Register Access By Intent: Towards Generative RAL Based Algorithms

Ahmed M. Allam, ICpedia, Cairo, Egypt (*ahmed.allam@icpedia.com*)

Abstract: RAL is a powerful tool used in various verification methodologies. It can abstract and hide information about DUT registers and provide a set of APIs to access those registers regardless of details about physical addresses, reset values, and BFM used for physical access of DUT registers. Although the design may change registers addresses from release to release, RAL can still be regenerated and completely hides these changes from testcases. This makes the verification environment stable and adaptive to design register changes. However, registers may change their addresses and their names and fields can also be migrated from one register to another. In this case, we must get new register names and field locations and update the test cases accordingly. This challenge lets one think about extending this concept and adding more abstraction and hiding capability to RAL. This paper proposes a novel methodology called *META-RAL*, where fields will be not only accessed by their name but also by functions and other features such as scope and association to other fields through Implemented lookups. In this case, scrambling of the registers and their fields would not affect the verification environment.

Furthermore, this methodology is used to develop what is termed Multi-View RAL (MV-RAL), allowing RAL to be accessed through various views based on the set of target registers for each test. While MV-RAL is a robust methodology, it also helps optimize the developed lookups so it does not impact simulation performance. Finally, this approach helps to develop generic register-based algorithms that can be applied to the DUT independent of any register and field name details. These algorithms can be implemented as a separate sequence or embedded as a novel approach inside the RAL class.

I. INTRODUCTION

Among UVM reference classes, UVM_RAL is one of the most powerful tools that plays a significant role in DUT verification. Ensuring chip configuration and status registers are written/read properly is crucial in the chip verification cycle[1]. Although the design may change register addresses from release to release, RAL can still be regenerated and completely hides these changes from test cases. This makes the verification environment stable and adaptive to design register changes. However, registers may change their addresses and names, and fields can be migrated from one register to another. In this case, we must get new register names and field locations and update test cases accordingly. This challenge lets one consider extending this concept and adding more abstraction and hiding capability to RAL.

The remainder of the paper is organized as follows. Section II presents the previous work done to improve RAL. Section III proposes the ***META-RAL*** approach for accessing registers by function and scope. Then, it shows how this can be used to enhance simulation performance and verification quality. Section IV shows how this architecture is useful with a case study in register testing by scope. Section V proposed a novel mechanism to implement RAL based sequences using the presented approach and a case study of the SerDes application. And finally, section VI concludes the paper.

II. RELATED WORK

Industrial experts have attempted to improve RAL due to its significant importance in the current ASIC industry. Advanced modeling for RAL is studied and presented in [2]. Another contribution in [3] focuses on automatically generating RAL sequences from RTL registers. In [4], some work is done on constrained randomization for registers to match DUT constraints. Functional coverage sampling for registers gains interest in [5] [6]. Enhancing the RAL structure will significantly enhance the ASIC verification flow. An internal analysis for UVM register internal implementation for front door access against high performance system is addressed in [7]. In [8], some aspects of

UVM-RAL are discussed, such as front door sequences and predictors. In [9], using backdoor access to initialize DUT states is explored to reduce simulation time with dynamic control of HDL path using callbacks.

III. META-RAL: A Flexible RAL architecture

While the current RAL model is considered very powerful for hiding DUT register details, it still misses some features that are worth exploration:

1. Testcases need updates when fields are migrated from one register to another, or register/field names change. Also, grouping registers or accessing register fields independent of their parent register would require additional coding in RAL classes. Some of these workarounds can be as follows:

    a. Register field objects can be created at the ***uvm_reg_block*** level and assigned to ***uvm_reg_blk.fld***

    b. Additional classes with queues need to be implemented and fields can be pushed back to these queues.

However, both methods still suffer from the same object-naming problem. Therefore, adding some property uniquely identifying a register field will be useful. A property such as a field function can be assigned to the field wherever it is in the RAL hierarchy. Therefore, this section introduces an extension for ***uvm_reg*** and ***uvm_reg_field*** implementation to track the register fields with a distinguished tag identifier representing the register field's function inside the DUT.

2. The RAL model is built as one inseparable unit. Once locked, the register structure cannot be changed. In large systems, this would impact simulation performance and system memory.

In this section, three main proposals are introduced to tackle the previous limitations and target generic RAL based algorithms:

1. Introduce a new RAL architecture to access RAL fields using different attributes regardless of the register structure.
2. Build the RAL model according to test purpose, which selects the RAL view.
3. Separate RAL sequence from physical DUT details to fit with different IPs and RTL releases.

A. Field access by new attributes (function,scope,associativity)

In order to add more flexibility to RAL, we aim to add some other lookups as follows:

1. **m_reg_scopes_registry[**scope_e**]:** This lookup stores handles of registers that belong to a specific scope.
2. **m_reg_field_scopes_registry[**scope_e]: This lookup handles of fields that belong to a specific scope.
3. **m_reg_field_func_registry[**tag_e**]:** This lookup stores the field's handle with its tag as the index.
4. **m_reg_field_assoc_registry**[tag_e][assoc_e]**:** This lookup store handles of two different fields by establishing a relationship between them, for instance, "*field_A*," with a specific associative property with *field_B*. The fields "*field_ovrd_val*" and "*field_ovrd_en*" may be found in different registers.
5. **m_reg_assoc_registry[**string**][**assoc_e**]:** This lookup stores the handle of *regB* which is related to *regA* by a specific associative property.

6. **m_reg_field_name_scopes_registry**[tag_e]: This lookup stores the names of scopes to which a certain field belongs.
7. **m_reg_name_scopes_registry**[string]: This lookup stores the names of scopes to which a certain register belongs.

There are two options to develop the new function tag:

1. Reuse the field ***name*** property and assign it a unique name irrelevant to the field class name.
2. Add new property, ***func_tag***, and use this in addition to the ***name*** property.

Considering the ***uvm-1.2*** implementation, to search for a field independent of its parent register, one can use the **uvm_reg_block::get_field_by_name** function, which iterates over all block registers, sub-blocks, and sub-registers, which would have significant undesired performance degradation. For this reason, the ***func_tag*** property is added, and *associative-array* based lookups are being developed to retrieve the specified field without looping through the entire block fields' space. The lookups are indexed by ***enumeration*** types to make a strictly typed index and catch any inconsistency between testcases and the generated RAL structure. Any mismatch would result in a compilation error. By employing those lookups, DUT register testing becomes more powerful, flexible, and adaptive to changes in DUT register structure. Fig.1 and Fig.2 show META-RAL lookups and their helping functions, while Fig.3 and Fig.4 show ***META-RAL*** sequence register read using ***func_tag*** and associative properties, respectively.

```
class uvm_meta_reg_block extends uvm_reg_block;

  uvm_reg m_reg_scopes_registry[scope_e][$];
  uvm_reg_field m_reg_field_scopes_registry[scope_e][$];
  uvm_reg_field  m_reg_field_func_registry[tag_e];
  uvm_reg_field m_reg_field_assoc_registry[tag_e][assoc_e];
  uvm_reg  m_reg_assoc_registry[string][assoc_e];
  scope_e m_reg_field_name_scopes_registry[tag_e][$];
  scope_e m_reg_name_scopes_registry[string][$];

  scope_e ral_view_scopes[ral_view_e][$];
  ral_view_e ral_view;
////////////////////////////////////////////////////////////////////////////////
function tag_field_by_function(uvm_reg_field fld, tag_e func_tag);
  m_reg_field_func_registry[func_tag] = fld;
endfunction
////////////////////////////////////////////////////////////////////////////////
function uvm_reg_field get_field_by_func(tag_e func_tag);
  return m_reg_field_func_registry[func_tag];
endfunction
////////////////////////////////////////////////////////////////////////////////
function add_field_scope(scope_e scope, tag_e func_tag);
  uvm_reg_field fld;
  fld =  m_reg_field_func_registry[func_tag];
  m_reg_field_scopes_registry[scope].push_front(fld);
  m_reg_field_name_scopes_registry[func_tag].push_front(scope);
endfunction
////////////////////////////////////////////////////////////////////
function bit check_reg_build(ral_view_e ral_view, string reg_name);
  foreach(m_reg_name_scopes_registry[reg_name][i]) begin
    if(m_reg_name_scopes_registry[reg_name][i] inside {ral_view_scopes[ral_view]})
      return 1;
  end
return 0;
endfunction
////////////////////////////////////////////////////////////////////////////////
```

Fig.1 META-RAL methodology implementation

```
function add_reg_name_scope(scope_e scope,  string reg_name);

  m_reg_name_scopes_registry[reg_name].push_front(scope);
endfunction
////////////////////////////////////////////////////////////////////
function get_regs_by_scope(scope_e scope, ref uvm_reg regs[$]);
  foreach(m_reg_scopes_registry[scope,i])begin
    regs.push_front(m_reg_scopes_registry[scope][i]);
  end
endfunction
////////////////////////////////////////////////////////////////////
function get_fields_by_scope(scope_e scope,ref uvm_reg_field flds[$]);
 foreach(m_reg_field_scopes_registry[scope,i])begin
    flds.push_front(m_reg_field_scopes_registry[scope][i]);
  end
endfunction
////////////////////////////////////////////////////////////////////////////////
function tag_field_by_assoc(tag_e fld_tag, assoc_e assoc, uvm_reg_field fld);
if(fld != null)begin
    if(m_reg_field_func_registry[fld_tag] != null)begin
      m_reg_field_assoc_registry[fld_tag][assoc] = fld;
    end
  end
endfunction
////////////////////////////////////////////////////////////////////////////////
function uvm_reg_field get_field_by_assoc(tag_e fld_tag, assoc_e assoc);
  return m_reg_field_assoc_registry[fld_tag][assoc];
endfunction
////////////////////////////////////////////////////////////////////
function set_ral_view_scopes(ral_view_e ral_view, ref scope_e view_scopes[$]);

  ral_view_scopes[ral_view] = view_scopes;

endfunction
////////////////////////////////////////////////////////////////////
function set_ral_view (ral_view_e ral_view);

   this.ral_view  =  ral_view;

endfunction
////////////////////////////////////////////////////////////////////
endclass : uvm_meta_reg_block
////////////////////////////////////////////////////////////////////////////
class rf_reg_block extends  uvm_meta_reg_block;
`uvm_object_utils(rf_reg_block)

  rand reg_status_general           m_reg_status_general;
  rand reg_control                  m_reg_control;

  virtual function void build();

    m_reg_control                  = reg_control::type_id::create("m_reg_control");
    m_reg_status_general           = reg_status_general::type_id::create("m_reg_status_general");
    m_reg_status_general.configure(this, null, "");
    m_reg_control.configure(this, null, "");
    m_reg_control.build();
    m_reg_status_general.build();

    tag_field_by_function(.fld(m_reg_control.irtry_to_send),.func_tag(TRY_SEND));
    tag_field_by_function(.fld(m_reg_status_general.sleep_mode),.func_tag(SLEEP_STATUS));
    tag_field_by_function(.fld(m_reg_control.set_hmc_sleep), .func_tag(SLEEP_CONTROL));
    tag_field_by_assoc(.fld_tag(SLEEP_CONTROL),.assoc(STATUS),.fld(m_reg_status_general.sleep_mode));
    add_field_scope(.scope(SLEEP), .func_tag(SLEEP_CONTROL));
    add_field_scope(.scope (SLEEP),.func_tag(SLEEP_STATUS));

  endfunction
endclass
```

Fig.2 META-RAL methodology implementation cont'd

```
task rf_control_read_seq::body();

  `uvm_info("rf_control_read_seq", print_reg,UVM_LOW)

  `uvm_info("META_RAL", $sformatf("We are going to call meta_ral"), UVM_LOW)
  fld = rf_rb.get_field_by_func(TRY_SEND);
 `uvm_info("META_RAL", $sformatf("irtry_to_send=%0x",fld.get()), UVM_LOW)

   fld = rf_rb.get_field_by_assoc(SLEEP_CONTROL,STATUS);
  `uvm_info("META_RAL", $sformatf("sleep_status name is %s",fld.get_name()), UVM_LOW)

  `uvm_info("META_RAL", $sformatf("Done calling meta_ral"), UVM_LOW)

endtask : body
```

Fig.3 Simple read sequence using META-RAL func_tag property

```
task rf_control_sleep_seq::body();

   uvm_reg_field sleep_ctrl, sleep_status;
   sleep_ctrl = rf_rb.get_field_by_func(SLEEP_CONTROL);
   // rf_rb.m_reg_status_general.sleep_mode.get();
   sleep_status  =  rf_rb. get_field_by_assoc(SLEEP_CONTROL,STATUS);
   sleep_status.get();
   sleep_ctrl.set(1'b0);  // rf_rb.m_reg_control.set_hmc_sleep.set(1'h0);

endtask : body
```

Fig.4 Simple read sequence using META-RAL associative property

Fig.3 shows before that the ***irtry_to_send*** field is being read properly with the same value written by the normal field ***uvm_reg_fiel::get()*** method. Then, Fig.5 shows the ***uvm_meta_reg_block*** structure and the simulation log for the mentioned code snippet.

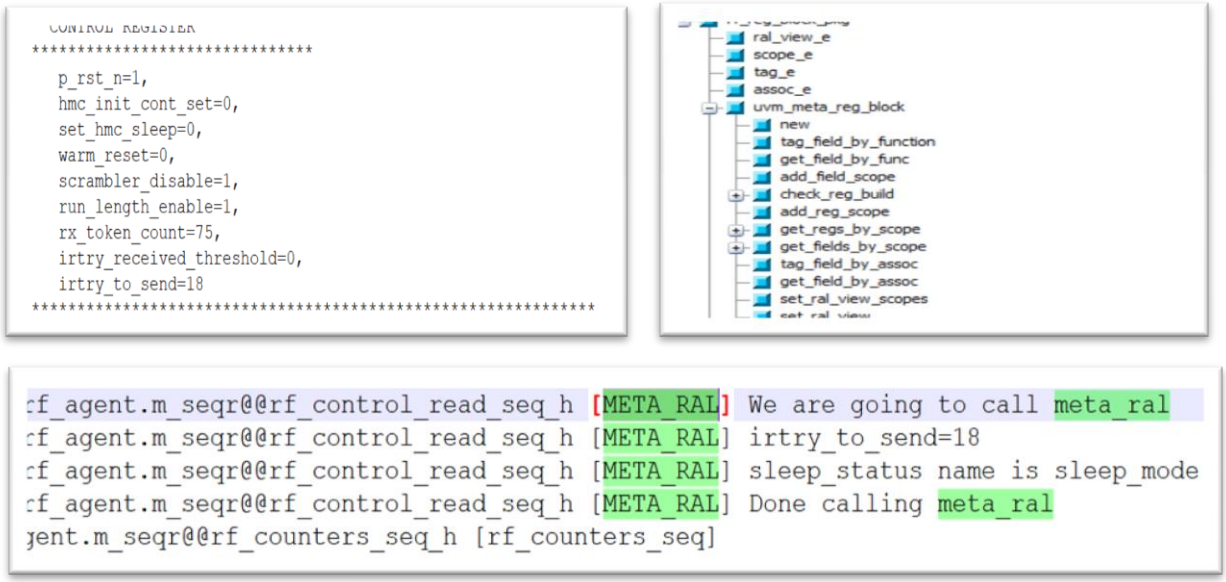


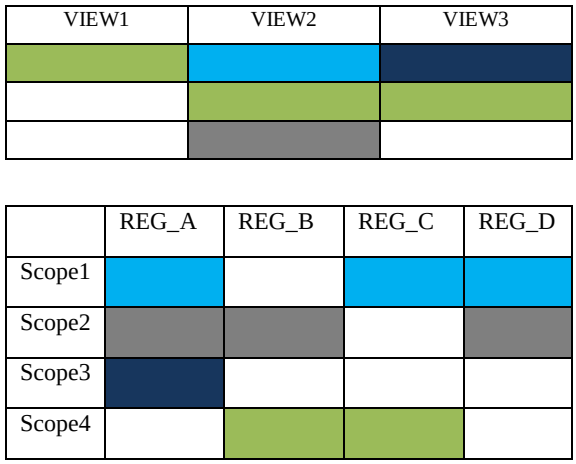
Fig.5 Simulation Log for code snippet in Fig.3

B. Multi-View RAL: Build registers on demand

In SoC, register size becomes significantly large, which would impact simulation performance. Using RAL to abstract these registers has to be carefully adopted to avoid degradation of simulation performance. In [2], some guidelines were proposed for such an impact by creating RAL registers without using the factory. This would reduce the impact of creating and loading thousands of proxy class objects. The study shows significant memory improvement, compile time, and build time when the factory is not used [2]. Also, it explores the possibility of creating the register on-demand only when and if it is to be accessed. However, due to UVM implementation, this seems to be unsuccessful due to the RAL model locking after all registers are built [2].

Here, we can propose a hybrid approach by building only registers of interest for the purpose of testing. By deploying the **META-RAL** methodology, it is possible to propose what can be called **Multi-*View RAL*** (***MV-RAL***). This methodology shows RAL registers and associated fields in different views according to test functionality. Fig.6 shows a visual representation of how ***MV-RAL*** is architected. Each RAL view consists of one or more register scopes. Each register may belong to one or more scopes. The ***ral_view*** variable can be configured in the test by setting ***ral_view*** in ***rf_reg_block,*** as demonstrated in the ***simple_test*** in Fig.6. With this approach, the RAL view can select one or more registers to build while skipping other registers that do not belong to that view. We achieved the following results by applying the MV-RAL concept.

1. VIEW_1 (SCOPES: 4) selects REG_B and REG_C.
2. VIEW_2 (SCOPES: 1, 2, 4) selects REG_A, REG_B, REG_C, and REG_D.
3. VIEW_3 (SCOPES: 3, 4) selects REG_A, REG_B, and REG_C.




```
class simple_test extends  base_test;
  `uvm_component_utils(simple_test)

view_e ral_view;

function void build_phase(uvm_phase phase);

   rf_rb.set_ral_view(rf_reg_block_pkg::SIMPLE);
   super.build_phase(phase);

endfunction : build_phase

endclass : simple_test
```

Fig.6 MV-RAL Architecture

According to the ***MV-RAL*** approach, one can select which registers and fields to build. In each test, a certain RAL view can be loaded. This view can inherently build a set of registers/fields according to some specific scope(s). In that way, registers are only created and built on demand which would reduce build time and memory as well. The ***rf_reg_block*** class implementation previously shown in Fig.1 can be modified to add multi-view RAL, as shown in Fig.7, which explains how to use the ***check_reg_build*** function to build registers of interest under specific views. The ***rf_reg_block*** explains the feature of MV-RAL and the same methodology can be applied to other registers in the register block.

```
class  rf_reg_block   extends  uvm_meta_reg_block;

`uvm_object_utils(rf_reg_block)

  string reg_name;
  ral_view_e ral_view;
  rand reg_status_general           m_reg_status_general;
  rand reg_control                  m_reg_control;

 virtual function void build();
    rf_map = create_map("rf_map", 'h0, 8, UVM_LITTLE_ENDIAN, 0);
      reg_name = "m_reg_status_general ";
  if(check_reg_build(ral_view, reg_name) )begin //Meta-RAL code
     /Meta-RAL functionality code
      m_reg_status_general = reg_control::type_id::create("m_reg_status_general ");
      add_reg_scope(STATUS_REGS,  m_reg_status_general  );
      m_reg_status_general .configure (this);
      m_reg_status_general.build();
     rf_map.add_reg(m_reg_status_general, 'h0, "RW"); // reg, offset, access
     tag_field_by_function(m_reg_status_general.sleep_mode,SLEEP_STATUS);
  end
   reg_name = "m_reg_control ";
  if(check_reg_build(ral_view, reg_name) )begin //Meta-RAL code
     /Meta-RAL functionality code
      m_reg_control  = reg_control::type_id::create("m_reg_control");
      add_reg_scope(CONTROL,  m_reg_control  );
      m_reg_control .configure (this);
     m_reg_control .build();
     rf_map.add_reg(m_reg_control, 'h2, "RW"); // reg, offset, access
     tag_field_by_function(m_reg_control.set_hmc_sleep,SLEEP_CONTROL);
     tag_field_by_function(m_reg_control.irtry_to_send,TRY_SEND);
     tag_field_by_assoc(SLEEP_CONTROL,STATUS,m_reg_status_general_sleep_mode);
  end

   lock_model ();

  endfunction
endclass
```

Fig.7 MV-RAL implementation

The study conducted in [2] shows that less than 500 of 14K registers are only accessed in most complex top-level scenarios. Therefore, the MV-RAL approach would significantly improve simulation performance. A sample verification environment based on HMC was downloaded from [10]. In order to mimic simulation conditions in [2], an additional 10,000 registers are added and simulated using ***QuestaSim***. Although the run ***simple_test*** is a short test, the proposed ***MV-RAL*** significantly enhanced the memory and wall time with a value of around 30-40%, as tabulated in Fig.8.

	UVM RAL	MV-RAL
# additional created registers	10,000	10
Memory size during simulation	375-397Mb	250-280Mb
Simulation wall time	63.5s	51.2s

Fig.8 MV-RAL performance evaluation

Since we will only create and build registers of interest, we will inherently apply this strategy to the RAL lookups. The added associative arrays will not impact the simulation performance. We accomplish this through the ***check_reg_build*** task, which is implemented as shown in Fig.1. In Fig 9, the simulation reveals that the **SIMPLE *ral_view*** does not allow access to the register ***m_reg_control_arr***, which raises an error when a test attempts to access an out-of-scope register.

```
** Fatal: (vsim-131) D:/Personal/Publications/dvcon2025/RALD/Graduation_project-main/Graduation_project-main/tb/seq_lib//rf_control_read_seq.svh(30): Null instance encountered when dereferencing 'this.rf_rb.m_reg_control_arr[0xf7d27424]'
    Time: 130950401 ps  Iteration: 1  Process: /uvm_pkg::uvm_sequence_base::start/#FORK#29_4f8ce7a4 File: D:/Personal/Publications/dvcon2025/RALD/Graduation_project-main/Graduation_project-main/tb/seq_lib//rf_control_read_seq.svh
# Fatal error in Task rf_control_read_seq::body at D:/Personal/Publications/dvcon2025/RALD/Graduation_project-main/Graduation_project-main/tb/seq_lib//rf_control_read_seq.svh line 30
```

Fig. 9 Out of test view access registers failure

C. Generic flow for RAL based algorithms: SerDes calibration as an example

In SerDes applications [12], several blocks require startup calibration algorithms to be applied. During release development, the register structure is not stable. Verifying that the calibration algorithm is working properly requires tracking and fixing these changes or developing scripts to automate RAL based algorithm generation. With the proposed methodology, RAL-based algorithms are easy to implement by accessing registers based on their scope and fields according to their functions, regardless of the naming conventions of registers, fields, or the structure of field parent registers or fields. Figure 10 depicts a generic flow for RAL based PLL calibration algorithm.

.

```
task meta_ral_pll_cal(serdes_ral_model ral_model);

 uvm_reg_field pll_cal_flds[$];
 uvm_reg_field fld;
 uvm_reg reg;
 uvm_status_e status;
 uvm_event cal_event;

 fld = ral_model.get_field_by_func(EN_PLL);
 fld.write(status,1);
 fld = ral_model.get_field_by_func(EN_PLL_CAL);
 fld.write(status,1);
 fld = ral_model.get_field_by_func(PLL_CAL_SEARCH_TYPE);
 fld.write(status,2'b00);
 fld = ral_model. get_field_by_func(PLL_CAL_TYPE);
 fld.write(status,2'b01);

 repeat(n)begin
   fld = ral_model.get_field_by_func(PLL_CAL_CODE);
   fld.write(status,pll_code);
   fld = ral_model. get_field_by_assoc(MPLL_CAL_CODE,TRIG_CLK);
   trigger_clk(fld); // to trigger a pulse for a clock field by writing 0-1-0
   cal_event = uvm_event_pool::get_global("pll_cal_code_updated");
   cal_event.wait_trigger();
   pll_code = next_pll_code();
 end

endtask
```

Fig.10 SerDes PLL Calibration Algorithm

Moreover, generic events can be embedded within the RAL sequence, such as the highlighted code snippet in Fig.10. This is very useful for keeping the algorithm generic and immune against design changes. This makes the RAL sequence reusable across different releases and even similar IPs independent of RAL structure and IP details. By implementing the events triggering actions in separate entities, one can extend this according to different IPs and releases as shown in Fig.11. This class can be inherited from ***uvm_monitor*** and instantiated either in the environment or in a specific agent that contains both events handlers and the RAL based sequence. Then, the event handler class can be overridden using ***uvm_factory*** according to IP/release.

```
class event_handler_ip1 extends meta_event_handler;

 virtual event_if e_vif;
 virtual task event_trigger;
  fork
    begin
     @(posedge e_vif.pll_code_updated);
     trig_event("pll_cal_code_updated ");
    end
   join
 endtask
endclass
```

```
task  meta_event_handler :: trig_event(string
sig_event);

  uvm_event e

  e = uvm_event_pool::get_global(sig_event);
  e.trigger();


endtask
```

Fig.11 Meta-RAL event handler

IV. META-RAL: APPLICATIONS

In this section, some applications are presented to make use of the proposed register access functions.

A. ATB Register Testing

Analog Test Bus (ATB) is a test methodology used to test analog mixed signals. By this methodology, analog nodes can be forced and sensed in a structured way to unhide analog bugs. By writing to a set of ATB registers, analog nodes can be forced or sensed via external ATB ports [11]. This technique allows one to access all ATB registers by looking for scope "ATB" instead of adding new wrapper ATB registers and fields. An example of ATB testing is exhibited in Fig.12.

```
task test_atb(serdes_ral_model ral_model);

  uvm_reg atb_regs[$];
  atb_regs = ral_model.get_regs_by_scope(ATB);

 foreach(regs[i])begin

    atb_regs[i]. enable.set(1'b1) ;
    atb_regs[i].update();

 end

endtask
```

Fig.12 ATB Testing

B. Secure Registers Testing

In some applications, sensitive information is stored in secure zones inside the chip. Secure registers are used to protect secret keys from malicious access. Those registers should not reveal those confidential parameters unless some secure protocol is applied. Missing some of those registers during the verification lifecycle is fatal. Therefore, adding the RAL scope property helps to get all secure registers by one command without the need to trace multiple documents to obtain information about those registers. Making RAL self-contained about register scopes and categories would save the verification effort and achieve coverage for the intended register testing within these secure applications.

C. RAL Scope coverage

This new approach can be instrumental in building a coverage model that samples the access to different scope' fields. This can be very useful in uncovering any gaps or missed functionalities. Finding the fields that are not accessed in certain scopes would flag an incomplete algorithm in the early stage of the verification cycle.

IV. INTER-RELEASES MANAGEMENT

Moving physical fields across registers makes tracking difficult throughout different RTL releases. Therefore, we must keep the field's unique tag tied to the physical field whenever the register structure changes. Some solutions have been deployed to achieve this.

1. The registers/field scopes and field functions are collected from system documents. Then, ***enumeration*** types are generated automatically to match these definitions. In this case, a compilation error would fire if a test tried to access an undefined ***view***, ***scope*** or ***function_tag***.
2. Like regular RAL access by name, when a register/field description does not exist, this should trigger a fatal error.
3. A Python script was developed to parse the verification environment and verify that all descriptions exist in RAL classes.
4. A ***Streamlit*** Python GUI script is developed to update any field with a specific tag. This script then, parses the verification environment and replaces the old tag with the new one. If the tag no longer exists and is recognized by the verification environment, it will raise a warning to fix/update the corresponding test case or the verification component. This can be depicted in Fig.13.

	Field Func	Release 1	Release 2	Release 3
field1	func1	RegA	RegA	RegA
field2	func2	RegB	RegB	RegA
field3	func3	RegA	RegB	RegB
field4	func4	RegC	RegC	RegC

Fig.13 GUI tool to manage register/field relationship

VI. CONCLUSION AND FUTURE WORK

This paper presents a new approach to register testing methodology. Adding new lookups for register/field scope and function makes it more flexible to test registers based on different testing purposes. A test can look for registers and apply the corresponding testing procedures according to certain policies. Additionally, this paves the way to enable generic RAL based sequences that are independent of register name or prior RAL model setup. Just by agreeing on register/field scopes and functions, complete sequences can be developed without concern about register naming changes due to DUT development instability.

Likewise, another framework is proposed to be built on top of META-RAL, called MV-RAL. In this approach, a test can set a specific RAL view which selects some RAL scopes. Each scope includes a set of registers to be created and built for the test purpose. These ease register testing by focusing on registers of interest and reducing the RAL structure for the unwanted registers. This is still a preliminary trial to add these new methodologies, and we can definitely go through different implementation alternatives to get the best performance and scalability. This should include, but not be limited to, integrating the implementation into the ***uvm_reg_field***, **uvm_reg**, **uvm_reg_block** and ***uvm_regmap*** reference classes. We encourage Accelera-Systems-Initiative to re-evaluate the UVM-RAL implementation and to consider the proposed ***META-RAL*** methodology, which offers greater flexibility in the development of RAL agents and sequences.

REFERENCES

[1] "Universal Verification Methodology (UVM) 1.2 Class Reference", Accelera, June 2014
[2] M. Litterick, M. Harnisch, Advanced UVM Register Modeling, DVCON 2014.

[3] V. Rousseau , S. Sinari, B. Applequist , T. McLean , G. Lallathin Automated Generation of RAL-based UVM Sequences, DVCON 2020.

[4] J. Anderson, Laura Montero, Random Stimuli Models for UVM Registers, DVCON 2019

[5] S. El-Ashry; A. Adel, Efficient Methodology of Sampling UVM RAL During Simulation for SoC Functional Coverage, International Workshop on Microprocessor and SOC Test and Verification (MTV), 2018.
[6] M. Shariff, R. Reddy, Functional-Coverage Sampling in UVM RAL, DVCON India 2019.
[7] A. Yehia, Boosting simulation performance of UVM registers in high performance, system, DVCON 2013.
[8] J, Aynsley, Doing Funny Stuff with the UVM Register Layer: Experiences Using Front Door Sequences, Predictors, and Callbacks, DVCON 2017
[9] R. Vincent, U. Nath and A. Chandran, Dynamic Control Over UVM Register Backdoor Hierarchy, DVCON 2019.
[10] https://github.com/AliMaher15/Graduation_project
[11] J. Huertas, Test and Design-for-Testability in Mixed-Signal Integrated Circuits, Springer 2004.
[12] "High Speed Serdes Devices and Applications", Springer 2000.

Appendix:

rf_reg_block_pkg.svh

```
package rf_reg_block_pkg;

  //uvm pakage and macros
  import uvm_pkg::*;
  `include "uvm_macros.svh"

  typedef enum {SIMPLE,COMPLEX} ral_view_e;
  typedef enum  {CONTROL,ADVANCED_CONTROL} scope_e;
  typedef enum  {TRY_SEND, SLEEP_CONTROL, SLEEP_STATUS} tag_e;
  typedef enum  {STATUS} assoc_e;


`include "uvm_meta_reg_block.svh"
 …
 …

endpackage : rf_reg_block_pkg
```

uvm_meta_reg_block.svh

```
`ifdef STANDALONE_COMPILE
   typedef enum {default_view} view_e;
   typedef enum {default_scope} scope_e;
`endif

class uvm_meta_reg_block extends uvm_reg_block;

  uvm_reg m_reg_scopes_registry[scope_e][$];
  uvm_reg_field m_reg_field_scopes_registry[scope_e][$];
  uvm_reg_field  m_reg_field_func_registry[tag_e];
  uvm_reg_field m_reg_field_assoc_registry[tag_e][assoc_e];
  uvm_reg  m_reg_assoc_registry[string][assoc_e];
  scope_e m_reg_field_name_scopes_registry[tag_e][$];
  scope_e m_reg_name_scopes_registry[string][$];
  scope_e ral_view_scopes[ral_view_e][$];
  ral_view_e ral_view;

function new(string name = "");
  super.new(name, UVM_NO_COVERAGE);
  ral_view = SIMPLE;
endfunction : new

function tag_field_by_function(uvm_reg_field fld, tag_e func_tag);
  if(fld != null)
    m_reg_field_func_registry[func_tag] = fld;
endfunction
```

```
function uvm_reg_field get_field_by_func(tag_e func_tag);
  return m_reg_field_func_registry[func_tag];
endfunction

function add_field_scope(scope_e scope, tag_e func_tag);
  uvm_reg_field fld;
  fld =  m_reg_field_func_registry[func_tag];
  m_reg_field_scopes_registry[scope].push_front(fld);
  m_reg_field_name_scopes_registry[func_tag].push_front(scope);
endfunction

function bit check_reg_build(ral_view_e ral_view, string reg_name);

`uvm_info("META_RAL", $sformatf("Check whether reg %s exists in view
%s",reg_name, ral_view.name()), UVM_LOW)

  foreach(m_reg_name_scopes_registry[reg_name][i]) begin

  `uvm_info("META_RAL", $sformatf("Check whether reg_name %s scope %s exists
in view %s",reg_name,m_reg_name_scopes_registry[reg_name][i].name(),
ral_view.name()), UVM_LOW)
     if(m_reg_name_scopes_registry[reg_name][i] inside
{ral_view_scopes[ral_view]}) begin

      return 1;
    end
  end
return 0;
endfunction

function add_reg_name_scope(scope_e scope, string reg_name);
  m_reg_name_scopes_registry[reg_name].push_front(scope);
endfunction

function add_reg_scopes(string reg_name, uvm_reg reg_s);

  scope_e scopes[$];
  scopes = m_reg_name_scopes_registry[reg_name];
  foreach(scopes[i])begin
    m_reg_scopes_registry[scopes[i]].push_front(reg_s);
  end

endfunction

function get_regs_by_scope(scope_e scope, ref uvm_reg regs[$]);
  foreach(m_reg_scopes_registry[scope,i])begin
    regs.push_front(m_reg_scopes_registry[scope][i]);
  end
endfunction

function get_fields_by_scope(scope_e scope,ref uvm_reg_field flds[$]);
 foreach(m_reg_field_scopes_registry[scope,i])begin
    flds.push_front(m_reg_field_scopes_registry[scope][i]);
  end
endfunction
```

```
function tag_field_by_assoc(tag_e fld_tag, assoc_e assoc, uvm_reg_field fld);
  if(fld != null)begin
    if(m_reg_field_func_registry[fld_tag] != null)begin
      m_reg_field_assoc_registry[fld_tag][assoc] = fld;
      end
  end
endfunction

function uvm_reg_field get_field_by_assoc(tag_e fld_tag, assoc_e assoc);
  return m_reg_field_assoc_registry[fld_tag][assoc];
endfunction

function set_ral_view_scopes(ral_view_e ral_view, ref scope_e view_scopes[]);
  ral_view_scopes[ral_view] = view_scopes;
endfunction

function set_ral_view (ral_view_e ral_view);
  this.ral_view  =  ral_view;
endfunction

endclass : uvm_meta_reg_block
```

rf_reg_block.svh

```
class rf_reg_block extends  uvm_meta_reg_block;

  `uvm_object_utils(rf_reg_block)

  rand reg_status_general             m_reg_status_general;
  rand reg_status_init                m_reg_status_init;
  rand reg_control                    m_reg_control;
  rand reg_control                    m_reg_control_arr[10000];

  …
  …
  …

virtual function void build();


ral_view_scopes[SIMPLE] = {CONTROL,ADVANCED_CONTROL};
ral_view_scopes[COMPLEX] = {ADVANCED_CONTROL};

add_reg_name_scope(.scope(ADVANCED_CONTROL),.reg_name("m_reg_control_arr"));
add_reg_name_scope(.scope(CONTROL),.reg_name("m_reg_control"));

// Create an instance for every register
m_reg_status_general          =
reg_status_general::type_id::create("m_reg_status_general");
m_reg_status_init             =
reg_status_init::type_id::create("m_reg_status_init");

// (Map name, base_addr, Number_of_bytes, Endianess, byte_addressing)
rf_map = create_map("rf_map", 'h0, 8, UVM_LITTLE_ENDIAN, 0);
```

```
if(check_reg_build(ral_view,"m_reg_control"))begin

 m_reg_control  = reg_control::type_id::create("m_reg_control");
 m_reg_control.configure(this, null, "");
 m_reg_control.build();
 rf_map.add_reg(m_reg_control, 4'h2, "RW");
 tag_field_by_function(m_reg_control.set_hmc_sleep,SLEEP_CONTROL);
 tag_field_by_function(m_reg_control.irtry_to_send,TRY_SEND);
 add_reg_scopes(.reg_name("m_reg_control"),.reg_s(m_reg_control));
end

if(check_reg_build(ral_view,"m_reg_control_arr"))begin

  foreach(m_reg_control_arr[i])begin

    m_reg_control_arr[i]  =
reg_control::type_id::create($sformatf("m_reg_control_%0d",i));
    m_reg_control_arr[i].configure(this, null, "");
    m_reg_control_arr[i].build();
    rf_map.add_reg(m_reg_control_arr[i], 20+i, "RW");

  end
end

// Configure every register instance
…
…
// Add these registers to the default map
…
…

tag_field_by_function(m_reg_status_general.sleep_mode,SLEEP_STATUS);
 tag_field_by_assoc(SLEEP_CONTROL,STATUS,m_reg_status_general.sleep_mode);
 lock_model();
endfunction : build
endclass : rf_reg_block
```

rf_control_sleep_seq.svh

```
class rf_control_sleep_seq extends  base_seq;

 `uvm_object_utils(rf_control_sleep_seq)

  uvm_event sleep_event_done,sleep_event_check;


task monitor_sleep_mode();

   while(!rf_rb.m_reg_status_general.sleep_mode.get()) begin

      rf_rb.m_reg_status_general.read(status, data, .parent(this));
```

```
      `uvm_info("SLEEP_SEQ","Reading Sleep mode in status register",UVM_LOW)
    end

    if(rf_rb.m_reg_status_general.sleep_mode.get())begin
    sleep_event_done = uvm_event_pool::get_global("sleep_activated");
    sleep_event_done.trigger();
   `uvm_info("SLEEP_SEQ","Done Sleep mode in status register",UVM_LOW)
  end

  endtask

  task body();
   time sleep_time = 10us;
   string print_reg;
   super.body();

   rf_rb.m_reg_control.read(status, data, .parent(this));
   rf_rb.m_reg_control.read(status, data, .parent(this));
                       );
   `uvm_info("SLEEP_SEQ",print_reg,UVM_LOW)

   rf_rb.m_reg_control.set_hmc_sleep.set(1'h1); // <<--
   rf_rb.m_reg_control.update(status);
   rf_rb.m_reg_control.read(status, data, .parent(this));
   rf_rb.m_reg_control.read(status, data, .parent(this));


   fork
     monitor_sleep_mode();
   join_none

   `uvm_info("SLEEP_SEQ",print_reg,UVM_LOW)

   `uvm_info("SLEEP_SEQ","Checking Sleep mode in status register",UVM_LOW)
   sleep_event_check = uvm_event_pool::get_global("sleep_activated");
   sleep_event_check.wait_trigger();
   rf_rb.m_reg_status_general.read(status, data, .parent(this));
   if (rf_rb.m_reg_status_general.sleep_mode.get()) begin

     …
     …

     `uvm_info("SLEEP_SEQ", print_reg,UVM_LOW)
   end
   //Stay in Sleep for up to 22 us
   sleep_time_rand_succeeds : assert (std::randomize(sleep_time)
                              with {sleep_time >= 2us && sleep_time < 22us;});
   //-- should be 1ms in real system
   #(sleep_time);
     `uvm_info("SLEEP_SEQ",$sformatf("SLEEP MODE: EXIT"),UVM_LOW)
   //Force openHMC controller to exit sleep mode
   `uvm_info("SLEEP_SEQ",$sformatf("Link is Up! from sleep"),UVM_LOW)

endtask : body

endclass : rf_control_sleep_seq
```