Optimizing CPU-Based Configuration Path Verification Through Automated C Test Case Generation with UVM RAL

Ravi Mangal, Google IT Services India Private Limited, Bangalore, India
(mangalravi@google.com)

Alisha Parvez, Google IT Services India Private Limited, Bangalore, India
(alishaparvez@google.com)

Apurva Shah, Scaledge India Private Limited, Bangalore, India
(apurva.s@scaledge.io)

***Abstract*-The verification of Central Processing Unit (CPU) accessible registers in complex System-on-Chips (SoCs) presents a significant challenge, often caught between the powerful abstraction of the Universal Verification Methodology (UVM) Register Abstraction Layer (RAL) and the bare-metal necessity of C-based tests. C-based tests, essential for architectural and boot-path validation, traditionally lack the sophisticated modeling capabilities of UVM RAL, leading to inefficient, brittle tests and a high incidence of false failures from undocumented or complex register behaviors. This paper introduces a novel, automated methodology that systematically bridges this gap. We leverage UVM RAL as the reference source to generate portable, robust, and efficient C-compatible register test collaterals. The proposed flow utilizes a UVM sequence to extract comprehensive register metadata—including waivers and access policies—into an intermediate Comma-Separated Values (CSV) format, which is then synthesized into C data structures. Furthermore, we detail an intelligent register pruning technique that reduced test suite execution time by over 60% in a production environment. This methodology enhances verification accuracy by seamlessly porting IP-level waivers to the SoC context, has directly led to the identification of 35+ critical design defects, and has fundamentally improved regression efficiency and time-to-market.**
Keywords—Register Verification; UVM RAL; C-based testing; Automated Test Generation; SoC Verification

I. Background and Problem Statement

The operational integrity of a modern SoC is fundamentally dependent on the correct functionality of its hardware registers. These registers, often numbering in the tens of thousands, form the primary interface between software and hardware, controlling everything from clock gating and power management to peripheral configuration and system status. Verifying the accessibility and behavior of every register from the vantage point of an embedded processor is a critical, non-negotiable phase of SoC validation. The industry-standard approach for register verification at the Intellectual Property (IP) or subsystem level is the UVM Register Abstraction Layer (RAL). UVM RAL provides a powerful, object-oriented model that abstracts register addresses, fields, and access policies, offering built-in sequences for comprehensive checks like reset value, bit-bash, and shared access.

However, a significant "verification chasm" emerges at the SoC level. While UVM provides coverage in a SystemVerilog simulation environment, final validation requires tests executed directly by the on-chip processor(s). These tests, typically written in C, are necessary to validate:

- **Architectural Path:** The physical path from the CPU’s instruction execution unit, through interconnects and bridges, to the target register.

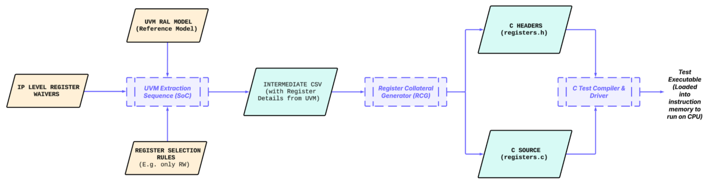

UVM RAL MODEL (Reference Model)
IP LEVEL REGISTER WAIVERS
UVM Extraction Sequence (SoC)
REGISTER SELECTION RULES (E.g. only RW)
INTERMEDIATE CSV (with Register Details from UVM)
Register Collateral Generator (RCG)
C HEADERS (registers.h)
C SOURCE (registers.c)
C Test Compiler & Driver
Test Executable (Loaded into instruction memory to run on CPU)

Figure 1: UVM to C Generation Flow

- **Boot & Firmware Integrity:** Ensuring essential registers are accessible in a bare-metal environment before an operating system is loaded.
- **Software-Hardware Interaction:** Validating the exact behavior observed by firmware

Conventional C-based testing is inefficient. Manually written or crudely scripted tests often lack an understanding of register properties, leading to critical problems:

- **Absence of Abstraction:** C tests treat registers as memory-mapped addresses, ignoring nuanced access policies (e.g., Write 1 to Clear, Read Only, self-clearing bits) explicitly modeled in UVM RAL. Tests must account for these policies to avoid spurious failures.
- **Decoupling from Reference Model:** C tests are separate from the UVM RAL model, the "reference." Frequent register updates necessitate manual C test modification to stay in sync with UVM RAL, unlike RAL tests.
- **Waiver Discontinuity:** Waivers for known design deviations (e.g., a reserved field not returning zero) managed in UVM (e.g., uvm_config_db) are lost in C, causing "bugs" to be refiled, wasting debug and regression time. SoC-level verification relies on IP and Sub-System UVM waivers, which must be ported to C for processor reuse.
- **Scalability Issues:** Manually verifying an address space with tens of thousands of registers is unsustainable. Long C tests drastically increase regression turnaround times, especially at the SoC level where runtimes are high.

This paper details a robust solution to eliminate these challenges by creating a formal, automated bridge between the UVM RAL domain and C-based processor testing.

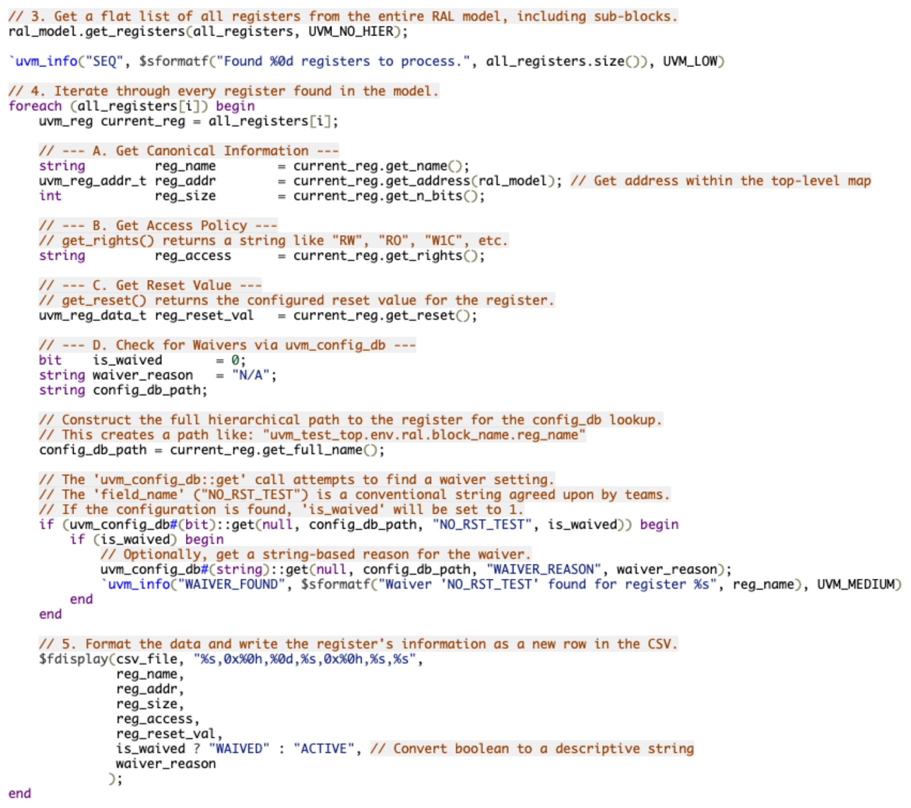
II. Solution: A UVM-to-C Generation Flow

Proposed methodology establishes a systematic, single-source flow for generating C test collaterals from the authoritative UVM RAL model. It is important to note that this flow does not generate the C test logic itself; rather, it generates the rich data structures that parameterize a pre-existing, generic C test driver. The process consists of four key stages, as illustrated in Figure 1.

A. Stage 1: UVM Sequence-based Metadata Extraction

The foundation of proposed flow is a generic UVM sequence that operates on the compiled RAL model of the design. Instead of performing traditional register writes and reads, this sequence acts as an introspective data miner. It iterates through every register defined in the RAL model and uses the rich UVM RAL Application Programming Interface (API) to extract critical metadata, as shown in Figure 2.For each register, the sequence queries:

- **Canonical Information:** Register name, absolute address, size in bits.
- **Access Policy:** The register's access rights (e.g., RO, RW, WO, W1C, RC - Read-to-Clear). This is obtained via the get_rights() method.



```
// 3. Get a flat list of all registers from the entire RAL model, including sub-blocks.
ral_model.get_registers(all_registers, UVM_NO_HIER);

`uvm_info("SEQ", $sformatf("Found %0d registers to process.", all_registers.size()), UVM_LOW)

// 4. Iterate through every register found in the model.
foreach (all_registers[i]) begin
    uvm_reg current_reg = all_registers[i];

    // --- A. Get Canonical Information ---
    string          reg_name         = current_reg.get_name();
    uvm_reg_addr_t reg_addr         = current_reg.get_address(ral_model); // Get address within the top-level map
    int             reg_size         = current_reg.get_n_bits();

    // --- B. Get Access Policy ---
    // get_rights() returns a string like "RW", "RO", "W1C", etc.
    string          reg_access       = current_reg.get_rights();

    // --- C. Get Reset Value ---
    // get_reset() returns the configured reset value for the register.
    uvm_reg_data_t reg_reset_val    = current_reg.get_reset();

    // --- D. Check for Waivers via uvm_config_db ---
    bit    is_waived        = 0;
    string waiver_reason    = "N/A";
    string config_db_path;

    // Construct the full hierarchical path to the register for the config_db lookup.
    // This creates a path like: "uvm_test_top.env.ral.block_name.reg_name"
    config_db_path = current_reg.get_full_name();

    // The 'uvm_config_db::get' call attempts to find a waiver setting.
    // The 'field_name' ("NO_RST_TEST") is a conventional string agreed upon by teams.
    // If the configuration is found, 'is_waived' will be set to 1.
    if (uvm_config_db#(bit)::get(null, config_db_path, "NO_RST_TEST", is_waived)) begin
        if (is_waived) begin
            // Optionally, get a string-based reason for the waiver.
            uvm_config_db#(string)::get(null, config_db_path, "WAIVER_REASON", waiver_reason);
            `uvm_info("WAIVER_FOUND", $sformatf("Waiver 'NO_RST_TEST' found for register %s", reg_name), UVM_MEDIUM)
        end
    end

    // 5. Format the data and write the register's information as a new row in the CSV.
    $fdisplay(csv_file, "%s,0x%0h,%0d,%s,0x%0h,%s,%s",
              reg_name,
              reg_addr,
              reg_size,
              reg_access,
              reg_reset_val,
              is_waived ? "WAIVED" : "ACTIVE", // Convert boolean to a descriptive string
              waiver_reason
             );
end
```

Figure 2: Example Sequence Snippet to extract register details from UVM and dump them into a CSV

- **Reset Value:** The expected value after a hardware reset, retrieved using get_reset().
- **Waiver Status:** The sequence queries the uvm_config_db for any waivers associated with the specific register or its fields. This allows IP-level waivers to be programmatically captured.

Example: Capturing a UVM Waiver

An IP team may waive a test on a specific register, DESIGN_STATUS, because a volatile status field within it is untestable for a fixed reset value. This is typically implemented in the UVM environment as shown in Figure 2. Extraction sequence detects this configuration setting and flags the DESIGN_STATUS register as waived.

B. *Stage 2: Generation of Intermediate CSV Collateral*

The extracted metadata is then written to a structured, human-readable Comma-Separated Values (CSV) file. This intermediate format serves as a clean, decoupled interface between the UVM and C domains. It also allows for easy inspection, debugging, and manual augmentation if necessary. A typical row in the CSV file is structured as follows: Register Name, Address,Access, Reset Value, Is Waived, Waiver Comment, as shown in Figure 3.

C. *Stage 3: Synthesis of C-Compatible Data Structures*

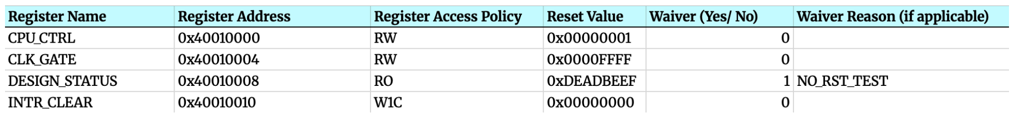
The final stage involves a post-processing Register Collateral Generator (RCG) tool that parses the intermediate CSV file and automatically generates C header (.h) and source (.c) files. These files define a global array of structures, where each structure element represents a register and its associated properties.

Intelligent Test Pruning
For initial system bring-up, rapid, high-confidence validation of core connectivity is key. A full register test suite is inefficient; a better strategy is to access a strategically selected subset, including the first and last register of each IP block to validate address decode boundaries, and a random internal register for data path integrity. Traditional C testing requires manual consultation of static documentation to identify boundary addresses, a tedious and error-prone process. The UVM RAL model, however, makes it trivial to programmatically query and identify this subset. Our methodology ports this UVM automation to C. During extraction, we capture register properties and hierarchical context, allowing our Register Collateral Generator (RCG) tool to automatically apply the "first, last, and random sample" filtering rule on a per-block basis. This translates a sophisticated UVM bring-up strategy into a minimal, automatically generated C test, eliminating manual lookups and ensuring synchronization with the golden RAL model.

Example: Generated C Code
From the CSV example above, the RCG would generate the following C code as shown in Figure 4. This generated collateral is then compiled alongside a generic C test driver. The driver simply iterates through the g_soc_registers array, performing access checks tailored to the access_policy and is_waived fields, thus bringing RAL-level intelligence directly into the processor's execution space.

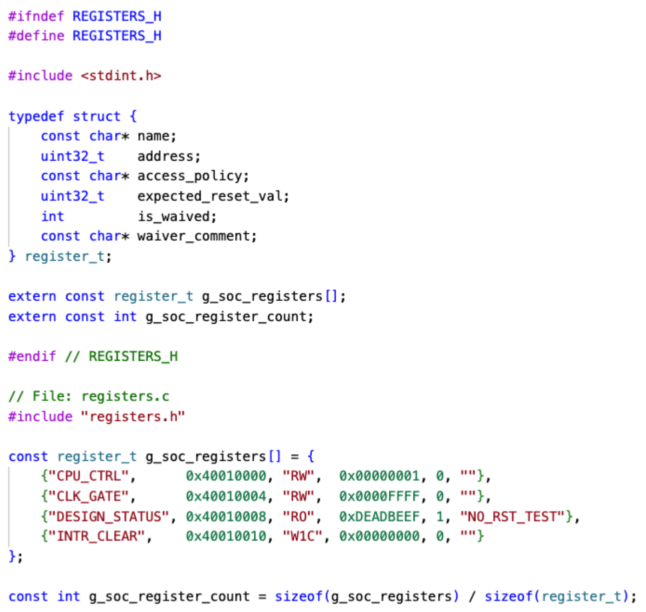
D. Stage 4: Continuous Integration and Automation

To ensure the C-test collateral remains perpetually synchronized with the reference register model, the entire generation process is automated within a Continuous Integration (CI) framework. The flow is not a manual, one-time event but a persistent, background process.A Cron job or a CI pre-submit hook (e.g., in Jenkins or GitLab) is configured to monitor the version control system for any changes to files defining the register address map. Upon detection of a committed change, the CI job automatically triggers the flow as shown in Figure 5. The resulting C-collateral files are checked into the repository, ensuring that any C-test compilation that follows will always use the most up-to-date register database. This "live" update mechanism eliminates the risk of test desynchronization and makes the system maintenance-free.



Register Name	Register Address	Register Access Policy	Reset Value	Waiver (Yes/ No)	Waiver Reason (if applicable)
CPU_CTRL	0x40010000	RW	0x00000001	0	
CLK_GATE	0x40010004	RW	0x0000FFFF	0	
DESIGN_STATUS	0x40010008	RO	0xDEADBEEF	1	NO_RST_TEST
INTR_CLEAR	0x40010010	W1C	0x00000000	0	

Figure 3: Generated Intermediate CSV Collateral containing Register Details from UVM RAL



```c
#ifndef REGISTERS_H
#define REGISTERS_H

#include <stdint.h>

typedef struct {
    const char* name;
    uint32_t    address;
    const char* access_policy;
    uint32_t    expected_reset_val;
    int         is_waived;
    const char* waiver_comment;
} register_t;

extern const register_t g_soc_registers[];
extern const int g_soc_register_count;

#endif // REGISTERS_H

// File: registers.c
#include "registers.h"

const register_t g_soc_registers[] = {
    {"CPU_CTRL",      0x40010000, "RW",  0x00000001, 0, ""},
    {"CLK_GATE",      0x40010004, "RW",  0x0000FFFF, 0, ""},
    {"DESIGN_STATUS", 0x40010008, "RO",  0xDEADBEEF, 1, "NO_RST_TEST"},
    {"INTR_CLEAR",    0x40010010, "W1C", 0x00000000, 0, ""}
};

const int g_soc_register_count = sizeof(g_soc_registers) / sizeof(register_t);
```

Figure 4: Example of generated C collateral

III. Applications and Advanced Capabilities

The true utility of this methodology is realized through its application to complex verification scenarios that are notoriously difficult to manage in traditional C testing.

A. *Intelligent Test Pruning via Rule-Based Filtering*

To manage the scale of modern SoCs, we employ a rule-based filtering algorithm during Stage 1. The generation sequence accepts inclusion/exclusion arguments to prune the final register list. The filtering algorithm operates on all registers in the UVM RAL model, enabling highly specific test suites, such as filtering by name, address range, access policy or metadata tag, as shown in Figure 6.This approach provides a powerful yet simple mechanism to create targeted sanity tests, full regression suites, or domain-specific tests from a single, reference source

B. *Targeted Verification using Metadata Tags*

Our flow extracts metadata "tags" from the UVM environment, primarily for managing registers across multiple reset domains. Registers can be tagged with a reset domain identifier in UVM, which our UVM sequence extracts and adds as a "Reset_Domain_ID" column to the CSV. The C test driver then uses this to perform targeted reset verification. For example, after a VDD_CORE reset, the C test validates reset values only for registers whose `reset_domain_id` matches VDD_CORE, enabling automated bare-metal reset domain testing of registers.

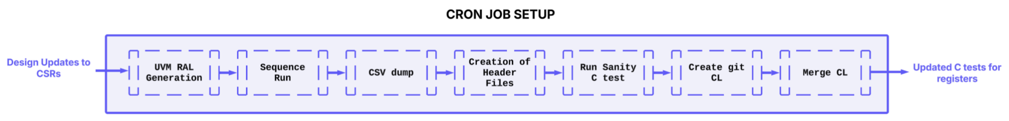


Figure 5: Cron job setup

```
uvm_reg all_registers[$];
ral_model.get_registers(all_registers, UVM_NO_HIER);

// --- 2. Iterate and Apply Filters for Each Register ---
foreach (all_registers[i]) begin
    uvm_reg current_reg = all_registers[i];
    bit should_include = 1'b1; // Assume inclusion by default

    // --- A. Filter by Name Pattern (Exclusion) ---
    if (filter_exclude_pattern != "" && str_match(current_reg.get_name(), filter_exclude_pattern)) begin
        should_include = 1'b0;
    end

    // --- B. Filter by Address Range (Inclusion) ---
    if (should_include && filter_include_range_start != -1) begin
        uvm_reg_addr_t addr = current_reg.get_address();
        if (addr < filter_include_range_start || addr > filter_include_range_end) begin
            should_include = 1'b0;
        end
    end

    // --- C. Filter by Access Policy (Exclusion) ---
    if (should_include && filter_exclude_access != "") begin
        if (current_reg.get_rights() == filter_exclude_access) begin
            should_include = 1'b0;
        end
    end

    // --- D. Filter by Metadata Tag (Inclusion) ---
    if (should_include && filter_include_tag != "") begin
        string reg_tag;
        // Tags are also managed via config_db for consistency
        if (uvm_config_db#(string)::get(null, current_reg.get_full_name(), "METADATA_TAG", reg_tag)) begin
            if (reg_tag != filter_include_tag) begin
                should_include = 1'b0;
            end
        else begin // If the tag doesn't exist for this reg, exclude it
            should_include = 1'b0;
        end
        end
    end

    // --- 3. Write to CSV only if all filters passed ---
    if (should_include) begin
        // (Extraction logic for name, addr, reset, waivers, etc. goes here)
        $fdisplay(csv_file, "%s,0x%0h,...", current_reg.get_name(), current_reg.get_address());
    end
end

// ... (File closing logic) ...
```

Figure 6: Rule Based Filtering

Case Study: Watchdog Timer (WDT) Registers

Consider a standard Watchdog Timer module. The generated CSV and subsequent C-array would look as shown in Figure 7 and Figure 8 respectively. The C test for the WDT has now the intelligence to cover the register access test for WDT_CUR_VAL register whenever the reset domain corresponding to POR_RESET is brought out of reset as part of multi-reset testing in the SoC.

Register Name	Address	Access	Reset Value	Is Waived	Waiver Comment	Reset_Domain_ID
WDT_CTRL	0x40080000	RW	0x00000000	0		POR_RESET
WDT_LOAD_VAL	0x40080004	RW	0x00FFFFFF	0		POR_RESET
WDT_CUR_VAL	0x40080008	RO	0x00FFFFFF	1	Volatile counter	POR_RESET
WDT_INT_CLR	0x4008000C	W1C	0x00000000	0		POR_RESET
WDT_INT_STAT	0x40080010	RO	0x00000000	0		POR_RESET

Figure 7: Watchdog Timer Generated CSV Collateral

```
// Part of the g_soc_registers array
{
    "WDT_CUR_VAL",        // name
    0x40080008,           // address
    "RO",                 // access_policy
    0x00FFFFFF,           // expected_reset_val
    1,                    // is_waived
    "Volatile counter",   // waiver_comment
    "POR_RESET"           // reset_domain_id (new field in struct)
}
```

Figure 8: Generated C collateral for Watchdog Timer registers

IV. Waiver Portability and Re-use

cornerstone of this methodology is its ability to create a seamless waiver pipeline from the IP level through Sub-System to SoC verification, ensuring maximum re-use and consistency.

A. Waiver Portability Flow

Waivers defined in an IP's local UVM environment are inherited by subsystem and SoC C tests, preventing re-discovery of known issues at higher integration levels. This flow includes:

- **IP-Level Waiver:** IP designers add waivers in their UVM testbench (using uvm_config_db) for benign deviations, passed to SoC via UVM sequences.
- **SoC RAL Integration:** The SoC RAL model instantiates the IP and its RAL model, reusing IP-level waivers in SoC UVM sequences.
- Automated Extraction: CI-driven UVM sequences on the top-level SoC RAL model discover and record uvm_config_db settings in a CSV.
- **C-Collateral Synthesis:** The RCG tool parses the CSV, setting \`is_waived\` to 1 and populating \`waiver_comment\` for the register.
- **SoC C-Test Execution:** The SoC C-test driver reads \`is_waived\`, skipping the check and logging it as waived.
- **SoC-Level Override:** SoC UVM sequences capture overrides, documented in the CSV with \`is_soc_waived\` set to 1, causing C tests to ignore these registers for testing.

This formal mechanism ensures knowledge preservation throughout the integration chain.

V. Results and Impact

The implementation of this methodology at multi-core CPU subsystem level yielded substantial, measurable improvements across key verification metrics as summarized in Table 1.

A. Analysis of Defects Found

The high-fidelity nature of the generated tests allowed for the confident identification of genuine hardware defects, including:

- **Incorrect Reset Values:** The most common defect class, critical for device boot-up.
- **Faulty Access Policies:** Registers specified as RW behaving as RO, or W1C bits failing to clear.
- **Architectural Path Failures:** Dropped or misrouted transactions from the processor to the target register due to interconnect or bridge misconfiguration or remapper bugs.
- **Security Flaws:** Privileged registers being accessible by non-privileged requests

B. Reduction in False Failures

The system's awareness of register metadata virtually eliminated common false failures that plague traditional C tests, such as:

- Attempting to verify a read-back value from a Write-Only (WO) register and write operations on a Read-Only (RO) register.
- Flagging a mismatch on a volatile status register that has no defined reset value.

Table 1: Results

Metric	Conventional C-Test Approach	UVM-to-C Methodology	Improvement
Total Registers in Design	> 20,000	> 20,000	-
Registers in Critical Test	~20,000 (unfiltered)	~8,000 (filtered)	60% Reduction
Regression Runtime	~4 hours	1.5 hours	62.5% Reduction
Initial Manual Effort	~200 hours per project	~20 hours (one-time setup)	90% Reduction
Maintenance Effort with Ongoing Design Updates	~1 day per update	0 (automatically updated)	1 day reduction per design update
False Failure Rate	High (~35% of failures)	1% of failures	> 93% Reduction
Register Coverage	Estimated ~70-80%	98% of reachable regs	> 22% Boost

- Failing a test on a register that was intentionally waived at the IP level for a known design behavior.

VI. Limitations and Future Work

While highly effective, the current methodology has areas for future enhancement. The static generation flow does not inherently handle dynamic register maps or complex, multi-step access sequences (e.g., indirect addressing that requires writing an address to one register to access data through another). Future work will focus on extending the C-test driver's intelligence to interpret more complex access policies directly from the generated collateral. Additionally, we are exploring the generation of more sophisticated C test sequences (beyond simple reset and access checks) by embedding sequence logic within the UVM extraction phase itself.

VII. Conclusion

This paper has presented a robust, scalable, and automated methodology for C-based hardware register verification. By systematically leveraging the UVM RAL model as a reference source, proposed flow closes the critical gap between high-level simulation and processor-centric validation. It transforms C-based register testing from a brittle, manual task into a disciplined, efficient, and highly accurate verification strategy. The demonstrated gains in regression time, debug efficiency, false failure reduction, bug-finding capability underscore the value of this approach in navigating the complexities of modern SoC verification and accelerating the path to market.

References

[1] Verification Academy – UVM Cookbook: https://verificationacademy.com/cookbook/registers/integrating
[2] Universal Verification Methodology (UVM) 1.1 Users Guide – Accellera, May 18, 2011
[3] Universal Verification Methodology (UVM) 1.2 Users Guide – Accellera, October 8, 2015
[4] Verification Academy Coverage Cookbook: https://verificationacademy.com/cookbook/coverage
[5] Vijayakrishnan Rousseau, Satyajit Sinari, Benjamin Applequist, Timothy McLean, Geddy Lallathin, “Automated Generation of RAL-based UVM Sequences, DVCON US 2020”