Parameters, UVM, Coverage & Emulation – Take Two and Call Me in the Morning

Michael Horn
Mentor Graphics Corporation
Colorado, USA
Mike_Horn@mentor.com

Bryan Ramirez
Mentor Graphics Corporation
Colorado, USA
Bryan_Ramirez@mentor.com

Hans van der Schoot
Mentor Graphics Corporation
Ottawa, Canada
Hans_vanderSchoot@mentor.com

***Abstract*- Parameterized IP continues to expand in usage. Parameterized IP also continues to expand in size which compounds verification complexity. Previous DVCon papers have addressed issues related to using parameters with UVM [1] and employing code, functional and assertion coverage with parameterized IP [2]. They focused on these aspects for conventional simulation. However, with the growing use of emulation for hardware-assisted simulation acceleration, additional considerations emerge. What capabilities does the use of an emulator facilitate, or perhaps impede? How can this be exploited to verify parameterized IP faster and more efficiently?**

I. Introduction

Semiconductor process technology and FPGA platforms continue to expand the number of transistors and gates that are available for engineers to exploit. The only way to utilize this additional capacity and still meet tight schedules for aggressive time-to-market windows is via the use of Intellectual Property (IP). Consequently, IP reuse is essential to the expansion of the semiconductor industry. Where does all this IP originate? It may come from companies dedicated to developing IP. It may come from FPGA vendors. It may be developed internally within a company.

Regardless of origin, most IP will be used in more than one context. This drives IP creation to yield flexible designs that are parameterized to enable varying configurations to fit different intended use cases. SystemVerilog parameters bring great modeling power, but they also bring great responsibility to design and verification engineers to ensure that parameterized models function correctly. Some of the areas of responsibility have already been investigated in previous DVCon papers [1] [2].

One paper, entitled "Parameters and OVM — Can't They Just Get Along?" [1] discussed various techniques associated with parameters and their interaction with an OVM based environment. Do these techniques continue to work with UVM? What about when emulation is a requirement to accelerate UVM? The following sections will revisit the original paper and address these questions. This involves such aspects as the configuration space, parameterized tests, parameterized virtual interfaces and parameter passing while creating a single unified testbench that can work in both simulation and emulation.

Another paper, entitled "Relieving the Parameterized Coverage Headache" [2] explored the topic of coverage for parameterized IP. The paper content remains largely applicable in the contexts of both UVM and emulation, though some adaptations are necessary. A specific area of emphasis is achieving parameter coverage within a dual domain environment as is required for emulation. Another area of exploration is the employment of SystemVerilog covergroups and cover directives on an emulator to achieve coverage closure faster and more effectively.

The work captured in these earlier papers provide valuable lessons, especially when updated for UVM and co-emulation. When an emulator is part of the verification strategy, what unique opportunities does it provide? The emulator brings orders of magnitude improvements in run time performance. This is one of the reasons why an emulator becomes a requirement for a verification effort. The emulator is also a huge resource of parallel execution units. How can that parallelism be exploited? To access the performance and capacity of an emulator, a dual top testbench architecture must be employed. How do two top levels impact parameter usage in the design and a UVM testbench? With the emulator utilizing real hardware, more detailed analysis and compile are required which consequently takes more time than simulation compiles. Targeting real hardware also brings certain limitations on what parts of the SystemVerilog language can be harnessed. How can these considerations be mitigated to allow for full design and parameter exploration? These questions and more will be investigated and answered below.

II. Parameters, UVM and Emulation

The paper "Parameters and OVM — Can't They Just Get Along?" [1] explores the usage of OVM for verifying highly parameterized IP. It is expected that the reader be familiar with that paper as information is principally not duplicated here. Most sections of the original paper will be visited and updated for both UVM and co-emulation with

the notable exception of the "Coverage with Parameters" section from [1]. That topic is fully covered by the second paper which is visited in Section IV.

A. UVM Configuration Space

The original paper [1] first discusses the usage of the OVM configuration space when parameters are used with a DUT. Usage of SystemVerilog parameters is required in some circumstances such as to control bus and address widths. Parameters are also used in conjunction with generate statements to control if a portion of a parameterized IP should exist in a configuration of the design or not. As originally recommended, if a configuration control must be a parameter because of SystemVerilog rules [5] such as generate statements and bus width definitions, then continue to use a parameter. If a mechanism exists to have the desired controllability and functionality without using a parameter, then avoid a parameter.

The OVM configuration space only allowed for strings, integral types and object handles. This limitation implies that configuration information had to be wrapped in an object to be placed into the configuration space if it was not a string or integer. UVM does not restrict the type of information that can be placed into the configuration space thus providing more flexibility. However, even with the added flexibility, good practice is to use an object to group configuration attributes together. This allows for easier randomization of configuration values as well as class-based coverage sampling.

B. Parameterized Tests

In both OVM and UVM, classes are registered with the factory to enable easy construction and potential replacement. In both OVM and UVM, the factory provides both a type-based and a string-based mechanism for maintaining the list of registered classes. The former is invoked when making a UVM call such as `<class_name>::type_id::create()` which is the primary user API. A parameterized class when registered with the factory using the `` `ovm_component_param_utils() `` or `` `uvm_component_param_utils() `` macro only registers with the type-based factory. `run_test()` only uses the string-based factory when instantiating the test selected to be run. A conflict exists here since we have the type-based factory used for registration, but the string-based factory is used for construction.

As described in [1], we need to expand the macro and augment it to explicitly register a parameterized test with both the type and the string-based factories. By expanding the `` `uvm_component_param_utils() `` macro for a hypothetical parameterized test called test1, we would end up with the following:

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

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

  virtual function uvm_object_wrapper get_object_type();
    return type_id::get();
  endfunction

  ...
endclass : test1
```

This code then needs to be modified to add in the second parameter argument to the uvm_component_registry typedef. This second argument is what registers the class with the string-based factory which is exactly what is needed. We also add a type_name data member and a function that returns that type_name data member to be consistent with the unparameterized factory registration. The needed additions are highlighted in blue.

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

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

  virtual function uvm_object_wrapper get_object_type();
    return type_id::get();
  endfunction

  const static string type_name = $sformatf("test1 #(%1d, %1d)",
                                             BUS_WIDTH, ADDR_WIDTH);

  virtual function string get_type_name ();
    return type_name;
  endfunction // get_type_name

  ...
endclass : test1
```

With the factory registration updated for UVM, a typedef is now needed to specialize the `test1` class in the top level. For a simulation-only testbench, a single top level module is generally used. To create an emulation-ready testbench, two top levels are required [3] [4]. Since `run_test()` is called from the HVL top level, that is also where typedefs need to be placed for any parameterized tests. The description of how this mechanism works is given in [1].

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

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

  typedef test1 #(.BUS_WIDTH(BUS_WIDTH), .ADDR_WIDTH(ADDR_WIDTH)) test1_t;

  initial begin
    run_test("test1");
  end
endmodule : hvl_top
```

When more than one parameterized test is defined, as is often the case, each parameterized test must be specialized with a typedef in the hvl_top module. With the specializations in place, the standard +UVM_TESTNAME mechanism can be used to select each parameterized test from the command line. The name that is used is the second parameter passed to the `uvm_component_registry` which in this case is "test1".

C. Regression Throughput Optimization

Many options exist for setting parameter values for each simulation/co-emulation run [1]. The original options included using `defines and tool switches along with multiple runs for a single parameter configuration. A pure simulation flow provides the most flexibility. When using a dual-top, emulation-ready testbench architecture, additional considerations are in order. Longer compile times and reduced flexibility must be pondered when determining how to set parameter values.

One option that is still possible in a co-emulation context is to utilize `defines for controlling the parameter values. As before in [1], when `defines are used a recompile is required to change parameter values. With an emulator in use, that means re-launching two compile flows; one for the simulator side and one for the emulator side as shown in Figure 1.

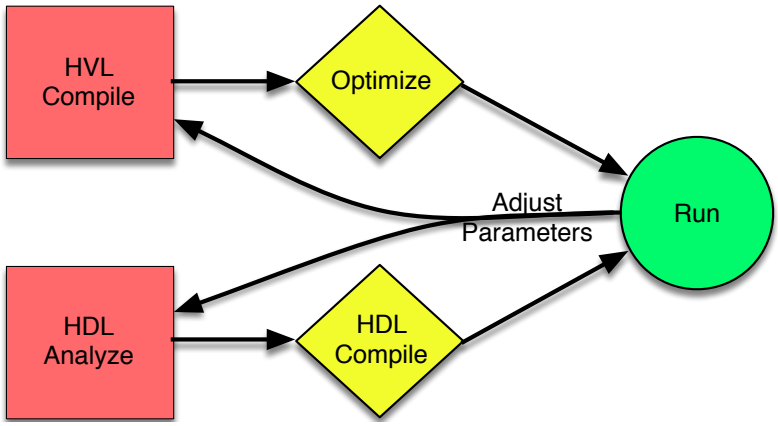

Figure 1. Parameterization Change Using Defines

The `defines should be placed into a single file that is shared between the HVL and HDL top level modules. With that in place the flow is to select a parameter configuration by setting the `define values, then compile, optimize, and ultimately simulate a suite of tests.

Using `defines will do the job, but at the expense of a full recompile with every parameter configuration change. An alternative that can reduce some of the compilation overhead would be to use switches provided by tool vendors. Most tools allow for parameter values to be specified at optimization/HDL compile time. This means that compiling the HVL code and analyzing the HDL code only has to be done a single time. This is a potentially significant time savings when parameter values are changed as shown in Figure 2.

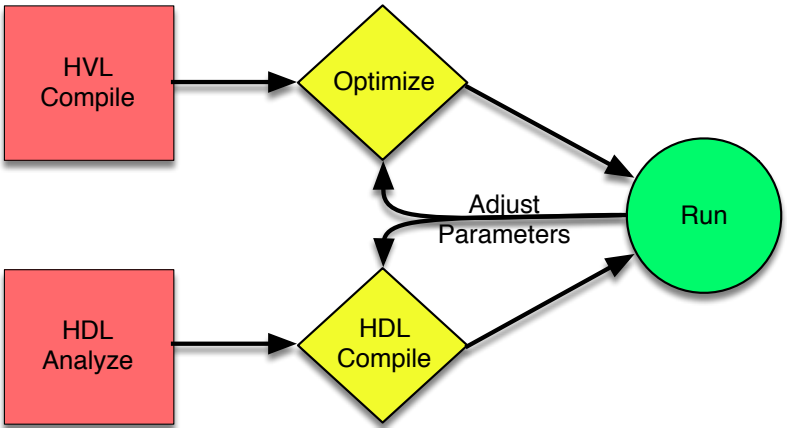

Figure 2. Parameterization Change Using Tool Switches

Using tool switches to set a parameter configuration involves two tool flows. It would be advantageous to have a single place to specify parameter values that are then passed to each tool flow. With the `define flow, a single file was used with `define values that was then `included into each top level module. To get the same functionality, several options exist depending on tool support. Parameter values could be controlled from a common script and/or makefile. The script/makefile would output files that are specific to the HVL and HDL compile flows. Another option is if both HVL and HDL compile flows understand the same format for specifying parameter value changes, allowing a single file to be used that can then be passed to both compile flows.

Regardless of the choice between using `defines or tool switches, multiple runs should be performed with each iteration of the compile flow as illustrated in Figure 3. This can be accommodated by different SystemVerilog seeds being supplied to the run command or different tests being chosen using the +UVM_TESTNAME plusarg. Such

flexibility is still available when using an emulator because the seed value and test selection is done in the HVL top level which remains in a SystemVerilog compliant simulator.

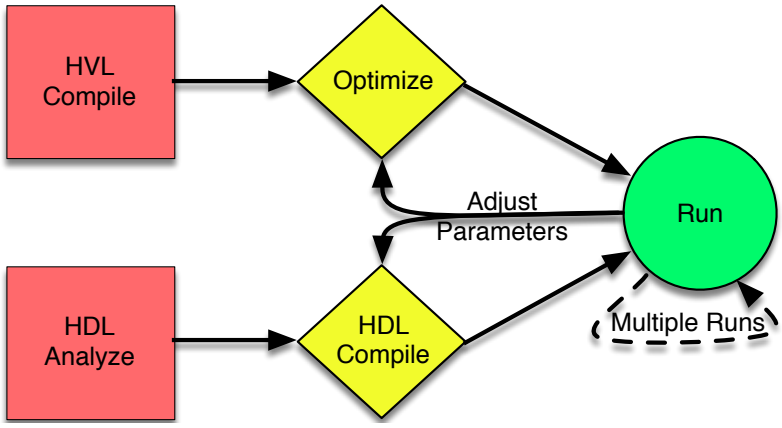

Figure 3. Multiple Run Optimization

D. Parameterized Virtual Interfaces

If a bus size of a design port is controlled by a parameter, the testbench will also need to understand any bus size changes to be able to communicate properly with the design. SystemVerilog-based testbenches usually use virtual interface handles pointing to interfaces to make the pin-level connection to the design. Hence, these interfaces will also need to be parameterized and share parameter values. How do you get the parameterized virtual interface handles first defined and then subsequently passed to the testbench?

With OVM an object wrapper must be created to pass the parameterized virtual interface handle because the OVM configuration database only supports integral values, strings and object handles [1]. With UVM, the configuration database allows for any type to be amassed including parameterized virtual interface handles. This means that no special accommodations need to be made other than to get the correct parameter values to the correct locations which is discussed further in sub-section E.

With parameterized virtual interface handle passing sorted, it needs to be determined what exactly the parameterized virtual interface handles will be pointing to. This is where emulation changes the picture a bit. As illustrated in Figure 4, in pure simulation, a driver contains all pin-wiggling code and a single interface contains the pins. With emulation, the pin-wiggling code is extracted from the class based environment and placed into a BFM interface (Monitor BFM and Driver BFM in Figure 4) to make it part of the HDL domain. The pin-wiggling code still interacts with a pin interface that contains the pins to maintain proper separation of concerns [3] [4].

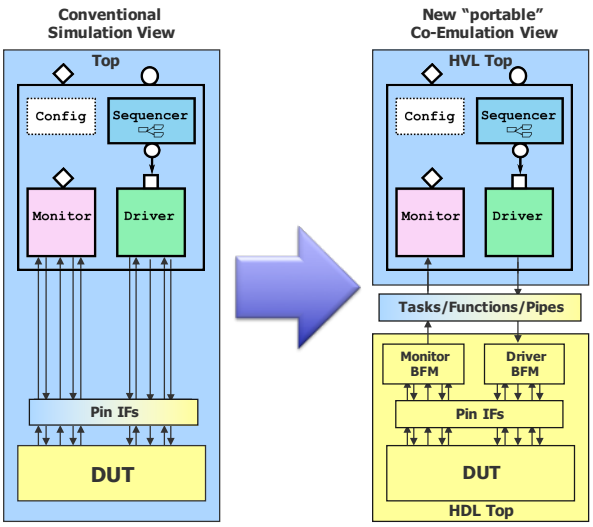

Figure 4. Co-Emulation Agents

With the UVM configuration mechanism and the additional SystemVerilog interfaces, two calls are made to place monitor BFM and driver BFM parameterized virtual interface handles into the configuration database from the HDL side. This results in code that is similar to the following:

```
interface driver_bfm #(int BUS_WIDTH = 32, ADDR_WIDTH = 4)
    ( pin_intf #(BUS_WIDTH, ADDR_WIDTH) );
  ...
endinterface : driver_bfm

interface monitor_bfm #(int BUS_WIDTH = 32, ADDR_WIDTH = 4)
    ( pin_intf #(BUS_WIDTH, ADDR_WIDTH) );
  ...
endinterface : monitor_bfm


module hdl_top ();
  parameter BUS_WIDTH = 32;
  parameter ADDR_WIDTH = 4;

  pin_intf #(BUS_WIDTH, ADDR_WIDTH) i_pin_intf(...);

  driver_bfm #(BUS_WIDTH, ADDR_WIDTH)  i_drv_bfm (i_pin_intf);
  monitor_bfm #(BUS_WIDTH, ADDR_WIDTH) i_mon_bfm (i_pin_intf);

  initial begin
    import uvm_pkg::uvm_config_db;
    uvm_config_db #(virtual driver_bfm #(BUS_WIDTH, ADDR_WIDTH))
      ::set(null, "*", "drv_bfm", i_drv_bfm);
    uvm_config_db #(virtual monitor_bfm #(BUS_WIDTH, ADDR_WIDTH))
      ::set(null, "*", "mon_bfm", i_mon_bfm);
  end

endmodule : hdl_top
```

Focus on the `uvm_config_db` call. The `uvm_config_db` class is a parameterized class with a parameter specifying the SystemVerilog type that is employed. In this case the type is itself a parameterized type that is a parameterized virtual interface handle. Each call of the `set()` function is now placing a parameterized virtual interface handle of the monitor BFM and the driver BFM, respectively, into the configuration database to be extracted by the testbench at the appropriate location.

The testbench also will be accessing a function defined in the `uvm_config_db` parameterized class. This means that the testbench needs to specify the same parameterized virtual interface handle with the same parameter values to be able to access values stored with a `set()` call. This looks as follows:

```
class monitor #(int BUS_WIDTH = 16, int ADDR_WIDTH = 5) extends uvm_monitor;
  `uvm_component_param_utils( monitor #(BUS_WIDTH, ADDR_WIDTH) )
  virtual monitor_bfm #(BUS_WIDTH, ADDR_WIDTH) m_bfm;

  function void build();
    uvm_config_db #(virtual monitor_bfm #(BUS_WIDTH, ADDR_WIDTH))
      ::get(this, "", "mon_bfm", m_bfm);
  endfunction : build
endclass : monitor
```

The question is now how to get the parameter values to the monitors, drivers, agents, etc. to allow them to pull the correct parameterized monitor and driver BFM handles from the configuration database.

E. Parameter Passing

As discussed in [1], the easiest way to work with all the parameters and to pass them through the testbench hierarchy starting at the test is to use a set of three macros. The three macros are shown here:

```
// 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)
```

The first macro expands 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 macros results in more concise code. Following are the HVL and HDL top levels re-coded to use the macros:

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

  dut `params_map i_dut(...);

  typedef test1 `params_map test1_t;

  initial begin
    run_test("test1");
  end
endmodule : hvl_top
```

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

  pin_intf `params_map i_pin_intf(...);

  driver_bfm `params_map i_drv_bfm (i_pin_intf);
  monitor_bfm `params_map i_mon_bfm (i_pin_intf);

  initial begin
    import uvm_pkg::uvm_config_db;
    uvm_config_db #(virtual driver_bfm `params_map)
      ::set(null, "*", "drv_bfm", i_drv_bfm);
    uvm_config_db #(virtual monitor_bfm `params_map)
      ::set(null, "*", "mon_bfm", i_mon_bfm);
  end

endmodule : hdl_top
```

Testbench code also becomes more concise as shown here:

```
class test1 `params_declare extends test_base_c;
  typedef uvm_component_registry #(test1 `params_map, "test1") type_id;

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

  virtual function uvm_object_wrapper get_object_type();
    return type_id::get();
  endfunction

  const static string type_name = {"test1 “, `params_string};

  virtual function string get_type_name ();
    return type_name;
  endfunction // get_type_name

  ...
endclass : test1
```

The original paper did not consider UVM or emulation when defining those macros. Fortunately, no changes are needed as the macros are completely orthogonal to OVM or UVM usage and they can be used seamlessly with both the simulation and emulation compilation flows.

III. Parameters, Coverage and Emulation

The paper “Relieving the Parameterized Coverage Headache” [2] explores all aspects of coverage in the context of a parameterized design and testbench. The majority of the paper describes how to develop coverage models when parameters are involved for a class-based testbench. Those techniques are still valid and supported since the class based testbench continues to reside and run in a SystemVerilog compliant simulator. Some techniques for module-

based coverage are also described [2] which are revisited here in particular because the modules end up being part of the HDL top level code base that will go onto the emulator.

A. Module-Based Coverage

Module-based coverage is generally used when covering internal aspects of a design [2]. Such coverage might be composed of SystemVerilog covergroups or SystemVerilog assert and cover properties. The modules that contain these constructs are generally bound into a design using the SystemVerilog `bind` statement. Additionally, these modules can contain generate statements that conditionally declare a covergroup, assertion and/or cover directive. The same name can be retained for each of the items in the generate options which then allows for merging with different parameter values as illustrated in [2]. Since the merge algorithm is not defined in the SystemVerilog LRM [5], a savvy tool choice will give the best results. With an emulator as part of the flow, again a savvy tool choice is required to ensure that statements such as `bind`, `generate`, `assert`, `cover`, etc. are supported by the emulator and the emulator tool flow.

When the emulator is part of the verification effort, additional thought must go into the creation and sampling of the coverage model in the synthesizable HDL domain. With emulator hardware, the synthesis step is performed to transform HDL code including coverage code to fit into the hardware. In general, hardware synthesis places modeling limitations on all aspects of the SystemVerilog language including the coverage constructs which must be considered when defining the HDL side coverage model. A savvy emulator choice will facilitate this by providing support for a rich subset of the coverage constructs.

In a pure simulation environment, it is generally not an issue to pass data collected via a module or interface bound into the design to the UVM-based testbench as the entire design and testbench all live in the same memory footprint. With an emulator in the picture, the design lives in the emulator and the testbench lives in memory on a server. To get coverage data to the testbench is now not just a simple matter of passing a pointer around. Instead, the entire coverage data set must be passed from the emulator to the simulator which results in additional overhead and potentially slower co-emulation run times. To eliminate this issue, some coverage data that used to reside in a testbench class, should now also be moved to a module/interface so that it can be bound into the design hierarchy and become part of the emulator image. This removes the overhead while still allowing for sampling of the required coverage data. At the end of the co-emulation run, a single transfer of the coverage data will take place instead of constant transfer throughout the simulation.

IV. Parameters and Emulation

When working to verify a parameterized IP with an emulator, the benefits of an emulator should be maximized. As discussed earlier, an emulator runs fast, though time must be spent up front to compile and target the emulator hardware. Consequently, strategies must be devised where the design can be compiled once while enabling as many emulation runs as possible.

A. Multiple Configurations Compiled Together

When working at a parameterized IP level, the design is generally going to be smaller than the capacity of an emulator. A natural thought is how this capacity can be more fully utilized to give more verification throughput. One approach is to instantiate multiple configurations of the same parameterized IP in the same top level that would result in code like the following:

```
module hdl_top ();
  parameter BUS_WIDTH1 = 32;
  parameter ADDR_WIDTH1 = 4;

  parameter BUS_WIDTH2 = 16;
  parameter ADDR_WIDTH2 = 2;

  param_design #(BUS_WIDTH1, ADDR_WIDTH1) i_dut1 (...);
  param_design #(BUS_WIDTH2, ADDR_WIDTH2) i_dut2 (...);

endmodule : hdl_top
```

More than two instances could be created if capacity and especially design configuration flexibility allows. The latter factors into our available options for how to stimulate and check the design functionality.

1) Optional Functionality

In many cases a design operates exactly the same irrespective of parameter configuration with the exception of some functionality that is conditionally present based on generate statements in a design. For example, a design might have a parity bit or a CRC calculation in one configuration with some simple logic to check/generate the parity and report any errors that are detected. If the rest of the design behaves the same, then why not inject the same stimulus into two versions of the same design.

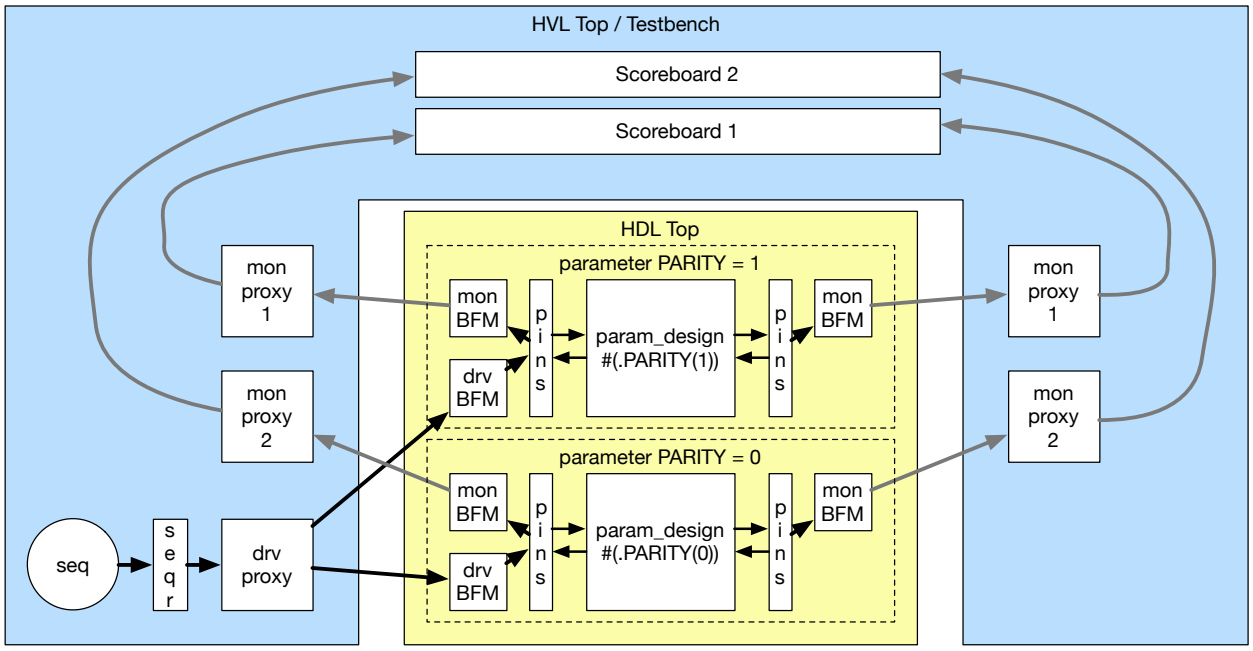

Figure 5. Same Stimulus, Different Analysis

As shown in Figure 5, the same stimulus is applied to two different driver BFM instances. The driver BFMs are parameterized to understand if parity needs to be generated or not. To get the same stimulus to two different BFMs requires some modification in the driver proxy. The driver proxy now must have handles to all of the driver BFMs and send data to each driver BFM as it comes in from the sequencer. This may not be the most performance optimal means of distributing the stimulus to multiple driver BFMs since it requires two round trip calls between the simulator and the emulator. Depending on the stimulus, a storage structure could be created on the emulator side to accept a full stimulus transaction and then distribute the transaction on the emulator side.

The analysis portion of the testbench also has to undergo modifications. With two (or more) configurations of the design are being exercised simultaneously, two independent scoreboards need to be in place to ensure that any problems with a design instance are isolated.

Coverage will be design dependent. In some cases, two coverage instances will be required and in other cases a single coverage instance should be collecting coverage for both instances of the design. More detail on usage of each case is available in [2]. If the coverage should be combined, then a `uvm_component` derived class will need to be defined that can handle data from both designs and then sample a single covergroup. The `type_option.merge_instances = 1` control could also be used to get combined coverage when multiple instances of a covergroup are used.

2) Non-Overlapping Configuration

Often a parameterized design can not have the same stimulus applied to it with the same timing. A simple example would be transferring a 1024-bit packet of data over a 32-bit bus or a 16-bit bus. The same data is transferred, but the 16-bit bus version will take twice as long as the 32-bit bus version. If a scheme was used as shown in Figure 5, then there would be very large gaps between data transfers when sending data to a 32-bit bus. This would not fully exercise any back-to-back corner cases that may exist in the design when parameterized to 32-bits. In cases where the

functionality and or timing does not align, a different option can be explored. The overhead of compilation for the emulator can still be minimized by instantiating multiple versions of the design, but now with a muxing scheme inserted to allow for selection of the design configuration a testbench will exercise. This will enable more co-emulation runs without having to recompile.

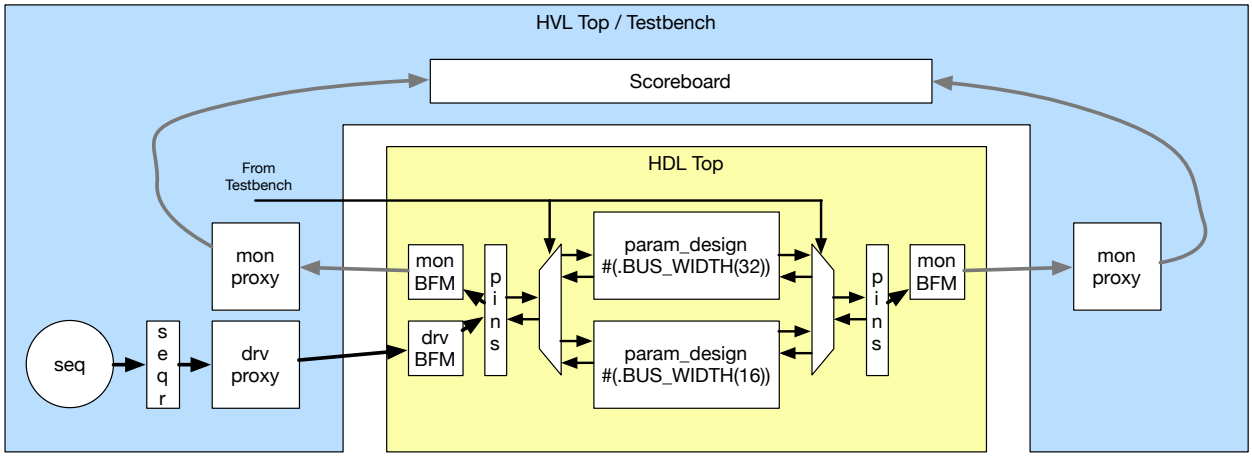

Figure 6. Muxes Control Design Activation

With the muxes in place, the testbench now utilizes a single driver BFM and a single monitor BFM. With the example from above, the BFMs will be built to handle the maximum size of data transfers available in a design configuration which means that buses and logic will utilize the maximum size. The BFMs will then be configured from the testbench to understand how much of a bus is actually valid to be used in the specific testcase. This information is passed across from the testbench during initialization time as the testbench now determines which design configuration will be active. The testbench will also control which design configuration is connected to the BFMs.

On the analysis side, a single checking path exists that again must be aware of the design configuration being exercised. This information can easily be passed in a configuration object as described in both [1] and [2]. Coverage can also utilize the same configuration object to determine what needs to be sampled and how.

Other mechanisms such as SystemVerilog plusargs could also potentially be used to control the mux select lines.

B. Parameter Sharing

When working with an emulator, two top level modules are created. How are parameters shared between the two top levels and between the two compile flows? As discussed above, `defines could be used as well as tool switches. Another option that is usually available is to place parameters in a SystemVerilog package. The SystemVerilog package can then be shared and compiled for both top levels. This results in a single location where parameters can be defined and manipulated.

C. Staged Configurations

In most cases an emulator is a shared resource that is not in abundant supply. With that in mind, it is advantageous to have the emulator performing verification runs as much as possible. This is in contrast to Linux based servers that are generally highly available. Engineers can use this situation to their advantage. Looking at the abstracted compile flow as shown in Figure 3, the only time the emulator is involved in the effort is when it comes time to actually run. Everything before that step is based on executing software on Linux-based servers. This leads to the strategy of pre-compiling as many different configurations of a design and testbench as possible and having them staged and ready to run on an emulator when it is available.

V. Conclusion

With the constant push to create faster designs more quickly, properly verified parameterized IP becomes extremely valuable. Ensuring that all the valid and viable configurations of parameterized IP is exercised in as short a time as possible greatly aids the overall goal of chip development with increased productivity and reduced risk. When a modern UVM testbench with a complete coverage model is married with emulation, they provide unmatched speed and hence parameterized IP can be verified more effectively than ever before.

References

[1] Bryan Ramirez and Michael Horn (2011) “Parameters and OVM — Can’t They Just Get Along?” DVCon 2011 - http://events.dvcon.org/2011/proceedings/papers/09_2.pdf

[2] Christine Lovett, Bryan Ramirez, Stacey Secatch and Michael Horn (2012) “Relieving the Parameterized Coverage Headache” DVCon 2012 - http://events.dvcon.org/2012/proceedings/papers/08_1.pdf

[3] Hans van der Schoot, Anoop Saha, Ankit Garg and K. Suresh (2011) “Off to the Races With Your Accelerated SystemVerilog Testbench” DVCon 2011 - http://events.dvcon.org/2011/proceedings/papers/05_3.pdf

[4] Hans van der Schoot and Ahmed Yehia (2015) “UVM & Emulation: How to Get Your Ultimate Testbench Acceleration Speed-Up” DVCon Europe 2015 - https://verificationacademy.com/news/featured-paper-dvcon-europe-2015

[5] IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language, IEEE Std. 1800-2012

[6] Open Verification Methodology (OVM) - http://ovmworld.org

[7] Universal Verification Methodology (UVM) - http://www.accellera.org/activities/working-groups/uvm