Perplexing Parameter Permutation Problems? Immunize Your Testbench

Alex Melikian

Paul Marriott

Verilab

Montreal, Quebec, Canada

verilab.com

@verilab

ABSTRACT

RTL parameters are used frequently in designs, especially IPs, in order to increase flexibility for reuse or different target applications and products. As demand for feature density or power efficiency increases, the number of permutations of valid RTL parameters becomes increasingly difficult to manage. A well structured and flexible verification testbench is therefore required to handle the multiplicity of combinations of these parameters. Failure to account for this could result in a testbench that would become difficult to maintain, or, worse, no longer adequate to find potential bugs in all possible permutations of the parameters. This paper proposes solutions for structuring various segments of a UVM test bench to handle designs with large amounts of DUT parameter combinations, minimizing maintenance effort and risk of bug escapes.

Table of Contents

1. Introduction

The challenges developing digital intellectual property (IP) continually increase as demands on flexibility for applications, SoC integration and power efficiency increase. These challenges are not just for third party commercial IPs, designed to suit a wide variety of customers, but internal IPs as well, in order to maximize the reuse across a wide variety of in-house designed chips.

For these reasons, digital IP developers turn to RTL parameters, which can be used to structure their IP to be scalable and flexible for different objectives. For example:

- Scalability in the number of identical sub-blocks (ie number of channels or ports)
- Exclusion of sub-blocks if related functionality is unnecessary, reducing footprint and power consumption of the IP
- Selection and tweaking of swappable sub-blocks for optimization based on specific target applications the IP is expected to perform

Hence a single IP may operate in a variety of configurations set by the RTL parameters, where many permutations are valid.

Verification teams face unique challenges when dealing with these IPs. Due to the presence of these RTL parameters, bugs may exist in RTL code when only a specific permutation of parameter values are applied. This implies testbench (TB) developers must employ a strategy where functionality is verified for all valid design RTL parameter permutations.

The typical verification strategies in such situations can be summarized in two approaches: a testbench that can be parameterizable and made as specific as the RTL parameter values as described in [4], or a more generic one that adapts to those same RTL parameter values. Though there are pros and cons with both strategies, this paper proposes solutions aligned with the latter approach in a range of areas of a typical UVM SystemVerilog testbench.

2. Overview of Proposed Solution

The core philosophy of the solutions proposed in this paper rest on two main principles:

- A mechanism allowing the testbench to easily and automatically extract RTL parameter values
- Code structures leveraging off the extracted RTL parameter values to be as flexible as possible.

The approach of a more flexible, generic testbench possesses great advantages over the approach of making testbenches parameterizable like the RTL.

Firstly, there is the potential for the testbench to necessitate only a single compilation and applicable against any RTL parameter setting. This eliminates costly computation time in regressions, especially if the DUT may have a large parameter permutation space. Secondly, coverage merging can also be simplified as the covergroup can be defined as unique for all RTL configurations. Furthermore, the testbench can be more readable and easier to understand, even in situations of large numbers of RTL configuration sets. A larger number of RTL parameters can lead to a proliferation of mirroring parameters in the testbench, making it more cumbersome to develop, maintain and understand.

With the above principles and their advantages established, the following sections will delve into greater elaboration and detail on how solutions can be implemented in a testbench. The particular

challenges this approach may pose to a typical UVM register model will also be explored in depth.

3. Implementation of Solution

3.1 Capturing RTL Parameters

The key to ascertaining RTL parameters from the DUT is by using a "bind" interface as detailed in [3], in addition to employing encapsulated members in that interface. For example, let's assume our DUT has the following top level module declaration:

```
module mydut
   #( parameter MYDUT_PARAM1 = 12,
       parameter MYDUT_PARAM2 = 8,
       …
       parameter MYDUT_PARAMN = 1)
     ( input reset,
       input clock,
       … );
```

A SystemVerilog testbench can declare an interface containing RTL parameters intended to map to the DUT's corresponding RTL parameters. This interface can also contain an embedded structure (or a `uvm_object` derived object) to encapsulate and capture these RTL parameters. Using the DUT example above, such an interface can be described like this:

```
interface tb_binding_to_mydut_iface
   #( parameter MYDUT_PARAM1 = 12,
       parameter MYDUT_PARAM2 = 8,
       …
       parameter MYDUT_PARAMN = 1)
     ( input reset,
       input clock,
       … );
       rtl_info_struct_t  rtl_info;

   function void capture_rtl_info(string path);
       rtl_info.mydut_param1 = MYDUT_PARAM1;
       rtl_info.mydut_param2 = MYDUT_PARAM2;
       rtl_info.mydut_paramN = MYDUT_PARAMN;
       uvm_config_db#(rtl_info_struct_t)::set(null,{path,".tb_env.*"},
                                          "rtl_info_struct", rtl_info);
   endfunction
endinterface
```

This interface will be coupled to the DUT (and hence its signals and RTL parameters) using the 'bind' statement and mapping corresponding RTL parameters into the binding interface:

```
bind  my_dut  tb_binding_to_dut_iface #(
  .MYDUT_PARAM1(MYDUT_PARAM1),
  .MYDUT_PARAM2(MYDUT_PARAM2),
  [...]
  .MYDUT_PARAMN(MYDUT_PARAMN))
  tb_binding_to_dut_harness(.*);
```

As suggested in [3], the `capture_rtl_info()` method should be called in a designated location in the test harness to ensure the RTL parameters are captured before the testbench environment components are 'built'. Typically this is done in non-reusable code areas such as the uvm_test derived classes, like this:

```
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  [...]
  env = my_dut_tb_env::type_id::create("env", this);
  top_module.dut_inst.tb_binding_to_dut_harness.capture_rtl_info(
                                                    env.get_full_name());
  [...]
endfunction
```

With the above setup, the captured RTL parameter information is stored in UVM the 'config_db'. We now have the capability to access this captured RTL parameter information from any component under our TB environment.

Though you would lose the advantages in automated testbench connectivity described in [3], an alternative but equally valid initialization would be to call the `capture_rtl_info()` method before the `run_test()` call. This is typically located in the 'initial' block of the top encapsulating TB module as such:

```
initial() begin
  tb_binding_to_dut_harness.capture_rtl_info("my_env_name");
  run_test();
end
```

This approach creates the following advantages:

- Eliminates the need for `` `define `` statements representing RTL parameter values to be spread across the testbench. This language construct is syntactically poor, prone to introducing difficult bugs and necessitates a large number of files to be created covering all permutations of RTL configurations.
- Instead of manually creating files covering all RTL parameter permutations (and associated headaches to maintain them), a separate script (or even SystemVerilog constrained random code) can create the parameter files to be loaded.
- Eliminates a coordinated two-pass randomization scheme for RTL parameters and testbench components where the randomization seed has to be carefully preserved. The

two segments are now decoupled and can be run separately. This will be elaborated further in the following section.

3.2 Two Pass Randomization Decoupled

Objectives for verifying a DUT with many RTL parameters is generally twofold. As discussed, the first objective would be to structure the testbench adaptable to all parameter value permutations. The other objective would be establishing a method to generate and apply all the necessary combinations of parameter values to the DUT achieving coverage goals.

A common method used to accomplish these dual goals would be a "two pass" approach. This approach consists of two main segments. Firstly, the testbench would randomize testbench components in addition to objects reflecting RTL parameter values, but be configured to stop after this initial point. The randomized RTL parameter values from the first pass would be recovered and applied to the DUT for compilation. Finally, a second pass of the testbench invocation would be launched, using the same randomization seed for consistency with the first segment (including RTL parameter values), and configured this time to run the simulation.

Though this method achieves the two goals, it has its drawbacks. Mainly it introduces some overhead in testbench development as well as burden users to book-keep the seeds needed to maintain randomization consistency in both segments of the "two pass".

Fortunately the interface-bind solution described in the previous section allows the testbench to be automatically aware of the RTL parameters. Therefore it enables testbench developers to avoid the drawbacks of a "two pass" approach as well as open new options for achieving the dual verification goals.

Firstly, code to create a file defining RTL parameter values (like a text file to be provided in the compilation invocation) can be generated by either a preferred scripting language, or even with a small executable SystemVerilog module. No DUT would be necessary, but rather a small construction of a SystemVerilog object creating a text-based file defining RTL parameter values, residing in a self-contained module with an 'inital' block executable by simulator invocation. The advantage of a SystemVerilog approach over a scripting language would be that the user could leverage off the constrained random generation for generating sets of RTL parameter values.

Secondly, thanks to the interface-bind solution, the testbench can automatically adjust to any permutation of RTL parameter values, regardless of how they have been set. This includes randomization of objects influenced by RTL parameter settings, be it at run time or even constraint declarations. The key would be to extract the structure reflecting RTL parameter values at the construction or build stage of an object, and using these values in the constraint declarations. This is demonstrated with the following sample code:

```
class my_tb_rand_class extends uvm_object;
      [...]
      rtl_info_struct_t  dut_rtl_param_info;
      protected int      dut_rlt_param_num_ports;
      rand      int      rand_port_number;
```

```
      // constraint decleration
      constraint port_num_constr {
       (rand_port_number>=0) && (rand_port_number<dut_rlt_param_num_ports);
      }

      // retrieve rtl_info_struct reflecting captured RTL parameter values
      function new(string name);
         if (!uvm_config_db#(rtl_info_struct_t)::get(null, "*",
                                    "rtl_info_struct", dut_rtl_param_info))
            `uvm_fatal(get_name(), "Failed to get …")
         dut_rlt_param_num_ports = dut_rtl_param_info.number_of_ports;
      endfunction
endclass
```

3.3 UVM Register Modeling

A DUT's RTL parameters may affect attributes of its registers such as field widths, reset values, or even exclude an entire field out of the design. The following are examples of how UVM register model code can be structured to be flexible to RTL parameter values.

3.3.1 Register Reset Values

In most applications, flip-flop based registers often reset to an all 0's value. However, an all 0's value in an IP register may be an invalid value for some SoC applications. Hence RTL parameter values can be used to allow flexibility in register reset values in the IP block.

The use of `define statements with matching values between RTL and the UVM testbech is possible, however we have covered all of the disadvantages of this approach previously. Instead, the following solution is proposed: referencing the captured RTL parameter structure and using the UVM built-in [1][2] uvm_reg::set_reset() and uvm_reg::reset() methods in the DUT's register model. This is demonstrated in the following example code where a DUT register field labeled 'myreg_featureA_regfield1' can have its reset value changed via an RTL parameter named 'MYDUT_FEATUREA_REGFIELD1_RESETVAL':

```
      rtl_info_struct_t  dut_rtl_param_info;
      uvm_reg_fields     dut_reg_fields[$];
      unsigned int       dut_field_resetval;
      [...]
       // retrieve rtl_info_struct reflecting captured RTL parameter values
       if (!uvm_config_db#(rtl_info_struct_t)::get(null, "*",
                                  "rtl_info_struct", dut_rtl_param_info))
        `uvm_fatal(…)

      // change affected register reset value to reflect RTL value
      dut_field_resetval = dut_rtl_param_info.mydut_featA_regfeild1_resetval;
      dut_regblock.featureA_reg.field1.set_reset(dut_field_resetval);
      dut_regblock.featureA_reg.field1.reset();
```

The above code acquires the structure containing captured RTL parameters via the config_db. Next, it would pass on the corresponding register field reset value onto the `set_reset()` method, changing the register field's modeled reset value (which can be updated dynamically). Finally the `reset()` method is called to ensure the register field model has updated its new reset value to be used in predictions during simulation execution.

Care is needed as to where this is implemented, as the register model would have to be built in the environment beforehand. The '`end_of_elaboration()`' phase is an appropriate location as the register model would typically be built at this stage, and the simulation has yet to begin executing its checks.

3.3.2 Register Inclusion/Exclusion

For silicon or power efficiency objectives, RTL parameters may be used to include or exclude features, along with their implementation logic, given a specific design configuration. Registers are also be impacted by such an exclusion. For example, a given address with registers associated with an excluded feature may become 'reserved', and expected to no longer have consequential effects on the rest of the design.

However, it is critical that this address should be examined in register tests for both scenarios where the RTL parameter would include or exclude it in the design. Simply omitting these addresses from examination in register tests if the feature is excluded could allow bugs, such as register mapping errors, to slip through.

To achieve this, a solution similar to the one in the previous section is proposed but with the addition of the built-in *`uvm_reg::set_access()`* method. Taking an example of a register declared 'myreg_featureA_reg1' under a register block modelling a DUT with an RTL parameter 'MYDUT_FEATUREA_ENABLE' determining its inclusion, the following code can be applied in the testbench:

```
rtl_info_struct_t  dut_rtl_param_info;
uvm_reg_fields     dut_reg_fields[$];
[...]

 // retrieve rtl_info_struct reflecting captured RTL parameter values
 if (!uvm_config_db#(rtl_info_struct_t)::get(null, "*",
                                "rtl_info_struct", dut_rtl_param_info))
  `uvm_fatal(…)

// change affected registers in the model to "RO" all 0's
if (dut_rtl_param_info.mydut_featureA_enable == 1'b0)
  begin
    dut_regblock.myreg_featureA_reg1.get_fields(dut_reg_fields);
      foreach(dut_reg_fields[r])
        begin
          dut_reg_fields[r].set_access("RO");
          dut_reg_fields[r].set_reset(0);
          dut_reg_fields[r].reset();
       end
  end
```

The above code has turned the 'myreg_featureA_reg1' into a reserved address in the register model if the 'MYDUT_FEATUREA_ENABLE' RTL parameter has been configured to exclude it from the design. However, its entry remains in the register model, which means that register test sequences will still access its address in addition to register value prediction schemes expecting it to behave like a reserved address. Therefore whether the register is included or not in the design, register tests will automatically verify the appropriate behavior in both scenarios.

3.3.3 Register Field Sizes

RTL parameters can also influence the sizes or widths of register fields. A common example would be the case of a register containing a per-channel vector of 'enable' bits (i.e. the first bit would be the enable for first channel, the second bit would be the enable for the second channel, etc.). For efficiency reasons, an IP can have an RTL parameter to set the number of channels in a particular application, impacting registers field lengths like the 'enable' example above. Therefore, if the RTL parameter is set to '1', a corresponding register field would have the width of 1 bit, whereas a setting of '2' would change the width to 2 bits, etc, etc.

Unlike the case of reset values, the UVM classes for register fields (`uvm_reg_fields`) do not have methods where field width settings can be dynamically changed. Register widths are set at creation (or building) of the model and cannot be modified thereafter, mimicking register field sizes in RTL which always maintain their size. However, UVM register model classes and factory class mechanisms can be used as a solution using a different approach.

This solution involves employing a small modification in how UVM register classes are defined to model a DUT, using the captured RTL values structure and finally employing the override mechanism of UVM factory's classes.

Firstly, the UVM register classes needs to be defined slightly differently. Using the "channel enabled" example described earlier, the following code would be a typical declaration should the register be the same size, regardless of RTL parameter settings:

```
class myreg_featA_channelen_reg extends uvm_reg;
      `uvm_object_utils(myreg_featA_channelen_reg)
      uvm_reg_field channel_en;

      virtual function void build();
        Channel_en = uvm_reg_field::type_id::create("channel_en");
        channel_en.configure(
          .parent  (this),
          .size    (8),       // *** same size regardless of RTL parameters
          .lsb_pos (0),
          .access  ("RW"),
          [...]
      endfunction
endclass
```

With a few simple modifications where instead of using static or hard-coded values in the `uvm_reg_field::configure()` calls, virtual methods can be used instead to achieve the same

result. Taking the previous example:

```
class myreg_featureA_channelen_reg extends uvm_reg;
      [...] // same UVM boiler-plate code and members

      virtual function void build();
        channel_en = uvm_reg_field::type_id::create("channel_en");
        channel_en.configure(
          .parent(this),
          .size  (get_channel_en_size()), //*** method replaced hardcoded
                                          //    value
          .lsb_pos(0),
          .access ("RW"),
          [...]
      endfunction

      virtual function int get_channel_en_size();
         return 8;  // simple method body, but can be overwritten
      endfunction
endclass
```

The key difference is the virtual method returning a value instead of hardcoded value applied directly in the configure calls. Code structured like this opens the opportunity to overwrite these methods and return something different based on factors, such as RTL parameters. Once again, the captured RTL parameter value structure can be acquired and used to return the desired value in the method call, configuring the register field size automatically. The following code demonstrates this:

```
class myreg_featA_channelen_extended_reg extends myreg_featA_channelen_reg;
      `uvm_object_utils(myreg_featA_channelen_extended_reg)

      virtual function int get_channel_en_size();
         rtl_info_struct_t  dut_rtl_param_info;
         if (!uvm_config_db#(rtl_info_struct_t)::get(null, "*",
                                  "rtl_info_struct", dut_rtl_param_info))
           `uvm_fatal(…)
         return dut_rtl_param_info.featureA_number_of_channels;
      endfunction
endclass
```

The last step is use UVM factory override calls to swap in the extended register class with the existing one. As described in [1], registers in a model are typically generated from the factory and capable of being overridden. Following the examples, the override statement would be done using a standard `set_type_override_by_type()` call in UVM's built-in factory as such:

```
set_type_override_by_type(myreg_featureA_channelen_reg::get_type(),
                          myreg_featureA_channelen_extended_reg::get_type());
```

A shrewd reader could question why the functionality is split between two classes along with an override to complete the solution. It can be easily argued that the solution to the problem in this section can exist under one class declaration. However, the proposed solution in this section considers that most development teams use some form of auto-generation tool for generating the code of a register model in their testbench.

This solution fits well with such a methodology and flow, where testbench specifics such as the `rtl_info_struct_t` structure in the `myreg_featureA_channelen_extended_reg` class can be overlaid over a more generic definition like the one in `myreg_featureA_channelen_reg`. The code structure of the `myreg_featureA_channelen_reg` class is easier to apply in various projects and to generate from a scripting tool reusable across multiple designs. Whereas the code of `myreg_featureA_channelen_extended_reg`, with its referencing of objects specific to a testbench, will reside only with its corresponding testbench project source code.

3.4 Multiple Instances For Override

The previous section demonstrated how the `set_type_override_by_type()` call in UVM is used to refine elements of a register model in a DUT, specializing them to corresponding RTL parameters. Though this remains a powerful tool in a UVM testbench, it would create the cumbersome situation if multiple instances of the same register exist, where each instance would need to be specialized differently.

For example, a DUT could have multiple ports with the same set of registers, however each port may be specialized differently as they all have different number of channels. The brute force approach would be to state an override call targeting each register instance of a port. Though effective, it could lead to considerable code bloat for essentially the same operation.

Fortunately, the built in UVM name of the object instance can provide an index number that can be exploited. For example, an array of UVM registers modeling corresponding instances of a DUT would look typically like the following when created:

```
myreg_chanen_p0 =
myreg_featureA_channelen_reg::type_id::create("myreg_chanen_p[0]");
myreg_chanen_p1 =
myreg_featureA_channelen_reg::type_id::create("myreg_chanen_p[1]");
myreg_chanen_p2 =
myreg_featureA_channelen_reg::type_id::create("myreg_chanen_p[2]");
[...]
```

Hence a simple 'for' loop can used to roll up the `set_inst_override_by_type()` calls, reducing code bloat as demonstrated below:

```
for (int pidx=0; […] )
  set_inst_override_by_type( $sformatf("tbenv.regmodel.myreg_chanen_p[%0d]",
                                       pidx),
                             myreg_featureA_channelen_reg::get_type(),
                             myreg_featureA_channelen_extended_reg::get_type());
```

Alternatively, small adjustments can be made from within the extending classes to automate the process of extracting information and matching it with the specific RTL register instance it is modelling.

This index number embedded in the UVM object name can be exploited to extract the corresponding information from the struct holding RTL parameter information. This is demonstrated in the following code in the `myreg_featureA_channelen_extended_reg` presented earlier, this time with modifications to reflect the multiple instances mirroring the DUT:

```
class myreg_featA_channelen_extended_reg extends myreg_featA_channelen_reg;
      `uvm_object_utils(myreg_featA_channelen_extended_reg)

      virtual function int get_channel_en_size();
         rtl_info_struct_t  dut_rtl_param_info;
         int                portnum

         //… get dut_rtl_param_info from uvm_config_db like previous examples

         // extract, convert port number integer from the
         // "myreg_chanen_p[n]" strings in the object name
         portnum = extract_portnum_from_name( get_name() );

         return dut_rtl_param_info.featureA_number_of_channels[portnum];
      endfunction
endclass
```

The difference here is the call to a method `extract_portnum_from_name()` which simply returns an integer by extracting and calculating from the object names' "`*.myreg_chanen_p[n]`" string patterns. This approach would be an alternative to the `set_inst_override_by_type()` factory method, where a tidy single call to `set_type_override_by_type()` would cover all instances of that register type in the DUT.

3.5 Coverage of RTL parameters

One great advantage of functional coverage over its RTL code coverage counterpart is that it allows seamless merging of functional coverage sampling data. RTL parameter information can be sampled like any other data in cover groups for coverage closure objectives. Furthermore, this data can be merged and analyzed after regression runs, even across different RTL snapshots which vary over parameter value permutations.

Cover groups are typically declared and housed within a single or set of defined classes in the test

bench, designated for coverage collection operations. By simply ensuring this class is derived from `uvm_object` or `uvm_component`, it will automatically gain access to the data stored in the structure reflecting RTL parameter values via the `uvm_config_db`. Therefore, RTL parameter values can now be included in functional coverage constructs and bins for desired analysis. This is demonstrated in example code below:

```
class mytb_covergroups extends uvm_object;
      `uvm_object_utils(my_tb_covergroups)
      rtl_info_struct_t  dut_rtl_param_info;

      covergroup  covgrp_my_dut_rtl_params;
            cp_rtl_params_num_ports: coverpoint dut_rtl_param_info.num_ports;
            [...]
      endgroup

      function new(string name);
         //… extract into dut_rtl_param_info via uvm_config_db like
         //  previously
         [...]
         covgrp_my_dut_rtl_params = new();
         covgrp_my_dut_rtl_params.sample();
      endfunction
endclass
```

Furthermore, it may be imperative for verification to ensure certain stimulus was created along with specific RTL parameter settings. Since both areas would be subject to randomization, there would be no implicit guarantee of a specific combination having been verified, without appropriate coverage modelling. The above scheme enables the creation of metrics covering functions of the DUT crossed with RTL parameters. This ensures verification teams would have the coverage setup needed to analyze and confirm the intended stimulus scenarios and RTL parameter combinations to be covered.

4. Conclusions

The proposed practices and code structures in this paper were employed for a verification project on an IP module possessing over fifty individual RTL parameters. The coverage space on valid IP parameter permutations was over 1500 distinct bins, necessitating over 100 different compilation snapshots of the DUT to fill.

Despite these large numbers, thanks to the application of solutions presented in this paper, only one compilation of the testbench code and related libraries was needed to cover all DUT RTL parameter configurations. This minimized compilation times in regression runs as well as enable the testbench to seamlessly adjust to any combination of RTL parameters the DUT possessed. Furthermore, the process of randomizing RTL parameters enabled the verification team to expose DUT bugs related to RTL parametrization.

As mentioned previously, the proposed solutions follow a philosophy of building a testbench which is generic and adaptable to the RTL parameters of the DUT. One of the common drawbacks of such

an approach would be code bloat in the test bench to enable its adaptability. This occurs typically in areas where individual and distinct code sections would have to be generated, corresponding for the each distinct value of an RTL parameters. An example would be pins on a DUT shared between two distinct interface protocols, of which only one can be included by an RTL parameter. Separate code in the testbench would have to exist to handle both cases even though they share the same pins.

However, in conclusion, it is the conviction of the authors that the benefits of the solutions in this paper considerably outweigh the associated costs and shortcomings. They remain a valuable tool in enabling flexibility for a testbench to handle a large number of functional permutations created by RTL parameters of a DUT.

5. References

[1] Accelera, "UVM User Guide, v.1.1" http://www.accellera.org/downloads/standards/uvm

[2] Accelera, "Standard UVM Class Reference, v1.2" http://www.accellera.org/downloads/standards/uvm

[3] "UVM Harness Whitepaper", David Larson, Synapse Design, http://www.synapse-da.com/Uploads/PDFFiles/03_UVM-Harness.pdf

[4] "Parameterized Classes, Interfaces & Registers" , Mark Litterick, Verilab, DVCon Europe 2015, http://www.verilab.com/files/verilab_dvcon_eu2015_6_params.pdf