UVM Register Map Dynamic Configuration

Matteo Barbati, STMicroelectronics, Digital Mixed Asic Division, Via Tolomeo 1, Cornaredo (Milano), Italy (*matteo.barbati@st.com*)

Alberto Allara, STMicroelectronics, Digital Mixed Asic Division, Via Tolomeo 1, Cornaredo (Milano), Italy (*alberto.allara@st.com*)

***Short abstract*—The UVM register map provides data structures and testsuites that allow checking the data integrity of the DUT register map during simulations. This object can be extended to allow the synchronization between the register model and any verification component present in the environment. Beside ad-hoc approaches based on code customization or custom code generation from IP-XACT, we present a solution that simplifies and automatizes such interaction.**

I. INTRODUCTION

One of the activities required for both IP and SoC Verification is the Register Map Verification. UVM [1][2] provides a dedicated infrastructure, called *UVM register map*, to allow verification engineers to perform Register Map verification and to build complex verification environments. This data structure supports the checking of data integrity on the DUT register map at every single access.

The UVM register map provides an abstraction layer that allows access to single register fields through logical names instead of portions of the physical register [3]. In addition, it provides a set of callbacks to adapt the behavior of the verification component to the needs of the verification engineers. A typical usage of the callbacks is the possibility to trigger, for instance, an event each time a register or a bitfield is accessed. Once that an event is triggered, any verification component or scoreboard in the environment may synchronize itself with the updated contents. This mechanism allows the creation of complex environments, whose behavior is dynamic and follows the register map configuration of the DUT.

Figure 1 depicts the topology of a generic UVM register map.

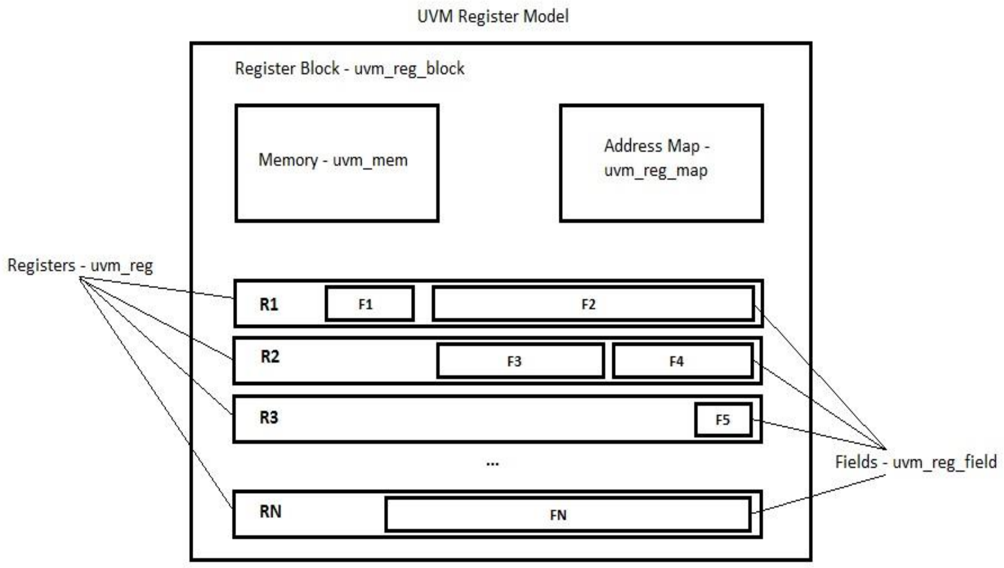


Figure 1: UVM register model topology

The root of the register map is represented by the *uvm_reg_block* class, which describes the overall register map structure. This object is mapped to a specific address through *uvm_reg_map* class. The register block class instantiates a set of *uvm_reg* objects modelling the registers of the actual Register map. Each *uvm_reg* is composed of a set of *uvm_reg_field,* the leaf objects used to model bitfields in the Register map. In addition, *uvm_reg_block* can also instantiate a memory model.

We can customize our UVM register maps to add new functionalities, thus creating more complex and flexible verification environments. The steps required to add such functionalities are:

- Specialize ad-hoc the class(es) of the register(s) that need these extra functionalities (for example the possibility to trigger a particular event) redefining the implementation of one of the dedicated callbacks available in the *uvm_reg* class (pre_read, pre_write, post_read and/or post_write)
- Replace the old class(es) with the new one through a UVM factory mechanism.

The piece of code in Figure 2 is a possible implementation of the steps described above for a register called MODE. We declare a uvm_event called "mode_type_e" which is triggered every time a Write Access is performed on MODE register.

```
class trig_MODE_type extends MODE_type;
  `uvm_register_cb(trig_MODE_type, uvm_reg_cbs)
  `uvm_set_super_type(trig_MODE_type, uvm_reg)
  `uvm_object_utils(trig_MODE_type)

  uvm_event_pool ep;
  uvm_event e;
  string event_name = "mode_type_e";

  rand uvm_reg_field field1;
…
  rand uvm_reg_field field4;


  function new(string name = "unnamed-trig_MODE_type");
    super.new(name);
    ep = uvm_event_pool::get_global_pool();
    e  = ep.get(event_name);
  endfunction

  virtual function void build();
     super.build();
     …
  endfunction

  virtual task post_write(uvm_reg_item    rw);
    `uvm_info("trig_MODE_type", "**** Firing Write trigger ****", UVM_HIGH);
    e.trigger();
  endtask: post_write
endclass

…
```

```
set_type_override_by_type(MODE_type::get_type(),trig_MODE_type::get_type());
```

Figure 2: An example of manual extension of the register map

This approach is efficient only when few registers require modifications. As the number of customized registers increases, it becomes time consuming and potentially error-prone. In particular, if N is the number of registers requiring the event generation, we need to write N times the same portion of code highlighted in Figure 2. Under those circumstances, any flow dedicated to automating the process is beneficial.

On this topic, CAD Vendors typically suggest the use of flows based on the IP-XACT [4] description of the DUT register map. Each of these flows, using a set of vendor-specific tags and tools, starts from an XML description of the register map, and generates the UVM register model code.

With such flows, if a verification engineer needs to extend the features of the register map, he/she has to modify the IP-XACT description of the DUT register map by manually adding vendor-specific IP-XACT tags. For example, if he/she wants to trigger an event after a register access, it is necessary to place a vendor specific tag in the IP-XACT register description, which embeds the UVM code needed to handle the user-defined event.

The following snippet shows the IP-XACT description of the MODE register from our previous example customized for Cadence iRegGen flow. In this representation dedicated IP-XACT tags (vendor specific) are added to allow the generation of an event once a write is performed on the register

```
<spirit:register>
  <spirit:name>MODE</spirit:name>
   …
   <spirit:field>
        <spirit:name>field1</spirit:name>
                …
   </spirit:field>
   …
   <spirit:field>
        <spirit:name>field4</spirit:name>
        …
   </spirit:field>
   <spirit:vendorExtensions>
        <vendorExtensions:rawCode>
        <vendorExtensions:source domain="SystemVerilog">
                       <vendorExtensions:description>My post_write
                       </vendorExtensions:description>
                       <vendorExtensions:code>
                         virtual task post_write(uvm_reg_item rw);
                           uvm_event_pool ep;
                           uvm_event e;
                           `uvm_info("trig_MODE_type","**Firing Write trigger**",UVM_HIGH);
                           ep=uvm_event_pool::get_global_pool();
                            e  = ep.get("mode_type_e");
                            e.trigger();
                         endtask: post_write
                       </vendorExtensions:code>
               </vendorExtensions:source>
        </vendorExtensions:rawCode>
    </spirit:vendorExtensions>
</spirit:register>
```

Figure 3: MODE register XML description

The resulting UVM register model generated by Cadence iRegGen flow is reported in Figure 4

```
class MODE_type extends uvm_reg;

  rand uvm_reg_field field1;
…
  rand uvm_reg_field field4;
…
  virtual task post_write(uvm_reg_item    rw);
    uvm_event_pool ep;
    uvm_event e;
    `uvm_info("trig_MODE_type", "** Firing Write trigger **", UVM_HIGH);
    ep = uvm_event_pool::get_global_pool();
    e  = ep.get(event_name);
    e.trigger();
  endtask: post_write

  `uvm_register_cb(MODE_type, uvm_reg_cbs)
  `uvm_set_super_type(MODE_type, uvm_reg)
  `uvm_object_utils(MODE_type)
  function new(input string name="unnamed-MODE_type");
    super.new(name, 16, UVM_NO_COVERAGE);
  endfunction : new
endclass : MODE_type
```

Figure 4: MODE_type class plus vendor_extension code

Also with this approach, if N registers require an event each, the same chunk of code needs to be replicated N times, as an IP-XACT vendor extension. The advantage of this approach is that extra features are present only in those registers that really need them. The major drawback is that the verification engineer usually is not the owner of the IP-XACT file and adding verification-oriented extensions to it can be a problem. In addition, in several cases, the IP-XACT is a file provided by a third party actor and there are organizations that in their flow explicitly forbid modifying and using customized version of the same. In such cases, a Verification version of the IP-XACT file is created in addition to the original one, leading to possible maintenance issues that potentially can create a misalignment between the two versions.

II. NOVEL APPROACH

Instead of working on the IP-XACT description as suggested by CAD Vendors, our solution focuses directly on the UVM register model data structure and in particular, on the leaf objects of the UVM regmap hierarchy represented by the *uvm_reg_field* class.

The idea is to extend the functionality provided by this class by adding the generation of the desired events and overriding the original *uvm_reg_field* class with the new extended version of the class. In this case, we replace the entire register map fields and in order to avoid the generation of a huge number of unwanted events, a mechanism is required to enable the events generation only where they are really needed.

UVM already provides a mechanism to enable/disable knobs through its internal database *uvm_config_db*. This mechanism is automatically managed in the *uvm_component* class and in all its specializations. Since a UVM register map is based on *uvm_object*, rather than *uvm_component*, it does not support natively such mechanism so it is necessary to add this extra functionality. The solution we have implemented makes use of UVM regmap callbacks. In particular, our solution is based on the use of:

- pre_read() and pre_write() callbacks;
- a "lazy initialization" design pattern to evaluate the *uvm_config_db* only on the first call of the above callbacks to emulate the behavior of the configuration mechanism available in *uvm_component*.

The following piece of code shows the implementation of the uvm_reg_field_ext class

```
class uvm_reg_field_ext extends uvm_reg_field;
   `uvm_object_utils(uvm_reg_field_ext)

  uvm_event_pool ep;
  uvm_event pre_rd_e, pre_wr_e, post_rd_e, post_wr_e;
  local string pre_rd_event_name; local string pre_wr_event_name;
  local string post_rd_event_name; local string post_wr_event_name;
  …
  local bit conf;

  function new(string name = "uvm_reg_field_ext");
    super.new(name);
    ep = uvm_event_pool::get_global_pool();
    conf = 1'b1;
    pre_rd_event_name = {"pre_rd_", get_name()};
    pre_wr_event_name = {"pre_wr_", get_name()};
    post_rd_event_name = {"post_rd_", get_name()};
    post_wr_event_name = {"post_wr_", get_name()};
  endfunction

  virtual task pre_read(uvm_reg_item    rw);
    if(conf) begin
      uvm_config_db#(bit)::get(uvm_root::get(), get_full_name(), "en_pre_rd_event", en_pre_rd_event);
      uvm_config_db#(bit)::get(uvm_root::get(), get_full_name(), "en_pre_wr_event", en_pre_wr_event);
      uvm_config_db#(bit)::get(uvm_root::get(), get_full_name(), "en_post_rd_event", en_post_rd_event);
      uvm_config_db#(bit)::get(uvm_root::get(), get_full_name(), "en_post_wr_event", en_post_wr_event);
      conf = 1'b0;
    end
    if(en_pre_rd_event) begin
      `uvm_info(get_name(), "**** Firing Pre Read trigger ****", UVM_HIGH)
      pre_rd_e.trigger();
    end
  endtask : pre_read
```

```
    virtual task pre_write(uvm_reg_item   rw);
      if(conf) begin
        uvm_config_db#(bit)::get(uvm_root::get(), get_full_name(), "en_pre_rd_event", en_pre_rd_event);
        uvm_config_db#(bit)::get(uvm_root::get(), get_full_name(), "en_pre_wr_event", en_pre_wr_event);
        uvm_config_db#(bit)::get(uvm_root::get(), get_full_name(), "en_post_rd_event", en_post_rd_event);
        uvm_config_db#(bit)::get(uvm_root::get(), get_full_name(), "en_post_wr_event", en_post_wr_event);
        conf = 1'b0;
      end
      if(en_pre_wr_event) begin
        `uvm_info(get_name(), "**** Firing Pre Write trigger ****", UVM_HIGH)
        pre_wr_e.trigger();
      end
    endtask : pre_write

    virtual task post_read(uvm_reg_item   rw);
      if(en_post_rd_event) begin
        `uvm_info(get_name(), "**** Firing Post Read trigger ****", UVM_HIGH)
        post_rd_e.trigger();
      end
    endtask : post_read

    virtual task post_write(uvm_reg_item  rw);
      if(en_post_wr_event) begin
        `uvm_info(get_name(), "**** Firing Post Write trigger ****", UVM_HIGH)
        post_wr_e.trigger();
      end
    endtask : post_write
  endclass: uvm_reg_field_ext
```

Figure 5: uvm_reg_field_ext class

In the pre_read() and pre_write() methods, on the very first access of each single field, we query the *uvm_config_db* to get the status of the knob controlling the event generation for that field; then, according to these settings, events are generated inside the regmap callbacks and can be used in the rest of the verification environment.

To be activated, this approach requires a factory override of the *uvm_reg_field* class with the new extended version. In addition, it is required to turn on the conditional event generation on the desired fields. Figure 6 shows an example where we configure three non-consecutive fields of our MODE register to generate an event after a write on those fields:

```
set_type_override_by_type(uvm_reg_field::get_type(),uvm_reg_field_ext::get_type());
uvm_config_db#(bit)::set(uvm_root::get(), "regmap.MODE.field1", "en_post_wr_event", 1);
uvm_config_db#(bit)::set(uvm_root::get(), "regmap.MODE.field3", "en_post_wr_event", 1);
uvm_config_db#(bit)::set(uvm_root::get(), "regmap.MODE.field4", "en_post_wr_event", 1);
```

Figure 6: configuration of uvm_reg_field_ext objects in top level env

The previous approach is the result of a study and analysis process that tries to find the most straightforward way to add a configuration mechanism to the uvm_reg_field class. In this analysis, we tried also another plausible strategy where we added the desired configuration features by means of the UVM register map setup functions.

We found this approach limited by the fact that *uvm_objects* do not provide the phasing mechanism available in the *uvm_components*. For this reason the usage of the functions *build* and *reset* is not fine. Even the usage of function new is not fine, since the field name is not available when the function new is called. The main issues related to functions build and reset are:

- The *build* function is not virtual, so it is not possible to replace this function using a factory override;
- The event configurations have to appear, in the code, before the build/reset function calls. The configurations that follow the build/reset function calls are ignored.

Our solution is affected by an overhead for each uvm_reg_field , in terms of size,w.r.t. to original solution without any modification This overhead is summarized in the following list:

- Number of required variables : 9
    - 1 bit to use to put in place the lazy initialization mechanism;
    - 1 bit used to manage the enable/disable of the event generation for 4 callbacks (pre_rd, pre_wr, post_rd and post_wr);
    - 4 strings used to store the event name.
- Number of events : 4

The pros and cons of the approaches presented in this paper are summarized in Table 1

	IP-XACT flow	Custom Code	Novel Approach
Pros	• Code optimized: events generated only where needed	• Code optimized: events generated only where needed	• More flexibility: Event generation can be dynamically configured • Reduce code impact in case of customization needed on high number of registers
Cons	• Error prone in case of high number of registers to customize • Need to rerun the entire flow in case of changes in register customization • Different version of the same IP-XACT description. One for Design and one for Verification	• Error prone in case of high number of registers to customize	• Size Overhead affecting all the fields of the entire register map:

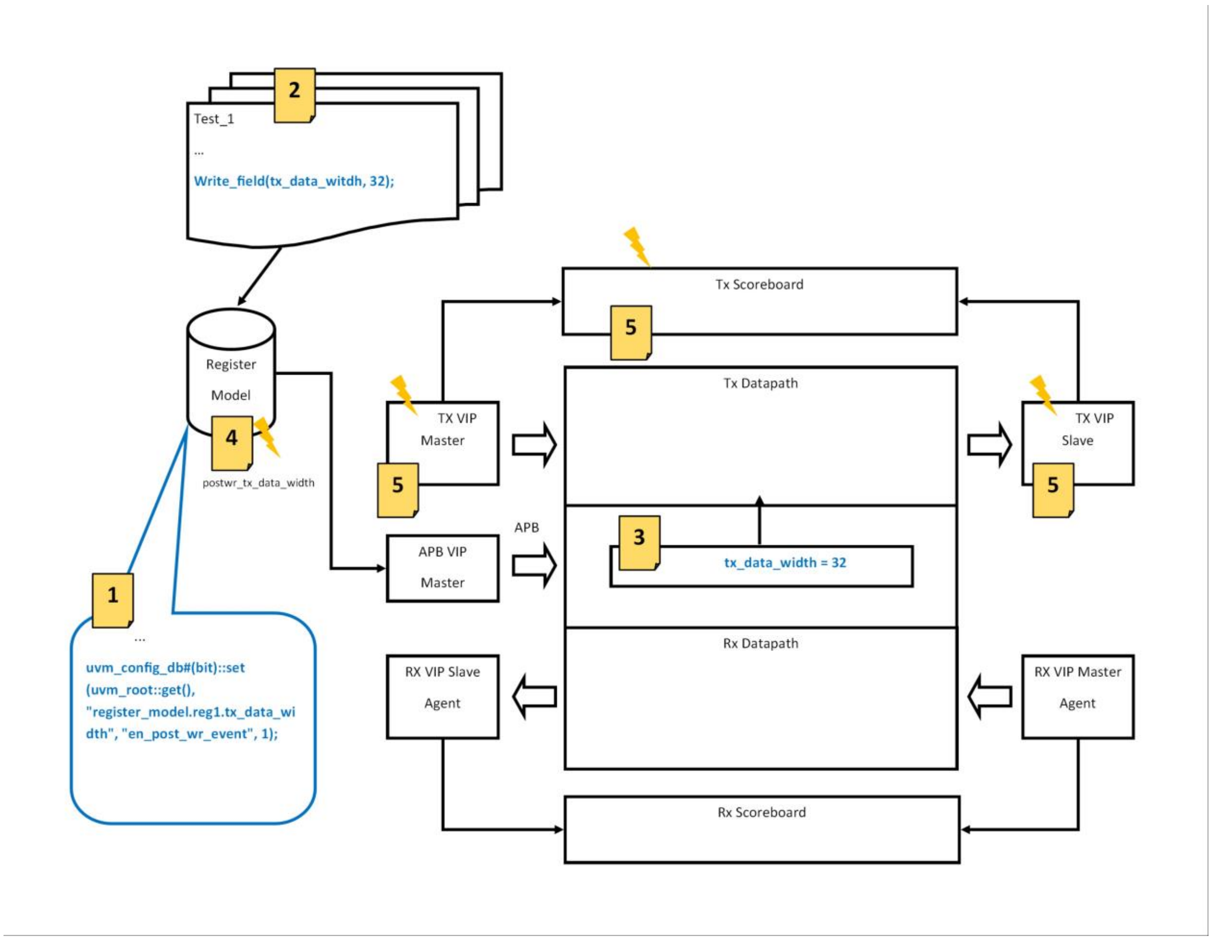
Table 1 - Pros and Cons recap

III. USE CASE

The proposed approach is currently used inside our verification environments. Our DUT is a SerDes IP, based on two separated and configurable datapaths. One dedicated to the transmitter logic and the other one dedicated to the receiver logic. These paths can be configured to work at different data rates (up to 10 Gbs), data widths (from 8 bits to 80 bits) and data codes (PAM4 or NRZ). The configuration of each single datapath is controlled through a set of registers of the register map.

Due to dynamically configurable nature of the DUT the approach we have proposed allow us to easily updated our Verification environment and Scenarios without the need to update frequently neither the Verification version of IP-XACT register map nor the Custom code to add synchronization functionalities to the UVM register map.

The main flow of our approach, currently in use can be summarized in Figure 7. The picture shows an ideal flow associated to the configuration of the Tx Data Width register variable.



```
...
Write_field(tx_data_witdh, 32);
```

```
...
uvm_config_db#(bit)::set
(uvm_root::get(),
"register_model.reg1.tx_data_wi
dth", "en_post_wr_event", 1);
```

Figure 7 - Tx Data Width events flow

1. Inside the Top Level Verification Environment we turn-on the event generation related through uvm_config_db. In this way we configure the register model to generated an event when a write is performend on field ”tx_data_width”;

2. The virtual sequence associated with “Test_1” performs a write on field “tx_data_width” to configure the size of the data stream equal to 32. The register model manage this write field operation and, through a register adapter, it instructs the APB VIP to perform a Write operation inside the DUT, to write tx_data_width to 32;

3. DUT Register field “tx_data_width” is updated with the new value equal to 32. According to this configuration the behavior of Tx Datapath changes to manage the new data width.

4. In the meanwhile the Register model generates an event to trigger the rest of the verification environment about the change of the tx_data_width field.

5. The event generated at step 4 triggers the Tx VIPs and the Tx Scoreboard about the change in field tx_data_width. The two VIPs and the Tx Scoreboard retrieve the information related to the new value of the field and update their behavior according to the new configuration.

This approach is used in general for all the fields related to data with, data rate and data code used in the Tx and Rx datapaths of our DUT.

Within our Verification environment, the same approach is used also to create a clock_predictor verification component. This VIP, based on register configurations, computes and checks the frequencies of internal clocks of the device.

IV. CONCLUSION AND FUTURE WORKS

UVM register map provides data structures and testsuites that allow checking the data integrity of the register map DUT during simulations. The Register model can be extended as required by verification needs to add capabilities to synchronize the content of the register map with the rest of the verification environment.

This extension mechanism can be both manual, modifying the classes instantiated inside the UVM register map, or automatic according to the suggested CAD Vendor flows by modifying the IP-XACT description of the Register map. In addition to these two approaches, a novel approach has been proposed in which the leaf of the register map data structure, represented by *uvm_reg_field*, is extended to add the new required functionalities to facilitate the synchronization with any verification components present in the environment.

We are working on a potential extension of the proposed solution that allow to create a cluster of events to reduce the number of events generated by a set of fields.

ACKNOWLEDGMENT

We would like to thanks and acknowledge Andrea Masi, Rinaldo Franco, Massimo Vincenzi and Matteo Ferranti for their contributions to the technical implementation of the uvm_reg_field configuration mechanism.

REFERENCES

[1] Accellera, “UVM User Guide, v1.1”, www.uvmworld.org

[2] Accellera, “UVM Reference Guide, v1.1d” , www.uvmworld.org

[3] Mark Litterick, Marcus Harnisch, “Advanced UVM Register Modeling There’s More Than One Way to Skin A Reg”, DVCon 2014

[4] IPXACT XML standard, http://standards.ieee.org/findstds/standard/1685-2009.html