) as testbenches written in OVM yield vast improvements in the construction and reuse of verification code. This is very important when verifying a parameterized DUT that will change functionality based upon the parameter settings.

Writing testbenches using the OVM yields huge improvements in the construction and reuse of verification code. In part this is because OVM provides a library of classes and a methodology for using those classes that promotes consistency in testbench development. However, OVM out-of-the-box is not set up by default to handle situations where both the design under test (DUT) and the testbench share parameters. Sharing parameters is an issue when the design needs to be tested in multiple, parameterized configurations. Several techniques will be described to facilitate parameter sharing and improve simulation throughput when parameters are involved. These techniques include allowing parameterized tests, selecting those tests at run time without having to recompile the code, and handling the required parameterized interfaces. Testbenches that utilize these techniques enable easy design reconfiguration and optimized simulation throughput. We will also describe a set of macros that save time and effort while safeguarding the quality of the design when passing parameters.

2. PARAMETERS AND THE OVM CONFIGURATION SPACE
Using these techniques will help to ease the use of parameters in OVM testbenches. The question might be asked, why are parameters used in the first place? Parameters can be used to control bus and address widths as seen in the examples below. Parameters can also control other aspects of a testbench. Parameters must be used in code locations where elaboration time constants are required. Another potential usage for parameters is in *generate* statements. Conversely, they should not be used for general testbench configuration, as the result is an inflexible environment. In other words, the only time parameters should be used in testbenches is when they must be used.

If a parameter does not have to be used to satisfy the language requirements of SystemVerilog, the OVM library provides a mechanism for storing configuration information that should be used instead. This configuration space can store strings, integer types, and objects. Utilizing the OVM configuration space allows individual tests to control what the testbench environment will look like at run time. As shown below, a test can be chosen after a design has been compiled and optimized once.

3. PARAMETERIZED TESTS
If a class (i.e., test) is parameterized, then the normal thing to do is to register the class with the factory using the *`ovm_component_param_utils()* macro. The downside of using this macro is that it registers the class only with the type-based factory. This is a problem, because in a standard OVM testbench, the test object is created by the *run_test()* task, which uses the string-based factory to create the test object. So there is a disconnect here.

In the standard OVM testbench, the test object is the top of the object hierarchy and instantiates the testbench environment, which instantiates any agents needed to communicate with interfaces on the DUT. Called from the top-level testbench module, the *run_test()* task does a string-based lookup in the OVM factory to determine which test to create and run. The string that *run_test()* uses to do the lookup is passed in either as an argument when *run_test("example_test")* is called or by using a *plusarg* called +OVM_TESTNAME on the command line when starting the simulation.

This setup works great when dealing with classes (tests) that do not have parameters because the *`ovm_component_utils()* macro is used to register the non-parameterized class with the factory. This registers the class with both the type-based factory and the string-based factory.

Since parameterized tests are registered using only the type-based factory, another tactic must be taken. The solution is to expand the contents of the *`ovm_component_param_utils()* macro and then add to this expanded code.

For example, if a test called *test1* (which was parameterized with BUS_WIDTH and ADDR_WIDTH) was part of the testbench, then the expanded macro would look like this:

```
class test1 #(int BUS_WIDTH = 16,
              int ADDR_WIDTH = 5)
extends test_base_c;

  typedef ovm_component_registry #(
   test1 #(BUS_WIDTH, ADDR_WIDTH)) type_id;

  static function type_id get_type();
    return type_id::get();
  endfunction : get_type

endclass
```

The *ovm_component_registry* type can take two parameters. One registers a type with the type-based factory. The other registers the class with the string-based factory. Since the goal is to register the class with the string-based factory, the code is changed to register with both, which looks like this:

```
class test1 #(int BUS_WIDTH = 16,
              int ADDR_WIDTH = 5)
extends test_base_c;

  typedef ovm_component_registry #(
   test1 #(BUS_WIDTH, ADDR_WIDTH),
   "test1") type_id;

  static function type_id get_type();
    return type_id::get();
  endfunction : get_type

endclass : test1
```

The class is now registered with the string-based factory, but there is another problem. If the code asks the string-based factory to create an instance of test1, the factory will return the default specialization of the test1 class (BUS_WIDTH = 16 and ADDR_WIDTH = 5). What if, in this configuration, the DUT has a BUS_WIDTH of 32 and an ADDR_WIDTH of 4? To allow the same parameters used by the DUT to be used by the test object created by *run_test()*, a specialization of the test that matches the DUT has to be created. Initially, the top-level testbench module looks like this:

```
module testbench ();
  parameter BUS_WIDTH = 32;
  parameter ADDR_WIDTH = 4;

  dut #(BUS_WIDTH, ADDR_WIDTH) i_dut(...);

  initial begin
    run_test("test1");
  end

endmodule
```

To establish the required specialization of the class, a *typedef* for the specialization of the *test1* class with the correct parameter values set is created. The *typedef* will not be used anywhere. It is there just to set the parameter values that are needed. The testbench module now looks like this:

```
module testbench ();
  parameter BUS_WIDTH = 32;
  parameter ADDR_WIDTH = 4;

  dut #(BUS_WIDTH, ADDR_WIDTH) i_dut(...);

  typedef test1 #(BUS_WIDTH, ADDR_WIDTH)
   test1_t;

  initial begin
    run_test("test1");
  end

endmodule
```

Now when *run_test()* is called, it will ask the string-based factory to create a *test1* object. The *test1* object that is created will be specialized with the parameter values that are needed per the *test1_t typedef* that was created. With this technique, parameters can be shared between the DUT and the testbench where required.

4. COVERAGE WITH PARAMETERS

Sharing parameters between the DUT and testbench is only one of many considerations when dealing with parameters and OVM. Verifying designs over a range of parameterizations, rather than one static configuration, can greatly increase the coverage space and, consequently, the time required to close on that coverage. Obviously this problem is compounded as the number of modifiable parameters increases. Although the problem is similar to adding additional crosses to functional coverage, it does have one difference. That is, functional coverage is closed during the simulation phase while parameterization coverage must be closed during compile or optimization, depending on how parameters are implemented. In either case, additional time outside of simulation must be spent to change parameters.

In an ideal world, all parameters would be crossed with each other to create parameterization coverage, and then that would be crossed with all functional coverage. This would ensure that each functional coverage item was tested against each specific parameterization. This quickly becomes infeasible as the coverage space grows exponentially as additional parameters or parameter values are added. Starting with a basic case as an example, if a design has only two parameters, each with two different values, crossing the parameters would result in four different parameterizations, which translates to four times the amount of functional coverage. What happens if the design has more parameters? Assume now the design has ten parameters, again with only two possible values each. This would translate to over 1000 different parameterizations. Closing on functional coverage is often difficult enough, but it would take an unreasonable amount of time if it had to be closed over 1000 times. Imagine the problem if a design has 50 to 100 parameters with more than two values for each parameter.

Realizing that most likely the entire parameterization space cannot be crossed with functional coverage, a different approach is needed. A couple of options exist to ensure functional coverage is adequately achieved relative to the various parameterizations. The first step is to cover all parameter settings, but without crossing them. This verifies that all parameter values have been tested. To further validate the parameter coverage, only important parameters or parameters that have a direct effect on another should be crossed. Finally, specific functional coverage items should be crossed with any parameters that affect that functionality.

Another issue when dealing with parameterized classes and coverage comes into play when trying to merge coverage results. The problem is that each specialization of a parameterized class creates a new type. If a covergroup is defined in a parameterized class, then each covergroup will be part of its own type and will not merge correctly. To help with this issue, covergroups should not be defined in parameterized classes. A parameterized class can contain a non-parameterized class, which then contains a covergroup without affecting merging.

5. SIMULATION THROUGHPUT OPTIMIZATION

Even by minimizing the number of coverage items, the amount of testing is greatly increased versus a non-parameterized flow. Care must be taken to run simulations as efficiently as possible and maximize the amount of time spent running a simulation (versus compilation or optimization). The parameterized test approach allows the design and testbench to be compiled once, with +OMV_TESTNAME used to change which test is run. This technique is standard for non-parameterized OVM testbenches and enables regressions to be as efficient as possible.

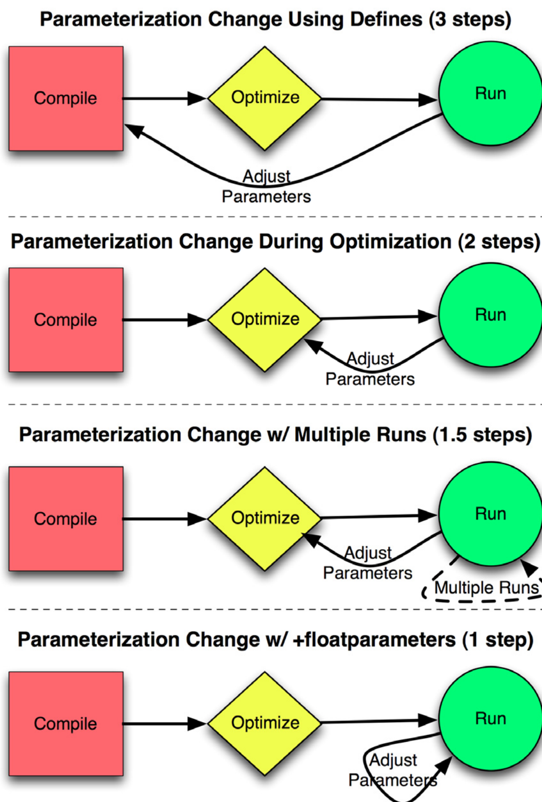

Figure 1. Simulation Flow Options when Using Parameters

There are various approaches to compiling, optimizing, and simulating a design that affect performance differently. One approach is to require all three of these steps to be rerun whenever the parameterization changes. This would be the case if the parameters were not passed through the testbench, as described in this paper, but instead required the design to be recompiled after each change. One example of this is if the parameters are tied to a *define* at the top-level, and those defines control the values of the parameters. Defines are compile time constants and, consequently, require the entire DUT and testbench to be recompiled when a new set of parameters is needed. The flow using this method is to select a parameterization, compile, optimize, and then simulate a suite of tests. Optionally, those tests could be rerun using the same parameterization (and thus avoiding recompilation) but with different seeds. Then a new parameterization could be chosen which would require recompilation, and finally a new set of simulations could take place. This process would then repeat until all parameter and functional coverage has been achieved. Requiring recompilation whenever the parameterization changes is the least efficient approach.

A more efficient method is a process which only requires two steps when changing parameters. In most simulator flows parameters are fixed during the optimization step. Most simulators also provide a mechanism for setting which parameters should be used when optimizing the design and testbench. Utilizing this knowledge, the flow can be optimized further, which results in compiling the code once. After compilation, a parameterization is selected and the design is optimized. A suite of tests can then be run using the optimized version of the design. Using +OVM_TESTNAME allows different tests to be selected and run with this optimized version of the design. After that parameterization has been simulated, a new parameterization can be selected and the design can be re-optimized. Once optimized, that parameterization is then simulated and the process repeats until all parameter and functional coverage has been achieved. By compiling once, optimizing a few times, and simulating many times, more time is spent where it is needed, resulting in a more efficient simulation and a faster time to coverage closure.

A variation of the “2-step” method of optimizing and simulating only when parameters change is to simulate the test suite multiple times (with different seeds) for a given parameterization. This “1.5-step” technique is more efficient than the standard 2-step process because each run of the test suite does not require a re-optimization. This approach works well when parameter coverage is easier to close than functional coverage because less time is spent optimizing and more time is spent running tests. The number of times a test suite is simulated per parameterization can be tuned using this method so that closure of functional and parameter coverage occurs in relatively the same amount of time. This leads to spending the minimal amount of time on the optimizations necessary to close parameter coverage while maximizing the amount of time for running tests to close functional coverage, resulting in the most efficient path to combined coverage closure.

These first three methods were benchmarked to demonstrate the efficiency improvements that can be achieved. In all three cases, the test suite consisted of seven tests. The test suite for each of the three methods was run 100 times for a total of 700 tests run. In the 3-step method, the design was compiled and optimized, and the test suite was simulated once for each of the 100 different parameterizations. The 2-step approach compiled only once but did re-optimize the design and run the test suite once for each of the 100 different parameterizations. The 1.5-step method also compiled only once, but it ran the test suite four times per optimization. Thus only 25 different parameterizations and optimizations were run, but 700 tests were still simulated.

Method	3-Step	2-Step	1.5-Step
Number of Compiles	100	1	1
Number of Optimizations	100	100	25
Number of Simulations	700	700	700
Total Minutes	416	373	341
Performance (compared to 1.5-step)	22% slower	10% slower	—

Table 1. Parameter Flow Comparisons

As shown in the Table 1, the approach that was able to run the 700 tests in the shortest amount of time was the 1.5-step method. This makes sense because it spent the least amount of time compiling and optimizing. The next fastest was the 2-step method since it only compiled once. This approach was 10 percent slower than the 1.5-step method. The real difference between these two approaches is that the 2-step technique ran four times the number of different parameterizations than the 1.5-step technique. (The four times difference in parameterization was an arbitrary number chosen for benchmarking). In other testbenches, the number of parameterizations per run of the test suite may vary, and as a result the improvement over the 3-step process will also vary. Because the 3-step approach must recompile and re-optimize every time a new parameterization is required, it is the least efficient method. It is 12 percent slower than the 2-step method and 22 percent slower than the 1.5-step method.

One thing to note is that this testbench is small and efficient and the tests ran quickly. Even the "slowest" 3-step approach ran 700 tests with 100 compilations and 100 optimizations in 416 minutes, which averages to less than one minute per test. Therefore, the amount of time spent in compilation and optimization is large, relative to the amount of time spent simulating a test. If tests run for hours rather than seconds, the compilation and optimization time will be a much smaller portion of the overall time, resulting in less improvement.

Although not benchmarked, another method requiring only a single step when changing parameters is to allow the parameters to float during the optimization step. Parameters can then be changed at run time without recompiling. A feature in the Mentor Graphics® Questa® verification environment makes this throughput optimization possible. It allows the setting of parameter values to be deferred until the simulation engine is invoked. To do this, the optimization engine needs to pass the *+floatparameters* switch. To maximize performance, the switch should be set up to minimize the number of parameters that are floating.

In some cases, using two steps when changing parameters is faster than using only a single step. This is because leaving parameters floating limits the amount of optimization that can be performed. If the parameters will be set once in optimization and then a thousand tests run, then that is probably faster. If ten tests are run with a specific parameterization and then a new parameterization is selected, then using *floatparameters* is probably faster. A crossover point exists where the extra simulation time incurred from using *floatparameters* adds up and passes the amount of time it takes to do an optimization. Regardless of the final process chosen, it is very clear that multiple simulation runs require only a single compile step.

6. PARAMETERIZED VIRTUAL INTERFACES

Another item to consider is how to deal with the interfaces that are parameterized. Instantiating the interface with the proper parameter values is straight forward. What is a little more complicated is how to associate the virtual interface handle—which points to the parameterized interface—with the OVM components that need the handle to operate. Drivers and monitors are the OVM components that generally need a handle to the virtual interface. The OVM configuration space is the best way to get the handles where they are needed. An object can be created with a virtual interface handle that is parameterized with the same parameters as the interface itself. That object can then be placed into the configuration space where any driver or monitor can access it.

Any easy way to accomplish this process is to use the *ovm_container*, which is available for download on http://ovmworld.org in the user contributions area. The *ovm_container* is a parameterized class itself that contains some static functions used to aid interaction with the configuration space. Our testbench module example was missing the interface connections required to enable communication between the design and the testbench. Adding the interface and the *ovm_container* results in the following:

```
module testbench ();
  parameter BUS_WIDTH = 32;
  parameter ADDR_WIDTH = 4;

  intf1 #(BUS_WIDTH, ADDR_WIDTH) i_intf1();
  dut #(BUS_WIDTH, ADDR_WIDTH) i_dut(i_intf1);

  typedef test1 #(BUS_WIDTH, ADDR_WIDTH)
    test1_t;

  initial begin
    //Use ovm_container to put the instance of
    //intf1 into the OVM Config space
    ovm_container #(virtual intf1 #(
      BUS_WIDTH, ADDR_WIDTH)
       )::set_value_in_global_config(
      "intf1", i_intf1);

    run_test("test1");
  end
endmodule
```

The static function *set_value_in_global_config()* creates a wrapper object and places the *i_intf1* handle into that wrapper object. The wrapper object is then put into the OVM configuration space with a value "*" set for its path and the configuration name *intf1*. This means that anyone can access the interface instance *i_intf1* as long as they look up the configuration name *intf1*. Another option would be to use a specific path instead of "*" when adding the virtual interface container object into the configuration space. This would allow tight control over which objects can see and utilize the virtual interface and, potentially, aid in debugging when printing the configuration information available to an object.

With the virtual interface handle now available in the configuration space, the driver and monitor can easily access the necessary signals. To do this, their code would look like this:

```
class monitor #(int BUS_WIDTH = 16,
                int ADDR_WIDTH = 5)
  extends ovm_monitor;

  `ovm_component_param_utils(monitor#(
    BUS_WIDTH, ADDR_WIDTH) )

  virtual intf1 #(BUS_WIDTH, ADDR_WIDTH)
   intf1_h;

  function void build();
    super.build();
    intf1_h = ovm_container #( virtual intf1
      #(BUS_WIDTH, ADDR_WIDTH) )::
      get_value_from_config(this, "intf1");
  endfunction : build

endclass : monitor
```

Using the configuration space with wrappers for parameterized virtual interfaces allows for easy transportation of the virtual interface handle to the location that it is needed.

7. PARAMETER PASSING

Now that a testbench can be created that shares parameters with the design, the method of passing the parameters through the testbench hierarchy can be improved. This is especially true for cases where a large number of parameters are involved. Maintaining the lists of parameters is both time consuming and error prone. Adding and/or removing a parameter manually involves editing numerous files. Additionally, the code becomes cluttered and hard to read as the list of parameters grows. To avoid these issues, a set of macros can be defined to help ease the burden.

Three macros make up the set. The first macro expands out to become the definition of all the parameters in the testbench. The second macro is used when creating types and handles. This macro contains the mappings of all the macros from the object, module, or interface to the type or handle that is created within the object, module, or interface. The third macro provides a string representation, which is useful for creating messages. Using the code presented above, these three macros are defined in a *params_defines.svh* file and look like this:

```
// Declarations
`define params_declare #(int BUS_WIDTH = 16,
                         int ADDR_WIDTH = 5)

// Instantiations / Mappings
`define params_map #(.BUS_WIDTH (BUS_WIDTH),
                     .ADDR_WIDTH (ADDR_WIDTH) )

// String Value
`define params_string $sformatf("#(%1d, %1d)",
 BUS_WIDTH, ADDR_WIDTH)
```

Using these macros results in the testbench module looking like this:

```
module testbench ();
  `include "params_defines.svh"

  parameter BUS_WIDTH = 32;
  parameter ADDR_WIDTH = 4;

  intf1 `params_map i_intf1();
  dut `params_map i_dut(i_intf1);

  typedef test1 `params_map test1_t;

  initial begin
    //Use ovm_container to put the instance of
    //intf1 into the OVM Config space
    ovm_container #(virtual intf1
      `params_map )::
      set_value_in_global_config(
      "intf1", i_intf1);

    run_test("test1");
  end

endmodule
```

The monitor is now defined as follows (the package that contains the monitor includes the *params_defines.svh* file):

```
class monitor `params_declare extends
  ovm_monitor;

  `ovm_component_param_utils(monitor
    `params_map)
  virtual intf1 `params_map intf1_h;

  function void build();
    super.build();
    intf1_h = ovm_container #(
     virtual intf1 `params_map)::
     get_value_from_config(this, "intf1");
  endfunction : build

endclass: monitor
```

These macros are defined only once and every class and object gets the full complement of parameters. This may result in parameters being added to a class that has no use for them. This is not an issue as the extra parameters will be ignored and not produce any side effects, while the needed parameters will always be available.

8. CONCLUSION

Using the techniques described above, a parameterized OVM testbench can be created that leverages all of the capabilities of OVM, including the reuse advantage of compiling once and running multiple times. These techniques also reduce code complexity and increase code readability when parameters are involved.

9. ACKNOWLEDGMENTS

Our thanks to: Todd Burkholder, Senior Writer, Mentor Graphics, for editorial support; Stacey Secatch, Senior Staff Verification Engineer, and Cristi Lovett, Design Engineer, Xilinx, for helping define and implement this solution; Adam Erickson, Verification Technologist, Mentor Graphics, for helping with the technical aspects of creating parameterized OVM tests.