2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Sequencer Coverage Exclusion Optimiser: Streamlining Coverage Closure in Dynamic Sequencer-Based Designs

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

Preethi Ashok Kumar, Google IT Services India Private Limited, Bangalore, India
(preethiashok@google.com)

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

Yashas Uppoor, Google IT Services India Private Limited, Bangalore, India
(yashasuppoor@google.com)

***Abstract*-A significant challenge in verifying complex sequencer-based System-on-Chip (SoC) designs lies in correlating low-level program counter (PC) coverage with high-level language (HLL) source code. The dynamic nature of PC spaces and the manual effort required for this mapping hinder efficient coverage analysis and exclusion management. This paper introduces an automated approach to bridge this gap by leveraging compiler output to link PC execution to the original HLL source. This method facilitates precise identification of uncovered code and enables exclusion management based on HLL source code references rather than volatile PC addresses. This automation streamlines coverage closure, reduces manual effort, and improves the accuracy of verification workflows across different Electronic Design Automation (EDA) environments.**

Keywords—Sequencer-based designs; Code Coverage; Automation; Coverage Exclusions;

I. Introduction

The escalating complexity of contemporary System-on-Chip (SoC) designs, particularly those employing intricate sequencer-based architectures that execute instructions defined in high-level languages (HLLs), presents formidable challenges to the domain of functional verification. As a standard design flow, HLL code, example shown in Figure 1, undergoes a transformation into program counter (PC) instructions, which are subsequently implemented in Register Transfer Level (RTL) hardware. This crucial translation process encompasses a compilation step, the byproduct of which is a detailed compilation log that inherently serves as a bridge between the software and hardware domains by documenting the mapping between the HLL source and the generated PC instructions.

During the verification phase, comprehensive RTL code coverage analysis, with a specific focus on PC coverage, is essential to ensure that all intended control flow branches and execution paths are adequately exercised. However, a significant impediment arises from the frequent disconnect between the granular PC-level coverage data obtained during this analysis and the original, semantically richer HLL source code. This semantic divide creates a highly inefficient verification loop. Any modification to the source HLL triggers a shift in the PC address space, rendering all previously established coverage exclusions null and void. This forces engineers to repeat the entire labor-intensive exclusion analysis process from scratch with each code update, significantly impacting project velocity. This design flow can be summarized as shown in Figure 1.

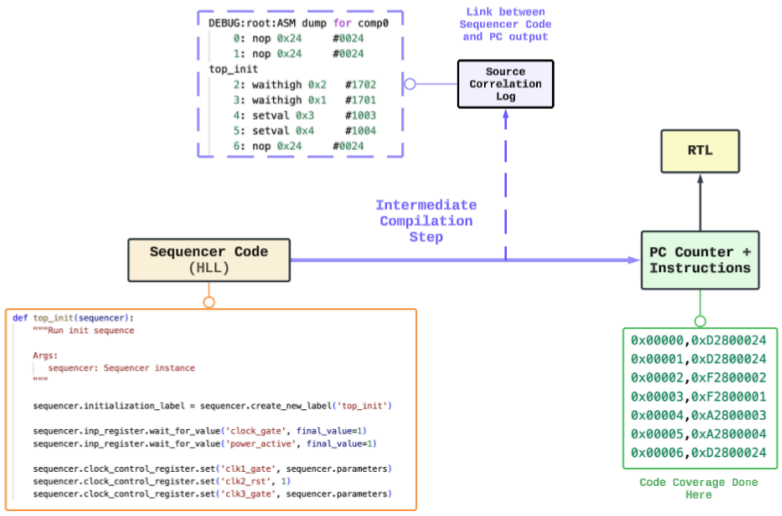

Figure 1: Design Flow in Sequencer-based Designs

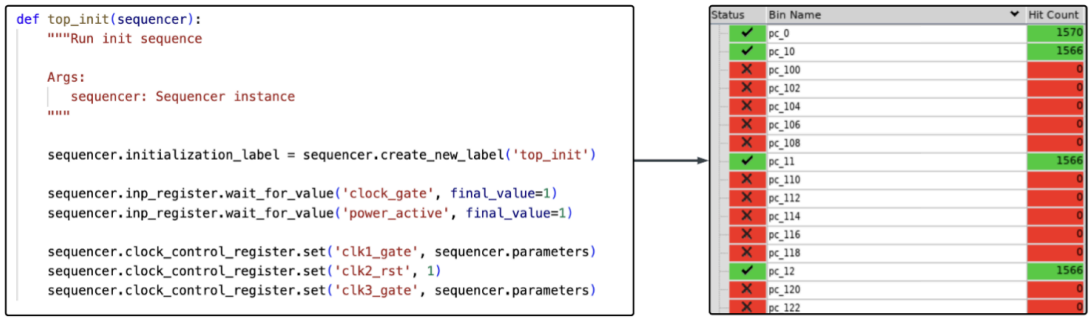

Figure 2: Intelligible Source Code (left) vs. Opaque PC Coverage Data (right)

II. Problem Statement

The verification of custom hardware sequencers relies on tracking execution at the Program Counter (PC) level. This methodology, while offering granular accuracy, identifies coverage gaps as nothing more than a list of numerical PC addresses, which lack any inherent link to the source code's logic or intent, as shown in Figure 2. The core problem is the semantic gap between this raw machine-level data and the human-authored source code—be it a High-Level Language (HLL) or assembly. This gap presents a significant and recurring analytical challenge to the verification process:

A. Semantic Opacity of PC Addresses

A PC address, such as 0x41B8, is an artifact devoid of investigative context. Its interpretation is a primary bottleneck in coverage analysis because the address alone fails to provide the necessary information for debugging or sign-off.

- **Functional Context:** A PC address does not indicate the functional role of the corresponding instruction. Whether it belongs to initialization, core logic, or an error-handling path is only discernible from the source code's structure, labels, and comments.
- **Relational Context:** From a list of uncovered addresses, it is impossible to determine if they constitute a single functional gap (e.g., one un-entered if clause) or multiple independent deficits. This ambiguity prevents engineers from assessing the architectural significance of a coverage hole.
- **Causal Context:** The address offers no insight into the cause of a coverage failure, be it a deficit in the stimulus, a design flaw, or unreachable code. Effective diagnosis requires understanding the code's original intent.

B. Compounding Effect of Address Space Volatility

This challenge of PC interpretation is severely compounded by the inherent instability of the PC address space. The assembly or synthesis process that generates the final machine code is deterministic only for a given source and build configuration. Any modification to the source code or build parameters triggers a regeneration of the instruction memory, which routinely invalidates the entire PC map. Consequently, all prior PC-based analysis becomes obsolete, forcing engineers to repeatedly re-investigate opaque data with every design iteration.

C. Non-Portable and Unreliable Exclusions

The integrity of verification sign-off requires waivers to be methodically justified, yet anchoring them to transient PC addresses is an untenable practice. Any source code change can shift the PC space and silently invalidate prior analysis, creating high-risk coverage regressions. A robust methodology must anchor exclusions to their only source of truth: the source code itself.

Our automated solution achieves this by creating a persistent PC-to-source mapping, transforming raw coverage data into an actionable framework where waivers become stable, portable assets. This allows justified exclusions to be automatically ported to mature revisions of a design after bug fixes. Critically, it also enables the entire waiver suite to serve as a validated starting point for new derivative projects, ensuring that vital verification knowledge is preserved and reused, not rediscovered with each development cycle.

D. Limitations of Current Verification Methodologies

Modern EDA verification tools from vendors like Cadence, Synopsys, and Siemens, while sophisticated, are designed for standard languages and architectures. Their coverage analysis and debug features are not built for the unique challenges of custom hardware sequencers with proprietary instruction sets, creating a tooling gap that this paper's framework addresses.

- **Lack of Native Support for Custom Instruction Sets:** Commercial tools like VCS, Xcelium, and Questa excel at linking coverage back to the source for standard HDLs, and many support software-driven verification for known CPU architectures e.g., Reduced Instruction Set Computer (RISC)-V, Advanced RISC Machine (ARM). However, they lack innate understanding of project-specific instruction sets or proprietary toolchains, rendering advanced source-level debug and analysis ineffective, and only reporting raw, context-less PC addresses.
- **Inadequate and Non-Portable Waiver Management:** Existing tools' waiver mechanisms, designed for standard language constructs, inadequately handle custom sequencers. Waivers tied to dynamic PC addresses become invalid after re-synthesis, leading to repetitive manual re-validation. Their "waiver porting" intelligence, meant for HDL code changes, doesn't apply here. Also, waivers cannot be ported across hierarchies, from Intellectual Property (IP) to Subsystem to SoC or, from an original project to a derivative one easily.

The framework proposed in this paper directly addresses a critical gap in commercial tooling: the inability to automatically map coverage data from custom sequencers to their source code and manage exclusions in a stable manner.

III. Solution

To solve the challenges of PC-based coverage analysis, an automated framework has been developed that creates a direct, actionable link between low-level coverage data and the human-authored source code. Proposed solution does not modify the core simulation or coverage collection flow; instead, it operates as a post-processing utility that ingests standard verification artifacts to produce an intelligible, source-centric coverage report.

The framework's workflow, depicted in Figure 3, is predicated on two key inputs:

- **PC Coverage Database:** The raw list of covered and uncovered Program Counter (PC) addresses. This data is extracted directly from the final coverage database (e.g., a Verification Database or, VDB file) shown in Figure 2.
- **The Source Correlation Log File (Compiler Log):** A critical artifact produced by the tool that synthesizes or assembles the source code (e.g., C++, Python, or assembly) into machine code for the sequencer, shown in Figure 1. This log file contains the explicit mapping between each generated PC address and the source file and line number that produced it.

The flow generates a Comma Separated Values (CSV) file as output, which contains the mapped uncovered HLL code as shown in Figure 4.

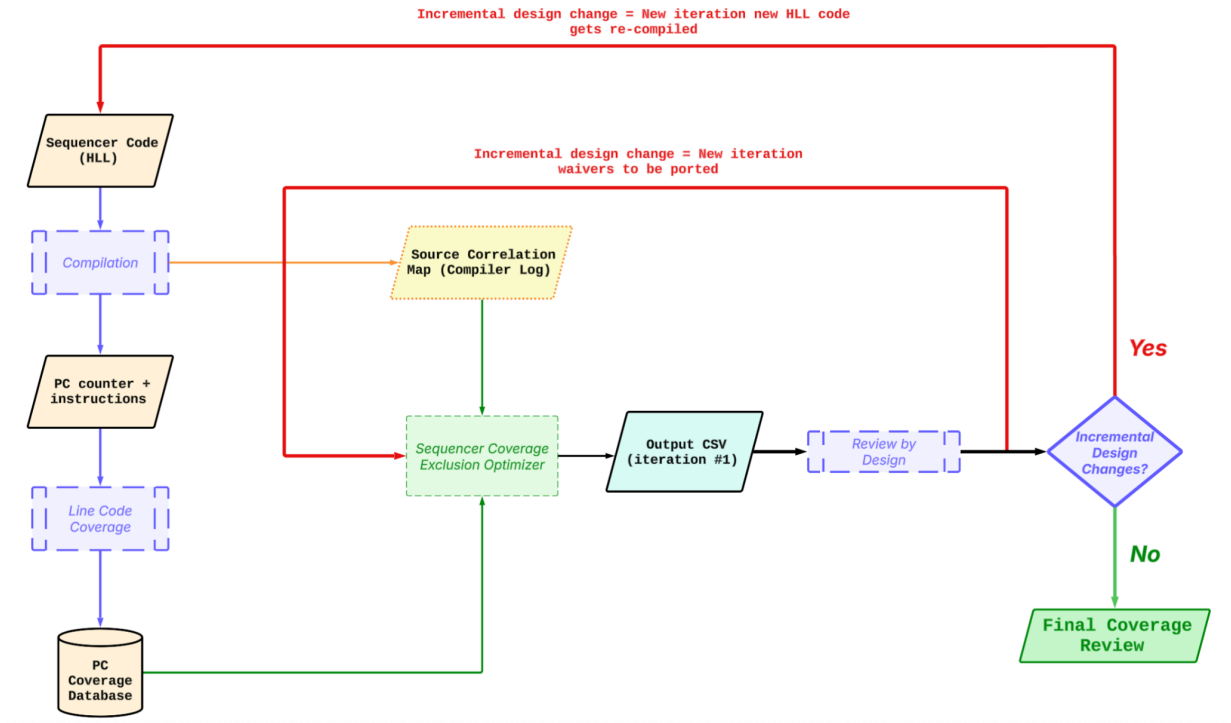

Figure 3: Proposed Solution

Sequencer HLL Source Code	All PCs and Instructions in Code Section	Uncovered PCs	Waiver decision
def top_init(sequencer): """Run init sequence Args: sequencer: Sequencer instance """ sequencer.initialization_label = sequencer.create_new_label('top_init') sequencer.inp_register.wait_for_value('clock_gate', final_value=1) sequencer.inp_register.wait_for_value('power_active', final_value=1) sequencer.clock_control_register.set('clk1_gate', sequencer.parameters) sequencer.clock_control_register.set('clk2_rst', 1) sequencer.clock_control_register.set('clk3_gate', sequencer.parameters)	DEBUG:root:ASM dump for comp0 0: nop 0x24 #0024 1: nop 0x24 #0024 top_init 2: waithigh 0x2 #1702 3: waithigh 0x1 #1701 4: setval 0x3 #1003 5: setval 0x4 #1004 6: nop 0x24 #0024	2, 3, 4	Test gap to be covered Filled by Design Team after Review

Figure 4: Final CSV output showing uncovered HLL code sections

A. Core Analysis and Mapping Engine

The core of the proposed tool is an analytical engine that systematically correlates the raw PC data with the source code via the source correlation log. For every uncovered PC address provided by the coverage database, the engine performs a lookup in the source correlation log to find its corresponding source location.

For example, consider a coverage tool reporting the following uncovered PCs 0x1A04, 0x1A05, 0x2B10. The engine parses the source correlation log, which may contain entries like Figure 1. The engine aggregates this information to build a comprehensive view of the coverage holes, pinpointing exactly which source code blocks were not executed as shown in Figure 4.

B. Collaborative CSV-Based Reporting

The primary output of the framework is a structured Comma-Separated Values (CSV) file designed for collaborative analysis and waiver management. This CSV report provides a granular, source-level view of all coverage gaps. Each row in the CSV corresponds to a logical block of source code containing one or more uncovered instructions. The CSV file is structured with the following key columns: component name, block hierarchical name, code snippet with PC mapping, uncovered PCs, as shown in Figure 4.

This report is delivered to the design and verification teams. Its clear format allows engineers to instantly see the exact code in question without needing to manually cross-reference PC values. The final column, "Waiver Rationale," is initially left blank. The design team analyzes each uncovered code block and fills in this column with their justification for any waivers (e.g., "Feature disabled in this configuration," or "Redundant check, unreachable by design"). This completed CSV becomes a durable, human-readable waiver artifact. Because the waivers are justified against specific source code lines and blocks, they remain logically valid even when PC addresses shift across builds. This framework can then use this completed CSV to automatically port waivers to future coverage runs, providing a stable, source-based exclusion mechanism that is essential for achieving efficient and reliable coverage closure.

IV. Intelligent Waiver Porting

A primary challenge in long-duration verification projects is preserving the integrity of the coverage exclusion database as the source code itself evolves. Our framework directly confronts this by providing a robust and intelligent exclusion porting mechanism. This capability leverages the Waiver Analysis Ledger (the generated CSV) as a durable, source-based artifact, ensuring that waivers remain valid across incremental bug fixes, feature additions, and even migration to derivative projects.

The porting engine's logic is designed to be resilient to common code modifications. It operates not on volatile PC addresses, but on a stable "source signature" derived from the source file, line numbers, and the code content itself (e.g., a function label or code block). This allows the engine to locate the correct context for a waiver even when surrounding code has changed. Upon finding a match, the engine automatically transcribes the waiver information. The (Auto-ported) tag provides clear traceability, indicating that the waiver was applied by the tool and not through new manual analysis as shown in Figure 3.

A. Porting to Derivative Projects

The most significant application of this methodology is in accelerating the verification of new, derivative designs. Instead of starting coverage analysis from scratch, a new project team can use the Waiver Analysis Ledger from a previous, completed project as a validated starting point. The porting engine is run using the reference ledger from the parent project against the initial coverage report of the derivative project. It will automatically find and apply all waivers for code blocks that have been inherited without modification. This single step can resolve a substantial percentage of the total coverage holes on day one, allowing the verification team to immediately focus their resources on the actual new or modified functionality of the derivative design. This transforms the waiver database from a project-specific artifact into a reusable, high-value asset that compounds its return on investment (ROI) across a product family.

B. Hierarchical Waiver Porting (IP to Subsystem to SoC)

Proposed framework facilitates a highly efficient, multi-level verification strategy.

- An IP design team can develop a sequencer and deliver a "golden" waiver CSV along with the IP, certifying the coverage status for their standalone unit.
- The sub-system integration team can then automatically port these IP-level waivers into their environment. Their analysis is immediately narrowed to only the new coverage issues that arise at the interfaces between IP blocks.
- Finally, the top-level SoC team inherits the justified waivers from the sub-system, allowing them to focus solely on chip-level integration issues. This hierarchical approach prevents the redundant re-verification of IP internals at each stage, saving hundreds of engineering hours and allowing teams to focus on their specific integration work.

C. Absorbing Incremental Source Code Changes

The waiver porting is most powerful when handling typical, ongoing development activities. The engine can successfully port existing waivers in scenarios like:

- **Bug Fixes in Unrelated Code:** Imagine a waiver exists for an error-handling block in function_A because it is deemed unreachable. Later, a bug is fixed in a completely separate function_B, which does not call function_A. While this fix will likely shift the PC addresses for all subsequent code, including function_A, our tool will still port the waiver correctly. It identifies the function_A block by its unique source signature, finds the new, shifted PC addresses for it, and correctly applies the original waiver rationale.

- **Addition of New Features:** A developer might add a new function_C to the source file, which is called after function_A and never calls function_A. This addition changes the file's structure and invalidates all old PC addresses for code that appeared before it. However, waivers for existing functions are unaffected because the porting engine locates them by their stable source context, not their transient hardware addresses

V. Applications and Scalability

The utility of this automated source-mapping framework extends significantly beyond a single use case, providing strategic value across diverse technology domains and at multiple stages of the SoC development lifecycle. Its ability to create an intelligible and stable bridge between source code and machine execution facilitates broad applicability, project-level scalability, and cross-organizational reuse.

A. Broad Applicability Across Diverse Domains

The fundamental problem of correlating low-level execution with high-level intent is ubiquitous in modern computing. Proposed framework is therefore directly applicable to any domain where source code is compiled or synthesized into a PC-addressable instruction set, and where coverage closure is critical.

- **Embedded Systems and Read Only Memory (ROM) Firmware:** For processors executing safety-critical C/C++ code, such as Interrupt Service Routines (ISRs) in an Arm Cortex-M core, our tool is essential. It provides the definitive link needed to ensure that all specified execution paths, particularly complex error-handling conditions, have been tested at the machine-code level.

- **Custom Hardware Accelerators:** These designs often rely on bespoke microcode sequencers to execute application-specific tasks. The framework provides crucial context to the cryptic micro-instructions, allowing verification engineers to confirm that the high-level algorithmic description has been implemented and tested exhaustively.

- **Graphics Processing Units (GPUs) and DSPs:** In these highly parallel domains, this tool can map low-level shader assembly or DSP instruction coverage back to the original source language (e.g., GLSL, CUDA, or C). This allows for rigorous verification of complex mathematical algorithms, ensuring that the implementation is fully exercised.

B. Scalability and Value within the Project Lifecycle

The framework enhances the verification process by automating regression analysis and maintaining sign-off velocity. Automated waiver porting in nightly/weekly regressions filters noise by reapplying known waivers to new builds, focusing engineers on new coverage gaps instead of re-investigating old issues. By creating a stable, source-based waiver artifact, the framework brings predictability to coverage closure, mitigating last-minute waiver re-analysis before tape-out and ensuring a smoother path to final sign-off.

VI. Results

To validate its effectiveness, the proposed framework was deployed on two large-scale, industrial System-on-Chip (SoC) projects. Prior to this deployment, the verification teams for both projects were confronted with a significant number of raw, un-analyzed coverage deficits reported only as Program Counter (PC) addresses. The objective was to use the framework to automatically process these deficits, map them to their source code origins, and quantify the impact on the verification closure process. The key metrics from the two deployments were captured and are summarized in Table 1. The deployment provided immediate and actionable results by successfully mapping 100% of the previously obscure PC-level deficits back to the source code—1,148 distinct blocks for Project A and 969 for Project B. This complete, source-level view enabled the engineering teams to rapidly and accurately classify the root cause of each coverage hole for the first time. It provided precise, actionable feedback both to the design team for dead code removal and to the verification team for developing new, targeted tests to exercise missed functionality.

By clearly distinguishing between unreachable code and genuine test gaps, both teams were able to increase their number of justified coverage waivers by approximately 30%. The automation of the entire mapping and analysis process eliminated the need for manual correlation. The results demonstrate that the tool successfully transforms raw

TABLE I
RESULTS

Sl. No.		Project 1	Project 2
	Metric	(Subsystem #1)	(Derivative of Subsystem #1)
1	Total Uncovered Instructions	2796	2333
2	Total Exclusions Identified	1148	969
3	Percentage increase in justified coverage waivers	~30%	~30%
4	Dead Code Blocks Identified	52	12
5	Test Gaps Identified	44	16
6	Engineering Weeks Saved	~6 weeks	~6 weeks

data into actionable insights, leading to a direct reduction in development time, an enhancement in product quality, and a significant return on investment.

VII. LIMITATIONS

The methodology's effectiveness relies on the nature of source code modifications. Automated porting is suitable for incremental code evolution, such as additions, modifications, and localized refactoring. However, it is not intended for major architectural rewrites or the complete deletion of waiver-associated code. In such cases, manual review is necessary, similar to how line coverage is invalidated with major rewrites. The accuracy of this process also depends on the quality of the compiler/synthesis tool's log file. Furthermore, new toolchains will require a dedicated parser for their specific log file format.

VIII. CONCLUSION

This paper introduces an automated framework to resolve the conflict between opaque Program Counter (PC) coverage data and human-authored source code. It systematically parses compiler logs to map raw PC addresses to source lines, transforming unintelligible coverage reports into actionable CSV-based analytical artifacts. This eliminates manual PC-to-source correlation and automatically ports justified waivers across builds, preserving engineering knowledge and preventing rework due to unstable PC addresses. This allows verification engineers to confidently identify unreachable dead code and focus on improving functional stimulus. Strategically, this methodology establishes a scalable and portable exclusion strategy by anchoring waivers to source code, facilitating hierarchical verification where exclusions propagate from IP to sub-system and SoC levels. This significantly reduces manual effort (saving several engineer-weeks per project) and provides a more accurate, predictable, and efficient path to final coverage sign-off.

REFERENCES

[1] K. Prabha , N. Rohit, G. Amit and T Bipul, “Understanding UVM Coverage for RISC-V Processor Designs,”, Synopsys White Paper, pp. 4–6, May 2023
[2] T.C. Surendhar, L Preetham, A Ashwani, L Youngchan, K Youngsik, B.C. Seonil “Achieving Faster Code Coverage Closure using High-Level Synthesis, DVCON Europe, 2021
[3] A. Tan, “Closing Coverage in HLS”. {Online}. Available: https://semiwiki.com/eda/7757-closing-coverage-in-hls/.
[4] Synopsys – VCS. {Online}. Available: https://www.synopsys.com/verification/simulation/vcs.html
Cadence – Xcelium Logic Simulation. {Online}. Available: https://www.cadence.com/ko_KR/home/tools/system-design-andverification/simulation-and-testbench-verification/xcelium-simulator.html8