One Compile to Rule Them All: An Elegant Solution for OVM/UVM Testbench Topologies

Galen Blake
Altera Corporation
Austin, TX

Steve Chappell
Mentor Graphics
Fremont, CA

ABSTRACT:

As a design verification (DV) project is first started then steadily moves towards completion, the addition of verification features of increasing complexity in the testbench are a natural part of the development cycle. As these new features are added, a variety of controls are usually added to the testbench to manage the operations of these new and existing features. In many cases, this can lead to a chaotic set of text macro settings, compiler directives, configuration storage, parameters, plusargs, etc. to configure and control the topology and behavior of the testbench. Alternatively, multiple top-level testbench files (testbench netlists) or Perl script generated testbenches might be used to tame the complexity. Then for a final twist, modules and interfaces in the RTL context and class objects in the OVM/UVM context must all be configured to work together.

During one of our projects, we had some well developed concepts in the testbench that required only one testbench netlist for simulation. Nevertheless at one point we still found ourselves on the path to chaos having no less than 7 different language constructs and mechanisms being used to program the configuration and topology of the DV testbench.

At that point, we stopped and took the time to redesign our methodology for programming the configuration and topology. We began by conducting some small experiments to test out a few ideas and concepts.

One of the key requirements was that we wanted to have a single compile process. Not only would this save time and disk space across multiple runs, this enabled us to point engineers writing software-based tests to a single nightly build directory with all the libraries they would need for any testbench/system configuration pre-compiled. This single-compile requirement essentially meant that `ifdefs were not an option. Instead, we took advantage of the fact that our simulator supported design parameter re-assignment, as well as library search path definition, at elaboration time. This enabled the novel approach of using parameters to selectively instantiate code blocks via “generates” and pass configuration information to the testbench via non-parameterized virtual interfaces.

After a successful proof of concept, this effort was scaled up and used to replace all of the existing constructs and mechanisms. The result was an extremely elegant system that has many benefits:

- Simplicity : It is very simple to use and is equally simple to implement and enhance.
- Clarity : One set of controls manage and configure testbench components and topology across both the RTL and the OVM/UVM contexts.
- Minimal footprint : Verification components and class objects that are not needed for a test in a given topology will not be included.
- Flexibility : Topology decisions are deferred until run time and these choices make a large number of topologies available.
- Efficiency : This is all achieved without using text macros, compiler directives or troublesome class parameters. Furthermore, all of these capabilities are available with only one single compile operation.

This paper will explore the pros & cons of this system, and some of its alternatives, via a walkthrough of the system implementation in a real OVM/UVM testbench environment.

1. Introduction

Modern System-On-Chip (SOC) designs have large numbers of peripherals and complex configurations available for end users to choose from. This maximizes the number of applications that a particular family or platform can service. SOC silicon vendors design in as many features as possible to maximize the total available market for the silicon thereby increasing the profitability of the design. No single customer ever needs all of these available features within a single system. Additionally, pin count limitations further limit the availability of the peripherals that can be accessed.

Even though no individual end user can ever access all of these features, they all have to be verified before the chip goes out. That means the testbench must be able to support each of the features and peripherals and confirm that any possible combination of features selected by the user will work together correctly. With so many complex features and connectivity options, building a single testbench to support each of these combinations is challenging and many teams resort to building a few or even a few dozen testbench netlists to support these variations.

The following sections first explore traditional approaches that have been used in Verilog and in SystemVerilog testbenches. The final section then shows how the shortcomings of these approaches can be overcome by combining some of them with some newer features and concepts. This ultimate solution provides a simple and elegant way to manage and maintain the topology needed to meet the challenges required for complex SOC verification.

2. Approaches To Consider

a. The "Many Netlists" Approach

One approach to addressing these challenges is to construct different top-level modules (either with the same name or different names) in different files that instantiate the DUT and verification components in different ways in order to handle each of the variations encountered. These verification netlists may be re-useable for many tests but the number of netlists may still grow to several dozens.

```
module hdl_top_emac_tests();
wire tx,rx,…;
emac_vip_if emac_vip_if0(.tx(tx),.rx(rx),…);
`include "dut_inst.svh"
endmodule

module hdl_top_nic_tests();
reg tx = 0; wire rx;
`include "dut_inst.svh"
endmodule.
```

Building many testbench netlists leads down a messy path that will be very difficult to maintain when global updates are needed across each of these netlists. This solution also requires users that are developing new tests or adding functionality to constantly ask themselves the questions: "Can I reuse one of these existing netlists or do I need to author yet another new one?" "If it needs to be a new one, which one would be the best one to clone?" "What do I do if I need some features from one existing netlist and some features from a second netlist but then discover that they require mutually exclusive resources for methodologies?" In such cases, users might find themselves either rewriting already existing capabilities several times to avoid conflicting concepts, or looking at text macros to try to minimize the copy/paste activity. This leads us to the next approach.

b. The Text Macro Approach

Text macros are the most commonly used approach by users for sizing vector signals and configuring testbench topology. Text macros are substitutions processed based on preprocessor commands (`ifdef, `define, et al). They can be used to choose values (e.g. sizing vectors) or for including/excluding certain blocks of code. These substitutions are done by the compilation tools before the code is passed on to the compile/analyze process. Thus, any change to these macros requires the files that use them (and the files that depend on those files, and so on) to be recompiled.

```
module hdl_top ();
`ifdef USE_EMAC0
  wire tx,rx,…;
  emac_vip_if emac_vip_if0(.tx(tx),.rx(rx),…);
`else
  reg tx = 0;
  wire rx;
`endif
  dut dut(….);
endmodule

class my_env extends uvm_env;
  function void build_phase( uvm_phase phase);
`ifdef USE_EMAC0
      emac0_agent = new(…); // or create from factory
`endif
  ….
```

For small models and block-level testbenches (that will never be re-used at the system-level), this may be a reasonable approach. However, as the testbench grows in size and complexity the widespread use of text macros and the potential hazards of name collisions or other conflicting settings start to grow. Eventually a lot of time gets wasted hunting down these types of issues. For example, someone might decide to use the text macro `MEM_WIDTH or `INCLUDE_MONITOR in a model where it is defined in the top of the file where that model is defined. Then someone else decides to use the same text macros in another model where they are set to different values and come from an include file or the compiler commandline. Suddenly the order in which these files are compiled becomes very important and the possibility of getting the wrong value into the wrong model is very real.

When text macros like `INCLUDE_MONITOR are used, they are usually defining the presence or absence of important testbench resources. If it is used in multiple models, then users must accept an all or nothing selection, live with redefinition (usually a compiler warning or error), or expand the number of text macros to make sure each of them have unique names. When multiple developers are actively working on a project it is easy and common to encounter conflicts. Then also consider that there may be thousands of files involved from many individuals and multiple development teams making the probability of conflicting text macros even higher.

Beyond the name collision/confusion dangers and maintenance, however, one huge disadvantage of the use of macros is the need to re-compile your design & testbench when macro definitions change. This means we need to limit our use of macros to things that are constant across all tests for this project (e.g. the width of the data bus on the APB port of the USB IP, which translates to a number of physical wires in the design). Otherwise we cannot have a single compilation that supports all of our configurations and topologies.

c. **The Parameter Approach**.

Similar to text macros, parameters are often used to size vector signals and to a lesser extent to configure the testbench topology. Unlike text macros though, parameters do not need to be resolved until elaboration time. Also, unlike text macros, parameters have a more narrow, well-defined scope. That is, they are better encapsulated and do not have the same compile-order/name-collision pitfalls that text macros do.

```
module hdl_top ();
  parameter USE_EMAC0 = 1;
if (USE_EMAC0 == 1) begin
  wire tx,rx,…;
  emac_vip_if  emac_vip_if0(.tx(tx),.rx(rx),…);
end else begin
  reg tx = 0;
  wire rx;
end
  dut dut(….);
endmodule

class my_env #(int USE_EMAC0=1) extends uvm_env;
  `uvm_component_param_utils(my_env);
  function void build_phase( uvm_phase phase);
    if (USE_EMAC0)
      emac0_agent = new(…); // or create from factory
  ….
```

For these reasons, parameters are popular and work well in HDL models (e.g. modules). However, when you move to the testbench side, having parameters in your classes (or even in the

SystemVerilog interfaces which will need virtual interface handles in the testbench), usually creates real headaches. The problem is that each unique combination of parameter values creates a new specialization of the "thing" you are parameterizing. And these specializations are not type-compatible with each other. Class object references and virtual interfaces are just handles with specific types that must be declared and exactly match the "thing" you are assigning them to. Modules don't have this problem because there is no such thing as a handle to a module; modules are static & are referred to by a hierarchical name only.

In OVM/UVM specifically, adding parameters to your classes also means you are restricted to the type-based factory and cannot use the string-based factory. There are some approaches that attempt to get around this problem [1][2], but in general it is just more desirable to keep the number of parameters in classes and interfaces to an absolute bare minimum.

d. **The Plusarg Approach**

Another approach is to try to bring in the configuration information via plusargs. This is used in OVM/UVM testbenches to select the test that will be run (+UVM_TEST=sanity_test), so a temptation of testbench architects might be to start down the path of bringing in all of the necessary configuration information as plusargs on the simulator commandline. This approach requires a register or set of registers to be defined at some location in the testbench. It is possible to access these registers from both HDL (modules and interfaces) and HVL (classes) contexts. Some reasonable default values for the registers are chosen, and then overridden in an initial block if the corresponding plusarg is assigned.

```
interface config_regs ();
  reg [`MAX_CFGS-1:0] tb_cfg;
  initial begin
    tb_cfg  = `DEFAULT_TB_CFG;
     if ( $value$plusargs("USE_EMAC0=%d", use_e0 ) )
        tb_cfg[`USE_EMAC0]   = use_e0;
  end
endinterface
```

The main issue here is that while this approach works for the dynamic testbench, this information comes too late to affect the static topology. Interfaces, modules, bus sizes, etc all need to be locked down at elaboration time. So now we end up duplicating a significant amount of configuration information across plusargs and whatever the HDL uses (pick from the approaches above). This leads to the potential for mismatch errors between the HDL & HVL contexts that are difficult to debug and results in much head slapping.

3. The HDL/HVL Coordination Challenge

Before getting into some of these details, it is important to consider that one of the challenges and requirements for an elegant solution is to coordinate the topology of modules and interfaces in the HDL context to associated classes or agents in the HVL context.

Maintaining multiple netlists grows even more impractical when they must be paired with multiple OVM or UVM environments. So this is clearly not a desirable solution. Moreover, this requires users to compile and store compiled databases for each variation making it more difficult to manage the workflows and increases storage and complexity. It is strongly desirable to have a single compiled database that is created once and then reused with different elaboration options for each of the needed topologies.

Using text macros is not a very elegant solution since the addition of the OVM or UVM to the HDL context just creates more places to set them or read them incorrectly and increase conflicts. Considering that the text macros used across many files are hard to manage, using them between the HVL and HDL only adds to the problem. And of course it violates one of our key goals of not having to recompile for every new topology/configuration.

The disadvantages of parameters in the OVM/UVM side may at first seem to rule out the possibility of using parameters. However, the problem of duplicated config settings experienced with the plusarg approach led to the question of whether the parameter information could be passed to the HVL side as something other than a parameter.

4. Our Approach

a. Parameter-controlled "generates"

First consider the HDL context alone. Within HDL models, parameters are widely used in both DUT models as well as testbench models for vector sizing. When the 'generate' statement was added to Verilog, the parameter became even more useful since it can be used in a manner exactly like the `ifdef text macro. So within the HDL context, a rich set of parameters such as USE_EMAC0 for example can be used along with a generate statement (explicit or implicit) to define the required topology. Within the body of the generate statement a user may instantiate modules or interfaces, make wired connections, or assign tie-off or initial values to wires and variables. In this case, the parameter, or some parameter expression, can be used to give the user the ability to dial in exactly the topology they need in the HDL. The HDL side follows the same template as the sample code example above under the "Parameter Approach". You may also note that there are some `defined macros for bus widths – again these are used for readability (and project-to-project reuse), but their values are constant for all tests/configurations of this project. Also, there are some VIP interfaces where parameterization could not be avoided, but again these are defining bus widths that are constant.

```
module hdl_top ();
  parameter USE_L3_ACP_AXI_VIP              = 0;
…
generate if (USE_L3_ACP_AXI_VIP) begin : i_acp_axi_vip
axi_monitor_vip #( .ADDR_WIDTH (`ACP_ADDR_WIDTH),
          .RDATA_WIDTH          (`ACP_RDATA_WIDTH), …
          .AXI_ID              ( "acp_axi_monitor" )
        )  acp_axi_monitor (.mif(acp_axi_vip_mst));
   end
  endgenerate
…
endmodule
```

b. Avoiding parameterized classes

Next consider the HVL context. In many cases for testbench components like interfaces, there are corresponding agents that exist in the HVL context. As we saw, the existence or configuration of these interfaces is controlled by parameters, so ideally it would be very useful to leverage these already existing parameters from the HDL. But we already know we don't want parameters in our classes if we can avoid them.

Since we need parameters in the exact location where we do not want them, we needed to find another way to leverage the valuable topology information they contained. We created a special configuration interface, a "parameter transport interface", which itself is not parameterized. In this parameter transport interface, we matched up a scalar signal for each of the binary USE_* parameters. Some additional non-binary parameters (e.g. data, address) were mapped to vector signals. The signals in these interfaces are assigned to the parameters and then carry their values across from the HDL to the HVL context. This works because our testbench is dynamic and created at runtime. Thus, it doesn't need the information in these parameters at elaboration time, like the static HDL components do.

```
interface config_tb_if();
    bit use_l3_acp_axi_vip;
…
endinterface

module hdl_top ();
  parameter USE_L3_ACP_AXI_VIP              = 0;
…
  config_tb_if cfg_tb();
  assign cfg_tb.use_l3_acp_axi_vip           =
                          USE_L3_ACP_AXI_VIP;
  initial uvm_config_db #( virtual config_tb_if )::set( …);
…
endmodule
```

Now in the HVL or class context, we instantiate a virtual interface for the configuration interface in the HDL and then we can test the values of any needed parameter without using a parameterized class. So taking the example above, if we need to conditionally construct and configure an agent for the EMAC0 interface in the HDL we can examine the associated interface signal that carries the exact value from the parameter in the HDL context and know if we should or should not be constructing and configuring the agent.

```
class my_env extends uvm_env;
  `uvm_component_utils(my_env);
virtual config_tb_if cfg_tb;

function void build_phase( uvm_phase phase);
  cfg_tb = uvm_config_db #( virtual config_tb_if )::get(.. );
  if ( cfg_tb.use_l3_acp_axi_vip) begin
      l3_acp_axi_a = axi_agent
        #(ACP_axi_p)::type_id::create("l3_acp_axi_a", this);
    end
. . .
```

The uniqueness of these parameters tends to avoid naming conflicts. And since they are all defined in the same scope, it is fairly obvious to the compiler and developers if conflicts occur.

5. Additional Considerations

a. Different Versions of the Same Module

The essential ideas described thus far work very well for a lot of decisions in the testbench and the top level DUT instantiating scope – do I want a VIP instantiated here, or just tie off the signals? What are the initial pin/register values?

But what about my desire to use this same test environment for a subsystem or even a block level test? Or what if I just want a different version of one or more of the modules in my DUT – like swapping out the real processor RTL for an ISS-based model? The parameter-conditional generate approach described above could work. But that would mean 1) giving all definitions of a given module a unique name, and 2) duplicating a lot of module instantiation code inside the if-else generate blocks.

So for this problem, we did choose to compile the different definitions of the modules into separate libraries and then select them via command-line-specified library path search order at elaboration time. This worked for some of these limited choice cases, but it did require some discipline to keep the number of work libraries to a minimum.

We also considered using Verilog "config" constructs, but didn't like dealing with the extra top level that this created. We probably could have done something as well with `uselib directives, but 1) `uselib is not part of the SystemVerilog standard, and 2) controlling the definition selection from a few characters on the commandline (-L libname …) via our scripts was far easier to manage than adding anything to the SystemVerilog code.

b. Simulator Requirements

While on the topic of commandline options, the parameter-conditional generate approach (aka the elegant solution), has significantly more benefit when your simulator supports parameter value overriding at elaboration time. If your parameter values are locked down at compile time, then you will end up needing to do a recompile of at least some subset of your top level module(s) with almost every run. If you can defer specifying the parameter values to a commandline option at elaboration/optimization time (e.g. "mysim –GUSE_FOO=1 –GUSE_BAR=0 …"), then you can make your topology decisions as granular as you please and still have a single compile process.

6. Conclusion

Using the techniques described above provided a very effective solution to maximizing the re-use of our test environment across a multitude of DUT configurations and verification targets. The ability to have a single compile phase whose output can be re-used by multiple simultaneous, and wildly different, simulation runs has significant advantages. Saving time on recompiles, saving disk space, ease of use, and consistency of results across disjoint teams all made for a smoother running and efficient project. We were also able to achieve these advantages without sacrificing performance because the simulator was able to optimize away any code inside parameter-disabled generate blocks before run time. Our one compile really did rule them all.

REFERENCES

[1] David Rich, Adam Erickson. "Using Parameterized Classes and Factories: The Yin and Yang of Object-Oriented Verification". Proceedings of DVCon 2009.

[2] Brian Ramirez, Michael Horn. "Parameters and OVM — Can't They Just Get Along?". Proceedings of DVCon 2011.