Instituto Tecnológico
y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática
MAESTRÍA EN DISEÑO ELECTRÓNICO

Comparative Analysis Between SystemVerilog and Python for Digital System Verification

---

Trabajo recepcional que para obtener el **grado** de
Maestro en Diseño Electrónico

Presenta: **MARCO ANTONIO RAMÍREZ ZEPEDA**

Director **DR. EDUARDO GARCÍA ESPINOSA**

Tlaquepaque, Jalisco. Mayo de 2026.

Dedication

To my family – Martha, Toño†, Diego, Israel, Darwin, and Rocco. You are my strength, my inspiration, and my reason for never giving up. Thank you for believing in me and the countless sacrifices you made along this journey. This work is a testament to your love and support.

Acknowledgements

I would like to express my deepest gratitude to my thesis advisor, Dr. Eduardo Garcia, and Alejandro Larios, who have been exceptional mentors throughout this research and in my professional journey. Your expertise, invaluable experience, and patience in guiding me through this work have shaped not only this research but also my growth as an engineer. I am truly honored to have worked under your guidance and sincerely hope this represents the beginning of many collaborative publications together.

I am profoundly grateful to the many coworkers, teachers, and friends who have supported me throughout this journey: Paola Rangel, Francisco Plascencia, Jorge Quiñones, Jose Preciado, Luis Nuñez, Paul Perez, Amanda Lozano, Maximiliano Ávila, Paulina Serrano, Paulina Veloso, and Valeria Fuentes, among countless others who have enriched this experience. Your thoughtful advice, unwavering support, intellectual challenges, and continuous encouragement have pushed me to grow both professionally and personally. I have deep admiration for each of you, not only as professionals but as extraordinary people who have made this journey meaningful.

My sincere appreciation extends to the faculty of the Electronics Department at ITESO, particularly Dr. Omar Longoria and Dr. José Luis Pizano, whose dedication to their students and willingness to address my questions and concerns have been instrumental in my academic and professional development. Finally, I am deeply grateful to Intel for their generous scholarship that made my master's studies possible, and for providing access to company tools and resources that enabled this research. This opportunity has been transformative, and I am honored to be part of this company.

Abstract

Digital systems have become increasingly complex over the years. It is essential to test these designs in a pre-silicon verification environment before sending them to fabrication. Skipping this stage risks costly respins and critical errors that can delay production and inflate budgets. The most widely used methodology for this purpose is the Universal Verification Methodology (UVM). This SystemVerilog library set provides a reusable, scalable framework enabling silicon design and verification teams to test their designs quickly and effectively. Despite its widespread use, this library can be hard to understand and master. Recently, a Python-based alternative has emerged. Cocotb and PyUVM are library sets that promise to boost verification engineer productivity by leveraging the full power of Python and its vast ecosystem of tools and packages. These frameworks offer a familiar syntax and a gentler learning curve than SystemVerilog for those already comfortable with software development. This document aims to conduct a comparative analysis between SystemVerilog and Python's verification libraries, evaluating the strengths and weaknesses of both frameworks across key dimensions such as learning curve, simulation capabilities, simulation time, and ease of implementation. To accomplish this, a set of verification environments will be developed for an industrial application in both languages.

Keywords

UVM, SystemVerilog, Cocotb, PyUVM, LTPI, Pre-Si, FPGA, Simulation

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Background

Digital design testing at the Register Transfer Level “RTL” encompasses multiple approaches; the two main categories are pre-silicon verification and post-silicon validation. Pre-silicon verification involves the simulation-based testing phase that validates digital designs before they reach SoC tape-out for fabrication. One of its main focuses is testing functional correctness, whereas post-silicon validation takes place after fabrication, usually in the laboratory, and it involves testing the electrical characteristics, thermal performance, manufacturing testing like JTAG boundary scan and protocol traffic injection, and the final implementation of the design.

Pre-silicon verification employs various testing methodologies across different abstraction levels. Unit-level testing targets individual design modules, providing focused verification of specific functional blocks. In contrast, system-level testing validates complete design implementations, subsystems, or entire topologies, ensuring comprehensive verification coverage before fabrication [1].

While unit-level testing offers valuable targeted verification, it presents inherent limitations. This level of testing is typically done without a set methodology, and it is very focused, resulting in limited test reusability and scalability. Additionally, it fails to identify integration issues that emerge when these individual modules are incorporated into a larger system. System-level validation addresses these limitations but introduces greater complexity; engineers need to develop a verification environment that can support several features, test sequences, automated checkers, and extensive coverage collection mechanisms so that they can test as much of the design as possible. This level of complexity creates the need to adopt a structured methodology to ensure thorough design validation while maintaining development efficiency, reusability of modules, and test maintainability [1].

The selection of an optimal verification methodology represents a fundamental challenge for engineers performing functional simulations of digital microelectronic systems, particularly given the critical importance of pre-silicon verification in preventing costly design errors and reducing time-to-market. The first verification methodology, called the e Reuse Methodology (eRM), was created in the early 2000s. Over the years, eRM methodology has paved the way for newer implementations, eventually leading to the creation of the Universal Verification Methodology (UVM) in 2011. Currently, this is the industry standard for verifying integrated circuits and digital systems [2], [3].

Universal Verification Methodology is a highly complex framework that can take years for verification engineers to master. This approach demands expertise in object-oriented programming, comprehensive knowledge of the UVM class library with its associated methods, proficiency in Verilog and SystemVerilog languages, and a deep understanding of RTL design principles, among other specialized skills [4].

1. Introduction

Regardless of its complexity, UVM has established itself as the industry standard for good reason; UVM provides a robust, standardized framework that enables systematic and scalable verification across complex digital systems. Its architecture promotes code reusability, maintainability, and modularity, allowing verification components to be easily shared across different projects and teams. The methodology's comprehensive feature set includes advanced capabilities such as constrained random stimulus generation, functional coverage collection, and sophisticated testbench automation. Furthermore, UVM's widespread industry adoption ensures extensive tool support, comprehensive training resources, and a large community of experienced engineers, making it the preferred choice for verification projects [5].

To address the complexity challenges that UVM presents and enhance the verification engineer productivity, the Co-Routine Co-Simulation (Cocotb) Python library was developed as an alternative verification framework, offering several distinct advantages over traditional SystemVerilog frameworks. Python's widespread adoption by the engineering community, combined with its extensive documentation and rich ecosystem of libraries, provides verification engineers with familiar tools that can be readily adapted to hardware verification tasks. The benefits of Python-based verification supposedly extend beyond familiarity. As an interpreted language, Python eliminates lengthy compilation cycles, reducing development time, and its intuitive syntax makes for an easier ramp-up process, which allows engineers from other areas to enter into this specialized field by lowering the complexity barrier [6], [7].

Cocotb was developed as an open-source solution under the BSD License, by a team of verification engineers from Potential Ventures with the support of Solarflare Communications Ltd, and contributions from Gordon McGregor and Finn Grimwood, who recognized the need for a more accessible yet equally powerful verification framework. The library includes design reusability and constrained random testing methodologies, preserving the proven benefits of established verification practices. Cocotb's Python foundation provides access to an extensive ecosystem of over half a million packages, offering ready-to-use solutions for common verification challenges such as file format parsing, tool integration, and results analysis and visualization that could otherwise result in a custom implementation [7].

Recognizing that system-level verification requires structured methodological frameworks regardless of the underlying programming language, and acknowledging UVM's established position as the industry's de facto verification standard, Ray Salemi, working at Siemens, developed PyUVM as a strategic bridge between Python's accessibility and UVM's proven verification principles. PyUVM represents a comprehensive translation of the UVM class library into Python, leveraging Cocotb as its simulation interface to manage event scheduling and simulator interactions [8].

PyUVM's implementation preserves the fundamental strengths that have made UVM the industry standard, including testbench component reusability across verification environments and standardized architectural components, which enable verification engineers to transition seamlessly between projects, organizations, and industry sectors.

By maintaining UVM's established verification patterns while operating within Python's more accessible programming environment, PyUVM aims to deliver the methodological rigor of traditional UVM while potentially reducing the time it takes to develop these verification environments [8].

This research aims to evaluate whether Python's Cocotb and PyUVM libraries can deliver equivalent verification capabilities to those provided by SystemVerilog's UVM class library. The hypothesis is that Python's inherently intuitive syntax and development paradigm should reduce verification development time while maintaining the same level of verification quality and coverage achieved with SystemVerilog and UVM. Furthermore, the availability of mature Python libraries should accelerate development cycles by eliminating the need to implement common functionality from scratch.

1.2 Problem Statement

While UVM has established itself as the industry standard for digital system verification, its adoption presents significant barriers that limit verification engineer productivity and accessibility. The complexity of mastering UVM's extensive capabilities, combined with SystemVerilog's specialized object-oriented programming requirements, creates a steep learning curve that can require months to years of dedicated experience. This complexity restricts the pool of qualified verification engineers to a specialized subset of the electronics engineering community.

Although Cocotb, alongside PyUVM, has emerged as a Python-based alternative claiming to offer equivalent verification capabilities with improved accessibility and reduced learning time, there is insufficient empirical evidence to validate these claims. The lack of a comprehensive comparative analysis between SystemVerilog and Python as the backbone for a verification environment creates uncertainty for organizations considering the latter for verification framework adoption, particularly in terms of verification quality, development efficiency, and long-term maintainability.

This research addresses the critical need to quantitatively evaluate whether Python's Cocotb framework, alongside PyUVM as a methodology, can deliver verification results equivalent to SystemVerilog's UVM class library while providing the promised benefits of reduced complexity and accelerated engineer proficiency, thereby informing evidence-based decisions in verification methodology selection.

1.3 General Objectives

- Perform a comparative analysis between the development of a digital verification environment using SystemVerilog's UVM and Python's Cocotb and PyUVM for an industrial design.

1.4 Specific Objectives

- Analyze the learning curve and ease of development that each framework has.
- Develop a verification environment in both frameworks for an industrial application.
- Compare the coverage collection for both frameworks after performing a regression.
- Compare the time it takes for each regression to finish.
- Compare performance metrics in terms of memory and CPU usage in both frameworks.

1.5 Justification

Verification environment development has become essential for validating digital systems, integrated circuits, and Field Programmable Gate Arrays (FPGAs) applications, serving as a critical safeguard against costly manufacturing errors. By identifying functional defects during pre-silicon verification, companies can significantly reduce fabrication iterations, saving substantial time and financial resources that would otherwise be lost due to silicon respins. Despite the critical importance of this field, the verification engineering effort remains limited due to the complexity of the process, while the life cycle of digital designs becomes increasingly smaller, challenging engineers to optimize their processes and execute ever more rapidly while ensuring that designs reaching the market are functionally correct and meet all specified requirements.

This research addresses this industry challenge by providing a comprehensive analysis of UVM and Cocotb/PyUVM verification methodologies. This study will deliver quantitative performance comparisons using identical verification environments on a common industrial design implementation, providing concrete evidence to support methodology selection decisions.

Currently, academic literature lacks rigorous empirical validation of Cocotb and PyUVM's claimed advantages over traditional SystemVerilog's UVM approaches, other than a couple of small-scale examples that may not translate properly into an industrial design and verification workflow. While their proponents assert benefits such as reduced learning curves, faster development cycles, and equivalent or improved verification coverage, these claims remain unsupported by systematic comparative studies. This research gap leaves verification teams and organizations without the evidence-based data necessary to make informed decisions about verification framework adoption, potentially hindering the broader adoption of a more accessible verification library and language that could address the industry's challenges.

2. State of the Art

This chapter provides an examination of the theoretical foundations and technological evolution that underpin modern digital design verification methodologies. The chapter begins with section 2.1, which traces the historical progression of digital design validation from its origins to the emergence of hardware description languages such as Verilog and SystemVerilog. This historical context establishes the technological trajectory that led to contemporary validation challenges and the subsequent development of standardized methodologies. The section concludes with an overview of the integrated circuit design and verification flow, illustrating how validation and verification have become an integral component of the modern semiconductor development process.

Section 2.2 focuses on functional verification using simulation tools, examining the productivity challenges facing the industry. The chapter then examines verification environments using SystemVerilog's UVM, presenting quantitative data on its industry adoption, performance metrics, and established benefits in large-scale semiconductor projects. Finally, the chapter explores verification environments using Cocotb and PyUVM, reviewing the limited but growing academic literature that compares these Python-based approaches with traditional SystemVerilog methodologies.

2.1 Digital Design and Verification Evolution

2.1.1 Historical Progression

The evolution of digital design verification began with manual circuit testing and progressed through transistor-level simulation tools like SPICE in the 1970s, but the increasing complexity of integrated circuits demanded higher-level abstraction methods [9], [10]. Verilog emerged in 1985 as Gateway Design Automation's response to the growing need for a unified Hardware Description Language (HDL) that could handle both design and verification complexity. Drawing from earlier languages like HILO-2 and Occam, Verilog provided syntactical structure, RTL abstraction capabilities, and concurrent procedural aspects that enabled engineers to describe and simulate digital systems more effectively. After Cadence placed Verilog in the public domain in 1990 and it became IEEE Standard 1364 in 1995, it rapidly gained widespread adoption across academic institutions and semiconductor companies worldwide, establishing itself as the primary language for hardware design engineers [3], [10], [11].

However, design complexity increased by a factor of 20x while Verilog remained relatively unchanged. The language became insufficient for large verification environments, prompting engineers to create custom C and C++ testbench environments. This fragmentation led to the development of SystemVerilog in the late 1990s and early 2000s, which aimed to reunify hardware design and verification by integrating advanced verification features back into the Verilog language.

2. State of the Art

SystemVerilog 3.0 eventually was approved by Accellera in 2002, incorporating contributions from industry leaders, ultimately merging with Verilog to become IEEE 1800-2009. The subsequent development of the Universal Verification Methodology (UVM) by industry collaboration between Synopsys, Cadence, and Mentor Graphics established SystemVerilog as the foundation for modern verification practices [3], [10], [11].

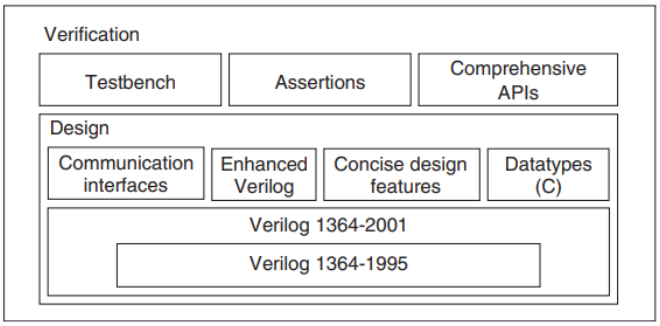


Figure 1: Key Components of SystemVerilog [11].

2.1.2 Integrated Circuit Design and Verification Workflow

The modern integrated circuit design flow represents a systematic approach to transforming conceptual ideas into fabricated silicon, progressing through distinct phases that manage increasing complexity and cost implications. The process described below can be visualized in Figure 2. The process begins with market-driven requirements that are refined into architectural specifications defining the Integrated Circuits (IC) operational parameters, data flows, clock frequencies, and power consumption estimates. The high-level architecture is then translated into a micro-architecture specification (MAS), which establishes major components and interfaces, leading to parallel RTL design and verification phases where HDLs like Verilog and SystemVerilog are used to implement and validate the design's functionality. The flow culminates in physical design tape-out, followed by post-silicon validation, where manufactured chips undergo testing. The critical importance of early-stage verification becomes evident when considering that any issues discovered post-fabrication require returning to the RTL phase and repeating the entire manufacturing process, making pre-silicon verification essential for both economic and schedule considerations [12].

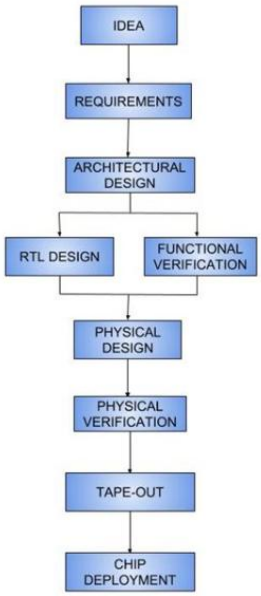


Figure 2: IC Design Flow [12].

Functional verification serves then as the cornerstone of modern electronic design automation, systematically validating that digital designs perform their intended functions before costly fabrication. This process encompasses specification analysis, testbench creation using verification languages and methodologies like UVM, comprehensive simulation under various conditions, coverage analysis to ensure thorough testing, debugging to resolve discrepancies, and formal verification for mathematical proof of correctness. The verification workflow employs multiple techniques, including static simulation of RTL, dynamic simulation for functional correctness under diverse stimuli, emulation for large design acceleration, and coverage-driven verification that has become the industrial standard for measuring verification completeness. Modern simulation-based verification relies heavily on automated test generation, checking, and coverage collection, with simulators serving as the backbone of the verification environment, though the event-driven nature and complex dependencies of RTL designs present ongoing challenges for parallelization and performance optimization. [1], [13]

2.2 Functional Verification Using Simulation Tools

2.2.1 Verification Productivity Challenges

Hardware verification is a mature topic, both in research and in industrial practice. Verification research dates back at least three decades, with a rich body of literature, and in industrial practice, verification is now firmly established as an integral component of the system development flow. However, despite these advancements, there remains a significant gap between the state of the art in technology today and the validation needs for modern industrial designs. The situation is exacerbated as we move rapidly to the era of automated vehicles, Internet of Things (IoT), and Artificial Intelligence (AI) [1].

2. State of the Art

One of the critical impacts on verification in our current day is the resources available. With the demand to produce billions of diverse computing systems, time-to-market requirements for design and system development have become more aggressive. A typical microprocessor life cycle from exploration to start of production used to range between three to four years; for some devices, this has shrunk to a year or less. Such aggressive shrinkage obviously implies inadequate time for thorough design review, potential misunderstanding of specifications and requirements from developers and stakeholders, and a consequent increase in errors. The shrinking life cycle also means less time for validation. The demand for verification has been to handle potentially more error-prone designs than ever before, with even less time and fewer resources [1].

Most electronic devices today are architected through a SoC paradigm: the idea is to develop a system through integration of predesigned hardware and software blocks, often collectively referred to as design intellectual properties (IPs). IPs are developed independently, either in-house or by third-party vendors, and an integration team collects these IPs based on the system requirements for the target device. The verification and validation of these devices involve two separate flows, one for ensuring the correct operation of the IPs and another for the assembled system. Verification and validation flows today are significantly complex, requiring careful upfront planning, and span almost the entirety of the design life cycle [1].

The pre-silicon verification is the major resource-intensive verification activity that takes place during hardware development and implementation. This is a continuous process, with an increasing level of maturity and complexity as the design matures. Most industrial SoC designs include a combination of legacy and new IPs, some created in-house and some collected from third-party IP providers. A verification team performs the verification of the IP being delivered, and the objective is to ensure that it functions on its own as expected. Subsequently, the SoC team would integrate the IPs into a SoC model and perform system-level validation, with the target being to ensure that the IPs function correctly together as an integrated system. An IP is delivered to the integration team as a hard IP or a soft IP, and the verification performed by the IP team depends on the form in which it is delivered. More recently, there has been a strong push to avoid "over-validation," i.e., to validate an SoC design only for its target use cases [1].

The exponential growth in devices has resulted in a shrinkage in the development life cycle, leaving little time for customized verification efforts. However, each device has different use-case requirements, functionality, performance, and security constraints. Verification engineers are required to create standardized, reusable verification flows and methodologies that can be easily adapted to a diversity of electronic devices. Two approaches have so far been taken to address this problem. The first is to improve tool scalability to eventually turn verification into a turn-key solution; however, achieving this goal remains elusive. The other approach entails making the systems themselves highly configurable, so that the same design may be patched to perform various use cases either through software or firmware update or through hardware reconfiguration. Developing such a configurable design also has a downside. Aside from the fact that it is impossible to determine all the different use cases of a hardware system in advance, this approach also significantly blows up the number of states of the system and consequently makes their verification even more challenging [1].

Given the aggressive time-to-market requirements, there has been a general move in verification today away from comprehensive coverage of the whole system and toward more narrowly defined coverage of intended usage scenarios. For example, for a device intended for low-power and low-performance applications, the intended usage would include scenarios where different components transition into various sleep modes but would not include sustained execution at high-clock speeds; conversely, a high-performance device would prioritize execution at high-clock speeds. This approach, while attempting to reduce over-validation, might induce significant complexity in the process. The usage scenarios are typically defined at a level of the device and involve complex interactions of hardware, firmware, and software; it is not trivial to determine from such high-level verification targets how to define verification goals for individual IPs, or even hardware blocks for the entire SoC. Finally, exercising the device-level use case requires hardware, firmware, and software at a reasonable maturity, which is available only late in the system life cycle. Bugs found this late may be expensive and may involve considerable design churn [1].

2.2.2 Verification Environment Using UVM

Universal Verification Methodology is a set of standards, tools, and APIs for creating a universal way for verifying designs. This methodology is meant for building functional test benches for SoCs. The main benefit of UVM verification has to do with standardization across the industry. The major EDA vendors and IP providers support UVM, resulting in broad support across different tools and verification IPs (VIPs). Interoperability and reuse are their major strengths; it allows engineers to easily integrate a VIP into a verification system due to the standardized interface. [14]

UVM has become ubiquitous in chip verification. Companies like Intel, NVIDIA, Qualcomm, ARM, and AMD use UVM to verify SoCs. A modern high-end verification environment might contain 200,000+ lines of UVM/SV code with several reusable agents for protocols like PCIe, AXI, DDR, I2C, and proprietary interfaces. Without UVM's standardization, each company would build custom frameworks, wasting engineering effort and preventing IP reuse across the industry [15].

It has established itself as the fundamental methodology for functional verification across multiple industries, transforming the landscape of complex hardware design validation. Industry surveys indicate substantial growth in UVM adoption, with implementation rates increasing by 76% between 2018 and 2023, resulting in over 82% of semiconductor companies now utilizing UVM as their primary verification methodology. This widespread implementation has yielded quantifiable improvements in verification performance, including a 42% reduction in verification cycles and a 58% enhancement in bug detection rates during early development phases [5].

This methodology's effectiveness is particularly evident in SoC verification applications. Empirical studies demonstrate that UVM-based verification environments consistently achieve up to 94% functional coverage within the initial verification cycle, representing a substantial improvement over the 69% coverage typically achieved with traditional verification methodologies. This enhanced verification performance translates directly into significant economic benefits, with organizations reporting estimated cost savings of $2.4 million per project for large-scale semiconductor developments. These quantifiable performance metrics have

solidified UVM's position as the industry standard, creating compelling evidence for its continued adoption and reinforcing its role as the benchmark against which alternative verification methodologies must be evaluated [5].

The quantitative benefits of UVM implementation demonstrate improvements across multiple verification metrics. Time-to-market optimization represents one of the most significant advantages, with organizations reporting a 44% decrease in overall verification time, 3.4x acceleration in testbench development cycles, and a 61% reduction in regression testing duration. Resource optimization benefits are equally compelling, including a 39% reduction in verification engineer workload, 75% code reusability across projects, and a 54% decrease in verification infrastructure costs [5].

Industry impact analysis from leading semiconductor companies reveals the broader implications of UVM adoption. Implementation data indicates an 84% reduction in post-silicon bugs, demonstrating the methodology's effectiveness in early detection. Test scenario coverage has improved by 91% while time-to-market for new chip designs has accelerated by 72%. Additionally, verification-related project delays have decreased by 67%, significantly improving project predictability and resource allocation [5].

These performance metrics become increasingly critical as design complexity continues to escalate. UVM's structured verification approach has proven particularly essential for validating designs exceeding 95 million gates, which now represent more than 58% of new chip projects. The methodology's ability to manage this complexity while maintaining verification quality and efficiency underscores its fundamental importance in contemporary hardware development, establishing the performance benchmarks against which emerging frameworks must be evaluated.

2.2.3 Verification Environment Using Cocotb and PyUVM

Industry adoption of Cocotb and PyUVM remains in its early stages without probable documentation on the matter; with standardized methodologies and strong industry support, UVM remains a preferred choice for large-scale verification projects. Despite this, there is some academic literature that is described in the following section. In paper [2] N. Suthira and his team performed a comparative analysis between UVM in SystemVerilog and Cocotb in Python. The case study for this analysis was the Advanced Encryption Standard (AES) algorithm. This algorithm was simulated with a sample of three random inputs of size 128 bits, three different keys of sizes 128 bits, 192 bits, and 256 bits. Three outputs were monitored during this analysis. Using UVM, two monitors were utilized, one to transmit the output of the design under test (DUT) to the scoreboard and another serving as a reference model. Using Cocotb, a testing process involving the generation of 1000 test cases was used to randomize 2^{16} Input values and a flag were used to confirm that the test cases were passing. The results presented showed that Cocotb presented superior overall coverage compared to UVM.

Code coverage	Overall	Block	Expression
UVM	87.49%	89.19%	100%
Cocotb	89.55%	85.81%	100%

Table 1: Comparative results between Cocotb and UVM in terms of code coverage.[2]

Functional Coverage	Expected	Unexpected	Covered
UVM	64	17	47
Cocotb	65536	0	65536

Table 2: Comparative results between Cocotb and UVM in terms of functional coverage. [2]

Simulation Time	Time in nanoseconds
UVM	1000 ns
Cocotb	10000.5ns

Table 3: Comparative results between Cocotb and UVM in terms of simulation time [2].

In paper [6] Hang Yang and his team presented a case study demonstrating Cocotb's application in verifying the design of a multi-master interconnection network based on the AMBA protocol, and an analysis of the Clock Domain Crossing (CDC) transmission and dynamic arbitration problems encountered in SoC design. The Cocotb testbench is connected to the simulation circuit via Cocotb's VPI interface. The testbench comprises the following components: Bus Functional Model (BFM), monitor, reference model, and scoreboard.

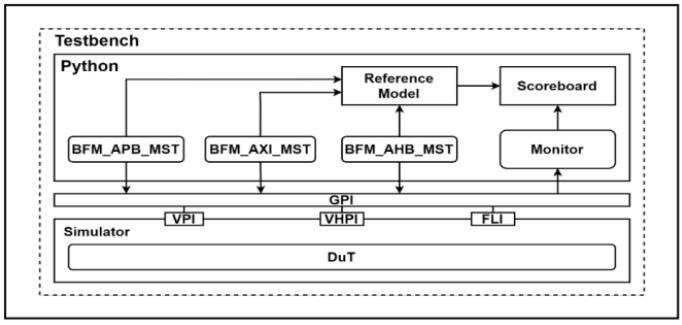


Figure 3: Cocotb's testbench architecture to test AMBA-based protocol [6].

The BFM is employed for each bus protocol, simulating its behavior as the host interface. The monitor was tasked with capturing the signal output from the DUT in real time and relaying it to the scoreboard. The reference model generated a desired output based on the input stimuli and serves as a benchmark for comparison with the actual output of the DUT. The scoreboard receives the output signals from both the reference model and the monitor and performs a comparison between them. There were five test cases analyzed in this study: QoS configuration, AHB Master access, AXI Master access, CDC, and dynamic arbitration. They demonstrated that Cocotb's concurrent mechanism and asynchronous operation are well-suited to address intricate scenarios, such as multi-protocol communication and arbitration. They concluded that Cocotb enhances the efficiency of verification procedures in comparison to conventional verification techniques [6].

2. State of the Art

In [16] Ş. Güney and I. Cicek presented a Cocotb verification approach for commonly used RISC-V R-type instructions, including ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLR, and SLT. They used PicoRV32, an open-source, compact, and efficient RISC-V processor core that supports the RV32I base instruction set and other optional extensions. The Cocotb environment consisted of two main components: PicoRV32Driver and PicoRV32Monitor. The driver manages reset signals, memory operations, and instruction/data transfers, simulating basic processor controls. The monitor tracks memory activity per clock cycle, logging timing, addresses, and data, while detecting errors and warnings. The test vectors are systematically generated using cocotb.regression instruction and have a testcase per R-type instruction of the RISC-V, including boundary values, bit patterns, and critical cases. TestFactory automatically creates different combinations of these values to verify that the processor operates correctly under all possible conditions [16]. The coverage analysis in this study was performed using RISCV Coverage classes. The DUT is monitored in terms of functional coverage.

The test vector generation involved randomizing input values targeting specific corner cases for each R-type instruction, managed by a dedicated test vector. Next, the environment was set up by initializing the instruction memory with the generated test instruction and preparing the memory model along with the coverage collector. Finally, the test execution and verification took place, coverage was collected for each execution, and the results were compared with the expected output. The team reported a coverage of 89% across all tested instructions, operating with precise timing parameters including a 10ns clock period, 20ns reset duration, single-cycle memory access, and instruction times ranging from 1 to 4 cycles [16].

3. Literature and Review of Key Concepts

This chapter establishes the theoretical foundation and technical context necessary for understanding the comparative analysis work that will be explained in the following chapters. Chapter 3 begins with section 3.1, which examines Object-Oriented Programming principles, providing essential background on the programming paradigm and underlining UVM's infrastructure and influences both Python and SystemVerilog environment implementations. Sections 3.2 to 3.4 explore fundamental verification concepts, including functional verification principles, coverage-driven verification methodologies, and constrained-random testing simulation techniques that form the backbone of modern verification practices. These sections establish the methodological background within which both SystemVerilog and Python-based alternatives operate.

The chapter then goes on to provide context for the specific verification frameworks under investigation. Section 3.5 presents a detailed examination of Universal Verification Methodology (UVM), including its architectural components, phases, and implementation strategies that have made it the industry standard. Sections 3.6 and 3.7 introduce Cocotb and PyUVM libraries, respectively, explaining their Python-based approaches to hardware verification and their relationship to traditional UVM concepts. Finally, section 3.8 provides a technical overview of the LVDS Tunneling Protocol & Interface, which serves as the device under test (DUT) for this comparative study. A DUT is an electronic product or system that is being evaluated to verify its performance, functionality, or regulatory compliance [17].This section covers the Open Compute Project context, DC-SCM architecture, LTPI protocol specifications, channel definitions, frame structures, and link training procedures, establishing the technical complexity and verification challenges that will be addressed using both verification languages in the subsequent experimental analysis.

3.1 Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around "objects" rather than functions and logic. The key strength of OOP is abstraction. Programs written with this paradigm contain objects and a set of relational classes that facilitate the reusability and flexibility of code. If a complex project is divided into a set of relational classes, then it is possible to achieve low production time and cost, and the complexity is reduced, which allows these projects to be better managed by a team of developers [18].

Object-Oriented Programming defines the following features. All the modern and old OOP languages might implement some or all these features:

1. Classes: A class is the basis of all OOP programs. It contains functions, called methods, and variables, called attributes. Classes abstract the code and data from outside interference and misuse. A class is a logical programming construct, meaning it does not exist in code, but rather it serves as a blueprint for creating objects. It allows other classes to derive from it. It is a blueprint that contains generalized methods that need to be reused or declared for specific implementation [18].

2. Objects: An object is an instance of a class. It contains attributes and behavior or data that describes specific functionality. Unlike classes, objects have an existence in the code. An object can contain private data that may only be accessed by specific classes and can have a specific implementation for accessing this data. It also allows for the passing of information through methods, which can send and receive data [18].
3. Encapsulation: Encapsulation is a mechanism of binding data and code together. In OOP, a class is the basic unit of a program, thereby making it the basic unit for encapsulation. Encapsulation is often used to hide the internal states of a class from the outside of the code [18].
4. Inheritance: It is a concept in OOP in which a class can acquire the properties of another class to expand on it. The main use of inheritance is the reusability of code. Reusing the attributes and methods lowers the risk of errors, enhances maintainability, and makes the task of programming easier. Inheritance has an "is-a" relationship with derived classes. Inheritance also makes it possible for a class to derive from multiple classes [18].

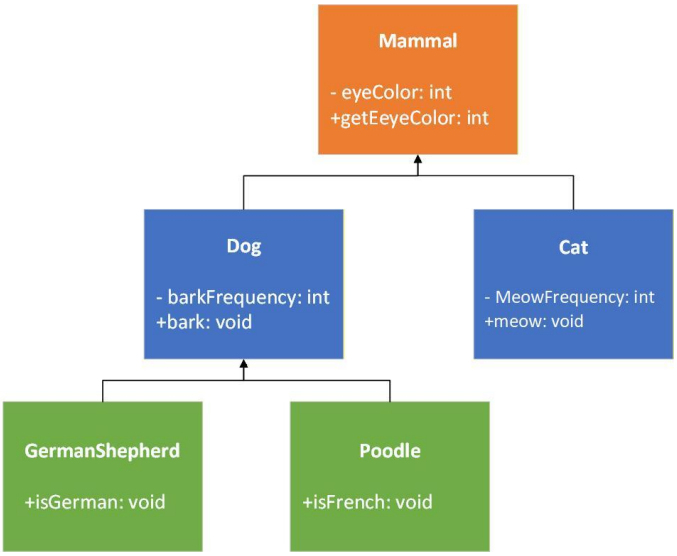


Figure 4: Example of inheritance in OOP.

5. Polymorphism: Polymorphism facilitates the user designing a common interface for a group of related actions. In other words, it allows for a function to get implemented in different ways in base classes, and derived classes can modify this implementation to meet their specific needs [18].
6. Abstraction: Abstraction is on top of all other OOP concepts. It allows users to manage the complexity of objects. Abstraction focuses on the essential objects and hides the implementation details from the outside world, giving the user interfaces to help manage an object without the need to review its inner design. The difference between abstraction and encapsulation is that abstraction only focuses on the relevant details of the object,

whereas encapsulation binds and hides the data and functions into a single unit and lays a protective wrapper around it from unauthorized access [18].

3.2 Functional Verification

In a typical IC design flow, functional verification ensures that the implementation of the design conforms to the specification. It has been reported that the functional verification process consumes more than 70% of the design effort. It is a critical step in the digital design workflow because an undetected design bug may result in significant financial loss for a company. Therefore, effective verification strategies and techniques have become ever more indispensable to the design flow to ensure high verification quality [19].

Simulation-based verification is the most widely used approach in functional verification. Simulation is based on testbenches and includes input stimuli and expects output responses from the design. The efficiency of the simulation determines the efficiency of the verification task, and hence, having compact and high-quality stimuli is critical to this approach. A general definition of verification is “the act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services, or documents conform to specified requirements.” Within the context of design automation of IC designs, functional verification is the step to ensure that the specifications and the implementation of the design at various abstraction levels are in accordance with the design intent [19].

3.3 Coverage-Driven Verification

Coverage-Driven Verification (CDV) is a systematic approach that promotes achieving coverage closure efficiently, generating effective tests to explore a DUT, efficient coverage data collection, and consequently efficient verification of the DUT with respect to its requirements. In CDV, a verification plan must be constructed before the testing process begins. This plan includes the aspects of the DUT that need to be verified and the coverage strategy. The coverage strategy indicates how to achieve effective coverage through the design of testbench components, especially the test generator, the checker, and the coverage collector. The coverage strategy also specifies how to measure the coverage [20].

The DUT is placed into a test environment or testbench. This environment represents a model of the universe outside the DUT. The process of testing is realized using the following four core components in a testbench, as shown in Figure 6:

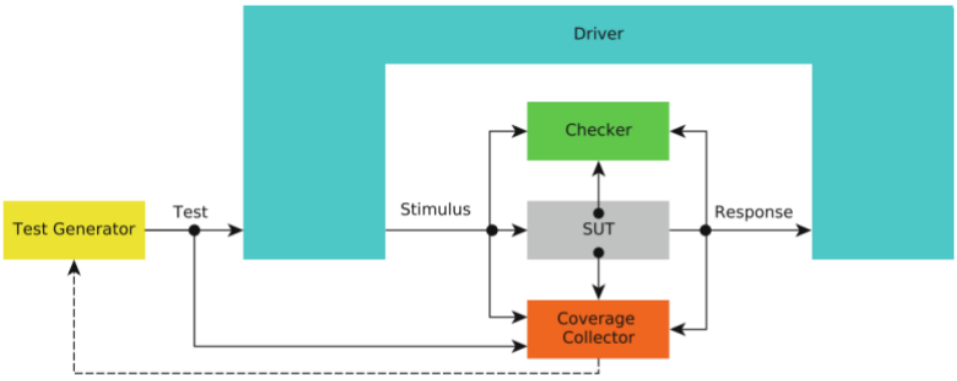


Figure 5: Structure of a basic CDV testbench [20].

The test generator is the component that generates stimulus for the DUT; the driver is the component that takes a test, potentially at a high-level of abstraction, translates it into the level of abstraction used in the simulation, and drives it to stimulate the DUT. The checker is the component that checks the response of the DUT to the stimulus and detects failures. And the coverage collector is the component that records the quality of the generated tests with respect to a set of complementing coverage models. A key objective in the design of a CDV testbench is to achieve a fully autonomous environment, so that verification engineers can concentrate on areas requiring intelligent input, namely efficient and effective test generation, bug detection, reliable tracking of progress, and timely completion [20].

3.4 Constrained-Random Testing Simulation

Constrained-Random Verification (CRV) is a methodology supported by SystemVerilog, which has its built-in constraint solver. This methodology allows the user to constrain the stimulus injected into a DUT to better target a design function, thereby allowing the user to reach the coverage goals faster with accuracy. The verification engineer checks the coverage of a test, reviews where the coverage holes are, and then constrains the stimulus to target those holes and further improve coverage. Fully random testing can end up wasting a lot of simulation cycles without improving coverage, by potentially missing corner cases of importance [21].

The design verification team starts putting together a verification environment and a test plan. The tests are written according to the test plan, and the verification cycle begins. The key component missing in this traditional methodology is planning for coverage. Without a comprehensive coverage plan, the verification team has no idea how their tests and test benches are performing. The first step in CRV is to create a functional coverage plan. Once the coverage plan is ready, the team measures the coverage after each simulation (regression) run. If the coverage is not complete, the holes can be identified. When directed testing is not enough, constrained random stimuli are generated to fill these gaps [21].

3.5 Universal Verification Methodology “UVM”

UVM is a transaction-level methodology (TLM) designed for testbench development. It is a class library that makes it easy to write configurable and reusable code. UVM designers created the class library whose components can be used to develop a testbench. Using the UVM class library, only the driver, scoreboard, basic transactions (sequence), and sequence library need to be changed. UVM is class-based and communicates between the classes via transactions. This helps keep the communication interface among class components separate from the implementation detail in the UVM agent driver [21].

UVM provides a set of transaction-level communication interfaces and channels that an engineer can use to connect components; this, in turn, isolates each component from changes in other components throughout the environment. When coupled with the phased, flexible-build infrastructure in UVM, TLM promotes reuse by allowing any component to be swapped for another, as long as they have the same interfaces [21].

3.5.1 UVM Environment

The top-level environment contains agents for specific DUT interfaces. It may contain a scoreboard as well, which compares the output with expected results [21].

3.5.2 UVM Agent

UVM agent comprises a sequencer, a driver, and/or a monitor. It serves as a container for a specific interface. An agent can be categorized as an active agent if it contains a driver, thus injecting stimuli to an interface, or as a passive agent, if it only contains a monitor, in which case its purpose is to review that transactions and logic are working in the DUT [21].

3.5.3 UVM Sequencer

The sequencer controls the flow of request and response sequence items between sequences and the driver. It is a simple component that serves as an arbiter for controlling transaction flows from multiple stimulus sequences. The sequencer and the driver use the TLM interface to communicate.

3.5.4 UVM Driver

The driver is where the TLM transaction-level world meets the DUT signals, clock, and pin-level world. A driver receives sequences from the sequencer, converts the received sequence into signal-level activities, and drives them on the DUT interface as per the interface protocol.

3.5.5 UVM Sequence

Sequences are an ordered collection of transactions; they shape transactions to a specific need and generate as many as the user defines. At this level, the transactions can be an abstraction of what the driver must implement, and the variables in the transaction are random, which can then be constrained by the sequence.

3.5.6 UVM Sequence Item

It is the fundamental lowest denominator in the UVM hierarchy. It is the definition of the basic transaction that will then be used to develop UVM sequences. It defines the basic transaction for data items. While the driver deals with signal activities at the bit level, the sequence item is an abstraction of these stimuli.

3.5.7 UVM Monitor

The role of the monitor is to take the DUT signal activity and convert it back into transactions to be sent out to the rest of the testbench for analysis.

3.5.8 UVM Scoreboard

The scoreboard checks the responses of the DUT against the expected response. The UVM scoreboard typically receives transactions from the monitor through the UVM agent's analysis port and then compares the expected output versus the received transaction from the monitor.

3.5.9 UVM Testbench

The UVM test bench typically instantiates the DUT module, and the UVM test class configures the connections between them [21].

3.5.10 UVM Test

The UVM test is the top-level UVM component in the testbench. The testbench merely instantiates the DUT and the UVM Test class and configures the connection between them. A test is a class that encapsulates test-specific instructions written by the verification team. By using classes, engineers can inherit and configure the top-level environment, and it is then extended to define scenario-specific configurations, such as which sequences to run, coverage parameters, etc. Typically, there is one base UVM Test with the UVM environment instantiation and other common items. Individual tests will extend this base and configure the environment differently, or select specific sequences to run [21].

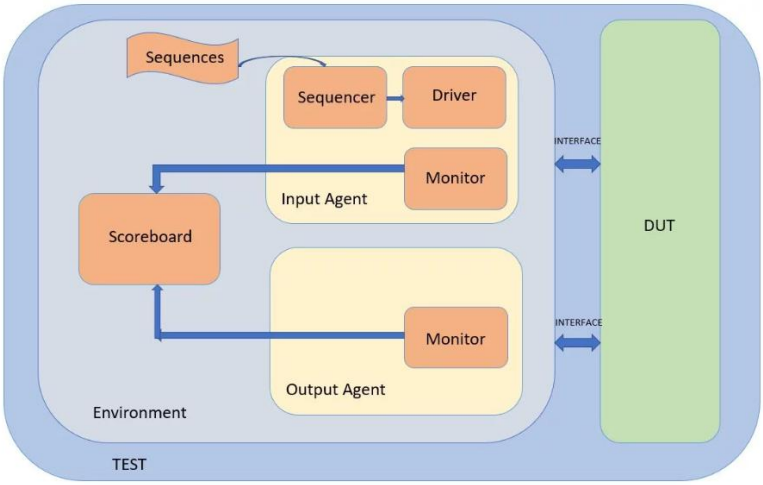


Figure 6: UVM Architecture Diagram [22].

3.5.11 UVM Phases

Another key topic in UVM is the concept of phases, since several files are working together during a simulation, phases control the order of testbench construction and execution. UVM defines standard phases that all components traverse:

1. Build phase: Construct components hierarchically (top-down).
2. Connect phase: Establish transaction-level model (TLM) connections between components.
3. End of elaboration phase: Performs final checks before simulation
4. Run Phase: Executes stimulus and monitoring (Can also be sub-categorized in reset phase, configure phase, main phase, and shutdown phase if necessary).
5. Extract phase: Collects coverage and results.
6. Check phase: Verify pass/fail criteria.
7. Report Phase: Final print summary [15].

Phases eliminate race conditions and ensure deterministic testbench behavior. All scoreboards are built before drivers start sending stimulus. All monitors complete before checking begins. This synchronization happens automatically by using the built-in phases without any manual coordination needed [15].

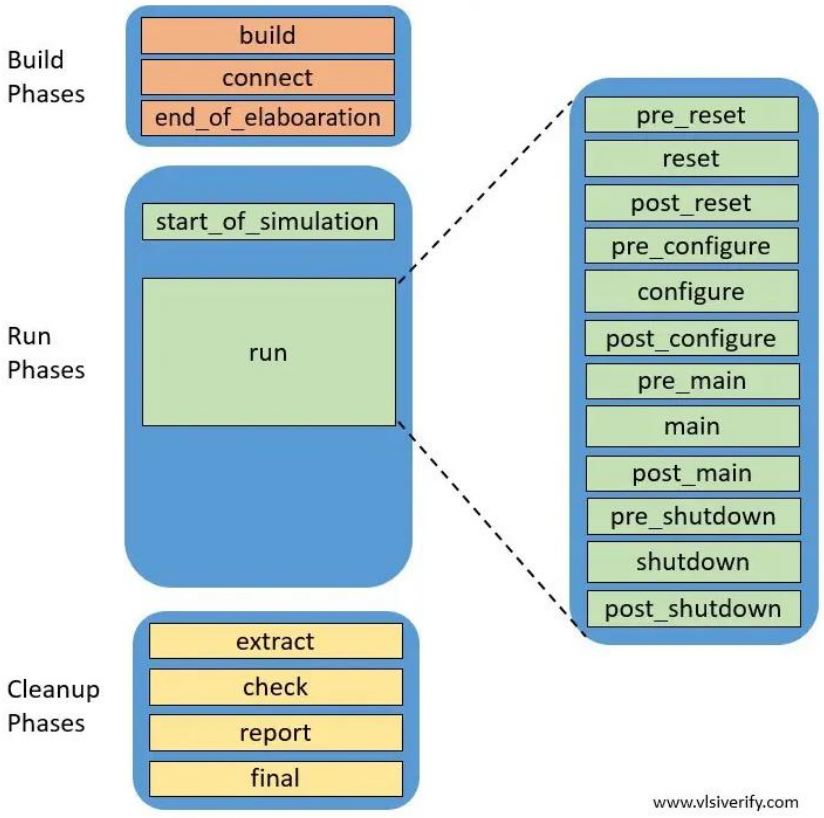


Figure 7: UVM Phases Execution Flow [23].

3.6 Cocotb Library

Cocotb is a coroutine-based Co-simulation Testbench environment for verifying VHDL and SystemVerilog RTL using Python. It is an open-source library under the BSD License and hosted on GitHub. Cocotb encourages the same philosophy of design reuse and randomized testing as UVM, the implementation is built in Python. With this library, VHDL or SystemVerilog are normally only used for the DUT itself, leaving the verification environment to be written fully in Python [7].

This verification framework unites the adaptability of the Python language with the functionalities of an HDL. It offers a concurrent testing methodology based on concurrency, which allows for the efficient construction of test scenarios and use cases. The fundamental aspect of the Cocotb platform is its external interaction mechanism with the HDL simulator using standard language interfaces, including Verilog Procedural Interface (VPI) and VHDL Procedural Interface (VHPI), and Foreign Language Interface (FLI). Cocotb can function with a multitude of commercial and open-source simulators, including VCS, ModelSim, GHDL, and Icarus Verilog. This architectural approach ensures compatibility with a diverse range of hardware description languages, including VHDL, Verilog, and SystemVerilog, while simultaneously offering substantial flexibility for hardware validation.

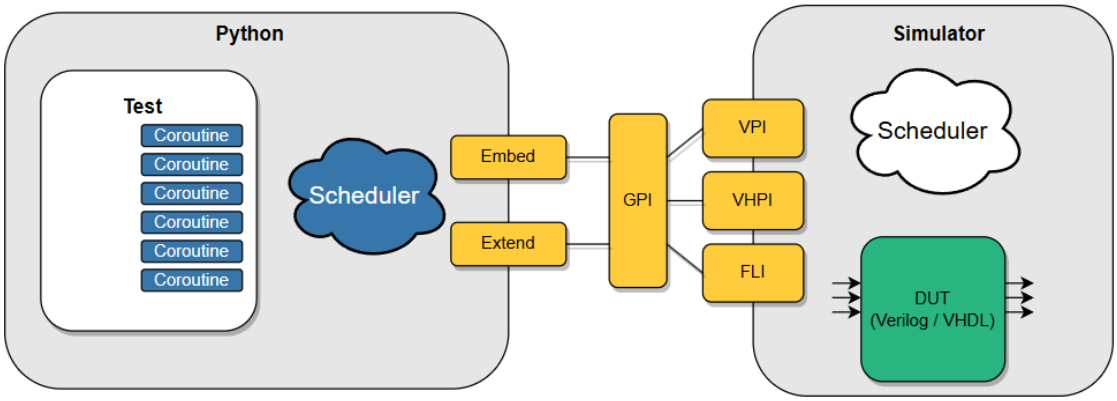


Figure 8: Cocotb connection mechanism with DUT [7].

Cocotb interacts with the simulator through a C++ library, designated as Generic Programming Interface (GPI). This abstraction layer encapsulates the VPI, VHPI, and FLI interfaces, enabling Cocotb to integrate seamlessly with simulators that support these interfaces. The Python layer interacts with the simulator through an extension module, designated as the simulator. This extension module interacts with the GPI and provides functions such as traversing the HDL hierarchy, accessing and setting signal values, and registering event callbacks. When a test requires interaction with the simulator, Cocotb initiates calls through the interfaces, enabling the testcase to control and observe the state of the DUT [6].

3.7 PyUVM Library

Universal Verification Methodology is the dominant RTL verification network across the IC industry. It allows engineers to reuse testbench components across several verification environments. IEEE defined the UVM in the IEEE Standard for Universal Verification Methodology Language Reference Manual, also known as the IEEE 1800.2 standard [24].

While the industry defines the 1800.2 standard in SystemVerilog, there is little in the standard that requires it to be written in said language. It could be implemented in other languages with sufficient object-oriented support, like Python. PyUVM is the Universal Verification Methodology implemented in Python. It uses Cocotb's constructs to interact with a simulator and schedule simulation events. PyUVM takes advantage of Python's ease of use and object-oriented power to implement the most-often used parts of the IEEE 1800.2 standard. One of the advantages that Python offers over SystemVerilog is the support of important object-oriented programming (OOP) concepts, such as multiple inheritance, that are not developed in SystemVerilog [8].

PyUVM is a clean implementation of IEEE1800.2, which has already implemented the following sections of the standard:

Section	Name	Description
5	Base Classes	Basic classes such as uvm_void and uvm_object.
6	Reporting Classes	PyUVM uses the logging package to implement reporting.
8	Factory Classes	It implements all the UVM factory functionality without using the macros needed in SystemVerilog.
9	Phasing	IEEE1800.2 describes basic phasing and complicated custom phasing. PyUVM only implements the basic phasing.
12	UVM TLM Interfaces	It fully implements the UVM TLM system.
13	Predefined Component Classes	It implements uvm_component with hierarchy, uvm_root singleton, and the run_test task(). It simplifies the uvm_config_db to Python's Config class.
14 & 15	Sequences, sequencer, and sequence items	PyUVM refactors the sequencer functionality to create a simpler implementation.
17	UVM Register Enum	PyUVM implements all the basic Enum types in the Register Access Layer (RAL).
18	UVM Register Block	It implements the RAL register block classes.
19	UVM Register Field	PyUVM implements the register fields as defined in the standard. Except for a few exceptions, like atomic Backdoor access. Field byte access and single-file access during reading or writing operations.
20	UVM Register	The main register class is defined, but is still missing a backdoor and the used backdoor to be leveraged from the Cocotb force.

21	UVM Register Map	The main register map class should be refactored to guarantee simplicity and backdoor access.
23	Register Item	Register item used across multiple classes.
24	Register Include File	Includes other Enum and types to be merged with s17.
25	UVM Register Adapter	Main register adapter.
26	UVM Register Predictor	The main register predictor should be disabled if auto_prediction is not set.
27	Register Package	The main package included should behave similarly to uvm_reg_pkg.

Table 4: PyUVM current implementation of IEEE1800.2 [8].

The following sections in this document provide an overview of the DUT employed in this comparative analysis. This overview begins with the Open Compute Project (OCP), the industry initiative that developed the Data Center Modular Hardware System (DC-MHS) and its component, the Data Center Security Control Module (DC-SCM). The discussion then focuses on the LVDS Tunneling Protocol Interface (LTPI), which serves as the communication protocol between the DC-SCM and the Host Processor Module (HPM), and constitutes the design specification used to evaluate both SystemVerilog and Python verification environments.

3.8 LVDS Tunneling Protocol & Interface “LTPI”

3.8.1 Open Compute Project “OCP”

The Open Compute Project (OCP) is a collaboration initiative founded by Facebook (now Meta) in 2011 to revolutionize data center infrastructure through open-source hardware design and standardization. Originally conceived to address the inefficiencies and proprietary limitations of traditional data center equipment, OCP has evolved into a global community that develops and shares open hardware specifications for servers, storage systems, networking equipment, and data center facilities. The project’s fundamental philosophy centers on transparency, efficiency, and collaborative innovation, enabling organizations to build more cost-effective and energy-efficient data center infrastructure while reducing vendor lock-in [25].

OCP’s primary functions encompass the development of open hardware specifications, reference designs, and best practices across multiple technology domains. The organization operates through specialized project groups that focus on areas such as server design, networking hardware, storage solutions, rack and power infrastructure, and emerging technologies like artificial intelligence and edge computing. These working groups collaborate to create detailed specifications that are freely available to the community, enabling manufacturers to produce compatible hardware and allowing end users to customize solutions according to their specific requirements. The project also facilitates knowledge sharing through conferences, technical workshops, and documentation repositories [25].

The OCP community includes major technology companies, hardware manufacturers, cloud service providers, and research institutions. Key contributors include founding member Meta, along with industry leaders such as Microsoft, Google, Intel, AMD, NVIDIA, Dell Technologies, HPE, and numerous other organizations spanning the entire data center ecosystem. This diverse membership ensures that OCP specifications address real-world deployment challenges while incorporating cutting-edge technological innovations. The collaborative nature of the project has resulted in widespread adoption of OCP-compliant hardware across hyperscale data centers, enterprise environments, and cloud infrastructure providers, fundamentally transforming how the industry approaches data center design and procurement [25].

3.8.2 Data Center-Secure Control Module "DC-SCM"

The Data Center-Secure Control Module (DC-SCM) defines a standardized approach to server management and security by consolidating common server management, security, and control features from traditional processor motherboard architectures onto a smaller, standardized form factor module. This modular design incorporates all firmware states previously distributed across typical processor motherboards into a separate module. From a data-center perspective, DC-SCM enables consistent management and security deployment across a broader range of platforms while facilitating management and security upgrades within platform generations without requiring redesign of complex components. For developers, this architecture allows solution providers to separate customer-specific requirements from complex components such as motherboards, enabling greater component leverage across diverse platforms [26], [27].

The DC-SCM architecture consists of the following primary elements

a) BMC – The Baseboard Management Controller is a specialized service processor that monitors the physical state of the server.
b) BMC Flash – At least one Flash device used to contain the BMC firmware image.
c) BIOS Flash – At least one Flash device used to contain the BIOS firmware image.
d) DC-SCM PLD – A programmable logic device that contains optional LTPI logic and any required additional application-specific logic.
e) RoT Security Processor – An optional security processor responsible for attesting the BMC, BIOS, and/or other firmware images on the system.
f) TPM – The Trusted Platform Module is a dedicated microcontroller designed to secure hardware through integrated cryptographic keys [26], [27].

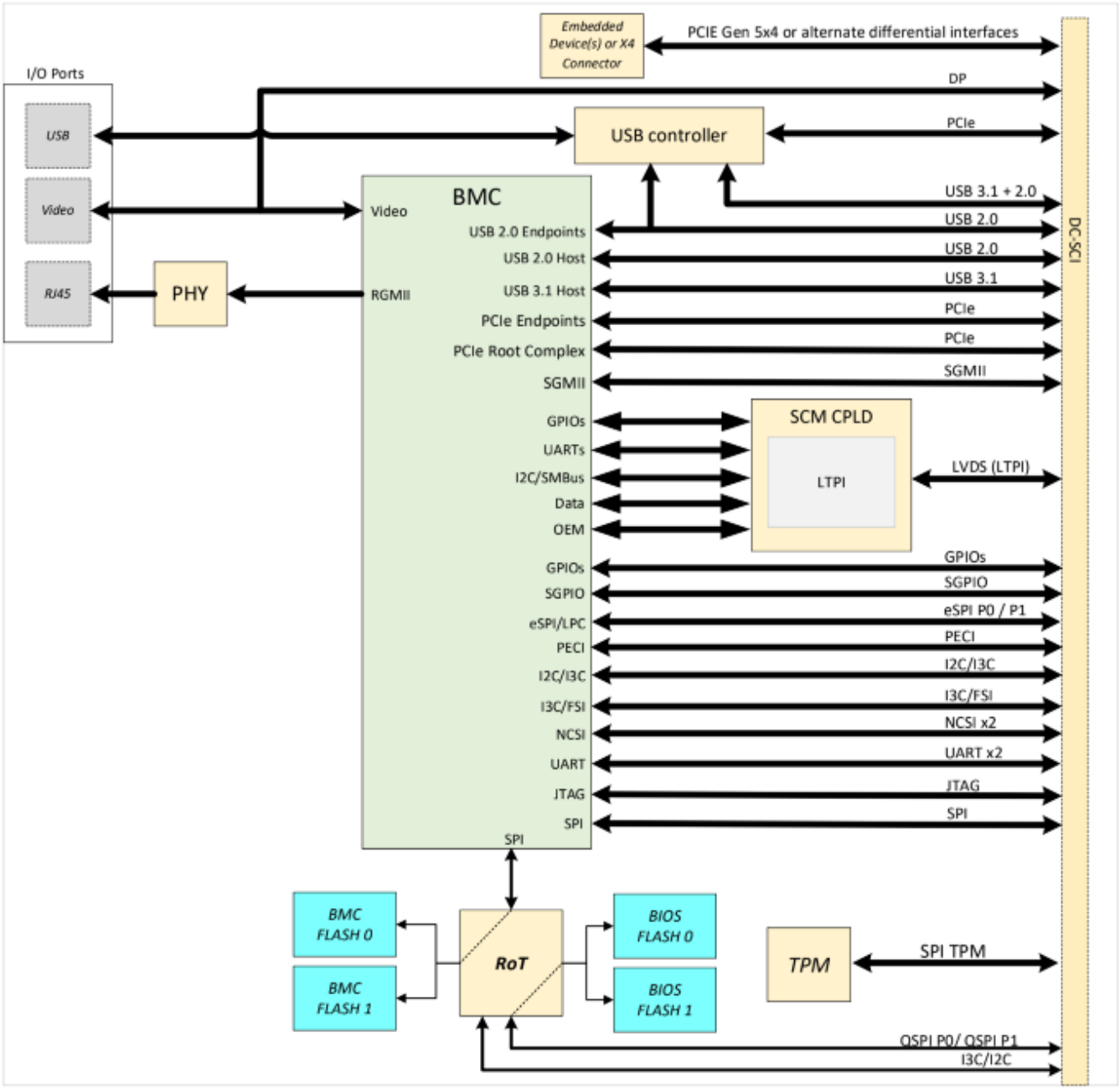


Figure 9: DC-SCM Example Block Diagram [26].

3.8.3 DC-SCM 2.0 LTPI Architecture

LVDS Tunneling Protocol & Interface (LTPI) is a protocol and interface designed for tunneling several low-speed signals between the Host Processor Module (HPM) and the Security Control Module (SCM) in a server platform. The LTPI protocol goes over the Low Voltage Differential Signals (LVDS) electrical interface supported by a majority of CPLDs and FPGAs. The LVDS allows for tunneling of several interfaces, such as GPIOs, and low-speed serial interfaces such as SMBus, I2C, and UART. It is also extensible with additional proprietary OEM interfaces and provides support for raw data tunneling between the HPM FPGA and the SCM FPGA. [27], [28]. The LTPI interface implements a concept of channels, where each channel is mapped to a specific type of physical interface on the SCM and HPM. Currently, the following channels are defined for the interface:

a) GPIO Channel
b) I2C/SMBus Channel
c) UART Channel
d) OEM Channel
e) Data Channel

The summary of the LTPI Channels is listed below:

LTPI Channel	Required	Description
GPIO	Yes	Tunnels GPIO signals from HPM to SCM and vice versa. The GPIO Channel allows for differentiation between Low Latency and Normal Latency GPIOs to allocate more bandwidth for time-critical GPIO signals and allow for scalability in extending the number of tunneled GPIOs.
I2C/SMBus	No	Tunnels I2C/SMBus Links from SCM to HPM and vice versa. The definition of DC-SCM LTPI I2C/SMBus is limited to a use case with a single I2C/SMBus Controller, which is located on either side of a given LTPI Link.
UART	No	Tunnels full-duplex UART interfaces with flow control support between SCM and HPM.
OEM	No	This channel is a placeholder for any proprietary OEM interface that can be tunneled with LTPI. This is for bandwidth allocation for any non-standard interfaces.
Data	No	The Data Channel allows for tunneling random access to addressable memory-mapped spaces. The intention is to allow remote access to standardized memory-mapped buses used to interconnect CPLD/FPGA IPs.

Table 5: LTPI Channels Summary [28].

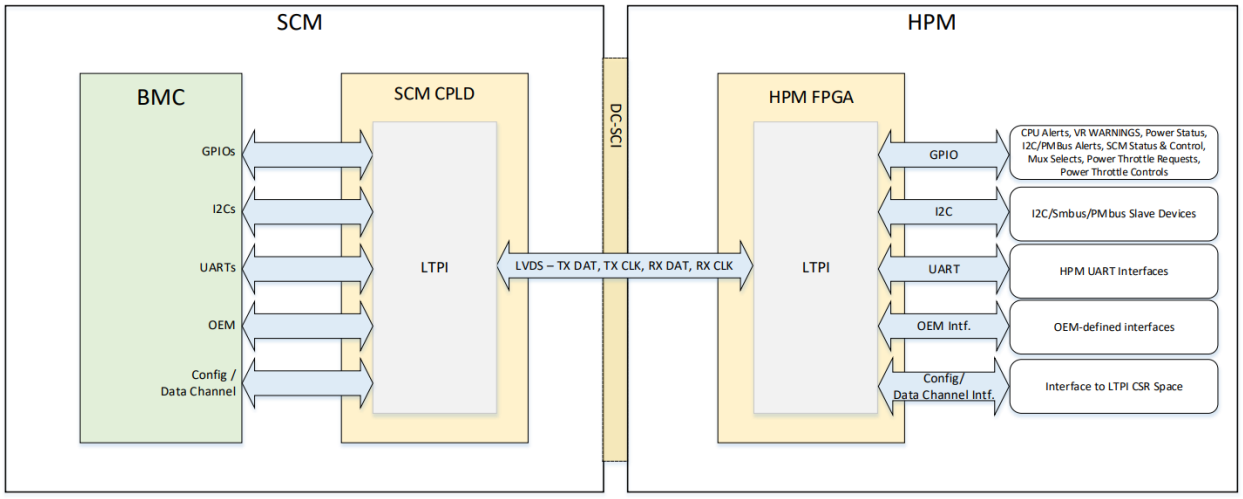


Figure 10: LTPI connection between HPM and SCM [28].

The LTPI uses Time Division Multiplexing (TDM) of a high-speed LVDS full-duplex link to transmit and receive LTPI channels between the SCM and HPM. In every equal time slot, T_N on the LVDS link, there is one LTPI frame being transmitted. Each LVDS channel is provided with a portion of the LVDS frame transported in each direction. The number of bits assigned to a specific channel in each frame sent over the LTPI interface is directly proportional to the LTPI bandwidth dedicated to each channel. To minimize the latency impact on LTPI channels, the frames are assumed to be transmitted back-to-back without any inter-frame gaps, even if channel states are not changing between frames. [28].

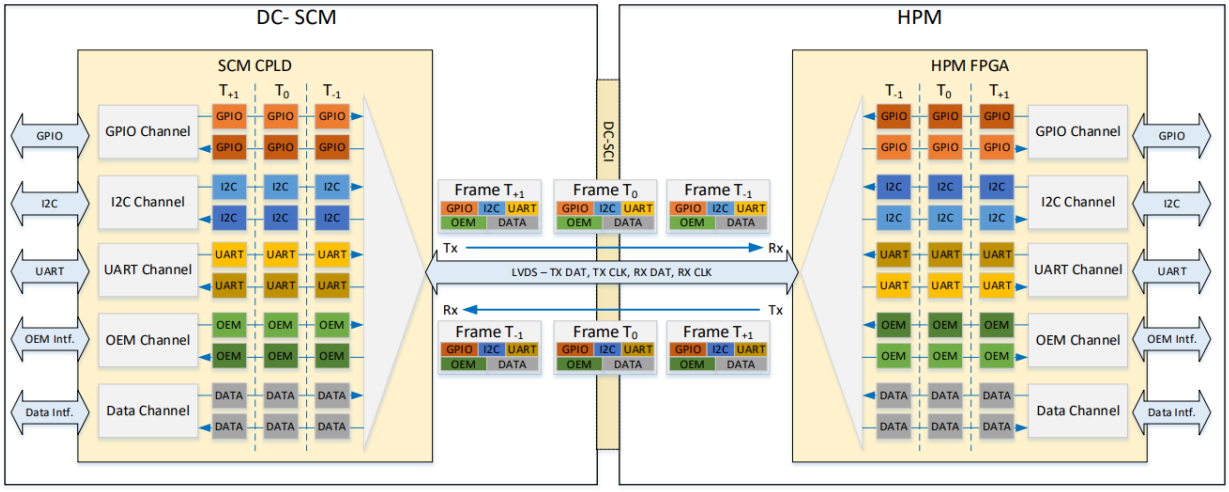


Figure 11: LTPI Channels encapsulation and serialization using the TDM method [28].

The LTPI interface uses two LVDS links in each direction. One of the links is a clock used to synchronize the transaction (TX) and reception (RX) on both sides of the link. Both sides of the link use an independent clock. The clock source synchronous mode is used for the TX clock source to align the phase shift on the LVDS IO links. The RX side will use the incoming clock as a sampling clock for the data link.

Each LTPI frame contains the CRC checksum. This is CRC-8 following the polynomial:

$$x^8 + x^2 + x^1 + 1$$

The polynomial initial value is defined as '0'. The CRC is calculated for the entire payload field after the Comma symbol of the LTPI frame [28].

After encapsulation of the data and the CRC generation, the LTPI encodes the data into 8b/10b encoding. It translates 8-bit characters into 10-bit symbols. Three of the most important features of 10b encoding are embedding a clock into the data, maintaining DC balance, and enhancing error detection. The LTPI uses AC coupling between the HPM and SCM, and hence, the 10b encoding is required to maintain DC balance. The definition of 8b/10b encoding is considered a standard practice outside of the scope of the LTPI specification, and it recommends the use of the IBM definition [28].

3.8.4 GPIO Channel

The GPIO Channel is used to tunnel low-speed HPM and SCM GPIOs through the LTPI interface. It defines low-latency and normal-latency GPIOs. It is a particular platform design choice which GPIO is mapped to low latency and to normal latency. The low-latency GPIOs are sent in each frame of the LTPI link, whereas normal-latency GPIOs are split across multiple frames, with a frame ID counter used to identify the GPIO subset [28].

3.8.5 I2C/SMBus Channel

The I2C/SMBus Channel is used to tunnel several buses through the LTPI interface for the links where there is only one controller on either the SCM or the HPM. The primary use case addressed by the DC-SCM LTPI I2C/SMBus tunneling is where the BMC acts as a controller of the link with Target devices located on the HPM. From the BMC perspective, the I2C/SMBus buses are accessible the same way as if they were physically routed through the Security Control Interface (SCI). The tunneling uses the clock stretching method defined in the I2C specification to compensate for the LTPI latency and turnaround time. The LTPI implements the event-based model where the encoded I2C/SMBus events are captured on one side and recovered on the other side of the LTPI. When an event is generated by the controller (i.e., START condition), the controller's local SCL line is stretched by the LTPI I2C/SMBus Relay until a remote event is received. When it is received, the controller's local bus recovers the event and releases the stretch condition [28].

3.8.6 UART Channel

The UART Channel is used to tunnel physical UART interfaces between the SCM and the HPM through LTPI. The channel supports tunneling of several full-duplex UART interfaces with flow control signals. The UART interface is tunneled through LTPI and reconstructed into a physical UART interface similar to GPIO and I2C/SMBus channels. [28].

The following signals are tunneled through LTPI:

a) TXD - Transmit data UART output signal from the UART Controller.
b) RXD - Receive data UART input signal from the UART Controller.
c) CTS - Clear to Send UART input signal from the UART Controller.
d) RTS - Ready to Send UART output signal from the UART Controller.

3.8.7 OEM Channel

The OEM channels allow for OEM-specific interface tunneling through LTPI. The OEM can implement proprietary specific interfaces by following the same approach as used in other channels [28].

3.8.8 Data Channel

The Data channel allows for tunneling data between the SCM and HPM FPGAs. The Data channel follows a concept of a Memory Mapped interface, such as Avalon Memory Mapped (AVMM) or Wishbone Bus. The Data channel implements an internal AVMM bus that allows for mapping memory spaces between the SCM and HPM FPGAs and the use of standardized IP blocks that follow AVMM bus support. Data Channel accesses are done on demand only when there's a request to write or read the data [28].

3.8.9 LTPI Frame

The LTPI defines several frame types that are used during different phases of LTPI operation. All frames used on the LTPI are aligned to 16 symbols, including the Comma Symbol and CRC, totaling 160 bits per frame. The symbols used for Frame Comma are the following:

Comma Symbol	Frame Type	Description
K28.5	Link Detect and Link Speed	The initial frame transmitted during the LTPI bring-up flow provides information about supported speed capabilities. It is also used to achieve DC balance on the link.
K28.6	Advertise capabilities and Configure/Accept	The frame is used to advertise supported capabilities (channels and features) and configure enabled capabilities on both endpoints.
K28.7	LTPI Operational Frame	Frame used during normal LTPI operation after interface initialization and configuration are completed. It carries all the channels supported by the given LTPI implementation.

Table 6: LTPI Frame Types [28].

The full description of each LTPI frame can be found in [28] pages 46-55, including clock frequency, frame offsets, and definitions for each Frame Type.

3.8.10 Control and Status Registers

The LTPI interface provides a configuration space register, which is primarily defined for the BMC to have control over the LTPI operation. The basic usage of LTPI CSR is to allow the BMC to access link status information, capabilities information, channel operational telemetry information, and reset capabilities. It is out of the scope of the LTPI definition to decide which interface is used by the BMC to access the LTPI CSR space. A specific design might use any interface with appropriate mapping in an FPGA to expose the LTPI CSR to BMC, such as I2C, I3C, SPI, UART, etc. The full specification of the CSR table can be found in [28] pages 55-59.

3.8.11 LTPI Link Training

The LTPI link training starts after the SCM FPGA and HPM FPGA come out of reset. It is executed on a base frequency of 25MHz Single Data Rate (SDR) mode. It is the initialization of the link where both sides start sending information to each other to establish what the operating conditions will be in terms of link speed, operational buses, and capabilities. The link training and initialization need to be executed in the following scenarios:

a) After SCM and HPM FPGA's power lost events
b) After SCM and HPM FPGA's reset de-assertion
c) On request from external entities such as the BMC
d) After a link lost error condition occurs.

The main purpose of the LTPI link training is to achieve the DC balance condition on the link, exchange link speed capabilities, switch to operational link frequency, exchange LTPI features capabilities, enable the selected operational configuration of LTPI, and to switch to operational state where it will continue for the remainder of the connection [28]. The link training is defined as a state machine on both sides of the link, and it can be divided into 3 major stages:

a) Link Training – Initial state of link initialization
b) Link Configuration – Configuration of LTPI features
    a. Exchange LTPI features capabilities
    b. Enable the selected operational configuration of the LTPI
    c. Switch to operational mode
c) Link Operational – State where the link will spend most of the LTPI lifecycle, transacting its data from one side of the link to the other.

The state machine is presented in Figure 13. The full description of the LTPI Link Training, including the conditions to switch states, the difference between Target and Controller state machines, and the error conditions that would result in a link lost error and link retraining process, can be found in [28] pages 60-74.

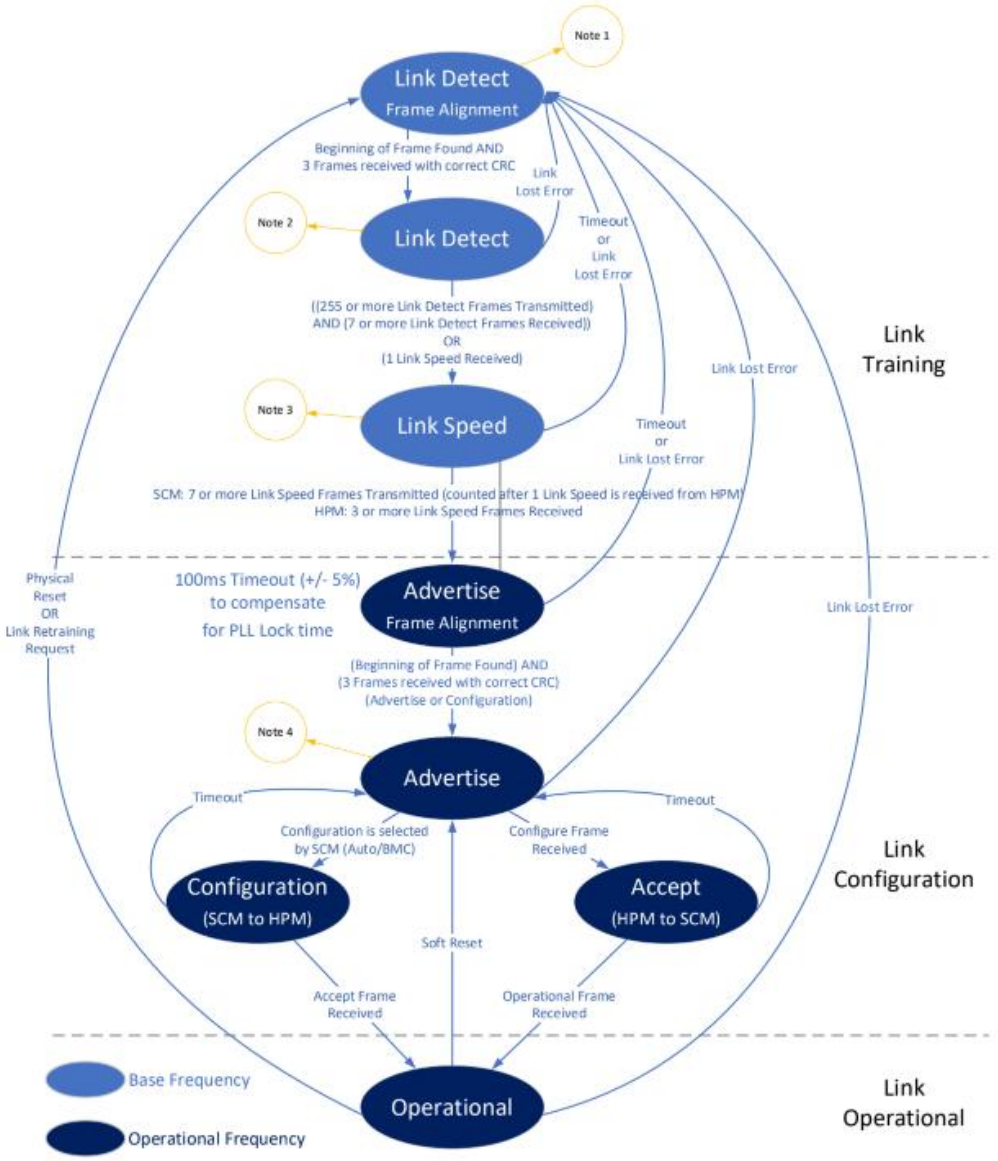


Figure 12: LTPI Link Training and Initialization Flow [28].

***Note 1**: The time it takes to align to the beginning of the frame depends on the time needed to achieve the DC-balance and how efficiently the comma symbol can be found (depends on SERDES design). This stage is the source of initial misalignment between SCM LTPI and HPM LTPI by means of the number of Frames Sent and Received. The number of Frames sent shall not be included in the minimum Frame Detect frames transmitted (255).*

***Note 2**: This stage is expected to be entered by SCM and HPM at different timings due to initial misalignment in the first stage. This means that one side can complete this stage earlier and move to the Link Speed stage. If it happens, then the 'slower' part shall switch to Link Speed even if the required number of TX and RX Frames is not achieved.*

***Note 3**: Both sides keep sending Link Speed. SCM is required to send a minimum of 7 Link Speed Frames, while HPM is required to receive 3 Link Speed Frames.*

***Note 4**: In advertise state, since the LTPI clock is reconfigured to a higher speed, the LTPI needs to re-align to the beginning of the frames again, similarly to the Link Detect stage. Typically, PLL is going to be reconfigured and requires some time to lock. To compensate for that, a 1ms timeout is used on both sides.*

4. Methodology

4.1 Design of Experiments

This research employs a controlled comparative experimental design to evaluate the effectiveness, usability, and performance characteristics of traditional UVM versus Cocotb/PyUVM verification frameworks. Moving forward, the SystemVerilog and UVM environment will be called the SystemVerilog framework, and its counterpart, the Python environment, comprised of Cocotb and PyUVM, will be called the Python framework. The experimental approach is structured to provide quantitative and qualitative metrics that enable objective assessment of both verification frameworks while maintaining rigorous scientific standards to ensure the validity and reliability of results. The DUT that will be used to test these verification environments is the LTPI protocol due to the complexity of this protocol and the fact that it is used in an industrial application, as part of the OCP components meant for server standardization. The purpose of this experiment is to test both verification environments as they would be used in an industrial scenario, utilizing the industrial simulator Synopsys VCS, testing a protocol that is currently used as part of a server platform infrastructure, and reviewing the performance metrics to determine if they both fit an industrial workflow.

4.1.1 Controlled Variables and Parameters

To ensure experimental validity, several variables are maintained as constants across both verification implementations. The DUT remains identical for both frameworks, utilizing the same LTPI RTL implementation with consistent design specifications, interface definitions, and functional requirements. The LTPI RTL selected for this experiment is the GitHub project given by the OCP Foundation, developed by Katarzyna Krzewska, Kasper Wszolek, and Kevin Kifer. All repository URLs can be found in Appendix B [29], [30], [31]. The simulation environment parameters are standardized, including the simulator, simulator version, compilation flags (except for the vpi flag needed for Cocotb to work appropriately), optimization settings, and runtime configurations. Test stimulus complexity and coverage targets are maintained at equivalent levels, with identical functional verification plans applied to both SystemVerilog and Python frameworks. Hardware platform specifications, including processor architecture, memory allocation, and operating system configurations, remain constant throughout all experimental runs to eliminate performance variations attributable to infrastructure differences.

4.1.2 Independent Variables

The primary independent variable in this study is the verification language employed, traditional UVM implemented in SystemVerilog versus Cocotb/PyUVM implemented in Python. This fundamental distinction encompasses the programming language paradigm, development environment, and associated toolchain dependencies. Secondary independent variables include the level of verification engineer experience with each methodology, implementation approach variations within each framework, and the specific verification components utilized as implemented in their respective languages.

4.1.3 Dependent Variables and Metrics

The experimental design measures multiple dependent variables to comprehensively evaluate both methodologies. Development metrics include total implementation time, lines of code required, and debugging time. Performance metrics encompass simulation runtime, memory utilization, compilation time, and regression execution duration. Quality metrics evaluate functional coverage achievement, code coverage percentages, branch and toggle coverage metrics, bug detection effectiveness, and verification completeness. Usability metrics assess learning curve characteristics, code readability scores, maintainability indices, and reusability potential. Additionally, qualitative metrics capture developer experience feedback, documentation quality assessment, and industry adoption feasibility.

4.1.4 Experimental Control and Fair Comparison Strategy

To ensure a fair comparison between methodologies, several experimental controls are implemented. Both verification environments are developed by the same engineering team to minimize skill-level bias, with adequate training provided for both SystemVerilog and Python approaches. The verification plan, test scenarios, and coverage goals are identical across both implementations, ensuring equivalent verification scope and depth. Simulation infrastructure remains consistent for all experimental runs. Statistical significance is ensured through multiple experimental iterations, with results averaged across multiple runs to account for variability.

4.1.5 LTPI as Device Under Test Selection

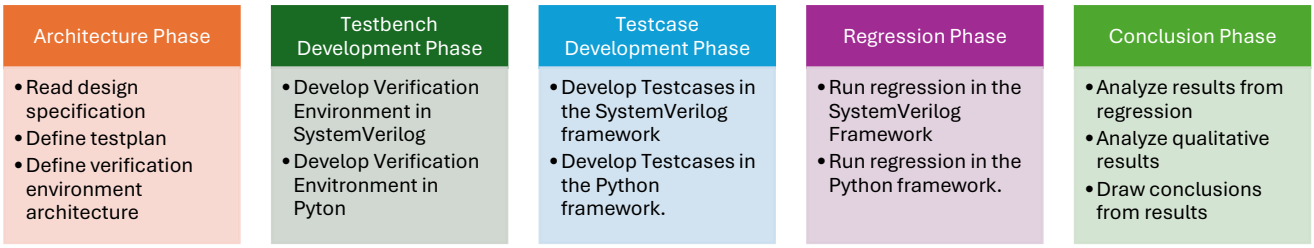
The LTPI protocol serves as the DUT for this comparative analysis due to its appropriate complexity level and verification requirements. LTPI provides sufficient design complexity to demonstrate the capabilities of both verification methodologies while remaining manageable within the scope of this research. The protocol encompasses multiple verification challenges, including state machine validation, protocol compliance checking, error injection and recovery, protocol timing verification, and multi-channel data integrity validation. This complexity level enables meaningful comparison for both methodologies' capabilities in handling real-world verification scenarios typical of modern semiconductor designs. The LTPI specification provides clear functional requirements and expected behaviors, facilitating objective assessment of verification completeness and effectiveness.

4.1.6 Dual Verification Environment Strategy

The experimental approach involves developing two complete, functionally equivalent verification environments: one implemented using traditional UVM in SystemVerilog and another using Cocotb/PyUVM in Python. Both environments target identical LTPI DUT implementations and follow the same verification plan, ensuring that differences in results can be attributed to the verification methodology and implementation language rather than scope or implementation variations.

Each environment includes the same testbench components, including agents and environments for all LTPI channels (GPIO, UART, SMBus, Memory Channel), monitors for protocol compliance and performance measurements, scoreboards for data integrity verification, and coverage collectors for completeness assessment. The dual implementation approach enables direct comparison of development effort, verification effectiveness, and maintenance requirements between the two methodologies while providing concrete evidence for the analysis.

4.2 Simplified Diagram



4.3 Proposed Solution

1. Define the test plan for LTPI sequence verification.
2. Define the verification environment's architecture for LTPI verification.
3. Develop the verification environment using the SystemVerilog framework.
4. Develop the testcases defined for LTPI verification in the SystemVerilog framework.
5. Develop the verification environment using the Python framework.
6. Develop the test cases defined for LTPI verification in the Python framework.
7. Run cycles of regression in both verification environments for the LTPI protocol, each cycle incrementing the number of repetitions per test that the regression must do. Identify in each regression the time it takes to simulate, and the coverage collection achieved.
8. Gather qualitative metrics such as ease of development and learning curve.
9. Compare regression results, both in terms of coverage collection and time to simulate.
10. Compare the ease of development in both methodologies and the setbacks of each language to draw conclusions from the results.

4.4 Verification Architecture

The verification architecture represents the foundational framework upon which the functional validation of the LTPI protocol implementation is conducted. This section delineates the components that comprise both SystemVerilog and Python frameworks, establishing the structural elements necessary for systematic protocol verification. To provide a precise comparison between these environments, the architecture definition is the same for both, except for minor differences that happen due to the languages these environments are written in. These differences will be explained later in this document. The architecture encompasses the interconnected verification components, including testbench infrastructure, stimulus generation mechanisms, response monitoring systems, coverage collection frameworks, and result analysis tools that collectively enable thorough validation of the DUT.

4. Methodology

Figure 14 encompasses the verification architecture defined for the LTPI testbench and further discussion on this topic will include a description of the main components of this architecture and the relationship between these components.

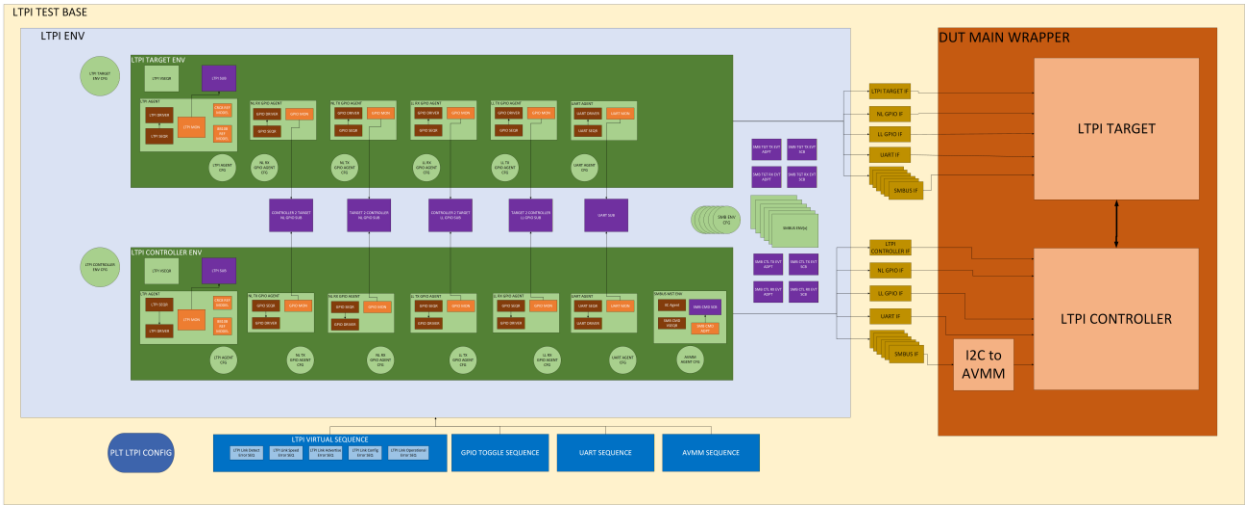


Figure 13: LTPI Verification Environment Architecture.

The right side of the diagram illustrates the DUT Main Wrapper; this module contains the entirety of the design that will be tested. LTPI is comprised of a target and a controller side sending data between them. The verification environment connects to the data input and output of these modules, and the wrapper connects both sides to form the link. The I2C to AVMM module was added as part of the DUT to simulate how the BMC would interact with the LTPI's AVMM memory map. In a server platform, the BMC contains an SMBus interface that connects to the FPGA to provide access to the CSR map, and the FPGA provides this I2C to the AVMM translator and connects this to the memory map of the LTPI.

The DUT connects to the LTPI Environment through several interfaces. To get a reusable environment, each interface is designed to work as a standalone connection for a specific protocol that the LTPI contains, as well as 2 extra interfaces meant to contain variables that generate link training errors into the design and monitor internal signals like state machines and control logic. The interfaces designed for this DUT are an SMBus interface, a GPIO interface, a UART interface, a target interface, and a controller interface.

The SMBus interface is instantiated 6 times per side of the link since the LTPI can carry up to 6 SMBus connections within its frames. The GPIO interface is instantiated 2 times for each link, one for the normal latency GPIOs and another for the low latency connection. The UART interface is instanced one for each side of the link. The controller side instances an extra SMBus interface connected to the I2C to AVMM module to get access to the Memory Map, and the target and controller interfaces are specific for their respective control and status signals. These interfaces are declared in the main wrapper, connected to their respective IOs, and set in the UVM factory so that the test base can get them and pass them on to their respective environments and agents.

The left side of the diagram illustrates the LTPI verification environment, which contains all the modules required for the tests defined in the verification plan. The LTPI environment is divided into two main sections: one environment for the target connection and another for the controller connection. Leveraging the encapsulation and reusability that UVM provides, these environments contain agents and sub-environments meant to contain the modules required to test each protocol that travels through LTPI, as well as an agent per side for error injection to stress the training and error handling capabilities of the link. The structure of these modules follows UVM methodology, where agents contain sequencer, driver, and monitor components. The scoreboards for the different protocols are instantiated in the LTPI Top environment since monitors from both sides of the connection send data to these scoreboards to ensure that data transmitted by one side is correctly received on the other side of the link.

The target and controller environments are nearly identical in structure. These environments share four instances of the GPIO agent each: one for transmission, another for reception of normal latency GPIOs, and an additional pair for low latency GPIOs. Each environment also contains an instance of a UART agent. The primary differences between the target and the controller environment are that each side has a specific agent responsible for LTPI error injection during the training stage, as well as monitoring and checkers for frame and state machine validation. The controller side includes, in addition to the aforementioned components, an instance of an SMBus master environment whose purpose is to simulate the BMC and send transactions to the memory map for accessing LTPI registers.

Each of these agents and environments has a class called environment/agent configuration. The purpose of these classes is to take the interfaces obtained in the base test and pass those instances to internal components such as drivers and monitors, ensuring that these components are connected to the interface declared in the main wrapper. The only components that are not contained within the target or controller environments are the SMBus environment instances. This design decision was made because these SystemVerilog UVM modules are leveraging code from previous projects and are structured such that one environment provides two agents, one for transmission and another for reception of an SMBus channel. Therefore, it was considered optimal to maintain the existing functional structure and port the code in its current form.

4.4.1 GPIO Agent

The GPIO agent is parameterizable for x number of ports and consists of a monitor, sequencer, driver, and analysis port. For low-latency GPIOs, this module is parameterized for 16 ports as defined by the LTPI specification; for normal latency GPIOs, these agents are parameterized for 1024 ports, simulating the GPIO connection of a server platform between HPM and SCM. The driver is responsible for taking information provided by the sequence and toggling the value of a specific GPIO port, simulating traffic on that 1-bit GPIO. The GPIO monitor creates x number of channels, one for each GPIO port, and each channel monitors for any change in value in the interface variable “m_gpio_value”. When one of these ports changes value, the channel that detects it captures the timestamp when this change occurred and sends the value and timing information of the event to the scoreboard.

4. Methodology

For proper monitoring and data checking functionality, the agent that sends GPIO data on one side of the LTPI link must be connected to the agent that receives GPIO data on the other side of the link through the scoreboard. The GPIO agent responsible for monitoring data reception is configured as a passive agent, so its driver is not created during simulation. The GPIO scoreboard contains one FIFO per GPIO port for each side of the link, that is, one FIFO for the transmission and another one for the reception for each bit. Each FIFO receives the information sent by its respective monitor during simulation time, and during the check phase, the scoreboard unpacks all the data sent to it and compares the values that transmission and reception had to ensure that data arrived appropriately. Additionally, it compares the time required for data transmission across the link to ensure that data is transmitted within the time specified by the LTPI protocol. If the data does not match or the transaction time exceeds the specification limits, the scoreboard generates an error and prints it to the console. Finally, the scoreboard prints to the console the transaction time information, the longest time it took to transmit, the shortest time, and the average time.

GPIO sequences are created and configured within a testcase. Each sequence can test only one GPIO port, requiring the creation of N sequences for the N ports to be exercised on each side of the link. Each sequence must receive information about how many times the connected port value should be toggled. If the repetition number is 0, only one data toggle is sent to that port. If the repetition number is greater than zero, the time delay between transactions must be specified or randomized. This approach allows tests to randomly define how many ports and which GPIO ports will be exercised, as well as randomize the number of transactions transmitted on each port, with each operating independently of the others. Figure 15 represents the LTPI environment components that are exclusive to the GPIO testcases.

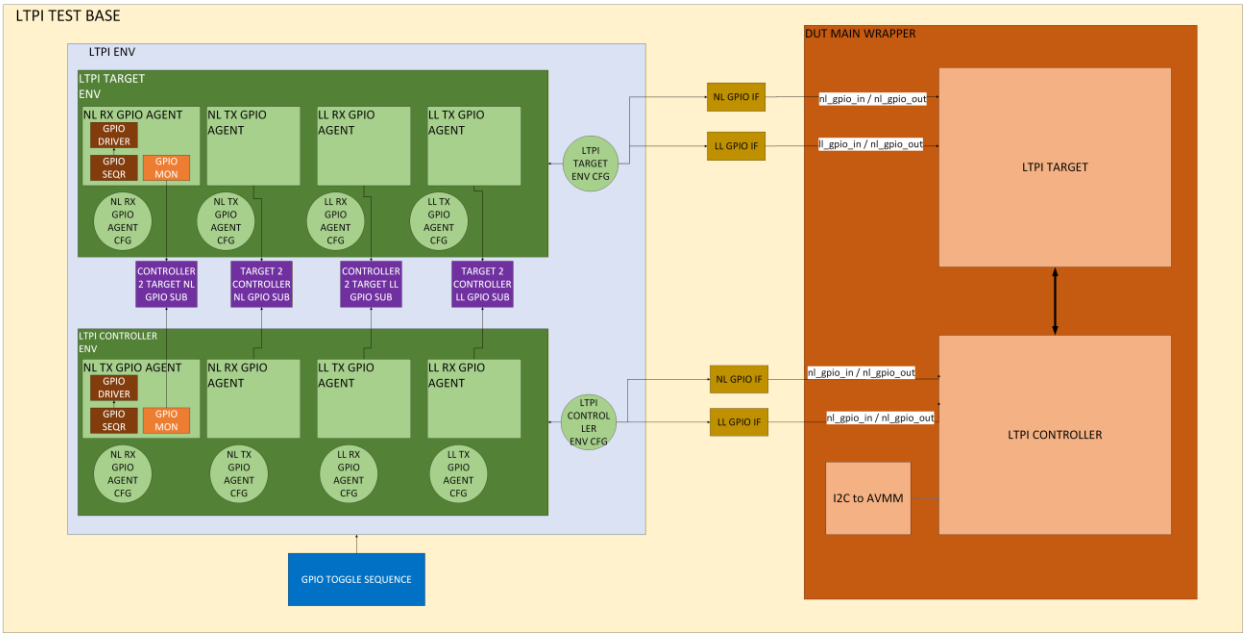


Figure 14: LTPI GPIO Environment components.

4.4.2 UART Agent

The UART agent is parameterizable to configure the number of bits transmitted per transaction, with a default setting of 8 bits without a parity bit. Like the GPIO agent, it contains a driver, monitor, sequencer, and an agent configuration class, the latter being responsible for sending the interface between the test base and the driver and monitor. This environment configuration class is created in the target and controller environments and sent through the UVM factory to the UART agent. Additionally, the UART agent has two analysis ports, one for data transmission and another for data reception. Unlike the GPIO agent, this works as an active agent always, since, according to the LTPI specification, the UART operates in full-duplex mode, meaning both sides of the link send and receive data simultaneously.

UART sequences are created within a test, requiring one for the target and another for the controller. The sequence receives from the test the number of transactions to be sent, with one transaction as the default. The sequence receives from the test the data to be transmitted and the baud rate at which it will be sent. The baud rate is specified through an enumeration which contains the common values of 4800, 9600, 19200, 38400, 57600, and 115200 Bd, and this value can also be randomized. The driver receives this information from the sequence and, upon receiving each sequence item, it calculates the signal period based on the baud rate and transmits the data through the UART Tx line.

The UART monitor, during the main phase of the test, launches two parallel tasks from which it does not exit during simulation time. One task monitors the transmission line while the other monitors the reception line. The test provides the monitor with the baud rate, and when both tasks start, they wait for the START bit, calculate the period, and begin capturing the data detected on their respective lines. Once a transmission is completed, both processes send information to the scoreboard through their analysis ports.

The UART scoreboard is instantiated in the LTPI Environment Top. It is connected to the transmission and reception of both sides of the link and contains four analysis ports connected to individual FIFOs, each of which stores the data for Tx and Rx of each side. During simulation time, the scoreboard launches 2 tasks, each task first waits for the transaction FIFO of one side of the link to receive data, then waits for the reception FIFO on the other side of the link to receive data. This data is printed on the console and then compared using a checker to ensure that data sent by one side of the link is correctly received by the other side. These processes terminate when the simulation’s main phase ends. Figure 16 below illustrates the LTPI environment components that were described above and that are exclusive to the UART test cases.

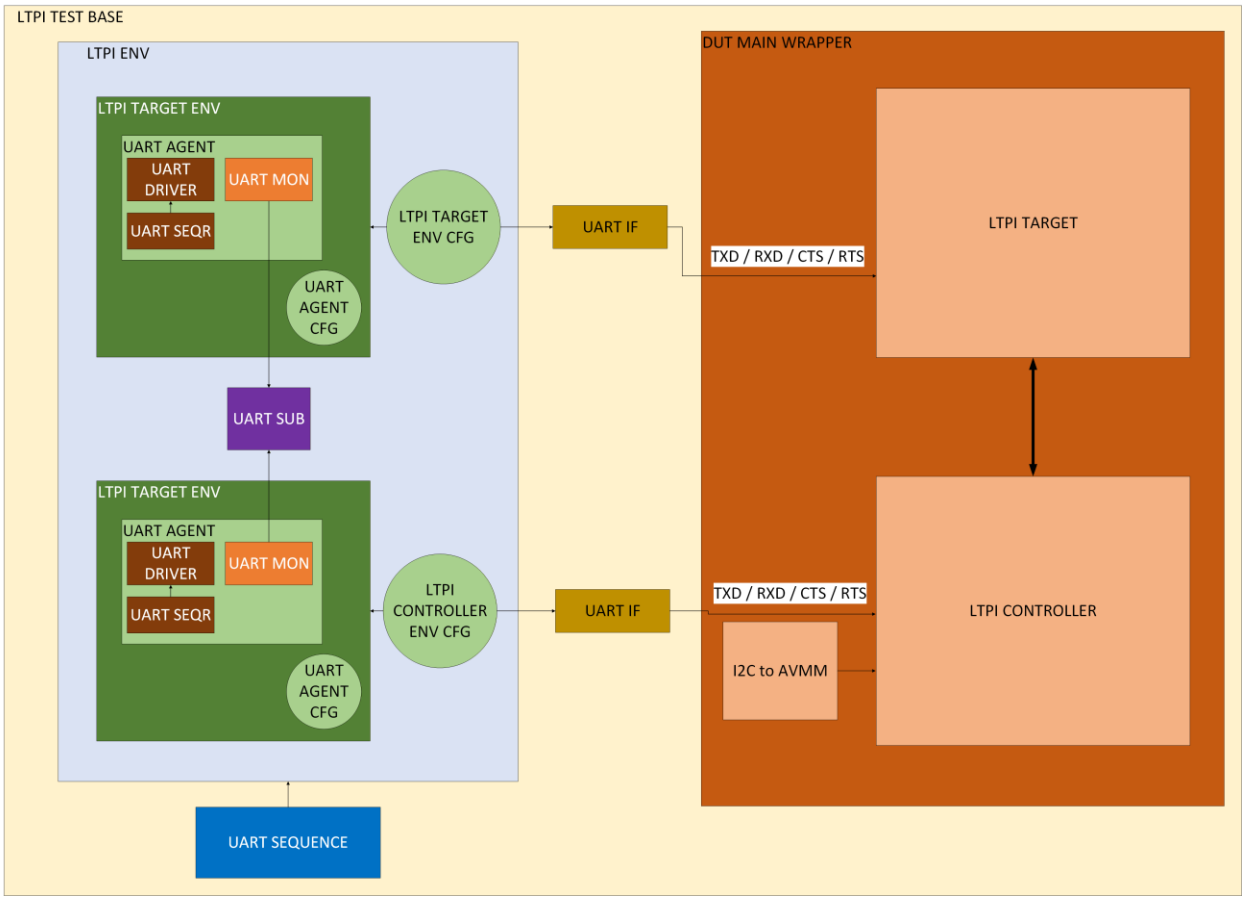


Figure 15: LTPI UART Environment components.

4.4.3 SMBus Environment

The SMBus environment is instantiated at the LTPI Environment Top. This environment can be configured to function as an SMBus master component, a slave component, or both at the same time, where the DUT serves only as the medium through which the SMBus transactions are communicated. The LTPI protocol sends SMBus transactions from one side to the other; however, it does not generate the transactions by itself, so the environment is configured such that both sides of the link have bus functional models (BFM) and the LTPI transmits the data from one side to the other.

Each SMBus environment instance contains two SMBus agents, which are only utilized when the DUT functions as one side of the channel, so they are not created in the LTPI environment. Additionally, the environment has two I2C agents responsible for handling the BFM generation. The I2C agent is a reuse of Carsten Thiele's work, which consists of an active agent without a monitor that handles I2C transactions.

The sequences of this agent are hierarchically organized into two levels. The lowest level sequences are specific bits of an I2C frame, with sequences such as Master Start Bit, Master Stop Bit, Slave Tx Bit, Slave Rx Bit, etc., transmitting 1 bit of I2C each. These implementations are instances of the I2C driver. The second level of sequences uses this first level to construct I2C bytes; however, an additional layer is needed to construct complete frames that include a Start bit, Address, R/W bit, Stop, etc.

Above this I2C agent implementation, a virtual sequence is utilized whose main function is to construct SMBus frames through state machines. The first set of virtual sequences is designed to act as a Master BFM, while the second set acts as the Slave BFM. Each virtual sequence is meant to simulate the following SMBus commands:

1. Quick Command
2. Send Byte
3. Receive Byte
4. Write Byte
5. Write Word
6. Read Byte
7. Read Word
8. Block Write
9. Block Read

A virtual sequence is used to simultaneously call the master and slave sequences, which in turn use state machines that call I2C sequences to construct the frames and enable communication between both sides. According to the LTPI specification, a multi-master SMBus model is not supported, and in a server platform, the transmission Master is the BMC, so the controller side will always work as a master, while the LTPI target side will act as a slave.

These state machines send control and status information to the SMBus Command Adapter class during simulation time. Data such as FSM state, transmitted and received data are received by the SMBus Command Adapter, which packages this information into a sequence item and sends it to the SMBus Command Scoreboard, responsible for storing this information in FIFOs. When both master and slave sides send information to the scoreboard, it performs a comparison between both sequence items using the compare method and sends an error to the console if they are not equal. Each adapter sends the sequence item to the scoreboard once it has detected that the sequence has sent the SMBus Stop bit, so the comparison occurs each time an SMBus transaction completes. Figure 17 illustrates the LTPI environment components that are specific to the SMBus testcases.

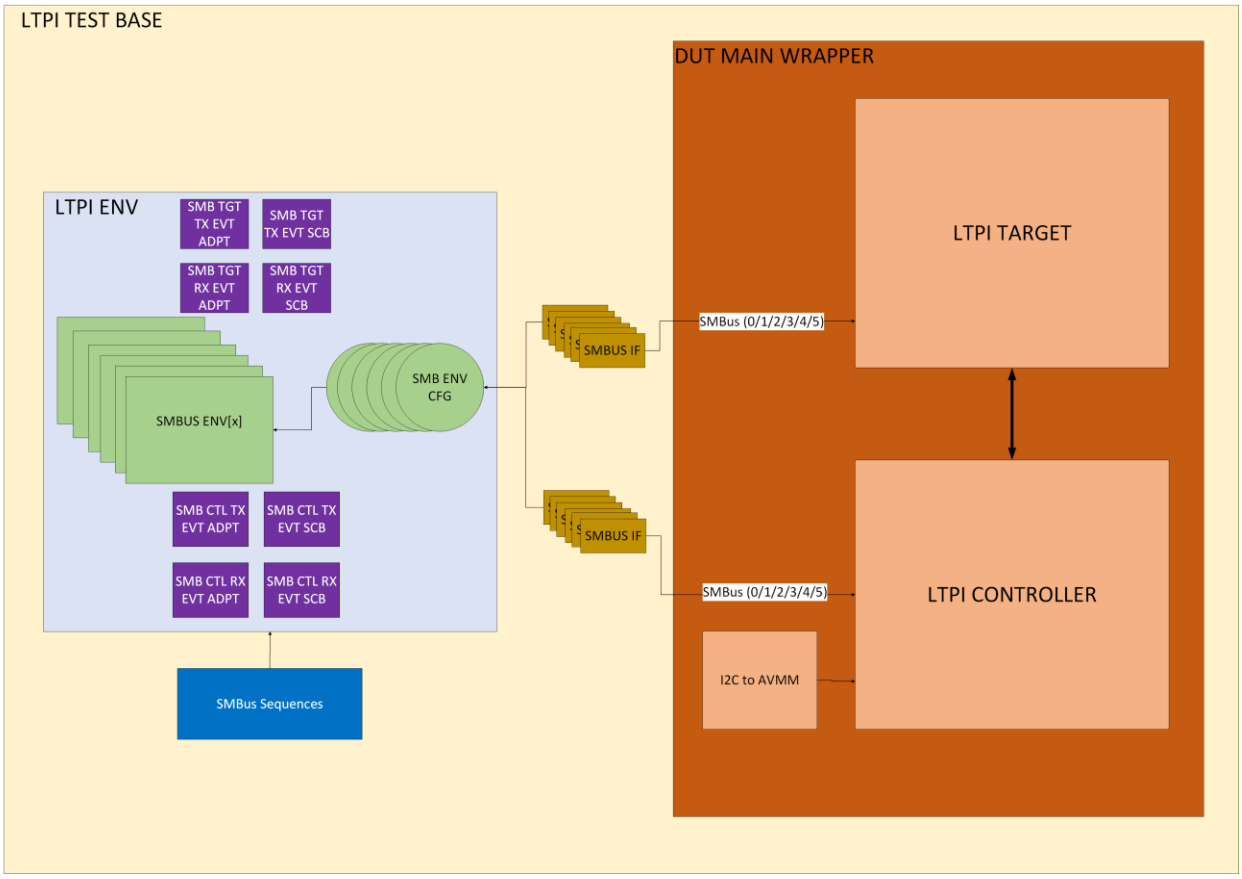


Figure 16: LTPI SMBus Environment components.

4.4.4 SMBus Master Environment

The SMBus master environment is a modification of the SMBus environment discussed previously. The difference is that this model always works as a master BFM; it does not have the SMBus agent responsible for functioning as a passive agent. Additionally, since it is only an SMBus master, it does not have two copies of the components and contains only one set of the I2C agent, monitor, and scoreboard.

The primary difference lies in the test and sequence implementation. The testcase instantiates a class called “plt_csr_config”. This class contains methods and attributes necessary for randomizing registers in the memory map. First, this class randomizes whether the target or controller registers will be accessed, then whether it will perform a read or write operation. With this information, it randomizes a register address, and, in the case of a write operation, it also randomizes the data that will be written into the register. The testcase instantiates this class and can be randomized or configured to test specific accesses. By default, the test randomizes this class and calls a virtual sequence. Like the SMBus environment mentioned in the previous section, the virtual sequence functions as an FSM that calls I2C sequences to construct SMBus frames compatible with the memory map. It sends a device address, 16-bit register address, and proceeds to read or write the desired register. Figure 18 shows the components in the LTPI environment responsible for SMBus master access and their connection to the DUT.

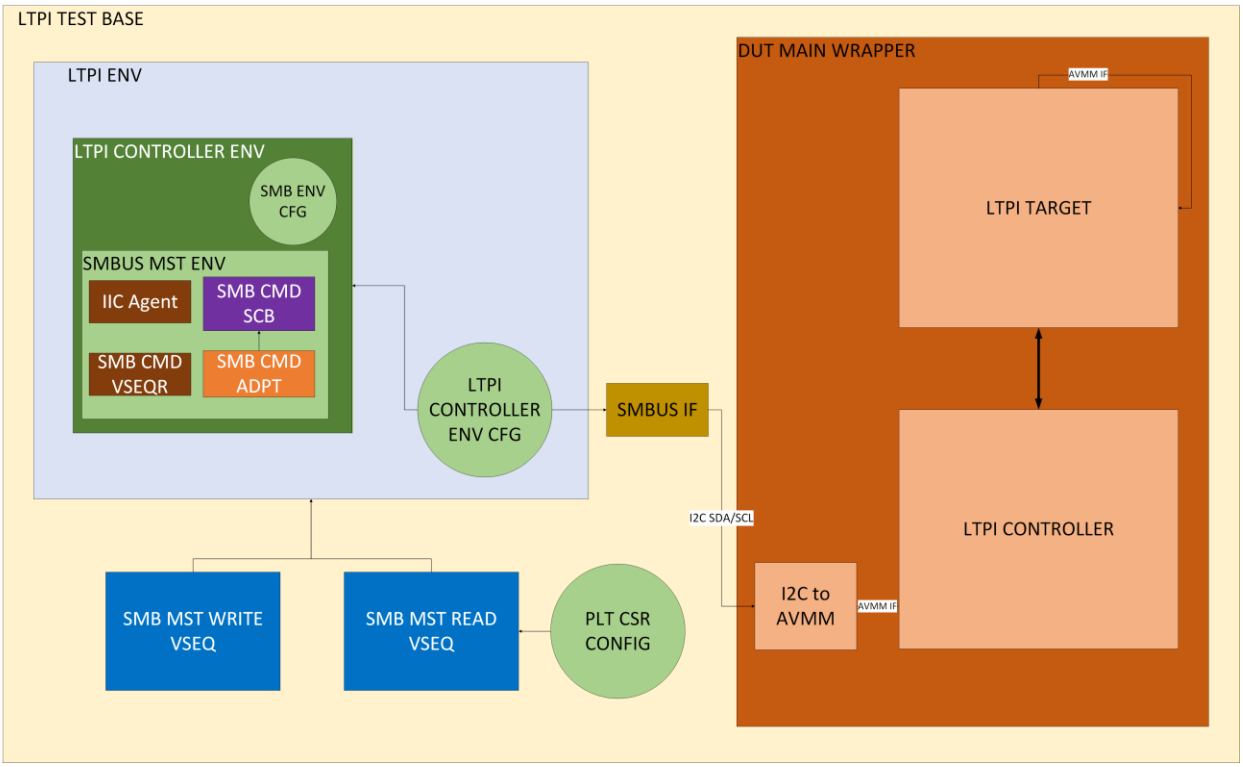


Figure 17: LTPI Master SMBus Environment components.

4.4.5 LTPI Target and Controller Agent

The target and controller agents are instances within their respective environments. These agents are virtually identical except for minor differences in the types of errors they can generate. The test cases utilizing these agents must configure the “ltpi_config” file, which contains the necessary attributes for randomizing different error types and various LTPI training stages during which these errors can be injected. A test can specify particular errors when targeting specific error characteristics; however, by default, tests randomize this configuration object and invoke a virtual sequence after. The virtual sequence encompasses multiple sub-sequences, each designed to reproduce the configurable errors defined by the ltpi_config class. These sequences are categorized according to the LTPI link training stage where the error is meant to be injected, with the virtual sequence managing their execution. The sequence transmits error information to the corresponding driver based on the link side targeted for error generation, which then enables a multiplexer on the LTPI line to inject the desired error condition.

The training stages, and consequently the existing sequences, include Link Detect, Link Speed, Link Advertise, Link Configuration or Accept, and Link Operational phases. Error types vary depending on the training stage, with complete documentation available in [28]. However, all stages include error handling for CRC failures, unexpected frame comma errors, and timeout conditions.

The sequence can be configured to inject the exact number of errors specified in the documentation to trigger a retraining process, or inject more or fewer errors to verify the margin thresholds. The driver implements error injections through a multiplexer connected between the LTPI transmission and reception lines, one multiplexer per side. When an error is meant to be driven, the multiplexer is activated so that the receiving side of the link receives driver-generated data instead of the actual transmission, thereby preventing multi-driven conflicts on the line.

LTPI monitors are connected to their respective interfaces with the primary purpose of ensuring proper LTPI training and sequence flow. A checker monitors the link training FSM states changes and logs them to the console, while a second checker continuously measures training completion time under both error and error-free conditions for console reporting. The monitor incorporates four reference models: two to verify the correctness of the CRC field of the frame, both in the transmission and reception lines, while the other two validate proper 8b10b encoding implementation. An additional checker categorizes whether driver-generated errors are correct and verifies appropriate FSM behavior, including transitions to Link Lost Error state followed by retraining sequences. Furthermore, the monitor transmits FSM information to a subscriber containing a covergroup that ensures all possible FSM paths are exercised during regression testing. Figure 19 shows the aforementioned LTPI target and controller agents and its relationship between files.

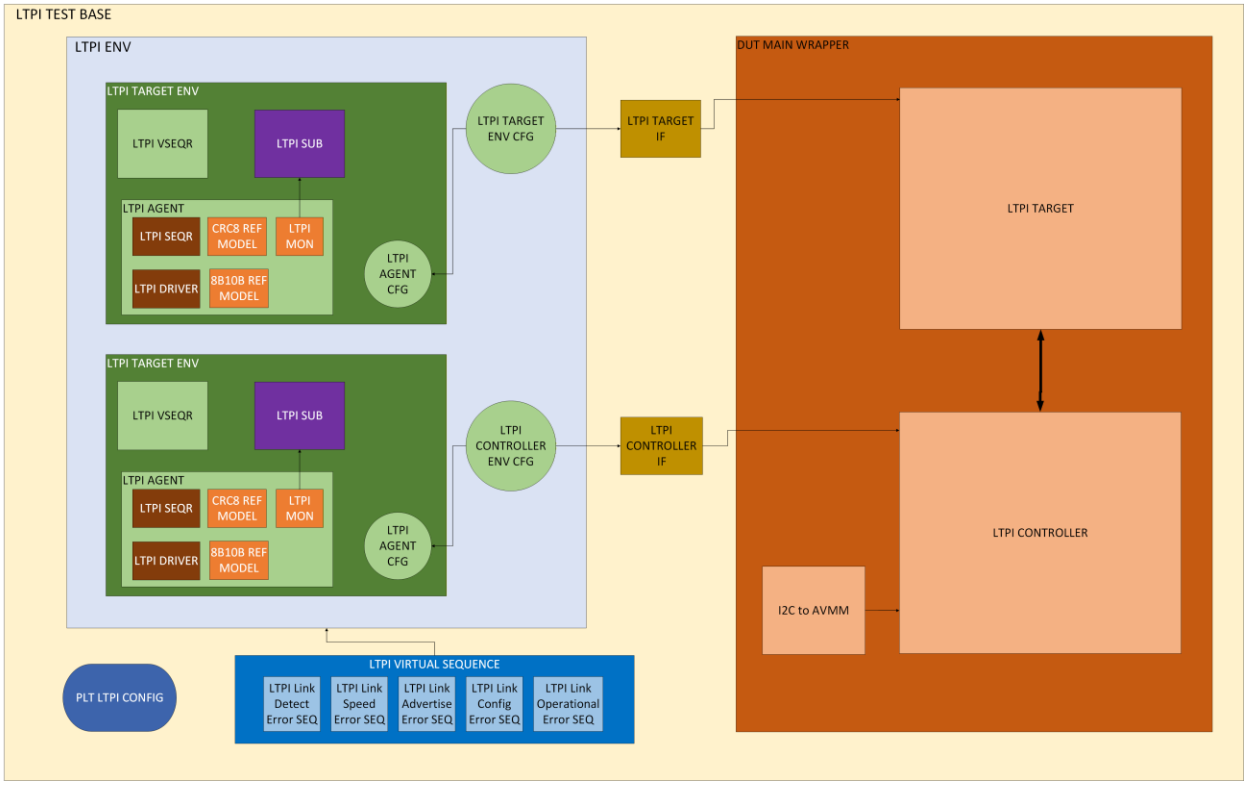


Figure 18: LTPI Link Training Error Injection Environment components.

4.5 Coverage Metrics

Regression testing in pre-silicon verification represents a systematic and automated process of re-executing a test suite to ensure the design modifications, bug fixes, or feature additions have not introduced unintended functional errors or compromised existing functionality. This verification practice involves running hundreds or thousands of previously developed test cases against the RTL design, encompassing directed tests for specific features, constrained random tests for broader coverage exploration, and stress tests for performance validation. Regression suites are typically executed across multiple simulation seeds to ensure statistical significance and are run frequently throughout the design cycle, particularly after code changes, to maintain verification confidence. The regression process serves as a quality gate in the verification workflow, providing rapid feedback on design integrity and enabling verification teams to quickly identify failures, assess verification completeness, and make informed decisions about design readiness for tape-out. There are several metrics that verification engineers review after a regression to ensure that a design has been tested thoroughly. These metrics are called coverage, and there are several types that a simulator provides after a regression.

Functional coverage is a verification metric that measures how thoroughly a testbench has exercised the intended functionality and design features of a DUT, focusing on whether the verification process has adequately tested the design's behavioral requirements and specifications rather than simply tracking code execution. This metric is implemented through covergroups and coverpoints that define specific events, signal transitions, operational conditions, and feature indications that must occur to demonstrate comprehensive verification, such as testing all possible combinations of input values, verifying all protocol states are reached, or ensuring critical design features operate correctly under various scenarios. Functional coverage serves as a quantitative measure of verification completeness, enabling verification engineers to identify gaps in their test scenarios, guide the development of additional test cases, and provide objective evidence that the design has been thoroughly validated, ultimately reducing the risk of undetected functional.

Line coverage, also known as statement coverage, is a fundamental code coverage metric that measures the percentage of executable lines or statements in the RTL source code that have been executed during simulation, providing insight into how thoroughly the testbench has exercised the design's implementation at the most basic code level. This white-box verification technique tracks whether each line of code has been reached at least once during testbench execution, helping identify dead code, unreachable statements, or insufficient stimulus generation that leaves portions of the design untested. While line coverage is easily measurable and provides a baseline understanding of code exercise, it represents the most superficial level of coverage analysis since a line can be executed without necessarily validating all possible behaviors or conditions within that statement, making it a necessary but insufficient metric for verification quality assessment.

Branch coverage and toggle coverage represent more sophisticated code coverage metrics that provide deeper insight into the design exercise and validation completeness. Branch coverage measures the percentage of conditional branches or decision points in the RTL code that have been executed in both directions during simulation, ensuring that control flow structures such as if-else statements, case statements, and ternary operators have been tested for both true and false conditions, thereby validating all possible execution paths through the code. Toggle coverage, on the other hand, measures the percentage of signals that have transitioned between their possible logic states during simulation, tracking whether single-bit signals have experienced both 0 to 1 and 1 to 0 transitions and whether each bit in multi-bit signals has toggled through all states. Together, these metrics help identify unused logic, stuck signals, incomplete stimulus generation, and potential optimization opportunities while ensuring that all implemented functionality is exercised during verification, making them essential for validating interface signals, control paths, and data buses under dynamic operating conditions.

5. Results

This chapter presents the results of the comparative analysis performed in the SystemVerilog and Python frameworks through quantitative measurements and qualitative assessments conducted using the LTPI protocol as the DUT. The experimental evaluation encompasses four primary categories of metrics: code development characteristics, performance measurements, quality assessments through various coverage metrics, and usability evaluations examining the learning process and usability of both frameworks. The results provide empirical evidence for evaluating the trade-offs between both frameworks, addressing key industry concerns regarding verification development efficiency, runtime performance, verification completeness, and long-term maintainability. The usability metrics assessments acknowledge the inherent subjectivity influenced by the engineering team's established expertise in SystemVerilog and the UVM framework. These findings enable evidence-based decision-making for verification teams considering methodology adoption while highlighting the fundamental differences in resource utilization, development approaches, and verification capabilities between the two frameworks.

5.1 Code Development Metrics

5.1.1 Total Implementation Time

The engineering team, comprised of one designer and two advisors, already possessed a solid background in SystemVerilog and UVM beforehand. However, reaching this level of proficiency in SystemVerilog and the UVM framework required approximately three years of dedicated learning and practice. For this analysis, acquiring competency in Python along with the Cocotb and PyUVM libraries took approximately three months, leveraging the existing foundation in object-oriented programming principles and UVM framework concepts.

Developing the SystemVerilog verification environment took approximately 3 months, considering that the IIC agent code was leveraged from other projects. Some files already existed for this environment; however, several components required adaptation to function exclusively within the LTPI verification framework. Developing the Python verification environment required approximately four months, though it is important to note that the SystemVerilog framework was fully operational at that point, allowing for the leveraging of existing architectural designs and their translation into the new language and libraries. The modules that took the longest to develop (approximately 6 weeks) were the IIC and SMBus modules. Because of its complexity and the number of modules that this library contains in SystemVerilog, developing these files and debugging them consumed most of the development time for the Python framework.

5.1.2 Lines of Code

The verification environment file structure remained the same for both frameworks, as well as the testcases developed and the environment architecture. To have a fair comparison, these results will only showcase the directories where the environment files are contained, excluding the design directory, as well as the ones containing the compilation and regression scripts, and the waveform file directory.

The results illustrated in Tables 7 and 8 demonstrate that the Python environment required 8,235 fewer lines of code compared to the SystemVerilog framework:

SystemVerilog framework Directories	Lines of Code
Sv	22,000 lines
Tests	2,097 lines
Top	1,103 lines
Total	25,200 lines

Table 7: Lines of code of the SystemVerilog framework.

Python framework Directories	Lines of Code
Sv	14,913 lines
Tests	1,280 lines
Top	772 lines
Total	16,965 lines

Table 8: Lines of code of the Python framework.

5.2 Performance Metrics

To perform simulations and regressions, the simulator used is Synopsys VCS version 2022. The tests were run on a Linux server. The workstation had assigned 6 cores; the CPU is an Intel Xeon E5-2643 @3.40GHz. With 32GB of available memory allocated.

5.2.1 Simulation Time

To measure the simulation time taken to run a specific test, testcases 00, 10, and 21 were run 5 times each in both frameworks, and the results were averaged out [1]. Table 9 shows the results of each testcase. Testcase 00 is the link training testcase of LTPI, and all other testcases are needed for this training to occur, so it represents the minimum time a simulation can take. Testcase 10 tests the NL GPIO transactions, and testcase 21 covers a memory map operation on the target side of the link. Testcase 00 ran, on average, 490.49 seconds faster in the SystemVerilog framework than in Python. Testcase 10 ran 621.59 seconds faster in the SystemVerilog framework as well, and testcase 21 ran 848.98 seconds faster. On average, this indicates that on singular testcase runs, the SystemVerilog framework runs 2.59 times faster than the Python framework.

Testcase ID	SystemVerilog Framework Simulation Time (s)	Python Framework Simulation Time (s)
00	321.11 s	811.60 s
10	481.34 s	1102.93 s
21	443.04 s	1291.02 s

Table 9: Simulation time of individual testcases in SystemVerilog and Python frameworks.

[1] Appendix A contains the testplan, explaining the purpose of the testcases.

5.2.2 Memory and CPU Utilization

To measure the memory and CPU utilization that a simulation costs to the system, the testcase 00 was simulated five times in each environment. All simulations were performed on the same hardware. The results shown in Tables 10 and 11 demonstrate that the SystemVerilog framework required, on average, 532.53 MB of memory, whereas the Python framework required, on average, 820.71 MB. Regarding CPU utilization, 100% means that 1 CPU core was used on average at maximum capacity, and a higher number represents the parallelism it was able to perform with a second core. On average, the SystemVerilog framework utilized 1.36 cores, and the Python framework used 1.04 cores:

Simulation	SystemVerilog Framework Maximum Memory Utilization (MB)	Python Framework Maximum Memory Utilization (MB)
1	481.87 MB	826.38 MB
2	559.00 MB	832.24 MB
3	576.10 MB	788.27 MB
4	565.75 MB	818.33 MB
5	479.92 MB	838.32 MB

Table 10: Memory Utilization for SystemVerilog and Python frameworks for testcase 00.

Simulation	SystemVerilog Framework Percent of CPU cores used (%)	Python Framework Percent of CPU cores used (%)
1	139%	105%
2	126%	105%
3	137%	107%
4	138%	99%
5	140%	104%

Table 11: CPU Utilization for SystemVerilog and Python frameworks for testcase 00.

5.2.3 Compilation Time

Although Python does not need to compile to run since it is an interpreted language, it still needs to compile the Verilog and SystemVerilog files that comprise the DUT, as well as the DUT main wrapper file. The compilations for both frameworks were performed in the same Linux workstation with the same hardware. The compilation test was performed 5 times for each framework. The average results are displayed in Table 12, where the Python framework can compile 4.24 seconds faster than its SystemVerilog equivalent:

Framework	Compilation time (s)
SystemVerilog	51.42 s
Python	47.18 s

Table 12: Compilation time for SystemVerilog and Python frameworks.

5.2.4 Regression Execution Time

Several regressions were performed for this analysis to review the amount of simulation time it takes in both environments, to compare if simulation time grows linearly or exponentially depending on the number of cycles, and to gather information about which framework was faster. Each regression ran a total of 19 testcases with different amounts of cycles per testcase. Table 13 below shows the number of cycles per testcase each regression ran, and the amount of time in seconds each regression took. The relationship between the times it takes to run a certain number of testcases in a regression is shown in Figure 20, where the data shows that, on average, the Python framework takes 7.54 times longer to finish running the same number of testcases than its SystemVerilog counterpart:

Cycles in regression	SystemVerilog Framework Regression Time (s)	SV Regression Time (d:hr:min:s)	Python Framework Regression Time (s)	Python Regression Time (d:hr:min:s)
1	4,017 s	0 d, 1 hr., 6 min, 57 s	28,346 s	0 d, 7 hrs., 52 min, 26 s
4	18,589 s	0 d, 5 hrs., 9 min, 49 s	136,741 s	1 d, 13 hrs., 59 min, 1 s
10	41,127 s	0 d, 11 hrs., 25 min, 27 s	313,761 s	3 d, 15 hrs., 9 min, 21 s
20	80,258 s	0 d, 22 hrs., 17 min, 38 s	644,048 s	7 d, 10 hrs., 54 min, 8 s
50	204,278 s	2 d, 8 hrs., 44 min, 38 s	1,598,846 s	18 d, 12 hrs., 7 min, 26 s
100	451,940 s	5 d, 5 hrs., 32 min, 20 s	3,320,763 s	38 d, 10 hrs., 26 min, 3 s

Table 13: Regression execution time for SystemVerilog and Python frameworks.

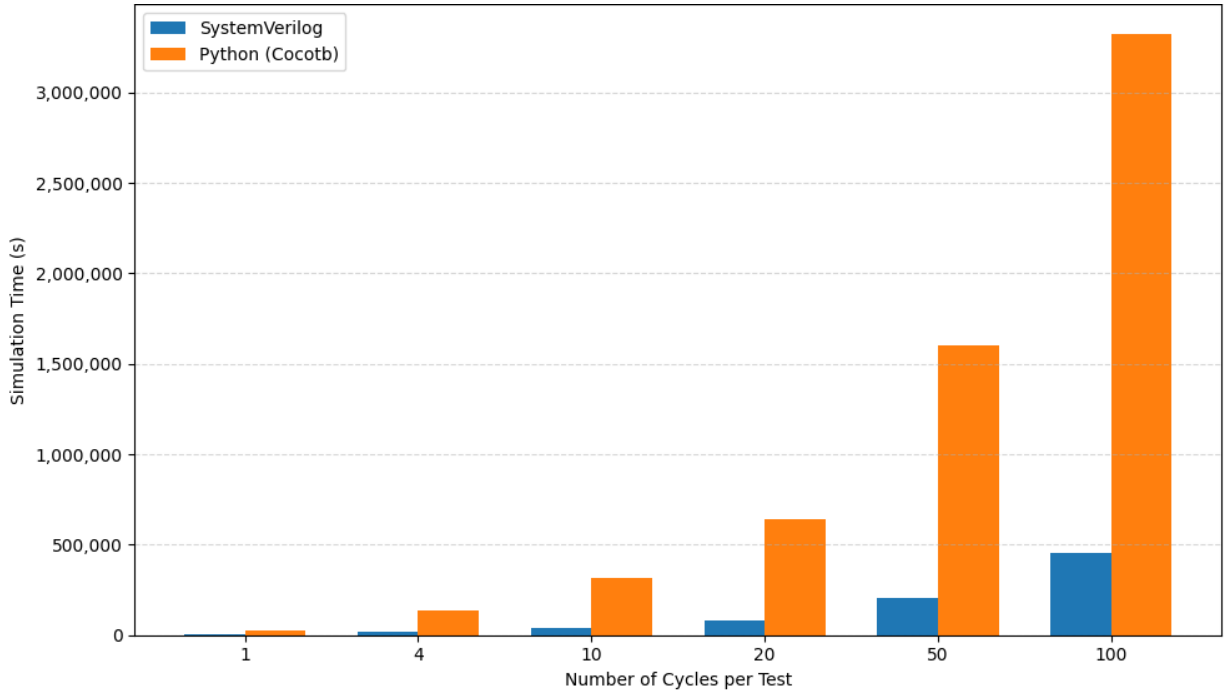


Figure 19: Time relationship between SystemVerilog and Python framework's regression.

5.3 Quality Metrics

5.3.1 Functional Coverage

SystemVerilog natively supports covergroups and coverpoints constructs, which enable the development of functional coverage. In Python, neither Cocotb nor PyUVM natively supports this feature. A third library called PyVSC was developed externally to provide support for random verification stimulus generation and coverage collection [32]. However, despite multiple installation attempts, the library could not be successfully downloaded and configured for proper operation. Consequently, functional coverage collection was not achievable in the Python framework. Table 14 shows the functional coverage gathered for the SystemVerilog framework, reaching a maximum of 93.54% at 20 cycles of each testcase in the regression:

Cycles in regression	SystemVerilog Framework Functional Coverage (%)	Python Framework Functional Coverage (%)
1	77.42%	N/A
4	91.93%	N/A
10	91.93%	N/A
20	93.54%	N/A
50	93.54%	N/A
100	93.54%	N/A

Table 14: Functional coverage for SystemVerilog framework.

5.3.2 Code Coverage

The code coverage achieved in both frameworks is given automatically by the compiler. Table 15 shows the amount of line coverage gathered by different regression runs, increasing the number of cycles per testcase in both environments. On average, the SystemVerilog framework rendered 0.79% more coverage than its Python counterpart:

Cycles in regression	SystemVerilog Framework Line Coverage (%)	Python Framework Line Coverage (%)
1	68.06 %	67.89 %
4	68.99 %	68.03 %
10	69.22 %	68.19 %
20	69.17 %	68.45 %
50	69.57 %	68.70 %
100	69.67 %	68.67 %

Table 15: Code coverage for SystemVerilog and Python frameworks.

5.3.3 Branch Coverage

Table 16 illustrates the branch coverage gathered by both environments during the regression testing mentioned previously. On average, the Python framework regression resulted in 6.76% more branch coverage than in the SystemVerilog framework:

Cycles in regression	SystemVerilog Framework Branch Coverage (%)	Python Framework Branch Coverage (%)
1	56.89%	64.58%
4	57.94%	64.60%
10	58.55%	65.16%
20	58.67%	65.44%
50	59.55%	66.03%
100	59.72%	66.05%

Table 16: Branch coverage for SystemVerilog and Python frameworks.

5.3.4 Toggle Coverage

Table 17 shows the toggle coverage results that both environments collected during the regression cycles, where, on average, the SystemVerilog framework gathered 7.34% more coverage than the Python framework:

Cycles in regression	SystemVerilog Framework Toggle Coverage (%)	Python Framework Toggle Coverage (%)
1	23.41%	20.76%
4	29.36%	21.33%
10	32.78%	24.09%
20	35.55%	26.89%
50	39.91%	30.68%
100	40.81%	34.00%

Table 17: Toggle coverage for SystemVerilog and Python frameworks.

5.4 Usability Metrics

The following results represent qualitative assessments that are inherently subjective and influenced by the evaluator's background and experience. The assessment of these characteristics relies on professional judgement regarding code clarity, documentation quality, architectural organization, and the perceived ease of future modifications or component reuse, all of which can vary significantly based on individual programming experience and methodological preferences.

5.4.1 Learning Curve

Python demonstrated to be significantly easier to learn than SystemVerilog. Figure 21 below displays a score comparing both languages. The metrics taken into consideration were OOP features supported, where Python supports OOP in all its components, and the number of keywords in each environment, which can result in the number of definitions that an engineer needs to learn to become proficient and fully exploit each language. The number of commands related to verification, and how complex it is to use UVM in each language.

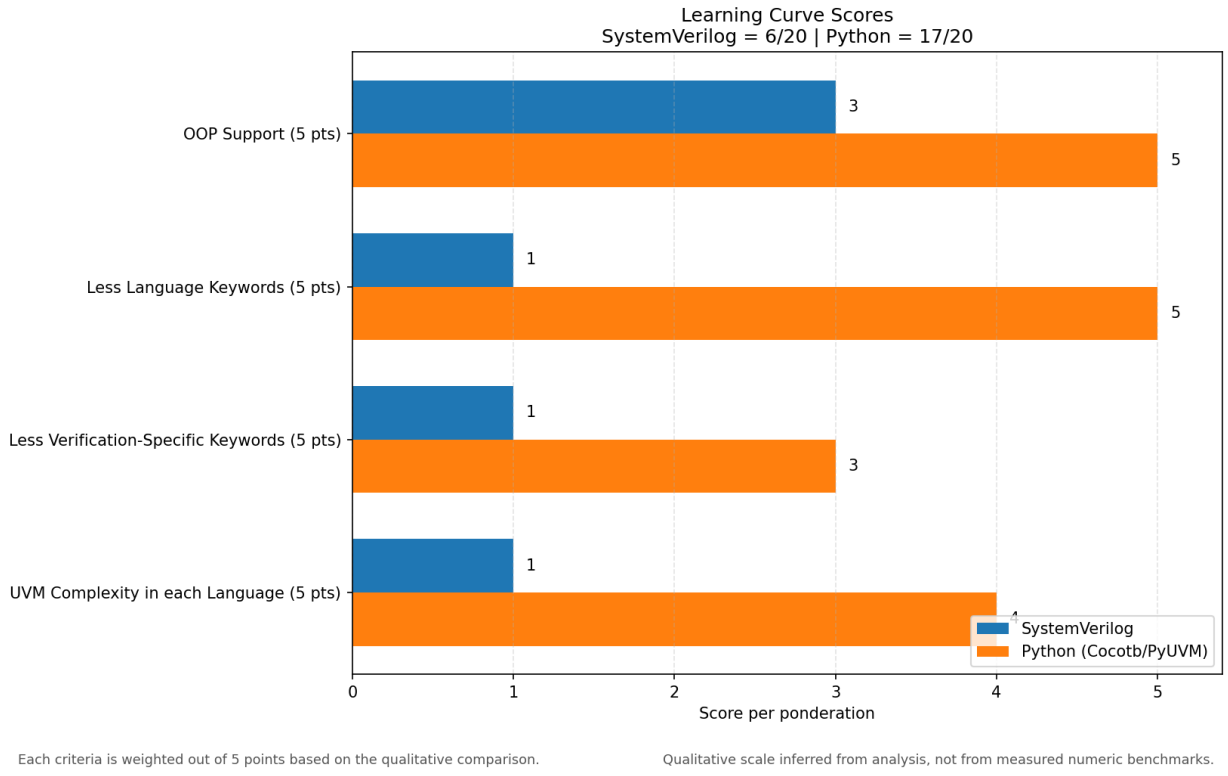


Figure 20: Learning curve results of Python and Systemverilog frameworks.

5.4.2 Code Readability Scores

Python is a more readable language than SystemVerilog. Figure 22 displays the score that each language gets. The topics considered are how simple the language syntax is for each language. The language capabilities, such as list comprehensions in Python and how easy it is to understand more complex code; the UVM-specific syntax for each language, where Python implements a simplified version of the methodology; the use of external libraries to simplify code; and the use of indentation as a tool that results in more readable code.

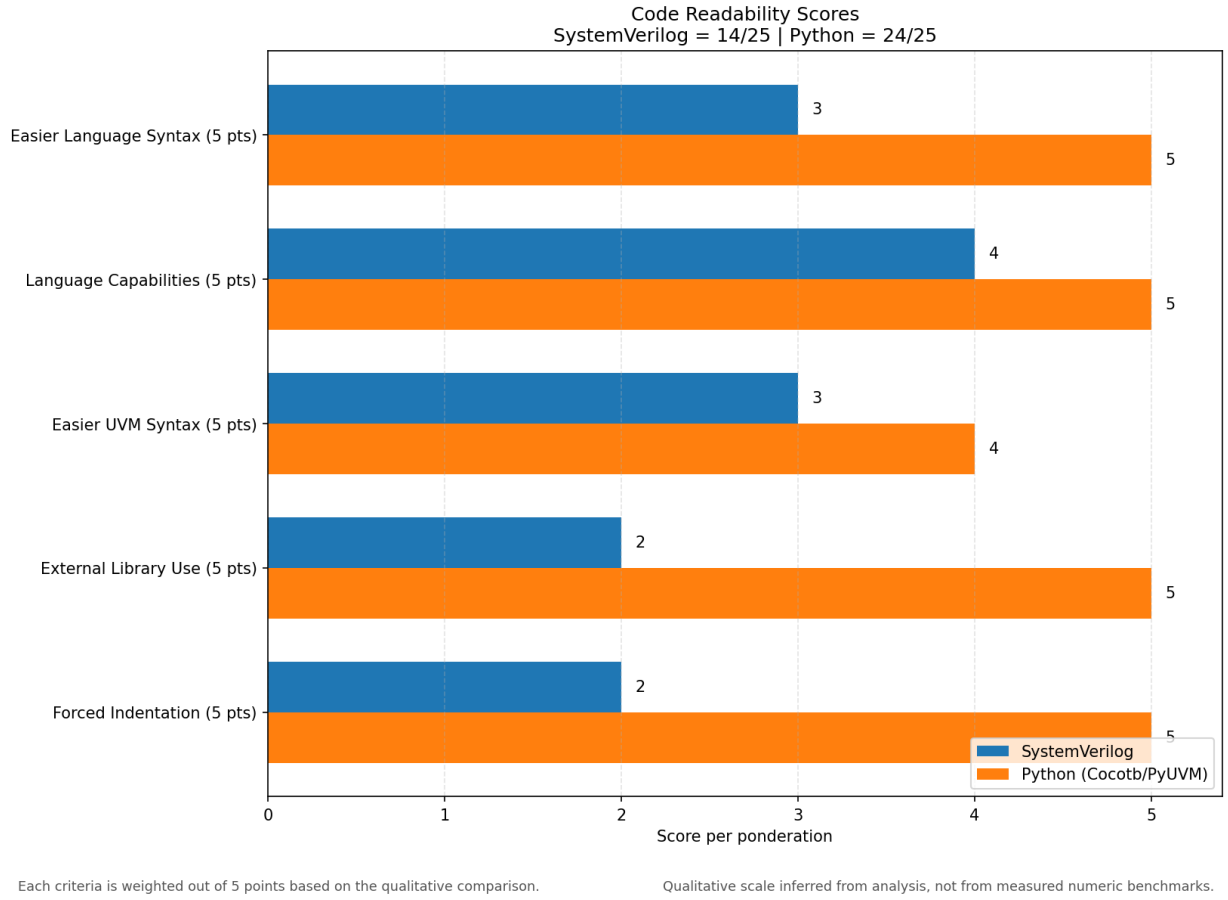


Figure 21: Code readability scores of Python and SystemVerilog frameworks.

5.4.3 Maintainability Indices

SystemVerilog is a more maintainable code when it comes to hardware verification than Python. Figure 23 shows the score of each language in this regard. The Python framework does not support certain features, such as interfaces and midparts, and it also does not natively support bit-wise operations or handling bits in a bus. The implementation of important features such as threading and processes is easier to handle syntax-wise in SystemVerilog; waiting for processes in several ways can be more complicated in Python. Regarding the UVM standard, PyUVM does not support all the phasing that SystemVerilog does, and there are some other features that, although they were not tested in this analysis, are still to be implemented in PyUVM.

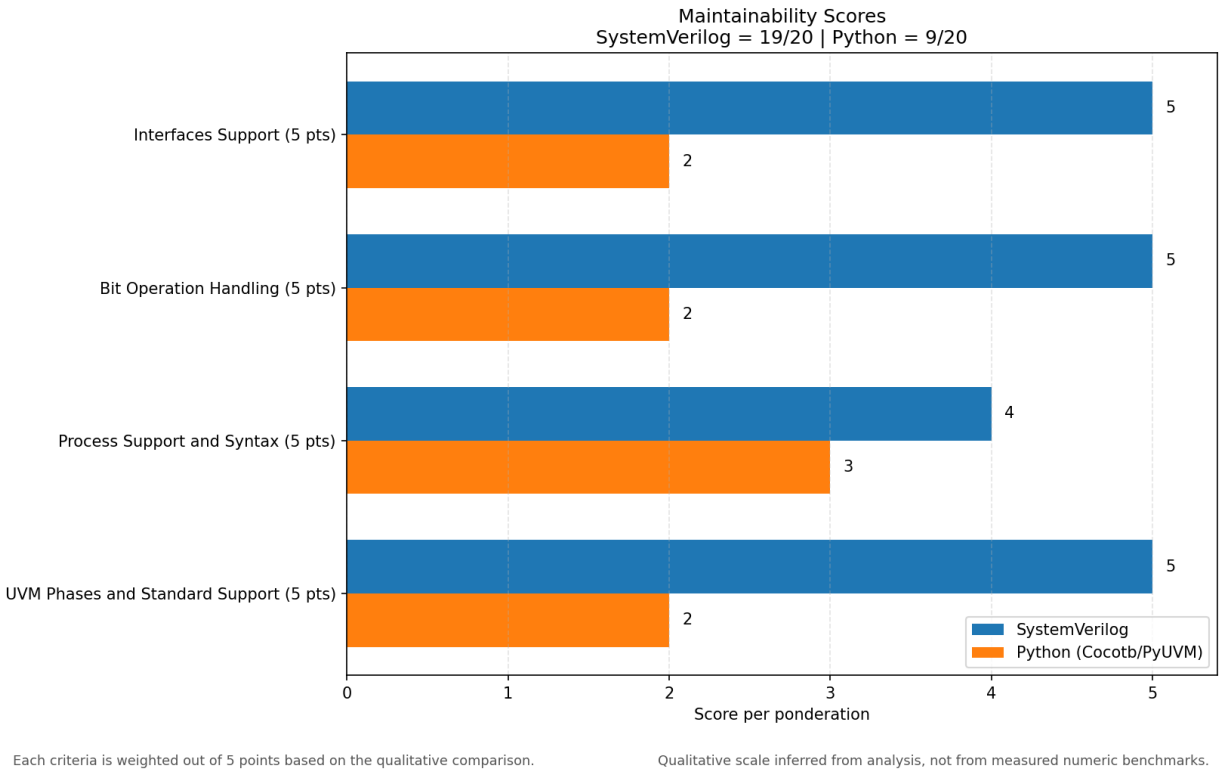


Figure 22: Code maintainability scores of Python and SystemVerilog frameworks.

5.4.4 Reusability Potential

Figure 24 shows the reusability score that each framework obtained in this analysis. Overall, it is considered that SystemVerilog is more reusable than Python because of the interface support and the full phase support that it has. However, since both languages are object-oriented, reusability is part of the design philosophy and should be relatively easy.

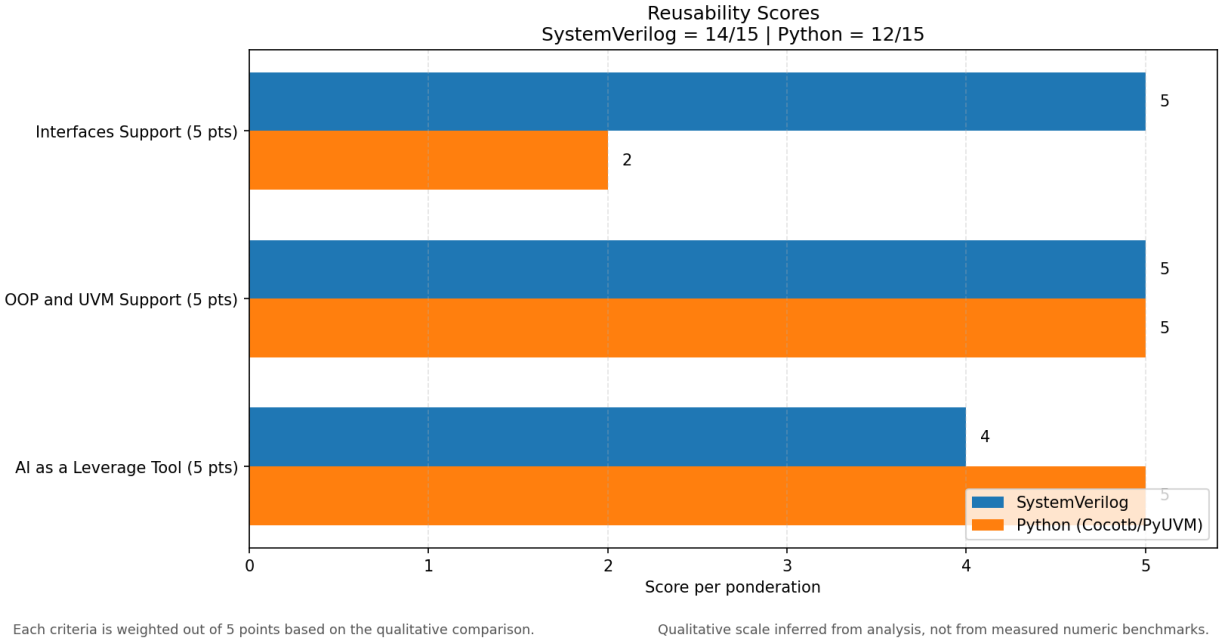


Figure 23: Code reusability scores of Python and SystemVerilog frameworks.

6. Discussion

This chapter presents an interpretation of the experimental results obtained from the comparative analysis between the UVM framework and Python's framework, examining the implications of each finding within the broader context of industrial verification practices. The analysis encompasses quantitative performance metrics such as development time, simulation efficiency, resource utilization, and coverage effectiveness, as well as qualitative assessments of usability characteristics such as learning curve for both languages, maintainability of the environment, and reusability potential. The interpretation considers the practical significance for verification teams making strategic methodology decisions in industrial applications.

6.1 Interpretation of Results

6.1.1 Total Implementation Time

One of the strongest arguments in favor of moving to the Python framework that its developers propose is the speed-up of the development process. Considering the learning process of both languages, the development of the environment itself, debugging the environment, and having a final working version, it is a faster development process than SystemVerilog. As mentioned previously, reaching a good level of proficiency in SystemVerilog and UVM can take a couple of years, and reaching an approximate level of proficiency in Python, Cocotb, and PyUVM only took a couple of months. If the learning process is not considered in this analysis, the development of the SystemVerilog environment was 25% faster, however, the SMBus environment already existed for the SystemVerilog framework and developing that structure in Python took 25-30% of the development process, so it is considered that developing, debugging and reaching a final working version on both environments take approximately the same amount of time and effort.

6.1.2 Lines of Code

The Python framework contains 8,235 fewer lines of code than the SystemVerilog framework. The biggest contribution to this smaller design is that Python requires less syntax to work as expected. SystemVerilog requires begin-end statements in every class, method, conditional statement, etc., and Python understands the content of all of these with the use of indentation. The second contributor to this is that since Python does not have interfaces, these do not have to be set up in the configuration database and extracted in other modules, the cocotb.top construct is added in the files that need DUT visibility; the implications for a lack of interfaces will be discussed further down in this document. Another contributor to this sleeker design is the lack of configuration classes, which are used in the SystemVerilog framework to pass the interfaces between classes. The last reason found is that Python contains powerful instructions, such as list comprehension, and that its components already possess some features that needed to be manually added in SystemVerilog. Overall, the code is smaller and easier to understand which can, to some degree, aid in the maintainability, readability, and reusability of this framework in comparison to its counterpart.

6.1.3 Simulation Time

The Python framework, on a single testcase simulation, takes approximately 2.59 times longer than the SystemVerilog framework. Considering that both environments are identical, this time can be attributed to Python needing to use the VPI to run a simulation, while the simulators are optimized to run SystemVerilog code. This time implies that developing and testing RTL can become a slower process overall.

If an RTL design team relies on a Python framework environment to test features, those simulations will take longer to finish. This time also needs to be considered when a feature in a testcase fails in a regression, and the developers need to simulate and debug; and finally, it can add extra development time when a verification engineer is implementing the environment itself. A testcase running 2.59 times slower in a simulation that only takes a couple of minutes, for example, in small environments; it could be discarded as a defining factor on whether to choose one or another framework; however, that time starts adding up depending on the number of testcases defined for the verification of a project. In an industrial application, a single testcase could potentially take several hours or even days to complete, which will result in this time doubling or tripling and could interfere with project deadlines.

6.1.4 Memory and CPU Utilization

The experimental result demonstrates that the Python framework consistently utilized approximately 70% more memory than the SystemVerilog implementation, with average peak memory usage of 820.71MB compared to 532.53MB, respectively. This increased memory consumption can be attributed to several factors inherent to Python's architecture, including the overhead of dynamic typic where every variable is treated as an object with associated metadata, the Python Virtual Machine's bytecode interpretation layer, and the garbage collection mechanism that delays memory deallocation. Additionally, the Cocotb framework's reliance on the VPI/GPI interface creates additional memory buffers for communication between the Python interpreter and the HDL simulator, while the loading of extensive Python libraries and the less optimized data structures in PyUVM compared to native SystemVerilog contribute to the overall memory footprint increase.

Conversely, the Python framework demonstrated significantly lower CPU parallelization efficiency, utilizing an average 104% CPU compared to SystemVerilog's 136%, representing approximately 25% less parallel processing capability. This performance limitation is primarily caused by Python's Global Interpreter Lock (GIL) [33], which restricts execution to a single thread at a time and prevents full utilization of multiple CPU cores during computation-intensive operations. The SystemVerilog framework benefits from native simulator integration that enables optimized parallel event processing, while the Python framework requires additional synchronization overhead and serialized communication through the VPI interface. These results indicate a fundamental trade-off between the two methodologies; while Python offers enhanced accessibility and reduced development complexity, it sacrifices runtime efficiency in terms of both memory utilization and parallel processing capabilities, making SystemVerilog more suitable for resource-constrained or performance-critical verification scenarios.

6.1.5 Compilation Time

The compilation time analysis revealed minimal differences between the two verification frameworks, with both requiring compilation of the Verilog and SystemVerilog RTL files comprising the LTPI DUT and main wrapper components. Despite Python's interpreted nature eliminating the need for testbench compilation, both frameworks must still compile the hardware description language files through the same simulator toolchain. The experimental results, averaged across five compilation runs on identical hardware, demonstrated that the Python framework achieved a marginal 4.24-second advantage over the SystemVerilog framework, with compilation times of 47.18 seconds and 51.42 seconds, respectively. The 8.2% difference represents a relatively insignificant performance variation, indicating that compilation time is not a determining factor in framework selection since both methodologies are constrained by the same RTL compilation requirements. It should be noted that incremental compilation techniques, where only modified files are recompiled rather than the entire design, could significantly improve these compilation times for both frameworks during iterative development cycles. This was, however, not tested in this analysis.

6.1.6 Regression Execution Time

The regression execution time analysis reveals a significant performance disparity favoring the SystemVerilog framework, with the Python framework requiring an average of 7.54 times longer execution duration for identical regression configurations. A baseline regression consisting of single runs per testcase completed in approximately 1 hour using SystemVerilog, compared to nearly 8 hours with the Python framework. This performance difference scales proportionally with simulation per testcase, and the 100 cycles per testcase regression already had a difference of approximately 33 days.

These performance characteristics have critical implications for industrial verification workflows, where RTL design candidates must undergo comprehensive regression testing to ensure functional integrity before deployment. Industrial verification environments require rapid feedback cycles to maintain development efficiency and provide timely results to design engineering teams. The observed performance difference represents a decisive factor in methodology selection: a basic acceptance test that could require 1 day in SystemVerilog would take 1 week of completion in Python, fundamentally impacting project schedule and resource allocation. Furthermore, coverage collection needs extensive regression cycles that amplify these performance differences. Extrapolating to industrial-scale scenarios with 1000 simulation runs per testcase, the SystemVerilog framework would require approximately 52 days compared to Python's 384 days. Such extended verification cycles would render the Python framework impractical for industrial deployment, likely resulting in its elimination during initial project methodology selection phases.

6.1.7 Functional Coverage

Functional coverage represents a pivotal verification metric for assessing whether the environment adequately exercises the design's intended functionality and for quantifying verification completeness. The ability to efficiently implement covergroups and coverpoints constitutes an essential capability for verification engineers to monitor test effectiveness and identify coverage gaps systematically. The experimental results demonstrate that while the SystemVerilog framework has improvement areas for functional coverage enhancement, the Python framework was unable to collect this coverage data due to technical limitations with the PyVSC library integration.

This absence of functional coverage capability in the Python verification framework represents a significant limitation for industrial verification applications. The inability to collect data and analyze functional coverage metrics inherently constrains the verification engineer's ability to assess test completeness, identify verification gaps, and demonstrate adequate design exercise according to industry standards. This capability gap suggests that the SystemVerilog framework provides more verification infrastructure aligned with established industry practices and methodologies, representing a critical consideration for verification teams requiring robust coverage analysis capabilities.

6.1.8 Code, Branch, and Toggle Coverage

The code coverage analysis demonstrates comparable performance between both verification frameworks, with SystemVerilog achieving a marginal 0.79% advantage on average. The similarity indicates that both verification environments possess equivalent capabilities for exercising RTL code and demonstrates that their respective randomization mechanisms provide comparable stimulus generation effectiveness. However, branch and toggle coverage metrics reveal more significant differences between the frameworks.

The Python framework achieved 6.76% more branch coverage on average compared to the SystemVerilog framework. This performance advantage can be attributed to several factors related to execution timing. Python's slower execution may create different race condition scenarios that exercise alternative conditional code paths not typically accessed during faster SystemVerilog execution. Additionally, Cocotb's asynchronous Coroutine execution model may access different decision branches compared to SystemVerilog's native execution patterns, while the VPI interface layer could potentially trigger additional conditional paths within the simulator that are not activated through direct SystemVerilog integration.

Conversely, the SystemVerilog framework achieved 7.34% higher toggle coverage on average than the Python implementation. This advantage can be attributed to SystemVerilog's better timing precision and direct simulator integration, which enables more accurate timing control and consequently generates more signal transitions. The VPI interface limitations inherent in the Python framework may result in missed internal signal transitions that are readily captured through SystemVerilog's native simulator access.

Furthermore, during Python framework development, it was observed that Python does not automatically discover the complete design hierarchy, requiring explicit configuration commands to locate RTL modules, particularly those instantiated with generate loops. This limitation may prevent Python from monitoring all internal signal toggle activities throughout the RTL design, thereby reducing overall toggle coverage metrics.

6.1.9 Learning Curve

Learning Python was deemed to be significantly easier to understand and start working on than SystemVerilog. Python is a software development language, whereas SystemVerilog is a combination of hardware description language and a software development language. The common ground that they both share is that they are object-oriented programming languages, meaning the basics of OOP apply to both. However, in Python, OOP concepts apply to everything from variables to entire libraries and provide more OOP features than SystemVerilog.

In the context of hardware verification, SystemVerilog provides more features than Python, which translates to more concepts and definitions that a user needs to learn to become proficient. This means that learning SystemVerilog naturally takes longer to fully exploit its features. Despite this, to develop an industrial-level verification environment, an engineer still needs to learn the UVM concepts, as well as RTL development, to be able to create appropriate testcases as well as debug the DUT if needed. This means that the learning curve applies only to the language itself and not the full verification workflow that a verification engineer will face.

6.1.10 Code Readability Scores

Python is more readable. The language itself contains fewer rules than SystemVerilog. Branching statements and object definitions are cleaner syntax-wise, which results in less code in general. Python also provides powerful features such as list comprehension, which allows the developer to create cleaner algorithms. The PyUVM library, which is a direct equivalent to the UVM library, is a cleaner implementation of the methodology syntax-wise, and it uses existing Python libraries, such as logging, to provide some features that the standard requires. The logging library also allows a verification engineer to tailor the content of the messages sent to the log file. Another feature that is embedded into Python is the use of code tabs in conditional statements and functions, which forces the developer to properly indent their code for it to run, which can result in a better reading experience overall when compared to a SystemVerilog code that is not properly indented.

6.1.11 Maintainability Indices

SystemVerilog is more maintainable. Despite the advantages that Python provides. SystemVerilog is better designed for hardware verification tasks. Python with Cocotb and PyUVM do not support components such as interfaces, which allows the creation of fully reusable environments with generic interface names.

Python also does not natively support bit-wise operations or handling bits in an array, forcing the user to rely on shifting data to get a specific bit out of an attribute, which can result in some algorithms being unnecessarily complicated and more prone to error and debugging. Another feature that is pivotal in hardware verification is the use of threading, having multiple checkers running simultaneously, for example. Cocotb supports this with commands like start and start_soon; however, implementing multiple processes and supporting features like waiting for processes, waiting for one, disabling a process, etc., is easier to implement in SystemVerilog, which translates into less complex algorithms compared to a Python implementation.

Another factor that was taken into consideration is that currently, PyUVM does not support all the phases that the UVM standard defines. PyUVM only supports the run phase, whereas in SystemVerilog, the run phase can be sub-categorized into 12 phases. This means that in SystemVerilog, modifying phases like the run phase or the main phase becomes more convenient since these sections can be separated. This is also the case when a child class needs to override a parent implementation; instead of overriding the entire run phase of the base test, a child test could potentially only override one of the phases contained inside the run phase.

6.1.12 Reusability Potential

Reusability can be achieved in SystemVerilog more easily. Primarily because of the interface support. In SystemVerilog, an environment can be fully agnostic to the design it is testing, with internal components writing and reading from an interface, which is in turn connected to the DUT in a wrapper file. In Python, modules like drivers and monitors need to get the signals they are interacting with directly from the DUT, which can result in an engineer needing to modify entire files or environments once it needs to be leveraged into another project. Other than this, UVM itself and OOP in general are designed with the idea of reusing existing code, so leveraging components in both languages for new projects is relatively easy.

6.2 Literature vs Results

Comparative results regarding Cocotb/PyUVM are limited, and they appear to favor the use of the Python framework by presenting better coverage results. However, the current consensus is that the Python implementation is slower, which represents an area for improvement that the library design team could address to make this framework a viable option when choosing methodologies. Regarding the advantages proposed by Cocotb developers, it is true that starting an environment from scratch and obtaining quick results is an advantage that the library possesses, with Python being a fast-learning language that can be more than sufficient for small RTL models. While not thoroughly tested, several existing Python libraries were utilized for the development of the PyUVM library and this analysis, confirming the advantage that this ecosystem has in terms of reusing existing libraries, although this point can be further explored in future projects.

A proposal that may not hold in industrial development is that it accelerates development time. While an environment can be built faster in Python, simulation time can quickly consume that advantage, since a verification engineer spends more time running simulations and regressions than developing the environment. Another disadvantage that Python has compared to SystemVerilog in this regard is the consolidation that SystemVerilog already has in this industry. Verification teams have had 10 years to consolidate their IP catalog and have become quite extensive, and in Python, this thorough level of consolidation has not happened yet. Verification teams will need to port their existing code into Python over the coming years to match what is already built in SystemVerilog. This translation into Python, however, could be rapidly accelerated with the use of artificial intelligence and interest in the industry to adopt this new framework.

Additionally, the fact that Python is an interpreted language does not necessarily save compilation time, since the design may need to be recompiled several times during development time, and tools already exist to accelerate this process in both methodologies, as previously discussed. Finally, while Python is used by more people in the technology industry compared to SystemVerilog, which could enable more people to switch into verification roles, the industrial requirement for pre-silicon verification engineers usually involves knowledge in UVM and RTL development, so an applicant needs to develop said knowledge regardless of the language used for verification.

6.3 Practical Implications

Based on the experimental results, PyUVM/Cocotb as a verification solution is not yet ready for large-scale industrial applications, mainly due to its significant simulation time disadvantage compared to a SystemVerilog solution. However, this does not preclude its adoption in specific use cases where its advantages can be leveraged effectively. The framework could serve as a feasible solution for smaller RTL designs where the simulation time differential is less pronounced, or in subsystem and unit-level verification scenarios where extensive, resource-intensive regressions are not required. In these contexts, the rapid development capabilities and accessibility of Python could outweigh the performance limitations.

The Cocotb/PyUVM framework demonstrates comparable verification power to SystemVerilog in terms of randomization capabilities, BFM development for testing protocol interfaces, and complex systems. This strength positions it as a valuable tool for verification engineers who need to rapidly prototype verification components or create focused models to test specific design functions while a comprehensive system-level environment is under development. This rapid prototyping can provide immediate feedback to design engineers during early development phases. However, for established verification teams with an existing IP catalog and mature SystemVerilog-like components, the incentive to migrate may be limited, particularly since their existing models can be seamlessly integrated into full-system environments without the performance penalties observed in Python implementations.

6. Discussion

It is important to note that Cocotb's design philosophy extends beyond UVM-like verification environments, offering potential applications in specialized validation domains that could provide unique industrial value. These include capabilities such as integrating board-level BFMs directly into simulation models or better supporting mixed-signal testing scenarios. While these applications represent promising areas for Cocotb adoption and could differentiate it from traditional SystemVerilog approaches, they fall outside the scope of this comparative analysis, which focuses specifically on functional verification methodologies for digital systems.

7. Conclusions

This comparative analysis has revealed significant insights into the trade-offs between traditional and emerging verification approaches in an industrial application. The most significant finding is that both methodologies render similar results in terms of code coverage and regression collection, but SystemVerilog has a significant advantage in performance, running simulations 7.54 times faster, as well as providing functional coverage without needing extra libraries, while consuming less memory and providing better parallelization capabilities.

This research contributes the first comprehensive empirical comparison between SystemVerilog and Cocotb/PyUVM methodologies in an industrial verification scenario, bridging the gap between academic claims and practical implementation realities. The findings challenge the assumption that Python-based verification can serve as a direct replacement for SystemVerilog, instead suggesting specific use cases where each methodology excels. For the verification engineering community, this research guides methodology selection, indicating that Cocotb/PyUVM solutions may be suitable for small-scale designs, rapid prototyping, and educational purposes, while SystemVerilog remains superior for performance-critical and large-scale verification projects.

The study acknowledges the limitations that constrain the generalization of the findings, including the focus on a single protocol, potential bias coming from an established expertise in SystemVerilog, and the inability to evaluate Python's functional coverage capabilities. Future research should extend this comparison to multiple design types, researching ways to optimize Cocotb/PyUVM's simulation time; extending PyUVM capabilities to fully support the UVM standard; and diving into other verification approaches using Cocotb outside of a UVM environment.

As the semiconductor industry faces increasing pressure to accelerate design development while designs keep growing, researching alternative and new verification solutions becomes essential for strategic industry adoption. This analysis contributes to the broader evolution of verification methodologies by establishing benchmarks that will guide future tool development and help shape the next generation of hardware verification practices in an increasingly complex industry landscape.

7. Conclusions

Appendix A: Testplan

Test ID	Name	Objective	Expected Result
00	Link Training	Wait for LTPI to reach operational state.	LTPI reaches operational state in less than 20ms.
01	Link Detect Error	Generate CRC or frame comma error in the Link Detect stage on either side of the link.	LTPI goes into Link Lost Error and performs the retraining process.
02	Link Speed Error	Generate CRC or frame comma error in the Link Speed stage on either side of the link.	LTPI goes into Link Lost Error and performs the retraining process.
03	Link Advertise Error	Generate CRC, frame comma error, or timeout error in the Link Advertise stage on either side of the link.	LTPI goes into Link Lost Error and performs the retraining process.
04	Link Configure Error	Generate a CRC or timeout error in the Link Configure stage on either side of the link.	LTPI goes into Link Lost Error and performs the retraining process.
05	Link Operational Error	Generate CRC, frame comma error, or frame lost error in the Link Operational stage on either side of the link.	LTPI goes into Link Lost Error and performs the retraining process.
06	Break PLL Error	Generate an error in the PLL lock signal during the link training process.	LTPI should not enter the error state and wait for the PLL lock to keep the training process.
10	NL GPIO Basic	Randomly selects between 10 and 100 Normal Latency GPIO ports on each side of the LTPI link and toggles each selected port once.	Selected GPIO ports should reach the other side of the link appropriately before a time threshold.
11	NL GPIO Mix	Randomly selects between 10 and 100 Normal Latency GPIO ports on each side of the LTPI link and toggles each selected port between 2 and 10 times.	Selected GPIO ports should reach the other side of the link appropriately before a time threshold.
12	LL GPIO Basic	Randomly selects between 1 and 16 Low Latency GPIO ports on each side of the LTPI link and toggles each selected port once.	Selected GPIO ports should reach the other side of the link appropriately before a time threshold.
13	LL GPIO Mix	Randomly selects between 1 and 16 Low Latency GPIO ports on each side of the LTPI link and toggles each selected port between 5 and 20 times.	Selected GPIO ports should reach the other side of the link appropriately before a time threshold.
14	UART Basic	The test will generate a random byte for each side of the LTPI link, randomize the baud rate for	Both UART transactions should reach the other side of the link at the expected baud rate.

		both, and transmit each byte over its respective UART interface.	
15	UART Advanced	The test will generate between 2 and 20 random bytes for each side of the LTPI link, randomize the baud rate for both sides, and transmit the bytes over its respective UART interface.	All UART transactions should reach the other side of the link at the expected baud rate.
16	UART Speed	The test will generate a random byte for each side of the LTPI link, set the baud rate at 115,200 Bd for both sides, and transmit each byte over its respective UART interface.	Both UART transactions should reach the other side of the link at the expected baud rate.
17	SMBus Basic	The test will randomize between an SMBus Send Byte or Receive Byte command on all SMBus ports, choose a random speed between 300 and 450 kHz, and send the commands over the LTPI interface.	The commands should be properly acknowledged by the other side of the link, and communication must work.
18	SMBus Advanced	The test will randomize between an SMBus Write Byte, Write Word, Read Byte, or Read Word command on all SMBus ports, choose a random speed between 300 and 450 kHz, and send the commands over the LTPI interface.	The commands should be properly acknowledged by the other side of the link, and communication must work.
19	SMBus Edge	The test will randomize between an SMBus Send Byte, Receive Byte, Write Byte, Write Word, Read Byte, or Read Word command on all SMBus ports, set the speed to 450 kHz, and send the commands over the LTPI interface.	The commands should be properly acknowledged by the other side of the link, and communication must work.
20	AVMM Controller	The test will choose a random LTPI Controller register and randomize between a write or read command and send it over the LTPI interface.	The selected register should be properly read or written, and AVMM communication must be met.
21	AVMM Target	The test will choose a random LTPI Target register and randomize between a write or read command and send it over the LTPI interface.	The selected register should be properly read or written, and AVMM communication must be met.

Appendix B: Repositories

LTPI Design

https://github.com/opencomputeproject/HWMgmt-Module-DCSCM-LTPI.git

branch: main

commit: 22710f8a30081068b90d6dbdc03b7130acf7c093

LTPI SystemVerilog Verification Environment

https://github.com/MarcoRamirezZ/ltpi-verifenv-sv.git

branch: main

commit: d8bf2ac31ddce45f8db563cd824aa4b8e3136d35

LTPI Python Verification Environment

https://github.com/MarcoRamirezZ/ltpi-verifenv-python.git

branch main

commit: 9058e51f27c9d630c3ca74b3693c34d5e25ec020

Appendix B: Repositories

Appendix C: Testcase Waveforms and Coverage Results for SystemVerilog Framework

In this section, some results of testcase simulations and full coverage reports will be presented for the SystemVerilog framework. Figures 25 to 28 showcase the simulation of Link Training, CRC error injection during the Link Detect Stage, a UART simulation testcase, and an example of the memory map access, respectively. Figures 29 to 34 show the regression results for 1 cycle per testcase, 4 cycles, 10, 20, 50, and 100, respectively.

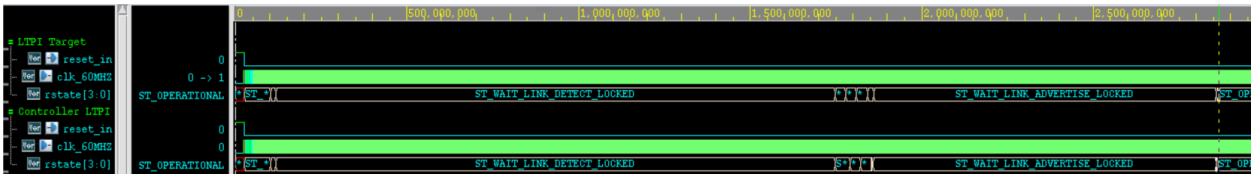


Figure 24: Link Training simulation - SV framework.

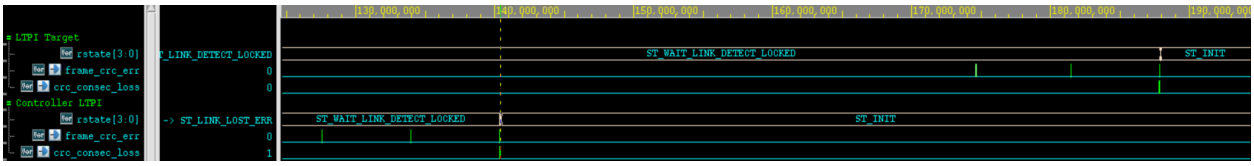


Figure 25: Link Detect error simulation - SV framework.

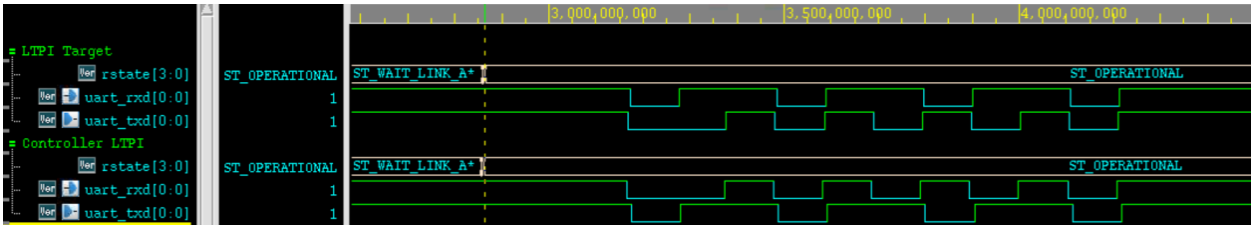


Figure 26: UART Basic simulation - SV framework.

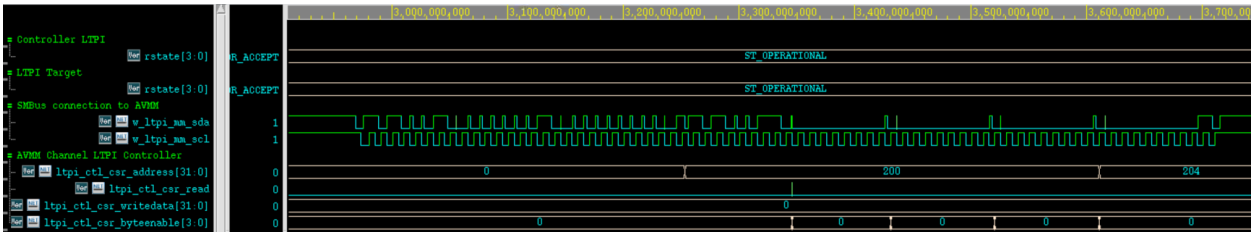
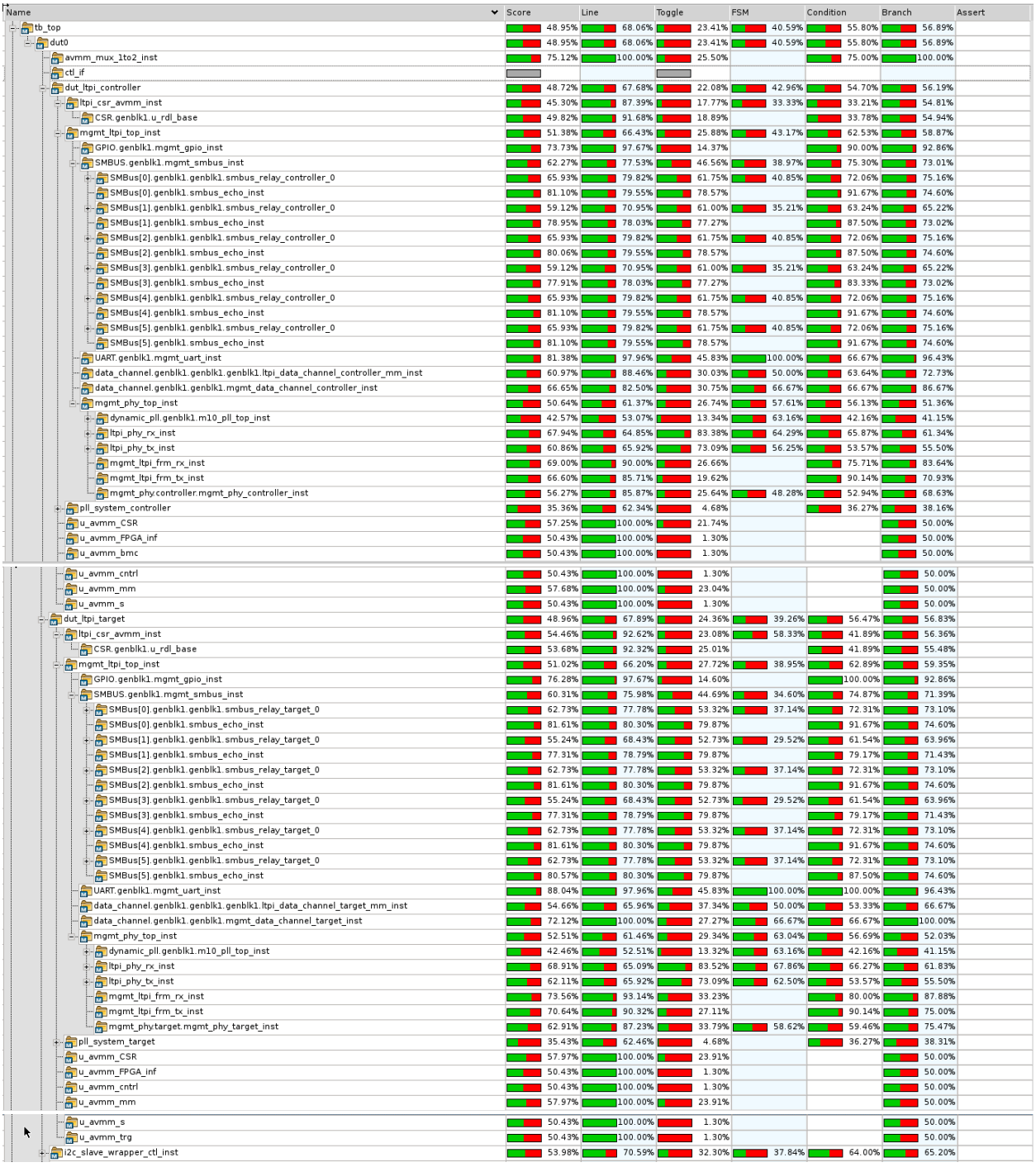


Figure 27: AVMM Controller simulation - SV framework.

Appendix B: Repositories

Regression Result 1 cycle per test

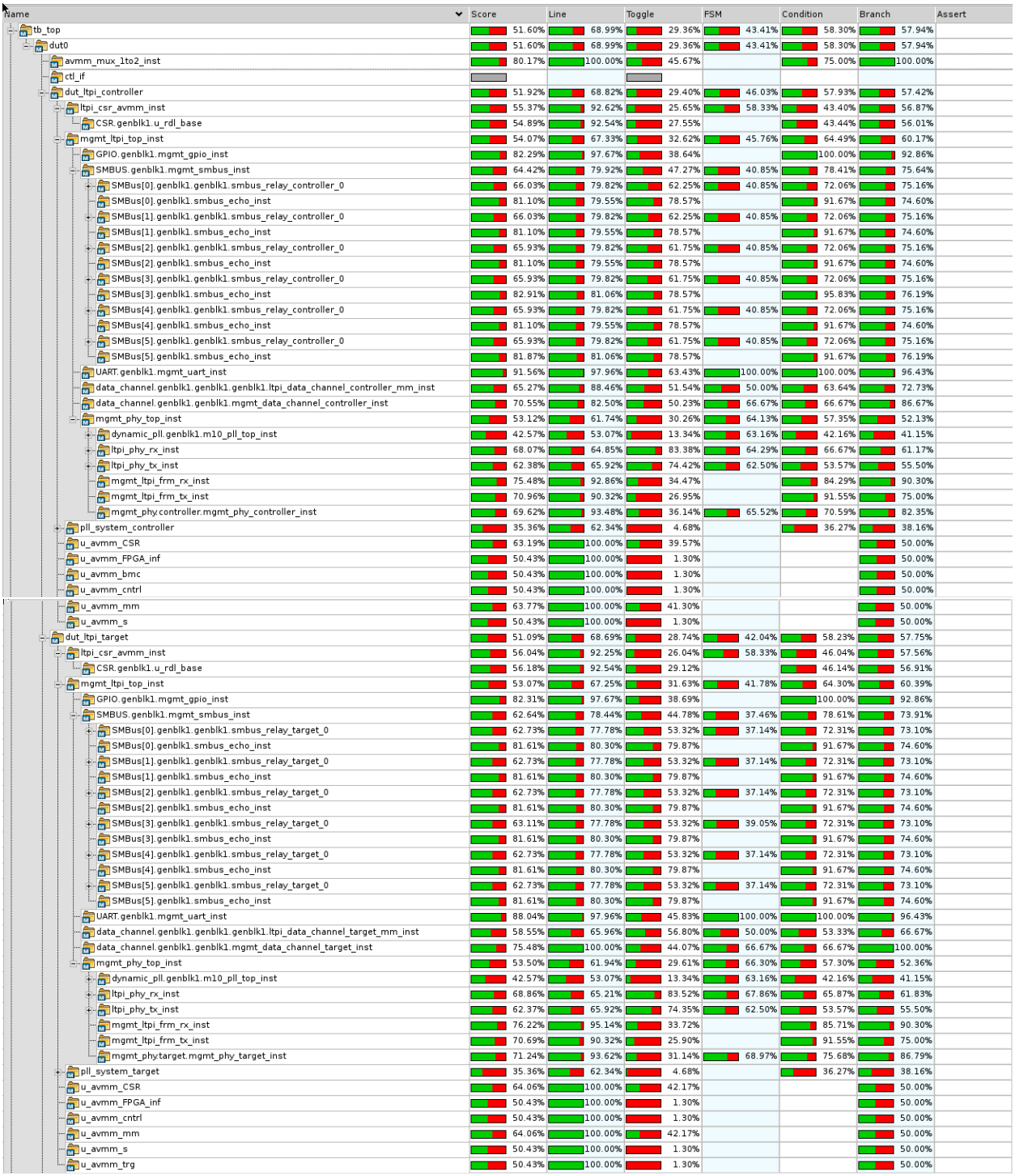


Name	Score	Line	Toggle	FSM	Condition	Branch	Assert
tb_top	48.95%	68.06%	23.41%	40.59%	55.80%	56.89%	
dut0	48.95%	68.06%	23.41%	40.59%	55.80%	56.89%	
avmm_mux_1to2_inst	75.12%	100.00%	25.50%		75.00%	100.00%	
ctl_if							
dut_ltpi_controller	48.72%	67.68%	22.08%	42.96%	54.70%	56.19%	
ltpi_csr_avmm_inst	45.30%	87.39%	17.77%	33.33%	33.21%	54.81%	
CSR.genblk1.u_rdl_base	49.82%	91.68%	18.89%		33.78%	54.94%	
mgmt_ltpi_top_inst	51.38%	66.43%	25.88%	43.17%	62.53%	58.87%	
GPIO.genblk1.mgmt_gpio_inst	73.73%	97.67%	14.37%		90.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	62.27%	77.53%	46.56%	38.97%	75.30%	73.01%	
SMBus[0].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[0].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_controller_0	59.12%	70.95%	61.00%	35.21%	63.24%	65.22%	
SMBus[1].genblk1.smbus_echo_inst	78.95%	78.03%	77.27%		87.50%	73.02%	
SMBus[2].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[2].genblk1.smbus_echo_inst	80.06%	79.55%	78.57%		87.50%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_controller_0	59.12%	70.95%	61.00%	35.21%	63.24%	65.22%	
SMBus[3].genblk1.smbus_echo_inst	77.91%	78.03%	77.27%		83.33%	73.02%	
SMBus[4].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[4].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[5].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
UART.genblk1.mgmt_uart_inst	81.38%	97.96%	45.83%	100.00%	66.67%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_controller_mm_inst	60.97%	88.46%	30.03%	50.00%	63.64%	72.73%	
data_channel.genblk1.genblk1.mgmt_data_channel_controller_inst	66.65%	82.50%	30.75%	66.67%	66.67%	86.67%	
mgmt_phy_top_inst	50.64%	61.37%	26.74%	57.61%	56.13%	51.36%	
dynamic_pll.genblk1.m10_pll_top_inst	42.57%	53.07%	13.34%	63.16%	42.16%	41.15%	
ltpi_phy_rx_inst	67.94%	64.85%	83.38%	64.29%	65.87%	61.34%	
ltpi_phy_tx_inst	60.86%	65.92%	73.09%	56.25%	53.57%	55.50%	
mgmt_ltpi_frm_rx_inst	69.00%	90.00%	26.66%		75.71%	83.64%	
mgmt_ltpi_frm_tx_inst	66.60%	85.71%	19.62%		90.14%	70.93%	
mgmt_phy.controller.mgmt_phy_controller_inst	56.27%	85.87%	25.64%	48.28%	52.94%	68.63%	
pll_system_controller	35.36%	62.34%	4.68%		36.27%	38.16%	
u_avmm_CSR	57.25%	100.00%	21.74%			50.00%	
u_avmm_FPGA_inf	50.43%	100.00%	1.30%			50.00%	
u_avmm_bmc	50.43%	100.00%	1.30%			50.00%	
u_avmm_cntrl	50.43%	100.00%	1.30%			50.00%	
u_avmm_mm	57.68%	100.00%	23.04%			50.00%	
u_avmm_s	50.43%	100.00%	1.30%			50.00%	
dut_ltpi_target	48.96%	67.89%	24.36%	39.26%	56.47%	56.83%	
ltpi_csr_avmm_inst	54.46%	92.62%	23.08%	58.33%	41.89%	56.36%	
CSR.genblk1.u_rdl_base	53.68%	92.32%	25.01%		41.89%	55.48%	
mgmt_ltpi_top_inst	51.02%	66.20%	27.72%	38.95%	62.89%	59.35%	
GPIO.genblk1.mgmt_gpio_inst	76.28%	97.67%	14.60%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	60.31%	75.98%	44.69%	34.60%	74.87%	71.39%	
SMBus[0].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[0].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_target_0	55.24%	68.43%	52.73%	29.52%	61.54%	63.96%	
SMBus[1].genblk1.smbus_echo_inst	77.31%	78.79%	79.87%		79.17%	71.43%	
SMBus[2].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[2].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_target_0	55.24%	68.43%	52.73%	29.52%	61.54%	63.96%	
SMBus[3].genblk1.smbus_echo_inst	77.31%	78.79%	79.87%		79.17%	71.43%	
SMBus[4].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[4].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[5].genblk1.smbus_echo_inst	80.57%	80.30%	79.87%		87.50%	74.60%	
UART.genblk1.mgmt_uart_inst	88.04%	97.96%	45.83%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_target_mm_inst	54.66%	65.96%	37.34%	50.00%	53.33%	66.67%	
data_channel.genblk1.genblk1.mgmt_data_channel_target_inst	72.12%	100.00%	27.27%	66.67%	66.67%	100.00%	
mgmt_phy_top_inst	52.51%	61.46%	29.34%	63.04%	56.69%	52.03%	
dynamic_pll.genblk1.m10_pll_top_inst	42.46%	52.51%	13.32%	63.16%	42.16%	41.15%	
ltpi_phy_rx_inst	68.91%	65.09%	83.52%	67.86%	66.27%	61.83%	
ltpi_phy_tx_inst	62.11%	65.92%	73.09%	62.50%	53.57%	55.50%	
mgmt_ltpi_frm_rx_inst	73.56%	93.14%	33.23%		80.00%	87.88%	
mgmt_ltpi_frm_tx_inst	70.64%	90.32%	27.11%		90.14%	75.00%	
mgmt_phy.target.mgmt_phy_target_inst	62.91%	87.23%	33.79%	58.62%	59.46%	75.47%	
pll_system_target	35.43%	62.46%	4.68%		36.27%	38.31%	
u_avmm_CSR	57.97%	100.00%	23.91%			50.00%	
u_avmm_FPGA_inf	50.43%	100.00%	1.30%			50.00%	
u_avmm_cntrl	50.43%	100.00%	1.30%			50.00%	
u_avmm_mm	57.97%	100.00%	23.91%			50.00%	
u_avmm_s	50.43%	100.00%	1.30%			50.00%	
u_avmm_trg	50.43%	100.00%	1.30%			50.00%	
i2c_slave_wrapper_ctl_inst	53.98%	70.59%	32.30%	37.84%	64.00%	65.20%	

Figure 28: Regression result 1 cycle - SV framework.

Appendix B: Repositories

Regression Result 4 cycle per test

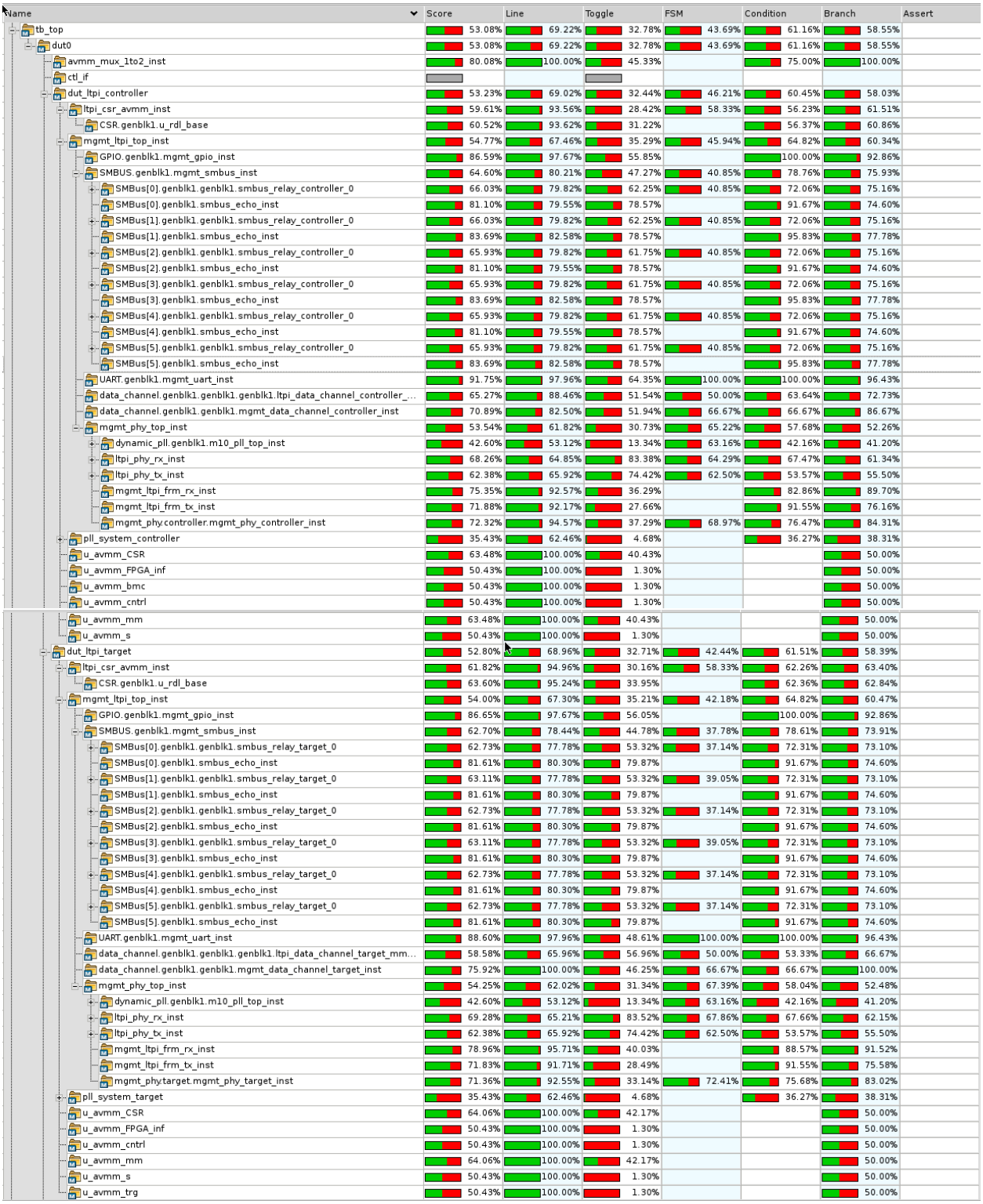


Name	Score	Line	Toggle	FSM	Condition	Branch	Assert
tb_top	51.60%	68.99%	29.36%	43.41%	58.30%	57.94%	
dut0	51.60%	68.99%	29.36%	43.41%	58.30%	57.94%	
avmm_mux_1to2_inst	80.17%	100.00%	45.67%		75.00%	100.00%	
ctl_if							
dut_ltpi_controller	51.92%	68.82%	29.40%	46.03%	57.93%	57.42%	
ltpi_csr_avmm_inst	55.37%	92.62%	25.65%	58.33%	43.40%	56.87%	
CSR.genblk1.u_rdl_base	54.89%	92.54%	27.55%		43.44%	56.01%	
mgmt_ltpi_top_inst	54.07%	67.33%	32.62%	45.76%	64.49%	60.17%	
GPIO.genblk1.mgmt_gpio_inst	82.29%	97.67%	38.64%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	64.42%	79.92%	47.27%	40.85%	78.41%	75.64%	
SMBus[0].genblk1.genblk1.smbus_relay_controller_0	66.03%	79.82%	62.25%	40.85%	72.06%	75.16%	
SMBus[0].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_controller_0	66.03%	79.82%	62.25%	40.85%	72.06%	75.16%	
SMBus[1].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[2].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[2].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[3].genblk1.smbus_echo_inst	82.91%	81.06%	78.57%		95.83%	76.19%	
SMBus[4].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[4].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[5].genblk1.smbus_echo_inst	81.87%	81.06%	78.57%		91.67%	76.19%	
UART.genblk1.mgmt_uart_inst	91.56%	97.96%	63.43%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_controller_mm_inst	65.27%	88.46%	51.54%	50.00%	63.64%	72.73%	
data_channel.genblk1.genblk1.mgmt_data_channel_controller_inst	70.55%	82.50%	50.23%	66.67%	66.67%	86.67%	
mgmt_phy_top_inst	53.12%	61.74%	30.26%	64.13%	57.35%	52.13%	
dynamic_pll.genblk1.m10_pll_top_inst	42.57%	53.07%	13.34%	63.16%	42.16%	41.15%	
ltpi_phy_rx_inst	68.07%	64.85%	83.38%	64.29%	66.67%	61.17%	
ltpi_phy_tx_inst	62.38%	65.92%	74.42%	62.50%	53.57%	55.50%	
mgmt_ltpi_frm_rx_inst	75.48%	92.86%	34.47%		84.29%	90.30%	
mgmt_ltpi_frm_tx_inst	70.96%	90.32%	26.95%		91.55%	75.00%	
mgmt_phy_controller.mgmt_phy_controller_inst	69.62%	93.48%	36.14%	65.52%	70.59%	82.35%	
pll_system_controller	35.36%	62.34%	4.68%		36.27%	38.16%	
u_avmm_CSR	63.19%	100.00%	39.57%			50.00%	
u_avmm_FPGA_inf	50.43%	100.00%	1.30%			50.00%	
u_avmm_bmc	50.43%	100.00%	1.30%			50.00%	
u_avmm_cntrl	50.43%	100.00%	1.30%			50.00%	
u_avmm_mm	63.77%	100.00%	41.30%			50.00%	
u_avmm_s	50.43%	100.00%	1.30%			50.00%	
dut_ltpi_target	51.09%	68.69%	28.74%	42.04%	58.23%	57.75%	
ltpi_csr_avmm_inst	56.04%	92.25%	26.04%	58.33%	46.04%	57.56%	
CSR.genblk1.u_rdl_base	56.18%	92.54%	29.12%		46.14%	56.91%	
mgmt_ltpi_top_inst	53.07%	67.25%	31.63%	41.78%	64.30%	60.39%	
GPIO.genblk1.mgmt_gpio_inst	82.31%	97.67%	38.69%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	62.64%	78.44%	44.78%	37.46%	78.61%	73.91%	
SMBus[0].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[0].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[1].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[2].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[2].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[3].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[4].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[4].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[5].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
UART.genblk1.mgmt_uart_inst	88.04%	97.96%	45.83%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_target_mm_inst	58.55%	65.96%	56.80%	50.00%	53.33%	66.67%	
data_channel.genblk1.genblk1.mgmt_data_channel_target_inst	75.48%	100.00%	44.07%	66.67%	66.67%	100.00%	
mgmt_phy_top_inst	53.50%	61.94%	29.61%	66.30%	57.30%	52.36%	
dynamic_pll.genblk1.m10_pll_top_inst	42.57%	53.07%	13.34%	63.16%	42.16%	41.15%	
ltpi_phy_rx_inst	68.86%	65.21%	83.52%	67.86%	65.87%	61.83%	
ltpi_phy_tx_inst	62.37%	65.92%	74.35%	62.50%	53.57%	55.50%	
mgmt_ltpi_frm_rx_inst	76.22%	95.14%	33.72%		85.71%	90.30%	
mgmt_ltpi_frm_tx_inst	70.69%	90.32%	25.90%		91.55%	75.00%	
mgmt_phy_target.mgmt_phy_target_inst	71.24%	93.62%	31.14%	68.97%	75.68%	86.79%	
pll_system_target	35.36%	62.34%	4.68%		36.27%	38.16%	
u_avmm_CSR	64.06%	100.00%	42.17%			50.00%	
u_avmm_FPGA_inf	50.43%	100.00%	1.30%			50.00%	
u_avmm_cntrl	50.43%	100.00%	1.30%			50.00%	
u_avmm_mm	64.06%	100.00%	42.17%			50.00%	
u_avmm_s	50.43%	100.00%	1.30%			50.00%	
u_avmm_trg	50.43%	100.00%	1.30%			50.00%	

Figure 29: Regression result 4 cycles - SV framework.

Appendix B: Repositories

Regression Result 10 cycle per test

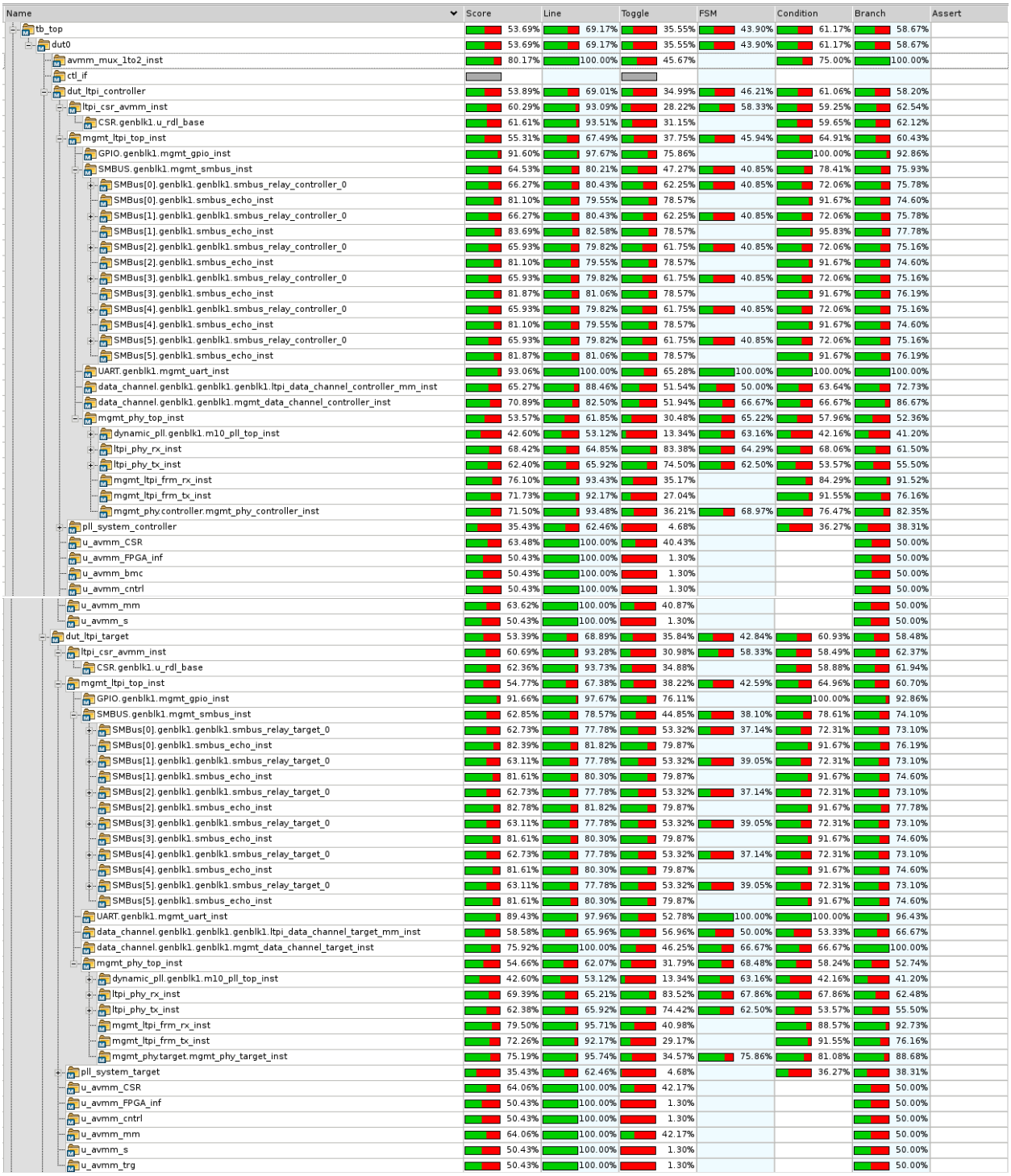


Name	Score	Line	Toggle	FSM	Condition	Branch	Assert
tb_top	53.08%	69.22%	32.78%	43.69%	61.16%	58.55%	
dut0	53.08%	69.22%	32.78%	43.69%	61.16%	58.55%	
avmm_mux_1to2_inst	80.08%	100.00%	45.33%		75.00%	100.00%	
ctl_if							
dut_ltpi_controller	53.23%	69.02%	32.44%	46.21%	60.45%	58.03%	
ltpi_csr_avmm_inst	59.61%	93.56%	28.42%	58.33%	56.23%	61.51%	
CSR.genblk1.u_rdl_base	60.52%	93.62%	31.22%		56.37%	60.86%	
mgmt_ltpi_top_inst	54.77%	67.46%	35.29%	45.94%	64.82%	60.34%	
GPIO.genblk1.mgmt_gpio_inst	86.59%	97.67%	55.85%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	64.60%	80.21%	47.27%	40.85%	78.76%	75.93%	
SMBus[0].genblk1.genblk1.smbus_relay_controller_0	66.03%	79.82%	62.25%	40.85%	72.06%	75.16%	
SMBus[0].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_controller_0	66.03%	79.82%	62.25%	40.85%	72.06%	75.16%	
SMBus[1].genblk1.smbus_echo_inst	83.69%	82.58%	78.57%		95.83%	77.78%	
SMBus[2].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[2].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[3].genblk1.smbus_echo_inst	83.69%	82.58%	78.57%		95.83%	77.78%	
SMBus[4].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[4].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[5].genblk1.smbus_echo_inst	83.69%	82.58%	78.57%		95.83%	77.78%	
UART.genblk1.mgmt_uart_inst	91.75%	97.96%	64.35%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_controller_...	65.27%	88.46%	51.54%	50.00%	63.64%	72.73%	
data_channel.genblk1.genblk1.mgmt_data_channel_controller_inst	70.89%	82.50%	51.94%	66.67%	66.67%	86.67%	
mgmt_phy_top_inst	53.54%	61.82%	30.73%	65.22%	57.68%	52.26%	
dynamic_pll.genblk1.m10_pll_top_inst	42.60%	53.12%	13.34%	63.16%	42.16%	41.20%	
ltpi_phy_rx_inst	68.26%	64.85%	83.38%	64.29%	67.47%	61.34%	
ltpi_phy_tx_inst	62.38%	65.92%	74.42%	62.50%	53.57%	55.50%	
mgmt_ltpi_frm_rx_inst	75.35%	92.57%	36.29%		82.86%	89.70%	
mgmt_ltpi_frm_tx_inst	71.88%	92.17%	27.66%		91.55%	76.16%	
mgmt_phy.controller.mgmt_phy_controller_inst	72.32%	94.57%	37.29%	68.97%	76.47%	84.31%	
pll_system_controller	35.43%	62.46%	4.68%		36.27%	38.31%	
u_avmm_CSR	63.48%	100.00%	40.43%			50.00%	
u_avmm_FPGA_inf	50.43%	100.00%	1.30%			50.00%	
u_avmm_bmc	50.43%	100.00%	1.30%			50.00%	
u_avmm_cntrl	50.43%	100.00%	1.30%			50.00%	
u_avmm_mm	63.48%	100.00%	40.43%			50.00%	
u_avmm_s	50.43%	100.00%	1.30%			50.00%	
dut_ltpi_target	52.80%	68.96%	32.71%	42.44%	61.51%	58.39%	
ltpi_csr_avmm_inst	61.82%	94.96%	30.16%	58.33%	62.26%	63.40%	
CSR.genblk1.u_rdl_base	63.60%	95.24%	33.95%		62.36%	62.84%	
mgmt_ltpi_top_inst	54.00%	67.30%	35.21%	42.18%	64.82%	60.47%	
GPIO.genblk1.mgmt_gpio_inst	86.65%	97.67%	56.05%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	62.70%	78.44%	44.78%	37.78%	78.61%	73.91%	
SMBus[0].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[0].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[1].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[2].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[2].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[3].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[4].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[4].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[5].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
UART.genblk1.mgmt_uart_inst	88.60%	97.96%	48.61%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_target_mm...	58.58%	65.96%	56.96%	50.00%	53.33%	66.67%	
data_channel.genblk1.genblk1.mgmt_data_channel_target_inst	75.92%	100.00%	46.25%	66.67%	66.67%	100.00%	
mgmt_phy_top_inst	54.25%	62.02%	31.34%	67.39%	58.04%	52.48%	
dynamic_pll.genblk1.m10_pll_top_inst	42.60%	53.12%	13.34%	63.16%	42.16%	41.20%	
ltpi_phy_rx_inst	69.28%	65.21%	83.52%	67.86%	67.66%	62.15%	
ltpi_phy_tx_inst	62.38%	65.92%	74.42%	62.50%	53.57%	55.50%	
mgmt_ltpi_frm_rx_inst	78.96%	95.71%	40.03%		88.57%	91.52%	
mgmt_ltpi_frm_tx_inst	71.83%	91.71%	28.49%		91.55%	75.58%	
mgmt_phy.target.mgmt_phy_target_inst	71.36%	92.55%	33.14%	72.41%	75.68%	83.02%	
pll_system_target	35.43%	62.46%	4.68%		36.27%	38.31%	
u_avmm_CSR	64.06%	100.00%	42.17%			50.00%	
u_avmm_FPGA_inf	50.43%	100.00%	1.30%			50.00%	
u_avmm_cntrl	50.43%	100.00%	1.30%			50.00%	
u_avmm_mm	64.06%	100.00%	42.17%			50.00%	
u_avmm_s	50.43%	100.00%	1.30%			50.00%	
u_avmm_trg	50.43%	100.00%	1.30%			50.00%	

Figure 30: Regression result 10 cycle - SV framework.

Appendix B: Repositories

Regression Result 20 cycle per test

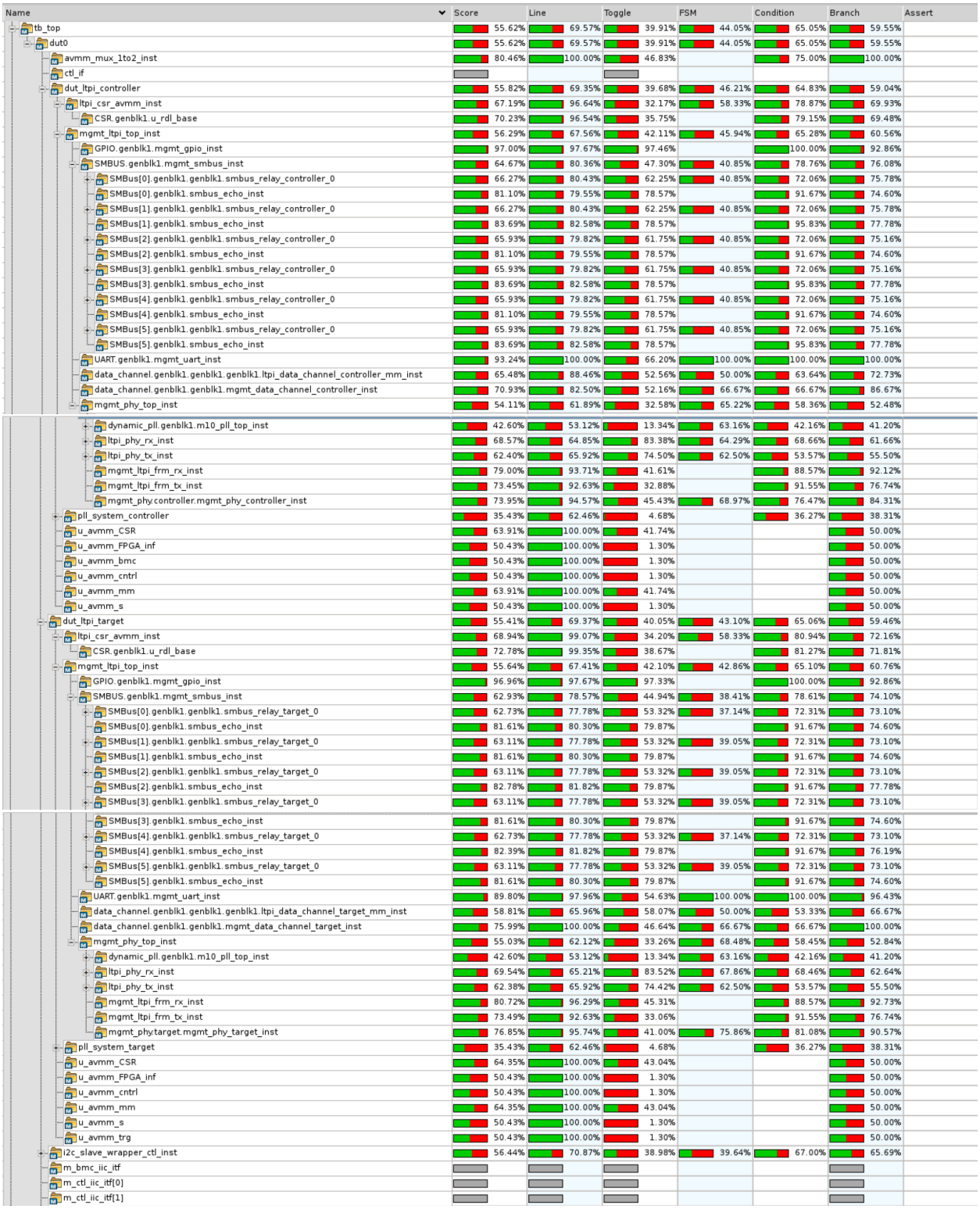


Name	Score	Line	Toggle	FSM	Condition	Branch	Assert
tb_top	53.69%	69.17%	35.55%	43.90%	61.17%	58.67%	
dut0	53.69%	69.17%	35.55%	43.90%	61.17%	58.67%	
avmm_mux_1to2_inst	80.17%	100.00%	45.67%		75.00%	100.00%	
ctl_if							
dut_ltpi_controller	53.89%	69.01%	34.99%	46.21%	61.06%	58.20%	
ltpi_csr_avmm_inst	60.29%	93.09%	28.22%	58.33%	59.25%	62.54%	
CSR.genblk1.u_rdl_base	61.61%	93.51%	31.15%		59.65%	62.12%	
mgmt_ltpi_top_inst	55.31%	67.49%	37.75%	45.94%	64.91%	60.43%	
GPIO.genblk1.mgmt_gpio_inst	91.60%	97.67%	75.86%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	64.53%	80.21%	47.27%	40.85%	78.41%	75.93%	
SMBus[0].genblk1.genblk1.smbus_relay_controller_0	66.27%	80.43%	62.25%	40.85%	72.06%	75.78%	
SMBus[0].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_controller_0	66.27%	80.43%	62.25%	40.85%	72.06%	75.78%	
SMBus[1].genblk1.smbus_echo_inst	83.69%	82.58%	78.57%		95.83%	77.78%	
SMBus[2].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[2].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[3].genblk1.smbus_echo_inst	81.87%	81.06%	78.57%		91.67%	76.19%	
SMBus[4].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[4].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[5].genblk1.smbus_echo_inst	81.87%	81.06%	78.57%		91.67%	76.19%	
UART.genblk1.mgmt_uart_inst	93.06%	100.00%	65.28%	100.00%	100.00%	100.00%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_controller_mm_inst	65.27%	88.46%	51.54%	50.00%	63.64%	72.73%	
data_channel.genblk1.genblk1.mgmt_data_channel_controller_inst	70.89%	82.50%	51.94%	66.67%	66.67%	86.67%	
mgmt_phy_top_inst	53.57%	61.85%	30.48%	65.22%	57.96%	52.36%	
dynamic_pll.genblk1.m10_pll_top_inst	42.60%	53.12%	13.34%	63.16%	42.16%	41.20%	
ltpi_phy_rx_inst	68.42%	64.85%	83.38%	64.29%	68.06%	61.50%	
ltpi_phy_tx_inst	62.40%	65.92%	74.50%	62.50%	53.57%	55.50%	
mgmt_ltpi_frm_rx_inst	76.10%	93.43%	35.17%		84.29%	91.52%	
mgmt_ltpi_frm_tx_inst	71.73%	92.17%	27.04%		91.55%	76.16%	
mgmt_phy.controller.mgmt_phy_controller_inst	71.50%	93.48%	36.21%	68.97%	76.47%	82.35%	
pll_system_controller	35.43%	62.46%	4.68%		36.27%	38.31%	
u_avmm_CSR	63.48%	100.00%	40.43%			50.00%	
u_avmm_FPGA_inf	50.43%	100.00%	1.30%			50.00%	
u_avmm_bmc	50.43%	100.00%	1.30%			50.00%	
u_avmm_cntrl	50.43%	100.00%	1.30%			50.00%	
u_avmm_mm	63.62%	100.00%	40.87%			50.00%	
u_avmm_s	50.43%	100.00%	1.30%			50.00%	
dut_ltpi_target	53.39%	68.89%	35.84%	42.84%	60.93%	58.48%	
ltpi_csr_avmm_inst	60.69%	93.28%	30.98%	58.33%	58.49%	62.37%	
CSR.genblk1.u_rdl_base	62.36%	93.73%	34.88%		58.88%	61.94%	
mgmt_ltpi_top_inst	54.77%	67.38%	38.22%	42.59%	64.96%	60.70%	
GPIO.genblk1.mgmt_gpio_inst	91.66%	97.67%	76.11%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	62.85%	78.57%	44.85%	38.10%	78.61%	74.10%	
SMBus[0].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[0].genblk1.smbus_echo_inst	82.39%	81.82%	79.87%		91.67%	76.19%	
SMBus[1].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[1].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[2].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[2].genblk1.smbus_echo_inst	82.78%	81.82%	79.87%		91.67%	77.78%	
SMBus[3].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[3].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[4].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[4].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[5].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
UART.genblk1.mgmt_uart_inst	89.43%	97.96%	52.78%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_target_mm_inst	58.58%	65.96%	56.96%	50.00%	53.33%	66.67%	
data_channel.genblk1.genblk1.mgmt_data_channel_target_inst	75.92%	100.00%	46.25%	66.67%	66.67%	100.00%	
mgmt_phy_top_inst	54.66%	62.07%	31.79%	68.48%	58.24%	52.74%	
dynamic_pll.genblk1.m10_pll_top_inst	42.60%	53.12%	13.34%	63.16%	42.16%	41.20%	
ltpi_phy_rx_inst	69.39%	65.21%	83.52%	67.86%	67.86%	62.48%	
ltpi_phy_tx_inst	62.38%	65.92%	74.42%	62.50%	53.57%	55.50%	
mgmt_ltpi_frm_rx_inst	79.50%	95.71%	40.98%		88.57%	92.73%	
mgmt_ltpi_frm_tx_inst	72.26%	92.17%	29.17%		91.55%	76.16%	
mgmt_phy.target.mgmt_phy_target_inst	75.19%	95.74%	34.57%	75.86%	81.08%	88.68%	
pll_system_target	35.43%	62.46%	4.68%		36.27%	38.31%	
u_avmm_CSR	64.06%	100.00%	42.17%			50.00%	
u_avmm_FPGA_inf	50.43%	100.00%	1.30%			50.00%	
u_avmm_cntrl	50.43%	100.00%	1.30%			50.00%	
u_avmm_mm	64.06%	100.00%	42.17%			50.00%	
u_avmm_s	50.43%	100.00%	1.30%			50.00%	
u_avmm_trg	50.43%	100.00%	1.30%			50.00%	

Figure 31: Regression result 20 cycles - SV framework.

Appendix B: Repositories

Regression Result 50 cycle per test

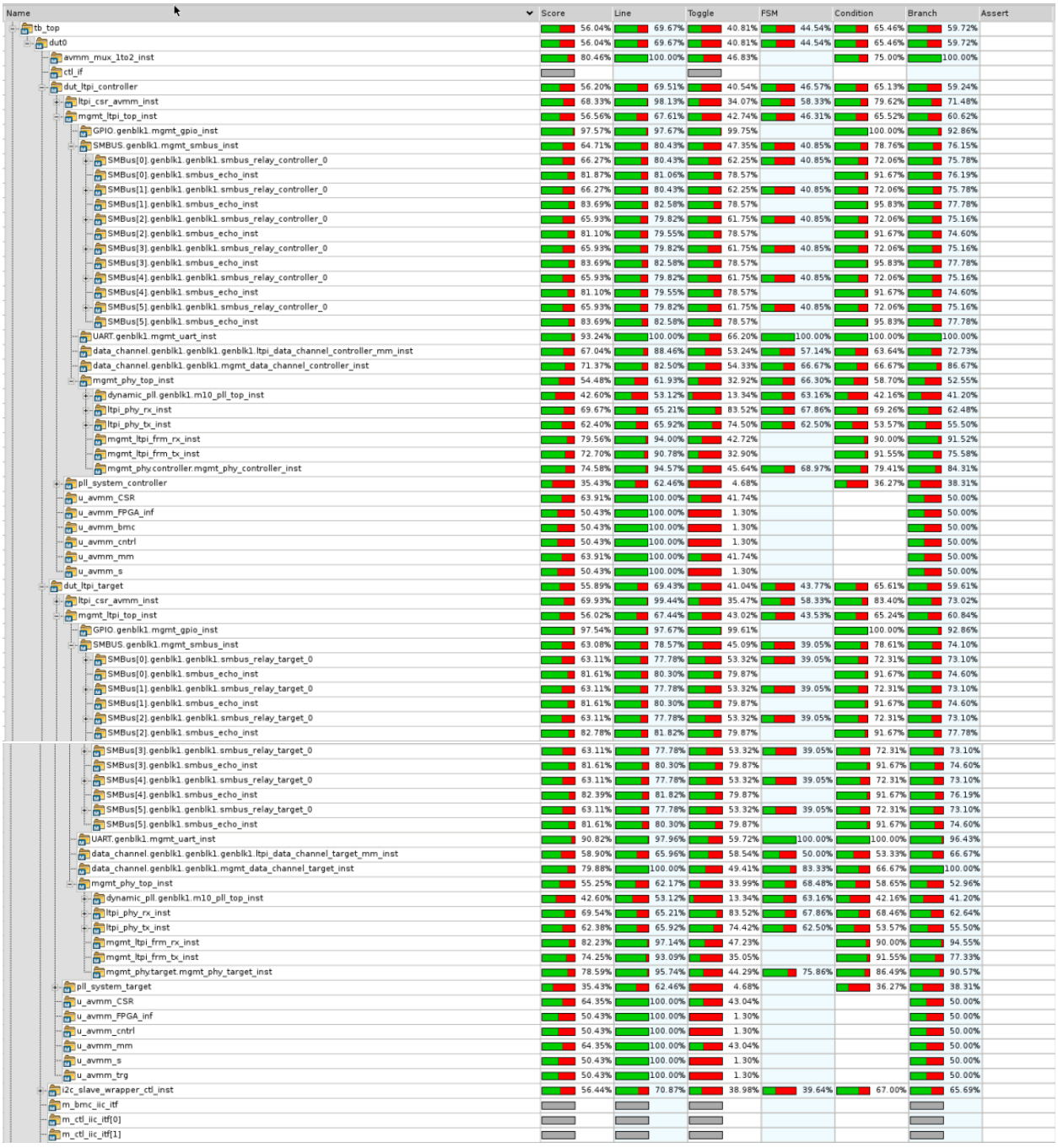


Name	Score	Line	Toggle	FSM	Condition	Branch	Assert
tb_top	55.62%	69.57%	39.91%	44.05%	65.05%	59.55%	
dut0	55.62%	69.57%	39.91%	44.05%	65.05%	59.55%	
avmm_mux_1to2_inst	80.46%	100.00%	46.83%		75.00%	100.00%	
ctl_if							
dut_ltpi_controller	55.82%	69.35%	39.68%	46.21%	64.83%	59.04%	
ltpi_csr_avmm_inst	67.19%	96.64%	32.17%	58.33%	78.87%	69.93%	
CSR.genblk1.u_rdl_base	70.23%	96.54%	35.75%		79.15%	69.48%	
mgmt_ltpi_top_inst	56.29%	67.56%	42.11%	45.94%	65.28%	60.56%	
GPIO.genblk1.mgmt_gpio_inst	97.00%	97.67%	97.46%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	64.67%	80.36%	47.30%	40.85%	78.76%	76.08%	
SMBus[0].genblk1.genblk1.smbus_relay_controller_0	66.27%	80.43%	62.25%	40.85%	72.06%	75.78%	
SMBus[0].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_controller_0	66.27%	80.43%	62.25%	40.85%	72.06%	75.78%	
SMBus[1].genblk1.smbus_echo_inst	83.69%	82.58%	78.57%		95.83%	77.78%	
SMBus[2].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[2].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[3].genblk1.smbus_echo_inst	83.69%	82.58%	78.57%		95.83%	77.78%	
SMBus[4].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[4].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[5].genblk1.smbus_echo_inst	83.69%	82.58%	78.57%		95.83%	77.78%	
UART.genblk1.mgmt_uart_inst	93.24%	100.00%	66.20%	100.00%	100.00%	100.00%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_controller_mm_inst	65.48%	88.46%	52.56%	50.00%	63.64%	72.73%	
data_channel.genblk1.genblk1.mgmt_data_channel_controller_inst	70.93%	82.50%	52.16%	66.67%	66.67%	86.67%	
mgmt_phy_top_inst	54.11%	61.89%	32.58%	65.22%	58.36%	52.48%	
dynamic_pll.genblk1.m10_pll_top_inst	42.60%	53.12%	13.34%	63.16%	42.16%	41.20%	
ltpi_phy_rx_inst	68.57%	64.85%	83.38%	64.29%	68.66%	61.66%	
ltpi_phy_tx_inst	62.40%	65.92%	74.50%	62.50%	53.57%	55.50%	
mgmt_ltpi_frm_rx_inst	79.00%	93.71%	41.61%		88.57%	92.12%	
mgmt_ltpi_frm_tx_inst	73.45%	92.63%	32.88%		91.55%	76.74%	
mgmt_phy_controller.mgmt_phy_controller_inst	73.95%	94.57%	45.43%	68.97%	76.47%	84.31%	
pll_system_controller	35.43%	62.46%	4.68%		36.27%	38.31%	
u_avmm_CSR	63.91%	100.00%	41.74%			50.00%	
u_avmm_FPGA_inf	50.43%	100.00%	1.30%			50.00%	
u_avmm_bmc	50.43%	100.00%	1.30%			50.00%	
u_avmm_cntrl	50.43%	100.00%	1.30%			50.00%	
u_avmm_mm	63.91%	100.00%	41.74%			50.00%	
u_avmm_s	50.43%	100.00%	1.30%			50.00%	
dut_ltpi_target	55.41%	69.37%	40.05%	43.10%	65.06%	59.46%	
ltpi_csr_avmm_inst	68.94%	99.07%	34.20%	58.33%	80.94%	72.16%	
CSR.genblk1.u_rdl_base	72.78%	99.35%	38.67%		81.27%	71.81%	
mgmt_ltpi_top_inst	55.64%	67.41%	42.10%	42.86%	65.10%	60.76%	
GPIO.genblk1.mgmt_gpio_inst	96.96%	97.67%	97.33%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	62.93%	78.57%	44.94%	38.41%	78.61%	74.10%	
SMBus[0].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[0].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[1].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[2].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[2].genblk1.smbus_echo_inst	82.78%	81.82%	79.87%		91.67%	77.78%	
SMBus[3].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[3].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[4].genblk1.genblk1.smbus_relay_target_0	62.73%	77.78%	53.32%	37.14%	72.31%	73.10%	
SMBus[4].genblk1.smbus_echo_inst	82.39%	81.82%	79.87%		91.67%	76.19%	
SMBus[5].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[5].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
UART.genblk1.mgmt_uart_inst	89.80%	97.96%	54.63%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_target_mm_inst	58.81%	65.96%	58.07%	50.00%	53.33%	66.67%	
data_channel.genblk1.genblk1.mgmt_data_channel_target_inst	75.99%	100.00%	46.64%	66.67%	66.67%	100.00%	
mgmt_phy_top_inst	55.03%	62.12%	33.26%	68.48%	58.45%	52.84%	
dynamic_pll.genblk1.m10_pll_top_inst	42.60%	53.12%	13.34%	63.16%	42.16%	41.20%	
ltpi_phy_rx_inst	69.54%	65.21%	83.52%	67.86%	68.46%	62.64%	
ltpi_phy_tx_inst	62.38%	65.92%	74.42%	62.50%	53.57%	55.50%	
mgmt_ltpi_frm_rx_inst	80.72%	96.29%	45.31%		88.57%	92.73%	
mgmt_ltpi_frm_tx_inst	73.49%	92.63%	33.06%		91.55%	76.74%	
mgmt_phy.target.mgmt_phy_target_inst	76.85%	95.74%	41.00%	75.86%	81.08%	90.57%	
pll_system_target	35.43%	62.46%	4.68%		36.27%	38.31%	
u_avmm_CSR	64.35%	100.00%	43.04%			50.00%	
u_avmm_FPGA_inf	50.43%	100.00%	1.30%			50.00%	
u_avmm_cntrl	50.43%	100.00%	1.30%			50.00%	
u_avmm_mm	64.35%	100.00%	43.04%			50.00%	
u_avmm_s	50.43%	100.00%	1.30%			50.00%	
u_avmm_trg	50.43%	100.00%	1.30%			50.00%	
i2c_slave_wrapper_ctl_inst	56.44%	70.87%	38.98%	39.64%	67.00%	65.69%	
m_bmc_iic_itf							
m_ctl_iic_itf[0]							
m_ctl_iic_itf[1]							

Figure 32: Regression result 50 cycles - SV framework.

Appendix B: Repositories

Regression Result 100 cycle per test



Name	Score	Line	Toggle	FSM	Condition	Branch	Assert
tb_top	56.04%	69.67%	40.81%	44.54%	65.46%	59.72%	
dut0	56.04%	69.67%	40.81%	44.54%	65.46%	59.72%	
avmm_mux_1to2_inst	80.46%	100.00%	46.83%		75.00%	100.00%	
ctl_if							
dut_ltpi_controller	56.20%	69.51%	40.54%	46.57%	65.13%	59.24%	
ltpi_csr_avmm_inst	68.33%	98.13%	34.07%	58.33%	79.62%	71.48%	
mgmt_ltpi_top_inst	56.56%	67.61%	42.74%	46.31%	65.52%	60.62%	
GPIO.genblk1.mgmt_gpio_inst	97.57%	97.67%	99.75%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	64.71%	80.43%	47.35%	40.85%	78.76%	76.15%	
SMBus[0].genblk1.genblk1.smbus_relay_controller_0	66.27%	80.43%	62.25%	40.85%	72.06%	75.78%	
SMBus[0].genblk1.smbus_echo_inst	81.87%	81.06%	78.57%		91.67%	76.19%	
SMBus[1].genblk1.genblk1.smbus_relay_controller_0	66.27%	80.43%	62.25%	40.85%	72.06%	75.78%	
SMBus[1].genblk1.smbus_echo_inst	83.69%	82.58%	78.57%		95.83%	77.78%	
SMBus[2].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[2].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[3].genblk1.smbus_echo_inst	83.69%	82.58%	78.57%		95.83%	77.78%	
SMBus[4].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[4].genblk1.smbus_echo_inst	81.10%	79.55%	78.57%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_controller_0	65.93%	79.82%	61.75%	40.85%	72.06%	75.16%	
SMBus[5].genblk1.smbus_echo_inst	83.69%	82.58%	78.57%		95.83%	77.78%	
UART.genblk1.mgmt_uart_inst	93.24%	100.00%	66.20%	100.00%	100.00%	100.00%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_controller_mm_inst	67.04%	88.46%	53.24%	57.14%	63.64%	72.73%	
data_channel.genblk1.genblk1.mgmt_data_channel_controller_inst	71.37%	82.50%	54.33%	66.67%	66.67%	86.67%	
mgmt_phy_top_inst	54.48%	61.93%	32.92%	66.30%	58.70%	52.55%	
dynamic_pll.genblk1.m10_pll_top_inst	42.60%	53.12%	13.34%	63.16%	42.16%	41.20%	
ltpi_phy_rx_inst	69.67%	65.21%	83.52%	67.86%	69.26%	62.48%	
ltpi_phy_tx_inst	62.40%	65.92%	74.50%	62.50%	53.57%	55.50%	
mgmt_ltpi_frm_rx_inst	79.56%	94.00%	42.72%		90.00%	91.52%	
mgmt_ltpi_frm_tx_inst	72.70%	90.78%	32.90%		91.55%	75.58%	
mgmt_phy.controller.mgmt_phy_controller_inst	74.58%	94.57%	45.64%	68.97%	79.41%	84.31%	
pll_system_controller	35.43%	62.46%	4.68%		36.27%	38.31%	
u_avmm_CSR	63.91%	100.00%	41.74%			50.00%	
u_avmm_FPGA_inf	50.43%	100.00%	1.30%			50.00%	
u_avmm_bmc	50.43%	100.00%	1.30%			50.00%	
u_avmm_cntrl	50.43%	100.00%	1.30%			50.00%	
u_avmm_mm	63.91%	100.00%	41.74%			50.00%	
u_avmm_s	50.43%	100.00%	1.30%			50.00%	
dut_ltpi_target	55.89%	69.43%	41.04%	43.77%	65.61%	59.61%	
ltpi_csr_avmm_inst	69.93%	99.44%	35.47%	58.33%	83.40%	73.02%	
mgmt_ltpi_top_inst	56.02%	67.44%	43.02%	43.53%	65.24%	60.84%	
GPIO.genblk1.mgmt_gpio_inst	97.54%	97.67%	99.61%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	63.08%	78.57%	45.09%	39.05%	78.61%	74.10%	
SMBus[0].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[0].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[1].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[2].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[2].genblk1.smbus_echo_inst	82.78%	81.82%	79.87%		91.67%	77.78%	
SMBus[3].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[3].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[4].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[4].genblk1.smbus_echo_inst	82.39%	81.82%	79.87%		91.67%	76.19%	
SMBus[5].genblk1.genblk1.smbus_relay_target_0	63.11%	77.78%	53.32%	39.05%	72.31%	73.10%	
SMBus[5].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
UART.genblk1.mgmt_uart_inst	90.82%	97.96%	59.72%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_target_mm_inst	58.90%	65.96%	58.54%	50.00%	53.33%	66.67%	
data_channel.genblk1.genblk1.mgmt_data_channel_target_inst	79.88%	100.00%	49.41%	83.33%	66.67%	100.00%	
mgmt_phy_top_inst	55.25%	62.17%	33.99%	68.48%	58.65%	52.96%	
dynamic_pll.genblk1.m10_pll_top_inst	42.60%	53.12%	13.34%	63.16%	42.16%	41.20%	
ltpi_phy_rx_inst	69.54%	65.21%	83.52%	67.86%	68.46%	62.64%	
ltpi_phy_tx_inst	62.38%	65.92%	74.42%	62.50%	53.57%	55.50%	
mgmt_ltpi_frm_rx_inst	82.23%	97.14%	47.23%		90.00%	94.55%	
mgmt_ltpi_frm_tx_inst	74.25%	93.09%	35.05%		91.55%	77.33%	
mgmt_phy.target.mgmt_phy_target_inst	78.59%	95.74%	44.29%	75.86%	86.49%	90.57%	
pll_system_target	35.43%	62.46%	4.68%		36.27%	38.31%	
u_avmm_CSR	64.35%	100.00%	43.04%			50.00%	
u_avmm_FPGA_inf	50.43%	100.00%	1.30%			50.00%	
u_avmm_cntrl	50.43%	100.00%	1.30%			50.00%	
u_avmm_mm	64.35%	100.00%	43.04%			50.00%	
u_avmm_s	50.43%	100.00%	1.30%			50.00%	
u_avmm_trg	50.43%	100.00%	1.30%			50.00%	
i2c_slave_wrapper_ctl_inst	56.44%	70.87%	38.98%	39.64%	67.00%	65.69%	
m_bmc_iic_itf							
m_ctl_iic_itf[0]							
m_ctl_iic_itf[1]							

Figure 33: Regression result 100 cycles - SV framework.

Appendix B: Repositories

Appendix D: Testcase Waveforms and Coverage Results for Python Framework

In this section, some results of testcase simulations and full coverage reports will be presented for the SystemVerilog framework. Figures 35 to 38 showcase the simulation of Link Training, CRC error injection during the Link Detect Stage, a UART simulation testcase, and an example of the memory map access respectively. Figures 39 to 44 show the regression results for 1 cycle per testcase, 4 cycles, 10, 20, 50, and 100, respectively.

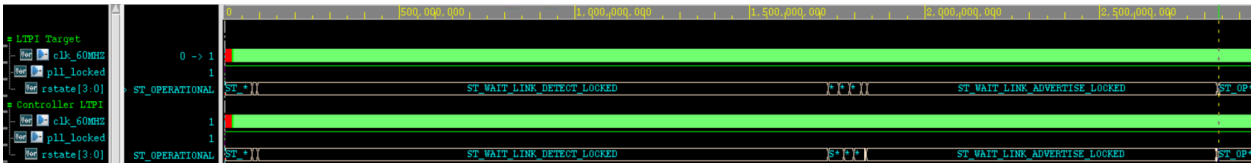


Figure 34: Link Training simulation - Python framework.

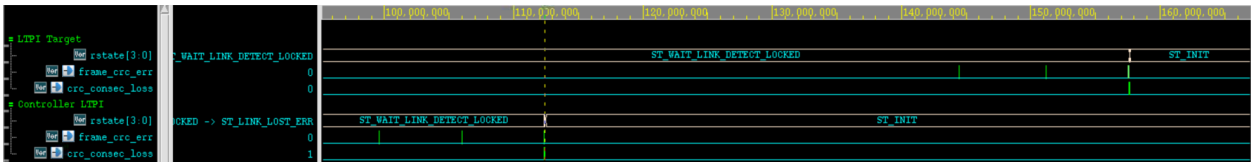


Figure 35: Link Detect error simulation - Python framework.

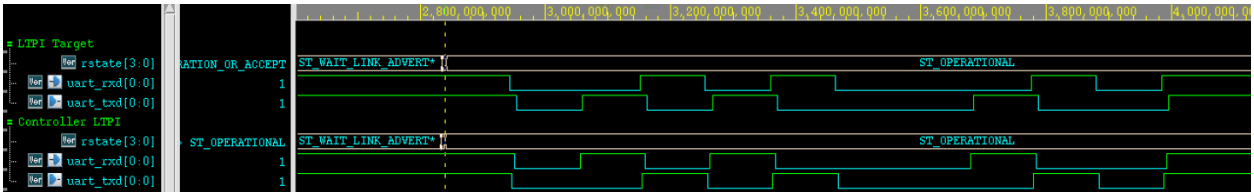


Figure 36: UART Basic simulation - Python framework.

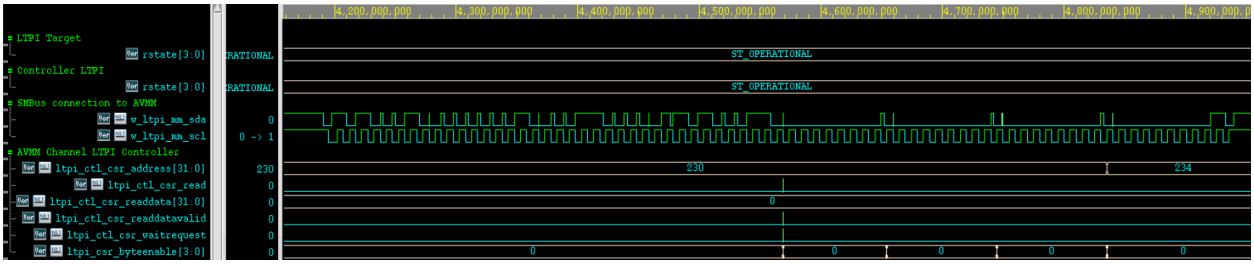
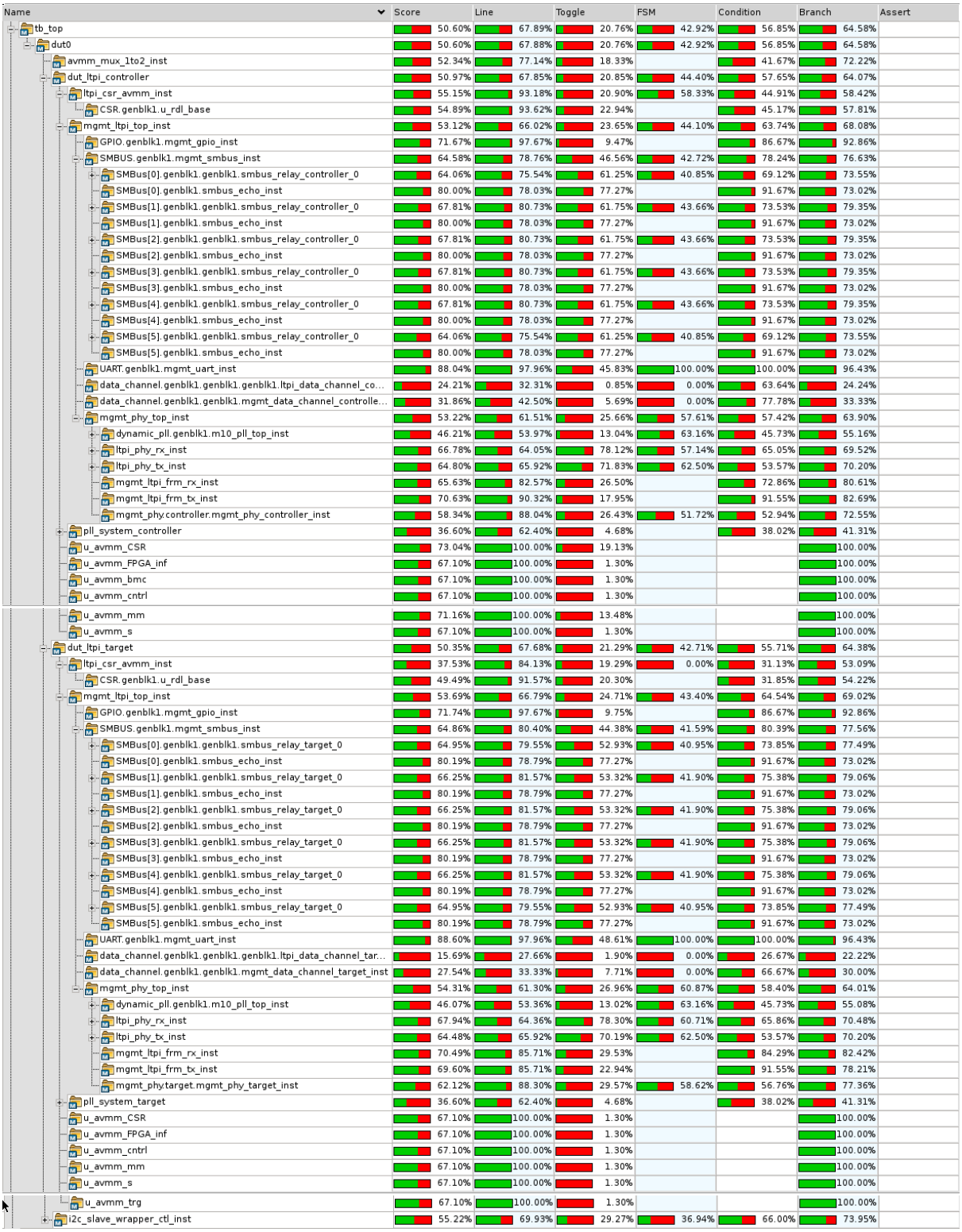


Figure 37: AVMM Controller simulation - Python framework.

Appendix B: Repositories

Regression Result 1 cycle per test

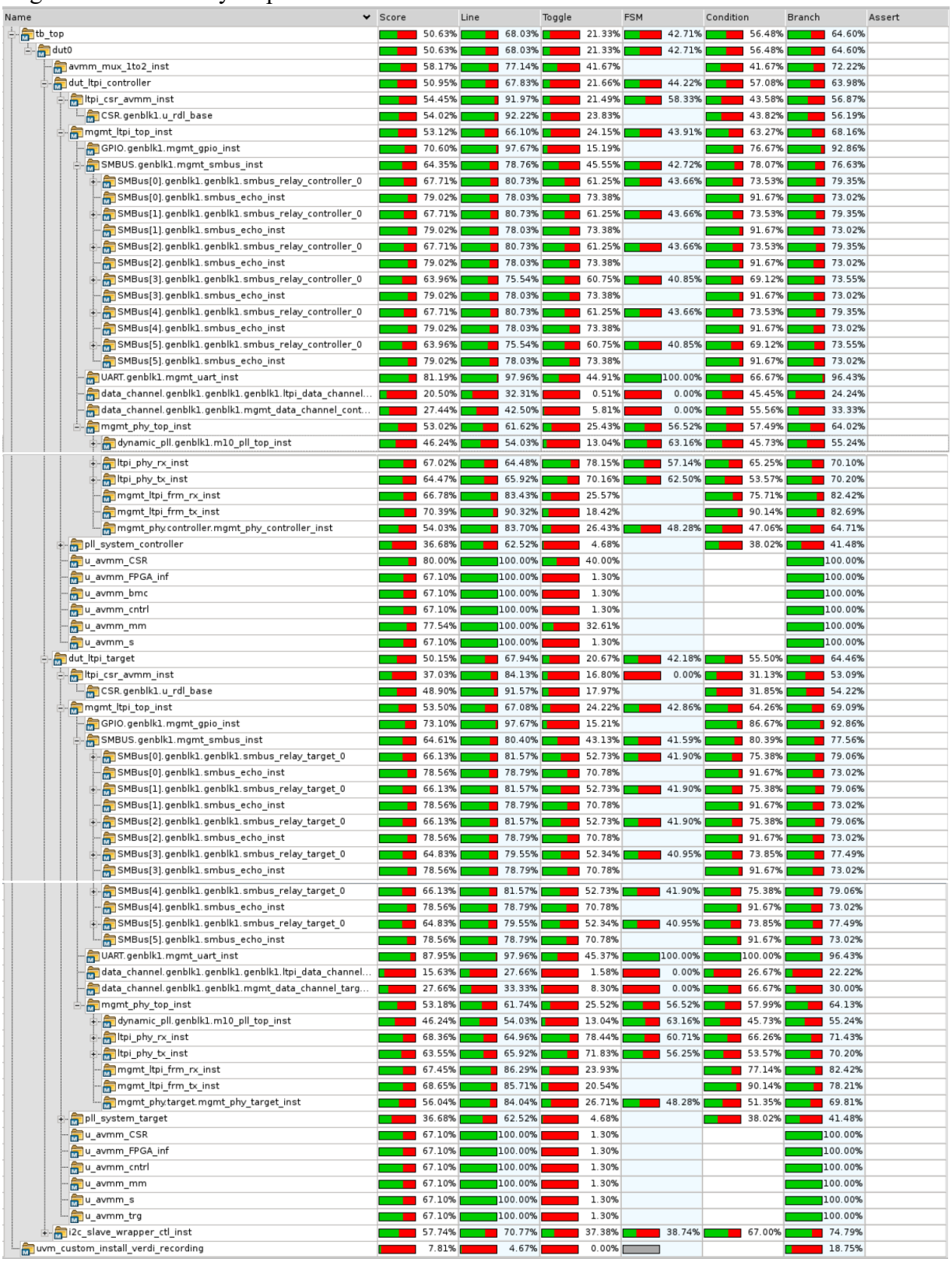


Name	Score	Line	Toggle	FSM	Condition	Branch	Assert
tb_top	50.60%	67.89%	20.76%	42.92%	56.85%	64.58%	
dut0	50.60%	67.88%	20.76%	42.92%	56.85%	64.58%	
avmm_mux_1to2_inst	52.34%	77.14%	18.33%		41.67%	72.22%	
dut_ltpi_controller	50.97%	67.85%	20.85%	44.40%	57.65%	64.07%	
ltpi_csr_avmm_inst	55.15%	93.18%	20.90%	58.33%	44.91%	58.42%	
CSR.genblk1.u_rdl_base	54.89%	93.62%	22.94%		45.17%	57.81%	
mgmt_ltpi_top_inst	53.12%	66.02%	23.65%	44.10%	63.74%	68.08%	
GPIO.genblk1.mgmt_gpio_inst	71.67%	97.67%	9.47%		86.67%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	64.58%	78.76%	46.56%	42.72%	78.24%	76.63%	
SMBus[0].genblk1.genblk1.smbus_relay_controller_0	64.06%	75.54%	61.25%	40.85%	69.12%	73.55%	
SMBus[0].genblk1.smbus_echo_inst	80.00%	78.03%	77.27%		91.67%	73.02%	
SMBus[1].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[1].genblk1.smbus_echo_inst	80.00%	78.03%	77.27%		91.67%	73.02%	
SMBus[2].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[2].genblk1.smbus_echo_inst	80.00%	78.03%	77.27%		91.67%	73.02%	
SMBus[3].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[3].genblk1.smbus_echo_inst	80.00%	78.03%	77.27%		91.67%	73.02%	
SMBus[4].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[4].genblk1.smbus_echo_inst	80.00%	78.03%	77.27%		91.67%	73.02%	
SMBus[5].genblk1.genblk1.smbus_relay_controller_0	64.06%	75.54%	61.25%	40.85%	69.12%	73.55%	
SMBus[5].genblk1.smbus_echo_inst	80.00%	78.03%	77.27%		91.67%	73.02%	
UART.genblk1.mgmt_uart_inst	88.04%	97.96%	45.83%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_co...	24.21%	32.31%	0.85%	0.00%	63.64%	24.24%	
data_channel.genblk1.genblk1.mgmt_data_channel_controlle...	31.86%	42.50%	5.69%	0.00%	77.78%	33.33%	
mgmt_phy_top_inst	53.22%	61.51%	25.66%	57.61%	57.42%	63.90%	
dynamic_pll.genblk1.m10_pll_top_inst	46.21%	53.97%	13.04%	63.16%	45.73%	55.16%	
ltpi_phy_rx_inst	66.78%	64.05%	78.12%	57.14%	65.05%	69.52%	
ltpi_phy_tx_inst	64.80%	65.92%	71.83%	62.50%	53.57%	70.20%	
mgmt_ltpi_frm_rx_inst	65.63%	82.57%	26.50%		72.86%	80.61%	
mgmt_ltpi_frm_tx_inst	70.63%	90.32%	17.95%		91.55%	82.69%	
mgmt_phy.controller.mgmt_phy_controller_inst	58.34%	88.04%	26.43%	51.72%	52.94%	72.55%	
pll_system_controller	36.60%	62.40%	4.68%		38.02%	41.31%	
u_avmm_CSR	73.04%	100.00%	19.13%			100.00%	
u_avmm_FPGA_inf	67.10%	100.00%	1.30%			100.00%	
u_avmm_bmc	67.10%	100.00%	1.30%			100.00%	
u_avmm_cntrl	67.10%	100.00%	1.30%			100.00%	
u_avmm_mm	71.16%	100.00%	13.48%			100.00%	
u_avmm_s	67.10%	100.00%	1.30%			100.00%	
dut_ltpi_target	50.35%	67.68%	21.29%	42.71%	55.71%	64.38%	
ltpi_csr_avmm_inst	37.53%	84.13%	19.29%	0.00%	31.13%	53.09%	
CSR.genblk1.u_rdl_base	49.49%	91.57%	20.30%		31.85%	54.22%	
mgmt_ltpi_top_inst	53.69%	66.79%	24.71%	43.40%	64.54%	69.02%	
GPIO.genblk1.mgmt_gpio_inst	71.74%	97.67%	9.75%		86.67%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	64.86%	80.40%	44.38%	41.59%	80.39%	77.56%	
SMBus[0].genblk1.genblk1.smbus_relay_target_0	64.95%	79.55%	52.93%	40.95%	73.85%	77.49%	
SMBus[0].genblk1.smbus_echo_inst	80.19%	78.79%	77.27%		91.67%	73.02%	
SMBus[1].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[1].genblk1.smbus_echo_inst	80.19%	78.79%	77.27%		91.67%	73.02%	
SMBus[2].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[2].genblk1.smbus_echo_inst	80.19%	78.79%	77.27%		91.67%	73.02%	
SMBus[3].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[3].genblk1.smbus_echo_inst	80.19%	78.79%	77.27%		91.67%	73.02%	
SMBus[4].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[4].genblk1.smbus_echo_inst	80.19%	78.79%	77.27%		91.67%	73.02%	
SMBus[5].genblk1.genblk1.smbus_relay_target_0	64.95%	79.55%	52.93%	40.95%	73.85%	77.49%	
SMBus[5].genblk1.smbus_echo_inst	80.19%	78.79%	77.27%		91.67%	73.02%	
UART.genblk1.mgmt_uart_inst	88.60%	97.96%	48.61%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_tar...	15.69%	27.66%	1.90%	0.00%	26.67%	22.22%	
data_channel.genblk1.genblk1.mgmt_data_channel_target_inst	27.54%	33.33%	7.71%	0.00%	66.67%	30.00%	
mgmt_phy_top_inst	54.31%	61.30%	26.96%	60.87%	58.40%	64.01%	
dynamic_pll.genblk1.m10_pll_top_inst	46.07%	53.36%	13.02%	63.16%	45.73%	55.08%	
ltpi_phy_rx_inst	67.94%	64.36%	78.30%	60.71%	65.86%	70.48%	
ltpi_phy_tx_inst	64.48%	65.92%	70.19%	62.50%	53.57%	70.20%	
mgmt_ltpi_frm_rx_inst	70.49%	85.71%	29.53%		84.29%	82.42%	
mgmt_ltpi_frm_tx_inst	69.60%	85.71%	22.94%		91.55%	78.21%	
mgmt_phy.target.mgmt_phy_target_inst	62.12%	88.30%	29.57%	58.62%	56.76%	77.36%	
pll_system_target	36.60%	62.40%	4.68%		38.02%	41.31%	
u_avmm_CSR	67.10%	100.00%	1.30%			100.00%	
u_avmm_FPGA_inf	67.10%	100.00%	1.30%			100.00%	
u_avmm_cntrl	67.10%	100.00%	1.30%			100.00%	
u_avmm_mm	67.10%	100.00%	1.30%			100.00%	
u_avmm_s	67.10%	100.00%	1.30%			100.00%	
u_avmm_trg	67.10%	100.00%	1.30%			100.00%	
i2c_slave_wrapper_ctl_inst	55.22%	69.93%	29.27%	36.94%	66.00%	73.95%	

Figure 38: Regression result 1 cycle - Python framework.

Appendix B: Repositories

Regression Result 4 cycle per test

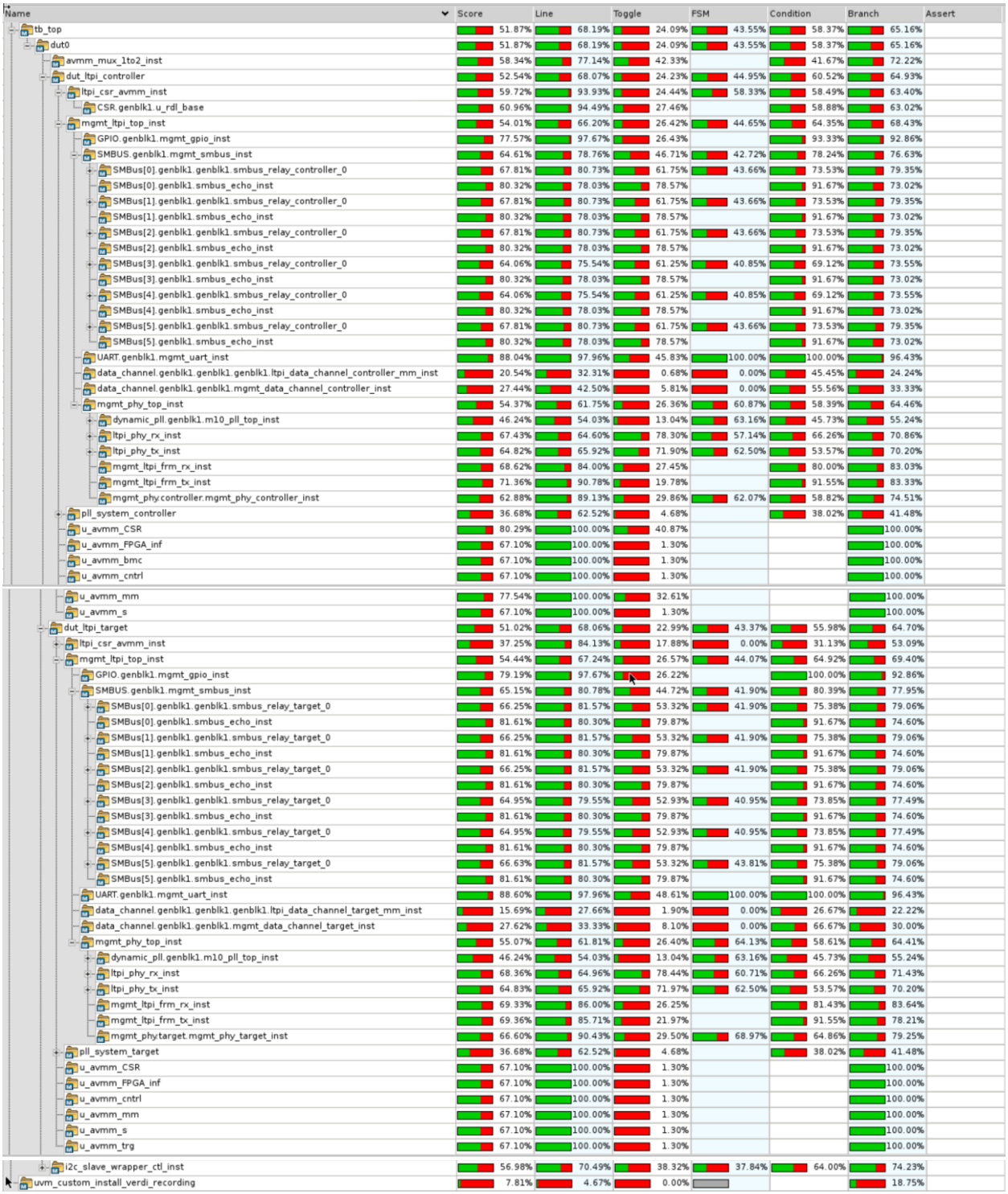


Name	Score	Line	Toggle	FSM	Condition	Branch	Assert
tb_top	50.63%	68.03%	21.33%	42.71%	56.48%	64.60%	
dut0	50.63%	68.03%	21.33%	42.71%	56.48%	64.60%	
avmm_mux_1to2_inst	58.17%	77.14%	41.67%		41.67%	72.22%	
dut_ltpi_controller	50.95%	67.83%	21.66%	44.22%	57.08%	63.98%	
ltpi_csr_avmm_inst	54.45%	91.97%	21.49%	58.33%	43.58%	56.87%	
CSR.genblk1.u_rdl_base	54.02%	92.22%	23.83%		43.82%	56.19%	
mgmt_ltpi_top_inst	53.12%	66.10%	24.15%	43.91%	63.27%	68.16%	
GPIO.genblk1.mgmt_gpio_inst	70.60%	97.67%	15.19%		76.67%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	64.35%	78.76%	45.55%	42.72%	78.07%	76.63%	
SMBus[0].genblk1.genblk1.smbus_relay_controller_0	67.71%	80.73%	61.25%	43.66%	73.53%	79.35%	
SMBus[0].genblk1.smbus_echo_inst	79.02%	78.03%	73.38%		91.67%	73.02%	
SMBus[1].genblk1.genblk1.smbus_relay_controller_0	67.71%	80.73%	61.25%	43.66%	73.53%	79.35%	
SMBus[1].genblk1.smbus_echo_inst	79.02%	78.03%	73.38%		91.67%	73.02%	
SMBus[2].genblk1.genblk1.smbus_relay_controller_0	67.71%	80.73%	61.25%	43.66%	73.53%	79.35%	
SMBus[2].genblk1.smbus_echo_inst	79.02%	78.03%	73.38%		91.67%	73.02%	
SMBus[3].genblk1.genblk1.smbus_relay_controller_0	63.96%	75.54%	60.75%	40.85%	69.12%	73.55%	
SMBus[3].genblk1.smbus_echo_inst	79.02%	78.03%	73.38%		91.67%	73.02%	
SMBus[4].genblk1.genblk1.smbus_relay_controller_0	67.71%	80.73%	61.25%	43.66%	73.53%	79.35%	
SMBus[4].genblk1.smbus_echo_inst	79.02%	78.03%	73.38%		91.67%	73.02%	
SMBus[5].genblk1.genblk1.smbus_relay_controller_0	63.96%	75.54%	60.75%	40.85%	69.12%	73.55%	
SMBus[5].genblk1.smbus_echo_inst	79.02%	78.03%	73.38%		91.67%	73.02%	
UART.genblk1.mgmt_uart_inst	81.19%	97.96%	44.91%	100.00%	66.67%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel...	20.50%	32.31%	0.51%	0.00%	45.45%	24.24%	
data_channel.genblk1.genblk1.mgmt_data_channel_cont...	27.44%	42.50%	5.81%	0.00%	55.56%	33.33%	
mgmt_phy_top_inst	53.02%	61.62%	25.43%	56.52%	57.49%	64.02%	
dynamic_pll.genblk1.m10_pll_top_inst	46.24%	54.03%	13.04%	63.16%	45.73%	55.24%	
ltpi_phy_rx_inst	67.02%	64.48%	78.15%	57.14%	65.25%	70.10%	
ltpi_phy_tx_inst	64.47%	65.92%	70.16%	62.50%	53.57%	70.20%	
mgmt_ltpi_frm_rx_inst	66.78%	83.43%	25.57%		75.71%	82.42%	
mgmt_ltpi_frm_tx_inst	70.39%	90.32%	18.42%		90.14%	82.69%	
mgmt_phy_controller.mgmt_phy_controller_inst	54.03%	83.70%	26.43%	48.28%	47.06%	64.71%	
pll_system_controller	36.68%	62.52%	4.68%		38.02%	41.48%	
u_avmm_CSR	80.00%	100.00%	40.00%			100.00%	
u_avmm_FPGA_inf	67.10%	100.00%	1.30%			100.00%	
u_avmm_bmc	67.10%	100.00%	1.30%			100.00%	
u_avmm_cntrl	67.10%	100.00%	1.30%			100.00%	
u_avmm_mm	77.54%	100.00%	32.61%			100.00%	
u_avmm_s	67.10%	100.00%	1.30%			100.00%	
dut_ltpi_target	50.15%	67.94%	20.67%	42.18%	55.50%	64.46%	
ltpi_csr_avmm_inst	37.03%	84.13%	16.80%	0.00%	31.13%	53.09%	
CSR.genblk1.u_rdl_base	48.90%	91.57%	17.97%		31.85%	54.22%	
mgmt_ltpi_top_inst	53.50%	67.08%	24.22%	42.86%	64.26%	69.09%	
GPIO.genblk1.mgmt_gpio_inst	73.10%	97.67%	15.21%		86.67%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	64.61%	80.40%	43.13%	41.59%	80.39%	77.56%	
SMBus[0].genblk1.genblk1.smbus_relay_target_0	66.13%	81.57%	52.73%	41.90%	75.38%	79.06%	
SMBus[0].genblk1.smbus_echo_inst	78.56%	78.79%	70.78%		91.67%	73.02%	
SMBus[1].genblk1.genblk1.smbus_relay_target_0	66.13%	81.57%	52.73%	41.90%	75.38%	79.06%	
SMBus[1].genblk1.smbus_echo_inst	78.56%	78.79%	70.78%		91.67%	73.02%	
SMBus[2].genblk1.genblk1.smbus_relay_target_0	66.13%	81.57%	52.73%	41.90%	75.38%	79.06%	
SMBus[2].genblk1.smbus_echo_inst	78.56%	78.79%	70.78%		91.67%	73.02%	
SMBus[3].genblk1.genblk1.smbus_relay_target_0	64.83%	79.55%	52.34%	40.95%	73.85%	77.49%	
SMBus[3].genblk1.smbus_echo_inst	78.56%	78.79%	70.78%		91.67%	73.02%	
SMBus[4].genblk1.genblk1.smbus_relay_target_0	66.13%	81.57%	52.73%	41.90%	75.38%	79.06%	
SMBus[4].genblk1.smbus_echo_inst	78.56%	78.79%	70.78%		91.67%	73.02%	
SMBus[5].genblk1.genblk1.smbus_relay_target_0	64.83%	79.55%	52.34%	40.95%	73.85%	77.49%	
SMBus[5].genblk1.smbus_echo_inst	78.56%	78.79%	70.78%		91.67%	73.02%	
UART.genblk1.mgmt_uart_inst	87.95%	97.96%	45.37%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel...	15.63%	27.66%	1.58%	0.00%	26.67%	22.22%	
data_channel.genblk1.genblk1.mgmt_data_channel_targ...	27.66%	33.33%	8.30%	0.00%	66.67%	30.00%	
mgmt_phy_top_inst	53.18%	61.74%	25.52%	56.52%	57.99%	64.13%	
dynamic_pll.genblk1.m10_pll_top_inst	46.24%	54.03%	13.04%	63.16%	45.73%	55.24%	
ltpi_phy_rx_inst	68.36%	64.96%	78.44%	60.71%	66.26%	71.43%	
ltpi_phy_tx_inst	63.55%	65.92%	71.83%	56.25%	53.57%	70.20%	
mgmt_ltpi_frm_rx_inst	67.45%	86.29%	23.93%		77.14%	82.42%	
mgmt_ltpi_frm_tx_inst	68.65%	85.71%	20.54%		90.14%	78.21%	
mgmt_phy.target.mgmt_phy_target_inst	56.04%	84.04%	26.71%	48.28%	51.35%	69.81%	
pll_system_target	36.68%	62.52%	4.68%		38.02%	41.48%	
u_avmm_CSR	67.10%	100.00%	1.30%			100.00%	
u_avmm_FPGA_inf	67.10%	100.00%	1.30%			100.00%	
u_avmm_cntrl	67.10%	100.00%	1.30%			100.00%	
u_avmm_mm	67.10%	100.00%	1.30%			100.00%	
u_avmm_s	67.10%	100.00%	1.30%			100.00%	
u_avmm_trg	67.10%	100.00%	1.30%			100.00%	
i2c_slave_wrapper_ctl_inst	57.74%	70.77%	37.38%	38.74%	67.00%	74.79%	
uvm_custom_install_verdi_recording	7.81%	4.67%	0.00%			18.75%	

Figure 39: Regression result 4 cycles - Python framework.

Appendix B: Repositories

Regression Result 10 cycle per test

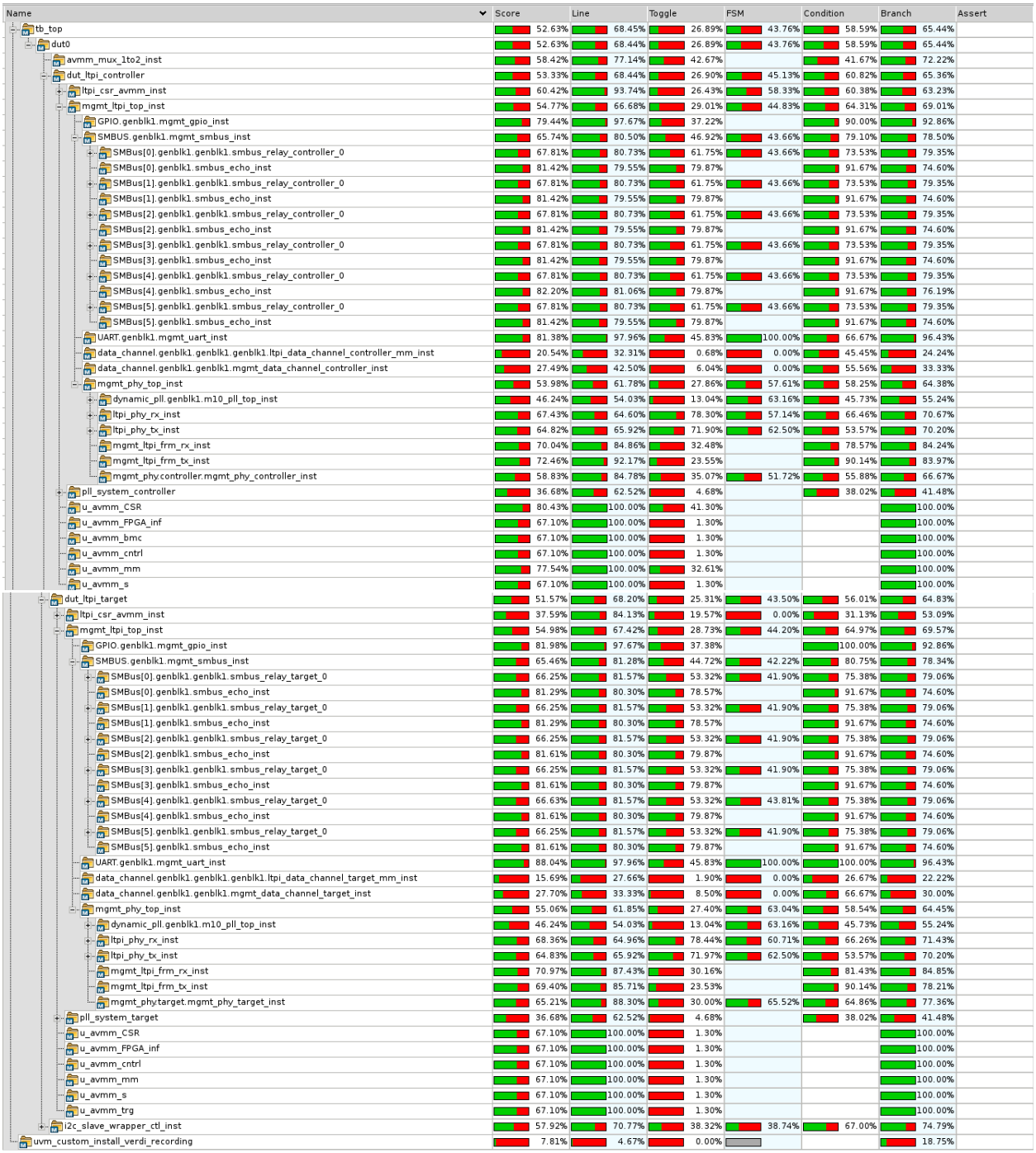


Name	Score	Line	Toggle	FSM	Condition	Branch	Assert
tb_top	51.87%	68.19%	24.09%	43.55%	58.37%	65.16%	
dut0	51.87%	68.19%	24.09%	43.55%	58.37%	65.16%	
avmm_mux_1to2_inst	58.34%	77.14%	42.33%		41.67%	72.22%	
dut_ltpi_controller	52.54%	68.07%	24.23%	44.95%	60.52%	64.93%	
ltpi_csr_avmm_inst	59.72%	93.93%	24.44%	58.33%	58.49%	63.40%	
CSR.genblk1.u_rdl_base	60.96%	94.49%	27.46%		58.88%	63.02%	
mgmt_ltpi_top_inst	54.01%	66.20%	26.42%	44.65%	64.35%	68.43%	
GPIO.genblk1.mgmt_gpio_inst	77.57%	97.67%	26.43%		93.33%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	64.61%	78.76%	46.71%	42.72%	78.24%	76.63%	
SMBus[0].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[0].genblk1.smbus_echo_inst	80.32%	78.03%	78.57%		91.67%	73.02%	
SMBus[1].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[1].genblk1.smbus_echo_inst	80.32%	78.03%	78.57%		91.67%	73.02%	
SMBus[2].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[2].genblk1.smbus_echo_inst	80.32%	78.03%	78.57%		91.67%	73.02%	
SMBus[3].genblk1.genblk1.smbus_relay_controller_0	64.06%	75.54%	61.25%	40.85%	69.12%	73.55%	
SMBus[3].genblk1.smbus_echo_inst	80.32%	78.03%	78.57%		91.67%	73.02%	
SMBus[4].genblk1.genblk1.smbus_relay_controller_0	64.06%	75.54%	61.25%	40.85%	69.12%	73.55%	
SMBus[4].genblk1.smbus_echo_inst	80.32%	78.03%	78.57%		91.67%	73.02%	
SMBus[5].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[5].genblk1.smbus_echo_inst	80.32%	78.03%	78.57%		91.67%	73.02%	
UART.genblk1.mgmt_uart_inst	88.04%	97.96%	45.83%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_controller_mm_inst	20.54%	32.31%	0.68%	0.00%	45.45%	24.24%	
data_channel.genblk1.genblk1.mgmt_data_channel_controller_inst	27.44%	42.50%	5.81%	0.00%	55.56%	33.33%	
mgmt_phy_top_inst	54.37%	61.75%	26.36%	60.87%	58.39%	64.46%	
dynamic_pll.genblk1.m10_pll_top_inst	46.24%	54.03%	13.04%	63.16%	45.73%	55.24%	
ltpi_phy_rx_inst	67.43%	64.60%	78.30%	57.14%	66.26%	70.86%	
ltpi_phy_tx_inst	64.82%	65.92%	71.90%	62.50%	53.57%	70.20%	
mgmt_ltpi_frm_rx_inst	68.62%	84.00%	27.45%		80.00%	83.03%	
mgmt_ltpi_frm_tx_inst	71.36%	90.78%	19.78%		91.55%	83.33%	
mgmt_phy_controller.mgmt_phy_controller_inst	62.88%	89.13%	29.86%	62.07%	58.82%	74.51%	
pll_system_controller	36.68%	62.52%	4.68%		38.02%	41.48%	
u_avmm_CSR	80.29%	100.00%	40.87%			100.00%	
u_avmm_FPGA_inf	67.10%	100.00%	1.30%			100.00%	
u_avmm_bmc	67.10%	100.00%	1.30%			100.00%	
u_avmm_cntrl	67.10%	100.00%	1.30%			100.00%	
u_avmm_mm	77.54%	100.00%	32.61%			100.00%	
u_avmm_s	67.10%	100.00%	1.30%			100.00%	
dut_ltpi_target	51.02%	68.06%	22.99%	43.37%	55.98%	64.70%	
ltpi_csr_avmm_inst	37.25%	84.13%	17.88%	0.00%	31.13%	53.09%	
mgmt_ltpi_top_inst	54.44%	67.24%	26.57%	44.07%	64.92%	69.40%	
GPIO.genblk1.mgmt_gpio_inst	79.19%	97.67%	26.22%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	65.15%	80.78%	44.72%	41.90%	80.39%	77.95%	
SMBus[0].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[0].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[1].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[2].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[2].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_target_0	64.95%	79.55%	52.93%	40.95%	73.85%	77.49%	
SMBus[3].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[4].genblk1.genblk1.smbus_relay_target_0	64.95%	79.55%	52.93%	40.95%	73.85%	77.49%	
SMBus[4].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_target_0	66.63%	81.57%	53.32%	43.81%	75.38%	79.06%	
SMBus[5].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
UART.genblk1.mgmt_uart_inst	88.60%	97.96%	48.61%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_target_mm_inst	15.69%	27.66%	1.90%	0.00%	26.67%	22.22%	
data_channel.genblk1.genblk1.mgmt_data_channel_target_inst	27.62%	33.33%	8.10%	0.00%	66.67%	30.00%	
mgmt_phy_top_inst	55.07%	61.81%	26.40%	64.13%	58.61%	64.41%	
dynamic_pll.genblk1.m10_pll_top_inst	46.24%	54.03%	13.04%	63.16%	45.73%	55.24%	
ltpi_phy_rx_inst	68.36%	64.96%	78.44%	60.71%	66.26%	71.43%	
ltpi_phy_tx_inst	64.83%	65.92%	71.97%	62.50%	53.57%	70.20%	
mgmt_ltpi_frm_rx_inst	69.33%	86.00%	26.25%		81.43%	83.64%	
mgmt_ltpi_frm_tx_inst	69.36%	85.71%	21.97%		91.55%	78.21%	
mgmt_phy_target.mgmt_phy_target_inst	66.60%	90.43%	29.50%	68.97%	64.86%	79.25%	
pll_system_target	36.68%	62.52%	4.68%		38.02%	41.48%	
u_avmm_CSR	67.10%	100.00%	1.30%			100.00%	
u_avmm_FPGA_inf	67.10%	100.00%	1.30%			100.00%	
u_avmm_cntrl	67.10%	100.00%	1.30%			100.00%	
u_avmm_mm	67.10%	100.00%	1.30%			100.00%	
u_avmm_s	67.10%	100.00%	1.30%			100.00%	
u_avmm_trg	67.10%	100.00%	1.30%			100.00%	
i2c_slave_wrapper_ctl_inst	56.98%	70.49%	38.32%	37.84%	64.00%	74.23%	
uvm_custom_install_verdi_recording	7.81%	4.67%	0.00%			18.75%	

Figure 40: Regression result 10 cycles - Python framework.

Appendix B: Repositories

Regression Result 20 cycle per test

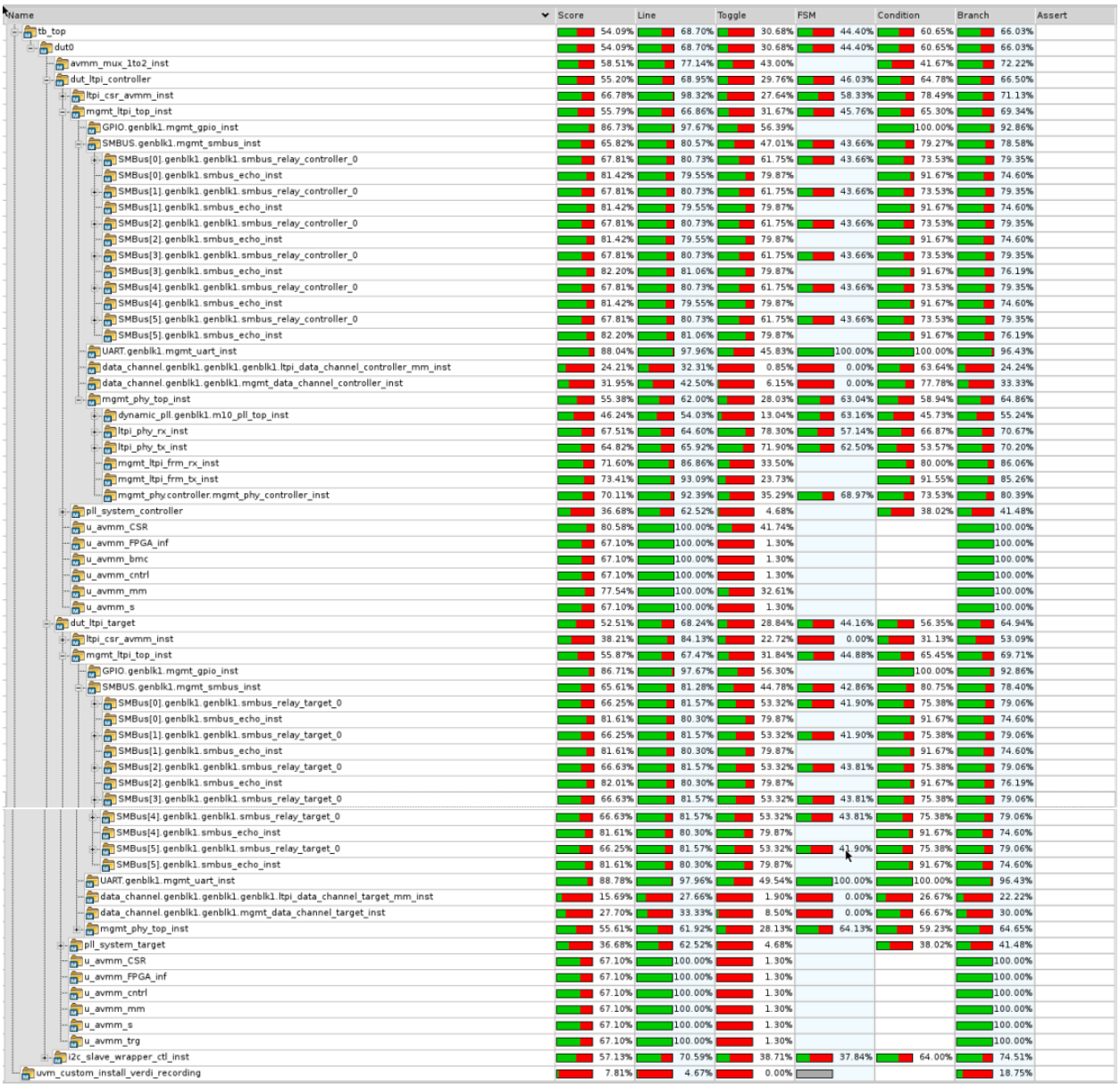


Name	Score	Line	Toggle	FSM	Condition	Branch	Assert
tb_top	52.63%	68.45%	26.89%	43.76%	58.59%	65.44%	
dut0	52.63%	68.44%	26.89%	43.76%	58.59%	65.44%	
avmm_mux_1to2_inst	58.42%	77.14%	42.67%		41.67%	72.22%	
dut_ltpi_controller	53.33%	68.44%	26.90%	45.13%	60.82%	65.36%	
ltpi_csr_avmm_inst	60.42%	93.74%	26.43%	58.33%	60.38%	63.23%	
mgmt_ltpi_top_inst	54.77%	66.68%	29.01%	44.83%	64.31%	69.01%	
GPIO.genblk1.mgmt_gpio_inst	79.44%	97.67%	37.22%		90.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	65.74%	80.50%	46.92%	43.66%	79.10%	78.50%	
SMBus[0].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[0].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[1].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
SMBus[2].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[2].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[3].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
SMBus[4].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[4].genblk1.smbus_echo_inst	82.20%	81.06%	79.87%		91.67%	76.19%	
SMBus[5].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[5].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
UART.genblk1.mgmt_uart_inst	81.38%	97.96%	45.83%	100.00%	66.67%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_controller_mm_inst	20.54%	32.31%	0.68%	0.00%	45.45%	24.24%	
data_channel.genblk1.genblk1.mgmt_data_channel_controller_inst	27.49%	42.50%	6.04%	0.00%	55.56%	33.33%	
mgmt_phy_top_inst	53.98%	61.78%	27.86%	57.61%	58.25%	64.38%	
dynamic_pll.genblk1.m10_pll_top_inst	46.24%	54.03%	13.04%	63.16%	45.73%	55.24%	
ltpi_phy_rx_inst	67.43%	64.60%	78.30%	57.14%	66.46%	70.67%	
ltpi_phy_tx_inst	64.82%	65.92%	71.90%	62.50%	53.57%	70.20%	
mgmt_ltpi_frm_rx_inst	70.04%	84.86%	32.48%		78.57%	84.24%	
mgmt_ltpi_frm_tx_inst	72.46%	92.17%	23.55%		90.14%	83.97%	
mgmt_phy_controller.mgmt_phy_controller_inst	58.83%	84.78%	35.07%	51.72%	55.88%	66.67%	
pll_system_controller	36.68%	62.52%	4.68%		38.02%	41.48%	
u_avmm_CSR	80.43%	100.00%	41.30%			100.00%	
u_avmm_FPGA_inf	67.10%	100.00%	1.30%			100.00%	
u_avmm_bmc	67.10%	100.00%	1.30%			100.00%	
u_avmm_cntrl	67.10%	100.00%	1.30%			100.00%	
u_avmm_mm	77.54%	100.00%	32.61%			100.00%	
u_avmm_s	67.10%	100.00%	1.30%			100.00%	
dut_ltpi_target	51.57%	68.20%	25.31%	43.50%	56.01%	64.83%	
ltpi_csr_avmm_inst	37.59%	84.13%	19.57%	0.00%	31.13%	53.09%	
mgmt_ltpi_top_inst	54.98%	67.42%	28.73%	44.20%	64.97%	69.57%	
GPIO.genblk1.mgmt_gpio_inst	81.98%	97.67%	37.38%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	65.46%	81.28%	44.72%	42.22%	80.75%	78.34%	
SMBus[0].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[0].genblk1.smbus_echo_inst	81.29%	80.30%	78.57%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[1].genblk1.smbus_echo_inst	81.29%	80.30%	78.57%		91.67%	74.60%	
SMBus[2].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[2].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[3].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[4].genblk1.genblk1.smbus_relay_target_0	66.63%	81.57%	53.32%	43.81%	75.38%	79.06%	
SMBus[4].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[5].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
UART.genblk1.mgmt_uart_inst	88.04%	97.96%	45.83%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_target_mm_inst	15.69%	27.66%	1.90%	0.00%	26.67%	22.22%	
data_channel.genblk1.genblk1.mgmt_data_channel_target_inst	27.70%	33.33%	8.50%	0.00%	66.67%	30.00%	
mgmt_phy_top_inst	55.06%	61.85%	27.40%	63.04%	58.54%	64.45%	
dynamic_pll.genblk1.m10_pll_top_inst	46.24%	54.03%	13.04%	63.16%	45.73%	55.24%	
ltpi_phy_rx_inst	68.36%	64.96%	78.44%	60.71%	66.26%	71.43%	
ltpi_phy_tx_inst	64.83%	65.92%	71.97%	62.50%	53.57%	70.20%	
mgmt_ltpi_frm_rx_inst	70.97%	87.43%	30.16%		81.43%	84.85%	
mgmt_ltpi_frm_tx_inst	69.40%	85.71%	23.53%		90.14%	78.21%	
mgmt_phytarget.mgmt_phy_target_inst	65.21%	88.30%	30.00%	65.52%	64.86%	77.36%	
pll_system_target	36.68%	62.52%	4.68%		38.02%	41.48%	
u_avmm_CSR	67.10%	100.00%	1.30%			100.00%	
u_avmm_FPGA_inf	67.10%	100.00%	1.30%			100.00%	
u_avmm_cntrl	67.10%	100.00%	1.30%			100.00%	
u_avmm_mm	67.10%	100.00%	1.30%			100.00%	
u_avmm_s	67.10%	100.00%	1.30%			100.00%	
u_avmm_trg	67.10%	100.00%	1.30%			100.00%	
i2c_slave_wrapper_ctl_inst	57.92%	70.77%	38.32%	38.74%	67.00%	74.79%	
uvm_custom_install_verdi_recording	7.81%	4.67%	0.00%			18.75%	

Figure 41: Regression result 20 cycles - Python framework.

Appendix B: Repositories

Regression Result 50 cycle per test

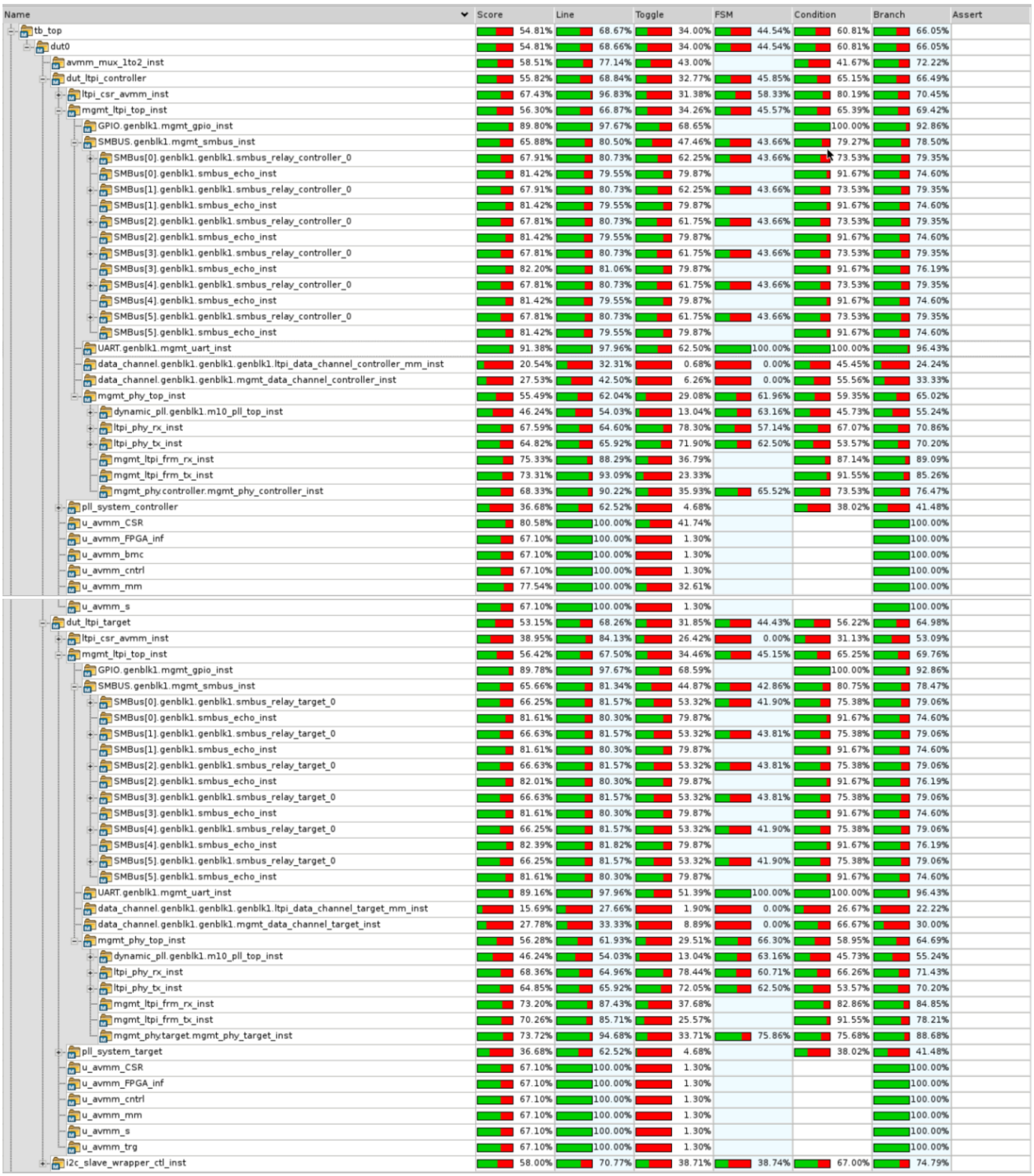


Name	Score	Line	Toggle	FSM	Condition	Branch	Assert
tb_top	54.09%	68.70%	30.68%	44.40%	60.65%	66.03%	
dut0	54.09%	68.70%	30.68%	44.40%	60.65%	66.03%	
avmm_mux_1to2_inst	58.51%	77.14%	43.00%		41.67%	72.22%	
dut_ltpi_controller	55.20%	68.95%	29.76%	46.03%	64.78%	66.50%	
ltpi_csr_avmm_inst	66.78%	98.32%	27.64%	58.33%	78.49%	71.13%	
mgmt_ltpi_top_inst	55.79%	66.86%	31.67%	45.76%	65.30%	69.34%	
GPIO.genblk1.mgmt_gpio_inst	86.73%	97.67%	56.39%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	65.82%	80.57%	47.01%	43.66%	79.27%	78.58%	
SMBus[0].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[0].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[1].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
SMBus[2].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[2].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[3].genblk1.smbus_echo_inst	82.20%	81.06%	79.87%		91.67%	76.19%	
SMBus[4].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[4].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[5].genblk1.smbus_echo_inst	82.20%	81.06%	79.87%		91.67%	76.19%	
UART.genblk1.mgmt_uart_inst	88.04%	97.96%	45.83%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_controller_mm_inst	24.21%	32.31%	0.85%	0.00%	63.64%	24.24%	
data_channel.genblk1.genblk1.mgmt_data_channel_controller_inst	31.95%	42.50%	6.15%	0.00%	77.78%	33.33%	
mgmt_phy_top_inst	55.38%	62.00%	28.03%	63.04%	58.94%	64.86%	
dynamic_pll.genblk1.m10_pll_top_inst	46.24%	54.03%	13.04%	63.16%	45.73%	55.24%	
ltpi_phy_rx_inst	67.51%	64.60%	78.30%	57.14%	66.87%	70.67%	
ltpi_phy_tx_inst	64.82%	65.92%	71.90%	62.50%	53.57%	70.20%	
mgmt_ltpi_frm_rx_inst	71.60%	86.86%	33.50%		80.00%	86.06%	
mgmt_ltpi_frm_tx_inst	73.41%	93.09%	23.73%		91.55%	85.26%	
mgmt_phy_controller.mgmt_phy_controller_inst	70.11%	92.39%	35.29%	68.97%	73.53%	80.39%	
pll_system_controller	36.68%	62.52%	4.68%		38.02%	41.48%	
u_avmm_CSR	80.58%	100.00%	41.74%			100.00%	
u_avmm_FPGA_inf	67.10%	100.00%	1.30%			100.00%	
u_avmm_bmc	67.10%	100.00%	1.30%			100.00%	
u_avmm_cntrl	67.10%	100.00%	1.30%			100.00%	
u_avmm_mm	77.54%	100.00%	32.61%			100.00%	
u_avmm_s	67.10%	100.00%	1.30%			100.00%	
dut_ltpi_target	52.51%	68.24%	28.84%	44.16%	56.35%	64.94%	
ltpi_csr_avmm_inst	38.21%	84.13%	22.72%	0.00%	31.13%	53.09%	
mgmt_ltpi_top_inst	55.87%	67.47%	31.84%	44.88%	65.45%	69.71%	
GPIO.genblk1.mgmt_gpio_inst	86.71%	97.67%	56.30%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	65.61%	81.28%	44.78%	42.86%	80.75%	78.40%	
SMBus[0].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[0].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[1].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[2].genblk1.genblk1.smbus_relay_target_0	66.63%	81.57%	53.32%	43.81%	75.38%	79.06%	
SMBus[2].genblk1.smbus_echo_inst	82.01%	80.30%	79.87%		91.67%	76.19%	
SMBus[3].genblk1.genblk1.smbus_relay_target_0	66.63%	81.57%	53.32%	43.81%	75.38%	79.06%	
SMBus[4].genblk1.genblk1.smbus_relay_target_0	66.63%	81.57%	53.32%	43.81%	75.38%	79.06%	
SMBus[4].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[5].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
UART.genblk1.mgmt_uart_inst	88.78%	97.96%	49.54%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_target_mm_inst	15.69%	27.66%	1.90%	0.00%	26.67%	22.22%	
data_channel.genblk1.genblk1.mgmt_data_channel_target_inst	27.70%	33.33%	8.50%	0.00%	66.67%	30.00%	
mgmt_phy_top_inst	55.61%	61.92%	28.13%	64.13%	59.23%	64.65%	
pll_system_target	36.68%	62.52%	4.68%		38.02%	41.48%	
u_avmm_CSR	67.10%	100.00%	1.30%			100.00%	
u_avmm_FPGA_inf	67.10%	100.00%	1.30%			100.00%	
u_avmm_cntrl	67.10%	100.00%	1.30%			100.00%	
u_avmm_mm	67.10%	100.00%	1.30%			100.00%	
u_avmm_s	67.10%	100.00%	1.30%			100.00%	
u_avmm_trg	67.10%	100.00%	1.30%			100.00%	
i2c_slave_wrapper_ctl_inst	57.13%	70.59%	38.71%	37.84%	64.00%	74.51%	
uvm_custom_install_verdi_recording	7.81%	4.67%	0.00%			18.75%	

Figure 42: Regression result 50 cycles - Python framework.

Appendix B: Repositories

Regression Result 100 cycle per test



Name	Score	Line	Toggle	FSM	Condition	Branch	Assert
tb_top	54.81%	68.67%	34.00%	44.54%	60.81%	66.05%	
dut0	54.81%	68.66%	34.00%	44.54%	60.81%	66.05%	
avmm_mux_1to2_inst	58.51%	77.14%	43.00%		41.67%	72.22%	
dut_ltpi_controller	55.82%	68.84%	32.77%	45.85%	65.15%	66.49%	
ltpi_csr_avmm_inst	67.43%	96.83%	31.38%	58.33%	80.19%	70.45%	
mgmt_ltpi_top_inst	56.30%	66.87%	34.26%	45.57%	65.39%	69.42%	
GPIO.genblk1.mgmt_gpio_inst	89.80%	97.67%	68.65%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	65.88%	80.50%	47.46%	43.66%	79.27%	78.50%	
SMBus[0].genblk1.genblk1.smbus_relay_controller_0	67.91%	80.73%	62.25%	43.66%	73.53%	79.35%	
SMBus[0].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_controller_0	67.91%	80.73%	62.25%	43.66%	73.53%	79.35%	
SMBus[1].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
SMBus[2].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[2].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
SMBus[3].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[3].genblk1.smbus_echo_inst	82.20%	81.06%	79.87%		91.67%	76.19%	
SMBus[4].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[4].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
SMBus[5].genblk1.genblk1.smbus_relay_controller_0	67.81%	80.73%	61.75%	43.66%	73.53%	79.35%	
SMBus[5].genblk1.smbus_echo_inst	81.42%	79.55%	79.87%		91.67%	74.60%	
UART.genblk1.mgmt_uart_inst	91.38%	97.96%	62.50%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_controller_mm_inst	20.54%	32.31%	0.68%	0.00%	45.45%	24.24%	
data_channel.genblk1.genblk1.mgmt_data_channel_controller_inst	27.53%	42.50%	6.26%	0.00%	55.56%	33.33%	
mgmt_phy_top_inst	55.49%	62.04%	29.08%	61.96%	59.35%	65.02%	
dynamic_pll.genblk1.m10_pll_top_inst	46.24%	54.03%	13.04%	63.16%	45.73%	55.24%	
ltpi_phy_rx_inst	67.59%	64.60%	78.30%	57.14%	67.07%	70.86%	
ltpi_phy_tx_inst	64.82%	65.92%	71.90%	62.50%	53.57%	70.20%	
mgmt_ltpi_frm_rx_inst	75.33%	88.29%	36.79%		87.14%	89.09%	
mgmt_ltpi_frm_tx_inst	73.31%	93.09%	23.33%		91.55%	85.26%	
mgmt_phy_controller.mgmt_phy_controller_inst	68.33%	90.22%	35.93%	65.52%	73.53%	76.47%	
pll_system_controller	36.68%	62.52%	4.68%		38.02%	41.48%	
u_avmm_CSR	80.58%	100.00%	41.74%			100.00%	
u_avmm_FPGA_inf	67.10%	100.00%	1.30%			100.00%	
u_avmm_bmc	67.10%	100.00%	1.30%			100.00%	
u_avmm_cntrl	67.10%	100.00%	1.30%			100.00%	
u_avmm_mm	77.54%	100.00%	32.61%			100.00%	
u_avmm_s	67.10%	100.00%	1.30%			100.00%	
dut_ltpi_target	53.15%	68.26%	31.85%	44.43%	56.22%	64.98%	
ltpi_csr_avmm_inst	38.95%	84.13%	26.42%	0.00%	31.13%	53.09%	
mgmt_ltpi_top_inst	56.42%	67.50%	34.46%	45.15%	65.25%	69.76%	
GPIO.genblk1.mgmt_gpio_inst	89.78%	97.67%	68.59%		100.00%	92.86%	
SMBUS.genblk1.mgmt_smbus_inst	65.66%	81.34%	44.87%	42.86%	80.75%	78.47%	
SMBus[0].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[0].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[1].genblk1.genblk1.smbus_relay_target_0	66.63%	81.57%	53.32%	43.81%	75.38%	79.06%	
SMBus[1].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[2].genblk1.genblk1.smbus_relay_target_0	66.63%	81.57%	53.32%	43.81%	75.38%	79.06%	
SMBus[2].genblk1.smbus_echo_inst	82.01%	80.30%	79.87%		91.67%	76.19%	
SMBus[3].genblk1.genblk1.smbus_relay_target_0	66.63%	81.57%	53.32%	43.81%	75.38%	79.06%	
SMBus[3].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
SMBus[4].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[4].genblk1.smbus_echo_inst	82.39%	81.82%	79.87%		91.67%	76.19%	
SMBus[5].genblk1.genblk1.smbus_relay_target_0	66.25%	81.57%	53.32%	41.90%	75.38%	79.06%	
SMBus[5].genblk1.smbus_echo_inst	81.61%	80.30%	79.87%		91.67%	74.60%	
UART.genblk1.mgmt_uart_inst	89.16%	97.96%	51.39%	100.00%	100.00%	96.43%	
data_channel.genblk1.genblk1.genblk1.ltpi_data_channel_target_mm_inst	15.69%	27.66%	1.90%	0.00%	26.67%	22.22%	
data_channel.genblk1.genblk1.mgmt_data_channel_target_inst	27.78%	33.33%	8.89%	0.00%	66.67%	30.00%	
mgmt_phy_top_inst	56.28%	61.93%	29.51%	66.30%	58.95%	64.69%	
dynamic_pll.genblk1.m10_pll_top_inst	46.24%	54.03%	13.04%	63.16%	45.73%	55.24%	
ltpi_phy_rx_inst	68.36%	64.96%	78.44%	60.71%	66.26%	71.43%	
ltpi_phy_tx_inst	64.85%	65.92%	72.05%	62.50%	53.57%	70.20%	
mgmt_ltpi_frm_rx_inst	73.20%	87.43%	37.68%		82.86%	84.85%	
mgmt_ltpi_frm_tx_inst	70.26%	85.71%	25.57%		91.55%	78.21%	
mgmt_phy.target.mgmt_phy_target_inst	73.72%	94.68%	33.71%	75.86%	75.68%	88.68%	
pll_system_target	36.68%	62.52%	4.68%		38.02%	41.48%	
u_avmm_CSR	67.10%	100.00%	1.30%			100.00%	
u_avmm_FPGA_inf	67.10%	100.00%	1.30%			100.00%	
u_avmm_cntrl	67.10%	100.00%	1.30%			100.00%	
u_avmm_mm	67.10%	100.00%	1.30%			100.00%	
u_avmm_s	67.10%	100.00%	1.30%			100.00%	
u_avmm_trg	67.10%	100.00%	1.30%			100.00%	
i2c_slave_wrapper_ctl_inst	58.00%	70.77%	38.71%	38.74%	67.00%	74.79%	

Figure 43: Regression result 100 cycles - Python framework.

Appendix B: Repositories

References

[1] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, "Challenges and Trends in Modern SoC Design Verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, Oct. 2017, doi: 10.1109/MDAT.2017.2735383.

[2] N. Susithra, S. Kanna S, S. Karthik K, N. Raja T, V. Yaswant, and K. Rajalakshmi, "Analyzing AES Verification: A Comparative Study of UVM and Cocotb Approaches," in *2024 International Conference on Smart Systems for Electrical, Electronics, Communication and Computer Engineering (ICSSEECC)*, Jun. 2024, pp. 478–482. doi: 10.1109/ICSSEECC61126.2024.10649439.

[3] P. McLellan, "UVM Is Now IEEE 1800.2 and There's a Ten-Year Story to That - Breakfast Bytes - Cadence Blogs - Cadence Community." Accessed: Feb. 21, 2026. [Online]. Available: https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/stan-dards

[4] B. Rosser, "Cocotb: a Python-based digital logic verification framework".

[5] Researcher, "UVM METHODOLOGY: INDUSTRY-SPECIFIC APPLICATIONS IN MODERN HARDWARE VERIFICATION," Nov. 2024, doi: 10.5281/ZENODO.14040112.

[6] H. Yang, L. Wang, C. Xia, and B. Gao, "Cocotb-Based Verification of Multi-Protocol Interconnection Network," in *2024 4th International Conference on Electronic Information Engineering and Computer (EIECT)*, Nov. 2024, pp. 791–794. doi: 10.1109/EIECT64462.2024.10866829.

[7] "cocotb," cocotb. Accessed: Feb. 21, 2026. [Online]. Available: https://www.cocotb.org/

[8] "Introduction — pyuvm v4.0.1 documentation." Accessed: Feb. 26, 2026. [Online]. Available: https://pyuvm.readthedocs.io/en/latest/docsources/README.html#

[9] C. M. University, "Ronald Rohrer - Electrical and Computer Engineering - College of Engineering - Carnegie Mellon University." Accessed: Feb. 19, 2026. [Online]. Available: http://www.ece.cmu.edu/directory/bios/rohrer-ronald.html

[10] P. McLellan, "A Brief History of Functional Verification," Semiwiki. Accessed: Feb. 19, 2026. [Online]. Available: https://semiwiki.com/eda/3348-a-brief-history-of-functional-verification/

[11] D. I. Rich, "The evolution of systemverilog," *IEEE Design & Test of Computers*, vol. 20, no. 4, pp. 82–84, Jul. 2003, doi: 10.1109/MDT.2003.1214355.

[12] "Functional Verification – lifting the veil - Tremend." Accessed: Feb. 19, 2026. [Online]. Available: https://tremend.com/blog/advanced-technologies/functional-verification-lifting-the-veil/

[13] "What is Functional Verification? – How it Works | Synopsys." Accessed: Feb. 19, 2026. [Online]. Available: https://www.synopsys.com/glossary/what-is-functional-verification.html

[14] "UVM Verification." Accessed: Feb. 19, 2026. [Online]. Available: https://www.cadence.com/en_US/home/explore/uvm-verification.html

[15] Admin, "UVM Tutorial," ChipVerify. Accessed: Feb. 19, 2026. [Online]. Available: https://www.chipverify.com/tutorials/uvm?form=MG0AV3

[16] Ş. Güney and I. Cicek, "Verification of RISC-V R-type Instructions Using A Custom Cocotb Based Approach," in *2025 21st International Conference on Synthesis, Modeling, Analysis and Simulation Methods, and Applications to Circuits Design (SMACD)*, Jul. 2025, pp. 1–4. doi: 10.1109/SMACD65553.2025.11092160.

[17] "DUT – Definition and Role in Compliance Testing," IB-Lenhardt AG. Accessed: May 15, 2026. [Online]. Available: https://ib-lenhardt.com/kb/glossary/dut

References

[18] "A Research on Object Oriented Programming and Its Concepts," *IJATCSE*, vol. 10, no. 2, pp. 746–749, Apr. 2021, doi: 10.30534/ijatcse/2021/401022021.

[19] "Functional verification," in *Electronic Design Automation*, Morgan Kaufmann, 2009, pp. 513–573. doi: 10.1016/B978-0-12-374364-0.50016-3.

[20] N. Piterman, *Hardware and Software: Verification and Testing: 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*. Springer, 2015.

[21] A. B. Mehta, *ASIC/SoC Functional Design Verification*. Cham: Springer International Publishing, 2018. doi: 10.1007/978-3-319-59418-7.

[22] "Typical UVM Testbench Architecture | The Art Of Verification." Accessed: Feb. 26, 2026. [Online]. Available: https://theartofverification.com/uvm-testbench-architecture/

[23] vlsiverify, "UVM Phases," VLSI Verify. Accessed: Feb. 26, 2026. [Online]. Available: https://vlsiverify.com/uvm/uvm-phases/

[24] "IEEE Standard for Universal Verification Methodology Language Reference Manual," *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, pp. 1–458, Sep. 2020, doi: 10.1109/IEEESTD.2020.9195920.

[25] "Open Compute Project," Open Compute Project. Accessed: Feb. 27, 2026. [Online]. Available: https://www.opencompute.org

[26] "OCP_DC-SCM_Rev2.2_Ver1.0.pdf," Google Docs. Accessed: Mar. 21, 2026. [Online]. Available: https://drive.google.com/file/d/1LJoHoI2L-UYKFeGAMR_BWSy12K665CUP/view?usp=drive_link&usp=embed_facebook

[27] "Server/MHS/DC-SCM-Specs-and-Designs - OpenCompute." Accessed: Mar. 21, 2026. [Online]. Available: https://www.opencompute.org/w/index.php?title=Server/MHS/DC-SCM-Specs-and-Designs

[28] "OCP_DC-SCM_2.2_LTPI_Ver1.0.pdf," Google Docs. Accessed: Mar. 21, 2026. [Online]. Available: https://drive.google.com/file/d/1GsIK7bo1Wz2ZvvAJcq5bGRpol_zdEP64/view?usp=drive_link&usp=embed_facebook

[29] "MarcoRamirezZ/ltpi-verifenv-python," GitHub. Accessed: May 11, 2026. [Online]. Available: https://github.com/MarcoRamirezZ/ltpi-verifenv-python

[30] "MarcoRamirezZ/ltpi-verifenv-sv," GitHub. Accessed: May 11, 2026. [Online]. Available: https://github.com/MarcoRamirezZ/ltpi-verifenv-sv

[31] *opencomputeproject/HWMgmt-Module-DCSCM-LTPI*. (Apr. 19, 2026). SystemVerilog. Open Compute Project. Accessed: May 11, 2026. [Online]. Available: https://github.com/opencomputeproject/HWMgmt-Module-DCSCM-LTPI

[32] "Introduction — pyvsc 0.0.1 documentation." Accessed: Apr. 09, 2026. [Online]. Available: https://pyvsc.readthedocs.io/en/latest/introduction.html

[33] "GlobalInterpreterLock." Accessed: May 15, 2026. [Online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock