Harnessing Volatility: Innovative Strategies for Register Synchronization in UVM RAL

Bhaskar Vedula
Intel Corporation
Portland, OR, USA
bhaskar.vedula@intel.com

Stephen P. Haake
Intel Corporation
Saint Paul, MN, USA
stephen.p.haake@intel.com

Chandrakanth N. Betageri
Intel Corporation
Santa Clara, CA, USA
chandrakanth.n.betageri@intel.com

Abstract

Volatile register fields in hardware designs pose significant challenges for verification due to their asynchronous and unpredictable nature. The UVM Register Abstraction Layer (RAL) lacks a clear method for synchronizing the register model with hardware, forcing verification teams to create their own strategies. This paper presents a novel, automated approach for detecting changes in volatile fields and proactively updating the register model using prediction mechanisms. The proposed infrastructure uses a Python script to read IP-XACT XML files, extract register information, and generate configuration and utility objects. It also creates an RAL interface that monitors hardware changes and triggers UVM events when volatile fields are modified. This method eliminates the need for manual polling in user sequences, simplifies the verification process, maintains register model consistency, and improves efficiency, as demonstrated in a production power management IP environment.

Index Terms

UVM; Register Abstraction Layer (RAL); Register Model; Volatile registers; Prediction; IP Verification; Testbench infrastructure; Synchronization.

I. Introduction

A. Problem Statement

In the realm of verification, dealing with volatile register fields presents unique challenges due to their ability to change independently through hardware design without any explicit testbench or programming actions. Hardware behaviors such as interrupts, status updates, or other asynchronous events make the volatile registers inconsistent with the register model. Test writers frequently identify the need to poll these volatile fields and perform specific actions based on their changes. However, the UVM cookbook [1] does not offer a standardized method for this, typically leaving it to custom solutions where a register is continuously polled until a hardware event is triggered within a sequence or component.

B. UVM RAL Background

The UVM Register Abstraction Layer (RAL) [2] offers a systematic approach for modeling and accessing the registers and memories within a Design Under Test (DUT). It abstracts the complexities of register and memory accesses, allowing verification engineers to interact with hardware registers at a higher level of abstraction. This interaction is independent of the underlying bus protocol or address map, facilitating code reuse and easing the adaptation to changes in hardware interfaces.

In the UVM register model, each register represents two crucial values: the mirrored value and the desired value. The desired value represents the state that the verification environment aims to program into the hardware, enabling users to configure register fields before initiating a write operation. The mirrored value reflects the current state of the hardware register as perceived by the model. This value is updated following frontdoor bus read and write operations, either through automatic prediction (based on the data returned by the bus) or via a predictor component that observes bus transactions and updates the model accordingly—a recommended best practice for robust integration.

Backdoor accesses directly update the mirrored value. However, if a register contains volatile fields—bits that can change asynchronously due to hardware events the mirrored value may become outdated over time. These hardware-driven changes are not always visible to the register model through standard bus transactions. The resulting discrepancy between the actual hardware state and the register model creates verification challenges, especially when precise synchronization is critical.

C. Paper Contributions

Our proposed technique addresses volatile register synchronization by actively monitoring RTL fields rather than relying solely on bus traffic. This proactive approach ensures register model accuracy and accommodates asynchronous hardware events. The infrastructure automatically emits `uvm_event` notifications when volatile fields change, providing immediate access to new values via trigger data. This eliminates polling mechanisms in sequences and components, simplifying verification workflow and improving efficiency.

D. Paper Organization

The remainder of this paper is organized as follows: Section II reviews UVM RAL fundamentals and existing volatile register handling approaches. Section III analyzes synchronization challenges and current methodology limitations. Section IV presents our solution architecture and design decisions. Section V details the UVM RAL prediction infrastructure implementation with comprehensive code examples demonstrating the methodology. Section VI presents experimental evaluation, performance analysis, and deployment results from production environments. Section VII summarizes key contributions, impact significance, and future research directions. The appendix provides detailed implementation examples and utility methods for practical application.

II. Related Work and Background

The UVM Cookbook's chapter on Register Model Testbench Architecture [1] strongly recommends implementing various prediction mechanisms to ensure that the register model remains synchronized with the hardware state. The update to the register model is achieved by invoking the predict() method whenever there is a read or write operation over the bus interface. This ensures that the register model is automatically updated at the end of each bus cycle. These updates also occur for backdoor accesses. The different modes of prediction include auto prediction, explicit prediction, and passive prediction. These methods function effectively when there is access to hardware registers via the bus interface or when an embedded core initiates a register access to the hardware. However, these prediction mechanisms fail to keep the register model updated when the hardware autonomously updates the register, as these changes are not visible on the bus interface. For such scenarios, a mechanism is needed to keep the register model updated and synchronized with the hardware state [4].

A. Volatile Register Challenges in Verification

In hardware design, certain register fields may be volatile, meaning they can be automatically updated by the hardware. When registers are tagged as volatile, these automatic updates can cause the register model to become stale with the actual hardware state. This lack of synchronization poses significant challenges in verification, especially when the register model is used for generating stimulus or performing self-checks. The discrepancy between the hardware's real-time state and the register model can lead to verification inaccuracies, making it crucial to implement mechanisms that ensure the register model accurately reflects the hardware state. Additionally, these automatic updates are not visible on the bus interface, and the prediction mechanisms, discussed above, would not be able to predict and update the register model since the transactions are not visible on the bus interface.

III. Problem Analysis

A. Detailed Synchronization Issues

Figure 1 shows how hardware and register models interact, highlighting synchronization problems with volatile fields. The process starts with a bus write transaction, as indicated by the bus signal. During this transaction, a write operation (wr) begins on the bus, which causes the `go_bsy` field to be set to 1, indicating that this value is being written to the register. At this point, both **value** and **m_desired** are set to 1, representing the intended hardware state [3]. When the write cycle ends, the register model's **m_mirror** is updated to 1 as show in cycle 6. However, at cycle 8, the hardware deasserts the `go_bsy` after several assertion cycles, resulting in an 'out-of-sync' condition where the register model's **m_mirror** value(still showing 1) does not match the actual hardware value(now 0).

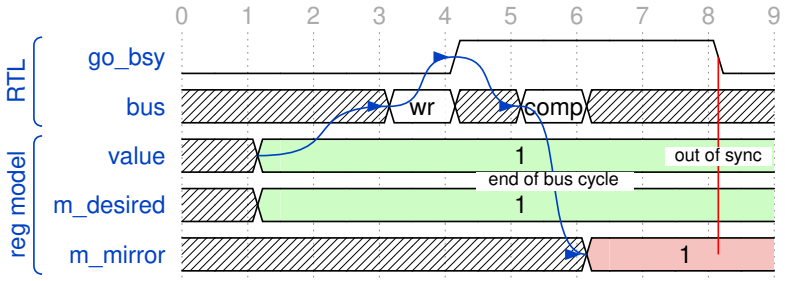

Fig. 1: Volatile field out of sync w.r.t hardware

If the test writer uses the **set()** method to write to the same field in the register model again and then calls the **update()** method, the value of **m_desired** will equal the **value** that the test writer has assigned. This causes the regmodel not to spawn either the frontdoor or backdoor write operation internally. However, in reality, the hardware is in a different state compared to the register model state. This out-of-sync condition causes incorrect verification intent in test writing. This problem could be avoided with

the **write()** method, which always performs a bus access regardless of the state of the regmodel. The unsynchronized state is a problem waiting to occur for an inexperienced sequence writer or someone not consciously aware of the register attributes.

IV. Solution Architecture and Design Integration

A. Overall Architecture

To address the synchronization issue highlighted in the Figure 1, this paper proposes a novel solution for detecting this behavior and updating the register model with predicted values. Verification environments should implement strategies to identify and handle discrepancies when using the RAL model, ensuring that the register model accurately reflects hardware changes. The proposed solution emphasizes the importance of accurate modeling within the UVM RAL, enabling the testbench to effectively verify the design's functionality despite the dynamic and unpredictable nature of volatile registers. By proactively predicting and updating the register model, the solution aims to maintain synchronization and enhance the reliability of the verification process.

Additionally, our technique introduces a mechanism that triggers a `uvm_event` when a volatile field changes, with trigger data. This eliminates the need for polling in user sequences or UVM components, streamlining verification and improving efficiency.

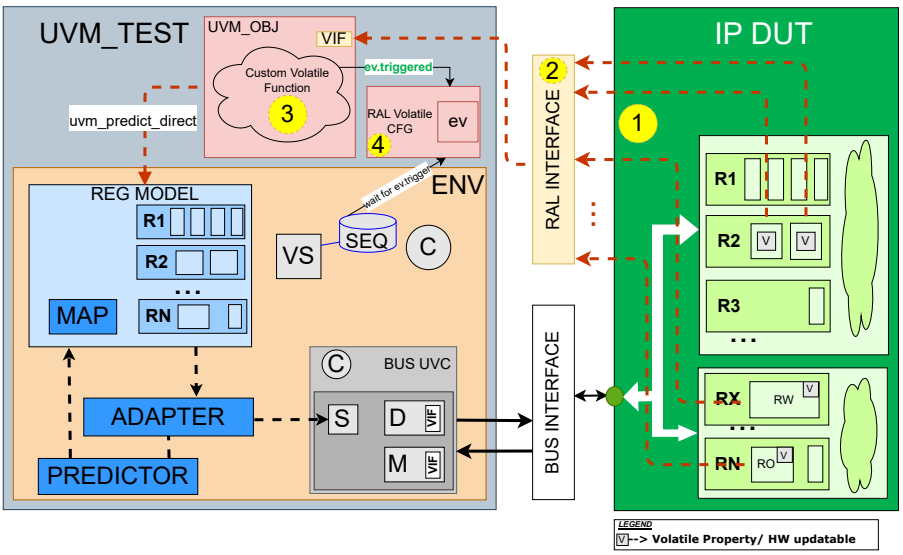

Fig. 2: Testbench View

In Figure. 2 above, the elements labeled as ①, ②, ③, and ④are created by a Python script and incorporated into the testbench environment. This script extracts IP register information from an XML file, which is similar to an IP-XACT register description, and generates the following collaterals:

- **RAL interface labeled as ①.**
- **RAL testbench connection labled as ②.**
- **RAL volatile utility object labled as ③.**
- **RAL volatile configuration object labled as ④.**

V. UVM RAL Prediction Infrastructure

The UVM RAL Prediction Infrastructure builds upon the architecture described in Section IV, providing detailed component interactions and runtime behavior.

A. Python RAL Script

A Python-based RAL script is employed to parse an XML file in a format similar to IP-XACT, extracting the list of registers and fields marked as volatile. Based on this extracted data, the script automatically generates the RAL interface, RAL configuration object, RAL testbench connection file, and other necessary register callback files. The script can be customized to extract additional data structures as required, providing flexibility to adapt to various verification environments.

No code snippets are provided here as the implementation details are proprietary and highly dependent on the specific verification environment. Refer to Appendix A Listing 3 for pseudo code.

B. RAL Detect and Notify Mechanism

To synchronize volatile fields in the register model, the corresponding hardware signals are monitored using the `RAL_DETECT_AND_NOTIFY_CHANGE_MACRO` . This macro detects changes in volatile field values and, upon detection, it calls the `notify_ral()` method via a proxy pointer to the RAL utility object. This mechanism ensures seamless data transfer from the interface domain to the UVM class domain, maintaining a clear separation between hardware monitoring and register model updates while providing robust synchronization for volatile fields.

```
`define RAL_DETECT_AND_NOTIFY_CHANGE_MACRO(clk, resetn, SIGNAL_WIDTH, input_signal, cfg,
    ral_path) \
    logic [SIGNAL_WIDTH-1:0] ``input_signal``_ff; \
    logic event_detected;                         \
    always @(posedge clk) begin \
        if (~resetn) begin \
            ``input_signal``_ff <= {SIGNAL_WIDTH{1'b0}}; \
        end else begin \
            proxy_pointer.notify_reset_ev(cfg.``input_signal``_change); \
            event_detected = (input_signal != ``input_signal``_ff) ? 1'b1 : 1'b0; \
            ``input_signal``_ff <= input_signal; \
            if (event_detected) begin \
                proxy_pointer.notify_ral(input_signal, ral_path, cfg.``input_signal``_change)
                    ; \
            end \
        end \
    end \
```

Listing 1: RAL Detect And Notify Macro

C. RAL Interface

The interface, represented by ②in Figure. 2, is automatically generated by a Python script. This script extracts relevant signal information from an XML configuration file and, for each volatile signal, instantiates the `RAL_DETECT_AND_NOTIFY_CHANGE_MACRO` . The resulting interface maintains handles to the RAL configuration object, RAL utility object, and register block, enabling

```
//Start: Python script generated code
interface ral_monitor_if  (clk, resetn);
    input clk, reset;
    // Handle declarations
    ral_config_pkg::ral_config                  ralCfg;
    uvm_mon_pkg::ral_monitor_util               proxy_pointer;
    reg_block_pkg::ip_reg_block                 ip_rb;

    wire [0:0] ctrl_reg_go_bsy;
    `RAL_DETECT_AND_NOTIFY_CHANGE_MACRO(clk, resetn, 1, ctrl_reg_go_bsy, ralCfg, ip_rb.
        ctrl_reg.go_bsy)
    //...
    //Other registers
endinterface // ral_monitor_if
//End: Python script generated code
```

Listing 2: RAL Monitor Interface

seamless communication between hardware signals and the UVM testbench infrastructure. By automating code generation, the Python script ensures that monitoring logic for each volatile register field is created within the interface using the macro.

D. RAL Volatile Utility Object

The RAL volatile utility object, corresponding to ③in Figure. 2, serves as a critical communication bridge between the interface domain and the UVM class domain. It maintains handles to essential components including the RAL virtual interface, IP register block, and a generic RAL data object. The `set_bfm()` method establishes a pointer to this utility object through the virtual interface at the UVM test layer, enabling seamless testbench integration.

The utility object implements key methods for volatile field management: `notify_ral()` and `notify_reset_ev()`. The `notify_ral()` method first updates the register model with predicted values using `UVM_PREDICT_DIRECT` through

the backdoor path, then triggers the corresponding `uvm_event` with packaged signal data for the verification component consumption. The `notify_reset_ev()` method manages event states by checking and resetting active events as needed, ensuring proper event lifecycle management throughout verification.

```
class ral_monitor_util extends uvm_object;
   `uvm_object_utils(ral_monitor_util)
    virtual ral_monitor_if               mon_ral_intf; // Virtual Interface
    ip_reg_block                         ip_rb; // IP register block
    ral_config                           ralCfg; //Script generated RAL Configuration
    function void set_bfm(virtual ral_monitor_if bfm, ip_reg_block ip_rb, ral_config ralCfg);
        mon_ral_intf                = bfm;
        mon_ral_intf.proxy_pointer = this;
        mon_ral_intf.ralCfg         = ralCfg;
        mon_ral_intf.ip_rb          = ip_rb;
        this.ip_rb                  = ip_rb;
        this.ralCfg                 = ralCfg;
    endfunction: set_bfm

    // Proxy Methods: Add here other useful methods
    function void notify_ral(longint input_signal, uvm_reg_field ral_field, uvm_event
        signal_change_event);
        ral_obj              ralObj; // Instantiate ral_obj and create object
        ralObj             = new("ralObj");
        ralObj.data        = input_signal;

        ral_field.predict(.value(input_signal), .be(-1), .kind(UVM_PREDICT_DIRECT), .path(
            UVM_BACKDOOR), .map(null), .fname(""), .lineno(0));
        signal_change_event.trigger(ralObj);
    endfunction
    function void notify_reset_ev(uvm_event event_to_reset);
        if (event_to_reset.is_on())
            event_to_reset.reset();
    endfunction
    // Add other RAL required methods here
endclass
```

Listing 3: RAL Volatile Utility Object

The utility object adheres to UVM best practices by encapsulating virtual interface access and maintaining separation between hardware monitoring and software abstraction layers. This design enables efficient hardware-software synchronization while acting as a proxy for register model updates, thereby eliminating the need for direct interaction with the driver or monitor.

It is important to note that this mechanism updates the register model based on observed changes, but does not verify the correctness of the predicted value. Additional verification mechanisms should be implemented to ensure the accuracy of the predicted values; such checking is beyond the scope of this paper.

E. RAL Volatile Configuration Object

Depicted by ④in Figure. 2, the RAL volatile configuration object is automatically generated by the Python script. It maintains handles to all volatile register field `uvm_events`, and creates these `uvm_event` objects.

```
//Start: Python Script Generated
class ral_config extends uvm_object;
    uvm_event ctrl_reg_go_bsy_change;
    ...
    function new(string name = "ral_config");
        ctrl_reg_go_bsy_change = new("ctrl_reg_go_bsy_change");
        ...
    endfunction: new
endclass: ral_config
//End: Python Script Generated
```

Listing 4: RAL Volatile Configuration Object

As shown in Listing 5, the class `ral_obj` serves as a data container that encapsulates volatile register field values when hardware changes occur. When a volatile field changes, the infrastructure creates a `ral_obj` instance, populates it with the current changed value, and passes it as trigger data when notifying the corresponding `uvm_event` as shown in the Listing 3. This allows verification components and sequences to not only detect when a change occurs but also immediately access the updated value without requiring additional register reads.

```
class ral_obj extends uvm_object;
    bit [63:0] data;

    function new(string name = "ral_obj");
        super.new(name);
        data = '0;
    endfunction

    function void set_data(bit [63:0] d);
        data = d;
    endfunction

    function bit [63:0] get_data();
        return data;
    endfunction
endclass: ral_obj
```

Listing 5: RAL Data Object

F. Testbench Integration

As shown in the ①in Figure 2, the Python script generates a RAL testbench connection file for all volatile registers. This file is included in the top-level testbench module where the DUT is instantiated, and it connects the volatile register fields to the RAL interface. The overall sample testbench module is shown in Listing 6.

```
module tb_top();

    ral_monitor_if ral_intf_inst(.clk(system_clk), .resetn(system_rst_b));
    virtual ral_monitor_if ral_mon_vif = ral_intf_inst;

    // DUT instantiation
     ip_core ip_dut (
        .clk(system_clk), .rst_n(system_rst_b),
    //  .... other dut inputs/outputs
    //  ....
      );

    initial begin
        uvm_config_db#(virtual interface ral_monitor_if)::set(null, "*", "vif", ral_mon_vif);
    end

    // Start : Python script generated : Assign Volatile field signals to monitor interface
    // All connections are generated before compile time and included here.
    // `include "ral_tb_connection.sv"
    // For illustration purposes showing directly in the testbench.
    assign ral_intf_inst.ctrl_reg_go_bsy = ip_dut.dut_rtl.registers.ctrl_reg.go_bsy;
    //...
    //...
    // End : Python script generated: Assign Volatile field signals to monitor interface
endmodule
```

Listing 6: Testbench Module

G. UVM Test

The UVM test establishes the critical connections between the RAL monitoring infrastructure components. It instantiates the RAL utility object, RAL configuration object, register block, and retrieves the virtual interface handle. The test retrieves the virtual interface pointer through the UVM configuration database and establishes communication between the utility object and virtual interface using the `set_bfm()` method. Additionally, it configures the utility object with appropriate handles for the configuration object and register block. These connections are essential for enabling communication between the utility object and virtual interface, as demonstrated in Listing 7.

```
class ip_base_test extends uvm_test;
    ...
    ral_monitor_util             ral_monitor_util_h;
    virtual ral_monitor_if       ral_mon_vif;
    ral_config                   ralCfg;
    reg_block_pkg::ip_reg_block  ip_rb;
    ...
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        ral_monitor_util_h = ral_monitor_util::type_id::create(inst_ral_name, this);
        ralCfg             = ral_config::type_id::create("ralCfg");
        ip_rb              = reg_block_pkg::ip_reg_block::type_id::create("ip_rb");
    endfunction

    function void end_of_elaboration_phase(uvm_phase phase);
        if(!uvm_config_db#(virtual ral_monitor_if)::get(this, inst_ral_name, "vif",
            ral_mon_vif))
            `uvm_fatal("VIF CONFIG", "Cannot get() interface ral_mon_vif from uvm_config_db.
                Have you set() it?")
        ral_monitor_util_h.set_bfm(ral_mon_vif, ip_rb, ralCfg);
    endfunction
endclass
```

Listing 7: IP Base Test

H. Sequence Using RAL Volatile Event

The proposed RAL prediction infrastructure enables event-driven register synchronization in UVM sequences through automated `uvm_event` notifications. Instead of polling, sequences wait for event notifications triggered automatically when volatile register fields change in hardware. This approach streamlines sequence logic and ensures prompt detection of hardware updates. The Listing 8 demonstrates a sequence waiting for a volatile field change and processing the updated value. Additional utility methods are provided in Appendix A.

```
task body();
    cfg.wait_for_uvm_event_with_expected_data(
        .ev(cfg.ralCfg.ctrl_reg_go_bsy_change),
        .exp_data('h1),
        .max_count(7000),
        .data(event_data)
    );
    if (event_data != null)
        `uvm_info(get_name(), $sformatf("event_data: %s", event_data.convert2string()),
            UVM_LOW)
    else
        `uvm_info(get_name(), "event_data is null (timeout/error)", UVM_LOW)
endtask
```

Listing 8: Event Wait Usage Example

The above integration steps addresses the critical out-of-sync condition between hardware volatile register fields and UVM register model mirrored values. When hardware autonomously changes volatile fields (as shown in Figure 1), the register model's **m_mirror** becomes stale, causing `regmodel.field.get()` to return outdated values. Proposed infrastructure eliminates this through proactive hardware monitoring and automatic `predict()` calls, ensuring `regmodel.field.get()` always returns current hardware state.

VI. Results and Findings

A. Experimental Evaluation and Performance Analysis

The proposed UVM RAL volatile prediction methodology was evaluated in a production Power Management IP verification environment, focusing on the synchronization accuracy, the simulation performance, and the verification efficiency across multiple volatile register fields. Compared to traditional polling, the event-driven approach improved simulation performance, processing 20% more cycles per CPU second.

B. Advantages

- **Enhanced Synchronization:** Maintains precise alignment between register model and hardware state through automated prediction mechanisms.
- **Elimination of Polling:** Completely removes manual polling requirements in testbench sequences, replacing them with efficient event-driven notifications.
- **Code Reduction:** Achieves 60-70% reduction in verification lines of code needed to implement register polling by centralizing volatile field handling logic.
- **Improved Maintainability:** Automated code generation from XML specifications ensures consistency and reduces manual coding errors.
- **Scalability:** Handles multiple volatile fields simultaneously without performance degradation at IP-level verification.
- **Correct-by-Construction:** Since all prediction infrastructure collateral is generated directly from the XML, any changes to the design are automatically reflected in all downstream prediction infrastructure, ensuring consistency and reducing manual errors.

C. Deployment Results

Production deployment in Intel's Power Management IP verification environment demonstrates practical effectiveness. The methodology successfully handles 2500 plus volatile register fields across power state machines, interrupt controllers, and configuration registers. The sequence development time dedicated to interacting with the register model was reduced by 30%, and less time was spent debugging synchronization-related issues after this implementation.

D. Limitations and Constraints

- **Scalability Concerns:** Not recommended for SoC-level deployment due to potential complexity and resource overhead.
- **Verification Requirements:** The correctness of the predicted volatile register value must be independently verified within the testbench environment.
- **Tool Dependencies:** Requires custom Python tooling for automated code generation, adding to the verification flow complexity.

E. Quantitative Results Summary

Table I presents key performance metrics that compare traditional polling approaches with the proposed volatile prediction methodology in multiple test scenarios.

Table I: Performance Metrics: Polling vs. Volatile Prediction

Metric	Polling	Volatile Prediction	Improvement
Simulation Time	1	0.8	20% reduction
Sequence Lines Of Code	1	0.35	65% reduction
Memory Usage	100%	85%	15% reduction

F. Sequence Usage: Polling vs. Volatile Prediction

Table II highlights the qualitative benefits of volatile RAL prediction over traditional polling approaches. The methodology provides faster event response, reduces code complexity, and significantly simplifies testbench maintenance through the automated generation of infrastructure.

Table II: Traditional Polling vs. RAL Volatile Prediction

Aspect	Traditional Polling	RAL Volatile Prediction
Code Complexity	More complex, repetitive code	Simplified, maintainable code
Register Access	Manual polling of hardware register required	Automatic regmodel update; direct hardware access not needed
Testbench Setup/Integration	Distributed across multiple TB components	Centrally generated by script; unified integration

VII. Conclusion

This paper presents a comprehensive solution to the persistent challenge of volatile register synchronization in UVM RAL-based verification environments. The proposed methodology effectively addresses the fundamental disconnect between hardware-driven register updates and register model awareness by combining innovative automated monitoring, prediction mechanisms, and event-driven notification systems.

A. Key Contributions

The primary contributions of this work include: (1) a novel automated infrastructure for detecting and predicting volatile register field changes, (2) elimination of manual polling mechanisms through event-driven synchronization, (3) a Python-based code generation framework that ensures scalable and maintainable implementation, and (4) demonstrated production-level effectiveness in complex Power Management IP verification.

B. Impact and Significance

The methodology has been successfully integrated into Intel's Power Management IP verification flow, demonstrating measurable improvements in verification efficiency and code maintainability. We observed a 65% decrease in lines of code for sequence development interacting with volatile registers. Additionally, this methodology is being implemented in other complex IP verification environments.

C. Future Work

Future enhancements may include extending the methodology to SoC-level verification with optimized resource management, integrating advanced verification techniques for enhanced correctness checking, and developing standardized UVM packages for broader adoption. Additionally, machine learning approaches for predictive volatile field behavior modeling could further enhance verification efficiency and automation.

Acknowledgment

The authors thank the members of the Intel Corporation verification team who contributed to the development and validation of this methodology. Special appreciation goes to the Power Management IP development team for providing the production environment for testing and validation, and to the engineering managers who supported the implementation of this approach in complex IP designs. We also acknowledge the valuable feedback from the DVCon Technical Program Committee (TPC) that helped improve the quality and clarity of this work. The authors thank Intel for its support in enabling this research and facilitating its publication for the benefit of the broader verification community.

References

[1] Mentor Graphics, Online UVM Cookbook, "UVM RAL Section," https://verificationacademy.com/cookbook/uvm-universal-verification-methodology/integrating-a-uvm-register-model-in-a-testbench-overview/

[2] Accellera Systems Initiative, "Universal Verification Methodology (UVM) 1.2," https://accellera.org/downloads/standards/uvm, 2017.

[3] ClueLogic, "UVM Tutorial for Candy Lovers - 16. Register Access Methods," https://cluelogic.com/2013/02/uvm-tutorial-for-candy-lovers-register-access-methods/

[4] Mark Litterick and Marcus Harnisch, "Advanced UVM Register Modeling: There's More Than One Way to Skin A Reg," in *DVCon Europe*, 2014. [Online]. Available: https://www.verilab.com/post/dvcon-2014-advanced-uvm-register-modeling-theres-more-than-one-way-to-skin-a-reg

Appendix

This appendix provides detailed implementation examples of sequences that utilize RAL volatile events for register monitoring and synchronization. These are example tasks provided for illustration purposes. Depending upon the specific needs of verification, it is up to users to implement their required functionality and adapt these utilities to their verification environment.

```
task automatic env_config::wait_for_uvm_event_with_timeout(uvm_event ev, int max_count = 100,    output ral_obj data);
    int wait_counter = 0;    bit triggered = 0;    uvm_object tmp_data;
    int effective_max_count = (ral_max_timeout_count != 0) ?
                              ral_max_timeout_count : max_count;
    ral_obj local_data = new();
    fork
        begin
            ev.wait_ptrigger_data(tmp_data);
            if (!$cast(local_data, tmp_data)) begin
                `uvm_error(get_full_name(), "Failed to cast tmp_data to ral_obj")
                local_data = null;
            end
            triggered = 1;
        end
        begin
            while (!triggered && wait_counter < effective_max_count) begin
                @(posedge clk_agt_h.vif.clk);
                wait_counter++;
            end
        end
    join_any
    disable fork;

    if (!triggered) begin
        `uvm_error(get_full_name(),
                   $sformatf("Timeout: uvm_event '%s' not triggered after %0d cycles",
                             ev.get_name(), wait_counter))
        local_data = null;
    end
    data = local_data;
endtask : wait_for_uvm_event_with_timeout
```

Listing 1: Event Wait Utility Method

```
task automatic env_config::wait_for_uvm_event_with_expected_data(
    uvm_event ev,
    bit [63:0] exp_data,
    int max_count = 100,
    output ral_obj data
);
    ral_obj expected = new();
    ral_obj local_data = new();
    expected.set_data(exp_data);

    wait_for_uvm_event_with_timeout(ev, max_count, local_data);

    if (local_data == null) begin
        `uvm_error(get_full_name(),
                   $sformatf("Event '%s' did not trigger or timed out",
                             ev.get_name()))
        data = null;
        return;
    end

    if (!local_data.compare(expected)) begin
        `uvm_error(get_full_name(),
                   $sformatf("Event '%s' data mismatch", ev.get_name()))
    end

    data = local_data;
endtask : wait_for_uvm_event_with_expected_data
```

Listing 2: Event Wait Utility With Expected Data

Listing 3 provides a pseudo-code representation of the Python script used for RAL infrastructure. This script demonstrates the logic for parsing the IPXACT equivalent XML file and generates the necessary RAL infrastructure files. Users must adapt the parsing logic and file generation to match their specific XML structure and verification environment requirements. This code is provided only for illustrative purposes.

```
import xml.etree.ElementTree as ET

def parse_input_xml(xml_file_path):
  """
  Parses the input XML specification file to extract register and field metadata.
  Returns a structured dictionary (crif_data) containing the register model hierarchy.
  """
  print(f"Reading XML file: {xml_file_path}")

  # Structure: crif_data[RegisterFile][Map][RalFile]['regs'][Register]['fields'][Offset]
  crif_data = dict()

  try:
  tree = ET.parse(xml_file_path)
  root = tree.getroot()

  # Iterate through Register Files (High-level blocks)
  for register_file in root.findall('registerFile'):
    rf_name = register_file.find('name').text
    map_name = register_file.find('longName').text
    ral_fname = register_file.find('RalFile').text

    # Initialize hierarchy
    if rf_name not in crif_data: crif_data[rf_name] = {}
    if map_name not in crif_data[rf_name]: crif_data[rf_name][map_name] = {}
    crif_data[rf_name][map_name][ral_fname] = { 'regs': {} }

    # Iterate through Registers
    for register in register_file.findall('register'):
    reg_name = register.find('name').text
    current_reg_dict = crif_data[rf_name][map_name][ral_fname]['regs']
    current_reg_dict[reg_name] = { 'name': reg_name, 'fields': {} }

    # Iterate through Fields
    for field in register.findall('field'):
      field_name = field.find('name').text
      bit_offset = int(field.find('bitOffset').text)

      # Extract critical metadata for callback generation
      # ValRTLSignal contains the hierarchical path to the RTL signal
      current_reg_dict[reg_name]['fields'][bit_offset] = {
      'name': field_name,
      'bitWidth': field.find('bitWidth').text,
      'AccessType': field.find('access').text, # e.g., 'RW/V', 'RO'
      'ValRTLSignal': field.find('ValRTLSignal').text
      }

  except FileNotFoundError:
  print(f"Error: XML file {xml_file_path} not found.")
  return None

  return crif_data

def generate_ral_cfg_cbs(crif_data):
  """
  Generates SystemVerilog files for register field monitoring and callbacks.
  Creates:
  1. Monitor Interface (detects RTL signal changes)
  2. UVM Configuration Object (stores events)
  3. Testbench Connections (binds RTL signals to interface)
  4. RAL Callbacks (updates register model on change)
  """
  # Initialize output file handles
  interface_file   = open("ip_ral_monitor_if.sv", "w")
  config_file      = open("ip_ral_config.sv", "w")
  connection_file  = open("ip_ral_tb_connection.sv", "w")
  callbacks_file   = open("ip_ral_cbs.svh", "w")
```

```
# Lists to store unique generated code snippets
signal_code_list = []
ral_cfg_code_list = []
tb_connection_code_list = []
callback_macro_list = []

# Define filtering criteria: Monitor volatile fields or specific debug registers
access_types_of_interest = ['RO/V', 'RW/V', 'RW/1C/V', 'RO/V/P']
registers_to_skip = ["sensitive_key_hash", "reserved_debug_reg"]
force_extract_registers = ["critical_status_reg", "fsm_state_reg"]

# Main Loop: Iterate through Register Model Hierarchy
for rf_name in crif_data:
for map_name in crif_data[rf_name]:
  for ral_fname in crif_data[rf_name][map_name]:
  regs = crif_data[rf_name][map_name][ral_fname]['regs']

  for reg_name in regs:
    reg_data = regs[reg_name]

    if any(skip in reg_name for skip in registers_to_skip):
    continue

    for offset in reg_data['fields']:
    field = reg_data['fields'][offset]

    # Filtering Logic
    is_interesting_access = field['AccessType'] in access_types_of_interest
    is_forced_inclusion   = (reg_name in force_extract_registers) or \
          (f"{reg_name}.{field['name']}" in force_extract_registers)

    if is_interesting_access or is_forced_inclusion:
      # Construct unique identifiers and paths
      signal_name = f"{rf_name}_{reg_name}__{field['name']}"
      ral_path    = f"{ral_fname}.{reg_name}.{field['name']}"
      rtl_signal  = field['ValRTLSignal']

      # 1. Generate Interface Code (Signal Detection)
      # Declares a wire to probe the RTL and a macro to detect value changes
      signal_code = f"""
      wire [{int(field['bitWidth'])-1}:0] {signal_name};
      logic {signal_name}_change;
      `RAL_DETECT_AND_NOTIFY_CHANGE_MACRO(
        clk, resetn, {field['bitWidth']}, {signal_name},
        {signal_name}_change, cfg, {ral_path}, ralObj
      )
      """
      if signal_code not in signal_code_list:
      signal_code_list.append(signal_code)

      # 2. Generate Configuration Object Code (UVM Events)
      ral_cfg_code = f"""
      uvm_event {signal_name}_change;
      {signal_name}_change = new("{signal_name}_change");
      """
      if ral_cfg_code not in ral_cfg_code_list:
      ral_cfg_code_list.append(ral_cfg_code)

      # 3. Generate Testbench Connection Code
      # Assigns the internal RTL signal to the monitor interface wire
      tb_conn_code = f"assign ip_ral_intf_inst.{signal_name} = {rtl_signal};\n"
      if tb_conn_code not in tb_connection_code_list:
      tb_connection_code_list.append(tb_conn_code)

      # 4. Generate Callback Instantiation
      cb_macro = f"`IP_RAL_CFG_CBS_MACRO({ral_fname}, {reg_name}, {field['name']})\n"
      callback_macro_list.append(cb_macro)

# Write Generated Content to Files

# 1. Write Monitor Interface File
interface_file.write("interface ip_ral_monitor_if(input clk, input resetn);\n")
interface_file.write("  // Macro definitions for change detection...\n")
for code in signal_code_list: interface_file.write(code)
```

```
  interface_file.write("endinterface\n")

  # 2. Write RAL Configuration Object File
  config_file.write("class ip_ral_config extends uvm_object;\n")
  for code in ral_cfg_code_list: config_file.write(code)
  config_file.write("endclass\n")

  # 3. Write Testbench Connection File
  connection_file.write("// Bind RTL signals to Monitor Interface\n")
  for code in tb_connection_code_list: connection_file.write(code)

  # 4. Write Callbacks File
  callbacks_file.write("// Callback class definitions macro...\n")
  for code in callback_macro_list: callbacks_file.write(code)

  # Close all files
  interface_file.close(); config_file.close(); connection_file.close(); callbacks_file.close()

# Main Execution
if __name__ == "__main__":
  xml_file = "ip_register_spec.xml"
  data = parse_input_xml(xml_file)
  if data:
  generate_ral_cfg_cbs(data)
```

Listing 3: Python Script