Common Challenges and Solutions to Integrating a UVM Testbench in Place of a Legacy Monolithic Testing Environment

Frank Verhoorn
Northwest Logic, Inc
1100 NW Compton Drive #100, Beaverton Oregon 97006
fverhoorn@nwlogic.com 503-533-5800

Michael Baird
Willamette HDL, Inc.
6107 SW Murray Blvd, #407, Beaverton OR 97008
mike@whdl.com 503-590-8499

Abstract

Developing a new verification testbench based on Unified Verification Methodology (UVM) to work within an existing company testing environment presents challenges beyond just creating the UVM testbench alone. Often companies will have existing scripting, reporting, logging etc. methodologies that the UVM testbench must conform to. This can require custom work above and beyond standard UVM testbench methodologies. Many of these challenges are likely to be common to companies who are upgrading older, simple Verilog-based verification environments to UVM. This paper discusses some specific challenges and solutions discovered while converting the Northwest Logic Memory testbench to UVM and integrating into existing test environment infrastructure.

I. Introduction

Northwest Logic is a vendor of logic block Intellectual Property (IP). The Northwest Logic Memory Controller IP internal code database is highly parameterized, to allow Northwest Logic to easily create and deliver unique configurations (for example deliveries with different internal datapath widths). Verification of such deliveries requires a testbench with the ability to vary parameter values at simulation time. However, heavy use of parameterization presents some unique challenges for a UVM framework containing scalable Bus Functional Model (BFM) instances.

- Recommendation 1: This paper discusses some solutions to UVM testbench issues due to parameterization.

Besides supporting parameterized simulation runs, the Northwest Logic verification infrastructure contains significant legacy support for multiple Physical Interface (PHY) models and merging Coverage data from multiple simulations.

- Recommendation 2: This paper provides some solutions and guidance for the UVM testbench toplevel hierarchy that can be beneficial for re-using existing infrastructures.

Many historical testbenches contain global register or wire declarations which communicate control/status to/from the Device Under Test (DUT). Older Verilog-based testbenches frequently assert stimulus registers/wires from many different statements in the testbench. Testbenches often contain hierarchal path references to set or query the registers/wires, or contain BFM-like task declarations to sequence assertions/deassertions of stimulus wires.

- Recommendation 3: This paper provides some guidance for implementing these operations into BFM-based access methods, in order to ease the effort of porting existing test suites to a new UVM testbench.

Finally, most Verilog testbenches make frequent use of the Verilog $display statement for outputting messages about test execution progress, fatal and nonfatal errors, and debug or diagnostic information. UVM libraries define standard macros to be used for outputting error/debug/info messaging. However, before simply converting all $display statements to UVM error/info macros, it is beneficial to define some methodology and conventions.

- Recommendation 4: This paper provides some suggested guidance and examples for converting older testbench $display statements to UVM error/fatal/info macro use.

II. Parameter-Agnostic Interface Proxies

Parameter-based object instantiation is extremely common in Hardware Description Language (HDL) - based logic designs for low-level logic block content. For example, a design might contain multiple instantiations of a primitive queue or FIFO element, where one instance might be parameterized to a different size or depth than another instance. However, parameterization used on DUT external interfaces (such as DUT external port-width parameterization), can be problematic for UVM BFM instantiations.

The traditional HDL-based portion of the testbench (HDL_TOP in Figure 1) presents no issues compiling one module of a BFM interface module but linking two uniquely-parameterized instances of that BFM interface module. However, in an object-oriented UVM testbench, these two interface instances once parameterized differently become two distinct object classes. When connecting a UVM driver to each BFM interface module, the UVM drivers themselves must be uniquely parameterized. This then necessitates a parameter-based instantiation hierarchy throughout the entire construction and link of the UVM Framework (HVL_TOP in Figure 1).

In the figure below, if two instances of interface 'A' exist, one parameterized with a transaction address width 32=X and another with transaction address width 48=Y, then the UVM framework must contain one instance of a driver using width X=32 and a separate driver using width Y=48.

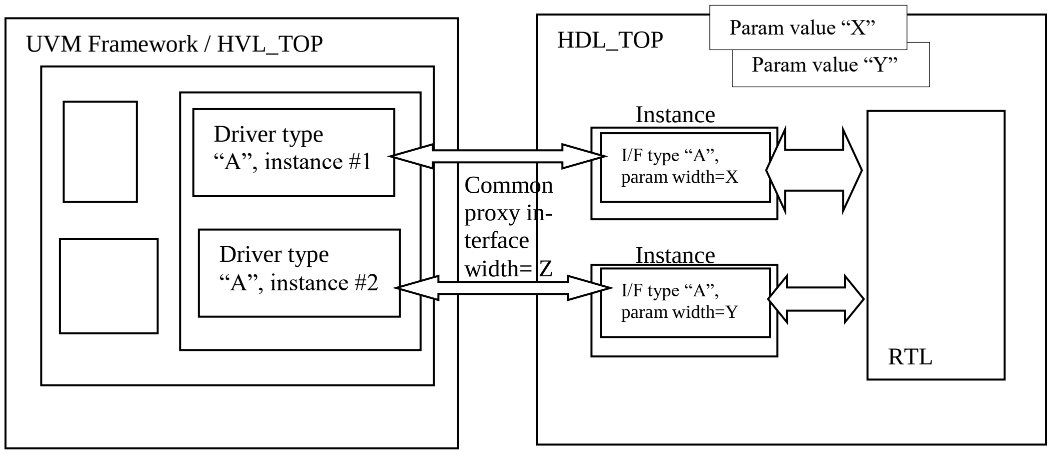

UVM Framework / HVL_TOP
Driver type "A", instance #1
Driver type "A", instance #2
Common proxy interface width= Z
HDL_TOP
Param value "X"
Param value "Y"
Instance
I/F type "A", param width=X
Instance
I/F type "A", param width=Y
RTL

Figure 1: Parameterized Interface Instances and Parameter-Agnostic Proxy Interface

Instead, a recommended solution is to define "Proxy" interfaces within each BFM interface module (see references [1][2]). The BFM interface module is defined with interface proxy methods that utilize a fixed, common width to its method arguments (e.g. proxy's address width Z=64). All driver instances may then be identical, and make calls to BFM proxy-interface methods using Z=64 wide address arguments. Within each (uniquely-parameterized) BFM interface module instance, however, the proxy interface method can then call a (uniquely-parameterized) BFM interface method. A base class is defined for objects containing the (fixed-width) proxy interface methods, with empty/placeholder method content, then BFM interface models are defined which override the proxy interface methods to call a (uniquely-parameterized) method of the BFM interface. Similar methods are employed for other parameter-based uniqueness, such as data widths.

```
interface localbus_txn_if #(
                     parameter ADDR_WIDTH=39,
                     parameter SDRAM_DSIZE=64)
…

 // BFM native (uniquely parameterized to this BFM instance) tasks
 task automatic bfm_write_burst(input logic [ADDR_WIDTH-1:0] addr  );
  …
 endtask
 task automatic bfm_set_write_data( input logic [SDRAM_DSIZE-1:0] datain_value );
  …
 endtask

  // BFM has a proxy interface object that will be registered into the UVM configDB
  class lb_proxy_if extends lb_pkg::lb_if_proxy_base;

          function new(string name = "lb_proxy_if");
             super.new(name);
          endfunction

         // Here are all the proxy-function to BFM native function mappings/calls.
         virtual task automatic write_burst(
                input logic [63:0]   addr               // fixed non-parameterized width
                );
                  // calls the uniquely parameterized task defined within this interface
                  bfm_write_burst(addr[ADDR_WIDTH-1:0]);
         endtask

          virtual task automatic set_write_data(
                input logic [1023:0] datain_value      // fixed non-parameterized width
                );
                   // calls the uniquely parameterized task defined within this interface
                   bfm_set_write_data(array_ptr, datain_value[SDRAM_DSIZE-1:0]);
         endtask
   endclass

   // above defined the proxy class, now create the instance of the proxy.
   lb_proxy_if proxy_if = new();

endinterface
```

Example 1: BFM Interface Module with Proxy Interface

Example 1 above shows representative code from a single BFM interface module for a bus "localbus" including both the native (uniquely parameterized) tasks/methods, as well as the proxy class object containing fixed-width interface methods.

On the following page, Example 2 shows the base-class proxy object defining these fixed-width interface templates before they are overridden by each uniquely parameterized BFM. Example 3 demonstrates the uniquely parameterized BFM instantiated in the HDL_TOP and registration of its proxy interface object into the UVM configDB. Finally, Example 4 demonstrates the actual UVM driver fetching the handle of the specific BFM's proxy interface and using a $cast to a non-parameterized base proxy interface.

Example 2: Interface Proxy Base Class Declaration

```
class lb_if_proxy_base extends if_proxy_base;

  // Here are all the (empty) proxy-function to BFM native function mappings/calls.
         virtual task  write_burst(
            input logic [63:0]   addr,
            );
         endtask
         virtual task  set_write_data( input logic [1023:0] datain_value
            );
         endtask
endclass
```

Example 3: Interface Proxy Registration with ConfigDB

```
localbus_txn_if
    #(
      .ADDR_WIDTH(ADDR_WIDTH),
      .SDRAM_DSIZE(SDRAM_DSIZE)
       )
    lbbfm_inst (
     );


initial
begin
    // Register this Virtual Interface Wrapper in the UVM ConfigDB, with the appropriate lookup/reference
string
    uvm_config_db #(if_proxy_base)::set(null,"uvm_test_top", “localbus_txn_bfm_instance”,
lbbfm_inst.proxy_if);
end
```

Example 4: Interface Driver type $cast of Proxy Interface handle

```
class lb_driver extends uvm_driver #(uvm_sequence_item);

  lb_configuration m_config; // configuration object
  lb_if_proxy_base p_if;     // proxy interface handle

…
    // fetch driver configuration object
    $cast(p_if, m_config.get_p_if());  // set base proxy handle to point to derived proxy object

endclass
```

III. HDL Parameters Conversion to UVM Sequence Variables

With a highly parameterized Verilog testbench, test sequence code embedded within the testbench will frequently *declare* parameterized test sequence variables or assign parameterized width values. For example:

```
reg [ADDR_WIDTH-1:0]   test_address;    //declaration of parameterized-width variable, OK in older Verilog
                                         // but cannot be made in a non-parameterized UVM sequence

for (int i=0; i<256; i++)|
    test_address[ADDR_WIDTH-1:0] =  {{ADDR_WIDTH-8{1'b0}}, i[7:0] };
                                         // Number of 0 bits in assignment is parameterized
```

Example 5: Parameterized Variable Declarations and Assignments

The Northwest Logic scripting infrastructure as-is did not support generating long lists of parameter overrides to two separate toplevel modules, and would have required extensive modification to do so. Furthermore, for uniquely parameterized interface instances, such as discussed in Section II above, instance parameterization in the UVM Framework requires additional complexity to pass parameter values through levels of UVM environment objects hierarchy. Unfortunately, without the complete UVM Framework also being parameterized, these width parameter values cannot be used to declare variables within UVM sequence code. Instead, variables in UVM sequence code need to be declared according to the fixed-widths established for interface base proxy classes, for example:

```
reg [63:0]   test_address;    // in UVM test sequences, declare as non-parameterized-width test variable
```

Example 6: Conversion to Non-Parameterized Declarations

Parameter values are often also used as variables within test sequence code. For example,

```
for (int i=0; i<ADDR_WIDTH; i++)
    test_address=  1'b << i;       // loop thru all valid address bit positions
```

Example 7: Parameter Use within Test Sequence Code

When parameters are used as actual test variables, in order to easily port tests from a historical testbench, it is desirable to continue these references as test variables. Fortunately, *this* use of parameter value as a true variable *can* be provided with ease. To do this, the Northwest Logic UVM testbench declares a set of variables using identical string/variable names as the historical parameters. Since parameter values are known at simulation link/elaborate (simulation time=0), the Northwest Logic UVM testbench explicitly copies the parameter value to a like-named variable. In this way, UVM-ported test sequences may still continue to make reference to a variable within test sequence code, as if it had been the historical parameter. This greatly simplifies the conversion/porting effort of existing Verilog test sequences into UVM test sequences.

```
In "top_params_pkg.sv":
   integer ADDR_WIDTH:      // will hold the value of the ADDR_WIDTH param determined at runtime from
                            // either the parameter default value or a runtime parameter override

In the HDL_TOP testbench.sv:
   initial
      begin
      top_params_pkg::ADDR_WIDTH = ADDR_WIDTH;     // at t=0, assign variable ADDR_WIDTH to
      end                                          // value of ADDR_WIDTH parameter

In the UVM test sequence code:

   for (int i=0; i<ADDR_WIDTH; i++)           // continue to use the ADDR_WIDTH as a variable
         test_address=  1'b << i;             // loop thru all valid address bit positions
```

Example 8: Test Sequence Conversion from Parameter use to Parameter-Valued Variable use

IV. Dual HDL/HVL Top Modules and Preserving Existing Verification Script Infrastructure

The UVM testbench user community is not fully consistent in practice of whether UVM testbenches are constructed as a dual-top simulation, versus creating a new toplevel wrapper module containing both the HDL_TOP and HVL_TOP (or UVM test top) as sub-modules. Leading practices and recommendations are to construct and simulate with a dual-top structure. Nonetheless, when converting from an older testbench to a UVM testbench, existing scripts and verification infrastructure should be considered as factors in making the decision between a dual-top versus single-toplevel testbench architecture.

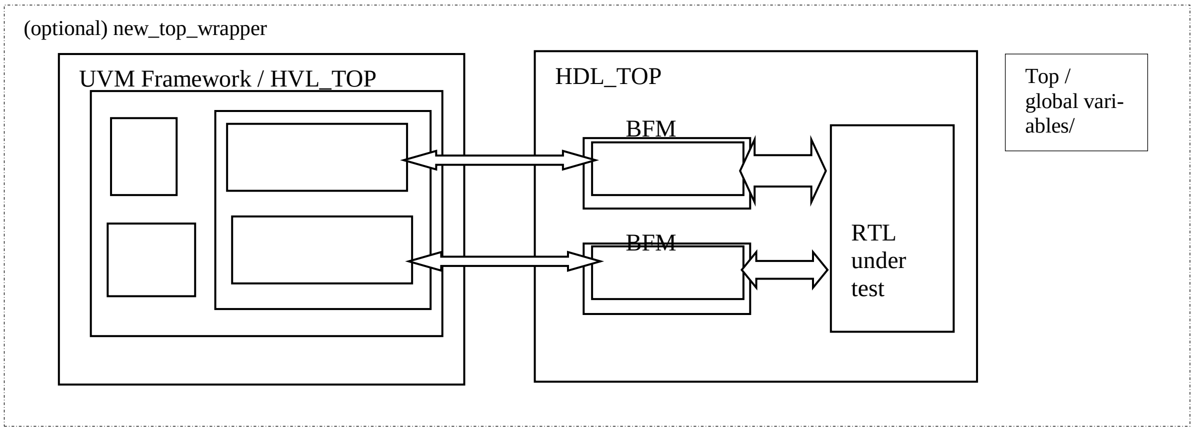


Figure 2: Dual HVL/HDL TOP

If the DUT and HDL_TOP are parameterized, one should assess the existing verification script infrastructure for setting/overriding simulation-time parameter values. For most simulators, parameters may be overridden on the simulator command-line interface. While the command-line argument to do so may vary between simulators, the parameter override may specify a hierarchical path into the simulation model. Constructing a UVM testbench using a new single top-level wrapper may require modifying simulation scripts and wrappers to insert additional path information into parameter overrides.

By comparison, utilizing a dual-top UVM simulation architecture may allow existing simulation script infrastructures to be used without change. For example,

```
(NCSIM simulator invocation argument):
        -DEFPARAM /hdl_top/ADDR_WIDTH=<override value>              arguments may require change to
        -DEFPARAM /new_top_wrapper/hdl_top/ADDR_WIDTH=<override_value>

(Questasim simulator invocation argument):
        -g/hdl_top/ADDR_WIDTH=<override value>                      arguments may require change to
        -g/new_top_wrapper/hdl_top/ADDR_WIDTH=<override_value>
```

Example 9: Parameter-Override simulator arguments

One consideration favoring implementation of a single top-level testbench wrapper might be variables' scope and possible need for global parameterization. In the Northwest Logic testbench, testbench 'global' variables are implemented and reside with a "top_pkg.sv" package which is compiled and lives alongside the dual-top simulation models. By virtue of the top_pkg existing as a (third) toplevel entity, both the HVL_TOP and HDL_TOP may import this package reference and/or specific variables from this package.

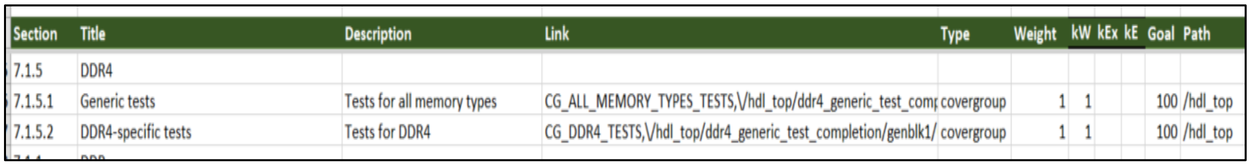
Global parameters, on the other hand, cannot be successfully declared or used from within this package. While a global parameter may be declare-able within the "top_pkg", a parameter cannot be overridden at simulation time without a path to the specific object parameterization being overridden. For a global parameter to be override-able, it must exist within the scope of a module. For example, if a new global parameter GLOBAL_PARAM in "top_pkg" is imported into a new "new_top_wrapper", then it is possible supply a runtime parameter override, as shown below in Example 10:

```
In “top_pkg.sv”
       parameter GLOBAL_PARAM = <default parameter value>;

In “new_top_wrapper.sv”:
       import top_pkg::*;

(NCSIM simulator invocation argument):
             -DEFPARAM /new_top_wrapper/GLOBAL_PARAM=<override_value>
(Questasim simulator invocation argument):
          -g/new_top_wrapper/GLOBAL_PARAM=<override_value>
```
Example 10: Runtime Override of a Global Parameter

Another consideration impacting the dual-top architecture is scripting infrastructure surrounding functional coverage declarations and metrics collection. Northwest Logic implements much of its functional-coverage objects within the testbench HDL_TOP, using “bind” statements to bind coverage to instantiations of RTL sub-modules. While companies may elect to aggregate functional coverage using alternative tools or methods, underlying aggregation of coverage from Unified Coverage DataBase (UCDB) files requires hierarchical path references to coverage data objects. For example, Northwest Logic utilizes the Questa Verification Manager (QVM) toolset for testplans, aggregating coverage, and producing coverage reports. An example of such testplans containing hierarchical path references is shown below; one can immediately see how an extra hierarchy in the path to coverage groups would have introduced significant change ripple-impact to multiple testplan documents.



Section	Title	Description	Link	Type	Weight	kW	kEx	kE	Goal	Path
7.1.5	DDR4									
7.1.5.1	Generic tests	Tests for all memory types	CG_ALL_MEMORY_TYPES_TESTS,\/hdl_top/ddr4_generic_test_comp	covergroup	1	1			100	/hdl_top
7.1.5.2	DDR4-specific tests	Tests for DDR4	CG_DDR4_TESTS,\/hdl_top/ddr4_generic_test_completion/genblk1/	covergroup	1	1			100	/hdl_top

Example 11: Testplan Collateral / Infrastructure with Testbench Hierarchical Path

After consideration of the factors, to preserve existing simulation script and functional coverage / testplan infrastructure, Northwest Logic opted to follow the UVM-recommended dual-top simulation approach. However, it is important to note that this did come at a cost in complexity and alternate solutions for porting the existing global parameterization from the older pre-UVM testbench.

V. Initialization via Legacy HDL Task Infrastructure

Most any HDL testbench will execute some sort of initialization sequence starting from simulation time=0. In some cases, this initialization could be as simple as a reset-signal de-assertion, may further include control-register initialization, or might even include some other real functional operation (in the case of the Northwest Logic Memory Controller IP, link training between PHY and memory models). Regardless of how simple or complex the initialization sequence, every conversion from a simple testbench to a UVM testbench will likely face the question, "how much of the HDL initialization sequence should be performed by a UVM sequence, versus executed within the scope of the HDL_TOP"? In theory, UVM sequences could provide all stimulus to the DUT starting at t=0. It is often likely that at least *some* initial clocking and reset time may need to occur within the HDL_TOP before any UVM sequences are created and executed.

The Northwest Logic Verilog testbench included a large number of initialization tasks located at the toplevel of the original testbench hierarchy. This is likely to be the case for many other companies' existing Verilog testbenches. As is common with such testbenches, many of these tasks which lived "flattened" at the toplevel of the Northwest Logic testbench hierarchy made calls/references to other such tasks by nature of all being located at the same module.

Northwest Logic Memory Controller IP testbenches include a variety of additional third-party PHY and DDR memory models. Most of these third-party models include uniquely custom libraries of Verilog tasks which execute initialization steps for the respect models. However, many of these third-party models pre-date UVM methods, and/or for whatever reason do not include Verilog interface module(s) for UVM sequences to be able to invoke these initialization tasks. This is not a problem unique to Northwest Logic -- it is highly likely that an existing pre-UVM testbench contains some in-house-developed or third-party models which do not supply UVM interface methods. To complicate matters further, the Northwest Logic used file `include mechanisms to include initialization task libraries for these third-party models, referencing different library versions via `include paths.

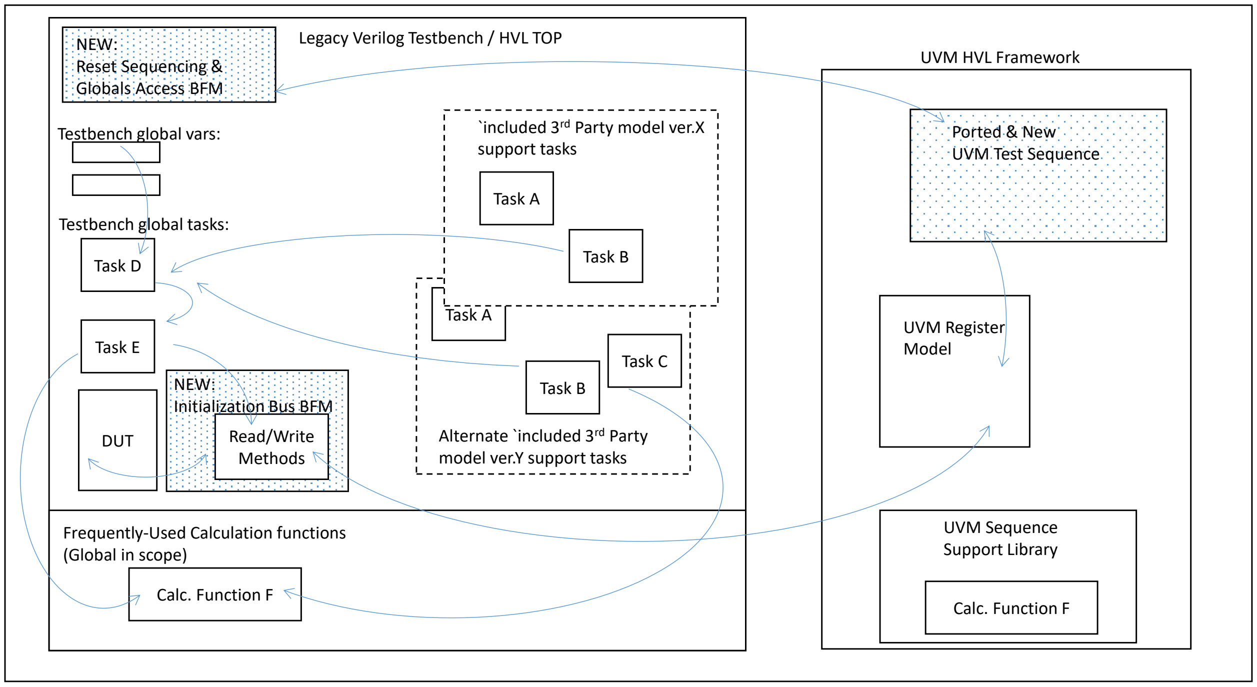


Figure 3: Initialization Tasks Hierarchy

For UVM initialization sequences, interface methods to at least assert/deassert reset signals are often needed. For these global initialization signals and enabling of UVM test main phase, a system global reset interface was constructed, with access methods to set resets and query reset-completion handshake signals.

Rather than restructure the entire testbench initialization, however, Northwest Logic made the decision to execute the base initialization sequence directly from within the HDL_TOP before initiating any UVM stimulation sequences. While there may have been some merit to re-locating many of these initialization tasks into interface modules, so that these tasks could potentially be called again later from a UVM sequence, the effort the resolve all the cross-interface task references was deemed an unnecessary effort compared to the priority focus on completing initialization and handing over control to new UVM-based test sequences. Anyways, any UVM register-block operations to place new configuration-register content into the DUT could not have been attempted prior to deassertion of the DUT reset (this would have been the case regardless of whether configuration-register reprograming was executed using simulation-time-advancing transactions or register backdoor methods).

If initialization tasks are desired to be re-invoked later in the simulation, there may be some merit at a future time in investing the time to develop interface module(s) so that UVM sequences can re-execute initialization steps. This might be a complicated endeavor depending upon how complex the set of initialization tasks are, however, whether they architecturally should belong in multiple different interface modules, and how complex their cross-interface task references may be.

In addition to initialization tasks, Northwest Logic opted to leave an existing set of historical support functions intact in a library instantiated within the HDL_TOP. Most of these were simply calculation functions, but dependent on the HDL_TOP parameterized values. Still, a small handful of these library functions are *also* used in UVM test sequences' execution post-initialization. Rather than code another interface module specifically for simple calculation functions, Northwest Logic opted to create a duplicate library of these calculation functions to be called by UVM test sequences. While this is sub-optimal from code-maintenance standpoint to have identical tasks/functions in two separate libraries, the number of duplications was considered small and manageable compared to carrying and referencing an additional interface mechanism just for calculation functions.

VI. Replicating Global Testbench Elements into Scalable BFMs for Rapid Test Porting

This paper has already discussed in section (III) how defining global variables with identical names as parameters can be utilized as a technique to simplify effort associated with porting a historical Verilog test/task sequence to a UVM sequence. In this section (VI), a few additional suggestions are made to further ease the effort of porting Verilog test/task sequences to UVM sequences. These recommendations may seem obvious or apparent, but the authors are of the opinion that they are worth explicitly mentioning.

Firstly, many historical pre-UVM testbenches make use of individual wire/register variables which are declared at the toplevel of the Verilog testbench (globals) and then assigned or queried by a large number of testbench tasks, test sequences, or functions. While some testbenches may implement their Bus Functional Models (BFMs) as discrete, instantiated sub-modules already, many historical testbenches implement BFM-like functions which simply set / sequence values assigned to global register variables. When converting a large task-based testbench to a UVM-based testbench, it is natural that stimulus functions all reside within some sort of interface BFM, so that instantiations may be easily scaled according to the number of interface instances in the DUT.

The authors recommend assessing the existing historical testbench for global variables which need to be scaled / replicated per BFM. For true global control bits, these may best be relocated to a single central interface module, such as a system reset BFM. For variables which control operation of multiply-instantiated BFMs, however, it is recommended that each BFM define its own internal version of the (previously-global) variable, and export a per-BFM method such as "bfm_instance1_if.set_xxxx()" method. With these methods available, one may find the porting effort between a historical test sequence task and a new UVM test sequence to be significantly easier.

For example, Example 12 below shows a comparison of an older (legacy) Verilog testbench sequence code, ported to new UVM sequence:

Historical Verilog test sequence:

```
        reg global_control_bit_A;
          task test_sequence1();
                global_control_bit_A = 1'b1;
                repeat (10) @ posedge (clk);
                transactional_bfm_controlbits_B[WIDTH:0] = 'value;
                global_control_bit_A = 1'b0;
          endtask
```

Equivalent UVM testbench sequence:

```
          class test_sequence1 extends seq_base;
            task body();
                global_signals_bfm.set_global_control_bitA(1'b1);
                repeat(10) clkreset_bfm.wait_posedge_clk();
                    // or, better-yet, clkreset_bfm.wait_posedge_clk(10)
                transactional_bfm_proxy_if.set_controlbitsB(value);
                global_signals_bfm.set_global_control_bitA(1'b0);
            endtask;
           endclass
```

Example 12: Legacy Testbench Variables' Replication in UVM Testbench for Ease of Sequence Porting

For some projects, setting control bits by assigning interface variables directly could be somewhat more direct and less verbose, however, due to Northwest Logic's heavy use of parameterization & parameterized-widths on many of these variables, the use of proxy interface methods to set/get values was necessary for most BFMs. For consistency to the UVM test writer, the use of set/get methods was used for all such interface controls, rather than utilizing different access methods for parameterized control-bits and non-parameterized control bits.

Similarly, time, effort, and errors can be significantly reduced by consciously re-implementing historical BFM-like tasks as interface methods. Intentionally re-implementing task calls within a BFM as identically as possible can enable rapid porting of historical test sequences to new UVM-based test sequences. For example, Example 13 below illustrates how choices in implementing legacy-like task methods in UVM BFM can ease test porting efforts.

```
Historical Verilog test sequence:
        reg write_datavalues_array[]
         task test_sequence1();
              for (int i=0; i<N; i++) begin
                   writetxn_datavalues_array[i] = <value>;
                   read_expected_value_array[i] = <value2>;
                   end
              perform_write_txn (arg1, arg2, arg3…)
              perform_read_txn  (arg4, arg5, arg6 …)
         endtask



Equivalent UVM test sequence:
         class test_sequence1 extends seq_base;
           task body();
                for (int i=0; i<N; i++) begin
                     transaction_bfm_instance1_proxy_if.set_writetxn_datavalues_array(i, value);
                     transaction_bfm_instance1_proxy_if.set_readtxn_expected_value_array(i, value2);
                      end
                transaction_bfm_instance1_proxy_if.perform_write_txn(arg1, arg2, arg3 …)
                transaction_bfm_instance1_proxy_if.perform_read_txn  (arg4, arg5, arg6 …)
           endtask;
          endclass
```

Example 13: Legacy Testbench BFM Tasks Replication into Scalable BFM interface methods

VII. GUIDANCE ON CONVERTING LEGACY $DISPLAY USE TO UVM INFO/ERROR MESSAGING

When replacing historical Verilog testbenches with a new UVM testbench, it is expected that historical use of Verilog $display() functions will be replaced with `uvm_info() macros. A mix of new 'uvm_info() messaging combined with historical $display() functions is highly discouraged by most UVM community members.

UVM libraries and macros for informational messaging define some standards for message verbosity levels, such as UVM_LOW, UVM_MEDIUM, UVM_HIGH, and UVM_DEBUG. The UVM library is highly customizable to change the relative verbosity level associated with these verbosity level strings or to insert new custom verbosity levels. Nonetheless, most UVM practitioners begin utilizing these UVM info macros without pre-considering any specific methodology or guidance on when to utilize each specific verbosity level.

Most engineers involved in porting an existing testbench to UVM can readily identify existing $display() statements that should map to the UVM 'uvm_error() or `uvm_fatal() macros. Most such existing $display statements already include some key-word strings such as "Error:…" , "Warning:….", or "Fatal error:.." with this existing $display statements. However, mapping existing non-error/non-fatal $display() statements to 'uvm_info() macros requires some up-front methodology planning. For example, some testbenches may already implement some conditionally-enabled extra debug verbosity:

```
If (debug_msgs_en == '1)
   $display("Info:  reached execution checkpoint 'M'  in test sequence"
```

Example 14: Typical Debug-Conditional $display use

A novice UVM testbench designer might think that this statement would most naturally be converted to a 'uvm_info() macro with UVM_DEBUG level of verbosity. However, UVM_DEBUG verbosity is utilized within the UVM library itself, and (if selected), in fact enables significant verbosity. A validation engineer enabling UVM_DEBUG verbosity will most assuredly be overwhelmed with far more debug output than he/she prefers.

Northwest Logic has adopted a specific convention for use of various 'uvm_info messaging. Each individual corporation or project may elect to adopt a suitable unique convention. It is recommended that a project define a convention up-front, in order to avoid excessive revision of test sequences' source later, especially once UVM messaging starts to appear in long multi-sequence regression logs.

UVM message macro	UVM message verbosity	UVM test sequence use	UVM testbench/ framework use
`uvm_fatal		Situations where simulator crash, segfault, or unexpected behavior is likely to ensue	
`uvm_error		Test errors (such as data miscompare) or other results miscompare vs expected are detected, but test execution can proceed to potentially test other conditions	
`uvm_info	UVM_DEBUG	Do not use	Do not use. Used within UVM macro libraries, UVM_DEBUG verbosity for debug into/including UVM macros
	UVM_HIGH	Do not use	Use for debug statements from UVM Framework elements, such as drivers, sequencers, testbench build
	UVM_MEDIUM	Use for debug of test sequence execution progress (maps to historical (if debug_en) $display statements)	
	UVM_LOW	Use for standard test sequence progress execution messages, similar to what historical $display statements would output to standard regression-test stdout/logfiles	

Table 1: Northwest Logic Conventions on UVM Message Verbosity

VIII. Customizing UVM Report Server

Use of the UVM fatal/error/info macros for display of all messages is strongly urged. However, many users of the UVM info macros may likely be disappointed/unhappy with the excess verbosity (wordiness) of individual messages. Altering the UVM_VERBOSITY level of the message does not change the length of the message string itself, only the conditions under which the UVM report server passes or filters the message from actually appearing.

Users of the UVM info messaging will quickly find that an individual message string (when not filtered by the UVM_VERBOSITY level) is output by the UVM report server as the concatenation of multiple sub-strings. These substrings include the severity_type name (eg "UVM_INFO:", "UVM_ERROR:", "UVM_FATAL:"), a hierarchical code location string, similar to the "%m" formatted path string, additional instance-specific information (again, similar to the %m formatted path string), an "ID" string (user-provided as an argument to the 'uvm_info macro), and finally the intended actual informational message. Needless to say, a UVM testbench user will soon find the standard 'uvm_info message output to be excessively wordy, and to contain repetitive, redundant path identifier strings. If simulation standard-output is typically fed to simulation post-processing script infrastructures, most of these extraneous characters will likely make their way through the post-processing filters and cause many engineers to gag on long wrapped-lines of repetitive text.

Instead, Northwest Logic elected to override/replace the standard UVM report server function with a customized report server, which significantly reduced the output verbosity. While a callback 'report catcher' can be added to the standard report server, such use is best tailored for filtering certain reports or modifying report parameters such as verbosity; Northwest Logic desired to wholesale alter the entire standard report formattings, rather than intercepting and modifying only certain reports, therefore the compose_message() function of the report serve was deemed the best location for this. Companies/projects may elect to tailor their replacement report-server to suit their specific desires. Northwest Logic's customization is provided in this paper as an example reference for others wishing to explore implementing their own customizations. To do so, Northwest Logic inserts the following code into the test_base class definition:

```
Within "test_base.svh":
class my_report_server extends uvm_report_server;
  virtual function string compose_message( uvm_severity severity,
                                   string name,
                                   string id,
                                   string message,
                                   string filename,
                                   int line );
  uvm_severity_type severity_type = uvm_severity_type'( severity );

 // omit additional of explicit timestamp for simple UVM_INFO messages, insert explicit timestamp for
 // WARNING/ERROR/FATAL messages since engineer is likely to want to investigate/debug
  if (severity <= UVM_WARNING)
     return $sformatf( "%-11s [%-40s] %s",        severity_type.name(), id, message );
  else
     return $sformatf( "%-11s [%-40s][@%12t] %s", severity_type.name(), id, $time, message );
  endfunction: compose_message
endclass: my_report_server

class test_base extends uvm_test;
  function void end_of_elaboration_phase(uvm_phase phase);
     my_report_server my_server;

    // Now that all testbench components have been built, change default verbosity level to UVM_LOW unless
specified
    uvm_top.set_report_verbosity_level_hier(UVM_LOW);

    // Replace the UVM_INFO report server with our own that is less wordy
    my_server = new;
    uvm_report_server::set_server(my_server);
  endfunction

endclass
```

Example 15: UVM INFO Messaging Customization via Custom Report Server

IX. COMMON REPORT TAG STRING VARIABLE IN SEQUENCE BASE CLASS

When using the UVM built-in macros for info/error/display, testbench and test sequence developers frequently utilize a simple fixed string as the message "id" string argument which is used by the UVM `info macro to formulate the resulting full info message displayed. Examples of common "id" string use are as follows:

```
`uvm_error("PROTOCOL CHECKER", $sformatf("detected protocol violation XYZ at time: %0t"))
`uvm_info("SCOREBOARD", $sformatf("checking transaction %0d for possible miscompare", j), UVM_MEDIUM)
`uvm_info("JTAG BFM", $sformatf("executing transaction %0d at time %0t", i, $time), UVM_LOW)
`uvm_info("TEST SEQ 3", $sformatf("starting sequence on sequencer %0d at time %0t", i, %time), UVM_LOW)
```

Example 16: Common use of UVM INFO report ID strings

However, especially when scaling a UVM testbench to include multiple instances of a specific BFM, or running multiple instances of test sequence(s), it is common to see output containing multiple output messages that all cite the same message "id" string. Northwest Logic elected to define a string variable as part of the sequence base class object, and have the sequence base class object automatically populate that string with a (brief) but unique-instance-identifying string, customized and significantly shorter than a %m hierarchical instance-path.

```
In file "seq_base.svh":

class seq_base extends uvm_sequence;
…
  string rpt_idstr;        // ID string to use for UVM_INFO reporter
  task pre_start();  // Configure base sequence
    …
     rpt_idstr = $sformatf("ch%0d ps%0d port%0d: %s", chan_number, ps_number, port_number,
this.get_type_name);
    …
  endtask
endclass

With a specific test sequence:
    `uvm_info(rpt_idstr, $sformatf("%0s, %t", testname, $time), UVM_LOW)
```

Example 17: Report Tag String defined In Sequence Base Class

By adopting a convention that all test sequences shall utilize this report ID string as the ID string argument to `uvm_info macros, Northwest Logic is able to produce concise, uniquely-identified logfile output, even when multiple instances of a specific sequence are spawned and executed in parallel. An example of resulting log output for multiple test sequences spawned in parallel on a multi-channel memory controller IP is shown below in Example 18, below:

```
# UVM_INFO   [use_test_select  ] ++++ Spawning sequences of selected test(s) in parallel on multiple control-
ler channels
# UVM_INFO   [ch7 address_test5_seq              ] ADDRESS TEST (#5),           6642500
# UVM_INFO   [ch6 address_test5_seq              ] ADDRESS TEST (#5),           6642500
# UVM_INFO   [ch5 address_test5_seq              ] ADDRESS TEST (#5),           6642500
# UVM_INFO   [ch4 address_test5_seq              ] ADDRESS TEST (#5),           6642500
# UVM_INFO   [ch3 address_test5_seq              ] ADDRESS TEST (#5),           6642500
# UVM_INFO   [ch2 address_test5_seq              ] ADDRESS TEST (#5),           6642500
# UVM_INFO   [ch1 address_test5_seq              ] ADDRESS TEST (#5),           6642500
# UVM_INFO   [ch0 address_test5_seq              ] ADDRESS TEST (#5),           6642500
# UVM_INFO   [ch0 long_writeread_test11_seq         ] LONG_WRITE_READ_NO_AP (#11),          24287500
# UVM_INFO   [ch1 long_writeread_test11_seq         ] LONG_WRITE_READ_NO_AP (#11),          24287500
# UVM_INFO   [ch2 long_writeread_test11_seq         ] LONG_WRITE_READ_NO_AP (#11),          24287500
# UVM_INFO   [ch3 long_writeread_test11_seq         ] LONG_WRITE_READ_NO_AP (#11),          24287500
# UVM_INFO   [ch4 long_writeread_test11_seq         ] LONG_WRITE_READ_NO_AP (#11),          24287500
# UVM_INFO   [ch5 long_writeread_test11_seq         ] LONG_WRITE_READ_NO_AP (#11),          24287500
….
```

Example 18: Sample Output using customized report server and common sequence Report ID String

X. Conclusions

When converting an existing older Verilog-based testbench to UVM, a number of decisions may be made which can significantly impact the ease of carrying the entire surrounding testbench infrastructure, scripts, and tests forward. This paper presents a number of topics which presented some difficulties and solutions during the Northwest Logic testbench conversion to UVM. The paper highlights some key decisions which were extremely valuable in reducing effort and schedule at Northwest Logic, and provides some recommendations and guidance for others who may be setting out to evolve existing Verilog-based testbenches to the UVM methodology & structure.

References:

[1] D. Rich, J. Bromley. Abstract BFMs outshine Virtual Interfaces for Advanced SystemVerilog Testbenches. DVCon February 2008

[2] Michael Baird, Coverage Driven Verification of an Unmodified DUT within an OVM Testbench. DVCon February 2010