Common Challenges and Solutions to Integrating a UVM Testbench

in place of a legacy monolithic Verilog Testbench

Frank Verhoorn – Northwest Logic, Inc

Mike Baird – Willamette HDL

Upgrading a legacy Verilog Testbench to UVM

- BACKGROUND: Many companies have existing Verilog-based testbench structures which have evolved over years, often containing elements such as:
  - Test sequences in initial() blocks or inline / included as library of task calls
  - Libraries of support functions & task calls, assuming existence of global variables
  - Reliance on either global scoping for variable/task references, or explicit hierarchical path references – I/we call this "monolithic" (not easily separable)

- Northwest Logic reasons for upgrading to a modern Object-Oriented testbench structure (such as UVM) included:
  - Object-oriented scalability (multiple ports/drivers/BFMs)
  - Ability to use Third-party Verification IP
  - More complex randomization, scoreboarding
- It's likely that other companies are looking to perform similar upgrades/migrations.

Common Challenges & Issues

- This presentation discusses some issues we encountered (likely to be common challenges for others as well). We will present some suggested solutions and recommendations based on our work.

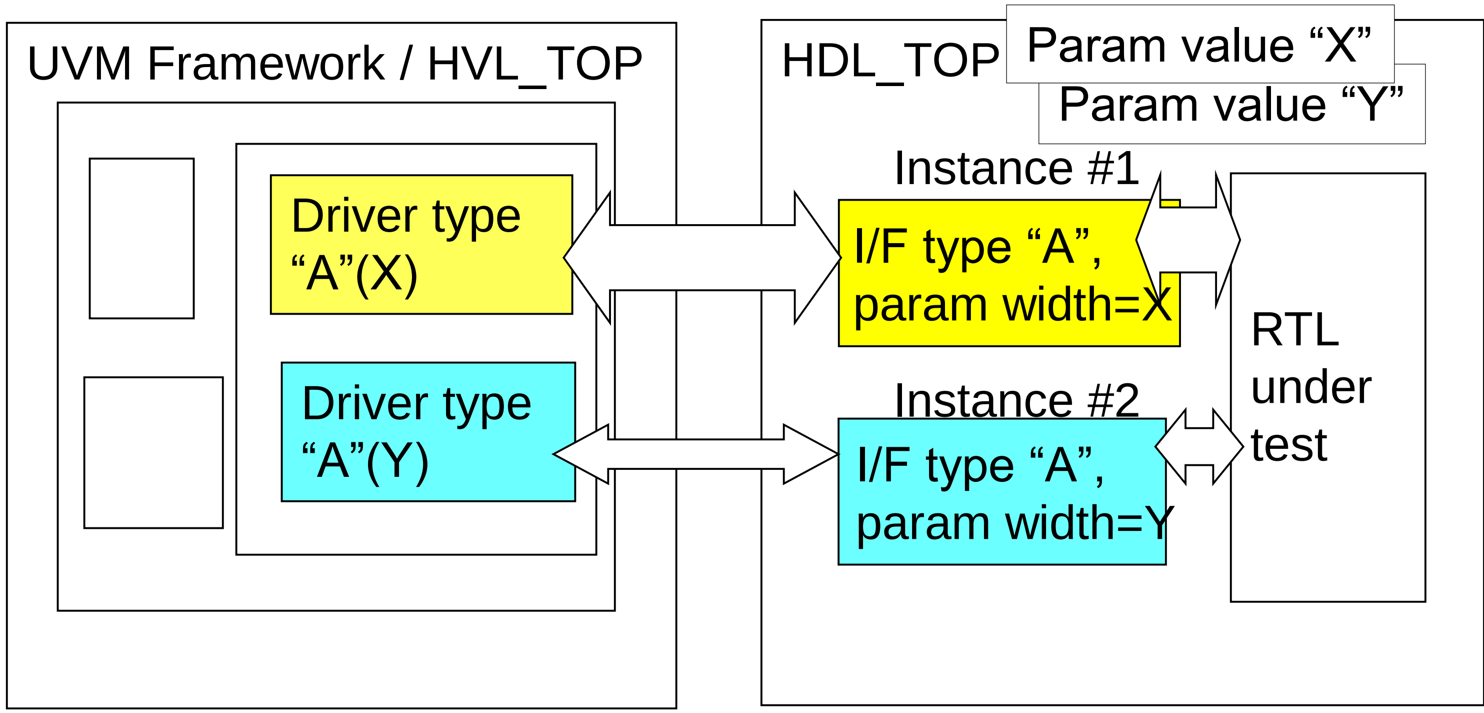
1. Issues and Solutions arising from Parameterized DUTs and Testbenches
2. Some Guidance for re-using support infrastructures (simulation scripts, etc)
3. Recommendations for migrating legacy global tasks/functions into BFM methods
4. Solutions and Suggestions for migrating legacy $display to UVM

Heavily Parameterized DUTs

- Parameters are great for implementing configurable library elements (such as a FIFO) with parameterized widths/depth/etc
- Northwest Logic’s existing IP design also heavily relies on parameters (rather then ‘define) for DUT interfaces (such as widths)

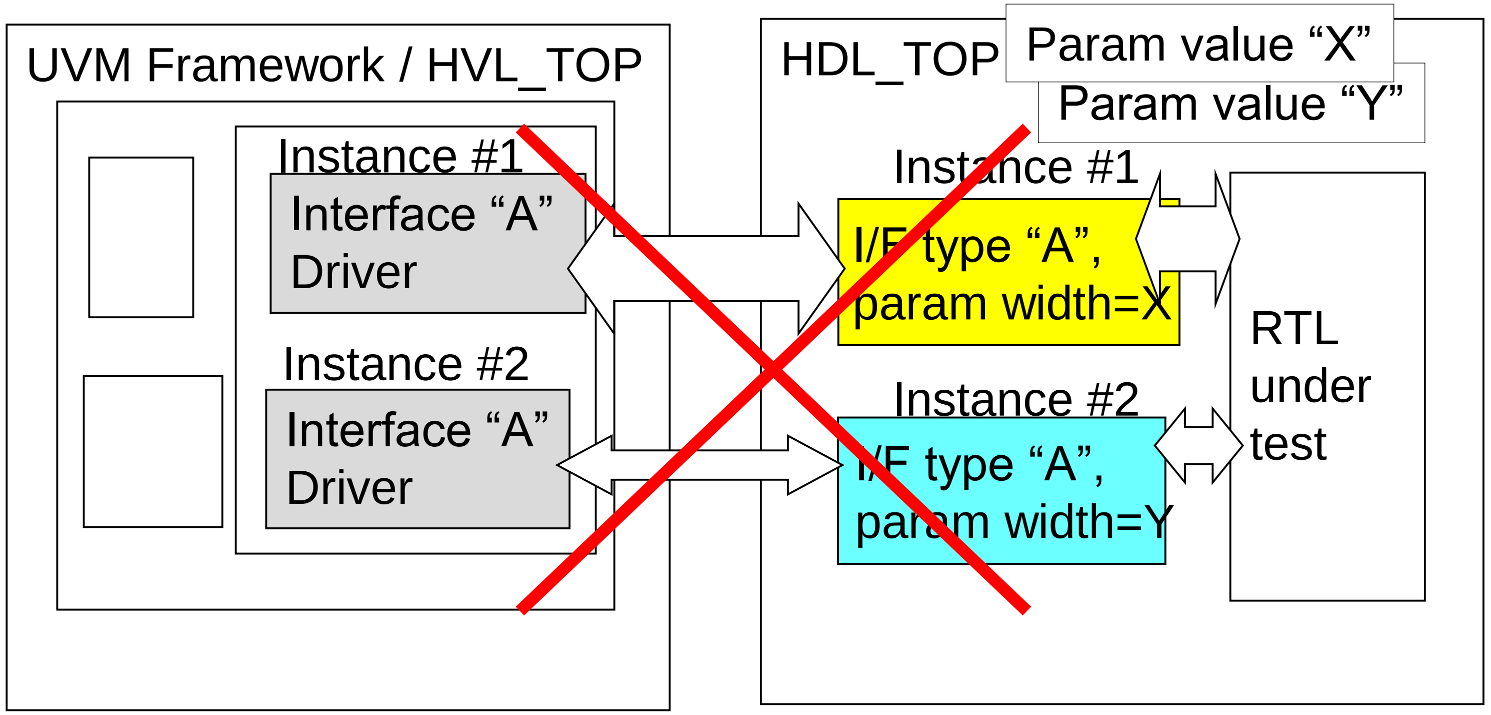
- Parameterization of DUT interfaces can present difficulties for scalable instantiations

- Unique set of parameters for interfaces makes each interface a unique type that must be matched in the UVM testbench drivers and monitors



Cannot use Common UVM Driver for unique interface instances

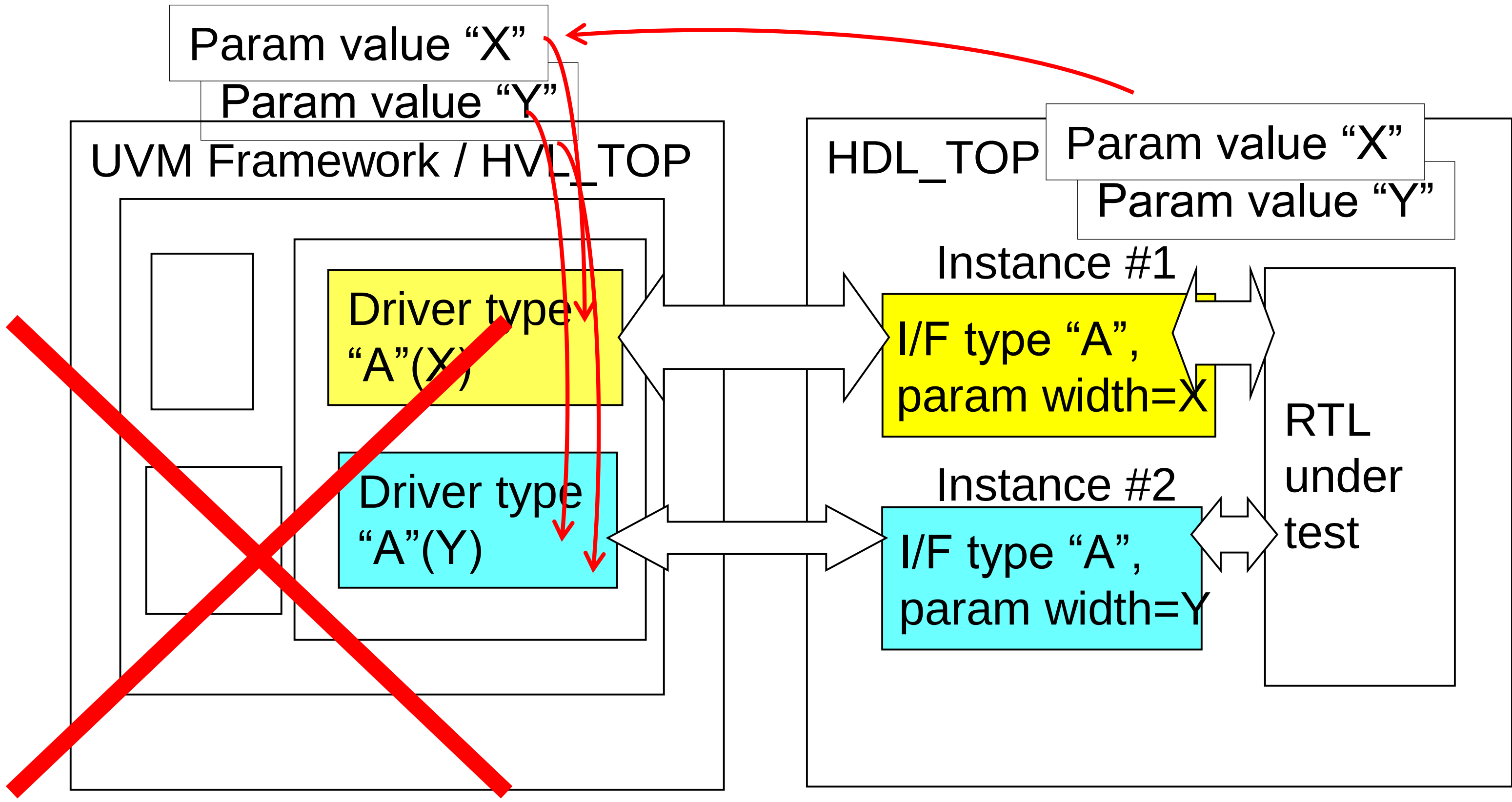
- Interface drivers cannot be instances of the same type, since the interfaces are parameterized with different values



Matching Parameterization in UVM Framework

- With hundred of parameters used, we could have collected them all into a single common parameters package

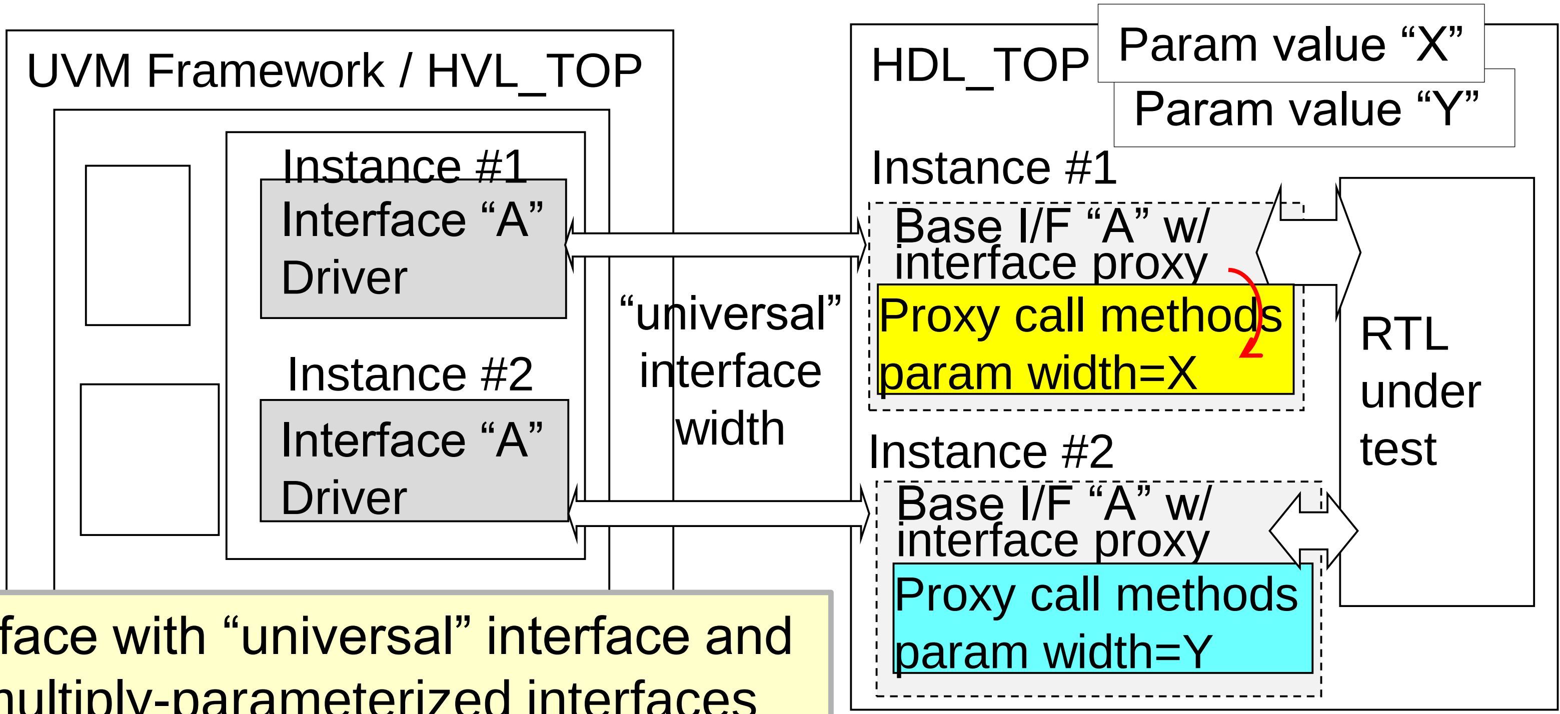
- However, distributing this many parameter values throughout the framework hierarchy was too cumbersome and unmaintainable



Use of Single base interface class with proxy interface

- Single base interface with a “universal” interface is presented as a “proxy” for the underlying (parameterized-width) method

- While this requires an additional step to establish the base interface & proxy methods, scalability and support is easy



RECOMMENDATION: use base interface with “universal” interface and proxy interface methods to handle multiply-parameterized interfaces

Another Issue w/ Parameters: use in test var. declarations and assignments

- Testbenches built around significant parameter use often rely upon parameter values in variable declarations and assignments

```
parameter ADDR_WIDTH;
reg [ADDR_WIDTH-1:0]test_address;
for (int i=0; i<256; i++)
     test_address[ADDR_WIDTH-1:0] =  {{ADDR_WIDTH-8{1'b0}}, i[7:0] };
```

- To avoid parameterizing the entire UVM testbench hierarchy, these values are made available as global vars in a "params" pkg

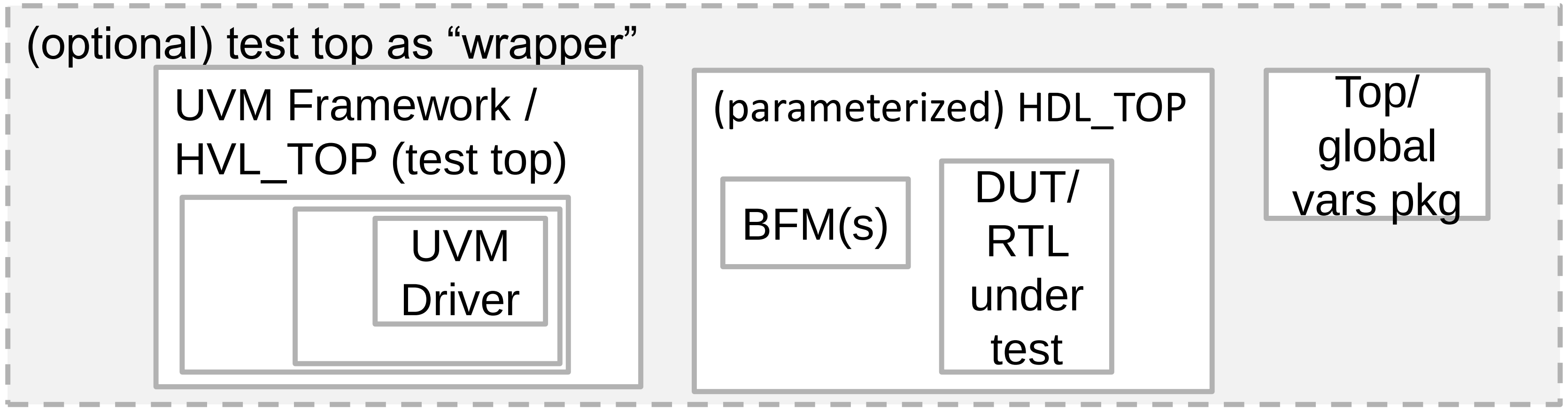
```
In "top_params_pkg.sv":     integer ADDR_WIDTH:

In the HDL_TOP testbench.sv, assign:
     initial
     // at t=0, assign var to parameter's value
     top_params_pkg::ADDR_WIDTH = ADDR_WIDTH;

Thence, used in the UVM test sequence code:
     for (int i=0; i<ADDR_WIDTH; i++) // use as var
             test_address=  1'b << i;
```

Dual/Single HDL TOP and Preserving Existing Verification Infrastr. / Scripts

- Practices differ in the UVM community re: use of a single toplevel wrapper for the entire UVM simulation
  - versus having “dual-top modules” (HVL_TOP and HDL_TOP)



For Northwest Logic, the advantages of a dual-top allowed for preservation of infrastructure around:

- Simulation scripts’ setting/overriding parameter values
- Test plan linkage to coverage via hierarchical paths

Simulation Script Infrastructure for setting Parameter Values

- Setting param values in a single top-level wrapper testbench:

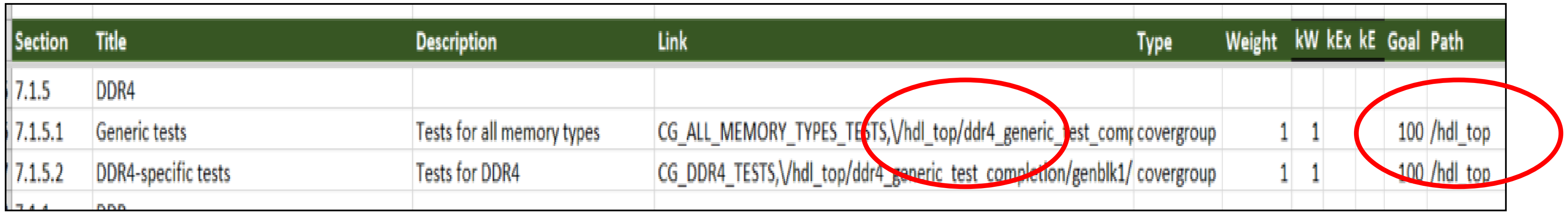
```
top_pkg.sv:            parameter GLOBAL_PARAM = <default parameter value>;
new_top_wrapper.sv: import top_pkg::*;
(Incisive):     -DEFPARAM /new_top_wrapper/GLOBAL_PARAM=<override_value>
                -DEFPARAM /new_top_wrapper/hdl_top/GLOBAL_PARAM=<override_value>
(Questasim):    -g/new_top_wrapper/GLOBAL_PARAM=<override_value>
                -g/new_top_wrapper/hdl_top/GLOBAL_PARAM=<override_value>
```

- Versus script infrastructure used without changing existing parameter hierarchy:

```
// no change to existing script infrastructure
(Incisive):     -DEFPARAM /hdl_top/ADDR_WIDTH=<override value>
(Questasim):    -g/hdl_top/ADDR_WIDTH=<override value>
```

Testbench Hierarchy & Existing Coverage-Based Testplan Links

- Northwest Logic uses the Mentor QVM toolset for testplans and aggregating / analyzing coverage
- Path to coverage items is embedded in the testplans, linking a testplan item to specific covergroup/coverpoints/directives/assertions/etc



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

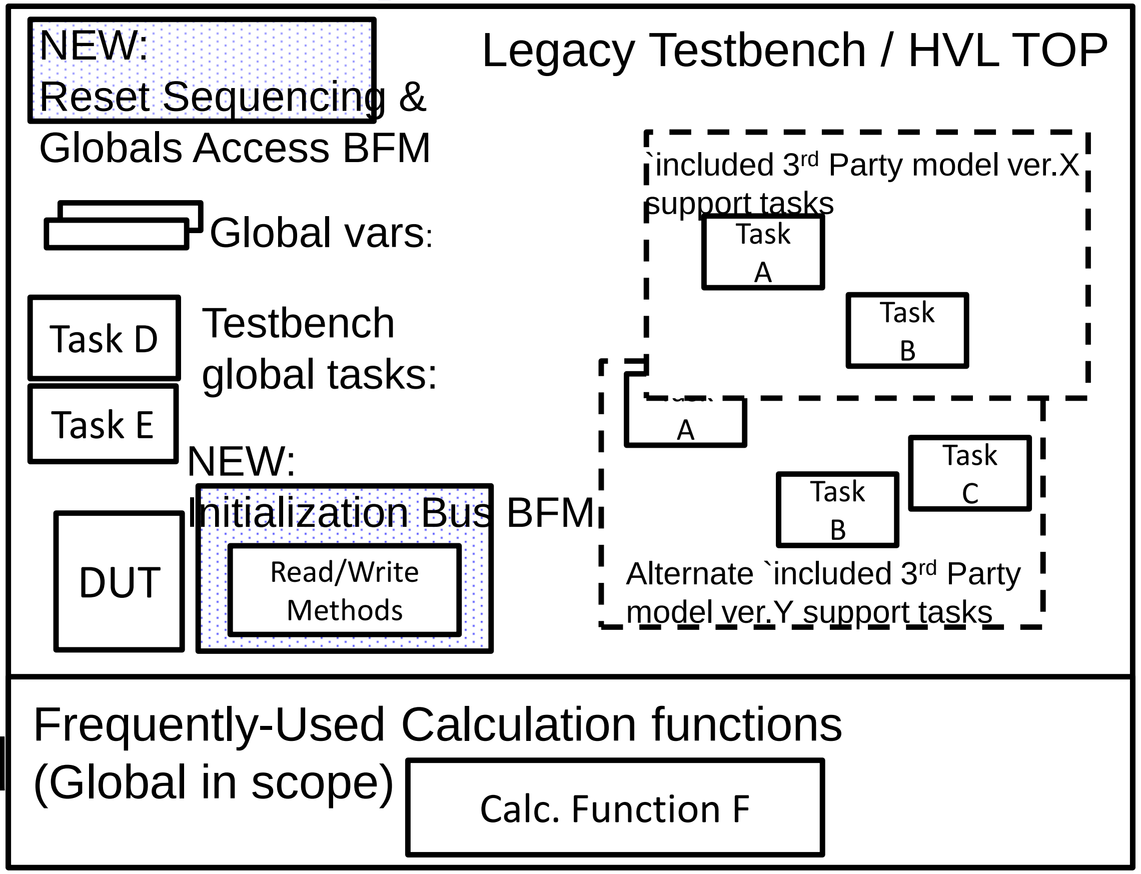
- Maintained existing hierarchy paths to coverage and parameters by adding the UVM Framework as a dual-top -- enabled us to preserve our testplans and analysis script infrastructure

RECOMMENDATION: consider impact / changes to your existing infrastructure (for example, related to Parameterization and Testplans) when determining your UVM testbench hierarchy choice

Un-Flattening a Legacy Testbench from Global variables & Tasks/Functions

- Northwest Logic's legacy testbench had evolved into a large, fairly flat testbench containing many global variables, tasks and functions

- Some globals such as clks and resets were simple to relocate to BFMs with access and control method
- Some tasks were clearly associated with a specific BFM, such as an I2C or APB register access task



Legacy Testbench / HVL TOP
NEW: Reset Sequencing & Globals Access BFM
Global vars:
Task D
Task E
Testbench global tasks:
`included 3rd Party model ver.X support tasks
Task A
Task B
Task A
Task B
Task C
Alternate `included 3rd Party model ver.Y support tasks
NEW: Initialization Bus BFM
Read/Write Methods
DUT
Frequently-Used Calculation functions (Global in scope)
Calc. Function F

Simplifying Test Sequence Porting to UVM Testbench

- For other (formerly global) testbench variables relocated into BFM's for scalability:
  - Choosing names of the BFM method(s) made porting of test sequence code to distinct UVM test sequences a manageable effort
  - For most, interface proxy methods were needed (vs simple interface BFM signal references) due to existence of multiple parameterized BFM instances,

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

```
class test_sequence1 extends seq_base;
task body();
    global_sigs_bfm.set_global_control_bitA(1'b1);
    clkreset_bfm.wait_posedge_clk(10)
    bfm_proxy_if.set_controlbitsB(value);
    global_sigs_bfm.set_global_control_bitA(1'b0);
endtask;
endclass
```

DUT Initialization via UVM sequence versus testbench HDL_TOP

- Every conversion to a UVM testbench begs the question:

  "how much of the HDL initialization sequence should be performed by UVM sequence, versus executed within the scope of the RTL/HDL_TOP"

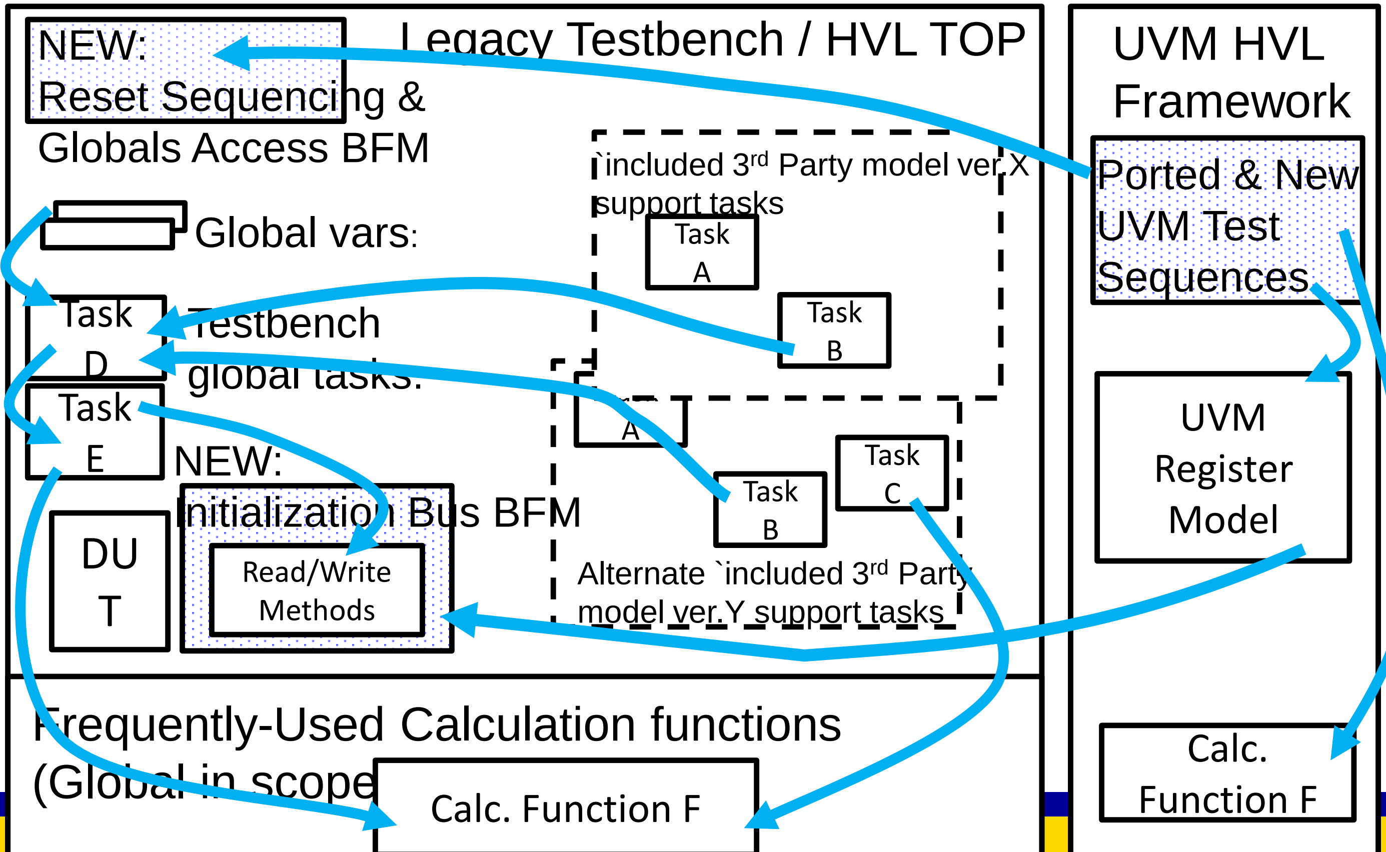
  - UVM sequence *could* provide all stimulus starting from t=0
  - Often at least *some* initial clocking/reset needs to occur before handoff to UVM sequence
  - Legacy Verilog testbench likely contains the initialization sequence and support tasks already

RECOMMENDATION: early on, determine how much of initialization task(s) need to be available/ ported to UVM sequences, and where the stimulus handoff will be to UVM sequencing

Northwest Logic decided to focus on new UVM test sequence capability

- Due to our initialization task complexity and support for multiple versions of global init tasks, the decision was made to execute initialization direct from HDL_TOP prior to beginning any UVM sequence

- Significant additional effort would have been required to
  - relocate tasks into interface modules
  - port/rewrite multiple versions of 3rd party model init sequences



Legacy Testbench / HVL TOP
NEW: Reset Sequencing & Globals Access BFM
Global vars:
Task D
Task E
Testbench global tasks:
`included 3rd Party model ver.X support tasks
Task A
Task B
A
Task B
Task C
Alternate `included 3rd Party model ver.Y support tasks
NEW: Initialization Bus BFM
Read/Write Methods
DUT
Frequently-Used Calculation functions (Global in scope)
Calc. Function F
UVM HVL Framework
Ported & New UVM Test Sequences
UVM Register Model
Calc. Function F

Converting legacy $display to UVM Info/Error Messaging

- UVM info messaging is already highly configurable
  - predefined verbosity levels UVM_LOW / MEDIUM / HIGH and UVM_DEBUG
  - Macros UVM FATAL and ERROR messages
- However, there is no standard usage recommendation for what to use / when
  - When to use FATAL (stop simulator) vs ERROR (continue execution)?
  - What parts of test sequence, BFM, scoreboards do you wish to hear from & under what conditions?   Under normal test-run output vs debug runs?
- Unless you consider / plan your usage model early, expect to do frequent revisit & revision of your code usage
  - Used throughout your test sequences, BFMs, UVM framework objects

RECOMMENDATION: define some convention up-front, to avoid excessive re-visit and revision to your BFM, Framework, and test sequence source code development

Example: Northwest Logic UVM_INFO Display Verbosity Conventions

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 vs expected) or other results (unexpected event observed), but test execution can proceed to potentially test other conditions	
`uvm_info	UVM_DEBUG	Do not use in user test sequences	Do not use here either. Used within UVM macro libraries. Reserve for debug of UVM library code
	UVM_HIGH	Do not use in user test sequences	Use for enabling debug statements from UVM Framework elements, such as drivers, sequencers, BFMs, testbench
	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	

Customized UVM Report Server

- The default UVM report server, while containing flexible Verbosity level filtering, is nonetheless excessively verbose in its output text

```
`uvm_info(rpt_idstr, $sformatf("%0s, %t", testname, $time), UVM_LOW)


# UVM_INFO
/home/user/projects/nwlogic/dram/verilog/testbench_uvm/seq/legacy_test_sequences/pow
erdown_test4_seq.svh(83) @ 8715625.0 ps: reporter@@powerdown_test4_seq
[powerdown_test4_seq] POWER_DOWN_TEST(#4), 8715625.0ps
```

  - UVM Verbosity levels do not significantly reduce the format or content of the report message, only which messages are output
  - While a UVM report “catcher” can be used to intercept and filter messages, there would be more overhead “catching” all the time to re format every message, vs modifying the initial report server output
  - Custom report server output:

```
# UVM_INFO [powerdown_test4_seq ] POWER_DOWN_TEST(#4), 8715625.0ps
```

Setting Report Tag String (ID) in Test Sequence Base Class

- UVM_info messaging macro accepts an ID string that is used frequently to sort/distinguish report output (examples):

```
`uvm_error(“PROTOCOL CHECKER”, $sformatf(“detected protocol violation XYZ at time: %0t”))
`uvm_info (“SCOREBOARD”, $sformatf(“checking transaction %0d for 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 seq on seqr %0d at time %0t”, i, %time), UVM_LOW)
```

- In a testbench where multiple instances of a test sequence may be spawned in parallel, it can be useful to uniquely distinguish each one’s output

```
# UVM_INFO    [use_test_select  ] ++++ Spawning parallel instances of selected sequence(s)
# 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_I                                                                        500
```

RECOMMENDATION: consider defining a report tag / ID string variable in the test sequence base, that you can assign to a per-sequence-instance value

Conclusion/Summary

This presentation discussed:

1. Issues and Solutions arising from Parameterized DUTs and Testbenches
2. Some Guidance for re-using support infrastructures (simulation scripts, etc)
3. Recommendations for migrating legacy global tasks/functions into BFM methods
4. Solutions and Suggestions for migrating legacy $display to UVM

While these reflect solutions chosen for implementation of a specific project, we hope that our recommendations & guidance may help others embarking on similar projects to upgrade or convert older Verilog testbenches to more modern UVM-style methodologies.

Thank you!