Perplexing Parameter Permutation Problems?

Immunize Your Testbench!

Alex Melikian
Verilab Canada Inc.

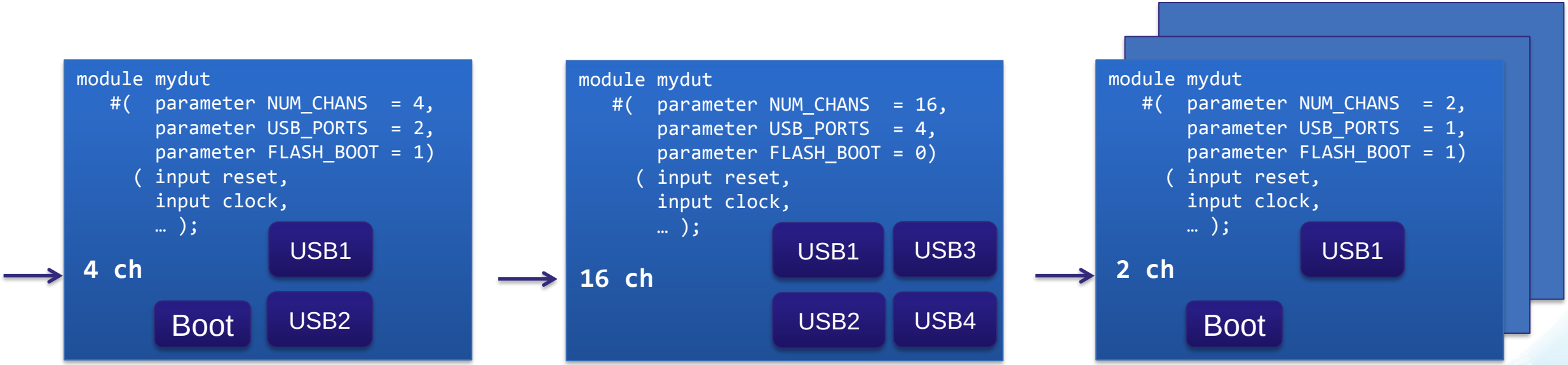
April 21, 2017
Canada

Agenda

- Perplexing Permutations from RTL Parameters
- Proposed Solution of Capturing Parameters
- Randomization with Captured Parameters in TestBench
- Solutions with UVM Register Modeling
- Conclusions

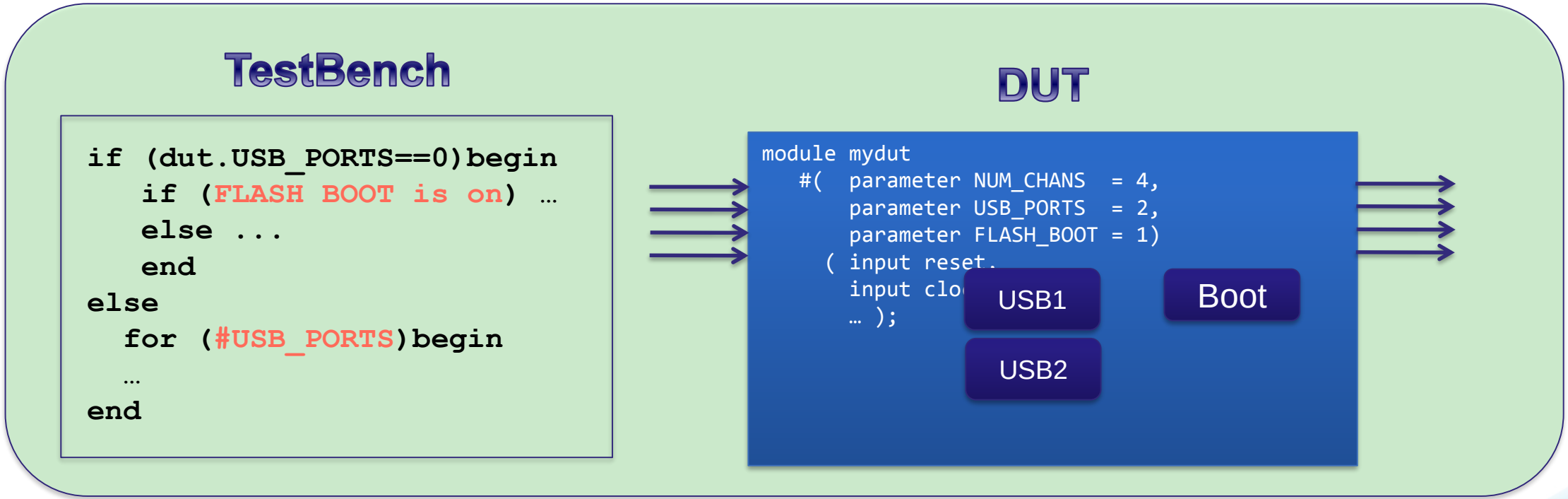
Perplexing Permutations

- IPs have LOTS of things that can be configured using RTL parameters
  - Multiplicity of instances or sizes
    - Number of ports, channels, address/data bits
  - Optional or swap-able sub-blocks
    - e.g. for power or cherry-picking feature sets
- Number of *valid* combinations can rapidly become enormous



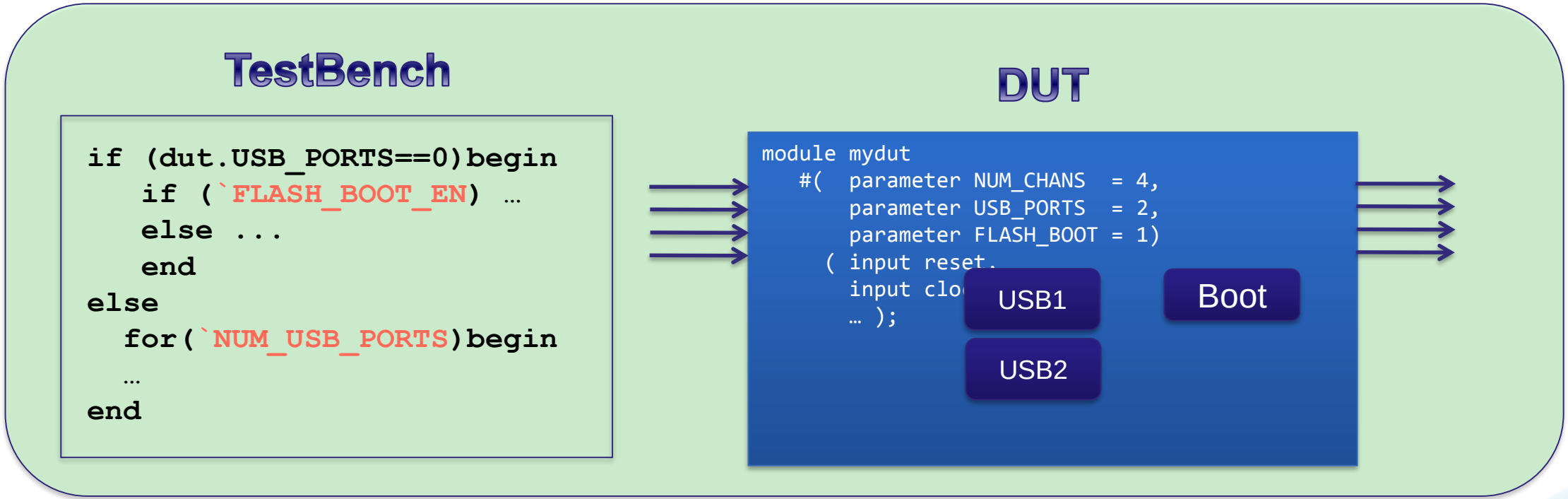
Perplexing Permutations

- RTL parameters cause verification engineers to be perplexed!
  - Testbench must match all (valid) parameter permutations
  - An RTL bug may not manifest in one RTL configuration, but can in another



Perplexing Permutations

- Traditional approach is to use `defines to reflect DUT parameter values
  - Makes the testbench more parametrizable, but can be unwieldy
  - Macros are syntactically poor
    - Can introduce subtle, difficult to find bugs in the testbench



Proposed Solution

- Two main principles …
- Mechanism to automatically extract and capture RTL parameters for the testbench
  - Avoids proliferation of macros in testbench code
  - Simplifies merging of functional coverage over different parameter sets

- Use extracted RTL parameters to make testbench flexible
  - Leads to a single compilation of TB
  - Saves compute time
  - More readable testbench

Capturing RTL Parameters

- Use of specially bound SystemVerilog Interface
  - Interface contains all of the RTL parameters with matching names
  - Captures and makes them available via UVM configuration database using a `struct`

```
interface tb_binding_to_mydut_iface #(
       parameter MYDUT_PARAM1 = 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
```

Matching names of DUT's RTL parameters

Struct or uvm_object reflecting RTL parameter values

Method capturing values into struct and put into config db

Capturing RTL Parameters

- Interface is bound to the DUT via the SystemVerilog `bind` statement
  - RTL parameter values are mapped on instantiation

```
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(.*)
```

Bind statement of i/f will capture DUT RTL parameters

```
tb_binding_to_dut_iface(
  .MYDUT_PARAM1(MYDUT_PARAM1)
  .MYDUT_PARAM2(MYDUT_PARAM2)
  .MYDUT_PARAM2(MYDUT_PARAM3))
   // reset,
   // clock …
tb_binding_to_dut_harness(.*)
```

```
mydut mydut_inst
   #(  MYDUT_PARAM1( 2 ),
       MYDUT_PARAM2( 1'b0 ),
       MYDUT_PARAM3( 8 )
     ( .reset(rst),
       .clock(clk),
       … );
```

Capturing RTL Parameters

- Interface is bound to the DUT via the SystemVerilog `bind` statement
  - RTL parameter values are mapped on instantiation
  - Capture method call made before testbench is created

```
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(.*)
```

Bind statement of RTL parameter capturing i/f

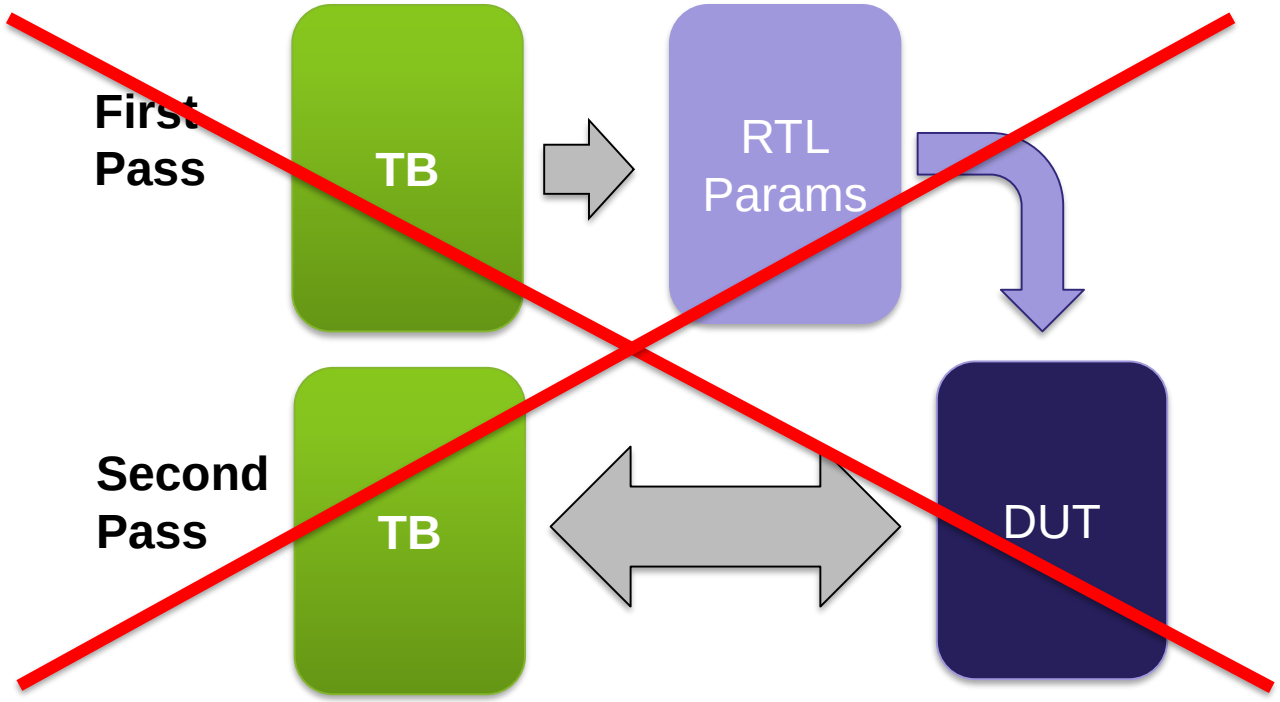
```
module tb_top;
  initial
    begin
      tb_binding_to_dut_harness.capture_rtl_info( env.get_full_name());
      run_test();
    end
endmodule
```

Capture method called in tb_top initial block …

… before UVM 'run_test' call

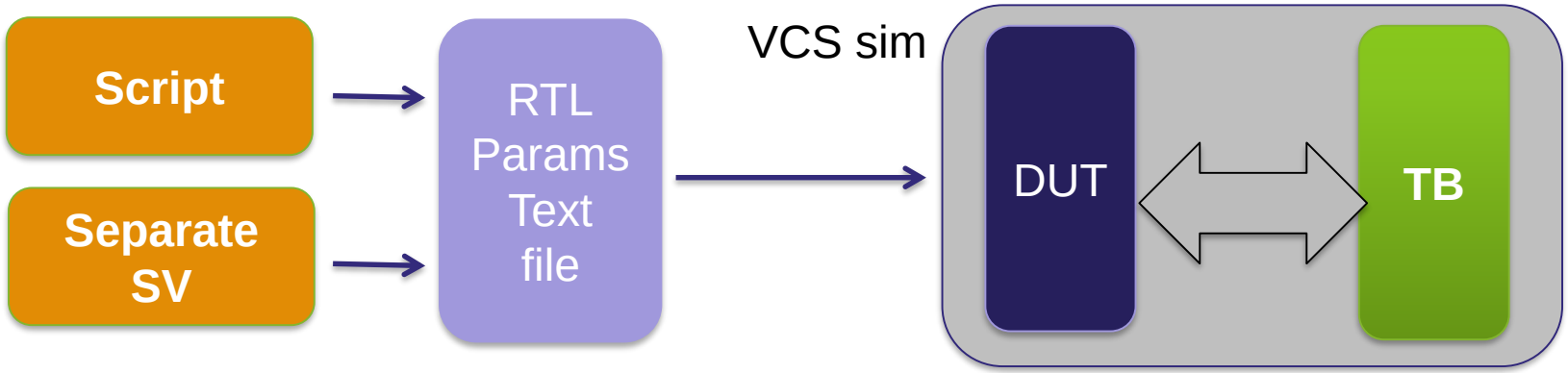
Capturing Parameters

- Advantages:
  - Eliminates use of `defines
  - Eliminates need for two-pass randomization
    - No need to preserve randomization seeds etc.



Capturing Parameters

- Advantages:
  - Parameter file to be loaded can be generated independently
    - All that is necessary is a means to generate the actual parameter values to be applied
    - These can either be passed on the command line or read from a file



- Disadvantages
  - Creating an interface to encapsulate the parameters used
  - Manual maintenance of the interface to DUT parameters in the `bind` instance

Decoupled Randomization

- Interface/bind method removed the need for two passes
  - No need to run a first TB pass to generate randomized RTL parameter values
  - Bound interface and struct provides parameter values for randomization

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

  constraint port_num_constr {
    (rand_port_number>=0) && (rand_port_number<dut_rtl_param_info.number_of_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 …")
   endfunction
endclass
```

Extracted parameter value used in constraint

Get RTL parameter struct from config db at construction

Parameters on UVM Reg Modeling

- RTL parameters often used to specify register attributes
  - E.g. reset values
- Apply register inclusion / exclusion with DUT
  - An excluded register or field becomes ‘reserved’ but still requires to be tested
- Affect Register Field Sizes
  - i.e Register field reflecting channel enables, varying with parameter settings affecting number of channels

UVM Register Modeling

- Register reset values

```
rtl_info_struct_t  dut_rtl_param_info;
uvm_reg_fields     dut_reg_fields[$];
        [...]
  if (!uvm_config_db#(rtl_info_struct_t)::get(null, "*", "rtl_info_struct", dut_rtl_param_info))
          `uvm_fatal(…)

   dut_regblock.featureA_reg.field1.set_reset(dut_rtl_param_info.mydut_featA_regfeild1_resetval);

   dut_regblock.featureA_reg.field1.reset();
```

Get RTL parameter struct as usual via config db

Struct contains register reset value to reflect RTL value

Extracted parameter value applied when the register is reset

- Code to be executed after build stage, but before checks run (end_of_elaboration)

UVM Register Modeling

- Register removal, turned into a ‘reserved’ address
  - A removed register or field is still required to be tested

```
if (!uvm_config_db#(rtl_info_struct_t)::get(null, "*",
                                            "rtl_info_struct", dut_rtl_param_info))
   uvm_fatal(…)

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
```

Get RTL parameter struct as usual via config db

Check to see if feature was included by parameter

Conditionally set the fields as read-only

UVM Register Modeling

- Register Field Sizes
  - These cannot be changed dynamically in reg model
    - But clever use of factory overrides provides solution

```
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),
            .lsb_pos (0),
            .access  ("RW"),
            [...]
        endfunction
endclass
```

Same size regardless of parameters

UVM Register Modeling

- Register Field Sizes
  - Use virtual method to return the size parameter

```
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    (get_channel_en_size(),
            .lsb_pos (0),
            .access  ("RW"),
            [...]
        endfunction
        virtual function int get_channel_en_size();
           return 8;
        endfunction
endclass
```

Use method to return value

Simple method body can be overridden

UVM Register Modeling

- Register Field Sizes
  - Create extended class with call to the config_db to return the size parameter
  - Then use factory override to use this class

```
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

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

Gets parameter value from the config db

Factory override

Register Modeling – Overriding Concerns

- Dealing with multiple instances for overrides
  - multiple instances of the same register
- For example
  - Data Port 0 has 5 channel enables
  - Data Port 1 has 3 channel enables
  - Data Port 2 has 8 channel enables
- You can create an overriding class along with override call for each instance
  - But this leads to code bloat!
- Better to use introspection in the overriding class itself to self-ascertain instance index
- Make use of uvm name "`dut_channelen_port[n]`" to get index

Register Modeling – Overriding Concerns

```
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
     if (!uvm_config_db#(rtl_info_struct_t)::get(…, dut_rtl_param_info)

     portnum = extract_portnum_from_name( get_name() );
     return dut_rtl_param_info.featureA_number_of_channels[portnum];
   endfunction
endclass
```

Get RTL parameter struct like usual via config db

Extract index # from uvm name (ie "dut_channel_en[3]"

Use index to self-ascertain correct parameter value

- Will only require one, tidy call to `set_type_override_by_type( )`

Conclusions

- Used in a real world project
  - 50 RTL Parameters in an IP
  - 1500 Individual coverage bins
  - 100 RTL compilation snapshots
    - (Pairwise/N-wise testing serves well here – Paper from Verilab.com)
  - Only one compilation of the TB source code required
  - Coverage of RTL parameters becomes very easy with struct in config db
    - May be important to cover specific monitored stimulus vs. specific RTL params
  - Code bloat was minimal and insignificant

verilab snug Synopsys Users Group Canada

Thank You

verilab.com
@verilab