Verilua: An Open Source Versatile Framework for Efficient Hardware Verification and Analysis Using LuaJIT

Ye Cai*, Chuyu Zheng*, Wei He†, Dan Tang†‡
†Beijing Institute of Open Source Chip, Beijing, China
‡Institute of Computing Technology, Chinese Academy of Sciences (lCT), Beijing, China
*College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
caiye@szu.edu.cn, zhengchuyu2023@email.szu.edu.cn, hewei@bosc.ac.cn, tangdan@ict.ac.cn

***Abstract*—The growing complexity of hardware verification highlights limitations in existing frameworks, particularly regarding flexibility and reusability. Current methodologies often require multiple specialized environments for functional verification, waveform analysis, and simulation, leading to toolchain fragmentation and inefficient code reuse. This paper presents Verilua, a unified framework leveraging LuaJIT and the Verilog Procedural Interface (VPI), which integrates three core functionalities: Lua-based functional verification, a scripting engine for RTL simulation, and waveform analysis. By enabling complete code reuse through a unified Lua codebase, the framework achieves a 12× speedup in RTL simulation compared to cocotb and a 70× improvement in waveform analysis over state-of-the-art solutions. Through consolidating verification tasks into a single platform, Verilua enhances efficiency while reducing tool fragmentation and learning overhead, addressing critical challenges in modern hardware design.**

***Index Terms*—hardware verification, verification framework, waveform analysis, LuaJIT**

I. Introduction

The growing complexity of integrated circuits under Moore's Law has intensified challenges in RTL verification, which consumes significant project time. While SystemVerilog and UVM are industry standards, their complexity raises entry barriers, necessitating more agile tools akin to Chisel's role in Verilog design [1]. Verification, fundamentally a software task, benefits from efficient languages or frameworks, yet current hardware verification languages (HVLs) like C++ in Verilator and Python in cocotb [2] face issues of performance, learning curves, and poor component reusability across simulation environments. While frameworks like PyMTL [4] and Fault [5] attempt to address these challenges by unifying design and verification or enabling cross-HVL compatibility, they still face limitations in handling complex verification needs (e.g., reusing verification components across different environments) or integrating simulation and waveform analysis.

We propose Verilua, a multifunctional framework based on LuaJIT and VPI, integrating HVL, hardware script engine (HSE), and waveform analysis language (WAL). Verilua enables using Lua as HVL, embeds Lua scripts in RTL simulations as an HSE, and performs waveform analysis via a VPI-compatible layer. It promotes reusability with a unified Lua-based environment, where verification components can be reused across all three scenarios, addressing the fragmentation of the verification ecosystem. Verilua achieves up to 12× speedup over cocotb in RTL simulation and up to 70× speedup over the original Lisp implementation in WAL scenarios, significantly improving verification efficiency while lowering the learning curve.

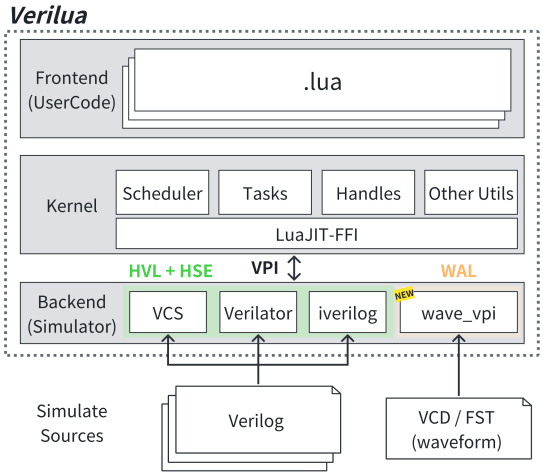

Fig. 1. Framework of Verilua

II. Framework Design

The Verilua framework adopts a front-end and back-end separation approach to decouple Lua code from specific simulators, enabling efficient verification code reuse as shown in Fig 1. The back-end includes RTL simulators such as VCS, Verilator, and Icarus Verilog, with *wave_vpi* as a custom back-end for waveform simulation. Lua interacts with simulators through the VPI (Verilog Procedural Interface), a standardized interface supported by most RTL simulators. This ensures Verilua's strong reusability, unifying verification and analysis scenarios such as HVL, HSE, and WAL.

Verilua implements lightweight concurrency using tasks, which are Lua coroutines. Tasks run in a quasi-parallel manner

978-3-9826741-0-0/DATE25/© 2025 EDAA

but are executed serially, allowing suspension and resumption during execution. This design enables resource sharing without multi-threading, reducing context-switching overhead and aligning with hardware verification requirements. For instance, a task can wait for signal changes (e.g., posedge or negedge) while suspending free CPU resources for other tasks.

The Kernel layer, situated between the front-end and back-end, includes a Scheduler for managing tasks and Handles for facilitating signal-simulator interactions. Handles are essential data structures that ensure efficient communication, making Verilua flexible and well-suited for complex verification tasks.

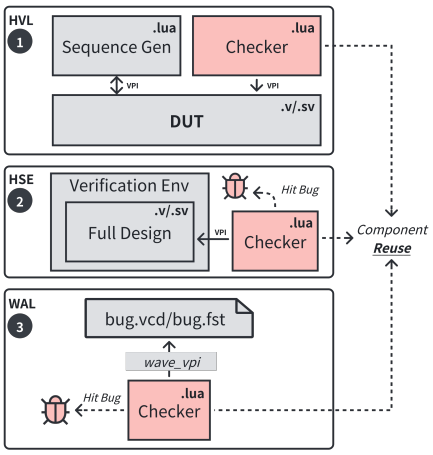

Fig. 2. Component Reuse in Verilua

III. Case Study

We employ three typical scenarios to demonstrate how Verilua addresses the fragmentation of the verification ecosystem and enhances verification efficiency. As illustrated in Fig 2 (1), Verilua can serve as an HVL to construct a complete verification environment by developing test stimulus generators, checkers, and other components using Lua. The Checker component is particularly noteworthy, as it is designed to read signal values from the DUT (Design Under Test) to verify the correctness of simulation behavior. If bugs are present, the Checker can also detect them. The Checker component functions effectively as long as a mechanism for reading hardware signals is provided, which Verilua achieves through the VPI. Furthermore, this Verilua-based Checker component can be utilized in non-Verilua verification environments, such as UVM or Cocotb, since the hardware simulators employed in these environments typically support VPI. As depicted in Fig 2 (2), in the HSE scenario, the same Verilua-based Checker component can be reused across different verification environments via VPI, enabling the detection of hardware bugs in these environments.

When Verilua is used as a WAL for waveform analysis (Fig 2 (3)), the Checker remains applicable, with the simulation target shifting to waveform files. This enables automated waveform analysis, where the Checker identifies potential bugs, significantly improving analysis efficiency for verification engineers.

These scenarios highlight Verilua's ability to enhance verification efficiency. Components developed in Verilua can be seamlessly reused across HVL, HSE, and WAL scenarios using VPI. This reusability allows engineers to accumulate Verilua-based components without redundancy, streamlining the verification process.

IV. Evaluation

We compare the simulation performance with cocotb, which shares similarities with Verilua, as both are based on VPI and coroutine. The performance test case uses the matrix multiplication module from the official cocotb repository. The test is conducted using different commonly used simulators. Additionally, the evaluation is divided into two categories—*has_startup* and *no_startup*—based on the startup times of cocotb and Verilua.

As shown in Table I, Verilua demonstrates superior performance across the three tested simulators, achieving faster simulation speeds and significantly shorter startup times compared to cocotb, enabling rapid iterative testbench development. Under Verilator, Verilua's simulation speed is approximately 12 times faster than cocotb, while it shows a 2-fold improvement under iverilog and vcs. This performance gap is attributed to Verilua's internal optimizations and framework design, which are particularly pronounced under Verilator.

We evaluate Verilua against a Lisp-based waveform simulator (lisp-WAL [3]) using FST-formatted waveform test cases from a processor cache subsystem. The evaluation tracks L2 cache hit/miss information, comparing lisp-WAL (Python 3.12 and PyPy 3.10) with Verilua, testing JIT-enabled and disabled performance. Results show Verilua achieves 70x and 60x speedup over lisp-WAL (Python and PyPy, respectively), due to lisp-WAL's Python-based core, which limits its performance. Enabling JIT in Verilua provides a modest 20% performance boost, limited by test simplicity and Lua code complexity.

TABLE I
Performance Result Between Cocotb & Verilua

type	has_startup			no_startup		
simulator	verilator	iverilog	vcs	verilator	iverilog	vcs
cocotb(s)	7.742	10.209	9.853	1.747	3.007	2.764
verilua(s)	0.501	2.138	3.342	0.147	1.775	1.542

References

[1] J. Bachrach et al., "Chisel: constructing hardware in a scala embedded language," in Proceedings of the 49th Annual Design Automation Conference, 2012, pp. 1216-1225.

[2] B. J. Rosser, "Cocotb: a Python-based digital logic verification framework," in Micro-electronics Section seminar, CERN, Geneva, Switzerland, 2018.

[3] L. Klemmer and D. Große, "WAL: a novel waveform analysis language for advanced design understanding and debugging, in 2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC), 2022, pp. 358–364.

[4] S. Jiang, P. Pan, Y. Ou, and C. Batten, "PyMTL3: a Python framework for open-source hardware modeling, generation, simulation, and verification," IEEE Micro, vol. 40, no. 4, pp. 58–66, 2020.

[5] L. Truong et al., "fault: A Python embedded domain-specific language for metaprogramming portable hardware verification components," in CAV 2020, Los Angeles, CA, USA, 2020, pp. 403–414.