accellera SYSTEMS INITIATIVE

Register Access by Intent: Towards Generative RAL based Algorithms

Ahmed M. Allam

AMS Verification Consultant, ICpedia

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES
SAN JOSE, CA, USA
FEBRUARY 24-27, 2025

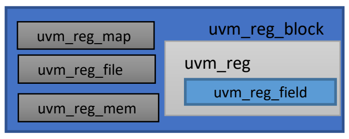
INTRODUCTION

Problem statement

- RAL is a verification tool that abstracts DUT registers, providing stable APIs for consistent access despite address changes, ensuring adaptability and stability in testing environments.
- Registers and fields may change, necessitating test case updates. This challenge promotes enhancing abstraction and hiding capabilities in RAL.
- This proposal presents META-RAL, enabling field access through names, functions, and implemented lookups effectively.

Execution challenges

- **Design churn:** During early design and cross-release phases, RAL fields shift across registers, requiring updates to test cases.
- **Performance degradation:** One-time RAL construction for the entire RAL, but test access few registers.



META-RAL Framework proposal

Design updates (Flexible RAL)

- Access fields by a unique func_tag.
- Access fields by associative properties.

RAL performance penalty (Multi-View RAL)

- RAL is organized into different views, can be chosen by the UVM test through scope.
- Each view comprises one or more scopes.
- A register or field can belong to multiple scopes.

Design dependent (Generic Events) :

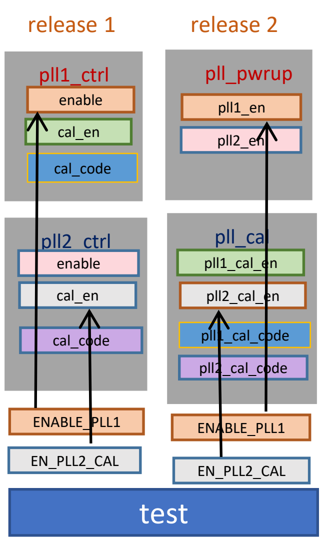
- Register events in uvm event pool
- Wait on generic events.
- Trigger events in a separate task or monitors.

Flexible RAL

- Lookups are implemented to keep track of RAL internal fields and can be accessed using a unique identifier known as **func_tag**.
- RAL structure changes across releases, but the test can still access fields regardless of field names or parent register names.

```
virtual function void rf_reg_block ::build();
  ..
  tag_field_by_function(.fld(m_pll1_ctrl.enable),.func_tag(ENABLE_PLL1));

  tag_field_by_function(.fld(m_pll1_ctrl.cal_code),.func_tag(PLL1_CAL_CODE));

  tag_field_by_function(.fld(m_pll1_ctrl.cal_en),.func_tag(ENABLE_PLL1_CAL));

  tag_field_by_assoc(.fld_tag(PLL1_CAL_CODE),.assoc(CAL_LS_CLK_EN)
  ,.fld(m_clks.clk50m_en));  // Static associativity

  tag_field_by_assoc(.fld_tag(PLL1_CAL_CODE),.assoc(CAL_HS_CLK_EN)
  ,.fld(m_clks.clk200m_en));  // Static associativity

endfunction
```

```
task rf_pll_cal_seq::body();
   uvm_reg_field pll1_en;
   pll1_en= rf_rb.get_field_by_func(ENABLE_PLL1);
   cal_hs_clk  = rf_rb.get_field_by_assoc(PLL1_CAL_CODE,CAL_HS_CLK_EN);
   cal_hs_clk.set(1'b1);
endtask : body
```



Multi-View RAL

- The RAL model consists of a single unit.
- All registers and fields are built regardless of test scenario.
- The build-up of all registers leads to decreased performance and increased resource consumption.

```
virtual function void rf_reg_block ::build();
..
..
   if(check_reg_build(ral_view,"m_pll1_ctrl" ) begin
..
      m_pll1_ctrl = reg_pll1_ctrl ::type_id::create(" m_pll1_ctrl ");
      tag_field_by_function(m_mpll1_ctrl.enable,ENABLE_PLL1);
   end
endfunction
```

```
rf_rb.set_ral_view(rf_reg_block_pkg::POWERUP);
```

- The Multi-View RAL framework addresses this problem by building registers for test purposes using a RAL hierarchical architecture with ral_view and ral_scope.
- Here we have 3 ral_views (powerup, basic_cal, adv_cal) and 4 ral_scopes (control, startup_cal, pvt_cal, vreg_cal)

powerup	basic_cal	adv_cal
▨	▨	▨
	🟦	🟦
		🟩
	🟨	🟨

- Each ral_scope can select which registers and fields to build.
- Memory can only be allocated to the test-accessed registers and fields.

scope/reg	pll_pwrup	vreg_pwrup	pll_cal	pll_fsm
control	▨	▨		
startup_cal	🟦	🟦	🟦	
pzvt_cal	🟩	🟩	🟩	🟩
vreg_cal		🟨		

Generic RAL based algorithm for Release/IP variations

- RAL-based algorithms face challenges due to waiting for IP-dependent signaling. In this proposal, each IP release will have its design-specific signal event handler.
- With the Multi-View RAL approach, RAL sequences can be written independently of IP design details.
- Algorithm and sequences are more compact and understandable.

```
task  meta_ral_pll_cal(serdes_ral_model ral_model);

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

  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(PLL_CAL_CODE,TRIG_CLK);
     trigger_clk(fld);
     cal_event = uvm_event_pool::get_global("pll_cal_code_updated");
     cal_event.wait_trigger();
     pll_code = next_pll_code();
   end

endtask
```

```
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
      // other events
    join

endtask
endclass
```

Results

- Flexible RAL reduces testcases maintenance effort due to RAL structural changes.
- MV-RAL can get rid of unwanted registers.
- Removing 10,000 registers reduced simulation memory by 30-40 %.
- Accessing registers out of scope/view would result in an error.
- Generic UVM events make RAL sequences reusable across different IP releases.

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

CONCLUSIONS

- This work introduces the META-RAL framework.
- Meta RAL consists of three frameworks.
  - ✓ Flexible RAL.
  - ✓ Multi-View RAL.
  - ✓ Generic RAL events.
- Lookups are built to track RAL fields (Function, Associative)
  - ✓ Testcases are immune to RAL structural changes.
  - ✓ Easy to generate RAL algorithms regardless reg/field name.
- Testcases select ral_view which builds registers on purpose
  - ✓ Attempting to access the field outside of ral_view will result in an access error.
  - ✓ Minimize the memory and performance impact of the build process by eliminating unnecessary registers.

ahmed.allam@icpedia.com

ICpedia