Accelerated Coverage Closure with Emulation: Covering Real-Time Use Case Corners

Bichu Sajeev
Multimedia
Qualcomm India Private Limited
Bangalore, India
bsajeev@qti.qualcomm.com

Vivek Tiwari
Multimedia
Qualcomm India Private Limited
Bangalore, India
vivetiwa@qti.qualcomm.com

***Abstract*- In hardware verification, comprehensive coverage is essential for ensuring design reliability and functionality. Traditional simulation methods are effective but slow and resource-intensive. Emulation offers faster execution, making it ideal for running complex, high-resolution scenarios. However, the lack of a structured matrix to measure use case scenarios often results in incomplete validation of the design, especially when legacy content is reused. To address this, an emulator-based coverage framework is proposed to improve verification efficiency. It enables both code and functional coverage during emulation runs. In addition to coverage, assertion checks are integrated into the emulation flow to monitor real-time design behavior, enabling early detection of functional issues during high-speed execution. Randomized high-resolution tests are used to target corner cases and hard-to-reach coverpoints. This approach helps shift coverage and verification closure earlier in the cycle. Achieving near 100% RTL coverage increases confidence in the design.**

Keywords— Emulation-Based coverage, HDL, UCDB, HVL.

I. INTRODUCTION

Modern hardware designs demand scalable and efficient verification strategies to ensure functional correctness and robustness. As test complexity increases, traditional simulation environments face limitations in speed and capacity, making them less effective for validating real-world or high-resolution Use Case (UC) scenarios. Emulation offers a practical alternative by enabling faster execution and broader scenario coverage. To fully leverage this capability, we introduce a coverage-driven framework tailored for emulation. This framework captures both code and functional coverage during emulation runs, enabling validation of complex, corner-case scenarios that are difficult to reach in simulation.

There are two primary approaches to implementing functional coverage in emulation. One approach is to embed the coverage logic directly within the HDL, which requires the coverage module to be written in a synthesizable manner and may increase emulation board resource usage. The second approach involves using transactors to collect coverage data at the testbench level, but this can significantly slow down emulation due to cycle-by-cycle data transactions between the host and the emulator. In our work, we adopt the first approach to balance accuracy and performance while maintaining efficient emulation runs. By using coverage as a metric to monitor verification progress and completeness, this framework supports early signoff and high-confidence validation of complex hardware systems.

To further enhance the verification process, assertion checks are integrated into the emulation flow. These assertions enable real-time monitoring of design behavior during high-speed execution, allowing early detection of functional issues that may not surface in slower simulation environments.

II. TRADITIONAL METHOD

In a traditional simulation-based coverage setup, both code and functional coverage are collected during RTL simulation runs. Code coverage includes metrics like line, toggle, and branch coverage, which help assess how thoroughly the RTL code has been exercised. Functional coverage is implemented using SystemVerilog covergroups, which are triggered by specific signal values or events in the testbench. These coverpoints are sampled during simulation to track whether key functional scenarios have been exercised. The coverage data is typically stored in UCDB or similar formats and analyzed using coverage tools to identify gaps. This setup is tightly integrated with the simulation environment, making it easy to debug and iterate. However, due to the slow nature of simulation, achieving full coverage—especially for complex or long-running scenarios—can be time-consuming. Additionally, simulation-based coverage is limited by the scope and depth of the testbench, which may not fully reflect real-world use cases. As a result, traditional setups often struggle to scale with increasing design complexity and verification demands. Assertion checks in simulation are used to validate design behavior by monitoring specific conditions and triggering violations when expectations are not met. However, their effectiveness is often constrained by simulation speed and runtime limitations, making it difficult to catch assertion failures that occur under rare timing conditions or extended test sequences.

III. LIMITATION OF SIMULATION ONLY COVERAGE AND ASSERTION

Relying solely on simulation for hardware verification presents several critical limitations, particularly when validating real-world Use Case (UC) scenarios. Simulation environments are inherently long-running and resource-intensive, making them unsuitable for executing high-resolution stimulus, scenarios with bulky transactions regressing DUT with mix of all length from min to max, or large-scale UC tests. This often results in incomplete validation, especially for complex corner cases and buffer fill conditions that are essential for ensuring functional correctness.

Moreover, Firmware-integrated scenarios are executed exclusively in emulation, not in simulation, enabling thorough validation of real-world corner cases and buffer fill conditions. Simulation presents limitations in achieving complete coverage closure for third-party IP cores, particularly when it comes to high-resolution and complex use cases. Since these IPs are typically verified only at the top level.

Additionally, there is no structured mechanism to track and validate test intent for full or high-resolution UC scenarios, making it difficult to ensure that the design has been thoroughly verified. The reuse of legacy content across design generations further complicates the verification process, as there is no standardized matrix to assess completeness across iterations. The absence of a unified view that integrates results from both simulation and emulation environments also limits the ability to confidently sign off on the design. While coverage data is a valuable tool for measuring verification progress, it alone cannot guarantee completeness without a comprehensive and scenario-driven validation strategy.

Furthermore, assertion checks in simulation are limited by execution speed and runtime constraints, making it difficult to observe violations that occur under rare timing conditions or extended test sequences. This restricts their effectiveness in capturing subtle functional issues that may only emerge in real-world or system-level scenarios.

IV. PROPOSED METHODOLOGY

To enhance verification efficiency and improve design validation, an emulator-based coverage framework is proposed. This approach, as shown in Figure 1, enables the execution of high-resolution and long-running tests in the emulation environment, where both code coverage (including toggle, statement, and branch) and functional coverage (covergroup-based) are enabled. A wide range of high-resolution tests are executed to target hard-to-reach coverpoints and corner-case scenarios, ensuring deeper exploration of the design space. Coverage data generated from these emulation runs is captured in Unified Coverage Database (UCDB) files. To provide a comprehensive view of verification completeness, UCDBs from both simulation and emulation are merged into a single, unified coverage report. This methodology ensures that real-world use case scenarios are thoroughly exercised, supporting faster and more confident design signoff.

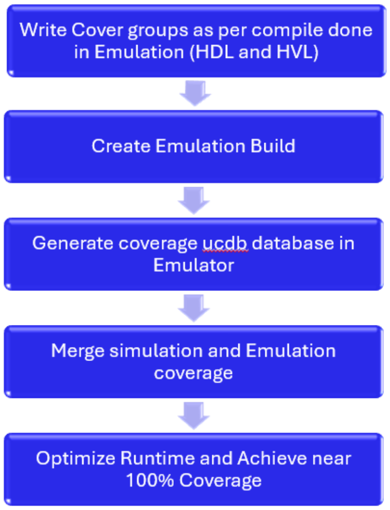
Write Cover groups as per compile done in Emulation (HDL and HVL)
Create Emulation Build
Generate coverage ucdb database in Emulator
Merge simulation and Emulation coverage
Optimize Runtime and Achieve near 100% Coverage

Figure 1- Proposed Methodology

In addition to coverage, assertion checks are integrated into the emulation flow to monitor real-time design behavior. These synthesizable assertions enable early detection of functional issues during high-speed execution, complementing coverage metrics and strengthening overall design validation.

V. IMPLEMENTATION DETAILS

The coverage setup in this framework shown in Figure 2 is divided into two parts: the HDL side and the HVL side. An interface is created that includes RTL signals, which are used in functional coverage coverpoints. On the HDL side, this setup is added to the hdl_top module to sample coverpoints using the defined interface. On the HVL side, the setup is integrated into the testbench environment to sample coverage based on testbench signals and structures. This dual setup generates two separate UCDB files—one for HDL coverage and one for HVL coverage. These files are then merged to produce a single, unified coverage report, providing a complete view of the design's verification status.

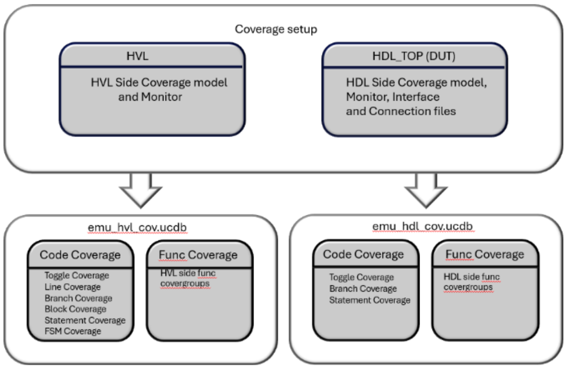

Figure 2- *HDL and HVL TB setup*

HDL Coverage setup coding Guidelines

1. Define functional coverage using covergroups inside a module instead of class, with triggering controlled by a sampling signal to determine when coverage is collected.
2. Specify coverpoints for the intended signals, including bins to track different value ranges or specific conditions relevant to the verification goals.
3. Define an interface for the DUT to group signals required for functional coverage.
4. Implement a coverage monitor by connecting coverage interface signals to DUT signals and defining logic conditions to capture various functional states for coverage analysis.
5. Instantiate the functional coverage module in the DUT using an interface handle, enabling access to and interaction with the interface signals for coverage monitoring.

HVL Coverage setup coding Guidelines

1. Declare and instantiate the covergroup inside the class, including the required coverpoints and bins.
2. Define a task inside the class that continuously monitors functional coverage conditions and assigns values to coverpoints accordingly
3. Create an instance of the functional coverage class and the coverage monitor in the environment to enable monitoring and data collection.

Emulation Assertion coding Guidelines

1. Define a dedicated assertion module that uses a SystemVerilog interface to access DUT signals, keeping the assertion logic modular and reusable.
2. Create an interface containing all required DUT signals for assertion checks, and ensure at least one signal is declared as input or output to prevent optimization during synthesis or emulation compilation.
3. Write synthesizable assertion logic inside the module using constructs like property, sequence, and assert. Avoid non-synthesizable elements or procedural blocks to maintain emulation compatibility.
4. Instantiate both the interface and the assertion module within the DUT, connecting the interface to DUT signals and passing the interface handle to the assertion module for real-time behavior monitoring.

Options to Enable Coverage in Veloce Emulation

1.Code Coverage:

Maintain the coverage hierarchy and specify the required coverage points in a single configuration file.
Eg Config file content(code_coverage_file.txt):

```
+cover=sb+test_bench.dut.u_module_1+R
+cover=sb+test_bench.dut.u_module_2+R
+cover=t+test_bench.dut.u_module_3+R
```

The following option must be passed from the Makefile to enable code coverage during emulation.

```
“runcmd_args+=\”VELOCE_CODE_COVERAGE_FILE=$code_coverage_file\””;
```

2.Functional Coverage:

To enable functional coverage during Veloce emulation, the following runtime arguments must be passed to configure the simulation environment appropriately.

```
-runcmd_args+=”TBX_CONFIG_OPTS_+= rtlc ”
-runcmd_args+=”TBX_CONFIG_OPTS_+=’-enable_fcov_support;’”
```

3.Emulation Assertion

1. At a minimum, add the following line to your veloce.config file:
   ```
   comp -assertcomp
   ```
   Add assertion analysis options as needed:
2. Single-step compile, add the following to the veloce.config file:
   ```
   veanalyze -assert_analyze_options <options>
   vhanalyze -assert_analyze_options <options>
   ```

Coverage Merge Command

To consolidate functional coverage data generated from both Hardware Verification Language (HVL) and Hardware Description Language (HDL) sources into a unified UCDB (Unified Coverage Database) file, as illustrated in Figure 3, the VCover tool can be used to perform a merge operation using the appropriate command
Eg: Vcover merge -input <coverage_list_file.list> <output_merged_coverage.cov>

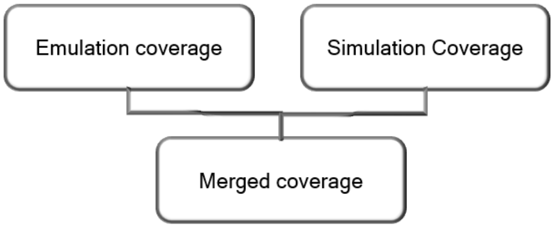

Figure 3. Final Coverage UCDB creation

VI. LIMITATIONS

Emulation-based coverage analysis is subject to specific limitations that must be considered to ensure valid and supported results, as outlined below:

- Toggle coverage builds must be executed independently and should not be combined with Statement or Branch coverage builds.
- Only Toggle, Statement, and Branch coverage types are supported in emulation-based code coverage; no other types are currently supported.
- Functional coverage on the HVL side cannot utilize interfaces to access RTL signals during emulation coverage.
- A single coverpoint is limited to a maximum of 4096 bins.
- Coverpoint expressions exceeding a size of 31 bits are not supported.
- Coverage on complex data types, such as struct and two-dimensional variables, is not supported.
- Coverage within class definitions is not supported.

VII. RESULTS

The proposed emulator-based coverage flow has been successfully developed and deployed in Camera projects, where it has demonstrated significant improvements in both verification efficiency and design quality. By integrating the flow early in the design and verification cycle, the methodology enables proactive detection and resolution of functional issues, ultimately reducing overall verification turnaround time. One of the key efficiency gains observed is a reduction of approximately 30% in simulation-directed tests, made possible by reusing emulation-based scenarios for coverage analysis. This reuse streamlines test development and improve resource utilization across platforms. In addition to efficiency, the flow enhances verification quality by introducing a structured matrix that maps high-resolution and firmware-driven tests to their intended verification objectives. This matrix enables systematic validation of real-world use cases and ensures that test intent is clearly defined and thoroughly exercised. As a result, the methodology has achieved close to 100% Register-Transfer Level (RTL) coverage, significantly increasing confidence in the functional correctness and completeness of the design. This high coverage metric serves as a strong indicator of verification robustness and supports the delivery of reliable hardware implementation. Third-party IP integration and signoff were conducted at the subsystem level, with both coverage closure and functional correctness serving as key signoff criteria. The subsystem was verified using code coverage, functional coverage, and functional runs, along with validation of functional correctness across a variety of use case scenarios. The graph below in Figure 4 highlights the difference in signoff time between the traditional and hybrid approaches. In the hybrid approach, we achieved signoff while successfully validating all corner cases. In contrast, the traditional approach left several corner cases unverified due to high-resolution scenarios, and it took significantly longer to reach signoff.

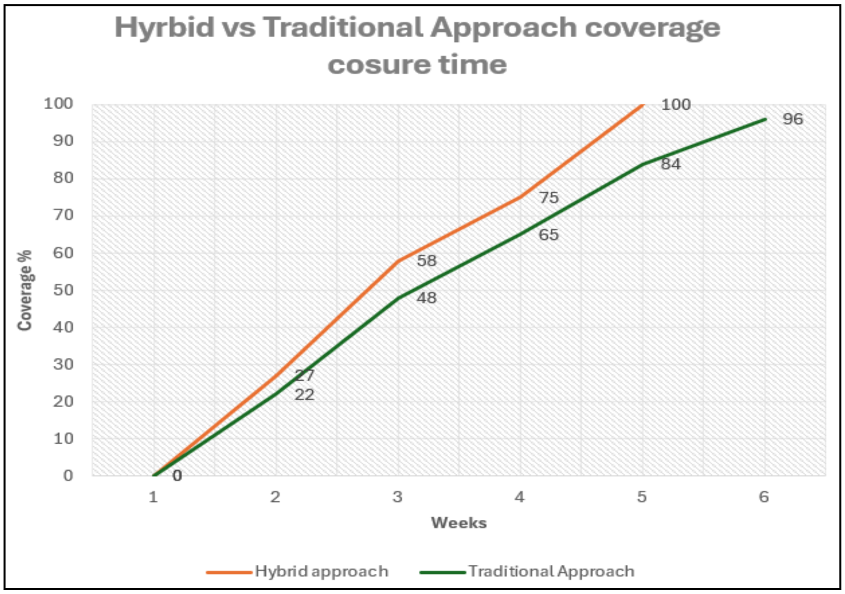

Figure 4 . Hybrid vs Traditional approach comparison

Additionally, the integration of assertion checks in emulation contributed to real-time validation of design behavior during long-running, high-resolution scenarios. These assertions continuously monitored functional conditions, enabling immediate detection of violations as they occurred. This eliminated the need to recreate issues in simulation, significantly reducing rerun cycles and saving debug effort. Assertion failures were captured in context, allowing faster root cause analysis and more efficient resolution. Overall, this enhancement improved verification robustness and accelerated the path to signoff.

VIII. CONCLUSION

A comprehensive coverage flow has been implemented using emulation, incorporating both code and functional coverage to enhance verification completeness. By leveraging the high execution speed and scalability of emulation platforms, the methodology enables thorough verification of Register-Transfer Level (RTL) designs through the execution of high-resolution and long-running tests that are impractical in simulation.

The key strength of this approach lies in its ability to verify complex RTL functionality using FW Scenarios, real-world scenarios in an emulation environment, ensuring that critical design behaviors are exercised and validated. This approach also helps acceleration in signing off verification of cores which are needed to be verified only at TOP environment rather than unit level with mix of simulation and emulation planned tests. While coverage data from both simulation and emulation can be merged to provide a unified view, this serves as a supporting tool rather than the primary objective. Looking ahead, the flow will be extended to additional projects, with ongoing efforts focused on enhancing scalability, automation, and adaptability across diverse verification platforms.

Emulation assertion points have been integrated to monitor critical design properties during runtime. These assertions help detect protocol violations and unexpected behaviors early in the emulation cycle. They enhance debug efficiency by providing immediate feedback during long-running tests. This addition strengthens overall verification quality and accelerates convergence toward sign-off.

IX. REFERENCES

[1] Siemens EDA Veloce coverage and Assertion application userguide.
[2] Unified Coverage Interoperability Standard (UCIS), Version 1.0, June 2, 2012.
[3] Verification Academy – UVM Cookbook: https://verificationacademy.com/cookbook/registers/integrating
[4] Universal Verification Methodology (UVM) 1.1 Users Guide – Accellera, May 18, 2011
[5] Universal Verification Methodology (UVM) 1.2 Users Guide – Accellera, October 8, 2015
[6] Verification Academy Coverage Cookbook: https://verificationacademy.com/cookbook/coverage
[7] Ramirez, Wilmer, Hector Gomez, and Elkim Roa. "On UVM Reliability in Mixed-Signal Verification." latin american symposium on circuits and systems, p. 233–236 (2019).
[8] Salah, Khaled. A UVM-based smart functional verification platform: Concepts, pros, cons, and opportunities. Intelligent Decision Technologies, p. 94–99 (2014).
[9] Melo R A, Valinoti B, Amador M B, Study of the Data Exchange Between Programmable Logic and Processor System of Zynq-7000 Devices, Southern Conference on Programmable Logic, p. 3–8 (2019).
[10] Chris Spear, SystemVerilog for Verification, Second Edition.